Treaps

In `prev`(lecture), we designed a generic interface for binary trees and showed how a wide range of sorted dictionary operations, such as searching, persistent updates, joining, and union, can be implemented using a small set of core primitives such as `expose` and `join_mid`, without depending on any particular balancing scheme.

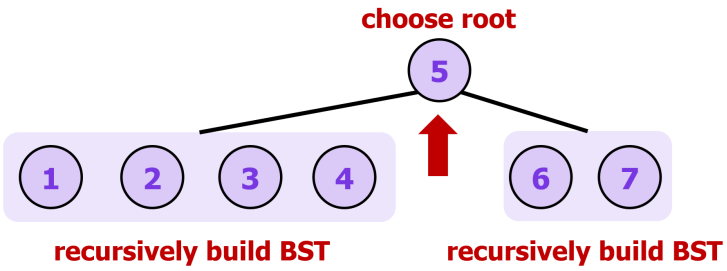
The key operation is `join_mid`, which is responsible for constructing new trees while maintaining their balancing invariants. Many classical balanced trees, such as AVL trees or red-black trees, enforce balance using carefully designed structural invariants. While these invariants guarantee logarithmic height, they can be subtle and tricky to maintain, especially in a purely functional and parallel setting.

In this lecture, 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

Rather than design another intricate and complex balancing scheme like AVL or Red-Black trees, our goal is to use *randomness* to obtain a much simpler algorithm that builds a binary search tree that is very unlikely to be unbalanced (i.e., balanced *with high probability*!).

Let's take inspiration from *Quicksort*. Recall that Quicksort chooses a random pivot, partitions the elements around it, and then recursively sorts the two sides. What if we used the same idea to build a binary search tree? We could choose a **random element as the root**, then recursively build the left and right subtrees from the elements smaller and larger than the root.



There are several ways to implement this idea. One approach is to randomly choose the root node whenever we construct a tree or join an existing pair of trees. This is known as a **randomized binary search tree**. However, making fresh random choices can sometimes cause unnecessary work. For example, if we split a tree into pieces and then put those pieces back together, we would ideally like to preserve the balancing decisions that were already made, rather than randomly choosing a new tree structure and potentially having to re-rebalance it again.

Treaps address this by assigning each element a random *priority*, putting the element with the highest priority at the root, then recursively applying the same rule to the two subtrees. This results in the same distribution of random tree shapes as a randomized BST. The advantage of this approach, however, is that the random choices are *stable*: once priorities have been assigned, they uniquely determine the shape of the tree, regardless of the order in which elements are inserted or removed or the ways in which the trees are split and re-joined.

The idea of placing the highest priority node at the root should remind you of a data structure you have already learned! The term "treap" is a portmanteau of the words "**tree**" and "**heap**":

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 two invariants:

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. 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. By randomly assigning priorities, we are effectively randomly choosing one of these possible tree shapes. Importantly though, not all shapes are equally likely!

Remark: Random priorities favor balanced trees

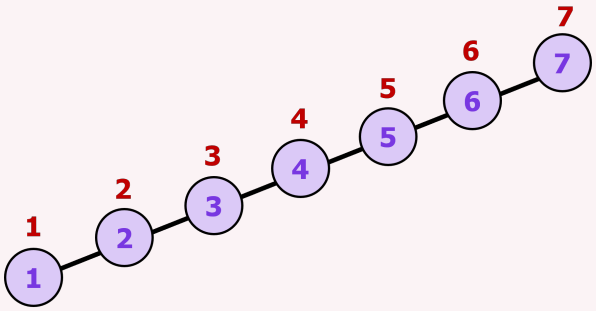
We said before that a given assignment of priorities induces a unique treap shape, which was crucial to our intuition for treaps. Interestingly, and importantly, the converse is not true. Given a tree shape, there may be many different priority permutations that produce it. This asymmetry is not just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

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

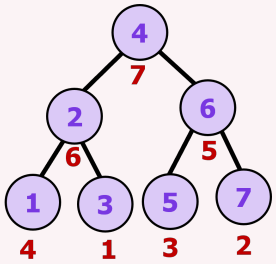
[1]The number of valid BSTs on n keys is the n^{th} *Catalan number* C_n, which grows like $C_n \sim 4^n/(\sqrt{\pi}n^{3/2})$.

uniformly from all binary search tree shapes produces trees whose average height is $\Theta(\sqrt{n})$, whereas treaps are strongly biased toward balanced ones.

