Unit - II

Linear Data structures

→ Stacks: Implementation, operations, Application Notation, infix, prefix & postfix, Conversion & evaluation of arithmetic expressions using stacks.

→ Queue - implementation, operation, types of queue doubleended, priority.

→ Linked list - types - Singly, doubly & Circularly linked list, Operation.

Stack → linear data structure, limited access is possible, Ordered list. follow rule for insertion & deletion

LIFO principle
FILO

Rule →

Insertion & deletion is possible only from one end.

LIFO

insert from top only

take out

plates

Operations

→ Push (x) → data — Insertion
→ POP() — Deletion
→ Peek() / top() — return top most data but donot remove
→ isEmpty() → T/F (Stack Empty)
→ isFull() → True (stack is full)

similar data type only

only one open end

top →

stack

Size = 5

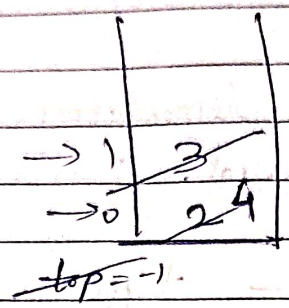
top = -1

logical representation of stack

Implementation → arrays
- → static memory allocation.
- → dynamic memory allocation → linked list.

if stack is empty & we give pop() operation then underflow condition.

```
Push (2)
top++ ;
Push (3)
top ++ ;
```




```
pop()
top--;
pop()
top--;

Push (4)
top++
```

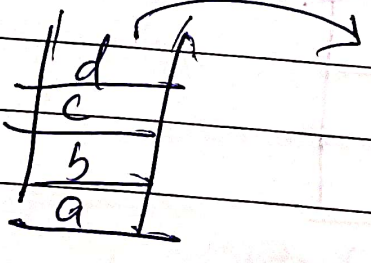
If stack is full & we give push(e) then it is overflow condition.

isFull() → True.

Application

→ reverse a string

abcd ⇒ dcba



→ undo mechanism in text editor.

→ Recursion / function call

→ Balancing of the paranthesis.

({ [ ] })

→ Convert Infix to postfix / ~~Prefix~~

→ Evaluation of postfix expression

→ maintain page visiting history in a web browser.

Indirect

(1) Graph traversal
(2) Tree traversal
(3) Push down automata

etc
Tower of Hanoi

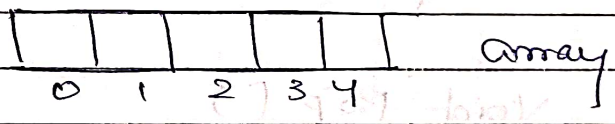
Implementation

1) Using Arrays
2) using linked-list

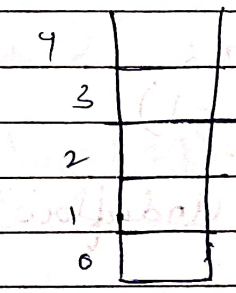
(1) Using Array.

declaration

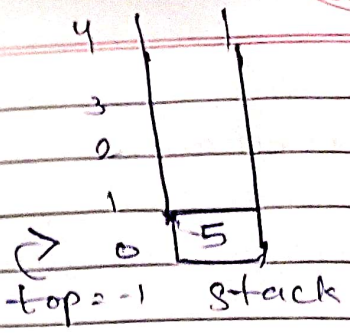
```
int a[5];
```



5 × 4 = 20 bytes



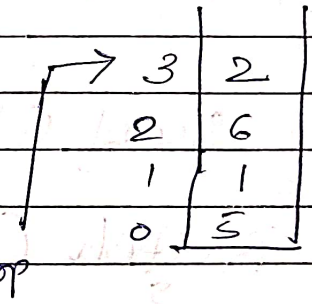

```
int stack[5];

Void main()
{ push();
  pop();
  peek();
```




```
#define N 5;
int Stack [N];
int top = -1

void push ()
{ int x;
  printf ("Enter data:");
  scanf ("%d", &x);

if (top == N-1)
    {
      printf ("overflow");
      }

else
    {
      top ++ ;
      Stack [top] = x;
      }
}

Void pop ()
{
 int item;
if (top==-1)
 {
printf ("Underflow") ;
    }
```

```
else
{
  item = stack[top];
  top--;
  printf("%d", item);
  {
```

```
Void peek()
{
  if (top == -1)
  {
    printf("stack is Empty")
  }
  else
  {
    printf("%d", stack[top]);
  }
```




```
Void display()
{
  int i;
for (i = top; i >= 0; i--)
  {
    printf("%d", stack[i]);
  }
}
```

