Lecture 18 — Binary Search Trees (Split, Join and Treaps)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 27, 2011

Today:

- Binary Search Trees
- Split and Join
- Treaps

1 Binary Search Trees (BSTs)

Search trees are 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. Probably, the most common use is to implement sets and tables (dictionaries, mappings). What's common among search trees is that they store a collection of elements in a tree structure and use values (most often called keys) in the internal nodes to navigate in search of these elements. A *binary search tree* (BST) is a search tree in which every node in the tree has at most two children.

If search trees are kept "balanced" in some way, then they can usually be used to get good bounds on the work and span for accessing and updating them. 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—but 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. You should convince yourself that it would be impossible to maintain a perfectly balanced tree while allowing efficient (e.g. $O(\log n)$) updates.

There are dozens of balanced search trees that have been suggested over the years, dating back at least to AVL trees in 1962. The trees mostly differ in how they maintain approximate balance. Most trees either try to maintain height balance (the two children of a node are about the same height) or weight balance (the two children of a node are about the same size, i.e., the number of elements). Here we list a few balanced trees:

1. *AVL trees.* Binary search trees in which the two children of each node differ in height by at most 1.
2. *Red-Black trees.* Binary search trees with a somewhat looser height balance criteria.
3. *2–3 and 2–3–4 trees.* Trees with perfect height balance but the nodes can have different number of children so they might not be weight balanced. These are isomorphic to red-black trees by grouping each black node with its red children, if any.
4. *B-trees.* A generalization of 2–3–4 trees that allow for a large branching factor, sometimes up to 1000s of children. Due to their large branching factor, they are well-suited for storing data on disks.

5. *Splay trees.*[1] Binary search trees that are only balanced in the amortized sense (i.e. on average across multiple operations).

6. *Weight balanced trees.* Trees in which the children all have the same size. These are most typically binary, but can also have other branching factors.

7. *Treaps.* A binary search tree that uses random priorities associated with every element to keep balance.

8. *Random search trees.* A variant on treaps in which priorities are not used, but random decisions are made with probabilities based on tree sizes.

9. *Skip trees.* A randomized search tree in which nodes are promoted to higher levels based on flipping coins. These are related to skip lists, which are not technically trees but are also used as a search structure.

Traditional treatments of binary search trees concentrate on three operations: search, insert and delete. Out of these, search is naturally parallel since any number of searches can proceed in parallel with no conflicts[2] However, insert and delete are inherently sequential, as normally described. For this reason, we'll discuss more general operations that are useful for parallel updates and of which insert and delete are just a special case.

2 BST Basic Operations

In this class, we will limit ourselves to binary search trees. We assume a BST is defined by structural induction as either a leaf; or a node consisting of a left child, a right child, a key, and optional additional data.

The data is for auxiliary information such as the size of the subtree, balance information, and a value associated with the key. The keys stored at the nodes must come from a total ordered set A. For all vertices v of a BST, we require that all values in the left subtree are less than v and all values in the right subtree are greater than v. This is sometimes called the binary search tree (BST) property, or the ordering invariant.

We now consider just two functions on BSTs that are useful for building up other functions, such as search, insert and delete, but also many other useful functions such as intersection and union on sets.

$\texttt{split}(T, k) : \texttt{BST} \times \texttt{key} \to \texttt{BST} \times (\texttt{data option}) \times \texttt{BST}$
: Given a BST T and key k, `split` divides T into two BSTs, one consisting of all the keys from T less than k and the other all the keys greater than k. Furthermore if k appears in the tree with associated data d then `split` returns $\texttt{SOME}(d)$, and otherwise it returns `NONE`.

$\texttt{join}(L, m, R) : \texttt{BST} \times (\texttt{key} \times \texttt{data})\ \texttt{option} \times \texttt{BST} \to \texttt{BST}$
: This takes a left BST L, an optional middle key-data pair m, and a right BST R. It requires that all

[1] Splay trees were invented back in 1985 by Daniel Sleator and Robert Tarjan. Danny Sleator is a professor of computer science at Carnegie Mellon.

[2] In splay trees and other self-adjusting trees, this is not true since a searches can modify the tree.

keys in L are less than all keys in R. Furthermore if the optional middle element is supplied, then its key must be larger than any n L and less than any in R. It creates a new BST which is the union of L, R and the optional m.

For both split and join we assume that the BST taken and returned by the functions obey some balance criteria. For example they might be red black trees. To maintain abstraction over the particular additional data needed to maintain balance (e.g. the color for a red-black tree) we assume the following function to expose the root of a tree without the balance data:

$\texttt{expose}(T) : \texttt{BST} \to (\texttt{BST} \times \texttt{BST} \times \texttt{key} \times \texttt{data})\ \texttt{option}$
Given a BST T, if T is empty it returns `NONE`. Otherwise it returns the left child of the root, the right child of the root, and the key and data stored at the root.

