Data Structures for Sets

by Kathi Fisler

1 Recap of Abstract Data Types (ADTs)

Last lecture, we introduced the idea of an abstract data type: a set of operations, properties, and behaviors (axioms) that together define a particular kind of data. We saw two examples of ADTs: Sets (collections of elements with no duplicates and no ordering), and Bags (collections of elements that allow duplicates but are not ordered). We showed that Java interfaces capture the operations, but the properties and behaviors do not have a direct representation in Java (much as types had no direct representation in Racket).

The word abstract in "abstract data type" is key: an ADT does not constrain how you organize the data or how you implement the operations. It only constrains which operations you provide and how they have to behave. If you need an actual Set for a program you are writing, you need some concrete data structure that implements the ADT.

In this lecture, we will discuss three concrete data structures that you can use to implement sets. Tomorrow, we will write the concrete code in Java for one of them.

1.1 Reminder: What are We Trying to Achieve in Code?

Remember our three motivating problems from last lecture: visitedURLs in a browser, words in a word game, and guests at a dinner party. If you wrote those programs, you would end up with code like

```
// the web browser program
class WebBrowser {
    ISet visitedURLs;

    // constructor
    WebBrowser() {
        this.visitedURLs = new _______ // start with an empty set of URLs
    }
}

// the word game program
class WordGame {
    ISet theWords;

    // constructor
    WordGame() {
        this.theWords = new _______ // make an empty set of words
    }
}
```

We need to be able to fill in the blank with the name of a Java class that provides the Set ADT (which we approximate by saying "implements the ISet interface"). We will have a concrete Java class to put into the blank by the end of the next lecture.

2 Implementing Sets with Lists

One obvious way to implement sets is to use lists. Even though we haven't yet covered lists in Java, you know about lists from some programming language you have used before. Roughly speaking, how would you implement sets with lists? Let's consider each operation:

- The code would have a variable of type list
- addElt would add the element to the list (i.e., cons in Racket, add or addFirst in Java, etc)
- remElt would require a method that traverses the list and removes the given element
- size would return the length of the list
- hasElt would traverse the list to check for the element

Seems straightforward, but what about the "no duplicates" property? Where do we enforce that? The answer might seem obvious: addElt could check whether the given element is in the list, and only modify the list if the element is not already there. There are other ways to do this though: the constraint on "no duplicates" is reflected in the behavior of the size method (or, as the axioms showed, the interaction between addElt and size). This raises a critical point:

As long as a program using your code never sees the duplicates, it is okay for your Set implementation to have them internally.

Stop and think about that. It's not intuitive, but it's essential for understanding how to implement ADTs (or any code for an external "client").

What might another implementation look like? Our addElt method could just add every element it receives, regardless of whether it is already there. Then, the size method would be responsible for returning the count of unique elements. This approach makes size expensive while leaving addElt cheap. Our original approach made addElt expensive. Which version you would want in practice depends on which operations you do often: if you add frequently but only rarely ask for the size, it might make more sense to use the second approach.

2.1 Performance Tradeoffs

This discussion raises one of the key motivations for looking at different data structures for the same problem: different approaches require different resources (time, space, private data, etc) to run. Computer science cares about the interplay between data organization and representation. This lecture starts you thinking about those issues.

2.2 Run-time Performance of Set Operations via Lists

What is the run-time performance when we use lists to implement sets? The easiest operation to think about is hasElt: to determine whether an element is in a list, we have to (potentially) traverse the entire list. The time to perform hasElt is therefore proportional to the length of the list. Formally, we say that the time required for hasElt is linear in the size of the list.

What about addElt? That depends on which implementation we use: if we add elements to the front without checking whether they are there, the time required is constant. If we add elements to the end of the list (the default add behavior for Java lists), the time required is linear (unless we do more clever programming). If we check whether each element is in the list before adding, the time required is linear (because we must run hasElt first).

Stop and think: what are the time requirements for remElt and size on lists?

remElt will also require you to traverse the list (unless you maintain a separate list of removed elements, which sometimes makes sense). A naive approach to size is linear (traverse the list), but typically we would maintain a separate variable with the size to save that cost - we'll come back to that approach later in the course.

