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

Section 2: Reductions & Dynamic Data Structures

Harvard SEAS - Fall 2022 *Sept. 15, 2022*

1 Preview of Reductions

Reductions are one of the most powerful tools we have as computer scientists. We'll talk more about reductions later in the course, but for now, here's a preview:

Informally, we say we can reduce problem A to problem B if we find ourselves saying: "if only I could solve problem B! Then, I could solve problem A pretty easily (with only a bit more time)." You can think of this as problem B being inside A's "belly", and A is able to use B as a subroutine for as many times as needed. We will need to keep track of the number of times that we call B and know the runtime of B, but you can always assume the correctness of B and treat it as a black-box.

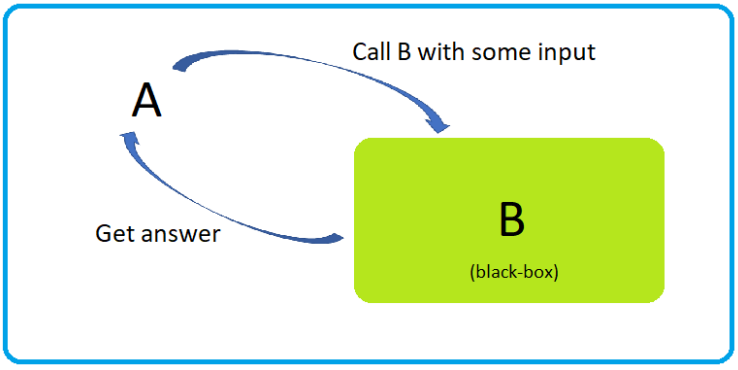


Figure 1: Illustration of a reduction from A to B. You can think of B as an oracle.

We can reason about reductions by pretending we have an oracle for problem B: some magic black-box function that instantly provides an answer to any possible input to B. Then, we can think of our reduction as a program for solving A that calls B's oracle as a subroutine. Here's an example, where we reduce the problem of getting someone's bank balance to that of getting their bank account password:

```
def get_bank_balance(username, encrypted_bank_info):
    password = ORACLE_get_bank_password(username)
    bank_info = encrypted_bank_info.decrypt(username, password)

    balance = 0
    for transaction in bank_info:
        balance += transaction.amount
    return balance
```

Here, we say `get_bank_balance` **reduces to** `ORACLE_get_bank_password` (it's very easy to get wrong the direction of the reduction, so be careful!). Getting the balance of someone's bank account given only their username is a hard problem, and so is getting their bank account password. But we can imagine all the steps we would need to get their balance IF we had some way of getting their password; we could do it in only $O(n)$ more steps. So, if we call the runtime of `ORACLE_get_bank_password` $= RT(ORACLE)$, we can be sure that the runtime of `get_bank_balance` is $O(n + RT(ORACLE))$. In general, if your transformation takes $O(f(n))$ takes, we call it an $O(f(n))$-time reduction.

Definition 1.1. Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and $\Gamma = (\mathcal{J}, \mathcal{Q}, g)$ be two computational problems. A *reduction* from Π to Γ is an algorithm that solves Π using as a subroutine a(ny) *oracle* that solves Γ. An oracle is a function that, given any input $x \in \mathcal{I}$ to a computational problem $P = (\mathcal{I}, \mathcal{O}, f)$, returns an element of $f(x)$.

If there exists a reduction from Π to Γ, then we write $\Pi \leq \Gamma$ (read "Pi reduces to Gamma").

Notice that reductions are useful both in a "positive" and a "negative" sense: Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:

1. (Positive.) If there exists an algorithm solving Γ, then there exists an algorithm solving Π.
2. (Negative.) If there does not exist an algorithm solving Π, then there does not exist an algorithm solving Γ.

This is also why we have to be very careful when writing the **direction** of the recursion.

