High Probability Bounds for Treaps (and Quicksort)

So far, we have argued about the expected work of quick sort and the expected depth of a node in a treap, but is this good enough? Let's say 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).

Fortunately, in many cases we can use expectations to prove much stronger bounds at least when we have independent events. These bounds are what we call high probability bounds, again as mentioned in an earlier lecture. Basically, it says that we guarantee some property with probability very close to 1 — in fact, so close that the difference is only inverse polynomially small in the problem size. To get the high probability bound, we will again use the Chernoff bound that we saw in some previous lecture:

**Theorem 1 (Chernoff Bound)** Given a set of independent random variables $X_1, \ldots, X_n$, each taking a real value between 0 and 1, with $X = \sum_{i=1}^{n} X_i$ and $E [X] = \mu$, then for any $\lambda > 0$

$$
\Pr \{ X > (1 + \lambda)\mu \} < \exp \left( -\frac{\mu \lambda^2}{2 + \lambda} \right).
$$

Recall that, the factor $\lambda$ specifies a distance away from the expectation, or, the weighted average. So, for example, by setting $\lambda = .2$, we are asking what the probability is that the actual result is more than 20% larger than the expected result. As expected, the farther we are away from the mean, the less likely our value will be — this probability decays exponentially fast in (roughly) $\lambda^2$.

Also recall that, the Chernoff bound requires that the random variables that you are summing together are independent, and in face, independence is key in getting this sharp bound —
it makes it unlikely that the random variables will “collude” to pull the average one way or another. Markov’s,\(^1\) in contrast, does not require independence and therefore gives a much weaker bound.

We now use the Chernoff bounds to get high-probability bounds for the depth of any node in a treap. Since we said the recursion tree for quicksort has the same structure as a treap, this will also give us high probability bounds for the depth of the quicksort tree, which can then be used to bound the span of quick sort with high probability.

Recall that the random variable \(B_{ij}\) indicates that \(j\) is an ancestor of \(i\). It turns out all we have to do is argue that for a given \(i\) the random variables \(B_{ij}\) are independent. If they are, we can then use the Chernoff bounds. So, let’s think about why \(A_{ik}\) and \(A_{ij}\) are independent, for \(j \neq k\). Let’s just consider the \(i < k < j\) (the others are true by a symmetrical argument), and scan from \(i\) forward. Each time we look at the \(j\) whether it is an ancestor only depends on whether its priority is larger than all priorities from \(i\) to \(j - 1\). It does not depend on the relative ordering of those priorities, and certainly whether \(B_{ik}\) is 1 or not does not affect the value of \(B_{ij}\). (Similarly, the value taken by \(B_{ij}\) does not affect the value of \(B_{ik}\).) They are therefore independent.

So we can use Chernoff bound to analyze the depth of a key \(i\). Let \(B_i = \sum_{j=1}^{n} B_{ij}\) (recall that this corresponds to counting the number of ancestors of \(i\) in the treap and thus the depth). As derived earlier the expectation \(\mu = \mathbb{E}[B_i]\) is \(H_i + H_{n-i-1}\) which lies between \(\ln n\) and \(2 \ln n\). If we set \(\lambda = 4\), we have

\[
\Pr \{X > (1 + 4)\mu\} < e^{-\frac{\mu(4)^2}{2\lambda^2}}
\]

Now since \(\mu \geq \ln n\) we have

\[
\Pr \{X > 5\mu\} < e^{-2\ln n} = \frac{1}{n^2}
\]

This means that the probability that a node is deeper than five times its expectation is at most \(\frac{1}{n^2}\), which is very small for reasonably large \(n\) (e.g. for \(n = 10^6\) it is \(10^{-12}\)). Since the expected depth of a key is at most \(2 \ln n\), we have

**Theorem 2** For a treap with \(n\) keys, the probability that a key \(i\) is deeper than \(10 \ln n\) is at most \(1/n^2\).

Note that this just gives us the probability that any one key is deeper than 5 times its expectation. To figure out the worst case over all keys, we have to take a union bound over the probability

\(^1\)Markov’s inequality states that \(\Pr \{X > (1 + \lambda)\mu\} < \frac{1}{1 + \lambda}e^\mu\).
that any one key is deeper than $5$ times the expectation; that is, multiply this probability by the number of keys. More formally, let $E_x$ be the “bad” event that the key $x$ is deeper than $5$ times its expectation. So then, by applying the union bound, we have

$$\Pr \{ \text{any key is deeper than } 10 \ln n \} = \Pr \{ \exists x. E_x \} \leq \sum_x \Pr \{ E_x \} \leq n \times \frac{1}{n^2} = \frac{1}{n}.$$ 

This gives us the following theorem:

**Theorem 3** *For a treap with* $n$ *keys, the probability that any key is deeper than* $10 \ln n$ *is at most* $1/n$.

Note that the Chernoff bounds are sloppy, and the actual bounds can be much stronger. Since we are interested in upper bounds on the probability, this is OK (our upper bounds are just a bit loose).

**Treap Operations**

We now consider some internal functions on treaps that are useful for building up other functions, such as insert and delete, but also many other useful functions that are more parallel, such as difference and union on sets. Note that, we will discuss the implementation of these functions in terms of what we need for treaps, but they are more broadly applicable to other binary search trees (although some details may vary, depending on the functions). In all functions, we assume for a given key $k$, it is implicitly associated with some priority $\text{priority}(k)$ that are chosen uniformly at random. (Note that, these priorities are not provided by the user. They are something internally generated by the treap data structure. This is important, because the bound we guarantee depends on the fact that these are chosen at random.)

