**Practical No:1**

**Title:** To create an ADT that implements the Set concept

**CODE:**

class MySet: # Define a custom class named MySet

def \_\_init\_\_(self, elements=None): # Constructor method with optional input list

if elements is None: # If no elements are passed

self.set = [] # Initialize an empty list

else:

self.set = [] # Initialize an empty list

for element in elements: # Loop through the input elements

if element not in self.set: # Add only unique elements

self.set.append(element) # Append the element to the list

def union(self, set2): # Method to perform union with another MySet

union\_set = self.set.copy() # Make a copy of the current set

for element in set2.set: # Loop through elements of set2

if element not in union\_set: # If element is not already in the union set

union\_set.append(element) # Add it to the union set

return union\_set # Return the union result

def intersection(self, set2): # Method to find intersection with set2

intersection\_set = [] # Initialize an empty list

for element in self.set: # Loop through current set elements

if element in set2.set: # If element is also in set2

intersection\_set.append(element) # Add it to intersection

return intersection\_set # Return intersection result

def is\_subset(self, set2): # Method to check if current set is a subset of set2

for element in self.set: # Loop through current set elements

if element not in set2.set: # If any element is not in set2

return False # Then it’s not a subset

return True # All elements matched, it's a subset

def difference(self, set2): # Method to find difference from set2

difference\_set = [] # Initialize an empty list

for element in self.set: # Loop through current set

if element not in set2.set: # If element is not in set2

difference\_set.append(element) # Add it to the result

return difference\_set # Return difference result

def display(self): # Method to display the set

print("Set:", self.set) # Print the list representing the set

def input\_set(prompt): # Function to take user input for a set

user\_input = input(prompt) # Prompt the user and get input as a string

elements = user\_input.split(',') # Split the input string by commas

elements = [element.strip() for element in elements] # Remove whitespace from each element

return MySet([str(element) for element in elements]) # Create and return a MySet object

def menu(): # Function to display the menu

print("\nMenu:") # Print a newline and "Menu" header

print("1. Union of sets") # Option 1

print("2. Intersection of sets") # Option 2

print("3. Check if set1 is a subset of set2") # Option 3

print("4. Difference of set1 and set2") # Option 4

print("5. Display sets") # Option 5

print("6. Exit") # Option 6

def main(): # Main function to control the flow

set1 = input\_set("Enter the elements of set1 (comma separated integers): ") # Input for set1

set2 = input\_set("Enter the elements of set2 (comma separated integers): ") # Input for set2

while True: # Start an infinite loop for the menu

menu() # Show menu options

choice = int(input("Enter your choice: ")) # Get user’s choice as an integer

if choice == 1: # If user chooses union

result = set1.union(set2) # Call union method

print("Union of set1 and set2:", result) # Print the result

elif choice == 2: # If user chooses intersection

result = set1.intersection(set2) # Call intersection method

print("Intersection of set1 and set2:", result) # Print result

elif choice == 3: # If user checks for subset

if set1.is\_subset(set2): # Call is\_subset method

print("Set1 is a subset of Set2.") # If True

else:

print("Set1 is not a subset of Set2.") # If False

elif choice == 4: # If user chooses difference

result = set1.difference(set2) # Call difference method

print("Difference of set1 and set2 (set1 - set2):", result) # Print result

elif choice == 5: # If user wants to display sets

print("Set1:", set1.set) # Print set1

print("Set2:", set2.set) # Print set2

elif choice == 6: # If user wants to exit

print("Exiting the program.") # Show exit message

break # Break out of the loop

else:

print("Invalid choice! Please try again.") # Handle invalid input

if \_\_name\_\_ == "\_\_main\_\_": # Check if this script is run directly

main() # Call the main function

**OUTPUT:**

Enter the elements of set1 (comma separated integers): 1,2,3,4

Enter the elements of set2 (comma separated integers): 3,4,5,6

Menu:

1. Union of sets

2. Intersection of sets

3. Check if set1 is a subset of set2

4. Difference of set1 and set2

5. Display sets

6. Exit

Enter your choice: 1

Union of set1 and set2: ['1', '2', '3', '4', '5', '6']

Menu:

1. Union of sets

2. Intersection of sets

3. Check if set1 is a subset of set2

4. Difference of set1 and set2

5. Display sets

6. Exit

Enter your choice: 2

Intersection of set1 and set2: ['3', '4']

Menu:

1. Union of sets

2. Intersection of sets

3. Check if set1 is a subset of set2

4. Difference of set1 and set2

5. Display sets

6. Exit

Enter your choice: 3

Set1 is not a subset of Set2.

Menu:

1. Union of sets

2. Intersection of sets

3. Check if set1 is a subset of set2

4. Difference of set1 and set2

5. Display sets

6. Exit

Enter your choice: 4

Difference of set1 and set2 (set1 - set2): ['1', '2']

Menu:

1. Union of sets

2. Intersection of sets

3. Check if set1 is a subset of set2

4. Difference of set1 and set2

5. Display sets

6. Exit

Enter your choice: 5

Set1: ['1', '2', '3', '4']

Set2: ['3', '4', '5', '6']

Menu:

1. Union of sets

2. Intersection of sets

3. Check if set1 is a subset of set2

4. Difference of set1 and set2

5. Display sets

6. Exit

Enter your choice: 6

Exiting the program.

=== Code Execution Successful ===

**Practical No:2**

**Title:** Hash Table implementation

**CODE:**

class Hashtable: # Define a class for the hashtable

def \_\_init\_\_(self): # Constructor to initialize hashtable properties

self.size = int(input("Enter the size of the hash table: ")) # Ask user for table size

self.count = 0 # Counter to track the number of inserted keys

self.list = [None] \* self.size # Create a list of given size filled with None

self.comparisons = 0 # Counter to track the number of comparisons during operations

def \_hashfunction\_(self, key): # Private method to compute hash index

return key % self.size # Return index using modulo hashing

def is\_full(self): # Method to check if hashtable is full

return self.count == self.size # Return True if all slots are filled

def insert(self, key): # Method to insert a key

if not isinstance(key, int): # Check if key is an integer

return "Key must be an integer"

if self.is\_full(): # If table is full, insertion is not possible

return "Hashtable is full"

index = self.\_hashfunction\_(key) # Calculate initial index

original\_index = index # Store original index to detect full cycle

while self.list[index] is not None: # If slot is occupied

self.comparisons += 1 # Increment comparison count

if self.list[index] == key: # If key already exists

return "Key already exists"

index = (index + 1) % self.size # Move to next index (linear probing)

if index == original\_index: # If we returned to start, table is full

return "Hashtable is full"

self.list[index] = key # Insert key at found index

self.count += 1 # Increment count of inserted elements

return "Key inserted successfully"

def search(self, key): # Method to search a key

if not isinstance(key, int): # Ensure key is an integer

return "Key must be an integer"

index = self.\_hashfunction\_(key) # Calculate initial index

original\_index = index # Store starting index

while self.list[index] is not None: # Loop until empty slot found

self.comparisons += 1 # Increment comparison counter

if self.list[index] == key: # Key found

return f"Key found at index {index}"

index = (index + 1) % self.size # Move to next index

if index == original\_index: # Full cycle completed

break

return "Key not found" # Key does not exist in table

def remove(self, key): # Method to remove a key

if not isinstance(key, int): # Validate key type

return "Key must be an integer"

index = self.\_hashfunction\_(key) # Calculate initial index

original\_index = index # Store original index

while self.list[index] is not None: # Loop until empty slot

self.comparisons += 1 # Track comparisons

if self.list[index] == key: # If key is found

self.list[index] = None # Set slot to None (remove key)

