Treaps

In the previous lecture, we designed a generic interface for balanced binary search trees and showed how a wide range of operations such as insertion, splitting and union can be implemented assuming the existence of a balancing function, `rebalance`. To complete the story, we must now choose a concrete balancing scheme and specify how `rebalance` is implemented.

Many classical balanced trees, such as AVL trees or red–black trees, enforce balance using carefully designed structural invariants. While these invariants guarantee logarithmic height, they can be subtle to define and maintain, especially in a purely functional and parallel setting.

In this course, we will instead focus on a randomized balancing scheme known as a **treap**. Treaps achieve logarithmic height *with high probability* using randomness, while remaining conceptually simple and well-suited to functional and parallel implementations.

1 Treaps

The term "treap" is a portmanteau of the words "tree" and "heap", reflecting the two invariants that treaps maintain.

Definition: Treap Invariants

A treap is a binary search tree in which each key k is associated with a priority $p(k)$. The tree must satisfy:

1. **BST Invariant (by key)**: For every node with key k, all keys in its left subtree are less than k, and all keys in its right subtree are greater than k.
2. **Heap Invariant (by priority)**: For every node with key k and priority $p(k)$, the priority $p(k)$ is greater than the priorities of its children.

We will assume that the priorities are unique to simplify the analysis, although it is not strictly necessary. Note that because of the heap invariant, the root of a treap is always the key with the highest priority among all keys stored in the tree.

The two invariants together imply a powerful structural characterization of treaps:

Claim: Uniqueness of Treap Shape

Given a set of distinct keys with distinct priorities, a treap is the **unique** binary tree that is simultaneously a binary search tree ordered by keys and a heap ordered by priorities.

Equivalently, given a fixed set of keys with assigned priorities, the treap is completely determined. No additional balancing decisions are required: the interaction between the BST and heap invariants uniquely fixes the tree shape.

Let's back up and consider why this matters. Given a set of n keys, there are exponentially many valid binary search trees that could store them[1]. Some of these trees are well balanced, while others are extremely unbalanced, and we'd like to get one of the balanced ones. The key observation behind treaps is that assigning priorities induces a unique BST shape. Choosing priorities is therefore equivalent to choosing one particular BST from among all possible BSTs.

The crucial idea is to choose these priorities *at random*. When priorities are assigned independently and uniformly at random, the resulting treap is unlikely to resemble one of the pathological, highly unbalanced trees. Instead, with high probability, the induced BST is balanced!

Remark: Random priorities favor balanced trees

We said before that a given assignment of priorities induces a unique treap shape, which was crucial to our intuition for treaps. Interestingly, and importantly, the converse is not true. Given a tree shape, there may be many different priority permutations that produce it. This asymmetry is not just a quirk, its actually *essential* for treaps to work.

As a result, sampling priorities uniformly at random does not sample tree shapes uniformly. In fact, it induces a highly non-uniform distribution over shapes. This is a good thing: sampling uniformly from all binary search tree shapes would frequently produce highly unbalanced trees, whereas treaps are strongly biased toward balanced ones.

To see this concretely, consider a treap on the keys $\{1, \ldots, 7\}$. The completely left-skewed tree (a decreasing chain) can occur only if priorities are assigned in strictly increasing order, a single permutation, so probability $1/7! = 1/5040$, making this shape extremely unlikely. The same is true for the completely right-skewed tree. In contrast, a perfectly balanced tree is compatible with many different priority orderings, since priorities only need to respect relative order within each subtree. This happens with probability $1/63$, almost 100 times more likely!

Thus, when priorities are chosen uniformly at random, balanced trees arise far more frequently than degenerate ones.

2 Implementing Rebalance for Treaps

We will now implement `rebalance` for treaps, which gives us all the interface for a BST for free since all previous functions were implemented in terms of `rebalance`. For simplicity we define a helper function that computes the priority of a tree to be the priority of its root key. We will define the priority of an empty tree to be $-\infty$ since this satisfies the heap invariant.