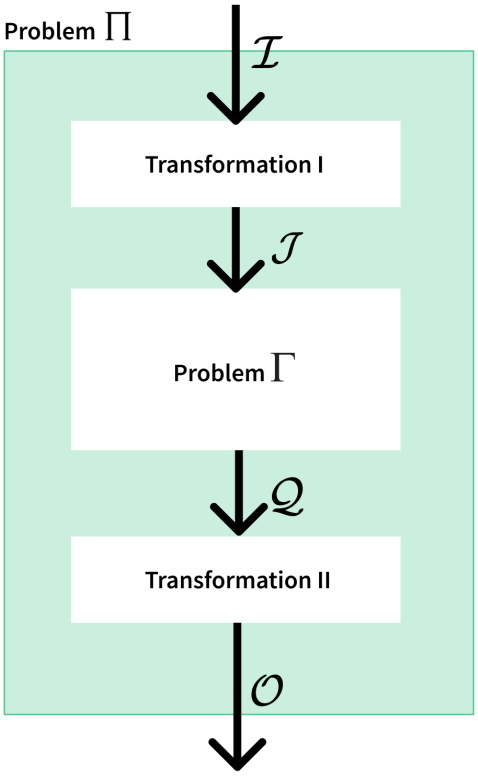


Figure 2: A reduction from Π to Γ which calls the oracle for Γ exactly once.

Now, recall the *IntervalSchedule* problem from Lecture 4, and think about the following questions:

Question 1.2. Conceptual questions about reduction

- Did we reduce Sorting to IntervalSchedule, or IntervalSchedule to Sorting? Why?

- What was the runtime of the transformation to/from the Sorting problem? How did we use this fact to bound the runtime of IntervalSchedule?

Now, let's try a reduction problem in full.

Question 1.3. Describe an algorithm to find the median of an array by a reduction to sorting.

Question 1.4. Given points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ in the $\mathbb{R}^2$ plane, we want to determine whether point $p = (x_0, y_0)$ is *collinear* with at least three other points in the set.

- Describe a brute force algorithm that solves this problem. What is its time complexity?

- Describe a faster algorithm via a reduction to sorting. What is its time complexity? *(Hint: Points that are collinear with point p all have the same slope relative to p. For example, if* $p = (0, 1)$*, points* $(1, 2)$*,* $(2, 3)$*, and* $(107, 108)$ *all have slope* 1 *relative to p.)*

2 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 2.1. 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 data structures that includes BSTs, AVL trees, linked lists, stacks, queues, and heaps.

