

JSPM‟s

**Imperial College of Engineering & Research, Wagholi,**

**Pune.**

Department Of Computer Engineering

**Lab Manual**

**Data Structure & Algorithms Laboratory (DSAL)**

Subject Code: 210256

Class: - SE (2019 PAT)

Prepared by: - Prof.S.R.Bhandari

Branch: - Computer Engg.

Examinations: - PR [25M] and TW [25M]

Required H/W and S/W: - 64 bit Open Source Linux 19. Programming Tool: Open Source C++ programming tool

(C++/GCC).

**INDEX**

|  |  |  |
| --- | --- | --- |
| Assig.  No | Practical | Page  No. |
| 1 | Consider telephone book database of N clients. Make use of a hash table  implementation to quickly look up client„s telephone number. | 4 |
| 2 | 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, and Keys must be unique Standard Operations: Insert (key, value), Find (key), Delete (key) | 11 |
| 3 | A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements  of your method. | 16 |
| 4 | Construct an expression tree from the given prefix expression  eg. +--a\*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree. | 19 |
| 5 | 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. | 22 |
| 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. | 32 |
| 7 | There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.  Check whether the graph is connected or not. Justify the storage representation used. | 35 |
| 8 | Given sequence k = k1 <k2 < … < kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key? | 38 |
| 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 | 42 |
| 10 | Consider a scenario for Hospital to cater services to different kinds of patients as Serious (top priority), b) non-serious (medium priority), c) General Checkup  (Least priority). Implement the priority queue to cater services to the patients. | 51 |

|  |  |  |
| --- | --- | --- |
| 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. | 53 |
| 12 | Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index  sequential file to maintain the data. | 58 |

# Problem Statement:

**GROUP A**

**Assignment 01**

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client„s telephone number

# Objectives:

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

# Learning Objectives

To understand concept of hashing.

To understand operations like insert and search record in the database. To understand the collision handling technique.

# Learning Outcome

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

# Theory:

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

# Keyed Arrays vs. Indexed Arrays

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

1employees[50];

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

1employees["Brown, John"];

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

# Hashing Functions

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

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

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

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

A --> 0

B --> 1

C --> 2

D --> 3

...

and so on until Z --> 25.

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

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

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

# Basic Operations

Following are the basic primary operations of a hash table. Search − Searches an element in a hash table.

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

# DataItem

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

struct DataItem

{

int data; int key;

};

# Hash Method

Define a hashing method to compute the hash code of the key of the data item. int hashCode(int key){

return key % SIZE;

}

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Example

struct DataItem \*search(int key)

{

//get the hash

int hashIndex = hashCode(key);

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

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

//go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

return NULL;

}

# Insert Operation

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

Example

void insert(int key,int data)

{

struct DataItem \*item = (struct DataItem\*) malloc(sizeof(struct DataItem)); item->data = data;

item->key = key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty or deleted cell

while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) { //go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}

# Delete Operation

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

Example

struct DataItem\* delete(struct DataItem\* item) { int key = item->key;

//get the hash

int hashIndex = hashCode(key);

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

struct DataItem\* temp = hashArray[hashIndex];

//assign a dummy item at deleted position hashArray[hashIndex] = dummyItem; return temp;

}

//go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

return NULL;

}

# Collisions and Collision Handling

Problems, of course, arise when we have last names with the same first letter. So "Webster" and "Whitney" would correspond to the same index number, 22. A situation like this when two keys get sent to the same location in the array is called a collision. If you're trying to insert an element, you might find that the space is already filled by a different one.

Of course, you might try to just make a huge array and thus make it almost impossible for collisions to happen, but then that defeats the purpose of using a hash table. One of the advantages of the hash table is that it is both fast and small.

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

# Assignment 02

**Problem Statement:**

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)

# Objectives:

1. To understand Dictionary (ADT)
2. To understand concept of hashing
3. To understand concept & features like searching using hash function.

# Learning Objectives:

* + To understand Dictionary(ADT)
  + To understand concept of hashing
  + To understand concept & features like searching using hash function.

# Learning Outcome:

* + Define class for Dictionary using Object Oriented features.
  + Analyze working of hash function.

# Theory:

**Dictionary ADT**

Dictionary (map, association list) 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 lookup for a value by the key. Values are not required to be unique. Simple usage example is an explanatory dictionary. In the example, words are keys and explanations are values.

**Dictionary** Operations

# Dictionary create()

creates empty dictionary

# boolean isEmpty(Dictionary d)

tells whether the dictionary **d** is empty

# put(Dictionary d, Key k, Value v)

associates key **k** with a value **v;** if key **k** already presents in the dictionary old value is replaced by **v**

# Value get(Dictionary d, Key k)

returns a value, associated with key **k**or null, if dictionary contains no such key

# remove(Dictionary d, Key k)

removes key **k** and associated value

# destroy(Dictionary d)

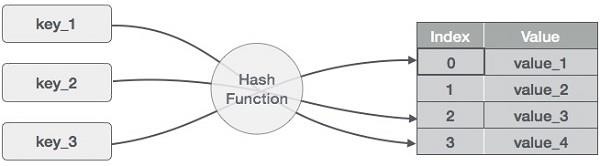
destroys dictionary **d**

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

# Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



# Basic Operations of hash table

Following are the basic primary operations of a hash table.

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

# DataItem

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

struct DataItem { int data;

int key;

};

# Hash Method

Define a hashing method to compute the hash code of the key of the data item. int hashCode(int key){

return key % SIZE;

}

# Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

# Example

struct DataItem \*search(int key) {

//get the hash

int hashIndex = hashCode(key);

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

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

//go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

return NULL;

}

# Insert Operation

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

# Example

void insert(int key,int data) {

struct DataItem \*item = (struct DataItem\*) malloc(sizeof(struct DataItem)); item->data = data;

item->key = key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty or deleted cell

while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {

//go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}

# Delete Operation

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

# Example

struct DataItem\* delete(struct DataItem\* item) { int key = item->key;

//get the hash

int hashIndex = hashCode(key);

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

if(hashArray[hashIndex]->key == key) {

struct DataItem\* temp = hashArray[hashIndex];

//assign a dummy item at deleted position hashArray[hashIndex] = dummyItem; return temp;

}

//go to next cell

++hashIndex;

//wrap around the table hashIndex %= SIZE;

}

return NULL;

}

**Software Required:** Dev C++ compiler- / 64 bit windows

**Input:** No. of. elements with key and value pair.

**Output:** Create dictionary using hash table and search the elements in table.

**Conclusion:** This program gives us the knowledge of dictionary(ADT).

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features.



**ELO2:** Understand & implement Dictionary (ADT) using hashing. 

# Problem Statement:

**GROUP B**

**Assignment 03**

A book consists of chapters, chapters consist of sections and sections consist of subsections.

