17/9/19

Page No.: 1
Date:
YOUVA

DATA STRUCTURES

1] LINKED LISTS → Physical DS

- if in Stack, programmer needs to hard code size of array.
- if in Heap, size needs to be pre-decided before working with Array. (imp)

Static DS → Array [We can't extend the size of same Array]

→ memory efficient (oper^n depend)

* Linked List	Array
- uses more Space	- less space.
- only in Heap	- stack / heap
- dynamic	- static.

[Efficiency of Array/LL depends on oper]

(i) Singly linked list

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

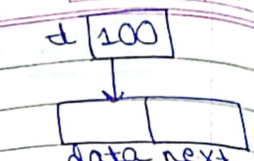
→ Self referential Structure

OR

```
t = new node;
t = (struct node *) malloc (sizeof (struct node));
```

t->data = 10;
t->next = NULL;

structure pointer.



imp

* while(p != NULL) -> if(p)
  while(p == NULL) -> if(!p)

eg -> Pg 52 (Rev Copy)

eg -> Pg 112 (Display of SLL) (1) -> iterative, recursive

eg -> Pg 54 [max]

imp

```
int fun(struct node *p)
{
  int x = 0;
  if(p)
  {
    x = fun(p->next);
    return p->data > x ? p->data : x;
  }
  return 0;
}
```

(2) * Search: [Linear Search] -> O(n) time

eg -> Pg 117 [run code & check]

[Frq accessed items to first]

(3) Reverse [use 3 pointers trailing] imp

=> Traversing/modifying links is much faster & cheaper than data movement imp

imp

eg -> Pg 120 (Recursive Reverse) Call

[Trace & See]

* INSERTION IN SLL

(i) Insert before Head: O(1) [1 Pointer]

(ii) Insert node at given Position: [2 Pointers] O(n) Time

(iii) Appending node at last Position.

```
while(p->next != NULL)
{ p = p->next;
}
```

1 Pointer

```
p->next = newnode;
p = p->next;
p->next = NULL;
```

reached last node.

* DELETION IN SLL:

(i) delete first node: [1 Pointer]

(ii) delete any node:

node to be deleted.

```
p->next = q->next;
free(q);
```

imp

2 Pointers

[or use memory leak]

-> p->next = p->next->ne

(All) [in DLL]

Q. If p is pointing to same node in LL, & we want to delete that node.

imp → Time → O(n) [in SLL].

→ We must go till Prev.

→ Binary Search not quite efficient in linked list [in Array, it is].

eg → Pg 59 (Detect loop in LL) imp

p → 1 step
q → 2 steps

2 Pointers req. O(n)

(iii) Circular linked list

head → 6 → 4 → 9 → 8 → 7

① Display →

```
p = head;
do
{
  printf("%d", p->data);
  p = p->next;
} while (p != head);
```

② Insertion

(i) before head : 1 pointer [O(n)]

(ii) after given position : 2 pointers (Same as SLL)

eg → Pg 61 (Rev Copy)

③ Deletion :

(ii) delete any node → 2 pointers

(iii) delete head node → 1 pointer

(iii) delete last node → 1 pointer

O(n) Time

imp [Text] (little modified)

eg → Pg 62 (Rev Copy) [Linked Circular Queue]

(iii) Doubly linked list

- cost increases, but flexibility increases

Head 100

10	5	200	⇄	100	8	300	⇄	200	3	10
100				200				300		

eg → Pg 131 [using flags] imp

Flags can be used to detect loops in O(n) Time. [no need to take 2 pointers].

① Insertion :

(i) Start : 1 pointer ⇒ O(1) time.

