DATA STRUCTURES WITH C

(Common to CSE & ISE)

Subject Code: 10CS35	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART – A

UNIT - 1 **8 Hours**

BASIC CONCEPTS: Pointers and Dynamic Memory Allocation, Algorithm Specification, Data Abstraction, Performance Analysis, Performance Measurement

UNIT - 2 **6 Hours**

ARRAYS and STRUCTURES: Arrays, Dynamically Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, Representation of Multidimensional Arrays

UNIT - 3 **6 Hours**

STACKS AND QUEUES: Stacks, Stacks Using Dynamic Arrays, Queues, Circular Queues Using Dynamic Arrays, Evaluation of Expressions, Multiple Stacks and Queues.

UNIT - 4 **6 Hours**

LINKED LISTS: Singly Linked lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials, Additional List operations, Sparse Matrices, Doubly Linked Lists

PART - B

UNIT - 5 **6 Hours**

TREES – 1: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees, Heaps.

UNIT - 6 **6 Hours**

TREES – 2, GRAPHS: Binary Search Trees, Selection Trees, Forests, Representation of Disjoint Sets, Counting Binary Trees, The Graph Abstract Data Type.

UNIT - 7 **6 Hours**

PRIORITY QUEUES Single- and Double-Ended Priority Queues, Leftist Trees, Binomial Heaps, Fibonacci Heaps, Pairing Heaps.

UNIT - 8 **8 Hours**

EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees, AVL Trees, Red-Black Trees, Splay Trees.

Text Book:

1. Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, University Press, 2007.
   (Chapters 1, 2.1 to 2.6, 3, 4, 5.1 to 5.3, 5.5 to 5.11, 6.1, 9.1 to 9.5, 10)

TABLE OF CONTENTS

UNIT 1: BASIC CONCETPS

VARIOUS PHASES OF SYSTEM LIFE CYCLE

1) Requirements

• These describe
→ information given to the programmers (i.e. input) &
→ results the programmer must produce (i.e. output)

2) Analysis

• There are 2 methods to analysis:
i) Bottom-up methods are unstructured strategies that place an early emphasis on coding fine points.
ii) Top-up methods
→ begin with the purpose that the program will serve &
→ use this end-product to divide the program into manageable-segments

3) Design

• Designer approaches the system from perspectives of both
→ data objects that the program needs &
→ operations performed on them
• First perspective leads to creation of ADTs(Abstract Data Types)
while second requires specification of algorithms.

4) Refinement & Coding

• We choose representations for the data-objects &
then write algorithms for each operation on them.

5) Verification

• This phase consists of
→ developing correctness proofs for the program
→ testing program with a variety of input-data &
→ removing errors
• Testing requires
→ working-code & → sets of test-data
• Test-data should be developed carefully so that it includes all possible scenarios.

POINTERS

• This is a memory-location which holds the address of another memory-location.
• The 2 most important operators used w.r.t pointer are:
→ & (address operator)
→ * (dereferencing/indirection operator)

```
#include<stdio.h•
void main()
{
        int a=10,b=20;                  //Declare a data variable
        int *p,*q;                      //Declare a pointer variable
        int p=&a, q=&b;                 //Initialize a pointer variable
        int x=*p + *q;
        printf("%d+%d=%d",*p,*q, x);    //Access data using pointer variable
}
```

Program 1.1: Add 2 numbers using pointers

NULL POINTER

• The null pointer points to no object or function.
i.e. it does not point to any part of the memory.

```
if(p==NULL)
    printf("p does not point to any memory");
else
    printf("access the value of p");
```

DYNAMIC MEMORY ALLOCATION

- This is process of allocating memory-space during execution-time (or run-time).
- This is used if there is an unpredictable storage requirement.
- Memory-allocation is done on a heap.
- Memory management functions include:
  - → malloc (memory allocate)
  - → calloc (contiguous memory allocate)
  - → realloc (resize memory)
  - → free (deallocate memory)
- ***malloc*** function is used to allocate required amount of memory-space during run-time.
- If memory allocation succeeds, then address of first byte of allocated space is returned.
  If memory allocation fails, then NULL is returned.
- ***free()*** function is used to deallocate(or free) an area of memory previously allocated by malloc() or calloc().

```
#include<stdio.h>
void main()
{
        int i,*pi;
        pi=(int*)malloc(sizeof(int));
        *pi=1024;
        printf("an integer =%d",pi);
        free(pi);
}
```

Program 1.2: Allocation and deallocation of memory

- If we frequently allocate the memory space, then it is better to define a macro as shown below:

```
#define MALLOC(p,s)                          \
if(!((p)==malloc(s)))                        \
{                                            \
        printf("insufficient memory");       \
        exit(0);                             \
}
```

- Now memory can be initialized using following:

```
MALLOC(pi,sizeof(int));
MALLOC(pf,sizeof(float))
```

DANGLING REFERENCE

- Whenever all pointers to a dynamically allocated area of storage are lost, the storage is lost to the program. This is called a *dangling reference*.

POINTERS CAN BE DANGEROUS

1) Set all pointers to NULL when they are not actually pointing to an object. This makes sure that you will not attempt to access an area of memory that is either
   - → out of range of your program or
   - → that does not contain a pointer reference to a legitimate object

2) Use explicit type casts when converting between pointer types.

```
pi=malloc(sizeof(int));      //assign to pi a pointer to int
pf=(float*)pi;               //casts an 'int' pointer to a 'float' pointer
```

3) Pointers have same size as data type 'int'. Since int is the default type specifier, some programmers omit return type when defining a function. The return type defaults to 'int' which can later be interpreted as a pointer. Therefore, programmer has to define explicit return types for functions.

```
void swap(int *p,int *q)        //both parameters are pointers to ints
{
        int temp=*p;     //declares temp as an int and assigns to it the contents of what p points to
        *p=*q;           //stores what q points to into the location where p points
        *q=temp;         //places the contents temp in location pointed to by q
}
```

Program 1.3: Swap function

ALGORITHM SPECIFICATION

- An algorithm is a finite set of instructions that accomplishes a particular task.
- Algorithm must satisfy following criteria:
  1) Input: There are zero or more quantities that are externally supplied.
  2) Output: At least one quantity is produced.
  3) Definiteness: Each instruction is clear & unambiguous.
  4) Finiteness: If we trace out instructions of an algorithm, then for all cases, algorithm terminates after a finite number of steps.
  5) Effectiveness: Every instruction must be basic enough and feasible.
- Algorithm can be described in following ways:
  1) We can use natural language consisting of some mathematical equations.
  2) We can use graphic representations such as flowcharts.
  3) We can use combination of C and English language constructs.
- Algorithm 1.1: Selection sort algorithm.

```
for(i=0;i<n;i++)
{
        Examine list[i] to list[n-1] and suppose that the smallest integer is at list[min];
        Interchange list[i] and list[min];
}
```

- Algorithm 1.2: finding the smallest integer.

```
assume that minimum is list[i]
compare current minimum with list[i+1] to list[n-1] and find smaller number and make it the new minimum
```

- Algorithm 1.3: Binary search.

```
assumption :sorted n(≥1) distinct integers stored in the array list
return i if  list[i] = searchnum;
        -1 if no such index exists
denote left and right as left and right ends of the list to be searched (left=0 & right=n-1)
let middle=(left+right)/2   middle position in the list
compare list[middle] with searchnum and adjust left or right
compare list[middle] with searchnum
     1) searchnum < list[middle]
        set right to middle-1
     2) searchnum = list[middle]
        return middle
     3) searchnum > list[middle]
        set left to middle+1
if searchnum has not been found and there are more integers to check recalculate middle and continue search
```

- Algorithm 1.4: Permutations.

```
given a set of n(≥1) elements
print out all possible permutations
of this set
e.g. if set {a,b,c} is given,
        then set of permutations is {(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)}
```

```
int binsearch(int list[], int searchnum, int left, int right)
{   // search list[0]<= list[1]<=...<=list[n-1] for searchnum
        int middle;
        while (left<= right)
        {
                middle= (left+ right)/2;
                switch(compare(list[middle], searchnum))
                {
                  case -1: left= middle+ 1;
                           break;
                  case 0: return middle;
                  case 1: right= middle- 1;
                }
        }
 return -1;
}
```

```
int compare(int x, int y)
{
        if (x< y) return -1;
        else if (x== y) return 0;
             else return 1;
}
```

Program 1.4: Iterative Implementation of Binary Search

RECURSIVE ALGORITHMS

- A function calls itself either directly or indirectly during execution.
- Recursive-algorithms when compared to iterative-algorithms are normally compact and easy to understand.
- Various types of recursion:

1) Direct recursion: where a recursive-function invokes itself.
   For Ex,

```
int fact(int n)                 //to find factorial of a number
{
        if(n==0)
                return 1;
        return n*fact(n-1);
}
```

2) Indirect recursion: A function which contains a call to another function which in turn calls another function and so on and eventually calls the first function.
   For Ex,

```
void f1()          void f2()          void f3()
{                  {                  {
  ....               ....               ....
 f2();              f3();              f1();
}                  }                  }
```

```
int binsearch(int list[], int searchnum, int left, int right)
{       // search list[0]<= list[1]<=...<=list[n-1] for searchnum
        int middle;
        if (left<= right)
        {
                middle= (left+ right)/2;
                switch(compare(list[middle], searchnum))
                {
                  case -1:return binsearch(list, searchnum, middle+1, right);
                  case 0: return middle;
                  case 1: return binsearch(list, searchnum, left, middle- 1);
                }
         }
        return -1;
}

int compare(int x, int y)
{
         if (x< y) return -1;
         else if (x== y) return 0;
              else return 1;
}
```

Program 1.5: Recursive Implementation of Binary Search

```
void perm(char *list,int i,int n)
{
        int j,temp;
        if(i==n)
        {
                for(j=0;j<=n;j++)
                printf("%c", list[j]);
                printf(" ");
        }
        else
        {
                for(j=i;j<=n;j++)
                {
                   SWAP(list[i],list[j],temp);
                   perm(list,i+1,n);
                   SWAP(list[i],list[j],temp);
                }
        }
}
```

Program 1.6: Recursive permutations generator

```
void Hanoi(int n, char x, char y, char z)
{
        if (n > 1)
         {
              Hanoi(n-1,x,z,y);
              printf("Move disk %d from %c to %c.\n",n,x,z);
              Hanoi(n-1,y,x,z);
        }
        else
        {
              printf("Move disk %d from %c to %c.\n",n,x,z);
        }
 }
```

Program 1.7: Recursive Implementation of tower of Hanoi

DATA ABSTRACTION

• The process of separating logical properties of data from implementation details of data is called data abstraction.

Data Type

• A data type is a collection of objects and a set of operations that act on those objects.
• For e.g., data type 'int' consists of
→ objects {0,+1,-1,+2,-2. . . . }
→ operations such as arithmetic operators + - * /

ADT (ABSTRACT DATA TYPE)

• This is a data type that is organized in such a way that
→ specification of objects is separated from representation of objects
→ specification of operations on objects is separated from implementation of operations
• For example,
Specification: The specification of operations on objects consists of names of functions, type of arguments and return type. But, no information is given about how to implement in a programming language. So, specifications are implementation independent.
Implementation: The implementation of operations consists of a detailed algorithm using which we can code (i.e. functions) using any programming language(C or C++).
• ADTs can be implemented in C++ using a concept called class.
• ADT definition contains 2 main sections:
→ Objects & → Functions
• Functions of a data type can be classified into
1) Constructor: These functions create a new instance of the designated type.
For ex,

NaturalNumber Zero() ::= 0

2) Transformers: These functions create an instance of the designated type, generally by using one or more other instances.
For ex,

```
NaturalNumber Successor(x) ::= if(x==INT_MAX)
                                   return INT_MAX
                               else
                                   return x+1
```

3) Reporters: These functions provide information about an instance of the type, but they do not change the instance.
For ex,

```
Boolean IsZero(x) ::= if(x is zero)
                          return TRUE
                      else
                          return FALSE
```

ADT *NaturalNumber* is
objects: An ordered subrange of the integers starting at zero and ending at the maximum integer (MAXINT) on the computer.
functions:
for all $x, y \in$ *NaturalNumber*; TRUE, FALSE $\in$ *Boolean*
and where +, −, <, ==, and = are the usual integer operations

Zero( ) : *NaturalNumber*	::=	0
IsZero(*x*) : *Boolean*	::=	**if** (x == 0) *IsZero* = **true** **else** *IsZero* = **false**
Add(*x*, *y*) : *NaturalNumber*	::=	**if** ($x + y$ <= MAXINT) *Add* = $x + y$ **else** *Add* = MAXINT
Equal(*x*, *y*) : *Boolean*	::=	**if** ($x == y$) *Equal* = TRUE **else** *Equal* = FALSE
Successor(*x*) : *NaturalNumber*	::=	**if** (x == MAXINT) *Successor* = x **else** *Successor* = $x + 1$
Subtract(*x*, *y*) : *NaturalNumber*	::=	**if** ($x < y$) *Subtract* = 0 **else** *Subtract* = $x - y$

end *NaturalNumber*

ADT 1.1: Abstract data type *NaturalNumber*

PERFORMANCE ANALYSIS

- The process of estimating time & space consumed by program is called *performance analysis.*
- Efficiency of a program depends on 2 factors:
  1) Space efficiency (primary & secondary memory) &
  2) Time efficiency (execution time of program)

SPACE COMPLEXITY

- Space complexity of a program is the amount of memory required to run the program completely.
- Total space requirement of any program is given by

  S(P)= fixed space requirement + variable space requirement

  $S(P)= c + S_p(I)$

  1) Fixed Space Requirements
  - This component refers to space requirements that do not depend on the number and size of the program's inputs and outputs.
  - Fixed requirements include
    → program space (space for storing machine language program)
    → data space (space for constants, variables, structures)

  2) Variable Space Requirements
  - This component consists of space needed by structured variables whose size depends on particular instance of problem being solved. This also includes additional space required when a function uses recursion.

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;          // Sabc(I) = 0
}
```

Program 1.8: Simple arithmetic function

```
int sum(int list[],int n)
{
        int temp=0;
        int i;
        for(i=0;i<n;i++)
        temp=temp+list[i];
        return temp;
}
```

Program 1.9: Iterative function for summing a list of numbers

- In above program, there is no variable space requirement. This has only fixed space requirement ie $S_{sum}(I) = 0$. However, if same program is expressed recursively, then it is as shown below.

