Lecture 19 — Union, Quick Sort, and Treaps

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

Lectured by Guy Blelloch — Nov 1, 2011

Today:

- Quick Sort Revisited
- Split and Join
- Treaps

1 Quick Review: Binary Search Trees

For a quick recap, recall that in the last lecture, we were talking about binary search trees. In particular, we looked at the following:

- *Many ways to keep a search tree almost balanced.* Such trees include red-black trees, 2-3 trees, B-trees, AVL trees, Splay trees, Treaps, weight balanced trees, skip trees, among others.

  Some of these are binary, some are not. In general, a node with k children will hold $k-1$ keys. But in this course, we will restrict ourselves to binary search trees.

- *Using* `split` *and* `join` *to implement other operations.* The `split` and `join` operations can be used to implement most other operations on binary search trees, including: `search`, `insert`, `delete`, `union`, `intersection` and `difference`.

- *An implementation of* `split` *and* `join` *on unbalanced trees.* We claim that the same idea can also be easily implemented on just about any of the balanced trees.

We then started describing a particular balanced tree, called Treaps (Trees + Heaps). The idea of a Treap is that we assign to each key a randomized priority and then maintain a tree such that

1. the keys satisfy the binary search tree (BST) property, and
2. the priorities satisfy the heap property (i.e. the priority of every node is greater than the priority of its children).

Today: before getting back to Treaps, we will discuss two other related topics. The first is the analysis of the cost of `union` based on the code we gave last time. Notice that the cost for `intersection`and set difference is the same. The second is an analysis of the quick sort algorithm. It may seem odd to analyze quick sort in the middle of a lecture on balanced search trees, but as we will see, the analysis of quick sort is the same as the analysis of Treaps. We basically will reduce the analysis of Treaps to the analysis of quick sort.

2 Cost of Union

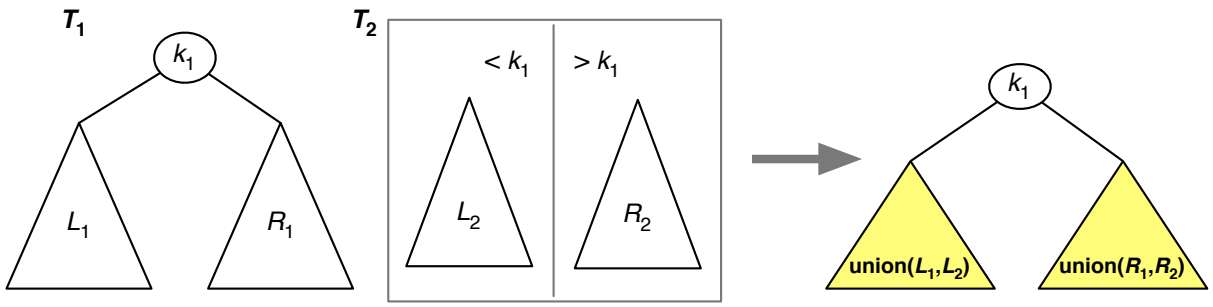
In the 15-210 library, `union` and similar functions (e.g., `intersection` and `difference` on sets and `merge`, `extract` and `erase` on tables) have $O(m \log(n/m))$ work, where m is the length of the shorter input and n the length of the longer one. At first glance, this may seem like a strange bound,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

To analyze this, first, we note that the work and span of `split` and `join` is proportional to the depth of the input tree(s). This is simply because the code just traverses a path in the tree (or trees for join). Therefore, if the trees are reasonably balanced and have depth $O(\log n)$, then the work and span of both `split` and `join` is $O(\log n)$. Indeed, most balanced trees have $O(\log n)$ depth. You have already argued this for red-black trees, and we will soon argue it for Treaps.

Let's recall the basic structure of $\texttt{union}(T_1, T_2)$.

- 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.
- Recursively find $L_u = \texttt{union}(L_1, L_2)$ and $R_u = \texttt{union}(R_1, R_2)$.
- Now $\texttt{join}(L_u, k_1, R_u)$.

