Unit IV:

Sorting: Bubble sort - Insertion sort – Selection sort – Quick sort – Merge sort – Radix sort – Heap sort. **Trees:** Trees – Binary Trees – Operations on Binary Trees –Traversal of a Binary Tree – Threaded Binary Tree - Binary Search Trees (BST) – Inserting and Deleting in a BST.

What is Sorting?

- ❖ Sorting is a process of ordering or placing a list of elements from a collection in some kind of order.
- ❖ It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order.
- ❖ It arranges the data in a sequence which makes searching easier.

Sorting Techniques

Sorting technique depends on the situation. It depends on two parameters.
1. Execution time of program that means time taken for execution of program.
2. Space that means space taken by the program.
3. Sorting techniques are differentiated by their efficiency and space requirements.

TYPES OF SORTING

Sorting can be performed using several techniques or methods, as follows:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
4. Merge Sort
5. Radix Sort
6. Heap Sort

1. Bubble Sort

- Bubble sort is a type of sorting.
- It is used for sorting 'n' (number of items) elements.
- It compares all the elements one by one and sorts them based on their values.

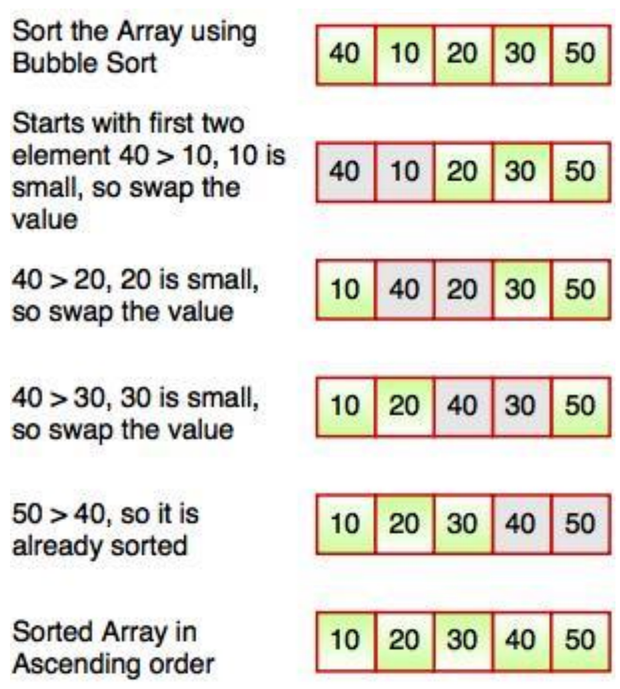

Sort the Array using Bubble Sort
40 10 20 30 50
Starts with first two element 40 > 10, 10 is small, so swap the value
40 10 20 30 50
40 > 20, 20 is small, so swap the value
10 40 20 30 50
40 > 30, 30 is small, so swap the value
10 20 40 30 50
50 > 40, so it is already sorted
10 20 30 40 50
Sorted Array in Ascending order
10 20 30 40 50

Fig. Working of Bubble Sort

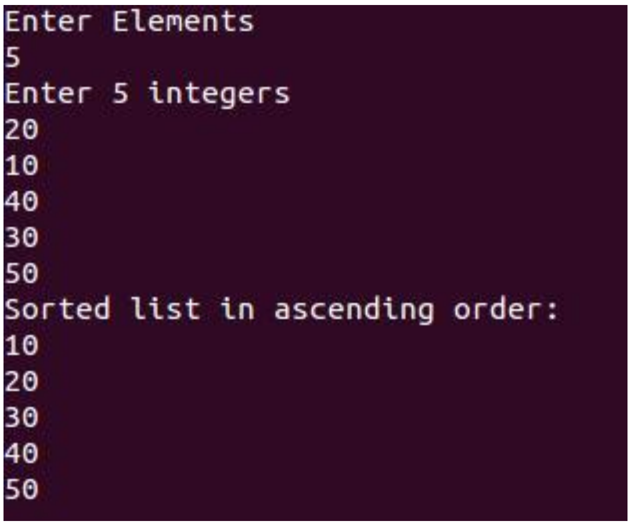
- The above diagram represents how bubble sort actually works. This sort takes $O(n^2)$ time. It starts with the first two elements and sorts them in ascending order.
- Bubble sort starts with first two elements. It compares the element to check which one is greater.
- In the above diagram, element 40 is greater than 10, so these values must be swapped. This operation continues until the array is sorted in ascending order.

Example: Program for Bubble Sort

```
#include <stdio.h>
void bubble_sort(long [], long);
int main()
{
  long array[100], n, c, d, swap;
  printf("Enter Elements\n");
  scanf("%ld", &n);
  printf("Enter %ld integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%ld", &array[c]);
  bubble_sort(array, n);
  printf("Sorted list in ascending order:\n");
```

```
 for ( c = 0 ; c < n ; c++ )
    printf("%ld\n", array[c]);
 return 0;
}
void bubble_sort(long list[], long n)
{
 long c, d, t;
 for (c = 0 ; c < ( n - 1 ); c++)
 {
   for (d = 0 ; d < n - c - 1; d++)
   {
     if (list[d] > list[d+1])
     {
       /* Swapping */
       t        = list[d];
       list[d]   = list[d+1];
       list[d+1] = t;
     }
   }
 }
}
```

Output:



```
Enter Elements
5
Enter 5 integers
20
10
40
30
50
Sorted list in ascending order:
10
20
30
40
50
```

Insertion Sort

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.
- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

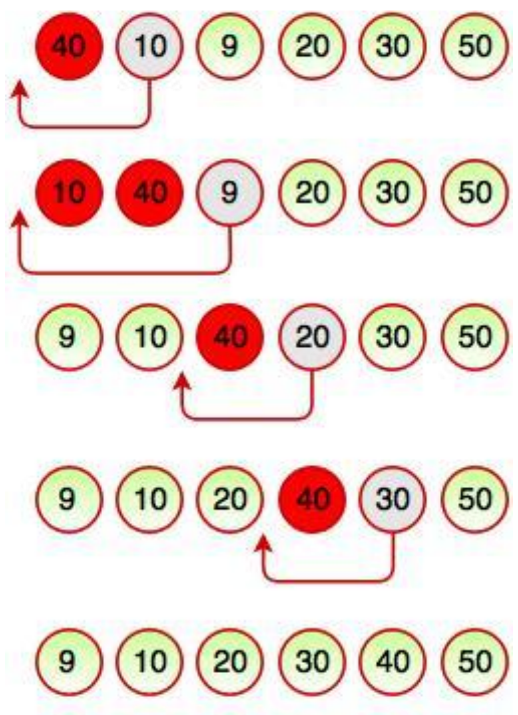


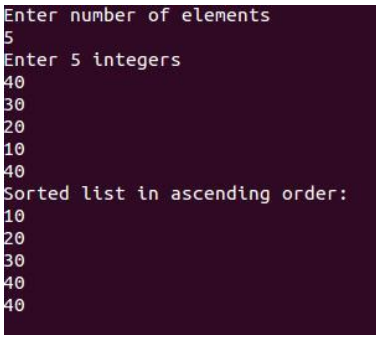
Fig. Working of Insertion Sort

- The above diagram represents how insertion sort works. Insertion sort works like the way we sort playing cards in our hands. It always starts with the second element as key. The key is compared with the elements ahead of it and is put it in the right place.
- In the above figure, 40 has nothing before it. Element 10 is compared to 40 and is inserted before 40. Element 9 is smaller than 40 and 10, so it is inserted before 10 and this operation continues until the array is sorted in ascending order.