```
float rsum(int list[], int n)
{
        if(n) return rsum(list,n-1)+list[n-1];
        return 0;
}
```

Program 1.10: Recursive function for summing a list of numbers

- Space needed for one recursive call for above program is given below

Type	*Name*	*Number of bytes*
parameter:float	list[]	2
parameter:integer	n	2
return address:		2
TOTAL	per recursive call	6

$S_{sum}(I)= S_{sum}(n)=6n$

TIME COMPLEXITY

• *Time complexity* of an algorithm is amount of computer-time required to run the program till the completion.
• Time complexity depends on
→ number of inputs → number of outputs → magnitudes of inputs & outputs
• Time complexity = compile time + run time.
• Compile time is similar to fixed space component.
• Time complexity of a program can be measured by counting number of operations that a program can perform.
• A *program step* is a syntactically or semantically meaningful program-segment whose execution-time is independent of instance characteristics.
• Time taken by one program-step may be same or different from another program-step.
Ex1: sum=0; *//this statement is a program step which takes less time*
Ex2: si=p*t*r/100; *//this statement is also a program step. But, it takes more time when compared to Ex1.*
• Number of program steps(or step-count) can be obtained using 2 methods:
1) Counting method 2) Tabular method

COUNTING METHOD

• Use a global variable 'count' with initial value of 0 and insert a statement that increment count by 1 for each executable statement.
• Consider a program for summing a list of numbers:

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
   int i;
   for (i = 0; i < n; i++) {
       count++;                          /*for the for loop */
       tempsum += list[i]; count++;  /* for assignment */
   }
   count++;       /* last execution of for */
   return tempsum;
   count++;       /* for return */
}
```

2n + 3 steps

Program 1.11: Program for summing a list of numbers with count statements

• Since we are interested in only the final count, we can eliminate the computations of sum in above program as shown below:

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
   int i;
   for (i = 0; i < n; i++)
      count += 2;
   count += 3;
   return 0;
}
```

2n + 3 steps

Program 1.12: Simplified version of program 1.11

• So, each invocation of sum executes a total of 2n+3 steps.
• The recursive function to add the elements of a given array can be written as shown below:

```
float rsum(float list[ ], int n)
{
       count++;      /*for if conditional */
       if (n) {
              count++;  /* for return and rsum invocation */
              return rsum(list, n-1) + list[n-1];
       }
       count++;
       return list[0];
}
```

2n+2

Program 1.13: Program 1.12 with count statements added

• So, each invocation of rsum executes a total of 2n+2 steps.

TABULAR METHOD

• The following procedure is used to obtain step-count:

1) Determine step count for each statement. This is called step/execution(s/e).

2) Find out number of times each statement is executed. This is called frequency.

The frequency of non-executable statement is zero.

3) Multiply s/e(obtained in 1) and frequency(obtained in 2) to get total steps for each statement.

4) Add the totals(obtained in 3) to get step count for entire function.

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Figure 1.2: Step count table

Statement	s/e	Frequency	Total steps
float rsum(float list[ ],int n)	0	0	0
{	0	0	0
if (n)	1	n+1	n+1
return rsum(list,,n-1)+list[n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
Total			2n+2

Figure 1.3: Step count table for recursive summing functions

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]000)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j= 0; j< cols; j++)	1	rows(cols+1)	rowscols+rows
c[i][j] = a[i][j] + b[i][j];	1	rowscols	rowscols
}	0	0	0
Total			2rowscols+2rows+1

Figure 1.4: Step count table for matrix addition

• The best case step count is the minimum number of steps that can be executed for the given parameters.

The worst-case step count is the maximum number of steps that can be executed for the given parameters.

The average step count is the average number of steps executed on instances with the given parameters.

ASYMPTOTIC NOTATION

• The asymptotic behavior of a function is the study of how the value of a function f(n) varies for large value of n, where n=input size.

• Various types of asymptotic notations are:

1) Big(oh) notation(worst case time complexity)
2) Omega notation (best case time complexity)
3) Theta notation (average case time complexity)

1) Big Oh Notation

• Big oh is a measure of the longest amount of time taken by algorithm to complete execution.

• This is used for finding worst case time efficiency.

• A function f(n)=O(g(n)) iff there exists positive constants c and n_0 such that

f(n)<=c.g(n) for all n, n>=n_0.

• Here, c.g(n) is the upper bound. The upper bound on f(n) indicates that function f(n) will not consume more than the specified time c.g(n) i.e. running time of function f(n) may be equal to c.g(n) but it will never be worse than the upper bound.

• For ex, 3n+2=O(n) as 3n+2<4n for all n>2.

2) Omega Notation

• Omega is a measure of the least amount of time taken by algorithm to complete execution.

• This is used for finding best case time efficiency.

• A function f(n)= Ω(g(n)) iff there exists positive constant c and n_0 such that

f(n)>=cg(n) for all n, n>=n_0

• Here, c.g(n) is the lower bound. The lower bound on f(n) will consume at least the specified time c.g(n) i.e. running time of function f(n) may be equal to c.g(n) but it will never be better than the lower bound.

• For ex, 3n+2= (n) as 3n+2• =3n for all n>=1.

3) Theta Notation

• This is a measure of the least as well as longest amount of time taken by the algorithm to complete.

• A function f(n)= Θ(g(n)) iff there exists positive constants c_1,c_2 and n_0 such that

c_1g(n)<=f(n)<=c_2g(n) for all n, n>=n_0.

• Theta notation is more precise than both the big oh and omega notations.

• This notation is used to denote both lower bound and upper bound on a function.

• For ex, 3n+2= (n) as 3n+2>=3n for all n>=2 and 3n+2<=4n for all n>=2,so c_1=3,c_2=4 and n_0=2.

MAGIC SQUARE

- This is an n*n matrix of the integers from 1 to n^2 such that the sum of each row & column and the two major diagonals is the same.
- Coxeter has given the following rule for generating a magic square when n is odd:

  "Put a one in the middle box of the top row. Go up and left assigning numbers in increasing order to empty boxes. If your move causes you to jump off the square, figure out where you would be if you landed on a box on the opposite side of the square. Continue with this box. If a box is occupied, go down instead of up and continue".

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.6: Magic square for n=5

```
void magic (int n)
// create a magic square of size n, n is odd
{
  const int MaxSize = 51;// maximum square size
  int square[MaxSize][MaxSize], k, l ;

  // check correctness of n
  if ((n > (MaxSize)) || (n < 1)) {cerr << "Error!..n out of range" << endl; return;}
  else if (!(n%2)) {cerr << "Error!..n is even" << endl; return;}

  //n is odd. Coxeter's rule can be used
  for (int i = 0; i < n ; i++)  // initialize square to 0
    for (int j = 0; j < n; j++)
      square[i][j] = 0;
  square[0][(n-1)/2] = 1;  // middle of first row

  // i and j are current position
  int key = 2; i = 0; int j = (n-1)/2;
  while (key <= n * n) {
  // move up and left
    if (i - 1 < 0) k = n - 1; else k = i - 1;
    if (j - 1 < 0) l = n - 1; else l = j - 1;
    if (square[k][l])  i = (i+1)%n; // square occupied, move down
    else { // square [k ][l ] is unoccupied
      i = k;
      j = l;
    }
    square[i][j] = key;
    key++;
  } // end of while

  // output the magic square
  cout << "magic square of size " << n << endl;
  for (i = 0; i < n ; i++) {
    for (j = 0; j < n; j++)
      cout << square[i][j] << " ";
    cout << endl;
  }
}
```

Program 1.14: Magic square program

PRACTICAL COMPLEXITIES

• The time complexities of a program are useful in determining better algorithm among the algorithms that perform the same task. i.e. by comparing the time complexities of two algorithms that perform the same task, we can determine which algorithm is better.

• For ex, let time complexities of two programs P and Q be (n) and (n^2) respectively. Since, order of P is less than order of Q, the program P is faster than Q.

• Time complexity of an algorithm is normally expressed as a function of 'n' as shown in following table.

log *n*	*n*	*n* log *n*	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 1.7: Function values

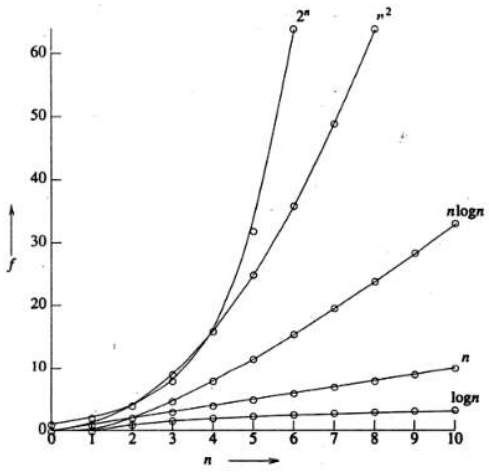


Figure 1.8: Plot of function values

PERFORMANCE MEASUREMENT

• The measure of how fast an algorithm is executed on a specific machine and how efficiently the algorithm uses space on that machine during execution is called performance measurement.

• Performance measurement wrt time can be obtained using 2 methods:

1) Using clock() function
2) Using time() function

	Method 1	Method 2
Start timing	start=clock();	start=time(NULL);
Stop timing	stop=clock();	stop=time(NULL);
Type returned	clock_t	time_t
Result in seconds	duration=((double)(stop-start)/CLK_TCK;	duration=(double)difftime(stop, start);

Figure 1.10: Event timing in C

Using clock() Function

• This function
→ returns amount of processor-time that has elapsed since the program running
→ is accessed through statement: #include<time.h>

• To find the time, we use clock() function twice in the program, once at:
→ start of event
→ end of event

• The difference between stop-time and start-time gives processor-time for the activity to be completed.

• To convert into seconds, we divide it by "ticks per second" which is identified using symbolic constant CLOCKS_PER_SEC.

Using time() Function

• This function
→ returns time measured in seconds
→ has one parameter which specifies a location to hold the time

• When we do not want to store the time, pass NULL as the parameter.

• To find the time, we use time() function twice in the program, once at:
→ start of the event
→ end of the event

• When start-time and stop-time are passed as parameters to the function time(), the function returns the difference between 2 times measured in seconds.

```
int seqsearch (int *a, const int n, const int x) // a [0], ... ,a [n]
 {
   int i = n; a [0] = x;
   while (a [i] != x)
     i--;
   return i;
 } // end of seqsearch
```

Program 1.15: Sequential search

```
void TimeSearch() {
   int a[1001], n[20];
   for (int j = 1; j <= 1000; j++) // initialize a
      a [j] = j;
   for (j = 0; j < 10; j++) {// values of n
      n [j] = 10 * j; n [j +10] = 100 * (j + 1);
   }
   cout << "  n   time" << endl;
   for(j = 0; j < 20; j++) { // obtain computing times
      long start, stop;
      time (start) ; // start timer
      int k = seqsearch(a, n [j ], 0); // unsuccessful search
      time (stop) ; // stop timer
      long runTime = stop - start ;
      cout << "  " << n [j ] << "  " << runTime << endl;
   }
   cout << "Times are in hundredths of a second." << endl ;
}
```

Program 1.16: program to time program 1.15

UNIT 2: ARRAYS AND STRUCTURES

ARRAY

• This is collections of elements of the same basic data type.

```
Structure Array is
objects: A set of pairs <index, value> where for each value of index there is a value from
         the set item. Index is a finite ordered set of one or more dimensions, for example,
         {0, … , n-1} for one dimension,
        {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions, etc.
  Functions:
  for all A ∈ Array, i ∈ index, x ∈ item, j, size ∈ integer
  Array Create(j, list)   ::= return an array of  j dimensions where list is a j-tuple whose ith
                              element is the size of the ith dimension. Items are undefined.
  Item Retrieve(A, i)     ::= if (i ∈ index)
                                  return the item associated with index value i in array A
                              else
                                  return error
  Array Store(A, i, x)    ::= if (i in index)
                                  return an array that is identical to array A except the new pair
                                      <i, x> has been  inserted
                              else
                                  return error
end array
```

ADT 2.1: Abstract data type Array

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main (void)
{
        for (i = 0; i < MAX_SIZE; i++)
        input[i] = i;
        answer = sum(input, MAX_SIZE);
        printf("The sum is: %f\n", answer);
}
float sum(float list[], int n)
{
        int i;
        float tempsum = 0;
        for (i = 0; i < n; i++)
                tempsum += list[i];
        return tempsum;
}
```

Program 2.1: Program to find sum of n numbers

ARRAYS IN C

- A one-dimensional array can be declared as follows:

```
int list[5];        //array of 5 integers
int *plist[5];      //array of 5 pointers to integers
```

- Compiler allocates 5 consecutive memory-locations for each of the variables 'list' and 'plist'.
- Address of first element list[0] is called base-address.
- Memory-address of list[i] can be computed by compiler as

  α+i*sizeof(int) where α=base address

```
void print1(int *ptr, int rows)
{
        /* print out a one-dimensional array using a pointer */
        int i;
        printf("Address Contents\n");
        for (i=0; i < rows; i++)
                printf("%8u%5d\n", ptr+i, *(ptr+i));
        printf("\n");
}

void main()
{
    int one[] = {0, 1, 2, 3, 4};
    print1(&one[0], 5)
}
```

Program 2.2:Program to print both address of ith element of given array & the value found at that address(Fig 2.1)

Address	Contents
1228	0
1230	1
1232	2
1234	3
1236	4

Figure 2.1: one dimensional array addressing

DYNAMICALLY ALLOCATED ARRAYS
ONE-DIMENSIONAL ARRAYS

- When writing programs, sometimes we cannot reliably determine how large an array must be.
- A good solution to this problem is to
  → defer this decision to run-time &
  → allocate the array when we have a good estimate of required array-size
- Dynamic memory allocation can be performed as follows:

```
int i,n,*list;
printf("enter the number of numbers to generate");
scanf("%d",&n);
if(n<1)
{
        printf("improper value");
        exit(0);
}
MALLOC(list, n*sizeof(int));
```

- The above code would allocate an array of exactly the required size and hence would not result in any wastage.

TWO DIMENSIONAL ARRAYS

- These are created by using the concept of array of arrays.
- A 2-dimensional array is represented as a 1-dimensional array in which each element has a pointer to a 1-dimensional array as shown below

```
int x[5][7];    //we create a 1-dimensional array x whose length is 5;
                //each element of x is a 1-dimensional array whose length is 7.
```

- Address of x[i][j] = x[i]+j*sizeof(int)

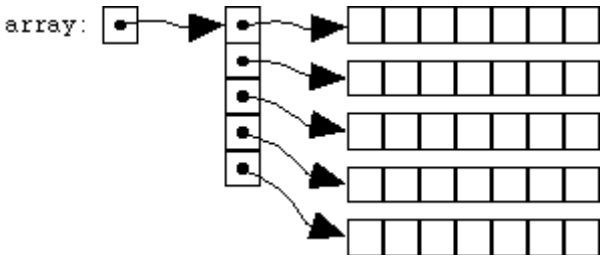


Figure 2.2: Array-of-arrays representation

```
#include <stdlib.h>
int **array;
array = malloc(nrows * sizeof(int *));
if(array == NULL)
{
        printf("out of memory\n");
        exit or return
}
for(i = 0; i < nrows; i++)
{
        array[i] = malloc(ncolumns * sizeof(int));
        if(array[i] == NULL)
        {
                printf("out of memory\n");
                exit or return
        }
}
```

Program 2.3: Dynamically create a two-dimensional array

CALLOC

- These functions
  → allocate user-specified amount of memory &
  → initialize the allocated memory to 0.
- On successful memory-allocation, it returns a pointer to the start of the new block.
  On failure, it returns the value NULL.
- Memory can be allocated using calloc as shown below:

```
int *p;
p=calloc(n, sizeof(int));          //where n=array size
```

- To create clean and readable programs, a CALLOC macro can be created as shown below:

```
#define CALLOC(p,n,s)              \
if((p=calloc(n,s))==NULL)          \
{                                  \
  printf("insufficient memory");   \
  exit(1);                           \
}                                  \
```

REALLOC

- These functions resize memory previously allocated by either malloc or calloc.
  For example,
  realloc(p,s); *//this changes the size of memory-block pointed at by p to s.*
- When s>oldSize, the additional s-oldSize have an unspecified value and
  when s<oldSize, the rightmost oldSize-s bytes of old block are freed.
- On successful resizing, it returns a pointer to the start of the new block.
  On failure, it returns the value NULL.
- To create clean and readable programs, the REALLOC macro can be created as shown below

```
#define REALLOC(p,s)                    \
if((p=realloc(p,s))==NULL)              \
{                                       \
        printf("insufficient memory");  \
        exit(0);                        \
}
```

STRUCTURES

• This is collection of elements whose data types are different.

```
typedef struct
{
        char name[10];
        int age;
        float salary;
}humanBeing;
```

• Dot operator(.) is used to access a particular member of the structure. For ex,

```
person.age=10;
person.salary=35000;
strcpy(person.name,"james");
```

• Variables can be declared as follows

```
humanBeing  person1,person2;
```

• Structures cannot be directly checked for equality or inequality. So, we can write a function to do this.

```
if(humansEqual(person1,person2))
        printf("two human beings are same");
else
        printf("two human beings are different");

------------------------------------------------------------------

int humansEqual(humanBeing person1,humanBeing person2)
{
        if(strcmp(person1.name,person2.name))
                return 0;
        if(person1.age!=person2.salary)
                return 0;
        if(person.salary!=person2.salary)
                return 0;
      return 1;
}
```

• We can embed a structure within a structure.

```
typedef struct
{
     int month;
     int day;
     int year;
}date;

typedef struct
{
    char name[10];
    int age;
    float salary;
    data dob;
}humanBeing;
```

SELF-REFERENTIAL STRUCTURES

• A self-referential structure is one in which one or more of its components is a pointer to itself.

• These require dynamic storage management routines (malloc & free) to explicitly obtain and release memory.

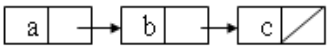
```
typedef struct
{
    char data;
    struct list *link;          //list is a pointer to a list structure
 }list;
```

• Consider three structures and values assigned to their respective fields:

```
list item1,item2,item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=item2.link=item3.link=NULL;
```

• We can attach these structures together as follows

```
item1.link=&item2;
tem2.link=&item3;
```



UNION

• This is similar to a structure, but the fields of a union must share their memory space. This means that only one field of the union is "active" at any given time. For ex,

```
typedef struct
{
    enum tagField{female,male} sex;
    typedef union
    {
        int children;
        int beard;
    }u;
}sexType;

typedef struct
{
    char name[10];
    int age;
    float salary;
    date dob;
    sexType sexInfo;
}humanBeing;

humanBeing person1,person2;
```

• We can assign values to person1 and person2 as:

```
person1.sexInfo.sex=male;
person1.sexInfo.u.beard=FALSE;
and
person2.sexInfo.sex=female;
person1.sexInfo.u.children=3;
```

INTERNAL IMPLEMENTATION OF STRUCTURES

• The size of an object of a struct or union type is the amount of storage necessary to represent the largest component, including any padding that may be required.