Construct a tree and print the nodes. Find the time and space requirements of your method.

# Objectives:

* 1. To understand concept of Tree.
  2. To understand to find the record of book which consist of no of chapters,sections and subsections.
  3. To understand concept & features of creating nodes in object oriented programming.

# Learning Objectives

To understand concept of tree.

To understand operations like insert nodes in tree.

# Learning Outcome

Learn object oriented Programming features

Understand & implement concept of creating nodes with insert and display operation on it.

# Theory

A tree is a hierarchical data structure defined as a collection of nodes. Nodes represent value and nodes are connected by edges. A tree has the following properties:

1. The tree has one node called root. The tree originates from this, and hence it does not have any parent.
2. Each node has one parent only but can have multiple children.
3. Each node is connected to its children via edge.

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value, a list of references to nodes (the “children”).

# Basic Terminology In Tree Data Structure:

* **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {**2}** is the parent node of {**6, 7}**.
* **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {**6, 7}** are the child nodes of {**2}**.
* **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {**1}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
* **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree. The degree of the node {**3}** is **3**.
* **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {**6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19}** are the leaf nodes of the tree.
* **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {**1, 2}** are the parent nodes of the node {**7}**
* **Descendant:** Any successor node on the path from the leaf node to that node. {**7, 14}** are the descendants of the node. {**2}**.
* **Sibling:** Children of the same parent node are called siblings. {**8, 9, 10}** are called siblings.
* **Depth of a node:** The count of edges from the root to the node. Depth of node {**14}** is **3**.
* **Height of a node**: The number of edges on the longest path from that node to a leaf. Height of node {**3}** is **2**.
* **Height of a tree:** The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is **3**.
* **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
* **Internal node:** A node with at least one child is called Internal Node.
* **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
* **Subtree**: Any node of the tree along with its descendant.

Properties of Binary Tree

* + At each level of i, the maximum number of nodes is 2i.
  + 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 (20 + 21 + 22+….2h) = 2h+1 -1.
  + The minimum number of nodes possible at height h is equal to **h+1**.
  + If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

# 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:

1. **struct** node 2. {

1. **int** data,
2. **struct** node \*left, \*right; 5. }

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.

**Software Required:** Dev C++ compiler- / 64 bit windows

**Input:** Book name, chapter name, section name,and subsection name.

**Output:** Create root node as book name having child chapters,sections and subsections.

**Conclusion:** This program gives us the knowledge of create and display nodes in tree.

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features for tree.



# Problem Statement:

**GROUP B**

**Assignment 04**

Construct an expression tree from the given prefix expression eg. +--a\*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree.

# Objectives:

* 1. To understand concept of expression tree.
  2. To understand the concept of converting the given expression into prefix and postfix order.
  3. To understand concept & features of creating expression tree in object oriented programming.

# Learning Objectives

To understand concept of expression tree.

To understand operations like create expression tree, delete tree.

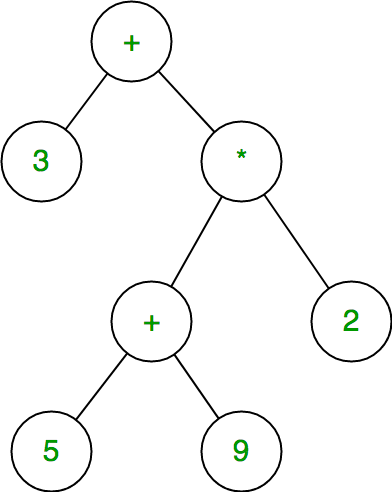
# Learning Outcome

Learn object oriented Programming features

Understand & implement concept of creating creating expression tree from given expression, delete tree

# Theory

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)\*2) would be:



# Evaluating the expression represented by an expression tree:

Let t be the expression tree If t is not null then

If t.value is operand then Return t.value

A = solve(t.left) B = solve(t.right)

// calculate applies operator 't.value'

// on A and B, and returns value Return calculate(A, B, t.value)

# Construction of Expression Tree:

Now For constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

**Prefix to Postfix step by step**

* Scan the given prefix expression from **right to left** character by character.
* If the character is an operand, push it into the stack.
* But if the character is an operator, pop the top two values from stack.

Concatenate this operator with these two values (**operator+1st top value+2nd top value**) to get a new string.

* Now push this resulting string back into the stack.
* Repeat this process untill the end of prefix expression. Now the value in the stack is the desired postfix expression.

**How to convert Postfix to Prefix?**

* Scan the given postfix expression from **left to right** character by character.
* If the character is an operand, push it into the stack.
* But if the character is an operator, pop the top two values from stack.

Concatenate this operator with these two values (**operator+2nd top value+1st top value**) to get a new string.

* Now push this resulting string back into the stack.
* Repeat this process untill the end of postfix expression. Now the value in the stack is the desired prefix expression.

**Software Required:** Dev C++ compiler- / 64 bit windows

**Input:** Expression in prefix order.

**Output:** Convert the given prefix expression into tree and write the post order traversal (non recursive) and then delete the entire tree.

**Conclusion:** This program gives us the knowledge of create and display expression tree.

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features for tree.

# Problem Statement:

**GROUP B**

**Assignment 05**

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

* 1. Insert new node
  2. Find number of nodes in longest path from root
  3. Minimum datavalue found in the tree
  4. Change a tree so that the roles of the left and right pointers are swapped at every node
  5. Search a value

# Objectives:

1. To understand concept of binary search tree .
2. To understand the concept of Insert new node, Find number of nodes, Minimum datavalue found in the tree

# Learning Objectives

To understand concept of binary search tree

# Learning Outcome

Learn object oriented Programming features

Understand & implement concept of Insert new node, Find number of nodes, Minimum datavalue found in the tree

# Theory

A node in Binary Search tree will be represented as follows:



* + Declare a structure to represent a node in the binary tree Struct BinTree

{

struct BinTree \*left; char data;

struct BinTree \*right;

};

* + Display a menu with the following options:
  1. Recursive Create
  2. Non – recursive create
  3. Insert New Node
  4. Find Height of the tree
  5. Smallest node value in the tree
  6. Mirror Image of the tree
  7. Search Value
  + Read the choice & switch according to that
  + If the choice is 1 or 2 create the root node & then give call to create function
  + Otherwise pass the root node as the input parameter to the function

