Dr. D. Y. Patil Pratishthan's

DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT & RESEARCH

Approved by A.I.C.T.E, New Delhi , Maharashtra State Government, Affiliated to Savitribai Phule Pune University Sector No. 29, PCNTDA , Nigidi Pradhikaran, Akurdi, Pune 411044. Phone: 020–27654470, Fax: 020-27656566 Website :www.dypiemr.ac.in Email : principal.dypiemr@gmail.com

Department of Artificial Intelligence and Data Science

LAB MANUAL Data Structures and Algorithms Laboratory

(Subject code:217532)

(SE 2019 pattern)

Semester II

Prepared by:
Mrs. Varsha Pandagre
Mrs. Deepali Hajare

Data Structures and Algorithms Laboratory

Course Code	Course Name	Teaching Scheme (Hrs./ Week)	Credits
217532	Data Structures and Algorithms Laboratory	4	2

Course Objectives:

- To **understand** practical implementation and usage of non linear datastructures for solving problems of different domain.
- To strengthen the ability to identify and **apply** the suitable data structure forthe given real world problems.
- To **analyze** advanced data structures including hash table, dictionary, trees, graphs, sorting algorithms and file organization.

Course Outcomes:

On completion of the course, learner will be able to–

CO1: Understand the ADT/libraries, hash tables and dictionary to design algorithms for a specific problem.

CO2: Choose most appropriate data structures and **apply** algorithms for graphical solutions of the problems.

CO3: Apply and **analyze** non linear data structures to solve real world complex problems.

CO4: Apply and **analyze** algorithm design techniques for indexing, sorting, multi-way searching, fileorganization and compression.

CO5: Analyze the efficiency of most appropriate data structure for creating efficient solutions forengineering design situations.

LIST OF ASSIGNMENTS

ASSIGNMENT NO.-01

Title: Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number. Make useof two collision handling techniques and compare them usingnumber of comparisons required to find a set of telephone numbers

Objectives:

1. To understand concept of Hashing
2. To understand to find record quickly using hash function.
3. To understand concept & features of object oriented programming.

Learning Objectives

- ✓ To understand concept of hashing.
- ✓ To understand operations like insert and search record in the database.

Learning Outcome

- ✓ Learn object oriented Programming features
- ✓ Understand & implement concept of hash table **.**

Theory:

Hash tables are an efficient implementation of a keyed array data structure, a structure sometimes known as an associative array or map. If you're working in C++, you can take advantage of the STL map container for keyed arrays implemented using binary trees, but this article will give you some of the theory behind how a hash table works.

Keyed Arrays vs. Indexed Arrays

One of the biggest drawbacks to a language like C is that there are no keyed arrays. In a normal C array (also called an indexed array), the only way to access an element would be through its index number. To find element 50 of an array named "employees" you have to access it like this:

1employees[50];

In a keyed array, however, you would be able to associate each element with a "key," which can be anything from a name to a product model number. So, if you have a keyed array of employee records, you could access the record of employee "John Brown" like this:

1employees["Brown, John"];

One basic form of a keyed array is called the hash table. In a hash table, a key is used to find an element instead of an index number. Since the hash table has to be coded using an indexed array, there has to be some way of transforming a key to an index number. That way is called the hashing function.

Hashing Functions

A hashing function can be just about anything. How the hashing function is actually coded depends on the situation, but generally the hashing function should return a value based on a key and the size of the arraythe hashing table is built on. Also, one important thing that is sometimes overlooked is that a hashing function has to return the same value every time it is given the same key.

Let's say you wanted to organize a list of about 200 addresses by people's last names. A hash table would be ideal for this sort of thing, so that you can access the records with the people's last names as the keys.

First, we have to determine the size of the array we're using. Let's use a 260 element array so that there can be an average of about 10 element spaces per letter of the alphabet.>

Now, we have to make a hashing function. First, let's create a relationship between letters and numbers:A

--> 0

B --> 1

C --> 2

D --> 3

and so on until Z --> 25.

The easiest way to organize the hash table would be based on the first letter of the last name.

Since we have 260 elements, we can multiply the first letter of the last name by 10. So, when a key like "Smith" is given, the key would be transformed to the index 180 (S is the 19 letter of the alphabet, so S --> 18, and 18 * 10 = 180).

Since we use a simple function to generate an index number quickly, and we use the fact that the index number can be used to access an element directly, a hash table's access time is quite small. A linked list of keys and elements wouldn't be nearly as fast, since you would have to search through every single key-element pair.

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.

- **Insert** − inserts an element in a hash table.
- **delete**− Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
structDataItem
{
  int data;
  int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
inthashCode(int key){

  return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is notfound at the computed hash code.

Example

```
structDataItem *search(int key)
{
  //get the hash
  inthashIndex = hashCode(key);

  //move in array until an empty
  while(hashArray[hashIndex] != NULL) {

    if(hashArray[hashIndex]->key == key)
      returnhashArray[hashIndex];

    //go to next cell
    ++hashIndex;
    hashIndex %= SIZE;
  }
  return NULL;
```

}

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(intkey,int data)
{
  structDataItem *item = (structDataItem*) malloc(sizeof(structDataItem));
  item->data = data;
  item->key = key;
  //get the hash
  inthashIndex = hashCode(key);
  //move in array until an empty or deleted cell
  while(hashArray[hashIndex] != NULL &&hashArray[hashIndex]->key != -1)
    { //go to next cell

    ++hashIndex;
    //wrap around the table
    hashIndex %= SIZE;
  }
  hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
structDataItem* delete(structDataItem* item) {
  int key = item->key;
  //get the hash
  inthashIndex = hashCode(key);
  //move    in    array    until    an    empty
  while(hashArray[hashIndex] !=NULL) {
    if(hashArray[hashIndex]->key    ==    key)    {
      structDataItem* temp = hashArray[hashIndex];
      //assign a dummy item at deleted position
      hashArray[hashIndex]    =    dummyItem;
      return temp;
```

```
        }
        //go to next cell
        ++hashIndex;
        //wrap around the table
        hashIndex %= SIZE;
    }
    return NULL;
}
```

Conclusion: In this way we have implemented Hash table for quick lookup using C++.

Assignment Questions:

1.What is Hash Function?

2.what is Good Hash function?

3.How many ways are there to implement hash function?

4.What are the Collision Resolution Strategies?

5.What is the Hash Table Overflow?

ASSIGNMENT NO.-02

Title:

To create ADT that implement the "set" concept. a. Add (new Element) -Place a value into the set , b. Remove (element) Remove the value c. Contains (element) Return true if element is in 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 sets, g. Difference between two sets, h. Subset

Objectives:

1. To understand concept of set and set ADT.
2. To understand concept python language.

Learning Objectives:

- ✓ To understand concept of set.
- ✓ To understand concept ADT.
- ✓ To understand python language

Learning Outcome:

- Define classes to show the concept of cohesion and coupling.
- Analyze working of functions.

Theory:

SETS:

Sets are a type of abstract data type that allows you to store a list of non-repeated values. Their name derives from the mathematical concept of finite sets.

Unlike an array, sets are unordered and unindexed. You can think about sets as a room full of people you know. They can move around the room, changing order, without altering the set of people in that room. Plus, there are no duplicate people (unless you know someone who has cloned themselves). These are the two properties of a set: the data is unordered and it is not duplicated.

Sets have the most impact in mathematical set theory. These theories are used in many kinds of proofs, structures, and abstract algebra. Creating relations from different sets and codomains are also an important applications of sets.

In computer science, set theory is useful if you need to collect data and do not care about their multiplcity or their order. As we've seen on this page, hash tables and sets are very related. In databases, especially for relational databases, sets are very useful. There are many commands that finds unions, intersections, and differences of different tables and sets of data.

The set has four basic operations.

Function Name*	Provided Functionality

insert(i)	Adds *i* to the set
remove(i)	Removes *i* from the set
size()	Returns the size of the set
contains(i)	Returns whether or not the set contains *i*

Sometimes, operations are implemented that allow interactions between two sets

Function Name*	**Provided Functionality**
union(S, T)	Returns the union of set *S* and set *T*
intersection(S, T)	Returns the intersection of set *S* and set *T*
difference(S, T)	Returns the difference of set *S* and set *T*
subset(S, T)	Returns whether or not set *S* is a subset of set *T*

Sample Python Implementation Using a Dictionary

The only way to adequately implement a set in python is by using a dictionary, or a hash table. This is because the dictionary is the only primitive data structure whose elements are unordered. This require a fewmore lines of code, but it keeps the sets' principles intact.

Note: Python also has its own Set primitive, but we want to implement our own to show how it works

```
class Set:
    def__init__(self,  data=None):
        self.data = {}
        if data != None:
            if len(data) != len(set(data)):
                data = set(data)
            for   d   in   data:
                self.data[d] = d
    def insert(self, i):
        if  i  in  self.data.keys():
            return 'Already in set'
        self.data[i] = i
    def remove(self, i):
        if i not in self.data.keys():
            return 'Not in set'
        self.data.pop(i)
    def size(self):
        return len(self.data.keys())
    def contains(self, i):
        if i in self.data.keys():
            return True
        return False
    def union(self, otherSet):
        setData = list(set(self.data.keys()) | set(otherSet.data.keys()))
        unionSet = Set(setData)
        return unionSet
    def intersection(self, otherSet):
        setData = list(set(self.data.keys()) & set(otherSet.data.keys()))
```

