**Practical 6**

**Aim:** Implementation of various stack applications such as Infix to postfix conversion, postfix evolution and balancing of parenthesis.

**Description:**

A stack is a data structure that stores items in a last-in, first-out (LIFO) order. This means that the last item added to the stack will be the first item to be removed. Stacks are often used in computer science for tasks such as function calls, undo and redo operations, and parsing. Some common operations performed on stacks include push (adding an item to the stack), pop (removing an item from the stack), and peek (looking at the top item on the stack without removing it).

There are many applications of stacks in computer science, some of the common ones are:

* Function calls: When a program calls a function, the return address and the function's parameters are pushed onto the stack. When the function returns, these values are popped off the stack.
* Parsing: Stacks are often used in parsing algorithms to keep track of the current context of the parser.
* Undo/Redo: Applications that support undo and redo functions use stacks to store the previous states of the document, so that they can be undone and redone.
* Memory management: Stacks are used in memory management to keep track of the memory allocation and deallocation.
* Graph algorithms: Stacks are used in graph traversal algorithms such as depth-first search.
* Expression evaluation: Stacks are used in the evaluation of mathematical expressions.
* Compiler design: Stacks are used in the compilation of programming languages for storing information about variables, function calls and other data.

1. **Infix to postfix and expression evolution**

* Algorithm for Infix to postfix conversion

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,

If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), then push it.

1. ‘^’ operator is right associative and other operators like ‘+’,’-‘,’\*’ and ‘/’ are left-associative. Check especially for a condition when both, operator at the top of the stack and the scanned operator are ‘^’. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack. In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
2. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
3. If the scanned character is an ‘(‘, push it to the stack.
4. If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
5. Repeat steps 2-6 until the infix expression is scanned.
6. Print the output
7. Pop and output from the stack until it is not empty.

* Algorithm for postfix expression evolution

1. Create a stack to store operands (or values).
2. Scan the given expression from left to right and do the following for every scanned element.
   1. If the element is a number, push it into the stack
   2. If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack
3. When the expression is ended, the number in the stack is the final answer

| #include <iostream>  #include <iomanip>  #include <math.h>  using namespace std;  int getPrecedance(char op)  {  int prec = 0;  if (op == '^') prec = 3;  else if (op == '%' || op == '/' || op == '\*')prec = 2;  else if (op == '+' || op == '-')prec = 1;  return prec;  }  bool isOperator(char ch)  {  return (ch == '+' || ch == '-' || ch == '/' || ch == '\*' || ch == '^' || ch == '%');  }  void evaluateExpression(string e, string p[],int top){  double eval[50];  int e\_top = -1, i = 0;  double ans = 0;  string post;  while(i <= top){  post += p[i] + " ";  if(isOperator(p[i].at(0))){  switch(p[i].at(0)){  case '\*': eval[--e\_top] = eval[e\_top-1] \* eval[e\_top]; break;  case '+': eval[--e\_top] = eval[e\_top-1] + eval[e\_top]; break;  case '-': eval[--e\_top] = eval[e\_top-1] - eval[e\_top]; break;  case '^': eval[--e\_top] = pow(eval[e\_top-1], eval[e\_top]); break;  case '%': eval[--e\_top] = (int)eval[e\_top-1] % (int)eval[e\_top]; break;  case '/': if(eval[e\_top] == 0){  cout << "Invalid Expression! : Can't divide by Zero" << endl;  return;  }  eval[--e\_top] = eval[e\_top-1] / eval[e\_top]; break;  }  }else{  eval[++e\_top] = stoi(p[i]);  }  i++;  }  cout << setfill(' ')<<setw(30)<<"expression"<< setfill(' ')<<setw(50)<<"postfix"<< setfill(' ')<<setw(15)<<"Answer"<<endl;  cout << setfill(' ')<<setw(30)<<e<< setfill(' ')<<setw(50)<<post<< setfill(' ')<<setw(15)<<eval[e\_top]<<endl;  }  void infix\_to\_postfix(string e)  {  char stack[50];  string p\_stack[100], num;  int top = -1, p\_top = -1;  for (int i = 0; i < e.size(); i++)  {  char ch = e.at(i);  // if character is digit  if (isdigit(ch))  {  num.push\_back(ch);  }  else  {  if(num != ""){  p\_stack[++p\_top] = num;  num = "";  }  // if opening bracket is found  if (ch == '(')  {  if(isdigit(e.at(i-1)))  stack[++top] = '\*';  stack[++top] = '(';  }  // if closing bracket founds  else if (ch == ')')  {  // pop until stack is empty or opening bracket is does not found  while (stack[top] != '(')  p\_stack[++p\_top] = stack[top--];  // remove last opening bracket  if (top != -1)top--;  }  // if character is operator  else if (isOperator(ch))  {  if(isOperator(e.at(i-1))){  cout << "invalid Expression "<<e<<endl;  return;  }  // push only when current operator precedence is greater than precedence of top  // of stack operator or stack is empty  if (top == -1 || getPrecedance(ch) > getPrecedance(stack[top]))  {  stack[++top] = ch;  }  else  { // pop element from stack and append to postfix until you did not found the greater precedence of  // current operator or end of stack  while (top != -1 && getPrecedance(ch) <= getPrecedance(stack[top]))  p\_stack[++p\_top] = stack[top--];  // now push current element into stack  stack[++top] = ch;  }  }  }  }  // if number is present at the end  if(num != "")  p\_stack[++p\_top] = num;  // if some operators are still in stack  while (top != -1)  p\_stack[++p\_top] = stack[top--];  evaluateExpression(e, p\_stack, p\_top);  }  void util(){  string expression;  cout<<setfill('\*')<<setw(30)<<" CALCULATOR " <<setfill('\*')<<setw(30) <<""<< endl;  while(true){  cout<<"Enter Expression: ";  cin >> expression;  infix\_to\_postfix(expression);  }  }  int main(int argc, char const \*argv[])  {  util();  return 0;  } |
| --- |
| **Output-** |

