Lecture 7: Binary Search Trees

Read: Chapt 4 of Weiss, through 4.3.6.

Searching: Searching is among the most fundamental problems in data structure design. Traditionally, we assume that we are given a set of elements $\{E_1, E_2, \ldots, E_n\}$, where each E_i is associated with a distinct *key* value, x_i over some totally ordered domain. Given an arbitrary search key x, we wish to determine whether there is a element containing this key.

Assuming that each object contains a key field is sometimes a bit awkward (since it may involve accessing private data). A somewhat more attractive way to handle this in modern languages like C++ and Java is to assume that the element E has a comparison method defined for it. (In Java we would say that the object is a subclass of type `Comparable`.) This means that the class provides a method `int compareTo()`, whose argument is of the same type as this class. Since we are not constrained to using one programming language, we will allow ourselves to overload the comparison operators (which Java does not permit) to make our pseudo-code easier to read.

Relationship	Expressed in Java	What we'll write
$r_1 < r_2$	`r1.compareTo(r2) < 0`	`r1 < r2`
$r_1 = r_2$	`r1.compareTo(r2) == 0`	`r1 == r2`
$r_1 > r_2$	`r1.compareTo(r2) > 0`	`r1 > r2`

Henceforth, we will not mention keys, but instead assume that our base `Element` type has such a comparison method. But we will abuse our own convention from time to time by refering to x sometimes as "key" (when talking about ordering and comparisons) and sometimes as an "element" (when talking about insertion).

The Dictionary ADT: Perhaps the most basic example of a data structure based on the structure is the dictionary. A *dictionary* is an ADT which supports the operations of insertion, deletion, and finding. There are a number of additional operations that one may like to have supported, but these seem to be the core operations. Throughout, let us assume that x is of type `Element`.

`insert(Element x)`: Insert x into the dictionary. Recall that we assume that keys uniquely identify records. Thus, if an element with the same key as x already exists in the structure, there are a number of possible responses. These include ignoring the operation, returning an error status (or printing an error message), or throwing an exception. Throwing the exception is probably the cleanest option, but not all languages support exceptions.

`delete(Element x)`: Delete the element matching x's key from the dictionary. If this key does not appear in the dictionary, then again, some special error handling is needed.

`Element find(Element x)`: Determine whether there is an element matching x's key in the dictionary? This returns a pointer to the associated object, or null, if the object does not appear in the dictionary.

Other operations that might like to see in a dictionary include printing (or generally iterating through) the entries, range queries (find or count all objects in a range of values), returning or extracting the minimum or maximum element, and computing set operations such as union and intersection. There are three common methods for storing dictionaries: sorted arrays, hash tables, and binary search trees. We discuss two of these below. Hash tables will be presented later this semester.

[1]Copyright, David M. Mount, 2001

Sequential Allocation: The most naive idea is to simply store the keys in a linear array and run sequentially through the list to search for an element. Although this is simple, it is not efficient unless the set is small. Given a simple unsorted list insertion is very fast $O(1)$ time (by appending to the end of the list). However searching takes $O(n)$ time in the worst case.

In some applications, there is some sort of locality of reference, which causes the same small set of keys to be requested more frequently than others over some subsequence of access requests. In this case there are heuristics for moving these frequently accessed items to the front of the list, so that searches are more efficient. One such example is called *move to front*. Each time a key is accessed, it is moved from its current location to the front of the list. Another example is called *transpose*, which causes an item to be moved up one position in the list whenever it is accessed. It can be proven formally that these two heuristics do a good job in placing the most frequently accessed items near the front of the list.

Instead, if the keys are sorted by key value then the expected search time can be reduced to to $O(\log n)$ through *binary search*. Given that you want to search for a key x in a sorted array, we access the middle element of the array. If x is less than this element then recursively search the left sublist, if x is greater then recursively search the right sublist. You stop when you either find the element or the sublist becomes empty. It is a well known fact that the number of probes needed by binary search is $O(\log n)$. The reason is quite simple, each probe eliminates roughly one half of the remaining items from further consideration. The number of times you can "halve" a list of size n is $\lg n$ (where lg means log base 2).

Although binary search is fast, it is hard to update the list dynamically, since the list must be maintained in sorted order. This would require $O(n)$ time for insertion and deletion. To fix this we use binary trees, which we describe next.

Binary Search Trees: In order to provide the type of rapid access that binary search offers, but at the same time allows efficient insertion and deletion of keys, the simplest generalization is called a *binary search tree*. The idea is to store the records in the nodes of a binary tree, such that an inorder traversal visits the nodes in increasing key order. In particular, if x is the key stored in the root node, then the left subtree contains all keys that are less than x, and the right subtree stores all keys that are greater than x. (Recall that we assume that keys are distinct, so no other keys are equal to x.)

