

PUNE INSTITUTE OF COMPUTER TECHNOLOGY
DEPARTMENT OF COMPUTER ENGINEERING
LAB MANUAL
ACADEMIC YEAR: 2020-2021

DEPARTMENT: COMPUTER ENGINEERING

CLASS: S.E.

SEMESTER: II

SUBJECT: Data Structure and Algorithm Laboratory

INDEX OF LAB EXPERIMENTS

Expt. No.	Problem Statement	Revised on
	Group A	
1.	<p>Beginning with an empty binary tree, Construct binary tree by inserting the values in the order given. After constructing a binary tree perform following operations on it-</p> <ul style="list-style-type: none"> • Perform inorder / preorder and post order traversal • Change a tree so that the roles of the left and right pointers are swapped at every node • Find the height of tree • Copy this tree to another [operator=] • Count number of leaves, number of internal nodes. • Erase all nodes in a binary tree. <p>(implement both recursive and non-recursive methods)</p>	14-1-2021
2.	A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.	14-1-2021
3.	Create an inordered threaded binary tree and perform inorder and preorder traversals. Analyze time and space complexity of the algorithm.	14-1-2021
	Group B	
4.	Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client's telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers (use linear probing with replacement and without replacement)	14-1-2021
5.	<p>Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement.</p> <p>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)</p>	14-1-2021

	Group C	
6.	Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.	14-1-2021
7.	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.	14-1-2021
	Group D	
8.	Given sequence $k = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i . Build the Binary search tree that has the least search cost given the access probability for each key.	14-1-2021
9.	A Dictionary stores keywords and its meanings. 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.	14-1-2021
	Group E	
10.	Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language OR 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. Analyze the algorithm.	14-1-2021
	Group F	
11.	Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.	14-1-2021
12.	Implementation of a direct access file -Insertion and deletion of a record from a direct access file	14-1-2021
	Mini-Projects/ Case Study using Python / Java / C++	
13.	Design a mini project using JAVA which will use the different data structure with or without Java collection library and show the use of specific data structure on the efficiency (performance) of the code.	14-1-2021

	<p>OR</p> <p>Design a mini project to implement Snake and Ladders Game using Python.</p> <p>OR</p> <p>Design a mini project to implement a Smart text editor.</p> <p>OR</p> <p>Design a mini project for automated Term work assessment of student based on parameters like daily attendance, Unit Test /Prelimperformance, Students achievements if any, Mock Practical.</p>	
--	---	--

Prof. Pujashree Vidap

Subject Coordinator

Prof. M.S. Takalikar

HOCD

ASSINGMENT NO.	1
TITLE	Binary tree
PROBLEM STATEMENT /DEFINITION	<p>Beginning with an empty binary tree, Construct binary tree by inserting the values in the order given. After constructing a binary tree perform following operations on it-</p> <ul style="list-style-type: none"> • Perform in order / preorder and post order traversal • Change a tree so that the roles of the left and right pointers are swapped at every node • Find the height of tree • Copy this tree to another [operator=] • Count number of leaves, number of internal nodes. • Erase all nodes in a binary tree. <p>(implement both recursive and non-recursive methods)</p>
OBJECTIVE	To understand binary tree data structure and its operation along with implementation
OUTCOME	After successful completion of this assignment, students will be able to understand and implement binary tree data structure with operations.
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • (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	<ol style="list-style-type: none"> 1. E. Horowitz S. Sahani, D. Mehata, “Fundamentals of data structures in C++”, Galgotia Book Source, New Delhi, 1995, ISBN: 1678298 2. Sartaj Sahani, —Data Structures, Algorithms andApplications in C++ , Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR WRITING JOURNAL	<ol style="list-style-type: none"> 1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Software / hardware 7. Concepts related Theory 8. Algorithm 9. Flowchart 10. Test cases 11. Conclusion/Analysis 12 Real time applications of data structure

Prerequisites:

- Basic knowledge of linked list, searching
- Linked list node deletion, addition, updating

- Object oriented programming features

Learning Objectives:

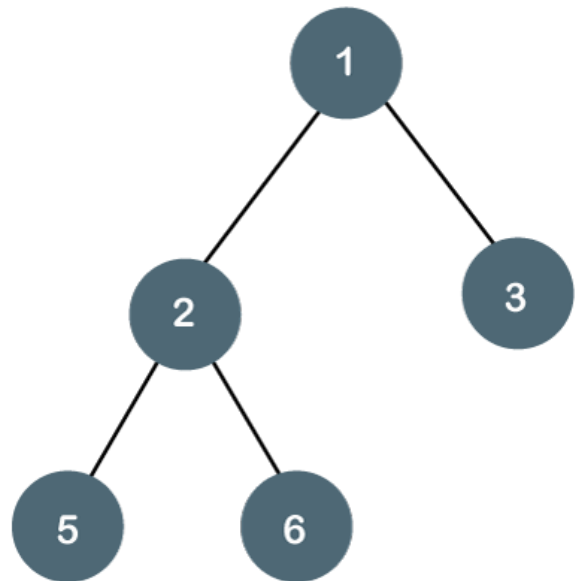
To understand binary tree data structure and its operation along with implementation

Learning Outcomes

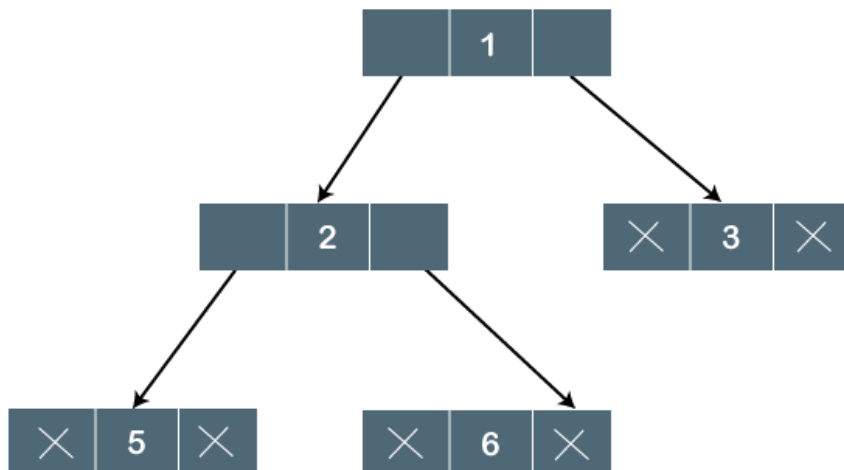
After successful completion of this assignment, students will be able to understand and implement binary tree data structure with operations.

Theory related to assignment:**Binary Tree**

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have 0, 1 or 2 children.

**Let's understand the binary tree through an example.**

The above tree is a binary tree because each node contains the at most two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i , the maximum number of nodes is 2^i .
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to $(1+2+4+8) = 15$. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **$h+1$** .

Subtree: any node in a tree and its descendants.

- **Depth of a node:** the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree:** the maximum depth of any of its leaves.
- **Height of a node:** the length of the longest downward path to a leaf from that node.
- **Full binary tree:** every leaf has the same depth and every nonleaf has two children.
- **Complete binary tree:** every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- **Traversal:** an organized way to visit every member in the structure.