o/p ⇒ i = ~~3 2 1 0~~
2 6 1 5

```
Void main ()
  {
  push ();
  pop ();
  Peek ();  /top ();
  display ();
  }
```

```
# define N 5;
  int stack[N];
  int top = -1;

void main ()
{ int ch;
  clrscr ();
do
{ printf ("Enter choice: 1: Push 2: Pop
          3: Peek 4: display);

  Scanf ("%d", &ch);

Switch (ch)
  {
Case 1:  push ();
         break;

Case 2:  pop ();
         break;

Case 3:  Peek ()
         break
```

```
case 4 : display ();
         break;
default: printf ("Invalid choice");
    }
} while (ch != 0);
    getch ();
}
```

Infix Prefix Postfix

Expression — combination of Contants, variables, parantheses, symbols, operators

↳ 5+1 ↳ a+5

↳ P+q

binary operators

↳ {<operand> <operator> <operand>.} Expression

(a-1) + 5

↓ operand ↓ operand

Infix ⇒ <Operand> <operator> <operand>

Notation

Rules

evaluate → 5+1 * 6 = 6 * 6 = 36
according to rules 5 + 6 = 11

Brice
Page :
Date : / /

Rules

precedence, associativity.

1) ( ) { } [ ] ~~R-L~~
2) ^ exponential/Power R-L
3) * / L-R
4) + - L-R

→ 5 + 1 * 6

5 + 6

= 11 ✓

→ (5+1) * 6

6 * 6 = 36 ✓

→ 1 + 2 * 5 + 30 / 5

* ⊗ ÷ same precedence
So Check associativity
Same precedence, L-R associativity
L-R Scan left to right
which we have first evaluate it first

Scan from left →

→ 1 + 10 + 30/5

→ 1 + 10 + 6 (L-R.)

→ 11 + 6

→ 17 Ans

The postfix Notation is more suitable for a Computer to evaluate any expression & etc is Universally accepted Notation for designing ALU & there is no need of Brackets while to evaluate it.

Brice

$2 \wedge 2 \wedge 3$ [R-L]

$\leftarrow$

$2^3 = 8 \quad 2 \wedge 8$

$2^8 = 256$

Processing of infix expression is difficult & Costly interm of time & memory here we have Prefix & Postfix ~~we don't need~~ any

Prefix & Postfix Expression →

Prefix ⇒ (polish Notation)

Prefix

Infix ⟨operator⟩ ⟨operand⟩ ⟨operand⟩

5+1 ⇒ +51

Infix → a * b + c ⇒ Prefix * ab + c

Check Precedence

* ab + c

+ * ab c

+ * abc

Ans

Postfix ⇒ (Reverse polish Notation)

⟨operand⟩ ⟨operand⟩ ⟨operator⟩

Infix 5+1 ⇒ 51+ postfix

a * b + c ⇒ ab * + c ⇒ ab * c +

postfix

Computer can read prefix & post more Expression easily

Human → Infix

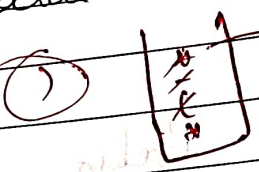
Infix to Postfix Conversion

① Print operands as they arrive.

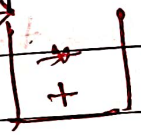
② If stack is empty or contains a left paranthesis on top, push the incoming operator onto the stack.

③ If incoming symbol is '(' push it onto stack. → left, open

④ If incoming symbol is ')', pop the stack & print the operators until left paranthesis is found. → right or close

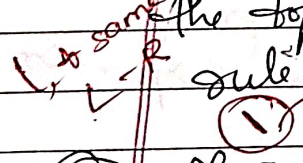


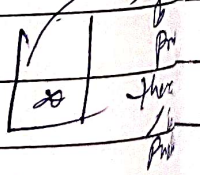
⑤ If incoming symbol has higher precedence than the top of the stack, push it on the stack.



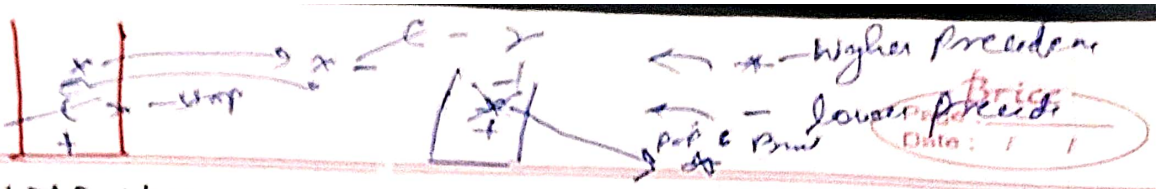
⑥ If incoming symbol has lower precedence than the top of the stack, pop & print the top. Then test the incoming operator against the new top of the stack.

