13/05/22

Stack

Stack is a collection of elements which follow discipline LIFO. (Last in first out)

* ADT of stack -

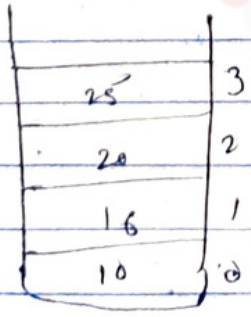
It gives us the definition of stack in terms of data representation and operations

→ Data -

1) space for storing elements
2) Top pointer

→ operations -

1) Push(x) → inserting a value
2) POP() → deleting a value
3) Peek(index) → looking at a value at a given position
4) stackTop() → knowing the top most value
5) isEmpty() → knowing the stack is empty or not
6) isFull() → ——————— full or not



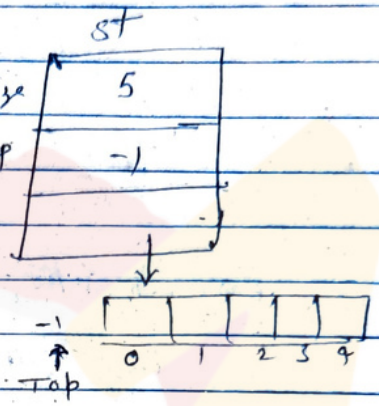
* Implimentation of stack using Array

Date
Page

For this we need -

i) size of the array
ii) Array
iii) Top most pointer

```
struct stock
{   int size;
    int TOP;
    int *S;
};
int main()
{   struct stock st;
    printf("Enter size of stack");
     scanf("%d", &st.size);
    st.S = new int(st.size);
     st.TOP = -1
```



* Empty

```
if (TOP == -1)
```

* FULL

```
if (TOP == size-1)
```

i) Push() operation in stack using array -

```
void Push(stack *st, int x)
  if (st->TOP == st->size - 1)
      printf("stack overflow.")
  else
    { st->TOP++;
      st->s(st->TOP) = x;
    }
}
```

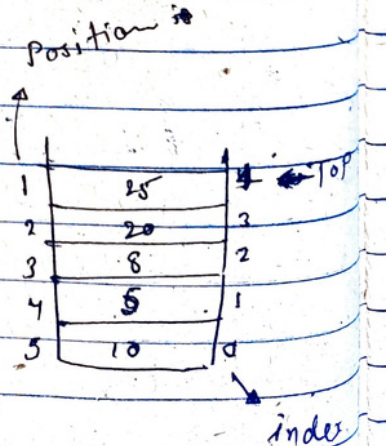
ii) POP() operation

```
int POP (stact *st)
 { int x = -1;
   if (st->TOP == -1)
     printf ("stack is underflow");
  else
   {
     x = st->s[st->TOP];
     st->TOP--;
   }
  return x;
}
```

* Peek() operation

formula

Pos	index (TOP - Pos + 1)
1	4 (4 - 1 + 1)
2	3 (4 - 2 + 1)
3	2 (4 - 3 + 1)
4	1 (4 - 4 + 1)
5	0 (4 - 5 + 1)



```
int Peek (stack st, int pos)
{ int x = -1
  if (st.top - pos + 1 < 0)
    printf ("Invalid Position");
  else
    x = st.s [st.top - pos + 1];
    return x;
}
```

iii) stack TOP operation -

```
int (stack st)
{ if (st.TOP == -1)
    return -1
  else
    return st.s [st.TOP];
```

iv) isEmpty

```
int isEmpty (stack st)
{ if (st.TOP == -1)
    return 1;
  else
    return 0;
```

iv) isFull

```
int isFull (stack st)
{ if (st.TOP == st.size - 1)
    return 1;
  else
    return 0;
```

* Stack using Linked List -

→ New element always insert on left side.

→ First node is the containing the TOP pointer.

