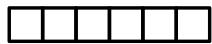


## Introduction to Trees

Linear data structures: Array, Stack, Linked List, Queue

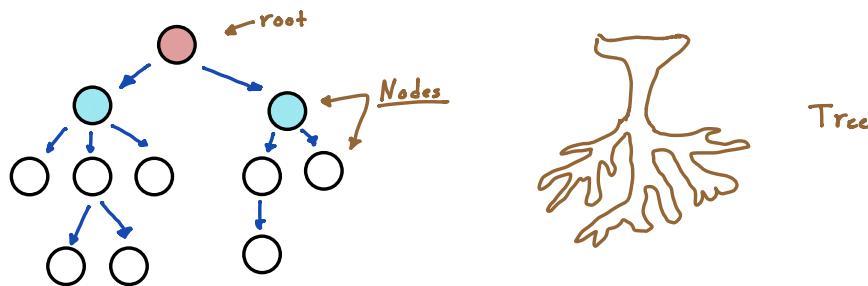


Array 5

How should I decide which data structure to use?

- What needs to be stored?
- Cost of operation
- Memory usage
- Ease of implementation

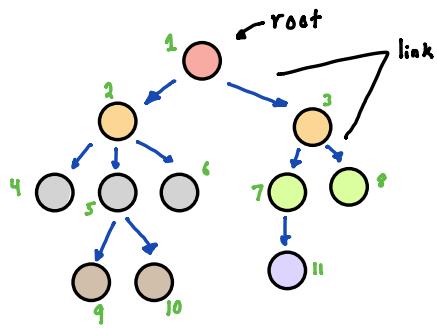
Tree used for Hierarchical data representation



Tree

- Non-linear

- Node contains data and possibly reference to another node



### Common Terminology

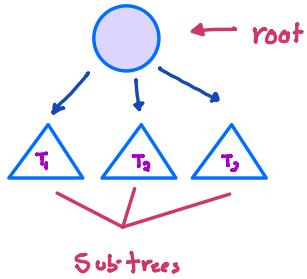
Note: it's don't represent anything in our example

- 2 and 3 linked to Node 1 and our "children" of node 1
- Node 1 is called "Parent" of Nodes 2 and 3
- Children of the same parent are called siblings
- Any node in the tree w/o a child is called a leaf Node
- All nodes with 1 child are called internal nodes
- Parent of a parent is called Grandparent
- When walking in the tree we can only go in one direction
- Ancestor and descendant
- Cousins, uncles

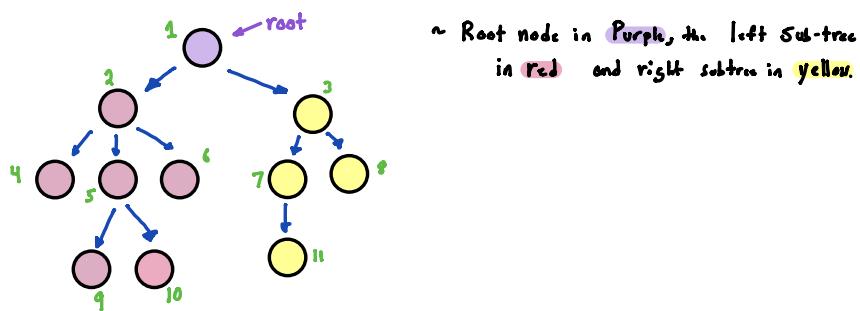
## Properties of Trees

### - Recursive data structure

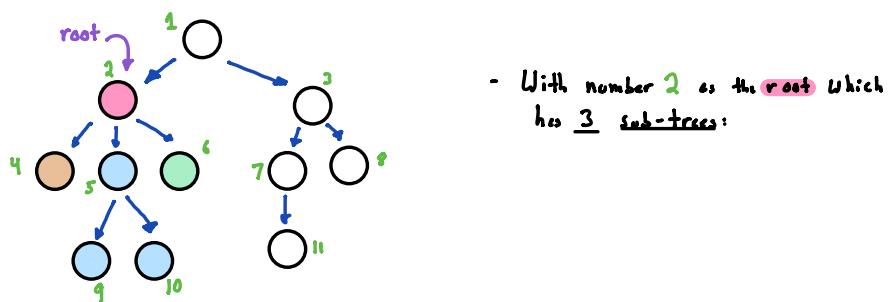
- We can define a tree recursively as a structure that consists of a distinguished node called a root and some sub-trees and the arrangement is such that root of the tree contains link to roots of all the sub-trees ( $T_1, T_2, T_3$ ) in our diagram.



