Module-V

Topics **Page Number**

SORTING TECHNIQUES

SORTING TECHNIQUES

One of the fundamental problems of computer science is ordering a list of items. There are solutions to this problem, known as sorting algorithms.

5.1 **Selection Sort**

This type of sorting is also called exchange sort. In this method we have to find the smallest element in the array and this element is stored in the first position. Then we have to find the second smallest element and it is stored in the second position, and so on.

Find the position L of the smallest among the elements A (1), A (2), A (3),…. A (n). Interchange A (L) with A (1).

Find the position L of the smallest among the elements A (2), A(3),…. A (n). Interchange A (L) with A (2).

Find the position L of the smallest among the elements A (n-1),…. A (n). Interchange A (L) with A (n-1).

Algorithm

Steps:

1. Start
2. Read the number of elements in the array say n
3. Read the elements in the array A
4. Assign the value of the variable i=0
5. Assign lowest=1
6. Repeat step 7 to 14 until i<n
7. Assign j=i+1
8. Repeat step 9 to 10 until j<n
9. If(A[j]<A[i])
    a) lowest=j
10. Increment j by 1
11. Assign temp=a[i]
12. Assign A[i]=A[lowest]
13. assign A[lowest]=temp
14. Increment I by 1
15. Stop

Program 5.1

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

```
                    a[i]=a[j];
                    a[j] =temp
                }

}
```

Example:

Original Table	: 19 13 05 27 01 26 31 16 02 09 11 21
After pass1	: 01 13 05 27 19 26 31 16 02 09 11 21
After pass2	: 01 02 05 27 19 26 31 16 13 09 11 21
After pass3	: 01 02 05 27 19 26 31 16 13 09 11 21
After pass4	: 01 02 05 09 19 26 31 16 13 27 11 21
After pass5	: 01 02 05 09 11 26 31 16 13 27 19 21
After pass6	: 01 02 05 09 11 13 31 16 26 27 19 21
After pass7	: 01 02 05 09 11 13 16 31 26 27 19 21
After pass8	: 01 02 05 09 11 13 16 19 26 27 31 21
After pass9	: 01 02 05 09 11 13 16 19 21 27 31 26
After pass10	: 01 02 05 09 11 13 16 19 21 26 31 27
After pass11	: 01 02 05 09 11 13 16 19 21 26 27 31

Time Complexity of selection sort

Line 2: Executed (n- 1) times.
Lines 3 & 4: Executed (n - 1) + (n - 2) + ... + 1 = n(n - 1) / 2 times.
Lines 6, 7 & 8: Executed between 0 and n(n - 1) / 2 times depending on how often a[i] > a[j].

Value of i	No. of comparisons
0	n-1
1	n-2
.	.
.	.
n-2	1

$T(n)= (n-1) + (n-2) + \dots + 1$
$= n(n-1)/2 \quad = \mathbf{O(n^2)}$

5.2 **Bubble Sort**

This algorithm compares the key to the data elements and swaps the elements if they are not in the desired order. Since, swapping occurs only among successive elements of the data structure only one element is placed in its sorted place after each pass. Once the elements area sorted they are considered for comparison in successive pass.
In this we can fix the last element by comparing 1st element with 2nd, 2nd with 3rd … (n-1)st with nth . Repeat the procedure to fix all the remaining elements.

Algorithm

Steps:

1. Start
2. Read the number of elements in the array say n
3. Read the number of elements in the array A

4. Initialize I=0
5. Repeat step 6 to 10 until I<n
6. Assign j=0
7. Repeat step 8 to 9 until j<n-(I+1)
8. If(a[j]>a[j+1])
    a) Assign temp=x[j]
    b) AssignA[j]=A[j+1]
    c) Assign A{j+1]=temp
9. Increment j by 1
10. Increment I by 1
11. Stop

Program 5.2

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

Example:

Original Array: 6 2 4 7 1 3 8 5

First Iteration

A[0]>A[1] i.e., 6>2 so we have to swap these values and the array becomes
2 6 4 7 1 3 8 5
A[1]>A[2] then swap these elements and the array becomes
2 4 6 7 1 3 8 5
A[2]<A[3] so the condition false and the array becomes
2 4 6 7 1 3 8 5
A[3]>A[4] then swap these elements and the array becomes
2 4 6 1 7 3 8 5
A[4]>A[5] then swap these elements and the array becomes
2 4 6 1 3 7 8 5
A[5]<A[6] so the condition false and the array becomes
2 4 6 1 3 7 8 5
A[6]>A[7] then swap these elements and the array becomes
2 4 6 1 3 7 5 8

After the **second iteration** the array A contains 2 4 1 3 6 5 7 8
After the **third iteration** the array A contains 2 1 3 4 5 6 7 8
After the **fourth iteration** the array A contains 1 2 3 4 5 6 7 8
After the **fifth iteration** the array A contains 1 2 3 4 5 6 7 8

Time Complexity

Value of i	No. of comparisons
1	n-1
2	n-2
.	
.	
n-1	1

i.e., (n-1) + (n-2) + … + 1
= n(n-1)/2
= **$O(n^2)$**

We can modify the above program such that it gives a complexity of O(n) for a sorted list. Modified program is given below.

Algorithm

Steps:

1. Start
2. Read the number of elements in the array say n
3. Read the number of elements in the array A
4. Initialize flag=1,and I=0
5. Repeat step 6 to 10 until I<n and flag=1
6. Assign swap=0 and j=0
7. Repeat step 8 to 9 until j<n-(I+1)
8. If(a[j]>a[j+1])
    a. Assign temp=x[j]
    b. AssignA[j]=A[j+1]
    c. Assign A{j+1]=temp
    d. Assign flag=1
9. Increment j by 1
10. Increment I by 1
11. Stop

Program 5.3

```
Void bubble_sort_modified( int a, int n)
{
    int flag = 1;
    for(i=0; i<n-1 && flag==1; i++)
    {
            flag = 0;
            for(j=0; j<n-i-1; j++)
                    if(a[j] >a[j+1])
                    {
                            swap(a[j],[j+1]);
                            flag = 1;
                    }
    }
}
```

Time Complexity

If the list is sorted, it won't enter in to if loop. Hence the flag will be false and the program comes out of for loop during the next pass. So complexity is **O(n).**

5.3 **Insertion Sort**

In this sorting technique insert the element into a sequence of ordered elements, $e_1,e_2..e_i$ such that resultant i+1 are also ordered. The general idea of the insertion sort method is that for each element, find the slot where it belongs.

This algorithm first orders A[0] and A[1] by inserting A[1] in front of A[0] if A[0]>A[1]. Using this ordered list, the rest of the data elements are iteratively inserted in the ordered list one at a time. After inserting an element A[k] then A[0] to A[k] are sorted.

Algorithm

Steps:

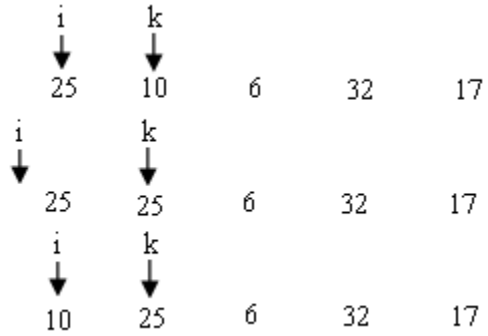
1. Start
2. Read the number of elements in the array say n
3. Read the number of elements in the array A
4. Declare another array S
5. Assign S[0]=A[0], k=1 and i=k-1
6. Repeat step 7 to 9 until k<n
7. Assign temp=k-1
8. Repeat the following steps until (S[i]>temp and i>=0)
    a) S[i+1]=S[i]
    b) Decrement i by 1
9. Increment k by 1
10. Assign k=0
11. Repeat the steps 12 to 13 until k<n
12. Assign X[k]=S[k]
13. Increment k by 1
14. Stop

Program 5.4

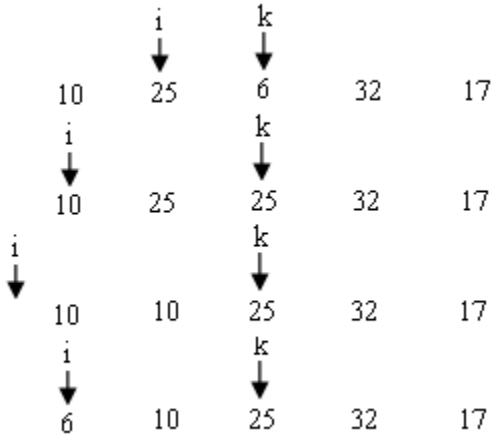
```
Void insertion_sort (int x[], int n)
{
    int i,k,y;
    for(k=1; k<n; k++)
    {
            y = x[k];
            for(i=k-1; i>=0 && y<x[i]; i--)
                    x[i+1] = x[i];
            x[i+1] = y;
    }
}
```

Example:
Original Array 25,10,6,32,17

Fix 10 in the current position by shifting all elements greater than 10 occurring before 10 in the list.



Fix 6 as given below



Similarly fix all the other elements.

Time Complexity
Case I *(Best case – List is already sorted)*:
Only One comparison is needed on each pass. Hence the complexity is **O(n).**

Case II *(Worse case – List is sorted in reverse order)*:

No. of comparison = $n + (n-1) + (n-2) + \ldots + 2$
= $n(n+1)/2 - 1$
= **$O(n^2)$.**

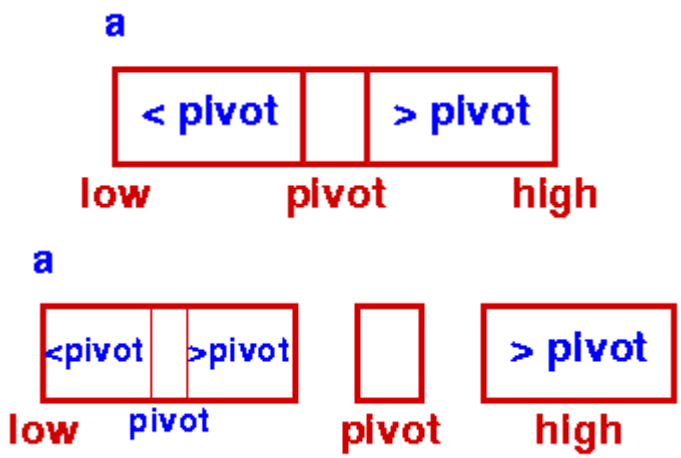
5.4 **Quick Sort**

Method

- Let **a** be an array with n elements.
- Select an element 'x' from any position, called a **pivot**, from the list.
- Partition **a** into two, so that x is placed in position j such that
    1. Elements from position 0 to j-1 is less than x
    2. Elements from position j+1 to n is greater than x

This is called a partition operation

- After this partitioning, the pivot is in its final position.
- Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

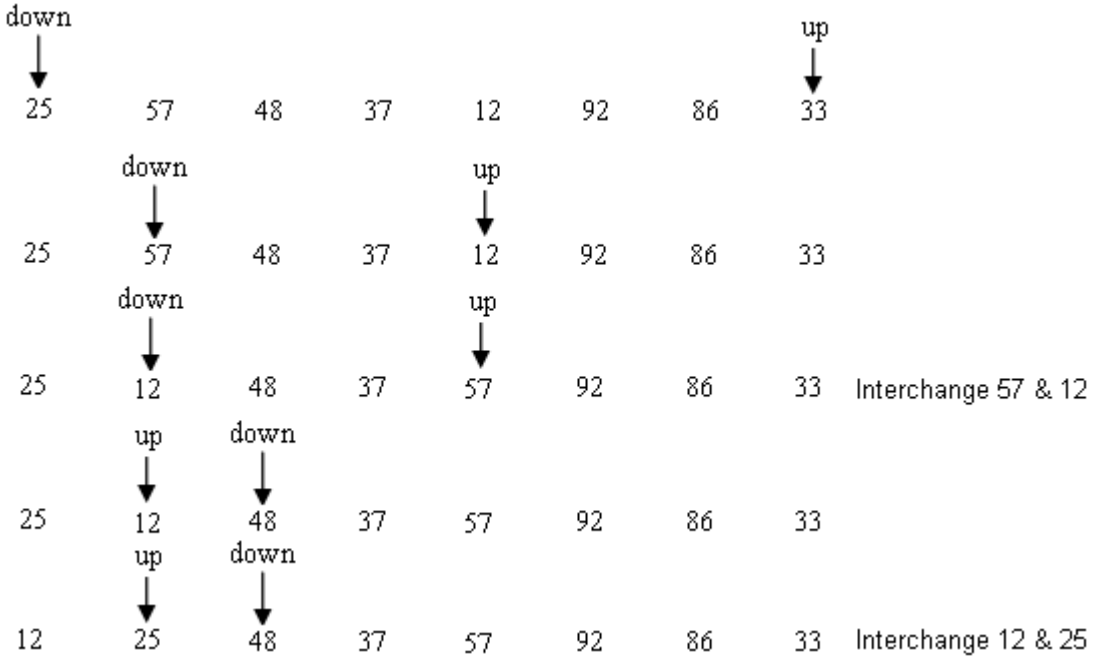


Program 5.5

```
Void quick_sort ( int a, int low, int high)
{
    if(low<high)
    {
            i=low; j=high+1;       k=a[low];
            do{
                    do
                            i++;
                    while(a[i]<=k);
                    do
                            j--;
                    while(a[j]>=k);
                    if (i<j)
                            swap(a[i],a[j]);
            }
            while(i<=j);
            swap(a[j],a[low]);
            quick_sort(a,low,j-1);
            quick_sort(a,j+1,high);
    }//if
}
```

Example:

Consider 8 elements 25,57,48,37,12,92,86,33



Split into two partitions as given below.

[12] 25 [48 37 57 92 86 33]

Perform quick sort with each partition and at the end we will the sorted array.

Time Complexity

***Case1*: Avearage case**

Assume that the file is of size n, and each time after fixing an element the file is divided into equal partitions.

$T(n)= 2T(n/2)+O(n)$(to find the pivote element)

$= 4\ T(n/4)+2\ O(n)$

…….

$=2^k\ T(n/2^k)+k*O(n)$ this iteration continued till $2^k=n$. so $k=\log_2 n$

$=2^{\log n} T(1) +\log n*O(n)$

$= O(n\log_2 n)$.

***CaseI1 (worse case)*:**

Assume that each time after fixing an element the file is divided into a smaller partition and a bigger partition. Say, during i^{th} partition it is divided into 1 and (i-1).

No. of comparison = $n + (n-1) + (n-2) + \ldots + 2$

= $n(n+1)/2 - 1$

= $O(n^2)$

5.5 Merge Sort

- Sort two sub files into a single array.
- Select the size of the sub file as multiples of 2, starting from 2^0.

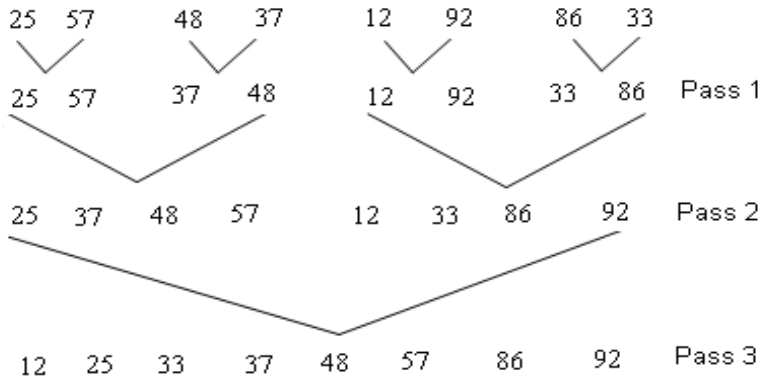
Program 5.6

```
Void Merge_sort ( int x[], int n)
{
    int aux[],i,j,k,l1,u1,l2,u2,size;

    size = 1;
    while(size<n)
    {
        l1=0; k =0;
        while(l1+size<n)
        {
            l2 = l1+size;      u2= l2-1;
            u2 = (l2+size-1<n) ? l2+size-1 : n-1;
            for(i=l1,j= l2; i<= u1 && j<= u2; k++)
                if (x[i]<=x[j])
                    aux[k] = x[i++];
                else
                    aux[k] = x[j++];
            for(; i<= u1; k++)
                    aux[k]=x[i++];
            for(; j<= u2; k++)
                    aux[k]=x[j++];
            l1= u2+1;
        }
        for(i= l1; k<n; i++)
                aux[k++]=x[i];
        for(i=0; i<n; i++)
                x[i]=aux[i];
        size *=2;
    }
}
```

Example:

Consider the array of 8 elements 25,57,48,37,12,92,86,33



Time Complexity

There are $\log_2 n$ passes, each with n comparisons. Hence the complexity is **$O(n\log_2 n)$** for all cases. But Quick sort is considered better than Merge sort even though its worse case complexity is $O(n^2)$ because Merge sort has twice as many assignment as Quick sort. Also Merge sort needs additional space to store the auxiliary array.

5.6 Radix Sort

- Based on the values of the actual digits in the positional representation of the numbers being considered.
- Partition the numbers into 10 groups depending on the least significant positional value in the order of there arrival. Place the element in the queue for the corresponding digit.
- Restore the elements in each queue to the original file starting with queue for number 0.
- Continue the process till the most significant position.

