



Dr. D. Y. Patil Educational Federation's
Dr. D. Y. Patil College of Engineering & Innovation
Approved by AICTE, Affiliated to SPPU Pune

Department of Computer Engineering

LAB MANUAL

Data Structures and Algorithms Laboratory

Class: SE Computer
Semester IV

Prepared By: Mrs Nikita Oswal

Savitribai Phule Pune University Second Year of Computer Engineering (2019 Course) 210256: Data Structures and Algorithms Laboratory		
Teaching Scheme Practical: 04 Hours/Week	Credit Scheme 02	Examination Scheme and Marks Term Work: 25 Marks Practical: 25 Marks
Companion Course : 210252: Data Structures and Algorithms, 210255: Principles of Programming Languages		

Course Objectives:

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

Course Outcomes:

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

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

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

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

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

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

@The CO-PO Mapping Matrix												
PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	1	2	2	-	-	-	-	-	-	-	-	-
CO2	-	2	2	-	-	-	-	-	-	-	-	-
CO3	-	2	2	1	-	-	-	-	-	-	-	-
CO4	1	2	1	1	-	-	-	-	-	-	-	-
CO5	1	1	2	2	-	-	-	-	-	-	-	-

Operating System recommended: 64-bit Open source Linux or its derivative, Windows

Programming tools recommended: Turbo C++, VS code, Online Compilers

Experiment List

Exp. No.	Title of Experiments
1	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.
2	Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement. 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)
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.
4	Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -i. Insert new node, ii. Find number of nodes in longest path from root, iii. Minimum data value found in the tree, iv. Change a tree so that the roles of the left and right pointers are swapped at every node, v. Search a value
5	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.
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.
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.
8	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.
9	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?
10	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
11	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.
12	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.
13	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.

14	Mini Project
----	--------------

EXPERIMENT NO: 1 (A)**Problem Statement:**

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.

Objectives:

1. Understand the concept of ADT functions
2. Understand the concept of Hashing

Outcomes:

1. Will be able to use concept of ADT
2. Will be able to store records in hash table using hashing.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-**1. Hashing :**

Hashing is finding an address where the data is to be stored as well as located using a key with the help of the algorithmic function. The resulting address is used as the basis for storing and retrieving records and this address is called as home address of the record. For array to store a record in a hash table, hash function is applied to the key being stored, returning an index within the range of the hash table. The item is then stored in the table of that index position.

2. Hash function:

Hashing is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the hash function. A function that maps a key into the range $[0 \text{ to } \text{Max} - 1]$, the result of which is used as an index (or address) to hash table for storing and retrieving record.

3. Hash table:

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster. A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping. A hash table is a data structure that stores records in an array, called a hash table. A Hash table can be used for quick insertion and searching.

Here, are **pros/benefits** of using hash tables:

1. Hash tables have high performance when looking up data, inserting, and deleting existing values.

2. The time complexity for hash tables is constant regardless of the number of items in the table.
3. They perform very well even when working with large datasets. Here, are **cons** of using hash tables:

1. You cannot use a null value as a key.
2. Collisions cannot be avoided when generating keys using hash functions. Collisions occur when a key that is already in use is generated.
3. If the hashing function has many collisions, this can lead to a performance decrease

Here, are the Operations supported by Hash tables:

1. Insertion – this Operation is used to add an element to the hash table
2. Searching – this Operation is used to search for elements in the hash table using the key
3. Deleting – this Operation is used to delete elements from the hash table

Real-world Applications

In the real-world, hash tables are used to store data for

1. Databases
2. Associative arrays
3. Sets
4. Memory cache.

Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is $O(1)$. Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is $O(n)$, and $O(\log n)$ in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time. In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored.

It can be written as: • $\text{Index} = \text{hash}(\text{key})$

Collision:

When the two different values have the same index, then the problem occurs between the two values, known as a collision.

Example:

If size of hash table is 10. The value 16 is stored at index 6. If the key value is 26, then the index would be: $h(26) = 26 \% 10 = 6$. Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem.

To resolve these collisions, we have some techniques known as collision techniques. The following are the collision techniques:

Open Hashing: It is also known as closed addressing. Closed Hashing: It is also known as open addressing.

Steps of Implementation:

1. Singly link list used for implementation of hashing. Create the structure for the node.
2. **Create hashing class** containing hashing () constructor for the initialization of hash table with , hash function Create() ,Search(), Delete(), Insert()Function for implementation of hashing, and perform collision handling in hashing using chaining.
3. **Create()-** Create new node to store element after a collision occurs used create() function.
4. **Display()-** To display hash table.
 - i) Starting from 0 to last index.. Create temp node i.e. new node and assign Value of index of hash table to temp node first.

ii) Display that value like a[0] while temp not equal to Null.

5. **Search()** -

To search particular number.

- i) Boolean flag have two value true or false. User enter telephone number which want to search. on that number we apply hash function. Get hash value i.e. index of element.
- ii) Whatever location of hash value move toward to that node that node is consider as Entry node. At particular location value not found then need to search next link list available at that particular index. While entry is not equal to null. If we are get match then print element position, element set flag to true otherwise flag is false.
- iii) If entry is null means element not found.

6. Delete()

To delete the element from hash tabl

- i) Calculate index by applying hash function on that value.
- ii) Then search value at that location of index. For that index value consider as entry value.
- iii) If entry value NULL then no element to delete.
- iv) Value of entry node is equal to value which need to delete if match then delete element and entry node point to next node.
- v) If entry of value not matching till last then message element not fond in hash table.

7. Insert()

To insert element in hash table

- i) Create two node one is temp and another is head.
- ii) Initial node consider as temp. Next node which is to be inserted consider as head node
- iii) When new element is inserted then consider head node as temp and new element node as head.

Conclusion: Thus we have implemented hashing and collision handling techniques & Make use of a hash table implementation to quickly look up client's telephone number.

EXPERIMENT NO: 2 (A)**Problem Statement:**

Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement. 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)

Objectives:

1. Understand the concept of ADT functions
2. Understand the concept of Hashing
3. Understand the concept of chaining with / without replacement

Outcomes:

1. Will be able to use concept of ADT
2. Will be able to apply concept of chaining with/without replacement.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-**1. Hashing :**

Hashing is finding an address where the data is to be stored as well as located using a key with the help of the algorithmic function. The resulting address is used as the basis for storing and retrieving records and this address is called as home address of the record. For array to store a record in a hash table, hash function is applied to the key being stored, returning an index within the range of the hash table. The item is then stored in the table of that index position.

2. Hash function:

Hashing is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the hash function. A function that maps a key into the range $[0 \text{ to } \text{Max} - 1]$, the result of which is used as an index (or address) to hash table for storing and retrieving record.

3. Hash table:

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster. A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping. A hash table is a data structure that stores records in an array, called a hash table. A Hash table can be used for quick insertion and searching.

Here, are the Operations supported by Hash tables:

1. Insertion – this Operation is used to add an element to the hash table
2. Searching – this Operation is used to search for elements in the hash table using the key
3. Deleting – this Operation is used to delete elements from the hash table

Real-world Applications

In the real-world, hash tables are used to store data for

1. Databases
2. Associative arrays
3. Sets
4. Memory cache.

Collision:

When the two different values have the same index, then the problem occurs between the two values, known as a collision.