# Rcreate ( ) function:

* + Main ( ) function has created the root node.
  + Display the data for root node.
  + Ask user if the new node is to be added to the left.
  + If choice is yes
    - Create a new node
    - Read the data for new node
    - Initialize left & right pointers of the new node to NULL.
    - Attach this new node to the left of the root
    - Give recursive call to Rcreate ( ) function with root->left as the new root.
  + Display the data for root node.
  + Ask user if the new node is to be added to the right.
  + If choice is yes
    - Create a new node
    - Read the data for new node
    - Initialize left & right pointers of the new node to NULL.
    - Attach this new node to the right of the root
    - Give recursive call to Rcreate ( ) function with root->right as the new root.
  + End of Rcreate()

# Nrcreate ( ) function

* + Main ( ) function has created the root node.
  + Let temp & new node be two node structures while(1)

begin

initialize temp to point to Root create a new node

initialize left & right pointer of new node to NULL accept data for new node

while(1) begin

If newnode->data < temp->data

If temp->left != NULL then temp = temp->left

else

break

temp->left = newnode

If newnode->data > temp->data

If temp->right != NULL then temp = temp->right

else

break

temp->right = newnode

End

Ask user if more nodes are to be added to the tree If the „no‟ break;

End

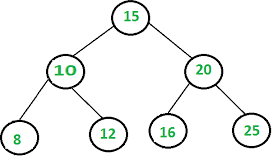
* + End of Nrcreate( )

# Find Height of the tree-

There are two conventions to define height of Binary Tree

1. Number of nodes on longest path from root to the deepest node.
2. Number of edges on longest path from root to the deepest node.

In this post, the first convention is followed. For example, height of the below tree is 3.



Recursive method to find height of Binary Tree is discussed [here.](http://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/#_blank) How to find height without recursion? We can use level order traversal to find height without recursion. The idea is to traverse level by level. Whenever move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level; stop traversing when count of nodes at next level is 0.

# Recursive Function-

int btree::nheight(node \*root)

{

int i, j, max=0;

i=1,j=1;

if(root!=NULL)

{

i=i+nheight(root->left); j=j+nheight(root->right); if(i>j)

else

}

max=i;

max=j;

return(max);

}

# Non-recursive function-

int BinaryTree :: TreeHeight(TreeNode \*Root)

{

int heightL, heightR; if(Root == Null)

return 0;

if(Root->Lchild == Null && Root->Rchild == Null) return 0;

heightL = TreeHeight(Root->Lchild); heightR = TreeHeight(Root->Rchild); if(heightR > heightL)

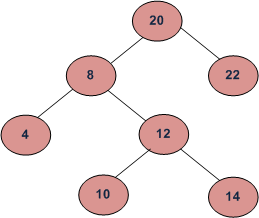
return(heightR + 1);

return(heightL + 1);

}

# Find the node with minimum value in a Binary Search Tree-

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

In binary search tree, the smallest node is in the left side and the largest node is in the right side. To find the smallest node, the process will check the parent node. In case that the parent node is not empty, if it doesn't have a left child node, the smallest node is the parent node; otherwise the smallest node is its left child node.

**Find Smallest Node function-( Non-recursive)**

void btree::smallest(node\* root)

{

node \*temp; temp=root;

while(temp->left!=NULL) temp=temp->left;

cout<<"\nMinimum data value="<<temp->data;

}

**Find Largest Node function-( Non-recursive)**

void btree::largest(node\* root)

{

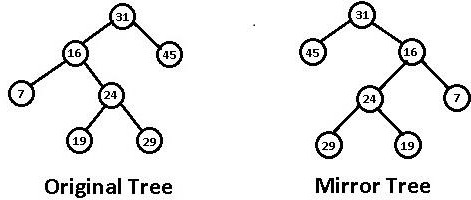
node \*temp; temp=root;

while(temp->right!=NULL) temp=temp->right;

cout<<"\nMaximum data value="<<temp->data;

}

# Getting Mirror, Replica, or Tree Interchange of Binary Tree

The Mirror() operation finds the mirror of the tree that will interchange all left and right subtrees in a linked binary tree.

# Recursive Function

void BinaryTree :: Mirror(TreeNode \*Root)

{

TreeNode \*Tmp; if(Root != Null)

{

Tmp = Root->Lchild;

Root->Lchild = Root->Rchild; Root->Rchild = Tmp; Mirror(Root->Lchild);

Mirror(Root->Rchild);

}

}

# Mirror( ) Function (Non-Recursive) using Queue-

Initialize a queue of nodes as empty Initialize a pointer temp = root Add(temp) to queue

while( queue not empty) begin

temp = delete

if (temp->left != NULL)

add(temp->left) if(temp->right != NULL)

add (temp->right) change = temp->left

temp->Left = temp->right temp->right = change

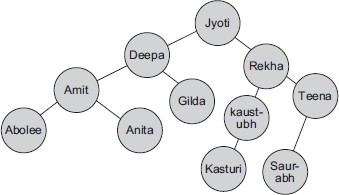
end end

# Searching for a Key-

To search for a target key, we first compare it with the key at the root of the tree. If it is the same, then the algorithm ends. If it is less than the key at the root, search for the target key in

the left subtree, else search in the right subtree. Let us, for example, search for the key

„Saurabh‟ in following Figure



We first compare „Saurabh‟ with the key of the root, „Jyoti‟. Since „Saurabh‟ comes after „Jyoti‟ in alphabetical order, we move to the right side and next compare it with the key „Rekha‟. Since

„Saurabh‟ comes after „Rekha‟, we move to the right again and compare with „Teena‟. Since

„Saurabh‟ comes before „Teena‟, we move to the left. Now the question is to identify what event will be the terminating condition for the search. The solution is if we find the key, the function finishes successfully. If not, we continue searching until we hit an empty subtree.

Program Code shows the implementation of search () function, both nonrecursive and recursive implementations.

# Non-recursive-

TreeNode \*BSTree :: Search(int Key)

{

TreeNode \*Tmp = Root; while(Tmp)

{

if(Tmp->Data == Key)

return Tmp;

else if(Tmp->data < Key) Tmp = Tmp->Lchild;

else Tmp = Tmp->Rchild;

}

return NULL;

}

# Recursive-

TreeNode \*BSTree :: Rec\_Search(TreeNode \*root, int key)

{ if(root == Null) return(root);

else

{

}

}

if(root->Data < Key)

root = Rec\_Search(root->Lchild); else if(root->data > Key)

root = Rec\_Search(root->Rchild);

**Software Required:** Dev C++ compiler- / 64 bit windows

**Input:** node names

**Output:** implement concept of Insert new node, Find number of nodes, Minimum data value found in the tree

**Conclusion:** Successfully implemented Binary search tree as an ADT using linked list in C++ language.

.

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features for binary search tree.

# Problem Statement:

**GROUP C**

**Assignment 06**

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.

