Stack

↓

linear data structure

Rules that it follows : LIFO

↓

Last In First out

FILO

↓

First In Last out

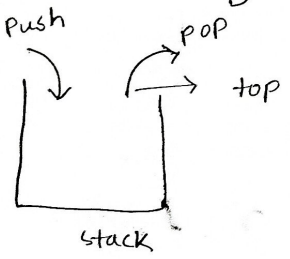
Insertion / Push

Deletion / Pop

} → from one end only

↓

that end is top



operations :

(I) push (x) → can insert only similar type of data

(II) pop ( )

↓ No arguement → will delete topmost data so need no arguement

(III) peek ( ) / top ( ) → it will return the top element of stack but won't delete it unlike pop.

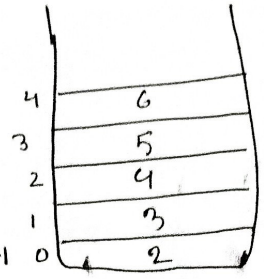
(IV) isEmpty ( ) → will return true if the stack is empty

(V) isFull ( ) → will return true if the stack is full.

Implementation of stack → static → Arrays

Dynamic memory allocation → Linked List

size = 5



Initially top=-1

→ pop ( )
↓
if the stack is empty and we call pop ( ) operation then it will show underflow condition

push (2)
top++ → top will become 0
push (3)
top++
pop ( ) → then 3 will be poped out
top --
pop ( )
top--

→ If the stack is full or the top is pointing to (maximum_size-1) then if we insert an element it will show overflow condition

Application:

(1) reverse a string

(2) Undo

(3) Recursion / function call

(4) Check the balance of parentheses

(5) Infix to postfix / prefix

(6) Evaluation of postfix expression

stack implemented by array

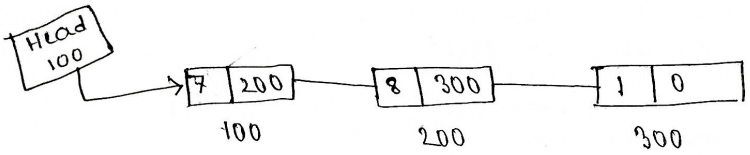
```
# include <bits/stdc++.h>
using namespace std;
int a[5], n=5, top=-1;
void push (int x) {
  if (top >= n-1) {
      cout << " stack overflow " << endl;
    }
  else {
    top++;
    a[top] = x;
   }
}
  void pop () {
int item;
if (top == -1) {
     cout << " stack underflow " <<endl;
   }
else {
  item = a[top];
  top--;
  cout << " the popped element is " << item <endl;
 }
}
void peek () {
 if (top == -1) {
   cout << " the stack is empty " << endl;
  }
else {
 cout << " the top element is " << a[top] <<endl;
}
```

```
void display (){
    int i;
  for(i=top; i>=0; i--){
    cout << a[i] << "  ";
  }
}

int main (){
  int ch, x;
  cout << "Enter choice : 1. push 2. pop 3. peek
          4. display " <<endl;

  do {
  cout << "enter choice " <<endl;
  cin >> ch;
  switch (ch) {
  case 1 : {
    cout << "enter the value to be pushed : " <<endl;
     cin >> x;
     push (x);
     break;
   }
 case 2 :{
   pop ();
   break;
  }
case 3 : {
    peek ();
   break;
  }
case 4 : {
  display ();
  break;
}
```

