Binary Search Trees

Contents

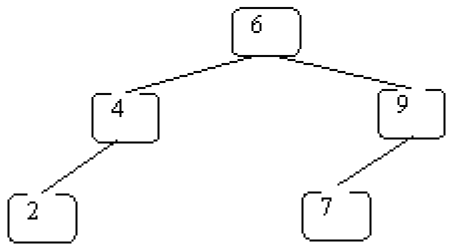
Introduction

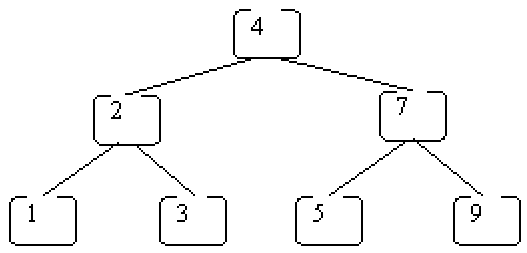
An important special kind of binary tree is the **binary search tree** (**BST**). In a BST, each node stores some information including a unique **key value** and perhaps some associated data. A binary tree is a BST iff, for every node n, in the tree:

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

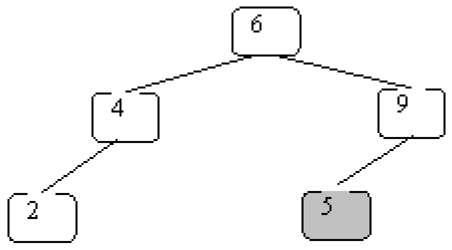
Note: if duplicate keys are allowed, then nodes with values that are equal to the key in node n can be either in n's left subtree or in its right subtree (but not both). In these notes, we will assume that duplicates are not allowed.

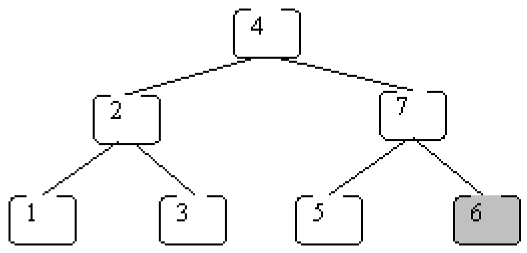
Here are some BSTs in which each node just stores an integer key:





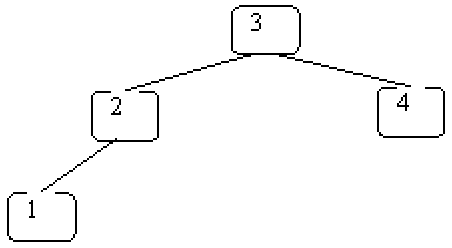
These are not BSTs:

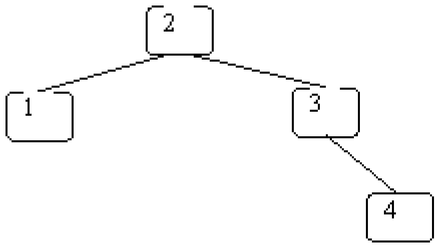




In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

Note that more than one BST can be used to store the same set of key values. For example, both of the following are BSTs that store the same set of integer keys:





The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

TEST YOURSELF #1

Question 1: Which of the following binary trees are BSTs? If a tree is ***not*** a BST, say why.

```
         A                  10               cat               15
        / \                /                /   \             /  \
       B   C              5               bat   rat          5    22
                         /               /                    \     \
                       -3              ant                     20    30
```

Question 2: Using which kind of traversal (pre-order, post-order, in-order, or level-order) visits the nodes of a BST in sorted order?

solution

Implementing BSTs

To implement a binary search tree, we will use two classes: one for the individual tree nodes, and one for the BST itself. The following class definitions assume that the BST will store only key values, no associated data.

```
class BSTnode<K> {
    // *** fields ***
    private K key;
    private BSTnode<K> left, right;

    // *** constructor ***
    public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {
        this.key = key;
        this.left = left;
        this.right = right;
    }

    // *** methods ***

    // accessors (access to fields)
    public K getKey() { return key; }
    public BSTnode<K> getLeft() { return left; }
    public BSTnode<K> getRight() { return right; }


    // mutators (change fields)
```

```
        public void setKey(K newK) { key = newK; }
        public void setLeft(BSTnode<K> newL) { left = newL; }
        public void setRight(BSTnode<K> newR) { right = newR; }
    }

    public class BST<K extends Comparable<K>> {
        // *** fields ***
        private BSTnode<K> root; // ptr to the root of the BST

        // *** constructor ***
        public BST() { root = null; }

        // *** methods ***

        public void insert(K key) throws DuplicateException { ... }
          // add key to this BST; error if it is already there

        public void delete(K key) { ... }
          // remove the node containing key from this BST if it is there;
          // otherwise, do nothing

        public boolean lookup(K key) { ... }
          // if key is in this BST, return true; otherwise, return false

        public void print(PrintStream p) { ... }
          // print the values in this BST in sorted order (to p)
    }
```

