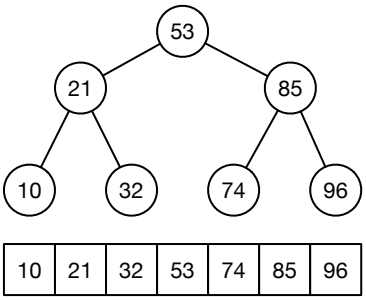
Lecture 14: Binary (Search) Trees II

built on 2016/02/25 at 13:54:57

1 Binary Search Tree: Not Just Any Binary Tree

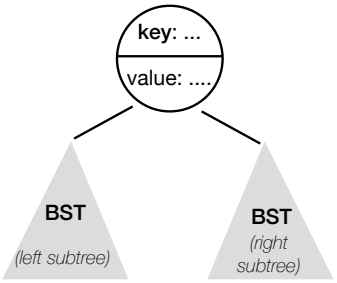
Consider the following sorted array and a tree that we superimpose on:

This figure shows a perfect binary tree on 7 nodes. These nodes coincidentally (or perhaps not) are labeled according to numbers from the sorted array below the tree. In many ways, this should remind us of binary search. When we look up a key using binary search, the walk from root to that key in the tree is exactly the comparisons we make in binary search.



In general, our collection is more dynamic than a fixed sorted list: there will be new items getting added, existing items getting removed, etc. One big motivating question for today's lecture is: *How can we maintain, perhaps implicitly, a sorted collection while supporting insertion, deletion, and search efficiently?*

A *binary tree* is a tree in which every node in the tree has at most two children. Binary search trees (BSTs) are binary-tree-based data structures that can be used to store and search for items that satisfy a total order. There are many types of search trees designed for a wide variety of purposes. The most common use is perhaps to implement sets and tables (dictionaries, mappings). For a bit of history, BSTs date back to around 1960, usually credited to Windley, Booth, Colin, and Hibbard.



We generally work with the assumption that the keys are unique. A *binary search tree* (BST) can be defined recursively as

(a) an empty tree; or

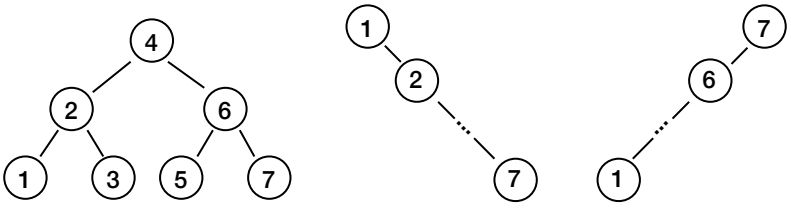
(b) a node storing a key-value pair $(key, value)$, together with two BSTs, known as the left and right subtrees, where the left subtree contains only keys that are smaller than *key* and the right subtree contains only keys that are larger than *key*.

Therefore, a binary search tree satisfies the following invariant:

> For any node v, all of the left subtree is smaller than the key at v, which is smaller still than all of the right subtree.

In a standard implementation, one keeps the following attributes for each node: `key`, `value`, `left`, and `right`, which represent, respectively, the key, the value, the (reference to the) left child, and the (reference to the) right child.

Example: We give a few examples of binary search trees on the keys $1, 2, \ldots, 7$, omitting their values. Notice that some of these trees may be lopsided, some completely balanced, as the BST definition doesn't prescribe any exact shape.



1.1 Representing BST Nodes

Let's first work with BST nodes assuming that both the key and the value are integers. In this case, we'll just need a class that keeps two integers—let's call them `key` and `value`—as well as the two children.

```
public class BST {
  int key;
  int value;
  BST left, right;
}
```

In many cases, we want our implementation to be more general and support a wide variety of types. In this case, the class will be declared as `BST<K, V>` with the intent that `K` is the key type and `V` is the value type. It is necessary that the key type is `Comparable` because otherwise we won't be able to make comparisons and navigate the tree. Hence, the declaration will have to require that, like so:

```
public class BST<K extends Comparable<? super K>, V> {
  K key;
  V value;
  BST<K,V> left, right;
}
```

1.2 Working Directly With Binary Search Trees

We consider performing two simple tasks on binary search trees. Despite their simplicity, these examples will help acquaint us with properties of the BST.