(ii) after given position :

2 pointers
O(n) Time.

eg → Pg 64 (Rev copy)
(links modified → 4)

imp

* If p points to any element in DLL, & we need to delete that element
→ O(1) time.

imp

```
node *t = p->prev;
t->next = p->next;
p->next->prev = t;
delete(p)
```

[using 1 extra Pointer]

For concatenation :
SLL → O(n)
C-SLL → O(n)
DLL → O(n)
C-DLL → O(1) → most used LL

* Circular doubly linked list is the most efficient linked list used

[used in Undo operation]

2 STACK → Logical DS

- Stack is an ADT → data present
→ Operations on data
- [Array, LL] = Physical DS
[Stack, Queue, tree, Graphs] = logical DS
imp
- Logical DS are implemented using physical DS.
- Stack works on LIFO
- Definition of user defined data type = ADT
{ Stack → Space, top
→ push(x), pop(), peek(pos), stackTop(), isFull(),

* Implementation of Stack using Array

(i) Initial : top = -1
(ii) Empty : if(top == -1);
(iii) Full : if(top == size-1);
(iv) void push(int x) → O(1)

```
{
  if(top == size-1)
    pf("Stack overflow");
  else
  { top++;
    S[top] = x;      or, stack[++top] = x;
  }
}
```

O(1)

(v) int pop()
{ int x=0;
if (top==-1)
pf("Stack underflow");
else
{
x=S[top]; OR, x=S[top--];
top--;
}
return x;
}

→ will be mentioned in ADT.

(vi) Peek: returns the element, if pos entered.

index = (top - position) + 1 (imp)

eg → Pg 67 (Peek eg) (Rev Copy)

Stack push(stack S, int x)
Stack pop(Stack S)

→ takes Stack & element as argument, pushes the element, & returns Stack.

* 2 Stack implementation using 1 Array

S	1	8	7	4	3	6	1	1	2	3

t_1 t_1 t_2 t_2

Test

empty: if (t_1==-1) && (t_2==-1)

imp full: if (t_1==t_2-1)

eg → Pg 142 (Imp Q)

3 Tap 4 board

* Valid Permutation Problem

eg → Stack Q3 & Q5 [Pg 143]

No of Valid Permutations = $\frac{^{2n}C_n}{n+1}$ (imp)

(n: no of diff. BTs formed. Inorder fixed)

* Avg. lifetime of element in Stack

eg → Pg 145 (Gate Q) (Vimp)

* INFIX TO POSTFIX

Infix: a*b
Prefix: *ab
Postfix: ab*

(one using operator stack)

- Infix: a+b*c
- Postfix: (a+(b*c))
  ⇒ a+[bc*]
  ⇒ abc*+
- Prefix: (a+(b*c))
  ⇒ a+[*bc]
  ⇒ +a*bc

* Compiler converts every infix exp to Postfix exp & then evaluate it. (imp)
(reduces no. of scans for compiler & therefore reduces time)

Oper	Prec	Assoc.
+, −	3	LR
*, /	2	LR
( )	1	LR

eg → Pg 148 [Q 4]

① * Infix to Postfix using Stack

operator stack

Infix : a+b*c-d/e.

Postfix : abc*+de/-

operator stack

gate

[if equal/lower prec opr, pop out]

imp

- If exp has n char
  Conversion (infix to postfix) : O(n)

② * Evaluation of Postfix exp

operand stack

(i) Infix : 3+8*2-6/3

Postfix : 382*+63/-

(ii)

operand stack
3
8
2

8*2 = 16
3+16 = 19
6/3 = 2
19 − 2 = 17. Ans

(iii)

∴ 2 Scans To achieve result (i) & (ii)

O(n) + O(n) = O(n)

- Order of execution depends on compiler, we can't Predict it.

1. (((a+b)+c)+d) → ab+c+d+ [LR]
2. (a=(b=(c=(d=5)))) → abcd5==== [RL]
3. a^b^c = abc^^ (RL)

imp

* Unary operators

( )	
! log −	[RL]
^	[RL]
*, /	[LR]
+, −	[LR]

precedence.

imp [unary opr]

eg → Pg 72 (Rev Poly)

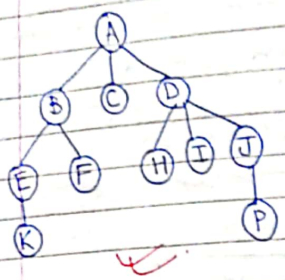
imp

* But DS to check balanced Parenthesis exp is Stack.

