Logrithm in CS always have base 2 for logrithms

get → O(1) ST pop → O(1) ST

set → O(1) ST arr[2] = 0

initialization → O(n)

transverse → O(N) T, O(1) S

copy → O(N) ST

insert → add extra space at the end of the array.

copy

O(N) T

O(1) S

DYNAMIC ARRAY
- vector (C++)
- linked link

when the array is expanded → O(N)
after expension → O(1)

AMORTIZE ANALYSIS

Static Vs Dynamic array

linked lists → Types

- array = [2,1,5,6]

has no pointer (tail)

- Linked list (forhead)

singly : 3 → 1 → 4 → 2

Node has two component
1. value
2. pointer of next value.

doubly :

Singlely ⇒ have 1 pointer only.

Doubly ⇒ Each nodes have 2 pointer

Node → • prev ⇐ 1 for previous node
• next ⇐ 1 for next node

☆ can do circular, tailing, rotation

array vs linked link

place back to back

any where in the memory just randomly.

Linked link (ST)

- always have to travel from the first index.

- get → O(i) T (∵ i=index)
  O(1) S
- set → " "

- init → O(n) S T
- copy → O(n) S T
- traverse → O(n) S T
- insert & deletion ⇒ only need to move (create only 1 node)

Value + pointer

the first element → O(i) = index of the node. / O(1) = if know the position

to know the position

Hash tables

object ← Java
Dict ← Python

Insert, delete, search → O(1) avg, O(n) worst

Hash fun
↳ ASCII combo. % 5002 % 2 = 2
length of str / 3

If ~~two~~ collide, PEEISING HASH TABLE

collided idx becomes linked list

Stacks & Queue

- Stacks (push) (pop) → Dynamic array
Last In First Out
LIFO

- Queue
First In First Out
FIFO

ST complexity

Insertion
Deletion → O(1) ST

searching → O(n) T
O(1) S

String "This is a string."

traverse → O(N) T
O(1) S

copy → O(N) ST

get → O(1) S T

mutate → O(n)
If u want O(1), split the array first O(1)
↳ ① copy (O(N))
② do mutation

☆☆☆ Graph

Circle → If a node is repeated when jumpy from one node to another; there exists a circle.
(cyclic group)

☆ to avoid infinite loop ☆

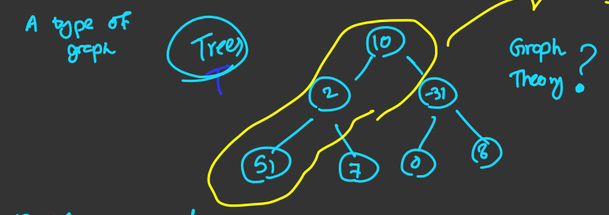
S → O(V+E) V = Vertice, E = Edges

Depth = Deep transversing

T → O(V+E)

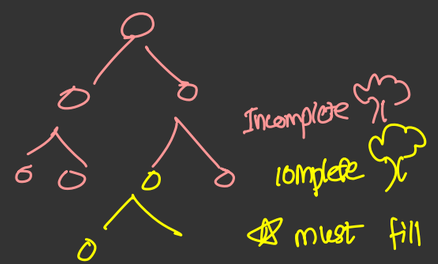
(∵ balanced binary tree) O(log n)

T =
S =



Root node / Top node = 10

child node = 2, -31
51, 7 0, 8



☆ must fill from left to right

☆ always acyclic

☆ each node only has one parent.