Searching for a given key. In this routine task, we're given a key k and we are asked to retrieve the value associated with that key or report that the key doesn't exist. Because a binary search tree maintains strict ordering of keys, when we compare k with the key at a node, we know right away which branch—left or right—to take next. For example, suppose v.`key` $> k$ at a node v. Then, we know that if k exists in the tree rooted at v, k must be in the subtree v.`left` since all keys smaller than v.`key` belong in the left subtree. This reasoning yields the following algorithm (assuming the `BST` has integer key and value):

```
public V search(K k) {
  int r = this.key.compareTo(k);

  if      (r==0)                return this.value;
  else if (r<0 && right!=null) return right.search(k);
  else if (r>0 && left!=null)  return left.search(k);
  else                         return null;
}
```

More generally, we might deal with `BSTNode` whose types are given by type parameters. This is left as an exercise.

Finding the largest key. Let's pause for a moment and think about how one might locate the largest key in the tree. A moment's thought shows that the largest key lies at the bottom-right tip of the tree. This is because if there's anything bigger than the root, it must be in the root's right subtree. And inside that subtree, if there's anything bigger than its root, it must be in that root's right subtree. But if at any point, that subtree no longer has a right subtree (it's empty), the root of that subtree itself is the biggest key in that subtree. Following this line of reasoning, we arrive at the following algorithm (once again assuming integer keys and values):

```
public K max() {
  if (right!=null) return right.max();
  else             return key;
}
```

Remarks: A common pattern so far—and one that will be recurring throughout the discussion of BSTs—is that the performance of operations on a BST depends largely on the height of the tree. In both the `search` and `max` algorithms, the running time is proportional to the length of the path that the algorithm traverses, which is never longer than the tree's height. Therefore, we strive to keep the height small.

2 An Abstraction for BSTs

Various techniques have been suggested in the literature to keep the tree height small to ensure good performance. Because of this, we would like a common abstraction that once provided, allows us to implement many tree algorithms easily—without having to worry too much about the particulars of these height-controlling techniques. Unlike the traditional treatment of binary search trees, we'll use these basic operations to imply standard operations such as insertion, deletion, and search.

- `empty()` — create an empty tree
- `singleton(key, value)` — create a tree with exactly one node containing the key `key` and its corresponding value `value`.
- `T.split(k)` — take a tree and a key, and return the following triplet `smaller, match, larger`, where (1) `smaller` is a BST containing all the keys strictly smaller than `k`, (2) `match` contains the value corresponding to `k` if `k` exists in the tree, and (3) `larger` is a BST containing all the keys strictly larger than `k`.
- `T1.join(T2)` — operate on the assumption that all of `T1` is smaller than all of `T2` (key-wise) and produce a tree that is the union of `T1` and `T2`.

Basic operations via `split` and `join`: We show how to implement basic tree operations `find`, `insert`, `remove` using `split` and `join`.

```
def find(T, k):
  smaller, match, larger = T.split(k)
  return match

def insert(T, k, v):
  smaller, match, larger = T.split(k)
  return smaller.join(singleton(k,v)).join(larger)

def remove(T, k):
  smaller, match, larger = split(T, k)
  return smaller.join(larger)
```

Implementing `pred`: To further demonstrate versatility of this abstraction, we consider implementing the predecessor operation using it. This works as follows:

```
def pred(T, x):
  smaller, match, larger = T.split(x)
  if match!=null: return x
  else: return smaller.max()
```

