PW SKILLS

Lesson Plan

Stacks - 3

PW SKILLS

Today's checklist:

- Infix, prefix, and postfix expressions
- Questions to practice

Infix,Prefix,Postfix

Infix expressions are mathematical expressions where operators are placed between operands, and infix evaluation involves calculating the result of such expressions while considering operator precedence and parentheses.

Let's break down infix evaluation with and without brackets:

Infix Evaluation (Without Brackets):

For infix evaluation without brackets, you need to consider operator precedence to correctly evaluate the expression. You can use the algorithm known as the shunting-yard algorithm to convert infix expressions to postfix (or Reverse Polish Notation, RPN) and then evaluate the postfix expression.

Here's an example of infix evaluation without brackets using the shunting-yard algorithm:

Expression: `3 + 4 * 2 / (1 - 5)^2`

1. Convert infix to postfix: `3 4 2 * 1 5 - 2 ^ / +`
2. Evaluate the postfix expression using a stack-based algorithm or by iterating through the postfix expression.

Infix Evaluation (With Brackets): When dealing with infix expressions that include brackets, you need to respect the precedence of operations within brackets and evaluate them first.

Expression: `(3 + 4) * 2 / (1 - 5)^2`

1. Start from left to right and identify the innermost brackets.
2. Evaluate expressions within the innermost brackets first.
3. Replace the evaluated expressions with their results in the original expression.
4. Repeat the process until there are no more brackets.

Steps:

- `(3 + 4) * 2 / (1 - 5)^2`
- `7 * 2 / (1 - 5)^2`
- `14 / (1 - 5)^2`
- `14 / 16`
- **Result: `0.875`**

Evaluating Expressions with Brackets Recursively:

The process involves recursively evaluating subexpressions within brackets until there are no more brackets in the expression.

This evaluation process maintains proper operator precedence by prioritizing operations within brackets, gradually simplifying the expression until the final result is obtained.

Prefix Notation (Polish Notation):

In prefix notation, the operator precedes its operands. For example, the infix expression (5 + 3) * 2 in prefix notation becomes * + 5 3 2.

Postfix Notation (Reverse Polish Notation):

In postfix notation, the operator follows its operands. Using the same infix expression (5 + 3) * 2, the postfix notation is 5 3 + 2 *.

Q. Given an infix expression, the task is to convert it to a prefix expression.

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

Input: (A – B/C) * (A/K-L)
Output: *-A/BC-/AKL

Explanation: To convert an infix expression to a prefix expression, we can use the stack data structure. The idea is as follows:

Step 1: Reverse the infix expression. Note while reversing each '(' will become ')' and each ')' becomes '('.

Step 2: Convert the reversed infix expression to "nearly" postfix expression.
While converting to postfix expression, instead of using pop operation to pop operators with greater than or equal precedence, here we will only pop the operators from stack that have greater precedence.

Step 3: Reverse the postfix expression.
The stack is used to convert infix expression to postfix form.

