U21EX – Intro to Trees

Table of Contents

INTRODUCTION

About

asdfasdf

Goals

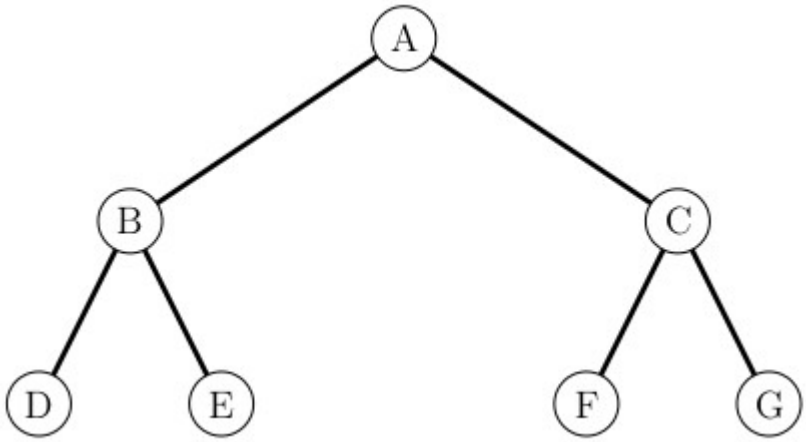
- asdfasdf

asdf

PART 1: INTRO TO TREES

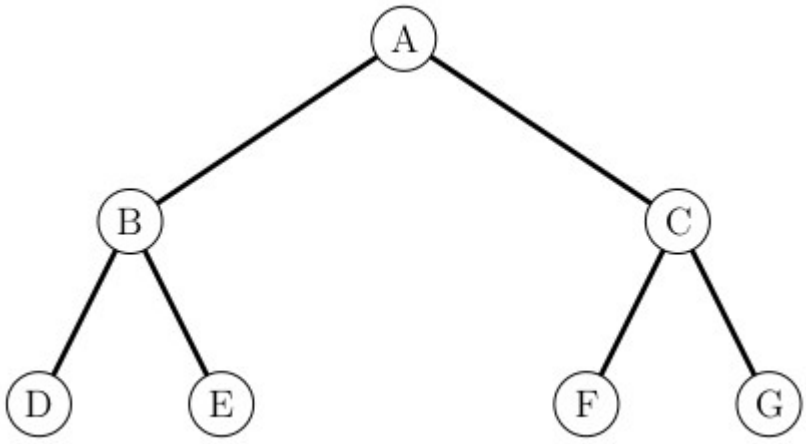
Question 1 – Node and Edge identification

Label the Nodes and Edges in the following diagram:



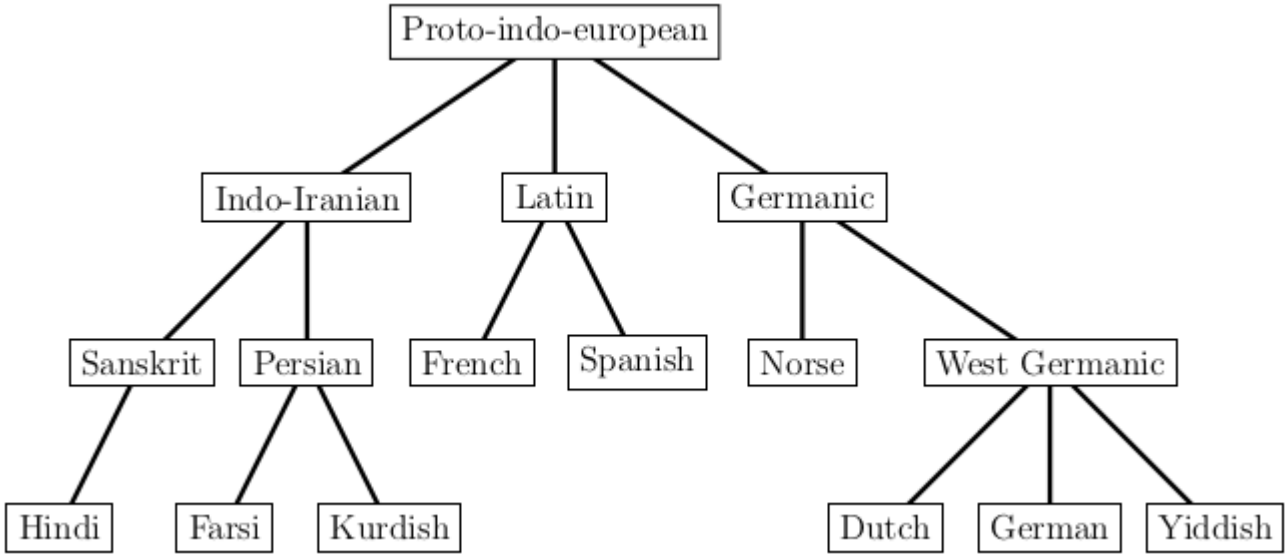
Question 2 – Leaf and Root identification

Label the leaf and root notes in the following diagram:



Question 3 – Ancestors and Descendants

Given the following tree:



An incomplete tree of language families related to Proto-Indo-European

Identify the following:

a) Children of Persian

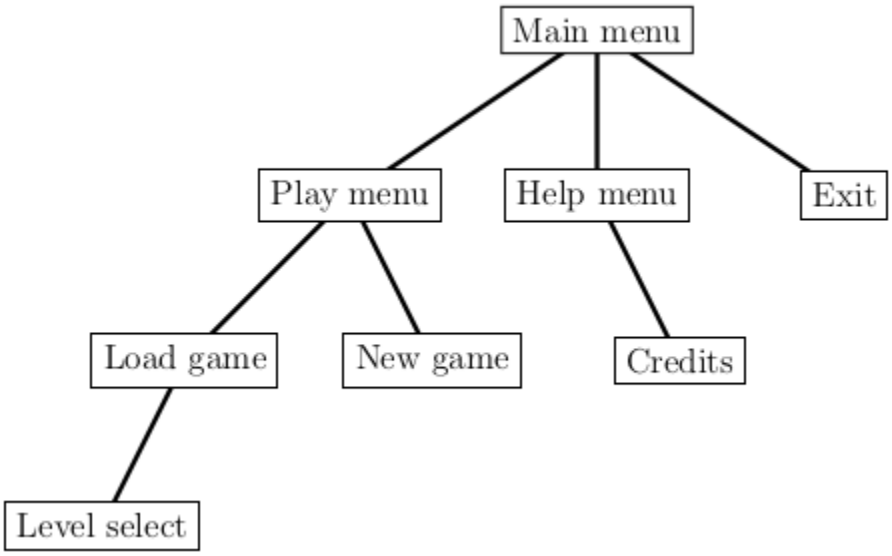
b) Parent of French

c) Ancestors of Spanish

d) Descendants of Germanic

Question 4 – Heights

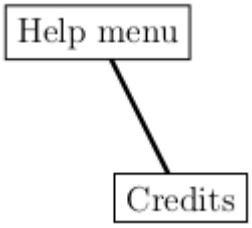
Given the following tree, identify the height:

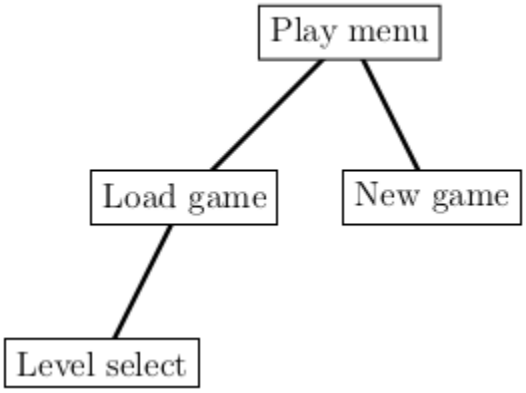


An example tree of screen navigation in a game.

