Assignment No 1

Problem statement : To create ADT that implements the "set" concept

a) Add (new element) - place a value into a set

b) Remove (element) - remove the value

c) Contains (element) - return true if element is in a collection

d) Size () - return number of values in collection iterator() return an iterator used to loop over collection

e) Intersection of two sets.

f) Union of two set

g) Difference between two sets

h) Subset

Pre-requisite

Knowledge of Python programming

Knowledge of STL, set operations

Objective

To understand how Create, Display and perform various operations on set.

Input

Set A elements and Set B elements

Output

As per set operations

Description:

What is abstract data type?

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

Python Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation**.**

Set (): Creates a new set initialized to the empty set.

Length (): Returns the number of elements in the set, also known as the cardinality. Accessed using the len () function.

Contains (element): Determines if the given value is an element of the set and returns the appropriate Boolean value.

Add (element): Modifies the set by adding the given value or element to the set if the element is not already a member. If the element is not unique no action is taken and the operation is skipped.

Remove (element): Removes the given value from the set if the value is contained in the set and raises an exception otherwise.

IsSubsetOf (setB): Determines if the set is a subset of another set and returns a Boolean value. For set A to be a Boolean value. For set A to be a subset of B , all elements in A must also be elements in B.

Union (set B): Creates and returns a new set that is the union of this set and set B. The new set created from the union of two sets. A and B, contains all elements in A plus those elements in B that are not in A Neither set A nor set B is modified by this operation.

Intersect (setB): Creates and returns a new set that is the intersection of this set and setB . the intersection of sets A and B contains only those elements that are in both A and B. Neither set A nor set B is modified by this operation**.**

Difference (set B): Creates and returns a new set that is the difference of this and setB. The set difference, A-B, contains only those elements that are in A but not in B. Neither set A nor set B is modified by this operation**.**

Iterator (): Creates and returns an iterator that can be used to iterator over the collection of itesm.

Algorithm:

1. **To add elements in set:**
   i)Enter Element to add.
   ii)Check element with all elements form i=0 to n
   iii)if match found ask for another element.
   iv) if no match found till last position then store element at last position**.**
2. **To remove element from set**
   i)Ask element which is to be removed.
   ii)if it is found at any position then shift all elements towards left by one position**.**
   iii)If no match for element is found then display error message.
3. **To check for contained element**
   i) Enter Element to check.
   ii)Check element with all elements of set from position 0 to 1
   iii)if match found display message stating element is found
   iv) if no match found till last position then display not found message.
4. **Intersection of sets**
   i)pick first element of set 1 compare with all elements of set2
   ii)if match found at any position then put that element in intersection set.
   iii)If match not found till end then ignore element and select next element in set 1
   iv)Repeat steps i) , ii) and iii) for all elements of set 1
   v)print intersection set.
5. **Union of sets**
   i)Copy all elements of set 1 in union set
   ii)pick element of set2 compare with all elements of set1
   iii)if match not found at any position then put that element in union set at last position.
   iv)If match found at any position then ignore element and choose next element in set2
   v)Repeat steps ii) , iii) and iv) for all elements of set 2
   vi)print union set.
5. **Difference of sets**
   i)Take empty set to collect difference (set1-set2)
   ii)pick element of set1 compare with all elements of set2
   iii)if match not found at any position then put that element in difference set.
   iv)If match found at any position then ignore element and choose next element in set1
   v)Repeat steps ii) , iii) and iv) for all elements of set 1
   vi)print difference set.
6. **Subset(Compare if set 2 is subset of set1)**
   i)pick up element of set 2 compare with all elements of set1
   ii)if match found at any position then stop comparing and select next element in set 2
   iii)If match not found till end then print message that set2 is not subset of set1
   iv)Repeat steps i) , ii) and iii) for all elements of set 2

Conclusion : We have studied in depth, the concept of ADT, function of sets, the implementation of set functions and working of set function.

Assignment 2

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers. Write a Python program for the same.

Pre-requisite:

- Knowledge of Python programming
- Basic knowledge of hashing and collision handling techniques

Objective:

- To analyze advanced data structure like hash table.
- To understand collision handling techniques.
- To understand practical implementation of hash table.

Input:

Client record: Name and Telephone number

Outcome:

- Hash table with all clients information
- Print number of comparisons required to find a telephone number

Description:

What is Hashing?

- ❖ Hashing is finding an address where the data is to be stored as well as located using a key with the help of the algorithmic function
- ❖ Hashing is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the hash function
- ❖ A hash table is an array-based structure used to store

  <key, information> pairs
- ❖ The resulting address is used as the basis for storing and retrieving records and this address is called as home address of the record
- ❖ For array to store a record in a hash table, hash function is applied to the key of the record being stored, returning an index within the range of the hash table
- ❖ The item is then stored in the table of that index

position Functions need to be implemented

1. **create_record**
2. **display_record**
3. **delete_record**
4. **search_record**
5. **update_record**

Need to take Input data from user - Name**, Telephone number**

What is Hash Table?

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

Fig. Hash Table

- The above figure shows the **hash table with the size of n = 10**. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. **The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.**

Collision Resolution Techniques

a. Linear Probing

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- The simplest method is called Linear Probing. Formula to compute linear probing is:

- P = (1 + P) % (MOD) Table_size

b. Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs.

The result of the second hash function will be the number of positions form the point of collision to insert.

There are a couple of requirements for the second function:

1. it must never evaluate to 0

2. must make sure that all cells can be probed

A popular second hash function is:

hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.

