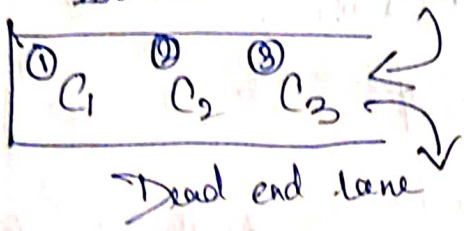
Stack (stack ADT) (LIFO)

1) Introduction / Application 2) ADT stack

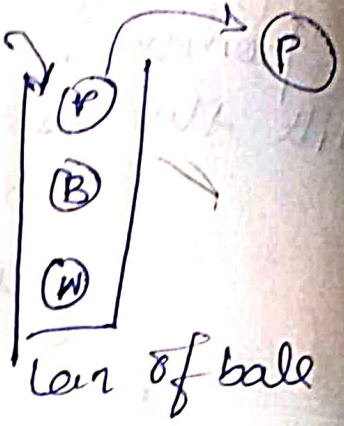
Last in first out

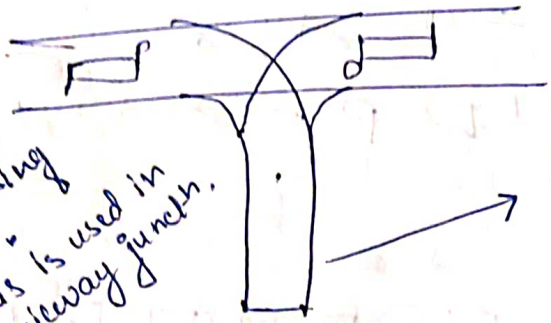
1st came 1st and 3rd go first

(a)



(b)





Recursion uses stack (internal stack)

Sometimes to convert R → Iteration

Programers creates stacks (called as programmers stack)

ADT of Stack

(Data representation + operation)

Data:

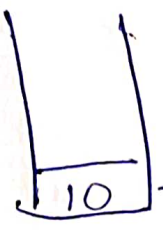
1) We need ~~start~~ space for storing

2) Top pointer (Pointing on the top most element of the stack. coz it is most important)

Operation

1) Push (x) ↓

2) Pop ( ) →



3) peek (index) ⇒ to know the element
↳ knowing a value at a particular index.

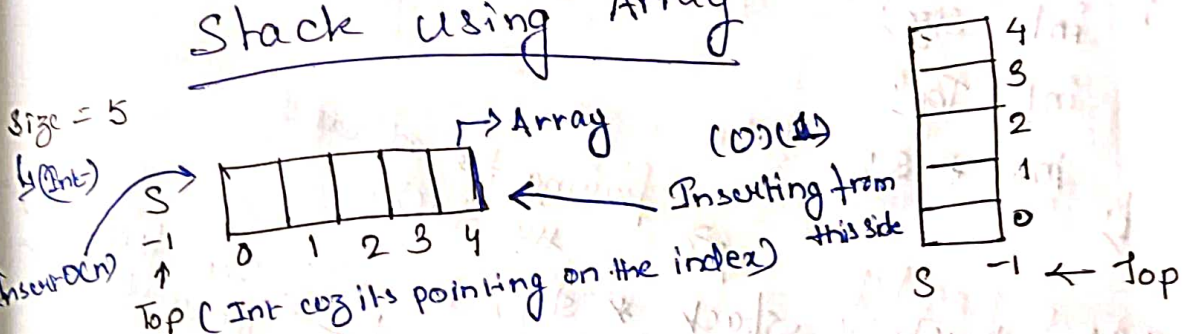
4) stackTop ( )

5) isEmpty ( )

6) isFull ( )

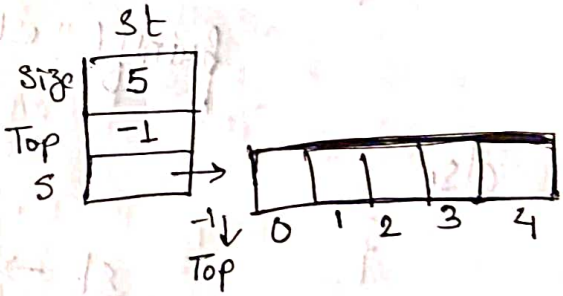
* stack is a collection of elements
So it can be implement by either
1) Array 2) Linked list.

