

CS 228: Introduction to Data Structures

Lecture 27

Wednesday, April 1, 2015

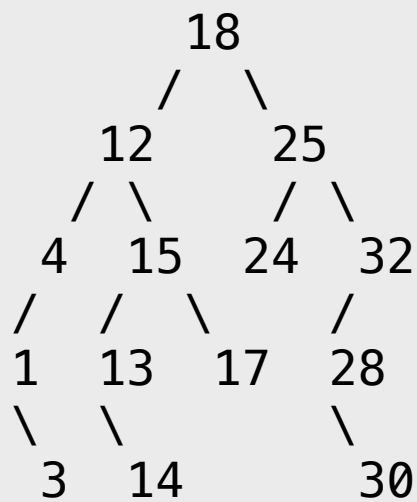
Binary Search Trees

Suppose we want to represent sets where the element type is Comparable. That is, the elements have a natural ordering, so that there is a minimum element, a maximum element, and any element, other than the min and max, has a successor (next) and a predecessor (previous). In sets like this, the elements are often called **keys**.

A **binary search tree** (BST) is one way to implement a set of comparable elements. A BST is a binary tree whose nodes hold the keys; the keys must satisfy the following.

Binary Search Tree Property: For any node X , every key in the left subtree of X is less than X 's key, and every key in the right subtree of X is greater than X 's key.

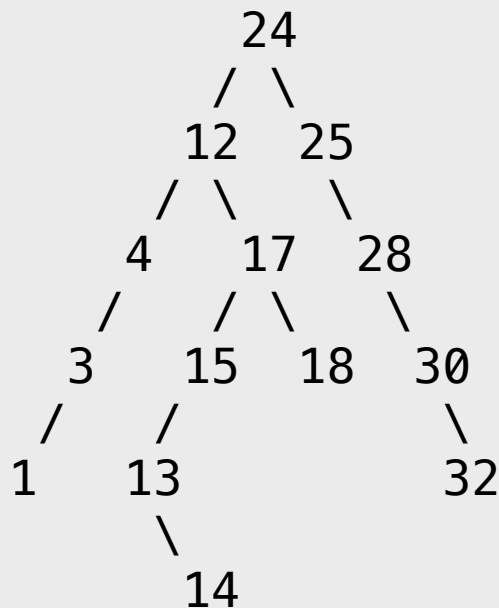
Example. Consider the BST below.



For instance, the root is 18, its left subtree (rooted at 12) contains numbers from 1 to 17, and its right subtree (rooted at 25) contains numbers from 24 to 30.

When a node has only one child, that child is either a left child or a right child, depending on whether its key is smaller or larger than its parent's key.

Note. The BST representation of a set is not unique. Here's another way to represent the set of keys from the previous example:



You should be able to verify the following:

Fact. 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. However, operations on a binary search tree are usually much faster than the same operations on a sorted linked list because you typically only traverse a small fraction of the nodes of a tree. Before explaining why, we need to provide a few implementation details.

Representing a Set as a BST

Our implementation of sets via BSTs builds on Java's `AbstractSet`, an abstract class that extends `AbstractCollection`, providing a partial implementation of the `Set` class. The details of the tree representation are hidden within the `Node` inner class. A `Node` is like `TreeNode`, except that, in addition to references to the left and right children, it has a parent reference, which is used to implement removal and iteration efficiently.

```
public class
BSTSet<E extends Comparable<? super E>>
extends AbstractSet<E>
{
    protected Node root;
    protected int size;

    protected class Node
    {
        public Node left;
        public Node right;
        public Node parent;
        public E data;
    }
}
```

```
public Node(E key, Node parent)
{
    this.data = key;
    this.parent = parent;
}
}
```

Searching for a Key in a BST

To look for a key k in a tree, we could iterate through its nodes using an inorder traversal, stopping when we either find the key, or we run out of elements to look at. In fact, the implementation in `AbstractSet` actually uses an iterator to implement `contains()`.

