Lecture 19-20 — Binary Search Trees

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

Lectured by Danny Sleator — 31 Oct (Halloween!) and Nov 5 2013

Today:

- Binary Search Trees (BST)
- Split and Join operations
- Treap: A randomized BST, and its analysis
- Union: The algorithm and analysis

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, all the keys in the left subtree of v are smaller than the key of v, which is in turn smaller than all the keys in the right subtree of v. For example, in the figure on the right, we have $k_L < k < k_R$. This ordering is useful navigating the tree.

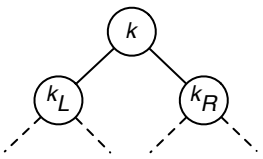


Figure 1: a binary tree

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 children of a node are about the same height) or weight balance (the children of a node are about the same size, i.e., the number of elements in the subtrees). 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.

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

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 (every leaf is at the same depth) 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 with slow access times.

5. *Weight balanced trees.* Trees in which each node's children have sizes all within a constant factor. These are most typically binary, but can also have other branching factors.

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

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

8. *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.

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

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]. Insert and delete, however, are inherently sequential, as normally described. For this reason, we'll discuss more general operations that are useful for implementing parallel updates, 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.

[1] Splay trees were invented 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.

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.

`split`(T,k) : BST $\times$ `key` $\rightarrow$ BST $\times$ (`data option`) $\times$ 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 `SOME`(d), and otherwise it returns `NONE`.

`join`(L,m,R) : BST $\times$ (`key` $\times$ `data`) `option` $\times$ BST $\rightarrow$ BST
: This function 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 in 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` quite simply:

```
1  function search T k =
2  let (_, v, _) = split(T, k)
3  in v
4  end
```

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

```
1  function delete T k =
2  let (L, _, R) = split(T, k)
3  in join(L, NONE, R)
4  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 `split` and `join` is asymptotically no more expensive than a direct implementation. There might be some constant

factor overhead, however, so in an optimized implementation search, insert, and delete might be implemented directly.

More interestingly, we can use `split` and `join` to implement union, intersection, or difference of two BSTs. Note that union differs from `join` since it does not require that all the keys in one appear after the keys in the other; the keys may overlap.

Exercise 2. *Implement* `union`*,* `intersection`*, and* `difference` *using* `split` *and* `join`*.*

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)
```

```
1  function 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 (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 (L′, r, R′) = split(R, k)
12           in (Node(L, L′, k′, v), r, R′) end
```

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

We claim that the same approach can be easily use to implemented `split` and `join` on just about any balanced search tree.

3 Quicksort and BSTs

Can we think of binary search trees in terms of an algorithm we already know? As is turns out, the quicksort algorithm and binary search trees are closely related: if we write out the recursion tree for quicksort and annotate each node with the pivot it picks, what we get is a BST.

Let's try to convince ourselves that the function-call tree for quicksort generates a binary search tree when the keys are distinct. To do this, we'll modify the quicksort code from a earlier lecture to produce the tree as we just described. In this implementation, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

```
1  function qs_tree(S) =
2    if |S| = 0 then LEAF
3    else let
4           p = pick a pivot from S
5           S1 = ⟨ s ∈ S | s < p ⟩
6           S2 = ⟨ s ∈ S | s > p ⟩
7           (TL, TR) = ( qs_tree(S1) || qs_tree(S2) )
8       in
9          NODE(TL, p, TR)
10      end
```

Notice that this is clearly a binary tree. To show that this is a binary *search* tree, we only have to consider the ordering invariant. But this, too, is easy to see: for `qs_tree` call, we compute S_1, whose elements are strictly smaller than p—and S_2, whose elements are strictly bigger than p. So, the tree we construct has the ordering invariant. In fact, this is an algorithm that converts a sequence into a binary search tree.