Stack using Array



↳ size (int) ↳ Top pointer (int) ↳ array

```
Struct stack
{ int size;
  int Top;
  int *s;
};
int main()
{
  struct stack st;
  printf("Enter size");
  scanf("%d", &st.size); //5
  st.s = new int [st.size];
  st.Top = -1;
}
```



basic structure

Empty (if (Top == -1))

Stack full Top (if (Top == Size - 1))

Push

If it will first check the top pointer if it is not equal to size-1 then we will insert a value in the stack then we increment top as it is int- value so we will just do top++.

Pseudo code

```
Struct stack
{ int size;
  int Top;
  int * s;
};

void push (stack * st, int x)
{ if (st -> Top == st -> size-1) // for checking overflow condition
    printf("stack overflow");

  else
    {
      st -> Top++;
      st -> s[st -> Top] = x;
    }
}
```

↑ passing the address of the stack.

↳ the element that we want to insert

Pop

First if we want to pop out an element from the stack, then we have to first check the underflow case. If top pointer is not equal to -1 then we will first declare a variable x and initialise it -1 then store the top element of the array [S] using st pointer in x and then decrement top at the end return x.

```
struct stack
{ int size;
  int top;
  int *s;
};
```

```
int pop (stack * st)
{ int x = -1;
  if (st->Top == -1)
      printf("Stack underflow");
  else
  {
      x = st->s[st->Top];
      st->Top--;
  }
  return x;
}
```

we are storing the value for returning back if the stack is already empty then it will return -1 that means stack underflow condition

Time - O(1) for both push and pop.

Peep function

Here they have given a index where we have to find the corresponding element of that index so for doing that first we have to compute that Top - pos + 1 from the top and return that x.

```
int peek (stack st, int pos)
{ int x = -1;
  if (st.Top - pos + 1 < 0) pf("Invalid pos");
  else
  { x = st.s[st.T - p+1];
    return x;
  }
```

Stack Top (To know about the topmost value of the stack.)

So first it will check if stack is not empty then it will return the value of top.

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

Stack Empty

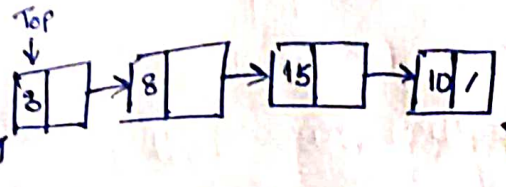
We will check if the Top is -1 then it will return 1 (True) else it will return

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

Stack Full

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

Stack Using Linked List



Inserting at this position will take O(1) time

Inserting from this end takes O(n) time

So every time we will push and pop the element from the top.

Empty condition

when top is not pointing to any node. i.e (Top == NULL)

Full condition

Node * t = newNode; { That is when the heap is full i.e if we create the Node and it is not created.

if (t == NULL)

Stack Operation Using Linked List

1) Push Operation

Time taken O(1)

```
void push (int x)
{  Node * t = new Node;
   if (t == NULL)
      printf("Stack Overflow"); } To check the overflow condition
   else
   {  t->data = x;
      t->next = Top;
      Top = t;
   }
}
```

} Inserting before the first node.

```
Struct Node
{  int data;
   Struct Node * next;
}*Top = NULL;
```

2) Pop function (To remove an element from stack)

```
int Pop()
{ Node *P;
  int x = -1;
  if (Top == NULL)
      printf("Stack is Empty");        } To check the underflow condition
  else
  {  P = Top;
     Top = Top->next;
     x = P->data;                      } Deleting the first node from linked list.
     free(P);
  }
  return x; // Returning the deleted value
}
```

3) Peek Operation (Find the corresponding index number)

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

Stack Top

```
int stackTop()
{ if(Top)
    return Top->data;
  return -1;
}
```

isEmpty

```
int isEmpty()
{ return Top?0:1;
}
```

isFull

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

Parenthesis Matching

Application of Stack

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

(for every open paranthesis there must be a closing Paranthesis).

