DR. D.Y. PATIL INSTITUTE OF TECHNOLOGY, PUNE

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

LAB MANUAL

Data Structures and Algorithm Laboratory

Subject Code: 217532

Prepared By:

Priyanka Gupta

DR. D.Y. PATIL INSTITUTE OF TECHNOLOGY, PUNE

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Lab Manual

Second Year Engineering

Semester-IV

Data Structures and Algorithm Laboratory

Subject Code: 217532

Class: SE AI&DS

Academic year 2023-24

Data Structures and Algorithm Laboratory

Subject Code: 317534

Teaching Scheme	Credit Scheme	Examination Scheme and Marks
Practical: 04 Hours/Week	02	Term Work: 25 Marks Practical: 50 Marks

Guidelines for Instructor's Manual

The instructor‘s manual is to be developed as a hands-on resource and reference. The instructor's manual need to include prologue (about University/program/ institute/ department/foreword/ preface), curriculum of course, conduction and Assessment guidelines, topics under consideration-concept, objectives, outcomes, set of typical applications/assignments/ guidelines, and references.

Guidelines for Student's Laboratory Journal

The laboratory assignments are to be submitted by student in the form of journal. Journal consists of prologue, Certificate, table of contents, and handwritten write-up of each assignment (Title, Objectives, Problem Statement, Outcomes, software and Hardware requirements, Date of Completion, Assessment grade/marks and assessor's sign, Theory- Concept in brief, algorithm, flowchart, test cases, Test Data Set(if applicable), mathematical model (if applicable), conclusion/analysis. Program codes with sample output of all performed assignments are to be submitted as softcopy.

As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to journal may be avoided. Use of DVD containing students programs maintained by Laboratory In-charge is highly encouraged. For reference one or two journals may be maintained with program prints at Laboratory.

Guidelines for Laboratory / Term Work Assessment

Continuous assessment of laboratory work should be done based on overall performance and Laboratory assignments performance of student. Each Laboratory assignment assessment should be assigned

grade/marks based on parameters with appropriate weightage. Suggested parameters for overall assessment as well as each Laboratory assignment assessment include- timely completion, performance, innovation, efficient codes, punctuality and neatness.

Guidelines for Laboratory Conduction

The instructor is expected to frame the assignments by understanding the prerequisites, technological aspects, utility and recent trends related to the topic. The assignment framing policy need to address the average students and inclusive of an element to attract and promote the intelligent students. The instructor may set multiple sets of assignments and distribute among batches of students. It is appreciated if the assignments are based on real world problems/applications. Encourage students for appropriate use of Hungarian notation, proper indentation and comments. Use of open source software is to be encouraged. In addition to these, instructor may assign one real life application in the form of a mini-project based on the concepts learned.

Instructor may also set one assignment or mini-project that is suitable to respective branch beyond the scope of syllabus.

Set of suggested assignment list is provided in groups- A, B, C, D, E, F and G. Each student must perform at least 12 assignments(at least 02 from group A, 03 from group B, 02 from group C, 2 from group D, 01 from group E, 02 from group F.)

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

Programming tools recommended: - Open Source Python - Group A assignments, C++ Programming tool like G++/GCC

Virtual Laboratory:

http://cse01-iiith.vlabs.ac.in/Courses%20Aligned.html?domain=Computer%20Science

ASSIGNMENT NO : 1

TITLE: IMPLEMENTATION OF COLLISION RESOLUTION STRATEGIES

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

OBJECTIVE:

1. Learn Hashing techniques
2. Learn how to solve the collision problem in hashing

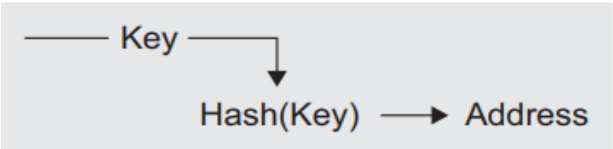
PREREQUISITE: -

1. Basic of Python Programming

THEORY:

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 hash table is an array-based structure used to store pairs. In this chapter, we will learn about hashing, hash functions, and other related aspects.

Hash functions transform a key into an address. Hashing is a technique used for storing and retrieving information associated with it that makes use of the individual characters or digits in the key itself



Key Terms:

Hash table: Hash table is an array [0 to Max − 1] of size Max.

Hash function: Hash function is one that maps a key in the range [0 to Max − 1], the result of which is used as an index (or address) in the hash table for storing and retrieving records

Bucket: A bucket is an index position in a hash table that can store more than one record

Probe: Each action of address calculation and check for success is called as a probe.

Collision: The result of two keys hashing into the same address is called collision.

Synonym: Keys that hash to the same address are called synonyms.

Collision Resolution Strategies:

1. Open Hashing
2. Separate Chaining or Linked List
3. Closed Hashing:
4. Linear Probing
5. Quadratic Probing
6. Double Hashing
7. Rehashing

Here we have implemented Linear Probing and Quadratic probing.

Linear Probing: A hash table in which a collision is resolved by placing the item in the next empty place following the occupied place is called linear probing. This strategy looks for the next free location until it is found.

(Hash(x)+ i) MOD Max

Initially i = 1, if the location is not empty then it becomes 2, 3, 4, …, and so on till an empty location is found.

Example: Linear probing without replacement For linear probing without replacement when collision occurs, if the location is occupied, the next empty location is linearly probed for synonyms.

Store the following data into a hash table of size 10 and bucket size 1. Use linear probing for collision resolution. 12, 01, 04, 03, 07, 08, 10, 02, 05, 14 Assume buckets from 0 to 9 and bucket size = 1 using hashing function key % 10.

Bucket	Initially empty	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	Insert 05	Insert 14
0								10	10	10	10
1			01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	12	12	12
3					03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	04
5									02	02	02
6										05	05
7						07	07	07	07	07	07
8							08	08	08	08	08
9											14

Quadratic Probing:

In quadratic probing, we add the offset as the square of the collision probe number. In quadratic probing, the empty location is searched by using the following formula:

(Hash(Key) + i 2) MO D Max where i lies between 1 and (Max - 1)/2

Example:

Consider the keys 22, 17, 32, 16, 5, and 24. Let Max = 7. Let us use quadratic probing to handle synonyms.

Index	Key		Index	Key
0			0	24
1	22		1	22
2	16	insert 24 →	2	16
3	17		3	17
4	32		4	32
5	5		5	5
6			6	

We can see that while inserting 24, the address we get is Hash(24) = 24 MOD 7 = 3

It is also noted that the location 3 is already occupied.

We may now go for the quadratic function as

[Hash(24) - (1)2 MOD 7] = (24 MOD 7) + 1 MOD 7 = (3 + 1) MOD 7 = 4 which is occupied.

Hence, Hash(24) + (2)2 MOD 7 = (3 + 4) MOD 7 = 0 which is empty, so store 24 there.

ALGORITHM:

Function linear_probe(key, value):

```
i = key
for i in range(key, size):
    key1 = (i + 1) % size  # Calculating next empty location address using linear probing formula
    if hashTable[key1] == -1:  # If location is empty, insert data
        hashTable[key1] = value
        print(value, "inserted at arr", key1)
        break
```

Function quadratic_probe(key, value):

```
for i in range(key, size):
    key1 = (key + i*i) % size  # Quadratic Probing formula
    if hashTable[key1] == -1:
        hashTable[key1] = value
        print(value, "inserted at arr", key1)
        break
```

Main Function :

```
size = 7
hashTable = array('i', [-1] * size)

# Example usage of linear probing
key = <some_key>
value = <some_value>
linear_probe(key, value)

# Example usage of quadratic probing
key = <some_key>
value = <some_value>
quadratic_probe(key, value)
```

CONCLUSION:

Hence we have implemented Collision resolution strategies for telephone book records of N Clients.

ASSIGNMENT QUESTION :

1. What is Hashing?
2. What is Hash Function?
3. What are the types of collision handling strategies?
4. Explain collision handling strategies with example.

ASSIGNMENT NO: 2

TITLE: CREATE ADT THAT IMPLEMENT THE "SET" CONCEPT.

PROBLEM STATEMENT: -

Create ADT that implements the "set" concept and perform following operations:

1. Add (new Element) -Place a value into the set,
2. Remove (element) Remove the value
3. Contains (element) Return true if element is in collection,
4. Size () Return number of values in collection Iterator () Return an iterator used to loop over collection,
5. Intersection of two sets,
6. Union of two sets,
7. Difference between two sets,
8. Subset

OBJECTIVE:

- To understand set ADT.
- To perform various operations on set.

PREREQUISITE: -

1. Basic of Python Programming

THEORY:

What is ADT?

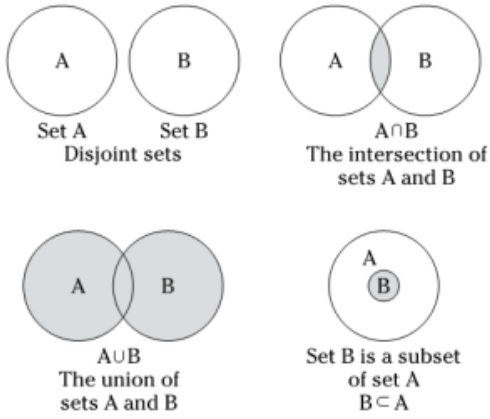
An ADT is like an interface that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies what each operation does, but not how it does it. Typically, an ADT can be implemented using one of many different data structures. A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.

In general, the steps of building ADT to data structures are:

1. Understand and clarify the nature of the target information unit.
2. Identify and determine which data objects and operations to include in the models.

3. Upon finalized specification, write necessary implementation. This includes storage scheme and operational detail. Operational detail is expressed as separate functions (methods).

Set : A set is an abstract data type that can store unique values, without any particular order. A set is a collection of unique data. That is, elements of a set cannot be duplicate. Their name derives from the mathematical concept of finite sets. Unlike an array, sets are unordered and unindexed. In computer science, set theory is useful if you need to collect data and do not care about their multiplicity or their order. Perhaps you recall learning about sets and set theory at some point in your mathematical education. Maybe you even remember Venn diagrams:



How to Create a Set in Python

There are following two ways to create a set in Python.

1. **Using curly brackets:** The easiest and straightforward way of creating a Set is by just enclosing all the data items inside the curly brackets {}. The individual values are comma-separated.
2. **Using set() constructor:** The set object is of type class 'set'. So we can create a set by calling the constructor of class ‘set’. The items we pass while calling are of the type iterable. We can pass items to the set constructor inside double-rounded brackets.