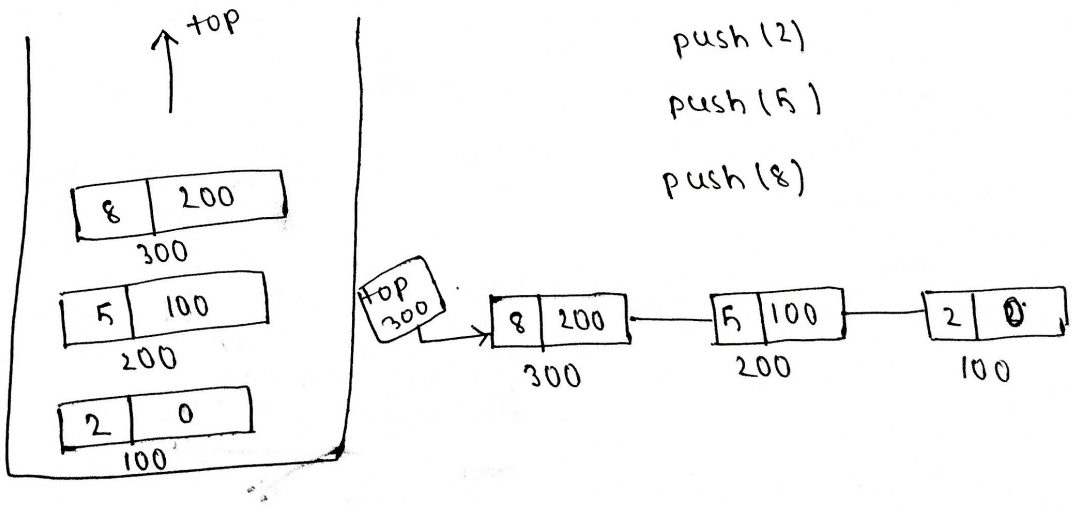
```
        default : {
            cout << "Invalid choice " <<endl;
        }
    }
    while (ch != 4);
    return 0;
}
```

time complexity for push and pop O(1)

stack implemented by linked list



We can insert or delete data from one end. If we insert the data at the end or delete it from the end then the time complexity become O(n) which cannot be happened. so we need to insert data or delete it from the head.



```
# include <bits/stdc++.h>
using namespace std;
class node {
  public :
     int info ;
     node * link;
  }
node *top = NULL;
void push (int x) {
node *ptr = new node () ;
ptr -> info = x;
ptr -> link = top;
top = ptr;
}
void display () {
  node *temp;
  temp = top;
 if (top = = NULL) {
  cout << "the stack is empty " << endl;
  }
else {
  while (temp ! = NULL) {
   cout << temp -> info << " " ;
   temp = temp -> link;
   }
  cout << endl;
  }
}
```

```
    void peek() {
      if (top == NULL) {
        cout << "the stack is empty" <<endl;
      }
      else {
      cout << "the top element is " << top->info
                                    << endl;
      }
    }

void pop() {
  node *temp;
  temp = top;
 if(top == 0) {
   cout << "stack underflow" <<endl;
  }
else {
 cout << "the popped element is " << top->info <<
                                      endl;
top = top->link;
delete temp;
}
}
```

Infix expression

Infix expression ⇒ <operand> <operator> <operand>

↓

It follows some rules like precedence & associativity of operators

(I) ( ) { } [ ]

(II) ^ — R-L

(III) * / — L to R

(IV) + — (L to R)

→ 1+2 *5 +30/5

⇒ 1+10 + 30/5

→ 1+10+6

→ 11+6

→ 17

→ 2^ 2^ 3

→ 2^ 8

→ 256

→ it is costly in terms of memory consumption and time.

Prefix expression (Polish Notation)

<operator> <operand> <operand>

5+1 → + 5 1

infix prefix

a*b+c → *ab+c

infix → · + * abc

prefix

postfix expression
(Reverse Polish Notation)

<operand> <operand> <operator>

5 + 1 ⇒ 5 1 +
infix postfix

a * b + c ⇒ a b * + c

→ a b * c +
infix postfix

Infix to Postfix

infix : A + B / C

stack
/
+

postfix expression
A B C / +

Infix : A − B / C * D + E

stack
+

LIFO.

postfix expression
A B C / D * − E +

Conversion Rules

(i) Print operands as they arrive.

(ii) If the stack is empty or contains a left parentheses on top, push the incoming operator onto the stack.

(iii) If incoming symbol is '(', push it onto stack.

(iv) If incoming symbol is ')', pop the stack and print the operators until left parentheses is found.

(v) If incoming symbol has higher precedence, than the top of the stack, push it on the stack.

(vi) If incoming symbol has lower precedence, than the top of the stack, pop and print the top. Then test the incoming operator against the new top of the stack.

(vii) If incoming symbol has equal precedence with the top of the stack, use associativity rule.

(1) associativity L to R then pop and print the top of the stack and then push the incoming operator.

(2) R to L, then push the incoming operator.

(viii) At the end of the expression, pop and print all operators of stack.

