# Lecture 14 — Search Trees I: BSTs — Split and Join

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

*Lectured by Kanat Tangwongsan — March 1, 2012*

This lecture covers *binary search trees* and how to use them to implement sets and tables.

## 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). As shown on right, a *binary tree* is a tree in which every node in the tree has at most two children. A *binary search tree* (BST) is a binary tree satisfying the following "search" property: for each node $v$, the key of the left child of $v$ is smaller than the key of $v$ and the key of the right child of $v$ is bigger than the key of $v$; that is, for the tree in the figure on right, we have $k_L < k < k_R$. This ordering is useful navigating the tree.

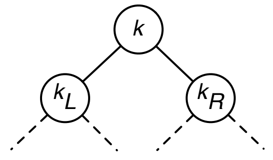


Figure 1: a binary tree

*Approximately Balanced Trees.* 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. It would be 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. The trees mostly differ in how they maintain 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 in the subtrees). Let us 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.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

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.

Traditionally, 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.

## 1.1 BST Basic Operations

We'll mostly focus on binary search trees in this class. 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. That is, we have

```
datatype BST = Leaf | Node of (BST * BST * key * 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'll rely on the following two basic building blocks to build up other functions, such as `search`, `insert`, and `delete`, but also many other useful functions such as intersection and union on sets.

$\texttt{split}(T,k) : \text{BST} \times \texttt{key} \to \text{BST} \times (\texttt{data option}) \times \text{BST}$

Given a BST $T$ and key $k$, `split` divides $T$ into two BSTs, one consisting of all the keys from $T$

[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.

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 `SOME`$(d)$, and otherwise it returns `NONE`.

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

`expose`$(T)$ : `BST` $\rightarrow$ (`BST` $\times$ `BST` $\times$ `key` $\times$ `data`) `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 implement `search`, `insert`, and `delete`. As our first example, here is how we implement `search`:

```
fun search T k =
let val (_, v, _) = split(T, k)
in v
end
```

```
fun insert T (k, v) =
let val (L, v', R) = split(T, k)
in join(L, SOME(k, v), R)
end
```

```
fun delete T k =
let val (L, _, R) = split(T, k)
in join(L, NONE, R)
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.

## 2 How to implement split and join on a simple 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 declare the following data type:

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

```
fun split($T, k$) =
  case $T$ of
    Leaf ⇒ (Leaf, NONE, Leaf)
  | Node($L, R, k', v$) ⇒
      case compare($k, k'$) of
        LESS ⇒
          let val ($L', r, R'$) = split($L, k$)
          in ($L', r$, Node($R', R, k', v$)) end
        EQUAL ⇒ ($L$, SOME($v$), $R$)
        GREATER ⇒
          let val ($L', r, R'$) = split($R, k$)
          in (Node($L, L', k', v$), $r, R'$) end
```

```
fun join($T_1, m, T_2$) =
  case $m$ of
    SOME($k, v$) ⇒ Node($T_1, T_2, k, v$)
  | NONE ⇒
      case $T_1$ of
        Leaf ⇒ $T_2$
      | Node($L, R, k, v$) ⇒ Node($L$, join($R$, NONE, $T_2$), $k, v$))
```

## 3 Union

Let's now consider a more interesting operation: taking the union of two BSTs. Note that this differs from `join` since we do not require that all the keys in one appear after the keys in the other. The code below implements the union function:

```
fun union($T_1, T_2$) =
  case expose($T_1$) of
    NONE ⇒ $T_2$
  | SOME($L_1, R_1, k_1, v_1$) ⇒
    let val ($L_2, v_2, R_2$) = split($T_2, k_1$)
    in join(union($L_1, L_2$), SOME($k_1, v_1$), union($R_1, R_2$))
    end
```

For simplicity, this version returns the value from $T_1$ if a key appears in both BSTs. We'll analyze the cost of `union` next. The code for set intersection and set difference is quite similar.

