CMSC 420: Lecture 4
Binary Search Trees

Searching: Searching is among the most fundamental problems in data structure design. Our objective is to store a set of *entries* $\{e_1, \ldots, e_n\}$, where each e_i is a pair (x_i, v_i), where x_i is a *key value* drawn from some totally ordered domain (e.g., integers or strings) and v_i is an associated *data value*. The data value is not used in the search itself, but is needed by whatever application is using our data structure.

We assume that keys are unique, meaning that no two entries share the same key. Given an arbitrary search key x, the basic search problem is determining whether there exists an entry matching this key value. To implement this, we will assume that we are given two types, `Key` and `Value`. We will also assume that key values can be compared using the usual comparison operators, such as `<`, `>`, `==`. We make no assumptions about `Value`, since it is used by the application only. (We will discuss Java implementation at the end of the lecture.)

The Dictionary (Map) ADT: A *dictionary* (also called a *map*) is an ADT that supports the operations insert, delete, and find:

`void insert(Key x, Value v):` Stores an entry with the key-value pair (x, v). We assume that keys are unique, and so if this key already exists, an error condition will be signaled (e.g., an exception will be thrown).

`void delete(Key x):` Delete the entry with x's key from the dictionary. If this key does not appear in the dictionary, then an error conditioned is signaled.

`Value find(Key x):` Determine whether there is an entry matching x's key in the dictionary? If so, it returns a reference to associated value. Otherwise, it returns a `null` reference.

There are a number of additional operations that one may like to support, such as iterating the entries, answering range queries (that is, reporting or counting all objects between some minimum and maximum key values), returning the kth smallest key value, and computing set operations such as union and intersection.

There are three common methods for storing dictionaries: sorted arrays, search trees, and hashing. We will discuss the first two in this a subsequent lectures. (Hashing will be presented later this semester.)

Sequential Allocation: The most naive approach for implementing a dictionary data structure is to simply store the entries in a linear array without any sorting. To find a key value, we simply run sequentially through the list until we find the desired key. Although this is simple, it is not efficient. Searching and deletion each take $O(n)$ time in the worst case, which is very bad if n (the number of items in the dictionary) is large. Although insertion only involves $O(1)$ to insert a new item at the end of the array (assuming we don't overflow), it would require $O(n)$ time to check that we haven't inserted a duplicate element.

An alternative is to sort the entries by key value. Now, *binary search* can be used to locate a key in $O(\log n)$ time, which is much more efficient. (For example, if $n = 10^6$, $\log_2 n$ is only around 20.) While searches are fast, updates are slow. Insertion and deletion require $O(n)$ time, since the elements of the array must be moved around to make space.

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. Each node, called a `BSTNode`, stores a key, `key`, and an associated value, `value`.

These nodes are organized so that an inorder traversal visits the nodes in increasing key order. In particular, if x is the key stored in some 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 (see Fig. 1(a)). (Recall that we assume that keys are distinct, so no other key can be equal to x.)

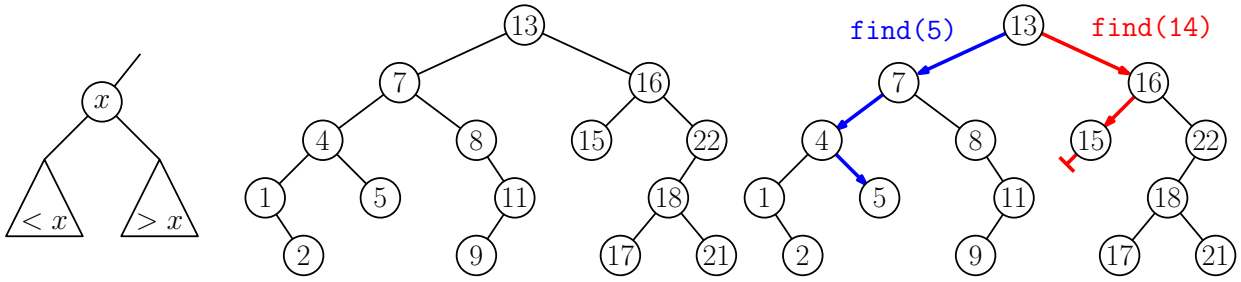

Fig. 1: A possible binary search tree for the key set $\{1, 2, 4, 5, 7, 8, 9, 11, 13, 15, 16, 17, 18, 21, 22\}$ and the searcg paths for `find(5)` and `find(14)`.