Example from our diagram above:

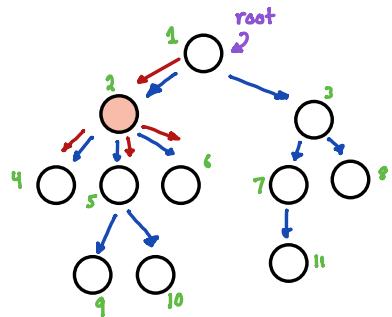


We can also look further at the sub-tree.



## Properties of Trees (Continued)

- In a tree with  $n$  nodes, there will be exactly  $n-1$  links or edges
- All nodes except the root node will have exactly one edge



At node 2, we have one incoming link and 3 outgoing links.

## Depth and Height Properties

Depth of  $x$ : length of path from root to  $x$

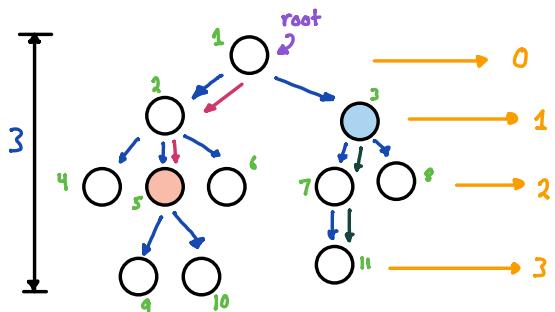
or

Number of edges in path from root to  $x$

Height of  $x$ : Number of edges in longest path from  $x$  to a leaf.

Height of tree: Height of root node

Example of Depth and Height:



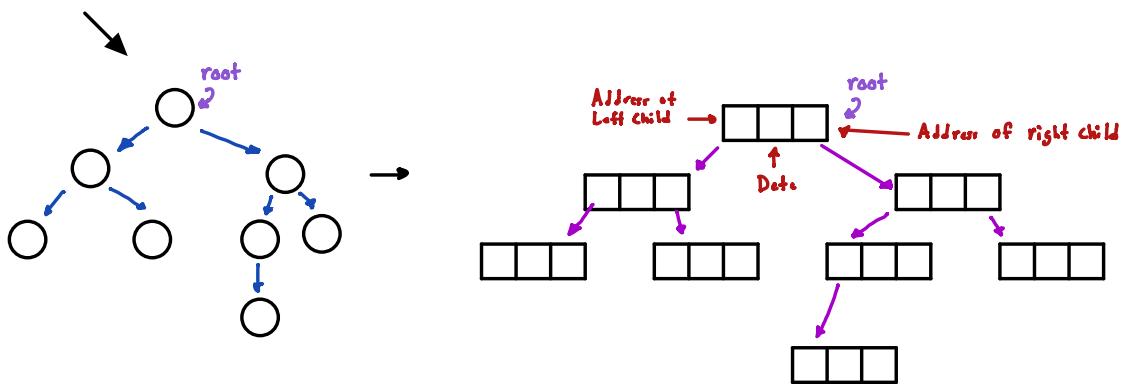
Depths of nodes in Orange

- In node 5 we have 2 edges in the path from root. So the depth is 2.
- In node 3, the longest path from this node to any leaf is 2.