Program 5.7

```
Void Radix_sort ( int x[], int n)
{
    int front[10],rear[10];
    struct{
            int info;
            int next;
    }node[10];
    int exp,first,i,j,k,p,q,y;

    for(i=0; i<n-1; i++)
    {
            node[i].info = x[i];
            node[i].next = i+1;
    }

    node[n-1].info = x[n-1];
    node[n-1].next = -1;
    first = 0;
    for(k=1; k<3; k++)
```

```
        {
                for(i=0; i<n; i++)
                {
                        front[i] = -1;
                        rear[i] = -1;
                }

                while(first != -1)
                {
                        p = first;
                        first = node[first].next;
                        y = node[p].info;
                        exp = power(10,k-1);
                        j = (y/exp)%10;
                        q = rear[j];
                        if(q==-1)
                                front[j] = p;
                        else
                                node[q].next = p;
                        rear[j] = p;
                }
                for(j=0; j<10 && front[j]==-1; j++);
                first = front[j];
                while(j<=9)
                {
                        for(i=j+1;  i<10 && front[i]==-1;  i++);
                        if(i<=9)
                        {
                                p=i;
                                node[rear[j]].next = front[i];
                        }
                        j=i;
                }
                node[rear[p]].next = -1;
        }
        for(i=0; i<n; i++)
        {
                x[i] = node[first].info;
                first = node[first].next;
        }
    }
```

Example:
Consider the array of elements 624 852 426 987 269 146 415 301 730 78 593

Queue	pass 1	pass 2	pass 3
0	730	301	78
1	301	415	146

2	852	624, 426	269
3	593	730	301
4	624	146	415, 426
5	415	852	593
6	426, 146	269	624
7	987	78	730
8	78	987	852
9	269	593	987

Time Complexity

- Depends on number of digits in the numbers (m) and number of elements ion the array (n).

  Outer loop executes m times and the inner loop executes n times, once for each element. Hence the complexity on **O(mn**) or m is approximately equal to **$\log_2 n$.**
- So time complexity of Radix sort is **O ($n\log_2 n$).**

5.7 Heap Sort

1. Create a complete binary tree for the elements given.
2. Convert the tree to a heap. i.e., a complete binary tree with the property that value of each node is at least as large as value of the child nodes.
3. Output sequence is generates in decreasing order by successively outputting the root and reconstructing the remaining into a heap.
4. Time complexity of heap sort is O($n\log_2 n$)

Program 5.8

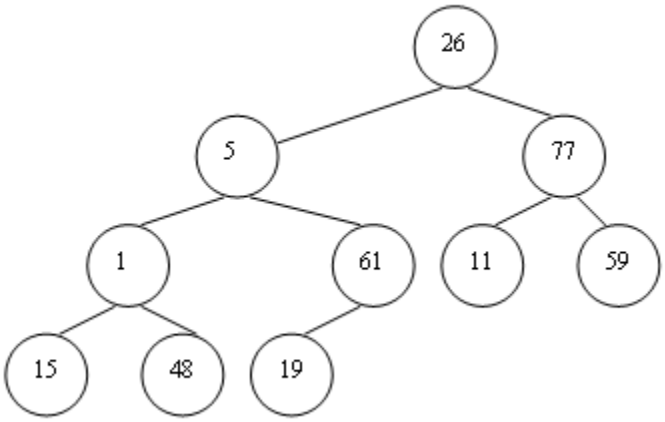
```
Void Heap_sort ( int x[], int n)
{
    int i;
    for(i=n/2; i>=1; i--)
            adjust(a,i,n);
    for(i=n-1; i>=1; i--)
    {
            interchange(a[i+1],a[1]);
            adjust(a,1,i);
    }
}
Void adjust(int a[], int i; int n)
{
    int j,k,done,r;
    done = 0;
    r = a[i];
    k = a[i];
    j = 2*i;
    while(j<=n && !done)
    {
            if (j<n)
```

```
                if(a[j] < a[j+1])
                        j = j+1;
        if (k>=a[j])
                done = 1;
        else
        {
                a[j div 2] = a[j];
                j = 2*j;
        }
    }
    a[j/ 2] = r;
}
```

Example:

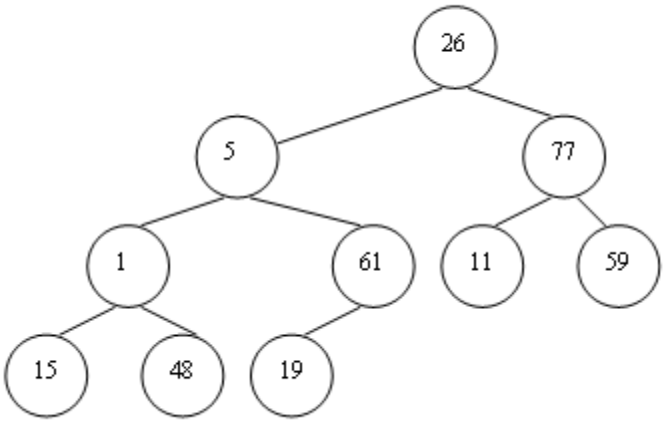
Consider 10 elements 26,5,77,1,61,11,59,15,48,19.

Form a complete binary tree with these elements as given below.

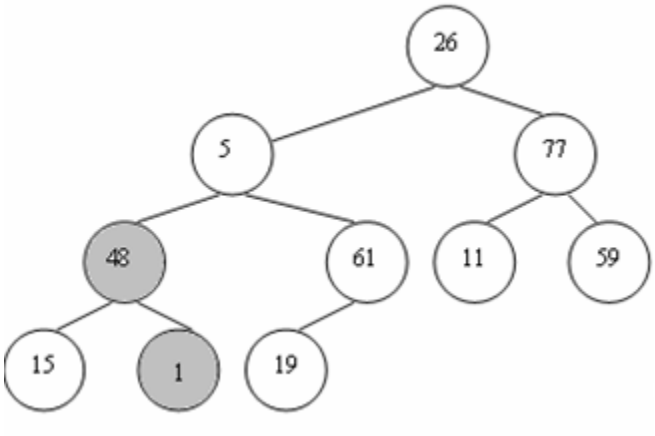


Create a heap by starting from the last parent.

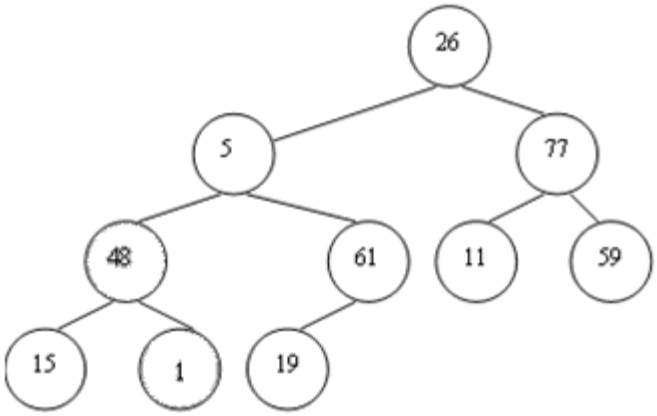
Adjusting 61 (No Change)



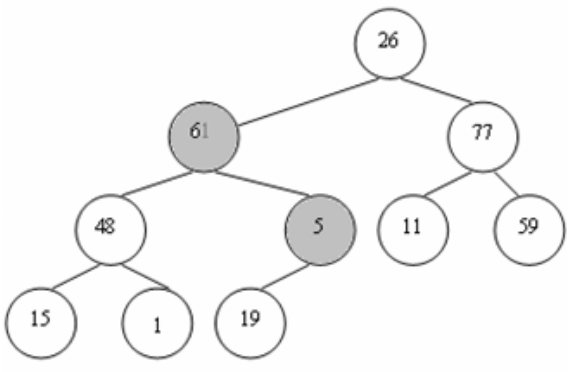
Adjusting 1

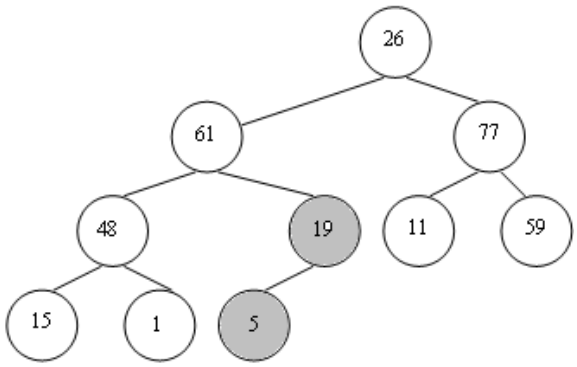


Adjusting 77 (No Change)

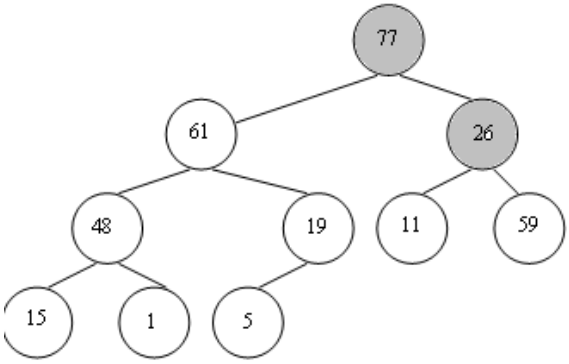


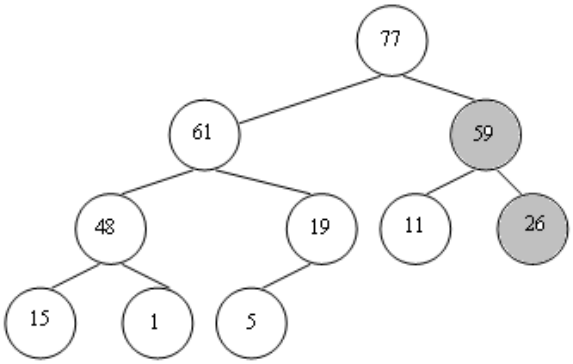
Adjusting 5





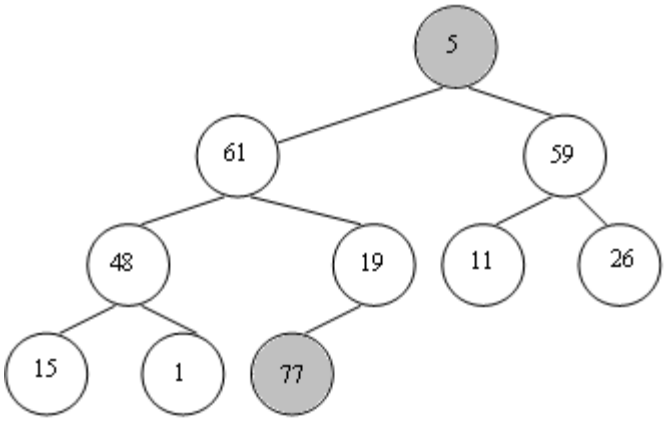
Adjusting 26



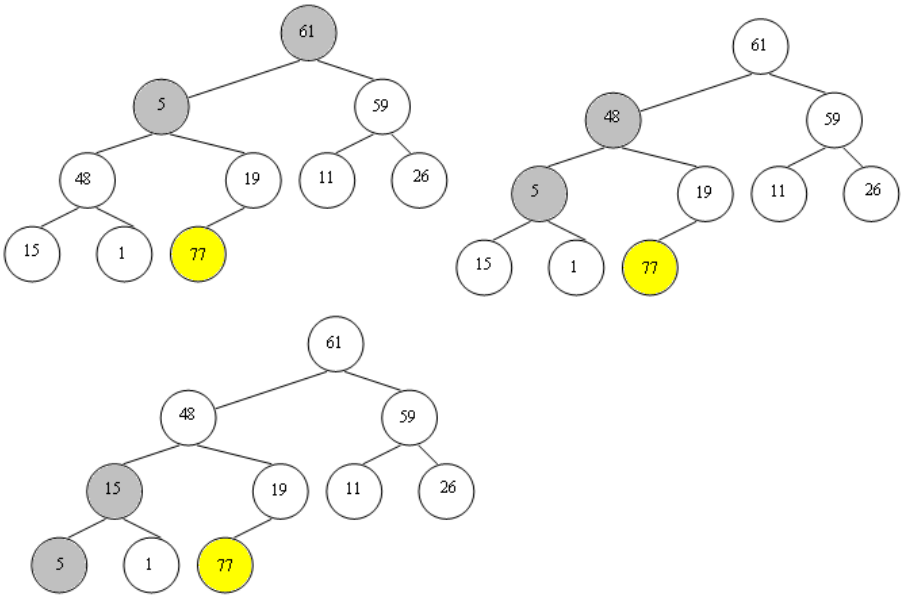


After building a heap tree Interchange root and the last element.

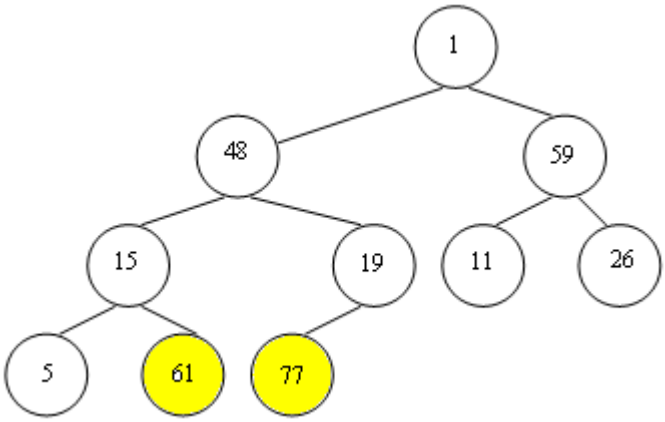
Interchange 77 and 5 (fix 77).



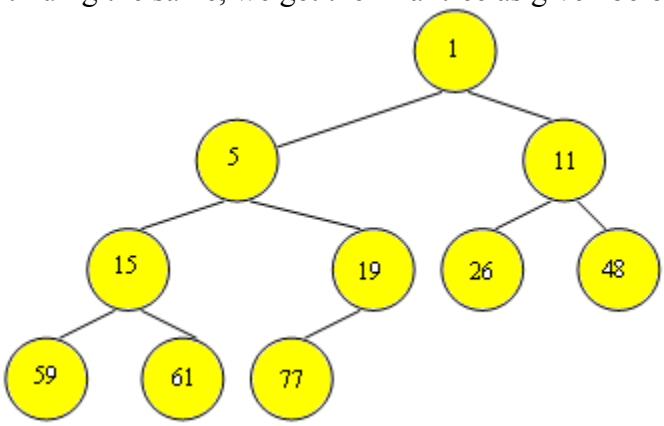
Adjust 5 which is the current root



Interchange 61 and 1 (fix 61).



Continuing the same, we get the final tree as given below.



Sorted list is 77 61 59 48 26 19 15 11 5 1

Time Complexity

At each step of the "adjust" algorithm, a node is compared to its children and one of them is chosen as the next root. We drop one level of the tree at each step of this process. Since this is a complete binary tree, there are at most log (n) levels. Thus, the worst case time complexity of "adjust" is O (log(n)). In the Heap sort algorithm, we simply use "adjust" (n/2 times in Phase 1 and (n-1) times in Phase 2. Thus, Heap Sort has a worst-case time complexity of O (n*log (n)).

Module-IV

TREES

Tree is a non linear data structure. The elements appear in anon linear fashion, which require two dimensional representations. Using tree it is easy to organize hierarchical representation of objects.

Tree is efficient for maintaining and manipulating data, where the hierarchy of relationship among the data is to be preserved.

Definition

Tree is a finite set of one or more nodes, such that

- There is a specially designated node called *root*.
- Remaining are partitioned into n>=0 disjoint sets $T_1,\ldots,T_n$ where each T_i is a tree. $T_1,\ldots,T_n$ are called the sub trees of the root.

A tree with 9 nodes is given below.

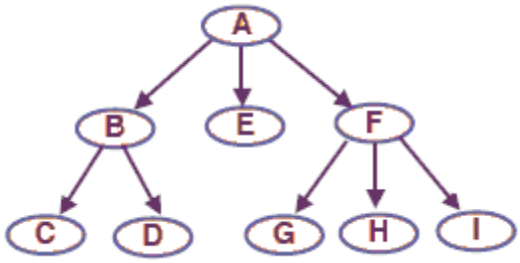


Fig 4.1

$T_1,\ldots,T_n$ are called the **sub tree(**any connected structure below the root) of the root.

e.g., BCD, E, FGHI

4.1 Basic Terminologies

A tree consists of a finite set of elements, called nodes, and a finite set of directed links, that connect the nodes.

1) A **node** may contain a value or a condition or represent a separate data structure or a tree of its own. The topmost node in a tree is called the **root node**.
   e.g., A .B are nodes in the figure.
2) A **link** is the pointer to anode in the tree.
3) **Parent** of a node is the immediate predecessor of a node. Root is a special node has no parent.
   e.g., A, B, F in fig 3.1
4) **Children** of a node are all immediate successors of a node.
   e.g., B, E, F, C, D, G, H, I in fig 3.1
5) Children of the same parent are said to be **siblings**.
   e.g., {B, E, F}, {C, D}, {G, H, I}
6) **Degree** of a node is the number of sub trees of a node.
   e.g., degree of node A is 3, node B is 2 etc.
7) **Degree of the tree** is the maximum degree of any node in the tree
   The degree of the above tree is 3.
8) Node have degree zero is called **leaf or terminal node.**
   e.g., C, D, E, G, H, I

9) Each node has to be reachable from the root through a unique sequence of links called **Path.** The number of links in the path called **length of the path.**
e.g., Path from the root to the leaf I is AFI.
10) The **ancestors** of a node are all the nodes along the path from root to that node.
e.g., Ancestors of node G are F and A.
11) **Level** of a node is the rank of hierarchy. Root is at level 1.
e.g., Level of node 2 is 1.
12) **Height / depth** of a tree is the maximum level of any node in the tree.
e.g., Height of the above tree is 3.
13) Set of n>=0 disjoint tree is called **forest**.

4.2 Binary Tree

A binary tree is a tree whose nodes have at most two children (no node with degree greater than two).

Properties of a binary tree

- Maximum number of nodes on level of a binary tree is 2^{i-1}, i>=1.
- Maximum number of nodes in a binary tree of depth k is 2^k-1, k>=1.