# Objectives:

* 1. To understand concept of adjacency matrix/list.
  2. To understand the concept of DFS and BFS

# Learning Objectives

To understand concept of adjacency matrix/list to perform DFS and using adjacency list to perform BFS

# Learning Outcome

Learn object oriented Programming features

Understand & implement concept of adjacency matrix/list to perform DFS and using adjacencylist to perform BFS

# Theory

Depth–first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root for a graph) and explore as far as possible along each branch before [backtracking.](https://www.techiedelight.com/backtracking-interview-questions/)

# Depth–first search in Graph

A [Depth–first search (DFS)](https://www.techiedelight.com/depth-first-search/) is a way of traversing graphs closely related to the preorder traversal of a tree. Following is the recursive implementation of preorder traversal:

procedure preorder(treeNode v)

{

visit(v);

for each child u of v preorder(u);

}

To turn this into a graph traversal algorithm, replace “child” with “neighbor”. But to prevent

infinite loops, keep track of the vertices that are already discovered and not revisit them.

The non-recursive implementation of DFS is similar to the [non-recursive implementation of](https://www.techiedelight.com/breadth-first-search/#iterative) [BFS](https://www.techiedelight.com/breadth-first-search/#iterative) but differs from it in two ways:

procedure dfs(vertex v)

{

visit(v);

for each neighbor u of v if u is undiscovered

call dfs(u);

}

**Iterative Implementation of DFS**

* It uses a [stack](https://www.techiedelight.com/stack-implementation/) instead of a [queue](https://www.techiedelight.com/circular-queue-implementation-c/).
* The DFS should mark discovered only after popping the vertex, not before pushing it.
* It uses a reverse iterator instead of an iterator to produce the same results as recursive DFS.

**Breadth–first search (BFS**

Breadth–first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a

„search key‟) and explores the neighbor nodes first before moving to the next-level neighbors.

[Breadth–first search (BFS)](https://www.techiedelight.com/bfs-interview-questions/) is a graph traversal algorithm that explores vertices in the order of their distance from the source vertex, where distance is the minimum length of a path from the source vertex to the node as evident from the above example.

# Applications of BFS

* Copying garbage collection, Cheney‟s algorithm.
* Finding the shortest path between two nodes and v , with path length measured by the

u

total number of edges (an advantage over depth–first search).

* Testing a graph for bipartiteness.
* Minimum Spanning Tree for an unweighted graph.
* Web crawler.
* Finding nodes in any connected component of a graph.
* Ford–Fulkerson method for computing the maximum flow in a flow network.
* Serialization/Deserialization of a [binary tree](https://www.techiedelight.com/binary-tree-interview-questions/) vs. serialization in sorted order allows the tree to be reconstructed efficiently.

Iterative Implementation of BFS

The non-recursive implementation of BFS is similar to the [non-recursive implementation of](https://www.techiedelight.com/depth-first-search/#iterative) [DFS](https://www.techiedelight.com/depth-first-search/#iterative) but differs from it in two ways:

* It uses a [queue](https://www.techiedelight.com/circular-queue-implementation-c/) instead of a [stack](https://www.techiedelight.com/stack-implementation/).
* It checks whether a vertex has been discovered before pushing the vertex rather than delaying this check until the vertex is dequeued.

**Software Required:** Dev C++ compiler- / 64 bit windows

**Input:** Graph

**Output:** implement concept of traveling BFS,DFS

**Conclusion:** Successfully implemented DFS and BFS using adjacency matrix and list in C++ language.

.

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features for DFS and BFS.

# Problem Statement:

**GROUP C**

**Assignment 07**

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

Check whether the graph is connected or not. Justify the storage representation used

# Objectives:

1. To understand concept of Graph

# Learning Objectives

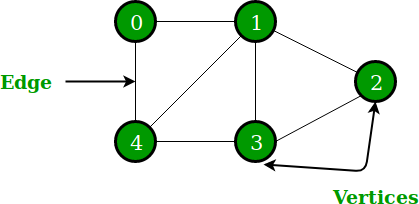
To understand concept of Graph for flight path between cities.

# Learning Outcome

Learn object oriented Programming features Understand & implement concept of Graph.

# Theory

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also

used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

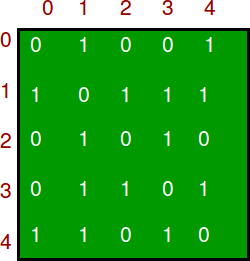
The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

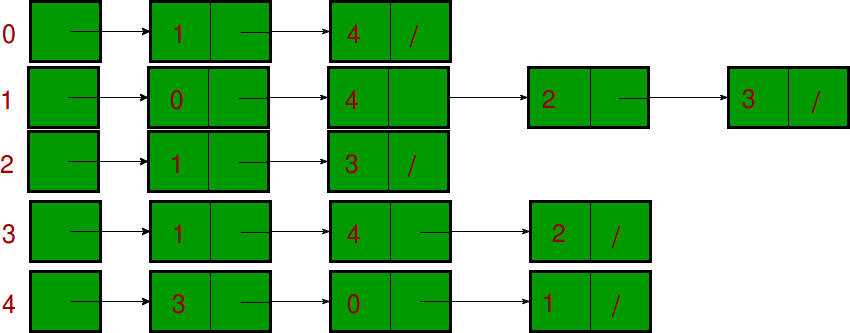
# Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



# Adjacency List:

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the 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 represented as lists of pairs. Following is the adjacency list representation of the above graph.



**Software Required:** Dev C++ compiler- / 64 bit windows

**Input:** Graph

**Output:** implement concept of Graph for flight path between two cities.

**Conclusion:** Successfully implemented graph in C++ language.

.

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features graph.

# Problem Statement:

**GROUP D**

**Assignment 08**

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

# Objectives:

1. To understand concept of OBST.
2. To understand concept & features like extended binary search tree.

# Learning Objectives:

* + To understand concept of OBST.
  + To understand concept & features like extended binary search tree.

# Learning Outcome:

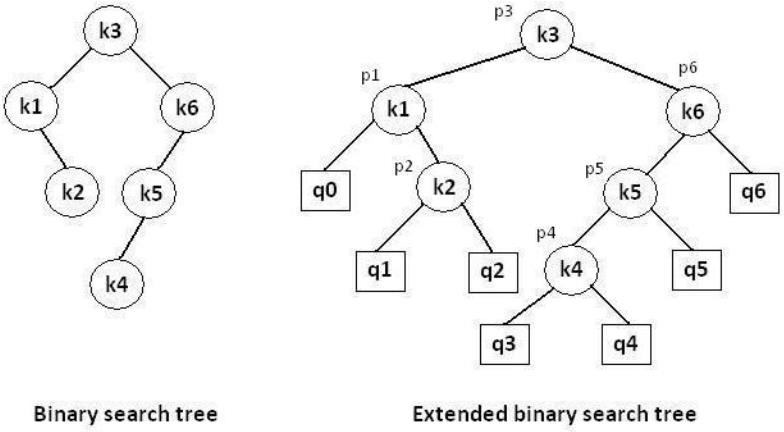
* + Define class for Extended binary search tree using Object Oriented features.
  + Analyze working of functions.

