CSE 220: Data Structures (Spring '14)

Binary search trees

Binary Search Trees

Table of contents

Introduction

See Sedgewick sections 5.4-5.6 for general discussion on trees, and 5.7 for recursive binary-tree algorithms. In these notes, we're going to focus on binary search trees, which are covered in Sedgewick section 12.6.

Binary search trees are binary trees with the additional properties:

for any internal node:

- the key in the left child must be less than the node's key, and
- the key in the right child must be larger than the node's key;

and this must hold recursively.

A consequence of this "must hold recursively" is the following:

for any internal node:

- all keys in the left subtree are less than the node's key, and
- all keys in the right subtree are greater than the node's key.

It is an excellent container for a Dictionary or Map ADT, especially if the tree is somewhat "balanced" (we'll talk more about this later), supporting fast insert, lookup and remove. In addition, it also supports fast minimum, maximum, successor, and predecessor operations. Note that a dictionary implemented with sorted arrays provide at least as fast operations except for in the cases of insert and remove.

Our binary tree has a simple Node class:

```
public class Node {
    public Object element;
    public Node left;
    public Node right;
    public Node parent;

    public Node (Object e) {
        this(e, null, null, null);
    }

    public Node (Object e, Node l, Node r, Node p) {
        element = e;
 left = l;
 right = r;
        parent = p;
    }
}
```

Note that we maintain a `parent` reference in each node, which may be needed to *speed up* certain operations, such as finding the successor or the predecessor of a key in a binary search tree.

Back to Table of contents

Operations on binary search trees

Some simple operations:

1. The **smallest key** is the **leftmost node** starting from the root of the tree. If you have a tree rooted at reference "root", calling `findLeftMost(root)` will return a reference to the leftmost node, and the key within the node is the smallest key in the tree.

```
Iteratively:

/**
 * Returns the leftmost node of this subtree rooted at "n".
 * @param n the root of this subtree.
 * @return the leftmost node.
 */
static Node findLeftMost(Node n) {
    while (n.left != null)
 n = n.left;
    return n;
}

Recursively:

static Node findLeftMost(Node n) {
    if (n.left == null)
 return n;
    else
 return findLeftMost(n.left);
}
```

2. The **largest key** is in the **rightmost node** starting from root of the tree. Just swap "left" with "right" in `findLeftMost` and you're done.
3. The **successor** of a key in node in a binary search tree is the key that is immediately larger this this key. Since the successor must be larger, it must belong in the right subtree. Specifically, it must be the **smallest key in the right subtree**! So, a partial implementation is the following:

```
/*
 * Returns the successor node of the given node.
 * @param n the node whose successor is sought.
 * @return the successor node, null if given node contains the largest
 *         key.
 */
public static Node findSuccessor(Node n) {
    return findRightMost(n.right);
}
```

However, this ONLY works if the given node has a right subtree. What do we do in the case where there is no right subtree? In the case where a node does not have a right subtree, the successor must be "above" somewhere. We move up the tree, and the first parent/grandparent node where we "turn right" is the successor. If we move to the root's parent, we get null, which is the case for the largest key in the tree.

```
public static Node findSuccessor(Node n) {
    if (n.right != null) {
 // If n has a right subtree, then the successor node is the
 // leftmost node in the right subtree.
 Node p = findLeftMost(n.right);
 return p;
    } else {
 // If n does not have a right subtree, the successor node is
 // an ancestor node, but only if it's to the "right" of n. The
 // successor to the largest node (which is the rightmost node
 // in the tree) is the parent of root, which is null.
 Node q = n;
 Node p = q.parent;
 while (p != null && p.right == q) {
     q = p;
     p = p.parent;
 }
 return p;
    }
}
```

This works for all cases.