How to handle Collisions?

There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing

Separate Chaining:

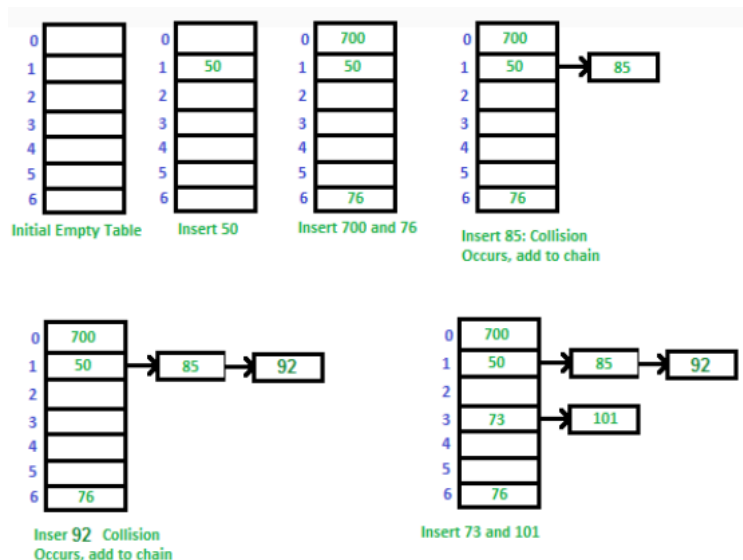
The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

The **linked list** data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

In separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

Steps of Implementation:

1. First draw the empty hash table which will have a possible range of hash values
2. Now insert all the keys in the hash table one by one by calculating mod value.
3. If slot is empty then put it at same place.
4. If slot is not empty, then collision occur, so put it in chain one after other.



Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as closed hashing. This entire procedure is based upon probing. We will understand the types of probing ahead:

- **Insert(k):** Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- **Search(k):** Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- **Delete(k): Delete operation is interesting.** If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".

Different ways of Open Addressing:

1. Linear Probing:

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: $\text{rehash}(\text{key}) = (\text{n}+1)\% \text{table-size}$.

Steps of Implementation:

1. First draw the empty hash table which will have a possible range of hash values
2. Now insert all the keys in the hash table one by one by calculating mod value.
3. If slot is empty then put it at same place.
4. If slot is not empty, then collision occur, so put it in next subsequent empty place. Follow the linear sequential order.

Conclusion: Thus we have implemented hashing and collision handling technique linear probing.

EXPERIMENT NO: 3 (B)**Problem Statement:**

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. Understand the concept of Tree.
2. Understand the concept of calculating time and space requirement.

Outcomes:

1. Will be able to use concept of Tree
2. Will be able to calculate time and space requirement

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

A tree is a data structure composed of nodes that has the following characteristics:

1. Each tree has a root node at the top (also known as Parent Node) containing some value (can be any datatype).
2. The root node has zero or more child nodes.
3. Each child node has zero or more child nodes, and so on. This creates a subtree in the tree. Every node has its own subtree made up of its children and their children, etc. This means that every node on its own can be a tree.

A binary search tree (BST) adds these two characteristics:

1. Each node has a maximum of up to two children.
2. For each node, the values of its left descendent nodes are less than that of the current node, which in turn is less than the right descendent nodes (if any).

Basic operations on a BST

- Create: creates an empty tree.
- Insert: insert a node in the tree.
- Search: Searches for a node in the tree.
- Delete: deletes a node from the tree.
- Inorder: in-order traversal of the tree.
- Preorder: pre-order traversal of the tree.
- Postorder: post-order traversal of the tree.

Here's a definition for a BST node having some data, referencing to its left and right child nodes.

```
struct node {  
    int data;  
    struct node *leftChild;  
    struct node  
    *rightChild;  
};
```

Since trees are recursively defined, it's very common to write routines that operate on trees that are themselves recursive.

So for instance, if we want to calculate the height of a tree, that is the height of a root node, We can go ahead and recursively do that, going through the tree. So we can say:

- For instance, if we have a nil tree, then its height is a 0.
- Otherwise, We're 1 plus the maximum of the left child tree and the right child tree.
- So if we look at a leaf for example, that height would be 1 because the height of the left child is nil, is 0, and the height of the nil right child is also 0. So the max of that is 0, then 1 plus 0.

Since trees are recursively defined, it's very common to write routines that operate on trees that are themselves recursive.

So for instance, if we want to calculate the height of a tree, that is the height of a root node, We can go ahead and recursively do that, going through the tree. So we can say:

- For instance, if we have a nil tree, then its height is a 0.
- Otherwise, We're 1 plus the maximum of the left child tree and the right child tree.
- So if we look at a leaf for example, that height would be 1 because the height of the left child is nil, is 0, and the height of the nil right child is also 0. So the max of that is 0, then 1 plus 0.

Height(tree) algorithm

if tree = nil:

 return 0

return 1 + Max(Height(tree.left),Height(tree.right))

Size(tree)algorithm:

1. If tree is empty then return 0

2. Else

 (a) Get the size of left subtree recursively i.e., call
 size(tree->left-subtree)

 (a) Get the size of right subtree recursively i.e., call
 size(tree->right-subtree)


```
tree_size = size(left-subtree) + size(right-  
subtree) + 1
```

(d) Return tree_size

Methods for Calculating Time Complexity

To calculate time complexity, you must consider each line of code in the program. Consider the multiplication function as an example. Now, calculate the time complexity of the multiply function:

```
1.  mul <- 1  
2.  i <- 1  
3.  While i <= n do  
4.      mul = mul * i  
5.      i = i + 1  
6.  End while
```

Let $T(n)$ be a function of the algorithm's time complexity. Lines 1 and 2 have a time complexity of $O(1)$. Line 3 represents a loop. As a result, you must repeat lines 4 and 5 $(n - 1)$ times. As a result, the time complexity of lines 4 and 5 is $O(n)$.

Finally, adding the time complexity of all the lines yields the overall time complexity of the multiply function $fT(n) = O(n)$.

Methods for Calculating Space Complexity

With an example, you will go over how to calculate space complexity in this section. Here is an

```
1.  int mul, i  
2.  While i <= n do  
3.      mul <- mul *  
4.      array[i] i <- i + 1  
5.  end while  
6.  return  
   mul
```

example of computing the multiplication of array elements:

Let $S(n)$ denote the algorithm's space complexity. In most systems, an integer occupies 4 bytes of memory. As a result, the number of allocated bytes would be the space complexity.

Line 1 allocates memory space for two integers, resulting in $S(n) = 4$ bytes multiplied by $2 = 8$ bytes. Line 2 represents a loop. Lines 3 and 4 assign a value to an already existing variable. As a result, there is no need to set aside any space. The return statement in line 6 will allocate one more memory case. As a result, $S(n) = 4 \text{ times } 2 + 4 = 12$ bytes.

Because the array is used in the algorithm to allocate n cases of integers, the final space complexity will be $fS(n) = n + 12 = O(n)$.