So,

H(Key) = (hash1(key) + i * hash2(key)) % TABLE_SIZE And hash1(key) = key % TABLE_SIZE

Algorithm for linear probing:

Assumption:
*Hash table with size 10 is created and all positions are initialized with None.
*Hash function= key % table size

Algorithm to insert record:

1. Check if hash table is full-then record can't be inserted
2. Else
   Apply hash function on record number and find out index for a record.
3. If index position is empty then insert record at index position.
4. If index position is occupied then check for next empty position by incrementing pos circularly.
5. Insert record at newly found empty position.

Algorithm to search record:

1. Enter record to search
2. Apply hash function on record number and find out index position of record in hash table.
3. Check record name and number from index position with record to be searched and increment comparison count by 1.
4. If match found then search is successful.
5. If match not found then increment index position circularly and keep comparing records.
6. If count of comparison count = table size-1 then record not found

Algorithm to display records from table:

1. Initialize I to 0
2. Print record from position I .
3. Increment I
4. Repeat step 2,3 till i< table size

Algorithm for Double Hashing:
*Hash table with size 10 is created and all positions are initialized with None.
*Hash function 1 =h1 = key % table size
*Hash function 2 =h2= 5-(element % 5)

Algorithm for double hashing:

1. Get record to double hash and initialize i=1
2. For finding new position for record use formula

   Pos= (h1(key) + i * h2(key)) % table size
3. If pos is not empty then Increment i and repeat step 2 and 3
4. Else return position found

Algorithm to insert record:

1. Check if hash table is full-then record can't be inserted
2. Else

   Apply double hash function on record number and find out position for a record.
3. Insert record at position return by doublehash() function

Conclusion: By this way, we can perform the operations on hash table and use collision handling techniques if needed.

ASSIGNMENT NO. 3

Problem Statement: Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -

i.Insert new node, ii. Find number of nodes in longest path from root, iii. Minimum data value found in the tree, iv. Change a tree so that the roles of the left and right pointers are swapped at every node, v. Search a value.

Pre-requisite:

- Knowledge of C++ programming
- Basic knowledge of Binary Tree, Traversal of binary tree

Objective: To illustrate the binary search tree as ADT and implement various operations on it.

Outcome: Student will be able to create a binary search tree and perform various operations on it for solving various problems.

Theory:

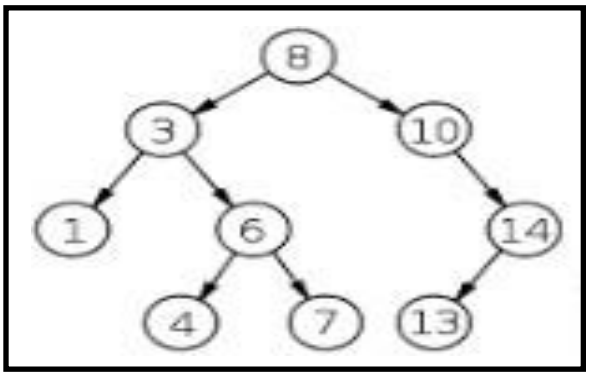
Binary Tree : A tree is said to be a Binary tree if each node of the tree have maximum of two child nodes. The children of the node of binary tree are ordered.

Binary Search Tree (BST) : A binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. This is called binary-search-tree property.

Operations on Binary search Tree :

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. In other words, we examine the root



and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

2. **Searching :** Searching a binary tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method. 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. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.
3. **Deletion :**There are three possible cases to consider:
    1. Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.
    2. Deleting a node with one child: Remove the node and replace it with its child.
    3. Deleting a node with two children: Call the node to be deleted N. Do not delete N. Instead, choose either its in-order successor node or its in-order predecessor node, R. Replace the value of N with the value of R, then delete R.

    As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

Algorithms for Operations on binary search tree:

// Create binary search tree

1. Create ( )

    Step 1: Allocate the memory by using new keyword for a new node. Make its left and right node Pointer NULL.

    Step 2: Accept the data from user and store it in the data part of new node.

    Step 3: Check whether the root is pointing to NULL , if it is then assign the value of new node Pointer to the root node pointer.

    Step 4: If the root node pointer is not NULL, then define a node pointer (temp) for traversing and store Root node address in it.

    Step 5: Compare the values of new node and the node pointed by temp.

    Step 6: If value in new node is greater than the value in temp, then perform temp=temp->right otherwise temp=temp->left

    Step 7: Repeat the step 5 and 6 until temp encounters the NULL value.

    Step 8: Link the new node to the parent node of temp properly according to its value.

// Traverse Binary search tree using Recursive Inorder Traversal

2. Inorder (node *temp)
   Step 1: Check whether temp==NULL
   Step 2: If not then call function inorder (temp->left).
   Step 3: Display the data and in the temp node.
   Step 4: Call the function inorder (temp->right).

//Traverse Binary search tree using Recursive Preorder Traversal

3. preorder (node *temp)
   Step 1: Check whether temp==NULL
   Step 2: If not then Display the data in the temp node.
   Step 3: Call function preorder (temp->left).
   Step 4: Call the function preorder (temp->right).

//Traverse Binary search tree using Recursive Postorder Traversal

4. postorder (node *temp)
   Step 1: Check whether temp==NULL
   Step 2: If not then Call function postorder(temp->left). Step 3: Call the function postorder (temp->right).
   Step 4: Display the data in the temp node.

// Search an element from binary search Tree

5. search ( )
   Step 1: Accept the data which is to be searched from the user.
   Step 2: Store the address of the root node in the current code pointer (cn).
   Step 3: Traverse the tree until the required data and the data in the cn matches.
   If the required data is greater than the word in the cn then perform perform cn=cn->right
   else
   Perform cn=cn->left
   If cn encounters the NULL value then come out of the loop statement by using break statemment.
   Step 4: Check whether the cn is NULL, if it is then required data is not found in the tree else the required data is present in the tree.