Special kinds of binary tree

1) Skewed tree

This is a binary tree with each node having exactly either left child or right child.

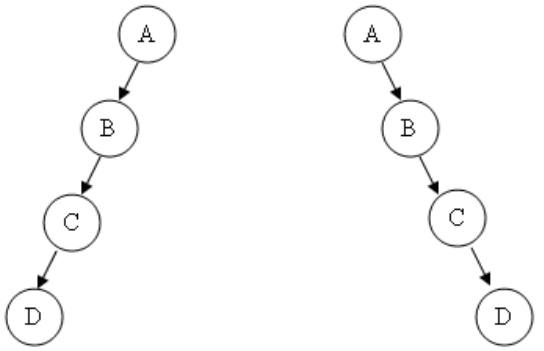


Fig 4.2

Left Skewed Right skewed

2) Full binary tree

Full binary tree of depth k is a binary tree having 2^k-1 nodes k>=0.

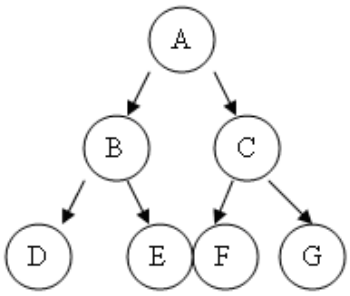


Fig 4.3 A Full binary Tree

3) Complete binary tree

A binary tree with n nodes and depth k is complete if and only if its nodes correspond to the nodes which are numbered 1 to n in the full binary tree of depth k.

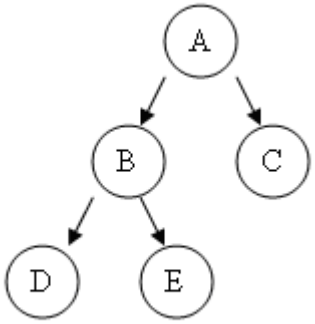


Fig 4.4 A Complete Binary Tree

Properties of a complete binary tree

- For any node i, Parent (i) is in location i/2 if i≠1. If i=1, then it is the root node and it has no parent.
- For any node i, LeftChild(i) is in location 2i if 2i<n.. If not, it don't have a left child
- For any node i, RightChild(i) is in location 2i+1 if 2i+1<n.. If not, it don't have a right child.

A binary tree can be represented as

```
struct binary
    {
       int data;
       struct binary *lchild, *rchild;
    }
```

Main operation on binary tree is traversal. When a node is visited, the required operations performed on it. Full traversal produces a linear order for the nodes in the tree.

There are three commonly used tree traversals. They are Inoder Traversal, Preorder Traversal and Postorder Traversal

4.2.1 Inorder Traverasal

Take any node

- traverse the left sub tree; and then
- visit the root; and then
- traverse the right sub tree.

Consider the tree given below.

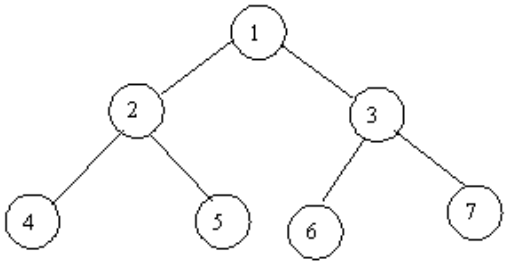


Fig 3.5

Inorder result of the above tree is 4,2,5,1,6,3,7.

Program 4.1

```
struct binary
{
        int data;
        struct binary *lchild, *rchild;
}

Void Inorder ( struct binary * tree)
{
    if (tree != NULL)
    {
            Inorder(tree->lchild);
            printf("%d",tree->data);
            Inorder(tree->rchild);
    }
}
```

4.2.2 **Preorder Traversal**

Take any node

- visit the root; and then
- traverse the left sub tree; and then
- traverse the right sub tree.

Preorder result of the **fig 3.5** is 1,2,4,5,3,6,7.

Program 4.2

```
Void Preorder ( struct binary * tree)
{
    if (tree != NULL)
    {
            printf("%d",tree->data);
            Preorder(tree->lchild);
            Preorder(tree->rchild);
```

```
    }
}
```

4.2.3 **Postorder Traversal**

Take any node

- traverse the left sub tree; and then
- traverse the right sub tree; and then
- visit the root

Postorder result of the above tree is 4,5,2,6,7,3,1.

Program 4.3

```
Void Postorder ( struct binary * tree)
{
    if (tree != NULL)
    {
            Postnorder(tree->lchild);
            Postorder(tree->rchild);
            printf("%d",tree->data);
    }
}
```

There are several types of binary tree possible each with its own properties.

- Expression Tree
- Binary Search Tree
- Heap Tree
- Decision Tree
- Height balanced Tree

4.3 Binary Search Tree

Binary search tree is a binary tree with all left descendants of a node *k* is less than *k* and all right descendants have keys greater than or equal to *k*. Hence Inorder traversal yields elements sorted in ascending order.

Searching, insertion and deletion can be performed effectively, with complexity between O ($\log_2 n$) and O (n). If the elements are entered in sorted order, then searching degenerates to sequential search.
Searching is very fast. In the application like where frequent searching operations are to be performed this data structure can be used.

4.3.1 Search an element

If the search item is less than the root node then proceeding to its left child else proceed to its right child. This process continued till item found or reach leaf node.

Program 4.4

```
void search ( struct binary *tree,int key)
{
        while (tree !=NULL &&key != tree->data)
           {
                if (key < tree->data)
                tree=tree->lchild;
                else
                tree=tree->rchild;

           }
        If(tree==NULL)
        printf("Element not found");
        else
        printf("Element found");
}
```

4.3.2 Insert an element into a Binary Search Tree

A binary tree is constructed by the repeated insertion of new nodes into a binary tree structure. Insertion must maintain the order of the tree. That is, values to the left of a given node must be less than that node, and values to the right must be greater.

Inserting a node into a tree is actually two separate operations. First, the tree must be searched to determine where the node is to be inserted. Second, on the completion of the search, the node is inserted into the tree.

Assuming that duplicate entries are not allowed in the tree, two cases must be considered when constructing a binary tree.

Case 1: *Insertion into an empty tree.*

In this case, the node inserted into the tree is considered the root node.

Example:

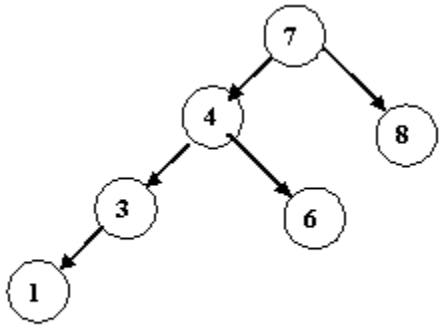
Before Insertion of 4

After Insertion of 4

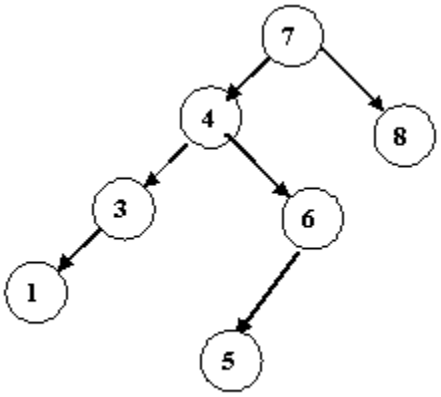
Case 2: *Inserting into a non-empty tree.*

Example:

Before insertion of 5



After insertion of 5



Algorithm

The tree must be searched, as was noted above, to determine where the node is to be inserted. The new node is compared first to the root node of the tree.

If the value of the new node is less than the value of the root node, the new node is:

- Appended as the left leaf of the root node if the left sub tree is empty
- Else, the search continues down the left sub tree.

If the value of the new node is greater than the value of the root node, the new node is:

- Appended as the right leaf of the root node if the right sub tree is empty
- Else, the search process continues down the right sub tree.

If this search procedure finds that the insertion node already exists in the tree, the procedure terminates as it can not insert it again.

Time Complexity

Average case is O ($\log_2 n$).

Worst case is O (n).

Program 4.5

```
struct binary insert (struct binary *tree, int key)
{
        struct binary *q, *temp,*head;
        head=tree;
        temp=(struct binary*)malloc(sizeof(struct binary));
        temp->data=key;
        temp->rchild=NULL;
        temp->lchild=NULL;
        q=NULL;
       while(tree!=NULL)
                {
                        q=tree;
                         if(key<tree->data)
                           tree=tree->lchid;
                        else
                           tree=tree->rchild;
                }

        if(q=NULL)
          head=temp;
        else
        {
                If(key<q->data)
                  q->lchid=temp;
                else
                  q->rchild=temp;
        }
        return(head);
}
```

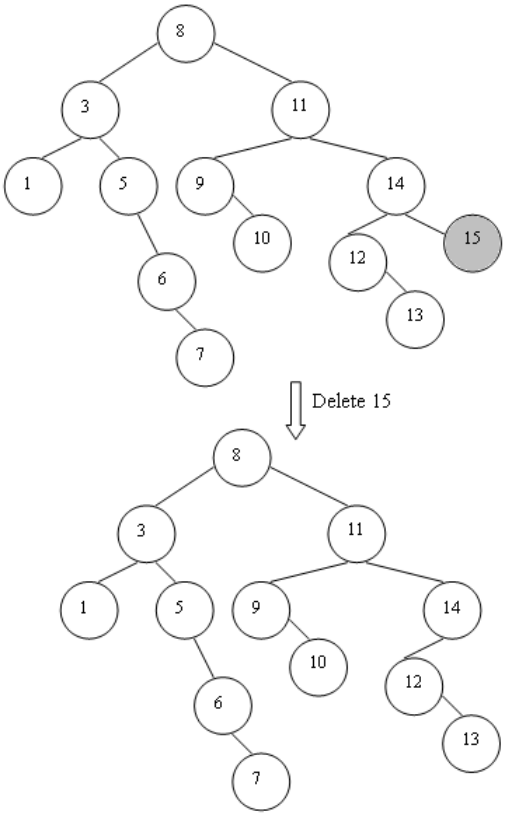
4.3.3 Deleting from a Binary Search Tree

The algorithm used for the delete function splits it into two separate operations, searching and deletion. Once the node which is to be deleted has been determined by the searching algorithm, it can be deleted from the tree. The algorithm must ensure that when the node is deleted from the tree, the ordering of the binary tree is kept intact.

Case 1: The node to be deleted has no children.

In this case the node may simply be deleted from the tree.

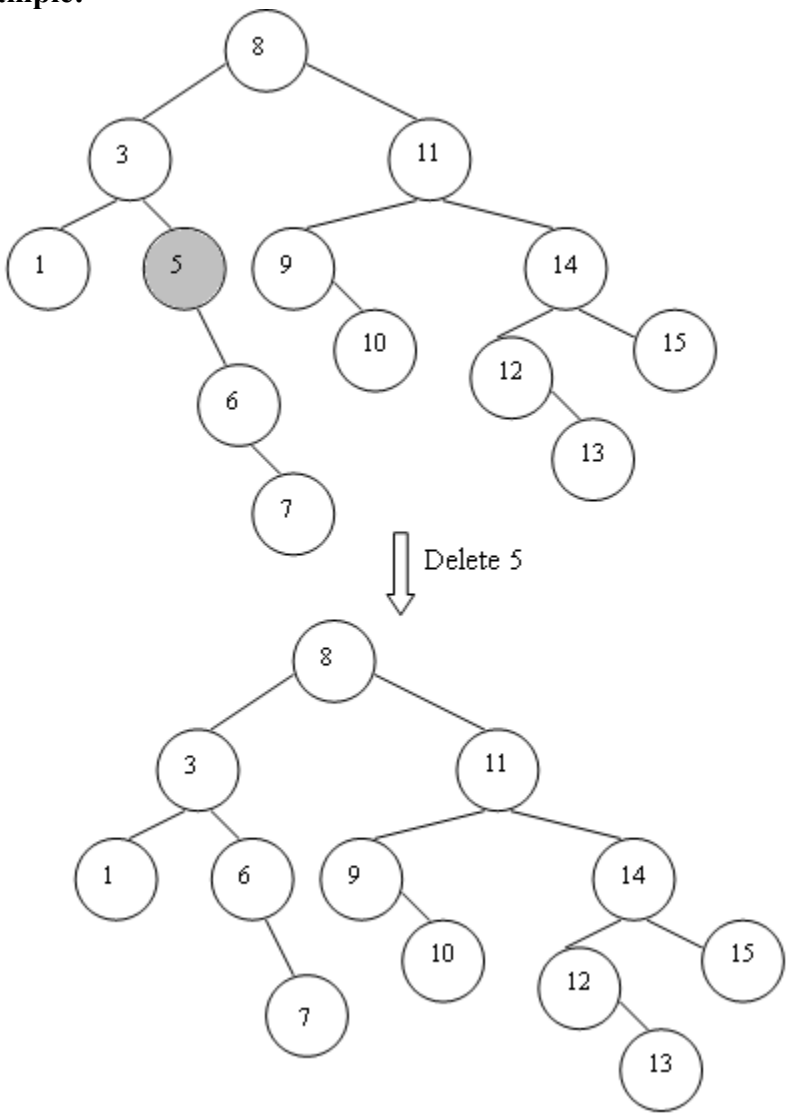
Example:



Case 2: *The node has one child.*

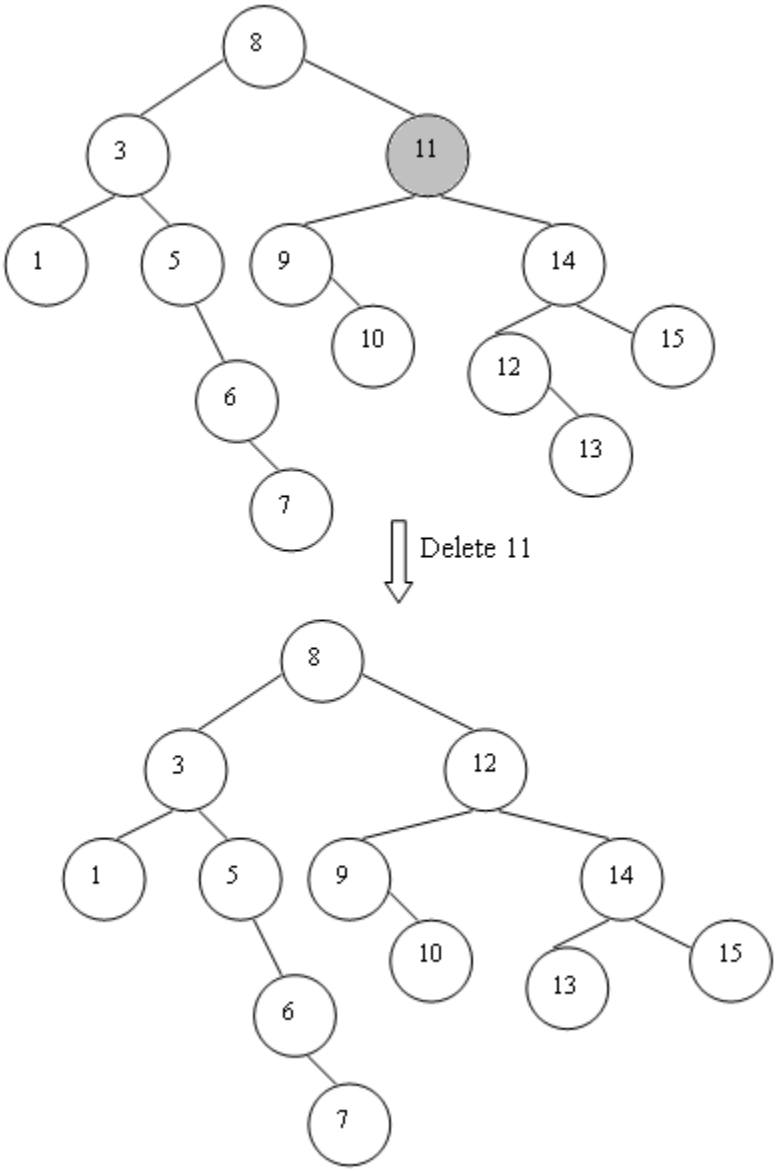
The child node is appended to its grandparent. (The parent of the node to be deleted.). If the node has one sub tree, move the only child to take the place of the current node.

Example:



Case 3: If node has both sub trees, its inorder successor takes its place.

Example:



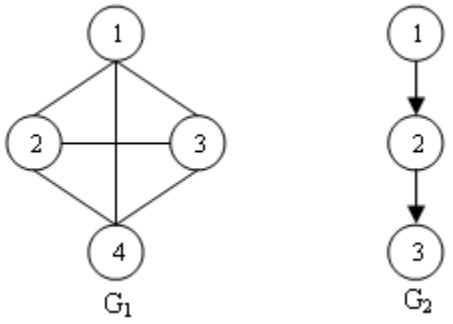
Time complexity
Average case is O ($\log_2 n$).
Worst case is O (n).

GRAPHS

A graph G is a data structure consisting of two sets V and E. V is a finite non-empty set of *vertices*. E is a set of pair of vertices; these pairs are called *edges* represented by V (G) and E(G) respectively. A graph can also be represented as G= (V,E).

There are two types of graphs, ***directed graph*** **and** ***undirected graph***. In a undirected graph, and edge is represented by an unordered pair (u,v). In a directed graph, each edge is represented by a directed pair <u,v> where, u is the tail and v is the head.

Examples:



G_1 is an undirected graph and G_2 is a directed graph.