self.count -= 1 # Decrease count

return f"Key {key} removed successfully"

index = (index + 1) % self.size # Move to next index

if index == original\_index: # Completed one full cycle

break

return "Key not found" # Key is not in the table

def display(self): # Method to display table contents

print("Hashtable contents:") # Heading for table

for i in range(self.size): # Loop through table

if self.list[i] is not None: # If slot has a key

print(f"Index {i}: {self.list[i]}") # Print index and key

else:

print(f"Index {i}: Empty") # Print empty if no key

print(f"Total comparisons: {self.comparisons}") # Show number of comparisons

self.comparisons = 0 # Reset comparison count after displaying

def menu(): # Function to provide menu-driven interface

h1 = Hashtable() # Create an instance of the Hashtable class

while True: # Infinite loop until user exits

print("\n--- Hashtable Operations ---") # Menu header

print("1. Insert a key") # Option 1

print("2. Search for a key") # Option 2

print("3. Remove a key") # Option 3

print("4. Display the hashtable") # Option 4

print("5. Exit") # Option 5

choice = input("Enter your choice: ") # Ask user for choice

if choice == '1': # If choice is 1 (insert)

key = int(input("Enter the key to insert: ")) # Input key

print(h1.insert(key)) # Call insert and print result

elif choice == '2': # If choice is 2 (search)

key = int(input("Enter the key to search: ")) # Input key

print(h1.search(key)) # Call search and print result

elif choice == '3': # If choice is 3 (remove)

key = int(input("Enter the key to remove: ")) # Input key

print(h1.remove(key)) # Call remove and print result

elif choice == '4': # If choice is 4 (display)

h1.display() # Call display method

elif choice == '5': # If choice is 5 (exit)

print("Exiting the program...") # Exit message

break # Exit the loop

else:

print("Invalid choice. Please try again.") # Invalid input handling

menu() # Call the menu function to start the program

Output:  
Enter the size of the hash table: 5

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 1

Enter the key to insert: 34

Key inserted successfully

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 1

Enter the key to insert: 23

Key inserted successfully

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 1

Enter the key to insert: 67

Key inserted successfully

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 1

Enter the key to insert: 89

Key inserted successfully

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 1

Enter the key to insert: 90

Key inserted successfully

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 2

Enter the key to search: 89

Key found at index 0

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 2

Enter the key to search: 90

Key found at index 1

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 3

Enter the key to remove: 90

Key 90 removed successfully

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit
6. Enter your choice: 4

Hashtable contents:

Index 0: 89

Index 1: Empty

Index 2: 67

Index 3: 23

Index 4: 34

Total comparisons: 8

--- Hashtable Operations ---

1. Insert a key
2. Search for a key
3. Remove a key
4. Display the hashtable
5. Exit

Enter your choice: 5

Exiting the program...

=== Code Execution Successful

**Practical No: 3**

**Title: Implementation of General Tree**

**CODE:**

#include <iostream> // For input-output operations

#include <string.h> // For using string functions

using namespace std; // Use standard namespace to avoid prefixing std::

struct node // Node structure declaration

{

string label; // To store the label (book, chapter, or section name)

int ch\_count; // Number of children (chapters or sections)

struct node \*child[10]; // Array of pointers to child nodes (max 10 children)

} \* root; // Global pointer to the root node (book)

class GT // GT = General Tree, class to manage tree operations

{

public:

void create\_tree(); // Function to create tree structure

void display(node \*r1); // Function to display tree structure

GT() // Constructor

{

root = NULL; // Initialize root to NULL

}

};

void GT::create\_tree() // Function to create book hierarchy tree

{

int tchapters, i, j; // Variables for counting chapters and sections

root = new node; // Allocate memory for root node (book)

cout << "Enter name of book : "<<endl;

cin.ignore(); // Clear newline from previous input

getline(cin, root->label); // Get full line input for book title

cout << "Enter number of chapters in book : " << endl;

cin >> tchapters; // Input number of chapters

root->ch\_count = tchapters; // Set chapter count in root

cin.ignore(); // Ignore newline character left by cin

for (i = 0; i < tchapters; i++) // Loop for each chapter

{

root->child[i] = new node; // Allocate memory for each chapter

cout << "Enter the name of Chapter " << i + 1 << " : ";

getline(cin, root->child[i]->label); // Input chapter name

cout << "Enter number of sections in Chapter " << root->child[i]->label << " : ";

cin >> root->child[i]->ch\_count; // Input number of sections in this chapter

cin.ignore(); // Ignore newline

for (j = 0; j < root->child[i]->ch\_count; j++) // Loop for each section

{

root->child[i]->child[j] = new node; // Allocate memory for each section

cout << "Enter Name of Section " << j + 1 << " : ";

getline(cin, root->child[i]->child[j]->label); // Input section name

}

}

}

void GT::display(node \*r1) // Function to display the tree

{

int i, j;

if (r1 != NULL) // If root is not null

{

cout << "\n-----Book Hierarchy---";

cout << "\n Book title : " << r1->label; // Display book title

for (i = 0; i < r1->ch\_count; i++) // Loop through chapters

{

cout << "\nChapter " << i + 1 << " : " << r1->child[i]->label; // Display chapter

cout << "\nSections : ";

for (j = 0; j < r1->child[i]->ch\_count; j++) // Loop through sections of chapter

{

cout << "\n" << r1->child[i]->child[j]->label; // Display section name

}

}

}

cout << endl;

}

int main()

{

int choice; // Variable to store user choice

GT gt; // Create object of GT class

while (1) // Infinite loop for menu

{

// Display menu

cout << "-----------------" << endl;

cout << "Book Tree Creation" << endl;

cout << "-----------------" << endl;

cout << "1.Create" << endl;

cout << "2.Display" << endl;

cout << "3.Quit" << endl;

cout << "Enter your choice : ";

cin >> choice;

switch (choice) // Based on user choice, perform actions

{

case 1:

gt.create\_tree(); // Call tree creation function

break;

case 2:

gt.display(root); // Call display function

break;

case 3:

cout << "Thanks for using this program!!!"; // Exit message

exit(0); // Exit program

default:

cout << "Invalid choice!!!" << endl; // Error for invalid input

}

}

return 0; // End of program

}

Output:

-----------------

Book Tree Creation

-----------------

1.Create

2.Display

3.Quit

Enter your choice : 1

Enter name of book :

Data Structures and Algorithm

Enter number of chapters in book :

5

Enter the name of Chapter 1 : Hashing

Enter number of sections in Chapter Hashing : 2

Enter Name of Section 1 : Hash Table

Enter Name of Section 2 : Skip List

Enter the name of Chapter 2 : Trees

Enter number of sections in Chapter Trees : 3

Enter Name of Section 1 : General Tree

Enter Name of Section 2 : Binary Tree

Enter Name of Section 3 : Huffman Tree

Enter the name of Chapter 3 : Graphs

Enter number of sections in Chapter Graphs : 1

Enter Name of Section 1 : Traversals

Enter the name of Chapter 4 : Search Trees

Enter number of sections in Chapter Search Trees : 1

Enter Name of Section 1 : Symbol Table

Enter the name of Chapter 5 : Indexing & Multiway Trees

Enter number of sections in Chapter Indexing & Multiway Trees : 2

Enter Name of Section 1 : Indexing techniques

Enter Name of Section 2 : B-Tree

-----------------

Book Tree Creation

-----------------

1.Create

2.Display

3.Quit

Enter your choice : 2

-----Book Hierarchy---

Book title : Data Structures and Algorithm

Chapter 1 : Hashing

Sections :

Hash Table

Skip List

Chapter 2 : Trees

Sections :

General Tree

Binary Tree

Huffman Tree

Chapter 3 : Graphs

Sections :

Traversals

Chapter 4 : Search Trees

Sections :

Symbol Table

Chapter 5 : Indexing & Multiway Trees

Sections :

Indexing techniques

B-Tree

-----------------

Book Tree Creation

-----------------

1.Create

2.Display

3.Quit

Enter your choice : 3

Thanks for using this program!!!