Conclusion: Thus we have implemented tree concept and calculated time and space

Experiment No. 4(B)

Problem Statement: Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree i. Insert new node, ii. Find number of nodes in longest path from root, iii. Minimum data value found in the tree, iv. Change a tree so that the roles of the left and right pointers are swapped at every node, v. Search a value.

Pre-requisite:

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

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

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

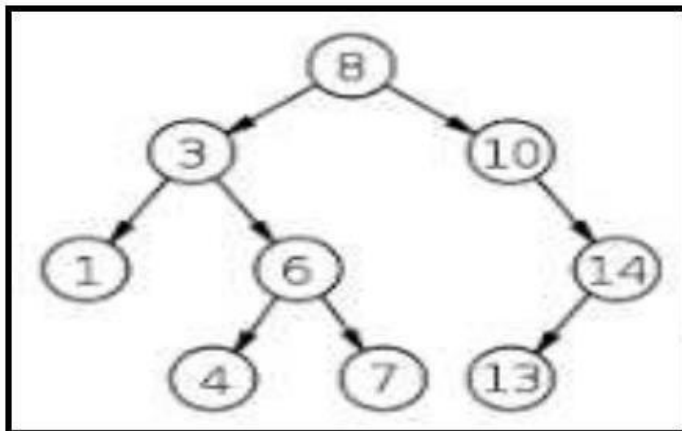
Theory:

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

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

Operations on Binary search Tree :**1. Insertion**

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right sub trees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root



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

2. Searching :

Searching a binary tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method. We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals

the root, the search is successful. If the value is less than the root, search the left subtree.

Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

3. Deletion :

There are three possible cases to consider:

1. Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree.
 2. Deleting a node with one child: Remove the node and replace it with its child.
 3. Deleting a node with two children: Call the node to be deleted N. Do not delete N. Instead, choose either its in-order successor node or its in-order predecessor node, R. Replace the value of N with the value of R, then delete R.
- As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

Algorithms for Operations on binary search tree:

// Create binary search tree

1. Create ()

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

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

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

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

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

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

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

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

// Traverse Binary search tree using Recursive Inorder Traversal

2. Inorder (node *temp)

Step 1: Check whether temp==NULL

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

Step 3: Display the data and in the temp node.

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

//Traverse Binary search tree using Recursive Preorder Traversal

3. preorder (node *temp)

Step 1: Check whether temp==NULL

Step 2: If not then Display the data in the temp node.

Step 3: Call function preorder (temp->left).

Step 4: Call the function preorder (temp->right).

//Traverse Binary search tree using Recursive Postorder Traversal

4. postorder (node *temp)

Step 1: Check whether temp==NULL

Step 2: If not then Call function postorder(temp- >left). Step 3: Call the function postorder (temp->right).

Step 4: Display the data in the temp node.

// Search an element from binary search Tree

5. search ()

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

Step 2: Store the address of the root node in the current code pointer (cn).

Step 3: Traverse the tree until the required data and the data in the cn matches.

If the required data is greater than the word in the cn then perform $cn = cn \rightarrow \text{right}$
else

Perform $cn = cn \rightarrow \text{left}$

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

Step 4: Check whether the cn is NULL, if it is then required data is not found in the tree
else the required data is present in the tree.

Time complexity:

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

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

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

Conclusion:

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

EXPERIMENT NO: 5 (B)**Problem Statement:**

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

Objectives:

1. Understand the concept of expression tree
2. Understand the traversal methods of tree for expressions.

Outcomes:

1. Will be able to use concept of expression tree
2. Will be able to use traversal methods of tree for expressions.

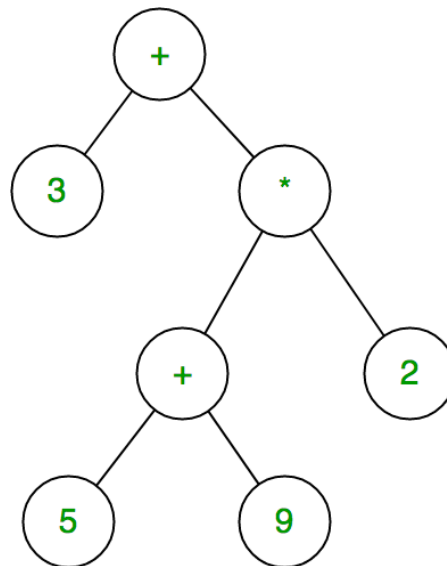
Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

An expression tree is basically a binary tree which is used to represent expressions. In an expression tree, internal nodes correspond to operators and each leaf nodes correspond to operands.

E.g. 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:



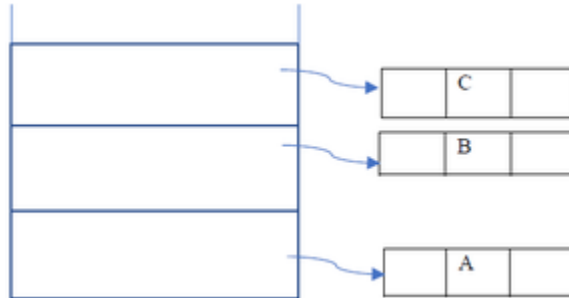
In expression trees, leaf nodes are operands and non-leaf nodes are operators. That means an expression tree is a binary tree where internal nodes are operators and leaves are operands. An expression tree consists of binary expressions. But for a unary operator, one subtree will be empty.

Construction of Expression Tree:

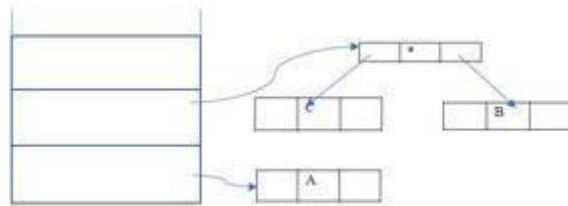
- The user will provide a postfix expression for which the program will construct the expression tree.
- Inorder traversal of binary tree/expression tree will provide Infix expression of the given input.

Example:**Input:** A B C*+ D/**Output:** A + B * C / D

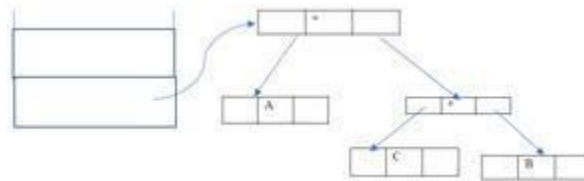
Step 1: The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



Step 2: In the Next step, an operator '*' will going read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack

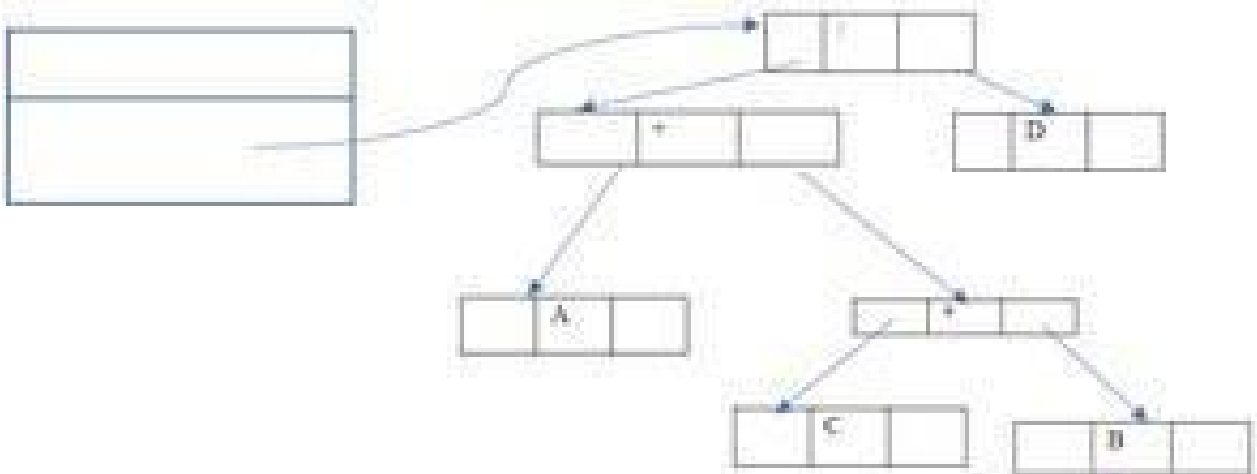


Step 3: In the Next step, an operator '+' will read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



Step 4: Similarly, as above cases first we push 'D' into the stack and then in the last step first, will read '/' and then as previous step topmost element will pop out and then will be right subtree of root '/' and other node will be right subtree.

Final Constructed Expression Tree is:

**Algorithm:**

Begin

class ExpressionTree which has following functions:

function push() to push nodes into the tree:

If stack is null

then push the node as first element

Else

push the node and make it top

function pop() to pop out nodes from the tree:

If stack is null

then print underflow

Else

Pop out the node and update top

function insert() to insert characters:

If it is digit

then push it.

Else if it is operator

Then pop it.

Else

Print "invalid Expression"

function postOrder() for postorder traversal:

If tree is not empty

postOrder(ptr->l)

postOrder(ptr->r)

Print root as ptr->d

function inOrder() for inorder traversal:

If tree is not empty

inOrder(ptr->l)

Print root as

ptr->d

inOrder(ptr->r)

function preOrder() for preorder traversal:

If tree is not empty

Print root as

ptr->d

preOrder(ptr->l)

preOrder(ptr->r)

End

Infix Expression:

When you wish to print the infix expression, an **opening and closing parenthesis must be added at the beginning and end of each expression**. As each subtree of the expression tree represents the subexpression, you have to print the opening parenthesis at its start and closing parenthesis after iterating all of its children.

Pseudo Code

Algorithm **infix** (T)

if (T not empty)

if (T token is operator)

 print (open parenthesis)

end if

infix (T

left-subtree) print

(T token)

infix (T right-subtree)

if (T token is operator)

 print (close

parenthesis)

 end **if**

end **if**

end

infix

Postfix Expression:

Every postfix expression is created by postorder traversal of the binary expression tree. Remember that it **does not require any parenthesis**, unlike infix expression.

Pseudo Code

```
Algorithm postfix (T)
if (T not empty)
    postfix (T left-subtree)
    postfix (T
    right-subtree) print (T
    token)
end if
end postfix
```

Prefix Expression:

Similar to postfix expression, prefix expression is also created by preorder traversal of the binary expression tree. Also, it **does not require any parenthesis** just like postfix expression. Check out the pseudo-code for prefix expression below

Pseudo Code

```
Algorithm prefix (T)
if (T not empty)
    print (T
    token)
    prefix (T left subtree)
    prefix (T right subtree)
end if
end prefix
```

Application of Expression Tree:

- Expression trees enable the richer interaction with the function arguments
- It can be used to provide generic operators
- An expression tree is also used as the compiler
- It is used in dynamic LINQ sorting
- It is used as symbolic manipulators
- An expression tree is used as object cloning

Conclusion: Thus we have implemented expression tree for given expression and we learn various methods of tree traversal.

EXPERIMENT NO: 6 (C)**Problem Statement:**

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 directed and undirected graph.
2. To implement program to represent graph using adjacency matrix and list.

Outcomes:

Student able to implement program for graph representation.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

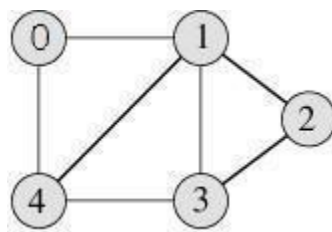
Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge.

The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: 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 and locale. See this for more applications of graph.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the 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 $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

The adjacency matrix for the above example graph is: Adjacency Matrix Representation

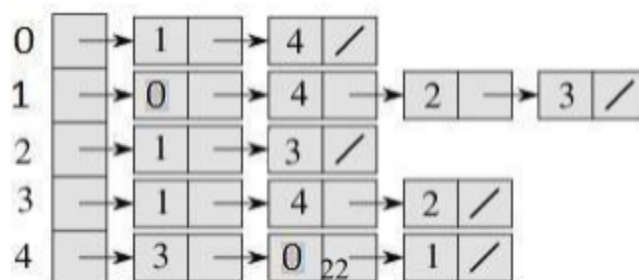
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

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.



Pros: Saves space $O(|V|+|E|)$. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

Conclusion: Thus we implemented program for graph presentation in adjacency matrix and list.

EXPERIMENT NO: 7 (C)**Problem Statement:**

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

Objectives:

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

Outcomes:

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

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

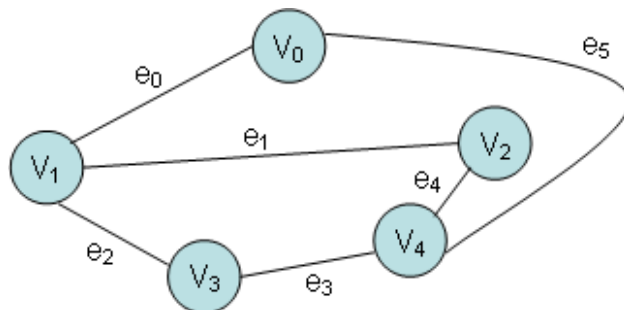
Graphs are the most general data structure. They are also commonly used data structures.

Graph definitions:

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

Undirected graph definition:

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

**Graph Implementation:**

Different kinds of graphs require different kinds of implementations, but the fundamental

concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed.

Representing Graphs with an Adjacency Matrix

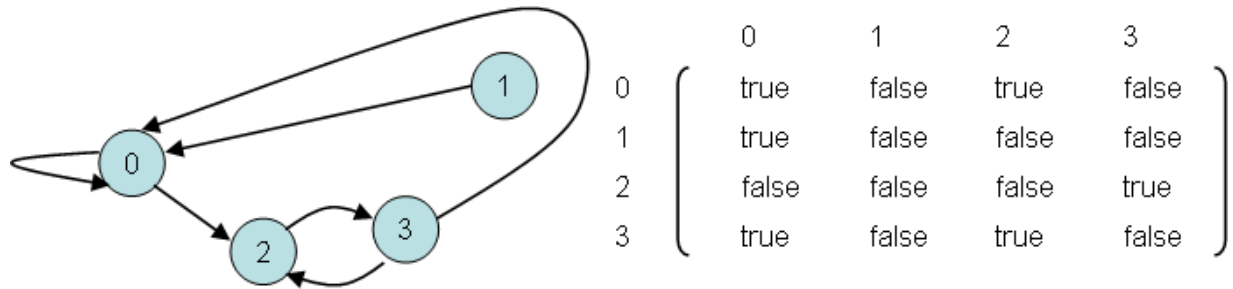


Fig: Graph and adjacency matrix

Definition:

☐ An adjacency matrix is a square grid of true/false values that represent the edges of a graph.

