DATE

Data Structure

Data Structure is a programming construct for respon representing or managing collection of data. It is a method of storting data for the purpose of efficient computation. It provides an organizati organizational scheme that shows the relationship among the individual elements and facilities efficient data manipulations.

Abtract Data Type (ADT)

ADT is a useful tool for specifying the logic properties of a data type It is a mathematical m with a collections of operations defined o the models by specifying the mathema and logical properties of a data structu a ADT is a useful guideline to implem & a useful tool for a programmer who wish to use data structure correctly.

Basic Operations

1. Insertion
2. Deletion
3. Searching
4. Sorting
5. Traversing
6. Updata.

Algorithm

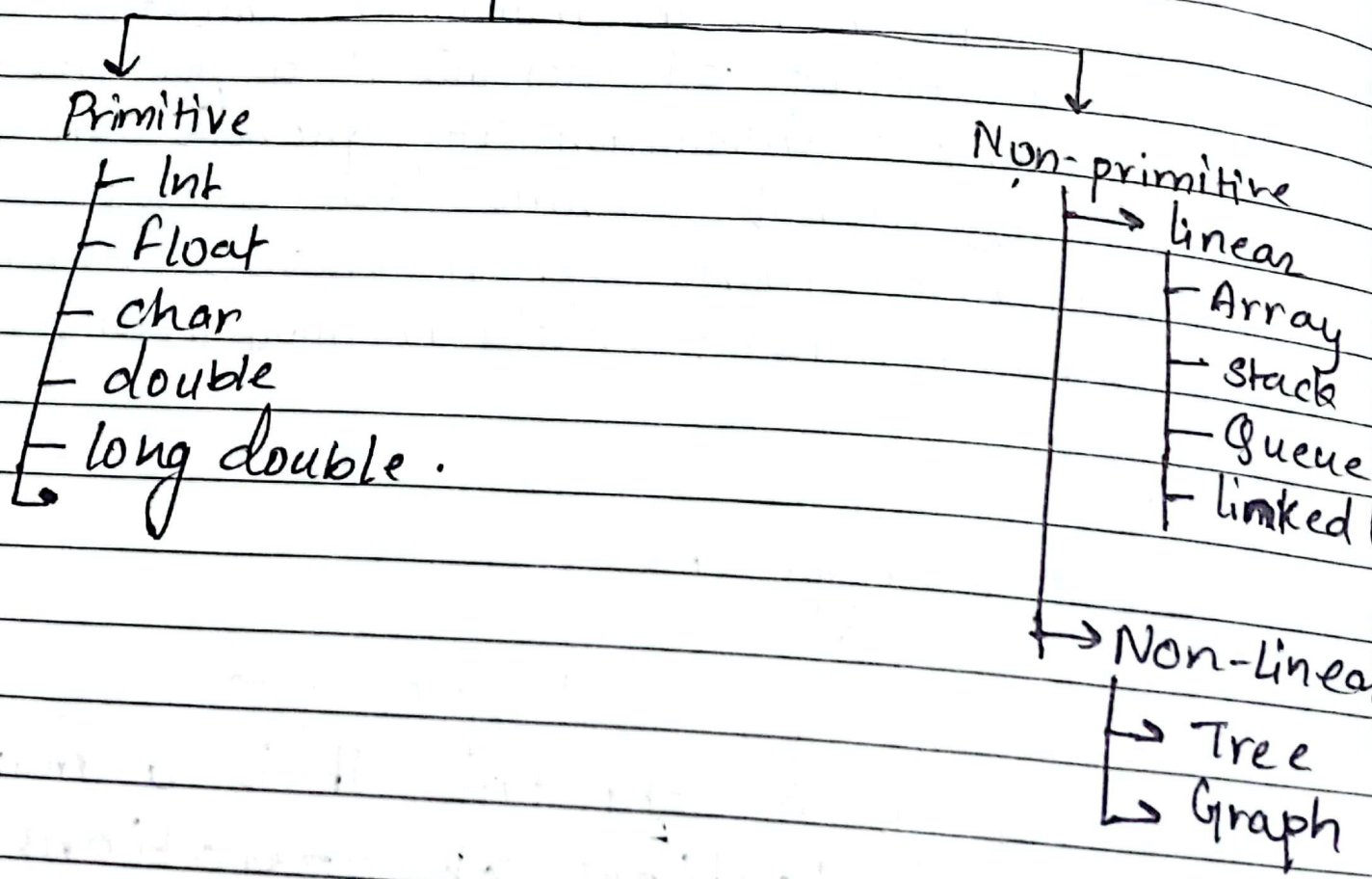
efficiency
- Time comple
- Space.

CLASSMATE

PAGE

DATE

Types of data Structure



Chapter 2

Stack data structure (FILO)

Linear data structure based on the principle of first in last out because of having a single end which is known as top of the stack (tos)

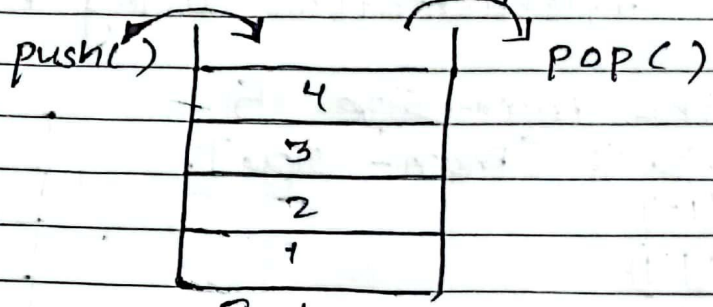


Fig: stack

Basic Operation of stack

Push(): Adding new element

POP(): Removing element from stack

IsFull(): True if stack is full

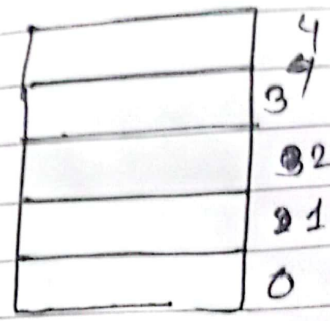
IsEmpty(): True if stack is empty

Stack Implementation

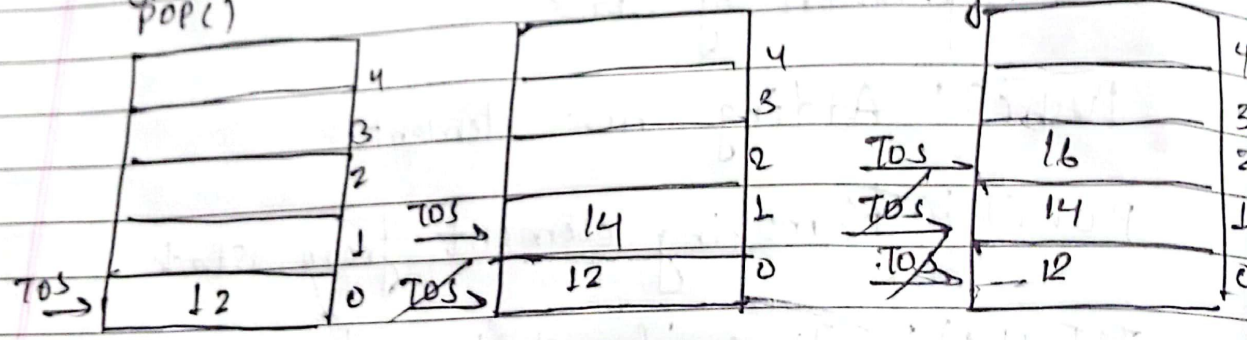
1. Static Implementation using array
2. Dynamic implementation using linked list

