Data Structures and Algorithms

Built in functions

1) List
2) Dictionary
3) Tuple
4) set

user defined data

1) Stack
2) Queue
3) Tree
4) Linked list
5) Graph
6) Hash Map

Big 'O' notation :

↳ running time & Space requirements

time $= (a * n) + b$

1) keep fastest growing term

term $= a * n$

2) drop constants

3) Time $= O(n)$

time $= (a*n) + b$

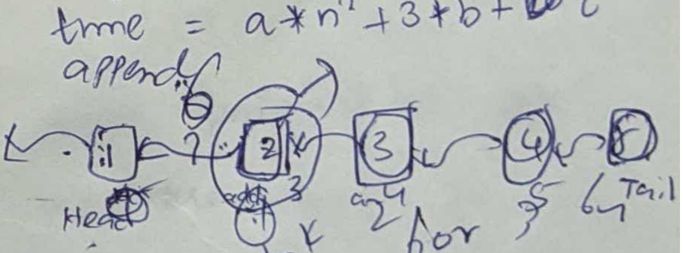
$100 \cdot 100 + 5$

$200 + 5$

$10000 + 5$

time $= a*n^2 + 3*b + c$

append

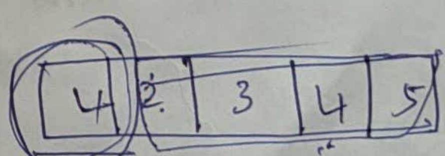


$k = n/2^k$

$i = 4 = 4$

$\frac{2}{2}$ 10

print (4)



$O(n)$

4 2

$O(n)$

1 = 4

2 = 4

3 = 4

(4)

1) Linear
2) non-Linear
(Hierarchical DS)

Data

1) Store
2) Search
3) Modify → Add
→ Delete

Advantages of
1) can modify easily
2) insert & delete are easy
3) can implement stack, queue
graph
4) represent and manipulate polynomials

Big 'O' Notation - Measuring the efficiency

Linked list : 1) Dynamic DS
2) No need continuous memory allocation
3) can store in different places.

Node1 ← Node2 ← Node3

Node

1) int
2) String Data Pointer → next node in list
3) class

↓
(Memory Location)

Disadvantages:
1) need extra memory
2) Random access not possible
3)

Head Node → Data, Pointer
Data (a), Pointer (null) → (b)

Tail node → Data, Point
Data (b), Point (null)

Add

1) Add to Head
2) middle
3) Tail

Remove

"

operations :

Store $O(n)$

Search $O(n)$

Add $O(n)$ or $O(1)$

Delete $O(n)$ or $O(1)$

Real world examples

1) Treasure Hunt:

clue1 → clue2 → clue3 → destination

destina

The information to Reach next clue stored in previous

3) Relay Running clue Run

uses :-

1) to implement Stack, Queue

2) Music player.

3) Web Browser

Doubly - Linked List :

↳ why needed ?

1) we can't go back

$\boxed{\text{Node}}$ $\boxed{\text{N}}$

Previous Pointer	Data	Next Pointer	⟷	PP	D	NP
(null)	(a)	(null)			(b)	

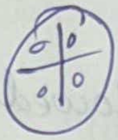
N N N

P a P ⟷ P b P ⟷ P c P

(null) (null)

Circular linked list:

1) Same adding methods as linked list

 Gunal (Both have different advantges)
can't go back.

4) Same Time Complexity as linked list:

uses:-

1) Back and forward

2) undo / Redo.

Dictionary : (Map)

1) Stores key value pairs

key - unique identifies

value - data

(or) key - aadhar

value - Address

Difference Array & dictionary.

	Sachin	kohli	Jee	Dhoni	yuvi
index	0	1	2	3	4
value	2	5	8	3	1

in dictionary we don't need index we can store player name directly in keys

1) keys can be anything int, strings

Each key → only once
" → only one value

Same value → multiples keys allowed

Time complexity :-

Dictionary $\xrightarrow{\text{implemented}}$ Hash Table

Hash ~~Map~~ :- Table :- (is efficient in storing values than array)

using array — if we to save this value we need 12600 index, it will be leave many null values before & , we use hashtable

keys	0	4	7	3	6	12600
value	2	5	8	3	1	4

Index	0	1	2	3	4	5	6	7	8	9
value	2	null	n	3	5	n	1	8	n	n

1) easily Accessible
2) Reduce null value

Hash Function :-

Key	1	10	100	1k	10k	100k	1M
value							

(Hash function) return (0)s

Index	0	1	2	3	4	5	6
value	2	5	8	3	1	5	8

(8) 1000 → Hash function → 3

Hash collision →

(2) 10000 → " → 3

} Both function will have same value

1) open Addressing

2) closed "