The type parameter `K` is the type of the key. Because most of the BST operations require comparing key values, we declare that `K extends Comparable<K>`, meaning that type `K` must implement a method

```
    int compareTo(K other)
```

To implement a BST that stores some data with each key, we would use the following class definitions (changes are in red):

```
    class BSTnode<K, V> {
        // *** fields ***
        private K key;
        private V value;
        private BSTnode<K, V> left, right;

        // *** constructor ***
        public BSTnode(K key, V value,
                             BSTnode<K, V> left, BSTnode<K, V> right) {
            this.key = key;
            this.value = value;
            this.left = left;
            this.right = right;
        }

        // *** methods ***

        ...
        public V getValue() { return value; }
        public void setValue(V newV) { value = newV; }
        ...
    }

    public class BST<K extends Comparable<K>, V> {
        // *** fields ***
        private BSTnode<K, V> root; // ptr to the root of the BST

        // *** constructor ***
        public BST() { root = null; }

        // *** methods ***

        public void insert(K key, V value) throws DuplicateException {...}
          // add key and associated value to this BST;
          // error if key is already there

        public void delete(K key) {...}
          // remove the node containing key from this BST if it is there;
          // otherwise, do nothing

        public V lookup(K key) {...}
          // if key is in this BST, return its associated value; otherwise, return null

        public void print(PrintStream p) {...}
          // print the values in this BST in sorted order (to p)
    }
```

From now on, we will assume that BSTs only store key values, ***not*** associated data. We will also assume that null is not a valid key value (i.e., if someone tries to insert or lookup a null value, that should cause an exception).

The lookup method

In general, to determine whether a given value is in the BST, we will start at the root of the tree and determine whether the value we are looking for:

- is in the root
- might be in the root's left subtree
- might be in the root's right subtree

There are actually ***two*** base cases:

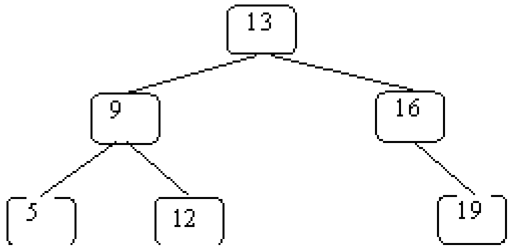
1. The tree is empty; return false.
2. The value is in the root node; return true.

If neither base case holds, a recursive lookup is done on the appropriate subtree. Since all values less than the root's value are in the left subtree and all values greater than the root's value are in the right subtree, there is no point in looking in ***both*** subtrees: if the value we're looking for is less than the value in the root, it can only be in the left subtree (and if it is greater than the value in the root, it can only be in the right subtree).