## Different types of Trees

### Binary Tree:

- a tree in which each node can have at most 2 children
- most famous



### Code in Python:

#### Node Class:

##### Class Node:

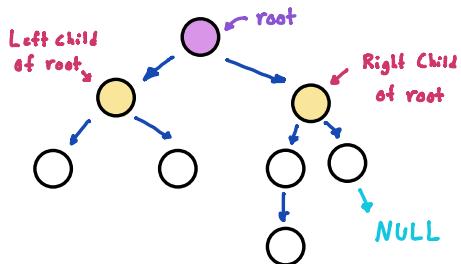
```
def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
```

### Application of trees

- 1) Storing naturally hierarchical data  
→ e.g. file system
- 2) Organizing data for quick search, insertion, deletion  
→ e.g. Binary Search trees
- 3) Trie  
→ Dictionary
- 4) Network Rooting Algorithm

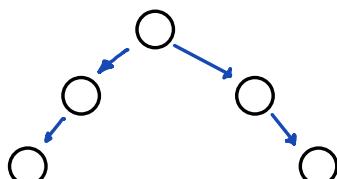
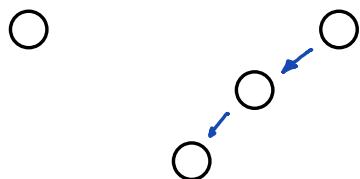
## Binary Tree

### Properties:



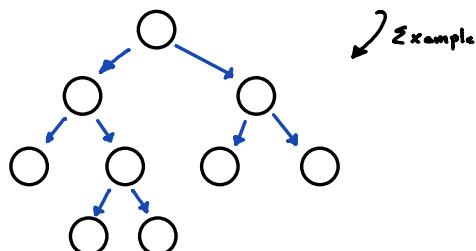
- A node may have both right and left child or a node can have either a right or Left child
- In a program we'll set reference of empty child to node
- For Leaf, we can set both Left and right to NULL

### Different types of Binary trees



### Strict/Proper binary Tree

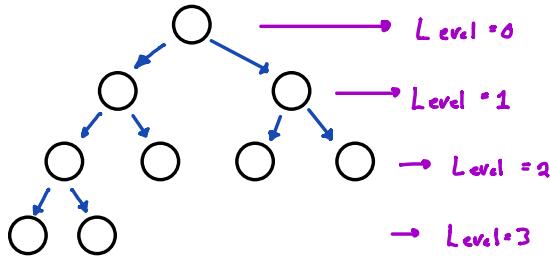
↳ each node can have either 2 or 0 children.



### Complete Binary Tree

all levels except possibly the last are completely filled  
and all nodes are as left as possible

- max number of nodes at level  $i = 2^i$



### Perfect Binary Tree

Maximum number of nodes  
in a binary tree with height  $h$

$$= 2^0 + 2^1 + \dots + 2^h$$

$$= 2^{h+1} - 1$$

$$= 2^{(\text{no. of levels})} - 1$$

$$n = 2^{h+1} - 1 \quad n = \text{# of nodes}$$

$$\rightarrow 2^{h+1} = (n+1)$$

$$h = \log_2(n+1) - 1$$

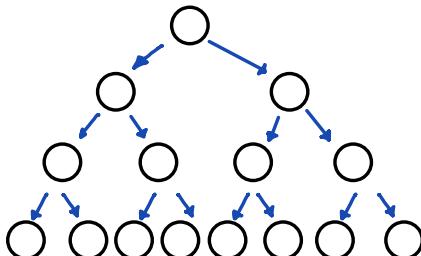
$$h = \log_2(15+1) - 1$$

$$h = \log_2 16 - 1$$

$$h = 4 - 1$$

$$h = 3 \qquad 2^x = 16$$

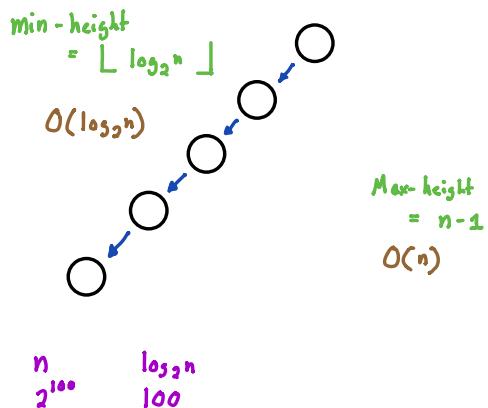
- Solve for  $x$



### Height of Complete Binary tree

$$= \lfloor \log_2 n \rfloor \sim \text{floor function}$$

Time Complexity:  $O(h)$   $\Rightarrow$   $h$  = height of tree



### Balanced Binary Tree

- difference b/w height of Left and right subtree for every node is not more than K (mostly 1)