• Structures must begin and end on the same type of memory boundary. For ex, an even byte boundary (2, 4, 6 or 8).

POLYNOMIALS
ABSTRACT DATA TYPE

• A polynomial is a sum of terms, where each term has a form ax^e,
where x=variable, a=coefficient and e=exponent.

For ex,

$A(x)=3x^{20}+2x^5+4$ and $B(x)=x^4+10x^3+3x^2+1$

• The largest(or leading) exponent of a polynomial is called its degree.

• Assume that we have 2 polynomials,

$A(x)= \sum a_i x^i$ & $B(x)= \sum b_i x^i$ then $A(x)+B(x)= \sum(a_i + b_i)x^i$

Structure *Polynomial* is
objects: $p(x)=a_1x^e + \ldots a_nx^e$; a set of ordered pairs of <*ei,ai*> where *ai* in *Coefficients* and *ei* in *Exponents*, *ei* are integers >= 0
functions:
for all *poly, poly1, poly2* ∈ *Polynomial*, *coef* ∈*Coefficients*, *expon* ∈ *Exponents*
Polynomial Zero( ) ::= **return** the polynomial, *p(x)* = 0
Boolean IsZero(*poly*) ::= **if** (*poly*)
return *FALSE*
else
return *TRUE*
Coefficient Coef(*poly, expon*) ::= **if** (*expon* ∈ *poly*)
return its coefficient
else
return Zero
Exponent Lead_Exp(*poly*) ::= **return** the largest exponent in *poly*
Polynomial Attach(*poly,coef, expon*) ::= **if** (*expon* ∈ *poly*)
return error
else
return the polynomial poly with the term <*coef, expon*> inserted
Polynomial Remove(*poly, expon*)::= **if** (*expon* ∈ *poly*)
return the polynomial *poly* with the term whose exponent is *expon deleted*
else
return error
Polynomial SingleMult(*poly, coef, expon*) ::= **return** the polynomial *poly* • *coef* • *xexpon*
Polynomial Add(*poly1, poly2*) ::= **return** the polynomial *poly*1 +*poly*2
Polynomial Mult(*poly1, poly2*)::= **return** the polynomial *poly*1 • *poly*2
End *Polynomia*

ADT 2.2: Abstract data type Polynomial

POLYNOMIAL REPRESENTATION: FIRST METHOD

```
#define MAX_DEGREE 100
typedef struct
{
        int degree;
        float coef[MAX_DEGREE];
}polynomial;

polynomial a;
```

```
/* d =a + b, where a, b, and d are polynomials */
d = Zero( )
while (! IsZero(a) && ! IsZero(b))
do
{
        switch COMPARE (Lead_Exp(a), Lead_Exp(b))
        {
                case -1: d = Attach(d, Coef (b, Lead_Exp(b)), Lead_Exp(b));
                         b = Remove(b, Lead_Exp(b));
                         break;
                case  0: sum = Coef (a, Lead_Exp (a)) + Coef ( b, Lead_Exp(b));
                         if (sum)
                         {
                                  Attach (d, sum, Lead_Exp(a));
                                 a = Remove(a , Lead_Exp(a));
                                 b = Remove(b , Lead_Exp(b));
                         }
                          break;
                 case 1: d = Attach(d, Coef (a, Lead_Exp(a)), Lead_Exp(a));
                         a = Remove(a, Lead_Exp(a));
         }
}
insert any remaining terms of a or b into d
```

Program 2.5: Initial version of padd function

• If a is of type 'polynomial' then $A(x)= \sum a_i x^i$ can be represented as:

a.degree=n

$a.coeff[i]=a_{n-i}$

• In this representation, we store coefficients in order of decreasing exponents, such that a.coef[i] is the coefficient of x^{n-i} provided a term with exponent n-i exists; otherwise, a.coeff[i]=0 (Figure 2.3a).

• Disadvantage: This representation wastes a lot of space. For instance, if a.degree<<MAX_DEGREE and polynomial is sparse, then we will not need most of the positions in a.coef[MAX_DEGREE] (sparse means number of terms with non-zero coefficient is small relative to degree of the polynomial).

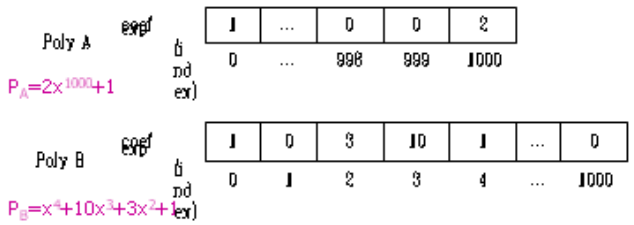


Figure 2.3a: Array representation of two polynomials

POLYNOMIAL REPRESENTATION: SECOND METHOD

```
#define MAX_TERMS 100
typedef struct polynomial
{
        float coef;
        int expon;
}polynomial;

polynomial terms[MAX_TERMS];
        int avail=0;
```

• $A(x)=2x^{1000}+1$ and $B(x)=x^4+10x^3+3x^2+1$ can be represented as shown below.

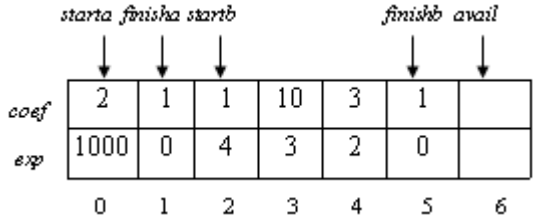


Figure 2.3b: Array representation of two polynomials

• startA & startB give the index of first term of A and B respectively (Figure 2.3b).
finishA & finishB give the index of the last term of A & B respectively
avail gives the index of next free location in the array.
• Any polynomial A that has 'n' non-zero terms has startA & finishA such that finishA=startA+n-1
• Advantage: This representation solves the problem of many 0 terms since $A(x)-2x^{1000}+1$ uses only 6 units of storage (one for startA, one for finishA, 2 for the coefficients and 2 for the exponents)
• Disadvantage: However, when all the terms are non-zero, the current representation requires about twice as much space as the first one.

POLYNOMIAL ADDITION

```
void padd (int starta, int finisha, int startb, int finishb,int * startd, int *finishd)
{
       /* add A(x) and B(x) to obtain D(x) */
       float coefficient;
      *startd = avail;
      while (starta <= finisha && startb <= finishb)
      {
          switch (COMPARE(terms[starta].expon, terms[startb].expon))
          {
            case -1: /* a expon < b expon */
                      attach(terms[startb].coef, terms[startb].expon);
                      startb++
                      break;
             case  0: /* equal exponents */
                      coefficient = terms[starta].coef + terms[startb].coef;
                      if (coefficient)
                          attach (coefficient, terms[starta].expon);
                      starta++;
                      startb++;
                      break;
             case 1: /* a expon > b expon */
                     attach(terms[starta].coef, terms[starta].expon);
                     starta++;
          }
               /* add in remaining terms of  A(x) */
          for( ; starta <= finisha; starta++)
               attach(terms[starta].coef, terms[starta].expon);
               /* add in remaining terms of B(x) */
          for( ; startb <= finishb; startb++)
               attach(terms[startb].coef, terms[startb].expon);
          *finishd =avail -1;
      }
}
```

Program 2.6:Function to add two polynomials

```
void attach(float coefficient, int exponent)
{
     /* add a new term to the polynomial */
     if (avail >= MAX_TERMS)
     {
          fprintf(stderr, "Too many terms in the polynomial\n");
          exit(1);
     }
     terms[avail].coef  = coefficient;
     terms[avail++].expon = exponent;
}
```

Program 2.7: Function to add a new term

ANALYSIS

- Let m and n be the number of non-zero terms in A and B respectively.
- If m>0 and n>0, the while loop is entered.

  At each iteration, we increment the value of startA or startB or both.
- Since the iteration terminates when either startA or startB exceeds finishA or finishB respectively, the number of iterations is bounded by m+n-1.

  This worst case occurs when

  $A(x)=\sum x^{2i}$ and $B(x)=\sum x^{2i+1}$
- The asymptotic computing time of this algorithm is O(n+m)

SPARSE MATRICES

- Sparse matrix contains many zero entries.
- When a sparse matrix is represented as a 2-dimensional array, we waste space (Figure 2.4b).
- For ex, if 100*100 matrix contains only 100 entries then we waste 9900 out of 10000 memory spaces.
- Solution: Store only the non-zero elements.

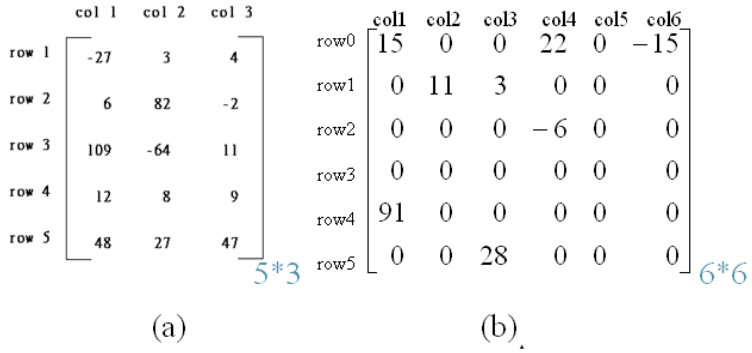


Figure 2.4: Two matrices

Structure *Sparse_Matrix* is
objects: a set of triples, <*row, column, value*>, where *row* and *column* are integers and form a unique combination, and*value* comes from the set *item.*
functions:
for all *a, b* ∈ *Sparse_Matrix*, *x* ∈ *item, i, j, max_col*, *max_row* ∈ *index*
Sparse_Marix Create(*max_row, max_col*) ::=
return a *Sparse_matrix* that can hold up to *max_items* = *max _row* × *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col.*
Sparse_Matrix Transpose(*a*) ::=
return the matrix produced by interchanging the row and column value of every triple.
Sparse_Matrix Add(*a, b*) ::=
if the dimensions of a and b are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else
return error
Sparse_Matrix Multiply(*a, b*) ::=
if number of columns in a equals number of rows in b
return the matrix *d* produced by multiplying a by *b* according to the formula:*d* [*i*] [*j*]=Σ(a[i][k]•b[k][j]) where *d (i, j)* is the *(i,j)*th element
else
return error.
End *Sparse_Matrix*

ADT 2.3: Abstract data type SparseMatrix

SPARSE MATRIX REPRESENTATION

• We can classify uniquely any element within a matrix by using the triple <row,col,value>. Therefore, we can use an array of triples to represent a sparse matrix (Figure 2.5).

```
SpareMatrix Create(maxRow,maxCol) ::=
        #define MAX_TERMS 101
        typedef struct term
        {
            int col;
            int row;
            int value;
        } term;

term a[MAX_TERMS];
```

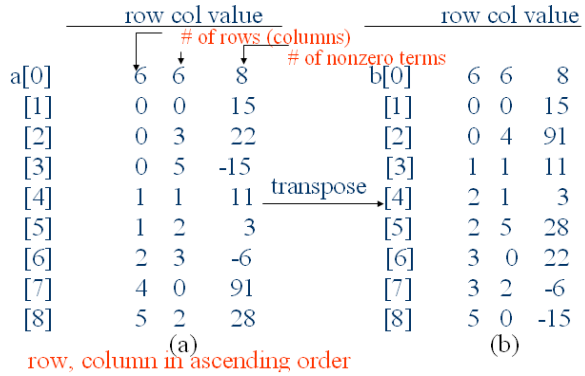


Figure 2.5: Sparse matrix and its transpose stored as triples

• a[0].row contains the number of rows;
a[0].col contains number of columns and
a[0].value contains the total number of nonzero entries.

TRANSPOSING A MATRIX

• To transpose a matrix ,we must interchange the rows and columns.
• Each element a[i][j] in the original matrix becomes element b[j][i] in the transpose matrix.
• Algorithm To transpose a matrix:

```
for all elements in column j
place element <i,j,value> in
element <j,i,value>
```

```
void transpose (term a[], term b[])
{       /* b is set to the transpose of a */
        int n, i, j, currentb;
        n = a[0].value;  /* total number of elements */
        b[0].row = a[0].col;  /* rows in b = columns in a */
        b[0].col = a[0].row;  /*columns in b = rows in a */
        b[0].value = n;
        if (n > 0)
        {           /*non zero matrix */
            currentb = 1;
            for (i = 0; i < a[0].col; i++)   /* transpose by columns in a */
                for( j = 1; j <=  n; j++)  /* find elements from the current column */
                    if (a[j].col == i)
                    {                   /* element is in current column, add it to b */
                        b[currentb].row = a[j].col;
                        b[currentb].col  = a[j].row;
                        b[currentb].value = a[j].value;
                        currentb++
                    }
        }
}
```

Program 2.7: Transpose of a sparse matrix

```
void fast_transpose(term a[ ], term b[ ])
{       /* the transpose of a is placed in b */
        int row_terms[MAX_COL], starting_pos[MAX_COL];
        int i, j, num_cols = a[0].col, num_terms = a[0].value;
        b[0].row = num_cols; b[0].col = a[0].row;
        b[0].value = num_terms;
        if (num_terms > 0)
        {       /*nonzero matrix*/
                for (i = 0; i < num_cols; i++)
                row_terms[i] = 0;
                for (i = 1; i  <= num_terms; i++)
                row_term [a[i].col]++
                starting_pos[0] = 1;
                for (i =1; i < num_cols; i++)
                starting_pos[i]=starting_pos[i-1] +row_terms [i-1];
                for (i=1; i <= num_terms, i++)
                {
                        j = starting_pos[a[i].col]++;
                        b[j].row = a[i].col;
                        b[j].col = a[i].row;
                        b[j].value = a[i].value;
                 }
        }
}
```

Program 2.8: Fast transpose of a sparse matrix

```
   void mmult (term a[ ], term b[ ], term d[ ] )
   {     /* multiply two sparse matrices */
        int i, j, column, totalb = b[].value, totald = 0;
        int rows_a = a[0].row, cols_a = a[0].col,
        totala = a[0].value; int cols_b = b[0].col,
        int row_begin = 1, row = a[1].row, sum =0;
        int new_b[MAX_TERMS][3];
        if (cols_a != b[0].row)
        {
             fprintf (stderr, "Incompatible matrices\n");
             exit (1);
        }
        fast_transpose(b, new_b);
        a[totala+1].row = rows_a;          /* set boundary condition */
        new_b[totalb+1].row = cols_b;
        new_b[totalb+1].col = 0;
        for (i = 1; i <= totala; )
        {
                column = new_b[1].row;
                for (j = 1; j <= totalb+1;)
                {        /* mutiply row of a by column of b */
                         if (a[i].row != row)
                         {
                                 storesum(d, &totald, row, column, &sum);
                                 i = row_begin;
                                 for (; new_b[j].row == column; j++);
                                          column =new_b[j].row
                         }
                         else
                             switch (COMPARE (a[i].col, new_b[j].col))
                             {
                                 case -1:  i++; break; /* go to next term in a */
                                 case 0: /* add terms, go to next term in a and b */
                                           sum += (a[i++].value * new_b[j++].value);
                                 case 1: j++   /* advance to next term in b*/

                             }
                }        /* end of for j <= totalb+1 */
                for (; a[i].row == row; i++);
                      row_begin = i; row = a[i].row;
        }       /* end of for i <=totala */
        d[0].row = rows_a;
        d[0].col = cols_b; d[0].value = totald;
}
```

Praogram 2.9: Sparse matrix multiplication

```
void storesum(term d[ ], int *totald, int row, int column, int *sum)
{
        /* if *sum != 0, then it along with its row and column
        position is stored as the *totald+1 entry in d */
         if (*sum)
              if (*totald < MAX_TERMS)
              {
                  d[++*totald].row = row;
                  d[*totald].col = column;
                  d[*totald].value = *sum;
               }
              else
              {
                fprintf(stderr, "Numbers of terms in product exceed %d\n", MAX_TERMS);
                exit(1);
              }
        }
}
```

Program 2.10: storsum function

UNIT 3: STACKS AND QUEUES

STACK

• This is an ordered-list in which insertions(called *push*) and deletions(called *pop*) are made at one end called the top (Figure 3.1).

• Since last element inserted into a stack is first element removed, a stack is also known as a LIFO list(Last In First Out).

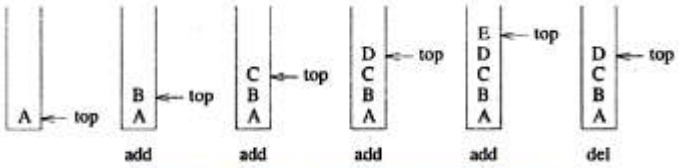


Figure 3.1: Inserting and deleting elements in a stack

SYSTEM STACK

• A stack used by a program at run-time to process function-calls is called system-stack (Figure 3.2).

