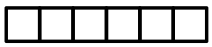
Introduction to Trees

Chapter 8

Linear data structures: Arrays, Stacks, Linked List, Queue

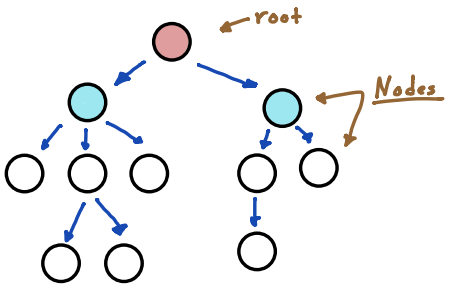


Array

How should I decide which data structure to use?

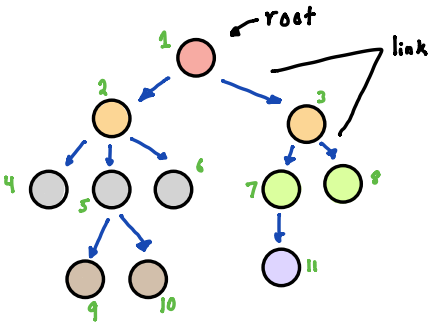
- What need to be stored?
- Cost of operation
- memory usage
- Ease of implementation

Tree used for Hierarchical data representation ~ Some objects "above" and "below"



Tree
~ from root

- Non-linear
- Node contains data and possibly reference to another node
- Natural Organization for data



Code in Python:

~ for creating a Node class

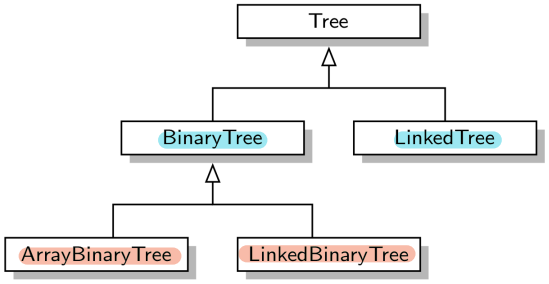
```
Class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data
```

Common Terminology

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

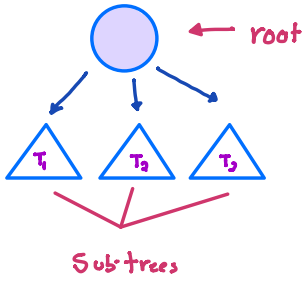
- 2 and 3 links to Node 1 and our "children" of node 1
- Node 1 is called "Parent" of Nodes 2 and 3

- 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 descendent
- Cousins, uncles

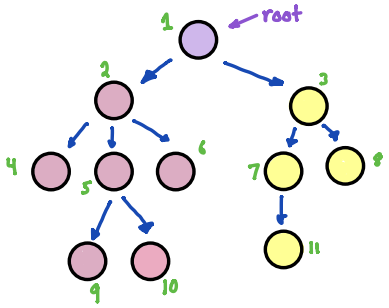


Properties of Trees

- 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 contain link to roots of all the sub-trees (T_1, T_2, T_3) in our diagram.

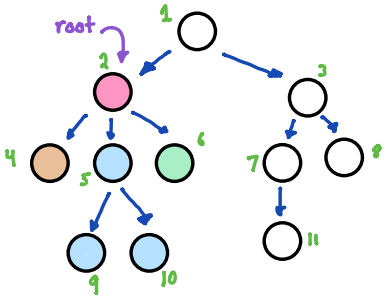


Example from our diagram above:



~ Root node in Purple, the left sub-tree in red and right subtree in yellow.