☐ If the graph contains n vertices, then the grid contains n rows and n columns.

☐ For two vertex numbers i and j , the component at row i and column j is true if there is an edge from vertex i to vertex j ; otherwise, the component is false.

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

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

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

Representing Graphs with Edge Lists

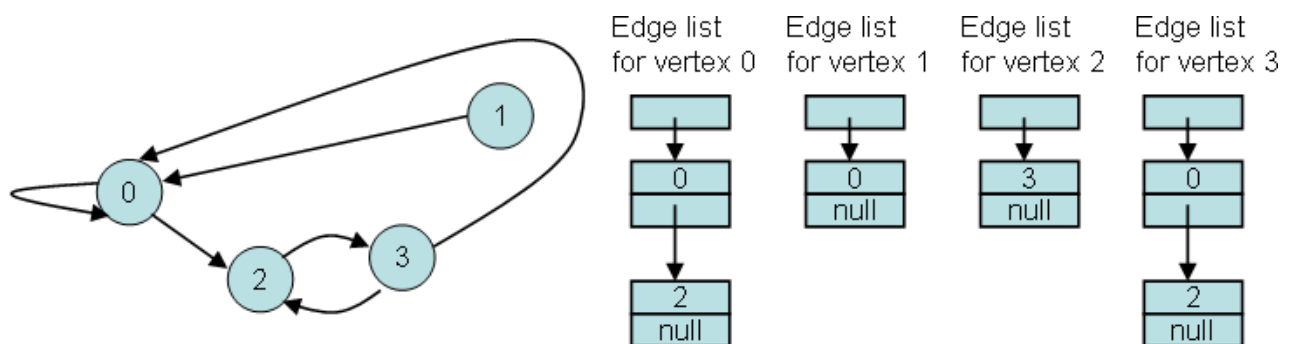


Fig: Graph and adjacency list for each node

Definition:

☐ A directed graph with n vertices can be represented by n different linked lists.

☐ List number i provides the connections for vertex i .

☐ For each entry j in list number i , there is an edge from i to j .

Loops and multiple edges could be allowed.

Representing Graphs with Edge Sets

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

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

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

Conclusion: Thus we have implemented adjacency matrix representation of the graph

EXPERIMENT NO: 8 (C)**Problem Statement:**

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

Objectives:

1. To understand minimum spanning tree of a Graph
2. To understand how Prim's algorithm works

Outcomes:

Students will be able to understand concept of minimum spanning tree with Prim's algorithm.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-**Data structures to be used:**

Array: Two dimensional array (adjacency matrix) to store the adjacent vertices & the weights associated edges. One dimensional array to store an indicator for each vertex whether visited or not.

```
#define max 20
```

```
int adj_ver[max][max]; int edge_wt[max][max]; int ind[max];
```

Concepts to be used:

- Arrays
- Function to construct head list & adjacency matrix for a graph.
- Function to display adjacency matrix of a graph.
- Function to generate minimum spanning tree for a graph using Prim's algorithm.

Spanning Tree:

A Spanning Tree of a graph $G = (V, E)$ is a sub graph of G having all vertices of G and no cycles in it.

Minimal Spanning Tree: The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph $G = (V, E)$ is called minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

- When a graph G is connected, depth first or breadth first search starting at any vertex visits all the vertices in G .

- The edges of G are partitioned into two sets i.e. T for the tree edges & B for back edges. T is the set of tree edges and B for back edges. T is the set of edges used or traversed during the search & B is the set of remaining edges.

- The edges of G in T form a tree which includes all the vertices of graph G and this tree is called as spanning tree.

Definition: Any tree, which consists solely of edges in graph G and includes all the vertices in G , is called as spanning tree. Thus for a given connected graph there are multiple spanning trees possible. For maximal connected graph having n vertices the number of different possible spanning trees is equal to n .

Cycle: If any edge from set B of graph G is introduced into the corresponding spanning tree T of graph G then cycle is formed. This cycle consists of edge (v, w) from the set B and all edges on the path from w to v in T .

- **Prim's algorithm:**

Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtech Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

An arbitrary node is chosen initially as the tree root (any node can be considered as root node for a graph). The nodes of the graph are then appended to the tree one at a time until all nodes of the graph are included. The node of the graph added to the tree at each point is that node adjacent to a node of the tree by an arc of minimum weight. The arc of minimum weight becomes a tree arc containing the new node to tree. When all the nodes of the graph have been added to the tree, a minimum spanning tree has been constructed for the graph.

Algorithm / Pseudo code:

- **Prim's Algorithm:**

All vertices of a connected graph are included in the minimum spanning tree. Prim's algorithm starts from one vertex and grows the rest of tree by adding one vertex at a time by adding associated edge in T . This algorithm iteratively adds edges until all vertices are visited.

```
void prims (vertex i)
```

1. Start
2. Initialize visited [] to 0 for $(i=0; i<n; i++)$
visited [i] = 0;
3. Find minimum edge from i for $(j=0; j<n; j++)$
{
if (min > a [i] [j])
{
min = a[i] [j];
x = i;
y = j;
}
}
}
4. Print the edge between i and j with weight.

5. Make visit [i++] = x

visit [j++] = y

6. Find next minimum edge starting from nodes of visit array.
7. Repeat step 6 until all the nodes are visited.
8. End.

Conclusion: Thus we have studied and implemented minimum spanning tree concept with the help of Prim's Algorithm.

EXPERIMENT NO: 9 (D)**Problem Statement:**

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?

Objectives:

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

Outcomes:

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

Software Requirements:

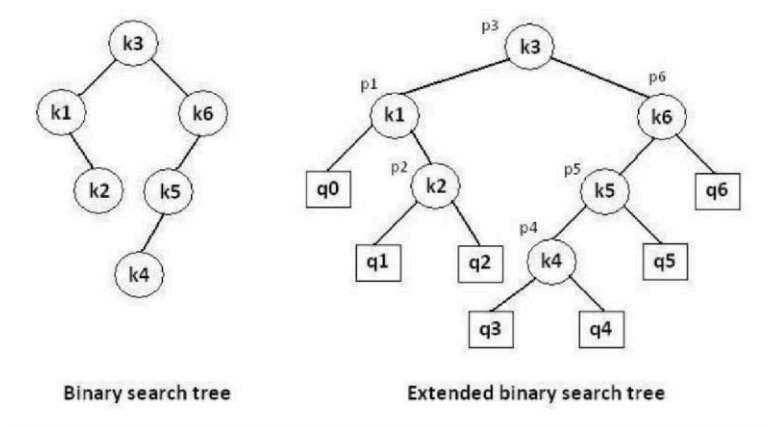
1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

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 k_1, k_2, \dots, 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 $k_1 < k_2 < \dots < k_n$.

