PUNE INSTITUTE OF COMPUTER TECHNOLOGY, PUNE

ACADEMIC YEAR: 2016-17

DEPARTMENT of COMPUTER ENGINEERING DEPARTMENT

CLASS: S.E. **SEMESTER: II**

SUBJECT: Advanced Data Structure Laboratory

ASSINGMENT NO.	1
TITLE	Binary Tree traversal
PROBLEM STATEMENT /DEFINITION	For given expression eg. a-b*c-d/e+f construct inorder sequence and traverse it using postorder and preorder traversal(non-recursive, recursive).
OBJECTIVE	To understand construction of binary tree and its traversal techniques.
OUTCOME	At the end of this assignment students will able to construct an Expression tree and perform basic operations on Binary tree.
S/W PACKAGES AND HARDWARE APPARATUS USED	1. (64-bit)64-BIT Fedora 17 or latest 64-BIT Update of Equivalent Open source OS 2. Programming Tools (64-Bit) Latest Open source update of Eclipse Programming frame work, TC++, GTK++.
REFERENCES	• E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 • Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
	1. Date

INSTRUCTIONS FOR WRITING JOURNAL	2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 9. Conclusion/Analysis

Prerequisites:

Object oriented programming, and basic concepts of data structures.

Concepts related Theory:

Binary tree is specific type of tree in which each node can have atmost(zero,one,two) two children namely left child and right child. Empty tree also a valid binary tree.

In computer science**,** tree traversal is a form of graph traversal and refers to the process of visiting each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

Expression tree: An expression tree is a tree whose leaves contain the operands of the expression,
and the other nodes contain the operators.

Constructing an expression tree:

The expression tree can be built using the postfix expression. We read the postfix expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it into a stack. If the symbol is an operator, we pop twice pointers to two trees T1 and T2 (T1 is popped first) and form a new tree whose root is the operator and whose left and right children are T2 and T1, respectively. A pointer to this new tree is then pushed onto the stack.

Data structures for tree traversal:

Traversing a tree involves iterating over all nodes in some manner. Because from a given node there is more than one possible next node then, assuming sequential computation, some nodes must be deferred—stored in some way for later visiting. This is often done via a stack (LIFO) or

queue(FIFO). As a tree is a self-referential (recursively defined) data structure, traversal can be defined by recursion

Depth-first search is easily implemented via a stack, including recursively, while breadth-first search is easily implemented via a queue, including corecursively.

Depth-first search:

These searches are referred to as *depth-first search* (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling. For a binary tree, they are defined as display operations recursively at each node, starting with the root.

***Operations of binary tree*:**

- Traversal
- Creation
- Deletion
- Compare
- Merge

Traversing: Traversal refers to the process of visiting all the nodes of binary tree once. There are three ways for traversing binary tree.

1.*Pre-order:*

- Check if the current node is empty /null
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.

2.*In-order:*

- Check if the current node is empty/null
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.

3.*Post-order:*

- Check if the current node is empty/null
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of the root (or current node).

Algorithm:

ALGORITHM INORDERTRAVERSE()
{
1. set top=0, stack[top]=NULL, ptr = root
2. Repeat while ptr!=NULL
    2.1 set top=top+1
    2.2 set stack[top]=ptr
    2.3 set ptr=ptr->left
3. Set ptr=stack[top], top=top-1
4. Repeat while ptr!=NULL
    4.1 print ptr->info
    4.2 if ptr->right!=NULL then
        4.2.1set ptr=ptr->right
        4.2.2 goto step 2
    4.3 Set ptr=stack[top], top=top-1
}

ALGORITHM PREORDERTRAVERSE()
{
1. set top=0, stack[top]=NULL, ptr = root
2. Repeat while ptr!=NULL
    2.1 print ptr -> info
    2.2 if (ptr -> right != NULL)
        2.2.1 top = top +1

2.2.2 set stack [top] = ptr -> right

2.3 if (ptr -> left != NULL)

2.3.1 ptr=ptr -> left

else

2.3.1 ptr=stack[top], top=top-1

}

ALGORITHM POSTORDERTRAVERSE()

{

1. set top = 0, stack [top] = NULL, ptr = root
2. Repeat while ptr!=NULL

2.1 top = top +1 , stack [top] = ptr

2.2 if (ptr -> right != NULL)

2.2.1 top = top +1

2.2.2 set stack [top] = - (ptr -> right)

2.3 ptr = ptr -> left

3. ptr = stack [top], top = top-1
4. Repeat while (ptr > 0)

4.1 print ptr -> info

4.2 ptr = stack [top], top = top-1

5. if (ptr < 0)

5.1 set ptr = - ptr

5.2 Go to step 2

}

Conclusion: After successfully completing this assignment, Students will be able create an Expression tree and performs various operations on Binary tree.

Review Questions:

1.Define the following terms

- Tree

- Binary tree
- Complete binary tree
- Strictly binary tree
- Almost complete binary tree
- Binary search tree
- Root
- Leaf

2.What are the operation which can be performed on tree ?

3.What are the operation which can be performed on binary tree ?

4.What are the 3 traversal techniques of binary tree?

5.What is binary search tree ?

6.What are the 3 traversal techniques of binary search tree?

