Lecture 14: Binary (Search) Trees II

built on 2019/06/05 at 08:55:50

1 Easy Operations Sorted Collections

Consider the following operations:

- `lowerKey(K key)`: returns the greatest key strictly less than the given key.
- `floorKey(K key)`: returns the greatest key less than or equal to the given key.
- `ceilingKey(K key)`: returns the least key greater than or equal to the given key.
- `higherKey(K key)`: returns the least key strictly greater than the given key.

Given a sorted sequence, all these operations can be supported quickly in $O(\log n)$ using, for example, some variants of binary search.

What if... we also want to support `add` (addition of a new element) and `remove` (removal of an element)? We want **all** of these operations to run quickly as well. *Is this even possible?*

The bulk of this lecture will be devoted to discussing how to support these operations. For now, let's review the idea of a map—or in Python speak, a dictionary.

1.1 Map: HashMap vs. TreeMap

The basic function of a map is to store a mapping between keys and values. More precisely, it remembers for each key of interest, the value the key corresponds to. Hence, the basic operations supported by a map are:

- `get(k)`: returns the value corresponding to the key `k`
- `put(k, v)`: tells the map to associate the value `v` with the key `k`.

Both the `HashMap` and the `TreeMap` support these basic operations.

- `HashMap` supports them in $O(1)$ time for reasons you'll see later.
- `TreeMap`, however, supports them in $O(\log n)$ time, where n is the number of keys in the collection.

So... *why on earth would one choose to use a TreeMap over a HashMap?* Of course, if all we care about are `get` and `put`, then the `HashMap` will be the implementation of choice. However, you may recall that there are the family of operations such as `lowerKey`, `floorKey`, `higherKey`, etc. that we often need.

At a high level:

- The `HashMap` doesn't care about the relative ordering of elements; it focuses exclusively on the task of storing and retrieving a particular key.
- The `TreeMap` understands the relative ordering of the elements, thereby being able to answer order-related queries (such as a nearby element).

In Java, there are two interfaces that capture these operations: `NavigableMap` and `SortedMap`[1]. In addition to what has been mentioned, you can find out about other operations on such collections by browsing the documentation.

1.2 Set: HashSet vs. TreeSet

While a `Map` maintains association between keys and values, we sometimes only care about keys—not at all the values associated with the keys or when the keys don't really have any value associated with them. In these cases, we're interested in keeping a set.

The `Set` interface in Java is implemented by both the HashSet and the TreeSet, with HashSet supporting each operation in $O(1)$ time. However, as with HashMap vs. TreeMap, the TreeMap supports order-related queries in $O(\log n)$ time.

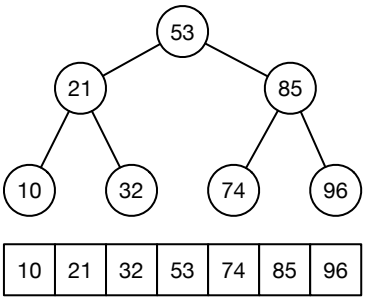
[1] Incidentally, `NavigableMap` "extends" the `SortedMap` interface, adding a few things for navigating the map.

The rest of the lecture today will be dedicated to answering the question, *how can we efficiently support TreeMap?* Sadly, we can't really answer this question fully in one lecture, but the ideas presented here will be useful in some other contexts as well and should give us enough intuition for how things work under the hood.

2 Binary Search Tree: Not Just Any Binary Tree

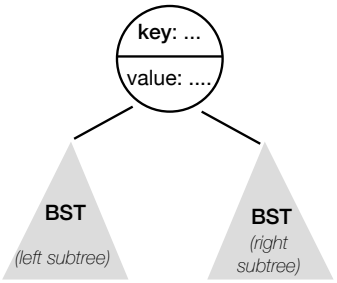
Consider the following sorted array and a tree that we superimpose on:

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

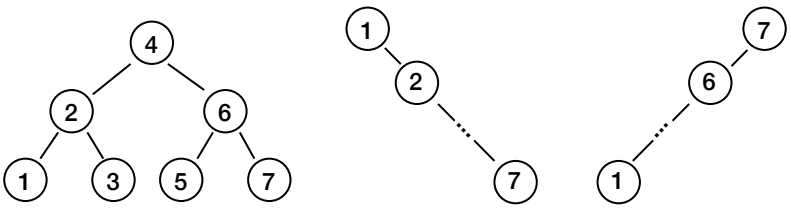
(b) a node storing a key-value pair $(key, value)$, together with two BSTs, known as the left and right subtrees, where the left subtree contains only keys that are smaller than *key* and the right subtree contains only keys that are larger than *key*.