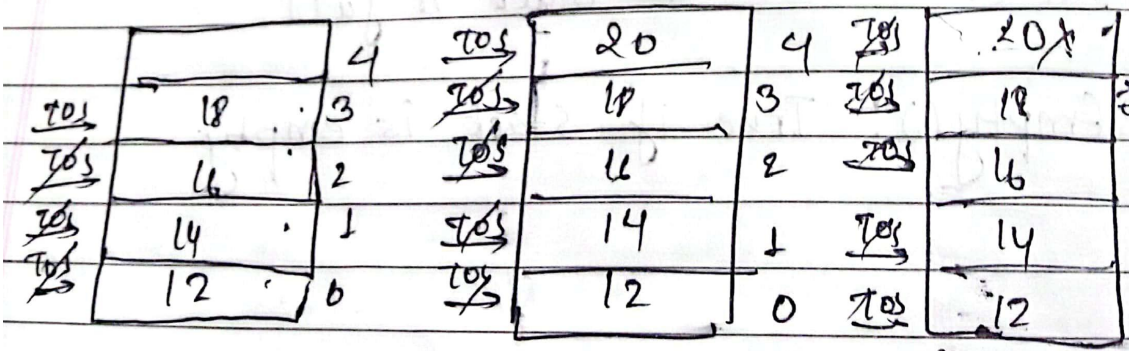
1. Static Implementation using Array.

```
#define max-size 5
Int stack [max-size].
push (12)
push (14)
push (16)
Stack [tos] = 12.
pop()
```



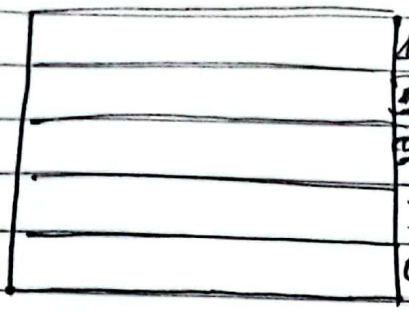
1st Fig.





#1 define maxsize 5

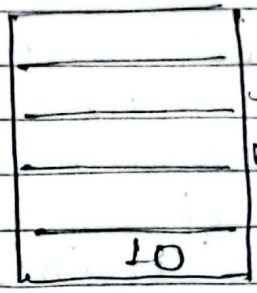
DATE



push(10)

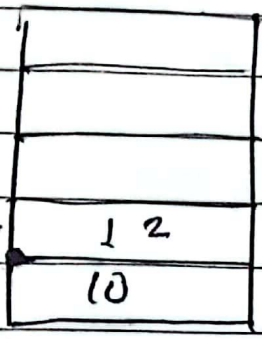
if (tos = max-size-1)

F

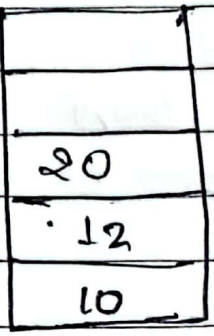


TOS = -1

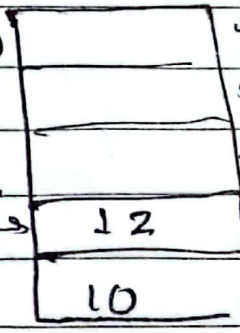
n(12)



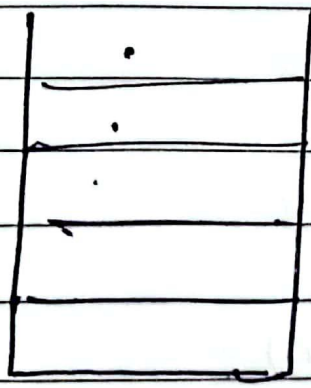
push(20).



pop()



p()



TOS = -1

Push (item, tos)

1. If (tos = max-size - 1) Then
2. Display "stack is full)"
3. Return
4. End If
5. Stack [++ tos] ← item

Pop (tos)

1. If tos = -1 Then
2. Display "Stack is empty"
3. Return
4. End If
5. item ← stack [tos]
6. tos ← tos - 1
7. Return item

item = stack [tos--]

Types of Expression.

1. Infix : a+b
2. Pre fix : +ab (polish notation)
3. post fix : ab+ (Reverse polish notation)

DATE

Manually:-

Precedence & Associativity.

1.	[ ], ( ), { }	~~R → L~~
2.	^ or $	R → L
3.	*, /	L → R
4.	+, −	L → R

a + b − d * (e + f) ^ g / h

Conval- to prefix & post fix

a + b − d * (+ e f) ^ g / h

a + b − d * p ^ g / h where p = + e p

a + b − d * (^ p g) / h

a + b − d * q / h where q = ^ p g

a + b − (* d q) / h

a + b − r / h where r = * d q

a + b − (/ r h)

a + b − s where s = / r h

(+ a b) − s

t − s where t = + a b

− + s

- ts
- + abs
- + ab/rh
- + ab/*dgh
- + ab/*d^pgh
- + ab/*d^efgh.

a+b-d*(e+f)^g/h
Prefix to Postfix
a+b-d*(ef+)^g/h
a+b-d*P^g/h where P = ef+
a+b-d*(Pg^)/h
a+b-d*Q/h where Q = Pg^
a+b-(dQ*)/h
a+b-R/h where R = dQ*
a+b-(/Rh)
a+b-S where S = /Rh
(ab+)-S
T-S where T = ab+
TS-

ab+S-
ab+Rh/-
ab+dQ*h/-
ab+dPg^*bh/-
ab+def+g^*h/-

Postfix

a - (b * d) - (e * f / g) * (h / k ^ j)

Postfix

= a - (bd*) - ((ef*)/g) * (h / (kj^))
= a - bd* P - (Q/g) * (h/R)
= a - P - (Qg/) * (hR/)
= a - P - S * T
= a - P - (ST*)
= a - P - U
= (aP-) - U
= V - U
= VU-
= aP-ST* aP-QST*-
= abd*-Qg/hR*-
= abd*-ef*g/hkj^/*-

P = bd*
Q = ef*
R = kj^
S = Qg/
T = hR/
U = ST*
V = aP-

aP-ST*-
abd*-Qg/hR/*-
abd*-ef*g/hkj^/*-

Infix to Postfix using stacks.

1. Start.
2. Declare all necessary variables eg: Infix [ ], postfix [ ], tos = = - - 1.
3. Scan the expression left to right
   3.1 If character is operand, insert in to postfix [ ] string
   3.2 Else if the character is left parenthesis push on to the stack.
   3.3 Else if character is right parenthesis
   * Repeatedly pop everything from stacks and add to postfix string until opening parenthesis doesn't come and discard it.
   3.4 Else if currently scanned character is an operator.
   * check the character at top of the stack lets say 'y'.

3.4.1 If stack is empty or there is only left paenthens push 'x' to stack

else

* Pop stack content
* check the procedure of 'y' one by one with character 'x' if procedence

DATE

of x is greater than 'y' push 'x' to the stack else if precedence of x is small or equals to 'y' pop y from stack and add to postfix string one by one. and push 'x' to post Stack

4. When end of the string is reached pop all the thing from stack and add to post fix [ ] string.
5. Display post fix string
6. Stop.

Convert

A * B + C * D to postfix

Scanned character	Stack content	Output string
A	empty	A
*	*	A
B	*	AB
+	+	AB*
C	+	AB*C
*	+*	AB*C
D	+*	AB*CD
Empty	Empty	AB*CD*+

(A+B)* (C-D/E

Scanned character	Stack content	Output string
(	Empty	Empty
A	(	A
+	(+	A
B	(+	AB
)	Empty	AB+
*	*	AB+
C	*	AB+C
-	-*	AB+C*
D	-*	AB+C*D
/	-/*	AB+C*D
E	-/*	AB+C*DE
Empty	Empty	AB+C*DE/-

DATE

1. Operand → postfix [ ]
~~2. Operand~~
2. Opening (⇒ stack
3. closing) ⇒ pop everything from stack add to postfix until we don't get the matching pair and discard both.
4. Operator → if stack empty or having opening parenthesis, push on to stack
5. Operator (X) (scan) and Operator (Y) (stack)

X is (heavier) → push that to stack ie X

X is (≤) → Y moves to postfix which need to be checked one by one & X comes to stack.

