CS 10, Winter 2017
Problem Solving via Object Oriented Programming
Hierarchies 2

- Binary search trees
- BST analysis
- Java notes

One very powerful use of trees is as a structure capturing the steps we would take in doing a binary search. Why is this useful? Recall binary search (we'll go over it in class in case you don't) — it quickly finds a key in a fixed-sized array by looking in the left portion or the right portion. But it works with a fixed array. A tree can more easily be dynamically modified, with data added and removed. It then supports efficient look-up.

All the code files for today: BST.java; InvalidKeyException.java

Slides from class

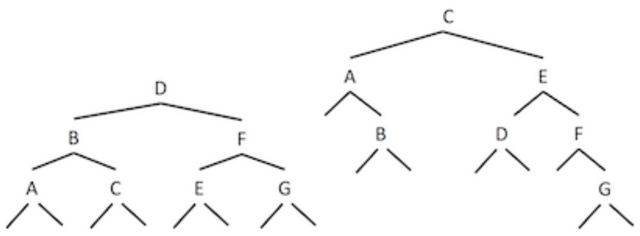
Binary search trees

A binary search tree (BST) is actually generic with two parameters: that of the key and that of the value.

```
public class BST<K,V> {
 private K key;
 private V value;
 private BST<K,V> left, right;
}
```

For example, we could have keys be Strings (e.g., names) and values be BufferedImages (their mug shots). One thing about generics: they need to be classes. So what do we do for primitive objects like ints and floats? Java actually provides corresponding "wrapper" classes, Integer and Float, that "wrap up" or "box" a primitive into an object. Java handles the conversion automatically, when it can infer what's going on ("autoboxing" and "unboxing"); we can also force it ourselves, e.g., by "`Integer i = new Integer(3);`". With this in mind, then, a BST could have as its keys Integers (e.g., zip codes) and the values also Integers (e.g., populations).

The key is what drives the BST — it's what we search by and how we organize the tree. The value just comes along for the ride. So I'll often just show the keys when drawing a BST, since any value could be associated with each key. Examples:



Note that these two trees have exactly the same keys, but different structures. The edges to nowhere represent null pointers, so the left tree has all leaves on the bottom row, while the right tree has a couple nodes ("A" and "F") with only one child, and a few other nodes with no children ("B", "D", and "G").

Both trees obey the **binary-search-tree property**, which is what defines a valid BST:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x, then the key in y is less than (or equal to) the key in x. If y is a node in the right subtree of x, then the key in y is greater than (or equal to) the key in x.

I put "or equal to" in parentheses, because some implementations do allow identicial keys, but in this class, we'll avoid that and assume each key is unique. In the left tree above, the key of the root is "D", the keys "A", "B", and "C", in its left subtree are less than that, and the keys "E", "F", and "G" in its right subtree are larger. The same property holds for every node in the tree. For example, the key "F" has a left child "E" that is smaller and a right child "G" that is larger.

Let's think about how a tree could be useful for searching. The BST property says that the keys in the left subtree are all less that those in the parent, and those in the right subtree are all greater than it. (We'll disallow equality.) So then how would we find the node with a given search key (or determine that it doesn't exist)? Compare to the node's key. If they're equal, we're done. If the search key is less than the node's key, recurse on the left; else recurse on the right.

BST.java has the class itself, and InvalidKeyException.java a special-purpose exception. (As with binary trees, this version is a bit simpler but not as powerful as the code from the book; read the book for more depth.) The algorithm just described is coded in find(). Our new exception type handles (and makes clear) the case where the key isn't in the tree.