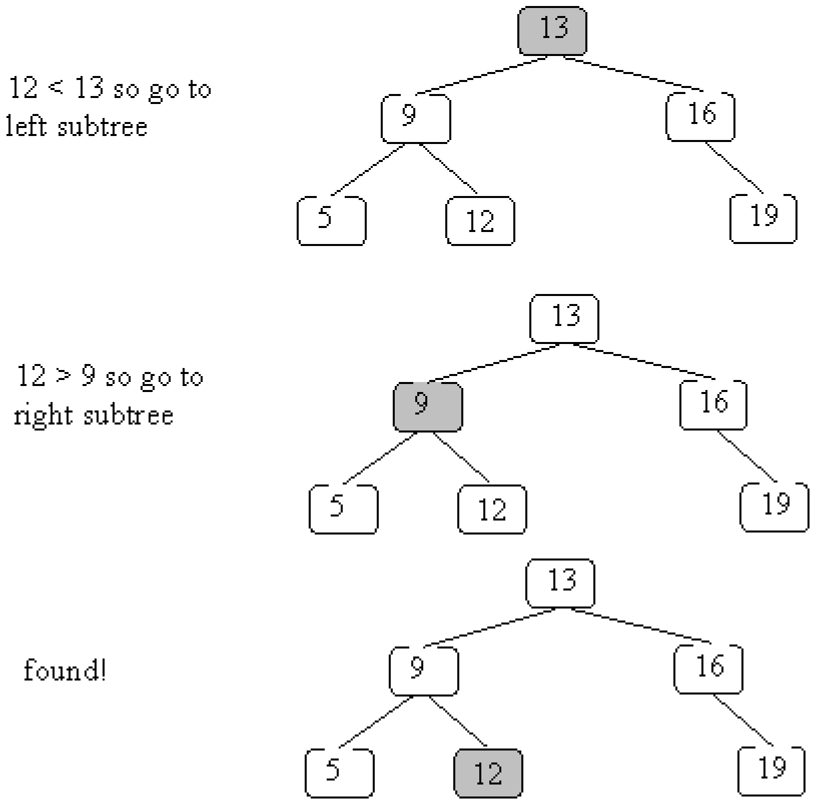
The code for the lookup method uses an auxiliary, recursive method with the same name (i.e., the lookup method is overloaded):

```
    public boolean lookup(K key) {
        return lookup(root, key);
    }



    private boolean lookup(BSTnode<K> n, K key) {
        if (n == null) {
            return false;
        }

        if (n.getKey().equals(key)) {
            return true;
        }

        if (key.compareTo(n.getKey()) < 0) {
            // key < this node's key; look in left subtree
            return lookup(n.getLeft(), key);
        }

        else {
            // key > this node's key; look in right subtree
            return lookup(n.getRight(), key);
        }
    }
```

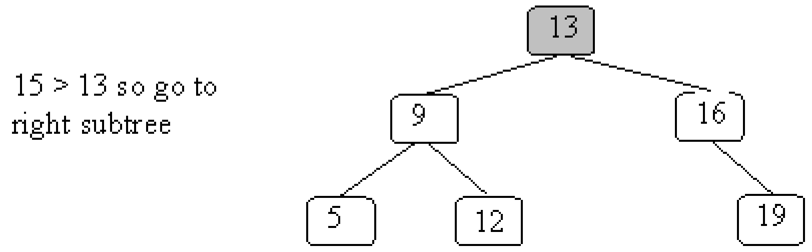
Let's illustrate what happens using the following BST:

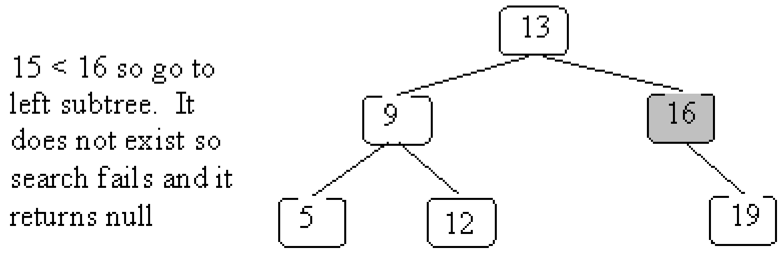


and searching for 12:



What if we search for 15:




15 < 16 so go to left subtree. It does not exist so search fails and it returns null
13
9
16
5
12
19

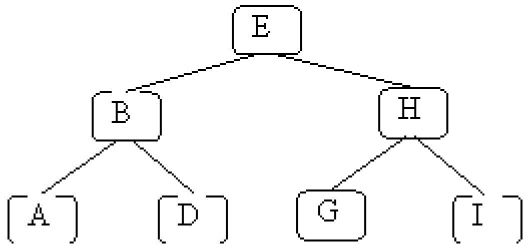
How much time does it take to search for a value in a BST? Note that lookup always follows a path from the root down towards a leaf. In the worst case, it goes all the way to a leaf. Therefore, the worst-case time is proportional to the length of the longest path from the root to a leaf (the height of the tree).

In general, we'd like to know how much time is required for lookup as a function of the number of values stored in the tree. In other words, what is the relationship between the number of nodes in a BST and the height of the tree? This depends on the "shape" of the tree. In the worst case, all nodes have just one child, and the tree is essentially a linked list. For example:

```
     50
    /
  10
    \
     15
       \
        30
       /
     20
```

This tree has 5 nodes and also has height = 5. Searching for values in the range 16-19 and 21-29 will require following the path from the root down to the leaf (the node containing the value 20), i.e., will require time proportional to the number of nodes in the tree.