\*/

**Practical No: 4**

**Title: Implementation of Binary Search tree**

**CODE:**

#include <iostream>

using namespace std;

// Tree node structure

struct Tree {

int data; // Data held by the node

Tree \*left; // Pointer to the left child

Tree \*right; // Pointer to the right child

};

class bstree {

public:

// Create a new tree node with the given data

Tree\* create(int data) {

Tree\* tempTree = new Tree;

tempTree->left = NULL;

tempTree->right = NULL;

tempTree->data = data;

return tempTree;

}

// Set the left child of a given tree node

void setLeft(Tree\* aTree, int data) {

aTree->left = create(data); // Create left child and assign it

}

// Set the right child of a given tree node

void setRight(Tree\* aTree, int data) {

aTree->right = create(data); // Create right child and assign it

}

// Insert a new node with given data into the tree

void insert(Tree\* aTree, int data) {

while (aTree != NULL) { // Traverse down the tree

// If the data is less than or equal, go to left subtree

if (data <= aTree->data) {

if (aTree->left != NULL) {

aTree = aTree->left; // Move to the left child

} else {

setLeft(aTree, data); // Create left child

break;

}

} else { // If the data is greater, go to right subtree

if (aTree->right != NULL) {

aTree = aTree->right; // Move to the right child

} else {

setRight(aTree, data); // Create right child

break;

}

}

}

}

// Inorder traversal of the tree (left, root, right)

void inOrderTraverse(Tree \*aTree) {

if (aTree == NULL) // Base condition: if tree is empty, return

return;

inOrderTraverse(aTree->left); // Traverse left subtree

cout << " " << aTree->data; // Print the root data

inOrderTraverse(aTree->right); // Traverse right subtree

}

// Calculate the height of the tree

int height(Tree \*aTree) {

if (aTree == NULL) { // Base case: tree is empty

return -1;

}

// Recursively find the height of left and right subtrees

int hl = height(aTree->left);

int hr = height(aTree->right);

return 1 + max(hl, hr); // Return the greater height plus 1 (for the root)

}

// Swap the left and right children of all nodes in the tree

void swap(Tree \*aTree) {

if (aTree != NULL) { // If the tree is not empty

swap(aTree->left); // Recursively swap left subtree

swap(aTree->right); // Recursively swap right subtree

Tree \*temp = aTree->left; // Temporary variable to swap

aTree->left = aTree->right;

aTree->right = temp; // Swap the left and right pointers

}

}

// Find the minimum value in the tree

int minValue(Tree \*aTree) {

if (aTree == NULL) return -1; // Return -1 if tree is empty

// Traverse down to the leftmost leaf node

while (aTree->left != NULL) {

aTree = aTree->left;

}

return aTree->data; // Return the data of the leftmost leaf node

}

// Search for a value in the tree (works after swapping too)

bool search(Tree \*aTree, int value) {

if (aTree == NULL) return false; // Return false if tree is empty

if (aTree->data == value) return true; // If the value is found

// Recursively search in the left and right subtrees

return search(aTree->left, value) || search(aTree->right, value);

}

};

int main() {

bstree bs; // Create a bstree object

int ch = 0, value;

Tree \*myTree = NULL; // Initialize the tree as empty

// Display the menu until the user chooses to exit

while (ch != 8) {

cout << "\n 1. Create";

cout << "\n 2. Insert";

cout << "\n 3. Display (Inorder Traversal)";

cout << "\n 4. Find height of the tree";

cout << "\n 5. Find minimum value of tree";

cout << "\n 6. Swap left and right children";

cout << "\n 7. Search";

cout << "\n 8. Exit";

cout << "\n Enter your choice: ";

cin >> ch;

switch (ch) {

case 1: // Case for creating the tree

cout << "Enter root: ";

cin >> value;

myTree = bs.create(value); // Create the root node

break;

case 2: // Case for inserting a new value

if (myTree == NULL) {

cout << "Please create the tree first!" << endl;

} else {

cout << "Enter value: ";

cin >> value;

bs.insert(myTree, value); // Insert the new value

}

break;

case 3: // Case for displaying inorder traversal

if (myTree == NULL) {

cout << "Tree is empty!" << endl;

} else {

cout << "Inorder Traversal:";

bs.inOrderTraverse(myTree); // Display inorder traversal

cout << endl;

}

break;

case 4: // Case for finding the height of the tree

if (myTree == NULL) {

cout << "Tree is empty!" << endl;

} else {

cout << "\n Height of the tree: " << bs.height(myTree) << endl;

}

break;

case 5: // Case for finding the minimum value in the tree

if (myTree == NULL) {

cout << "Tree is empty!" << endl;

} else {

cout << "\n Minimum Data Value in the tree: " << bs.minValue(myTree) << endl;

}

break;

case 6: // Case for swapping left and right children

if (myTree == NULL) {

cout << "Tree is empty!" << endl;

} else {

bs.swap(myTree); // Swap left and right children

cout << "\n Tree after swapping:";

bs.inOrderTraverse(myTree); // Display inorder traversal after swap

cout << endl;

}

break;

case 7: // Case for searching a value in the tree

if (myTree == NULL) {

cout << "Tree is empty!" << endl;

} else {

cout << "\n Enter the value to search: ";

cin >> value;

if (bs.search(myTree, value)) {

cout << "Found " << value << " in tree" << endl;

} else {

cout << "Could not find " << value << " in tree" << endl;

}

}

break;

case 8: // Case to exit the program

cout << "Exiting program..." << endl;

break;

default: // Invalid input handling

cout << "Enter a valid input!" << endl;

}

}

return 0;

}

**Practical No:5**

**Title:** Implementation of Expression Tree from the given Prefix Expression.

CODE:

#include <iostream>

#include <string.h>

using namespace std;

// Define the node structure for the binary tree

struct node {

char data; // Character data stored in each node

node \*left; // Pointer to the left child

node \*right; // Pointer to the right child

};

// Tree class to represent the expression tree and its operations

class tree {

public:

node \*top; // Top of the tree (root node)

// Method to build the expression tree from a prefix expression

void expression(char[]);

// Method to display the tree in pre-order

void display(node \*);

// Method to perform non-recursive post-order traversal

void non\_rec\_postorder(node \*);

// Method to delete the tree (free memory)

void del(node \*);

};

// Stack class to handle stack operations for nodes (used to build the expression tree)

class stack1 {

node \*data[30]; // Stack to store node pointers

int top; // Top index for the stack

public:

stack1() {

top = -1; // Initialize the stack as empty

}

// Method to check if the stack is empty

int empty() {

return top == -1; // Return 1 if empty, 0 otherwise

}

// Method to push a node onto the stack

void push(node \*p) {

data[++top] = p; // Increment top and push the node

}

// Method to pop a node from the stack

node \*pop() {

return data[top--]; // Return the node and decrement top

}

};

