**Exp No 1:** Consider a student database of SEIT class (at least 15 records). Database contains different fields of every student like Roll No, Name and SGPA.(array of structure)

a) Design a roll call list, arrange list of students according to roll numbers in ascending order . (Use Bubble Sort)

b) Arrange list of students alphabetically. (Use Bubble Sort)

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  // Define the structure for a student record  struct Student {  int rollNo;  string name;  float sgpa;  };  // Function to perform Bubble Sort to arrange students by roll number in ascending order  void bubbleSortByRollNo(Student arr[], int n) {  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (arr[j].rollNo > arr[j + 1].rollNo) {  // Swap the students  Student temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  // Function to perform Bubble Sort to arrange students alphabetically by name  void bubbleSortByName(Student arr[], int n) {  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (arr[j].name > arr[j + 1].name) {  // Swap the students  Student temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  int main() {  const int maxSize = 15;  Student students[maxSize];  // Input student records  cout << "Enter student records (Roll No, Name, SGPA):" << endl;  for (int i = 0; i < maxSize; i++) {  cout << "Student " << i + 1 << ":\n";  cout << "Roll No: ";  cin >> students[i].rollNo;  cout << "Name: ";  cin.ignore();  getline(cin, students[i].name);  cout << "SGPA: ";  cin >> students[i].sgpa;  }  // Sort by roll number in ascending order  bubbleSortByRollNo(students, maxSize);  // Display the roll call list sorted by roll number  cout << "\nRoll Call List (Sorted by Roll No):" << endl;  cout << "Roll No\tName\t\tSGPA" << endl;  for (int i = 0; i < maxSize; i++) {  cout << students[i].rollNo << "\t" << students[i].name << "\t\t" << students[i].sgpa << endl;  }  // Sort alphabetically by name  bubbleSortByName(students, maxSize);  // Display the list sorted by name  cout << "\nList of Students (Sorted Alphabetically by Name):" << endl;  cout << "Roll No\tName\t\tSGPA" << endl;  for (int i = 0; i < maxSize; i++) {  cout << students[i].rollNo << "\t" << students[i].name << "\t\t" << students[i].sgpa << endl;  }  return 0;  } |

**Exp No 2:** Implement stack as an abstract data type using singly linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix and prefix expression.

|  |
| --- |
| #include <iostream>  #include <stack>  #include <string>  using namespace std;  // Define the Node structure for the linked list  struct Node {  char data;  Node\* next;  Node(char data) : data(data), next(nullptr) {}  };  // Define the Stack class  class Stack {  private:  Node\* top;  public:  Stack() : top(nullptr) {}  // Function to push an element onto the stack  void push(char data) {  Node\* newNode = new Node(data);  newNode->next = top;  top = newNode;  }  // Function to pop an element from the stack  char pop() {  if (isEmpty()) {  cerr << "Stack is empty." << endl;  exit(1);  }  char data = top->data;  Node\* temp = top;  top = top->next;  delete temp;  return data;  }  // Function to check if the stack is empty  bool isEmpty() {  return top == nullptr;  }  // Function to return the top element without popping  char peek() {  if (isEmpty()) {  cerr << "Stack is empty." << endl;  exit(1);  }  return top->data;  }  };  // Function to check if a character is an operator  bool isOperator(char c) {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  // Function to get the precedence of an operator  int precedence(char c) {  if (c == '+' || c == '-') return 1;  if (c == '\*' || c == '/') return 2;  return 0; // For '('  }  // Function to convert infix expression to postfix  string infixToPostfix(string infix) {  Stack stack;  string postfix = "";  for (char c : infix) {  if (isalnum(c)) {  postfix += c;  }  else if (c == '(') {  stack.push(c);  }  else if (c == ')') {  while (!stack.isEmpty() && stack.peek() != '(') {  postfix += stack.pop();  }  stack.pop(); // Pop '('  }  else {  while (!stack.isEmpty() && precedence(c) <= precedence(stack.peek())) {  postfix += stack.pop();  }  stack.push(c);  }  }  while (!stack.isEmpty()) {  postfix += stack.pop();  }  return postfix;  }  // Function to convert infix expression to prefix  string infixToPrefix(string infix) {  // Reverse the infix expression and swap '(' with ')' and vice versa  string reversedInfix = "";  for (char c : infix) {  if (c == '(') {  reversedInfix += ')';  }  else if (c == ')') {  reversedInfix += '(';  }  else {  reversedInfix += c;  }  }  reverse(reversedInfix.begin(), reversedInfix.end());  // Convert the reversed infix expression to postfix and reverse it again  string postfix = infixToPostfix(reversedInfix);  reverse(postfix.begin(), postfix.end());  return postfix;  }  // Function to evaluate a postfix expression  int evaluatePostfix(string postfix) {  stack<int> operands;  for (char c : postfix) {  if (isdigit(c)) {  operands.push(c - '0');  }  else if (isOperator(c)) {  int operand2 = operands.top();  operands.pop();  int operand1 = operands.top();  operands.pop();  switch (c) {  case '+':  operands.push(operand1 + operand2);  break;  case '-':  operands.push(operand1 - operand2);  break;  case '\*':  operands.push(operand1 \* operand2);  break;  case '/':  operands.push(operand1 / operand2);  break;  }  }  }  return operands.top();  }  // Function to evaluate a prefix expression  int evaluatePrefix(string prefix) {  stack<int> operands;  for (int i = prefix.size() - 1; i >= 0; i--) {  char c = prefix[i];  if (isdigit(c)) {  operands.push(c - '0');  }  else if (isOperator(c)) {  int operand1 = operands.top();  operands.pop();  int operand2 = operands.top();  operands.pop();  switch (c) {  case '+':  operands.push(operand1 + operand2);  break;  case '-':  operands.push(operand1 - operand2);  break;  case '\*':  operands.push(operand1 \* operand2);  break;  case '/':  operands.push(operand1 / operand2);  break;  }  }  }  return operands.top();  }  int main() {  string infixExpression;  cout << "Enter an infix expression: ";  cin >> infixExpression;  string postfixExpression = infixToPostfix(infixExpression);  string prefixExpression = infixToPrefix(infixExpression);  cout << "Postfix Expression: " << postfixExpression << endl;  cout << "Prefix Expression: " << prefixExpression << endl;  int postfixResult = evaluatePostfix(postfixExpression);  int prefixResult = evaluatePrefix(prefixExpression);  cout << "Postfix Evaluation Result: " << postfixResult << endl;  cout << "Prefix Evaluation Result: " << prefixResult << endl;  return 0;  } |