⑦ If incoming operator has equal precedence with the top of the stack, use associativity rule.





⑧ At the end of the expression, pop & print all operators of stack.



Associativity

L-to-R — then pop & print the top of the stack & then push the incoming operator.

R-to-L — then push the incoming operator.

→ A + B / C

higher ↑

Input	stack	Exp.
A	-	A
+	+	A
B	+	AB
/	+ /	AB
C	+ /	ABC
	+	ABC/
	-	ABC/+

Algorithm

1) Examine the next element in the Input exp.

2) If it is an operand then output it.

3) If it is opening parenthesis push it on stack.

4) If it is an operator then

(a) If stack is empty push operator on stack & goto step 6.

(b) If the top of the stack is opening parenthesis push operator on stack & go to step 6.

(c) If it has higher priority than the top of the stack, push operator on stack & go to step 6.

(d) Else pop the operator from the stack & output it, repeat step 4.

Brice
Page :
Date : / /

5) If it is a closing parenthesis, pop operator from the stack and output them, until an opening parenthesis is encountered. pop & discard the opening parenthesis.

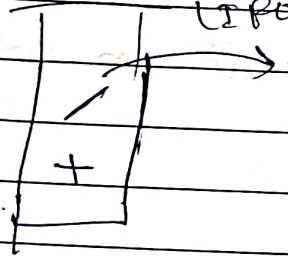
6) If there is more Input go to step 1.

7) If there is no Input unstack the remaining operator to output.

A + B / C

Input	Stack	Postfix
A	-	A
+	+	AB
B	-	AB
/	+/	AB
C	+/	ABC
	+	ABC/+

Stack



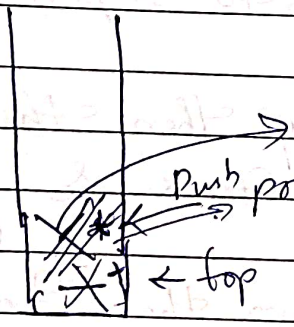
Postfix / Exp

ABC/+

A - B / C * D + E

Input	Stack	Postfix
A		A
-	-	A
B		AB
/	-/	AB
C		ABC
*	-*	ABC/
D		ABC/D
+	+	ABC/D*-
E	+	ABC/D*-E+

Stack



Postfix exp.

ABC/D*-E+

POP /

/ → higher prec.

*, / same precedence

Associativity & L R

+ lower than *

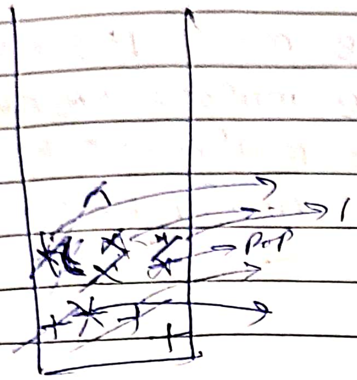
+, - same

Brice
Page:
Date: / /

K+L−M×N+(O^P)×W/U/V×T+Q

Stack Postfix Exp

K L+MN×−OP^W×U/V/T×+Q+



−,+
L·R
×,/
L·R
/,/
×,/
×,+
Pop ×
then compare
new top

A−B+(M^N)×(O+P)−Q/R^S×T+Z

Stack Postfix

× + higher preced. AB−MN^OP+×−Q
(− × Pop −
+

Input	Stack	Postfix
A		A
−	−	A
B	−	AB
+	+	AB−
(	(	
M		AB−M
^	(^	AB−M
N	(^	AB−MN
)		AB−MN^
×	×	AB−MN^
(	×(	AB−MN^O
O		
+	×(+	AB−MN^OP
P		
)	×	AB−MN^OP+
−	−	AB−MN^OP+×
Q	−	AB−MN^OP+×Q

/	- /	AB-MN^OP+*Q
R	- /	AB-MN^OP+*QR
A	- / ^	AB-MN^OP+*QR^
S	- / ^	AB-MN^OP+*QRS^
*	- *	AB-MN^OP+*QRS^/
T	- *	AB-MN^OP+*QRS^/T
+	+	AB-MN^OP+*QRS^/T*-
Z	+	AB-MN^OP+*QRS^/T*-Z+