// Method to build the expression tree from a prefix expression

void tree::expression(char prefix[]) {

stack1 s; // Create a stack to hold nodes

node \*t1, \*t2;

int len = strlen(prefix); // Length of the prefix expression

// Traverse the prefix expression from right to left

for (int i = len - 1; i >= 0; i--) {

top = new node; // Create a new node

top->left = NULL; // Set left child to NULL

top->right = NULL; // Set right child to NULL

// If the character is an operand, create a leaf node

if (isalpha(prefix[i])) {

top->data = prefix[i]; // Set node data to the character

s.push(top); // Push the node onto the stack

}

// If the character is an operator, create an internal node

else if (prefix[i] == '+' || prefix[i] == '\*' || prefix[i] == '-' || prefix[i] == '/') {

t2 = s.pop(); // Pop two nodes from the stack

t1 = s.pop();

top->data = prefix[i]; // Set node data to the operator

top->left = t2; // Set left child to the second operand

top->right = t1; // Set right child to the first operand

s.push(top); // Push the newly created operator node onto the stack

}

}

top = s.pop(); // The final node in the stack is the root of the tree

}

// Method to display the tree in pre-order traversal (root, left, right)

void tree::display(node \*root) {

if (root != NULL) {

cout << root->data; // Display the node's data

display(root->left); // Traverse the left subtree

display(root->right); // Traverse the right subtree

}

}

// Method to perform non-recursive post-order traversal (left, right, root)

void tree::non\_rec\_postorder(node \*top) {

stack1 s1, s2; // Create two stacks for post-order traversal

node \*T = top;

cout << "\n";

s1.push(T); // Push the root onto the first stack

// Traverse the tree using stack s1

while (!s1.empty()) {

T = s1.pop(); // Pop a node from s1

s2.push(T); // Push the node onto s2

if (T->left != NULL)

s1.push(T->left); // Push left child if it exists

if (T->right != NULL)

s1.push(T->right); // Push right child if it exists

}

// Pop nodes from s2 and print their data in post-order

while (!s2.empty()) {

top = s2.pop();

cout << top->data;

}

}

// Method to delete the tree and free memory

void tree::del(node \*node) {

if (node == NULL) // If the node is NULL, return

return;

// Recursively delete the left and right subtrees

del(node->left);

del(node->right);

// Display message and free memory for the current node

cout << endl << "Deleting node : " << node->data << endl;

delete node; // Use delete instead of free in C++

}

int main() {

char expr[20]; // Array to hold the prefix expression

tree t; // Create a tree object

cout << "Enter prefix Expression : ";

cin >> expr; // Input the prefix expression

cout << expr;

t.expression(expr); // Build the expression tree

// Perform non-recursive post-order traversal and display the result

cout << endl;

t.non\_rec\_postorder(t.top);

// Delete the tree and free the allocated memory

t.del(t.top);

}

**Practical No: 6**

**Title: Implemntation of a graph data structure**

**CODE:**

DFS:

#include <iostream>

#include <stdlib.h>

using namespace std;

// Declare the adjacency matrix and other global variables for graph representation

int cost[10][10], i, j, k, n, u, v; // cost matrix, loop variables, and graph parameters

int stk[10], top, visit1[10], visited1[10]; // stack for DFS, visited flags

int main()

{

int m;

// Prompt the user for the number of vertices in the graph

cout << "Enter number of vertices : ";

cin >> n; // Number of vertices (nodes) in the graph

// Prompt the user for the number of edges in the graph

cout << "Enter number of edges : ";

cin >> m; // Number of edges (connections between vertices)

// Input the edges of the graph and update the adjacency matrix

cout << "\nEDGES :\n";

for (k = 1; k <= m; k++) // Loop to input edges one by one

{

// Ask the user for the pair of vertices that form an edge (U, V)

cout << "Enter U and V:";

cin >> i >> j; // i and j are the vertices that are connected by an edge

cost[i][j] = 1; // Mark an edge between vertex i and j

cost[j][i] = 1; // Since the graph is undirected, mark the reverse edge as well

}

// Display the adjacency matrix representation of the graph

cout << "The adjacency matrix of the graph is : " << endl;

for (i = 1; i <= n; i++) // Loop to display the matrix row by row

{

for (j = 1; j <= n; j++) // Loop to display each element in the row

{

cout << " " << cost[i][j]; // Print the value at position (i, j) in the matrix

}

cout << endl; // Move to the next line after printing a row

}

// Prompt the user to enter the initial vertex for DFS traversal

cout << endl << "Enter initial vertex : ";

cin >> v; // Starting vertex for DFS traversal

// Print the starting vertex and begin the DFS traversal

cout << "The DFS of the Graph is\n";

cout << v; // Print the initial vertex

visited1[v] = 1; // Mark the initial vertex as visited

k = 1; // Initialize a counter to track the number of visited vertices

// Stack initialization for DFS traversal

top = 0; // Initialize the stack top pointer to 0

// Perform the DFS traversal using the stack

while (k < n) // Loop until all vertices are visited

{

// Look for unvisited adjacent vertices of the current vertex

for (j = n; j >= 1; j--) // Iterate through all vertices in reverse order (from n to 1)

if (cost[v][j] != 0 && visited1[j] != 1 && visit1[j] != 1) // Check if there's an edge and the vertex is unvisited

{

visit1[j] = 1; // Mark the vertex as part of the stack (to visit later)

stk[top] = j; // Push the unvisited vertex onto the stack

top++; // Increment the stack top index

}

// Pop the next vertex from the stack and make it the current vertex for DFS

v = stk[--top]; // Decrement the stack top index and get the next vertex from the stack

cout << v << " "; // Print the current vertex as part of the DFS traversal

k++; // Increment the counter for visited vertices

visit1[v] = 0; // Mark the current vertex as no longer in the stack

visited1[v] = 1; // Mark the current vertex as visited

}

return 0; // End the program

}

BFS:

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// Define a Graph class

class Graph {

private:

// Adjacency matrix to store the graph (2D array)

vector<vector<int>> adjMatrix;

public:

// Constructor that initializes the adjacency matrix with all values set to 0

Graph(int numVertices) : adjMatrix(numVertices, vector<int>(numVertices, 0)) {}

// Function to print the adjacency matrix of the graph

void printAdjMatrix() {

for (int i = 0; i < adjMatrix.size(); ++i) { // Loop through each row (vertex)

for (int j = 0; j < adjMatrix[i].size(); ++j) { // Loop through each column (connected vertex)

cout << adjMatrix[i][j] << " "; // Print the value in the matrix

}

cout << endl; // Print a newline after each row

}

}

// Function to add an edge between two vertices (i and j) in the adjacency matrix

void addEdge(int i, int j) {

adjMatrix[i][j] = 1; // Mark an edge between i and j

adjMatrix[j][i] = 1; // Since the graph is undirected, also mark the reverse edge

}

// Function to perform Breadth-First Search (BFS) starting from the specified vertex

void bfs(int startVertex) {

vector<bool> visited(adjMatrix.size(), false); // Create a visited array initialized to false

queue<int> q; // Queue to store vertices to be visited

visited[startVertex] = true; // Mark the start vertex as visited

q.push(startVertex); // Add the start vertex to the queue

while (!q.empty()) { // Continue as long as there are vertices in the queue

int currentVertex = q.front(); // Get the vertex at the front of the queue

cout << currentVertex << " "; // Print the vertex being visited

q.pop(); // Remove the vertex from the queue

// Loop through all vertices to check for adjacent vertices to visit

for (int i = 0; i < adjMatrix.size(); ++i) {

if (adjMatrix[currentVertex][i] == 1 && !visited[i]) { // If there's an edge and the vertex is unvisited

visited[i] = true; // Mark the adjacent vertex as visited

q.push(i); // Add the adjacent vertex to the queue

}

}

}

}

};

int main() {

int n, e, k, i, j, start;

