UNIT - I

Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks- Operations, array and linked representations of stacks, stack applications, Queues- operations, array and linked representations.

Data Structure

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned.

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

Processor speed: To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down thesearch process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process
in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

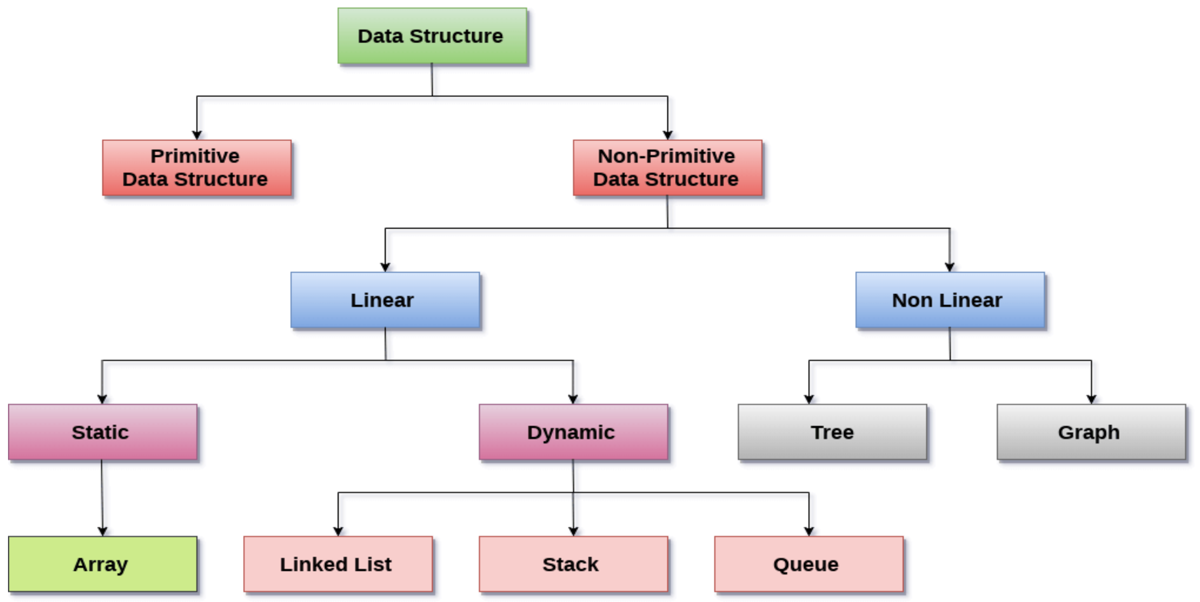
Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element.hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Linear Data Structures

If a data structure organizes the data in sequential order, then that data structure is called a Linear DataStructure.

Example

1. Arrays
2. List (Linked List)
3. Stack
4. Queue

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are: age[0], age[1], age[2], age[3],…age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.
A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.
It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures:
This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Non - Linear Data Structures

If a data structure organizes the data in random order, then that data structure is called as Non-Linear Data Structure.

Example

1. Tree
2. Graph
3. Dictionaries
4. Heaps
5. Tries, Etc.,

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottom most nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.
Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.
If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.
If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

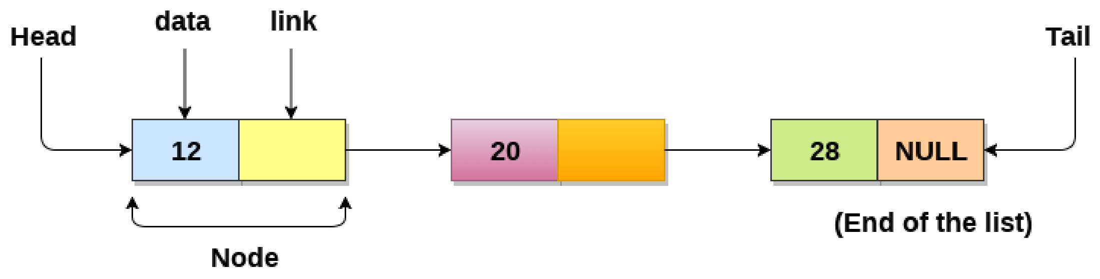
6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

Abstract Data Type:

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of theimplementation, we are hiding them from the user's view. This is called information hiding. The implementation of an abstract data type, often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs andprimitive data types.

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can’t be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.
2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Differences between the array and linked list in a tabular form.

ARRAYS	LINKED LISTS
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address
Array elements store in a contiguous memory location	Linked list elements can be stored anywhere in the memory or randomly stored
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time	In the case of a linked list, memory is allocated at run time
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement

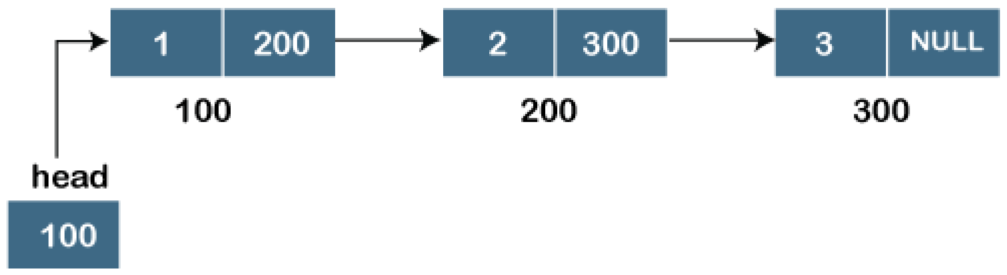
Types of Linked List

The following are the types of linked list:

1. **Singly linked list**
2. **Doubly linked list**
3. **Circular linked list**

Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a pointer.
Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



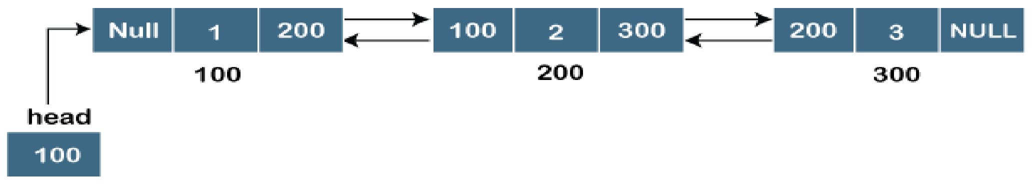
We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a ***head pointer***.
The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.
Representation of the node in a singly linked list

```
struct node
        {
          int data;
          struct node *next;
        }
```

In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one datapart, a pointer to its previous node, and a pointer to the next node.
Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the ***address of the next*** while the other part of the node stores the ***previous node's address***. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

```
struct node
{
  int data;
  struct node *next;
 struct node *prev;
}
```

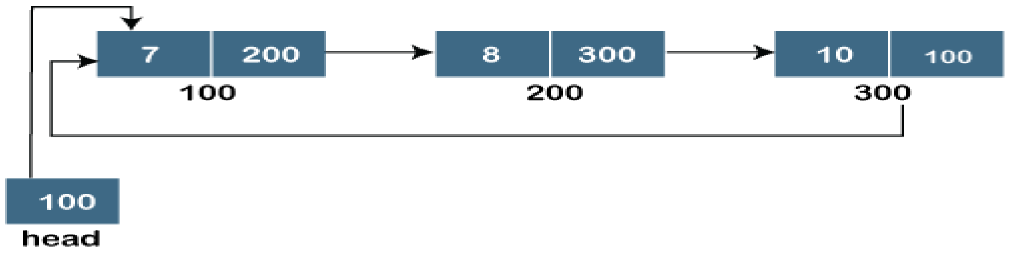
In the above representation, we have defined a user-defined structure named ***a node*** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next and prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next andprev** is **struct node** as both the pointers are storing the address of the node of the ***struct node*** type.

Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the ***singly linked list*** and a ***circular linked*** list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

```
struct node
        {
          int data;
          struct node *next;
        }
```

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:

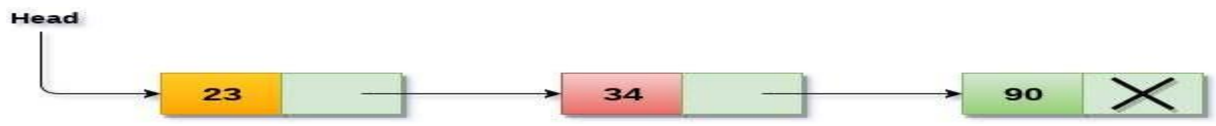


Singly linked list

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by thenull pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

1. Inserting at Beginning
2. Inserting at the End of the LIst
3. Inserting after specified node

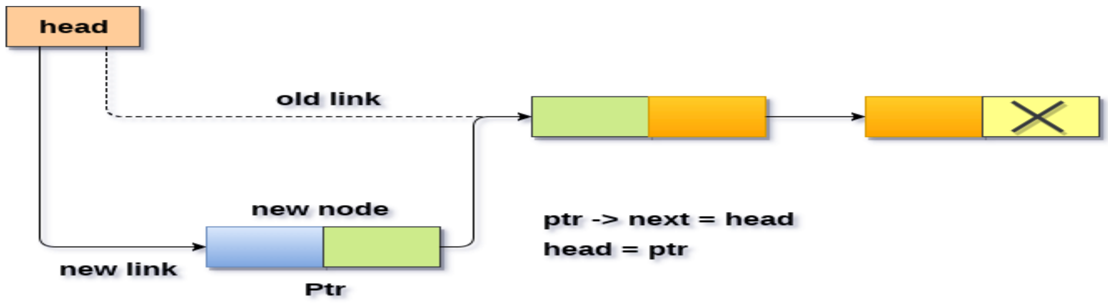
Insertion in singly linked list at beginning

Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need to be followed in order to inser a new node in the list at beginning.

1. Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.
   ```
   ptr = (struct node *) malloc(sizeof(struct node *));
           ptr → data = item
   ```
2. Make the link part of the new node pointing to the existing first node of the list. This will be done by using the following statement.
   ```
           ptr->next = head
   ```

3. At the last, we need to make the new node as the first node of the list this will be done by using the following statement.

head = ptr;



Algorithm

```
○ Step 1: IF PTR = NULL
  Write OVERFLOW
     Go to Step 7
    [END OF IF]
○ Step 2: SET NEW_NODE = PTR
○ Step 3: SET PTR = PTR → NEXT
○ Step 4: SET NEW_NODE → DATA = VAL
○ Step 5: SET NEW_NODE → NEXT = HEAD
○ Step 6: SET HEAD = NEW_NODE
○ Step 7: EXIT
```

Function for inserting element at beginning of the list

```
void beginsert()
{
   struct node *ptr;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node *));
   if(ptr == NULL)
   {
      printf("\n memory insufficient to allocate");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&item);
      ptr->data = item;
      ptr->next = head;     head = ptr;
```

```
        printf("\nNode inserted");
      }
    }
```

2.Inserting at the End of the List

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

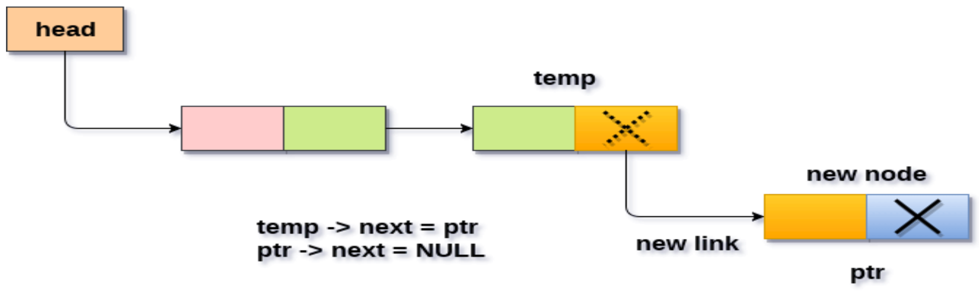
1. The node is being added to an empty list(CASE 1)
2. The node is being added to the end of the linked list(CASE2)

in the first case,(CASE1)

- The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.
  ```
  ptr->data = item;
              ptr -> next = NULL;
  ```
- Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.
  Head = ptr

In the second case: CASE(2):

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.
  Temp = head
- Then, traverse through the entire linked list using the statements:
  ```
  while (temp→ next != NULL)
           temp = temp → next;
  ```
- At the end of the loop, the temp will be pointing to the last node of the list. Now, allocate the space for the new node, and assign the item to its data part. Since, the new node is going to be the last node of the list hence, the next part of this node needs to be pointing to the **null**. We need to make the next part o
- If the temp node (which is currently the last node of the list) point to the new node (ptr) .temp = head;
  ```
      while (temp -> next != NULL)
      {
         temp = temp -> next;
      }
      temp->next = ptr;
      ptr->next = NULL;
  ```



Inserting node at the last into a non-empty list

Algorithm

```
Step 1: IF PTR = NULL Write OVERFLOW
    Go to Step 1
  [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR - > NEXT
Step 4: SET NEW_NODE - > DATA = VAL
Step 5: SET NEW_NODE - > NEXT = NULL
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 while PTR - > NEXT != NULL
Step 8: SET PTR = PTR - > NEXT
[END OF LOOP]
Step 9: SET PTR - > NEXT = NEW_NODE
Step 10: EXIT
```

Function for inserting element at the end of the list

```c
void lastinsert()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
     printf("\nOVERFLOW");
   }
   else
   {
     printf("\nEnter value?\n");          scanf("%d",&item);
      ptr->data = item;
      if(head == NULL)
```

```
        {
          ptr -> next = NULL;
          head = ptr;
          printf("\nNode inserted");
        }
        else
        {
          temp = head;
          while (temp -> next != NULL)
          {
            temp = temp -> next;
          }
          temp->next = ptr;
          ptr->next = NULL;
          printf("\nNode inserted");

        }
      }
    }
```

Insertion in singly linked list after specified Node

- In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. This will be done by using the following statements.

  ```
  emp=head;
              for(i=0;i<loc;i++)
              {
                temp = temp->next;
                if(temp == NULL)
                {
                  return;
                }
              }
  ```

- Allocate the space for the new node and add the item to the data part of it. This will be done by using the following statements.

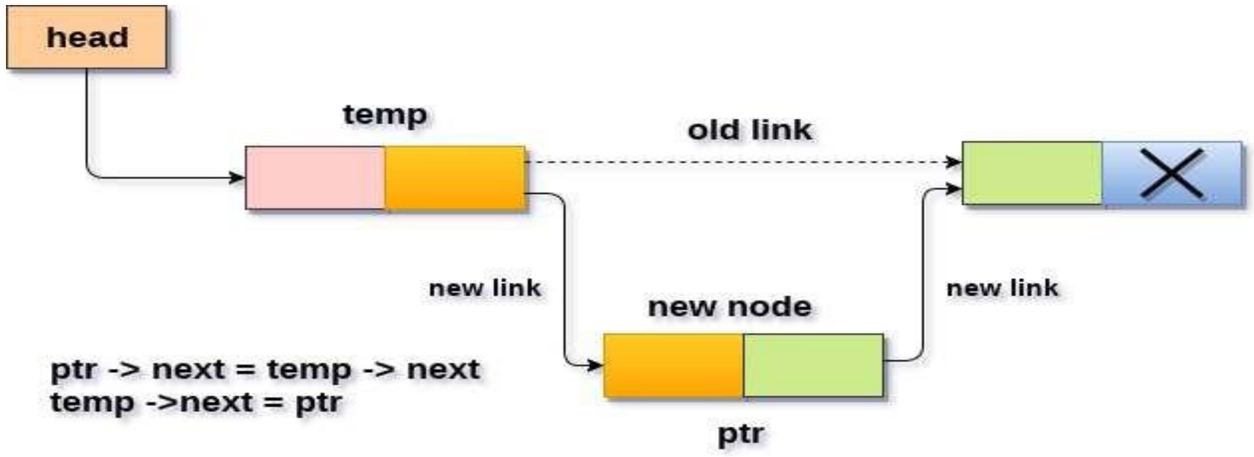
  ```
  ptr = (struct node *) malloc (sizeof(struct node));
  ptr->data = item;
  ```

- Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part ofthe new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp). This will be done by using the following statements.

ptr→ next = temp → next

now, we just need to make the next part of the temp, point to the new node ptr. This will insert the new node ptr, at the specified position.

temp ->next = ptr;



Algorithm

```
o STEP 1: IF PTR = NULL
    WRITE OVERFLOW
        GOTO STEP 12
      END OF IF
o STEP 2: SET NEW_NODE = PTR
o STEP 3: NEW_NODE → DATA = VAL
o STEP 4: SET TEMP = HEAD
o STEP 5: SET I = 0
o STEP 6: REPEAT STEP 5 AND 6 UNTIL I<loc< li=""></loc<>
o STEP 7: TEMP = TEMP → NEXT
o STEP 8: IF TEMP = NULL
    WRITE "DESIRED NODE NOT PRESENT"
        GOTO STEP 12
      END OF IF
    END OF LOOP
o STEP 9: PTR → NEXT = TEMP → NEXT
o STEP 10: TEMP → NEXT = PTR
o STEP 11: SET PTR = NEW_NODE
o STEP 12: EXIT
```

C Function

```
void randominsert()
{
   int i,loc,item;
```

```
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=1;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
```

Deletion

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

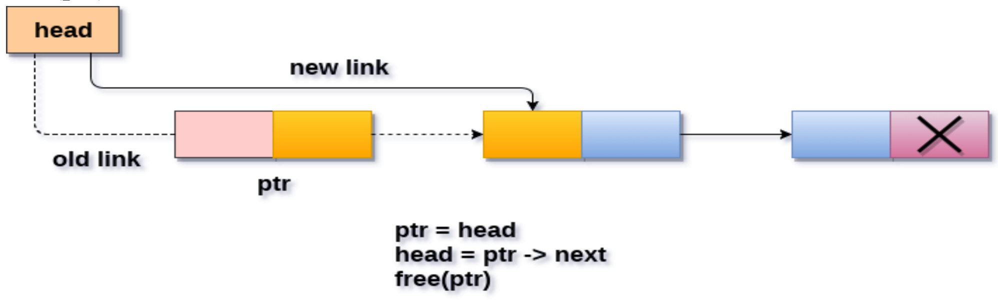
1. Deleting at Beginning
2. Deleting at the End of the List
3. Deleting after specified node

Deletion in singly linked list at beginning

Deleting a node from the beginning of the list is the simplest operation of all. It just need a few adjustments in the node pointers. Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head. This will be done by using the following statements

ptr = head;
head = ptr->next;

Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.
free(ptr)



Deleting a node from the beginning

Algorithm

```
○ Step 1: IF HEAD = NULL
          Write UNDERFLOW
             Go to Step 5
            [END OF IF]
○ Step 2: SET PTR = HEAD
○ Step 3: SET HEAD = HEAD -> NEXT
○ Step 4: FREE PTR
○ Step 5: EXIT
```

C function

```
void begdelete()
   {
      struct node *ptr;
      if(head == NULL)
      {
         printf("\nList is empty");
      }
      else
      {
         ptr = head;
         head = ptr->next;
         free(ptr);
         printf("\n Node deleted from the begining ...");}}
```

Deletion in singly linked list at the end

Here are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

In the first scenario,

the condition head → next = NULL will survive and therefore, the only node head of the list will be assigned to null. This will be done by using the following statements.

```
ptr = head
head = NULL
free(ptr)
```

In the second scenario,

The condition head → next = NULL would fail and therefore, we have to traverse the node in order to reach the last node of the list.

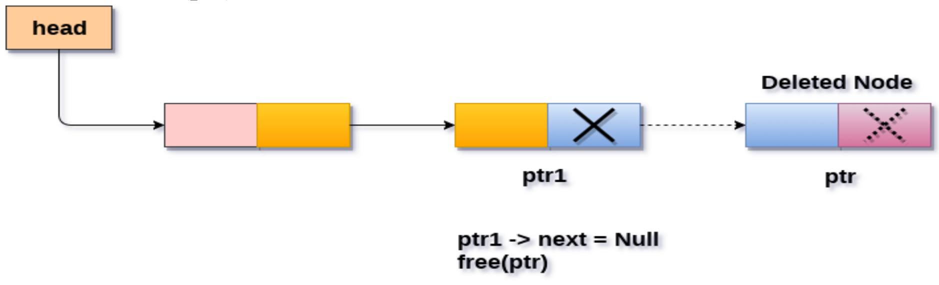
For this purpose, just declare a temporary pointer temp and assign it to head of the list. We also need to keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

this all will be done by using the following statements.

```
ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
```

Now, we just need to make the pointer ptr1 point to the NULL and the last node of thelist that is pointed by ptr will become free. It will be done by using the following statements.

```
      ptr1->next = NULL;
        free(ptr);
```



Deleting a node from the last

Algorithm

- **Step 1:** IF HEAD = NULL

```
Write UNDERFLOW
  Go to Step 8
 [END OF IF]
```

- **Step 2:** SET PTR = HEAD
- **Step 3:** Repeat Steps 4 and 5 while PTR -> NEXT!= NULL
- **Step 4:** SET PREPTR = PTR
- **Step 5:** SET PTR = PTR -> NEXT
  [END OF LOOP]
- **Step 6:** SET PREPTR -> NEXT = NULL
- **Step 7:** FREE PTR
- **Step 8:** EXIT

C Function

```c
void end_delete()
   {
        struct node *ptr,*ptr1;
      if(head == NULL)
       {
          printf("\nlist is empty");
       }
       else if(head -> next == NULL)
       {
          head = NULL;
          free(head);
          printf("\nOnly node of the list deleted ...");
       }
       else
       {
          ptr = head;
          while(ptr->next != NULL)
              {
                 ptr1 = ptr; ptr = ptr ->next;
              }
              ptr1->next = NULL;
              free(ptr);
              printf("\n Deleted Node from the last ...");

          }
     }
```

```
        }
```

Deletion in singly linked list after the specified node

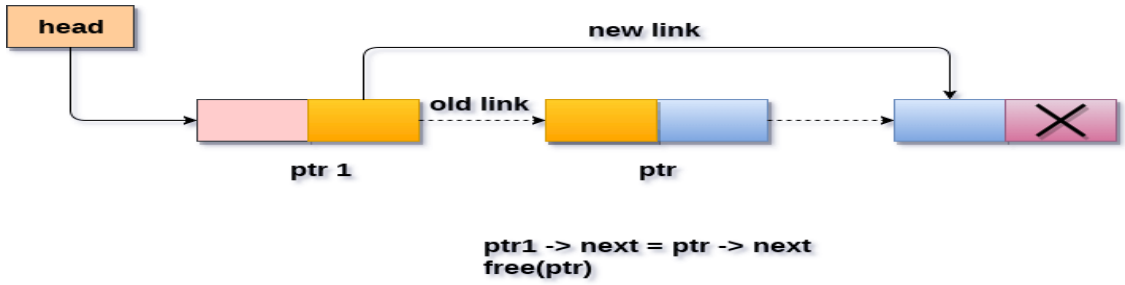
In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1.

Use the following statements to do so.

```
ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nThere are less than %d elements in the list..",loc);
            return;
       }
    }
```

Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

This will be done by using the following statements.



Deletion a node from specified position

Algorithm

```
o   STEP 1: IF HEAD = NULL

    WRITE UNDERFLOW
       GOTO STEP 10
      END OF IF
o   STEP 2: SET TEMP = HEAD
o   STEP 3: SET I = 0
```

- **STEP 4:** REPEAT STEP 5 TO 8 UNTIL I<loc< li=""></loc<>
- **STEP 5:** TEMP1 = TEMP
- **STEP 6:** TEMP = TEMP → NEXT
- **STEP 7:** IF TEMP = NULL

  ```
      WRITE "DESIRED NODE NOT PRESENT"
      GOTO STEP 12
        END OF IF
  ```
- **STEP 8:** I = I+1

  ```
      END OF LOOP
  ```
- **STEP 9:** TEMP1 → NEXT = TEMP → NEXT
- **STEP 10:** FREE TEMP
- **STEP 11:** EXIT

Searching in singly linked list

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Algorithm

- **Step 1:** SET PTR = HEAD
- **Step 2:** Set I = 0
- **STEP 3:** IF PTR = NULL

  ```
  WRITE "EMPTY LIST"
  GOTO STEP 8
  END OF IF
  ```
- **STEP 4:** REPEAT STEP 5 TO 7 UNTIL PTR != NULL
- **STEP 5:** if ptr → data = item

  ```
  write i+1
  End of IF
  ```
- **STEP 6:** I = I + 1
- **STEP 7:** PTR = PTR → NEXT

  ```
  [END OF LOOP]
  ```
- **STEP 8:** EXIT

C Function

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }
```

Traversing in singly linked list

Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operationon that. This will be done by using the following statements.

```
ptr = head;
    while (ptr!=NULL)
      {
```

```
    ptr = ptr -> next;
}
```

Algorithm

```
○ STEP 1: SET PTR = HEAD
○ STEP 2: IF PTR = NULL
    WRITE "EMPTY LIST"
    GOTO STEP 7
    END OF IF
○ STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
○ STEP 5: PRINT PTR→ DATA
○ STEP 6: PTR = PTR → NEXT
  [END OF LOOP]
○ STEP 7: EXIT
```

SINGLY LINKED LIST ADVANTAGE

1) Insertions and Deletions can be done easily.
2) It does not need movement of elements for insertion and deletion.
3) It space is not wasted as we can get space according to our requirements.
4) Its size is not fixed.
5) It can be extended or reduced according to requirements.
6) Elements may or may not be stored in consecutive memory available
7) It is less expensive.

DISADVANTAGE

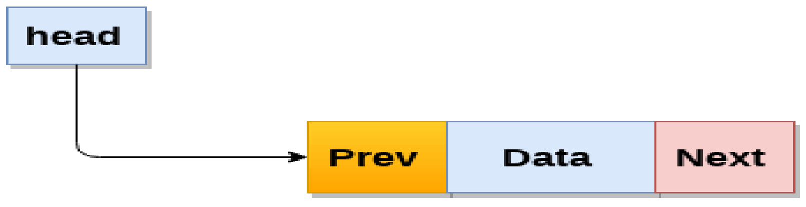
1) It requires more space as pointers are also stored with information.
2) Different amount of time is required to access each element.
3) If we have to go to a particular element then we have to go through all those elements that come before that element.
4) we can not traverse it from last & only from the beginning.
5) It is not easy to sort the elements stored in the linear linked list.

Applications of Linked Lists

Graphs, queues, and stacks can be implemented by using Linked List.

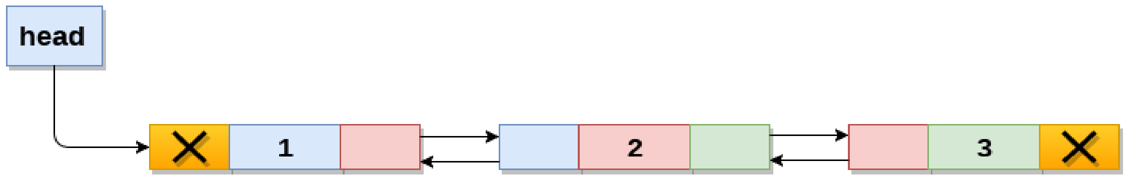
DOUBLE LINKED LIST

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in thefigure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

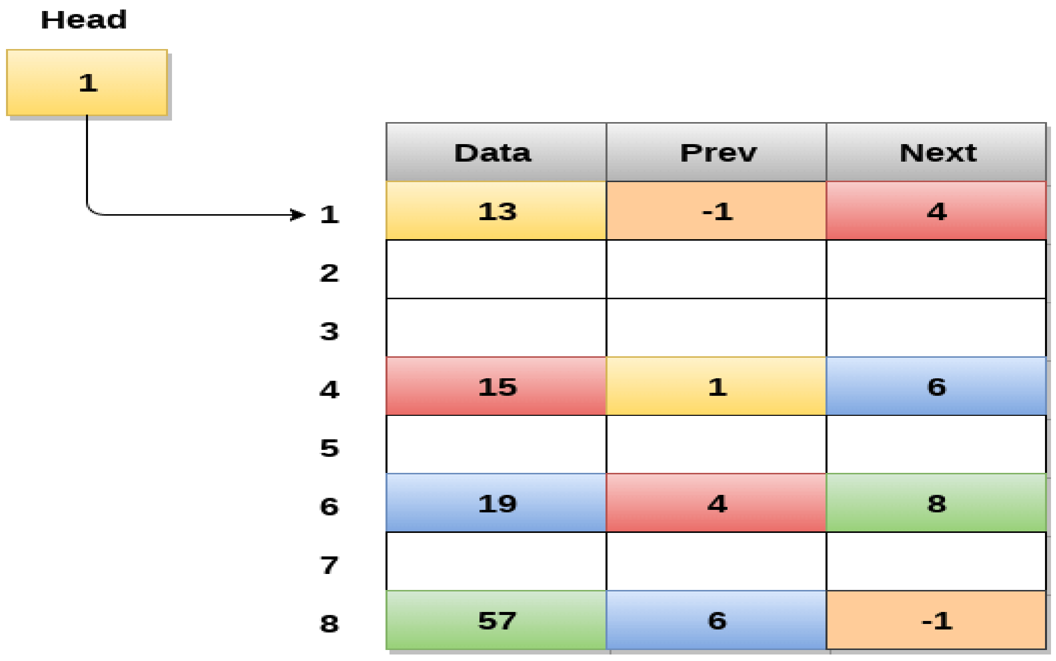
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Operations on doubly linked list

The following operations are performed on double linked list

1) Insertion
   - Insertion at beginning

- Insertion at End
- Insertion at specified position

1) Deletion
    - Deletion from the Beginning
    - Deletion from the End
    - Deletion of the node having specified data
2) Searching
3) Traversing

Node Creation

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
};
struct node *head;
```

INSERTION

Insertion in doubly linked list at beginning

As in doubly linked list, each node of the list contain double pointers therefore we have to maintain more number of pointers in doubly linked list as compare to singly linked list.

There are two scenarios of inserting any element into doubly linked list. Either the list is empty or it contains at least one element. Perform the following steps to insert a node in doubly linked list at beginning.

- Allocate the space for the new node in the memory. This will be done by using the following statement.

```
ptr = (struct node *)malloc(sizeof(struct node));
```

- Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

```
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
head=ptr;
```

- In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

- This will be done by using the following statements.

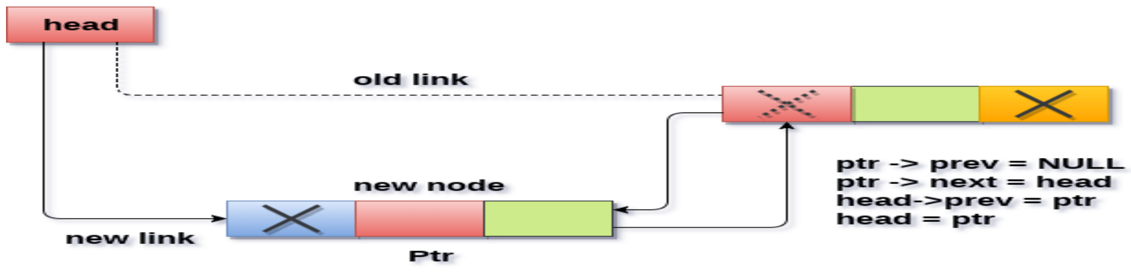
  ptr->next = head;
  head→prev=ptr;

  Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.

  ptr→prev =NULL
  head = ptr

Algorithm :

```
o Step 1: IF ptr = NULL
          Write OVERFLOW
          Go to Step 9
          [END OF IF]
o Step 2: SET NEW_NODE = ptr
o Step 3: SET ptr = ptr -> NEXT
o Step 4: SET NEW_NODE -> DATA = VAL
o Step 5: SET NEW_NODE -> PREV = NULL
o Step 6: SET NEW_NODE -> NEXT = START
o Step 7: SET head -> PREV = NEW_NODE
o Step 8: SET head = NEW_NODE
o Step 9: EXIT
```



Insertion into doubly linked list at beginning

C Function

```
void insertbeginning( )
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  int item;
  printf("enter the value");
```

```
    scanf("%d",&item);
    if(ptr == NULL)
    {
       printf("\nOVERFLOW");
    }
    else
    {
          if(head==NULL)
          {
                  ptr->next = NULL;
                  ptr->prev=NULL;
                  ptr->data=item;
                  head=ptr;
          }
          else
          {
                  ptr->data=item;
                  ptr->prev=NULL;
                  ptr->next = head;
                  head->prev=ptr;
                  head=ptr;
           }
  }
```

Insertion in doubly linked list at the end

In order to insert a node in doubly linked list at the end, we must make sure whether the list isempty or it contains any element. Use the following steps in order to insert the node in doublylinked list at the end.

- Allocate the memory for the new node. Make the pointer **ptr** point to the new node being inserted.

  ```
  ptr = (struct node *) malloc(sizeof(struct node));
  ```

- Check whether the list is empty or not. The list is empty if the condition **head == NULL** holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

  ```
  ptr->next = NULL;
   ptr->prev=NULL;
   ptr->data=item;
   head=ptr;
  ```

- In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer **temp** to head and traverse the list by using this pointer.

```
Temp = head;
while (temp != NULL)
{
        temp = temp → next;
 }
```

the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.

```
temp→next =ptr;
```

make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.

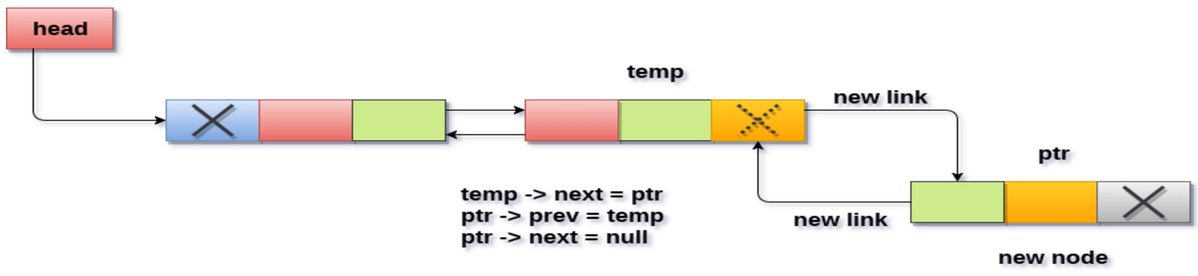
```
ptr → prev = temp;
```

make the next pointer of the node ptr point to the null as it will be the new last node of the list.

```
ptr → next = NULL
```

Algorithm

```
○ Step 1: IF PTR = NULL
        Write OVERFLOW
         Go to Step 11
         [END OF IF]
○ Step 2: SET NEW_NODE = PTR
○ Step 3: SET PTR = PTR -> NEXT
○ Step 4: SET NEW_NODE -> DATA = VAL
○ Step 5: SET NEW_NODE -> NEXT = NULL
○ Step 6: SET TEMP = START
○ Step 7: Repeat Step 8 while TEMP -> NEXT != NULL
○ Step 8: SET TEMP = TEMP -> NEXT
        [END OF LOOP]
○ Step 9: SET TEMP -> NEXT = NEW_NODE
○ Step 10C: SET NEW_NODE -> PREV = TEMP
○ Step 11: EXIT
```



Insertion into doubly linked list at the end

C Program

```
void insertlast()
{
  struct node *ptr = (struct node *) malloc(sizeof(struct node));
  int item;
  printf("enter the value");
  scanf("%d",&item);
  struct node *temp;
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
      ptr->data=item;
     if(head == NULL)
     {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
     }
     else{
       temp = head;
       while(temp->next!=NULL)
       {
          temp = temp->next;
       }
       temp->next = ptr;
```

```
            ptr ->prev=temp;
            ptr->next = NULL;
          }
printf("\nNode Inserted\n");
    }
}
```

Insertion in doubly linked list after Specified node

In order to insert a node after the specified node in the list, we need to skip the required number of nodes in order to reach the mentioned node and then make the pointer adjustments as required.Use the following steps for this purpose.

- Allocate the memory for the new node. Use the following statements for this.

  ```
  ptr = (struct node *)malloc(sizeof(struct node));
  ```

- Traverse the list by using the pointer **temp** to skip the required number of nodes in order to reach the specified node.

  ```
          temp=head;
     for(i=0;i<loc;i++)
     {
        temp = temp->next;
        if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentio
            ned location
        {
             return;
        }
     }
  ```

- The temp would point to the specified node at the end of the **for** loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of **ptr** point to the next node of temp.

  ```
  ptr → next = temp → next;
  ```

  make the **prev** of the new node ptr point to temp.

  ```
  ptr → prev = temp;
  ```

  make the **next** pointer of temp point to the new node ptr.

  ```
  temp → next = ptr;
  ```

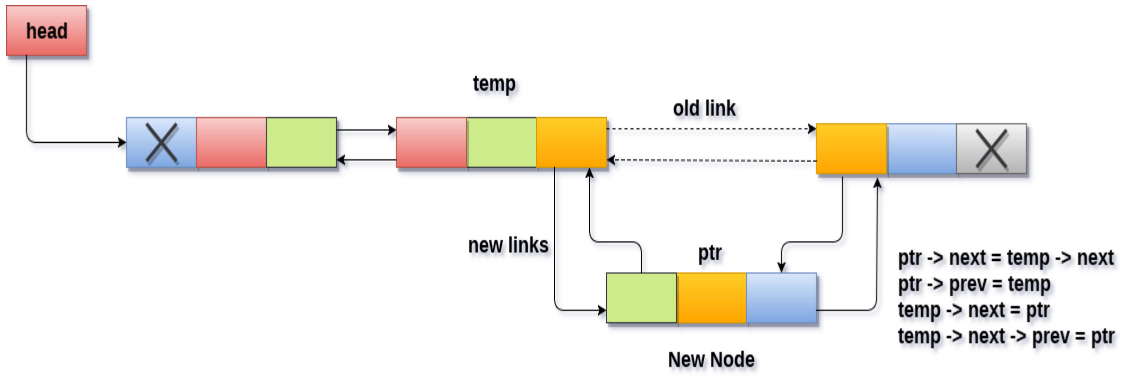
  make the **previous** pointer of the next node of temp point to the new node.

  ```
  temp → next → prev = ptr;
  ```

Algorithm

```
o Step 1: IF PTR = NULL
      Write OVERFLOW
      Go to Step 15
      [END OF IF]
```

```
o Step 2: SET NEW_NODE = PTR
o Step 3: SET PTR = PTR -> NEXT
o Step 4: SET NEW_NODE -> DATA = VAL
o Step 5: SET TEMP = START
o Step 6: SET I = 0
o Step 7: REPEAT 8 to 10 until I<="" li="">
o Step 8: SET TEMP = TEMP -> NEXT
o STEP 9: IF TEMP = NULL
o STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
      GOTO STEP 15
      [END OF IF]
      [END OF LOOP]
o Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
o Step 12: SET NEW_NODE -> PREV = TEMP
o Step 13 : SET TEMP -> NEXT = NEW_NODE
o Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE
o Step 15: EXIT
```



Insertion into doubly linked list after specified node

C Function

```
void insert_specified(int item)
{
  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  struct node *temp;
  int i, loc;
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
```

```
    else
    {
        printf("\nEnter the location\n");
        scanf("%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr->data = item;
        ptr->next = temp->next;
        ptr -> prev = temp;
        temp->next = ptr;
        temp->next->prev=ptr;
        printf("Node Inserted\n");
    }
}
```

DELETION OPERATION

Deletion at beginning

Deletion in doubly linked list at the beginning is the simplest operation. We just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

```
Ptr = head;
head = head → next;
```

now make the prev of this new head node point to NULL. This will be done by using the following statements.

```
head → prev = NULL
```

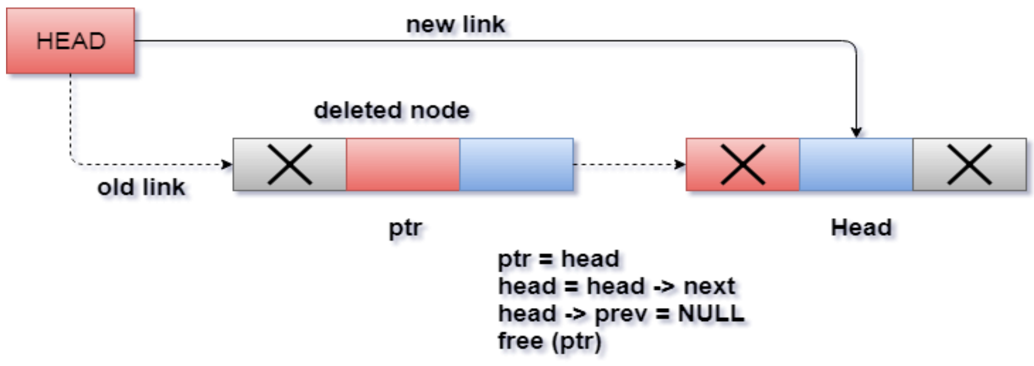
Now free the pointer ptr by using the **free** function.

```
free(ptr)
```

Algorithm

```
○ STEP 1: IF HEAD = NULL
        WRITE UNDERFLOW
        GOTO STEP 6
```

- **STEP 2:** SET PTR = HEAD
- **STEP 3:** SET HEAD = HEAD → NEXT
- **STEP 4:** SET HEAD → PREV = NULL
- **STEP 5:** FREE PTR
- **STEP 6:** EXIT



Deletion in doubly linked list from beginning

C FUNCTION

```
void beginning_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW\n");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```

Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.

```
ptr = head;
if(ptr->next != NULL)
{
        ptr = ptr -> next;
}
```

- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

```
ptr → prev → next = NULL
```

free the pointer as this the node which is to be deleted.

```
free(ptr)
```

ALGORITHM

- **Step 1:** IF HEAD = NULL

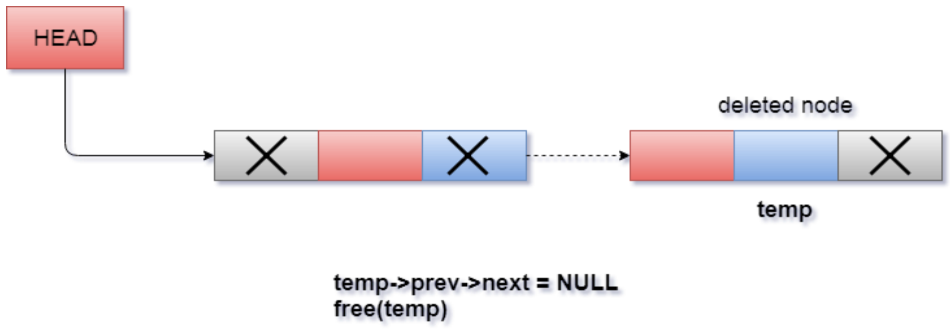
  Write UNDERFLOW
  Go to Step 7
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT

  [END OF LOOP]

- **Step 5:** SET TEMP ->PREV-> NEXT = NULL

- **Step 6:** FREE TEMP
- **Step 7:** EXIT



Deletion in doubly linked list at the end

C PROGRAM

```
void last_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW\n");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nNode Deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nNode Deleted\n");
    }
}
```

Deletion in doubly linked list after the specified node

In order to delete the node after the specified data, we need to perform the following steps.

- Copy the head pointer into a temporary pointer temp.

```
temp = head
```

- Traverse the list until we find the desired data value.

```
while(temp -> data != val)
temp = temp -> next;
```

- Check if this is the last node of the list. If it is so then we can't perform deletion.

```
if(temp -> next == NULL)
{
        return;
 }
```

- Check if the node which is to be deleted, is the last node of the list, if it so then we have to make the next pointer of this node point to null so that it can be the new last node of the list.

```
if(temp -> next -> next == NULL)
{
        temp ->next = NULL;
 }
```

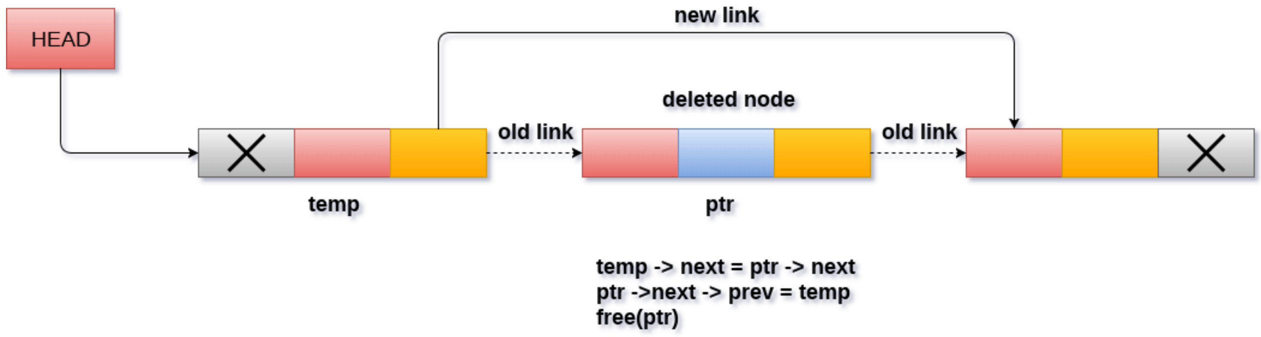
- Otherwise, make the pointer ptr point to the node which is to be deleted. Make the next of temp point to the next of ptr. Make the previous of next node of ptr point to temp. free the ptr.

```
ptr = temp -> next;
temp -> next = ptr -> next;
ptr -> next -> prev = temp;
free(ptr);
```

Algorithm

```
o Step 1: IF HEAD = NULL
  Write UNDERFLOW
  Go to Step 9
 [END OF IF]
o Step 2: SET TEMP = HEAD
o Step 3: Repeat Step 4 while TEMP -> DATA != ITEM
o Step 4: SET TEMP = TEMP -> NEXT
  [END OF LOOP]
o Step 5: SET PTR = TEMP -> NEXT
o Step 6: SET TEMP -> NEXT = PTR -> NEXT
o Step 7: SET PTR -> NEXT -> PREV = TEMP
```

- **Step 8:** FREE PTR
- **Step 9:** EXIT



Deletion of a specified node in doubly linked list

C FUNCTION

```
void delete_specified( )
{
   struct node *ptr, *temp;
   int val;
   printf("Enter the value");
   scanf("%d",&val);
   temp = head;
   while(temp -> data != val)
   temp = temp -> next;
   if(temp -> next == NULL)
   {
      printf("\nCan't delete\n");
   }
   else if(temp -> next -> next == NULL)
   {
      temp ->next = NULL;
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = temp -> next;
      temp -> next = ptr -> next;
      ptr -> next -> prev = temp;
      free(ptr);
      printf("\nNode Deleted\n");
```

```
        }
    }
```

Searching for a specific node in Doubly Linked List

We just need traverse the list in order to search for a specific element in the list. Perform following operations in order to search a specific operation.

- Copy head pointer into a temporary pointer variable ptr.

  ptr = head
- declare a local variable I and assign it to 0.

  i=0
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.

Algorithm

```
o Step 1: IF HEAD == NULL
   WRITE "UNDERFLOW"
   GOTO STEP 8
   [END OF IF]
o Step 2: Set PTR = HEAD
o Step 3: Set i = 0
o Step 4: Repeat step 5 to 7 while PTR != NULL
o Step 5: IF PTR → data = item
  return  i
  [END OF IF]
o Step 6: i = i + 1
o Step 7: PTR = PTR → next
o Step 8: Exit
```

C FUNCTION

```
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }
```

Traversing in doubly linked list

Traversing is the most common operation in case of each data structure. For this purpose, copy the head pointer in any of the temporary pointer ptr.

```
            Ptr = head
```

then, traverse through the list by using while loop. Keep shifting value of pointer variable **ptr** until we find the last node. The last node contains **null** in its next part.

```
      while(ptr != NULL)
      {
            printf("%d\n",ptr->data);
            ptr=ptr->next;
      }
```

Although, traversing means visiting each node of the list once to perform some specific operation. Here, we are printing the data associated with each node of the list.

Algorithm

```
o Step 1: IF HEAD == NULL
            WRITE "UNDERFLOW"
          GOTO STEP 6
          [END OF IF]
o Step 2: Set PTR = HEAD
o Step 3: Repeat step 4 and 5 while PTR != NULL
o Step 4: Write PTR → data
o Step 5: PTR = PTR → next
o Step 6: Exit
```

C Function

```
int traverse()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        ptr = head;
        while(ptr != NULL)
        {
            printf("%d\n",ptr->data);
            ptr=ptr->next;
        }
    }
 }
```

Differences between Singly linked list and Doubly linked list

Singly linked list (SLL)	**Doubly linked list (DLL)**
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.

Singly linked list (SLL)	**Doubly linked list (DLL)**
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.

3. Write a program that uses functions to perform the following operations on circular linked list:

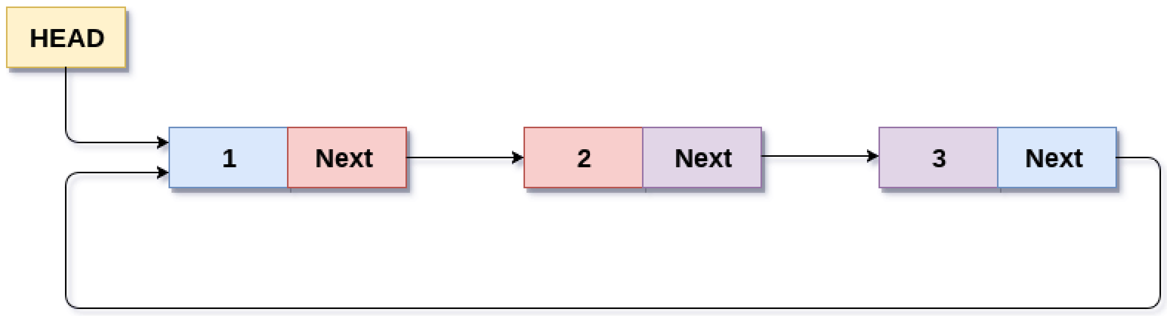
i) Creation ii) Insertion iii) Deletion iv) Traversal

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

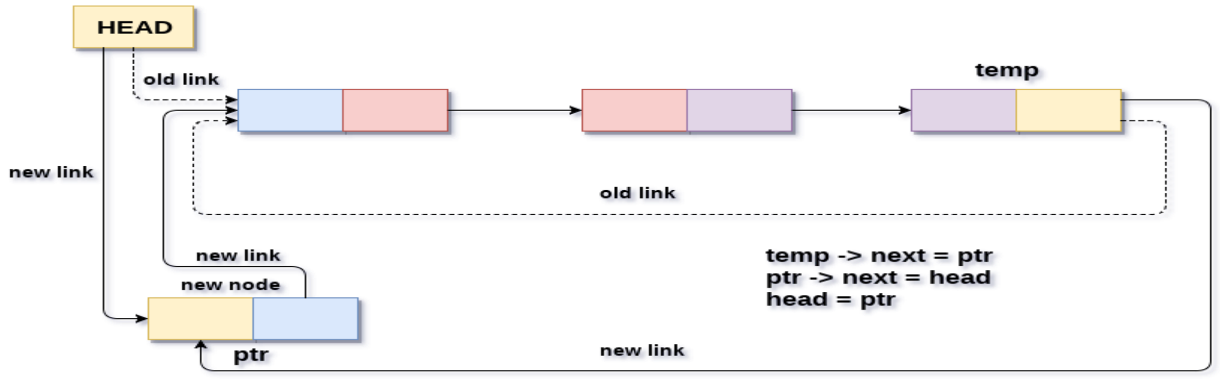
Write a program that uses functions to perform the following operations on circular linked list: i) Creation ii) Insertion iii) Deletion iv) Traversal

i)Creation

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
struct node
{
    int data;
    struct node *next;
};
struct node *head;
void main ()
{
    int choice,item;
    do
    {
        printf("1.Append List\n2.Exit\n3.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
            case 2:
            exit(0);
            break;
            default:
            printf("\nPlease enter valid choice\n");
        }

    }while(choice != 3);
}
void create(int item)
{

    struct node *ptr = (struct node *)malloc(sizeof(struct node));
    struct node *temp;
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
    printf("\nNode Inserted\n");
    }
}
```

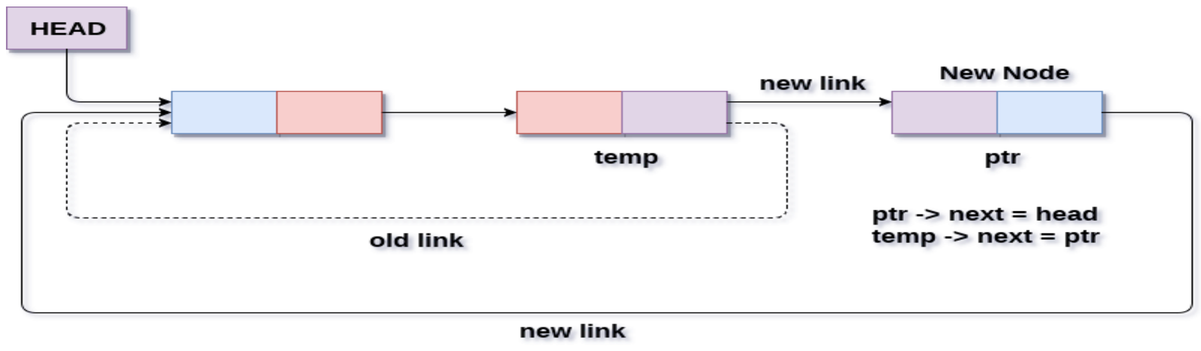
ii)Insertion

Insertion into circular singly linked list at beginning



Insertion into circular singly linked list at beginning

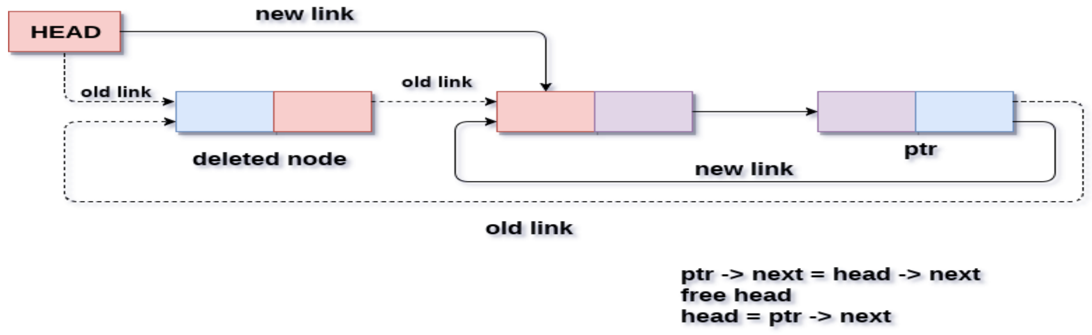
Insertion into circular singly linked list at the end



Insertion into circular singly linked list at end

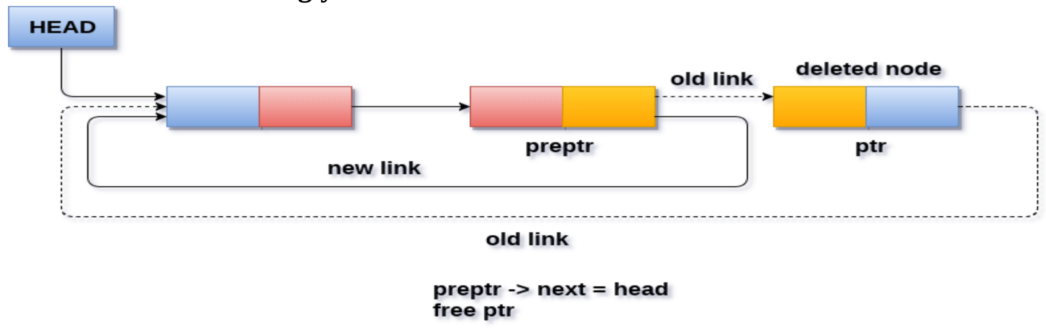
```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void beginsert ();
void lastinsert ();
void display();
void main ()
{
   int choice =0;
   while(choice != 4)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.display\n4.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         beginsert();
         break;
         case 2:
         lastinsert();
         break;
         case 3:
         display();
         case 4:
            exit(0);
            break;
            default:
            printf("Please enter valid choice.."); }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }
}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
        canf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }

        printf("\nnode inserted\n");
    }

}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }
    }
```

Deletion in circular singly linked list at beginning



Deletion in circular singly linked list at beginning

Deletion in Circular singly linked list at the end



Deletion in circular singly linked list at end

iii) Deletion

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void create();
void begin_delete();
void last_delete();
void display();
void main ()
{
   int choice =0;
   while(choice != 5)
```

```
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.create\n2.Delete from Beginning\n3.Delete from last\n4.Show\n5.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {   case 1:
            create();
            break;
            case 2:
            begin_delete();
            break;
            case 3:
            last_delete();
            break;
            case 4:
            display();
            break;
            case 5:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}

void create()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
           temp = head;
           while(temp -> next != head)
           {
              temp = temp -> next;
           }
           temp -> next = ptr;
           ptr -> next = head;
        }
        printf("\nnode inserted\n");
     }
  }
  void begin_delete()
  {
     struct node *ptr;
     if(head == NULL)
     {
        printf("\nUNDERFLOW");
     }
     else if(head->next == head)
     {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
     }

     else
     {  ptr = head;
        while(ptr -> next != head)
           ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

     }
  }
  void last_delete()
  {
     struct node *ptr, *preptr;
     if(head==NULL)
     {
        printf("\nUNDERFLOW");
     }
     else if (head ->next == head)
     {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");

    }
    else
    {
        ptr = head;
        while(ptr ->next != head)
        {
            preptr=ptr;
            ptr = ptr->next;
        }
        preptr->next = ptr -> next;
        free(ptr);
        printf("\nnode deleted\n");

    }
}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}
```

iv) Traversal

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void traverse();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         traverse();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      ptr -> data = item;
```

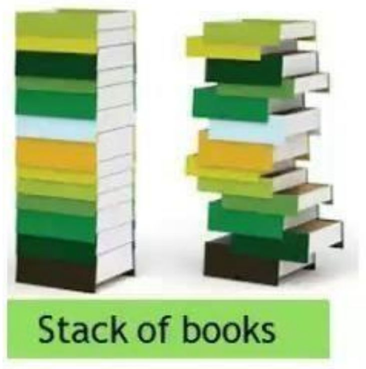
```
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)temp =
                temp->next;
            ptr->next = head; temp ->
            next = ptr;head = ptr;
        }
    printf("\nNode Inserted\n");
    }

}
void traverse()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);ptr =
            ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}
```

Stack

A Stack is linear data structure. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the **top of the stack**. Stack principle is **LIFO (last in, first out)**. Which element inserted last on to the stack that element deleted first from the stack.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

Real life examples of stacks are:



Operations on stack:

The two basic operations associated with stacks are:

1. Push
2. Pop

While performing push and pop operations the following test must be conducted on the stack.

a) Stack is empty or not b) stack is full or not

1. **Push:** Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

2. **Pop:** Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack (or) Implementation of stack:
The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:
Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a ***stack overflow*** condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a ***stack underflow*** condition.

1. **push():**When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

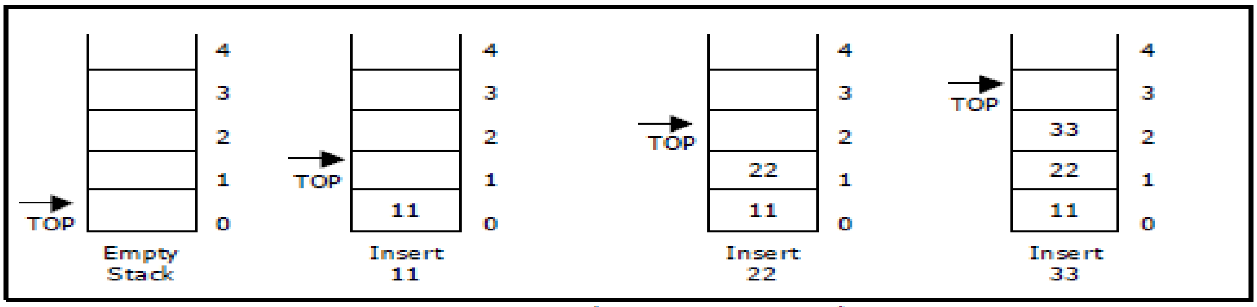


Figure . Push operations on stack

Initially **top=-1**, we can insert an element in to the stack, increment the top value i.e **top=top+1**. We can insert an element in to the stack first check the condition is stack is full or not. i.e **top>=size-1**. Otherwise add the element in to the stack.

Algorithm: Procedure for push():

```
Step 1: START
Step 2: if top>=size-1 then
            Write " Stack is Overflow"
Step 3: Otherwise
        3.1: read data value 'x'
        3.2: top=top+1;
        3.3: stack[top]=x;
Step 4: END
```

2. **Pop():** When an element is taken off from the stack, the operation is performed by pop(). Below

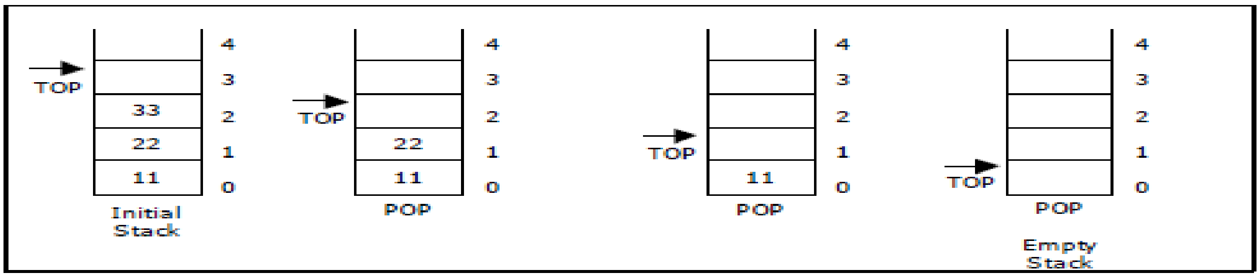


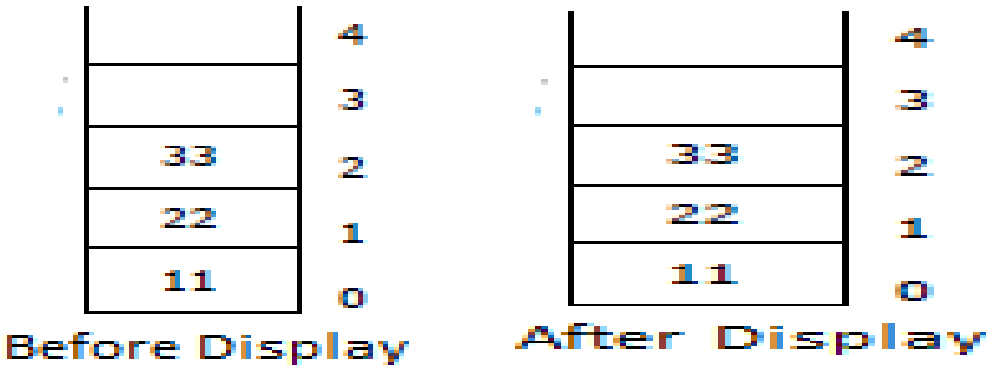
Figure Pop operations on stack

figure shows a stack initially with three elements and shows the deletion of elements using pop().
We can insert an element from the stack, decrement the top value i.e **top=top-1**.
We can delete an element from the stack first check the condition is stack is empty or not.
i.e **top==-1**. Otherwise remove the element from the stack.

Algorithm: procedure pop():

```
Step 1: START
Step 2: if top==-1 then
            Write "Stack is Underflow"
Step 3: otherwise
           3.1: print "deleted element"
            3.2: top=top-1;
Step 4: END
```

3. **display():** This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.