The minimum height can be computed as:

As we know that,

$$n = 2^{h+1} - 1$$

$$n+1 = 2^{h+1}$$

Taking log on both the sides,

$$\log_2(n+1) = \log_2(2^{h+1})$$

$$\log_2(n+1) = h+1$$

$$h = \log_2(n+1) - 1$$

The maximum height can be computed as:

As we know that,

$$n = h+1$$

$$h = n-1$$

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

```
struct node
{
    int data,
    struct node *left, *right;
}
```

Or

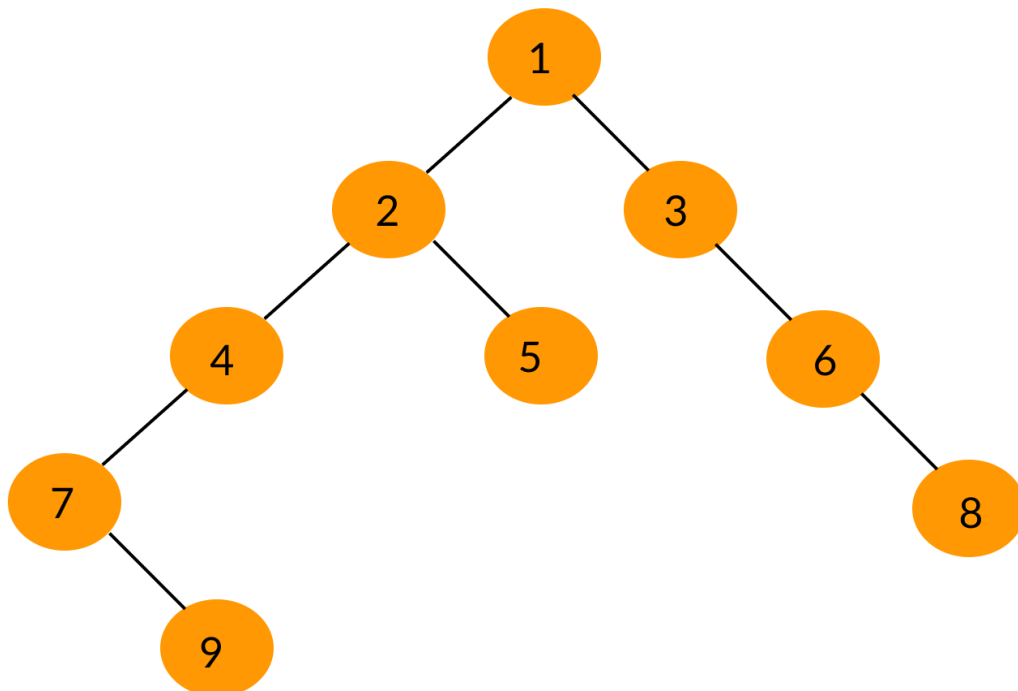
```
class Node{
    int data,
    Node *left, *right;
}
```

In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.

Binary tree traversal can be done in the following ways.

- Inorder traversal
- Preorder traversal
- Postorder traversal

Consider the given binary tree,



Inorder Traversal: 7 9 4 2 5 1 3 6 8

Preorder Traversal: 1 2 4 7 9 5 3 6 8

Postorder Traversal: 9 7 4 5 2 8 6 3 1

Inorder Traversal: For binary search trees (BST), Inorder Traversal specifies the nodes in non-descending order. In order to obtain nodes from BST in non-increasing order, a variation of inorder traversal may be used where inorder traversal is reversed.

Preorder Traversal: Preorder traversal will create a copy of the tree. Preorder Traversal is also used to get the prefix expression of an expression.

Postorder Traversal: Postorder traversal is used to get the postfix expression of an expression given

Algorithms:

Algorithm for binary tree traversal

Inorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Visit and print the root node.
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Preorder(root)

- Visit and print the root node.
- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).

Postorder(root)

- Traverse the left sub-tree, (recursively call inorder(root -> left).
- Traverse the right sub-tree, (recursively call inorder(root -> right).
- Visit and print the root node.

Algorithms:

/* Function to create a new node */

```
struct node *newNode(int data)
```

```
{
```

```
struct node *temp = (struct node *) malloc(sizeof(struct node));
```

```
temp -> data = data;
```

```
temp -> left = NULL;
```



```
temp -> right = NULL;
```

```
return temp;
```

```
};
```

```
/* Function to insert a node in the tree */
```

```
void insert_node(struct node *root, int n1, int n2, char lr)
```

```
{
```

```
if(root == NULL)
```

```
return;
```

```
if(root -> data == n1)
```

```
{
```

```
switch(lr)
```

```
{
```

```
case 'l' : root -> left = newNode(n2);
```

```
break;
```

```
case 'r' : root -> right = newNode(n2);
```

```
break;
```

```
}
```

```
}
```

```
else
```

```
{
```

```
insert_node(root -> left, n1, n2, lr);
```

```
insert_node(root -> right, n1, n2, lr);
```

```
}
```

```
}
```

```
/* Function to print the inorder traversal of the tree */
```

```
void inorder(struct node *root)
{
if(root == NULL)
return;
inorder(root -> left);
cout << root -> data << " ";
inorder(root -> right);
}
```

/* Function to print the preorder traversal of the tree */

```
void preorder(struct node *root)
{
if(root == NULL)
return;
cout << root -> data << " ";
preorder(root -> left);
preorder(root -> right);
}
```

/* Function to print the postorder traversal of the tree */

```
void postorder(struct node *root)
{
if(root == NULL)
return;
postorder(root -> left);
postorder(root -> right);
cout << root -> data << " ";
}
```

}

Pseudo code using non recursive approach:

Algorithm inordertraversal()

{

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

4.2.1 set ptr=ptr->right

4.2.2 goto step 2

4.3 Set ptr=stack[top], top=top-1

}

Algorithm preodertraversal()

{

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
}

```

Operations–

In a binary tree, a node can have maximum two children. Consider the left skewed binary tree shown in Figure



- **Searching:** For searching element 2, we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of $O(n)$.

- **Insertion:** For inserting element as left child of 2, we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of $O(n)$.
- **Deletion:** For deletion of element 2, we have to traverse all elements to find 2 (assuming we do breadth first traversal). Therefore, deletion in binary tree has worst case complexity of $O(n)$.

A node is a leaf node if both left and right child nodes of it are NULL.

Algorithm to get the leaf node count:

Algorithm getLeafCount(node) {

 If node is NULL then return 0.

 Else If left and right child nodes are NULL return 1.

 Else recursively calculate leaf count of the tree using below formula.

 Leaf count of a tree = Leaf count of left subtree + Leaf count of right subtree

}

Conclusion: After successfully completing this assignment, Students have learned construction of binary tree and various operations on 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 binary tree. Explain its common uses.
5. Explain implementation of traversal of a binary tree.

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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Dictionary implementation using Binary Search Tree • Various operations performed on tree
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • 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	<ol style="list-style-type: none"> 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 –

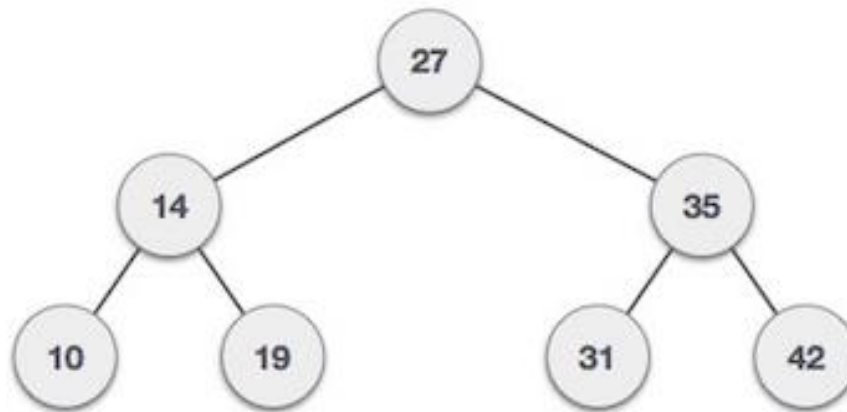
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

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:

```

SearchBST (root, key, LOC, PAR ){
//key is the value to be searched.
//This procedure find the location LOC of key and also the location PAR of the parent of the key.
    1. If ( root == NULL) Then
        1.1 Print ("Tree does not exist")
        1.2 LOC = NULL and PAR = NULL
        1.3 exit
    2. PAR = NULL, LOC = NULL
    3. ptr = root
    4. While ( ptr != NULL )
        4.1 if ( key = ptr -> info ) then
            4.1.1 LOC = ptr
            4.1.2 print PAR AND LOC
            4.1.3 exit
        4.1 else if ( key < ptr -> info ) then
            4.1.1 PAR = ptr
            4.1.2 ptr = ptr -> left
        4.1 else
            4.1.1 PAR = ptr
            4.1.2 ptr = ptr -> right
    5. If ( LOC = NULL ) then
        5.1 Print ("Key not found")
}

```

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:

```

Algorithm InsertBST (root, key, LOC, PAR) { // key is the value to be inserted.
    1. call SearchBST (root, key, LOC , PAR) // To find the parent PAR of the new node
    2. If ( LOC != NULL)
        2.1 Print " Node already exist"
        2.2 Exit
    3. create a new node [new1 = new Node()]
    4. new1 -> info = key

```



```

5. new1 -> left = NULL , new1 -> right = NULL
6. If ( PAR = NULL ) Then
    6.1 root = new1
    6.2 exit
elseif ( new1 -> info < PAR -> info)
    6.1 PAR -> left = new1
    6.2 exit
else
    6.1 PAR -> right = new1
    6.2 exit
}

```

Deletion:

- Delete operation in a binary search tree refers to the process of deleting the specified node from the tree.
- Before implementing a delete operation, you first need to locate the position of the node to be deleted and its parent.
- To locate the position of the node to be deleted and its parent, you need to implement a search operation.
- Once the nodes are located, there can be three cases:
 - Case I: Node to be deleted is the leaf node
 - Case II: Node to be deleted has one child (left or right)
 - Case III: Node to be deleted has two children

Algorithm:

Delete1BST (key, root, LOC, PAR) { // When leaf node has no child or only one child

```

1. if ( ( LOC -> left = NULL) and ( LOC -> right = NULL))
    1.1 Child = NULL
else If ( LOC -> left != NULL)
    1.1 Child = LOC -> left
else
    1.1 Child = LOC -> right
2. if ( PAR != NULL)
    2.1 if ( LOC = PAR -> left )
        2.1.1 PAR -> left = Child
    2.1 else
        2.1.1 PAR -> right = Child
    else
        2.1 root = Child
}

```

Algorithm Delete2BST (key, root, LOC, PAR) { // When node has both child

```

1. ptr1 = LOC
2. ptr2 = LOC -> right
3. While ( ptr2 -> left != NULL )
    3.1 ptr1 = ptr2
    3.2 ptr2 = ptr2 -> left
4. call Delete1BST (info,root, ptr2, ptr1)
5. If ( PAR != NULL)
    5.1 If LOC = PAR -> left

```

```

                                4.1.1 PAR -> left = ptr2
                    5.1      else
                                4.1.1 PAR -> right = ptr2
                else
                    5.1 root = ptr2
        6. ptr2 -> left = LOC -> left
        7. ptr2 -> right = LOC -> right
    }

```

Algorithm DeleteBST (root, key)

```

{
    // key is the value to be deleted.
    1. call SearchBST (root, key, LOC, PAR )
    // To find the location LOC and parent PAR of the node to be deleted.
    2. If ( LOC = NULL ) Then
        2.1 Print “Node does not exist”
        2.2 exit
    3. if ( ( LOC -> left != NULL) and ( LOC -> right != NULL))
        // when the node to be deleted has both child
        3.1 call Delete2BST (key, root, LOC, PAR)
    else
        3.1 call Delete1BST (key, root, LOC, PAR)
}

```

Review Questions:

1. What is binary search tree?
2. How do you find the depth of a binary tree?
3. Explain pre-order and in-order tree traversal?
4. Explain implementation of deletion from a binary search tree.

ASSINGMENT NO.	3
TITLE	Threaded binary tree
PROBLEM STATEMENT /DEFINITION	Convert given binary tree into inorder and preorder threaded binary tree. Analyze time and space complexity of the algorithm.
OBJECTIVE	To understand construction of inorder and preorder Threaded binary tree from given binary tree.
OUTCOME	At the end of this assignment students will be able to construct threaded binary tree and able to perform basic operations on Threaded binary tree.
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • (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 framework, TC++, GTK++.
REFERENCES	<ul style="list-style-type: none"> • 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++I, Second Edition, University Press, ISBN:81-7371522 X.
INSTRUCTIONS FOR WRITING JOURNAL	<ol style="list-style-type: none"> 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:

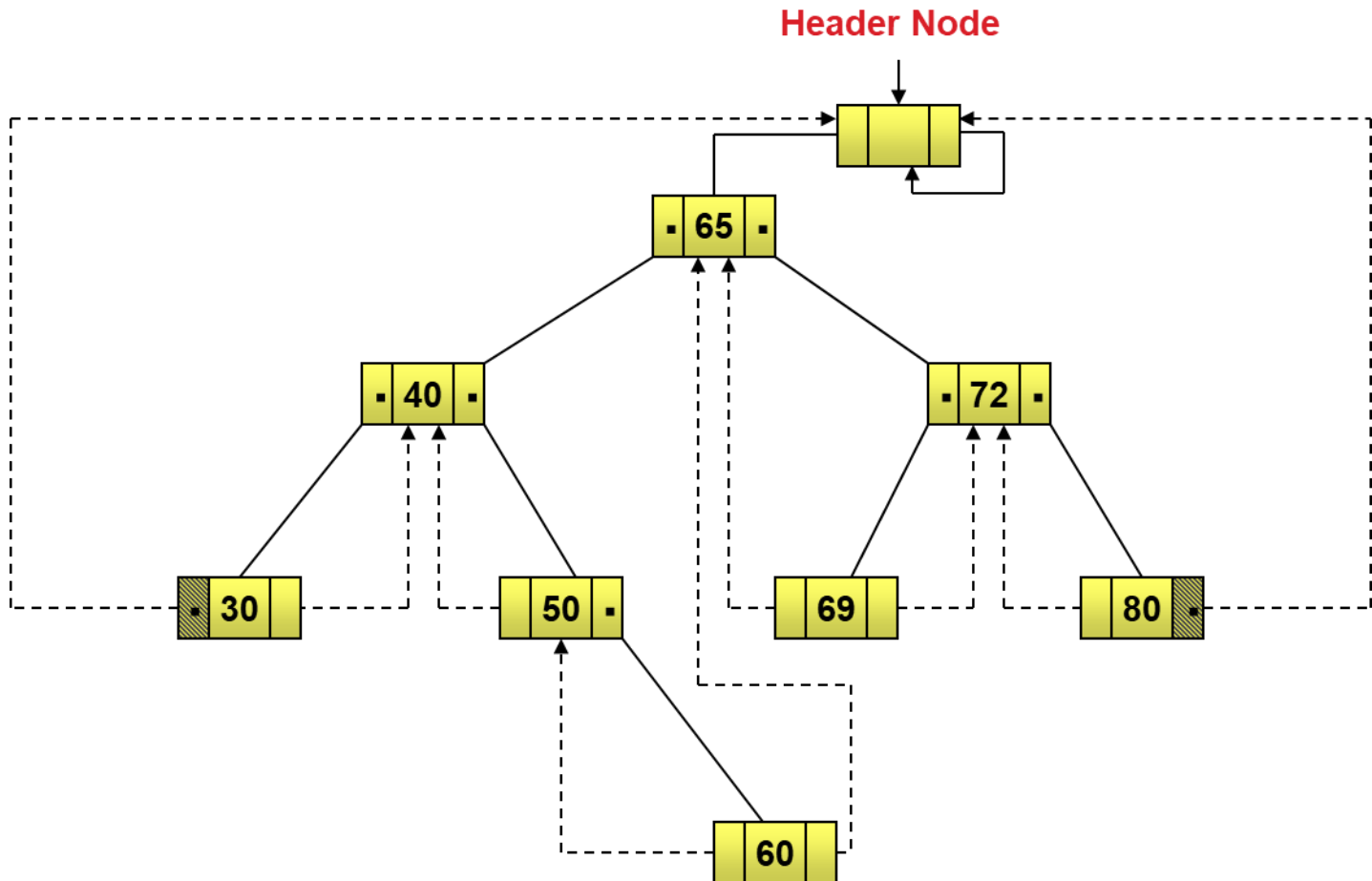
Single threaded: Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor or all left null pointers will point to inorder predecessor.

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.



Representing a Threaded Binary Tree

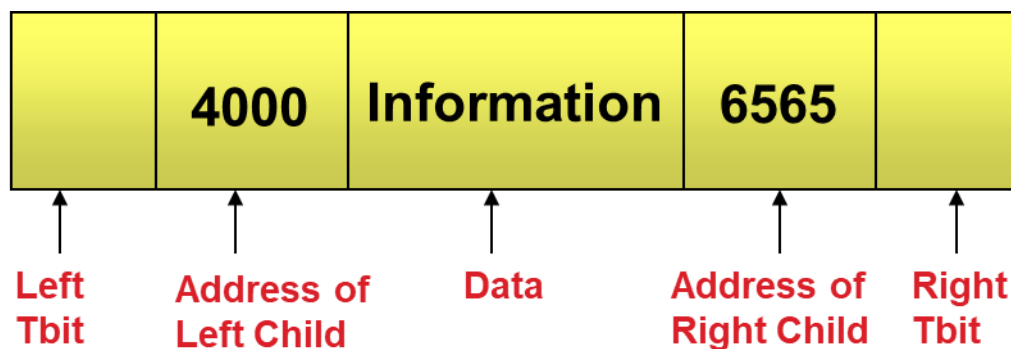
The structure of a node in a threaded binary tree is a bit different from that of a normal binary tree.

Unlike a normal binary tree, each node of a threaded binary tree contains two extra pieces of information, namely left thread and right thread.

The left and right thread fields of a node can have two values:

1: Indicates a normal link to the child node

0: Indicates a thread pointing to the inorder predecessor or inorder successor



Algorithms:

Algorithm InsertInTBT(key){

1. If(root==NULL) {

header=new Node(-99)

header->lbit=0

header->rbit=0

header->right=header

root=new Node(key)

root->lbit=0

root->rbit=0

root->left=header

root->right=header

header->left=root

}

else {

ptr=root

temp=new Node(key)

while(1) {

If(ptr->data>key){

 If(ptr->lbit != 0)

 ptr=ptr->left

else {

 temp->lbit=0

 temp->rbit=0

 temp->left=ptr->left

 temp->right=ptr

 ptr->lbit=1

 ptr->left=temp

 return

}

} else {

