Exp 6
Aim : Implementation of operations on BST

Theory:

Binary search tree

In computer science, a **binary search tree** (**BST**) is a binary tree data structure which has the following properties

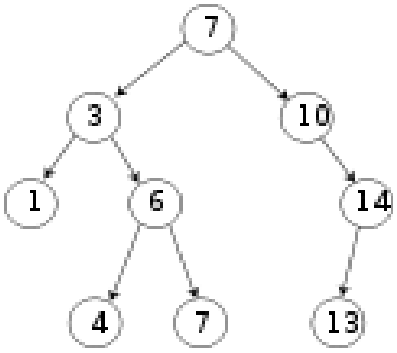
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.

Generally, the information represented by each node is a **record** rather than a single data element. However, for sequencing purposes, nodes are compared according to their **keys** rather than any part of their their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.



Operations

1. Insertion: Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value.

```
void InsertNode(Node* &treeNode, Node *newNode)
{
        if (treeNode == NULL)
        treeNode = newNode;
         else if (newNode->key < treeNode->key)
          InsertNode(treeNode->left, newNode);
         else
        InsertNode(treeNode->right, newNode);
}
```

2. **Searching:** We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree.

```
bool BinarySearchTree::search(int val)
{
    Node* next = this->root();
    while (next != 0)
    {
       if (val == next->value())
        {
          return true;
        }
       else if (val < next->value())
            next = next->left();
       else if (val > next->value())
            next = next->right();
     }
    //not found
    return false;
}
```

3. **Deletion:** There are several cases to be considered:

Deleting a leaf: Deleting a node with no children is easy, as we can simply remove it from the tree.

Deleting a node with one child: Delete it and replace it with its child.

Deleting a node with two children: Call the node to be deleted "N". Do not delete N. Instead, chose either its in-order successor node "R". Replace the value of N with the value of R, then delete R. (Note: R itself have up to one child)

conclusion : Thus we have implemented **operations on BST**

Exp 7

Aim : Implementation of DFS and BFS

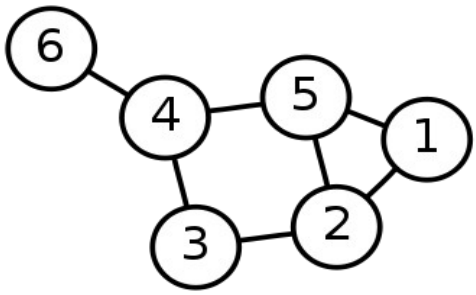
Theory:

Graph:

A graph G can be defined as a pair (V,E), where V is a set of vertices, and E is a set of edges between the vertices E.

Directed graph: A *graph* whose *edges* are *ordered* pairs of *vertices*.

Undirected graph: A *graph* whose *edges* are *unordered* pairs of *vertices*.



There are different ways to store graphs in a computer system

1. Adjacency list

   Much like the incidence list, each vertex has a list of which vertices it is adjacent to. This causes redundancy in an undirected graph: for example, if vertices A and B are adjacent, A's adjacency list contains B, while B's list contains A. Adjacency queries are faster, at the cost of extra storage space.

2. Adjacency matrix

   This is the *n* by *n* matrix *A*, where *n* is the number of vertices in the graph. If there is an edge from some vertex *x* to some vertex *y*, then the element $a_{x,y}$ is 1 (or in general the number of *xy* edges), otherwise it is 0. In computing, this matrix makes it easy to find subgraphs, and to reverse a directed graph.

Graph Traversal Techniques:

To traverse a graph is to process every node in the graph exactly once. Because there are many paths leading from one node to another, the hardest part about traversing a graph is making sure that you do not process some node twice.

1. **Depth-First Search:**

   DFS follows the following rules:

   i. Select an unvisited node s, visit it, and treat as the current node

   ii. Find an unvisited neighbor of the current node, visit it, and make it the new current node;

   iii. If the current node has no unvisited neighbors, backtrack to the its parent, and make that the new current node;

   Repeat the above two steps until no more nodes can be visited.

   iv. If there are still unvisited nodes, repeat from step 1.

2. Breadth-First Search:

   BFS follows the following rules:

i. Select an unvisited node s, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.

ii. From each node x in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of x. The newly visited nodes from this level form a new level that becomes the next current level.

iii. Repeat the previous step until no more nodes can be vsisted.

iv. If there are still unvisited nodes, repeat from Step 1.