To see this concretely, say we build a treap on the keys $\{1,\ldots,7\}$ and consider the odds that we obtain the decreasing chain $7 \to 6 \to \ldots \to 1$. This only occurs if priorities are drawn in 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. The middle key (4) must draw the highest priority which happens $1/7^{\text{th}}$ of the time, and then within the two subtrees, their middle elements (2 and 6) must draw the highest priorities which happens $1/3^{\text{rd}}$ of the time each, hence this happens with probability $1/63$, almost 100 times more likely than the first tree!



For example, in the tree above, notice that we get the same shape even if we switch priorities 6 and 5, or if we switch any of the leaf priories with each other. Thus, random priorities strongly favor balanced tree shapes over degenerate ones. As we will prove later, this results in treaps having logarithmic height with high probability.

2 Implementing Treaps

Recall that our generic `BinaryTree<Elem>` interface requires four operations: `size`, `expose`, `empty`, and `join_mid`. Interestingly, almost all of the implementation is identical to the unbalanced binary tree we implemented in the previous lecture. We can use exactly the same tree representation: each node stores its element, its left and right subtrees, and the size of the tree. The *only* part that will differ is the implementation of `join_mid`!

Algorithm: Treap Datatype

```
type Treap<Elem> =
    Empty
  | Node { left:  Treap,
           e :    Elem,
           size : int,
           right: Treap }

empty : Treap<Elem> = Empty
```

The implementations of `size` and `expose` are exactly the same as our unbalanced tree implementation. They don't touch any balancing logic, because that is entirely `join_mid`'s job!

```
fun size(T : Treap<Elem>) -> int:
  match T with:
    Empty: return 0
    Node(_,_,s,_): return s

fun expose(T : Treap<Elem>) -> optional<NodeView<Treap<Elem>>>:
  match T with:
    Empty: return NONE
    Node(L,e,_,R): return SOME(NodeView(L,e,R))
```

As usual, `expose` hides implementation details (the stored size field) and only returns the information needed for a generic algorithm to traverse the tree, which is the root element and its left and right subtrees. **All of the algorithms we wrote last lecture therefore work on our treaps with zero changes!** This leaves just one operation to implement: `join_mid`.

2.1 join_mid For Treaps

When implementing non-trivial tree data structures, it will be convenient to have a helper/factory function called `make_node`. This function constructs a node given its root element and two subtrees, and takes care of computing and storing invariant metadata, such as the `size` field.

Its purpose is twofold: it saves us from having to compute the size inside `join_mid`, which would make the code harder to understand, and it will serve as a convenient customization point when we discuss augmentation next lecture.

```
// Precondition: p(e) > priority(L) and p(e) > priority(R)
fun make_node(L: Treap<Elem>, e: Elem, R: Treap<Elem>) -> Treap<Elem>:
  return Node(L, e, 1 + size(L) + size(R), R)
```

Importantly, `make_node` does *not* perform any balancing. It simply constructs a node, assuming that the caller has already ensured that the resulting tree will satisfy the treap invariants. In other words, it is a *precondition* that the priority of e is greater than the priority of (the root elements of) L and R, since otherwise the resulting tree would not be a valid treap.

