Com S 228 - Day 33 – start BSTs

A lot of work and research in CS has gone into the innocent-sounding question: What is the most efficient way to store a bunch of records, and look them up again?

When we use the word “record”, think of this as something like a bunch of data along with a unique “key” or identifier. For instance, a student record may have a name, address, transcript, list of overdue library books, etc and also a university ID number that can be used to look up a student’s records. The keys have to be unique: no two students have the same ID number, even if they have the same name. From our perspective, we care mainly about the keys. So we will be looking at the situation of either storing a set of unique keys, or possibly a set of *pairs*,

(key, value)

where “value” is the catch-all for some object containing all the rest of the data in the record. For the algorithms we’ll look at this week it makes no difference.

The case we’re looking at this week will be the on in which the keys themselves are ordered, that is, we need to be able to iterate over the keys in a particular order.

Suppose we take a data structure such as a linked list and use it to store our keys or key/value pairs in sorted order:

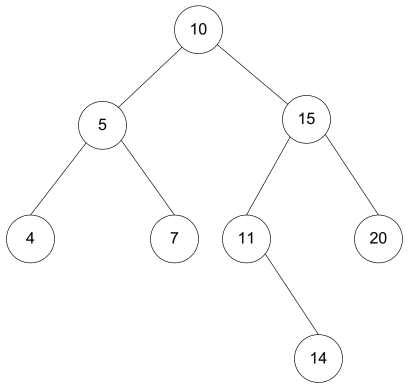
adding an element: $O(n)$
finding an element: $O(n)$
finding and removing an element: $O(n)$

Harrumph. If we use an array, we might be able to speed things up since we can use binary search. Then finding an element is $O(\log n)$. But we still have the overhead of $O(n)$ for adding and removing, because of having to shift elements.

adding an element: $O(n + \log n) = O(n)$
finding an element: $O(\log n)$
finding and removing an element: $O(n + \log n) = O(n)$

The motivation for a binary search tree is to get some of the advantages of a binary search, without the extra overhead of shifting elements around when using an array.

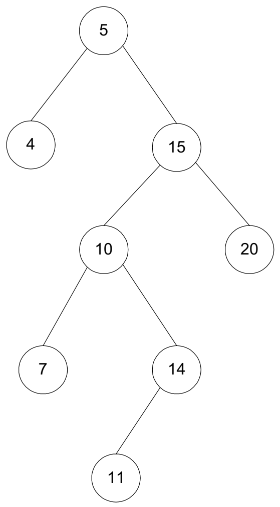
Here is an example

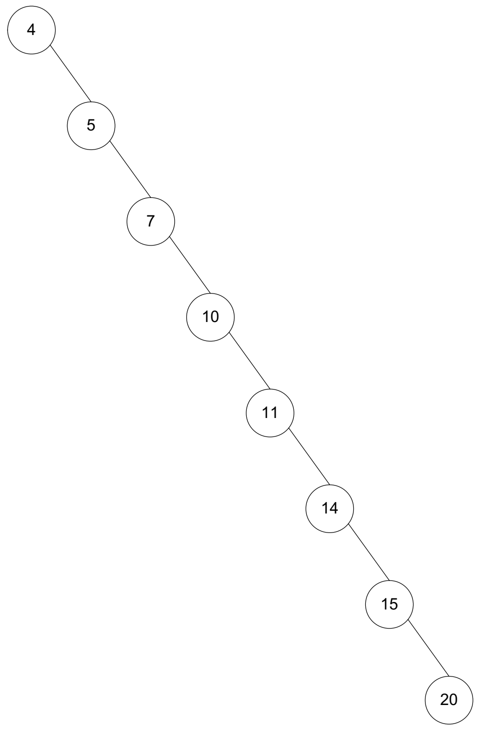


The definition of a BST is that it is a binary tree such that the following conditions hold for every node n:

- Every key in the left subtree of n is less than the key at n
- Every key in the right subtree of n is greater than the key at n

Note the definition ensures that there are no duplicate keys. One thing to notice is that for a given set of keys, there is more than one possible binary tree containing exactly those keys, for example





What do you get when you perform an inorder traversal on a BST? That gives you a simple mechanism for iterating in sorted order.