13 TREES

Level	Height	Depth
1	0	3
2	1	2
3	2	1
4	3	0

(can also start from 1 (depend on Q))



- EBA: ancestors of K
- HIJP: descendants of D

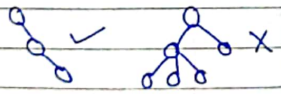
deg(D) = 3 (imp) [Tree]

⇒ deg = 0 (Leaf / terminal / external)
deg > 0 (non-terminal / internal)
↳ [Branch]

NOTE: If deg of tree = n, it can have max n children. (imp)

(1) BINARY TREES → deg = 2

- atmost 2 children ✓ ✗



(ii) Unlabelled Nodes

If n unlabelled nodes, no of different binary trees formed / diff shapes are:

(imp) $T(n) = \frac{2n C_n}{n+1}$

(imp) If n unlabelled Nodes, no of trees drawn with max height:

$T(n) = 2^{n-1}$

eg → Pg 2 (Rev copy) (n = 3)

$T(n) = \sum_{k=1}^{n} T(k-1)\,T(n-k)$ (Gate)

h	0	1	2	3	4	5
T(n)	1	1	2	5	14	42

(imp) $T(5) = 1\cdot14 + 1\cdot5 + 2\cdot2 + 5\cdot1 + 14\cdot1$

[Recurrence Relⁿ]

(iii) Labelled Nodes → A B C

∴ 3 Lab Nodes → 30

Total BTs = $\underbrace{\frac{2n C_n}{n+1}}_{\text{Shapes}} * \underbrace{n!}_{\text{Permuth in 1 Shape}}$

* HEIGHT V/S NODES

Substitute h=2 → n=7

(1) if height (h) is given (h ≥ 0)

h = 2, n = 7.

min nodes = h + 1 → skewed
max nodes = $2^{h+1} - 1$

$\frac{h \geq 1}{2^h - 1}$

(2) if 'n' nodes are given (h ≥ 0)

min h = $\log_2(n+1) - 1$
max h = n − 1

h = 3, h = 1

$l=3,\ n=4$

max nodes in level $l = 2^{l-1}$

min nodes in level $l = 1$

eg → Pg 4 (Rev Copy) (Derivation)

② Complete Binary Tree [Strict BT]

- every node have exactly 2 children.

$e = i+1$; $n = i+e$ $(2i+1)$

imp

In CBT = no. of nodes must be add

eg → Pg 183 [10 nodes]

imp

* In a not complete Binary Tree:

$deg(0) = deg(2)+1$ imp

③ n-Ary Tree

→ atmost n children.

* Complete n-Ary Tree

if $n=3$ → exactly 3 children.

$e = (n-1)i+1$ [Complete n-Ary tree] Strict n-ary tree

imp

eg → Pg 185 $[2n+1/3]$

* Representation of BT

① → Array
② → Linked

imp

if node at index i
left = 2i
right = 2i + 1
parent = $\lfloor i/2 \rfloor$

eg → Pg 197 [Q.2]

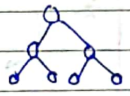
⑧ Linked Representation

```
struct node
{
    struct node *lchild;
    int data;
    struct node *rchild;
}
```

imp

n nodes = (n+1) NULL Pointers

④ Full Binary tree

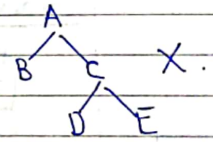


Given height h. ; $h \geq 0$.

$\max n = 2^{h+1} - 1$

⑤ Almost complete Binary tree

- If in array: No Gaps.
- In last level, elements will be inserted from left to right, & full BT till previous level.



