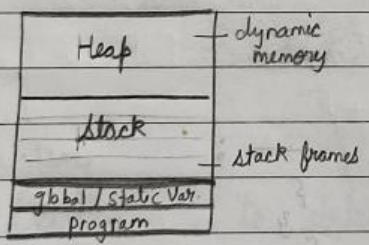
Data Structures & Algorithms

Dt. Page

Data Structures : Specialised format for organising data in main memory for efficient usage

- Data Structures
  - Primitive (single memory allocation)
    - Int
    - float
    - char
    - Boolean
  - Non-Primitive (group of data)
    - Linear (sequential)
      - Arrays
      - Stacks
      - Queues
      - Lists
        - Single LL
        - Double LL
        - Circular LL
    - Non Linear (Random)
      - Trees
      - Graphs

Main memory



- Heap memory is accessed using pointers in C & ref. var in java
- Memory used by var in stack retains till the end of function but memory in heap can be released as required
- Memory leak : too much memory in heap

ADT → DS
(implementation)

Abstract datatypes : (ADT) - Independent of any computer language. Blueprint for creating a DS

Set of data values and associated operations that are precisely specified independent of any particular implementation

eg : int

details are abstracted

We provide the ADTs & the user can simply use the functionalities without knowing the procedure

→ Data structures are the implementations of ADT

LIST, → Array List, Linked List

ADT DS

Spiral

Algos → implementation → program

Dt. Page

Algorithms: step by step procedure for solving a computational problem
finite set of steps to solve a problem

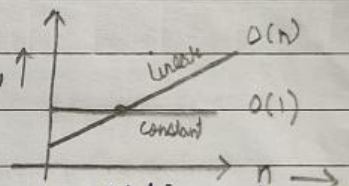
Characteristics of algorithms
- It must contain finite no. of steps.
- should contain unambiguous instructions
- It should be language independent
- It should have well-defined inputs & outputs
- It should be feasible with available resources.

logic should be correct

Analysis of Algorithms — Prior approx // Posterior exact (depends on hardware)

Based on two factors:

1) **Time complexity**:



- It is the amount of time taken for completion of an algorithm depending on input
- It is a theoretical estimate.
- Big-O notation → $O(n)$, $O(n^2)$

how time depends on input ← doesn't depend on n (whatever be the input time = constant)

simple statement : $O(1)$ —— ; const. time
loop (single) : $O(n)$ { } linear time
loop (nested once) : $O(n^2)$ { { } }

2) **Space complexity**

Amt. of space acquired by algo program to complete the task

fixed part : const etc independent
variable part : var. dependent

recursive $O(n)$ or factorial

$S(p) = C + Sp$
fixed var.

Spiral

* Some operations to be performed on DS

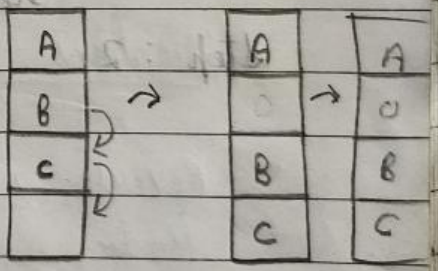
- **Traversing** : under this operation, each element is visited atleast once.

  eg: print, count, sum

  Traverse (Linear Array (LA), Lower bound (LB), Upper bound (UB))
  Step : 1 Repeat for K = LB to UB, inc by 1
  Apply the process to LA[K]
  step : 2 exit.

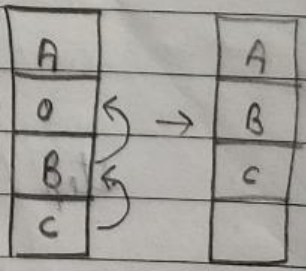
- **Insertion** (LA must have space)
  inserting an item at a particular place.

  eg: Insertion (LA, LB, UB, item, POS) → if 0 ≤ POS ≤ UB
  Step 1 : Repeat for K = UB to POS, dec by 1
  LA[K+1] = LA[K]
  end
  Step 2 : LA[POS] = item
  exit.



- **Deletion** : to delete an item

  eg: delete (LA, UB, K)
  Step 1 : Repeat for J = K To UB-1
  LA[J] = LA[J+1]
  end
  Step : 2 exit.



Codeforces → speed   Rajit Jain – linked list
Gourav Sain
Abdul   code n code
leet code   gfg placement series   simply algos

Dt.
Page

- Searching for searching an item

Linear search (A, item)
step :1 set location = -1
step :2 Repeat for k = 0 to length-1
step :3 if (A[k] = item)
location = k
break
step :4 print loc
exit

4

1	2	3	4	5
0	1	2	3	

ans = 3
for 5 – worst case
for 1 – best case

Binary search (A, item)

step : 1 beg = 0
end = length - 1
result = -1
step : 2 while (beg <= end)
{ mid = (beg + end)/2
if (A[mid] <= item)
{ beg = mid + 1
result = mid
}
else
end = mid - 1
}
step : 3 print result
end

5

1	2	3	4	5	6
0	1	2	3	4	5

- mid = (0+5)/2 = 2
A[2] = 3
beg = 3
result = 2

4	5	6
3	4	5

mid = (3+5)/2 = 4
A[4] = 5
beg = 5
result = 4

6

mid = (5+5)/2 = 5
A[5] = 6
end = 4
// end of loop
print = result = 4

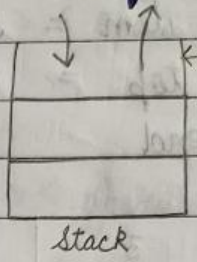
Real life scenario - disks stacked one above other

STACK (Linear data str.) ordered list

LIFO/FILO

only one type of data can be stored

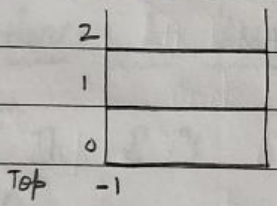
- It is a linear data structure which follow serves as a collection of elements
- Insertion & deletion can be performed only from one end.



- Operations :

- Peek/Top ( ) → top value
- isEmpty ( ) → boolean
- isFull ( ) → boolean → Time complexity: O(1)
- push (x) → insert x on the top
- pop ( ) → returns & removes top value

- When u create a stack, top = -1



Max cap of stack = 3

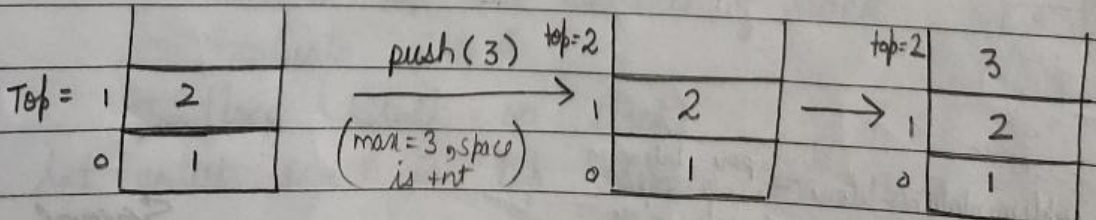
So, if top = -1, isEmpty( ) → true
top = max-1, isFull ( ) → true

Push (*) operation

Push (stack, top, max, item)

- If top = max-1, ~~OVF~~ stack overflow
- otherwise top++
  stack [top] = item

end.

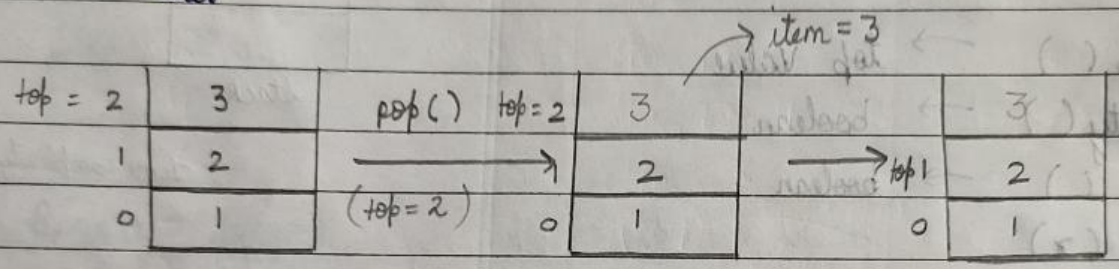


Spiral

Pop() Operation

Pop (stack , top , item)

- If top = -1 , return stack underflow (nothing to pop)
- else
  item = stack [top]
  top = top - 1
  end



all items above top is serves as garbage.
(bcz once u apply push operation, it will be overwritten)

Applications of stack

- Reverse a string
- Undo operations in text editor
- Recursion
- To check balance of parenthesis
- evaluate postfix expressions
- convert infix to postfix
- Keeping track of page visited history of web browser.

implementation done - ° pare - balancing

⊛ Evalution of of arithmetic & logical expressions :

- infix notation : <operand> <operator> <operand>
  eg A + B

Polish notation
- Prefix notation : <operator> <operand> <operand>
  eg + A B

Reverse polish notation
- Postfix notation : <operand> <operand> <operator>
  eg A B +

Prefix & postfix expressions takes less time & memory
Infix expressions are good for humans to understand while postfix & prefix are good for computer

★ Precedence & Associativity for infix expressions

[ ] { } ( )	
^	R → L
* /	L → R
+ −	L → R

★ Evaluating postfix expressions O(n)

- Add a right parenthesis at the end of expression P
- Scan P from left to right until sentinel is encountered
- If operand is encountered, put it on stack
- If operator ⊗ is encountered
  - remove top two elements of stack

A
B

  - Evaluate B ⊗ A
  - Place result on stack
- Set value equal to top element on stack

end

Spiral

brackets are not used in pre & post

no of operands = no. of operators + 1

eg: 5, 6, 2, + , * , 12 , 4 , / , −)

5 6 2 + * 12 4 / −

5 8 * 12 4 / −

40 12 4 / −

40 3 −

37

Symbol scanned	stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
−	(37) pop() returned
)	& stack becomes empty

5, 3, + , 6, 2, / , * , 3, 5, * , +)

symbol scanned	stack
5	5
3	5, 3
+	8
6	8, 6
2	8, 6, 2
/	8, 3
*	24
3	24, 3
5	24, 3, 5
*	24, 15
+	(39) pop()
)	

- Postfix / Prefix can be evaluated using stacks
  so infix is converted to post / pre
- while evaluating post / pre, stack contains operands only
- while converting infix → postfix, stack contains operators only

Dt. Page

* Converting infix exp. into postfix

- Put (on stack &) at the end of expr.
- Print operands as they arrive.
- If stack is empty or contains left parenthesis push incoming operator onto the stack
- If incoming symbol is '(', push it onto the stack
- If incoming symbol is ')', pop the stack & print the operators until left parenthesis is found.
- If incoming operator has higher precedence than top of the stack, push it on the stack
- If incoming operator has lower precedence than top of the stack; pop & print the top. Then test the incoming operator against the new top of the stack.
- If incoming operator has equal precedence with the top, use associativity rule
  - For L to R, pop & print the top & push incoming operator. Also check against the new stack
  - For R to L, push the incoming operator
- At the end, pop & print all operators of stack.

eg: (A-B)* (D/E))

Symbol scanned	Stack	P
(	((	
A	((	A
-	((-	A
B	((-	AB
)	(	AB-
*	(*	AB-
(	(*(	AB-
D	(*(	AB-D
/	(*(/	AB-D
E	(*(/	AB-DE
)	(*	AB-DE/
)		AB-DE/*

Simple conclusion : While evaluating prefix → Reverse string
reverse so op1 op2
While evaluating postfix → ✗
✗ so op2 op1

op1 op2

op1
op2

Dt. Page

- (A+B^D)/(E-F)+G)

Symbol scanned	Stack	P
(	(	
A	((	A
+	((+	A
B	((+	AB
^	((+ ^	AB
D	((+ ^	ABD
)	(	ABD^+
/	(/	ABD^+
(	(/(	ABD^+
E	(/(	ABD^+E
-	(/(-	ABD^+E
F	(/(-	ABD^+EF
)	(/	ABD^+EF-
+	(+	ABD^+EF-/
G	(+	ABD^+EF-/G
)		ABD^+EF-/G+

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

Symbol scanned	Stack	P
A	( )	
-	(-)	A
B	(-)	A
/	(-	AB
C	(-/)	AB
*	(-/	ABC
D	(-*	ABC/
+	(-*	ABC/D
E	(+	ABC/D*-
)	(+	ABC/D*-E Spiral
		ABC/D*-E+

without stack A + (B * C)
A + BC *
ABC * +
Multiple scans

with stack in one scan

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

(
(*(+
(*
(/
(- * (+ /

ABD+*E/FGHK/+*-

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

(+
(+*
(+(^
(+*
(+*

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

K+

* Converting prefix to postfix expression O(n)

- Read the prefix expression in reverse order
- If symbol is operand, push it on the stack
- If symbol is operator, pop two operands from the stack
  Then create a string by concatenating the two operands and the operator after them
  string = operand1 + operand2 + operator
  Push the resultand string back to stack
- Repeat above steps until the end

eg: *- AB/DE
Reverse → ED/BA-*

Scan	Stack	Postfix
E	E	
D	E, D	
/	DE/	DE/
B	DE/, B	
A	DE/, B, A	
-	DE/, AB-	AB-
*	AB-DE/*	AB-DE/*

Spiral

• / * - + A B C ^ D ^ E F G

Rev: G F E ^ D ^ C B A + - * /

Scan	Stack
G	G
F	G, F
E	G, F, E
^	G, EF^
D	G, EF^, D
^	G, DEF^^
C	G, DEF^^, C
B	G, DEF^^, C, B
A	G, DEF^^, C, B, A
+	G, DEF^^, C, AB+
-	G, DEF^^, AB+C-
*	~~G, DEF^^~~ G, AB+C-DEF^^*
/	AB+C-DEF^^*G/

* ~~*~~ *-A/BC-/AKL

Rev : LKA/-CB/A-*

Scan	Stack
	L, K, A
	L, AK/
	AK/L-, C, B
	AK/L-, BC/, A
	AK/L-, ABC/-
	ABC/-AK/L-*

• +/+A^BD-EFG

Rev: GFE-BD^A+/+

G, F, E

G, EF-, B, D

G, EF-, DB^, A

G, EF-, ADB^+

G, ADB^+EF-/ → ADB^+EF-/G+

Spiral

★ Converting postfix to prefix

O(n)

- Scan left to right
- If the symbol is an operand, push it on the stack
- If the symbol is an operator, pop two operands from the stack
  Create a string = operator + operand 2 + operand 1
  Push this string onto the stack
- Repeat the above steps until the end.

eg: ~~LKA/-CB/A-*~~

ABD + * E/F G HK / + * -

Scan	Stack
A	A
B	A, B
D	A, B, D
+	A, +BD
*	*A+BD
E	*A+BD, E
/	/*A+BDE
F	/*A+BDE, F
G, H, K	/*A+BDE, F, G, H, K
/	/*A+BDE, F, G, /HK
+	/*A+BDE, F, +G/HK
*	/*A+BDE, *F+G/HK
-	-/*A+BDE*F+G/HK

Directly → (in steps)

A + (B * C)

A + (* BC)

+ A * BC

using stack → (In one go)

(C * B) + A

(	(	
C		C
*	(*	C
B	*	CB
)		CB *
+	+	
A	+	CB * A
		CB * A +

↔ + A * BC

Dt. Page

★ Infix to Prefix conversion

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

K + L − M * N + (^ OP) * W / U / V * T + Q

MULTIPLE SCANS

K + L − (* MN) + (* ^ OPW) / U / V * T + Q

K + L − (* MN) + (/ * ^ OPW U) / V * T + Q

K + L − (* MN) + (// * ^ OPW U V) * T + Q

K + L − (* MN) + (* // * ^ OPW U V T) + Q

∴ Stack is better

(+ KL) − (* MN) + (* // * ^ OPW U V T) + Q

(− + KL * MN) + (* // * ^ OPW U V T) + Q

(+ − + KL * MN * // * ^ OPW U V T) + Q

++ − + KL * MN * // * ^ OPW U V T Q

Algorithm

- Reverse the expression & swap ')' with '(' & vice versa.
- Print operands as they arrive.
- If stack is empty or contains left parenthesis on top, push incoming operator onto the stack.
- If incoming symbol is (, push it on the stack.
- If incoming symbol is), pop the stack & print the operators until a left parenthesis is encountered.
- If incoming operator has higher precedence than the top of the stack, push it on the stack.
- If incoming operator has lower precedence than top of the stack, pop & print the top. Then test the incoming operator against the new top of the stack.
- If incoming operator has the equal precedence with the top of the stack, use associativity rule
  - L to R push the incoming operator
  - R to L pop & print the top & check the incoming operator with new top of the stack
- At the end, pop & print all operators of the stack
- Finally reverse the expression.

Spiral

Example:

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

Reverse : (E/D) * (B-A)

Scan	Stack	P
(	(	
E	(	E
/	(/	
D	(/	ED
)		ED/
*	*	
(	*(	
B		ED/B
-	*(-	
A	*(-	ED/BA
)	*	ED/BA-
		ED/BA-*

Reverse : *-AB/DE

- (A + B^D) / (E-F) + G

Reverse : G + (F-E) / (D ^ B + A)

+ (~~(~~ GFE-DB^A+/+

+ / (^

+ / (+

Ans -> +/+A^BD-EFG

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

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

/ (^ GFE^D^CBA+-*/

/ (* (- +

Ans /*-+ABC^D^EFG

Spiral

O(n)

★ Balancing the brackets

- Scan expression from left to right
- If '(', '{', '[' is encountered, push it to stack.
- else if stack is not empty & element at the top of stack is matching starting bracket to the currently scanned character, pop out the stack.

  { → } , (→) , [→]

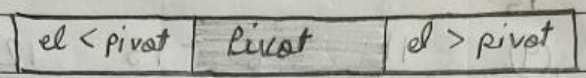
  else return parenthesis are not balanced.
- At the end if stack is not empty return false otherwise return true.

