CS2040

Tutorial 7: BST and bBST

Nicholas **Russell** Saerang (russellsaerang@u.nus.edu)

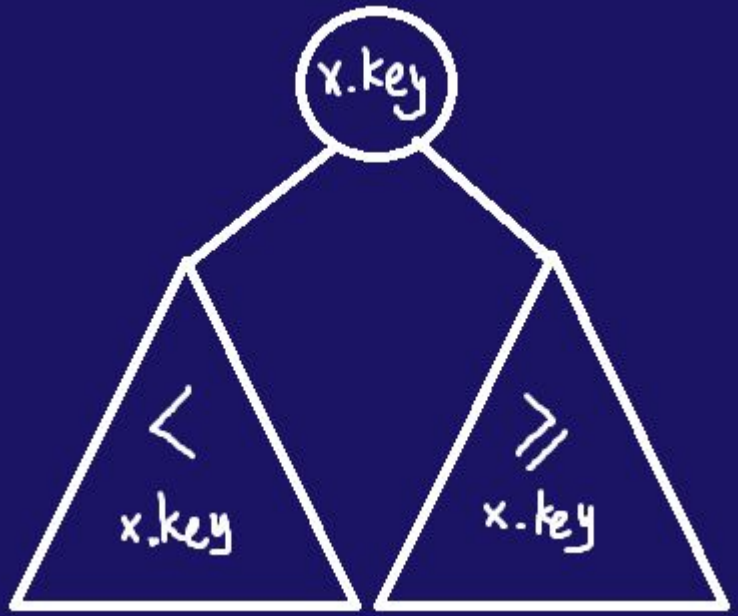
:D

BST AND BBST

Binary Search Tree (BST)

Property:
For every vertex x and y
- y.key < x.key if y is in left subtree of x
- y.key ≥ x.key if y is in right subtree of x

search, findMin, findMax, insert, successor, predecessor, delete runs in O(h), where h is the height of the tree - worst case O(N) → see lecture slides :)

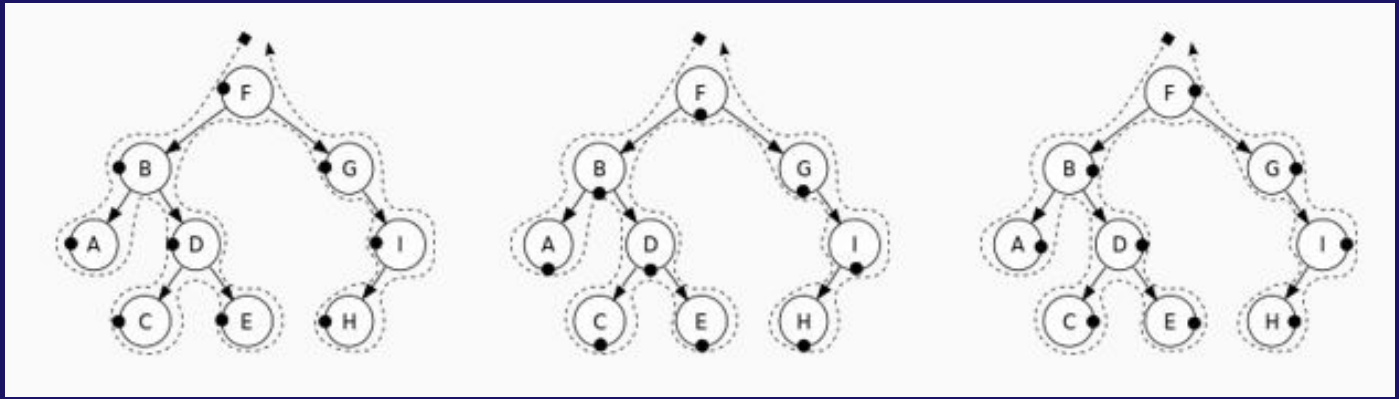


Standard Traversals

```
Pre-Order(T)
    if T is null: return
    Visit(T)
    Pre-Order(T.left)
    Pre-Order(T.right)
```

```
In-Order(T)
    if T is null: return
    In-Order(T.left)
    Visit(T)
    In-Order(T.right)
```

```
Post-Order(T)
    if T is null: return
    Post-Order(T.left)
    Post-Order(T.right)
    Visit(T)
```

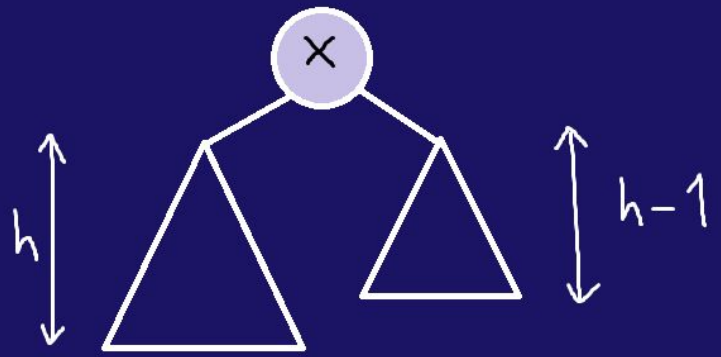


Balanced Binary Search Tree (bBST)