1. **Parentheses Balancing**

Parenthesis balancing is a common problem that can be solved using a stack data structure. The goal of the algorithm is to determine whether a given string of parentheses is properly balanced.

Here's one way to implement the algorithm using a stack:

* Initialize an empty stack.
* Iterate through each character in the input string.
* If the character is an opening parenthesis (e.g. '(' or '{' or '['), push it onto the stack.
* If the character is a closing parenthesis (e.g. ')' or '}' or ']'), check if the stack is empty. If it is, the parentheses are not balanced, so return false.
* If the stack is not empty, check if the most recent opening parenthesis on the stack matches the closing parenthesis. If it does not, the parentheses are not balanced, so return false.
* If the character is a closing parenthesis and matches the most recent opening parenthesis, pop the opening parenthesis off the stack.
* Repeat steps 3-6 for each character in the input string.
* If the stack is empty at the end of the iteration, the parentheses are balanced, so return true. Otherwise, return false.

This algorithm checks for the balance of the parenthesis by checking the opening and closing Parenthesis by comparing the last open parenthesis with the current closing Parenthesis and if it's matching it will pop it out otherwise it will return false.

| #include <iostream>  #include <iomanip>  using namespace std;  bool isValid(string e){  char stack[50];  int top = -1;  for(int i = 0; i < e.size(); i++){  char c = e.at(i);  if(c == '(' || c == '[' || c == '{')stack[++top] = c;  else if(c == ')' && stack[top] != '(')return false;  else if(c == ']' && stack[top] != '[')return false;  else if(c == '}' && stack[top] != '{')return false;  else --top;  }  return top == -1;  }  void util(){  string expression;  cout<<setfill('\*')<<setw(30)<<" PARENTHESIS BALANCING " <<setfill('\*')<<setw(4) <<""<< endl;  while(true){  cout<<"Enter Expression: ";  cin >> expression;  cout<<setfill(' ')<<setw(20)<<"expression" <<setfill(' ')<<setw(20) <<"result"<< endl;  cout<< setfill(' ')<<setw(20) << expression  << setfill(' ')<<setw(20)  << (isValid(expression)?"valid string":"invalid string") << endl;  }  }  int main(int argc, char const \*argv[])  {  util();  return 0;  } |
| --- |
| **Output:** |

**Aim:** Stack Implementation has done successfully using application such as postfix evolution and parenthesis balancing