Sorting Algorithm

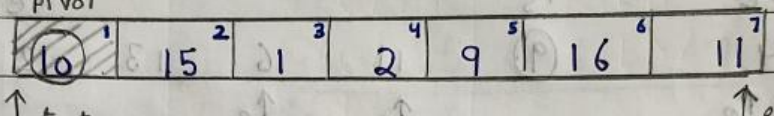
QUICK SORT (Path of stock)

- use divide & conquer method
- selects a pivot element and sets it at its position (when arranged in a particular order)

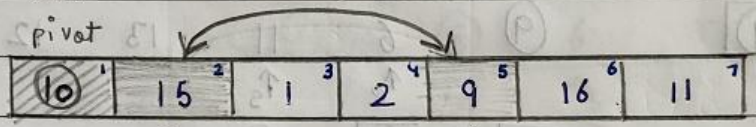
(equal elements can be either side)

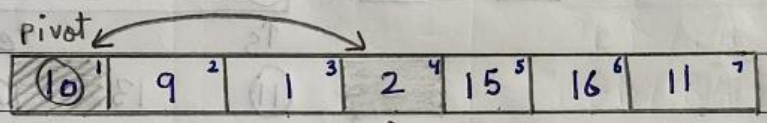


Example :

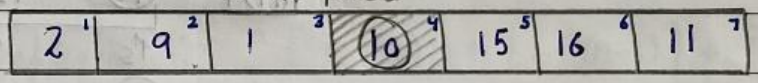


Case 1 :

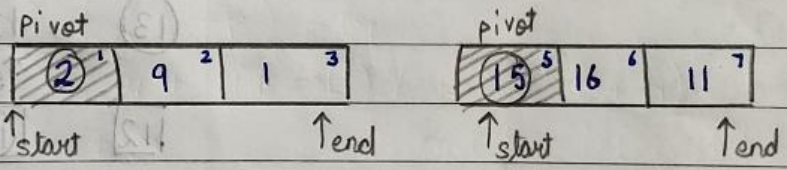


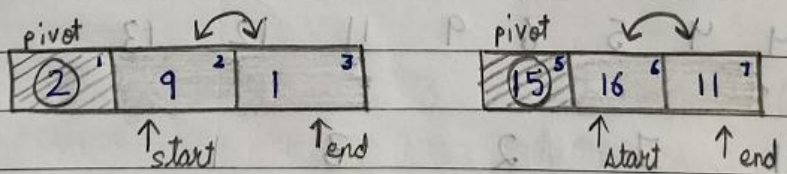


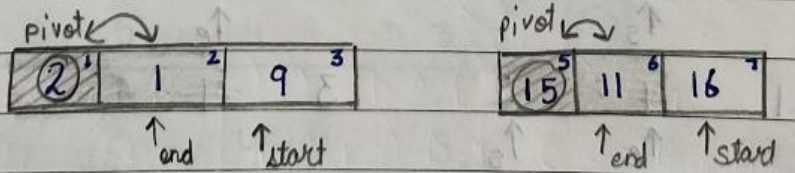
10 is placed

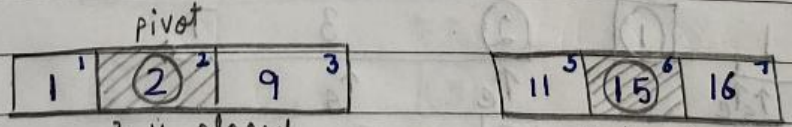


2 :

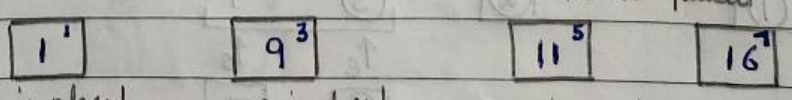








3 :



So we get: 1 | 2 | 9 | 10 | 11 | 15 | 16 Spiral

TIPS :

start
- inc by 1 : el ≤ pivot
- stops : el > pivot

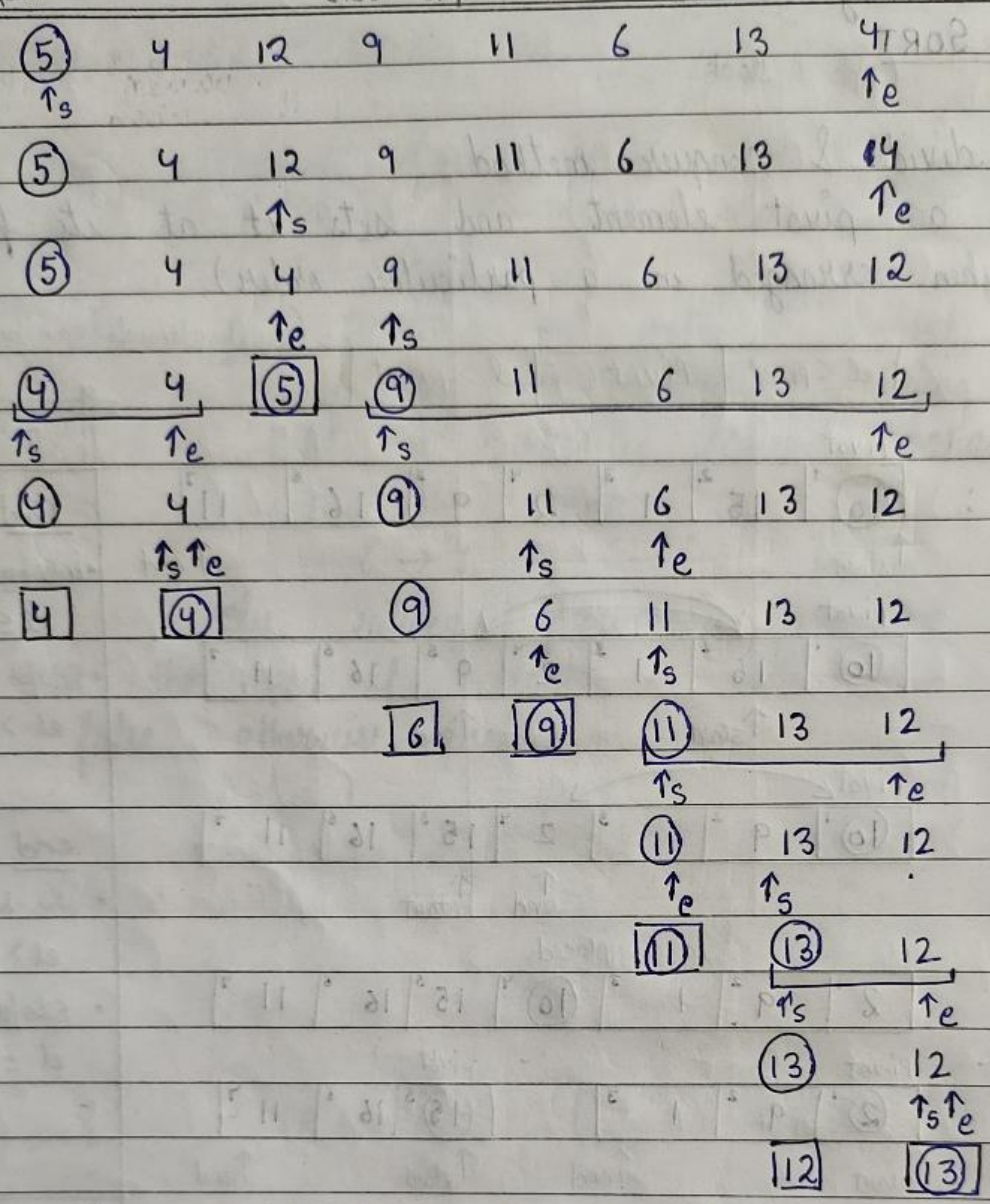
end
- dec by 1 : el > pivot
- stops : el ≤ pivot

if start < end :
swap : start & end

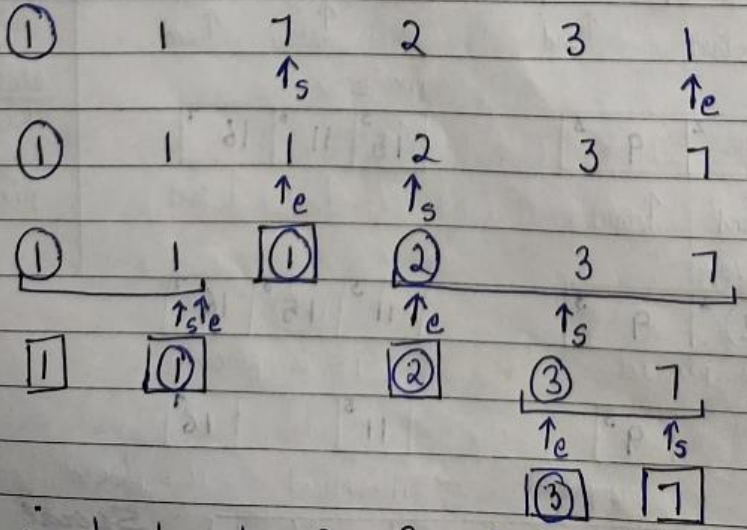
else
break
swap : pivot & end

∴ we put ∞ at the end always has an option to stop even if all elements are greater bcz it crosses pivot & when el ≤ pivot end stops

Recommended But start may not stop ever as (if all el are smaller) start stops when el > pivot



∴ 4 4 5 6 9 11 12 13



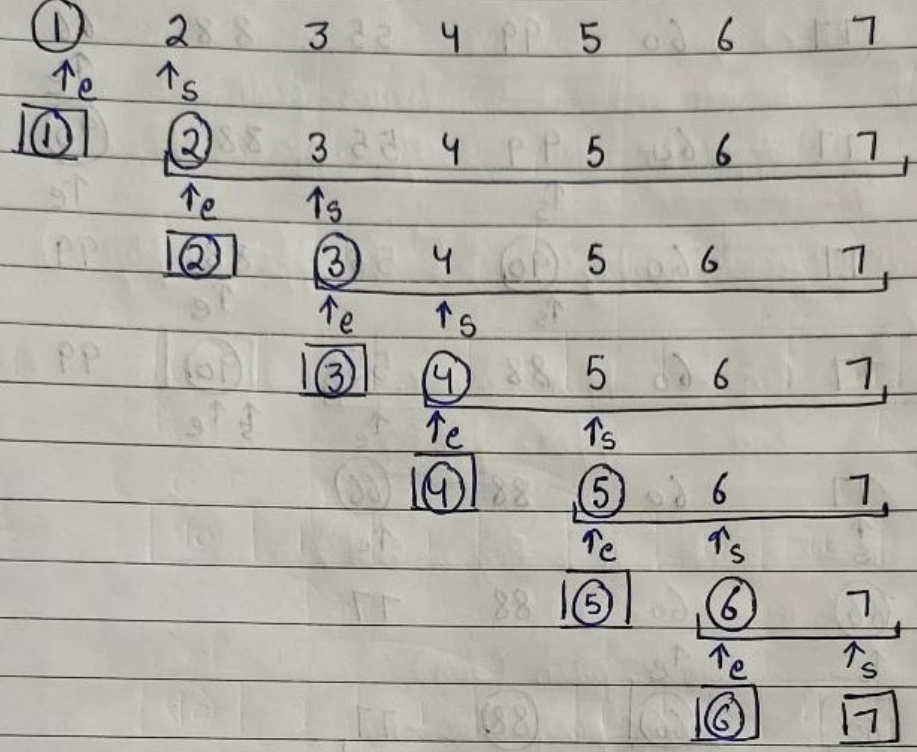
∴ 1 1 1 2 3 7

using this algo, el = pivot are placed on LHS

Spiral

otherwise in avg case
Time complexity = nlogn

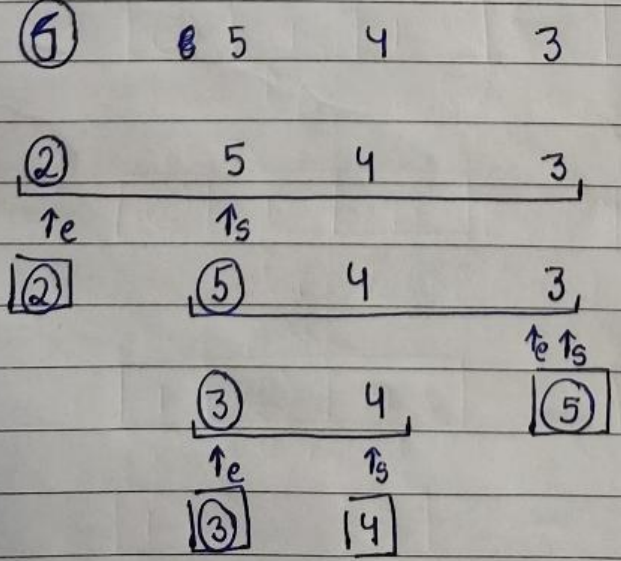
→ **Worst cases** : when elements are already arranged



We get 1 2 3 4 5 6 7

Time complexity : $[1+n] + [1+(n-1)] + [1+(n-2)] + \cdots + [1+2] + [1+1]$

$$n + \frac{n(n+1)}{2} = O(n^2)$$



$(n) + (n-1) + (n-2) + \cdots$

$\frac{n(n+1)}{2}$

$O(n^2)$

Another method :

- 90 77 60 99 55 88 66
  s e
- 66 77 60 99 55 88 90
  s e
- 66 77 60 90 55 88 99
  s e
- 66 77 60 88 55 90 99
  s e s e
- 55 77 60 88 66
  s e
- 55 66 60 88 77
  s e e
- 55 60 66 88 77
  s e s e
- 77 88
  s e
- 55 60 66 77 88 90 99

Algorithm

```
Quick ( A , Beg , End )
{
① Left = Beg ;
  Right = end ;
  Loc = Beg ;

② while ( A[Loc] ≤ A[Right] && Loc ≠ Right )        s<e
    {     Right -- ;
    }
  If ( Loc = Right )                                s≥e
     return Loc ;
  else
    { swap ( A[loc] , A[Right] ) ;
      Loc = Right ;
      go to step 3
    }

③ while ( A[Left] ≤ A[Loc] && Left ≠ Loc)           s<e
      Left ++ ;
  If ( Loc = Left)                                  s≥e
     return loc ;
  else
    {   swap ( A[Left] , A[Loc]);
        Loc = Left;
        goto step 2
    }
```

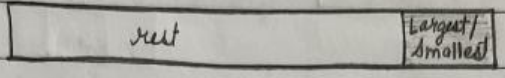
```
Quicksort ( A )
{
    If (N > 1)
     {   ~~Lower [1] = 1 ;~~   Lower. push (1)
         ~~Upper [1~~          Upper. push (N)
     } else return A
while (! lower. empty ()  && ! upper. empty ())
 {
      Beg = ~~upper.~~ lower. pop ()
      End = upper. pop ()

  loc = Quick ( A , Beg , End)
  If ( Beg < loc - 1)
    {   lower. push (beg)
        upper. push (loc - 1)
    }
  If ( loc + 1 < End)
    {   lower. push ( loc + 1)
        upper. push (end)
    }

}
return A
```

BUBBLE SORT

- Each element is compared to its succeeding element (in multiple passes) positioning the greatest / smallest element at the last position.



Example : 32 51 27 85 66 23 13 57

Pass : 1

32 27 51 85 66 23 13 57

32 27 51 66 85 23 13 57

32 27 51 66 23 85 13 57

32 27 51 66 23 13 85 57

32 27 51 66 23 13 57 | 85

Pass : 2

27 32 51 66 23 13 57

27 32 51 23 66 13 57

27 32 51 23 13 66 57

27 32 51 23 13 57 | 66

Pass : 3

27 32 23 51 13 57

27 32 23 13 51 | 57

Pass : 4 27 32 23 13 51

27 23 32 13 51

27 23 13 32 **51**

Pass : 5 23 27 13 32

23 13 27 **32**

Pass : 6 13 23 **27**

Pass : 7 **13** **23**

∴ 13 23 27 32 51 57 66 85

For n elements → K : 1 → n-1 pass

in each pass → 1 → n-K

Time complexity (large) : $(n-1) + (n-2) + \dots + 2.1$

$\frac{n(n-1)}{2}$

$O(n^2)$

Algorithm

Bubble Sort (Data , N)

1. Repeat for k = 1 to N-1 — 0 to N-2
2. { Repeat for PTR = 1 to N-k — 0 to N-k-1
   a) If Data [PTR] > Data [PTR+1]
      swap (Data [PTR], Data [PTR+1])

   b) PTR = PTR+1 }

exit.

* Implementation of stack using arrays
(static memory allocation)

create an array stack [N] ;
top = -1 ;

If top++ ;
↳ u don't overwrite the value
u will be handling a garbage value
∴

	0	1	2	3	4	5
Stack	1	2	3	4	5	6

↓

top = 5	6
4	5
3	4
2	3
1	2
0	1
-1	

→ push (x)
O(1)
```
{   If ( top = N-1)
        print ( stack overflow)
    else
    {   top ++ ;
        stack [ top ] = x ;
    }
}
```

→ pop ( )
O(1)
```
{   If ( top = -1 )
        print ( stack underflow )
    else
    {   print ( stack [top]);
        top -- ;
    }
}
```

→ peek ( )
O(1)
```
{   If ( top = -1 )
        print (stack underflow)
    else
        print (stack [top]);
}
```

→ display ( ) Traverse
O(n)
if bottom to top → 0 – top
i=0 ; i <= top ; i++
```
{   for( int i = top ; i >= 0 ; i -- )
    {   print stack [i] ;
    }
}
```

In postfix → scan from left AB + A+B

In prefix → scan from right − AB A-B

Dt. Page

(Q4)

Evaluation of prefix expression

infix : 2 + 3 * 4 - 16 / 2 ^ 3 = 12

~~infix : a + b *~~

- Reverse the expression
- Scan from left to right
- If operand is encountered, push it on stack
- If operator ⊗ is encountered
  - remove top two elements of stack
  - Evaluate A ⊗ B
  - Place result on stack
- return top value of stack at the end

A
B

eg: 3 2 ^ 16 / 4 3 * 2 + −

Scan	Stack
3	3
2	3, 2
^	8
16	8, 16
/	2
4	2, 4
3	2, 4, 3
*	2, 12
2	2, 12, 2
+	2, 14
−	(12) →

− + 2 * 3 4 / 16 ^ 2 3

− + 2 * 3 4 / 16 8

− + 2 * 3 4 2

− + 2 12 2

− 14 2

(12)

Multiple scans

postfix 2 3 4 * + 16 2 3 ^ / −

2 12 + 16 2 3 ^ / −

14 16 2 3 ^ / −

14 16 8 / −

14 2 − = (12)

* Postfix to Infix conversion

- Scan from left to right
- If operand comes, push it on the stack
- If operator comes
  - pop two elements from stack
  - string = (operand 2 + operator + operand 1)
  - Push string on stack
- repeat till the end of expression

ab + ef / *

Scan	Stack
a	a
b	a , b
+	a + b
e	a+b , e
f	(a+b), e , f
/	(a+b), (e / f)
*	((a+b) * (e / f))

* Prefix to infix conversion

- Reverse the string
- Scan from left to right
- If operand comes, push it on the stack
- If operator comes : - pop two elements
  - push → (operand 1 + operator + operand 2)
- Repeat till the end