Let's start thinking about how to implement a BST. We can do this in the context of the Java Set interface, which is just a type of collection in which there are no duplicates. We will require the element type to be Comparable. For simplicity in implementing the interface, we'll extend the partial implementation in the Java libraries called AbstractSet.

```
public class BSTSet<E extends Comparable<? super E>> extends AbstractSet<E>
{
  protected Node root;
  protected int size;

  protected class Node
  {
    public Node left;
    public Node right;
    public Node parent;
    public E data;

    public Node(E key, Node parent)
    {
      this.data = key;
      this.parent = parent;
    }
  }
```

Notice that we have made Node an inner class, as we did for our implementations of List. Also note that in addition to the pointers to the left and right children, we have added a "parent" pointer. This will be needed to efficiently implement the removal and traversal operations.

To warm up, let's think about implementing contains(Object). We might imagine doing this with a helper method **`Node findEntry(E key)`**

We could do this using a recursive traversal as we did for ordinary binary trees, but it turns out that this problem is much simpler.) For example, take our first example tree and imagine you're searching for the key "12".)

The idea is, you can look at any node and you can immediately tell whether to look in the left subtree or the right subtree. It's really just like traversing a linked list, but with a conditional statement to decide which way to go.

```
protected Node findEntry(E key)  // returns null if not found
{
  Node current = root;
  while (current != null)
  {
    int comp = current.data.compareTo(key);
    if (comp == 0)
    {
      return current;
    }
    else if (comp > 0)
    {
      current = current.left;
    }
    else
    {
      current = current.right;
    }
  }
  return null;
}
```

Then to implement contains, we just want to say

```
public boolean contains(Object obj)
{
  return findEntry(obj) != null; // won't compile, since obj isn't type E
}
```

There is an annoying technical detail here, however. The API requires the argument to be Object, but we have to use type E within our **`find`** method, in order to apply the **`compareTo`** method. The solution is to add an unsafe cast. The cast is itself meaningless, but allows the code to compile. What happens at runtime if "obj" is not compatible with type E? The compareTo method will throw a ClassCastException. If we carefully read the documentation of the Set API, we find that throwing a ClassCastException is exactly the correct, documented behavior.

```
public boolean contains(Object obj)
{
  E key = (E) obj;
  return findEntry(key) != null;
}
```

The add() method is very similar to find(). How do you know where you can put the element in the tree, so as to preserve the BST property? Well, you look for it, and wherever you expected to find it, that's where it belongs! And of course if you do find it, then the add method should just return false – remember, there are no duplicates.

```
public boolean add(E key)
{
  if (root == null)
  {
    root = new Node(key, null);
    ++size;
    return true;
  }

  Node current = root;
  while (true)
  {
    int comp = current.data.compareTo(key);
    if (comp == 0)
    {
      // key is already in the tree
      return false;
    }
    else if (comp > 0)
    {
      if (current.left != null)
      {
        current = current.left;
      }
      else
      {
        current.left = new Node(key, current);
        ++size;
        return true;
      }
    }
    else
    {
      if (current.right != null)
      {
        current = current.right;
      }
      else
```

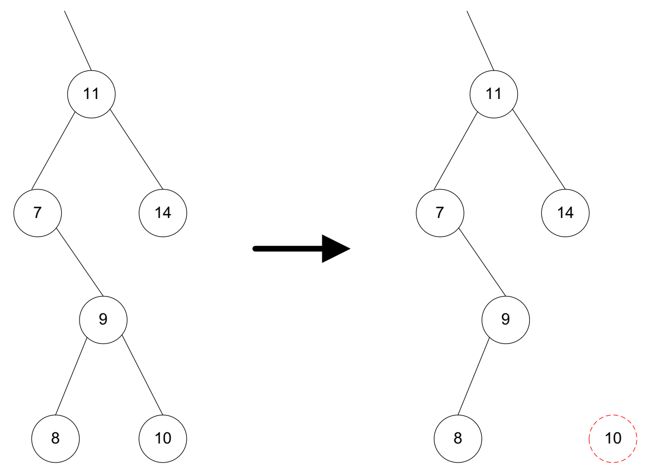
```
          {
            current.right = new Node(key, current);
            ++size;
            return true;
          }
        }
      }
    }
```

It is worth reflecting on whether we've accomplished anything. What is the complexity of the contains and add methods? If you look at the "degenerate" tree in our first set of illustrations, it's basically just a linked list, so it's clear that the *worst* case for **`add`** and **`contains`** is O(n). However, this only happens if the elements are inserted in sorted order. It is possible to prove that if the insertion order is random, then the expected (average) height of each node is O(log n). Since linking in a new node is O(1), that means both our **`contains`** and **`add`** methods are "typically" O(log n).