Algorithm: procedure pop():

```
Step 1: START
Step 2: if top==-1 then
            Write "Stack is Underflow"
Step 3: otherwise
            3.1: print "Display elements are"
            3.2: for top to 0
            Print 'stack[i]'
Step 4: END
```

Stack Implementation Using Arrays

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void display();
void main ()
{
   printf("Enter the number of elements in the stack ");
   scanf("%d",&n);
   printf("*********Stack operations using array\n*********");
   while(choice != 4)
   {
      printf("Chose one from the below options...\n");
      printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
      printf("\n Enter your choice \n");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("Exiting… ");
            break;
         }
         default:
         {
            printf("Please Enter valid choice ");
         } } };
}
```

```
  void push ()
{
    int val;
    if (top == n )
    printf("\n Stack Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}
  void pop () {
    if(top == -1)
    printf("Stack Underflow");
    else
    top = top -1;
}
void display()
{
   if(top == -1)
    {
        printf("Stack is empty");
    }
    printf("stack elements are\n ")
    for (i=top;i>=0;i--)
    {
        printf(" %d ",stack[i]);
    }
}
```

OUTPUT:

```
Enter the number of elements in the stack 5
*********Stack operations using array*********
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
5
```

```
Please Enter valid choice Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
stack elements are
 12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
1
Enter the value?12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
3
stack elements are
 12
12
Chose one from the below options...
1.Push
2.Pop
3.Show
4.Exit
 Enter your choice
4
Exiting....
```

Applications of STACK:

Application of Stack :

- Recursive Function.
- Expression Evaluation.
- Expression Conversion.
    - Infix to postfix
    - Infix to prefix
    - Postfix to infix
    - Postfix to prefix
    - Prefix to infix
    - Prefix to postfix
- Reverse a Data
- Processing Function Calls

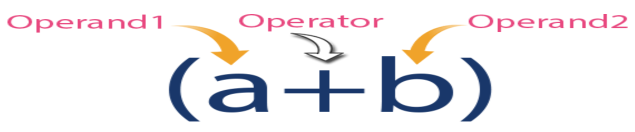
Expressions:

- An expression is a collection of operators and operands that represents a specific value.
- Operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
- Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location

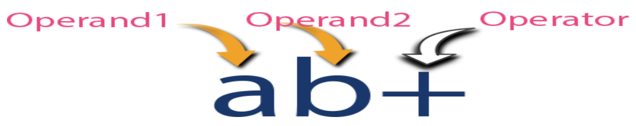
Expression types:

Based on the operator position, expressions are divided into THREE types. They are as follows.

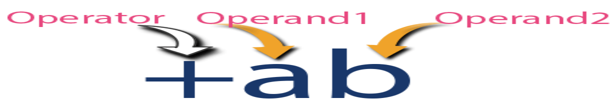
- **Infix Expression**
    - In infix expression, operator is used in between operands.
    - Syntax : operand1 operator operand2
    - Example



- **Postfix Expression**
    - In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".
    - Syntax : operand1 operand2 operator
    - Example:



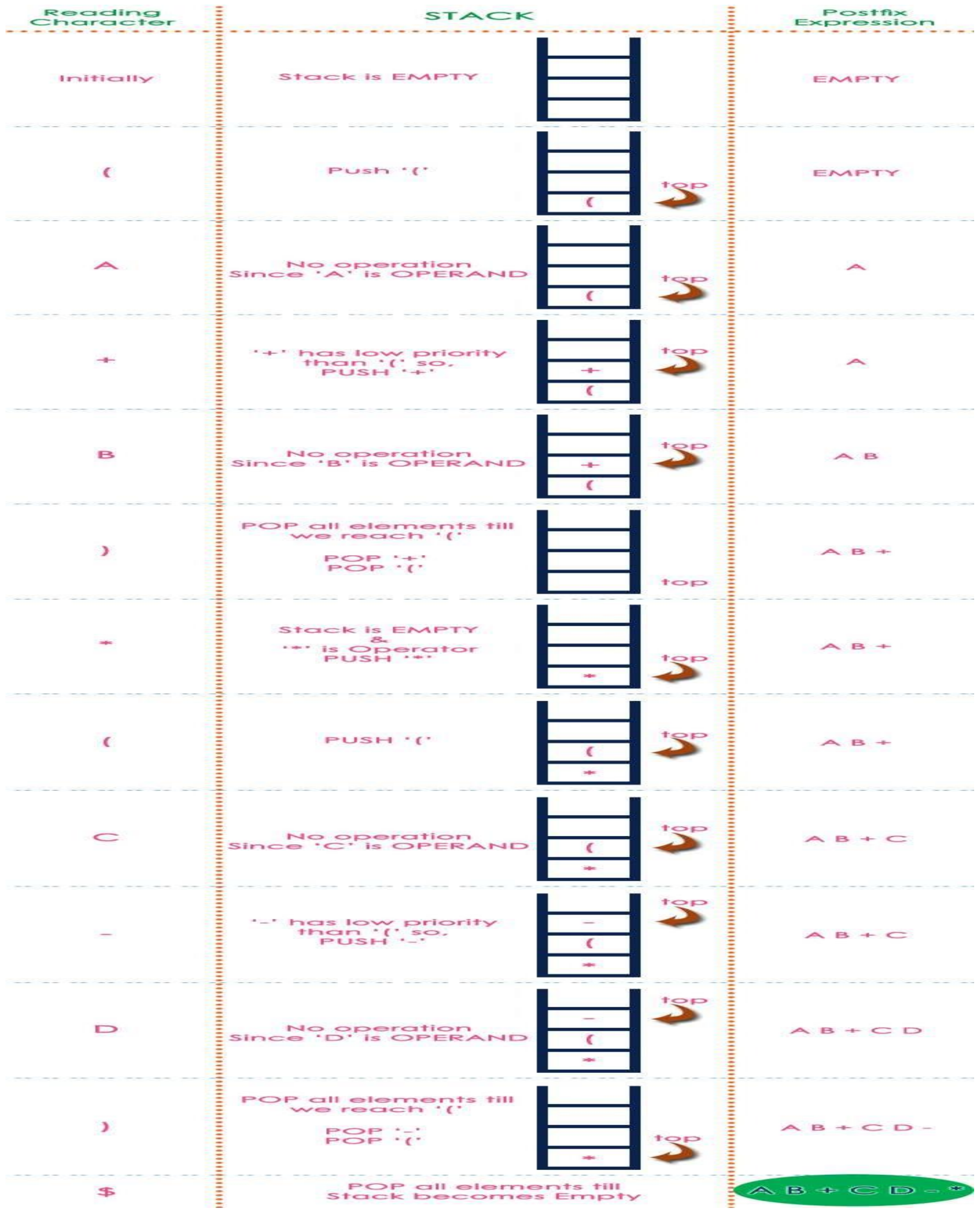
- **Prefix Expression**
    - In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".
    - Syntax : operator operand1 operand2
    - Example:



<u>Infix to postfix conversion using stack:</u>

- Procedure to convert from infix expression to postfix expression is as follows:
- Scan the infix expression from left to right.
- If the scanned symbol is left parenthesis, push it onto the stack.
- If the scanned symbol is an operand, then place directly in the postfix expression (output).
- If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
- If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedenceof the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example-1



Reading Character	STACK		Postfix Expression
Initially	Stack is EMPTY		EMPTY
(	Push '('	(	EMPTY
A	No operation Since 'A' is OPERAND	(	A
+	'+' has low priority than '(' so, PUSH '+'	+ (	A
B	No operation Since 'B' is OPERAND	+ (	A B
)	POP all elements till we reach '(' POP '+' POP '('		A B +
*	Stack is EMPTY & '*' is Operator PUSH '*'	*	A B +
(	PUSH '('	(*	A B +
C	No operation Since 'C' is OPERAND	(*	A B + C
–	'–' has low priority than '(' so, PUSH '–'	– (*	A B + C
D	No operation Since 'D' is OPERAND	– (*	A B + C D
)	POP all elements till we reach '(' POP '–' POP '('	*	A B + C D –
$	POP all elements till Stack becomes Empty		A B + C D – *

Example2:

Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of String	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example3

Convert a + b * c + (d * e + f) * g the infix expression into postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	a		
+	a	+	
B	a b	+	
*	a b	+ *	
C	a b c	+ *	
+	a b c * +	+	

(	a b c * +	+ (	
D	a b c * + d	+ (	
*	a b c * + d	+ (*	
E	a b c * + d e	+ (*	
+	a b c * + d e *	+ (+	
F	a b c * + d e * f	+ (+	
)	a b c * + d e * f +	+	
*	a b c * + d e * f +	+ *	
G	a b c * + d e * f + g	+ *	
End of String	a b c * + d e * f + g * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 3:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of String	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 4:

Convert the following infix expression A+(B *C–(D/E↑F)*G)*H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
(	A	+ (	
B	A B	+ (	
*	A B	+ (*	
C	A B C	+ (*	
-	A B C *	+ (-	
(	A B C *	+ (- (	
D	A B C * D	+ (- (	
/	A B C * D	+ (- (/	
E	A B C * D E	+ (- (/	
↑	A B C * D E	+ (- (/ ↑	
F	A B C * D E F	+ (- (/ ↑	
)	A B C * D E F ↑ /	+ (-	
*	A B C * D E F ↑ /	+ (- *	
G	A B C * D E F ↑ / G	+ (- *	
)	A B C * D E F ↑ / G * -	+	
*	A B C * D E F ↑ / G * -	+ *	
H	A B C * D E F ↑ / G * - H	+ *	
End of string	A B C * D E F ↑ / G * - H * +	The input is now empty. Pop the output symbols from the stack until it is empty.	

Evaluation of postfix expression:

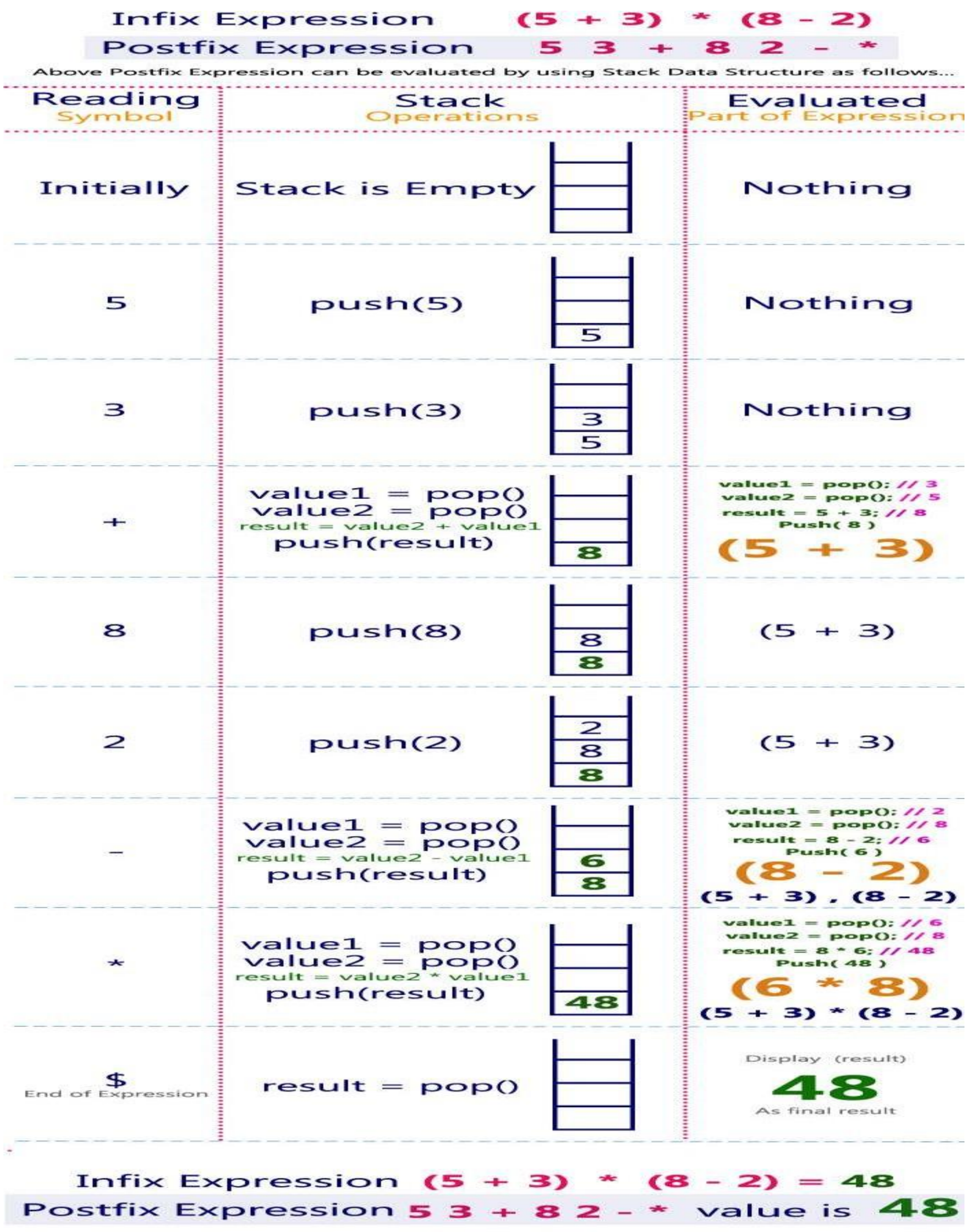
- The postfix expression is evaluated easily by the use of a stack.
- When a number is seen, it is pushed onto the stack;
- when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.
- When an expression is given in postfix notation, there is no need to know any precedence rules.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example2



Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)	5	Nothing
3	push(3)	3 5	Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
		8	value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)	8 8	(5 + 3)
2	push(2)	2 8 8	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
		48	value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)

$ End of Expression	result = pop()		Display (result) 48 As final result

Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Example 3:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

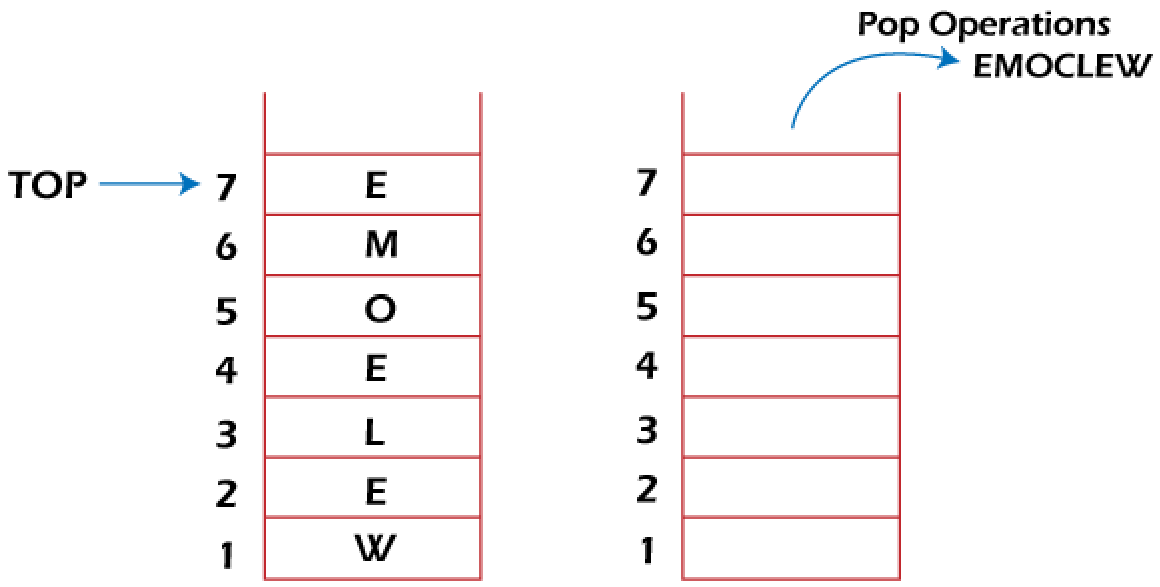
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

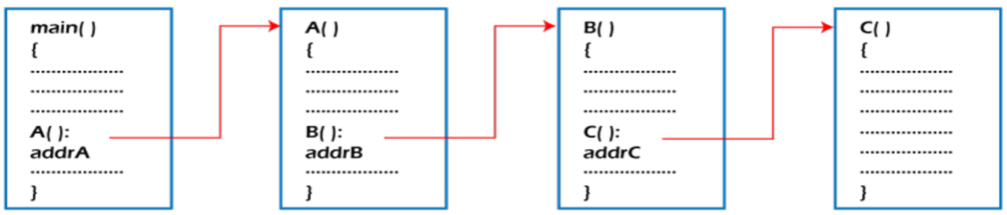
Reverse a String

A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Processing Function Calls:

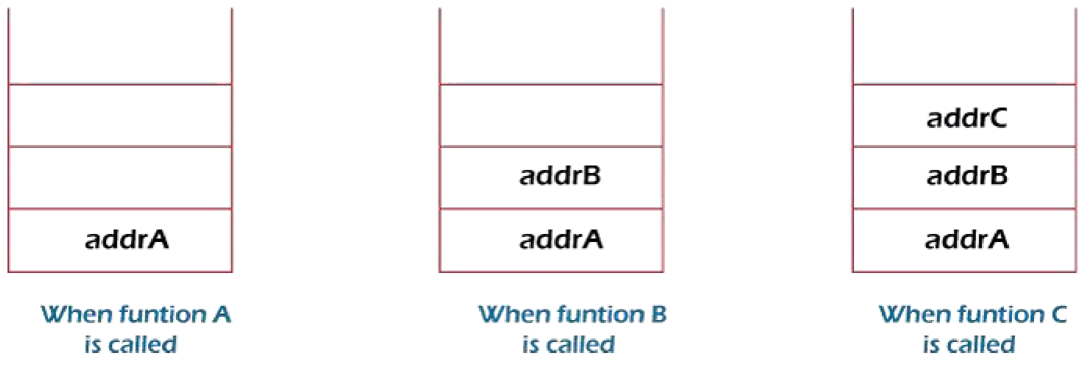
Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.


main( ) { .... A( ): addrA .... }
A( ) { .... B( ): addrB .... }
B( ) { .... C( ): addrC .... }
C( ) { .... }

Function call

When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



Different states of stack

The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

QUEUE

A queue is linear data structure and collection of elements. A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. The principle of queue is a "FIFO" or "First-in-first-out".
Queue is an abstract data structure. A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets theticket.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

More real-world examples can be seen as queues at the ticket windows and bus-stops and our college library.

The operations for a queue are analogues to those for a stack; the difference is that the insertions go at the end of the list, rather than the beginning.

Operations on QUEUE:

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

- **Enqueue or insertion**: which inserts an element at the end of the queue.
- **Dequeue or deletion**: which deletes an element at the start of the queue.

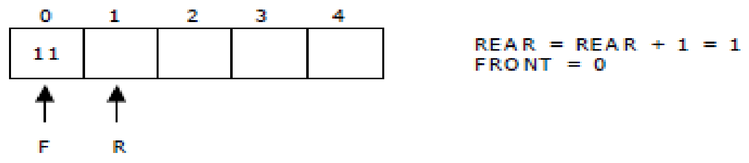
Representation of Queue (or) Implementation of Queue:

The queue can be represented in two ways:

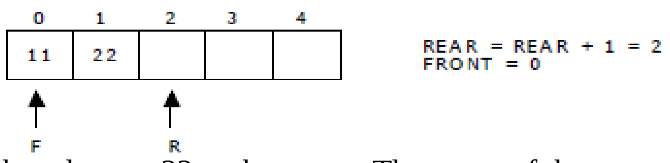
1. Queue using Array
2. Queue using Linked List

1. Queue using Array:

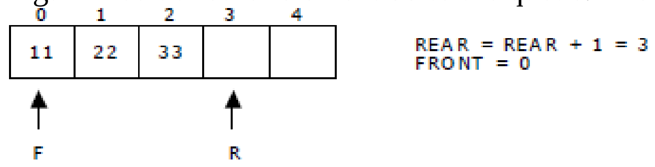
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.
Now, insert 11 to the queue. Then queue status will be:



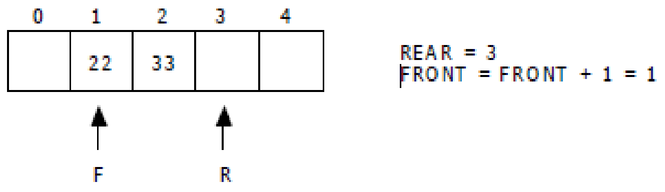
Next, insert 22 to the queue. Then the queue status is:



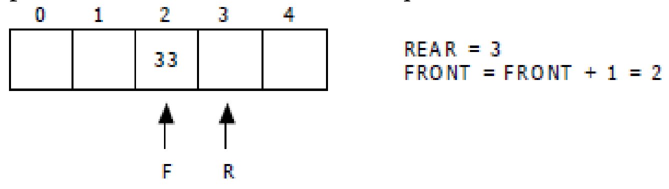
Again insert another element 33 to the queue. The status of the queue is:



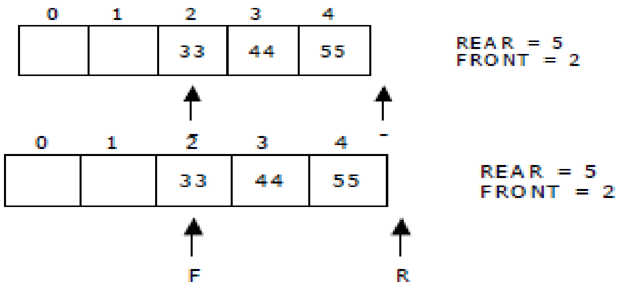
Now, delete an element. The element deleted is the element at the front of the queue.So the status ofthe queue is:



Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So,22 is deleted. The queue status is as follows:

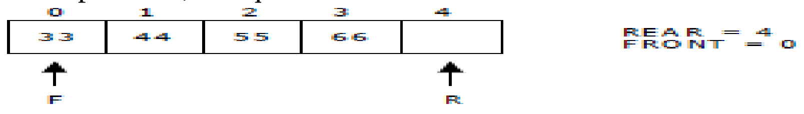


Now,insert new elements 44 and 55 into the queue. The queue status is:



xt insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

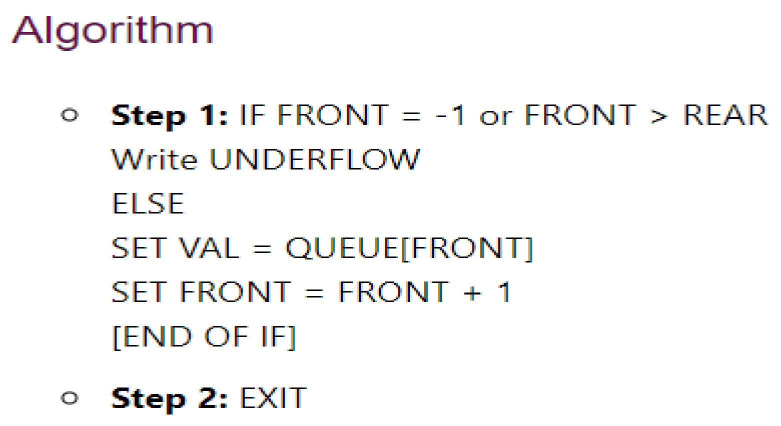
Algorithm

```
○ Step 1: IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]

○ Step 2: IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]

○ Step 3: Set QUEUE[REAR] = NUM

○ Step 4: EXIT
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.



Algorithm

```
○ Step 1: IF FRONT = -1 or FRONT > REAR
          Write UNDERFLOW
          ELSE
          SET VAL = QUEUE[FRONT]
          SET FRONT = FRONT + 1
          [END OF IF]
○ Step 2: EXIT
```

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1 -** Check whether **queue** is **EMPTY**.
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

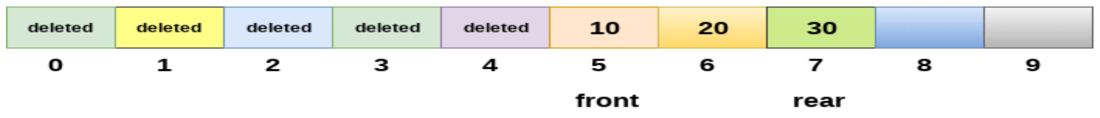
Queue Implementation using Arrays

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void   insert();
void   delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
   int choice;
   while(choice != 4)
   {
        printf("\n*************************Main
Menu*****************************\n");
printf("\n=================================================================
===\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");

}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;

    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1 ;
        }
        else
        {
            front = front + 1;
        }
        printf("\nvalue deleted ");
    }

}

void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values ..... \n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
        }
    }
}
```

Drawback of array implementation of Queue

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

One of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Types of Queues

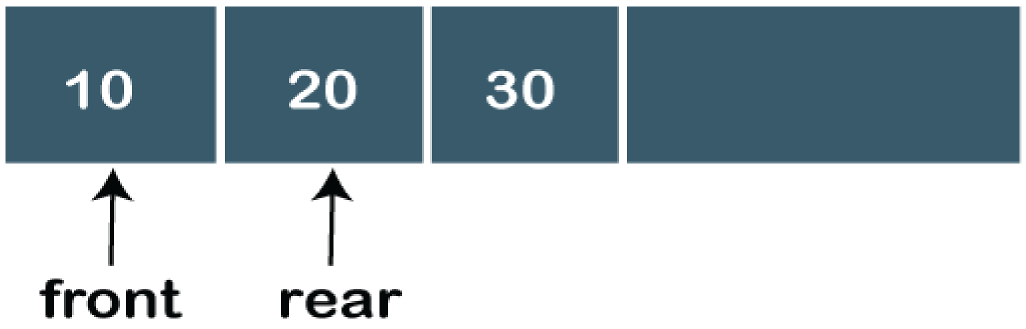
There are four types of Queues**:**

1. Linear Queue
2. Circular Queue
3. Priority Queue
4. Deque

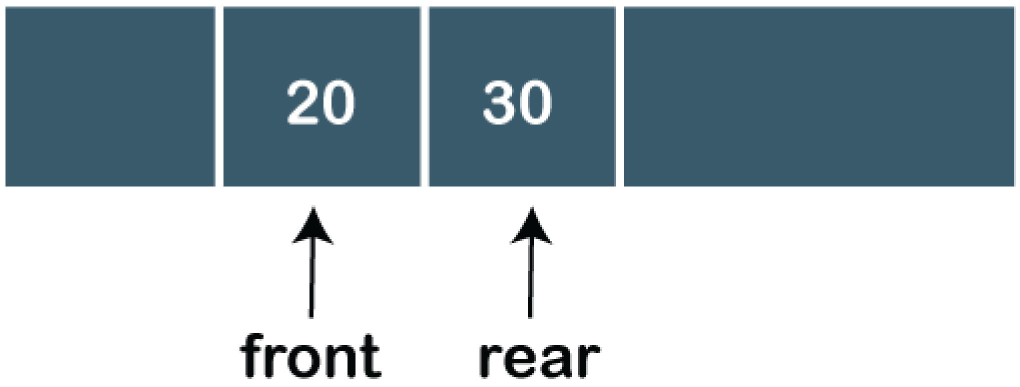
1. Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule. The linear Queue can be represented, as shown in the below

figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:

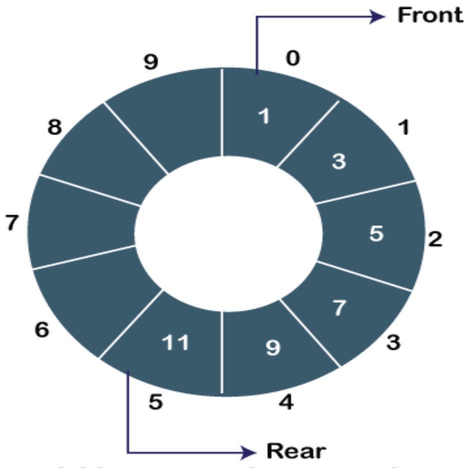


In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue showsthe overflow condition as the rear is pointing to the last element of the Queue

2. Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also knownas Ring Buffer as all the ends are connected to another end. The circular queue can be represented as:
he drawback that occurs in a linear queue is overcome by using the circular queue. If the empty



space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

3. Priority Queue

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.
In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:
The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

4. Deque

Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occurfrom both ends.

UNIT II

UNIT - II

Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.

Basic operations that can be performed on dictionary are:

1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

A dictionary is a general-purpose data structure for storing a group of objects.

- A dictionary has a set of keys and each key has a single associated value.
- When presented with a key the dictionary will return the associated value.
- A dictionary is also called a hash, a map, a hashmap in different programming languages.
- The keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.
- Different languages enforce different type restrictions on keys and values in a dictionary.
- Dictionaries are often implemented as hash tables.
- Keys in a dictionary must be unique an attempt to create a duplicate key will typically overwrite the existing value for that key.
- Dictionary is an abstract data structure that supports the following operations: –

    search(K key) (returns the value associated with the given key)

    insert(K key, V value) – delete(K key)

• Each element stored in a dictionary is identified by a key of type K.

• Dictionary represents a mapping from keys to values.

Dictionaries have numerous applications.

– contact book • key: name of person; value: – telephone number

--table of program variable identiers • key: identier; value: address in memory

– property-value collection • key: property name; value: associated value –

--natural language dictionary • key: word in language X; value: word in language Y – etc

operations on dictionaries

Dictionaries typically support several operations:

– retrieve a value (depending on language, attempting to retrieve a missing key may give a default value or throw an exception)

– insert or update a value (typically, if the key does not exist in the dictionary, the key-value pair is inserted; if the key already exists, its corresponding value is overwritten with the new one)

– remove a key-value pair

– test for existence of a key

Note that items in a dictionary are unordered, so loops over dictionaries will return items in an arbitrary order.

Linear List Representation The dictionary can be represented as a linear list. The linear list is a collection of pair and value. There are two method of representing linear list.

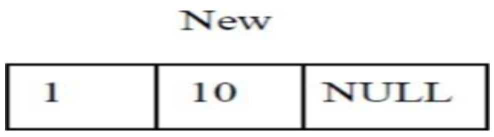
Structure of linear list for dictionary:

To Represent the dictionary with linear list , each node contain the 3 fields : those are key , value and pointer to the next node .

The below is the node structure :

Key	**value**	**Pointer to the next node**

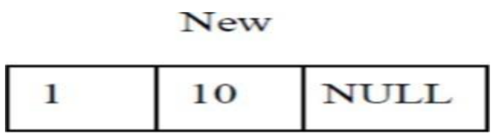
Example :