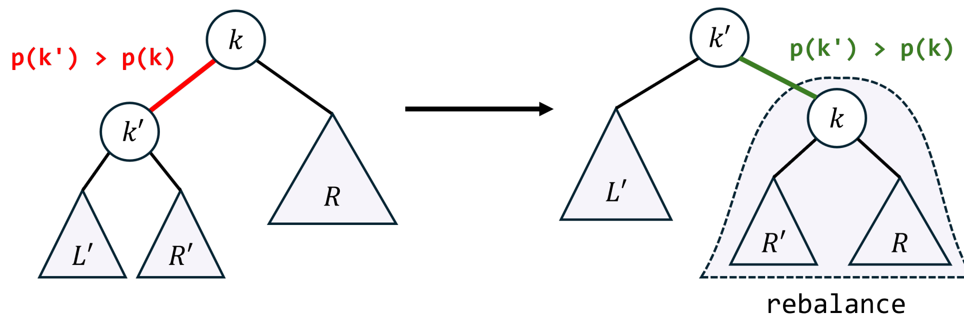
```
function p(T: BST<K,V>) -> int:
  match T with:
    case Empty: return -inf
    case Node(_,k,_,_,_): return p(k)
```

So what does `rebalance` for treaps do, exactly? Given a Node, it checks if the heap invariant is already satisfied at this node, by comparing the priority of the key to the priorities of the keys

[1]The number of valid BSTs on n keys is the nth *Catalan number*, which grows on the order of 4^n.

at the root of each subtree. If the priority of k is the greatest, rebalance can simply return the node as is. However, if either of the subtrees has a larger priority, rebalancing must occur.

Let's consider the case where $p(L) > p(k)$ and $p(L) > p(R)$. In this case, we inspect the subtrees of L. Note that L cannot be Empty given that $p(L) > p(k)$, and the priority of an empty tree is $-\infty$. We then make L the new root, and move k and R' into the right subtree and recursively `rebalance`. The left subtree of L stays the same. Conceptually, this transformation is exactly a single tree rotation (just like AVL and red–black trees) that restores the heap invariant while preserving the BST invariant. The case when $p(R)$ has the greatest priority of the three is symmetric.



Algorithm: Treap: rebalance

```
fun rebalance(T: BST<K,V>) -> BST:
  match T with:
    case Empty: return Empty
    case Node(L,k,v,s,R):
      if p(k) > p(L) and p(k) > p(R): return T
      else if p(L) > p(R):
        Node(L', k', v', _, R') = L
        return makeNode(L',k',v',rebalance(makeNode(R',k,v,R)))
      else:  // p(L) < p(R)
        Node(L', k', v', _, R') = R
        return makeNode(rebalance(makeNode(L,k,v,L'),k',v',R'))
```

Recall the `makeNode` function is a helper function which allows us to create a node out of two subtrees and the key and value for a given node. The main purpose of it is to fill in the size field of the node which is computed from the size of the subtrees.

And with one function, we are done with implementing a treap! There is no need to implement any further functions, because the rest of our library was defined in terms of `rebalance`. Thus, implementing `rebalance` is sufficient to implement every other function.

3 Analysis of Treaps

Our analysis closely mirrors our earlier analysis of quicksort and quickselect. The idea is to reason about which keys become ancestors of which other keys, using indicator random variables.

3.1 Setup and Notation

Let $S = \{k_0, k_1, \ldots, k_{n-1}\}$ be the set of keys, listed in sorted order by key. We refer to k_i as the *rank-i key*. Recall that priorities are assigned independently and uniformly at random, and (assuming distinctness) that the treap shape is uniquely determined by these priorities.

For indices $i, j \in \{0, \ldots, n-1\}$, define the indicator random variable

$$A_j^i = \begin{cases} 1 & \text{if } k_i \text{ is an ancestor of } k_j \text{ in the treap,} \\ 0 & \text{otherwise.} \end{cases}$$

These variables allow us to express structural properties of the treap as sums of simple random variables. The depth of the node containing key k_j is exactly the number of its ancestors:

$$\text{depth}(j) = \sum_{i=0}^{n-1} A_j^i.$$

Similarly, the size of the subtree rooted at k_j is the number of nodes for which k_j is an ancestor:

$$\text{size}(j) = \sum_{i=0}^{n-1} A_i^j.$$

These equalities are deterministic; the randomness comes from the treap structure induced by the random priorities.

3.2 Probability of an Ancestor Relationship

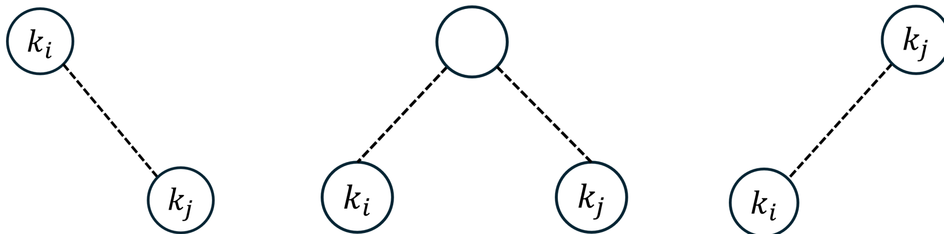
To compute quantities defined in terms of indicator variables, we need to compute the probability of their corresponding events, since their expected value is always precisely that. So, lets compute $\mathbb{E}[A_j^i]$, the probability that k_i is an ancestor of k_j.