$V(G_1) = \{1,2,3,4\}$

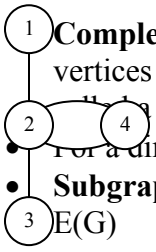
$E(G_1) = \{(1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,3)\}$

$V(G_2) = \{1,2,3\}$

$E(G_2) = \{<1,2>, <2,3>\}$

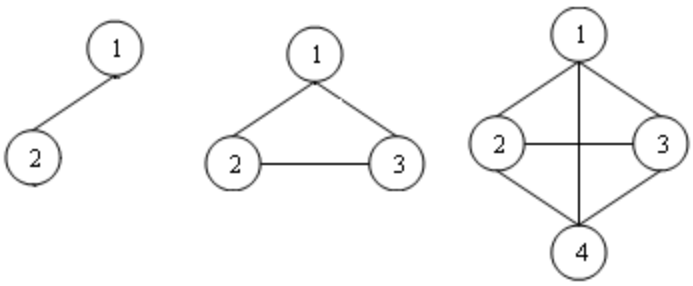
4.4 Basic Definitions

- **Multigraph**: An object with multiple occurrence of the same edge (Normally this is not considered as a graph)



- **Complete graph:** Maximum number of distinct pair (u,v) in any graph with n vertices is n(n-1)/2. An n vertex undirected graph with exactly **n(n-1)/2** edges is called a complete graph.
- For a directed graph with n vertices, the maximum number of edge is **n(n-1).**
- **Subgraph** : Subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G)

e.g, some of the sub graphs of graph G_1 are given below.



- **Path :** Path from vertex V_i to vertex V_j is a sequence of vertices $V_i, V_1, V_2, \ldots, V_n, V_j$ such that (V_i, V_i), (V_1, V_2),…..(V_n, V_j) are edges in E(G). If G is a directed graph, then $<V_i, V_i>$, $<V_1, V_2>$,….. $<V_n, V_j>$ should be edges in E(G).
  e.g.; Path from 1 to 3 in Graph G_1 is 1 2 4 3 and (1,2)(2,4)(4,3) in $E(G_1)$.
- **Length of a path:** It is the number of edges in the path.
  e.g.; Length of the above path is 3
- **Simple path:** A path with all vertices except possibly first and last are distinct.
  e.g.;1 2 4 3 is a simple path
- **Cycle:** A simple path with first and last vertices are same.
  e.g.; 1,2,3,1 is a cycle
- **Connected graph:** A graph is connected if or every pair of distinct vertices V_i, V_j in V(G), there is a path from V_i to V_j in G.
- In a directed graph for every pair of vertex I and J there is an edge between I and J, then the graph is called **strongly connected**.
- **Degree** of a vertex in a directed is the number of edges incident to that vertex.
  e.g.; degree of 1 in G1 is 3
- **In degree** of a vertex in a directed graph is the number of edges in which the vertex is the head.
- **Out degree** of a vertex in a directed graph is the number of edges in which the vertex is the tail.

4.5 Graph Representations

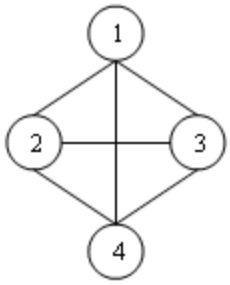
4.5.1 Adjacency matrix

The adjacency matrix of a graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (V_i, V_j) according to whether V_i and V_j are adjacent or not. Let G be a graph with "n" vertices that are assumed to be ordered from v_1 to v_n.
The n x n matrix A, in which

$a_{ij} = 1$ if there exists a path from v_i to v_j
$a_{ij} = 0$ otherwise

is called an adjacency matrix.

Example1

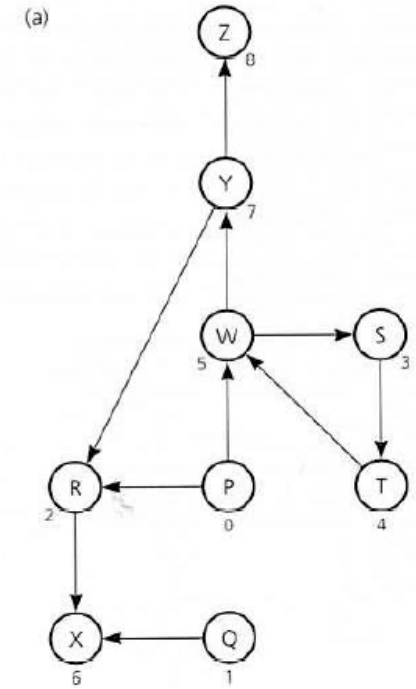


Adjacency matrix of the above graph is

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Example 2:

(a)



(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Space needed to represent a graph using adjacency matrix is n^2 bits.

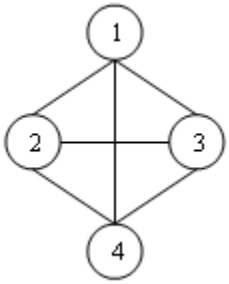
For an undirected graph, the adjacency matrix is symmetric

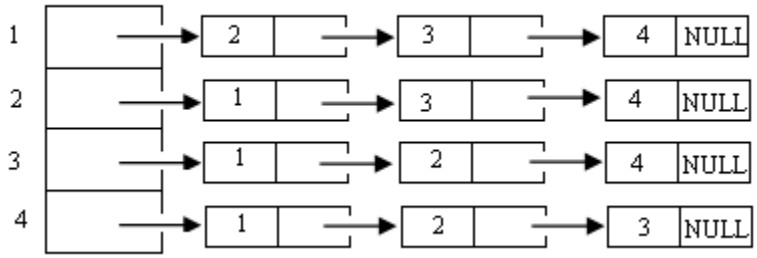
4.5.2 Adjacency List

An *adjacency list* consists of an *n*-element array of pointers, where the *i*th element points to a linked list of the edges incident on vertex *i*. To test whether edge (*i*,*j*) is in the graph, we search the *i*th list for *j*. This takes O(d) where d is the degree of the *i*th vertex.

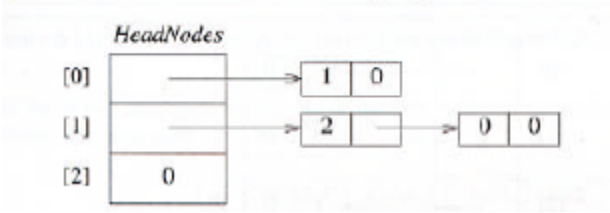
The adjacency list of a graph is a linked list with one list per vertex. Adjacency list of the following graph is given below.

Example1:





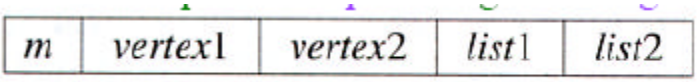
Example2:



Each list has a head node. Head nodes are sequential. Easy random access to adjacency list for a particular vertex is possible.

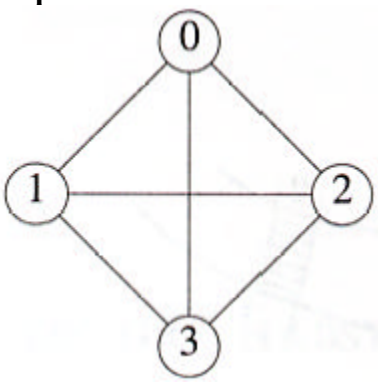
4.5.3 Adjacency Multi List

In adjacency list representation for an undirected graph, each edge (u,v) is represented by two entries, one on the list for u and the other on the list for v. In adjacency multilists (lists in which nodes may be shared among several lists.) for each edge there will be exactly one node, but this node will be in two lists. The node structure is given below.

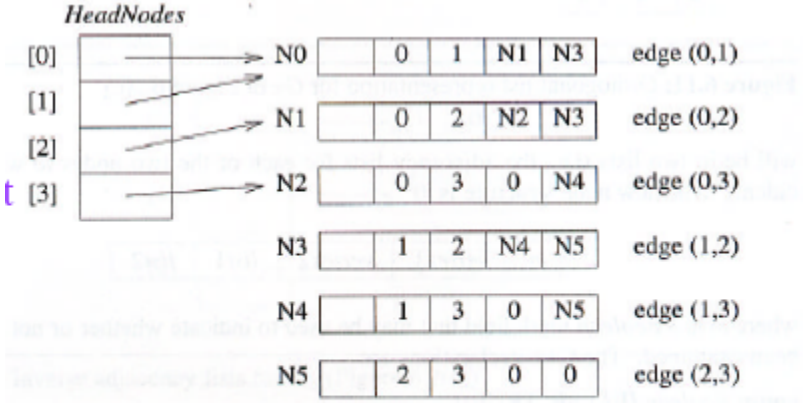


M is the Boolean mark field that may be used to indicate whether or not the edge has been examined.

Example:



The adjacency multilists for the above graph is



The lists are
vertex 0: N0 → N1 → N2
vertex 1: N0 → N3 → N4
vertex 2: N1 → N3 → N5
vertex 3: N2 → N4 → N5

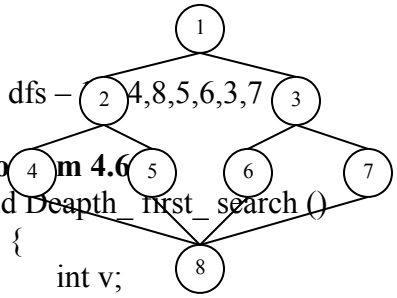
4.6 Graph Traversals

4.6.1 Depth First search

Algorithm

1. Initialize the adjacency matrix p to all zeroes.
2. Accept the number of vertices from the user, say n.
3. Initialize the visited array v to all zeroes.
4. Accept the graph.
    a. Initialize i to 0.
    b. Accept the vertex adjacent to ith vertex, say j.
    c. Make p[i][j] = p[j][i] = 1, as they are adjacent to each other.
    d. If more vertices are adjacent to ith vertex then goto step (a).
    e. Now consider the next vertex i.e. increment i and repeat from step(a).
5. Accept the starting vertex say V. Make it as visited.
6. Select unvisited vertex adjacent to V and perform DFS on that
7. When vertex u is reached such that all its adjacent vertices are visited, go back to the last vertex visited which has an unvisited vertex w adjacent to it and perform DFS with w.
8. Search terminated when no unvisited vertex can be reached from any of the visited vertex.

Example:



dfs – 1,2,4,8,5,6,3,7

Program 4.6

```
void Deapth_ first_ search ()
    {
        int v;
        for (v=1;v<=n;v++)
        visited[v]false;
        for (v=1;v<=n;v++)
        if(!visited[v])
        DFS(v);
    }
void DFS (int v)
    {
        int u;
        visited[v]=true;
        printf("%d",v);
        for (u=1;u<=n;u++)
        {
                If(adjmatrix[v][u])
                {
                        if(!visited[u])
```

```
                    DFS(u);
                }
            }

        }
```

Time complexity (adjacency matrix): O (n^2)
Time complexity (adjacency list): O (e)

4.6.2 Breadth First search

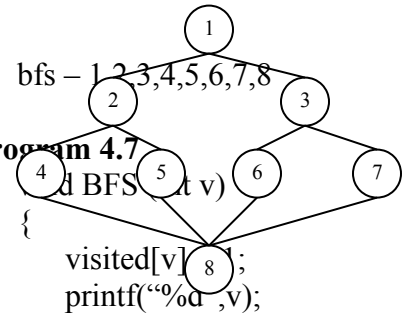
Suppose the graph is represented as an adjacency matrix, and we are required to have the breadth first search of the graph. Here we will require the starting vertex from which this search will begin. First that vertex will be printed, and then all the vertices, which are adjacent to it, are printed and so on.
If we have a matrix and there are n vertices. Let the starting vertex be j. Now the j^{th} vertex will be printed first, and to find all the vertices adjacent to this vertex, we must travel along j^{th} row of the matrix, and whenever we find 1 we will print the corresponding column number. Next time we will require each of the vertices printed recently so that we can travel level by level. For the same purpose we will push into a queue all the columns with the value one in the j^{th} row. Next time pop the vertex number from the queue and print it. Replace the value of j, which is currently printed. Again push all the vertices that are adjacent to this vertex into the queue and continue the above process until all the vertices are dealt with.

Remember there will be many vertices, which will be connected to more than a vertex, and therefore there are chances that we may repeat some of the vertices or there will be an infinite loop. To avoid this problem, we use what is known as visited array. It will be initially all zeroes. Whenever a vertex is pushed into the queue the corresponding position the visited array is changed to 1. Now we use another rule that we push only those vertices into the queue whose corresponding value in the visited array at that point is zero. When all the vertices are printed we will stop. Sometimes it happens that a particular vertex or a group of vertices is non reachable from the current vertex and in this case the graph is '*not connected'*.

Algorithm

1. Initialize the adjacency matrix p to all zeroes.
2. Accept the number of vertices from the user, say n.
3. Initialize the visited array v to all zeroes.
4. Accept the graph.
    a. Initialize i to 0.
    b. Accept the vertex adjacent to ith vertex, say j.
    c. Make p[i][j] = p[j][i] = 1, as they are adjacent to each other.
    d. If more vertices are adjacent to ith vertex then goto step (a).
    e. Now consider the next vertex i.e. increment i and repeat from step(a).
5. Start from any vertex say i. Make it as visited.
6. Go to all vertices adjacent to v and make it as visited
7. Then unvisited vertices adjacent to these newly visited vertices are then visited and so on.

Example:



bfs – 1,2,3,4,5,6,7,8

Program 4.7

```
void BFS (int v)
{
    visited[v] = 1;
    printf("%d ",v);
    AddQueue(v); // Add v to queue
    while(!IsEmpty()) // Check for empty queue
    {
        w = Delete(); // Delete from the queue
        for(u=1; u<=n; u++)
        {
            if (a[w][u] == 1 && !visited[u])
            {
                AddQueue(w);
                visited[w] = 1;
                printf("%d",w);
            }
        }
    }

}
```

Adjacency lists, time complexity: (n+e)
Adjacency matrix, time complexity: $O(n^2)$

Module-III

Topics **Page Number**

2.1 **Introduction to Data Structures**

A data object is a set of instances or values.

e.g., boolean = {true, false)
digit = {0,1,….,9}
letter = {a..z, A..Z}

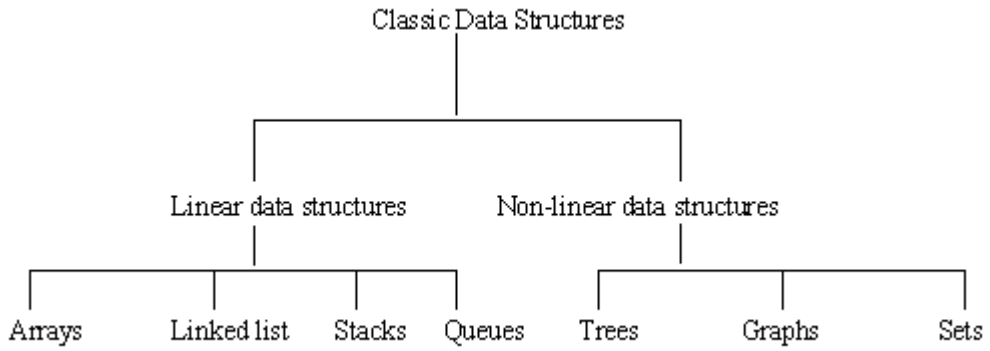
Here boolean, digit and letter are data objects. True and false are instances of data object, Boolean. Individual instance of a data object s are called atomic or primitive. If instance composed of instance of other objects, then it is called an element.

A ***data structure*** is a data object together with the relationship that exists among the individual elements that compose the instance. Study of the data structure involves 2 goals:

1. Identify and develop useful mathematical entity and operations and to determine what class of problems can be solved using these entities and operations.
2. Determine representation for these abstract entities and to implement the abstract operation on these concrete representations.

Data Structure D is a triplet, that is D= (d, f, a) where **d** is a set of data object, **f** is a set of functions and **a** is a set of rules to implement the functions.
Eg. Consider integer data type in C programming language the structure include the following.
d= (0,+/- 2……)
f= (+,-,*,/,5)
a=(set of binary arithmetic to perform addition, subtraction, division etc.)

In computer science, several data structures are known depending on area of applications. Of them, few data structures are there which are frequently used almost in all application areas. These data structures are called fundamental data structures or classic data structures. The following figure gives a classification of all classic data structures.



2.2 **Sparse Matrix**

A matrix is a mathematical object that arises in physical problems. As computer scientists, we are interested in studying ways to represent matrices so that the operations to be performed on them can be carried out efficiently. A general matrix consists of **m**

rows and **n** columns of numbers say A[m][n]. We can work with any element by writing A[i][j], and this element can be found very quickly.

$$A = \begin{pmatrix} 5 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

If we look at the above figure we see that it has many zero entries. Such matrix called sparse matrix. In the above matrix 4 out of 20 are non zeros. For effective memory utilization we store only the nonzero elements. Each element is uniquely identified by its row and column position.

(row, column, element)

This means that we can use an array of triples to represent a sparse matrix. It is stored in the row major form. i.e., in the ascending order of rows. Inside each row, store elements in the ascending order of columns.

Example18

$$A = \begin{pmatrix} 5 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \Rightarrow \quad S = \begin{pmatrix} 5 & 4 & 4 \\ 0 & 0 & 5 \\ 0 & 3 & 2 \\ 1 & 3 & 1 \\ 3 & 1 & 3 \end{pmatrix}$$

S[0][0] = number of rows in original matrix
S[0][1] = number of columns in original matrix
S[0][2] = number of non zero elements

Program 2.1

```
function sparse (int a[][], int m, int n)
{
    int i, j, k,S[][];
    S[0][0] = m;    S[0][1] = n;     k=1;
    for (i=0; i<m; i++)
        for (i=0; i<n; i++)
            if (a[i][j] != 0)
            {
                S[k][0] = i;
                S[k][1] = j;
                S[k][2] = a[i][j];
                k++;
            }
    S[0][2] = k-1;
```

2.2.1 Transpose of a Sparse Matrix

To transpose a matrix, we must interchange the rows and columns. This means that if an element is at position [i][j] in the original matrix ,then it is at position [j][i] in the transposed matrix. When i=j, the elements on the diagonal will remain unchanged.

