PRATHYUSHA

ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3301 - DATASTRUCTURES

(REGULATION R2021 – III SEMESTER)

CS3301 DATA STRUCTURES L T P C 3 0 0 3

OBJECTIVES:

- To understand the concepts of ADTs
- To Learn linear data structures – lists, stacks, and queues
- To understand non-linear data structures – trees and graphs.
- To understand sorting, searching and hashing algorithms
- To apply Tree and Graph structures

UNIT I LISTS 9

Abstract Data Types (ADTs) – List ADT – Array-based implementation – Linked list implementation – Singly linked lists – Circularly linked lists – Doubly-linked lists – Applications of lists – Polynomial ADT – Radix Sort – Multilists.

UNIT II STACKS AND QUEUES 9

Stack ADT – Operations – Applications – Balancing Symbols – Evaluating arithmetic expressions- Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Circular Queue – DeQueue – Applications of Queues.

UNIT III TREES 9

Tree ADT – Tree Traversals - Binary Tree ADT – Expression trees – Binary Search Tree ADT – AVL Trees – Priority Queue (Heaps) – Binary Heap.

UNIT IV MULTIWAY SEARCH TREES AND GRAPHS 9

B-Tree – B+ Tree – Graph Definition – Representation of Graphs – Types of Graph - Breadth-first traversal – Depth-first traversal — Bi-connectivity – Euler circuits – Topological Sort – Dijkstra's algorithm – Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES 9

Searching – Linear Search – Binary Search. Sorting – Bubble sort – Selection sort – Insertion sort – Shell sort –. Merge Sort – Hashing – Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

TOTAL: 45 PERIODS

OUTCOMES:

At the end of the course, the student should be able to:

- Define linear and non-linear data structures.
- Implement linear and non–linear data structure operations.
- Use appropriate linear/non–linear data structure operations for solving a given problem.
- Apply appropriate graph algorithms for graph applications.
- Analyze the various searching and sorting algorithms.

TEXT BOOKS:

1. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, 2nd Edition, Pearson Education,1997.
2. Kamthane, Introduction to Data Structures in C, 1st Edition, Pearson Education, 2007

REFERENCES:

1. Langsam, Augenstein and Tanenbaum, Data Structures Using C and C++, 2nd Edition, Pearson Education, 2015.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein, “Introduction to Algorithms”, Second Edition, Mcgraw Hill, 2002.
3. Alfred V. Aho, Jeffrey D. Ullman,John E. Hopcroft ,Data Structures and Algorithms, 1st edition, Pearson, 2002.
4. Kruse, Data Structures and Program Design in C, 2nd Edition, Pearson Education, 2006.

UNIT I LISTS 9

Abstract Data Types (ADTs) – List ADT – Array-based implementation – Linked list implementation – Singly linked lists – Circularly linked lists – Doubly-linked lists – Applications of lists – Polynomial ADT – Radix Sort – Multilists.

Abstract Data Types

An abstract data type (ADT) is a set of operations. Abstract data types are mathematical abstractions; ADT's defines how the set of operations is implemented. This can be viewed as an extension of modular design.

Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, reals, and booleans are data types.

Use of ADT

Reusability of the code, that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function.

The List ADT

A general list of the form a1, a2, a3, . . . , an. We say that the size of this list is n. We will call the special list of size 0 a null list. For any list except the null list, we say that a_{i+l} follows (or succeeds) ai (i < n) and that ai-1 precedes ai (i > 1).

The first element of the list is $\mathbf{a}_1$, and the last element is $\mathbf{a_n}$. there is no predecessor of a1 or the successor of a_n. The position of element ai in a list is i.

Some operations in list are,

1. Find, which returns the position of the first occurrence of a key;

2. Insert and delete, which generally insert and delete some key from some position in the list;

3. Find_kth, which returns the element in some position (specified as an argument).

Example:

1. If the list is 34, 12, 52, 16, 12, then find(52) might return 3;

2. Insert(x,3) might make the list into 34, 12, 52, x, 16, 12 (if we insert after the position given);
3. Delete (3) might turn that list into 34, 12, x, 16, 12.

IMPLEMENTATION OF THE LIST

1. Array implementation
2. Linked list implementation

Simple Array Implementation of Lists

Normally array is static allocation which causes more wastage of memory.

Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, especially if there are many lists of unknown size.

LIST ADT

List is an ordered set of elements.

The general form of the list is

$A_1, A_2, A_3, \ldots, A_N$

A_1 - First element of the list

A_N - Last element of the list

N - Size of the list

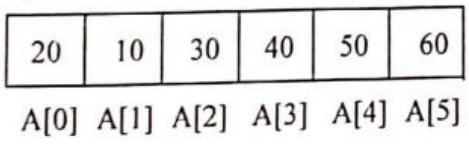
If the element at position i is A_i then its successor is A_{i+1} and its predecessor is A_{i-1}.

Various operations performed on List

1. Insert (X, 5) - Insert the element X after the position 5.
2. Delete (X) - The element X is deleted
3. Find (X) - Returns the position of X.
4. Next (i) - Returns the position of its successor element i+1.
5. Previous (i) - Returns the position of its predecessor i-1.
6. Print list - Contents of the list is displayed.
7. Makeempty - Makes the list empty.

ARRAY BASED IMPLEMENTATION OF LIST

Array is a collection of specific number of data stored in a consecutive memory locations.



Operations on Array

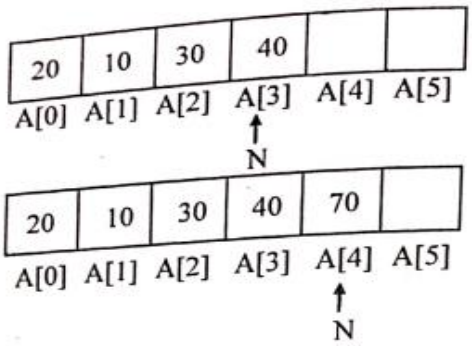
- Insertion
- Deletion
- Merge
- Traversal
- Find

3.2

Insertion Operation on Array :

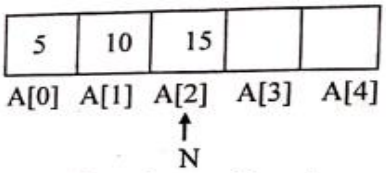
Insertion is the process of adding an element into the existing array. It can be done at any position. Inserting an element at the end is easy as it is done by adding one position towards right of last element if it does not exceeds the array size.

Example :



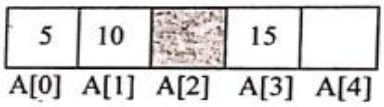
If an element is to be inserted at the specified position, then it will require all the subsequent elements to be shifted one position to the right.

Example :

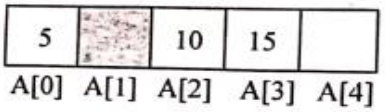


Insert (18, 1, A) // Insert an element 18 at the position 1

First shift last element (15) one position right (location 2 to 3)



Shift (10) one position right (from location 1 to 2)



Now insert (18) at position 1 and update the array index N as N+1

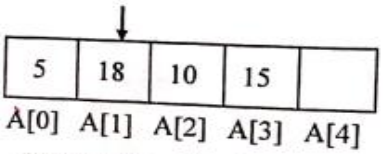


Fig. 3.2 Insertion in List

Fig. 3.2 Insertion in List

Routine to Insert an element in the Array

```
void insert (int X, int P, int A[ ], int N)      // X is an element, P is a position, N is
{                                                //number of elements in the array A,
        if (P = = N)
```

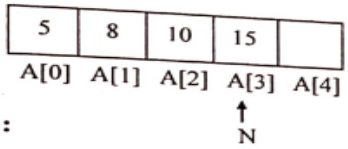


```
        printf ("Array Overflow")
else
{
for (int i = N-1; i > = P; i - - )
        A[i + 1] = A[i];
A[P] = X ;
N = N + 1;
}
```

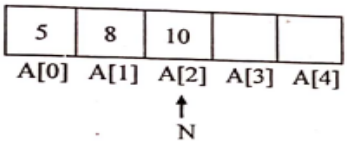
Deletion Operation on Array

Deletion is the process of removing an element from the array at any position. Deleting an element from the end is easy since after removing an element, array size alone gets decremented by one.

Example :

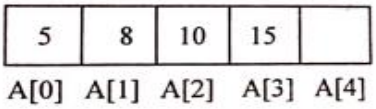


After Deleting **Last element 15 :**



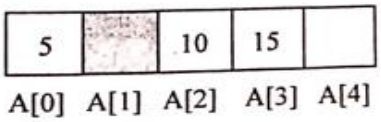
N

If an element is to be deleted from any particular position other than end, then it will require all the subsequent element from that position is shifted one position towards left.

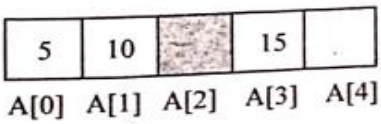


Delete an element from the position 1 :

Remove an element (8) from position 1



Shift the element (10) from location 2 to 1



Shift the element (15) from location 3 to 2

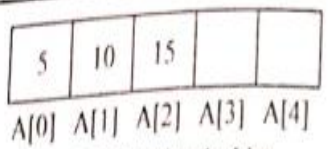


Fig. 3.3 Deletion in List

Finally update the Array Index N as N-1

Merge

Merging is the process of combining two sorted array into single sorted array.

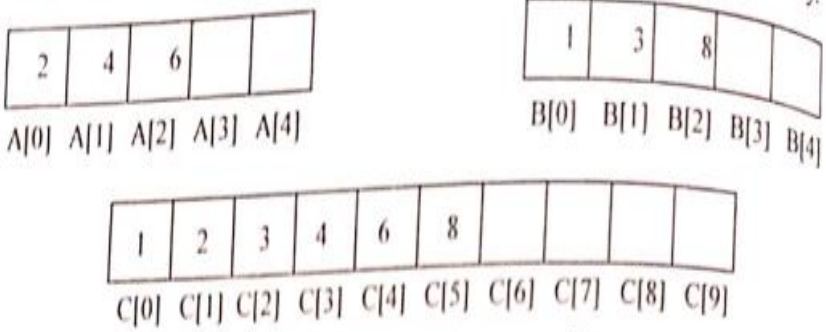


Fig. 3.3 Merging of two sorted list

Routine to delete an element from an array

```
int deletion (int P, int A[ ], int N)
{
    if (P = = N - 1)
        temp = A[P];    // temp is the element to be deleted
    else
    {
        temp = A[P];
        for (i = P; i<N-1; i ++)
            A[i] = A[i + 1]; // movement towards left.
    }
    N = N-1;
    return temp;
}
```

Routine to Merge two sorted array

```
void merge (int a[ ]; int N, int b[ ], int m)
{
        int c[n+m];
        int i = j = k = 0;
```



```
while (i < n && j < m)
{
        if (a[i] < b[j])
        {
                c[k] = a[i];
                i ++;
                k++;
        }
        else
        {
                c[k] = b[j];
                j ++;
                k++;
        }
}
while (i < n)
{
        c[k] = a[i];
        i++;
        k++;
}
```

Find Operation

Find is the process of searching an element in the given array. If the element is found, it returns the position of the search element otherwise NULL.

3.6

```
int find (int x, array a[ ], int N)
{
    int flag = 0;
    for (int i = 0; i < N; i++)
    {
        if (x = = a[i])
        {
            flag = 1;
            pos = i;
            break ;
        }
    }
    if (flag = = 1)
        print ("Element found at position pos")
    else
        printf ("element not found");
    return pos;
}
```

Other limitations are,

- Printing the list element and find to be carried out in linear time, which is as good as can be expected, and the find_kth operation takes constant time.
- Insertion and deletion are expensive. Because the running time for insertions and deletions is so slow and the list size must be known in advance.

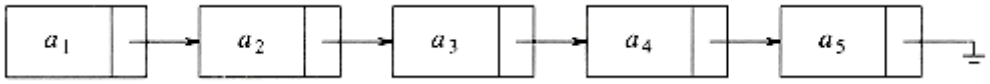
Linked Lists Implementation

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved.

Definition:

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the next pointer. The last cell's next pointer is always NULL.

Structure of linked list

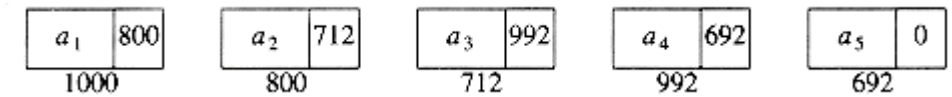


P is declared to be a pointer to a structure, then the value stored in p is interpreted as the location, in main memory, where a structure can be found.

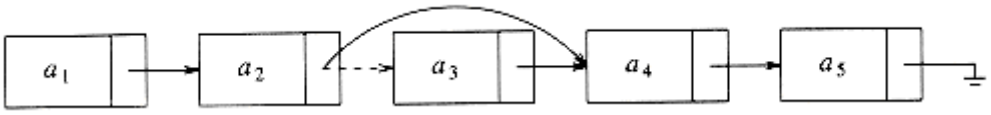
A field of that structure can be accessed by p->field_name, where field_name is the name of the field.

Consider the list contains five structures, which happen to reside in memory locations 1000, 800, 712, 992, and 692 respectively. The next pointer in the first structure has the value 800, which provides the indication of where the second structure is located.

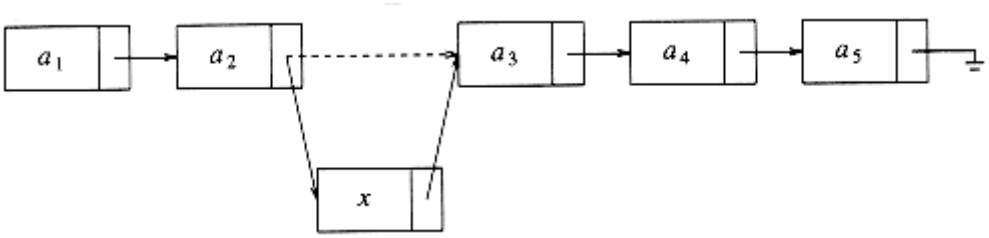
Actual representation of linked list with value and pointer



Deletion from a linked list



Insertion into a linked list



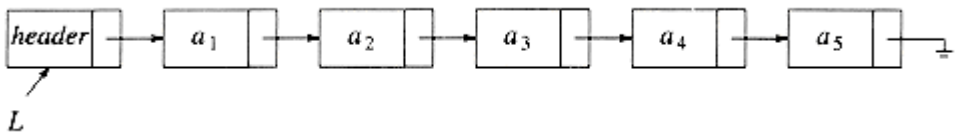
- The delete command can be executed in one pointer change. Above diagram shows the result of deleting the third element in the original list.
- The insert command requires obtaining a new cell from the system by using an malloc call function and then changing two pointer.

Programming Details

- First,It is difficult to insert at the front of the list from the list given.
- Second, deleting from the front of the list is a special case, because it changes the start of the list;
- A third problem concerns deletion in general. Although the pointer moves above are simple, the deletion algorithm requires us to keep track of the cell before the one that we want to delete.

In order to solve all three problems, we will keep a **sentinel node**, which is called as a **header or dummy node. (a header node contains the address of the first node in the linked list)**

Linked list with a header



The above figure shows a linked list with a header representing the list a1, a2, . . . , a5.

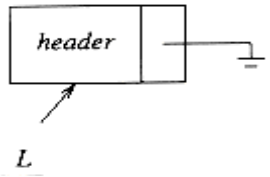
To avoid the problems associated with deletions, we need to write a routine find_previous, which will return the position of the predecessor of the cell we wish to delete.

If we use a header, then if we wish to delete the first element in the list, find_previous will return the position of the header.

Type declarations for linked lists

```
typedef struct node *node_ptr;
struct node
{
element_type element;
node_ptr next;          };
typedef node_ptr LIST;
typedef node_ptr position;
```

Empty list with header



Function to test whether a linked list is empty

```
int is_empty( LIST L )
        {
        return( L->next == NULL );          }
```

Function to test whether current position is the last in a linked list

```
int is_last( position p, LIST L )
{
return( p->next == NULL );
}
```

Function to find the element in the list

```
/* Return position of x in L; NULL if not found */
Position find ( element_type x, LIST L )
{
position p;
p = L->next;
while( (p != NULL) && (p->element != x) )
p = p->next;
return p;         }
```

Function to delete an element in the list

This routine will delete some element x in list L. We need to decide what to do if x occurs more than once or not at all. Our routine deletes the first occurrence of x and does nothing if x is not in the list. First we find p, which is the cell prior to the one containing x, via a call to find_previous.

```
/* Delete from a list. Cell pointed  to by p->next is wiped out. */
/* Assume that the position is legal. Assume use of a header node. */
    Void delete( element_type x, LIST L )
    {       position p, tmp_cell;
    p = find_previous( x, L );
    if( p->next != NULL ) /* Implicit assumption of header use */
    { /* x is found: delete it */
    tmp_cell = p->next;
    p->next = tmp_cell->next; /* bypass the cell to be deleted */
    free( tmp_cell ); } }
```

Function to find previous position of an element in the list

The find_previous routine is similar to find.

```
/* Uses a header. If element is not found, then next field of returned value is NULL */
    Position find_previous( element_type x, LIST L )
    {
    position p;
    p = L;
    while( (p->next != NULL) && (p->next->element != x) )
    p = p->next;
    return p;
    }
```

Function to insert an element in the list

Insertion routine will insert an element after the position implied by p. It is quite possible to insert the new element into position p which means before the element currently in position p.

```
/* Insert (after legal position p). Header implementation assumed. */
        Void insert ( element_type x, LIST L, position p )
        {
        position tmp_cell;
        tmp_cell = (position) malloc( sizeof (struct node) );
        if( tmp_cell == NULL )
        fatal_error("Out of space!!!");
        else
        {
        tmp_cell->element = x;
        tmp_cell->next = p->next;
        p->next = tmp_cell;
        } }
```

Function to delete the list

```
/* Incorrect way to delete a list*/
        delete_list( LIST L )
        {
        position p;
        p = L->next; /* header assumed */
        L->next = NULL;
        while( p != NULL )
        {
        free( p );
        p = p->next;
        }       }
```

Function to delete the list

```
/* correct way to delete a list*/
        Void delete_list( LIST L )
        {
        position p, tmp;
        p = L->next; /* header assumed */
```

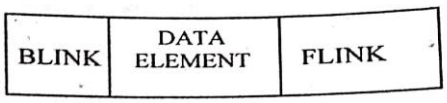
```
L->next = NULL;
while( p != NULL )
{
tmp = p->next;
free( p );
p = tmp;
        }       }
```

Doubly Linked Lists

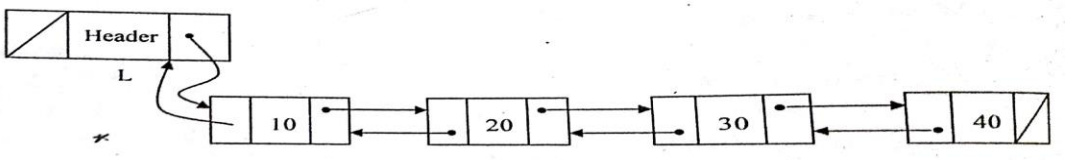
A linked list is called as doubly when it has two pointers namely forward and backward pointers. It is convenient to traverse lists both forward and backwards.

An extra field in the data structure, containing a pointer to the previous cell; The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix.

Node



A doubly linked list



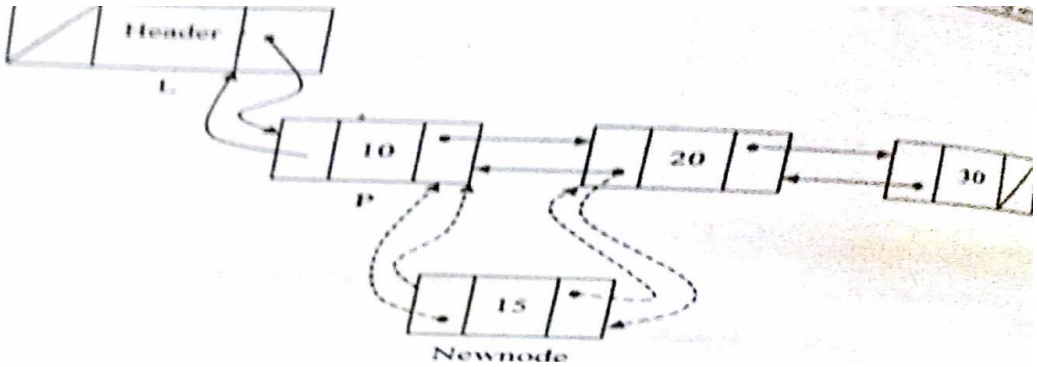
Structure declaration

```
struct node
{
int Element;
struct node *FLINK;
struct node *BLINK;
}
```

Insertion

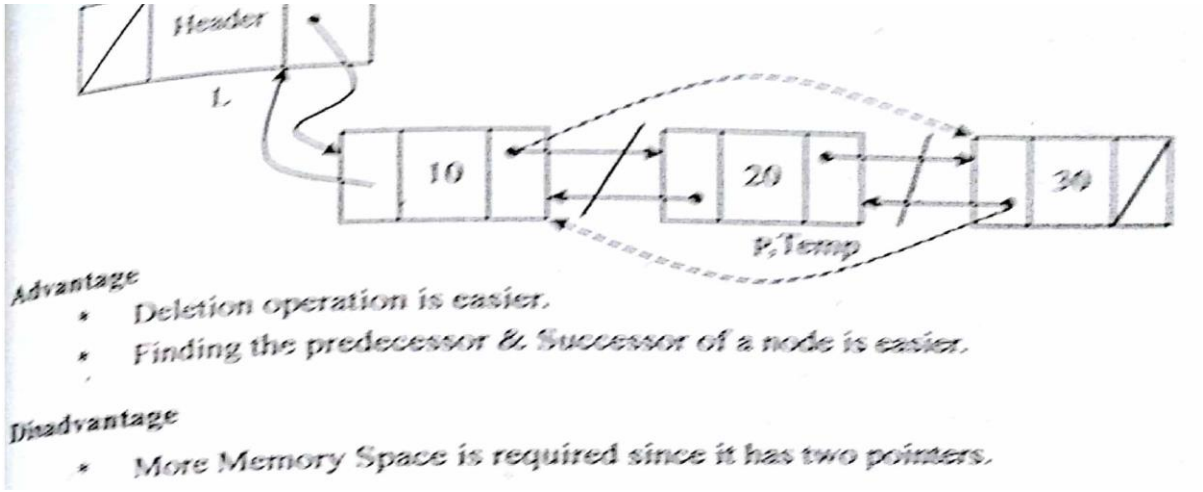
```
void Insert (int X, List L, Position P)
{
        struct    node * Newnode;
        Newnode = malloc (sizeof (struct node));
        if (Newnode ! = NULL)
        {
                Newnode → Element = X;
                Newnode → Flink = P → Flink;
                P → Flink →Blink = Newnode;
                P → Flink = Newnode ;
                Newnode →Blink = P;
        }
}
```

Insert(15,L,P)



Deletion:

```
void Deletion (int X, List L)
{
        Position P;
        P = Find (X, L);
        if ( IsLast (P, L))
        {
                Temp = P;
                P → Blink →Flink = NULL;
                free (Temp);
        }
        else
        {
                Temp = P;
                P → Blink →Flink = P →Flink;
                P →Flink →Blink = P →Blink;
                free (Temp);
        }
}
```

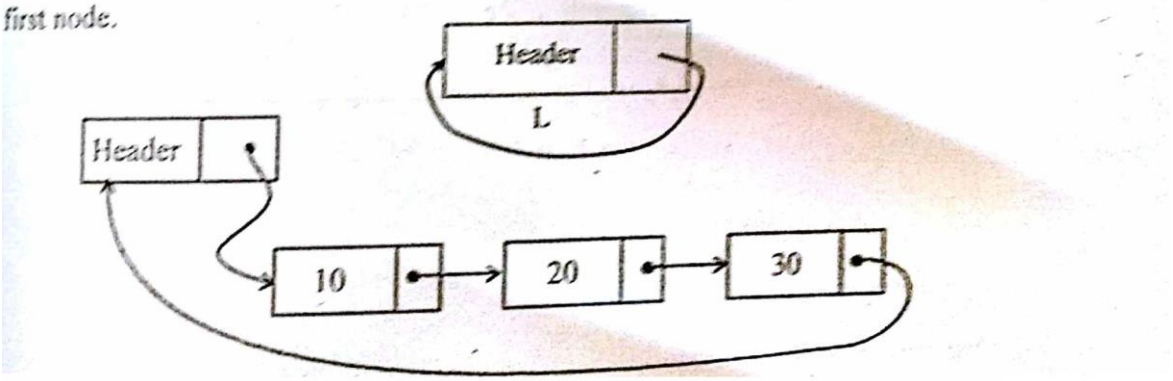


Circularly Linked Lists

A linked list is called as circular when its last pointer point to the first cell in the linked list forms a circular fashion. It can be **singly circular and doubly circular with header or without header.**

Singly Circular linked list:

A singly linked circular list is a linked list in which the last node of the list points to the first node.



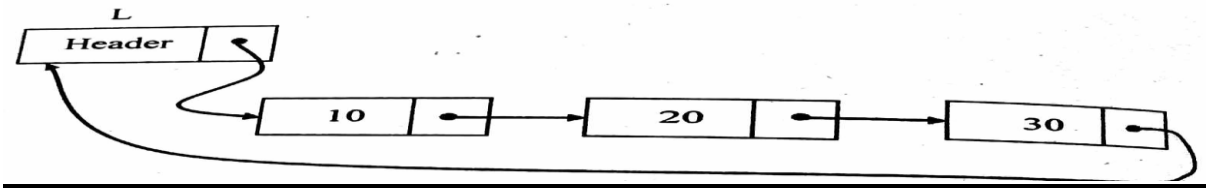
Structure declaration:

```
struct node
{
int Element;
struct node *Next;          }
```

Insert at beginning:

void Insert_beg(int X, List L)

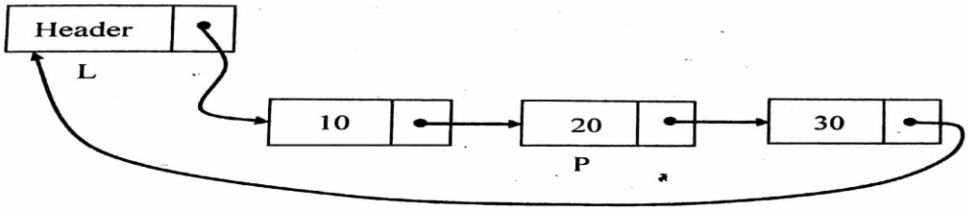
```
{
        struct node *Newnode;
        Newnode = malloc (sizeof (struct node));
        if (Newnode! = NULL)
        {
                Newnode → Element = X;
                Newnode → Next = L → Next ;
                L → Next = Newnode;
    }
```



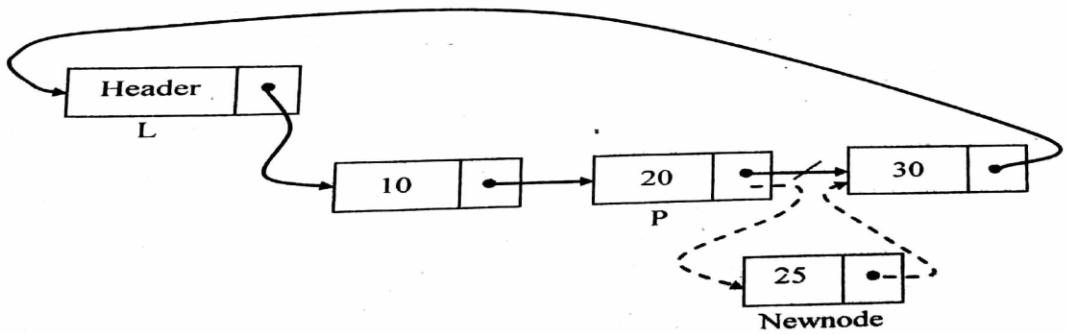
Insert in middle:

void insert_mid(int X, List L, Position P)

```
{
        struct node   *Newnode;
        Newnode = malloc(sizeof (struct node));
        if (Newnode ! = NULL)
        {
                Newnode → Element = X;
                Newnode → Next = P → Next;
                P → Next = Newnode;
        }
}
```

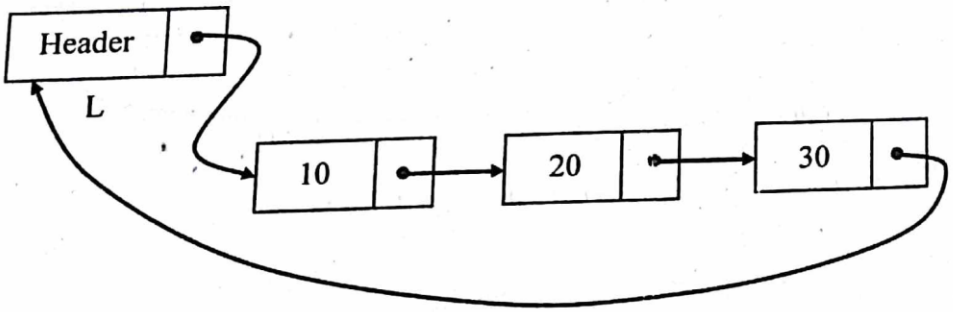


rt (25, L, P)

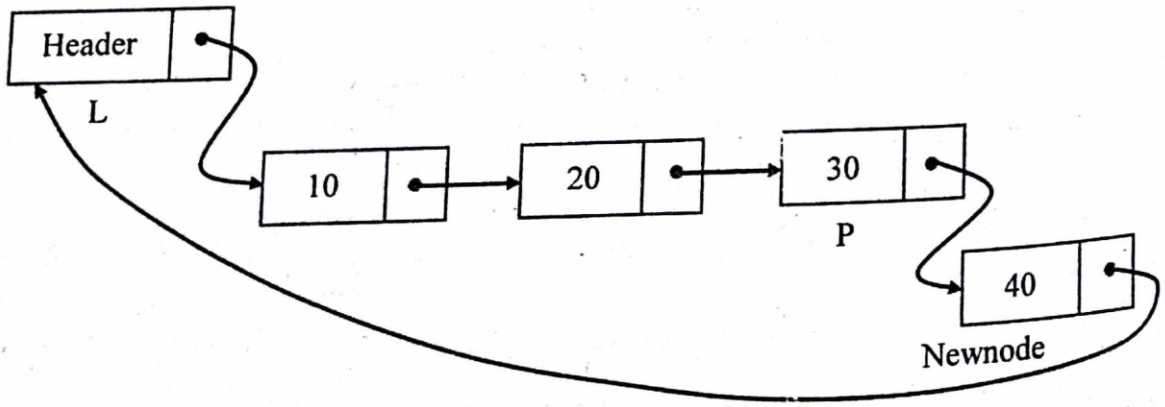


Insert at Last

```
void Insert_last (int X, List L)
{
        struct node * Newnode;
        Newnode = malloc(sizeof (struct node));
        if (Newnode ! = NULL)
        {
                P = L;
                while (P → Next ! = L)
                        P = P → Next ;
                Newnode → Element = X;
                P → Next = Newnode;
                Newnode → Next = L;
        }
}
```

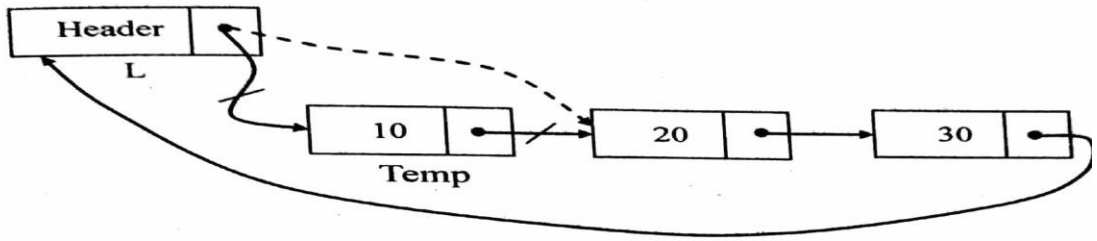


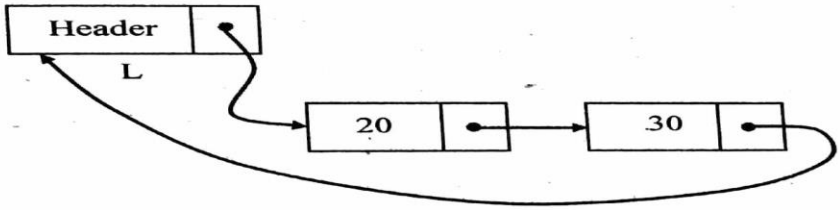
t (40, L)



Deletion at first node:

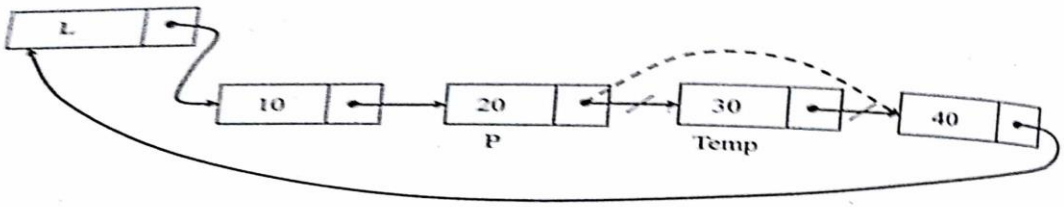
```
void dele_First (List L)
{
        position Temp;
        Temp = L → Next;
        L → Next = Temp → Next;
        free (Temp);
}
```



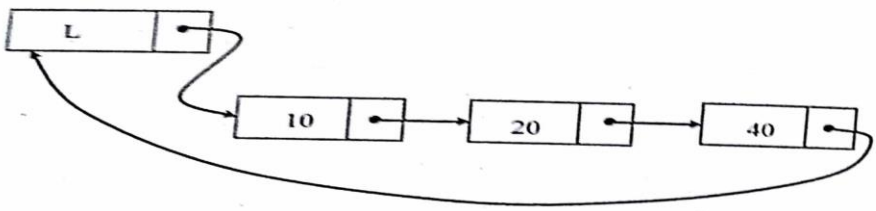


Deletion at middle

```
void dele_mid (int X, List L)
{
        Position P, Temp;
        P = FindPrevious (X,L);
        if (! Islast (P,L))
        {
                Temp = P → Next;
                P → Next = Temp → Next;
                free (Temp);
```



Delete (30, L)

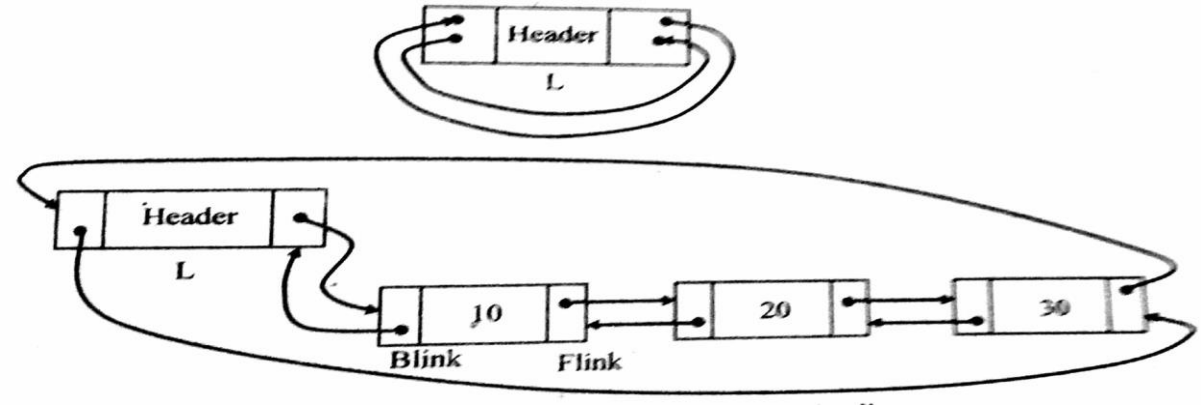


Deletion at last:

```
void dele_last (List L)
{
        Position Temp, P;
        P = L;
        while (P → Next → Next ! = L)
                        P = P → Next ;
        Temp = P → Next ;
        P → Next = L;
        free (Temp):
}
```

Doubly Linked list

A doubly circular linked list is a doubly linked list in which forward link of the last node points to the first node and backward link of first node points to the last node of the list.