• When functions are invoked, programs
  → create a stack-frame(or activation-record) &
  → place the stack-frame on top of system-stack

• Initially, stack-frame for invoked-function contains only
  → pointer to previous stack-frame &
  → return-address

• The previous stack-frame pointer points to the stack-frame of the invoking-function while return-address contains the location of the statement to be executed after the function terminates.

• If one function invokes another function, local variables and parameters of the invoking-function are added to its stack-frame.

• A new stack-frame is then
  → created for the invoked-function &
  → placed on top of the system-stack

• When this function terminates, its stack-frame is removed (and processing of the invoking-function, which is again on top of the stack, continues).

• Frame-pointer(fp) is a pointer to the current stack-frame.

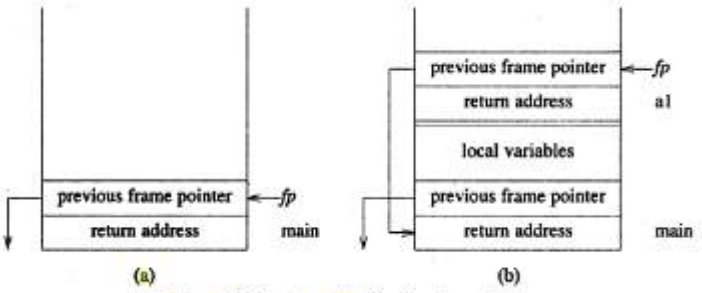


Figure 3.2: System stack after function call

```
structure Stack is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all stack ∈ Stack, item ∈ element, max_stack_size ∈ positive integer
   Stack CreateS(max_stack_size) ::=
            create an empty stack whose maximum size is max_stack_size

   Boolean IsFull(stack, max_stack_size) ::=
            if(number of elements in stack == max_stack_size)
                return TRUE
            else
                return FALSE

   Stack Add(stack, item) ::=
            if (IsFull(stack))
                stack_full
            else
                insert item into top of stack and return

    Boolean IsEmpty(stack) ::=
            if(stack == CreateS(max_stack_size))
                return TRUE
            else
                return FALSE

    Element Delete(stack) ::=
           if(IsEmpty(stack))
               return
           else
               remove and return the item on the top of the stack
```

ADT Stack 3.1

- Here, MAX_STACK_SIZE=maximum number of entries.
- The first element of the stack is stored in stack[0].
- 'top' points to the top element in the stack (top=-1 to denote an empty stack).
- The CreateS() function can be implemented as follows:

```
Stack CreateS(maxStackSize)::=
        #define MAX_STACK_SIZE 100
        struct element
        {
                int key;
        };
element stack[MAX_STACK_SIZE];
int top=-1;
Boolean IsEmpty(Stack)::= top<0;                          //used to check if stack is empty
Boolean IsFull(Stack) ::= top>=MAX_STACK_SIZE-1;          //used to check if stack is full
```

```
void add(int top, element item)
{
    if (top >= MAX_STACK_SIZE-1)
   {
        stack_full( );
        return;
    }
    stack[++top] = item;
}
```

Program 3.1: Add an item to a stack

```
element delete(int top)
{
    if (top == -1)
        return stack_empty( );  /* returns and error key */
    return stack[(top)--];
 }
```

Program 3.2: Delete from a stack

- Function push() checks to see if the stack is full. If it is, it calls stackFull, which prints an error message and terminates execution.
- When the stack is not full, we increment top and assign item to stack[top].

STACK USING DYNAMIC ARRAYS

• Shortcoming of static stack implementation: is the need to know at compile-time, a good bound(MAX_STACK_SIZE) on how large the stack will become.

• This shortcoming can be overcome by

→ using a dynamically allocated array for the elements &

→ then increasing the size of the array as needed

• Initially, capacity=1 where capacity=maximum no. of stack-elements that may be stored in array.

• The CreateS() function can be implemented as follows

```
Stack CreateS()::=
        struct element
        {
                int key;
        };

element *stack;
MALLOC(stack,sizeof(*stack));
int capacity=-1;
int top=-1;
Boolean IsEmpty(Stack)::= top<0;
Boolean IsFull(Stack) ::= top>=capacity-1;

void stackFull()
{
        REALLOC(stack,2*capacity*sizeof(*stack));
        capacity=2*capacity;
}
```

• Once the stack is full, realloc() function is used to increase the size of array.

• In array-doubling, we double array-capacity whenever it becomes necessary to increase the capacity of an array.

ANALYSIS

• In worst case, the realloc function needs to

→ allocate 2*capacity*sizeof(*stack) bytes of memory and

→ copy capacity*sizeof(*stack) bytes of memory from the old array into the new one.

• The total time spent over all array doublings = $O(2^k)$ where capacity=2^k

• Since the total number of pushes is more than 2^{k-1}, the total time spend in array doubling is O(n) where n=total number of pushes.

QUEUES

- This is an ordered-list in which insertions & deletions take place at different ends (Figure 3.4).
- The end at which new elements are added is called the rear & the end from which old elements are deleted is called the front.
- Since first element inserted into a queue is first element removed, queues are known as FIFO lists.

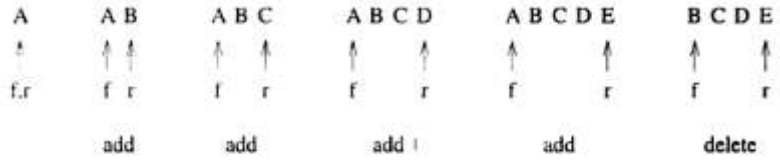


Figure 3.4: Inserting and deleting elements in a queue

```
structure Queue is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all queue ∈ Queue, item ∈ element, max_ queue_ size ∈ positive integer
    Queue CreateQ(max_queue_size) ::=
          create an empty queue whose maximum size is max_queue_size

    Boolean IsFullQ(queue, max_queue_size) ::=
          if(number of elements in queue == max_queue_size)
               return TRUE
          else
               return FALSE

    Queue AddQ(queue, item) ::=
          if (IsFullQ(queue))
               queue_full
          else
               insert item at rear of queue and return queue

     Boolean IsEmptyQ(queue) ::=
          if (queue ==CreateQ(max_queue_size))
               return TRUE
          else
               return FALSE

    Element DeleteQ(queue) ::=
          if (IsEmptyQ(queue))
               return
          else
               remove and return the item at front of queue
```

ADT 3.2: Abstract data type Queue

- The CreateQ() function can be implemented as follows

```
Queue CreateQ(maxQueueSize)::=
        #define MAX_QUEUE_SIZE 100
        struct element
        {
                int key;
        };

        element queue[MAX_QUEUE_SIZE];
        int rear=-1;
        int front=-1;
        Boolean IsEmptyQ(queue)::= front==rear;
        Boolean IsFullQ(queue)::= rear=MAX_QUEUE_SIZE-1;
```

```
void addq(int rear, element item)
{
   if (rear == MAX_QUEUE_SIZE_1)
   {
     queue_full( );
     return;
   }
   queue [++rear] = item;
}
```

Program 3.5: Add to a queue

```
element deleteq(int front, int rear)
{
   if ( front == rear)
         return queue_empty( );
   return queue [++ front];
}
```

Program 3.6: Delete from a queue

JOB SCHEDULING

- Queues are used for the creation of a job-queue by an operating-system (Figure 3.5).
- If operating-system does not use, then the jobs are processed in the order they enter the system.

front	rear	Q[0]	[1]	[2]	[3]	[4]	[5]	[6]	...	Comments
−1	−1		queue		empty					initial
−1	0	J1								Job 1 joins Q
−1	1	J1	J2							Job 2 joins Q
−1	2	J1	J2	J3						Job 3 joins Q
0	2		J2	J3						Job 1 leaves Q
0	3		J2	J3	J4					Job 4 joins Q
1	3			J3	J4					Job 2 leaves Q

Figure 3.5: Insertion and deletion from a queue

CIRCULAR QUEUE

- In a circular-queue, the elements are arranged implicitly in a circle (Figure 3.7).
- When the array is viewed as a circle, each array position has a next and a previous position.

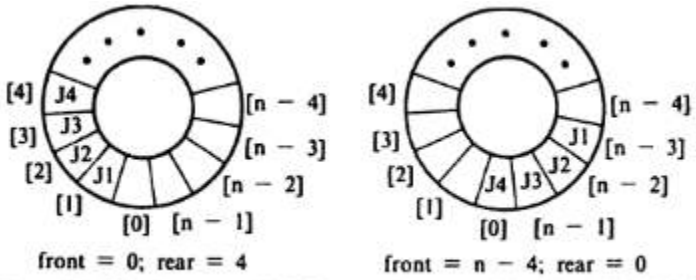


Figure 3.7: Circular queue of *MaxSize* = *n* elements and four jobs: J1, J2, J3, J4

```
void addq(int front, int rear, element item)
{
   rear = (rear +1) % MAX_QUEUE_SIZE;
    if (front == rear)              /* reset rear and print error */
         return;
    queue[rear] = item;
}
```

Program 3.7: Add to a circular queue

```
element deleteq(int front, int rear)
{
  element item;
  if (front == rear)
        return queue_empty( );                  /* queue_empty returns an error key */
  front = (front+1) % MAX_QUEUE_SIZE;
  return queue[front];
}
```

Program 3.8: Delete from a circular queue

CIRCULAR QUEUES USING DYNAMICALLY ALLOCATED ARRAYS

• Shortcoming of static queue implementation: is the need to know at compile-time, a good bound(MAX_QUEUE _SIZE) on how large the queue will become.
• This shortcoming can be overcome by
→ using a dynamically allocated array for the elements &
→ then increasing the size of the array as needed
• In array-doubling, we double array-capacity whenever it becomes necessary to increase the capacity of an array (Figure 3.8).

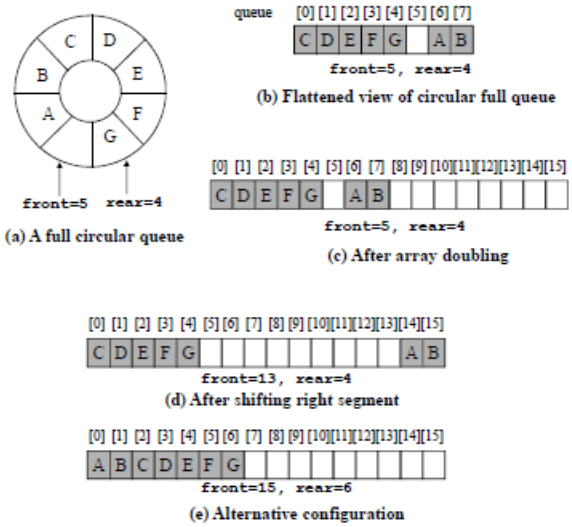


Figure 3.8: Doubling queue capacity

• To get a proper circular queue configuration, we must slide the elements in the right segment to the right end of the array (Figure 3.8d).
• The array doubling and the slide to the right together copy at most 2*capacity-2 elements.
• The number of elements copied can be limited to capacity-1 by customizing the array doubling code so as to obtain the configuration of Figure3.8e. This configuration may be obtained as follows:
1) Create a new array newQueue of twice the capacity.
2) Copy the second segment to positions in newQueue begining at 0.
3) Copy the first segment to positions in newQueue begining at capacity-front-1.

```
void addq(element item)
{
  /* add an item to the queue */
  rear = (rear + 1)% capacity;
  if (front == rear)
      queueFull();
  queue[rear] = item;
}
```

Program 3.9: Add to a circular queue

```
Void queueFull()
{
  /* allocate an array with twice the capacity */
  element * newQueue;
  MALLOC(newQueue, 2*capacity*sizeof(*queue));

  /* copy from queue to newQueue*/
  int start = (front + 1) % capacity;
  if(start < 2)
  copy(queue+start, queue+start+capacity-1, newQueue );
  else
    {
        copy(queue+start, queue+capacity, newQueue );
        copy(queue, queue+rear+1, newQueue+capacity-start );
    }
 /* Switch to newQueue */

 front = 2*capacity-1;
 rear = capacity-2;
 capacity = capacity * 2;
 free(queue);
 queue = newQueue;
```

Program 3.10: Doubling queue capacity

```
precedence get_token(char *symbol, int *n)
{
  *symbol =expr[(*n)++];
    switch (*symbol)
   {
           case '(' : return lparen;
           case ')' : return rparen;
           case '+': return plus;
           case '-' : return minus;
           case '/' :  return divide;
           case '*' : return times;
           case '%' : return mod;
           case '\0' : return eos;
           default  : return operand;
    }
}
```

Program 3.14: Function to get a token from the input string

INFIX TO POSTFIX

- Algorithm for producing a postfix expression from an infix is as follows (Figure 3.14):
  1) Fully parenthesize the expression
  2) Move all binary operators so that they replace their corresponding right parentheses
  3) Delete all parentheses
- For ex, a/b-c+d*e-a*c when fully parenthesized becomes:
  ((((a/b)-c)+(d*e))-a*c))
- Performing steps 2 and 3 gives
  ab/c-de*+ac*-
- This algorithm is inefficient on a computer because it requires 2 passes.
- The first pass reads the expression and parenthesizes it
  while the second pass moves the operators.
- Since the order of operands is same in infix and postfix, we can form the postfix equivalent by scanning the infix expression left-to-right.
- During this scan, operands are passed to the output expression as they are encountered.
- However, the order in which the operators are output depends on their precedence (Figure 3.12).
- Since we must output the higher precedence operators first, we save operators until we know their correct placement. A stack is one way of doing this.

Infix	Postfix
2+3*4	2 3 4*+
a*b+5	ab*5+
(1+2)*7	1 2+7*
a*b/c	ab*c/
((a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*+ac*-

Figure 3.14: Infix and postfix notation

priority	operator
1	unary minus, !
2	*, /, %
3	+, −
4	<, <=, >=, >
5	==, !=
6	&&
7	||

Figure 3.12: Priority of operators

Example 3.3[Simple expression]: Consider simple expression a+b*c which yields abc*+ in postfix.

- The operands are output immediately, but the two operators need to be reversed (Figure 3.15):
- In general, operators with higher precedence must be output before those with lower precedence. Therefore, we stack operators as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator.

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

Figure 3.15:Translation of a+b*c to postfix

EVALUATING POSTFIX EXPRESSIONS

• In infix notation, a binary-operator is in-between 2 operands.
In postfix notation, a binary-operator follows the 2 operands.
In prefix notation, a binary-operator precedes the 2 operands.

• Although infix notation is the most common way of writhing expressions, it is not the one used by compilers to evaluate expressions.

• Instead compilers typically use a parenthesis-free postfix notation.

• Steps for evaluating postfix expression (Figure 3.13):

1) Scan the symbol from left to right.
2) If the scanned-symbol is an operand, push it on to the stack.
3) If the scanned-symbol is an operator, pop 2 elements from the stack. And perform the indicated operation.
4) Push the result on to the stack.
5) Repeat the above procedure till the end of input is encountered.

priority	operator
1	unary minus, !
2	*, /, %
3	+, −
4	<, <=, >=, >
5	==, !=
6	&&
7	||

Figure 3.12: Priority of operators

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Figure 3.13: Postfix evaluation of 6 2/3-4 2*+

```
int eval(void)
{
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0;                    /* counter for the expression string */
  int top = -1;
  token = get_token(&symbol, &n);
  while (token != eos)
  {
          if (token == operand)
                  add(&top, symbol-'0');   /* stack add */
          else
          {
                  /* remove two operands, perform operation, and
                  return result to the stack */
                  op2 = delete(&top);  /* stack delete */
                  op1 = delete(&top);
                  switch(token)
                  {
                          case plus: add(&top, op1+op2); break;
                          case minus: add(&top, op1-op2); break;
                          case times: add(&top, op1*op2); break;
                          case divide: add(&top, op1/op2); break;
                          case mod: add(&top, op1%op2);
                   }
            }
             token = get_token (&symbol, &n);
  }
   return delete(&top); /* return result */
}
```

Program 3.13: Function to evaluate a postfix expression

Example 3.4[Parenthesized expression]: Consider the expression a*(b+c)*d, which yields abc+*d* in postfix (Figure 3.16).

- We stack operators until we reach the right parenthesis.
- At this point, we unstack until we reach the corresponding left parenthesis.
- We then delete the left parenthesis from the stack.(The right parenthesis is never put on the stack).