* + ab / ef ⟷ fe / ba + *

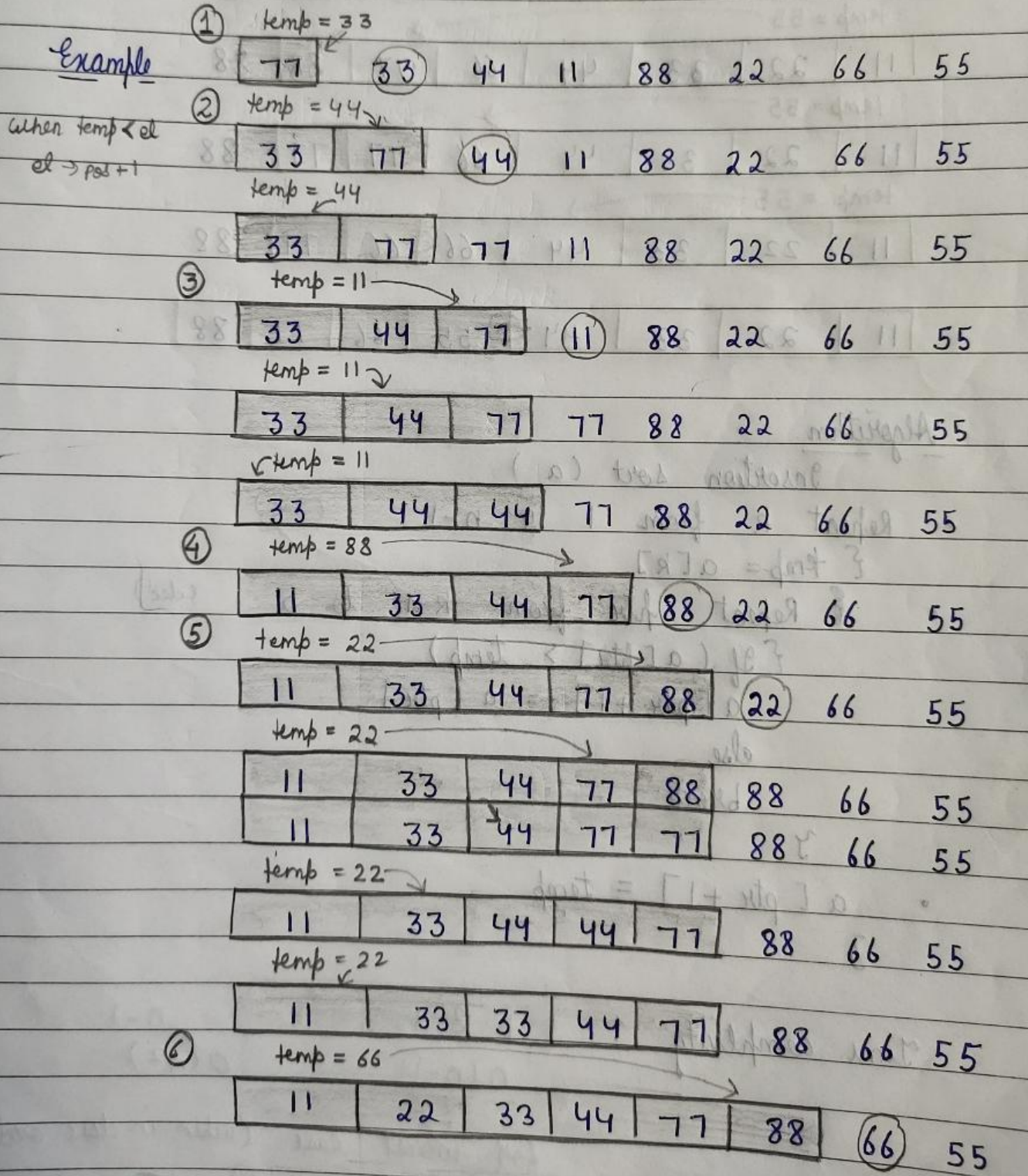
Scan	Stack
f	f
e	f , e
/	(e / f)
b	(e / f) , b

Scan	Stack
a	(e / f) , b , a
+	(e / f) , (a + b)
*	((a+b) * (e / f))

INSERTION SORT

In this sort, we take a sorted list (start from one element) then, place the remaining elements in the sorted list at their correct position.

Example



Spiral

temp = 66

11	22	33	44	77	88	88	55

temp = 66

11	22	33	44	77	77	88	55

⑦ temp = 55

11	22	33	44	66	77	88	(55)

temp = 55

11	22	33	44	66	77	88	88

temp = 55

11	22	33	44	66	77	77	88

temp = 55

11	22	33	44	66	66	77	88

11	22	33	44	55	66	77	88

Algorithm

Insertion sort (a)

- Repeat k from 1 to n-1 (inc)
  { temp = a[k]
  - Repeat ptr from k-1 to 0 (dec)
    { if (a[ptr] > temp)
      a[ptr+1] = a[ptr]
      else
      break
    }
  - a[ptr+1] = temp
  }

Time complexity : $1 + 2 + 3 + \cdots + n-1$

$$\frac{n(n-1)}{2} = O(n^2)$$

for worst case (when in desc order)

best case (when already in asc order)

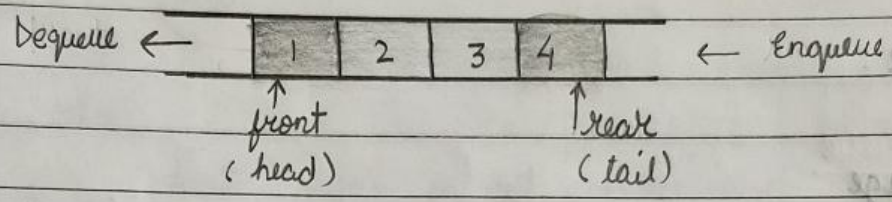
$1 + 1 + 1 \cdots n-1 = n-1 = O(n)$

Spiral

Real life scenario : At ticket counter, queue of people

QUEUE FIFO / LILO

It is a linear data structure in which insertion is performed from one end (i.e. rear end) and deletion is performed from other end (i.e. front end)



- In stack, one open end existed, push & pop were performed from only one end.
  But in queue, two open ends exist.

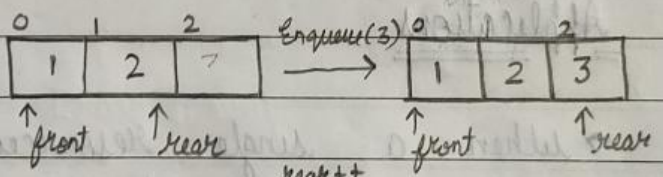
Operations : TC: O(1)

Enqueue()
Dequeue()
Front/peek()

1) Enqueue (x)

if rear != max-1 (simple)

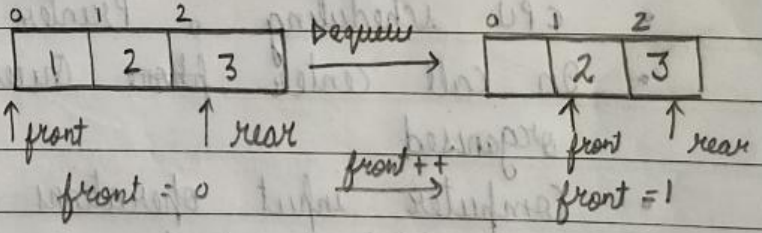
```
rear ++
queue [rear] = x
```



2) Dequeue ()

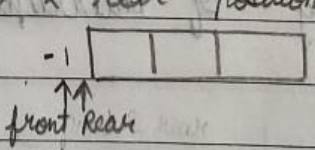
if front != -1 && front ≤ rear (simple)

```
item = queue [front]
front ++
return item
```



→ queue exists between front & rear positions rest all is garbage

→ When u create a queue



3) Front () / Peek

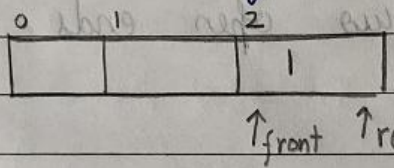
returns value at front

4) isEmpty()
rear = -1 (though front = -1)

5) isFull()
rear = max-1 (front can have any value)

Major Drawback:

Space wastage
When rear = max-1, we can't enqueue elements further even if front too is max-1
This leads to wastage of memory



Applications

→ When a single resource is sharable queue of requests is made.

- CPU scheduling, Printer
- In call center phone Queues of customers are organised
- Computer input operations are performed through mouse / keyboard & are also stored in queues.
- Handling of interrupts

Operations – Queue

- isEmpty (Queue , rear , front)
  { if front = -1 and rear = -1 then
      print " queue is empty "
      exit
    end
    else
      print " queue is not empty "
      exit
    end
  }

- isFull (Queue , rear front , size)
  { if rear = size - 1 then
      print " queue is full "
      exit
    end
    else
      print " queue is not full "
      exit
    end
  }

- enqueue (Queue , front , rear , size, Value)
  { if rear = size - 1 then
      print " queue is full "
      exit
    end
    else if rear = -1 then
      set front = front + 1
      set rear = rear + 1
      set Queue [rear] = value
      exit
    end

```
else
    set rear = rear + 1
    set Queue [rear] = value
    exit
end.
```

- Dequeue (Queue , rear , front)

```
{ if rear = -1 and front = -1 then
    print " queue is empty"
    exit
end
else if rear = front
    print Queue [ front ]
    set rear = -1 and front = -1
    exit
end
else
    print Queue [ front ]
    set front = front + 1
    exit
end
```

- front (Queue , front)

```
{ if front = -1
    print " queue is empty"
    exit
end
else
    print Queue [ front ]
    exit
end
```

- rear (Queue , rear)

```
{ if rear = -1
    print "queue is empty"
    exit
end
else
    print Queue [rear]
    exit
end
```

Operations - Circular Queue

- isEmpty (Queue , front , rear)
  { if front = -1 and rear = -1 then
      print " queue is empty"
      exit
    end
    else
      print " queue is not empty"
      exit
    end

- isFull (Queue , front , rear , size)
  { if (rear + 1) % size = front then
      print " queue is full"
      exit
    end
    else
      print " queue is not full"
      exit
    end
  }

- enqueue (Queue , front , rear , size, Value)
  {
    if (rear + 1) % size = front then
      print " queue is full"
      exit
    end
    else if rear = -1 and front = -1 then
      set rear = 0 and front = 0
      set Queue [rear] = value
      exit
    end

```
    else
        set rear = (rear + 1) % size
        set Queue [rear] = value
        exit
}end
```

- dequeue (Queue , rear , front , size)

```
{ if front = -1 and rear = -1 then
        print " queue is empty "
        exit
  end
  else if front = rear then
        print Queue [ front ]
        set front = -1 and rear = -1
        exit
  end
  else
        print Queue [ front ]
        set front = ( front + 1 ) % size
        exit
  end
}
```

- front (Queue , front)

```
{ if front = -1 then
        print " queue is empty "
        exit
  end
  else
        print Queue [ front ]
        exit
  end
}
```

- rear (Queue , rear)

```
{ if rear = -1 then
        print " queue is empty "
        exit
  end
  else
        print Queue [ rear ]
        exit
  end
```

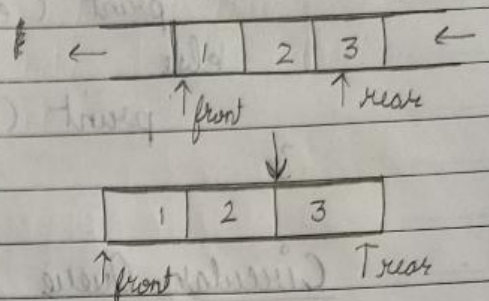
Spiral

When queue is empty → inc f, r both
When queue has last element left → reset both f, r

Dt. Page

* Implementation of Queue using arrays

Create an array queue[N];
front = -1;
rear = -1;

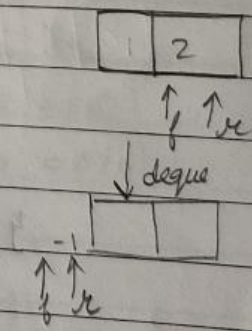


→ enqueue (x)

```
{   if (rear == N-1)
      print (overflow);
    else if (front == -1 && rear == -1)
    {   rear ++;
        front ++;
        queue [rear] = x;
    }
    else
    {   rear ++;
        queue [rear] = x;
    }
}
```

→ dequeue ()

```
{   if (front = -1 || front > rear)
      print (empty queue)
    else if (front == rear)
    { front = rear = -1; }
    else
    {   front ++; }
}
```



→ display ()

```
{   if (front == -1 && rear == -1)
      print (empty queue)
    else
    {   for (int i = front; i <= rear; i++)
          print (queue [i])
    }
```

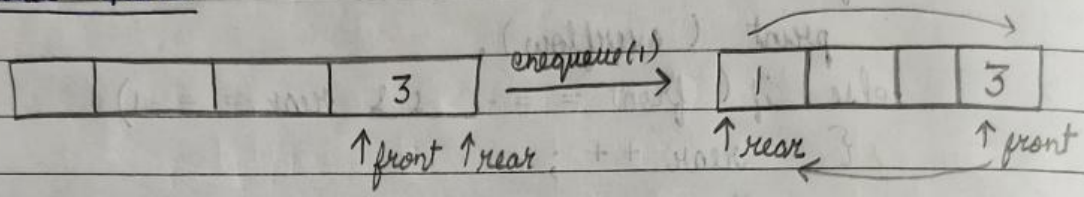
Spiral

⇒ Front()
⇒ Rear()

→ peek()

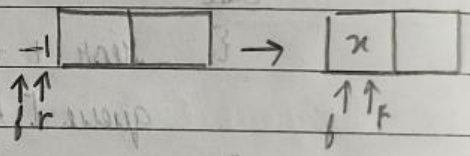
```
{   if ( front == -1 && rear == -1)
        print ( empty queue)
    else
        print ( queue [ front ])
}
```

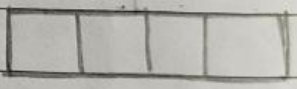
Circular Queue



→ enqueue (x)

```
{   if ( rear == -1 && front == -1)
    { ~~print(~~
        front ++;
        rear ++;
        queue [ rear ] = x
    }
    else if ( (rear +1) % N == front)
        print ( overflow)
    else
    { rear = (rear +1) % N
        queue [ rear ] = x;
    }
}
```





→ dequeue()

```
{   if ( rear == -1 && front == -1)
        print (underflow)
    else if ( front == rear)
    { front = -1 ; rear = -1 }
```

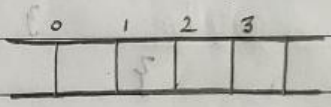
Spiral

```
else
{   front = (front+1)%N
}
```

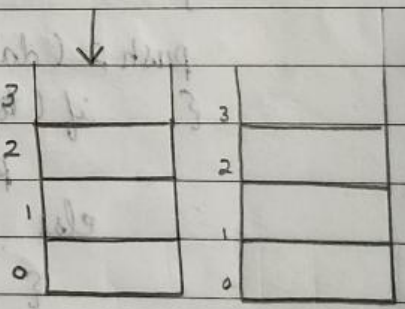
→ display()

```
{   if ( front==-1 && rear == =-1)
        print (empty queue)
    else        int i=0;
    {   for( int i = front ; i != rear ; i = (i+1)%N)
            {   print ( queue [i])
            }
        print( queue [i]
    }
}
```

★ Queue using Stack



Create two stacks S1[N], S2[N]
int top1 = -1 , top2 = -1 , count = 0;



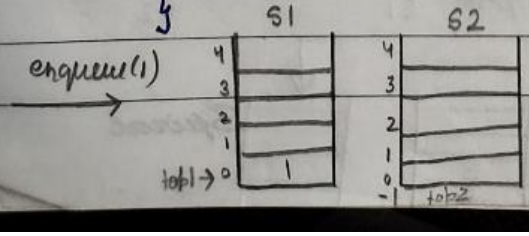
→ enqueue (x)

```
{
  S1.push1(x);    in S1
}
```

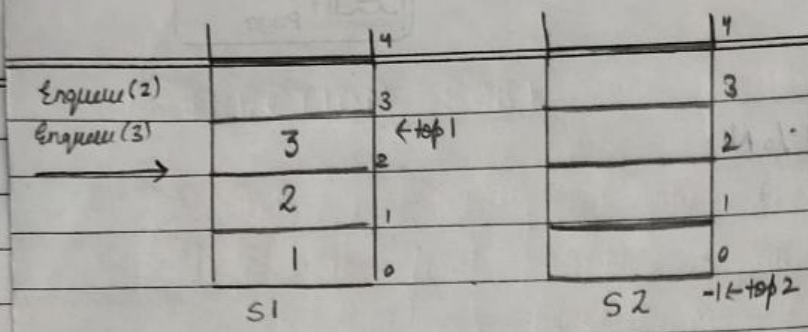
push1 (data)

```
{   if( top1 = = N-1)
        print ( stack overflow)
    else
    {   top1++;
        S1[top1] = data;
    }
}
```

enqueue = O(1)
dequeue = O(n)



[count = 1]



→ dequeue ()

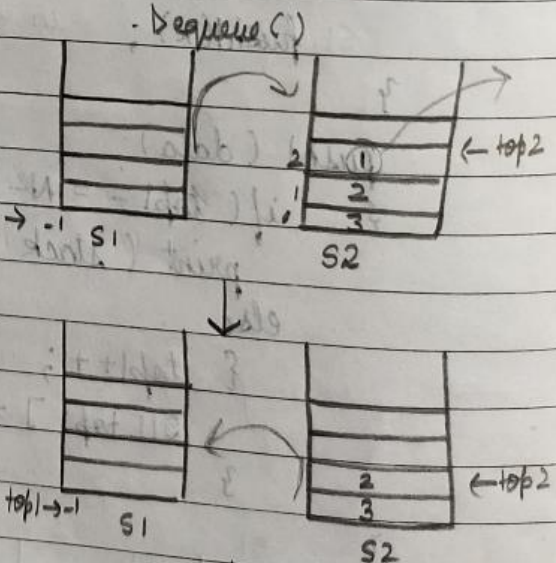
```
{   if (top1 == -1 && top2 == -1)
        print - empty queue
    else
    {   for (int i = 0; i < count; i++)
        {  push2(pop1());
        }
        a = pop2(); print(a); count--;
        for (int i = 0; i < count; i++)
        {   push1(pop2());
        }
    }
}
```

```
push2 (data)
{   if (top2 == N-1)
        print (stack overflow)
    else
    {   top2++
        S2[top2] = data;
    }
}
```

not req (×) already checked in stack1

```
pop1 ()
{   return S1[top1--];
}
```

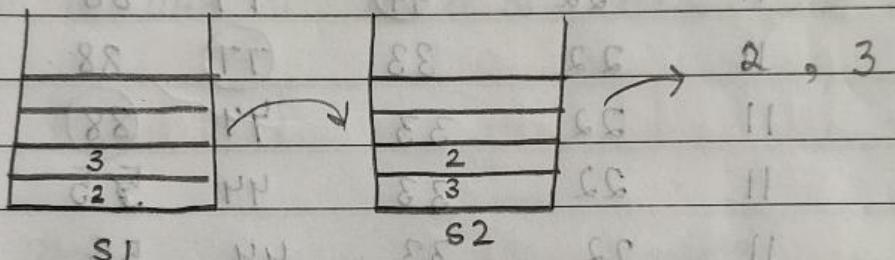
```
pop2 ()
{   return S2[top2--];
}
```



→ display ()

```
{   if (top1 == -1)
        print (empty stack)
    else
    {   for (int i = 0; i < count; i++)
            push2 (pop1());

        for (int i = 0; i < count; i++)
            print (pop2());
```



SELECTION SORT

In this sort we find the smallest element, then place it on the first position. Again 2nd smallest element is discovered & placed on 2nd position & so on.

