Lecture 5: Tree

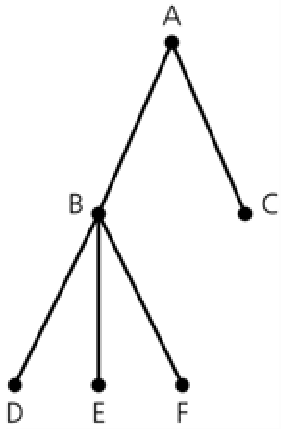
김수경

이화여자대학교 인공지능융합전공 소속

저작권 안내

Definition of Tree

- A general tree *T* is partitioned into disjoint subsets:
  - A single node *r*, the **root**
  - Sets of general trees, called **subtrees** of *r*

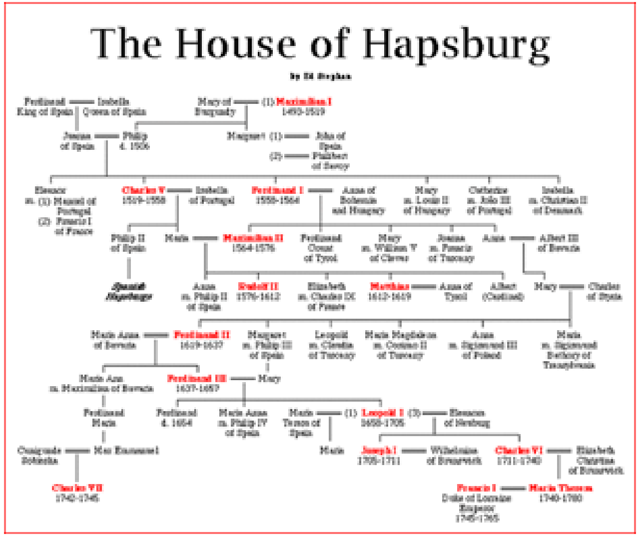


- Terminology
  - node (vertex)
  - edge
  - parent
  - child
  - siblings
  - root
  - leaf
  - ancestor
  - descendant
  - subtree

upstage Education

Tree Examples

[1]



[2]

[1] http://18president.pa.go.kr/cheongwadae/organization/government.php
[2] https://kr.pinterest.com/pin/146578162866571659/

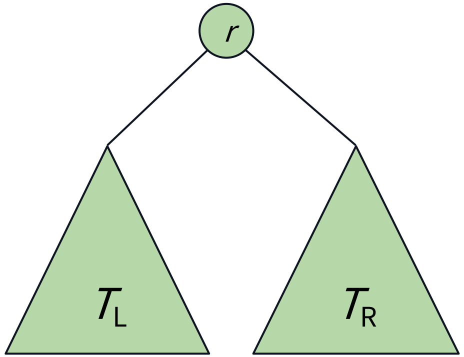
01

Binary Tree

upstage Education

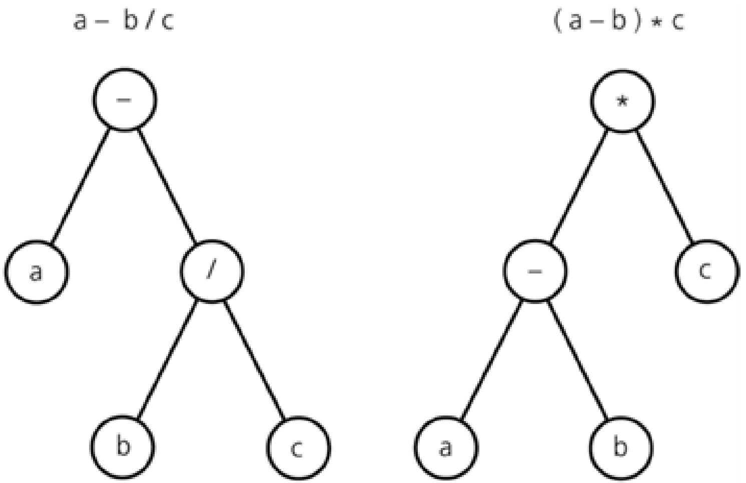
Definition of Binary Tree

- T is **binary tree**, if
  - T is empty, or
  - T is partitioned into three disjoint subsets:
    - A single node r, the root
    - Up to two sets that are **binary trees**, called left and right subtrees of r



An Example of Binary Tree

- Algebraic expressions

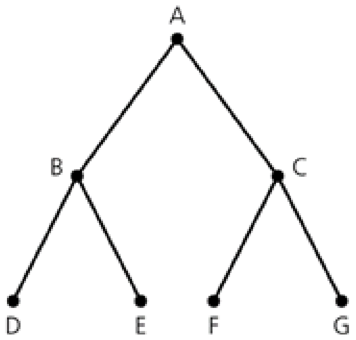


[1]

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week10.pdf

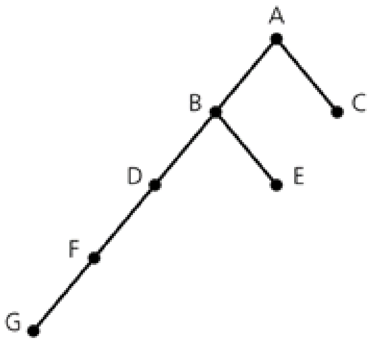
Height of a Tree

- Def. **Height** of a Tree: the **number of nodes** on the longest path from the root to a leaf.