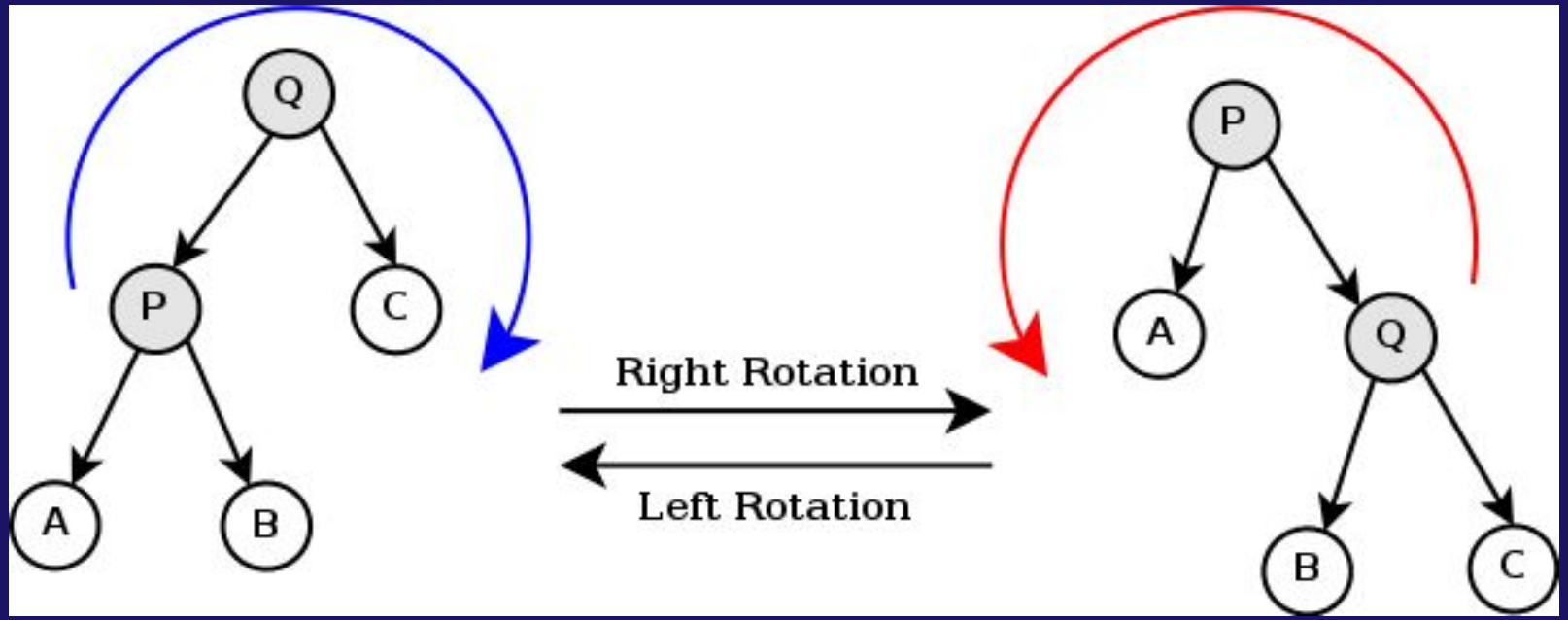
On BBST, all operations run in $O(\log n)$ time.

AVL is one implementation of BBST. (other example of BBST: red black tree)

In AVL tree, the height of the left subtree and the height of the right subtree differ by **at most 1**, i.e. $|x.left.height - x.right.height| \leq 1$



AVL Tree Rotation



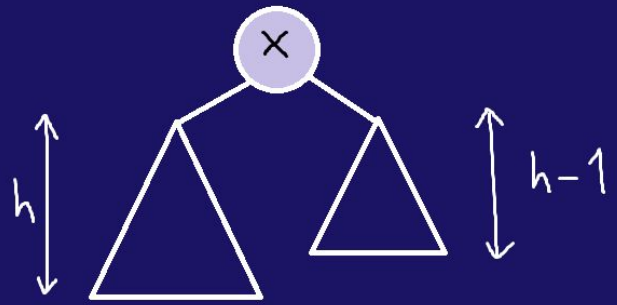
AVL Tree Rebalancing

bf = balance factor
bf(v) = height(v.left) − height(v.right)

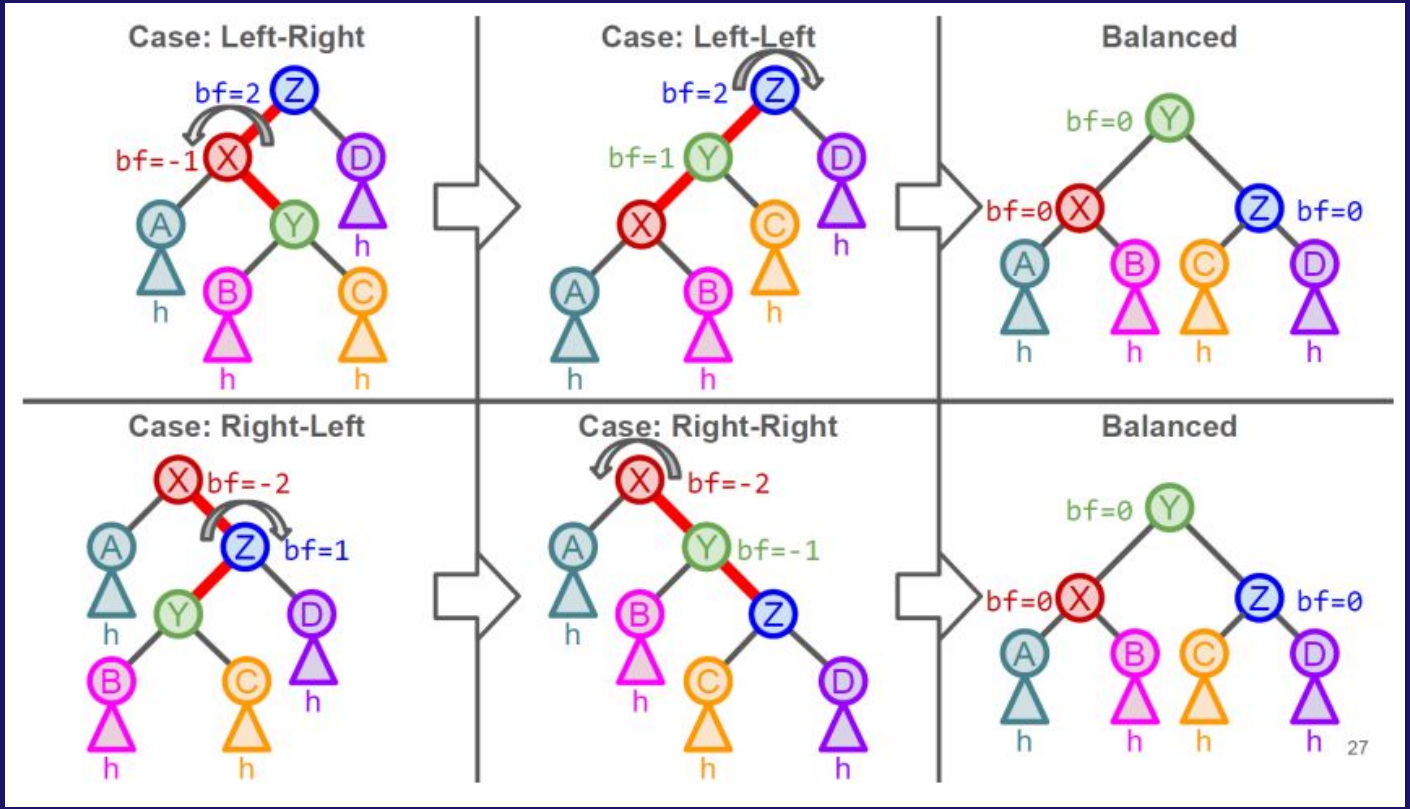
A tree is balanced if for every node v, bf(v) ∈ {−1, 0, 1}.
→ AVL tree!

