CSC111 Lecture 10: Binary Search Tree Deletion and Running-Time Analysis

Hisbaan Noorani

February 10, 2021

Contents

1 Exercise 1: Implementing `BinarySearchTree._delete_root`

We saw in lecture that we can implement the `BinarySearchTree.remove` method as follows, using a helper to do the "hard" part:

```python
class BinarySearchTree:
    def remove(self, item: Any) -> None:
        """Remove *one* occurrence of <item> from this BST.

        Do nothing if <item> is not in the BST.
        """
        if self.is_empty():
            pass
        elif item == self._root:
            self._delete_root()
        elif item < self._root:
            self._left.remove(item)
        else:
            self._right.remove(item)


    def _delete_root(self) -> None:
        """Remove the root of this BST.

        Preconditions:
```

```
        - not self.is_empty()
    """
```

In this exercise, we'll lead you through the cases to develop an implementation of the `BinarySearchTree._delete`
in a similar fashion to what we did in lecture for `Tree._delete_root` last week.

1.1 Case 1: `self` is a leaf

Suppose `self` is a leaf. Implement this case below, by filling both:

1. The if condition to check whether `self` is a leaf.
2. The if branch to update `self`'s instance attributes to delete the root.

```python
class BinarySearchTree:
    def _delete_root(self) -> None:
        # Case 1: this BST is a leaf
        if self._left.is_empty() and self._right).is_empty():
            self._root = None
            self._left = None
            self._right = None
```

1.2 Case 2: exactly one of `self`'s subtrees are empty

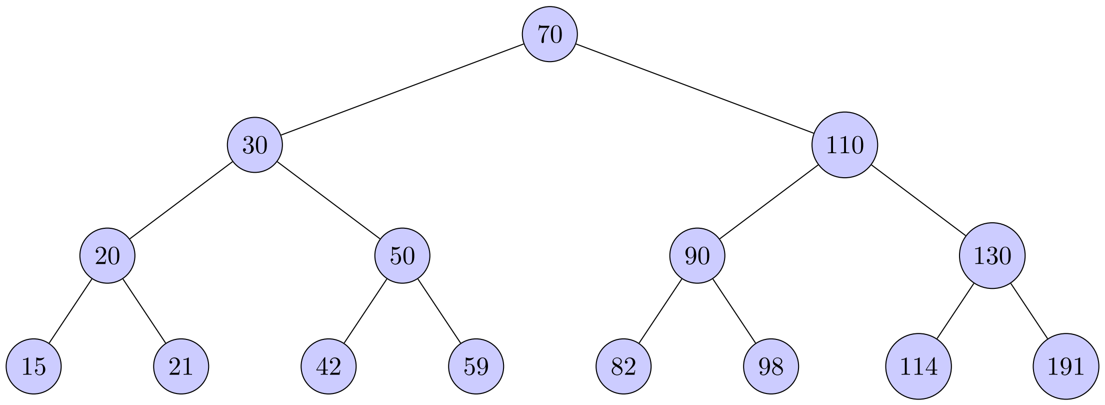
Suppose we want to delete the root of a BST where one of the subtrees is empty. The simplest approach is to "promote the non-empty subtree", similar to the technique from last week. Review this idea, and then fill in the conditions and implementations of each `elif`.

Hint: this is actually easier than what we did trees last week, since you can just reassign all three instance attributes—no need to `list.extend` the list of subtrees.

```python
class BinarySearchTree:
    def _delete_root(self) -> None:
        # Case 1: this BST is a leaf
        if self._left.is_empty() and self._right).is_empty():
            self._root = None
            self._left = None
            self._right = None
        # Case 2a: empty left subtree, non-empty right subtree
        elif self._left.is_empty():
            self._root, self._left, self._right =\
                self._right._root, self._right._left, self._right._right
        # Case 2b: non-empty left subtree, empty right subtree
        elif self._right.is_empty():
            self._root, self._left, self._right =\
                self._left._root, self._left._left, self._left._right
```

1.3 Case 3: both subtrees are non-empty

Consdier the following BST, whose left and right subtrees are *both* non-empty.