Pictorially, the process looks like this:

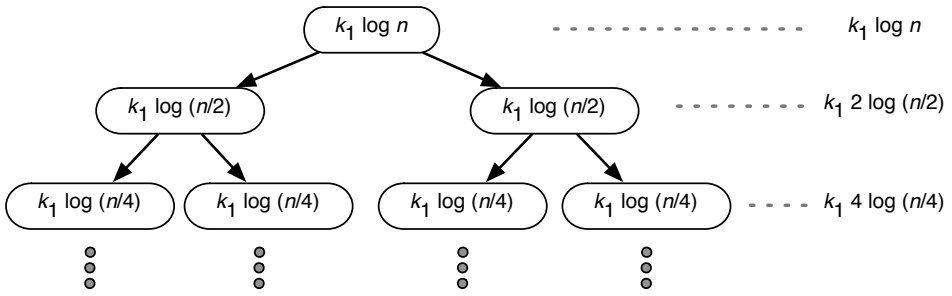


Note that each call to `union` makes one call to `split` and one to `join` each which take $O(\log n)$ work, where n is the size of T_2. We assume that T_1 is the smaller tree with size m. For staters, we'll make the following assumptions to ease the analysis:

1. let's assume that T_1 it is perfectly balanced, and
2. each time a key from T_1 splits T_2, it splits it exactly in half.

Now if we draw the recursion tree[1], we obtain the following:

[1] If we want to write out a work recurrence, it will be $W(m, n) = W(m/2, n/2) + \Theta(\log n)$



Is this tree root or leaf dominated, or evenly sized? And how many levels will it have? It is not hard to see that this tree is dominated at the leaves. In fact, what's happening is that when we get to the bottom level, each leaf in T_1 has to split a subtree of T_2 of size n/m. This takes $O(\log(\frac{n}{m}))$ work. Since there are $O(m)$ such splits, the total work at the leaves is $O(m \log(\frac{n}{m}))$. Furthermore, the work going up the tree decreases geometrically.

To bound this more formally, since T_1 has m keys and it is split exactly in half, it will have $\log_2 m$ levels, so if we start counting from level 0 at the root, we have that the i-th level has 2^i nodes, each costing $\Theta(\log(n/2^i))$. Therefore, for example, the bottom level will have $\frac{m}{2}$ nodes—and the whole bottom level will cost

$$k_1 \cdot \frac{m}{2} \log\left(n/2^{\log m - 1}\right) = k_1 \cdot \frac{m}{2} \left(\log n - \log(m/2)\right).$$

Since the tree is dominated at the leaves, the total work will be $O(m \log(\frac{n}{m}))$, as desired. Hence, if the trees satisfy our assumptions, we have that `union` runs in $O(m \log(\frac{n}{m}))$.

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 it is not difficult to show that the recursion tree remains leaf dominated. So, once again, it suffices to consider the bottom level of recursion. Since T_1 is balanced, there will be $k := m/2$ subtrees of T_2 that need to be split. Suppose these subtrees have sizes $n_1, n_2, \ldots, n_k$, where $\sum_{i=1}^{k} n_i = n$ since the subtrees are a partition of the original tree T_2. Therefore, the total cost of splitting these subtrees is

$$\sum_{i=1}^{k} k_1 \log(n_i) \quad \leq \quad \sum_{i=1}^{k} k_1 \log(n/k),$$

where we use the fact that the logarithm function is concave[2] This shows that the total work is $O(m \log(\frac{n}{m}))$.

Still, in actuality, T_1 doesn't have to be perfectly balanced as we assumed. Although this, too, is not too hard to show, 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, this means that `union` can be implemented in $O(m \log(n/m))$ work and span $O(\log n)$. The same holds for the other similar operations (e.g. `intersection`).

3 Quick Sort

We now turn to analyzing quick sort. As mentioned earlier, the motivation for analyzing this now is that this analysis is a nice segue into the Treap analysis. The argument here is essentially the same as the

[2]Technically, we're applying a varaint of the so-called Jensen's inequality.

analysis we needed to show that the expected depth of a node in a Treap is $O(\log n)$. Of course, being able to analyze randomized quick sort is very important on its own, and the analysis gives an elegant example of randomized analysis.

The randomized quick sort algorithm is as follows:

```
1  fun quickSort(S) =
2    if |S| ≤ 1 then S
3    else let
4           val p = select a random key from S
5           val S1 = ⟨ s ∈ S | s < p ⟩
6           val S2 = ⟨ s ∈ S | s = p ⟩
7           val S3 = ⟨ s ∈ S | s > p ⟩
8         in
9           quickSort(S1) @ S2 @ quickSort(S3)
10        end
```

For the analysis, we'll consider a completely equivalent algorithm which will be slightly easier to analyze. Before the start of the algorithm, we'll pick for each element a random priority uniformly at random from the real interval $[0, 1]$—and in Line 4 in the above algorithm, we'll instead pick the key with the highest priority (sound familiar?). Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of the algorithm are fully equivalent (modulo the technical details about how we might store the priority values).