→ Empty cond -

if (TOP == NULL)

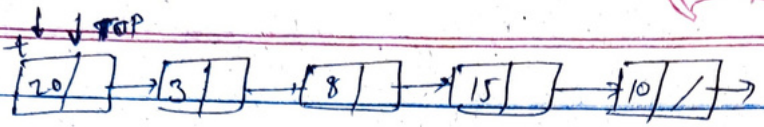
→ FULL cond -

When creating a node and it is not created means stack is Full. It happens when heap memory is fulled.

```
Node *t = New Node;
if (t == NULL)
```

* struct Node
{ int data
struct Node * Next;
};

i) Push operation -

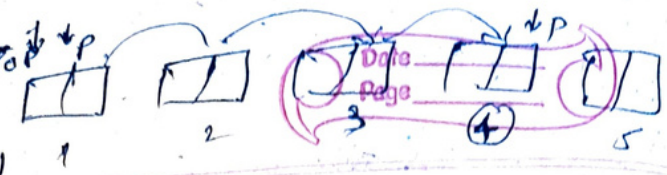



```
void push (int x)
{
    Node *t = new node;
    if (t == NULL)
        printf ("Stack overflow");
    else
    {
        t->data = x;
        t->next = TOP;
        TOP = t;
    }
}
```

ii) POP operation -

```
int POP ()
{ Node *P
    int x = -1;
    if (TOP == NULL)
        printf ("Stack is Empty");
    else
    {
        P = TOP;
        TOP = TOP->next;
        x = P->data;
        free (P);
    }
    return x;
}
```

iii) Peek operation




```
int Peek (int Pos)
{ int i;
  Node *P = TOP;
  for(i=0; P!=NULL && i<Pos-1; i++)
  {
    P = P->next;
  }
  if (P!=NULL)
    return P->data;
  else
    return -1;
}
```

iv) stackTOP()

```
int stackTOP()
{ if (TOP)
    return TOP->data;
  return -1;
}
```

v) isEmpty()

```
int isEmpty()
{
  return TOP ? 0 : 1
```

v) isFull()

```
int isFull()
{ Node *t = new Node
  int r = t ? 1 : 0;
  free(t);
  return r;
}
```

20/05/22

* Application of stack ( ) → Paranthesis bracket

Date
Page

→ Paranthesis matching -

In this case, we have to find out whether the paranthesis are balanced or not means for every opening paranthesis, there must be a closing paranthesis. For this checking stack is to be used.

Eg:- $\left((a+b)*(c-d)\right)$

Rule-1) When you get a opening bracket, ~~push~~ it into the stack.

2) When you get any letter or symbols, just ~~ignore it~~ and move ahead.

3) When you get closing bracket pop out the one bracket from the stack ~~, don't push it into stack~~ and move ahead.

4) After reaching the end of expression, check your stack, if it is empty means paranthesis's matched.

5) If at the end any opening bracket is remaining in the stack, means it's not matching.

6) And if we have one closing bracket in the stack, but the stack is empty, there is no opening bracket to pop out, then also it's not matching.

* Program for paranthesis matching :-

Date
Page

Exp - [(((a+b)*((c-d)))+0]

```
int isBalance(char *exp)
{  struct stack st;
   st.size = strlen(exp);
   st.Top = -1;
   st.s = new char[st.size];
   for(i=0; exp[i]!='\0'; i++)
   {  if(exp[i]=='(')
      push(&st, exp[i]);
      else if(exp[i]==')')
      {  if(isEmpty(st))
            return false;
         else
            pop(&st);
      }
   }
   return isEmpty(st) ? true : false;
}
```

student's exercise -

{ ([a+b] * [c-d]) / e }

Write a program that check given ~~paranthesis~~ expression are paranthesised.

ASCII
40 - (
41 -)
91 - [
93 -]
123 - {
125 - }

```
for (i = 0; exp[i] != '\0'; i++)
{ if (exp[i] == '(' || exp[i] == '{'
        || exp[i] == '[')
    push(&st, exp[i]);
  else if (exp[i] == ')' || exp[i] == '}'
        || exp[i] == ']');
}
if
else if ((exp[i] == ')' && stackTop() != '(') ||
        exp[i] == '}' && stackTop() != '{') ||
        exp[i] == ']' && stackTop() != '[' ))
{
    return false;
}
else
    pop(&st);
}
}
```

* Infix, prefix & postfix expression -

These are three notation which is used to write a expression.

1) Infix - (This is the common method which is used in mathematics by us.)