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.



2.1 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;
}
```

2.2 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):

```
// not real Java code. need to use compareTo
public V search(K k) {
  if      (k==this.key)                return this.value;
  else if (k>this.key && right!=null) return right.search(k);
  else if (k<this.key && left!=null)  return left.search(k);
  else                           return null;
}
```

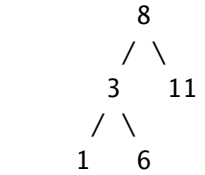
Finding the largest key. In Java, this is known as the `lastKey()` method. Let's pause for a moment and think about how one might locate the largest key in the tree. A moment's thought shows that the largest key lies at the bottom-right tip of the tree. This is because if there's anything bigger than 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 lastKey() {
  if (right!=null) return right.lastKey();
  else             return key;
}
```

Greatest key less than or equal to a given key: In Java, this is known as the `floorKey()` method. How do we implement such a method?

```
// this is not real Java code. Need to use compareTo, not ==, <, and >
public K floorKey(K k) {
    if      (k == key) return this.key;
    else if (k < key && left !=null) return left.floorKey(k);
    else if (k > key) {
       K rightFloor = (right!=null)?right.floorKey(k):null;
       return (rightFloor==null)?this.key:rightFloor;
    }
    else return null;
}
```

Before we move on, do you see how we can add a new element to the tree? How about deleting an element? To add, just do a search, it will lead you to where you need to insert. Try adding a 4.



Deleting is more complicated. If it's a leaf, you can just let that node go; otherwise, you'll have to find a replacement. The details would be beyond the scope of this class.

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.

3 Balanced Binary Search Trees

In most cases, the performance of a tree operation depends largely on the height of the tree. Hence, we generally want to the tree's height to be small—as small as possible. It is not hard to convince ourselves that a tree with n keys has height at least $\log_2 n$. More precisely, we have the following lemma:

Lemma 3.1 *An n-node tree has height at least* $\log_2(n+1)$.

Proof: First, consider that the largest tree with height h (i.e. the tree with the most number of nodes) has exactly $2^h - 1$ nodes. This is the tree where all h levels are full, so the total number of nodes is $2^0 + 2^1 + 2^2 + \ldots 2^{h-1} = 2^h - 1$, by the geometric sum formula. Therefore, if an n-node tree has height h, then $n \leqslant 2^h - 1$, which means $h \geqslant \log_2(n+1)$, proving the lemma. ■

Approximately Balanced Trees. From the intuition developed so far, if search trees are kept "balanced" in some way, then their heights will be small and they can usually be used to get good bounds. We refer to such trees as *balanced search trees*. If trees are never updated but only used for searching, then balancing is easy—it needs only

be done once. What makes balanced trees interesting is their ability to efficiently maintain balance even when updated. To allow for efficient updates, balanced search trees do not require that the trees be strictly balanced, but rather that they are approximately balanced in some way. In fact, it is impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

Dozens of balanced search trees have been suggested over the years, dating back to at least AVL trees in 1962. These trees mostly differ in how they maintain balance. Let's look briefly at a couple of them:

1. *AVL trees.* Invented in 1962 by two Russians G. M. Adelson-Velskii and E. M. Landis, these are binary search trees in which for any node, the heights of the two child subtrees can differ by at most 1. It has been proved that the height of an AVL tree is at most

$$\log_\varphi(\sqrt{5}(n+2)) - 2 \leqslant 1.44 \log_2(n+2),$$

   where $\varphi = \frac{1+\sqrt{5}}{2}$ is the Golden ratio, which is approximately $1.6180339887\cdots$.
2. *Red-Black trees.* More popular in practice this AVL trees, red-black trees are binary search trees with a somewhat looser height balance criteria. The basic idea is to label each node either red or black (requiring one extra bit of storage) and impose certain criteria about which nodes can be red/black. Overall, this leads to a tree with height at most $2\log_2(n+1)$, which is larger than the height of an AVL tree. (Indeed, in practice, AVL trees have better search time but slower update time than red-black trees.)
3. *Splay trees.* Binary search trees that are only balanced in the amortized sense—more concretely, for any sequence of m operations, the cost is never more than $m \log n$. Technically, splay trees have no height guarantees, but the design favors "hot" items, giving very good performance on applications where certain keys are more frequently accessed than others. It is one of the most common trees used in real-world applications.
4. *Treaps.* A binary search tree that uses random priorities associated with every element to keep balance. For this reason, it only has good height bounds in a probabilistic sense: with high probability, the height is at most $O(\log n)$.