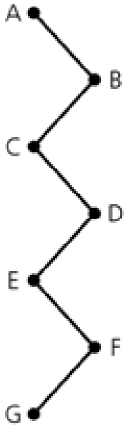
[1]

Height: 3



Height: 5

Height: 7

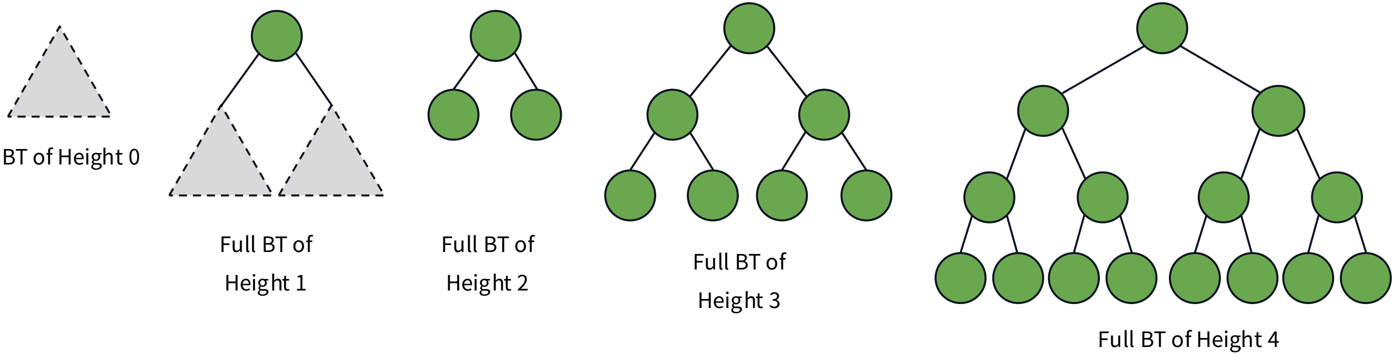


[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week10.pdf

upstage Education

Full Binary Tree

- <u>Def</u>. A binary tree T of height h is called **Full binary tree**, if
  - it is empty ($h = 0$), or
  - both its left and right subtrees are full (of height $h - 1$).

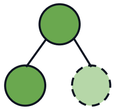


So now, you may feel why this is called **full** binary tree:

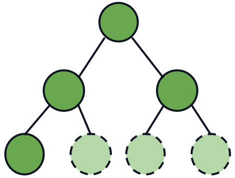
it is filled with nodes at all possible positions!

Complete Binary Tree

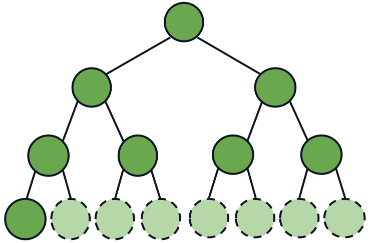
- Def. A binary tree T is called **Complete binary tree**, if
  - T is full down to level h - 1, and
  - with level h filled in from left to right.



Complete BT of Height 2



Complete BT of Height 3



Complete BT of Height 4

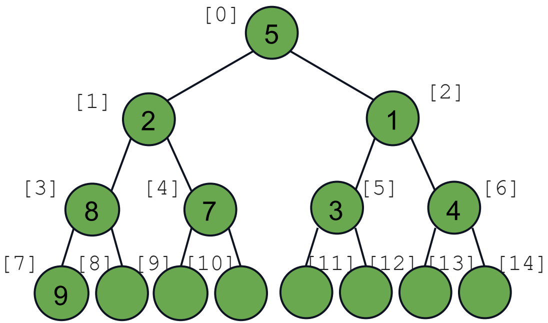
Must be filled (to be complete)

Optionally filled (to be complete)

upstage Education

Array-based Implementation

- Recall: Array is a fundamental data structure with **contiguous fixed-length chunk** of memory space.
  - When we implement `List` with an array, it was straightforward to map the indices.
  - How can we assign designated indices for a binary tree?



- Let's assume the tree is full, and assign the indices from root, left to right.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
5	2	1	8	7	3	4	9	...	...