```

        If(ptr->rbit !=0)
            ptr=ptr->right
    else {
        temp->left=ptr
        temp->right=ptr->right
        temp->lbit=temp->rbit=0
        ptr->right=temp
        ptr->rbit=1
        return
    } } }

```

Algorithm inorder(){

```

t=root
while(t->lbit!=0)
    t=t->left
while(t!=header){
    print(t->data)
    if(t->rbit!=0){
        t=t->right
    }
    while(t->lbit!=0)
        t=t->left
}
else
    t=t->right
}
}

```

Algorithm preorder(){

```

t=root
while(t!=header){

```

```

        print(t->data)
        if(t->lbit==1)
            t=t->left
        else
        {
            if(t->rbit==1)
                t=t->right
            else {
                while(t->rbit!=1)
                    t=t->rc
                t=t->rc
            }
        }
    }
}
}
}

```

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

Review Questions:

6. What is binary tree? Explain its uses.
7. How do you find the depth of a binary tree?
8. Explain pre-order and in-order tree traversal.
9. Define threaded binary tree. Explain its common uses.
10. Explain implementation of traversal of a binary tree.
11. Why do we need Threaded binary search trees ?
12. How to create a threaded BST?
13. How to convert a Given Binary Tree to Threaded Binary Tree?
14. What are the drawbacks of bi-threaded trees? Are single threaded trees enough to do traversals on the tree? Justify.
15. Did you require stacks to do get the output along with threads? Justify.
16. Justify that only single threads are required to traverse the tree efficiently.
17. 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	<ol style="list-style-type: none"> 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	<ul style="list-style-type: none"> • (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	<ul style="list-style-type: none"> • 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	<ol style="list-style-type: none"> 1. Date 2. Assignment no. 3. Problem definition 4. Learning objective

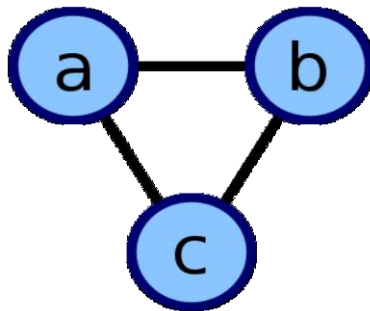
	5. Learning Outcome 6. Concepts related Theory 7. Algorithm 8. Test cases 9. Conclusion/Analysis
--	--

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.

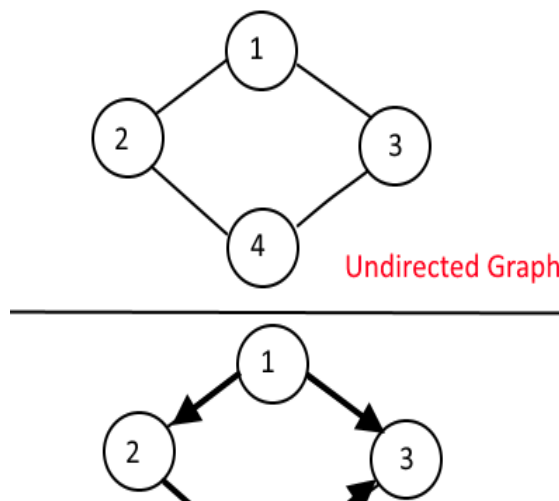
What is Graph ?

$$G = (V, E)$$

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 $V \times V$, where V are the number of vertices in the graph. See the example below, the Adjacency matrix for the graph shown above.

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

Undirected Graph

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

Directed Graph

adjMat

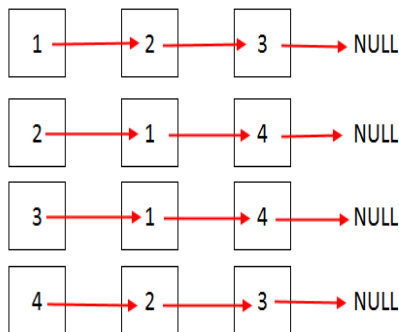
0.

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

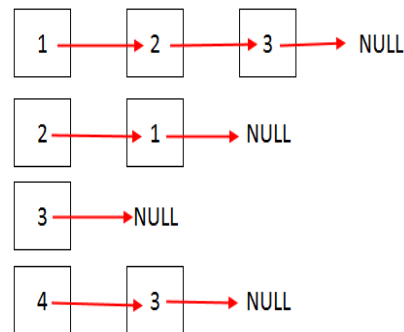
But the drawback is that it takes $O(V^2)$ space even though there are very less edges in the graph.

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.



Adjacency List - Undirected Graph



Directed Graph

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();
}
}
};
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(vptr2);
g.insert(vptr3);
g.insert(vptr4);
t Displaying City Transport Map Using Adjacency List"<<endl;
g.Display();
return 1;
}

```

TEST CASES:

Check 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	<ul style="list-style-type: none"> • To understand concept of graph & minimum cost spanning tree. • To understand different minimum cost spanning tree algorithms. • To implement minimum spanning tree algorithms.
OUTCOME	<ul style="list-style-type: none"> • Graph implementation using Adjacency matrix or Adjacency list • Find total minimum cost using MST Algorithm
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • 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	<ol style="list-style-type: none"> 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

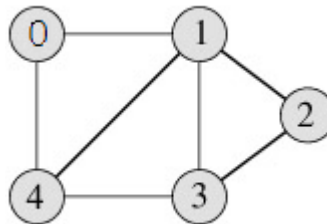
After successful completion of this assignment, students will be able to

- 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 $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[i][j]$, 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.