```
Struct node
{
    Int key;
    Int value;
   Struct node *next;
};
struct node *head;
```

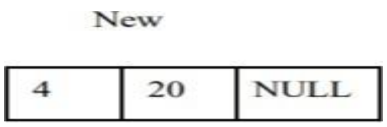
Insertion of new node in Dictionary

Consider that initially dictionary is empty then head = NULL
We will create a new node with some key and value contained in it.

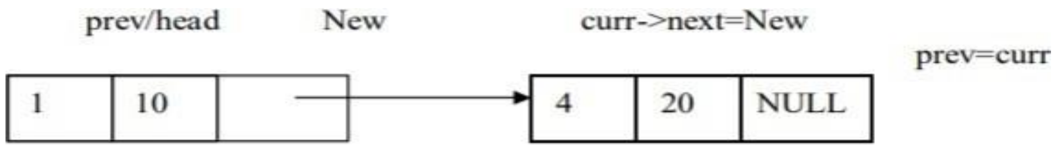


Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node

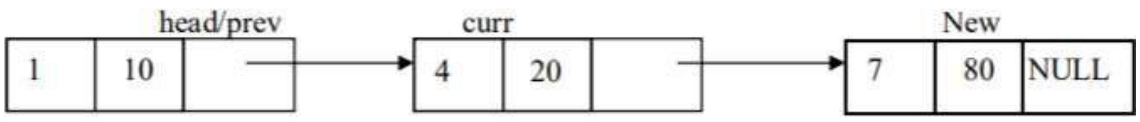
Insert a record, key=4 and value=20,



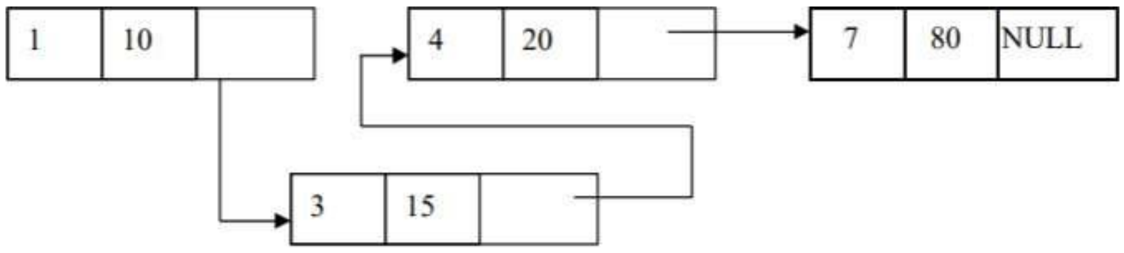
Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node.



Add a new node then

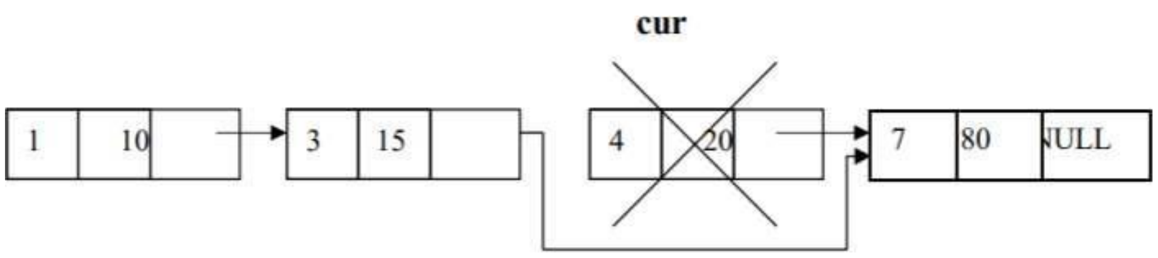


If we insert then we have to search for it proper position by comparing key value. (curr->key < New->key) is false. Hence else part will get executed.

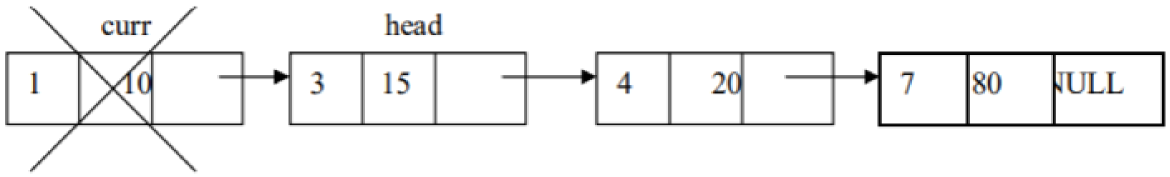


The delete operation:

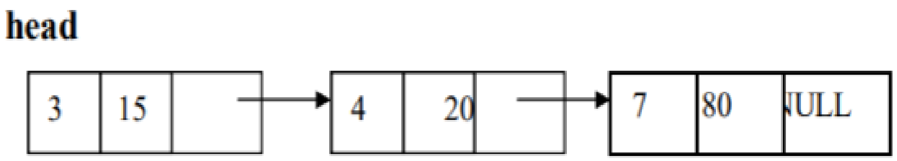
Case 1: Initially assign 'head' node as 'curr' node.Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node's key value. We will get node which is to be deleted in variable 'curr'. The node given by variable 'prev' keeps track of previous node of 'cuu' node. For eg, delete node with key value 4 then



Case 2: If the node to be deleted is head node i.e.. if(curr==head) Then, simply make 'head' node as next node and delete 'curr'



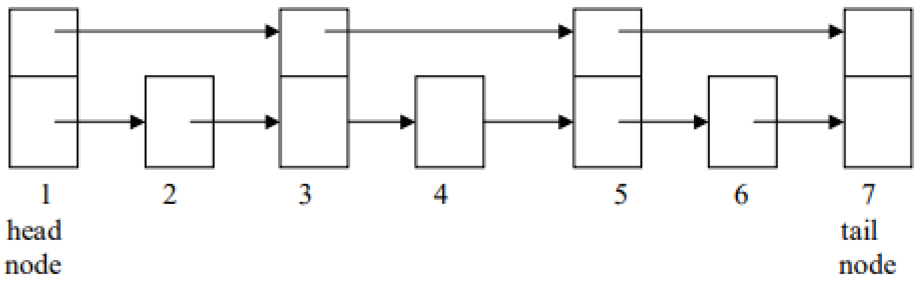
Hence the listbecomes



SKIP LIST REPRESENTATION

A skip list is a data structure that is used for storing a sorted list of items with a help of hierarchy of linked lists that connect increasingly sparse subsequences of the items. A skip list allows the process of item look up in efficient manner. The skip list data structure skips over many of the items of the full list in one step, that's why it is known as skip list.

Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level. There are two special nodes in the skip list oneis head node which is the starting node of the list and tail node is the last node of the list.



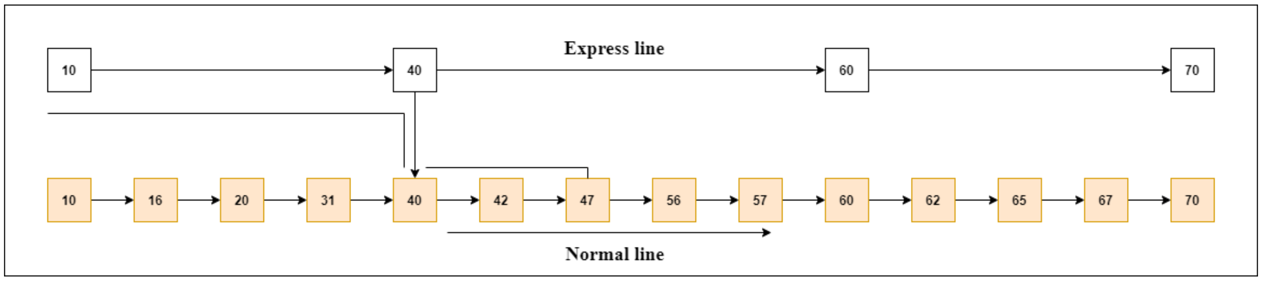
The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.

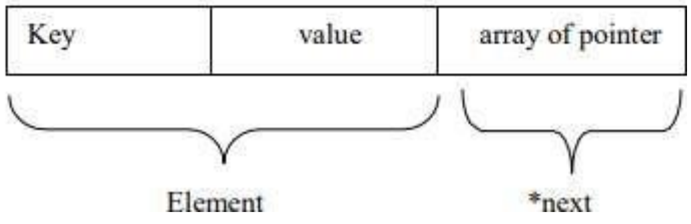


Skip List Basic Operations

There are the following types of operations in the skip list.

- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list

The individual node looks like this:

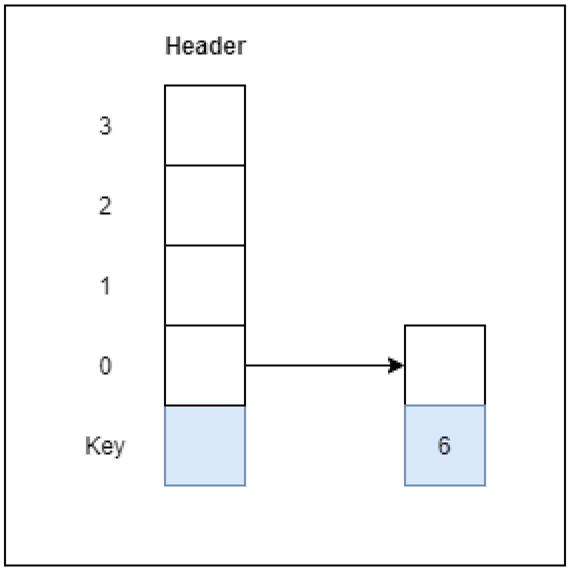


Example 1: Create a skip list, we want to insert these following keys in the empty skip list.

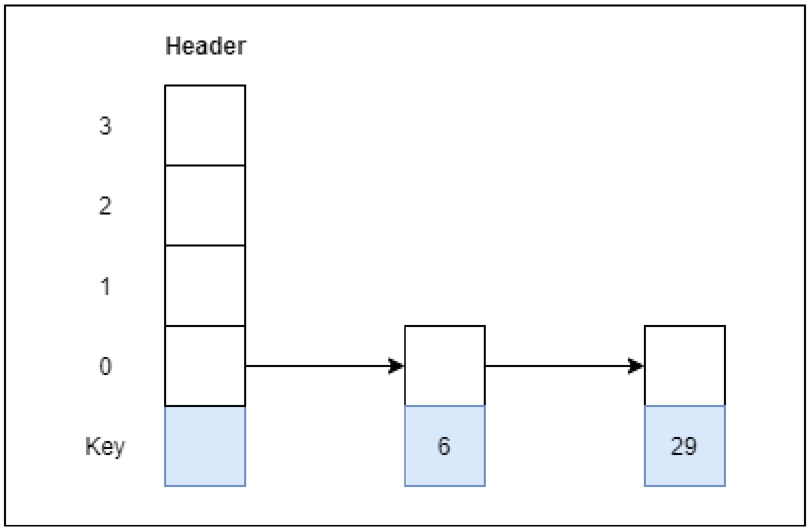
1. 6 with level 1.
2. 29 with level 1.
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

Ans:

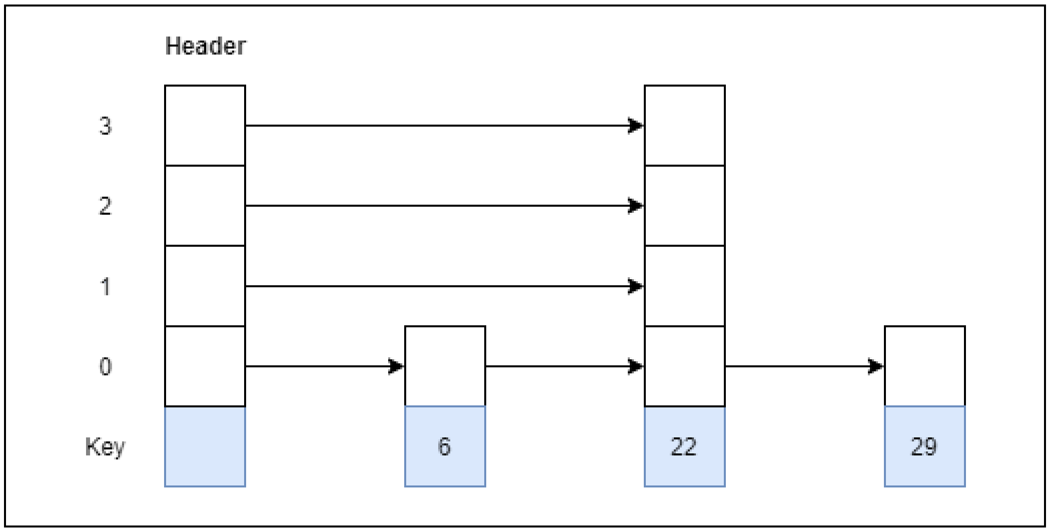
Step 1: Insert 6 with level 1



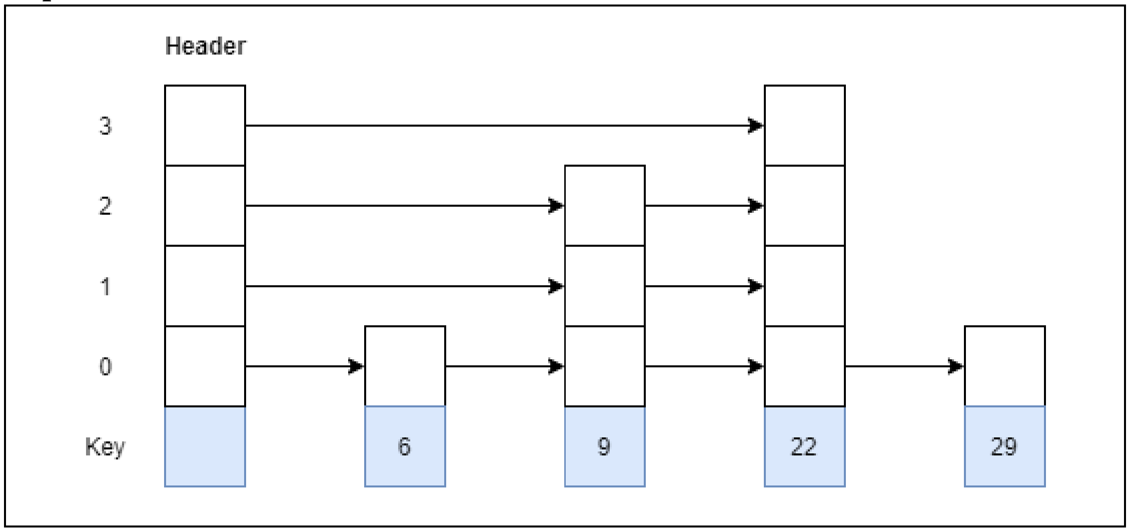
Step 2: Insert 29 with level 1



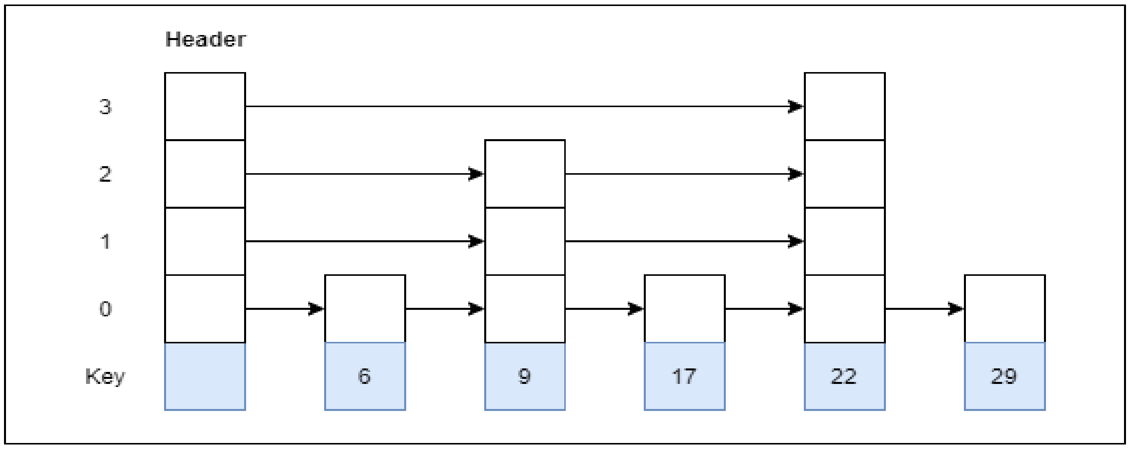
Step 3: Insert 22 with level 4



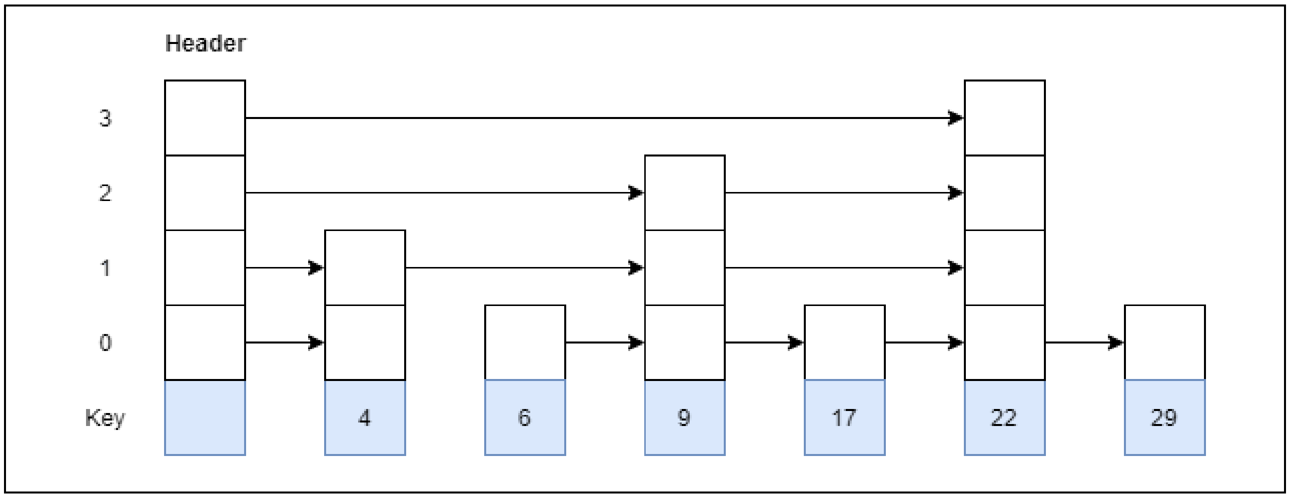
Step 4: Insert 9 with level 3



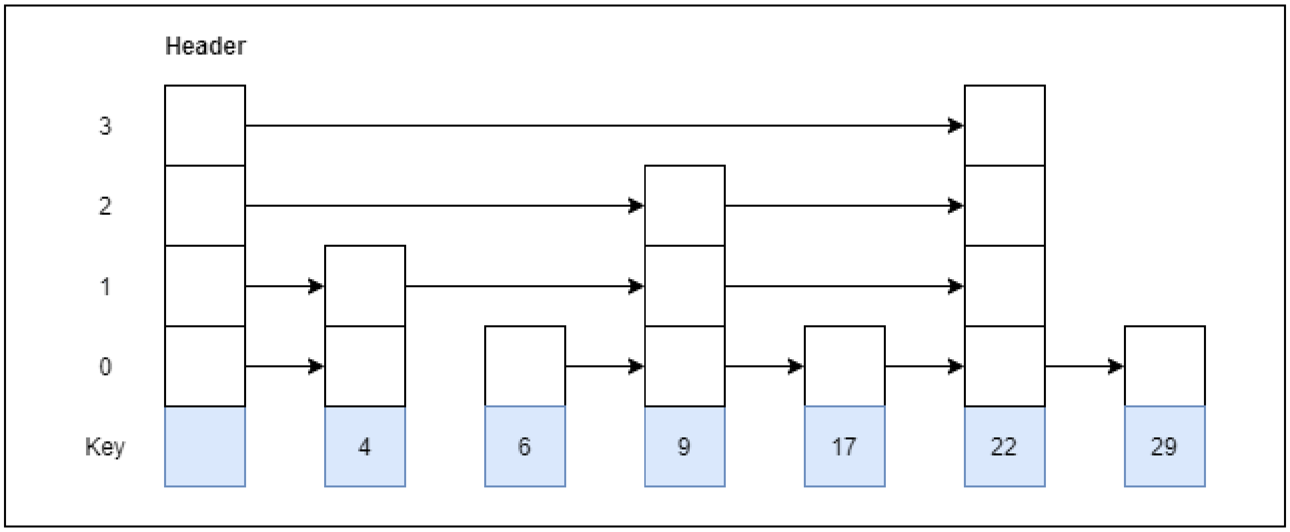
Step 5: Insert 17 with level 1



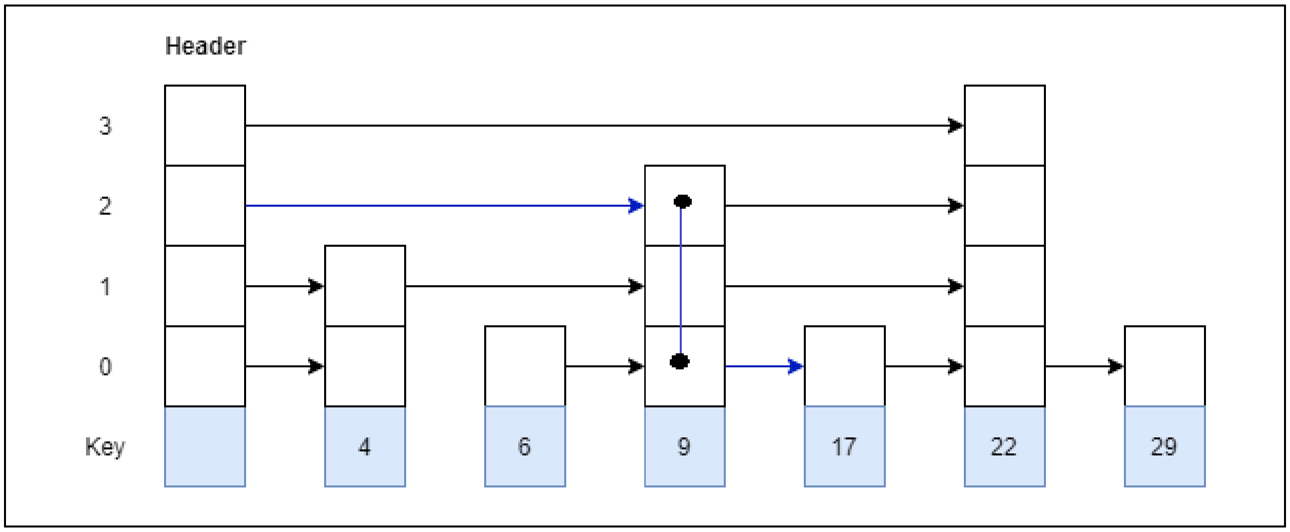
Step 6: Insert 4 with level 2



Example 2: Consider this example where we want to search for key 17.



Ans:



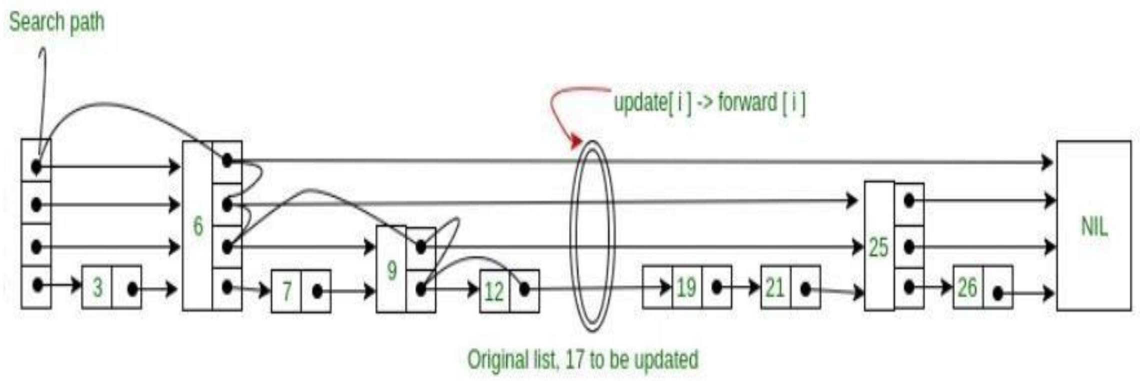
INSERTION IN SKIP LIST

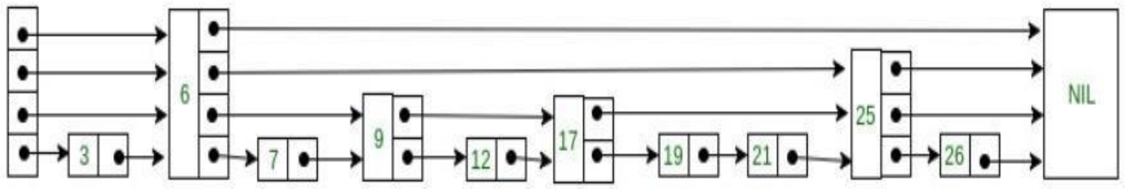
We will start from highest level in the list and compare key of next node of the current node with the key to be inserted. Basic idea is If –

1. Key of next node is less than key to be inserted then we keep on moving forward on the same level
2. Key of next node is greater than the key to be inserted then we store the pointer to current node and move one level down and continue our search.

At the level 0, we will definitely find a position to insert given key.

Consider this example where we want to insert key 17 –





SEARCH IN SKIP LIST

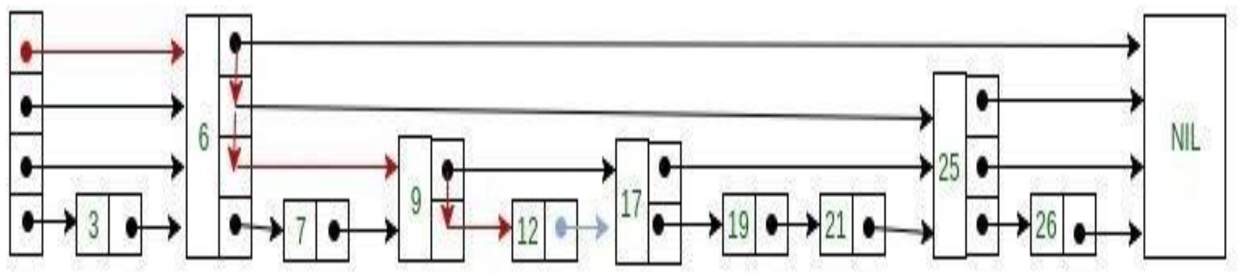
Searching an element is very similar to approach for searching a spot for inserting an element in Skip list. The basic idea is if –

1. Key of next node is less than search key then we keep on moving forward on the same level.
2. Key of next node is greater than the key to be inserted then we store the pointer to current node and move one level down and continue our search.

At the lowest level (0), if the element next to the rightmost element has key equal to the search key, then we have found key otherwise failure.

EXAMPLE:

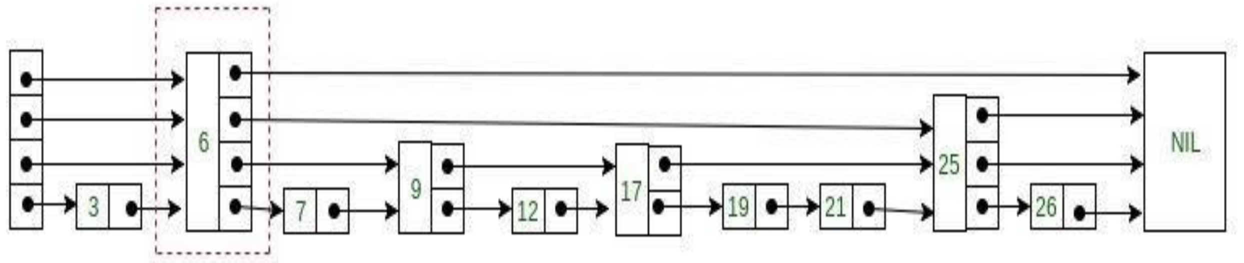
Consider this example where we want to search for key 17-

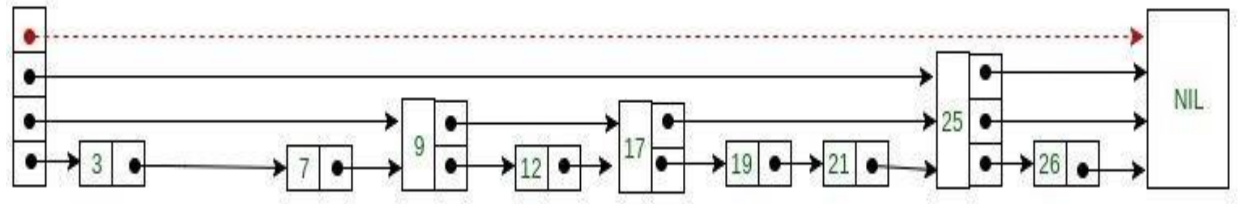


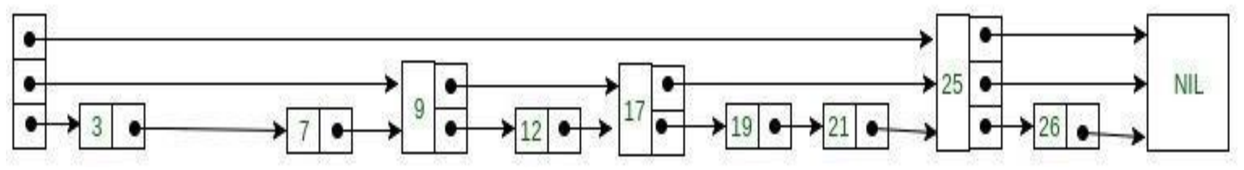
Deleting an element from the Skip list

Deletion of an element k is preceded by locating element in the Skip list using above mentioned search algorithm. Once the element is located, rearrangement of pointers is done to remove element form list just like we do in singly linked list. We start from lowest level and do rearrangement until element next is not k.
After deletion of element there could be levels with no elements, so we will remove these levels as well by decrementing the level of Skip list.

Consider this example where we want to delete element 6







Here at level 3, there is no element (arrow in red) after deleting element 6. So we will decrement level of skip list by 1.

Advantages of the Skip list

1. If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2. The skip list is simple to implement as compared to the hash table and the binary search tree.
3. It is very simple to find a node in the list because it stores the nodes in sorted form.
4. The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5. The skip list is a robust and reliable list.

Disadvantages of the Skip list

1. It requires more memory than the balanced tree.
2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

1. It is used in distributed applications, and it represents the pointers and system in the distributed applications.
2. It is used to implement a dynamic elastic concurrent queue with low lock contention.
3. The indexing of the skip list is used in running median problems.
4. The skip list is used for the delta-encoding posting in the Lucene search.

Hash Table Representation: hash functions, collision resolution-separate chaining, open addressing linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

HASHING

- There are several searching techniques like linear search, binary search, search trees etc.
- In these techniques, time taken to search any particular element depends on the total number of elements.
- Linear Search takes O(n) time to perform the search in unsorted arrays consisting of n elements.
- Binary Search takes O(logn) time to perform the search in sorted arrays consisting of n elements.
- It takes O(logn) time to perform the search in Binary Search Tree consisting of n elements

Drawbacks
The main drawback of these techniques is-

- As the number of elements increases, time taken to perform the search also increases.
- This becomes problematic when total number of elements become too large.

Hashing in Data Structure
In data structures,

- Hashing is a well-known technique to search any particular element among several elements.
- It minimizes the number of comparisons while performing the search.

Advantages

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

Hashing Mechanism

In hashing,

- An array data structure called as Hash table is used to store the data items.
- Based on the hash key value, data items are inserted into the hash table.

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers)can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use *hashing*.

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Hash function

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Need for a good hash function

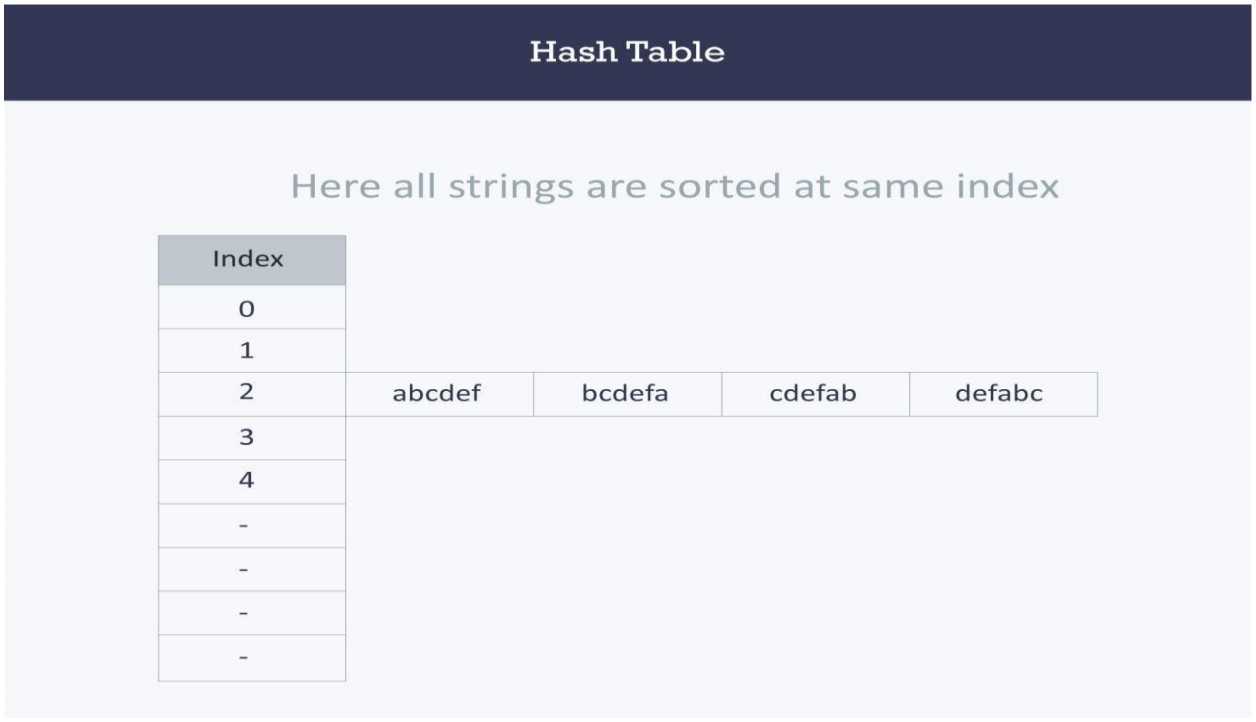
Let us understand the need for a good hash function. Assume that you have to store strings in the hash table by using the hashing technique {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

To compute the index for storing the strings, use a hash function that states the following:

The index for a specific string will be equal to the sum of the ASCII values of the characters modulo 599.

As 599 is a prime number, it will reduce the possibility of indexing different strings (collisions). It is recommended that you use prime numbers in case of modulo. The ASCII values of a, b, c, d, e, and f are 97, 98, 99, 100, 101, and 102 respectively. Since all the strings contain the same characters with different permutations, the sum will 599.

The hash function will compute the same index for all the strings and the strings will be stored in the hash table in the following format. As the index of all the strings is the same, you can create a list on that index and insert all the strings in that list.



Hash Table

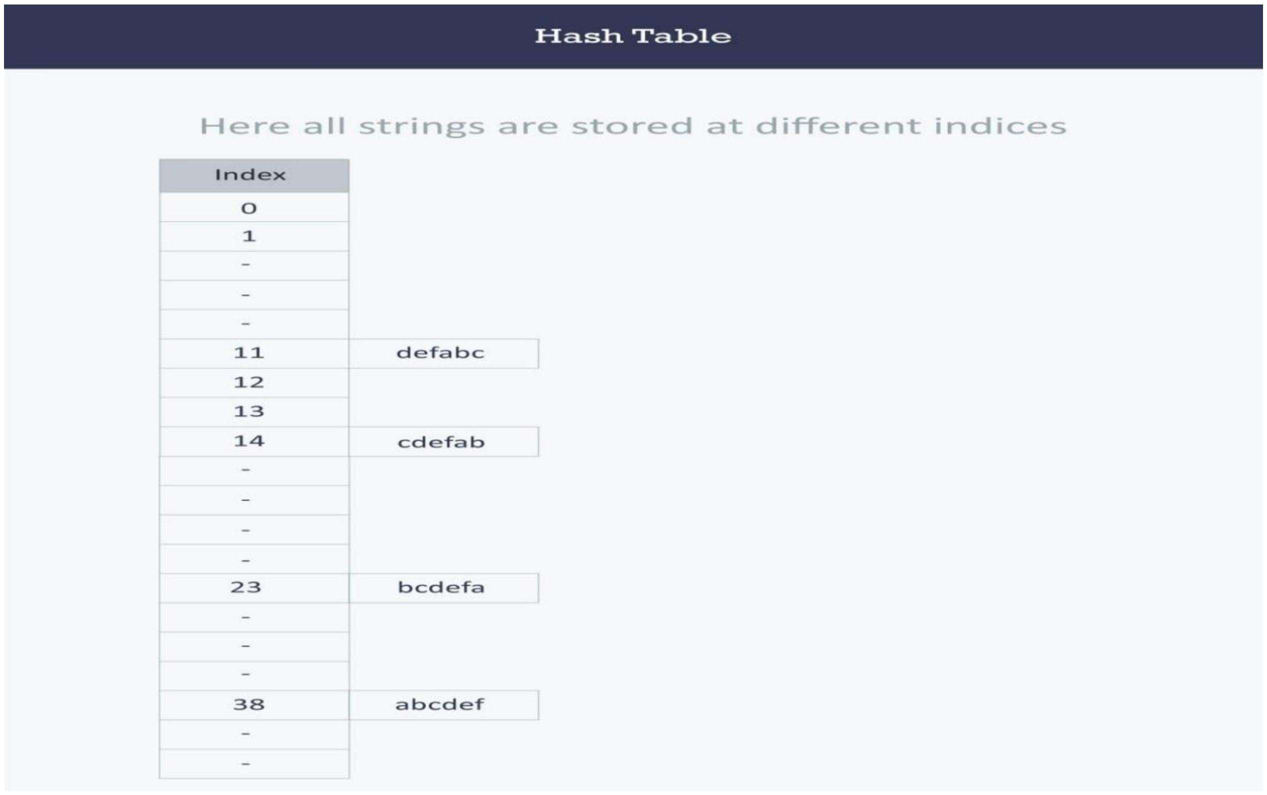
Here all strings are sorted at same index

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

Let's try a different hash function. The index for a specific string will be equal to sum of ASCII values of characters multiplied by their respective order in the string after which it is modulo with 2069 (prime number).

String	Hash function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23

cdefab (991 + 1002 + 1013 + 1024 + 975 + 986)%2069 14
defabc (1001 + 1012 + 1023 + 974 + 985 + 996)%2069 11



Hash Table

Here all strings are stored at different indices

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Hash table

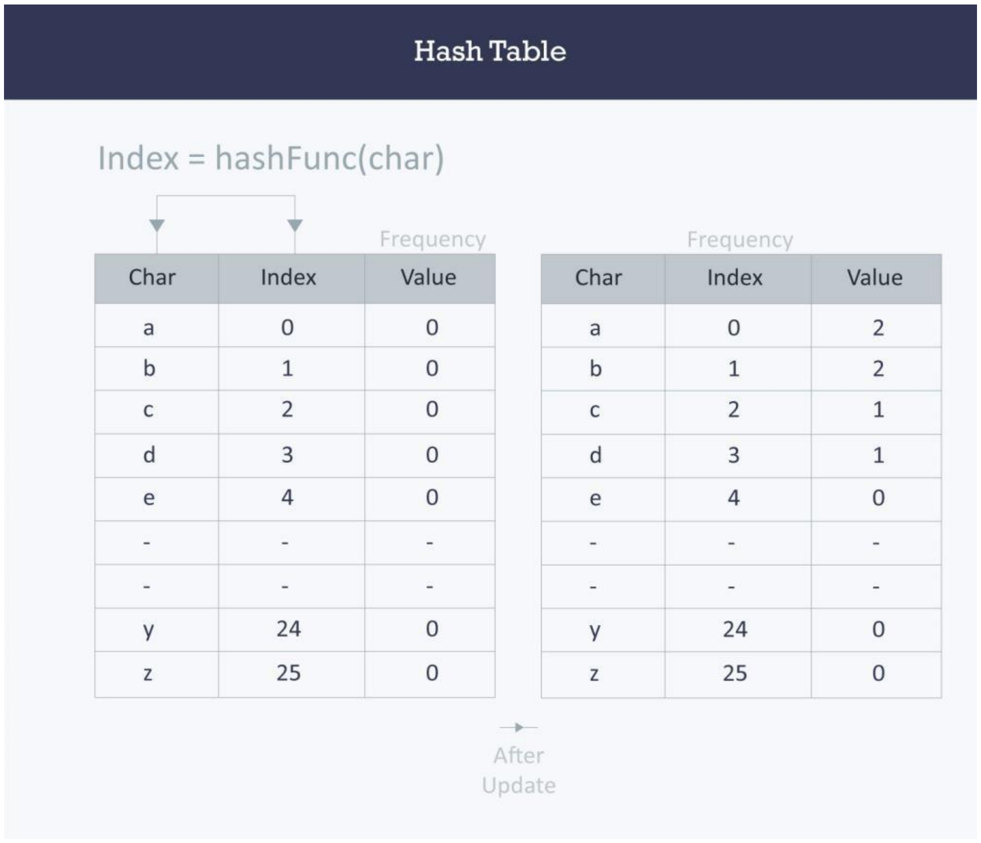
A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched. By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

Let us consider string S. You are required to count the frequency of all the characters in this string.

string S = “ababcd”

The simplest way to do this is to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is **O(26*N)** where **N** is the size of the string and there are 26 possible characters.

Let us apply hashing to this problem. Take an array frequency of size 26 and hash the 26 characters with indices of the array by using the hash function. Then, iterate over the string and increase the value in the frequency at the corresponding index for each character. The complexity of this approach is **O(N)** where **N** is the size of the string


Hash Table
Index = hashFunc(char)
Frequency
Frequency
After Update

Char	Index	Value
a	0	0
b	1	0
c	2	0
d	3	0
e	4	0
-	-	-
-	-	-
y	24	0
z	25	0

Char	Index	Value
a	0	2
b	1	2
c	2	1
d	3	1
e	4	0
-	-	-
-	-	-
y	24	0
z	25	0

TYPES OF HASH FUNCTION

There are various types of hash functions that are used to place the record in the hash table-

1. Division Method:
2. Mid Square:
3. Digit Folding:

Division Method: The hash function depends upon the remainder of division.Typically the divisor is table length.
For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

h(key) = record % table size

54%10=4
72%10=2
89%10=9
37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number.
Consider that if we want to place a record 3111 then

$3111^2 = 9678321$
for the hash table of size
1000 H(3111) = 783 (the
middle 3 digits)

Digit Folding:
The key is divided into separate parts and using some simple operation these parts arecombined to produce the hash key.
For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and theseare added together
H(key) = 123+654+12= 789
The record will be placed at location 789

COLLISION

The hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

Definition: The situation in which the hash function returns the same hash key (home bucket) formore than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions

EXAMPLE

For example,

Consider a hash function.

H(key) = recordkey%10 having the hash table size of 10

The record keys to be placed are

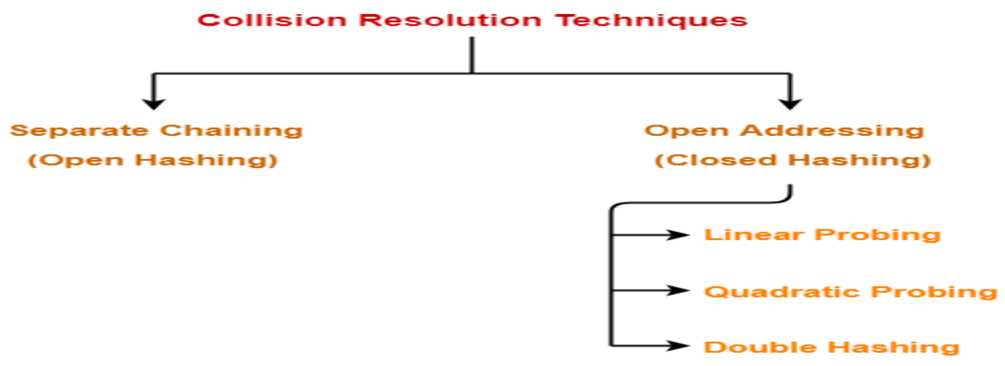
131, 44, 43, 78, 19, 36, 57 and 77
131%10=1
44%10=4
43%10=3
78%10=8
19%10=9
36%10=6
57%10=7
77%10=7

0	
1	131
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8.9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

COLLISION RESOLUTION TECHNIQUES

Collision Resolution Techniques are the techniques used for resolving or handling the collision. Collision resolution techniques are classified as-



1. Separate Chaining
2. Open Addressing

Separate Chaining-

To handle the collision,

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as **separate chaining**.

For Searching-

- In worst case, all the keys might map to the same bucket of the hash table.
- In such a case, all the keys will be present in a single linked list.
- Sequential search will have to be performed on the linked list to perform the search.
- So, time taken for searching in worst case is O(n).

For Deletion-

- In worst case, the key might have to be searched first and then deleted.
- In worst case, time taken for searching is O(n).
- So, time taken for deletion in worst case is O(n).

Load Factor (α)-

Load factor (α) is defined as-

$$\text{Load Factor } (\alpha) = \frac{\text{Number of elements present in the hash table}}{\text{Total size of the hash table}}$$

If Load factor (α) = constant, then time complexity of Insert, Search, Delete = Θ(1)

EXAMPLE:

Using the hash function 'key mod 7', insert the following sequence of keys in the hash table-

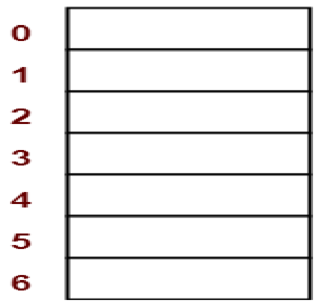
50, 700, 76, 85, 92, 73 and 101

Use separate chaining technique for collision resolution.

The given sequence of keys will be inserted in the hash table as-

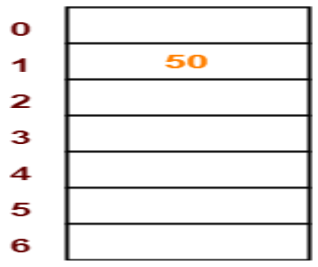
Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-



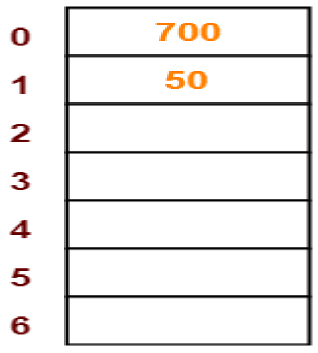
Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-



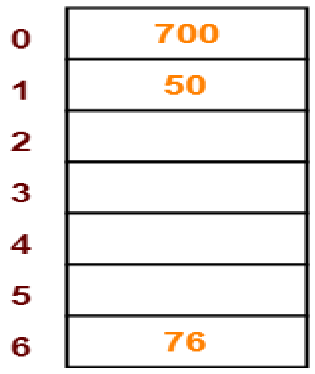
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



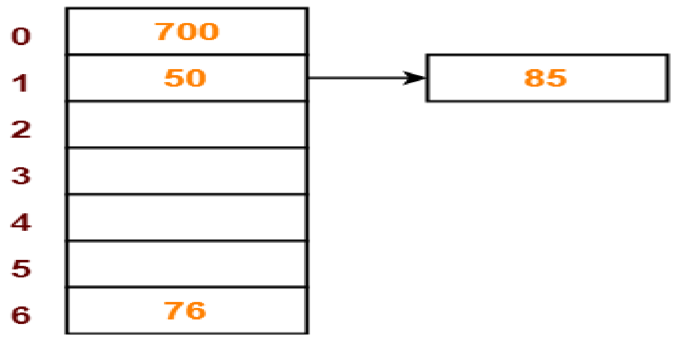
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



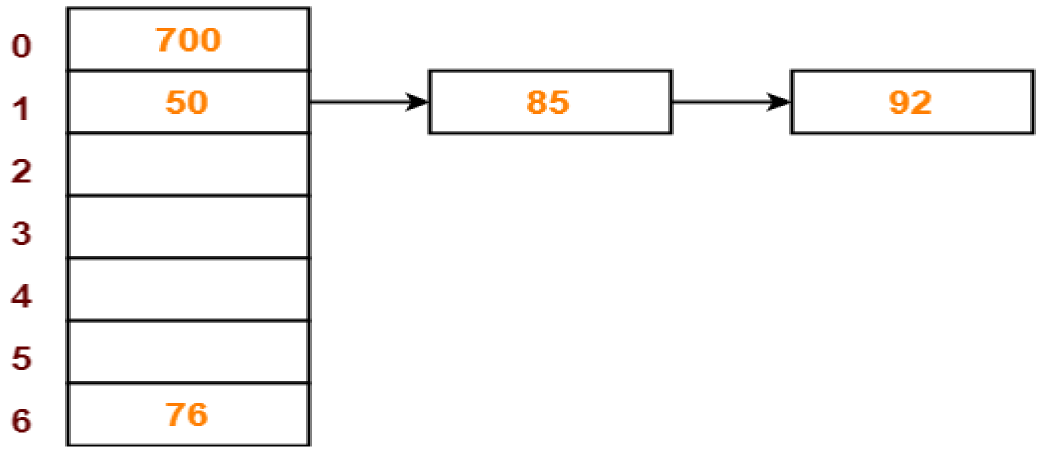
Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 85 will be inserted in bucket-1 of the hash table as-



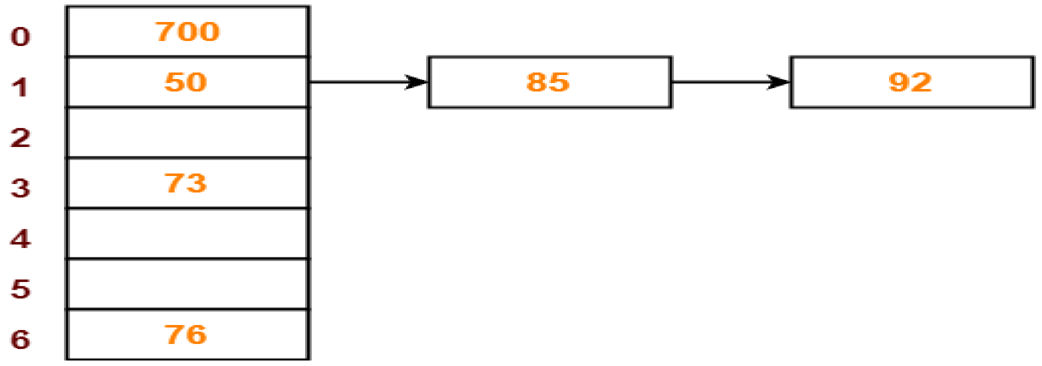
Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-1.
- So, key 92 will be inserted in bucket-1 of the hash table as-



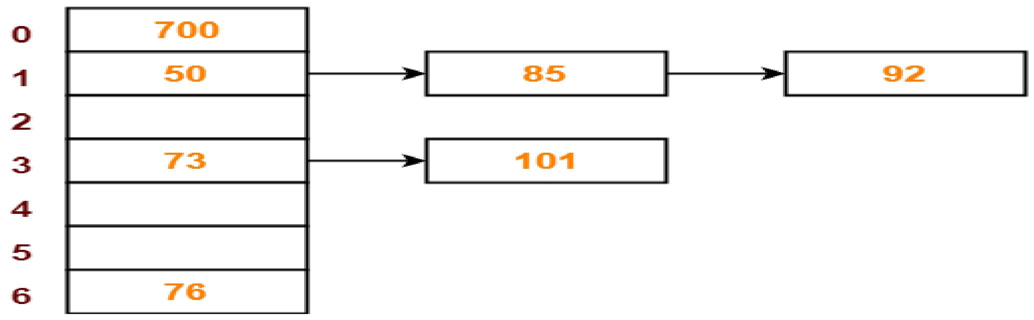
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- So, key 73 will be inserted in bucket-3 of the hash table as-



Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- Separate chaining handles the collision by creating a linked list to bucket-3.
- So, key 101 will be inserted in bucket-3 of the hash table as-



The **advantages** of separate chaining hashing are as follows −

- Separate chaining technique is not sensitive to the size of the table.
- The idea and the implementation are simple.

The **disadvantages** of separate chaining hashing are as follows −

- Keys are not evenly distributed in separate chaining.
- Separate chaining can lead to empty spaces in the table.
- The list in the positions can be very long.

Open Addressing-

In open addressing,

- Unlike separate chaining, all the keys are stored inside the hash table.
- No key is stored outside the hash table.

Techniques used for open addressing are-

- Linear Probing
- Quadratic Probing
- Double Hashing

Operations in Open Addressing-

Let us discuss how operations are performed in open addressing-

Insert Operation-

- Hash function is used to compute the hash value for a key to be inserted.
- Hash value is then used as an index to store the key in the hash table.

In case of collision,

- Probing is performed until an empty bucket is found.
- Once an empty bucket is found, the key is inserted.
- Probing is performed in accordance with the technique used for open addressing.

Search Operation-

To search any particular key,

- Its hash value is obtained using the hash function used.
- Using the hash value, that bucket of the hash table is checked.

- If the required key is found, the key is searched.
- Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
- The empty bucket indicates that the key is not present in the hash table.

Delete Operation-

- The key is first searched and then deleted.
- After deleting the key, that particular bucket is marked as “deleted”.

NOTE-

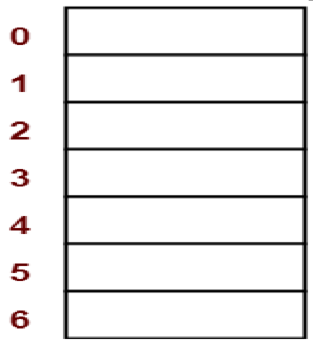
- During insertion, the buckets marked as “deleted” are treated like any other empty bucket.
- During searching, the search is not terminated on encountering the bucket marked as“deleted”.
- The search terminates only after the required key or an empty bucket is found.

Linear Probing

Using the hash function ‘key mod 7’, insert the following sequence of keys in the hash table-
50, 700, 76, 85, 92, 73 and 101
Use linear probing technique for collision resolution.

Step-01:

- Draw an empty hash table.
- For the given hash function, the possible range of hash values is [0, 6].
- So, draw an empty hash table consisting of 7 buckets as-

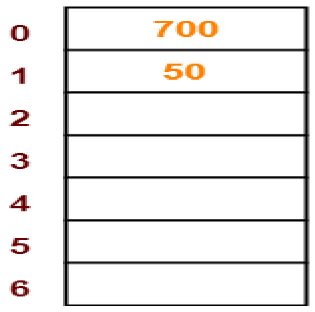


Step-02:

- Insert the given keys in the hash table one by one.
- The first key to be inserted in the hash table = 50.
- Bucket of the hash table to which key 50 maps = 50 mod 7 = 1.
- So, key 50 will be inserted in bucket-1 of the hash table as-

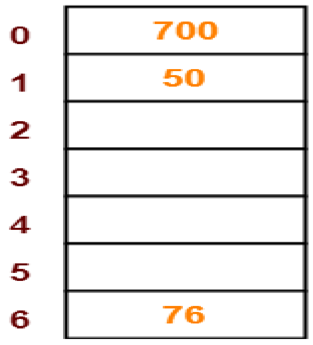
Step-03:

- The next key to be inserted in the hash table = 700.
- Bucket of the hash table to which key 700 maps = 700 mod 7 = 0.
- So, key 700 will be inserted in bucket-0 of the hash table as-



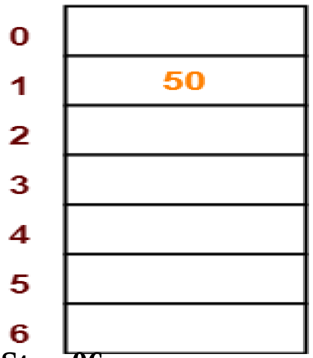
Step-04:

- The next key to be inserted in the hash table = 76.
- Bucket of the hash table to which key 76 maps = 76 mod 7 = 6.
- So, key 76 will be inserted in bucket-6 of the hash table as-



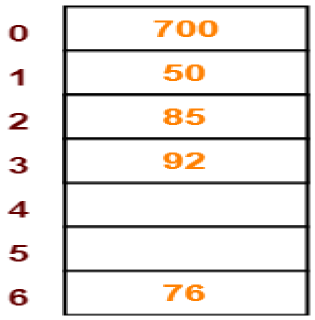
Step-05:

- The next key to be inserted in the hash table = 85.
- Bucket of the hash table to which key 85 maps = 85 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-2.
- So, key 85 will be inserted in bucket-2 of the hash table as-



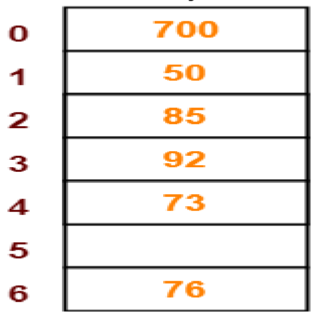
Step-06:

- The next key to be inserted in the hash table = 92.
- Bucket of the hash table to which key 92 maps = 92 mod 7 = 1.
- Since bucket-1 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-3.
- So, key 92 will be inserted in bucket-3 of the hash table as-



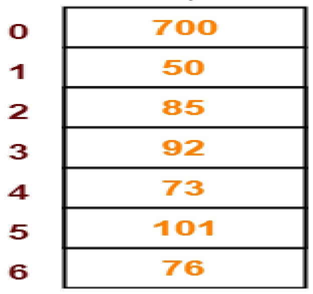
Step-07:

- The next key to be inserted in the hash table = 73.
- Bucket of the hash table to which key 73 maps = 73 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-4.
- So, key 73 will be inserted in bucket-4 of the hash table as-



Step-08:

- The next key to be inserted in the hash table = 101.
- Bucket of the hash table to which key 101 maps = 101 mod 7 = 3.
- Since bucket-3 is already occupied, so collision occurs.
- To handle the collision, linear probing technique keeps probing linearly until an empty bucket is found.
- The first empty bucket is bucket-5.
- So, key 101 will be inserted in bucket-5 of the hash table as-



0	700
1	50
2	85
3	92
4	73
5	101
6	76

The **advantages** of linear probing are as follows –

- Linear probing requires very less memory.
- It is less complex and is simpler to implement.

The **disadvantages** of linear probing are as follows –

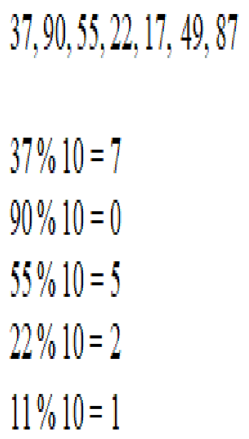
- Linear probing causes a scenario called "primary clustering" in which there are large blocks of occupied cells within the hash table.
- The values in linear probing tend to cluster which makes the probe sequence longer and lengthier.
-

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$\boxed{H(key) = (Hash(key) + i^2) \% m)}$$

where m can be table size or any prime number.
for eg; If we have to insert following elements in the hash table with table size 10:



37, 90, 55, 22, 17, 49, 87

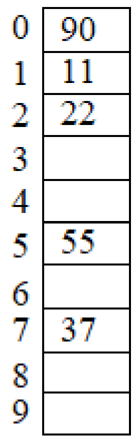
37 % 10 = 7

90 % 10 = 0

55 % 10 = 5

22 % 10 = 2

11 % 10 = 1

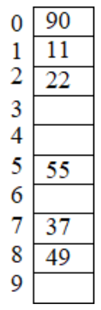


0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

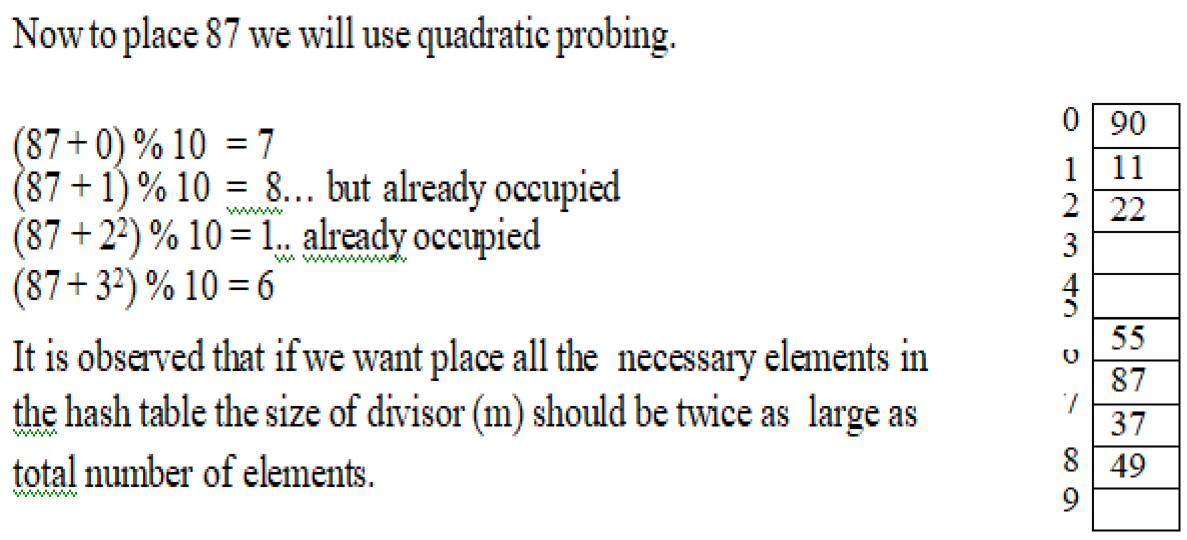
Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.
Hi (key) = (Hash(key) + i^2) % m
Consider i = 0 then
(17 + 0^2) % 10 = 7
(17 + 1^2) % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8.
Then comes 49 which will be placed at index 9.

49 % 10 = 9



0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	



Now to place 87 we will use quadratic probing.

$(87 + 0) \% 10 = 7$
$(87 + 1) \% 10 = 8$… but already occupied
$(87 + 2^2) \% 10 = 1$.. already occupied
$(87 + 3^2) \% 10 = 6$

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	

The **advantages** of quadratic probing is as follows −

- Quadratic probing is less likely to have the problem of primary clustering and is easier to implement than Double Hashing.

The **disadvantages** of quadratic probing are as follows −

- Quadratic probing has secondary clustering. This occurs when 2 keys hash to the same location, they have the same probe sequence. So, it may take many attempts before an insertion is being made.
- Also probe sequences do not probe all locations in the table.

Double hashing

Double Hashing is a hashing collision resolution technique where we use 2 hash functions.
Double Hashing - Hash Function 1
h_i = (Hash(X) + F(i)) % Table Size
where

- F(i) = i * $hash_2(X)$
- X is the Key or the Number for which the hashing is done
- i is the i^{th} time that hashing is done for the same value. Hashing is repeated only when collision occurs
- Table size is the size of the table in which hashing is done

This F(i) will generate the sequence such as $hash_2(X)$, 2 * $hash_2(X)$ and so on. Double Hashing - Hash Function 2
We use second hash function as
$hash_2(X)$ = R - (X mod R)
where

- R is the prime number which is slightly smaller than the Table Size.

- X is the Key or the Number for which the hashing is done
  Double Hashing Example - Closed Hash Table

Let us consider the same example in which we choose R = 7.

0	68
1	
2	39
3	89
4	
5	
6	
7	
8	28
9	79

A Closed Hash Table using Double Hashing

Key	**Hash Function h(X)**	**Index**	**Collision**	**Alt Index**
79	$h_0(79)$ = (Hash(79) + F(0)) % 10 = ((79 % 10) + 0) % 10 = 9	9		
28	$h_0(28)$ = (Hash(28) + F(0)) % 10 = ((28 % 10) + 0) % 10 = 8	8		
39	$h_0(39)$ = (Hash(39) + F(0)) % 10 = ((39 % 10) + 0) % 10 = 9	9	first collision occurs	
	$h_1(39)$ = (Hash(39) + F(1)) % 10 = ((39 % 10) + 1(7-(39 % 7))) % 10 = (9 + 3) % 10 =12 % 10 = 2	2		2
68	$h_0(68)$ = (Hash(68) + F(0)) % 10 = ((68 % 10) + 0) % 10 = 8	8	collision occurs	
	$h_1(68)$ = (Hash(68) + F(1)) % 10 = ((68 % 10) + 1(7-(68 % 7))) % 10 = (8 + 2) % 10 =10 % 10 =0	0		0

89	$h_0(89)$ = (Hash(89) + F(0)) % 10 = ((89 % 10) + 0) % 10 = 9	9	collision occurs	
	$h_1(89)$ = (Hash(89) + F(1)) % 10 = ((89 % 10) + 1(7-(89 % 7))) % 10 = (9 + 2) % 10 =10 % 10 = 0	0	Again collision occurs	
	$h_2(89)$ = (Hash(89) + F(2)) % 10 = ((89 % 10) + 2(7-(89 % 7))) % 10 = (9 + 4) % 10=13 % 10 =3	3		3

he **advantage** of double hashing is as follows −

- Double hashing finally overcomes the problems of the clustering issue.

The **disadvantages** of double hashing are as follows:

- Double hashing is more difficult to implement than any other.

Comparison of Open Addressing Techniques-

	Linear Probing	Quadratic Probing	Double Hashing
Primary Clustering	Yes	No	No
Secondary Clustering	Yes	Yes	No
Number of Probe Sequence (m = size of table)	m	m	m^2
Cache performance	Best	Lies between the two	Poor

Separate Chaining Vs Open Addressing-

Separate Chaining	**Open Addressing**
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.
Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

Which is the Preferred Technique?

The performance of both the techniques depend on the kind of operations that are required to be performed on the keys stored in the hash table-

Separate Chaining-

Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table-

- Insertion Operation
- Deletion Operation
- Searching Operation

NOTE-

- Deletion is easier in separate chaining.
- This is because deleting a key from the hash table does not affect the other keys stored in the hash table.

Open Addressing-

Open addressing is advantageous when it is required to perform only the following operations on the keys stored in the hash table-

- Insertion Operation
- Searching Operation

NOTE-

- Deletion is difficult in open addressing.
- This is because deleting a key from the hash table requires some extra efforts.
- After deleting a key, certain keys have to be rearranged.

REHASHING

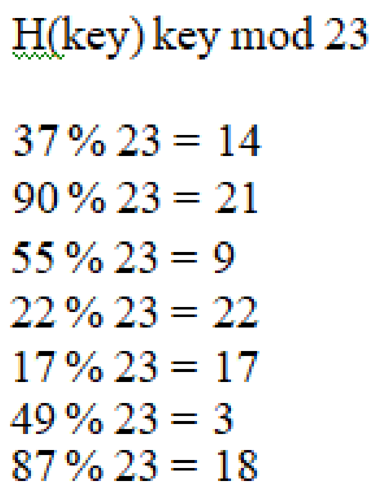
Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

When table is completely full
With quadratic probing when the table is filled half.
When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by recomputing their positions using hash functions.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,
H(key) = key mod tablesize
37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7
Collision solved by linear probing
49 % 10 = 9

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be



H(key) key mod 23

37 % 23 = 14

90 % 23 = 21

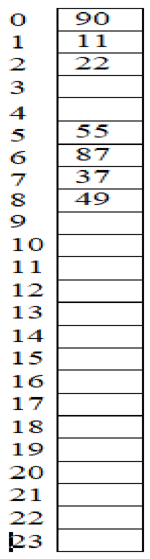
55 % 23 = 9

22 % 23 = 22

17 % 23 = 17

49 % 23 = 3

87 % 23 = 18



0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Now the hash table is sufficiently large to accommodate new insertions.

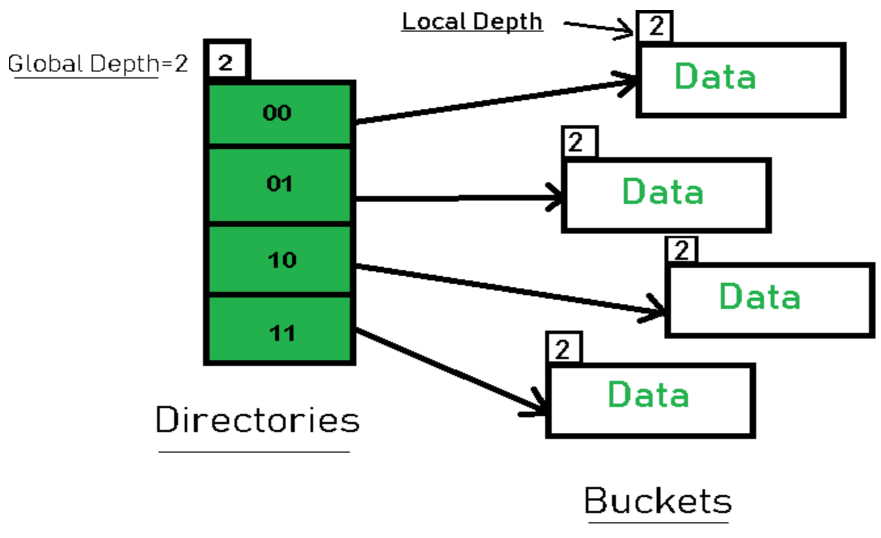
Advantages:
This technique provides the programmer a flexibility to enlarge the table size if required.
Only the space gets doubled with simple hash function which avoids occurrence of collisions

Extendible Hashing

Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.
Main features of Extendible Hashing: The main features in this hashing technique are:

- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

Basic Structure of Extendible Hashing:

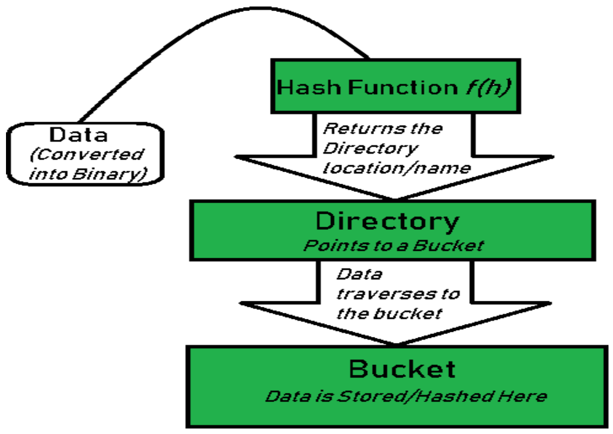


Extendible Hashing

Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

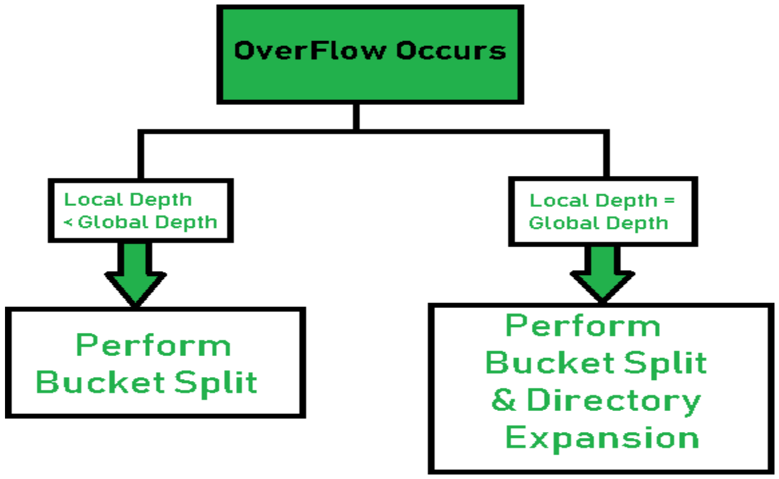
Basic Working of Extendible Hashing:


Data (Converted into Binary)
Hash Function f(h)
Returns the Directory location/name
Directory
Points to a Bucket
Data traverses to the bucket
Bucket
Data is Stored/Hashed Here

- **Step 1 – Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
- **Step 2 – Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binaryform is 110001.
- **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
- **Step 4 – Identify the Directory:** Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id.
  Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
- **Step 5 – Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.

- **Step 6 – Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
- **Step 7 – Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data. First, Check if the local depth is less than or equal to the global depth. Then choose oneof the cases below.

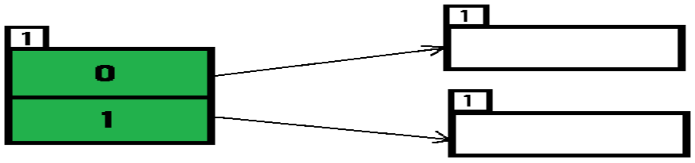
- **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.
  Directory expansion will double the number of directories present in the hash structure.
- **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.


OverFlow Occurs
Local Depth < Global Depth
Local Depth = Global Depth
Perform Bucket Split
Perform Bucket Split & Directory Expansion

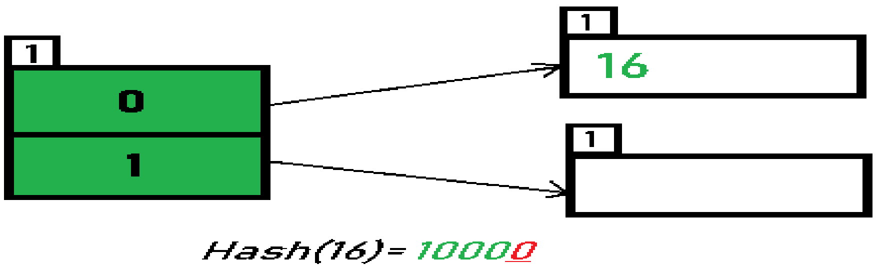
- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9** – The element is successfully hashed.

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

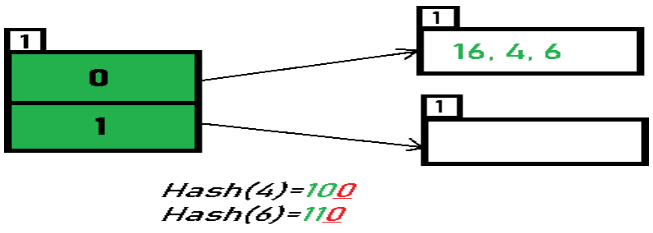
- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001
  20- 10100
  26- 11010
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



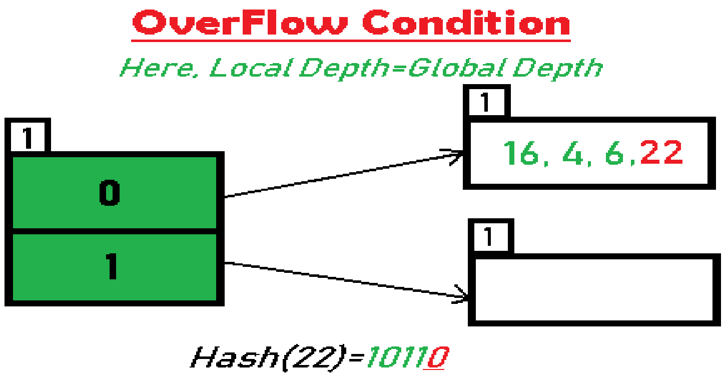
- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSBof 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



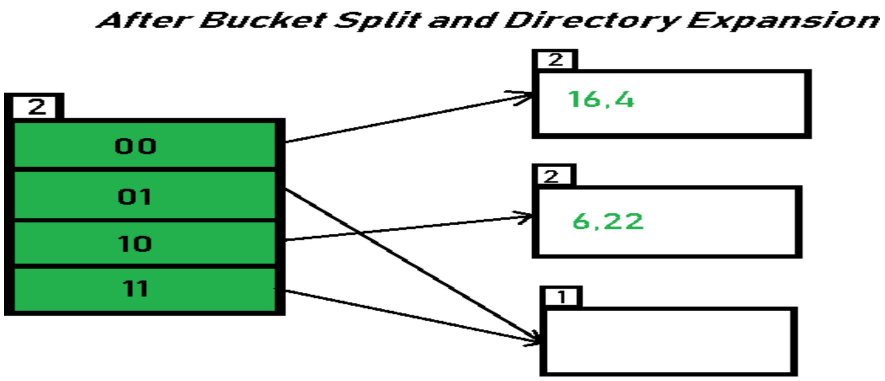
- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows



- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

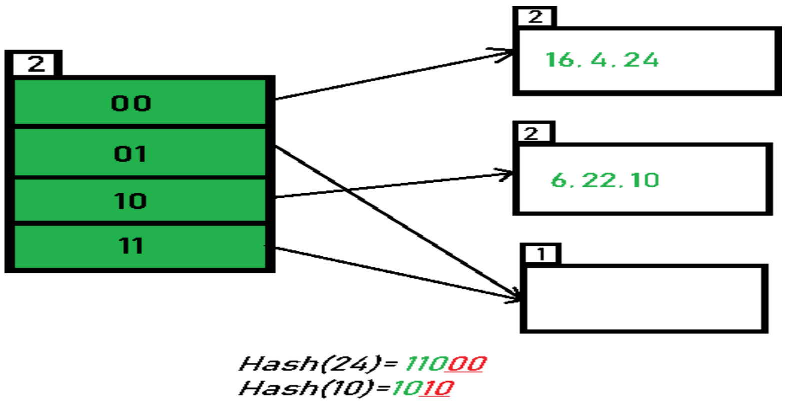


- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]

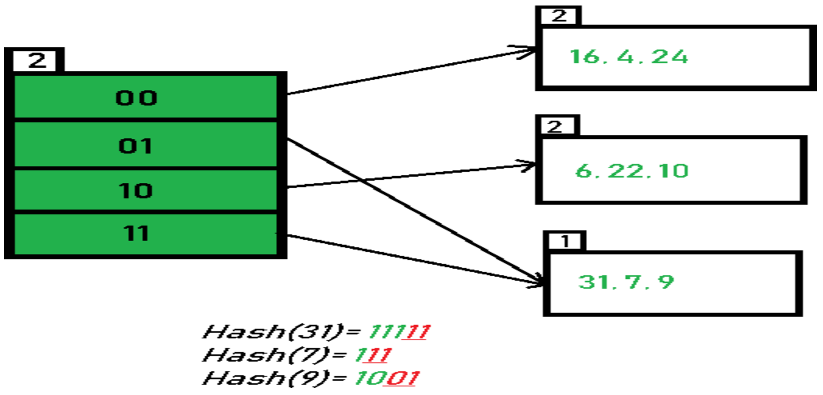


**Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. Thisis because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*

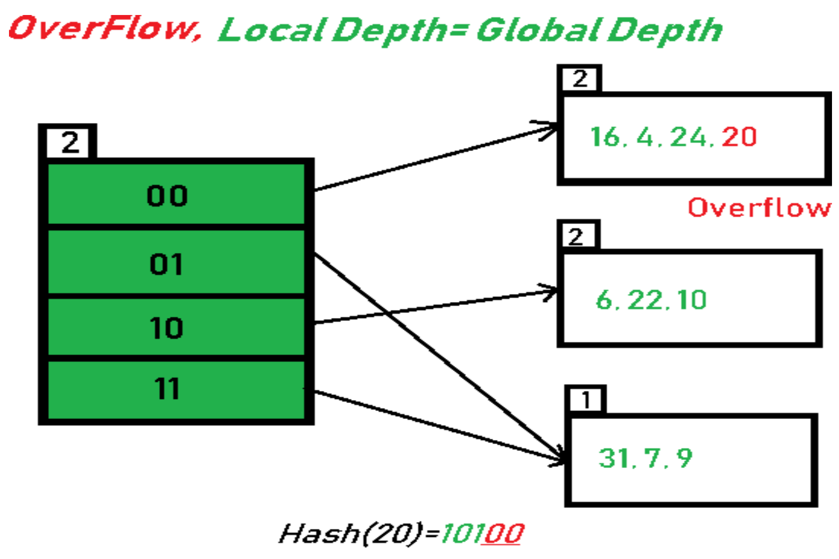
- **Inserting 24 and 10:** 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



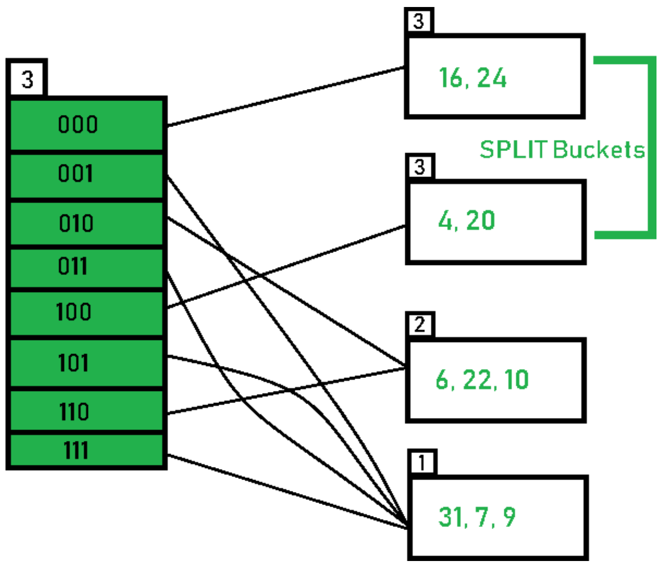
- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We donot encounter any overflow condition here.



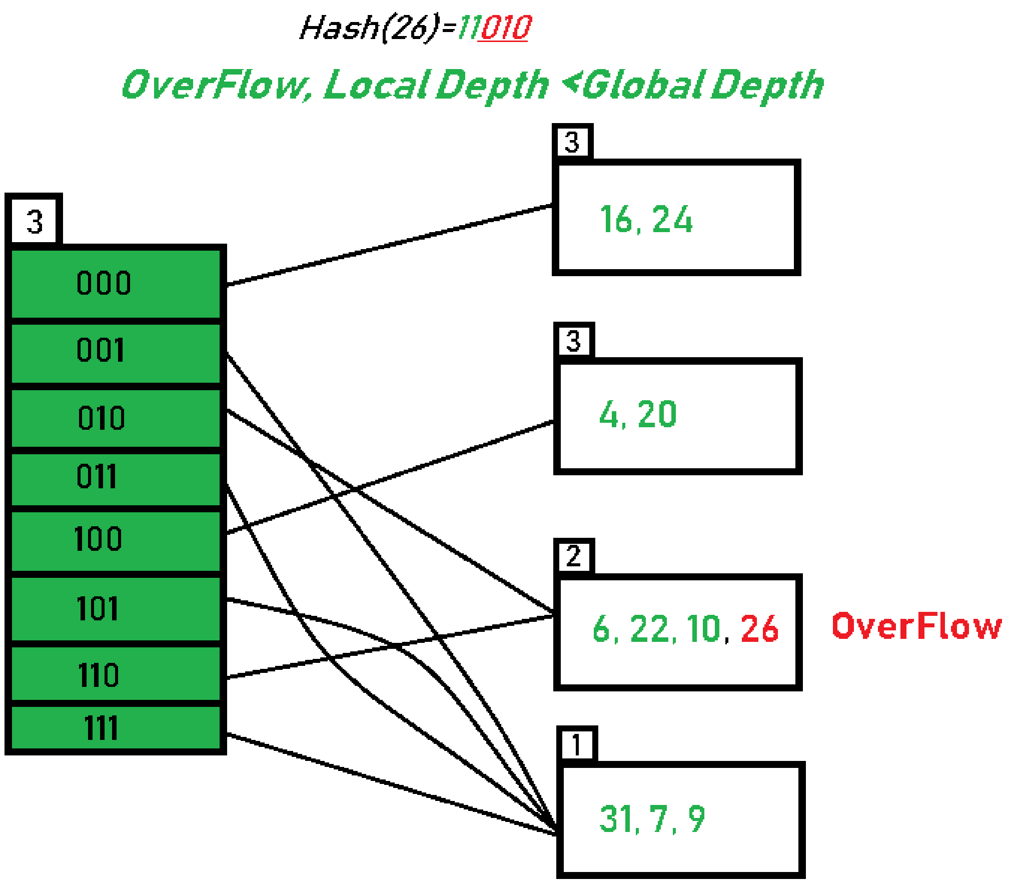
- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



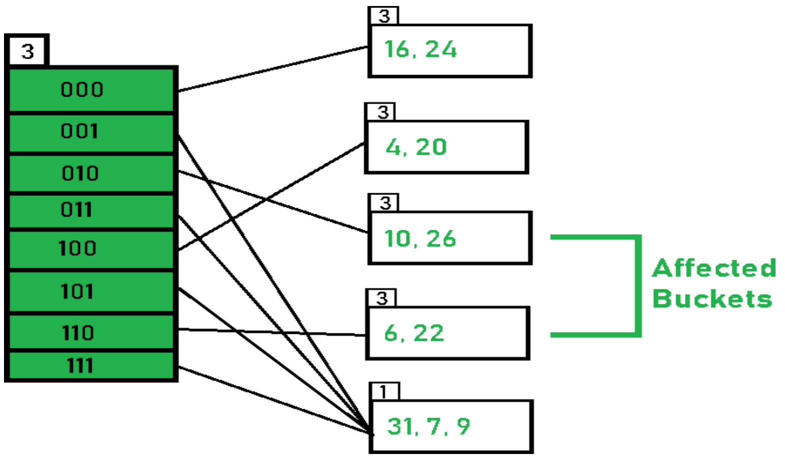
- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.



Finally, the output of hashing the given list of numbers is obtained.

Key Observations:

1. A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
2. When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
3. If Local Depth of the overflowing bucket
4. The size of a bucket cannot be changed after the data insertion process begins.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.

lOMoARcPSD|16210324

UNIT III

UNIT III

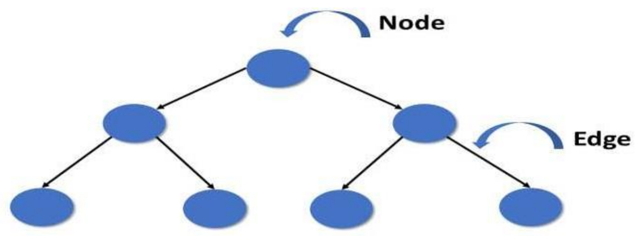
Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Red –Black, Splay Trees.

TREES INTRODUCTION

The tree is a nonlinear hierarchical data structure and comprises a collection of entities known as nodes. It connects each node in the tree data structure using "edges”, both directed and undirected.

The image below represents the tree data structure. The blue-colored circles depict the nodes of the tree and the black lines connecting each node with another are called edges.

You will understand the parts of trees better, in the terminologies section.



The Necessity for a Tree in Data Structures

Other data structures like arrays, linked-list, stacks, and queues are linear data structures, and all these data structures store data in sequential order. Time complexity increases with increasing data size to perform operations like insertion and deletion on these linear data structures. But it isnot acceptable for today's world of computation.

The non-linear structure of trees enhances the data storing, data accessing, and manipulation processes by employing advanced control methods traversal through it. You will learn about tree traversal in the upcoming section.

Tree Terminologies

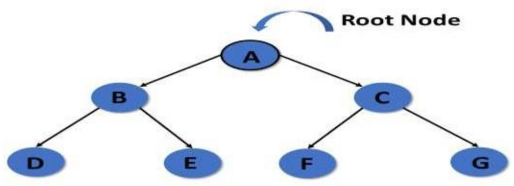
The following are some of the basic tree terms

- Root Node
- Edge
- Parent node
- Child node

- Siblings
- Leaf nodes or external nodes
- Internal nodes
- Degree
- Level
- Height
- Depth
- Path
- Subtree

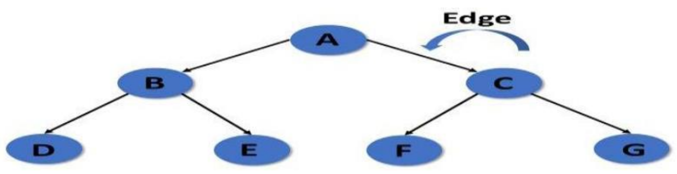
Root

- In a tree data structure, the root is the first node of the tree. The root node is the initial node of the tree in data structures.
- In the tree data structure, there must be only one root node.



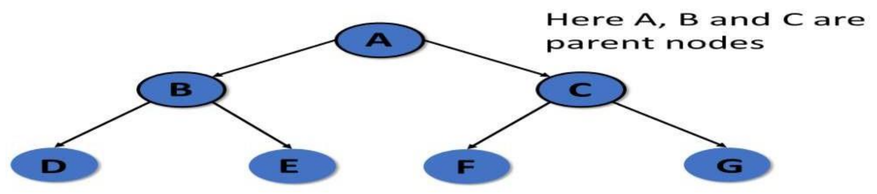
Edge

- In a tree in data structures, the connecting link of any two nodes is called the edge of the tree data structure.
- In the tree data structure, N number of nodes connecting with N -1 number of edges.



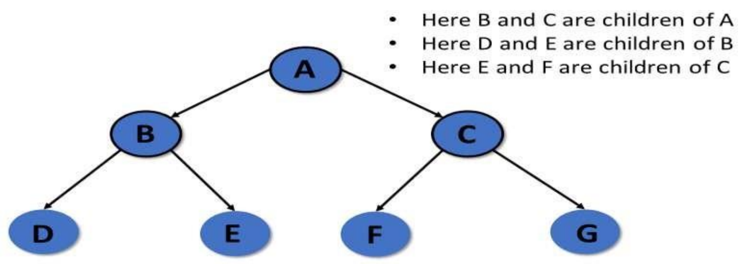
Parent

In the tree in data structures, the node that is the predecessor of any node is known as a parent node, or a node with a branch from itself to any other successive node is called the parent node.



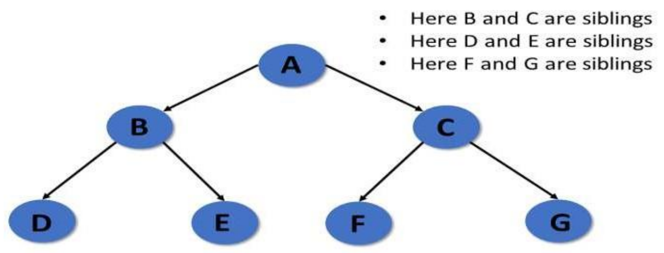
Child

- The node, a descendant of any node, is known as child nodes in data structures.
- In a tree, any number of parent nodes can have any number of child nodes.
- In a tree, every node except the root node is a child node.


Here B and C are children of A
Here D and E are children of B
Here E and F are children of C
A
B
C
D
E
F
G

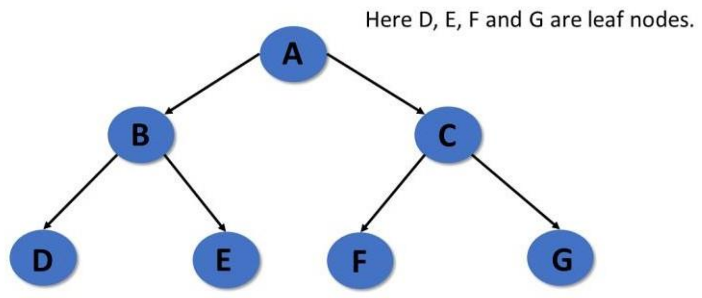
Siblings

In trees in the data structure, nodes that belong to the same parent are called siblings.


Here B and C are siblings
Here D and E are siblings
Here F and G are siblings
A
B
C
D
E
F
G

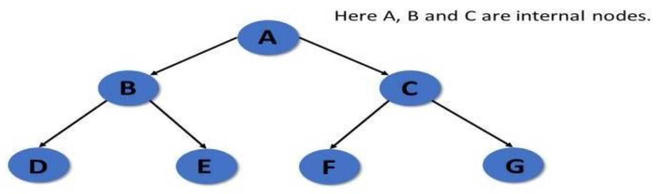
Leaf

- Trees in the data structure, the node with no child, is known as a leaf node.
- In trees, leaf nodes are also called external nodes or terminal nodes.



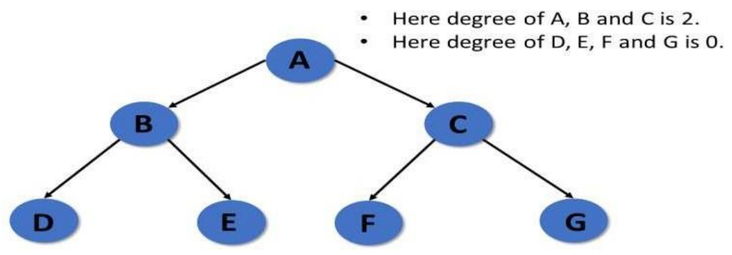
Internal nodes

- Trees in the data structure have at least one child node known as internal nodes.
- In trees, nodes other than leaf nodes are internal nodes.
- Sometimes root nodes are also called internal nodes if the tree has more than one node.



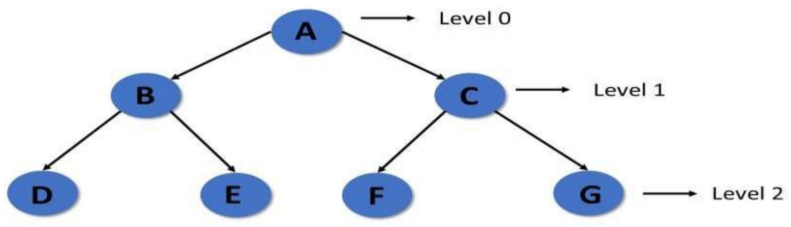
Degree

- In the tree data structure, the total number of children of a node is called the degree of the node.
- The highest degree of the node among all the nodes in a tree is called the Degree of Tree.


Here degree of A, B and C is 2.
Here degree of D, E, F and G is 0.
A
B
C
D
E
F
G

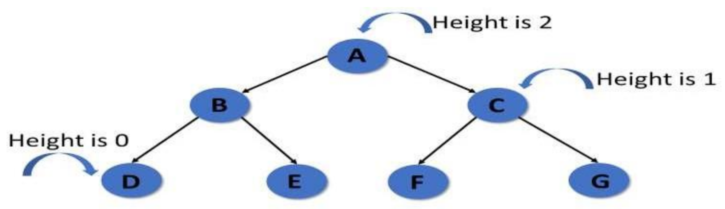
Level

In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.



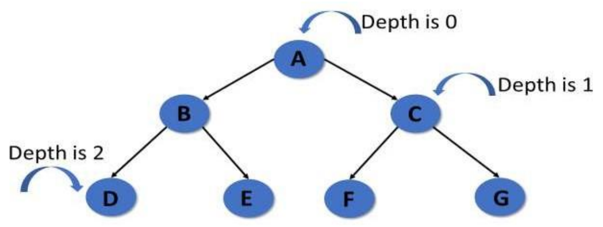
Height

- In a tree data structure, the number of edges from the leaf node to the particular node in the longest path is known as the height of that node.
- In the tree, the height of the root node is called "Height of Tree".
- The tree height of all leaf nodes is 0.



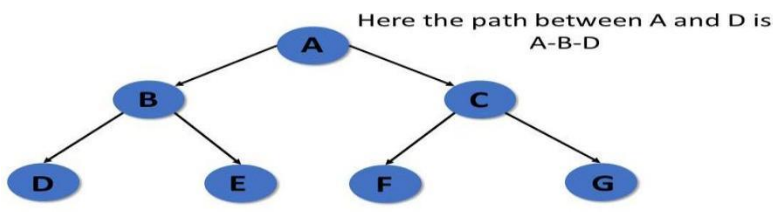
Depth

- In a tree, many edges from the root node to the particular node are called the depth of the tree.
- In the tree, the total number of edges from the root node to the leaf node in the longest path is known as "Depth of Tree".
- In the tree data structures, the depth of the root node is 0.



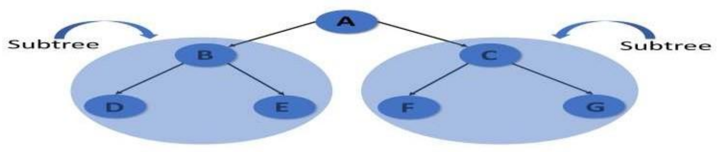
Path

- In the tree in data structures, the sequence of nodes and edges from one node to another node is called the path between those two nodes.
- The length of a path is the total number of nodes in a path.zx



Subtree

In the tree in data structures, each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node.

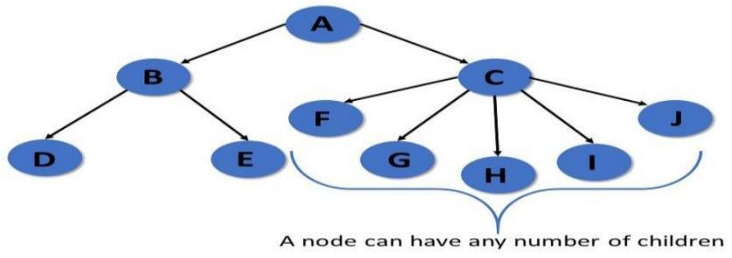


General Tree

The general tree is the type of tree where there are no constraints on the hierarchical structure.

Properties

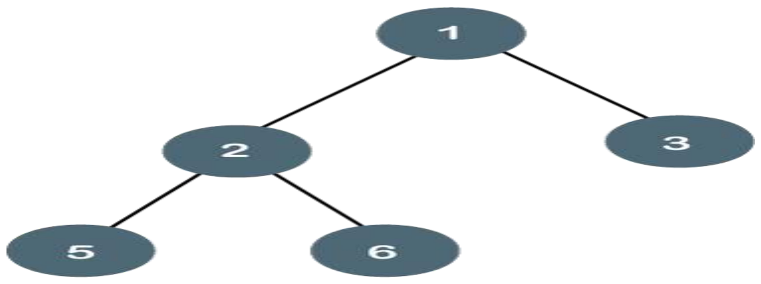
- The general tree follows all properties of the tree data structure.
- A node can have any number of nodes.



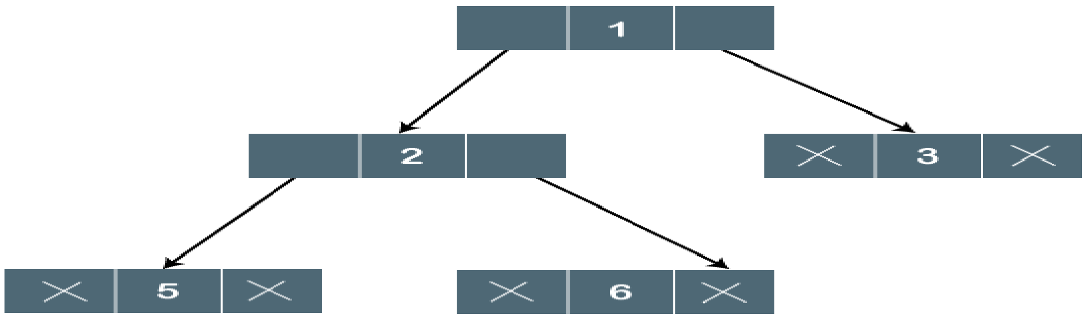
BINARY TREES

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots . 2^h) = 2^{h+1} - 1$.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$\mathbf{h = \log_2(n+1) - 1}$

The maximum height can be computed as:

As we know that,

n = h+1

h= n-1

Types of Binary Tree

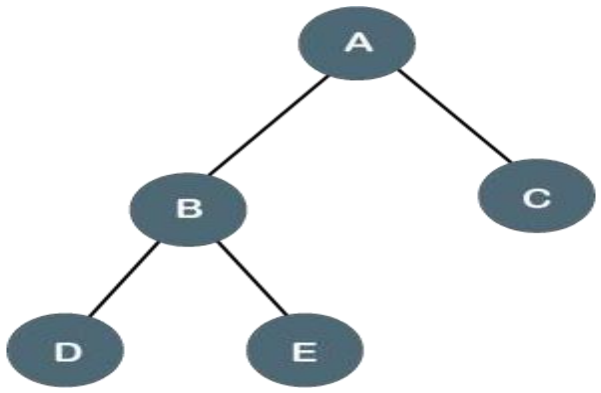
There are four types of Binary tree:

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to

6.

- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

  n= 2*h - 1

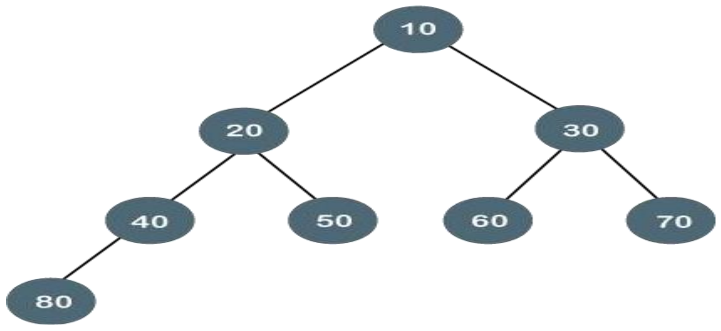
  n+1 = 2*h

  h = n+1/2

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



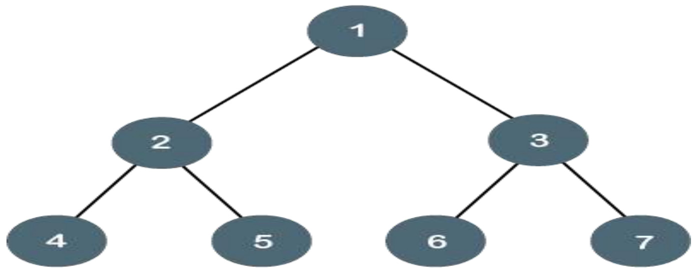
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is 2^{h+1} - 1.
- The minimum number of nodes in complete binary tree is 2^h.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

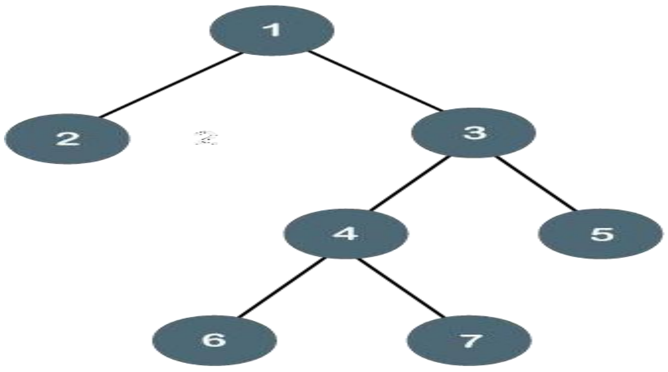
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

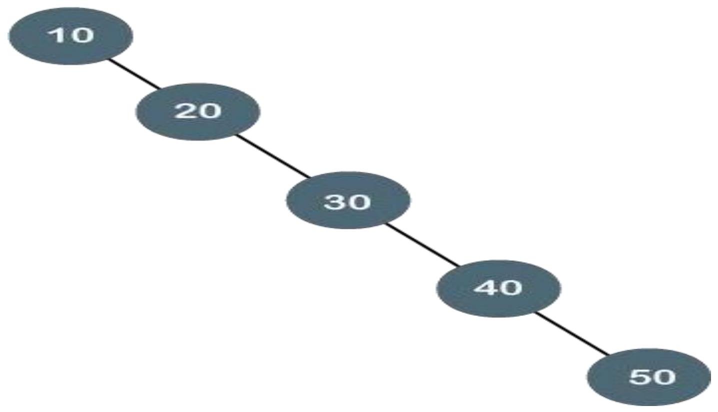
The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.



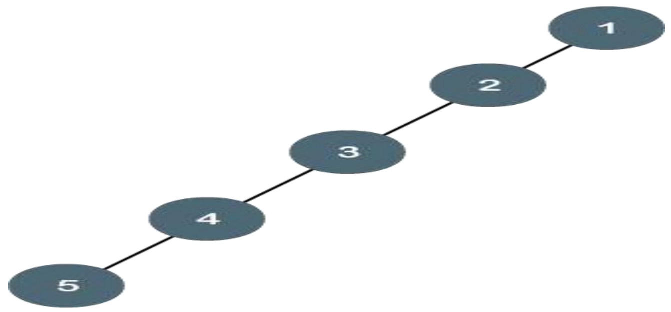
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

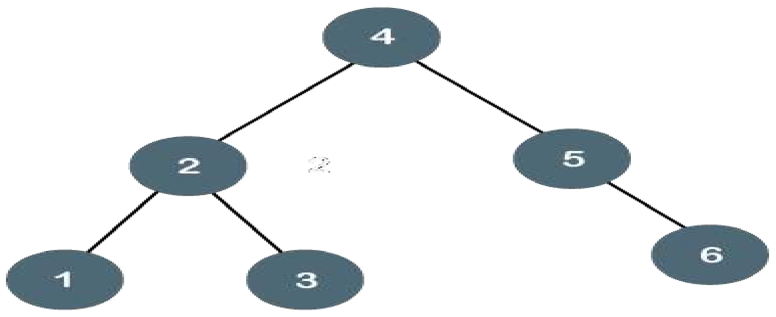


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

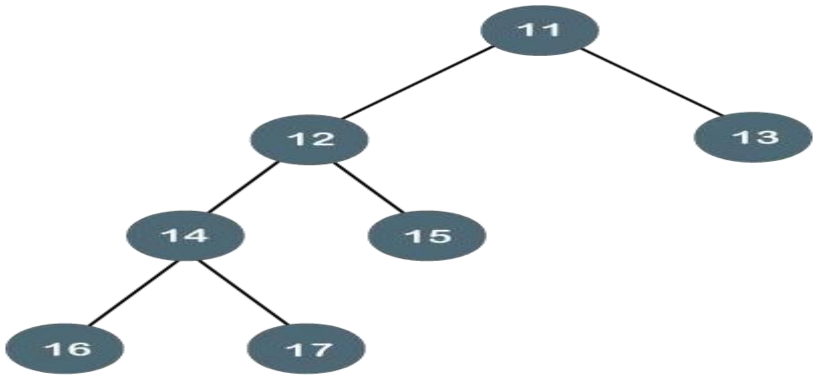
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees height differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the height of left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the height of left subtree and the right subtree is greater than 1.

Tree Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees**.**

Depth First Traversals:

(a) Inorder (Left, Root, Right)
(b) Preorder (Root, Left, Right)
(c) Postorder (Left, Right, Root)

Breadth-First or Level Order Traversal

Let see each traversal method with example.

Inorder Traversal

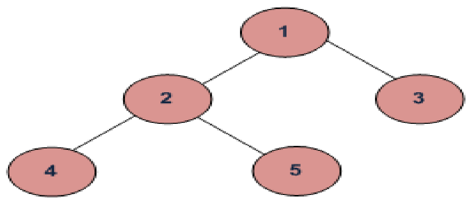
Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Example:



In order traversal for the above-given figure is 4 2 5 1 3.

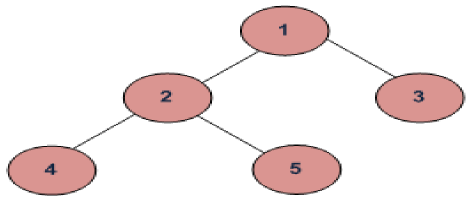
Preorder Traversal

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on an expression tree.



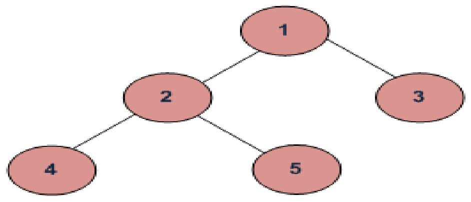
Example: Preorder traversal for the above-given figure is 1 2 4 5 3.

Postorder Traversal

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

EXAMPLE



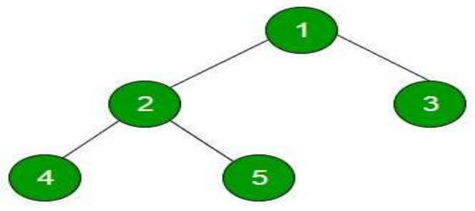
Example: Postorder traversal for the above-given figure is 4 5 2 3 1.

Uses of Postorder

Postorder traversal is also useful to get the postfix expression of an expression tree.

Level Order Binary Tree Traversal

Level order traversal of a tree is breadth first traversal for the tree.



Level order traversal of the above tree is 1 2 3 4 5

Construct a binary tree from inorder and postorder traversals

The idea is to start with the root node, which would be the last item in the postorder sequence, and find the boundary of its left and right subtree in the inorder sequence. To find the boundary, search for the index of the root node in the inorder sequence. All keys before the root node in theinorder sequence become part of the left subtree, and all keys after the root node become part of the right subtree. Repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider the following inorder and postorder sequence:

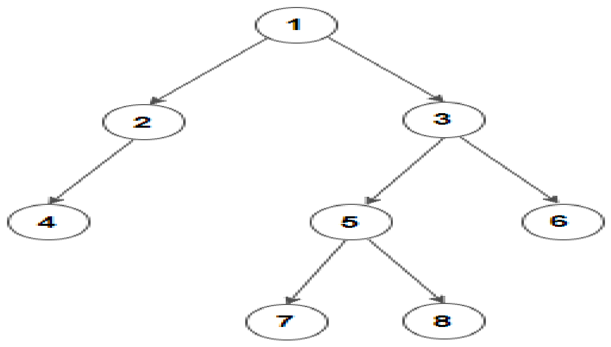
Inorder : { 4, 2, 1, 7, 5, 8, 3, 6 }
Postorder : { 4, 2, 7, 8, 5, 6, 3, 1 }

Root would be the last element in the postorder sequence, i.e., 1. Next, locate the index of the root node in the inorder sequence. Now since 1 is the root node, all nodes before 1 in the inorder sequence must be included in the left subtree of the root node, i.e., {4, 2} and all the nodes after 1 must be included in the right subtree, i.e., {7, 5, 8, 3, 6}. Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Left subtree:
Inorder : {4, 2}
Postorder : {4, 2}
Right subtree:
Inorder : {7, 5, 8, 3, 6}
Postorder : {7, 8, 5, 6, 3}

The idea is to recursively follow the above approach until the complete tree is constructed.

The final tree will be

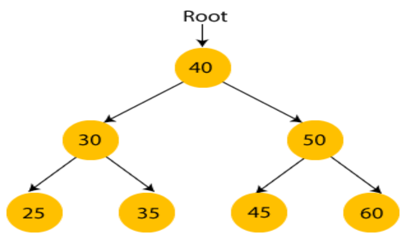


Binary Search Tree(BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

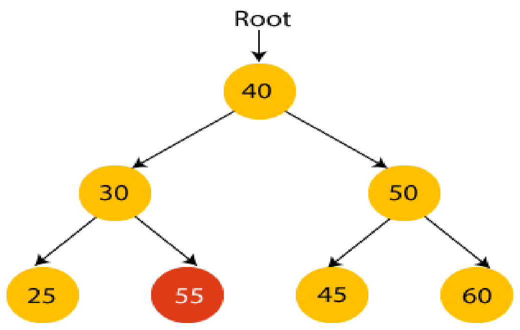
let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a Binary Search Tree (BST)

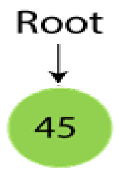
Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

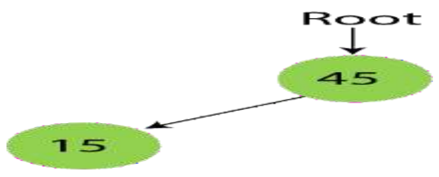
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.



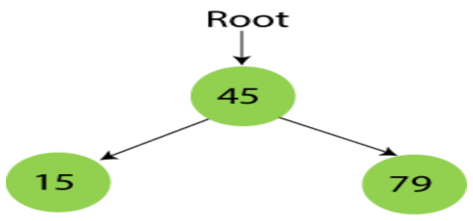
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



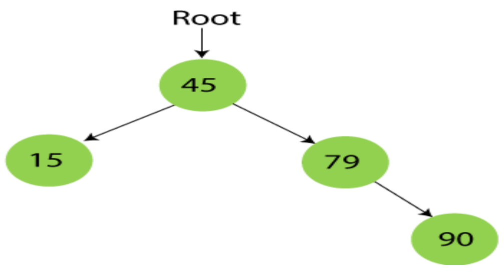
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



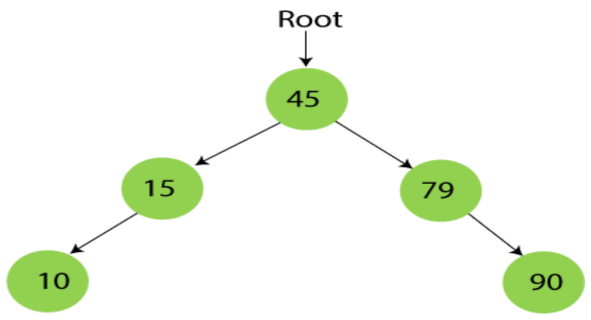
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



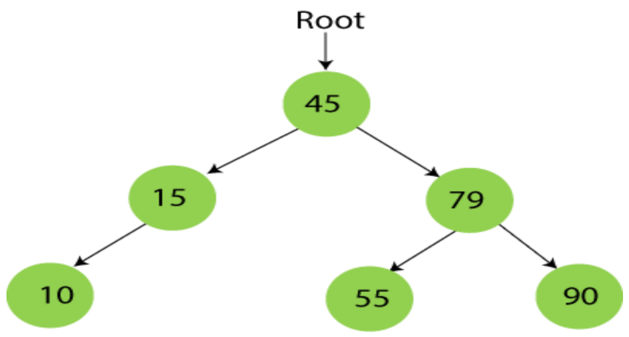
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



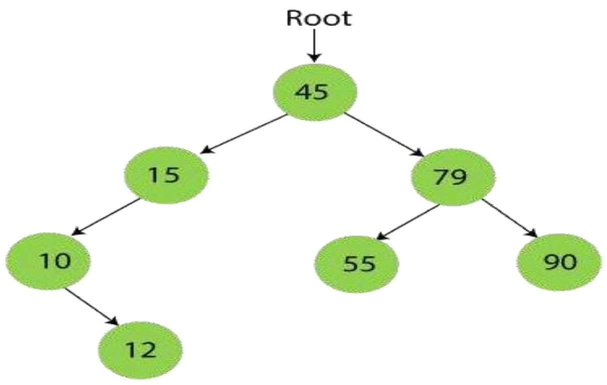
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



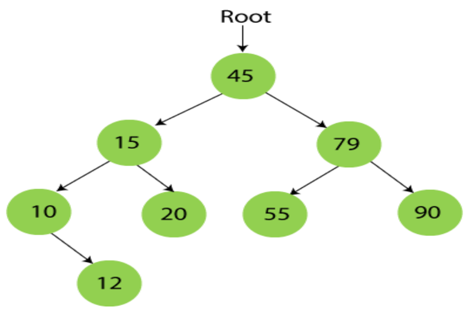
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



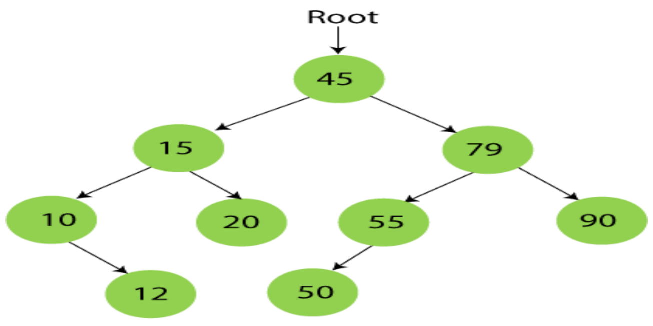
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

We can perform insert, delete and search operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

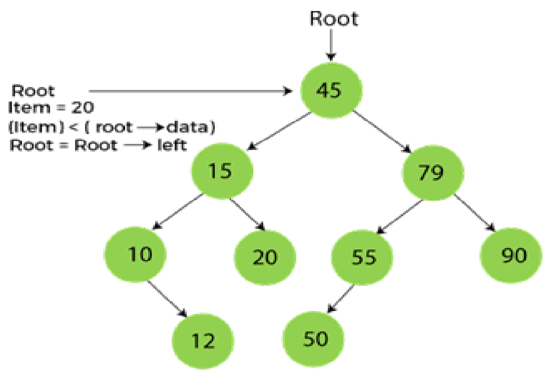
Searching in Binary search tree(BST)

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

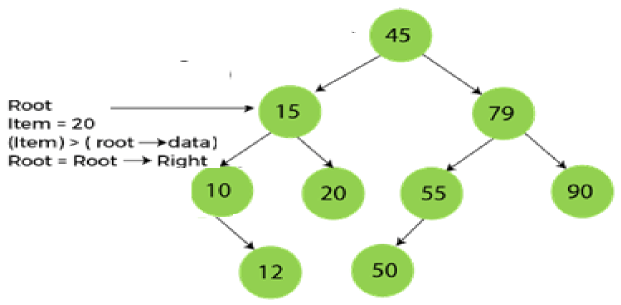
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

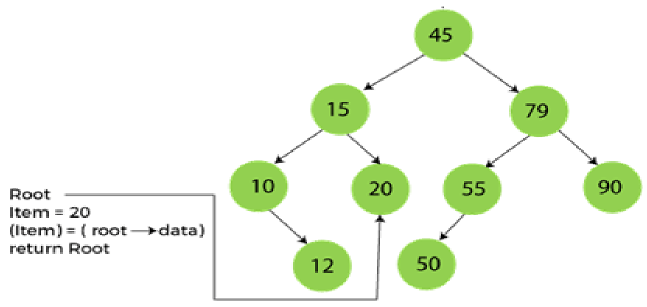
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.

Deletion in Binary Search tree(BST)

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

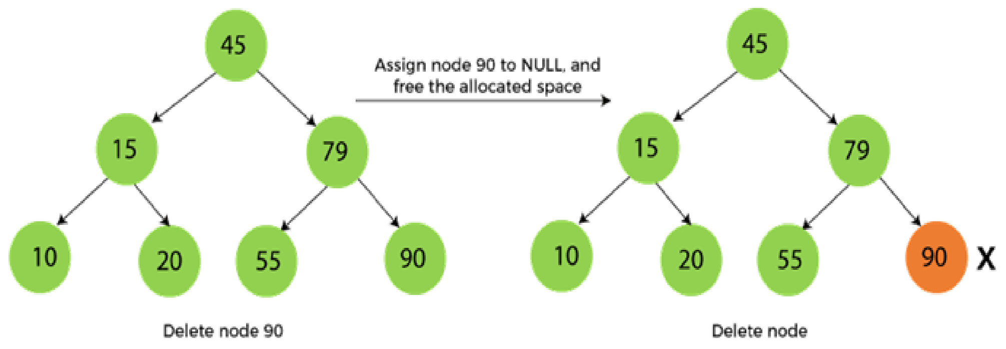
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

We will understand the situations listed above in detail.

When the node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



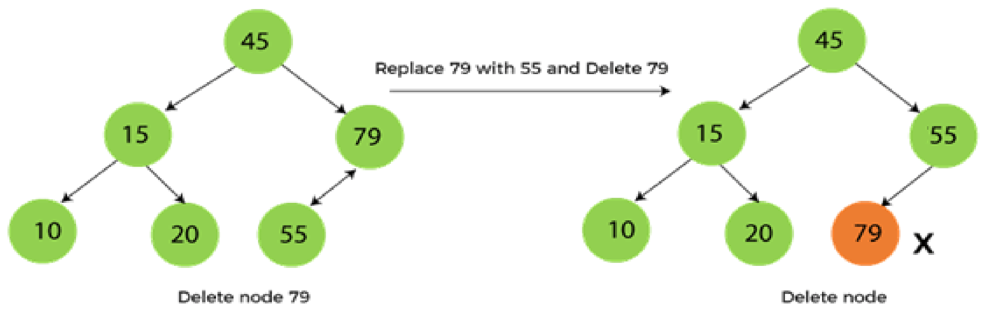
When the node to be deleted has only one child

In this case, we have to replace the target node(Deleting node) with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image.

In the below image, suppose we have to delete the node 79, as the node to be deleted has onlyone child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.



When the node to be deleted has two children

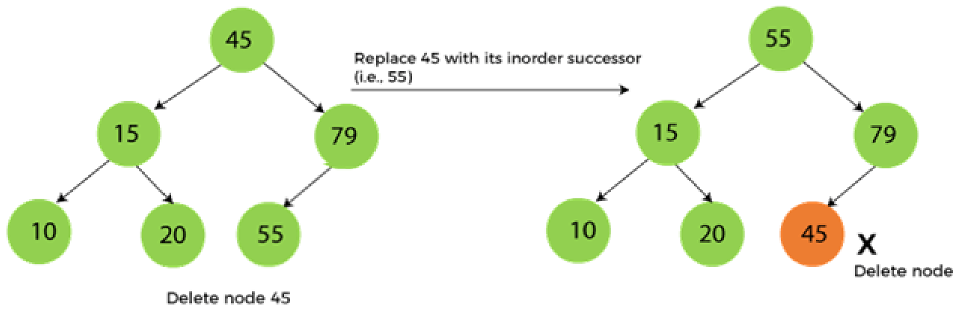
This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.

- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

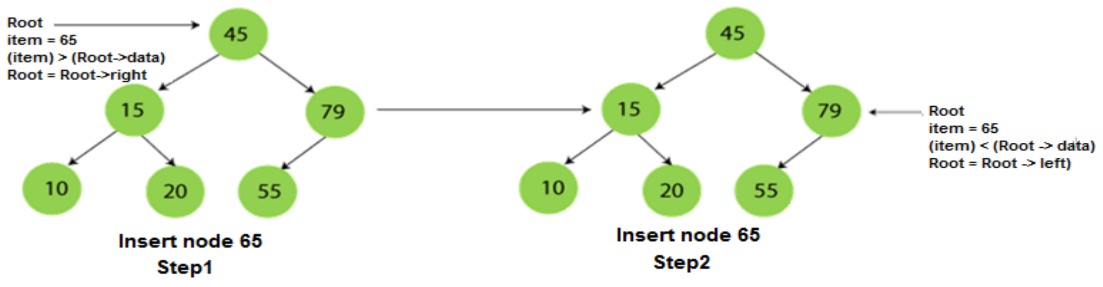


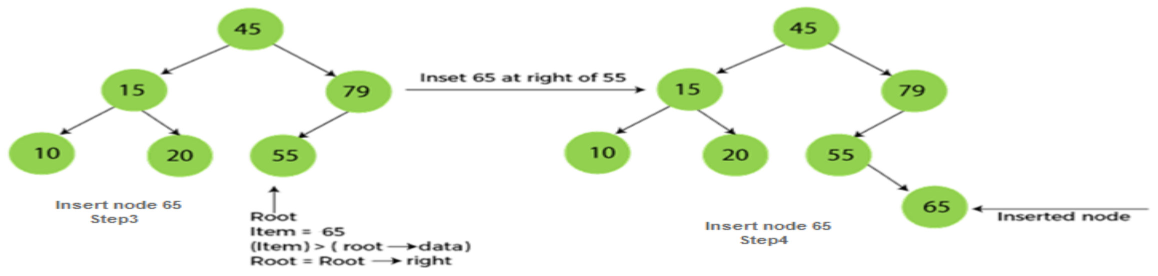
Now let's understand how insertion is performed on a binary search tree.

Insertion in Binary Search tree(BST)

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

Now, let's see the process of inserting a node into BST using an example.





The complexity of the Binary Search tree
Let's see the time and space complexity of the Binary search tree. We will see the time complexity for insertion, deletion, and searching operations in best case, average case, and worst case.

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Worst case scenario indicates the BST is the Degenerated BST for all the operations (insertion, deletion and search)

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

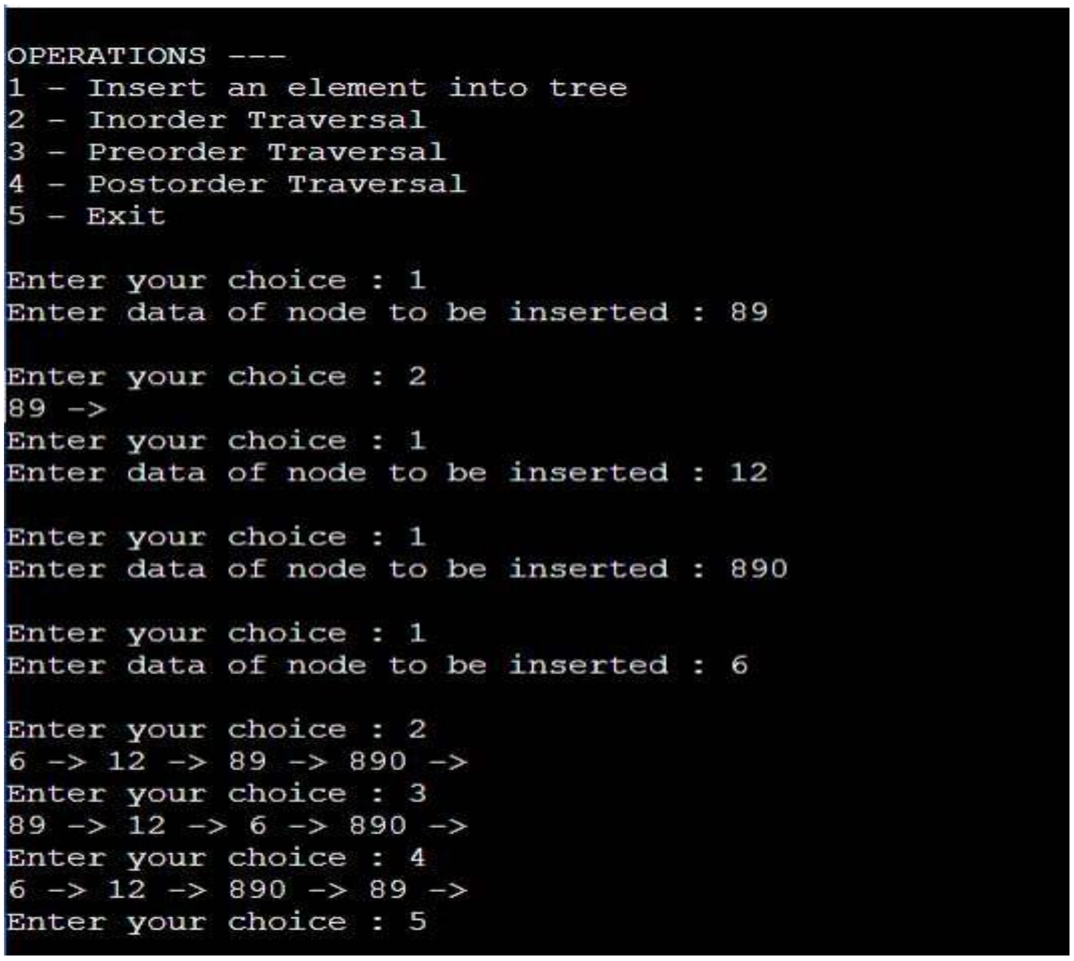
- The space complexity of all operations of Binary search tree is O(n).

Implementation of binary Search Tree traversal method

Program:

```
#include  <stdio.h>
#include <stdlib.h>
struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;
void insert();
void create();
void search( struct btnode *root);
void inorder(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void main()
{
    int ch;
    printf("\nOPERATIONS ---");
    printf("\n1 - Insert an element into tree\n");
    printf("2 - Inorder Traversal\n");
    printf("3 - Preorder Traversal\n");
    printf("4 - Postorder Traversal\n");
    printf("5 - Exit\n");
    while(1)
    {
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            insert();
            break;
        case 2:
            inorder(root);
            break;
        case 3:
            preorder(root);
            break;
        case 4:
            postorder(root);
            break;
        case 5:
            exit(0);
        default :
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}
/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}
/* To create a node */
void create()
{
    int data;
    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}
/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))      /* value more than root node value insert
at right */
        search(t->r);
    else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != NULL))      /* value less than root node value
insert at left */
        search(t->l);
    else if ((temp->value < t->value) && (t->l == NULL))
        t->l = temp;
}
/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}
/* To find the preorder traversal */
void preorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    printf("%d -> ", t->value);
    if (t->l != NULL)
        preorder(t->l);
    if (t->r != NULL)
        preorder(t->r);
}
 /* To find the postorder traversal */
void postorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display ");
        return;
    }
    if (t->l != NULL)
        postorder(t->l);
    if (t->r != NULL)
        postorder(t->r);
    printf("%d -> ", t->value);
}
```

OUTPUT:



```
OPERATIONS ---
1 - Insert an element into tree
2 - Inorder Traversal
3 - Preorder Traversal
4 - Postorder Traversal
5 - Exit

Enter your choice : 1
Enter data of node to be inserted : 89

Enter your choice : 2
89 ->
Enter your choice : 1
Enter data of node to be inserted : 12