TREE TRAVERSALS

DFS
- Preorder: visit(node), pre(left), pre(right)
- Inorder: in(left), visit(node), in(right)
- Postorder: post(left), post(right), visit(node)

eg → Pg 8 [Rev Copy]

⇒ Given a traversal, 1 shape/ unlab. BT can be filled in just 1 way. (imp)

If n labelled nodes,
No. of Binary trees, given a preorder/ inorder, we can generate total of
$\left[\frac{^{2n}C_n}{n+1}\right]$ BTS (imp)

if no traversal: $^{2n}C_n/(n+1) * n!$

Generating Binary tree from Traversal

$O(n^2)$

We need 2 traversals & one must be Inorder (imp)

pre: left → right
post: right → left

eg → Pg 194

Time taken for generating binary tree from traversal = $O(n^2)$

Time = O(n)
Space = O(n)

Preorder

eg → Pg 10 (Rev Copy) → n nodes, n+1 NULLP, (2n+1) funct calls. (imp)

O(n) = Time

```
void MyInorder(struct node *p)
{ if(p != NULL)
  {
    MyInorder(p->rchild);
    printf("%c", p->data);
    MyInorder(p->lchild);
    printf("%c", p->data);
  }
}
```

Pg 195 [Tree]

→ GGCFFCAEEBDDBA

eg → Pg 197 [Postorder count] (2n+1) fun calls [Vimp] ∴ O(n)

eg → Pg 198 [Counting] (Vimp)

eg → Pg 199 (Practice)

Inorder / Preorder / Postorder traversal / counting → O(n) ∵ (2n+1) funct calls.

6 BINARY SEARCH TREE

- Distinct Set of elements, where value of elem is greater than all elements in left ST & smaller than right ST.

Search time depends upon ht of tree

eg → Pg 12 (Rev copy) (Keys 1 → 100)

* Inorder of BST ⇒ always sorted order of elements. (imp)

* Generating BST from Traversal

→ 1 traversal sufficient to create BST [Pre / Post]

eg → Pg 203 [Can this be preorder of BST?]

Gate

* Algo to create BST from pre/post: O(n)
Algo to create BT from pre/post: O(n²) (imp)

NOTE: Preorder can't tell you, whether its BST or not [only inorder says] (imp)

* If n labelled Nodes, (imp)
No. of BST possible = $\frac{2n C_n}{n+1}$

∵ each structure labelled in 1 way (∵ inorder given).

* Operations on BST:

① Insert:

- depends upon height of tree
- O(h) = O(log n) [best case]

② Create BST

eg → Pg 206.

③ Deletion. (imp) [Imp]

- Replace with inorder predecessor / successor.

eg → Pg 14 (Rev Copy)

all depends upon ht of BST.

* Drawback of BST (imp)

∵ Height of BST, depends upon order of insertion of elements.

h > 0 → min h = log(n+1) [Imp]
max h = n [Skewed]

⑦ HEIGHT BALANCED / AVL TREES

Balance factor ⇒ ht of left SubTree − ht of right SubTree

i) balanced → $Bf = \{-1, 0, 1\}$ (imp)
ii) not balanced → $|Bf| > 1$ ✓.

eg → Pg 211 [LL, LR, RR, RL rotations]

AVL tree is made Step by Step using rotations, it's not that, you get a Skewed tree & you make AVL Tree. (imp)

eg → Pg 216 (Example)

* Height V/S Nodes in AVL Tree

⇒ If height 'h' is Given : $(h \geq 1)$.

$\min h = \text{Table}$
$\max n = 2^h - 1$ ✓

⇒ If n nodes are given :

$\min h = \lceil \log_2(n+1) \rceil$
$\max h = \text{Table}$ ✓

$h \geq 1$

Height	h	0	1	2	3	4	5	6	7	8	9	10
Nodes	n	0	1	2	4	7	12	20	33	54	88	143

↳ like Fibbonacci

↳ 5 ht. AVL tree ka lis min 12 nodes / 12 nodes ka lis max ht = 5. (imp)

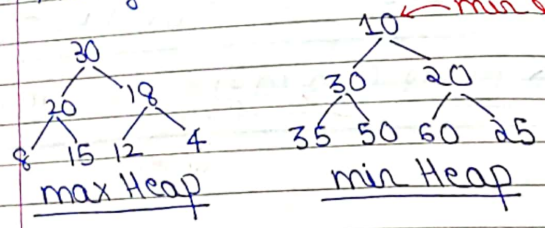
eg → Pg 218 (Q2/Q3) (imp)

[Huffman Coding, dykstra, Tab Sequencing use Heap.]

