1 Let’s Randomize!

- Thinking about our tree construction, if we receive the elements to go into the tree in a random order, we get a pretty good tree on average.

- That is, the average depth (and therefore average insert and search time) for a node is not too big.

- Unfortunately, this result depends strongly on the assumption that we receive elements to insert in random order by their value.

- What if we could ensure this?

We’re going to construct an algorithm that always gives us the tree that we’d get if we received elements in a random order!

- Suppose we are willing to wait until we receive all \( n \) elements before constructing the tree.

- Then we can scramble the elements into a random order before we insert them.

- To run this algorithm, we need to use a random number generator.

- Algorithms that use random numbers to determine their execution are called randomized algorithms.

- In this case, we might produce a different result (tree) every time we run, even if the input is the same each time.

- By our previous result, the tree produced has a good depth on average (over our choices of random numbers to reorder the elements), no matter what order we receive the inputs.

- In general, we use randomization to make the average behavior of an algorithm good regardless of the input.

- We may also be able to make bad behavior a provably low-probability event.

- We’ll see an example of the latter, stronger kind of claim next.
2 Randomization Nearly Always Produces Good Trees

Saying that randomization “nearly always” avoids bad behavior is a stronger claim than that it is good on average.

- In the context of the tree, our previous claim was that a node has low depth on average.
- However, some nodes might have high depth and so might be consistently expensive to look up – even if the average is still good.
- A stronger claim would be that with high probability, no node has high depth, so all are cheap to look up.
- If “high depth” means $O(\log n)$, then this claim implies that with high probability, the tree is balanced.

Let’s formalize our claim.

- **Thm:** suppose we insert $x_1 < x_2 < \ldots x_n$ into a binary tree in a random order. Then for any $\delta > 0$, the tree has depth $O(\log(n/\delta))$ with probability at least $1 - \delta$.

  (Hence, if we set $\delta$ to any constant “failure” probability, e.g. 1%, the chance of the tree not failing, i.e. having logarithmic depth, is at least $1 - \delta$.)

- **Pf:** As before, we’ll combine some understanding of the problem (cool!) with some probability hacking (yuck!).

  - First, imagine the elements $x_1 \ldots x_n$ laid out on a line in order by value.
  - At each step, we pick a random element to put into the tree and delete it from the line.
  - Note that this process matches the condition of our theorem – any particular ordering is chosen with the same probability
    \[
    \frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{1}{n-2} \cdots \frac{1}{2} \cdot 1 = \frac{1}{n!}.
    \]

  - Let’s consider what happens to a fixed element $x_i$.
    - If we pick some $x_j < x_i$ before $x_i$, then all the elements to the left of $x_j$ cannot be ancestors of $x_i$ – if $x_i$ goes below $x_j$, it would go into a different subtree than these elements.
    - Similarly, if we pick some $x_k > x_i$ before $x_i$, then all the elements to the right of $x_k$ cannot be ancestors of $x_i$.
• Finally, note that the number of “picks” prior to picking \( x_i \) then upper-bounds the depth of \( x_i \) in the tree.

We now consider how many picks will happen before we insert \( x_i \) into the tree.

• Say that a **good pick** for \( x_i \) is one that eliminates at least half the elements to its left or right.

• If we make \( 2 \log n \) good picks, then we must eliminate all the elements except \( x_i \), since it takes at most \( \log n \) good picks to eliminate all elements to its left, and similarly to its right.

• Moreover, each time we do not pick \( x_i \), we **make a good pick with probability at least** 1/2.

• (Indeed, if we choose a random element to the left of \( x_i \), then we have probability at least 1/2 of picking one with half the other elements to its left, and similarly if we choose an element to the right of \( x_i \).)

• So, to summarize, we know that
  1. the depth of \( x_i \) is at most the number of picks prior to inserting \( x_i \);
  2. it takes only \( 2 \log n \) good picks before \( x_i \) must be inserted;
  3. each pick is independently good with probability at least 1/2.

• **Question**: how many total picks are needed before we’ve almost certainly made \( 2 \log n \) good picks, and hence are forced to insert \( x_i \)?

OK, time for some probability hacking.

• Picking is a random binomial process – each time we pick, we flip a coin, and with probability \( p \geq 1/2 \) it comes up heads (“good”).