Conclusion : Thus we have implemented **DFS and BFS**

Exp 8

Aim : Implementation of quick sort

Theory :

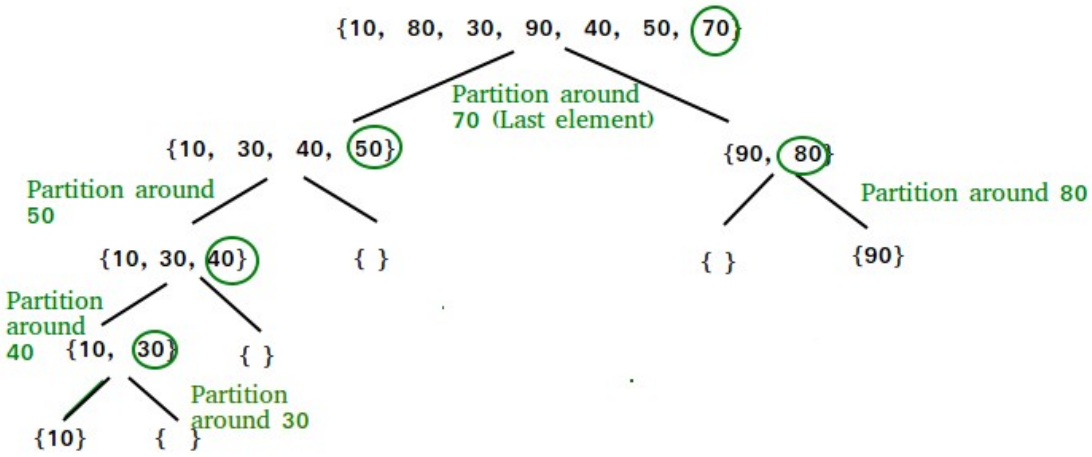
Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which are always sorted

Example:



Algorithm:

Algorithm QuickSort(p, q)
// Sorts the elements $a[p], \ldots, a[q]$ which reside in the global
// array $a[1:n]$ into ascending order; $a[n+1]$ is considered to
// be defined and must be $\geq$ all the elements in $a[1:n]$.
{
    if $(p < q)$ **then** // If there are more than one element
    {
        // divide P into two subproblems.
            $j :=$ Partition$(a, p, q+1)$;
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort$(p, j-1)$;
            QuickSort$(j+1, q)$;
        // There is no need for combining solutions.
    }
}

Algorithm partition(a, m, p)

```
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

Conclusion: Thus we have implemented Quick sort

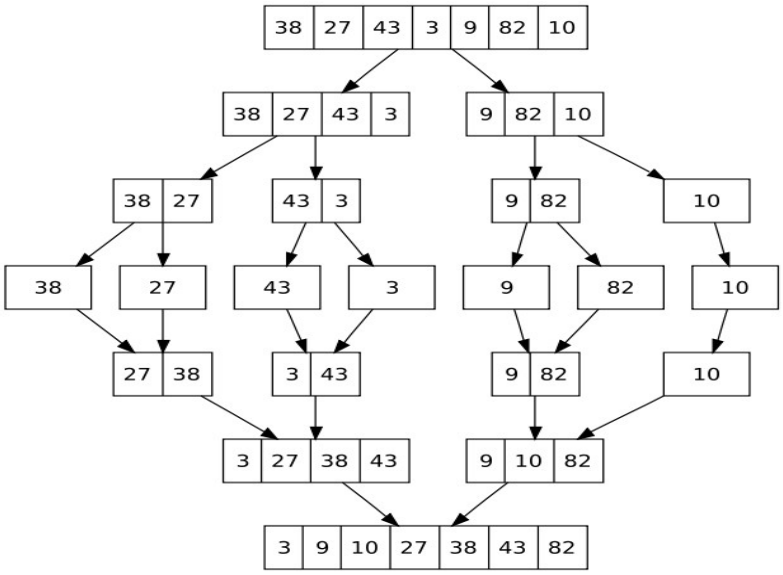
Exp: 9

Aim: Implementation of Merge sort

Theory :

A *sort* algorithm that splits the items to be sorted into two groups, *recursively* sorts each group, and *merges* them into a final, sorted sequence.
To sort an array of *n* elements, we perform the following three steps in sequence:

- If *n<2* then the array is already sorted. Stop now.
- Otherwise, *n>1*, and we perform the following three steps in sequence:
    1. Sort the left half of the the array.
    2. Sort the right half of the the array.
    3. Merge the now-sorted left and right halves.



Algorithm:

Algorithm MergeSort$(low, high)$
// $a[low : high]$ is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if $(low < high)$ **then** // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                $mid := \lfloor (low + high)/2 \rfloor$;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort$(mid + 1, high)$;
        // Combine the solutions.
            Merge$(low, mid, high)$;
    }
}