Algorithm

1. Fix the first row of transpose matrix is
   T[0][0] = S[0][1]; T[0][1] = S[0][0]; T[0][2] = S[0][2];

2. Iterate through the 2nd column n times, where n is the number of columns in original matrix, ie, S[0][1].
   i. Iterate through the 2nd column t times, where t is the number of non zero elements
      - Select all elements in that column and store into the transpose matrix.

Program 2.2

```
function Transpose (int a[][])
{
    int m, n, p, q, t, r, b[][];
    m = a[0][0];    n = a[0][1];    t = a[0][2];
    b[0][0] = n;    b[0][1] = m;    b[0][2] = a[0][2];
    if ( t > 0 )
    {
            q = 1;
            for (r=0; r<n; r++)
                for (p=1; p<=t; p++)
                    if (a[p][1] == r)
                    {
                            b[q][0] = a[p][1];
                            b[q][1] = a[p][0];
                            b[q][2] = a[p][2];
                            q++;
                    }
    }
}
```

Time Complexity:

Line no. 10 : nt times
Line no. 12 – 15 : t times
Hence time complexity is O (nt).
Worse case (when t is of order mn) complexity is O (mn^2).

2.2.2 Fast Transpose

The time complexity of the transpose algorithm is **O (nt)** where n is the number of columns and t is the number of nonzero elements. We can do much better by using a little more storage. We can transpose a matrix represented as a sequence of triples in time O (terms+columns).

Algorithm

1. Fix the first row of transpose matrix is
   T[0][0] = S[0][1]; T[0][1] = S[0][0]; T[0][2] = S[0][2];
2. Determine number of nonzero elements in each column. From this find the starting position of elements corresponding to each column as
   start [i] = start[i-1] + number[i-1]
3. Fill the elements corresponding to each column.

Program 2.3

```
function FastTranspose (int a[][])
{
    int i, pos, n, terms, s[], no[], b[][];
    m = a[0][0];    n = a[0][1];    terms = a[0][2];
    b[0][0] = n;    b[0][1] = m;    b[0][2] = a[0][2];
    if (terms > 0 )
    {
            for (i=0; i<n; i++)
                    no[i] = 0;
            for (i=1; i<=terms; i++)
                    no[a[i][1]] += 1;
            s[0] = 0;
            for (i=1; i<n; i++)
                    s[i] = s[i-1] + no[i-1];
            for (i=1; i<=terms; i++)
            {
                    pos = s[a[i][1]];
                    b[pos][0] = a[i][1];
                    b[pos][1] = a[i][0];
                    b[pos][2] = a[i][2];
                    s[a[i][1]]++;
            }
    }
}
```

There are four loops executing n, terms, n-1 and terms times respectively. Each iteration of the loops takes only constant amount of time, hence complexity is O(n+t) or O(n). It becomes O(mn) when t is of the order mn. Hence this algorithm gives better performance that previous one but space complexity increases.

2.2.3 Sparse Matrix Addition

Program 2.3

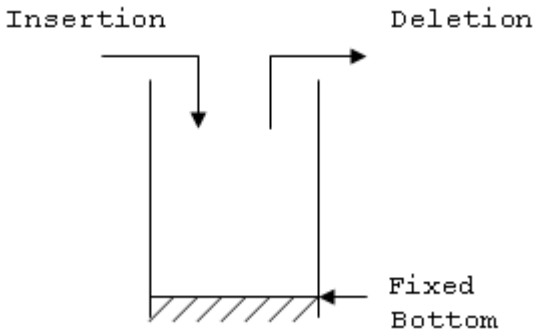
```
function SparseAdd (int a[][])
{
    int a_index, b_index, c_index, a_terms, b_terms, c_terms, c[][];

    if (a[0][0] != b[0][0] || a[0][1] != b[0][1])
            printf("Incompatible matrix");
```

```
    else
    {
            c[0][0] = a[0][0];  c[0][1] = a[0][1];
            a_terms = 1;   b_terms = 1;   c_terms = 1;
            while (a_terms <= a[0][2]  &&  b_terms <= b[0][2])
            {
                a_index = a[a_terms][0] * a[0][1] + a[a_terms][1];
                b_index = b[b_terms][0] * b[0][1] + b[b_terms][1];
                if (a_index < b_index)
                {
                     c[c_terms][0] = a[a_terms][0];
                     c[c_terms][1] = a[a_terms][1];
                     c[c_terms][2] = a[a_terms][2];
                     a_terms++;
                }
                else if (a_index > b_index)
                {
                     c[c_terms][0] = b[b_terms][0];
                     c[c_terms][1] = b[b_terms][1];
                     c[c_terms][2] = b[b_terms][2];
                     b_terms++;
                }
                else
                {
                     c[c_terms][0] = a[a_terms][0];
                     c[c_terms][1] = a[a_terms][1];
                     c[c_terms][2] = a[a_terms][2] + b[b_tems][2];
                     a_terms++;
                     b_terms++;
                }
                c_terms++;
            }
            for (; a_terms < a[0][2]; a_terms++, c_terms++)
            {
                     c[c_terms][0] = a[a_terms][0];
                     c[c_terms][1] = a[a_terms][1];
                     c[c_terms][2] = a[a_terms][2];
            }
            for (; b_terms < b[0][2]; b_terms++, c_terms++)
            {
                     c[c_terms][0] = b[b_terms][0];
                     c[c_terms][1] = b[b_terms][1];
                     c[c_terms][2] = b[b_terms][2];
            }
            c[0][2]=c_terms-1;
    }
}
```

2.3 The Stack Abstract Data Type

Stack is an ordered collection of items into which insertions and deletions are made at one end, called **top**. Last element inserted will be taken out first. Hence called **Last In First Out (LIFO).** A stack can pictorially represent as below.



Given a stack S= (a_0,.....a_{n-1}), we say that $a_0$is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1},0<i<n. If we add A, B, C, D, E to the stack then E is the first element delete from the stack.

Applications of Stack

- Recursion implementation
- Parenthesis matching
- Expression evaluation

2.3.1 Stack Operations

a) *Push*

An item is put on top of the stack, increasing the stack size by one. As stack size is usually limited, this may provoke a stack overflow if the maximum size is exceeded. The first element of stack is stored in stack S[o], the second is stored in S [1] and the ith in S[i-1]. A variable top points to the top element of the stack. Initially top is set to -1 to denote an empty stack.

Algorithm

I/P: The new element item.
O/P: Stack with newly pushed element item
Data structure: An array S with top as the pointer.

Steps:

1. If top>=size then
    a) print stack is full
2. Else
    a) top=top+1
    b) s[top]=item
3. EndIf
4. Stop

Program 2.4

```
void push(int s[], int stacksize, int item)
{
    int top= -1;
    if (top >== stacksize -1)
            printf("Stack full");
    else
    {
            top++;
            s[top] = item;
    }
}
```

b) *Pop or pull*

The top item is taken from the stack, decreasing stack size by one. In the case where there was no top item (i.e. the stack was empty), a *stack underflow* occurs

Algorithm

I/P: A Stack S with elements
O/P: Remove item from top of the stack
Data structure: An array S with top as the pointer.

Steps:

1. If top<1
    a) Print stack is empty
2. Else
    a) item=S[top]
    b) top=top-1
3. EndIf
4. Stop

Program 2.5

```
int pop(int s[])
{
    if (top == -1)
            printf("Stack empty");
    else
    {       top=top-1;
            return(s[top+1]);
    }
}
```

The combination of pushing and pulling the stack is sometimes referred to as "shaking" the stack (because of the pushing and pulling).

2.3.2 Application of Stack- Evaluation of Arithmetic Expressions

An arithmetic expression consists of operands and operators. Operands are variables or constants and operators are of various types like arithmetic unary and binary operators. In addition to this parenthesis are also used. The problem to evaluate this expression is the order of evaluation. Whatever the way we specify the order of evaluations, the problem is

that we must scan the expression from left to right repeatedly, this process is inefficient because of the repeated scanning. Another problem is that how compiler can generate correct code for a given expression. The last problem mainly occurs for a partially parenthesized expression. These problems can be solved with the following two steps:

1. Conversion of a given expression into a special notation
2. Evaluation/production of object code using stack.

Every expression can be denoted in three types and they are

1. Infix
2. Postfix
3. Prefix

Infix notation is the common arithmetic and logical formula notation, in which operators are written between the operands they act on. In infix notation, parentheses surrounding groups of operands and operators are necessary to indicate the intended order in which operations are to be performed. In the absence of parentheses, certain precedence rules determine the order of operations.

eg., 3 + 4

Postfix notation is one in which operands precede the operator, thus dispensing with the need for parentheses.

eg., 34+

Prefix notation is one in which operator precede the operands, thus dispensing with the need for parentheses.

eg., +34

Various examples are given below.

Sl. No.	Infix	Postfix	Prefix
1	A+B	AB+	+AB
2	A+B-C	AB+C-	-+ABC
3	(A+B)*(C-D)	AB+CD-*	*+AB-CD
4	A^B*C-D+E/F/(G+H)	AB^C*D-EF/GH+/+	+-*^ABCD//EF+GH
5	((A+B)*C-(D-E))^(F+G)	AB+C*DE--FG+^	^-*+ABC-DE+FG
6	A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Converting the Infix expressions to Postfix form.

Out of the 3 notations postfix notation has certain advantages over other notations from the computational point of view. The main advantage is its evaluation. During the evaluation it does not require to scan the expression several times, but exactly once. This is possible using stack and which will be discussed in the following topic.

The first problem with understanding the meaning of expression is to decide in that order the operations are carried out. This means every language must uniquely define such an order. The programmer could specify the order of evaluation by using parenthesis. To fix the order of evaluation, we assign to each operator a priority. The operator with highest priority will be evaluated first.

The postfix notations are mainly used to reduce the parenthesis from the infix notation. The postfix form of expression calls for each operator to appear after its operands.

Algorithm

Method 1:

1. Fully parenthesize the expression.
   A^B*C-D+E/F/(G+H) → ((((A^B)*C)-D)+((E/F)/(G+H)))
2. Move all operators so that they replace their corresponding right parenthesis.
   ((((A^B)*C)-D)+((E/F)/(G+H))) → ((((AB^C*D-((EF/(GH+/+
3. Delete all left parenthesis
   ((((AB^C*D-((EF/(GH+/+ → AB^C*D-EF/GH+/+

Method 2:

1. Append a special delimiter '#' at the end of the expression
2. Initialize stack with '#'
3. Read a character
   i. if an operand, print it.
   ii. else if an operator, check whether it is ')', pop and print till '('.
   iii. else compare incoming symbol priority and in stack priority, if incoming priority is high push element to stack. Else pop and print elements until incoming element's priority is high.
4. When expression is over, pop and print the remaining elements from stack.
5. Stop

Program 2.6

```
void postfix(char e[])
{
    stack[top] = '#';
    top = 1;
    x= nexttoken(e);
    while( x!= '#')
    {
        if ( x is an operand)
            prinft("%c",x);
        else if (x == ')')
        {
            while (stack[top] != '(')
            {
                    y = pop(s);
                    printf("%c", y);
            }
         }
        else
        {
            while (isp(stack[top]) >= icp(x))
            {
                    y = pop(s);
                    printf("%c", y);
            }
            push(s,size,x);
         }
```

```
            x = nexttoken(e);
        }
        while (top > 0 )
        {
            y = pop(s);
            printf("%c", y);
        }
    }
```

Postfix Evaluation

Consider the following expression
Infix: A/B-C+D*E-A*C
Postfix: AB/C-DE*+AC*-
Suppose that every time we compute a value, we store it in the temporary location T_i,i>=1. If we read a postfix expression left to right, the first operation is division. The two operands that precede this are A and B. So the result of A/B is stored in T_1 and the postfix expression is modified as in the following figure.

Operation	Postfix
T_1=A/B	T_1C-DE*+AC*-
T_2= T_1-C	T_2DE*+AC*-
T_3=D*E	T_2 T_3+AC*-
T_4=T_2+T_3	T_4AC*-
T_5=A*C	T_4 T_5-
T_6=T_4-T_5	T_6

Algorithm

1. Append special delimiter '#' at the end of the postfix expression.
2. Read a character.
   i. if an operand, push to stack.
   ii. else pop as many operands as we need to evaluate the expressions. Push the result to stack.
3. When expression is over, pop and print the result from stack.
4. Stop

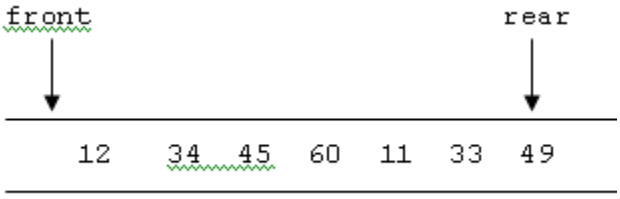
Program 2.7

```
void evaluate(char e[])
{
    top = 0;
    x= nexttoken(e);
    while( x!= '#')
    {
        if x is an operand
            push(s, size, x);
        else
        {
            z = pop(s);
```

```
                y = pop(s);
                y = eval (y, x, z);
                push(s, size,y);
            }
           x= nexttoken(e);
      }
   value =pop(s);
   printf("%d",value);
   }
```

2.4 The Queue Abstract Data Type

Queue is an ordered list in which insertions takes place at one end and deletions at the opposite end. It is called First In First Out (FIFO). Given a queue Q= (a_0, $a_{1,\ldots\ldots a_{n-1}}$), a_0 is the front element, a_{n-1} is the rear element, and a_i is behind a_{i-1}, 1<=i<n. If we insert A, B, C, D and E then A is the first element deleted from queue.



For insertion pointer rear will be used and for deletion pointer front will be used.

Three states of queue is given below.

Queue is empty

front= 0
rear= 0

Queue is full

front= 1
rear= n(queue size)

Queue contains elements>=1

Front <= rear
Number of elements= rear-front+1

Applications of Queue

- Most of the process scheduling or disk scheduling algorithms in operating systems use queues.
- Computer hardware like processor or a network card also maintain buffers in the form of queues for incoming resource requests. A stack like data structure causes starvation of the first requests, and is not applicable in such cases.
- A mailbox or port to save messages to communicate between two users or processes in a system is essentially a queue like structure

2.4.1 Queue Operations

Adding an element into the queue

The representation of a queue in sequential locations is more difficult than that of the stack. It is implemented using two pointers *front* and *rear*. *front* points to one position prior the starting element and *rear* points to the last element.

Algorithm

Input: An element item that has to be inserted
Output: The item is at the rear of the queue
Data Structure: Array representation of queue with two pointers rear and front.

Steps:

1. If(rear=n) then
    a) print queue full
    b) exit
2. Else
    a) if rear=0 &front=0
        i) front=1
    b) Endif
    c) Rear=rear+1
    d) Q[rear]=item
3. EndIf
4. Stop

Program 2.8

```
void add_queue(int item)
{
    if (rear == n)
        printf("Queue full");
    else
    {
        rear = rear + 1;
        q[rear] = item;
    }
}
```

Deleting an element from the queue

Algorithm

Input: A Queue with elements
Output: The deleted element is stored in item
Data Structure: Array representation of queue with two pointers rear and front.

Steps:

1. If(front=0) then
    a) print queue is empty
    b) exit

2. Else
    a. item=Q[front]
    b. If(front=rear)
        i) rear=0
        ii) front =0
    c. Else
        i) front =front+1
    d. Endif
3. EndIf
4. Stop

Program 2.9

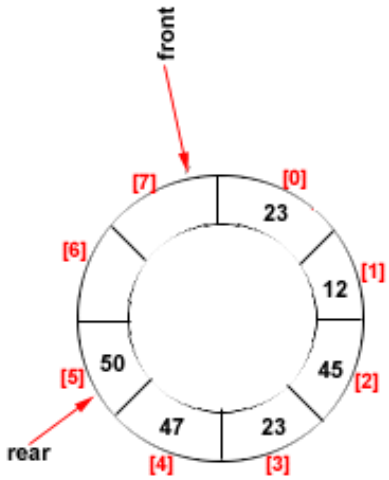
```
int delete_queue()
{
    if (front == rear)
    {
        printf("Queue empty");
        return(-1);
    }
    else
    {
        front = front + 1;
        return(q[front-1]);
    }
}
```

Various Queue Structures

2.4.2 Circular Queue

A very big drawback of linear queue is that, once the queue is FULL, even though we delete few elements from the "front" and relieve some occupied space, we are not able to add anymore elements, as the "rear" has already reached the Queue's rear most position. The solution lies in a queue in which the moment "rear" reaches the Queue's watermark; the "first" element will become the queue's new "rear".

Initially, when such a queue is empty, the "front" and the "rear" values are n-, where n is the size of the queue. Every time we add an element to the queue, the "rear" value increments by 1 till the time it reaches the upper limit of queue; after which it starts all over again from 0. Similarly, every time we delete an element from queue, the "front" value increments by 1 till the time it reaches the upper limit of queue; after which it starts all over again from 0. A circular queue can be diagrammatically represented as below.



The states of queue is given below

Circular queue is empty

front=0

rear=0

Circular queue is full

front= (rear mod length)+1

In this topic we are discussing basic set of operations, including insert an item, delete an item from the queue.

a) ***Adding an element into the queue***

Algorithm

Input: An element item to be inserted into the circular queue.
Output: Circular queue with the item at front, if the queue is not full.
Data Structure: Circular Queue with two pointers front and rear.