Search in Binary Search Trees: The search for a key x proceeds as follows. We start by assigning a pointer p to the root of the tree. We compare x to p's key, that is, `p.key`. If they are equal, we have found it and we are done. Otherwise, if x is smaller, we recursively search p's left subtree, and if x is larger, we recursively visit p's right subtree. The search proceeds until we either find the key (see Fig. 1(b)) or we fall out of the tree (see Fig. 1(b)). The code block below shows a possible pseudocode implementation of the find operation. The initial call is `find(x, root)`, where `root` is the root of the tree. It is easy to see based on the definition of a binary tree why this is correct.

Recursive Binary Tree Search

```
Value find(Key x, BSTNode p) {
    if (p == null) return null;          // unsuccessful search
    else if (x < p.key)                  // x is smaller?
        return find(x, p.left);          // ... search left
    else if (x > p.key)                  // x is larger?
        return find(x, p.right);         // ... search right
    else return p.value;                 // successful search
}
```

Query time: What is the running time of the search algorithm? Well, it depends on the key you are searching for. In the worst case, the search time is proportional to the height of the tree. The height of a binary search tree with n entries can be as low as $O(\log n)$ for the case of *balanced tree* (see Fig. 2 left) or as large as $O(n)$ for the case of a *degenerate tree* (see Fig. 2 right). However, we shall see that if the keys are inserted in random order, the expected height of the tree is just $O(\log n)$.

Can we devise ways to force the tree height to be $O(\log n)$? The answer is yes, and in future lectures we will see numerous approaches for guaranteeing this.

Insertion: To insert a new key-value entry (x, v) in a binary search tree, we first try to locate the key in the tree. If we find it, then the attempt to insert a duplicate key is an error. If not,

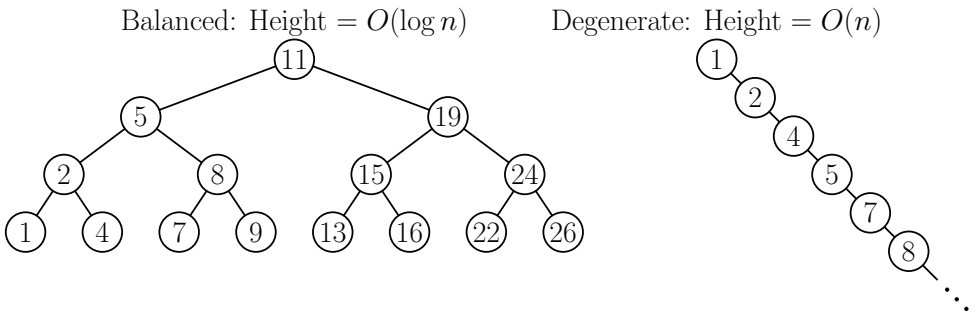

Fig. 2: Balanced and degenerate binary trees.

we effectively "fall out" of the tree at some node p. We insert a new leaf node containing the desired entry as a child of p. It turns out that this is always the right place to put the new node. (For example, in Fig. 3, we fall out of the tree at the left child of node 15, and we insert the new node with key 14 here.)

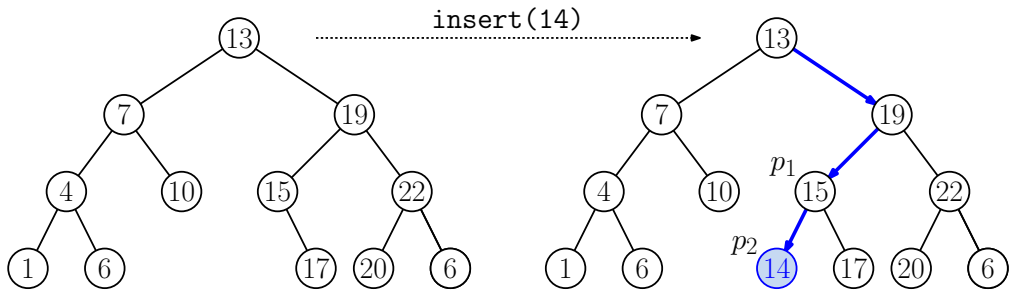

Fig. 3: Binary tree insertion.

This can naturally be implemented recursively. If the inserted key is smaller than the current node's key, we insert recursively on the left. If it is greater, we insert on the right. (It cannot be equal, or we have a duplicate key!) When we fall out of the tree (a `null` link), we create a new node here. The pseudocode is presented in the code block below. The initial call is `root = insert(x, v, root)`. We assume that there is a constructor for the `BSTNode`, which is given the key, value, and the initial values of the left and right child pointers.