Time complexity:
Time complexity of the various functions of the binary tree are as follows:

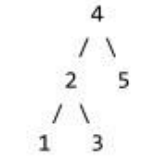
Sr.No.	Function	Time Complexity
1	create ( )	O(nlogn)
2	inorder ( )	O(n)
3	preorder ( )	O(n)
4	postorder ( )	O(n)
5	search ( )	O(log n)
6	display ( )	O(n)
7	delete_node ( )	O(log n)
8	Insert()	O(log n)

The overall time complexity of the program is O(n log n).

mirror()

Change a tree so that the roles of the left and right pointers are swapped at every node.

So the tree...



is changed to...

The solution is short, but very recursive. As it happens, this can be accomplished without changing the root node pointer, so the return-the-new-root construct is not necessary. Alternately, if you do not want to change the tree nodes, you may construct and return a new mirror tree based on the original tree.

Program: Write your own program and attach printouts

Output:

INPUT: integer data

OUTPUT:
Display result of each operation with error checking.

Conclusion:
Thus we studied the binary search tree and its operations and successfully implemented it for solving the given problem.

ASSIGNMENT NO.4

Problem Statement: A Dictionary stores keywords & its meaning. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.
Pre-requisite:

- Knowledge of C++ programming
- Basic knowledge of Binary Tree and Binary Search Tree.

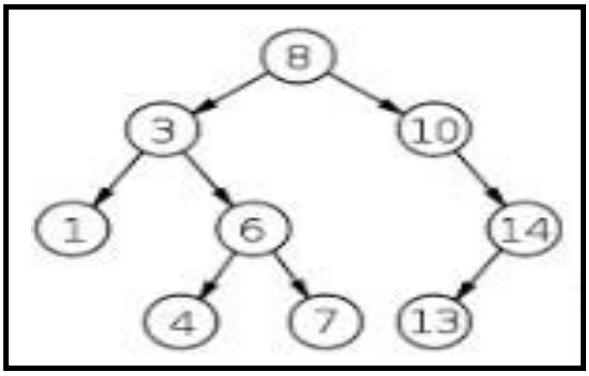
Theory:

Binary Search Tree (BST) : A binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. This is called binary-search-tree property.

Operations on Binary search Tree :

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. In other words, we examine the root



and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

2. **Searching :** Searching a binary tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method. 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. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.
3. **Deletion :**There are three possible cases to consider:
    1. Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.
    2. Deleting a node with one child: Remove the node and replace it with its child.
    3. Deleting a node with two children: Call the node to be deleted N. Do not delete N. Instead, choose either its in-order successor node or its in-order predecessor node, R. Replace the value of N with the value of R, then delete R.

    As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

Algorithms for creating keyword Dictionary using BST:

1. **Create ( )**

    Step 1: Allocate the memory by using new keyword for a new node(temp). Make its left andright node Pointer NULL.

    Step 2: Accept the keyword and it's meaning from user and store it in the Keyword and Meaning part of new node.

    Step 3: Check whether the root is pointing to NULL , if it is then assign the value of new node Pointer to the root node pointer.

    Step 4: If the root node pointer is not NULL, then call insert function to insert new node temp in the tree at its correct position.

    Step 5: stop

2. **Insert function()**

    Step 1: Compare the keyword field of temp and keyword field of trav node (which is at root initially) .Use strcmp function to compare the strings(keywords)

    Step 2: If temp-> Keyword is greater than trav ->Keyword, then perform

temp=temp->right otherwise temp=temp->left

Step 7: Repeat the step 1 and 2 until temp encounters the NULL value.

Step 8: Link the new node to the parent node of temp properly according to its value.

Algorithm to display entries in the dictionary in ascending order

3. **Inorder_display (node *root)//to display keywords in ascending order**

Step 1: Check whether root==NULL

Step 2: If not then call function inorder (root->left).

Step 3: Display Key word and it's meaning stored in root node.

Step 4: Call the function inorder (root->right).

Step 4: Stop

// Search an element from binary search Tree

4. **search ( )**

Step 1: Accept the Keyword which is to be searched from the user.

Step 2: Store the address of the root node in the temp node pointer

Step 3: Traverse the tree until the required keyword and the keyword in the temp matches.

If the required data is greater than the keyword in the temp then perform perform temp=temp->right

else

Perform temp=temp->left

If temp encounters the NULL value then come out of the loop statement by using break statemment.

Step 4: return temp

5. **Update entry ()**

Step 1: Accept the Keyword for which entry in Dictionary needs to be updated.

Step 2: call search function to search the keyword

Step 3: If the pointer temp which is returned by search function is NULL then required keyword is not present in tree.

If the required keyword is present then ask for new meaning of keyword and update it in the meaning field of the temp node.

Step 4: stop.

6. **Delete entry()**

Step 1: Accept the Keyword x for which entry in Dictionary needs to be deleted

Step 2: Check if root= NULL then delete function will return NULL

Step 3: keyword x and root->keyword is compared using strcmp function

- If x is less than root->keyword then call delete function recursively to delete keyword from left subtree
- If x is greater than root->keyword then call delete function recursively to delete keyword from right subtree
- If keyword and root->keyword is matching then
  -If check if root is having left child then left child will take place of root
  -If check if root is having right child then right child will take place of root
  -If root is having both child nodes then find inorder successor of root(temp) from right subtree of root
  Replace root->keyword with temp->keyword and delete temp from right subtree of root.