Steps

1. If (front=0) then
    a) front=1
    b) rear=1
    c) CQ[front]=item
2. Else
    a) new_rear=(rear mod length)+1
    b) if(new_rear!= front) then
        i) rear=new_rear
        ii) CQ[rear]=item
    c) Else
        i) print Q is full
    d) EndIf
3. End If

4. Stop

Program 2.10

```
void add_Cqueue(int item)
{
    int new_rear = (rear mod n) +1;
    if (front == new_rear)
        printf("Queue full");
    else
    {
        rear = new_rear;
        q[rear] = item;
    }
}
}
```

b) Deleting an element from the queue

Algorithm

Input: A Circular Queue with elements.
Output: The deleted element is item if the queue is not empty.
Data Structure: Circular Queue with two pointers front and rear.

Steps

1. If (front=0) then
    a) Print Queue is empty
    b) Exit
2. Else
    a) item=CQ[front]
    b) if(front=rear) then
        i) front=0
        ii) rear=0
    c) Else
        i) front=(front mod length)+1
    d) EndIf
3. EndIf
4. Stop

Program 2.11

```
int delete_Cqueue()
{
    if (front == rear)
    {
        printf("Queue empty");
        return(-1);
    }
    else
    {
            front = (front mod n) +1;
```

```
        return(q[front]);
    }
}
```

2.4.3 **Priority Queue**

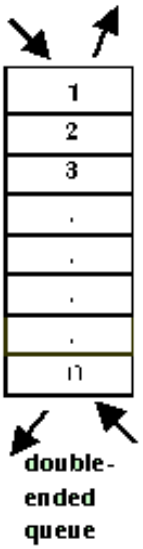
An abstract data type to efficiently support finding the item with the highest priority across a series of operations. There are two types of priority queues, ascending priority queues and descending priority queues.
Ascending priority queues: Elements can be inserted arbitrarily, but only the smallest element can be deleted.
Descending priority queues: Elements can be inserted arbitrarily, but only the biggest element can be deleted.

If we want to delete middle elements, there are different methods

1. Give special indicator if element is deleted. This can be a never existing value like -1 or separate filed for each element to indicate whether empty. Insertion can be performed in the normal way, but if *rear* reaches *max_size*, elements are to be compacted. Disadvantage is that, deletion involves searching deleted elements also. Once in a while we have to search every single position.
2. Deletion labels position as in the previous case, but insert elements into first possible empty position. This reduces efficiency of insertion.
3. After each deletion, shift element to one position and change *rear* accordingly. Another method is to shift either preceding or succeeding elements depending on which is economical. Disadvantage is that, we have to maintain *front* and *rear* pointers.
4. Instead of an unordered array, maintain ordered circular array. Here deletion involves increasing front by 1 (for ascending queues) or incrementing rear by 1 (for descending queues). Insertion will be complicated, because we have to insert in the exact position since queue is a sorted array.

2.4.4 **DeQueue**

A double-ended queue is an abstract data type similar to an ordinary queue, except that it allows you to insert and delete from both sides



In this topic we are discussing basic set of operations, including insert an item, delete an item from the queue.

a) Adding an element into the queue at the front

Algorithm
Input: New element item to be inserted at the front
Output: Deque with newly inserted element item
Data Structure: Dequeue with two pointers front and rear.

Steps
1. If (front=0&rear=0) then
    a) front=1
    b) rear=1
    c) DQ[front]=item
2. Else
    a) If (front=1)then
        i. F=length
        ii) if (F=rear)
            a) Print DQ full
        iii) Else
            a) front=F
            b) DQ[front]=item
        iv) EndIF
    b) Else
        i) F=front-1
        ii)) if(F=rear)
            a) Print DQ full
        iii) Else
            a) Front=F
            b) DQ[front]=item
        iv) EndIf
    c) EndIf

3. EndIf
4. Stop

b) Adding an element into the queue at the rear

Algorithm

Input: An element item that has to be inserted

Output: The item is at the rear of the queue

Data Structure: Array representation of queue with two pointers rear and front.

Steps:

5. If(rear=n) then
    a) print queue full
    b) exit
6. Else
    a) if rear=0 &front=0
        ii) front=1
    b) Endif
    c) Rear=rear+1
    d) Q[rear]=item
7. EndIf
8. Stop

Program 1.17

```
void add_queue(int item)
{
    if (rear == n)
        printf("Queue full");
    else
    {
        rear = rear + 1;
        q[rear] = item;
    }
}
```

c) ***Deleting an element from the rear end of the dequeue***

Algorithm

Input: A Dequeue with elements.
Output: The item is deleted from the rear if the queue is not empty.
Data Structure: Dequeue with two pointers front and rear

Steps

1. If(front=0) then
    a) print queue is empty
    b) exit
2. Else
    a) If (front=rear) then
        i) item=DQ[rear]

ii) front=rear=0
b) Else
i) If (rear=1) then
1. item=DQ[rear]
2. rear=length
ii) Else
1. item=DQ[rear]
2. rear=rear-1
iii) EndIf
c) EndIf
3. End If
4. Stop

d) Deleting an element from the front end of the dequeue
Algorithm

Input: A Queue with elements

Output: The deleted element is stored in item

Data Structure: Array representation of queue with two pointers rear and front.

Steps:
1. If(front=0) then
c) print queue is empty
d) exit
5. Else
a. item=Q[front]
b. If(front=rear)
i) rear=0
ii) front =0
c. Else
i) front =front+1
d. Endif
6. EndIf
7. Stop

Program 1.18

```
int delete_queue()
{
    if (front == rear)
    {
        printf("Queue empty");
        return(-1);
    }
    else
    {
```

```
        front = front + 1;
        return(q[front-1]);
    }
}
```

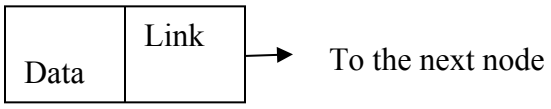
Topics **Page Number**

LINKED LISTS

LINKED LISTS

Array is a data structure where the elements are stored in consecutive memory locations. So block of memory should allocate before hand. Once the memory is allocated it can not extended. This is called **static data structure.** Linked list is a **dynamic data structure**, where amount of memory required can be varied during its use. In this adjacency between the elements are maintained by means of links. This link is a pointer, points to the address of the subsequent element.

In Linked list **data** (actual content) and **link** (point to next data) are maintained as a node.

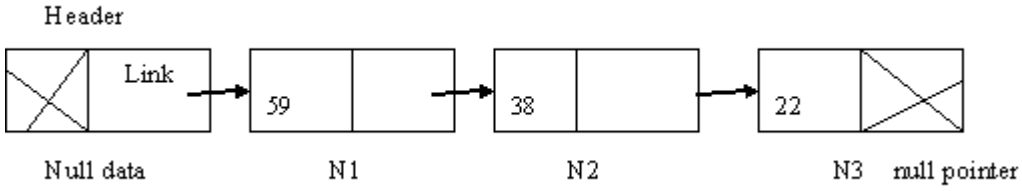


A linked list is an ordered collection of finite, homogenous data elements called nodes, where the linear order is maintained by means of links or pointers.
Depending on the requirements, the pointers maintained in the linked list can be classified into three groups.

Singly linked list, doubly linked list, Circular linked list

3.1 **Singly Linked List**

Each node contains only one link which points to the next node in the list.



Header stores a pointer to the first node N1. If we know the address of the header, next node can be traced and so on. In singly linked list one can move from left to right only so it is termed as **one way list**.

It can be represented as

```
struct name
{
    data_type item1;
    data_type item2;
    .
    .
    .
```

```
    data_type item n;
    struct name * next_pointer;
};
```

Possible operations on a singly linked list are listed below.

- Creating a list
- Display the contents of list
- Insertion of a node into a list
- Deletion of a node from the list
- Copy a linked list to make duplicate
- Merging two linked list into a larger list
- Searching for an element in a list

3.1.1 Creating a Singly linked list

Given below is a program to insert 10 elements to a singly linked list.

Program 3.1

```
struct node{
    int data;
    struct node * next;
};
void main()
{
    struct node *list, *temp, *head;
    int i;
    list = (struct node *) malloc (sizeof (struct node));
    list-> data = 1;
    list-> next = NULL;
    head = list;

    for(i=2; i<11; i++)
    {
            temp = (struct node*) malloc (sizeof(struct node));
            temp-> data = i;
            temp-> next = NULL;
            list-> next = temp;
            list = list-> next;
    }
}
```

3.1.2 Display a Singly linked list

Program 3.2

```
void display(struct node * list)
{
    while (list  != NULL)
    {
            printf("%d", list->data);
```

```
            list = list->next;
        }
    }
```

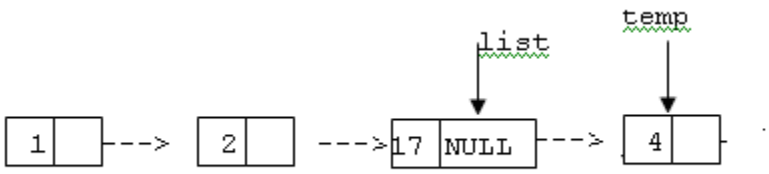
3.1.3 Insertion of a node in to a singly linked list

There are various positions where a node can be inserted:

i) Insert at end (as last element)
ii) Insert at front (as first element)
iii) Insert after a given element
iv) Insert before a given element

3.1.3.1 Insert at the end

Insert an element at the end of the existing list



Program 3.3

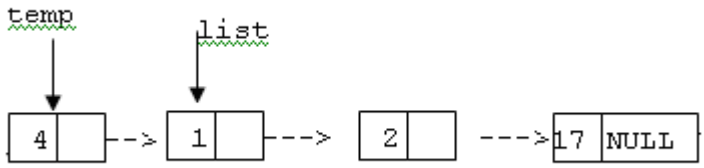
```
struct node* insert_end (struct node * list, int item)
{
    struct node *temp,*head;
    temp = (struct node*) malloc (sizeof(struct node));
    temp-> data = item;
    temp-> next = NULL;
    head=list;
    if (list == NULL)
        {
          list = temp;
          head=list;
         }
    else
    {
          while (list-> next != NULL)
                list = list->next;
          list->next = temp;
    }
    return(head);
}
```

Call the function “insert_end” from the main program as given below.

```
    head = insert_end (head, new_item, search_item);
```

3.1.3.2 Insert at front

The following program describes how to insert a node into a singly linked list at the front of the list.



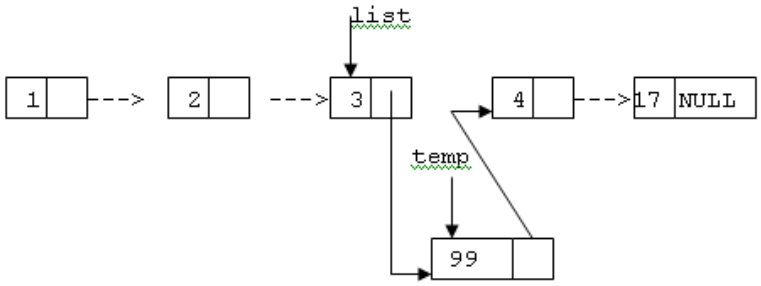
Program 3.4

```
struct node* insert_front (struct node * list, int item)
{
    struct node *temp,*head;
    temp = (struct node*) malloc (sizeof(struct node));
    temp-> data = item;
    temp-> next = NULL;
    temp->next = list;
    head=temp;
    return (head);
}
```

Call the function "insert_front" from the main program as given below.

```
    head = insert_front (head, new_item, search_item);
```

3.1.3.2 Insert an element after a given element



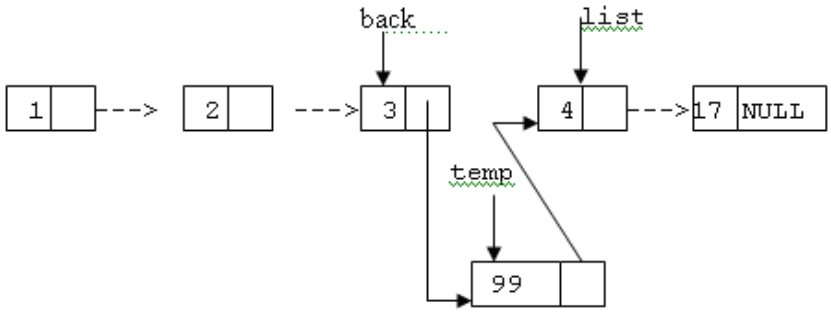
Program 3.5

```
void insert_after (struct node * list, int new_item, int search_item)
{
    struct node *temp;
    temp = (struct node*) malloc (sizeof(struct node));
    temp->data = item;
    temp->next = NULL;
    if ( list != NULL)
    {
       while(list != NULL)
       {
            if (list->data == search_item)
```

```
                {
                        temp->next = list->next;
                        list->next = temp;
                        break;
                }
            else
              list = list->next;
        }

    }
}
```

3.1.3.3 Insert an element before a given element



Program 3.6

```
void  insert_before (struct node * list, int new_item, int search_item)
{
    node *temp, *back,*head;
    temp = (struct node*) malloc (sizeof(struct node));
    temp->data = item;
    temp->next = NULL;
    if ( list != NULL)
        {
            while(list != NULL)
            {
                    if (list->data == search_item)
                            {
                                    temp->next = list;
                                    back->next = temp;
                                    break;
                            }
                    else
                    {
                            back = list;
                            list = list->next;
                    }
            } // while
```

```
        }
}
```

3.1.4 Deletion of a node form the list

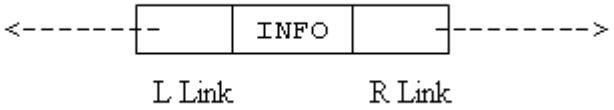
The following algorithm is to delete a node from any position in the singly linked list

Program 3.7

```
struct node*  delete (struct node * list, int search_item)
{
    node *temp, *back,*head;
    head = list;
    if ( list != NULL)
    {
        if (list->data == search_item)
        {
            temp  = list;
            list = list->next;
            free(temp);
            head = list;
        }
        else
        {
            while(list != NULL)
            {
                if (list->data == search_item)
                    {
                        temp = list;
                        back->next = list->next;
                        free(temp);
                    }       break;
                else
                {
                    back = list;
                    list = list->next;
                }
            } // while
        } //else
    } //if
    return(head);
}
```

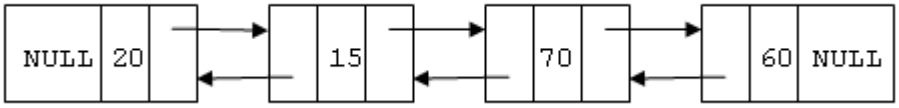
3.2 **Doubly Linked List**

In a singly linked list we can traverse in only one direction (forward), i.e. each node stores the address of the next node in the linked list. It has no knowledge about where the previous node lies in the memory. If we are at the 12th node (say) and if we want to reach 11th node in the linked list, we have to traverse right from the first node. This is a cumbersome process. Some applications require us to traverse in both forward and backward directions. Here we can store in each node not only the address of the next node but also the address of the previous node in the linked list. This arrangement is often known as a **doubly linked list**. The structure of a node is represented below.



From the figure, it can noticed that, two links L Link (left pointer) and R Link(right pointer) point to the nodes at the right side and left side of the node, respectively. Thus every node except the header node and the last node points to its immediate predecessor and immediate successor.

A doubly linked list with four elements is given below:



The doubly linked list is a **two-way** because one can move in either direction, either from left to right or from right to left.
The left pointer of the leftmost node and the right pointer of the rightmost node are NULL indicating the end in each direction. It can be represented as

```
struct name
{
    data_type item1;
    data_type item2;
    .
    .
    .
    data_type item n;
    struct name * llink , * rlink;
};
```

All the operations as mentioned for a singly linked list can be implemented more efficiently fro doubly linked list.

3.2.1 Creating a Doubly linked list

Given below is a program to insert 10 elements to a singly linked list.