Code:

```
// C++ program to convert infix to prefix
#include <bits/stdc++.h>
using namespace std;

// Function to check if the character is an operator
bool isOperator(char c)
{
 return (!isalpha(c) && !isdigit(c));
}

// Function to get the priority of operators
int getPriority(char C)
{
 if (C == '-' || C == '+')
  return 1;
 else if (C == '*' || C == '/')
  return 2;
 else if (C == '^')
  return 3;
 return 0;
}
```

```
// Function to convert the infix expression to postfix
string infixToPostfix(string infix)
{
 infix = '(' + infix + ')';
 int l = infix.size();
 stack<char> char_stack;
 string output;

 for (int i = 0; i < l; i++) {

  // If the scanned character is an
  // operand, add it to output.
  if (isalpha(infix[i]) || isdigit(infix[i]))
   output += infix[i];

  // If the scanned character is an
  // '(', push it to the stack.
  else if (infix[i] == '(')
   char_stack.push('(');

  // If the scanned character is an
  // ')', pop and output from the stack
  // until an '(' is encountered.
  else if (infix[i] == ')') {
   while (char_stack.top() != '(') {
    output += char_stack.top();
    char_stack.pop();
   }

   // Remove '(' from the stack
   char_stack.pop();
  }

  // Operator found
  else {
   if (isOperator(char_stack.top())) {
    if (infix[i] == '^') {
     while (
      getPriority(infix[i])
       <= getPriority(char_stack.top())) {
      output += char_stack.top();
      char_stack.pop();
     }
    }
    else {
     while (
      getPriority(infix[i])
      < getPriority(char_stack.top())) {
      output += char_stack.top();
      char_stack.pop();
     }
    }
```

```
    // Push current Operator on stack
    char_stack.push(infix[i]);
   }
  }
 }
 while (!char_stack.empty()) {
  output += char_stack.top();
  char_stack.pop();
 }
 return output;
}

// Function to convert infix to prefix notation
string infixToPrefix(string infix)
{
 // Reverse String and replace ( with ) and vice versa
 // Get Postfix
 // Reverse Postfix
 int l = infix.size();

 // Reverse infix
 reverse(infix.begin(), infix.end());

 // Replace ( with ) and vice versa
 for (int i = 0; i < l; i++) {

  if (infix[i] == '(') {
   infix[i] = ')';
  }
  else if (infix[i] == ')') {
   infix[i] = '(';
  }
 }

 string prefix = infixToPostfix(infix);

 // Reverse postfix
 reverse(prefix.begin(), prefix.end());

 return prefix;
}

// Driver code
int main()
{
 string s = ("x+y*z/w+u");

 // Function call
 cout << infixToPrefix(s) << std::endl;
 return 0;
}
```

Q. Write a program to convert an Infix expression to Postfix form.

Input: A + B * C + D
Output: ABC*+D+

Input: ((A + B) – C * (D / E)) + F
Output: AB+CDE/*-F+

Idea: To convert infix expression to postfix expression, use the stack data structure. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.

Code:

```
// C++ code to convert infix expression to postfix

#include <bits/stdc++.h>
using namespace std;

// Function to return precedence of operators
int prec(char c)
{
 if (c == '^')
  return 3;
 else if (c == '/' || c == '*')
  return 2;
 else if (c == '+' || c == '-')
  return 1;
 else
  return -1;
}

// The main function to convert infix expression
// to postfix expression
void infixToPostfix(string s)
{

 stack<char> st;
 string result;

 for (int i = 0; i < s.length(); i++) {
  char c = s[i];

  // If the scanned character is
  // an operand, add it to output string.
  if ((c ≥ 'a' && c ≤ 'z') || (c ≥ 'A' && c ≤ 'Z')
   || (c ≥ '0' && c ≤ '9'))
   result += c;

  // If the scanned character is an
```

```
  // '(', push it to the stack.
  else if (c == '(')
   st.push('(');

  // If the scanned character is an ')',
  // pop and add to output string from the stack
  // until an '(' is encountered.
  else if (c == ')') {
   while (st.top() != '(') {
    result += st.top();
    st.pop();
   }
   st.pop();
  }

  // If an operator is scanned
  else {
   while (!st.empty()
    && prec(s[i]) <= prec(st.top())) {
    result += st.top();
    st.pop();
   }
   st.push(c);
  }
 }

 // Pop all the remaining elements from the stack
 while (!st.empty()) {
  result += st.top();
  st.pop();
 }

 cout << result << endl;
}

// Driver code
int main()
{
 string exp = "a+b*(c^d-e)^(f+g*h)-i";

 // Function call
 infixToPostfix(exp);

 return 0;
}
```

Q. Given a postfix expression, the task is to evaluate the postfix expression.

Input: str = "2 3 1 * + 9 -"
Output: -4

Explanation: If the expression is converted into an infix expression, it will be 2 + (3 * 1) – 9 = 5 – 9 = -4.