Height =

Question 5 – Identify the height of each subtree

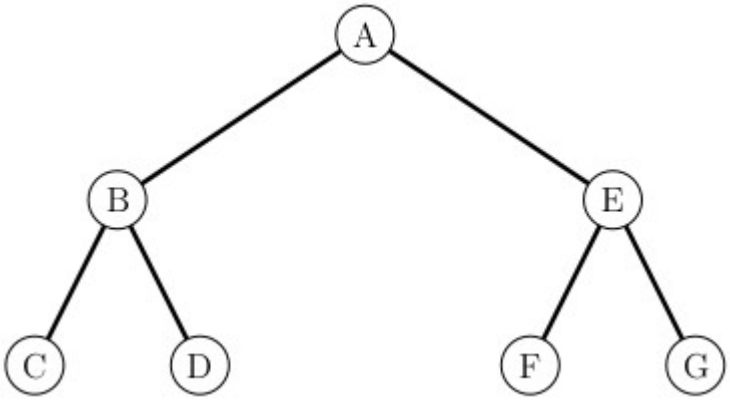




Height = Height = Height =

Question 6 – Tree traversals

Identify the preorder, inorder, and postorder traversals of the following binary tree:



a) Preorder:

b) Inorder:

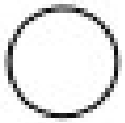
c) Postorder:

PART 2: BINARY SEARCH TREES

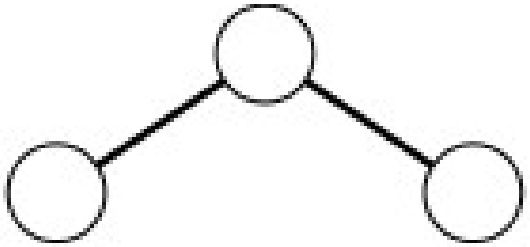
Question 7 – Building a binary search tree

Label the binary tree after each Push. The first step will just be adding the root node, and after that smaller values go left and larger values go right.

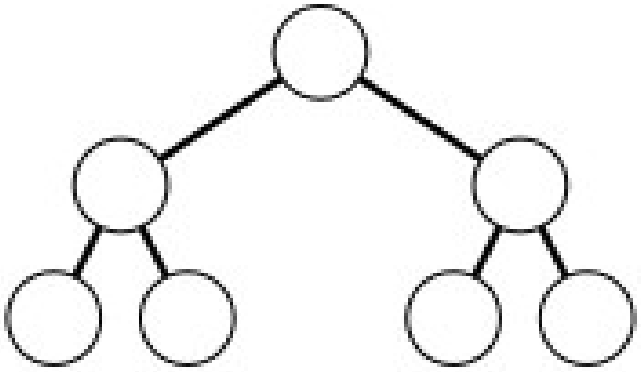
Push(5);



Push(2);
Push(7);



Push(1);
Push(8);
Push(4);
Push(6);



Question 8 – Building a Binary Search Tree

Draw the Binary Search Tree that results after the following series of commands:

```
Push(5);
Push(2);
Push(9);
Push(10);
```

Question 9 – Building a Binary Search Tree

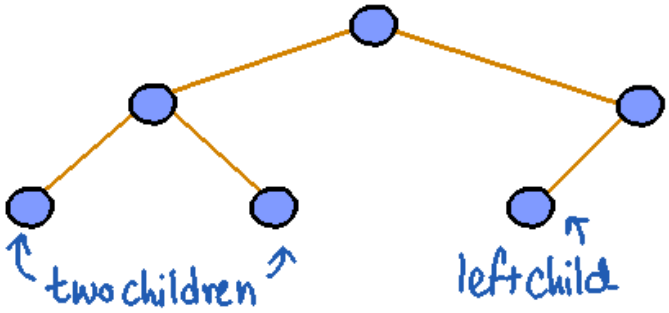
Draw the Binary Search Tree that results after the following series of commands:

```
Push(9);
Push(10);
Push(7);
Push(8);
Push(12);
Push(6);
```

PART 3: HEAPS

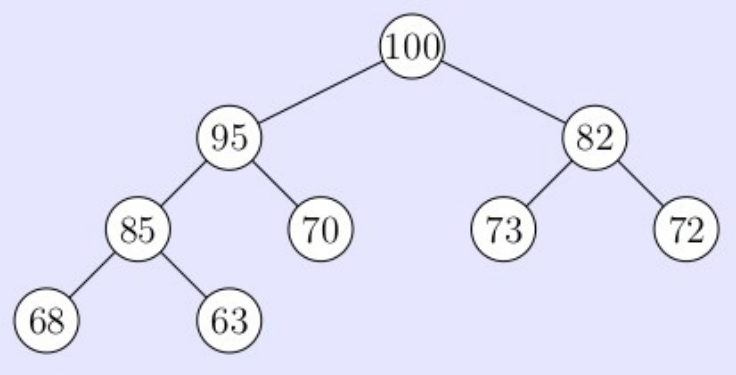
Complete binary tree: A complete binary tree is one where all nodes have 2 children, except possibly in the lowest level. Additionally:

- In the 2nd-to-lowest level, if a node has children, then all its siblings to its left must have 2 children each.
- If a node in the 2nd-to-lowest level has only one node, it must be a left child.
- This mean the tree fills from left-to-right.



Heaps:

"A heap container is a binary tree data structure that is not used for searching, [...] In a heap data structure the node keys are always larger than their child nodes, but their children nodes are not in any particular order. This differs from the binary tree, in which the left child is always the smaller node, while the right node is always the larger value when compared to their parents."



"Three major characteristics make a heap data structure what it is: (a) A heap is a binary tree. (b) A heap is a complete data structure, meaning the rows of nodes are completely filled in from left to right without any gaps, [...] (c) Every node in a heap is larger than or equal to its child nodes."

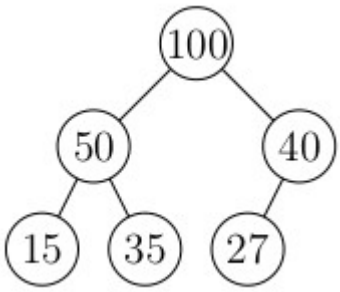
From Data Structures and Algorithms for Game Developers, by Allen Sherrod, page 328-329

If a heap has the larger values as parents, it is known as a maxheap. Otherwise, if the parents have smaller values than their children, the heap is a minheap.

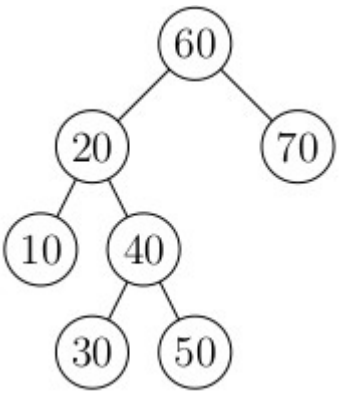
Question 10 – Identifying Heaps

Remember that the criteria for a heap is (A) a heap is a binary tree, (B) a heap is complete, and (C) every node in a heap is larger than or equal to its child nodes (for a maxheap). Identify which of the following are heaps. If a graph is not a heap, please specify what makes it invalid.

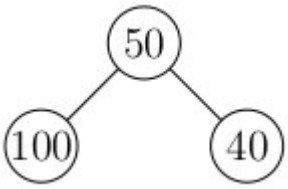
a)



b)

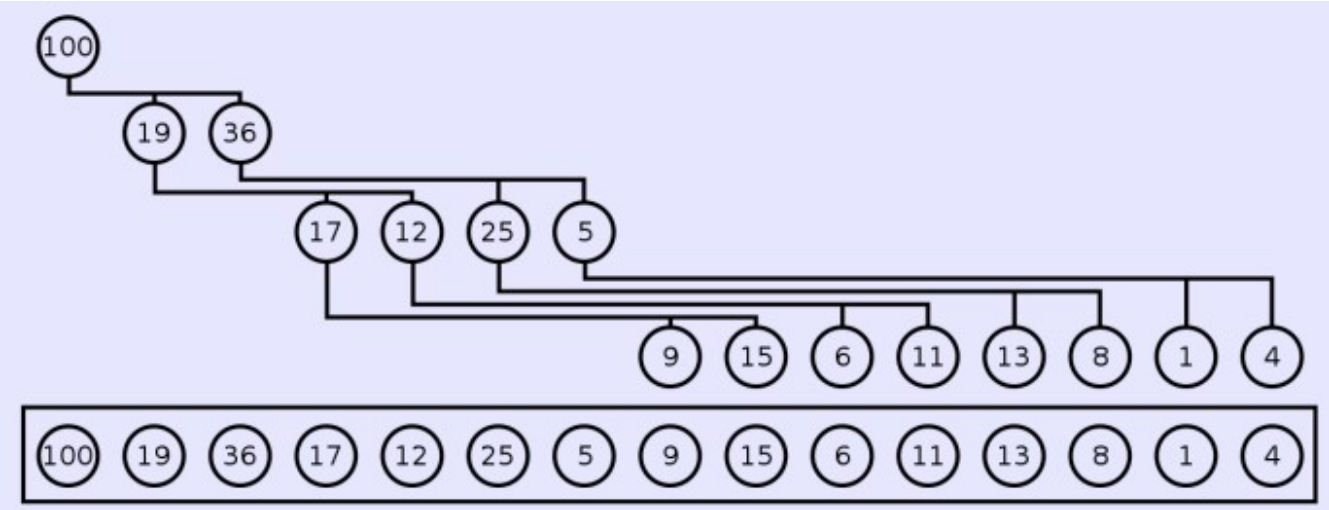


c)



Implementation of a Heap

Even though heaps are visualized as a binary tree, they are usually implemented with an array structure, with the root node (the node with the greatest value) being at index 0.