Example: Program for Insertion Sort

```
#include <stdio.h>
int main()
{
 int n, array[1000], c, d, t;
 printf("Enter number of elements\n");
 scanf("%d", &n);
 printf("Enter %d integers\n", n);
 for (c = 0; c < n; c++)
 {
   scanf("%d", &array[c]);
 }
 for (c = 1 ; c <= n - 1; c++)
 {
   d = c;
   while ( d > 0 && array[d] < array[d-1])
   {
      t          = array[d];
      array[d]   = array[d-1];
      array[d-1] = t;
      d--;
   }
 }
 printf("Sorted list in ascending order:\n");
 for (c = 0; c <= n - 1; c++)
 {
    printf("%d\n", array[c]);
 }
 return 0;
}
```

Output:



```
Enter number of elements
5
Enter 5 integers
40
30
20
10
40
Sorted list in ascending order:
10
20
30
40
40
```

Selection Sort

- Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position. Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

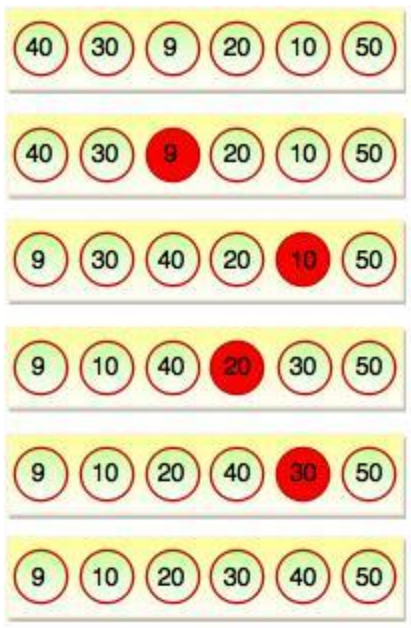


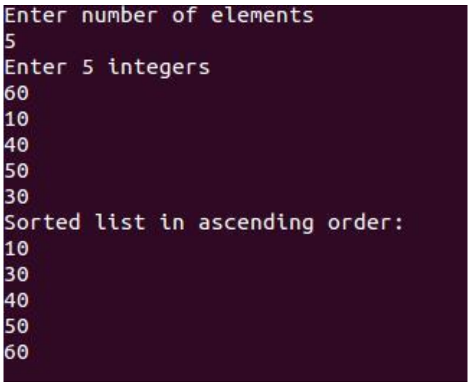
Fig. Working of Selection Sort

- In the above diagram, the smallest element is found in first pass that is 9 and it is placed at the first position. In second pass, smallest element is searched from the rest of the element excluding first element. Selection sort keeps doing this, until the array is sorted.

Example: Program for Selection Sort

```
#include <stdio.h>

int main()
```

```
{
  int array[100], n, c, d, position, swap;
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for ( c = 0 ; c < n ; c++ )
    scanf("%d", &array[c]);
  for ( c = 0 ; c < ( n - 1 ) ; c++ )
  {
    position = c;
    for ( d = c + 1 ; d < n ; d++ )
    {
      if ( array[position] > array[d] )
        position = d;
    }
    if ( position != c )
    {
      swap = array[c];
      array[c] = array[position];
      array[position] = swap;
    }
  }
  printf("Sorted list in ascending order:\n");
  for ( c = 0 ; c < n ; c++ )
    printf("%d\n", array[c]);
  return 0;
}
```
Output:



```
Enter number of elements
5
Enter 5 integers
60
10
40
50
30
Sorted list in ascending order:
10
30
40
50
60
```

Merge Sort

Merge sort is another sorting technique and has an algorithm that has a reasonably proficient space-time complexity - O(n log n) and is quite trivial to apply. This algorithm is based on splitting a list, into two comparable sized lists, i.e., left and right and then sorting each list and then merging the two sorted lists back together as one.

Merge sort can be done in two types both having similar logic and way of implementation. These are:

- Top down implementation
- Bottom up implementation

Below given figure shows how Merge Sort works:

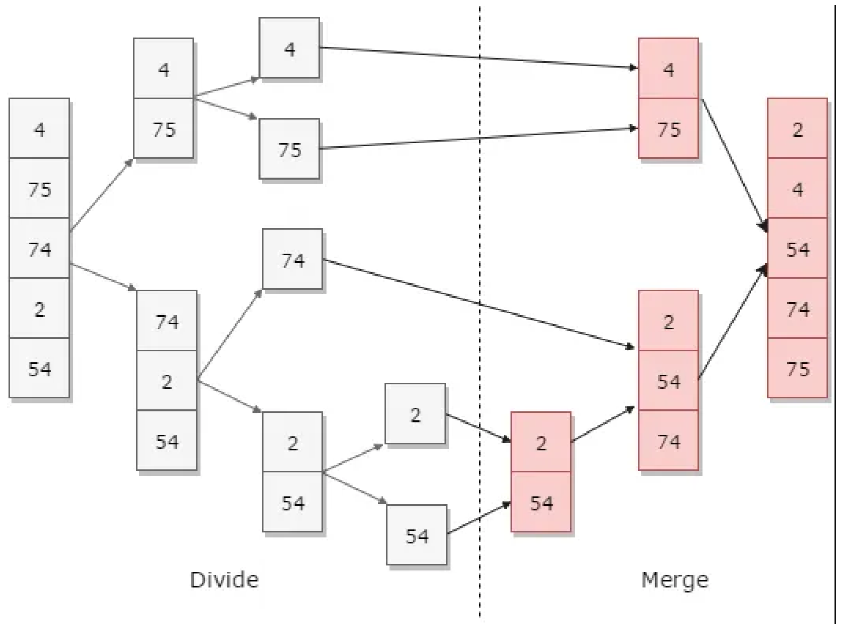
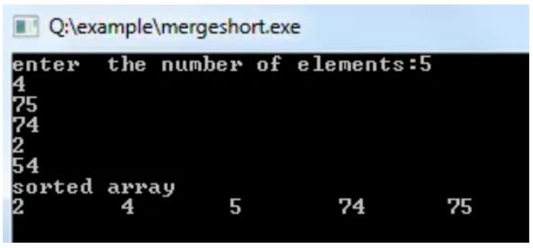


Fig: Merge Sort Technique

```
#include <iostream>
void merge(int *,int, int , int );
void mergesort(int *a, int low, int high)
{
  int mid;
  if (low < high)
  {
    mid=(low+high)/2;
```

```
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,high,mid);
    }
    return;
}
void merge(int *a, int low, int high, int mid)
{
    int i, j, k, c[50];
    i = low;
    k = low;
    j = mid + 1;
    while (i <= mid && j <= high)
    {
        if (a[i] < a[j])
        {
            c[k] = a[i];
            k++;
            i++;
        }       else
        {
            c[k] = a[j];
            k++;
            j++;
        }   }
    while (i <= mid)
    {
        c[k] = a[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        c[k] = a[j];
        k++;
        j++;
    }
    for (i = low; i < k; i++)
    {
        a[i] = c[i];
    }
}
int main()
{
    int a[30], i, b[30];
    cout<<"enter  the number of elements:";
    for (i = 0; i <= 5; i++) { cin>>a[i];
    }
    mergesort(a, 0, 4);
    cout<<"sorted array\n";
    for (i = 0; i < 5; i++)
    {
        cout<<a[i]<<"\t";
    }
    cout<<"enter  the number of elements:";
    for (i = 0; i < 5; i++) { cin>>b[i];
    }
    mergesort(b, 0, 4);
    cout<<"sorted array:\n";
    for (i = 0; i < 5; i++)
    {
        cout<<b[i]<<"\t";
    }
    getch();
}
```

Output:



Q:\example\mergeshort.exe

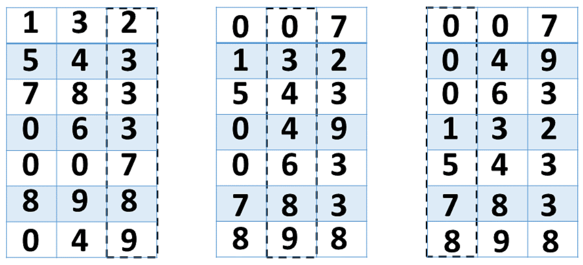
```
enter  the number of elements:5
4
75
74
2
54
sorted array
2       4       5       74      75
```

Radix Sort

What Is a Radix Sort Algorithm?