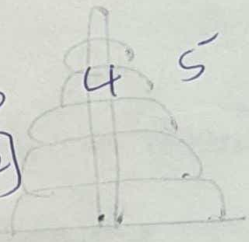
open addressing :

0	1	2	3	4	5
			8	2	

closed Addressing :

1) Store as a list

0	1	2	3	4	5
			[8,2]		



Time complexity :

1) Not worst case (for Hash Table, there is less chance)

store → O(1)

search → O(1)

add → O(1)

Delete → O(1)

Only average case

Best time complexity

uses :

1) non numerical key (we can use any key into string etc.

2) Speed. (Powerful datastructure).

3) Password Hashing.

http Raja raja

https Raja

(salting)

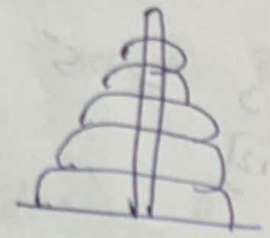
Array :-

1) int, Boolean, String, characters (stored)

operation :-

Store	O(1)
search	O(n)
~~se~~add	(dn)
Delete	O(n)

Stack :- (LIFO)



Methods in Stack :

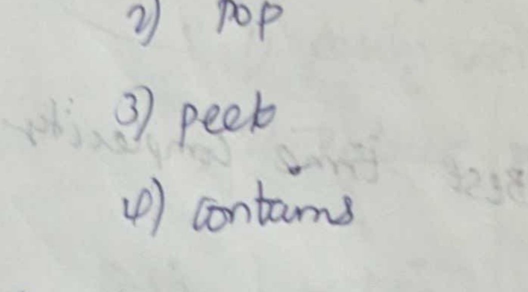
1) Push
2) Pop
3) peek
4) contains

1) Push

Operation

Stack.push ("a")

" ("b")

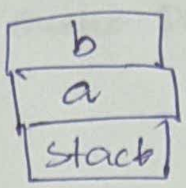


Size = 0

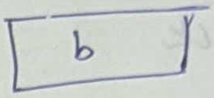
2) Pop (Remove the thing at top)

operation

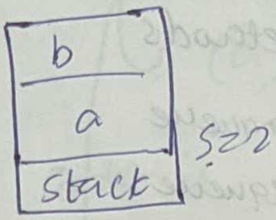
stak. pop()



Return

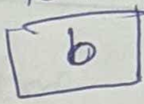


3) peek (Return the thing at top

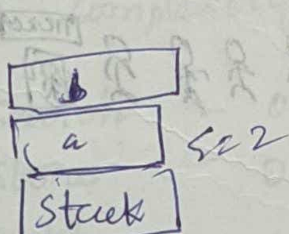


stack. peek()

Return

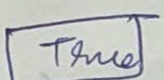


4) contains



stack. contains("a")

Returns



Time complexity :-

store O(n)

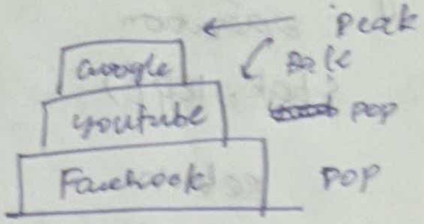
Search O(n)

Add O(1) (insert, delete only in top)

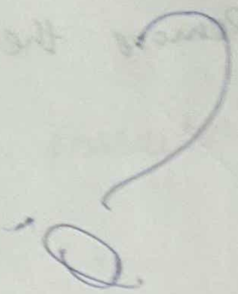
delete O(1)

uses :-

1) undo / redo
2) Recursion (calling same function).
3) Back Button



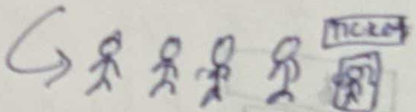
Stack



Queue :- (FIFO)

