Stacks

02 July 2022 18:11

Its work on principal LIFO (Last in first out)

FILO

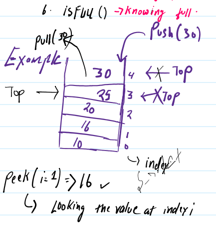
ADT of stack ⇒ Abstract DataType.

ADT contain data representation or operation on the stack.

Data 1. Space for storing elements
2. Top pointer ⇒ which pointing the topmost element.

Operations.
1. push(x)
2. pop()
3. peek(index)
4. Stacktop() → for knowing what is topmost value in stack.
5. isEmpty() → knowing empty.
6. isFull() → knowing full.

Example



peek(i=1) ⇒ 16 ✓
↳ Looking the value at index i

Implementation.

It is done by using two storing data structure.
1) Array
2) Linked list.

1) Array

First Understand Structure/Class.

<make structure or Class first>

```
#include <iostream>
using namespace std; // Implementation by Dynamically.
class Stack {
    public: // for accessing in another class.
    int size; int top; int *s;                 initialize
// Make Constructor to initialize.
    Stack(int siz){ this->top = -1; this->size = siz;
            s = new int[size]; // this new allocate memory
                               // in heap so its dynamically.
    // Now Code of IsEmpty, push, pop, peek>.
      bool IsEmpty(){
              if (top==-1)
              return  true ;
              return false ;
      }
    void push(int val){
        if(top>=size){
            cout<<"overloaded" ;
        }
        else{
            // top ++ ;
            s[++top]=val ;
        }
    }
      int peek(){
       return s[top] ;
      }
       int pop(){
           if(IsEmpty()){
               cout<<"underflow" ;
               return -1 ;
           }
           else{
               return s[top--] ;
           }
       }
};
```

IsEmpty / push / peck / pop

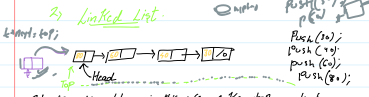
// If we call stack variable outside the class then we do ⇒ st.top etc.

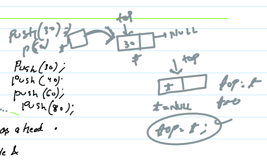
```
    void display(Stack st) {
        for(int i=0; i<= st.top; i++){
            Cout << st.s[i] << " "; }
    }

int main() {
    Stack st (input-Size); // Here st is object
     st.push(5);

     st.push(1000) ;
     st.push(100)  ;
     cout<<"deleted value -> "<<st.pop()<<endl; // 100
     cout<<st.peek()<<endl;   // 1000.
     display(st);      // 1000 5  → // that st (jesme initilize kiye the)
     return 0 ;
  }
```

2) Linked List.





Stack using LL, in this we make top as a head.
In this first push element get into last node &
The top / last push / pop / peek is in the head pointer.

t->next = top;
top=t;

Note:- Push() we can say insert a node at first position.
pop() we can say delete a first Element in LL.

Empty
if(Top==NULL)

Full
Node *t=new Node;
if(t==NULL)

Rough Work:

```
struct/
Class Node {
  int data;
  Node Next;
Node(int val){ this->data=val;
               this->next=NULL;
             }
Push(int val){
node t= new node(val);
 if( t->next==NULL){
        return;
 }
 else{
   t->next=top;
   top=t;
 }
```

Infix to Postfix Conversion:

1) What is Postfix.
2) Why Postfix.
3) Precedence.
4) Manual Conversion.

There are Notation :-

1) Infix : operand operator operand.
 eg a+b

2) Prefix : opt oprnd oprnd
 eg +ab

3) Postfix : oprnd oprnd opt.
 eg. ab+

Note ⇒ Postfix is mostly used.

Infix to Postifix Conversion:

Symbol	Precedence
+, -	1
*, /	2
( )	3

(i) a+(b*c)

prefix
⇒ (a+[*bc])
⇒ (+a*bc)

postfix
a+[bc*]
(abc*+)

(ii) a+b+c*d
 2 3 1 ← precedence

Here both is same operators but give left ope. more precedence.

prefix:
a+b+[*cd]
[+ab]+[*cd]
++ab*cd

postfix:
a+b+[cd*]
[ab+]+[cd*]
ab+cd*+

(iii) (a+b)*(c-d)
 1 3 2

prefix:
[+ab]*[-cd]
*+ab-cd

Postfix:
[ab+]*[cd-]
ab+cd-*

(iv) abc++
a(b+c)+
(a+(b+c))

(vi) ab*c+
(axb)c+
((axb)+c)

Associativity

If Prece.. is same then go with Associativity

Syb	Prece	Assi
+, -	1	L-R
*, /	2	L-R
^	3	R-L
-	4	R-L
()	5	L-R

Ex
(i) a+b+c
 1 2
L-R
(a+b)+c
((a+b)+c)

(ii) a^b^c,
R-L (a^(b^c))

Unary Operator

i) -a → Precendence 4
 → Asso. R-L

if
--a R-L
(-(-a))
Pre → --a
Pos. → (-[a-]) → a--

--a R-L
(-(-a))
Pre → --a
Pos → (-[a-]) ⇒ a--

(ii) *p Infix form
Pre: *p
Post: p*

**p → (*(*p))
Asso: (R-L)

(iii) n!
Pre: !n
Pos: n!

(iv) log x
Pre: log x
Pos: x log

Unary → - + ! log

Example: Convert into Postfix
-a + b * log n!
Using precedence and associativity. R-L

-a + b * log[n!]
-a + b * [n! log]
[a-] + b * [n! log]
[a-] + [b n! log *]
(a- b n! log * +) Ans
Postfix

Infix To Postfix
Conversion
Using Stack.

a+b *c - d/e

⇒ abc* + de/ -
Conv.

Sym	Pre	Asso
+, -	1	L-R
*, /	2	L-R
a, b, c	3	L-R

a+b*c-d/e
End

Pop

Postfix
a b c * + d e / -

Top
+ < * (Push(*))
Pre 1 2

If operator in stack is in lower precedence. then push. otherwise Pop.

* > -
Pre 2 > 1
Pop()
+ -
1 = 1
then also pop()

Now stack is empty then push() that '-'.

At End of expression. Empty the stack and put into postfix. if stack is not empty.

- < /
1 < 2 → Push

End
pop()
(Null)

Post
a + b + c abc++ → ++abc
1 1
Pre.

L-R

Love Push equal push

a-b*c*d/e

string postfix;
postfix = abc*+de/-

char c
int prec ( ) {
if (c == + || c == -) {
return 1;
if (c == * || c == /) {
ret 2
if (c == '(')
ret
-1;

a+b+(c*d+e)
(cd*)+(e)
a+b+ ; cd*e-
ab+ +
ab+ cd*e-+

a b+c

+ -
*
(

Symb	Stack	Postfix
a	Top -	a
+	+	a
b	+	ab
*	* +	ab
c	* +	abc
-	-	abc* +
d	-	abc*+d
/	/ -	abc*+d

c	* +	abc
–	–	abc* +
d	–	abc*+d
/	/ –	abc*+d
e	/ –	abc*+de
–		abc*+de/–

Conclusion :- * All the symbols in stack which is greater or equal than pop().
* If the operator in stack is lower precedence or empty than push().