We can also look further at the sub-tree



- With number 2 as the root which has 3 sub-trees: (5, 6, 7)

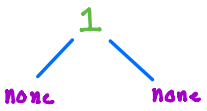
Basic Tree Python Implementation

Creating a Node Class:

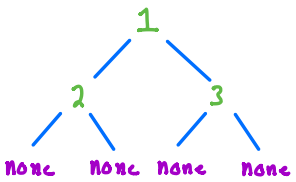
```
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key


root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
```

How it visually looks

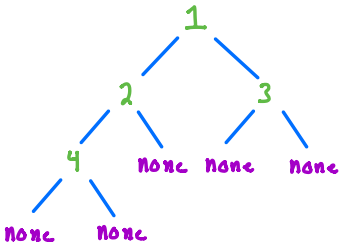
root = Node(1)



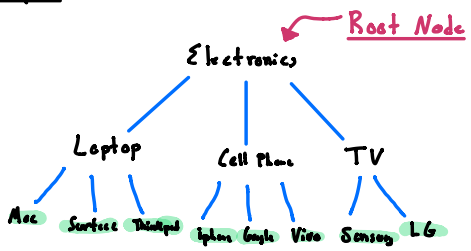
root.left = Node(2)
root.right = Node(3)



root.left.left = Node(4)



More Basic Tree Python Example

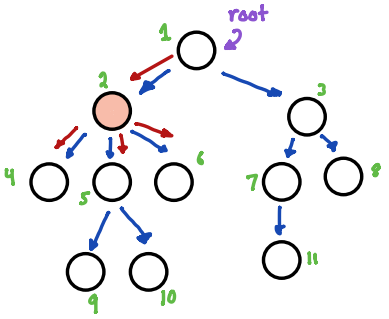



```python
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        self.parent = None

    def get_level(self):
        level = 0
        p = self.parent
        while p:
            level += 1
            p = p.parent

        return level

    def print_tree(self):
        spaces = ' ' * self.get_level() * 3
        prefix = spaces + "|__" if self.parent else ""
        print(prefix + self.data)
        if self.children:
            for child in self.children:
                child.print_tree()

    def add_child(self, child):
        child.parent = self
        self.children.append(child)

def build_product_tree():
    root = TreeNode("Electronics")

    laptop = TreeNode("Laptop")
    laptop.add_child(TreeNode("Mac"))
    laptop.add_child(TreeNode("Surface"))
    laptop.add_child(TreeNode("Thinkpad"))

    cellphone = TreeNode("Cell Phone")
    cellphone.add_child(TreeNode("iPhone"))
    cellphone.add_child(TreeNode("Google Pixel"))
    cellphone.add_child(TreeNode("Vivo"))

    tv = TreeNode("TV")
    tv.add_child(TreeNode("Samsung"))
    tv.add_child(TreeNode("LG"))

    root.add_child(laptop)
    root.add_child(cellphone)
    root.add_child(tv)

    root.print_tree()

if __name__ == '__main__':
    build_product_tree()
```

```
Electronics
   |__Laptop
      |__Mac
      |__Surface
      |__Thinkpad
   |__Cell Phone
      |__iPhone
      |__Google Pixel
      |__Vivo
   |__TV
      |__Samsung
      |__LG
```

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

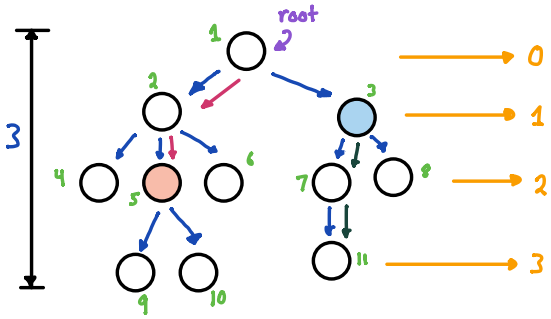
or

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:



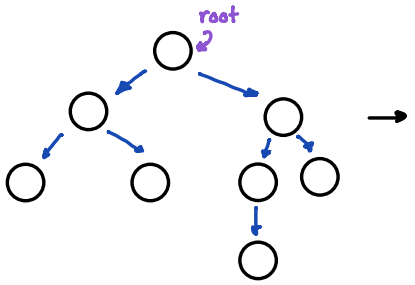
Depths of nodes in Orange

- 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.