Priorities The core idea behind treaps is that every element has an associated random *priority*. There are several ways to implement this. For example, we could use random hashing to generate

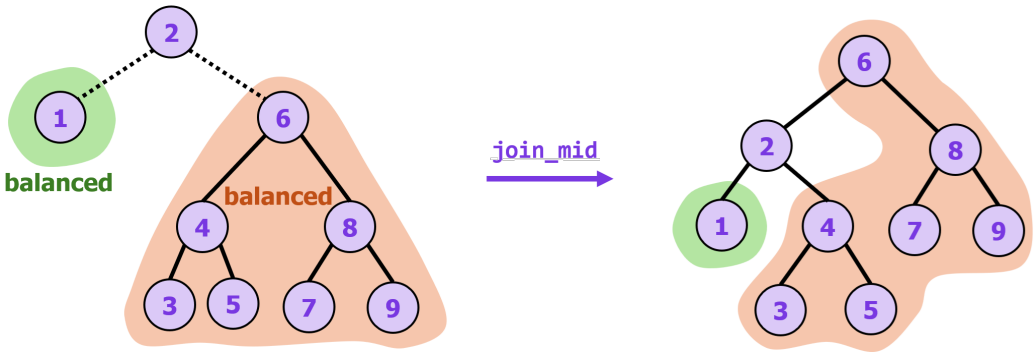
pseudorandom priorities from the elements. For now, we will simply assume that we have access to a function $p(e)$ that returns a stable, randomly assigned priority for each element.[2]

For convenience, we also define the priority of an entire tree to be the priority of its root element. An empty tree has priority $-\infty$, ensuring that any nonempty tree has higher priority.

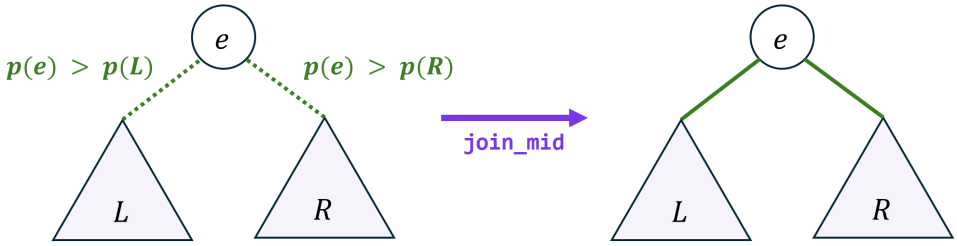
```
fun priority(T: Treap<Elem>) -> int:
  match T with:
    case Empty: return -∞
    case Node(_, e, _, _): return p(e)
```

2.1.1 Implementation

Recall that `join_mid(L,e,R)` takes two *already-balanced* (i.e., they satisfy the heap invariant) trees L and R and an element e, and constructs a new tree containing all of their elements in order, while maintaining the balancing invariants (i.e., satisfying the heap invariant).



`join_mid` has three cases, two of which are symmetric. The simplest case is when e has higher priority than the roots of both L and R. Then e can simply become the root of the new tree, with L and R as its children. The heap invariant is already satisfied, so no rebalancing is necessary.

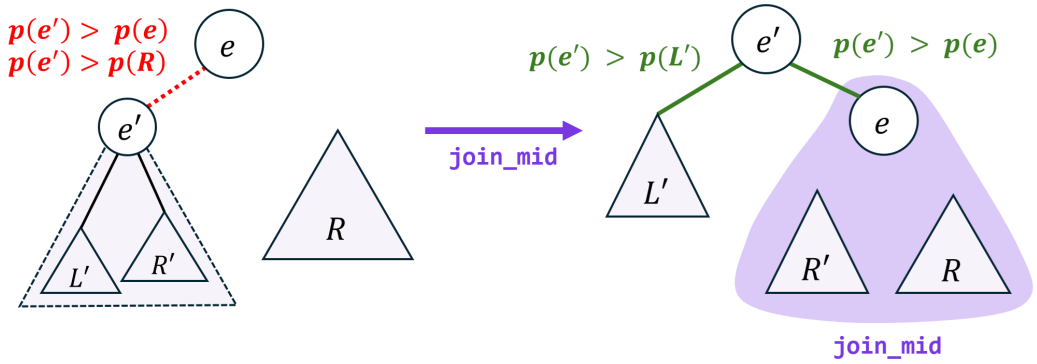


Otherwise, either L or R has a higher-priority root than e. Since L and R already satisfy the heap invariant, their roots have the highest priorities in their respective trees and hence the highest-priority element of the combined tree *must* be the root of L or R.