Given a tree rooted at v,

- if bf(v) > 1, the left subtree is *taller* than the right one.
- if bf(v) < −1, the left subtree is *shorter* than the right one.



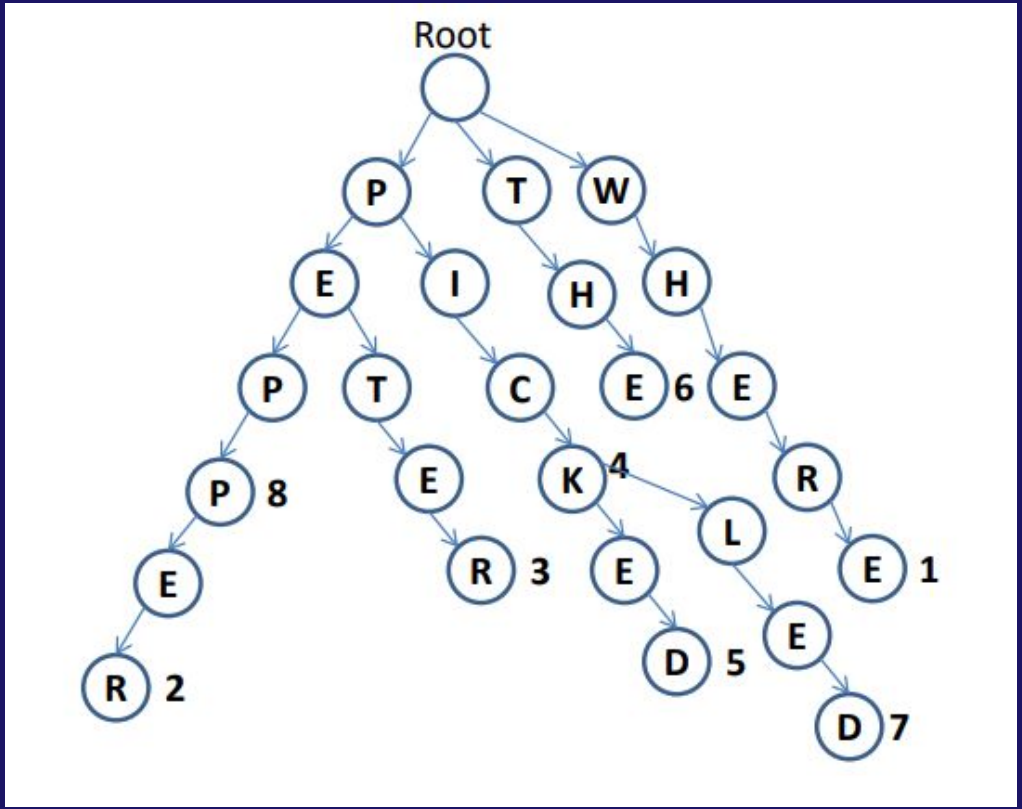
AVL Tree Rebalancing



TRIE

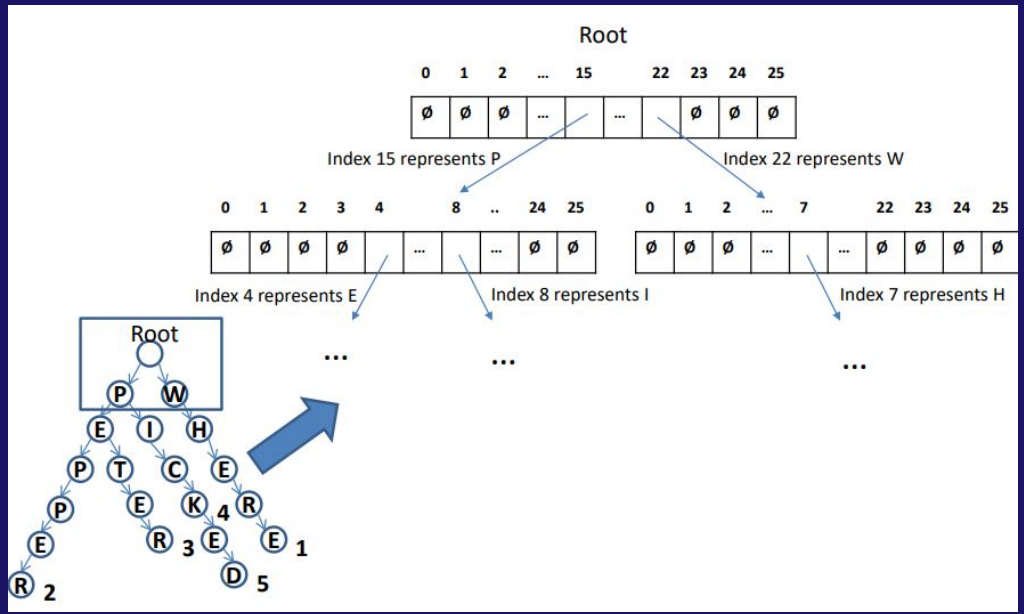
this is only for CS2040S, please skip otherwise :)

Trie Anatomy



Representing Trie

- Same as BST/AVL but with R children per node instead of at most 2 (usually R = 26)
- Store attributes in a node such as count and value



Trie Time Complexities

- Insertion: worst case O(L), where L is the length of the key
- Successful retrieval: worst case O(L)
- Deletion: worst case O(L)
- Unsuccessful retrieval: average O(log N), N = number of keys in trie
- Sorting keys: worst case O(NL'), where L' is the length of the longest key

Trie Space Complexities

Average O(wNR)

- w = average key length
- N = number of keys in trie
- R = alphabet size, usually 26

Compare with:

- AVL: O(wN)
- Hash tables: O(wN)

Space-time trade-off, sacrifice space for more optimal operations!