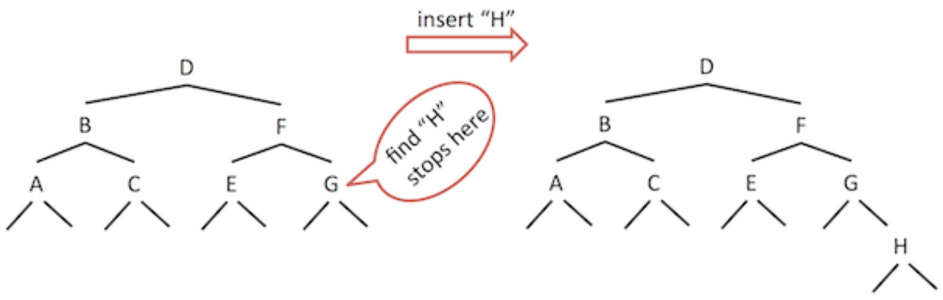
One tricky thing is that, as we discussed above, we have to be able to compare. But we don't know what kinds of things are in the tree — it's generic. If we knew the keys were ints, we could use < and >, but what do we do for generics? Java provides an interface `Comparable` that defines a method `compareTo` to test two objects. It returns 0 if they are "equal", a negative number if the one being asked is "less than" the parameter, and a positive number if the one being asked is "greater" (by whatever definition of equal/less/greater is appropriate). So we have to specify that it's not any old class for the key, but one that implements Comparable.

```
public class BST<K extends Comparable<K>, V> {
 private K key;
 private V value;
 private BST<K,V> left, right;
}
```

Thus we know we can rely on `K` to provide a `compareTo` method. Confusingly, we say "extends" even though it's an interface.

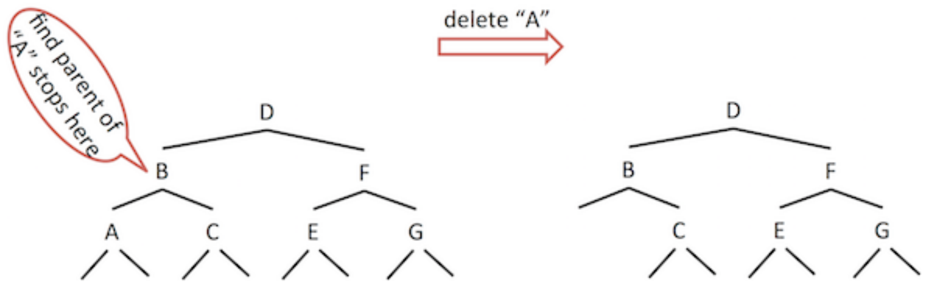
How would you find the minimum or maximum value in a BST? Where is it, and how do you get there? I provided a couple implementations, one recursive and one iterative. The reason we can use a loop here is that we only need to proceed to one child, not to both, so it's basically like looping down a linked list.

We can also readily modify a BST on the fly (compare to inserting into a sorted array for binary search, which might require pushing over a lot of entries to make room or to close a hole). Inserting is very similar to searching. If we find the key, we replace the value with the new one. If we reach a leaf, we add the new element there. That is where it belongs — any search for it in the future will follow the same path through the tree and will find it. We insert the element in the leaf and add two leaves below it. Example insertion:

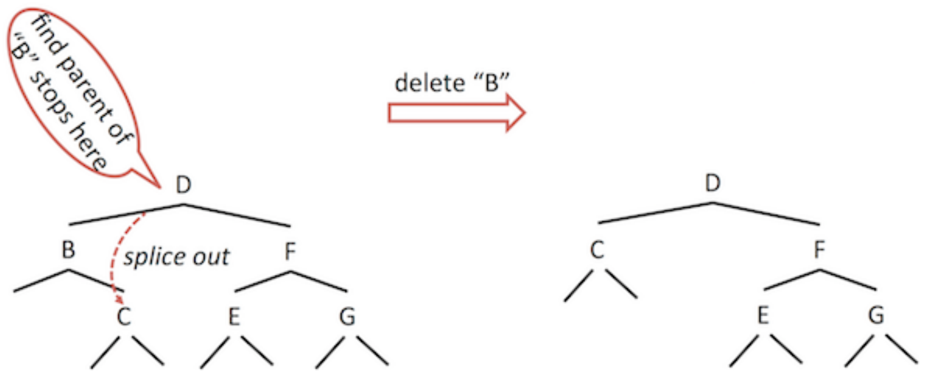


Deletion is a bit trickier. Consider a few possibilities:

no children
For example, to undo the "H" insertion we just did, read right to left in the figure, making the right child of "G" be null again. More generally, in the parent of whatever node we're deleting, null out the appropriate left/right child pointer. Another example: get rid of "A" by replacing it with null as the left child of "B".