[2] For BSTs, this can be implemented as a pseudorandom function (e.g., a hash) of the key, since keys are unique. For arbitrary binary (non-search) trees, however, elements may repeat, so this approach would assign identical priorities to distinct nodes which would break the high-probability height bounds of treaps. Supporting arbitrary elements therefore requires a more sophisticated priority scheme, which we will not discuss.

Without loss of generality, suppose the root of L has the highest priority. Let e' be the root of L, with left and right subtrees L' and R'. Since e' has the highest priority, it must become the root of the result. Its left subtree remains L', because all those elements precede e and e'. You can think of this as conceptually similar to performing a *right rotation* in an AVL or Red-Black tree.

The right subtree must contain the elements of R', e, and R, in that order. However, e may not necessarily be the root, because it may have lower priority than R' or R. Instead, the algorithm *recursively* computes `join_mid(R',e,R)` and uses the result as the right subtree of e'.



Algorithm: Join Mid For Treaps

```
fun join_mid(L: Treap<Elem>, e: Elem, R: Treap<Elem>) -> Treap<Elem>:
  if p(e) > priority(L) and p(e) > priority(R):
    return make_node(L, e, R)
  else if priority(L) > priority(R):
    Node(L', e', _, R') = L
    return make_node(L', e', join_mid(R', e, R))
  else:
    Node(L', e', _, R') = R
    return make_node(join_mid(L, e, L'), e', R')
```

And that's it! In under ten lines of code, we have implemented the entire balancing scheme for treaps. Together with our implementations of `size`, `expose`, and `empty`, this is a complete implementation of our `BinaryTree<Elem>` interface.

All of the generic algorithms we developed in the previous lecture, including insertion, removal, splitting, joining, filtering, and union, now work on treaps without changing a single line of code! All of their balancing logic is encapsulated in this one surprisingly simple function.

3 Cost Analysis of Split and Join on Treaps

Recall that our generic implementation of `split` for a BST was written in terms of `join_mid`. The generic implementation of `join` was then written in terms of `split` and `join_mid`. Their costs therefore depend on two factors: the height of the tree and the cost of `join_mid`.

Unfortunately, it turns out that just knowing that a tree T has $h(T)$ height and that `join_mid` costs $O(h(T))$ in the worst case is not sufficient to prove that these algorithms also cost $O(h(T))$. Since `split` recurs for $O(h(T))$ levels and calls `join_mid` *at each level*, a straightforward analysis gives only an $O((h(T))^2)$ bound.

What can we do about this? One option is to implement specialized versions of `split` and `join` for each balanced tree type. While this can sometimes improve performance in practice, it is somewhat disappointing from a theoretical standpoint, because it defeats much of the purpose of designing generic algorithms in the first place.

Fortunately, we can do better! For standard balancing schemes such as AVL trees, Red-Black trees, and treaps, the generic algorithms actually do achieve $O(h(T))$ work, without requiring specialized implementations. For AVL and Red-Black trees, proving this requires a somewhat sophisticated amortized analysis. For treaps, however, the argument is extremely simple!