"Example of a complete binary max-heap with node keys being integers from 1 to 100 and how it would be stored in an array." by Maxiantor; downloaded from https://en.wikipedia.org/wiki/Heap data structure#Implementation

With a heap implemented in this way, you can locate children and parents by modifying the index of a given node at position i (that is, item[i])

- Its left child, if it exists, is **`items[2 * i + 1]`**
- Its right child, if it exists, is **`items[2 * i + 2]`**
- Its parent child, if it exists, is **`items[(i - 1) / 2]`**

Remember that only the root in **`items[0]`** does not have a parent.

From Data Abstraction & Problem Solving with C++: Walls and Mirrors 7th ed, by Carrano and Henry, page 489

Question 11 – Array to Tree

Draw the heap in binary tree form that corresponds to the given array. Remember that, for a node i, the node's left child is at 2i + 1 and the node's right child is at 2i + 2.

	items
0	100
1	80
2	50
3	70
4	60
5	40
6	
7	

Inserting into a Heap

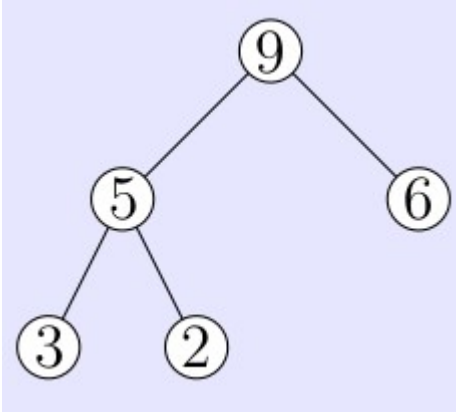
"Although a heap is weakly ordered, the purpose of the data structure is to allow for fast removal from the top of the heap. When an item is inserted into the data structure, it is initially placed on the bottom of the list. The element cannot stay at this position because its value might violate the heap condition that states that every child must be smaller than its parent. Thus, when an element is inserted into the container, it is moved up through the list until it finds an index where the element is smaller than its parent but larger than its children."

From Data Structures and Algorithms for Game Developers, by Allen Sherrod, page 329

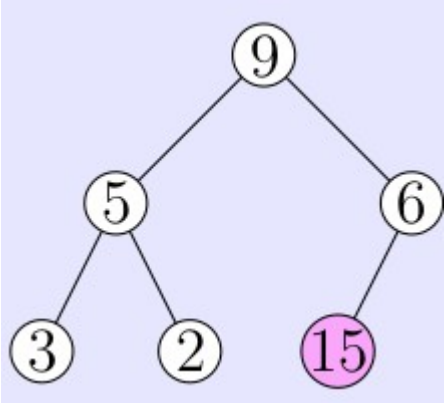
Example: Adding the number 15 to the following Heap

From Data Abstraction & Problem Solving with C++: Walls and Mirrors 7th ed, by Carrano and Henry, page 523

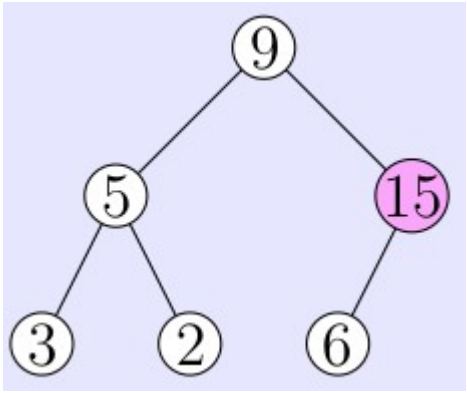
Original heap



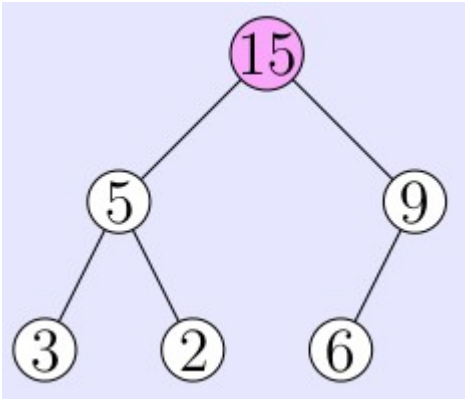
Add 15



Bubble up



Bubble up

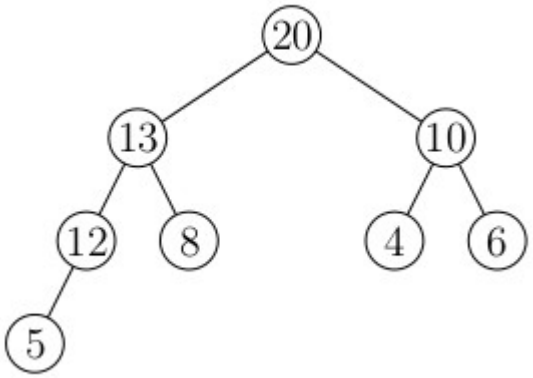


- First, the new node is added to the next available space. Again, the Heap fills from left to right because it is a complete binary tree.
- If the new location invalidates the heap, then it “bubbles up” and it and its parent swap positions.
- It continues bubbling up until (1) its parent is greater than it, and (2) its children are less than it.

Question 12 – Bubble up

With the Heap given, draw each step as you insert the new node and bubble it up to a valid location.

Insert: 15



Removing from a Heap

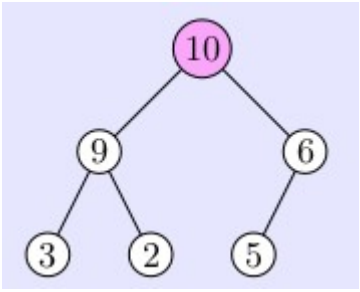
“Removing is done from the top of the heap. When a heap is implemented as an array, this means removing element 0 since the element at index 0 is always the root node. Once the root has been removed, the heap is no longer complete because of the newly made hole, and it must be replaced with the next elements in line.”

From Data Structures and Algorithms for Game Developers, by Allen Sherrod, page 329

Example:

From Data Abstraction & Problem Solving with C++: Walls and Mirrors 7th ed, by Carrano and Henry, page 521

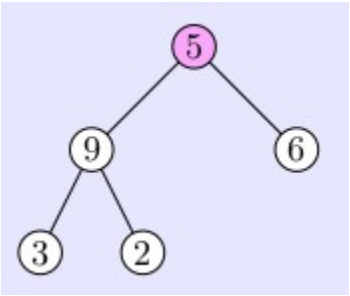
Heap



Remove 10...

10	9	6	3	2	5
0	1	2	3	4	5

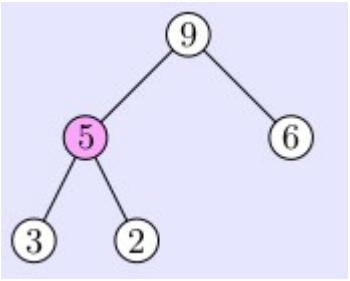
Semi-heap



Last entry moved to root

5	9	6	3	2	
0	1	2	3	4	5

Restored heap



Bubble down

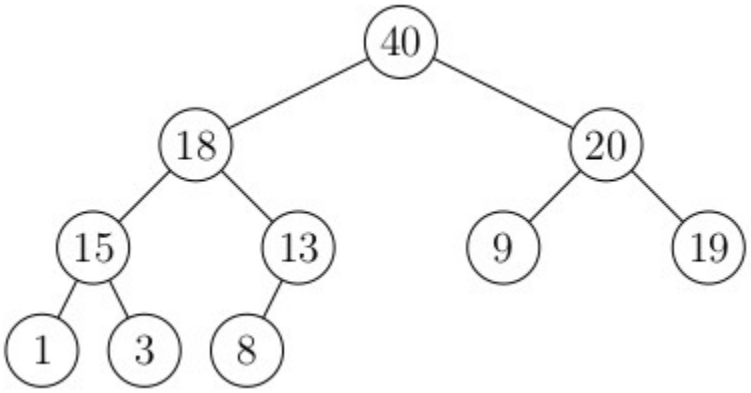
9	5	6	3	2	
0	1	2	3	4	5

When removing an item from a heap, we remove the item at position 0 – the root. In order to maintain the tree structure, we replace that root item with the last item in the heap, remembering that the heap fills from left-to-right.