Program 3.8

```
struct node {
    int data;
    struct node * llink, * rlink;
} ;

void main()
{
    struct node *list, *temp, *head;
    int i;
    list = (struct node *) maloc (sizeof (struct node));
    list->data = 1;
    list->llink = NULL;
    list->rlink = NULL;
    head = list;

    for(i=2; i<11; i++)
    {
            temp = (struct node*) malloc (sizeof(struct node));
            temp->data = i;
            temp-> rlink = NULL;
            list-> rlink = temp;
            temp->llink = list
            list = list-> rlink;
    }
}
```

3.2.2 Display a Doubly linked list

Program 3.9

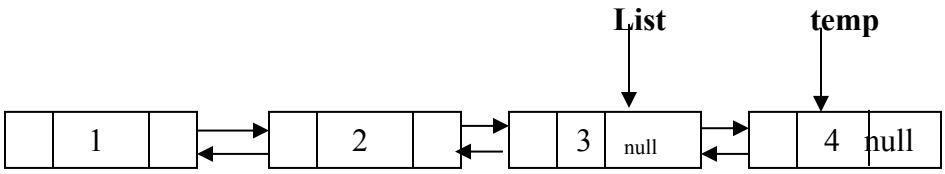
```
void display(struct node* list)
{
    while (list  != NULL)
    {
            printf("%d", list->data);
            list = list->rlink;
    }
}
```

3.2.3 Insertion of a node in to a doubly linked list

There are various positions where a node can be inserted:

i) Insert at end (as last element)
ii) Insert at front (as first element)
iii) Insert after a given element
iv) Insert before a given element

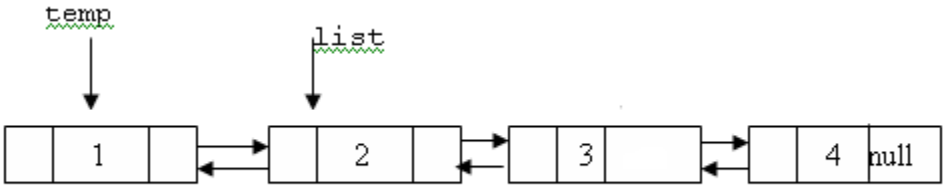
3.2.3.1 Insert at the end



Program 3.10

```
struct node* insert_end (struct node * list, int item)
{
    struct node *temp,*head;
    temp = (struct node*) malloc (sizeof(struct node));
    temp->data = item;
    temp->llink = NULL;
    temp->rlink = NULL;
    head=list;
    if (list == NULL)
    {       list = temp;
            head=list
    }
    else
    {
            while (list->rlink != NULL)
                    list = list->rlink;
            temp->llink = list;
            list->rlink = temp;
    }
return(head);
}
```

3.2.3.2 Insert at front

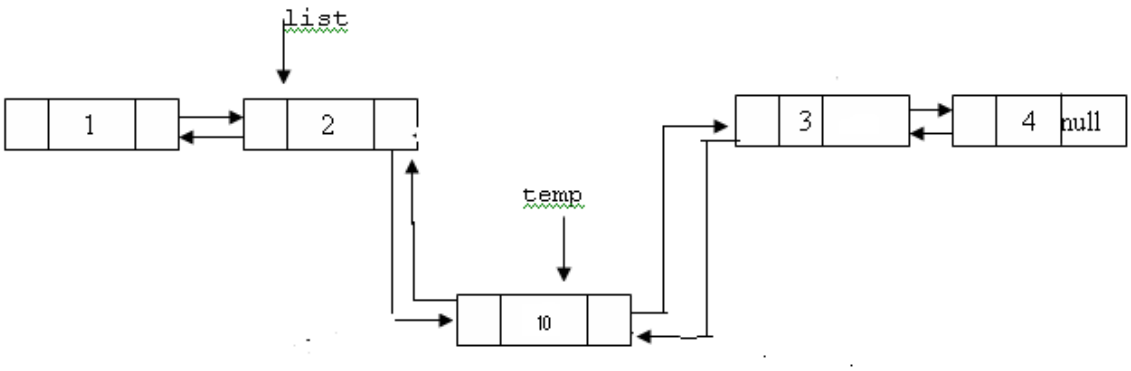


Program 3.11

```
struct node* insert_front (struct node * list, int item)
{
    struct node *temp,*head;
    temp = (struct node*) malloc (sizeof(struct node));
    temp->data = item;
    temp->llink = NULL;
    temp->rlink = NULL;
    head=temp;
```

```
    if (list == NULL)
            list = temp;
    else
    {
            temp->rlink = list;
            list->llink = temp;
    }
    return(head);
}
```

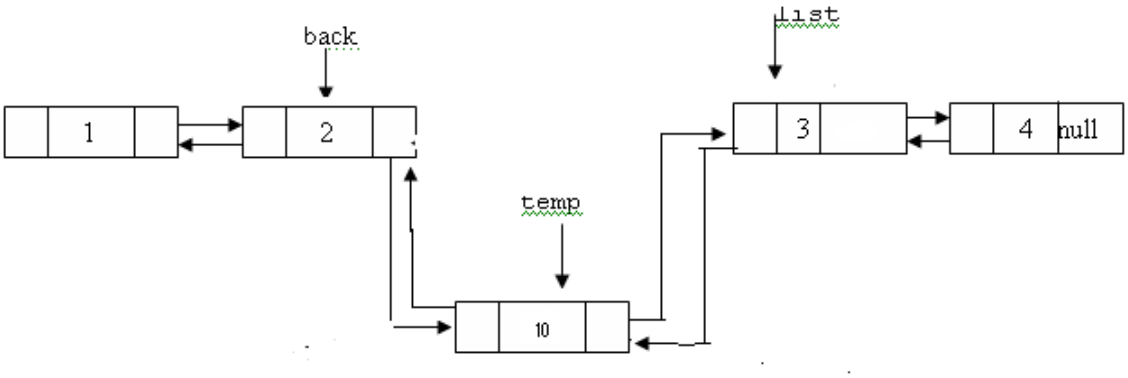
3.2.3.2 Insert an element after a given element



Program 3.12

```
void insert_after (struct node * list, int new_item, int search_item)
{
    struct node *temp;
    temp = (struct node*) malloc (sizeof(struct node));
    temp->data = item;
    temp->llink = NULL;
    temp->rlink = NULL;
    if ( list != NULL)
    {
        while(list != NULL)
        {
            if (list->data == search_item)
            {
                temp->llink = list;
                temp->rlink = list->rlink;
                list->rlink->llink = temp;
                list->rlink = temp;
                break;
            }
            else
              list = list->rlink;
        }
    }
}
```

3.2.3.3 Insert an element before a given element



Program 3.13

```
void  insert_before (struct node * list, int new_item, int search_item)
{
    struct node *temp;
    temp = (struct node*) malloc (sizeof(struct node));
    temp->data = item;
    temp->llink = NULL;
    temp->rlink = NULL;
    if ( list != NULL)
    {
        while(list != NULL)
            {
                if (list->data == search_item)
                {
                    temp->rlink = list;
                    temp->llink = list->llink;
                    list->llink = temp;
                    list->llink->rlink = temp;
                    break;
                }
                else
                    list = list->rlink;

            } // while
        } //else
    }
}
```

3.1.4 Deletion of a node from the list

The following algorithm is to delete a node from any position in the doubly linked list

Program 3.14

```
 struct node{
     int data;
     struct node * llink, * rlink;
} ;

struct node*  delete (struct node * list, int search_item)
{
     struct node *temp, *head;
     head = list;
     if ( list != NULL)
     {
          if (list->data == search_item)
          {
               temp  = list;
               list = list ->rlink;
               head = list;
               free(temp);
          }
          else
          {
               while(list != NULL)
               {
                         if (list->data == search_item)
                         {
                                   temp = list;
                                   list->rlink->llink = list->llink;
                                   list->llink->rlink = list->rlink;
                                   free(temp);
                                   break;
                         }
                         else
                                   list = list->next;
               } // while
          } //else
     } //if
     return(head);
}
```

3.3 **Circular Linked List**

In the previous two linked lists the link field of the last node is NULL. A linked list where the last node points the header node is called a circular linked list. In a circular linked list, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end.

Circular linked list has the following advantages.

This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list.

Accessibility of a member node in the list: In an ordinary list, a member node is accessible from a particular node, that is, from the header node only. But in the circular linked list, every member node is accessible from any node.

Null link problem: Null value in the link field may create some problem during the execution of programs if a proper care is not taken.

Circular linked list can be done for both singly and doubly linked lists.

a) Circular singly linked list

In a circular singly linked list, each node has one link, similar to an ordinary singly linked list, except that the link of the last node points back to the first node. As in a singly-linked list, new nodes can only be efficiently inserted after a node we already have a reference to. For this reason, it's usual to retain a reference to only the last element in a singly linked list, as this allows quick insertion at the beginning, and also allows access to the first node through the last node's next pointer.

b) Circular doubly linked list

In a circular doubly linked list, each node has two links, similar to doubly linked list, except that previous link of the first node points to the last node and the next link of the last node points to the first node. As in doubly linked lists, insertions and removals can be done at any point with access to any nearby node.

3.4 **Advantages and Disadvantages of linked list**

Advantages

1) In case of an array, data are stored in contiguous memory location, so insertion and deletion operation are quite time consuming, as before in insertion we have to make room for the new element at a desired position by shifting down the trailing elements, similarly in case of deletion, in order to fill the location of deleted element, all the trailing elements are required to shift upwards. But in the linked list, it is a matter of changing pointers.
2) Array is based on the static allocation of memory (amount of memory required for an array must be known before hand). Once it is allocated we can not expand its size. This is why for an array, general practice is to allocate memory space, which

is much more than the memory actually will be used. But this is simply wastage of memory. This problem is not there in the linked lists. Linked list uses the dynamic memory management scheme that is memory allocation is decided during the run time as and when require. Also if a memory is no more needed, it can be returned to free storage space, so that other program can utilize this.

3) A program using an array may not execute although the memory required for the data are available but not in contiguous locations rather dispersed. As link structure do not necessarily require to store data in adjacency memory locations. So the program of that kind, using linked list can then executed.

Disadvantages

1) Pointers if not managed carefully, may lead to serious errors in execution.
2) Linked list consumes extra space than the space for actual data as we are maintaining the links among the nodes.

3.5 Applications of linked list

The disadvantages of linked list are nothing compared to its advantages. This is why, the linked list is applicable in many situations, wherever it is required to manipulate data. Some of the applications are given below.

3.5.1 Implementation of Stack using linked list

a) Push

Algorithm

I/P: The new element item.

O/P: A singly linked list with newly pushed element item

Data structure: A singly linked list whose pointer to the header is known from head and top is the pointer to the first node.

Steps:

1. Create a new node say temp
2. temp->data=item
3. temp->next=top
4. top=temp
5. Stop

Program 3.15

```
Void push ( int item)
    {
        Struct node *temp;
        temp = (struct node*) malloc (sizeof(struct node));
      temp->data = item;
      temp->next = NULL;
      temp->next=top;
      top=temp;
    }
```

b) Pop

Algorithm

I/P: A stack with elements

O/P: Remove item from the top of the stack

Data structure: A singly linked list whose pointer to the header is known from head and top is the pointer to the first node.

Steps:

1. If top=NULL
    a) Print stack empty
    b) Exit
2. Else
    a) Top is pointing to next data (top=top->next)
3. EndIf
4. Stop

Program 3.16

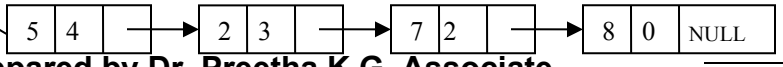
```
Void pop()
        {
                Struct node *temp;
                If (top==NULL)
                   Printf( "stack empty");
                else
                   {
                        Printf("%d"'top->data);
                        temp=top;
                        top=top->next;
                        free(temp);
                    }
         }
```

3.5.2 Polynomial representation using linked list

Main advantage of linked list for polynomial representation is that it can accommodate a number of polynomials of growing sizes so that their combined size does not exceed the total memory available.

Let us consider a polynomial $5x^4 + 2x^3 + 7x^2 + 8$. In this each node should consist of three elements, namely coefficient, exponent and a link to the next item. While maintaining the polynomial it is assumed that the exponent of each successive term is less than that of the previous term. Once we build a linked list to represent the polynomial we can perform operations like addition and multiplication. It can be represented as

```
struct poly
{
    int coeff;
    int exp;
    struct poly * next;
};
```



Polynomial addition

Two polynomials P and Q is added and to get the resultant polynomial R. We have to compare their terms starting at the first nodes and moving towards the end one-by one. Two pointers pptr and qptr are used to move along the terms of P and Q. There may be three cases during the comparison.

Case1:

Exponents of terms are equal. In this case coefficients in two nodes are to be added and a new term is created with the values

rptr->coef=pptr->coef + qptr->coef

and

rptr->exp=pptr->exp

Case2:

If pptr->exp>qptr->exp then a duplicate of current term in P is created and to be inserted in the polynomial R

Case3:

If pptr->exp<qptr->exp then duplicate of current term in Q is created and to be inserted in the polynomial R

Data Module-I

Topics **Page Number**

Introduction to Algorithms

1.1 Algorithm Specification

Algorithm, named for the 9th century Persian mathematician Al-Khowarizmi, is simply finite set of instructions that, if followed, accomplish a specified task. Algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison) until the task is completed. Different algorithms may complete the same task with a different set of instructions in more or less time, space, or effort than others. Correctly performing an algorithm will not solve a problem if the algorithm is flawed or not appropriate to the problem. Algorithms are essential to the way computers process information, because a computer program is essentially an algorithm that tells the computer what specific steps to perform (in what specific order) in order to carry out a specified task.

In order to carry out a task using a computer the following steps are followed:

1. The given task is analyzed.
2. Based on the analysis an algorithm to carry out the task is formulated. The algorithm has to be precise, concise and unambiguous.
3. The algorithm is expressed in a precise notation. This can be interpreted and executed by a computing machine and is called a computer program. The notation is called a computer programming language.
4. The computer program is fed to a computer.
5. The computers interprets the program, carries out the instructions given in the program and computer the results.

Example

This is an algorithm to find out the sum of two numbers.

Step 1: Start

Step 2: Declare variables and enter the values for two input numbers

Step 3: Add two numbers and store the result into another variable, sum

Step 4: Print the value of sum

Step 5: Stop

The algorithm should satisfy the following criteria.

- Input : Zero of more quantities that are externally applied
- Output : At least one quantity is produced
- Definiteness : Each instruction should be clear and unambiguous
- Finiteness : Algorithm terminates after finite number of steps for all test cases.
- Effectiveness : Each instruction is basic enough for a person to carried out using a pen and paper. This means that should ensure not only definite but also check whether feasible or not.

In computational theory, one distinguishes algorithm and program, the later of which does not have to satisfy for fourth condition (finiteness). For example, we can think of an operating system that continues in a 'wait' loop until more jobs are entered. Such a program does not terminate, unless the system crashes.

Algorithms come in different types

- **Recursive algorithm** is one that invokes (makes reference to) itself repeatedly until a certain condition matches
- **Greedy algorithm** works by making a series of simple decisions that are never reconsidered
- **Divide-and-conquer algorithm** recursively reduces an instance of a problem to one or more smaller instances of the same problem, until the instances are small enough to be directly expressible in the programming language employed
- **Dynamic programming algorithm** works bottom-up by building progressively larger solutions to sub problems arising from the original problem, and then uses those solutions to obtain the final result
- **Graph exploration algorithm** specifies rules for moving around a graph and is useful for developing chess
- **Probabilistic algorithms** are those that make some choices randomly
- **Heuristic algorithms**, whose general purpose is not to find a final solution, but an approximate solution where the time or resources to find a perfect solution are not practical

Once we have an algorithm, next is to check how effective the algorithm is. Performance evaluation can be divided into two major phases.

1. Priori estimate (Performance analysis)
2. Posteriori testing (Performance measurement)

1.2 Performance Analysis

There are many criteria upon which we can judge a program. Two criteria have more relationship to performance is computing time (Time complexity) and storage requirements (Space complexity).

1.2.1 Space Complexity

Space complexity of an algorithm is the amount to memory needed by the program for its completion. Space needed by a program has the following components,

1. **Instruction Space**
   Space needed to store the compiled version of program. It depends on
   i. Compiler used
   ii. Options specified at the time of compilation
   e.g., whether optimization specified, Is there any overlay option etc.
   iii. Target computer
   e.g., for performing floating point arithmetic, if hardware present or not.

2. **Data Space**
   Space needed to store constant and variable values. It has two components:
   i. Space for constants:
   e.g., value '3' in program 1.1
   Space for simple variables:
   e.g., variables a,b,c in program 1.1

Program 1.1

```
int add (int a, int b, int c)
{
        return (a+b+c)/3;
}
```

ii. Space for component variables like arrays, structures, dynamically allocated memory.
e.g., variables a in program 1.2

Program 1.2

```
int add1 (int a[], int n)
{
    If (n>0)
        return add1 (a, n-1) + a[n-1];
     else
        return 0;
}
```

3. **Environment stack space**
   Environment stack is used to store information to resume execution of partially completed functions. When a function is invoked, following data are stored in Environment stack.
   i. Return address.
   ii. Value of local and formal variables.
   iii. Binding of all reference and constant reference parameters.

Space needed by the program can be divided into two parts.

i. Fixed part independent of instance characteristics. E.g., code space, simple variables, fixed size component variables etc.
ii. Variable part. Space for component variables with space depends on particular instance. Value of local and formal variables.

The space requirement of a program p is written as S (P) = c + S_p (instance characteristics), where c is a constant. When analyzing space complexity of a program, we will concentrate on estimating S_p (instance characteristics).

For a given problem, we shall need to first determining which instance characteristics to use to measure the space requirement. So this is very problem specific.

Example 1 (Refer Program 1.1)

Space for variables a, b, c and return variable add depends on the type of the variables. But this utilizes negligible space. No instance characteristics. Hence S_p(TC) = 0.

Example 2 (Consider the program given below)

Program 1.3

```
    int sum (int *a, int n)
1   {
2       int s=0;
3       for (i=0; i<n; i++)
4           s+ = a[i];
5       return s;
6   }
```

Space for a [] is address of a [0] and space for i, and n requires one word. No instance characteristics. Hence S_p(TC) = 0.

Example 3 (Refer Program 1.2)

Instance characteristics depend on values of n. Recursive stack space includes space for formal parameters, local variables and return address. So one word each for a[],n, return address and return variables. Hence for each pass it needs 4 words. Since the depth of recursion is n+1, the recursion stack space needed is 4(n+1). So S_p (TC) = 4(n+1).

1.2.2 Time Complexity

Time complexity of an algorithm is the amount to time needed by the program for its completion. Time taken is the sum of the compile time and the execution time (run time). Compile time does not depend on instantaneous characteristics. Hence we can ignore it. We can assume program will run several times without recompilation. So we concentrating only on run time $\mathbf{t_p}$. if we know the characteristics of compiler used, we can determine the number of additions, subtractions, multiplications, divisions, loads, stores etc.

So $t_p(n) = C_a$ ADD(n)+ C_s SUM(n) + C_m MUL(n) + C_d DIV(n)……..
Where, n denotes the instance characteristics. C_a, C_s, C_m, C_d are the time needed for addition, subtraction, multiplication, division etc. ADD, SUB, MUL, DIV are functions whose value is number of additions, subtractions etc.

But this is not an easy procedure. By experimenting we can determine the time. But it may vary for multi-user environment. So we count only **Program steps**. This is syntactically or semantically meaningful segment of a program that has an execution time and that is independent of the instance characteristics.