It clear that, whatever the pivot strategy is, the maximum depth of the binary search tree resulting from `qs_tree` is the same as the maximum depth of the recursion tree for quicksort using that strategy. As shown in lecture, the expected depth of the recursion tree for *randomized* quicksort is $O(\log n)$

Can we maintain a tree data structure that centers on this random pivot-selection idea? If so, we automatically get a nice BST.

4 Treaps

Unlike quicksort, when building a BST we don't necessarily know all the elements that will be in the BST at the start, so we can't randomly pick an element (in the future) to be the root of the BST. So how can we use randomization to help maintain balance in a BST?

A treap (tree + heap) is a randomized BST that maintains balance in a probabilistic way. 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:

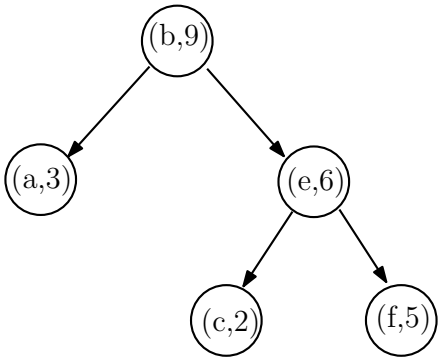
BST Property: Their keys satisfy the BST property (i.e., keys are stored in-order in the tree).

Heap Property: The associated priorities 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-priority 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 unique key-priority pairs, there is exactly one treap T containing the key-priority pairs in S which satisfies the treap properties.*

Proof. The empty tree is clearly unique (base case). The key k with the highest priority in S must be the root node, since otherwise the tree would not be in heap order. Only one key has the highest priority. 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. □

Note, there is a subtle distinction here with respect to randomization. With quicksort the algorithm is randomized. With treaps, none of the functions for treaps are randomized. It is the data structure itself that is randomized[3].

Split and Join on Treaps

As mentioned earlier, for any binary tree all we need to implement is split and join and these can be used to implement the other BST operations. Recall that split takes a BST and a key and splits the BST into two BST and an optional value. One BST only has keys that are less than the given key, the other BST only has keys that are greater than the given key, and the optional value is the value of the given key, if it is the tree. Join takes two BSTs and an optional middle (key,value) pair, where the maximum key on the first tree is less than the minimum key on the second tree. It returns a BST that contains all the keys the given BSTs and middle key.

We claim that the split code given above for unbalanced trees does not need to be modified for treaps.

Exercise 3. *Convince yourselves than when doing a split none of the priority orders change (i.e. the code will never put a larger priority below a smaller priority).*

[3] In contrast, for the so called Randomize Binary Search Trees, it is the functions that update the tree that are randomized.

The join code, however, does need to be changed. The new version has to check the priorities of the two roots, and use whichever is greater as the new root. In the algorithm shown below, we assume that the priority of a key can be computed from the key (e.g., priorities are a hash of the key).

```
1  function join(T1, m, T2) =
2  let
3    function singleton(k, v) = Node(Leaf, Leaf, k, v)
4    function join'(T1, T2) =
5      case (T1, T2) of
6        (Leaf, _) ⇒ T2
7      | (_, Leaf) ⇒ T1
8      | (Node(L1, R1, k1, v1), Node(L2, R2, k2, v2)) ⇒
9          if (priority(k1) > priority(k2)) then
10           Node(L1, join'(R1, T2), k1, v1)
11         else
12           Node(join'(T1, L2), R2, k2, v2)
13 in
14   case m of
15     NONE ⇒ join'(T1, T2))
16   | SOME(k, v) ⇒ join'(T1, join'(singleton(k, v), T2))
17 end
```

In the code join′ is a version of join that has no middle element as an argument. Note that line 9 compares the priorities of the two roots and then places the key with the larger priority in the new root causing a recursive call to join on one of the two sides.

We refer to the *left spine* of the tree as the path from the root to the leftmost node in the tree, and the *right spine* as the path from the root to the rightmost node in the tree. What join′(T_1, T_2) does is to interleave pieces of the right spine of T_1 with pieces the left spine of T_2, in a way that ensures that the priorities are in decreasing order down the path.