Example

(77)	33	44	(11)	88	22	66	55
11	(33)	44	77	88	(22)	66	55
11	22	(44)	77	88	(33)	66	55
11	22	33	(77)	88	(44)	66	55
11	22	33	44	(88)	77	66	(55)
11	22	33	44	55	(77)	(66)	88
11	22	33	44	55	66	(77)	88

Algorithm

```
Selection sort (A, N)
• Repeat k from 0 to N-2
  { loc = call min (A, N, k)
    set temp Swap ( A[k] , A[loc])
  }

min(A, N, K, loc)
{  set min = A[K]  & loc = K
   Repeat for j from k+1 to N-1
   {  If min > A[j]
      { set min = A[j] &
        loc = j
      }
   }
   return loc
}
```

TC:
$(n-1)+(n-2)+\dots 3,2,1$
$= \frac{n(n-1)}{2}$
$O(n^2)$

SC = O(1)

Spiral

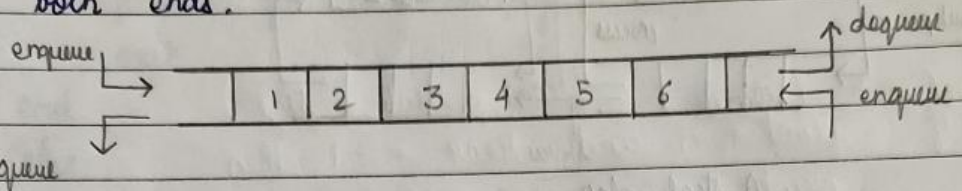
In circular array, condition for
inc. : $(i+steps)\%\ size$ $\quad i \rightarrow$ reference index
dec. : $(size-1)-[(size-1)-(i-1)+(steps-1)]\%\ size$
$(size-1)-[size-i+steps-1]\%\ size$

Dequeue [DEQ]

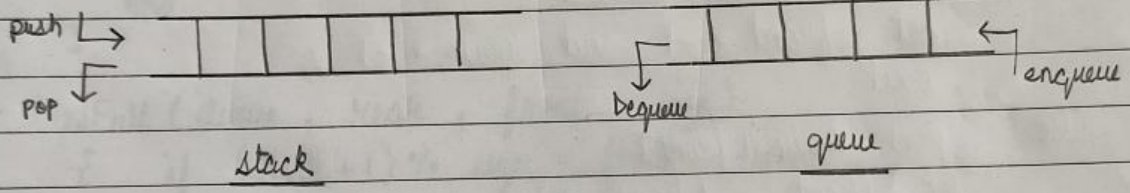
FILO + LIFO

(Double ended queue)

Queue in which insertion and deletion can be performed at both ends.



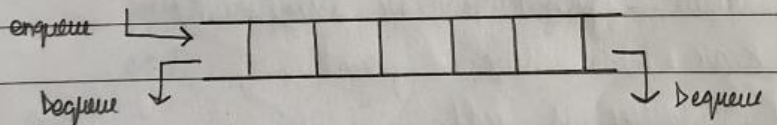
It can be used as stack as well as queue



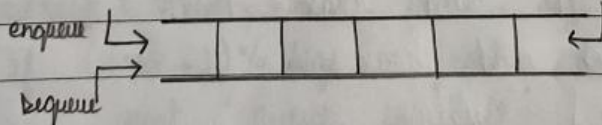
stack queue

Types:

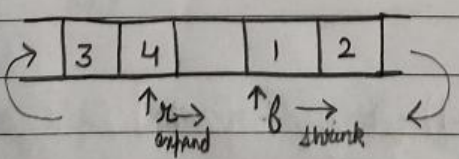
- Input-restricted: enqueue - from one end only (from ~~front~~, say)



- Output-restricted: dequeue - from one end only (from ~~rear~~, say)

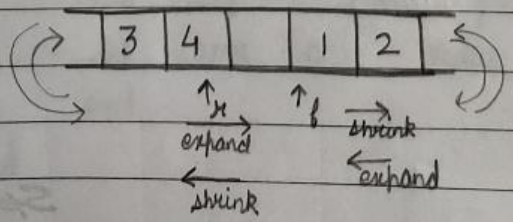


c queue :



both front and rear are inc. to dequeue & enqueue

DEqueue

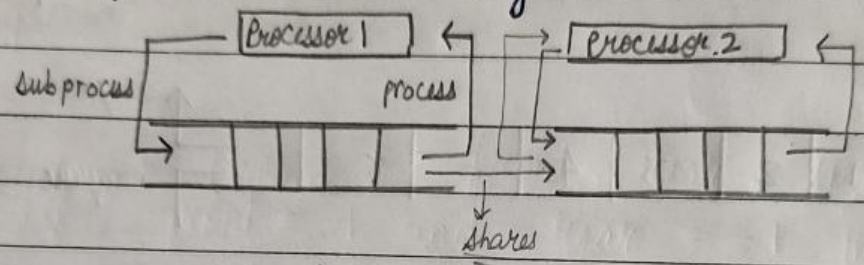


both front & rear are inc as well as dec to dequeue & enq.

Spiral

Applications

- For redo / undo operations.
- For palindrome checking
- In multiprocessor - scheduling



A steal algo

Operations - Dequeue

- isEmpty (Queue , rear , front)
  { if rear = -1 and front = -1 then
      print " queue is empty "
      exit
    end
    else
      print " queue is not empty "
      exit
    end
  }

- isFull (Queue , rear , front , size)
  { if (rear + 1) % size = front then
      print " queue is full "
      exit
    end
    else
      print " queue is not full "
      exit
    end
  }

- enqueueRear (Queue , rear , front , size, Value)
  { if (rear + 1) % size = front then
      print " queue is full "
      exit
    end
    else if rear = -1 and front = -1 then
      set rear = 0 and front = 0
      set Queue [0] = Value
      exit
    end

```
    else
        rear = (rear + 1) % size
        set Queue [ rear ] = value
        exit
    end
}
```

- dequeueFront (Queue , front , rear , size)

```
{   if front = -1 and rear = -1 then
        print " queue is empty "
    else if front = rear
        print Queue [ front ]
        set front = -1 and rear = -1
        exit
    else    print Queue [ front ]
        set front = (front + 1) % size
        exit
    end
}
```

- enqueueFront (Queue , front , rear , size, Value)

```
{   if (rear + 1) % size = front
        print " queue is full "
        exit
    else if front = -1 and rear = -1
        set front = 0 and rear = 0
        set queue [0] = value
        exit
    else
        set front = (size - 1) - [ size - front ] % size
        set queue [ front ] = value
        exit
    end
}
```

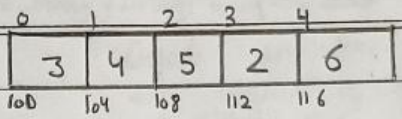
- dequeueRear (queue , rear , front , size)

```
{   if  front = -1  and  rear = -1
        print " queue is empty"
        exit
    else  if  front = rear
        print  queue [ rear ]
        set  rear = (size -1) - (size - rear) % size   and front = -1
        exit
    else
        print  queue [ rear ]
        set  rear = (size -1) - ( size - rear) % size
        exit
    end
}
```

ptr [int *) malloc (n * sizeof(int))

Dt. Page

ARRAYS ADTs



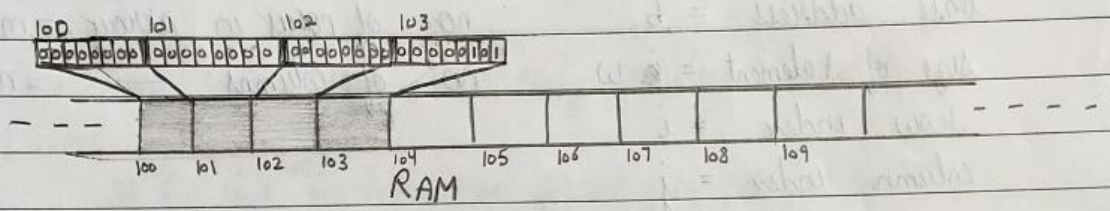
(fixed size sequential collection of homo data)

Contiguous area of memory consisting of equal-sized homogenous elements indexed by contiguous integers.

add- hexa

★ what happens in the memory ;

int a = 5; (5 = 101) int → 4 bytes



Every byte has an address
Address is in hexadecimal form

- space is wasted
- can't be resized

Key points :

- arrays can't be resized
- index of arrays generally start with 0.
- Constant time access to any element.
- Constant time addition / removal at the end / updation
- Linear time ~~so~~ insertion / removal at an arbitrary location.

→ Mechanism of accessing / updating etc elements of ~~of~~ an array

For 1D array :

index (provided by user) = i
base address = b
size of each element = n

int : n=4
float : n=8
char : n=1

memory address for i^{th} element = b + n * i

This whole process is abstracted from the user & dealt with by the compiler
Compiler finds the address using the index & access any element.

Spiral

RAM → linear storage

- In C, bound is not checked → if out of bound index → garbage value
- u can store more elements a[2] = {1, 2, 3}
  actual array, garbage value

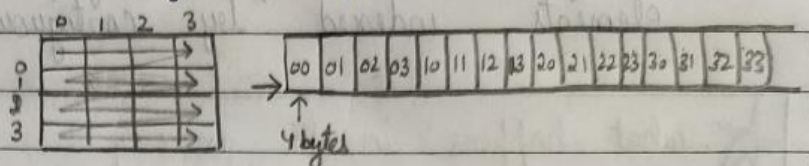
Dt. Page

For 2D arrays :

2D arrays can follow two major ways for storing the elements :

C/C++

1) Row - major order



row index changes slowly

base address = b
size of 1 element = w
row index = i
column index = j

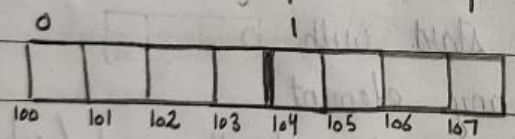
no. of rows in array = m
no. of columns — = n

memory address of A[i][j] = b + w * [n * i + j]

actually i = row index - lower bound of row (lr)
j = column index - ——— column (lc)

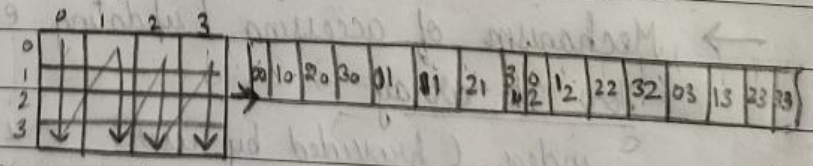
but we take lr & lc = 0 so i = row index j = column index

index is started from 0 for simplification



Memory address for index 1 = 100 + 4(1 - 0) = 104
But if we start it from 1 = 100 + 4(2 - 1) = 104

2) Column - major order



column index changes slowly

memory address of A[i][j] = b + w[i + j * m]

For multi : row major → last index change fastest
column — → first ———

3 ways to implement : • create array, size, capacity globally & use directly
• create array in main & use through pointers
• create array through struct
struct is like a pack which contains the array, its size, its capacity

Dt. Page

Operations on Arrays:

Create an array A & 2 variables capacity & size
set value of capacity
set size = 0

→ Traversal (A, UB)
{ Repeat for K = 0 to UB
Apply the process to A[K]
exit
}

TC : O(n)

→ Insertion (A, size, capacity, item, pos)
{
If (size ~~= capa~~ is equal to capacity
print "overflow"
else if pos > capacity - 1
print "position is not valid"
else
Repeat K from size-1 to pos
set A[K+1] = A[K]
end
set A[pos] = item
set size = size + 1
exit
}

TC : O(n)
(worst case)

→ Deletion (A, size, pos)
{
If pos > size - 1
print "position is not valid"
else if size = 0
print "underflow"
else
Repeat K from pos to size - 2
set A[K] = A[K+1]
end
set size = size - 1
exit
}

TC : O(n)
(worst case)

Spiral

int size = sizeof(arr) / sizeof(int)

* → "value at" operator (dereferencing operator)

& → "address at" operator

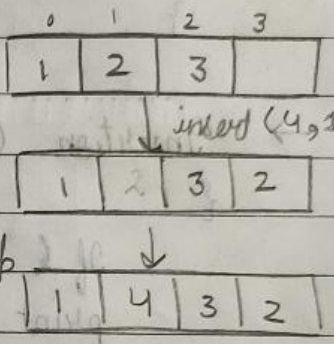
Dt. Page

Unsorted array

We can insert & delete items from an unsorted array with TC = O(1)

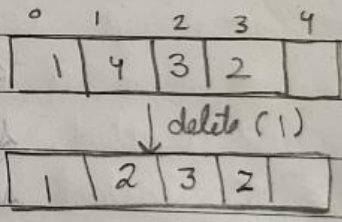
→ Insertion (A , UB , capacity , item , pos)

```
{  If ~~siz~~ = UB = capacity -1
     print " overflow "
   else if pos > capacity -1 "
     print " position is not valid "
   else
      A[UB+1] = A[pos] // simply swap
      A[pos] = item
      set UB ++
   exit
}
```



→ Deletion (A , UB , pos)

```
{  If pos > ~~siz~~ UB
     print " position is not valid"
   else
      A[UB] = A[pos]
      set UB --
   exit
}
```



← Linear search

→ Binary search (A , size , item) For sorted array

```
{ set beg = 0
  set end = size - 1
  set mid = 0
  while ( beg <= end)
     mid = (beg + end)/2
     if ( item = A[mid])
        return mid
```

Keeps on shrinking the array. Meanwhile if middle el is matched returns mid. else → shrink. At last single el remains if matched → mid returned else shrinks more i.e. end < beg ∴ comes out & return -1

Spiral

%p , %x , %u
to print address in hexa form
printf ("%p", p) or printf ("%p", &a)

```
int a = 5;
int *p;
p = &a;
*p = a;
```

array var is a pearl Dt. Page

```
    else if ( item < A[mid] )
        set end = mid - 1
    else
        set beg = mid + 1
    end
    return -1
}
```

TC : O(log n)

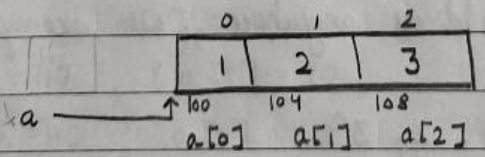
Pointers concepts :

Asterisk * : 'value at' operator / dereferencing operator
Ampersand & : 'address at' operator

eg: int a = 5;
int *p;
p = &a; or *p = a;

→ To print address (a hexadecimal no)
printf ("%p", p); or printf ("%p", &a);
• u can use %x or %u also.

→ array name itself is a pointer, ~~pointing~~ stores ~~to~~ the beg address of array.
~~var. are~~ eg: int a[] = {1, 2, 3}



∴ pointer a stores address of a[0]
Here, a[0], a[1], a[2] are variables

→ pointers can't be added / multiplied etc. X
assignment operator works ✓
inc / dec ——— ✓
eg: a++ X But array pointers can't be inc / dec.
u can't change base address of array.

u can have pointer of a pointer

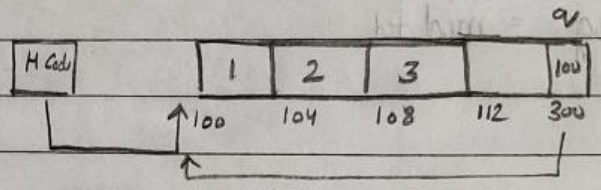
int *p;
p = a
p is pointing to a[0]
only
a is an array pointer
p is an int pointer

Spiral a++ X
p++ → p = 104

u can store base address of some other array in a

a++; ✗ bcz a can store base address of array only as it is an array pointer

(a+1) return → 104



say
int *q;
q = a

if a++ was valid a = 104 but it is not the base address of array.

a[0] ≈ *a ≈ 0[a]

a[1] ≈ *(a+1) ≈ 1[a]

a ≈ &a

so a[i] ≈ *(a+i) ≈ *(i+a) ≈ i[a]

∴ a[i] ≈ i[a] ≈ i[q]

so i[p] means value at (p+i)

↑ any pointer

(a+1) → 104 but &a+1 = 112

&a[0]+1 → 104

*a = 6

*a+1 = 7

*(a+1) = 2

we can use q instead of a

except q[i] ✗ q is not pointing to array

Remember,

we are not storing array address anywhere, we are just retrieving the address

int *p; int a[3] = {1, 2, 3}

p = a; → here we stored the address

java-stored

Spiral

2D arrays

→ Initialisation during runtime - use loops & scanf

→ Initialisation during compile time :

- `int a[2][3] = {1, 2, 3, 4, 5, 6};`
- `int a[2][3] = { {1, 2, 3}, {4, 5, 6} };`

1 2 3
4 5 6

- `int a[ ][3] = {1, 2, 3, 4, 5, 6};`

compulsory (bcz compiler grepts acc to it)

2 rows ← if 7 is also in 3rd row starts where last two columns are empty.

java → `int a[][] = new int[3][];`

compulsory

1 4
2 5
3 6

but for column major matrix :

`int a[3][] = {1, 2, 3, 4, 5, 6};`

compulsory

2 columns

Pointers' concepts :

`int a[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9}`

`int *p;` p = a ✗

	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

row major storage

→ p = &a[0][0] , p = a[0] p = &a[0]

→ print p = 100 , print a = 100 ,
print &a[0][0] = 100 , &a = 100 ,

*a = a[0]

*a = 100 , a[0] = 100

→ a+1 = 112 , &a[1] = 112 ,
*(a+1) = 112 , a[1] = 112
&a[1][0] = 112

→ *(a+1) + 2 = 120
((a+1) + 2) = 6
a[1][2] = 6

∴ a[i][j] = *(*(a+i)+j)
*(a[i]+j)

p

	00	01	02	10	11	12	20	21	22	
--	1	2	3	4	5	6	7	8	9	--
	100	104	108	112	116	120	124	128	132	

Row 1 | Row 2 | Row 3

a[0] a[1] a[2]

a

a = &a
a[0] = &a[0]
a[1] = &a[1]
a[2] = &a[2]

holds true for any array

Spiral

→ *(*a + 1) = 2

**a = 1

**p =

a[1] + 1 = 116

&a[1] + 1 = 124

int a[3];

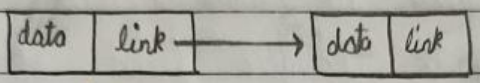
a → points to 1st element

&a → gives base address of complete array

→ pointers take 4 bytes

LINKED LISTS

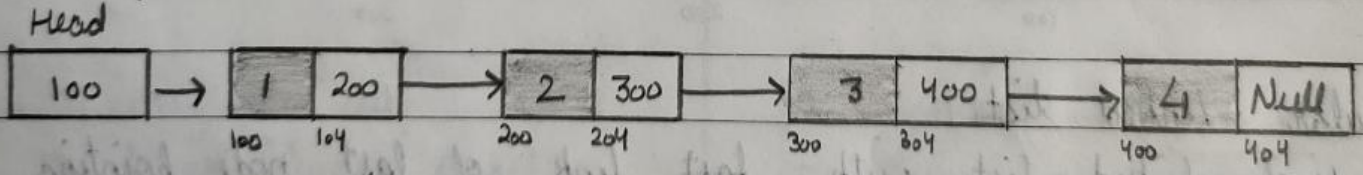
- Linked list is a linear collection of homo data elements which need not be sequentially stored in the memory.
- It has dynamic size.
- It consists of nodes
  - Node contains data and a link to the next node.