For large sets, the linear time required to search can be prohibitive. We therefore would like some other data structure in which hasElt runs faster. How might we do this? Think about why hasElt is slow: as you traverse the list, you compare the current list element to the one you are looking for. If you did not find it, you throw away that item and go on to the next. On each comparison, you throw away at most one element, so you potentially have to check all of them. If you want better performance on hasElt, you need an approach that throws away more than one element after each (failed) comparison.

This sounds like a job for trees. If the data is placed in a tree in a useful way, we should be able to discard some portion of the tree (such as the left or right subtree) after each search. There are several tree-based data structures with different placements of data. We will contrast three in detail, using them to illustrate how we start to account for efficiency concerns in program design, as well as how to implement these in Java.

3 Implementing Sets with Binary Search Trees

To simplify the presentation, we will switch to sets of numbers, rather than sets of Strings (though what we present will work just fine for Strings—or other kinds of data—as well). Our starting point is therefore to assume that you have a binary tree with a number at each node and two subtrees (left and right).

A binary search tree (BST) is a binary tree in which every node has the following property:

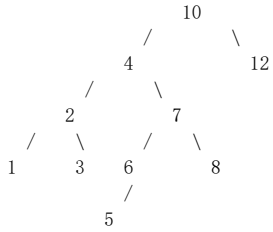
> Every element in the left subtree is smaller than the element at the root,
> every element in the right subtree is larger than the element at the root,
> and both the left and right subtrees are BSTs.

(this statement assumes no duplicates, but that is okay since we are modeling sets.) Wikipedia's entry on binary search trees has diagrams of trees with this property.

Constraints on all instances of a data structure are called invariants. Invariants are an essential part of program design. Often, we define a new data structure via an invariant on an existing data structure. You need to learn how to identify, state, and implement invariants.

3.1 Understanding BSTs

Consider the following BST (convince yourself that it meets the BST property):



As a way to get familiar with BSTs and to figure out how the addElt and other set operations work on BSTs, work out (by hand) the trees that should result from each of the following operations on the original tree: addElt(9), remElt(6), remElt(3), remElt(4).

With your examples in hand, let's describe how each of the set operations has to behave to maintain or exploit the BST invariant:

- size behaves as in a plain binary tree.
- hasElt optimizes on hasElt on a plain binary tree: if the element you're looking for is not in the root, the search recurs on only one of the left subtree (if the element to find is smaller than that in the root) or the right subtree (if the element to find is larger than that in the root).
- addElt always inserts new elements at a leaf in the tree. It starts from the root, moving to the left or right subtree as needed to maintain the invariant. When it hits a empty tree, addElt replaces it with a new node with the data to add and two empty subtrees.
- remElt traverses the BST until it finds the node with the element to remove at the root. If the node has no children, remElt returns the empty tree. If the node has only one child, remElt replaces it with its child node. If the node has two children, remElt replaces the value in the node with either the largest element in the left subtree or the smallest element in the right subtree; remElt then removes the moved node value from its subtree.

  The wikipedia entry also has diagrams illustrating this operation.

This description illustrates that any operation that modifies the data structure (here, addElt and remElt) must maintain the invariant. Operations that merely inspect the data structure (such as hasElt) are free to exploit the invariant, though the invariant does not necessarily affect all operations (such as size).

3.2 Run-time Performance of Set Operations via BSTs

Our discussion of hasElt on BSTs suggests that we've made progress on our performance problems with hasElt: on each comparison, we throw away one subtree, which is more than the single element we got to throw away on each comparison within lists.

Stop and think: do we always get to throw away more than one element in hasElt on a BST?

Consider the BST resulting from the following sequence of calls:

addElt(5), addElt(4), addElt(3), addElt(2), addElt(1).

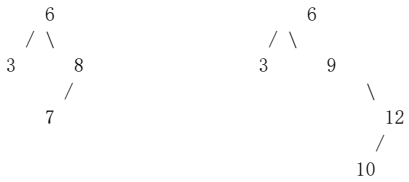
Draw the BST - what does it look like? It looks like a list. So we didn't actually gain any performance from hasElt in this case.