Input: str = "100 200 + 2 / 5 * 7 +"
Output: 757

Idea: Iterate the expression from left to right and keep on storing the operands into a stack. Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

Code:

```
int evaluatePostfix(string exp)
{
    // Create a stack of capacity equal to expression size
    stack<int> st;

    // Scan all characters one by one
    for (int i = 0; i < exp.size(); ++i) {

        // If the scanned character is an operand
        // (number here), push it to the stack.
        if (isdigit(exp[i]))
            st.push(exp[i] - '0');

        // If the scanned character is an operator,
        // pop two elements from stack apply the operator
        else {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            switch (exp[i]) {
            case '+':
                st.push(val2 + val1);
                break;
            case '-':
                st.push(val2 - val1);
                break;
            case '*':
                st.push(val2 * val1);
                break;
            case '/':
                st.push(val2 / val1);
                break;
            }
        }
    }
    return st.top();
}
```

Q. Prefix Evaluation

Input : -+8/632
Output : 8

Input : -+7*45+20
Output : 25

Algorithm: EVALUATE_PREFIX(STRING)

Step 1: Put a pointer P at the end of the end

Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack, return it.

Step 6: End

Code:

```
double evaluatePrefix(string exprsn)
{
    stack<double> Stack;

    for (int j = exprsn.size() - 1; j ⩾ 0; j--) {

        // if jth character is the delimiter ( which is
        // space in this case) then skip it
        if (exprsn[j] == ' ')
            continue;

        // Push operand to Stack
        // To convert exprsn[j] to digit subtract
        // '0' from exprsn[j].
        if (isdigit(exprsn[j])) {

            // there may be more than
            // one digits in a number
            double num = 0, i = j;
            while (j < exprsn.size() && isdigit(exprsn[j]))
                j--;
            j++;

            // from [j, i] exprsn contains a number
            for (int k = j; k ⩽ i; k++)
                num = num * 10 + double(exprsn[k] - '0');

            Stack.push(num);
        }
        else {
```

```
            // Operator encountered
            // Pop two elements from Stack
            double o1 = Stack.top();
            Stack.pop();
            double o2 = Stack.top();
            Stack.pop();

            // Use switch case to operate on o1
            // and o2 and perform o1 O o2.
            switch (exprsn[j]) {
            case '+':
                Stack.push(o1 + o2);
                break;
            case '-':
                Stack.push(o1 - o2);
                break;
            case '*':
                Stack.push(o1 * o2);
                break;
            case '/':
                Stack.push(o1 / o2);
                break;
            }
        }
    }

    return Stack.top();
}
```

Q. Given a Prefix expression, convert it into a Infix expression.

Input : Prefix : *+AB-CD
Output : Infix : ((A+B)*(C-D))

Input : Prefix : *-A/BC-/AKL
Output : Infix : ((A-(B/C))*((A/K)-L))

Algorithm for Prefix to Infix:

Read the Prefix expression in reverse order (from right to left)
If the symbol is an operand, then push it onto the Stack
If the symbol is an operator, then pop two operands from the Stack
Create a string by concatenating the two operands and the operator between them.
string = (operand1 + operator + operand2)
And push the resultant string back to Stack
Repeat the above steps until the end of Prefix expression.
At the end stack will have only 1 string i.e resultant string

Code:

```
// C++ Program to convert prefix to Infix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x) {
switch (x) {
case '+':
case '-':
case '/':
case '*':
case '^':
case '%':
 return true;
}
return false;
}

// Convert prefix to Infix expression
string preToInfix(string pre_exp) {
stack<string> s;

// length of expression
int length = pre_exp.size();

// reading from right to left
for (int i = length - 1; i ⩾ 0; i--) {

 // check if symbol is operator
 if (isOperator(pre_exp[i])) {

 // pop two operands from stack
 string op1 = s.top(); s.pop();
 string op2 = s.top(); s.pop();

 // concat the operands and operator
 string temp = "(" + op1 + pre_exp[i] + op2 + ")";

 // Push string temp back to stack
 s.push(temp);
 }

 // if symbol is an operand
 else {

 // push the operand to the stack
 s.push(string(1, pre_exp[i]));
 }
}
```

```
// Stack now contains the Infix expression
return s.top();
}

// Driver Code
int main() {
string pre_exp = "*-A/BC-/AKL";
cout << "Infix : " << preToInfix(pre_exp);
return 0;
}
```