- Radix Sort is a linear sorting algorithm.
- Radix Sort's time complexity of O(nd), where n is the size of the array and d is the number of digits in the largest number.
- It is not an in-place sorting algorithm because it requires extra space.
- Radix Sort is a stable sort because it maintains the relative order of elements with equal values.
- Radix sort algorithm may be slower than other sorting algorithms such as merge sort and Quicksort if the operations are inefficient. These operations include sub-inset lists and delete functions, and the process of isolating the desired digits.
- Because it is based on digits or letters, radix sort is less flexible than other sorts. If the type of data changes, the Radix sort must be rewritten.

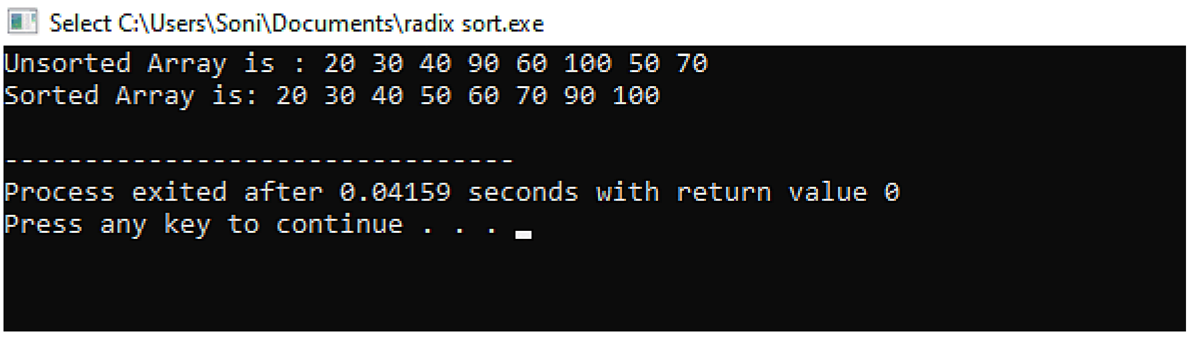
Sorting values into ones, tens, and hundreds place units



1	3	2		0	0	7		0	0	7
5	4	3		1	3	2		0	4	9
7	8	3		5	4	3		0	6	3
0	6	3		0	4	9		1	3	2
0	0	7		0	6	3		5	4	3
8	9	8		7	8	3		7	8	3
0	4	9		8	9	8		8	9	8

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int Max_value(int Array[], int n) // This function gives maximum value in array[]
{
  int i;
  int maximum = Array[0];
  for (i = 1; i < n; i++){
    if (Array[i] > maximum)
      maximum = Array[i];
    }
    return maximum;
}
void radixSortalgorithm(int Array[], int n)
{
    int i,digitPlace = 1;
    int result_array[n]; // resulting array
    int largest = Max_value(Array, n);
    while(largest/digitPlace >0){
        int count_array[10] = {0};
        for (i = 0; i < n; i++) //Store the count of "keys" or digits in count[]
            count_array[ (Array[i]/digitPlace)%10 ]++;
        for (i = 1; i < 10; i++)
            count_array[i] += count_array[i - 1];
        for (i = n - 1; i >= 0; i--) // Build the resulting array
        {
            result_array[count_array[ (Array[i]/digitPlace)%10 ] - 1] = Array[i];
            count_array[ (Array[i]/digitPlace)%10 ]--;
        }
        for (i = 0; i < n; i++)
            Array[i] = result_array[i];
            digitPlace *= 10;
    }
}
void displayArray(int Array[], int n)
{
    int i;
    for (i = 0; i < n; i++)
    printf("%d ", Array[i]);
    printf("\n");
}
int main()
{
    int array1[] = {20,30,40,90,60,100,50,70};
    int n = sizeof(array1)/sizeof(array1[0]);
    printf("Unsorted Array is : ");
    displayArray(array1, n);
    radixSortalgorithm(array1, n);
    printf("Sorted Array is: ");
    displayArray(array1, n);
    return 0;
}
```

Output



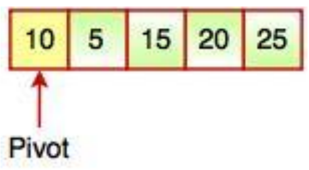
```
Select C:\Users\Soni\Documents\radix sort.exe
Unsorted Array is : 20 30 40 90 60 100 50 70
Sorted Array is: 20 30 40 50 60 70 90 100

--------------------------------
Process exited after 0.04159 seconds with return value 0
Press any key to continue . . .
```

Quick Sort

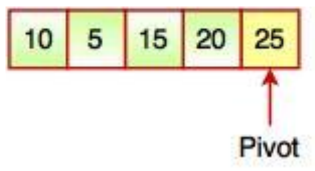
- Quick sort is also known as **Partition-exchange sort** based on the rule of **Divide and Conquer.**
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space, only O(n log n) space is required.
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

There are different versions of quick sort which choose the pivot in different ways:

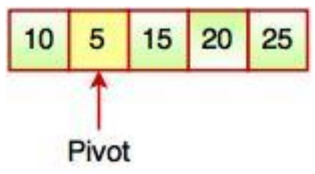
1. First element as pivot



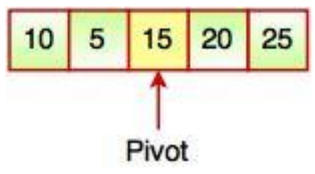
2. Last element as pivot



3. Random element as pivot



4. Median as pivot



Algorithm for Quick Sort

Step 1: Choose the highest index value as pivot.

Step 2: Take two variables to point left and right of the list excluding pivot.

Step 3: Left points to the low index.

Step 4: Right points to the high index.

Step 5: While value at left < (Less than) pivot move right.

Step 6: While value at right > (Greater than) pivot move left.

Step 7: If both Step 5 and Step 6 does not match, swap left and right.

Step 8: If left = (Less than or Equal to) right, the point where they met is new pivot.

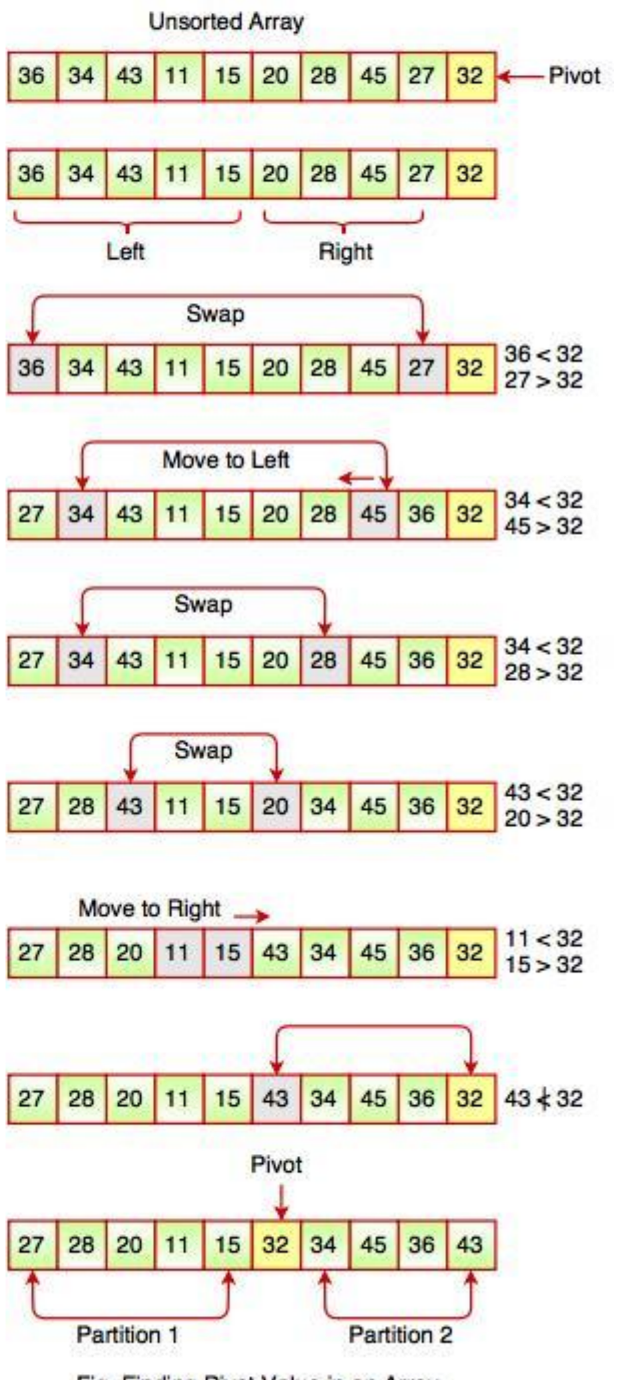


Fig. Finding Pivot Value in an Array

The above diagram represents how to find the pivot value in an array. As we see, pivot value divides the list into two parts (partitions) and then each part is processed for quick sort. Quick sort is a recursive function. We can call the partition function again.

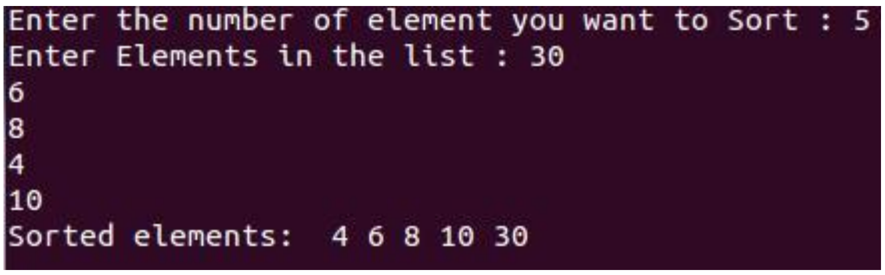
Example: Demonstrating Quick Sort

