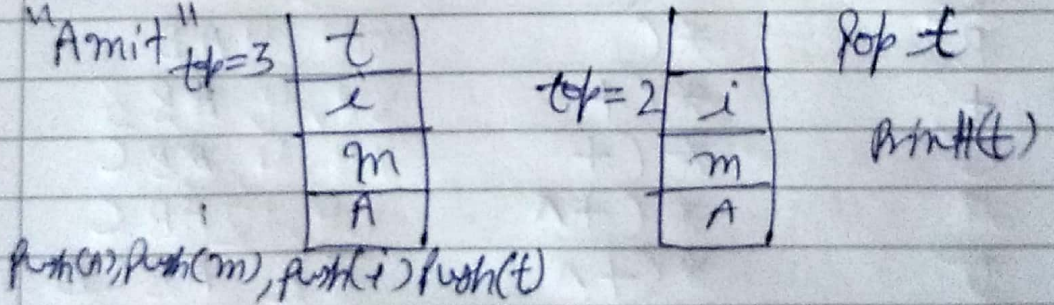
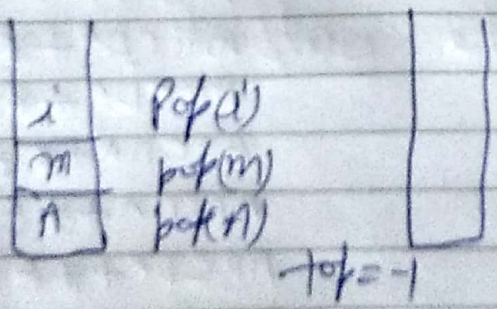
~~Priority queue~~

Application of stack

1. Reversal of a string.
2. Checking validity of an expression containing nested parentheses.
3. Conversion of infix expression to postfix and prefix form.
4. Evaluation of postfix and prefix expression.
etc.

1. **Reverse of string** — We can reverse of string by using stack. For reverse of string first we insert (push) each character in the stack and then pop each character one by one from the stack.





string is "timA"

2. Checking validity of an expression.
An expression will be valid if it satisfies these two conditions.

1) The total number of left parentheses should be equal to the total number of right parentheses in the expression.
2. For every right parenthesis there should be a left parenthesis of the same type.

Ex 1. [19-5*(9+4) Invalid
2. (1+5} "
3. [5+4*(9-2)] valid
4. [2+(4*2)-{6%4}] valid.

Conversion infix to postfix expression

Let expression is A*(B+C^D)-E^F*(G/H)#

Step	Symbol	operator in stack	Postfix expression
1	A		A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	+	*(+	AB
6	C	*(+	ABC
7	^	*(+^	ABC
8	D	*(+^	ABCD
9	)	*	ABCD^+
10	-	-	ABCD^+*
11	E	-	ABCD^+*E
12	^	-^	ABCD^+*E
13	F	-^	ABCD^+*EF
14	*	-*	ABCD^+*EF^
15	(	-*(	ABCD^+*EF^
16	G	-*(	ABCD^+*EF^G
17	/	-*(/	ABCD^+*EF^G
18	H	-*(/	ABCD^+*EF^GH
19	)	-*	ABCD^+*EF^GH/
20			ABCD^+*EF^GH/*-

__/__/____

Ex-2 Convert infix expression to postfix expression

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

Step.	Input exp.	stack	Postfix expression.
1	K		K
2	+	+	K
3	L	+	KL
4	-	-	KL+
5	M	-	KL+M
6	*	-*	KL+M
7	N	-*	KL+MN
8	+	+	KL+MN*-
9	(	+(	KL+MN*-
10	O	+(	KL+MN*-O
11	^	+(^	KL+MN*-O
12	P	+(^	KL+MN*-OP
13	)	+	KL+MN*-OP^
14	*	+*	KL+MN*-OP^
15	W	+*	KL+MN*-OP^W
16	/	+/	KL+MN*-OP^W*
17	U	+/	KL+MN*-OP^W*U
18	/	+/	KL+MN*-OP^W*U/
19	V	+/	KL+MN*-OP^W*U/V
20	*	+*	KL+MN*-OP^W*U/V/
21	T	+*	KL+MN*-OP^W*U/V/T
22	+	+	KL+MN*-OP^W*U/V/T*+
23	Q	+	KL+MN*-OP^W*U/V/T*+Q
24			KL+MN*-OP^W*U/V/T*+Q+

So the final Postfix expression is

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

Evaluation of postfix expression.

Ex-1 Postfix expression is

4, 5, 4, 2, ^, +, *, 2, 2, ^, 9, 3, /, *, −

Step	Symbol	operator in stack.
1.	4	4
2.	5	4, 5
3.	4	4, 5, 4
4.	2	4, 5, 4, 2
5.	^	4, 5, 16
6.	+	4, 21
7.	*	84
8.	2	84, 2
9.	2	84, 2, 2
10.	^	84, 4
11.	9	84, 4, 9
12.	3	84, 4, 9, 3
13.	/	84, 4, 3
14.	*	84, 12
15	−	72

So 72 is the output.

Abstract Data Type (ADT)

ADT is a mathematical model of the data objects that make up a datatype as well as the functions that operate on these objects. It describes the way components are related to each other and the operations that can be performed on the datatype.

ADT is a mathematical definition of objects, with operations ⇒ defined on them ⇒ It consists three parts.

1. Data — Describes the structure of data used in ADT.
2. Operations } Describes valid operations on the data.
3. Describes how to deal with errors that occurs.

Ex1 Stack ADT

1. Data or (objects) —
   ⇒ A finite sequence of nodes or elements. It follow 'LIFO' properties.
2. Operations }
   (1) Push() — For insert element on top.
   (2) Pop() — Delete ~~operation for~~ element from top.
   (3) Top() — Remove and return top element.
3. Errors —

The Definition of stack with fixed data and operations is called Stack ADT.

<Queue ADT>

① Queue ADT store arbitrary objects.
Insertions and deletions follow the (FIFO) first in first out properties.
Insertion at rear end ~~and~~ of the que and deletion from the front.

② Operation } ① Insert (object o)
② Delete (object o)
③ Display ()

③ Exceptions }
Empty Queue Exception
etc.