7. What is List ? Explain operations on list.

8.What is Linked List ? List types of Linked List.

9.What is Singly Linked List explain with it's all operations.

10.Draw a full binary tree with at least 6 nodes.

11.Draw a complete binary tree with exactly six nodes. Put a different value in each node. Then draw an array with six components and show where each of the six node values would be placed in the array (using the usual array representation of a complete binary tree).

12.Suppose that we want to create a binary search tree where each node contains information of some data type called Item (which has a default constructor and a correct value semantics). What additional factor is required for the Item data type?

13.Consider the node of a complete binary tree whose value is stored in data[i] for an array implementation. If this node has a right child, where will the right child's value be stored?

14.Explain, in your own words, how a binary search works on a sorted array [1..N] of integers. Why is this search method preferred over a regular search ?

15. What is an Expression tree in data structure?

ASSINGMENT NO.	2
TITLE	To write a program for implementing dictionary using binary search tree.
PROBLEM STATEMENT /DEFINITION	A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry, assign a given tree into another tree(=). 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.
OBJECTIVE	• To understand Binary Search Tree implementation. • To understand operation performed on tree such as addition, deletion, updating of keywords. • To connect one tree to another tree • To understand sorting of tree.
OUTCOME	• Dictionary implementation using Binary Search Tree • Various operations performed on tree
S/W PACKAGES AND HARDWARE APPARATUS USED	• 64-bit Open source Linux or its derivative. • Open Source C++ Programming tool like G++/GCC.
REFERENCES	Data structures in C++ by Horowitz, Sahni.

INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 10. Conclusion/Analysis

Assignment 2:

Aim: To write a program for Dictionary implementation using Binary Search Tree

Prerequisites:

Basic knowledge of linked list, searching

Linked list node deletion, addition, updating

Object oriented programming features

Learning Objectives:

To understand binary search tree implementation

Learning Outcomes

After successful completion of this assignment, students will be able to

- Implement graph using adjacency matrix or adjacency list.
- Create minimum cost spanning tree using Prim's or Kruskal's algorithm.

Concepts related Theory:

In computer science, binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

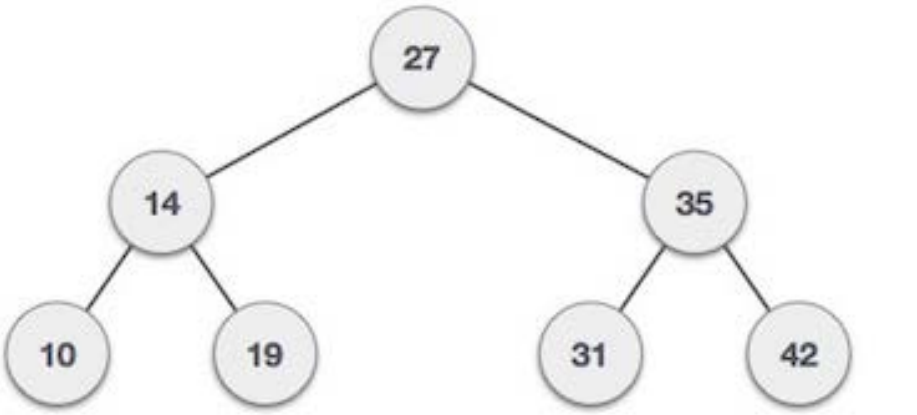
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments;

Representation:

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations:

Following are the basic operations of a tree −

- Search − Searches an element in a tree.
- Insert − Inserts an element in a tree.
- Deletion − Delete an element in a tree.

Node:- Define a node having some data, references to its left and right child nodes.

Search Operation:

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm:

```
struct node* search(int data){
  struct node *current = root;
  printf("Visiting elements: ");
  while(current->data != data){
```

```
      if(current != NULL) {
        printf("%d ",current->data);
        //go to left tree
        if(current->data > data){
          current = current->leftChild;
        }//else go to right tree
        else {
          current = current->rightChild;
        }
        //not found
        if(current == NULL){
          return NULL;
        }
      }
    }
    return current;
  }
```

Insert Operation:

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
```

```
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;
  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;
    while(1) {
      parent = current;
      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left
        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }//go to right of the tree
      else {
        current = current->rightChild;
        //insert to the right
        if(current == NULL) {
```

```
            parent->rightChild = tempNode;
            return;
          }
        }
      }
    }
}
```

Deletion:

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node 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. Copy the value of R to N, then recursively call delete on the original R until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of the first 2 cases.

Algorithm:

```
bool BinarySearchTree::remove(int value) {
    if (root == NULL)
        return false;
   else {
        if (root->getValue() == value) {
            BSTNode auxRoot(0);
            auxRoot.setLeftChild(root);
            BSTNode* removedNode = root->remove(value, &auxRoot);
```

```
            root = auxRoot.getLeft();
            if (removedNode != NULL) {
                delete removedNode;
                return true;
            } else
                return false;
        } else {
            BSTNode* removedNode = root->remove(value, NULL);
            if (removedNode != NULL) {
                delete removedNode;
                return true;
            } else
                return false;
           }
    }
}
BSTNode* BSTNode::remove(int value, BSTNode *parent) {
    if (value < this->value) {
        if (left != NULL)
            return left->remove(value, this);
        else
            return NULL;
    } else if (value > this->value) {
        if (right != NULL)
            return right->remove(value, this);
        else
            return NULL;
```