- Insertion and deletion is easy - takes constant time.
- Access time is linear.
- Binary search can't be performed on a linked list.

in memory (for int data)

Representation:



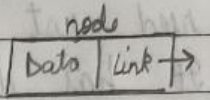
Types of List

```
struct node
{ int data;
  struct node * next;
};
```

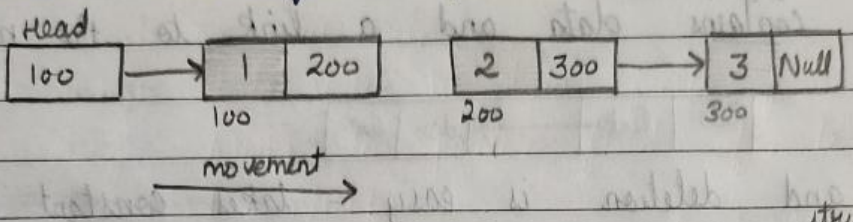
1) Singly Linked List

1 Null link

- most commonly used linked list.
- A node contains one data and one link



- forward Traversingal is possible (in one dir. only)

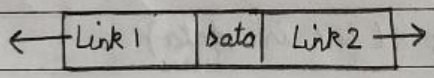


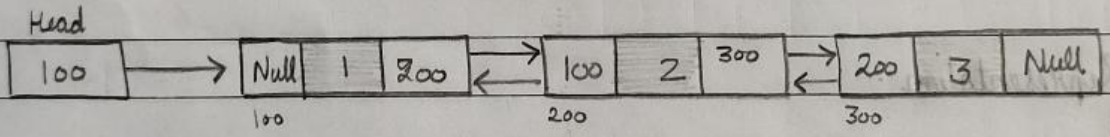

```
struct node
{ int data;
  struct node * next;
  struct node * prev;
};
```

2) Doubly linked list

2 null link

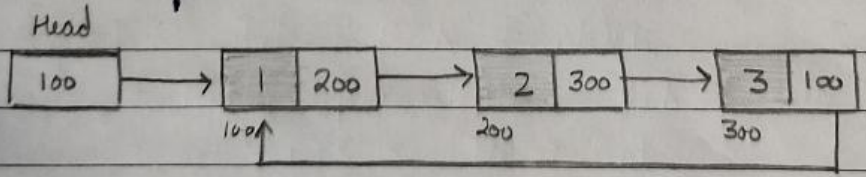
- A node contains 3 parts → data, 2 links (previous & next)
- traversal in both directions





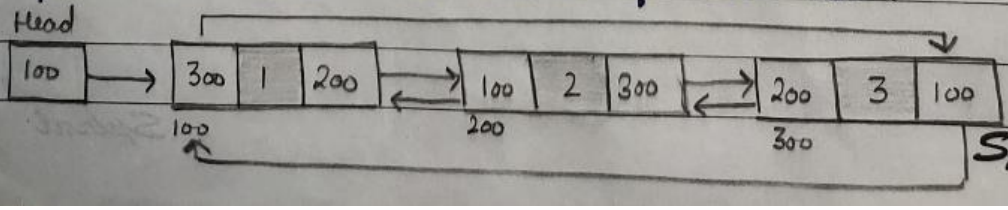
3) Circular Linked list

- singly linked list with ~~last~~ link of last node pointing to the first node. (rather than being null)



4) Doubly circular linked list

doubly linked list in which ~~last~~ next link of last node stores the address of first node while previous link of first node stores the address of last node.



In arrays we create space at start & then insert.
In LL creation & insertion takes place at same time

Array vs. Linked list

Array	Linked List
- contiguous	- non-contiguous
- Accessing an element : TC = O(1)	- Accessing an element : TC = O(n)
- fixed size	- dynamic size
- ~~inef~~ inefficient memory utilisation (memory utilisation)	- very efficient memory utilisation
~~- can't be resized~~	
- no extra space required. (Memory requirement)	- extra space is required to store addresses in the nodes.
- Insertion / deletion • at beg : O(n) • at end : O(1) • at i^{th} position : O(n)	- Insertion / deletion • at beg : O(1) • at end : O(n) • at i^{th} position : O(n)
- easy to use	- Complex
- linear / binary search Traversal through indices	- only linear search traversal through pointers

Creation

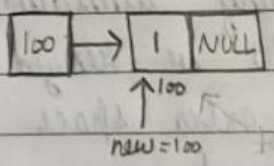
Create struct node ~~with~~ two variables data, link

- ~~Assign~~ Create three pointer variables : head, new, temp
- Set head = NULL
- Repeat for K=1 to N

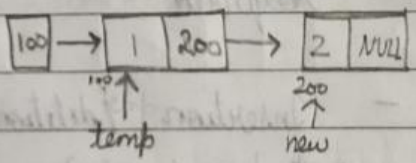
{

- create a new node
- Assign ~~data~~ value to data field of node
- Assign NULL to ~~pointer~~ link field of node
- if head = NULL
  set head = new

we need if and DC2 we can't set temp to point to head as head is pointer but temp can point to struct node only

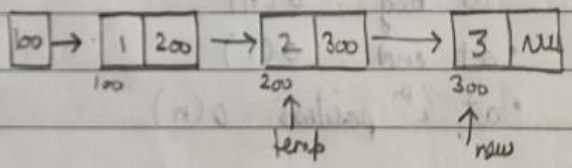


- else
  set temp → link = new



- Set temp = new

}

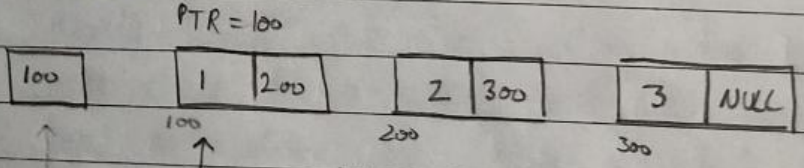


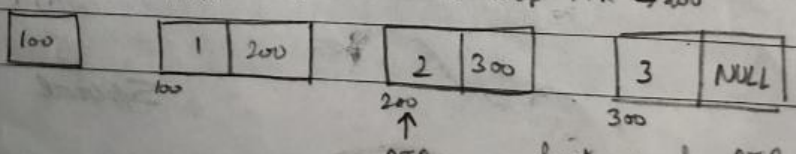
- end

temp will point to previous node so that we can access the link part & store address of new node

Traversal (head)

- Set PTR = head
- Repeat while PTR ≠ NULL
  - Apply process to PTR → data
  - Set PTR = PTR → link
  - exit
- end



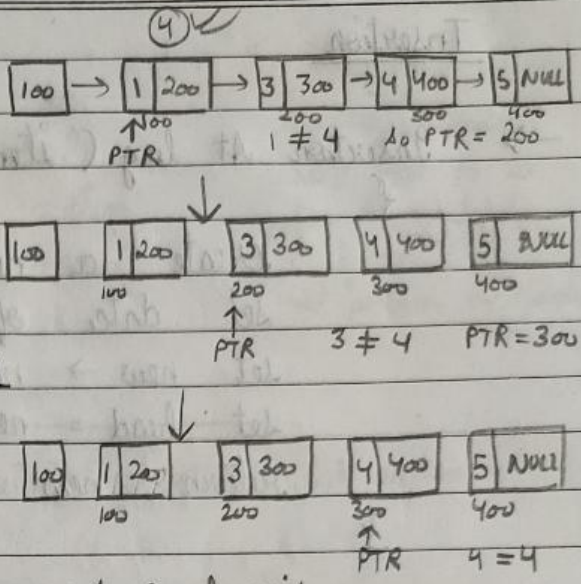


Spiral

Sorting Searching

→ For unsorted ~~arr~~ linked list

```
Linear Search (item)
Set Loc = NULL
Set PTR = head
Repeat while PTR ≠ NULL
        If item = PTR → data
            set Loc = PTR
            exit
        else
            PTR = PTR → link
return Loc
```



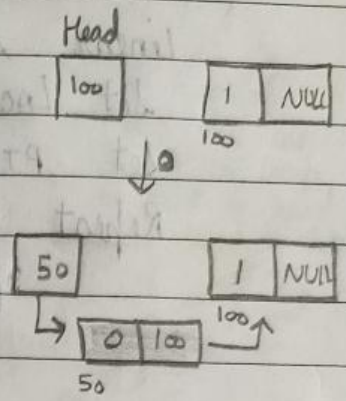
→ For sorted linked list

```
Set PTR = head
Set Loc = NULL
Repeat while PTR ≠ NULL
        If item > PTR → data
            set PTR = PTR → link
        else if item = PTR → data
            set Loc = PTR
            exit
        else
            exit
return loc.
```

Insertion

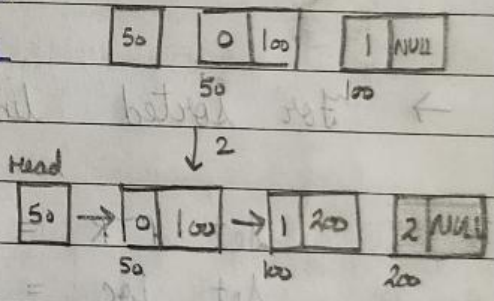
→ Insertion At beg (head, item) TC = O(1)

{
Create a new node - new
Set data of new node = item
Set new → next = head
~~Set head = new~~
return new
}



→ Insertion At end (head, item) TC = O(n)

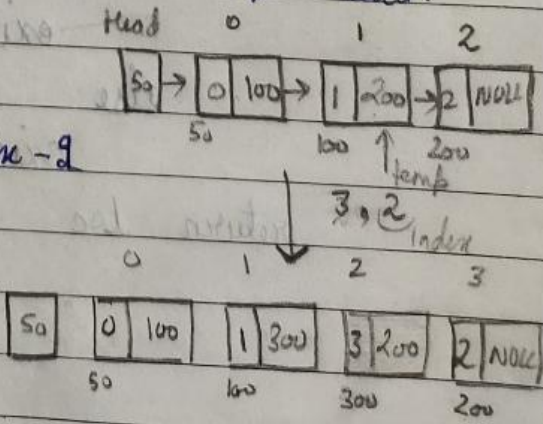
{
~~int~~ Create a pointer variable temp & set temp = head
Create a new node - new
set data of new node = item
Repeat while temp → next ≠ NULL
temp = temp → next
set new → next = NULL
set temp → next = new
return head
}



→ Insertion At Index (item, index) TC = O(n)

{
create a pointer variable temp & set temp = head
create a new node - new
set data of new node = item
Repeat k from 0 to index - 2
temp = temp → next
set new → next = temp → next
set temp → next = new
return head
}

u can check for count 100 if Pos > count
else →



Spiral

For singly list addition / deletion reflects changes in only two nodes.
new node & previous node (in insertion)
node to be del & previous node (in deletion)
But for doubly three nodes are affected.

→ Insert After node (item , node) TC = O(1)
{
Create a new node - new.
set data of new node = item
set new → next = node → next
set node → next = new
return head
}

In this node need not to be found - already given just perform the operation

Can combine all 3 in one - at beg, at end, at index
if (index = 0)
{ new → next = head } → for beg
head = new
else { Repeat temp } → for index / last

Deletion

→ Deletion At beg ( ) TC = O(1)
{ If ~~the~~ head = NULL return NULL else
Create a pointer variable temp.
Set temp = head
Set head = head → next
free (temp)
return head
}

→ Deletion At End ( ) TC = O(n)
{ if head = NULL return NULL else -
Create two pointer variables temp1 & temp2.
set temp1 = head
set temp 2 = head → next
Repeat while (temp2 → next ≠ NULL)
temp1 = temp1 → next
temp 2 = temp2 → next
temp1 → next = NULL
free (temp2)
return head
}

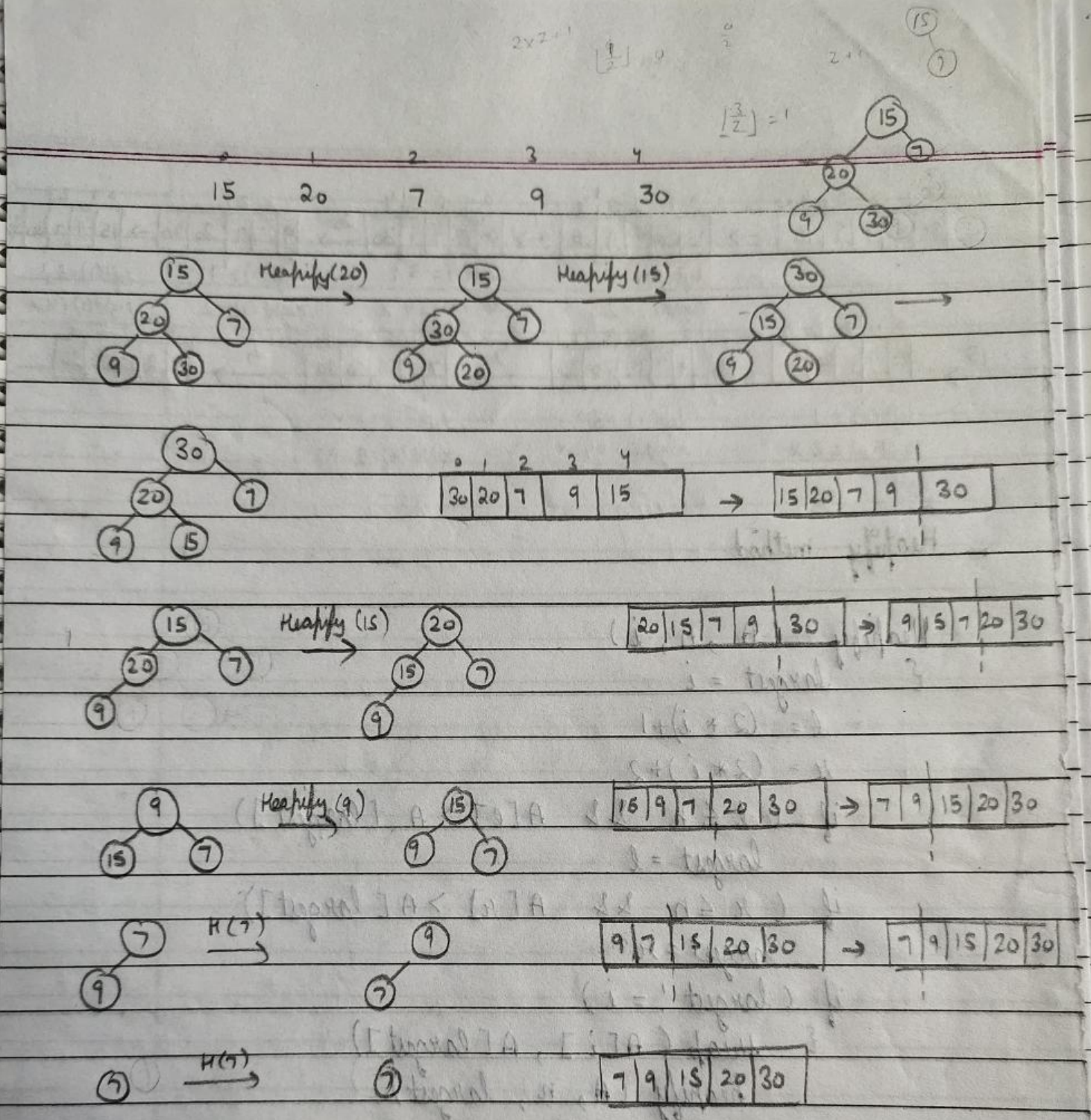
Spiral

Write separate for head & rest
We require two nodes → node we are dealing with & the prev. one
head in case of 1st node

→ Delete at Index (index)
{
Create a pointer variable temp1
Set temp1 = head
Repeat while for K = 0 to index - 2
temp1 = temp1 → next
Creat another pointer variable temp2
Set temp2 = temp1 → next
Set temp1 → next = temp2 → next
free (temp 2)
return head
}

→ Delete Value (value)
{
If head = NULL return NULL
else if (head → data = value) {temp1 = head ; head = head → next ; free(temp)}
else create two pointer variable temp1 & temp2
Set temp1 = head
Set temp2 = head → next
Repeat while (temp 2 → data ≠ value && temp 2 → next ≠ NULL)
temp1 = temp1 → next
temp2 = temp2 → next
If (temp 2 → data = value)
temp1 → next = temp 2 → next
free (temp 2)
else Return NULL
return head
}

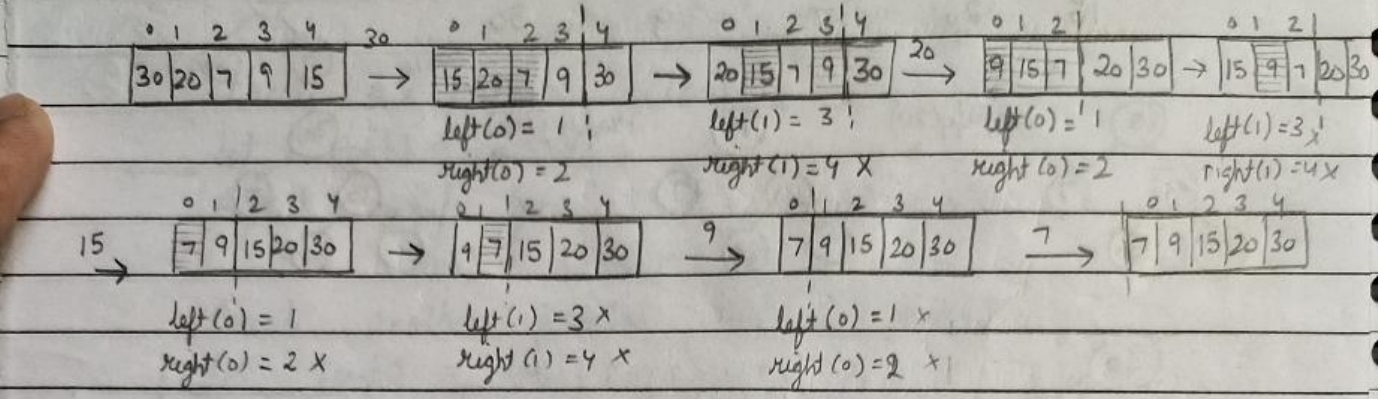
Can combine : temp1 = head
temp2 = head → next
if (index = 0)
{ head = head → next
}
else { loop }
free temp
return head



TC = heapify = ~~O(n)~~ O(logn)