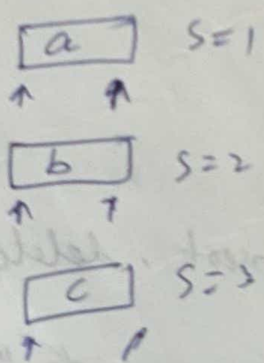
Stack Methods		Queue Methods
Push	→	Enqueue
Pop	→	Dequeue
peek		Peek
contains		contains

1) Enqueue (insert in back.)



operation

queue . enqueue ('a')
" " ('b')



2) Dequeue :

operation

queue. Dequeue ('a') → Return (a)

queue

| a | remove

| b |

| c |

3) peek (return the thing at front)

→ (a)

4) contains

queue. contains('a') → Return true.

→ True.

Time complexity :

Search - O(n)

Store - O(n)

Add - O(1)

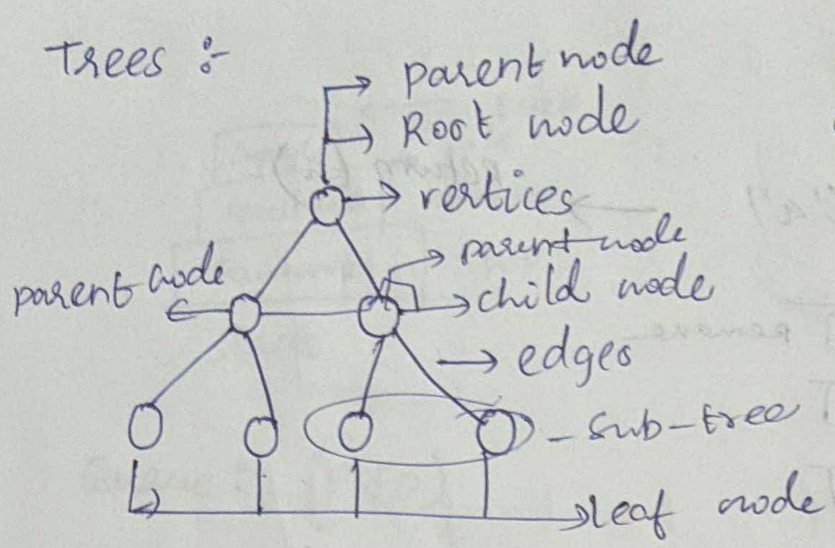
delete - O(1)

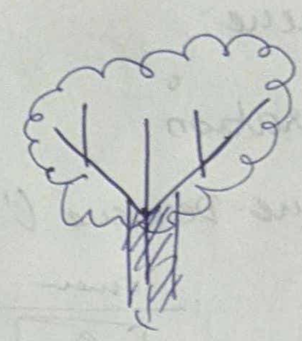
uses :

1) cpu jobs Scheduling

2) printers. (Printer order)

Trees :-

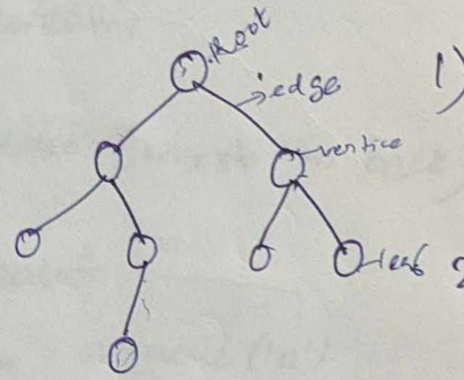




Node 1 → Node 2 (can use multiple nodes)

Node 1 → Node 3

(noder == vertices) verten



1) Height of tree → 3

Longest root to leaf

2) Depth of node

No. of edges from node to root

we have to find for each leaf node (differ for each")

Rules :

1) Binary search tree
2) Red Black tree
3) AVL tree

uses :

1) Folder
2) Hierarchical Data.