A faster way is to exploit the BST property to go down just one path in the tree, starting at the root. By examining the key c in the current node, we can decide whether to

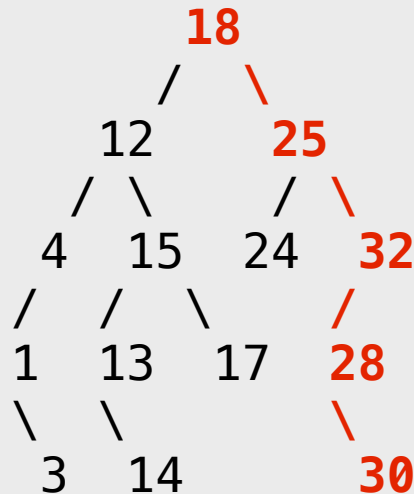
- (a) **stop**, if c is the same as k ,
- (b) look for k in the **left subtree**, if k is less than c , or
- (c) look for k in the **right subtree**, if k is greater than c .

This is like traversing a linked list, but with a conditional statement to decide which way to go. The algorithm is implemented in the `findEntry()` method below.

```
protected Node findEntry(E key)
{
    Node current = root;
    while (current != null)
    {
        int comp =
            current.data.compareTo(key);
        if (comp == 0)
        {
            return current;
        }
        else if (comp > 0)
        {
            current = current.left;
        }
        else
        {
            current = current.right;
        }
    }
    return null;
}
```

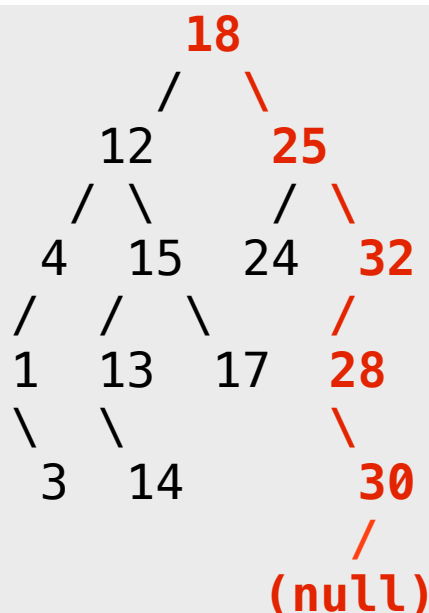
Note the use of `compareTo()`, because type `E` extends `Comparable<? super E>`.

Example. Here's the path `findEntry()` follows in a search for key 30 on the first of the two preceding trees.



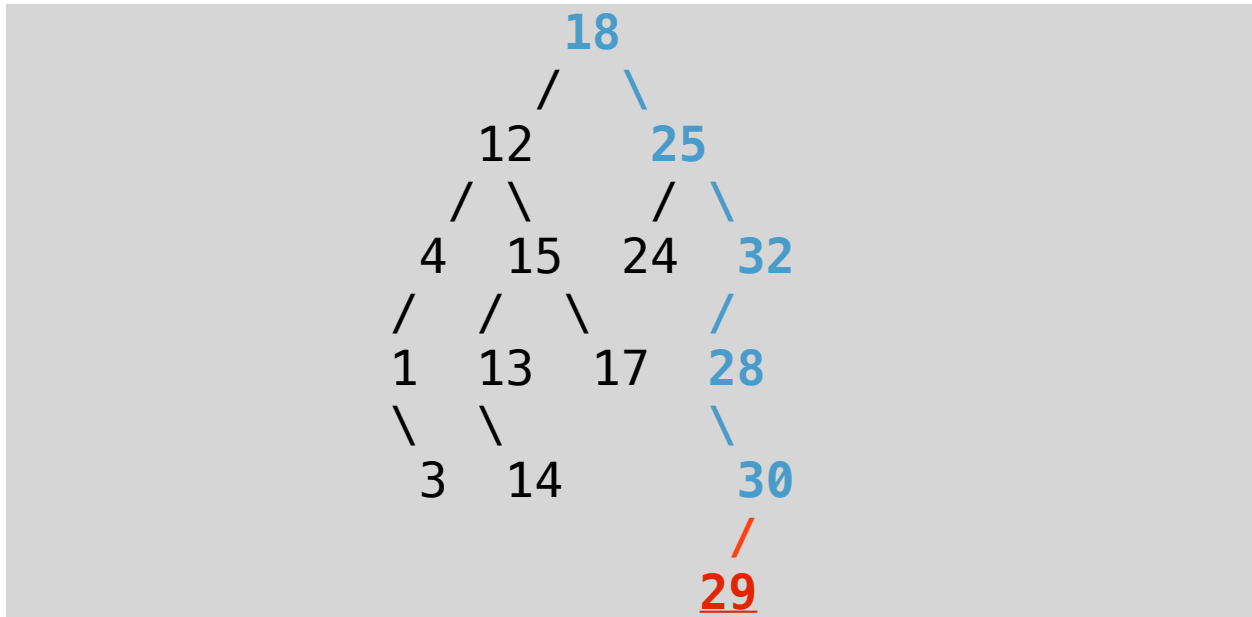
Note that we only had to look at 5 keys, instead of the 12 keys that precede 30.