// Ask the user for the number of vertices and edges in the graph

cout << "Enter no of vertices: ";

cin >> n; // Read the number of vertices

cout << "Enter no of edges: ";

cin >> e; // Read the number of edges

Graph g(n); // Create a graph object with n vertices

// Input the edges and add them to the graph

for (k = 0; k < e; k++) {

cout << "Enter U and V: ";

cin >> i >> j; // Read the two vertices U and V

g.addEdge(i, j); // Add the edge between U and V to the graph

}

// Display the adjacency matrix of the graph

g.printAdjMatrix();

// Ask the user for the starting vertex for BFS traversal

cout << "Enter the starting vertex: ";

cin >> start; // Read the starting vertex

// Perform BFS and display the result

cout << "BFS starting from vertex: " << start << endl;

g.bfs(start); // Perform BFS starting from the specified vertex

return 0; // End of the program

}

**Practical No:7**

**Title: Implementation of minimum spanning Tree**

**CODE:**

#include<iostream> // Include input-output stream library

using namespace std; // Use the standard namespace

int main()

{

int n, i, j, k, row, col, mincost = 0, min; // Declare necessary variables

char op; // For user input 'y' or 'n'

cout << "Enter no. of vertices: "; // Ask for number of vertices

cin >> n; // Take input for number of vertices

int cost[n][n]; // Create adjacency matrix to store weights

int visit[n]; // Array to track visited vertices

for(i = 0; i < n; i++) // Initialize all vertices as unvisited

visit[i] = 0; // Set visited value to 0

for(i = 0; i < n; i++) // Loop through matrix rows

for(int j = 0; j < n; j++) // Loop through matrix columns

cost[i][j] = -1; // Initialize cost with -1 (no edge)

for(i = 0; i < n; i++) // Input edge weights from user

{

for(j = i + 1; j < n; j++) // Avoid duplicate and self edges

{

cout << "Do you want an edge between " << i << " and " << j << ": "; // Ask user if edge exists

cin >> op; // Read user choice

if(op == 'y' || op == 'Y') // If user wants the edge

{

cout << "Enter weight: "; // Ask for edge weight

cin >> cost[i][j]; // Store weight in matrix

cost[j][i] = cost[i][j]; // Make the matrix symmetric (undirected)

}

}

}

visit[0] = 1; // Start from vertex 0, mark it visited

for(k = 0; k < n - 1; k++) // Loop to find n-1 edges for MST

{

min = 999; // Initialize minimum to a large value

for(i = 0; i < n; i++) // Loop through all vertices

{

for(j = 0; j < n; j++) // Loop through all possible connections

{

if(visit[i] == 1 && visit[j] == 0) // If i is visited and j is not

{

if(cost[i][j] != -1 && min > cost[i][j]) // Check if edge exists and is smallest

{

min = cost[i][j]; // Update minimum

row = i; // Store source vertex of edge

col = j; // Store destination vertex of edge

}

}

}

}

mincost += min; // Add minimum edge cost to total cost

visit[col] = 1; // Mark new vertex as visited

cost[row][col] = cost[col][row] = -1; // Remove the edge from matrix (avoid reusing it)

cout << row << "->" << col << endl; // Print selected edge

}

cout << "\nMin. Cost: " << mincost; // Print total cost of the MST

return 0; // Return from main

}

**Practical NO: 8**

**Title:Implementation of an optimal binary serach tree data structure**

**CODE:**

#include<iostream> // Include input-output stream

using namespace std;

#define SIZE 10 // Define constant SIZE as 10

class OBST

{

int p[SIZE]; // Array to hold successful search probabilities

int q[SIZE]; // Array to hold unsuccessful search probabilities

int a[SIZE]; // Array to hold keys

int w[SIZE][SIZE]; // Weight matrix

int c[SIZE][SIZE]; // Cost matrix

int r[SIZE][SIZE]; // Root matrix

int n; // Number of nodes

public:

void get\_data() // Function to input keys and probabilities

{

int i;

cout << "\n Optimal Binary Search Tree \n";

cout << "\n Enter the number of nodes: ";

cin >> n;

cout << "\n Enter the data as…\n";

for(i = 1; i <= n; i++) // Input the keys

{

cout << "\n a[" << i << "]: ";

cin >> a[i];

}

for(i = 1; i <= n; i++) // Input successful probabilities

{

cout << "\n p[" << i << "]: ";

cin >> p[i];

}

for(i = 0; i <= n; i++) // Input unsuccessful probabilities

{

cout << "\n q[" << i << "]: ";

cin >> q[i];

}

}

int Min\_Value(int i, int j) // Function to find index with minimum cost

{

int m, k;

int minimum = 32000; // Start with large value

for(m = r[i][j - 1]; m <= r[i + 1][j]; m++) // Loop from left root to right root

{

if((c[i][m - 1] + c[m][j]) < minimum) // Check if new min found

{

minimum = c[i][m - 1] + c[m][j]; // Update min

k = m; // Store root index

}

}

return k; // Return root index with minimum cost

}

void build\_OBST() // Function to build OBST using dynamic programming

{

int i, j, k, l, m;

for(i = 0; i < n; i++)

{

w[i][i] = q[i]; // Weight of empty subtree

r[i][i] = c[i][i] = 0; // Root and cost of empty tree is 0

w[i][i + 1] = q[i] + q[i + 1] + p[i + 1]; // Weight of subtree with 1 key

r[i][i + 1] = i + 1; // Root is the key itself

c[i][i + 1] = w[i][i + 1]; // Cost is equal to weight for single node

}

w[n][n] = q[n]; // For last element

r[n][n] = c[n][n] = 0; // Empty subtree

for(m = 2; m <= n; m++) // For chain length from 2 to n

{

for(i = 0; i <= n - m; i++) // i is starting index

{

j = i + m; // j is ending index

w[i][j] = w[i][j - 1] + p[j] + q[j]; // Compute weight

k = Min\_Value(i, j); // Find root that gives minimum cost

c[i][j] = w[i][j] + c[i][k - 1] + c[k][j]; // Compute total cost

r[i][j] = k; // Store root index

}

}

}

void build\_tree() // Function to display structure of OBST

{

int i, j, k;

int queue[20], front = -1, rear = -1;

cout << "The Optimal Binary Search Tree For the Given Node Is…\n";

cout << "\n The Root of this OBST is: " << r[0][n]; // Display root

cout << "\nThe Cost of this OBST is: " << c[0][n]; // Display total cost

cout << "\n\n\t NODE \t LEFT CHILD \t RIGHT CHILD \n";

queue[++rear] = 0; // Initialize queue for level-order display

queue[++rear] = n;

while(front != rear) // While queue not empty

{

i = queue[++front]; // Get start index

j = queue[++front]; // Get end index

k = r[i][j]; // Get root of subtree

cout << "\n\t" << k; // Print current node

if(r[i][k - 1] != 0) // If left child exists

{

cout << "\t\t" << r[i][k - 1]; // Print left child

queue[++rear] = i; // Add to queue for traversal

queue[++rear] = k - 1;

}

else

cout << "\t\t";

if(r[k][j] != 0) // If right child exists

{

cout << "\t" << r[k][j]; // Print right child

queue[++rear] = k;

queue[++rear] = j;

}

else

cout << "\t";

}

cout << "\n";