```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid + 1;
    while ((h ≤ mid) and (j ≤ high)) do
    {
        if (a[h] ≤ a[j]) then
        {
            b[i] := a[h];h := h + 1;
        }
        else
        {
            b[i] := a[j]; j := j + 1;
        }
        i := i + 1;
    }
    if (h > mid) then
        for k := j to high do
        {
            b[i] := a[k]; i := i + 1;
        }
    else
        for k := h to mid do
        {
            b[i] := a[k]; i := i + 1;
        }
    for k := low to high do a[k] := b[k];
}
```

conclusion : Thus we have implemented Merge sort

Exp: 10

Aim: Implementation of Linear and Binary searched

Theory:

Linear Search:

In computer science, **linear search** is a search algorithm, also known as **sequential search**, that is suitable for searching a list of data for a particular value.

It operates by checking every element of a list one at a time in sequence until a match is found. Linear search runs in O(n). If the data are distributed randomly, the expected number of comparisons that will be necessary is:

$$\begin{cases} N, & k = 0 \\ \frac{N+1}{K+1} & 1 <= k <= N \end{cases}$$

where *n* is the number of elements in the list and *k* is the number of times that the value being searched for appears in the list. The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed. The worst case is that the value is not in the list (or it appears only once at the end of the list), in which case *n* comparisons are needed

The simplicity of the linear search means that if just a few elements are to be searched it is less trouble than more complex methods that require preparation such as sorting the list to be searched or more complex data structures, especially when entries may be subject to frequent revision. Another possibility is when certain values are much more likely to be searched for than others and it can be arranged that such values will be amongst the first considered in the list.

Algorithm:

```
public int linearSearch(int a[], int valueToFind) {
  //a[] is an array of integers to search.
  //valueToFind is the number that will be found.
  //The function returns the position of the value if found.
  //The function returns -1 if valueToFind was not found.
  for (int i=0; i<a.length; i++) {
    if (valueToFind == a[i]) {
      return i;
    }
  }
  return -1;
}
```

Binary Search:

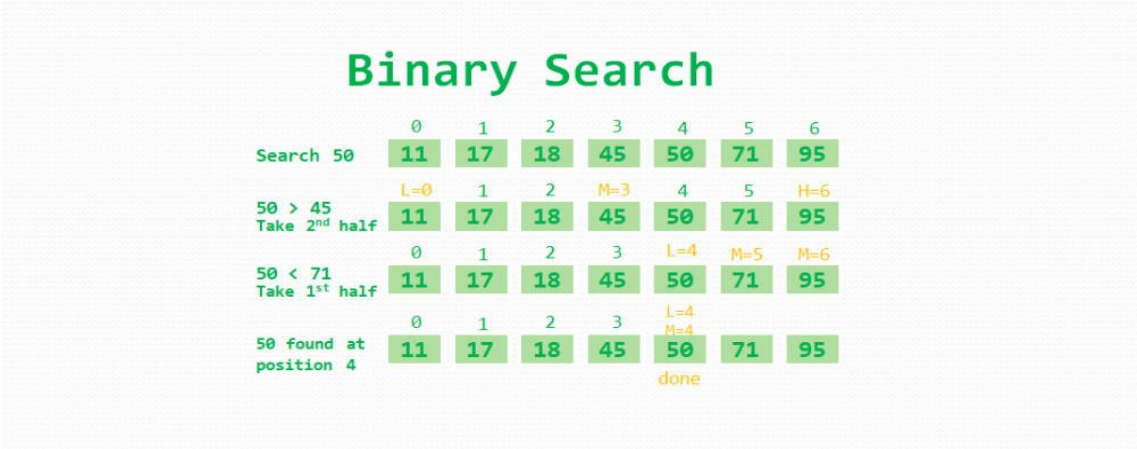
Search a *sorted array* by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search *key* is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

The *binary search* technique is a fundamental method for locating an element of a particular value within a sequence of sorted elements (see Sorting Problem). The idea is to eliminate half of the search space with each comparison.

First, the middle element of the sequence is compared to the value we are searching for. If this element matches the value we are searching for, we are done. If, however, the middle element is ``less than" the value we are chosen for (as specified by the relation used to specify a total order over the set of elements), then we know that, if the value exists in the sequence, it must exist

somewhere *after* the middle element. Therefore we can eliminate the first half of the sequence from our search and simply repeat the search in the exact same manner on the remaining half of the sequence. If, however, the value we are searching for comes before the middle element, then we repeat the search on the first half of the sequence.

Example:



Algorithm:

```
min := 1;
  max := N; {array size: var A : array [1..N] of integer}
  repeat
    mid := (min + max) div 2;
    if x > A[mid] then
      min := mid + 1
    else
      max := mid - 1;
  until (A[mid] = x) or (min > max);
```

Conclusion: thus we have Implemented linear and Binary search