An unsuccessful search ends up in an empty subtree. For example, here's a search for 29.



Adding a Key to a BST

Suppose we want to **add** 29 to the tree in the preceding example. We can just put it in the empty slot where our unsuccessful search ended:



In general, to add a key k , we do as follows.

- (1) Search for k in the tree.
- (2) If k is found, return `false`, indicating that we could not add it, since it's already in the set.
- (3) If k is not found, add it in the empty slot where the unsuccessful search ended (and return `true`).

To convince yourselves that this works, try adding some more keys — e.g., 2 or 26 — into the tree above.

Implementation Details

To conform to the specifications of the Java Set interface, the implementations of `add()` and `contains()` must address a few technical details. We explain these details next.

`contains()`

The `findEntry()` method does most of what we need to implement `contains()`. There is, however, one issue to consider. It is tempting to write the method as follows.

```
public boolean contains(Object obj)
{
    return findEntry(obj) != null;
}
```

Unfortunately, this will not compile: as far as Java is concerned, `obj` is not of type `E`. This seems to leave us in a bind. On the one hand, to apply `compareTo()`, we must use type `E` within `findEntry()`. On the other, the

Set API requires the argument of `contains()` to be `Object`. The solution is to add an unsafe cast. The cast is itself meaningless, but it allows the code to compile.

```
public boolean contains(Object obj)
{
    E key = (E) obj;
    return findEntry(key) != null;
}
```

If the runtime type of `obj` is incompatible with type `E`, the `compareTo` method will throw a `ClassCastException`, which is exactly the behavior specified for the Set API.