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

Input	Stack	Exp	Comment
K		K	Simply Print 'K'
+	+	K	Push in stack
L	+	KL	Print 'L'
-	-	KL+	+,- same precedence Pop + & Push - in stack & print
M	-	KL+M	print M
	-*	KL+M	higher precedence & push on stack
	-*	KL+MN	Print N

Brice
Page :
Date : / /

	Stack	Expression		Comment
K	+	~~KL+MN*~~	KL+MN*-	Print * & (+, -) same
L	+(	KL+MN*-		Push '(' on stack
+	+(	KL+MN*-(		print +.
M	+(	KL+MN*-(+		Push on stack.
N	+(^	KL+MN*-(+		
*	+(^	KL+MN*-(+		Pop & Print '^'
)	+	KL+MN*-(+^		Push * on stack
*	+*	KL+MN*-(+^		print W
W	+*	KL+MN*-(+^W		Print ~~Push~~ * & push /
/	+/	KL+MN*-(+^W*		Print V
V	+/	KL+MN*-(+^W*V		Print '/' & Push end ')'
/	+/	KL+MN*-(+^W*V/		Print V
V	+/	KL+MN*-(+^W*V/V		Print ~~Push~~ / & Push *
*	+*	KL+MN*-(+^W*V/V/		Print T
T	+*	KL+MN*-(+^W*V/V/T		Print ~~Push~~ * & + & Push +
+	+	KL+MN*-(+^W*V/V/T*		
Q	+	KL+MN*-(+^W*V/V/T*+Q		
	Postfix exp.	KL+MN*-(+^W*V/V/T*+Q+		

Date : / /

Stack

→ A stack is a simple data structure used for storing data.

→ A stack is an ordered list in which insertion & deletion are done at one end that is called top.

→ The last element inserted is the first one to be deleted. Hence it is called "last-in-first-out". (LIFO)
or
"First-in-last-out" (FILO)

→ Stack is the one of the most important & useful non-primitive ds in Computer science.

→ It is a homogeneous collection of item arranged linearly with access at one end only. That is called top of stack.

→ Top is the point of stack where the elements are addressed added or removed.

→ Top of the stack is the part of the stack from where push & pop operations are performed.

→ Stack is a linear ds that supports LIFO + philosophy.

→ Stack is abstract data type because we define operation part of stack well but implement

Brice
Page :
Date : / /

Past is not well defined

Stack Operations

Main Stack Operations

(a) push (data) — Insert data on to stack.
(b) pop — remove & return the last inserted element from the stack.

Auxillary Stack Operations

(a) int Top() - return the last inserted element without removing it.

(b) int size() :- Return the no. of element store in stack.

(c) int IsEmptyStack() - Indicate whether any element are stored in stack or not (check underflow condition)
→ pop() is not possible.

(d) int IsFullStack() - Indicate wether stack is full or not (overflow condition)
→ push() is not possible

Algorithm for Push

push (stack[MAX], ITEM)

1. [check overflow condition]
   If TOP = MAX then
      a) write overflow
      b) exit
2. [Increment in top by one]
   Set TOP = TOP + 1
3. [Add item at current position of TOP]
   Set stack [TOP] := ITEM
4. EXIT

Algo for Pop

pop (stack[MAX])

1. [Check Underflow condition]
   If Top = NULL then
      a) write underflow
      b) exit
2. [Remove current top element of stack]
   set ITEM = Stack [TOP]
3. [Decrement in Top by one]
   Set Top := Top - 1;
4. Return Item.

Brice
Page :
Date : / /

Traversal of stack

Stacktraversal (stack [max])
1. check Underflow
   if Top = NULL Then
   a) write : underflow
   b) exit
2. [Initialize a temp Variable]
   Set I := Top
3. Repeat 4 & 5 while I > 0
4. print element
   process : Stack [i]
5. [Decreement i by one]
   Set i = i - 1
6. exit

① Reverse the expression
② If operand directly print
③ If higher precedence than top simply push
④ If same precedence
⑧ If lower precedence pop out & ... with ...