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.  $T \leftarrow \{v\}$  //the nodes in the MST
3.  $E \leftarrow \{\}$  //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
     $\{\text{weight}(u,w) : w \text{ in } G - T \text{ and } u \text{ in } T\}$ 
     $T \leftarrow T \cup \{v'\}$ 
     $E \leftarrow E \cup \{(v,v')\}$ 
5. return E
```

Kruskal's Algorithm:

```
// input: a graph G with n nodes and m edges
// output: E: a MST for G
•  $EG[1..m] \leftarrow$  Sort the m edges in G in increasing weight order
•  $E \leftarrow \{\}$  //the edges in the MST
•  $i \leftarrow 1$  //counter for EG
• While  $|E| < n-1$  do
    if adding  $EG[i]$  to E does not add a cycle then
         $E \leftarrow E \cup \{EG[i]\}$ 
         $i \leftarrow i + 1$ 
• 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	<ul style="list-style-type: none"> • (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	<ul style="list-style-type: none"> • 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	<ol style="list-style-type: none"> 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:

1. Computing the so-called hash function $H: K \rightarrow A$.
2. 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	<p>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:</p> <ol style="list-style-type: none"> (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	<ol style="list-style-type: none"> 1) To understand concept of symbol table. 2) Why symbol table is needed.
OUTCOME	<ol style="list-style-type: none"> 1) Use of symbol table 2) Various methods of implementing symbol table.
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • 64-bit Open source Linux or its derivative. • Open Source C++ Programming tool like G++/GCC.
REFERENCES	<ul style="list-style-type: none"> • Data structures in C++ by Horowitz, Sahni.
INSTRUCTIONS FOR WRITING JOURNAL	<ol style="list-style-type: none"> 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?

ASSINGMENT NO.	9
TITLE	To write a program for implementing dictionary using Height balance tree.
PROBLEM STATEMENT /DEFINITION	A Dictionary stores keywords & its meanings. 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	<ol style="list-style-type: none"> 1. To understand concept of height balanced tree data structure. 2. To understand procedure to create height balanced tree.
OUTCOME	<ul style="list-style-type: none"> • Define class for AVL using Object Oriented features. • Analyze working of various operations on AVL Tree.
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • 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	<ol style="list-style-type: none"> 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

Assignment 9:

Aim: To write a program for Dictionary implementation using height balanced tree.

Prerequisites:

- ✓ **Learning Objectives:** To understand concept of height balanced tree data structure.
- ✓ To understand procedure to create height balanced tree.

Learning Outcomes

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

- ☐ Define class for AVL using Object Oriented features.
- ☐ Analyze working of various operations on AVL Tree.

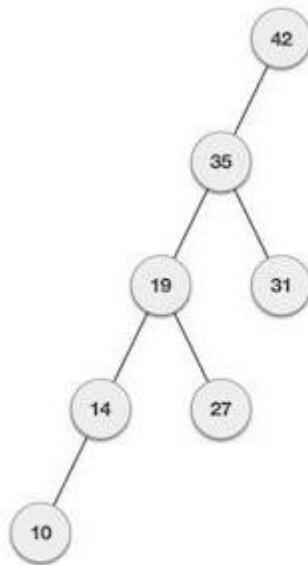
Concepts related Theory:

An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

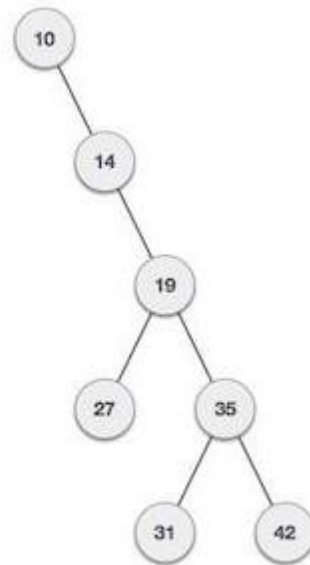
AVL (Adelson- Velskii and Landis) Tree: A balance binary search tree. The best search time, that is $O(\log N)$ search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1.

Representation:

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



If input 'appears' non-increasing manner

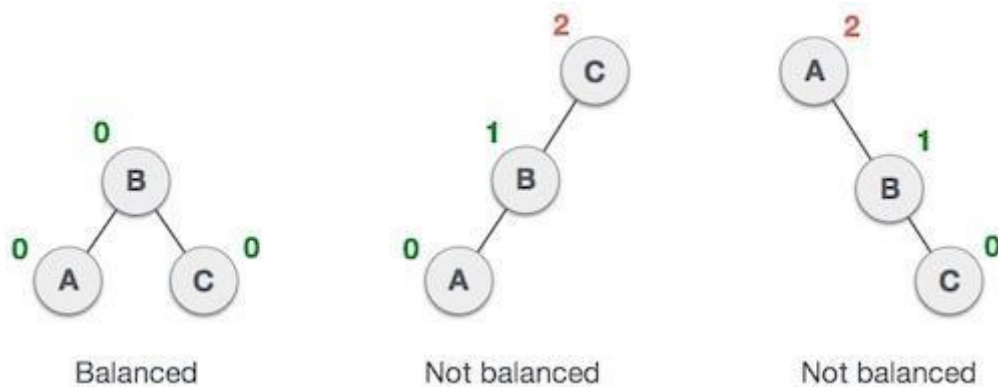


If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is $O(n)$. In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski & Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

$$\text{BalanceFactor} = \text{height}(\text{left-sutree}) - \text{height}(\text{right-sutree})$$

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

Basic Operations:

AVL Rotations

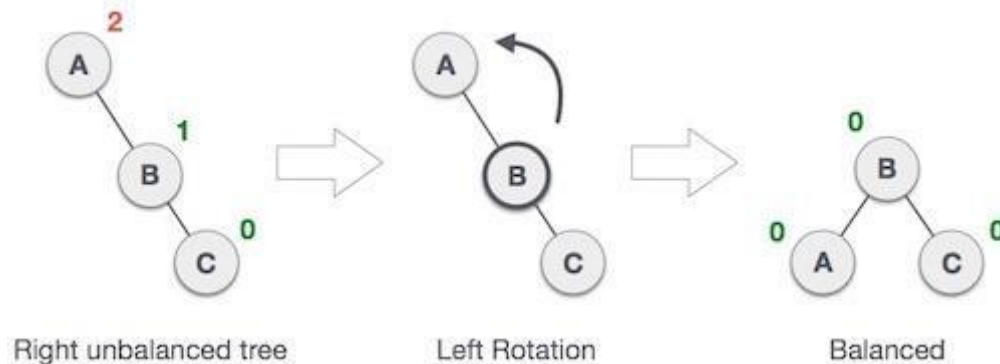
To balance itself, an AVL tree may perform the following four kinds of rotations –

- ☐ Left rotation
- ☐ Right rotation
- ☐ Left-Right rotation
- ☐ Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