```
29         intersectionSet = Set(setData)
30         return intersectionSet
31     def difference(self, otherSet):
32         setData = list(set(self.data.keys()) ^ set(otherSet.data.keys()))
33         differenceSet = Set(setData)
34         return    differenceSet
35     def subset(self, otherSet):
36         if set(self.data.keys()) < set(otherSet.data.keys()):
37             return True
38         return    False
39     def___repr__(self):
40       return '['+', '.join(str(x) for x in self.data.keys())+']'
```

This code is a little more complex than other data types like the associative array, so let's break it down.

On line 2, we instantiate our class using a python dictionary as our data structure. We use a dictionary to keep the set unordered (dictionaries are unordered because they are implemented using a hash table).

The functions `insert`, `remove`, `size`, and `contains` all simply search through the dictionary's list of keys to determine how to proceed with the operation.

The functions `union`, `intersection`, `difference`, and `subset` all utilize difference pythonic set operators.

For example, in the `union` function, we first turn both Set's data into a `set()`. Then we say we'll take data values in either set and put them in a new set. We do that using the OR operator. This looks like a `|`. Then,we change the `set()` into a `list()` because that is what our class takes as an argument. Then we create anew instance of our `Set`, and we return it. There are many other operators such as `&` and `<`.

Built-in Set Functionality in Python

Now let's look at the built-in set primitive in Python. This is *not* the same implementation that used above. This is data structure provided to you by Python itself. It is very similar to our implementation, but it is far more efficient and has many more functions.

Here are the basic functions:

```
>>> set1 = set(['Alex', 'Brittany', 'Joyce'])
>>> set1.add('John')
>>> set1
set(['John', 'Alex', 'Brittany', 'Joyce'])
>>> set1.remove('Alex')
>>> set1
set(['John', 'Brittany', 'Joyce'])
```

Here are functions that relate sets to one another:

```
>>> odds = set([1, 3, 5, 7, 9])
>>> evens = set([0, 2, 4, 6, 8])
>>> odds | evens                    #Performs a union
```

```
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> odds & evens                  #Performs an intersection
set([])
>>> evens - odds                  #Performs a difference
set([0, 8, 2, 4, 6])
```

There are many more operations that Python's set can use.

Conclusion: This program gives us the knowledge of SET ADT

Assignment Questions:

1. What is Sets? Explain its operations..
2. What is Set as ADT?

ASSIGNMENT NO.-03

Title:
A book consists of chapters, chapters consist of sections and sections consist of subsections. Constructa 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.

Theory:

Introduction to Tree:

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

Recursive definition
• T is either empty
• or consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r

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.

A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no childs is called a leaf, and that node is of course at the bottommost level of the tree. The height of a node is the length of the longest path to a leaf from that node. The height of the root is the height of the tree. In other words, the "height" of tree is the "number of levels" in the tree. Or more formaly, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent.

The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. (In the formal definition, a path from a root to a node, for each different node is always unique). In diagrams, it is typically drawn at the top.

In some trees, such as heaps, the root node has special properties.

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below his height, that are reachable from the node, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Every node in a tree can be seen as the root node of the subtree rooted at that node.

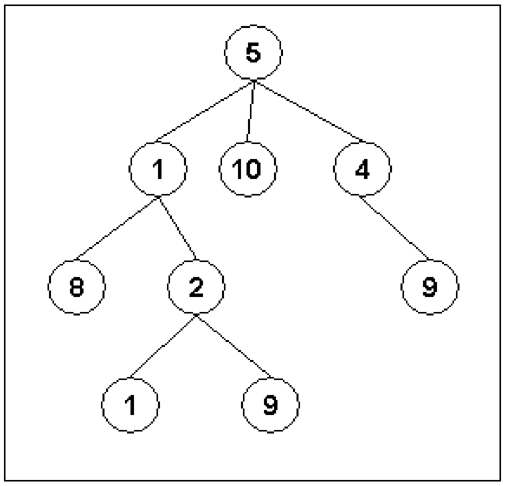


Fig. An example of a tree

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node. A tree on which an order is imposed — for example, by assigning different natural numbers to each child of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data structure. Binary search trees are one kind of ordered tree.

Important Terms

Following are the important terms with respect to tree.

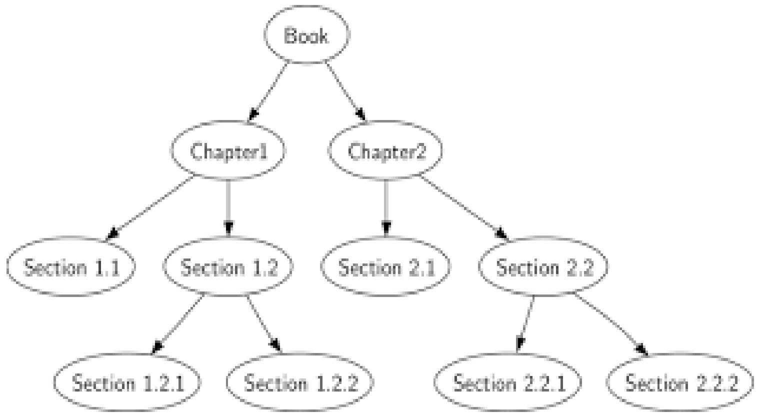
- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a 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

  For this assignment we are considering the tree as follows.



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.

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

Questions asked in university exam.

1. What is class, object and data structure?
2. What is tree data structure?
3. Explain different types of tree?

Conclusion: This program gives us the knowledge of Tree ADT

Assignment Questions:

1. What is tree?
2. What are different types of tree?
3. What is binary tree?
4. What is BST?
5. Explain Tree as an ADT.

ASSIGNMENT NO.-04

TITLE:
A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, & updating values of any entry. Also provide facility to display whole data sorted in ascending/ Descending order, Also find how many maximum comparisons may require for finding any keyword. Make use of appropriate data structures.

Objectives:
1. To understand concept of Tree data structure
2. To understand concept & features of object oriented programming.

Learning Objectives:
- ✓ To understand concept of Tree
- ✓ To understand concept of array, structure.

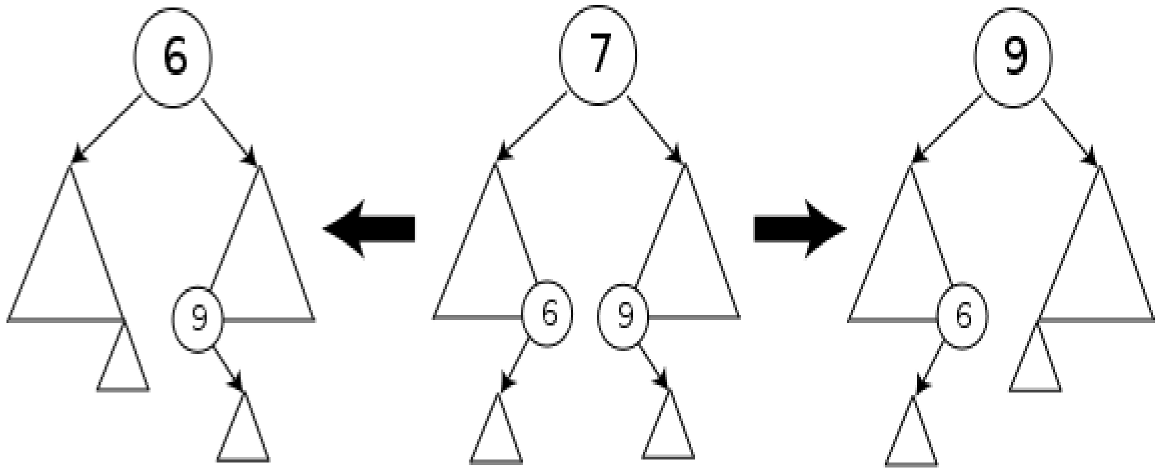
Learning Outcome:
- Implementation of sorting, searching on dictionary.Analyze
- working of functions

THEORY:
Binary search tree (BST), which may sometimes also be called an ordered or sorted binary tree, is a node-basedbinary treedata 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. 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 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. Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

ALGORITHMS:
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. Here's how a typical binary search tree insertion might be performed in C++:

```
/* Inserts the node pointed to by "newNode" into the subtree rooted at "treeNode" */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);
 }
```

Deletion: There are three possible cases to consider:
Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.
Deleting a node with one child: Remove the node and replace it with its child.
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 sub tree, and a node's in-order predecessor is the right-most child of its left sub tree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



Deleting a node with two children from a binary search tree. The triangles represent sub trees of arbitrary size, each with its leftmost and rightmost child nodes at the bottom two vertices. Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so good implementations add inconsistency to this selection.

STRCMP()
strcmp - compare two strings
Usage:
   include<string.h>
intstrcmp(const char *s1, const char *s2);
Description: The strcmp() function shall compare the string pointed to by s1 to the string pointedto by s2.

The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type **unsigned char**) that differ in the strings being compared.

RETURN VALUE
Upon completion, strcmp() shall return an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2, respectively

ALGORITHM:
Create Function:

```
Step 1: Start.
Step 2: Take data from user & create new node n1.
Step 3: Store data in n1 & make left & right child "NULL".
Step 4: If root is "NULL" then assign root to n1.
Step 5: Else call insert function & pass root & n1.
Step 6: Stop.
```

Insert Function:

```
Step 1: Start.
Step 2: We have new node in passed argument; so check data in temp
is less than data in root. If yes then check if Left child of root is
        "NULL" or not. If yes, store temp in left child otherwise call
insert function again & pass left address of node & temp.
Step 3: If data in root is less than data in temp then check if right child
of root is "NULL". If yes then store temp in right child field of
        root & if not call insert function again & pass right child address
of node & temp.
Step 4: Stop.
```

Display Function:

```
Step 1: Start.
Step 2: If root is "NULL" then display tree is not created.
Step 3: Otherwise call inorder function & pass root.
Step 4: Stop.
```

Inorder Function:

```
Step 1: Start.
Step 2: Taking root from inorder function, check if it is "NULL" or not.
Step 3: Again call inorder function & pass left child address of node Step
4: Display data of root.
Step 5: Again call inorder function & pass right child address of node.
Step 6: Stop.
```

Search Function:
Step 1: Start.
Step 2: If root is “NULL” then display tree is not created.
Step 3: Read the key to be searched in a variable, say k.
Step 4: Search for the key in the root, left and right side of the root.
Step 5: Display appropriate message for successful and unsuccessful search.
Step 6: Stop

Update Function:
Step 1: Start.
Step 2: If root is “NULL” then display tree is not created.
Step 3: Read the key to be Updated in a variable, say k.
Step 4: Search for the key in the root, left and right side of the root.
Step 5: If key is not present, display appropriate message.
Step 6: If key is present, update its meaning.
Step 7: Stop

Delete Function:
Step 1: Start.
Step 2: If root is “NULL” then display tree is not created.
Step 3: Read the key to be Deleted in a variable, say k.
Step 4: Search for the key in the root, left and right side of the root.
Step 5: If key is not present, display appropriate message.
Step 6: If key is present, Delete the key.
Step 7: Stop

Main Function:
Step 1: Start.
Step 2: Create necessary class objects & declare necessary variables.
Step 3: Print menu as below & take choice from user:
Create
Display
Search
Update
Delete
Exit
Step 4: If choice is 1, call create function.
Step 5: If choice is 2, call display function.
Step 6: If choice is 3, call Search function.
Step 7: If choice is 4, call Update function.
Step 8: If choice is 5, call Delete function.
Step 9: If choice is 6, then exit.
Step 10: Ask user whether he wants to continue or not.
Step 11: If yes then go to step 3.

Step 12: Stop.

Mathematical Modeling:
Let I be a set of Inputs {n, i_n } where n is no. of keywords of dictionary & i_n is an individual node values which stores keyword & its meaning.
Let O be a set of Outputs {C, Dk, N, D, S, U} where
C is Create Dictionary function = c(I) = L (c is function of create which gives output as a L setof keywords & its meaning of dictionary)
Dk is display keyword function = dk(L) = i_n (dk is display function on L which gives output as i_n an individual node values stores keyword & its meaning)
N is Insert New keyword function = n(L) = i_{n+1} (n is insert new keyword function on L which gives output as i_{n+1} an individual node values (stores keyword & its meaning) + 1)
D is Delete keyword function = d(L) = i_{n-1} (d is delete keyword function on L which gives output as i_{n-1} an individual node values (stores keyword & its meaning) - 1)
S is Search keyword Function = s(L) = a (s is search function on L which gives output as a where a € A = {Keyword Found, Keyword Not Found})
U is Update keyword Function = u(L) = a (u is update function on L which gives output as a i_n an individual node updates meaning of keyword.

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

INPUT: Enter data (numbers) to be stored in the binary search tree. Every node in the BST would contain 3 fields: data, left child pointer and right child pointer.

OUTPUT: Element should be inserted/deleted at the proper places in the BST.

Conclusion: This program gives us the knowledge of Tree and operations performed on Tree.

Questions asked in university exam:

1) Explain strcmp() with its return values?
2) What is Binary Search Tree? Explain with an example?
3) Explain various operations on BST?
4) What is inorder traversal of BST?

ASSIGNMENT NO.-05

TITLE:
Convert given binary tree into threaded binary tree. Analyze time and space complexity of the algorithm.

OBJECTIVES:

1. To write simple program in C++.

2. To understand concept & features C++ programming.

3. To implement algorithms in C++.

Learning Objectives

- ✓ To understand concept of TBT algorithm.
- ✓ To implement concept of TBT algorithm.

Learning Outcome
After successfully completing this assignment, students will be able to
- ✓ Learn C++ Programming features
- ✓ Understand & implement TBT algorithm using C++**.**

RELEVANT THEORY:
THREADED BINARY SEARCH TREE:

Threaded Binary Search Tree

A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal.
It is also possible to discover the parent of a node from a threaded binary tree, without explicit useof parent pointers or a stack

Tree Traversals of Threaded Binary Tree:
Inorder Traversals: A B C D E F F H I

Preorder Traversals: F B A D C E G I H
Postorder Traversals: A C E D B H I G F

ALGORITHM:

Create Function:

Step 1: Start.
Step 2: Take data from user & create new node n1.
Step 3: Store data in n1 & make left & right child "NULL".
Step 4: If root is "NULL" then root is equal to n1. Create a dummy or head node. Head of left thread=1, right thread = 0,right = NULL, left field will store address of root; assign root of left field points to dummy & right field to dummy.
Step 5: Else call insert function & pass root & n1.
Step 6: Stop.

Insert Function:

Step 1: Start.
Step 2: We have new node in passed argument; so check data in temp is less than data in root. If yes then check if Left thread of root is or not. If yes, store temp in left child, store left thread equal to 1, temp of left field equal to root of left field. Temp of right field equal to root. otherwise call insert function again & pass left address of node & temp.
Step 3: If data in root is less than data in temp then check if right thread of root is 1 or not. If yes, store temp in right child, store right thread equal to 1,temp of right field equal to root of right field. Temp of left field equal to root. otherwise call insert function again & pass right address of node & temp.
Step 4: Stop.

Preorder Function:

Step 1: Start.
Step 2: If root is "NULL" then display tree is not created.
Step 3: Otherwise store root in temp.
Step 4: Display data of temp.
Step 5: if left thread of temp is equal to 1 then go to left of temp. repeat same step 4.
Step 6: if right thread is equal to 0 then then go to left of temp.
if temp is equal to dummy then go to step 9.otherwise repeat step 6.
Step 7: go to right of temp.
Step 8: if temp is equal to dummy then go to Step 9.
otherwise repeat step 4.
Step 9: Stop.

Postorder Function:

Step 1: Start.

```
Step 2: If root is "NULL" then display tree is not created.
Step 3: Otherwise store root in temp.
Step 4: if left thread of temp is equal to 1 then go to left of temp. repeat same step 4.
Step 5: if right thread is equal to 0 then then go to left of temp.
if temp is equal to dummy then go to step 9.otherwise repeat step 6.
Step 6: go to right of temp.
Step 7: Display data of temp.
Step 8: if temp is equal to dummy then go to Step 9.
        otherwise repeat step 4.
Step 9: Stop.
```

Inorder Function:

```
Step 1: Start.
Step 2: If root is "NULL" then display tree is not created.
Step 3: Otherwise store root in temp.
Step 4: if left thread of temp is equal to 1 then go to left of temp. repeat same step 4.
Step 5: Display data of temp.
Step 6: if right thread is equal to 0 then then go to left of temp.
if temp is equal to dummy then go to step 9. Display data of temp. Repeat step 6.Step
7: go to right of temp.
Step 8: if temp is equal to dummy then go to Step 9.
otherwise repeat step 4.
Step 9: Stop.
```

Main Function:

```
Step 1: Start.
Step 2: Declare necessary variables.
Step 3: Print menu as follow & take choice from user:
        1. Create threaded binary search tree
        2. Display preorder
        3. Display inorder
        4. Display postorder
        5. Exit
Step 4: If choice is 1, call create function.
Step 5: If choice is 2, call preorder function.
Step 6: If choice is 3, call inorder function.
Step 7: If choice is 4, call postorder function.
Step 8: If choice is 4, exit from function.
Step 9: Stop.
```

Mathematical Modelling:

Let I be a set of Inputs {n, i_n } where n is number of nodes & i_n is an individual node values which stores integer data.

Let O be a set of Outputs {C, Din, Dpre, Dpost } where
C is Create function = c(I) = L (c is function of create which gives output as a L set of nodes) Din is display inorder function = in(L) = i_n (in is display function on L which gives output asinorder traversal of i_n an individual node values)
Dpre is display preorder function = pre(L) = i_n (pre is display function on L which gives output as preorder traversal of i_n an individual node values)
Dpost is display postorder function = post(L) = i_n (in is display function on L which gives outputas postorder traversal of i_n an individual node values)

SOFTWARE REQUIRED: CPP compiler- / Fedora PC

Input:
Give integer numbers as data. And Read user choice.

Output:
Conversion of binary tree into threaded binary tree.

Conclusion: This program gives us the knowledge of Binary Tree and operations performed on Tree

<u>Questions asked in university exam:</u>

1) What is Threaded Binary Tree?
2) What is difference in BT & TBT?
3) What is Advantages of TBT Tree?
4) What is use of thread in TBT?

ASSIGNMENT NO.-06

Title:

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Justify the storage representation used.

Objectives:

1. To understand concept of Graph data structure
2. To understand concept of representation of graph.

Learning Objectives:

- ✓ To understand concept of Graph data structure
- ✓ To understand concept of representation of graph.

Learning Outcome:

- Define class for graph using Object Oriented features.
- Analyze working of functions.

Theory:

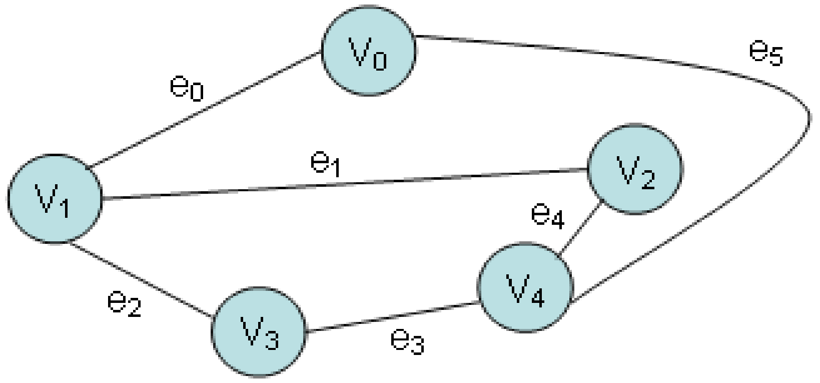
Graphs are the most general data structure. They are also commonly used data structures.

Graph definitions:

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

Undirected graph definition:

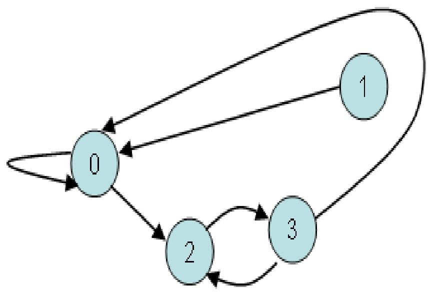
- An undirected graph is a set of nodes and a set of links between the nodes.
- Each node is called a **vertex**, each link is called an **edge**, and each edge connects two vertices.
- The order of the two connected vertices is unimportant.
- An undirected graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph.



Graph Implementation:

Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed.

Representing Graphs with an Adjacency Matrix



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & \text{true} & \text{false} & \text{true} & \text{false} \\ 1 & \text{true} & \text{false} & \text{false} & \text{false} \\ 2 & \text{false} & \text{false} & \text{false} & \text{true} \\ 3 & \text{true} & \text{false} & \text{true} & \text{false} \end{array}$$

Fig: Graph and adjacency matrix

Definition:

- An adjacency matrix is a square grid of true/false values that represent the edges of a graph.
- If the graph contains n vertices, then the grid contains n rows and n columns.
- For two vertex numbers i and j, the component at row i and column j is true if there is an edge from vertex i to vertex j; otherwise, the component is false.

We can use a two-dimensional array to store an adjacency matrix:

boolean[][] adjacent = new boolean[4][4];

Once the adjacency matrix has been set, an application can examine locations of the matrix to determine which edges are present and which are missing.

Representing Graphs with Edge Lists

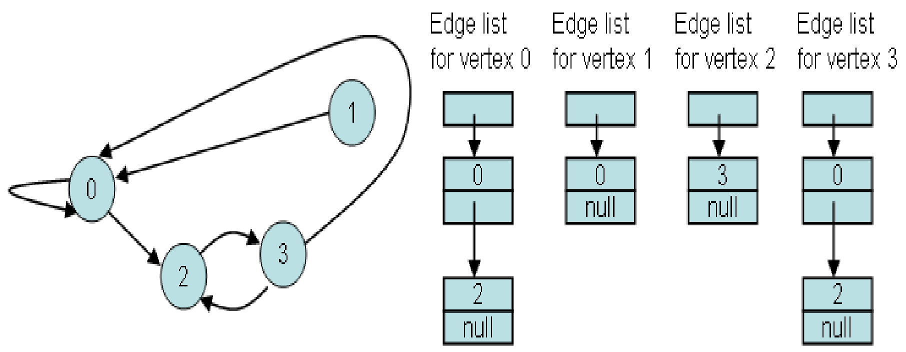


Fig: Graph and adjacency list for each node

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.

Loops and multiple edges could be allowed.

Representing Graphs with Edge Sets

To represent a graph with n vertices, we can declare an array of n sets of integers. For example:

IntSet[] connections = new IntSet[10]; // 10 vertices

A set such as connections[i] contains the vertex numbers of all the vertices to which vertex i is connected.

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

Input: 1.Number of cities.
2.Time required to travel from one city to another.

Output: Create Adjacency matrix to represent path between various cities.

Conclusion: This program gives us the knowledge of graph representation and operations on graph.

Questions asked in university exam.

1. What are different ways to represent the graph? Give suitable example.
2. What is time complexity of function to create adjacency matrix?

ASSIGNMENT NO.-07

TITLE:
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.

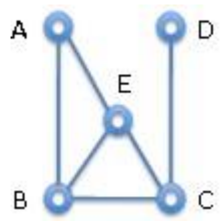
Objectives:
1. To understand concept of graph data structure
2. To understand concept & features of object oriented programming.

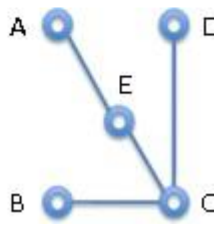
Learning Objectives:
- ✓ To understand concept of class
- ✓ To understand concept & features of object oriented programming.
- ✓ To understand concept graph data structures

Learning Outcome:
- ✓ learn the working of minimum spanning tree
- ✓ Analyze working of functions

THEORY:

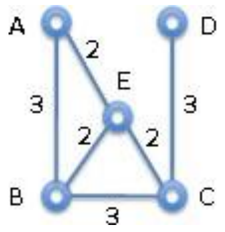
Spanning Tree:A spanning tree for a connected graph G is a tree T containing all the vertices of G.

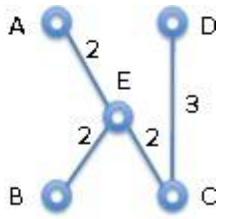




.
G **T**

Minimum Spanning Tree: A spanning tree whose weight of edges is minimum is called asMinimum Spanning Tree.



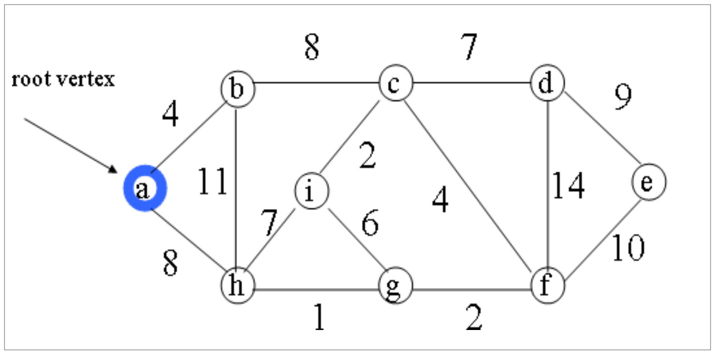


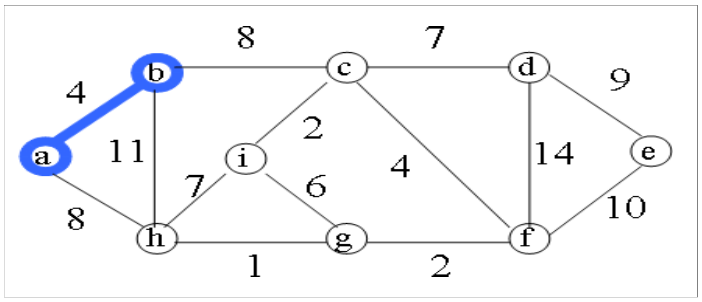
Two Methods to find Minimum Spanning Tree:

Prims Algorithm:

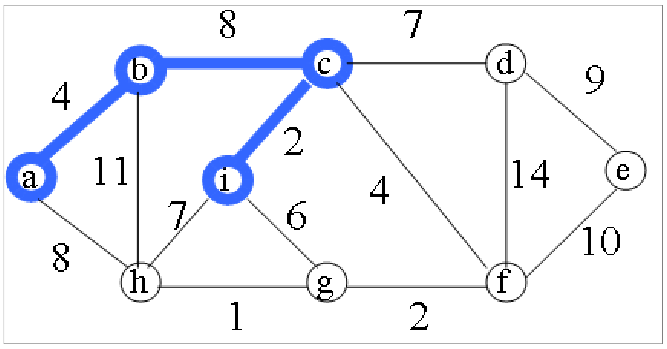
This algorithm is used to find minimum spanning tree and minimum cost of spanning tree.

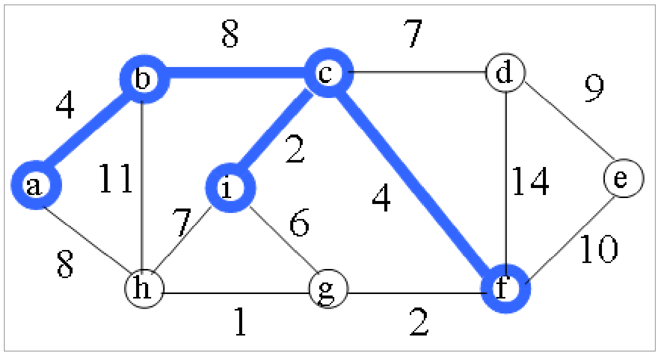
Steps:: 1) This algorithm starts with start vertex.

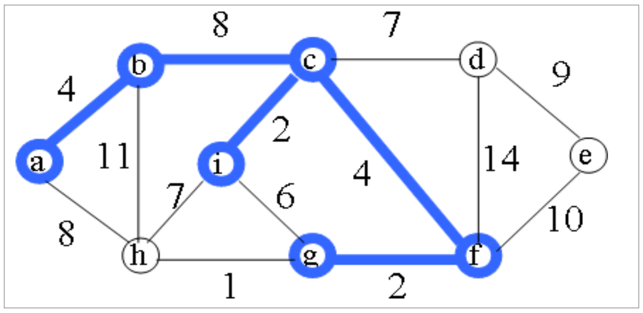


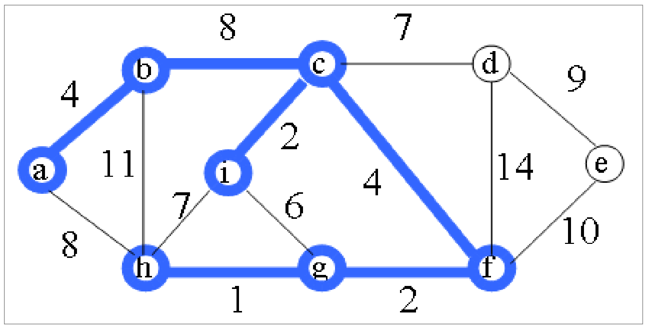


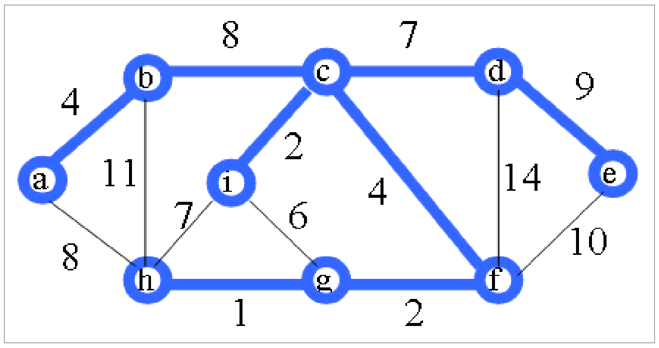
2) Then it finds edge which has minimum weight.
3) It continues the finding such that one vertex of edge is already visited & another vertex is not visited with minimum weight.


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2

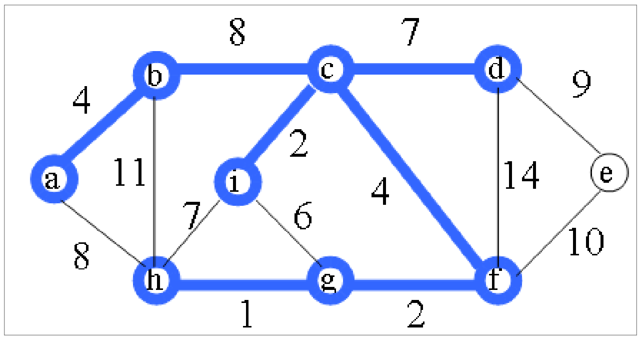