```
        } else {
            if (left != NULL && right != NULL) {
                this->value = right->minValue();
                return right->remove(this->value, this);
            } else if (parent->left == this) {
                parent->left = (left != NULL) ? left : right;
                return this;
            } else if (parent->right == this) {
                parent->right = (left != NULL) ? left : right;
                return this;
            }
        }
    }
    int BSTNode::minValue() {
        if (left == NULL)
            return value;
        else
            return left->minValue();}
```

Threaded Binary Tree:

Threaded binary tree is a binary tree variant that allows fast traversal: given a pointer to a node in a threaded tree, it is possible to cheaply find its in-order successor (and/or predecessor).

Binary trees, including (but not limited to) binary search trees and their variants, can be used to store a set of items in a particular order. For example, a binary search tree assumes data items are somehow ordered and maintain this ordering as part of their insertion and deletion algorithms. One useful operation on such a tree is traversal: visiting the items in the order in which they are stored (which matches the underlying ordering in the case of BST).

Algorithm:

1. For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.

2. Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.
3. Print the node and If node has right child then go to step 4 else go to step 5.
4. Make right child as current node.
5. if there is a thread node then make it the current node.
6. if all nodes have been printed then END else go to step 1.

Review Questions:

1. What is binary search tree?
2. What are threaded binary tree?
3. How do you find the depth of a binary tree?
4. Explain pre-order and in-order tree traversal?
5. Define threaded binary tree. Explain its common uses.
6. Explain implementation of traversal of a binary tree.
7. Explain implementation of deletion from a binary tree.

ASSINGMENT NO.	3
TITLE	Threaded binary tree
PROBLEM STATEMENT /DEFINITION	Convert given binary tree into inordered and preordered threaded binary tree. Analyze time and space complexity of the algorithm.
OBJECTIVE	To understand construction of inordered and preordered Threaded binary tree from given binary tree.
OUTCOME	At the end of this assignment students will able to construct threaded binary tree and able to perform basic operations on Threaded binary tree.
S/W PACKAGES AND HARDWARE APPARATUS USED	• (64-bit)64-BIT Fedora 17 or latest 64-BIT Update of Equivalent Open source OS • Programming Tools (64-Bit) Latest Open source update of Eclipse Programming frame work, TC++, GTK++.
REFERENCES	E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome

	6. Concepts related Theory 7. Algorithm 8. Test cases 9. Conclusion/Analysis

Prerequisites:

Object oriented programming,features and basic concepts of data structures.

Concepts related Theory:

Binary Tree: is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list. A binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers each point to a binary tree.

In a binary search tree, there are many nodes that have an empty left child or empty right child or both. You can utilize these fields in such a way so that the empty left child of a node points to its inorder predecessor and empty right child of the node points to its inorder successor

Threaded Binary Tree: A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node**.**

It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable.

One way threading:-A thread will appear in the right field of a node and will point to the next node in the inorder traversal.

Two way threading:- A thread will also appear in the left field of a node and will point to the preceding node in the inorder traversal.

Types of threaded binary tree

Single threaded: Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor or all left null pointers will point to inorder predecessor.

Double threaded: Each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.

Threaded binary search tree: Threaded binary search tree is BST in which all right pointers of node which point to NULL are changed and made to point to inorder successor current node (These are called as single threaded trees). In completely threaded tree (or double threaded trees), left pointer of node which points to NULL is made to point to inorder predecessor of current node if inorder predecessor exists.

Now, there is small thing needs to be taken care of. A right or left pointer can have now two meanings : One that it points to next real node, second it is pointing inorder successor or predecessor of node, that means it is creating a thread. To store this information, we added a bool in each node, which indicates whether pointer is real or thread.

Approach:

1. we can do the inorder traversal and store it in some queue. Do another inorder traversal and where ever you find a node whose right reference is NULL , take the front element from the queue and make it the right of the current node.

2.Now we will see the solution which will convert binary tree into threaded binary tree in one single traversal with no extra space required.

- Do the reverse inorder traversal, means visit right child first.
- In recursive call, pass additional parameter, the node visited previously.
- whenever you will find a node whose right pointer is NULL and previous visited node is not NULL then make the right of node points to previous visited node and mark the boolean right threaded as true.
- Important point is whenever making a recursive call to right subtree, do not change the previous visited not and when making a recursive call to left subtree then pass the actual previous visited node.

Algorithm:

```
public void convert(Node root){

inorder(root, null);

}
```

```
public void inorder(Node root, Node previous){
f(root==null){
return;
}else{
inorder(root.right, previous);
if(root.right==null && previous!=null){
root.right = previous;
root.rightThread=true;
}
inorder(root.left, root);
}
}
public void print(Node root){
//first go to most left node
Node current = leftMostNode(root);
//now travel using right pointers
while(current!=null){
System.out.print(" " + current.data);
//check if node has a right thread
if(current.rightThread)
current = current.right;
current = current.right;
else // else go to left most node in the right subtree
current = leftMostNode(current.right);
}
```

```
System.out.println()