Lemma: Ancestor Probability

For $i \neq j$,

$$\mathbb{E}[A_j^i] = \Pr[k_i \text{ is an ancestor of } k_j] = \frac{1}{|i-j|+1}.$$

Proof. Consider the keys with ranks between i and j, inclusive. There are exactly $|i-j|+1$ such keys. Consider the subtree consisting only of these keys. There are three possible subtree shapes that could occur relative to k_i and k_j.



i.e., either k_i is the root, neither is the root, or k_j is the root. For k_i to be the root, it must have the *highest priority*, in which case k_j lies in its subtree, so k_i is an ancestor of k_j. If k_j has the highest priority, by symmetry, k_j will be an ancestor of k_i. If any other key has the highest priority, then the interval is split and k_i will be in the left subtree and k_j will be in the right, so neither is an ancestor of the other.

Since priorities are chosen uniformly at random, each key in the interval is equally likely to have the highest priority, giving probability $1/(|i-j|+1)$ that k_i is an ancestor of k_j. □

3.3 Expected Depth of a Node

We now have all the ingredients to compute the expected depth of any particular key k_j.

Theorem: Expected Depth

For any key k_j,

$$\mathbb{E}[\text{depth}(j)] = O(\log n).$$

Proof. By linearity of expectation,

$$\mathbb{E}[\text{depth}(j)] = \sum_{i=0}^{n-1} \mathbb{E}[A_j^i] = \sum_{i=0}^{n-1} \frac{1}{|i-j|+1}.$$

To get rid of the absolute value, we split the sum at j:

$$\sum_{i=0}^{n-1} \frac{1}{|i-j|+1} = \sum_{i=0}^{j-1} \frac{1}{j-i+1} + \sum_{i=j}^{n-1} \frac{1}{i-j+1}.$$

If we expand the two summations, we get:

$$\mathbb{E}[\text{depth}(j)] = \left(\frac{1}{j+1} + \frac{1}{j} + \cdots + \frac{1}{2}\right) + \left(1 + \frac{1}{2} + \cdots + \frac{1}{n-j}\right)$$

Then we observe that these sums are both *harmonic numbers*, and we get

$$\mathbb{E}[\text{depth}(j)] = (H_{j+1} - 1) + H_{n-j},$$

Since $H_k = \Theta(\log k)$, both terms are $\Theta(\log n)$, and therefore

$$\mathbb{E}[\text{depth}(j)] = O(\log n).$$

□

So, we have shown that for each *fixed key* the expected depth of the corresponding node is $O(\log n)$ in expectation. What does that imply about the cost of fundamental operations, such as `split` and `join`?

These operations have cost proportional to the *height* of the treap, which is the maximum depth of any node in the tree. It is tempting to conclude that these operations also run in $O(\log n)$ expected time. **This is not a correct proof!**

The issue is a subtle but important distinction in probability. These quantities are *not equal*:

$$\mathbb{E}\left[\text{height of the treap}\right] = \mathbb{E}\left[\max_{0 \le j < n} \text{depth}(j)\right] \neq \max_{0 \le j < n} \mathbb{E}\left[\text{depth}(j)\right]$$

In other words, we don't have "linearity of expectation" for maximums, only for summations! While every individual key has logarithmic expected depth, this alone does *not* justify a logarithmic bound on the expected height of the treap. A separate and stronger analysis is required.

3.4 High probability bounds

To strengthen our claim about depth, instead of relying on the expected depth, we will prove a *high probability* bound on the depth of any node. For this, the proof is extremely similar to our analysis of Quickselect and Quicksort, where we conclude by invoking the Skittles Lemma.

Theorem: High Probability Depth

For any key k_j, $\text{depth}(j) = \Theta(\log n)$ with high probability.

Proof. Fix a key k_j and consider searching for it in the treap starting from the root. Each recursive step of the search moves from a node to one of its children, so the number of recursive steps taken is exactly the depth of k_j in the tree.