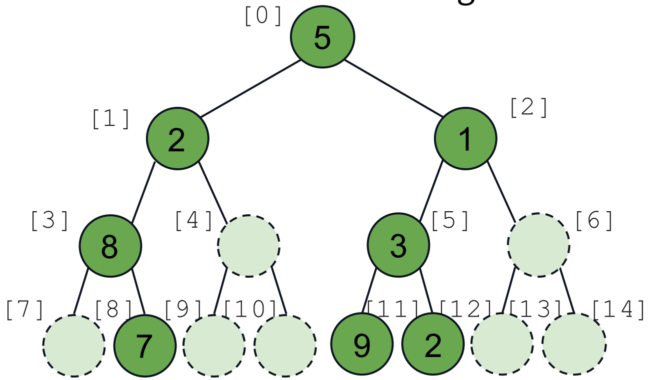
Height 1 Height 2 Height 3 Height 4

This implementation is efficient if the tree is (almost) full or complete.

upstage Education

Array-based Implementation

- One more thing to consider: How can we figure out **direct children** of a node?
  - With a tree, we usually access data from the root to the leaf.
  - Common rule connecting a node to its children?



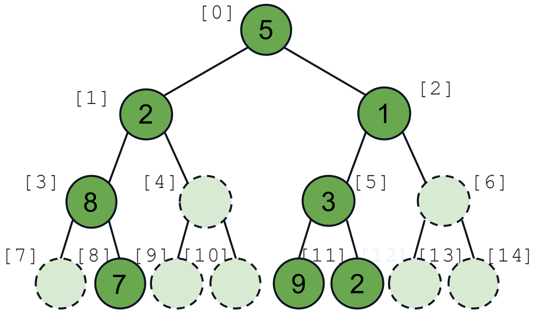
- From parent, its left child:
  `[parent index] * 2 + 1` 😀
- Right child:
  `[parent index] * 2 + 2` 😁

- Similarly, from a child, its parent:
  `[child index - 1] // 2`

upstage Education

Array-based Implementation

- Now, think about more general case: the tree may not be close to complete.
  - If we don't have a node, we may set `None` (or some other indicator for emptiness).



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
5	2	1	8	None	3	None	None	7	…

- This implementation becomes **less efficient** if the tree gets **far from full or complete**.

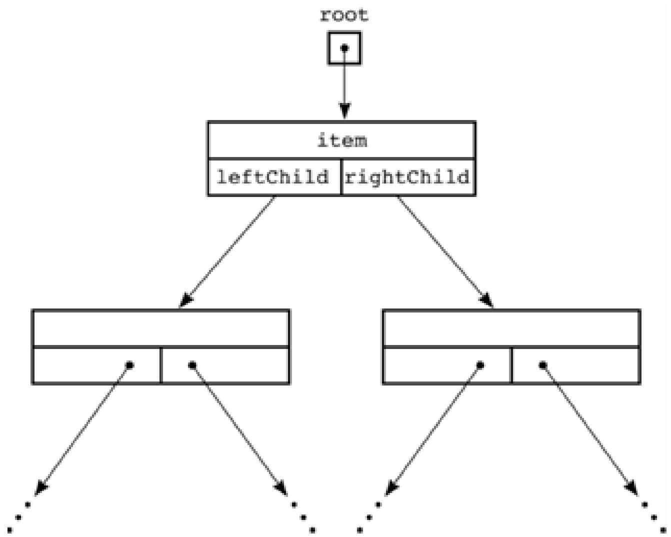
upstage Education

Reference-based Implementation

- It is similar to the linked list; instead of having a single link to another node, Binary Tree Node has **two references** to other nodes!

```
class TreeNode():
  def __init__(self, x):
    self.item = x
    self.left = None
    self.right = None
```

```
class BinaryTree():
  def __init__(self, node):
    self.root = node
```

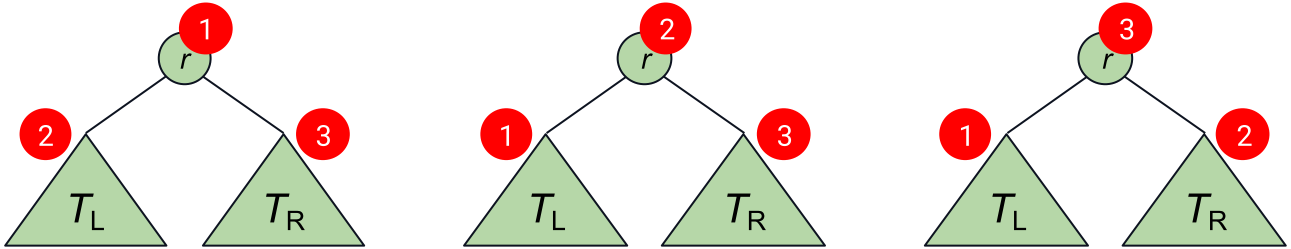
[1] 

[1] http://ocw.utm.my/file.php/31/Module/ocwSearchTreeDec2011.pdf

upstage Education

Tree Traversal

- Traversal: **visiting all nodes once**
  - It is not straightforward to visit all nodes in a given tree, unlike an array or a linked list.
- There can be many different ways of traversal. We cover a few widely-used ones:



Preorder	**Inorder**	**Postorder**
Root - Left - Right	Left - Root - Right	Left - Right - Root

- Inside each subtree, the same rule applies recursively.

upstage Education

Tree Traversal Example

- Preorder traversal?