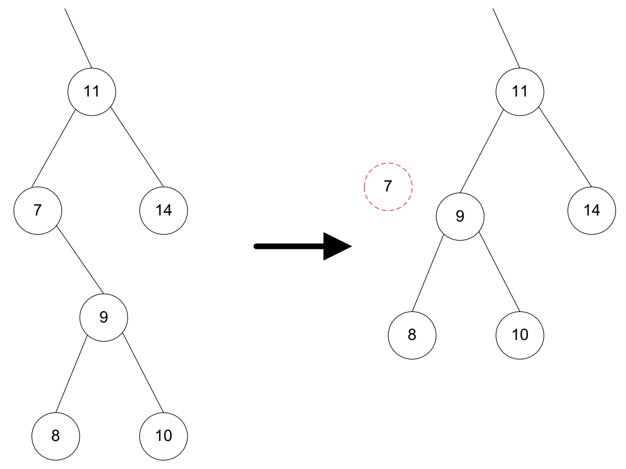
In practice, we tend to use BSTs with extra features to keep them from getting too far out of balance, so that we can get guaranteed O(log n) worst case or amortized behavior. For example, the Java library implementation of a sorted set is called a TreeSet. It is based on one of these sophisticated cousins of a BST, called a red-black tree.

An interesting point to notice is that in previous tree examples, we had a tree because the data itself was hierarchical. When using a BST, the data isn't "tree-like" at all, but we are using a tree to get some performance advantages in accessing the data.

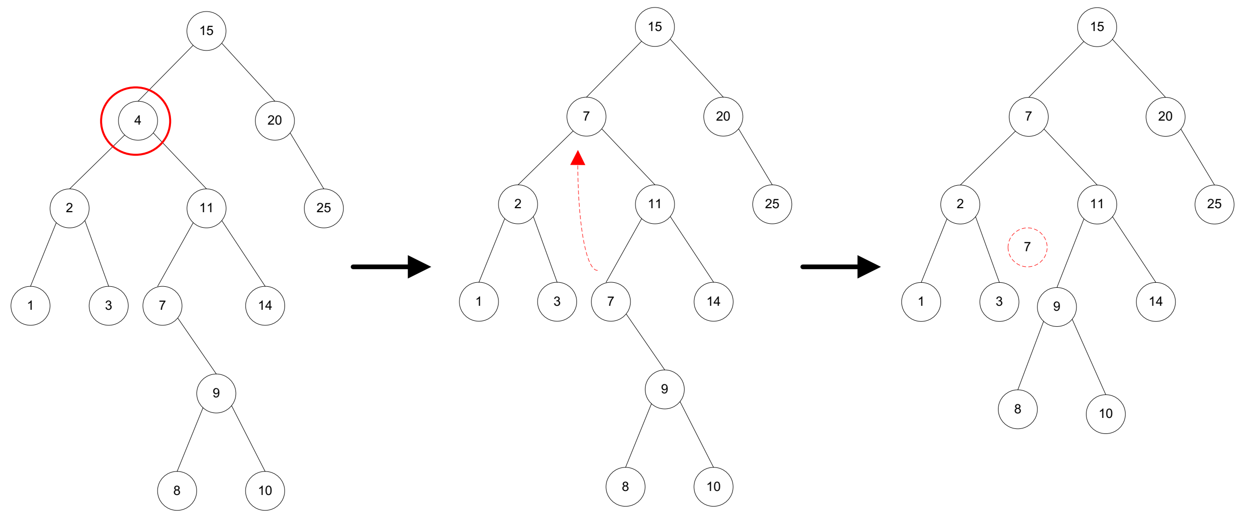
Removing nodes gets a bit more tricky. Let's look at a couple of examples. First, look at the case where the node has no children. We can just unlink the node, and that's it.



Removing a node with one child is not hard either. Just replace the deleted node with its child, and the BST property is preserved.