Because the keys and priorities determine a treap uniquely, spliting a tree and joining it back together results in the same treap. This property is not true of most other kinds of balanced trees; the order that operations are applied can change the shape of the tree.

Because the cost of split and join depends on the depth of the i^{th} element in a treap, we now analyze the expected depth of a key in the tree.

5 Expected Depth of a Key in a Treap

Consider a set of keys K and associated priorities p : key → int. For this analysis, we assume the priorities are unique and random. Consider the keys laid out in order, and as with the analysis of quicksort, we use i and j to refer to the i^{th} and j^{th} keys in this ordering. Unlike quicksort analysis, though, when analyzing the depth of a node i, i and j can be in any order, since an ancestor of i in a BST can be either less than or greater than i.

```
| | | | | | | | | | | | | | | | | | | |
          i                 j
```

If we calculate the depth starting with zero at the root, the expected depth of a key is equivalent to the number of ancestors it has in the tree. So we want to know how many ancestors a particular node i has. We use the indicator random variable A_i^j to indicate that j is an ancestor of i. (Note that the superscript here does not mean A_i is raised to the power j; it simply is a reminder that j is the ancestor of i.) By the linearity of expectatons, the expected depth of i can be written as:

$$\mathbf{E}\left[\text{depth of } i \text{ in } T\right] = \mathbf{E}\left[\sum_{j=1}^{n} A_i^j\right] = \sum_{j=1}^{n} \mathbf{E}\left[A_i^j\right].$$

To analyze A_i^j let's just consider the $|j-i|+1$ keys and associated priorities from i to j inclusive of both ends. As with the analysis of quicksort, if an element k has the highest priority and k is less than both i and j or greater than both i and j, it plays no role in whether j is an ancestor of i or not. The following three cases do:

1. The element i has the highest priority.
2. One of the elements k in the middle has the highest priority (i.e., neither i nor j).
3. The element j has the highest priority.

What happens in each case?

1. If i has the highest priority then j cannot be an ancestor of i, and $A_i^j = 0$.
2. If k between i and j has the highest priority, then $A_i^j = 0$, also. Suppose it was not. Then, as j is an ancestor of i, it must also be an ancestor of k. That is, since in a BST every branch covers a contiguous region, if i is in the left (or right) branch of j, then k must also be. But since the priority of k is larger than that of j this cannot be the case, so j is not an ancestor of i.
3. If j has the highest priority, j must be an ancestor of i and $A_i^j = 1$. Otherwise, to separate i from j would require a key in between with a higher priority. We therefore have that j is an ancestor of i exactly when it has a priority greater than all elements from i to j (inclusive on both sides).

Therefore j is an ancestor of i if and only if it has the highest priority of the keys between i and j, inclusive. Because priorities are selected randomly, there a chance of $1/(|j-i|+1)$ that $A_i^j = 1$ and we have $\mathbf{E}\left[A_i^j\right] = \frac{1}{|j-i|+1}$. (Note that if we include the probability of either j being an ancestor of i or i being an ancestor of j then the analysis is identical to quicksort. Think about why.)

Now we have

$$\begin{aligned}
\mathbf{E}\left[\text{depth of } i \text{ in } T\right] &= \sum_{j=1, j\neq i}^{n} \frac{1}{|j-i|+1} \\
&= \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= \sum_{k=2}^{i} \frac{1}{k} + \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&= H_i - 1 + H_{n-i+1} - 1 \\
&< \ln i + \ln(n-i+1) \\
&= O(\log n)
\end{aligned}$$

Recall that the harmonic number is $H_n = \sum_{i=1}^{n} \frac{1}{n}$. It has the following bounds: $\ln n < H_n < \ln n + 1$, where $\ln n = \log_e n$. Notice that the expected depth of a key in the treap is determined solely by it relative position in the sorted keys.