1. Suppose we want to delete the root 70 by replacing it with a value from one of its subtrees. Which possible values we could use to replace the root and maintain the BST property? (*This should be review from yesterday.*

2. Since there are two possible values, you have a choice about which one you want to pick. Complete the final case of `BinarySearchTree._delete_root`, using a helper method to extract your desired value.

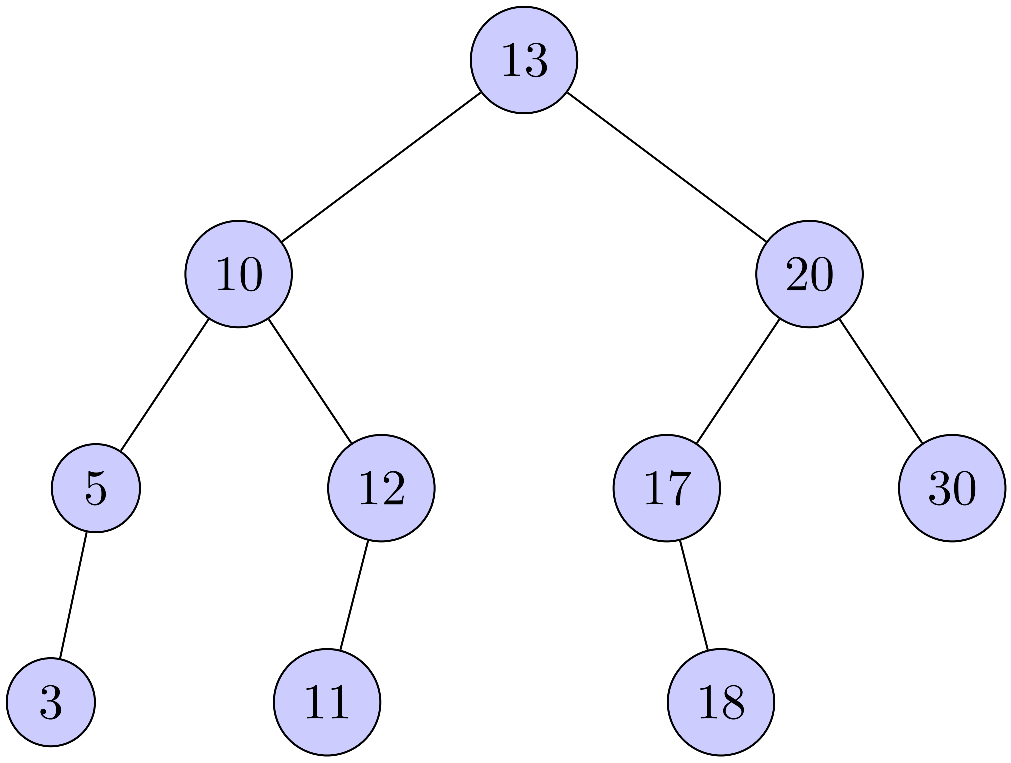
```python
class BinarySearchTree:
    def _delete_root(self) -> None:
        # Case 1: this BST is a leaf
        if self._left.is_empty() and self._right).is_empty():
            self._root = None
            self._left = None
            self._right = None
        # Case 2a: empty left subtree, non-empty right subtree
        elif self._left.is_empty():
            self._root, self._left, self._right =\
                self._right._root, self._right._left, self._right._right
        # Case 2b: non-empty left subtree, empty right subtree
        elif self._right.is_empty():
            self._root, self._left, self._right =\
                self._left._root, self._left._left, self._left._right
        # Case 3: non-empty left and right subtrees
        else:
            # if we take the max from the left subtree, it will be a good contender for replac
            self._root = self._left.extract_max()

    # For practice, implement the same method but with extract min instead of max
    def _extract_max(self) -> Any:
        if self._right.is_empty():
            max_item = self._root

            # Promote the left subtree
            self._root, self._left, self._right =\
                self._right._root, self._right._left, self._right._right

            return max_item
        else:
            return self._right._extract_max()
```

3. Check your assumptions: did you assume that the value you were extracting is a leaf? Consider deleting the root of the following tree:



Notice that if your chosen value is *not* a leaf, one of its subtrees must be empty—you can use the "promote a subtree" strategy in this case!