• If we flip \( k \) times, the expected number of heads is \( pk \).

• Intuitively, it’s very unlikely that we would need way more than \( k \) flips to get \( pk \) heads.

• We can formalize this intuition with the following **Chernoff bound**:

• **Thm**: suppose we flip a coin \( k \) times, and that each flip yields heads with probability \( p \). Then the probability that \( k \) flips yield fewer than \( (1-\epsilon)pk \) heads is at most \( e^{-\frac{1}{2}\epsilon^2 pk} \).

• (See the end of Chapter 13 in your book for the proof.)

• Now, let \( \delta \) be the failure probability in the statement of our original theorem (i.e., the prob that we don’t get a balanced tree.)

• We will use the Chernoff bound to argue that with high probability, \( O(\log(n/\delta)) \) picks are enough to obtain \( 2 \log n \) good picks.

• Set \( p = 1/2, \epsilon = 1/2, \) and \( k = 16 \log_2 \frac{n}{\delta} \).
Then the probability that \( k \) flips contains fewer than

\[
(1 - \epsilon)pk = \frac{1}{2} \cdot \frac{1}{2} \cdot 16 \log_2 \frac{n}{\delta}
\]

\[
= 4 \log_2 n + 4 \log_2 \frac{1}{\delta}
\]

\[
> 2 \log_2 n
\]

heads is at most

\[
e^{-\frac{1}{2} \left( \frac{1}{2} \right)^2 \frac{n}{\delta}} = \left( \frac{\delta}{n} \right)^{1/\ln 2}
\]

\[
< \frac{\delta}{n},
\]

where the last step follows because \( 1/\ln 2 > 1 \) and \( \delta/n < 1 \).

Conclude that \( x_i \) has more than \( O(\log(n/\delta)) \) ancestors in the tree with probability \( < \delta/n \).

We are almost done.

- The above argument shows that the depth of any one \( x_i \) is at most \( O(\log(n/\delta)) \) with a certain probability.
- But we originally claimed that all nodes have at most this depth with probability \( 1 - \delta \).
- Can we take the last step and prove the original claim?
- Yes, with the union bound.
- **Thm**: if we have events \( e_1 \ldots e_n \), such that event \( e_i \) occurs with probability \( p_i \), then the probability that at least one event occurs is at most \( \sum_{i=1}^n \Pr(e_i) \).
- (This theorem holds even when the events are not independent! It’s a special case of the general inclusion/exclusion bounds a.k.a. Bonferroni inequalities, for joint probability.)
- Hence, if each \( x_i \) fails to have depth at most \( O(\log(n/\delta)) \) with probability \( < \delta/n \), then at least one \( x_i \) fails to have this depth with probability \( < \sum_{i=1}^n \delta/n = \delta \).
- Conclude that with probability \( > 1 - \delta \), every \( x_i \) has depth at most \( O(\log(n/\delta)) \) as claimed. QED

### 3 Dynamic Construction of Good Trees – Treaps

Randomized tree construction is no problem if we can see all the elements before we build the tree. But this is a severe limitation!
• If we have access to all the elements initially, we could deterministically build a tree of optimal depth – make the middle element the root, and recursively build trees from left and right halves.

• What if we instead receive the elements in some arbitrary order and have to insert each element immediately?

• We’ll now devise a scheme to obtain the same tree that we would have gotten from a random ordering if we knew all the elements ahead of time.

The key idea is to assign each element a priority.

• Let \( N \) be a number \( \gg n \).

• As each element \( x_i \) arrives, assign it a random priority in the range \([0..N - 1]\).

• As long as we don’t get more than about \( \sqrt{N} \) elements, all priorities will almost certainly be distinct.

• Now suppose we were to wait until all elements show up, and then insert them in increasing order by priority.

• The result is the same as if we were to insert elements in a random order.

• Moreover, the resulting tree has the property (*) that the priority of each node is less than that of its children, and hence less than all of its descendants.

• (This is because a child of a node must have been inserted after that node.)

• Lemma: Given unique priorities for each element \( x_i \), there is a unique valid BST on \( x_1 \ldots x_n \) that has property (*).