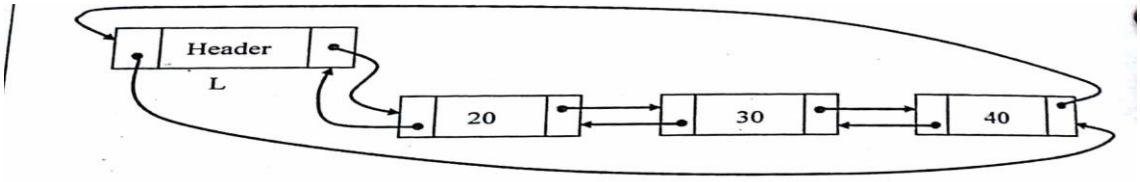


Structure Declaration:

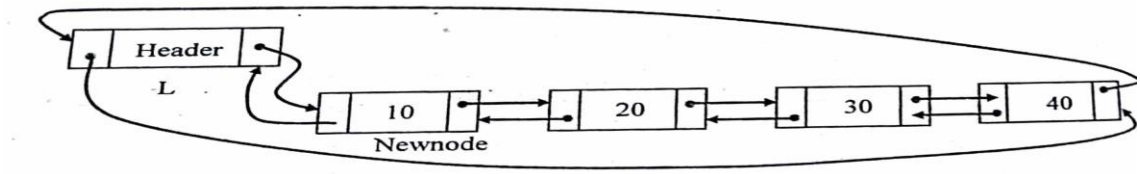
```
struct node
{
int Element;
struct node *FLINK;
struct node *BLINK;
}
```

Insert at beginning:

```
void Insert_beg (int X, List L)
{
        Position Newnode;
        Newnode = malloc (sizeof (struct node));
        if (Newnode ! = NULL)
        {
                {
                        Newnode → Element = X;
                        Newnode → Flink = L → Flink;
                        L → Flink → Blink = Newnode;
                        L → Flink = Newnode;
                        Newnode → Blink = L;
                }
        }
}
```

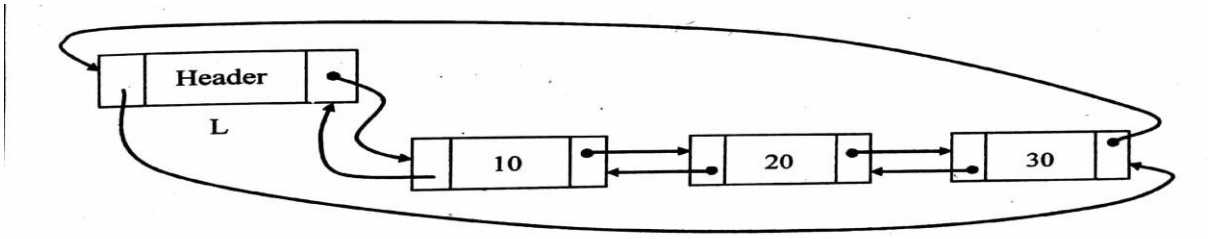


Insert (10,L)

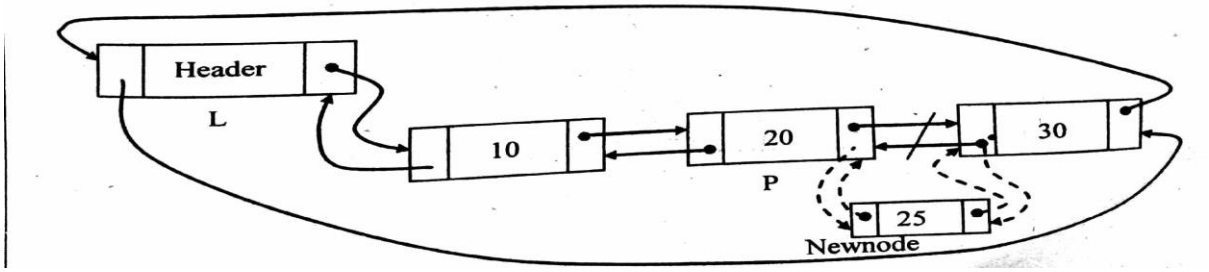


Insert at Middle:

```
void Insert_mid (int X, List L, Position P)
{
        Position Newnode;
        Newnode = malloc (sizeof (struct node));
        if (Newnode ! = NULL)
        {
                Newnode → Element = X;
                Newnode → Flink = P → Flink ;
                P → Flink → Blink = Newnode;
                Newnode → Blink = P;
                P → Flink = Newnode;
        }
}
```

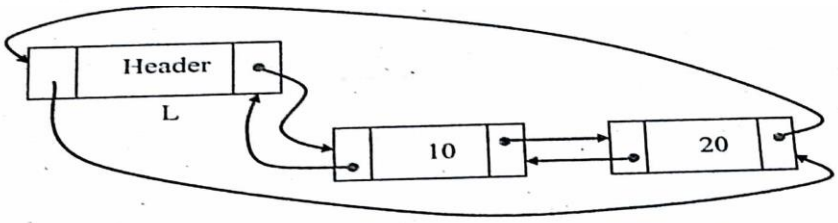


Insert (25, L, P)

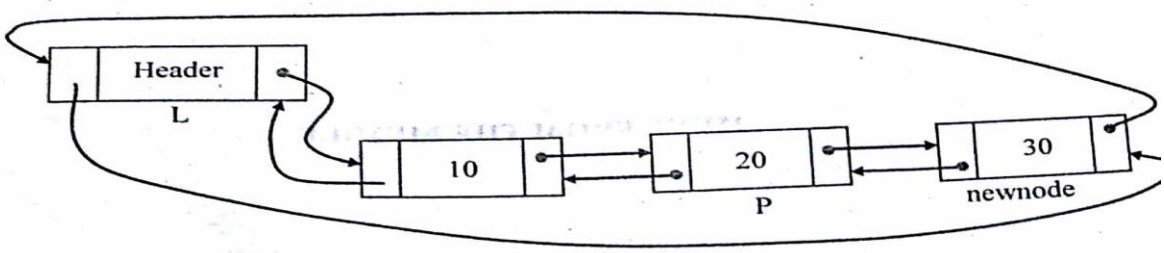


Insert at Last:

```
void Insert_last (int X, List L)
{
        Position Newnode, P;
        Newnode = malloc (sizeof (struct node));
        if (Newnode ! = NULL)
        {
                P = L;
                while (P → Flink ! = L)
                        P = P → Flink;
                Newnode → Element = X;
                P → Flink = Newnode;
                Newnode → Flink = L;
                Newnode → Blink = P;
                L → Blink = Newnode
        }
}
```



Insert (30, L)

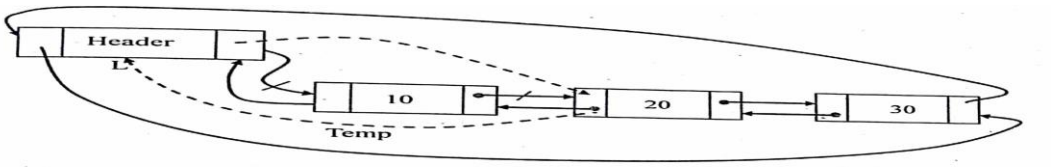


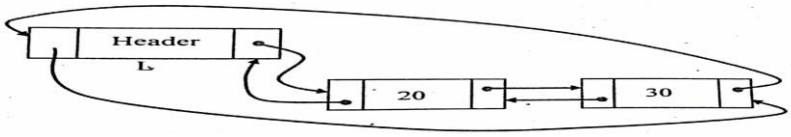
Deletion

Deleting First node

void dele_first(List L)

```
{
        Position Temp;
        if (L → Flink ! = NULL)
        {       Temp = L → Flink;
                L → Flink = Temp → Flink;
                Temp → Flink → Blink = L;
                free (Temp);
        }
}
```





Deletion at middle:

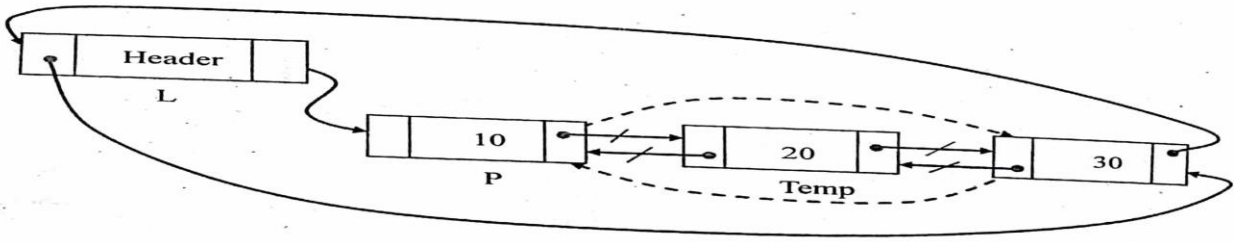
void dele_mid (int X, List L)

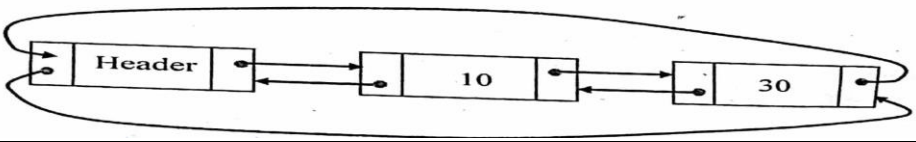
{ Position P, Temp;

P=FindPrevious(X);

```
        if (! Islast (P,L))
        {
                Temp = P → Flink;
                P → Flink = Temp → Flink;
                temp → Flink → Blink = P;
                free (Temp);
        }
}
```

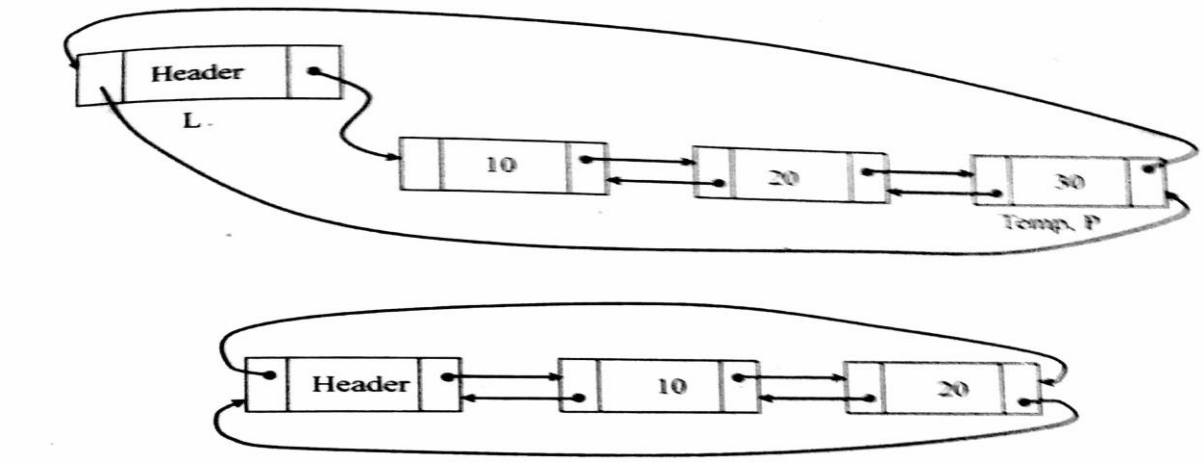
Delete (20, L)





Deletion at Last node:

```
void dele_last (List L)
{
        Position Temp;
        P = L;
        while (P → Flink ! = L)
                P = P → Flink;
        Temp = P;
        P → Blink → Flink = L;
        L → Blink = P → Blink;
        free (Temp);
}
```



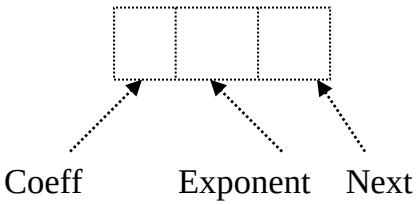
Application of linked list

Three applications that uses linked lists are,

1. The Polynomial ADT
2. Radix sort
3. Multilist

1) Polynomial ADT:

- To overcome the disadvantage of array implementation an alternative way is to use a singly linked list.
- Each term in the polynomial is contained in one cell, and the cells are sorted in decreasing order of exponents.



A first example where linked lists are used is called The Polynomial ADT.

Example:

P1:$4X^{10}+5X^{5}+3$

P2:$10X^{6}-5X^{2}+2X$

Strucuture declarations for Linked List Implementation of the polynomial ADT:

```
struct  link
  {
        int coeff;
        int pow;
        struct link *next;
   };struct link *poly1=NULL,*poly2=NULL,*poly=NULL;
```

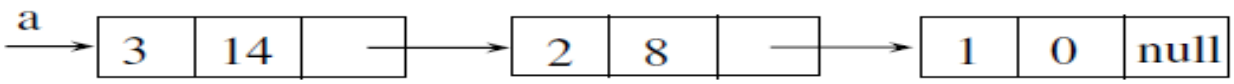
Procedure to add two polynomials

```
void polyadd(struct link *poly1,struct link *poly2,struct link *poly)
  {
        while(poly1->next != NULL && poly2->next != NULL)
         {
                if(poly1->pow > poly2->pow)
                  {
                        poly->pow=poly1->pow;
                        poly->coeff=poly1->coeff;
                        poly1=poly1->next;              }
                else if(poly1->pow < poly2->pow)
                  {
                        poly->pow=poly2->pow;
```

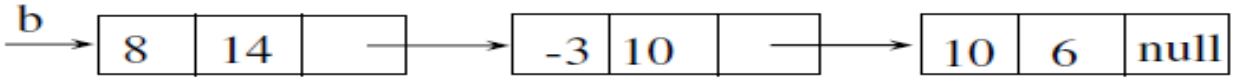
```
                poly->coeff=poly2->coeff;
                poly2=poly2->next;
              }
            else
              {
                poly->pow=poly1->pow;
                poly->coeff=poly1->coeff+poly2->coeff;
                poly1=poly1->next;
                poly2=poly2->next;
              }
            poly->next=(struct link *)malloc(sizeof(struct link));
            poly=poly->next;
            poly->next=NULL;
        }
  if(poly1->next != NULL)
      {
        poly->coeff = poly1->coeff;
        poly->pow = poly1->pow;
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
      }
  else
      {
        poly->coeff = poly2->coeff;
        poly->pow = poly2->pow;
        poly->next=(struct link *)malloc(sizeof(struct link));
        poly=poly->next;
        poly->next=NULL;
      }
}
```

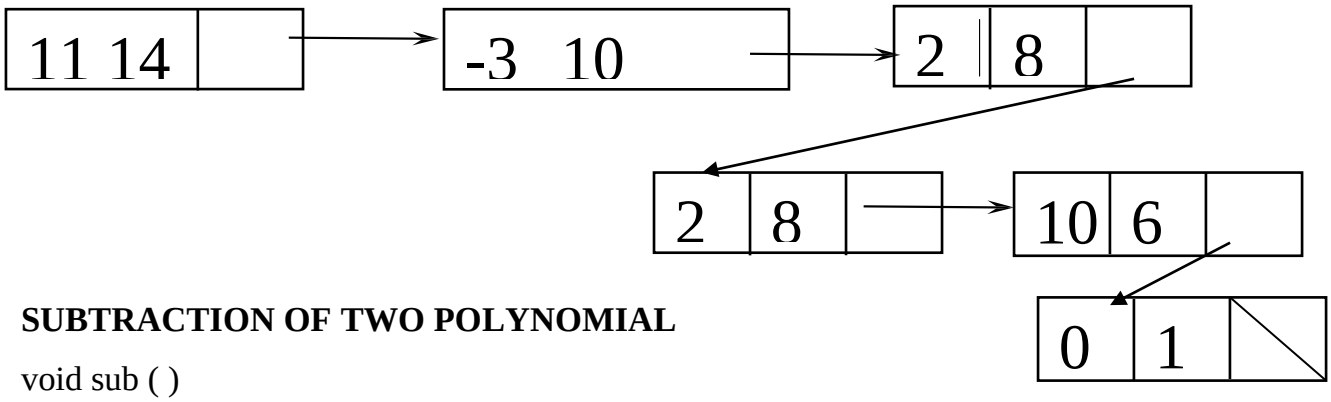
Finally we get the polynomial C as

$$a = 3x^{14} + 2x^{8} + 1$$



$$b = 8x^{14} - 3x^{10} + 10x^{6}$$





SUBTRACTION OF TWO POLYNOMIAL

```
void sub ( )
{
poly *ptr1, *ptr2, *newnode;
ptr1 = list1 ;
ptr2 = list 2;
while (ptr1! = NULL && ptr2! = NULL)
{
newnode = malloc (sizeof (Struct poly));
if (ptr1    power = = ptr2    power)
{
newnode→coeff = (ptr1    coeff) - (ptr2    coeff);
newnode→power = ptr1    power;
newnode→next = NULL;
list3 = create (list 3, newnode);
ptr1 = ptr1→next;
```

```
ptr2 = ptr2→next;    }
else
{
if (ptr1→power > ptr2→power)
{
newnode→coeff = ptr1→coeff;
newnode→power = ptr1→power;
newnode→next = NULL;
list 3 = create (list 3, newnode);
ptr1 = ptr1→next;    }
else
{
newnode→coeff = - (ptr2→coeff);
newnode→power = ptr2→power;
newnode→next = NULL;
list 3 = create (list 3, newnode);
ptr2 = ptr2    next;    }                }        }
```

POLYNOMIAL DIFFERENTIATION

```
void diff ( )
{
poly *ptr1, *newnode;
ptr1 = list 1;
while (ptr1 ! = NULL)
{
newnode = malloc (sizeof (Struct poly));
newnode    coeff = ptr1    coeff *ptr1    power;
newnode    power = ptr1    power - 1;
newnode    next = NULL;
list 3 = create (list 3, newnode);
ptr1 = ptr1→next;    }        }
```

Radix Sort

A second example where linked lists are used is called radix sort. Radix sort is also known as card sort. Because it was used, until the advent of modern computers, to sort old-style punch cards.

If we have n integers in the range 1 to m (or 0 to m - 1) 9, we can use this information to obtain a fast sort known as bucket sort. We keep an array called count, of size m, which is initialized to zero. Thus, count has m cells (or buckets), which are initially empty.

When ai is read, increment (by one) counts [ai]. After all the input is read, scan the count array, printing out a representation of the sorted list. This algorithm takes O(m + n); If m = (n), then bucket sort takes O(n) times.

The following example shows the action of radix sort on 10 numbers. The input is 64, 8, 216, 512, 27, 729, 0, 1, 343, and 125. The first step (Pass 1) bucket sorts by the least significant digit.. The buckets are as shown in below figure, so the list, sorted by least significant digit, is 0, 1, 512, 343, 64, 125, 216, 27, 8, 729. These are now sorted by the next least significant digit (the tens digit here)

Pass 2 gives output 0, 1, 8, 512, 216, 125, 27, 729, 343, 64. This list is now sorted with respect to the two least significant digits. The final pass, shown in Figure 3.26, bucket-sorts by most significant digit.

The final list is 0, 1, 8, 27, 64, 125, 216, 343, 512, and 729.

The running time is O(p(n + b)) where p is the number of passes, n is the number of elements to sort, and b is the number of buckets. In our case, b = n.

Buckets after first step of radix sort

0	**1**	**512**	**343**	**64**	**125**	**216**	**27**	**8**	**729**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Buckets after the second pass of radix sort

8 **1** **0**	**216** **512**	**729** **27** **125**		**343**		**64**			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

Buckets after the last pass of radix sort

64 27 8 1 0	125	216	343		512		729		
0	1	2	3	4	5	6	7	8	9

Multilists

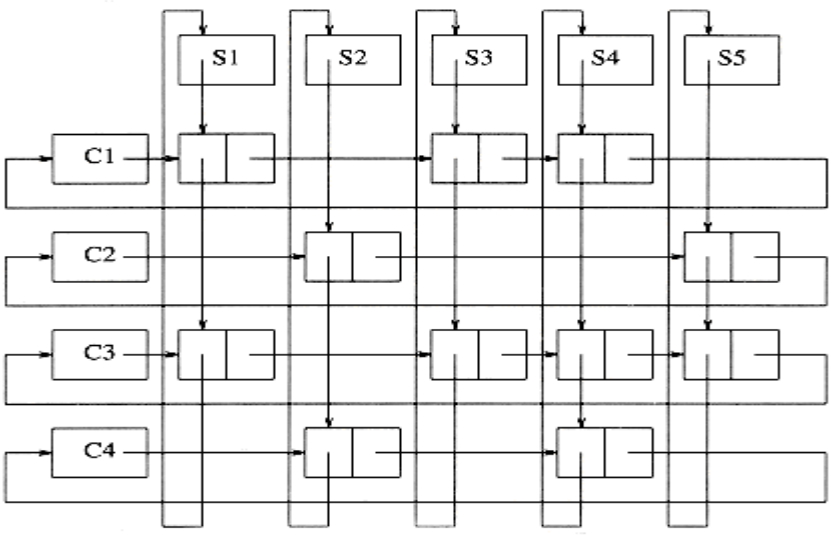
A university with 40,000 students and 2,500 courses needs to be able to generate two types of reports. The first report lists the class registration for each class, and the second report lists, by student, the classes that each student is registered for.

If we use a two-dimensional array, such an array would have 100 million entries. The average student registers for about three courses, so only 120,000 of these entries, or roughly 0.1 percent, would actually have meaningful data.

To avoid the wastage of memory, a linked list can be used. We can use two link list one contains the students in the class. Another linked list contains the classes the student is registered for.

All lists use a header and are circular. To list all of the students in class C3, we start at C3 and traverse its list . The first cell belongs to student S1.

Multilist implementation for registration problem



Linked List Implementation of Multilists:

- Multilists can be used to represent the above scenario.
    - One list to represent each class containing the students in the class.
    - One list to represent each student containing the classes the student is registered for.
- All lists use a header and are circular.
- To list all the students is class C3:
    - Start the traversal at C3 and traverse its list (by going right).
    - The first cell belongs to student S1.
    - The next cell belongs to student S3. By continuing this it is found that student S4 and student S5 also belongs to the class C3.
- In a similar manner, for any student, all of the classes in which the student is registered can be determined.
- Advantage of Using Linked List:
    - Saves memory space.
- Disadvantage of Using Linked List:
    - Saves memory space only at the expense of time.

UNIT II STACKS AND QUEUES 9

Stack ADT – Operations – Applications – Balancing Symbols – Evaluating arithmetic expressions- Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Circular Queue – DeQueue – Applications of Queues.

The Stack ADT

Stack Model

A stack is a list with the restriction that inserts and deletes can be performed in only one position, namely the end of the list called the top. Stacks are sometimes known as **LIFO** (last in, first out) lists.

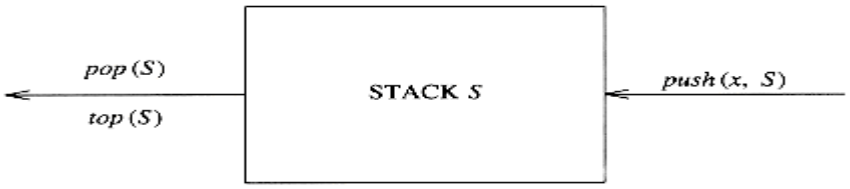
The fundamental operations on a stack are

1. **Push**, which is equivalent to an insert,
2. **Pop,** deletes the most recently inserted element.
3. **Top,** display the topmost element in the stack.

Error conditions

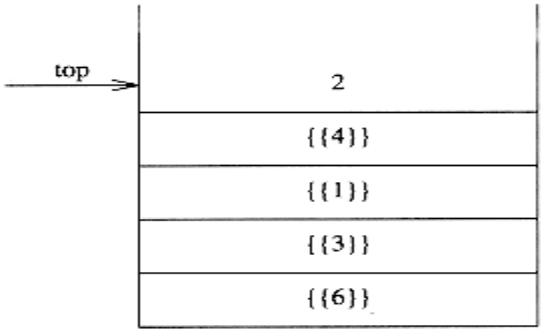
Push onto the Full Stack and Pop or Top on an empty stack is generally considered an error in the stack ADT.

Stack model: input to a stack is by Push, output is by Pop



The model depicted in above figure signifies that pushes are input operations and pops and tops are output.

Stack model: only the top element is accessible



Implementation of Stacks

A stack is a list, gives two popular implementations.

1. Array implementation
2. Linked list implementation

Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a push by inserting at the front of the list. We perform a pop by deleting the element at the front of the list. A top operation merely examines the element at the front of the list, returning its value. Sometimes the pop and top operations are combined into one.

Creating an empty stack is also simple. We merely create a header node; make_null sets the next pointer to NULL.

The push is implemented as an insertion into the front of a linked list, where the front of the list serves as the top of the stack.

The top is performed by examining the element in the first position of the list.

The pop will delete from the front of the list.

It should be clear that all the operations take constant time, because less a loop that depends on this size.

Drawbacks and solution

These implementations uses the calls to malloc and free are expensive, especially in comparison to the pointer manipulation routines. Some of this can be avoided by using a second stack, which is initially empty. When a cell is to be disposed from the first stack, it is merely placed on the second stack. Then, when new cells are needed for the first stack, the second stack is checked first.

Type declaration for linked list implementation of the stack ADT

```
struct Node;
typedef struct node *ptrToNode;
typedef ptrToNode Stack;
int IsEmpty(Stack S);
Stack CreateSatck(void);
```

```
void DisposeStack(Stack S);
void MakeEmpty(Stack S);
void Push(ElementType X, Stack S);
ElementType Top (Stack S);
Void Pop(Stack S);

struct node
{
Element_type element;
PtrToNode next;
};
```

Routine to test whether a stack is empty-linked list implementation

This routine checks whether Stack is empty or not. If it is not empty it will return a pointer to the stack. Otherwise return NULL

```
int  is_empty( STACK S )
{
return( S->next == NULL );
}
```

Routine to create an empty stack-linked list implementation

This routine creates a Stack and return a pointer of the stack. Otherwise return a warning to say Stack is not created.

```
STACK create_stack( void )
{
STACK S;
S = malloc( sizeof( struct node ) );
if( S == NULL )
fatal_error("Out of space!!!");
return S;       }
```

Routine to make the stack as empty-linked list implementation

This routine makes Stack as empty and return NULL pointer.

```
Void makeEmpty( STACK S )
{
if( S == NULL )
error ("Must use create_stack first");
else
while (!IsEmpty(S))
pop(S);         }
```

Routine to push onto a stack-linked list implementation

This routine is to insert the new element onto the top of the stack.

```
Void push( element_type x, STACK S )
{
node_ptr tmp_cell;
tmp_cell = (node_ptr) malloc( sizeof ( struct node ) );
if( tmp_cell == NULL )
fatal_error("Out of space!!!");
else
{
tmp_cell->element = x;
tmp_cell->next = S->next;
S->next = tmp_cell;           }         }
```

Routine to return top element in a stack--linked list implementation

This routine is to return the topmost element from the stack.

```
element_type top( STACK S )
{
if( is_empty( S ) )
error("Empty stack");
else
return S->next->element;
}
```

Routine to pop from a stack--linked list implementation

This routine is to delete the topmost element from the stack.

```
Void pop( STACK S )
{
PtrToNode first_cell;
if( is_empty( S ) )
error("Empty stack");
else
{
first_cell = S->next;
S->next = S->next->next;
free( first_cell );
}       }
```

Array implementation of Stacks

An alternative implementation to avoid pointers is that by using an array implementation. One problem here is that we need to declare an array size ahead of time. Generally this is not a problem, if the actual number of elements in the stack is knows in advance. It is usually easy to declare the array to be large enough without wasting too much space.

Associated with each stack is the top of stack, **tos,** which is -1 for an empty stack. To push some element x onto the stack, we increment tos and then set STACK[tos] = x, where STACK is the array representing the actual stack.

To pop, we set the return value to STACK[tos] and then decrement tos.

Notice that these operations are performed in not only constant time, but very fast constant time.

Error checking:

The efficiency of implementation in stacks is error testing. linked list implementation carefully checked for errors.

A pop on an empty stack or a push on a full stack will overflow the array bounds and cause a crash. Ensuring that this routines does not attempt to pop an empty stack and Push onto the full stack.

A STACK is defined as a pointer to a structure. The structure contains the top_of_stack and stack_size fields.

Once the maximum size is known,the stack array can be dynamically allocated.

Stack Declaration

```
Struct Stack Record
typedef struct StackRecord  *Stack;
int IsEmpty(Stack S);
Stack CreateStack(int MaxElements);
void DisposeStack(Stack S);
void MakeEmpty(Stack S);
void Push(ElementType X, Stack S);
ElementType Top (Stack S);
Void Pop(Stack S);
ElementType TopandPop (Stack S);
        struct StackRecord
        {
        Int Capacity;
        int TopofSatck;
        ElementType *array;
        };
#define EmptyTOS  (-1)     /* Signifies an empty stack */
#define MinStackSize (5)
```

Routine to create an empty stack- Array implementation

This routine creates a Stack and return a pointer of the stack. Otherwise return a warning to say Stack is not created.

```
Stack CreateStack( unsigned int MaxElements )
{
STACK S;
if( MaxElements < MinStackSize )
error("Stack size is too small");
S = (malloc( sizeof( struct StackRecord ) );
```

```
if( S == NULL )
fatal_error("Out of space!!!");
S->Array = malloc( sizeof( ElementType ) * MaxElements );
if( S->Array == NULL )
fatalerror("Out of space!!!");
S->Capacity = MaxElements;
MakeEmpty(S);
return( S );  }
```

Routine for freeing stack--array implementation

This routine frees or removes the Stack Structure itself by deleting the array elements one by one.

```
Void dispose_stack( Stack S )
{
if( S != NULL )
{       free( S->Array );
free( S );      }       }
```

Routine to test whether a stack is empty--array implementation

This routine is to check whether stack is empty or not.

```
int IsEmpty( Stack S )
{
return( S->top_of_stack == EmptyTOS ); }
```

Routine to create an empty stack--array implementation

This routine helps to make the Stack as empty one.

```
Void MakeEmpty( STACK S )
{
S->top_of_stack = EMPTY_TOS;  }
```

Routine to push onto a stack--array implementation

This routine will insert the new elemnt onto the top of the stack using stack pointer.

```
Void push( ElementType X, Stack S )
{       if( IsFull( S ) )
```

```
        Error("Full stack");
    else
        S->Array[ ++S->TopofStack ] = X;        }
```

Routine to return top of stack--array implementation

This routine is to return the topmost element from the stack.

```
ElementType Top( Stack S )
{
if( !IsEmpty( S ) )
return S->Array[ S-> TopofStack];
error("Empty stack");
return 0;
}
```

Routine to pop from a stack--array implementation

This routine is to delete the topmost element from the stack.

```
    Void pop( Stack S )
    {
    if( IsEmpty( S ) )
    error("Empty stack");
    else
    S->TopofStack--;
    }
```

Routine to give top element and pop a stack--array implementation

This routine is to return as well as remove the topmost element from the stack.

```
    ElementType TopandPop( Stack S )
    {
        if( IsEmpty( S ) )
        error("Empty stack");
        else
        return S->Array[ S->TopofStack-- ];
        }
```

Stack Applications

Stack is used for the following applications.

1. Reversing of the string
2. Tower's of Hanoi's problem
3. Balancing Symbols
4. Conversion of Infix to postfix expression
5. Conversion of Infix to prefix expression
6. Evaluation of Postfix expression
7. Used in Function calls

Balancing Symbols

Compilers check your programs for syntax errors, but frequently a lack of one symbol (such as a missing brace or comment starter) will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error.

A useful tool in this situation is a program that checks whether everything is balanced. Thus, every right brace, bracket, and parenthesis must correspond to their left counterparts.

The sequence [()] is legal, but [(]) is wrong. That it is easy to check these things. For simplicity, we will just check for balancing of parentheses, brackets, and braces and ignore any other character that appears.

The simple algorithm uses a stack and is as follows:

- Make an empty stack.
- Read characters until end of file.
- If the character is an open anything, push it onto the stack.
- If it is a close anything, then
  - ✓ If the stack is empty report an error.
  - ✓ Otherwise, pop the stack.
  - ✓ If the symbol popped is not the corresponding opening symbol, then report an error.
- At end of file, if the stack is not empty report an error.

Expression:

Expression is defined as a collection of operands and operators. The operators can be arithmetic, logical or Boolean operators.

Rules for expression

- ✓ No two operand should be continuous
- ✓ No two operator should be continuous

Types of expression:

Based on the position of the operator, it is classified into three.

1. Infix Expression / Standard notation
2. Prefix Expression/ Polished notation
3. Postfix Expression / Reversed Polished notation

Infix Expression:

In an expression if the operator is placed in between the operands, then it is called as **Infix Expression.**

Eg : A+B

Prefix Expression:

In an expression if the operator is placed before the operands, then it is called as **Prefix Expression.**

Eg : +AB

Postfix Expression:

In an expression if the operator is placed after the operands, then it is called as **Postfix Expression.**

Eg : AB+

Conversion of infix to Postfix Expressions

Stack is used to convert an expression in standard form (otherwise known as infix) into postfix. We will concentrate on a small version of the general problem by allowing only the operators +, *, and (,), and insisting on the usual precedence rules.

Suppose we want to convert the infix expression

a + b * c + (d * e + f) * g .

A correct answer is a b c * + d e * f + g * +.

Algorithm:

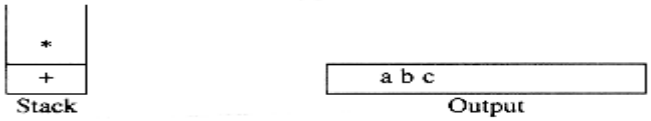
1. We start with an initially empty stack
2. When an operand is read, it is immediately placed onto the output.
3. Operators are not immediately placed onto the output, so they must be saved somewhere. The correct thing to do is to place operators that have been seen, but not placed on the output, onto the stack. We will also stack left parentheses when they are encountered.
4. If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.
5. If we see any other symbol ('+','*', '('), then we pop entries from the stack until we find an entry of lower priority. One exception is that we never remove a '(' from the stack except when processing a ')'. For the purposes of this operation, '+' has lowest priority and '(' highest. When the popping is done, we push the operand onto the stack.
6. Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

To see how this algorithm performs, we will convert the infix expression into its postfix form.

a + b * c + (d * e + f) * g

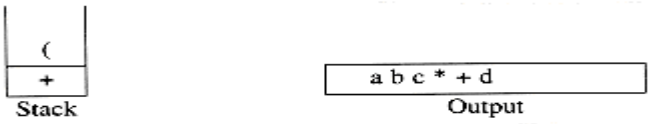
First, the symbol a is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next b is read and passed through to the output. Then the stack will be as follows.

Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, c is read and output.

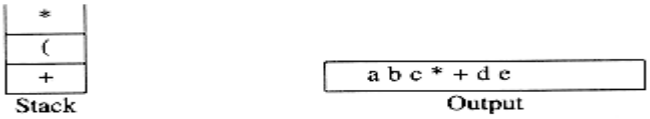


The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output, pop the other '+', which is not of lower but equal priority, on the stack, and then push the '+'.

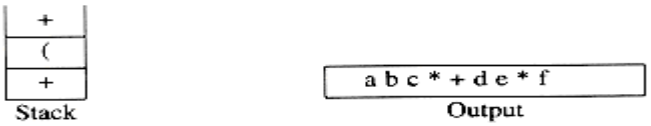
The next symbol read is an '(', which, being of highest precedence, is placed on the stack. Then d is read and output.



We continue by reading a '*'. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.

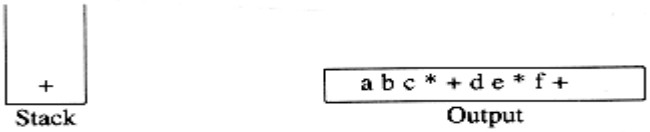


The next symbol read is a '+'. We pop and output '*' and then push '+'. Then we read and output f.

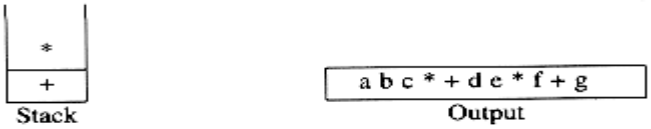


.

Now we read a ')', so the stack is emptied back to the '('. We output a '+' 0nto the stack.



We read a '*' next; it is pushed onto the stack. Then g is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.


Stack

a b c * + d e * f + g * +
Output

As before, this conversion requires only O(n) time and works in one pass through the input. We can add subtraction and division to this repertoire by assigning subtraction and addition equal priority and multiplication and division equal priority.

A subtle point is that the expression a - b - c will be converted to ab - c- and not abc - -. Our algorithm does the right thing, because these operators associate from left to right. This is not necessarily the case in general, since exponentiation associates right to left: 223 = 28 = 256 not 43 = 64.

Evaluation of a Postfix Expression

Algorithm:

When a number is seen, it is pushed onto the stack;

When an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stackand the result is pushed onto the stack.

For example, the postfix expression **6 5 2 3 + 8 * + 3 + *** is evaluated as follows:

The first four symbols are placed on the stack. The resulting stack is

TopofStack	3 2 5 6

Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.

TopofStack	5 5 6

Next 8 is pushed.

TopofStack	8 5 5 6

Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed.

TopofStack	40 5 6

Next a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed.

TopofStack	45 6

Now, 3 is pushed.

TopofStack	3 45 6

Next '+' pops 3 and 45 and pushes 45 + 3 = 48.

TopofStack	48 6

Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed.

TopofStack	288

The time to evaluate a postfix expression is O(n), because processing each element in the input consists of stack operations and thus takes constant time. The algorithm to do so is very simple.

Advantage of postfix expression:

When an expression is given in postfix notation, there is no need to know any precedence rules;

Function Calls

- When a call is made to a new function, all the variables local to the calling routine need to be saved by the system. Otherwise the new function will overwrite the calling routine's variables.
- The current location in the routine must be saved so that the new function knows where to go after it is done.
- The reason that this problem is similar to balancing symbols is that a function call and function return are essentially the same as an open parenthesis and closed parenthesis, so the same ideas should work.
- When there is a function call, all the important information that needs to be saved, such as register values (corresponding to variable names) and the return address is saved "on a piece of paper" in an abstract way and put at the top of a pile. Then the control is transferred to the new function, which is free to replace the registers with its values.
- If it makes other function calls, it follows the same procedure. When the function wants to return, it looks at the "paper" at the top of the pile and restores all the registers. It then makes the return jump.
- The information saved is called either an activation record or stack frame.
- There is always the possibility that you will run out of stack space by having too many simultaneously active functions. Running out of stack space is always a fatal error.
- In normal events, you should not run out of stack space; doing so is usually an indication of runaway recursion. On the other hand, some perfectly legal and seemingly innocuous program can cause you to run out of stack space.