60 {20 subtree} 70

60 20 10 {40 subtree} 70

60 20 10 40 30 50 70

- Inorder traversal?

{20 subtree} 60 70

10 20 {40 subtree} 60 70

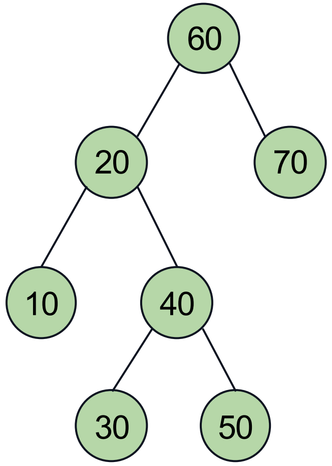
10 20 30 40 50 60 70

- Postorder traversal?

{20 subtree} 70 60

10 {40 subtree} 20 70 60

10 30 50 40 20 70 60



upstage Education

Tree Traversal Implementation

- Use recursion!
  - Base case: empty tree. Do nothing!
  - General case: 2 recursive calls + visiting the root.
  - The order depends on {pre, in, post}-order.

```
def preorder(node):
  if node is not None:
    print(node.item)
    preorder(node.left)
    preorder(node.right)

preorder(root)
```

```
def inorder(node):
  if node is not None:
    inorder(node.left)
    print(node.item)
    inorder(node.right)

inorder(root)
```

upstage Education

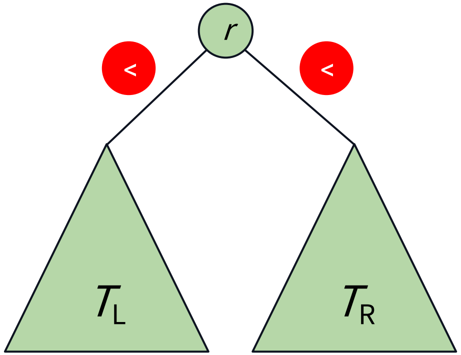
02

Binary Search Tree

Definition of Binary Search Tree

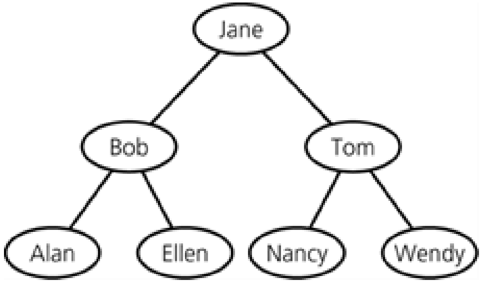
- Each node has a **search key**.
  - There are **no duplicates** among the search keys in a binary search tree.

- For each node *n*, it satisfies:
  - *n*'s key is greater than all keys in its left subtree T_L.
  - *n*'s key is less than all keys in its right subtree T_R.
  - Both T_L and T_R are binary search trees.

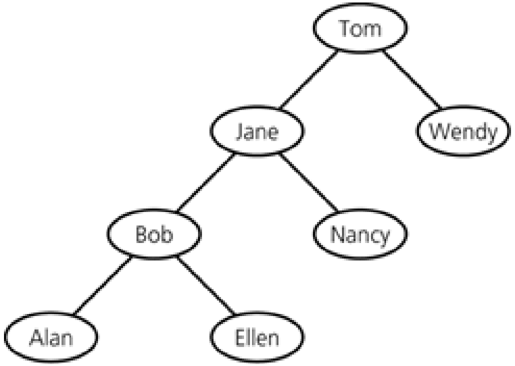


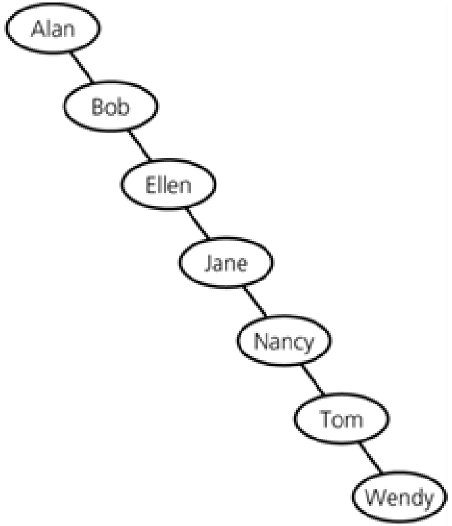
upstage Education

Binary Search Tree Examples



[1]





Check! All these 3 BSTs contain the same data.