# Theory:

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

For the purpose of a better presentation of optimal binary search trees, we will consider “extended binary search trees”, which have the keys stored at their internal nodes. Suppose “n” keys k1, k2, … k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn.

An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



# In the extended tree:

* The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;
* The round nodes represent internal nodes; these are the actual keys stored in the tree;
* Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.
* If the user searches a particular key in the tree, 2 cases can occur:
* 1 – the key is found, so the corresponding weight „p‟ is incremented;
* 2 – the key is not found, so the corresponding „q‟ value is incremented.

# GENERALIZATION:

The terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successes between ki and ki-1 in an inorder traversal represent all key values not stored that lie between ki and ki - 1.

# ALGORITHMS

# COMPLEXITY ANALYSIS:

The algorithm requires O (n2) time and O (n2) storage. Therefore, as „n‟ increases it will run out of storage even before it runs out of time. The storage needed can be reduced by almost half by implementing the two-dimensional arrays as one-dimensional arrays.

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

**Input: 1.**No.of Element.

1. key values
2. Key Probability

**Output:** Create binary search tree having optimal searching cost.

**Conclusion:** This program gives us the knowledge OBST, Extended binary search tree.

# OUTCOME

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features.



**ELO2:** Understand & implement extended binary search tree.

# GROUP D

**Assignment 09**

**Problem 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.

# Prerequisite:

1. Basic concepts of thread
2. Concepts of in-Order & pre-Order traversals.

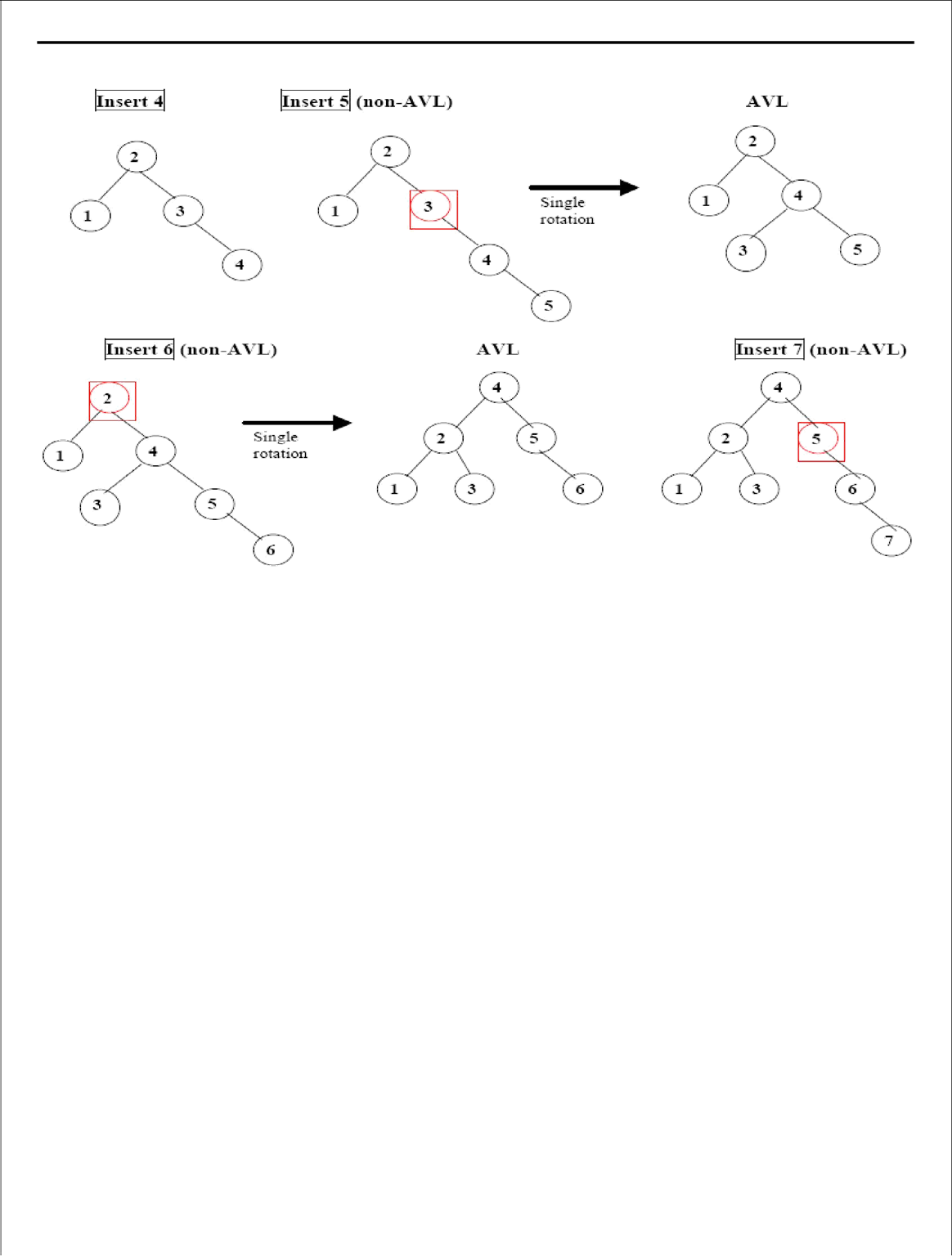
# Theory :

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.