Recursive Binary Tree Insertion

```
BSTNode insert(Key x, Value v, BSTNode p) {
    if (p == null)                          // fell out of the tree?
        p = new BSTNode(x, v, null, null);  // ... create a new leaf node here
    else if (x < p.key)                     // x is smaller?
        p.left  = insert(x, v, p.left);     // ...insert left
    else if (x > p.key)                     // x is larger?
        p.right = insert(x, v, p.right);    // ...insert right
    else throw DuplicateKeyException;       // x is equal ...duplicate key!
    return p                                // return ref to current node (sneaky!)
}
```

Coding Trick: (*Be sure you understand this!*) You will see one technical oddity in the above pseudocode implementation. Consider the line: "`p.left = insert(x, v, p.left)`". Why

does the insert function return a value, and why do we replace the left-child link with this value?

Here is the issue. Whenever we create the new node, we need to "reach up" and modify the appropriate child link in the parent's node. Unfortunately, this is not easy to do in our recursive formulation, since the parent node is not a local variable. To do this, our insert function will return a reference to the modified subtree after the insertion. We store this value in the appropriate child pointer for the parent.

To better understand how our coding trick works, let's refer back to Fig. 3. Let p_1 denote the node containing 15. Since $14 < 15$, we effectively invoked the command "`p1.left = insert(x, v, p1.left)`". But since this node has no left child (`p1.left == null`), this recursive call creates a new `BSTNode` containing 14 and returns a pointer to it, call it p_2. Since this is the return value from the recursive call, we effectively have executed "`p1.left = p2`", thus linking the 14 node as the left child of the 15 node. Slick!

Deletion: Next, let us consider how to delete an entry from the tree. Deletion is a more involed than insertion. While insertion adds nodes at the leaves of the tree, but deletions can occur at any place within the tree. Deleting a leaf node is relatively easy, since it effectively involves "undoing" the insertion process (see Fig. 4(a)). Deleting an internal node requires that we "fill the hole" left when this node goes away. The easiest case is when the node has just a single child, since we can effectively slide this child up to replace the deleted node (see Fig. 4(b)).

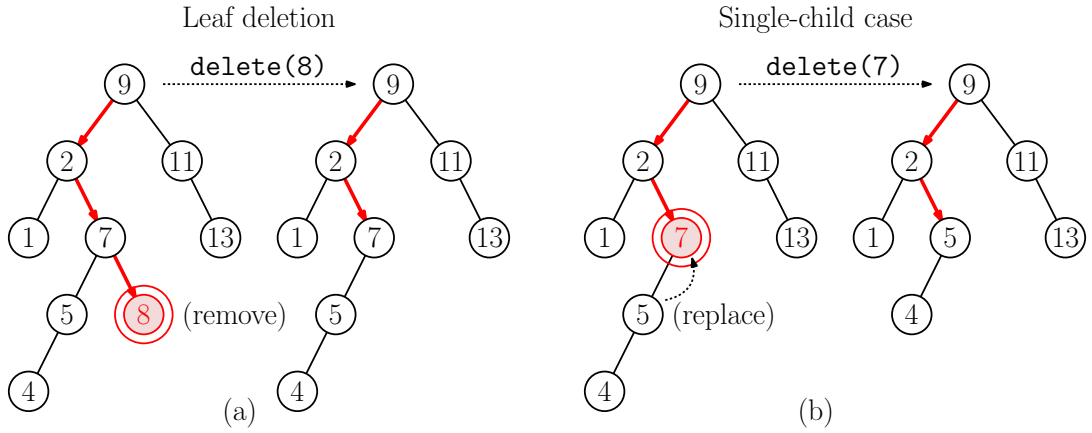

Fig. 4: Deletion: (a) Leaf and (b) single-child case.

The hardest case is when the deleted node that has two children. Let p denote the node to be deleted (see Fig. 5(a)):

- First off, we we need to find a suitable *replacement node*, whose key/value will fill the hole left by deletion. We can either take the largest key from p's left subtree (its inorder predecessor) or the smallest key from its right subtree (its inorder successor). Let's arbitrarily decide to do the latter. Call this node r (see Fig. 5(b)). Note that because p has two children, its inorder successor is the "leftmost" node of p's right subtree. Call r the *replacement node*.
- Copy the contents of r (both key and value) to p (see Fig. 5(c)).
- Recursively delete node r from p's right subtree (see Fig. 5(d)).