4. Finding the predecessor works the same way, except that you swap "left" for "right" in the successor case.
5. How do we find a key in a binary search tree? Easy, we start at root, looking for the key in a node - if found, we're done; otherwise, we move left or right depending on the key value.

```
Iteratively:

/**
 * Finds the node containing the given key in tree rooted at "n".
 * @param key the key to find in a tree node.
 * @param n the root of this subtree.
 * @return the node if found, or null otherwise.
 */
static Node findNode(Object key, Node n) {
    while (n != null) {
 if (key.equals(n.element))
     return n;
 else if (((Comparable)key).compareTo(n.element) > 0)
     n = n.right;
 else
     n = n.left;
    }
    return n;
}

Recursively:

/**
 * Finds the node containing the given key in tree rooted at "n".
 * @param key the key to find in a tree node.
 * @param n the root of this subtree.
 * @return the node if found, or null otherwise.
 */
static Node findNode(Object key, Node n) {
    if (n == null)
 return null;
    else if (key.equals(n.element))
 return n;
    else if (((Comparable)key).compareTo(n.element) > 0)
 return findNode(key, n.right);
    else
 return findNode(key, n.left);
}
```

6. How would you get the keys in sorted manner? Two ways - you can visit the nodes "in-order" starting at the root; or, you can start at the leftmost node, and iterate by visiting the successor at each step. The first way has a "natural" recursive implementation, while the second way has a "natural" iterative implementation.

   First way - recursively (iteratively is not easy at all - just try by using a stack to emulate recursion; see Sedgewick for details):

```
/**
 * Prints the keys in sorted order using in-order traversal.
 * @param n the root of this subtree.
 */
public static void printSorted(Node n) {
    if (n == null)
 return;
    else {
 printSorted(n.left);
 System.out.println(n.element);
 printSorted(n.right);
    }
}
```

Second way - iteratively:

```
/**
 * Prints the keys in sorted order using iteration.
 * @param n the root of this subtree.
 */
public static void printSorted(Node n) {
    n = findLeftMost(n);
    while (n != null) {
 System.out.println(n.element);
 n = findSuccessor(n);
    }
}
```

Second way also works recursively (but of course!), but needs a "helper" method (see the two methods below - the first one is the helper one):

```
/**
 * Prints the keys in sorted order using iteration.
 * @param n the root of this subtree.
 */
public static void printSorted(Node n) {
    n = findLeftMost(n);
    printSortedRec(n);
}

private static void printSortedRec(Node n) {
    if (n == null)
 return;
    else {
 System.out.println(n.element);
 printSortedRec(findSuccessor(n));
    }
}
```

7. Ok, now we need to start insert and remove operations. Let's start with insertion. Remember that an insertion into a binary search tree ALWAYS replaces an external node with a new internal node containing the key that is being inserted.

   First, create a new Node object with the key in it. Now, if the tree is empty, then the new node becomes the root! Simple.

   Otherwise, we use lock-step method to move down the tree to find the external node that we'll replace (keeping a parent reference as we go down), and then add the new node in the appropriate place.

```
/**
 * Inserts a key in this binary search tree rooted at "root". Let's
 * assume distinct keys only, so we'll throw an exception if the key
 * already exists.
 *
 * @param key the key to insert.
 * @param root the root of this tree.
 * @return reference to the root, which may have changed.
 * @exception DuplicateKeyException if the key already exists.
 */
public static Node insert(Object key, Node root) {
    // The tree may be empty, in which case we set root to the new
    // node containing the given element.
    if (root == null) {
 root = new Node(key);
    } else {
 // Otherwise, tree is not empty, so we find the *parent* node
 // to which we'll attach the new node as a child. We use two
 // references in lock-step until one reaches an external node.
 Node p = null; // the parent node.
 Node n = root;
 while (n != null) {
     int compare = ((Comparable) key).compareTo(n.element);
     if (compare == 0) {
  throw new DuplicateKeyException();
     } else if (compare > 0) {
  p = n;
  n = n.right;
     } else {
  p = n;
  n = n.left;
     }
 }

 // At this point, n points to the external node which will be
 // replaced by the new node, and p points to the parent of n.
 // So we just have to attach the new node to either left or
 // the right child of p, depending on the relative values of
 // p.element and element.

 Node newNode = new Node(key);
 if (((Comparable) key).compareTo(p.element) > 0)
     p.right = newNode;
 else
     p.left = newNode;

 // p will become newNode's parent.
 newNode.parent = p;
    }
    return root;
}
```

There is a **much easier** and **much more elegant** recursive algorithm, but takes a bit of getting used to. See Sedgewick Sec 12.6 for details.