3 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 3.1. 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
```

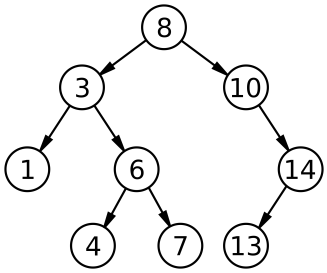
Question 3.2. Suppose you want to write a recursive Python function `isValidBST` that returns `True` if a given tree `t` is a valid BST, `False` otherwise. Your friend Binary Bob proposes the following code:

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

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

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



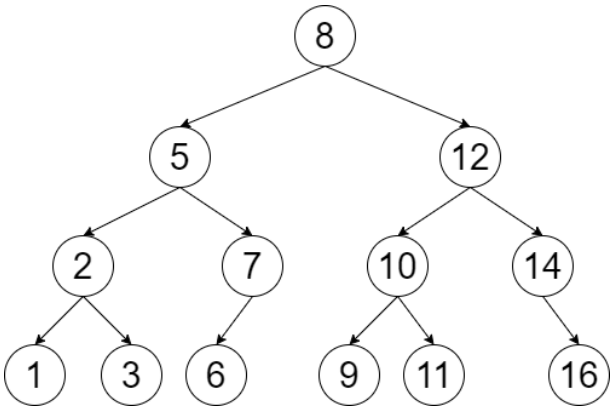
Question 3.3. Perform the following operations on a binary tree and draw each intermediate tree.

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

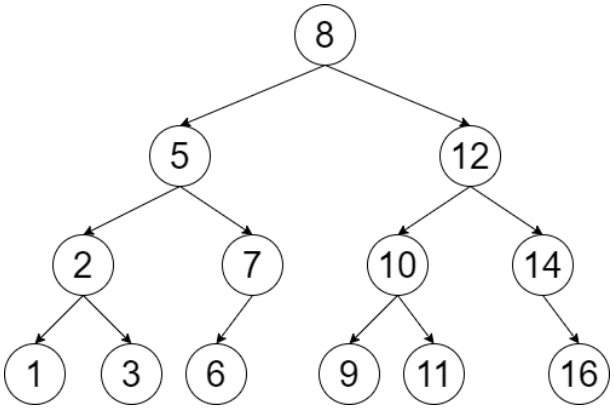
Question 3.4. What is the runtime for insert and delete updates, search, minimum, maximum, next-smaller, and next-larger queries? State the runtime in terms of h, where h denotes the height of the BST.

Insert	Delete	Search	Minimum	Maximum	Next-smaller	Next-larger

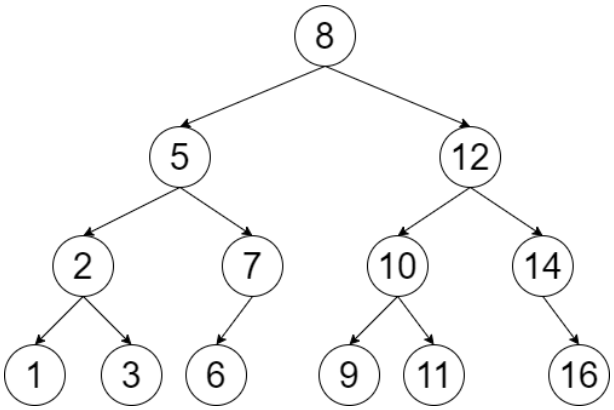
Question 3.5. Given the following BST, mark the succession of nodes you would visit in order to find the next-smaller of 15, and then describe the generic algorithm that you would follow for an arbitrary BST T and query q.



Question 3.6. 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.

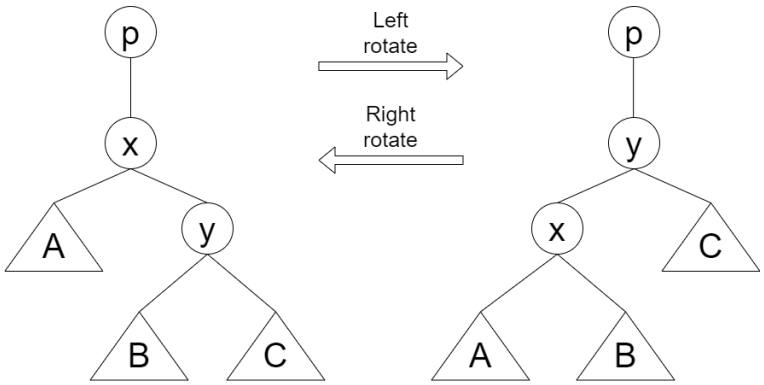


Question 3.7. Given the same BST, mark the succession of nodes you would visit in order to find the minimum element, and then describe the generic algorithm that you would follow for any given BST.