Defining such an object in C++ or Java typically involves two class definitions. One class for the tree itself, `BinarySearchTree` (which is publically accessible) and another one for the individual nodes of this tree, `BinaryNode` (which is a protected friend of the BinarySearchTree). The main data object in the tree is a pointer to root node. The methods associated with the binary search tree class are then transfered to operations on the nodes of the tree. (For example, finding a node in the tree is might be transformed to a find operation starting with the root node of the tree.) See our text for an example.

A node in the binary search tree would typically consist of three members: an associated *Element*, which we will call `data`, and a `left` and `right` pointer to the two children. Of course other information (parent pointers, level links, threads) can be added to this.

Search in Binary Search Trees: The search for a key x proceeds as follows. The search key is compared to the current node's key, say y. If it matches, then we are done. Otherwise if $x < y$ we recursively search the left subtree and if $x > y$ then we recursively search the right subtree. A natural way to handle this would be to make the search procedure a recursive member function of the BinaryNode class. The one technical hassle in implementing this in C++ or Java is that we cannot call a member function for a null pointer, which happens naturally if the search fails and we fall out the bottom of the tree. There are a number of ways to deal with this. Our book proposes making the search function a member of the BinarySearchTree class, and passing in the pointer to the current BinaryNode as an argument. (But this is not the only way to handle this.)

By the way, the initial call is made from the find() method associated with the BinarySearchTree class, which invokes find(x, root), where root is the root of the tree.

Recursive Binary Tree Search

```
Element find(Element x, BinaryNode p) {
    if (p == null) return null;            // didn't find it
    else if (x < p.data)                   // x is smaller?
        return find(x, p.left);            // search left
    else if (x > p.data)                   // x is larger?
        return find(x, p.right);           // search right
    else return p.data;                    // found it!
}
```

It should be pretty easy to see how this works, so we will leave it to you to verify its correctness. We will often express such simple algorithms in recursive form. Such simple procedures can often be implemented iteratively, thus saving some of the overhead induced by recursion. Here is a nonrecursive version of the same algorithm.

Nonrecursive Binary Tree Search

```
Element find(Element x) {
    BinaryTreeNode p = root;
    while (p != null) {
        if (x < p.data) p = p.left;
        else if (x > p.data) p = p.right;
        else return p.data;
    }
    return null;
}
```

Given the simplicity of the nonrecursive version, why would anyone ever use the recursive version? The answer is the no one would. However, we will see many more complicated algorithms that operate on trees, and these algorithms are almost always much easier to present and understand in recursive form (once you get use to recursive thinking!).

What is the worst-case running time of the search algorithm? Both of them essentially take constant time for each node they visit, and then descend to one of the descendents. In the worst-case the search will visit the deepest node in the tree. Hence the running time is $O(h)$, where h is the height of the tree. If the tree contains n elements, h could vary anywhere from $\lg n$ (for a balanced tree) up to $n-1$ for a degenerate (path) tree.

Insertion: To insert a new element in a binary search tree, we essentially try to locate the key in the tree. At the point that we "fall out" of the tree, we insert a new leaf node containing the desired element. It turns out that this is always the right place to put the new node.

As in the previous case, it is probably easier to write the code in its nonrecursive form, but let's try to do the recursive version, since the general form will be useful for more complex tree operations. The one technical difficulty here is that when a new node is created, we need to "reach up" and alter one of the pointer fields in the parent's node. To do this the procedure returns a pointer to the subtree with the newly added element. Note that "most" of the time, this is returning the same value that is passed in, and hence there is no real change taking place.

The initial call from the BinarySearchTree object is `root = insert(x, root)`. We assume that there is a constructor for the BinaryNode of the form `BinaryNode(data, left, right)`. An example is shown below.

Binary Tree Insertion

```
BinaryNode insert(Element x, BinaryNode p) {
    if (p == NULL)
        p = new BinaryNode(x, null, null);
    else if (x < p.data) p.left  = insert(x, p.left);
    else if (x > p.data) p.right = insert(x, p.right);
    else ...Error: attempt to insert duplicate!...
    return p
}
```

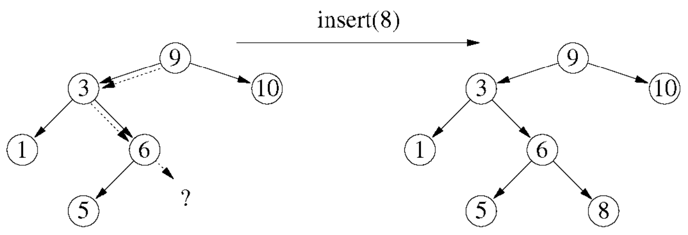

Figure 18: Binary tree insertion.