CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Section 2: Dynamic Data Structures & Binary Trees

Harvard SEAS - Fall 2023 *Sept. 21, 2023*

1 Dynamic Data Structures

When we first solved the *IntervalScheduling* problem, we took a list of intervals as input and checked for conflicts by sorting the start times. But instead of ingesting the list of all the intervals at once, we might prefer to keep a running tally of intervals and notify users of conflicts as they come up. Dynamic data structures allow us to accomplish this goal.

Definition 1.1. A *dynamic data structure problem* Π is given by

- a set $\mathcal{I}$ of *inputs* (or *instances*)
- a set $\mathcal{U}$ of *updates*,
- a set $\mathcal{Q}$ of *queries*, and
- for every $x \in \mathcal{I}$, $u_0, u_1, \ldots, u_{n-1} \in \mathcal{U}$, and $q \in \mathcal{Q}$, a set $f(x, u_0, \ldots, u_{n-1}, q)$ of *valid answers*

This definition encompasses a wide set of abstract data structures that includes dictionaries and dynamic predecessors, which could be implemented by a wide set of data structure implementations including BSTs, AVL trees, linked lists, stacks, queues, and heaps.

2 Binary Search Trees

Binary Search Trees (BSTs) are dynamic data structures that allow us to store and query sorted data efficiently. By structuring data in a hierarchical form, they simplify operations that would be $O(n)$ for arrays and linked lists to $O(\log n)$. They find applications in data indexing, searching algorithms, and other problems that benefit from their nested properties.

Definition 2.1. A *binary search tree (BST)* is a recursive data structure. Every nonempty BST has a root vertex r, and every vertex v has:

- a key K_v
- a value V_v
- a pointer to a left child v_L, which is another binary tree (possibly `None`, the empty BST)
- a pointer to a right child v_R, which is another binary tree (possibly `None`, the empty BST)
- (optionally) a pointer to a parent p

Crucially, we also require that the keys satisfy the *BST Property*:

If v has a left-child v_L, then the keys of v_L and all its descendants are no larger than K_v, and similarly, if v has a right-child, then the keys of v_R and all of its descendants are no smaller than K_v.

Note that the empty set satisfies all the properties above, and is a BST.

Here is a sample class specification for a generic binary tree.

```
class BinaryTree:
    left: BinaryTree
    right: BinaryTree
    key: string
    item: int
```

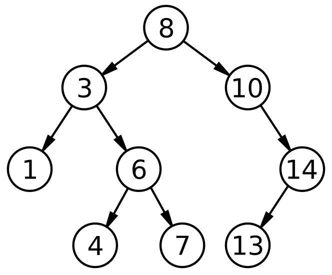
Question 2.2. Suppose you want to write a recursive Python function `isValidBST` that returns `True` if a given tree `t` is a valid BST, `False` otherwise. Your friend Binary Bob proposes the following code:

```
def isValidBST(T: BinaryTree):
    if T is None:
        return True

    leftCond = T.left is None or T.left.value <= T.value
    rightCond = T.right is None or T.right.value >= T.value

    return leftCond and rightCond and isValidBST(T.left) and isValidBST(T.right)
```

What is the conceptual error in Bob's solution? What changes would you make to his code to implement `isValidBST` correctly?

Below is an example of a binary tree that satisfies the BST Property.

Figure 1: A binary tree that satisfies the BST Property. *(Image source: Wikipedia)*



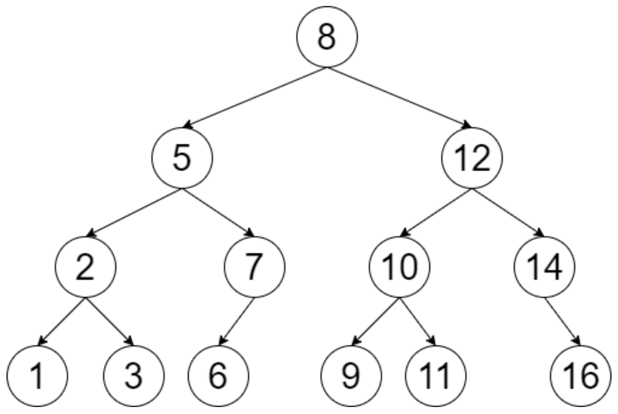
Question 2.3. Perform the following operations on a binary tree and draw each intermediate tree.

```
T = BinaryTree(5)
T.insert(3)
T.insert(9)
T.insert(10)
T.insert(12)
T.insert(2)
T.insert(6)
```

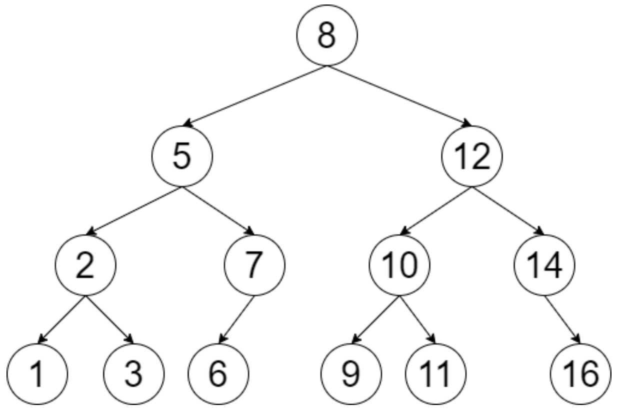
Question 2.4. What is the runtime for insert and delete updates, search, minimum, maximum, next-smaller, and next-larger queries? State the runtime in terms of h, where h denotes the height of the BST.

