Pre-Class Work 14: Binary Search Tree (BST)

Binary Search Intuition

□ Intuition

Imagine a wizard □ is thinking about a random integer in 1 to 10 inclusively and asking you to guess that number. If you guess it wrong, he'll tell you if your guess is too large or too small. The less number of tries you do, the more extra credits you will get at the end of the quarter. How would you approach this problem in order to get the most extra credits by using the fewest number of tries?

□ Iterative Search

Suppose that I don't care about the extra credits. I just want to guess the right number and finish the game regardless of the number of tries. I will start guessing from 1, 2, 3, ... until the wizard tells me I've guessed the number correctly. However, what if the number is 10 and we **do** care about extra credits? An iterative search would result in the worst-case scenario -- it'll take me 10 tries to get the number correctly!

□ Binary Search

What if instead, we started guessing from **5**? Suppose that the wizard tells me "Hmm.. it's too low." I can now consider the range **from 6 to 10** and get rid of **half** of the original candidates to consider.

Then, we use a similar strategy to guess **8**. The wizard tells me "it's still too low." Now, I can just consider the range **from 9 to 10** by getting rid of half of the original candidates.

Notice that at each step, I'm always guessing the middle number of the remaining range, based off of the information from the wizard. This method of guessing is called **Binary Search**, and it is a core algorithm in computer science!

Observe that with iterative search, when the wizard said we got the wrong number, we only eliminated one number from our pool of possible guesses. With binary search, we got rid of half of the remaining numbers, which allows us to find the correct answer much quicker!

Main Points

- If we care about the time we spend (or credits we spend) trying to search for something, an **iterative search** would result in the *worst-case scenario*.
    - We would have to go through *all* options!
- **Binary search** can only work if we have immediate information for which range is left to consider.
- **Binary search** takes advantage of lists of items which are **sorted** -- the idea of something being "too low" or "too high" is what allows binary search to be so efficient!
- An example of **binary search** would be us guessing a number in between 1 and 10. If we guess 5, and the wizard says our guess is too low, then we only have to search through numbers 6-10 now rather than all 10 numbers!

Binary Search Tree (BST)

🌲 Tree Review

Properties

- Trees are recursively defined in terms of their nodes.
- Nodes can be:
    - null (an "empty node")
    - A node with a piece of data, and a left and right child
        - The children of a non-null node are **subtrees** which can either be empty, or a non-empty tree with left/right subtrees (and so on)
- Each node has a parent
- Each non-empty node stores a piece of data

🌲 Binary Search Tree (BST)

BSTs are like regular trees in that they provide a tree-like structure to represent some data. In addition to being a regular tree, BSTs are special because they have an **extra property/rule** that needs to be preserved:

- For **any given node in the tree (including the root):**
    - All nodes in the node's left subtree should have smaller values than the `root`
    - All nodes in the node's right subtree should have larger values than the `root`

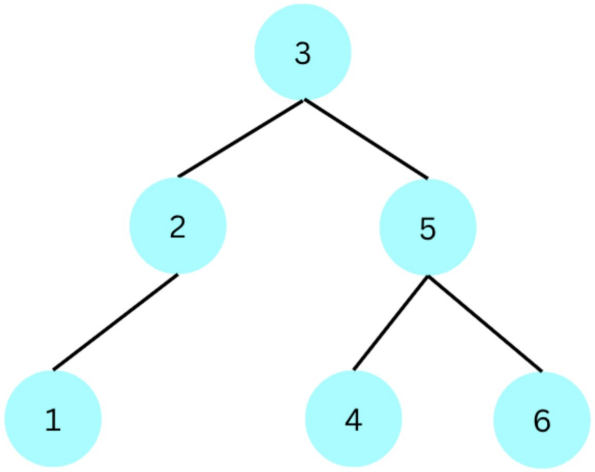
Notice that the above property holds for **all nodes** in the tree, i.e., all subtrees are also binary search trees.

🤔 Why does this property actually matter?

With this special property, binary search becomes possible! Imagine we wanted to see if the number 6 was inside a BST. At the beginning, we can compare some 6 with the root's `data` field and immediately get the information whether this `data` is too large or too small. If the `data` matches 6, then we're done!