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 ($p_1 \dots p_6$). 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 k_1 can be interpreted as representing all key values that are not stored and are less than k_1 . Similarly, the terminal node in the extended tree that is the right successor of k_n , represents all key values not stored in the tree that are greater than k_n . The terminal node that is successives between k_i and k_{i-1} in an inorder traversal represent all key values not stored that lie between k_i and k_{i-1} .

ALGORITHMS

We have the following procedure for determining $R(i, j)$ and $C(i, j)$ with $0 \leq i \leq j \leq n$:

PROCEDURE COMPUTE_ROOT($n, p, q; R, C$)

begin

for $i = 0$ to n do $C(i, i) \leftarrow 0$

$W(i, i) \leftarrow q(i)$ for $m = 0$ to n do

for $i = 0$ to $(n - m)$ do $j \leftarrow i + m$

$W(i, j) \leftarrow W(i, j - 1) + p(j) + q(j)$

*find $C(i, j)$ and $R(i, j)$ which minimize the tree cost

end

The following function builds an optimal binary search tree

FUNCTION CONSTRUCT(R, i, j)

begin

*build a new internal node N labeled (i, j) $k \leftarrow R(i, j)$ if $i = k$ then

*build a new leaf node N'' labeled (i, i) else

* $N'' \leftarrow \text{CONSTRUCT}(R, i, k)$

* N'' is the left child of node N if $k = (j - 1)$ then

*build a new leaf node N''' labeled (j, j) else

* $N''' \leftarrow \text{CONSTRUCT}(R, k + 1, j)$

* N''' is the right child of node N return

N end

COMPLEXITY ANALYSIS:

The algorithm requires $O(n^2)$ time and $O(n^2)$ 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.

Conclusion: Thus we have studied and implemented OBST concept with extended binary

EXPERIMENT NO: 10 (D)**Problem Statement:**

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

Objectives:

1. To understand concept of height balanced tree data structure.
2. To understand procedure to create height balanced tree.

Outcomes:

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

Software Requirements:

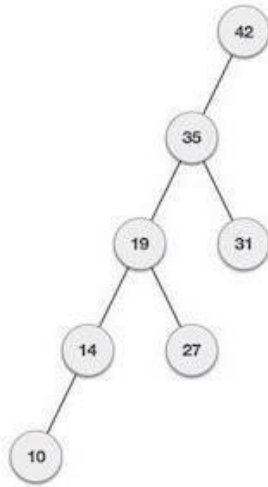
1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

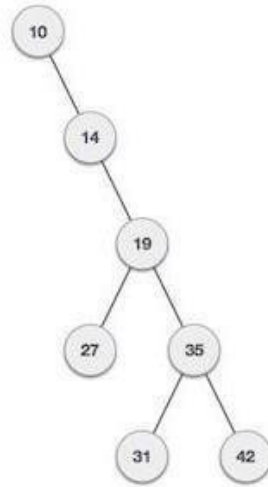
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if its balance factor is 0, 1, -1.

AVL (Adelson- Velskii and Landis) Tree: A balance binary search tree. The best searchtime, 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.

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

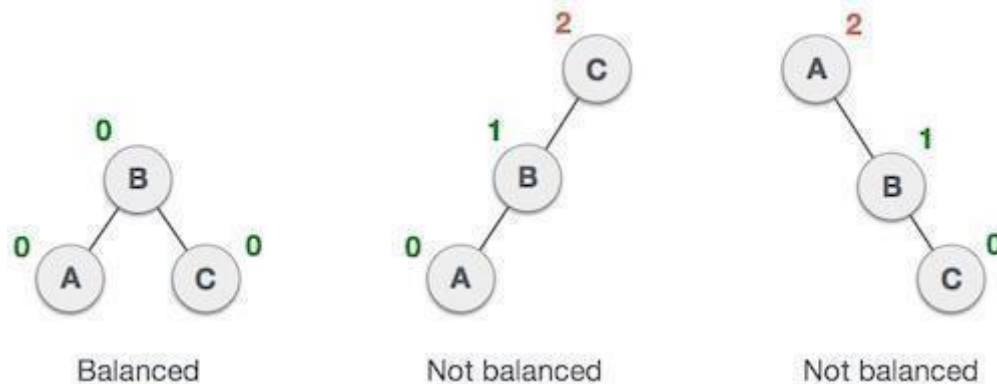


If input 'appears' in non-decreasing manner

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.

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

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

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

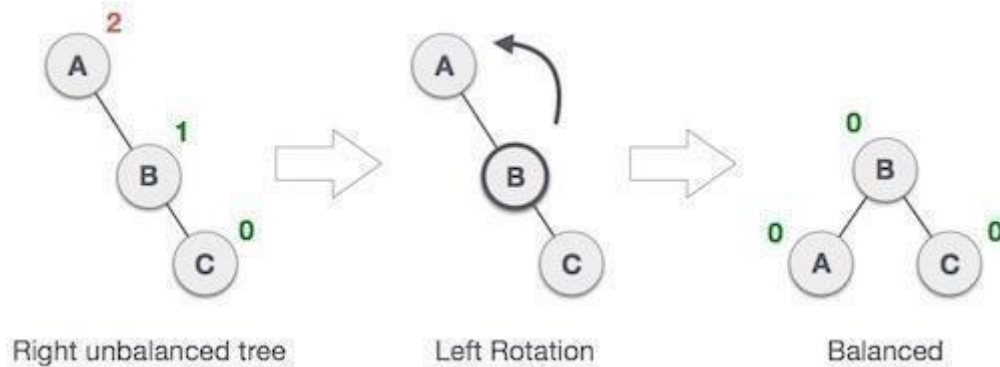
The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

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 –

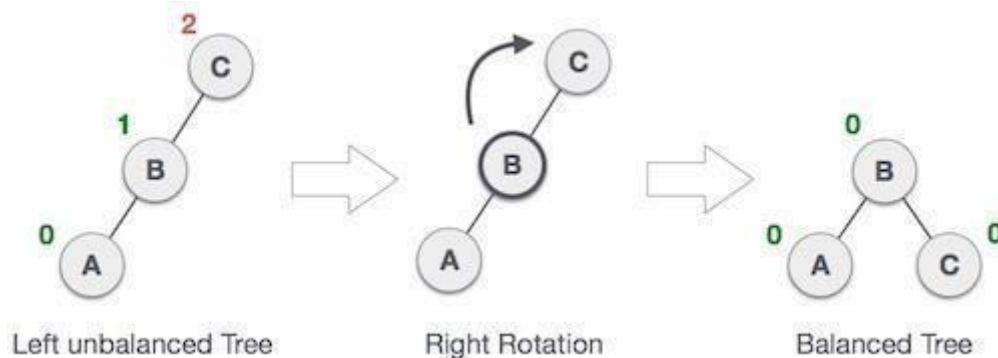
In our example, node **A** has become unbalanced as a node is inserted in the right subtree of

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.