Q. Given a Prefix expression, convert it into a Postfix expression.

Input : Prefix : *+AB-CD
Output : Postfix : AB+CD-*

Explanation : Prefix to Infix : (A+B) * (C-D)
Infix to Postfix : AB+CD-*

Input : Prefix : *-A/BC-/AKL
Output : Postfix : ABC/-AK/L-*

Explanation : Prefix to Infix : (A-(B/C))*((A/K)-L)
Infix to Postfix : ABC/-AK/L-*

Algorithm: Read the Prefix expression in reverse order (from right to left)
If the symbol is an operand, then push it onto the Stack
If the symbol is an operator, then pop two operands from the Stack
Create a string by concatenating the two operands and the operator after them.
string = operand1 + operand2 + operator
And push the resultant string back to Stack
Repeat the above steps until end of Prefix expression.

Code:

```
// C++ Program to convert prefix to Infix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x) {
switch (x) {
case '+':
case '-':
case '/':
case '*':
case '^':
```

```
case '%':
 return true;
}
return false;
}

// Convert prefix to Infix expression
string preToInfix(string pre_exp) {
stack<string> s;

// length of expression
int length = pre_exp.size();

// reading from right to left
for (int i = length - 1; i ≥ 0; i--) {

 // check if symbol is operator
 if (isOperator(pre_exp[i])) {

 // pop two operands from stack
 string op1 = s.top(); s.pop();
 string op2 = s.top(); s.pop();

 // concat the operands and operator
 string temp = "(" + op1 + pre_exp[i] + op2 + ")";

 // Push string temp back to stack
 s.push(temp);
 }

 // if symbol is an operand
 else {

 // push the operand to the stack
 s.push(string(1, pre_exp[i]));
 }
}

// Stack now contains the Infix expression
return s.top();
}

// Driver Code
int main() {
string pre_exp = "*-A/BC-/AKL";
cout << "Infix : " << preToInfix(pre_exp);
return 0;
}
```

Q. Given a Prefix expression, convert it into a Postfix expression.

Input : Prefix : *+AB-CD
Output : Postfix : AB+CD-*

Explanation : Prefix to Infix : (A+B) * (C-D)
Infix to Postfix : AB+CD-*

Input : Prefix : *-A/BC-/AKL
Output : Postfix : ABC/-AK/L-*

Explanation : Prefix to Infix : (A-(B/C))*((A/K)-L)
Infix to Postfix : ABC/-AK/L-*

Algorithm: Read the Prefix expression in reverse order (from right to left)
If the symbol is an operand, then push it onto the Stack
If the symbol is an operator, then pop two operands from the Stack
Create a string by concatenating the two operands and the operator after them.
string = operand1 + operand2 + operator
And push the resultant string back to Stack
Repeat the above steps until end of Prefix expression.

Code:

```
// CPP Program to convert prefix to postfix
#include <iostream>
#include <stack>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x)
{
 switch (x) {
 case '+':
 case '-':
 case '/':
 case '*':
  return true;
 }
 return false;
}

// Convert prefix to Postfix expression
string preToPost(string pre_exp)
{

 stack<string> s;
 // length of expression
 int length = pre_exp.size();

 // reading from right to left
 for (int i = length - 1; i ⩾ 0; i--)
 {
```

```
  // check if symbol is operator
  if (isOperator(pre_exp[i]))
  {
   // pop two operands from stack
   string op1 = s.top();
   s.pop();
   string op2 = s.top();
   s.pop();

   // concat the operands and operator
   string temp = op1 + op2 + pre_exp[i];

   // Push string temp back to stack
   s.push(temp);
  }

  // if symbol is an operand
  else {

   // push the operand to the stack
   s.push(string(1, pre_exp[i]));
  }
 }

 // stack contains only the Postfix expression
 return s.top();
}

// Driver Code
int main()
{
 string pre_exp = "*-A/BC-/AKL";
 cout << "Postfix : " << preToPost(pre_exp);
 return 0;
}
```