}

};

int main()

{

OBST obj; // Create object of class OBST

obj.get\_data(); // Input data

obj.build\_OBST(); // Build OBST using dynamic programming

obj.build\_tree(); // Display OBST structure

return 0; // Return from main

}

**Practical No: 9**

**Title: Implementation of a hight balanced tree data structure**

**CODE:**

#include<iostream> // For input/output stream operations

#include<cstring> // For using string handling functions like strcpy, strcmp

#include<cstdlib> // For using exit() function

#define MAX 50 // Maximum size for meaning string

#define SIZE 20 // Maximum size for word string

using namespace std; // Use standard namespace

// Structure to represent a node in AVL tree

struct AVLnode

{

char cWord[SIZE], cMeaning[MAX]; // Word and its meaning

AVLnode \*left, \*right; // Pointers to left and right child nodes

int iHt; // Height of the node

};

// AVL tree class

class AVLtree

{

public:

AVLnode \*root; // Root of AVL tree

// Constructor to initialize root

AVLtree() {

root = NULL;

}

// Function declarations

int height(AVLnode\*); // Returns height of node

int bf(AVLnode\*); // Returns balance factor

AVLnode\* insert(AVLnode\*, char[SIZE], char[MAX]); // Insert a word

AVLnode\* rotate\_left(AVLnode\*); // Left rotation

AVLnode\* rotate\_right(AVLnode\*); // Right rotation

AVLnode\* LL(AVLnode\*); // Left-Left case

AVLnode\* RR(AVLnode\*); // Right-Right case

AVLnode\* LR(AVLnode\*); // Left-Right case

AVLnode\* RL(AVLnode\*); // Right-Left case

AVLnode\* delet(AVLnode\*, char[SIZE]); // Delete a word

void inorder(AVLnode\*); // Inorder traversal

};

// Function to return height of node

int AVLtree::height(AVLnode\* curr) {

if (curr == NULL) return -1; // Height of null node is -1

int lh = height(curr->left); // Height of left subtree

int rh = height(curr->right); // Height of right subtree

return max(lh, rh) + 1; // Return max height + 1

}

// Function to calculate balance factor

int AVLtree::bf(AVLnode\* curr) {

if (curr == NULL) return 0; // Balance factor of null node is 0

return height(curr->left) - height(curr->right); // left height - right height

}

// Function to perform right rotation

AVLnode\* AVLtree::rotate\_right(AVLnode\* curr) {

AVLnode\* temp = curr->left; // Store left child in temp

curr->left = temp->right; // Make temp's right child as curr's left child

temp->right = curr; // Make curr as right child of temp

curr->iHt = height(curr); // Update height of curr

temp->iHt = height(temp); // Update height of temp

return temp; // Return new root

}

// Function to perform left rotation

AVLnode\* AVLtree::rotate\_left(AVLnode\* curr) {

AVLnode\* temp = curr->right; // Store right child in temp

curr->right = temp->left; // Make temp's left child as curr's right child

temp->left = curr; // Make curr as left child of temp

curr->iHt = height(curr); // Update height of curr

temp->iHt = height(temp); // Update height of temp

return temp; // Return new root

}

// Left-Left rotation case

AVLnode\* AVLtree::LL(AVLnode\* curr) {

return rotate\_right(curr); // Perform right rotation

}

// Right-Right rotation case

AVLnode\* AVLtree::RR(AVLnode\* curr) {

return rotate\_left(curr); // Perform left rotation

}

// Left-Right rotation case

AVLnode\* AVLtree::LR(AVLnode\* curr) {

curr->left = rotate\_left(curr->left); // Perform left rotation on left child

return rotate\_right(curr); // Then right rotation on current node

}

// Right-Left rotation case

AVLnode\* AVLtree::RL(AVLnode\* curr) {

curr->right = rotate\_right(curr->right); // Perform right rotation on right child

return rotate\_left(curr); // Then left rotation on current node

}

// Function to insert a word and its meaning into AVL tree

AVLnode\* AVLtree::insert(AVLnode\* root, char newword[SIZE], char newmeaning[MAX]) {

if (root == NULL) { // If tree is empty

root = new AVLnode; // Create new node

root->left = root->right = NULL; // Initialize child pointers

strcpy(root->cWord, newword); // Copy word

strcpy(root->cMeaning, newmeaning); // Copy meaning

root->iHt = 0; // Set height to 0

return root; // Return new root

}

// If new word is smaller, go to left subtree

if (strcmp(root->cWord, newword) > 0) {

root->left = insert(root->left, newword, newmeaning);

if (bf(root) == 2) { // Check balance

if (strcmp(root->left->cWord, newword) > 0)

root = LL(root); // LL rotation

else

root = LR(root); // LR rotation

}

}

// If new word is larger, go to right subtree

else if (strcmp(root->cWord, newword) < 0) {

root->right = insert(root->right, newword, newmeaning);

if (bf(root) == -2) { // Check balance

if (strcmp(root->right->cWord, newword) < 0)

root = RR(root); // RR rotation

else

root = RL(root); // RL rotation

}

}

else {

cout << "\nDuplicate word, not inserting!"; // If duplicate word

}

root->iHt = height(root); // Update height

return root; // Return root

}

// Function to delete a word from AVL tree

AVLnode\* AVLtree::delet(AVLnode \*curr, char x[SIZE]) {

if (curr == NULL) // If tree is empty

return NULL;

// If word is greater, go right

if (strcmp(x, curr->cWord) > 0) {

curr->right = delet(curr->right, x);

}

// If word is smaller, go left

else if (strcmp(x, curr->cWord) < 0) {

curr->left = delet(curr->left, x);

}

// If word found

else {

// If it's a leaf node

if (curr->left == NULL && curr->right == NULL) {

delete curr;

return NULL;

}

// If node has only right child

else if (curr->left == NULL) {

AVLnode\* temp = curr->right;

delete curr;

return temp;

}

// If node has only left child

else if (curr->right == NULL) {

AVLnode\* temp = curr->left;

delete curr;

return temp;

}

// If node has both children

else {

AVLnode\* temp = curr->right;

while (temp->left != NULL) temp = temp->left; // Find inorder successor

strcpy(curr->cWord, temp->cWord); // Replace with successor

strcpy(curr->cMeaning, temp->cMeaning);

curr->right = delet(curr->right, temp->cWord); // Delete successor

}

}

curr->iHt = height(curr); // Update height

// Rebalance the tree

if (bf(curr) == 2) {

if (bf(curr->left) >= 0)

return LL(curr);

else

return LR(curr);

}

if (bf(curr) == -2) {

if (bf(curr->right) <= 0)

return RR(curr);

else

return RL(curr);

}

return curr;