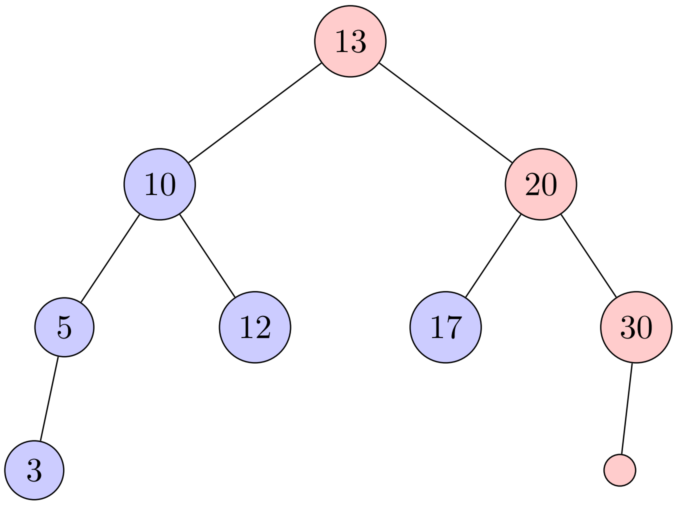
2 Exercise 2: Efficiency of `BinarySearchTree.__contains__`

Recall our implementation of `BinarySearchTree.__contains__`:

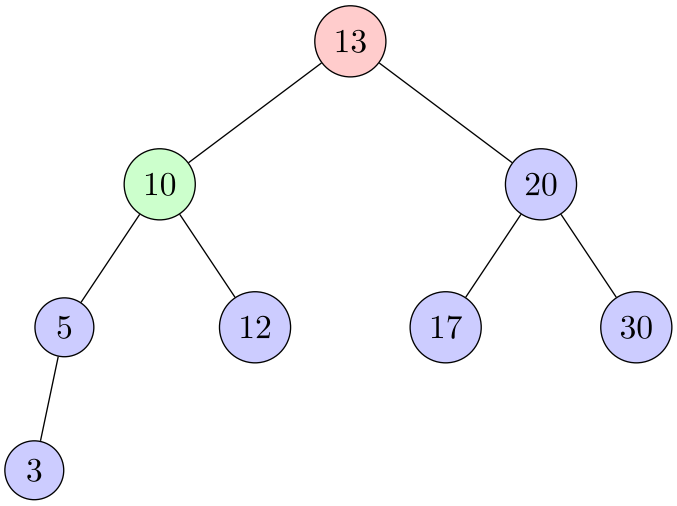
```python
class BinarySearchTree:
    def __contains__(self, item: Any) -> bool:
        if self.is_empty():
            return False
        elif item == self._root:
            return True
        elif item < self._root:
            return self._left.__contains__(item)
        else:
            return self._right.__contains__(item)
```

Crucially, after comparing the item against the root of the BST, only *one* subtree is recursed into. To make sure you understand this idea, suppose we have the following BST, and perform three searches, for the items `25`, `10`, and `4`, respectively. Circle all the values that are compared against the target item when we do each search. *Also, draw a new circle for any empty subtree where a recursive call is made.*