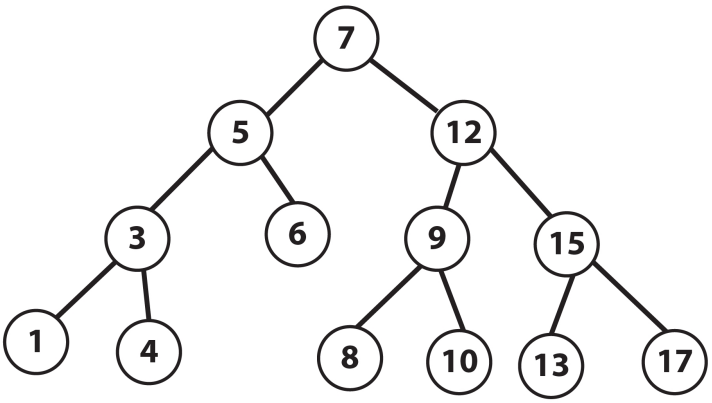


3.1 Rotations on BSTs

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



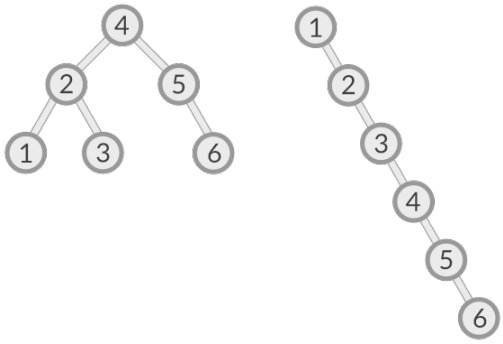
Question 3.8. Let `rotate(direction, child)` be a function that can be called on a node to rotate its child. Consider the following binary tree below. Let T be the root node. `T.rotate("R","L")`. Draw the tree after the operation is called.


7
5
12
3
6
9
15
1
4
8
10
13
17

4 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 4: 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)*

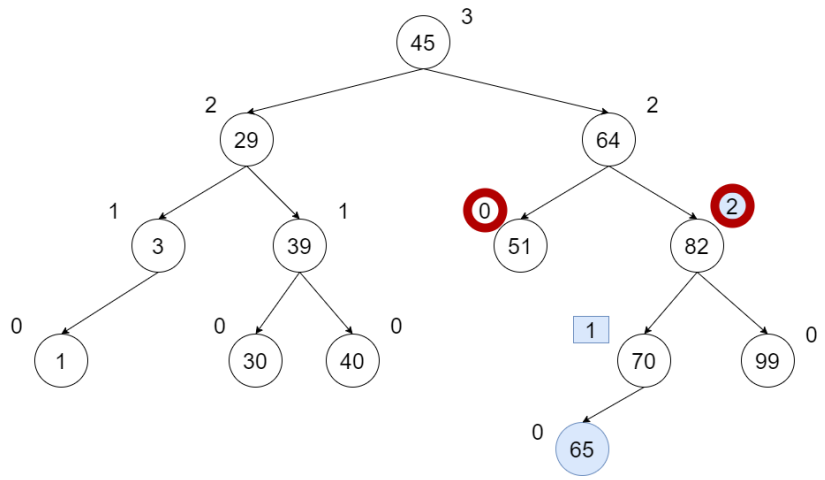


In class, we proved that we can perform a single rotation in $O(1)$ time, and stated (without proof) that $O(\log n)$ rotations suffice to restore balance to an AVL tree in which one key has just been inserted. In fact, the same is true for deletion, but we'll first need to cover deletion in (unbalanced)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

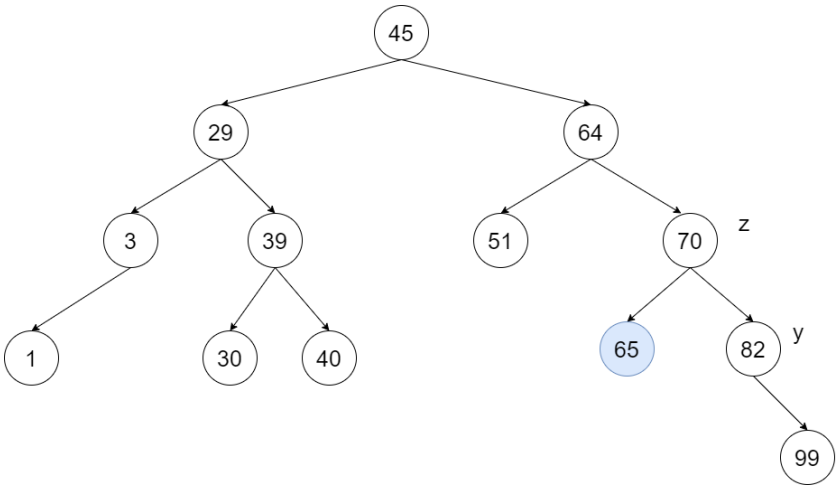
Question 4.1. Given a tree as an input, write a program to determine if it is a valid AVL Tree. Implement `is_avl_tree(T: BinaryTree)`.

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

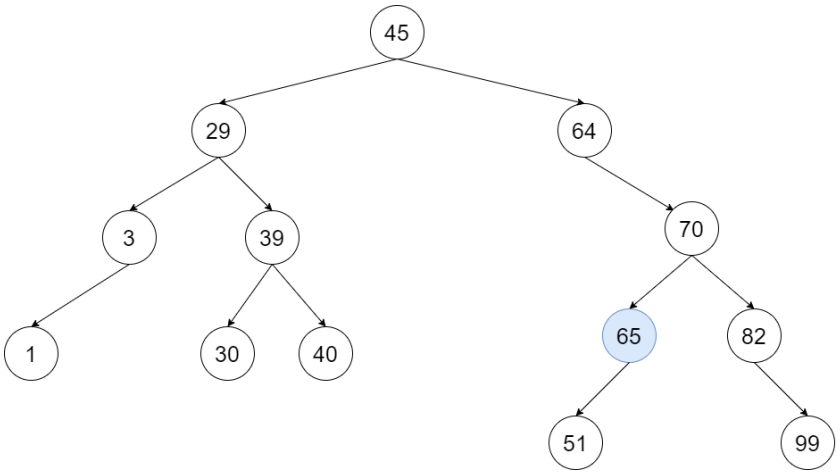
Question 4.2. What's wrong with the following attempted sequence of rotations to restore the AVL property of a tree?



To fix this, we then use AVL rotations. Let y be the vertex of key 82, and to the above tree we apply the operation `Left.rotate(y)`.



But the above tree still does not preserve the AVL property, so next we apply the operation `Left.rotate(z)`.


45
29
64
3
39
70
1
30
40
65
82
51
99