Enter your choice : 1
Enter data of node to be inserted : 890

Enter your choice : 1
Enter data of node to be inserted : 6

Enter your choice : 2
6 -> 12 -> 89 -> 890 ->
Enter your choice : 3
89 -> 12 -> 6 -> 890 ->
Enter your choice : 4
6 -> 12 -> 890 -> 89 ->
Enter your choice : 5
```

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

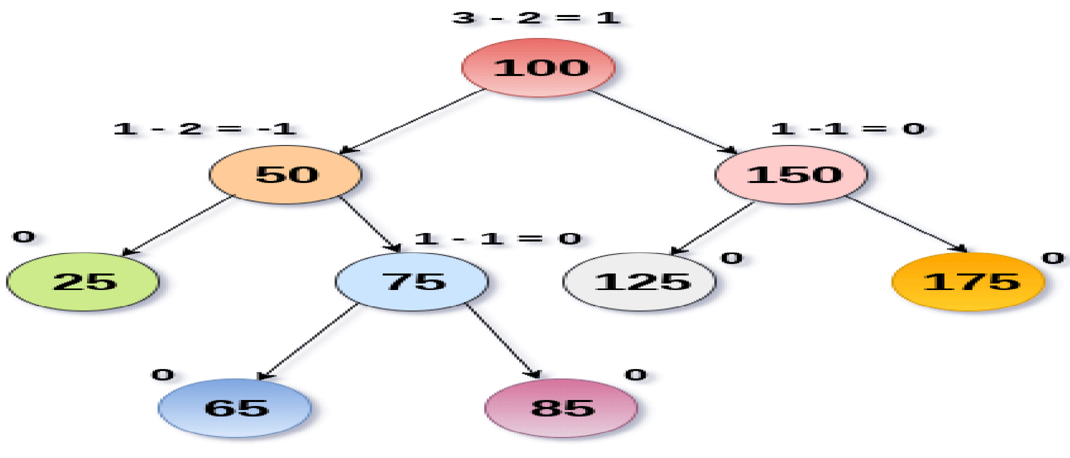
AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extendedto **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

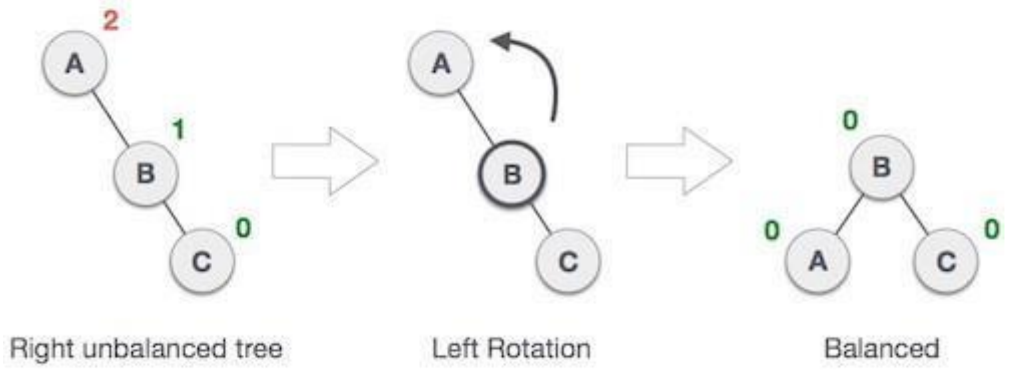
We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.
The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

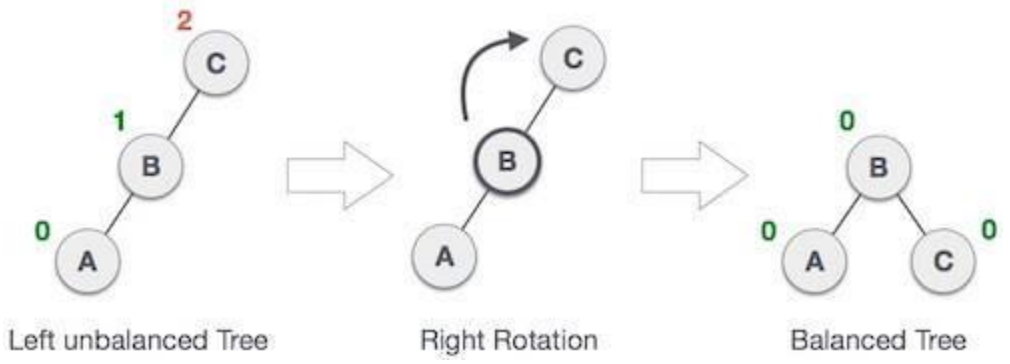
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation
is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation,
LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

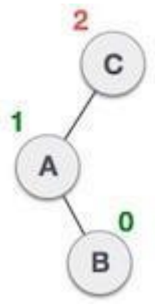
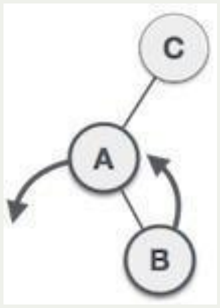
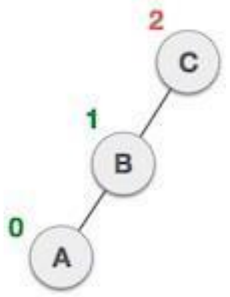


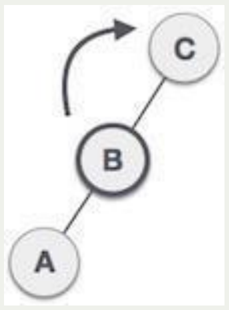
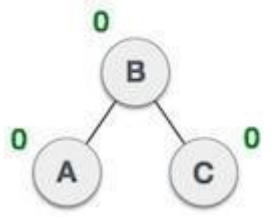
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

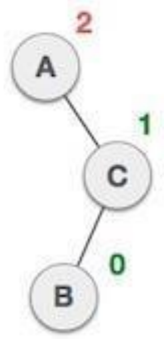
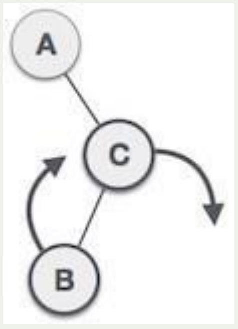
Let us understand each and every step very clearly:

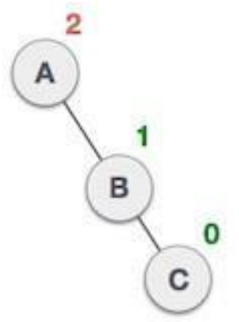
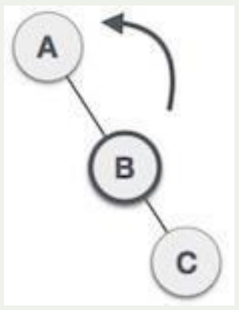
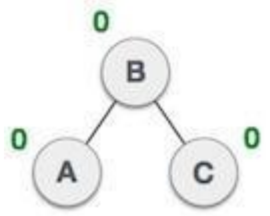
State	**Action**
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**

	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

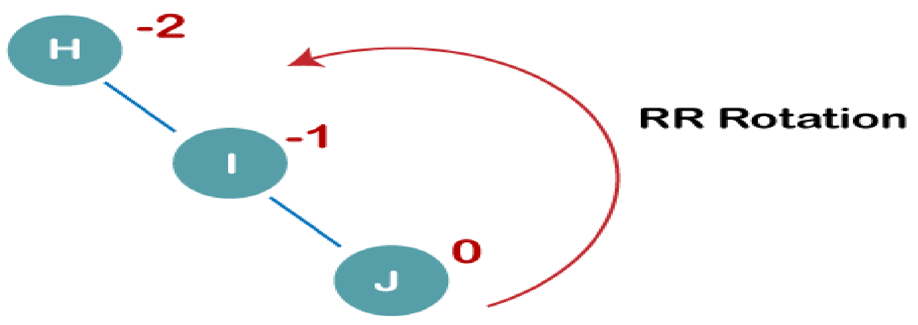
4. RL Rotation

. R L rotation= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	**Action**
	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor -2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.

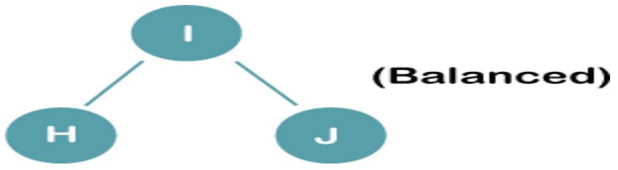
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

AVL TREE CONSTRUCTION
Construct an AVL tree having the following elements
H, I, J, B, A, E, C, F, D, G, K, L
1. Insert H, I, J

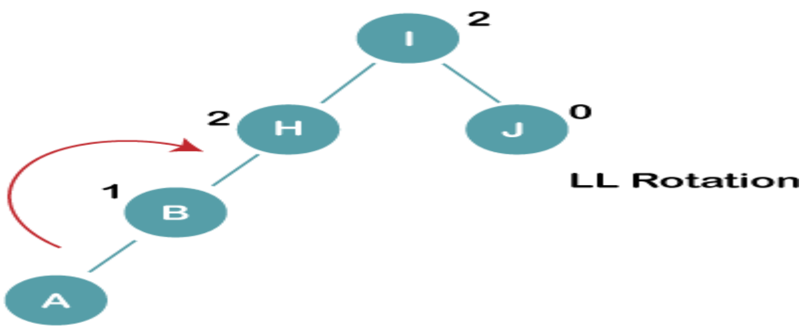


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

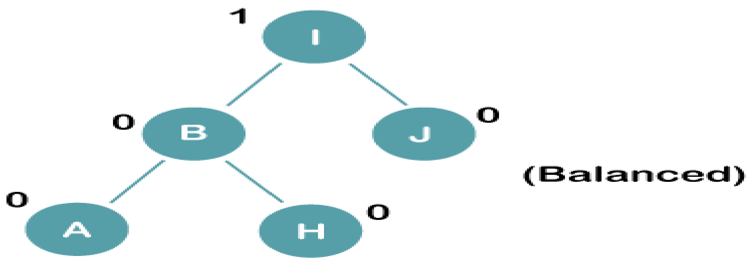
The resultant balance tree is:



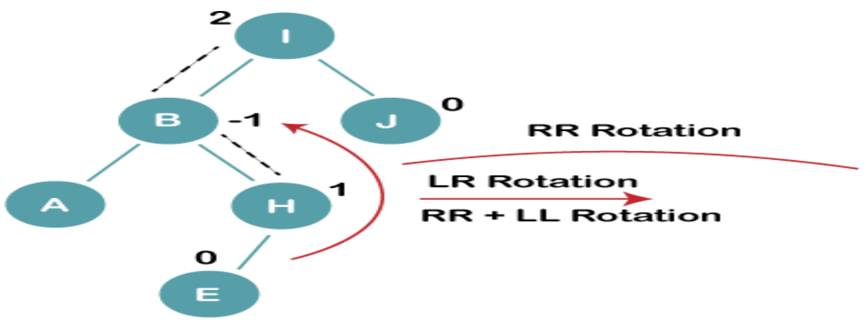
2. Insert B, A



On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.
The resultant balance tree is:



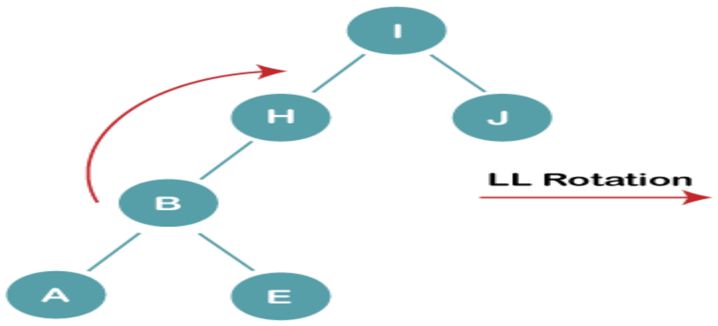
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

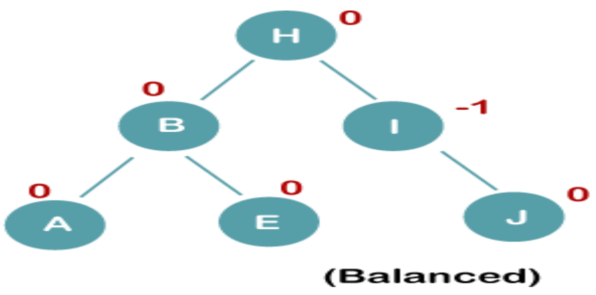
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

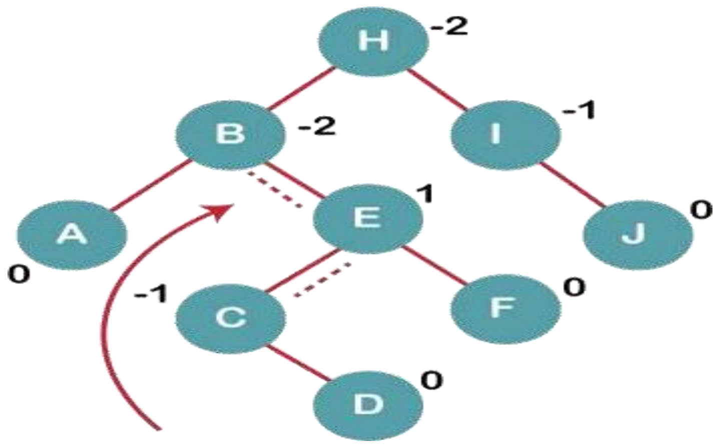


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is**:**



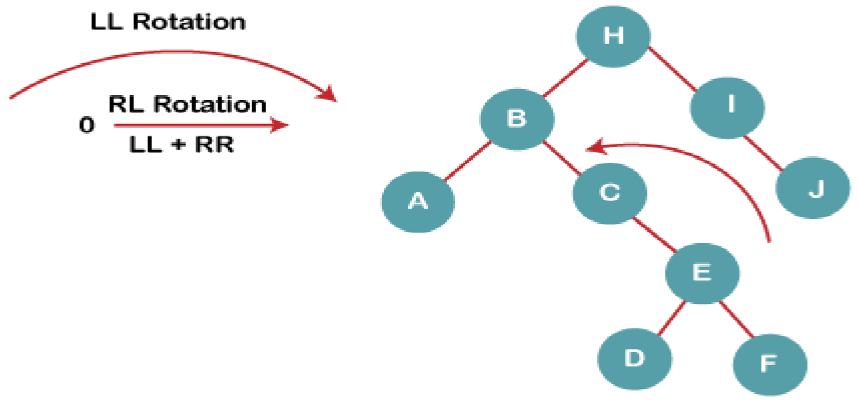
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

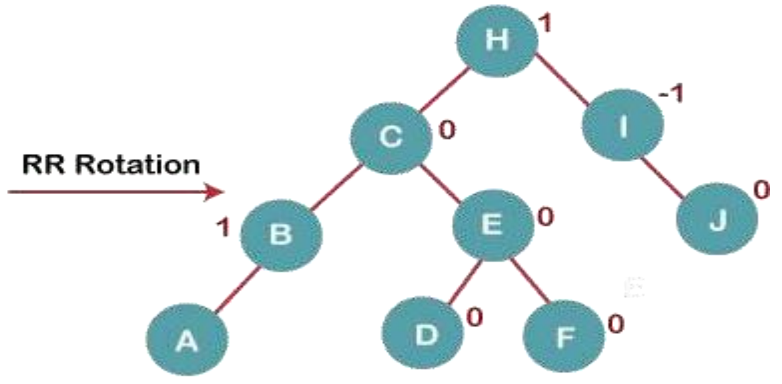
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:



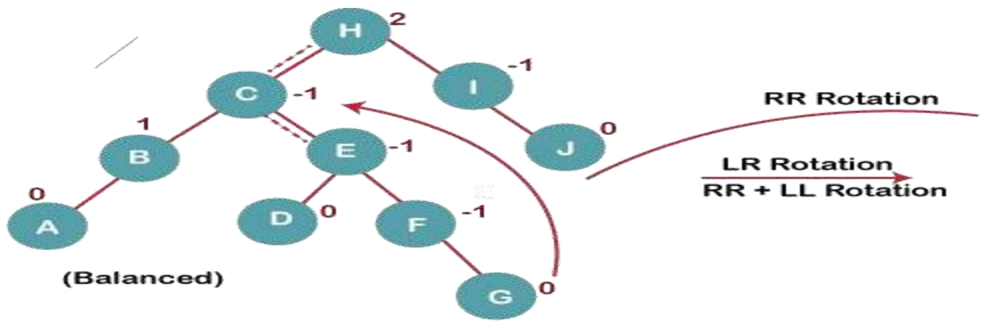
4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



(Balanced)

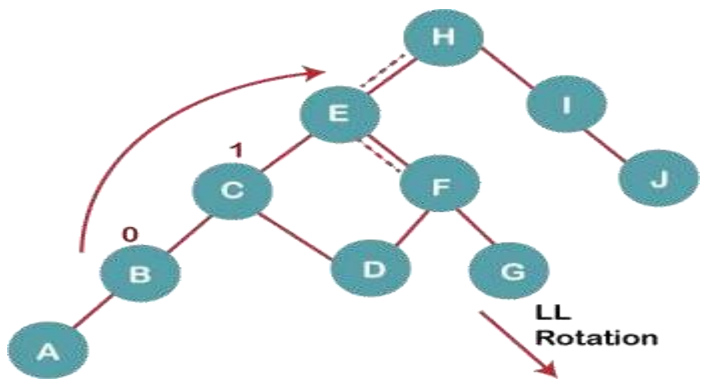
5. Insert G



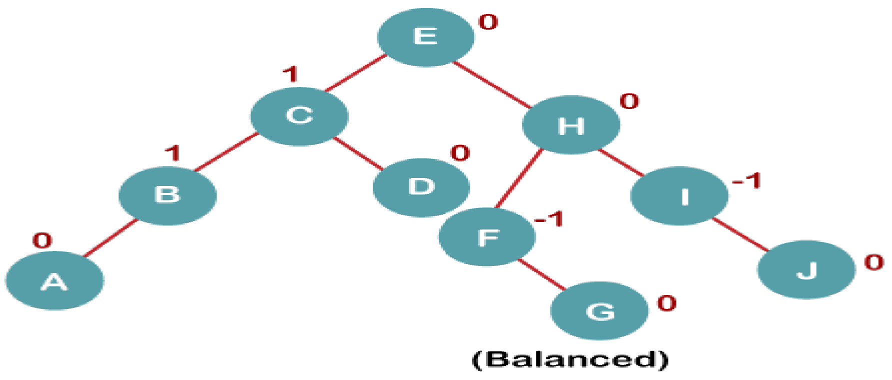
On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from

G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

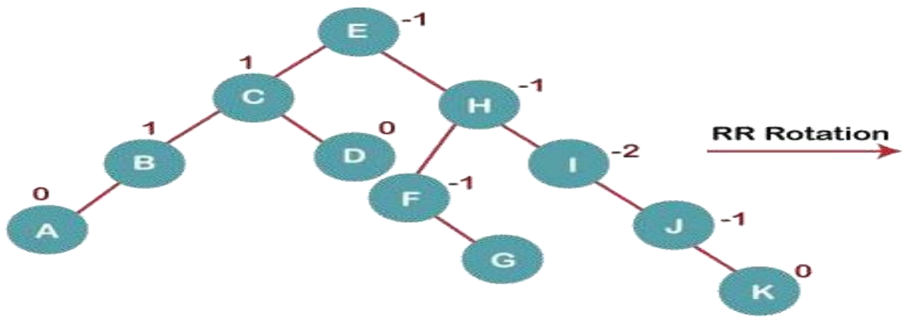
5 a) We first perform RR rotation on node C
The resultant tree after RR rotation is:



5 b) We then perform LL rotation on node H
The resultant balanced tree after LL rotation is:

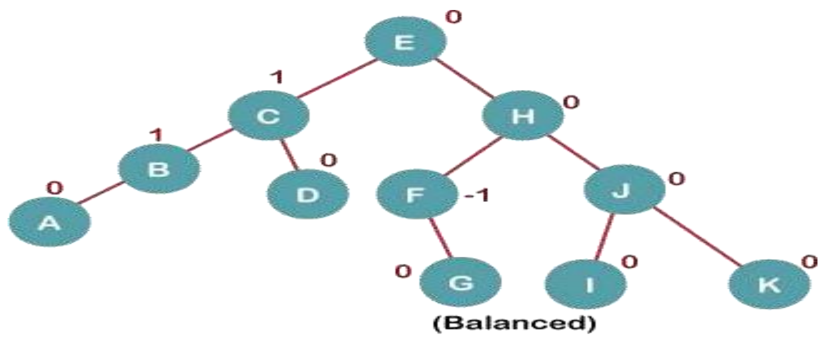


6. Insert K



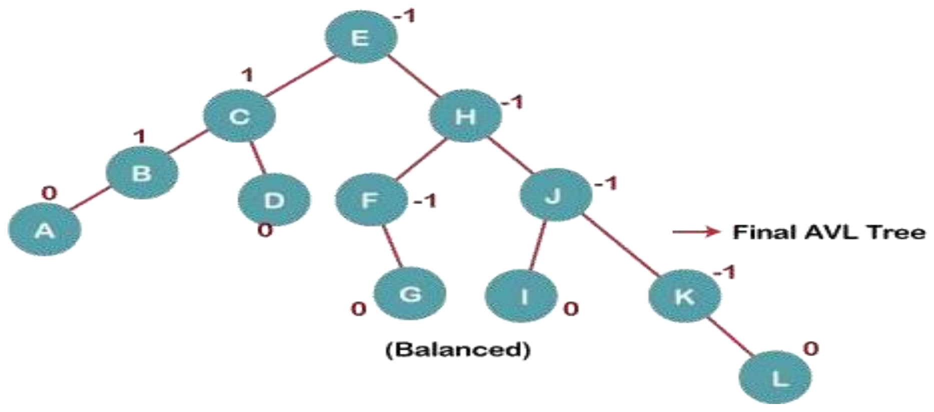
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



Operations on AVL tree

The following operations are performed on AVL Tree

1. Insertion
2. Deletion
3. Search

Insertion Operation on AVL Tree

Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.

The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

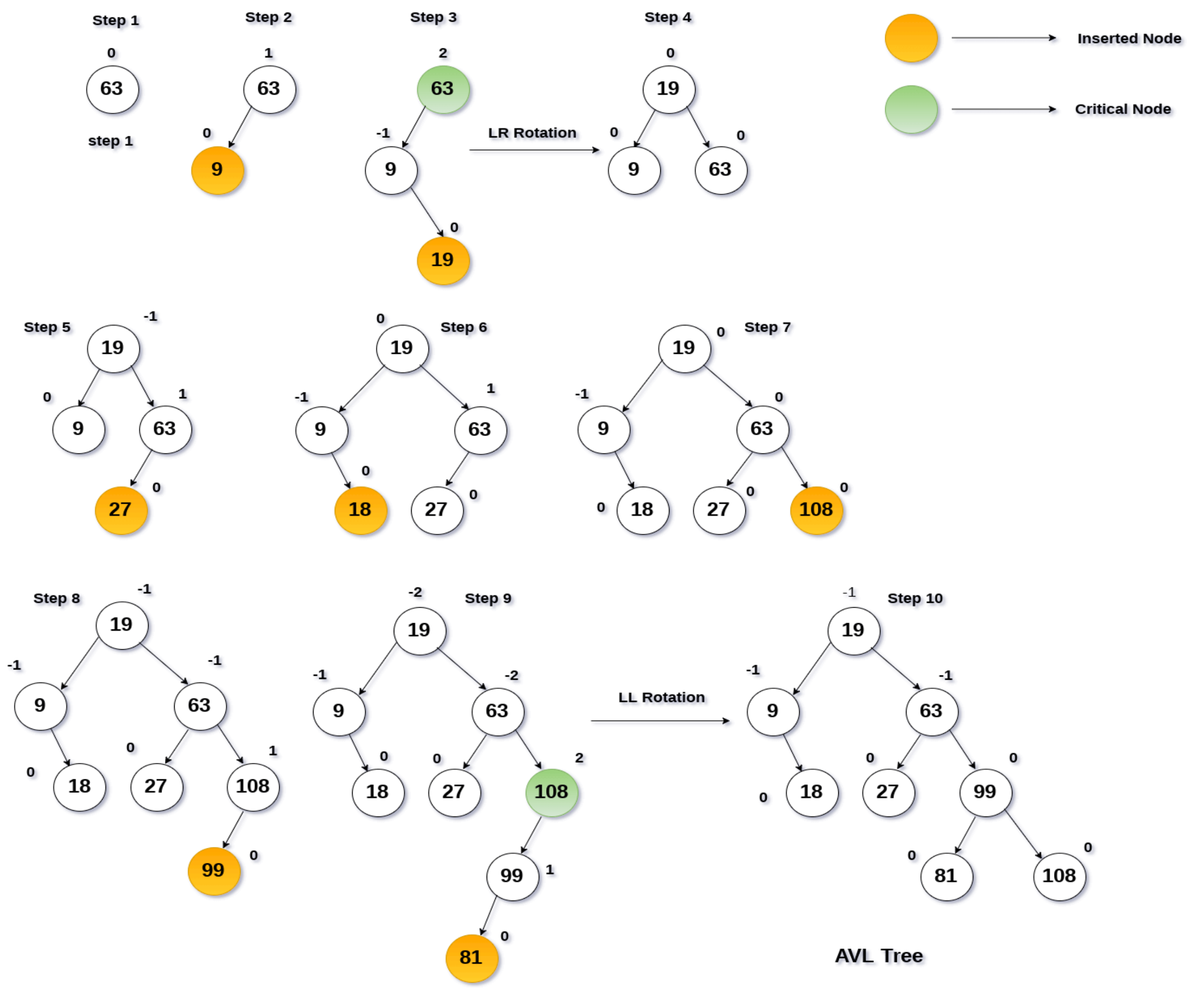
EXAMPLE

Construct an AVL tree by inserting the following elements in the given order.
63, 9, 19, 27, 18, 108, 99, 81

The process of constructing an AVL tree from the given set of elements is shown in thefollowing figure.

At each step, we must calculate the balance factor for every node, if it is found to be more than 2 or less than -2, then we need a rotation to rebalance the tree. The type of rotation will be estimated by the location of the inserted element with respect to the critical node.

All the elements are inserted in order to maintain the order of binary search tree.

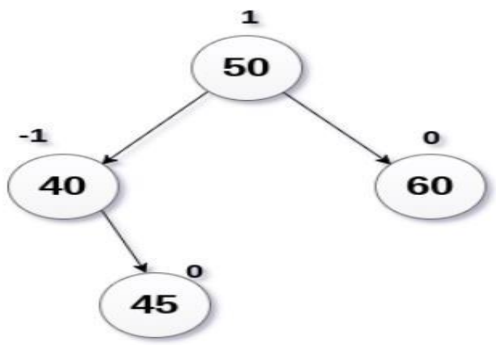


Deletion in AVL Tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations.

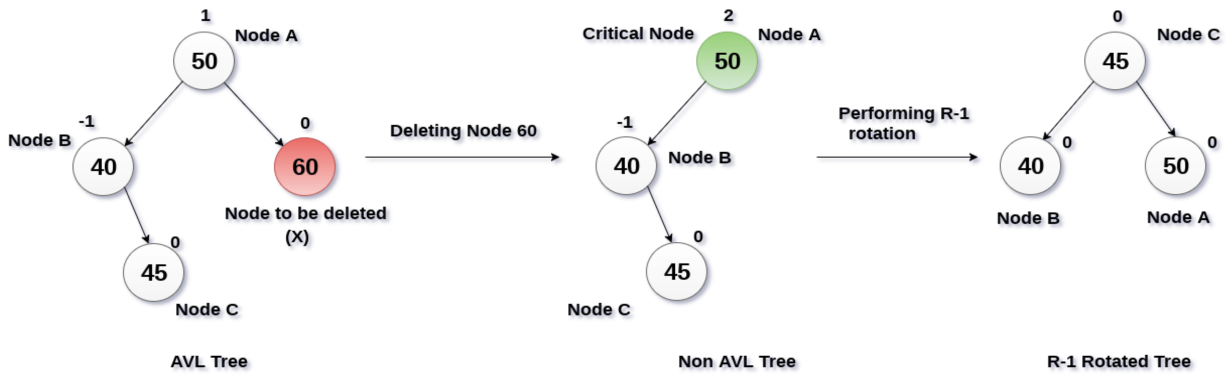
Example

Delete the node 60 from the AVL tree shown in the following image.



Solution:

in this case, node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the node B(40) and A(50) as its left and right child.



Search Operation in AVL Tree

Search operation in AVL tree is similar to binary search tree search operation.

RED-BLACK TREE

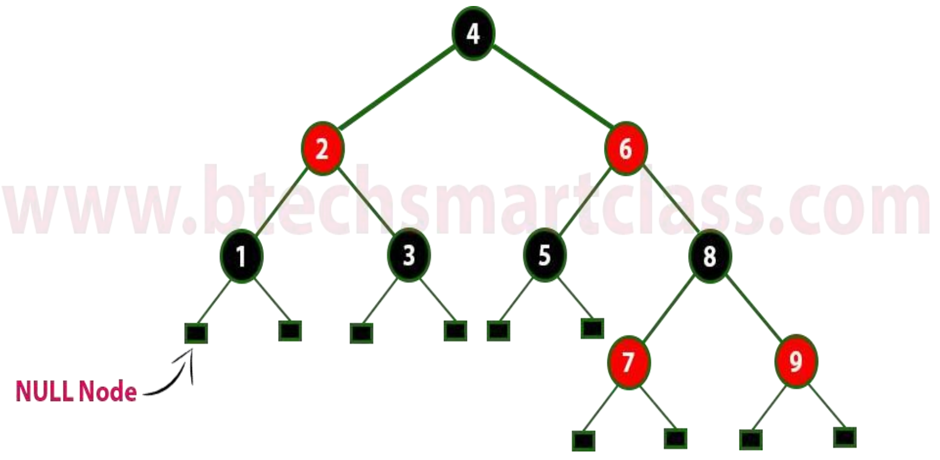
Introduction:

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, This tree was invented in 1972 by Rudolf Bayer.

Rules That Every Red-Black Tree Follows:

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.

EXAMPLE



www.btechsmartclass.com

The above tree is a Red-Black tree where every node is satisfying all the properties of Red-Black Tree.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a Red- Black tree is always O(log n) where n is the number of nodes in the tree. Where **"n" is the total number of elements in the red-black tree.**

Comparison with AVL Tree:

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Interesting points about Red-Black Tree:

1. Black height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height h has black height >= h/2.
2. Height of a red-black tree with n nodes is $h \le 2 \log_2(n + 1)$.
3. All leaves (NIL) are black.
4. The black depth of a node is defined as the number of black nodes from the root to that node i.e the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

Black Height of a Red-Black Tree :

Black height is the number of black nodes on a path from the root to a leaf. Leaf nodes are also counted black nodes. From the above properties 3 and 4, we can derive, **a Red-Black Tree of height h has black-height >= h/2**.

NOTE: Every Red Black Tree with n nodes has height <= 2Log$_2$(n+1)

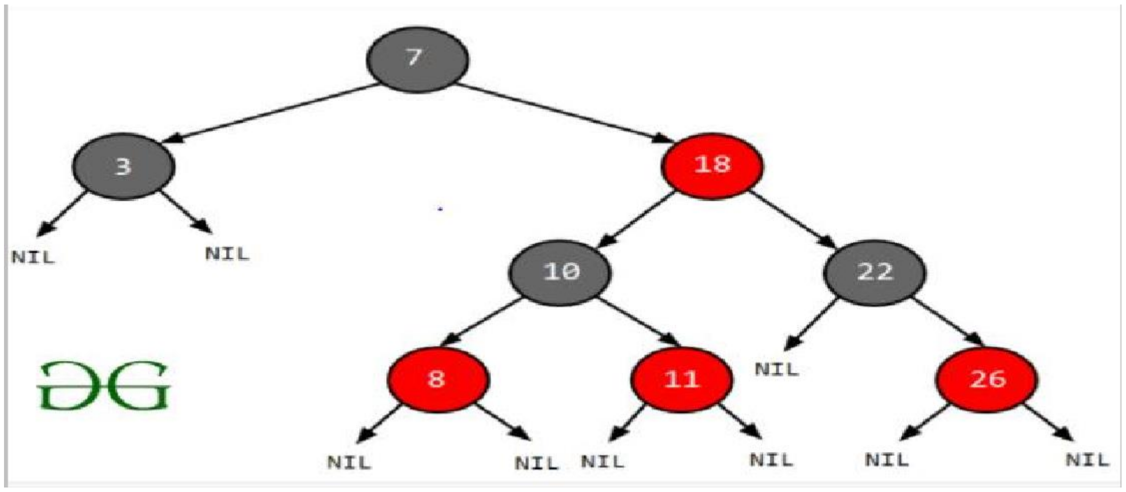
The following operations are performed on Red-Black Tree

1. Search
2. Insertion
3. Deletion

Search operation in Red-Black Tree

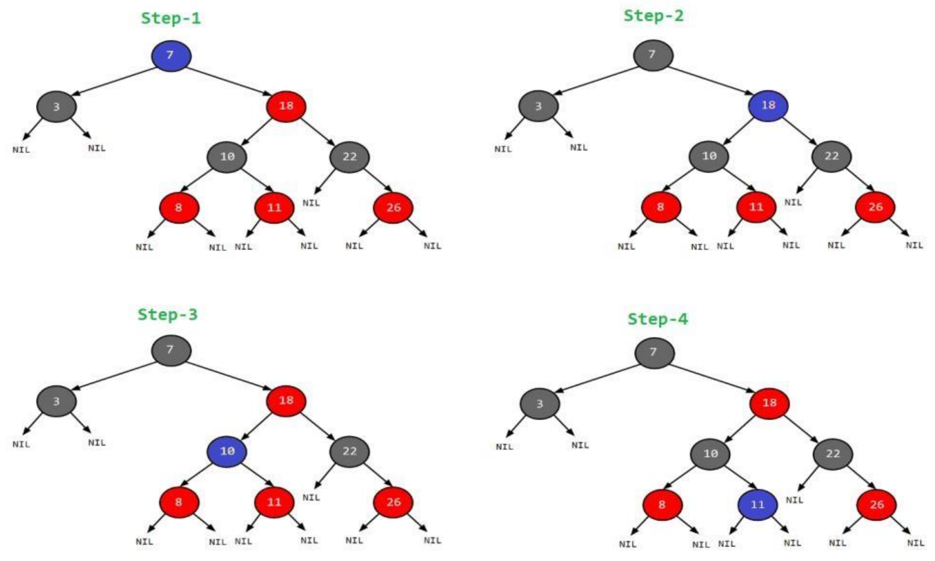
Every red-black tree is a special case of a binary tree so the searching algorithm of a red-black tree is similar to that of a binary tree.

Example: Searching 11 in the following red-black tree.



Solution:

1. Start from the root.
2. Compare the inserting element with root, if less than root, then recurse for left, else recurse for right.
3. If the element to search is found anywhere, return true, else return false.



Insertion operation in Red-Black Tree

In the Red-Black tree, we use two tools to do the balancing.

1. Recoloring
2. Rotation

Re-coloring is the change in color of the node i.e. if it is red then change it to black and vice versa. It must be noted that the color of the NULL node is always black. Moreover, we always try re-coloring first, if re-coloring doesn't work, then we go for rotation.

Following is a detailed algorithm. The algorithms have mainly two cases depending upon the color of the uncle(Uncle means new node parent sibling). If the uncle is red, we do recolor. If theuncle is black, we do rotations and/or re-coloring.

Logic:

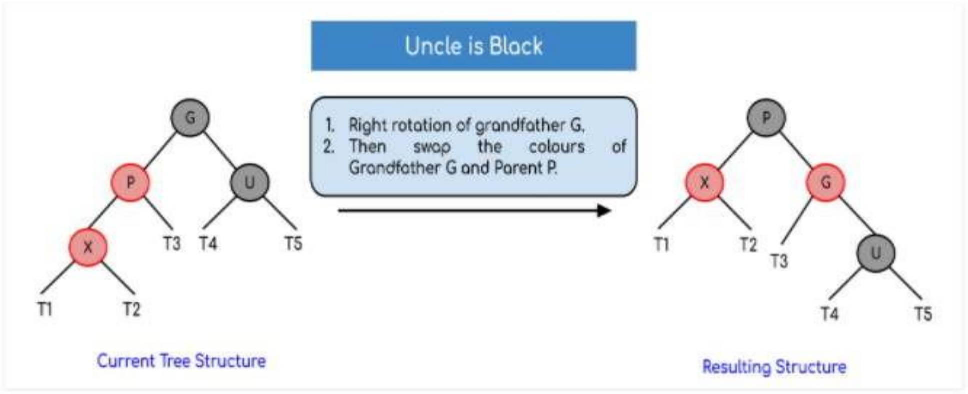
First, you have to insert the node similarly to that in a binary tree and assign a red color to it. Now, if the node is a root node then change its color to black, but if it is not then check the color of the parent node. If its color is black then don't change the color but if it is not i.e. it is red then check the color of the node's uncle. If the node's uncle has a red color then change the color of the node's parent and uncle to black and that of grandfather to red color and repeat the same process for him (i.e. grandfather).

Algorithm

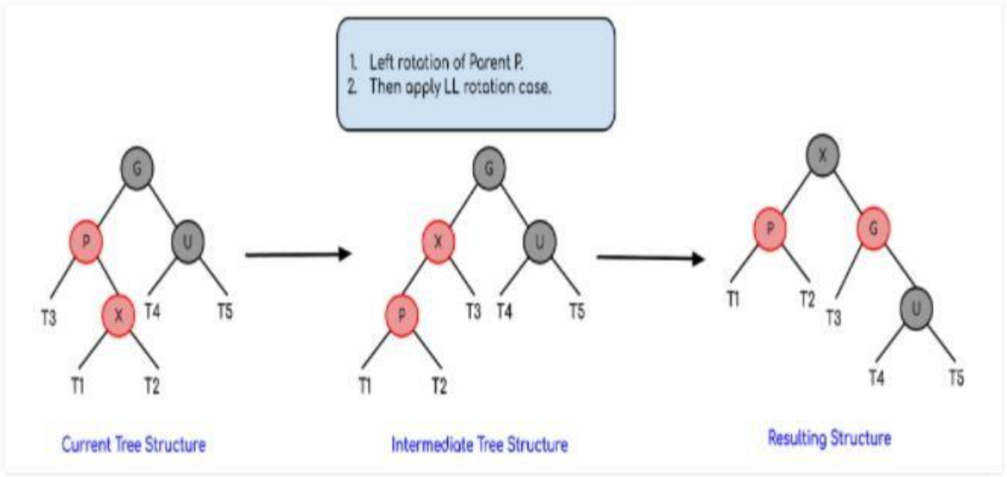
1. Perform standard BST insertion and make the color of newly inserted nodes as RED.
2. If new node (x) is the root, change the color of x as BLACK
3. Do the following if the color of new node (x's) parent is not BLACK **and** x is not the root.
    a) **If x's uncle(**Uncle means new node parent sibling) **is RED** (Grandparent must have been black from Red-Black Tree Property)
        - Change the colour of parent and uncle as BLACK.
        - Colour of a grandparent as RED.
        - Change x = x's grandparent, repeat steps 2 and 3 for new x.

    b). **If x's uncle is BLACK**, then there can be four configurations for x, x's parent (**p**) and x's grandparent (**g**)
    - Left Left Case (p is left child of g and x is left child of p)
    - Left Right Case (p is left child of g and x is the right child of p)
    - Right Right Case
    - Right Left Case

- Left Left Case (LL rotation):

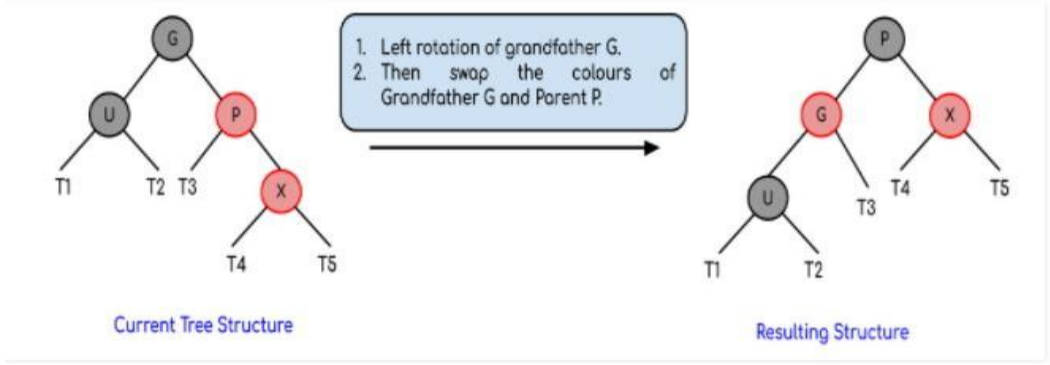

Uncle is Black
1. Right rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.
G
P
U
X
T1
T2
T3
T4
T5
Current Tree Structure
Resulting Structure

- Left Right Case (LR rotation):

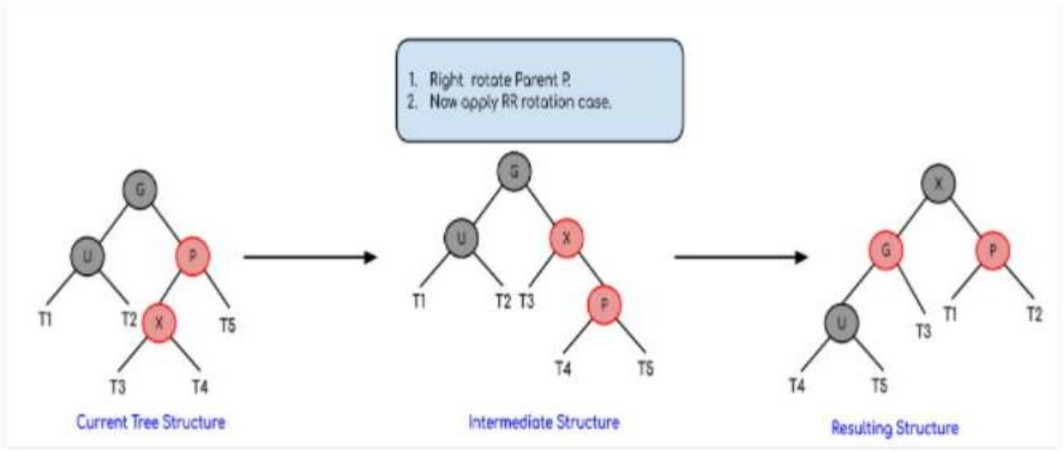

1. Left rotation of Parent P.
2. Then apply LL rotation case.
G
P
U
X
T1
T2
T3
T4
T5
Current Tree Structure
Intermediate Tree Structure
Resulting Structure

➢

- Right Right Case (RR rotation):


1. Left rotation of grandfather G.
2. Then swap the colours of Grandfather G and Parent P.
G
U
P
X
T1
T2
T3
T4
T5
Current Tree Structure
Resulting Structure

- Right Left Case (RL rotation):

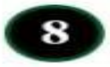


EXAMPLE:

Create a Red-Black Tree with the following sequence of numbers 8,18,5,15,17,25,40 and 80

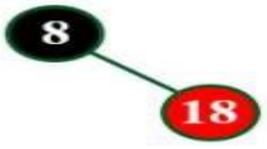
insert (8)

Tree is Empty. So insert newNode as Root node with black color.



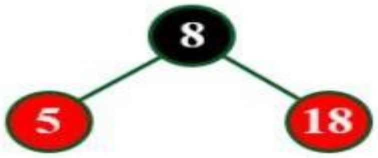
insert (18)

Tree is not Empty. So insert newNode with red color.



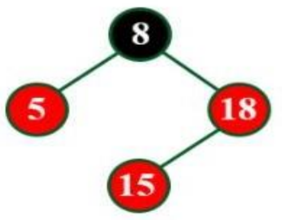
insert (5)

Tree is not Empty. So insert newNode with red color.

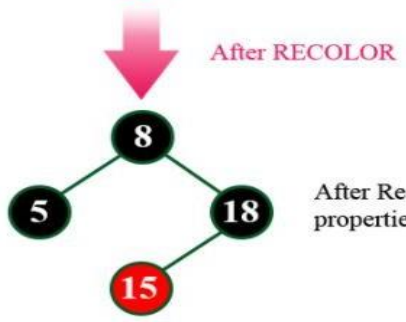


insert (15)

Tree is not Empty. So insert newNode with red color.



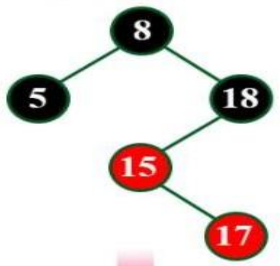
Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.



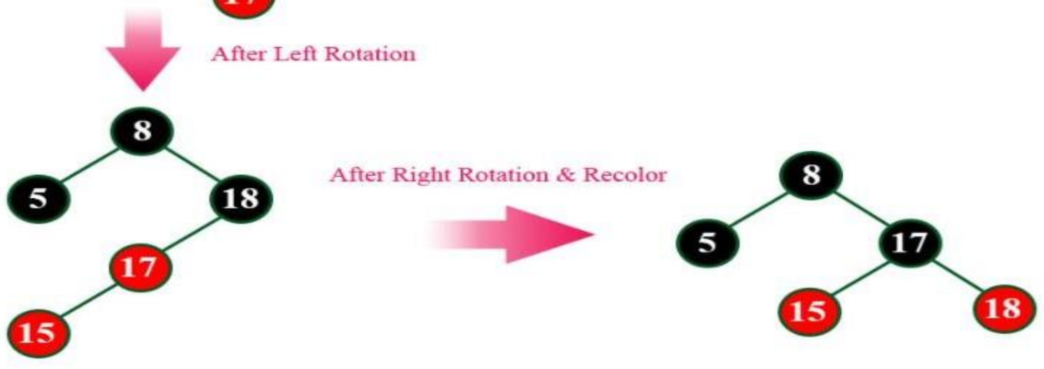
After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (17)

Tree is not Empty. So insert newNode with red color.

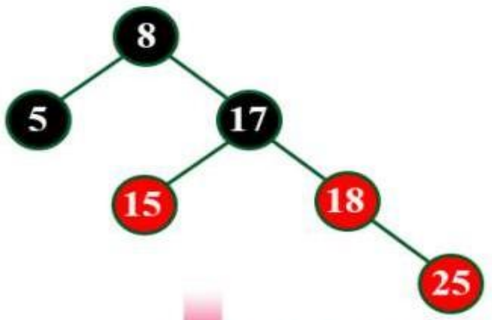


Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.



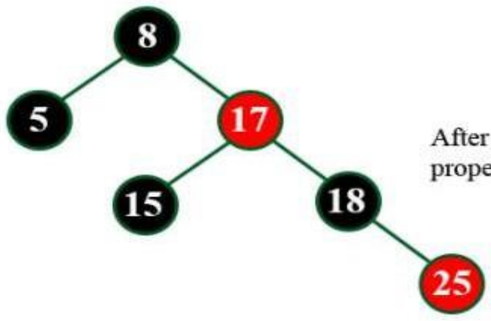
insert (25)

Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

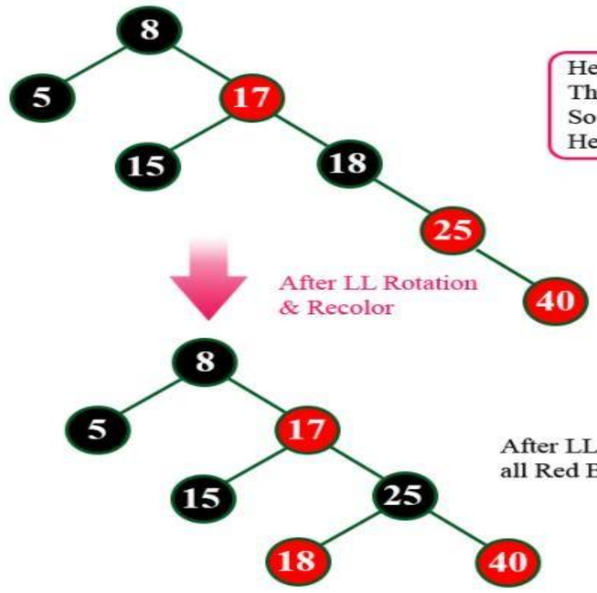
After Recolor



After Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (40)

Tree is not Empty. So insert newNode with red color.

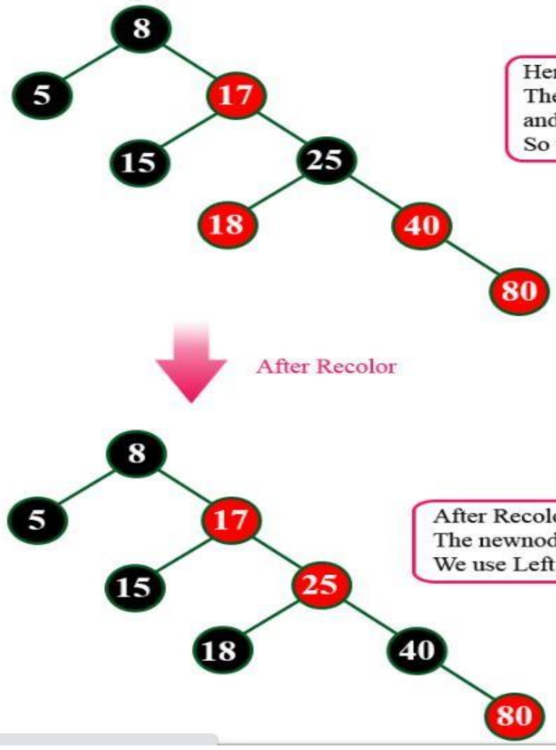


Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.

insert (80)

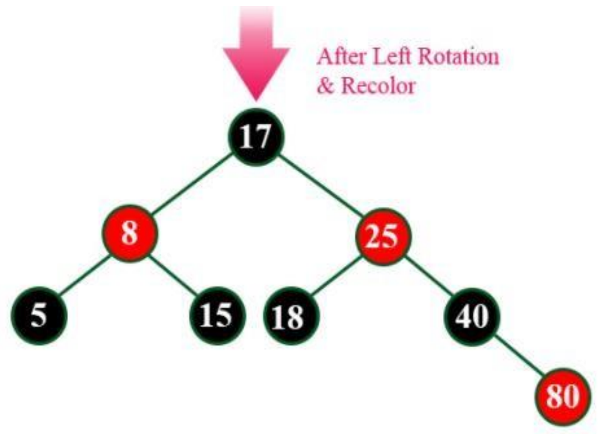
Tree is not Empty. So insert newNode with red color.



Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

Activat
Go to Set



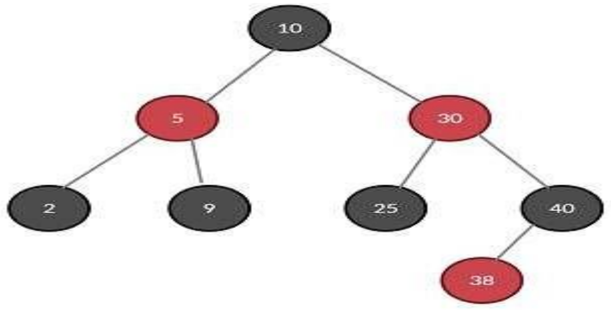
Finally above tree is satisfying all the properties of Red Black Tree and it is a perfect Red Black tree.

Deletion in Red Black Tree

The below table is useful to identify the case and its corresponding set of actions to be performed when deleting a node from Red Black Tree

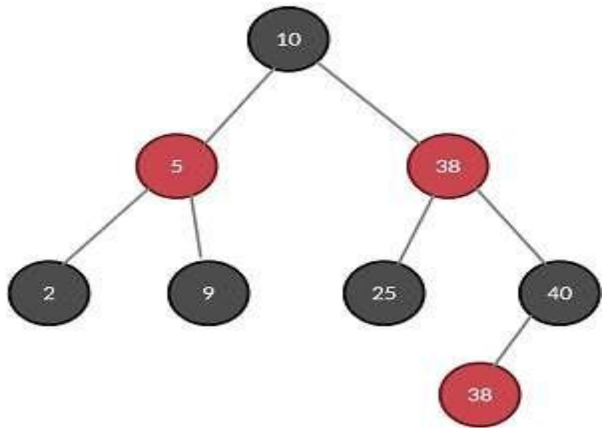
Case #	Check condition	Action
1	If node to be delete is a red leaf node	Just remove it from the tree
2	If DB node is root	Remove the DB and root node becomes black.
3	(a) If DB's sibling is black, and (b) DB's sibling's children are black	(a) Remove the DB (if null DB then delete the node and for other nodes remove the DB sign) (b) Make DB's sibling red. (c) If DB's parent is black, make it DB, else make it black
4	If DB's sibling is red	(a) Swap color DB's parent with DB's sibling (b) Perform rotation at parent node in the direction of DB node (c) Check which case can be applied to this new tree and perform that action
5	(a) DB's sibling is black (b) DB's sibling's child which is far from DB is black (c) DB's sibling's child which is near to DB is red	(a) Swap color of sibling with sibling's red child (b) Perform rotation at sibling node in direction opposite of DB node (c) Apply case 6
6	(a) DB's sibling is black, and (b) DB's sibling's far child is red (remember this node)	(a) Swap color of DB's parent with DB's sibling's color (b) Perform rotation at DB's parent in direction of DB (c) Remove DB sign and make the node normal black node (d) Change colour of DB's sibling's far red child to black.

Example 1: Delete 30 from the RB tree in fig. 3



Initial RB Tree

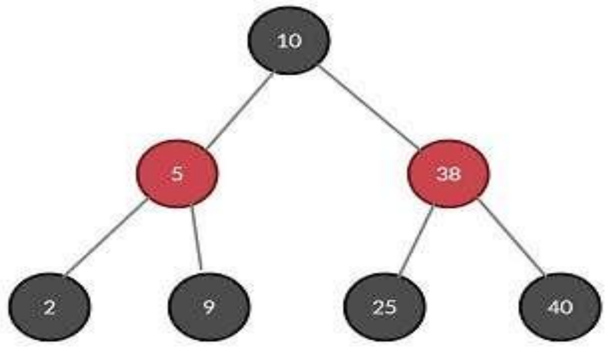
You first have to search for 30, once found perform BST deletion . For a node with value '30', find either the maximum of the left subtree or a minimum of the right subtree and replace 30 with that value. This is BST deletion .



RB Tree after replacing 30 with min element from right subtree

The resulting RB tree will be like one in fig. 4. Element 30 is deleted and the value is successfully replaced by 38. But now the task is to delete duplicate element 38.

Go to the table above and you'll observe **case 1** is satisfied by this tree.

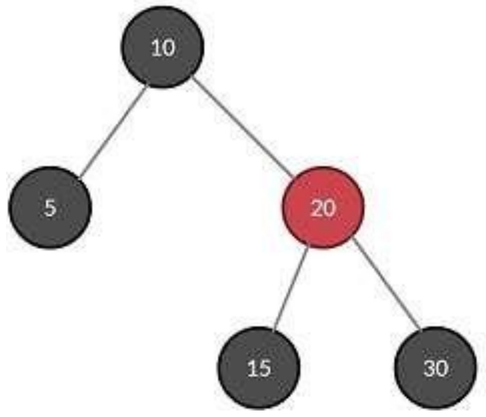


After removing the red leaf node

Since node with element 38 is a *red* leaf node, remove it and the tree looks like the one in fig. 5.

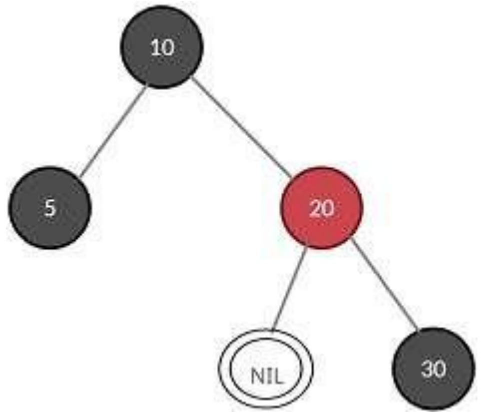
Observe that if you perform correct actions, the tree will still hold all the properties of the RB tree.

Example 2: Delete 15 from below RB tree



Initial RB Tree

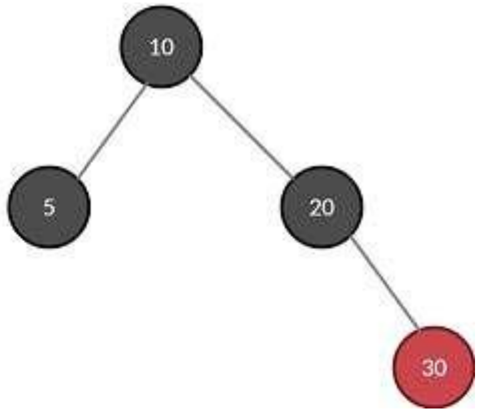
15 can be removed easily from the tree (BST deletion). In the case of RB trees, if a leaf node is deleted you replace it with a **double black (DB)** nil node . It is represented by a double circle.



NIL node added in place of 15

The entire problem is now drilled down to get rid of this bad boy, DB, via some actions.

Go back to our rule book (table) and **case 3** fits perfectly.



NIL Node removed after applying actions

In short, remove DB and then swap the color of its sibling with its parent

Example 3: Delete '15' from fig.(A).

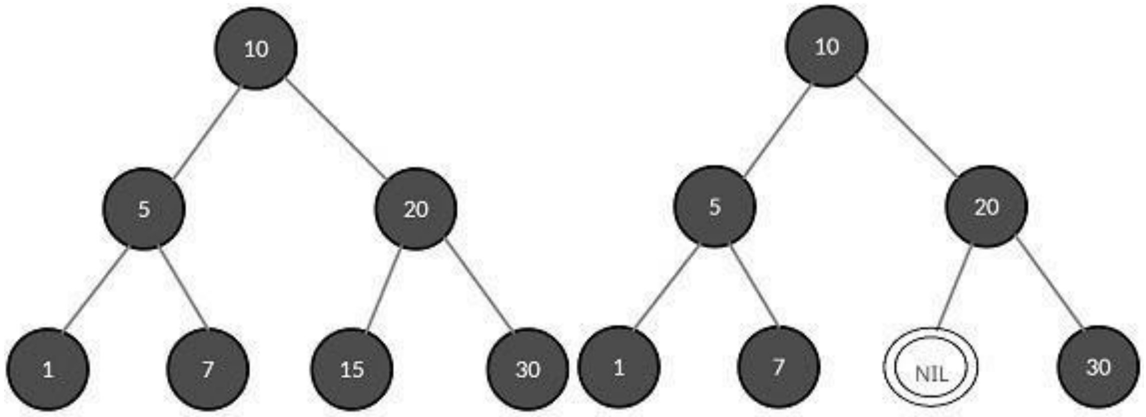


Fig: (A) Initial RB Tree, (B) NIL node added in place of 15

Delete node with value 15 and, as a rule, replace it with DB nil node as shown. Now, DB's sibling is black and sibling's both children are also black (don't forget the hidden NIL nodes!), it satisfies all the conditions of case 3. Here,

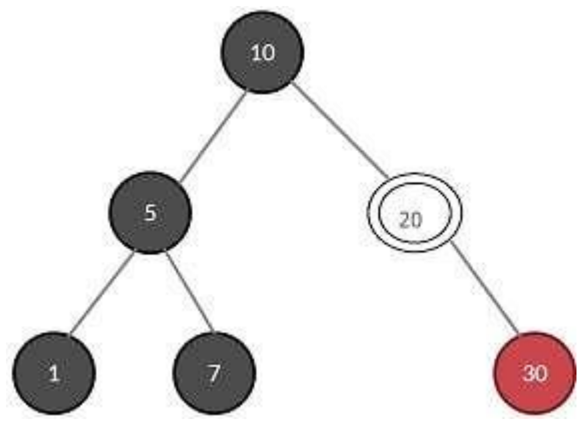


Fig: RB Tree after case 3 is applied

1. DB's parent is 20
2. DB's parent is *black*
3. DB's sibling is 30

With these points in mind perform the actions and you get an RB tree as in fig. 10.

20 becomes DB and hence the problem is not resolved yet. Reapply case 3

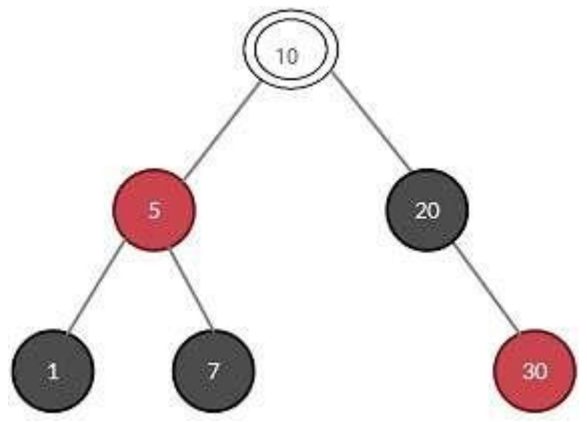


Fig. RB Tree after case 3 is applied

The resulting tree looks like the one in the above fig.

The DB still exist . Recheck which case will be applicable.

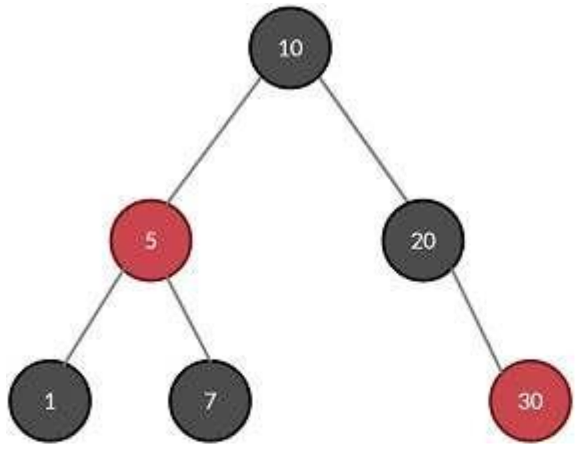
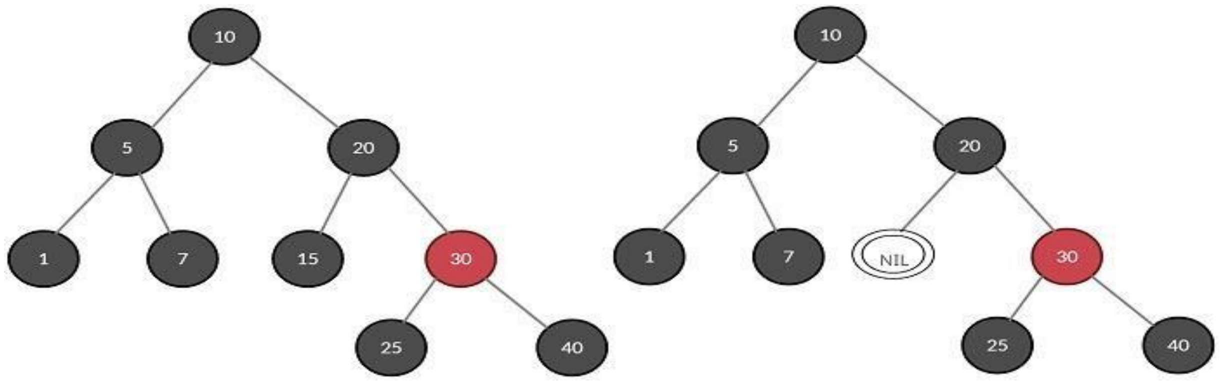


Fig.: NIL Node removed after applying actions

Found it? It’s case 2, the simplest of all!

The root resolved DB and becomes a *black* node. And you’re done deleting 15 successfully.

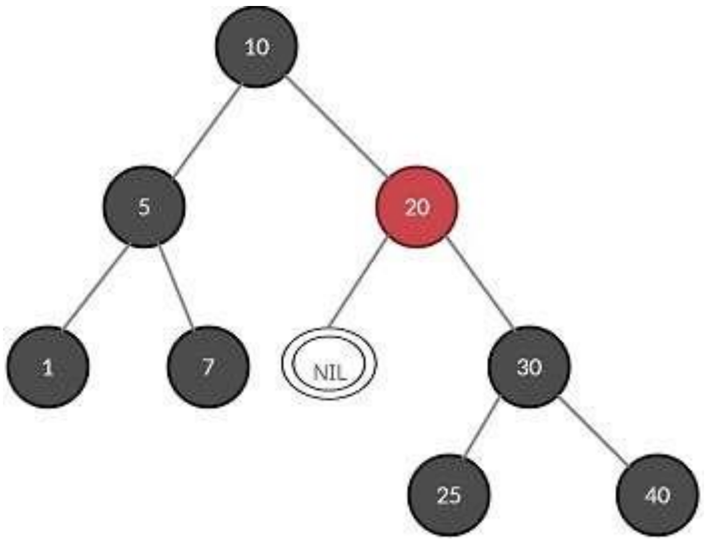
Example 4: Delete ‘15’ from below fig. (A).



(A) Initial RB Tree, (B) NIL node added in place of 15

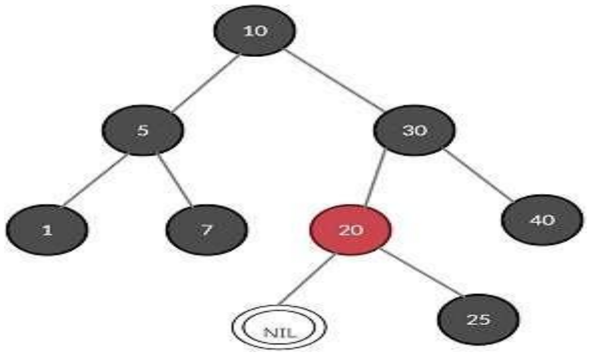
First, Search 15 as per BST rules and then delete it. Second, replace deleted node with DB NIL node as shown in fig. 13 (B).

DB’s sibling is *red.* Clearly**, case 4** is applicable.



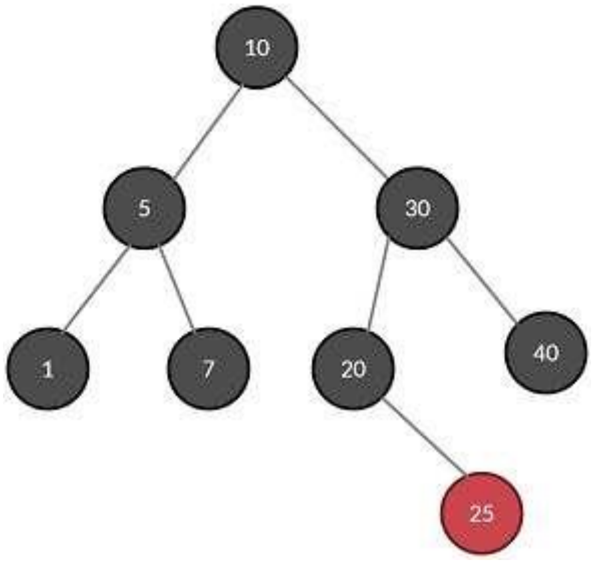
RB Tree after case 4 is applied

(a) Swap DB’s parent’s color with DB’s sibling’s color. I know this is confusing, but take it easy and keep following. The tree looks like fig. 14.



(b) Perform rotation at parent node in direction of DB. The tree becomes like the one in fig. 15. DB is still there (what’s its problem!).

(c) Check which case can be applied in the current tree. And got it, **case 3.**

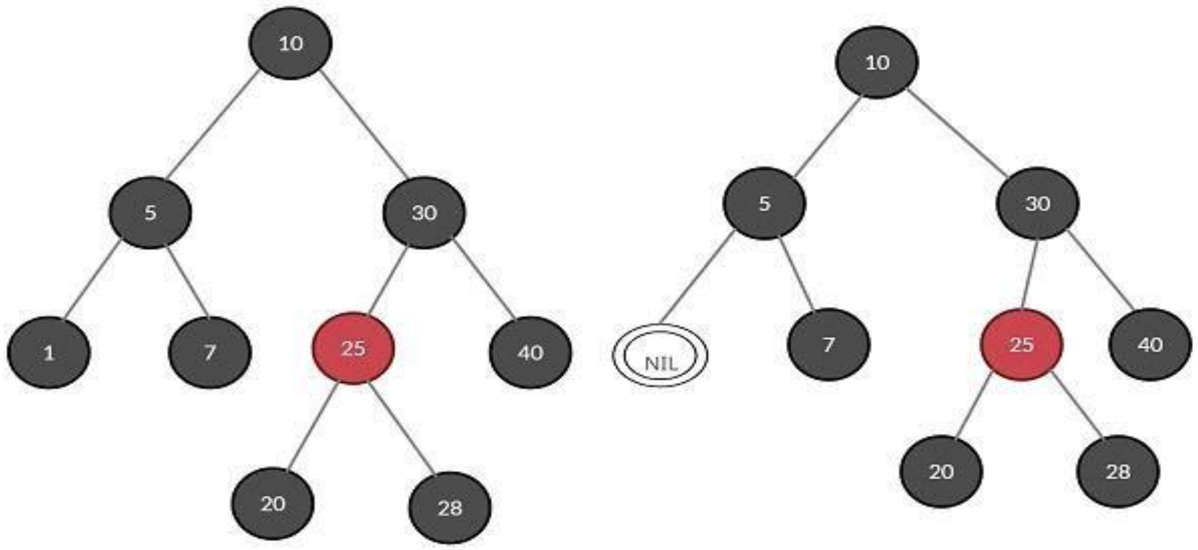


NIL Node removed after applying actions

(d) Apply case 3 as explained and the RB tree is free from the DB node as shown in fig. 16.

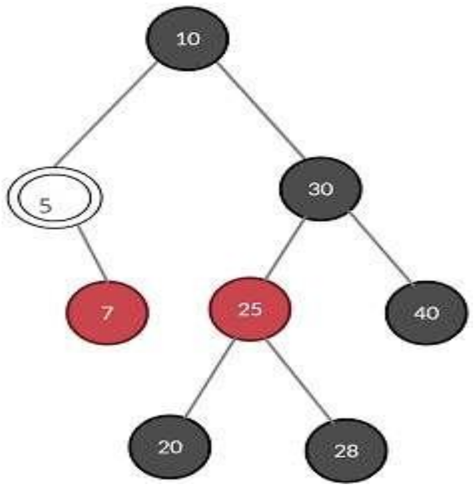
I know it’s tiresome, but I swear if you practice these examples 2–3 times, you will have a good grasp of the concept of deletion in RB trees.

Example 5: Delete ‘1’ from below fig(A).



(A) Initial RB Tree, (B) NIL node added in place of 1

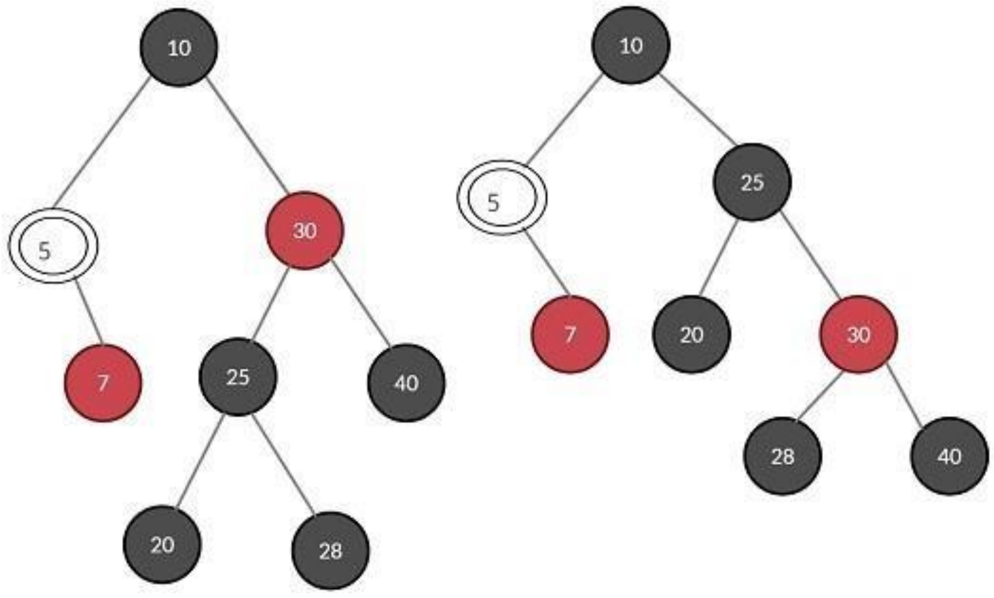
Perform the basic preliminary steps- delete the node with value 1 and replace it with DB NIL node as shown in fig. 17(B). Check for the cases which fit the current tree and it’s case 3(DB’s sibling is *black*).



RB Tree after case 3 is applied

Node 5 has now become a *double black* node. We need to get rid of it.

Search for cases that can be applied and case 5 seems to fit here (not case 3).



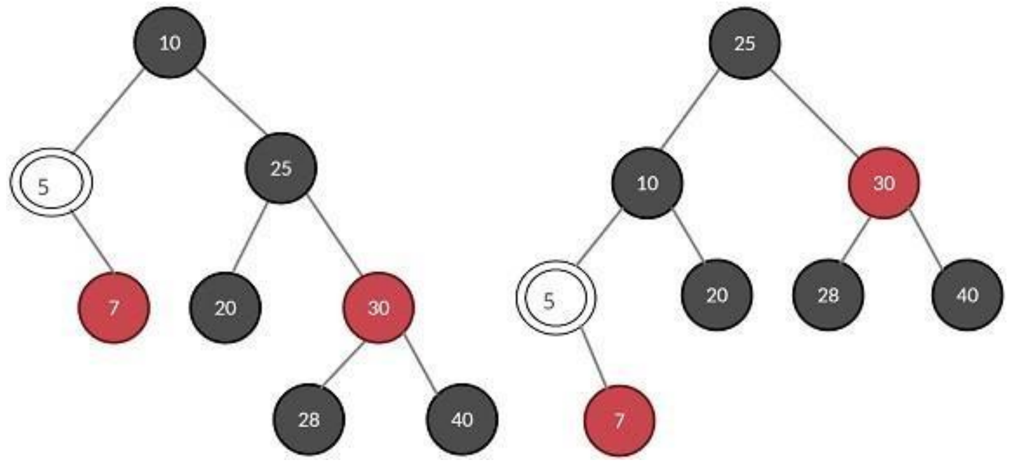
(A) Tree after swapping colors of 30 & 25 (B) Tree after rotation

Case 5 is applied as follows-

(a) swap colors of nodes 30 and 25 (fig. 19(A))

(b) Rotate at sibling node in the direction opposite to the DB node. Hence, perform right rotation at node 30 and the tree becomes like fig. 19 (B).

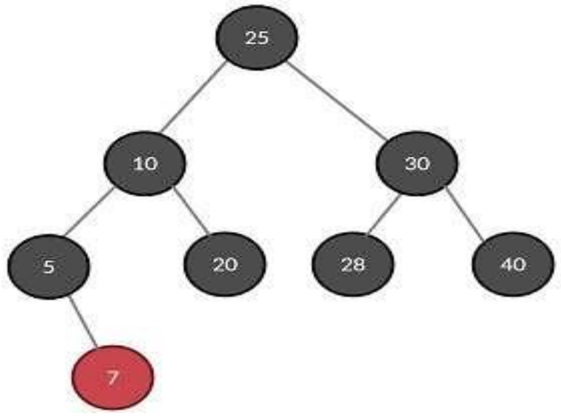
The *double black* node still haunts the tree! Re-check the case that can be applied to this tree and we find that case 6 (don't fall for case 3) seems to fit.



Apply **case 6** as follow-

(a) Swap colors of DB's parent with DB's sibling.

(b) Perform rotation at DB's parent node in the direction of DB (fig, 20(B)).



NIL Node removed after applying actions

(c) Change DB node to *black* node. Also, change the color of DB's sibling's far-*red* child to black and the final RB tree will look fig. 21.

And, voilà! The RB tree is free of element 1 as well as of any *double node*. Life is good now.

Applications of Red-Black Trees

Real-world uses of red-black trees include TreeSet, TreeMap, and Hashmap in the Java Collections Library.

Splay Tree

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees. The prerequisite for the splay trees that we should know about the binary search trees.

As we already know, the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is **O(logn)** and the time complexity in the worst case is O(n). In a binary search tree, the value of the left subtree is smaller than the root node, and the value of the right subtree is greater than the root node; in such case, the time complexity would be **O(logn)**. If the binary tree is left-skewed or right-skewed, then the time complexity would be O(n). To limit the skewness, the AVL and Red-Black tree came into the picture, having **O(logn)** time complexity for all the operations in all the cases. We can also improve this time complexity by doing more practical implementations, so the new Tree data structure was designed, known as a Splay tree.