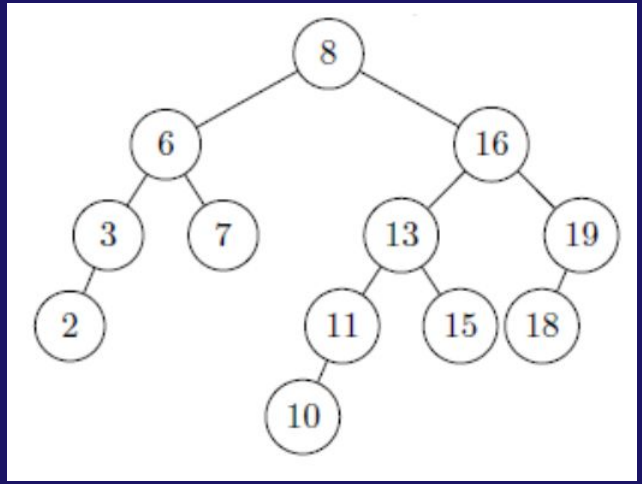
01

TRUE OR FALSE?

Question 1a

(AY19/20 Sem 1 Final Exam)
Given any AVL tree of height 4, deleting any vertex in the tree will not result in more than 1 rebalancing operation.
(Not rotation but rebalancing operations!)

False. For the binary search tree beside, deleting vertex 7 triggers two rebalancing operations.



Important: Note that a rebalancing operation is not the same as a rotation: a rebalancing operation is one of the four rebalancing cases mentioned in the lecture notes (LL, LR, RL, RR), and can consist of more than one rotation.

Question 1b

The minimum number of vertices in an AVL tree of height 5 is 21.

False. The minimum is 20.

The minimum number of vertices n_h in an AVL tree of height h is given by the recurrence

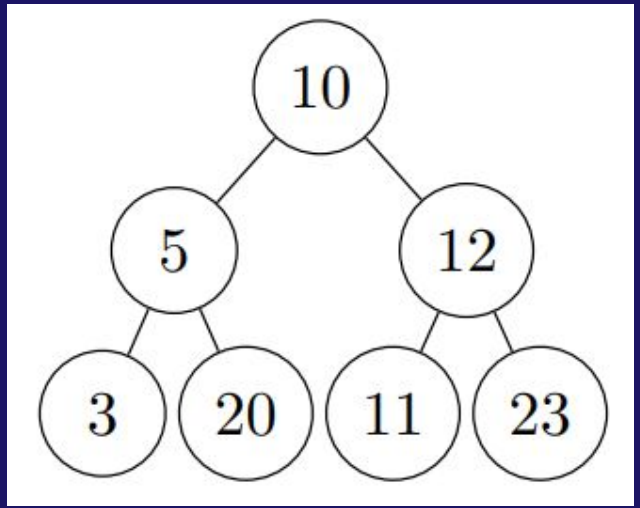
$n_0 = 1$, $n_1 = 2$, $n_i = n_{i-1} + n_{i-2} + 1$ → how to derive?

Using the recurrence, we have $n_0 = 1$, $n_1 = 2$, $n_2 = 4$, $n_3 = 7$, $n_4 = 12$, $n_5 = 20$.

Question 1c

In a tree, if for every vertex x that is not a leaf, x.left.key < x.key if x has a left child and x.key < x.right.key if x has a right child, the tree is a BST.

False. Consider the following binary tree. Note that the above statement holds for every node in the binary tree, but the binary tree is not a BST.



For a binary tree to be a BST, you need all vertices y in the left subtree of x to have y.key < x.key and not just the left child y of x having y.key < x.key. Similar argument for vertices in the right subtree of x and the right child of x.

02

IN-ORDER TRAVERSAL

Question 2a

What is the running time of Algorithm 1?

Algorithm 1 In-Order Traversal

```
1: procedure InOrderTraversal(T)                    ▷ bBST T is supplied as input
2:     currentNodeValue ← T.findMin()
3:     while currentNodeValue ≠ −1 do
4:         output currentNodeValue
5:         currentNodeValue ← T.successor(currentNodeValue)
6:     end while                                    ▷ successor returns -1 if there is no successor
7: end procedure
```

Answer: O(n log n)

T.findMin() and T.successor(node) runs in O(log n).

Question 2b

Propose modifications to the successor function such that Algorithm 1 runs in O(n) time.

Notice that in a standard in-order traversal, we perform something that is similar to Algorithm 1, we go one node to its successor, then from the next node, we go to its successor again, and so on, until we visit all nodes in the BST.

However, the difference is that in a standard in-order traversal, we move directly from one node to its successor, instead of locating its successor starting from the root node. Hence, we can modify Algorithm 1 such that it works in a way similar to the standard in-order traversal.

In particular, for the successor function, we can get it to accept a reference to the node that we want to find the successor of, so that we can begin searching for the successor from the node itself, rather than from the root node.