e.g binary tree

trinary tree

k-ary tree

Heap are under tree

?? Heaps

Traversing

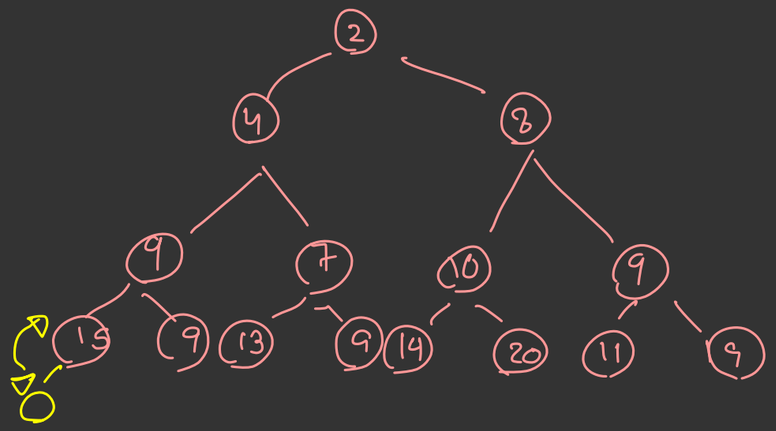
S → O(n) n = total no. of node,

T →

Heaps min & max

min

① root node is the smallest.

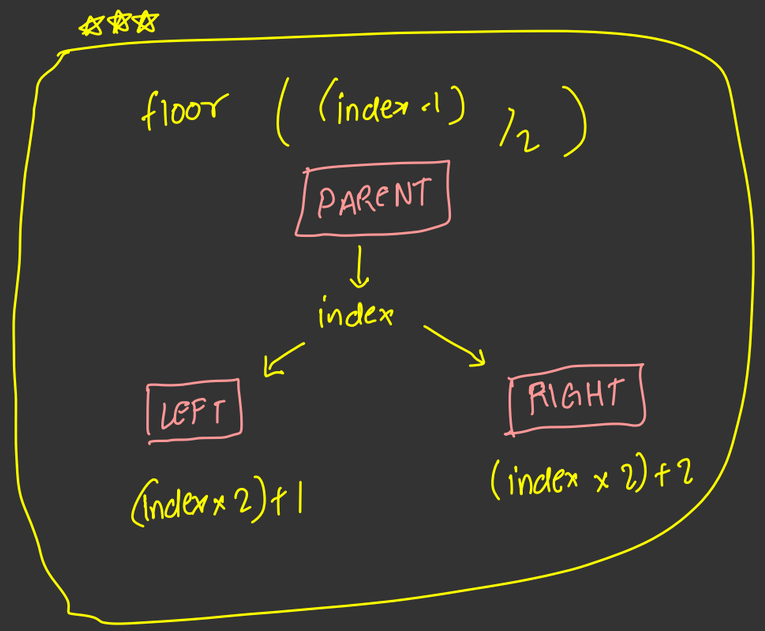


adding an element

① add as a last

② swap w the parent node

③ compare w child & swap w the smaller one.



To remove a node

① move it to the last child. (Sift down)

② pop it off.

Adding a node

① add as a last child.

② move it up (sift up)