Page No.: 22 Date: Youva

4] HEAP

- Every heap is almost complete BT, there must not be any Gaps in Array.



eg → Pg 220 [Which are is heap]

Height of heap = $\lceil \log n \rceil$ (Vimp)
Height of BST = $\log n \le h \le n$
Height of AVL = $\log n \le h \le$ Table

* Operations of Heap:

① **Insert** — O(log n)

- Insert the element & heapify. O(log n)

O(log n)

② **Delete**: delete the root & heapify. O(1)

replace with last element

largest elem → max Heap
smallest elem → min Heap

∴ O(log n) ← [contributed by heapify].

Page No.: 23 Date: Youva

eg → Pg 25 (Rev Copy) (imp)

③ Create Heap = O(n)

④ Heap Sort → n + n*log n
- Create heap: O(n)
- delete element one by one & heapify: O(n log n)

T = O(n log n)

- In Place Sorting. (imp) [No auxiliary array].

eg → Pg 19 [2015]

In a max Heap, smallest element must be in lowest level, ∴ it must not have children.

eg → Pg 20 (Rev Copy) [7th ele] (imp)

eg → Pg 21 (Both Qs)

Max Heapify = O(log n)
Build max Heap = O(n)

∴ Create Heap: O(n) (Vimp)
Heapify: O(log n)

∴ n/2 nodes (Leaf)

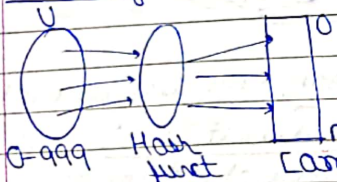
Delete min elem from max Heap: O(n)
max Heap → min Heap: O(n)
delete: O(log n).
Search: O(n) // find (imp) (Always)
delete 7th elem: O(log n)

5) HASHING

- In not DS, search time is costly
  [LL → O(n), BT → O(n), BST → O(n) [SK], AVL BST → O(log n)] O(log n).

But, Hashing → O(1) (imp)

* Hashing



eg: Keys → (121, 145, 132, 999)

0	1	2	3	4	5	6	7	8	9
	121	132			145				999

hf = mod 10 ∴ m = 10

Insertion = O(1)
Search = O(1) (imp)

* Problem: Collision

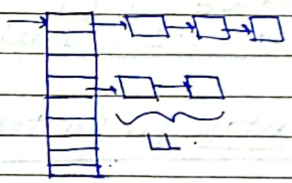
[When more than 1 element map to same space].

Solution to collision:

1) Coming up with better hash functions
2) Chaining.
3) Open Addressing.
   - → Linear Probing
   - → Quadratic Probing
   - → Double Hashing.

[Hash structured Page Table]

4) CHAINING



Insertion = O(1)
Search = O(n)
Deletion = O(n)

all item might map to single entry in a list.

(imp)

Load factor (α) = $\frac{n}{m}$ — n → item in HT, m → size of HT.

In uniform dist funct^n, Avg length of list = α.

∴ Avg search = $\theta(1)$ (imp) [Average].
Avg deletion = $\theta(1)$ (Vimp)

NOTE: For deletion, chaining is much better than OA; most oper^n is O(1).
Drawback: stored outside table (not space efficient)

eg → Pg 5 (Ravi) [LF = avg. L of list] (imp)

eg → Pg 6 $(97/100)^3$ [Probab.] (Opt.)

eg → Pg 6 [Q 1] (Vimp) (imp)

3) OPEN ADDRESSING

- We can't insert more elements than table size.

$\alpha = n/m$

OA → $\alpha \in [0, 1]$
Chain → $\alpha \geq 1$ (possible) (imp)

Adv: No Space Wastage for Pointers.