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos	*			0	abc+*d*

Figure 3.16:Translation of a+(b*c)*d to postfix

- We have two types of precedence:
  1) in-stack precedence(isp)
  2) incoming precedence(icp).
- The declarations that establish these precedences are:

```
/* isp and icp arrays -- index is value of precedence lparen ,rparen, plus, minus ,times, divide, mod, eor */
int isp[]={0,19,12,12,13,13,13,0};
int icp[]={20,19,12,12,13,13,13,0};
```

- We remove an operator from stack only if its in-stack-precedence(isp) is greater than or equal to the incoming precedence(icp) of the new operator.

```
void postfix(void)
{
        /* output the postfix of the expression. The expression string, the stack, and top are global */
          char symbol;
          precedence token;
          int n = 0;
          int top = 0;              /* place eos on stack */
          stack[0] = eos;
          for (token = get _token(&symbol, &n); token != eos; token = get_token(&symbol, &n))
          {
            if (token == operand)
                    printf ("%c", symbol);
            else if (token == rparen )
                    {
                            /*unstack tokens until left parenthesis */
                            while (stack[top] != lparen)
                                    print_token(delete(&top));
                            delete(&top);           /*discard the left parenthesis */
                    }
                    else
                    {
                            /* remove and print symbols whose isp is greater
                            than or equal to the current token's icp */
                            while(isp[stack[top]] >= icp[token] )
                                    print_token(delete(&top));
                            add(&top, token);
                     }
           }
          while ((token = delete(&top)) != eos)
                    print_token(token);
          print("\n");
}
```

Program 3.15:Function to convert Infix to Postfix

MULTIPLE STACKS

- Assume that we have 'n' stacks, we can divide the available memory into 'n' segments (Fig: 3.18).
- Let 'i' = stack number of one of n stacks.
- Let boundary[i] (0<=i<MAX_STACKS) points to position immediately to left of bottom element of stack i, while top[i] (0<=i<MAX_STACKS) points to top element.
- Stack i is empty iff boundary[i]=top[i]
- The relevant declaration are:

```
#define MEMORY_SIZE 100                 /* size of memory */
#define MAX_STACKS 10                   /* max number of stacks plus 1 */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int n;                                  /* number of stacks entered by the user */
```

- To divide the array into equal segment, we use the following code:

```
top[0]=boundary[0]=-1;
for(j=1;j<n;j++)
top[j]=boundary[j]=(MEMORY_SIZE/n)*j;
boundary[n]=MEMORY_SIZE-1;
```

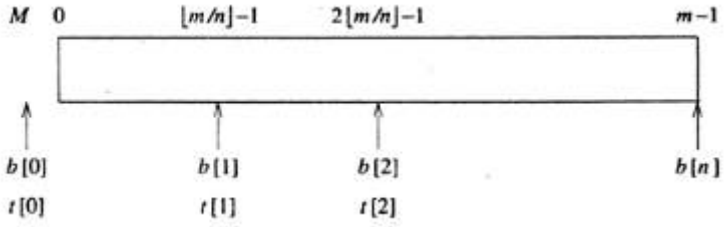


Figure 3.18: Initial configuration for n stacks in memory[m]

```
void add(int i, element item)
{
   /* add an item to the ith stack */
   if (top[i] == boundary [i+1])
      stack_full(i);     may have unused storage
      memory[++top[i]] = item;
}
```

Program 3.16:Add an item to the ith stack

```
element delete(int i)
{
   /* remove top element from the ith stack */
   if (top[i] == boundary[i])
      return stack_empty(i);
   return memory[top[i]--];
}
```

Program 3.17:Delete an item from the ith stack

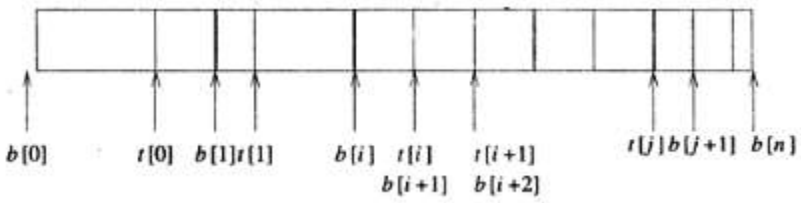


Figure 3.19:Configuration when stack i meets stack i+1,but the memory is not full

- In push function, top[i]==boundary[i+1] condition implies only that a particular stack ran out of memory, not that the entire memory is full.(In fact, there may be a lot of unused space between other stacks in array memory).
- Therefore, we create an error recovery function, stackFull, which determines if there is any free space in memory.
- If there is space available, it should shift the stacks so that space is allocated to the full stack.
- We can guarantee that stackFull adds elements as long as there is free space in array memory if we:

  1) Determine the least j, i<j<n such that there is free space between stacks j and j+1 i.e. top[j]<boundary[j+1]. If there is such a j, then move stacks i+1,i+2 . . . . . .j one position to the right. This creates a space between stacks i and i+1.

  1) If there is no j as in (i),then look to the left of stack i. Find the largest j such that 0<=j<i and there is space between stacks j and j+1 i.e. top[j]<boundary[j+1]. If there is such a j, then move stacks j+1,j+2. . . . . .i one space to the left. This also creates a space between stacks i and i+1.

  3) If there is no j satisfying either (i) or (ii),then all MEMORY_SIZE spaces of memory are utilized and there is no free space. In this case, stackFull terminates with an error message.

UNIT 5: TREES

TREE

- This is a finite set of one or more nodes such that
  1) There is a specially designated node called *root*.
  2) Remaining nodes are partitioned into disjoint sets T_1, T_2. . . . . T_n where each of these are called subtrees of root(Figure 5.2).
- Consider the tree shown below

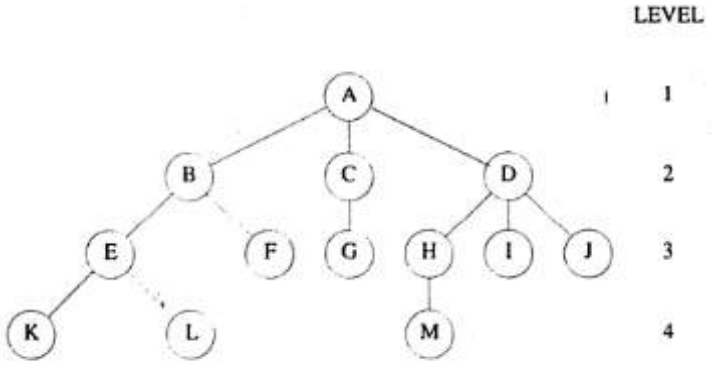


Figure 5.2: A sample tree

TERMINOLOGIES USED IN A TREE

- ***Node*** contains
  → item of information &
  → links to other nodes
- Number of subtrees of a node is called its ***degree***.
  For e.g., degree of A=3; degree of C=1
- Nodes with degree=0 are called ***terminal(leaf) nodes*** (For e.g., K, L, F, G, M, I, J)
  whereas other nodes are referred to as ***non-terminals*** (For e.g., B, E, F, C, H, I, J).
- The subtrees of a node A are the ***children*** of A. A is the ***parent*** of its children.
  For e.g., children of D are H, I and J. Parent of D is A.
- Children of same parent are called ***siblings***.
  For e.g., H, I and J are siblings.
- ***Degree of a tree*** is maximum of the degree of the nodes in the tree.
  Degree of given tree=3.
- ***Ancestors*** of a node are all nodes along the path from root to that node.
  For e.g., ancestors of M are A, D and H.
- If a node is at level 'l', then its children are at level 'l+1'.
  Height(or depth) of a tree is defined as maximum level of any node in the tree.
  For e.g., Height of given tree = 4.

REPRESENTATION OF TREES

- A tree can be represented in three forms, namely:
  1) List representation
  2) Left-child right-sibling representation
  3) Degree-two tree representation (Binary Tree)

LIST REPRESENTATION

- Consider the tree shown below

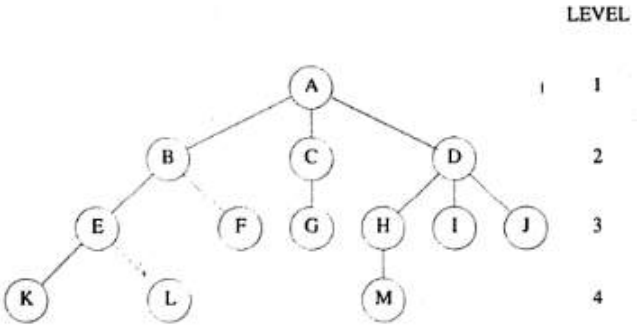
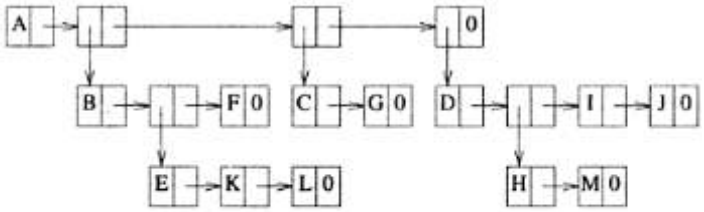


Figure 5.2: A sample tree

- The tree can be drawn as a list: (A(B(E(K,L),F),C(G),D(H(M),I,J)))
- The information in the root node comes first, followed by a list of subtrees of that node.
- Each tree-node can be represented by a memory-node that has
  → fields for data &
  → pointers to children of tree-node (Figure 5.3)



tag fields not shown

Figure 5.3: List representation of the tree

- For a tree of degree 'k', we can use the node-structure as shown below (Figure 5.4).

Figure 5.4: Possible node structure for a tree of degree *k*

LEFT CHILD-RIGHT SIBLING REPRESENTATION

• Figure 5.5 shows the node-structure used in left child-right sibling representation.

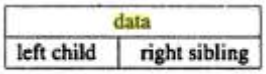


Figure 5.5: Left child-right sibling node structure

• Left-child field of each node points to its leftmost child(if any),and
right-sibling field points to the closest right sibling(if any). (Figure 5.6).

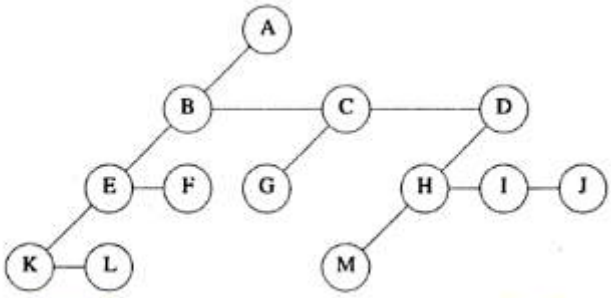


Figure 5.6: Left child-right sibling representation of tree

DEGREE-TWO TREE REPRESENTATION

• In this representation, we refer to 2 children of a node as left & right children (Fig 5.7).
• Left child-right child trees are also known as binary trees.

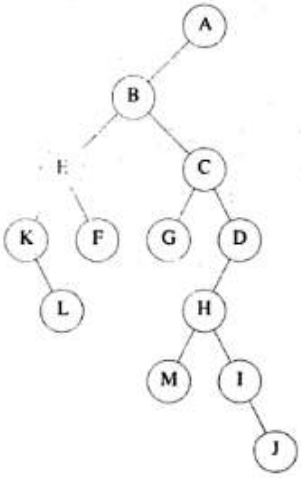


Figure 5.7: Left child-right child tree representation of tree

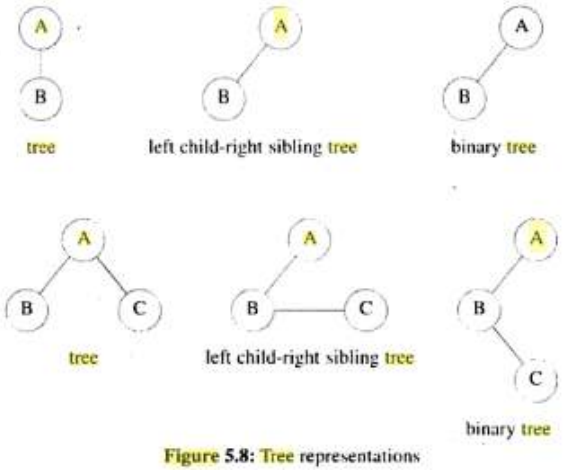


Figure 5.8: Tree representations

BINARY TREE

• This is a finite set of nodes that is either empty or consists of
→ a root &
→ two disjoint binary trees called left subtrees and right subtrees

```
structure Binary_Tree(abbreviated BinTree) is
objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree,
and right Binary_Tree.
functions:
for all bt, bt1, bt2 ∈ BinTree, item ∈ element
Bintree Create()::= creates an empty binary tree
Boolean IsEmpty(bt)::= if (bt==empty binary tree)
                          return TRUE
                       else
                          return FALSE
BinTree MakeBT(bt1, item, bt2)::= return a binary tree whose left subtree is bt1, whose
                          right subtree is bt2, and whose root node contains the data item
Bintree Lchild(bt)::= if (IsEmpty(bt))
                          return error
                       else
                          return the left subtree of bt
element Data(bt)::= if (IsEmpty(bt))
                          return error
                       else
                          return the data in the root node of bt
Bintree Rchild(bt)::= if (IsEmpty(bt))
                          return error
                       else
                          return the right subtree of bt
```

ADT 5.1: Abstract data type Binary_Tree

• Difference between a binary tree and a tree:
1) There is no tree having zero nodes, but there is an empty binary tree.
2) In a binary tree, we distinguish between the order of the children while in a tree we do not.

TYPES OF BINARY TREE

1) Skewed tree is a tree consisting of only left subtree or only right subtree (Figure 5.10a).
2) Full binary tree is a binary tree of depth k having 2^k-1 nodes, k>=0 (Figure 5.10b).
3) Complete tree is a binary tree in which every level except possibly last level is completely filled. A binary tree with n nodes & depth k is complete iff its nodes correspond to nodes numbered from 1 to n in full binary tree of depth k (Figure 5.11).

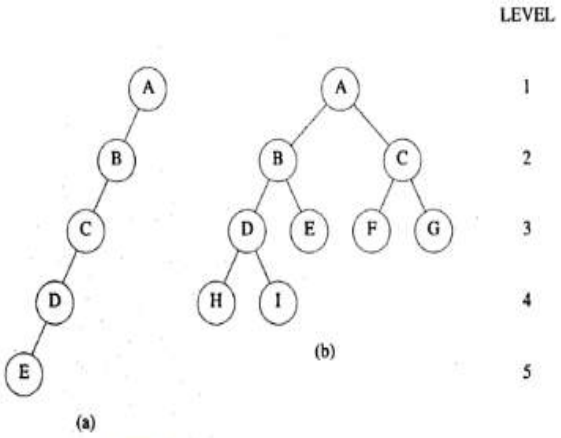


Figure 5.10: Skewed and complete binary trees

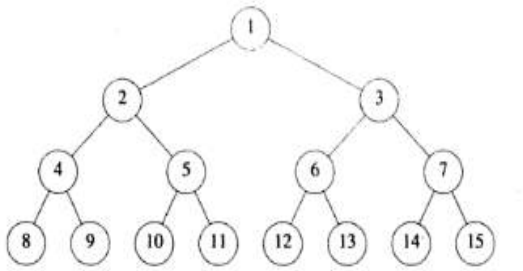


Figure 5.11: Full binary tree of depth 4 with sequential node numbers

PROPERTIES OF BINARY TREES

- The maximum number of nodes on level 'i' of a binary tree is 2^{i-1}, i>=1.
  (For e.g. maximum number of nodes on level $4=2^{4-1}=2^3=8$).
- The maximum number of nodes in a binary tree of depth 'k' is 2^k-1, k>=1.
  (For e.g. maximum number of nodes with depth $4=2^4-1=16-1=15$).
- Relation between number of leaf nodes and degree-2 nodes: For any non-empty binary tree 'T', if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$.

BINARY TREE REPRESENTATIONS

- A binary tree can be represented in two forms, namely:
  1) Array Representation
  2) Linked Representation

ARRAY REPRESENTATION

- We can use a one-dimensional array to store nodes of binary tree (Figure 5.12).
- If a complete binary tree with 'n' nodes is represented sequentially,
  then for any node with index i(1<=i<=n) ,we have
  1) parent(i) is at [i/2] if i!=1.
     If i=1, i is the root and has no parent.
  2) leftChild(i) is at 2i if 2i<=n.
     If 2i>n, then i has no left child.
  3) rightChild(i) is at 2i+1<=n.
     If 2i+1>=n, then i has no right child.
- Consider the tree shown below

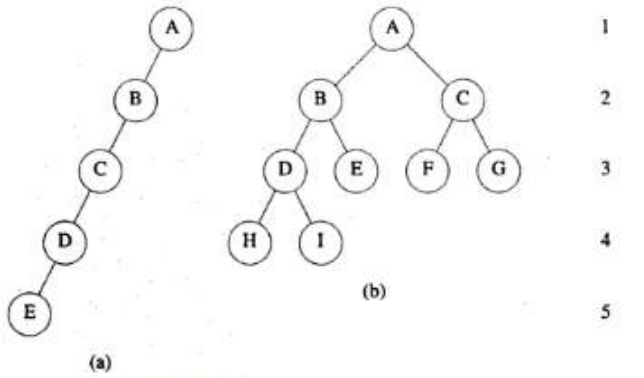


Figure 5.10: Skewed and complete binary trees

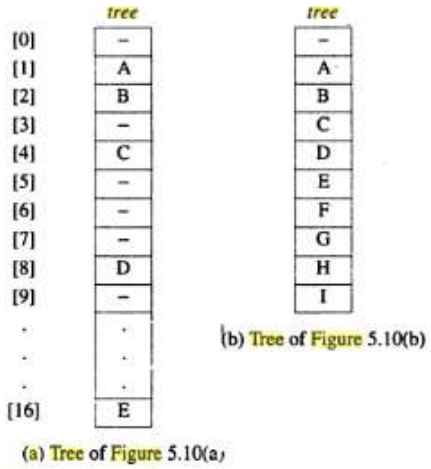


Figure 5.12: Array representation of the binary trees

- Advantage: For complete binary tree, array representation is ideal, as no space is wasted.
  Disadvantage: For skewed tree, less than half the array is utilized. In the worst case, a skewed tree of depth k will require 2^k-1 spaces. Of these, only k will be used.