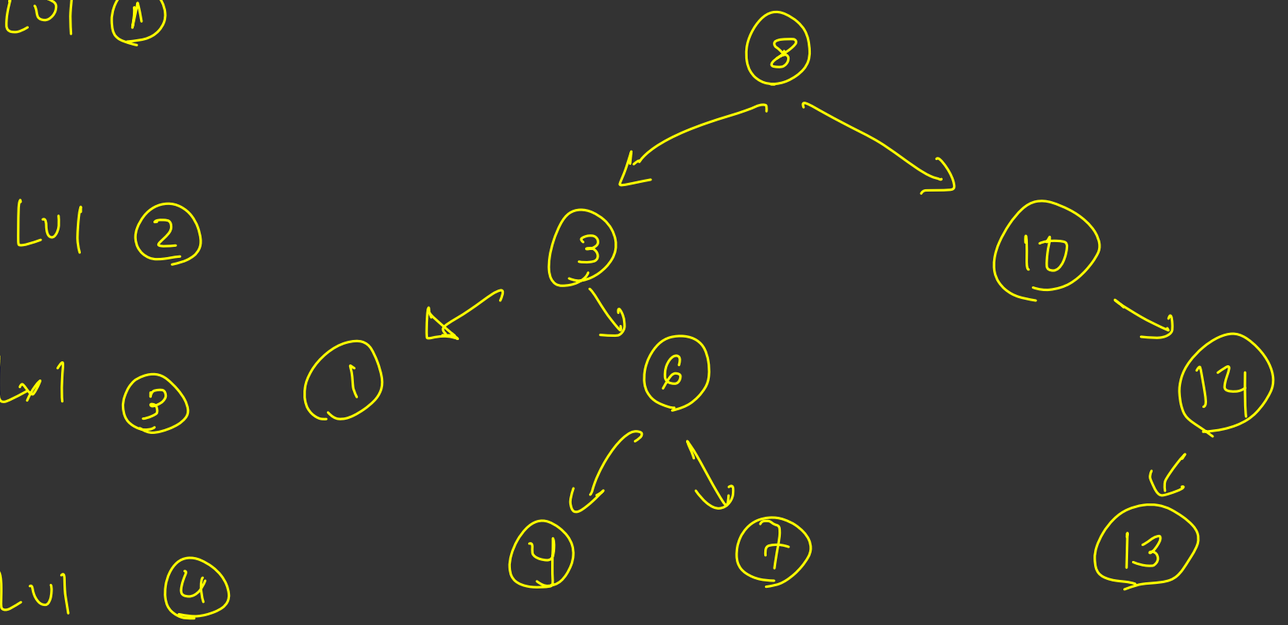
Binary tree

- left child node must be always smaller than the parent node.
- Rt " " " " larger than the parent node.

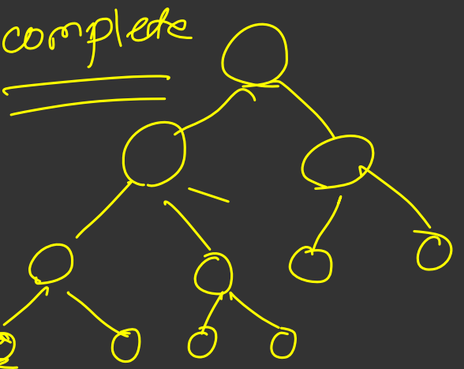
=

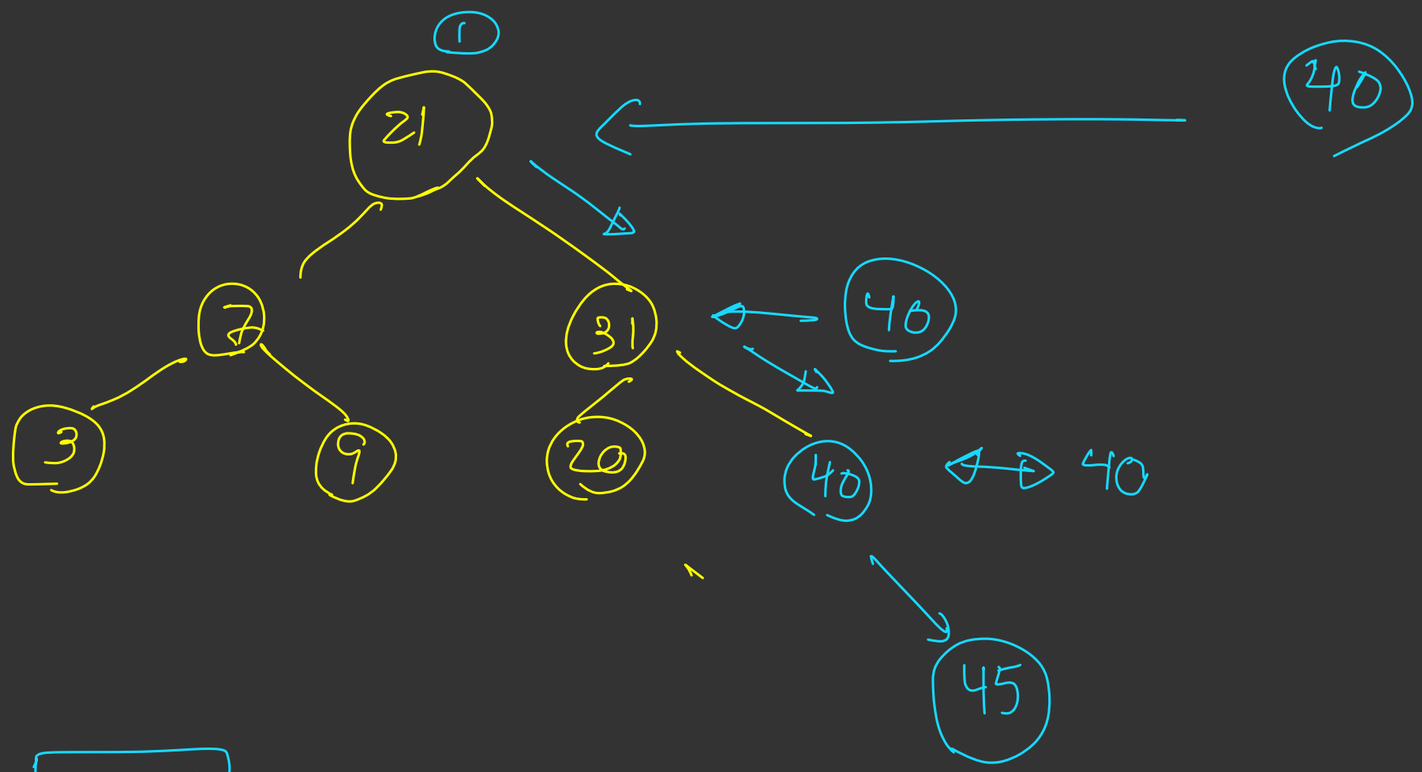
.