L-R → simply put in the stack

Infix to Prefix Expression

⑤ closing parenthesis simply put in stack
⑥ any operator come after closing parenthesis simply push in the stack

A + B * C → * higher solve B*C
* BC
+ A * BC

⑦ opening parenthesis pop till closing

Input/Exp	Stack	Prefix	
Q		Q	directly print Q
+	+	Q	
T	+	QT	check higher precedence of incoming operator
*	+*	QT	* > + simply push
V	+*	QTV	
/	+*/	QTV	*, / same Associativity L-R simply push
U	+*/	QTVU	simply push in Prefix
/	+*//	QTVU	
W	+*//	QTVUW	print W
*	+*//*	QTVUW	simply push in stack
)	+*//*)	QTVUW	(closing) simply put in stack
P	+*//*)	QTVUWP	print P
^	+*//*)^	QTVUWP	any operator) simply push
O	+*//*)^	QTVUWPO	
(	+*//*	QTUVWPO^	if any opening parenthesis pop out till closing
+	+	QTUVWPO^*//*	top * higher than incoming + so pop out * from stack
N	++	QTUVWPO^*//*N	
*	++*	QTUVWPO^*//*N	
M	++*	QTUVWPO^*//*NM	
-	++-	QTV — — — M*	
L	++-	QT — — M*L	+ + same
+	++-+		L-R
K	++-+	QT — — M*LK	simply push

+5 * 4 6 ← | 4 | 5 | 6 | 24   4 * 6 = 24
5 + 24 = 29

* + 6 7 - 3 1
= 30

QTVWPO^*//*Q·N·M*LKT-++   Pop out the stack till stack is Empty

Convert ⇓

++--*KL*MN*//*^OPWVTQ

Evaluation of prefix & postfix expression

(prefix: scan R to L; postfix: scan L to R)

Infix: a+b*c - d/e ^f   a=2, b=3, c=4, d=16, e=2, f=3

Prefix: -+a*bc/d^ef

-+2*34/16^23   ← R to L scan.

→ Scan from R to L
→ find out first operator, find two operand.
<operator> <operand> <operand>
^ 2 3

2 ^ 3 = 2^3 = 8

-+2*34/16 8

/ 16 8   Immediate two.
16/8 = 2

-+2*34 2

3*4. = 12

-+2 12 2

2+12 = 14

Expression many times so time taken

= 14 - 2

= 14 - 2

= 12

without stack

With stack (Scan only once)

- + 2 * 3 4 | 16 ^ 2 3

1) start scanning
2) If find operand simply push into stack
3) operator ⇒ pop out 2 values from stack. (top & next top)
4) after evaluation put in stack
5) end of exp. only value in stack & it valid prefix
6) return the top it is the ans

op1 operator op2

2 * 3 = 8

op1 = 16
op2 = 8
operator = /

16 / 8 = 2

op1 = 3
op2 = 4
Operator = *

3 * 4
= 12

op1 = 2
op2 = 12
operator +

12 + 2
= 14

op1 = 14 | 14 - 2
op2 = 2 | = 12
Operator = -

Algorithm

Scan Prefix ~~to~~ expr. from right to left
for each char in prefix exp.

do
if operand is there, push it onto stack
else if operator is there, pop2 elements
op1 = top element
op2 = next to top element
result = op1 operator op2
push result onto stack
return stack [top]

Evaluation of postfix

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

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

→ 2 3 4 * + 16 2 3 ^ / -

<opnd> <opnd> <operator>

1) Scan from left to Right
2) find first operator
3) find 2 preceding, Immediate two operand.

without stack

op1 = 3 op 2 = 4. operater = *

3 * 4 = 12

2 12 + 16 2 3 ^ / -

→ - - -

2 12 +

2 + 12 = 14

14 16 2 3 ^ /-

2 3 ^

2 ^ 3 = 8

14 16 8 /-

16 / 8 = 2

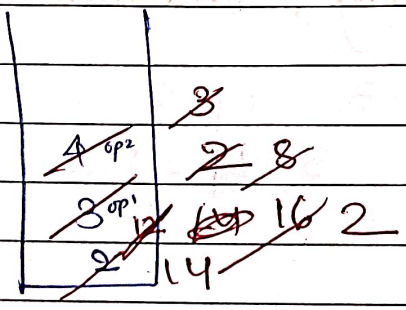
14 2 -

14 - 2

= 12

Using Stack

1. Start scan from left to right
2. push operand in stack until operator is found.



when popout
top → op2 & next to top is op1

op2 = 4

op1 = 3

op1 op op2

3 * 4 = 12

op2 = 12 op1 = 2 op = +

12 + 2 = 14

op2 = 16 op1 = 14

op2 = 8 op1 = 2 op = ^

2 ^ 3 = 8

op2 = 8 op1 = 16 op = /

op1 op op2

16 / 8 = 2

op2 = 2 op1 = 14 op = -

op1 = op1 - op2

14 - 2 = 12

return the top of the stack.

Algo for Evaluation of postfix exp

Begin:

for each char in postfix expr, 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,

Queue