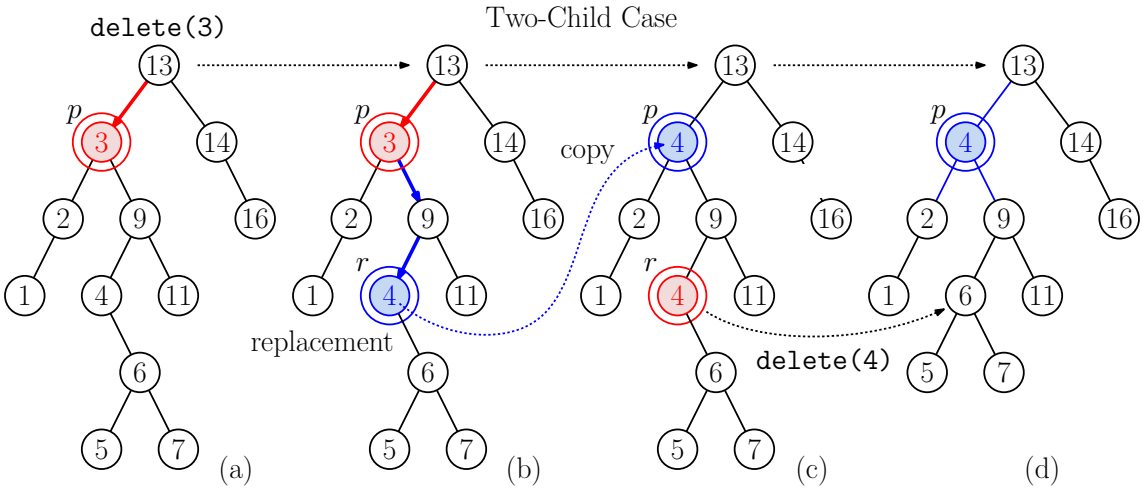

Fig. 5: Deletion: Two-child case.

It may seem that we have made no progress, because we have just replaced one deletion problem (for p) with another (for r). However, the task of deleting r is much simpler. The reason is, since r is p's inorder successor, r is the leftmost node of p's right subtree. It follows that r has no left child. Therefore, r is either a leaf or it has a single child, implying that it is one of the two "easy" deletion cases that we discussed earlier.

Deletion Implementation: Before giving the code for deletion, we first present a utility function, `findReplacement()`, which returns a pointer to the node that will replace p in the two-child case. As mentioned above, this is the inorder successor of p, that is, the leftmost node in p's right subtree. As with the insertion method, the initial call is made to the root of the tree, `delete(x, root)`. Again, we will employ the sneaky trick of returning a pointer to the revised subtree after deletion, and store this value in the child link. See the code fragment below.

Replacement Node for the Two-child Case

```
BSTNode findReplacement(BSTNode p) {
    BSTNode r = p.right;
    while (r.left != null) r = r.left;
    return r;
}
```

```
// find p's replacement node
// start in p's right subtree
// go to the leftmost node
```

The full deletion code is given in the following code fragment. As with insertion, the code is quite tricky. For example, can you see where the leaf and single-child cases are handled in the code? We do not have a conditional that distinguishes between these cases. How can that be correct. (But it is!)

Analysis of Binary Search Trees: It is not hard to see that all of the procedures `find()`, `insert()`, and `delete()` run in time that is proportional to the height of the tree being considered. (The `delete()` procedure is the only one for which this is not obvious. Because the replacement node is the inorder successor of the deleted node, it is the leftmost node of the right subtree. This implies that the replacement node has no left child, and so it will fall into one of the easy cases, which do not require a recursive call.)

Binary Tree Deletion

```
BSTNode delete(Key x, BSTNode p) {
    if (p == null)                                  // fell out of tree?
        throw KeyNotFoundException;                 // ...error - no such key
    else {
        if (x < p.data)                             // look in left subtree
            p.left = delete(x, p.left);
        else if (x > p.data)                        // look in right subtree
            p.right = delete(x, p.right);
                                                    // found it!
        else if (p.left == null || p.right == null) { // either child empty?
            if (p.left == null) return p.right;     // return replacement node
            else                return p.left;
        }
        else {                                      // both children present
            r = findReplacement(p);                 // find replacement node
            copy r's contents to p;                 // copy its contents to p
            p.right = delete(r.key, p.right);       // delete the replacement
        }
    }
    return p;
}
```

The question is, given a binary search tree T containing n keys, what can be said about the height of the tree? It is easy to devise "terrible" insertion orders so that the tree has the worst possible height of $n-1$. Conversely, it is possible to devise "perfect" insertion orders, which result in a balanced tree of height $\lfloor \log_2 n \rfloor$. (As an exercise, think about how you would do either of these.)