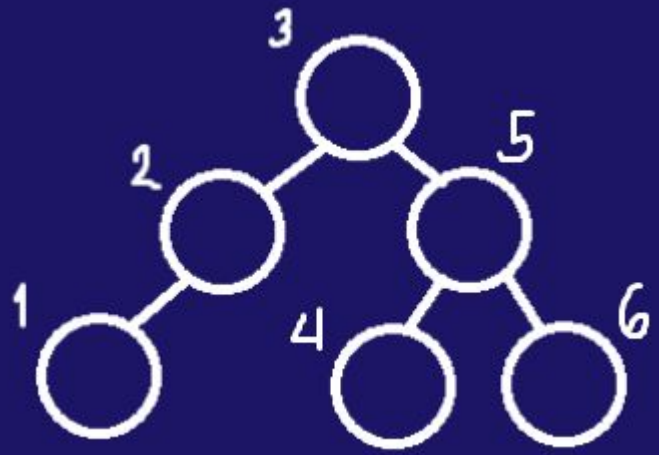
03

RANK AND SELECT

Question 3a: Rank

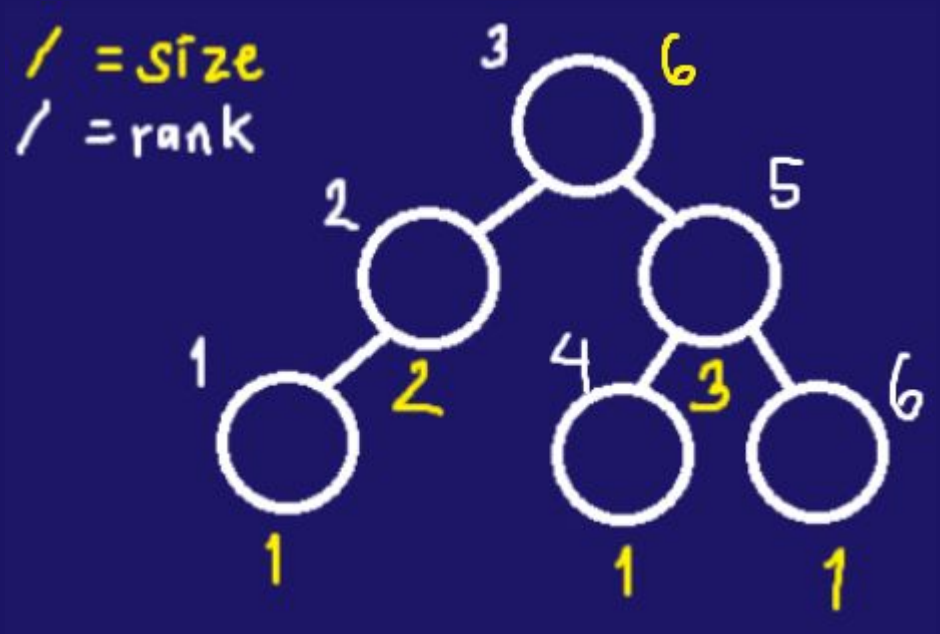
A node x has rank k in a BST if there are k - 1 nodes that are smaller than x in the BST. The rank operation finds the rank of a node in a BST.

Describe an algorithm that finds the rank of a given node in the BST in O(h) time, where h is the height of the BST.



Question 3a: Rank

Augment every node in the BST with the size of its subtree. Then, we can compute the rank of a given node using the following algorithm (next slide).



Question 3a: Rank

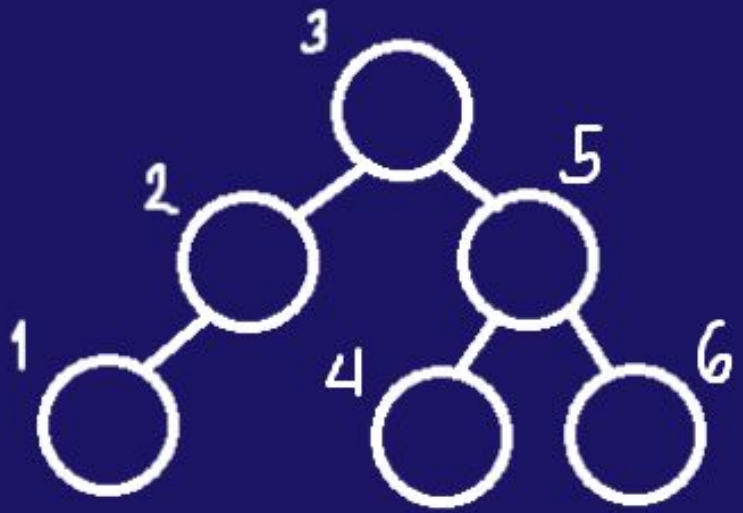
Algorithm 2 BST Rank

```
1: function RANK(node, v)
2:     if node.key = v then
3:         return node.left.size + 1
4:     else if node.key > v then
5:         return RANK(node.left, v)
6:     else
7:         return node.left.size + 1 + RANK(node.right, v)
8:     end if
9: end function
```

Question 3b: Select

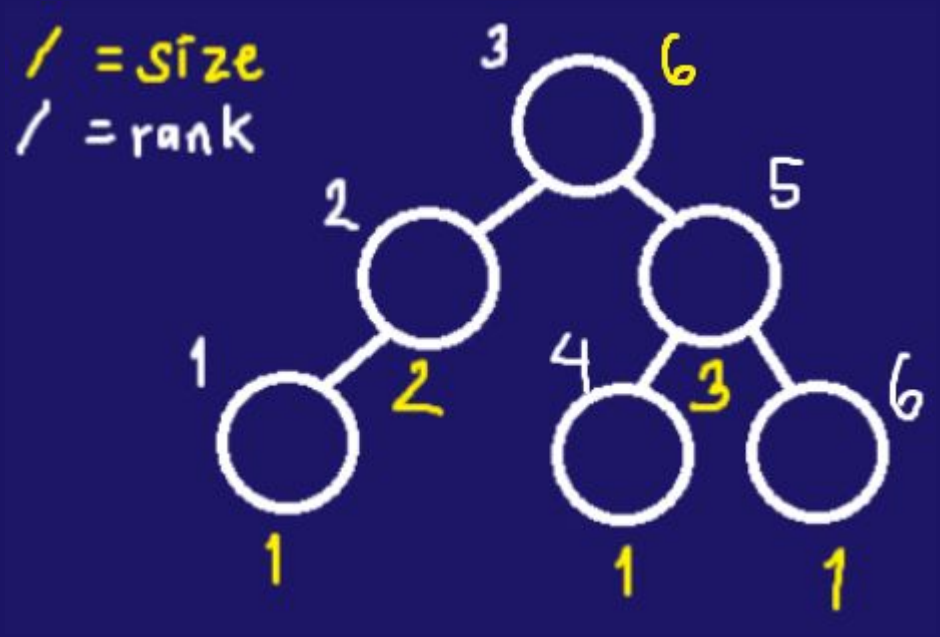
The select operation returns the node with rank k in the BST.