```
#include<stdio.h>
#include<conio.h>
```

```
void quicksort(int array[], int firstIndex, int lastIndex)
{
  int pivotIndex, temp, index1, index2;
  if(firstIndex < lastIndex)
  {

    pivotIndex = firstIndex;
    index1 = firstIndex;
    index2 = lastIndex;
        while(index1 < index2)
    {
      while(array[index1] <= array[pivotIndex] && index1 < lastIndex)
      {
        index1++;
      }
      while(array[index2]>array[pivotIndex])
      {
        index2--;
      }
      if(index1<index2)
      {
        //Swapping opertation
        temp = array[index1];
        array[index1] = array[index2];
        array[index2] = temp;
      }
    }
    //At the end of first iteration, swap pivot element with index2 element
    temp = array[pivotIndex];
    array[pivotIndex] = array[index2];
    array[index2] = temp;
    //Recursive call for quick sort, with partiontioning
    quicksort(array, firstIndex, index2-1);
    quicksort(array, index2+1, lastIndex);
  }
}
```

```
int main()
{
      int array[100],n,i;
   printf("Enter the number of element you want to Sort : ");
   scanf("%d",&n);
   printf("Enter Elements in the list : ");
   for(i = 0; i < n; i++)
   {
      scanf("%d",&array[i]);
   }
   quicksort(array,0,n-1);
   printf("Sorted elements: ");
   for(i=0;i<n;i++)
     printf(" %d",array[i]);
   getch();
   return 0;
}
```

Output:



```
Enter the number of element you want to Sort : 5
Enter Elements in the list : 30
6
8
4
10
Sorted elements:  4 6 8 10 30
```

<u>Heap Sort</u>

- Heap sort is a comparison based sorting algorithm.
- It is a special tree-based data structure.
- Heap sort is similar to selection sort. The only difference is, it finds largest element and places the it at the end.
- This sort is not a stable sort. It requires a constant space for sorting a list.
- It is very fast and widely used for sorting.

It has following two properties:

1. Shape Property
2. Heap Property

1. Shape property represents all the nodes or levels of the tree are fully filled. Heap data structure is a complete binary tree.

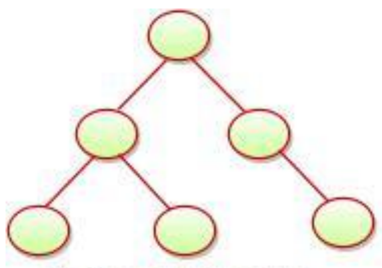


Fig. Complete-Binary Tree

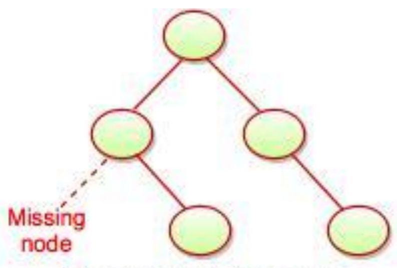


Fig. In-Complete-Binary Tree

2. Heap property is a binary tree with special characteristics. It can be classified into two types:

I. Max-Heap
II. Min Heap

I. Max Heap: If the parent nodes are greater than their child nodes, it is called a **Max-Heap.**

II. Min Heap: If the parent nodes are smaller than their child nodes, it is called a **Min-Heap.**

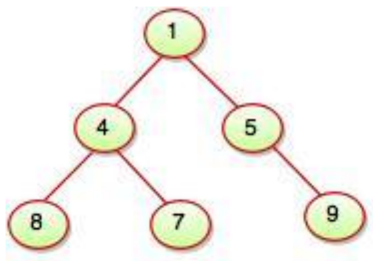


Fig. Min Heap

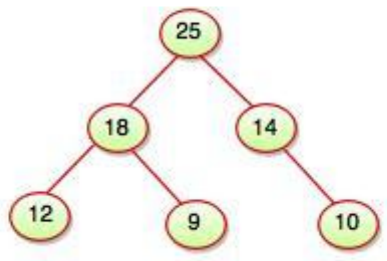


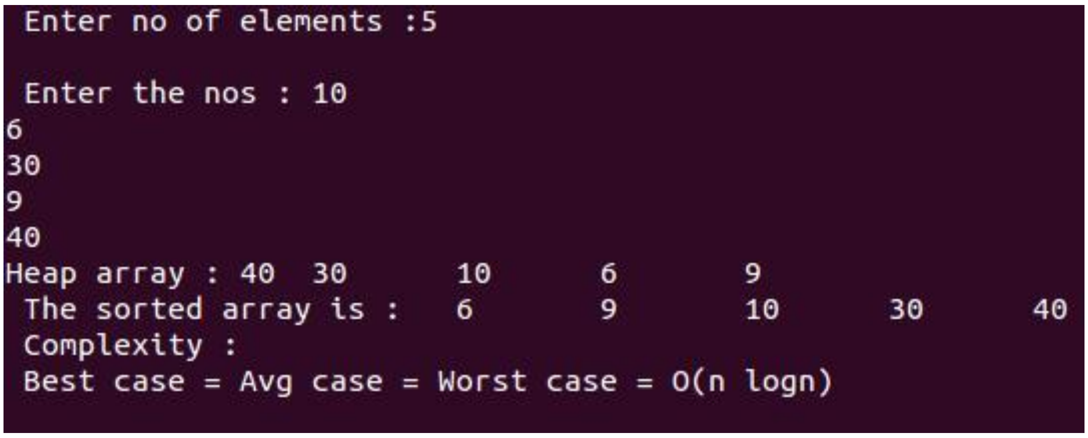
Fig. Max Heap

Example: Program for Heap Sort

```
#include <stdio.h>
void main()
{
   int heap[10], no, i, j, c, root, temp;
   printf("\n Enter no of elements :");
   scanf("%d", &no);
   printf("\n Enter the nos : ");
   for (i = 0; i < no; i++)
     scanf("%d", &heap[i]);
```