(i) Linear Probing

- Probing in a linear manner.

(imp) (1, 2, 3, ···· m−1, 0)
(2, 3, 4, ··· m−1, 0, 1)] probe sequence

No. of probe sequences possible = m

Note: If 2 elements map to same initial index & follow the same Probe Sequence:
⇒ Secondary Clustering

(imp) If a run of elements filled up, prob of next slot to be filled up is very high: Primary clustering

disadvantage

(imp) [due to Primary Clust.]

Search time = O(n) [Avg]

eg → Pg 12 (Gate 2008)

eg → Pg 13 [Q1: Just Prob
Q2: (i > 10] → Primary Clustering.

eg → Pg 14 [Gate 2010] [Vimp]

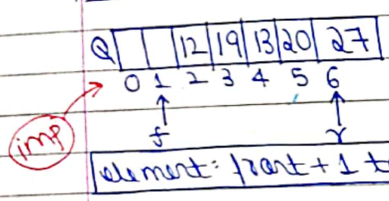
(Vimp) [Pnc]

5 QUEUE

- Works on FIFO.
- Elements are inserted from rear end & deleted from front.
- Implemented as ADT.

(1) Queue using Array

(i) Linear Queue



r → till what idx, we have elem.
f → beginning of Q.

imp

element: front+1 to rear | f → empty S

- Initially: front = rear = -1
- Empty: if (front == rear)
- Full: if (rear == size-1)

imp

Code → Pg 74 (Rev Copy) (Enque & Dequeue)

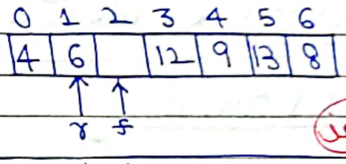
* Drawback of Linear Queue: front++ rear++ (both).
- Empty space can be used, if rear is already pointing to last element.

* Solutions → Resetting Pointers
→ Circular Queue.

* Resetting Pointers: at any point, if Q becomes empty, reset front & rear to -1.

(iii) Circular Queue → Solⁿ to drawback of linear Queue

- front & rear move in circular fashion.



imp

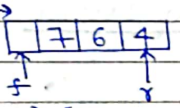
same as linear queue

element: front+1 to rear | f → empty S

* If size of array is n, we can store n-1 elements.

- Initially: front = rear = 0.
- Empty: if (front == rear)
- Full: if (rear+1 % size == front)

imp



(full)

Code → Pg 76 (Rev Copy) [Enqueue & Dequeue]

Circular Queue: always used.
Linear Queue: Space wastage

(2) Queue using linked list

imp

- Initially: front = rear = NULL;
  [When first node created, front = rear = 1]
- Empty: if (front == NULL)

O(1)

Code → Pg 162 [Enqueue & Dequeue]

(iii) Double Ended Queue

Queue:

	insert	delete
front	X	✓
rear	✓	X

DE-Queue

	insert	delete
front	✓	✓
rear	✓	✓

i/p restricted: insertion = rear, deletion = f/r

imp

(iv) Priority Queue

- fixed no. of priorities [OS]
- unlimited no. of priorities → Insert = O(n), delete = O(1)

or vice-versa for max priority

Heap is best DS to implement Priority Queue → Insert: O(log n), delete: O(log n)

(Max Heap) #1
[But max element: O(log n), Min element: O(n)]
So use Min-Max Heap → max: O(log n), min: O(log n)

Gate

⇒ In Priority Queue, Priority of every element must be distinct.

eg → Pg 166 [Running Queue]

eg → Pg 168 [Q1 & Q2]

imp

* Implementing Queue using 2 Stacks

eg → Pg 78 [Rev Copy]

S1 → used for enqueue
S2 → used for dequeue

Concept

Code → Pg 169 [EnQ & DeQ]

imp

eg → Pg 171 [Imp Q]

Imp

Appliction

Test → Enque: O(1), Deque: O(n) (or vice-versa)