Rule = operand operator operand

Eg:- a + b

ii) Prefix -

opt opnd opnd

Eg:- + a b

iii) Postfix -

opnd opnd opt

Eg:- ab +

(In computer science, to write an expression, prefix and postfix (most) methods are used because by using this method we can perform all the operation only in one scan but in infix method this is not possible.)

* Infix to Postfix ⇔ conversion -

Date
Page

NOTE - Whenever we write an expression, it should alway fully paranthesised, because compiler needs fully paranthesised expression. no operator should left open.

→ When expression is not paranthesised then the compiler makes it paranthesised logically by using "precedence".

→ So precedence and associativity are ment for precedence not for order of execution. Means that has higher precedence that will paranthesised first.

Eg:- a + b * c

sym	precedence
+, -	1
*, /	2
( )	3

(a + (b * c)) → this is fully paranthesised expression that gives clear cut meaning what operation should be performed on which set of operands -

Q. (a + (b * c))

Prefix

(a + [*bc])

+a*bc

Postfix

(a + [bc*])

abc*+

If the precedence are same then solve it left to right.

Q1 a+b+c*d

Prefix

a+b+c*d

a+b+[*cd]

[+ab]+[*cd]

++ab*cd

Postfix

a+b+c*d

a+b+[cd*]

[ab+]+[cd*]

ab+cd*+

Q2 (a+b)*(c-d)

Prefix

(a+b)*(c-d)

[+ab]*(c-d)

[+ab]*[-cd]

*+ab-cd

Postfix

(a+b)*(c-d)

[ab+]*(c-d)

[ab+]*[cd-]

ab+cd-*

Q3 a+b×c-d^e^f

a+b×c-d^[ef^]

: a+b×c-[def^^]

: a+[bc×]-[def^^]

[abc**]-[def^^]

abc**def^^-

* Associativity -

Date
Page

When the precendence are same then we use associativity Rule.

Sym	Pre	Asso
+, -	1	L - R
*, /	2	L - R
^	3	R - L
- (unary minus)	4	R - L
( )	5	L - R

Q Make the expression fully paranthesised -

1) a + b + c - d

(Here all operators has same precedence then to solve it we use left to right associativity.

a + b + c - d
1 2 3 — L - R

(((a+b)+c)-d)

Post - ab+c+d-

2) a = b = c = 5 ('=' has Right to left asso)
3 2 1 — R - L

(a=(b=(c=5)))

Post - abc5===

3) a ^ b ^ c
2 1

(a^(b^c))

Post - abc^^

* Uniary operator :-

1) - a (Negation of a)

Pre	Post
- a	a -

Eg :- - - a (R - L)

(- (- a))

2) * P

Pre	Post
* P	P *

Eg :- * * P

(Right to left)

(* (* P))

3) n !

Pre	Post
! n	n !

4) log x

Pre	Post
log x	x log

Eg :- - a + b * log n !

(Here uniary [-, log, !] has higher precedence and right to left asso :)

- a + b * log [n !]

- a + b * [n ! log]

[a -] + b * [n ! log]

[a -] + [b n ! log *]

Post - a - b n ! log * +

* Infix to postfix conversion using stack-

Method-1

Rule-1) ~~When~~ Firstly we have to scan the whole given expression.

2) In scanning, when we get an operator, we should push it into the stack and if it is any letter or symbol send it into postfix variable

3) Before pushing that operator, check what are the operators precedence present in the stack, if the operators in the stack have a lower precedence then push that one, otherwise pop out and move ahead and even operator has equal precedence then also pop-out that operator which is present in the stack. and check it till the operator has lower precedence than this checking process is done with all the operators till operator is not pushed.

Eg:- a + b * c - d / e

sym	stack	postfix
a	———	a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
-	-	abc*+
d	-	abc*+d
/	/, -	abc*+d
e	/, -	abc*de

time-O(n)

4) At the end of expression, whatever store in the stacks send to the postfix. ∴ abc*de/-

* Infix to postfix conversion

Date
Page

Method - 02 .

In this procedure, every symbol from infix expression will go into the stack. Previously (method-1) we were sending ~~operands direct~~ operators only into the stack.

Rules - Same as the previous.

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

* Program for infix to postfix conversion -