```
for (i = 1; i < no; i++)
{
   c = i;
   do
   {
      root = (c - 1) / 2;
      if (heap[root] < heap[c])   /* to create MAX heap array */
      {
         temp = heap[root];
         heap[root] = heap[c];
         heap[c] = temp;
      }
      c = root;
   } while (c != 0);
}
printf("Heap array : ");
for (i = 0; i < no; i++)
   printf("%d\t ", heap[i]);
for (j = no - 1; j >= 0; j--)
{
   temp = heap[0];
   heap[0] = heap[j];
   heap[j] = temp;
   root = 0;
   do
   {
      c = 2 * root + 1;
      if ((heap[c] < heap[c + 1]) && c < j-1)
         c++;
      if (heap[root]<heap[c] && c<j)
      {
         temp = heap[root];
         heap[root] = heap[c];
         heap[c] = temp;
      }
      root = c;
      } while (c < j);
    }
    printf("\n The sorted array is : ");
    for (i = 0; i < no; i++)
      printf("\t %d", heap[i]);
    printf("\n Complexity : \n Best case = Avg case = Worst case = O(n logn) \n");
}
```

Output:



```
 Enter no of elements :5

 Enter the nos : 10
6
30
9
40
Heap array : 40  30      10      6       9
 The sorted array is :   6       9       10      30      40
 Complexity :
 Best case = Avg case = Worst case = O(n logn)
```

TREE

What are trees?

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

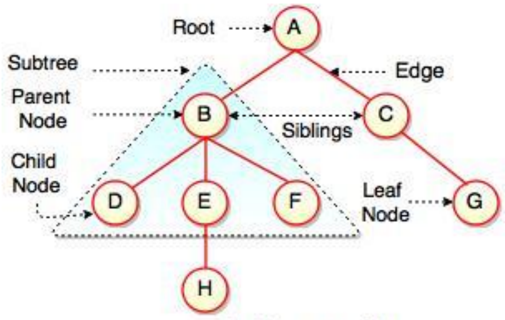


Fig. Structure of Tree

The above figure represents structure of a tree. Tree has 2 subtrees.

A is a parent of B and C.
B is called a child of A and also parent of D, E, F.

Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

Field	**Description**
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

In the above figure, D, F, H, G are **leaves**. B and C are **siblings**. Each node excluding a root is connected by a direct edge from exactly one other node
parent → children.

BINARY TREE

Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**. It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM).

Definition:
"A tree in which every node can have maximum of two children is called as Binary Tree."

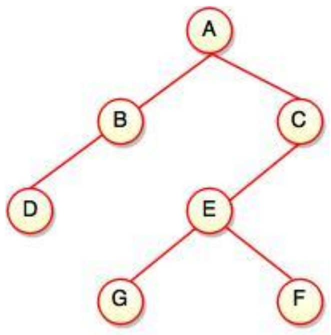


Fig. Binary Tree

The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

Representation of Binary Tree using Array
Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

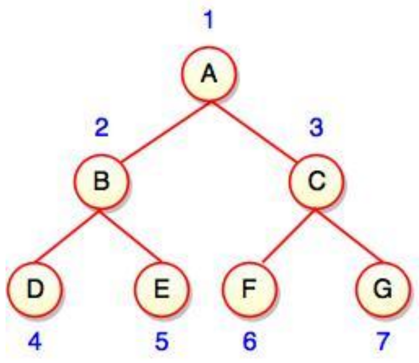


Fig. Binary Tree using Array

Array index is a value in tree nodes and array value gives to the parent node of that particular index or node. Value of the root node index is always -1 as there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

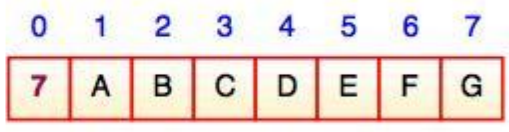


Fig. Location Number of an Array in a Tree

Location number of an array is used to store the size of the tree. The first index of an array that is '0', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index i is put into the array as its i th element.

The above figure shows how a binary tree is represented as an array. Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array.

Binary Search Tree

- Binary search tree is a binary tree which has special property called BST.
- BST property is given as follows:

For all nodes A and B,

I. If B belongs to the left subtree of A, the key at B is less than the key at A.

II. If B belongs to the right subtree of A, the key at B is greater than the key at A.

Each node has following attributes:

I. Parent (P), left, right which are pointers to the parent (P), left child and right child respectively.

II. Key defines a key which is stored at the node.

Definition:

"Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."

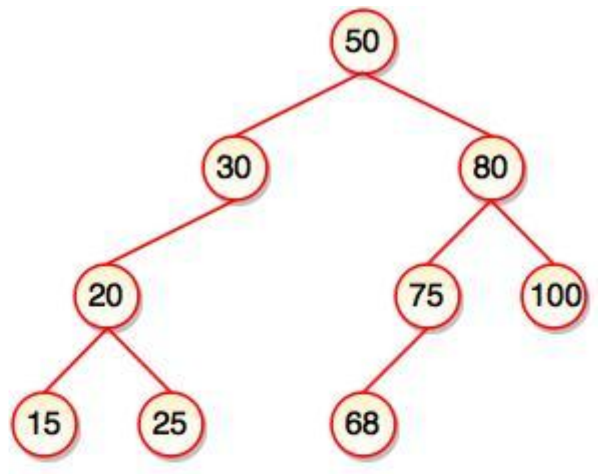


Fig. Binary Search Tree

- The above tree represents binary search tree (BST) where left subtree of every node contains smaller values and right subtree of every node contains larger value.
- Binary Search Tree (BST) is used to enhance the performance of binary tree.
- It focuses on the search operation in binary tree.

Note: Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.

Binary Search Tree Operations

Following are the operations performed on binary search tree:

1. Insert Operation

- Insert operation is performed with O(log n) time complexity in a binary search tree.
- Insert operation starts from the root node. It is used whenever an element is to be inserted.