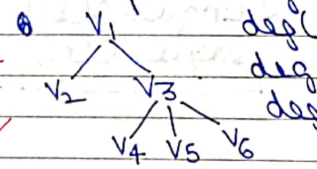
PREV YEAR PAPERS

1) TREES

* Insertion of an element if it is not already present in the Set (imp)
  → find element
  → if not found, insert.
  Heap: $O(n) + O(\log n) = O(n)$
  Balanced BST: $O(\log n)$ [AVL]

* Cost of Search/insertion/deletion in a complete binary tree: $\Theta(n)$
  Cost of Search in AVL/Balanced BST: $\Theta(\log n)$
  Cost of Search in BST: $O(n)$ → worst case.

* If degree of node is defined as the number of its neighbours.
  (like def of vertex in undirected Graph)



$deg(V_1) = 2$
$deg(V_2) = 1$
$deg(V_3) = 4$

* Q.5.48 → Imp Q on PnC & BST. [LLLLHHH] 60
  Q.5.79 [How many diff orders are possible?]
  → 2 choices at each level.
  (Imp)

* In order to count the no. of nodes/number of subtrees having exactly 4 nodes: Preorder/Post order traversal needed
  → $O(n)$
  ∵ $(2n+1)$ junction cells

* Inorder Traversal of a Ternary tree: (imp)
  left → root → middle → right.

2) HASHING → already in Notes

* Q.7.11 → PnC & Hashing

* QUEUES

- In a Priority Queue/Heap, 2 elements can't have same Priority.

* Reverse Polish → Postfix form expression

* Linked list is not Suitable in Binary Searches [BS not quite efficient in LL]

* Compact single dimensional array representation for lower triangular matrix → Q 2.1.
  $(j) + \frac{i(i-1)}{2}$
  [Fill in Stack]
* 2D Array → Address notation.
  Q. 2.11 - Gate 2015 (Set 2) (imp)

* Queue implemented as 1 Stack.
  Enqueue: reverse, push, reverse
  dequeue: pop (imp)

7 GRAPHS

G=(V,E) represented as:
(i) adjacency matrix: $O(V^2)$ / $O(n^2)$
(ii) adjacency list
→ $O(|V|+|E|) = O(n)$. (imp)

* Graph Traversal Method → BFS
  → DFS.

(i) Breadth first Search: If you visit a node, visit it completely (all its neighbours) before exploring some other node.
→ Queue DS is used.
eg → Pg 55 [Vani]

→ In BFS Spanning tree: day maintained. (imp)
Cross edge: n to n+1 level
[or same level]

eg → Pg 55 [all 5 Qs] (Valid BFS order)

(ii) Depth first Search: start visit. Once a new vertex w is visited, suspend it into Stack, & start exploring w (neighbour).
Back edge: any level - level (imp)
eg → Pg 56 [DFS spanning tree]

eg → Pg 292 [2 qs] (Ht of a tree must be ≤ to no of nodes, if ... → articulation point present) (imp) Gate
eg → Pg 293 [Discovery & finish]
eg → Pg 20 [Self Note]
(Most imp → BFS, DFS tree).

PREVIOUS YEARS.

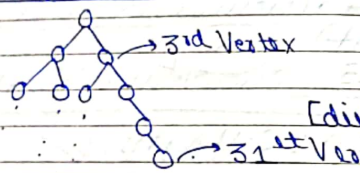
* The most efficient algo for finding n of connected components → BFS/DF
  $\Theta(m+n)$ → (if with adjacency

* In a sorted doubly linked list: (im
  delete → O(1) → if pa
  Insert → O(n) → an no
  Find → O(n)
  delete key → O(n) [Vimp]
  (find key & delete)

	find	insert	delete	decrease key
Unsorted Array	O(n)	O(1)	O(1)	O(1)
Min Heap	O(n)	O(log n)	O(log n)	O(log n)
Sorted Array	O(log n)	O(n)	O(n)	O(n)
Sorted doub. linked List	O(n)	O(n)	O(1)	O(n)

4 parts.