The number of steps of any program is to be assigned is depend on the nature of the statement. So we can calculate complexity in terms of

1. Comments
Non executable statement, hence step count = 0

2. Declarative Statements
Define or characterize variables and constants like (*int* , *long*, *enum*, …)
Statement enabling data types (*class, struct, union, template*)
Determine access statements (*public, private, protected, friend*)
Character functions (*void, virtual)*
Cost of the above statements are lumped into the cost of involving the function they are associated with. So these are considered as non executables, hence step count = 0

3. Expressions and Assignment Statements
For simple expressions, Step count = 1. But if expressions contain function call, step count is the cost of the invoking functions. This will be large if parameters are passed as call by value, because value of the actual parameters must assigned to formal parameters.

Assignment statements, general form is <variable> = <expr>. Step count = expr, unless size of <variable> is a function of instance characteristics. eg., a = b, where a and b are structures. In that case, Step count = size of <variable> + size of < expr >

4. Iterative Statements
While <expr> **do**
While <expr> **do**
Do .. While <expr**>**
Step count = Number of step count assignable to <expr>

For (<init-stmt>; <expr1>; <expr2>)
Step count = 1, unless the <init-stmt>, <expr1>,<expr2> are function of instance characteristics. If so, first execution of control part has step count as sum of count of <init-stmt> and <expr1>. For remaining executions, control part has step count as sum of count of <expr1> and <expr2>.

5. Switch Statements

Switch (<expr>) {
Case cond1 : <statement1>
Case cond2 : <statement2>
.
.
Default : <statement>
}

Switch (<expr>) has step count = cost of <expr>
Cost of **Cond** statements is its cost plus cost of all preceding statements.

6. If-else Statements

If (<expr>) <statement1>;
Else <statement2>;
Step count of **If** and **Else** is the cost of <expr>.

7. Function invocation

All function invocation has Step count = 1, unless it has parameters passed as called by value which depend s on instance characteristics. If so, Step count is the sum of the size of these values.
If function being invoked is recursive, consider the local variables also.

8. Memory management Statements

new object, **delete** object, **sizeof**(object), Step count =1.

9. Function Statements

Step count = 0, cost is already assigned to invoking statements.

10. Jump Statements

continue, break, goto has Step count =1
return <expr>: Step count =1, if no *expr* which is a function of instance characteristics. If there is, consider its cost also.

There are two ways to determine the number of steps needed by the program to solve a particular problem instance. First method is to introduce a new variable called count as globally. Initial value=0. Statements to increment count by the appropriate amount are introduced in to the program.

Example 4

Introducing a counter for each executable line we can rewrite the **program 1.2** as

```
int add1 (int a[], int n)
{
    count++ // for if
    If (n>0)
    {
        count++ // for return
        return add1 (a, n-1) + a[n-1];
     }
    else
    {
        count++ // for return
        return 0;
    }
 }
```

Case 1: n=0

$t_{add1} = 2$

Case 2: n>0

$2 + t_{add1} (n-1)$

$= 2 + 2 + t_{add1} (n-2)$

$= 2 * 2 + t_{add1} (n-2)$

.

.

$= 2n + t_{add1} (0)$

$= 2n + 2$

Example 5

Consider the following program given below.

Program 1.4

```
    int Madd (int a[][], int b[][], int c[][], int n)          //Add 2 mxn matrix a and b.
1   {
2       For (int i=0; i<m; i++)
3           For (int j=0; j<n; j++)
4                   c[i][j] = a[i][j] + b[i][j];
5   }
```

Introducing a counter for each executable line we can rewrite the program as

```
int Madd (int a[][], int b[][], int c[][], int n) //Add 2 mxn matrix a and b.
{
    For (int i=0; i<m; i++)
    {
        count++ //for i
        For (int j=0; j<n; j++)
        {
                count++ //for j
                c[i][j] = a[i][j] + b[i][j];
```

```
                count++ //for assignment
            }
            count++ //for last j
        }
        count++ //for last i
    }
```

Line 4 is executed n times for each value of i. or simply n times.
Line 7 is executed n times for each value of I or total of mn times.
So step count=2mn+2m+1 (for j and for assignment both are executed mn times.

Step count does not reflect the complexity of statement. It is reflected in step per execution (s/e). **s/e** of a statement is the amount by which count changes as a result of the execution of that statement.

Second method is to determine the step count of the program is to build a table, in which we list the total number of steps contributed by each statement- number of steps/execution of the statement and the total number of times each statement is executed. By combining these quantities, the total contribution of each statement is obtained. By adding these, the step count for the entire program is obtained.

Example 6 (Refer program 1.2)

Line	**s/e**	**Frequency**		**Total Steps**	
		n=0	**n>0**	**n=0**	**n>0**
1	0	1	1	0	0
2	1	1	1	1	1
3	$1 + t_{add1}$ (n-1)	0	1	0	$1 + t_{add1}$ (n-1)
4	0	1	0	0	0
5	1	1	0	1	0
Total no. of steps				**2**	**$2 + t_{add1}$ (n-1)**

Example 7 (Refer program 1.3)

Line	**s/e**	**Frequency**	**Total Steps**
1	0	1	0
2	1	1	1
3	1	n+1	n+1
4	1	n	n
5	1	1	1
6	0	1	0
Total no. of steps			**2n + 3**

Example 8 (Refer program 1.4)

Line	**s/e**	**Frequency**	**Total Steps**

1	0	1	0
2	1	m+1	m+1
3	1	m(n+1)	m(n+1)
4	1	mn	mn
5	0	1	0
		Total no. of steps	**2mn + 2m + 1**

The time complexity of a program is given by the number of steps taken by the program to compute the function it was written for. The number of steps itself a function of instance characteristics like number of inputs, number of outputs, magnitude of input & output etc. Usually we wish to know how the computing time increases as the number of inputs increases. So the number of will be computed as a function of inputs alone.

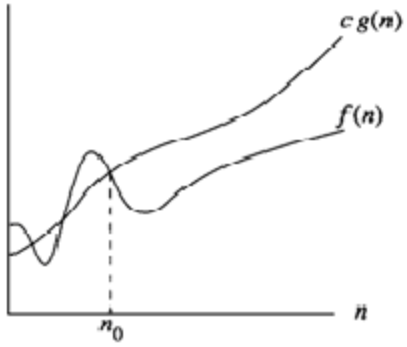
But some cases n is non adequate, and the time is depend on the position of the element. For example in searching algorithm the complexity is not only depend on number of inputs but also depend on the position of the key element. In these cases we have to determine three kinds of step counts- best case, worst case and average case.

Best case step count: Is the minimum number of steps that can be executed for the given parameters.
Worst case step count: Is the maximum number of steps that can be executed for the given parameters.
Average case step count: Is the average number of steps that can be executed for the given parameters.

1.2.3 Asymptotic Notations

Step count is to compare time complexity of two programs that compute same function and also to predict the growth in run time as instance characteristics changes. Determining exact step count is difficult and not necessary also. Because, these values are not exact quantities. We need only comparative statements like $c_1n^2 \leq t_p(n) \leq c_2n^2$ or $t_p(n,m)=c_1m+c_2n$ where c_1 and c_2 are constants.

For example, consider two programs with complexities $\mathbf{c_1n^2 + c_2n}$ and $\mathbf{c_3n}$ respectively.
$\mathbf{c_3n}$ is faster for large n.
For small values of n, complexity depend upon values of c_1, c_2 and c_3. But there will also be an n beyond which complexity of c_3n is better than that of $c_1n^2 + c_2n$.
If $\mathbf{c_3} = 100$, $\mathbf{c_2} = 2$, $\mathbf{c_1} = 1$
$\mathbf{c_1n^2 + c_2n <= c_3n}$ if $\mathbf{n <= 98}$
$\mathbf{c_3n <= c_1n^2 + c_2n}$ if $\mathbf{n > 98}$
So whatever be the value of c_1, c_2, c_3, there will be an **n** beyond which complexity of $\mathbf{c_3n}$ is better than that of $\mathbf{c_1n^2 + c_2n}$. This value of n is called **break-even point**. If this point is zero, c_3n is always faster (or at least as fast).The exact **break-even point** cannot determine analytically.



f and g are non negative functions. n_0 is the break even point.

Common asymptotic functions are given below.

Function	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	n log n
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

1.2.3.1 **Big 'Oh' Notation (O)**

O(g(n)) = { f(n) : there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }
It is the upper bound of any function. Hence it denotes the worse case complexity of any algorithm. We can represent it graphically as

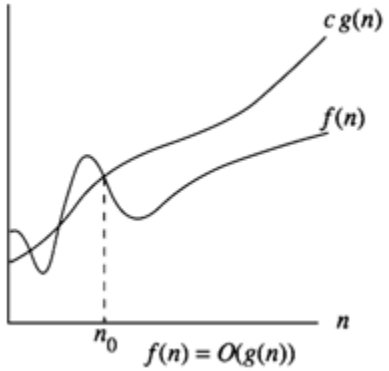


Fig 1.1

Examples

- **Linear Functions**

Example 9

$f(n) = 3n + 2 = O(n)$
General form is $f(n) \leq cg(n)$

When $n \geq 2$, $3n + 2 \leq 3n + n = 4n$
Hence $f(n) = O(n)$, here $c = 4$ and $n_0 = 2$

When $n \geq 1$, $3n + 2 \leq 3n + 2n = 5n$
Hence $f(n) = O(n)$, here $c = 5$ and $n_0 = 1$

Hence we can have different c, n_0 pairs satisfying for a given function.

Example 10
$f(n) = 3n + 3 = O(n)$
When $n \geq 3$, $3n + 3 \leq 3n + n = 4n$
Hence $f(n) = O(n)$, here $c = 4$ and $n_0 = 3$

Example 11
$f(n) = 100n + 6 = O(n)$
When $n \geq 6$, $100n + 6 \leq 100n + n = 101n$
Hence $f(n) = O(n)$, here $c = 101$ and $n_0 = 6$

- **Quadratic Functions**

Example 12
$f(n) = 10n^2 + 4n + 2 = O(n^2)$
When $n \geq 2$, $10n^2 + 4n + 2 \leq 10n^2 + 5n$
When $n \geq 5$, $5n \leq n^2$, $10n^2 + 4n + 2 \leq 10n^2 + n^2 = 11n^2$
Hence $f(n) = O(n^2)$, here $c = 11$ and $n_0 = 5$

Example 13
$f(n) = 1000n^2 + 100n - 6 = O(n^2)$
$f(n) \leq 1000n^2 + 100n$ for all values of n.
When $n \geq 100$, $5n \leq n^2$, $f(n) \leq 1000n^2 + n^2 = 1001n^2$
Hence $f(n) = O(n^2)$, here $c = 1001$ and $n_0 = 100$

- **Exponential Functions**

Example 14
$f(n) = 6*2^n + n^2 = O(2^n)$
When $n \geq 4$, $n^2 \leq 2^n$
So $f(n) \leq 6*2^n + 2^n = 7*2^n$
Hence $f(n) = O(2^n)$, here $c = 7$ and $n_0 = 4$

- **Constant Functions**

Example 15

f(n) = 10
f(n) = O(1), because f(n) ≤ 10*1
f=O(g(n)) means that *asymptotically* (as n gets really large), g(n) grows at least as fast as f(n).

1.2.3.2 **Omega Notation (Ω)**

Ω (g(n)) = { f(n) : there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ }
It is the lower bound of any function. Hence it denotes the best case complexity of any algorithm. We can represent it graphically as

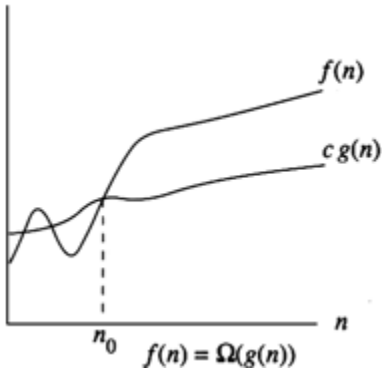


Fig 1.2

Example 16
f(n) = 3n + 2
3n + 2 > 3n for all n.
Hence f(n) = Ω(n)
Similarly we can solve all the examples specified under Big 'Oh'.

1.2.3.3 **Theta Notation (Θ)**

Θ(g(n)) = {f(n) : there exist positive constants c_1,c_2 and n_0 such that $c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$ }
If f(n) = Θ(g(n)), all values of n right to n_0 f(n) lies on or above c_1g(n) and on or below c_2g(n). Hence it is asymptotic tight bound for f(n).

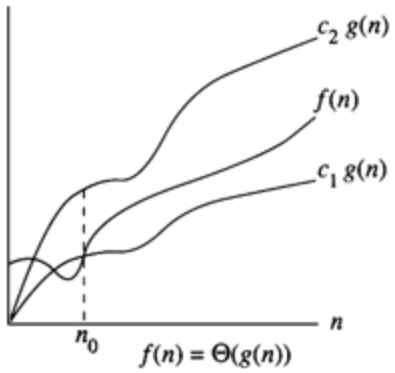


Fig 1.3

Example 17
f(n) = 3n + 2
f(n) = Θ(n) because f(n) = O(n) , n ≥ 2.

Similarly we can solve all examples specified under Big'Oh'.

The time complexity of a program can be represented as some instance characteristics. This can be used to compare two programs performing same task. But this is actually meant for greater values of n.
Fig 1.4 shows how the various functions grow with n.

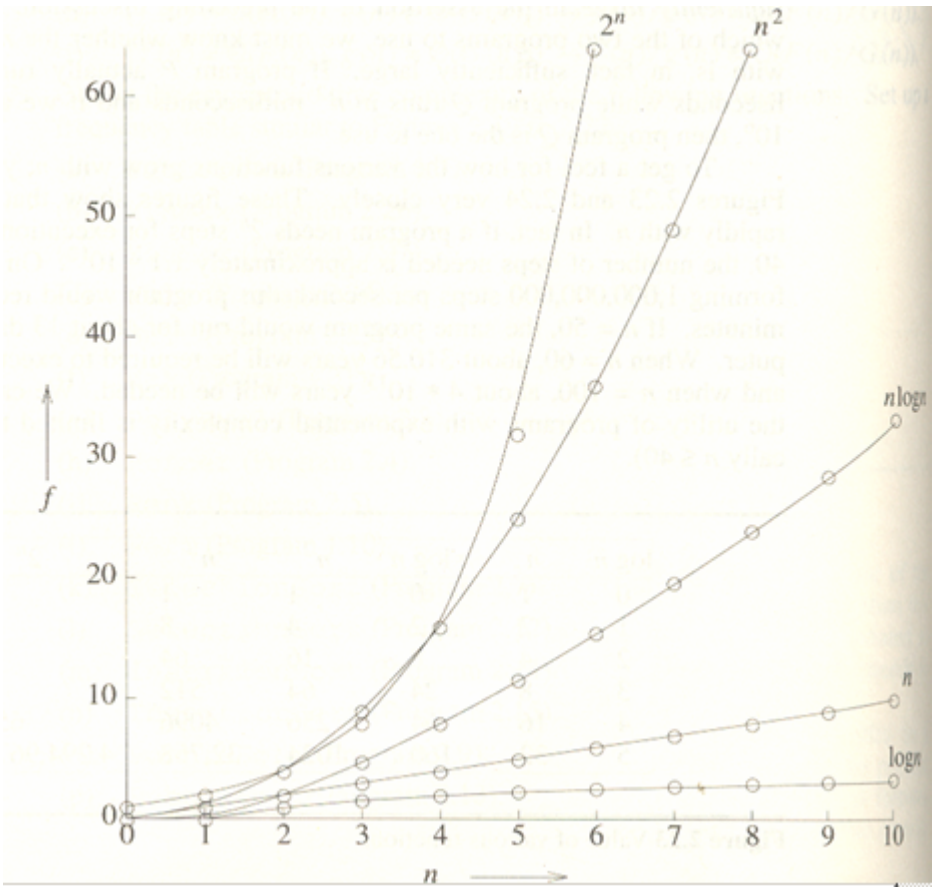


Fig 1.4

1.3 Performance measurement

Performance measurement is concerned with obtaining the actual space and time requirements of a program. These quantities are dependent on the particular compiler used. We make the assumption that each program will be compiled once and then execute several times. The space and time is needed for compilation is important for program testing. We shall not consider measuring the run time space requirements of a program. We will focus on measuring the computing time of a program.

To obtain the run time of a program, we need a clocking function. For each instance size, a repletion factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.

1.4 System Life Cycle

Large scale computer program is considered as a system. This system contains many complex interacting parts. These programs undergo a development process called system life cycle. The life cycle consists of different phases like requirement, analysis, design, and verification. These phases are highly interrelated.

The SYSTEM LIFE CYCLE is a term used to describe the stages in an IT project. The various stages are described below.

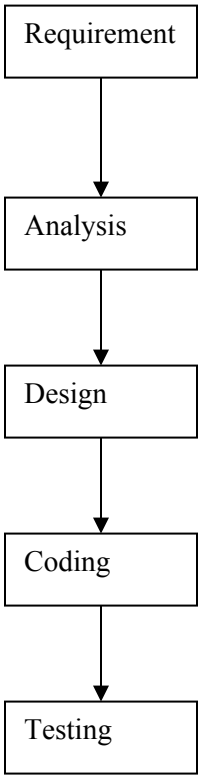
a) **Requirement Specification**: All large programming projects begin with a set of specifications that define the purpose of the project. This describes the information that the programmers are given (input) and the result that we must produce (output). This gives a set of specification.

b) **Analysis**: In this phase we begin to break the problem into manageable pieces.
There are two approaches to analysis: Bottom Up and Top Down.
The bottom up is an older one, unstructured strategy that places an early emphasis on the coding fine projects.
Top down approach begins by developing a high level plan for dividing the program. This technique generates diagrams that are used to design the system.

c) Design: In this phase the designer approaches the system from the perspectives of the data object that the program needs and operation performed on them. This describes how the segments are interconnected.

d) Coding/Implementation: Write the algorithm for each operation and implement using any programming language.

e) Verification/Testing: This phase consists of developing correctness of the program, testing the program with variety of input data and removing the errors.


Requirement
Analysis
Design
Coding
Testing