As we choose a direction for the root to bubble down, keep in mind we want the maximum value to be the root. Keep bubbling down the node until it gets to a position where its parent is greater than itself (for maxheap).

Question 13 – Remove from Heap

With the heap given, draw each step as you remove the root, replace it with the last node, and then bubble down to restore the heap to a valid state.

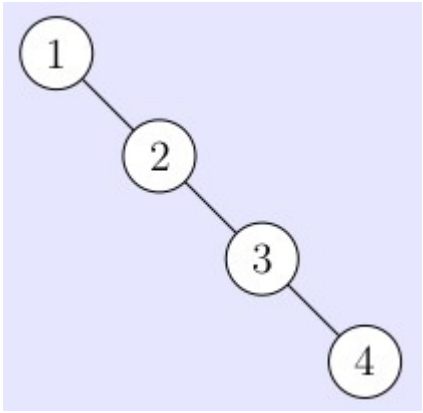


PART 4: BALANCED SEARCH TREES

Unbalanced trees

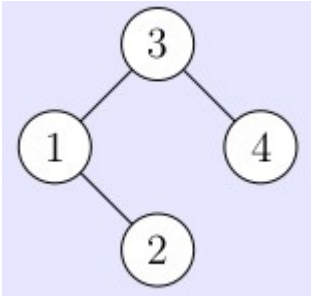
There are multiple types of Balanced Search Tree data structures available for us to use, with the common feature of them being that they remain balanced after any kind of action (insert, remove, etc.) We will look at a few of these tree types and how they work, so that you can gain an understanding of their operations in general. Further down the road, if you need to implement them later, you'll at least know in general how they work.

Recall that with a binary search tree, if you happen to insert items in an ascending or descending order, you end up just getting a straight line...:




```
Push(1); Push(2); Push(3); Push(4);
```

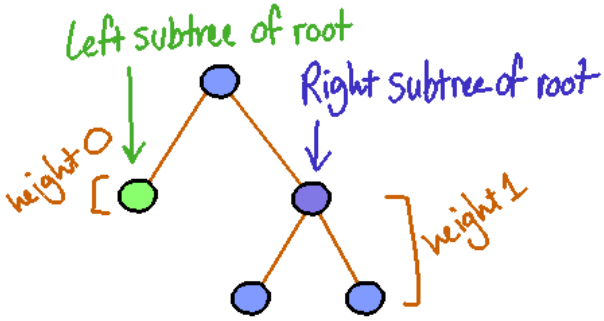
However if you insert things at a more-or-less random unsorted order, we will generally get a more balanced tree:




```
Push(3); Push(1); Push(2); Push(4);
```

Here, the downfall of the Binary Search Tree is that it can become unbalanced, with many nodes falling to one side or the other. Instead, we can use a Balanced Search Tree, which will rebalance itself when it detects an imbalance…

Balanced binary tree: A binary tree is height balanced when, for each node of the tree, each node's left subtree and right subtree differ in height by no more than 1 level.



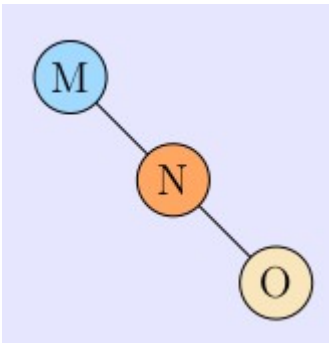
AVL Trees

For an AVL tree, the two subtrees of each node differ by no more than one. After an operation (e.g., insert, remove), if the height difference is more than one, the tree rebalances itself.

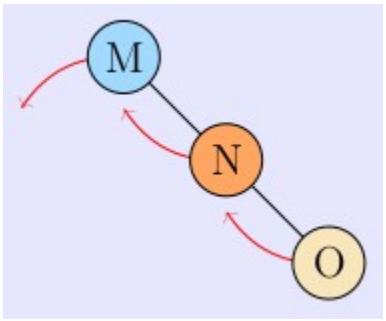
Example:

Diagram from https://en.wikipedia.org/wiki/AVL tree

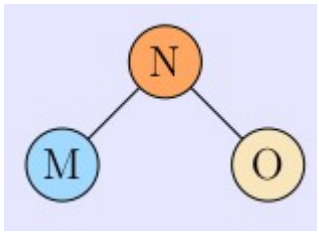
a. Say that we have the following unbalanced tree:



b. We can balance it by pushing the N node **up**, and push the M node **down…**



c. Which gives us this tree:



(The wikipedia page has a nice animated version, which can be helpful to look at!)

Height and Balance Factor

A height-balanced tree: A tree is height balanced (aka, simply balanced), if the height of any node's right subtree differs from the height of the node's left subtree by no more than 1.

Left subtree of n: A subtree of a tree T is a tree consisting of a node T and all of its descendants in T .

Height and Balance Factor (continued)

Balance Factor: After an operation is performed on an AVL tree, the Balance Factor of each node is calculated. The equation for a node n's balance factor is:

$$BalanceF\ actor(n) = Height(RightSubtree(n)) - Height(LeftSubtree(n))$$

If the absolute value of the Balance Factor is more than 1, then we need to rebalance. In other words, the balance factor for every node must be -1, 0, or 1. BF (x) < 0 is left-heavy, BF (x) > 0 is right-heavy.

Height of a node: The height of a node is the amount of edges on the longest path between that node and a leaf. A leaf node has a height of 0. An empty tree has a height of -1.