* Here first we have to create a stack and then go through the expression.

3 cases may arrive

1) If the symbol is (→ opening bracket
↳ push it into the stack

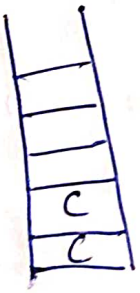
2) If it is the) → closing bracket
↳ pop out the last opening bracket.

3) If it is neither) nor (then ignore more next.

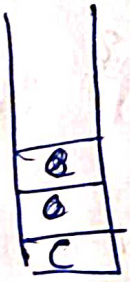
Eg ((a+b) * (c-d))

S1

S2



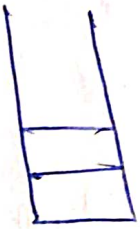
S3



S4

S5

S6



Thus the paranthesis is matched.

Eg: (((a+b) * (c-d))↑

If we reached the end of the expression but still the stack is not empty then the parenthesis is not matched.

Eg: ((a+b) * (c-d)))
↑

if we have a closing bracket but the stack is empty i.e there is no match found for the closing bracket then the parenthesis is not matched.

Eg: (a+) b(* c-d) ✓

This procedure dosent check if the parenthesis given obey the correct operand or not it only check wether the parenthesis is balanced or not.

Pseudo code

```
struct Node
{ char data;
  struct Node *next;
} *top = NULL;

int isBalanced (char *exp)
{ int i;
  for (i=0; exp[i] != '\0'; i++)
  { if (exp[i] == '(')
      push (exp[i]);
    else if (exp[i] == ')')
      if (top == NULL) return 0; pop ();
  }
}
```

```
if (top == NULL)
    return 1;
else
    return 0;
}
```

Infix to Postfix conversion...

1. what is postfix.
2. why postfix
3. Precedence
4. Manual conversion.

* There are 3 notation of writting a expression

1. Infix : operand, operator, operand
(used in common mathematics and comp)
Eg: a+b (a added with b)
↳ binary operator.

2. Prefix : operator, operand, operand
mostly used in comp
Eg: +ab (add a and b)
↳ this operation has to perform on a and b.

3. Postfix : operand, operand, operator
Eg: ab+ (a and b perform operation)
↳

Why we need Prefix and Postfix

Infix: $8 \overset{6}{+} 3 \overset{5}{*} (9-6) \overset{1}{/} 2^{2} \overset{3}{+} 6 \overset{2}{/} 2$

If we want the ans to the above expression then with BODMAS Rule we will solve it.

Postfix: $8\ 3\ 9\ 6\ -\ 2\ ^{2}\ /\ *\ +\ 6\ 2\ /\ +$

(1 2 3)

→ when we go to solve this expression then we have to jump here and there to perform which operation we have to perform and then next which one.

for finding out which one I have to perform next I have to scan the whole expression.

So we cannot able to write a program which randomly jumps from here and there.

* If we want to find the result in 1 scan it is not possible in Infix form.

So we write the Postfix form (↑)

Here we can perform the whole operation in 1 single scan.

(If we find an operand then add it in the stack else do operation on the last 2 operand)

Conversion from Infix to Postfix

If a + b * c

If a + b + c * d

Precedence table

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

Equal →

Key Points

If whenever you are writing any expression note that you should make the expression fully Parenthesis

compiler needs fully parenthesis expression.

If when the expression is not parenthesised then compiler will make it parenthesised according to the precedence table.

Eg If a + b * c

S1 Mul has ↑ precedence so compiler will first parenthesis (b * c)..

↳ a + (b * c)

S2 (a + (b * c))

↙

S3 since (b * c) has a higher precedence

(a+(b*c))

↓

(a+[*bc])

↓

[+[a][*bc]]

↓

+a*bc → prefix form

when we put it in the square bracket then we will have to consider it to operand.

for postfix form

(a+(b*c))

↓

(a+[bc*])

↓

[a][bc*]+

↓

abc*+ → postfix expression.