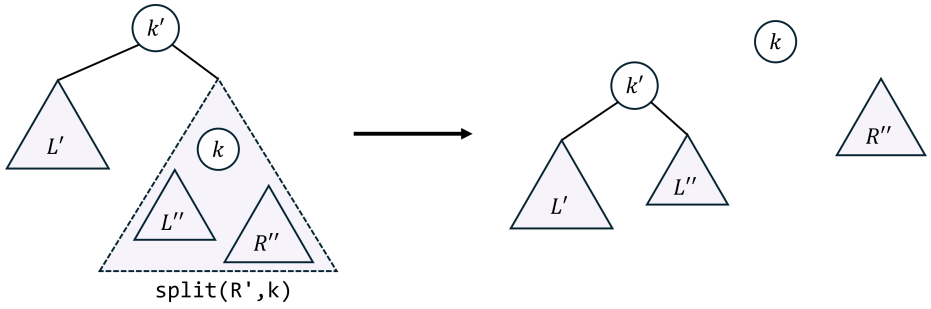
Theorem: Cost of split and join on Treaps

On a treap, the worst-case cost of the **generic** `split` and `join` algorithms are $\Theta(h(T))$.

Proof. Recall from our implementation of `split`, how the recursively joined tree is constructed with respect to the rest of the tree. Without loss of generality, consider the case where $k > k'$.

```
fun split(T: BST<K,V>, k: K) -> (BST, option<V>, BST):
  match expose(T) with:
    case NONE:
      return (empty, NONE, empty)
    case SOME(NodeView(L', (k',v'), R')):
      if k < k':
        L'', optV, R'' = split(L', k)
        return (L'', optV, join_mid(R'', (k',v'), R'))
      else if k > k':
        L'', optV, R'' = split(R', k)
        return (join_mid(L', (k',v'), L''), optV, R'')
      else:
        return (L', SOME(v'), R')
```

Observe that k' is the **original root** of the tree that has L', R' as its children, so by the heap invariant, it had the highest priority among all keys in those subtrees. Since L'' consists entirely of elements originally contained in R', every element in L'' has a lower priority than k'. Thus, k' still has higher priority than the roots of both L' and L''.



Therefore, after reassembling the new tree containing L', k', and L'', it must be the case that k' *still* has a higher priority than L' and L'', and hence `join_mid(L',(k',v'),L'')` is always a *best case* input for `join_mid`: a node that *already* satisfies the heap invariant. As a result, these calls to `join_mid` cost constant work.

The recursion depth of `split` is at most $h(T)$, since each call traverses one level down the tree. At each level, the algorithm performs constant work since every call to `join_mid` costs its best case, and hence the total cost of `split` is $\Theta(h(T))$. Since `join` just calls `split` once and then `join_mid` once (which itself costs $\Theta(h(T))$ in the worst case), its total cost is also $\Theta(h(T))$. □

This is really cool. The cost improvement arises naturally from the treap's invariants rather than needing specialized algorithms: the code for `split` and `join` is *generic* for all tree implementations, but it happens to be faster for certain implementations because of their invariants!

This is also true for AVL Trees and Red-Black trees, but the proofs are substantially more complicated. You can find them in the same research paper as the non-handwavy cost analysis of `union`, *Joinable Parallel Balanced Binary Trees* by **Blelloch** et al!

4 Height Analysis of Treaps

Our analysis closely mirrors our earlier analysis of Quicksort and Quickselect. When we analyzed the work of Quicksort, we used **indicator random variables** to count how many pairs of elements were ever compared. The analysis for treaps is extremely similar; the idea is to reason about which keys become ancestors of which other keys, using indicator random variables.

4.1 Setup and Indicator Random Variables

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.

Our analysis is going to be *almost identical* to the expected work analysis of Quicksort. This should not be surprising, since treaps were inspired by Quicksort in the first place! 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 are analogous to the indicator random variables we used for Quicksort, which counted whether key i was ever compared to key j. These indicator variables allow us to express structural properties of the treap as sums of them.

The quantity we are interested in, the depth of k_j, is equal to *how many ancestors it has*:

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

For simplicity, we will consider a node to be an ancestor of itself.