Convert

(A+B - H + (D * E / F)* G)

Scanned character	Stack content	Output string
(	(	Empty.
A	(	A
+	(+	A
B	(+	AB
-	(-	AB+
H	(-	AB+H
(	(-(	AB+H-
D	(-(	AB+H-D
*	(-(*	AB+H-D
E	(-(*	AB+H-DE
/	(-(/	AB+H-DE*
F	(-(/	AB+H-DE*F
)	(-	AB+H-DE*F/
*	(-*	AB+H-DE*F/
G	(-*	AB+H-DE*F/G
)	Empty	AB+H-DE*F/G*-

C	C-C	AB+H
D	C-C	AB+HD
*	C-C*	AB+HD
E	C-C*	AB+HDE
/	C-C/	AB+HDE*
F	C-C/	AB+HDE*F
)	C-	AB+HDE*F/
	C-+	AB+HDE*F/
G	C-*	AB+HDE*F/G*-

(A+B-H+(D*E/F)*G)

Scanned characters	Stack content	Output string
	(	(empty)
(	(	
A	(	A
+	(+	A
B	(+	AB
-	(-	AB+
H	(-	AB+H
+	(+	AB+H-
(	(+(	AB+H-
D	(+(	AB+H-D
*	(+(*	AB+H-D
E	(+(*	AB+H-DE
/	(+(/	AB+H-DE*
F	(+(/	AB+H-DE*F
)	(+	AB+H-DE*F/
*	(+*	AB+H-DE*F/
G	(+*	AB+H-DE*F/G
empty	empty	AB+H-DE*F/G*+

A + (B * C + D / E) - F * G / (H - I + J) / K

Scanned characters	Stack contents	Output String
A	Empty	A
+	+	A
C	+C	A
B	+ C	AB
*	+C*	AB
C	+C*	ABC
+	+C*+	ABC*
D	+C+	ABC*D
/	+C+/	ABC*D
E	+C+/	ABC*DE
)	+	ABC*DE/+
-	-	ABC*DE/++
F	-	ABC*DE/++F
*	-*	ABC*DE/++F
G	-*	ABC*DE/++FG
/	-/	ABC*DE/++FG*
C	-/C	ABC*DE/++FG*
H	-/C	ABC*DE/++FG*H
-	-/C-	ABC*DE/++FG*H
I	-/C-	ABC*DE/++FG*HI
+	-/C+	ABC*DE/++FG*HI-
J	-/C+	ABC*DE/++FG*HI-J
)	-/	ABC*DE/++FG*HI-J+
/	-/	ABC*DE/++FG*HI-J+/
K	-/	ABC*DE/++FG*HI-J+/K
Empty	Empty	ABC*DE/++FG*HI-J+/K/-

A+CB*C+D/E) - F*G/(H-I+J)/k

Scanned character	stack content	postfix string
A	empty	A
+	+	A
C	+C	A
B	+C	AB
*	+C*	AB
E	+C*	ABC
+	+C+	ABC*
D	+C+	ABC*D
/	+C+/	ABC*D
E	+C+/	ABC*DE
)	+	ABC*DE/+
-	-	ABC*DE/++
F	-	ABC*DE/++F
*	-*	ABC*DE/++F
G	-*	ABC*DE/++FG
/	-/	ABC*DE/++FG*
C	-/C	ABC*DE/++FG*
H	-/C	ABC*DE/++FG*H
-	-/C-	ABC*DE/++FG*H
I	-/C-	ABC*DE/++FG*HI
+	-/C+	ABC*DE/++FG*HI-
J	-/C+	ABC*DE/++FG*HI-J
)	-/	ABC*DE/++FG*HI-J+
/	-/	ABC*DE/++FG*HI-J+/
k	-/	ABC*DE/++FG*HI-J+/k
Empty	Empty	ABC*DE/++FG*HI-J+/k-

DATE

Algorithm to evaluate postfix expression.

1. Start
2. Scan postfix string from left to right.
3. If it is an operand push it onto the stack.
   else (if operator)
   * pop top two operand from stack
   i.e pop1 = pop( ); pop2 = pop( )
   * perform result = pop2 * pop1
   * push result back to stack
4. Repeat step 3 until postfix string is not finished.
5. Pop the result from stack and display it.
6. Stop.

A + (B * C) / D - E ^ F

A = 1, B = 2, C = 3 D = 2, E = 4 F = 2

Convert to postfix and evalute using stack.

= A + (BC*) / D - E ^ F

= A + P / D - E ^ F P = BC*

= A + P / D - (EF^)

= A + P / D - Q Q = EF^

= A + (PD/) - Q R = PD/

= A + R - Q

= (AR+) - Q S = AR+

= S - Q

= SQ -

= AR + Q -

= A PD/ + Q -

= A PD/ + EF^ -

= A BC * D/ + EF^ -

Subsituting

1 2 3 * 2 / + 4 2 ^ -

2
4
4

pop1 = 2 result
pop2 = 4 = pop2 ^ pop1
= 4 ^ 2 = 16

16
4

pop1 = 16
pop2 = 4
result = 4 - 16 = -12

-12

TOS

push(1) TOS → | 1 |

push(2) TOS → | 2 |
| 1 |

push(3) TOS → | 3 |
| 2 |
| 1 |

pop1 = 3
pop2 = 2
result = pop2 * pop1
result = 2 * 3 = 6

TOS → | 6 |
| 1 |

push(4)

