Trees → Graphs

↓ BT ↓ BST ↓ Heap

(1) (2) (3)

BT (3) BST (1) Heap (2)

Different world → DS. | not as exciting as problem solving.

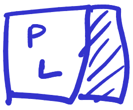
Boring (Abstract)

Recursion, Greedy, MCM, Knapsack, Sorting, Searching, 2ptr, sliding window → Algorithms.

DS → Whats the best way to keep the data?

$O(2^n) \to O(n^2) \to O(n\log n)$

1. in-built data structure → programming lang → array.



→ import stuff | STL in C++ | vector → dynamic array

Collections in Java | stack, queue, map, set

↳ libraries

auxiliary data structures

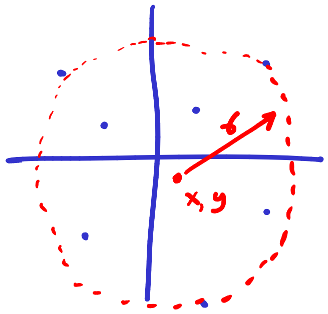
Is array or containers sufficient to put all kinds of data in world?

No? Problem: N points (x_i, y_i) ← How are you storing these points?

give me the minimum area circle such that all points are within this circle.

o/p: x y r → Algorithmic

↳ Optimization.



arr: $[(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)]$ → Is this the best?

2D points $\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{bmatrix}$ $[x_n]$ → n-dimensional vector.

→ AI ↑ LLMs ↑

SQL DBs

NoSQL DBs

Vector databases

Optimize: learn different ways to keep your data so that certain ops become easier.

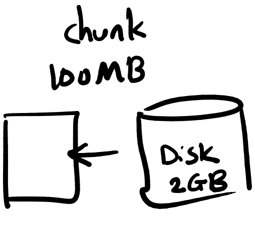
BT ✓

↓

BST ✓

↓

self balancing BST ✓ → Interval trees → B Trees → B+ Trees



Postgres ↗ B+ Trees

How is this data even kept in disk/storage?

index	col1	col2	..	..
–	–	–	–	–
–	–	–	–	–

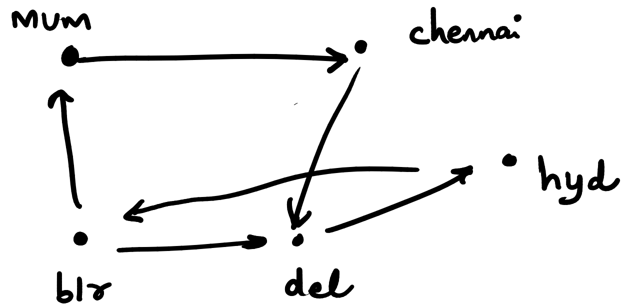
CSV format: Is this even fast?

index, col1, col2, ...
1, – – –
2, – – –

Social media : Follower/Following → Graph DB → graphs + metadata.

static graphs → ✓

dynamic graphs   Every 1 min graph needs update.



Some edges → static } persistent DS
remaining → dynamic

DS

Q. ⇒ How should you keep your data ? → Once data is kept thoughtfully,

what problems can I solve ? ← DP, greedy, ... algo

Everything is already there! Bloom Filters → probabilistic DS

Analogy : You learn guitar (basics) → appreciate music !
You learn chess (1500 elo) → magnus carlsen (2830)
you appreciate/understand.

TREES.

DS for Interviews
↳ 40% of companies.
atleast 1 question

99%

```
import model from tf
model.fit(data) #more args
model.test(test_data)
```

lucky → 3 questions
1. Sorting
2. Searching
3. DP
} many companies follow. (uber)

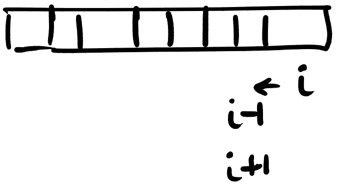
Tree : 1st non-linear datastructure.

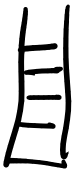
You must do it yourself.

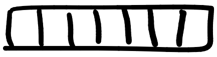
```
class node {
    int data
    node left
    node right
}
```

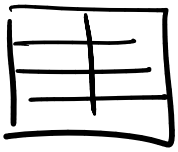
```
__init__ :
self.data
self.left
⋮
```

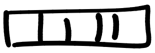
Containers internally arrays.