∴ TC = O(nlogn)

nlogn

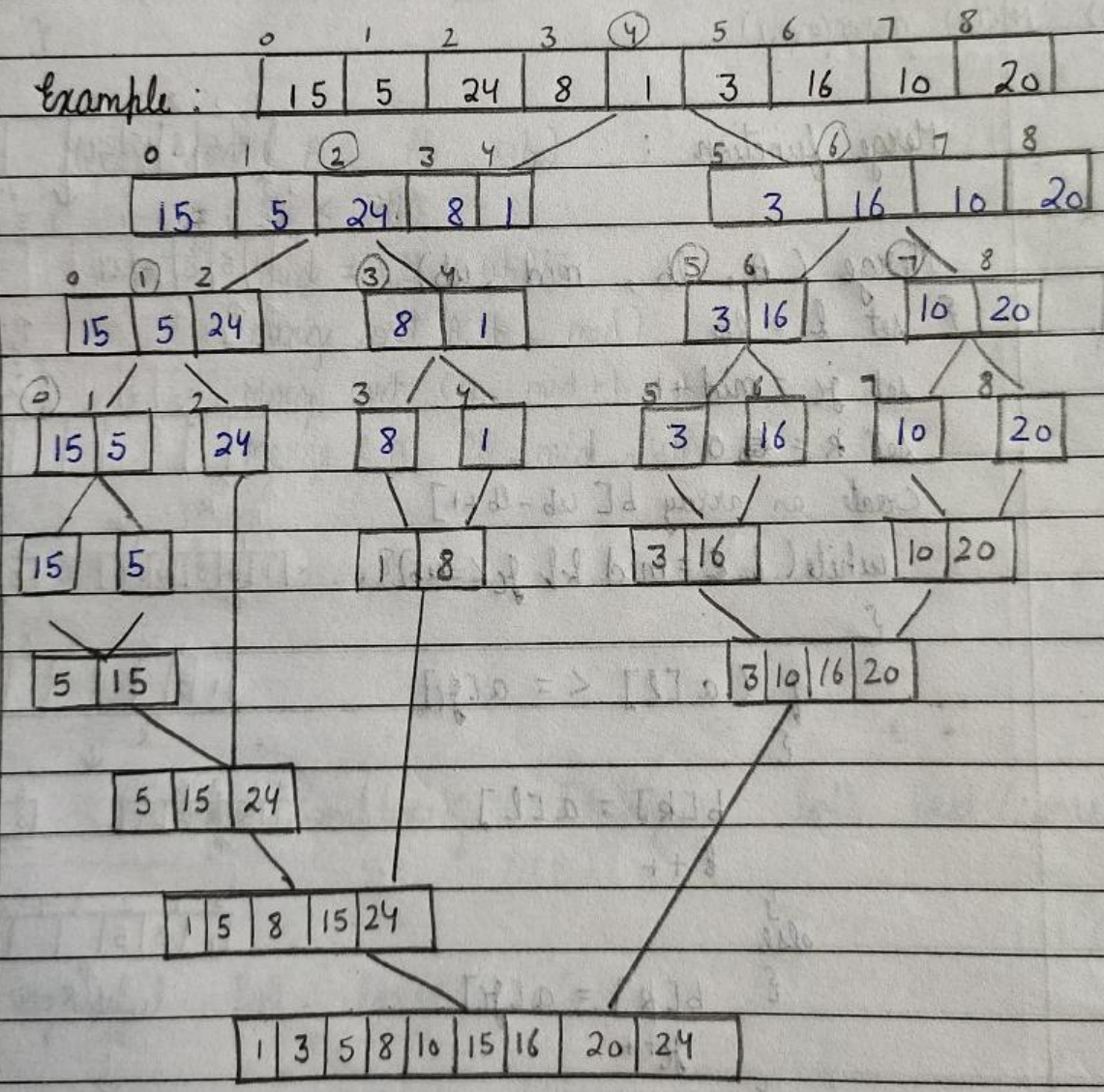


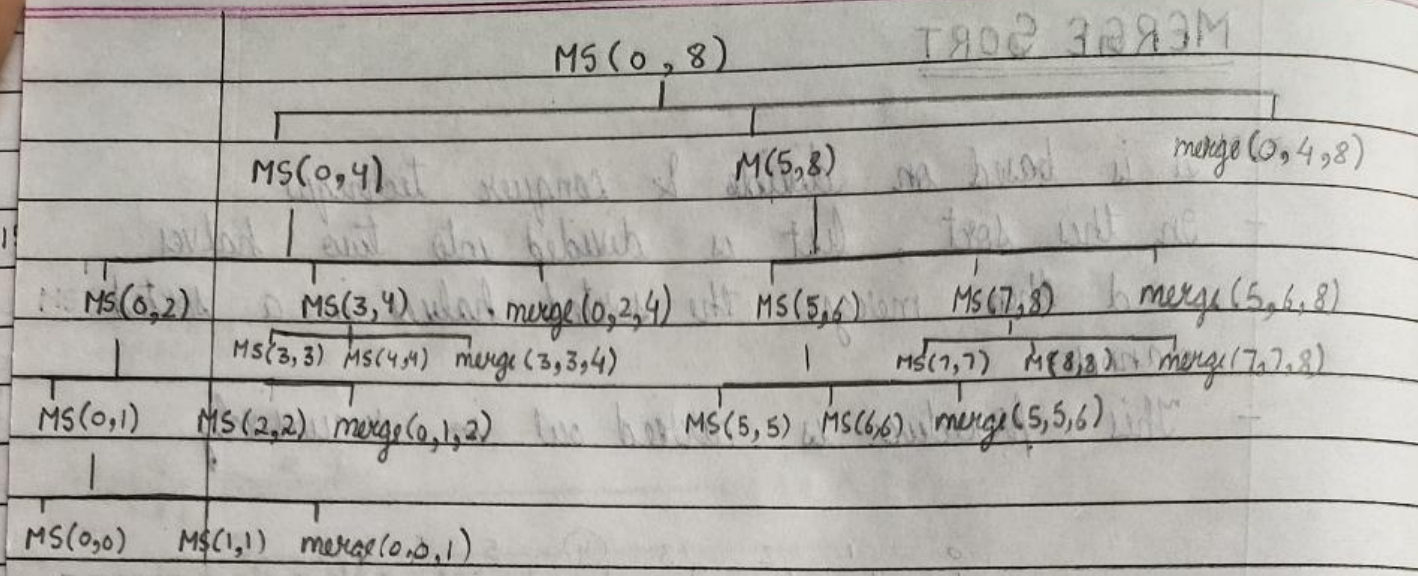
Heapify method

```
Heapify (A, n, i)
{   largest = i
    l = (2 * i)+1
    r = (2*i)+2
    if (l ≤ n && A[l] > A[largest])
        largest = l
    if (r ≤ n && A[r] > A[largest])
        largest = r
    if (largest != i)
      { swap (A[i], A[largest])
        heapify (A, n, largest)
      }
}
Heap_sort (A, n)
{   for (i = n/2 ; i >=0 ; i--)
      { Heapify (A, n, i)
      }
    for (i = n-1; i >=0 , i--)
      { swap (A[i], A[1])
        Heapify (A, n, 1)
}     }
```

MERGE SORT

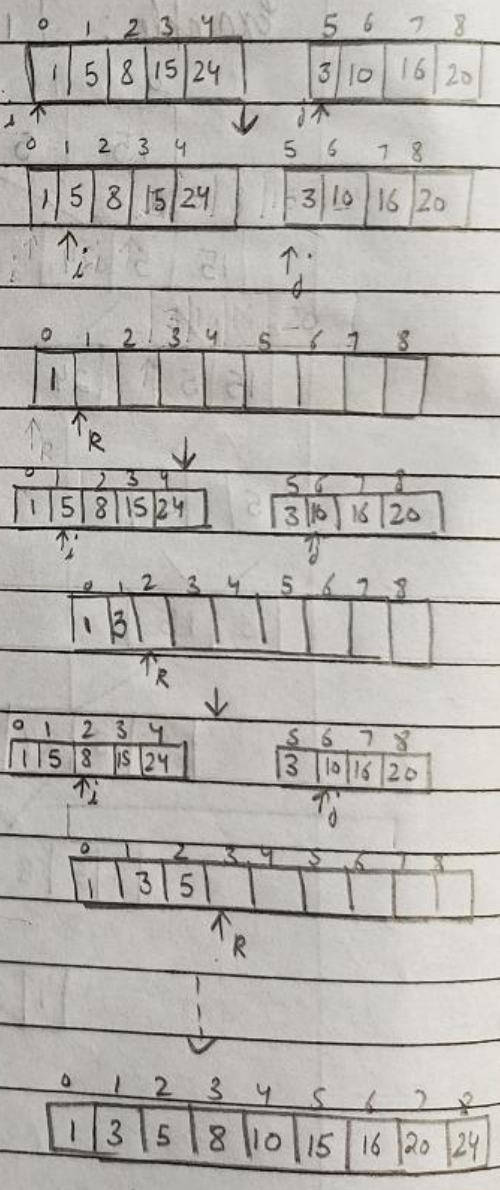
- It is based on divide & conquer technique.
- In this sort, list is divided into two halves and then merges the sorted halves in a sorted manner.
- This procedure is carried out ~~in~~ recursively.





Merge function :

```
Merge ( A, lb, mid, ub)
{ set l = lb
  set jr = mid +1
  set k = 0
  Create an array b[ ub - lb + 1]
  while ( l <= mid && jr <= ub)
  {
     if ( a[l] <= a[jr]
     {
        b[k] = a[l]
        l ++
     }
     else
     {  b[k] = a[jr]
        jr++
     }
     k++
  }
  if ( l > mid )
  {  while (jr <= ub)
     b[k] = a[jr]
     jr++ & k++
  }
```



PTO

```
else
{   while (l <= mid)
    {   b[k] = a[l]
        l++ & k++
    }
}
set j = 0
for (i = lb, i <= ub, i++)
    a[i] = b[j]   & j++
}
```

```
Merge_sort (A, lb, ub)
{   if (lb < ub)
    {   mid = (lb + ub)/2
        Merge_sort (A, lb, mid)
        Merg_sort (A, mid+1, ub)
        merge (A, lb, mid, ub)
    }
}
```

Time complexity $O(n \log n)$ both best / worst

SC $= O(n)$

Adv :
- very useful for large list.

Disadv :
- slower comparatively for smaller lists
- Requires additional memory space of $o(n)$ for merging
- Time complexity is same even if array is already sorted.

Quick sort

based on divide & conquer method. A pivot element is selected & list is partitioned into two sub-lists one of which consists of elements smaller than the pivot element & the other one consists of elements larger than the pivot element

el < pivot	pivot	el > pivot

elements which as equal to the pivot element may come on either side.

Then obtained sublists are further partitioned recursively.

Time Complexity: $O(n^2)$ worst case [ase]
$O(n \log n)$ best case

Algorithm

```
Partition ( A , lb , ub)
{   pivot = A[lb]
    start = lb
    end = ub
    while ( start < end)
    {
        while ( A[start] <= pivot)
            start ++
        while ( A[end] > pivot)
            end --
        if ( start < end)
            swap ( A , start , end)
    }
    swap (A , lb , end)
    return end
}
```

```
Quick_sort ( A , lb , ub)
{
    if (lb < ub)
    {   loc = partition (A, lb, ub)
        Quick_sort ( A , lb , loc-1)
        Quick_sort (A , loc+1 , ub)
    }
}
```

Selection sort :

- In this sort, list is divided into two parts, sorted and unsorted.
- Initially sorted list is empty & unsorted list is the entire list.
- The smallest element is searched from the unsorted list & it swapped with the leftmost element of unsorted list & the element becomes part of sorted list.
- This process is continued till unsorted list is left with only one element.

Advantages : Extra space is not required in-place sorting.
simple algo

disadv. : not suitable for large lists

- For n elements, no. of pass = n-1
- no. of comparisons in pth pass = n - p
- max swaps = n-1 [for desc.
  min — = 0 [for asc.

Time complexity : $O(n^2)$ best / worst both
SC $O(1)$

Bubble Sort :

- In this sort each element is compared to its succeeding element (in multiple passes) positioning the greatest / smallest element at the last position
- In each pass, the largest element bubbles out at the end.
- List - divided - sorted / unsorted start, sorted = empty, unsorted - whole
  bubbled out el is added to the sorted list.
  - For n elements, no. of passes = n-1
    in pth pass no. of comparisons = n-p
  - max. ~~comparisons~~ swaps = $n-1 + n-2 + \cdots$
    $= \frac{n(n-1)}{2}$
    min swaps = 0 (for asc)

Time Complexity : = $O(n^2)$ best / worst)
SC = $O(1)$

For optimised : TC = $O(n)$ best case [asc]
= $O(n^2)$ worst case

Insertion sort :

- In this sort, list is divided into 2 subparts - sorted & unsorted.
- At beg. sorted list has 1 element & rest of the list is unsorted
- One by one all elements of the unsorted list are inserted into the sorted list at their correct position by performing sequential search.

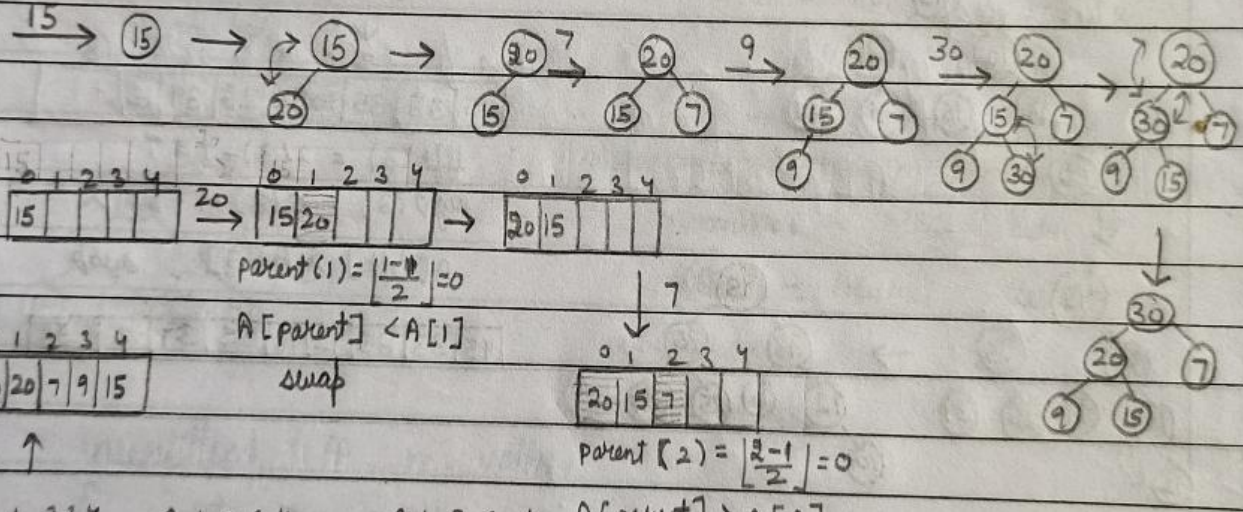
For n elements, no. of passes = n-1

Time Complexity = $O(n^2)$ worst case [desc]
$O(n)$ best case [asc]

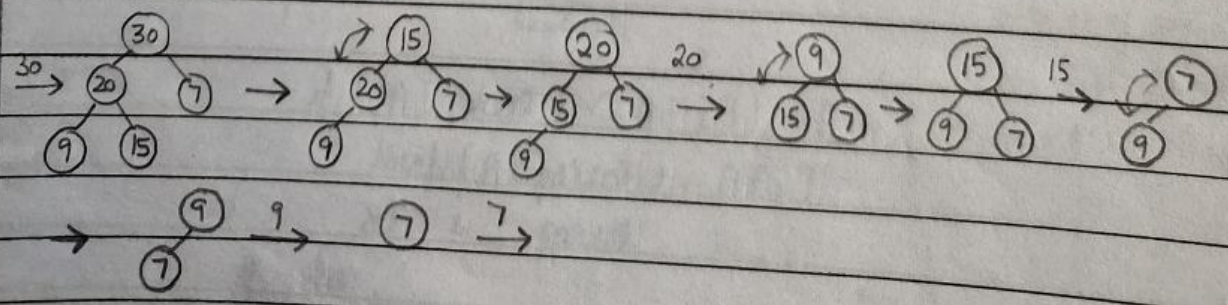
Heap Sort

- This is a comparison based sort.
In this sort, a binary max heap is created from the input data and then root element is deleted one by one. At a given point of time, Root element of a ~~heap~~ binary max heap is always the largest.
∴ Each time the largest element (root node) is deleted & inserted at the end of heap.
So the resulting array is sorted in ascending order.

Example 15 20 7 9 30



Deletion



In complete tree, $\lfloor \frac{n}{2} \rfloor$ to $n-1$ → leaf nodes

Min & Max Heaps

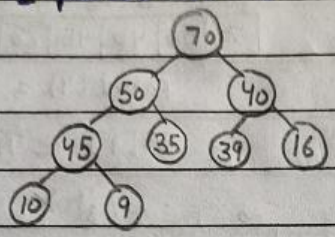
Heap - a complete binary tree

Max Heap:

For every node i, the value of node is less than or equal to its parent node value [except root]

$$A[\text{parent}(i)] \geq A[i]$$

eg:



0	1	2	3	4	5	6	7	8
70	50	40	45	35	39	16	10	9

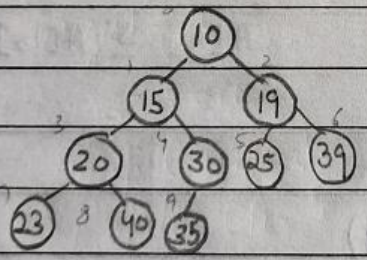
root is max

Min Heap:

For every node i, the value of node is greater than or equal to its parent node value [except root]

$$A[\text{parent}(i)] \leq A[i]$$

eg:

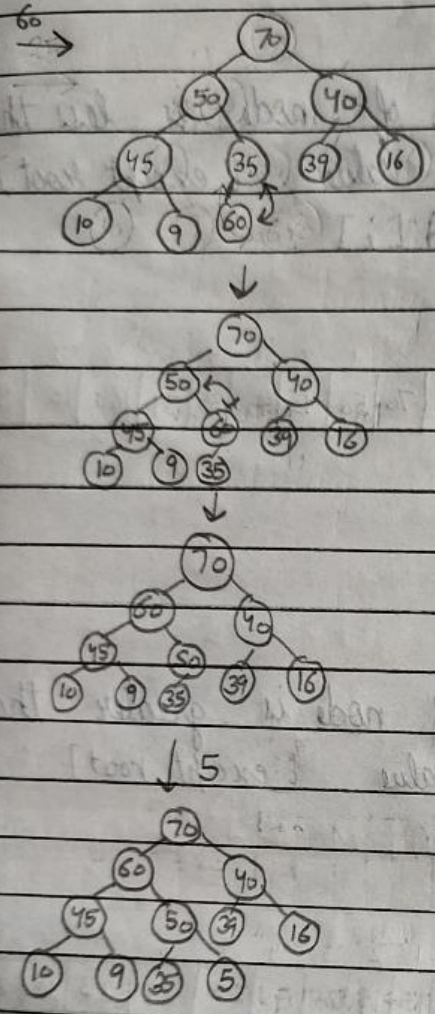


0	1	2	3	4	5	6	7	8	9
10	15	19	20	30	25	39	23	40	35

root is min

$\lfloor \frac{10}{2} \rfloor = 5$ — 9

Max Heap Insertion



↓ insert (9, 60)