Let’s see each one of them with an example.

```
# create a set using {}
# set of mixed types intger, string, and floats
  sample_set = {'Mark', 'Jessa', 25, 75.25}
  print(sample_set)
# Output {25, 'Mark', 75.25, 'Jessa'}
# create a set using set constructor
# set of strings
  book_set = set(("Harry Potter", "Angels and Demons", "Atlas Shrugged"))
```

```
print(book_set)
# output {'Harry Potter', 'Atlas Shrugged', 'Angels and Demons'}
print(type(book_set))
# Output class 'set'
```

Operations:

Four operations are common on sets: adding elements, removing elements, determining how many elements are in the set, and asking whether a specific item is in the set.

The set has four basic operations:

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

Other operations on set:

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

Operations on Sets:

1. **Union of Sets:** Union of Sets A and B is defined to be the set of all those elements which belong to A or B or both and is denoted by A∪B.

   A∪B = {x: x ∈ A or x ∈ B}

   Example: Let A = {1, 2, 3}, B= {3, 4, 5, 6}

   A∪B = {1, 2, 3, 4, 5, 6}.

2. **Intersection of Sets:** Intersection of two sets A and B is the set of all those elements which belong to both A and B and is denoted by A ∩ B.

   A ∩ B = {x: x ∈ A and x ∈ B}

Example: Let A = {11, 12, 13}, B = {13, 14, 15} A ∩ B = {13}.

3. **Difference of Sets:** The difference of two sets A and B is a set of all those elements which belongs to A but do not belong to B and is denoted by A - B.
   A - B = {x: x ∈ A and x ∉ B}
   Example: Let A = {1, 2, 3, 4} and B = {3, 4, 5, 6} then A - B = {1, 2} and B - A = {5, 6}

4. **Subset of set:** A set A is a subset of another set B if all elements of the set A are elements of the set B. In other words, the set A is contained inside the set B. The subset relationship is denoted as A ⊆ B.
   A⊆B = {for each x of A: x ∈A and x ∈B}
   Example: Let A = {2, 3, 4} and B = {2,3, 4, 5, 6} then A is subset of B.

5. **Proper Subset of set:** A set A is a proper subset of set B if A is a subset of B and A is not equivalent to B. It is expressed as A⊂B.
   A⊂B = {for each x of A : x ∈A and x ∈B and B>A}
   Example: Let A = {2, 3, 4} and B = {2, 3, 4, 6} then A is subset of B.As A is not equal to B, A is proper subset of B.

ALGORITHMS:

Algorithm Union (set A, set B)

a) Initialize union U as empty.
b) Copy all elements of set A to U.
c) Do following for every element x of set B:
   - If x is not present in A, then copy x to U.
d) Return U.

Algorithm Intersection (set A, set B)

1) Initialize intersection set I as empty.
2) Do following for every element x of set A:
a) If x is present in set B, then copy x to I.
3) Return I.

Algorithm Difference (set A, set B)

1) Initialize difference set D as empty.

2) Do following for every element x of set A:

a) If x is not present in set B, then copy x to D.

3) Return D.

Algorithm Subset (set A, set B)

1) Do following for every element x of set A:

a) If x is present in set B, then return A is subset of B

else

return A is not subset of B

Algorithm Proper Subset (set A, set B)

1) Do following for every element x of set A:

a) If x is present in set B and B>A, then return A is proper subset of B

else

return A is not proper subset of B

CONCLUSION:

Thus we have successfully implemented Set ADT using python and perform various operations on it.

ASSIGNMENT QUESTION:

1. What is Set Data Structure?
2. Why there is a need for Set Data Structure?
3. Types of Set Data Structure?

ASSIGNMENT NO: 3

TITLE: TO STUDY BST AND OPERATIONS ON IT.

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

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

OBJECTIVE:

Students are able to understand implementation of Binary Search Tree using c++ programming.

PREREQUISITE: -

1. Basic of c++ programming
2. Concept of Binary Search Tree.

THEORY:

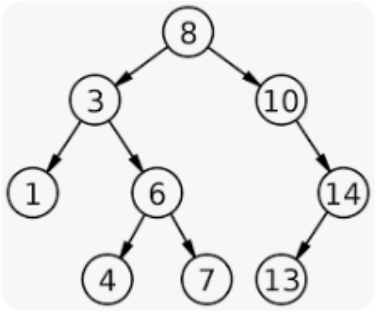
Binary Search Tree:

It is a binary Tree with following Properties.

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.
4. The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

Example:

Consider following binary search tree



OPERATIONS:

Operations on a binary search tree require comparisons between nodes.

1] Searching: Searching a binary search 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.

2] Insertion: Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

3] Deletion: There are three possible cases to consider:

Deleting a leaf (node with no children): Deleting a leaf is easy, as we can simply remove it from the tree. Deleting a node with one child: Remove the node and replace it with its child. Deleting a node with two children: Call the node to be deleted *N*. Do not delete *N*. Instead, choose **either its in-order successor node or its in-order predecessor node,** *R*. Replace the value of *N* with the value of *R*, then delete *R*. As with all binary trees, a node's in-order successor is the left-most child of its right 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.

ALGORITHM :

Algorithm CreateBinaryTree():

// This algorithm is used to create a binary tree with no input and return nothing.

```
Repeat
Read data element from the user into 'Value';
Create a new node;
temp = new node;
temp->data = value;
temp->left = NULL;
temp->right = NULL;
    If (root is NULL) then // Tree is Empty.
        root = temp;
    Else // Add a new node into the existing tree.
        Insert(root, temp);
    Until (false);
```

Algorithm Insert(root, temp):

// This algorithm is used to add a new node in an existing tree pointed to by root. It returns nothing.

```
If (root is not NULL)
 {
   If (root->data > temp->data) then // Left Subtree
   {
   If (temp->left is NULL) then // Left child is NULL
   temp->left = temp;
   Else
   Insert(temp->left, temp); // Check next left link
 }
Else
{
   If (ch is Right side) then // Right Subtree
   {
   If (temp->right is NULL) then // Right child is NULL
temp->right = temp;
```

```
        Else
        Insert(temp->right, temp); // Check next right link
        }
 }
}
```

Algorithm Preorder(temp):

// This algorithm is used to display the nodes in VLR (Value-Left-Right) way.

```
If (temp is not NULL) then
{
Write(temp->data);
        Call Preorder(temp->lptr);
        Call Preorder(temp->rptr);
   }
```

Algorithm Inorder(temp):

// This algorithm is used to display the nodes in LVR (Left-Value-Right) way.

```
    If (temp is not NULL) then
    {
        Call Inorder(temp->lptr);
        Write(temp->data);
        Call Inorder(temp->rptr);
    }
```

Algorithm Postorder(temp):

// This algorithm is used to display the nodes in LRV (Left-Right-Value) way.

```
    If (temp is not NULL) then
    {
        Call Postorder(temp->lptr);
        Call Postorder(temp->rptr);
        Write(temp->data);
    }
```

Algorithm Search(root, val, parent):

// This algorithm is used to search for an element in a Binary Search Tree (BST).

```
If (root is not NULL) then // Tree exists.
        {
                If (root->data > val) then // Search in the Left subtree.
                Search(root->llink, val, root);
                Else If (root->data < val) then // Search in the Right subtree.
                Search(root->rlink, val, root);
                If (root->data == val) then // Successful search.
                {
                        Write("Value is found successfully");
                        Return parent;
                }
        }
  Else // Unsuccessful search.
        Return NULL;
```

Algorithm Smallest(root):

// This algorithm finds the minimum data value found in the tree.

```
While (root->ln is not NULL)
    {
        root = root->ln;
    }
    Print root->key;
```

Algorithm LongestPath(root):

// This algorithm finds the number of nodes in the longest path in the tree.

```
If (root is NULL)
  Return 0;
  L = LongestPath(root->ln);
  R = LongestPath(root->rn);
If (L > R)
        Return (L + 1);
  Else
```

```
Return (R + 1);
```

Algorithm swapNodes(root):

// This algorithm changes a tree so that the roles of the left and right pointers are swapped at every node.

```
Node* temp;
If (root is NULL)
Return NULL;
temp = root->ln;
root->ln = root->rn;
root->rn = temp;
swapNodes(root->ln);
swapNodes(root->rn);
```

CONCLUSION:

In this way we have implemented of Binary Search Tree using c++ programming.

ASSIGNMENT QUESTION:

1. What is Binary Search Tree (BST)?

2. Define the property of the Binary Search Tree?

3. What is the difference between the Binary Search tree & Binary Tree?

ASSIGNMENT NO: 4

TITLE: TO CONSTRUCT AN EXPRESSION TREE FROM THE GIVEN PREFIX EXPRESSION

PROBLEM STATEMENT: -

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

PREREQUISITE: -

1. Basic of c++ programming
2. Concept of Binary Search Tree.

THEORY:

- **Binary Search Tree:**

  It is a binary Tree with following Properties.

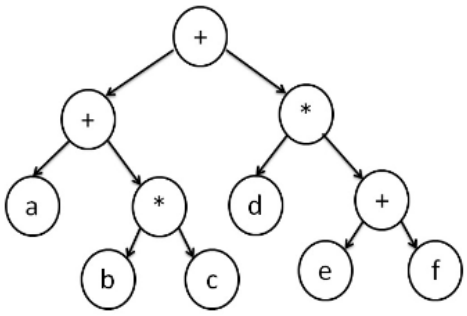
  1. The left subtree of a node contains only nodes with keys less than the node's key.
  2. The right subtree of a node contains only nodes with keys greater than the node's key.

3. Both the left and right subtrees must also be binary search trees.
4. The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

- **Expression Tree** :

  1. Expression Tree is used to represent expressions.
  2. An expression and expression tree shown below

a + (b * c) + d * (e + f)



Expression Tree

There are different types of expression formats:

- Prefix expression

- Infix expression and
- Postfix expression

Expression Tree is a special kind of binary tree with the following properties:

- Each leaf is an operand. Examples: a, b, c, 6, 100
- The root and internal nodes are operators. Examples: +, -, *, /, ^
- Subtrees are subexpressions with the root being an operator.

Traversal Techniques :

There are 3 standard traversal techniques to represent the 3 different expression formats.