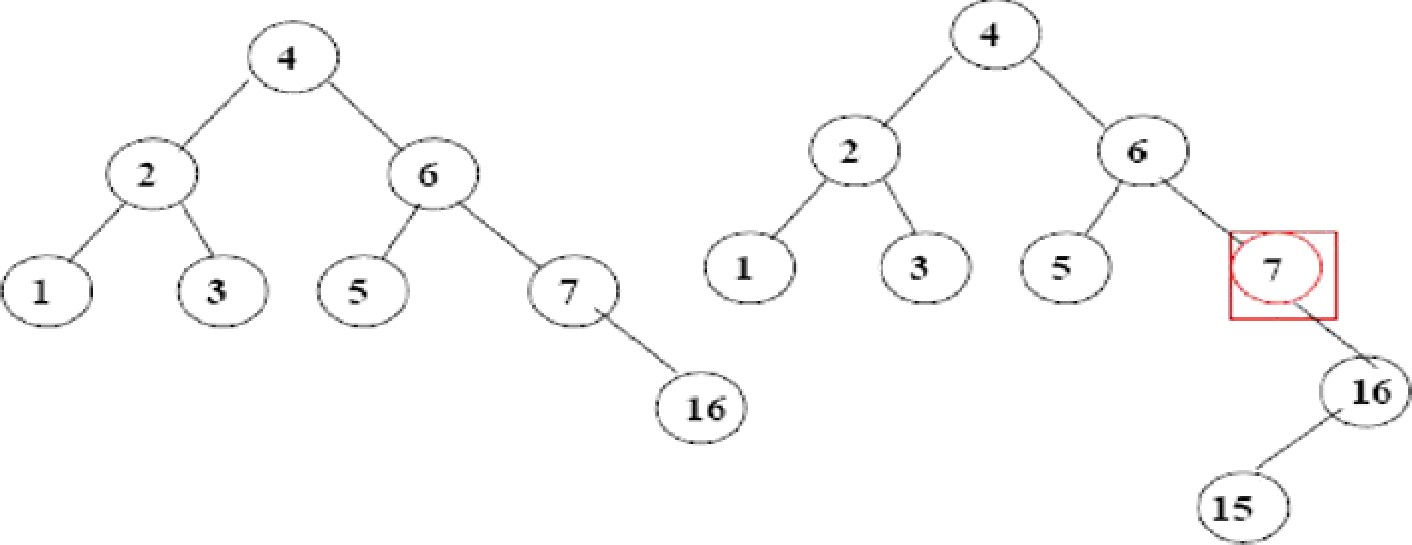
**AVL (Adelson- Velskii and Landis) Tree:** A balance binary search tree. The best search time, that is O (log

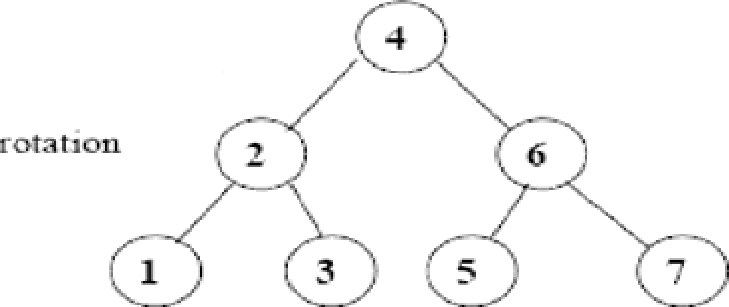
N) search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1



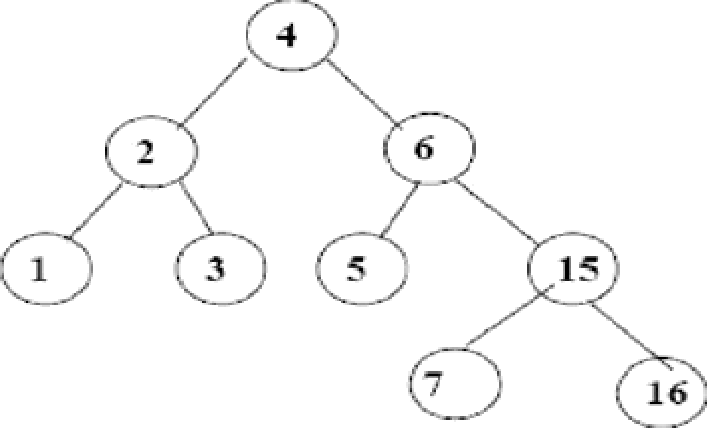
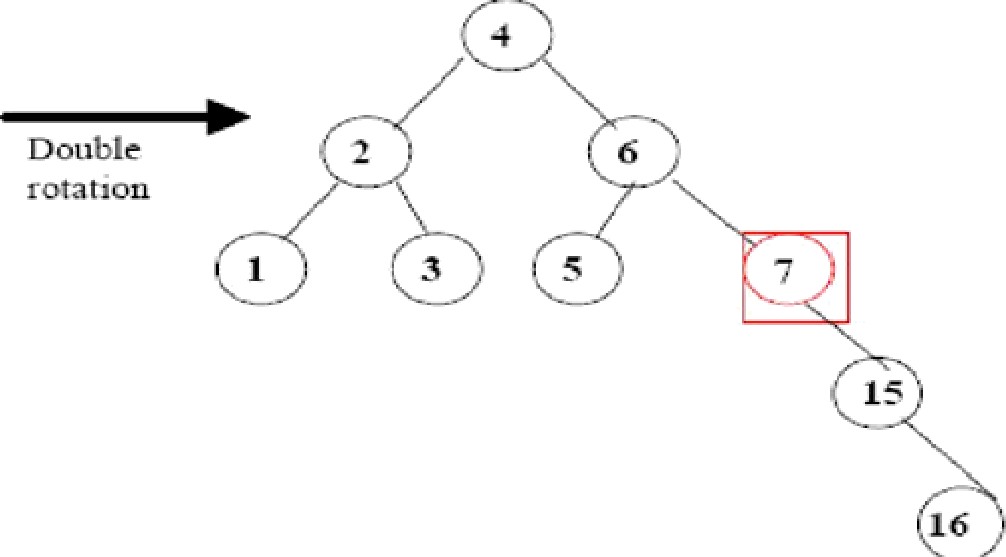


sql (non-A4\* )





Steta 1: Rotate ckilcl a nd gra uilcliild Stcp 2: Rota te oo‹te aucl ueu child (A4” )



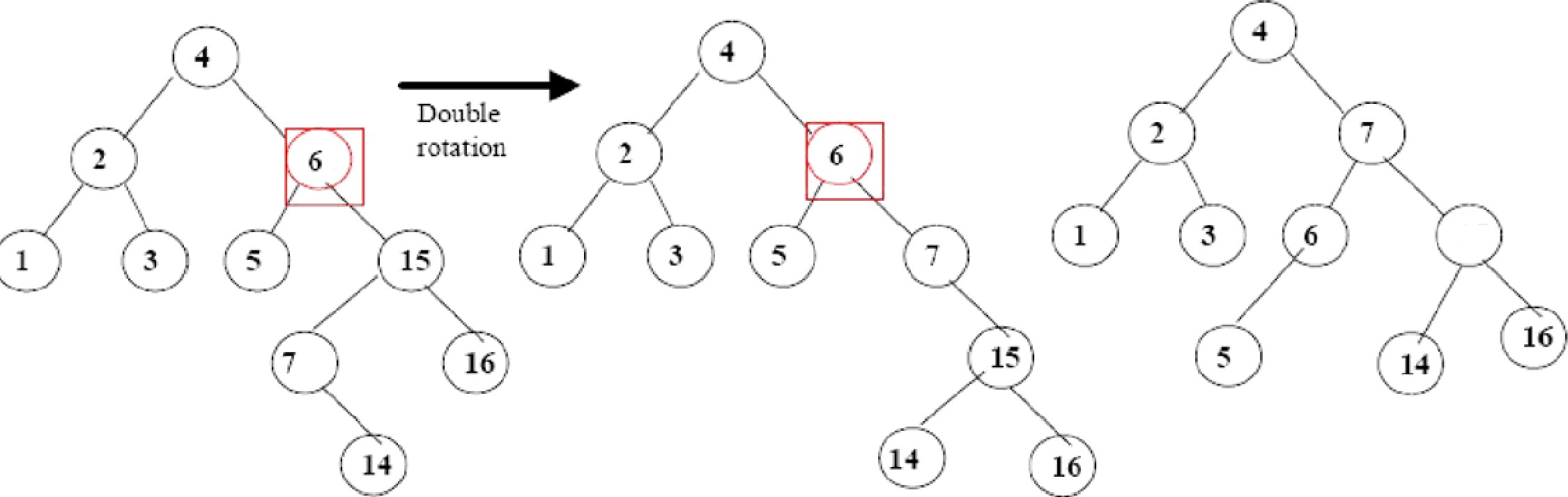
s (non-A3\*L)