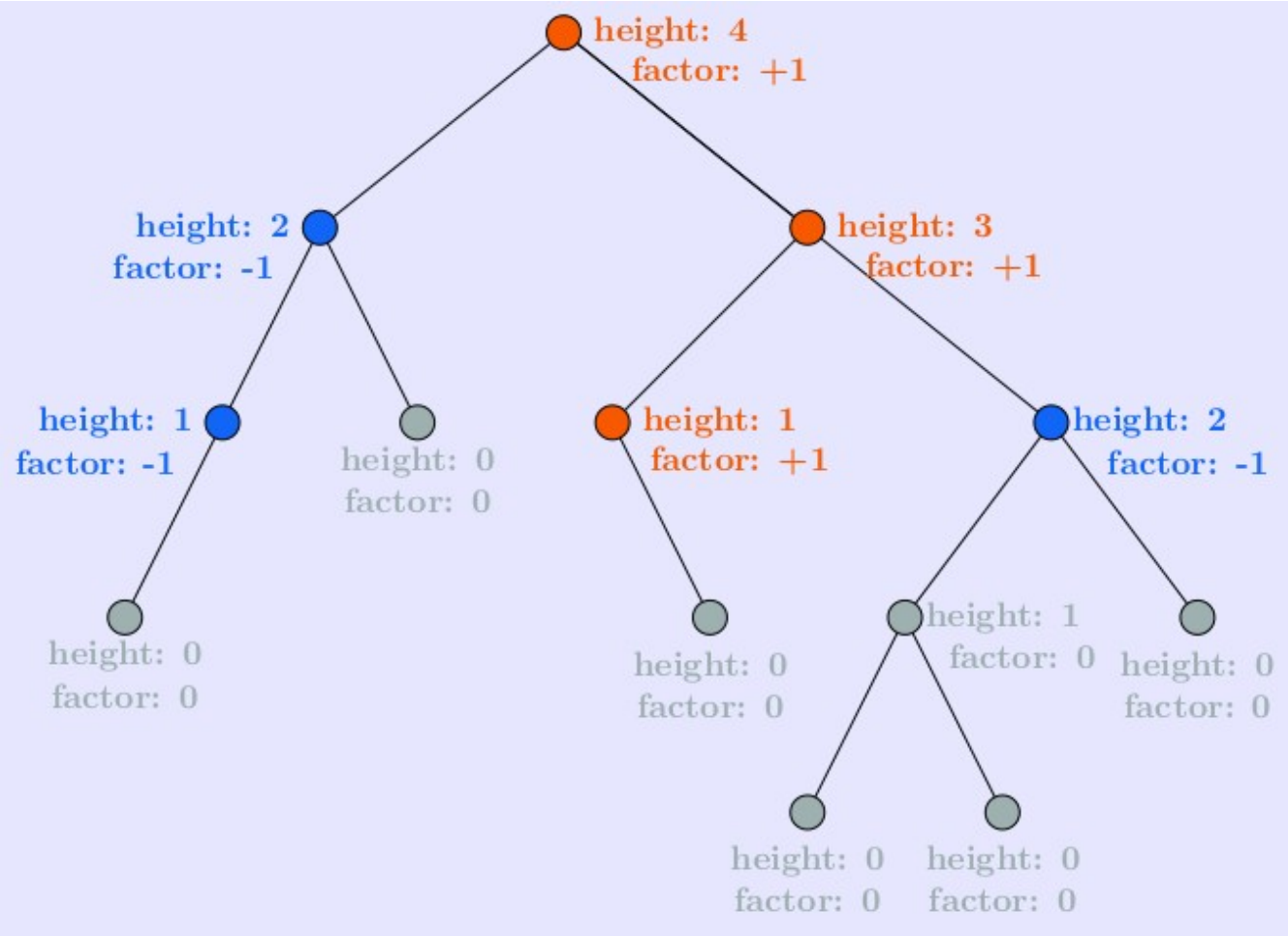


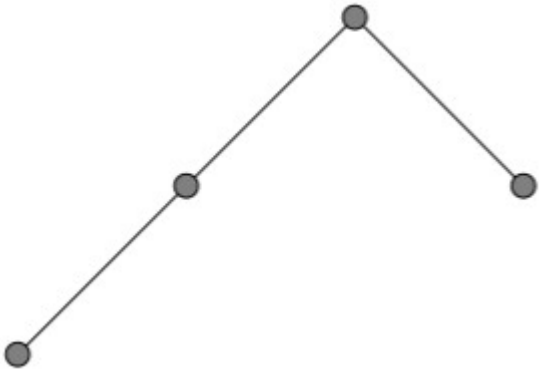
Diagram adapted -

from https://en.wikipedia.org/wiki/AVL tree#/media/File:AVL-tree-wBalance K.svg

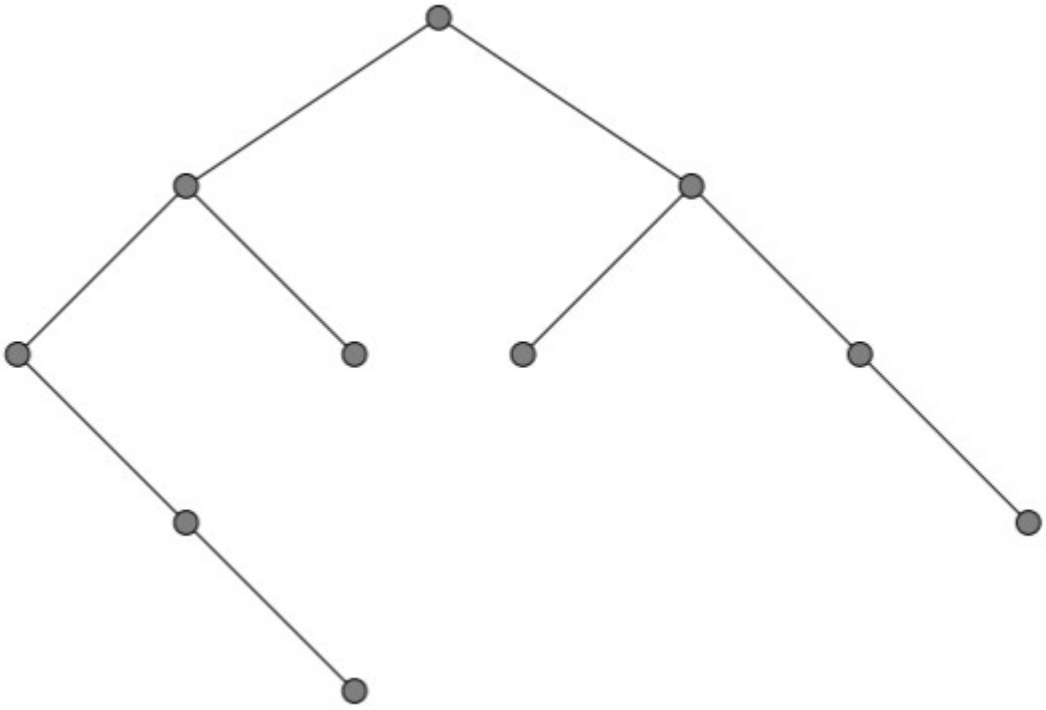
Question 14 – Identifying height and balance factor

For the given trees, label the **height** and **balance factor** of each node.

a)



b)



Rotation examples

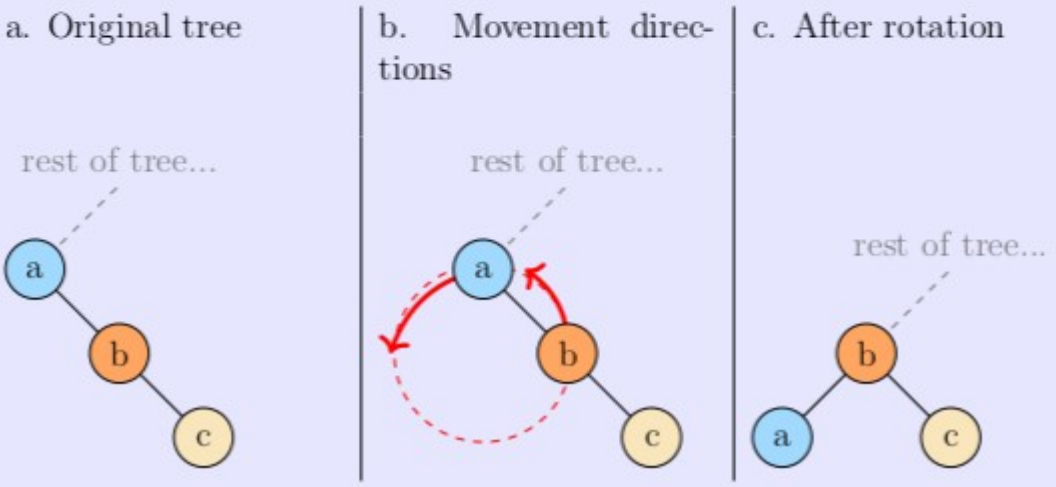
These examples of rotations in AVL trees are from https://www.cise.ufl.edu/ ñemo/cop3530/AVL-Tree-Rotations.pdf

Left Rotation (LL):

Left rotation example

Let's take a subtree that has a balance factor of +2 like in the following example. Node a has a balance factor of +2 because its left subtree is a height of 0 (there are no children to the left), and its right subtree is height 2. Therefore, we are going to do a **left rotation on** a.

1. a's right child, b, becomes the new root of this subtree.
2. If b has a left child, it becomes a's right child.
3. a becomes b's left child.

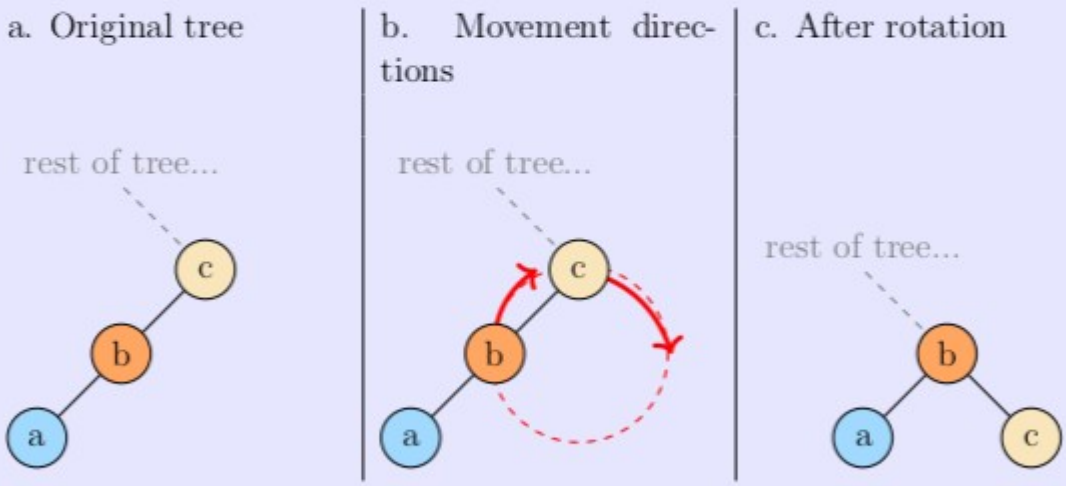


Right Rotation (RR):

Right rotation example

Let's take a subtree that has a balance factor of -2 like in the following example. This time, c has a balance factor of -2 because its right subtree is height 0, and its left subtree is height 2. We are going to do a **right rotation on c.**

1. c's left child, b, becomes the new root of this subtree.
2. If b has a right child, it becomes c's left child.
3. c becomes $b's$ right child.