With these functions we can easily implement search, insert and delete:

```
1  fun search T k =
2  let
3     val (_, v, _) = split(T, k)
4  in
5     v
6  end
```

```
1  fun insert T (k, v) =
2  let
3     val (L, v', R) = split(T, k)
4  in
5     join(L, SOME(k, v), R)
6  end
```

```
1  fun delete T k =
2  let
3     val (L, _, R) = split(T, k)
4  in
5     join(L, NONE, R)
6  end
```

Exercise 1. *Write a version of* `insert` *that takes a function* f : `data` $\times$ `data` *and if the insertion key* k *is already in the tree applies* f *to the old and new data.*

As we will show later, implementing search, insert and delete in terms of these other operations is asymptotically no more expensive than a direct implementation. However there might be some constant factor overhead so in an optimized implementation they could be implemented directly.

Now we consider a more interesting operation, taking the union of two BSTs. Note that this is different than `join` since we do not require that all the keys in one appear after the keys in the other.

```
1  fun union(T1, T2) =
2    case expose(T1) of
3      NONE ⇒ T2
4    | SOME(L1, R1, k1, v1) ⇒
5      let
6        val (L2, v2, R2) = split(T2, k1)
7      in
8        join(union(L1, L2), SOME(k1, v1), union(R1, R2))
9      end
```

This version returns the value from T_1 if a key appears in both BSTs. We will analyze the cost of this algorithm for particular balanced trees later. We note, however, that as written the code only matches our desired bounds if $|T_1| \leq |T_2|$.

The code for intersection is quite similar.

3 Implementing Split and Join on A Plain BST

We now consider a concrete implementation of *split* and *join* for a particular BST. For simplicity we consider a version with no balance criteria. For the tree we have:

```
datatype BST = Leaf | Node of (Tree * Tree * key * data)
```

```
1  fun split(T, k) =
2    case (T) of
3      Leaf  ⇒ (Leaf, NONE, Leaf)
4    | Node(L, R, k', v) ⇒
5        case compare(k, k') of
6          LESS ⇒
7            let val (L', r, R') = split(L, k)
8            in (L', r, Node(R', R, k', v)) end
9          EQUAL ⇒ (L, SOME(v), R)
10         GREATER ⇒
11           let val (L', r, R') = split(R, k)
12           in (Node(L, L', k', v), r, R') end
```

```
1  fun join(T1, m, T2) =
2    case m of
3      SOME(k, v) ⇒ Node(T1, T2, k, v)
4    | NONE ⇒
5        case T2 of
6          Leaf ⇒ T1
7        | Node(L, R, k, v) ⇒ Node(L, join(R, NONE, T2), k, v))
```

4 Treaps

A Treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. In a few lectures, we will show that with high probability, a treap with n keys will have depth $O(\log n)$. In a Treap, a random priority is assigned to every key. In practice, this is often done by performing a hash on a key, or by assigning a different random number to each key. We will assume that each priority is unique although it is possible to remove this assumption.

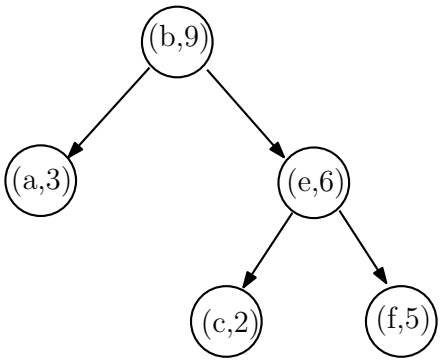
The nodes in a treap must satisfy two properties:

- **BST Property.** Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).
- **Heap Property.** The associated priority satisfy the *heap* property. The (max) heap property requires for every node that the value at a node is greater than the value of its two children.

Consider the following key-value pairs:

(a,3), (b,9), (c, 2), (e,6), (f, 5)

These elements would be placed in the following treap.



Theorem 4.1. *For any set S of key-value pairs, there is exactly one treap T containing the key-value pairs in S which satisfies the treap properties.*

Proof. The key k with the highest priority in S must be the root node, since otherwise the tree would not be in heap order. Then, to satisfy the property that the treap is ordered with respect to the nodes' keys, all keys in S less than k must be in the left subtree, and all keys greater than k must be in the right subtree. Inductively, the two subtrees of k must be constructed in the same manner. □

Parting thoughts: Notice that implementing split is identical to the plain BST version. How would you implement join?