Step 1: Rotate child rind

grandchild

Step 2: ROtRte code and

new’ child (ñA”L)



15

An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1. An example of an AVL tree where the heights are shown next to the nodes

AVL tree is a height balance tree.

The height of the right sub tree and height of the left sub tree for any node cannot differ by more than one.

This process is usually done through rotation.

# Operation on AVL Tree:- Insertion of a node:-

Inserting in AVL tree is same as in binary search tree. Here also we will search for the position where the new node is to be inserted and then insert the node.

To restore the property of AVL tree we should convert the tree in such a way that, the new converted tree is balance tree i.e. the balance factor of each node should be -1, 0, 1.

The new converted node should be a binary search tree with in order traversals same as that of original tree.

The outline of the procedure to insert of a node is as- insert node to its proper place follow the same process as in binary search tree.

Calculate the balance factor of the entire path starting from the inserted node to the root node.



If the tree become unbalance after insertion then there is need to convert the above tree by performing rotations.

# Deletion of a node at any position:-

Read the node from the user which he wants to delete. Find out the node position and delete the node.



Check for the balance factor.

If tree is imbalance then perform the rotations. Stop.

# Algorithm:

**Insertion** To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation

1. Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

y x

/ \ Right Rotation / \ x T3 – - – - – - – > T1

y / \ < / \

T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3) So BST property is not violated anywhere.

**Steps to follow for insertion** Let the newly nserted node be w **1)** Perform standard BST insert for w. **2)** Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z. **3)** Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements: a) y is left child of z and x is left child of y (Left Left Case) b) y is left child of z and x is right child of y (Left Right Case) c) y is right child of z and x is right child of y (Right Right Case) d) y is right child of z and x is left child of y (Right Left Case)

Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re- balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion.

# Left Left Case

T1, T2, T3 and T4 are subtrees. z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - -> / \ /

\ x T3 T1 T2 T3 T4

/ \

T1 T2

* 1. **Left Right Case** z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y z / \ - - - - - - - - -> / \ > / \ / \

T1 x y T3 T1 T2 T3

T4 / \ / \

T2 T3 T1 T2

# Right Right Case z y

/ \ / \

T1 y Left Rotate(z) z x / \ > / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

# Right Left Case

z z x

/ \ / \ / \

T1 y Right Rotate (y) T1 x Left Rotate(z) z x

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ /

\ x T4 T2 y T1 T2 T3 T4

/ \ / \

T2 T3 T3 T4

**deletion**. To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re- balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

y x

/ \ Right Rotation / \

x T3 – – – – – – – > T1 y

/ \ < - - - - - - - / \

T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order keys(T1) < key(x) < keys(T2) < key(y) < keys(T3) So BST property is not violated anywhere.

Let w be the node to be deleted **1)** Perform standard BST delete for w. **2)** Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here. **3)** Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements: a) y is left child of z and x is left child of y (Left Left Case) b) y is left child of z and x is right child of y (Left Right Case) c) y is right child of z and x is right child of y (Right Right Case) d) y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won‟t fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

# Left Left Case

T1, T2, T3 and T4 are subtrees. z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - -> / \ /

\ x T3 T1 T2 T3 T4

/ \

T1 T2

1. **Left Right Case** z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y z

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ /

\ T1 x y T3 T1 T2 T3 T4

/ \ / \

T2 T3 T1 T2

# Right Right Case z y

/ \ / \

T1 y Left Rotate(z) z x / \ > / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

# Right Left Case

z z x

/ \ / \ / \

T1 y Right Rotate (y) T1 x Left Rotate(z) z x

/ \ - - - - - - - - -> / \ - - - - - - - -> / \ /

\ x T4 T2 y T1 T2 T3 T4

/ \ / \

T2 T3 T3 T4

Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of

z. Thus, we must continue to trace the path until we reach the root.

**OUTPUT** :-

1. Create 2.Insert 3.Delete 4.Display 5.Exit

Enter a choice1

Enter the number of words3 Enter word 1 :ANIKET Enter the meaning : DALAL Enter word 2 :SACHIN

Enter the meaning : DHAMAL Enter word 3 :BALRAM

Enter the meaning : CHAVAN

1. Create 2.Insert 3.Delete 4.Display 5.Exit

Enter a choice4

ANIKET : DALAL(BF=0) BALRAM :

CHAVAN(BF=0) SACHIN : DHAMAL(BF=0) 1.Create

1. Insert 3.Delete 4.Display 5.Exit

Enter a choice2

Enter word :RAMESH

Enter the meaning : YADAV 1.Create

2.Insert 3.Delete 4.Display 5.Exit

Enter a choice4

ANIKET : DALAL(BF=0) BALRAM : CHAVAN(BF=-

1) RAMESH :

YADAV(BF=0) SACHIN : DHAMAL(BF=1) 1.Create

2.Insert 3.Delete 4.Display 5.Exit

Enter a choice3

Enter which word you want to delete :BALRAM 1.Create

1. Insert 3.Delete

4.Display 5.Exit

Enter a choice4

ANIKET : DALAL(BF=0) RAMESH :

YADAV(BF=0) SACHIN : DHAMAL(BF=0)

1.Create 2.Insert 3.Delete 4.Display 5.Exit

Enter a choice5

**Conclusion:**In this way we have implemented dictionary programme in cpp

# Problem Statement:

**GROUP E**

**Assignment No.10**

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

# Prerequisite:

**Theory:**Basics of the Cpp.

1. Knowledge of the Object Oriented Language like C++.
2. Concept of dynamic allocation.

This is a Cpp Program to implement Heap Sort on an integer array. Heapsort is a comparison-based sorting algorithm to create a sorted array (or list), and is part of the selection sort family. Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime. Heapsort is an in-place algorithm, but it is not a stable sort.