0	1	2	3	4	5	6	7	8	9
70	50	40	45	35	39	16	10	9	60

parent (9) = $\lfloor \frac{9-1}{2} \rfloor = 4$

A[4] ≥ A[9] ? No → swap

0	1	2	3	4	5	6	7	8	9
70	50	40	45	60	39	16	10	9	35

parent (4) = $\lfloor \frac{4-1}{2} \rfloor = 1$

A[1] ≥ A[4] ? No swap

0	1	2	3	4	5	6	7	8	9
70	60	40	45	50	39	16	10	9	35

parent (1) = $\lfloor \frac{1-0}{2} \rfloor = 0$

A[0] ≥ A[1] ? Yes

↓ insert (10, 5)

0	1	2	3	4	5	6	7	8	9	10
70	60	40	45	50	39	16	10	9	35	5

Parent (10) = $\lfloor \frac{10-1}{2} \rfloor = 4$

A[4] ≥ A[10] ? Yes.

```
insertHeap ( A , n , value)
{   set  n = n+1
    set  A[n] = value
    set  i = n
    while ( i > 0)
    { set parent = ⌊(i-1)/2⌋

      if ( A[parent] < A[i]) then,
          swap (A[parent], A[i]
          set  i = parent
      else return
    }
}
```

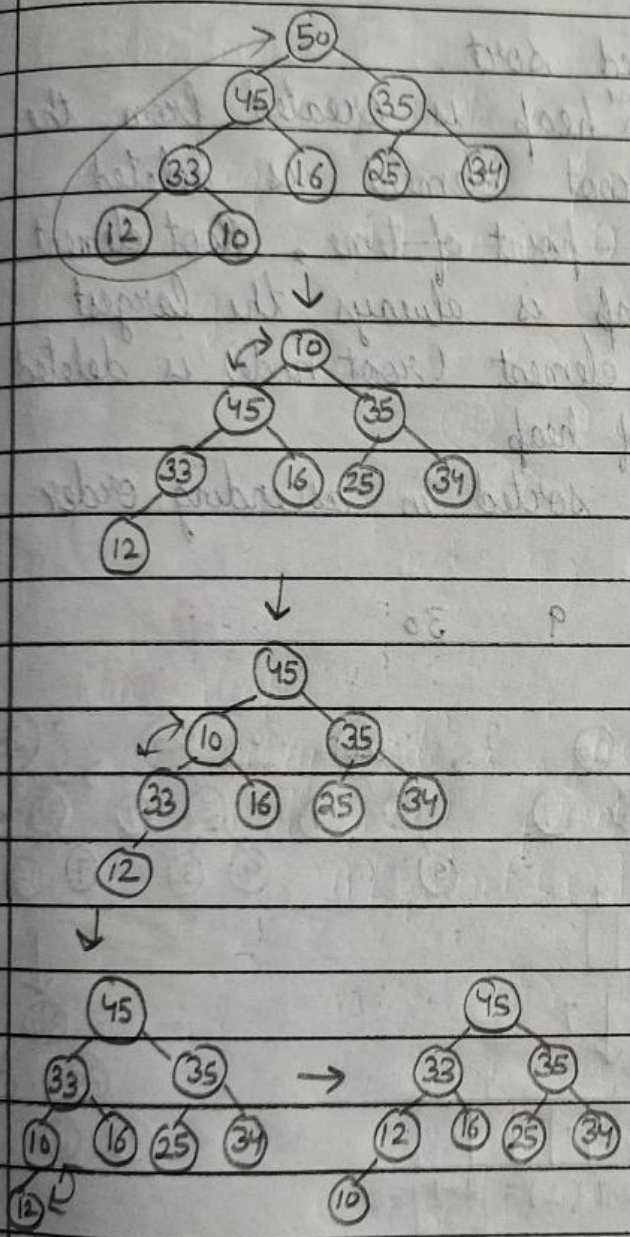
Similar for Min Heap

TC = O(log n) [worst]
bcz no of comparisons ≈ height of Tree

= O(1) [best]

Heap Deletion

always root element is deleted



0	1	2	3	4	5	6	7	8
50	45	35	33	16	25	34	12	10

A[0] = A[8]

n--

delete (50)

0	1	2	3	4	5	6	7
10	45	35	33	16	25	34	12

left (0) = 2(0)+1 = 1

right (0) = 2(0)+2 = 2

A[left] > A[right]

A[left] > A[0] swap

0	1	2	3	4	5	6	7	8
45	10	35	33	16	25	34	12	

left (1) = 2(1)+1 = 3

right (1) = 2(1)+2 = 4

A[left] > A[right]

A[left] > A[1] swap

0	1	2	3	4	5	6	7	8
45	33	35	10	16	25	34	12	

left (3) = 2(3)+1 = 7

right (3) = 2(3)+1 = 8 X

A[left] > A[3] swap

45	33	35	12	16	25	34	12	50

3 pointers req (temp)
move any one & obtain the rest with prev & next

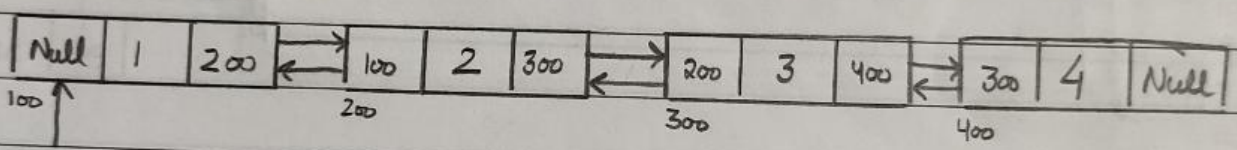
You can maintain tail pointer also along with head

Doubly Linked List

A linked list in which each node stores the data, address of previous node & the address of next node.

Node : ← | Previous node address | Data | Next node address | →

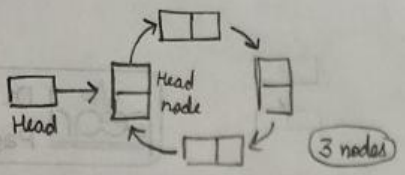
- Traversal in both direction
- extra memory required (drawback)



Head | 100 |

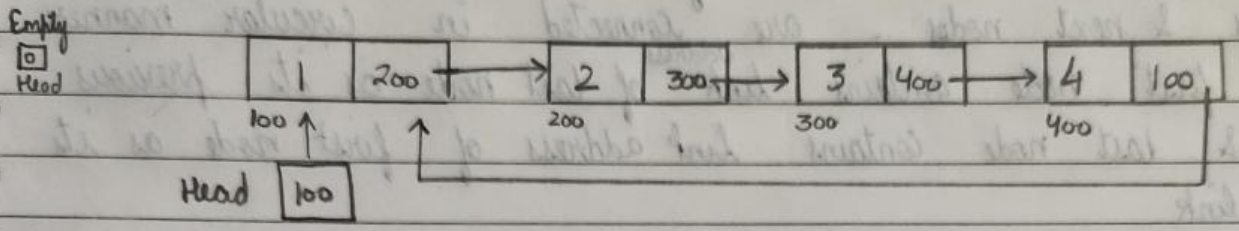
If next node = Null → last node
prev node = Null → First node (head points to it)

Three nodes involved :
Prev. node : [head in case of 1st node (req)]
req node :
next node : [NULL in case of last (req)] cond required

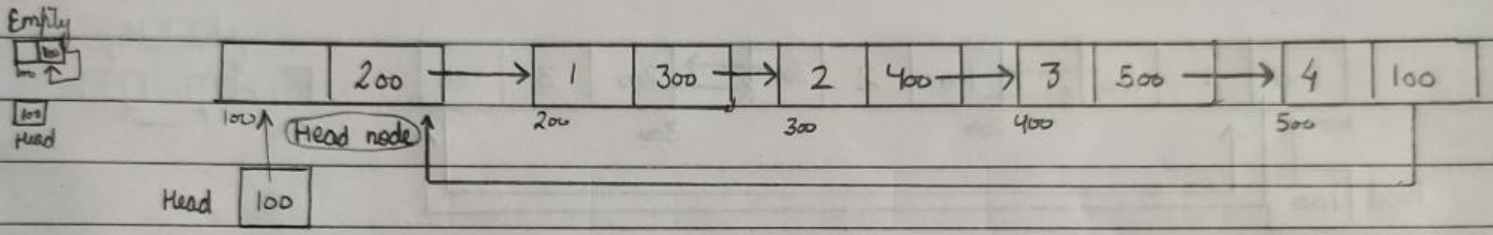


Circular Linked List

Linked list in which last node contains the link of 1st one.

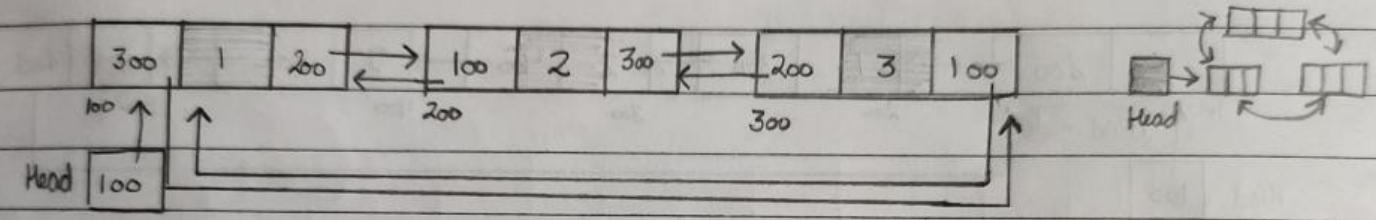


or

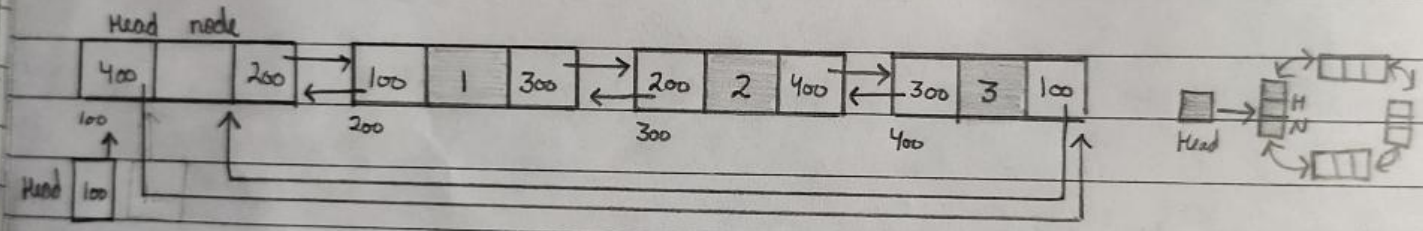


Doubly Circular linked list

A LL in which nodes, containing previous data & links to previous & next nodes, are connected in circular manner where first node contains link address of last node as its previous link & last node contains link address of first node as its next link



or



➤ Traversal in both directions in circular manner.

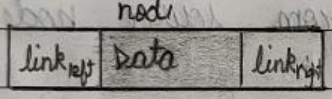
TREES

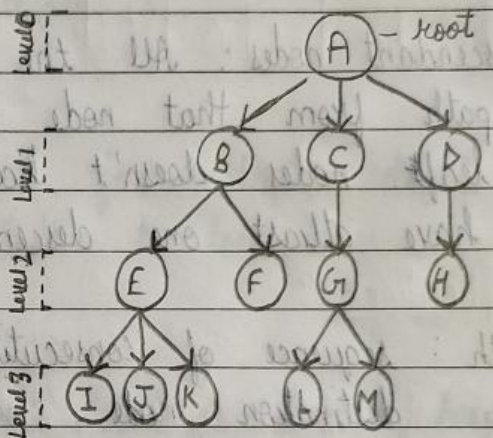
It is a non-linear data structure which consists of nodes & linked together in the form of hierarchy.

- top-down approach i.e. unidirectional.
  consists of multiple levels (non-linear)

Terms:

- Node: node consists of data item and links to other nodes (child nodes)





Tree

- Root node: The topmost node in the tree. (begin of hierarchy)
  - A tree has only one root.
  - It doesn't have any parent A

- parent node: the immediate predecessor of a node is called its parent node.
  - A child can have only one parent
- Child node: the immediate successor of a node is called its child node.
  - A parent can have multiple children.

- leaf node: node which doesn't have any child node (end of hierarchy)
  (external node) bottom-most node in a tree.
  A tree can have multiple leaf nodes. F, H, I, J, K, L, M

1

Spiral

A tree can have common ancestor but not descendant

- Non-leaf node: A node which has atleast one child node.
  - also called internal node. A, B, C, D, E, G
- Ancestor nodes: All the predecessor nodes that fall in the path from the root node to that node.
  - root node doesn't have any ancestor. Rest all nodes have atleast one ancestor.
- Descendant nodes: All the successor nodes that fall in the path from that node to the leaf nodes.
  - leaf nodes doesn't have any descendant. Rest all nodes have atleast one descendant.
- path: sequence of consecutive edges from source node to destination node.
  eg: path from A to I : AB → BE → EI
  it is unique.
- edges: links between two nodes
  eg AB , BE
- subtree: A node with all its descendants in a tree forms the subtree of a tree.
- siblings: The nodes that have same parent.
- Degree of a node: total no. of children of that node.
  degree of leaf node = 0
  no. of parents always 1 (except root node=0)
- Degree of a tree: max. degree in a tree.

- Depth of a node: length of path from root node to that node.
  - length of 1 edge = 1 unit.
    ∴ depth of node = no. of edges between root node & that node.
  - depth of root node = 0.

- depth of a tree: length of path from root node to the deepest leaf node i.e. max depth. (depth of deepest leaf node)

- Height of a node: length of path from that node to the deepest leaf node i.e. longest path to leaf node.
  ∴ height of leaf nodes = 0.

- height of a tree: height of root node i.e. max height.

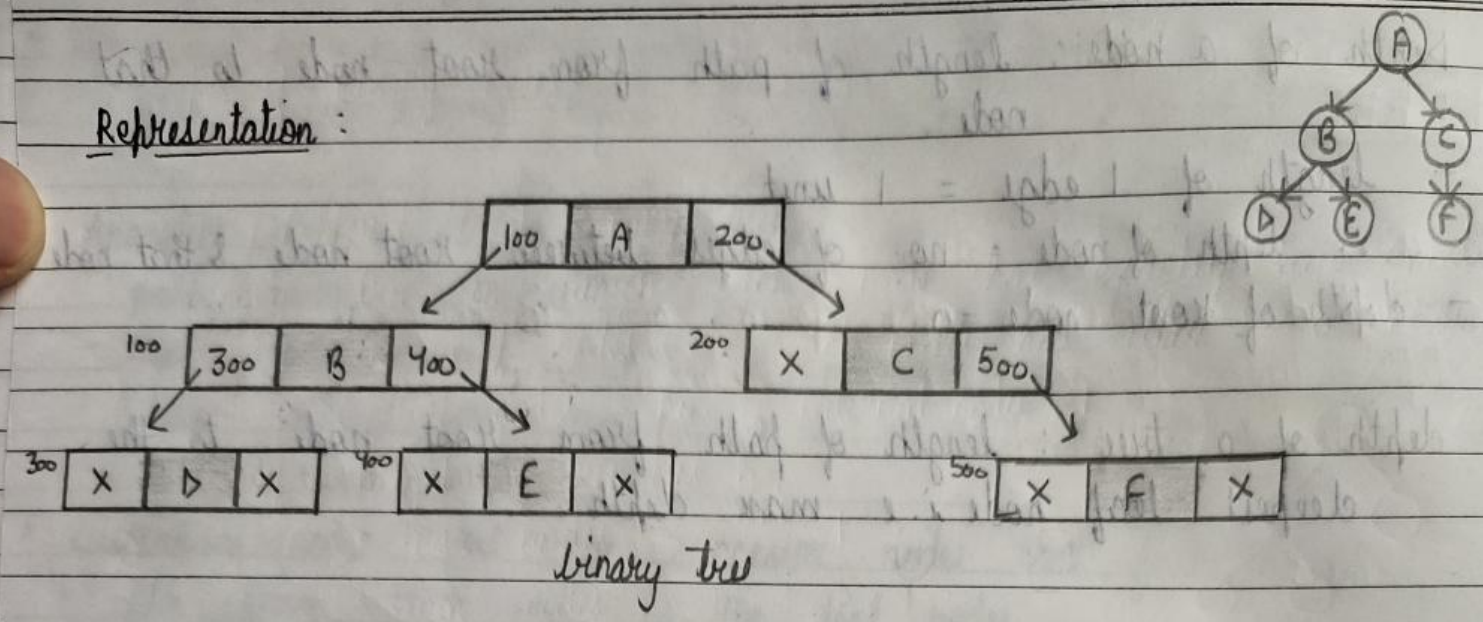
- Level of a node: generation of node from root node.

- level of trees: total no. of levels present in a tree.

A node ~~can~~ have a common ancestor but not descendant in a tree.

height of a tree = level of tree = depth of tree
level of node = depth of node
If a tree contains n nodes, then
no. of edges = n-1 (1 edge for each node except root node)
if no of edges > n-1 → cyclic (there are more than 1 edge for any node)

Representation :



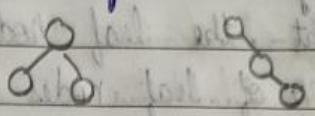
binary tree

Applications :

- In the file system, stored on the disc drive, file and folder are stored in the form of trees. (natural hierarchy)
-

Binary Tree degree <= 2

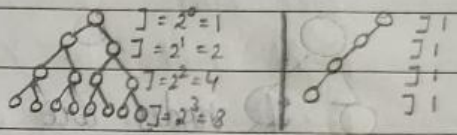
- A tree which has atmost two children for each node.
- They are named as
  - left child
  - right child



Node :

left link	Data	Right link

At nth level, • max nodes = 2^n
• min nodes = 1



If height of a tree is h,
• max nodes = $2^h + 2^{h-1} + 2^{h-2} + \cdots + 1 = \frac{1(2^{h+1}-1)}{2-1} = 2^{h+1} - 1$
[height = level]

• min nodes = $h + 1$

If nodes in a tree are n
max height = $n - 1$
(when min nodes)

min height $\Rightarrow$ $n = 2^{h+1} - 1 \Rightarrow \log_2(n+1) = \log_2 2^{h+1}$
(when max nodes)

$\Rightarrow h = [\log_2(n+1)] - 1$

At same height h of a tree, min nodes$_{perfect}$ > min nodes$_{complete}$ > min nodes$_{full}$ > min nodes$_{deg.}$

For n nodes, height$_{full}$ > height$_{complete}$ > height$_{perfect}$

Dt. Page

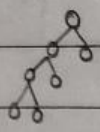
Types of Binary trees

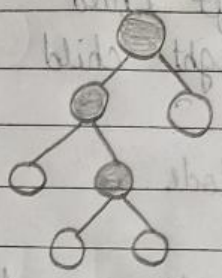
- Full / Proper / Strict Binary tree:

  Binary tree in which each node has exactly two children except the leaf nodes. (i.e. each nodes has either 2 or 0 children)
  (i.e. all internal nodes)

  - no of leaf nodes = no. of internal nodes + 1
  - If height of tree is h,
    - max nodes = $2^{h+1} - 1$
    - min nodes = 1, 3, 5, 7 ... $2h + 1$ ($a+(n-1)d$, $n = h+1$, $1+(n-1)2$)

      ($2h$: 2 at each level; 1: root)

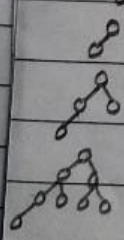
  

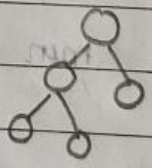
  - If nodes are n,
    - max height = $(n-1)/2$
    - min height = $[\log_2(n+1)] - 1$

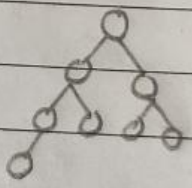
- Complete Binary tree:

  Binary tree in which every level is completely filled except possibly the last one in which nodes are as left as possible.