Inorder Traversal

We can produce an infix expression by recursively printing out

- the left expression,
- the root, and
- the right expression.

Postorder Traversal

The postfix expression can be evaluated by recursively printing out

- the left expression,
- the right expression and
- then the root

Preorder Traversal

We can also evaluate prefix expression by recursively printing out:

- the root,
- the left expression and
- the right expression.

If we apply all these strategies to the sample tree above, the outputs are:

Infix expression:

(a+(b*c))+(d*(e + f))

Postfix Expression:

a b c * + d e f + * +

Prefix Expression:

+ + a * b c * d + e f

Construction of Expression Tree:

Let us consider a postfix expression is given as an input for constructing an expression tree. Following are the step to construct an expression tree:

1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and push a pointer onto a stack
4. If the symbol is an operator, pop two pointers from the stack namely T1 & T2 and form a new tree with root as the operator, T1 & T2 as a left and right child
5. A pointer to this new tree is pushed onto the stack

Thus, An expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right subtree

ALGORITHM :

Algorithm: Construct Expression Tree from Prefix Expression

Include Necessary Libraries:

1. Include iostream for input/output.
2. Include stack for stack data structure.
3. Include cctype for character type checking (isalpha).

Define TreeNode Structure:

1. Define a structure named TreeNode with character data, left, and right pointers.
2. Define Operator Checking Function:
   a. Define a function isOperator(c) that checks if a given character is an operator (+, -, *, /).

Construct Expression Tree Function:

Define a function constructExpressionTree(prefix) that takes a prefix expression as input.

1. Create an empty stack of TreeNode pointers.
2. Iterate through the characters of the prefix expression in reverse order:
3. If the current character is an operand (isalpha), create a TreeNode and push it onto the stack.
4. If the current character is an operator, create a TreeNode with the operator, pop two elements from the stack and set them as the left and right children of the new TreeNode. Push the new TreeNode onto the stack.
5. The top of the stack now contains the root of the expression tree. Return it.

Post-order Traversal Function:

Define a function postOrderTraversal(root) that performs post-order traversal of the expression tree.

1. Create two stacks: one for tree traversal and another to store the result characters.
2. Push the root onto the traversal stack.
3. While the traversal stack is not empty:
4. Pop a node from the traversal stack.
5. Push the data of the node onto the result stack.
6. If the node has a left child, push it onto the traversal stack.
7. If the node has a right child, push it onto the traversal stack.
8. Print the characters from the result stack in reverse order.

Delete Tree Function:

Define a function deleteTree(root) that deletes the entire expression tree.

1. Use a post-order traversal approach:
2. Create a stack and push the root onto it.
3. While the stack is not empty:
4. Pop a node from the stack.
5. If the node has a left child, push it onto the stack.
6. If the node has a right child, push it onto the stack.
7. Delete the current node.

Main Function:

In the main function:

1. Declare a prefix expression string, e.g., "+--a*bc/def".
2. Call constructExpressionTree to obtain the root of the expression tree.
3. Print the post-order traversal of the expression tree.

4. Call deleteTree to free up memory.

End of Algorithm.

CONCLUSION:

In this way we have Constructed an expression tree from the given prefix expression

ASSIGNMENT QUESTION:

1. Define a binary tree. With example show array and linked representation of binary tree?
2. Write an expression tree for an expression

a. A / B + C * D + E

b. ((6+(3-2)*5)^2+3)

3. Mention different types of binary trees and explain briefly.

ASSIGNMENT NO: 5

TITLE : TO STUDY DICTIONARY AND PERFORM DIFFERENT OPERATIONS ON IT USING BINARY SEARCH TREE.

PROBLEM STATEMENT: A Dictionary stores keywords & its meanings. Now provide facility for adding new keywords, deleting keywords, & updating values of any entry. Also provide facility to display whole data sorted in ascending/ Descending order, Also we have to find how many maximum comparisons may require for finding any keyword.

OBJECTIVE OF THE ASSIGNMENT: Students are able to implement Dictionary using BST concepts in C++

PREREQUISITE:

1. Basic of C++ Programming

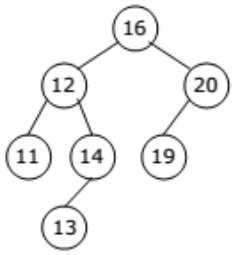
Theory:

BINARY SEARCH TREE:

In computer science, a binary search tree (BST), which may sometimes also be called an ordered or sorted binary tree, is a node-based binary tree data structure.

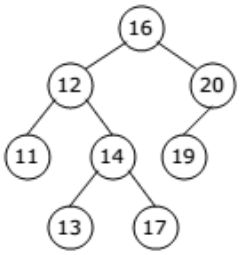
The Binary Search Tree which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.
- The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.



Binary Search Tree

(a)



Not a Binary Search Tree

(b)

ALGORITHM:

Algorithm: Dictionary Operations Using BST

1. Creating a Dictionary

1. Start the CreateDictionary function.
2. Use a do-while loop to repeatedly read input for creating a new node in the DICTIONARY.
3. Create a new node temp to store the key and meaning.
4. Prompt the user to enter the key and meaning.
5. Initialize left and right pointers of temp to nullptr.
6. Check if the BST is empty (root is nullptr).
7. If empty, set root to temp.
8. If not empty, call the Insert function with root and temp as arguments.

2. Inserting a New Node

1. Start the Insert function with parameters Node* root and Node* temp.
2. Check if the root is not nullptr.
3. If not nullptr, compare the lengths of the keys (l1 and l2) and determine the minimum length (l).
4. Iterate through the characters of the keys up to the minimum length.
5. Compare characters at the same position.
6. If the character in the root's key is greater, check the left subtree.
7. If the left child is nullptr, set temp as the left child.
8. If not nullptr, recursively call Insert with left child as the new root.
9. If the character in the root's key is lesser, check the right subtree.
10. If the right child is nullptr, set temp as the right child.
11. If not nullptr, recursively call Insert with right child as the new root.
12. If the root is nullptr, print "No tree exists."

3. Tree Traversal Algorithms

1. **Preorder Traversal:**
    a. Start the Preorder function with parameter Node* temp.
    b. If temp is not nullptr:
    c. Print the key of the current node.
    d. Recursively call Preorder for the left subtree.
    e. Recursively call Preorder for the right subtree.

2. **Inorder Traversal:**
   a. Start the Inorder function with parameter Node* temp.
   b. If temp is not nullptr:
   c. Recursively call Inorder for the left subtree.
   d. Print the key of the current node.
   e. Recursively call Inorder for the right subtree.
3. **Postorder Traversal:**
   a. Start the Postorder function with parameter Node* temp.
   b. If temp is not nullptr:
   c. Recursively call Postorder for the left subtree.
   d. Recursively call Postorder for the right subtree.
   e. Print the key of the current node.
4. **Printing the Dictionary**
   a. **In Ascending Order:**
      1. Start the InorderPrint function with parameter Node* temp.
      2. Call the Inorder function with temp as an argument.
   b. **In Descending Order:**
      1. Start the DescendingOrderPrint function with parameter Node* temp.
      2. Call the descending_order function with temp as an argument.

5. Searching an Element in BST

1. Start the Search function with parameters Node* root, Node* temp, and Node*& parent.
2. Check if the root is not nullptr.
3. If not nullptr, compare the lengths of the keys (l1 and l2) and determine the minimum length (l).
4. Iterate through the characters of the keys up to the minimum length.
5. Compare characters at the same position.
6. If the character in the root's key is greater, recursively call Search with the left subtree.
7. If the character in the root's key is lesser, recursively call Search with the right subtree.
8. If characters are equal and strings match, print "Value is found successfully" and return the parent node.
9. If the root is nullptr, return nullptr (indicating an unsuccessful search).

CONCLUSION:

Thus, We have studied and implemented the dictionary using Binary Search Tree and performed different operations on it.

ASSIGNMENT QUESTION:

1. List and discuss the additional operations of binary tree with appropriate examples.
2. Explain and write the analysis of searching and inserting a node in BST.
3. Describe the binary search tree with an example. Write the iterative and recursive function to search for a key value in a binary search tree .
4. Construct a binary search tree for the inputs

    i. 22, 14, 18, 50, 9, 15, 7, 6, 12, 32, 25

    ii. 14, 5, 6, 2, 18, 20, 16, -1, 21

ASSIGNMENT NO: 6

TITLE: IMPLEMENTATION OF GRAPH DATA STRUCTURES

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 takes to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.

OBJECTIVE:

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

PREREQUISITE:

1. Basic of C++ Programming
2. Concept of graph Data structure.

THEORY:

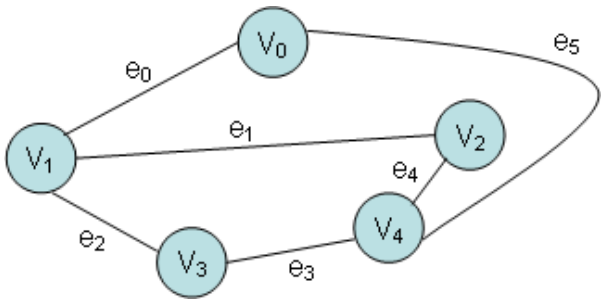
Graphs are the most general data structure. They are also commonly used data structures.