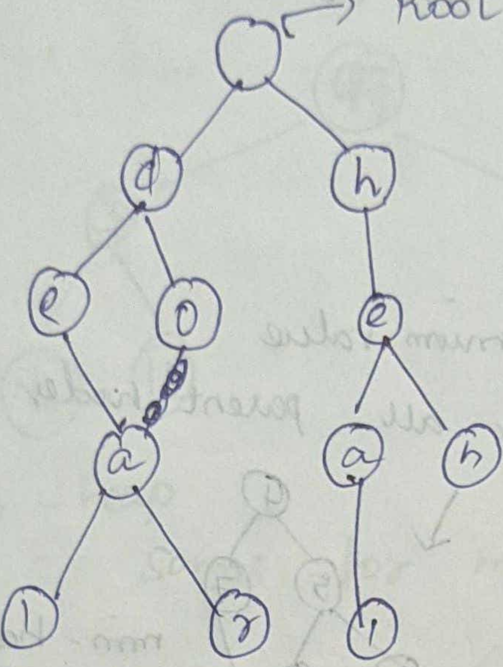
If we use different rules we can form Tries

(tree) Tries :- (Autocompletion) of words.

1) Nodes have letters

Root Node must be empty.

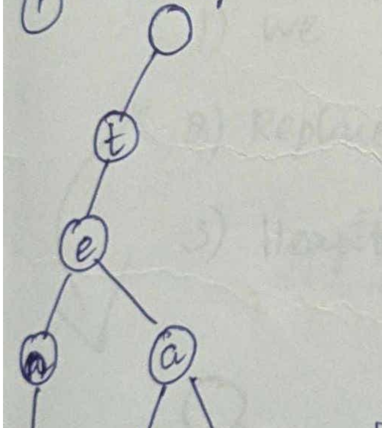
(Retrieval of Data)



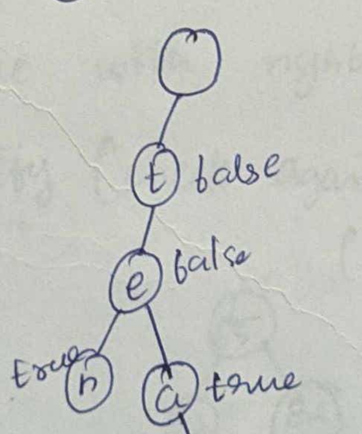
1) In an array it will take lot of place.

↓ (reusable)

① Flagging



② Boolean



Ten, Tea, Team

1) Flagging
   1) adding full stop.
2) Boolean usage.

1) Autocomplete words
2) Folder / cmd directories

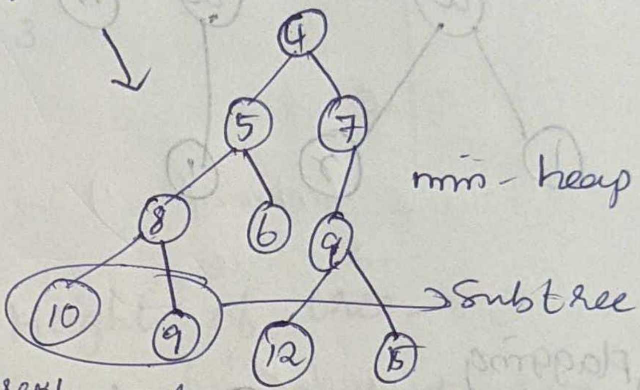
(Tree)
Heap :

1) It must be Binary Tree
2) only two childs can be there

Types of Heap :-

1) Min-Heap :

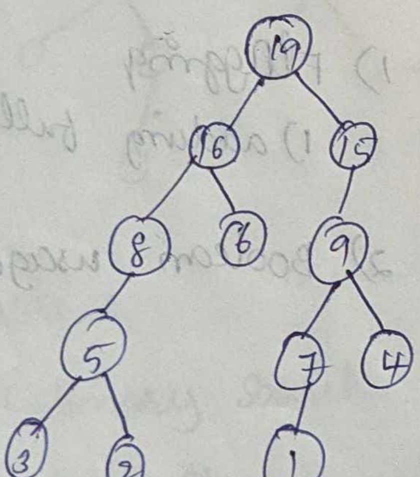
~~2) Max~~ 1) Root has minimum value
2) Same rule for all parent nodes



2) Max-Heap

1) Root node max value

2) Same rule for all parent node.