The following algorithm shows the insert operation in binary search tree:

Step 1: Create a new node with a value and set its left and right to NULL.

Step 2: Check whether the tree is empty or not.

Step 3: If the tree is empty, set the root to a new node.

Step 4: If the tree is not empty, check whether a value of new node is smaller or larger than the node (here it is a root node).

Step 5: If a new node is smaller than or equal to the node, move to its left child.

Step 6: If a new node is larger than the node, move to its right child.

Step 7: Repeat the process until we reach to a leaf node.

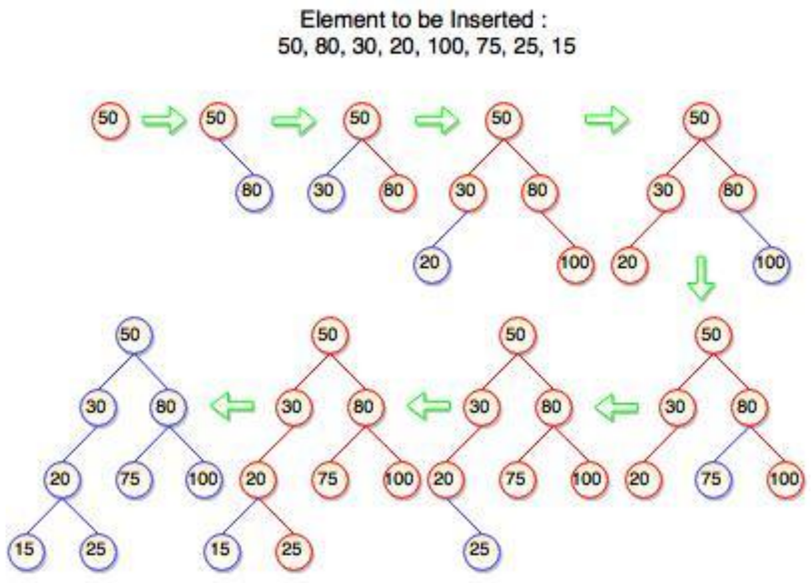


Fig. Insert Operation

The above tree is constructed a binary search tree by inserting the above elements {50, 80, 30, 20, 100, 75, 25, 15}. The diagram represents how the sequence of numbers or elements are inserted into a binary search tree.

2. Search Operation

- Search operation is performed with O(log n) time complexity in a binary search tree.
- This operation starts from the root node. It is used whenever an element is to be searched.

The following algorithm shows the search operation in binary search tree:

Step 1: Read the element from the user .

Step 2: Compare this element with the value of root node in a tree.

Step 3: If element and value are matching, display "Node is Found" and terminate the function.

Step 4: If element and value are not matching, check whether an element is smaller or larger than a node value.

Step 5: If an element is smaller, continue the search operation in left subtree.

Step 6: If an element is larger, continue the search operation in right subtree.

Step 7: Repeat the same process until we found the exact element.

Step 8: If an element with search value is found, display "Element is found" and terminate the function.
Step 9: If we reach to a leaf node and the search value is not match to a leaf node, display "Element is not found" and terminate the function.

BINARY TREE TRAVERSAL

Binary tree traversing is a process of accessing every node of the tree and exactly once. A tree is defined in a recursive manner. Binary tree traversal also defined recursively.

There are three techniques of traversal:
1. Preorder Traversal
2. Postorder Traversal
3. Inorder Traversal

1. Preorder Traversal
Algorithm for preorder traversal
Step 1 : Start from the Root.
Step 2 : Then, go to the Left Subtree.
Step 3 : Then, go to the Right Subtree.

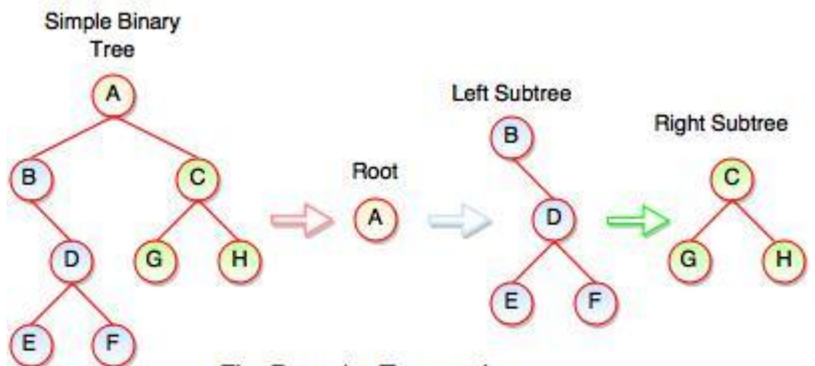


Fig. Preorder Traversal

The above figure represents how preorder traversal actually works.

Following steps can be defined the flow of preorder traversal:

Step 1 : A + B (B + Preorder on D (D + Preorder on E and F)) + C (C + Preorder on G and H)

Step 2 : A + B + D (E + F) + C (G + H)

Step 3 : A + B + D + E + F + C + G + H

Preorder Traversal : A B C D E F G H

2. Postorder Traversal

Algorithm for postorder traversal

Step 1 : Start from the Left Subtree (Last Leaf).

Step 2 : Then, go to the Right Subtree.

Step 3 : Then, go to the Root.

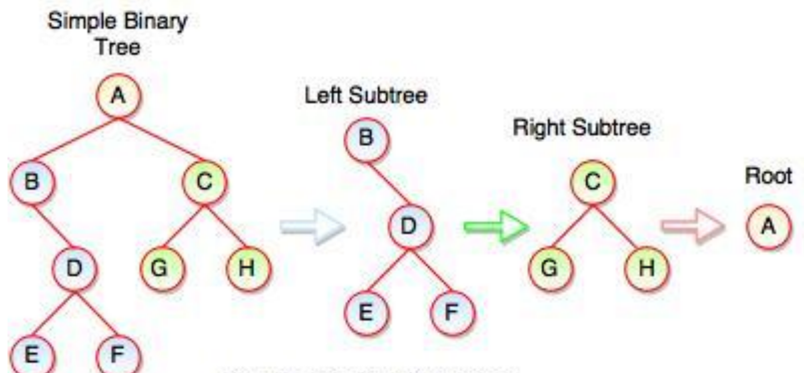


Fig. Postorder Traversal

The above figure represents how postorder traversal actually works.

Following steps can be defined the flow of postorder traversal:

Step 1 : As we know, preorder traversal starts from left subtree (last leaf) ((Postorder on E + Postorder on F) + D + B)) + ((Postorder on G + Postorder on H) + C) + (Root A)

Step 2 : (E + F) + D + B + (G + H) + C + A

Step 3 : E + F + D + B + G + H + C + A

Postorder Traversal : E F D B G H C A

3. Inorder Traversal

Algorithm for inorder traversal

Step 1 : Start from the Left Subtree.

Step 2 : Then, visit the Root.

Step 3 : Then, go to the Right Subtree.

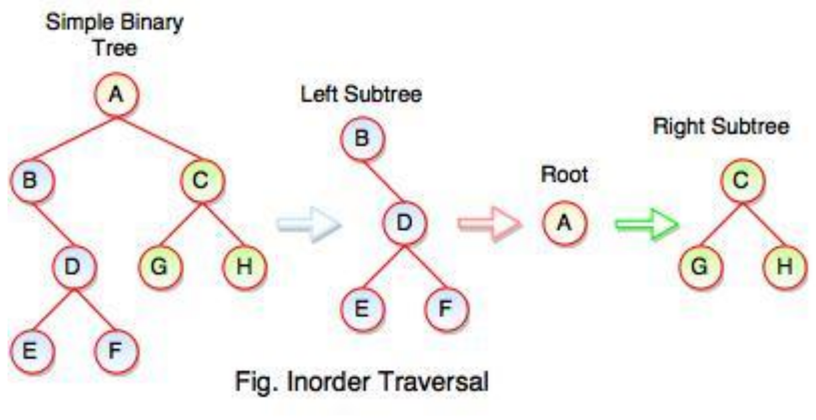


Fig. Inorder Traversal

The above figure represents how inorder traversal actually works.

Following steps can be defined the flow of inorder traversal:

Step 1 : B + (Inorder on E) + D + (Inorder on F) + (Root A) + (Inorder on G) + C (Inorder on H)

Step 2 : B + (E) + D + (F) + A + G + C + H

Step 3 : B + E + D + F + A + G + C + H

Inorder Traversal : B E D F A G C H

Example: Program for Binary Tree

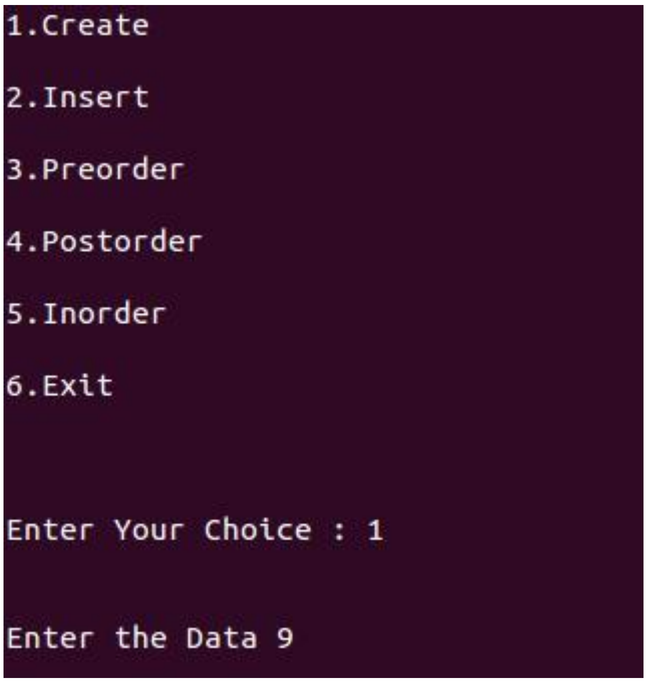
```
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *rlink;
    struct node *llink;
}*tmp=NULL;
```