(not much required).

when 2 operations are on the same precedence then go from left to right.

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

Prefix

(a+b+[*cd])

↓

([+ab]+[*cd])

↓

[+[+ab][*cd]]

↓

++ab*cd

Postfix

(a+b+[cd*])

↓

([ab+]+[cd*])

↓

[[ab+][cd*]+]

↓

ab+cd*+

Q3 (a+b)*(c-d) (when parenthesis is already done)

Prefix

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

↓

([+ab]*(c-d))

↓

([+ab]*[-cd])

↓

[*[+ab][-cd]]

↓

*+ab-cd

Postfix

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

↓

([ab+]*(c-d))

↓

([ab+]*[cd-])

↓

[ab+][cd-]*

↓

ab+cd-*

Associativity

1) Associativity

2) Unary operators

Symb	Pre	Asso
+, -	1	L-R
*, /	2	L-R
^	3	R-L
-	4	R-L
( )	5	L-R
*		R-L
log		R-L
!		R-L

Pow ←

Unary (-) ← (Minus)

defreserv ←

log ←

factorial ←

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

1

2

3

Eg (a=(b=(c=5))) → R--L

1

2

3

Postfix - abc5===

$(a \wedge (b \wedge c)) = (a^{b^c})$ [a][b][c] ^

$a \wedge b \wedge c = (a^b)^c$

Postfix

$(a \wedge [bc\wedge])$

$[[a][bc\wedge]\wedge]$

$abc\wedge\wedge$

Unary Operator

(negation of a)

1) $-a$ (Uniary Minus)

Pre $-a$ $\quad$ $a-$ → Post

2) $(-(-a)) \rightarrow$ R-L

3) $*p$ $\quad$ Pre $*p$ $\quad$ Post $p*$

(dereferencing operator in C)

(for accessing the data of pointer)

4) $(*(*p)) \rightarrow$ R-L

5) $n!$ $\quad$ Pre $!n$ $\quad$ Post $n!$

6) $\log x$ $\quad$ Pre $\log x$ $\quad$ Post $x\log$

Eg: $-a+b*\log n!$

convert to postfix

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

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

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

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

$[a-][bn!\log*]+$

$[a-bn!\log*+]$

$a-bn!\log*+$ Ans

Prefix do it in hw

$+-a*b\log!n$

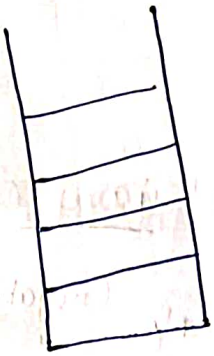
$+-a*b\log!n$ prefix

Infix To Postfix Using Stack (M1)

Precedence Table

Syb	Pre	Asso
+,-	1	L-R
*,/	2	L-R

Eg a + b * c - d / e



Postfix

Step1 If the value is a operand then send it to the postfix.

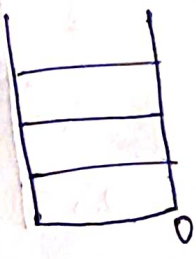
Step2 If the stack is empty then push the operator in it.

Step3 If the stack is not empty check have any operator of lower precedence then push it in the stack

Step4 If the stack contain greater or equal precendence operator then pop them out and push the current operator.

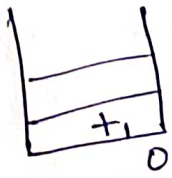
Step5 When the whole expression is traversed then pop out the whole stack one by one from top

Eg S1 a + b * c - d / e
↑



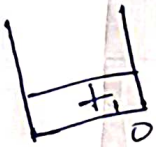
Postfix → a

S2 a + b * c - d / e
↑



Post → a

S3 a + b * c - d / e
↑



Post → ab

S4 a + b * c - d / e
↑

Post → ab

S5 a + b * c - d / e
↑

Post → abc

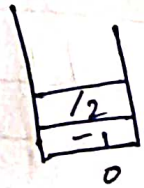
S6 a + b * c - d / e
↑

Post → abc*+

S7 a + b * c - d / e
↑

Post → abc*+d

S8 a + b * c - d / e
↑



Post → abc*+d

S9 a + b * c - d / e
↑

Post → abc*+de

S10 a + b * c - d / e Postfix **abc*+de/-** Ans

pop out all the stack.