Graph definitions:

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

Undirected graph definition:

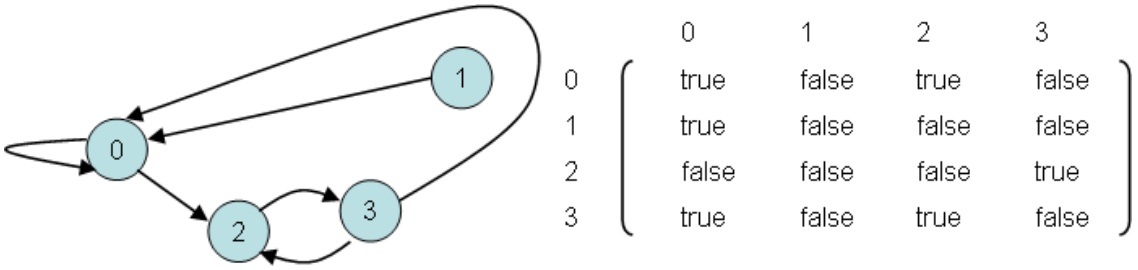
- An undirected graph is a set of nodes and a set of links between the nodes.
- Each node is called a **vertex**, each link is called an **edge**, and each edge connects twovertices.
- The order of the two connected vertices is unimportant.
- An undirected graph is a finite set of vertices together with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



	0	1	2	3
0	true	false	true	false
1	true	false	false	false
2	false	false	false	true
3	true	false	true	false

Fig: Graph and adjacency matrix

Definition:

- An adjacency matrix is a square grid of true/false values that represent the edges of agraph.
- 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 anedge 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

Representing Graphs with Edge Lists

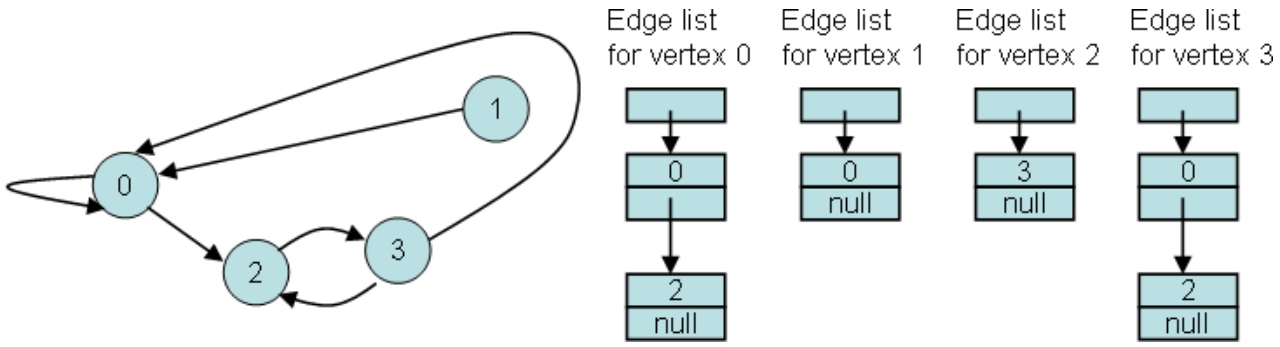


Fig: Graph and adjacency list for each node

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.

TRAVERSAL OF A GRAPH:

Let G = (V,E) be an undirected graph and a vertex v in V(G), we are interested in visiting all vertices in G that are reachable from vertex v i.e. all vertices connected to v. There are two ways to do this:

1) Depth First Search (DFS) and

2) Breadth First search (BFS)

DEPTH FIRST SEARCH:

DFS is a recursive algorithm in which we start from a given vertex v, next an unvisited vertex w adjacent to v is selected and Depth First Search from w is initiated. When a vertex u is reached such that all of it's adjacent vertices have been visited, we back up to the immediate previous vertex visited which has an unvisited vertex w adjacent to it and initiate a DFS from w and so on.

For example, if we start from vertex 1 in graph G1 and apply DFS procedure, the vertices will be visited

in following sequence: **1 2 4 8 5 6 3 7**

BREADTH FIRST SEARCH:

Starting at vertex v & making it as visited; Breadth First Search differs from Depth First Search in that all unvisited vertices adjacent to v are visited next. Then unvisited vertices adjacent to these vertices (visited in previous pass) are visited and so on. For example, if we start from vertex 1 in graph G1 and apply BFS procedure, the vertices will be visitedin following sequence: 1 2 3 4 5 6 7 8

ANALYSIS OF ALGORITHMS:

1) Time Complexity:

- For the construction of an undirected graph with 'n' vertices using adjacency matrix is **O (n^2),** since for every pair (i, j) we need to store either '0' or '1' in an array of size n x n.
- For the construction of an undirected graph with 'v' vertices & 'e' edges using adjacency list is O **(v + e)**, since for every vertex v in G we need to store all adjacent edges to vertex v.
- For DFS and BFS traversals of an undirected graph with 'n' vertices using adjacency matrixis **O (n^2),** since we visit to every value in an array of size n x n.
- For DFS and BFS traversals of an undirected graph with 'v' vertices using adjacency list **O (e),** since we visit to each node in adjacency list exactly ones which is equal to number of edges in a graph.

2) Space Complexity :

Additional space required for DFS and BFS to store the intermediate elements in STACK and QUEUE respectively while traversing the graph .

ALGORITHM :

This program represents a flight network with cities as nodes and flight paths as edges. It uses an adjacency list representation and checks for connectivity using a Depth-First Search (DFS) algorithm.

- **Justification for Adjacency List:**

Since the number of flight paths might be significantly smaller than the total number of cities, the space-efficient nature of the adjacency list is better suited than the denser matrix representation.

Input Format:

The program expects flight data through two vectors:

cities: A vector of strings containing the names of all cities.

flights: A vector of tuples (source city index, destination city index, cost) where indices correspond to the positions in the cities vector.

Algorithm: Check Flight Network Connectivity

1**. Define Edge Structure**

Create a structure named Edge with members destination (representing the destination city index) and cost (representing the cost of the flight).

2. **Define Graph Class**

1. Create a class named Graph with the following members:
2. cities vector to store city names.
3. adjList unordered map to represent the adjacency list of the graph.
4. addFlight method to add a flight edge to the graph.
5. isConnected method to check if the flight network is connected.
6. dfs method (private) to perform depth-first search for connectivity checking.

3**. Add Flight to Graph**

1. In the main function:
2. Initialize a Graph object.
3. Replace the cities and flights vectors with actual flight data.
4. Iterate through the flights, extracting source, destination, and cost, and add each flight to the graph using addFlight method.

4. Check Connectivity

1. Call the isConnected method on the graph object to check if the flight network is connected.
2. Print the result based on the connectivity status.

Detailed Steps:

1. Start the main function.
2. Initialize a Graph object named graph.
3. Replace the cities vector with the actual list of cities in the flight network.
4. Replace the flights vector with the actual flight data represented as tuples of (source, destination, cost).

5. Iterate through each flight in the flights vector:
    - Extract source, destination, and cost from the tuple.
    - Call the addFlight method on the graph object with source, destination, and cost as arguments.
    - Call the isConnected method on the graph object.
6. Inside the isConnected method:
    - Create a vector visited to track visited cities, initialized to false.
    - Call the dfs method starting from any city (assume city at index 0).
    - Check if all cities are visited.
    - If true, print "The flight network is connected!"
    - If false, print "The flight network is not connected."
7. End of the algorithm.

- **Justification for Adjacency Matrix:**

  While less space-efficient than the adjacency list, the matrix simplifies checking for direct connections between any two cities with a single lookup. This becomes important if your analysis prioritizes fast access to flight connections over memory usage.

  ALGORITHM:

  Algorithm: Check Flight Network Connectivity Using Adjacency Matrix

  1. Function to Add a Flight Path Between Two Cities

  a. Start the addFlight function with parameters source, destination, cost, cities, and adjMatrix.

  b. Map the source and destination city names to matrix indices using the mapCityToIndex function.

  c. Set the cost of the flight in the adjacency matrix at the corresponding indices.

  2. Function to Map City Names to Matrix Indices

  a. Start the mapCityToIndex function with parameters city and cities.

  b. Iterate through the cities vector to find the index of the given city.

c. If found, return the index.

d. If not found, print an error message and exit with an appropriate code.

3. Function to Check if the Graph is Connected Using DFS

a. Start the isConnected function with parameters adjMatrix and numCities.

b. Initialize a vector visited to track visited cities, initialized to false.

c. Call the dfs function starting from any city (assume city at index 0).

d. Check if all cities are visited.

e. If true, return true (the graph is connected).

f. If false, return false (the graph is not connected).

4. Helper Function for DFS Traversal on the Matrix

a. Start the dfs function with parameters city, visited, and adjMatrix.

b. Mark the current city as visited.

c. Iterate through neighbors of the current city in the adjacency matrix.

d. If there is a flight connection (cost > 0) to an unvisited neighbor, recursively call dfs on that neighbor.

5. Main Function

a. Start the main function.

b. Replace the cities vector with the actual list of cities in the flight network.

c. Initialize an adjacency matrix adjMatrix with all values initialized to zero.

d. Add flights to the network using the addFlight function.

e. Check if the flight network is connected using the isConnected function.

f. Print the result based on the connectivity status.

Detailed Steps:

a. Start the main function.

b. Initialize a vector of cities and an adjacency matrix with all values initialized to zero.

c. Replace the vector of cities and flights with actual flight data.

d. Iterate through each flight in the flight data, calling addFlight for each flight.

e. Check if the flight network is connected using the isConnected function.

f. Inside the isConnected function:

g. Create a vector visited to track visited cities, initialized to false.

h. Call the dfs function starting from any city (assume city at index 0).

i. Check if all cities are visited.

j. If true, print "The flight network is connected!"

k. If false, print "The flight network is not connected."

l. End of the algorithm.

CONCLUSION:

This program gives us the knowledge of adjacency matrix graph.

ASSIGNMENT QUESTIONS:

1. Define Graph?
2. Explain Breadth First Search traversal of Graph using an example.
3. Explain Depth First Search traversal of Graph using an example.
4. Discuss following with reference to graphs. (i) Directed graph (ii) Undirected graph (iii) Degree of vertex (iv)Null graph (v) Acyclic Graph

5.

ASSIGNMENT NO: 7

TITLE: MINIMUM SPANNING TREE (MST)

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

OBJECTIVE:

- To understand the concept and basic of spanning.
- To understand Graph in Data structure.

PREREQUISITE:

a. Basic of C++ Programming

b. Concept of Graph, MST.

THEORY:

Properties of a Greedy Algorithm:

1. At each step, the best possible choice is taken and after that only the sub-problem issolved.
2. Greedy algorithm might be depending on many choices. But it cannot ever be depending upon any choices of future and neither on sub-problems solutions.
3. The method of greedy algorithm starts with a top and goes down, creating greedy choices in aseries and then reduces each of the given problems to even smaller ones.

Minimum Spanning Tree:

A Minimum Spanning Tree (MST) is a kind of a sub graph of an undirected graph in which, the subgraph spans or includes all the nodes has a minimum total edge weight.

To solve the problem by a prim's algorithm, all we need is to find a spanning tree of minimum length,where a spanning tree is a tree that connects all the vertices together and a minimum spanning tree isa spanning tree of minimum length.

Spanning Tree:

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

Minimum Spanning Tree:

The minimum cost or minimum weightage spanning tree is called as Minimum Spanning Tree. To obtain a minimum spanning tree for a given connected graph, we can use Prim,s algorithm (vertex by vertex) or Kruskal‟ s algorithm (edge by edge).