In the best case, all nodes have 2 children and all leaves are at the same depth, for example:



This tree has 7 nodes and height = 3. In general, a tree like this (a **full** tree) will have height approximately $\log_2(N)$, where N is the number of nodes in the tree. The value $\log_2(N)$ is (roughly) the number of times you can divide N by two before you get to zero. For example:

```
7/2 = 3      // divide by 2 once
3/2 = 1      // divide by 2 a second time
1/2 = 0      // divide by 2 a third time, the result is zero so quit
```

So $\log_2(7)$ is approximately equal to 3.

The reason we use $\log_2$. (rather than say $\log_3$) is because every non-leaf node in a full BST has ***two*** children. The number of nodes in each of the root's subtrees is (approximately) 1/2 of the nodes in the whole tree, so the length of a path from the root to a leaf will be the same as the number of times we can divide N (the total number of nodes) by 2.

However, when we use big-O notation, we just say that the height of a full tree with N nodes is O(log N) -- we drop the "2" subscript, because $\log_2(N)$ is proportional to $\log_k(N)$ for any constant k, i.e., for any constants B and k and any value N:

$$\log_B(N) = \log_k(N) / \log_k(B)$$

and with big-O notation we always ignore constant factors.

To summarize: the worst-case time required to do a lookup in a BST is O(height of tree). In the worst case (a "linear" tree) this is O(N), where N is the number of nodes in the tree. In the best case (a "full" tree) this is O(log N).

The insert method

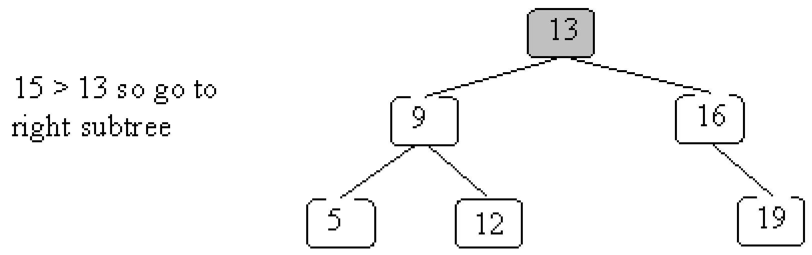
Where should a new item go in a BST? The answer is easy: it needs to go where you would have found it using lookup! If you don't put it there then you won't find it later.

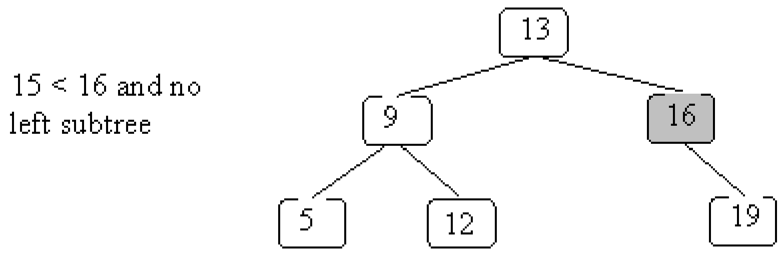
The code for insert is given below. Note that:

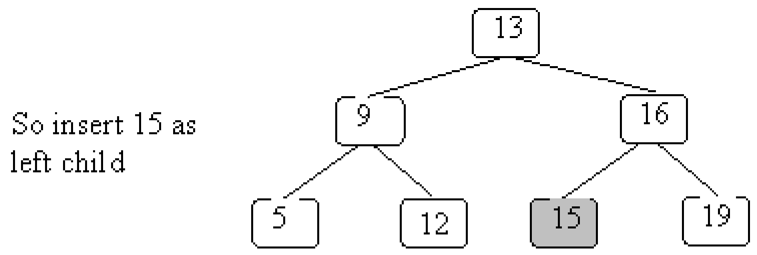
- We assume that duplicates are not allowed (an attempt to insert a duplicate value causes an exception).
- The public insert method uses an auxiliary recursive "helper" method to do the actual insertion.
- The node containing the new value is always inserted as a ***leaf*** in the BST.
- The public insert method returns `void`, but the helper method returns a `BSTnode`. It does this to handle the case when the node passed to it is null. In general, the helper method is passed a pointer to a possibly empty tree. Its responsibility is to add the indicated key to that tree and return a pointer to the root of the modified tree. If the original tree is empty, the result is a one-node tree. Otherwise, the result is a pointer to the same node that was passed as an argument.

```
public void insert(K key) throws DuplicateException {
    root = insert(root, key);
}



private BSTnode<K> insert(BSTnode<K> n, K key) throws DuplicateException {
    if (n == null) {
        return new BSTnode<K>(key, null, null);
    }

    if (n.getKey().equals(key)) {
        throw new DuplicateException();
    }

    if (key.compareTo(n.getKey()) < 0) {
        // add key to the left subtree
        n.setLeft( insert(n.getLeft(), key) );
        return n;
    }

    else {
        // add key to the right subtree
        n.setRight( insert(n.getRight(), key) );
        return n;
    }
}
```