A bad use of recursion: printing a linked list

```
void /* Not using a header */
print_list( LIST L )
{       if( L != NULL )
{
print_element( L->element );
print_list( L->next );}       }
```

- The above routine prints out a linked list, is perfectly legal and actually correct. It properly handles the base case of an empty list, and the recursion is fine. This program can be proven correct.
- Activation records are typically large because of all the information they contain, so this program is likely to run out of stack space. This program is an example of an extremely bad use of recursion known as **tail recursion**. **Tail recursion refers to a recursive call at the last line.**
- **Tail recursion** can be mechanically eliminated by changing the recursive call to a goto receded by one assignment per function argument.
- This simulates the recursive call because nothing needs to be saved -- after the recursive call finishes, there is really no need to know the saved values. Because of this, we can just go to the top of the function with the values that would have been used in a recursive call.

The below program is the improved version. Removal of tail recursion is so simple that some compilers do it automatically.

Printing a list without recursion

```
Void print_list( LIST L ) /* No header */
{
top:
if( L != NULL )
{
print_element( L->element );
L = L->next;
goto top;
}
}
```

Recursion can always be completely removed. But doing so can be quite tedious. The non-recursive programs are generally faster than recursive programs; the speed advantage rarely justifies the lack of clarity that results from removing the recursion.

The Queue ADT

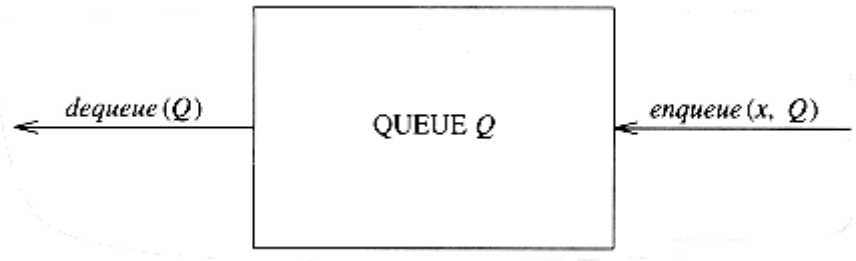
Queue is also a list in which insertion is done at one end, whereas deletion is performed at the other end. Insertion will be at rear end of the queue and deletion will be at front of the queue. It is also called as FIFO (**F**irst **I**n **F**irst **O**ut) which means the element which inserted first will be removed first from the queue.

Queue Model

The basic operations on a queue are

1. enqueue, which inserts an element at the end of the list (called the rear)
2. dequeue, which deletes (and returns) the element at the start of the list (known as the front).

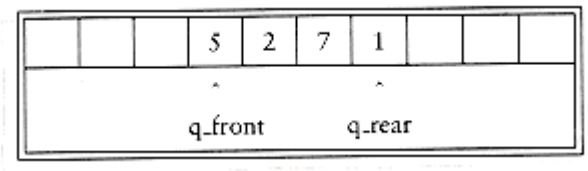
Abstract model of a queue



Array Implementation of Queues

- Like stacks, both the linked list and array implementations give fast O(1) running times for every operation. The linked list implementation is straightforward and left as an exercise. We will now discuss an array implementation of queues.
- For each queue data structure, we keep an array, QUEUE[], and the positions q_front and q_rear, which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, q_size.

The following figure shows a queue in some intermediate state.



- By the way, the cells that are blanks have undefined values in them. In particular, the first two cells have elements that used to be in the queue.

- To enqueue an element x, we increment q_size and q_rear, then set QUEUE[q_rear] = x.
- To dequeue an element, we set the return value to QUEUE[q_front], decrement q_size, and then increment q_front.. After 10 enqueues, the queue appears to be full, since q_front is now 10, and the next enqueue would be in a nonexistent position.
- However, there might only be a few elements in the queue, because several elements may have already been dequeued.
- The simple solution is that whenever q_front or q_rear gets to the end of the array, it is wrapped around to the beginning. This is known as **a circular array implementation.**

If incrementing either q_rear or q_front causes it to go past the array, the value is reset to the first position in the array.

There are two warnings about the circular array implementation of queues.

- First, it is important to check the queue for emptiness, because a dequeue when the queue is empty will return an undefined value.
- Secondly, some programmers use different ways of representing the front and rear of a queue. For instance, some do not use an entry to keep track of the size, because they rely on the base case that when the queue is empty, q_rear = q_front - 1.

If the size is not part of the structure, then if the array size is A_SIZE, the queue is full when there are A_SIZE -1 elements.

In applications where you are sure that the number of enqueues is not larger than the size of the queue, **the wraparound is not necessary**.

The routine **queue_create and queue_dispose** routines also need to be provided. We also provide routines to test whether a queue is empty and to make an empty queue.

Notice that q_rear is preinitialized to 1 before q_front. The final operation we will write is the enqueue routine.

Type declarations for queue--array implementation

```
struct QueueRecord
{
int Capacity;
int Front;
int Rear;
int Size; /* Current # of elements in Q */
ElementType *Array;
};
typedef struct QueueRecord  * Queue;
```

Routine to test whether a queue is empty-array implementation

```
int isempty( Queue Q )
{
return( Q->q_size == 0 );    }
```

Routine to make an empty queue-array implementation

```
Void makeempty ( Queue Q )
{
Q->size = 0;
Q->Front = -1;
Q->Rear = -1;                    }
```

Routines to enqueue-array implementation

```
static int  succ(int value, Queue Q )
{
if( ++value = = Q->Capacity )
value = 0;
return value;  }
Void enqueue( Elementtype x, Queue Q )
{
if( isfull( Q ) )
error("Full queue");
```

```
else
{
Q->Size++;
Q->Rear = succ( Q->Rear, Q );
Q->Array[ Q->Rear ] = x;
}       }
```

Applications of Queues

The applications are,

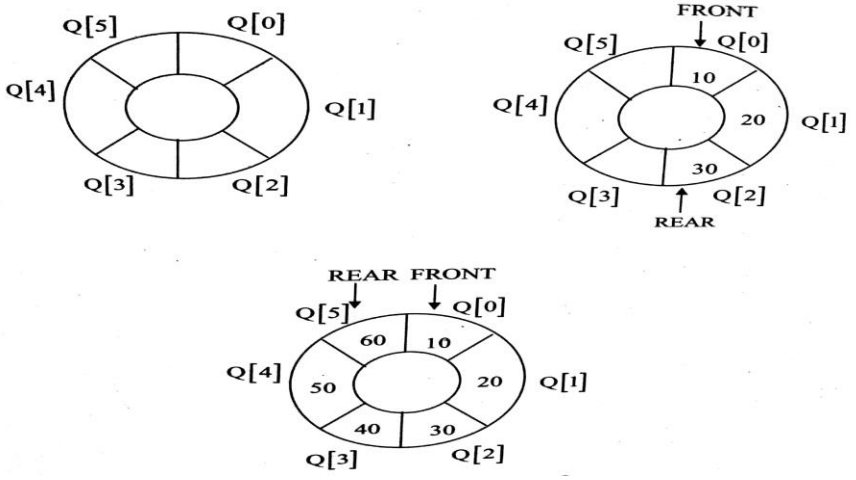
1. When jobs are submitted to a printer, they are arranged in order of arrival. Then jobs sent to a line printer are placed on a queue.
2. Lines at ticket counters are queues, because service is first-come first-served.
3. Another example concerns computer networks. There are many network setups of personal computers in which the disk is attached to one machine, known as the file server.
4. Users on other machines are given access to files on a first-come first-served basis, so the data structure is a queue.

Circular Queue:

In Circular Queue, the insertion of a new element is performed at the very first locations of the queue if the last location of the queue is full, in which the first element comes after the last element.

Advantages:

It overcomes the problem of unutilized space in linear queue, when it is implemented as arrays.



To perform the insertion of the element to the queue, the position of the element is calculated as **rear= (rear+1) % queue_size and set Q[rear]=value.** Similarly the element deleted from the queue using front = (front + 1) % queue_size.

Enqueue:

This routine insert the new element at rear position of the circular queue.

```
void CEnqueue (int X)
{
if (Front = = (rear + 1) % Maxsize)
        print ("Queue is overflow");
else
{
        if (front = = -1)
                front = rear = 0;
        else
                rear = (rear + 1)% Maxsize;
                CQueue [rear] = X;
}
}
```

Dequeue:

This routine deletes the element from the front of the circular queue.

```
void CQ_dequeue( )
{
 If(front==-1 && rear==-1)
Print("Queue is empty");
Else
{
Temp=CQueue[front];
If(front==rear)
Front=rear=-1;
Else
Front=(front+1)% maxsize;
} }
```

Priority Queue:

In an priority queue, an element with high priority is served before an element with lower priority.

If two elements with the same priority, they are served according to their order in the queue.

Two types of priority Queue.

1. Max Priority Queue
2. Min Priority Queue

1. Max Priority Queue

In Max Priority Queue, elements are inserial in the order in which they arrive they queue and always maxmum value is removed first from the queue.

ex : insert in order 8, 3, 2, 5 removed in the order 8, 5, 3, 2.

Max Priority queue, the following operation are performed

1. is Empty ( ) - check whether queue is empty
2. insert ( ) - Inserts a new value into the queue.
3. findmax ( ) - Find max value in the queue.
4. remove ( ) - Delete max value from the queue.

2. Min Priority Queue Representation

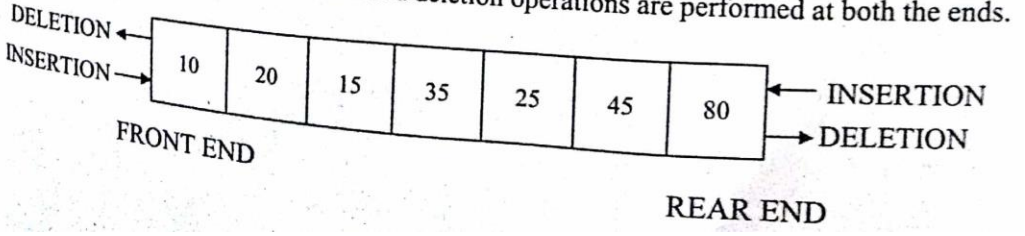
Min Priority Queue is similar to Max priority queue except removing maximum elements first, we remove min. element first in min priority queue.

In Min Priority Queue the following operation are performed

1. is empty ( ) - check whether queue is empty
2. insert ( ) = inserts a new value into the queue
3. findMin ( ) = find min. value in the queue
4. remove ( ) - Delete min value from the queue.

2.7 DOUBLE ENDED QUEUE (DEQUE)

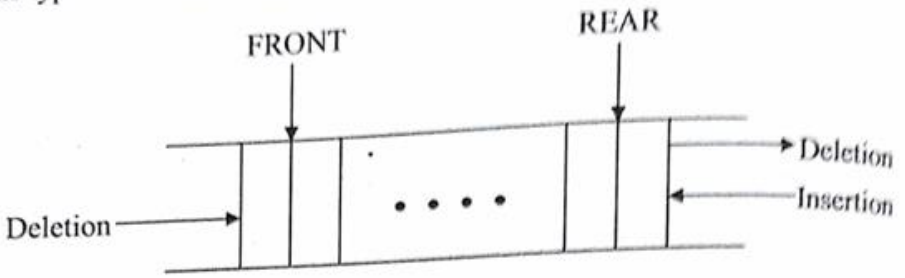
In Double Ended Queue, insertion and deletion operations are performed at both the ends.



DEQUE

INPUT RESTRICTED DEQUE

In this type insertions are allowed at one end and deletions are allowed at both ends.



OUTPUT RESTRICTED DEQUE

In this type deletions are allowed at one end and insertions are allowed at both ends.

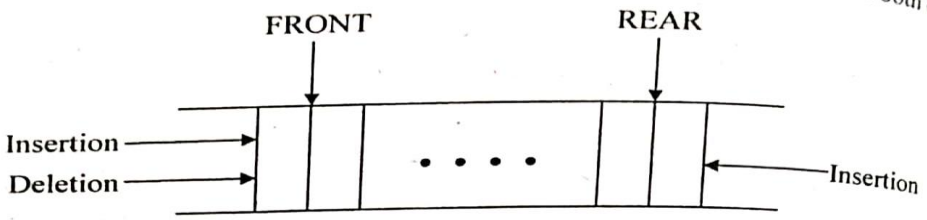


Fig. 4.20 Output restricted output

Insertion at the rear end

Step 1 : Check for the overflow condition.

Step 2 : If it is true, display that the queue is full

Step 3 : Otherwise, If the rear and front pointers are at the initial values (-1). Increment both the pointers. Goto step 5.

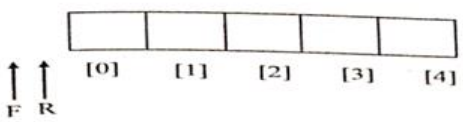
Step 4 : Increment the rear pointer

Step 5 : Assign the value to Q[rear]

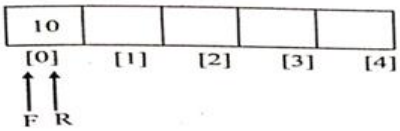
ROUTINE TO INSERT AN ELEMENT REAR AT END

```
void Insert_rear (int X, DQueue DQ)
{
        if (Rear = = Arraysize -1)
        {
         printf ("Queue Overflow");
         return ;
        }
        else
        {
         Rear = Rear + 1;
         DQ[Rear] = x;
         if (Front = = -1)
                Front = 0;
        }
}
```

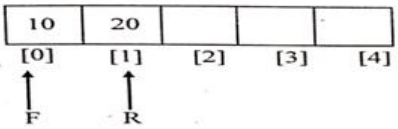
Insertion at the rear end



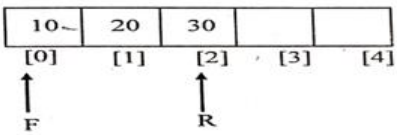
Insert_rear (10)



Insert_rear (20)



Insert_rear (30)



Insertion at front end

Step 1 : Check the front pointer, if it is in the first position (0) then display an error message that the value cannot be inserted at the front end.

Step 2 : Otherwise, decrement the front pointer

Step 3 : Assign the value to Q[front]

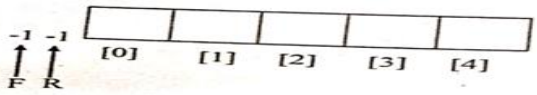
ROUTINE TO INSERT AN ELEMENT AT FRONT END

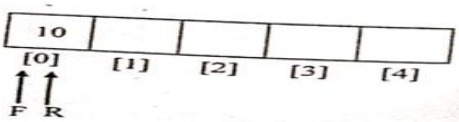
```
void Insert_front(int x, DQueue DQ)
{
        if(Front == 0)                          // element already exists in first position
        {
                printf("cannot Insert at the front position");
                return;
        }
        else
        {
                if(Front == -1)
                {
                        Front = Front + 1;
                        DQ[Front] = X;
                        if(Rear == -1)
                                Rear = 0;
                }
                else
                {
                        Front = Front -1;
                        DQ[Front] = X;
                }
        }
}
```

Insertion at the front end :

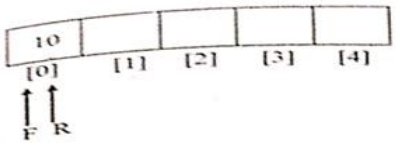
Case 1

Insert_front (10)



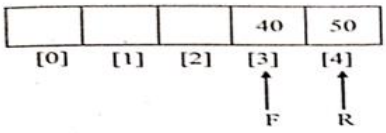


Insert_front (5)

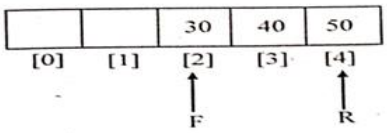


Here decrement of the front pointer may point to the initial (-1) position. So the value cannot be inserted even there is a space in the queue.

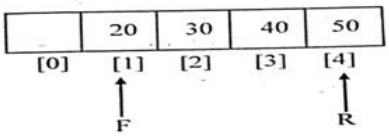
Case 2



Insert_front (30)



Insert_front (30)



Deletion from Rear End

Step 1 : Check the rear pointer. If it is in the initial value then display that the value cannot be deleted.

Step 2 : Otherwise, delete element at the rear position.

Step 3 : If the rear and front pointers are at the same position, reinitialize both the pointers.

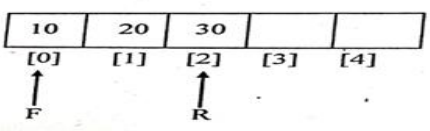
Step 4 : Otherwise, decrement the rear pointer.

4.30

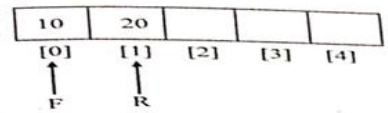
ROUTINE TO DELETE AN ELEMENT FROM

```
void Dequeue_rear (DQueue DQ)
{ int X;
  if (Rear = = -1)
  {
        printf ("Queue is empty");
        return ;
  }
  else
  {
        X = DQ [Rear];
        if (Front = = Rear)
        {
              Front = -1;
              Rear = -1;
        }
        else
        {
              Rear = Rear -1;
        }
  }
}
```

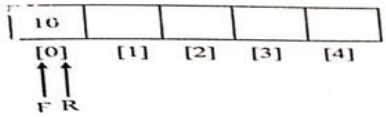
Deletion from Rear End



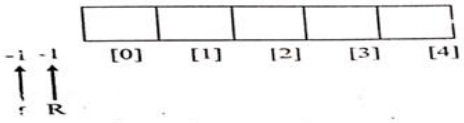
Dequeue_rear ()



Dequeue_rear ()



Dequeue_rear ()



Deletion from Front End :

Step 1 : Check for the underflow condition. If it is true display that the queue is empty.

Step 2 : Otherwise, delete the element at the front position, by asigning X as Q[front]

Step 3 : If the rear and front pointer points to the same position (ie) only one value is present, then reinitialize both the pointers.

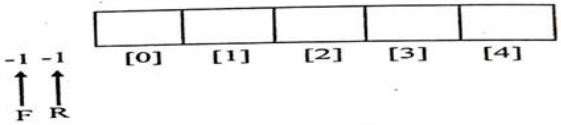
Step 4 : Otherwise, Increment the front pointer

ROUTINE TO DELETE AN ELEMENT FROM FRONT END

```
void Dequeue_front(DQueue DQ)
{
 int X;
 if(Front == -1)
 {
        printf("Queue is Underflow");
        return;
 }
 else
 {
        X = DQ[Front];
        if(Front == Rear)
        {
                Front = -1;
                Rear = -1;
        }
        else
        {
                Front = Front + 1;
        }
 }
}
```

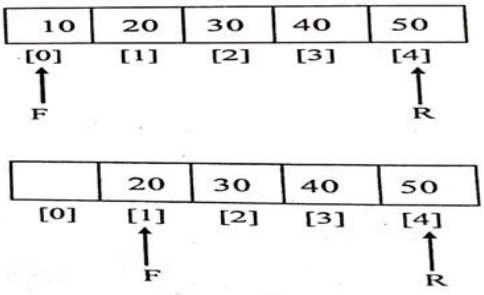
Deletion from front end

Dequeue_ front ()



Q – is empty

Dequeue_ front ()



UNIT III TREES 9

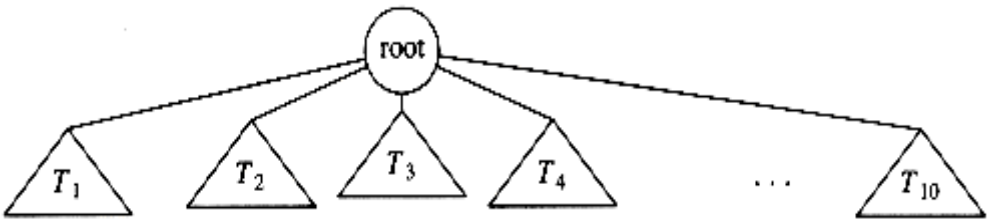
Tree ADT – Tree Traversals - Binary Tree ADT – Expression trees – Binary Search Tree ADT – AVL Trees – Priority Queue (Heaps) – Binary Heap.

TREES

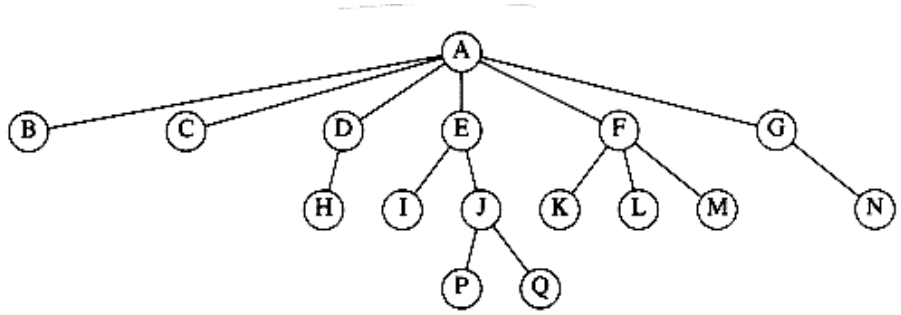
Tree is a Non- Linear datastructure in which data are stored in a hierarchal manner. It is also defined as a collection of nodes. The collection can be empty. Otherwise, a tree consists of a distinguished node r, called the root, and zero or more (sub) trees T1, T2, . . . , Tk, each of whose roots are connected by a directed edge to r.

The root of each subtree is said to be a child of r, and r is the parent of each subtree root. A tree is a collection of n nodes, one of which is the root, and n - 1 edges. That there are n - 1 edges follows from the fact that each edge connects some node to its parent and every node except the root has one parent

Generic tree



A tree



Terms in Tree

In the tree above figure, the **root** is A.

- ✓ Node F has A as a **parent** and K, L, and M as children.
- ✓ Each node may have an arbitrary number of children, possibly zero.

- ✓ Nodes with no children are known as **leaves**;
- ✓ The **leaves** in the tree above are B, C, H, I, P, Q, K, L, M, and N.
- ✓ Nodes with the same parent are **siblings**; thus K, L, and M are all siblings. **Grandparent** and **grandchild** relations can be defined in a similar manner.
- ✓ A **path** from node n1 to nk is defined as a sequence of nodes n1, n2, . . . , nk such that ni is the parent of ni+1 for 1 i < k.
- ✓ The **length of this path** is the number of edges on the path, namely k -1.
- ✓ There is a path of length zero from every node to itself.
- ✓ For any node ni, the **depth of ni** is the length of the unique path from the root to ni. Thus, the root is at depth 0.
- ✓ The **height of ni** is the longest path from ni to a leaf. Thus all leaves are at height 0.
- ✓ The height of a tree is equal to the height of the root.

  Example: For the above tree,

  E is at depth 1 and height 2;

  F is at depth 1 and height 1; the height of the tree is 3. T

  Note:

  - ✓ **The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.**
  - ✓ **If there is a path from n1 to n2, then n1 is an ancestor of n2 and n2 is a descendant of n1. If n1 n2, then n1 is a proper ancestor of n2 and n2 is a proper descendant of n1.**
  - ✓ **A tree there is exactly one path from the root to each node.**

Types of the Tree

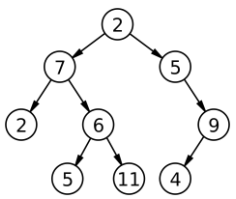
Based on the no. of children for each node in the tree, it is classified into two to types.

1. Binary tree
2. General tree

Binary tree

In a tree, each and every node has a maximum of two children. It can be empty, one or two. Then it is called as Binary tree.

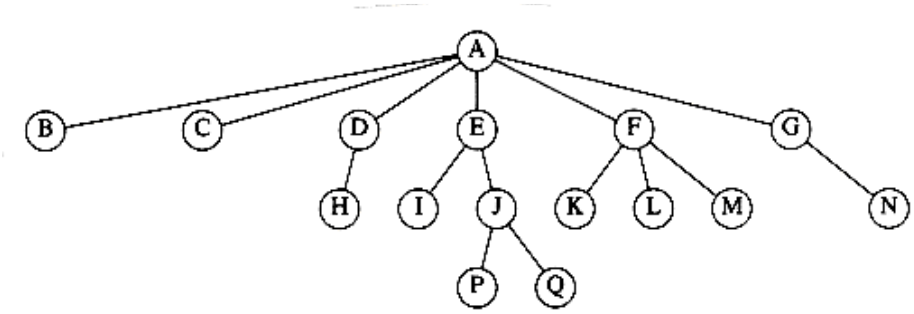
Eg:



General Tree

In a tree, node can have any no of children. Then it is called as general Tree.

Eg:



Implementation of Trees

Tree can be implemented by two methods.

1. Array Implementation
2. Linked List implementation

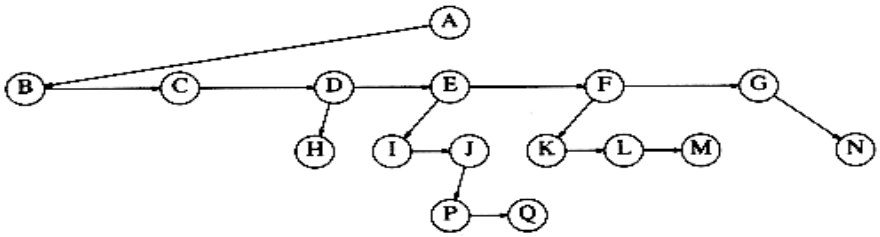
Apart from these two methods, it can also be represented by First Child and Next sibling Representation.

One way to implement a tree would be to have in each node, besides its data, a pointer to each child of the node. However, since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the children direct links in the data structure, because there would be too much wasted space. The solution is simple: Keep the children of each node in a linked list of tree nodes.

Node declarations for trees

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
element_type element;
tree_ptr first_child;
tree_ptr next_sibling;
};
```

First child/next sibling representation of the tree shown in the below Figure



Arrows that point downward are first_child pointers. Arrows that go left to right are next_sibling pointers. Null pointers are not drawn, because there are too many. In the above tree, node E has both a pointer to a sibling (F) and a pointer to a child (I), while some nodes have neither.

Tree Traversals

Visiting of each and every node in a tree exactly only once is called as **Tree traversals.** Here Left subtree and right subtree are traversed recursively.

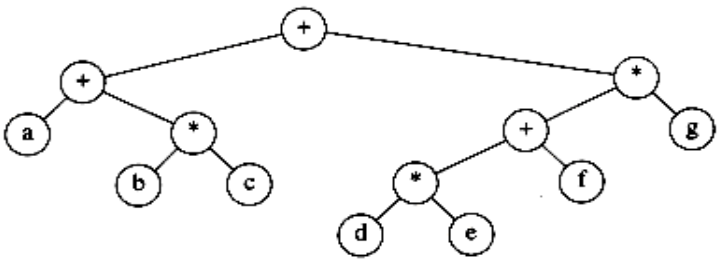
Types of Tree Traversal:

1. **Inorder Traversal**
2. **Preorder Traversal**
3. **Postorder Traversal**

Inorder traversal:

Rules:

- Traverse Left subtree recursively
- Process the node
- Traverse Right subtree recursively



Eg

Inorder traversal: a + b*c + d*e + f*g.

Preorder traversal:

Rules:

- Process the node
- Traverse Left subtree recursively
- Traverse Right subtree recursively

Preorder traversal: ++a*b c*+*d e f g

Postorder traversal:

Rules:

- Traverse Left subtree recursively
- Traverse Right subtree recursively
- Process the node

Postorder traversal: a b c*+de*f + g* +

Tree Traversals with an Application

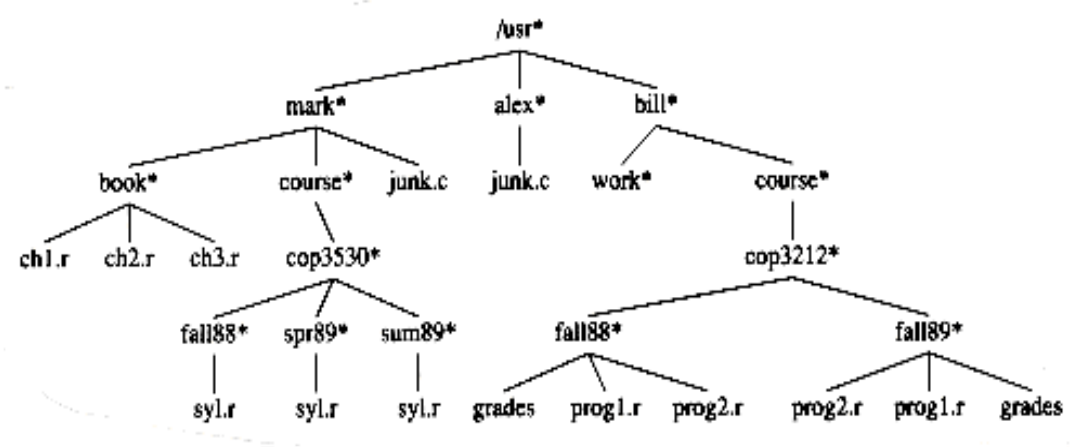
There are many applications for trees. Most important two applications are,

1. Listing a directory in a hierarchical file system
2. Calculating the size of a directory

1. Listing a directory in a hierarchical file system

One of the popular uses is the directory structure in many common operating systems, including UNIX, VAX/VMS, and DOS.

Typical directories in the UNIX file system (UNIX directory)



- ✓ The root of this directory is /usr. (The asterisk next to the name indicates that /usr is itself a directory.)
- ✓ /usr has three children, mark, alex, and bill, which are themselves directories. Thus, /usr contains three directories and no regular files.
- ✓ The filename /usr/mark/book/ch1.r is obtained by following the leftmost child three times. Each / after the first indicates an edge; the result is the full pathname.
- ✓ Two files in different directories can share the same name, because they must have different paths from the root and thus have different pathnames.

- ✓ A directory in the UNIX file system is just a file with a list of all its children, so the directories are structured almost exactly in accordance with the type declaration.
- ✓ Each directory in the UNIX file system also has one entry that points to itself and another entry that point to the parent of the directory. Thus, technically, the UNIX file system is not a tree, but is treelike.

Routine to list a directory in a hierarchical file system void

```
list_directory ( Directory_or_file D )
{
list_dir ( D, 0 );          }
Void list_dir ( Directory_or_file D, unsigned int depth )
{
if ( D is a legitimate entry)
{
        print_name ( depth, D );
        if( D is a directory )
        for each child, c, of D
        list_dir( c, depth+1 );
        }       }
```

The logic of the algorithm is as follow.

- ✓ The argument to list_dir is some sort of pointer into the tree. As long as the pointer is valid, the name implied by the pointer is printed out with the appropriate number of tabs.
- ✓ If the entry is a directory, then we process all children recursively, one by one. These children are one level deeper, and thus need to be indenting an extra space.

This **traversal strategy is known as a preorder traversal**. In a preorder traversal, work at a node is performed before (pre) its children are processed. If there are n file names to be output, then the running time is O (n).

The (preorder) directory listing

```
/usr
        mark
                book
                        chr1.c
                        chr2.c
                        chr3.c
```

course
cop3530
fall88
syl.r
spr89
syl.r
sum89
syl.r
junk.c
alex
junk.c
bill
work
course
cop3212
fall88
grades
prog1.r
prog2.r
fall89
prog1.r
prog2.r
grades

2. Calculating the size of a directory

As above UNIX Directory Structure, the numbers in parentheses representing the number of disk blocks taken up by each file, since the directories are themselves files, they have sizes too. Suppose we would like to calculate the total number of blocks used by all the files in the tree. Here the work at a node is performed after its children are evaluated. So it follows Postorder traversal.

The most natural way to do this would be to find the number of blocks contained in the subdirectories /usr/mark (30), /usr/alex (9), and /usr/bill (32). The total number of blocks is then the total in the subdirectories (71) plus the one block used by /usr, for a total of 72.

Routine to calculate the size of a directory unsigned

```
int size_directory( Directory_or_file D )
{
        unsigned int total_size;
        total_size = 0;
        if( D is a legitimate entry)
        {
        total_size = file_size( D );
        if( D is a directory )
        for each child, c, of D
        total_size += size_directory( c );
}
return( total_size );
}
```

Size of the UNIX Directory

```
                ch1.r                   3
                ch2.r                   2
                ch3.r                   4
        book                            10
                        syl.r           1
                fall88                  2
                        syl.r           5
                spr89                   6
                        syl.r           2
                sum89           3
                cop3530                 12
        course                          13
        junk.c                          6
mark                                    30
        junk.c                          8
alex                                    9
        work                            1
                        grades          3
                        prog1.r         4
                        prog2.r         1
```

		fall88	9
		prog2.r	2
		prog1.r	7
		grades	9
		fall89	19
	cop3212		29
course			30
bill			32
/usr			72

If D is not a directory, then size_directory merely returns the number of blocks used by D. Otherwise, the number of blocks used by D is added to the number of blocks (recursively) found in all of the children.

.

Binary Trees

A binary tree is a tree in which no node can have more than two children.

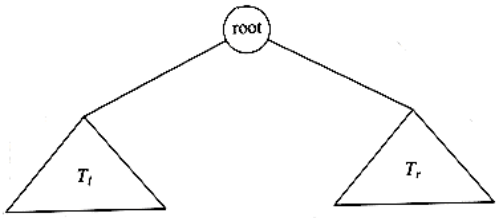
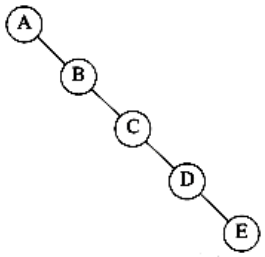


Figure shows that a binary tree consists of a root and two subtrees, Tl and Tr, both of which could possibly be empty.

Worst-case binary tree



Implementation

A binary tree has at most two children; we can keep direct pointers to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the key information plus two pointers (left and right) to other nodes.

Binary tree node declarations

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
element_type element;
tree_ptr left;
tree_ptr right;
};
typedef tree_ptr TREE;
```

Expression Trees

When an expression is represented in a binary tree, then it is called as an **expression Tree**. The leaves of an expression tree are operands, such as constants or variable names, and the other nodes contain operators. It is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the unary minus operator.

We can evaluate an expression tree, T, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.

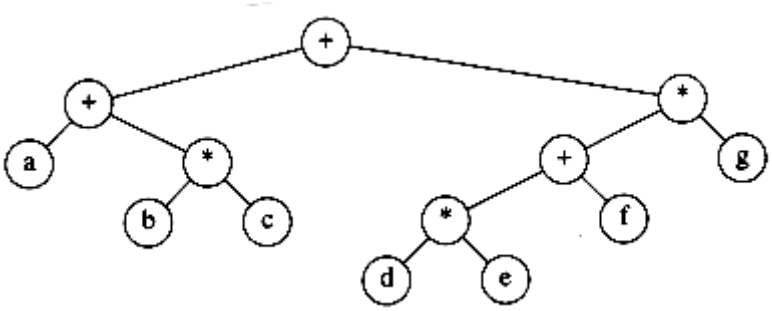
In our example, the left subtree evaluates to a + (b * c) and the right subtree evaluates to ((d *e) + f) *g. The entire tree therefore represents (a + (b*c)) + (((d * e) + f)* g).

We can produce an (overly parenthesized) infix expression by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This general strattegy (left, node, right) is known as an **inorder traversal**; it gives **Infix Expression.**

An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. If we apply this strategy to our tree above, the output is a b c * + d e * f + g * +, which is called as **postfix Expression**. This traversal strategy is generally known as a postorder traversal.

A third traversal strategy is to print out the operator first and then recursively print out the left and right subtrees. The resulting expression, + + a * b c * + * d e f g, is the less useful prefix notation and the traversal strategy is a preorder traversal

Expression tree for (a + b * c) + ((d * e + f) * g)



Constructing an Expression Tree

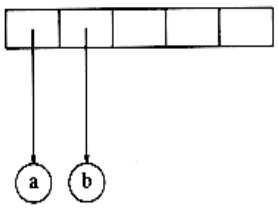
Algorithm to convert a postfix expression into an expression tree

1. Read the postfix expression one symbol at a time.
2. If the symbol is an operand, then
    a. We create a one node tree and push a pointer to it onto a stack.
3. If the symbol is an operator,
    a. We pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively.
4. A pointer to this new tree is then pushed onto the stack.

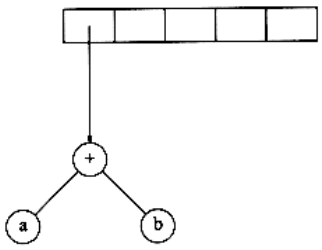
Suppose the input is

**a b + c d e + * *

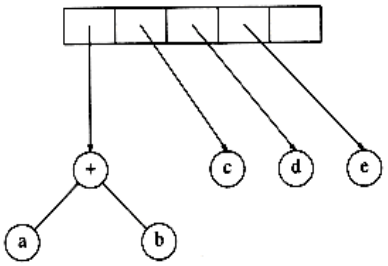
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



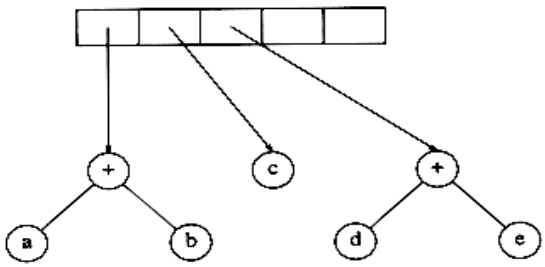
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



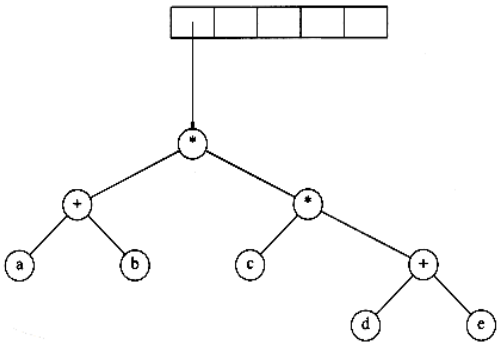
Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Now a '+' is read, so two trees are merged. Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



The Search Tree ADT-<u>Binary Search Tree</u>

The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X, and the values of all the keys in the right subtree are larger than the key value in X.