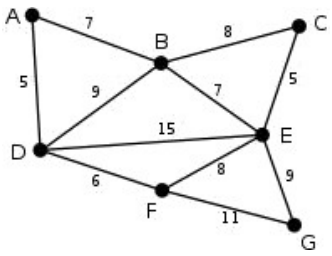
Prim's Algorithm:

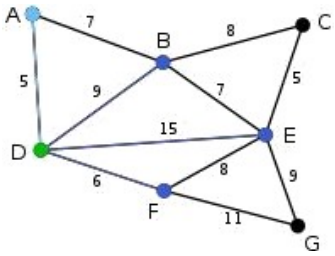
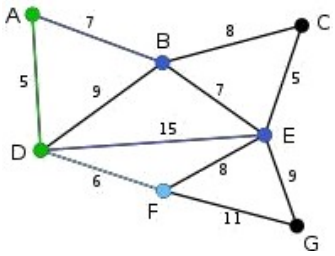
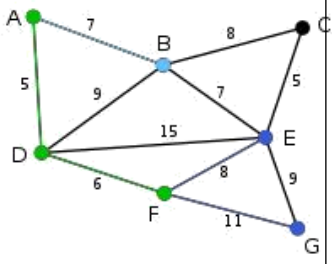
In computer science, Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

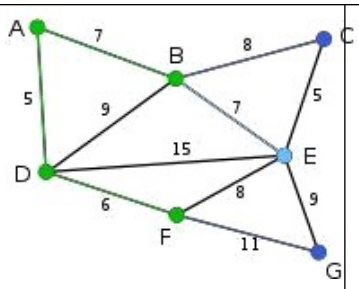
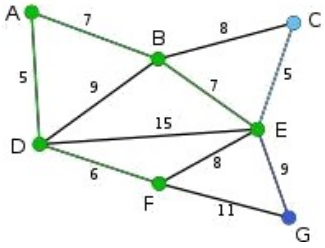
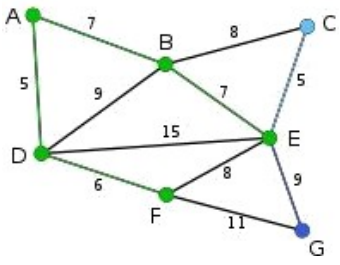
All vertices of a connected graph are included in minimum cost spanning tree. Prims algorithm starts from one vertex and grows the rest of tree by adding one vertex at a time by addingassociated edge in set T. This algorithm builds a tree by iteratively adding edges until all vertices are visited. The resultant tree is a minimum spanning tree. At each iteration it **selects the vertex** and associated edge having minimum cost or weight age that does not create a cycle. The algorithm is:

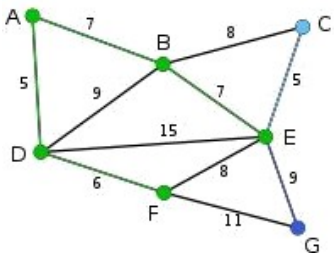
Applications of spanning trees:

- To find independent set of circuit equations for an electrical network. By adding an edge from set B to spanning tree we get a cycle and then Kirchhoff‟ s second law is used on the resulting cycle to obtain a circuit equation. Thus the total number of independent circuit equation we get is equal to the number of edges in set B.
- Using the property of spanning trees we can select the spanning tree with (n-1) edges such that total cost is minimum if each edge in a graph represents cost (weightage). For example, a communication network between number of cities.

Image	U	Edge(u,v)	V \ U	Description
	{}		{A,B,C,D, E,F,G}	This is our original weighted graph. The numbers near the edges indicate their weight.
		(D,A) = 5 **V**	{A,B,C,E, F,G}	Vertex **D** has been arbitrarily chosen as a starting point.

	{D}	(D,B) = 9 (D,E) = 15 (D,F) = 6		Vertices **A**, **B**, **E** and **F** are connected to **D** through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the
	{A,D}	(D,B) = 9 (D,E) = 15 (D,F) = 6 **V** (A,B) = 7	{B,C,E,F,G}	The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.
	{A,D,F}	(D,B) = 9 (D,E) = 15 (A,B) = 7 **V** (F,E) = 8 (F,G) = 11	{B,C,E,G}	The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.
		(B,C) = 8 (B,E) = 7		In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and**G** is 11 away from

	{A,B,D,F }	**V** (D,B) = 9 cycle (D,E) = 15 (F,E) = 8 (F,G) = 11	{C,E,G}	**F**. **E** is nearest, so we highlight the vertex **E** and the
	{A,B,D, E,F}	(B,C) = 8 (D,B) = 9 cycle (D,E) = 15 cycle (E,C) = 5 **V** (E,G) = 9 (F,E) = 8 cycle (F,G) = 11	{C,G}	Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the
	{A,B,C, D,E,F }	(B,C) = 8 cycle (D,B) = 9 cycle (D,E) = 15 cycle (E,G) = 9 **V** (F,E) = 8	{G}	Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight **G** and the arc **EG**.

		cycle (F,G) = 11		
	{A,B,C,D,E,F,G}	(B,C) = 8 cycle (D,B) = 9 cycle (D,E) = 15 cycle (F,E) = 8 cycle (F,G) = 11 cycle	{}	Now all the vertices have been selected and the minimum spanning tree is shown in green. Inthis case, it has weight 39.

DATA STRUCTURES:

a. **Cities:** Represent each office by its city name or a unique identifier. You can use a vector<string> or an array of custom City struct for storing city information.

b. **Connections:** Store connection options between cities with their associated costs. A two-dimensional array (matrix) or an adjacency list are suitable options.

c. **Matrix:** int costMatrix[numCities][numCities]; This holds the cost of connecting each pair of cities (i, j) at costMatrix[i][j].

d. **Adjacency List:** unordered_map<City, vector<pair<City, int>>> connectionMap; This maps each city to a list of connected cities and their respective costs.

ALGORITHM:

1. **Minimum Spanning Tree (MST):** We want to find a network connecting all offices such that the total cost of lines is minimized. Prim's algorithm is a popular choice for finding MSTs on weighted graphs.

Algorithm: Prim's Minimum Spanning Tree

1. Define Edge Structure
   - Create a structure named **Edge** with members **to** (representing the destination vertex) and **cost** (representing the edge cost).
2. Function to Find Minimum Spanning Tree using Prim's Algorithm
   1. Start the **primMST** function with the parameter **graph** (an adjacency list representation).
   2. Initialize necessary data structures:
      - **numVertices** to store the total number of vertices.
      - **visited** vector to track visited vertices.
      - **parent** vector to store the parent of each vertex in the MST.
      - **key** vector to store the minimum cost to connect each vertex to the MST (initialized to infinity).
   3. Set the key value of the starting vertex (vertex 0) to 0 and push it into a priority queue.
   4. Initialize **totalCost** to 0.
   5. While the priority queue is not empty:
      - Pop the vertex with the minimum key from the priority queue.
      - If the vertex is already visited, continue to the next iteration.
      - Mark the vertex as visited.
      - Update the key values of adjacent vertices if they are not visited and have a smaller weight.
      - Add the vertex and its key to the priority queue.
   6. Calculate the total cost of the MST by summing up the key values.
   7. Return the total cost.
3. Main Function
   1. Start the **main** function.
   2. Create an example graph represented as an adjacency list (replace with your actual data).
   3. Call the **primMST** function with the graph.
   4. Print the minimum total cost for connecting all offices.
   5.

Detailed Steps:

1. Start the **main** function.
2. Create an example graph (replace with actual data) using an adjacency list representation.
3. Call the **primMST** function with the example graph.

- Inside the **primMST** function:
- Initialize necessary data structures.
- Set the key value of the starting vertex to 0 and push it into a priority queue.
- Iterate through the priority queue until it's empty.
- Update key values and push vertices into the priority queue.
- Calculate and return the total cost of the MST.

4. Print the minimum total cost for connecting all offices.
5. End of the algorithm.

CONCLUSION:

Understood the concept and basic of spanning and to find the minimum distance between the vertices of Graph in Data structure.

ASSIGNMENT NO: 8

TITLE : BUILD OPTIMAL BINARY SEARCH TREE

PROBLEM STATEMENT:

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

OBJECTIVE:

- To learn implementation of Optimal Binary Search Tree.

PREREQUISITE:

1. Basic of C++ Programming
2. Basic of Optimal Binary Search Tree.

THEORY:

Optimal Binary Search Tree:

An optimal binary search tree (Optimal BST), sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities).

Given a sorted array keys [0.. n-1] of search keys and an array freq[0.. n-1] of frequency counts, where freq[i] is the number of searches to keys[i]. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible

- **Examples:**

  1. Input: keys[] = {10, 12}, freq[] = {34, 50}

  There can be following two possible BSTs 10 12

  \ /

  12 10

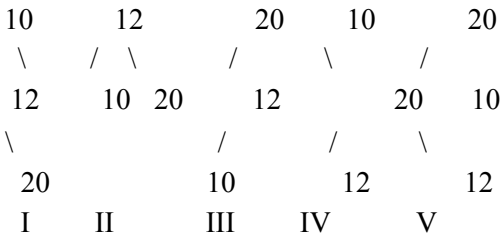
  I II

  Frequency of searches of 10 and 12 are 34 and 50 respectively. The cost of tree I is 34*1 + 50*2 = 134

  The cost of tree II is 50*1 + 34*2 = 118

2. Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs



Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142

Since there are "n" possible keys as candidates for the root of the optimal tree, the recursive solution must try them all. For each candidate key as root, all keys less than that key must appear in its left sub tree while all keys greater than it must appear in its right sub tree.

Stating the recursive algorithm based on these observations requires some notations:

- OBST(i, j) denotes the optimal binary search tree containing the keys ki, ki+1, …, kj;
- Wi, j denotes the weight matrix for OBST(i, j)
- Wi, j can be defined using the following formula: W[i,j]=W[i,j-1]+ pi+ qi
- Ci, j, 0<= i<= j <= n denotes the cost matrix for OBST(i, j)
- Ci, j can be defined recursively, in the following manner: Ci, i = Wi, j
  Ci, j = Wi, j + min i <=k<=j (Ci, k - 1 + Ck, j)
- Ri, j, 0<= i<= j <= n denotes the root matrix for OBST(i, j)

Assigning the notation Ri, j to the value of k for which we obtain a minimum in the above relations, the optimal binary search tree is OBST(0, n) and each subtree OBST(i, j) has the root k Rij and as subtrees the trees denoted by OBST(i, k-1) and OBST(k, j).