}

public Node leftMostNode(Node node){

if(node==null){

return null;

}else{

while(node.left!=null)

node = node.left

}

return node;

}

}
```

Conclusion: After successfully completing this assignment, Students have learned construction of Threaded binary tree and various operations on Threaded binary tree.

Review Questions:

1. What is binary tree? Explain its uses.
2. How do you find the depth of a binary tree?
3. Explain pre-order and in-order tree traversal.
4. Define threaded binary tree. Explain its common uses.
5. Explain implementation of traversal of a binary tree.
6. Why do we need Threaded binary search trees ?
7. How to created a threaded BST?
8. How to convert a Given Binary Tree to Threaded Binary Tree?
9. What are the drawbacks of bi-threaded trees? Are single threaded trees enough to do traversals on the tree? Justify.
10. Did you require stacks to do get the output along with threads? Justify.

11. Justify that only single threads are required to traverse the tree efficiently.

12. If a tree is bi threaded for postorder traversal, and then is it enough to traverse it in in-order and post-order. Justify

ASSINGMENT NO.	**4**
TITLE	Adjacency list representation of the graph or use adjacency matrix representation of the graph.
PROBLEM STATEMENT /DEFINITION	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 takes 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.(Operation to be performed adding and deleting edge, adding and deleting vertices, calculated in-degree and out-degree for both directed and undirected graph)
OBJECTIVE	1. Define a graph (undirected and directed), a vertex/node, and an edge. 2. Given the figure of a graph, give its set of vertices and set of edges. 3. Given the set of vertices and set of edges of a graph, draw a figure to show the graph. 4. Given a graph, show its representations using an adjacency list and adjacency matrix. Also give the space required for each of those representations. 6. Given a DAG, show the steps in topologically sorting the vertices, and give the time complexity of the algorithm.
OUTCOME	The adjacency list easily find all the links that are directly connected to a particular vertex i.e. edge between the cities.
S/W PACKAGES AND HARDWARE APPARATUS USED	• (64-bit)64-BIT Fedora 17 or latest 64-BIT Update of Equivalent Open source OS • Programming Tools (64-Bit) Latest Open source update of Eclipse Programming frame work, TC++, GTK++.
REFERENCES	E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 Sartaj Sahani, ―Data Structures, Algorithms andApplications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.

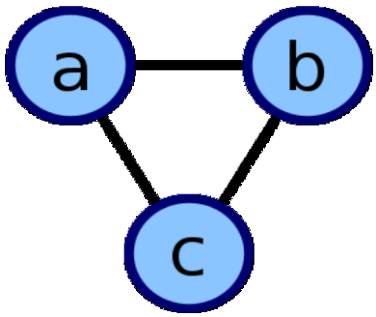
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 9. Conclusion/Analysis

Prerequisites:

Object oriented programming,features and basic concepts of data structures.

Concepts related Theory:

An adjacency list representation for a graph associates each vertex in the graph with the collection of its neighboring vertices or edges. There are many variations of this basic idea, differing in the details of how they implement the association between vertices and collections, in how they implement the collections, in whether they include both vertices and edges or only vertices as first class objects, and in what kinds of objects are used to represent the vertices and edges.



This undirected cyclic graph can be described by the three unordered lists {b, c}, {a, c}, {a, b}

The graph pictured above has this adjacency list representation:		
a	adjacent to	bc
b	adjacent to	ac
c	adjacent to	ab

OPERATION:

The main operation performed by the adjacency list data structure is to report a list of the neighbors of a given vertex. this can be performed in constant time per neighbor. In other words, the total time to report all of the neighbors of a vertex *v* is proportional to the degree of *v*.

