Lecture 16 — Treaps; Augmented BSTs

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

Lectured by Margaret Reid-Miller — 8 March 2012

Today:
- More on Treaps
- Ordered Sets and Tables
- Augmenting Balanced Trees

1 Today

Today we will continue looking at treaps and show that the expected depth of a treap is $O(\log n)$. The analysis is similar to the analysis of quicksort work. In doing so, we show that quicksort has expect span $O(\log^2 n)$. Then we will extend tables and sets interfaces to take advantage of ordering to enable more functionality Finally, we will give one example of augmenting balanced trees to implement additional operations efficiently.

Last time 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. Since a binary search tree is a dynamic data structure, it cannot change order in which operations are requested. So instead of randomizing the input order, it adds randomization so that the data structure itself is random.

2 Expected Depth of a Node in a Treap

Recall that a treap is a binary search tree (BST) in which we associate with each key a random priority. The tree is maintained so that the priorities associated with the keys are in (max) heap order, i.e. the priority at a node is larger than the priorities of both of its children. We will now analyze the expected depth of a key in the tree. This analysis is similar to the analysis we did for quicksort.

Consider a set of keys K and associated priorities $p : \texttt{key} \rightarrow int$. We assume the priorities are unique. Consider the keys laid out in order, and as with the analysis of quicksort, we use i and j to

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

refer to keys at two positions 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 in a BST can be either less than or greater than node 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.) Now 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?

In the first case j cannot be an ancestor of i since i has a higher priority, and $A_i^j = 0$. In the second case $A_i^j = 0$, 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. Finally in the third case, 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).

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. Recall from last lecture that the recursion tree for quicksort is identical to the structure of the corresponding treap (assuming the same keys and priorities).

Now we have

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

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

Exercise 1. *Including constant factors how does the expected depth for the first key compare to the expected depth of the middle ($i = n/2$) key.*

Split and Join on Treaps

As mentioned last week for any binary tree all we need to implement is split and join and these can be used to implement the other operations.

We claimed that the split code given in lecture 14 for unbalanced trees does not need to be modified at all for Treaps.

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).

```
fun join($T_1, m, T_2$) =
let
  fun singleton($k, v$) = Node(Leaf, Leaf, $k, v$)
  fun join'($T_1, T_2$) =
    case ($T_1, T_2$) of
      (Leaf, _) ⇒ $T_2$
    | (_, Leaf) ⇒ $T_1$
    | (Node($L_1, R_1, k_1, v_1$), Node($L_2, R_2, k_2, v_2$)) ⇒
        if (priority($k_1$) > priority($k_2$)) then
          Node($L_1$, join'($R_1, T_2$), $k_1, v_1$)
        else
          Node(join'($T_1, L_2$), $R_2, k_2, v_2$)
in
    case $m$ of
      NONE ⇒ join'($T_1, T_2$))
    | SOME($k, v$) ⇒ join'($T_1$, join'(singleton($k, v$), $T_2$))
end
```

In the code `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.

Because the keys and priorities determines a treap uniquely, repeated splits and joins on the same key, results in the same treap. This property is not always true of most other kind of balanced trees; the order that operations are applied can change the shape of 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. What $\texttt{join}'(T_1, T_2)$ does is to interleave pieces of the left spine of T_1 with pieces the right spine of T_2, where the size of each piece depends on the priorities.

Theorem 2.1. *For treaps the cost of* $\texttt{join}(T_1, m, T_2)$ *returning* T *and of* $\texttt{split}(T)$ *is* $O(\log |T|)$ *expected work and span.*

Proof. 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. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$. For $\texttt{join}(T_1, m, T_2)$ the code only traverses 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 which is expected $O(\log |T|)$. □

We note that these bounds for `split` and `join` also give us the worst-case expected $O(m \log(n/m))$ work bounds for `union` and related functions in expectation.

2.1 Expected span of quicksort and height 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 height of a treap is. The reason is, the height of a treap is the same as the depth of the node with maximum depth. As you have saw in lecture 13, $\mathbf{E}\left[\max_i\{A_i\}\right] \neq \max_i\{\mathbf{E}\left[A_i\right]\}$. We use a similar analysis used to analyze the work of `SmallestK` to find the expected span of quicksort and the expected height of treaps.