How would we handle duplicate keys? We have to use the relations < and >= or <= and > for the left and right subtrees. How we handle it in the insert method depends on the problem on how to handle it. If we're using binary search tree to implement a Set ADT (remember that we used a sorted array to implement a Set earlier), we may choose to ignore the duplicates. In the case of a phone book application, there is a value associated with a key, so for a duplicate key, it may simply update the value. Like I said, depends on the problem.

8. Ah, now we look at removing a key. Removing a key involves finding the node that contains the key, and removing that node from the binary search tree, ensuring that the resulting tree remains a binary search tree.

   Sedgewick uses a beautiful recursive algorithm to do this, but that may be slightly tricky to understand. So I'll cover the iterative one as well.

   Let's look at the 3 cases we had discussed in class:

   1. Leaf node with no children - removing this is trivial, as we simply have to the set the appropriate child reference in the parent to null.
   2. Node with 1 child - also simple, as we add a "bypass" from its parent to its only child. Have to keep track of left and right, etc, but that can easily be done by comparing the keys.
   3. Node with both children - here's the tough one. Removing such a node with destroy the tree's properties. There is one thing to note - since it has both its children, its predecessor and successor are both "below" it. The predecessor is the rightmost node in its left subtree, so it must not have more than 1 child; likewise, the successor has *at most* 1 child. Removing the predecessor or the successor then falls under cases 1 or 2, since it will have either 0 or 1 child, which we already know how

to do. How does that help us here? We simply replace the key in the node to be removed with its successor (or predecessor), and then remove the successor (or predecessor). Basically, we're reducing the problem of removing a node with two children to the one of removing a node with 0 or 1 child.

9. Rotating a tree around a given node. I'll leave this one for you to study from the textbook (Sec. 12.8).

Back to Table of contents

Performance of binary search trees

Now the big question - what's the **maximum number of comparisons** we need to make to find a key in a binary search tree? Well, in the worst case, we start from the root, work our way down to either a leaf node (success) or an external node (failure). Both cases, the maximum number of comparisons is 1 more than the tree's height (remember that a tree with a single node has a height of 0). We know that the maximum and minimum heights of a binary tree:

```
    log_2(N+1) <= h <= N - 1

Which means that the maximum number of comparisons, equal to 1+h, can be
computed easily as well: if C is the number of comparisons:

    C <= h + 1     (in the worst case, it's equal to h + 1)
       = N - 1 + 1 = N

    max # of comparisons = N

just like linear search! Not good at all. However, if the tree were nicely
balanced (see the text for definition), we have a "short" tree. If we now
use the minimum height, the number of comparisons is:

    C >= log_2(N+1) + 1
```

so at least approximately log_2(N) comparisons! Note that binary search (on sorted arrays) provides a guarantee of log_2(N) comparisons in the WORST case, and for a binary search tree there is actually a range from log_2(N) to h.

This should tell you that we really really want a balanced tree! But how? We'll do that next semester.

Back to Table of contents

Discussion

Back to Table of contents

Last modified: Thursday, 14 July 2011, 09:32 AM

- View all of my entries
    - Add a new entry
  - Messages
  - My private files
- My courses
  - cse220-01-sp14
    - Participants
      - Blogs
      - MUHAIMINUR RAHMAN
        - View profile
        - Forum posts
          - Posts
          - Discussions
        - Blogs
          - View all of my entries
          - Add a new entry
        - Messages
        - My private files
    - Course information
    - Arrays and linked lists
    - Stacks and Queues
    - Searching and sorting
    - Recursion
    - Trees and Graphs
      - Trees
      - Trees (PDF)
      - Binary search trees
      - Binary search trees (PDF)
      - Introduction to Graphs
      - Introduction to Graphs (PDF)
  - cse260-01-sp14

Skip Settings

Settings

- Course administration