7. Max Comparisons()

For find out maximum comparisons to search any keyword from tree, we need to find out height of tree

Step 1: when root =NULL height of tree =0
Step 2: when root has no left or right child then height of tree = 0
Step3:when root has left and/or right subtree then return maximum from height of left subtree and height of right subtree as a height of tree.
Step 4: Total number of comparison to find any keyword from dictionary = height of tree +1

If the required keyword is present then ask for new meaning of keyword and update it in the meaning field of the temp node.
Step 4: stop.

Conclusion:

Thus we studied the binary search tree and its operations and successfully implemented it for solving the given problem.

Assignment No 5

Title:

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

Objectives:

1. To understand concept of tree data structure
2. To understand concept & features of object oriented programming.

Learning Objectives:

- ✓ To understand concept of class
- ✓ To understand concept & features of object oriented programming.
- ✓ To understand concept of tree data structure.

Learning Outcome:

- Define class for structures using Object Oriented features.
- Analyze tree data structure.

Software Required: g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Input: Book name & its number of sections and subsections along with name.

Output: Formation of tree structure for book and its sections.

Theory:

Definition:

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

• if T is not empty, T has a special tree called the root that has no parent

• each node v of T different than the root has a unique parent node w; each node with parent w is a child of w

Tree is a widely-used data structure that emulates a tree structure with a set of linked nodes. The tree graphicaly is represented most commonly as on *Picture 1*. The circles are the nodes and the edges are the links between them.

Trees are usualy used to store and represent data in some hierarhical order. The data are stored in the nodes, from which the tree is consisted of.The topmost node in a tree is called the root node.

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself.

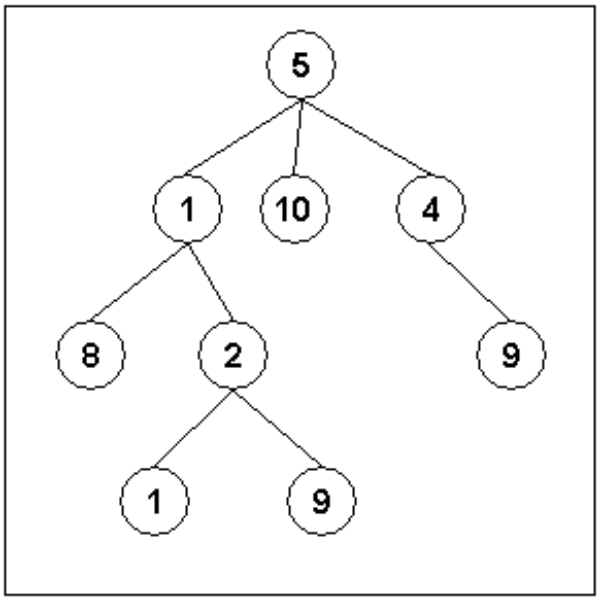
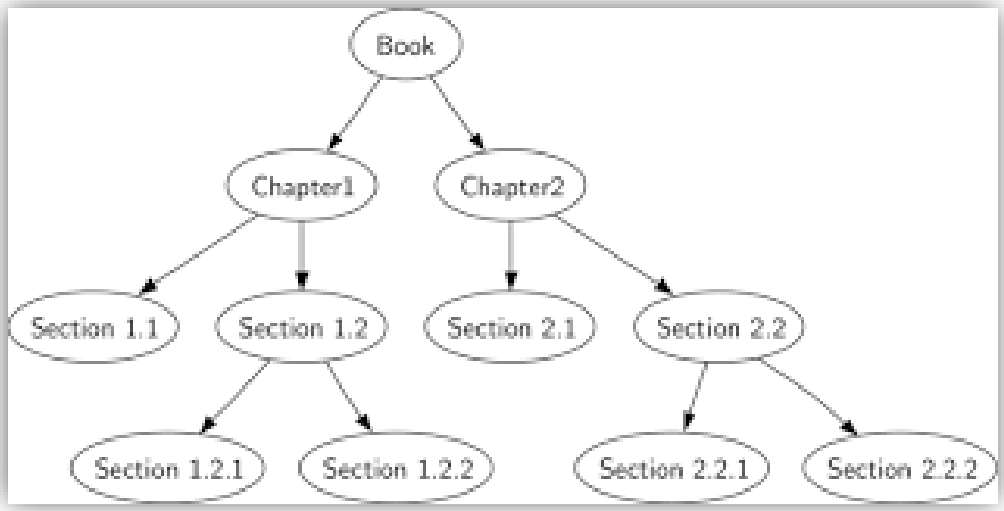


Fig1. An example of a tree

For this assignment we are considering the tree as follows.



Structure of every node in this tree is:

```
struct book_node
{
        char title[15];
```

```
    int chapt_count;
    struct book_node *down[10];

}
```

Algorithm:

Creating a Tree:

1)Allocate memory for new book node.
2)store following details in the node
  book name and no of chapters
3)for total no of chapters do
    i)Allocate memory for new chapter node and assign its address as downpointer of book node.
    ii)store chapter name and total no of sections in it.
    iii) for total no of sections do
        a) Allocate memory for new section node and assign its address as downpointer of chapter node.
        b) store section name and total no of subsections in it.
        c)for total number of subsections do
            -Allocate memory for new sub section node and assign its address as down pointer of section node.
            -Store details of subsections in the node.

Display Tree:

1)If root is not NULL
    i)Display Book Title
    ii)for total no of chapters do
        a)Display chapter name
        b) for total no of sections do
            - Display section name
            - for total number of subsections do
                -Store name of subsections in the node.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies

- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort

Conclusion: This program gives us the knowledge of tree data structure.