It is also possible, but not as efficient, to use adjacency lists to test whether an edge exists or does not exist between two specified vertices. In an adjacency list in which the neighbors of each vertex are unsorted, testing for the existence of an edge may be performed in time proportional to the minimum degree of the two given vertices, by using a sequential search through the neighbors of this vertex. If the neighbors are represented as a sorted array, binary search may be used instead, taking time proportional to the logarithm of the degree.

Trade-offs:

The main alternative to the adjacency list is the adjacency matrix, a matrix whose rows and columns are indexed by vertices and whose cells contain a Boolean value that indicates whether an edge is present between the vertices corresponding to the row and column of the cell. For a sparse graph (one in which most pairs of vertices are not connected by edges) an adjacency list is significantly more space-efficient than an adjacency matrix (stored as an array): the space usage of the adjacency list is proportional to the number of edges and vertices in the graph, while for an adjacency matrix stored in this way the space is proportional to the square of the number of vertices. However, it is possible to store adjacency matrices more space-efficiently, matching the linear space usage of an adjacency list, by using a hash table indexed by pairs of vertices rather than an array.

The other significant difference between adjacency lists and adjacency matrices is in the efficiency of the operations they perform. In an adjacency list, the neighbors of each vertex may be listed efficiently, in time proportional to the degree of the vertex. In an adjacency matrix, this operation takes time proportional to the number of vertices in the graph, which may be significantly higher than the degree. On the other hand, the adjacency matrix allows testing whether two vertices are adjacent to each other in constant time; the adjacency list is slower to support this operation.

Graph Representation — Adjacency Matrix and Adjacency List:

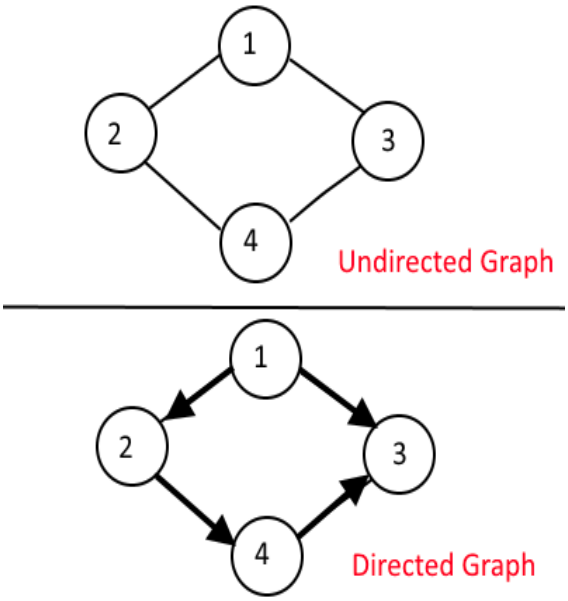
What is Graph ?

G = (V,E)

Graph is a collection of nodes or vertices (V) and edges(E) between them. We can traverse these nodes using the edges. These edges might be weighted or non-weighted.

There can be two kinds of Graphs

- Un-directed Graph — when you can traverse either direction between two nodes.
- Directed Graph — when you can traverse only in the specified direction between two nodes.



Now how do we represent a Graph, There are two common ways to represent it:

- Adjacency Matrix
- Adjacency List

Adjacency Matrix:

Adjacency Matrix is 2-Dimensional Array which has the size VxV, where V are the number of vertices in the graph. See the example below, the Adjacency matrix for the graph shown above.

adjMa

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

P:F-LTL-

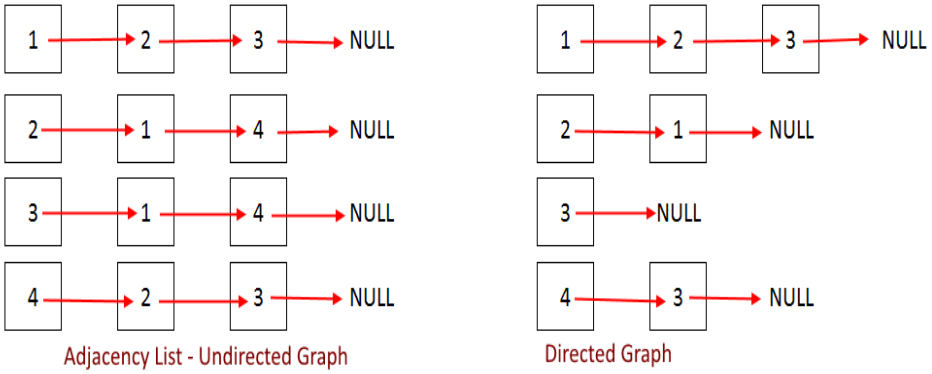
	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	0	0	0	0
4	0	0	1	0

It’s easy to implement because removing and adding an edge takes only O(1) time.

But the drawback is that it takes $O(V^2)$ space even though there are very less edges in the graph.

Adjacency List:

Adjacency List is the Array[] of Linked List, where array size is same as number of Vertices in the graph. Every Vertex has a Linked List. Each Node in this Linked list represents the reference to the other vertices which share an edge with the current vertex. The weights can also be stored in the Linked List Node.



Algorithm:

```
class Edge

{
private:
Vertex *source;
Vertex *destination;
int distance;
public:
Edge(Vertex *s, Vertex *d, int dist)
{
source = s;
destination = d;
distance = dist;
}

Vertex *getSource()
{
return source;
}
Vertex * getDestination()
{
```

```
return destination;
}
int getDistance()
{
return distance;
}
};
class Vertex
{
private : string city;
vector<Edge> edges;//vector edges created for edges
public:
Vertex(string name)

{
city = name;
}
void addEdge(Vertex *v, int dist)
{
Edge newEdge(this,v,dist);// creating object for edge
edges.push_back(newEdge);//creating adjusting list
}
void showEdge()
{
cout<<"From"<<city<<"to"<<endl;
for(int i=0; i<(int)edges.size();i++)
{
Edge
e = edges[i];
cout<<e.getDestination()
-
>getCity()<<"requires"<<e.getDistance()<<"hrs"<<endl;
}
cout<<endl;
}
string getCity()
{
return city;
}
vector<Edge> getEdges()
{
return edges;
}
};
class Graph
{
vector<Vertex*> v;
```

```
public:
Graph(){}
void insert(Vertex *val)
{
v.push_back(val);
}
void Display()
{
for(int i=0;i<(int)v.size();i++)
{
// v[i].showEdge();
v[i]
-
>showEdge();
}
}
};
int main()
{
Graph g;
// crea
ting verticex ot nodes for each city
Vertex v1 = Vertex("Mumbai");
Vertex v2 = Vertex("Pune");
Vertex v3 = Vertex("Kolkata");
Vertex v4 = Vertex("Delhi");
//creating pointers to nodes
Vertex *vptr1 = &v1;
Vertex *vptr2 = &v2;
Vertex *vptr3 = &v3;
Vertex *vptr4 = &v4;
//attaching the nodes by adding edges
v1.addEdge(vptr4,2);
v2.addEdge(vptr1,1);
v3.addEdge(vptr1,3);
v4.addEdge(vptr2,2);
v4.addEdge(vptr3,3);
//cretaing graph
g.insert(vptr1);
g.insert(vpt
r2);
g.insert(vptr3);
g.insert(vptr4);
cout<<"
\
n
```

```
\
t Displaying City Transport Map Using Adjacency List"<<endl;
g.Display();
return 1;
}
```

TEST CASES:

Check The is there adjacency list find all the edges that are directly connected to a cities or not.

Conclusion:

Thus we have studies adjacency list representation of the graph successfully for cities.

Review Questions:

1. What is Graph? Explain its uses.
2. Explain Adjacency List.
3. Explain Adjacency Matrix.
4. What is the difference between undirected and directed graph?
5. Explain Sparse graph.

ASSINGMENT NO.	5
TITLE	To write a program for Graph creation and find its minimum cost using Prim's or Kruskal's algorithm.
PROBLEM STATEMENT /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.
OBJECTIVE	• To understand concept of graph & minimum cost spanning tree. • To understand different minimum cost spanning tree algorithms. • To implement minimum spanning tree algorithms.
OUTCOME	• Graph implementation using Adjacency matrix or Adjacency list • Find total minimum cost using MST Algorithm
S/W PACKAGES AND HARDWARE APPARATUS USED	• 64-bit Open source Linux or its derivative. • Open Source C++ Programming tool like G++/GCC.
REFERENCES	Data structures in C++ by Horowitz, Sahni.
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 10. Conclusion/Analysis

Assignment 5

Aim: To write a program for Graph creation and find its minimum cost using Prim's or Kruskal's algorithm.

Prerequisites:

Basic knowledge of graph

Graph representation method (Adjacency matrix or Adjacency list)

Object oriented programming features

Learning Objectives

To understand concept of graph and minimum cost spanning tree.

To understand minimum cost spanning tree algorithms.

Learning Outcomes

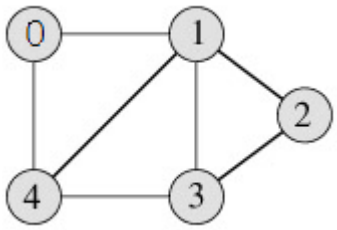
After successful completion of this assignment, students will be able to

- Implement graph using adjacency matrix or adjacency list.
- Create minimum cost spanning tree using Prim's or Kruskal's algorithm.

Concepts related Theory:

- **Representation of Graph**

Following is an example undirected graph with 5 vertices.



➢ **Using Adjacency Matrix**

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from

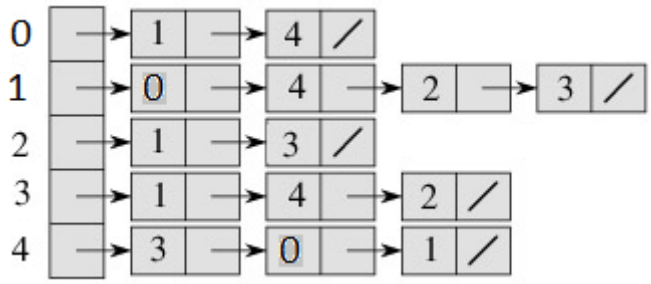
vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

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

➢ **Using Adjacency list**

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



- **Minimum Spanning Tree:**

Give a graph G = (V, E), the minimum spanning tree (MST) is a weighted graph G' = (V, E') such that:

- E' Í E
- G' is connected
- G' has the minimum cost

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components. There are quite a few use cases for minimum spanning trees. One example would be a telecommunications company which is trying to lay out cables in new neighborhood.

- **Prim's Algorithm**

Step 1: Select any vertex

Step 2: Select the shortest edge connected to that vertex.

Step 3: Select the shortest edge connected to any vertex already connected

Step 4: Repeat step 3 until all vertices have been connected

- **Kruskal's Algorithm:**

Step 1: Enter number of cities (vertices in graph).

Step 2: Enter the cost of connectivity between each pair of cities (edges in graph).

Step 3: Initialize cost_of_connectivity to 0.

Step 4: Sort all the edges in non-decreasing order of their cost.

Step 5: Pick the smallest cost edge

Step 6: Check if it forms a cycle with the already include edges in the minimum spanning tree