Recall that in randomized quicksort, at each recursive call, we partition the input sequence S of length n into three subsequences L, E, and R, such that elements in the subsequences are less than, equal, and greater than the pivot, respectfully. Let $X_n = \max\{|L|, |R|\}$, which is the size of larger subsequence; The span of quicksort is determined by the sizes of these larger subsequences. For ease of analysis, we will assume that $|E| = 0$, as more equal elements will only decrease the span. As this partitioning uses `filter` we have the following recurrence for span:

$$S(n) = S(X_n) + O(\log n)$$

Let $\overline{S}(n)$ denote $\mathbf{E}\left[S(n)\right]$. As we did for `SmallestK` we will bound $\overline{S}(n)$ by considering the $\mathbf{Pr}\left[X_n \leq 3n/4\right]$ and $\mathbf{Pr}\left[X_n > 3n/4\right]$ and use the maximum sizes in the recurrence to upper bound $\mathbf{E}\left[S(n)\right]$. Now, the $\mathbf{Pr}\left[X_n \leq 3n/4\right] = 1/2$, since half of the randomly chosen pivots results in the larger partition to be at most $3n/4$ elements: Any pivot in the range $T_{n/4}$ to $T_{3n/4}$ will do, where T is the sorted input sequence.

So then, by the definition of expectation, we have

$$\begin{aligned}
\overline{S}(n) &=\leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{S}(i) + c\log n \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{S}(\tfrac{3n}{4}) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{S}(n) + c \cdot \log n \\
&\leq \tfrac{1}{2}\overline{S}(\tfrac{3n}{4}) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\implies (1-\tfrac{1}{2})\overline{S}(n) \leq \tfrac{1}{2}\overline{S}(\tfrac{3n}{4}) + c\log n \\
&\implies \overline{S}(n) \leq \overline{S}(\tfrac{3n}{4}) + 2c\log n,
\end{aligned}$$

which we know is a balanced cost tree and solves to $O(\log^2 n)$.

That is, with probability $1/2$ we will be lucky and the subproblem size will go down by at least $3n/4$ and with probability $1/2$ we will be unlucky and we have to start again. In the end, the expected span is twice what it would be if we could guarantee partition sizes of $n/4$ and $3n/4$.

The analysis for the expected height of a treap is almost the same. But this time L and R represent the nodes to the left and the right of the root, the element with the maximum priority. The height of the whole treap is the depth of the node with maximum depth. It is not hard to show, this node will be in the root's subtree of maximum size, and the depth of this node is 1 plus its depth in this subtree. Again, let $X_n = \max\{|L|, |R|\}$. We get the following recurrence for the height H of the treap.

$$D(n) = H(X_n) + 1,$$

and the expected depth $\overline{H}$ is

$$\begin{aligned}
\overline{H}(n) &=\leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{H}(i) + 1 \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{H}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{H}(n) + 1 \\
&\leq \tfrac{1}{2}\overline{H}(3n/4) + \tfrac{1}{2}\overline{H}(n) + 1 \\
&\implies \overline{H}(n) \leq \overline{H}(3n/4) + 2 = O(\log n)
\end{aligned}$$

Thus, the expected height of a treap 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 $1/n$[1]. That is, for large n a treap with random priorities has height $O(\log n)$ with *high probability*. Since the recursion tree for quicksort has the same distribution as a treap, this also gives the same bounds for the depth of recursion of quicksort.

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

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

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.

3 Ordered Sets and Tables

The set and table interfaces described so far do not give any operations that make use of the ordering of the elements, which allows them to be defined on types that don't have a natural ordering. These interfaces are also well suited for an implementation based on hash tables. In many applications, however, it is useful to take advantage of the order of the keys. For example in a database one might want to find all the customers who spent between \$50 and \$100, all emails in the week of Aug 22, or the last stock transaction before noon on October 11th. Here we assume the data is organized by transaction value, date or any other ordered key.