a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
14
4
7
6
10
8
1
2



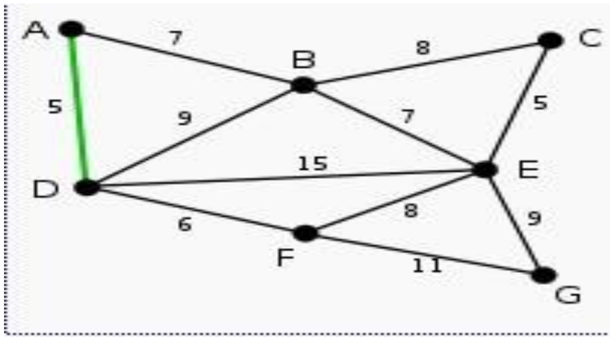
4) These steps are repeated to cover all the vertices of graph.

2. Kruskal’s Algorithm

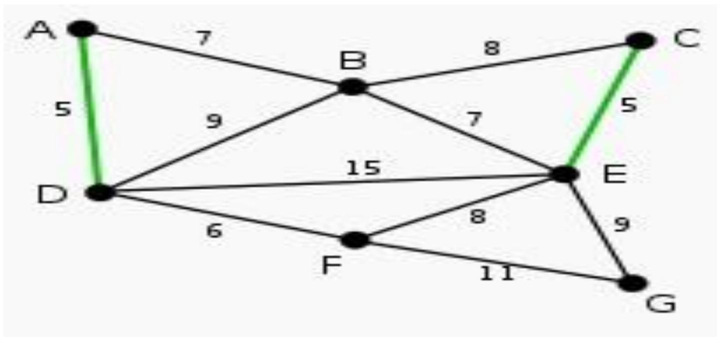


This algorithm is used to find minimum spanning tree and minimum cost of spanning tree.

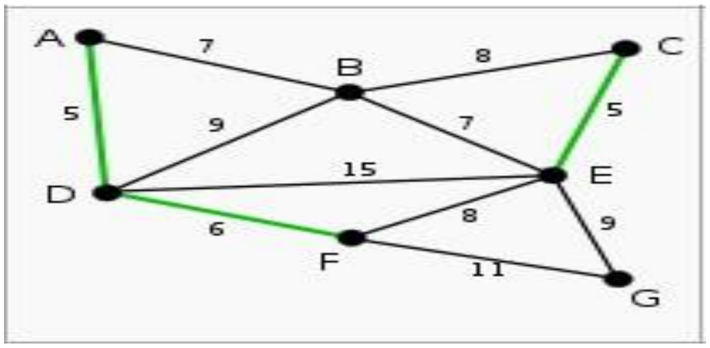
Steps:: 1) This algorithm starts with edge with minimum weight. AD and CE are the shortest edges, with length 5, and AD has been arbitrarily chosen, so it is highlighted.



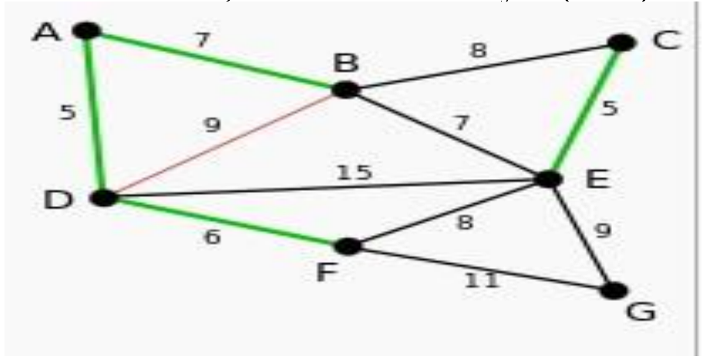
2) CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.



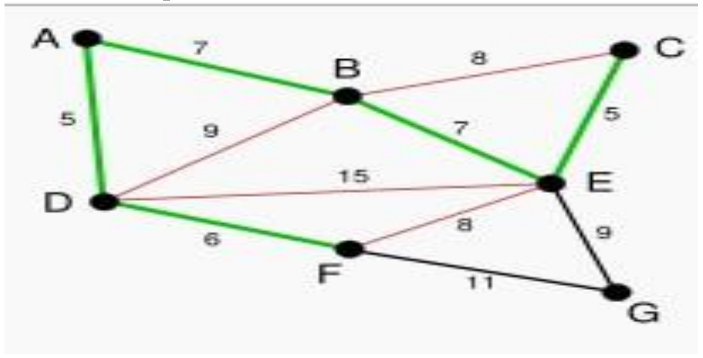
3) The next edge, DF with length 6, is highlighted using much the same method.



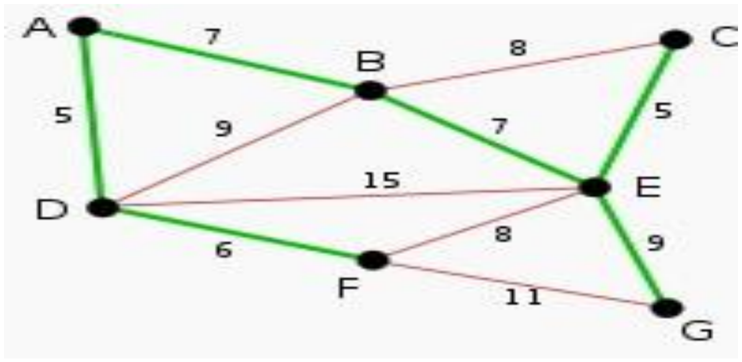
4) The next-shortest edges are AB and BE, both with length 7. AB is chosen arbitrarily, and is highlighted. The edge BD has been highlighted in red, because there already exists a path (in green) between B and D, so it would form a cycle (ABD) if it were chosen.



5) The process continues to highlight the next-smallest edge, BE with length 7. Many more edges are highlighted in red at this stage: BC because it would form the loop BCE, DE because it would form the loop DEBA, and FE because it would form FEBAD.



6) Finally, the process finishes with the edge **EG** of length 9, and the minimum spanning tree is found.



Application of Minimum Spanning Tree:
Minimum spanning trees are useful in constructing networks, by describing the way to connect aset of sites using the smallest total amount of wire.

SOFTWARE REQUIRED: C compiler- / Fedora PC

INPUT: number of vertices as offices and provide edge between vertices/offices as telephone lease lines(here leased line are nothing but edges and offices are nothing but vertices).

Output: The path which have minimum distance.

Conclusion: This program gives us the knowledge of spanning tree and different algorithms..

Questions asked in university exam:

1. What is Spanning Tree?
2. What is Minimum Spanning Tree?
3. What is Prim’s Algorithm?
4. What is Kruskal’s Algorithm?
5. What are Applications of Minimum Spanning Tree?
6. How to decide whether to select Kruskal’s algorithm or Prim’s algorithm?

Assignment No. 08

Problem Statement:

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.

Learning Objectives:

To learn concept of OBST. To

implement program OBST

Learning Outcome: Students implement the program for above problem.

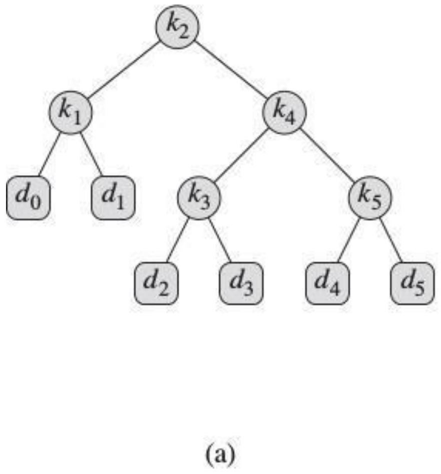
Theory and Algorithm:

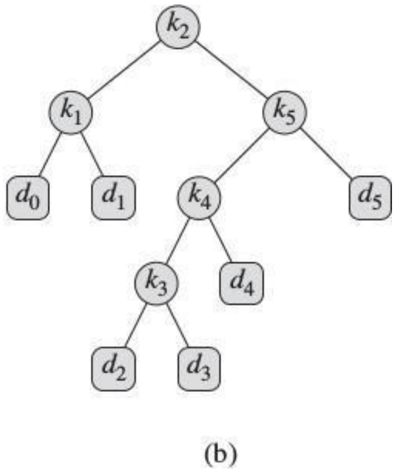
We are given an ordered sequence K={k1, k2, ..., kn}of distinct keys in sorted order so that ki < k+1. We wish to build a binary search tree from these keys For each key ki we have a probability pi that a search will be for ki. Some searches may be for values not in K. So we have "dummy" keys d0, d1, ..., dn representing values not inK. In the sorted order these would look like: d0 < k1 < d1 < k2 < ..., kn < dn (meaning that di represents values in between ki and ki+1. The probability qi associated with di is the probability that the search will ask for values in between ki and ki + 1. q0 is the probability that values less than k1 will be asked for and qn is the probability that values greater than kn will be requested.

Recall, that in a binary search tree if y is a node in the left sub-tree of x and z is a node in right sub-tree of x,then key [y] ≤ key [x] ≤ key[z].If the frequencies with which these keys occur is uniform, a balanced tree is the best to minimize the average search time. If, however, they are not uniform, then it may be advantageous to have tree that is not balanced in order to minimize the expected number of comparisons. For example, consider:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Two Binary search trees shown below for this example.





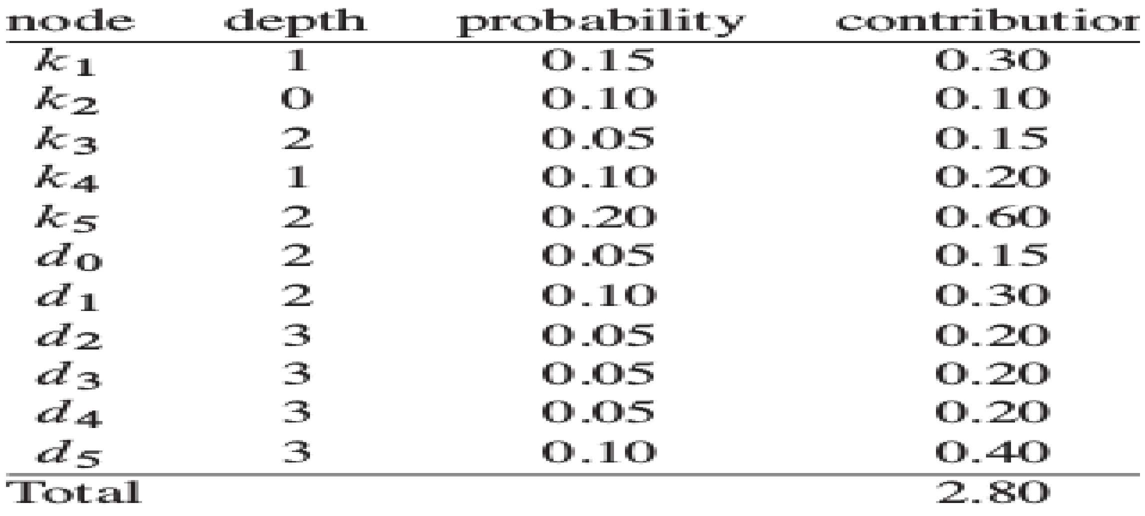
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 .$$

(15.10)

Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given binary search tree T . Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in T , plus 1. Then the expected cost of a search in T is

$$\begin{aligned} \text{E}[\text{search cost in } T] &= \sum_{i=1}^{n}(\text{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n}(\text{depth}_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n}\text{depth}_T(k_i)\cdot p_i + \sum_{i=0}^{n}\text{depth}_T(d_i)\cdot q_i , \qquad (15.11) \end{aligned}$$

where depth T denotes a node’s depth in the tree T . The last equality follows from equation (15.10). In Figure 15.9(a), we can calculate the expected search cost node by node:

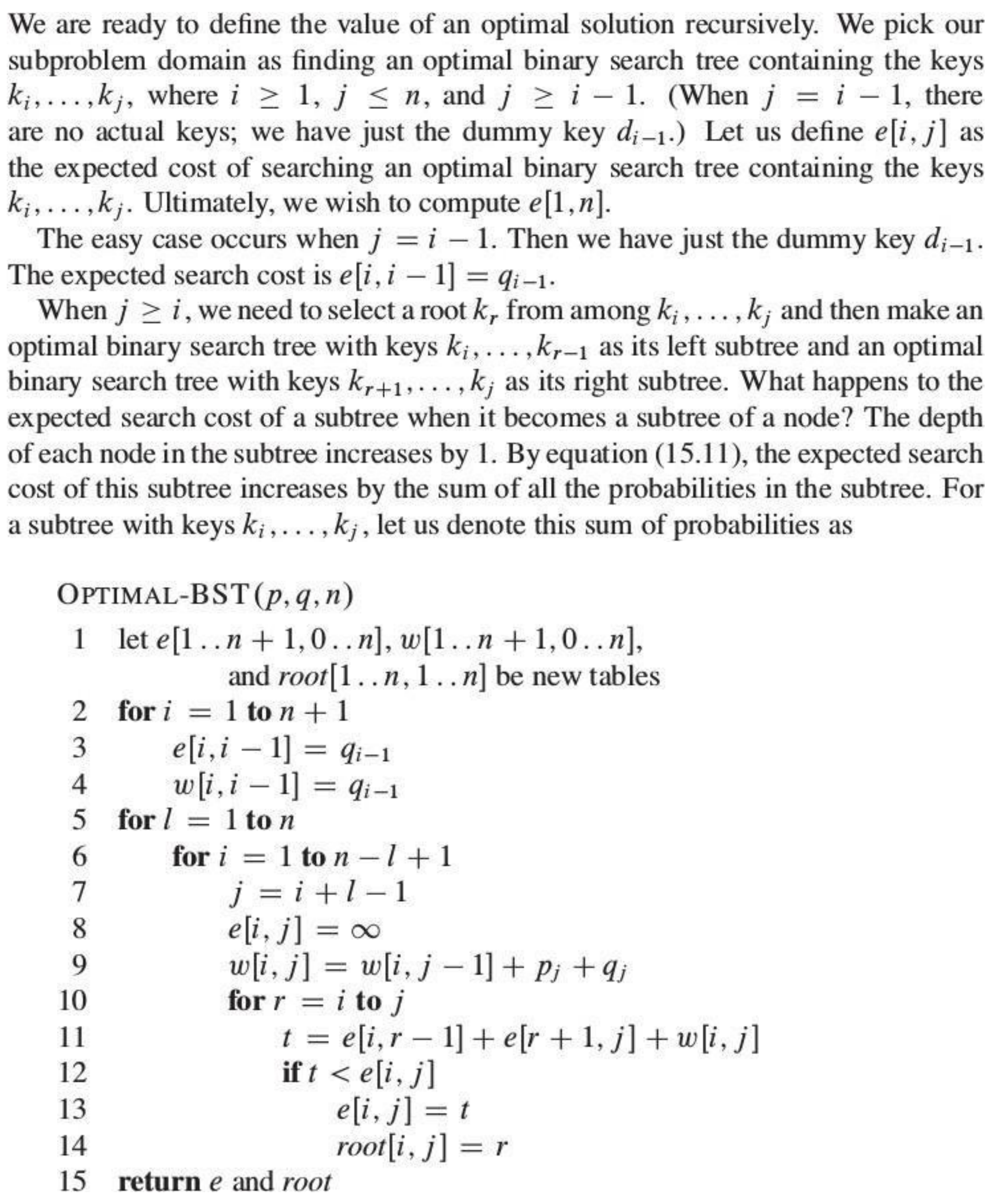


node	depth	probability	contributior
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such a tree an optimal binary search tree. Figure (b) shows an optimal binary search tree for the probabilities given in the figure caption; its expected cost is 2.75. This example shows that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor can we necessarily construct an optimal binary search tree by always putting the key with the greatest probability at the root. Here, key k 5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k 2 .

(The lowest expected cost of any binary search tree with k 5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. We can label the nodes of any n-node binary tree with the eys k 1 ; k 2 ; : : : ; k n to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4, we saw that the number of binary trees with n nodes is .4 ^{n} =n $^{3/2}$, and so we would have to examine an exponential number of binary search trees in an exhaustive search. Not surprisingly, we shall solve this problem with dynamic programming.



We are ready to define the value of an optimal solution recursively. We pick our subproblem domain as finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \geq 1$, $j \leq n$, and $j \geq i - 1$. (When $j = i - 1$, there are no actual keys; we have just the dummy key d_{i-1}.) Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Ultimately, we wish to compute $e[1, n]$.

The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.

When $j \geq i$, we need to select a root k_r from among $k_i, \ldots, k_j$ and then make an optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree and an optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (15.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys $k_i, \ldots, k_j$, let us denote this sum of probabilities as