```
typedef struct node NODE;
NODE *create();
void preorder(NODE *);
void inorder(NODE *);
void postorder(NODE *);
void insert(NODE *);

int main()
{
    int n,i,ch;
    do
    {
        printf("\n\n1.Create\n\n2.Insert\n\n3.Preorder\n\n4.Postorder\n\n5.Inorder\n\n6.Exit
\n\n");
        printf("\n\nEnter Your Choice : ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                tmp=create();
                break;
            case 2:
                insert(tmp);
                break;
            case 3:
                printf("\n\nDisplay Tree in Preorder Traversal : ");
                preorder(tmp);
                break;
            case 4:
                printf("\n\nDisplay Tree in Postorder Traversal : ");
                postorder(tmp);
                break;
            case 5:
                printf("\n\nDisplay Tree in Inorder Traversal : ");
                inorder(tmp);
                break;
            case 6:
                exit(0);
                default:
                printf("\n Inavild Choice..");
        }
    }
    while(n!=5);
}
void insert(NODE *root)
{
    NODE *newnode;
    if(root==NULL)
    {
        newnode=create();
        root=newnode;
    }
    else
    {
        newnode=create();
        while(1)
        {
            if(newnode->data<root->data)
            {
                if(root->llink==NULL)
                {
                    root->llink=newnode;
                    break;
                }
                root=root->llink;
            }
            if(newnode->data>root->data)
            {
                if(root->rlink==NULL)
                {
                    root->rlink=newnode;
                    break;
                }
                root=root->rlink;
            }
        }
    }
}
NODE *create()
{
    NODE *newnode;
    int n;
    newnode=(NODE *)malloc(sizeof(NODE));
    printf("\n\nEnter the Data ");
    scanf("%d",&n);
    newnode->data=n;
    newnode->llink=NULL;
    newnode->rlink=NULL;
    return(newnode);
}
void postorder(NODE *tmp)
{
    if(tmp!=NULL)
    {
        postorder(tmp->llink);
        postorder(tmp->rlink);
        printf("%d->",tmp->data);
    }
}
void inorder(NODE *tmp)
{
    if(tmp!=NULL)
    {
        inorder(tmp->llink);
        printf("%d->",tmp->data);
        inorder(tmp->rlink);
    }
}
```

```
void preorder(NODE *tmp)
{
    if(tmp!=NULL)
    {
        printf("%d->",tmp->data);
        preorder(tmp->llink);
        preorder(tmp->rlink);
    }
}
```

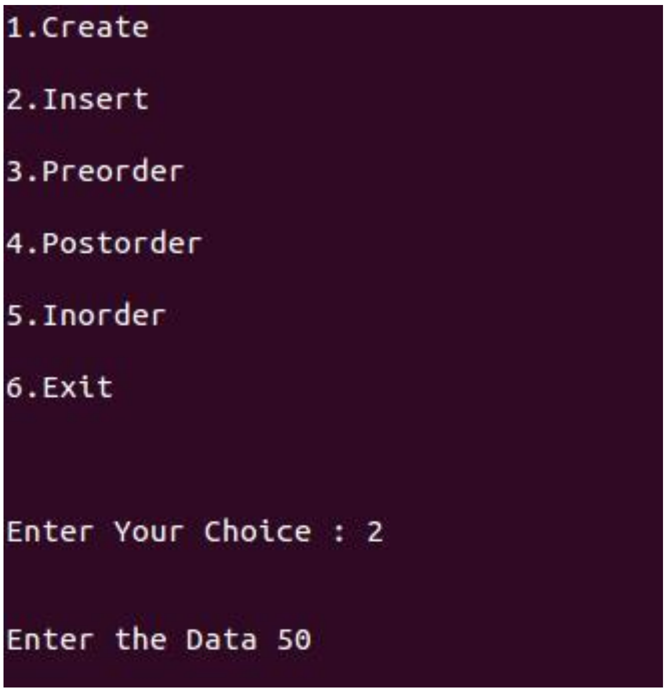
Output:

1. Create



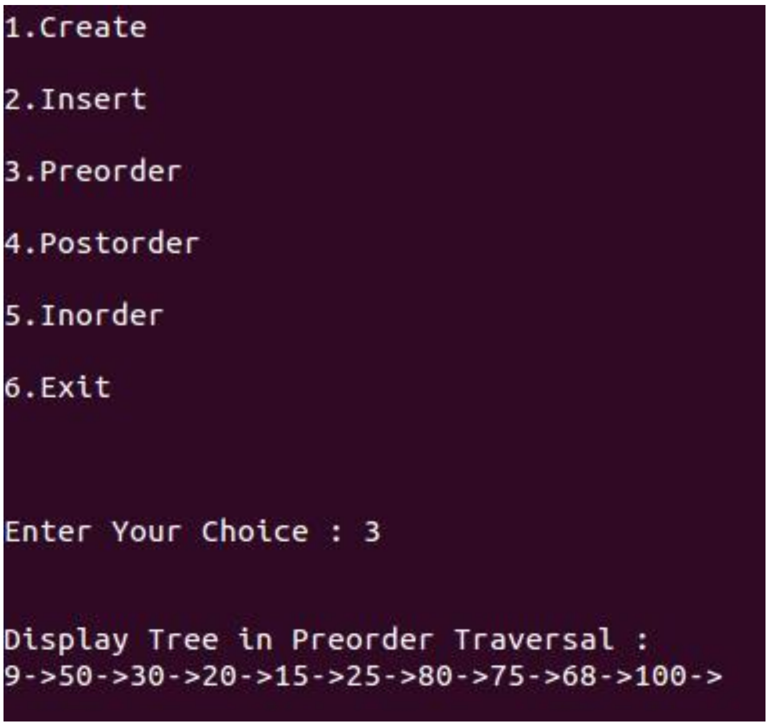
```
1.Create

2.Insert

3.Preorder

4.Postorder

5.Inorder

6.Exit


Enter Your Choice : 1


Enter the Data 9
```

2. Insert



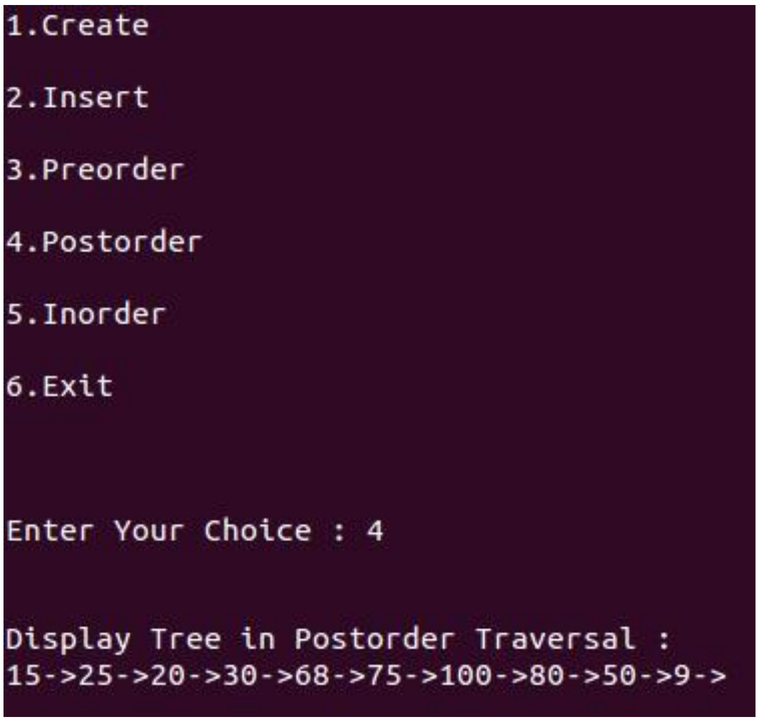
```
1.Create