This illustrates one of the subtleties of arguing about run-time performance: we have to distinguish what happens in the best case, worst case, and average case. With lists, the performance of hasElt is linear in each of the best, worst, and average cases. With BSTs, the best case performance of hasElt is logarithmic (the mathematical term for "we throw away half each time"), but the worst case is still linear. Ideally, we would like a data structure in which the worst case performance of hasElt is also logarithmic.

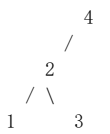
4 Implementing Sets with AVL Trees

BSTs have poor worst-case performance on hasElt because we cannot guarantee that half of the elements are in each subtree. If we had that guarantee, hasElt would have logarithmic worst-case performance. In other words, we need to constrain our trees to be balanced. There are several data structures for balanced BSTs: we will look at one of them.

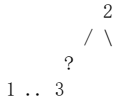
AVL trees are a form of balanced binary search trees (BBST). BBSTs augment the binary search tree invariant to require that the heights of the left and right subtrees at every node differ by at most one ("height" is the length of the longest path from the root to a leaf). For the two trees that follow, the one on the left is a BBST, but the one on the right is not (it is just a BST):



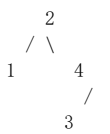
We already saw how to maintain the BST invariants, so we really just need to understand how to maintain balance. We'll look at this by way of examples. Consider the following BST. It is not balanced, since the left subtree has height 2 and the right has height 0:



How do we balance this tree efficiently? Efficiency will come from not structurally changing more of the tree than necessary. Note that this tree is heavy on the left. This suggests that we could "rotate" the tree to the right, by making 2 the new root:



When we do this, 4 rotates around to the right subtree of 2. But on the left, we have the trees rooted at 1 and 3 that both used to connect to 2. The 3 tree is on the wrong side of a BST rooted at 2. So we move the 3-tree to hang off of 4 and leave 1 as the left subtree of 2:

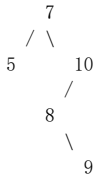


The 4 node must have an empty child, since one of its original children became the root of the new tree. We can always hang the former right subtree of the new root (2) off the old root (4).

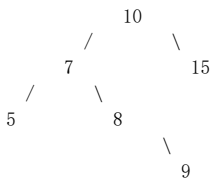
Even though the new tree has the same height as the original, the number of nodes in the two subtrees is closer together. This makes it more likely that we can add elements into either side of the tree without rebalancing later.

A similar rotation counterclockwise handles the case when the right subtree is taller than the left.

One more example:

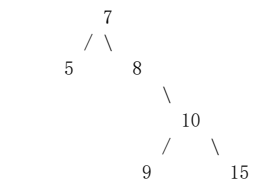


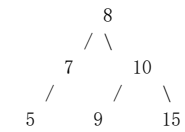
This tree is heavier on the right, so we rotate counterclockwise. This yields



which is no better than what we started with. Rotating clockwise would give us back the original tree. The problem here is that when we rotate counterclockwise, the left child of the right subtree (rooted at 8 in this example) moves to the left subtree (rooted at 7). If that left child is larger than its sibling on the right (rooted at 15), the tree can end up unbalanced.

The solution in this case is to first rotate the tree rooted at 10 (in the original tree) clockwise, then rotate the resulting whole tree (starting at 7) counterclockwise:





An astute reader would notice that the original tree in this example was not balanced, so maybe this case doesn't arise in practice. Not so fast. The original tree without the 9 is balanced. So we could start with the original tree sans 9, then addElt(9), and end up with a tree that needs a double rotation. You should convince yourself, however, that we never need more than two rotations if we had a balanced tree before adding or removing an element.

We tie the rotation algorithm into a BBST implementation by using it to rebalance the tree after every addElt or remElt call. The rebalance calls should be built into the addElt or remElt implementations.

Wikipedia has a good description of AVL trees. Refer to that for details on the rotation algorithm.