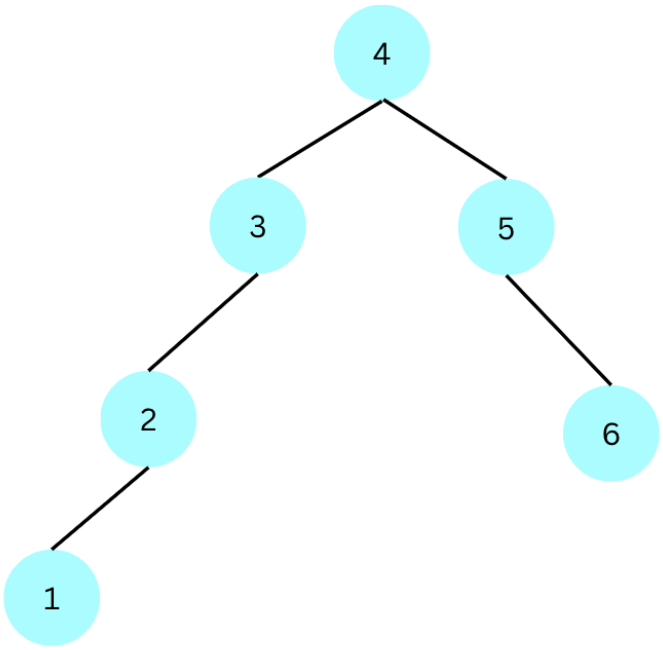
However, if it's too large, then we should look for the number 6 in the left subtree, because *all nodes in the left subtree have smaller values than 6.* If it's too small, we would like to go to the right subtree

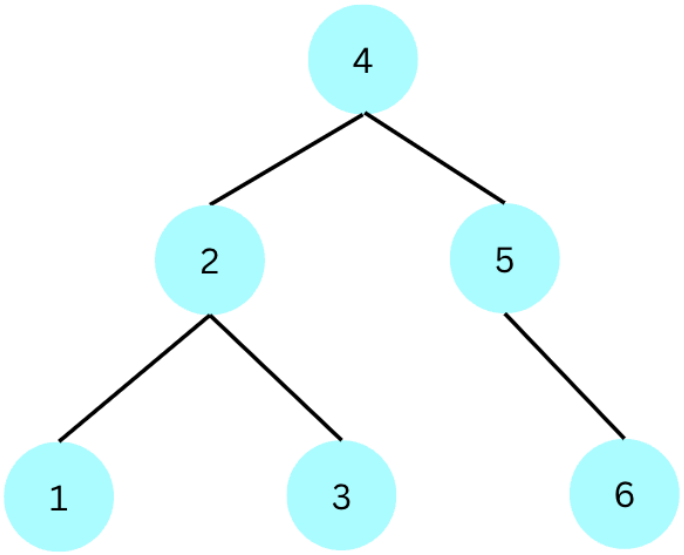
because *all nodes in the right subtree would have larger values than 6.* (This is the example you can see below)

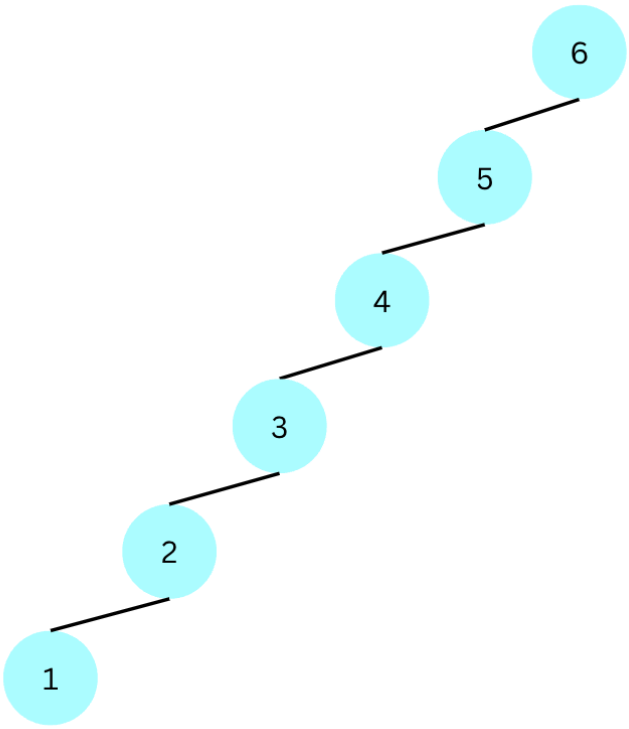


🌳 Binary Tree Zoo

Binary Search Trees can look very different, even though they store the exact same set of numbers! The following visualizations are all valid BSTs that represent a list of integers `[1, 2, 3, 4, 5, 6]`.







Notice that the last one looks like a linked list, but it also a tree! Furthermore, this fulfills all properties of BST so it would be considered a BST!

Main Points

- Trees are recursively defined structures with nodes.
  - Nodes can be null (an "empty node") or contain data and have left and right children.
  - Nodes have parents, and non-empty nodes store data.
- A **binary search tree** is a tree with an **extra property**: for any given node in the tree, all nodes in its left subtree have smaller values than the node, and all nodes in its right subtree

have larger values than the node.

- **BST**s are awesome for **binary search**! When we search for a value, we can compare it with the root node and from there we can decide whether to go left or right.
- We can visually represent **BST**s in a variety of ways, even with a linked-list-like structure!

x = change(x)

`x = change(x)` is a common pattern to recursively modify a tree. From the binary tree section, we learned how to traverse through a tree recursively and read the values from the nodes in order. Modifying a node is trickier and uses recursion with the `x = change(x)` pattern.