Describe an algorithm that finds the node with rank k in the BST in O(h) time, where h is the height of the BST.



Question 3b: Select

Augment every node in the BST with the size of its subtree. Then, we can find the node with rank k using the following algorithm (next slide).



Question 3b: Select

Algorithm 3 BST Select

```
1:  function SELECT(node, k)
2:      q ← node.left.size
3:      if q + 1 = k then
4:          return node.key
5:      else if q + 1 > k then
6:          return SELECT(node.left, k)
7:      else
8:          return SELECT(node.right, k − q − 1)
9:      end if
10: end function
```

04

TRANSMISSION SYSTEM

Transmission System

Abridged problem description:

There are n servers numbered 1 to n. Each server can be enabled/disabled.

- Server i can send directly to server i + 1, $1 \leq i < n$.
- Server i can send indirectly to j if $i < j$, server i, j, and all servers between them are enabled.

You need to support q of the following three types of operations:

- Enable(i): Enable the i-th server. If the i-th server is already enabled, nothing happens.
- Disable(i): Disable the i-th server. If the i-th server is already disabled, nothing happens.
- Send(i, j): Return true if it is possible to send data from server i to j, false otherwise.

Describe the most efficient algorithm you can think of for each of the three types of operations. What is the running time of your algorithm for each of the three types of operations?

Transmission System

Answer:

1. We can use a bBST to maintain the servers that are currently disabled. When a server is disabled, we insert it into the bBST, and when a server is enabled, we remove it from the bBST. O(log n)
2. To check if we are able to send a message from server i to server j, we need to check that i is not in the bBST, and either i has no successor in the bBST, or the successor of i in the bBST is greater than j. O(log n)

Transmission System

Algorithm 4 Solution to Problem 4

```
1: T ← bBST, initially empty
2: procedure ENABLE(i)
3:     T.delete(i)
4: end procedure
5: procedure DISABLE(i)
6:     T.insert(i)
7: end procedure
8: function SEND(i, j)
9:     if i is in T or T.successor(i) ≤ j then
10:        return false
11:    else
12:        return true
13:    end if
14: end function
```

Transmission System

But i is not in the bBST, so what is T.successor(i)?

Here, successor(x) refers to the node in the bBST containing the smallest key that is greater or equal to x. This definition of successor is similar to the ceilingEntry method in the Java TreeMap, ceiling method in the Java TreeSet (or lower_bound in C++).

A simple way to implement this is to first do a search to check for existence, then do an insertion and call the usual successor, and then doing a deletion. However, there is way to do this directly without any insertions or deletions by modifying the search operation, and is left as an exercise.

Transmission System

What if I want to keep track of the enabled server instead?

This leads us to an alternative solution!

1. Enable: add into bBST O(log n)
2. Disable: delete from bBST O(log n)
3. Send(i, j): check if i and j are enabled (i,j in bBST) and rank(j) - rank(i) = j - i (all intermediate servers from i + 1 to j are enabled) O(log n)

05

LOWEST COMMON ANCESTOR

Lowest Common Ancestor

(CS2040S AY19/20 Sem 1, Quiz 2)

The Lowest Common Ancestor (LCA) of two nodes a and b in a BST is the node furthest from the root that is an ancestor of both a and b.

Consider a Balanced Binary Search Tree (bBST) T containing n nodes with unique keys, where nodes do not have parent pointers (i.e. given a node, you cannot find its parent in O(1) time).

Describe the most efficient algorithm you can think of to find the LCA of two given nodes in the bBST. What is the running time of your algorithm?

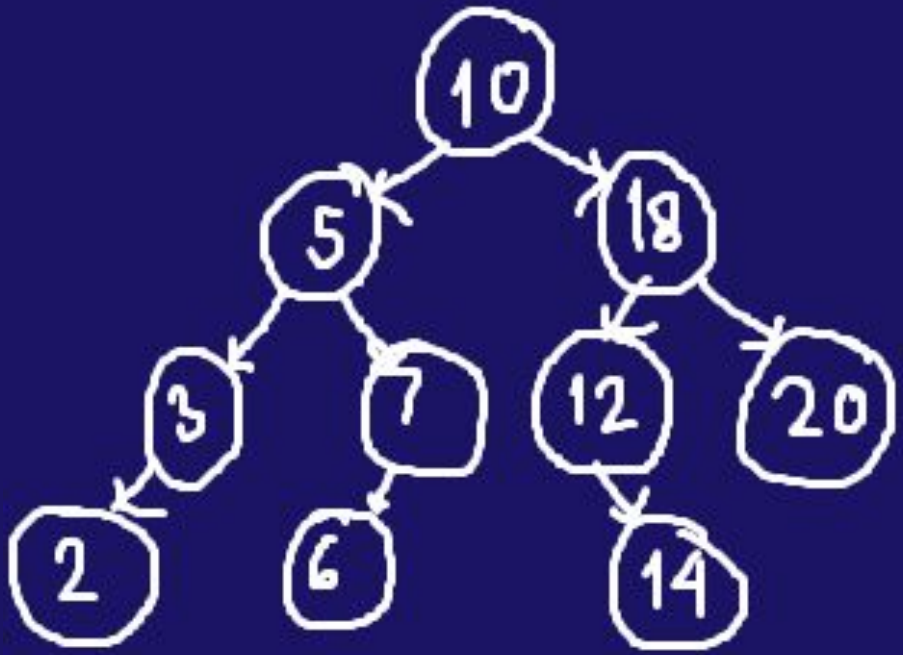
Lowest Common Ancestor

Here, we can exploit the property of a Binary Search Tree (BST). At any node, there are essentially four cases to consider:

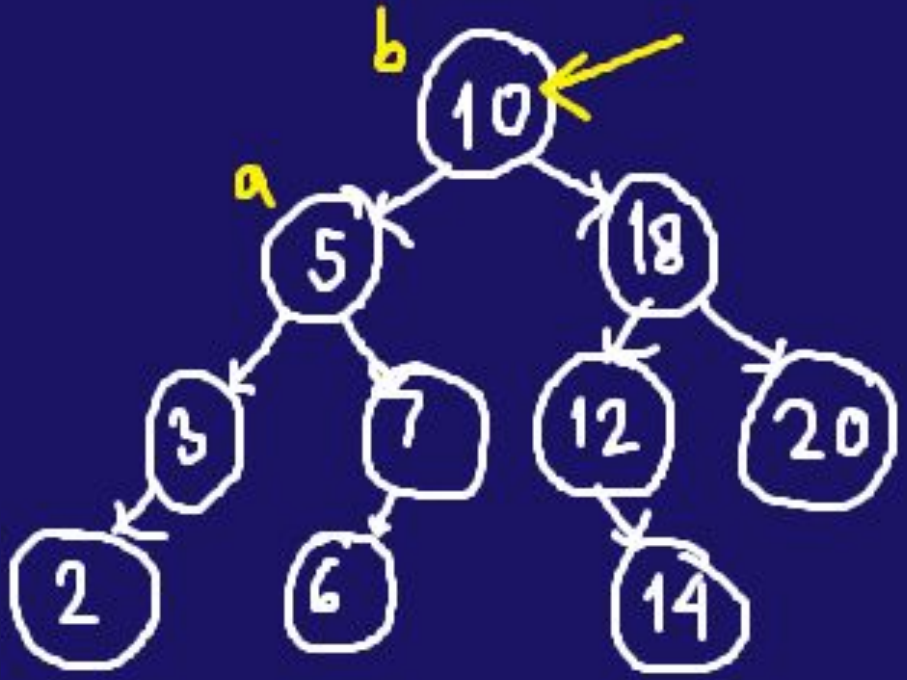
1. If either a or b is the same value as the current node, we can stop and we have found our LCA.
2. If a and b are both smaller than the value at the current node, we move to the left child of the current node and it becomes the new current node.
3. If a and b are both greater than the value at the current node, we move to the right child of the current node and it becomes the new current node.
4. If a is smaller than the value at the current node, and b is greater than the value at the current node, then the current node is the LCA. This is also true if we swap the positions of a and b.

The algorithm takes O(log n) time and O(1) space, since the BST is balanced.