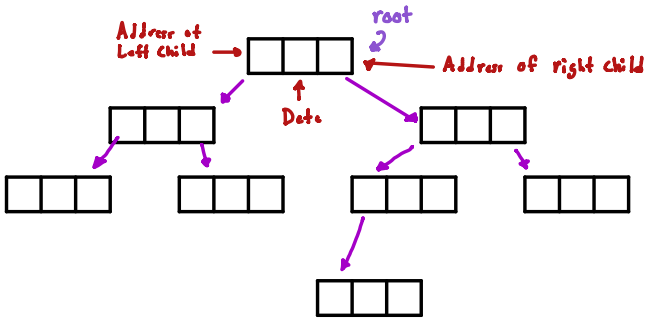
Different types of Trees

Binary Tree:

- a tree in which each node can have at most 2 children
- Each child is labeled as either being Left or Right
- most famously



How a tree would look:



Application of trees

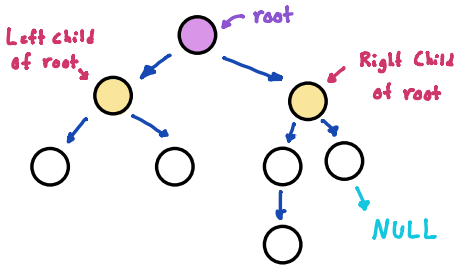
1) Storing naturally hierarchical data
   → e.g. file system

2) Organizing data for quick search, insertion, deletion
   → e.g. Binary Search trees

3) Trie
   → Dictionary

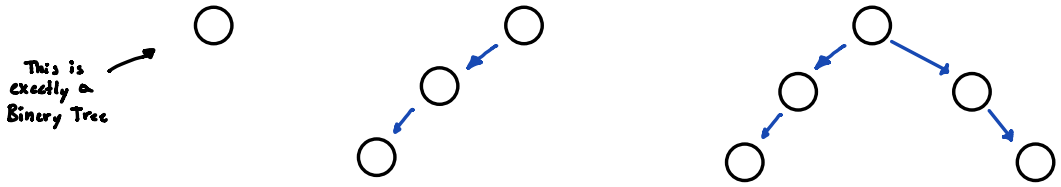
4) Network Routing Algorithm

Binary Tree Properties:



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

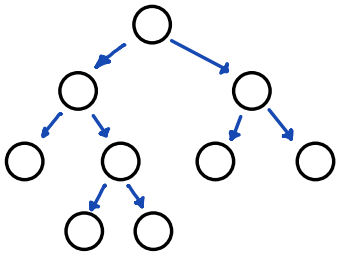
Different types of Binary trees



Strict/Proper Binary Tree

↳ each node can have either 2 or 0 children.

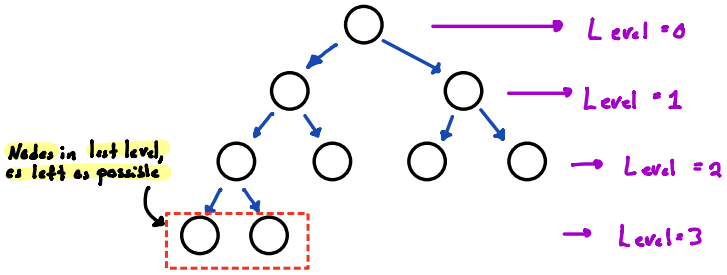
Example



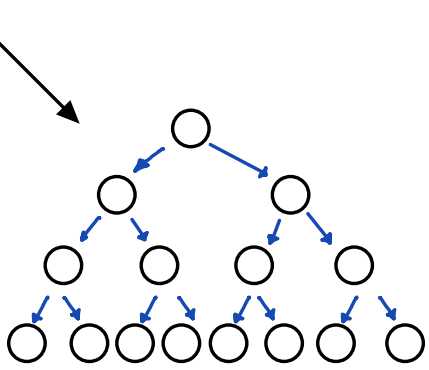
Complete Binary Tree

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 ~ Full Tree



Maximum number of nodes in a binary tree with height h

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

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

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

$n = 2^{h+1} - 1$ ← n = # of nodes

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

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

↓ is

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

$h = \log_2 16 - 1$

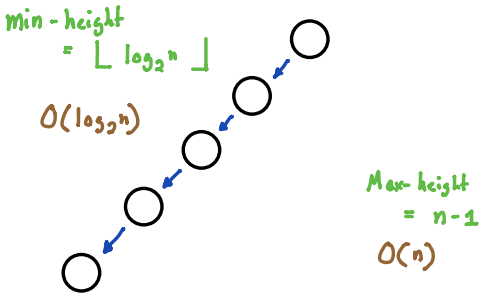
$h = 4 - 1$ $\quad 2^x = 16$

$h = 3$ $\quad$ - Solve for x

Height of Complete Binary tree

$= \lfloor \log_2 n \rfloor$ ← floor function (discrete Math?)