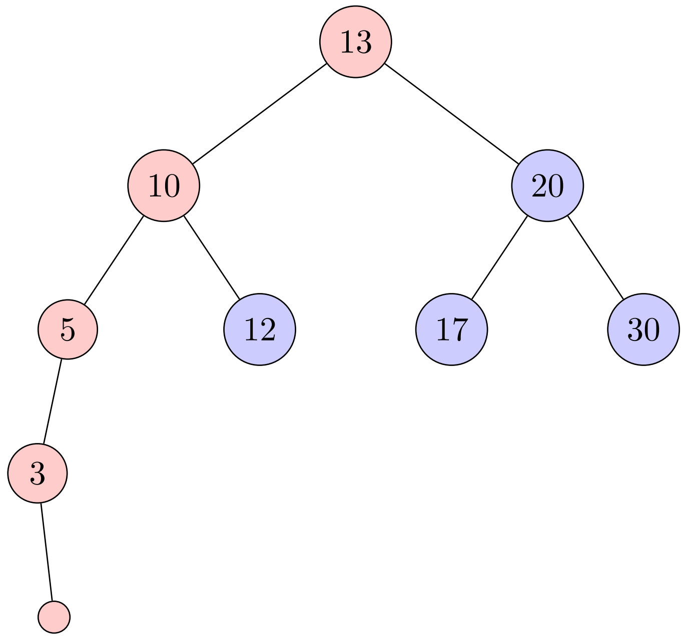
1. Search for `25`:



2. Search for `10`:



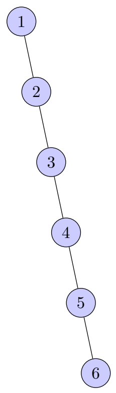
1. Search for `4`:



3 Exercise 3: Investigating the height of binary search trees

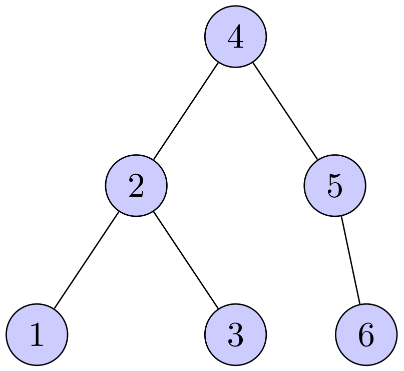
1. Recall that the *BST insertion algorithm* always inserts a new item into an "empty spot" at the bottom of a BST, so that the new item is a leaf of the tree.

   (a) Suppose we start with an empty BST, and then insert the items `1, 2, 3, 4, 5, 6` into the BST, in that order. Draw the final BST.



Height: 5

(b) Suppose we start with an empty BST, and then insert the items `4, 2, 3, 5, 1, 6` into the BST, in that order. Draw the final BST.



Height: 2

2. Write down the *height* of each BST you drew in the previous question. (You can do so beside or underneath the diagram.)

3. Let $n \in \mathbb{N}$ and suppose we want a BST that contains the values `1, 2, ..., n`. What is the *maximum* possible height of the BST?

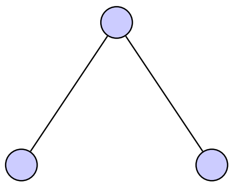
   $n - 1$ (as illustrated by 1a)

4. Now suppose we want to find the *minimum* possible height of the BST. We're going to investigate this in an indirect way: for every height h, we're going to investigate what the maximum number of values can be stored in a binary tree of height n. Let's try to find a pattern.

   (a) Suppose we have a BST of height **0**. What's the maximum possible number of values in the BST? (Hint: there's only one kind of BST with height 0.)