Infix to Postfix

Infix : A + B * C
A + (B*C)
A + BC *

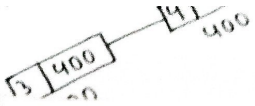
postfix : ABC*+ (without using stack)

↓

time consumping method

Infix : K + L - M * N + (O ^ P) * W / U / V * T + Q

Input	stack	postfix exp.
K	•	K
+	+	K
L	+	KL
-	-	KL+
M	-	KL+M
*	-*	KL+M
N	-*	KL+MN
+	+	KL+MN* -
(	+(	KL+MN*-
O	+(	KL+MN*-O
^	+(^	KL+MN*-O
P	+(^	KL+MN*-OP
)	+	KL+MN*-OP^
*	+*	KL+MN*-OP^
W	+*	KL+MN*-OP^W
/	+/	KL+MN* -OP^W*
U	+/	KL+MN*-OP^W*U
/	+/	KL+MN*-OP^W*U/
V	+/	KL+MN*-OP^W*U/V
*	+*	KL+MN*-OP^W*U/V/
T	+*	KL+MN* -OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T*+
	+	KL+MN*-OP^W*U/V/T*+Q



KL +MN*-OP^W* U/V/T *+Q+

↓

postfix

Exp.

Infix Exp : K+L-MXN+(O^P) X W/U/V X T+Q

Infix Exp : A-B+(M^N) X (O+P) -Q/R ^S XT+Z

stack

Postfix Exp

AB-MN^OP+X+Q RS^/TX-Z+

Infix to Prefix using stack

A +B *C → A+(B*C)

→ A+ *BC

→ +A*BC (without using stack)

time consumping .

Infix to prefix

K + L − M * N + (O ∧ P) * W / U / V * T + Q

Q + T * V / U / W *) P ∧ O (+ N * M − L + * K

Input	Stack	Prefix
Q		Q
+	+	Q
T	+	QT
*	+ *	QT
V	+ *	QTV
/	+ * /	QTV
U	+ * /	QTVU
/	+ * //	QTVU
W	+ * //	QTVUW
*	+ * // *	QTVUW
)	+ * // *)	QTVUW
P	+ * // *)	QTVUWP
∧	+ * // *) ∧	QTVUWP
O	+ * // *) ∧	QTVUWPO
(	+ * // *	~~QTVU~~ QTVUWPO∧
+	+ +	QTVUWPO∧*//*
~~+~~ N	+ +	QTVUWPO∧*//*N
*	+ + *	QTVUWPO∧*//*N
M	+ + *	QTVUWPO∧*//*NM
−	+ + −	QTVUWPO∧*//*NM*
L	+ + −	QTVUWPO∧*//*NM*L
+	+ + − +	QTVUWPO∧*//*NM*L
K	+ + * − +	QTVUWPO∧*//*NM*LK

QTVUWPO∧*//*NM*LK+−++

↓

++−+KL*MN*//*∧OPWUVTQ

↓

prefix Exp.

K + L - M * N + (O ^ P) * W / U / V * T + Q

~~* M N * ^ O P W~~

+ + - + K L * M N * / / * ^ O P W U V T Q

conversion rules:

(I) If the stack is empty or contains right parentheses on top, push the incoming operator onto the stack.

(II) If the incoming symbol is ')', push it onto the stack

(III) If the incoming symbol is '(', pop the stack and print the operators until the right parentheses is found.

(IV) precedence is same as postfix.

(V) Associativity rule

(1) If L to R, then push the incoming operator.

(2) If R to L, then pop and print the top of the stack and then push the incoming operator

→ Reverse the infix Exp and solve it. then reverse the solved one. then will get the prefix Exp.

Evaluation of Prefix Exp

Infix : - a + b * c - d / e ^ f

prefix : - + a * bc / d ^ ef

a = 2, b = 3, c = 4, d = 16, e = 2, f = 3

⇒ - + 2 * 3 4 / 16 ^ 2 3 (right to left)

2 ^ 3 ← find operator then immediate next two operands

~~pattern~~ pattern should be <operator> <operand> <operand>

2 ^ 3

⇒ - + 2 * 3 4 / 16 8 (16/8)

⇒ - + 2 * 3 4 2 ← (3 * 4)

⇒ - + 2 12 2 (2 + 12)