Exercise 4. *Including constant factors how does the expected depth for the first key compare to the expected depth of the middle ($i = n/2$) key?*

Theorem 5.1. *For treaps the cost of* `join`(T_1, m, T_2) *returning* T *and of* `split`$(T, (k, v))$ *is* $O(\log |T|)$ *expected work and span.*

Proof. The `split` operation only traverses the path from the root down to the node at which the key lies or to a leaf if it is not in the tree. The work and span are proportional to this path length. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$.

For `join`(T_1, m, T_2) the code traverses only the right spine of T_1 or the left spine of T_2. Therefore the work is at most proportional to the sum of the depth of the rightmost key in T_1 and the depth of the leftmost key in T_2. The work of `join` is therefore the sum of the expected depth of these nodes. Since the resulting treap T is an interleaving of these spines, the expected depth is bound by $O(\log |T|)$. □

5.1 Expected overall depth of treaps

Even though the expected depth of a node in a treap is $O(\log n)$, it does not tell us what the expected maximum depth of a treap is. As you have saw in lecture 15, $\mathbf{E}\left[\max_i\{A_i\}\right] \neq \max_i\{\mathbf{E}\left[A_i\right]\}$. As you might surmise, the analysis for the expected depth is identical to the analysis of the expected span of randomized quicksort, except the recurrence uses 1 instead of $c \log n$. That is, the depth of the recursion tree for randomized quicksort is $D(n) = D(Y_n) + 1$, where Y_n is the size of the larger partition. Thus, the expected depth is $O(\log n)$.

It turns out that is possible to say something stronger: For a treap with n keys, the probability that any key is deeper than $10 \ln n$ is at most $1/n^4$. That is, for large n a treap with random priorities

[4] The bound base on Chernoff bounds which relies on events being independent.

has depth $O(\log n)$ with *high probability*. It also implies that randomized quicksort $O(n \log n)$ work and $O(\log^2 n)$ span bounds hold with high probability.

Being able to put high probability bounds on the runtime of an algorithm can be critical in some situations. For example, suppose my company DontCrash is selling you a new air traffic control system and I say that in expectation, no two planes will get closer than 500 meters of each other—would you be satisfied? More relevant to this class, let's say you wanted to run 1000 jobs on 1000 processors and I told you that in expectation each finishes in an hour—would you be happy? How long might you have to wait?

There are two problems with expectations, at least on their own. Firstly, they tell us very little if anything about the variance. And secondly, as mentioned in an earlier lecture, the expectation of a maximum can be much higher than the maximum of expectations. The first has implications in real time systems where we need to get things done in time, and the second in getting efficient parallel algorithms (e.g., span is the max span of the two parallel calls). Proving these high probability bounds is beyond the scope of this course.

6 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:

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

For simplicity, this version returns the value from T_1 if a key appears in both BSTs. Notice that `union` uses `split` and `join`, so it can be used for any BST with this with these two operations.

We'll analyze the cost of `union` next. The code for set intersection and set difference is quite similar.

6.1 Cost of Union

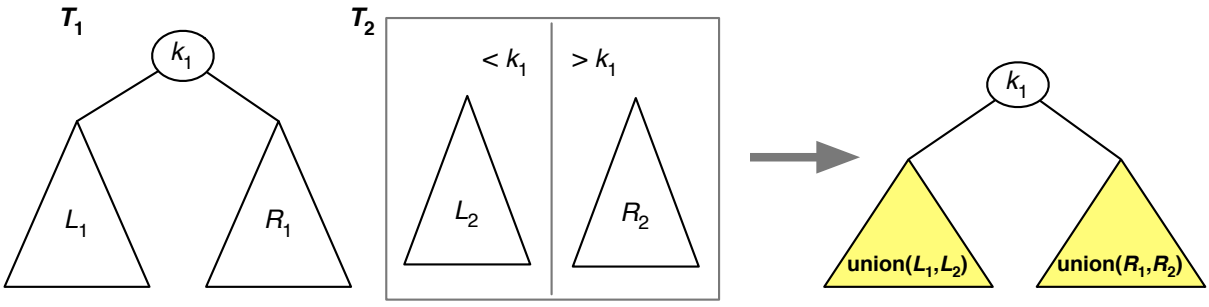
In the 15-210 library, `union` and similar functions (e.g., `intersection` and `difference` on sets and `merge`, `extract` and `erase` on tables) have expected $O(m \log(1 + \frac{n}{m}))$ work, where m is the size of the smaller input and n the size of the larger one. This bound is the same as the lower bound for merging two sorted sequences. We will see how this bound falls out very naturally from the `union` code.

