LAB 1

**Stack**

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted from one end called the top of the stack.

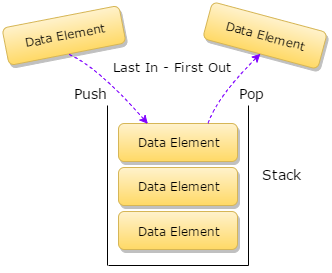


Fig: Stack

**Operations on stack**

There are four major operations that can be performed using a stack. They are as follows:

1. Push: It is the term used to add an item to the stack
2. Pop: It is the term used to delete an item from the stack
3. Peek: It is the term used to get the top item of the stack without removing
4. Display: It is the term used to display all elements of the stack

**Overflow and Underflow conditions**

Overflow condition in stack arises when we try to insert an item to stack beyond the limit of the stack. Underflow condition arises when we try to remove an item although the stack is empty.

**Algorithm to perform operations on stack**

In this lab we implement stack using an array which can hold an item of same data type.

* **Push operation**

PUSH (Stack, MAXSIZE, TOP, item)

Initial condition: TOP:=-1

1. If TOP:=MAXSIZE-1 then print OVERFLOW and exit
2. Set TOP:=TOP+1
3. Set Stack[TOP]:=item
4. Return

* **Pop operation**

POP (Stack, item)

1. If TOP:=-1 then print UNDERFLOW and exit
2. Set item:=Stack[TOP]
3. TOP:=TOP-1
4. Return



Fig: Push and Pop operation in stack

**Application of Stack**

Stacks are widely used to evaluate postfix, prefix, infix expressions. Also, stacks are used in recursive function. In our lab we implement stack to convert an

infix expression to a postfix expression and evaluate the postfix operation. We also used stack for matching the nested parentheses.

* **Algorithm to convert infix expression to postfix expression**

1. Start
2. Scan the infix string from left to right
3. Initialize the empty stack
4. If the scanned character is operand then add it to the postfix string
5. If a left parenthesis is encountered push it into the stack
6. If the scanned character is an operator and stack is not empty then continue
   1. If the scanned operator has the precedence greater than the precedence of the element at top of the stack then push it to stack
   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 this push the scanned character to stack.
7. If a right parenthesis is encountered then
   1. Repeatedly pop the element from stack and add to postfix string until a left parenthesis encountered.
   2. Remove the left parenthesis of stack without adding it to postfix string
8. Repeat 4-6 until null character is encountered.
9. Exit

* **Algorithm to evaluate postfix operation**

1. Start
2. Add the right parenthesis “)” at the end of the postfix string p
3. Scan the postfix string from left to right and repeat 4 to 5 until the stack is empty
4. If an operand is encountered push it into the stack
5. If an operator is encountered (say @) then
   1. pop two elements from the stack, where A is the element at top of stack and B is the element next to top element
   2. Evaluate B@A as result
   3. Push result to the stack
6. Set value equal to the top element of the stack
7. Exit

* **Algorithm to convert infix operation to prefix operation**

1. Start
2. Reverse the given infix operation and also replace right parenthesis with left parenthesis and vice versa if exists (say R)
3. Convert R into postfix expression (say P)
4. Reverse P which gives prefix expression
5. Stop

* **Algorithm to match the nested parentheses**

1. Start
2. Scan the expression from left to right and repeat 3 and 4 until stack is empty
3. If you encounter a left parenthesis then push it into the stack and continue scanning
4. If you encounter a right parenthesis then
   1. If stack is empty then display expression is not correct and exit
   2. Else pop the element from the stack
5. If null is encountered then
   1. If stack is empty then display expression is correct and exit
   2. Else display expression is incorrect and exit

**C++ implementation of a stack**

#include<iostream>

const int MAX=15;

//Error that can occur in stack

class STACK\_ERR

{

private:

std::string exception;

public:

STACK\_ERR()

{

exception='\0';

}

STACK\_ERR(std::string exception)