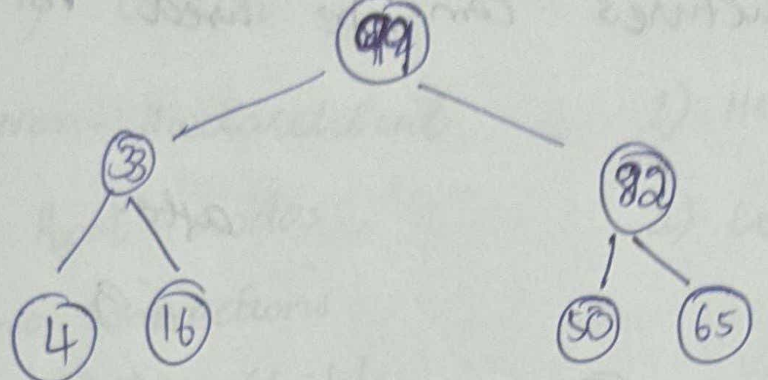


Array to max-Heap :-

a [50, 4, 82, 33, 16, 65, 99]

1) left most child
↳ If not Right

Recursively compare parent and swap
if necessary.



Max - heap

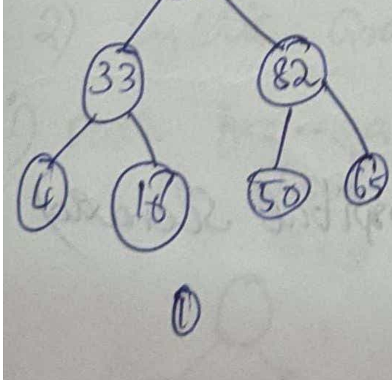
Same for min heap.

(Deleting from Max - Heap)

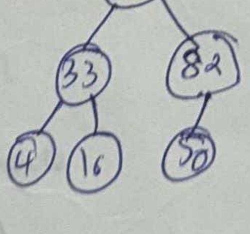
1) We can only remove the root node

2) Replace with right most node.

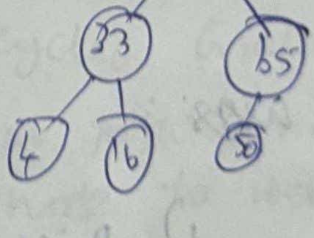
3) Heapify (check again and sort the max-heap
(Catching the mistakes in max Heap)



1



2



3

Same for min - heap

Heap Sort :- (ALGORITHM)

1) Heaps will be useful in Heap Sort algorithm. ~~Data Stru~~

2) Data Structures can be used in algorithms

before a

a

50

4

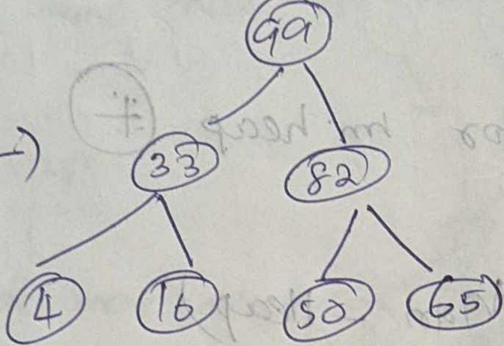
82

33

16

65

99



after a

99

82

65

50

33

16

4

why these datastructure ?

for algorithms

uses :

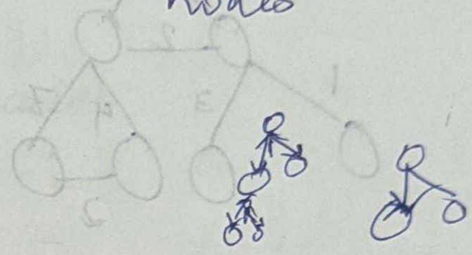
1) Priority Queue (Hospital Scenario)

2) Sorting.

Graph :-

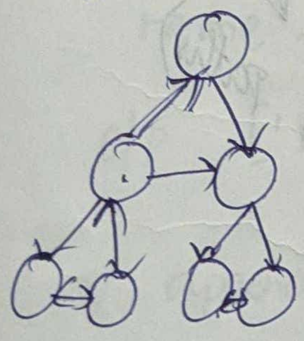
Difference between Graph / Trees :

i) Graph	ii) Trees.
1) Non-Heirarchical	1) Heirarchical
2) Root nodes no connections between childs	2) Connection between nodes