**Exp 3:** Implement Circular Queue using Array. Perform following operations on it.

a) Insertion (Enqueue)

b) Deletion (Dequeue)

c) Display

|  |
| --- |
| #include <iostream>  using namespace std;  class CircularQueue {  private:  int\* arr;  int front;  int rear;  int maxSize;  int currentSize;  public:  CircularQueue(int size) {  maxSize = size;  arr = new int[maxSize];  front = rear = -1;  currentSize = 0;  }  ~CircularQueue() {  delete[] arr;  }  bool isEmpty() {  return currentSize == 0;  }  bool isFull() {  return currentSize == maxSize;  }  void enqueue(int data) {  if (isFull()) {  cout << "Queue is full. Cannot enqueue." << endl;  return;  }  if (isEmpty()) {  front = rear = 0;  } else {  rear = (rear + 1) % maxSize;  }  arr[rear] = data;  currentSize++;  }  void dequeue() {  if (isEmpty()) {  cout << "Queue is empty. Cannot dequeue." << endl;  return;  }  if (front == rear) {  front = rear = -1;  } else {  front = (front + 1) % maxSize;  }  currentSize--;  }  void display() {  if (isEmpty()) {  cout << "Queue is empty." << endl;  return;  }  int i = front;  cout << "Queue elements: ";  while (true) {  cout << arr[i] << " ";  if (i == rear) {  break;  }  i = (i + 1) % maxSize;  }  cout << endl;  }  };  int main() {  int size;  cout << "Enter the size of the circular queue: ";  cin >> size;  CircularQueue queue(size);  int choice, data;  while (true) {  cout << "\nCircular Queue Operations:" << endl;  cout << "1. Enqueue" << endl;  cout << "2. Dequeue" << endl;  cout << "3. Display" << endl;  cout << "4. Exit" << endl;  cout << "Enter your choice: ";  cin >> choice;  switch (choice) {  case 1:  cout << "Enter data to enqueue: ";  cin >> data;  queue.enqueue(data);  break;  case 2:  queue.dequeue();  break;  case 3:  queue.display();  break;  case 4:  exit(0);  default:  cout << "Invalid choice. Try again." << endl;  }  }  return 0;  } |

**Exp 4:** Construct an Expression Tree from postfix expression. Perform recursive and non- recursive In-order, pre-order and post-order traversals.

Algorithm

|  |
| --- |
| Begin  **Function r()** has a character variable as parameter.        If the characters are + or - or \* or / then           Return will be -1        If the characters are from A to Z then           Return will be 1.        If the characters are from a to z then           Return will be 1.        Else           Return -100.  **Function construct\_expression\_tree()** to construct the expression tree  **Function push()** to push values in the stack  **Function pop()** to pop values from the stack  **Function preOrder()** for pre-order traversal  **Function inOrder()** for in-order traversal  **Function postOrder()** for post-order traversal  End. |

|  |
| --- |
| #include <iostream>  using namespace std;  struct n {     char d;     n \*l;     n \*r;  };  char pf[50];  int top = -1;  n \*a[50];  int r(char inputch) {     if (inputch == '+' || inputch == '-' || inputch == '\*' || inputch== '/')        return (-1);     else if (inputch >= 'A' || inputch <= 'Z')        return (1);     else if (inputch >= 'a' || inputch <= 'z')        return (1);     else        return (-100);  }  void push(n \*tree) {     top++;     a[top] = tree;  }  n \*pop() {     top--;     return (a[top + 1]);  }  void construct\_expression\_tree(char \*suffix) {     char s;     n \*newl, \*p1, \*p2;     int flag;     s = suffix[0];     for (int i = 1; s != 0; i++) {        flag = r(s);        if (flag == 1) {           newl = new n;           newl->d = s;           newl->l = NULL;           newl->r = NULL;           push(newl);        } else {           p1 = pop();           p2 = pop();           newl = new n;           newl->d = s;           newl->l = p2;           newl->r = p1;           push(newl);        }        s = suffix[i];     }  }  void preOrder(n \*tree) {     if (tree != NULL) {        cout << tree->d;        preOrder(tree->l);        preOrder(tree->r);     }  }  void inOrder(n \*tree) {     if (tree != NULL) {        inOrder(tree->l);        cout << tree->d;        inOrder(tree->r);     }  }  void postOrder(n \*tree) {     if (tree != NULL) {        postOrder(tree->l);        postOrder(tree->r);        cout << tree->d;     }  }  int main(int argc, char \*\*argv) {     cout << "Enter Postfix Expression : ";     cin >> pf;     construct\_expression\_tree(pf);     cout << "\nIn-Order Traversal : ";     inOrder(a[0]);     cout << "\nPre-Order Traversal : ";     preOrder(a[0]);     cout << "\nPost-Order Traversal : ";     postOrder(a[0]);     return 0;  } |