Q. Postfix to Infix

Input : abc++
Output : (a + (b + c))

Input : ab*c+
Output : ((a*b)+c)

Algorithm:

1. While there are input symbol left
...1.1 Read the next symbol from the input.

2. If the symbol is an operand
...2.1 Push it onto the stack.

3. Otherwise,
...3.1 the symbol is an operator.
...3.2 Pop the top 2 values from the stack.
...3.3 Put the operator, with the values as arguments and form a string.
...3.4 Push the resulted string back to stack.

4. If there is only one value in the stack
...4.1 That value in the stack is the desired infix string.

Below is the implementation of above approach:

Code:

```
// CPP program to find infix for
// a given postfix.
#include <bits/stdc++.h>
using namespace std;

bool isOperand(char x)
{
return (x >= 'a' && x <= 'z') ||
  (x >= 'A' && x <= 'Z');
}

// Get Infix for a given postfix
// expression
string getInfix(string exp)
{
 stack<string> s;

 for (int i=0; exp[i]!='\0'; i++)
 {
  // Push operands
  if (isOperand(exp[i]))
  {
  string op(1, exp[i]);
  s.push(op);
  }

  // We assume that input is
  // a valid postfix and expect
  // an operator.
  else
  {
   string op1 = s.top();
   s.pop();
   string op2 = s.top();
   s.pop();
```

```
    s.push("(" + op2 + exp[i] +
     op1 + ")");
   }
  }

  // There must be a single element
  // in stack now which is the required
  // infix.
  return s.top();
}

// Driver code
int main()
{
  string exp = "ab*c+";
  cout << getInfix(exp);
  return 0;
}
```

Q. Postfix to Prefix Conversion

Input : Postfix : AB+CD-*
Output : Prefix : *+AB-CD

Explanation : Postfix to Infix : (A+B) * (C-D)
Infix to Prefix : *+AB-CD

Input : Postfix : ABC/-AK/L-*
Output : Prefix : *-A/BC-/AKL

Explanation : Postfix to Infix : ((A-(B/C))*((A/K)-L))
Infix to Prefix : *-A/BC-/AKL

Algorithm for Postfix to Prefix:

Read the Postfix expression from left to right
If the symbol is an operand, then push it onto the Stack
If the symbol is an operator, then pop two operands from the Stack
Create a string by concatenating the two operands and the operator before them.
string = operator + operand2 + operand1
And push the resultant string back to Stack
Repeat the above steps until end of Postfix expression.

Code:

```
// CPP Program to convert postfix to prefix
#include <bits/stdc++.h>
using namespace std;

// function to check if character is operator or not
bool isOperator(char x)
{
```

```
 switch (x) {
 case '+':
 case '-':
 case '/':
 case '*':
  return true;
 }
 return false;
}

// Convert postfix to Prefix expression
string postToPre(string post_exp)
{
 stack<string> s;

 // length of expression
 int length = post_exp.size();

 // reading from left to right
 for (int i = 0; i < length; i++) {

  // check if symbol is operator
  if (isOperator(post_exp[i])) {

   // pop two operands from stack
   string op1 = s.top();
   s.pop();
   string op2 = s.top();
   s.pop();

   // concat the operands and operator
   string temp = post_exp[i] + op2 + op1;

   // Push string temp back to stack
   s.push(temp);
  }

  // if symbol is an operand
  else {

   // push the operand to the stack
   s.push(string(1, post_exp[i]));
  }
 }

 string ans = "";
 while (!s.empty()) {
  ans += s.top();
  s.pop();
 }
 return ans;
}
```

```
// Driver Code
int main()
{
 string post_exp = "ABC/-AK/L-*";

 // Function call
 cout << "Prefix : " << postToPre(post_exp);
 return 0;
}
```

PW SKILLS

THANK YOU !

PW SKILLS