CS61B Summer 2006
Instructor: Erin Korber
Lecture 17: 25 July

1 Ordered Dictionaries

A *dictionary* is a collection of items stored as key-element pairs, used to look up items quickly by key. We should be able to test keys for equality; for some applications, we might want to not allow more than one element with the same key (ex: student records stored by student ID number), but for others this is acceptable (ex: an English dictonary, since many words have multiple definitions). All dictionaries should support certain basic operations, like `size()`, `isEmpty()`, `find(Key k)`, `findAllVals(Key k)`, `insert(KeyValPair p)`, `remove(Key k)`, `removeAllVals(Key k)`. We will be studying several ways to implement dictionaries.

An *ordered dictionary* is a particular kind of dictionary in which we have a total ordering on the keys. (Whereas before, we could only test keys for equality, now we can compare them.) An ordered dictionary will support all the same methods as a generic dictionary, plus some additional ones, namely `closestBefore(Key k)`, `closestAfter(Key k)`.

Note that we will need to decide on a behavior for when the user tries to access a key that is not in the dictionary (with, e.g., a `find` operation). There are several ways to deal with this. We could just return `null` when they do this; this is usually a bad idea since this effectively prevents us from storing values that are null in the dictionary. We could return a special `NO_SUCH_ENTRY` sentinel value. We could throw a `NoSuchEntry` exception. Both of the latter two work well and are easy to implement; I will use the exception method here but either is fine.

I will use "Key" and "Value" to refer to the key type and value type of the pairs stored in the dictionary; what these actually are will depend on the specific application.

```
public interface OrderedDictionary {

  //returns the size of the dictionary
  public int size();

  //returns true if there are no items in the dictionary
  public boolean isEmpty();
```

```
  //if the dic. contains an item with key k, return that value
  public Value find(Key k) throws NoSuchEntry;

  //return a list of all the values that have key k (empty if there are none)
  public List findAllVals(Key k);

  //insert p into the dictionary
  public void insert(KeyValPair p);

  //remove an item with key k from the dic. and return its value
  public Value remove(Key k) throws NoSuchEntry;

  //remove all items with key k from the dic. and return a list of their values
  public List removeAllVals(Key k);

  //Return the item with the largest key less than or equal to k
  public KeyValPair closestBefore(Key k) throws NoSuchEntry;

  //Return the item with the smallest key greater than or equal to k
  public KeyValPair closestAfter(Key k) throws NoSuchEntry;
}
```

2 Binary Trees: Representation (review)

Recall that a binary tree is a rooted tree in which no node has more than two children. Additionally, every child is either a *left child* or a *right child* of its parent, even if its parent has only one child.

In the most popular binary tree representation, each tree node has three references to neighboring tree nodes: a "parent" reference, and "left" and "right" references for the two children. (For some algorithms, the "parent" references are unnecessary.) Each node also has an "item" reference.

```
public class BinaryTree {
  BinaryTreeNode root;
  int size;
}

public class BinaryTreeNode {
  Object item;
  BinaryTreeNode parent;
```

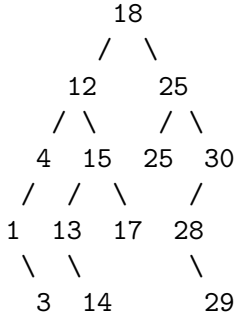
```
  BinaryTreeNode left;
  BinaryTreeNode right;

//Recall the ``inorder'' traversal method from before.
  public void inorder() {
    if (left != null) {
      left.inorder();
    }
    this.visit();
    if (right != null) {
      right.inorder();
    }
  }
}
```

3 Binary Search Trees

A binary search tree is the simplest implementation of an ordered dictionary. A *binary search tree* (BST) is a binary tree which satisfies the *BST invariant* - that for any node X, every key in X's left subtree is less than or equal to X's key, and every key in X's right subtree is greater than or equal to X's key.

When a node has only one child, that child is either the left child or the right child, depending on whether its key is smaller or larger than its parent's key. (A key equal to the parent's key can go into either subtree.)

Here is an example of a binary search tree:



An inorder traversal of a binary search tree visits the nodes in sorted order. In this sense, a search tree maintains a sorted list of entries, so it is at least as useful as a sorted linked list in that sense. However, operations

on a search tree are usually more efficient than the same operations on a sorted linked list. Let's look at several of them.

- `find` -

```
public Value find(Key k) throws NoSuchEntry {
  BinaryTreeNode node = root;               // Start at the root.
  while (node != null) {
    Key k2 = the key of node's item
    if (k < k2) {                           // Repeatedly compare search
      node = node.left;                     // key k with current node; if
    } else if (k > k2) {                    // k is smaller, go to the left
      node = node.right;                    // child; if k is larger, go to
    } else {    /* The keys are equal */    // the right child.  Stop when
      return (the value of the node's item) // we find a match (success;
    }                                       // return the entry) or reach
  }                                         // a null pointer (failure)
  throw new NoSuchEntry();
  return null;
}
```

- `closestBefore` - Search for the key k in the tree, just like in `find()`. As you go down the tree, keep track of the largest key not larger than k that you've encountered so far. If you find the key k, you can return it immediately. If you reach a null pointer, return the best key you found on the path.

  `closestAfter` can be implemented similarly.

  Why does this work? Because when we search for a key k not in the tree, we are guaranteed to encounter the two keys that bracket it (the closest larger and closest smaller). Here's why: Let x be the smallest key in the tree greater than k. Because k and x are "adjacent" keys, the result of comparing k with any other key y in the tree is the same as comparing x with y. Hence, find(k) will follow exactly the same path as find(x) until it reaches x. (After that, it may continue downward.) The same argument applies to the largest key less than k.