LINKED REPRESENTATION

• Shortcoming of array representation: Insertion and deletion of nodes from middle of a tree requires movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be overcome easily through the use of a linked representation (Figure 5.14).

• Each node has three fields:

1) leftChild,
2) data and
3) rightChild (Figure 5.13).

```
typedef struct node *treePointer;
typedef struct
{
        int data;
        treePointer leftChild,rightChild;
}node;
```

• Root of tree is stored in the data member 'root' of Tree. This data member serves as access-pointer to the tree.

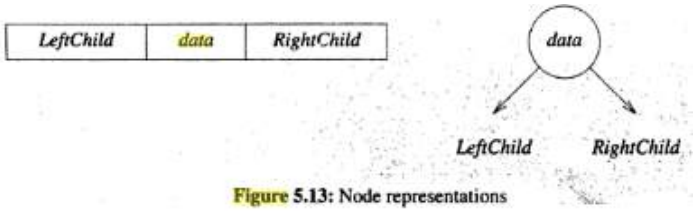


Figure 5.13: Node representations

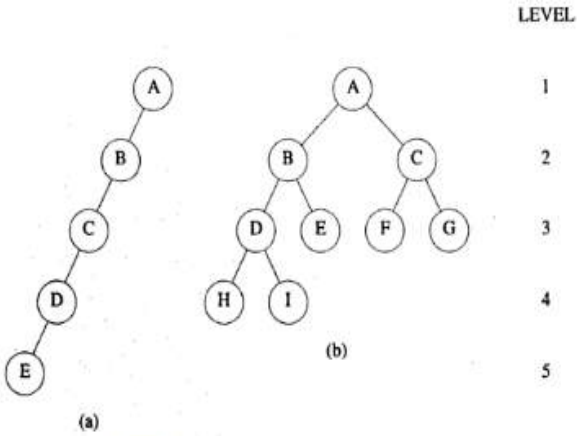


Figure 5.10: Skewed and complete binary trees

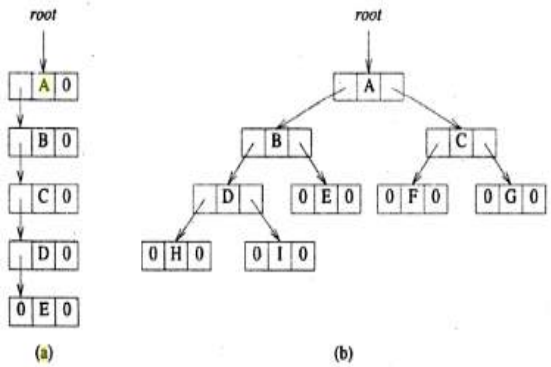


Figure 5.14: Linked representation for the binary trees

BINARY TREE TRAVERSALS

- Tree traversal refers to process of visiting all nodes of a tree exactly once (Figure 5.16).
- There are 3 techniques, namely:
  1) Inorder traversal(LVR); 2) Preorder traversal(VLR); 3) Postorder traversal(LRV). (Let L= moving left, V= visiting node and R=moving right).
- In postorder, we visit a node after we have traversed its left and right subtrees.
  In preorder, the visiting node is done before traversal of its left and right subtrees.
  In inorder, firstly node's left subtrees is traversed, then node is visited and finally node's right subtrees is traversed.

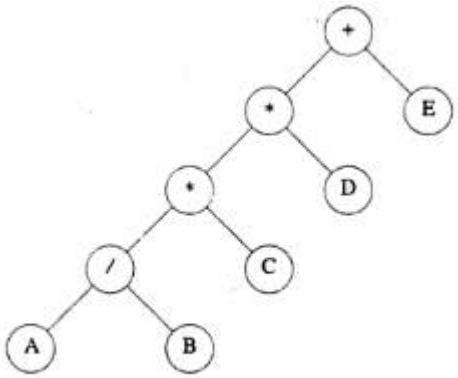


Figure 5.16: Binary tree with arithmetic expression

INORDER TRAVERSAL

- Inorder traversal calls for moving down tree toward left until you can go no farther (Program 5.1).
- Then, you "visit" the node, move one node to the right and continue.
- If you cannot move to the right, go back one more node.

Call of *inorder*	Value in *CurrentNode*	Action	Call of *inorder*	Value in *CurrentNode*	Action
Driver	+		10	C	
1	*		11	0	
2	*		10	C	cout << 'C'
3	/		12	0	
4	A		1	*	cout << '*'
5	0		13	D	
4	A	cout << 'A'	14	0	
6	0		13	D	cout << 'D'
3	/	cout << '/'	15	0	
7	B		Driver	+	cout << '+'
8	0		16	E	
7	B	cout << 'B'	17	0	
9	0		16	E	cout << 'E'
2	*	cout << '*'	18	0	

Figure 5.17: Trace of Program 5.1

```
void inorder(tree_pointer ptr)
{       /* inorder tree traversal */
        if (ptr)
        {
              inorder(ptr->left_child);
              printf("%d", ptr->data);
              indorder(ptr->right_child);
        }
}
```

Program 5.1: Inorder traversal of binary tress

- Each step of the trace shows the call of inorder, the value in the root, and whether or not the printf function is invoked (Figure 5.17).
- Since there are 19 nodes in the tree, inorder() is invoked 19 times for the complete traversal. The nodes of figure 5.16 would be output in an inorder as

  A/B*C*D+E

PREORDER TRAVERSAL

• Visit a node, traverse left, and continue (Program 5.2).

• When you cannot continue, move right and begin again or move back until you can move right and resume.

• The nodes of figure 5.16 would be output in preorder as

+**/ABCDE

```
void preorder(tree_pointer ptr)
{          /* preorder tree traversal */
          if (ptr)
          {
                    printf("%d", ptr->data);
                    preorder(ptr->left_child);
                    predorder(ptr->right_child);
           }
}
```

Program 5.2: Preorder traversal of binary tress

POSTORDER TRAVERSAL

• Visit a node, traverse right, and continue (Program 5.3).

• When you cannot continue, move left and begin again or move back until you can move left and resume.

• The nodes of figure 5.16 would be output in postorder as

AB/C*D*E+

```
void preorder(tree_pointer ptr)
{          /* preorder tree traversal */
          if (ptr)
          {
                    printf("%d", ptr->data);
                    preorder(ptr->left_child);
                    predorder(ptr->right_child);
          }
}
```

Program 5.3: Postorder traversal of binary tress

ITERATIVE INORDER TRAVERSAL

• wrt figure 5.17, a node that has no action indicates that the node is added to the stack, while a node that has a printf action indicates that the node is removed from the stack (Program 5.4).
• The left nodes are stacked until a null node is reached, the node is then removed from the stack, and the node's right child is stacked.
• The traversal continues with the left child.
• The traversal is complete when the stack is empty.

```
void iter_inorder(tree_pointer node)
{
        int top= -1;      /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for (;;)
        {
                for (; node; node=node->left_child)
                add(&top, node);            /* add to stack */
                node= delete(&top);
                  /* delete from stack */
                if (!node) break;           /* empty stack */
                        printf("%D", node->data);
                node = node->right_child;
        }
}
```

Program 5.4: Iterative Inorder traversal of binary tress

LEVEL-ORDER TRAVERSAL

• This traversal uses a queue (Program 5.7).
• We visit the root first, then the root's left child followed by the root's right child.
• We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.

```
void level_order(tree_pointer ptr)
{       /* level order tree traversal */
        int front = rear = 0;
        tree_pointer queue[MAX_QUEUE_SIZE];
        if (!ptr) return;           /* empty queue */
        addq(front, &rear, ptr);
        for (;;)
        {
                ptr = deleteq(&front, rear);
                if (ptr)
                {
                     printf("%d", ptr->data);
                     if (ptr->left_child)
                          addq(front, &rear, ptr->left_child);
                     if (ptr->right_child)
                          addq(front, &rear, ptr->right_child);
                }
                else
                    break;
        }
}
```

Program 5.7: Level order traversal of binary tress

THREADED BINARY TREES

• Shortcoming of linked representation of binary tree: There may be more null links than actual pointers.

• Solution: This drawback can be overcome by replacing null links by pointers, called threads, to other nodes in the tree.

• To construct the threads, we use the following rules:

1) If ptr->leftChild =NULL, we replace null link with a pointer to the inorder predecessor of ptr (Figure 5.20).

2) If ptr->rightChild =NULL, we replace null link with a pointer to the inorder successor of ptr.

• The node structure is given by following C declarations:

```
typedef struct threadedtree  *threadedPointer;
typedef struct
{
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
}threadedTree;
```

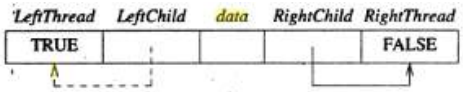


Figure 5.21: An empty threaded binary tree

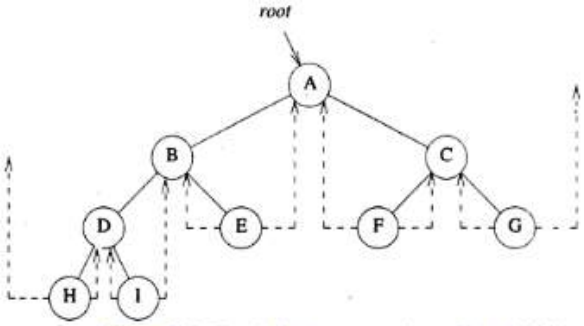


Figure 5.20: Threaded tree corresponding to Figure 5.10(b)

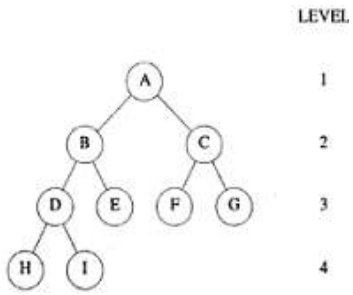


Figure 5.10: Skewed and complete binary trees

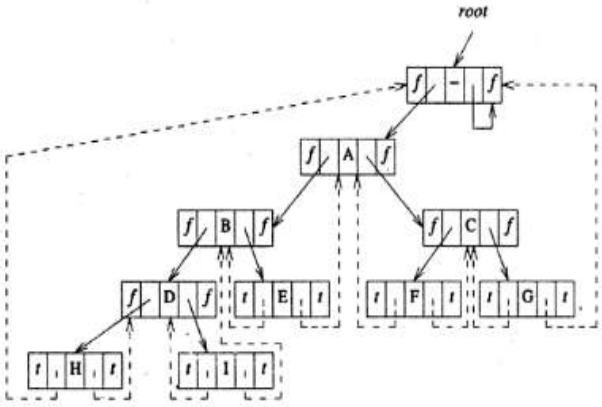


Figure 5.22: Memory representation of threaded tree

• When we represent the tree in memory, we must be able to distinguish between threads and normal pointers. This is done by adding two additional fields to the node structure, leftThread and rightThread (Figure 5.21).

• Assume that ptr is an arbitrary node in a threaded tree. If ptr->leftThread=TRUE, then ptr->leftChild contains a thread; otherwise it contains a pointer to the left child (Fig 5.23).

Similarly, if ptr->rightThread=TRUE, then ptr->rightChild contains a thread; otherwise it contains a pointer to the right child (Figure 5.22).

• We handle the problem of the loose threads by having them point to the header node, root.

• The variable 'root' points to the header.

INORDER TRAVERSAL OF A THREADED BINARY TREE

```
threaded_pointer insucc(threaded_pointer tree)
{
        threaded_pointer temp;
        temp = tree->right_child;
        if (!tree->right_thread)
                while (!temp->left_thread)
                        temp = temp->left_child;
        return temp;
}
```

Program 5.8: Finding inorder successor in threaded BT

```
void tinorder(threaded_pointer tree)
{
        /* traverse the threaded binary tree inorder */
        threaded_pointer temp = tree;
        for (;;)
        {
                temp = insucc(temp);
                if (temp==tree) break;
                    printf("%3c", temp->data);
        }
}
```

Program 5.9: Inorder traversal of threaded BT

INSERTING A NODE INTO A THREADED BINARY TREE

- Let new node 'r' be has to be inserted as the right child of a node 's' (Figure 5.23).
- The cases for insertion are

  1) If s has an empty right subtree, then the insertion is simple and diagrammed in fig 5.23a.

  2) If the right subtree of s is not empty, then this right subtree is made the right subtree of r after insertion. When this is done, r becomes the inorder predecessor of a node that has a leftThread==true field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s. Figure 5.23b illustrates the insertion for this case.

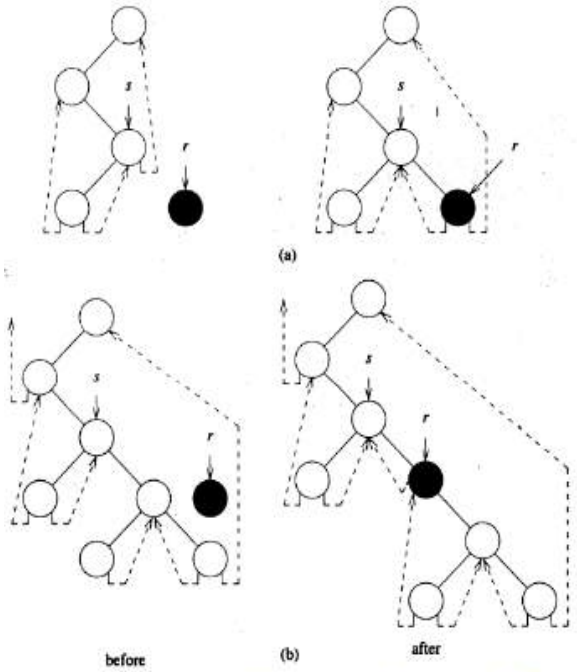


Figure 5.23: Insertion of *r* as a right child of *s* in a threaded binary tree

```
void insert_right(threaded_pointer s, threaded_pointer r)
{
        threaded_pointer temp;
        r->right_child = s->right_child;
        r->right_thread = s->right_thread;
        r->left_child = s;
        r->left_thread = TRUE;
        s->right_child = r;
        s->right_thread = FALSE;
        if (!r->right_thread)
        {
                temp = insucc(r);
                temp->left_child = r;
        }
}
```

Program 5.10: Inserting r as the right child of s

HEAPS (PRIORITY QUEUES)

- Heaps are frequently used to implement priority queues.
- In this kind of queue, the element to be deleted is the one with highest (or lowest) priority.
- At any time, an element with arbitrary priority can be inserted into the queue.

```
structure MaxHeap
  objects: a complete binary tree of n > 0 elements organized so that
the value in each node is at least as large as those in its children
  functions:
    for all heap ∈ MaxHeap, item ∈ Element, n, max_size ∈ integer
    MaxHeap Create(max_size)::= create an empty heap that can  hold a maximum of
                                  max_size elements
    Boolean HeapFull(heap, n)::= if (n==max_size)
                                    return TRUE
                                 else
                                    return FALSE
    MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap,n))
                                       insert  item into heap and return the resulting heap
                                     else
                                       return error
    Boolean HeapEmpty(heap, n)::= if (n>0)
                                    return FALSE
                                  else
                                    return TRUE
    Element Delete(heap,n)::= if (!HeapEmpty(heap,n))
                                 return one instance of the largest element in the heap and
                                 remove it from the heap
                              else
                                 return error
```

ADT 5.2: Abstract data type MaxHeap

MAX(MIN) HEAP

- A max tree is a tree in which key value in each node is larger than key values in its children (if any).
  A min tree is a tree in which key value in each node is smaller than key values in its children (if any).
- A max heap is a complete binary tree that is also a max tree (Figure 5.24).
  A min heap is a complete binary tree that is also a min tree (Figure 5.25).
- The key in the root of a max tree is the largest key in the tree,
  whereas key in the root of a min tree is the smallest key in the tree.