{

this->exception=exception;

}

std::string get\_exception()

{

return exception;

}

};

class STACK\_OVERFLOW:public STACK\_ERR

{

public:

STACK\_OVERFLOW():STACK\_ERR("Stack overflow"){}

};

class STACK\_UNDERFLOW:public STACK\_ERR

{

public:

STACK\_UNDERFLOW():STACK\_ERR("Stack underflow"){}

};

//Creating stack array implementation

template<class T>

class Stack

{

private:

T arr[MAX];

int top;

public:

Stack();

void push(T data);

T pop();

T peek();

void display();

bool is\_empty();

bool is\_full();

};

template<class T>

Stack<T>::Stack()

{

top=-1;

}

template<class T>

void Stack<T>::push(T data)

{

if(is\_full())

{

throw STACK\_OVERFLOW();

}

else

{

top++;

arr[top]=data;

}

}

template<class T>

T Stack<T>::pop()

{

if(is\_empty())

{

throw STACK\_UNDERFLOW();

}

else

{

T temp;

temp=arr[top];

top--;

return temp;

}

}

template<class T>

T Stack<T>::peek()

{

if(is\_empty())

{

throw STACK\_UNDERFLOW();

}

return arr[top];

}

template<class T>

void Stack<T>::display()

{

if(is\_empty())

{

std::cout<<"Stack is empty"<<std::endl;

}

for(int i=0;i<=top;i++)

{

std::cout<<arr[i]<<" ";

}

std::cout<<std::endl;

}

template<class T>

bool Stack<T>::is\_empty()

{

return top==-1?true:false;

}

template<class T>

bool Stack<T>::is\_full()

{

return top==MAX-1?true:false;

}

int main()

{

Stack<int> s;

try

{

s.display();

s.push(1);

s.push(2);

s.push(3);

s.display();

s.push(4);

s.display();

s.peek();

s.pop();

s.display();

s.pop();

s.pop();

s.display();

s.pop();

s.display();

s.pop();

}

catch(STACK\_UNDERFLOW s)

{

std::cout<<"Error:"<<s.get\_exception()<<std::endl;

}

catch(STACK\_OVERFLOW s)

{

std::cout<<"Error:"<<s.get\_exception()<<std::endl;

}

return 0;

}

*Output*

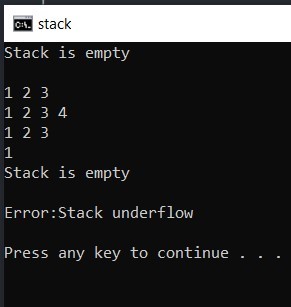


Fig: Push pop operation using stack

**C++ implementation to show applications of stack**

* **Conversion from infix to postfix string**

//To check the precedence

int get\_precedence(char op)

{

if(op=='^')

{

return 3;

}

else if(op=='\*' || op=='/')

{

return 2;

}

else if(op=='+' || op=='-')

{

return 1;

}

else

{

return -1;

}

}

//Converting from infix to postfix

std::string infix\_to\_postfix(std::string text)

{

Stack<char> a;

std::string postfix;

for(int i=0;text[i]!='\0';i++)

{

if((text[i]>='A' && text[i]<='Z') || (text[i]>='a' && text[i]<='z'))

{

postfix+=text[i];

}

else if(text[i]=='(')

{

a.push(text[i]);

}

else if(text[i]=='+' || text[i]=='-' || text[i]=='/' ||text[i]=='\*' || text[i]=='^')

{

if(a.is\_empty())

{

a.push(text[i]);

}

else

{

if(get\_precedence(a.peek())<get\_precedence(text[i]))

{

a.push(text[i]);

}

else

{

while(get\_precedence(a.peek())>=get\_precedence(text[i]))

{

postfix+=a.pop();

if(a.is\_empty())

{

break;

}

}

a.push(text[i]);

}

}

}

else if(text[i]==')')