2.Insert

3.Preorder

4.Postorder

5.Inorder

6.Exit


Enter Your Choice : 2


Enter the Data 50
```

3. Pre-order Traversal



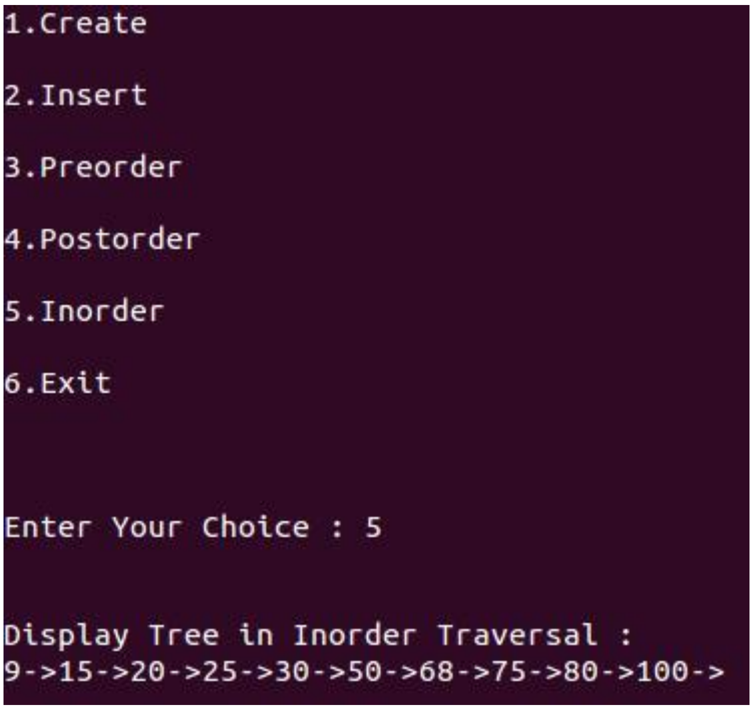
```
1.Create

2.Insert

3.Preorder

4.Postorder

5.Inorder

6.Exit


Enter Your Choice : 3


Display Tree in Preorder Traversal :
9->50->30->20->15->25->80->75->68->100->
```

4. Post-order Traversal



```
1.Create

2.Insert

3.Preorder

4.Postorder

5.Inorder

6.Exit


Enter Your Choice : 4


Display Tree in Postorder Traversal :
15->25->20->30->68->75->100->80->50->9->
```

5. In-order Traversal



```
1.Create

2.Insert

3.Preorder

4.Postorder

5.Inorder

6.Exit


Enter Your Choice : 5


Display Tree in Inorder Traversal :
9->15->20->25->30->50->68->75->80->100->
```

Threaded Binary Tree

There are four types of binary tree:

1. Full Binary Tree
2. Complete Binary Tree
3. Skewed Binary Tree
4. Extended Binary Tree

1. Full Binary Tree

- If each node of binary tree has either two children or no child at all, is said to be a **Full Binary Tree**.
- Full binary tree is also called as **Strictly Binary Tree**.

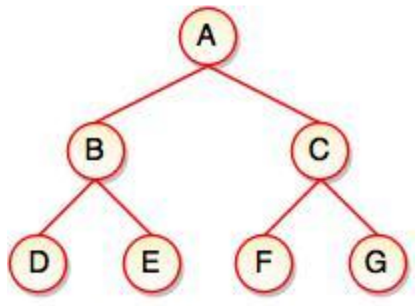


Fig. Full Binary Tree

- Every node in the tree has either 0 or 2 children.
- Full binary tree is used to represent mathematical expressions.

2. Complete Binary Tree

- If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.
- Complete binary tree is also called as **Perfect Binary Tree**.

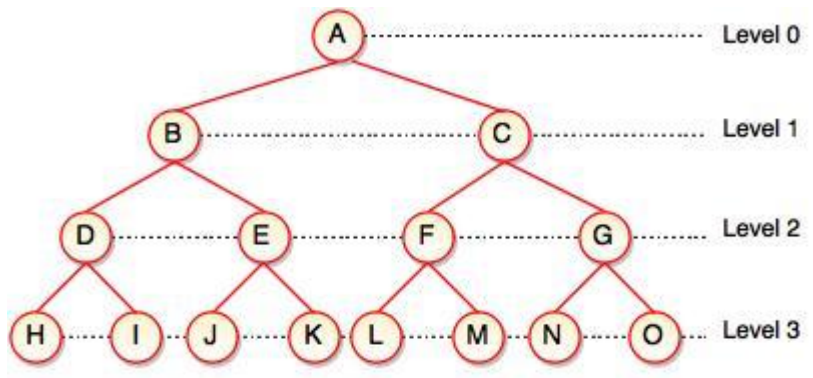


Fig. Complete Binary Tree

- In a complete binary tree, every internal node has exactly two children and all leaf nodes are at same level.
- For example, at Level 2, there must be $2^2 = 4$ nodes and at Level 3 there must be $2^3 = 8$ nodes.

3. Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

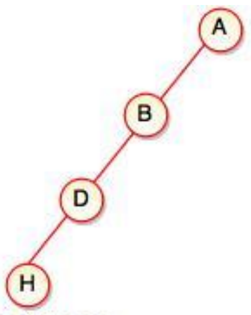


Fig. Left Skewed Binary Tree

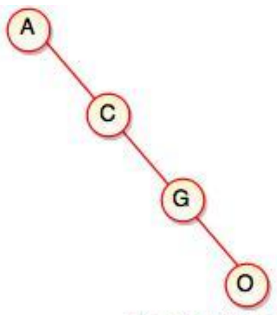


Fig. Right Skewed Binary Tree

- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

4. Extended Binary Tree

- Extended binary tree consists of replacing every null subtree of the original tree with special nodes.
- Empty circle represents internal node and filled circle represents external node.
- The nodes from the original tree are internal nodes and the special nodes are external nodes.
- Every internal node in the extended binary tree has exactly two children and every external node is a leaf. It displays the result which is a **complete binary tree**.

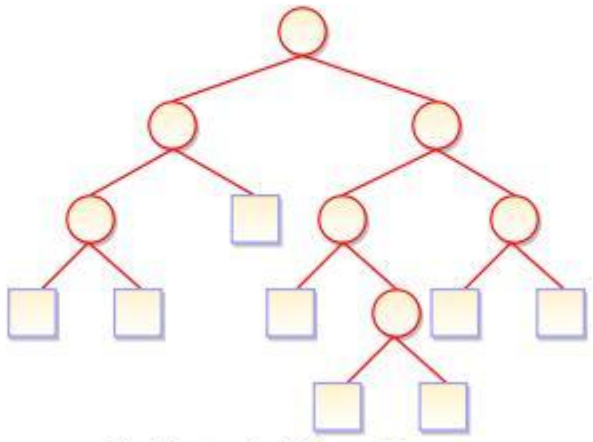


Fig. Extended Binary Tree

AVL Tree

- AVL tree is a height balanced tree.
- It is a self-balancing binary search tree.
- AVL tree is another balanced binary search tree.
- It was invented by **A**delson-**V**elskii and **L**andis.
- AVL trees have a faster retrieval.
- It takes O(logn) time for addition and deletion operation.
- In AVL tree, heights of left and right subtree cannot be more than one for all nodes.

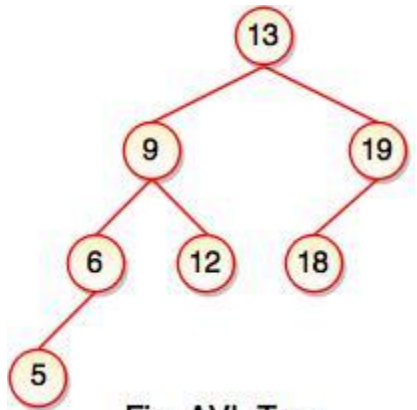
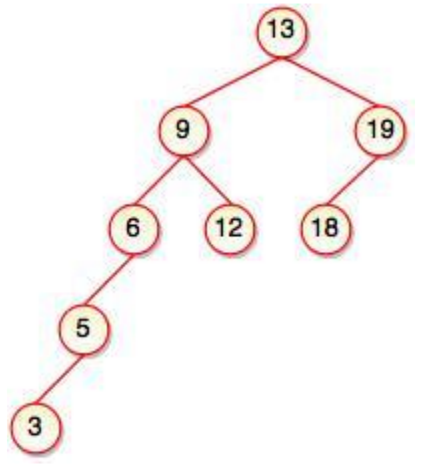


Fig. AVL Tree

- The above tree is AVL tree because the difference between heights of left and right subtrees for every node is less than or equal to 1.



- The above tree is not AVL because the difference between heights of left and right subtrees for 9 and 19 is greater than 1.
- It checks the height of the left and right subtree and assures that the difference is not more than 1. The difference is called balance factor.