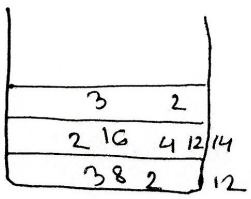
⇒ - 14 2 (14 - 2)

⇒ 12 (without using stack)

⇒ - + 2 * 3 4 / 16 ^ 2 3

from right to left & if found any operand, then push it onto the stack

& if operator ~~for~~ found, then pop two operands



op1 = 2 op2 = 3

2 ^ 3 = 8

16 / 8 = 2

3 * 4 = 12

12 + 2 = 14

14 - 2 = 12

Algorithm:

Scan Prefix exp. from right to left

for each character in prefix expr.

do

if operand is there, push it onto stack.

else if operator is there, pop 2 elements.

op1 = top element

op2 = next to top element

result = op1 operator op2

Push result onto stack

return stack [top]

Evaluation of Postfix Expn

Postfix : <operand> <operand> <operator>

Infix : a + b * c - d / e ^ f

postfix : a b c * + d e f ^ / -

2 3 4 * + 16 2 3 ^ / -

→ 2 12 + 16 2 3 ^ / - (3*4) (from left to right)

find a operator and immediate 2 operands

→ 14 16 2 3 ^ / - (2+12)

→ 14 16 8 / - (2^3)

→ 14 2 - (16/8)

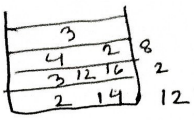
→ 12 (14-2)

(without using stack)

using stack

2 3 4 * + 16 2 3 ^/-

op2 = 4
op1 = 3



op1 op op2

3 * 4
= 12

2 + 12
= 14

$2^3 = 8$

16/8 = 2

14-2 = 2

Begin

for each character in postfix Exp, do

if operand is encountered, push it onto stack

else if operator is encountered, pop 2 elements

A → top element

B → next to top element

result = B operator A

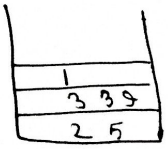
push result onto stack

return element of stack top

End.

~~2 x 3~~

→ 2 3 1 x + 9 -

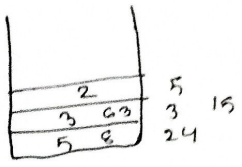


3 x 1 = 3

2 + 3 = 5

5 - 9 = -4 result = -4

→ 5 3 + 6 2 / * 3 5 * +



5+3 = 8

6/2 = 3

8*3 = 24

3*5 = 15

24+15 = 39

result is 39.

Postfix to Infix

postfix : a b + e f / *

(from Left to right)

A = b

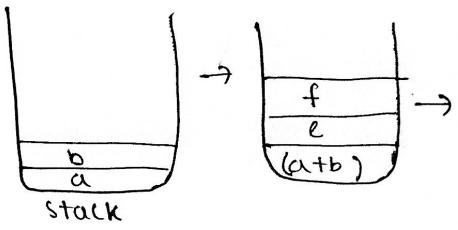
B = a

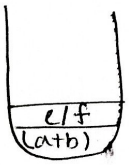
B op A

(a + b)

(e/f)

((a+b) * (e/f))





→ if operands come, simply put operands onto the stack

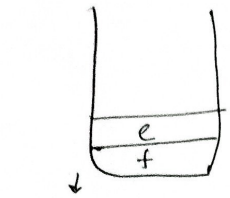
→ if operator comes, pop two operands from the stack and place the operator between the operands.

Prefix to Infix

* + ab / ef ←

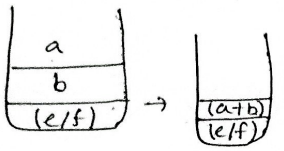
(from right to left)

A op B
↓ top element
B ↓ next to top element



A = e
B = f

A op B
(e/f)
(a+b)