Assignment No. 6

Problem Definition:

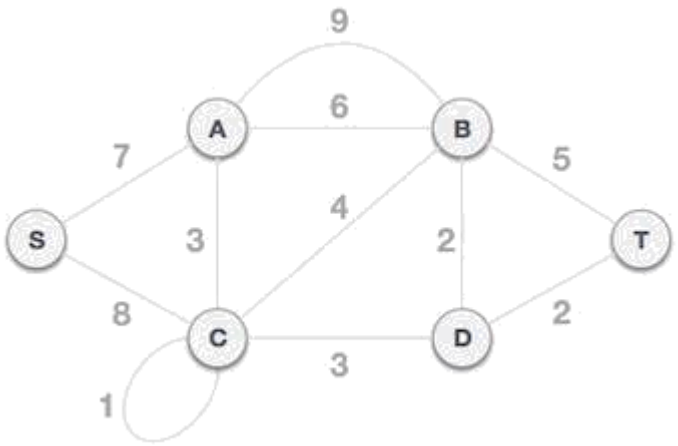
You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures

Theory:

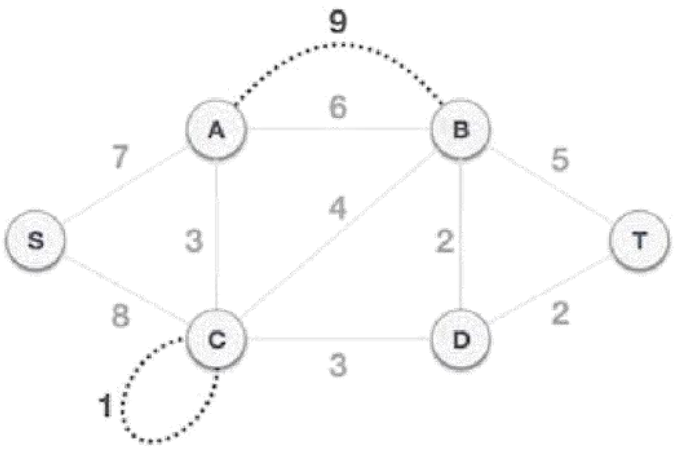
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

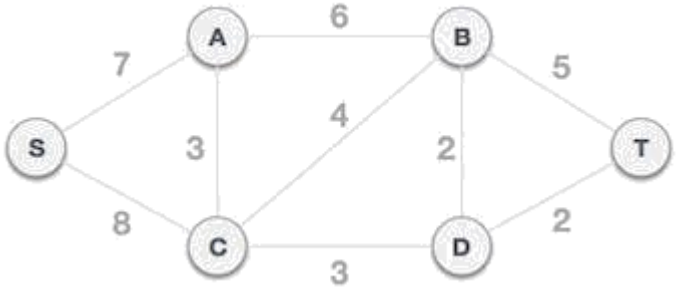
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

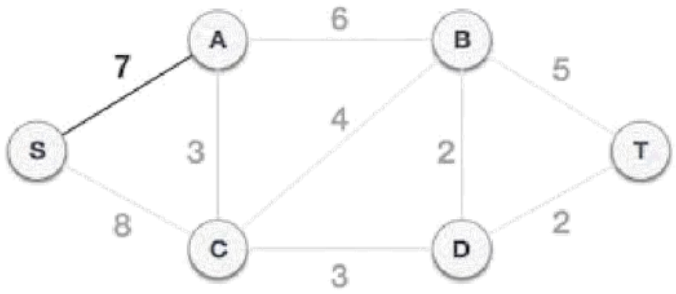


Step 2 - Choose any arbitrary node as root node

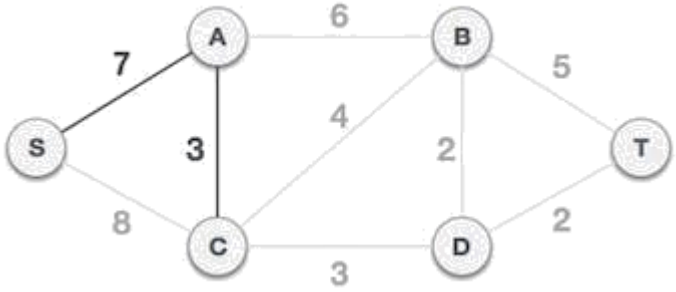
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

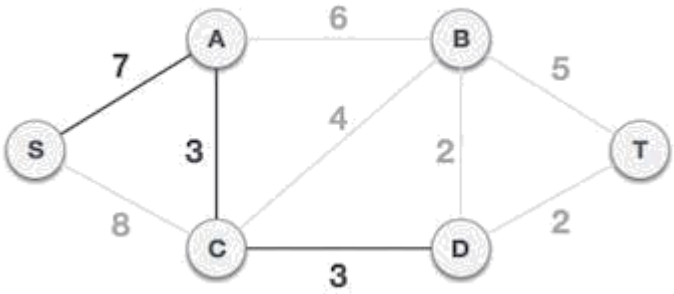
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.

1. Create a tree containing a single vertex, chosen arbitrarily from the graph

2. Create a set containing all the edges in the graph

3. Loop until every edge in the set connects two vertices in the tree
a. Remove from the set an edge with minimum weight that connects a vertex in the tree with a vertex not in the tree

b. Add that edge to the tree

ALGORITHM:

Create Function:

Step 1: Start.

Step 2: Create char, integer variable.

Step 3: Take how many vertices of graph and then take vertices ofedges and cost of
edges. Step 4: Store zero in visited array and matrix array for graph.

Step 5: Vertex is -99 then break loop, otherwise stored arrayg[v1][v2]=cost. If cost is stored then also g[v2][v1] alsostore same cost.