### 3.1 Cost of Union

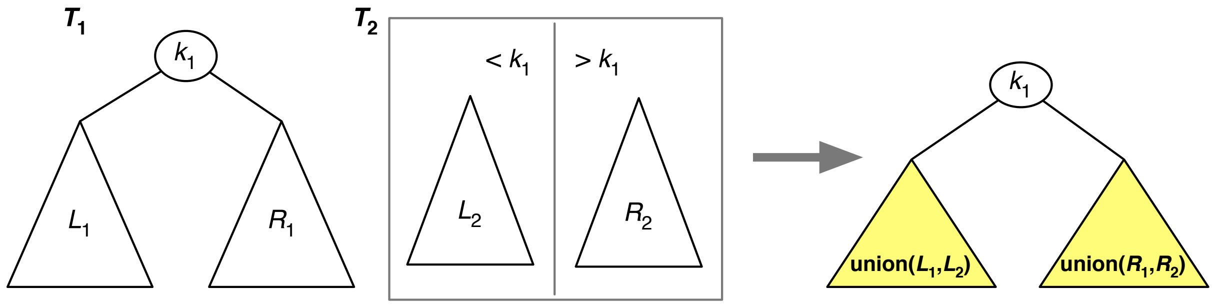
In the 15-210 library, `union` and similar functions (e.g., `intersection` and `difference` on sets and `merge`, `extract` and `erase` on tables) have $O(m\log(1+\frac{n}{m}))$ work, where $m$ is the size of the smaller input and $n$ the size of the larger one. At first glance, this may seem like a strange bound, but we will see how it falls out very naturally from the `union` code.

To analyze this, we'll first assume that the work and span of `split` and `join` is proportional to the depth of the input tree(s). In a reasonable implementation, these operations traverse a path in the tree (or trees in the case of `join`). Therefore, if the trees are reasonably balanced and have depth $O(\log n)$, then the work and span of both `split` and `join` on a tree of $n$ nodes is $O(\log n)$. Indeed, most balanced trees have $O(\log n)$ depth.

The `union` algorithm we just wrote has the following basic structure. On input $T_1$ and $T_2$, the function `union`$(T_1, T_2)$ performs:

1. For $T_1$ with key $k_1$ and children $L_1$ and $R_1$ at the root, use $k_1$ to split $T_2$ into $L_2$ and $R_2$.
2. Recursively find $L_u =$ `union`$(L_1, L_2)$ and $R_u =$ `union`$(R_1, R_2)$.
3. Now `join`$(L_u, k_1, R_u)$.

Pictorially, the process looks like this:



We'll begin the analysis by examining the cost of each `union` call. Notice that each call to `union` makes one call to `split` and one to `join`, each costing $O(\log |T_2|)$. To ease the analysis, we will make the following assumptions:

1. $T_1$ it is perfectly balanced (i.e., `expose` returns subtrees of size $n/2$), and
2. each time a key from $T_1$ splits $T_2$, it splits the tree exactly in half.

With these assumptions, we can write a recurrence for the work of `union` as follows:

$$W(|T_1|, |T_2|) = 2W(|T_1|/2, |T_2|/2) + O(\log |T_2|),$$

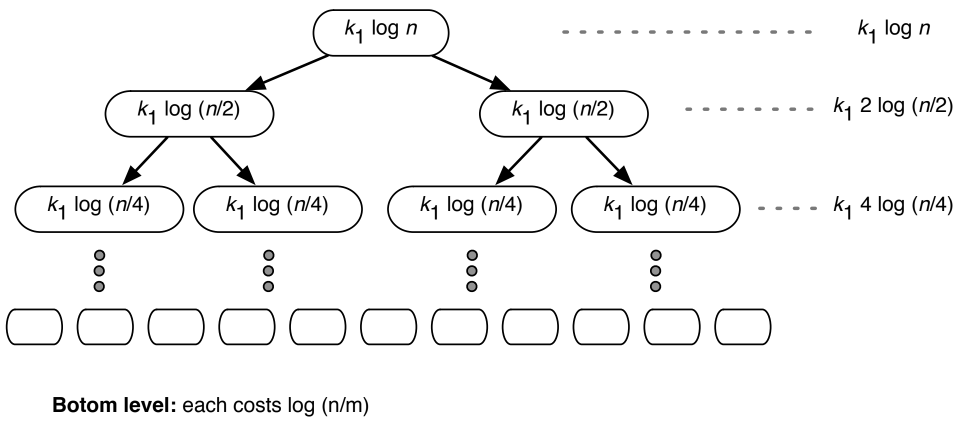
and

$$W(1, |T_2|) = O(\log |T_2|).$$

This recurrence deserves more explanation: When $|T_1| > 1$, `expose` gives us a perfect split, resulting in a key $k_1$ and two subtrees of size $|T_1|/2$ each—and by our assumption (which we'll soon eliminate), $k_1$ splits $T_2$ perfectly in half, so the subtrees `split` that produces have size $|T_2|/2$.