Notice that this implies that all the elements in the tree can be ordered in some consistent manner.

In the above figure, the tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with key 7 in the left subtree of a node with key 6. The average depth of a binary search tree is O(log n).

Binary search tree declarations

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
element_type element;
tree_ptr left;
tree_ptr right;
};
typedef tree_ptr SEARCH_TREE;
```

Make Empty:

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely.

Find

This operation generally requires returning a pointer to the node in tree T that has key x, or NULL if there is no such node. The structure of the tree makes this simple. If T is , then we can just return . Otherwise, if the key stored at T is x, we can return T. Otherwise, we make a recursive call on a subtree of T, either left or right, depending on the relationship of x to the key stored in T.

Routine to make an empty tree

```
SearchTree makeempty (search tree T)
{
        if(T!=NULL)
        {
        Makeempty (T->left);
        Makeempty (T->Right);
```

```
        Free( T);
        }
        return NULL;  }
```

Routine for Find operation

```
Position find( Elementtype X, SearchTree T )
{
        if( T == NULL )
                return NULL;
        if( x < T->element )
                return( find( x, T->left ) );
        else
        if( x > T->element )
                return( find( x, T->right ) );
        else
                return T;
        }
```

FindMin & FindMax:

These routines return the position of the smallest and largest elements in the tree, respectively.

To perform a findmin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

The findmax routine is the same, except that branching is to the right child.

Recursive implementation of Findmin for binary search trees

```
 Position findmin( SearchTree T )
{
        if( T == NULL )
        return NULL;
        else
                if( T->left == NULL )
                return( T );
        else
                return( findmin ( T->left ) );
        }
```

Recursive implementation of FindMax for binary search trees

```
Position findmax( SearchTree T )
{
        if( T == NULL )
        return NULL;
        else
                if( T->Right == NULL )
                return( T );
        else
                return( findmax( T->right ) );
}
```

Nonrecursive implementation of FindMin for binary search trees

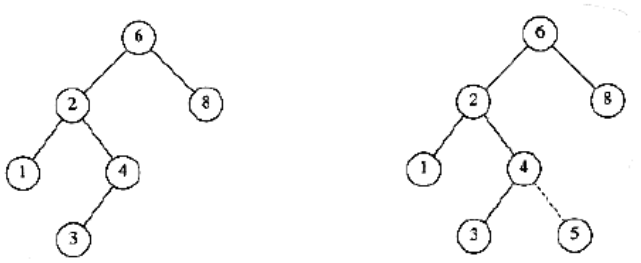
```
Position findmin( SearchTree T )
{
        if( T != NULL )
                while(  T->left != NULL )
                T=T->left;
                return(T);
                }
```

Nonrecursive implementation of FindMax for binary search trees

```
Position findmax( SearchTree T )
{
        if( T != NULL )
                while(  T->right != NULL )
                T=T->right;
                return(T);        }
```

Insert

To insert x into tree T, proceed down the tree. If x is found, do nothing. Otherwise, insert x at the last spot on the path traversed.



To insert 5, we traverse the tree as though a find were occurring. At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Insertion routine

Since T points to the root of the tree, and the root changes on the first insertion, insert is written as a function that returns a pointer to the root of the new tree.

```
searchTree insert( elementtype x, SearchTree T )
{
        if( T == NULL )
        {
        T = (SEARCH_TREE) malloc ( sizeof (struct tree_node) );
        if( T == NULL )
        fatal_error("Out of space!!!");
        else
        {
        T->element = x;
        T->left = T->right = NULL;   }
        }
        else
        if( x < T->element )
        T->left = insert( x, T->left );
        else
        if( x > T->element )
        T->right = insert( x, T->right );
        /* else x is in the tree already. We'll do nothing */
        return T;  }
```

Delete

Once we have found the node to be deleted, we need to consider several possibilities.

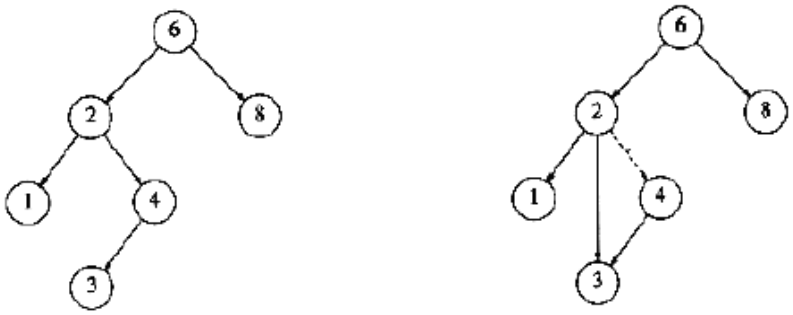
If the node is a leaf, it can be deleted immediately.

If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node

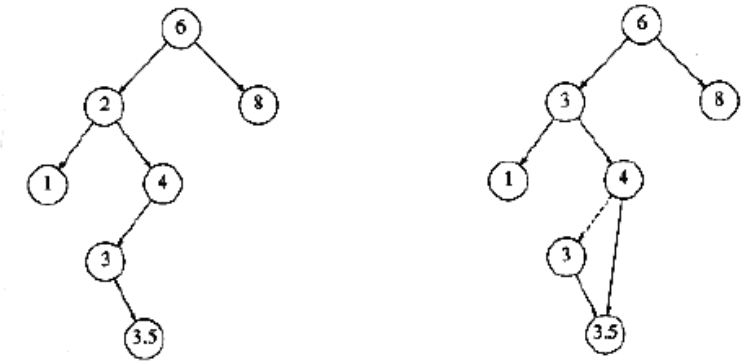
if a node with two children. The general strategy is to replace the key of this node with the smallest key of the right subtree and recursively delete that node. Because the smallest node in the right subtree cannot have a left child, the second
delete is an easy one.

The node to be deleted is the left child of the root; the key value is 2. It is replaced with the smallest key in its right subtree (3), and then that node is deleted as before.

Deletion of a node (4) with one child, before and after



Deletion of a node (2) with two children, before and after



If the number of deletions is expected to be small, then a popular strategy to use is lazy deletion: When an element is to be deleted, it is left in the tree and merely marked as being deleted.

Deletion routine for binary search trees

```
Searchtree delete( elementtype x, searchtree T )
{
Position tmpcell;
if( T == NULL )
error("Element not found");
else
if( x < T->element ) /* Go left */
T->left = delete( x, T->left );
else
if( x > T->element ) /* Go right */
T->right = delete( x, T->right );
else /* Found element to be deleted */
if( T->left && T->right ) /* Two children */
{
tmp_cell = find_min( T->right );
T->element = tmp_cell->element;
T->right = delete( T->element, T->right );
}
else /* One child */
{
tmpcell = T;
if( T->left == NULL ) /* Only a right child */
T= T->right;
if( T->right == NULL ) /* Only a left child */
T = T->left;
free( tmpcell );
}
return T;
}
```

Average-Case Analysis of BST

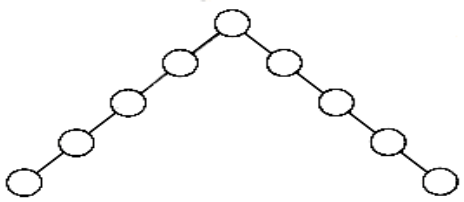
- ✓ All of the operations of the previous section, except makeempty, should take O(log n) time, because in constant time we descend a level in the tree, thus operating on a tree that is now roughly half as large.
- ✓ The running time of all the operations, except makeempty is O(d), where d is the depth of the node containing the accessed key.
- ✓ The average depth over all nodes in a tree is O(log n).
- ✓ The sum of the depths of all nodes in a tree is known as the internal path length.

AVL Trees

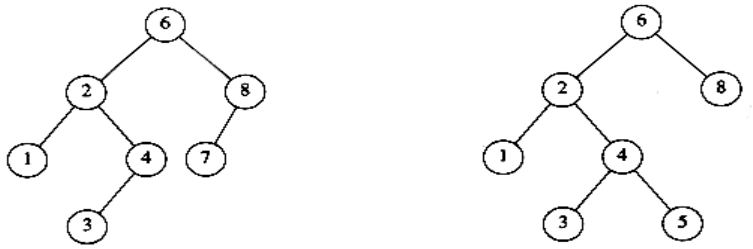
The balance condition and allow the tree to be arbitrarily deep, but after every operation, a restructuring rule is applied that tends to make future operations efficient. These types of data structures are generally classified as **self-adjusting.**

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.)

An AVL (**Adelson-Velskii and Landis**) tree is a binary search tree with a balance condition. The simplest idea is to require that the left and right subtrees have the same height. The balance condition must be easy to maintain, and it ensures that the depth of the tree is O(log n).



The above figure shows, a bad binary tree. Requiring balance at the root is not enough.



In Figure, the tree on the left is an AVL tree, but the tree on the right is not.

Thus, all the tree operations can be performed in O(log n) time, except possibly insertion.

When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is difficult is that inserting a node could violate the AVL tree property.

Inserting a node into the AVL tree would destroy the balance condition.

Let us call the unbalanced node α. Violation due to insertion might occur in four cases:

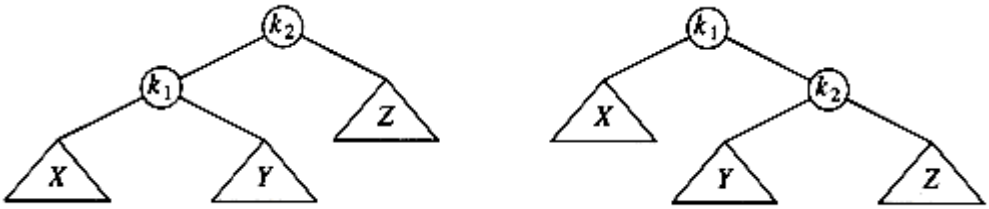
1. **An insertion into the left subtree of the left child of α**
2. **An insertion into the right subtree of the left child of α**
3. **An insertion into the left subtree of the right child of α**
4. **An insertion into the right subtree of the right child of α**

Violation of AVL property due to insertion can be avoided by doing some modification on the node α. This modification process is called as **Rotation**.

Types of rotation

1. Single Rotation
2. Double Rotation

Single Rotation (case 1) – Single rotate with Left

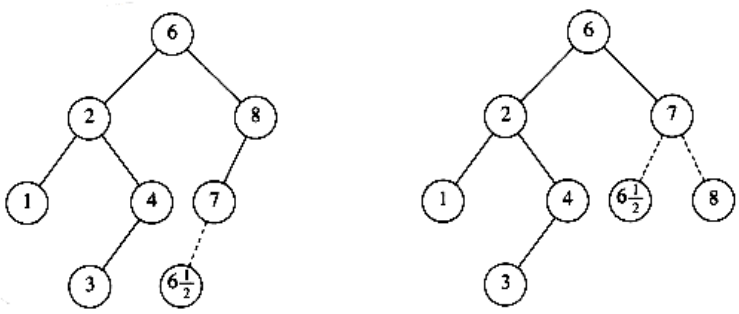


The two trees in the above Figure contain the same elements and are both binary search trees.

First of all, in both trees k1 < k2. Second, all elements in the subtree X are smaller than k1 in both trees. Third, all elements in subtree Z are larger than k2. Finally, all elements in subtree Y are in between k1 and k2. The conversion of one of the above trees to the other is known as **a rotation.**

In an AVL tree, if an insertion causes some node in an AVL tree to lose the balance property: Do a rotation at that node.

The basic algorithm is to start at the node inserted and travel up the tree, updating the balance information at every node on the path.



In the above figure, after the insertion of the in the original AVL tree on the left, node 8 becomes unbalanced. Thus, we do a single rotation between 7 and 8, obtaining the tree on the right.

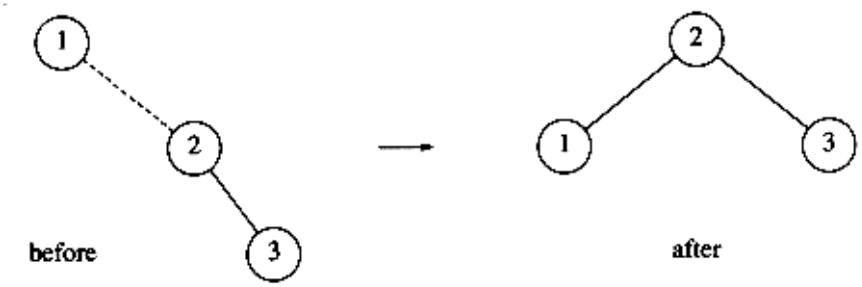
Routine :

```
Static position Singlerotatewithleft( Position K2)
    {
    Position k1;
    K1=k2->left;
    K2->left=k1->right;
    K1->right=k2;
    K2->height=max(height(k2->left),height(k2->right));
    K1->height=max(height(k1->left),k2->height);
    Return k1;
    }
```

Single Rotation (case 4) – Single rotate with Right

(Refer diagram from Class note)

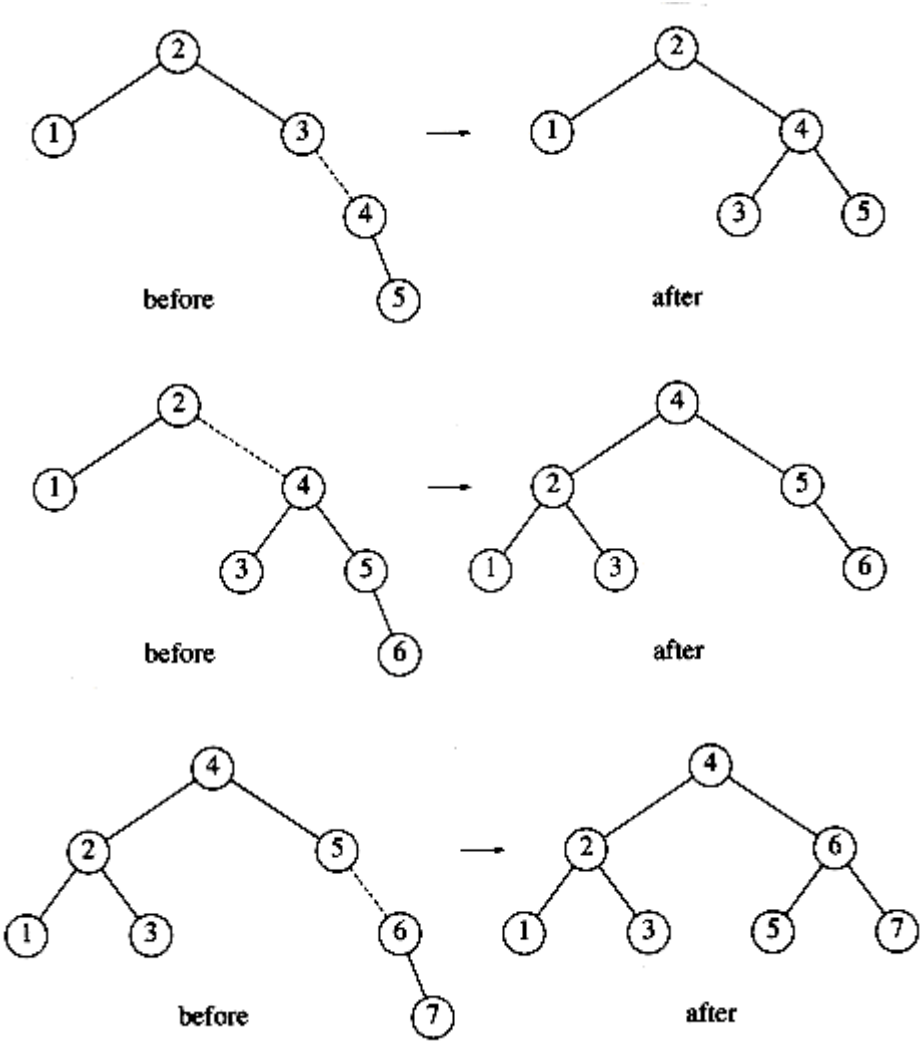
Suppose we start with an initially empty AVL tree and insert the keys 1 through 7 in sequential order. The first problem occurs when it is time to insert key 3, because the AVL property is violated at the root. We perform a single rotation between the root and its right child to fix the problem. The tree is shown in the following figure, before and after the rotation.



A dashed line indicates the two nodes that are the subject of the rotation. Next, we insert the key 4, which causes no problems, but the insertion of 5 creates a violation at node 3, which is fixed by a single rotation.

Next, we insert 6. This causes a balance problem for the root, since its left subtree is of height 0, and its right subtree would be height 2. Therefore, we perform a single rotation at the root between 2 and 4.

The rotation is performed by making 2 a child of 4 and making 4's original left subtree the new right subtree of 2. Every key in this subtree must lie between 2 and 4, so this transformation makes sense. The next key we insert is 7, which causes another rotation.

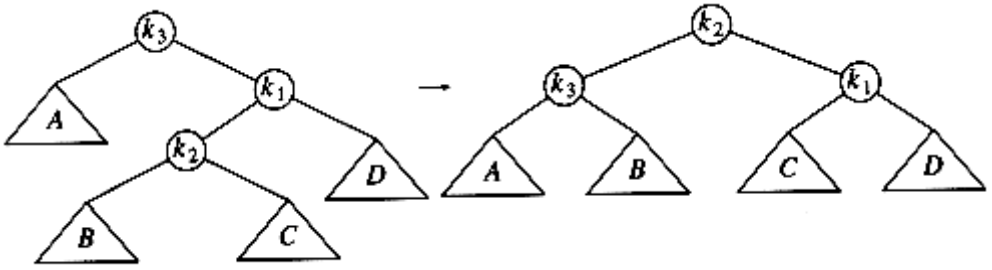


Routine :

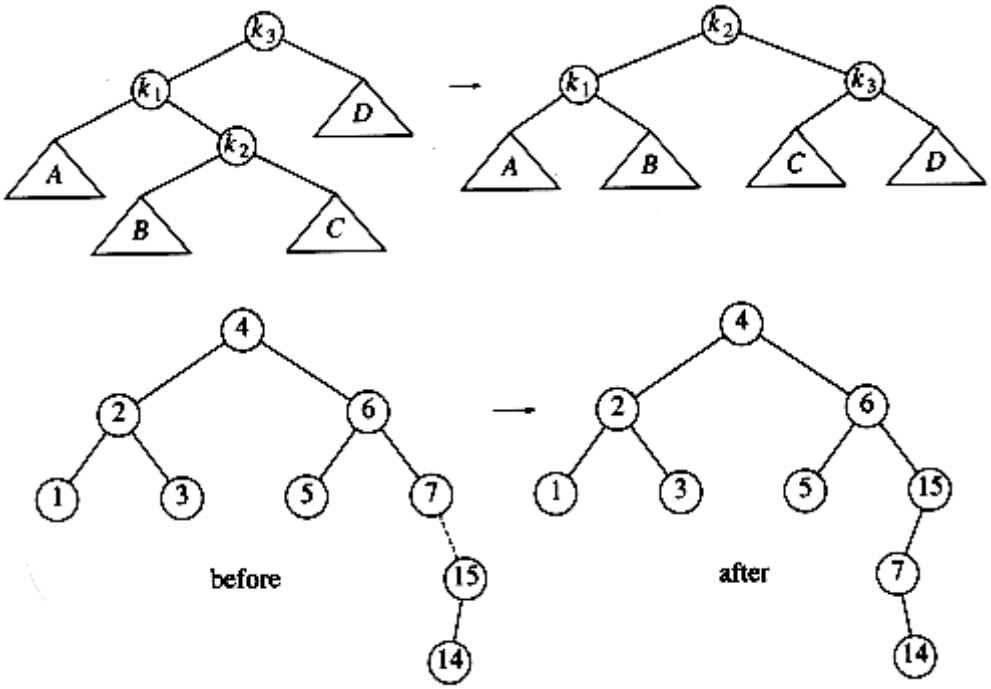
```
Static position Singlerotatewithright( Position K1)
        {
        Position k2;
        K2=k1->right;
        K1->right=k2->left;
        K2->left=k1;
        K1->height=max(height(k1->left),height(k1->right));
        K2->height=max(height(k2->left),k1->height);
        Return k2;
        }
```

Double Rotation

(Right-left) double rotation

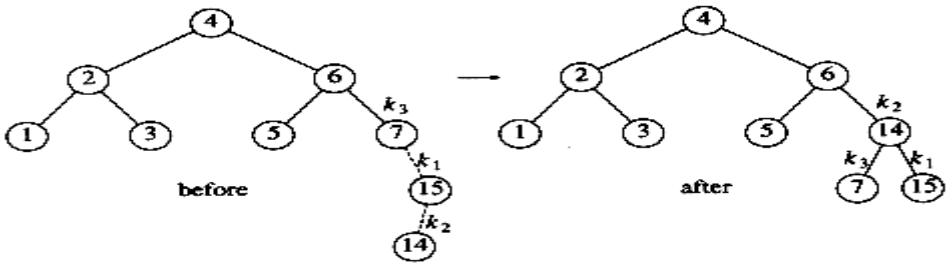


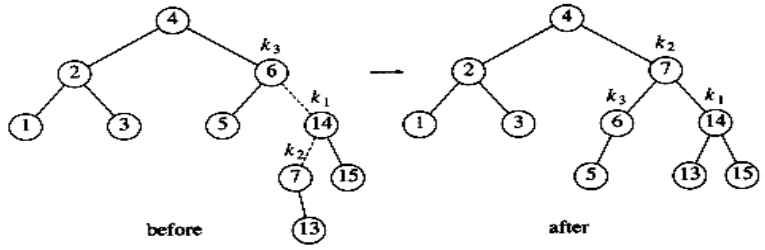
(Left-right) double rotation

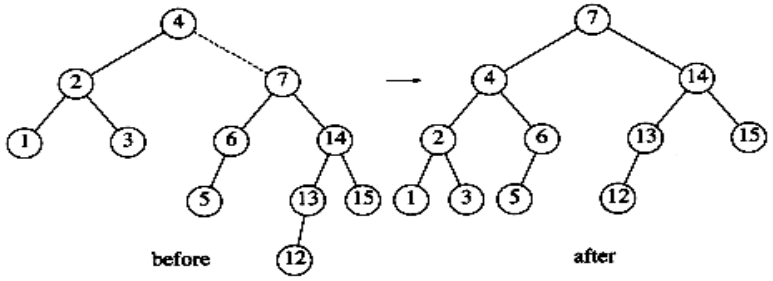


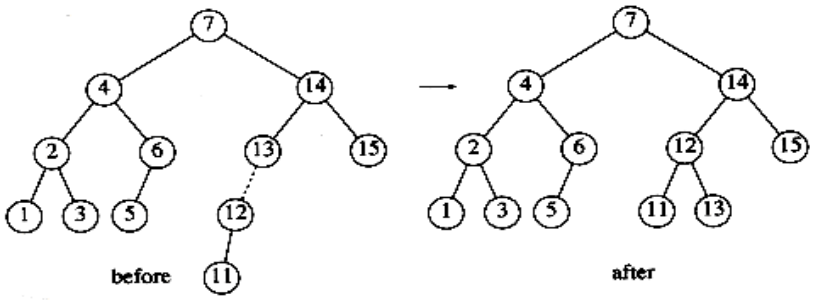
In the above diagram, suppose we insert keys 8 through 15 in reverse order. Inserting 15 is easy, since it does not destroy the balance property, but inserting 14 causes a height imbalance at node 7.

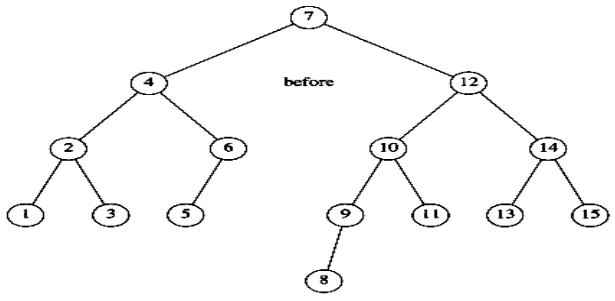
As the diagram shows, the single rotation has not fixed the height imbalance. The problem is that the height imbalance was caused by a node inserted into the tree containing the middle elements (tree Y in Fig. (Right-left) double rotation) at the same time as the other trees had identical heights. This process is called as **double rotation**, which is similar to a single rotation but involves four subtrees instead of three.

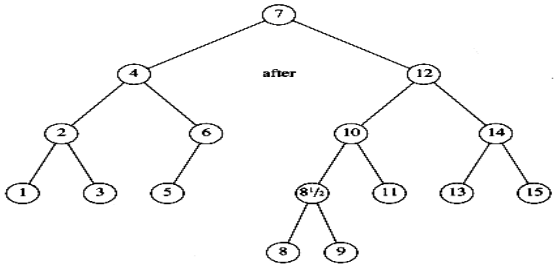

before
after


before
after


before
after


before
after


before


after

In our example, the double rotation is a right-left double rotation and involves 7, 15, and 14. Here, k3 is the node with key 7, k1 is the node with key 15, and k2 is the node with key 14.

Next we insert 13, which require a double rotation. Here the double rotation is again a right-left double rotation that will involve 6, 14, and 7 and will restore the tree. In this case, k3 is the node with key 6, k1 is the node with key 14, and k2 is the node with key 7. Subtree A is the tree rooted at the node with key 5, subtree B is the empty subtree that was originally the left child of the node with key 7, subtree C is the tree rooted at the node with key 13, and finally, subtree D is the tree rooted at the node with key 15.

If 12 is now inserted, there is an imbalance at the root. Since 12 is not between 4 and 7, we know that the single rotation will work. Insertion of 11 will require a single rotation:

To insert 10, a single rotation needs to be performed, and the same is true for the subsequent insertion of 9. We insert 8 without a rotation, creating the almost perfectly balanced tree.

Routine for double Rotation with left (Case 2)

```
Static position doublerotatewithleft(position k3)
    {
    K3->left=singlerotatewithright(k3->left);
    Return singlerotatewithleft(k3);
    }
```

Routine for double Rotation with right (Case 3)

```
Static position doublerotatewithlright(position k1)
    {
    K1->right=singlerotatewithleft(k1->right);
    Return singlerotatewithright(k1);
    }
```

Node declaration for AVL trees:

```
typedef struct avlnode *position;
typedef struct avlnode *avltree;
struct avlnode
{
elementtype element;
avltree left;
avltree right;
int height;
};
typedef avl_ptr SEARCH_TREE;
```

Routine for finding height of an AVL node

```
Int height (avltree p)
{
if( p == NULL )
return -1;
else
return p->height;
}
```

Routine for insertion of new element into a AVL TREE

```
SEARCH_TREE
insert1( element_type x, SEARCH_TREE T, avl_ptr parent )
{
avl_ptr rotated_tree;
if( T == NULL )
{  /* Create and return a one-node tree */
T = (SEARCH_TREE) malloc ( sizeof (struct avl_node) );
```

```
if( T == NULL )
fatal_error("Out of space!!!");
else
{
T->element = x; T->height = 0;
T->left = T->right = NULL;
}
}
else
{
if( x < T->element )
{
T->left = insert1( x, T->left, T );
if( ( height( T->left ) - height( T->right ) ) == 2
{
if( x < T->left->element )
rotated_tree = s_rotate_left( T );
else
rotated_tree = d_rotate_left( T );
if( parent->left == T )
parent->left = rotated_tree;
else
parent->right = rotated_tree;
}
else
T->height = max( height(T->left), height(T->right) ) + 1;


}
else
/* Symmetric Case for right subtree */;
/* Else x is in the tree already. We'll do nothing */
}
return T;
}
```

PRIORITY QUEUES (HEAPS)

A queue is said to be priority queue, in which the elements are dequeued based on the priority of the elements.

A priority queue is used in,

- Jobs sent to a line printer are generally placed on a queue. For instance, one job might be particularly important, so that it might be desirable to allow that job to be run as soon as the printer is available.
- In a multiuser environment, the operating system scheduler must decide which of several processes to run. Generally a process is only allowed to run for a fixed period of time. One algorithm uses a queue. Jobs are initially placed at the end of the queue. The scheduler will repeatedly take the first job on the queue, run it until either it finishes or its time limit is up, and place it at the end of the queue. This strategy is generally not appropriate, because very short jobs will seem to take a long time because of the wait involved to run. Generally, it is important that short jobs finish as fast as possible. This is called as Shortest Job First (SJF). This particular application seems to require a special kind of queue, known as a **priority queue.**

Basic model of a priority queue

A priority queue is a data structure that allows at least the following two operations:

1. Insert, equivalent of enqueue
2. Deletemin, removes the minimum element in the heap equivalent of the

Queue's dequeue operation.

Implementations of Priority Queue

1. Array Implementation
2. Linked list Implementation
3. Binary Search Tree implementation
4. Binary Heap Implementation

Array Implementation

Drawbacks:

1. There will be more wastage of memory due to maximum size of the array should be define in advance
2. Insertion taken at the end of the array which takes O (N) time.
3. Delete_min will also take O (N) times.

Linked list Implementation

It overcomes first two problems in array implementation. But delete_min operation takes O(N) time similar to array implementation.

Binary Search Tree implementation

Another way of implementing priority queues would be to use a binary search tree. This gives an O(log n) average running time for both operations.

Binary Heap Implementation

Another way of implementing priority queues would be to use a binary heap. This gives an O(1) average running time for both operations.

Binary Heap

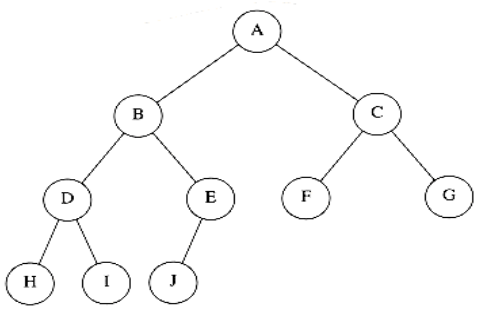
Like binary search trees, heaps have two properties, namely, **a structure property and a heap order property.** As with AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order.

1. **Structure Property**
2. **Heap Order Property**

Structure Property

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a complete binary tree.

A complete Binary Tree

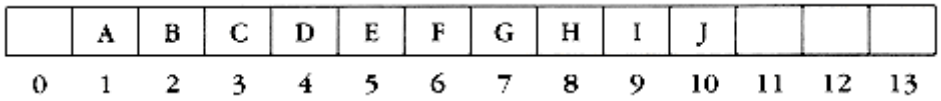


A complete binary tree of height h has between 2h and 2h+1 - 1 nodes. This implies that the height of a complete binary tree is log n, which is clearly O(log n).

Array implementation of complete binary tree

Note:

For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child (2i + 1), and the parent is in position *i*/2 .

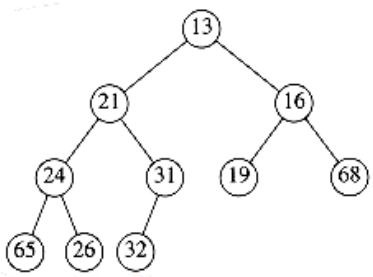


The only problem with this implementation is that an estimate of the maximum heap size is required in advance.

Types of Binary Heap

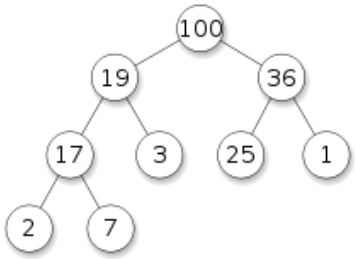
Min Heap

A binary heap is said to be Min heap such that any node x in the heap, the key value of X is smaller than all of its descendants children.



Max Heap

A binary heap is said to be Min heap such that any node x in the heap, the key value of X is larger than all of its descendants children.



It is easy to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node should be smaller than all of its descendants.

Applying this logic, we arrive at the heap order property. In a heap, for every node X, the key in the parent of X is smaller than (or equal to) the key in X.

Similarly we can declare a (max) heap, which enables us to efficiently find and remove the maximum element, by changing the heap order property. Thus, a priority queue can be used to find either a minimum or a maximum.

By the heap order property, the minimum element can always be found at the root.

Declaration for priority queue

```
struct heapstruct
        {
        int capacity;
        int size;
        element_type *elements;
        };

typedef struct heapstruct *priorityQ;
```

Create routine of priority Queue

```
priorityQ create (int max_elements )
{
        priorityQ H;
        if( max_elements < MIN_PQ_SIZE )
        error("Priority queue size is too small");
         H = (priorityQ) malloc ( sizeof (struct heapstruct) );
        if( H == NULL )
        fatal_error("Out of space!!!");
        H->elements = (element_type *) malloc( ( max_elements+1) * sizeof (element_type)
        );
        if( H->elements == NULL )
        fatal_error("Out of space!!!");
        H->capacity= max_elements;
        H->size = 0;
        H->elements[0] = MIN_DATA;
        return H;        }
```

Basic Heap Operations

It is easy to perform the two required operations. All the work involves ensuring that the heap order property is maintained.

1. **Insert**
2. **Deletemin**

Insert

To insert an element x into the heap, we create a hole in the next available location, since otherwise the tree will not be complete.

If x can be placed in the hole without violating heap order, then we do so and are done. Otherwise we slide the element that is in the hole's parent node into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the hole.

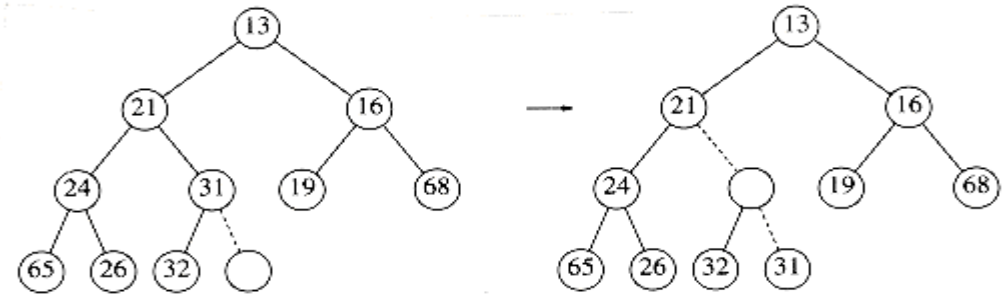
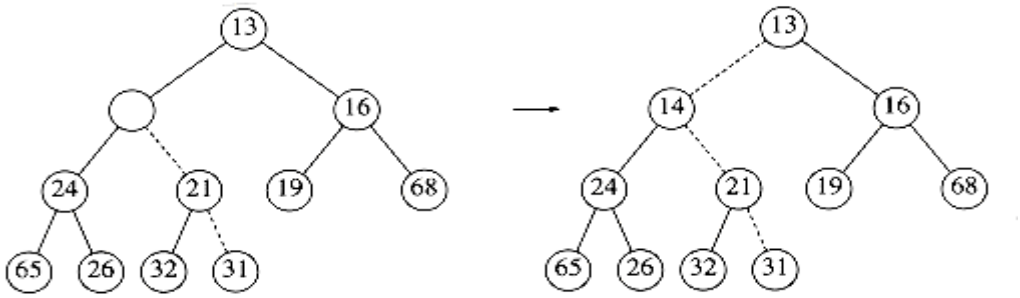


Figure shows that to insert 14, we create a hole in the next available heap location. Inserting 14 in the hole would violate the heap order property, so 31 is slide down into the hole.



This strategy is continued until the correct location for 14 is found. This general strategy is known as a **percolate up**; the new element is percolated up the heap until the correct location is found.

We could have implemented the percolation in the insert routine by performing repeated swaps until the correct order was established, but a swap requires three assignment statements. If an element is percolated up d levels, the number of assignments performed by the swaps would be 3d. Our method uses d + 1 assignments.

Routine to insert into a binary heap

