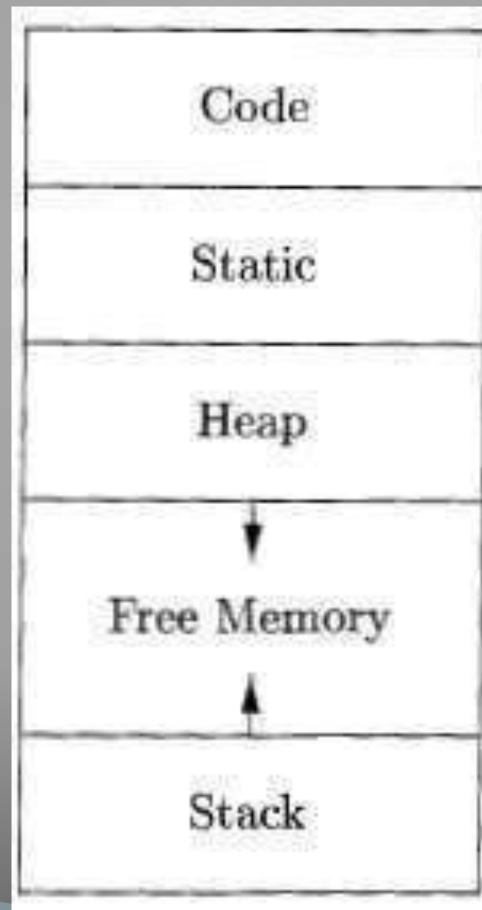
UNIT-IV

Run-Time Environments

Outline

- Compiler must do the storage allocation and provide access to variables and data
- Memory management
  - Stack allocation
  - Heap management
  - Garbage collection

Storage Organization



Static vs. Dynamic Allocation

NRCM
your roots to success...

- Static: Compile time, Dynamic: Runtime allocation
- Many compilers use some combination of following
  - Stack storage: for local variables, parameters and so on
  - Heap storage: Data that may outlive the call to the procedure that created it
- Stack allocation is a valid allocation for procedures since procedure calls are nested