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week10.pdf

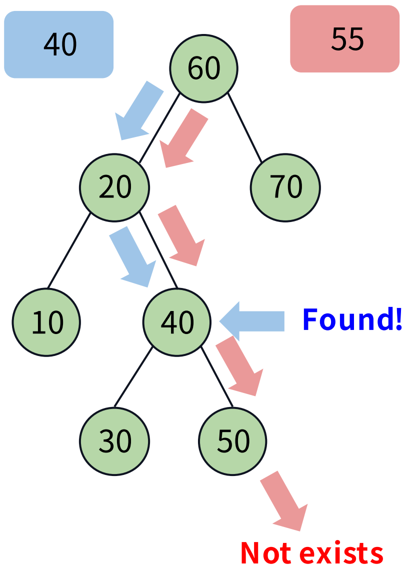
Binary Search Tree Operations

- **Insert** a new item into a binary search tree.
- **Retrieve** (**search**) the item with a given search key from a binary search tree.
- **Delete** the item with a given search key from a binary search tree.
- For all, according to the rule that {left subtree} < root < {right subtree} at all levels!

upstage Education

Search (Retrieval)

- Task: Search if there is a node with the given key, and output the item if so.
  - If the given key is not in the tree, we should be able to figure it out as well.
- Main Idea: At each node, decide which subtree to search further. Only 3 cases:
  - [Case 1] If the **search key = item** at the node, we **found** it!
  - [Case 2] If the **search key < item** at the node, the target must be in its **left subtree** if exists.
  - [Case 3] If the **search key > item** at the node, the target must be in its **right subtree** if exists.
  - For Case 2 & 3, move to the corresponding subtree, then repeat the same testing.
  - Repeat until you reach at a leaf. If you still do not meet Case 1, we conclude the key is **not in the tree**.



upstage Education

Search (Retrieval): Implementation

- Implementation based on recursive calls:

```
def search(root, key):
  if root is None:
    return None   # Not found
  elif key == root.item:
    return root   # Found
  elif key < root.item:
    return search(root.left, key)
  else:  # key > root.item
    return search(root.right, key)
```

Base case (empty BST): Not found!

General case: case 1, 2, 3

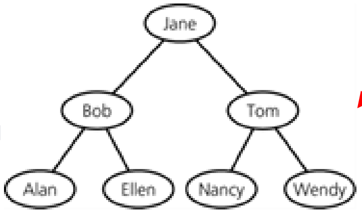
upstage Education

Search (Retrieval): Time Complexity

- <u>Time complexity</u> of search operation?
  - At each node, we compare two values once, and decide where to go. → O(1)
  - How many times? Length from the root to the leaf!

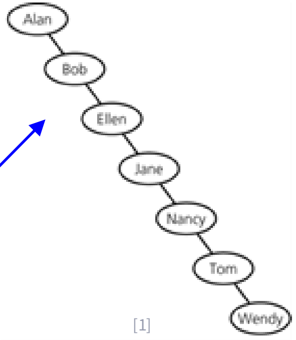
- Thus, the worst time performance = tree height!
- In terms of N (the number of data points in the tree), what's the asymptotic (Big O) complexity of tree search?

[1]



If the tree is **balanced** (close to full), the height gets closer to **O(log N)**.

If the tree is unbalanced (close to linked list), the height gets closer to **O(N)**.



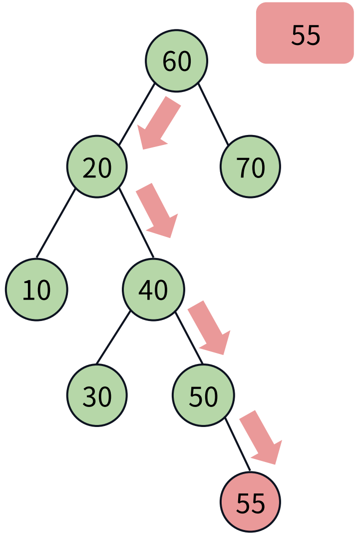
[1]

Then, how the shape of a tree is determined? We will revisit this soon.