Array


Stacks


queue


map


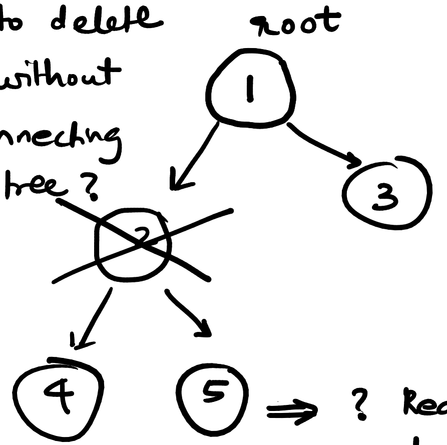
set


hashtable

```
root = new node(1) ;
root.left = new node(2) ;
root.right = new node (3) ;
root.left.left = new node (4) ;
root.left.right = new node (5);
```

Constructor → data = 1
left = null
right = null

9. how to delete 2 without disconnecting the tree?



⇒ ? Reach here.

Hierarchical data structure !!

root.data ⇒ 1

Any data structure

1. CRUD

Create a DS ↑ Insert data.

Read the DS ↓ get the data back.

problem.

Update ↓
1. search (what to update)
2. Delete it
3. Insert (new updated value)

T.C. O(n)

To insert n values.

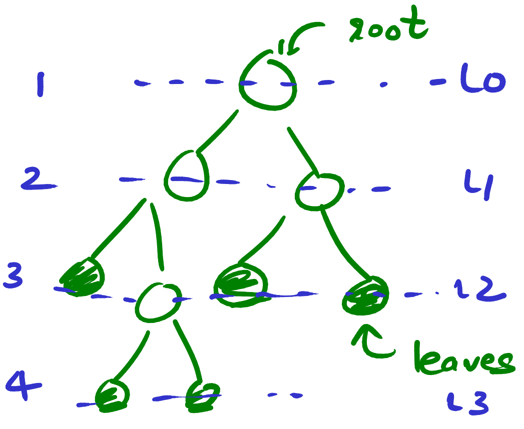
2. Specific ops → Know this DS pros vs. cons. ↓ Special cases where this DS performs better than array.

Delete.

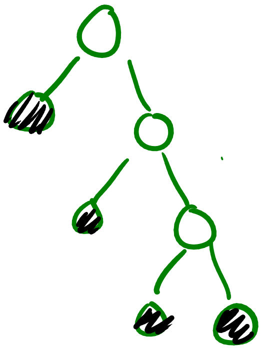
Certain problems (s) → Tree → can solve it without tree knowledge !!

Terminology :

1. root → head of the tree → only entry point.
2. leaves → nodes with 0 children.
3. Internal nodes → not leaves.
4. Branches → root → leaf path
5. height of tree = 4 (h).
6. levels → 0 to h-1



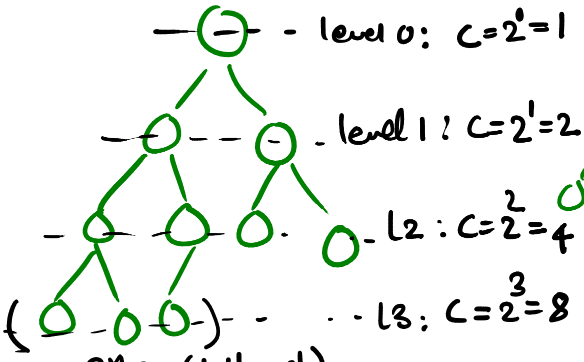
Types of trees. (4)



Full

every node should have 2 children or 0. (leaves)

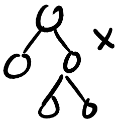
#leaves = internal nodes +1



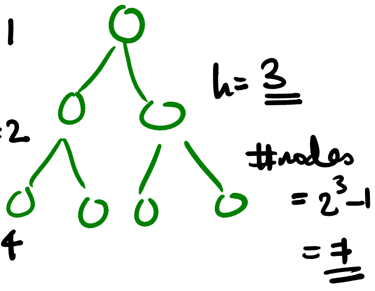
extra (leftmost)

Complete.

every level is complete ie filled to its capacity except the last level. any extra nodes must be leftmost.



why?

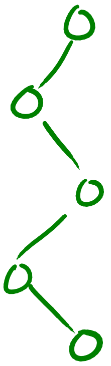


Perfect

all leves filled up.

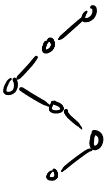
h

#nodes $= 2^h - 1$



Skewed.

1 node at each level.



h = #nodes

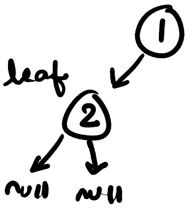
Coding time. 8 problems : solved without any knowledge of tree → CRUD.

① Find the sum of all nodes of BT given root.

```
int sum(Node root):
    if root == null : return 0
    return root.data + sum(root.left) +
             1         sum(root.right)
```

v.used ↓

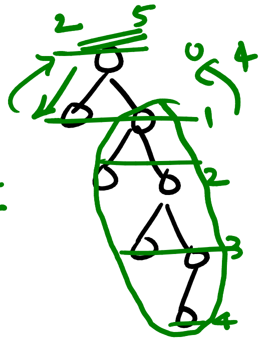
variation: count nodes in tree?