Sketch of a quicksort program

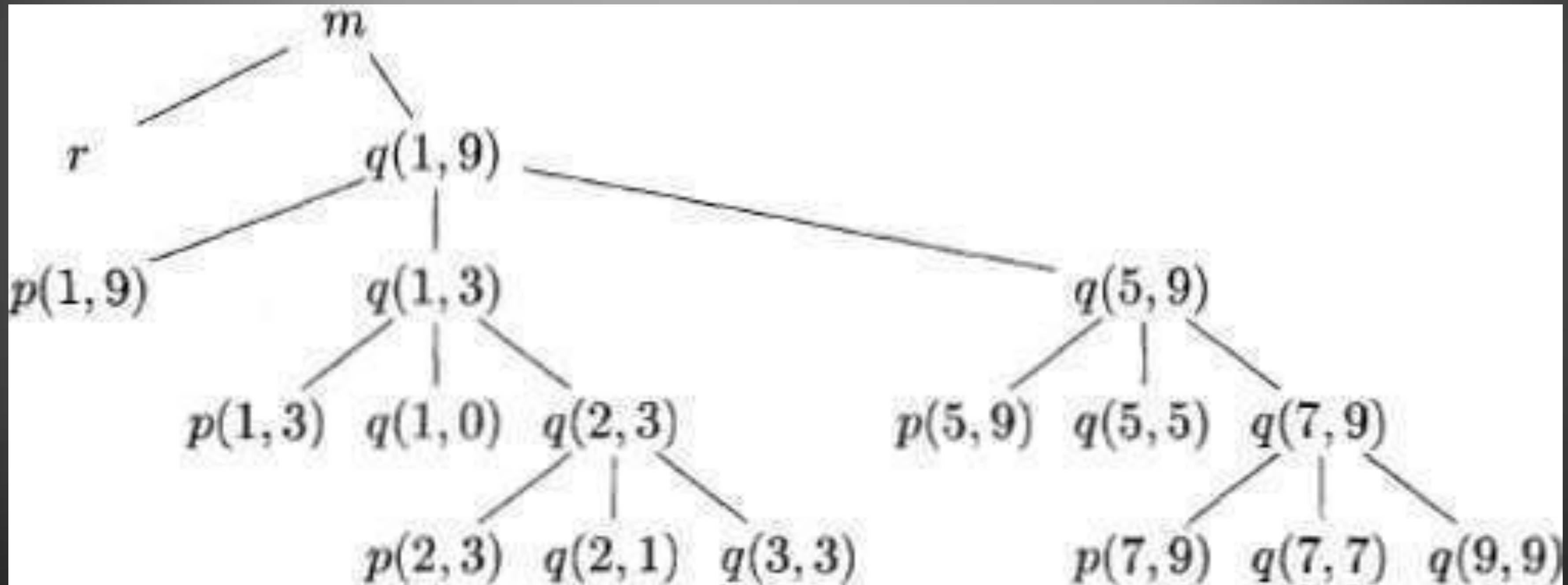
```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Activation for Quicksort

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

NRCM
your roots to success...

Activation tree representing calls during an execution of quicksort



your roots to success...

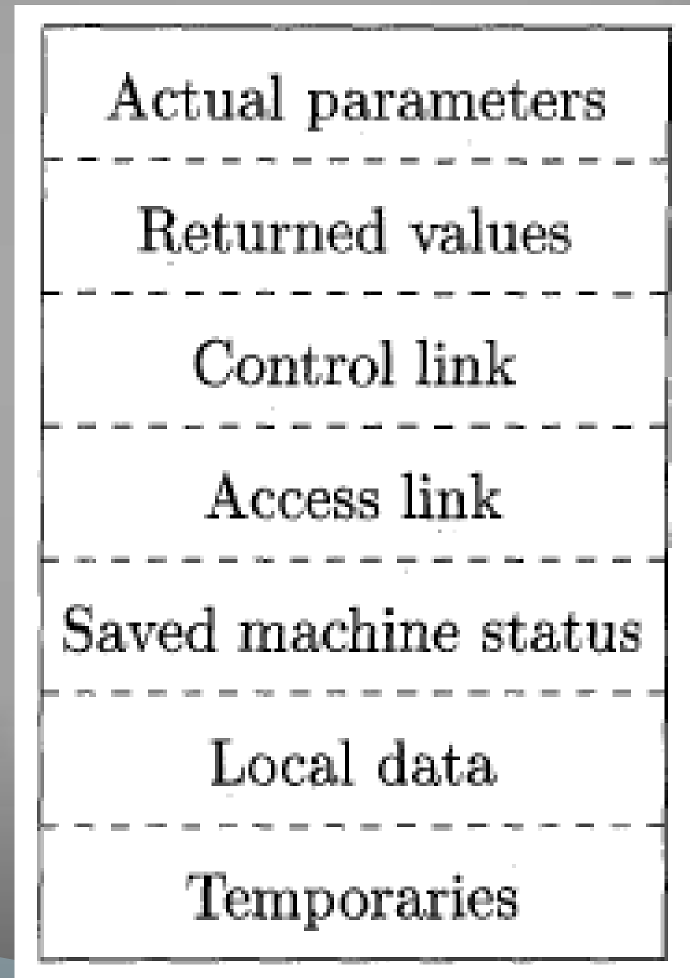
Activation records

NRCM
your roots to success...

- Procedure calls and returns are usaully managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last activation has record in the top of the stack.

A General Activation Record

your roots to success...



Activation Record

- Temporary values
- Local data
- A saved machine status
- An "access link"
- A control link
- Space for the return value of the called function
- The actual parameters used by the calling procedure

Access to dynamically allocated arrays

Control link and saved status

. . .

Pointer to *a*

Pointer to *b*

Pointer to *c*

. . .

Array *a*

Array *b*

Array *c*

Control link and saved status

top_sp

top

Activation record for *p*

Arrays of *p*

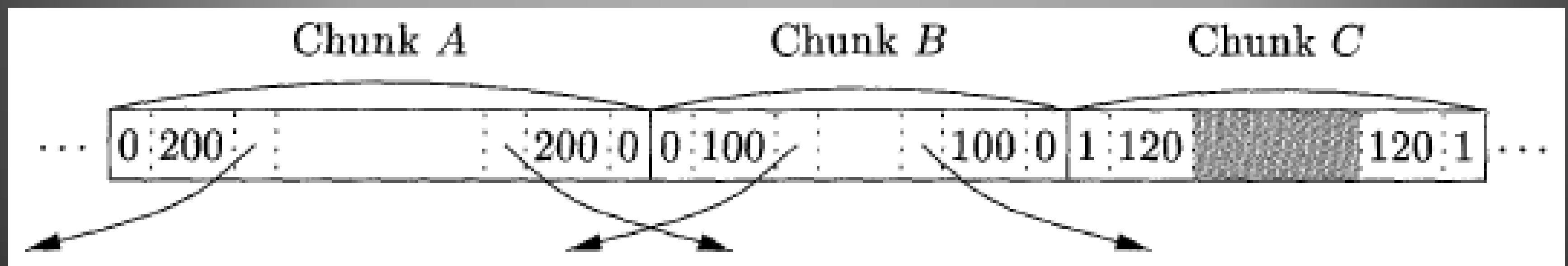
Activation record for procedure *q* called by *p*

Arrays of *q*

Memory Manager

- Two basic functions:
  - Allocation
  - Deallocation
- Properties of memory managers:
  - Space efficiency
  - Program efficiency
  - Low overhead

Part of a Heap



Introduction

- The final phase of a compiler is code generator
- It receives an intermediate representation (IR) with supplementary information in symbol table
- Produces a semantically equivalent target program
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Insrtuction ordering

Issues in the Design of Code Generator

- The most important criterion is that it produces correct code
- Input to the code generator
  - IR + Symbol table
  - We assume front end produces low-level IR, i.e. values of names in it can be directly manipulated by the machine instructions.
  - Syntactic and semantic errors have been already detected
- The target program
  - Common target architectures are: RISC, CISC and Stack based machines
  - In this chapter we use a very simple RISC-like computer with addition of some CISC-like addressing modes

complexity of mapping

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

x=y+z