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week10.pdf

Insertion

- Task: Insert a new key into the BST, preserving BST conditions.
  - We need to find the right location to put the new node.
- Main Idea: Insertion is basically a **failed search**. When we conclude the item does not exist, insert the new node right there!
  - [Case 1] If the **search key = item** at the node, we **found** it!
  - [Case 2] If the **search key < item** at the node, the target must be in its **left subtree** if exists.
  - [Case 3] If the **search key > item** at the node, the target must be in its **right subtree** if exists.
  - For Case 2 & 3, move to the corresponding subtree, then do the same testing.
  - Repeat until you reach at a leaf. Insert the new node there!



upstage Education

Insertion: Implementation

- Implementation based on recursive calls:

```
def insert(root, item):
  if root is None:
    new_node = TreeNode(item)
    return new_node
  elif key == root.item:
    # ERROR: already exists
  elif key < root.item:
    new_subtree = insert(root.left, item)
    root.left = new_subtree
    return root
  else:  # key > root.item
    new_subtree = insert(root.right, item)
    root.right = new_subtree
    return root
```

Base case: Create new node

General case: 3 ways

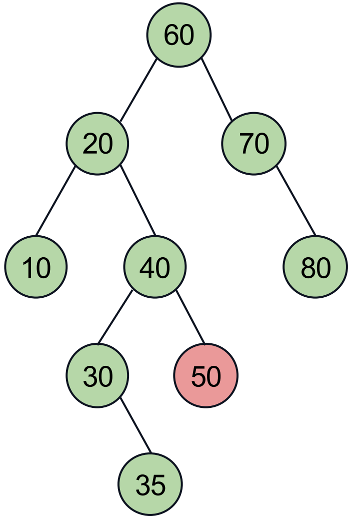
Deletion

- Task: Delete a node with the given key, preserving BST conditions after deletion.
    - After deletion of an intermediate node, the tree will be broken 😨.
- Main Idea:
    - [Case 1] If the node to delete is a **leaf**, simply remove it.
    - [Case 2] If the node to delete has a **single child**, the subtree will take it over.
    - [Case 3] If the node to delete has **both children**, elect the leftmost item in the right subtree as the new root.

upstage Education

Deletion (Case 1)

- When the target node is a **leaf**, we can simply remove it.
- After deletion, the resulting tree
  - is still **connected**, and
  - still **satisfies the BST conditions**.

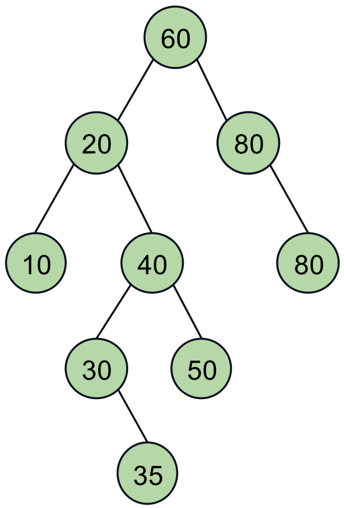


upstage Education

Deletion (Case 2)

- When the target node **has one child**, the existing child takes the position of the target node, taking its descendents (subtree).
- After the right subtree takes over, the resulting tree
  - is no longer broken,
  - still satisfies the BST conditions.
- After deletion, the resulting tree
  - is broken,
  - still satisfies the BST conditions.

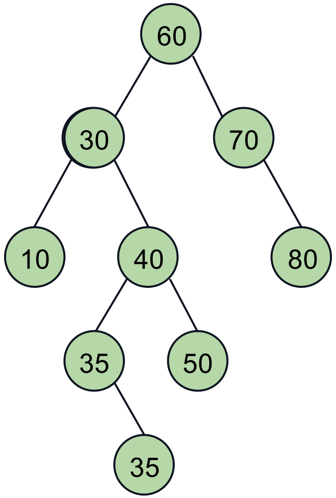
This slide is best seen with animations.



This slide is best seen with animations.

Deletion (Case 3)

- When the target node **has both children**, we elect the target's immediate successor (=left-most node in the right subtree) as the new root.
- After deletion, the resulting tree is broken.
  - Cannot simply take one subtree as new root, since we have two.
- To make the resulting tree satisfy BST conditions, we elect the immediate successor of the deleted node. The resulting tree, however, is still broken!
  - Why immediate successor? With it, it's guaranteed to satisfy the BST conditions!
- The deleted immediate successor node may need to adopt its orphan right child (if any).
  - Here, it is guaranteed that no left child exists.



Similarly, we may nominate the immediate predecessor (=right-most item in the left subtree.)

Deletion: Implementation

```
def delete(root, key):
  if root is None:
    return root

  if key < root.key:
    root.left = delete(root.left, key)
  elif key > root.key:
    root.right = delete(root.right, key)
  else:

  ... (to be continued)
```

Locating the target

`delete` function returns the subtree to replace the target node to be deleted.

upstage Education

Deletion: Implementation

Case 1: no child if `root.right` is also `None`

```
... (continuing)

else:
  if root.left is None:
    new_root = root.right
    root = None
    return new_root
  elif root.right is None:
    new_root = root.left
    root = None
    return new_root
  else:
    im_su = get_immediate_successor(root)
    root.key = im_su.key
    root.right = delete(root.right, im_su.key)

  return root
```

Case 2: single child

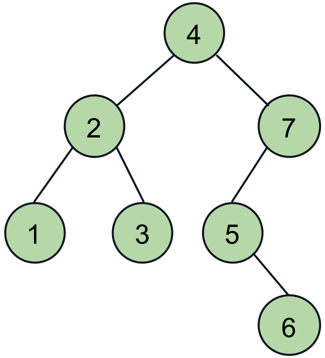
Case 3: both children

```
def get_immediate_successor(target):
    curr = target.right
    while curr.left is not None:
      curr = curr.left
    return curr
```

After this line, we now delete the node `im_su`.