② height of tree.

```
int height(root):
    if root == null : return 0
    return 1 + max(height(root.left), height(root.right))
```

Done



random

h=5

③ Search for a key : ⟶ Yes/No (∵ no index)

```
bool ~~int~~ search(root, int key):
    if root == null : return false
    if root.data == key : return true
    return search(root.left, key)
        || search(root.right, key)
```

④ max element of a tree ?

```
int max_element (root) :
    if root == null : return 0
    return max ( root.data, max (max_ele (root.left),
                                 max_ele (root.right)) )
```

Warm ups.

⑤ Given root node, if the tree is skewed or not. → Yes/No

```
bool is_skewed (root) :
  int h = height (root)
  int n = count (root)
  return h == n ;
```

Method 1

Method 2

(more useful)

```
bool  is_skewed (root) :
    if root == null : return True
    if root.left != null and
       root.right != null :
              return False
    if root.left :
        return is_skewed (root.left)
    return is_skewed (root.right)
```

⑥ Check if a tree is full or not ? → every node : 2 children or 0.

```
bool is_full (root) :
    if root == null : return true          ✓
    if root.left == null and root.right == null :   ✓
        return true     ✓

    if root.left and root.right :
        return is_full (root.left) and is_full (root.right)

    return False
```

+1 corner case:

root is a leaf node.

do not forget recursive calls.

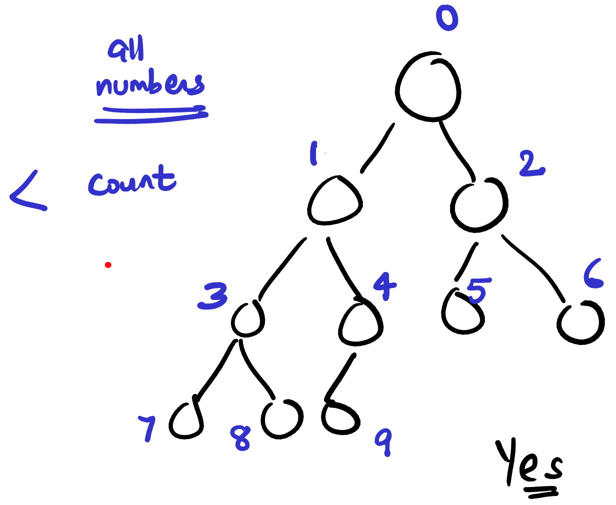
⑦ Is your tree perfect ?

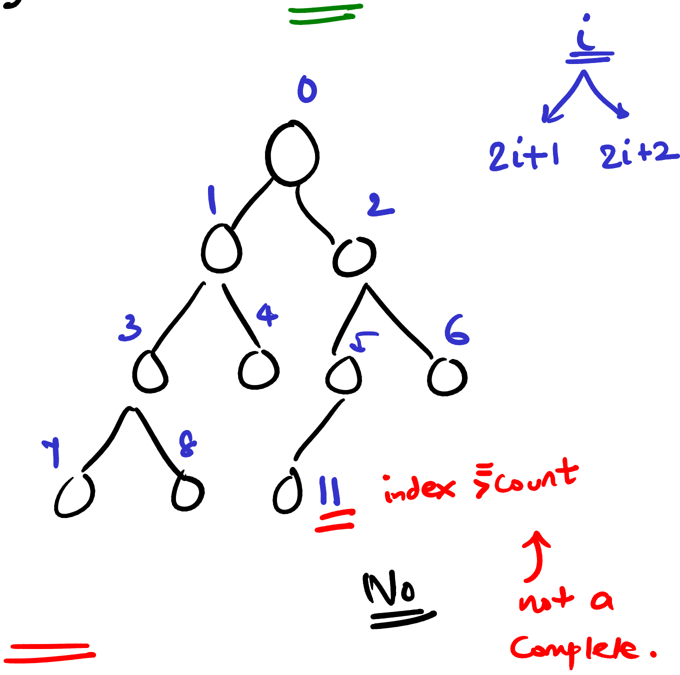
```
bool is_perfect (root) :
    int n = count (root)
    int h = height (root)          O(n)
    return n == 2**h - 1
```

```
is_perfect (root, height(root), 0)

                         count   i
bool is_perfect (root ,  h ,  level ) :
    if root == null : return True

    if root.left == null and
       root.right == null :
          return level + 1 == h ;

    if root.left == null || root.right == null :      I am sure root is not leaf node. *
          return False
                                                 2i+1
    return is_perfect (root.left , h , level + 1)
       and is_perfect (root.right, h , level + 1)
                                                 2i+2
```

⑧ is_complete ( ) :

all levels must be filled except the last one , extra nodes must be left side . } Recursively !





Exercise : is_complete ( )

Recursion ♡ .