TOS → | 2 |
| 6 |
| 1 |

pop1 = 4
pop2 = 6
result = pop2 / pop1 = 6/2

pop1 = 3
pop2 = 1
result = 1 + 3 = 4

4

push(4) TOS → | 4 |
| 4 |

push(2) → | 2 |
| 4 |
| 4 |

pop1 = 2 = 3
pop = 4

DATE

A * B + C / D - E * F

A = 5 B = 4 C = 6, D = 2 E = 7, F = 8

A * B + C / D - E * F

= (AB*) + C / D - E * F P = AB*

= P + (CD/) - E * F Q = CD/

= P + Q - (EF*) R = EF*

= P + Q - R

= (PQ+) - R S = PQ+

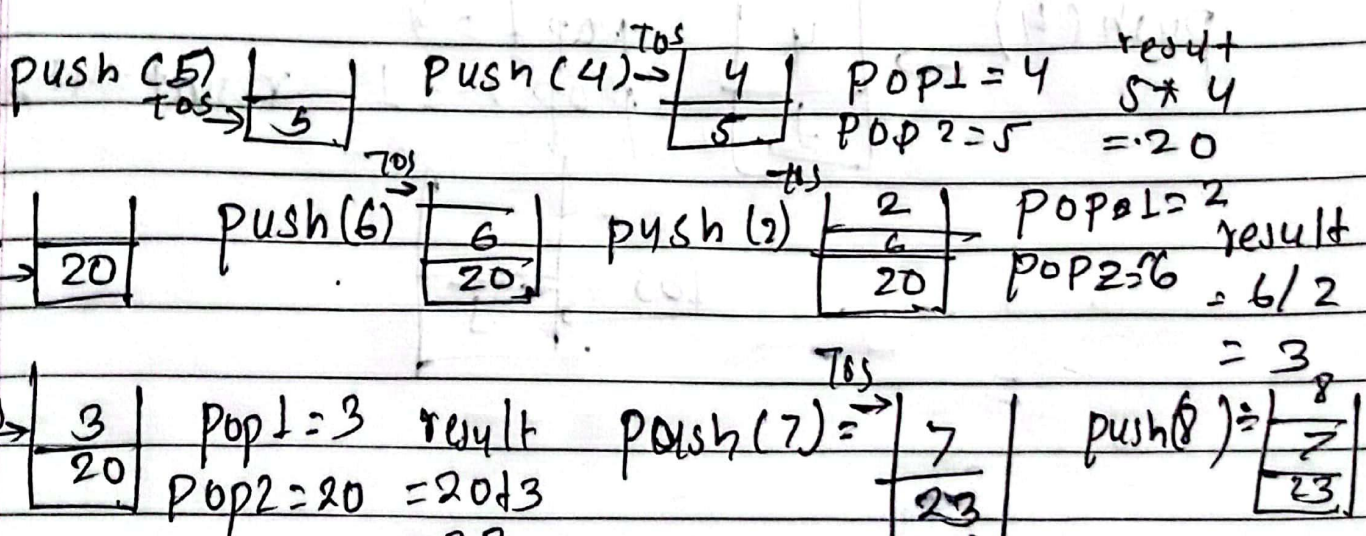
= S - R

= SR-

= PQ* + R-

= AB* CD/ + EF* -

5 4 * 6 2 / + 7 8 * -

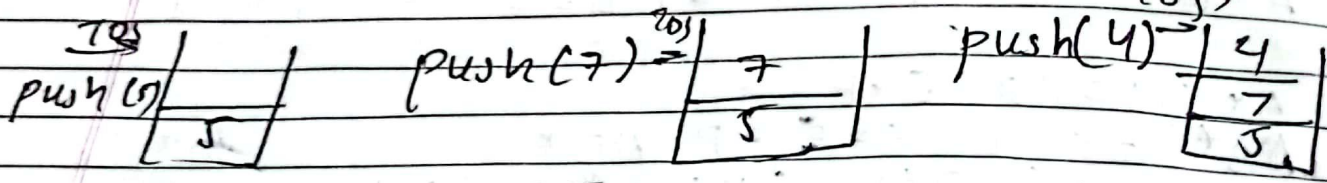


56
-23
33

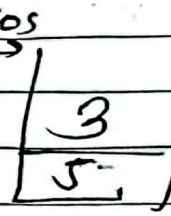
classmate

PAGE

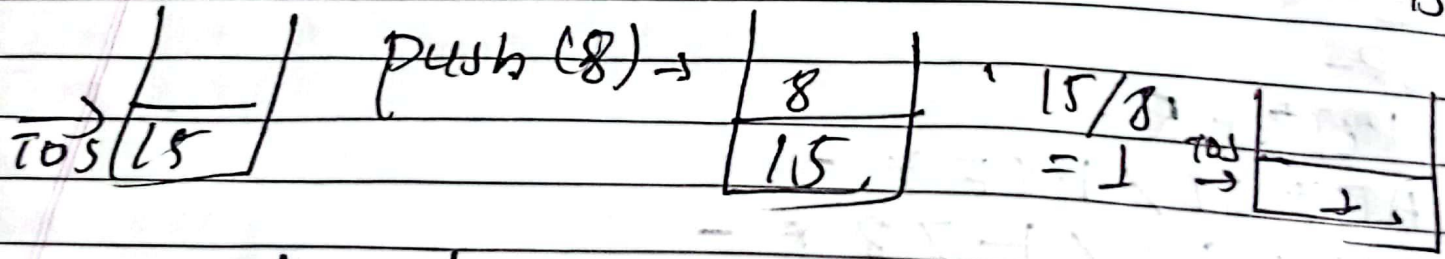
(2) 5 7 4 - * 8 / 4 +

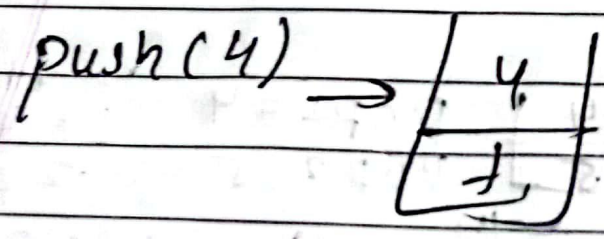


pop1 = 4 result = 7 - 4 = 3
pop2 = 7

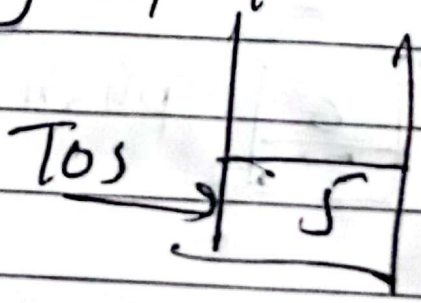


pop1 = 3
pop2 = 5
result = 5 * 3 = 15





pop1 = 4
pop2 = 1
result = 4 + 1 = 5



Algorithm to convert infix to prefix

1. Take the mirror image of given infix expression.
2. Scan from left to right.
   - *Assume openning as closing & vice versa
   - *If π is higher or equal push it to the stack else pop and add to postfix string one by one.
5. Take the mirror image of final expression

a + (b * c)

~~* a b c * +~~) c * b c + a

In ⇒ pre
(A + B * D)

Reverse the infix expression)D * B + A(

Scanned	Stack	Output
)	)	Empty
D	)	D
*	) *	D
B	) *	D B
+	) +	D B *
A	) +	D B * A
(	Empty	D B * A +
Empty	Empty	D B * A +

Infix: + A * B D

Reverse :- (A + B * C / D) + E * F - (G * H +

=) J - I + H * G (- F * E +) D / C * B -

Scanned	Stack	Outpu
)	)	Empt
J	)	J
	) -	

(A+B*C/D)+E*F-((G*H+I-J)

Reverse

)J-I+H*G(-F*E+)D/C*B+A(

Scanned chars	Stack Content	Output String
)	)	
J	)	J
-	)-	JI
I	)-	JI
+	)-+	JIH
H	)-+	JIH
*	)-+*	JIHG
G	)-+*	JIHG*+-
(	empty	JIHG*+-
-	-	JIHG*+-E
E	-	JIHG*+-F
*	-*	JIHG*+-FE
G	-*	JIHG*+-FE*
+	-+	JIHG*+-FE*D
)	-+)	JIHG*+-FE*D
D	-+)	JIHG*+-FE*DC
/	-+)/	JIHG*+-FE*DCB
C	-+)/	JIHG*+-FE*DCB
*	-+)/*	JIHG*+-FE*DCB
B	-+)/*	JIHG*+-FE*DCB*/
+	-+)+	JIHG*+-FE*DCB*/A
A	-+)+	JIHG*+-FE*DCB/A+-
(	-+	
	empty	

Regard:

(A+B * C / D) + E * F - (G * H + I - J)

Reverse

) J - I + H * G (- F * E +) D / C * B + A (

Scanned char	Stack Content	Output string
)	)	J
J	)	J
-	) -	JI
I	) -	JI
+	) - +	JIH
H	) - +	JIH
*	) - + *	JIHG
G	) - + *	JIHG*+-
(	Empty	JIHG*+-
-	-	JIHG*+-E
E	-	JIHG*+-FE
*	- *	JIHG*+-FE
F	- *	JIHG*+-FE*
+	- +	JIHG*+-FE*D
)	- +)	JIHG*+-FE*DC
D	- +)	JIHG*+-FE*DC
/	- +) /	JIHG*+-FE*DCB
C	- +) /	JCHG*+-FE*DCB
*	- +) / *	JIHG*+-FE*DCB
B	- +) / *	JIHG*+-FE*DCB*/
+	- +) +	JIHG*+-FE*DCB*/A+
A	- +) +	JIHG*+-FE*DCB/A+-
(	- +	
	Empty	

Regang.

DATE

Evaluation of prefix string:

* Scan the expression from right to left
* If operand push it on to the stack
* If operator pop top two operands pop 1 = pop(), pop 2 = pop and evaluate the result = pop1 ⊕ pop2 push the result back to stack.
* Final value on stack is the result.

A * B + C / D - E * F

Where

A=5, B=4, C=6, D=2, E=7, F=8.

(*AB) + C/D - E*F P=*AB

P + (/CD) - E*F Q = /CD

P + Q - (*EF) R = *EF

P + Q - R So ⇒

(+PQ) - R S = +PQ

S - R

- SR

= - +PQR

= - +*AB/CD*EF

= - +* 5 4/6 2 *78

DATE

push(8) TOS → | 8 |

push(7) TOS → | 7 | 8 |

pop1 = 7
pop2 = 8
result = 7*8 = 15
TOS → | 56 |

push(2) TOS → | 2 | 56 |

push 6 TOS → | 6 | 2 | 56 |

pop1 = 6
pop2 = 2

push = result = 6/2 = 3
TOS → | 3 | 56 |

push(4) TOS → | 4 | 3 | 56 |

push(5) TOS → | 5 | 4 | 3 | 56 |

pop1 = 5
pop2 = 4
result = 5*4 = 20

TOS → | 20 | 3 | 56 |

pop1 = 20
pop2 = 3
result = 20+3 = 23
TOS → | 23 | 56 |

pop1 = 23
pop2 = 56.
result = 23-56 = -33

TOS → | -33 |

DATE

Application of Stack

* To Evaluate mathmatical expressions
* Reverse the string
* Handling interrupt in computer System.
* Matching of nested parenthesis
* Browser history
* Undo sequence in text editor
* Evalution of recurcive calls.

Chapter 3 :

Queue

- linear data structure which follows the principle of first in first out (FIFO) as it has got two different ends to perform the insertion & deletion.

