CMPSCI 187: Programming With Data Structures

Lecture #29: More on Binary Search Trees
23 November 2011

More on Binary Search Trees

- Implementing an Ordered List
- Comparing Linked and Tree Implementations
- How to Keep a Tree Balanced
- Ordinary Right and Left Rotations
- Compound Rotations: RightLeft and LeftRight
- AVL Trees (Basic Idea Only)
- Red/Black Trees (Basic Idea Only)

Implementing an Ordered List

- This is actually pretty anticlimactic. For each of the adding, removing, and finding list operations we have a corresponding operation already in `LinkedBinarySearchTree<T>`. The operations `contains`, `isEmpty`, and `size` are just inherited directly from `LinkedBinarySearchTree<T>` or its ancestor classes with the same names, so they don't appear in the code.

```
public class BSTList<T> extends LinkedBinarySearchTree<T>
      implements ListADT<T>, OrderedListADT<T>, Iterable<T> {
   public BSTList ( ) {super( );}
   public void add (T element) {addElement (element);}
   public T removeFirst ( ) {return removeMin( );}
   public T removeLast ( ) {return removeMax( );}
   public T remove (T element) {return removeElement (element);}
   public T first ( ) {return findMin( );}
   public T last ( ) return findMax( );}
   public Iterator<T> iterator( ) {return iteratorInOrder( );}}
```

Comparing Linked and Tree Implementations

- The linked list implementation of an ordered list did some things in O(1) time: `removeFirst`, `first`, `isEmpty`, and `size`. (With a doubly linked list we could also perform `removeLast` and `last` in O(1) time.) The other operations were all O(n), because in the worst case they involved a linear search of the whole list.

- *If* our binary search tree is balanced (meaning that its height is O(log n)), then all the O(n) operations become O(log n) operations, because the searches now travel in the worst case from the root of the tree to a leaf. With the exception of `isEmpty` and `size`, which stay O(1) because they just look up the size field, the O(1) operations get slower in the worst case -- also O(log n).

- If we do nothing to maintain balance, our worst case is a **degenerate** binary tree, with just one branch and a height equal to the size. All the operations (again except for `isEmpty` and `size`) become O(n) time in the worst case.

How to Keep a Tree Balanced

- If our ordered list is not going to gain or lose any elements while we are using it, we can arrange our tree optimally. We make the exact median element the root, and make every other element have right and left subtrees as close to each other in size as possible. This tree will clearly have height O(log n).

- But inserts and deletes to the list can easily mess this up. If the left subtree gets much larger than the right subtree or vice versa, the larger tree will need leaves farther down than the other. Suppose, for example, we take the original tree, remove all the elements except those on the leftmost path, and then add new elements, each smaller than any seen before.

- We have to be prepared to change root nodes, since the root should be near the median. In fact, we may want to replace any node with one that more evenly splits the elements under it. But in the course of this we can't allow the BST rules to be broken. We need primitive operations to change the root of a subtree while keeping the BST rules in force.

Ordinary Right and Left Rotations

- Suppose A has children B and C and we would prefer B to A as the root of the subtree. A **right rotation** does this, as follows. Call B's children D and E, and C's children F and G. After the rotation, B is the root, A is its right child, and D is still its left child. E, and any descendants it has, must be in the right subtree under A, and in fact E can be the left child of A while C is its right child. (E and its family are all greater than B but less than A.)

- A **left rotation** on the same original tree makes C the new root, with left child A and right child still G. A's two children are now B on the left and F on the right.

- Each of these operations takes only O(1) time because only O(1) pointers need to be reset. The right rotation reduces the height of part of the left subtree (under D) by one and increases that of the whole right subtree by one. Of course the left rotation reduces part of the right and increases the left.

Compound Rotations: RightLeft and LeftRight

- But suppose we want to reduce the height of F in our example. Neither a simple right or left rotation does this. But if we first do a right rotation on the root of the right subtree (C) instead of on the root, and then do a left rotation on the root, we see that F has moved up two levels, and its children L and M have moved up one level each.

- Similarly, we can reduce the height of E by doing a left rotation on B followed by a right rotation on the root A. This makes E the new root, brings up its children J and K up from level 3 to level 2, and lowers the parts of the tree under C and D.

- The basic idea to maintain balance is simple -- when an insertion or deletion changes the height of a section of the tree, we do rotations to compensate. The various balanced tree schemes each take a different approach to measuring the balance of the tree and deciding which rotations to make.

AVL Trees (Basic Idea Only)

- The **AVL tree** was developed by two Russians (Adel'son-Vel'skii and Landis) in 1962.
- Each node knows the *difference in height* between its two subtrees -- insertions, deletions, and rotations must maintain this.
- The key property of the AVL tree is that the heights of the subtrees of any two *sibling* nodes differ by at most one.
- An insertion or deletion changes heights of subtrees by at most one, so we need instructions to deal with O(1) different cases, using O(1) rotations to restore the balance at each node within a difference of one.
- Because height of the tree remains O(log n), all operations take at most that time. The rotation sequences are O(1) at each of up to O(log n) levels.

Red-Black Trees (Basic Idea Only)

- As you will see in much more detail in CMPSCI 311, **red-black trees** are a more robust solution to the balanced BST problem, in that they can be extended to structures with other capabilities.
- In a red-black tree, each node has a color. The root is black, all children of a red node are black, and every path from the root to a leaf contains the same number of black nodes.
- The height of a tree with n nodes is at most 2 log n -- two extreme cases are an all black tree (height log n) and a tree with alternating red and black levels on one side and all black on the other (which has all but about sqrt(n) of its nodes on one side).
- The rotation rules are somewhat complex but understandable, with O(1) rotations to deal with an insertion or deletion in any possible color context.