`add()`

The `add()` method begins by trying to find the key we're trying to add. If we *do* find it, then the method should return `false` — remember, there can be no duplicates. Otherwise, our search must have ended in the empty slot where the new key should be inserted. We insert a new node with the given key in this slot and return `true`. The code is on the next pages.

```
public boolean add(E key)
{
    if (root == null)
    {
        root = new Node(key, null);
        ++size;
        return true;
    }

    Node current = root;

    while (true)
    {
        int comp =
            current.data.compareTo(key);
        if (comp == 0)
        {
            // key is already in the tree
            return false;
        }
    }
}
```

```

        else if (comp > 0)
        {
            if (current.left != null)
            {
                current = current.left;
            }
            else
            {
                current.left =
                    new Node(key, current);
                ++size;
                return true;
            }
        }
        else
        {
            if (current.right != null)
            {
                current = current.right;
            }
            else
            {
                current.right =
                    new Node(key, current);
                ++size;
                return true;
            }
        }
    }
}

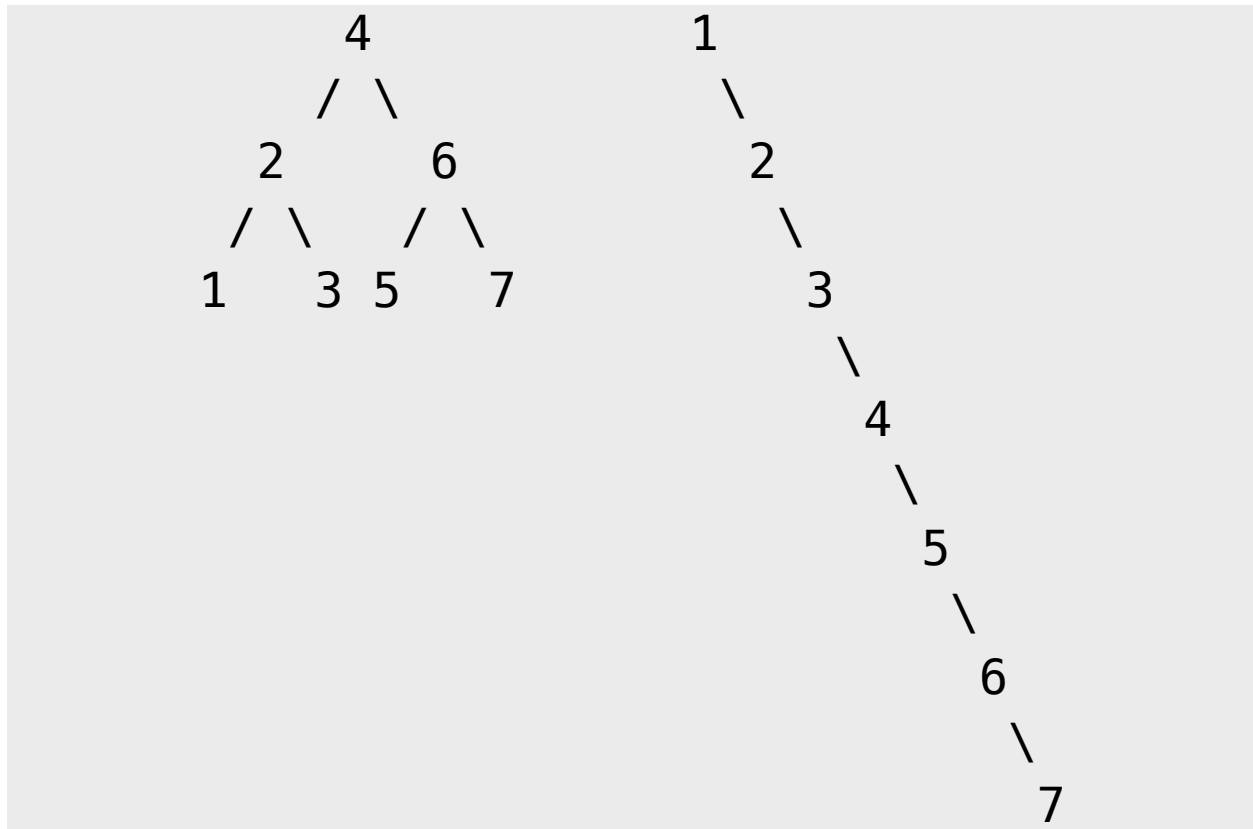
```

The Time Complexity of Searching and Insertion

Searching for a key and adding a new key require following a path from the root, doing a comparison at each step. Suppose we make the (very reasonable) assumptions that a key comparison takes $O(1)$ time and that following a pointer and linking in a single node also take $O(1)$ time. Then, the time complexity to search for a key or add a key is proportional to the length of the path from the root that is followed when executing these operations. Since the longest path from the root in a tree T has $\text{height}(T) + 1$ nodes, we have the following.

Fact. The worst-case time needed to search for a key or to add a key to a BST T is $O(\text{height}(T))$.

As we will see, the time to delete a key is also $O(\text{height}(T))$. Note that the height of a tree can vary considerably. For example, here are the minimum- and maximum-height trees on 7 keys.



Let n be the number of nodes in the BST. In general, there are two extreme cases:

- **Best case: well-balanced tree.** At each node, the number of nodes in the left subtree is roughly the same as the number on the right subtree. Then, the height is $O(\log n)$ and so is the time complexity of the operations.
- **Worst case: very unbalanced (degenerate) tree.** Each node has only one child. Then, the height of the tree is $n-1$ and the operations are $O(n)$.

The actual height will fall somewhere in between. It can be proved mathematically that, under certain conditions, the *expected*¹ height is $O(\log n)$, so the expected running time is $O(\log n)$. In practice, this kind of argument is unsatisfactory, and we would prefer a performance *guarantee*. Fortunately, there are more advanced BSTs with clever extra features to keep them from getting too far out of balance, ensuring $O(\log n)$ time for all operations — AVL trees² and red-black trees³ are two examples. Indeed, TreeSet, the Java library implementation of a sorted set, uses red-black trees. Next week, we will see splay trees, a kind of “self-adjusting” BST where the *amortized time* of all operations is $O(\log n)$.

¹ For a formal definition of expected value, see http://en.wikipedia.org/wiki/Expected_value.

² http://en.wikipedia.org/wiki/AVL_tree. For a nice demo of AVL trees, see <http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html>.

³ http://en.wikipedia.org/wiki/Red%E2%80%93black_tree