Here are pictures illustrating what happens when we insert the value 15 into the example tree used above.







It is easy to see that the complexity for insert is the same as for lookup: in the worst case, a path is followed all the way to a leaf.

TEST YOURSELF #2

As mentioned above, the order in which values are inserted determines what BST is built (inserting the same values in different orders can result in different final BSTs). Draw the BST that results from inserting the values 1 to 7 in each of the following orders (reading from left to right):

1. 5 3 7 6 2 1 4
2. 1 2 3 4 5 6 7
3. 4 3 5 2 6 1 7

solution

The delete method

As you would expect, deleting an item involves a search to locate the node that contains the value to be deleted. Here is an outline of the code for the delete method.

```
public void delete(K key) {
    root = delete(root, key);
}

private BSTnode<K> delete(BSTnode<K> n, K key) {
    if (n == null) {
        return null;
    }

    if (key.equals(n.getKey())) {
       // n is the node to be removed
       // code must be added here
    }

    else if (key.compareTo(n.getKey()) < 0) {
        n.setLeft( delete(n.getLeft(), key) );
        return n;
    }

    else {
        n.setRight( delete(n.getRight(), key) );
        return n;
    }
}
```

There are several things to note about this code:

- As for the lookup and insert methods, the BST delete method uses an auxiliary, overloaded delete method to do the actual work.
- If k is not in the tree, then eventually the auxiliary method will be called with `n == null`. That is not considered an error; the tree is simply unchanged in that case.
- The auxiliary delete method returns a value (a pointer to the updated tree). The reason for this is explained below.

If the search for the node containing the value to be deleted succeeds, there are three cases to deal with:

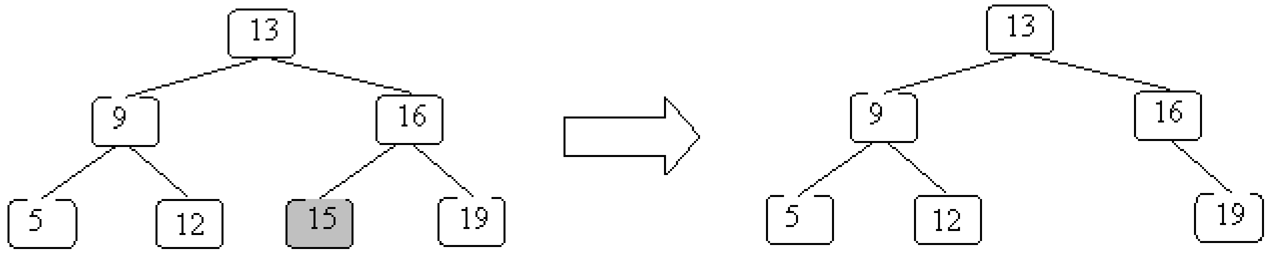
1. The node to delete is a leaf (has no children).
2. The node to delete has one child.
3. The node to delete has two children.

When the node to delete is a leaf, we want to remove it from the BST by setting the appropriate child pointer of its parent to null (or by setting root to null if the node to be deleted is the root and it has no children). Note that the call to delete was one of the following:

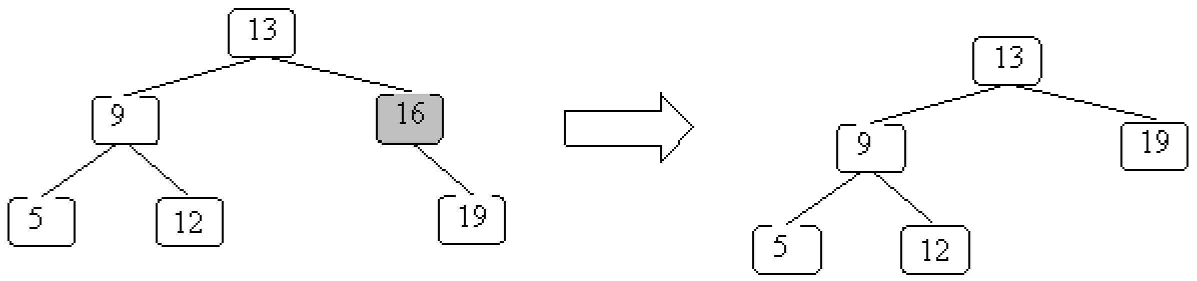
- `root = delete(root, key);`
- `n.setLeft( delete(n.getLeft(), key) );`
- `n.setRight( delete(n.getRight(), key) );`

So in all three cases, the right thing happens if the delete method just returns null.

Here's what happens when the node containing the value 15 is removed from the example BST:



When the node to delete has one child, we can simply replace that node with its child by returning a pointer to that child. As an example, let's delete 16 from the BST just formed:



Here's the code for delete, handling the two cases we've discussed so far (the new code is shown in red):

```
private BSTnode<K> delete(BSTnode<K> n, K key) {
    if (n == null) {
        return null;
    }

    if (key.equals(n.getKey())) {
        // n is the node to be removed
        if (n.getLeft() == null && n.getRight() == null) {
            return null;
        }
        if (n.getLeft() == null) {
            return n.getRight();
        }
        if (n.getRight() == null) {
            return n.getLeft();
        }

        // if we get here, then n has 2 children
        // code still needs to be added here...
    }

    else if (key.compareTo(n.getKey()) < 0) {
        n.setLeft( delete(n.getLeft(), key) );
        return n;
    }

    else {
        n.setRight( delete(n.getRight(), key) );
        return n;
    }
}
```

The hard case is when the node to delete has two children. We'll call the node to delete n. We can't replace node n with one of its children, because what would we do with the other child? Instead, we will replace the key in node n with the key value v from another node lower down in the tree, then (recursively) delete value v.

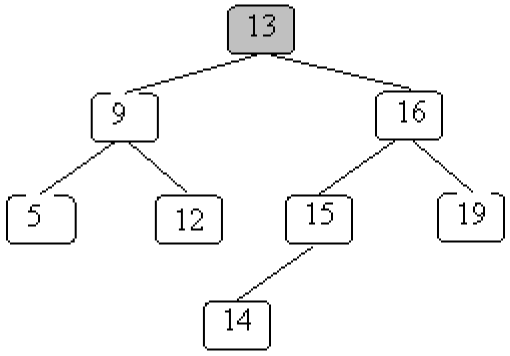
The question is what value can we use to replace n's key? We have to choose that value so that the tree is still a BST, i.e., so that all of the values in n's left subtree are less than the value in n, and all of the values in n's right subtree are greater than the value in n. There are two possibilities that work: the ***largest*** value in n's left subtree or the ***smallest*** value in n's right subtree. We'll arbitrarily decide to use the smallest value in the right subtree.

To find that value, we just follow a path in the right subtree, always going to the ***left*** child, since smaller values are in left subtrees. Once the value is found, we copy it into node n, then we recursively delete that value from n's right subtree. Here's the final version of the delete method:

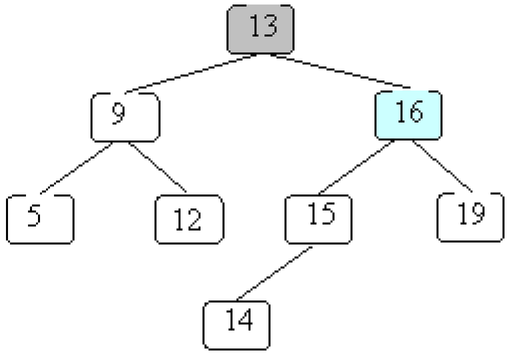
```
private BSTnode<K> delete(BSTnode<K> n, K key) {
    if (n == null) {
        return null;
    }

    if (key.equals(n.getKey())) {
        // n is the node to be removed
        if (n.getLeft() == null && n.getRight() == null) {
            return null;
        }
        if (n.getLeft() == null) {
            return n.getRight();
        }
        if (n.getRight() == null) {
            return n.getLeft();
        }

        // if we get here, then n has 2 children
        K smallVal = smallest(n.getRight());
        n.setKey(smallVal);
        n.setRight( delete(n.getRight(), smallVal) );
        return n;
    }

    else if (key.compareTo(n.getKey()) < 0) {
        n.setLeft( delete(n.getLeft(), key) );
        return n;
    }

    else {
        n.setRight( delete(n.getRight(), key) );
        return n;
    }
}
```