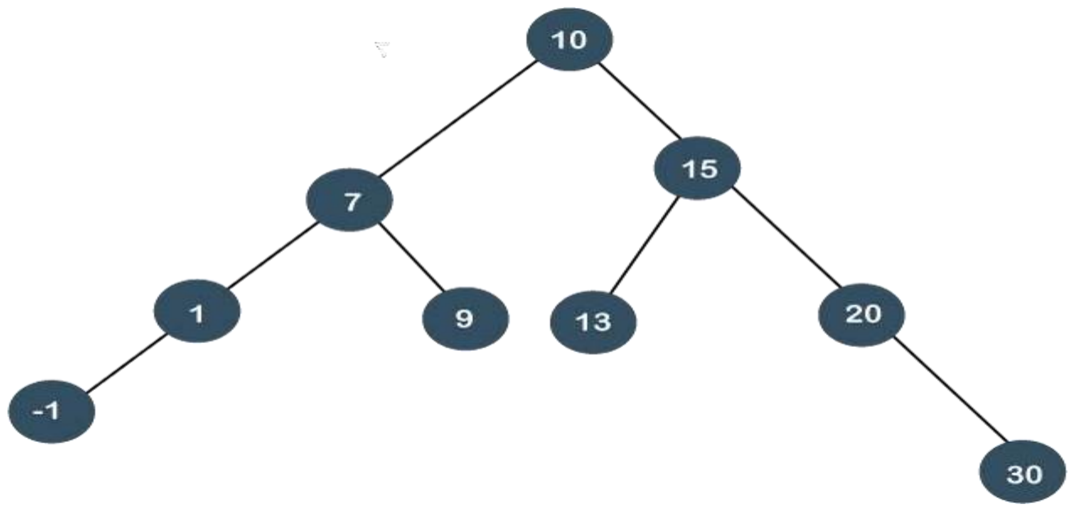
What is a Splay Tree?

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

Splay trees are not strictly balanced trees, but they are roughly balanced trees. Let's understand the search operation in the splay-tree.

Suppose we want to search 7 element in the tree, which is shown below:



To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

Note: The splay tree can be defined as the self-adjusted tree in which any operation performed on the element would rearrange the tree so that the element on which operation has been performed becomes the root node of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertionoperation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree. The search operation in a splay tree is nothing but searching the element using binary search process and then splayingthat searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

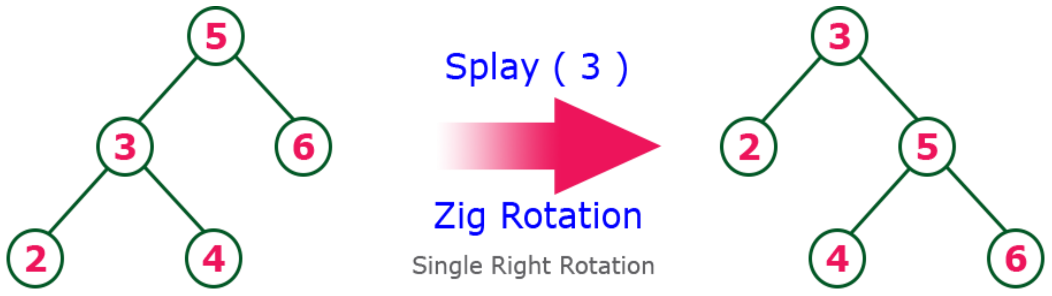
Rotations in Splay Tree

- **1. Zig Rotation**
- **2. Zag Rotation**
- **3. Zig - Zig Rotation**
- **4. Zag - Zag Rotation**
- **5. Zig - Zag Rotation**
- **6. Zag - Zig Rotation**

Example

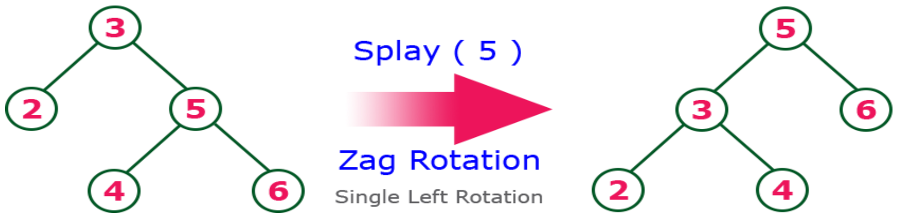
Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...



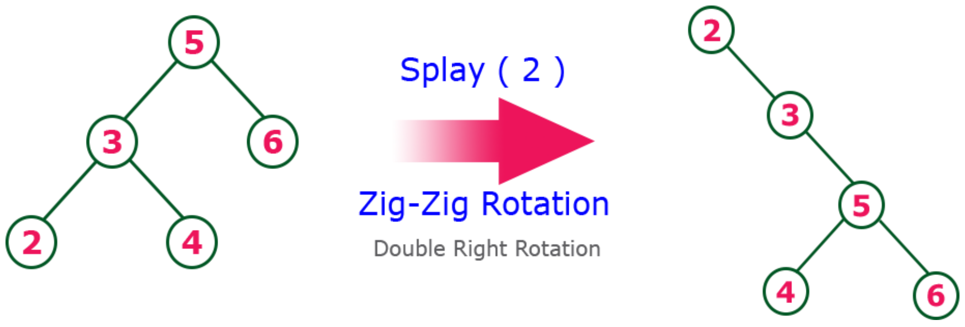
Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...



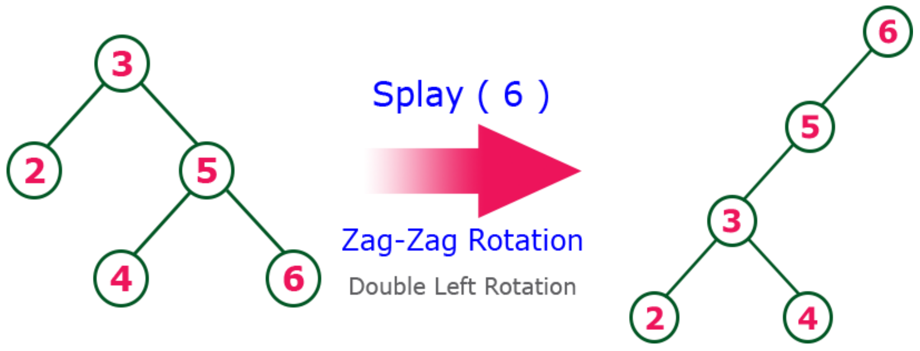
Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...



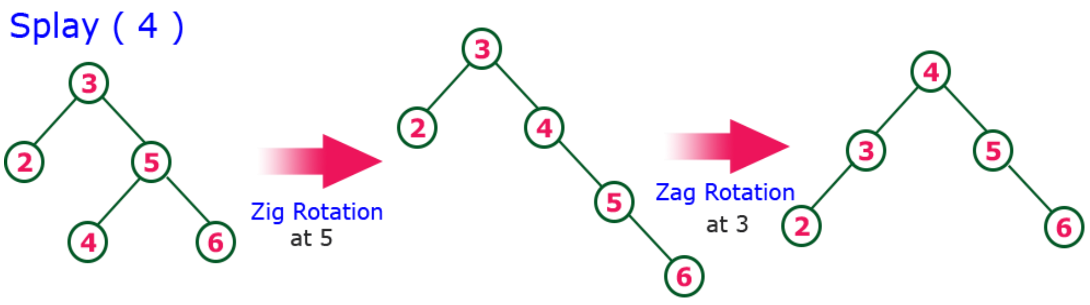
Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...



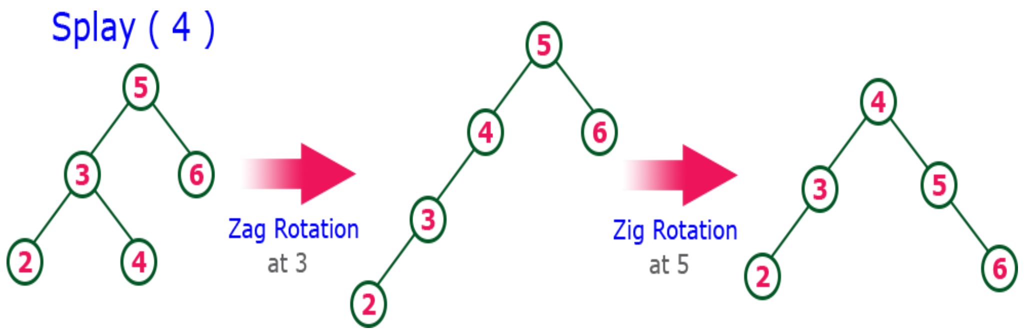
Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...



Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...



Rotations

There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

Scenario 1: If the node is the right of the parent and the parent is also right of its parent, then ***zig zig right right rotation*** is performed.

Scenario 2: If the node is left of a parent, but the parent is right of its parent, then ***zig zag right left rotation*** is performed.

Scenario 3: If the node is right of the parent and the parent is right of its parent, then ***zig zig left left rotation*** is performed.

Scenario 4: If the node is right of a parent, but the parent is left of its parent, then ***zig zag right-left rotation*** is performed.

Now, let's understand the above rotations with examples.

To rearrange the tree, we need to perform some rotations. The following are the types of rotations in the splay tree:

- **Zig rotations**

The zig rotations are used when the item to be searched is either a root node or the child of a root node (i.e., left or the right child).

The following are the cases that can exist in the splay tree while searching:

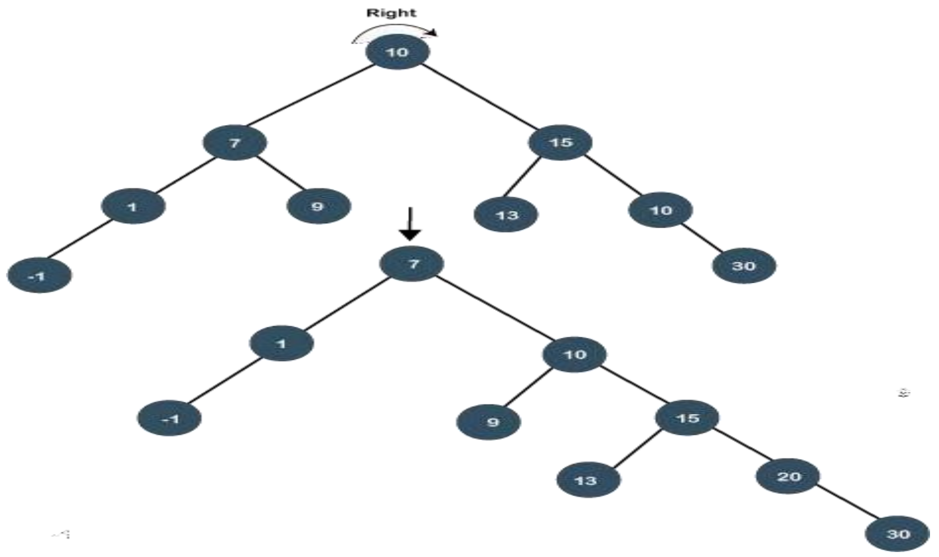
Case 1: If the search item is a root node of the tree.

Case 2: If the search item is a child of the root node, then the two scenarios will be there:

1. If the child is a left child, the right rotation would be performed, known as a zig right rotation.
2. If the child is a right child, the left rotation would be performed, known as a zig left rotation.

Let's look at the above two scenarios through an example.

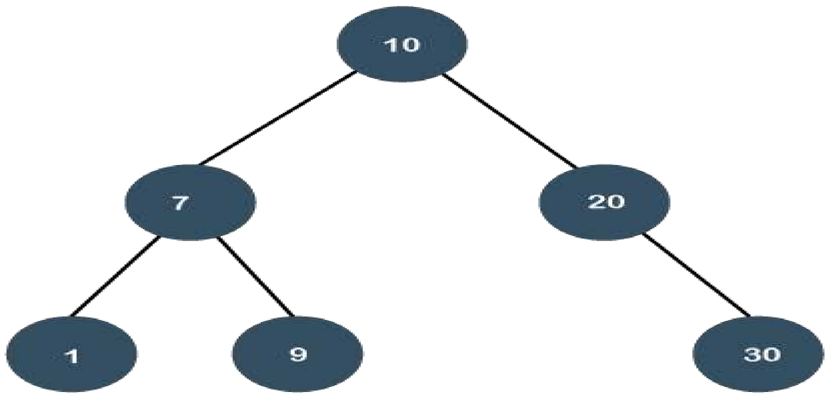
Consider the below example:
In the above example, we have to search 7 element in the tree. We will follow the below steps:
Step 1: First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.
Step 2: Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:



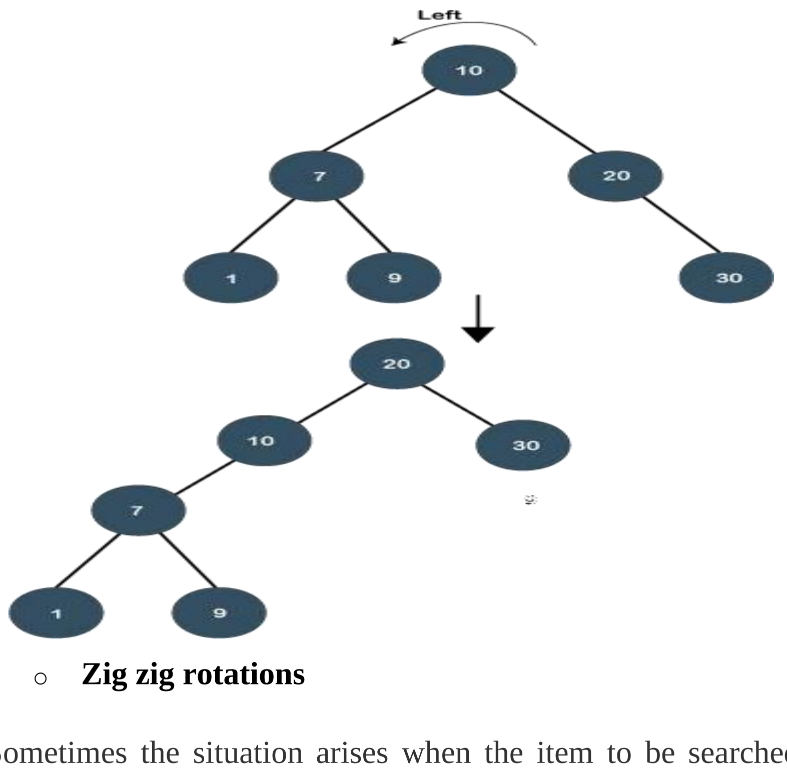
Let's consider another example.

In the above example, we have to search 20 element in the tree. We will follow the below steps:

Step 1: First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.



Step 2: Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



- **Zig zig rotations**

Sometimes the situation arises when the item to be searched is having a parent as well as a grandparent. In this case, we have to perform four rotations for splaying.

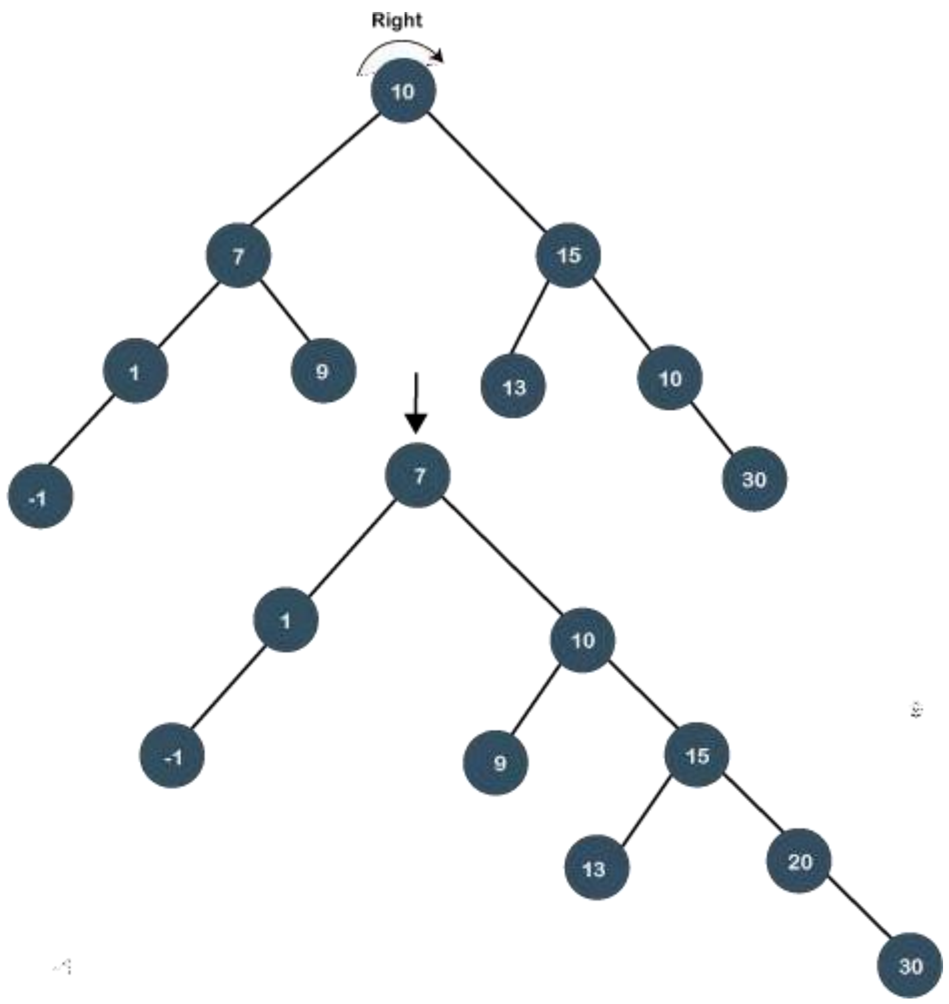
Let's understand this case through an example.

Suppose we have to search 1 element in the tree, which is shown below:

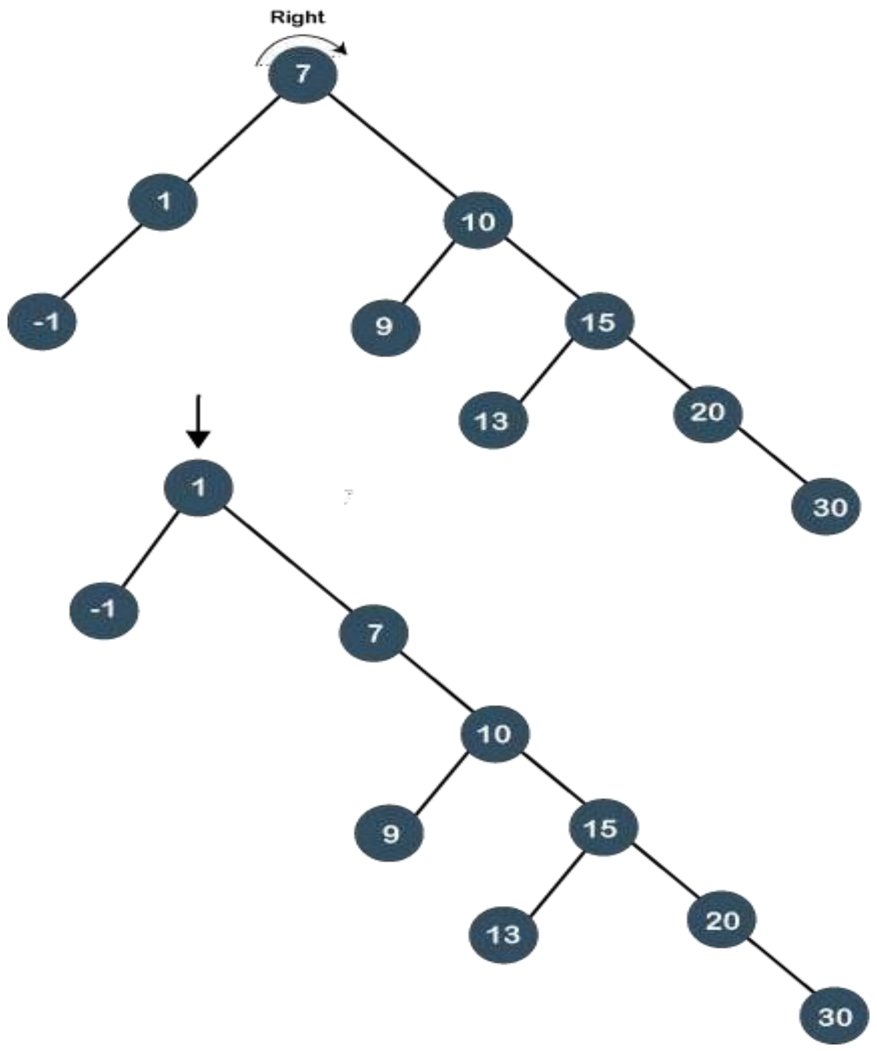
Step 1: First, we have to perform a standard BST searching operation in order to search the 1 element. As 1 is less than 10 and 7, so it will be at the left of the node 7. Therefore, element 1 is having a parent, i.e., 7 as well as a grandparent, i.e., 10.

Step 2: In this step, we have to perform splaying. We need to make node 1 as a root node with the help of some rotations. In this case, we cannot simply perform a zig or zag rotation; we have to implement zig zig rotation.

In order to make node 1 as a root node, we need to perform two right rotations known as zig zig rotations. When we perform the right rotation then 10 will move downwards, and node 7 will come upwards as shown in the below figure:



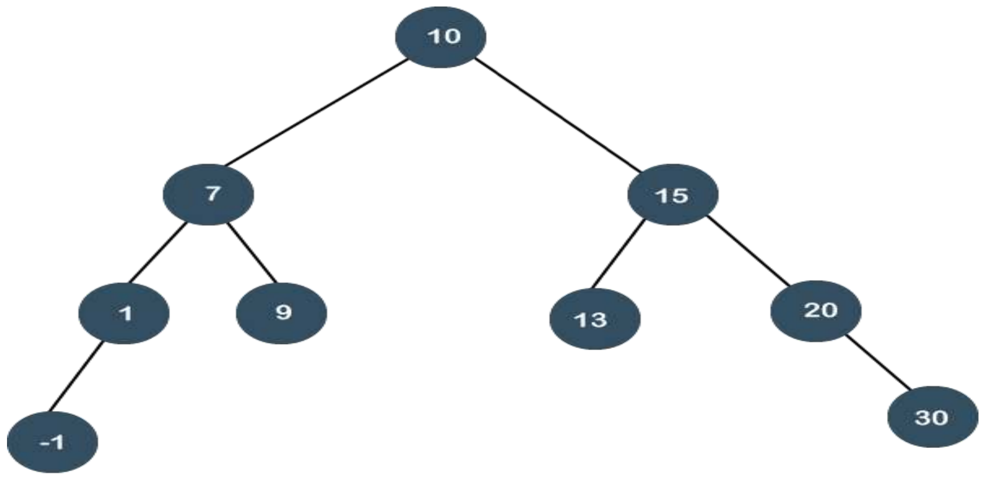
Again, we will perform zig right rotation, node 7 will move downwards, and node 1 will come upwards as shown below:



As we observe in the above figure that node 1 has become the root node of the tree; therefore, the searching is completed.

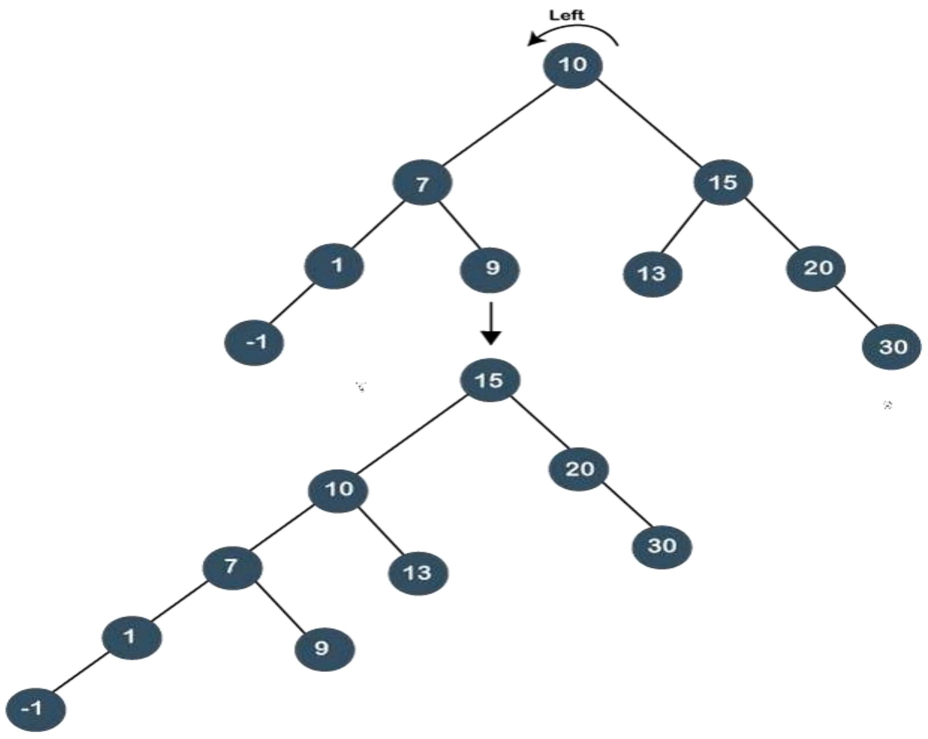
Suppose we want to search 20 in the below tree.

In order to search 20, we need to perform two left rotations. Following are the steps required to search 20 node:

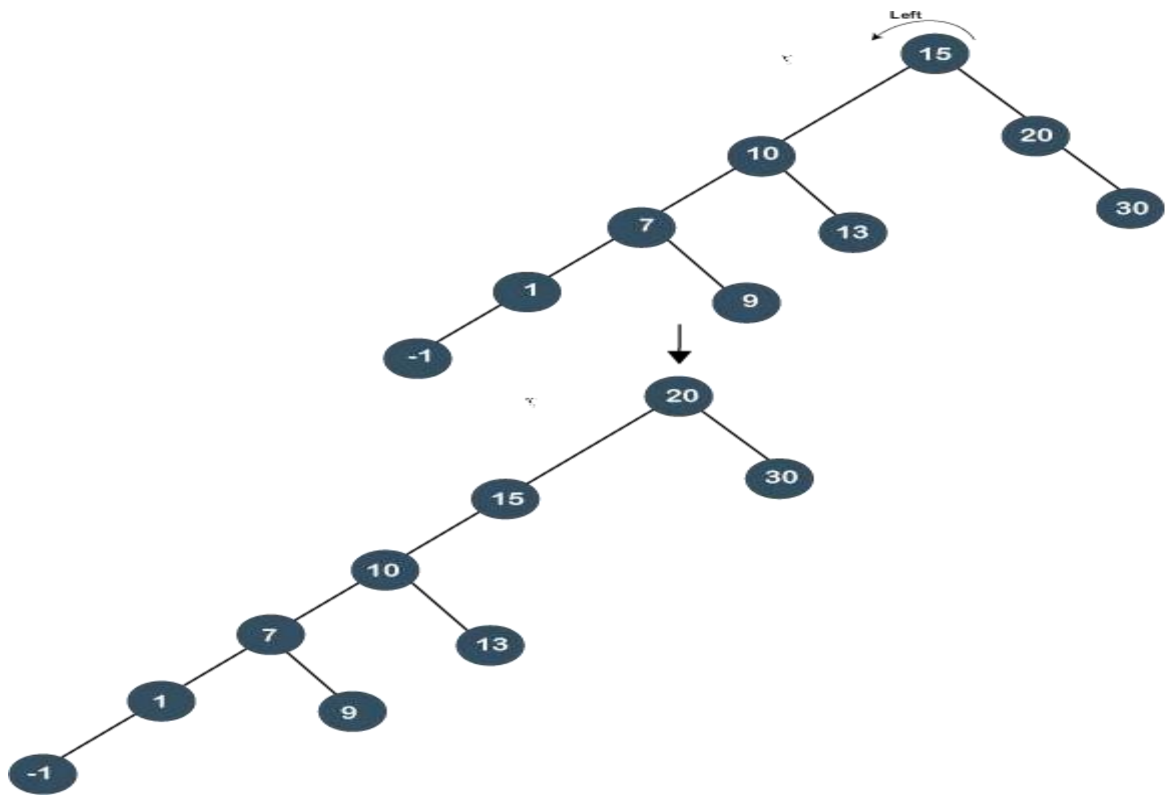


Step 1: First, we perform the standard BST searching operation. As 20 is greater than 10 and 15, so it will be at the right of node 15.

Step 2: The second step is to perform splaying. In this case, two left rotations would be performed. In the first rotation, node 10 will move downwards, and node 15 would move upwards as shown below:



In the second left rotation, node 15 will move downwards, and node 20 becomes the root node of the tree, as shown below:



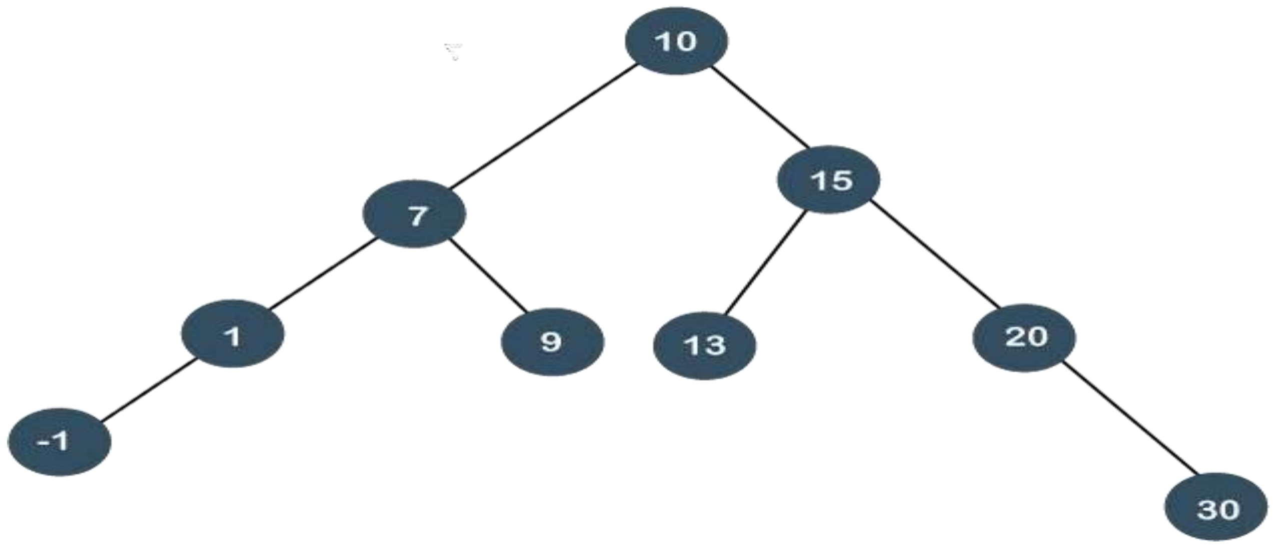
As we have observed that two left rotations are performed; so it is known as a zig zig left rotation.

- **Zig zag rotations**

Till now, we have read that both parent and grandparent are either in RR or LL relationship. Now, we will see the RL or LR relationship between the parent and the grandparent.

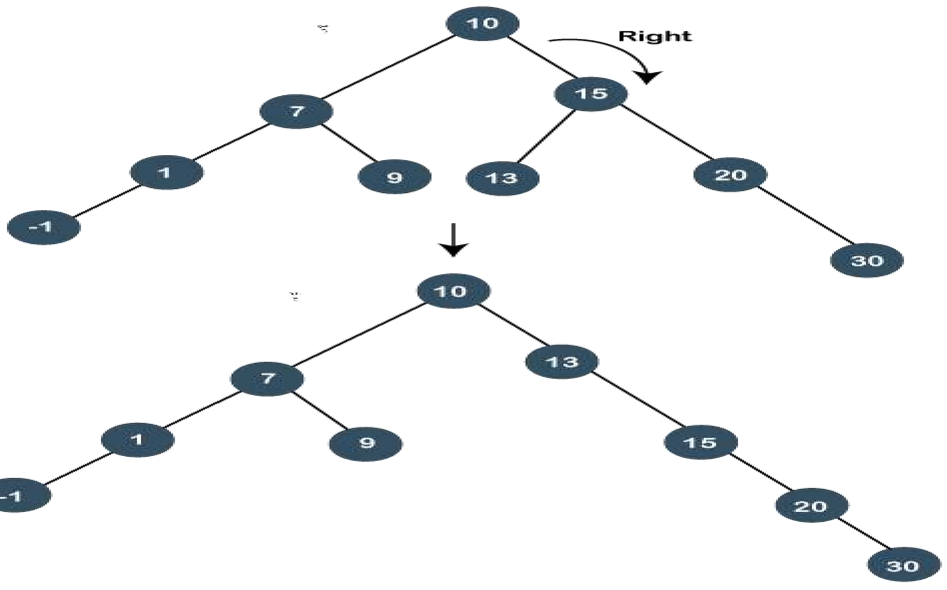
Let's understand this case through an example.

Suppose we want to search 13 element in the tree which is shown below:

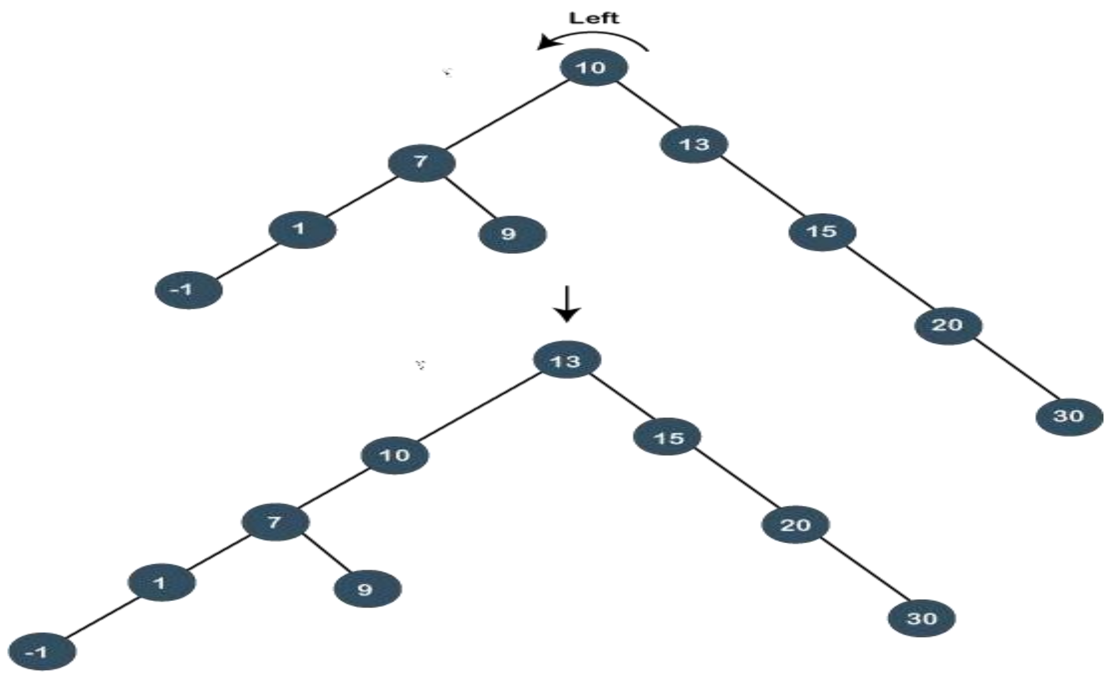


Step 1: First, we perform standard BST searching operation. As 13 is greater than 10 but less than 15, so node 13 will be the left child of node 15.

Step 2: Since node 13 is at the left of 15 and node 15 is at the right of node 10, so RL relationship exists. First, we perform the right rotation on node 15, and 15 will move downwards,and node 13 will come upwards, as shown below:



Still, node 13 is not the root node, and 13 is at the right of the root node, so we will perform left rotation known as a zag rotation. The node 10 will move downwards, and 13 becomes the root node as shown below:

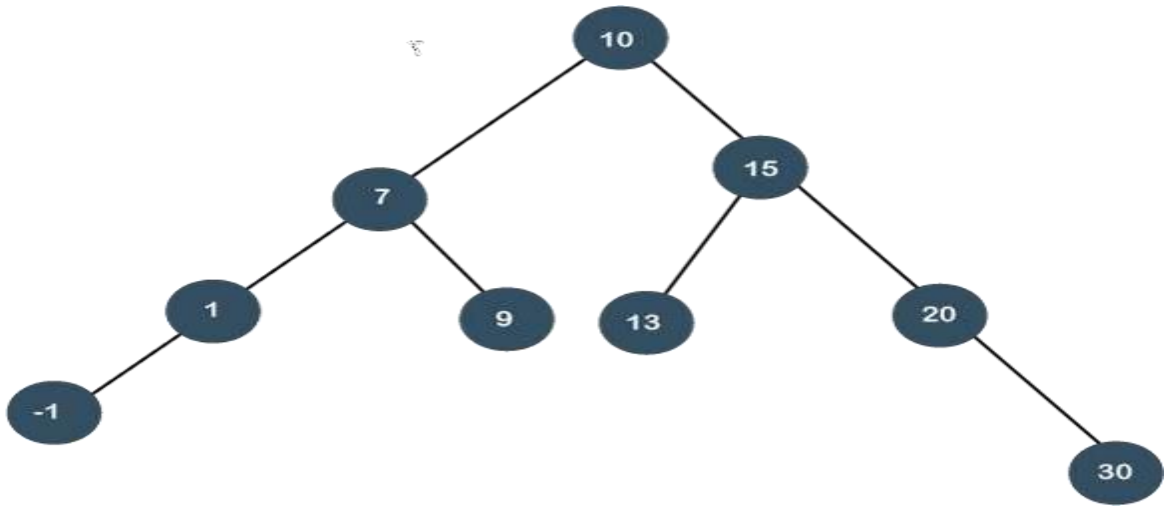


As we can observe in the above tree that node 13 has become the root node; therefore, the searching is completed. In this case, we have first performed the zig rotation and then zag rotation; so, it is known as a zig zag rotation.

- **Zag zig rotation**

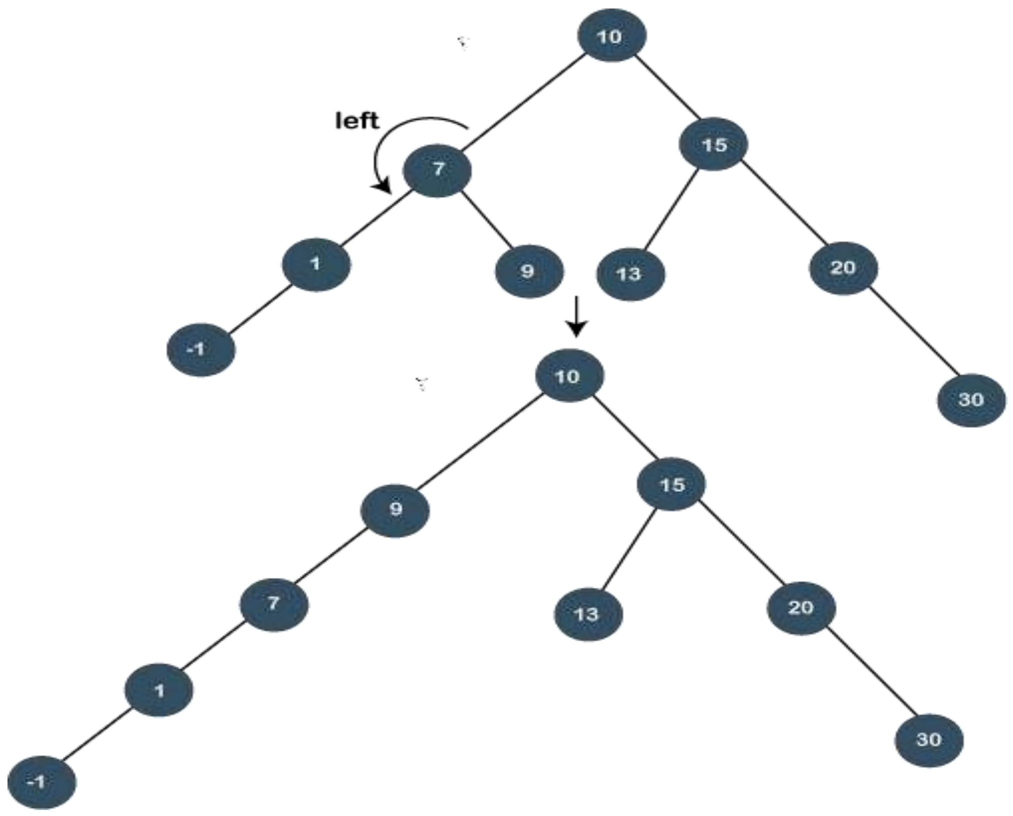
Let's understand this case through an example.

Suppose we want to search 9 element in the tree, which is shown below:

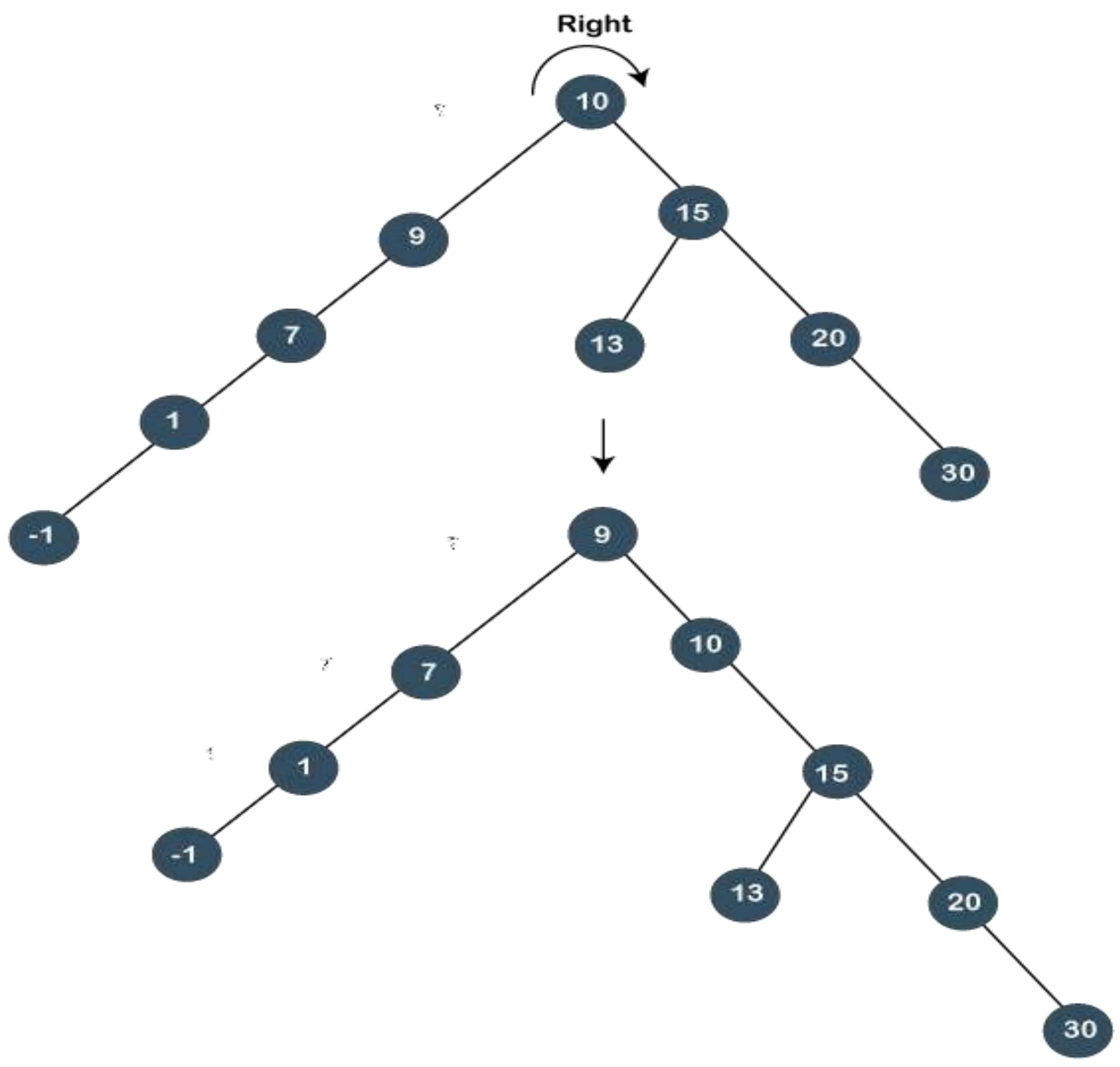


Step 1: First, we perform the standard BST searching operation. As 9 is less than 10 but greater than 7, so it will be the right child of node 7.

Step 2: Since node 9 is at the right of node 7, and node 7 is at the left of node 10, so LR relationship exists. First, we perform the left rotation on node 7. The node 7 will move downwards, and node 9 moves upwards as shown below:



Still the node 9 is not a root node, and 9 is at the left of the root node, so we will perform the right rotation known as zig rotation. After performing the right rotation, node 9 becomes the root node, as shown below:



As we can observe in the above tree that node 13 is a root node; therefore, the searching is completed. In this case, we have first performed the zag rotation (left rotation), and then zig rotation (right rotation) is performed, so it is known as a zag zig rotation.

Advantages of Splay tree

- In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.
- It is the fastest type of Binary Search tree for various practical applications. It is used in **Windows NT and GCC compilers**.
- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in

the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

Drawback of Splay tree

The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

Insertion operation in Splay tree

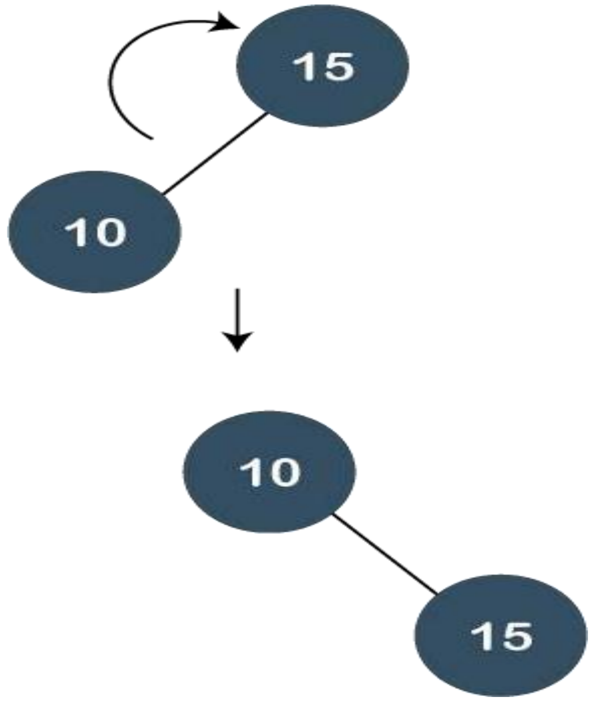
In the ***insertion*** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

15, 10, 17, 7

Step 1: First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.

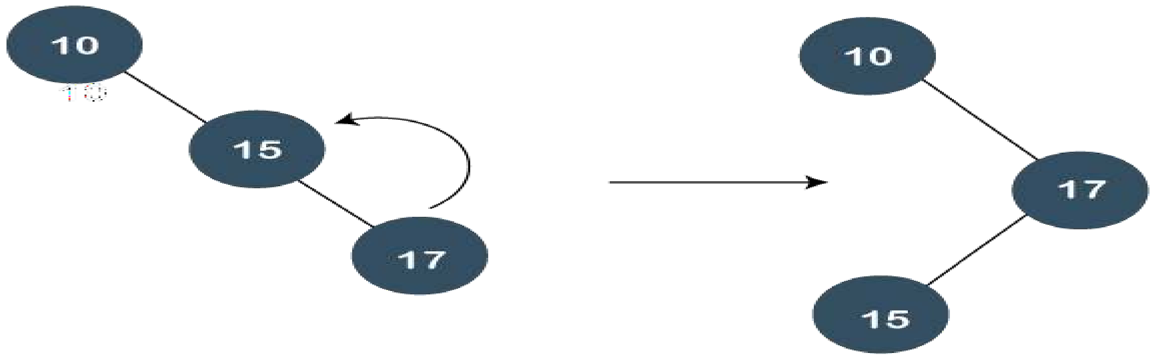
Step 2: The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

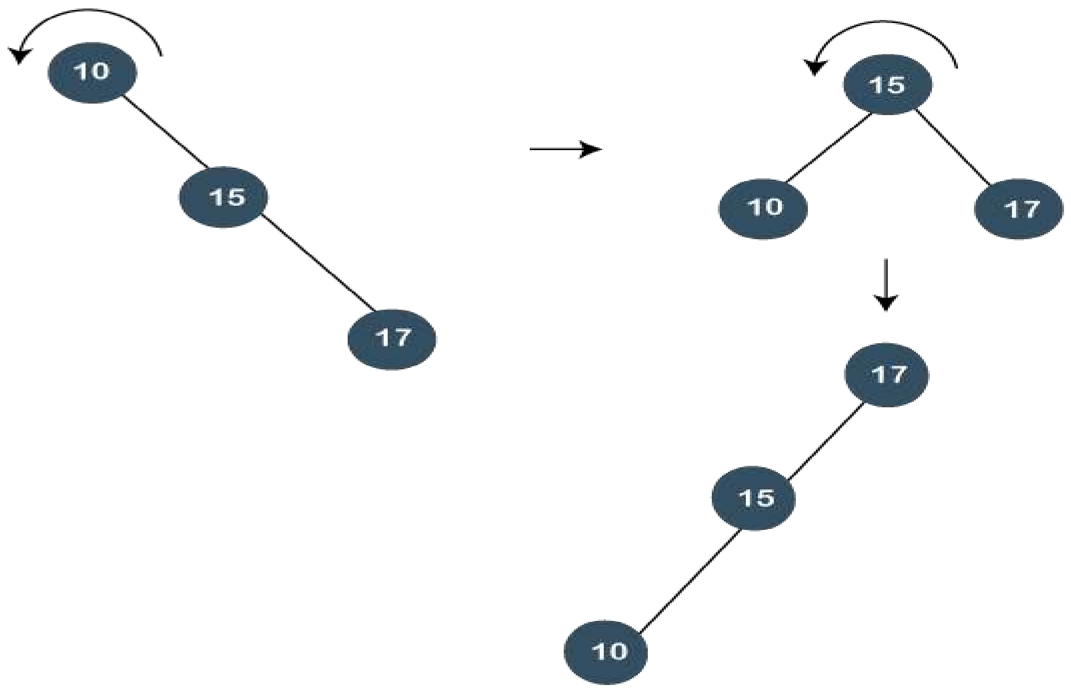
Now, we perform ***splaying***. To make 10 as a root node, we will perform the right rotation, as shown below:



Step 3: The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zig zig rotations

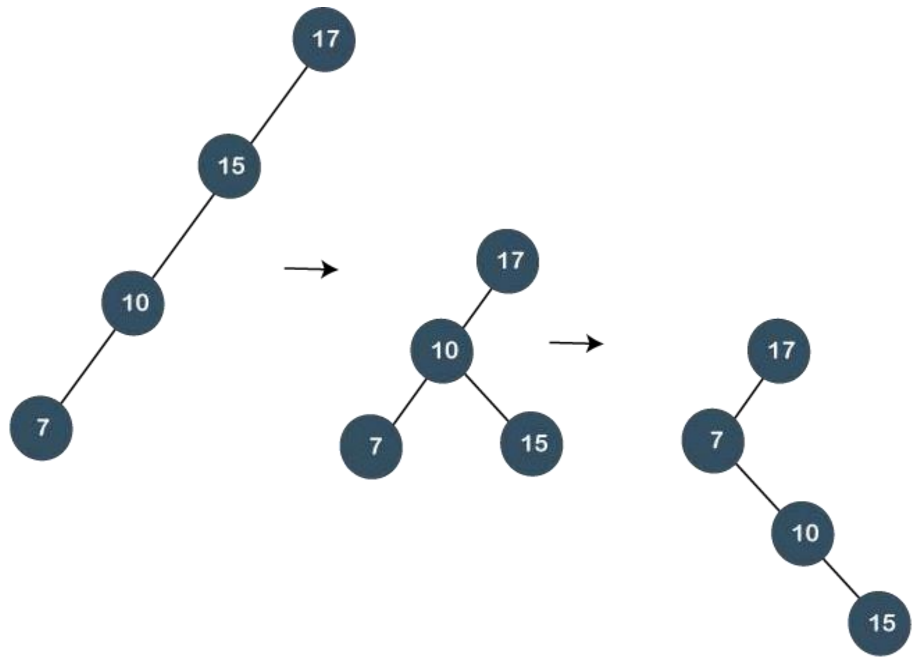




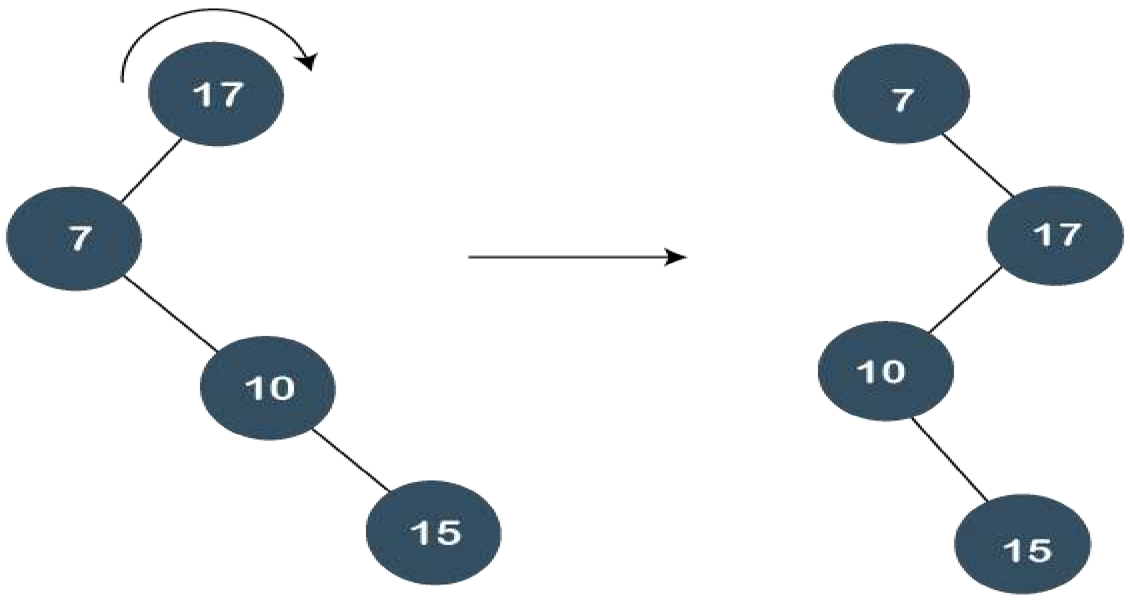
In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

Step 4: The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:



Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:



Deletion in Splay tree

As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

Types of Deletions:

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

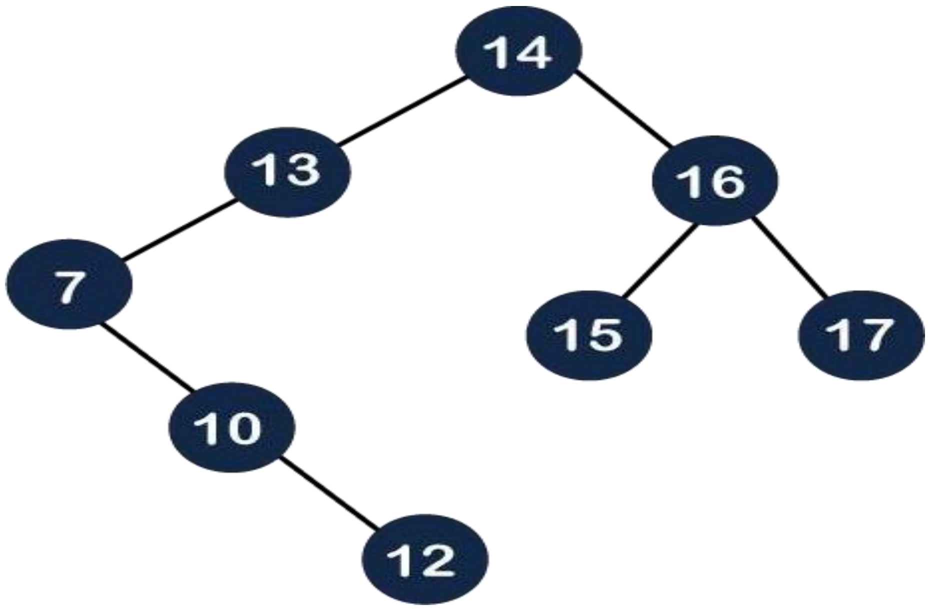
Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the deleted node.

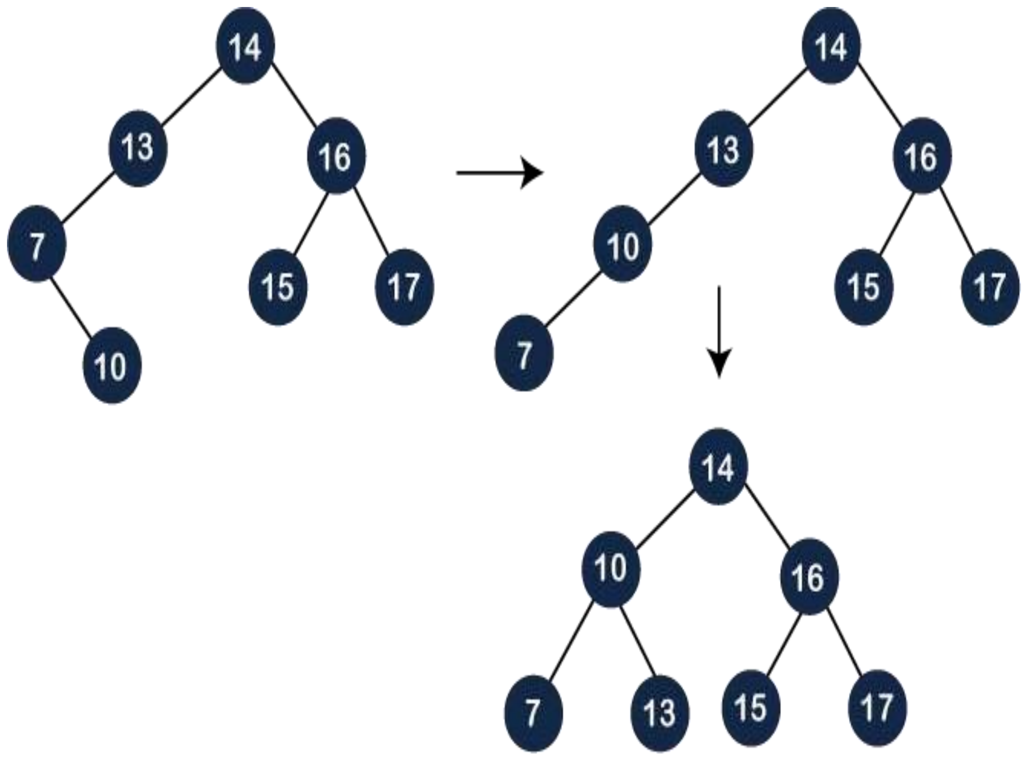
Let's understand the deletion in the Splay tree.

Suppose we want to delete 12, 14 from the tree shown below:

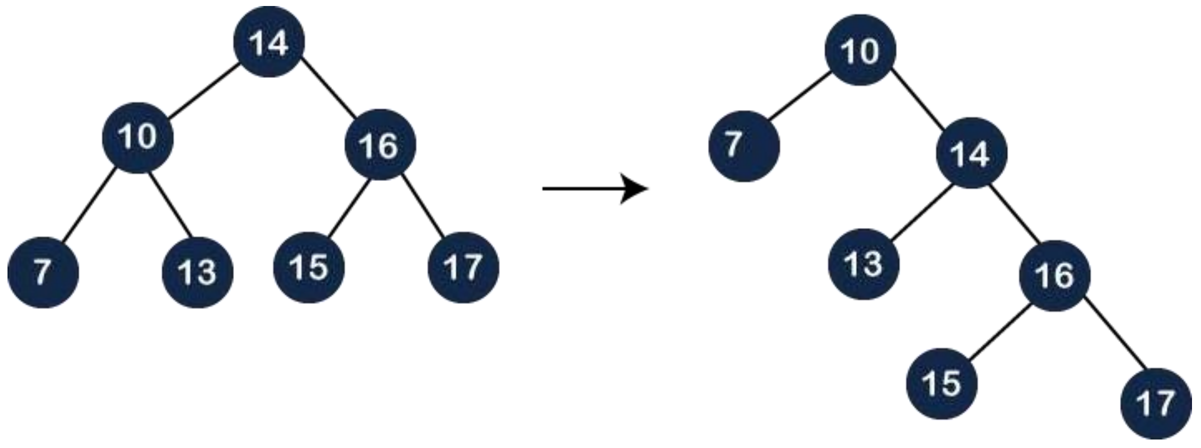
- First, we simply perform the standard BST deletion operation to delete 12 element. As 12 is a leaf node, so we simply delete the node from the tree.



The deletion is still not completed. We need to splay the parent of the deleted node, i.e., 10. We have to perform ***Splay(10)*** on the tree. As we can observe in the above tree that 10 is at the right of node 7, and node 7 is at the left of node 13. So, first, we perform the left rotation on node 7 and then we perform the right rotation on node 13, as shown below:

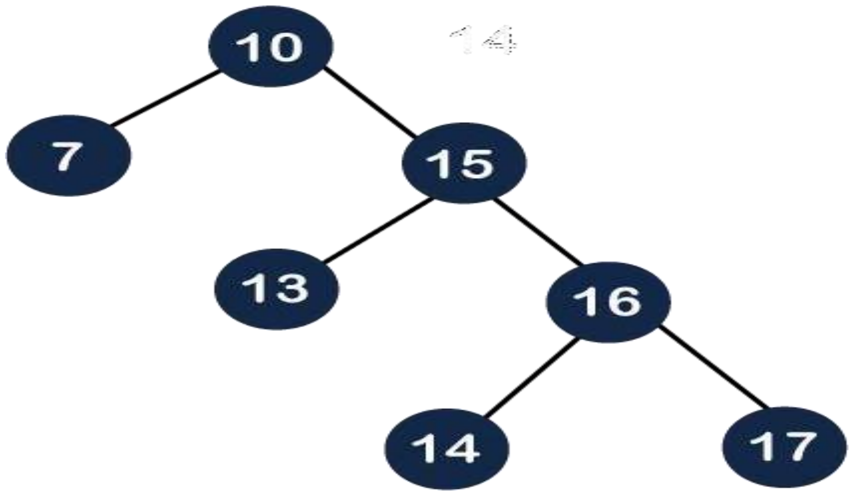


Still, node 10 is not a root node; node 10 is the left child of the root node. So, we need to perform the right rotation on the root node, i.e., 14 to make node 10 a root node as shown below:

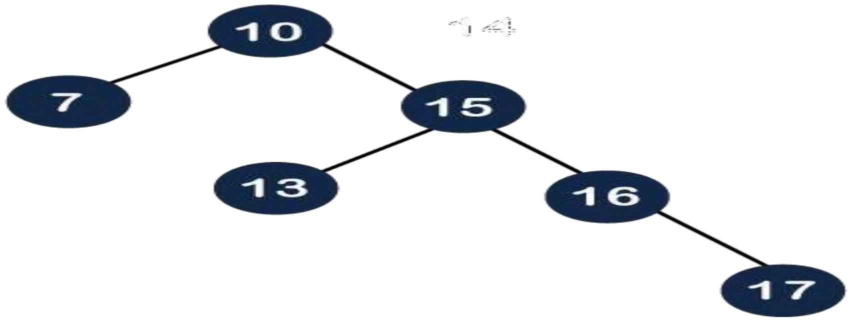


- Now, we have to delete the 14 element from the tree, which is shown below:

As we know that we cannot simply delete the internal node. We will replace the value of the node either using ***inorder predecessor*** or ***inorder successor***. Suppose we use inorder successor in which we replace the value with the lowest value that exist in the right subtree. The lowest value in the right subtree of node 14 is 15, so we replace the value 14 with 15. Since node 14 becomes the leaf node, so we can simply delete it as shown below:



Still, the deletion is not completed. We need to perform one more operation, i.e., splaying in which we need to make the parent of the deleted node as the root node. Before deletion, the parent of node 14 was the root node, i.e., 10, so we do need to perform any splaying in this case.

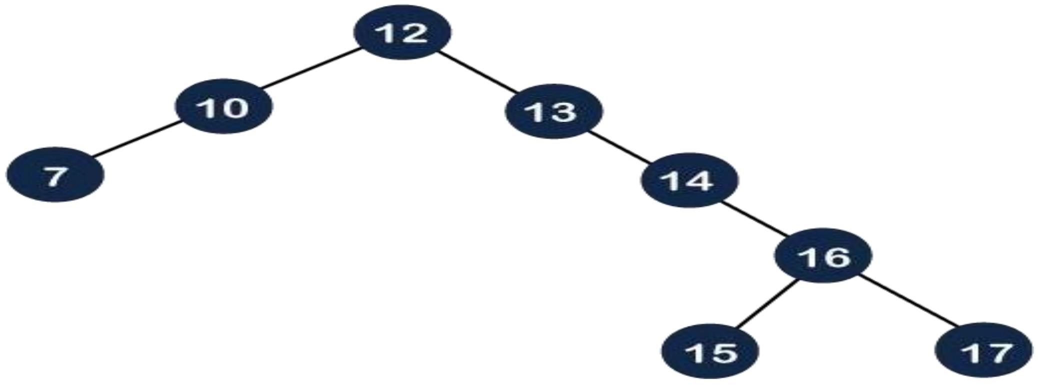


Top-down splaying

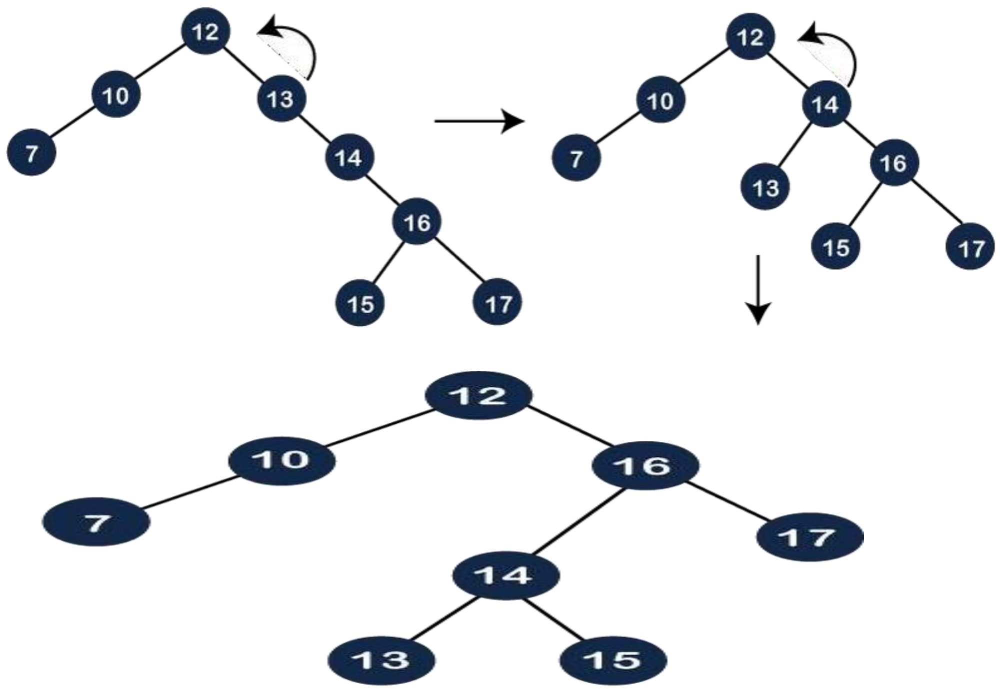
In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

Let's understand the top-down splaying through an example.

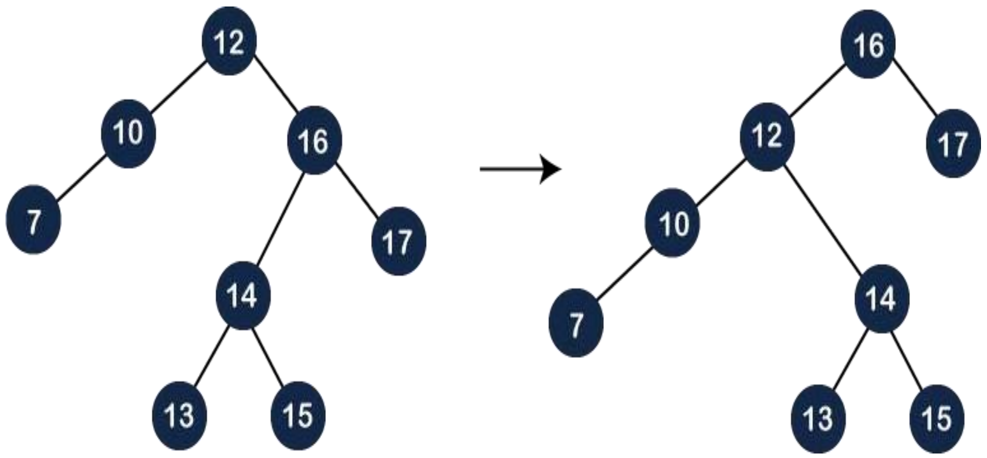
Suppose we want to delete 16 from the tree which is shown below:



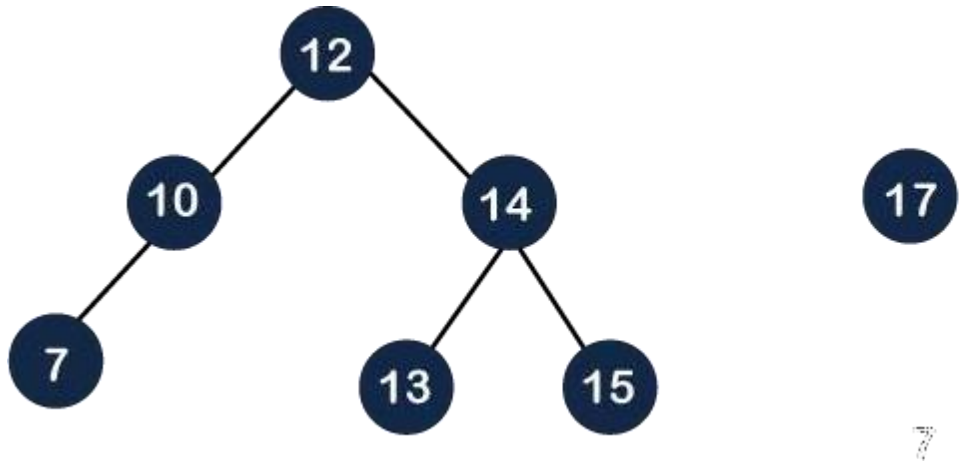
Step 1: In top-down splaying, first we perform splaying on the node 16. The node 16 has both parent as well as grandparent. The node 16 is at the right of its parent and the parent node is also at the right of its parent, so this is a zag zag situation. In this case, first, we will perform the left rotation on node 13 and then 14 as shown below:



The node 16 is still not a root node, and it is a right child of the root node, so we need to perform left rotation on the node 12 to make node 16 as a root node.

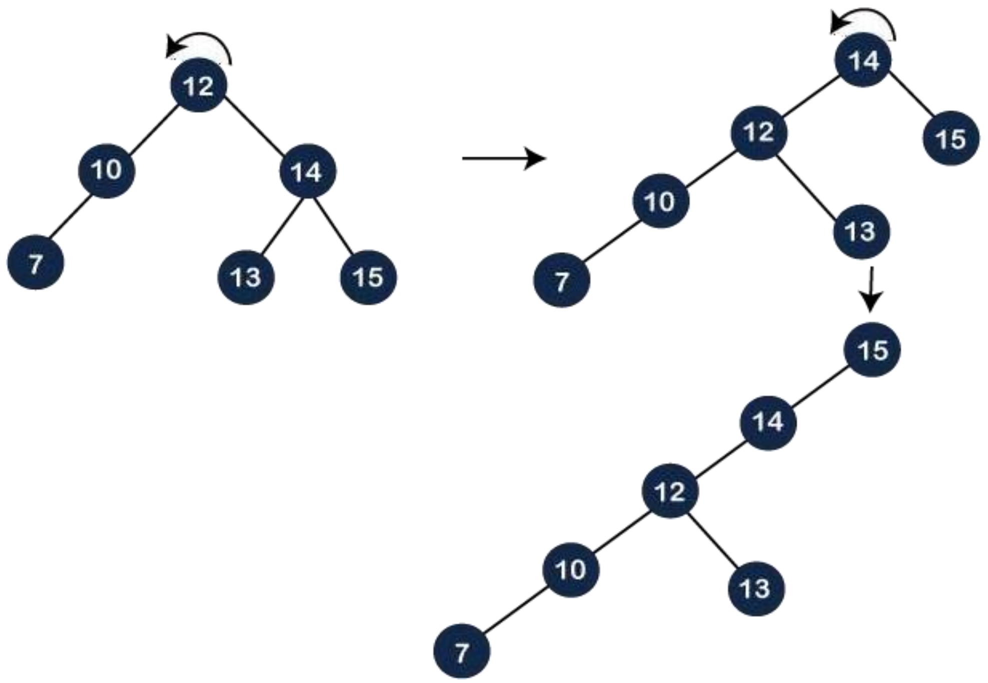


Once the node 16 becomes a root node, we will delete the node 16 and we will get two different trees, i.e., left subtree and right subtree as shown below:

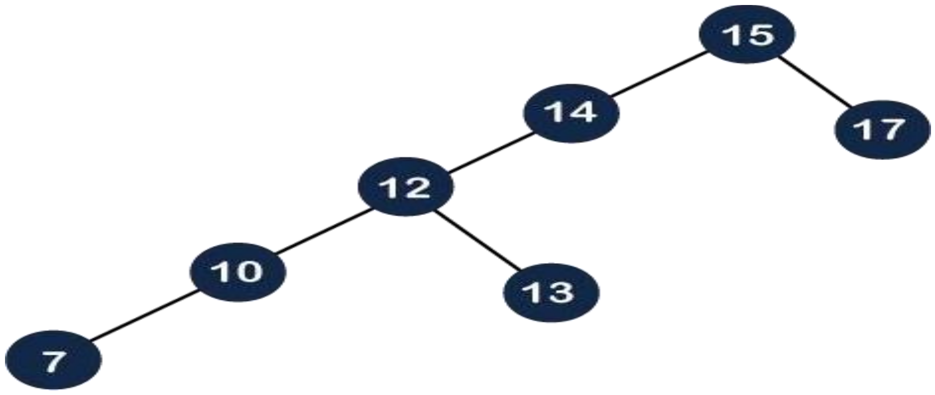


As we know that the values of the left subtree are always lesser than the values of the right subtree. The root of the left subtree is 12 and the root of the right subtree is 17. The first step is tofind the maximum element in the left subtree. In the left subtree, the maximum element is 15,and then we need to perform splaying operation on 15.

As we can observe in the above tree that the element 15 is having a parent as well as a grandparent. A node is right of its parent, and the parent node is also right of its parent, so we need to perform two left rotations to make node 15 a root node as shown below:



After performing two rotations on the tree, node 15 becomes the root node. As we can see, the right child of the 15 is NULL, so we attach node 17 at the right part of the 15 as shown below, and this operation is known as a ***join*** operation.


15
14
17
12
10
13
7

lOMoARcPSD|16210324

UNIT IV

UNIT IV
Graphs: Graph Implementation Methods. Graph Traversal Methods. (DFS,BFS)
Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort

Introduction to Graphs

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

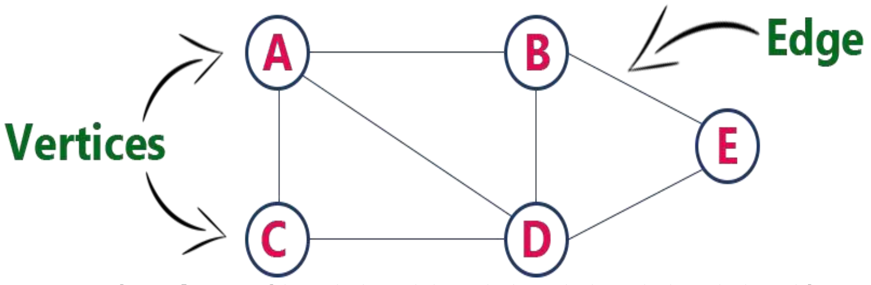
Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)



Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.

DFS (A-

BFS

Graph Terminology

We use the following terms in graph data structure...

Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge

between vertices A and B then edge (A , B) is equal to edge (B , A).

2. **Directed Edge -** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is a edge with value (cost) on it.

Undirected Graph
A graph with only undirected edges is said to be undirected graph.

Directed Graph
A graph with only directed edges is said to be directed graph.

Mixed Graph
A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints
The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin
If a edge is directed, its first endpoint is said to be the origin of it.

Destination
If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent
If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident
Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge
A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge
A directed edge is said to be incoming edge on its destination vertex.

Degree
Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree
Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree
Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges
If there are two undirected edges with same end vertices and two directed edges with same origin

and destination, such edges are called parallel edges or multiple edges.

Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

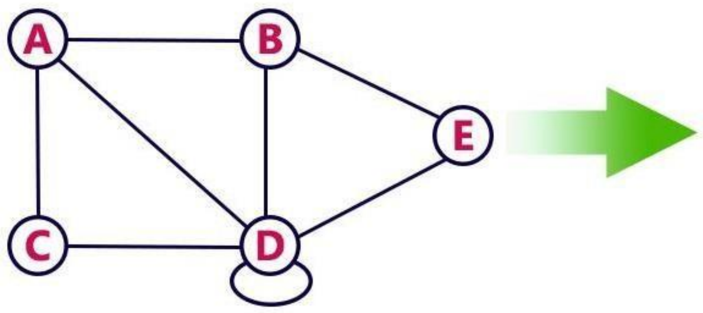
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

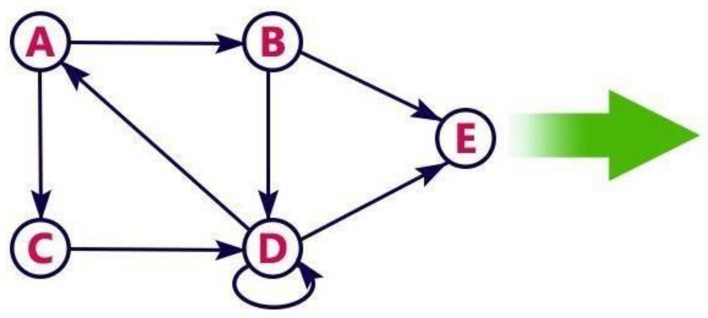
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



$$\begin{array}{c|ccccc|} & A & B & C & D & E \\ A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Directed graph representation...

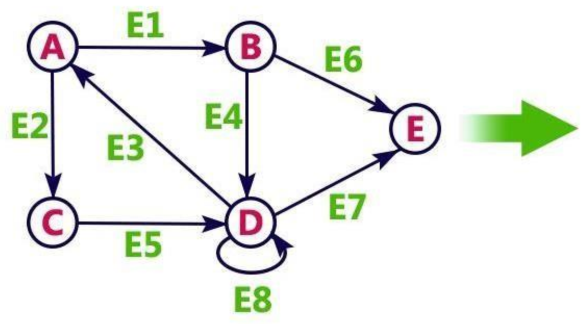


$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 1 & 1 & 0 & 0 \\
B & 0 & 0 & 0 & 1 & 1 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 0 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0
\end{array}$$

Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

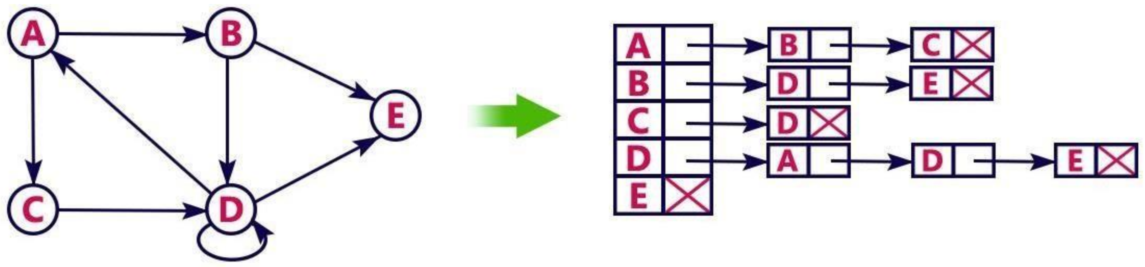


$$\begin{array}{c|cccccccc}
 & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\
\hline
A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\
E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0
\end{array}$$

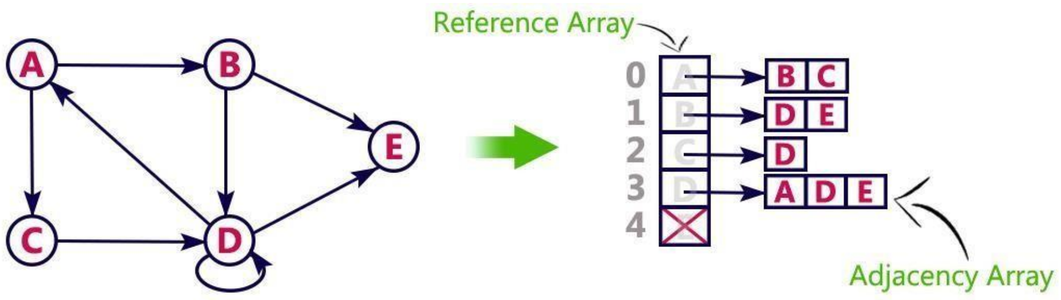
Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

- Step 1 - Define a Stack of size total number of vertices in the graph.
- Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

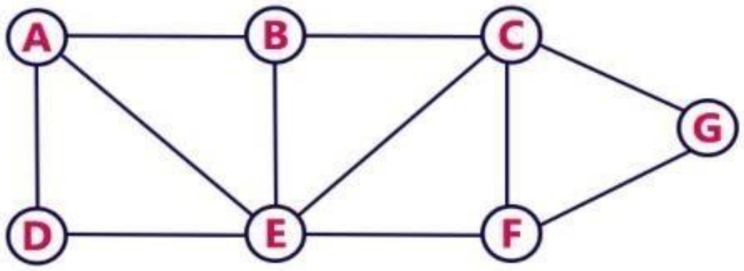
- Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

- Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
- Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.
- Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we reached the current vertex.

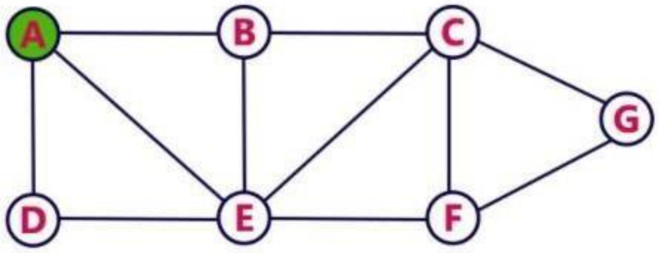
Example

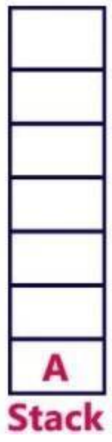
Consider the following example graph to perform DFS traversal



Step 1:

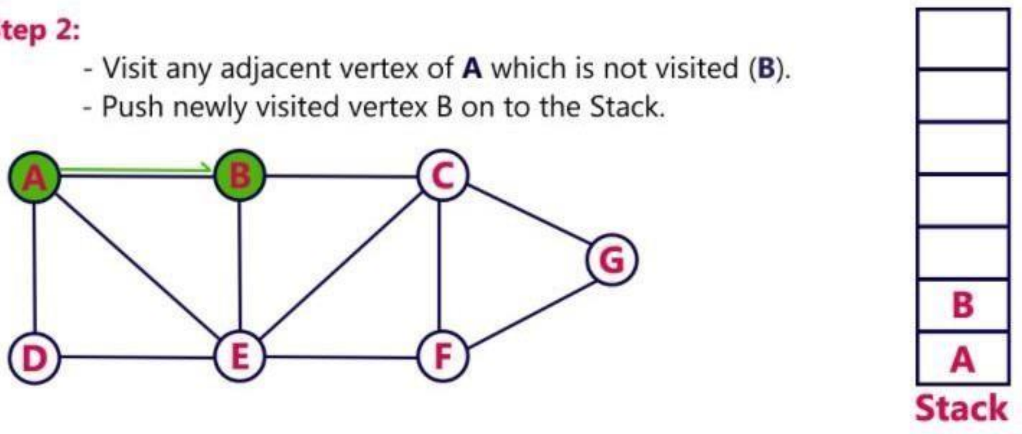
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.





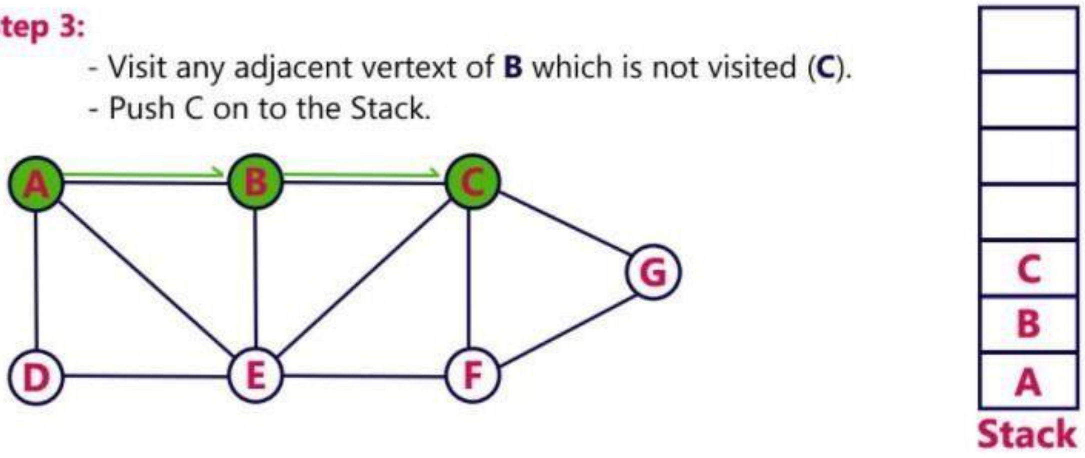
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



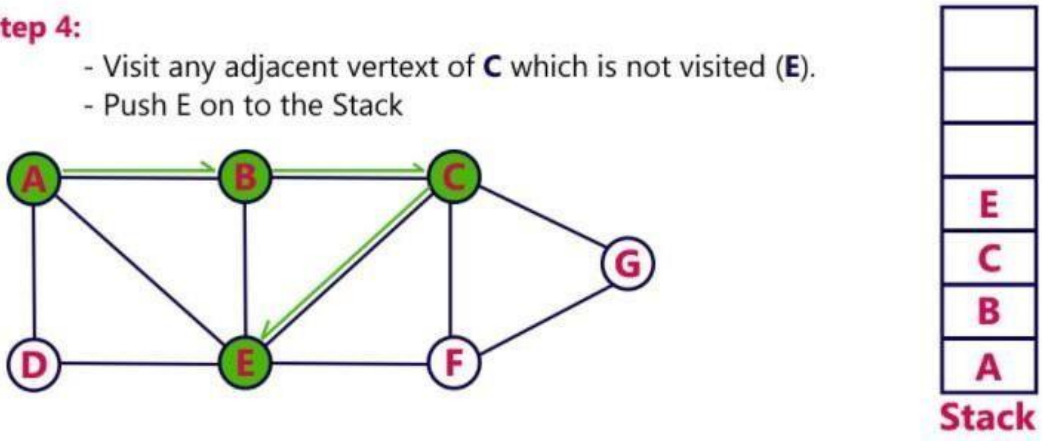
Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



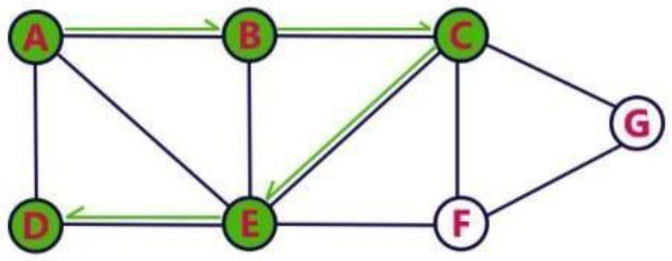
Step 4:

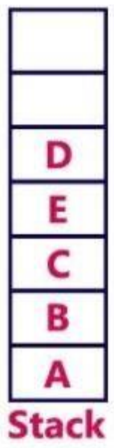
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



Step 5:

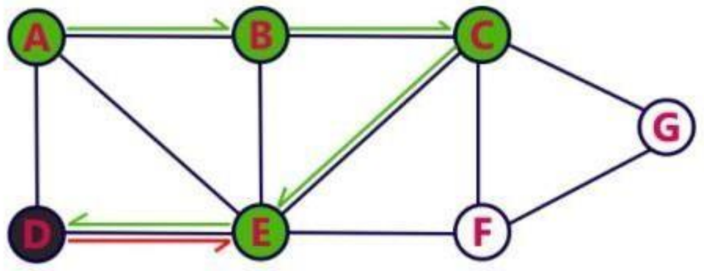
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

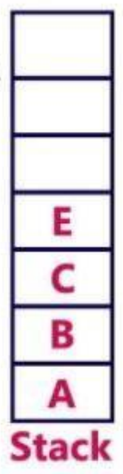




Step 6:

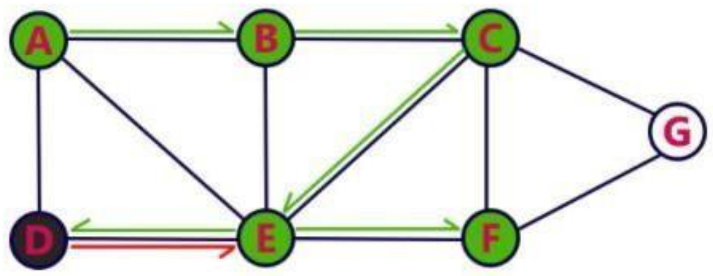
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

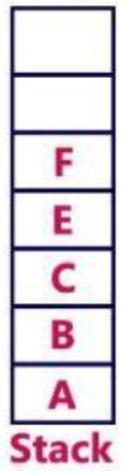




Step 7:

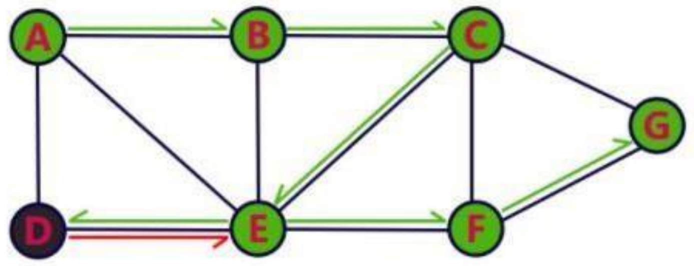
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

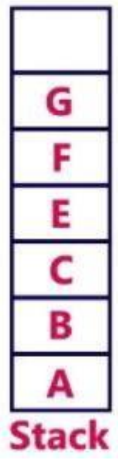




Step 8:

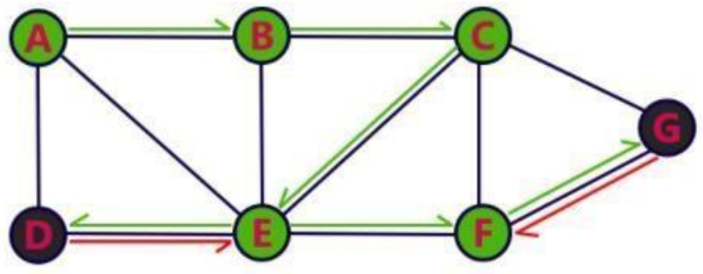
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

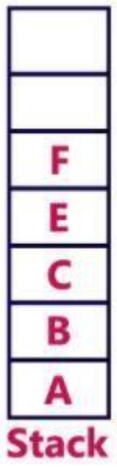




Step 9:

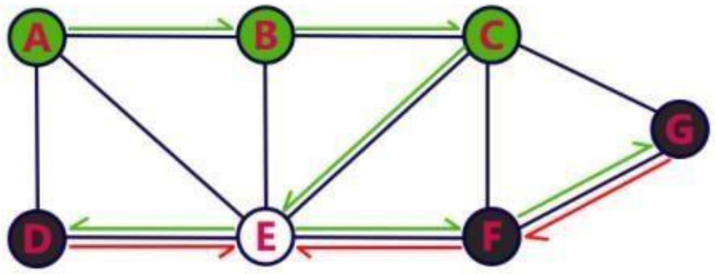
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

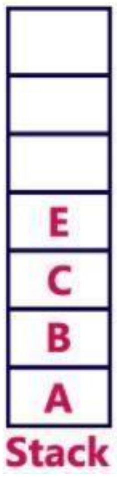




Step 10:

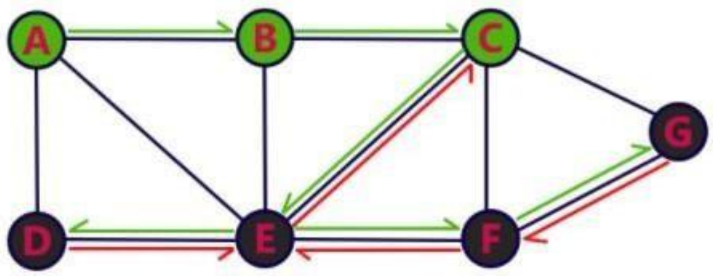
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

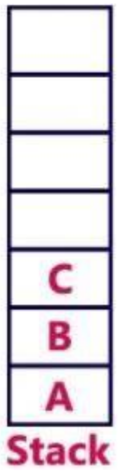




Step 11:

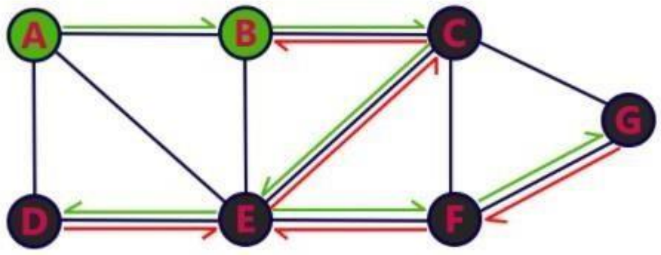
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

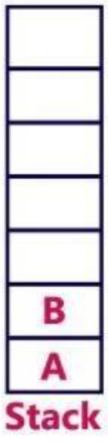




Step 12:

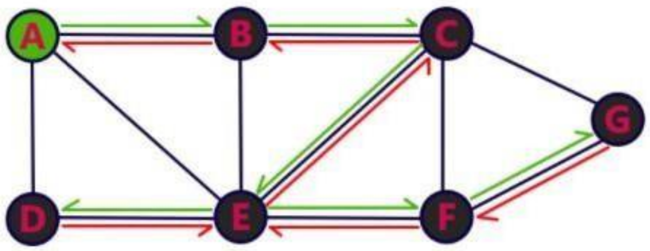
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

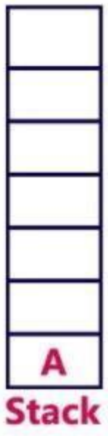




Step 13:

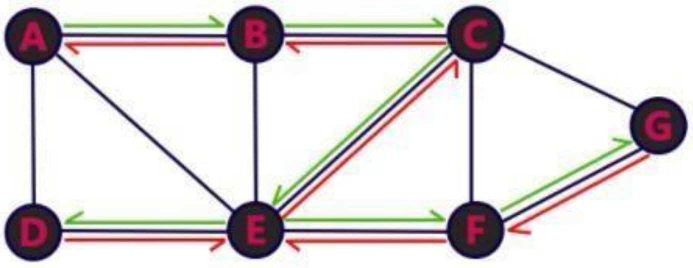
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

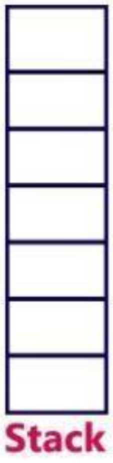




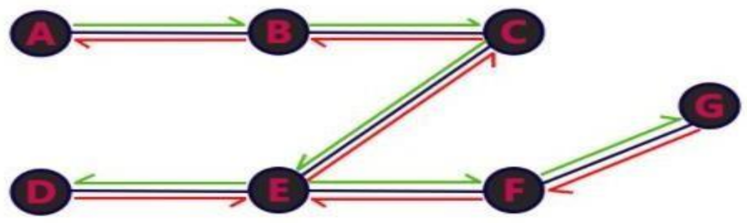
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

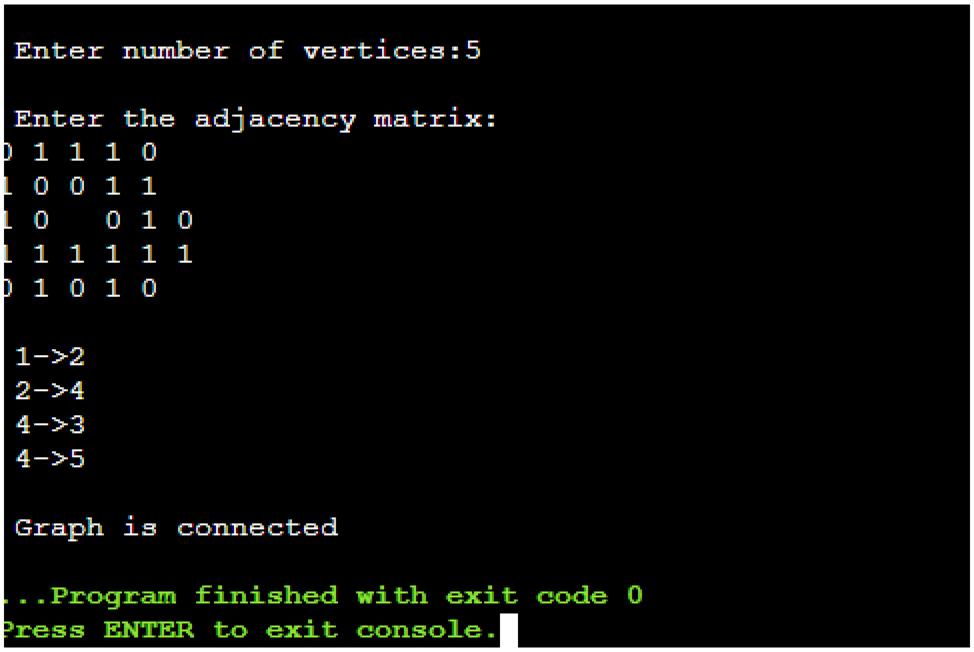


Program