Tabular Representation

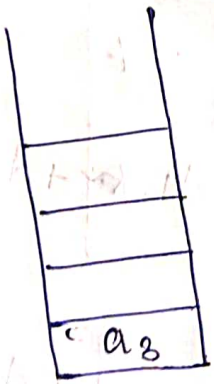
a + b * c - d / e

Symb	stack	Postfix
a	——	a
+	+	a
b	+	ab
*	*, +	ab
c	*, +	abc
-	-	abc*+
d	-	abc*+d
/	/-	abc*+d
e	/-	abc*+de

End of Exp → abc*+de/-

Infix To Postfix (Method 2)

a + b * c - d / e



Postfix.

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

Sym	Stack	Postfix
a	a_3	
+	$+_1$	a
b	$+_1$ b_3	a
*	$+_1$ $*_2$	ab
c	$+_1$ $*_2$ c_3	ab
-	-	abc*+
d	-d	abc*+
/	-/	abc*+d
e	-/e	abc*+d

abc*+de/-

Muggle,

Infix to Postfix conversion

```
int IsOperand (char x)
{ if (x == '+' || x == '-' || x == '*' || x == '/')
        return 0;
    else
        return 1;
}
```

// to check if a character is operand or not if operand then return 1 (true) else false (0)

```
int pre (char x)
{ if (x == '+' || x == '-')
    return 1;
  else if (x == '*' || x == '/')
    return 2;
  return 0;
}
```

// this function is for returning the precedence of an operator

```
char * convert (char * infix)
{ struct stack st; // initialise (char type stack)
  char * postfix = new char [strlen (infix)+1];
  int i=0, j=0;
  while (infix [i] != '\0')
  { if (isoperand (infix [i]))
        postfix [j++] = infix [i++];
    else
    { if (pre (infix [i]) > pre (stackTop (st)))
        push (&st, infix [i++]);
      else
        postfix [j++] = pop (&st);
    }
  }
```

→ creating a array of char type for storing postfix expression

```
while ( ! isEmpty (st))
{
    postfix[j++] = pop(&st);
}
postfix[j] = '\0';
return postfix;
}
```

infix

0	1	2	3	4	5	6	7	8	9
a	+	b	*	c	-	d	/	e	\0

Sym	Precedence	Associativity
+, -	1	L-R
*, /	2	L-R

Evaluation Of Postfix

Infix 3 * 5 + 6 / 2 - 4

Postfix 3 5 * 6 2 / + 4 -

Rule If operand then push it into the stack.

If operator then pop out last 2 operand

put top on Right side

put top-1 on left side

put the operator in the middle

L R

3 * 5 = 15

6 / 2 = 3

15 + 3 = 18

18 - 4 = 14 Ans

Symbol	Stack	Operation
3	3	
5	5, 3	
*	15	3 * 5 = 15
6	6, 15	
2	2, 6, 15	
/	3, 15	6 / 2 = 3
+	18	15 + 3 = 18
4	4, 18	
-	14	18 - 4 = 14

Time = O(N)

Function for Evaluation of Postfix Expression

Postfix array

8	5	*	6	2	/	+	4	-	\0

```
int Eval (char * postfix)
{ struct stack st;
  int i, x1, x2, r;
for (i=0; postfix[i] != '\0'; i++)
{
  if (isOperand (postfix[i]))
      push (&st, postfix[i] - '0');
  else
  { x2 = pop (&st);
    x1 = pop (&st);
  switch (postfix[i])
{ case '+': r = x1 + x2;
            push (&st, r);
            break;
  case '-': r = x1 - x2;
            "do"
  case '*': r = x1 * x2; "do"
  case '/': r = x1 / x2; "do"
}
}
return pop (&st);
}
```

if operand push it in stock

if operator

↳ because it will store the ascii value of operand.