For these purposes we can extend the operations on sets and tables with some additional operations that take advantage of ordering. Here we will just describe the operations on ordered sets. The operations on ordered tables are completely analogous.

Definition 3.1. For a totally ordered universe of elements $\mathbb{U}$ (e.g. the integers or strings), the *Ordered Set* abstract data type is a type $\mathbb{S}$ representing the powerset of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

all operations supported by the Set ADT, and

`last`(S)	:	$\mathbb{S} \to \mathbb{U}$	=	$\max S$
`first`(S)	:	$\mathbb{S} \to \mathbb{U}$	=	$\min S$
`split`(S,k)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S} \times bool \times \mathbb{S}$	=	as with trees
`join`(S_1,S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	as with trees
`getRange`(S,k_1,k_2)	:	$\mathbb{S} \times \mathbb{U} \times \mathbb{U} \to \mathbb{S}$	=	$\{k \in S \mid k_1 \le k \le k_2\}$

Note that `split` and `join` are the same as the operations we defined for binary search trees. Here, however, we are abstracting the notion to ordered sets.

If we implement an Ordered Set using trees, then we can use the tree implementations of `split` and `join` directly. Implementing `first` is straightforward since it only requires traversing the tree down the left branches until a left branch is empty. Similarly `last` need only traverse right branches. The `getRange` operation can easily be implemented with two calls to `split`.

4 Augmenting Balanced Trees

Often it is useful to include additional information beyond the key and associated value in a tree. In particular the additional information can help us efficiently implement additional operations. Here we will consider the first of two examples: (1) locating positions within an ordered set or ordered table, and (2) keeping "reduced" values in an ordered or unordered table. Later we will consider the second example.

4.1 Tracking Sizes and Locating Positions

Let's say that we are using binary search trees (BSTs) to implement ordered sets and that in addition to the operations already described, we also want to efficiently support the following operations:

$$\begin{array}{llll}
\texttt{rank}(S,k) & : \mathbb{S} \times \mathbb{U} \to int & = & |\{k' \in S \mid k' < k\}| \\
\texttt{select}(S,i) & : \mathbb{S} \times int \to \mathbb{U} & = & k \text{ such that } |\{k' \in S \mid k' < k\}| = i \\
\texttt{splitIdx}(S,i) & : \mathbb{S} \times int \to \mathbb{S} \times \mathbb{S} & = & (\{k \in S \mid k < \texttt{select}(S,i)\}, \\
 & & & \ \{k \in S \mid k \geq \texttt{select}(S,i)\})
\end{array}$$

In the previous lectures the only things we stored at the nodes of a tree were the left and right children, the key and value, and perhaps some balance information. With just this information implementing the `select` and `splitIdx` operations requires visiting all nodes before the i^{th} location to count them up. There is no way to know the size of a subtree without visiting it. Similarly, `rank` requires visiting all nodes before k. Therefore all these operations will take $\Theta(|S|)$ work. In fact even implementing `size`(S) requires $\Theta(|S|)$ work.

To fix this problem we can add to each node an additional field that specifies the size of the subtree. Clearly this makes the `size` operation fast, but what about the other operations? Well it turns out it allows us to implement `select`, `rank`, and `splitIdx` all in $O(d)$ work assuming the tree has depth d. Therefore for balanced trees the work will be $O(\log |S|)$. Lets consider how to implement `select`:

```
fun select(T, i) =
  case expose(T) of
    NONE ⇒ raise Range
   | SOME(L, R, k) ⇒
      case compare(i, size L) of
        LESS ⇒ select(L, i)
       | EQUAL ⇒ k
       | GREATER ⇒ select(R, i − (size L) − 1)
```

To implement `rank` we could simply do a split and then check the size of the left tree. The implementation of `splitIdx` is similar to `split` except when deciding which branch to take, base it on the sizes instead of the keys. In fact with `splitIdx` we don't even need `select`, we could implement it as a `splitIdx` followed by a `first` on the right tree.

We can implement sequences using a balanced tree and this approach.