Insert	Delete	Search	Minimum	Maximum	Next-smaller	Next-larger

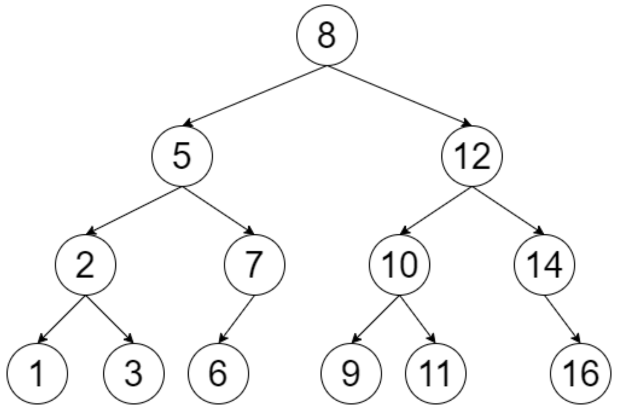
Question 2.5. Given the following BST, mark the succession of nodes you would visit in order to find the next-smaller of 15, and then describe the generic algorithm that you would follow for an arbitrary BST T and query q.



Question 2.6. Given the same BST, mark the succession of nodes you would visit in order to find the next-larger of 4, and then describe the generic algorithm that you would follow for an arbitrary BST T and query q.

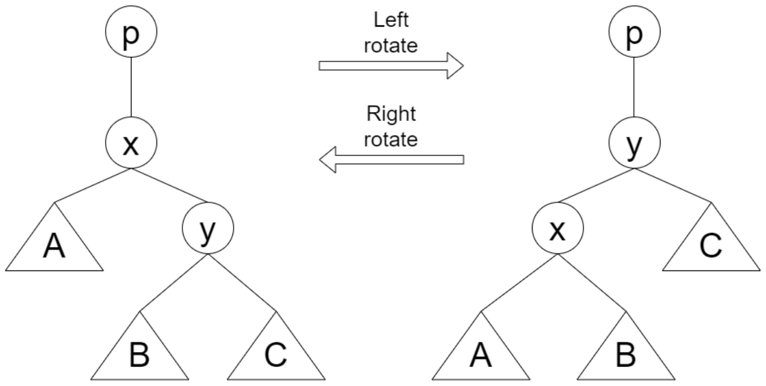


Question 2.7. Given the same BST, mark the succession of nodes you would visit in order to find the minimum element, and then describe the generic algorithm that you would follow for any given BST.

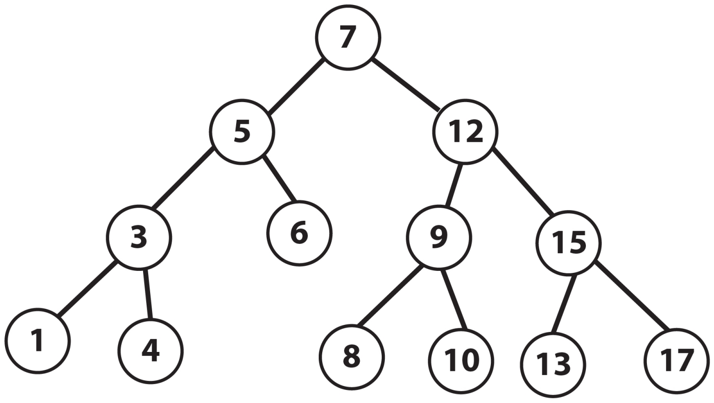


2.1 Rotations on BSTs

Recall from Lecture 5 where we learned about rotation in a binary search tree:



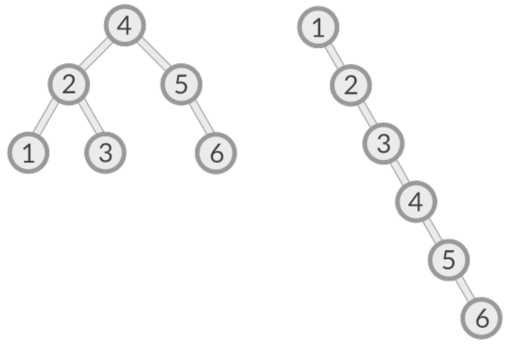
Question 2.8. Let `rotate(direction, child)` be a function that can be called on a node to rotate its child. In the example above, if x is a left child of p, then the left rotation is denoted by `p.rotate("L", "L")` and the right rotation is denoted by `p.rotate("R", "L")`. Consider the following binary tree below. Let T be the root node. `T.rotate("R","L")`. Draw the tree after the operation is called.



3 AVL Trees

AVL Trees[1], also known as *height-balanced trees*, are BSTs with the additional property that the difference between the height of the left and right subtrees of any node is at most 1.

Figure 2: In the worst case, an unbalanced BST has a height of $O(n)$. In fact, searching the second tree has the same time complexity as searching a linked list. *(Image source: Applied Go)*



In class, we mentioned that we can perform a single rotation in $O(1)$ time, and stated (without proof) that $O(\log n)$ rotations suffice to restore balance to an AVL tree in which one key has just been inserted. In fact, the same is true for deletion, but we'll first need to cover deletion in (unbalanced) Binary Search Trees, which is the topic of Thursday's Sender-Receiver Exercise.

Question 3.1. Given a BST as an input, write a program to determine if it is a valid AVL Tree. Implement `is_avl_tree(T: BinaryTree)`.

[1] AVL Trees are named after their inventors, Adelson-Velskii and Landis.