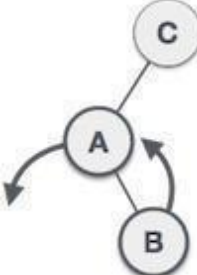
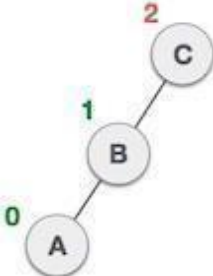
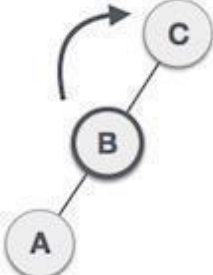
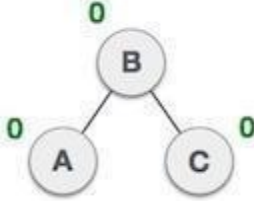


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

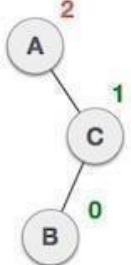
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right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

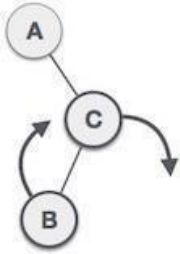
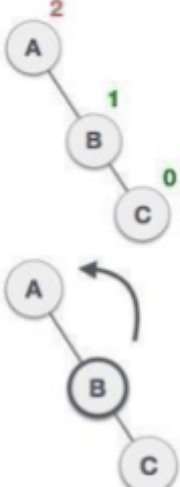
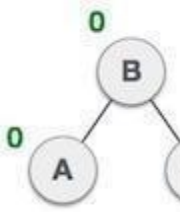
State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>

	<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>

Algorithm AVL TREE:

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 $x > 1 \Rightarrow x < P$ -

>elementa.) 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->left
 ->element P = singlerotateleft(P)
 else
 P = doublerotateleft(P)
3. else if x < P -
 >element
 a.) insert(x, P->right)
 b.) if height(P->right) - height(P->left) = 2 if (x < P->right)
 ->element P = singlerotateright(P)
 else
 P = doublerotateright(P)
4. else
 Print already exists
5. int m, n, d.
6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->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);

Conclusion: Thus we have studied and implemented concept of AVL Tree with its operations.

EXPERIMENT NO: 11 (E)**Problem Statement:**

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

Objectives:

1. To understand concept of heap
2. To understand concept & features like max heap, min heap.

Outcomes:

1. Define class for heap using Object Oriented features.
2. Analyze working of functions.

Software Requirements:

4. 64-bit Open source Linux or its derivative
5. Open Source C++ Programming tool like G++/GCC.
6. Turbo C++ compiler

Theory:-

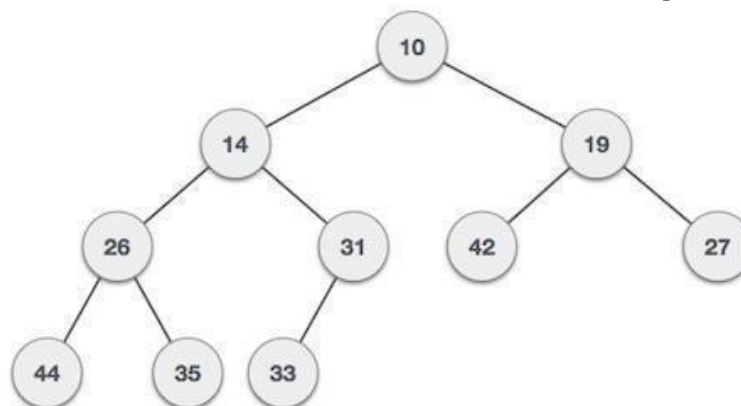
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then –
 $\text{key}(\alpha) \geq \text{key}(\beta)$

As the value of parent is greater than that of child, this property generates

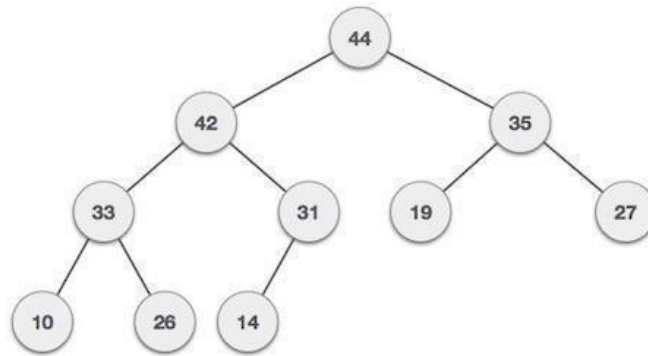
Max Heap. Based on this criteria, a heap can be of two types –

For Input $\rightarrow 35\ 33\ 42\ 10\ 14\ 19\ 27\ 44\ 26\ 31$

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



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



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

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

Step 2 – Assign new value to the node.

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

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

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

Note – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

INPUT: 35, 33, 42, 10, 14, 19, 27, 44, 16, 31

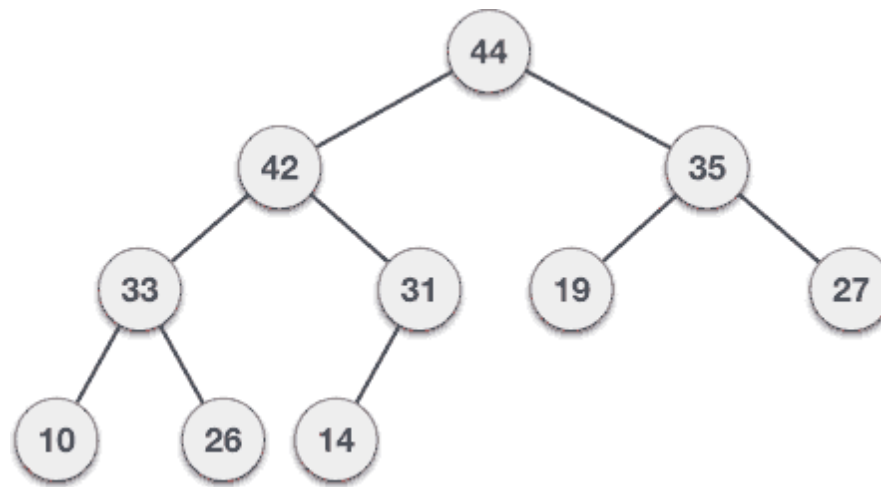
Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 – Remove root node.

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

Step 3 – Compare the value of this child node with its parent. **Step 4** – If value of parent is less than child, then swap them. **Step 5** – Repeat step 3 & 4 until Heap property holds.



Conclusion: Thus we have studied and implemented concept of heap data structure using Object Oriented features.

EXPERIMENT NO: 12 (F)**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.

Objectives:

1. To understand concept of file organization in data structure.
2. To understand concept & features of sequential file organization.

Outcomes:

1. Define class for sequential file using Object Oriented features.
2. Analyze working of various operations on sequential file .