```
/* H->element[0] is a sentinel */
Void insert( element_type x, priorityQ H )
```

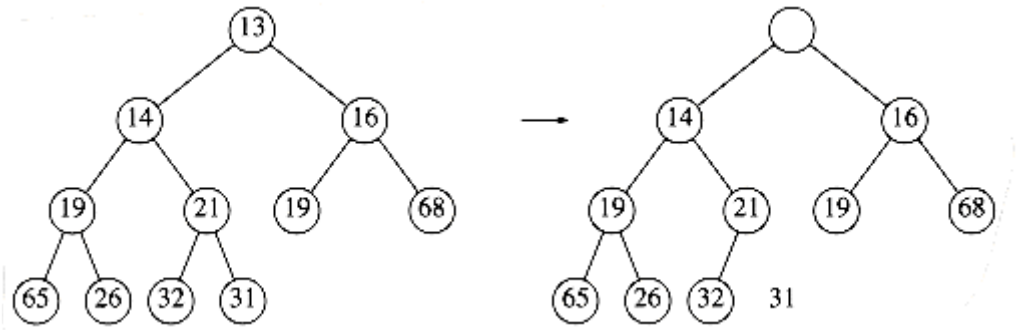
```
{
        int i;
        if( is_full( H ) )
        error("Priority queue is full");
        else
        {
                i = ++H->size;
                while( H->elements[i/2] > x )
                {
                H->elements[i] = H->elements[i/2];
                i /= 2;
        }       H->elements[i] = x;             }       }
```

If the element to be inserted is the new minimum, it will be pushed all the way to the top. The time to do the insertion could be as much as O (log n), if the element to be inserted is the new minimum and is percolated all the way to the root. On

Deletemin

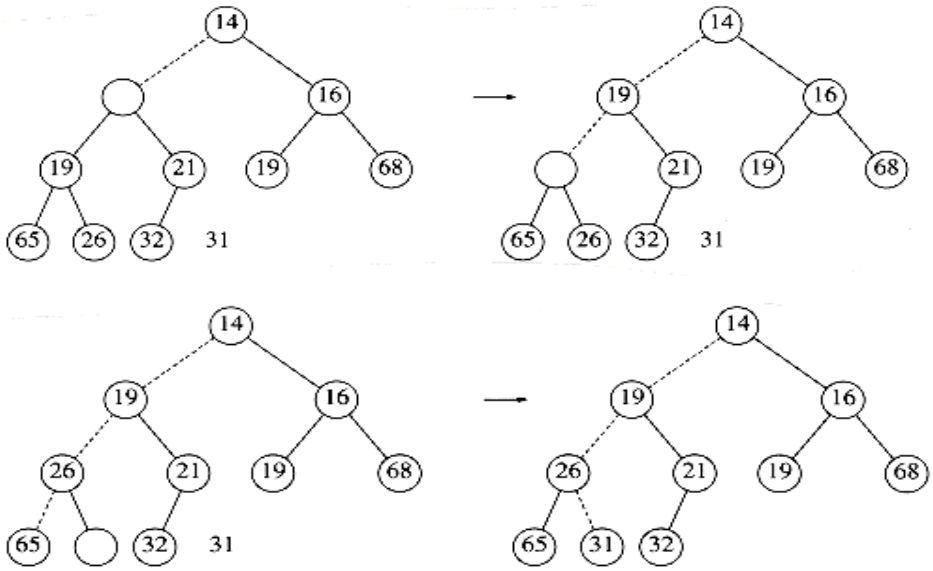
Deletemin are handled in a similar manner as insertions. Finding the minimum is easy; the hard part is removing it.

When the minimum is removed, a hole is created at the root. Since the heap now becomes one smaller, it follows that the last element x in the heap must move somewhere in the heap. If x can be placed in the hole, then we are done. This is unlikely, so we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until x can be placed in the hole. This general strategy is known as a **percolate down.**



In Figure, after 13 is removed, we must now try to place 31 in the heap. 31 cannot be placed in the hole, because this would violate heap order. Thus, we place the smaller child (14) in the hole, sliding the hole down one level. We repeat this again, placing 19 into the

hole and creating a new hole one level deeper. We then place 26 in the hole and create a new hole on the bottom level. Finally, we are able to place 31 in the hole.



Routine to perform deletemin in a binary heap

element_type delete_min(priorityQ H)

```
{
        int i, child;
        element_type min_element, last_element;
        if( is_empty( H ) )
        {
        error("Priority queue is empty");
        return H->elements[0];
        }
        min_element = H->elements[1];
        last_element = H->elements[H->size--];
        for( i=1; i*2 <= H->size; i=child )
        {
        child = i*2;
        if( ( child != H->size ) && ( H->elements[child+1] < H->elements [child] ) )
        child++;
        if( last_element > H->elements[child] )
        H->elements[i] = H->elements[child];
        else
        break;
```

```
    }
    H->elements[i] = last_element;
    return min_element;
}
```

The worst-case running time for this operation is O(log n). On average, the element that is placed at the root is percolated almost to the bottom of the heap, so the average running time is O (log n).

Other Heap Operations

The other heap operations are

1. Decreasekey
2. Increasekey
3. Delete
4. Buildheap

Decreasekey

The decreasekey(x, Δ, H) operation lowers the value of the key at position x by a positive amount Δ. Since this might violate the heap order, it must be fixed by a percolate up.

USE:

This operation could be useful to system administrators: they can make their programs run with highest priority.

Increasekey

The increasekey(x, Δ, H) operation increases the value of the key at position x by a positive amount Δ. This is done with a percolate down.

USE:

Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

Delete

The delete(x, H) operation removes the node at position x from the heap. This is done by first performing decreasekey(x,Δ , H) and then performing deletemin(H). When a process is terminated by a user, it must be removed from the priority queue.

Buildheap

The buildheap(H) operation takes as input n keys and places them into an empty heap. This can be done with n successive inserts. Since each insert will take O(1) average and O(log n) worst-case time, the total running time of this algorithm would be O(n) average but O(n log n) worst-case.

UNIT IV MULTIWAY SEARCH TREES AND GRAPHS 9

B-Tree – B+ Tree – Graph Definition – Representation of Graphs – Types of Graph - Breadth-first traversal – Depth-first traversal — Bi-connectivity – Euler circuits – Topological Sort – Dijkstra's algorithm – Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm

---

B-Trees

AVL tree and Splay tree are binary; there is a popular search tree that is not binary. This tree is known as a B-tree.

A B-tree of order m is a tree with the following structural properties:

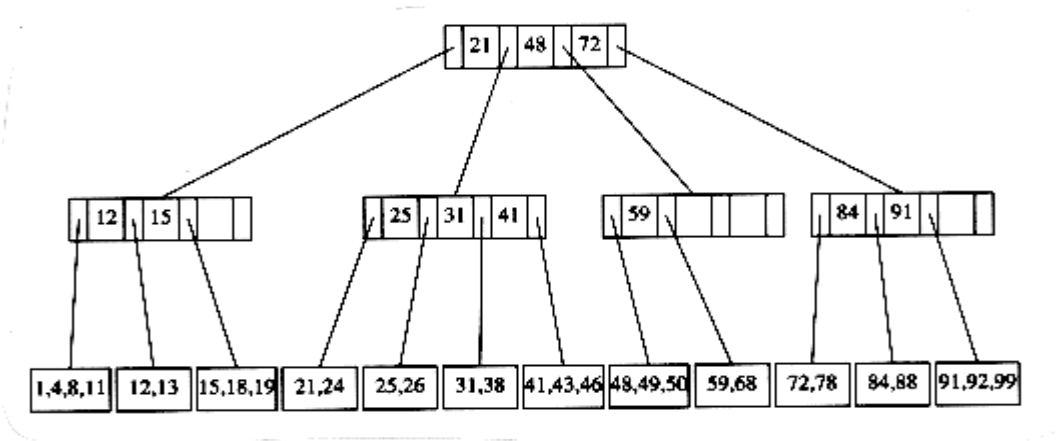
a. The root is either a leaf or has between 2 and m children.
b. All nonleaf nodes (except the root) have between m/2 and m children.
c. All leaves are at the same depth.

All data is stored at the leaves. Contained in each interior node are pointers p1, p2, . . . , pm to the children, and values k1, k2, . . . , km - 1, representing the smallest key found in the subtrees p2, p3, . . . , pm respectively. Some of these pointers might be NULL, and the corresponding ki would then be undefined.

For every node, all the keys in subtree p1 are smaller than the keys in subtree p2, and so on. The leaves contain all the actual data, which is either the keys themselves or pointers to records containing the keys.

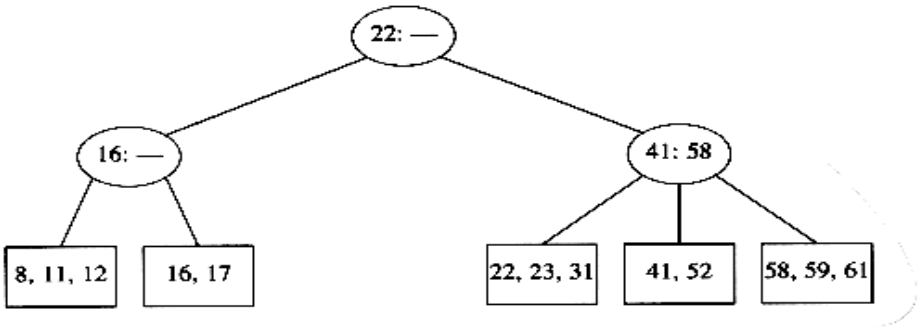
The number of keys in a leaf is also between *m*/2 and m.

An example of a B-tree of order 4



A B-tree of order 4 is more popularly known as a 2-3-4 tree, and a B-tree of order 3 is known as a 2-3 tree

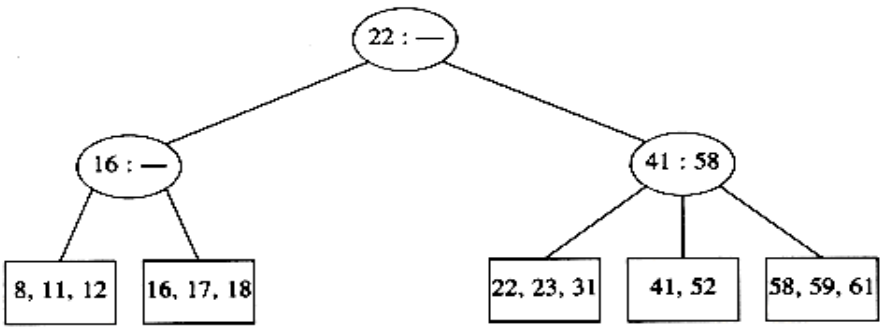
Our starting point is the 2-3 tree that follows.



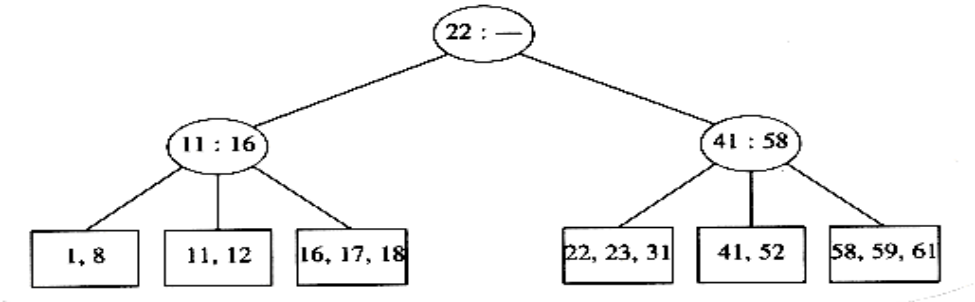
We have drawn interior nodes (nonleaves) in ellipses, which contain the two pieces of data for each node. A dash line as a second piece of information in an interior node indicates that the node has only two children. Leaves are drawn in boxes, which contain the keys. The keys in the leaves are ordered.

To perform a find, we start at the root and branch in one of (at most) three directions, depending on the relation of the key we are looking for to the two values stored at the node.

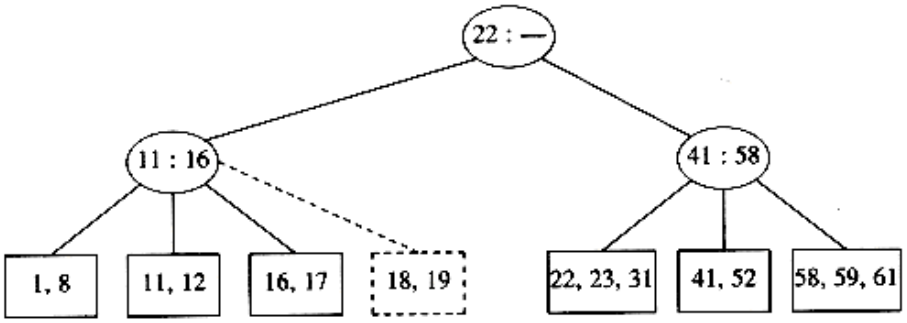
When we get to a leaf node, we have found the correct place to put x. Thus, to insert a node with key 18, we can just add it to a leaf without causing any violations of the 2-3 tree properties. The result is shown in the following figure.



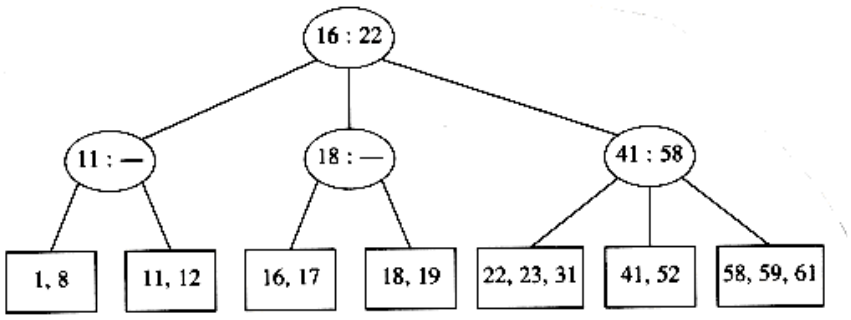
If we now try to insert 1 into the tree, we find that the node where it belongs is already full. Placing our new key into this node would give it a fourth element which is not allowed. This can be solved by making two nodes of two keys each and adjusting the information in the parent.



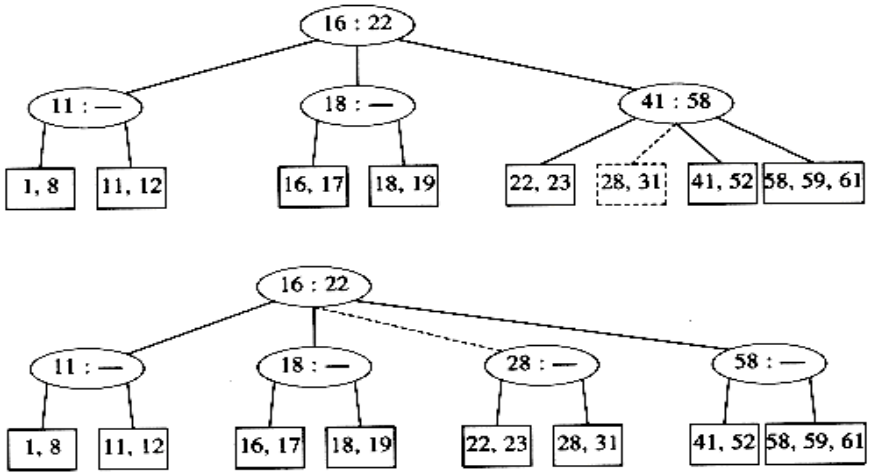
To insert 19 into the current tree, two nodes of two keys each, we obtain the following tree.



This tree has an internal node with four children, but we only allow three per node. Again split this node into two nodes with two children. Now this node might be one of three children itself, and thus splitting it would create a problem for its parent but we can keep on splitting nodes on the way up to the root until we either get to the root or find a node with only two children.



If we now insert an element with key 28, we create a leaf with four children, which is split into two leaves of two children.



This creates an internal node with four children, which is then split into two children. Like to insert 70 into the tree above, we could move 58 to the leaf containing 41 and 52, place 70 with 59 and 61, and adjust the entries in the internal nodes.

Deletion in B-Tree

- If this key was one of only two keys in a node, then its removal leaves only one key. We can fix this by combining this node with a sibling. If the sibling has three keys, we can steal one and have both nodes with two keys.
- If the sibling has only two keys, we combine the two nodes into a single node with three keys. The parent of this node now loses a child, so we might have to percolate this strategy all the way to the top.
- If the root loses its second child, then the root is also deleted and the tree becomes one level shallower.

We repeat this until we find a parent with less than m children. If we split the root, we create a new root with two children.

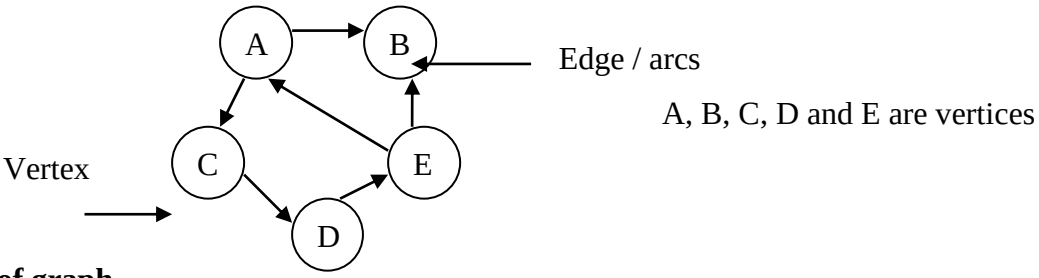
The depth of a B-tree is at most log m/2 n.

The worst-case running time for each of the insert and delete operations is thus **O(m logm n) = O((m / log m) log n), but a find takes only O(log n).**

Definitions

Graph

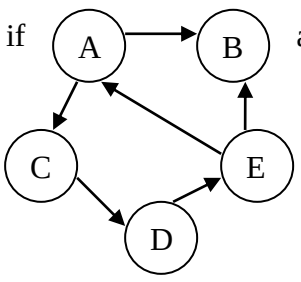
A graph G = (V, E) consists of a set of vertices, V, and a set of edges, E. Each edge is a pair (v,w), where v,w € V. Edges are sometimes referred to as **arcs**.



Types of graph

1. Directed Graph

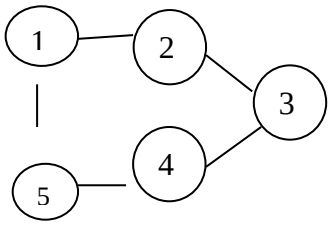
If the pair is ordered, then the graph is directed. In a graph, if all the edges are directionally oriented, then the graph is called as **directed Graph.** Directed graphs are sometimes referred to as **digraphs**.

Vertex w is adjacent to v if 

and only if (v, w) has an edge E.

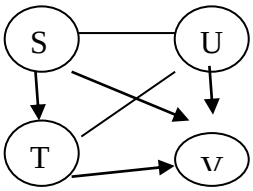
2. Undirected Graph

In a graph, if all the edges are not directionally oriented, then the graph is called as **undirected Graph.** In an undirected graph with edge (v,w), and hence (w,v), w is adjacent to v and v is adjacent to w.



3. Mixed Graph

In a graph if the edges are either directionally or not directionally oriented, then it is called as mixed graph.



Path

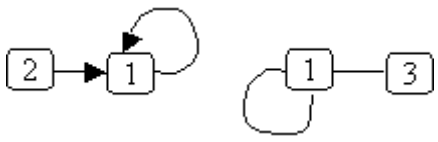
A path in a graph is a sequence of vertices w1, w2, w3, . . . , wn such that (wi, wi+i) € E for 1< i < n.

Path length

The length of a path is the number of edges on the path, which is equal to n – 1 where n is the no of vertices.

Loop

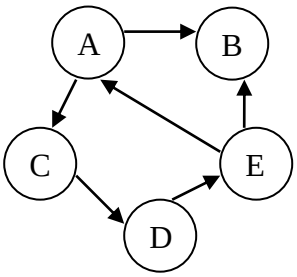
A path from a vertex to itself; if this path contains no edges, then the path length is 0. If the graph contains an edge (v,v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop**.



Simple Path

A simple path is a path such that all vertices are distinct, except that the first and last could be the same.

A->C->D->E



Cycle

In a graph, if the path starts and ends to the same vertex then it is known as **Cycle.**

A->C->D->E->A

Cyclic Graph

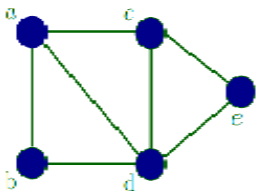
A directed graph is said to be cyclic graph, if it has cyclic path.

Acyclic Graph

A directed graph is acyclic if it has no cycles. A directed acyclic graph is also referred as **DAG.**

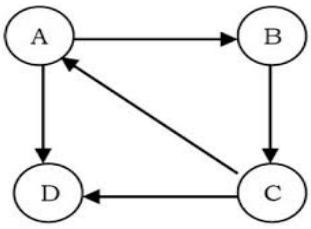
Connected Graph

An undirected graph is connected if there is a path from every vertex to every other vertex.



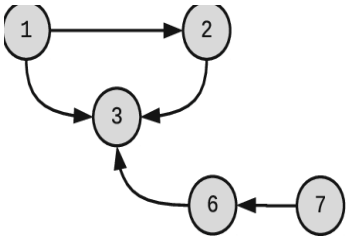
Strongly connected Graph

A directed graph is called strongly connected if there is a path from every vertex to every other vertex.



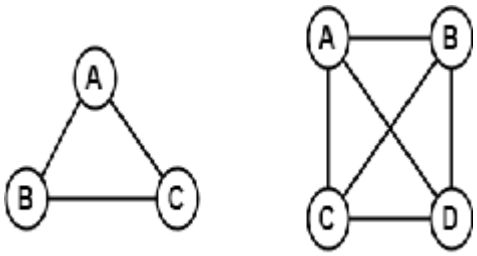
Weakly connected Graph

If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be weakly connected.



Complete graph

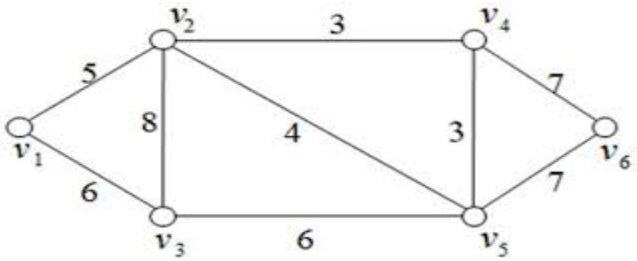
A complete graph is a graph in which there is an edge between every pair of vertices.



Weighted Graph

In a directed graph, if some positive non zero integer values are assigned to each and every edges, then it is known as **weighted graph**. Also called as **Network**

An example of a real-life situation that can be modeled by a graph is the airport system. Each airport is a vertex, and two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices. The edge could have a weight, representing the time, distance, or cost of the flight.



Indegree and Outdegree

Indegree : number of edges entering or coming towards a vertex is called Indegree.

Outdegree: Number of edges exiting or going out from a vertex is called Outdegree.

Degree : Number of edges incident on a vertex is called Degree of a vertex.

Degree = Indegree + Outdegree

Source / Start Vertex: A vertex whose indegree is zero is called sink vertex

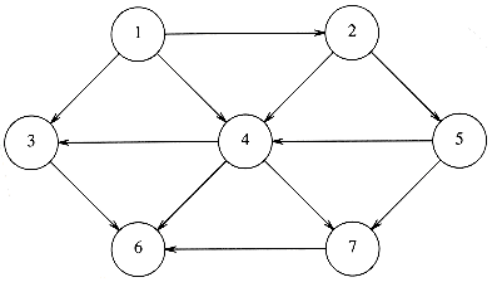
Sink / Destination Vertex : A vertex whose outdegree is zero is called sink vertex

Representation of Graphs

1. Adjacency matrix / Incidence Matrix
2. Adjacency Linked List/ Incidence Linked List

Adjacency matrix

We will consider directed graphs. (Fig. 1)



Now we can number the vertices, starting at 1. The graph shown in above figure represents 7 vertices and 12 edges.

One simple way to represent a graph is to use a two-dimensional array. This is known as an adjacency matrix representation.

For each edge (u, v), we set a[u][v]= 1; otherwise the entry in the array is 0. If the edge has a weight associated with it, then we can set a[u][v] equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.

Advantage is, it is extremely simple, and the space requirement is $(|V|^2)$.

For directed graph

A[u][v]= { 1, if there is edge from u to v

0 otherwise }

For undirected graph

A[u][v]= { 1, if there is edge between u and v

0 otherwise }

For weighted graph

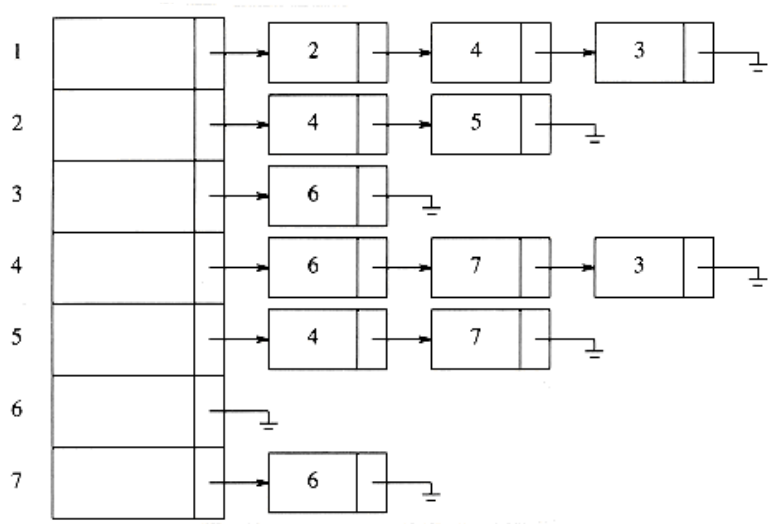
A[u][v]= { value , if there is edge from u to v

∞, if no edge between u and v }

Adjacency lists

Adjacency lists are the standard way to represent graphs. Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially doubles. A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v, and this can be done, in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list.

An adjacency list representation of a graph (See above fig 5.1)

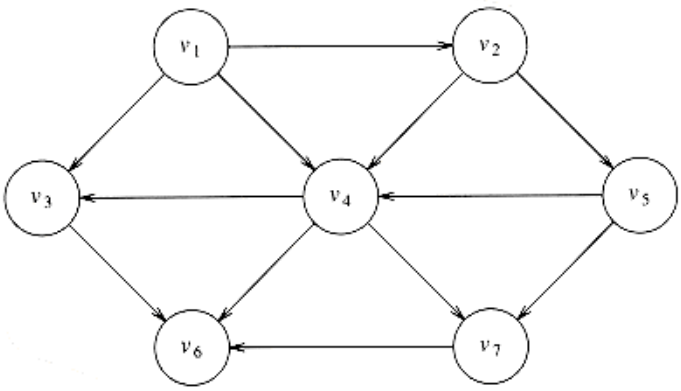


Topological Sort

A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from vi to vj, then vj appears after vi in the ordering.

It is clear that a topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

Directed acyclic graph



In the above graph v1, v2, v5, v4, v3, v7, v6 and v1, v2, v5, v4, v7, v3, v6 are both topological orderings.

A simple algorithm to find a topological ordering

First, find any vertex with no incoming edges (Source vertex). We can then print this vertex, and remove it, along with its edges, from the graph.

To formalize this, we define the indegree of a vertex v as the number of edges (u,v). We compute the indegrees of all vertices in the graph. Assuming that the indegree array is initialized and that the graph is read into an adjacency list,

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
Enqueue	v1	v2	v5	v4	v3	v7	v6
Dequeue	v1	v2	v5	v4	v3	v7	v6

Simple Topological Ordering Routine

```
Void   topsort( graph G )
{
unsigned int counter;
vertex v, w;
for( counter = 0; counter < NUM_VERTEX; counter++ )
{
v = find_new_vertex_of_indegree_zero( );
if( v = NOT_A_VERTEX )
{
error("Graph has a cycle");
break;
}
```

```
top_num[v] = counter;
for each w adjacent to v
indegree[w]--;
}
}
```

Explanation

The function find_new_vertex_of_indegree_zero scans the indegree array looking for a vertex with indegree 0 that has not already been assigned a topological number. It returns NOT_A_VERTEX if no such vertex exists; this indicates that the graph has a cycle.

Routine to perform Topological Sort

```
Void topsort( graph G )
{
QUEUE Q;
unsigned int counter;
vertex v, w;
Q = create_queue( NUM_VERTEX );
makeempty( Q );
counter = 0;
for each vertex v
if( indegree[v] = 0 )
enqueue( v, Q );

while( !isempty( Q ) )
{
v = dequeue( Q );
top_num[v] = ++counter; /* assign next number */
for each w adjacent to v
if( --indegree[w] = 0 )
enqueue( w, Q );
}
if( counter != NUMVERTEX )
error("Graph has a cycle");
dispose_queue( Q ); /* free the memory */
```

}

Graph Traversal:

Visiting of each and every vertex in the graph only once is called as **Graph traversal.** There are two types of Graph traversal.

1. Depth First Traversal/ Search (DFS)
2. Breadth First Traversal/ Search (BFS)

Depth First Traversal/ Search (DFS)

Depth-first search is a generalization of preorder traversal. Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v. If this process is performed on a tree, then all tree vertices are systematically visited in a total of O(|E|) time, since |E| = (|V|).

We need to be careful to avoid cycles. To do this, when we visit a vertex v, we mark it visited, since now we have been there, and recursively call depth-first search on all adjacent vertices that are not already marked.

The two important key points of depth first search

1. If path exists from one node to another node walk across the edge – **exploring the edge**
2. If path does not exist from one specific node to any other nodes, return to the previous node where we have been before – **backtracking**

Procedure for DFS

Starting at some vertex V, we process V and then recursively traverse all the vertices adjacent to V. This process continues until all the vertices are processed. If some vertex is not processed recursively, then it will be processed by using backtracking. If vertex W is visited from V, then the vertices are connected by means of tree edges. If the edges not included in tree, then they are represented by back edges. At the end of this process, it will construct a tree called as DFS tree.

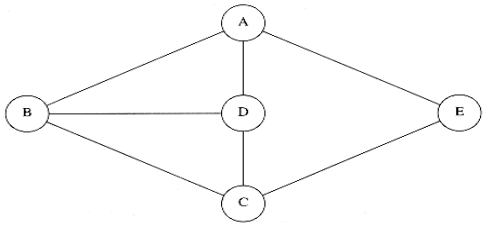
Routine to perform a depth-first search void

```
void dfs( vertex v )
{
visited[v] = TRUE;
for each w adjacent to v
if( !visited[w] )
dfs( w ); }
```

The (global) boolean array visited[ ] is initialized to FALSE. By recursively calling the procedures only on nodes that have not been visited, we guarantee that we do not loop indefinitely.

* An efficient way of implementing this is to begin the depth-first search at v1. If we need to restart the depth-first search, we examine the sequence vk, vk + 1, . . . for an unmarked vertex,where vk - 1 is the vertex where the last depth-first search was started.

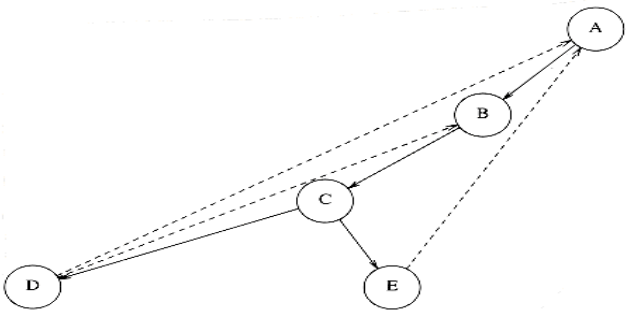
An undirected graph



Steps to construct depth-first spanning tree

a. We start at vertex A. Then we mark A as visited and call dfs(B) recursively. dfs(B) marks B as visited and calls dfs(C) recursively.
b. dfs(C) marks C as visited and calls dfs(D) recursively.
c. dfs(D) sees both A and B, but both these are marked, so no recursive calls are made. dfs(D) also sees that C is adjacent but marked, so no recursive call is made there, and dfs(D) returns back to dfs(C).
d. dfs(C) sees B adjacent, ignores it, finds a previously unseen vertex E adjacent, and thus calls dfs(E).
e. dfs(E) marks E, ignores A and C, and returns to dfs(C).
f. dfs(C) returns to dfs(B). dfs(B) ignores both A and D and returns.
g. dfs(A) ignores both D and E and returns.

Depth-first search of the graph



---------> Back edge

Tree edge

The root of the tree is A, the first vertex visited. Each edge (v, w) in the graph is present in the tree. If, when we process (v, w), we find that w is unmarked, or if, when we process (w, v), we find that v is unmarked, we indicate this with a **tree edge.**

If when we process (v, w), we find that w is already marked, and when processing (w, v), we find that v is already marked, we draw a dashed line, which we will call a **back edge**, to indicate that this "edge" is not really part of the tree.

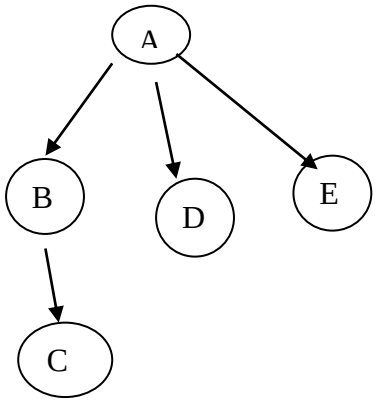
Breadth First Traversal (BFS)

Here starting from some vertex v, and its adjacency vertices are processed. After all the adjacency vertices are processed, then selecting any one the adjacency vertex and process will continue. If the vertex is not visited, then backtracking is applied to visit the unvisited vertex.

Routine:

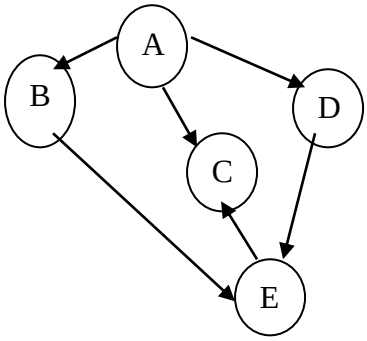
```
void  BFS (vertex v)
{
visited[v]= true;
For each w adjacent to v
If (!visited[w])
visited[w] = true;
}
```

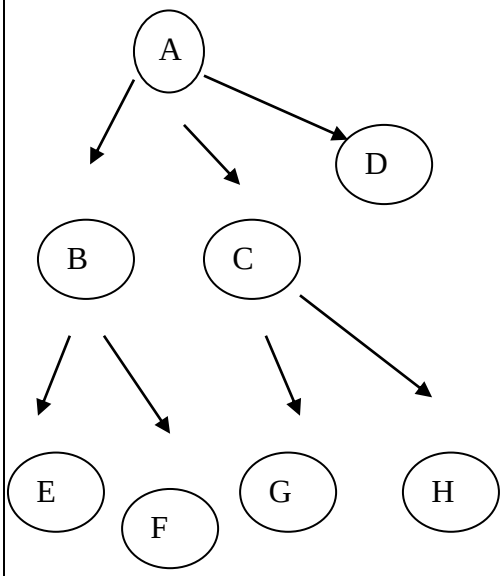
Example: BFS of the above graph



Difference between DFS & BFS

S. No	DFS	BFS
1	Back tracking is possible from a dead end.	Back tracking is not possible.
2	Vertices from which exploration is incomplete are processed in a LIFO order.	The vertices to be explored are organized as a FIFO queue.
3	Search is done in one particular direction at the time.	The vertices in the same level are maintained parallel. (Left to right) (alphabetical ordering)

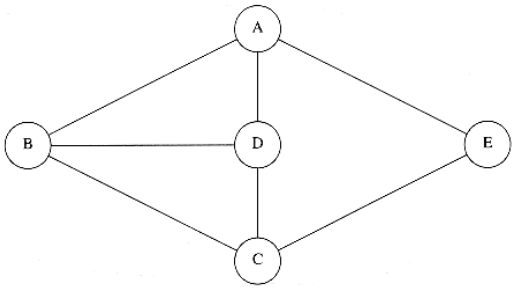
<table>
<tr>
<td>4</td>
<td> 

Order of traversal:
A→B→C→D→E</td>
<td> 

Order of traversal:
A→B→C→D→E→F→G→H</td>
</tr>
</table>

Bi-connectivity / Bi connected Graph:

A connected undirected graph is biconnected if there are no vertices whose removal disconnects the rest of the graph.

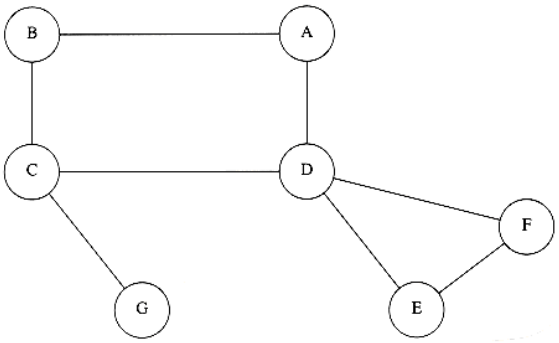


The graph in the example above is biconnected.

It is used in computer networks. If the nodes are computers and the edges are links, then if any computer goes down, network mail is unaffected if it is a biconnected network.

Articulation points

If a graph is not biconnected, the vertices whose removal would disconnect the graph are known as articulation points.



The above graph is not biconnected: C and D are articulation points.

The removal of C would disconnect G, and the removal of D would disconnect E and F, from the rest of the graph.

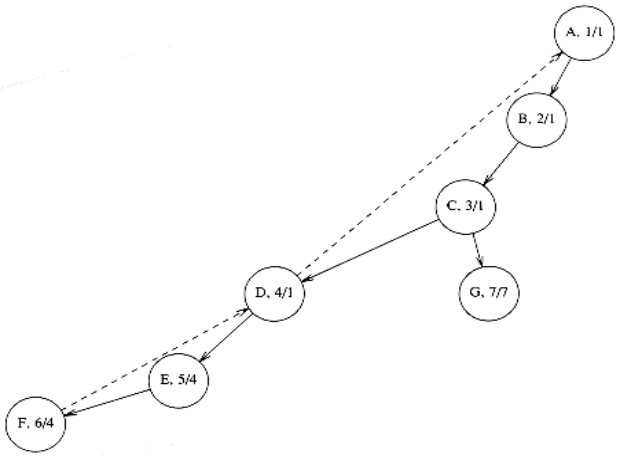
Depth-first search provides a linear-time algorithm to find all articulation points in a connected graph.

- First, starting at any vertex, we perform a depth-first search and number the nodes as they are visited.
- For each vertex v, we call this preorder number num (v). Then, for every vertex v in the depth-first search spanning tree, we compute the lowest-numbered vertex, which we call low(v), that is reachable from v by taking zero or more tree edges and then possibly one back edge (in that order).

By the definition of low, low (v) is the minimum of

1. num(v)
2. the lowest num(w) among all back edges (v, w)
3. the lowest low(w) among all tree edges (v, w)

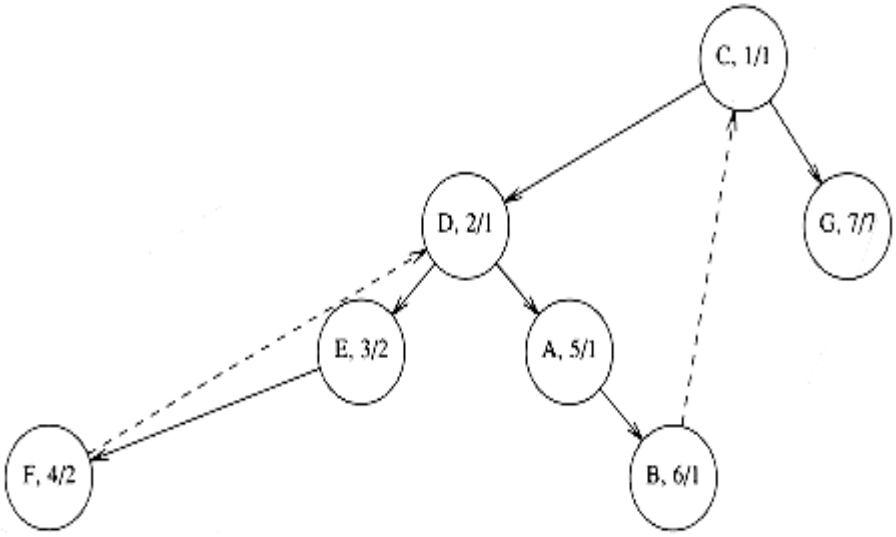
The first condition is the option of taking no edges, the second way is to choose no tree edges and a back edge, and the third way is to choose some tree edges and possibly a back edge.



The depth-first search tree in the above Figure shows the preorder number first, and then the lowest-numbered vertex reachable under the rule described above.

The lowest-numbered vertex reachable by A, B, and C is vertex 1 (A), because they can all take tree edges to D and then one back edge back to A and find low value for all other vertices.

Depth-first tree that results if depth-first search starts at C



To find articulation points,

- The root is an articulation point if and only if it has more than one child, because if it has two children, removing the root disconnects nodes in different subtrees, and if it has only one child, removing the root merely disconnects the root.
- Any other vertex v is an articulation point if and only if v has some child w such that low (w)>= num (v). Notice that this condition is always satisfied at the root;

We examine the articulation points that the algorithm determines, namely C and D. D has a child E, and low (E)>= num (D), since both are 4. Thus, there is only one way for E to get to any node above D, and that is by going through D.

Similarly, C is an articulation point, because low (G)>= num (C).

Routine to assign num to vertices