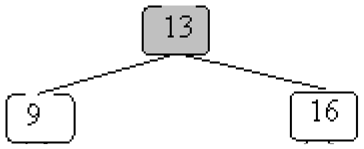
Below is a slightly different example BST; let's see what happens when we delete 13 from that tree.

Original BST with 13 located

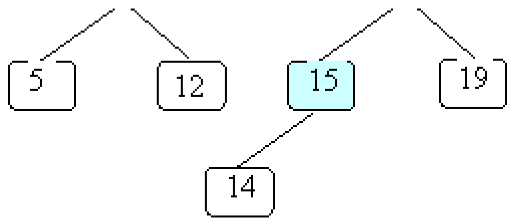


Step into right subtree.

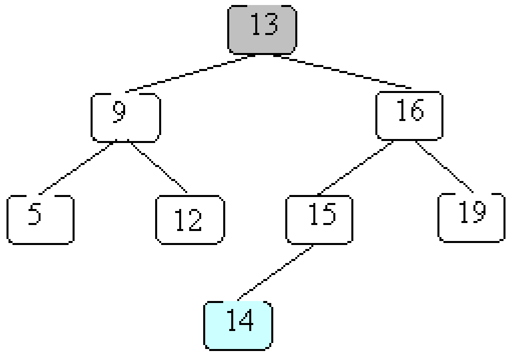




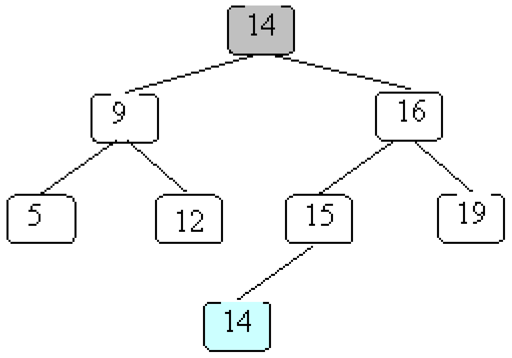
Go to left child.



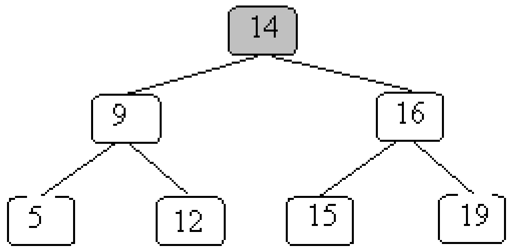
Continue to left child. This is last one.



Replace node to delete with far left child of right subtree.



Remove far left child of right subtree.



TEST YOURSELF #3

Write the auxiliary method **`smallest`** used by the delete method given above. The header for `smallest` is:

```
private K smallest(BSTnode<K> n)
// precondition: n is not null
// postcondition: return the smallest value in the subtree rooted at n
```

solution

What is the complexity of the BST delete method?

If the node to be deleted has zero or one child, then the delete method will "follow a path" from the root to that node. So the worst-case time is proportional to the height of the tree (just like for lookup and insert).

If the node to be deleted has two children, the following steps are performed:

1. Find the node to be deleted (follow the path from the root to that node).

2. Find the smallest value v in the right subtree (continue down the path toward a leaf).
3. Recursively delete value v (follow the same path followed in step 2).

So in the worst case, a path from the root to a leaf is followed twice. Since we don't care about constant factors, the time is still proportional to the height of the tree.