Incomplete



complete





Binary Search Trees CONS

① If the input values (sorted) → BST → LL or linked list

Binary Search Tree

	Average	Worst
Space	O(n)	O(n)
Access	O(log n)	O(n)
Search	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)

Array

	Average	Worst
Space	O(n)	O(n)
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Binary Search Tree

```
class node:
    def __init__(self, value = None):
        self.value = value
        self.left_child = None
        self.right_child = None

class Binary_Search_Tree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root == None:          # If BST is empty.
            self.root = node(value)
        else:                          # the tree is not empty ∴ call a private function
            self._insert(value, self.root)
```

- a private function
- recursion will be done here
- so use to avoid confusion

```
    def _insert(self, value, cur_node):
        if value < cur_node.value:
            if cur_node.left_child == None:
                cur_node.left_child = node(value)
            else:
                self._insert(value, cur_node.left_child)
        elif value > cur_node.value:
            # audio (3)
        else:                          # (value == cur_node.value)
            print("Value already in tree")

    def print_tree(self):
        if self.root != None:
            self._print_tree(self.root)

    def _print_tree(self, cur_node):
        if cur_node != None:
            self._print_tree(cur_node.left_child)
            print str(cur_node)
            self._print_tree(cur_node.right_child)
```

```
def height(self): #check the no: of level
    if self.root != None
        return self._height(self.root, 0)
    else
        return 0
def _height_(self, cur_node, cur_height):
    if cur_node == None:    return cur_height:
    else:
        left_neigh = self._height(cur_node.left_child, cur_height+1)
        right_neght =      n        n      .right_child, .. ---
return max(left_child, right_child)
```

recursion exit condition

```
def search(self, value):
    if self.root != None:
        return self._serach(value, self.root)
    else
        return False        # tree doesn't have any value.

def _search(self, value, cur_node)
    if value == cur_node:
        return True
    elif value < cur_node & cur_node.left_child != None:
        return self._search(value, cur_node.left_child)
    elif ___ > ___ & ___.right_child != None:
        return ___.right_child)

    else
        return False (NO value throughout the tree)
```

recursion end point

```
tree = binary_search_tree()
tree = fill_tree(tree)

tree.print_tree()

print "tree height: " + str(tree height)
```

```
def fill_tree(tree, num_elem = 100, max_int = 1000):
    from random import randint
    for _ in range(num_elem):
        cur_elem = randint(0, max_int)
        tree.insert(cur_elem)
    return tree
```