Revisiting the shape of BST

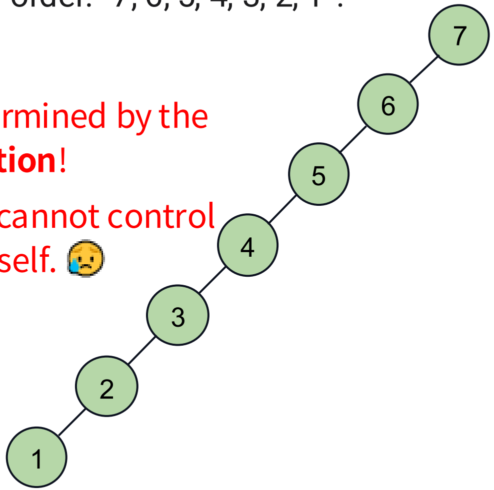
- For a given data, how is the shape of BST determined?
  - Let's try to insert "4, 7, 2, 3, 5, 1, 6":



Shape of the tree is determined by the **order of insertion**!

Unfortunately, we often cannot control the datastream itself. 😥

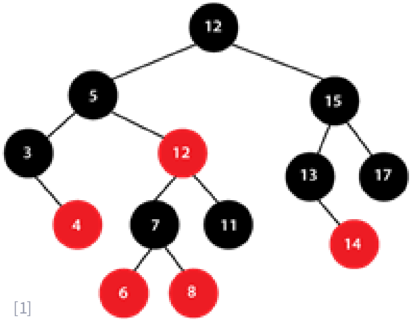
  - Suppose the same data is given in a different order: "7, 6, 5, 4, 3, 2, 1":



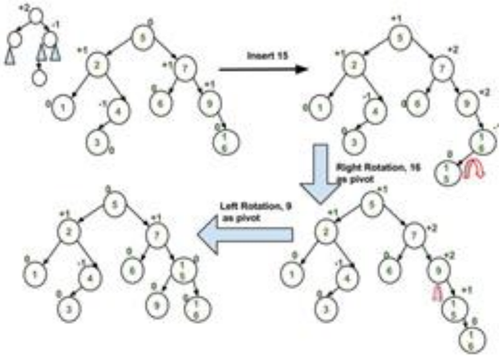
upstage Education

Balanced Trees

- We desire to make the tree **more balanced** for faster operations.
- There are some special trees that **guarantee balance** up to some level, but details of them are beyond of this course.



[1]



[2]

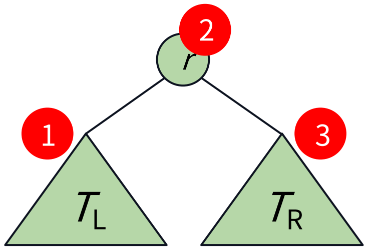
Red-black tree
: guarantees that the height is lower than 2 log(N+1).

AVL tree
: guarantees that heights of the two child subtrees of any node differ by at most 1.

[1] https://kostja.github.io/misc/2017/02/23/tarantool-data-structures.html
[2] https://blog.naver.com/luexr/223479906039

Tree Sort

- **Inorder traversal** on a binary search tree lists the data in sorted order.
  - Due to the BST conditions, all values in the left subtree < root value < all values in the right subtree, at all nodes.
  - Inorder traversal visits nodes in the order of left - root - right!
- Time complexity?
  - We first need to insert all data into a BST.
    → O(log N) per each element × N of them = O(N log N)
    (This assumes use one of the balanced trees!)
  - Then, inorder traversal: O(N)
    ← we visit one node at a time.
  - Overall, **O(N log N)** if the tree is balanced.
    Otherwise, worst performance will be **O(N^2)**.



Inorder

Left - Root - Right

upstage Education

Time Complexity

- Time complexity of Binary Search Tree operations?

Task	Average-case	Worst-case
Insertion	O(log N)	O(N)
Retrieval	O(log N)	O(N)
Deletion	O(log N)	O(N)
Traversal	O(N)	O(N)

When?

= When the tree is significantly unbalanced.