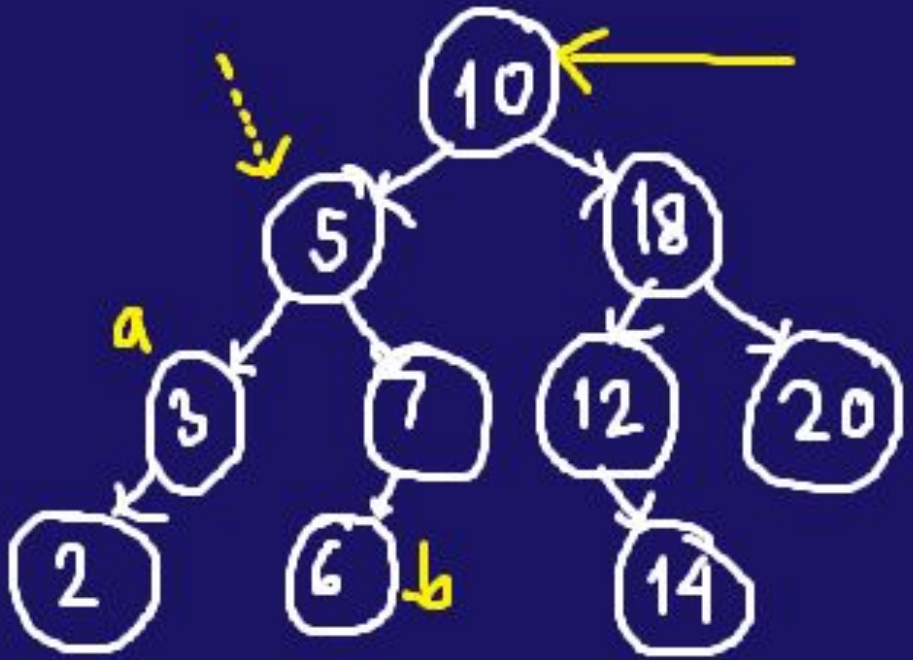
Lowest Common Ancestor



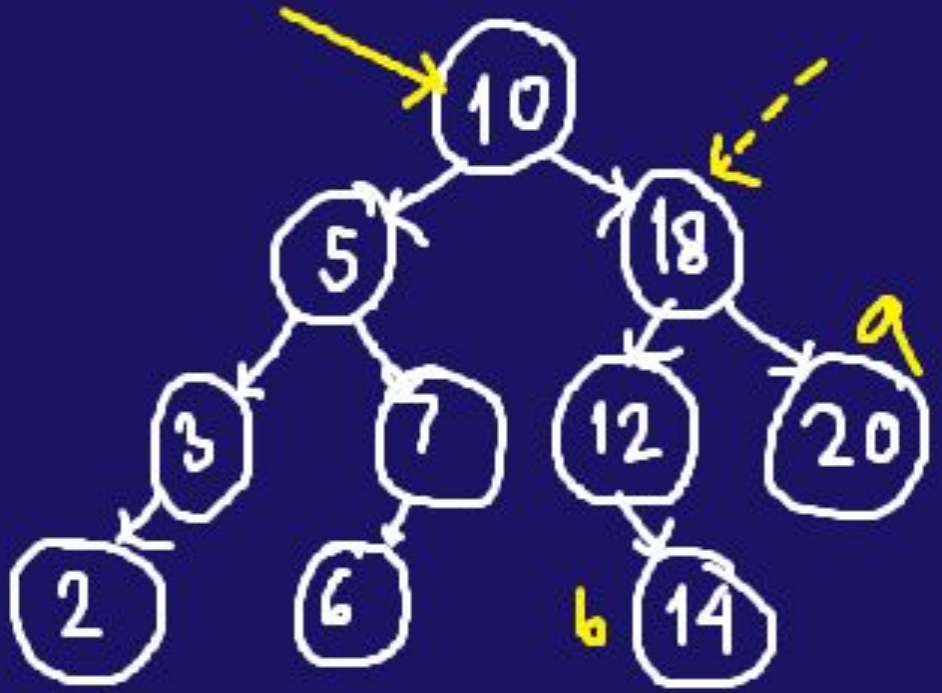
LCA Case 1



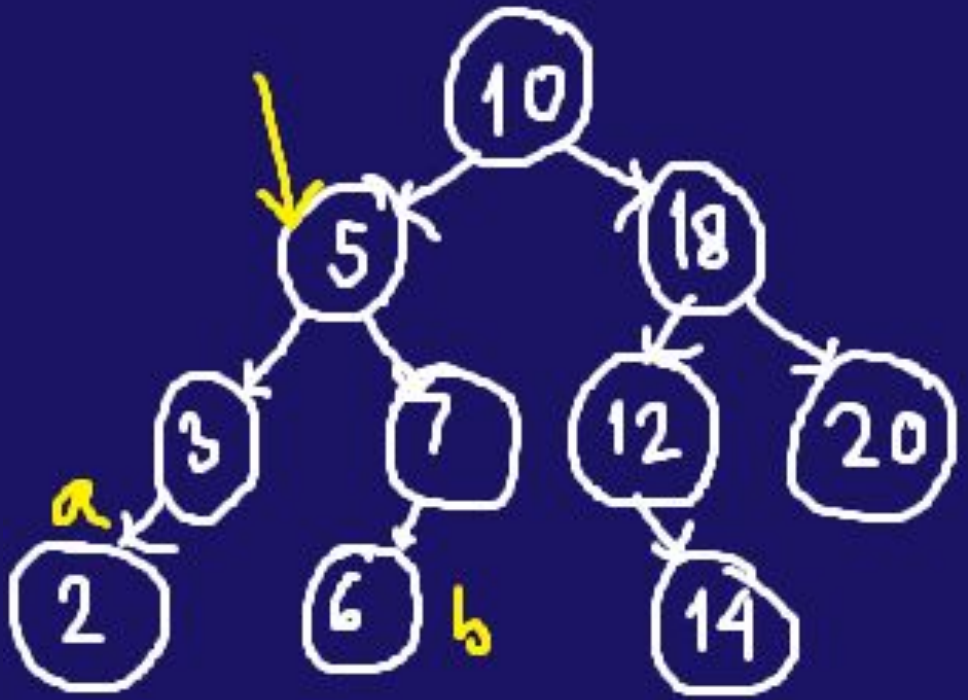
LCA Case 2



LCA Case 3



LCA Case 4



Extra: Standard Traversals

Some extra questions:

Given only the in-order traversal (or a sorted array), can you construct the AVL?

Extra: Standard Traversals

Some extra questions:

Given only the in-order traversal (or a sorted array), can you construct the AVL?

> Yes. Take the median as the root of the AVL, then recurse the construction on both halves of the in-order traversal. This should take O(n) time where n is the number of elements in the traversal.
>
> Note that the constructed AVL based on the in-order traversal is **not unique**, meaning there can be different AVLs that lead to the same in-order traversal.

Extra: Standard Traversals

Some extra questions:

Implement a procedure split(x) that splits an AVL tree into two AVL trees such that the first AVL tree contains all elements <= x in the original AVL tree, and the second AVL tree contains all elements > x in the original AVL tree. This should take O(n) time, where n is the number of elements in the original AVL tree.

Extra: Standard Traversals

Some extra questions:

Implement a procedure split(x) that splits an AVL tree into two AVL trees such that the first AVL tree contains all elements <= x in the original AVL tree, and the second AVL tree contains all elements > x in the original AVL tree. This should take O(n) time, where n is the number of elements in the original AVL tree.

> Do an in-order traversal and split those traversals into <= x and > x. Each partition should still be a valid in-order traversal, to which we can apply the previous question for each partition.

Extra: Standard Traversals

Some extra questions:

Implement a procedure `merge(t1, t2)` that merges two AVL trees t1 and t2 into a single AVL tree. This should take `O(m+n) time`, where m and n are the number of elements of the respective trees.

Extra: Standard Traversals

Some extra questions:

Implement a procedure `merge(t1, t2)` that merges two AVL trees t1 and t2 into a single AVL tree. This should take `O(m+n) time`, where m and n are the number of elements of the respective trees.

> Get the in-order traversal of both t1 and t2. You can use the `merge method` to combine these two sorted numbers into a single in-order traversal, to which you can construct the AVL tree from.

Extra: Standard Traversals

More extra questions:

- Given pre-order traversal and in-order traversal, can you construct the BST? Is this BST unique?
- Given post-order traversal and in-order traversal, can you construct the BST? Is this BST unique?
- Given pre-order traversal and post-order traversal, can you construct the BST? Is this BST unique?
- Given only the post-order traversal, can you construct the BST? Is this BST unique?

Answers on Google (or Leetcode?) :)

06

DOES THIS PREFIX EXIST

this is only for CS2040S, please skip otherwise :)

Does this prefix exist?

Given a set of N strings which are all of length L, give an algorithm to check if a string k is a prefix of some string in the set in O(k.length) time after spending at most O(NL) time preprocessing the set of strings.

Example:
N = 6, L = 5
S = {HELLO, BELLY, BELLO, JELLY, JELLO, MELLO}

Some possible strings for k:
- HELL? Yes.
- PELL? No.
- BELL? Yes.
- B? Yes.
- K? No.

Does this prefix exist?

Use a trie right away :)

Preprocessing: insert all strings to the trie, but you have to show why is this O(NL) time!

Each insertion takes O(L) time so doing for N strings will take O(NL) time.

To answer the query:
Just search for k in T.

There is a path from root that matches all characters in k that means k is a prefix of some string in S. The time taken for this is O(k.length) since we go through each character one-by-one.

THE END!