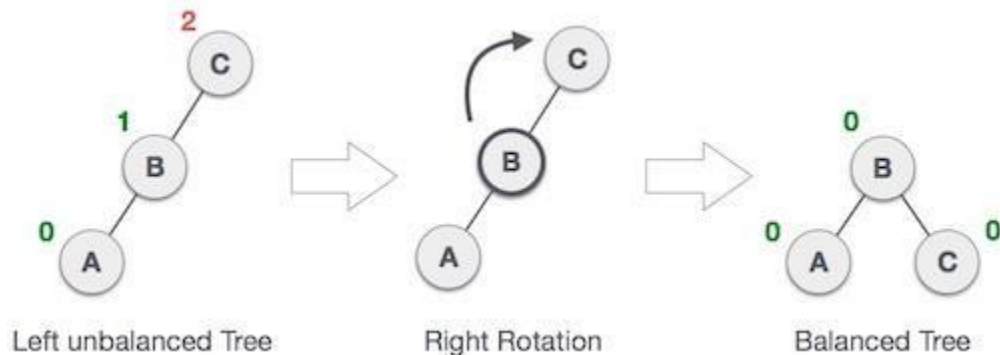
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

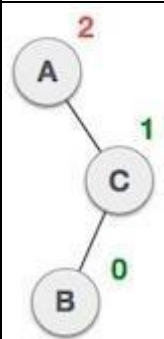
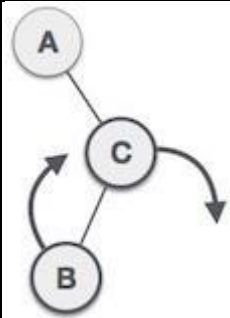
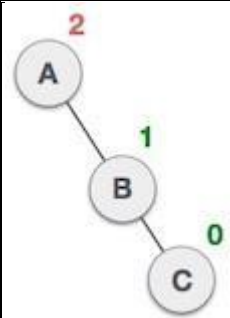
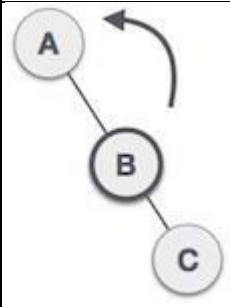
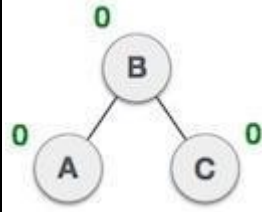
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.</p>
	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>
	<p>First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.</p>
	<p>Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.</p>
	<p>The tree is now balanced.</p>

Representation: Basic Operations:

Algorithm:

Insert:-

1. If P is NULL, then
 - I. P = new node
 - II. P ->element = x
 - III. P ->left = NULL
 - IV. P ->right = NULL
 - V. P ->height = 0
2. else if $x > P \rightarrow \text{element}$
 - a.) insert(x, P ->left)
 - b.) if height of P ->left - height of P ->right = 2
 1. insert(x, P ->left)
 2. if height(P ->left) - height(P ->right) = 2
if $x < P \rightarrow \text{left} \rightarrow \text{element}$
P = singlerotateleft(P)

else

P = doublerotateleft(P)
3. else
if $x < P \rightarrow \text{element}$
 - a.) insert(x, P -> right)
 - b.) if height (P -> right) - height (P ->left) = 2
if $(x < P \rightarrow \text{right}) \rightarrow \text{element}$
P = singlerotateright(P)

else

P = doublerotateright(P)
4. else
Print already exists
5. int m, n, d.

6. $m = \text{AVL height}(P \rightarrow \text{left})$
7. $n = \text{AVL height}(P \rightarrow \text{right})$
8. $d = \max(m, n)$
9. $P \rightarrow \text{height} = d + 1$
10. Stop

`RotateWithLeftChild(AvlNode k2)`

- `AvlNode k1 = k2.left;`
- `k2.left = k1.right;`
- `k1.right = k2;`
- `k2.height = max(height(k2.left), height(k2.right)) + 1;`
- `k1.height = max(height(k1.left), k2.height) + 1;`
- `return k1;`

`RotateWithRightChild(AvlNode k1)`

- `AvlNode k2 = k1.right;`
- `k1.right = k2.left;`
- `k2.left = k1;`
- `k1.height = max(height(k1.left), height(k1.right)) + 1;`
- `k2.height = max(height(k2.right), k1.height) + 1;`
- `return k2;`

`doubleWithLeftChild(AvlNode k3)`

- `k3.left = rotateWithRightChild(k3.left);`
- `return rotateWithLeftChild(k3);`

`doubleWithRightChild(AvlNode k1)`

- `k1.right = rotateWithLeftChild(k1.right);`
- `return rotateWithRightChild(k1);`

Review Questions:

1. What is AVL tree?
2. In an AVL tree, at what condition the balancing is to be done
3. When would one want to use a balance binary search tree (AVL) rather than an array data structure

ASSINGMENT NO.	10
TITLE	Implement the Heap/Shell sort algorithm
PROBLEM STATEMENT / DEFINITION	Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language.
OBJECTIVE	<ol style="list-style-type: none"> 1. To understand concept of heap in data structure. 2. To understand concept & features of java language.
OUTCOME	<ul style="list-style-type: none"> • Define class for heap using Object Oriented features. • Analyze working of heap sort .
S/W PACKAGES AND HARDWARE APPARATUS USED	<ul style="list-style-type: none"> • 64-bit Open source Linux or its derivative. <p>Open Source C++ Programming tool like G++/GCC.</p>
REFERENCES	Data structures in C++ by Horowitz, Sahni.
INSTRUCTIONS FOR WRITING JOURNAL	<ol style="list-style-type: none"> 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

Assignment 10:

Aim: Implement the Heap/Shell sort algorithm

Prerequisites: you must know about a complete binary tree and heap data structure.

Learning Objectives:

- To understand concept of heap in data structure.
- To understand concept & features of java language.

Learning Outcomes

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

: Understand heap data structure.

: Implementation heap sort

Concepts related Theory:

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I , the left child can be calculated by $2 * I + 1$ and right child by $2 * I + 2$ (assuming the indexing starts at 0).

Basic Operations:

Heap Sort Algorithm for sorting in increasing order:

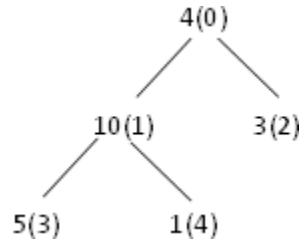
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

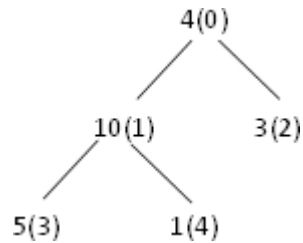
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1



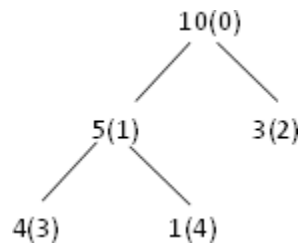
The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:



Applying heapify procedure to index 0:

The heapify procedure calls itself recursively to build heap in top down manner.



Algorithm:

STEP 1: Logically, think the given array as Complete Binary Tree,

STEP 2: For sorting the array in ascending order, check whether the tree is satisfying Max-heap property at each node, (For descending order, Check whether the tree is satisfying Min- heap property) Here we will be sorting in Ascending order,

STEP 3: If the tree is satisfying Max-heap property, then largest item is stored at the root of the heap. (At this point we have found the largest element in array, Now if we place this element at the end(nth position) of the array then 1 item in array is at proper place.)

We will remove the largest element from the heap and put at its proper place(nth position) in array.

After removing the largest element, which element will take its place? We will put last element of the heap at the vacant place. After placing the last element at the root, The new tree formed may or may not satisfy max-heap property. So, If it is not satisfying max-heap property then first task is to make changes to the tree, So that it satisfies max-heap property.

(Heapify process: The process of making changes to tree so that it satisfies max-heap property is called heapify)

When tree satisfies max-heap property, again largest item is stored at the root of the heap. We will remove the largest element from the heap and put at its proper place(n-1 position) in array. Repeat step 3 until size of array is 1 (At this point all elements are sorted.)

Input: Number of elements to be sort and element values.