Step 6: Ask user whether he wants to enter more edges if 'Yes' thengoto Step 7.

Step 7: Stop.

Display Function:

Step 1: Start.

Step 2: Using two for loop print cost of edges of matrix array onscreen.

Step 3: Stop

Prims Function:

Step 1: Start.

Step 2: Initially declare cost, matrix, visited from distance array.

Step 3: Using two for loop check if edge is present or not. If notthen stored infinity value else stored cost of G matrix incost matrix and then stored zero in st matrix.

Step 4: Take first vertex and using for take all distances in from thatvertex in distance array and put

zero in
from and visited arrayat that vertex position.

Step 5: Initialize min cost =0 and no of edge=n-1.

Step 6: Using while loop take min distance infinity and again use forloop, check vertex is not visited and distance of that vertexis less min distance. If true then v = i and min distance=thatdistance.

Step 7: Stored that cost in st matrix for both vertices, edges anddecrease no of edge. Visited array is initialized = 1.

Step 8: Using for loop check vertex is not visited and distance isminimum. If true then stored that vertex in from array andCost of that edges in distance array.

Step 9: Add all min cost stored in min
cost and return min cost. Stop 10: Stop.

Main Function:

Step 1: Start.

Step 2: Create object of graph and Integer and character variables.

Step 3: Print Menu like

1. Create.

2. Display.

3. Spanning Tree and
find minimum cost.

Step 4: Take choice
from user.

Step 5: If choice is 1 then
call Create Function. Step
6: If choice is 2 then call
Display Function. Step 7:
If choice is 3 then call
prims Function.

Step 8: Ask user whether he
wants to continue or not. Step 9:
If yes then go to step 3.

Step 10: Stop.

Conclusion: Hence we studied how to use Prim's algorithm to implement network of telephone lines.

Assignment No 7

Title:

Represent a given graph using adjacency list /matrix to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent landmarks as nodes and perform DFS and BFS on that.

Objectives:

1. To understand concept of Graph traversal
2. To understand concept Breadth first search and Depth first search..

Theory:

Graphs are the most general data structure. They are also commonly used data structures.

A non-linear data structure consisting of nodes and links between nodes.

Representing Graphs using Adjacency Lists

Definition:

- A directed graph with *n* vertices can be represented by *n* different linked lists.
- List number *i* provides the connections for vertex *i*.
- For each entry *j* in list number *i*, there is an edge from *i* to *j*.

Breadth First Search:

BFS stands for Breadth First Search is a vertex-based technique for finding the shortest path in the graph. It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.
Example:

- **Input:**

- **Output:**

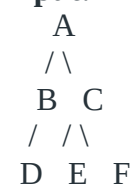
A, B, C, D, E, F

Depth First Search:

DFS stands for Depth First Search is an edge-based technique. It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if

there are no vertices then visited vertices are popped.
Example:

Input:



Output:
A, B, D, C, E, F

Algorithms:

BFS Algorithm:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The DFS algorithm

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Conclusion: This program gives us the knowledge about using stack and queue data structures for graph traversals.

Assignment No. 8

Problem Definition:

Given sequence k = k1 <k2 < ... < kn of n sorted keys, with a search probability pi for each key ki . Build the Binary search tree that has the least search cost given the access probability for each key.

Theory:

A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node

Search time of an element in a BST is ***O(n)***, whereas in a Balanced-BST search time is ***O(log n)***. Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

Dynamic method

Let us understand OBST problem using the dynamic approach.
We can consider the table mentioned below with the frequencies and the keys.

	1	2	3	4
Keys(k)	10	20	30	40
Frequencies(pi)	4	2	6	3

We will then go on to form a matrix with i and j,
Let us start by calculating the values for when j-i =0.
When j-i is 0,
i=0, j=0
i=1, j=1
i=2, j=2
i=3, j=3
i=4, j=4
Hence, c[0,0] = 0, and same for c[1,1], c[2,2], c[3,3], c[4,4].

Then, we calculate for when j-i=1,
When j-i=1,
i=0, j=1
i=1, j=2
i=2, j=3

i=3, j=4
Hence, c[0,1] = 4, c[1,2] = 2, c[2,3]= 6, c[3,4]=3.

Then, we calculate for when j-i=,
When j-i=2,
i=0, j=2
i=1, j=3
i=2, j=4
Hence, c[0,2] = 8.

Here, only two binary trees will be possible out of which cost of [0,2] is the smallest one, so we choose it.
Following the same method, we keep on testing for different values of j-i, and find the minimum cost in each case.

The general formula for finding the minimal cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j)

Pseudocode for OBST

Optimal-Binary-Search-Tree(p, q, n)

```
C[1…n + 1, 0…n],
w[1…n + 1, 0…n],
root[1…n + 1, 0…n]


for i = 1 to n + 1 do
  C[i, i - 1] := qi - 1
  w[i, i - 1] := qi - 1

for l = 1 to n do
  for i = 1 to n – l + 1 do
    j = i + l – 1 c[i, j] := ∞
    w[i, i] := w[i, i -1] + pj + qj

   for r = i to j do
       t := c[i, r - 1] + c[r + 1, j] + w[i, j]
       if t < c[i, j]
          c[i, j] := t
          root[i, j] := r
return c and root
```

Conclusion: OBST for given keys can be constructed using dynamic programming approch

Assignment No 9

Problem Statement:

Consider a scenario for Hospital to cater services to different kinds of patients as Serious (top priority), b) non-serious (medium priority), c) General Checkup (Least priority). Implement the priority queue to cater services to the patients.