Types of Queue
1. Linear Queue
2. Circular Queue
3. Priority Queue
4. Double ended Queue (Deque)

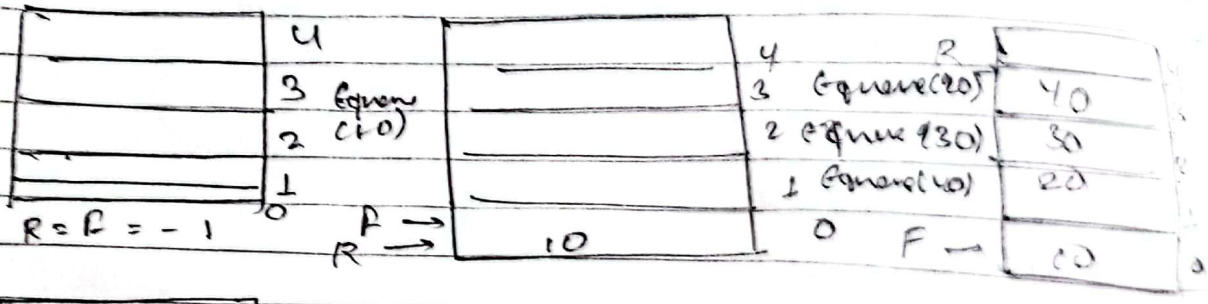
Queue basic Operations:

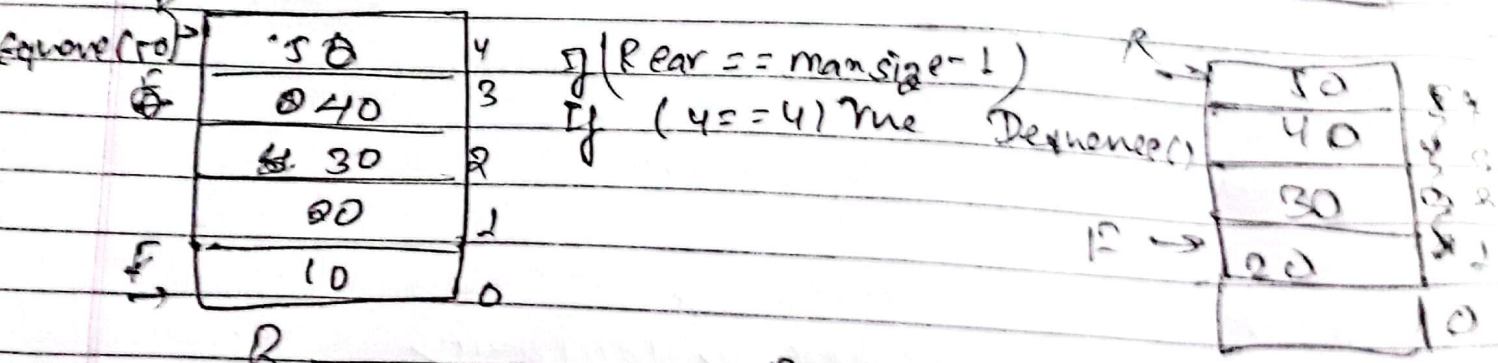
1) Enqueue() : Process of inserting elements
2) Dequeue() : Process of removing elements
3) Status check Operation : Empty and full condition
   * Is full() : Returns true if queue is full.
   * Is Empty() : Returns true if queue is empty.

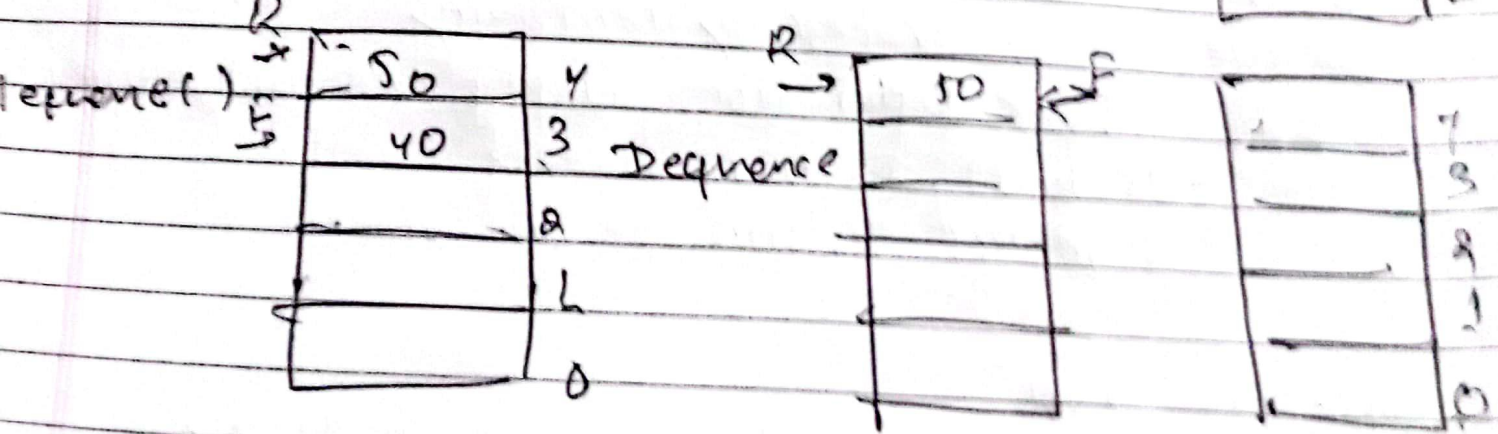
Implementation
1. Static Implementation : using array
2. Dynamic Implementation : using linked list

Static Implementation

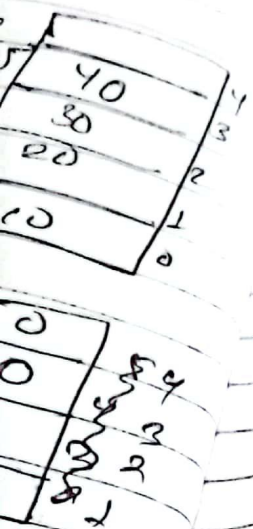
```
#define max-size 5
int queue [max-size], rear = front = -1
```







front = -1

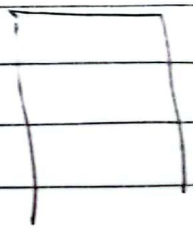


Enqueue (item, Rear, front)

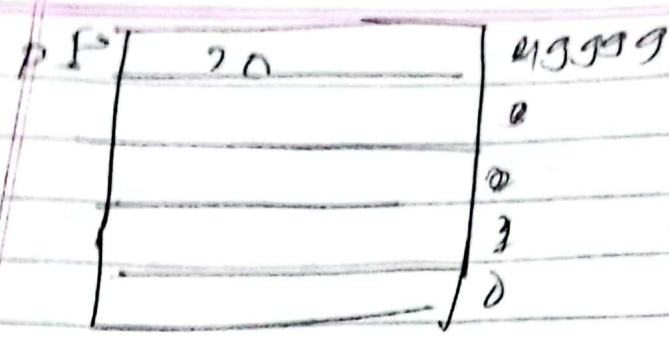
1. If (Rear = max-size - 1) Then
2. Display "Queue is full"
3. Return
4. End If
5. If Rear = front = -1 Then
6. Front = front + 1
7. End If
8. Rear ← Rear + 1
9. Queue [Rear] ← item

Dequeue (Rear, Front)

1. If (Rear = front = -1) Then
2. Display "Queue is empty"
3. Return
4. End If
5. ~~If (Rear = front)~~
5. item ← Queue (front)
6. If (Rear = Front)
   Rear ← Front ← -1
8. End if
   front ← front + 1
   Return item



DATE



Equeue (30) → Queue is full

if (Rear = maxsize-1)

if (9999 = 9999) true

Inefficient space utilization.