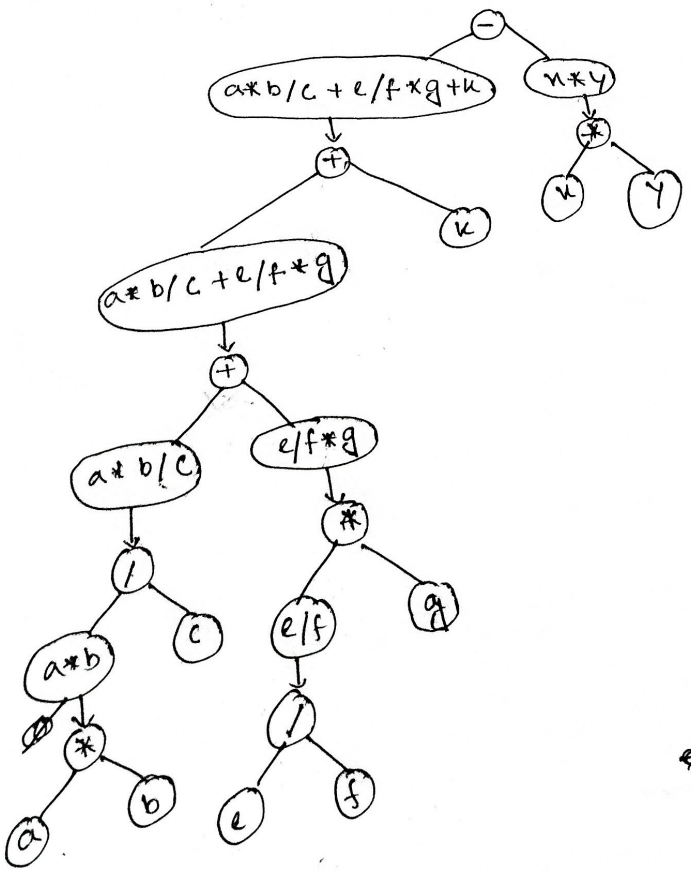


→ ((a+b) * (e/f))
result

Binary Expression Tree

(Infix)

Infix Exp : a * b / c + e / f * g + k - x * y



leaves : operands

internal nodes : operator

Roots : low precedence operator

→ preorder will give prefix Exp

→ postorder will give postfix Exp.

preorder = Root L Right

postorder = Left Right Root

- - + + / * a b c * / e f g k * x y
prefix

a b * c / e f / g * + k + x y * -
~~post~~ postfix

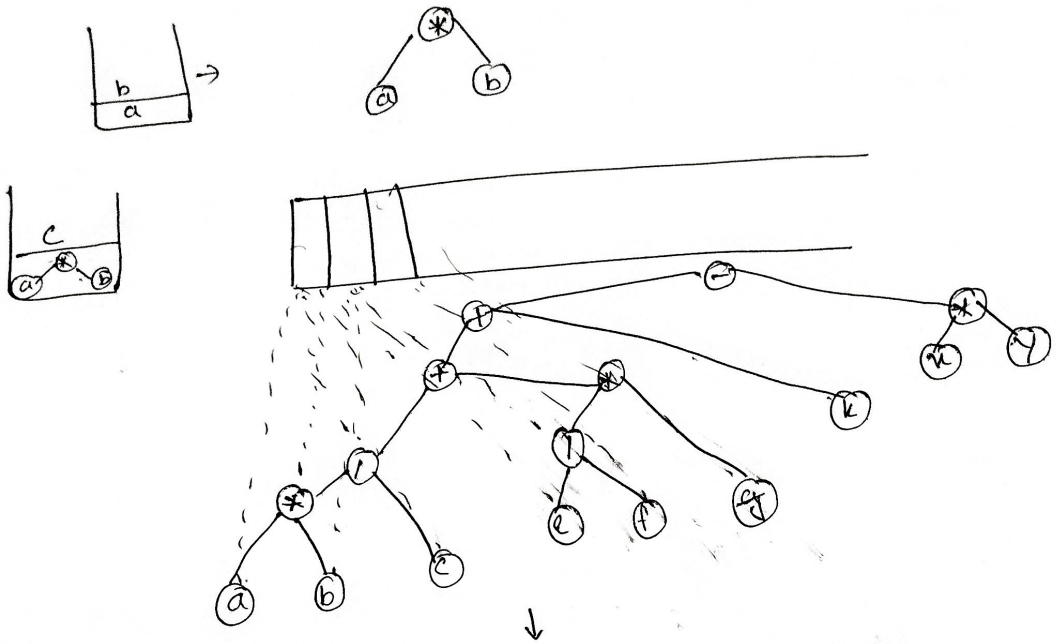
Binary Expression Tree

Postfix

Infix : a * b / c + e / f * g + k - x * y

postfix : ab * c / ef / g * + k + xy * -

(Left to right)



postfix Exp