{

while(a.peek()!='(')

{

postfix+=a.pop();

}

a.pop();

}

}

while(!a.is\_empty())

{

postfix+=a.pop();

}

return postfix;

}

int main()

{

std::string a;

std::cout<<"Enter infix string >>";

std::cin>>a;

std::string pf;

pf=infix\_to\_postfix(a);

std::cout<<pf;

return 0;

}

*Output*

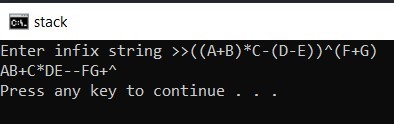


Fig: Conversion from infix to postfix expression

* **Evaluation of postfix string**

#include<math.h>

#include<string>

//evaluation of postfix expression

void evaluate\_postfix(std::string text)

{

Stack<std::string> a;

int num1,num2,result,temp=0;

for(int i=0;text[i]!='\0';i++)

{

if(text[i]==',')

{

continue;

}

else if(isdigit(text[i]))

{

if(text[i+1]!=',')

{

while(text[i]!=',')

{

temp=temp\*10+(text[i]-'0');

i++;

}

a.push(std::to\_string(temp));

}

else

{

a.push(std::string(1,text[i]));

}

}

else

{

num1=std::stoi(a.pop());

num2=std::stoi(a.pop());

switch(text[i])

{

case '+':

result=num2+num1;

break;

case '-':

result=num2-num1;

break;

case '/':

result=num2/num1;

break;

case '\*':

result=num2\*num1;

break;

case '^':

result=pow(num2,num1);

break;

}

a.push(std::to\_string(result));

}

a.display();

}

std::cout<<”Ans:”<<a.pop();

}

int main()

{

std::string text;

text="6,2,3,+,-,3,8,2,/,+,\*,2,^,3,+";

evaluate\_postfix(text);

return 0;

}

*Output*

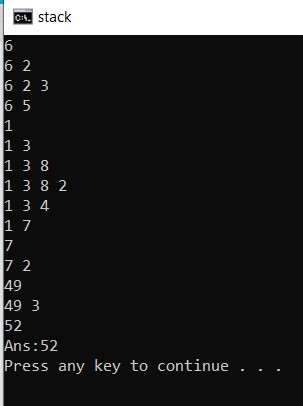
**

Fig: Evaluation of postfix string

* **Converting infix expression to prefix expression**

//converting from infix to prefix

std::string infix\_to\_prefix(std::string text)

{

return reverse(infix\_to\_postfix(reverse(text)));

}

int main()

{

std::string a;

std::cout<<"Enter infix string >>";

std::cin>>a;

std::string pf;

pf=infix\_to\_prefix(a);

std::cout<<pf;

return 0;

}

*Output*

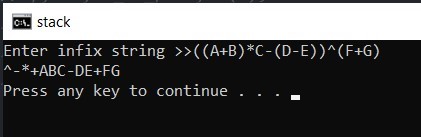


Fig: Conversion from infix to prefix expression

* **Balanced Parenthesis**

//Checking for balanced paranthesis

bool balanced\_parenthesis(std::string expr)

{

Stack<char> a;

for(int i=0;expr[i]!='\0';i++)

{

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

{

a.push(expr[i]);

}

else if(expr[i]==')')

{

if(a.is\_empty())

{

return false;

}

else

{

a.pop();

}

}

}

return a.is\_empty()?true:false;

}

int main()

{

std::string text;

std::cout<<”Enter string to check>>”;

std::cin>>text;

if(balanced\_parenthesis(text))

{

std::cout<<”expression is correct”<<std::endl;

}

else

{

std::cout<<”expression is not correct”<<std::endl;

}

return 0;

}

*Output*

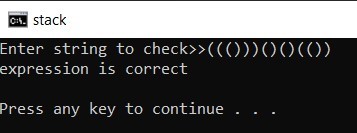


Fig: Checking paired parenthesis