So in

1. Shifting after every dequeue
2. Implementing circular queue

Circular queue

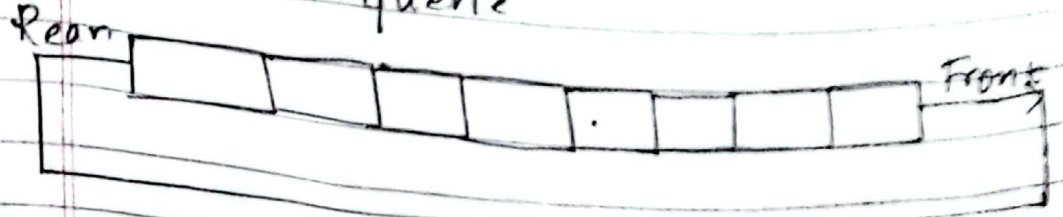


Fig:- circular queue

~~Algorithm~~

Enqueue (Q [max-size], Rear, Front)

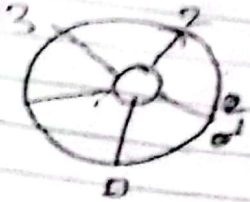
1. If (Rear + 1) % Max-size = front Then
2. Display " Queue is full "
3. Return
4. End If
5. Rear = front = -1 Then
6. Front = Front + 1
8. End If
9. Rear ← (Rear + 1) % max-size
10. Q [Rear] ← item

DATE

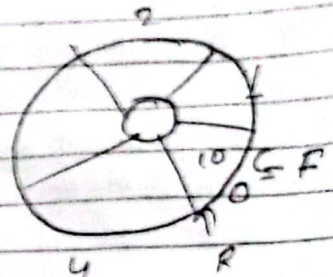
Dequene(Q [max-size], Rear, Front)

1. If Rear = Front = -1 Then
2. Display "Queue is empty".
3. Return
4. End If
5. item ← Q[Front]
6. If (Rear = Front) Then
   7. Rear ← Front ← -1
8. ELSE
   9. Front ← (Front + 1) % max-size
10. End If
11. Return Item

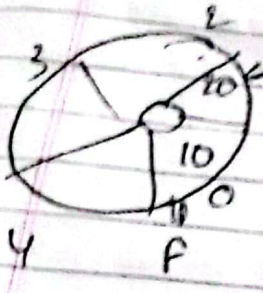
Enquence & Dequence

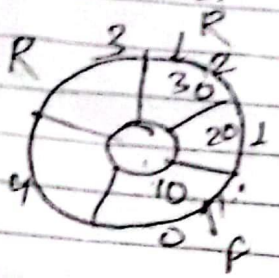


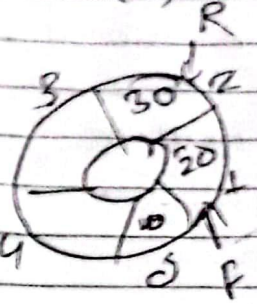
Enque (10)
Enque (10)
Enque (30)
2 Deque ( )

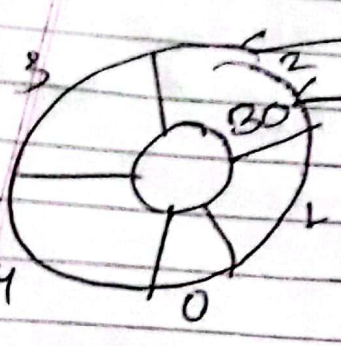


P ∈ F ∈ -1

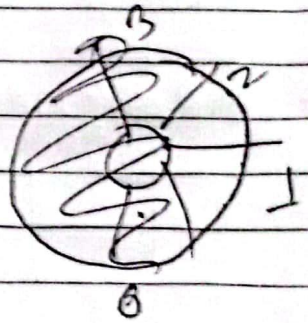


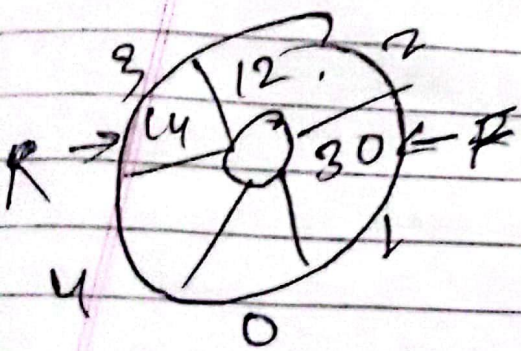


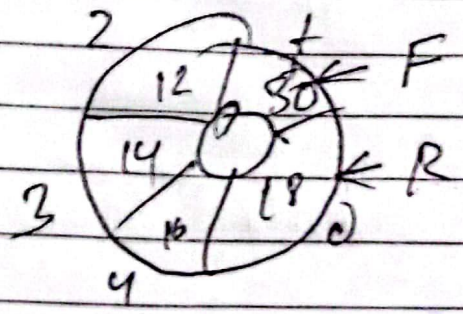




Enque (12)
Enque (14)
Enque (16)
Enque (18)







Enque ( ) → Queue is Full.

DATE

Priority Queue

A prority queue is another type of queue structure in which elements can be inserted or deleted based on priority. A priority queue is a data structure in which each item has been assigned a value called the priority of the element and an element can be inserted or deleted not only at ends but at any position of the queue.

Priority queue will follow the idea of FIFO if there are more than one element assigned with same priority number.

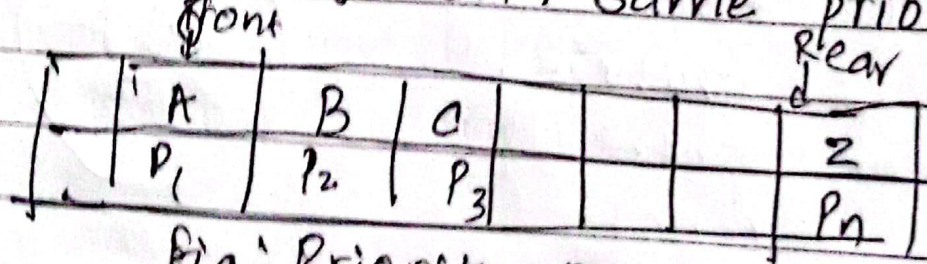


Fig: Priority Queue

Types

1. Ascending Priority Queue.
2. Descending Priority Queue.

14

DATE

1. Ascending Priority Queue

In Ascending priority queue elements can be inserted in any order. But while deleting elements from queue, always a small element only to be deleted first that comes the turn of elements having second smaller priority and so on.

2. Descending Priority Queue

In descending priority queue elements can arrive in the queue in any order but while removing it the element having largest priority number will be deleted from the queue.

3.

Double Ended Queue (Deque)

In double ended queue, both insertion and deletion operations are performed at either ends of queue. That is, we can insert a element from rear end or the front end. Also deletion is possible from either end.

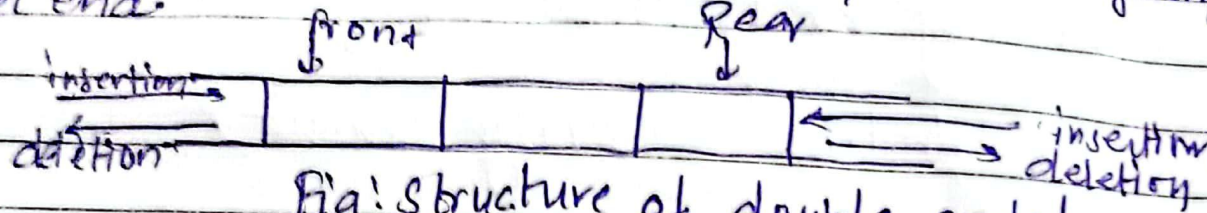


Fig: structure of double ended queue

Implementation
- Double linked list
- Circular Array

Types of Deque
1. Input restricted queue
2. Output restricted queue

In Input restricted deque, element can be added at only one end but we can delete the elements from both ends. An Output restricted deque where deletion takes place at only one end but allows insertion at both ends.

Fig: Input restricted deque

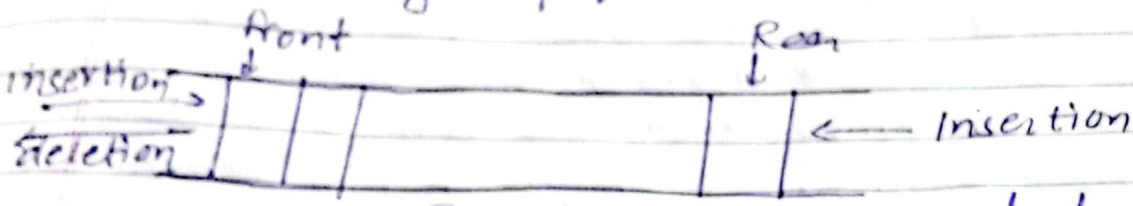


Fig: Output restricted deque

* Possible Operation
1. Add an element at the rear end
2. Add an element at the front end
3. Delete an element from the front end
4. Delete an element from the Rear end

* Application of Queue
1. Several algorithms that use queues to solve the problem such as Breadth first search tree traversing, kruskal and Prims.
2. Round Robin technique for processor scheduling.
3. When jobs are submitted or need to provide the service we need to implement FIFO for the queue is used.