When $|T_1| = 1$, we know that `expose` give us two empty subtrees $L_1$ and $R_1$, which means that both `union`$(L_1, L_2)$ and `union`$(R_1, R_2)$ will return immediately with values $L_2$ and $R_2$, respectively. Joining these together costs at most $O(\log |T_2|)$. Therefore, when $|T_1| = 1$, the cost of `union` (which involves one `split` and one `join`) is $O(\log |T_2|)$.

Let $n$ denote the size of $T_2$ initially. If we draw the recursion tree, we obtain the following:



There are several features of this tree that's worth mentioning: First, ignoring the somewhat-peculiar cost in the base case, we know that this tree is leaf-dominated. Therefore, excluding the cost at the bottom level, the cost of `union` is $O(\#$ of leaves$)$.

*But how many leaves are there? And how deep is this tree?* To find the number of leaves, we'll take a closer look at the work recurrence. Notice that in the recurrence, the tree bottoms out when $|T_1| = 1$ and before that, $T_1$ always gets split in half (remember that $T_1$ is perfectly balanced). Nowhere in there does $T_2$ affects the shape of the recursion tree or the stopping condition. Therefore, this is yet another recurrence of the form $f(m) = f(m/2) + O(...)$, which means that *it has $m$ leaves and is $1 + \log_2 m$ deep.*

Next, we'll determine the size of $T_2$ at the leaves. Remember that as we descend down the recursion tree, the size of $T_2$ gets halved, so the size of $T_2$ at a node at level $i$ (counting from 0) is $n/2^i$. But we knew already that the depth of the recursion tree is $\log_2 m$, so the size of $T_2$ at each of the leaves is

$$n/2^{\log_2 m} = \frac{n}{m}.$$

Therefore, each leaf node costs $O(\log(\frac{n}{m}))$. Since there are $m$ leaves, the whole bottom level costs $O(m \log(\frac{n}{m}))$. Hence, if the trees satisfy our assumptions, we have that `union` runs in $O(m + m\log(\frac{n}{m})) = O(m\log(1 + \frac{n}{m}))$ work. We write $\log(1 + \frac{n}{m})$ instead of $\log(\frac{n}{m})$ to avoid dealing with the case when $n = m$ and we get 0 after applying a log.

**Removing An Assumption:** Of course, in reality, our keys in $T_1$ won't split subtrees of $T_2$ in half every time. But it turns out this only helps. We won't go through a rigorous argument, but if we keep the assumption that $T_1$ is perfectly balanced, then the shape of the recursion tree stays the same. What is now different is the cost at each level. Let's try to analyze the cost at level $i$. At this level, there are $k := 2^i$ nodes. Say the sizes of $T_2$ at these nodes are $n_1, \ldots, n_k$, where $\sum_j n_j = n$. Then, the total cost for this level is

$$c \cdot \sum_{j=1}^{k} \log(n_j) \;\leq\; c \cdot \sum_{j=1}^{k} \log(n/k) = c \cdot 2^i \cdot \log(n/2^i),$$

where we used the fact that the logarithm function is concave[3]. Thus, the tree remains leaf-dominated and the same reasoning shows that the total work is $O(m \log(1 + \frac{n}{m}))$.

Still, in reality, $T_1$ doesn't have to be perfectly balanced as we assumed. A similar reasoning can be used to show that $T_1$ only has to be approximately balanced. We will leave this case as an exercise. We'll end by remarking that as described, the span of `union` is $O(\log^2 n)$, but this can be improved to $O(\log n)$ by changing the the algorithm slightly.

In summary, `union` can be implemented in $O(m \log(1 + \frac{n}{m}))$ work and span $O(\log n)$. The same holds for the other similar operations (e.g. `intersection`).

[3]Technically, we're applying the so-called Jensen's inequality.