Output: Elements in sorted order.

Conclusion: This program gives us the knowledge of heap data structure

Review Questions:

- What is a heap in data structure?
- What is shell sort?
- What is min heap and max heap?
- Explain heapify operation?

ASSINGMENT NO.	11
TITLE	Sequential File Operations
PROBLEM STATEMENT /DEFINITION	Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.
OBJECTIVE	To understand the concept and basics of sequential file and its use in Data structure
OUTCOME	At the end of this assignment students will able 1. Understand how sequential files are organized. 2. Understand the processing limitations imposed by an unordered Sequential file 3. To implement the concept and basic of sequential file and to perform basic operation as adding record, display all record, search record from sequential file and its use in Data structure.
S/W PACKAGES AND HARDWARE APPARATUS USED	64-bit Open source Linux or its derivative Open Source C++ Programming tool like G++/GCC Online mode: MS Team
REFERENCES	<ul style="list-style-type: none"> • Sartaj Sahani, “Data Structures, Algorithms and Applications in C++” • Aho, J. Hopcroft, J. Ulman, “Data Structures and Algorithms”
INSTRUCTIONS FOR WRITING JOURNAL	<ul style="list-style-type: none"> • Date & Roll No. • Aim: • Assignment no. • Problem definition • Learning objective • Learning Outcomes • Related Theory • Algorithm/Pseudocode with Analysis • Test Cases • Output • Result

Aim: Use Sequential File to maintain data

Prerequisites:

- Introduction to Sequential files
- basic concepts of data structures.
- Add, delete operations.

Concepts related Theory:

Types of File Organization-

1. Sequential access file organization
2. Indexed sequential access file organization
3. Direct access file organization

Sequential file: A file that contains records or other elements that are stored in a chronological order based on account number or some other identifying data.

Sequential access file organization

- Storing and sorting in contiguous block within files on tape or disk is called as sequential access file organization.
- In sequential access file organization, all records are stored in a sequential order. The records are arranged in the ascending or descending order of a key field.
- Sequential file search starts from the beginning of the file and the records can be added at the end of the file.
- In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.
- A sequential data file is a straight text file, like the type of file you can create with a program like Notepad.

Primitive Operations on Sequential files :

- Open—This opens the file and sets the file pointer to immediately before the first record
- Read-next—This returns the next record to the user. If no record is present, then EOF condition will be set.

- Close—This closes the file and terminates the access to the file.
- Write-next—File pointers are set to next of last record and write the record to the file.
- EOF—If EOF condition occurs, it returns true, otherwise it returns false.
- Search—Search for the record with a given key.
- Update—Current record is written at the same position with updated values.

When you want to write code that uses a data file, you must first import the `oi` package of the `java` language:

Sample Input Set:

Record Type Name	Field name	Data Type
Student	Roll No	Integer
	Name	Character or String
	Division	Character or String
	Address	Character or String

ADT:

```
class Student
{
int rollno;
char name[20],address_city[20];
char div;
int year;
public:
Student()
{
strcpy(name," ");
strcpy(address_city," ");
rollno=year=div=0;
}
}
```

Algorithms:

1. Algorithm for main function MAIN FUNCTION ()
- S1: Read the filenames from user from database.
- S2: Read the operations to be performed from the keyboard
- S3: If the operation specified is create go to the create function,
if the operation specified is display go to the display function,
if the operation specified is add go to the add function,
if the operation specified is delete go to delete function,

if the operation specified is display particular record go to the search function,
if the operation specified is exit go to step 4.

S4: Stop

//File Operations

```
class FileOperations
```

```
{
```

```
fstream file;
```

```
public:
```

```
FileOperations(char* filename)
```

Algorithm to create function

S1: Open the file in the write mode, if the file specified is not found or unable to open then display error message and go to step5, else go to step2.

S2: Read the no: of records N to be inserted to the file.

S3: Repeat the step4 N number of times.

S4: Read the details of each student from the keyboard and write the same to the file.

S5: Close the file.

S6: Return to the main function.

Algorithm to add a record

S1: Open the file in the append mode, if the file specified is not found or unable to open then display error message and go to step5 , else go to step2

S2: Scan all the student details one by one from file until end of file is reached.

S3: Read the details of the form the keyboard and write the same to the file

S4: Close the file.

S5: Return to the main function

```
void insertRecord(int rollno, char name[MAX],int year,  
char div,char city[MAX])
```

```
{
```

```
Student s1(rollno,name,year,div,city);
```

```
file.seekp(0,ios::end);
```

```
file.write((char *)&s1,sizeof(Student));
```

```
file.clear();
```

```
}
```

Algorithm for deleting a record

S1: Open the file in the append mode, if the file specified is not found or unable to open then display error message and go to step5, else go to step2

S2: Accept the roll no from the user to delete the record

S3: Search for the roll no in file. If roll no. exists, copy all the records in the file except the one to be deleted in another temporary file.

S4: Close both files

S5: Now, remove the old file & name the temporary file with name same as that of old file name.

```
void deleteRecord(int rollno)
{
    ofstream outFile("new.dat",ios::binary);
    file.seekg(0,ios::beg);
    bool flag=false;
    Student s1;
    while(file.read((char *)&s1, sizeof(Student)))
    {
        if(s1.getRollNo()==rollno)
        {
            flag=true;
            continue;
        }
        outFile.write((char *)&s1, sizeof(Student));
    }
    if(!flag)
    {
        cout<<"\nRecord of "<<rollno<<" is not present.";
    }
}
```

Algorithm for displaying particular record (search)

S1: Open the file in the read mode, if the file specified is not found or unable to open then display error message and go to step6, else go to step2.

S2: Read the roll number of the student whose details need to be displayed.

S3: Read each student record from the file.

S4: Compare the students roll number scanned from file with roll number specified by the user.

S5: If they are equal then display the details of that record else display required roll number not found message and go to step6.

S6: Close the file.

S7: Return to the main function.

```
while(file.read((char *)&s1,sizeof(Student)))
{
    if(s1.getRollNo()==rollNo)
    {
        s1.displayRecord();
        flag=true;
        break;
    }
}
```

```

}
if(flag==false)
{
cout<<"\nRecord of "<<rollNo<<"is not present.";
}
file.clear();
}

```

Test Conditions:

1. Input valid filename.
2. Input valid record.
3. Check for opening / closing / reading / writing file errors

Test Cases

Test Case No	Description	Input	Expected O/P	Actual O/P	Result
1	Insert Record	Roll No: 21145 Name: Sri Division: IV Address: Pune	Record Inserted Successfully	Record Inserted Successfully	Pass
2	Search Record	41424	Record of 41424 not present	Record of 41424 not present	Pass

Result: We implemented a program for sequential file operations using C++.

Review Questions:

1. Define File? What are the factors affecting the file organization?
2. Compare Text and Binary File?
3. Explain the different File opening modes in C++?
4. What is indexed sequential file? Explain the primitive operations on indexed sequential file.
5. Write a note on Direct Access File?
6. What is sequential data?
7. What is a sequential access storage device?
8. How to add record in sequential file?
9. How to delete record from sequential file?
10. What are the advantages and disadvantages of indexed-sequential file organization?

11. What are the advantages of Sequential File Organization?
12. What are serial and sequential files and how are they used in organization?
13. Distinguish between logical and physical deletion of records and illustrate it with suitable examples.
14. Compare and contrast sequential file and random access file organization?
15. Explain the different types of external storage devices?
16. With the prototype and example, explain following functions:
17. seekg() ii) tellp() iii) seekp() iv) tellp()
18. What is the sequential file?
19. What is sequential data?
20. What is a sequential access storage device?
21. How to add record in sequential file?
22. How to add record in sequential file?