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



n	$\log_2 n$
2^{100}	100

Balanced Binary Tree

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

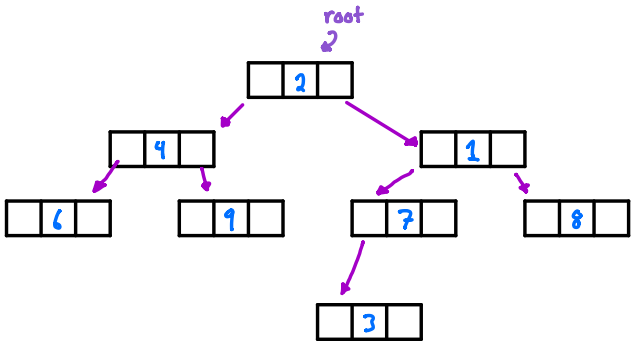
> Height → number of edges in longest path from root to a leaf
>
> Height of an Empty tree = -1
> Height of tree with 1 node = 0

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

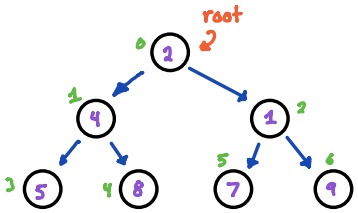
We can implement Binary tree using:

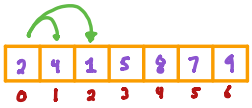
a) Dynamically created nodes

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

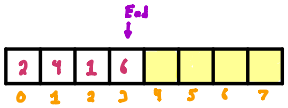
★ Storing integers (for simplicity) ★

Array (unsorted)

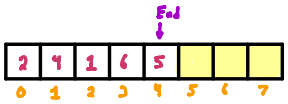
Time Complexity	
Search (x)	O(n)
Insert (x)	O(1)
Removal (x)	O(n)

n being # of elements in the array

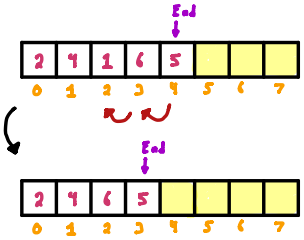
Constant time



Insert(5)



Remove(1)



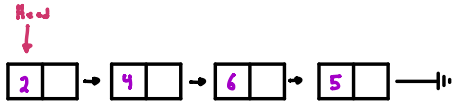
We have to shift all records to the right by one position.

Linked List

Time Complexity	
Search (x)	O(n)
Insert (x)	O(1)
Removal (x)	O(n)

n being the # of nodes in the linked list

At head



For search, n comparisons in worst case

Let's say cost of 1 comparison = 10^{-6} sec $n = 10^8$, T = 100 sec

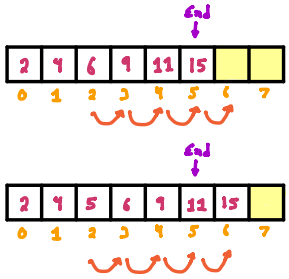
We can perform a **Binary Search** of **an array** if it's **sorted**

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

Array (Sorted)

Time Complexity	
Search (x)	$O(\log n)$
Insert (x)	$O(n)$
Remove (x)	$O(n)$

= Search function is improved!