Since the worst case is obviously very bad, let's consider the expected case. Suppose that the keys are inserted in random order. To be more precise, given n keys, let us assume that each of the $n!$ insertion orders is equally likely. What is the *expected height* of the resulting tree? It turns out to be pretty good!

Theorem: Given a set of n keys $x_1 < x_2 < \ldots < x_n$, let $H(n)$ denote the expected height of a binary search tree, under the assumption that every one of the $n!$ insertion orders is equally likely. Then $H(n) = O(\log n)$.

Proving this is not an trivial exercise. (If you saw the expected-case analysis for Quicksort in CMSC 351, it is quite similar.)

Quick and Dirty Analysis: (*Optional*)

Rather than give the full proof, we will provide a much simpler one, which will hopefully convince you that the assertion is reasonable. Instead of proving that *every* node of the tree is at expected depth $O(\log n)$, we will prove that the *leftmost* node of the tree (that is, the node associated with the smallest key value) will be at expected depth $O(\log n)$.

Theorem: Given a set of n keys $x_1 < x_2 < \ldots < x_n$, let $D(n)$ denote the expected depth of *leftmost* node x_1 after inserting all these keys in a binary search tree, under the assumption that all $n!$ insertion orders are equally likely. Then $D(n) \leq 1 + \ln n$, where ln denotes the natural logarithm.

Proof: We will track the depth of the leftmost node of the tree through the sequence of insertions. Suppose that we have already inserted $i-1$ keys from the sequence, and we are in the process of inserting the ith key. The only way that the leftmost node changes is when the ith key is the lowest key value that has been seen so far in the sequence. That is, the i element to be inserted in the new minimum value among all the keys in the tree.

For example, consider the insertion sequence $S = \langle 9, 5, 10, 6, 3, 4, 2\rangle$. Observe that the minimum value changes three times, when 5, 3, and 2 are inserted. If we look at the binary tree that results from this insertion sequence we see that the depth of the leftmost node also increases by one with each of these insertions.

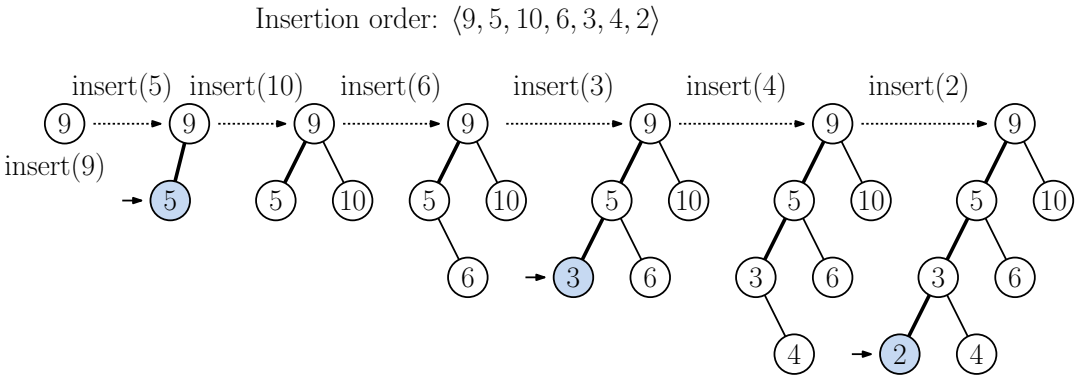

Fig. 6: Length of the leftmost chain.

To complete the analysis, it suffices to determine (in expectation) the number of times that the minimum in a sequence of n random values changes. To make this formal, for $2 \le i \le n$, let X_i denote the random variable that is 1 if the ith element of the random sequence is the minimum among the first i elements, and 0 otherwise. (In our sequence S above, $X_2 = X_5 = X_7 = 1$, because the minimum changed when the second, fifth, and seventh elements were added. The remaining X_i's are zero.)