```
OPTIMAL-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
          and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i−1] = q_{i−1}
    w[i, i−1] = q_{i−1}
for l = 1 to n
    for i = 1 to n − l + 1
        j = i + l − 1
        e[i, j] = ∞
        w[i, j] = w[i, j−1] + p_j + q_j
        for r = i to j
            t = e[i, r−1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

Software Required : g++ compiler.

Time Complexity : O(n^3) as we have to calculate c(i,j), w(i,j) and r(i,j).

C**onclusion:** This program gives us the knowledge of OBST.

Assignment No. 09

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 Height balance tree and find the complexity for finding a keyword

Objective:

To understand the concept and basic of Height balanced Binary Search tree or AVL tree in Data structure.

Outcome:

To implement the basic concept of AVL tree and to perform basic operation insert, search and delete an element in AVL tree also able to apply correct rotation when tree is unbalance after insertion and deletion operation in Data structure.

Software & Hardware Requirements:

2. 64-bit Open source Linux or its derivative

3. Open Source C++ Programming tool like G++/GCC

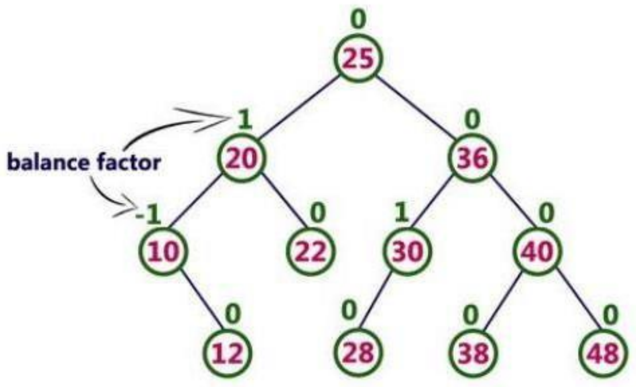
Theory Concepts:

AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.



Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

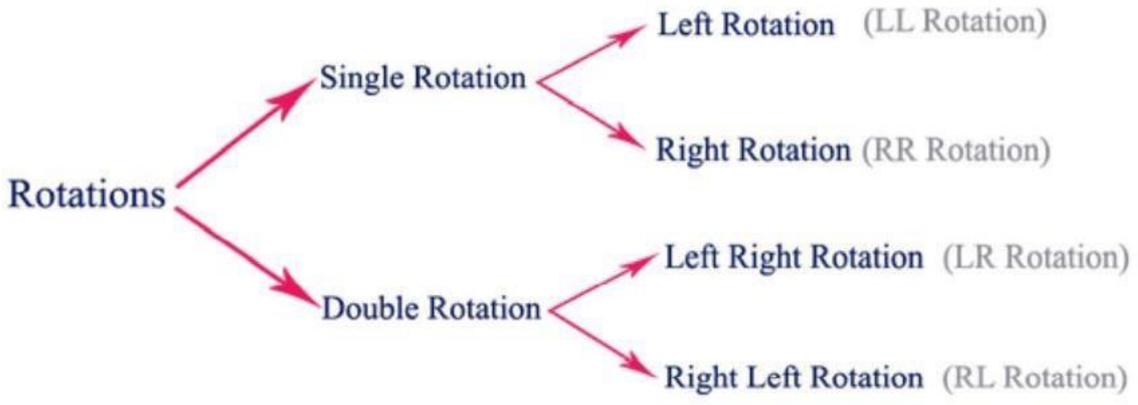
Every AVL Tree is a binary search tree but all the Binary Search Trees need not to beAVL trees.

AVL Tree Rotations

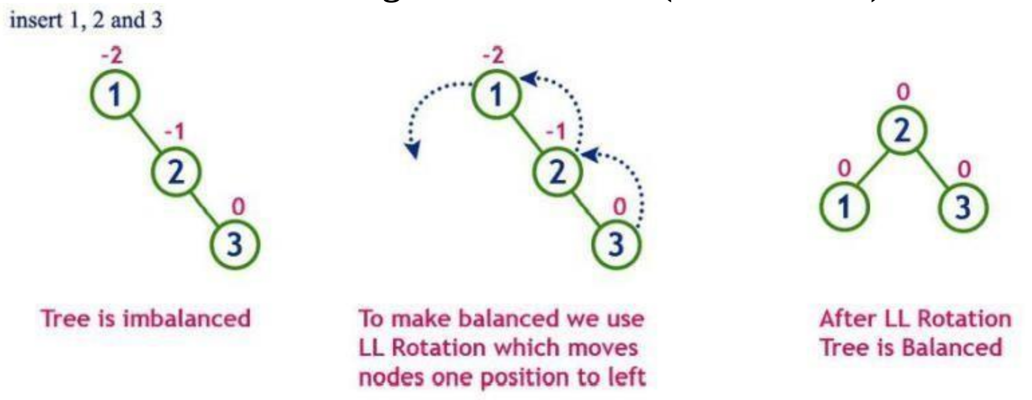
In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

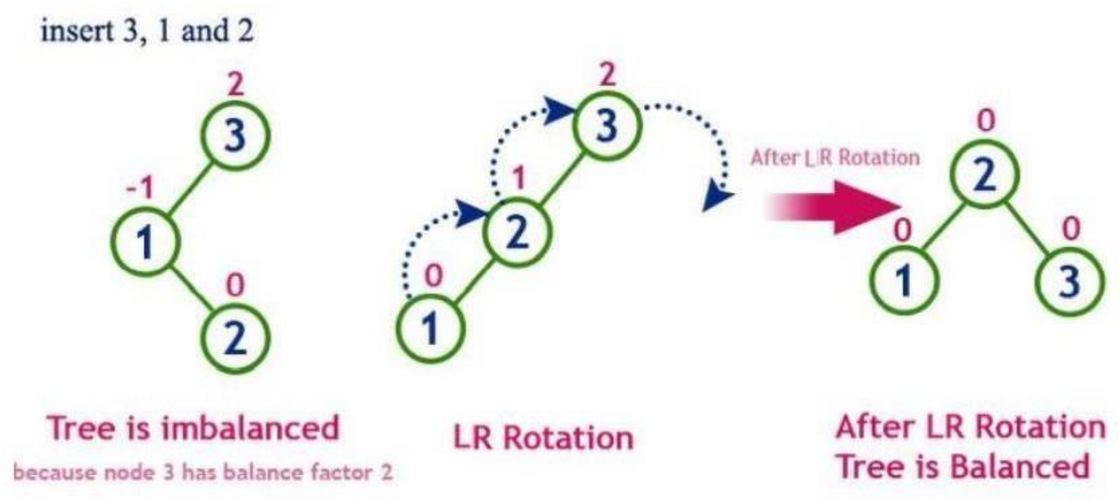
There are four rotations and they are classified into two types.


Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

Single Left Rotation (LL Rotation)


insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
0
2
0
1
0
3
After LL Rotation Tree is Balanced

Double Rotation (LR Rotation)



Similarly, RR and RL rotation can be performed

Algorithm:

The following operations are performed on an AVL tree...

1. Search

2. Insertion

3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps

to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than thatnode value.

Step 5: If search element is smaller, then continue the search process in left subtree. Step

6: If search element is larger, then continue the search process in right subtree. Step 7:

Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVLTree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.Step

2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

Conclusion: In this way, we implemented the concept and basic of Height balanced Binary Search tree or AVL tree in Datastructure

Assignment No. 10

Problem Statement:

Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure. Analyse the algorithm.

Objective:

To understand the basic concept of Heap Data structure.

Outcome:

To implement the concept and basic of Max Heap and Min Heap Data Structure and its use in Data structure.

Software & Hardware Requirements:

1. 64-bit Open source Linux or its derivative

2. Open Source C++ Programming tool like G++/GCC

A double ended priority queue is a data structure that support the following operation

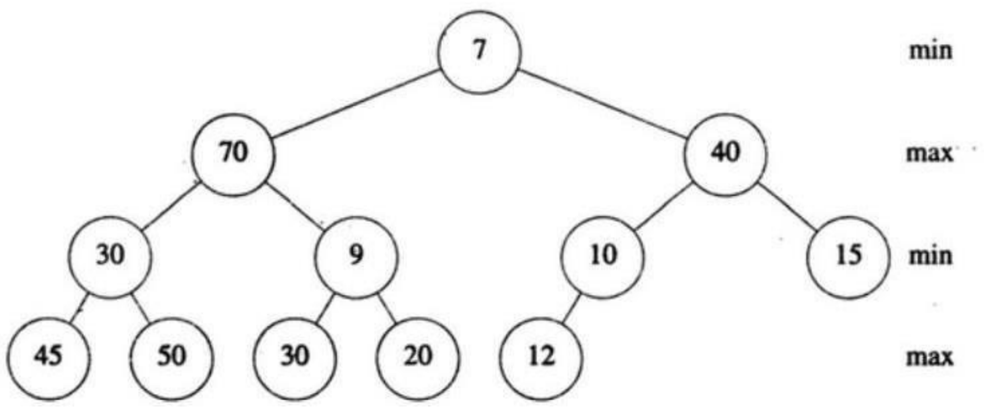
1. Inserting an element with an arbitrary key

2. Deleting an element with largest key

3. Deleting an element with smallest key

When only insertion and one of the two deletion operation re to be supported, a max heap or min heap may be used. A max-min heap support all of above operations.

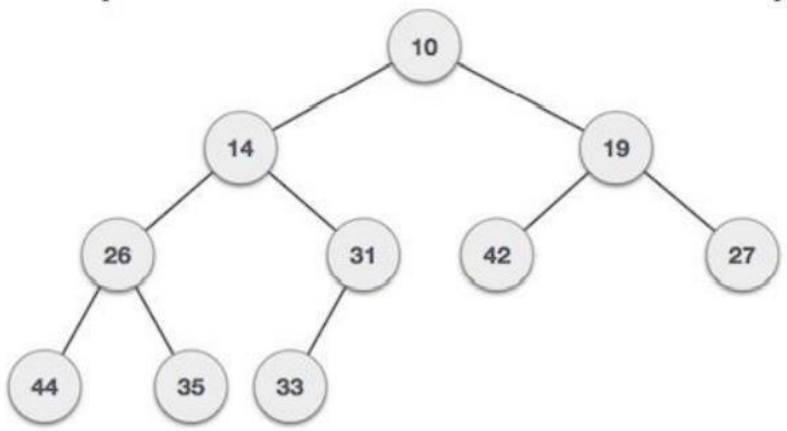
Definition: A min-Max heap is a complete binary tree such that if it is not empty, each element has a data member called key. Alternating levels of this tree are min levels and max level, respectively. the root is on min level.



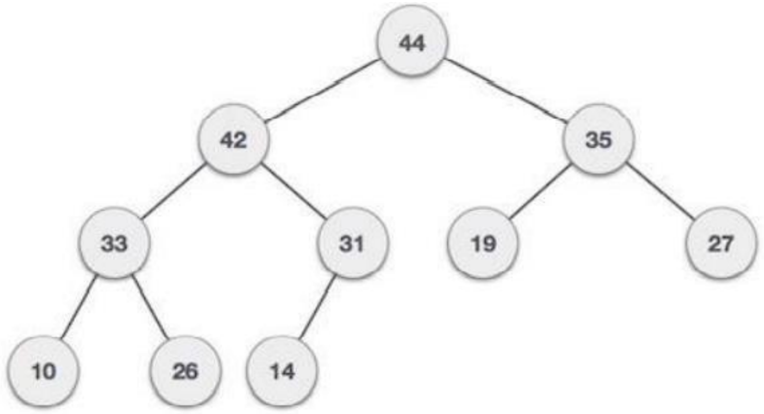
A 12 element Max-Min heap

```
template <class KeyType>
class DEPQ {
public:
    virtual void Insert(const Element<KeyType>&) = 0 ;
    virtual Element<KeyType>* DeleteMax(Element<KeyType>&) = 0 ;
    virtual Element<KeyType>* DeleteMin(Element<KeyType>&) = 0 ;
};
```

Min Heap: Where the value of the root node is less than or equal to either of its children



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



Algorithm:

Max heap Construction

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds

Max heap Deletion Algorithm

Step 1 − Remove root node.

Step 2 − Move the last element of last level to root.