* Depth first Search requires $O(V+E)$ time, if implemented with adjacency list & $O(V^2)$ time, if with an adjacency matrix. (imp)



* Tree edges: edges that appear in the final DFS tree.

$n = 10$
$K = 4+2+1 = 7$ (tree edges)

(imp) $\therefore$ connected components $= n - K$

* So, any Graph with minimum cycle length 3, will have at least 3 Spanning trees. (imp)

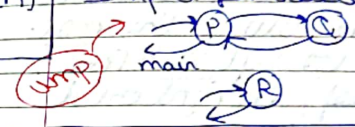
$K_3 \rightarrow n^{n-2} = 3^1 = 3$

(imp)

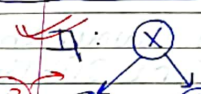
* eg → 6.5 [Group → GO SQL] – computes whether subGroup is connected / not.

eg → 6.10

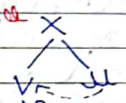
Imp Q for discovery / finish time. array discov. & finish times.



(imp)

If: 

dfs tree



$d(u) < d(v)$ [v discovered after u discovered] X

acyclic Graph

$d(u) < f(v)$ X

$f(u) < f(v)$ X

$f(u) > f(v)$

* If a node is a leaf in the DFS tree, and deg 2 in Graph, (atleast 2). then definitely, it has to be in a cycle. (Gate)

→ In undirected connected Graph, only back edges possible in DFS tree (cross edge not poss.)

→ In a BFS tree, if uv edge, [cross edge] u → depth i, v → depth j.

$\therefore |i-j| \leq 1$

(it can't be > 1)

TEST-2019 (Subject Wise)

* In a BST, the insertion/deletion oprn is not commutative.

* Predecessor of a node (in its inorder traversal), is always the max. element of its left SubTree.
∴ either a leaf / its right child must be empty.
Successor of a node is always the min. element of its right SubTree.
∴ either leaf / its left child must be empty.

* Every recursive prog can be converted to iterative prog [and vice-versa]
recursive

If tail recursion ⟷ loop.
[similar stack space]

* char (*(*x())[])();

char x1(); // funct^n return char.
char (*x2[])(); // array of pointer to funct^n returning char.
char (*(*x())[])(); // funct^n returning pointer to above.

reading: (Function Pointer)

* strlen(): counts length of string. (excludes NULL)
sizeof(str): counts length of string. (includes NULL)

* Consider a 2D Array A[40...95, 40...95] → lower triangular matrix.
BA = 1000, add. of A[66,50]

	40	41	42	...	95
40	a_{00}				
41	a_{10}	a_{11}			
42	a_{20}	a_{21}	a_{22}		
⋮					
66	.	.	.	.	
⋮					
95	.	.	.	.	.

Size of elem = 1B

⇒ To reach row no 66 = (66−40) = 26 rows covered.
∴ No. of element = $\frac{26 \times 27}{2}$ = 351.B
⇒ To move to col. 50
⇒ (50−40) = 10 elements covered.
∴ 1000 + 351 + 10 = 1361

* int S[6] = {128, 256, 512, 1024, 2048, 4096}
2000 2-1 2A 12 1E 20

int *x = (int*)(&S+1);
x = 2024 [& array object]

* if we are provided with a pointer to the node which is to be deleted in SLL, it won't work in all cases like (if it's last node) only
[Copy from next & del. next]

imp

* Given Linked List is palindrome/not
→ first identify mid. element.
→ reverse second half.
→ compare first half with second half.
∴ O(n) + O(n) + O(n)
∴ T = O(n)
S = O(1) [no extra Array]

eg → Q31 [Address corresponding to A[100][55] if; elements in Z representation].
99 + 99 + 54 (99) + 1000 = 1252
(100-1) (55-1)

* In BST, time taken to find a non-existent element 'x' in the best case is: O(1) 2 3 4 find 1.

imp

* In minHeap, time taken to find a non-existent element in the best case is: O(1)
∵ [smallest element at root; if x < root; → O(1): (not found)]