To analyze X_i, let's just focus on the first i elements and ignore the rest. Since every permutation of the numbers is equally likely, the minimum among the first i is equally likely to come at any of the positions first, second, ..., up to ith. The minimum changes only if comes last out of the first i. Thus, $\Pr(X_i = 1) = \frac{1}{i}$ and $\Pr(X_i = 0) = 1 - \frac{1}{i}$. Whenever this random event occurs ($X_i = 1$), the minimum has changed one more time. Therefore, to obtain the expected number of times that the minimum changes, we just need to sum the probabilities that $X_i = 1$, for $i = 2, \ldots, n$. Thus we have

$$D(n) \;=\; \sum_{i=2}^{n} \frac{1}{i} \;=\; \left(\sum_{i=1}^{n} \frac{1}{i}\right) - 1.$$

This summation is among the most famous in mathematics. It is called the *Harmonic Series*. Unlike the geometric series ($1/2^i$), the Harmonic Series does not converge. But it is known that when n is large, its value is very close to $\ln n$, the natural log of n. (In fact, it is not more than $1 + \ln n$.)

Therefore, we conclude that the expected depth of the leftmost node in a binary search tree under n random insertions is at most $1 + \ln n = O(\log n)$, as desired.

Random Insertions and Deletions: Interestingly, the analysis of the expected-case tree height breaks down if we perform both insertions and deletions. Suppose that we consider a very long sequence of insertions and deletions, which occur at roughly the same rate so that, in steady state, the tree has roughly n nodes. Let us also assume that insertions are random (drawn say from some large domain of candidate elements) and deletions are random in the sense that a random element from the tree is deleted each time.

It is natural to suppose that the $O(\log n)$ bound should apply, but remarkably it does not! It can be shown that over a long sequence, the height of the tree will converge to a significantly larger value of $O(\sqrt{n})$.[1]

The reason has to do with the fact that the replacement element was chosen in a "biased" manner, always taking the inorder successor. Over the course of many deletions, this repeated bias causes the tree's structure to skew away from the ideal. This can be remedied by selecting the replacement node in an unbiased manner, choosing randomly between the inorder successor or inorder predecessor. It has been shown experimentally that this resolves the issue, but (to the best of my knowledge) it is not known whether the expected height of this balanced version of deletion matches the expected height for the insertion-only case (see Culberson and Munro, Algorithmica, 1990).

Java Implementation: So far, we have been expressing our functions in pseudocode. Defining a binary search tree object in a modern object-oriented language like Java would be done in a slightly different manner. First, rather than fixing specific types for the key and value, we would make the parameterized (generic) types. Let's call these `Key` and `Value`, respectively. We should not assume that we can simply apply operators such as "`<`", "`>`", and "`==`" for comparing keys. For this reason, we will assume that our `Key` object implements the `Comparable<Key>` interface, which means that it defines a method `compareTo`, which compares two objects of type `Key`. (By the way, this is not the only way to achieve this. Another approach is by associating a `Comparator` with the tree. Comparators offer some added flexibility, since multiple comparators can be defined for the same object, allowing you to sort them according to different criteria.)

In order to implement nodes, the class `BinarySearchTree` can define an *inner class*, called `BSTNode`. In Java, we can use the default access modifier. This means that the `BSTNode` members are accessible within the same package but hidden from the outside. Public functions, like `find`, `insert`, and `delete`, each invoke corresponding local functions, each of which is invoked on the root of the tree. A partial example (showing just the `find` function) is shown in the following code fragment.

[1] There is an interesting history regarding this question. It was believed for a number of years that random deletions did not alter the structure of the tree. A theorem by T. N. Hibbard in 1962 proved that the tree structure was probabilistically unaffected by deletions. The first edition of D. E. Knuth's famous book on data structures, quotes this result. In the mid 1970's, Gary Knott, a Ph.D. student of Knuth and later a professor at UMD, discovered a subtle flaw in Hibbard's result. While the structure of the tree is probabilistically the same, the distribution of keys is not. However, Knott could not resolve the asymptotic running time. The analysis showing that $O(\sqrt{n})$ bound was due to Culberson and Munro in the mid 1980's.

(Partial) Java Binary Tree

```
public class BinarySearchTree<Key extends Comparable<Key>, Value> {

    class BSTNode {                         // A node of the tree
        Key key;
        Value value;
        BSTNode left;
        BSTNode right;

        // ... other utility methods omitted
    }

    Value find(Key x, BSTNode p) {          // local find function
        if (p == null) return null;         // unsuccessful search
        else if (x.compareTo(p.key) < 0)    // x is smaller?
            return find(x, p.left);         // ... search left
        else if (x.compareTo(p.key) > 0)    // x is larger?
            return find(x, p.right);        // ... search right
        else return p.value;                // successful search
    }

    // ... other methods (insert, delete, ...) omitted

    private BSTNode root;                   // root of tree

    public Value find(Key x) {              // public find key
        return find(x, root);               // invoke local find
    }
}
```