```
#include<stdio.h>
#include<conio.h>
int a[20][20],reach[20],n;
void dfs(int v) {
        int i;
        reach[v]=1;
        for (i=1;i<=n;i++)
          if(a[v][i] && !reach[i]) {
                printf("\n %d->%d",v,i);
                dfs(i);
        }
}
void main()
{
        int i,j,count=0;
        printf("\n Enter number of vertices:");
        scanf("%d",&n);
        for (i=1;i<=n;i++) {
                reach[i]=0;
                for (j=1;j<=n;j++)
                  a[i][j]=0;
}
        printf("\n Enter the adjacency matrix:\n");
        for (i=1;i<=n;i++)
          for (j=1;j<=n;j++)
           scanf("%d",&a[i][j]);
        dfs(1);
        printf("\n");
        for (i=1;i<=n;i++) {
                if(reach[i])
```

```
                    count++;

        }
        if(count==n)
          printf("\n Graph is connected"); else
          printf("\n Graph is not connected");
}
```

OUTPUT:



```
 Enter number of vertices:5

 Enter the adjacency matrix:
0 1 1 1 0
1 0 0 1 1
1 0   0 1 0
1 1 1 1 1 1
0 1 0 1 0

 1->2
 2->4
 4->3
 4->5

 Graph is connected

...Program finished with exit code 0
Press ENTER to exit console.
```

BFS (Breadth First Search)

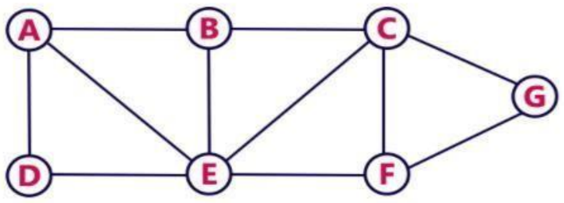
BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

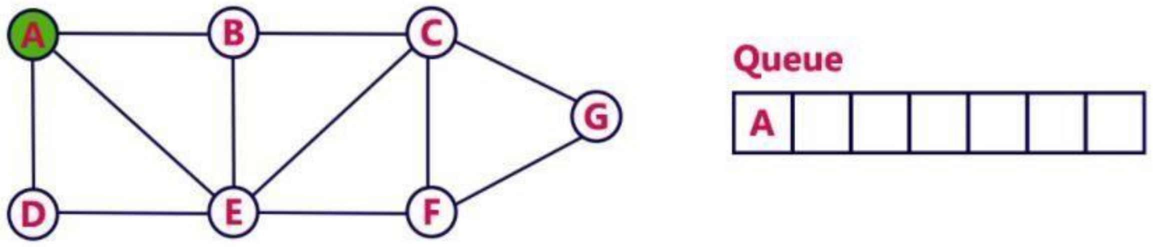
EXAMPLE

Consider the following example graph to perform BFS traversal



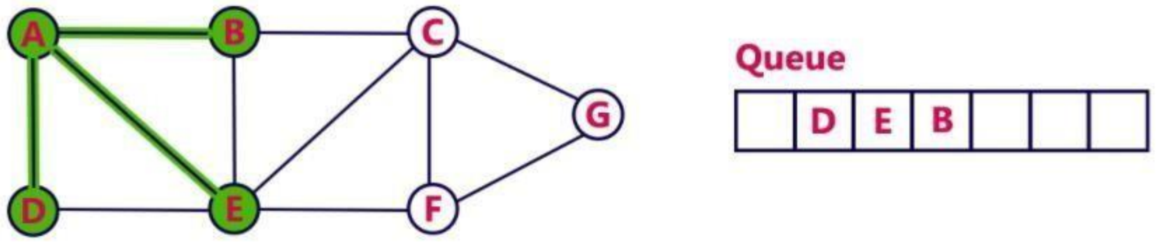
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



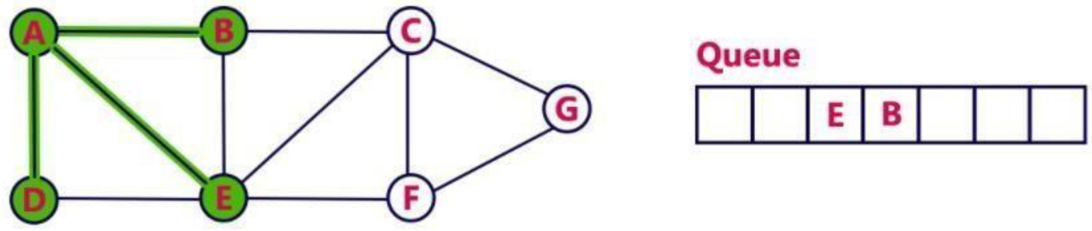
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



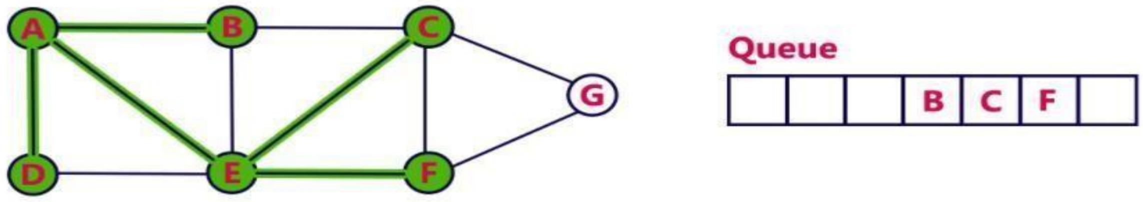
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



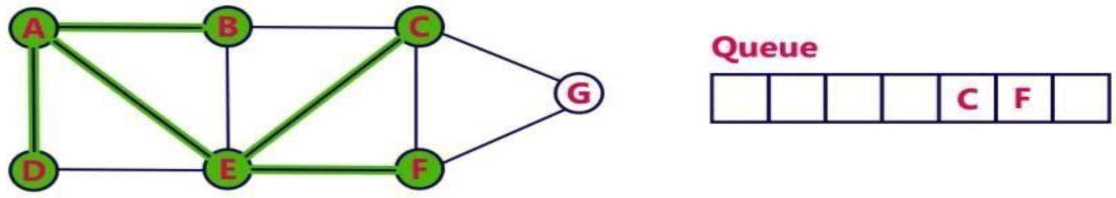
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



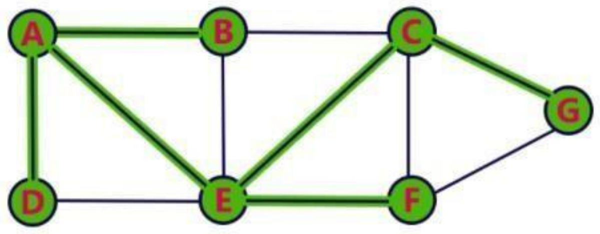
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

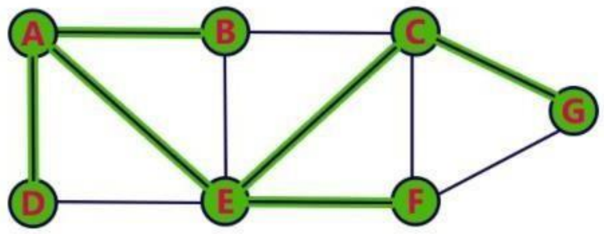


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

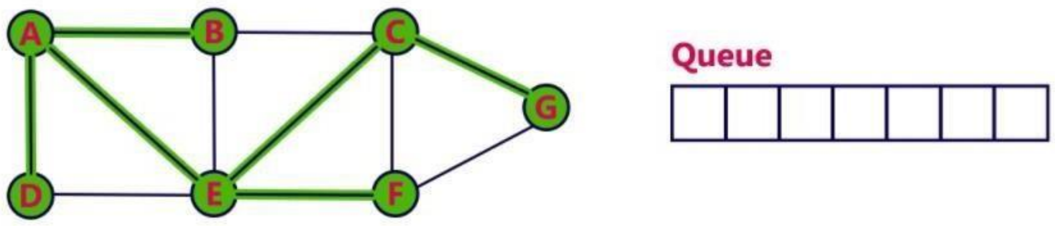


Queue

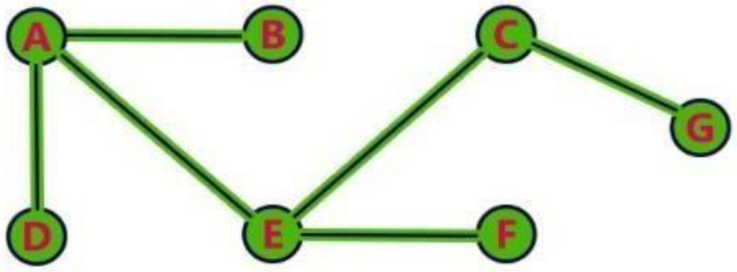
						G

Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



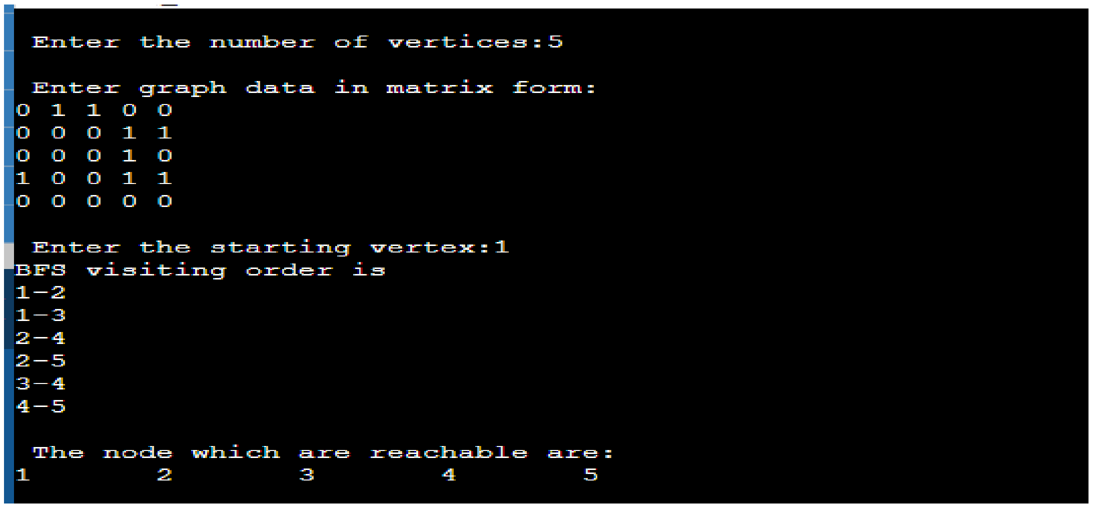
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



PROGRAM :

```
#include<stdio.h>
#include<conio.h>
int a[20][20],q[20],visited[20],n,i,j,f=0,r=-1;
void bfs(int v)
{    visited[v]=1;
            for (i=1;i<=n;i++)
            {
              if(a[v][i] && !visited[i])
              {
               printf("%d-%d\n",v,i);
               q[++r]=i;
              }
            }
            if(f<=r)
            {
                        visited[q[f]]=1;
                        bfs(q[f++]);
            }
}
void main()
{
            int v;
            printf("\n Enter the number of vertices:");
            scanf("%d",&n);
            for (i=1;i<=n;i++)
            {
                        q[i]=0;
                        visited[i]=0;
            }
           // GRAPH IS GIVEN AS ADJACENCY MATRIX
            printf("\n Enter graph data in matrix form:\n");
            for (i=1;i<=n;i++)
              for (j=1;j<=n;j++)
               scanf("%d",&a[i][j]);
            printf("\n Enter the starting vertex:");
            scanf("%d",&v);
            printf("BFS visiting order is\n");
            bfs(v);
            printf("\n The node which are reachable are:\n");
            for (i=1;i<=n;i++)
              if(visited[i])
               printf("%d\t",i);  else
               printf("\n Bfs is not possible");
}
```

OUTPUT :



```
 Enter the number of vertices:5

 Enter graph data in matrix form:
0 1 1 0 0
0 0 0 1 1
0 0 0 1 0
1 0 0 1 1
0 0 0 0 0

 Enter the starting vertex:1
BFS visiting order is
1-2
1-3
2-4
2-5
3-4
4-5

 The node which are reachable are:
1       2       3       4       5
```

Sorting: Heap Sort, External Sorting- Model for external sorting, Merge Sort

SORTING INTRODUCTION

Sorting is nothing but arranging the data in ascending or descending order.

The term **sorting** came into picture, as humans realized the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

The two main criteria to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering here.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Merge Sort
5. Heap Sort

Sorting Terminology

What is in-place sorting?

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.
For example, Insertion Sort and Selection Sorts are in-place sorting algorithms as they do not use any additional space for sorting the list and a typical implementation of Merge Sort is not in-place.

What are Internal and External Sorting?

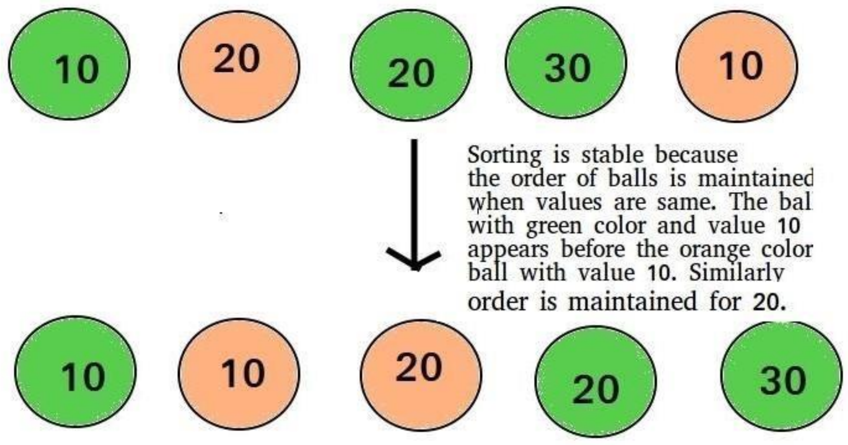
When all data that needs to be sorted cannot be placed in-memory at a time, the sorting is called external sorting. External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting. Some external storage like hard-disk, CD, etc is used for external storage.
When all data is placed in-memory, then sorting is called internal sorting.

Example of external sorting is Merge Sort.

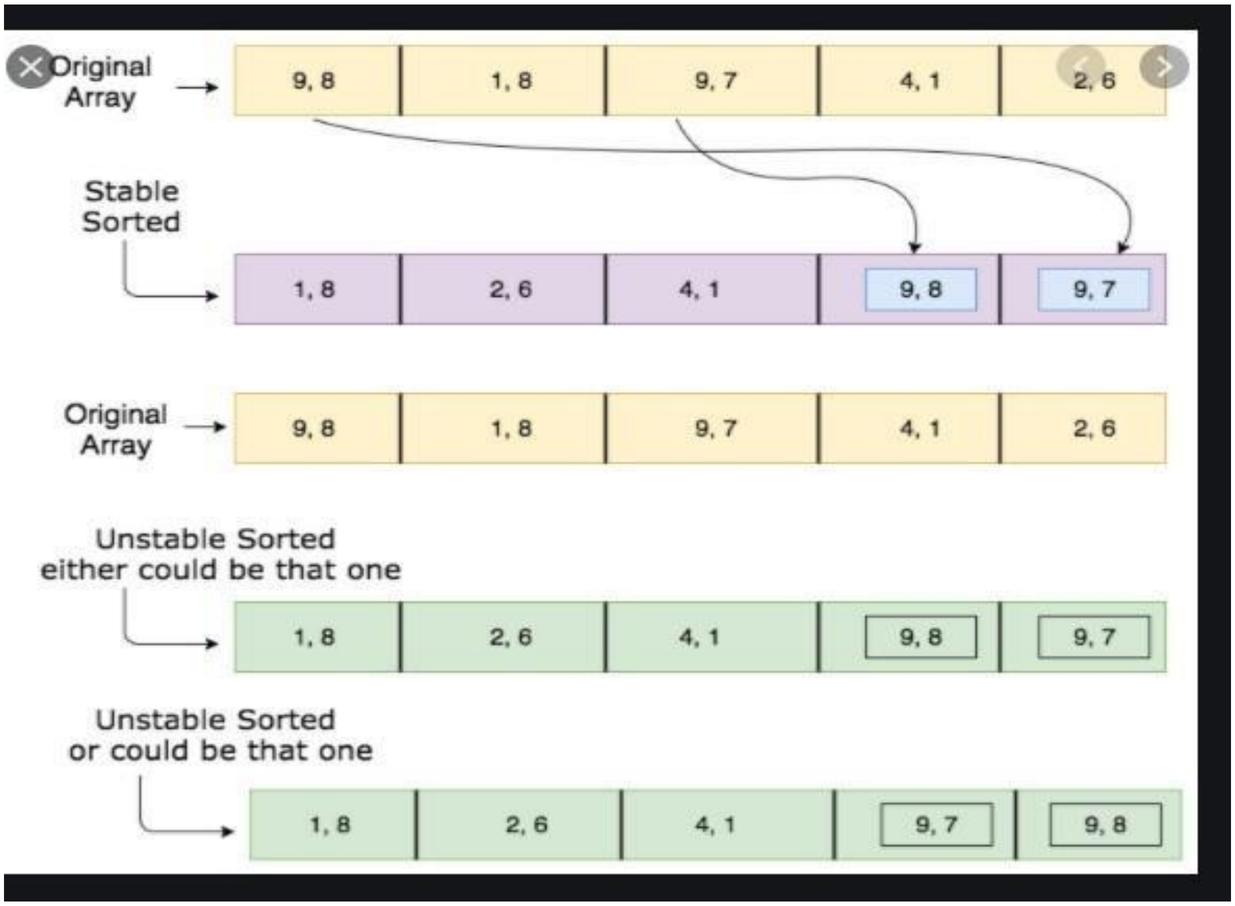
What is stable sorting?

Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values). And we wish to sort these objects by keys.

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.
Informally, stability means that equivalent elements retain their relative positions, after sorting.


10
20
20
30
10
Sorting is stable because the order of balls is maintained when values are same. The bal with green color and value 10 appears before the orange color ball with value 10. Similarly order is maintained for 20.
10
10
20
20
30

When equal elements are indistinguishable, such as with integers or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.



An example where it is useful

Consider the following dataset of Student Names and their respective class sections.

$$
\begin{aligned}
&(Dave, A)\\
&(Alice, B)\\
&(Ken, A)\\
&(Eric, B)\\
&(Carol, A)
\end{aligned}
$$

If we sort this data according to name only, then it is highly unlikely that the resulting dataset will be grouped according to sections as well.

$$\begin{array}{l}(Alice, B)\\(Carol, A)\\(Dave, A)\\(Eric, B)\\(Ken, A)\end{array}$$

So we might have to sort again to obtain list of students section wise too. But in doing so, i f the sorting algorithm is not stable, we might get a result like this-

$$\begin{array}{l}(Carol, A)\\(Dave, A)\\(Ken, A)\\(Eric, B)\\(Alice, B)\end{array}$$

The dataset is now sorted according to sections, but not according to names.
In the name-sorted dataset, the tuple (alice , B)was before (ERIC,B), but since the sorting algorithm is not stable, the relative order is lost.
If on the other hand we used a stable sorting algorithm, the result would be-

$$\begin{array}{l}(Carol, A)\\(Dave, A)\\(Ken, A)\\(Alice, B)\\(Eric, B)\end{array}$$

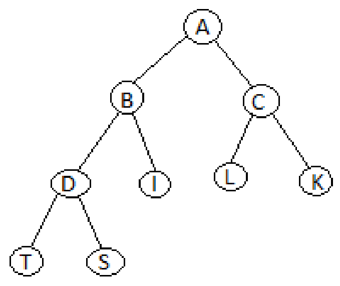
HEAP SORT

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.

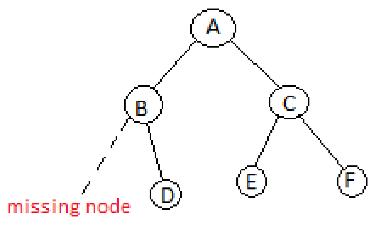
What is a Heap?

Heap is a special tree-based data structure that satisfies the following special heap properties:

1. Shape Property: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

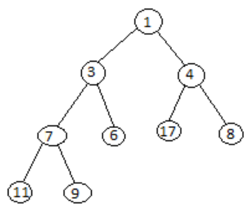


Complete Binary Tree



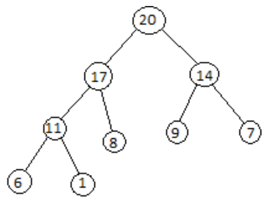
In-Complete Binary Tree

Heap Property: All nodes are either **greater than or equal to** or **less than or equal to** each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Building Heap from Array

Algorithm

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Example:

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}

Corresponding Complete Binary Tree is:

```
          1
        /    \
      3        5
    /   \    / \
   4     6 13 10
  / \   / \
 9 8 15 17
```

The task to build a Max-Heap from above array.

Total Nodes = 11.

Last Non-leaf node index = (11/2) - 1 = 4.

Therefore, last non-leaf node = 6.

To build the heap, heapify only the nodes:

[1, 3, 5, 4, 6] in reverse order.

Heapify 6: Swap 6 and 17.

```
            1
          /    \
        3        5
      /   \     / \
     4      17 13 10
   / \   / \
  9 8 15   6
```

Heapify 4: Swap 4 and 9.

```
            1
          /    \
        3        5
      /   \     / \
     9      17 13 10
   / \   / \
  4 8 15   6
```

Heapify 5: Swap 13 and 5.

```
            1
         /    \
       3        13
     /   \     / \
    9      17 5   10
   / \   / \
  4 8 15   6
```

Heapify 3: First Swap 3 and 17, again swap 3 and 15.

```
            1
         /    \
       17        13
     /   \     / \
    9      15  5  10
   / \   / \
  4  8 3  6
```

Heapify 1: First Swap 1 and 17, again swap 1 and 15, Finally swap 1 and 6.

```
            17
         /     \
     15      13
   /   \    / \
   9    6   5  10
  / \   / \
 4  8 3    1
```

Heap Sort Algorithm

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heap sort algorithm uses one of the tree concepts called **Heap Tree**. In this sorting algorithm, we use **Max Heap** to arrange list of elements in Descending order and **Min Heap** to arrange list elements in ascending order.

Step by Step Process

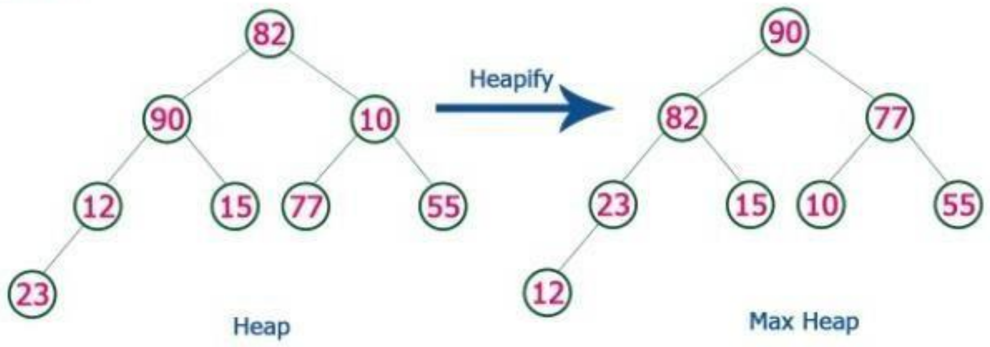
The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap (descending order/max heap (Ascending order).**
- **Step 3 -** Delete the root element from Min Heap/max heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap

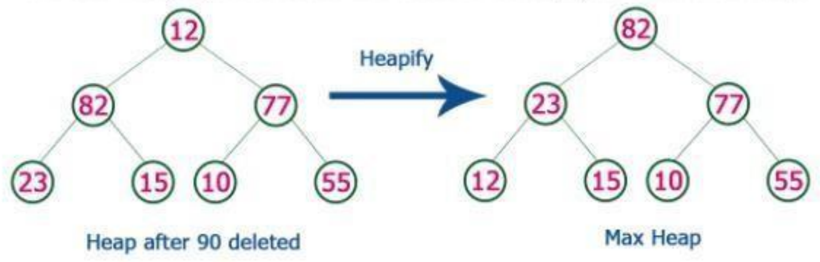


list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

90, 82, 77, 23, 15, 10, 55, 12

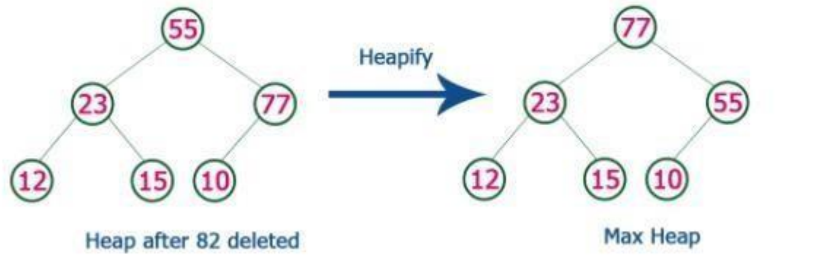
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

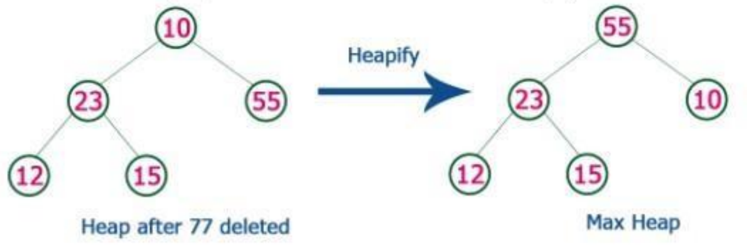
Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

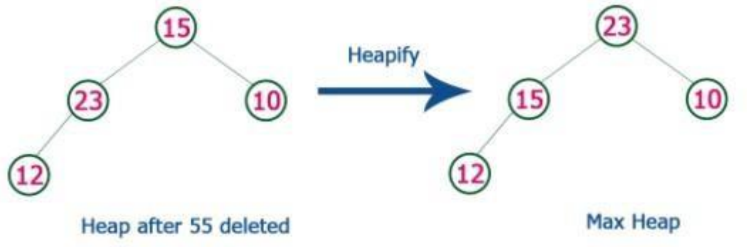
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77, 82, 90**

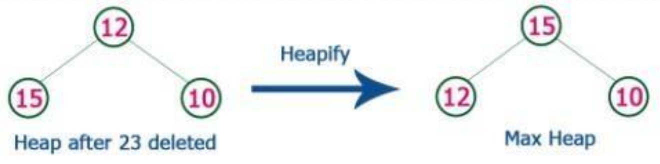
Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55, 77, 82, 90**

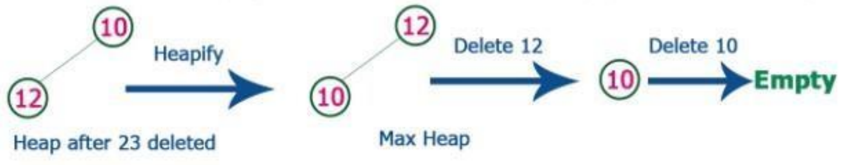
Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Note:

Heap sort is an in-place algorithm.

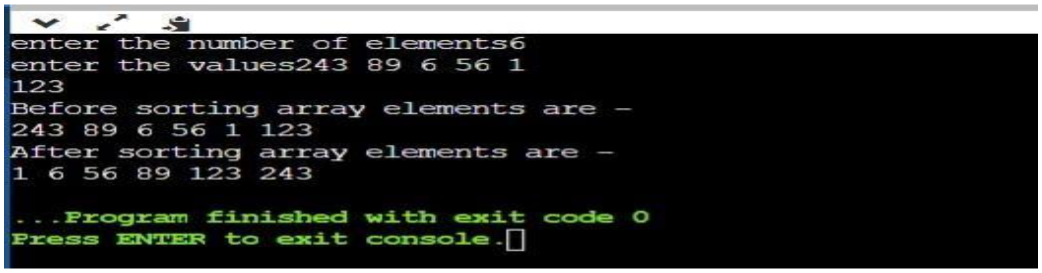
Its typical implementation is not stable, but can be made stable.

PROGRAM

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
   int largest = i; // Initialize largest as root
   int left = 2 * i + 1; // left child
   int right = 2 * i + 2; // right child
   // If left child is larger than root
   if (left < n && a[left] > a[largest])
      largest = left;
   // If right child is larger than root
   if (right < n && a[right] > a[largest])
      largest = right;
   // If root is not largest
   if (largest != i) {
      // swap a[i] with a[largest]
      int temp = a[i];
      a[i] = a[largest];
      a[largest] = temp;
      heapify(a, n, largest);
   }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
   for (int i = n / 2 - 1; i >= 0; i--)
      heapify(a, n, i);
   // One by one extract an element from heap
   for (int i = n - 1; i >= 0; i--) {
      /* Move current root element to end*/
      // swap a[0] with a[i]
      int temp = a[0];
```

```
        a[0] = a[i];
        a[i] = temp;
        heapify(a, i, 0);
    }
}
/* function to print the array elements */
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
    {
        printf("%d", arr[i]);
        printf(" ");
    }
}
int main()
{
    int a[100],n ;
    printf("enter the number of elements");
    scanf("%d",&n);
    printf("enter the values");
    for(int i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    heapSort(a, n);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);
    return 0;
}
```

Output:



```
enter the number of elements6
enter the values243 89 6 56 1
123
Before sorting array elements are -
243 89 6 56 1 123
After sorting array elements are -
1 6 56 89 123 243

...Program finished with exit code 0
Press ENTER to exit console.
```

Time Complexity:

Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

MERGE SORT

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

Before jumping on to, how merge sort works and its implementation, first let's understand what the rule of Divide and Conquer is?

Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

Let's take an example, **Divide and Rule**.

When Britishers s came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

Well that was history, and a socio-political policy (**Divide and Rule**), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

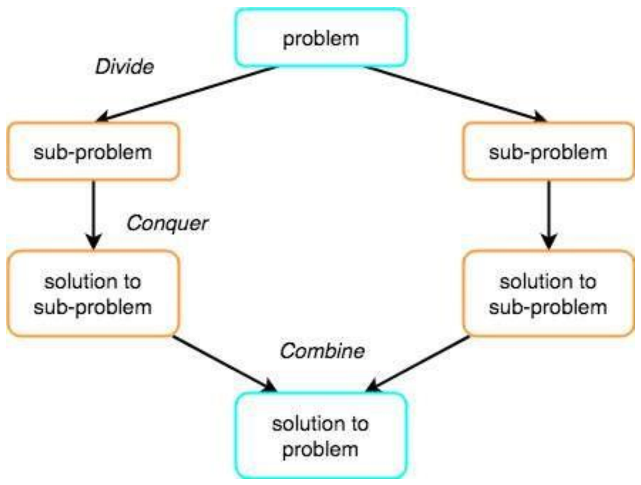
In **Merge Sort**, the given unsorted array with n elements is divided into n sub arrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these sub arrays, to produce new sorted sub arrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.

2. **Conquer** the sub problems by solving them. The idea is to break down the problem into atomic sub problems, where they are actually solved.

3. **Combine** the solutions of the sub problems to find the solution of the actual problem.



How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two sub arrays with 3 elements each.

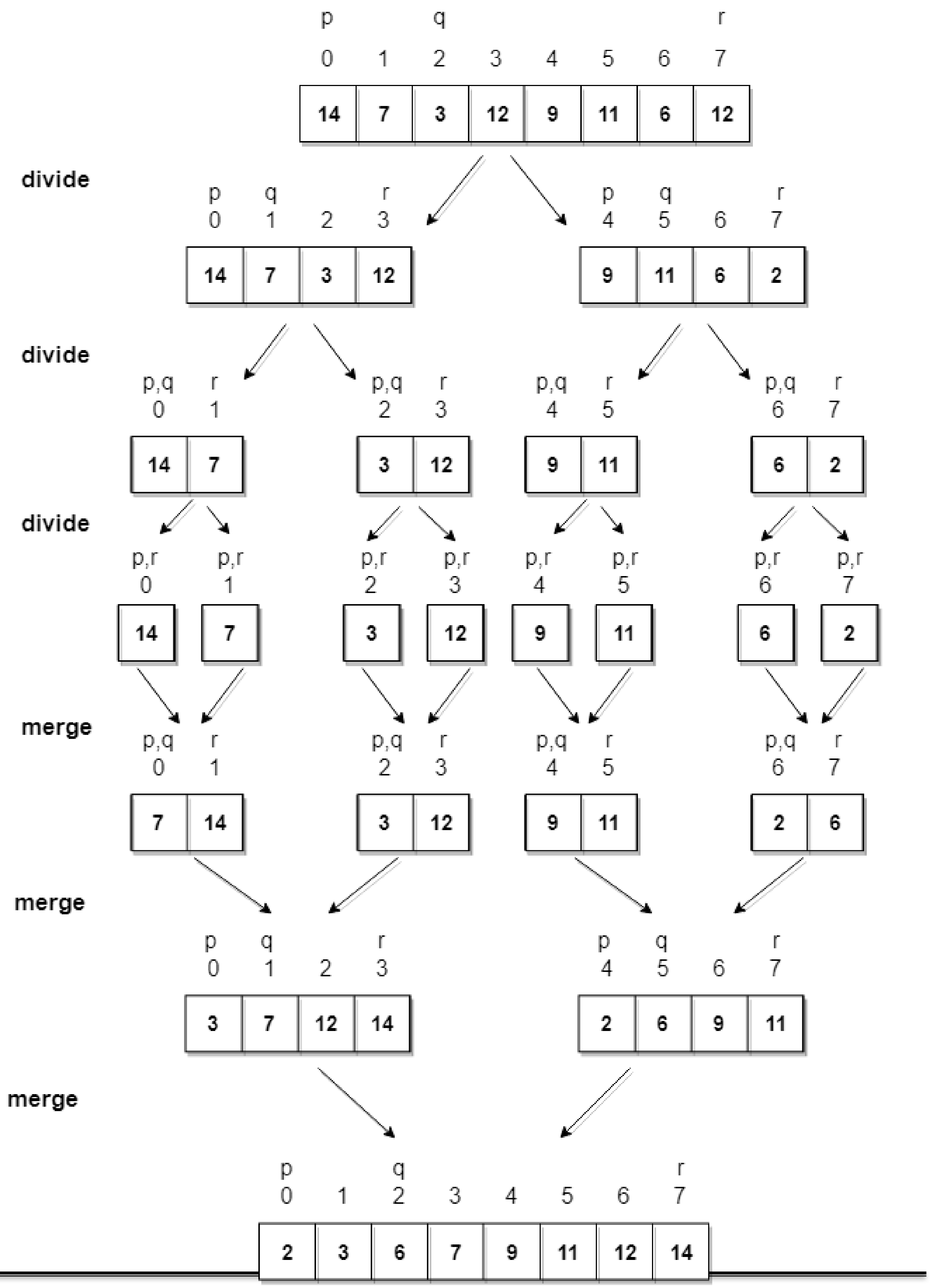
But breaking the original array into 2 smaller sub arrays is not helping us in sorting the array.

So we will break these sub arrays into even smaller sub arrays, until we have multiple sub arrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into sub arrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted sub arrays, step by step to form one single sorted array.

Let's consider an array with values {14, 7, 3, 12, 9, 11, 6, 12}

Below, we have a pictorial representation of how merge sort will sort the given array.

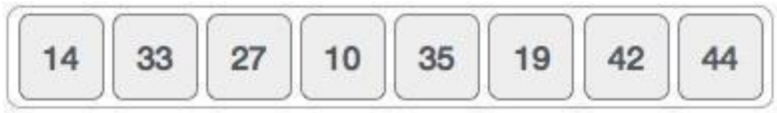

p
q
r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 12
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r 0 14
p,r 1 7
p,r 2 3
p,r 3 12
p,r 4 9
p,r 5 11
p,r 6 6
p,r 7 2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

In merge sort we follow the following steps:

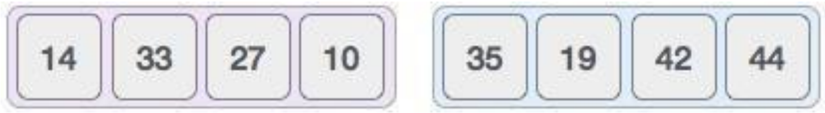
1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two sub arrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 sub arrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into sub arrays with single elements, then we start merging the sub arrays.

Example

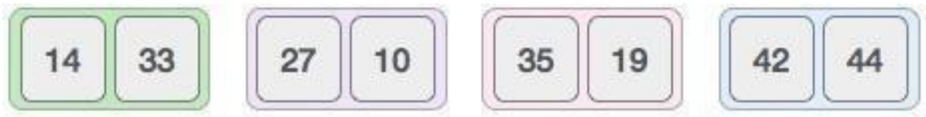
To understand merge sort, we take an unsorted array as the following –



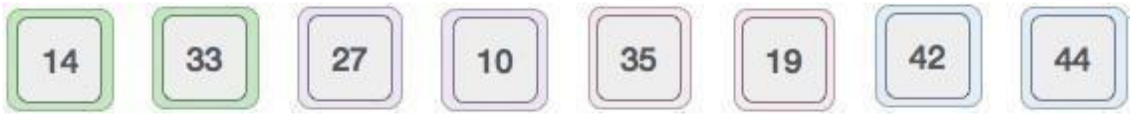
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

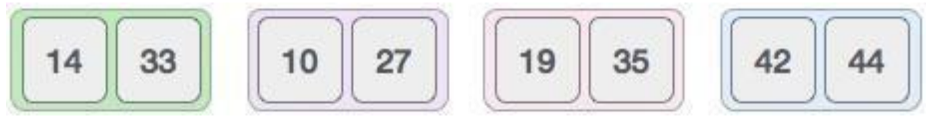


We further divide these arrays and we achieve atomic value which can no more be divided.

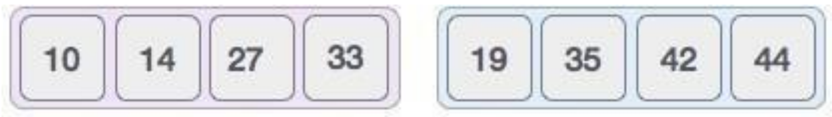


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

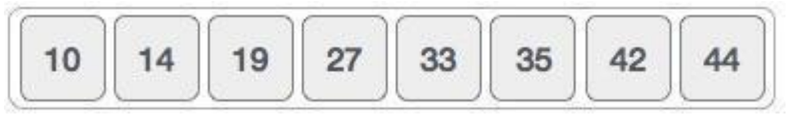
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this –



PROGRAM

```
#include <stdio.h>
void mergeSort(int [], int, int, int);
void partition(int [],int, int);
int main()
{
   int list[50];
   int i, size;
   printf("Enter total number of elements:");
   scanf("%d", &size);
   printf("Enter the elements:\n");
   for(i = 0; i < size; i++)
   {
      scanf("%d", &list[i]);
   }
```

```
    partition(list, 0, size - 1);
    printf("After merge sort:\n");
    for(i = 0;i < size; i++)
    {
        printf("%d  ",list[i]);
    }
   return 0;
}
void partition(int list[],int low,int high)
{
    int mid;
    if(low < high)
    {
        mid = (low + high) / 2;
        partition(list, low, mid);
        partition(list, mid + 1, high);
        mergeSort(list, low, mid, high);
    }
}
void mergeSort(int list[],int low,int mid,int high) {
    int i, mi, k, lo, temp[50];
    lo = low;
    i = low;
    mi = mid + 1;
    while ((lo <= mid) && (mi <= high))
    {
        if (list[lo] <= list[mi])
        {
            temp[i] = list[lo];
            lo++;
        }
        else
        {
            temp[i] = list[mi];
            mi++;
        }
        i++;  }
    if (lo > mid)
    {
       for (k = mi; k <= high; k++)
       {
          temp[i] = list[k];
          i++;
       }
    }
    else
    {
       for (k = lo; k <= mid; k++)
       {
           temp[i] = list[k];
           i++;
       }
    }

    for (k = low; k <= high; k++)
    {
       list[k] = temp[k];
    }
}
```

TIME COMPLEXITY

The time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as the merge sort always divides the array into two halves and takes linear time to merge two halves

COMPARISON OF SORTING TECHNIQUES

Insertion Sort

Properties:

- **INSERTION-SORT** can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are.
- In INSERTION-SORT, the **best case** occurs if the array is already sorted.

T [Best Case]= O(n)

- If the array is in reverse sorted order i.e. in decreasing order, INSERTION-SORT gives the **worst case** results.

T [Worst Case]= *o*(*n^2*)

- **Average Case**: When half the elements are sorted while half not
- The running time of insertion sort therefore belongs to both Ω(n) and O(n^2)

Pros:

- For nearly-sorted data, it's incredibly efficient (very near O(n) complexity)
- It works in-place, which means no auxiliary storage is necessary i.e. requires only a constant amount O(1) of additional memory space
- Efficient for (quite) small data sets.
- Stable, i.e. does not change the relative order of elements with equal keys

Cons:

- It is less efficient on list containing more number of elements
- Insertion sort needs a large number of element shifts

Merge Sort:

Properties

- Merge Sort's running time is **0(nlogn) in best, worst and average case**
- The **space complexity** of Merge sort is **O(n)**. This means that this algorithm takes a lot of space and May slower down operations for the last data sets.
- Merge sort is external sorting.

Pros:

- It is **quicker for larger lists** because unlike insertion it doesn't go through the whole list several times.
- The merge sort is **slightly faster than the heap sort** for larger sets
- ($nlogn$) worst case asymptotic complexity.
- Stable sorting algorithm
- Not a in-place sorting technique

Cons

- **Slower** comparative to the other sort algorithms **for smaller data sets**
- Marginally slower than quick sort in practice
- Goes through the whole process even if the list is sorted
- It uses more memory space to store the sub elements of the initial split list.
- It requires twice the memory of the heap sort because of the second array.

Insertion sort vs. Merge Sort

Similarity

- Both are comparison based sorting algorithms

Difference:

- To work on an almost sorted array, Insertion sort takes linear time i.e. O(n) while Merge takes O(n*logn) complexity to sort

Heap Sort

Properties:

- Heap sort involves **building a Heap data structure** from the given array and then **utilizing the Heap to sort the array**
- Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled
- A.heap_size of an array is initially the size of the array. At first iteration, after exchanging root of the max_heap tree (A[1]) with A[i] = A[A.length] (last element inside array A)
- Doing extract_max(), A.heap_size value will be decreased by 1
- max_heap structure should be max_heapified: A[Parent(i)] >= A[i], where Parent(i) returns i/2 of heap tree.
- Initially create a Heap. extract_max(), put element of the heap in the array until we have the complete sorted list in our array.
- Time complexity of heap sort is o(nlogn) in all the cases
- The Heap Sort sorting algorithm seems to have a worst case complexity of O(n log(n))
- Heap sort is in place sorting techniques.

Pros:

- Heap sort and merge sort are asymptotically optimal comparison sorts

Cons: N/A

Heap Sort vs. Merge Sort:

- The time required to merge in a merge sort is counterbalanced by the time required to build the heap in heap sort
- **Heap Sort is better :**

The Heap Sort sorting algorithm uses O(1) space for the sorting operation while Merge Sort which takes O(n) space

- **Merge Sort is better**
  * The merge sort is slightly faster than the heap sort for larger sets
  * Heap sort is not stable because operations on the heap can change the relative order of equal items.

Heap Sort vs. Insertion Sort:

Similarity

- Heap sort and insertion sort are both used comparison based sorting technique

Differences

- Heap Sort is not stable whereas Insertion Sort is.
- When already sorted, Insertion Sort will not sort every element again where as Heap Sort will use extract max and heapify again and again When already sorted, Insertion Sort takes O(n) TC whereas Heap Sort will take O(n log(n)) time Insertion Sort is not efficient for large input data whereas Heap Sort is.

lOMoARcPSD|16210324

UNIT V

UNIT - V

Pattern Matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

Pattern Matching

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

A typical problem statement would be-
Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

Examples:

Input: txt[] = "THIS IS A TEST TEXT"

pat[] = "TEST"

Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

Different Types of Pattern Matching Algorithms

1. Navie Based Algorithm or Brute Force Algorithm
2. Boyer Moore Algorithm
3. Knuth-Morris Pratt (KMP) Algorithm

Navie Based Algorithm or Brute Force Algorithm

When we talk about a string matching algorithm, every one can get a simple string matching technique. That is starting from first letters of the text and first letter of the pattern check whether these two letters are equal. if it is, then check second letters of the text and pattern. If it is not equal, then move first letter of the pattern to the second letter of the text. then check these two letters. this is the simple technique everyone can thought.

Brute Force string matching algorithm is also like that. Therefore we call that as Naive string matching algorithm. Naive means basic.

Brute Force Algorithm

```
do
    if (text letter == pattern letter)
        compare next letter of pattern to next letter of text
    else
        move pattern down text by one letter
while (entire pattern found or end of text)
```

Lets learn this method using an example.

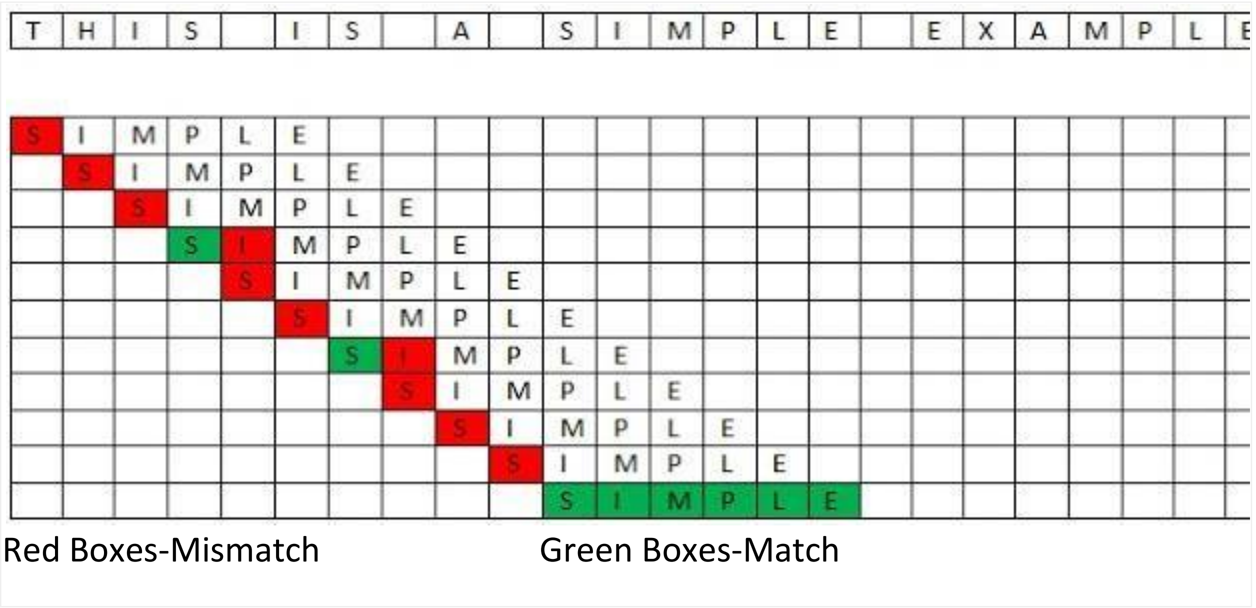
EXAMPLE 1

Let our text (T) as,

THIS IS A SIMPLE EXAMPLE

and our pattern (P) as,

SIMPLE

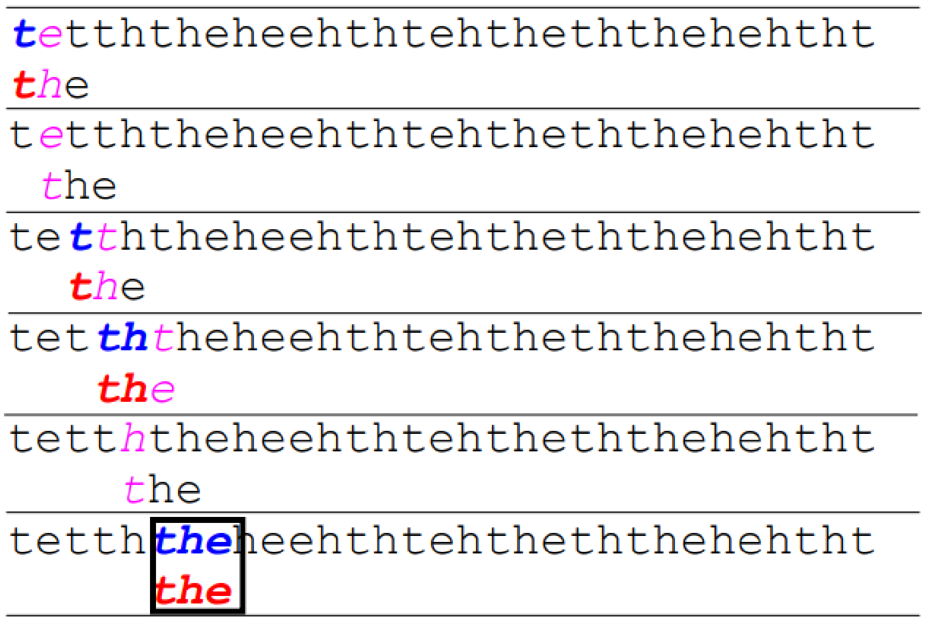


In above red boxes says mismatch letters against letters of the text and green boxes says match letters against letters of the text. According to the above

In first raw we check whether first letter of the pattern is matched with the first letter of the text. It is mismatched, because "S" is the first letter of pattern and "T" is the first letter of text. Then we move the pattern by one position. Shown in second raw.

Then check first letter of the pattern with the second letter of text. It is also mismatched. Likewise we continue the checking and moving process. In fourth raw we can see first letter of the pattern matched with text. Then we do not do any moving but we increase testing letter of the pattern. We only move the position of pattern by one when we find mismatches. Also in last raw, we can see all the letters of the pattern matched with the some letters of the text continuously.

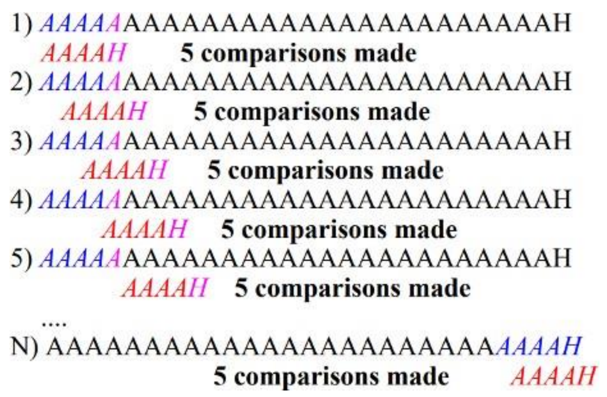
Example 2


tetththeheehthtehtheththeheh tht
the
tetththeheehthtehtheththeheh tht
the
tetththeheehthtehtheththeheh tht
the
tetththeheehthtehtheththeheh tht
the
tetththeheehthtehtheththeheh tht
the
tetththeheehthtehtheththeheh tht
the

Running Time Analysis Of Brute Force String Matching Algorithm

Worst Case

Given a pattern M characters in length, and a text N characters in length...
• Worst case: compares pattern to each substring of text of length M.
For example, M=5



```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
    AAAAH      5 comparisons made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
     AAAAH      5 comparisons made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
      AAAAH      5 comparisons made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
       AAAAH      5 comparisons made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
        AAAAH      5 comparisons made
 ....
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
            5 comparisons made      AAAAH
```

• Total number of comparisons: M (N-M+1) • Worst case time complexity: O(MN)

• Total number of comparisons: M (N-M+1)

• Worst case time complexity: O(MN)

Best case

Given a pattern M characters in length, and a text N characters in length...