Chapter : 5

Recursion (Recursive Problem)

Recursion is a technique that allows us to break down a problem into one or more sub-problems that are similar in form to the original problem. A fn which calls itself is known as recursive function and the process is recursion.

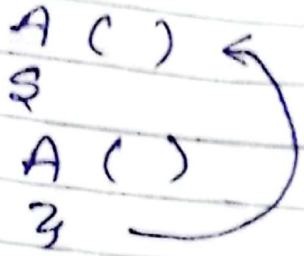
Recursion is useful for problems that can be represented by a simpler version of the same problem.

eg: $n! = n * (n-1)!$

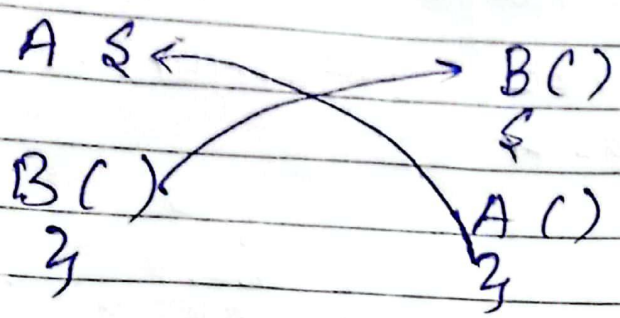
An algorithm is called recursive if it solves a problem by reducing it an instance of the same problem with smaller input.

Type of recursion

Direct Recursion

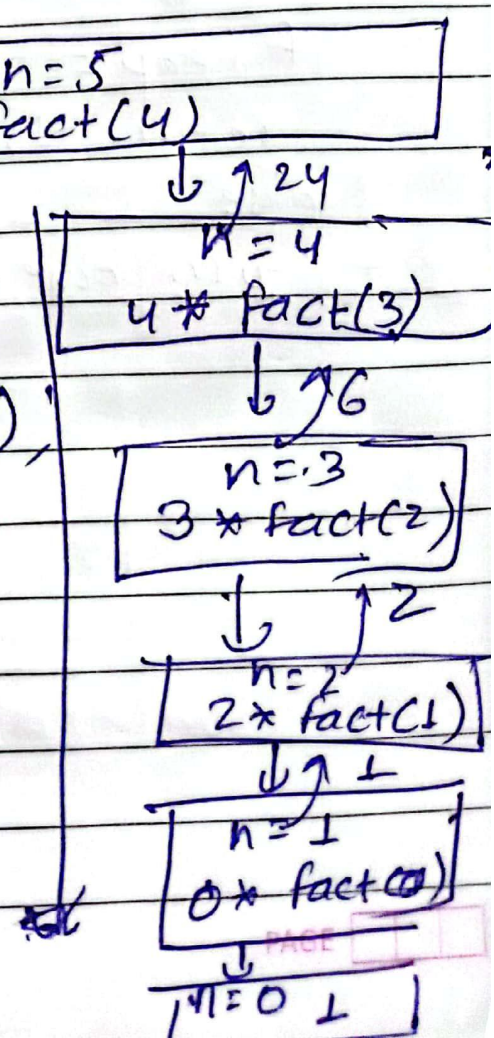


Indirect recursion




```
int fact (int n)
{
int (n=0)  return 1;   ← base
else return (n * fact (n-1));   ← recursive
}
```

Calling exact recursive order.



Principle of recursion

1. Recursive case:
   Reduce the problem in to smaller one every recursive call.

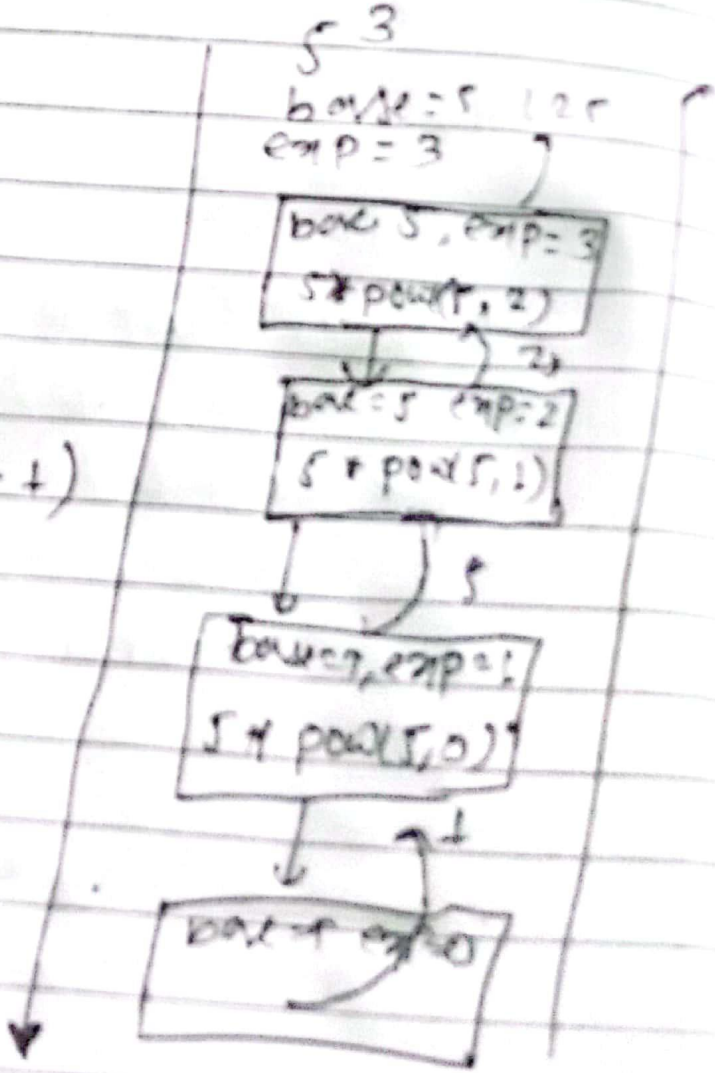
2. Base case:-
   Determine when recursive call is going to stop.

base case

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n * fact(n-1) & \text{else} \end{cases}$$

recursive case

```
pow(base,exp)
{
if (exp == 0)
return 1;
else
return base * pow(base, exp-1)
```