  - If height of tree is h,
    - max nodes = $2^{h+1} - 1$
    - min nodes = 2^h

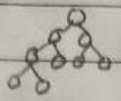
  

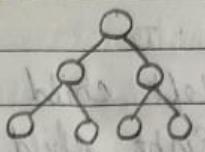
  

  - If nodes are n,
    - max height = $\log_2 n$
    - min height = $[\log_2(n+1)] - 1$

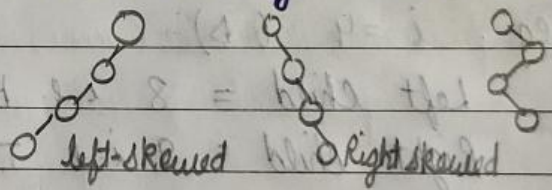
- Perfect Binary tree :
  Binary tree in which all the interior nodes have two children and all leaf nodes are at same level.
  - A perfect tree is always a full as well as complete tree.
    vice versa not true





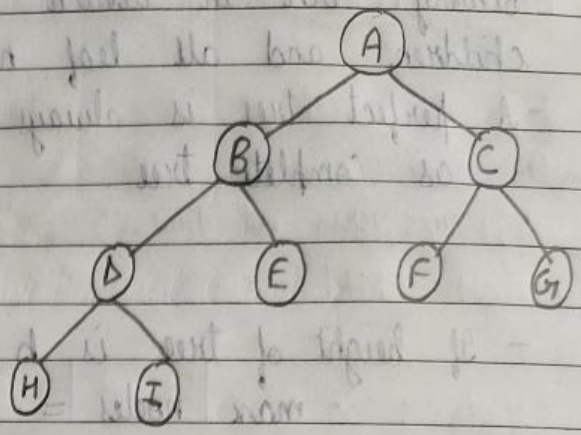
  - If height of tree is h,
    - max nodes = $2^{h+1} - 1$
    - min nodes = $2^{h+1} - 1$
  - If nodes are n,
    - max height = $[\log_2(n+1)] - 1$
    - min height = $[\log_2(n+1)] - 1$

- Degenerate tree :
  Binary tree in which each nodes has only 1 child node except the leaf node.
  - behaves like a linked list.



  - If height of tree is h,
    - max nodes = h+1
    - min nodes = h+1
  - If nodes are n,
    - max height = n-1
    - min height = n-1

Array Representation of Binary Tree (Sequential Representation)



If a node is at i^{th} index,

Case : I

- Left child is at : $[(2 * i)+1]$
- Right child is at : $[(2*i)+2]$
- Parent is at : $\lfloor \frac{(i-1)}{2} \rfloor$

eg: $i = 4$ (E)

Left child = 9 X

Right Child = 10 X

Parent = 1 i.e. B

Case : I

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

Fill one by one each level from left to R (must be complete)

Case : II

- Left child is at : $(2 * i)$
- Right child is at $[(2*i)+1]$
- Parent : $\lfloor \frac{i}{2} \rfloor$

Case : II

A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9

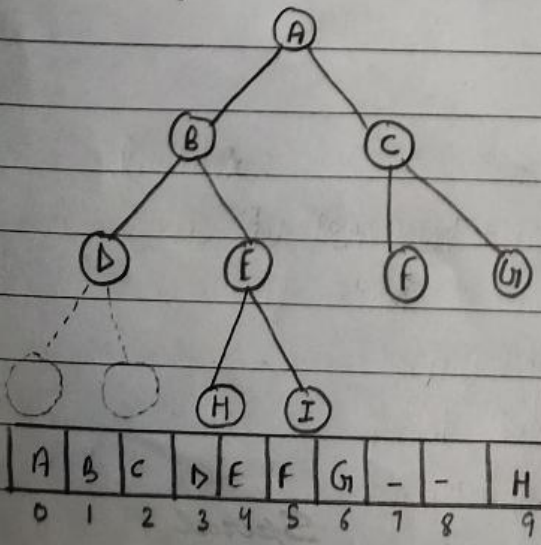
eg: $i = 4$ (D)

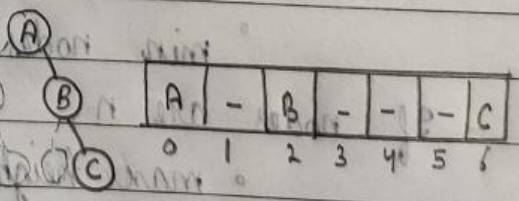
Left child = 8 i.e. H

Right child = 9 i.e. I

Parent = 2 i.e. B

→ If tree is not complete binary tree make it complete



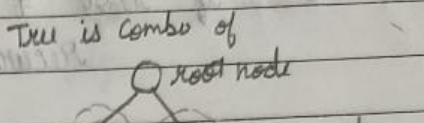


for skewed right - space wastage

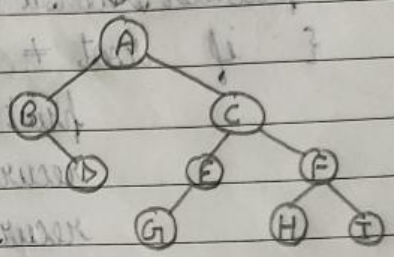
TRAVERSAL OF TREES
(inorder / preorder / postorder)

Linear data structures have only one way of traversal.
But trees can be traversed in different ways

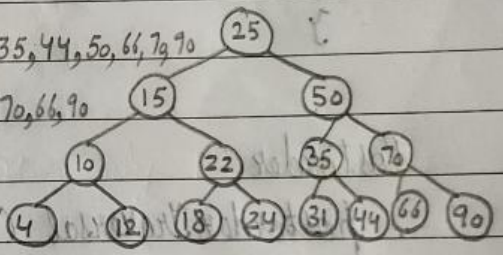
Inorder : Left Root Right
Preorder : Root Left Right
Postorder : Left Right Root



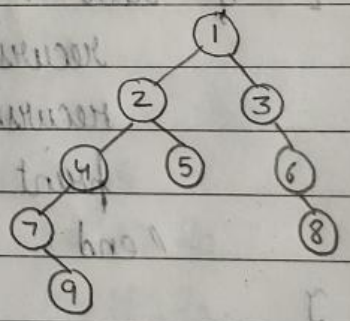
Example: Inorder : B D A G E C H F I root at mid
Preorder : A B D C E G F H I root at first
Postorder : D B G E H I F C A root at last



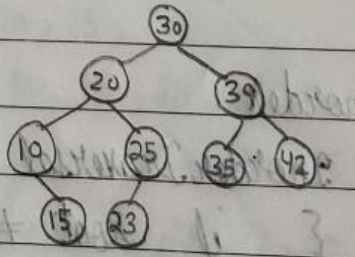
Inorder : 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90
Preorder : 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90
Postorder : 4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



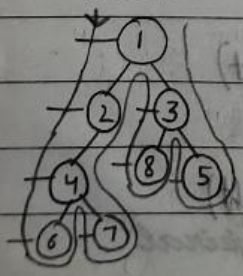
Inorder : 7 9 4 2 5 1 3 6 8
Preorder : 1 2 4 7 9 5 3 6 8
Postorder : 9 7 4 5 2 8 6 3 1



Inorder : 10, 15, 20, 23, 25, 30, 35, 39, 42
Preorder : 30, 20, 10, 15, 25, 23, 39, 35, 42
Postorder : 15, 10, 23, 25, 20, 35, 42, 39, 30

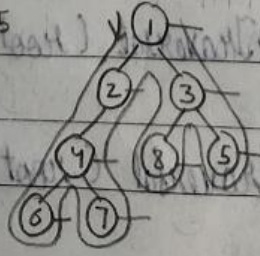


Preorder



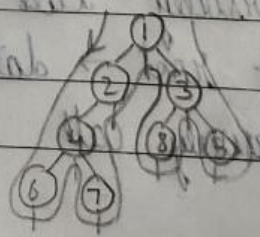
12467385

Postorder



67428531

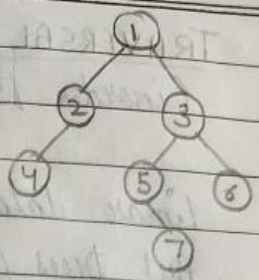
Inorder



64721835

Spiral

Preorder:
preorderTraversal (Tree root)



Preorder:
preorderTraversal (root)
{ if root ≠ null, then
Recursively call preorderTraversal (root →

Preorder: O(n)
preorderTraversal (root) each node is visited once
{ if root ≠ null, then
print root → data
recursively call preorderTraversal (root → left)
recursively call preorderTraversal (root → right)
// end
}

Postorder: O(n)
postorderTraversal (root)
{ if root ≠ null, then
recursively call postorderTraversal (root → left)
recursively call postorderTraversal (root → right)
print root → data
// end
}

Inorder: O(n)
InorderTraversal (root)
{ if root ≠ null, then
recursively call InorderTraversal (root → left)
print root → data
recursively call InorderTraversal (root → right)
// end
}

Spiral

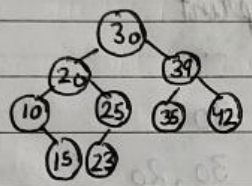
Without recursion

O(n)

→ Inorder (root)

1) Create an empty stack S
2) Initialise current node as root
3) Push current node to S & set current = current → left until current = NULL
4) If current is NULL and stack is not empty
   a) pop the item from stack.
   b) Print popped item, set current = popped item → right
   c) go to step 3.
5) When current is NULL & stack is empty exit.

Example :

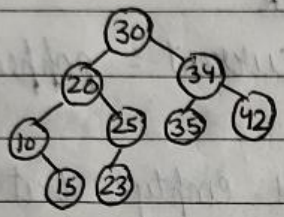


Current	Stack	Print
30	–	–
20	30	–
10	30, 20	–
NULL	30, 20, 10	–
15	30, 20	10
NULL	30, 20, 15	10
NULL	30, 20	10, 15
25	30	10, 15, 20
23	30, 25	10, 15, 20
NULL	30, 25, 23	10, 15, 20
NULL	30, 25	10, 15, 20, 23
NULL	30	10, 15, 20, 23, 25
39	–	10, 15, 20, 23, 25, 30
35	39	"
NULL	39, 35	"
~~42~~ NULL	39	10, 15, 20, 23, 25, 30, 35
42	–	Spiral , 39
NULL	42	10, 15, 20, 23, 25, 30, 35, 39, 42
NULL	–	

→ Preorder (root) O(n)

1) Create an empty stack & push root node to it.
2) Do while stack is not empty:
   a) pop an item from stack & print it.
   b) push right child of popped item to stack
   c) push left child of popped item to stack

eg:



stack	print
30	–
~~34, 20~~	30
34, 25, 10	30, 20
34, 25, 15	30, 20, 10
34, 25	30, 20, 10, 15
34, 23	30, 20, 10, 15, 25
34	30, 20, 10, 15, 25, 23
42, 35	30, 20, 10, 15, 25, 23, 34
42	30, 20, 10, 15, 25, 23, 34, 35
–	30, 20, 10, 15, 25, 23, 34, 35, 42

Postorder (root) $O(n)$

1 Create an empty stack & set current to root

2.1 Do while current is not NULL

a) Push current's right child & then current to stack.

b) set current as current's left child.

2.2 Pop an item from stack & set it as current

a) If popped item has a right child & right child is at top of stack, remove right child from stack, push current back & set current as current's right child

b) Else print current root's data & set current root as NULL

3 Repeat step 2.1 & 2.2 while stack is not empty.

Current	Stack	Print
30	–	
20	39, 30	–
10	39, 30, 25, 20	
NULL	39, 30, 25, 20, 15, 10	
10	39, 30, 25, 20, 15	
15	39, 30, 25, 20, 10	
NULL	39, 30, 25, 20, 10, 15	
15	39, 30, 25, 20, 10	
NULL	39, 30, 25, 20, 10	15
10	39, 30, 25, 20	15, ~~10~~
NULL	39, 30, 25, 20, ~~15, 10~~	15, 10
20	39, 30, 25, ~~20~~, ~~15~~	15, 10
25	39, 30, 20	
23	39, 30, 20, 25	
NULL	39, 30, 20, 25, 23	
23	39, 30, 20, 25	
NULL	39, 30, 20, 25	15, 10, 23
25	39, 30, 20	
NULL	39, 30, 20	15, 10, 23, 25

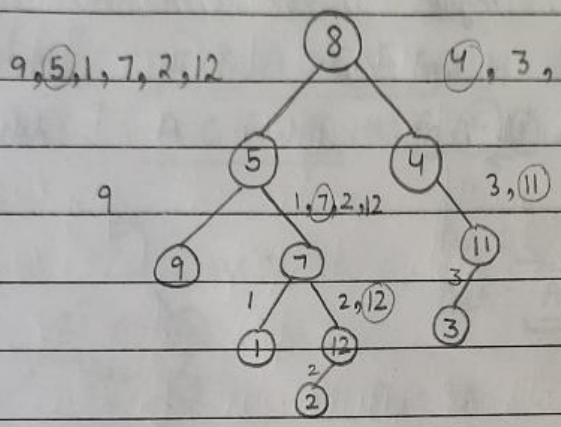
Spiral

20	39, 30	15, 10, 23, 25
NULL	39, 30	15, 10, 23, 25, 20
30	39	
39	30	
35	30, 42, 39	
NULL	30, 42, 39, 35	
35	30, 42, 39	
NULL	30, 42, 39	15, 10, 23, 25, 20^ 35
39	30, 42	
42	30, 39	
NULL	30, 39, 42	
42	30, 39	
NULL	30, 39	15, 10, 23, 25, 20^ 35, 42
39	30	
NULL	30	15, 10, 23, 25, 20^ 35, 42, 39
30	-	
-	-	15, 10, 23, 25, 20^ 35, 42, 39, 30

★ Construct a binary tree from post-order & Inorder

Postorder : 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

Inorder : 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

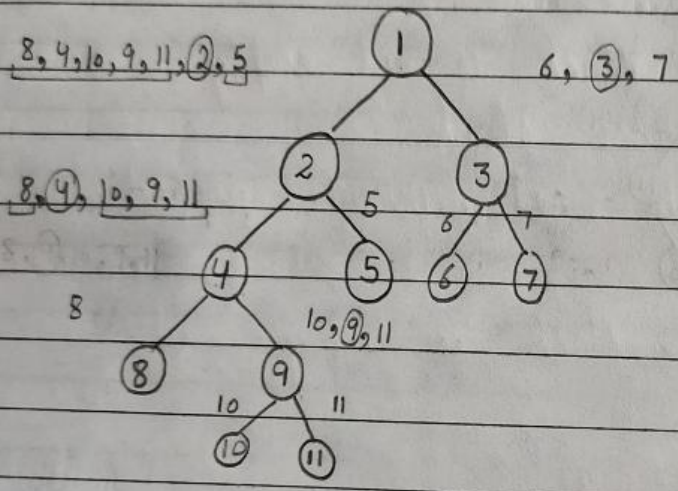


★ Construct binary tree from preorder and Inorder

Preorder : 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 → to decide root

Inorder : 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 → to find left/right

Left | Right



If obtained tree is not full binary tree – it is not unique
If ____ is full ____ – it is unique

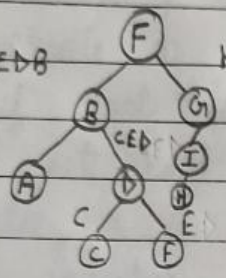
Dt. Page

* Construct binary tree from preorder and postorder

It is not possible to construct unique binary tree but a unique full binary tree can be constructed.

Repeat recursively

Preorder : F B A D C E G I H
Postorder : A C E D B H I G F



Pre : B A D C E
Post : A C E D B

Pre : G I H
Post : H I G

Pre : A
Post : A

Pre : D C E
Post : C E D

Pre : I H
Post : H I

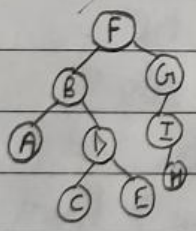
Pre : C
Post : C

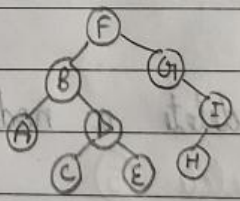
Pre : E
Post : E

To find pre → see post
post → see pre

---

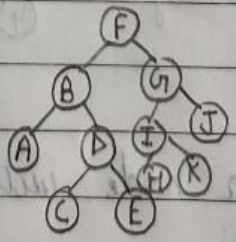
directly : Traverse preorder L → R, locate first root to the current el RHS
First fill left position, if occupied → right position
* All elements to the left of a root in postorder are its children





This tree also have same pre & postorder
So unique BST is not possible

→ Pre : F B A D C E G I H K J
Post : A C E D B H K I J G F



This is unique

We can also traverse Post order, locate root on LHS & fill right pos. first
All el to the right of a root in pre are its children

An element occurs only once in a binary search tree so every el has its particular place in BST.

Dt. Page

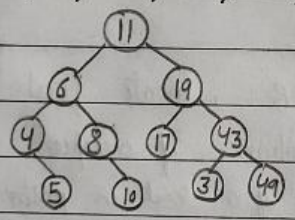
Binary Search tree

binary tree with following properties:
- Left subtree of a node contains nodes with lesser values.
- Right subtree of a node contains node with greater values.
- Left and right subtrees are also BST

inorder traversal : ascending order [To check whether BST or not]
Left most leaf node : smallest
Right most leaf node : largest
no duplicate values

→ Insertion

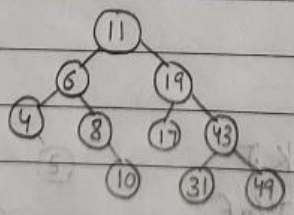
11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



→ Deletion

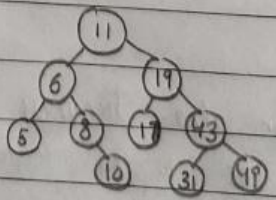
Case: 1 delete a node with 0 child

delete (5)
delete directly



Case: 2 delete a node with 1 child

delete (4)
Replace the node with child node



Case: 3 delete a node with 2 children

2 ways : replace with

- inorder predecessor (largest on the left)
- inorder successor (smallest in the right)

Spiral