At any point during the search, the current subtree containing k_j consists of a contiguous interval of keys $S = \{k_\ell, \dots, k_r\}$ in sorted order. The root of this subtree is the key in S with the maximum priority. If k_j has the maximum priority, the search terminates.

Otherwise, since priorities are assigned uniformly at random, every key in S is equally likely to have the maximum priority. At least half of the keys in S lie in the middle 50% of this interval. If the maximum-priority key lies in this middle region, which happens with probability at least $1/2$, then the subtree containing k_j after the recursive step has size at most $3|S|/4$.

Thus, with constant probability, each step reduces the size of the current subtree by a constant factor. Applying the Skittles Lemma, after $\Theta(\log n)$ steps, with high probability the subtree size drops below 1, meaning the search has terminated. Therefore, $\text{depth}(j) = \Theta(\log n)$ w.h.p □

Corollary: Height of a Treap

The height of a treap, i.e., the depth of the deepest node, is $\Theta(\log n)$ with high probability.

Proof. For any fixed key k_j, by the definition of high probability, $\text{depth}(k_j) = \Theta(k \log n)$ with probability at least $1 - 1/n^{k+1}$, for any constant k. Applying a union bound over all n keys, the probability that *any* key has depth greater than $c \log n$ is at most $n \cdot (1/n^{k+1}) = 1/n^k$. Therefore, with high probability, every key has depth $\Theta(\log n)$, and hence the height of the treap is $\Theta(\log n)$.
□

4 Cost of split and join on treaps

The implementation of `split` for a generic BST is written entirely in terms of recursion and calls to `rebalance`. The implementation of `join` is then written in terms of `split`. Their cost therefore depends on two factors: the height of the tree and the cost of `rebalance`.

Generic balanced BSTs. For generic balanced binary search trees, `rebalance` may perform $O(\log n)$ work in the worst case. Since `split` is recursive for $O(\log n)$ levels, its total cost is $O(\log^2 n)$. Since `join` calls split and then calls `rebalance` once, its cost is also $O(\log^2 n)$. Perhaps this is too pessimistic. Can we do better than this?

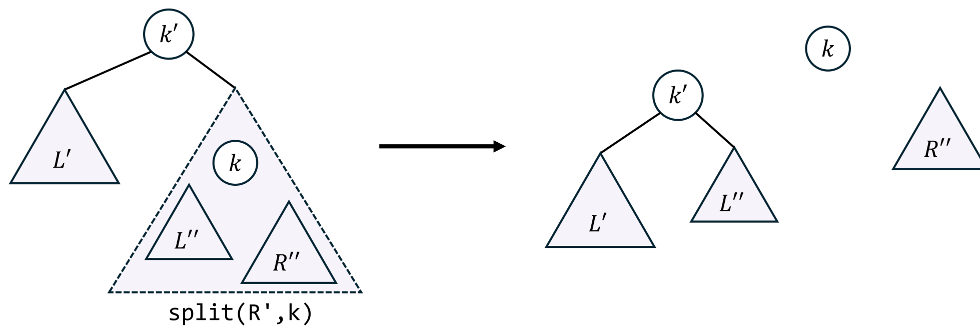
Treaps. For treaps, we don't need a different implementation of `split` and `join`; the same code works. Remarkably, however, the same code admits a stronger analysis when applied to treaps. In particular, we can show that on treaps, these functions are much more efficient.

Theorem: Cost of split and join in Treaps

For treaps, the `split` and `join` operations run in $O(\log n)$ time with high probability.

Proof. During `split`, recall how the recursive result is rebalanced with the rest of the tree:

```
L'', optV, R'' = split(R', k)
return (rebalance(makeNode(L', k', v', L'')), optV, R'')
```



Note that k' is the original root of the tree that has L', R' as its children, so it has the highest priority among all keys in those subtrees. Therefore, k' still has a higher priority than any element of L' or L'', and hence `makeNode(R',k',v,R)` yields a node that already satisfies the heap invariant. As a result, the call to `rebalance` performs only a constant amount of work.

Since the recursion depth is $O(\log n)$ w.h.p. and the algorithm performs just constant work per level, the total cost of `split` is $O(\log n)$ w.h.p. Since `join` just calls `split` once and then `rebalance` once, its total cost is also $O(\log n)$ w.h.p. □

This is really cool; this improvement comes from the treap invariants rather than from specialized algorithms: the code for `split` and `join` is identical across all balanced BST implementations, but it happens to be faster for treaps in particular because of their structure.