We're interested in counting how many comparisons `quickSort` makes. This immediately bounds the work for the algorithm because this is where the bulk of work is done. That is, if we let

$$X_n \quad = \quad \#\text{ of comparisions } \texttt{quickSort} \text{ makes on input of size } n,$$

our goal is to find an upper bound on $\mathbf{E}[X_n]$ for any input sequence S. For this, we'll consider the final sorted order[3] of the keys $T = \texttt{sort}(S)$. In this terminology, we'll also denote by p_i the priority we chose for the element T_i.

We'll derive an expression for X_n by breaking it up into a bunch of random variables and bound them. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence T. We use the random indicator variables A_{ij} to indicate whether we compare the elements T_i and T_j during the algorithm—i.e., the variable will take on the value 1 if they are compared and 0 otherwise.

Looking closely at the algorithm, we have that if some two elements are compared, one of them has to be a pivot in that call. So, then, the other element will be part of S_1, S_2, or S_3—but the pivot element will be part of S_2, which we don't recurse on. This gives the following observation:

Observation 3.1. *In the quick sort algorithm, if some two elements are compared in a* `quickSort` *call, they will never be compared again in other call.*

Therefore, with these random variables, we can express the total comparsion count X_n as follows:

$$X_n \quad \leq \quad 3 \sum_{i=1}^{n} \sum_{j=i+1}^{n} A_{i,j}$$

[3]Formally, there's a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ between the positions of S and T.

This is because our not-so-optimized quick sort compares each element to a pivot 3 times. By linearity of expectation, we have $\mathbf{E}[X_n] \leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}[A_{i,j}]$. Furthermore, since each $A_{i,j}$ is an indicator random variable, $\mathbf{E}[A_{i,j}] = \mathbf{Pr}[A_{i,j} = 1]$. Our task therefore comes down to computing the probability that T_i and T_j are compared (i.e., $\mathbf{Pr}[A_{i,j} = 1]$) and working out the sum.

Computing the probability Pr $[A_{i,j} = 1]$. The crux of the matter is in descibing the event $A_{i,j} = 1$ in terms of a simple event that we have a handle on. Before we prove any concrete result, let's take a closer look at the quick sort algorithm to gather some intutions. Notice that the top level takes as its pivot p the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For each of these parts, we run `quickSort` recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further. With this view, the following observation is not hard to see:

Claim 3.2. *For $i < j$, T_i and T_j are compared if and only if p_i or p_j has the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.*

Proof. We'll show this by contradition. Asssume there is a key T_k, $i < k < j$ with a higher priority between them. In any collection of keys that include T_i and T_j, T_k will become a pivot before either of them. Since T_k "sits" between T_i and T_k (i.e., $T_i \leq T_k \leq T_j$) , it will separate T_i and T_j into different buckets, so they are never compared. □

Therefore, for T_i and T_j to be compared, p_i or p_j has to be bigger than all the priorities inbetween. Since there are $j - i + 1$ possible keys inbetween (including both i and j) and each has equal probability of being the highest, the probability that either i or j is the greatest is $2/(j - i + 1)$. Therefore,

$$\begin{aligned}\mathbf{E}[A_{i,j}] &= \mathbf{Pr}[A_{i,j} = 1] \\ &= \mathbf{Pr}[p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \ldots, p_j\}] \\ &= \frac{2}{j - i + 1}.\end{aligned}$$

Hence, the expected number of comparisons made is

$$\begin{aligned}\mathbf{E}[X_n] &\leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}[A_{i,j}] \\ &= 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{2}{j - i + 1} \\ &= 3\sum_{i=1}^{n}\sum_{k=1}^{n-i}\frac{2}{k + 1} \\ &\leq 3\sum_{i=1}^{n} H_n \\ &= 3nH_n \in O(n \log n)\end{aligned}$$

4 Treaps

Let's go back to Treaps. We claim that the split code given in the last lecture for unbalanced trees doesn't need to be modified at all for Treaps.

Exercise 1. *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).*

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. Here is the algorithm:

```
1  fun join(T1, m, T2) =
2  let
3    fun singleton(k, v) = Node(Leaf, Leaf, k, v)
4    fun 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 $\texttt{join}'$ is a version of join that does not take a middle element. 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.

The cost of `split` and `join` on treaps is proportional to the depth of nodes in the tree. 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. The $\texttt{join}(T_1, m, T_2)$ code only traverses the right spine of T_1 or the left spine of T_2. The work and span is therefore proportional to the sum of the length of these spines. Similarly, the `split` operation only traverses the path from the root down to the node being split at. The work and span are proportional to this path length. Therefore, if we can show that the depth of all nodes are at $O(\log n)$ then the work and span of `join` and `split` are $O(\log n)$.