To analyze `union`, we'll first assume that the work and span of `split` and `join` is proportional to the depth of the input tree and the output tree, respectively. 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 `split` on a tree of n nodes and `join` resulting in a tree of n nodes is $O(\log n)$. Indeed, most balanced trees have $O(\log n)$ depth. This is true both for red-black trees and treaps.

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` costing $O(\log|T_2|)$ and one to `join`, each costing $O(\log(|T_1|+|T_2|))$. To ease the analysis, we will make the following assumptions:

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

Later we will relax these assumptions.

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

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

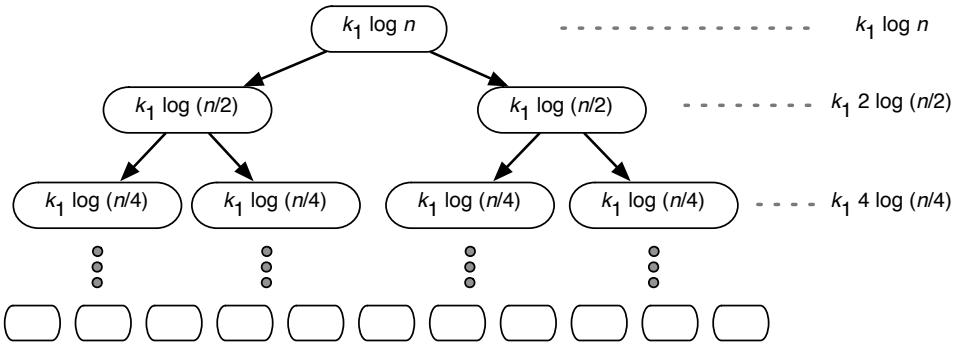
and

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

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 with T_1 costs at most $O(\log(|T_1| + |T_2|))$. Therefore, when $|T_1| = 1$, the cost of `union` (which involves one `split` and one `join`) is $O(\log(1 + |T_2|))$.

Let $m = |T_1|$ and $n = |T_2|, m < n$ and $N = n + m$. If we draw the recursion tree that shows the work associated with splitting T_2 and joining the results, we obtain the following:



Botom level: each costs log (n/m)

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) times the cost of each leaf.

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 know already that leaves are at level $\log_2 m$, 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 $O(\log(1 + \frac{n}{m}))$. Since there are m leaves, the whole bottom level costs $O(m \log(1 + \frac{n}{m}))$. Hence, if the trees satisfy our assumptions, we have that `union` runs in $O(m \log(1 + \frac{n}{m}))$ work.

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 in the recursion tree. 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[5]. 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`).

Summary

Earlier we showed that randomized quicksort has worst-case expected $O(n \log n)$ work, and this expectation was independent of the input. That is, there is no bad input that would cause the work to be worse than $O(n \log n)$ all the time. It is possible, however, (with extremely low probability) we could be unlucky, and the random chosen pivots could result in quicksort taking $O(n^2)$ work.

It turns out the same analysis shows that a deterministic quicksort will on average have $O(n \log n)$ work. Just shuffle the input randomly, and run the algorithm. It behaves the same way as randomized quicksort on that shuffled input. Unfortunately, on some inputs (e.g., almost sorted) the deterministic quicksort is slow, $O(n^2)$, every time on that input.

Treaps take advantage of the same randomization idea. But a binary search tree is a dynamic data structure, and it cannot change the order in which operations are applied to it. So instead of randomizing the input order, it adds randomization to the data structure itself.

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