Height  $\Rightarrow$  number of edges in longest path from root to a leaf

Height of an Empty tree = -1

Height of tree with 1 node = 0

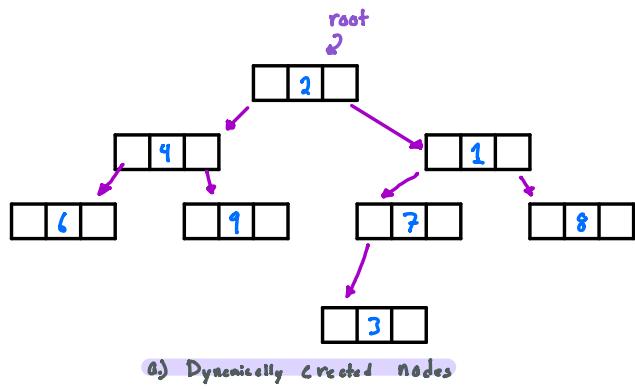
$$\text{diff} = | h_{\text{left}} - h_{\text{right}} |$$

We can implement Binary tree using:

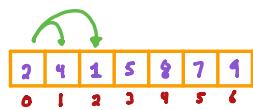
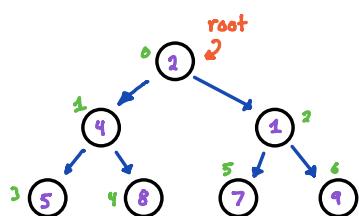
a) Dynamically created nodes

b) arrays

↳ used for heaps



a) Dynamically Created nodes



for node at index  $i$ ,  
left-child-index =  $2i + 1$   
right-child-index =  $2i + 2$  ] → in a Complete Binary tree

## Running Time of different Data Structures

★ Storing integers (for simplicity) ★

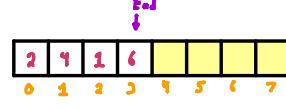
### Array (Unsorted)

#### Time Complexity

Search ( $n$ )

$O(n)$

$n$  being # of elements in the array

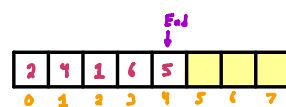


Insert ( $n$ )

$O(1)$

Constant time

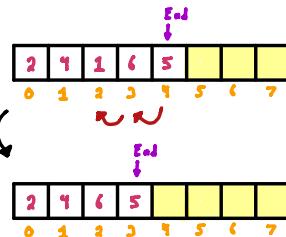
Insert (5)



Removal ( $n$ )

$O(n)$

Remove (4)



We have to shift all records to the right by one position.

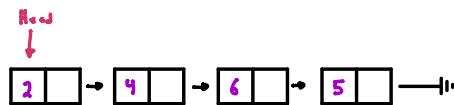
### Linked List

#### Time Complexity

Search ( $n$ )

$O(n)$

$n$  being the # of nodes in the linked list



Insert ( $n$ )

$O(1)$

At head

Removal ( $n$ )

$O(n)$

For Search,  $n$  Comparisons in Worst Case

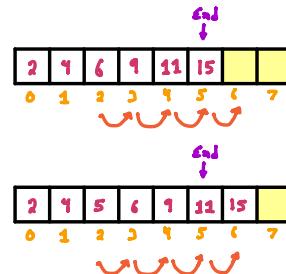
Let's say cost of 1 comparison =  $10^{-6}$  sec     $n = 10^6$ ,  $T = 100$  sec

We can perform a Binary Search of an array if it's sorted

- Can perform in  $O(\log n)$

<u>Sorted Array</u>	
<u>Time Complexity</u>	
Search ( $x$ )	$O(\log n)$
Insert ( $x$ )	$O(n)$
Removal ( $x$ )	$O(n)$

*← Search function is improved!*

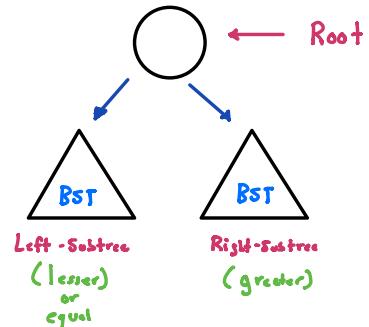


- Insertion and Removal of an element still takes  $O(n)$  operations in an Sorted Array.