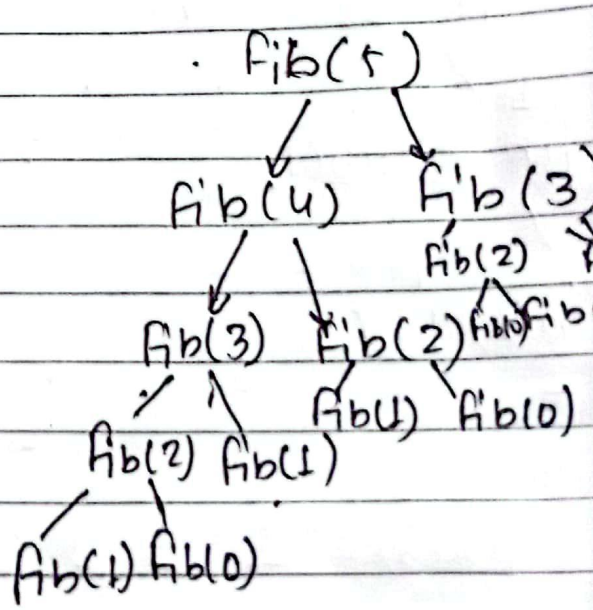
DATE

Fibonacci Sequence

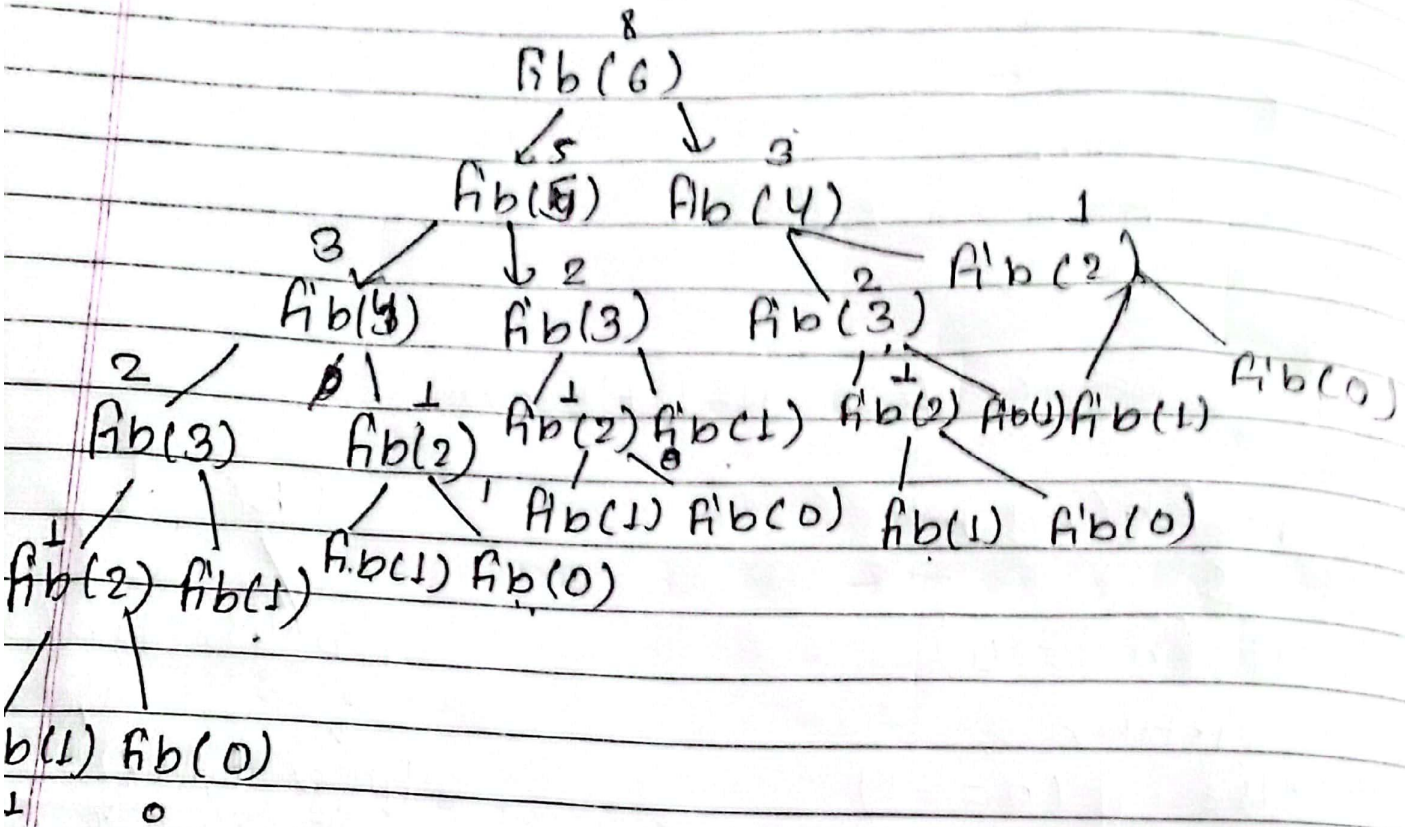
0 1 1 2 3
1 1 2 3 5

1. If (n == 0)
   return 0;
2. elseIF (n == 1) Then
   return 1;
3. else
   return(F.b(n-1) + fib(n-2))

```
int fib(int n)
{
if (n == 0)
  return 0;
else if (n == 1)
  return 1;
else
return (fib(n-1) + fib(n-2))
}
```

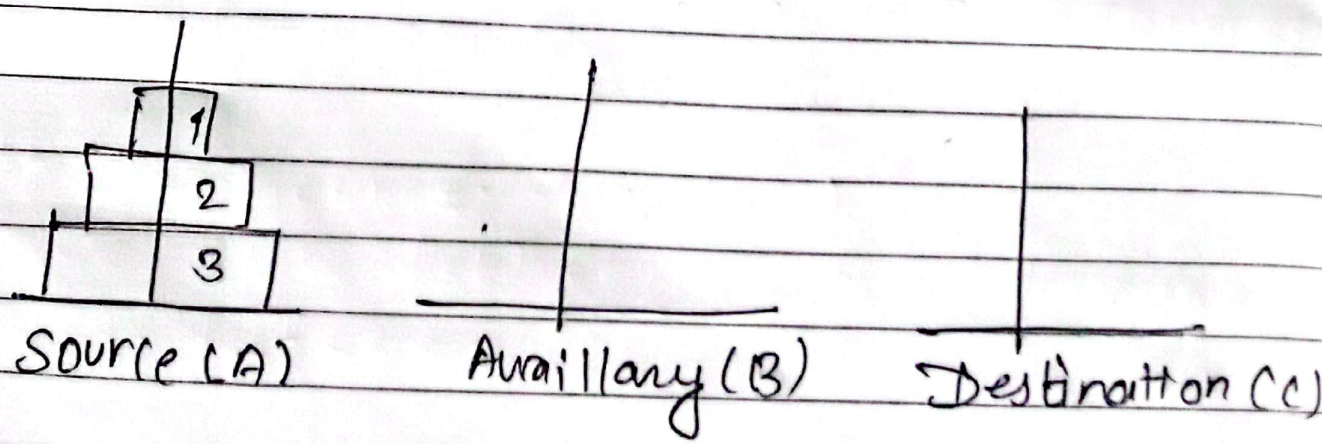


Fib (6)



Fib(6) = 1 1 2 3 5 8

Tower of Hanoi



Tower of Hanoi (TOH)

Problem defn
Given three different towers name d; source auxillary and destination Blocks are present in the source tower where the largest block (desk) is at the bottom and second largest on its top and so on. Our objective is to move disks from source to destination using auxillary tower.

Rules

1. One disk can be moved at a time
2. larger desk cannot be placed on top of the smaller.

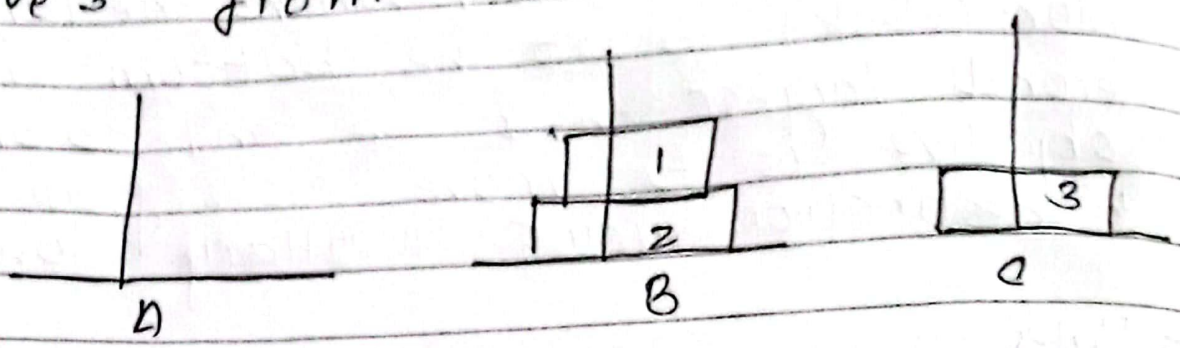
Steps required $= 2^n - 1$ where n = no. of disk

Algorithm

1. shift (n-1) disks from source (A), to Aux (B) using destination tower.
2. Shift one disk from source to destinatio
3. Shift (n-1) disk from ~~Auxiliary~~ Aux (B) to destination (C) using source (A) tower.

Steps:

1. Move 1 from A to C
2. Move 2 from A to B
3. Move 1 from C to B
4. Move 3 from A to C
5.



5. Move 1 from B to A
6. Move 2 from B to C
7. Move 1 from A to C

1 (algorithm), 2 (Algorithm), 3 (Algorithm)

```
void TOH(int n, char source, char chest, char aux)
{
if (n==1)
{
cout<<"Move disk."<<n<<"from"<<source<<
"to"<<dest<<endl;
}
Toh(n-1, source, aux, dest)
cout<<"Move disk"<<n<<"from "<<source<<
"to"<<dest<<endl;

Toh(n-1, aux, dest, source);
```

Q 2.

```
void main()
{
clrscr();
TOH(3,'A','C','B');
}
```

The tree is as :