4.2 Probability of an Ancestor Relationship

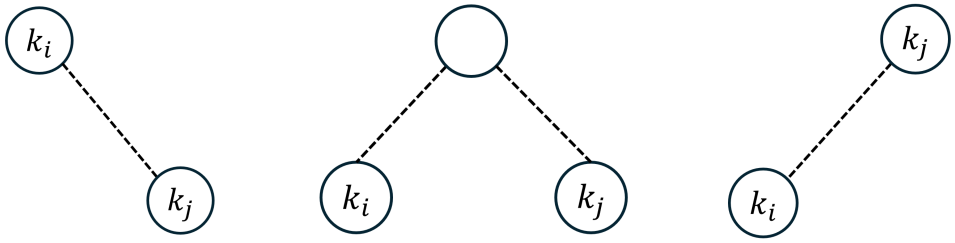
To compute quantities defined in terms of indicator variables, we need to compute the probability of their corresponding events, since their expected value is always precisely that. So, lets compute $\mathbb{E}[A^i_j]$, the probability that k_i is an ancestor of k_j.

Lemma: Ancestor Probability

For $i \neq j$,

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

Proof. Consider the keys with ranks between i and j, inclusive. There are exactly $|i-j|+1$ such keys. Consider the interval 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. □

4.3 Expected Depth of a Node

We now have all the ingredients to compute the expected depth of any particular key k_j. Once again, the math is extremely similar to the expected work analysis of Quicksort.

Theorem: Expected Depth

For any key k_j,

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

Proof. By linearity of expectation,

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

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

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

Using the fact that $\ln k < H_k < \ln k + 1$, the sum of the terms is $O(\log n)$, and at least one of them is $H_{n/2}$ which is $\Omega(\log n)$ and therefore $\mathbb{E}[\text{depth}(j)] = \Theta(\log n)$. □

So, we have shown that for each *fixed key* the expected depth of the corresponding node is $\Theta(\log n)$ in expectation. What does that imply about the cost of binary search tree algorithms, such as `split` and `join`, on a treap?

As we showed in the previous section, the cost of these operations on treaps is bounded by the height of the tree, which is the maximum depth of any node. 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.

4.4 High probability bounds

To prove a height bound on treaps, we need to strengthen our claim about depths. Instead of relying on the *expected depth of an arbitrary node*, we need to prove a **high probability bound** on the *depth of every node*. This gives a high probability bound on the height of a treap, which informs the cost bounds of all of our algorithms.

We could prove this from scratch using an upper bound on the expected subtree size and then applying Markov's inequality or the Skittles Lemma, but there's a much simpler approach: we can re-use all of the maths that we already did for Quicksort!

Theorem: Quicksort/Treap Equivalence

The distribution of tree shapes for the randomized Quicksort recursion tree, and for treaps with random distinct priorities is identical (assume $|S| > 0$).

Proof. By (structural) induction:

- **Base case**: only one possible shape (single node).
- **Induction**: Every element of S can be picked with equal probability for both, so distribution of root is the same. Since the priorities are independently randomized, the remaining keys are still in uniformly random priority order, so the same argument applies recursively. By induction, the distribution of each subtree is the same.

By induction on trees, for any input S, the distribution of tree shapes is identical. □

Since we proved earlier that the recursion depth of Quicksort is $O(\log n)$ w.h.p., we immediately obtain the corollary that treaps have $O(\log n)$ height w.h.p.

Corollary: Height of a Treap

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

Finally, since we proved that `split` and `join` cost $O(h(T))$ on treaps, we can conclude with tight high-probability bounds on their cost.

Corollary: High-Probability Cost of split and join on Treaps

When invoked on a treap, the worst-case cost of the **generic** `split` and `join` algorithms is $O(\log n)$ with high probability.

For `join(A,B)`, n refers to the size of the *resulting tree*, i.e., $n = |A| + |B|$.