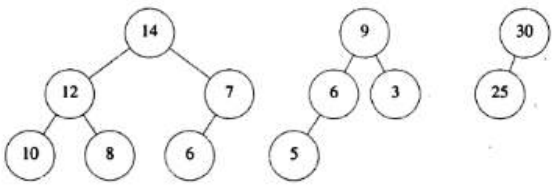


Figure 5.24: Max heaps

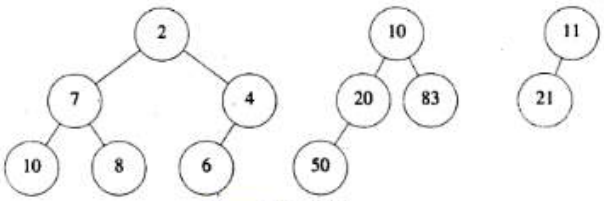


Figure 5.25: Min heaps

INSERTION INTO A MAX HEAP

• To determine the correct place for the element that is being inserted, we use a bubbling up process that begins at the new node of the tree and moves toward the root (Figure 5.26).

• The element to be inserted bubbles up as far as is necessary to ensure a max heap following the insertion.

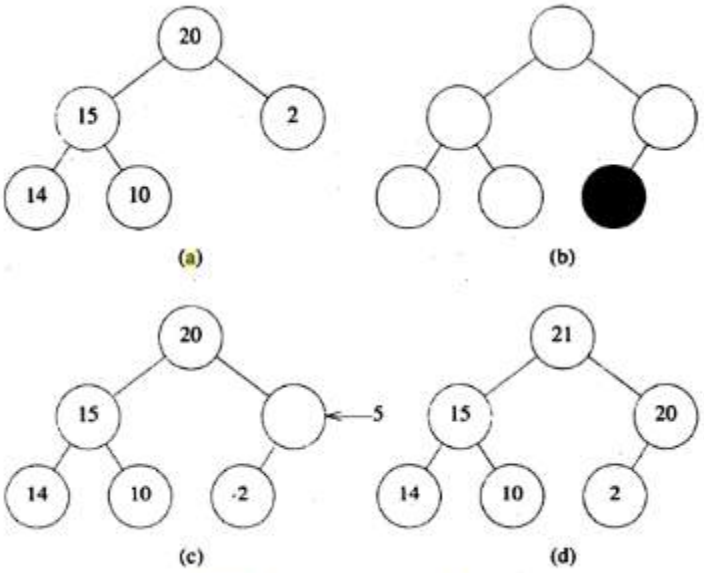


Figure 5.26: Insertion into a max heap

• The heap is created using the following C declarations

```
#define MAX_ELEMENTS 200                    /* maximum heap size+1*/
#define HEAP_FULL(n)  (n==MAX_ELEMENTS-1)
#define HEAP_EMPTY(n)   (!n)
typedef struct
{
        int key;
}element;
element heap[MAX_ELEMENTS];
int n=0;
```

```
void insert_max_heap(element item, int *n)
{
          int i;
          if (HEAP_FULL(*n))
          {
                    fprintf(stderr, "the heap is full.\n");
                    exit(1);
          }
          i = ++(*n);
          while ((i!=1)&&(item.key>heap[i/2].key))
          {
                    heap[i] = heap[i/2];
                    i /= 2;
          }
          heap[i]= item;
}
```

Program 5.11: Insertion into a Max Heap

DELETION FROM A MAX HEAP

• When an element is to be deleted from a max heap, it is taken from the root of the heap (Figure 5.27 & Program 5.12).

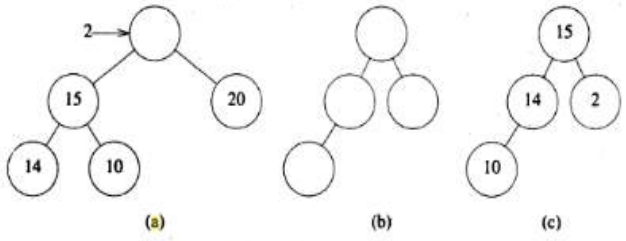


Figure 5.27: Deletion from a heap

```
element delete_max_heap(int *n)
{
        int parent, child;
        element item, temp;
        if (HEAP_EMPTY(*n))
        {
                fprintf(stderr, "The heap is empty\n");
                exit(1);
        }
        /* save value of the element with the highest key */
        item = heap[1];
        /* use last element in heap to adjust heap */
        temp = heap[(*n)--];
        parent = 1;
        child = 2;
        while (child <= *n)
        {
                /* find the larger child of the current parent */
                 if ((child < *n) &&  (heap[child].key<heap[child+1].key))
                        child++;
                if (temp.key >= heap[child].key)
                        break;
                /* move to the next lower level */
                heap[parent] = heap[child];
                child *= 2;
        }
        heap[parent] = temp;
        return item;
}
```

Program 5.12: Deletion from a Max Heap

UNIT 6: TREES (CONT.)

BINARY SEARCH TREE(BST)

• This is a tree that satisfies the following properties:

1) Each node has exactly one key and the keys in the tree are distinct (Figure 5.8).
2) The keys in the left subtree are smaller than the key in the root.
3) The keys in the right subtree are larger than the key in the root.
4) The left and right subtrees are also binary search trees.

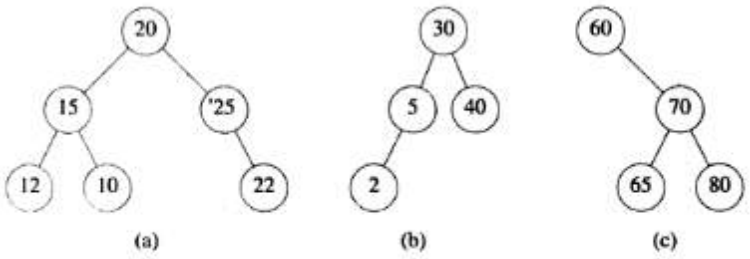


Figure 5.8: Binary trees

SEARCHING A BINARY SEARCH TREE

• Assume that we have to search for a node whose key value is k. The search begins at the root(Program 5.13)

1) If the root is NULL, then the tree contains no nodes and hence the search is unsuccessful.
2) If the key value k matches with the root's data then search terminates successfully.
3) If the key value is less than the root's data, then we should search in the left subtree.
4) If the key value is greater than the root's data, then we should search in the right subtree.

• Analysis: If h is the height of the binary search tree, then the search operation can be performed in O(h) time.

```
tree_pointer search(tree_pointer root, int key)
{
        /* return a pointer to the node that contains key. If there is no such
        node, return NULL */
        if (!root)
                return NULL;
        if (key == root->data)
                return root;
        if (key < root->data)
                return search(root->left_child,key);
        else
                search(root->right_child,key);
}
```

Program 5.13: Recursive searching a Binary Search Tree

```
tree_pointer search2(tree_pointer tree, int key)
{
        while (tree)
        {
                if (key == tree->data)
                        return tree;
                if (key < tree->data)
                        tree = tree->left_child;
                else
                        tree = tree->right_child;
        }
        return NULL;
}
```

Program 5.14: Iterative searching a Binary Search Tree

INSERTING INTO A BINARY SEARCH TREE

1) We must first verify if the tree already contains the node with the same data (Figure 5.29 & Program 5.15).
2) If the search is successful, then the new node cannot be inserted into the binary search tree.
3) If the search is unsuccessful, then we can insert the new node at that point where the search terminated.

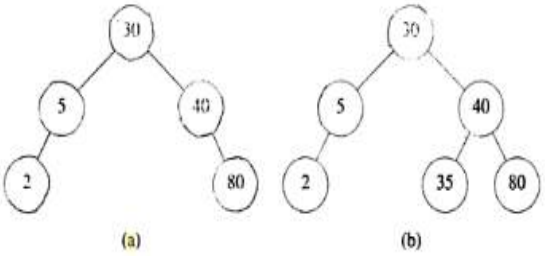


Figure 5.29: Inserting into a binary search tree

```
void insert_node(tree_pointer *node, int num)
{
        tree_pointer ptr,
        temp = modified_search(*node, num);
        if (temp || !(*node))
        {
                ptr = (tree_pointer) malloc(sizeof(node));
                if (IS_FULL(ptr))
                {
                        fprintf(stderr, "The memory is full\n");
                        exit(1);
                }
                ptr->data = num;
                ptr->left_child = ptr->right_child = NULL;
                if (*node)
                        if (num<temp->data)
                                temp->left_child=ptr;
                        else
                                temp->right_child = ptr;
                else
                        *node = ptr;
         }
}
```

Program 5.15: Insertion into a Binary Search Tree

DELETION FROM A BINARY SEARCH TREE

1) *Deletion of a leaf:* To delete 35 from the tree of figure 5.29b, the left-child field of its parent is set to NULL.
2) *Deletion of a non-leaf that has only one child:* The node containing the dictionary pair to be deleted is freed, and its single-child takes the place of the freed node. So, to delete the 5 from the tree in figure 5.29a, we simply change the pointer from the parent node to the single-child node.
3) *The pair to be deleted is in a non-leaf node that has two children*: The pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree. For instance, if we wish to delete the pair with key 30 from the tree in figure 5.29b, then we replace it by key 5 as shown in figure 5.30b.

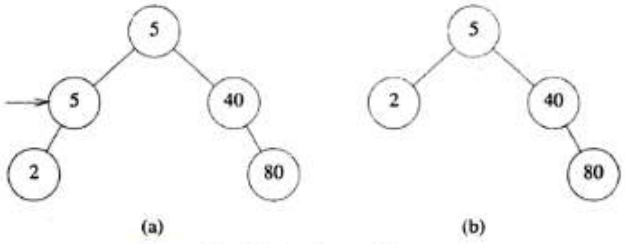


Figure 5.30: Deletion from a binary search tree

JOINING BINARY TREE

• There are two types of join operation on a binary search tree:

1) threeWayJoin(small,mid,big): We simply create a new node and set its data field to mid, its left-child pointer to small and its right-child pointer to big.

2) twoWayJoin(small,big):

i) If small is empty, then the result is big.

ii) If big is empty, then the result is small.

iii) If both are non-empty, then we have to first delete from 'small' the pair mid with the largest key.After this, a 3-way join of small, mid and big must be performed.

SPLITTING BINARY TREE

• Splitting a binary search tree will produce three trees: small, mid and big.

1) If key is equal to root->data, then root->llink is the small, root->data is mid & root->rlink is big.

2) If key is lesser than the root->data, then the root's along with its right subtree would be in the big.

3) if key is greater than root->data, then the root's along with its left subtree would be in the small.

SELECTION TREE

• This is also called as a tournament tree. This is such a tree data structure using which the winner (or loser) of a knock out tournament can be selected.

• There are two types of selection trees namely: winner tree and loser tree.

WINNER TREE

• This is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree (Figure 5.31 & 32).

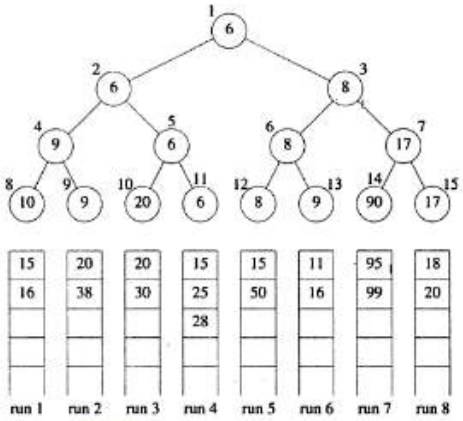


Figure 5.31: Winner tree for $k = 8$, showing the first three keys in each of the eight runs

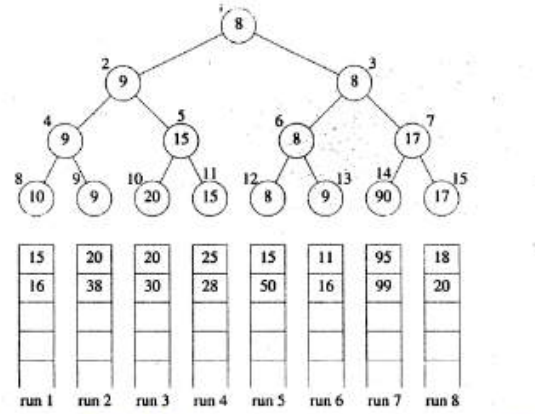


Figure 5.32: Winner tree of Figure 5.31 after one record has been output and the tree restructured (nodes that were changed are shaded)

• The construction of the winner tree may be compared to the playing of a tournament.

• The winner is a record with the smaller key.

• Each non-leaf node represents winner of a tournament, and root node represents overall winner (or smaller key).

LOSER TREES

• This is a selection tree in which each non-leaf node retains a pointer to the loser (Fig 5.33).

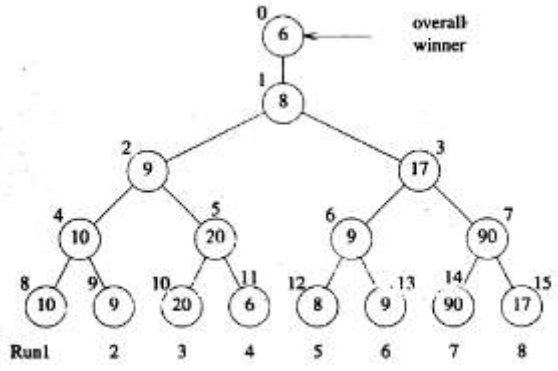


Figure 5.33: Loser tree corresponding to winner tree of Figure 5.31

FORESTS

• This is a set of n>=0 disjoint trees (Figure 5.34).

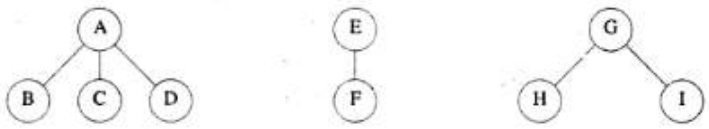


Figure 5.34: Three-tree forest

TRANSFORMING A FOREST INTO A BINARY TREE

• If T_1, T_2. . . .T_n is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1,T_2. . . . . T_n)$,

1) is empty if n=0
2) has root equal to root(T_1);has left subtree equal to $B(T_{11},T_{12}. . . . . .T_{1m})$
3) has right subtree $B(T_2. . . . . T_n)$

where $T_{11},T_{12}. . . . . .T_{1m}$ are the subtrees of root(T_1).

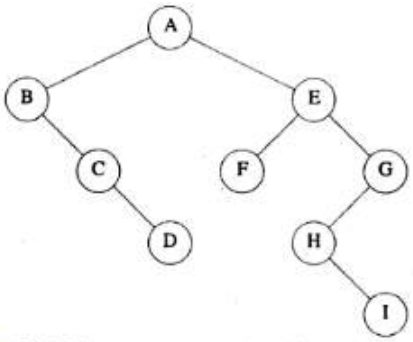


Figure 5.35: Binary tree representation of forest of Figure 5.34

FOREST TRAVERSALS

• There are 3 forest traversal techniques namely: preorder, inorder and postorder traversal.
• Preorder traversal of forest F can be recursively defined as follows
1) If F is empty then return.
2) Visit the root of the first tree of F.
3) Traverse the subtrees of the first tree in forest preorder.
4) Traverse the remaining trees of F in forest preorder.
• Inorder traversal of forest F can be recursively defined as follows
1) If F is empty then return.
2) Traverse the subtrees of the first tree in forest inorder.
3) Visit the root of the first tree of F.
4) Traverse the remaining trees of F in forest inorder.
• Postorder traversal of forest F can be recursively defined as follows
1) If F is empty then return.
2) Traverse the subtrees of the first tree in forest postorder.
3) Traverse the remaining trees of F in forest postorder.
4) Visit the root of the first tree of F.

REPRESENTATION OF DISJOINT SETS

- A set is a collection of elements.
- Disjoint sets are such sets in which common elements are not present.
- Consider the 3 sets,

  S_1={0,6,7,8} S_2={1,4,9} S_3={2,3,5}
- The above three sets are disjoint since none of the elements are common between the sets.