```
Void assignnum( vertex v )
{
vertex w;
num[v] = counter++;
visited[v] = TRUE;
```

```
for each w adjacent to v
if( !visited[w] )
{
parent[w] = v;
assignnum ( w );        }        }
```

Routine to compute low and to test for articulation

```
Void assignlow( vertex v )
{
vertex w;
low[v] = num[v]; /* Rule 1 */
for each w adjacent to v
{
if( num[w] > num[v] ) /* forward edge */
{
assignlow( w );
if( low[w] >= num[v] )
printf( "%v is an articulation point\n", v );
low[v] = min( low[v], low[w] ); /* Rule 3 */
}
else
if( parent[v] != w ) /* back edge */
low[v] = min( low[v], num[w] ); /* Rule 2 */          }        }
```

Testing for articulation points in one depth-first search (test for the root is omitted) void

```
findart( vertex v )
{
vertex w;
visited[v] = TRUE;
low[v] = num[v] = counter++; /* Rule 1 */
for each w adjacent to v
{
if( !visited[w] ) /* forward edge */
{
parent[w] = v;
```

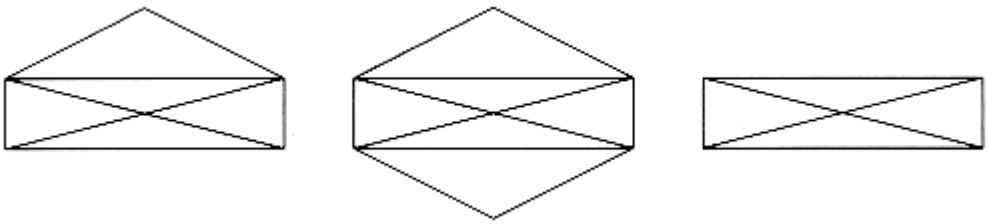
```
findart( w );
if( low[w] >= num[v] )
printf ( "%v is an articulation point\n", v );
low[v] = min( low[v], low[w] ); /* Rule */
}

else
if( parent[v] != w ) /* back edge */
low[v] = min( low[v], num[w] ); /* Rule 2 */
}
}
```

Euler Circuits

We must find a path in the graph that visits every edge exactly once. If we are to solve the "extra challenge," then we must find a cycle that visits every edge exactly once. This graph problem was solved in 1736 by Euler and marked the beginning of graph theory. The problem is thus commonly referred to as an **Euler path or Euler tour or Euler circuit problem**, depending on the specific problem statement.

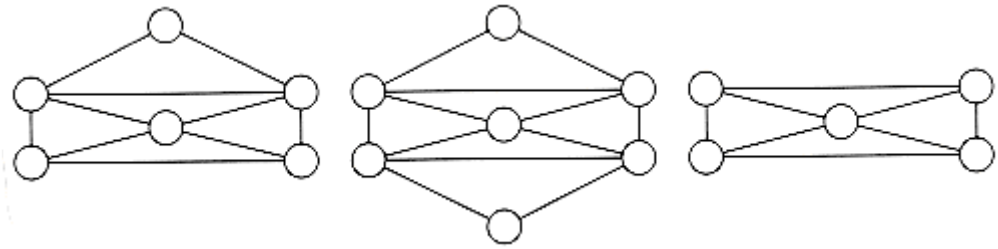
Consider the three figures as shown below. A popular puzzle is to reconstruct these figures using a pen, drawing each line exactly once. The pen may not be lifted from the paper while the drawing is being performed. As an extra challenge, make the pen finish at the same point at which it started.

Three drawings



1. The first figure can be drawn only if the starting point is the lower left- or right-hand corner, and it is not possible to finish at the starting point.
2. The second figure is easily drawn with the finishing point the same as the starting point.
3. The third figure cannot be drawn at all within the parameters of the puzzle.

We can convert this problem to a graph theory problem by assigning a vertex to each intersection. Then the edges can be assigned in the natural manner, as in figure.



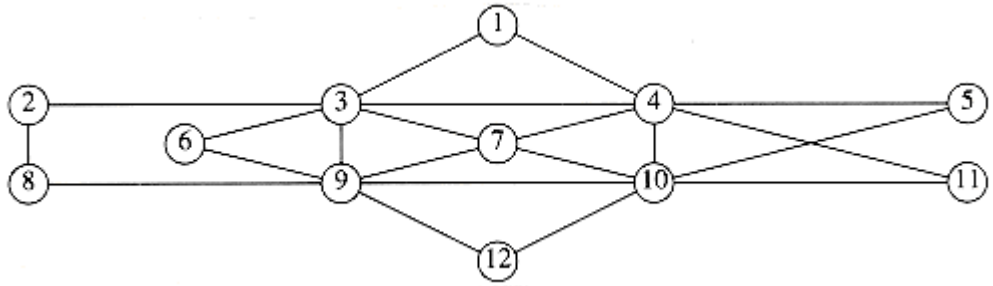
The first observation that can be made is that an Euler circuit, which must end on its starting vertex, is possible only if the graph is connected and each vertex has an even degree (number of edges). This is because, on the Euler circuit, a vertex is entered and then left.

If exactly two vertices have odd degree, an Euler tour, which must visit every edge but need not return to its starting vertex, is still possible if we start at one of the odd-degree vertices and finish at the other.

If more than two vertices have odd degree, then an Euler tour is not possible.

That is, **any connected graph, all of whose vertices have even degree, must have an Euler circuit**

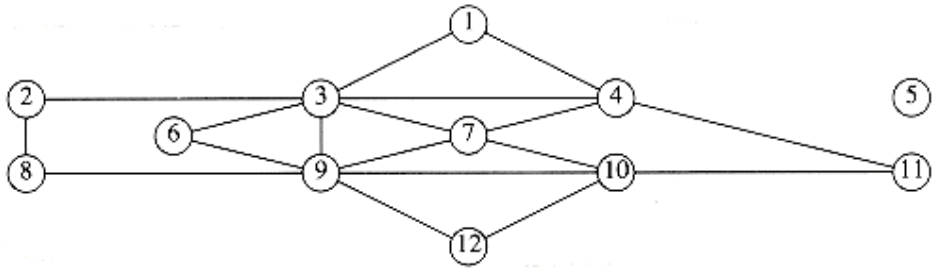
As an example, consider the graph in



The main problem is that we might visit a portion of the graph and return to the starting point prematurely. If all the edges coming out of the start vertex have been used up, then part of the graph is untraversed.

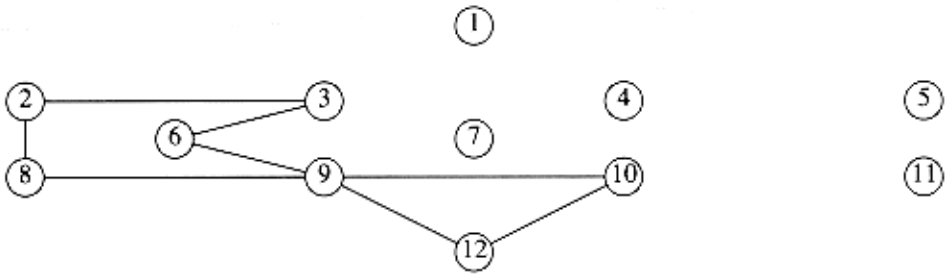
The easiest way to fix this is to find the first vertex on this path that has an untraversed edge, and perform another depth-first search. This will give another circuit, which can be spliced into the original. This is continued until all edges have been traversed.

Suppose we start at vertex 5, and traverse the circuit 5, 4, 10, 5. Then we are stuck, and most of the graph is still untraversed. The situation is shown in the Figure.



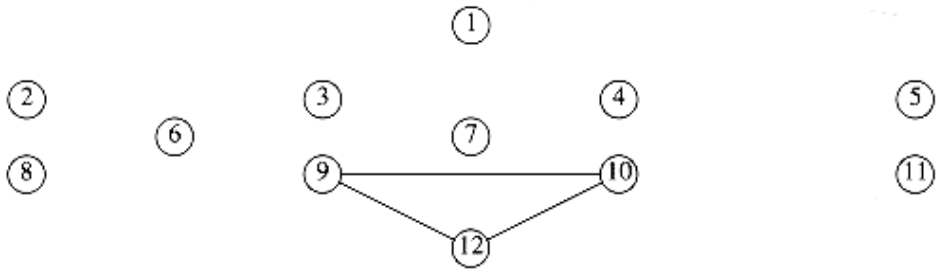
We then continue from vertex 4, which still has unexplored edges. A depth-first search might come up with the path 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4. If we splice this path into the previous path of 5, 4, 10, 5, then we get a new path of 5, 4, 1, 3, 7 ,4, 11, 10, 7, 9, 3, 4, 10, 5.

The graph that remains after this is shown in the Figure



The next vertex on the path that has untraversed edges is vertex 3. A possible circuit would then be 3, 2, 8, 9, 6, 3. When spliced in, this gives the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5.

The graph that remains is in the Figure.



On this path, the next vertex with an untraversed edge is 9, and the algorithm finds the circuit 9, 12, 10, 9. When this is added to the current path, a circuit of 5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5 is obtained. As all the edges are traversed, the algorithm terminates with an Euler circuit.

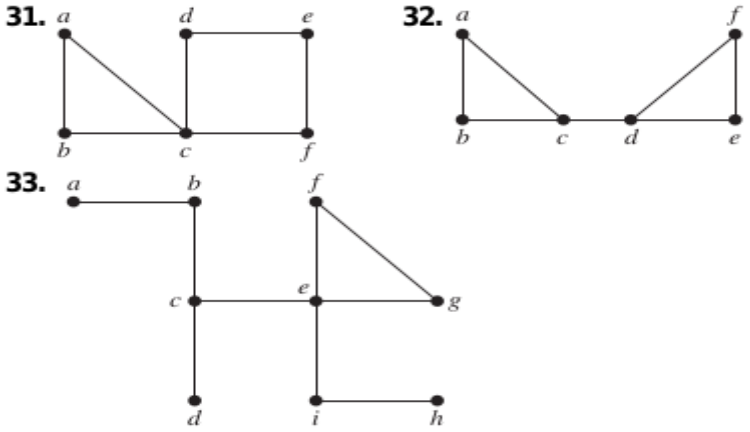
Then the Euler Path for the above graph is **5, 4, 1, 3, 2, 8, 9, 12, 10, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5**

Cut vertex and edges

A cut vertex is a vertex that when removed (with its boundary edges) from a graph creates more components than previously in the graph.

A cut edge is an edge that when removed (the vertices stay in place) from a graph creates more components than previously in the graph.

Find the cut vertices and cut edges for the following graphs



Answers

31) The cut vertex is c. There are no cut edges.

32) The cut vertices are c and d. The cut edge is (c,d)

33) The cut vertices are b,c,e and i. The cut edges are: (a,b),(b,c),(c,d),(c,e),(e,i),(i,h)

Applications of graph:

Minimum Spanning Tree

Definition:

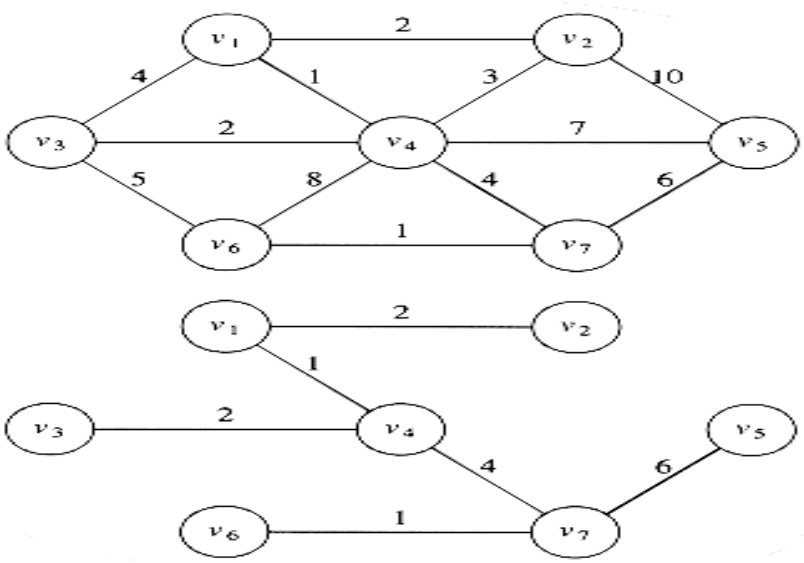
A minimum spanning tree exists if and only if G is connected. A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost.

The number of edges in the minimum spanning tree is |V| - 1. The minimum spanning tree is a tree because it is acyclic, it is spanning because it covers every edge.

Application:

- House wiring with a minimum length of cable, reduces cost of the wiring.

A graph G and its minimum spanning tree



There are two algorithms to find the minimum spanning tree

1. Prim's Algorithm
2. Kruskal's Algorithm

Kruskal's Algorithm

A second greedy strategy is continually to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.

Formally, **Kruskal's algorithm maintains a forest**. **Forest is a collection of trees.**

Procedure

- Initially, there are |V| single-node trees.
- Adding an edge merges two trees into one.
- When the algorithm terminates, there is only one tree, and this is the minimum spanning tree.
- The algorithm terminates when enough edges are accepted.

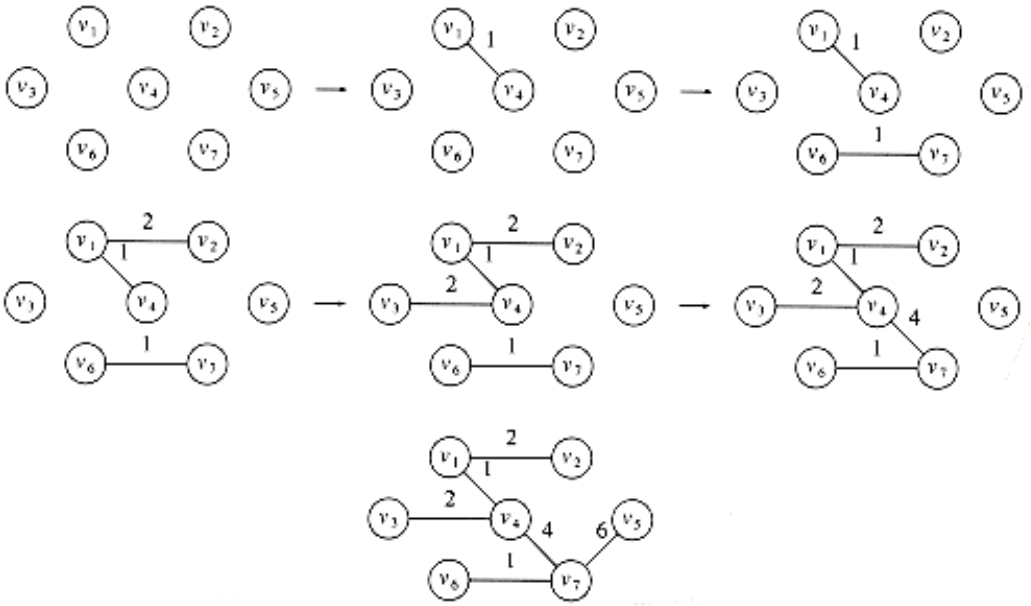
At any point in the process, two vertices belong to the same set if and only if they are connected in the current spanning forest. Thus, each vertex is initially in its own set.

- If u and v are in the same set, **the edge is rejected**, because since they are already connected, adding (u, v) would form a cycle.
- Otherwise, **the edge is accepted**, and a union is performed on the two sets containing u and v.

Action of Kruskal's algorithm on G

Edge	Weight	Action
(v1,v4)	1	Accepted
(v6,v7)	1	Accepted
(v1,v2)	2	Accepted
(v3,v4)	2	Accepted
(v2,v4)	3	Rejected
(v1,v3)	4	Rejected
(v4,v7)	4	Accepted
(v3,v6)	5	Rejected
(v5,v7)	6	Accepted

Kruskal's algorithm after each stage



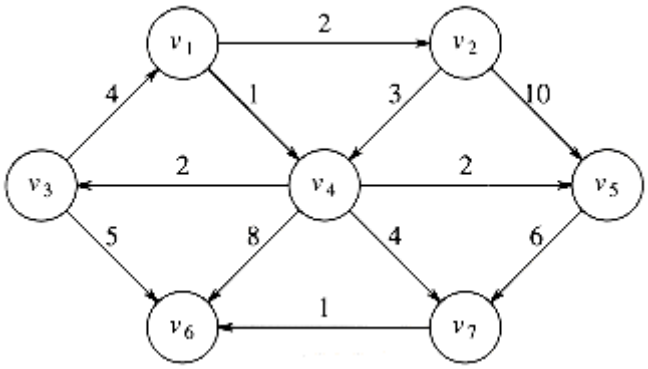
Routine for Kruskal's algorithm

```
void Graph:: kruskal( )
{
        int edgesaccepted = 0;          DISJSET ds ( Numvertex);
        PRIORIT_QUEUE < edge> pg( getedges ( ));
        Edge e;
```

```
Vertex U, V;
        while( edgesaccepted < NUMVERTEX-1 )
        {
                Pq. deletemin( e );      // e = (u, v)
                Settype Uset =ds. find( U, S );
                Settype Vset = ds.find( V, S );
                if( Uset != Vset )
                        {
                        // accept the edge
                        edgesaccepted++;
                        ds.setunion( S, Uset, Vset );
                                }       }       }
```

Dijkstra's Algorithm

If the graph is weighted, the problem becomes harder, but we can still use the ideas from the unweighted case.



Each vertex is marked as either known or unknown. A tentative distance dv is kept for each vertex. The shortest path length from s to v using only known vertices as intermediates.
The general method to solve the single-source shortest-path problem is known as Dijkstra's algorithm.

Dijkstra's algorithm proceeds in stages, just like the unweighted shortest-path algorithm. At each stage, Dijkstra's algorithm selects a vertex v, which has the smallest dv among all the unknown vertices, and declares that the shortest path from s to v is known.

In the above graph, assuming that the start node, s, is v1. The first vertex selected is v1, with path length 0. This vertex is marked known. Now that v1 is known.

Initial configuration table

v	Known	dv	pv
v1	0	0	0
v2	0	∞	0
v3	0	∞	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

The vertices adjacent to v1 are v2 and v4. Both these vertices get their entries adjusted, as indicated below

After v1 is declared known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	∞	0
v4	0	1	v1
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Next, v4 is selected and marked known. Vertices v3, v5, v6, and v7 are adjacent.

After v4 is declared known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	3	v4
v4	1	1	v1
v5	0	3	v4
v6	0	9	v4
v7	0	5	v4

Next, v2 is selected. v4 is adjacent but already known, so no work is performed on it. v5 is adjacent but not adjusted, because the cost of going through v2 is 2 + 10 = 12 and a path of length 3 is already known. **After v2 is declared known**

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	0	3	v4
v4	1	1	v1
v5	0	3	v4
v6	0	9	v4
v7	0	5	v4

The next vertex selected is v5 at cost 3. v7 is the only adjacent vertex, but it is not adjusted, because 3 + 6 > 5. Then v3 is selected, and the distance for v6 is adjusted down to 3 + 5 = 8.

After v5 and v3 are declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	0	8	v3
v7	0	5	v4

Next v7 is selected; v6 gets updated down to 5 + 1 = 6. The resulting table is

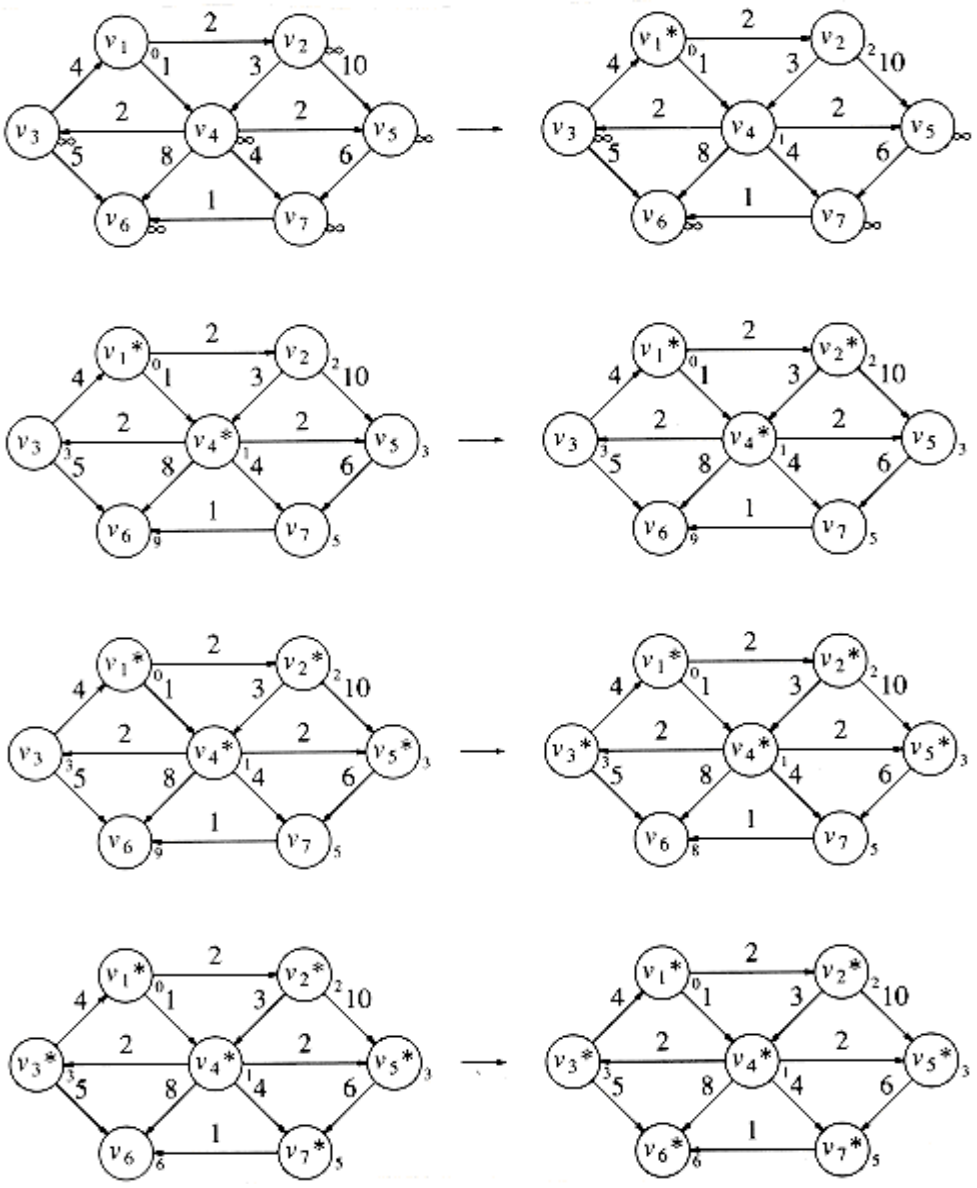
After v7 is declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	0	6	v7
v7	1	5	v4

Finally, v6 is selected. The final table is shown

After v6 is declared known and algorithm terminates

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	1	6	v7
v7	1	5	v4



Vertex class for Dijikstra's algorithm

```
struct Vertex
{
  List adj;
 Bool  known;
 disttype dist;
 Vertex path;  };
#define NOTAVERTEX  0
```

Routine for Dijkstra's algorithm

```
void graph :: dijkstra( Vertex S )
{
        for each vertex v
        {
         v.dist = INFINITY;
        v.known = false; }
        s.dist =0;
for( ; ; )
{
        v = smallest unknown distance vertex;
if(v== NotAVertex)
        break;
v. known = TRUE;
for each w adjacent to v
if( !w. known )
        if( v.dist + Cv,w < w.dist )
        {
                decrease( w.dist to v.dist + Cv,w );
                w.path = v;     }       }       }
```

Routine to print the actual shortest path

```
void Graph:: printpath( Vertex v)
{
        if( v.path != NOTAVERTEX )
        {
                printpath(v.path);
                cout<< " to " ; }
                cout<< v ;      }
```

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES 9

Searching – Linear Search – Binary Search. Sorting – Bubble sort – Selection sort – Insertion sort – Shell sort –. Merge Sort – Hashing – Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

---

1 SORTING ALGORITHMS

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms that require sorted lists to work correctly and for producing in human readable format input.

Sorting algorithms are often classified by :

* Computational complexity (worst, average and best case) in terms of the size of the list (N). For typical sorting algorithms good behaviour is O(NlogN) and worst case behaviour is $O(N^2)$ and the average case behaviour is O(N).
* Memory Utilization
* Stability - Maintaining relative order of records with equal keys.
* No. of comparisions.
* Methods applied like Insertion, exchange, selection, merging etc.

Sorting is the process of arranging the elements in either ascending or decending order.

Sorting techniques are categorized into

⇒ Internal Sorting

⇒ External Sorting

Internal Sorting takes place in the main memory of a computer.

Example :

Bubble sort, Insertion sort, Shell sort, Quick sort, Heap sort, etc.

External Sorting takes place in the secondary memory of a computer, as the number of objects to be sorted is too large to fit in main memory.

Example :

Merge Sort, Multiway Merge, Polyphase merge.

5.1.1 INSERTION SORT

Insertion sorts works by taking elements from the list one by one and inserting them in their current position into a new sorted list. Insertion sort consists of N - 1 passes, where N is the number of elements to be sorted. The i^{th} pass of insertion sort will insert the i^{th} element A[i] into its rightful place among A[1], A[2] --- A[i - 1]. After doing this insertion the records occupying A[1]....A[i] are in sorted order.

INSERTION SORT PROCEDURE

```
void Insertion_Sort (int a[ ], int n)
{
    int i, j, temp ;
    for (i = 0; i < n ; i++)
    {
            temp = a[i] ;
            for (j = i ; j>0 && a[j-1] > temp ; j--)
            {
                    a[j] = a[ j - 1] ;
            }
            a[j] = temp ;
    }
}
```

Example :

Consider an unsorted array as follows,

20 10 60 40 30 15

PASSES OF INSERTION SORT

ORIGINAL	20	10	60	40	30	15	POSITIONS MOVED
After i = 1	10	20	60	40	30	15	1
After i = 2	10	20	60	40	30	15	0
After i = 3	10	20	40	60	30	15	1
After i = 4	10	20	30	40	60	15	2
After i = 5	10	15	20	30	40	60	4
Sorted Array	10	15	20	30	40	60	

ANALYSIS OF INSERTION SORT

BEST CASE ANALYSIS	:	O(N)
AVERAGE CASE ANALYSIS	:	$O(N^2)$
WORST CASE ANALYSIS	:	$O(N^2)$

LIMITATIONS OF INSERTION SORT :

* It is relatively efficient for small lists and mostly - sorted lists.
* It is expensive because of shifting all following elements by one.

5.1.2 Selection Sort

Selection sort selects the smallest element in the list and place it in the first position then selects the second smallest element and place it in the second position and it proceeds in the similar way until the entire list is sorted.

Selection Sort Procedure

```
void Selection_Sort (int a[ ], int n)
{
        int temp, i, j;
        for (i = 0; i<n; i++)
        {
                for (j = i + 1; j<n; j++)
                {
                        if (a [i] > a [j])
                        {
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        }
                }
        }
}
```

Example :

Consider an unsorted list as follows

8	3	9	5	1
A[0]	A[1]	A[2]	A[3]	A[4]

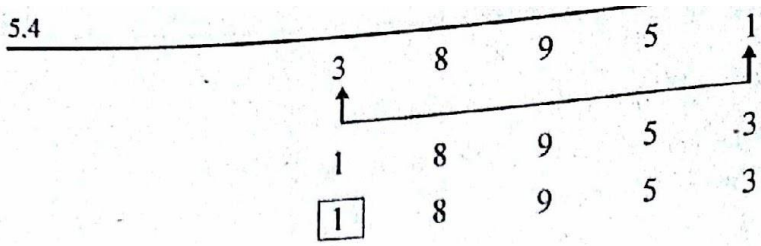
Original List 8 3 9 5 1

Pass 1 8 3 9 5 1

3 8 9 5 1

3 8 9 5 1

5.4



Pass 1 Output :

The smallest element is placed in the 1st position. The output of the pass 1 will be the input fo pass 2 and it starts processing i + 2 to n^{th} element in the same manner

Passes of Selection Sort

	A[0]	A[1]	A[2]	A[3]	A[4]
i = 0	1	8	9	5	3
i = 1	1	3	9	8	5
i = 2	1	3	5	8	9
i = 3	1	3	5	8	9
i = 4	1	3	5	8	9
Sorted list	1	3	5	8	9

Analysis Of Selection Sort

BEST CASE ANALYSIS	:	$O(N^2)$
AVERAGE CASE ANALYSIS	:	$O(N^2)$
WORST CASE ANALYSIS	:	$O(N^2)$

Limitations Of Selection Sort

It is inefficient and expensive for large sized list as it requires $O(n^2)$ comparisons

It doesn't stop even if the original list is sorted as it looks every element in the list for all passes

Advantages Of Selection Sort

It requires minimum space as the elements are swapped without using any addition storage.

5.1.3 Shell Sort

Shell sort was invented by Donald Shell. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. It works by arranging the data sequence in a two - dimensional array and then sorting the columns of the array using insertion sort.

In shell short the whole array is first fragmented into K segments, where K is preferably a prime number. After the first pass the whole array is partially sorted. In the next pass, the value of K is reduced which increases the size of each segment and reduces the number of segments. The next value of K is chosen so that it is relatively prime to its previous value. The process is repeated until K = 1, at which the array is sorted. The insertion sort is applied to each segment, so each successive segment is partially sorted. The shell sort is also called the Diminishing Increment Sort, because the value of K decreases continuously.

SHELL SORT ROUTINE 5.5

```
void shellsort (int A[ ], int N)
{
        int i, j, k, temp;
        for (k = N/2; k > 0 ; k = k/2)
                for (i = k; i < N ; i++)
                {
                        temp = A[i];
                        for (j = i; j >= k && A [ j - k] > temp ; j = j - k)
                        {
                                A[j] = A[j - k];
                        }
                                A[j] = temp;
                }
        }
```

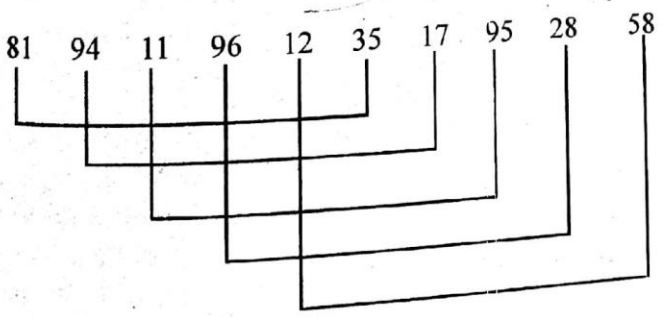
Example :

Consider an unsorted array as follows.

81 94 11 96 12 35 17 95 28 58

Here N = 10, the first pass as K = 5 (10/2)

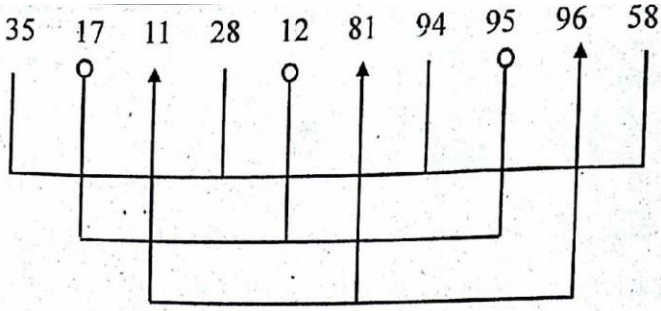
81 94 11 96 12 35 17 95 28 58



After first pass

35 17 11 28 12 81 94 95 96 58

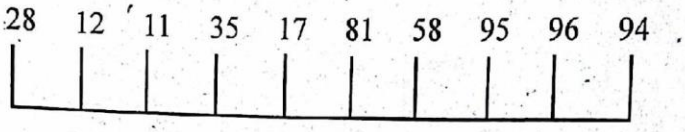
In second Pass, K is reduced to 3



After second pass,

28 12 11 35 17 81 58 95 96 94

In third pass, K is reduced to 1



The final sorted array is

11 12 17 28 35 58 81 94 95 96

ANALYSIS OF SHELL SORT :

BEST CASE ANALYSIS	: O(N log N)
AVERAGE CASE ANALYSIS	: $O(N^{1.5})$
WORST CASE ANALYSIS	: $O(N^2)$

ADVANTAGES OF SHELL SORT :

* It is one of the-fastest algorithms for sorting small number of elements.
* It requires relatively small amounts of memory.

5.1.4 Bubble Sort

Bubble sort is one of the simplest internal sorting algorithm. Bubble sort works by comparing two consecutive elements and the largest element among these two bubbles towards right at the end of the first pass the largest element gets sorted and placed at the end of the sorted list. This process is repeated for all pairs of elements until it moves the largest element to the end of the list in that iteration. Bubble sort consists of (n-1) passes, where 'n' is the number of elements to be sorted. In 1st pass the largest element will be placed in the n^{th} position. In 2nd pass the second largest element will be placed in the $(n-1)^{th}$ position. In $(n-1)^{th}$ pass only the first two elements are compared.

Bubble Sort Procedure

```
void bubble_sort (int a[ ], int n)
{
```

```
int i, j, temp;
for (i = 0; i < n - 1; i++)
{
        for (j = 0; j < n - 1; j++
        {
                if (a[j] > a [j + 1])
                {
                        temp = a[j];
                        a[j] = a[j + 1];
                        a[j + 1] = temp;
                }
        }
}
```

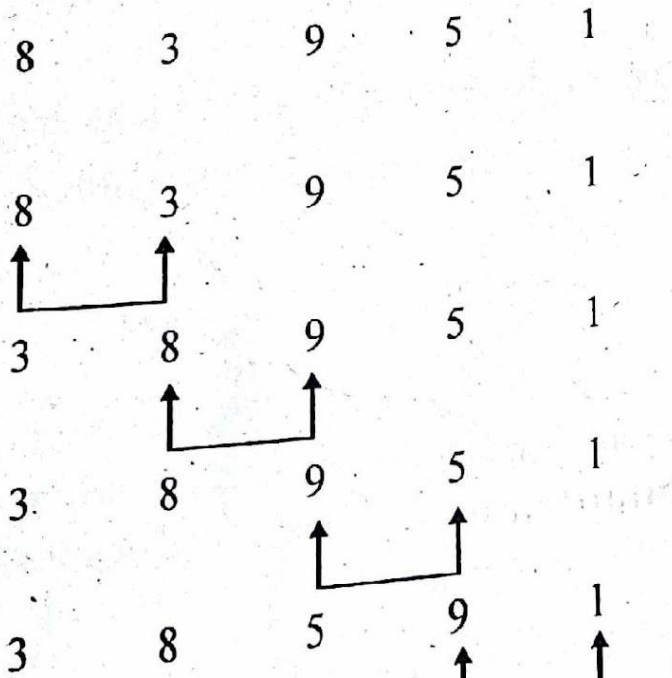
Example :

Consider an unsorted array as follows :

8	3	9	5	1
A[0]	A[1]	A[2]	A[3]	A[4]

Original List 8 3 9 5 1

Pass 1



At the end of 1st pass, the largest element is placed in the n^{th} position

The output of pass 1 will be the input for pass 2 and processed in the same manner leaving the n^{th} element.

Passes of Bubble Sort

	A[0]	A[1]	A[2]	A[3]	A[4]
i = 0	3	8	5	1	[9]
i = 1	3	5	1	[8]	9
i = 2	3	1	[5]	8	9
i = 3	1	[3]	5	8	9
Sorted list	1	3	5	8	9

Anlaysis of Bubble Sort :

BEST CASE ANALYSIS	:	$O(N^2)$
AVERAGE CASE ANALYSIS	:	$O(N^2)$
WORST CASE ANALYSIS	:	$O(N^2)$

Advantages of Bubble Sort :

It is popular and easy to implement.

It requires minimum space as the elements are swapped in place without using the additional temporary storage.

Limitations of Bubble Sort :

It is not well suited for large number of elements as it requires large amount of swapping.

It is slow in execution for large sized list and requires $O(n^2)$ for both swapping and comparisons

5.1.5 Quick Sort

Quick Sort is the most efficient internal sorting technique. It posseses a very good average case behaviour among all the sorting techniques. It is also called partioning sort which uses divide and conquer techniques.

The quick sort works by partioning the array A[1], A[2].....A[n] by picking some keyvalue in the array as a pivot element. Pivot element is used to rearrange the elements in the array. Pivot can be the first element of an array and the rest of the elements are moved so that the elements on left side of the pivot are lesser than the pivot, whereas those on the right side are greater than the pivot. Now the pivot element is placed in its correct position. Similarly the quicksort procedure is applied for left array and right array in a recursive manner.

Sorting, Searching

QUICK SORT ROUTINE 5.9