Maps and Sets

The Java standard library has built into it an industrial-strength version of binary search trees, so if you are programming in Java and need this functionality, you would be better off using the library version rather than writing your own. The class that most closely matches the outline above, in which the nodes contain only keys and no other data, is called `TreeSet`. Class `TreeSet` is an implementation of the `Set` interface. (There is another implementation, called `HashSet`, that we will study later in this course.) Here's an example of how you might use a `Set` to implement a simple spell-checker.

```
Set<String> dictionary = new TreeSet<String>();
Set<String> misspelled = new TreeSet<String>();
// Create a set of "good" words.
while (...) {
    String goodWord = ...;
    dictionary.add(goodWord);
}

// Look up various other words
while (...) {
    String word = ...;
    if (! dictionary.contains(word)) {
        misspelled.add(word);
    }
}

// Print a list

if (misspelled.size() == 0) {
    System.out.println("No misspelled words!");
} else {
    System.out.println("Misspelled words are:");

    for (String word : misspelled) {
        System.out.println("    " + word);
    }
}
```

This example used a set of `String`. You could also have a set of `Integer`, a set of `Float`, or a set of any other type of object, so long as the type `implements Comparable`. For example,

```
public class Employee implements Comparable<Employee> {
    private int employeeNumber;
    private String firstName;
    private String lastName;
    private float salary;

    public int compareTo(Employee that) {
        return this.employeeNumber - that.employeeNumber;
    }
    // ... various other methods
}

public class HumanResources {
    private Set<Employee> staff;
```

```
        public void hire(Employee recruit) {
            staff.add(recruit);
        }
        // ... etc.
    }
```

If you want to associate data with each key, use interface Map and the corresponding class `TreeMap`. A `Map<K,V>` associates a value of type `V` with each key of type `K`. For example, if you want to quickly look up an `Employee` given his employee number, you should use a `Map` rather than a `Set` to keep track of employees.

```
    // the following data member and methods are inside a class definition
    private Map<Integer, Employee> staff = new TreeMap<Integer, Employee>();

    public Employee getEmployee(int number) {
        return staff.get(number);
    }

    public void addEmployee(int number, Employee emp) {
        staff.put(number, emp);
    }
```

As another example, here is a complete program that counts the number of occurrences of each word in a document.

```
    import java.util.*;
    import java.io.*;

    public class CountWords {
        public static void main(String[] args) throws Exception {
            if (args.length != 1) {
                System.err.println("usage: java CountWords file_name");
                return;
            }

            Map<String, Integer> wordCount = new TreeMap<String, Integer>();
            Scanner in = new Scanner(new File(args[0]));
            in.useDelimiter("\\W+");
            while (in.hasNext()) {
                String word = in.next();
                Integer count = wordCount.get(word);
                if (count == null) {
                    wordCount.put(word, 1);
                } else {
                    wordCount.put(word, count + 1);
                }
            }
            for (String word : wordCount.keySet()) {
                System.out.println(word + " " + count.get(word));
            }
        }
    } // CountWords
```

(The statement `in.useDelimiter("\\W");` tells the `Scanner` that words are delimited by non-word characters. Without it, the program would look for "words" separated by spaces, considering "details" and "details.)" to be (different) words. See the documentation for `Scanner` and `Pattern` for more details.)

The value type `V` can be any class or interface. The key type `K` can be any class or interface that implements `Comparable`, for example,

```
    class MyKey implements Comparable<MyKey> {
        // ...
        public int compareTo(MyKey other) {
            // return a value < 0 if this key is less than other,
            // == 0 if this key is equal to other, and
            // > 0 if this key is greater than other, and
        }
    }
```

The method `put(key, value)` returns the value previously associated with `key` if any or `null` if `key` is a new key. The method `get(key)` returns the value associated with `key` or `null` if there is no such value. Both keys and values can be `null`. If your program stores `null` values in the map, you should use `containsKey(key)` to check whether a particular key is present.

`Map` has many other useful methods. Of particular note are `size()`, `remove(key)`, `clear()`, and `keySet()`. The `keySet()` method returns a `Set` containing all the keys currently in the map. The `CountWords` example uses it to list the words in the document.

Summary

A binary search tree can be used to store any objects that implement the `Comparable` interface (i.e., that define the `compareTo` method). A BST can also be used to store `Comparable` objects plus some associated data. The advantage of using a binary search tree (instead of, say, a linked list) is that, if the tree is reasonably balanced (shaped more like a "full" tree than like a "linear" tree), the insert, lookup, and delete operations can all be implemented to run in O(log N) time, where N is the number of stored items. For a linked list, although insert can be implemented to run in O(1) time, lookup and delete take O(N) time in the worst case.

Logarithmic time is generally ***much*** faster than linear time. For example, for N = 1,000,000: $\log_2 N = 20$.

Of course, it is important to remember that for a "linear" tree (one in which every node has one child), the worst-case times for insert, lookup, and delete will be O(N).

Back

Next: Continue with Red Black Trees