}

// Function for inorder traversal (sorted order)

void AVLtree::inorder(AVLnode\* curr) {

if (curr != NULL) {

inorder(curr->left); // Visit left subtree

cout << "\n\t" << curr->cWord << "\t" << curr->cMeaning; // Print node

inorder(curr->right); // Visit right subtree

}

}

// Main function

int main() {

int iCh; // Variable to store menu choice

AVLtree a; // Create AVL tree object

char cWd[SIZE], cMean[MAX]; // Word and Meaning

cout << "\n--------------------------------------";

cout << "\n\tAVL TREE IMPLEMENTATION"; // Heading

cout << "\n--------------------------------------";

do {

// Display menu

cout << "\n--------------------------------";

cout << "\n\t\tMENU";

cout << "\n--------------------------------";

cout << "\n1.Insert\n2.Inorder\n3.Delete\n4.Exit";

cout << "\n--------------------------------";

cout << "\nEnter your choice :";

cin >> iCh; // Read user choice

switch (iCh) {

case 1:

cout << "\nEnter Word : ";

cin >> cWd; // Read word

cout << "\nEnter Meaning : ";

cin.ignore(); // Clear input buffer

cin.getline(cMean, MAX); // Read meaning (can include spaces)

a.root = a.insert(a.root, cWd, cMean); // Insert into tree

break;

case 2:

cout << "\n\tWORD\tMEANING";

a.inorder(a.root); // Display dictionary in sorted order

break;

case 3:

cout << "\nEnter the word to be deleted : ";

cin >> cWd; // Read word to delete

a.root = a.delet(a.root, cWd); // Delete from tree

cout << "\nWord deleted successfully!";

break;

case 4:

exit(0); // Exit program

}

} while (iCh != 4); // Continue until user chooses to exit

return 0;

}

**Practical No: 10**

**Title: Implementation of a priority queue as ADT**

**CODE:**

#include<iostream> // For input/output operations

#include<string> // For using string type

// #define N 20 // (Unused define) Could define queue size here

using namespace std;

string Q[10]; // Array to store patient names (queue data)

int Pr[10]; // Array to store priorities corresponding to patients

int r = -1, f = -1, n; // r = rear, f = front, n = number of patients

// Function to insert patient data with priority into the queue

void enqueue(string data, int p) {

int i;

if ((f == 0) && (r == n - 1)) // Check if queue is full

cout << "Queue is full";

else {

if (f == -1) { // If queue is empty

f = r = 0; // Initialize front and rear to 0

Q[r] = data; // Insert patient name

Pr[r] = p; // Insert priority

} else {

for (i = r; i >= f; i--) { // Find correct position based on priority

if (p > Pr[i]) { // Higher priority values mean more urgent

Q[i + 1] = Q[i]; // Shift data to make space

Pr[i + 1] = Pr[i]; // Shift corresponding priority

} else break; // Stop if current priority is higher or equal

}

Q[i + 1] = data; // Insert patient name at correct position

Pr[i + 1] = p; // Insert corresponding priority

r++; // Increment rear

}

}

}

// Function to print the queue with patient names and their priorities

void print() {

int i;

for (i = f; i <= r; i++) { // Traverse from front to rear

cout << "Patient's Name - " << Q[i]; // Print patient name

switch (Pr[i]) { // Print priority meaning

case 1:

cout << " Priority - 'Checkup' " << endl;

break;

case 2:

cout << " Priority - 'Non-serious' " << endl;

break;

case 3:

cout << " Priority - 'Serious' " << endl;

break;

default:

cout << " Priority not found" << endl;

}

}

}

// Function to delete the front element of the queue

void dequeue() {

if (f == -1) { // If queue is empty

cout << "Queue is Empty";

} else {

cout << "Deleted Element = " << Q[f] << endl; // Show deleted name

cout << "Its Priority = " << Pr[f] << endl; // Show its priority

if (f == r) f = r = -1; // If only one element was present

else f++; // Move front to next element

}

}

int main() {

string data; // Variable to store patient name

int opt, i, p; // opt = menu option, p = priority

cout << "Enter Your Choice:-" << endl;

do {

// Display menu options

cout << "1 for Insert the Data in Queue" << endl;

cout << "2 for show the Data in Queue " << endl;

cout << "3 for Delete the data from the Queue" << endl;

cout << "0 for Exit" << endl;

cin >> opt; // Take user's choice

switch (opt) {

case 1:

cout << "Enter the number of patients" << endl;

cin >> n; // Read number of patients to insert

i = 0;

for (i = 0; i < n; i++) {

cout << "Enter your name of the patient : ";

cin >> data; // Read patient name

cout << "Enter your Prioritys (3: serious, 2: non-serious, 1: general checkup) : ";

cin >> p; // Read priority

enqueue(data, p); // Call enqueue function

}

break;

case 2:

print(); // Show queue contents

break;

case 3:

dequeue(); // Remove front element from queue

break;

}

} while (opt != 0); // Repeat menu until user exits

return 0; // Exit program

}

**Practical No: 11**

**Title: Implementation of sequential file organization concept using cpp**

**CODE:**

#include<iostream> // Required for input/output

#include<fstream> // Required for file handling

#include<cstring> // Required for string operations

#include<stdlib.h> // Required for general utilities (like remove, rename)

using namespace std;

// Define a class to manage student operations

class Student

{

// Define a structure to hold individual student record

typedef struct studentinfo

{

char name[50]; // Student name

int rollno; // Student roll number

char division[5]; // Division of student

char address[100]; // Address of student

} rec;

rec records; // Object of structure to hold student data

public:

void create(); // Function to create/add new student records

void display(); // Function to display all student records

void search(); // Function to search student record by roll number

void Delete(int a); // Function to delete student record by roll number

};

// Function to create and write student records to file

void Student::create()

{

char ch = 'y';

fstream seq;

seq.open("StudentRecord.txt", ios::out); // Open file for writing

do {

// Take input from user for student data

cout << "Enter name : ";

cin >> records.name;

cout << "Enter roll number : ";

cin >> records.rollno;

cout << "Enter division : ";

cin >> records.division;

cout << "Enter address : ";

cin >> records.address;

// Write the structure object to file

seq.write((char\*)&records, sizeof(records));

cout << "\nDo you want to add more records: ";

cin >> ch;

} while (ch == 'y'); // Repeat until user says no

seq.close(); // Close the file

}

// Function to display all student records

void Student::display()

{

fstream seq;

int n;

seq.open("StudentRecord.txt", ios::in); // Open file for reading

seq.seekg(0, ios::beg); // Set pointer to beginning

cout << "\nContent of file are... " << endl;

// Read and print each record until EOF

while (seq.read((char\*)&records, sizeof(records)))

{

if (records.rollno != -1) // Skip deleted (invalid) records

{

cout << "\nName: " << records.name;

cout << "\nRoll No: " << records.rollno;

cout << "\nDivision: " << records.division;

cout << "\nAddress: " << records.address << endl;

}

}

int lastrecord = seq.tellg(); // Current position (end of file)

n = lastrecord / (sizeof(rec)); // Total number of records (optional use)

}

// Function to search student by roll number

void Student::search()

{

fstream seq;

int id;

cout << "\nEnter the roll number to search: ";

cin >> id;

seq.open("StudentRecord.txt", ios::in | ios::binary); // Open for binary reading

seq.seekg(0, ios::beg);

bool found = false;

// Read and search for matching roll number

while (seq.read((char\*)&records, sizeof(records)))

{

if (records.rollno == id)

{

found = true;

cout << "Student record found";

cout << "\nRoll Number: " << records.rollno << endl;

cout << "Name: " << records.name << endl;

cout << "Division: " << records.division << endl;

cout << "Address: " << records.address << endl;

break; // Stop after finding the record

}

}

seq.close(); // Close the file

if (!found)

{

cout << "Roll No: " << id << " is not found!" << endl;

}

}

// Function to delete student by roll number

void Student::Delete(int id)

{

ifstream infile;

ofstream outfile;

infile.open("StudentRecord.txt", ios::in); // Open original file to read

outfile.open("temp.txt", ios::app); // Create temp file for writing

infile.seekg(0, ios::beg); // Start from beginning

bool flag = false;

// Copy all records except the one to delete

while (infile.read((char\*)&records, sizeof(records)))

{

if (records.rollno == id)