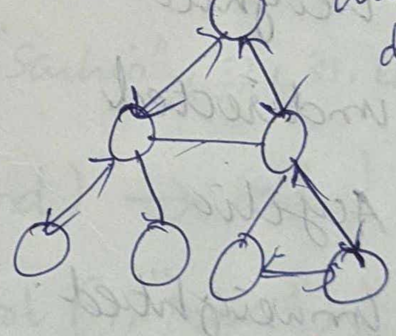
Types of graphs :

1) Directed graphs
eg: Instagram followers

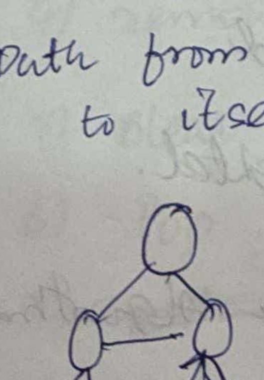


2) undirected graph
eg: Facebook Friends

allways double arrow

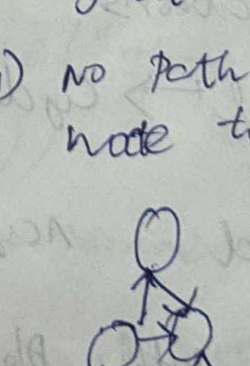


2) cyclic Graph
i) path from a node to itself



All undirected graphs are cyclic

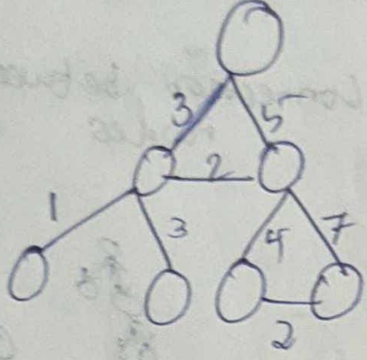
Acyclic Graph.
i) No path from a node to itself



7 ~~All~~

1) undirected graphs are cyclic

weighted graphs :-



Facebook friends (years)
using edges

Graphs types :-

1) Directed
2) Cyclic
3) weighted
4) undirected
5) Acyclic
6) unweighted.

10

Dijkstra algori
(Read).

Google maps
(shortest paths)

Interpretations:

Directed ⟷ Cyclic   weighted

undirected   Acyclic   unweighted.

1) Shortest path Algorithms / Dijkstra Algorithm
2) Followers / Friends / Subscribers

Tuple :- ~~[]~~ - ~~Shudre Brackets~~
() Parenthesis

1) immutable . So, we can't change Tuple.

uses :-

1) Longitude and latitude of a place.

~~Set~~ List : [] - Square Bracket

1) extend.

name = ["kohli", "Dhoni", "Sachin"]
age = [32, 39, 45]

name.extend(age)

O/P ['Kohli', "Dhoni", "Sachin", 32, 39, 45]

2) append ((name.append) → add)
3) insert ((name.insert (1, "Dravid")))
4) remove (name.remove("Dhoni"))
5) clear (name.clear())
6) Pop (name.Pop()) - remove last element
~~7) push (name.Push~~()
7) index (name.index())
8 Count (name.count())
9) Sort (name.Sort())

10) reverse (name.reverse
11) copy
name1 = copy()
Print(name1)

set : {} - curly brackets

1) No duplicate elements

↳ list can have

2) set ({1, 2, 1, 3})

set1 = {} is a dictionary

set add set keyword

type print (type (set1))

3) No definite order

function

1) add

2) remove

{1, 2, 3}

~~set~~. set1. discard (4)

o/p (no error)

3) discard → will do nothing. if the element not in set.

4) clear

5) pop - Random element in set

6) union

↳ {1, 2, 3}

set1 . pop ()

o/p 3

{1, 2}

odd = {1, 3, 5}

even - {0, 2 4}

prime - {2, 3, 5}

{0, 1, 2, 3, 4, 5}

7) intersection

? 1) Subset
2) Sub arrays

8) Set1 = {1, 2, 3}
Set 2 = set 1
Set 2. add (4)
print (set 2)
print (set1)

O/p
{1, 2, 3, 4}
{1, 2, 3, 4}

1) No new set will be added
2) use the set which already created
3) So, 4 is printed in second set

9) Copy - for creating new unchanged set

set 1 = {1, 2, 3}
set 2 = set1 . copy () => set2 = set (set1)
set 2 . add (4) or "
print (set2)
print (set1)

This function will create new set

O/p {1, 2, 3, 4}
{1, 2, 3}