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

for n records, $\log_2 n$ comparisons

if 1 comparison = 10^{-6} sec

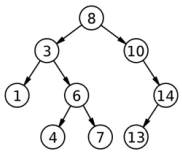
$n = 2^{31} \rightarrow 31 \times 10^{-6}$ sec

Binary Search Tree (BST)

Time Complexity

- making sure the tree is **balanced**

Search (x)	$O(\log n)$
Insert (x)	$O(\log n)$
Remove (x)	$O(\log n)$



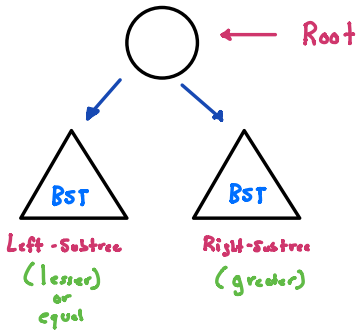
Allows for **faster operations**

Binary Search Tree		
	Average	Worst Case
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*)

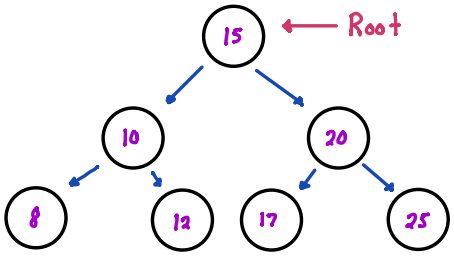
Array		
	Average	Worst Case
Space	O(*n*)	O(*n*)
Access	O(*1*)	O(*1*)
Search	O(*n*)	O(*n*)
Insertion	O(*n*)	O(*n*)
Deletion	O(*n*)	O(*n*)

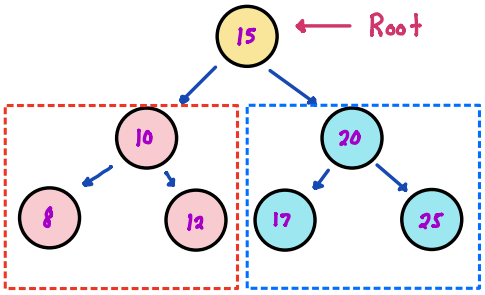
Binary Search Tree (BST):

→ 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.

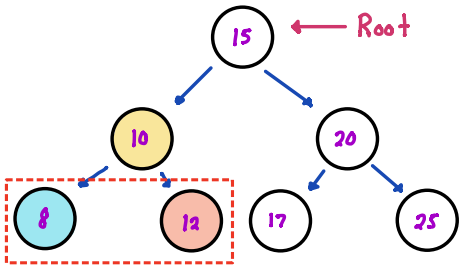


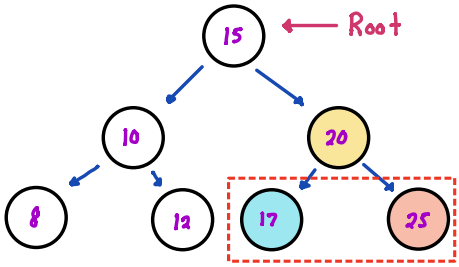
Example: Is this a BST?





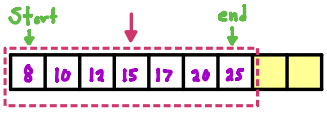
- All values in the left are less than our root
- In our right subtree the values are greater than our root





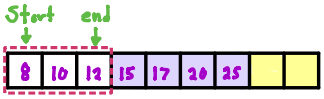
- Both are good ✓ · Binary Search Tree

What do we do in Binary Search?

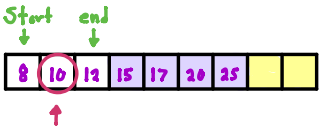


Search(10)

- 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 to the middle again, and then Boom 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.

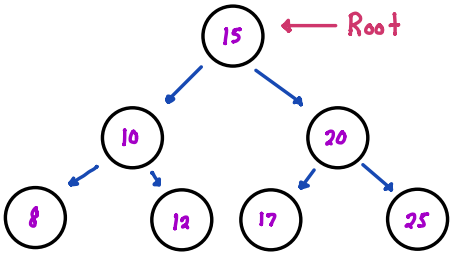
$\log_2 n$ Steps

$\frac{n}{2^k} = 1$

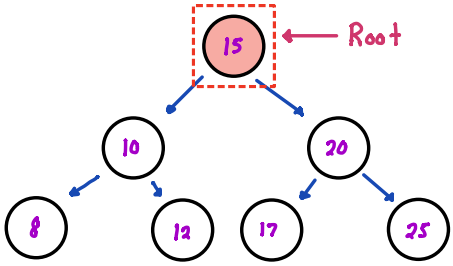
$\rightarrow 2^k = n$

$\rightarrow k = \log_2 n$

Implementing Binary Search in a Tree:

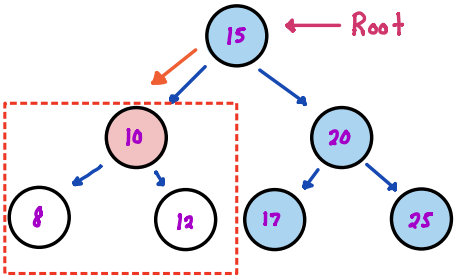


Search (12)



- For searching we will start at the root
- Compare it with root, if it is less than our root we will go Left and if it's greater than our root we will proceed to the right-Subtree

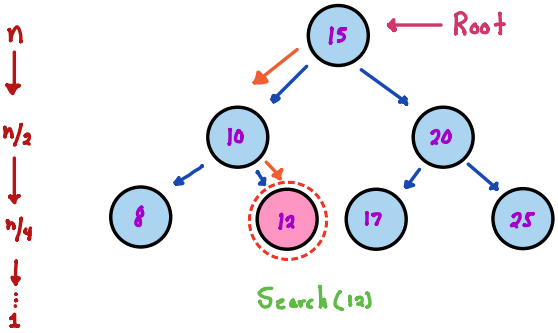
Search (12)



- We know that number 12 can only exist in this subtree only and anything apart from this subtree can be discarded.

Search (12)

- Then we compare our node of 10 to our search value of 12, since 12 is greater we go to the right and discard everything else.



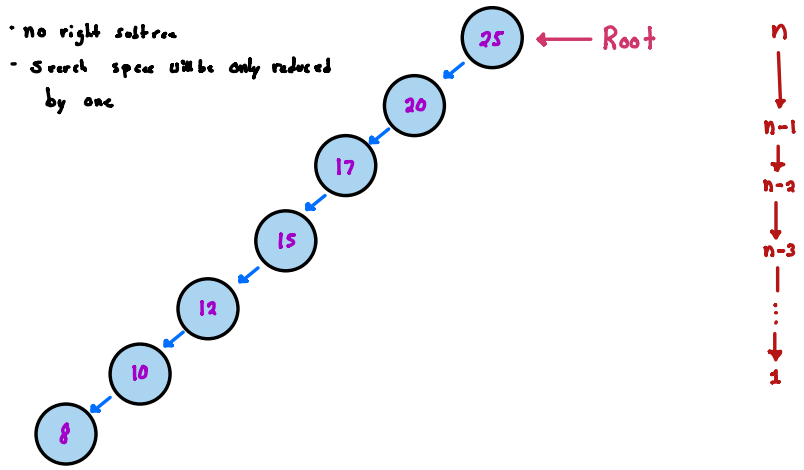
Search (12)

- We compare the value at the node and we have a match.

Searching a tree you either go Left or Right

Unbalanced Binary Search Tree:

- no right subtree
- Search space will be only reduced by one

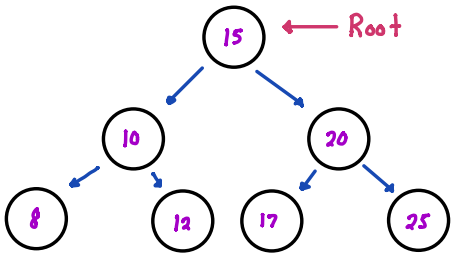


In Binary Search tree in average case cost of Search, insertion or deletion is $O(\log n)$

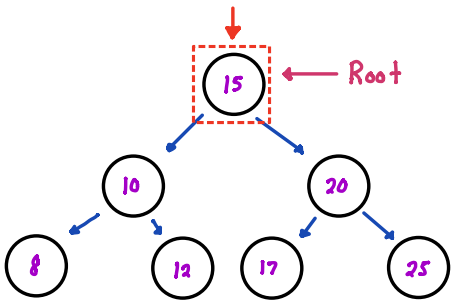
Worst case is $O(n)$

Binary Tree Insertion

To insert some records in Binary Search Tree, we will first have to find the position in $O(\log n)$