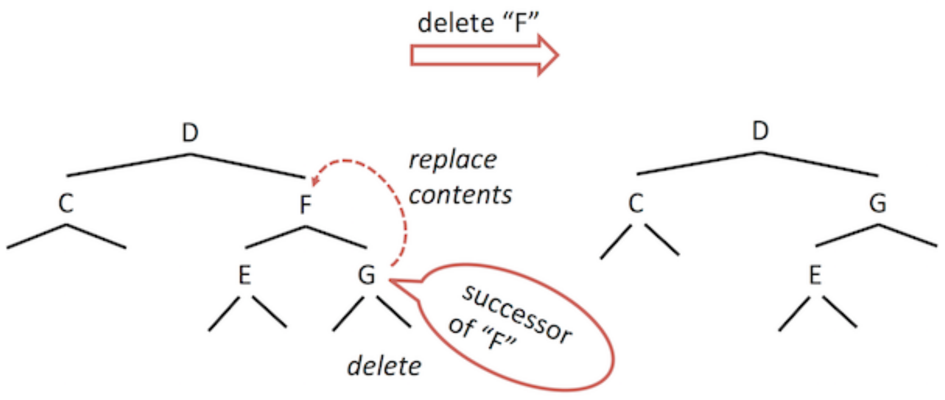


1 child
Same basic idea, in that we change the appropriate child pointer of the parent. But now we splice it out by setting the parent's child to be the child's child (feels like linked list, right?). E.g., get rid of "B" setting the left child of "D" to instead be "C" (the only child of "B").



2 children

We can't directly splice out a node with two children — that would leave a hole. So instead, find either the *predecessor* (largest on the left) or the *successor* (smallest on right). Delete that node from the tree, and use its contents in place of the 2-child node we want to delete. Note that by definition, the predecessor can't have a right child (else that child would be the predecessor), and similarly the successor can't have left child. Thus deletion is easy, from the case above. It's also correct, since the precedessor is larger than everything on the left side (since it's predecessor) but smaller than everything on the right side (since it's on the left), so can go above it all; similarly with the successor.

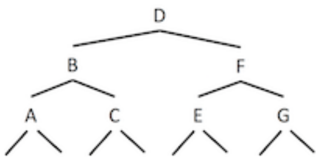


The code formulates this intuition in a recursive `delete` method that removes a key from a tree and returns an appropriate version of the tree. So if the key is to the left, replace the left child with whatever left.delete returns, and similar if it's on the right. If the key is here, then we consider the cases above. If the node has 0 children, return null. If the node has 1 child, return that child. (Note that these two cases are actually handled together, since a 0-child node has null as a child.) And if the node has 2 children, modify the node to use the successor's key & value and the updated right child with the successor deleted.

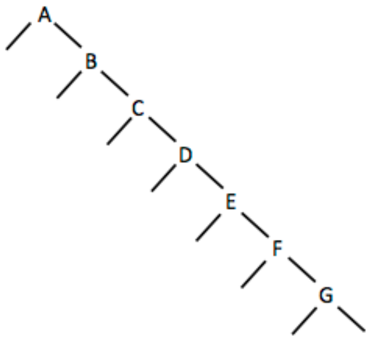
In general, a bunch of inserts and deletes can lead to an unbalanced tree — more nodes along one side than the other. In the worst case, we end up with a linear structure (a list represented as a tree) and get a vine instead of a tree. Sometimes we don't want to risk a bad run time when the data is not random. (Especially because a bad case is when the data is inserted in increasing or decreasing order!) To ensure balance requires more bookkeeping and occasional fix-ups; later we will see a type of BST called a red-black tree that handles that.

Binary search tree analysis

How long does it take to find something in a nicely balanced binary search tree?



How about a "vine"?



Java notes

primitive wrapping / boxing

To use a primitive piece of data (int, float, etc.) where we need an object (e.g., to go in an ArrayList, or in our BST), we use a wrapper class (Integer, Float, etc., Note the capital letter that starts each type) that boxes the data in an object. Java can automatically box and unbox for us in many cases.