Bracket check using stack

#include<bits/stdc++.h>

using namespace std;

// function to check if paranthesis are balanced

bool areParanthesisBalanced(string expr)

{

stack<char> s;

char x;

// Traversing the Expression

for (int i=0; i<expr.length(); i++)

{

if (expr[i]=='('||expr[i]=='['||expr[i]=='{')

{

// Push the element in the stack

s.push(expr[i]);

continue;

}

// IF current current character is not opening

// bracket, then it must be closing. So stack

// cannot be empty at this point.

if (s.empty())

return false;

switch (expr[i])

{

case ')':

// Store the top element in a

x = s.top();

s.pop();

if (x=='{' || x=='[')

return false;

break;

case '}':

// Store the top element in b

x = s.top();

s.pop();

if (x=='(' || x=='[')

return false;

break;

case ']':

// Store the top element in c

x = s.top();

s.pop();

if (x =='(' || x == '{')

return false;

break;

}

}

// Check Empty Stack

return (s.empty());

}

// Driver program to test above function

int main()

{

string expr = "{()[]}";

if (areParanthesisBalanced(expr))

cout << "Balanced";

else

cout << "Not Balanced";

return 0;

}

Implement stack using queue

#include<bits/stdc++.h>

#include<iostream>

using namespace std;

template<class T>

class Stack{

queue<int> q;

public:

void push(T d)

{

int s=q.size();

q.push(d);

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

q.push(q.front());

q.pop();

}

}

T pop()

{

cout<<endl;

if(q.empty()){

cout<<"empty\n";

}

else

q.pop();

}

T top()

{

return (q.empty())? -1 : q.front();

}

void display(){

if(!q.empty()){

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

cout<<q.front()<<" ";

q.pop();

}

}

}

};

main(){

Stack<int> s;

s.push(5);

s.push(12);

s.push(21);

cout<<s.top();

s.pop();

cout<<s.top();

}

Binary search + shell sort

#include<iostream>

using namespace std;

void sort(int \*a,int n)

{

for(int g=n/2;g>0;g/=2){

int j=0;

for(int i=g;i<n;i++){

int t=a[i];

for(j=i; j>=g && a[j-g]>t ; j-=g)

{a[j]=a[j-g];}

a[j]=t;

}

}

}

void binsearch(int \*a,int n,int x)

{

int l=0,u=n-1,m,flag=0;

while(l<=u)

{

m=(l+u)/2;

if(x==a[m]){

cout<<"\nFound!";

flag=1;

break;

}

else if(x>a[m])l=m+1;

else u=m-1;

}

if(flag==0)cout<<"\nNot found!";

}

main(){

int \*a,s,item;

cout<<"Enter size : ";

cin>>s;

a=new int[s];

cout<<"enter the elements of array :\n";

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

cin>>a[i];

}

sort(a,s);

cout<<"\nEnter element to be searched : ";

cin>>item;

binsearch(a,s,item);

}

Recursive Sort

#include<iostream>

using namespace std;

int getMax(int \*a,int n){

int m=a[0],mI=0;

for(int i=1;i<n;i++){

if(m<a[i])

{

m=a[i];

mI=i;

}

}

return mI;

}

sort(int \*a,int n){

int max,t;

if(n>1){

max=getMax(a,n);

t=a[n-1];

a[n-1]=a[max];

a[max]=t;

sort(a,n-1);

}

}

main(){

int \*a =new int[20];

int n,i,j;

cout<<"Enter no of elements :";cin>>n;

cout<<"\nEnter the elements of array : \n";

for(i=0;i<n;i++)cin>>\*(a+i);

sort (a,n);

for(i=0;i<n;i++)cout<<\*(a+i)<<" ";

delete a;

}

Infix to postfix

Stack | Set 2 (Infix to Postfix)

**Infix expression:**The expression of the form a op b. When an operator is in-between every pair of operands.

**Postfix expression:**The expression of the form a b op. When an operator is followed for every pair of operands.

**Why postfix representation of the expression?**  
The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression b \* c, then again scan the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is: abc\*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

**Algorithm**

**1.** Scan the infix expression from left to right.

**2.** If the scanned character is an operand, output it.

**3.**Else,

…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a ‘(‘ ), push it.

…..**3.2** 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.)

**4.** If the scanned character is an ‘(‘, push it to the stack.

**5.** If the scanned character is an ‘)’, pop the stack and and output it until a ‘(‘ is encountered, and discard both the parenthesis.

**6.** Repeat steps 2-6 until infix expression is scanned.

**7.** Print the output

**8.**Pop and output from the stack until it is not empty.

// C program to convert infix expression to postfix

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

// Stack type

struct Stack

{

    int top;

    unsigned capacity;

    int\* array;

};

// Stack Operations

struct Stack\* createStack( unsigned capacity )

{

    struct Stack\* stack = (struct Stack\*) malloc(sizeof(struct Stack));

    if (!stack)

        return NULL;

    stack->top = -1;

    stack->capacity = capacity;

    stack->array = (int\*) malloc(stack->capacity \* sizeof(int));

    if (!stack->array)

        return NULL;

    return stack;

}

int isEmpty(struct Stack\* stack)

{

    return stack->top == -1 ;

}

char peek(struct Stack\* stack)

{

    return stack->array[stack->top];

}

char pop(struct Stack\* stack)

{

    if (!isEmpty(stack))

        return stack->array[stack->top--] ;

    return '$';

}

void push(struct Stack\* stack, char op)

{

    stack->array[++stack->top] = op;

}

// A utility function to check if the given character is operand

int isOperand(char ch)

{

    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// A utility function to return precedence of a given operator

// Higher returned value means higher precedence

int Prec(char ch)

{

    switch (ch)

    {

    case '+':

    case '-':

        return 1;

    case '\*':

    case '/':

        return 2;

    case '^':

        return 3;

    }

    return -1;

}

**// The main function that converts given infix expression**

**// to postfix expression.**

**int infixToPostfix(char\* exp)**

**{**

**int i, k;**

**// Create a stack of capacity equal to expression size**

**struct Stack\* stack = createStack(strlen(exp));**

**if(!stack) // See if stack was created successfully**

**return -1 ;**

**for (i = 0, k = -1; exp[i]; ++i)**

**{**

**// If the scanned character is an operand, add it to output.**

**if (isOperand(exp[i]))**

**exp[++k] = exp[i];**

**// If the scanned character is an ‘(‘, push it to the stack.**

**else if (exp[i] == '(')**

**push(stack, exp[i]);**

**// If the scanned character is an ‘)’, pop and output from the stack**

**// until an ‘(‘ is encountered.**

**else if (exp[i] == ')')**

**{**

**while (!isEmpty(stack) && peek(stack) != '(')**

**exp[++k] = pop(stack);**

**if (!isEmpty(stack) && peek(stack) != '(')**

**return -1; // invalid expression**

**else**

**pop(stack);**

**}**

**else // an operator is encountered**

**{**

**while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))**

**exp[++k] = pop(stack);**

**push(stack, exp[i]);**

**}**

**}**

**// pop all the operators from the stack**

**while (!isEmpty(stack))**

**exp[++k] = pop(stack );**

**exp[++k] = '\0';**

**printf( "%sn", exp );**

**}**

// Driver program to test above functions

int main()

{

    char exp[] = "a+b\*(c^d-e)^(f+g\*h)-i";

    infixToPostfix(exp);

    return 0;

}