Step 3 − Compare the value of this child node with its parent

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Max-Min Heap

Insert:

To add an element to a min-max heap perform following operations:

1. Append the required key to the array representing the min-max heap. This will likely break the min-max heap properties, therefore we need to adjust the heap.
2. Compare this key with its parent:

1. If it is found to be smaller (greater) compared to its parent, then it is surely

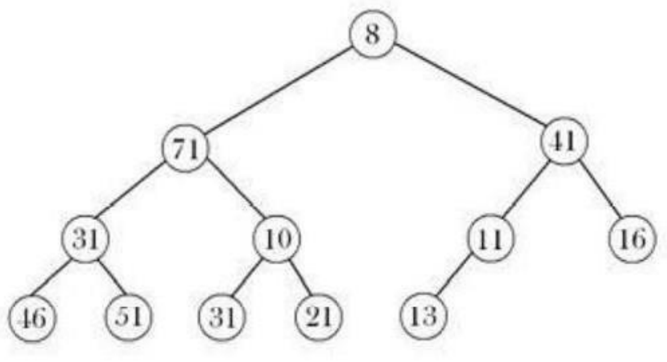
   smaller (greater) than all other keys present at nodes at max(min) level thatare on the path from the present position of key to the root of heap. Now, just check for nodes on Min(Max) levels.

2. If the key added is in correct order then stop otherwise swap that key with itsparent.

Example

Here is one example for inserting an element to a Min-Max Heap.

Say we have the following min-max heap and want to install a new node with value 6.



Initially, element 6 is inserted at the position indicated by j. Element 6 is less than its parent element. Hence it is smaller than all max levels and we only need to check the min levels. Thus, element 6 gets moved to the root position of the heap and the former root, element 8, gets moveddown one step.

If we want to insert a new node with value 81 in the given heap, we advance similarly. Initially thenode is inserted at the position j. Since element 81 is larger than its parent element and the parentelement is at min level, it is larger than all elements that are on min levels. Now we only need to check the nodes on max levels.

Delete :

Delete the minimum element

To delete min element from a Min-Max Heap perform following

operations.The smallest element is the root element.

Remove the root node and the node which is at the end of heap. Let it be x.

1. Reinsert key of x into the min-max heap

Reinsertion may have 2 cases:

1. If root has no children, then x can be inserted into the root.

2. Suppose root has at least one child. Find minimum value (Let this is be node m). m is in one of the children or grandchildren of the root. The following condition must be considered:

   1. x.key <= h[m].key: x must be inserted into the root.

   2. x.key > h[m].key and m are child of the root L: Since m is in max level, it has no descendants. So, the element h[m] is moved to the root and x is inserted into node m.

   3. x.key > h[m].key and m is grandchild of the root: So, the element h[m] is movedto the root. Let p be parent of m. if x.key > h[p].key then h[p] and x are interchanged.

Algorithm Average Worst

Case Insert O(log 2

n) O(log 2 n) Delete

O(log 2 n) O(log 2 n)

Conclusion: This program gives us the knowledge of basic of Max Heap and Min Heap Data Structure and its use in Datastructure.

Assignment No 11

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 thesystem displays the student details. Use sequential file to main the data.

Objectives:

1. To understand concept of file organization in data structure.

2. To understand concept & features of sequential file organization.

Learning Objectives:

To understand concept of file organization in data structure.

To understand concept & features of sequential file organization.

Learning Outcome:

Define class for sequential file using Object Oriented features. Analyse working of various operations on sequential file.

Theory:

File organization refers to the relationship of the key of the record to the physical location of thatrecord in the computer file. File organization may be either physical file or a logical file. A physicalfile is a physical unit, such as magnetic tape or a disk. A logical file on the other hand isa completeset of records for a specific application or purpose. A logical file may occupy a partof physical file or may extend over more than one physical file.

There are various methods of file organizations. These methods may be efficient for certain typesof access/selection meanwhile it will turn inefficient for other selections. Hence it is up to the programmer to decide the best suited file organization method depending on his requirement.

Some of the file organizations are

1. Sequential File Organization

2. Heap File Organization

3. Hash/Direct File Organization

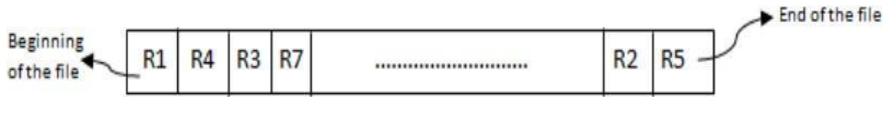
4. Indexed Sequential Access Method

5. B+ Tree File Organization

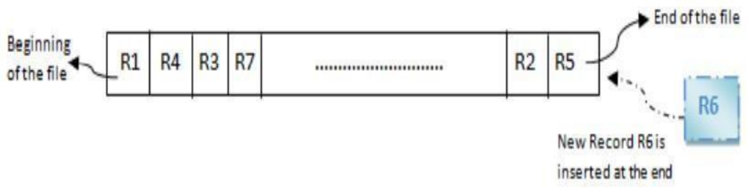
6. Cluster File Organization

Sequential File Organization:

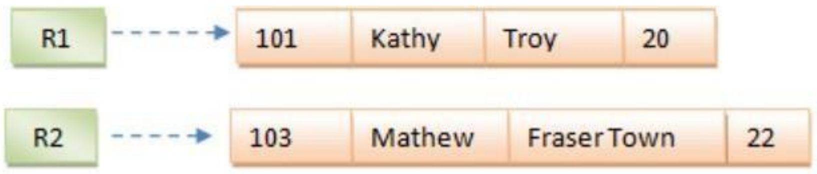
It is one of the simple methods of file organization. Here each file/records are stored oneafter the other in a sequential manner. This can be achieved in two ways: Records are stored one after the other as they are inserted into the tables. This method is called pile file method.
When a new record is inserted, it is placed at the end of the file. In the case of any modification or deletion of record,
The record will be searched in the memory blocks. Once itis found, it will be marked for deleting and new block of record is entered



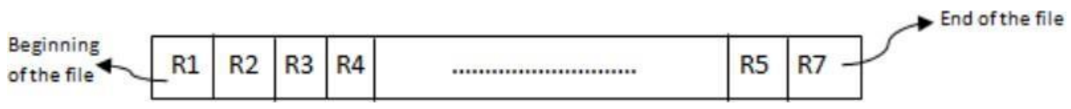
Inserting a new record:



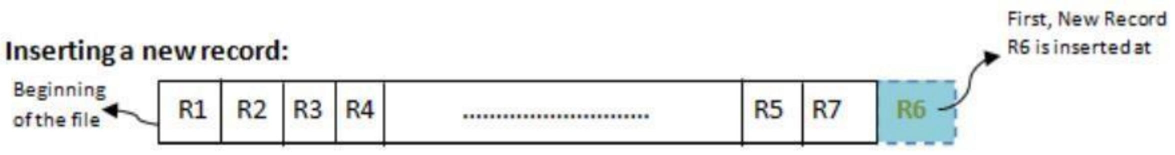
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2,R3 etc can be considered as one full set of attributes.

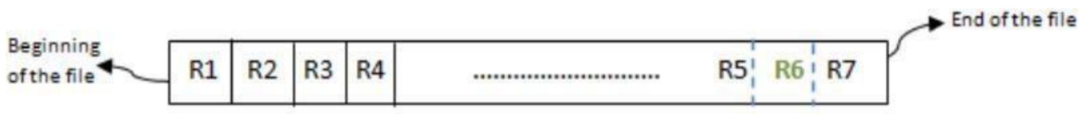


In the second method, records are sorted (either ascending or descending) each time they are inserted into the system. This method is called sorted file method. Sorting of records may be basedon the primary key or on any other columns. Whenever a new record is inserted, it will be insertedat the end of the file and then it will sort – ascending or descending based on key value and placedat the correct position. In the case of update, it will update the record and then sort thefile to placethe updated record in the right place. Same is the case with delete.



Inserting a new record:





Advantages:

- Simple to understand.

- Easy to maintain and organize
- Loading a record requires only the record key.
- Relatively inexpensive I/O media and devices can be used.
- Easy to reconstruct the files.
- The proportion of file records to be processed is high.

Disadvantages:

- Entire file must be processed, to get specific information.
- Very low activity rate stored.
- Transactions must be stored and placed in sequence prior to processing.
- Data redundancy is high, as same data can be stored at different places with differentkeys.
- Impossible to handle random enquiries.

Software Required: g++ / gcc compiler

Input: Details of student like roll no, name, address division etc.

Output: If record of student does not exist an appropriate message is displayed otherwise the student details are displayed.

Conclusion: This program gives us the knowledge of file organization.

Assignment No 12

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.

Objectives:

1. To understand concept of file organization in data structure.

2.To understand concept & features of indexed sequential file organization

Learning Objectives:

To understand concept of file organization in data structure.

To understand concept & features of indexed sequential file organization

Learning Outcome:

Define class for sequential file using Object Oriented features.

Analyse working of various operations on indexed sequential file.

Theory:

When there is need to access records sequentially by some key value and also to access records directly by the same key value, the collection of records may be organized in an effective mannedcalled Indexes Sequential Organization.

You must be familiar with search process for a word in a language dictionary. The data in the dictionary is stored in sequential manner. However an index is provided in terms of thumb tabs. To search for a word we do not search sequentially. We access the index that is the appropriate thumb tab, locate an approximate location for the word and then proceed to find the word sequentially.

To implement the concept of indexed sequential file organizations, we consider an approach in which the index part and data part reside on a separate file. The index file has a tree structure anddata file has a sequential structure. Since the data file is sequenced, it is not necessary for the indexto have an entry for each record Following figure shows a sequential file with a two-level index.

Level 1 of the index holds an entry for each three-record section of the main file. The level 2 indexes level 1 in the same way.

When the new records are inserted in the data file, the sequence of records need to be preserved and also the index is accordingly updated.

Two approaches used to implement indexes are static indexes and dynamic indexes.

As the main data file changes due to insertions and deletions, the static index contents may changebut the structure does not change. In case of dynamic indexing approach, insertions and deletionsin the main data file may lead to changes in the index structure. Recall the change in height of B-Tree as records are inserted and deleted. Both dynamic and static indexing techniques are useful depending on the type of application.

Conclusion: This program gives us the knowledge of file organization