• **Best case if pattern found**: Finds pattern in first M positions of text.

For example, M=5.

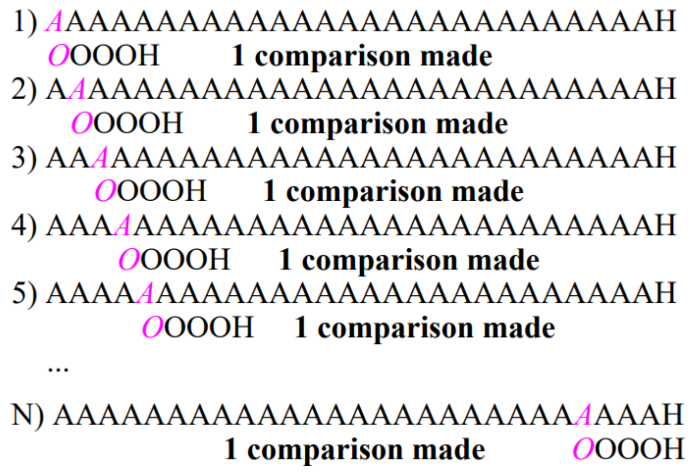
```
AAAAAAAAAAAAAAAAAAAAAAAAAAAH
AAAAA          5 comparisons made
```

• Total number of comparisons: M

• Best case time complexity: O(M)

Best case if pattern not found:

Always mismatch on first character. For example, M=5.



```
1) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
   OOOOH      1 comparison made
2) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
    OOOOH      1 comparison made
3) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
     OOOOH      1 comparison made
4) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
      OOOOH      1 comparison made
5) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
       OOOOH      1 comparison made
  ...
N) AAAAAAAAAAAAAAAAAAAAAAAAAAAH
           1 comparison made      OOOOH
```

• Total number of comparisons: N

• Best case time complexity: O(N)

Advantages

1. Very simple technique and also that does not require any preprocessing. Therefore total running time is the same as its matching time.

Disadvantages

1. Very inefficient method. Because this method takes only one position movement in each time

Boyer Moore Algorithm for Pattern Searching

The B-M algorithm takes a backward approach . the pattern string(p) is aligned with the start of the text string(T) and then compare the characters of pattern from right to left beginning with rightmost character

If a character is compared that is not within the pattern, no match can be found by comparing any furher characters at this position so the pattern can be shifted completely past the mismatching character.

For determining the possible shifts , B-M algorithm uses 2 preprocessing strategies simultaneously whenever a mismatch occurs, the algorithm computes a shift using both strategies and selects the longer one. thus it makes use of the most efficient stategy for each individual case

NOTE : Boyer Moore algorithm starts matching from the last character of the pattern.

The 2 strategies are called heuristics of B-M as they are used to reduce the search. They are

1) Bad Character Heuristic
2) Good Suffix Heuristic

Bad Character Heuristic

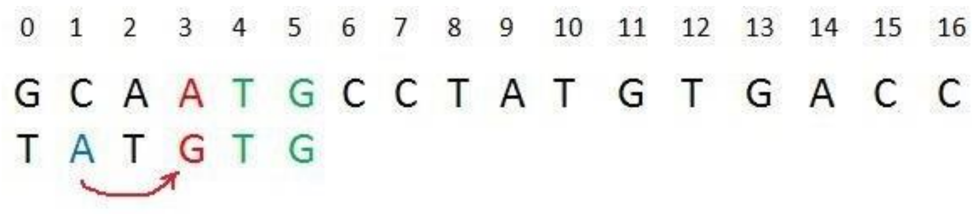
The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of the pattern is called the **Bad Character**. Upon mismatch, we shift the pattern until –

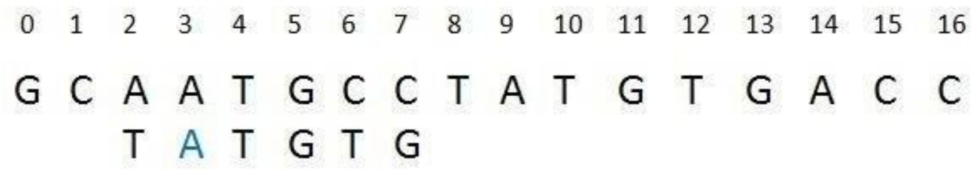
1) The mismatch becomes a match

2) Pattern P move past the mismatched character.

Case 1 – Mismatch become match

We will lookup the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then we'll shift the pattern such that it get aligned to the mismatching character in text T.



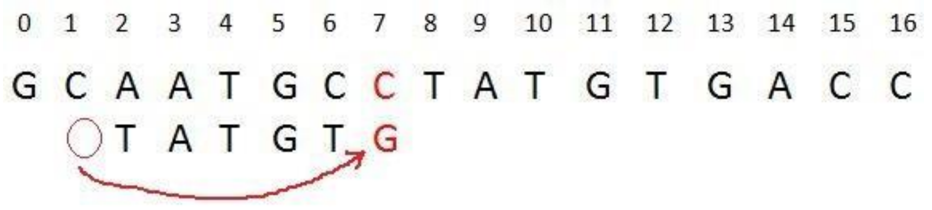


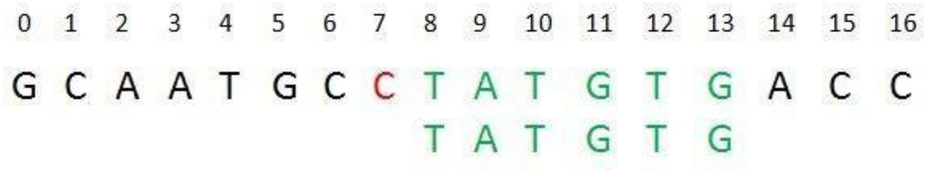
case 1

Explanation: In the above example, we got a mismatch at position 3. Here our mismatching character is "A". Now we will search for last occurrence of "A" in pattern. We got "A" at position 1 in pattern (displayed in Blue) and this is the last occurrence of it. Now we will shift pattern 2 times so that "A" in pattern get aligned with "A" in text.

Case 2 – Pattern move past the mismatch character

We'll lookup the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.





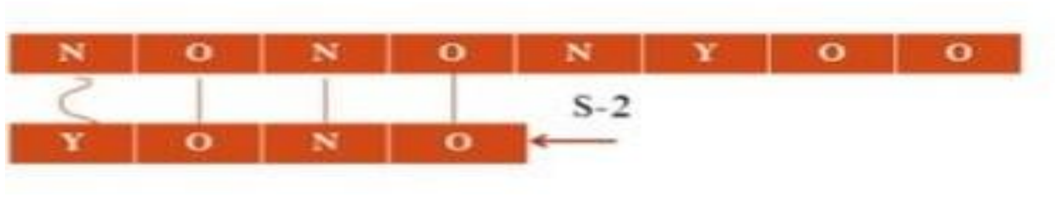
case2

Explanation: Here we have a mismatch at position 7. The mismatching character "C" does not exist in pattern before position 7 so we'll shift pattern past to the position 7 and eventually in above example we have got a perfect match of pattern (displayed in Green). We are doing this because, "C" do not exist in pattern so at every shift before position 7 we will get mismatch and our search will be fruitless.

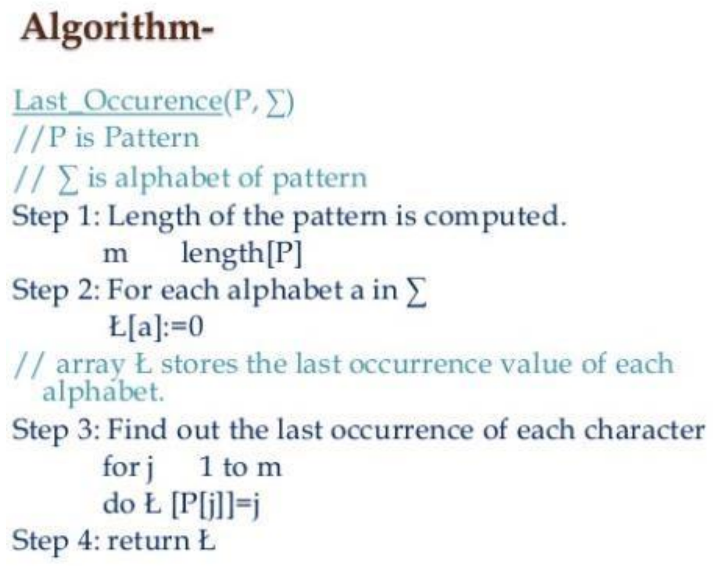
Problem in Bad Character Heuristic

In some cases Bad Character Heuristic produces negative results

For Example:



This means we need some extra information to produce a shift an encountering a bad character. The information is about last position of evry character in the pattern and also the set of every character in the pattern and also the set of characters used in the pattern



Algorithm-

```
Last_Occurence(P, ∑)
//P is Pattern
// ∑ is alphabet of pattern
Step 1: Length of the pattern is computed.
        m      length[P]
Step 2: For each alphabet a in ∑
        Ł[a]:=0
// array Ł stores the last occurrence value of each
   alphabet.
Step 3: Find out the last occurrence of each character
        for j     1 to m
        do Ł [P[j]]=j
Step 4: return Ł
```

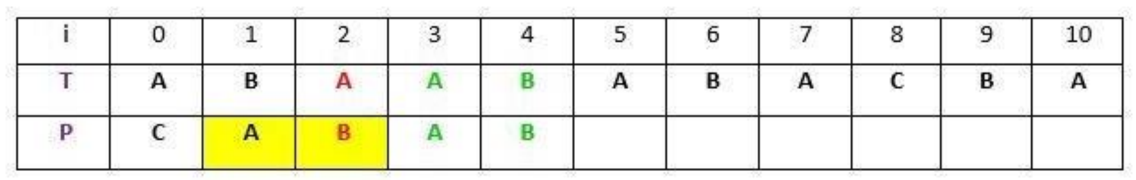
2.Good Suffix Heuristic

Let **t** be substring of text **T** which is matched with substring of pattern **P**. Now we shift pattern until :

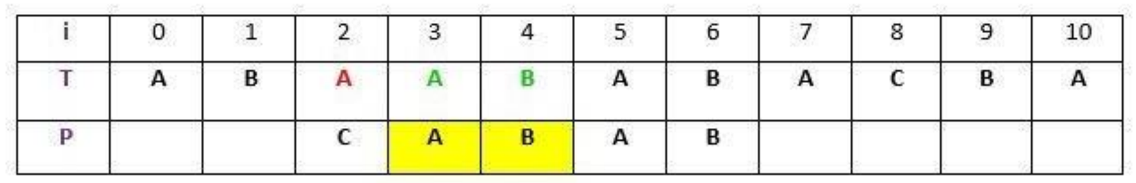
1) Another occurrence of t in P matched with t in T.

2) A prefix of P, which matches with suffix of t
3) P moves past t

Case 1: Another occurrence of t in P matched with t in T
Pattern P might contain few more occurrences of **t**. In such case, we will try to shift the pattern to align that occurrence with t in text T. For example-



i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

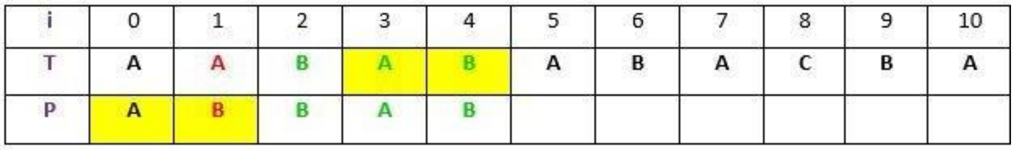


i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

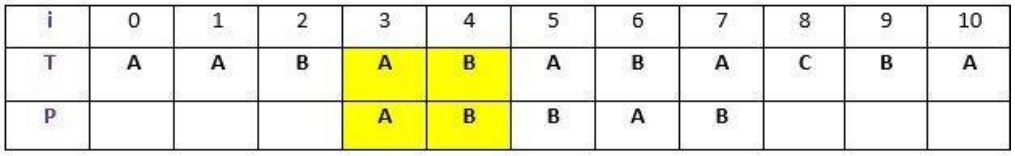
Figure – Case 1

Explanation: In the above example, we have got a substring t of text T matched with pattern P (in green) before mismatch at index 2. Now we will search for occurrence of t ("AB") in P. We have found an occurrence starting at position 1 (in yellow background) so we will right shift the pattern 2 times to align t in P with t in T. This is weak rule of original Boyer Moore

Case 2: A prefix of P, which matches with suffix of t in T
It is not always likely that we will find the occurrence of t in P. Sometimes there is no occurrence at all, in such cases sometimes we can search for some **suffix of t** matching with some **prefix of P** and try to align them by shifting P. For example –



i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P	A	B	B	A	B						



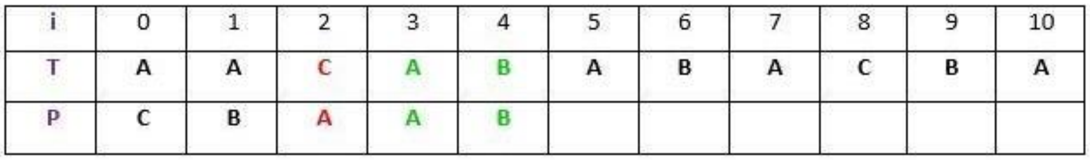
i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	B	A	B	A	B	A	C	B	A
P				A	B	B	A	B			

Figure – Case 2

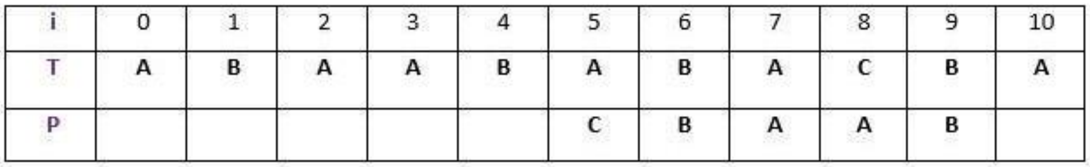
Explanation: In above example, we have got t ("BAB") matched with P (in green) at index 2-4 before mismatch . But because there exists no occurrence of t in P we will search for some prefix of P which matches with some suffix of t. We have found prefix "AB" (in the yellow background) starting at index 0 which matches not with whole t but the suffix of t "AB" starting at index 3. So now we will shift pattern 3 times to align prefix with the suffix.

Case 3: P moves past t

If the above two cases are not satisfied, we will shift the pattern past the t. For example –



i	0	1	2	3	4	5	6	7	8	9	10
T	A	A	C	A	B	A	B	A	C	B	A
P	C	B	A	A	B						



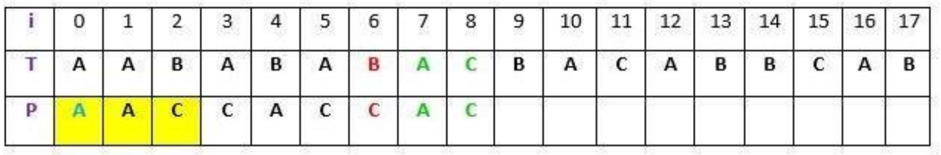
i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P						C	B	A	A	B	

Figure – Case 3

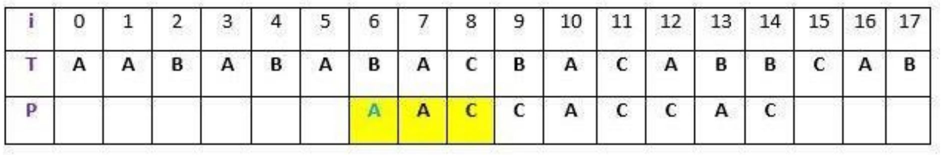
Explanation: If above example, there exist no occurrence of t ("AB") in P and also there is no prefix in P which matches with the suffix of t. So, in that case, we can never find any perfect match before index 4, so we will shift the P past the t ie. to index 5.

Strong Good suffix Heuristic

Suppose substring **q = P[i to n]** got matched with **t** in T and **c = P[i-1]** is the mismatching character. Now unlike case 1 we will search for t in P which is not preceded by character **c**. The closest such occurrence is then aligned with t in T by shifting pattern P. For example –



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P	A	A	C	C	A	C	C	A	C									



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	A	A	B	A	B	A	B	A	C	B	A	C	A	B	B	C	A	B
P							A	A	C	C	A	C	C	A	C			

Figure – strong suffix rule

Explanation: In above example, **q = P[7 to 8]** got matched with t in T. The mismatching character **c** is "C" at position P[6]. Now if we start searching t in P we will get the first occurrence of t starting at position 4. But this occurrence is preceded by "C" which is equal to c,so we will skip this and carry on searching. At position 1 we got another occurrence of t (in the yellow background). This occurrence is preceded by "A" (in blue) which is not equivalent to c.So we will shift pattern P 6 times to align this occurrence with t in T.We are doing this because we already know that character **c = "C"** causes the mismatch. So any occurrence of t preceded by c will again cause mismatch when aligned with t, so that's why it is better to skip this.

Preprocessing for Good suffix heuristic

As a part of preprocessing, an array **shift** is created. Each entry **shift[i]** contain the distance pattern will shift if mismatch occur at position **i-1**. That is, the suffix of pattern starting at position **i** is matched and a mismatch occur at position **i-1**. Preprocessing is done separately for strong good suffix and case 2 discussed above.

1) Preprocessing for Strong Good Suffix

Before discussing preprocessing, let us first discuss the idea of border. A **border** is a substring which is both proper suffix and proper prefix. For example, in string **“ccacc”**, **“c”** is a border, **“cc”** is a border because it appears in both end of string but **“cca”** is not a border.

As a part of preprocessing an array **bpos** (border position) is calculated. Each entry **bpos[i]** contains the starting index of border for suffix starting at index i in given pattern P.
The suffix **ϕ** beginning at position m has no border, so **bpos[m]** is set to **m+1** where **m** is the length of the pattern.
The shift position is obtained by the borders which cannot be extended to the left.

Complexity of Boyer Moore Algorithm

This algorithm takes o(mn) in the worst case and O(nlog(m)/m) on average case, which is the sub linear in the sense that not all characters are inspected

Applications

This algorithm is highly useful in tasks like recursively searching files for virus patterns,searching databases for keys or data ,text and word processing and any other task that requires handling large amount of data at very high speed

Knuth-Morris Pratt (KMP) Algorithm for Pattern Searching

The Naive pattern searching algorithm doesn’t work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

txt[] = "AAAAAAAAAAAAAAAAAB"

pat[] = "AAAAB"

txt[] = "ABABABCABABABCABABABC"

pat[] = "ABABAC" (not a worst case, but a bad case for Naive

KMP Algorithm is one of the most popular patterns matching algorithms. KMP stands for Knuth Morris Pratt. KMP algorithm was invented by Donald Knuth and Vaughan Pratt together and independently by James H Morris in the year 1970. In the year 1977, all the three jointlypublished KMP Algorithm.

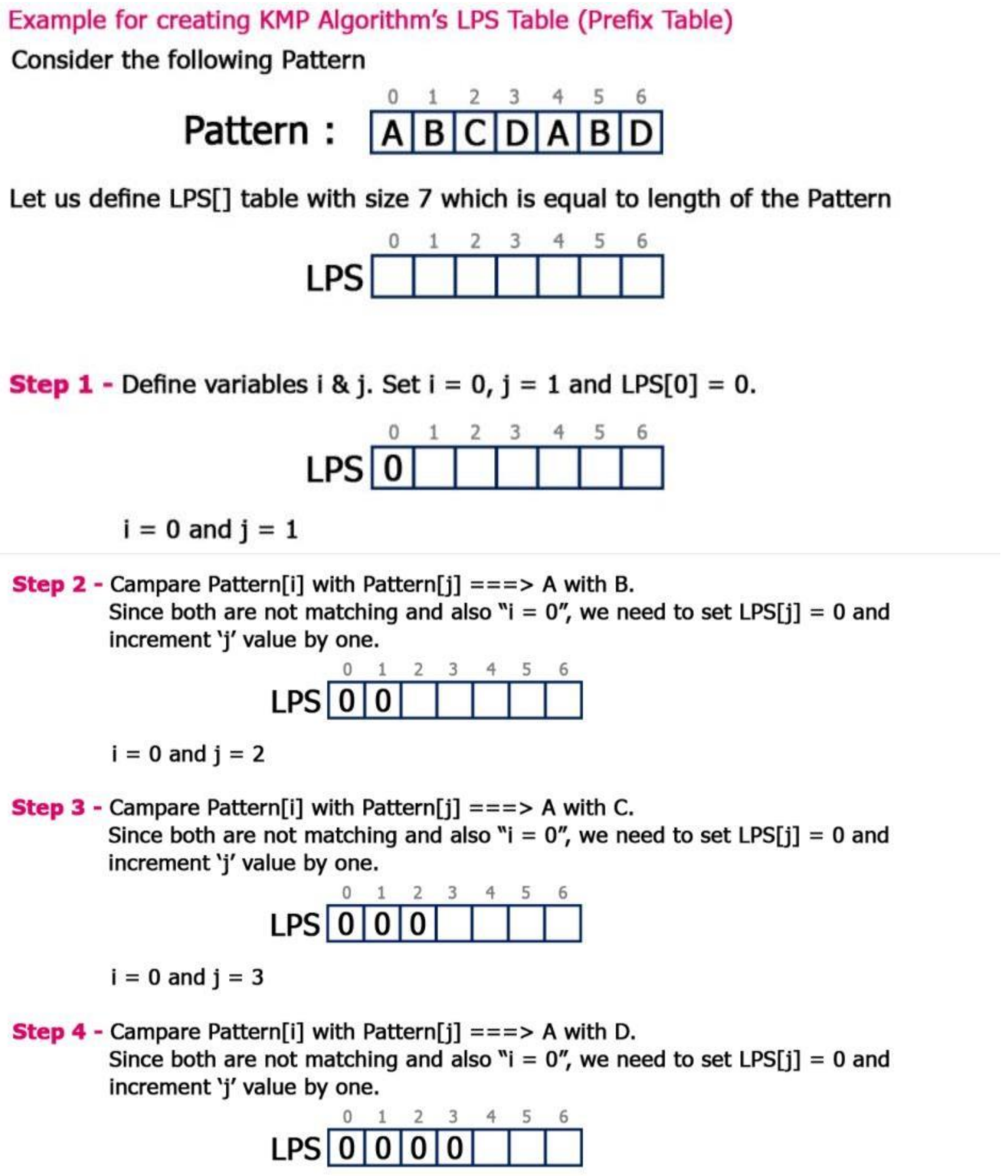
KMP algorithm was the first linear time complexity algorithm for string matching.
KMP algorithm is one of the string matching algorithms used to find a Pattern in a Text.

KMP algorithm is used to find a "Pattern" in a "Text". This algorithm campares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "**Prefix Table**" to skip characters comparison while matching. Some times prefix table is also known as **LPS Table**. Here LPS stands for "**Longest proper Prefix which is also Suffix".**

Steps for Creating LPS Table (Prefix Table)

- **Step 1** - Define a one dimensional array with the size equal to the length of the Pattern. (LPS[size])
- **Step 2** - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.
- **Step 3 -** Compare the characters at Pattern[i] and Pattern[j].
- **Step 4** - If both are matched then set LPS[j] = i+1 and increment both i & j values by one. Goto to Step 3.
- **Step 5** - If both are not matched then check the value of variable 'i'. If it is '0' then set LPS[j] = 0 and increment 'j' value by one, if it is not '0' then set i = LPS[i-1]. Goto Step 3.
- **Step 6**- Repeat above steps until all the values of LPS[] are filled.

Let us use above steps to create prefix table for a pattern...



Example for creating KMP Algorithm's LPS Table (Prefix Table)

Consider the following Pattern

Pattern :

0	1	2	3	4	5	6
A	B	C	D	A	B	D

Let us define LPS[] table with size 7 which is equal to length of the Pattern

LPS

0	1	2	3	4	5	6

Step 1 - Define variables i & j. Set i = 0, j = 1 and LPS[0] = 0.

LPS

0	1	2	3	4	5	6
0						

i = 0 and j = 1

Step 2 - Campare Pattern[i] with Pattern[j] ===> A with B.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

LPS

0	1	2	3	4	5	6
0	0					

i = 0 and j = 2

Step 3 - Campare Pattern[i] with Pattern[j] ===> A with C.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

LPS

0	1	2	3	4	5	6
0	0	0				

i = 0 and j = 3

Step 4 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

LPS

0	1	2	3	4	5	6
0	0	0	0			

i = 0 and j = 4

Step 5 - Campare Pattern[i] with Pattern[j] ===> A with A.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1		

i = 1 and j = 5

Step 6 - Campare Pattern[i] with Pattern[j] ===> B with B.
Since both are matching set LPS[j] = i+1 and increment both i & j value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 2 and j = 6

Step 7 - Campare Pattern[i] with Pattern[j] ===> C with D.
Since both are not matching and i !=0, we need to set i = LPS[i-1]
===> i = LPS[2-1] = LPS[1] = 0.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	

i = 0 and j = 6

Step 8 - Campare Pattern[i] with Pattern[j] ===> A with D.
Since both are not matching and also "i = 0", we need to set LPS[j] = 0 and increment 'j' value by one.

	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Here LPS[] is filled with all values so we stop the process. The final LPS[] table is as follows...

	0	1	2	3	4	5	6
LPS	**0**	**0**	**0**	**0**	**1**	**2**	**0**

How to use LPS Table

We use the LPS table to decide how many characters are to be skipped for comparison when a mismatch has occurred.
When a mismatch occurs, check the LPS value of the previous character of the mismatched

character in the pattern. If it is '0' then start comparing the first character of the pattern with the next character to the mismatched character in the text. If it is not '0' then start comparing the character which is at an index value equal to the LPS value of the previous character to the mismatched character in pattern with the mismatched character in the Text.

How the KMP Algorithm Works

Let us see a working example of KMP Algorithm to find a Pattern in a Text

EXAMPLE 1

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

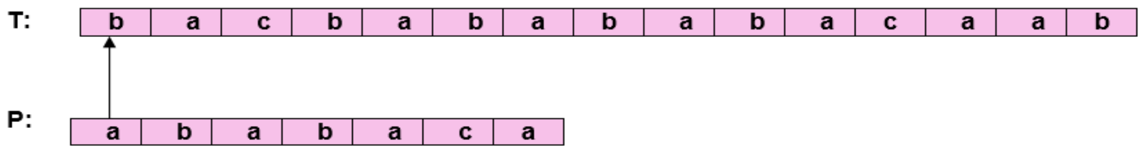
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

```
Initially: n = size of T = 15
m = size of P = 7
```

Step1: i=1, q=0

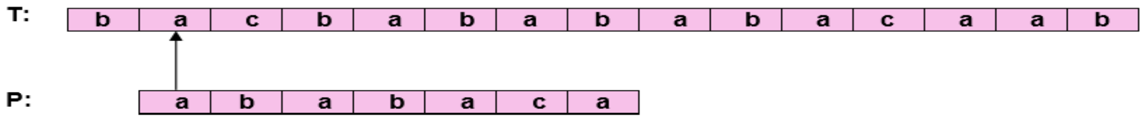
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

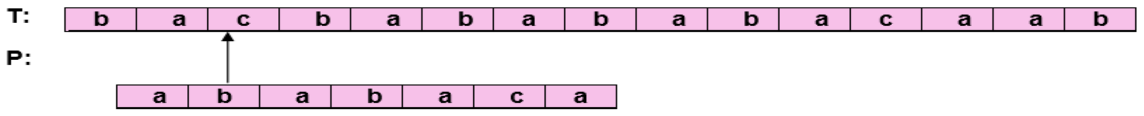
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

Comparing P [2] with T [3]   P [2] doesn't match with T [3]

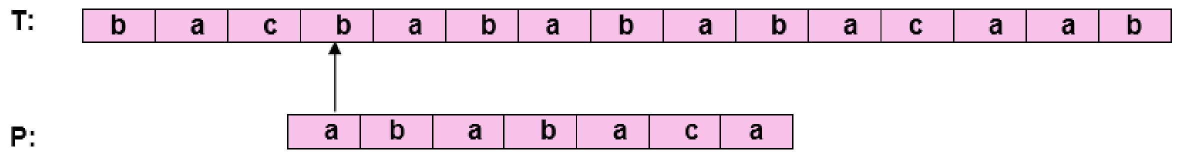


Backtracking on p, Comparing P [1] and T [3]

Activ
Go to S

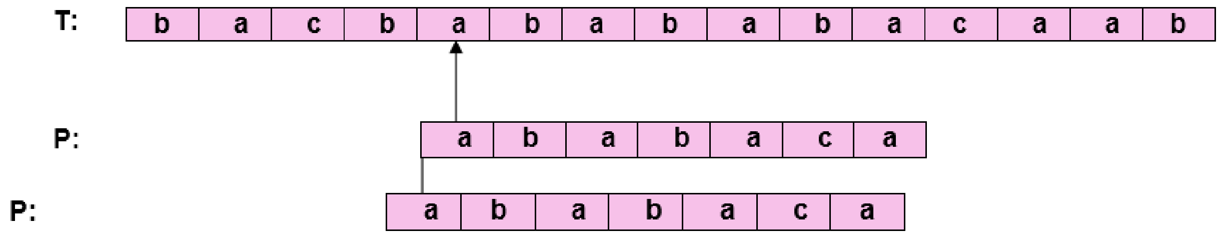
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



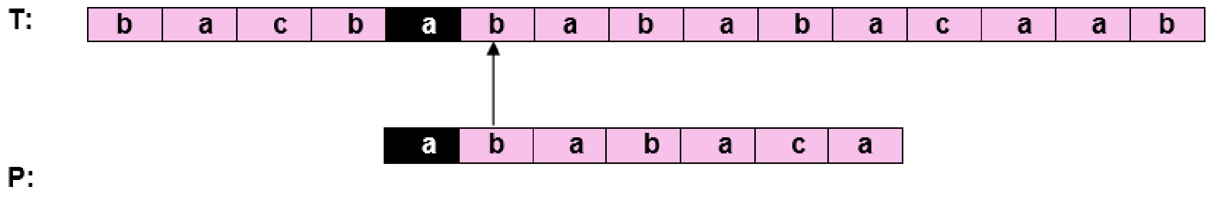
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



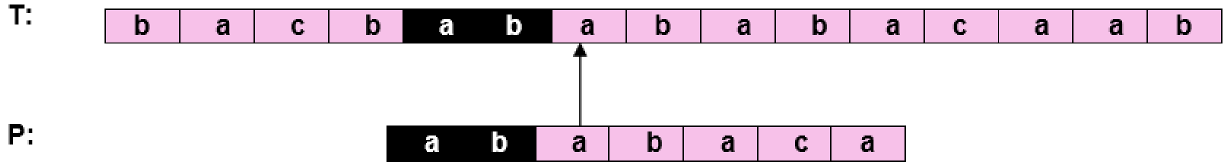
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



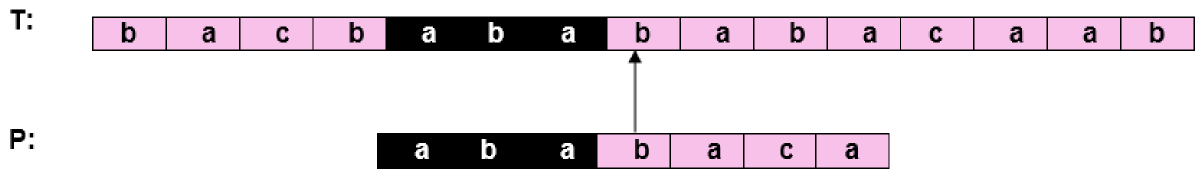
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



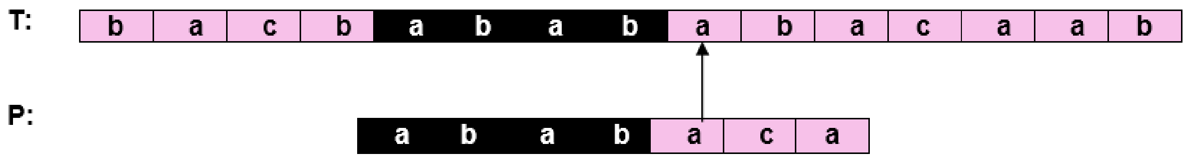
Step8: i = 8, q =3

Comparing P [4] with T [8]　　　　P [4] matches with T [8]



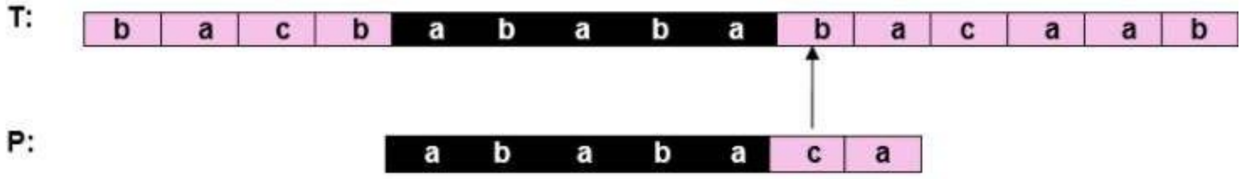
Step9: i = 9, q = 4

Comparing P [5] with T [9]　　　　P [5] matches with T [9]



Step10: i = 10, q = 5

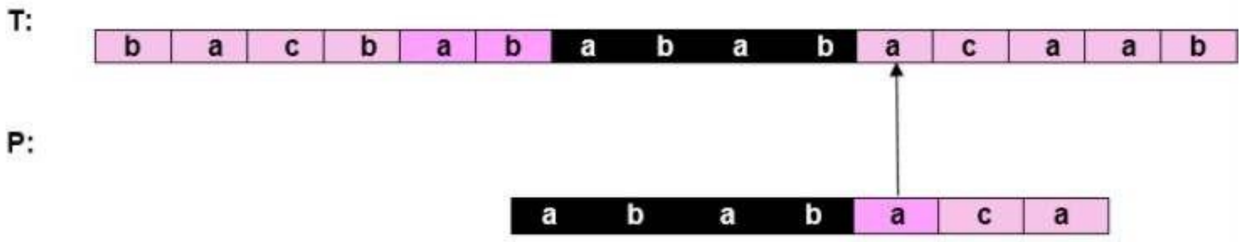
Comparing P [6] with T [10]　　　　P [6] doesn't match with T [10]



Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

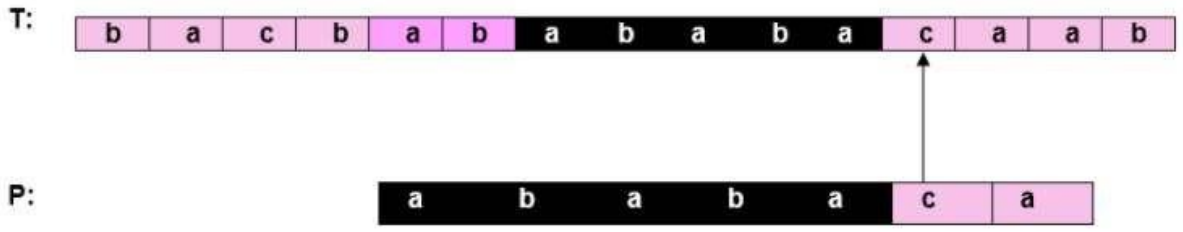
Step11: i = 11, q =4

Comparing P [5] with T [11]　　　　P [5] match with T [11]



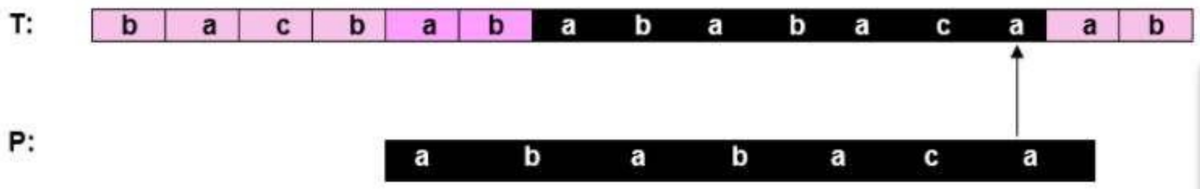
Step12: i = 12, q = 5

Comparing P [6] with T [12] P [6] matches with T [12]



Step13: i = 3, q = 6

Comparing P [7] with T [13] P [7] matches with T [13]



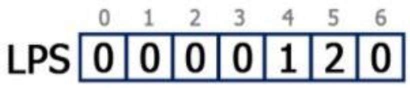
Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

Example 2

Consider the following Text and Pattern

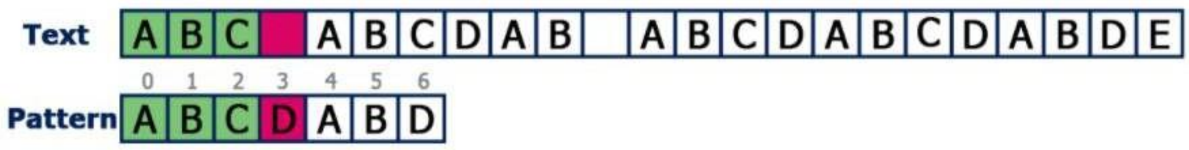
Text : ABC ABCDAB ABCDABCDABDE
Pattern : ABCDABD

LPS[] table for the above pattern is as follows...



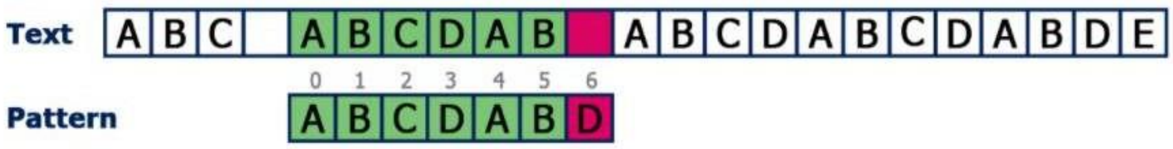
	0	1	2	3	4	5	6
LPS	0	0	0	0	1	2	0

Step 1 - Start comparing first character of Pattern with first character of Text from left to right



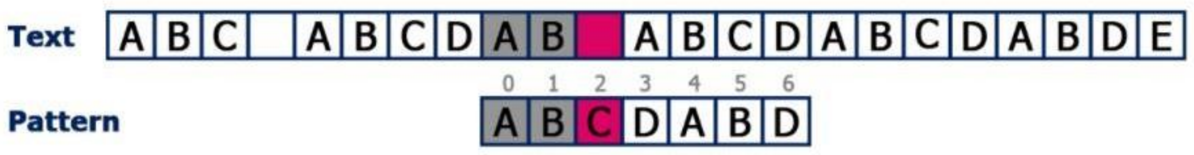
Here mismatch occured at Pattern[3], so we need to consider LPS[2] value. Since LPS[2] value is '0' we must compare first character in Pattern with next character in Text.

Step 2 - Start comparing first character of Pattern with next character of Text.



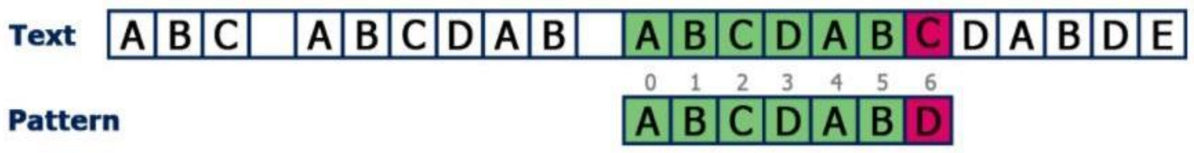
Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 3 - Since LPS value is '2' no need to compare Pattern[0] & Pattern[1] values



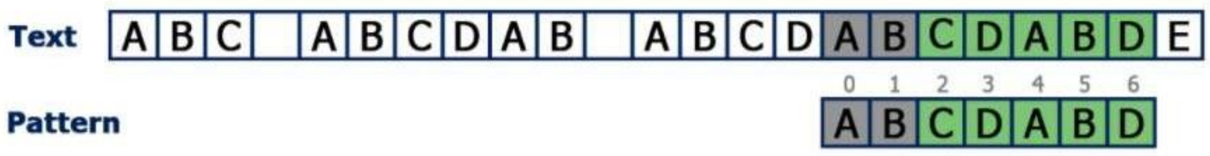
Here mismatch occured at Pattern[2], so we need to consider LPS[1] value. Since LPS[1] value is '0' we must compare first character in Pattern with next character in Text.

Step 4 - Compare Pattern[0] with next character in Text.



Here mismatch occured at Pattern[6], so we need to consider LPS[5] value. Since LPS[5] value is '2' we compare Pattern[2] character with mismatched character in Text.

Step 5 - Compare Pattern[2] with mismatched character in Text.



Here all the characters of Pattern matched with a substring in Text which is starting from index value 15. So we conclude that given Pattern found at index 15 in Text.

KMP ALGORITHM COMPLEXITY

O(m)- it is to compute to prefix function values
O(n)-it is to compare the pattern to the text
O(n+m)- Total time taken by KMP Algorithm.

Advantages

- The running time of KMP algorithm is O(n+m). which is very fast
- The algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files.

Disadvantages

- Does not work well as the size of the alphabet increase. By which more chances of mismatch occurs

TRIES DATA STRUCTURE

Trie is an efficient information re***Trie***val data structure. The term tries comes from the word retrieval

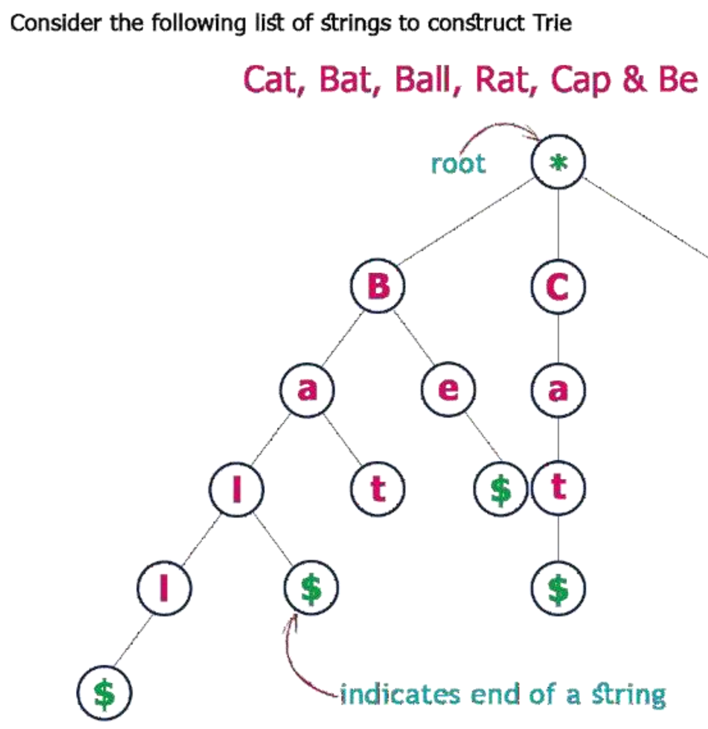
Definition of a Trie

- Data structure for representing a collection of strings
- In computer science , a trie also called digital tree or radix tree or prefix tree.
- Tries support fast string matching.

Properties of Tries

- A Multi way tree
- Each node has from 1 to n children
- Each edge of the tree is labeled with a character
- Each leaf node corresponds to the stored string which is a concatenation of characters on a path from the root to this node.

EXAMPLE


Consider the following list of strings to construct Trie
Cat, Bat, Ball, Rat, Cap & Be
root
*
B
C
R
a
e
a
a
l
t
$
t
t
l
$
$
$
$
indicates end of a string

Trie | (Insert and Search)

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length).
Given multiple strings. The task is to insert the string in a Trie

Examples:

Example 1: str = {"cat", "there", "caller", "their", "calling", “bat”}

```
   root
  /    \
 c      t
 |      |
 a      h
 | \    |
 l  t   e
 |      | \
 l      i  r
 | \    |  |
 e  i   r  e
 |  |
 r  n
    |
    g
```

Example 2: str = {"Candy", "cat", "Caller", "calling"}

```
   root
    |
    c
    |
    a
 /   | \
l    n  t
|    |
l    d
| \ |
e i y
| |
r  n
   |
   g
```

Approach: An efficient approach is to treat every character of the input key as an individual trie node and insert it into the trie. Note that the children are an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Trie deletion

Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Time Complexity: The time complexity of the deletion operation is O(n) where n is the key length

Advantages of Trie Data Structure

Tries is a tree that stores strings. The maximum number of children of a node is equal to the size of the alphabet. Trie supports search, insert and delete operations in O(L) time where **L** is the length of the key.

Hashing:- In hashing, we convert the key to a small value and the value is used to index data. Hashing supports search, insert and delete operations in O(L) time on average.

Self Balancing BST : The time complexity of the search, insert and delete operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(L * Log n) where n is total number words and L is the length of the word. The advantage of Self-balancing BSTs is that they maintain order which makes operations like minimum, maximum, closest (floor or ceiling) and kth largest faster.

Why Trie? :-

1. With Trie, we can insert and find strings in ***O(L)*** time where *L* represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)
2. Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.
3. We can efficiently do prefix search (or auto-complete) with Trie.

Issues with Trie :-

The main disadvantage of tries is that they need a lot of memory for storing the strings. For each node we have too many node pointers(equal to number of characters of the alphabet), if space is concerned, then **Ternary Search Tree** can be preferred for dictionary implementations. In Ternary Search Tree, the time complexity of search operation is O(h) where h is the height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing, and nearest neighbor search.
The final conclusion is regarding *tries data structure* is that they are faster but require *huge memory* for storing the strings.

APPLICATIONS OF TRIES

String handling and processing are one of the most important topics for programmers.
Many real time applications are based on the string processing like:

1. Search Engine results optimization

2. Data Analytics

3. Sentimental Analysis

The data structure that is very important for string handling is the **Trie** data structure that is based on **prefix of string**

TYPES OF TRIES

Tries are classified into three categories:

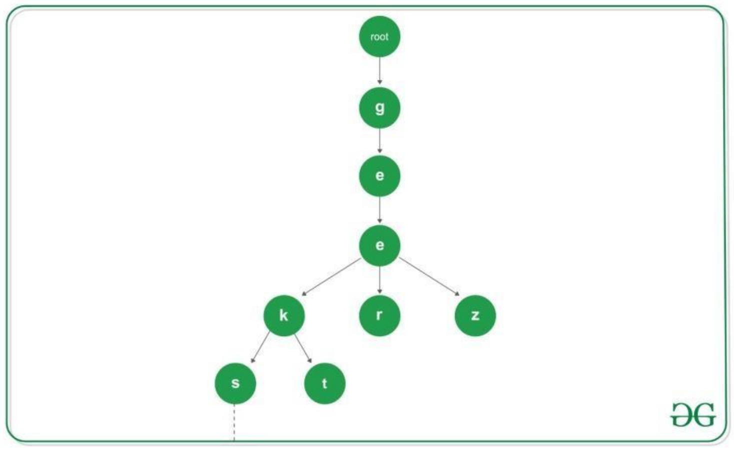
1. Standard Tries
2. Compressed Tries
3. Suffix Tries

STANDARD TRIES

A standard trie have the following properties:}

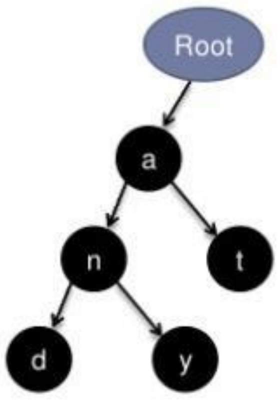
- It is an ordered tree like data structure.
- Each node(except the root node) in a standard trie is labeled with a character.
- The children of a node are in alphabetical order.
- Each node or branch represents a possible character of keys or words.
- Each node or branch may have multiple branches.
- The last node of every key or word is used to mark the end of word or node.
- The path from external node to the root yields the string of S.

Below is the illustration of the Standard Trie



Standard Trie Insertion

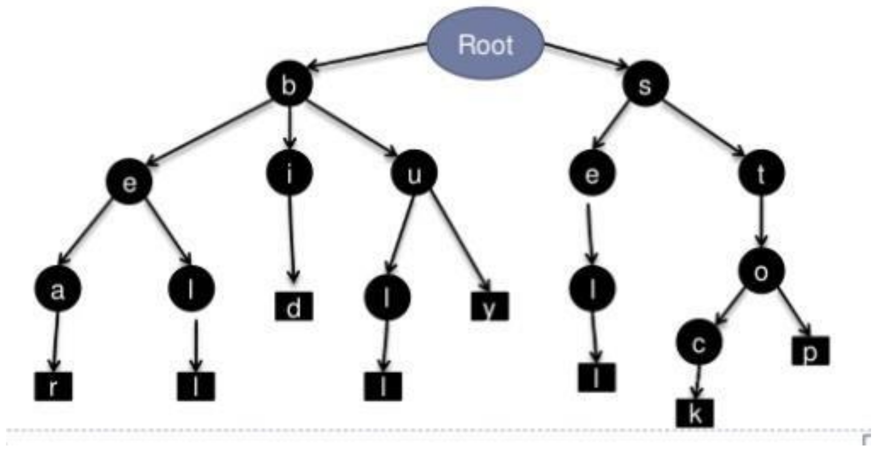
Strings={ a,an,and,any}



Example of Standard Trie

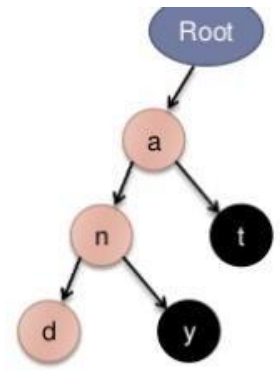
Standard trie for the following strings

S={ bear, bell, bid, bull, buy, sell, stock, stop}



Handling Keys(strings)

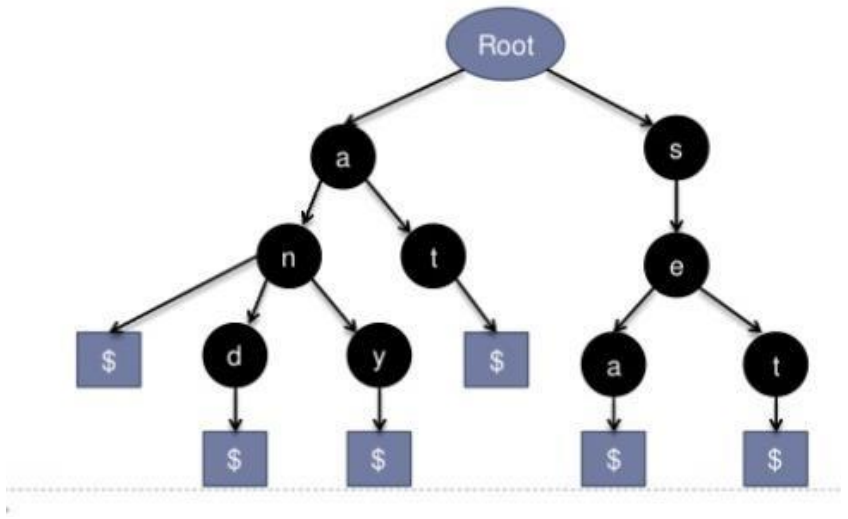
- When a key is prefix of another key
  How can we know that “an “ is a word
  Example : an, and



Standard Trie Searching

Search hit where search node has a $ symbol

▸ Search - sea

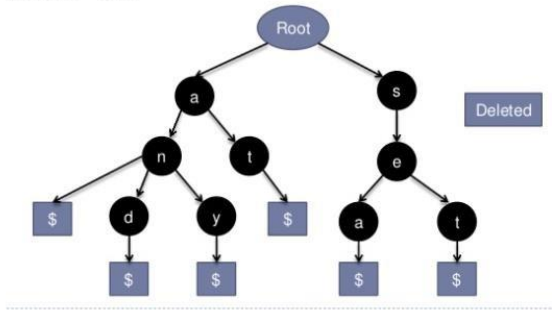


Standard Trie Deletion

To perform the deletion there exist cases

1. Word not found
   Return false
2. Word exist as a standalone word
   I. Part of any other node
      Example:

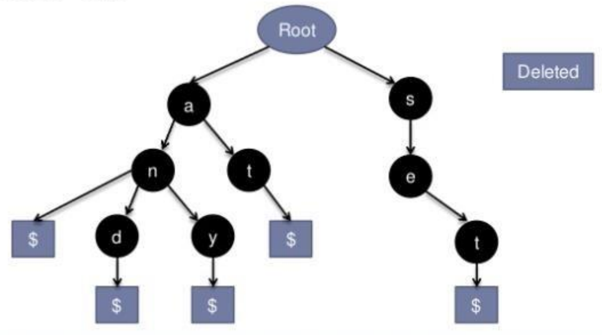
Delete - sea



II. Does not part of any other node

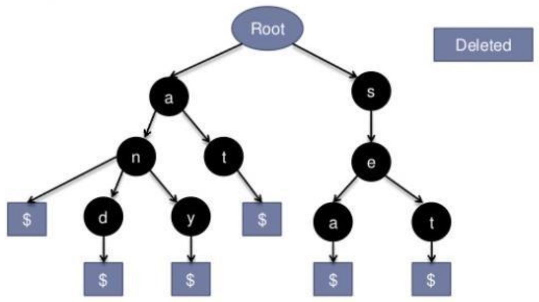
EXAMPLE

Delete - set



3. Word exist as a prefix of another word.

Delete - an



COMPRESSED TRIE

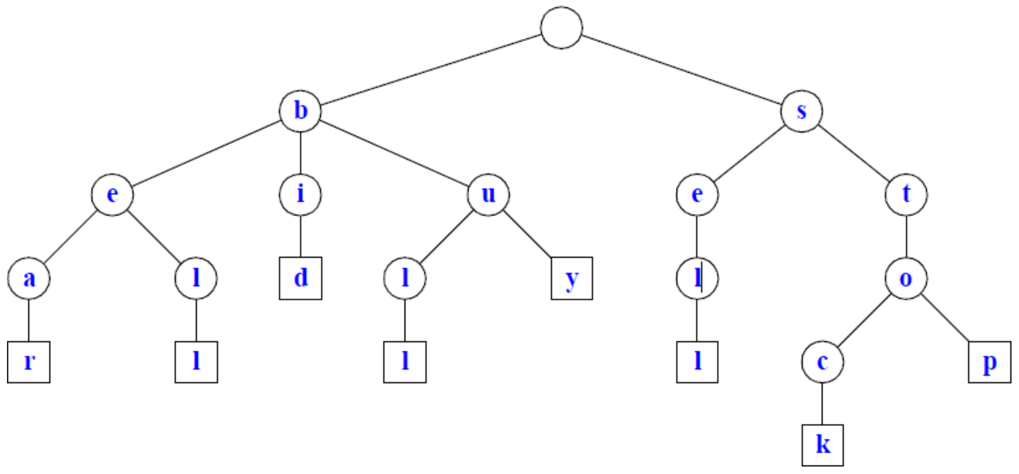
A Compressed trie have the following properties:

1. A Compressed Trie is an advanced version of the standard trie.
2. Each nodes(except the **leaf** nodes) have atleast 2 children.
3. It is used to achieve space optimization.
4. To derive a Compressed Trie from a Standard Trie, compression of chains of redundant nodes is performed.

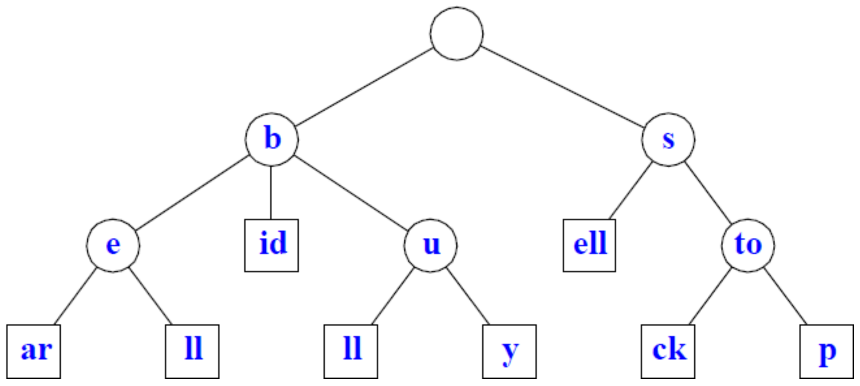
5. It consists of grouping, re-grouping and un-grouping of keys of characters.
6. While performing the insertion operation, it may be required to un-group the already grouped characters.
7. While performing the deletion operation, it may be required to re-group the already grouped characters.

Compressed trie is constructed from standard trie

- Standard Trie:



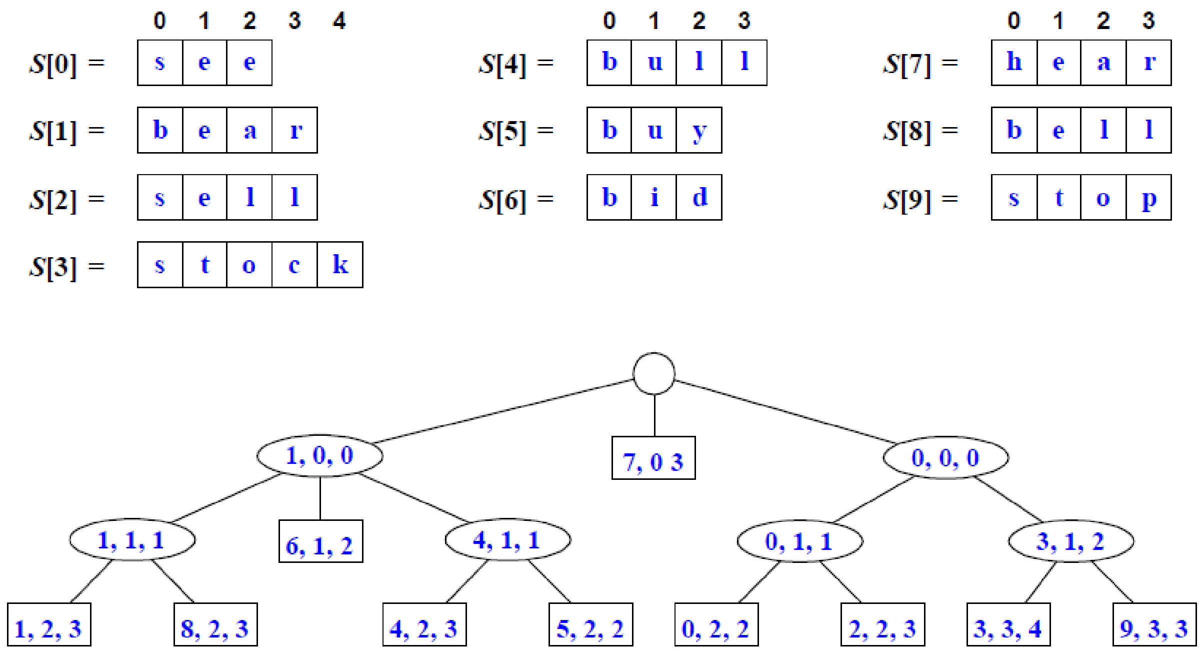
- Compressed Trie:

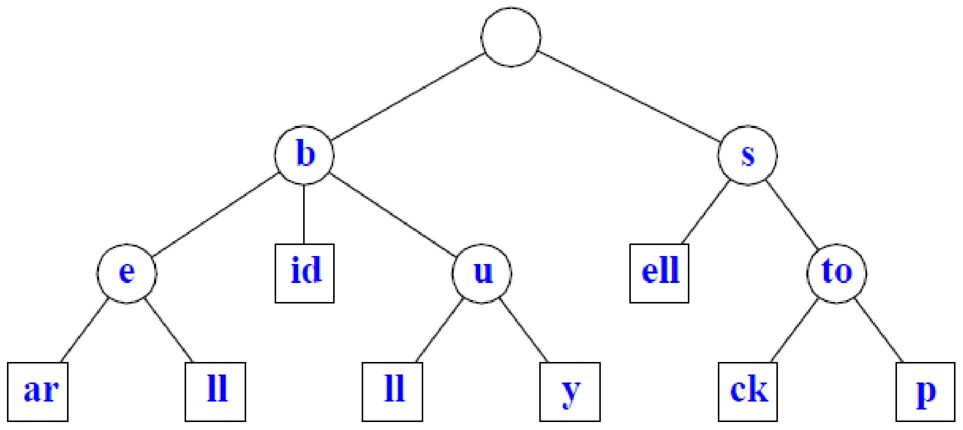


Storage of Compressed Trie

A compressed Trie can be stored at O9s) where s= | S| by using O(1) Space index ranges at the nodes

In the below representation each node is represented with (I,j,k) value
I-----indicate index of the string
j—starting index of the character of string I
k -- ending index of the character of the string I
Ex: In the given diagram node (4,2,3) having the characters**(ll)** which belongs to s[4] so i=4, index of l character in s[4] is 2 so j=2 and ending index is 3 so k=3




b
s
e
id
u
ell
to
ar
ll
ll
y
ck
p

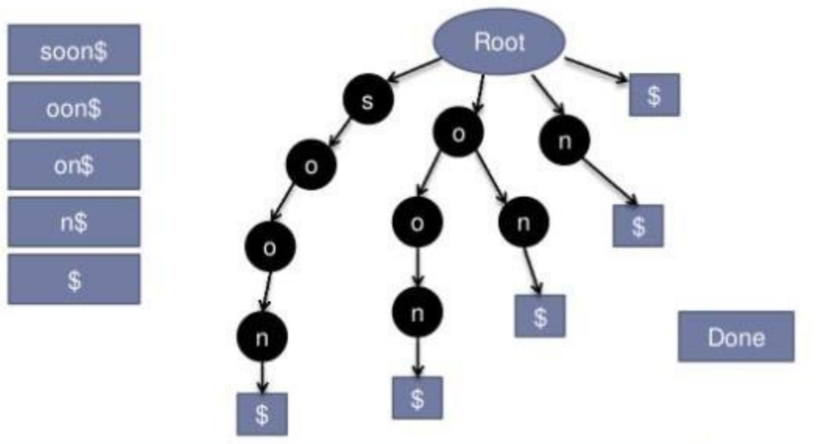
SUFFIX TRIES

A Suffix trie have the following properties:

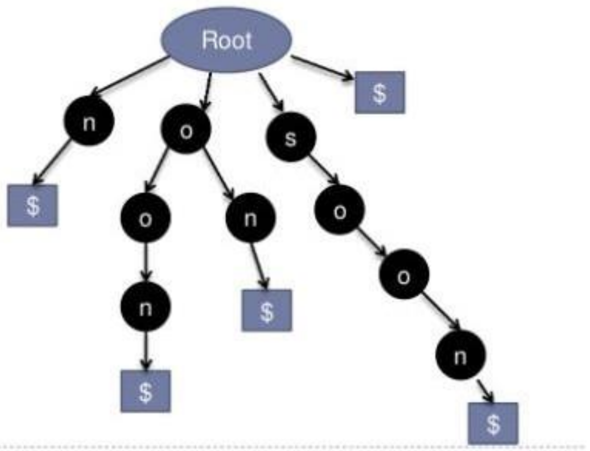
1. Suffix trie is a compressed trie for all the suffixes of the text
2. Suffix trie are space efficient data structure to store a string that allows many kinds of queries to be answered quickly.

Example

Let us consider an example text “soon$”



After alphabetically order the trie look like



Advantages of suffix tries

1. Insertion is faster compared to the hash table
2. Look up is faster than hash table implementation
3. There are no collision of different keys in tries