- `first(), last()`

  `first()` is very simple. If the tree is empty, return null. Otherwise, start at the root. Repeatedly go to the left child until you reach a node with no left child. That node has the minimum key.

`last()` is the same, except that you repeatedly go to the right child. In the example tree, observe the locations of the minimum (1) and maximum (30) keys.

- `insert(KeyValPair p)`

  `insert()` starts by following the same path through the tree as `find()`. (`find()` works *because* it follows the same path as `insert()`.) When it reaches a null reference, replace the null with a reference to a new node referencing the pair p.

  Duplicate keys are allowed. If `insert()` finds a node that already has the key k, it puts it the new entry in the left subtree of the older one. (We could just as easily choose the right subtree; it doesn't matter.)

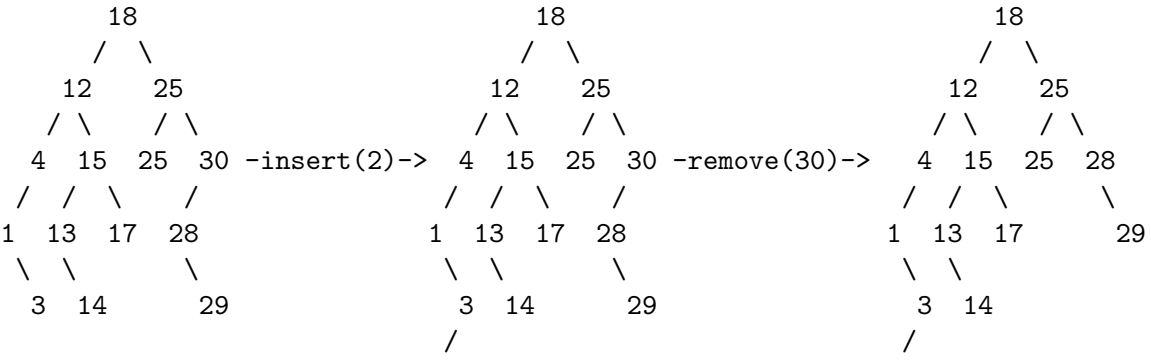
- `remove(Key k)`

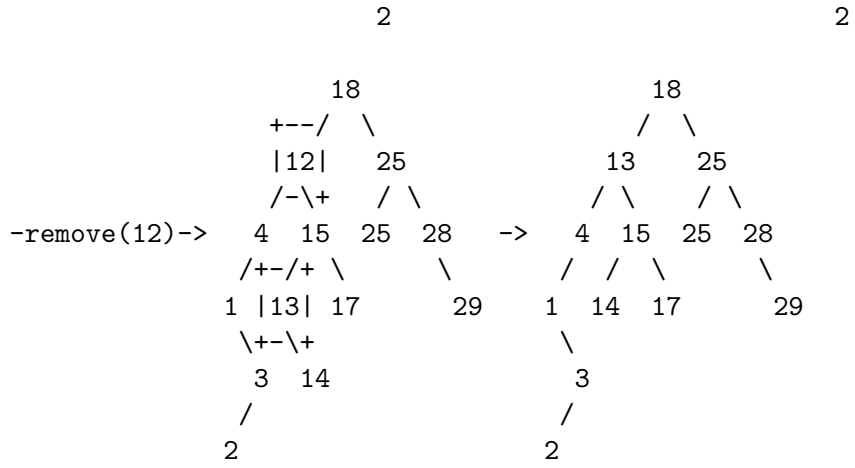
  `remove()` is the most difficult operation. First, find a node with key k using the same algorithm as `find()`. Throw an exception if k is not in the tree; otherwise, let n be the first node we find with key k.

  If n has no children, we easily detach it from its parent (set the reference that the parent has to it to null) and return it.

  If n has one child, move n's child up to take n's place. n's parent becomes the parent of n's child, and n's child becomes the child of n's parent. Dispose of n.

  If n has two children, however, we have to be a bit more clever. Let x be the node in n's right subtree with the smallest key. Remove x; since x has the minimum key in the subtree, x has no left child and is easily removed. Finally, replace n's entry with x's entry. x has the closest key to k that isn't smaller than k, so the binary search tree invariant still holds.





4 Running Times of BST Operations

In a perfectly balanced binary tree with depth d, the number of nodes n is $2^{d+1} - 1$. Why is this? Observe that at depth i, there are at most 2^i nodes. So the total number of nodes in a tree of depth d is $\sum_{i=0}^{d} 2^i = 2^{d+1} - 1$. Therefore, no node has depth greater than $\log_2 n$. The running times of `find()`, `insert()`, and `remove()` are all proportional to the depth of the last node encountered, so they all run in O(log n) worst-case time on a perfectly balanced tree.

On the other hand, it's easy to form a severely imbalanced tree (e.g. every node has only 1 child), wherein these operations will often take linear time.

There's a vast middle ground of binary trees that are reasonably well-balanced, albeit certainly not perfectly balanced, for which search tree operations will run in O(log n) time. You may need to resort to experiment to determine whether any particular application will use binary search trees in a way that tends to generate somewhat balanced trees or not. Binary search trees offer O(log n) performance on insertions of randomly chosen or randomly ordered keys (with high probability).

Unfortunately, there are occasions where you might fill a tree with entries that happen to be already sorted. In this circumstance, a disastrously imbalanced tree will be the result. Because of this, technically, most of these operations on binary search trees have $\Theta(n)$ worst-case running time.

For this reason, researchers have developed a variety of algorithms for keeping search trees balanced. We'll be discussing several of these over the

next couple of days.