THEORY:

What is Priority Queue ?

Priority Queue is an abstract data type that performs operations on data elements per their priority. To understand it better, first analyze the real-life scenario of a priority queue.

A priority Queue is a type of queue that follows the given below properties:

- An item with higher priority will be dequeued before the item with lower priority.
- If two elements present in the priority queue are having the same priority, then they will be served according to the order in which they are present in the queue.

The hospital emergency queue is an ideal real-life example of a priority queue. In this queue of patients, the patient with the most critical situation is the first in a queue, and the patient who doesn't need immediate medical attention will be the last. In this queue, the priority depends on the medical condition of the patients.

he priority queue in data structure resembles the properties of the hospital emergency queue. Thus, it is highly used in sorting algorithms. It behaves similar to a linear queue except for the fact that each element has some priority assigned to it. The priority of elements determines the order of removal in a queue, i.e., the element with higher priority will leave the queue first, whereas the element with the lowest priority at last.

ALGORITHM:

structure for Queue--
Store name of patients in name array
Store priority of patients in Pr array.

Insert Function/Enqueue :

1. Check if queue is full? Display message "Queue is full
2. Else
3. If queue is empty then insert patient name at first position in an array store it's priority in Pr array at first position
4. If queue is not empty then compare priority of new patient with priorities of previous patients in queue .Find correct position 'p' to insert queue.
5. Shift all records from position 'p' to r towards right by one position. Shift respective

priorities from Pr array.

6. Insert new record at position ‘p’.
7. Increment rear.

Delete Patient details from Queue after patient get treatment:

1.Check if queue is empty display message-“ queue is empty”
2. IF queue is not empty then display name of the patient present at front position as Deleted patient.
3.Increment front
Step 2: Front = Front->Next;
Step 3: return Temp

Display Queue:

1. For i= front to rear position
   i) display name[i]
   ii) Check priority and display status of patients (Serious ,non serious or general checkup) according to priority in Pr array.
2. Stop

Conclusion: Hence we studied how to use priority queue for implementing hospital scenario.

Assignment No .10

Problem statement:

Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data

Theory:

File structure–

A file is a collection of records which are related to each other. The size of file is limited by the size of memory and storage medium.

Two characteristics determine how the file is organised:

I. File Activity:

It specifies that percent of actual records proceeds in single run. If a small percent of record is accessed at any given time, the file should be organized on disk for the direct access in contrast.

If a fare percentage of records affected regularly than storing the file on tape would be more efficient & less costly.

II. File Volatility:

It addresses the properties of record changes. File records with many changes are highly volatile means the disk design will be more efficient than tape.

File organisation –

A file is organised to ensure that records are available for processing. It should be designed with the activity and volatility information and the nature of storage media, Other consideration are cost of file media, enquiry, requirements of users and file's privacy, integrity, security and confidentiality. There are four methods for organising files-

1. Sequential organisation
2. Indexed Sequential organisation
3. Inverted list organisation

4. Direct access organisation
5. Chaining

Sequential organization:

Sequential organization means storing and sorting in physical, contiguous blocks within files on tape or disk. Records are also in sequence within each block. To access a record previous records within

the block are scanned. In a sequential organization, records can be added only at the end of the file. It is not possible to insert a record in the middle of the file without rewriting the file.

In a sequential file update, transaction records are in the same sequence as in the master file. Records from both the files are matched, one record at a time, resulting in an updated master file. In a

personal computer with two disk drives, the master file is loaded on a diskette into drive A, while the transaction file is loaded on another diskette into drive B. Updating the master file transfers data from drive B to A controlled by the software in memory.

Advantages:

1.Simple to
design.

2.Easy to
program

iii.Variable length and blocked records
available

iv.Best use of storage space

Disadvantages

i.Records cannot be added at the middle of the file.

Algorithm:

Insert Record

1. Allocate Memory for new record S1
2. Get details for new record S1.
3. Open target file in append mode.
4. Write Record in target file at the end.
5. Close file.

Display Records

1.Open Existing File in read mode(ios::in)
2.Print headings for record tabs.
3.Read one record at a time.
4.Repeat step 3 till end of the file
5.Close the file.

Search Record:

1.Open file in read mode.
2.Enter target roll number to search.
3.Read one record form file.

If roll no matches with target roll no then
Print record details as record found and stop searching.

4.Repeat step 3 till last record or till record not found.
5.Print message if no such record present.

Delete Record

1.Open Existing file in read mode
2.Create new empty file temp.txt in write mode.
3.Enter roll number to be deleted.
4.Read record from original file.
If roll number of read record doesn't matches with target roll number
-Write it in temp.txt
If roll number matches then ignore that record.
5.Repeat step 4 till end of file.
6.Close both files.
7.Remove original file.
8.Rename temp.txt with original file name.

Conclusion:

Thus, we studied sequential file organization which is easy to implement where record are stored sequentially as they entered insertion in middle is not possible.

Assignment No 11

Problem Statement:

Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data

Theory:

What is the Indexed sequential access method (ISAM)?

ISAM (Indexed sequential access method) is an advanced sequential file organization method. In this case, records are stored in the file with the help of the primary key. For each primary key, an index value is created and mapped to the record. This index contains the address of the record in the file.

If a record has to be obtained based on its index value, the data block's address is retrieved, and the record is retrieved from memory.

Pros of ISAM

- Because each record consists of the address of its data block in this manner, finding a record in a large database is rapid and simple.
- Range retrieval and partial record retrieval are both supported by this approach. We may obtain data for a specific range of values because the index is based on primary key values. Similarly, the partial value can be simply found, for example, in a student's name that begins with the letter 'JA'.