Step 7: If cycle is not formed, include this edge in MST, else discard it

Step 8: Add weight of the selected edge to the cost_of_connectivity.

Step 9: Repeat step 5 ,6, 7 until there are (v-1) edges in the graph.

Step 10: cost_of_connectivity will have minimum cost in the end

Algorithm:

- **Prim's Algorithm**

// input: a graph G

// output: E: a MST for G

1. Select a starting node, v
2. T ß {v} //the nodes in the MST
3. E ß {} //the edges in the MST
4. While not all nodes in G are in the T do

Choose the edge v' in G − T such that there is a v in T:

weight (v,v') is the minimum in

{weight(u,w) : w in G − T and u in T}

T ß T È {v'}

E ß E È {(v,v')}

5. return E

- **Kruskal's Algorithm:**

// input: a graph G with n nodes and m edges

// output: E: a MST for G

- EG[1..m] □ Sort the m edges in G in increasing weigh
- E □ { } //the edges in the MST
- i □ 1 //counter for EG
- While |E|<n-1 do

if adding EG[i] to E does not add a cycle then

E □ E □{EG[i]}

i □ i + 1

- return E

Conclusion: We have successfully calculated total minimum cost of graph using minimum spanning tree algorithm.

Review Questions:

1) What is minimum spanning tree?

2) What are the algorithms to find minimum spanning tree?

3) What is Time and space complexity of the algorithm used?

4) What is adjacency list and adjacency matrix?

5) Difference between adjacency list and adjacency matrix.?

6) Draw and compare graph using Prim's and Kruskal's algorithm?

7) Explain steps in kruskal's algorithm ?

8) Explain steps in prim's algorithm?

9) What are Applications of minimum spanning tree?

10) Explain number of edges in any Minimum Spanning Tree?

ASSINGMENT NO.	6
TITLE	The Dictionary ADT
PROBLEM STATEMENT /DEFINITION	Implement all the functions of a dictionary (ADT) using hashing. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique Standard Operations: Insert(key, value), Find(key), Delete(key)
OBJECTIVE	To understand implementation of all the functions of a dictionary (ADT) and standard operations on Dictionary.
OUTCOME	At the end of this assignment students will able to perform standard operations on Dictionary ADT.
S/W PACKAGES AND HARDWARE APPARATUS USED	• (64-bit)64-BIT Fedora 17 or latest 64-BIT Update of Equivalent Open source OS • Programming Tools (64-Bit) Latest Open source update of Eclipse Programming frame work, TC++, GTK++.
REFERENCES	• E. Horowitz S. Sahani, D. Mehata, "Fundamentals of data structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 • Sartaj Sahani, ―Data Structures, Algorithms and Applications in C++‖, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 9. Conclusion/Analysis

Prerequisites:

- Basic knowledge of Dictionary and Hashing.
- Object oriented programming,features and basic concepts of data structures.

Concepts related Theory:

The Dictionary ADT: A dictionary is an ordered or unordered list of key-element pairs, where keys are used to locate elements in the list.

Dictionary is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and look up for a value by the key. Values are not required to be unique.

Example: consider a data structure that stores bank accounts; it can be viewed as a dictionary, where account numbers serve as keys for identification of account objects.

A Dictionary (also known as Table or Map) can be implemented in various ways: using a list, binary search tree, hash table, etc.

In each case: the implementing data structure has to be able to hold key-data pairs and able to do insert, find, and delete operations paying attention to the key.

Hashing: Hashing is a method for directly referencing an element in a table by performing arithmetic transformations on keys into table addresses. This is carried out in two steps:

3. Computing the so-called hash function H: K -> A.
4. Collision resolution, which handles cases where two or more different keys hash to the same table address.

Implementation of Hash table:

Hash tables consist of two components: a *bucket array* and a *hash function*.

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on.

Consider a dictionary, where keys are integers in the range [0, N-1]. Then, an array of size N can be used to represent the dictionary. Each entry in this array is thought of as a "bucket". An element *e* with key *k* is inserted in A[k]. Bucket entries associated with keys not present in the dictionary contain a special NO_SUCH_KEY object. If the dictionary contains elements with the same key, then two or more different elements may be mapped to the same bucket of A. In this case, we say that a *collision* between these elements has occurred. One easy way to deal with collisions is to allow a sequence of elements with the same key, k, to be stored in A[k].

Assuming that an arbitrary element with key k satisfies queries findItem(k) and removeItem(k), these operations are now performed in O(1) time, while insertItem(k, e) needs only to find where on the existing list A[k] to insert the new item, e. The drawback of this is that the size of the bucket array is the size of the set from which key are drawn,which may be huge.

Algorithm:

HashNode Class Declaration:

```
class HashNode

{
public:
int key;
int value;
HashNode* next;
HashNode(int key, int value)
{
this->key = key;
this->value = value;
this->next = NULL;
}
};
```

Insertion:

```
void Insert(int key, int value)
{
int hash_val = HashFunc(key);
HashNode* prev = NULL;
HashNode* entry = htable[hash_val];
while (entry != NULL)
{
prev =entry;
entry = entry->next;
}
if (entry == NULL)
{
entry = new HashNode(key, value);
if (prev == NULL)
{
```

```
htable[hash_val] = entry;
}
else
{
prev->next = entry;
}
}
else
{
entry->value = value;
}
}
```

Deletion:

```
void Remove(int key)
{
int hash_val = HashFunc(key);
HashNode* entry = htable[hash_val];
HashNode* prev = NULL;
if (entry == NULL || entry->key != key)
{
cout<<"No Element found at key "<<key<<endl;
return;
}
while (entry->next != NULL)
{
prev = entry;
entry = entry->next;
}
if (prev != NULL)
{
prev->next = entry->next;
}
delete entry;
cout<<"Element Deleted"<<endl;
}
```

Search:

```
int Search(int key)
```

```
{
bool flag = false;
int hash_val = HashFunc(key);
HashNode* entry = htable[hash_val];
while (entry != NULL)
{
if (entry->key == key)
{
cout<<entry->value<<" ";
flag = true;
}
entry = entry->next;
}
if (!flag)
return -1;
}
};
```

Conclusion: After successfully completing this assignment, Students have learned implementation of Dictionary(ADT) using Hashing and various Standard operations on Dictionary ADT .

Review Questions:

- In what ways is a dictionary similar to an array? In what ways are they different?
- What does it mean to hash a value? What is a hash function?
- What is a perfect hash function?
- What is a collision of two values?
- What does it mean to probe for a free location in an open address hash table?
- What is the load factor for a hash table?
- Why do you not want the load factor to become too large?
- Can you come up with a perfect hash function for the names of the week? The names of the months? The names of the planets?

- How to define a good hash function?
- What is the best definition of a collision in a hash table?
- Describe in reasonable detail a way to implement the Dictionary ADT such that the **insertItem**, **findItem**, and **removeItem** methods would all run in $O(1)$ time, assuming that all of the keys associated with elements in the structure are integers in the range

ASSINGMENT NO.	7
TITLE	To write a program for implementation of symbol table. And perform various operations.
PROBLEM STATEMENT /DEFINITION	The symbol table is generated by compiler. From this perspective, the symbol table is a set of name-attribute pairs. In a symbol table for a compiler, the name is an identifier, and the attributes might include an initial value and a list of lines that use the identifier. Perform the following operations on symbol table: (1) Determine if a particular name is in the table (2) Retrieve the attributes of that name (3) Modify the attributes of that name (4) Insert a new name and its attributes (5) Delete a name and its attributes
OBJECTIVE	1) To understand concept of symbol table. **2)** Why symbol table is needed.
OUTCOME	1) Use of symbol table **2)** Various methods of implementing symbol table.
S/W PACKAGES AND HARDWARE APPARATUS USED	• 64-bit Open source Linux or its derivative. • Open Source C++ Programming tool like G++/GCC.
REFERENCES	• Data structures in C++ by Horowitz, Sahni.

INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 10. Conclusion/Analysis

Assignment 7

- Aim: To write a program for implementation of symbol table. And perform various operations.

Prerequisites:

- Basic knowledge of array implementation
- Linked list implementation
- Basic knowledge of binary search tree
- Object oriented programming features

Learning Objectives

- To understand concept of symbol table and its use.
- To perform basic operations on symbol table.

Learning Outcomes

After successful completion of this assignment, students will be able to

- Implement symbol table.

- Become familiar with compiler working.

Concepts Related Theory:

Symbol Table:

In computer science, a symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier (a.k.a. symbol) in a program's source code is associated with information relating to its declaration or appearance in the source.

A symbol table may only exist during the translation process, or it may be embedded in the output of that process, such as in an ABI object file for later exploitation. For example, it might be used during an interactive debugging session, or as a resource for formatting a diagnostic report during or after execution of a program. And used only in compilers mostly.

Symbol table is used to store information related to various entities like as function name, variable name, objects, classes, interfaces, etc. When identifiers are found, they will be entered into a symbol table, which will hold all relevant information about identifiers. Symbol table is type of data structure that captures scope information. One symbol table for each scope is used. It stores all entities in structured form at one place. By using symbol table it checks if variable is declared or not. It is also used for syntax checking.

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the format as *<symbol name, type, attribute>*. For example table has to store information about following variable declaration as *static int interest;* then it should store the entry such as *<interest, int, static>* The attribute clause contains the entries related to the name.

There are two types of symbol tables. Static symbol table and Dynamic symbol table.

Static symbol tables are tree tables. They are implemented when symbols are known in advance and no addition and deletion is allowed.

Dynamic symbol tables are used when symbol are not known in advance and insertion and deletion can be done any time.

Symbol Table Implementation Methods:

- Unordered Array Implementation
- Ordered Array Implementation
- Unordered Or Ordered List

- Binary Search Trees
- Balanced Binary Search Trees
- Hashing

Unordered Array Implementation:

It maintains arrays of keys and values. Instance variables are used to store data. Array *keys[]* holds the keys and *vals[]* holds the values, integer *N* holds the number of entries.

Ordered Array Implementation:

In ordered array implementation, keys are comparable to each other. Ordered array implementation for symbol table used because it provides ordered iteration. Binary search can speed up search.

Ordered or Unordered Linked-list Implementation:

Maintain a linked list with keys and values. Advantage of keeping linked list in order for comparable key, support ordered iterator and cuts search or insert time in half.

Binary Search Tree:

Keep data stored in parent to child format and sorted. Insertion, deletion, etc. operation can be done faster than other compared methods.

Balanced Binary Search Tree:

Space overhead is directly proportional to the number of items in the table. Insertion takes time compared to other methods of implementation.

Hash Table:

We can work faster in hashing methods. Hashing table method used into most compilers. Complexity is O(1) for hashing table.

Review Questions:

1. What method used into symbol table searching?
2. Which method used mostly in symbol table implementation?
3. How key and values are stored into symbol table?
4. Difference between symbol table and hash table?
5. What data is stored into symbol table?