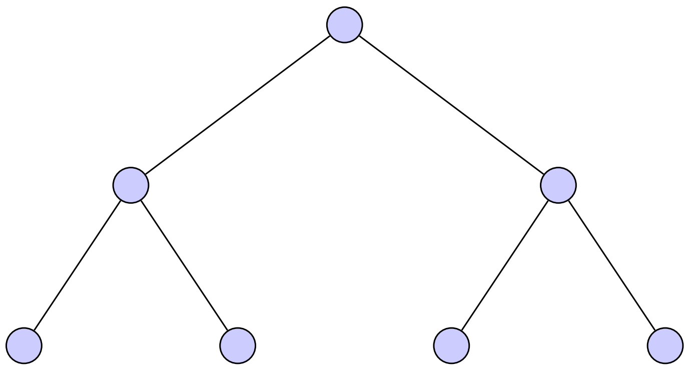
   Max Values: $1 = 2^0 = 2^1 - 1$

   (b) Suppose we have a BST of height **1**. What's the maximum possible number of values in the BST? (Draw an example.)



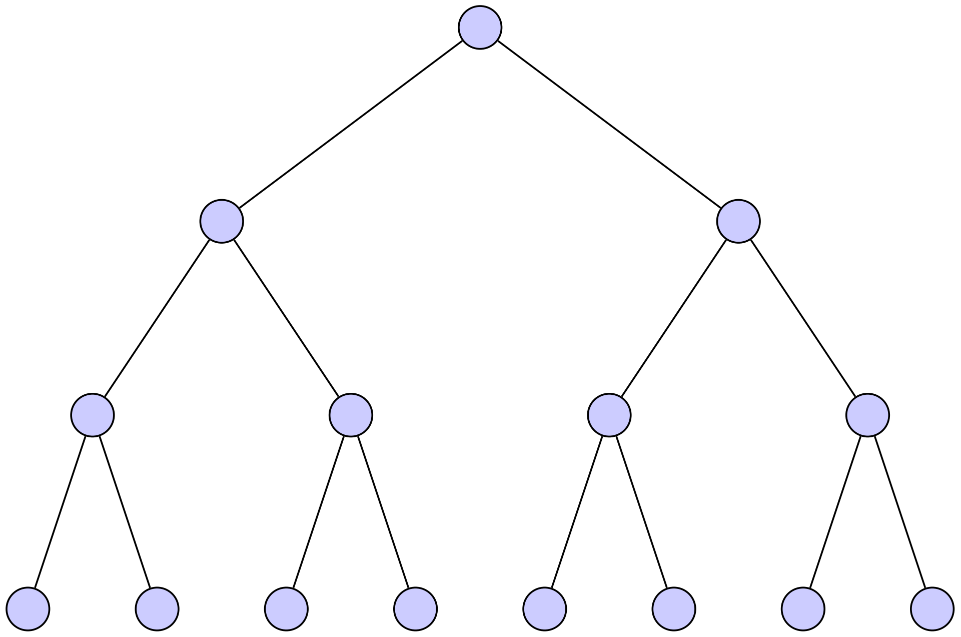
   Max Values: $3 = 2^0 + 2^1 = 2^2 - 1$

(c) Suppose we have a BST of height **2**. What's the maximum possible number of values in the BST? (Draw an example.)



Max Values: $7 = 2^0 + 2^1 + 2^2 = 2^3 - 1$

(d) Repeat for height 3.



Max Values: $15 = 2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$

(e) Suppose we have a BST of height **111**. What's the maximum possible number of values in the BST? (*Don't* draw an example, but find a pattern from your previous answers.)

Max Values: $2^0 + 2^1 + 2^2 + \cdots + 2^{110} = \sum_{i=0}^{110} 2^i = 2^{111} - 1$

(f) Suppose we have a BST of height h and that contains n values. Write down an inequality of the form $n \leq \ldots$ to relate n and h. (This is a generalization of your work so far.)

$n \leq 2^h - 1$

(g) Finally, take your inequality from the previous part and isolate h. This produces the answer to our original question: what is the minimum height of a BST with n values?

$$\begin{aligned} n+1 &\leq 2^h \\ \log(n+1) &\leq h\log(2) \\ \frac{\log(n+1)}{\log(2)} &\leq h \\ h &\geq \frac{\log(n+1)}{\log(2)} \end{aligned}$$