Cons of ISAM

- This approach necessitates additional disc space to hold the index value.
- When new records are added, these files must be reconstructed in order to keep the sequence.
- When a record is erased, the space it occupied must be freed up. Otherwise, the database's performance will suffer.

Algorithm for different operations of Index sequential file:

Algorithm for Creating Sequential file and index file

1. Open Sequential file (emp.DAT) and Index file (ind.DAT) in read mode
2. Input details of employee(Empid, Name and Salary) and store it in record variable.
3. Write record in emp.DAT
4. Copy employee_id and position of record from emp.DAT file into index file.
5. Repeat steps 2,3,4 for all records .
6. Close file pointers for emp.DAT and ind.DAT

7. Stop.

Algorithm for Displaying records from sequential File.

1. Open both files(EMP.DAT and IND.DAT) in read mode
2. Set file pointers for both files at the beginning.
3. Read record from index file.
4. Calculate offset of same record in sequential files .
5. Read record from sequential based on calculated offset.
6. Print record on screen.
7. Repeat steps3,4,5,6 for all records .

Algorithm for deleteing record from file(Logical Deletion)

1. Ask for employee id of a record to be deleted.
2. Get employee id in variable id
3. Open both files (IND.DAT and EMP.DAT) for read and write purpose.
4. Set file pointers at the beginning.
5. Read record .
6. If employee if of record is not -1 then check if employee id of the record =id to be deleted.
7. If id matches then set employee name = " " and employee id and Employee salary=-1 to indicate logical deletion of a record.
8. Close file pointers
9. Stop.

Algorithm to search record

1. Ask for employee id of a record to be searched.
2. Get employee id in variable id
3. Open both files (IND.DAT and EMP.DAT) for read purpose.
4. Set file pointers at the beginning.
5. Read record .
6. If employee if of record is not -1 then check if employee id of the record =id to be searched.
7. If id matches then Display details of employee from sequential file .
8. If record not found till end or employee id for a record is -1 then display message as "Record not found"

Conclusion:

Hence we studied how to Implement indexing for sequential file to decrease search time for a record.

Assignment No 12

Problem Statement:

Implement the Heap sort algorithm in Java demonstrating heap data structure with modularity of programming language

Theory:

What is a heap?

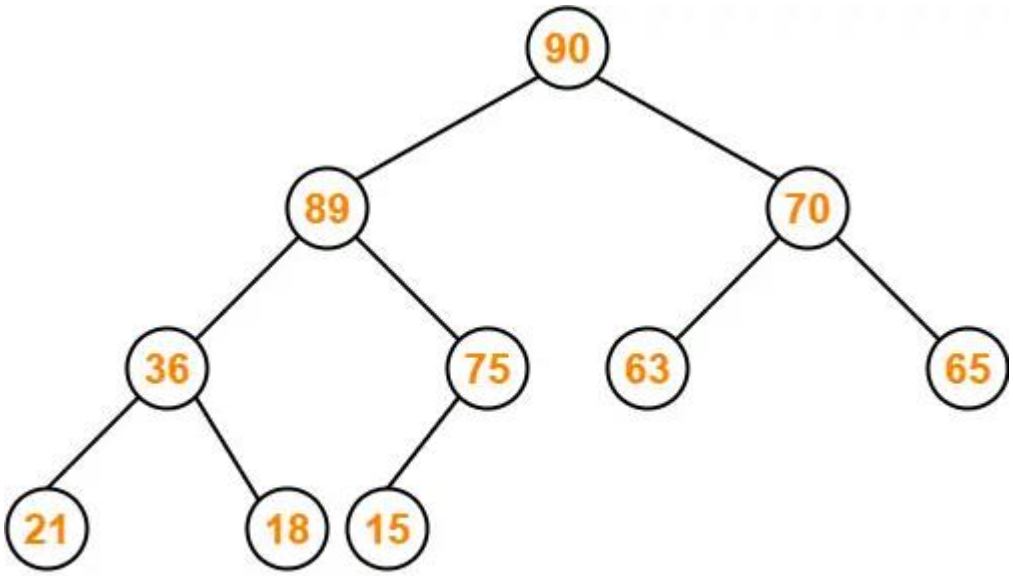
A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Heaps are mainly of two types — max heap and min heap:

- In a max heap, the value of a node is always >= the value of each of its children.
- In a min heap, the value of a parent is always <= the value of each of its children.

Example of max heap-

The following heap is an example of a max heap-



Max Heap Example

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

- Step 1 - Construct a Binary Tree with given list of Elements.
- Step 2 - Transform the Binary Tree into Max Heap.
- Step 3 - Delete the root element from Max Heap using Heapify method.
- Step 4 - Put the deleted element into the Sorted list.
- Step 5 - Repeat the same until Max Heap becomes empty.
- Step 6 - Display the sorted list

Pseudo code for heap sort

```
Heapsort(A) {
  BuildHeap(A)
  for i ← length(A) downto 2
 {
    exchange A[1]  and A[i]
    heapsize ←heapsize -1
    Heapify(A, 1)
}
BuildHeap(A) {
  heapsize ← length(A)
  for i ←floor( length/2 ) downto 1
    Heapify(A, i)
}
Heapify(A, i) {
  le ← left(i)
  ri ← right(i)
  if (le<=heapsize) and (A[le]>A[i])
    largest ← le
  else
```

```
    largest ← i
  if (ri<=heapsize) and (A[ri]>A[largest])
    largest ←ri
  if (largest != i) {
    exchange A[i] and A[largest]
    Heapify(A, largest)
  }
}
```

Time Complexity:

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

Conclusion: The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**