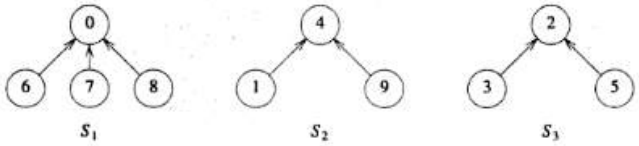


Figure 5.36: Possible tree representation of sets

- A set can be represented in two forms:
  1) Tree representation (linked list representation)
  2) Array representation
- The 3 sets S_1, S_2 and S_3 can be represented in the form of a tree as shown below

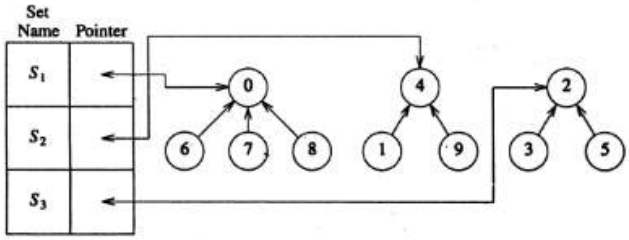


Figure 5.38: Data representation for S_1, S_2, and S_3

- The same 3 sets S_1, S_2 and S_3 can be represented in the form of array as shown below

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	−1	4	−1	2	−1	2	0	0	0	4

Figure 5.39: Array representation of S_1, S_2, and S_3 of Figure 5.36

OPERATIONS THAT CAN BE PERFORMED ON A SET

1) Disjoint set union: If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$={all elements, x such that x is in S_i or S_j}.

If S_1={0,6,7,8} S_2={1,4,9} then $S_1 \cup S_2$={0,6,7,8,1,4,9}

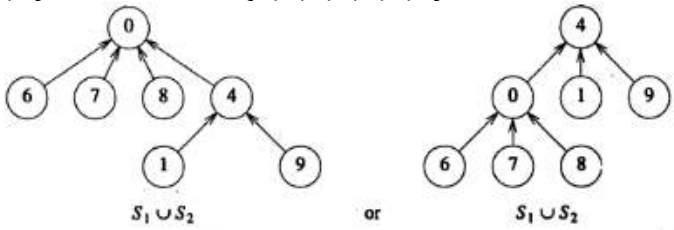


Figure 5.37: Possible representations of $S_1 \cup S_2$

2) Find(i):Find the set containing the element i. For example, 3 is in set S_3 and 8 in set S_1.

```
int find1(int i)
{
   for (; parent[i]>=0; i=parent[i]);
   return i;
}

void union1(int i, int j)
{
   parent[i]= j;
}
```

Program 5.16: iterative searching a Binary Search Tree

WEIGTING RULE FOR UNION(I,J)

• If the number of nodes in tree i is less than the number in tree j then make j the parent of i; otherwise make i the parent of j (Figure 5.41 & Program 5.17).

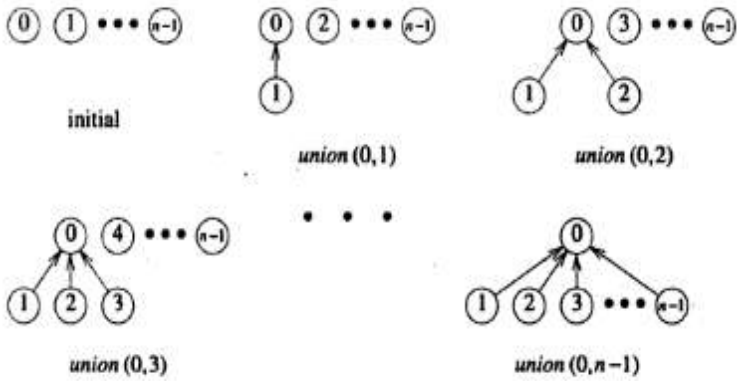


Figure 5.41: Trees obtained using the weighting rule

```
void union2(int i, int j)
{
        int temp = parent[i]+parent[j];
        if (parent[i]>parent[j])
        {
                parent[i]=j;
                parent[j]=temp;
        }
        else
        {
                parent[j]=i;
                 parent[i]=temp;
        }
}
```

Program 5.17: Union Operation with weighted rule

COUNTING BINARY TREES
STACK PERMUTATIONS

• Suppose we have the preorder sequence A B C D E F G H and the inorder sequence B C A E D G H F I of the same binary tree (Figure 5.47 & 48).

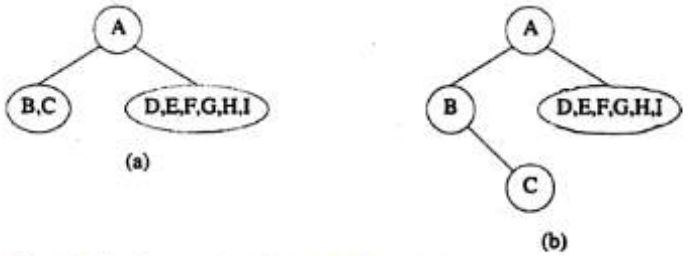


Figure 5.47: Constructing a binary tree from its inorder and preorder sequences

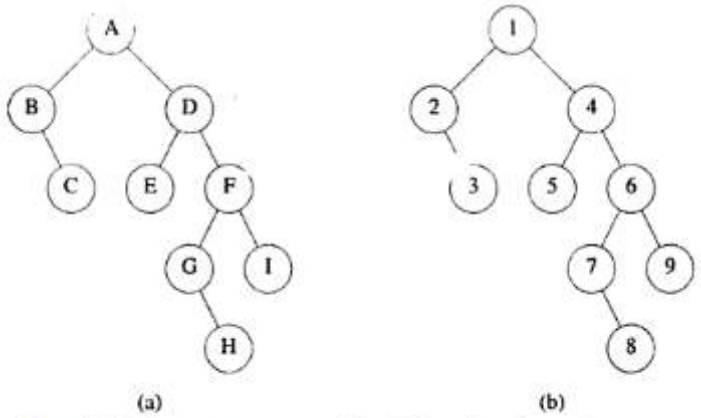


Figure 5.48: Binary tree constructed from its inorder and preorder sequences

• The number of distinct binary trees is equal to the number of distinct inorder permutations obtainable from binary trees having the preorder permutation 1, 2 . . . . . n. (Figure 5.49).
• If we start with the numbers 1,2 and 3,then the possible permutations obtainable by a stack are
(1,2,3) (1,3,2) (2,1,3) (2,3,1) (3,2,1)

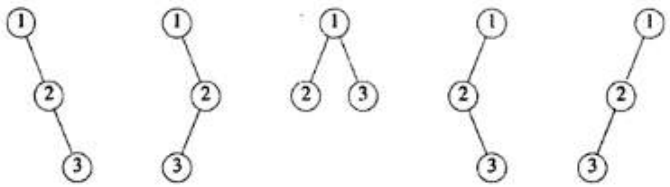


Figure 5.49: Binary trees corresponding to five permutations

NUMBER OF DISTINCT BINARY TREES

To obtain the number of distinct binary trees with n nodes, we must solve the recurrence of Eq. (5.3). To begin we let

$$B(x) = \sum_{i \geq 0} b_i x^i \tag{5.4}$$

which is the generating function for the number of binary trees. Next observe that by the recurrence relation we get the identity

$$xB^2(x) = B(x) - 1$$

Using the formula to solve quadratics and the fact that $B(0) = b_0 = 1$ (Eq.(5.3)), we get

$$B(x) = \frac{1 - \sqrt{1-4x}}{2x}$$

We can use the binomial theorem to expand $(1 - 4x)^{1/2}$ to obtain

$$B(x) = \frac{1}{2x}\left[1 - \sum_{n \geq 0} \binom{1/2}{n} (-4x)^n\right] = \sum_{m \geq 0} \binom{1/2}{m+1} (-1)^m 2^{2m+1} x^m \tag{5.5}$$

Comparing Eqs. (5.4) and (5.5), we see that b_n, which is the coefficient of x^n in $B(x)$, is

$$\binom{1/2}{n+1} (-1)^n 2^{2n+1}$$

Some simplification yields the more compact form

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

which is approximately

$$b_n = O(4^n/n^{3/2})$$

UNIT 6(CONT.): GRAPHS

GRAPH

• A graph G consists of 2 sets, V and E.

V is a finite, on empty set of vertices.

E is a set of pairs of vertices, these pairs are called edges.

V(G) and E(G) represents the set of vertices and edges respectively of graph G (Fig 6.2).

• In an **undirected graph**, the pair of vertices representing any edge is unordered. Thus, the pairs (u,v) and (v,u) represent the same edge.

• In a **directed graph**, each edge is represented by a directed pair <u,v>; u is the tail and v is the head of the edge. Therefore, <u,v> and <v,u> represent two different edges.

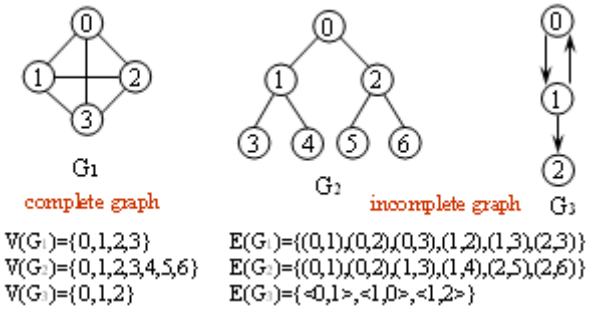


Figure 6.2: Three sample graphs

Following are the restrictions on graphs

1) A graph may not have an edge from a vertex v back to itself. Such edges are known as **self loops** (Figure 6.3).

2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as **multigraph**.

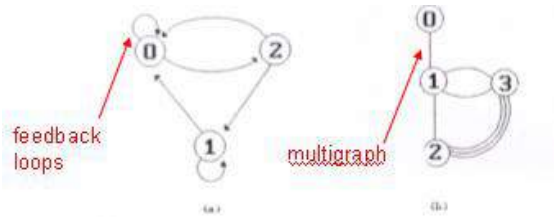


Figure 6.3: Examples of graph like structures

• Maximum number of edges in any n-vertex, undirected graph is n(n-1)/2.

Maximum number of edges in any n-vertex, directed graph is n(n-1).

TERMINOLOGIES USED IN A GRAPH

• **Subgraph** of G is a graph G' such that V(G') ∈ V(G) and E(G') ∈ E(G) (Fig 6.4).

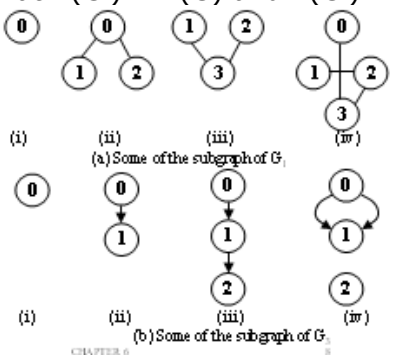


Figure 6.4:Some subgraphs

• A **path** from vertex u to vertex v in graph G is a sequence of vertices $u,i_1,i_2 \ldots i_k$, v such that $(u,i_1),(i_1,i_2) \ldots \ldots .(i_k, v)$ are edges in E(G).

• A **simple path** is a path in which all vertices except possibly the first and last are distinct.

• A **cycle** is a simple path in which the first and last vertices are the same.

• A undirected graph is said to be **connected** iff

for every pair of distinct vertices u & v in V(G) there is a path from u to v in G.

• A **connected component H** of an undirected graph is a maximal connected subgraph (Figure 6.5).

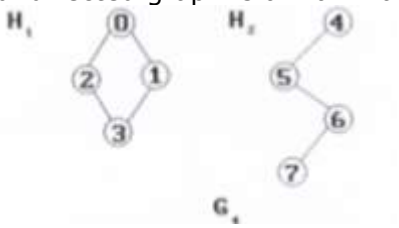


Figure 6.5:A graph with two connected components

• A **tree** is a connected acyclic(i.e. has no cycles) graph.

• A directed graph G is said to be **strongly connected** iff for every pair of distinct vertices u and v in V(G),there is a directed path from u to v and also from v to u (Figure 6.6).

Figure 6.6:Strongly connected components of G3

• The **degree** of a vertex is the number of edges incident to that vertex. (Degree of vertex 0 is 3)

• In a directed graph G, **in-degree** of a vertex v defined as the number of edges for which v is the head. The **out-degree** is defined as the number of edges for which v is the tail. (Vertex 1 of G3 has in-degree 1, out-degree 2 and degree 3).

GRAPH ABSTRACT DATA TYPE

```
structure Graph is
  objects: a nonempty set of vertices and a set of undirected edges, where each edge is a
pair of vertices
  functions: for all graph ∈ Graph, v, v1 and v2 ∈ Vertices
    Graph Create()::=return an empty graph
    Graph InsertVertex(graph, v)::= return a graph with v inserted. v has no incident edge.
    Graph InsertEdge(graph, v1,v2)::= return a graph with new edge between v1 and v2
    Graph DeleteVertex(graph, v)::= return a graph in which v and all edges incident to it
                                    are removed
    Graph DeleteEdge(graph, v1, v2)::=return a graph in which the edge (v1, v2) is
                                    removed
    Boolean IsEmpty(graph)::= if (graph==empty graph)
                                    return TRUE
                                else
                                    return FALSE
    List Adjacent(graph,v)::= return a list of all vertices that are adjacent to v
```

ADT 6.1:Abstract data type Graph

GRAPH REPRESENTATIONS

• Three commonly used representations are:

1) Adjacency matrices,
2) Adjacency lists and
3) Adjacency multilists

Adjacency Matrix

• Let G=(V,E) be a graph with n vertices, n>=1.

• The adjacency matrix of G is a two-dimensional n*n array(say a) with the property that a[i][j]=1 iff the edge (i,j) is in E(G). a[i][j]=0 if there is no such edge in G (Figure 6.7).

• The space needed to represent a graph using its adjacency matrix is n^2 bits.

• About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.

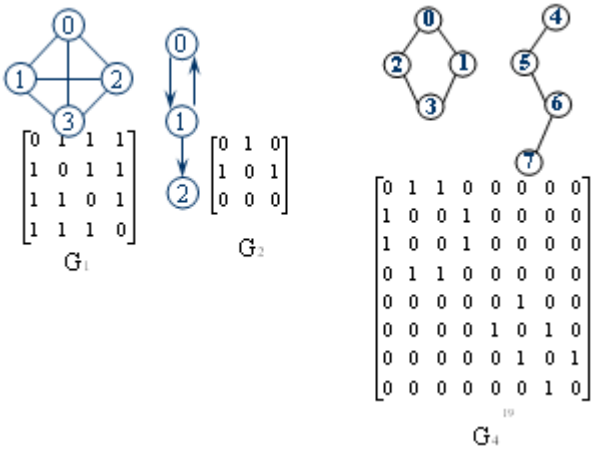


Figure 6.7 Adjacency matrices

Adjacency Lists

• The n rows of the adjacency matrix are represented as n chains.

• There is one chain for each vertex in G.

• The data field of a chain node stores the index of an adjacent vertex (Figure 6.8).

• For an undirected graph with n vertices and e edges, this representation requires an array of size n and 2e chain nodes.

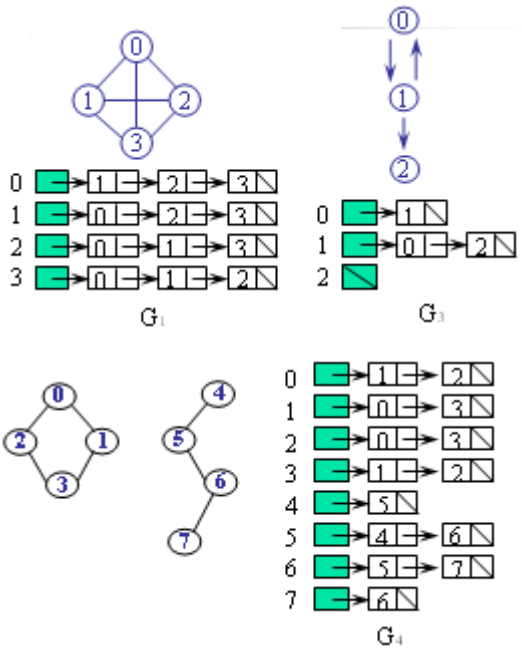
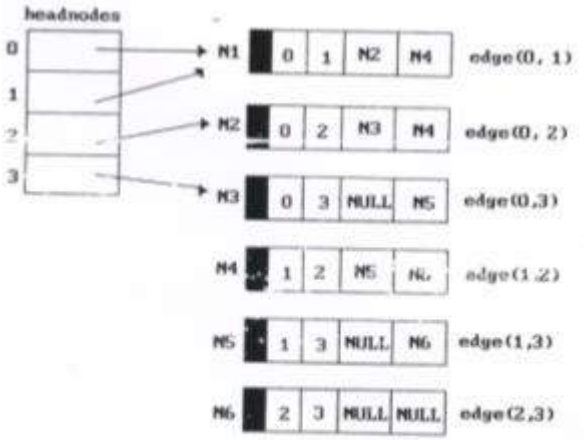


Figure 6.8 adjacency lists

Adjacency Multilists

- Each edge (u,v) is represented by two entries, one on the list for u and the other on the list for v.
- The new node structure is



The lists are: vertex 0: M1 → M2 → M3
vertex 1: M1 → M4 → M5
vertex 2: M2 → M4 → M6
vertex 3: M3 → M5 → M6

Figure 6.12:Adjacency multilists for G1