```
LD      R0, y
ADD     R0, R0, z
ST      x, R0
```

a=b+c
d=a+e

```
LD      R0, b
ADD     R0, R0, c
ST      a, R0
LD      R0, a
ADD     R0, R0, e
ST      d, R0
```

Register allocation

- Two subproblems
  - Register allocation: selecting the set of variables that will reside in registers at each point in the program
  - Resister assignment: selecting specific register that a variable reside in
- Complications imposed by the hardware architecture
  - Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d

L     R1, a
A     R1, b
M     R0, c
D     R0, d
ST    R1,
```

```
t=a+b
t=t+c
T=t/d

L     R0, a
A     R0, b
M     R0, c
SRDA  R0, 32
D     R0, d
ST    R1, t
```

A simple target machine model

- Load operations: LD r,x and LD r1, r2
- Store operations: ST x,r
- Computation operations: OP dst, src1, src2
- Unconditional jumps: BR L
- Conditional jumps: Bcond r, L like BLTZ r, L

NRCM your roots to success...

Addressing Modes

- variable name: x
- indexed address: a(r) like LD R1, a(R2) means

  R1=contents(a+contents(R2))
- integer indexed by a register : like LD R1, 100(R2)
- Indirect addressing mode: *r and *100(r)
- immediate constant addressing mode: like LD R1, #100

b = a [i]

LD R1, i **//R1 = i**

MUL R1, R1, 8 **//R1 = Rl * 8**

LD R2, a(R1)

//R2=contents(a+contents(R1))

ST b, R2 **//b = R2**

a[j] = c

```
LD R1, c                  //R1 = c
LD R2, j                  // R2 = j
MUL R2, R2, 8             //R2 = R2 * 8
ST  a(R2), R1
        //contents(a+contents(R2))=R1
```