CMPSCI 187: Programming With Data Structures

Lecture #28: More on Binary Search Trees
David Mix Barrington
16 November 2012

More on Binary Search Trees

- Review of BST Definition
- Review of BST Implementation
- Comparing BST’s and Lists
- Balancing a BST: The Wrong Methods
- Balancing a BST: The Right Method
- Storing a Balanced Tree in an Array
- Introducing the Frequency Counter

Review of BST Definition

- A **binary search tree** is a tree of binary nodes, each with a value from some comparable class.

- The **BST rule** is that every node in the left subtree of x has a value *less than or equal to* that of x, and every node in the right subtree has a *greater than or equal* value. This means that the **inorder traversal** gives the elements in order.

- The **BST interface** gives the list operations of add, get, remove, and contains, plus three "iterators", one for each kind of order on the tree's nodes. These work by `reset` and `getNext`, unlike standard `java.util iterator` objects.

- The `BinarySearchTree` class implements the `BSTInterface` interface.

Review of BST Implementation

- Each node of the tree has three fields, `info`, `left`, and `right`.

- The tree has a root and three queues, one for each order type.

- Most of the methods are recursive, using a helper method to generalize a question about the whole tree into a question about a subtree, given as a parameter in the form of its root node.

- The observer methods `size`, `contains`, and `get` recurse from a node to the correct subtree until the desired content is found at a node.

- The transformers add and remove are more complicated. To add, we find a leaf where the new node will fit. Removing a childless or single-parent node is easy -- to remove a node with two children we find the node's inorder predecessor, move the content from it to the node, then remove it recursively. (It can't have a right child, so the recursion stops.)

Comparing BST’s and Lists

- It’s natural to compare the performance of BST’s and linear lists on the same tasks. There’s a problem, though, with our worst-case analysis. In the worst case, a BST essentially is a linear list, if every node has at most one child.

- Searching into a BST takes a worst-case time equal to the height of the tree. If the tree is **balanced**, this height is O(log n) for an n-node tree, but for an unbalanced tree it could be as large as O(n).

- DJW have a chart on page 579 comparing BST’s, array-based lists, and linked lists under the assumption that the BST’s are balanced. The only BST operation that takes more than O(log n) is `reset`, which copies all the info of the tree into a queue. The array-based list can do `get` in O(log n), but `add` and `remove` each take O(n) because of the need to move data. The linked list takes O(1) to process adding or removing, but O(n) to find where to do it.

Balancing a BST: The Wrong Methods

- How can we balance a tree? In CMPSCI 311 you will learn about self-balancing trees, in at least one of the several versions. These trees always have a height of O(log n) when their size is n. Adding and removing take O(log n) time, because there may be a restructuring step of O(log n) time after each such move to restore the balance.

- DJW consider only a simpler method of rebalancing that takes O(n) time whenever we choose to do it. This is to copy the entire tree into an array of elements, then add each element of the array in turn back into a new tree. We naturally form our array by using one of our traversals.

- If we use inorder, the array becomes a sorted list. But now if we copy in the most obvious way, we produce a very unbalanced tree. If we use preorder, it turns out that the new tree is identical to the old one. What happens if we use postorder?

Balancing a BST: The Right Method

- The right idea here is to study how we want the new tree to be arranged. Suppose we have used inorder to form the array, so that the array is sorted. If the tree is balanced, with an equal number of nodes on each side, the root node will be the middle element of the array. Its left child should be about 1/4 of the way along, and its right child 3/4 of the way, and so forth. This should suggest a recursion:

```
private void insertTree (low, high) {
// copies range from nodes[low] through nodes[high] into tree
   if (low == high) tree.add(nodes[low]);
   else if (low + 1 == high) {
      tree.add(nodes[low]); tree.add(nodes[high]);}
   else {int mid = (low + high)/2;
      tree.add(nodes[mid]);
      insertTree (low, mid - 1);
      insertTree (mid + 1, high);}}
```

Storing a Balanced Tree in an Array

- Remember that in general array-based structures will be faster to use than linked structures because of locality of memory -- the machine may be able to keep the entire array in fast memory at once.

- If our tree has a particular structure, we can store it in an array so that we don't need any explicit pointers at all. We declare that each node A[i] has a left child A[2i+1] and a right child A[2i+2]. So A[0] has children A[1] and A[2], A[1] has A[3] and A[4], and so forth. Finding a particular child of a particular node thus involves only arithmetic on the indices.

- We say that a binary tree is **full** if all its leaves are on the same level. A full binary tree of height h has exactly $2^{h+1} - 1$ nodes, exactly 2^h of which are leaves. DJW call a binary tree **complete** if it is either full or full through its next-to-last level, with the leaves on the last level left-justified.

- An array of length n, with our implicit pointers, becomes a complete binary tree with n nodes. We can embed an unbalanced tree into a larger one.

Introducing the Frequency Counter

- Next lecture we'll present DJW's case study for Chapter 8, which will also be the foundation of our Project 5. This is an application to determine **word frequencies** in a piece of text. The frequency of a word is the number of times it occurs in the text.
- We consider the words of the text to be consecutive strings of letters and/or digits, with a delimiter on each side. We'll define delimiters to be spaces, line breaks, and punctuation marks. So the text `"catch as catch can"` has two occurrences of the word `"catch"` but no occurrences of the word `"cat"`.
- Our application will read the text and build a BST with a node for each word that has occurred in the text, except that we will allow it to ignore short words. Each node will also keep track of the number of times the word has occurred. In Project 5 we'll compare their implementation with another one that you will write, using a priority queue in place of the BST.