```
void qsort (int A[ ], int left, int right)
{
   int i, j, pivot, temp ;
   if (left < right)
   {
         pivot = left;
         i = left + 1;
         j = right;
         while (i < j)
         {
               while (A [PIVOT] > = A[i])
                     i = i + 1;
               while (A[PIVOT] < A[j])
                     j = j - 1 ;
               if (i < j)
               {
                     temp = A[i];     // swap  A[i] & A[j]
                     A[i] = A[j];
                     A[j] = temp;
               }
         }
               temp = A[PIVOT];       // swap A[PIVOT] & A[j]
               A[PIVOT] = A[j];
               A[j] = temp;
               qsort (A, left, j - 1);
               qsort (A, j+1, right);              // recursively done for partitioned array
   }
}
```

Example :

Consider an unsorted array as follows,

40	20	70	14	60	61	97	30

Here PIVOT = 40, i = 20, j = 30

⇒ The value of i is incremented till a[i] ≤ Pivot and the value of j is decremented till a[j] > pivot, this process is repeated until i < j.

⇒ If a[i] > pivot and a[j] < pivot and also if i < j then swap a[i] and a[j].

⇒ If i > j then swap a[j] and a[pivot].

Once the correct location for PIVOT is found, then partition array into left sub array and right subarray, where left sub array contains all the elements less than the PIVOT and right sub array contains all the elements greater than the PIVOT

PASSES OF QUICK SORT

(40)	20	70	14	60	61	97	30
Pivot	i						j
(40)	20	70	14	60	61	97	30
Pivot		i		i			j
			// if i < j then Swap (a[i], a[j])				
(40)	20	30	14	60	61	97	70
Pivot		i		i			j
(40)	20	30	14	60	61	97	70
Pivot			i			j	
(40)	20	30	14	60	61	97	70
Pivot				i		j	
(40)	20	30	14	60	61	97	70
Pivot				i			j
(40)	20	30	14	60	61	97	70
Pivot				i,	j		
(40)	20	30	14	60	61	97	70
Pivot				i, j			
(40)	20	30	14	60	61	97	70
Pivot			j	i			
			As i > j swap (a[j], a[pivot])				
			i.e swap (60, 40)				
{14	20	30}	40	{60	61	97	70}

Now, the pivot element has reached its correct position. The elements lesser than the Pivot {14, 20, 30} is considered as left sub array. The elements greater than the pivot {60, 61, 97, 70} is considered as right sub array. Then the q sort procedure is applied recursively for both these arrays.

The resultant sorted array is 14 20 30 40 60 61 70 97

ANALYSIS OF QUICK SORT

BEST - CASE ANALYSIS	:	O(N log N)
AVERAGE - CASE ANALYSIS	:	O(N log N)
WORST - CASE ANALYSIS	:	$O(N^2)$

Advantages of Quick Sort

1. It is faster than other O(N log N) algorithms.
2. It has better cache performance and high speed.
3. It is a best sorting algorithm as it applies divide and conquer principle.
4. It works well for large data set without additional storage.

Limitation

* Requires more memory space
* Poor selection of pivot leads to $0(n^2)$ complexity.

5.1.6 Merge Sort

The most common algorithm used in external sorting is the mergesort. This algorithm follows divide and conquer strategy. In dividing phase, the problem is divided into smaller problem and solved recursively. In conquering phase, the partitioned array is merged together recursively. Merge sort is applied to the first half and second half of the array. This gives two sorted halves, which can then be recursively merged together using the merging algorithm.

The basic merging algorithm takes two input arrays A and B and an output array C. The first element of A array and B array are compared, then the smaller element is stored in the output array C and the corresponding pointer is incremented. When either input array is exhausted the remainder of the other array is copied to an output array C.

MERGE SORT ROUTINE

```
void mergesort (int A [ ], int temp [ ], int n)
{
    msort (A, temp, 0, n-1);
}
    void msort (int A [ ], int temp [ ], int left, int right)
    {
            int center;
            if (left < right)
            {
```

5.12

```
            center = (right + left)/2;
            msort (A, temp, left, center);
            msort (A, temp, center + 1, right);
            merge (A, temp, left, center + 1, right);
        }
    }
```

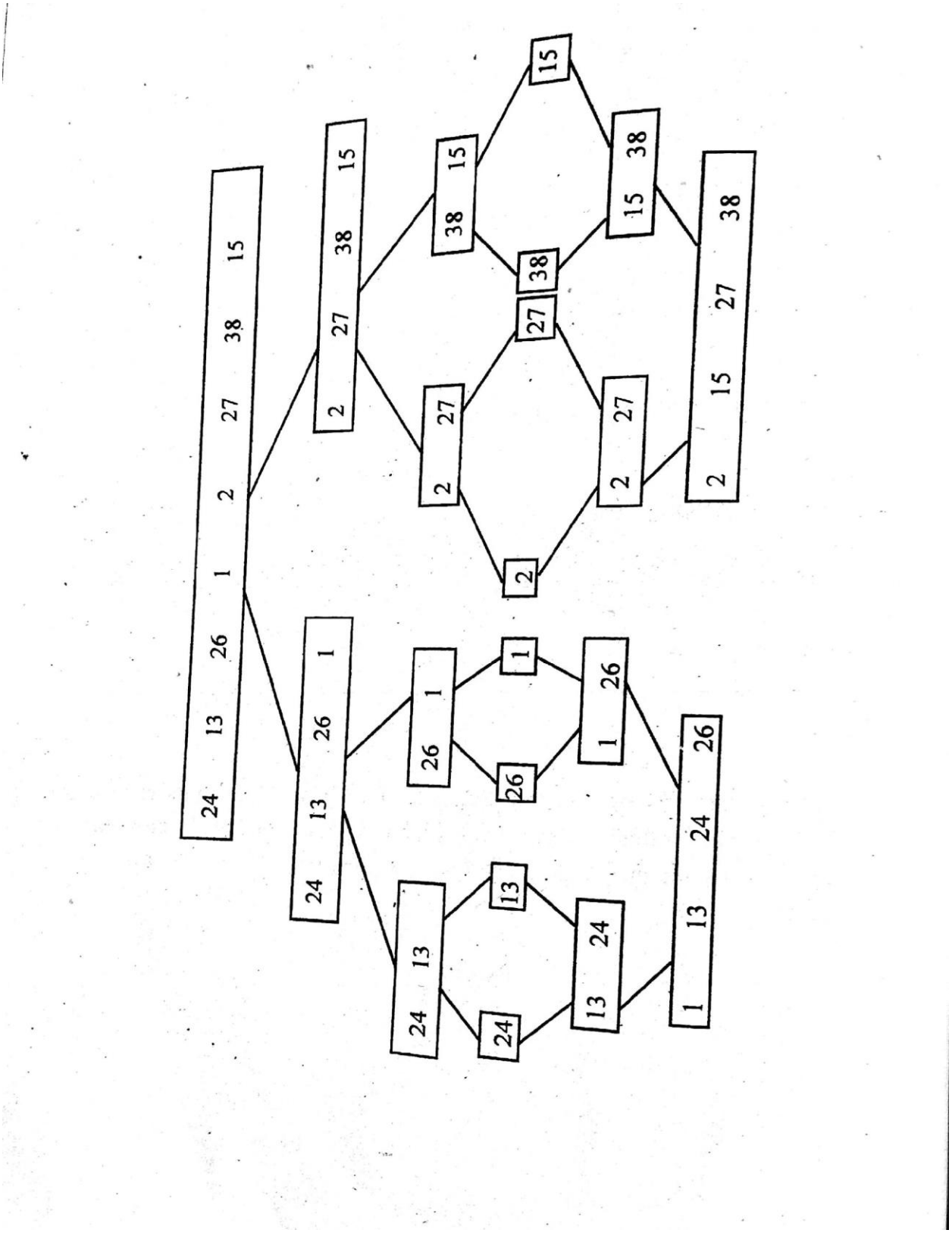
MERGE ROUTINE

```
    void merge (int A [ ], int temp [ ], int left, int center, int right)
    {
        int i, left _ end, num_elements, tmp_pos;
        left_end = center - 1;
        tmp_pos = left;
        num_elements = right - left + 1 ;
        while (( left < = left_end) && (center < = right))
        {
            if (A [left] < = A[center])
            {
                temp [tmp_pos] = A [left];
                tmp_pos++;
                left++;
            }
        else
        {
            temp [tmp_pos] = A [center];
            tmp_pos++;
            center ++ ;
        }
    }
        while (left < = left_end)
```

```
{
        temp [tmp_pos] = A[left];
        left ++;
        tmp pos++;
}
while (center < = right)
{
        temp [tmp_pos] = A [center];
        center ++ ;
        tmp_pos++;
}
for (i = 0; i < = num_elements ; i++)
{
        A[right] = temp [right];
        right - -;
}
}
```

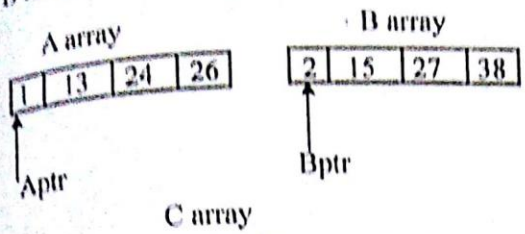
Example :

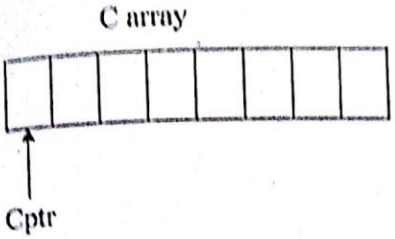
For instance, to sort the eight element array 24, 13, 26. 1, 2, 27, 38, 15, first four and last four elements are recursively sorted to obtain 1, 13, 24, 26, 2, 15, 27, 38. Finally, these array is divided into two halves and the merging algorithm is applied to get the sorted array.


24 13 26 1 2 27 38 15
2 27 38 15
38 15
15
38
15 38
2 15 27 38
2 27
27
2 27
2
1
26 1
26
1 26
24 13 26 1
24 13
13
13 24
1 13 24 26

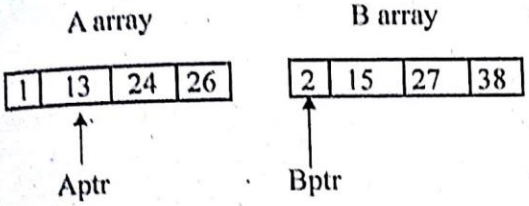
Now, the merging algorithm is applied as follows,

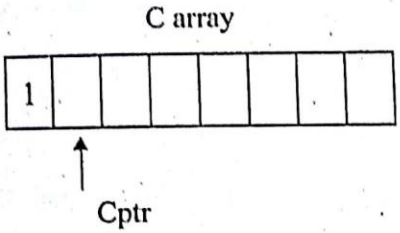
Let us consider first 4 elements 1, 13, 24, 26 as A array and the next four elements 2, 15, 27, 38 as B array.



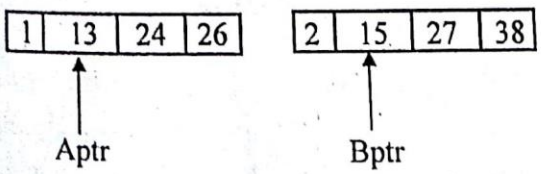


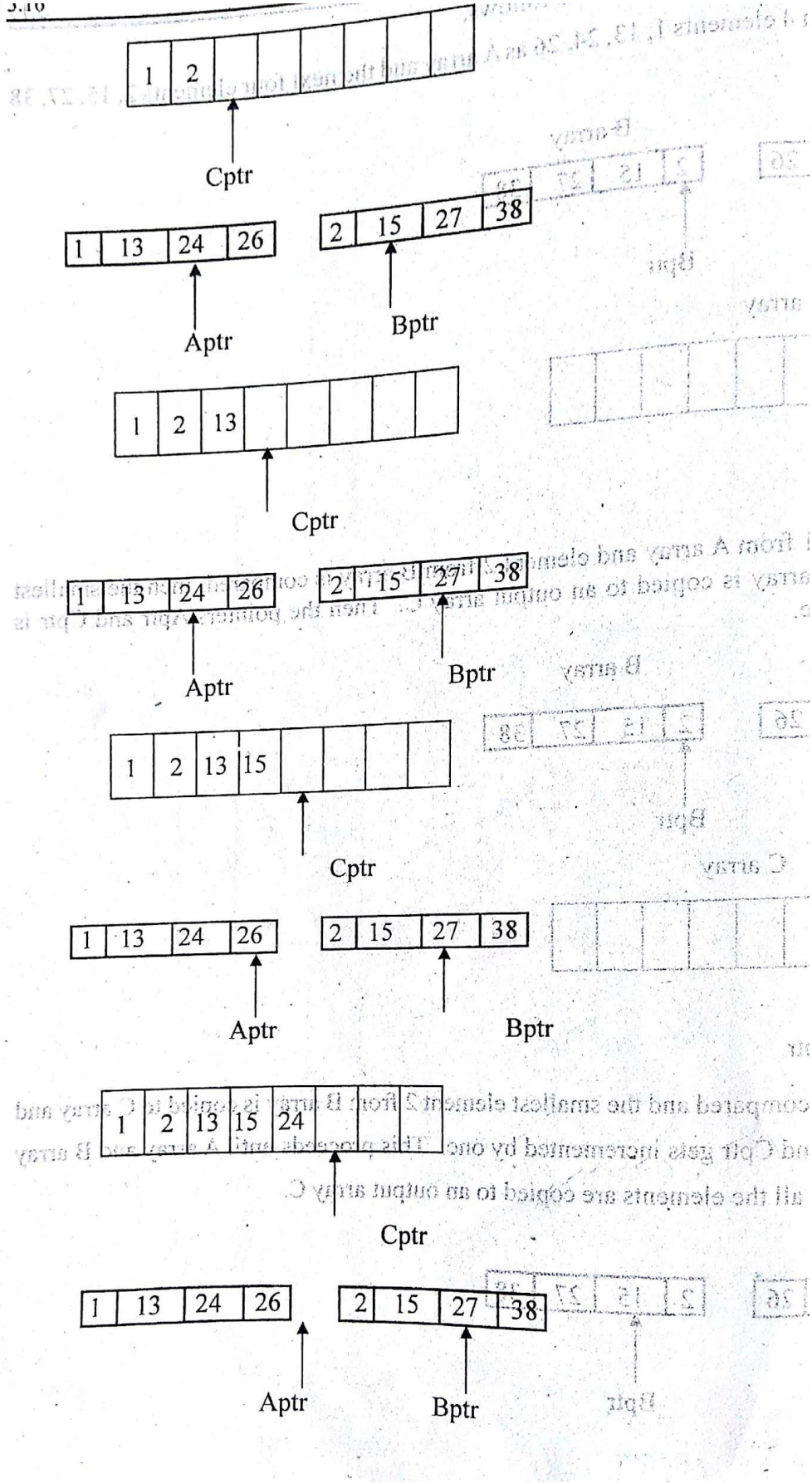
First, the element 1 from A array and element 2 from B array is compared, then the smallest element 1 from A array is copied to an output array C. Then the pointers Aptr and Cptr is incremented by one.

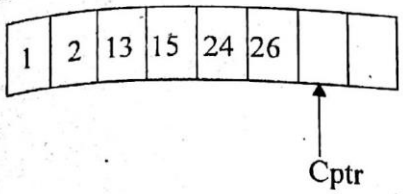




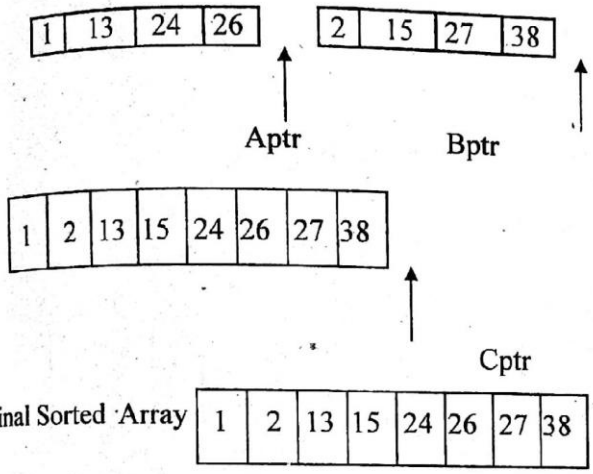
Next, 13 and 2 are compared and the smallest element 2 from B array is copied to C array and the pointers Bptr and Cptr gets incremented by one. This proceeds until A array and B array are exhausted, and all the elements are copied to an output array C.




1 2
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1 2 13
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1 2 13 15
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr
1 2 13 15 24
Cptr
1 13 24 26
Aptr
2 15 27 38
Bptr



Since A array is exhausted, the remaining elements of B array is then copied to C array.



ANALYSIS OF MERGESORT

BEST CASE ANALYSIS	:	O(N log N)
AVERAGE CASE ANALYSIS	:	O(N log N)
WORST CASE ANALYSIS	:	O (N log N)

Limitations of Merge Sort

* Merge sort sorts the larger amount of data making use of external storage device.
* It requires extra memory space

Advantages

* It has better cache performance
* Merge sort is a stable sort
* It is simpler to understand.

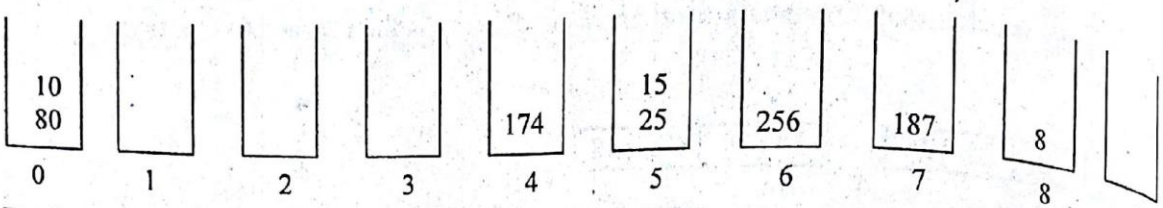
5.1.7Radix Sort

Radix sort is one of the linear sorting algorithm for integers. It is a generalised form of bucket sort. It can be performed using buckets from 0 to 9. It is also called as Binsort, card sort. It works by sorting the input based an each digit. In First pass, all the elements are sorted

according to the least significant digit. In second pass, the elements are arranged according to the next least significant digit and so on till the most significant digit. The number of passes in a Radix Sort depends upon the number of digits in the given numbers.

PASS 1 :

INPUT : 25, 256, 80, 10, 8, 15, 174, 187

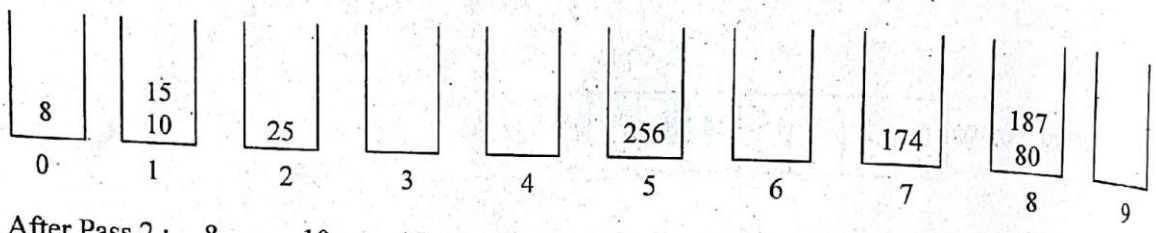


Buckets

After Pass 1 : 80, 10, 174, 25, 15, 256, 187, 8

PASS 2 :

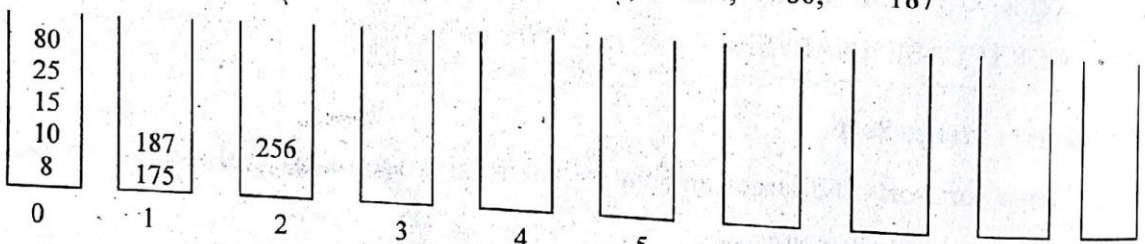
INPUT : 80, 10, 174, 25, 15, 256, 187, 8



After Pass 2 : 8, 10, 15, 25, 256, 174, 80, 187

PASS 3 :

INPUT : 8, 10, 15, 25, 256, 174, 80, 187



After pass 3 : 8, 10, 15, 25, 80, 175, 187, 256

Maximum number of digits in the given list is 3. Therefore the number of passes required to sort the list of elements is 3.

BEST CASE ANALYSIS	:	O(N log N)
AVERAGE CASE ANALYSIS	:	O(N log N)
WORST CASE ANALYSIS	:	O(N log N)

5.2 SEARCHING ALGORITHMS

Searching is a method to search a data item in the given set. There are two types of searching. They are

(i) Linear Search

(ii) Binary Search

5.2.1 Linear Search

Linear Search is used to search a data item in the given set in the sequential manner, starting from the first element. It is also called as sequential search.

ROUTINE FOR LINEAR SEARCH

```
void Linear_Search (int X, int a [ ], int n );
{
        int flag = 0, i;
        for (i = 0; i < n; i++)
        {
                if (X = = a[i])
                {
                        flag = 1
                        break;
                }
        }
        if (flag = = 1)
                print("The element is found")
        else
                print("The element is not found);
}
```

ANALYSIS OF LINEAR SEARCH

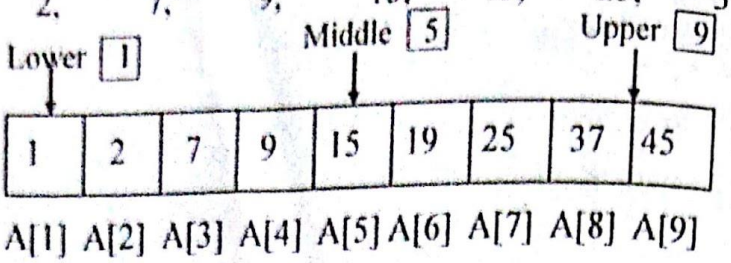
BEST CASE ANALYSIS	:	O(1)
AVERAGE CASE ANALYSIS	:	O(N)
WORST CASE ANALYSIS	:	O(N)

5.2.2 Binary Search

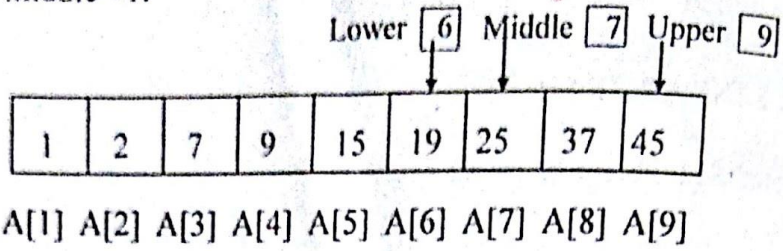
Binary Search is used to search an item in a sorted list. In this method, initialise the lower limit as 1 and upper limit as N. the middle position is computed as (lower + upper)/2 and check the element in the middle position with the data item to be searched. If the data item is greater than the middle value then the lower limit is adjusted to one greater than the middle value. Otherwise, the upper limit is adjusted to one less than the middle value.

For example :- X = 25

Let A = 1, 2, 7, 9, 15, 19, 25, 37



check 25 and 15 (ie) X > A[middle] since the data item is greater than the middle value. the lower limit as middle +1.



Since the data item is equal to the middle value. The element is found at the position 7.

ROUTINE FOR BINARY SEARCH

```
void Binary_Search (int X, int a [ ], int n );
{
        int lower, upper, mid;
        lower = 1;
        upper = n;
        while (lower < upper)
        {       mid = (lower + upper) / 2;
                if (X > a [mid])
                lower = mid + 1;
                else if (X < a[mid])
                upper = mid -1;
                else
                        {
                                print ("Element is found");
                                break;
                        }
        }
}
```

ANALYSIS OF BINARY SEARCH

BEST CASE ANALYSIS	:	O(1)
AVERAGE CASE ANALYSIS	:	O(log N)
WORST CASE ANALYSIS	:	O (log N)

5.3 HASHING TECHNIQUES

Hashing is a technique used for performing insertions, deletions and search operation in constant average time by implementing Hash Table data structure.

5.3.1 Types of Hashing

Static Hashing

In static hashing, the hash function maps search key values to a fixed set of locations.

Dynamic Hashing

In Dynamic hashing, the hash table can grow to handle more items. The associated hash function must change as the table grows.

Hash table

The hash table data structure is an array of some fixed size table, containing the keys. A key is a value associated with each record.

* A hash table is partioned into array of buckets.
* Each bucket has many slots and each slot holds one record.

Location	Slot 1
1	5
2	2
3	
4	

Fig 5.1 Hash Table

5.3.2 Hashing Functions

A hashing function is a key-to-address transformation which acts upon a given key to compute the relative position of the key in an array.

- A key can be a number, a string, a record etc.

A simple Hash Function

Hash (key value) = key_value Mod Table_size

Or

Hash table

The hash table data structure is an array of some fixed size table, containing the keys. A key is a value associated with each record.

* A hash table is partioned into array of buckets.
* Each bucket has many slots and each slot holds one record.

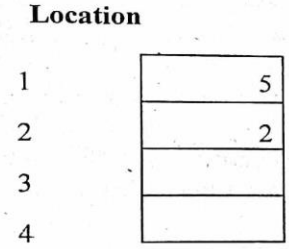


Fig 5.1 Hash Table

5.3.2 Hashing Functions

A hashing function is a key-to-address transformation which acts upon a given key to compute the relative position of the key in an array.

- A key can be a number, a string, a record etc.

A simple Hash Function

Hash (key value) = key_value Mod Table_size

Routine for Hash Function

```
INDEX hash( char *key, int tablesize )
{
int hash_val = 0;
while( *key != '\0' )
hash_val += *key++;
return( hash_val % H_SIZE );
}
```

Collision:

Collision occurs when a ash value of a record being inserted hashes to an address that already contain a different record (i.e) when two key values hash to the same position.

Example

Values 89 and 39 are hash to the same address 9, if the table size is 10.

Collision Resolution Methods:

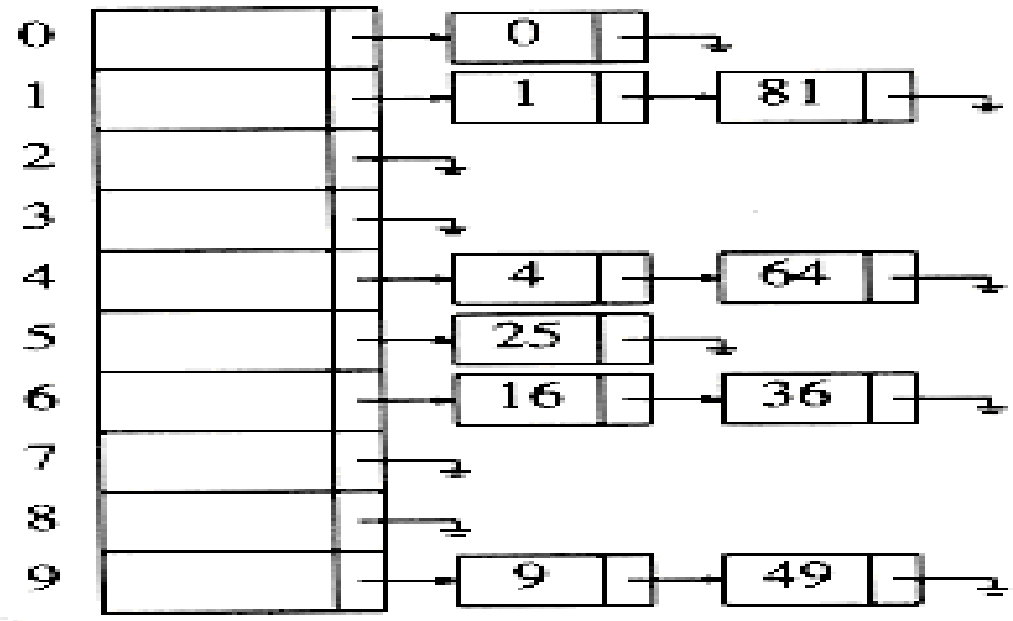
1. **Open Hashing –** Each bucket in the hashtable is the head of a Linkedlist. Collide elements are stored outside the table. **Eg.Separate Chaining**

2. **Closed Hashing –** Collide elements are stored at another slot in the table. Ensures that all the elements are stored directly into the hash table. **Eg Open addressing. Rehashing and Extendible Hashing**

Open addressing : Linear Probing, Quadratic Probing and Double Hashing

1. Separate Chaining / Open Hashing

The first strategy, commonly known as either open hashing, or separate chaining, is to keep a list of all elements that hash to the same value. For convenience, our lists have headers.
hash(x) = x mod 10. (The table size is 10)
To perform an insert, we traverse down the appropriate list to check whether the element is already in place. If the element turns out to be new, it is inserted either at the front of the list or at the end of the list. New elements are inserted at the front of the list.




```
struct listnode
{
elementtype element;
position next;
};

struct hashtbl
{
int tablesize;
```

```
LIST *thelists;
};
```

Initialization routine for open hash table

```
HASHTABLE initializetable(int tablesize )
{
HASHTABLE H;
int i;
if( table size < MIN_TABLE_SIZE )
{
error("Table size too small");
return NULL;
}
H = (HASH_TABLE) malloc ( sizeof (struct hashtbl) );
if( H == NULL )
fatalerror("Out of space!!!");
H->tablesize = nextprime( tablesize );
H->thelists = malloc( sizeof (LIST) * H->tablesize );
if( H->thelists == NULL )
fatalerror("Out of space!!!");
for(i=0; i<H->tablesize; i++ )
{
H->thelists[i] = malloc( sizeof (struct listnode) );
if( H->thelists[i] == NULL )
fatalerror("Out of space!!!");
else
H->thelists[i]->next = NULL;
}
return H;
}
```

Routine for Find operation

```
Position find( elementtype key, HASHTABLE H )
```

```
{
position p;
LIST L;
L = H->thelists[ hash( key, H->tablesize) ];
p = L->next;
while( (p != NULL) && (p->element != key) )
p = p->next;
return p;
}
```

Routine for Insert Operation

```
Void insert( elementtype key, HASHTABLE H )
{
position pos, newcell; LIST L;
pos = find( key, H );
 if( pos == NULL )
{
newcell = (position) malloc(sizeof(struct listnode));
if( newcell == NULL )
fatalerror("Out of space!!!");
else
{
L = H->thelists[ hash( key, H->table size ) ];
newcell->next = L->next;
newcell->element = key;
L->next = newcell; }    } }
```

Closed Hashing (Open Addressing)

Separate chaining has the disadvantage of requiring pointers. This tends to slow the algorithm down a bit because of the time required to allocate new cells, and also essentially requires the implementation of a second data structure.

Closed hashing, also known as open addressing, is an alternative to resolving collisions with linked lists.

In a closed hashing system, if a collision occurs, alternate cells are tried until an empty cell is found. More formally, cells h0(x), h1(x), h2(x), . . . are tried in succession where hi(x) = (hash(x) + F(i) mod tablesize), with F(0) = 0.

The function, F , is the collision resolution strategy. Because all the data goes inside the table, a bigger table is needed for closed hashing than for open hashing. Generally, the load factor should be below = 0.5 for closed hashing.

Three common collision resolution strategies are

1. Linear Probing
2. Quadratic Probing
3. Double Hashing

Linear Probing

In linear probing, F is a linear function of i, typically F(i) = i. This amounts to trying cells sequentially (with wraparound) in search of an empty cell.

F (i) = i.

The below Figure shows the result of inserting keys {89, 18, 49, 58, 69} into a closed table using the same hash function as before and the collision resolution strategy, The first collision occurs when 49 is inserted; it is put in the next available spot, namely 0, which is open. 58 collides with 18, 89, and then 49 before an empty cell is found three away.

{89, 18, 49, 58, 69}

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect-the collision function is

quadratic. The popular choice is $F(i) = i^2$

When 49 collide with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next 58 collides at position 8. Then the cell one away is tried but another collision occurs. A vacant cell is found at the next cell tried, which is $2^2 = 4$ away. 58 is thus placed in cell 2.

{89, 18, 49, 58, 69}

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing

The last collision resolution method we will examine is double hashing. For double hashing, one popular choice is f(i) = i h2 (x). This formula says that we apply a second hash function to x and probe at a distance h2 (x), 2 h2 (x), . . ., and so on. A function such as h2 (x) = R - (x mod R), with R a prime smaller than H_SIZE, will work well.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Rehashing

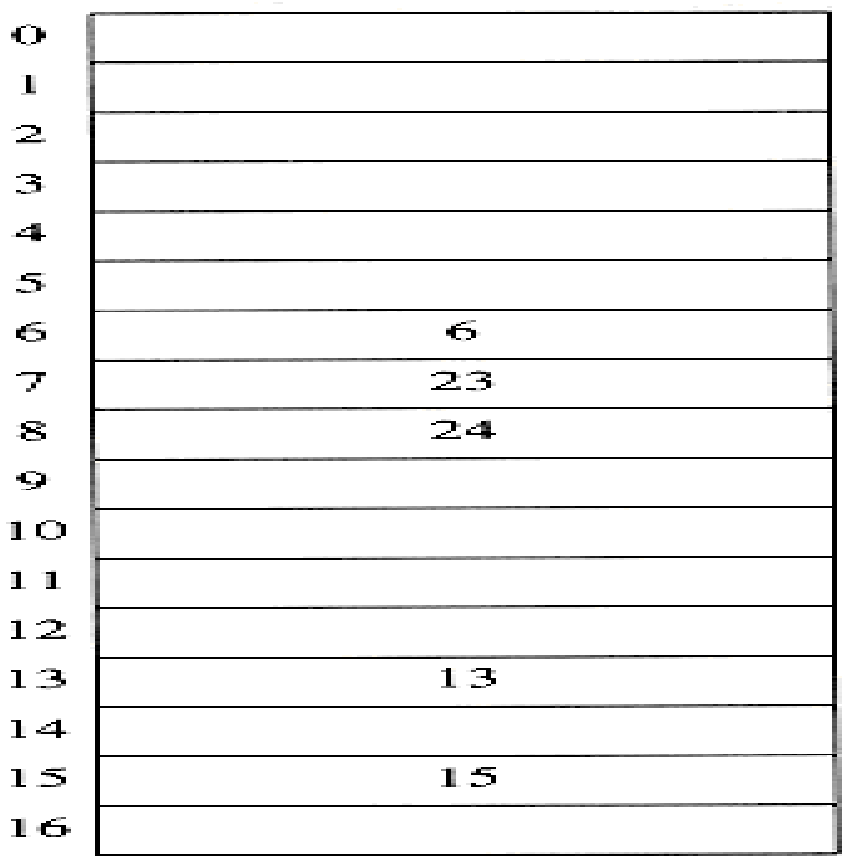
If the table gets too full, the running time for the operations will start taking too long and inserts might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions.

A solution, then, is to build another table that is about twice as big and scan down the entire original hash table, computing the new hash value for element and inserting it in the new table.

As an example, suppose the elements 13, 15, 24, and 6 are inserted into a closed hash table of size 7. The hash function is h(x) = x mod 7. Suppose linear probing is used to resolve collisions.

0	6
1	15
2	
3	24
4	
5	
6	13

0	6
1	15
2	23
3	24
4	
5	
6	13



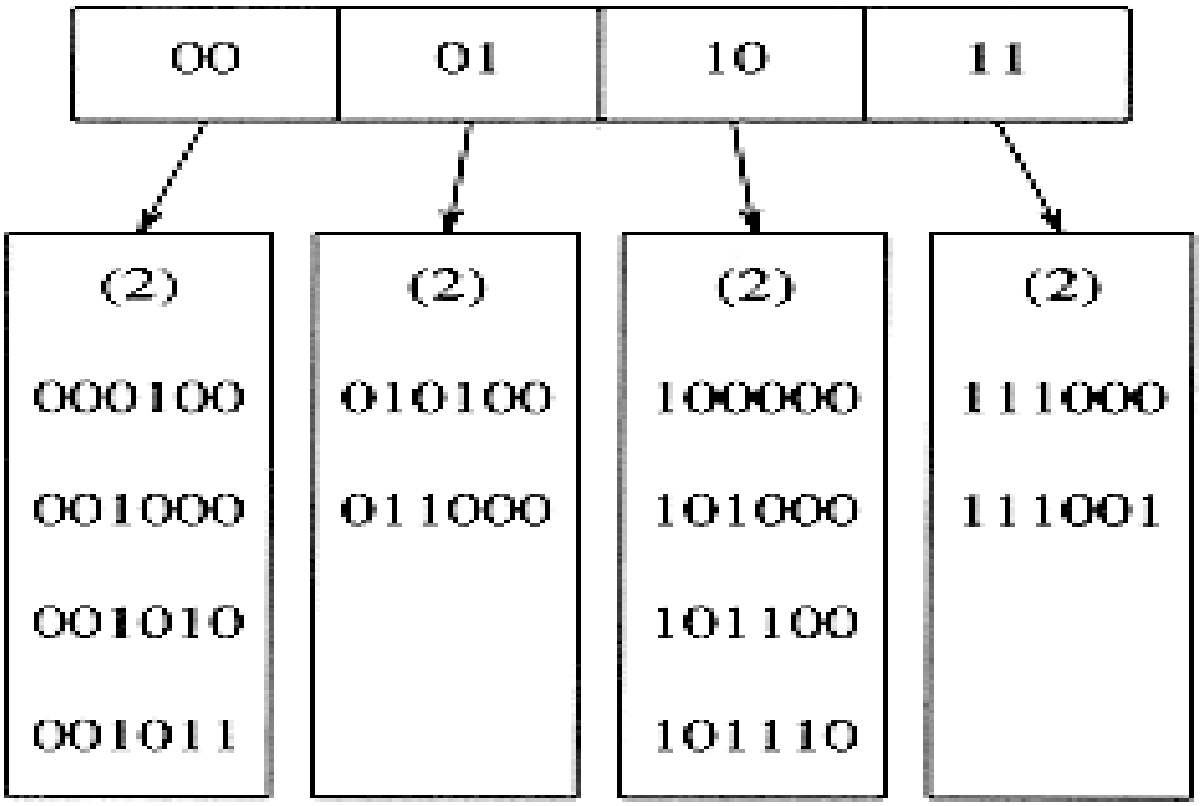
Rehashing routines