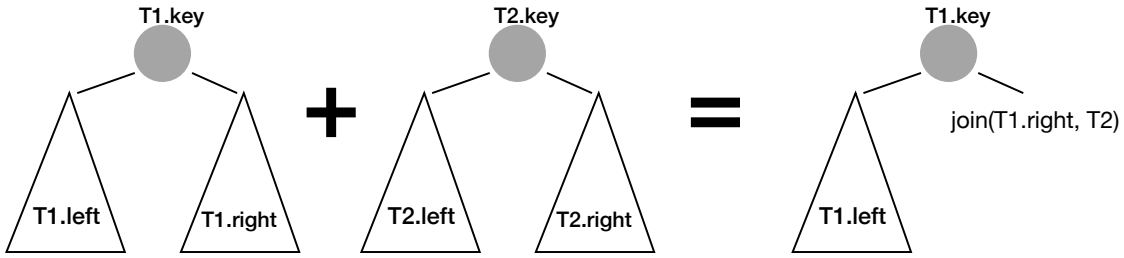
We just saw a way to organize data in a binary tree so that it's easy to navigate to the key of interest. We also saw an abstraction that shields us from details such as how to keep the tree nice and bushy. We have two main goals next:

- We want to implement that abstraction. We'll see how to support the operations we promised to provide.
- We want to improve that implementation so the the trees are always nice and bushy with good probability.

3 Implementation of `split` and `join`

While the operations `empty` and `singleton` are straightforward to support, the operations `split` and `join` are more interesting, which we now discuss. For this discussion, we assume that we can create a new BSTNode with `link(left,k,v,right)`. Therefore, as an example, the call `link(null, key, value, null)` creates a singleton node with the key `key` and value `value`.

The join operation: We present a recursive implementation of `join`(T_1, T_2)—which joins trees T_1 and T_2 together. The simplest case is when T_1 is empty because the resulting tree is simply T_2. If, however, T_1 is not empty, since all of T_1 is smaller than anything in T_2, we have that all of $T_1.left$, $T_1.key$, and all of $T_1.right$ are smaller than any key in T_2. Hence, we can join the trees as depicted in the diagram below:



We can implement this idea as follows:

```
public BST<K, V> join(BST<K,V> other) {
  return link(this.left, this.key, this.value, right.join(other));
}
```

The split operation: We now consider the operation opposite to `join`. We describe a recursive implementation of the operation `split(T, k)`. Once again, splitting an empty tree is trivial. For a non-empty tree, there are three cases to consider depending on the outcome of comparing $T.key$ with k. The easiest case is when $T.key == k$: it follows directly from the ordering property of BST that we should return T's left subtree, T's value, and T's right subtree as the three parts, respectively.

Let's consider the case when $T.key > k$, as the other case is symmetric. If $T.key > k$, we know that both T's key and T's right subtree will fall into the larger-than-k group whereas T's left subtree needs to be further worked on. To work on this subtree, we recursively call `split(T.left, k)`. By definition, if (smaller, match, larger) = `split(T.left, k)`, then we would like to return, as our answer to `split(T, k)`, the following tuples: smaller, match, `TreeNode(larger, T.key, T.value, T.right)`. Hence, we have the following Java code:

```
class SplitReturn {
  BST<K,V> smaller;
  V match;
  BST<K,V> larger;

  public SplitReturn(BST<K,V> sm_, V m_, BST<K,V> lg_) {
    smaller=sm_; match=m_; larger=lg_;
```

```
  }
}

private SplitReturn split(BST<K, V> T, K k) {
  if (T==null) return new SplitReturn(null, null, null);
  else         return T.split(k);
}
private BST<K,V> link(BST<K,V> l, K k, V v, BST<K,V> r) {
  return new BST<>(l,k,v,r);
}
public SplitReturn split(K k) {
  int r = key.compareTo(k);

  if      (r==0) return new SplitReturn(left, value, right);
  else if (r > 0) {
    SplitReturn leftSplit = split(left, k);
    return new SplitReturn(leftSplit.smaller, leftSplit.match,
                           link(leftSplit.larger, key, value, right));
  } else {
    SplitReturn rightSplit = split(right, k);
    return new SplitReturn(link(left, key, value, rightSplit.smaller),
                           rightSplit.match, rightSplit.larger);
  }
}
```