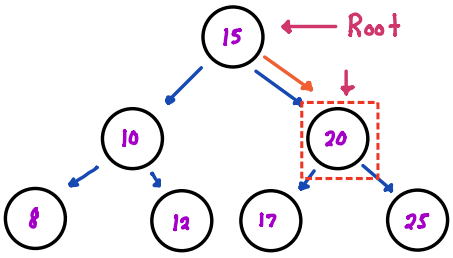


Insert(19)



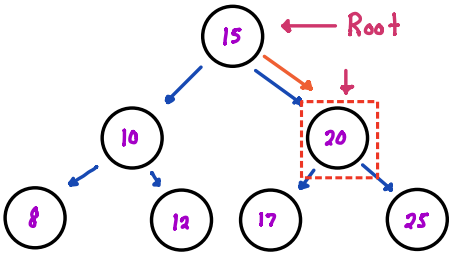
Insert(19)

- If the Value to be inserted is lesser or equal, if there is no child, insert as left child or go left
- If the Value to be inserted is greater and if there is no child, insert as right child or go right.



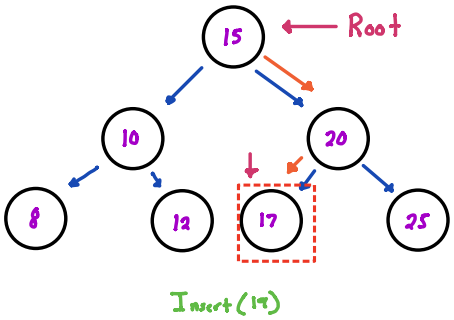
Insert(19)

- In our example 19 is greater than our node of 15, so go right



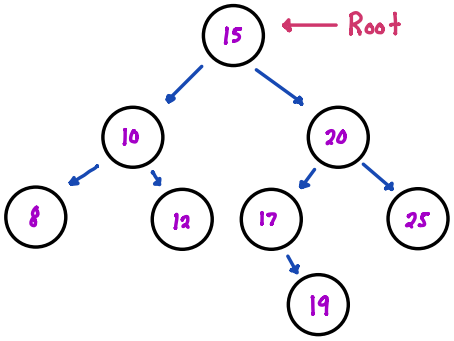
Insert(19)

- At node of 20, 19 is lesser so we go left

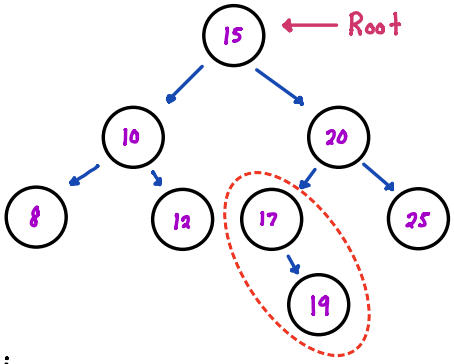


Insert(19)

- Our Value 19 is greater than 17 so it should go in right of 17.



- There is no right child of 17, So, We will create a node with Value 19 and link this node with Value 17 as right child (using reference, like linked lists).



- no shifting is needed like an array

- Creating a link will take constant time

Delete a Node:

- Search: O(logn)
- Deleting the node will only mean adjusting some links

- Binary Tree gets unbalanced during insertion and deletion

```
class Node(object):
    def __init__(self,value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

#            1
#          /   \
#        2       3
#      / \     / \
#     4   5   6   7


# set up Tree
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)
tree.root.right.left = Node(6)
tree.root.right.right = Node(7)
```

Example:

Node 3

```
class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_print(tree.root, "")
        elif traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        elif traversal_type == "postorder":
            return self.postorder_print(tree.root, "")

        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False

    def preorder_print(self, start, traversal):
        """Root->Left->Right"""
        if start:
            traversal += (str(start.value) + "-")
            traversal = self.preorder_print(start.left, traversal)
            traversal = self.preorder_print(start.right, traversal)
        return traversal

    def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        return traversal

    def postorder_print(self, start, traversal):
        """Left->Right->Root"""
        if start:
            traversal = self.postorder_print(start.left, traversal)
            traversal = self.postorder_print(start.right, traversal)
            traversal += (str(start.value) + "-")
        return traversal
```

```
# 1-2-4-5-3-6-7-
# 4-2-5-1-6-3-7
# 4-2-5-6-3-7-1
#             1
#           /       \
#          2          3
#         /  \      /   \
#        4    5     6   7

# Set up tree:
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)
tree.root.right.left = Node(6)
tree.root.right.right = Node(7)

print(tree.print_tree("preorder"))
print(tree.print_tree("inorder"))
print(tree.print_tree("postorder"))
```

Lucid Programming Ude : Video 34

Binary Tree Traversal

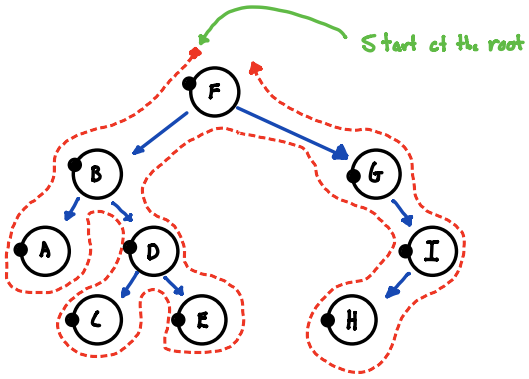
Tree Traversal: Process of Visiting (checking or updating) each node in a tree data structure exactly, once.

- Trees can be traversed in multiple ways, depth-first or Breadth-first order

- 3 common ways to traverse them in depth-first order:
  - in-order
  - pre-order
  - post-order

Pre-Order Traversal

1. Check if the current node is empty/null
2. Display the data part of the node
3. Traverse the left subtree by recursively calling the pre-order function
4. Traverse the Right subtree by recursively calling the pre-order function

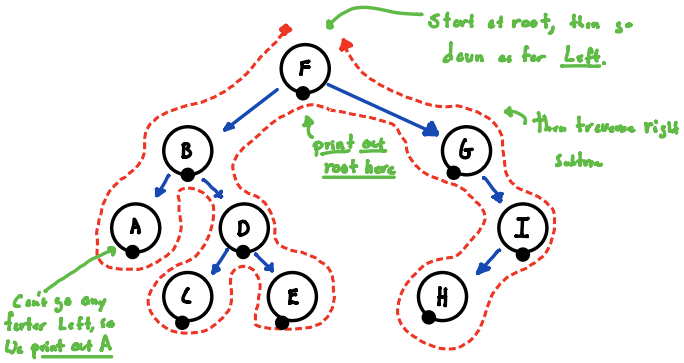


Pre-Order: F, B, A, D, C, E, G, I, H

Code for Pre-Order Traversal

In-Order Traversal

1. Check if the current node is empty/null
2. Traverse the left subtree by recursively calling the pre-order function
3. Display the data part of the node
4. Traverse the Right subtree by recursively calling the pre-order function

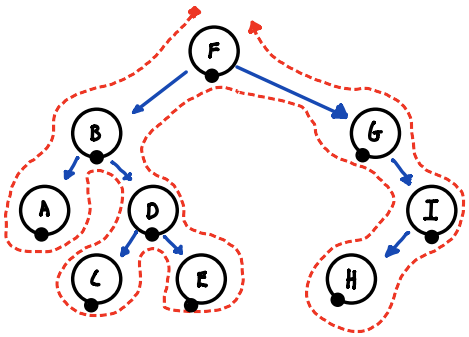


In-Order: A, B, C, D, E, F, G, H, I

In-Order Traversal Code

Post-Order Traversal

1. Check if the current node is empty/null
2. Traverse the left subtree by recursively calling the pre-order function
3. Display the data part of the node
4. Traverse the Right subtree by recursively calling the pre-order function



Post-Order Traversal Code