• Pf: Suppose \( x_i \) has priority \( p_i \), and assume WLOG that \( p_1 < p_2 \ldots < p_n \) (i.e. some fixed priority order).

• Consider the trees \( T_i \) obtained by inserting \( x_1 \ldots x_i \) in priority order.

• \( T_1 \) is clearly unique – there is only one single-node tree, and hence only one with property (*).

• Now suppose inductively that \( T_i \) is unique. In any tree \( T' \) on \( x_1 \ldots x_{i+1} \) with property (*), element \( x_{i+1} \) must occur at a leaf of \( T' \), since its priority is greater than that of any other element in the tree.

• Hence, if we remove \( x_{i+1} \) from \( T' \), we are left with a BST on \( x_1 \ldots x_i \) with property (*), which by our IH must be \( T_i \).

• But adding \( x_{i+1} \) to \( T_i \) places it at a leaf uniquely determined by its value (else we would not have a valid BST).

• Hence, \( T_{i+1} \) is the unique BST on \( x_1 \ldots x_{i+1} \) with property (*). QED

We now know that if we assign unique priorities to each element \( x_i \) and compute a BST on them so that every node has priority less than its children, we get a tree equivalent to inserting the elements in priority order!
• We can build this tree even if the elements arrive out of order, if we’re willing to rearrange it dynamically during insertion.

• Think about a binary min-first heap of elements keyed by priority.

• Whenever a new item shows up, we put it at the end of the heap, then “bubble it up” until its priority is greater than that of its parent.

• We can do something similar to a BST.

• When an item $x$ with priority $p_x$ comes in, we first place $x$ at the leaf needed to maintain a valid BST.

• But now $x$’s parent $y$ might have priority $p_y > p_x$.

• We can swap $x$ and $y$ while maintaining a valid BST by one of the following rotation operations (which you might recall from red-black trees):

  • **Lemma**: rotation takes constant time and maintains the BST property for the resulting tree. (Proof left as exercise.)

  • **Lemma**: If the subtree rooted at $y$ satisfied property (*) before rotation, except that $p_x < p_y$, then the subtree rooted at $x$ satisfies property (*) after rotation.

  • **Pf**: Before rotation, subtrees $T_1$ and $T_3$ are direct children of $y$ and $x$ respectively.

  • Their parents do not change after rotation, so if they satisfied property (*) internally and w/r to their parents before, they still do.

  • Moreover, $T_2$’s parent changes from $x$ to $y$.

  • All elements in $T_2$ had priority $> p_x$ before, so they all have priority $> p_y$ as well. Hence property (*) still holds. QED

  • After $O(\log n)$ rotations, each in constant time, $x$ bubbles up to its proper place to maintain property (*).

  • As we showed previously, the resulting BST is the same as if we had inserted all the elements in priority order!

This method for building almost-certainly-balanced BSTs is called a *treap* – we apply heap-like priority maintenance using random priorities to obtain balance. For more on treaps, including how to delete elements, see [https://en.wikipedia.org/wiki/Treap](https://en.wikipedia.org/wiki/Treap).
4 One More Thing – QuickSort

Our analysis of randomized tree construction also tells us something about sorting!

- The QuickSort algorithm has a recursive structure similar to BST construction.
- Initially, we are given an array $A$ to sort. Suppose for simplicity that all the elements of $A$ are distinct.
- We pick a partition element $A[x]$ and divide $A$ into subsets $A_L$ made up of all elements $< A[x]$, and $A_R$ made up of elements $> A[x]$.
- Finally, we recursively sort $A_L$ and $A_R$ and place $A[x]$ between them.
- Now suppose that we pick our partition element randomly from $A$.
- The same analysis that we did for randomized trees tells us that with probability $1 - \delta$, every element will be picked, and hence placed correctly in the sorted order, by the time the recursion reaches depth $O(\log(n/\delta))$.
- Dividing $A$ into $A_L$ and $A_R$ takes time $O(|A|)$.
- At each level of the tree, all the arrays we need to divide have total length at most $O(n)$, so the total time to divide them all is also $O(n)$.
- Conclude that with probability $1 - \delta$, QuickSort will do total work $O(n \log(n/\delta))$.
- We can also prove the slightly weaker result that the expected running time of QuickSort is $O(n \log n)$. 