Let's think about what to do when removing a node with two children. Suppose in the tree below, we want to remove the node "4". If you just remove the node, it is not obvious what to do to fix the tree. If we try to put either of 4's children in its place, we end up having to move one of its children, and so on. We would like to avoid having to restructure the tree too much. Instead, let's try leaving the node where it is, and finding some other element in the tree that can go there without violating the BST property. There are two possible choices, the 3 or the 7. Notice that 3 is the predecessor of 4, and 7 is its successor, in the ordering of this tree. We'll (arbitrarily) pick the successor. So we copy the 7 up to where the 4 was. Now what? Well, the node with the 7 only has one child, so it is easy to delete. The thing is that *the node containing the successor can never have two children*. (Why?)



So, let’s assume for a moment that we have a successor method:

```
  // returns the successor of n, or null if there is no successor
  protected Node successor(Node n)
```

Then we can write a helper method to remove the contents of a given node. The idea is to first reduce the two-child case to the zero or one-child case.

```
  protected void unlinkNode(Node n)
  {
```

The idea is to first reduce the two-child case to the zero or one-child case. As in the example we copy the successor’s data into the given node, and then arrange to delete the successor, by assigning it to the parameter n.

```
    if (n.left != null && n.right != null)
    {
      Node s = successor(n);
      n.data = s.data;
      n = s; // causes s to be deleted in code below
    }
```

Now we know that n has at most one child, so we are going to delete n and replace it with the child (or possibly null). First figure out what the replacement node should be (could be left child, right child, or null).

```
    Node replacement = null;
    if (n.left != null)
    {
      replacement = n.left;
    }
    else if (n.right != null)
    {
      replacement = n.right;
    }
```

Then we link the replacement node to n's parent (which could be the root).

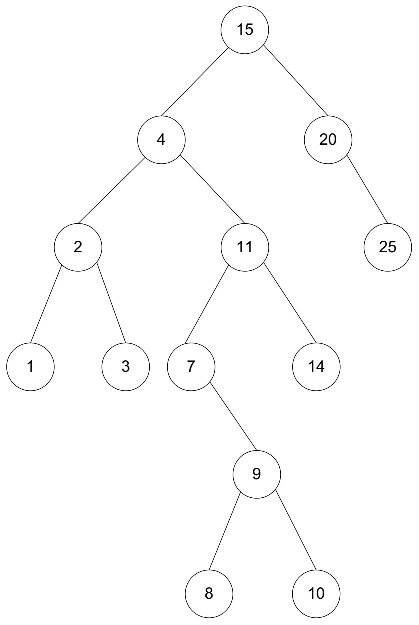
```
    if (n.parent == null)
    {
      root = replacement;
    }
    else
    {
      if (n == n.parent.left)
      {
        n.parent.left = replacement;
      }
      else
      {
        n.parent.right = replacement;
      }
    }

    if (replacement != null)
    {
      replacement.parent = n.parent;
    }

    --size;
  }
```

Writing a remove method is easy now, because we've done all the hard bits in findEntry and unlinkNode.

```
  public boolean remove(Object obj)
  {
    E key = (E) obj;
    Node n = findEntry(key);
    if (n == null)
    {
      return false;
    }
    unlinkNode(n);
    return true;
  }
```

Next: writing the successor method. Two cases:

1) The node has a right subtree. Go down to leftmost element in right subtree. (e.g. successor of 4 is 7).
2) The node doesn't have a right subtree. That means the node is the rightmost element of some subtree. If there actually is a successor, then it had better be the left subtree of somebody. Go up the tree until you find that somebody (e.g., successor of 10 is 11).



Next: writing the iterator.

Notice that we have to start out at the smallest, or leftmost, element of the tree. Implementing hasNext() and next() are easy, using the successor method. For remove, we can use the unlinkNode helper method. There is a little twist, though, because of what happens when we remove a node with two children. The node we're "removing" doesn't actually get removed, it gets the successor copied into it. Well, that successor is exactly what we want the next call to next() to return, so we have to reassign the cursor to point to that node.

Performance of the iterator? This is an interesting question, because the successor method is potentially O(log n) in a typical case and O(n) in the worst case. That suggests that iterating over n elements would be O(n^2) in the worst case. But we know that an inorder traversal is only O(n), and iterating with an iterator is essentially an inorder traversal. One way to think about it is to notice that in an iteration over all elements using the successor method, each link - left, right, and parent - is used exactly once. The total number of such links is 3n. So the total cost of iterating over the whole tree using the successor method must be O(n), even in the worst case.

We can say that the “amortized” cost of the successor method is O(1) per call, even though any particular invocation might be O(n).