OBST(i, j) will involve the weights qi-1, pi, qi, …, pj, qj.

All possible optimal subtrees are not required. Those that are consist of sequences of keys that are immediate successors of the smallest key in the subtree, successors in the sorted order for the keys.

The bottom-up approach generates all the smallest required optimal subtrees first, then all next smallest, and so on until the final solution involving all the weights is found. Since the algorithm requires access to each subtree's weighted path length, these weighted path lengths must also be retained to avoid their recalculation. They will be stored in the weight matrix 'W'. Finally, the root of each sub tree must also be

stored for reference in the root matrix 'R'.

Detailed approach to building an optimal BST with the least search cost, given search probabilities:

1. Dynamic Programming:

Define a table e[i, j] to store the minimum expected cost of a BST for keys ki to kj.

Base cases:

e[i, i] = pi (cost of a single-node tree)

e[i, i + 1] = pi + pi+1 (cost of a two-node tree)

Recursive relation:

e[i, j] = min(e[i, r - 1] + e[r + 1, j] + sum(pi from i to j)) for i <= r <= j

Minimize over all possible roots r within the subtree.

2. Root Selection:

Find the root that minimizes e[i, j] using the dynamic programming table.

This root splits the keys into optimal left and right subtrees.

3. Recursive Construction:

Recursively construct the left and right subtrees using the same process.

ALGORITHM:

Algorithm: Construct Optimal Binary Search Tree (OBST)

1. Define Node Structure
   - Create a structure named **Node** with members **key** (representing the value of the key) and **probability** (representing the probability of the key).
2. Function to Calculate Optimal Cost
   1. Start the **optimalCost** function with parameters **keys**, **i**, and **j** representing a range of keys from index **i** to **j**.
   2. If **i** is greater than **j**, return 0 (base case).
   3. Initialize **minCost** to positive infinity.
   4. Iterate from **i** to **j**:
      a. Calculate the cost of the subtree rooted at key **k** by recursively calling **optimalCost** for the left and right subtrees.
      b. Update **minCost** with the minimum cost among all possible roots.
   5. Return the minimum cost.
3. Function to Construct Optimal Binary Search Tree
   1. Start the **constructOBST** function with parameters **keys**, **i**, and **j**.

2. If **i** is greater than **j**, return nullptr (base case).
3. Initialize **minCostIndex** to **i** and **minCost** to positive infinity.
4. Iterate from **i** to **j**:
   - Calculate the cost of the subtree rooted at key **k** by recursively calling **optimalCost** for the left and right subtrees.
   - Update **minCost** and **minCostIndex** with the minimum cost among all possible roots.
5. Create a new **Node** with the key and probability of the minimum cost index.
6. Recursively call **constructOBST** for the left and right subtrees using the minimum cost index as the root.
7. Return the constructed root.

4. Main Function

1. Start the **main** function.
2. Create a vector of **Node** named **keys** representing keys and their probabilities.
3. Call **constructOBST** with **keys**, **0** as the starting index, and **keys.size() - 1** as the ending index.
4. The resulting root of the constructed OBST is stored in the variable **root**.
5. Perform any necessary operations on the constructed OBST.

Detailed Steps:

1. Start the **main** function.
2. Create a vector of **Node** named **keys** representing keys and their probabilities.
3. Call **constructOBST** with **keys**, **0** as the starting index, and **keys.size() - 1** as the ending index.
   - Inside the **constructOBST** function:
   - Recursively find the optimal root and construct the left and right subtrees.
4. The resulting root of the constructed OBST is stored in the variable **root**.
5. Perform any necessary operations on the constructed OBST.
6. End of the algorithm.

CONCLUSION:

In this way we have done text data analysis using TF IDF algorithm.

ASSIGNMENT QUESTION:

ASSIGNMENT NO: 9

TITLE: IMPLEMENTATION OF AVL TREE

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. (AVL Tree)

OBJECTIVE:

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

PREREQUISITE:

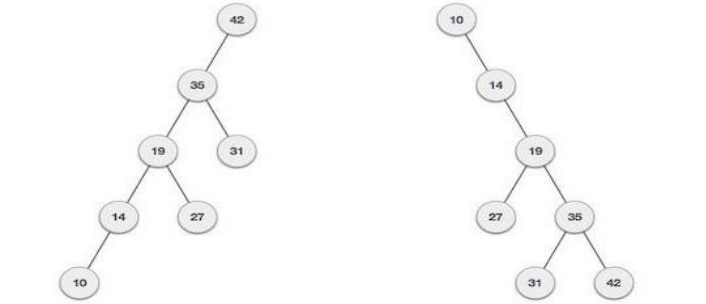
- Basic of C++ Programming.
- Concept of Height balance tree, AVL Tree.

THEORY:

An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR asits 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 theleft and right sub-trees of a node are either equal or if they differ only by 1.

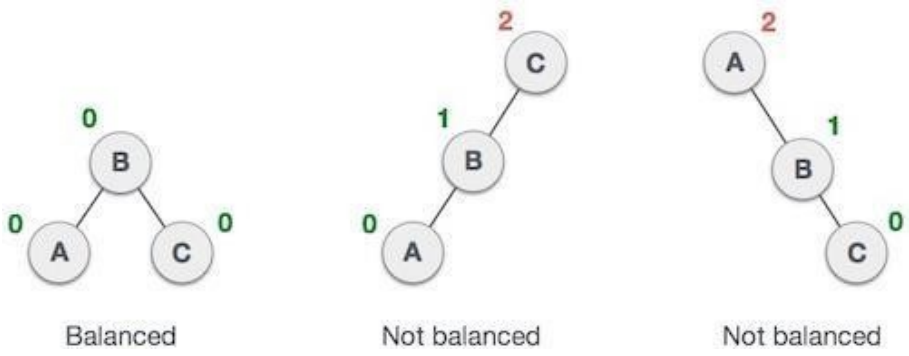
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this



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.

BalanceFactor = height(left-sutree) − height(right-sutree)

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

AVL Rotations

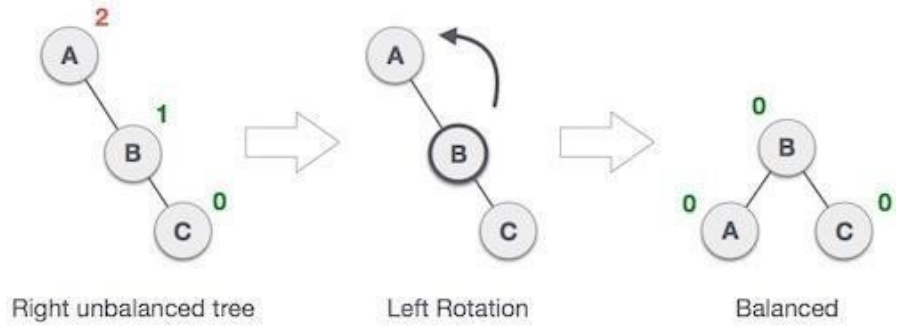
To balance itself, an AVL tree may perform the following four kinds of rotations −

- 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

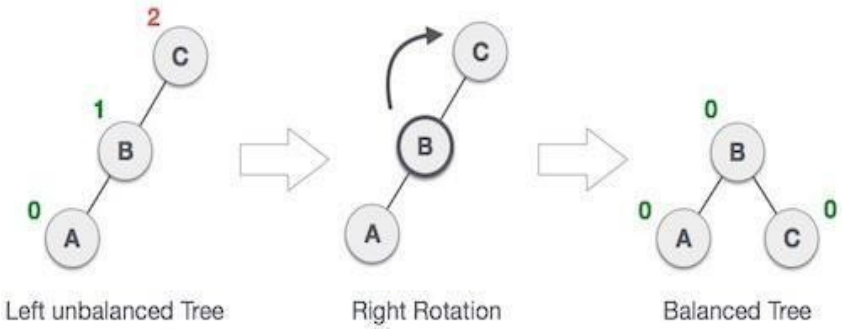
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



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

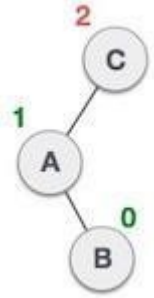
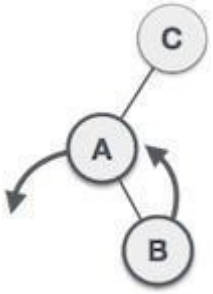
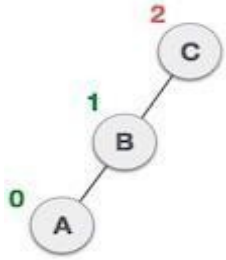
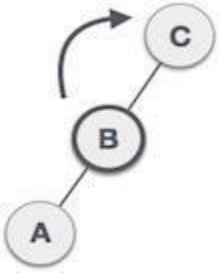
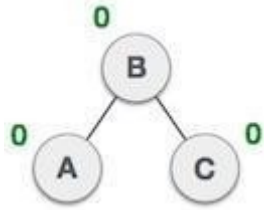
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



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

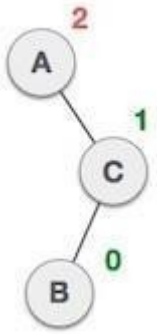
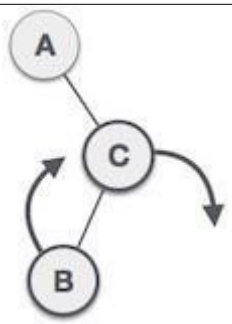
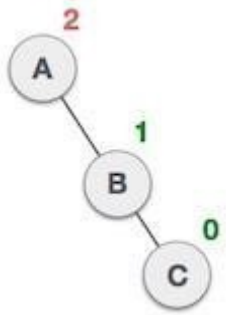
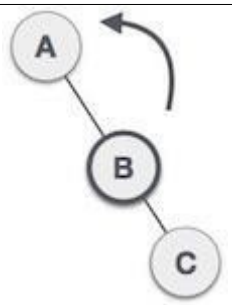
Left-Right Rotation

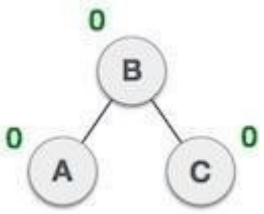
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right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

State	Action
	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.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node C is still unbalanced, however now, it is because of the left- subtree of the left-subtree.
	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.
	The tree is now balanced.