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

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

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

Some of the file organizations are

1. Sequential File Organization
2. Heap File Organization
3. Hash/Direct File Organization
4. Indexed Sequential Access Method
5. B+ Tree File Organization
6. Cluster File Organization

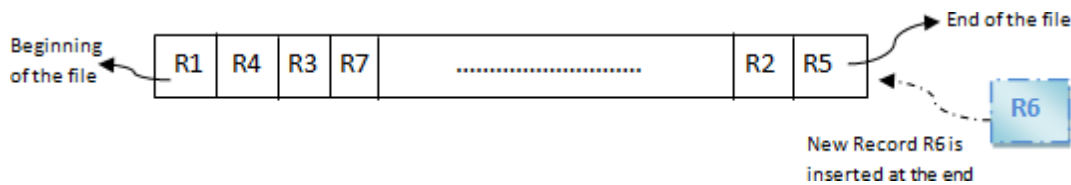
Sequential File Organization:

It is one of the simple methods of file organization. Here each file/records are stored one after the other in a sequential manner. This can be achieved in two ways:

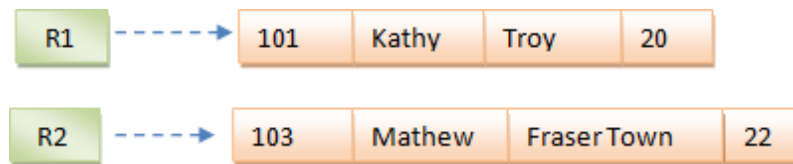
☐ Records are stored one after the other as they are inserted into the tables. This method is called pile file method. When a new record is inserted, it is placed at the end of the file. In the case of any modification or deletion of record, the record will be searched in the memory blocks. Once it is found, it will be marked for deleting and new block of record is entered.



Inserting a new record:

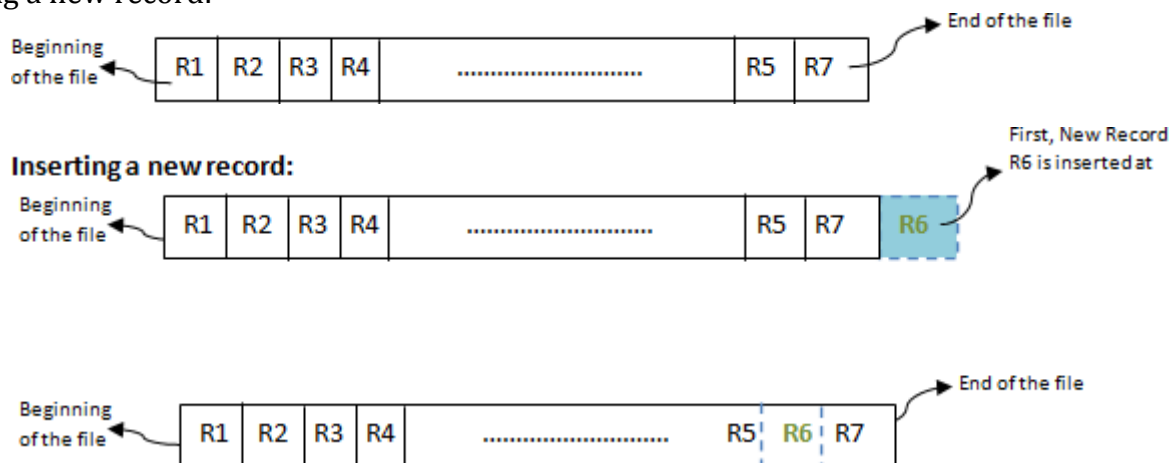


In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.



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

Inserting a new record:



Advantages:

1. Simple to understand.
2. Easy to maintain and organize
3. Loading a record requires only the record key.
4. Relatively inexpensive I/O media and devices can be used.
5. Easy to reconstruct the files.
6. The proportion of file records to be processed is high.

Disadvantages:

1. Entire file must be processed, to get specific information.
2. Very low activity rate stored.
3. Transactions must be stored and placed in sequence prior to processing.
4. Data redundancy is high, as same data can be stored at different places with different keys.

Conclusion: Thus we have studied and implemented concept of file organization in data structure

EXPERIMENT NO: 13 (F)**Problem Statement:**

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

Objectives:

1. To understand concept of file organization in data structure.
2. To understand concept of index sequential file organization.

Outcomes:

Student will be able to understand concept of file organization in data structure.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

A *file* is the basic entity for permanent retention of data in secondary storage devices such as hard drives, CDs, DVDs, flash drives, and tapes. But, the problem is that the common file imposes no structure of data storage and is not fit for high level-data processing. Files store a sequence of data as raw data bytes.

In C++, typically a file is opened if it already exists or created as an object associated to the stream is created. The most common stream objects in C++ are: *cin*, *cout*, *cerr*, and *clog*. These objects are created as we include the *iostream* header into a program. The *cin* object associates with the standard input stream, *cout* with the standard output stream, and *cerr* and *clog* with the standard error stream.

File Processing: There are two indispensable headers that must be included to perform file processing in C++: `<iostream>` and `<fstream>`. There are three important stream class templates in the `<fstream>` header: the *basic_ifstream* for performing file input, *basic_ofstream* for file output, and *basic_fstream* for both file input and output. .

Note that the *ifstream*, *ofstream*, and *fstream* objects that we often use in a C++ program is nothing but a specialization of *basic_ifstream*, *basic_ofstream*, and *basic_fstream*, respectively. The `<fstream>` library provides these *typedef* aliases for the basic template specialization classes to provide convenience by performing character I/O to and from files.

Sequential Files: A file created with the help of C++ standard library functions does not impose any structure on how the data is to be persisted. However, we are able to impose structure programmatically according to the application requirement. Suppose a minimal “payroll” application stores an employee record in a sequential file as employee id, name of the employee, and salary. The data obtained for each employee constitutes a record. The records are stored and written to the file sequentially and retrieved or read from the file in the same manner.

Opening a File: Here, we have opened the file using an instance of the *ofstream* class using two arguments: filename and opening mode. A file can be opened in different modes, as shown in the following table.

<i>File Opening mode</i>	<i>Description</i>
ios::in	Open file for reading data.
ios::out	Creates file if it does not exist. All existing data is erased from the file as new data is written into the file.
ios::app	Creates file if it does not exist. New data is written at the end of the file without disturbing the existing data.
ios::ate	Open file for output. Data can be written anywhere in the file, similar to append.
ios::trunc	Discard file content, which is the default action for ios::out.
ios::binary	Open file in binary (non-text) mode.

There is an *open* function that also can be used to open a file and set the mode. This is particularly useful when we first create an *ofstream* object and then open it as follows:

```
ofstream ofile;      ofile.open(filename, ios::out);
```

After creating an *ofstream* or *ifstream* object, it typically is checked whether it has successfully opened or not. We do this with the following code:

```
if(!fin){ cerr<<"Cannot open file for writing"<<endl;      exit(EXIT_FAILURE); }
```

Data Processing

Data processing occurs after a file has been opened successfully. The stream insertion (>>) and extraction (<<) operators are overloaded for convenient writing and reading data to and from a file. The file we have created is a simple text file and can be viewed by any text editor.

Closing a File: A file opened must be closed. In fact, as we invoke the *close* operation associated with the *ofstream* or *ifstream* object, it invokes the destructor, which closes the file. The operation is automatically invoked as the stream object goes out of scope. As a result, an explicit invocation of the *close* function is optional, but still is good programming practice.

Conclusion: Thus we have studied and implemented concept of index sequential file organization.

EXPERIMENT NO: 14

Problem Statement: Mini Project