```
Hashtable rehash( HASH_TABLE H )
{
unsigned int i, old_size;
cell *old_cells;
old_cells = H->the_cells;
old_size = H->table_size;
/* Get a new, empty table */
H = initialize_table( 2*old_size );
/* Scan through old table, reinserting into new */
for( i=0; i<old_size; i++ )
if( old_cells[i].info == legitimate )
insert( old_cells[i].element, H );
free( old_cells );
return H;
}
```

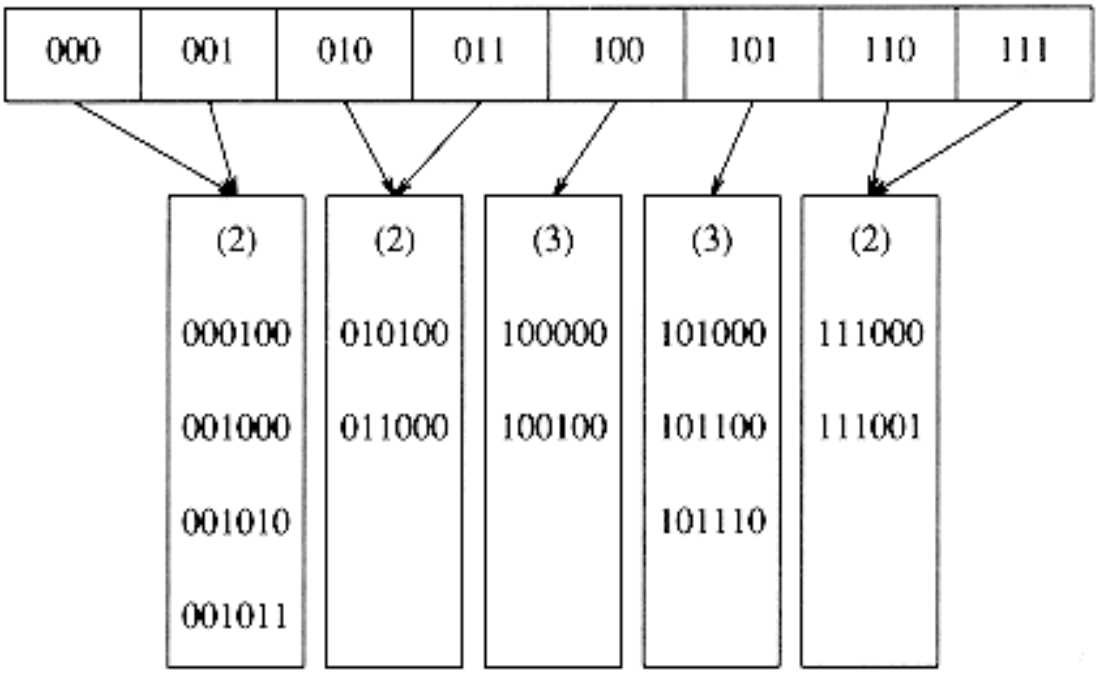
Extendible Hashing

If the amount of data is too large to fit in main memory, then is the number of disk accesses required to retrieve data. As before, we assume that at any point we have n records to store; the value of n changes over time. Furthermore, at most m records fit in one disk block. We will use m = 4 in this section. To be more formal, D will represent the number of bits used by the root, which is sometimes known as the directory. The number of entries in the directory is thus 2^D . dL is the

number of leading bits that all the elements of some leaf have in common. dL will depend on the particular leaf, and dL<=D.

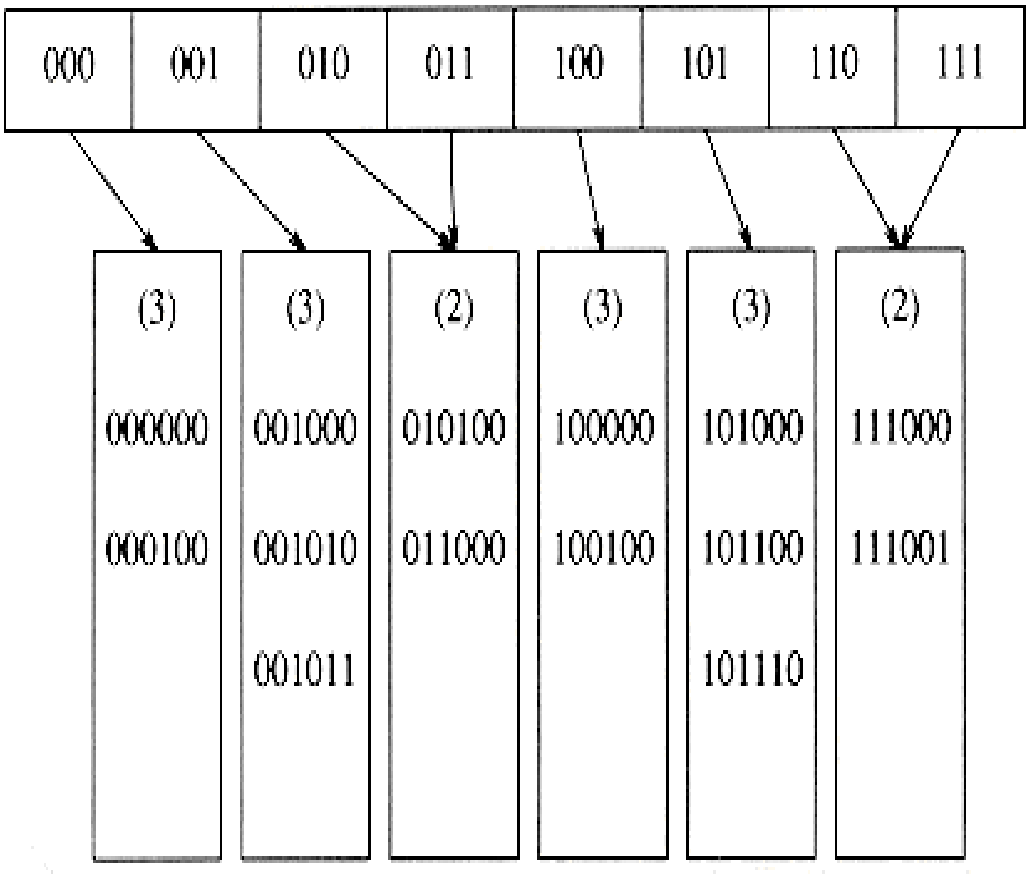


Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room. We thus split this leaf into two leaves, which are now determined by the first three bits. This requires increasing the directory size to 3.



If the key 000000 is now inserted, then the first leaf is split, generating two leaves with dL = 3.

Since D = 3, the only change required in the directory is the updating of the 000 and 001 pointers.



Different **Methods of Hashing function**

METHODS OF HASHING FUNCTION

(a) Mid Square Method

In this method, the key is squard and the middle part of the result based on the number of digits required for addressing is taken as the hash value. This method works will if the keys do not contain a lot of leading or trailing zeros.

H(X) = return middle digits of X^2

For example : Map the key 2453 into a hash table of size 1000; there, X = 2453

$X^2 = 6017209$

Extract Middle Value 172 as the hash value

(b) Module Division or Division Remainder

This method computes hash value from key using the (%) modulo operator. Here, Table size that is power of 2 like 32, 64 and 1024 should be avoided as it leads to more collisions. It is always better to select table size not close to power of 2.

H(Key) = return Key % Table_size

Let X = 123203241

Partition X into 123, 203, 241 then add these three values

123 + 203 + 241 = 567 to get the hash value.

Fold Boundary Method

Key is broken into several parts and reverse the digits in the outermost partitions and then add the partition to form the hash value.

For example : X = 123 203 241

Partition = 123, 203, 241

Reverse the boundary partition = 321, 203, 142

Add the partition = 321 + 203 + 142

Hash value = 666

(d) Pseudo Random Number Generator Method

This method generates random number given a seed as parameter and the resulting random number then scaled into the possible address range using modulo division. It must ensure that it always generates the same random value for a given key. The random number produced can be transformed to produce a valid hash value.

$X_{n+1} = A X_n$ Mod Tablesize

5.24

(e) Digit or Character Extraction Method

This method extracts the selected digits from the key and used as it is as the address or it can be reversed to give the hash value.

Example :

Map the key 123 203 241 to a range between 0 to 999.

select the digits from the positions. 2, 4, 5 and 8. Therefore hash value = 2204.

Similarly the hash value can also be obtained by considering the above digit positions and reversing it, which yields the hash value as 4022.

(f) Radix Transformation

In this method, a key is transformed into another number base to obtain the hash value.

For example :

Map the key $(8465)_{10}$ in the range 0 to 9999 using base 15.

$(8465)_{10} = (2795)_{15}$

The key 8465 is placed in the hash address 2795

5.24

(e) Digit or Character Extraction Method

This method extracts the selected digits from the key and used as it is as the address or it can be reversed to give the hash value.

Example :

Map the key 123 203 241 to a range between 0 to 999.

select the digits from the positions. 2, 4, 5 and 8. Therefore hash value = 2204.

Similarly the hash value can also be obtained by considering the above digit positions and reversing it, which yields the hash value as 4022.

(f) Radix Transformation

In this method, a key is transformed into another number base to obtain the hash value.

For example :

Map the key $(8465)_{10}$ in the range 0 to 9999 using base 15.

$(8465)_{10} = (2795)_{15}$

The key 8465 is placed in the hash address 2795

5.3.3 Applications of Hash Tables

- Database Systems
- Symbol Tables
- Data Dictionaries
- Network Processing Algorithms
- Browse Cashes

PRATHYUSHA ENGINEERING COLLEGE

COMPUTER SCIENCE AND ENGINEERI NG

CS3301-DATA STRUCTURES QUESTION BANK

UNIT I

2MARKS

1. Explain the term data structure.

The data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements. Formally data structure can be defined as a data structure is a set of domains D, a set of domains F and a set of axioms A. this triple (D,F,A) denotes the data structure d.

2. What do you mean by non-linear data structure? Give example.

The non-linear data structure is the kind of data structure in which the data may be arranged in hierarchical fashion. For example- Trees and graphs.

3. What do you linear data structure? Give example.

The linear data structure is the kind of data structure in which the data is linearly arranged. For example- stacks, queues, linked list.

4. Enlist the various operations that can be performed on data structure.

Various operations that can be performed on the data structure are

- Create
- Insertion of element
- Deletion of element
- Searching for the desired element
- Sorting the elements in the data structure
- Reversing the list of elements.

5. What is abstract data type? What are all not concerned in an ADT?

The abstract data type is a triple of D i.e. set of axioms, F-set of functions and A-Axioms in which only what is to be done is mentioned but how is to be done is not mentioned. Thus ADT is not concerned with implementation details.

6. List out the areas in which data structures are applied extensively.

Following are the areas in which data structures are applied extensively.

- Operating system- the data structures like priority queues are used for scheduling the jobs in the operating system.
- Compiler design- the tree data structure is used in parsing the source

program.

Stack data structure is used in handling recursive calls.

- Database management system- The file data structure is used in database management systems. Sorting and searching techniques can be applied on these data in the file.
- Numerical analysis package- the array is used to perform the numerical analysis on the given set of data.
- Graphics- the array and the linked list are useful in graphics applications.
- Artificial intelligence- the graph and trees are used for the applications like building expression trees, game playing.

7. What is a linked list?

A linked list is a set of nodes where each node has two fields ‘data’ and ‘link’. The data field is used to store actual piece of information and link field is used to store address of next node.

8. What are the pitfall encountered in singly linked list?

Following are the pitfall encountered in singly linked list

- The singly linked list has only forward pointer and no backward link is provided. Hence the traversing of the list is possible only in one direction. Backward traversing is not possible.
- Insertion and deletion operations are less efficient because for inserting the element at desired position the list needs to be traversed. Similarly, traversing of the list is required for locating the element which needs to be deleted.

9. Define doubly linked list.

Doubly linked list is a kind of linked list in which each node has two link fields. One link field stores the address of previous node and the other link field stores the address of the next node.

10. Write down the steps to modify a node in linked lists.

- ➢ Enter the position of the node which is to be modified.
- ➢ Enter the new value for the node to be modified.
- ➢ Search the corresponding node in the linked list.
- ➢ Replace the original value of that node by a new value.
- ➢ Display the messages as “ the node is modified”.

11. Difference between arrays and lists.

In arrays any element can be accessed randomly with the help of index of array, whereas in lists any element can be accessed by sequential access only.

Insertion and deletion of data is difficult in arrays on the other hand insertion and deletion of data is easy in lists.

12. State the properties of LIST abstract data type with suitable example.

Various properties of LIST abstract data type are

(i) It is linear data structure in which the elements are arranged adjacent to each other. (ii) It allows to store single variable polynomial.

(iii)If the LIST is implemented using dynamic memory then it is called linked list.

Example of LIST are- stacks, queues, linked list.

13. State the advantages of circular lists over doubly linked list.

In circular list the next pointer of last node points to head node, whereas in doubly linked list each node has two pointers: one previous pointer and another is next pointer. The main advantage of circular list over doubly linked list is that with the help of single pointer field we can access head node quickly. Hence some amount of memory get saved because in circular list only one pointer is reserved.

14. What are the advantages of doubly linked list over singly linked list?

The doubly linked list has two pointer fields. One field is previous link field and another is next link field. Because of these two pointer fields we can access any node efficiently whereas in singly linked list only one pointer field is there which stores forward pointer.

15. Why is the linked list used for polynomial arithmetic?

We can have separate coefficient and exponent fields for representing each term of polynomial. Hence there is no limit for exponent. We can have any number as an exponent.

16. What is the advantage of linked list over arrays?

The linked list makes use of the dynamic memory allocation. Hence the user can allocate or de allocate the memory as per his requirements. On the other hand, the array makes use of the static memory location. Hence there are chances of wastage of the memory or shortage of memory for allocation.

17. What is the circular linked list?

The circular linked list is a kind of linked list in which the last node is connected to the

first node or head node of the linked list.

18. What is the basic purpose of header of the linked list?

The header node is the very first node of the linked list. Sometimes a dummy value such -

999 is stored in the data field of header node.

This node is useful for getting the starting address of the linked list.

19. What is the advantage of an ADT?

- **Change:** the implementation of the ADT can be changed without making changes in the client program that uses the ADT.
- **Understandability:** ADT specifies what is to be done and does not specify the implementation details. Hence code becomes easy to understand due to ADT.
- **Reusability:** the ADT can be reused by some program in future.

20. What is static linked list? State any two applications of it.

- The linked list structure which can be represented using arrays is called static linked list.
- It is easy to implement, hence for creation of small databases, it is useful.
- The searching of any record is efficient, hence the applications in which the record need to be searched quickly, the static linked list are used.

16 MARKS

1. Explain the insertion operation in linked list. How nodes are inserted after a specified node.
2. Write an algorithm to insert a node at the beginning of list?
3. Discuss the merge operation in circular linked lists.
4. What are the applications of linked list in dynamic storage management?
5. How polynomial expression can be represented using linked list?
6. What are the benefit and limitations of linked list?
7. Define the deletion operation from a linked list.
8. What are the different types of data structure?
9. Explain the operation of traversing linked list. Write the algorithm and give an example.

UNIT II

2MARKS

1. Define Stack

A Stack is an ordered list in which all insertions (Push operation) and deletion (Pop operation) are made at one end, called the top. The topmost element is pointed by top. The top is initialized to -1 when the stack is created that is when the stack is empty. In a stack S = (a1,an), a1 is the bottom most element and element a is on top of element ai-1. Stack is also referred as Last In First Out (LIFO) list.

2. What are the various Operations performed on the Stack?

The various operations that are performed on the stack are

CREATE(S) – Creates S as an empty stack.

PUSH(S,X) – Adds the element X to the top of the stack. POP(S) – Deletes the top most elements from the stack. TOP(S) – returns the value of top element from the stack. ISEMTPTY(S) – returns true if Stack is empty else false. ISFULL(S) - returns true if Stack is full else false.

3.How do you test for an empty stack?

The condition for testing an empty stack is top =-1, where top is the pointer pointing to the topmost element of the stack, in the array implementation of stack. In linked list implementation of stack the condition for an empty stack is the header node link field is NULL.

4.Name two applications of stack?

Nested and Recursive functions can be implemented using stack. Conversion of Infix to Postfix expression can be implemented using stack. Evaluation of Postfix expression can be implemented using stack.

5.Define a suffix expression.

The notation used to write the operator at the end of the operands is called suffix notation.

Suffix notation format : operand operand operator

Example: ab+, where a & b are operands and '+' is addition operator.

6.What do you meant by fully parenthesized expression? Give example.

A pair of parentheses has the same parenthetical level as that of the operator to which it corresponds. Such an expression is called fully parenthesized expression.

Ex: (a+((b*c) + (d * e))

7.Write the postfix form for the expression -A+B-C+D?

A-B+C-D+

8.What are the postfix and prefix forms of the expression?

A+B*(C-

D)/(P-R)

Postfix form: ABCD-

*PR-/+ Prefix form:

+A/*B-CD-PR

9.**Explain the usage of stack in recursive algorithm implementation?**

In recursive algorithms, stack data structures is used to store the return address when a recursive call is encountered and also to store the values of all the parameters essential to the current state of the function.

10.**Define Queues.**

A Queue is an ordered list in which all insertions take place at one end called the rear, while all deletions take place at the other end called the front. Rear is initialized to -1 and front is initialized to 0. Queue is also referred as First In First Out (FIFO) list.

11.**What are the various operations performed on the Queue?**

The various operations performed on the queue are

CREATE(Q) – Creates Q as an empty Queue.

Enqueue(Q,X) – Adds the element X to the Queue.

Dequeue(Q) – Deletes a element from the Queue.

ISEMTPTY(Q) – returns true if Queue is empty else

false. ISFULL(Q) - returns true if Queue is full else

false.

12. **How do you test for an empty Queue?**

The condition for testing an empty queue is rear=front-1. In linked list implementation of queue the condition for an empty queue is the header node link field is NULL.

13.**Write down the function to insert an element into a queue, in which the queue is implemented as an array. (May 10)**

Q – Queue

X – element to added to the queue Q

IsFull(Q) – Checks and true if Queue Q is full

Q->Size - Number of elements in the queue

Q->Rear – Points to last element of the queue Q

Q->Array – array used to store queue elements

```
void enqueue (int X, Queue Q) {
        if(IsFull(Q))
            Error ("Full
              queue");
        else       {
                Q->Size++;
                    Q->Rear = Q->Rear+1;
                    Q->Array[ Q->Rear ]=X;
                } }
```

14.Define Dequeue.

Deque stands for Double ended queue. It is a linear list in which insertions and deletion are made from either end of the queue structure.

15.**Define Circular Queue.**

Another representation of a queue, which prevents an excessive use of memory by arranging elements/ nodes $Q_1,Q_2,\ldots Q_n$ in a circular fashion. That is, it is the queue, which wraps around upon reaching the end of the queue

16 MARKS

1. Write an algorithm for Push and Pop operations on Stack using Linked list. (8)
2. Explain the linked list implementation of stack ADT in detail?
3. Define an efficient representation of two stacks in a given area of memory with n words and explain.
4. Explain linear linked implementation of Stack and Queue?
    a. Write an ADT to implement stack of size N using an array. The elements in the stack are to be integers. The operations to be supported are PUSH, POP and DISPLAY. Take into account the exceptions of stack overflow and stack underflow. (8)
    b. A circular queue has a size of 5 and has 3 elements 10,20 and 40 where F=2 and R=4. After inserting 50 and 60, what is the value of F and R. Trying to insert 30 at this stage what happens? Delete 2 elements from the

queue and insert 70, 80 &
90. Show the sequence of steps with necessary diagrams with the value of F & R. (8 Marks)

5. Write the algorithm for converting infix expression to postfix (polish) expression?
6. Explain in detail about priority queue ADT in detail?

7.Write a function called 'push' that takes two parameters: an integer variable and a stack into
which it would push this element and returns a 1 or a 0 to show success of addition or failure.

8. What is a DeQueue? Explain its operation with example?
9. Explain the array implementation of queue ADT in detail?
10. Explain the addition and deletion operations performed on a circular queue with necessary algorithms.(8) (Nov 09)

UNIT III

1.Define tree

Trees are non-liner data structure, which is used to store data items in a shorted sequence. It represents any hierarchical relationship between any data Item. It is a collection of nodes, which has a distinguish node called the root and zero or more non-empty sub trees T1, T2,….Tk. each of which are connected by a directed edge from the root.

2. Define Height of tree?

The height of n is the length of the longest path from root to a leaf. Thus all leaves have height zero. The height of a tree is equal to a height of a root.

3. Define Depth of tree?

For any node n, the depth of n is the length of the unique path from the root to node n. Thus for a root the depth is always zero.

4. What is the length of the path in a tree?

The length of the path is the number of edges on the path. In a tree there is exactly one path form the root to each node.

5. Define sibling?

Nodes with the same parent are called siblings. The nodes with common parents are called siblings.

6. Define binary tree?

A Binary tree is a finite set of data items which is either empty or consists of a single

item called root and two disjoin binary trees called left sub tree max degree of any node is two.

7. What are the two methods of binary tree implementation?

Two methods to implement a binary tree are,

a. Linear representation.

b. Linked representation

8. What are the applications of binary tree?

Binary tree is used in data processing.

a. File index schemes

b. Hierarchical database management system

9. List out few of the Application of tree data-structure?

- Ø The manipulation of Arithmetic expression
- Ø Used for Searching Operation
- Ø Used to implement the file system of several popular operating systems
- Ø Symbol Table construction
- Ø Syntax analysis

10. Define expression tree?

Expression tree is also a binary tree in which the leafs terminal nodes or operands and non-terminal intermediate nodes are operators used for traversal.

11. Define tree traversal and mention the type of traversals?

Visiting of each and every node in the tree exactly is called as tree

traversal. Three types of tree traversal

1. Inorder traversal
2. Preoder traversal
3. Postorder traversal.

12. Define in -order traversal?

In-order traversal entails the following steps;

a. Traverse the left subtree

b. Visit the root node

c. Traverse the right subtree

13. Define threaded binary tree.

A binary tree is threaded by making all right child pointers that would normally be null point to the in order successor of the node, and all left child pointers that would normally be null
point to the in order predecessor of the node.

14. What are the types of threaded binary tree?

i. Right-in threaded
binary tree ii. Left-in
threaded binary tree iii.
Fully-in threaded binary tree

15. Define Binary Search Tree.

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X and the values of all the keys in its right
subtree are larger than the key value in X.

16.What is AVL Tree?

AVL stands for Adelson-Velskii and Landis. An AVL tree is a binary search tree which has the following properties:

1.The sub-trees of every node differ in height by at most one.

2.Every sub-tree is an AVL tree.

Search time is O(logn). Addition and deletion operations also take O(logn) time.

17. List out the steps involved in deleting a node from a binary search tree.

- Deleting a node is a leaf node (ie) No children
- Deleting a node with one child.
- Deleting a node with two Childs.

18. What is 'B' Tree?

A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time. Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.

Important properties of a B-tree:

- B-tree nodes have many more than two children.
- A B-tree node may contain more than just a single element.

19. What is binomial heaps?

A binomial heap is a collection of binomial trees that satisfies the following binomial-heap properties:

1. No two binomial trees in the collection have the same size.
2. Each node in each tree has a key.
3. Each binomial tree in the collection is heap-ordered in the sense that each non-root has a key strictly less than the key of its parent.The number of trees in a binomial heap is O(log n).

20. Define complete binary tree.

If all its levels, possible except the last, have maximum number of nodes and if all the nodes in the last level appear as far left as possible.

16 MARKS

1.Explain the AVL tree insertion and deletion with suitable example.

2.Describe the algorithms used to perform single and double rotation on AVL tree.

3.Explain about B-Tree with suitable example.

4.Explain about B+ trees with suitable algorithm.

5.Write short notes on

I Binomial heaps ii. Fibonacci heaps

6.Explain the tree traversal techniques with an example.

7.Construct an expression tree for the expression (a+b*c) + ((d*e+f)*g). Give the outputs when you apply inorder, preorder and postorder traversals.

8.How to insert and delete an element into a binary search tree and write down the code for the insertion routine with an example.

9.What are threaded binary tree? Write an algorithm for inserting a node in a threaded binary tree.

10.Create a binary search tree for the following numbers start from an empty binary search tree.

45,26,10,60,70,30,40 Delete keys 10,60 and 45 one after the other and show the trees at each stage.

UNIT IV

PART A

1. Write the definition of weighted graph?

A graph in which weights are assigned to every edge is called a weighted graph.

2. Define Graph?

A graph G consist of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set of edges E to set of pairs of elements of V. It can also be represented as G=(V, E).

3.**Define adjacency matrix?**

The adjacency matrix is an n x n matrix A whose elements aij are given by

Aij =1 if(vi,vj)exists, otherwise 0

4.**Define adjacent nodes?**

Any two nodes, which are connected by an edge in a graph, are called adjacent nodes. For example, if an edge x E is associated with a pair of nodes (u,v) where u, v V, then we say that the edge x connects the nodes u and v.

5.**What is a directed graph?**

A graph in which every edge is directed is called a directed graph.

6.**What is an undirected graph?**

A graph in which every edge is undirected is called an undirected graph.

7.**What is a loop?**

An edge of a graph, which connects to itself, is called a loop or sling.

8.**What is a simple graph?**

A simple graph is a graph, which has not more than one edge between a pair of nodes.

9.**What is a weighted graph?**

A graph in which weights are assigned to every edge is called a weighted graph.

10.**Define indegree and out degree of a graph?**

In a directed graph, for any node v, the number of edges, which have v as their initial node, is called the out degree of the node v.

Outdegree: Number of edges having the node v as root node is the outdegree of the node v.

11.Define path in a graph?

The path in a graph is the route taken to reach terminal node from a starting node.

12.**What is a simple path?**

i. A path in a diagram in which the edges are distinct is called a simple path. ii. It is also called as edge simple.

13.**What is a cycle or a circuit?**

A path which originates and ends in the same node is called a cycle or circuit.

14.**What is an acyclic graph?**

A simple diagram, which does not have any cycles, is called an acyclic graph.

15.**What is meant by strongly connected in a graph?**

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

16.**When a graph said to be weakly connected?**

When a directed graph is not strongly connected but the underlying graph is connected, then the graph is said to be weakly connected.

17.**Name the different ways of representing a graph? Give examples (Nov 10)**

a. Adjacency

matrix b.

Adjacency list

18.**What is an undirected acyclic graph?**

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph. It is also called as undirected forest.

19.**What is meant by depth?**

The depth of a list is the maximum level attributed to any element with in the list or with in any sub list in the list.

20.**What is the use of BFS?**

BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph. The shortest distance is the minimum number of edges traversed in order to travel from the start node the specific node being examined.

21.**What is topological sort?**

It is an ordering of the vertices in a directed acyclic graph, such that: If there is a path from u to v, then v appears after u in the ordering.

22.Write BFS algorithm

1. Initialize the first node’s dist number and place in queue
2. Repeat until all nodes have been examined
3. Remove current node to be examined from queue
4. Find all unlabeled nodes adjacent to current node
5. If this is an unvisited node label it and add it to the queue
6. Finished.

23.**Define biconnected graph?**

A graph is called biconnected if there is no single node whose removal causes the graph to break into two or more pieces. A node whose removal causes the graph to become disconnected is called a cut vertex.

24.**What are the two traversal strategies used in traversing a graph?**

a. Breadth first search

b. Depth first search

25.**Articulation Points (or Cut Vertices) in a Graph**

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

Following are some example graphs with articulation points encircled with red color.

16 MARKS

1. Explain the various representation of graph with example in detail?
2. Explain Breadth First Search algorithm with example?
3. Explain Depth first and breadth first traversal?
4. What is topological sort? Write an algorithm to perform topological sort?(8) (Nov 09)
5. (i) write an algorithm to determine the biconnected components in the given graph. (10) (may 10)

   (ii)determine the biconnected components in a graph. (6)
6. Explain the various applications of Graphs.

UNIT – V

2 MARKS

1.What is meant by Sorting?

Sorting is ordering of data in an increasing or decreasing fashion according to some linear relationship among the data items.

2. List the different sorting algorithms.

- Bubble sort
- Selection sort

- Insertion sort
- Shell sort
- Quick sort
- Radix sort
- Heap sort
- Merge sort

3. Why bubble sort is called so?

The bubble sort gets its name because as array elements are sorted they gradually

"bubble" to their proper positions, like bubbles rising in a glass of soda.

4. State the logic of bubble sort algorithm.

The bubble sort repeatedly compares adjacent elements of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two

elements of the array are compared and swapped if out of order.

5. What number is always sorted to the top of the list by each pass of the Bubble sort algorithm?

Each pass through the list places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs.

6. When does the Bubble Sort Algorithm stop?

The bubble sort stops when it examines the entire array and finds that no "swaps" are needed. The bubble sort keeps track of the occurring swaps by the use of a flag.

7. State the logic of selection sort algorithm.

It finds the lowest value from the collection and moves it to the left. This is repeated until the complete collection is sorted.

8. What is the output of selection sort after the 2nd iteration given the following sequence? 16 3 46 9 28 14

Ans: 3 9 46 16 28 14

9.How does insertion sort algorithm work?

In every iteration an element is compared with all the elements before it. While comparing if it is found that the element can be inserted at a suitable position, then space is created for it by shifting the other elements one position up and inserts the desired

element at the suitable position. This procedure is repeated for all the elements in the list until we get the sorted elements.

10.What operation does the insertion sort use to move numbers from the unsorted section to the sorted section of the list?

The Insertion Sort uses the swap operation since it is ordering numbers within a single list.

11. How many key comparisons and assignments an insertion sort makes in its worst case?

The worst case performance in insertion sort occurs when the elements of the input array are in descending order. In that case, the first pass requires one comparison, the second pass requires two comparisons, third pass three comparisons,….kth pass requires (k-1), and finally the last pass requires (n-1) comparisons. Therefore, total numbers of comparisons are:

$f(n) = 1+2+3+\ldots\ldots+(n-k)+\ldots..+(n-2)+(n-1) = n(n-1)/2 = O(n2)$

12. Which sorting algorithm is best if the list is already sorted? Why?

Insertion sort as there is no movement of data if the list is already sorted and complexity is of the order O(N).

13. Which sorting algorithm is easily adaptable to singly linked lists? Why?

Insertion sort is easily adaptable to singly linked list. In this method there is an array link of pointers, one for each of the original array elements. Thus the array can be thought of as a linear link list pointed to by an external pointer first initialized to 0. To insert the k^{th} element the linked list is traversed until the proper position for x[k] is found, or until the end of the list is reached. At that point x[k] can be inserted into the list by merely adjusting the pointers without shifting any elements in the array which reduces insertion time.

14. Why Shell Sort is known diminishing increment sort?

The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared. In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted.

15. Which of the following sorting methods would be especially suitable to sort alist L consisting of a sorted list followed by a few "random" elements?

Quick sort is suitable to sort a list L consisting of a sorted list followed by a few "random" elements.

16.What is the output of quick sort after the 3rd iteration given the following sequence?

24 56 47 35 10 90 82 31

Pass 1:- (10) 24 (56 47 35 90 82 31) Pass 2:- 10 24 (56 47 35 90 82 31) Pass 3:- 10 24 (47 35 31) 56 (90 82)

17.Mention the different ways to select a pivot element.

The different ways to select a pivot element are

- Pick the first element as pivot
- Pick the last element as pivot
- Pick the Middle element as pivot
- Median-of-three elements
- Pick three elements, and find the median x of these elements
- Use that median as the pivot.
- Randomly pick an element as pivot.

18.What is divide-and-conquer strategy?

- Divide a problem into two or more sub problems
- Solve the sub problems recursively
- Obtain solution to original problem by combining these solutions

19. Compare quick sort and merge sort.

Quicksort has a best-case linear performance when the input is sorted, or nearly sorted. It has a worst-case quadratic performance when the input is sorted in reverse, or nearly sorted in reverse.

Merge sort performance is much more constrained and predictable than the performance of quicksort. The price for that reliability is that the average case of merge sort is slower than the average case of quicksort because the constant factor of merge sort is larger.

20.Define Searching.

Searching for data is one of the fundamental fields of computing. Often, the difference between a fast program and a slow one is the use of a good algorithm for the data set. Naturally,

the use of a hash table or binary search tree will result in more efficient searching, but more often than not an array or linked list will be used. It is necessary to understand good ways of searching data structures not designed to support efficient search.

21.What is linear search?

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

22.**What is Binary search?**

A binary search, also called a dichotomizing search, is a digital scheme for locating a specific object in a large set. Each object in the set is given a key. The number of keys is always a power of 2. If there are 32 items in a list, for example, they might be numbered 0 through 31 (binary
00000 through 11111). If there are, say, only 29 items, they can be numbered 0 through 28 (binary 00000 through 11100), with the numbers 29 through31 (binary 11101, 11110, and
11111) as dummy
keys.

23.Define hash function?

Hash function takes an identifier and computes the address of that identifier in the hash table using some function.

24.Why do we need a Hash function as a data structure as compared to any other data structure? (may 10)

Hashing is a technique used for performing insertions, deletions, and finds in constant average time.

25.What are the important factors to be considered in designing the hash function? (Nov10)

- To avoid lot of collision the table size should be prime
- For string data if keys are very long, the hash function will take long to compute.

26.. What do you mean by hash table?

The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value. Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell.

27. **What do you mean by hash function?**

A hash function is a key to address transformation which acts upon a given key to compute the relative position of the key in an array. The choice of hash function should be simple and it must distribute the data evenly. A simple hash function is hash_key=key mod tablesize.

28.**What do you mean by separate chaining?**

Separate chaining is a collision resolution technique to keep the list of all elements that hash to the same value. This is called separate chaining because each hash table element is a separate chain (linked list). Each linked list contains all the elements whose keys hash to the same index.

16 MARKS

1. Write an algorithm to implement Bubble sort with suitable example.
2. Explain any two techniques to overcome hash collision.
3. Write an algorithm to implement insertion sort with suitable example.
4. Write an algorithm to implement selection sort with suitable example.
5. Write an algorithm to implement radix sort with suitable example.
6. Write an algorithm for binary search with suitable example.
7. Discuss the common collision resolution strategies used in closed hashing system.
8. Given the input { 4371, 1323, 6173, 4199, 4344, 9679, 1989 } and a hash function of h(X)=X (mod 10) show the resulting:
    a. Separate Chaining hash table
    b. Open addressing hash table using linear probing
9. Explain Re-hashing and Extendible hashing.
10. Show the result of inserting the keys 2,3,5,7,11,13,15,6,4 into an initially empty extendible hashing data structure with M=3. (8) (Nov 10)
11. what are the advantages and disadvantages of various collision resolution strategies? (6)

ALL THE BEST