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
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.
	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**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	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**.

	The tree is now balanced. 42

- **Data Structure:** Utilize a balanced binary search tree (e.g., AVL tree, red-black tree) to maintain efficient operations.
- **Node Structure:** Each node in the tree stores a keyword-meaning pair.
- **Operations:**
    a. **Adding a Keyword:** Insert a new node with the keyword and its meaning into the tree while maintaining balance.
    b. **Deleting a Keyword:** Remove the node containing the keyword, adjusting the tree structure to preserve balance.
    c. **Updating a Meaning:** Search for the keyword's node, modify its meaning, and potentially rebalance if necessary.
    d. **Displaying Data (Sorted):** Perform an in-order traversal to display keywords in ascending or descending order.
    e. **Finding a Keyword:** Traverse the tree, comparing the keyword with nodes' keys until a match is found or the search space is exhausted.
- **Maximum Comparisons for Finding a Keyword:**
- **Worst-Case:** The maximum comparisons needed is equal to the height of the tree, which is logarithmic in the number of nodes.
- **Height-Balanced Trees:** Ensure the height remains logarithmic, leading to efficient search operations.
- **Complexity for Finding a Keyword:**
- Average and Worst-Case: O(log n), where n is the number of keywords.

ALGORITHM:

Algorithm: AVL Tree Insertion and Deletion

1. Define Node Structure

- Create a structure named **Node** with members **key**, **left**, **right**, and **height**.

2. Function to Get Height of a Node

- Start the **height** function with parameter **N** (Node).
- If **N** is **NULL**, return 0.
- Return the **height** of node **N**.

3. Function to Get Balance Factor

- Start the **Balance** function with parameter **N** (Node).
- If **N** is **NULL**, return 0.
- Return the balance factor by subtracting the height of the right subtree from the height of the left subtree.

4. Function to Get Maximum of Two Numbers

- Start the **max** function with parameters **x** and **y**.
- Return the maximum of **x** and **y**.

5. Right Rotation Function

- Start the **RightRotation** function with parameter **y** (Node).
- Perform a right rotation on the nodes and update heights.
- Return the new root after rotation.

6. Left Rotation Function

- Start the **LeftRotation** function with parameter **x** (Node).

- Perform a left rotation on the nodes and update heights.
- Return the new root after rotation.

7. Insertion Function

- Start the **insert** function with parameters **node** (current root) and **key** (value to insert).
- Perform normal Binary Search Tree (BST) insertion.
- Update the height of the current node.
- Get the balance factor (BF) to check if the tree is unbalanced.
- Perform rotations (right, left, LR, RL) if needed to maintain the balance.
- Return the new root.

8. Function to Find Minimum Node in a Subtree

- Start the **minNode** function with parameter **node** (root of the subtree).
- Find and return the node with the minimum key value in the subtree.

9. Deletion Function

- Start the **Delete** function with parameters **node** (current root) and **key** (value to delete).
- Perform normal BST deletion.
- Update the height of the current node.
- Get the balance factor (BF) to check if the tree is unbalanced.
- Perform rotations (right, left, LR, RL) if needed to maintain the balance.
- Return the new root.

10. Level-Wise Printing Functions

- Define functions to print nodes at a given level and to print the entire tree level-wise.

- Main Function
- Start the **main** function.
- Initialize AVL tree root as **NULL**.
- Loop until the user chooses to exit:
    a. Prompt user to choose an operation (insertion, deletion, exit).
    b. Call respective function based on user's choice.
    c. Print the AVL tree and its height after each operation.

Detailed Steps:

1. Start the **main** function.
2. Initialize AVL tree root as **NULL**.
3. Loop until the user chooses to exit:
    - Prompt user to choose an operation (insertion, deletion, exit).
    - If insertion:
        1. Prompt user for the number of values to insert.
        2. Insert each value into the AVL tree using the **insert** function.
    - Print the AVL tree and its height after insertion.
    - If deletion:
        - Prompt user for the value to delete.
        - Delete the value from the AVL tree using the **Delete** function.
        - Print the AVL tree and its height after deletion.
    - If exit, end the loop.
4. End of the algorithm.

CONCLUSION:

This program gives us the knowledge height balanced binary tree.

ASSIGNMENT QUESTIONS:

1. Construct an AVL tree having the following elements
2. **H, I, J, B, A, E, C, F, D, G, K, L**
3. What are the key properties of an AVL Tree that make it a good data structure for maintaining a self-balancing binary search tree?
4. What is the meaning of height balanced tree? How rebalancing is done in height balanced tree.

ASSIGNMENT NO: 10

TITLE : IMPLEMENT THE PRIORITY QUEUE USING ARRAY

PROBLEM STATEMENT:

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.

OBJECTIVE:

1. To learn the priority queue.
2. To understand the practical implementation of priority queue using array.

PREREQUISITE:

1. Basic of C++ Programming
2. Concept of priority queue.

THEORY:

Queue: A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer. Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists. Queue is a data structure that maintains "First In First Out" (FIFO) order. And can be viewed as people queueing up to buy a ticket. In programming, queue is usually used as a data structure for BFS (Breadth First Search).

Operations on queue:

1. enqueue - insert item at the back of queue Q
2. dequeue - return (and virtually remove) the front item from queue Q
3. displayfront - return (without deleting) the front item from queue Q

Priority Queue:

Priority queue is an abstract data type in computer programming that supports the following three operations:

1. insertWithPriority: add an element to the queue with an associated priority
2. getNext: remove the element from the queue that has the highest priority, and return it (also known as "PopElement(Off)", or "GetMinimum")
3. peekAtNext (optional): look at the element with highest priority without removing it.

The rule that determines who goes next is called a queueing discipline. The simplest queueing discipline is called FIFO, for "first-in-first-out." The most general queueing discipline is priority queueing, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are "fair," but fairness is in the eye of the beholder.

Priority Queue (as the name suggests) uses the priority queueing policy. As with most ADTs, there are a number of ways to implement queues. Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections: arrays, lists, or vectors. Our choice among them will be based in part on their performance--- how long it takes to perform the operations we want to perform--- and partly on ease of implementation.

Implementation

A priority queue can be implemented using data structures like arrays, linked lists, or heaps. The array can be ordered or unordered.

Using an ordered array

The item is inserted in such a way that the array remains ordered i.e. the largest item is always in the end. The insertion operation is illustrated in figure 1.

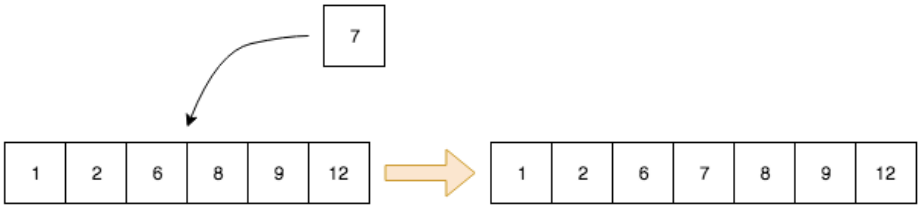


Fig 1: Insertion operation in an ordered array

The item with priority 7 is inserted between the items with priorities 6 and 8. We can insert it at the end of the queue. If we do so the array becomes unordered. Since we must scan through the queue in order to find the appropriate position to insert the new item, the worst-case complexity of this operation is O(n). Since the item with the highest priority is always in the last position, the dequeue and peek operation takes a constant time.

Using an unordered array

We insert the item at the end of the queue. While inserting, we do not maintain the order. The complexity of this operation is O(1). Since the queue is not ordered, we need to search through the queue for the item with maximum priority. Once we remove this item, we need to move all the items after it one step to the left. The dequeue operation is illustrated in figure 2.

It is obvious that the complexity of dequeue and peek operation is O(n).

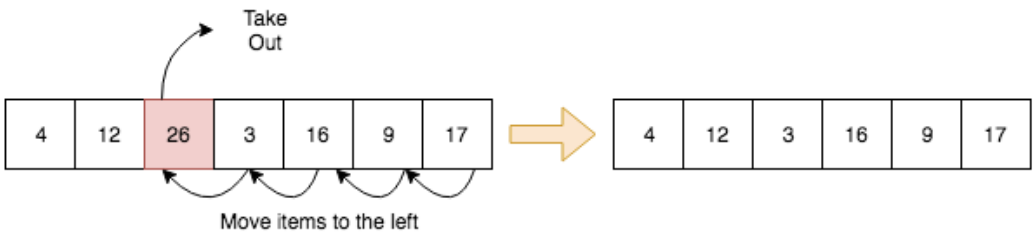


Fig 2: Dequeue operation in an unordered array

Using a linked list

The linked can be ordered or unordered just like the array. In the ordered linked list, we can insert item so that the items are sorted and the maximum item is always at the end (pointed by head or tail). The complexity of enqueue operation is O(n) and dequeue and peek operation is O(1).

In an unordered linked list, we can insert the item at the end of the queue in constant time. The dequeue operation takes linear time (O(n)) because we need to search through the queue in order to find the item with maximum priority.

Using a binary heap

In above implementations using arrays and linked lists, one operation always takes linear time i.e. O(n). Using binary heaps, we can make all the operations faster (in logarithmic time). Please read about the binary heaps before using them in a priority queue.

Using a binary heap, the enqueue operation is insert operation in the heap. It takes O(log n) time in the worst case. Similarly, the dequeue operation is the extract-max or remove-max operation which also takes O(log n) time. The peek operation is a constant time operation. In this way, the binary heap makes the priority queue operations a way faster.

ALGORITHM:

Explanation:

1. Data Structures:

1. Priority enum for representing patient priorities.
2. Patient class to store patient information.
3. priority_queue with custom comparison for priority-based ordering.

2. Comparison Function:

1. comparePatients prioritizes patients based on priority.
2. You can add logic for breaking ties based on arrival time or other factors.

3. Adding Patients:

1. Use push to add patients with their priority and arrival time.

4. Serving Patients:

1. Use top to access the highest priority patient.
2. Use pop to remove the served patient from the queue.

5. Output:

1. The program prints the served patients in priority order.

CONCLUSION:

After successful implementation of this assignment, we understood the priority queue as ADT using array.