Note that we are only moving nodes "vertically", so that when we traverse the tree, the in-order (left to right) order is preserved. [a]

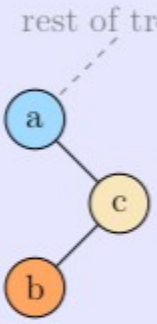
[a] https://en.wikipedia.org/wiki/AVL_tree#Rebalancing

Left-Right Rotation (LR) / aka Double-Left Rotation:

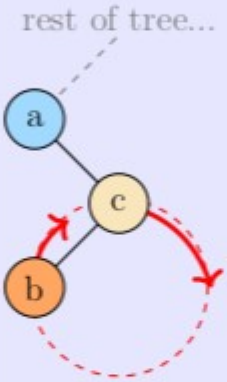
Left-Right rotation example

In some cases, we can't balance a tree with one rotation and have to rotate twice. We will need a Left-Right rotation in cases where our tree is right-heavy, but its right-subtree is left-heavy.

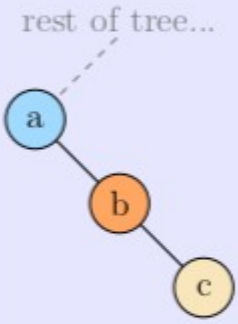
a. Unbalanced subtree



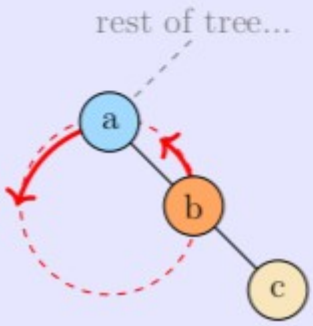
b. First rotation direction



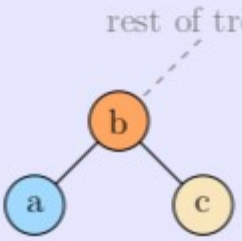
c. After RIGHT rotation on *c*



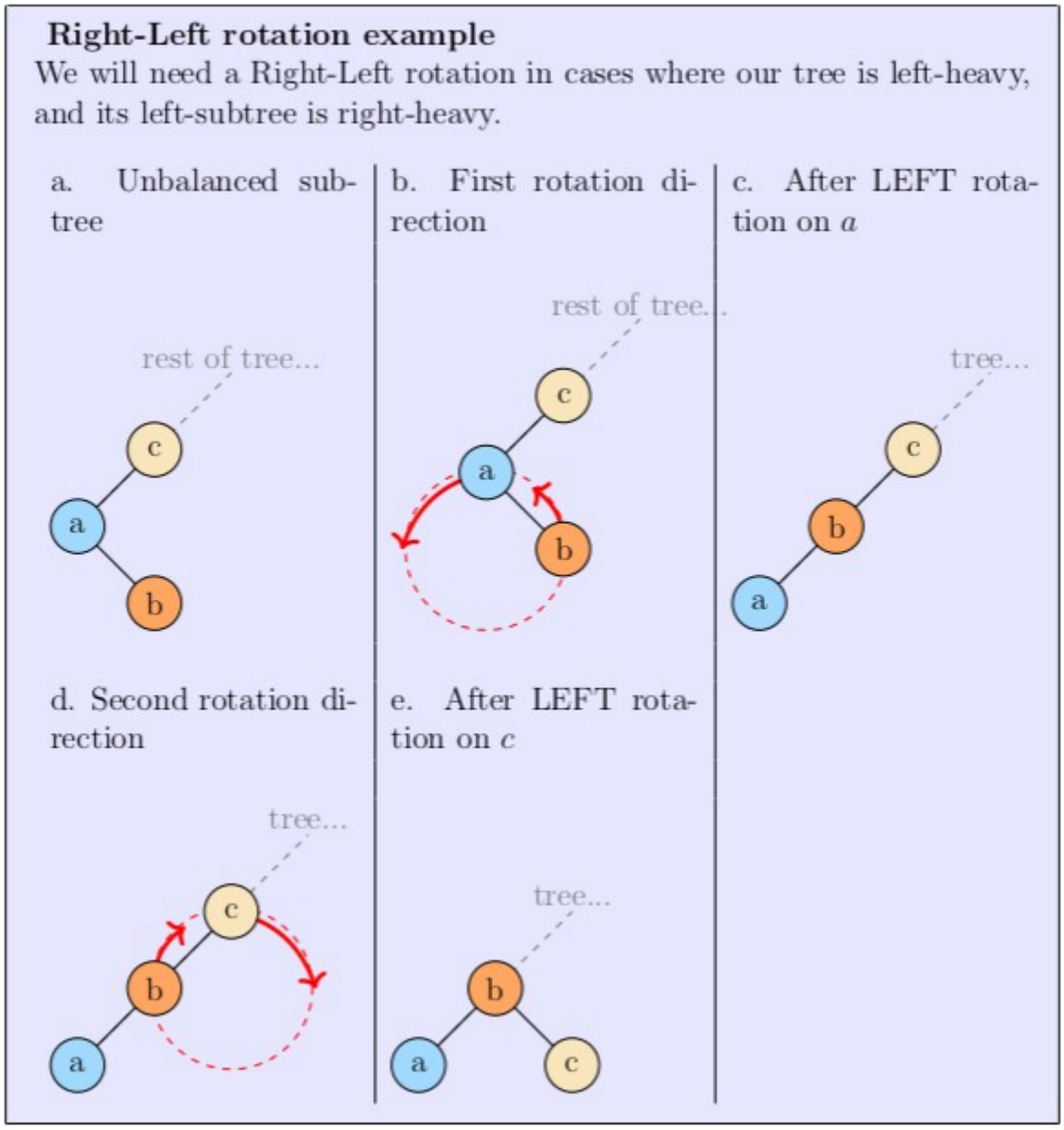
d. Second rotation direction



e. After RIGHT rotation on *A*



Right-Left Rotation (RL) / aka Double-Right Rotation



Pseudocode

Pseudocode

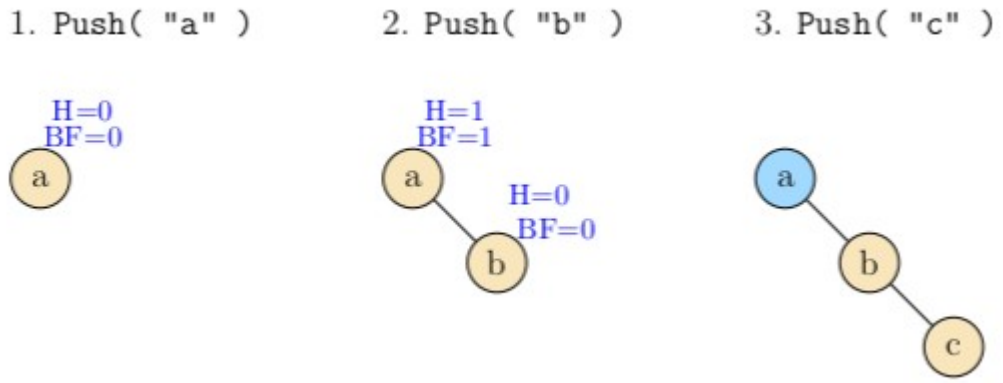
Pseudocode also from
https://www.cise.ufl.edu/~nemo/cop3530/AVL-Tree-Rotations.pdf

```
IF tree is right heavy
{
      IF tree's right subtree is left heavy
      {
          DO double left rotation (LR)
      }
      ELSE
      {
          DO single left rotation (LL)
      }
}

ELSE IF tree is left heavy
{
      IF tree's left subtree is right heavy
      {
          DO double right rotation (RL)
      }
      ELSE
      {
          DO single right rotation (RR)
      }
}
```

Question 15 – Rebalancing trees

An AVL tree is being built…

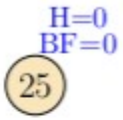


At this point, the tree is unbalanced. Identify each node's height and balance factor. Identify which balance operation will be executed and draw the tree after this rebalance.

