CMPSCI 187: Programming With Data Structures

Lecture #26: Binary Search Trees
David Mix Barrington
9 November 2012

Binary Search Trees

- Why Binary Search Trees?
- Trees, Binary Trees and Vocabulary
- The BST Rule
- Traversals
- The `BSTInterface`
- An Application Example
- Beginning the Implementation

Why Binary Search Trees?

- We've seen that **binary search** is a powerful technique to search sorted arrays, finding an element in O(log n) time rather than O(n) for linear search. But inserting or deleting into arrays is in general O(n) due to shifting elements.

- Linked structures are good for inserting and deleting quickly, once you have a pointer to the place where it is happening, but they don't allow the **random access** to arbitrary elements that we need to do binary search.

- We could speed up our access to the middle elements of a linked list by adding more pointers, say from the beginning of the list to the middle, or from the middle to the three-quarters mark. Doing this right, we can add the abilities we need to get to the element we want in O(log n) time.

- The best way to arrange this sort of thing turns out to be a **tree** structure.

Trees, Binary Trees, and Vocabulary

- We've seen **trees** in the directory structures of operating systems, in the organization of a textbook or a company, and in Discussion #6 where we wrote a recursive method to compare binary trees for equality.

- A **binary tree** is one where each node may have a **left child** or a **right child** or both. There is one node called the **root**, and every node except for the root is the child of another node. A **leaf** is a node with no children.

- We use genealogical language to describe trees: **parent**, **ancestor**, **descendant**, sometimes even uncle or niece.

- We say that the root node of a tree is at **level** 0, its children are at level 1, its grandchildren at level 2, and so forth. The highest level of any node in the tree is the tree's height.

The BST Rule

- A binary search tree is a binary tree with an element, from some class `T` that has a `compareTo` method defined, at every node.

- The **BST Rule** is that for every node x, the element at x is *greater than* the elements in x's left child and all descendants of the left child, and is *less than* the elements in x's right child and all descendants of the right child. Thus x's element splits the smaller values in the l**eft subtree** from the greater values in the **right subtree**.

- This allows us to find an arbitrary element by a form of binary search. We look at the root. If it is our target we win, and if not we recurse to the root's left child (if the target is smaller than the root's element) or to its right child (if the target is greater). If the element is there we will find it this way, with a number of tries bounded by the height of the tree. If it is not there, we will find out when we recurse to a node that is null.

Traversals

- Just as we had prefix, infix, and postfix strings to represent formulas, we have three ways to traverse the nodes of a binary tree in order: **preorder**, **inorder**, and **postorder**. If our tree is the parse tree of a formula with binary operators and values at the leaves, each order gives the order of the characters in the corresponding string.

- Each traversal is defined recursively, so that we define how to **visit** each node. Visiting a node means processing it in some way, and our recursive definition of "traversing a node" x will give us a way to visit all the nodes in the subtree under x (x and all its descendants).

- A preorder traversal of x first visits x, then traverses its left child, then traverses its right child. An inorder traversal traverses the left child, visits the node, and traverses the right child. A postorder traversal traverses the left child, traverses the right child, and finally visits the node.

The `BSTInterface`

- The basic operations of a BST are exactly those of a list.

- The main difference (and the reason we don't just extend) is that the reset and getNext operations take a parameter. There are three different "current positions", one for each type of traversal.

```
public interface BSTInterface<T extends Comparable<T>> {
   static final int INORDER = 1, PREORDER = 2, POSTORDER = 3;
   boolean isEmpty( );
   int size( );
   boolean contains (T element);
   boolean remove (T element);
   T get (T element);
   void add (T element);
   int reset (int orderType);
   T getNext (int orderType);}
```

An Application Example

```
public class GolfApp2 {
   public static void main (String [ ] args) {
      Scanner conIn = new Scanner(System.in);
      String name; int score;
      BSTInterface<Golfer> golfers =
         new BinarySearchTree<Golfer>( );
      Golfer golfer; int numGolfers; String skip;
      System.out.println("Golfer name (press Enter to end): ");
      name = conIn.nextLine( );
      while (!name.equals("")) {
         System.out.println("Score: "); score = conIn.nextLine( );
         skip = conIn.nextLine( );
         golfer = new Golfer (name, score);
         golfers.add(golfer);
         System.out.println("Name: "); name = conIn.nextLine( );}
      System.out.println("The final results are: ");
      numGolfers = golfers.reset(BinarySearchTree.INORDER);
      for (int i = 1; i <= numGolfers; i++) {
         System.out.println(golfers.getNext(BST.INORDER);}}}
```

Beginning the Implementation

- Like a list, a tree is a linked structure whose nodes are allocated dynamically. We begin by defining a class for the nodes, which looks a lot like the classes for singly or doubly linked lists.

- Note that any binary tree with elements at every node could use a similar type of node -- this is a *search* tree node because T must be a class that supports the `compareTo` method.

```
public class BSTNode<T extends Comparable<T>> {
   protected T info;
   protected BSTNode left;
   protected BSTNode right;
   public BSTNode (T info) {
      this.info = info; left = right = null;}
   // getters and setters
   }
```

The `BinarySearchTree` Class

- You would think that we would keep a “current position” for each of the three types of traversals, but instead the reset operation will make a queue of all the elements, in the correct order, and the getNext method will just dequeue and return the next queue element.

```
public class BinarySearchTree<T extends Comparable<T>>
             implements BSTInterface<T> {
   protected BSTNode root;
   boolean found; // used by remove
   protected LinkUnbndQueue<T> inOrderQueue; // for traversal
   protected LinkUnbndQueue<T> preOrderQueue;  // ditto
   protected LinkUnbndQueue<T> postOrderQueue; // ditto
   public BinarySearchTree( ) {
      root = null;}
   public boolean isEmpty( ) {
      return (root == null);}
   // more methods to come next lecture!
```