Worst Case Performance:O(nlogn) Best Case Performance:O(nlogn) Average case performance : O(n log n)

# Algorithm:

# Output:

Heap Sort Test

Enter number of integer elements 20 Enter 20 integer elements

488 667 634 380 944 594 783 584 550 665 721 819 285 344 503 807 491 623 845 300

Elements after sorting

285 300 344 380 488 491 503 550 584 594 623 634 665 667 721 783 807 819 845 944

**Conclusion:** In this way we have implemented heap sort by using java.

GROUP F

# Assignment 11

**Problem statement:**

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

# Prerequisite:

**Theory:**

1. Knowledge of the Object Oriented Language like C++.
2. Concept of dynamic allocation.

# File structure–

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

Two characteristics determine how the file is organised:

# File Activity:

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

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

# File Volatility:

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

# File organisation –

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

There are four methods for organising files-

1. Sequential organisation
2. Indexed Sequential organisation
3. Inverted list organization
4. Direct access organisation
5. Chaining

# Sequential organization:

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

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

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

In a

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

# Advantage

* + 1. Simple to design ii.Easy to program

iii.Variable length and blocked records available iv.Best use of storage space

# Disadvantages

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

# Indexed sequential organization:

Like sequential organization, keyed sequential organization stores data in physicallycontiguous blocks. The difference is in the use of indexes to locate records. There are three areas in disk storage: prime area, overflow area and index area.The prime area contains file records stored by key or id numbers. All records are initially stored in the prime area.The overflow area contains records added to the file that cannot be placed in logical sequence in the prime area.The index area is more like a data dictionary. It contains keys of records and their locations on the disk. A pointer associated with each key is an address that tells the system where to find a record.

# Advantages:

* + 1. Indexed sequential organization reduces the magnitude of the sequential search and provides quick access for sequential and direct processing.

ii Records can be inserted in the middle of the file.

# Disadvantages:

i.It takes longer to search the index for data access or retrieval. ii.Unique keys are required.

iii.Periodic reorganization is require

# Inverted list organization:

Like the indexed- sequential storage method the inverted list organization maintains an index. The two methods differ, however, in the index level and record storage. The indexed sequential method has a multiple index for a given key, where as the inverted list method has a single index for each key type. In an inverted list, records are not necessarily stored in a particular sequence. They are placed in the

data storage area, but indexes are updated for the record key and location. The inverted keys are best for applications that request specific data on multiple keys. They are ideal for static files because additions and deletions cause expensive pointer updating.

# Advantages

i. Used in applications requesting specific data on multiple keys.

# Direct access organization:

In direct access file organization, records are placed randomly throughout the file. Records need not be in sequence because they are updated directly and rewritten back in the same location. New records are added at the end of the file or inserted in specific locations based on software commands.

Records are accessed by addresses that specify their disk locations. An address is required for locating a record, for linking records, or for establishing relationships. Addresses are of two types:

i. Absol ute ii Relati ve.

**A absolute address** represents the physical location of the record. It is usually stated in the format of sector/track/record number. One problem with absolute address is that they become invalid when the file that contains the records is relocated on the disk.

**A relative address** gives a record location relative to the beginning of the file. There must be fixed length records for reference. Another way of locating a record is by the number of bytes it is from the beginning of the file. When the file is moved, pointers need not be updated because the relative location remains the same.

# Advantages:

* + 1. Records can be inserted or updated in the middle of the file.
    2. Better control over record allocation.

# Disadvantages:

1. Calculating address required for processing.
2. Impossible to process variable length records.

# Chaining:

File organization requires that relationships be established among data items. It must show how characters form fields, fields form files and files relate to each other. Establishing relationship is done through chaining. It uses pointers.

**Example:** The file below contains auto parts that are an indexed sequential file sequenced by part no. A record can be retrieved by part no. To retrieve the next record, the whole file has to be searched. This can be avoided by the use of pointers.

# Algorithm:

# Output:

Enter File Name: employee.txt Enter details of the employee:

Name:Akash Post: Worker

Name:Shrikant Post:Manager

The content of employee.txt file Name:Akash Post: Worker Name:Shrikant Post: Manager

**Conclusion:** In this way we have implemented sequential file in cpp.

# Assignment 12

**Problem Statement:**

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

**Objectives:**

1. To understand index sequential file

# Learning Objectives:

* + To understand index sequential file to maintain the data.

# Learning Outcome:

* + Define class for employee information using index sequential file

# Theory:

**Indexed sequential access file organization**

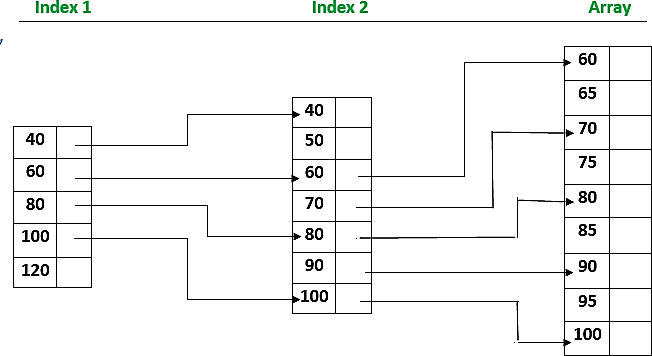
* Indexed sequential access file combines both sequential file and direct access file organization.
* In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.
* This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.
* The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

# Characteristics of Indexed Sequential Search:

* + In Indexed Sequential Search a sorted index is set aside in addition to the array.
  + Each element in the index points to a block of elements in the array or another expanded index.
  + The index is searched 1st then the array and guides the search in the array.

**Note:** Indexed Sequential Search actually does the indexing multiple time, like creating the index of an index.

# Explanation by diagram “Indexed Sequential Search”:



**Advantages of Indexed sequential access file organization**

* In indexed sequential access file, sequential file and random file access is possible.
* It accesses the records very fast if the index table is properly organized.
* The records can be inserted in the middle of the file.
* It provides quick access for sequential and direct processing.
* It reduces the degree of the sequential search.

# Disadvantages of Indexed sequential access file organization

* Indexed sequential access file requires unique keys and periodic reorganization.
* Indexed sequential access file takes longer time to search the index for the data access or retrieval.
* It requires more storage space.
* It is expensive because it requires special software.
* It is less efficient in the use of storage space as compared to other file organizations

**Conclusion:** In this way we have implemented index sequential file in cpp