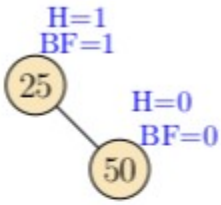
Question 16 – Rebalancing trees

An AVL tree is being built…

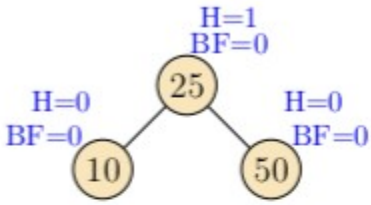
1. Push(25)



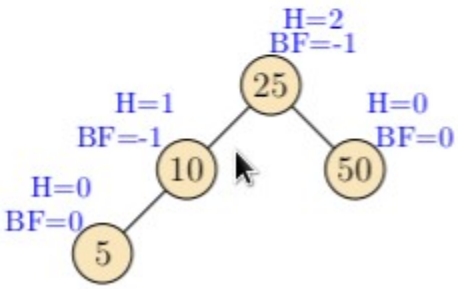
2. Push(50)



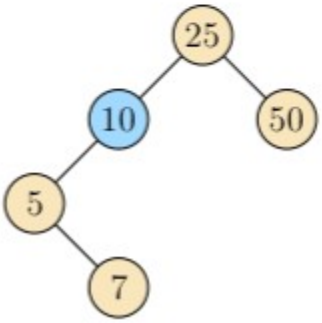
3. Push(10)



4. Push(5)



5. Push(7)

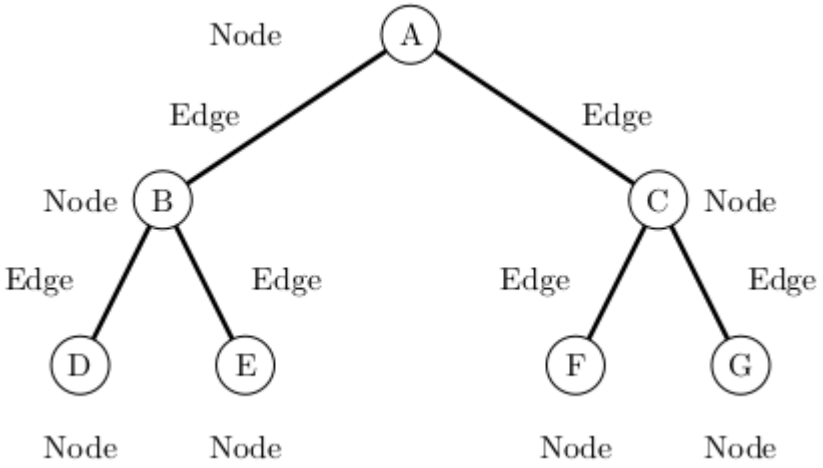


At this point, the tree is unbalanced. Identify each node's height and balance factor. Identify which balance operation will be executed and draw the tree after this rebalance.

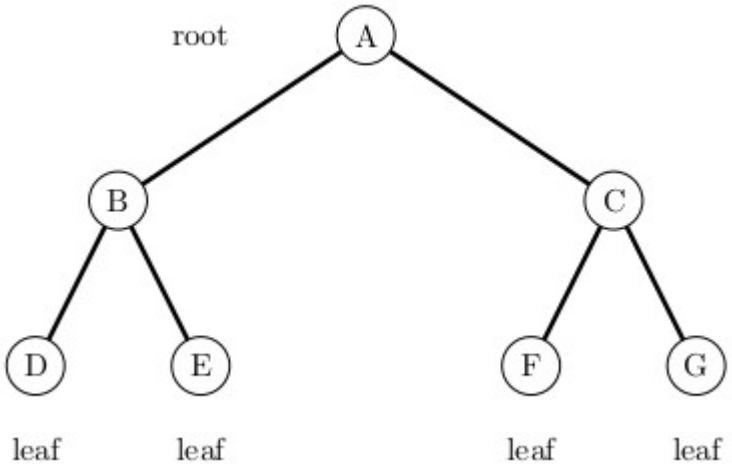
SOLUTIONS

Intro to Trees

Question 1



Question 2



Question 3

a) Children of Persian: Farsi, Kurdish
b) Parent of French: Latin
c) Ancestors of Spanish: Latin, Proto-indo-european
d) Descendants of Germanic: West Germanic, Dutch, German, Yiddish, Norse

Question 4 Height = 3

Question 5 Height = 1 / Height = 2 / Height = 0

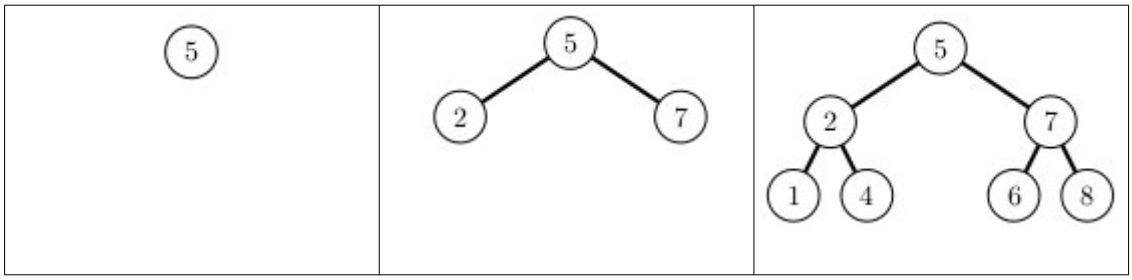
Question 6

a) Preorder: ABCDEFG
b) Inorder: CBDAFEG
c) Postorder: CDBFGEA

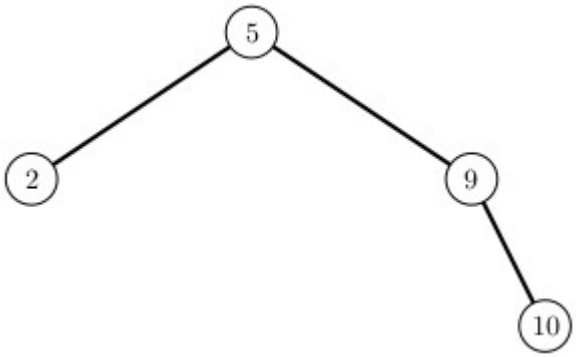
Binary Search Trees

Question 7

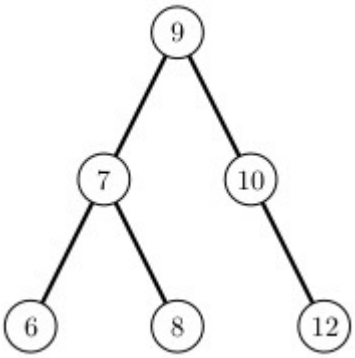
Step 1	Step 2	Step 3



Question 8



Question 9



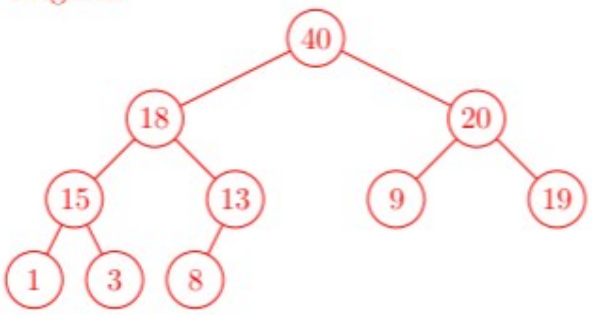
Heaps

Balanced Sear

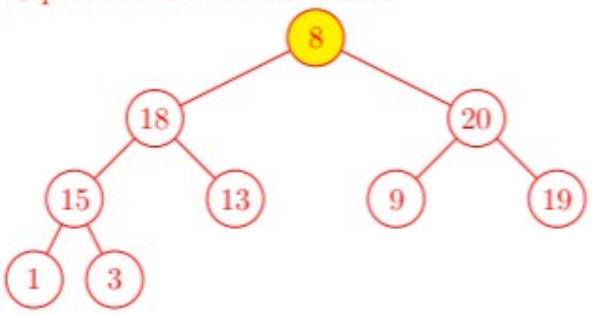
Question 14

Removing root

(a) Original:

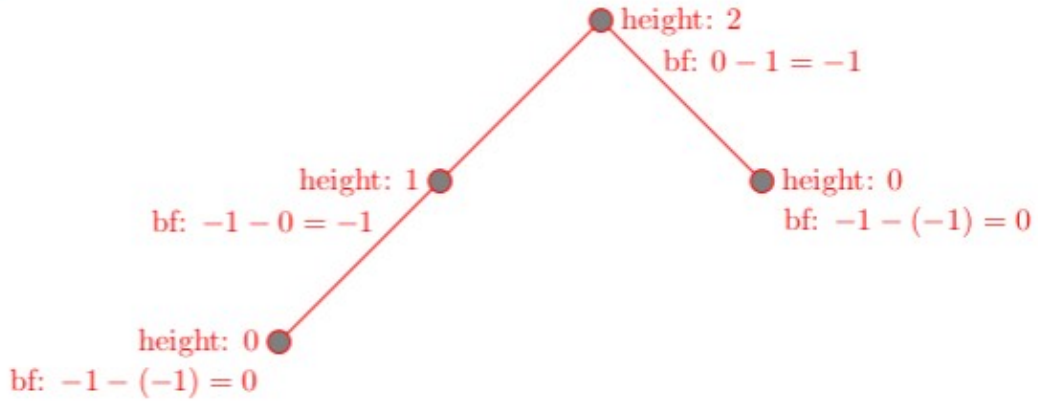


(b) Replace root with last node:



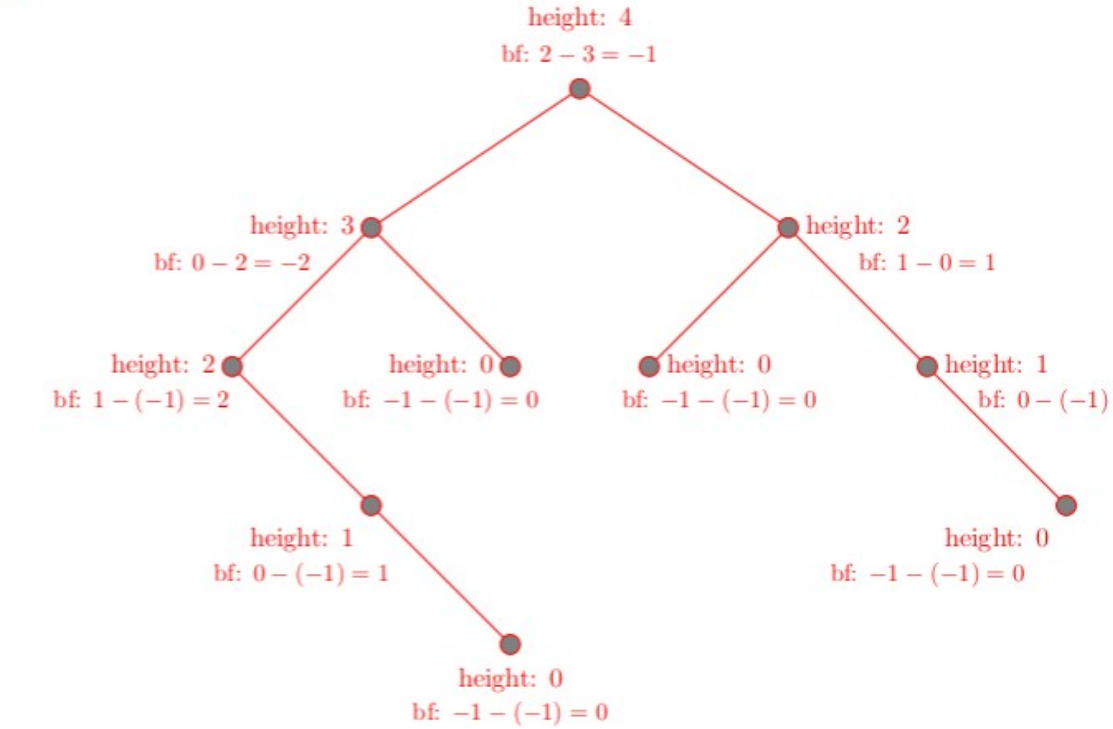
(c) The root should be the max value. $8 < 18$ but also $8 < 20$, so swap 8 and 20

a.



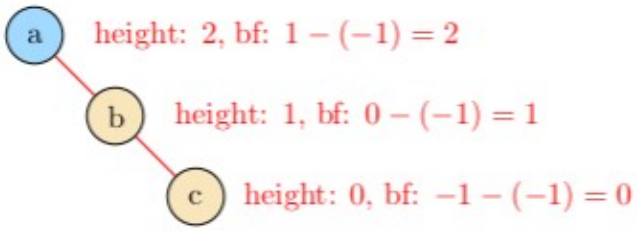
The height of an empty tree is -1. The leaf nodes have no children, so we can think of it as having an "empty" left and right subtree.
When calculating the balance factor, do $right - left$ for each node.

b.

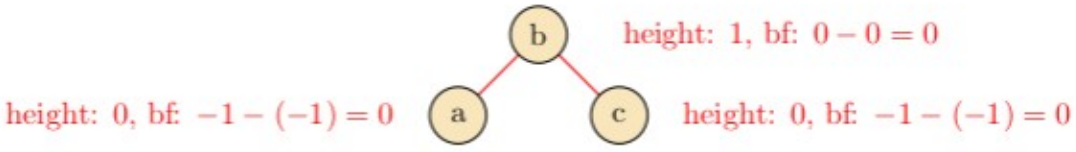


Question 15

Unbalanced tree:

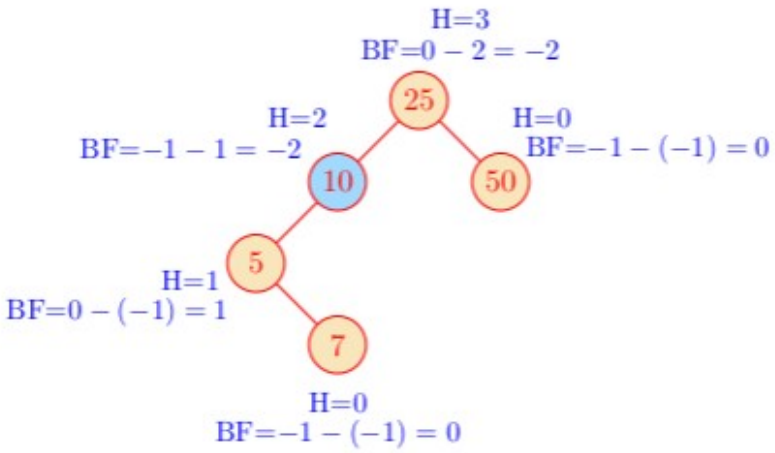


After a Left rotation:



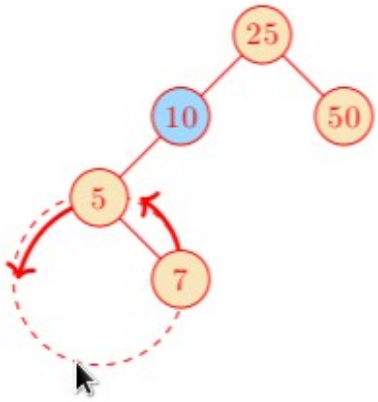
Question 16

Unbalanced tree:

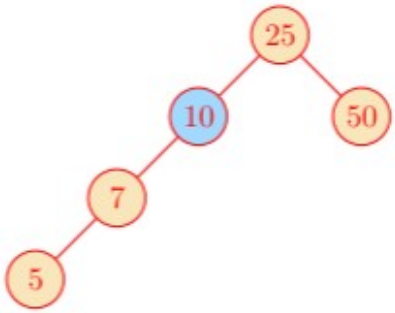


Need a Right-Left rotation:

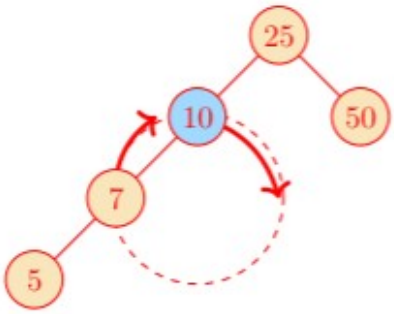
a. Direction of rotation



b. After rotation



c. Direction of rotation



d. After rotation