ASSINGMENT NO.	12
TITLE	Direct Access File Operations
PROBLEM STATEMENT /DEFINITION	Implementation of a direct access file -Insertion and deletion of a record from a direct access file
OBJECTIVE	<ul style="list-style-type: none"> To understand & implement direct file access organization
OUTCOME	<p>After completion of this assignment students will be able to</p> <ul style="list-style-type: none"> contain student record in Sequential file and also Perform add, delete operation successfully and system will be displays student records.
S/W PACKAGES AND HARDWARE APPARATUS USED	<p>64-bit Open source Linux or its derivative</p> <p>Open Source C++ Programming tool like G++/GCC</p> <p>Online mode: MS Team</p>
REFERENCES	<ul style="list-style-type: none"> Sartaj Sahani, “Data Structures, Algorithms and Applications in C++” Aho, J. Hopcroft, J. Ulman, “Data Structures and Algorithms”
INSTRUCTIONS FOR WRITING JOURNAL	<ul style="list-style-type: none"> Date & Roll No. Aim: Assignment no. Problem definition Learning objective Learning Outcomes Related Theory Algorithm/Pseudocode with Analysis Test Cases Output Result

Prerequisites:

- Introduction to direct access files
- Basic concepts of data structures.
- Add, delete operations.

Concepts related Theory:

- Direct access file is also known as random access or relative file organization.
- In direct access file, all records are stored in direct access storage device (DASD), such as hard disk. The records are randomly placed throughout the file.
- The records do not need to be in sequence because they are updated directly and rewritten back in the same location.
- This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.
- It is also called as hashing.

When using direct access method, the record occurrences in a file do not have to be arranged in any particular sequence on storage media.

In the direct access method, an algorithm is used to compute the address of a record.

Advantages:

- Transactions need not be sorted.
- Different discs or disc units are not required for updating records as existing records may be amended by overwriting.
- It is also possible to process direct file records sequentially in a record key sequence.
- A direct file organization is most suitable for interactive on line applications such as air line or railway reservation systems, teller facility in banking application, etc.
- Immediate access to records for updating purposes is possible.
- Random inquiries which are too frequent in business situations can be easily handled.

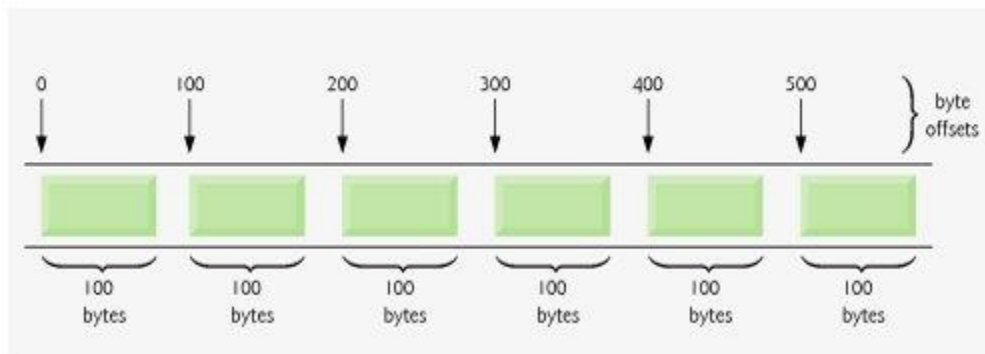
Drawbacks:

May be less efficient in the use of storage space than sequentially organized file.

- Expensive hardware and software resources are required.
- Relative complexity of programming.
- System design around it is complex and costly.
- Data may be accidentally erased or even written unless special precautions are taken.
- Special security measures are necessary for on line direct files that are accessible from several stations.

- File updating (addition and deletion records) is more difficult as compared to sequential files.
- Records in the on line may be expressed to the risks of loss of accuracy and a breach of security Special back up and reconstruction procedures must be established.

Representation: Figure below illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long). A random-access file is like a railroad train with many same-size cars some empty and some with contents.



Basic Operations:

The ostream member function write outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream. When the stream is associated with a file, function write writes the data at the location in the file specified by the "put" file-position pointer. The istream member function read inputs a fixed number of bytes from the specified stream to an area in memory beginning at a specified address. If the stream is associated with a file, function read inputs bytes at the location in the file specified by the "get" file-position pointer.

Writing Bytes with ostream Member Function write

When writing an integer number to a file, instead of using the statement

```
outFile << number;
```

which for a four-byte integer could print as few digits as one or as many as 11 (10 digits plus a sign, each requiring a single byte of storage), we can use the statement

```
outFile.write( reinterpret_cast< const char * >( &number ),
sizeof( number ) );
```

Algorithm: Simple example:

```
class ClientData
{
public:
// default ClientData constructor
ClientData( int = 0, string = "", string = "", double = 0.0 );

//accessor functions for accountNumber
void setAccountNumber( int );
int getAccountNumber() const;

// accessor functions for lastName
void setLastName( string );
string getLastName() const;

// accessor functions for firstName
void setFirstName( string );
string getFirstName() const;

// accessor functions for balance
void setBalance( double );
double getBalance() const;
private:
int accountNumber;
char lastName[ 15 ];
char firstName[ 10 ];
double balance;
}; // end class ClientData

#endif

// Class ClientData stores customer's credit information.
#include
using std::string;
#include "ClientData.h"
// default ClientData constructor
ClientData::ClientData( int accountNumberValue,
string lastNameValue, string firstNameValue, double balanceValue )
{
setAccountNumber( accountNumberValue );
setLastName( lastNameValue );
```

```

setFirstName( firstNameValue );
setBalance( balanceValue );
} // end ClientData constructor

// get account-number value
int ClientData::getAccountNumber() const
{
return accountNumber;
} // end function getAccountNumber

// set account-number value
void ClientData::setAccountNumber( int accountNumberValue )
{
accountNumber = accountNumberValue; // should validate
} // end function setAccountNumber

// get last-name value
string ClientData::getLastName() const
{
return lastName;
} // end function getLastName

// set last-name value
void ClientData::setLastName( string lastNameString )
{
// copy at most 15 characters from string to lastName
const char *lastNameValue = lastNameString.data();
int length = lastNameString.size();
length = ( length < 15 ? length : 14 );
strncpy( lastName, lastNameValue, length );
lastName[ length ] = '\0'; // append null character to lastName
} // end function setLastName

// get first-name value
string ClientData::getFirstName() const
{
return firstName;
} // end function getFirstName
// set first-name value
void ClientData::setFirstName( string firstNameString )
{
// copy at most 10 characters from string to firstName
const char *firstNameValue = firstNameString.data();
int length = firstNameString.size();
length = ( length < 10 ? length : 9 );
strncpy( firstName, firstNameValue, length );
firstName[ length ] = '\0'; // append null character to firstName
} // end function setFirstName
// get balance value

```

```

double ClientData::getBalance() const
{
    return balance;
} // end function getBalance

// set balance value
void ClientData::setBalance( double balanceValue )
{
    balance = balanceValue;
} // end function setBalance

#include
using std::cerr;
using std::endl;
using std::ios;

#include
using std::ofstream;

#include
using std::exit; // exit function prototype

#include "ClientData.h" // ClientData class definition

int main()
{
    ofstream outCredit( "credit.dat", ios::binary );

    // exit program if ofstream could not open file
    if ( !outCredit )
    {
        cerr << "File could not be opened." << endl;
        exit( 1 );
    } // end if

    ClientData blankClient; // constructor zeros out each data member

    // output 100 blank records to file
    for ( int i = 0; i < 100; i++ )
        outCredit.write( reinterpret_cast< const char * >( &blankClient ),
            sizeof( ClientData ) );

    return 0;
} // end main

```

Review Questions:

1. What is the sequential file?

2. Compare the features of sequential file, index sequential file & direct access file.
3. What is a sequential access storage device?
4. How to add record in sequential file?
5. What are primitive operations on sequential file?