**Split**($T, k$): Given a treap $T$ and a key $k$, Split divides $T$ into two treaps, 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 value $v$ then Split returns the data, and otherwise it returns nothing. That is, Split($T, k$) returns ($L, v, R$) where $L$ contains keys less than $k$ and $R$ contains keys greater than $k$, and $v$ is the value associated with $k$ in $T$, which could be nil.

**Join**($L, m, R$) : Join takes a left treap $L$, an optional middle key-data pair $m$, and a right treap $R$. 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 greater than any key in $L$ and less than any key in $R$. It creates a new treap which is the union of $L$, $R$ and the optional $m$. 

3
For both \textsc{Split} and \textsc{Join}, we assume that the treaps taken and returned by the functions obey the treap criteria. That is, they satisfy both the BST and Heap properties required by treaps. (If you are implementing these functions for other kinds of BST, they satisfy whatever balance criteria necessary for the particular BST.)

To implement \textsc{Split} and \textsc{Join}, it’s useful to define the following internal functions:

\textbf{EXPOSE}(T): Given a BST $T$, if $T$ is empty it returns nothing. Otherwise it returns the left child of the root, the right child of the root, and the key and data stored at the root.

\textbf{TREE}(L, R, root): Given two BSTs $L$ and $R$ and a node $root$ create a BST. Assume that all invariants needed are satisfied. This is different from \textsc{Join} for two reasons. $root$ is never null, and we already assume that all other invariants of height, color, etc are satisfied.

Note these are simple functions and can be done in constant time — \textsc{Expose} simply expose the root of the given treap $T$ (and its subtrees). If $T$ is a legal treap, the return values should also be legal treaps. \textsc{Tree} simply construct a treap out of the given inputs (i.e., connect the $L$ and $R$ treaps to $root$), so the $root$ is required to have the highest priority than any other keys in $L$ and $R$ already.

With these functions we can easily implement insert and delete. In order to insert key $k$, you can split around key $k$ and then join by adding key $k$ as the middle element. For delete, you can split using key $k$ and then join with no key in the middle.

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 \textsc{Join} since we do not require that all the keys in one appear after the keys in the other.

\textbf{UNION}(T_1, T_2)
1. if $T_1$ is empty,
2. then return $T_2$
3. $(L_1, R_1, (k_1, v_1)) \leftarrow \text{EXPOSE}(T_1)$
4. $(L_2, v_2, R_2) \leftarrow \text{SPLIT}(T_2, k_1)$
5. $L \leftarrow \text{spawn} \text{UNION}(L_1, L_2)$
6. $R \leftarrow \text{UNION}(R_1, R_2)$
7. \textbf{sync}
8. return \textsc{Join}$(L, (k_1, v_1), R)$

Let’s think for a second why this is correct — for \textsc{Union}, one of the correctness criteria is that we merge the two sets and remove duplicates. In this version, if a key is in both trees, you keep the
value from the first tree. How about the BST and Heap properties? Since all keys smaller are still on the left and all keys larger are still on the right, we are good. The JOIN operation will make sure that the Heap property is maintained, as we will see later. We note, however, that as written the code only matches our desired bounds if $|T_1| \leq |T_2|$ (i.e., when you call UNION at the top-level, check the sizes and pass the small treap as the first argument).

The code for taking the difference (i.e., intersection) is quite similar; we simply discard keys that are in both treaps instead of keeping the first key.

**Exercise 1** Write the code for intersection.

**Implement Split and Join**

\begin{verbatim}
SPLIT(T, k)
    1  if T is empty
    2    then return nil
    3  (L, R, (k', v)) ← EXPOSE(T)
    4  if k' = k
    5    then return (L, v, R)
    6  if k < k'
    7    then (L', r, R') ← SPLIT(L, k)
    8    return (L', r, TREE(R', R, (k', v)))
    9  if k > k'
10    then (L', r, R') ← SPLIT(R, k)
11    return (TREE(L, L', (k', v)), r, R')
\end{verbatim}

Note that this SPLIT code does not look anything special; it can be SPLIT code for any regular BST, because it doesn’t do anything for rebalancing or checking priority. But for treap, this is correct. Let’s think why. Again, we will check if it indeed implements the desired functionality and if the return values satisfy both the BST and Heap properties.

The JOIN code, however, does need to account for the priority explicitly. We will check the priorities of the two roots and use whichever is greater as the new root. In this code, we assume that you are only given two trees and no middle element. If you do have a middle element, you can call this function twice, once with the left (or right) tree and the middle element and then with the result and the remaining tree.
JOINHELPER($T_1, T_2$)
1  if either $T_1$ or $T_2$ is empty, return the other tree.
2  ($L_1, R_1, (k_1, v_1)$) ← EXPOSE($T_1$)
3  ($L_2, R_2, (k_2, v_2)$) ← EXPOSE($T_2$)
4  if priority($k_1$) > priority($k_2$)
5    then return TREE($L_1, JOINHELPER(R_1, T_2), (k_1, v_1)$)
6    else return TREE($JOINHELPER(T_1, L_2), R_2, (k_2, v_2)$)

**Theorem 4** For treaps the cost of JOIN($T_1, m, T_2$) returning $T$ and of SPLIT($T$) is $O(\log |T|)$ expected work and span.

*Proof.* The cost of SPLIT is proportional to the depth of the node where we are splitting at. Since the expected depth of a node is $O(\log n)$, the expected cost of split is $O(\log n)$. For JOIN($T_1, m, T_2$) note that the code only follows $T_1$ down the right child and terminates when the right child is empty. Similarly it only follows $T_2$ down the left child and terminates when it is empty. 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 $O(\log |T|)$.

In the next lecture, we will see that these bounds for SPLIT and JOIN give us the $O(m \log (1 + n/m))$ work bounds for UNION and related functions in expectation.