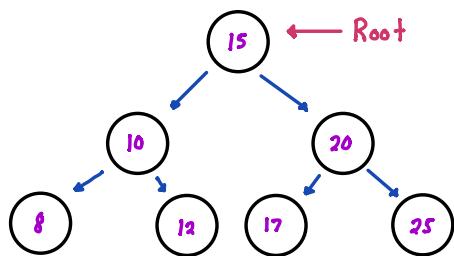
## Binary Search Tree

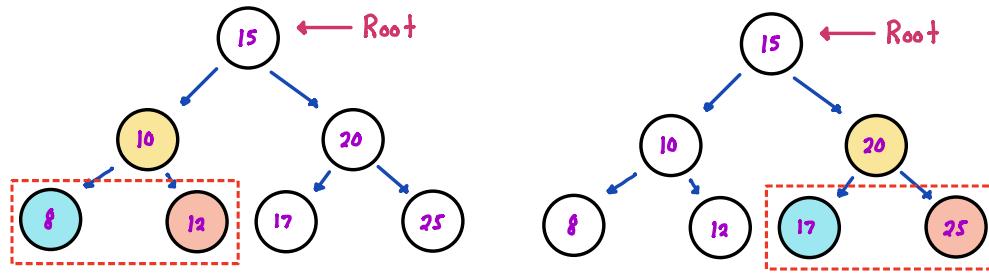
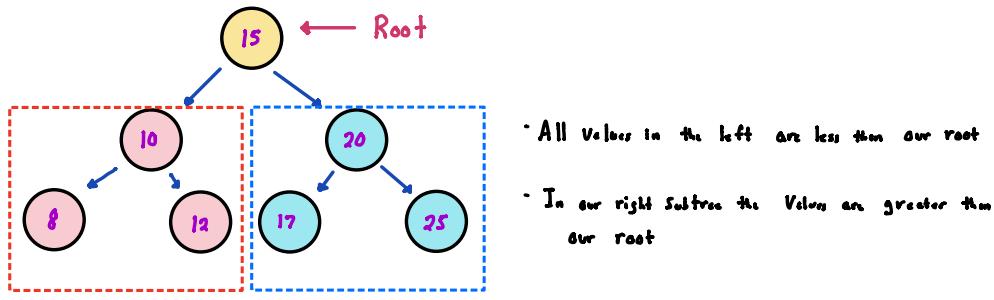
### Binary Search Tree (BST):

- A binary tree, in which for each node, value of all the nodes in left subtree is lesser or equal and value of all nodes in right subtree is greater.



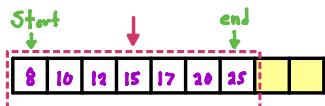
Example: Is this a BST?



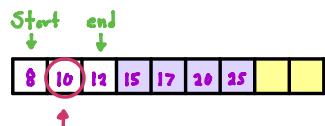
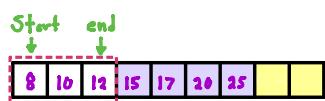


- Both are good ✓ · Binary Search Tree

What do we do in Binary Search of an Array?



Search(10)



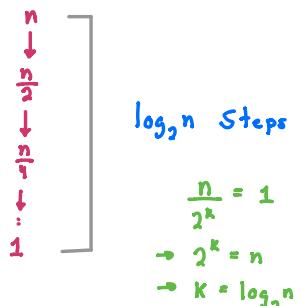
- Search Space: we start in the middle and determine if our search value is lesser or greater. Then we will adjust our search area.

- Our Search Space is reduced now

- Then we will compare in the middle again, and then Base we have a match.

In Binary Search, we start:

- ~ Keep reducing the search space by half until we find our value or get to one value and return.



$$\begin{aligned} \frac{n}{2^k} &= 1 \\ \rightarrow 2^k &= n \\ \rightarrow k &= \log_2 n \end{aligned}$$

left or e 14.22