{

flag = true; // Mark as found

continue; // Skip writing the record to delete

}

outfile.write((char\*)&records, sizeof(records)); // Write valid records

}

infile.close();

outfile.close();

if (flag == false)

{

remove("temp.txt"); // No matching record, remove temp

cout << "\nRoll no: " << id << " is not present in record.";

}

else

{

remove("StudentRecord.txt"); // Delete old file

rename("temp.txt", "StudentRecord.txt"); // Rename temp file as original

cout << "Record deleted successfully.";

}

}

// Main function to handle user interaction

int main()

{

Student s;

char ans = 'y';

int ch, id;

do

{

// Display menu

cout << "\n1. Create";

cout << "\n2. Display";

cout << "\n3. Search";

cout << "\n4. Delete";

cout << "\n5. Exit";

cout << "\nEnter your choice: ";

cin >> ch;

switch (ch)

{

case 1: s.create(); // Add new students

break;

case 2: s.display(); // Show all records

break;

case 3: s.search(); // Search by roll number

break;

case 4:

cout << "Enter the roll no to delete: ";

cin >> id;

s.Delete(id); // Delete by roll number

break;

case 5:

exit(0); // Exit the program

default:

cout << "\nEnter valid choice"; // Invalid menu choice

break;

}

cout << "\nDo you want to go back to main menu? (y/n): ";

cin >> ans;

} while (ans == 'y'); // Repeat until user exits

return 0;

}

**Practical No:12**

**Title: Implementation of index sequential file organization concept using cpp**

**CODE:**

#include<iostream> // For input and output operations

#include<fstream> // For file stream operations

#include <sstream> // For parsing strings (used with getline and istringstream)

#include <string> // For using the string class

using namespace std; // Use the standard namespace

// Class to manage employee records

class employee

{

// Define a structure to hold employee information

typedef struct empinfo

{

int empid; // Employee ID

char empName[50]; // Employee Name

char empDesignation[50]; // Employee Designation

float empSalary; // Employee Salary

} rec;

rec records; // Structure variable to store one employee's data

public:

void create(); // Function to create and add employee records

void Delete(int id); // Function to delete employee by ID

void display(); // Function to display employee record by ID

void print(); // Helper function to print employee details

};

// Function to print employee record stored in 'records'

void employee::print()

{

cout << "------------Details of Employee---------" << endl;

cout << "Employee ID: " << records.empid << endl;

cout << "Employee Name: " << records.empName << endl;

cout << "Employee Designation: " << records.empDesignation << endl;

cout << "Employee Salary: " << records.empSalary << endl;

}

// Static function to find the file position of an employee record using employee ID from the index file

static int findEmployeePosition(int employeeID)

{

ifstream indexFile("index.txt"); // Open the index file to search for position

if (!indexFile) // Check if file is opened successfully

{

cout << "Error opening index file." << endl;

return -1; // Return -1 if file can't be opened

}

string line; // To hold each line from index file

while (getline(indexFile, line)) // Read the file line by line

{

istringstream iss(line); // Create string stream for parsing the line

int id, position; // Variables to store parsed values

if (iss >> id >> position) // Extract ID and position

{

if (id == employeeID) // Check if ID matches

{

indexFile.close(); // Close the file

return position; // Return the position of the employee record

}

}

}

indexFile.close(); // Close the file

return -1; // Return -1 if ID not found

}

// Function to create employee records

void employee::create()

{

fstream file("employee.txt", ios::app | ios::binary); // Open data file in append and binary mode

fstream indexfile("index.txt", ios::app); // Open index file in append mode

if (!file || !indexfile) // Check if files opened successfully

{

cout << "Error opening files." << endl;

return;

}

char ch = 'y'; // Initialize user choice to 'y'

do

{

// Input employee details

cout << "Enter employee Id: ";

cin >> records.empid;

cin.ignore(); // Ignore newline left in the input buffer

cout << "Enter employee Name: ";

cin.getline(records.empName, 50); // Read full name including spaces

cout << "Enter employee Designation: ";

cin.getline(records.empDesignation, 50); // Read designation

cout << "Enter employee salary: ";

cin >> records.empSalary;

cin.ignore(); // Ignore leftover newline for next iteration

int position = file.tellp(); // Get current file pointer position (before writing record)

file.write((char\*)&records, sizeof(records)); // Write binary employee record to file

indexfile << records.empid << " " << position << endl; // Write employee ID and position to index file

cout << "Employee added successfully." << endl;

cout << "\nDo you want to add more records (y/n): ";

cin >> ch; // Read user decision

cin.ignore(); // Clear newline before next loop

} while (ch == 'y' || ch == 'Y'); // Continue loop if user wants to add more

file.close(); // Close the data file

indexfile.close(); // Close the index file

}

// Function to display a specific employee's details

void employee::display()

{

int empId; // Variable to store employee ID input

cout << "Enter employee Id: ";

cin >> empId; // Get employee ID from user

if (empId > 0) // Validate employee ID

{

int pos = findEmployeePosition(empId); // Get the file position from index

if (pos < 0) // If position not found

{

cout << "No matching Employee Record available." << endl;

return;

}

else

{

fstream file("employee.txt", ios::in | ios::binary); // Open file in binary read mode

file.seekg(pos); // Move to correct position in file

file.read((char\*)&records, sizeof(records)); // Read the record into 'records'

print(); // Print the record

file.close(); // Close the file

}

}

}

// Function to delete an employee record by removing its entry from index file

void employee::Delete(int employeeId)

{

fstream indexFile("index.txt", ios::in); // Open original index file for reading

if (!indexFile) // Check if opened

{

cout << "Error opening index file." << endl;

return;

}

ofstream tempIndexFile("tempIndex.txt", ios::out); // Create a temporary index file

if (!tempIndexFile) // Check if created successfully

{

cout << "Error creating temporary index file." << endl;

indexFile.close();

return;

}

string line; // To hold each line from index file

int id, position;

bool found = false; // Flag to check if record was found

while (getline(indexFile, line)) // Read original index line-by-line

{

istringstream iss(line); // Use stringstream to parse

if (iss >> id >> position) // Extract employee ID and file position

{

if (id == employeeId) // If record matches ID to delete

{

found = true;

continue; // Skip writing this record

}

}

tempIndexFile << line << endl; // Write all other records to temporary index

}

indexFile.close(); // Close original index file

tempIndexFile.close(); // Close temporary index file

if (!found) // If record not found in index

{

remove("tempIndex.txt"); // Remove temp file

cout << "Employee record not found." << endl;

}

else

{

remove("index.txt"); // Delete old index

rename("tempIndex.txt", "index.txt"); // Rename temp to original

cout << "Employee deleted successfully." << endl;

}

}

// Main function: provides menu-based interface

int main()

{

employee emp; // Create object of class employee

int employeeId; // Variable to store employee ID input

int choice; // Menu choice input

do // Loop for continuous menu

{

// Display menu

cout << "\n1. Add Employee" << endl;

cout << "2. Delete Employee" << endl;

cout << "3. Display Employee" << endl;

cout << "5. Exit" << endl;

cout << "\nEnter your choice: ";

cin >> choice; // Get user choice

switch (choice) // Perform action based on choice

{

case 1:

emp.create(); // Call function to add employee

break;

case 2:

cout << "Enter employee Id to delete: ";

cin >> employeeId; // Get ID to delete

emp.Delete(employeeId); // Call delete function

break;

case 3:

emp.display(); // Call display function

break;

case 5:

return 0; // Exit the program

default:

cout << "Enter a valid choice." << endl;

break;

}

} while (true); // Keep looping until exit

return 0;

}