✏ Delete a node from BST

Consider the following operation: we want to delete the leaf node in a binary tree that contains the value `6`. Intuitively, we might consider traversing through the tree and check if the node we are looking at is a leaf node and check if the value is `6`.

The code might look like:

```
public void intuitiveDelete6(IntTreeNode root) {
    if (root != null) {
        // Check if it's leaf node
        if (root.left == null && root.right == null) {
            // Check if it's value is 6
            if (root.data == 6) {
                // TODO: delete this node
            }
        } else {
            // root has a left child
            if (root.left != null) {
                intuitiveDelete6(root.left);
            }
            // root has a right child
            if (root.right != null) {
                intuitiveDelete6(root.right);
            }
        }
    }
}
```

However, we get stuck @ line 7 because we don't have access to the parent of this nodein order to delete this node. You can also think of it as when deleting a node from a `LinkedList`, we usually need to access the previous node and change its `next` field to ignore the `curr` node. Instead of preserving the parent node here in binary tree, we adopt the pattern `x = intuitiveDelete6(x)`.

We **let the `IntuitiveDelete6` return the root of the subtree after deleting the 6.** The trust that we put on the recursive call of `intuitiveDelete6` is that **whatever root we put in as the parameter, this method will return the root with its subtree after deleting possible 6 in it.**

Once we hit the leaf node with 6 in it, we would like to **return** `null` because `null` is deleting the node with value 6 in the subtree with `root` being the leaf node containing `6`.

Our Approach

To sum up the modification we need,

- (1) we change the **return** to be **a root of the subtree** and believe that this returned subtree will not contain a leaf node of value 6,
- (2) we change the **base case** to **return** `null` because the leaf node of value 6 will be null after the deletion,
- (3) we change the recursive calls to **modify** the left and right children of the current `root` with the pattern `x = change(x)`, and
- (4) **return** the modified root after we finish modifying its left and right children. The modified version looks like:

```
public IntTreeNode delete6(IntTreeNode root) {
    if (root != null) {
        // Check if it's leaf node
        if (root.left == null && root.right == null) {
            // Check if it's value is 6
            if (root.data == 6) {
                return null;
            } else {
                return root;
            }
        } else {
            // root has a left child
            if (root.left != null) {
                root.left = delete6(root.left);
            }
            // root has a right child
            if (root.right != null) {
                root.right = delete6(root.right);
            }
            // return the root after we modify it's left and right children
            return root;
        }
    }
    // note: the return root statements on lines 9 and 21 can technically be
    // removed as they will eventually exit out of their conditionals and return here
    return root;
}
```

✏ Insert a node into BST

Insertion is also a kind of modification. To insert a **leaf node** and preserve the ordering property of BST, we will adopt the pattern of `x = change(x)` and change the conditions according to the ordering property of BST.

Believe that `insert(root, value)` will insert a new node at a position that preserves the BST order. The only decision left to decide is that, given the current root, **how do you know which subtree to insert this new node considering its value?**

Suppose that this new node has a value of `7` and the current `root` we are having is `5`. Because of the BST property, we have to insert this node to the **right subtree** because `7 > 5`. Thus, we will modify the right subtree of the current root by `root.right = insert(root.right, value)`. Once we hit the case of `root = null`, it indicates that we find a place to `insert` this node without violating any BST property.

At this point, we can simply create a new node here with the `value` and **return** it for the parent to point to as their `left` or `right` subtree. The implementation looks like:

```
public IntTreeNode insert(IntTreeNode root, int value) {
    if (root == null) {
        return new IntTreeNode(value);
    } else {
        if (root.data > value) {
            // Should insert the left subtree
            root.left = insert(root.left, value);
        } else {
            // Should insert the right subtree
            root.right = insert(root.right, value);
        }
        // return the root after modification
        return root;
    }
}
```

□ Main Points

- To *delete* a node from a **BST** (in this case the node that contains the value 6) we used `x = intuitiveDelete6(x)`, which we developed from `x = change(x);`
    - We change the return value to be the root of the subtree without the target value.
    - The **base case** returns `null` because the leaf node with the target value should become `null` after deletion.
    - We modify the left and right children of the current root using the pattern `x = change(x)` and return the modified root after updating its left and right children.
- To *insert* a node into a **BST** (in this case a node that contains the value 7) while maintaining its ordering property, we also used the pattern `x = change(x);`

- Once we hit the case of `root = null`, we found a place to `insert` this node
- Otherwise, if the new node's value is less than the current root's value, we insert it into the left subtree.
- If the new node's value is greater, we insert it into the right subtree.
- We return the modified root of the subtree after insertion.