ASSIGNMENT QUESTIONS

1. Define priority queue with diagram and give the operations.
2. Give the applications of priority queues.
3. Explained the concept of Priority Queue and Priority Queue ADT.
4.

ASSIGNMENT NO : 11

TITLE: FILE ORGANIZATION USING C++

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.

OBJECTIVE:

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

PREREQUISITE:

1. Basic of C++ Programming
2. File organization using c++.

Theory:

File organization refers to the relationship of the key of the record to the physical locationof 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 handis a complete set of records for a specific application or purpose. A logical file may occupy a partof physical file or may extend over more than one physical file.

There are various methods of file organizations. These methods may be efficient for certain 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.

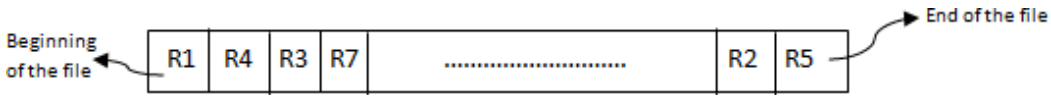
Some of the file organizations are

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

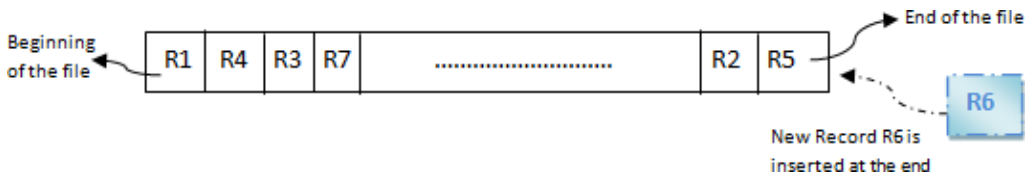
Sequential File Organization:

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

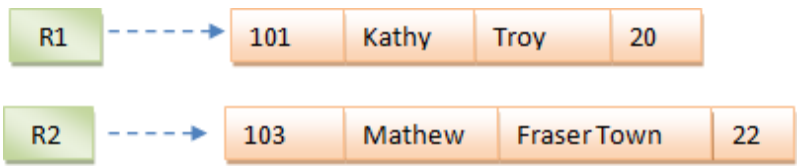
- Records are stored one after the other as they are inserted into the tables. This method is called any modification or deletion of record, the record will be searched in the memory blocks. Once it is found, it will be marked for deleting and new block of recordis entered.



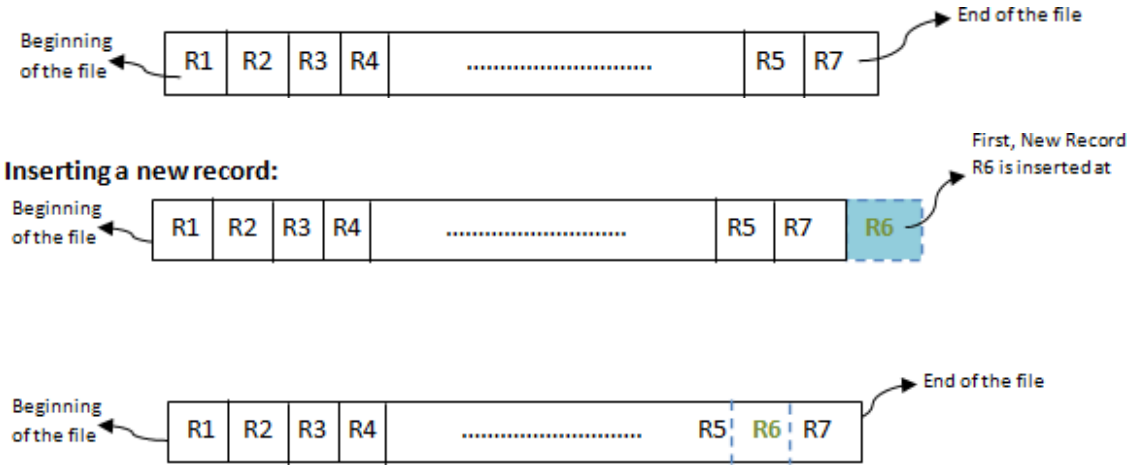
Inserting a new record:



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:

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

Disadvantages:

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

ALGORITHM:

Algorithm: Student Information System

1. Define Student Structure
   - Create a structure named **Student** with members **rollNo**, **name**, **division**, and **address**.
2. Function to Add a Student
   - Open the file "students.dat" in append mode.
   - Read student details (roll number, name, division, address) from the user.
   - Write the student structure to the file.
   - Close the file.
   - Print a success message.
3. Function to Delete a Student
   - Open the file "students.dat" for reading and create a temporary file "temp.dat" for writing.
   - Read the roll number to delete from the user.
   - Traverse through the file and copy records to "temp.dat" excluding the one to be deleted.
   - Close both files.
   - Delete the original file and rename "temp.dat" to "students.dat".
   - Print a success message if the student is found, else print a message indicating the student was not found.
4. Function to Display Student Information
   - Open the file "students.dat" for reading.
   - Read the roll number to display from the user.
   - Traverse through the file and print details if the roll number matches.
   - Close the file.
   - Print a message indicating whether the student was found or not.
5. Main Function
   - Initialize a variable choice to 0.
   - Start a do-while loop until the user chooses to exit.
   - Display a menu with options to add, delete, display, or exit.
   - Read the user's choice.
   - Execute the corresponding function based on the choice.
   - Repeat until the user chooses to exit.
6. Detailed Steps:
   1. Start the program by initializing the choice variable to 0.
   2. Enter a do-while loop to display the menu until the user chooses to exit.
   3. Display the menu options:
      1. Add Student
      2. Delete Student
      3. Display Student Information
      4. Exit
   4. Read the user's choice.
   5. Execute the corresponding function based on the choice:
      a. Case 1: Add a student using the **addStudent** function.
         - Case 2: Delete a student using the **deleteStudent** function.

- Case 3: Display student information using the **displayStudent** function.
- Case 4: Print an exit message and end the loop.
- Default: Print an invalid choice message.

6. Repeat the loop until the user chooses to exit.
7. End of the algorithm.

CONCLUSION: This program gives us the knowledge sequential file organization.

ASSIGNMENT QUESTIONS:

1. Compare sequential file organization with direct access file organization.
2. Write short notes on: (a) Factors affecting the file organization (b) Indexed sequential files (c) Indexing techniques
3. Describe the basic types of file organization each with one example.
4. State the advantages, disadvantages, and primitive operation of sequential files.

ASSIGNMENT NO : 12

TITLE: IMPLEMENTATION OF INDEX SEQUENTIAL FILE

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.

OBJECTIVE:

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

PREREQUISITE:

1. Basic of C++ Programming
2. File organization using c++.

Theory:

File organization refers to the relationship of the key of the record to the physical locationof 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 handis a complete set of records for a specific application or purpose. A logical file may occupy a partof physical file or may extend over more than one physical file.

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

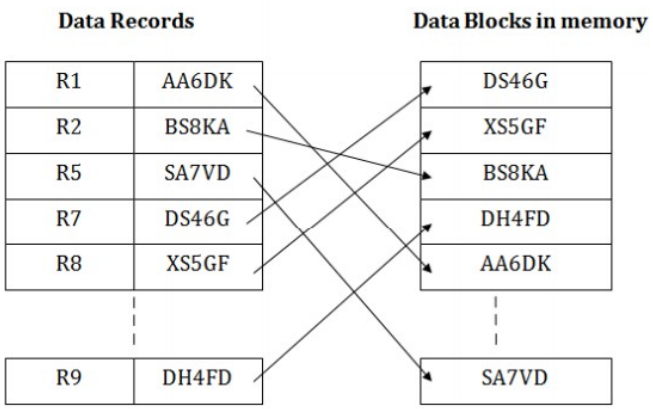
Some of the file organizations are

1. Sequential File Organization Heap File Organization

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

1. Indexed sequential access method (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and therecord is retrieved from the memory.

Pros of ISAM:

- In this method, each record has the address of its data block, searching a record in a hugedatabase is quick and easy.
- This method supports range retrieval and partial retrieval of records. Since the index is basedon the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Cons of ISAM

- This method requires extra space in the disk to store the index value. When the new recordsare inserted, then these files have to be reconstructed to maintain the sequence.

- When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

Key Points:

- Use fstream for binary file operations.
- Create separate files for data and index.
- Implement functions for adding, deleting, and displaying employees, handling index maintenance.
- Consider error handling and data validation.

ALGORITHM :

1. Include necessary header files:
   - iostream: for input/output operations
   - fstream: for file operations
   - cstring: for string-related operations
2. Define Employee structure:
   - Define a structure named Employee with fields such as employeeID, name, designation, and salary.
3. Function Prototypes:
   - Declare function prototypes for addEmployee, deleteEmployee, and displayEmployee.
4. Main Function:
   - Open a file stream (fstream) with the name "employee_data.dat" for input, output, binary, and append operations.
   - Display a menu with options to add, delete, display employees, and exit.
5. Add Employee Function (addEmployee):
   - Create an Employee structure variable (emp).
   - Prompt the user to enter employee details (ID, name, designation, salary).
   - Check if the employee with the entered ID already exists in the file.
   - If yes, display a message and return.
   - If no, write the employee details to the file.
   - Display a success message.
6. Delete Employee Function (deleteEmployee):

- Prompt the user to enter the employee ID to delete.
- Create an Employee structure variable (emp).
- Create a temporary file stream (fstream) named "temp.dat" for output and binary operations.
- Read each employee record from the main file.
- If the ID matches the one to be deleted, skip that record.
- Otherwise, write the record to the temporary file.
- Close both file streams.
- Remove the original file and rename the temporary file to the original file name.
- Display a success message.

7. Display Employee Function (displayEmployee):
   - Prompt the user to enter the employee ID to display.
   - Create an Employee structure variable (emp).
   - Read each employee record from the main file.
   - If the ID matches the one to be displayed, print the details and return.
   - If the ID is not found, display a message indicating that the employee was not found.
8. Repeat:
   - Continue displaying the menu and processing user input until the user chooses to exit.
9. Close File Stream:
   - Close the file stream before exiting the program.

CONCLUSION:

This program gives us the knowledge index sequential file organization

ASSIGNMENT QUESTIONS:

1. What are indexed files? Explain with a suitable example. Compare sequential and direct access files.
2. What is a multi-index file? Give suitable examples.