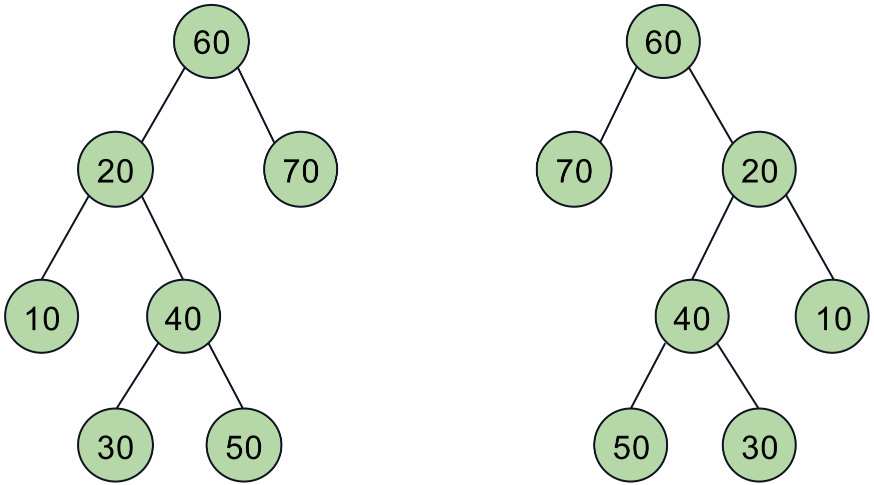
03

Applications of Binary Search Trees

upstage Education

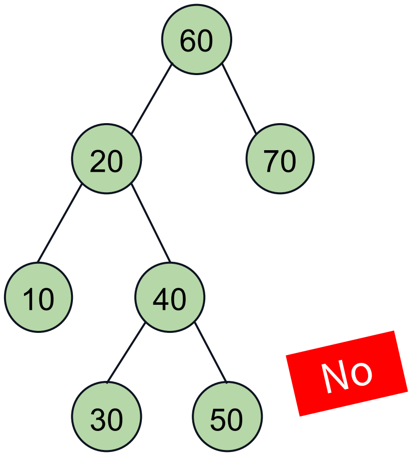
Problem 1

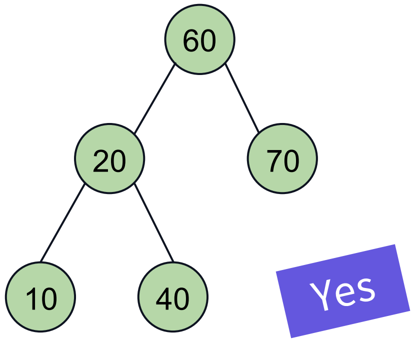
- Test if two binary trees are symmetric.



Problem 2

- Test if a binary tree is balanced.
  - A binary tree is balanced if the difference in the height of its left and right subtrees is at most 1 for all nodes in the tree.





upstage Education

Problem 3

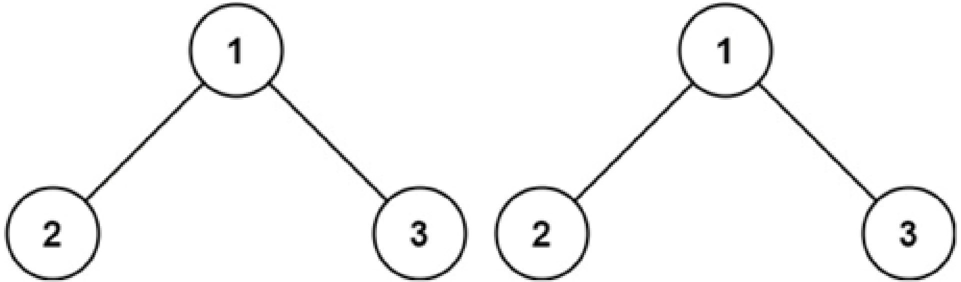
100. Same Tree

Easy Topics Companies

Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

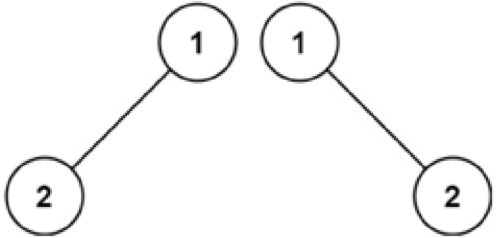
Example 1:




```
Input: p = [1,2,3], q = [1,2,3]
Output: true
```

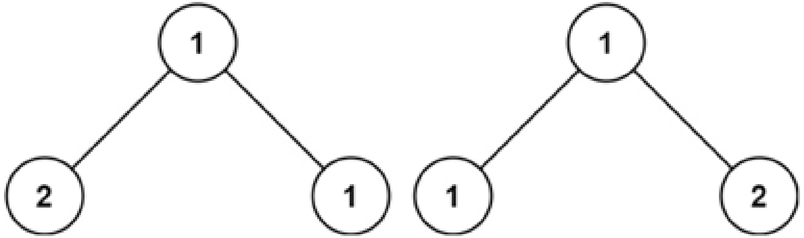
Problem 3

Example 2:




```
Input: p = [1,2], q = [1,null,2]
Output: false
```

Example 3:



Building intelligence for the future of work