Parallel And Sequential Data Structures and Algorithms

Binary Search Trees II: Treaps

Learning Objectives

- Analyze the **union** algorithm on balanced binary search trees
- Learn about an implementation of balanced binary search trees based on randomization: **Treaps**
- Analyze Treaps and prove that they yield good cost bounds in expectation and with high probability

Binary Search Trees

Review: A Generic BST Data Type

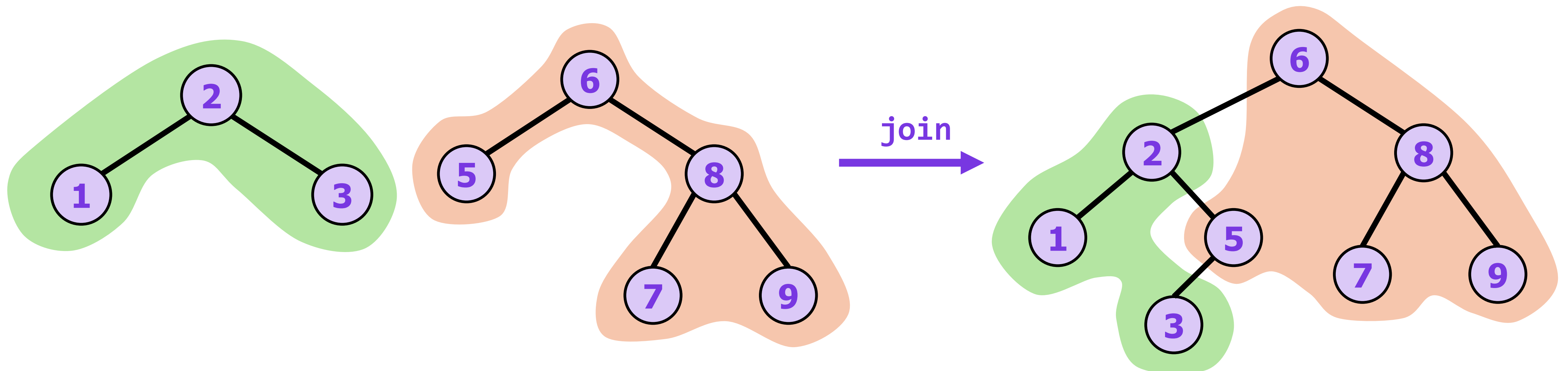
```
type BST<K : Ordered, V> =
    Empty
  | Node { left: BST,
           key: K,
           value: V,
           size: int,
           right: BST }
```

- A generic BST is parameterized by the type of the **keys** and **values** it stores at the nodes
- The key type `K` must be **ordered**, meaning we can compare keys, i.e., $k_1 < k_2$, $k_1 = k_2$, $k_1 > k_2$.

- The `BST` type is **recursive**. A `BST` is either empty, or it's a Node which itself has two `BST`s as children/subtrees.
- The implementation in C++ versus SML (or generally, any OOP language versus any functional language), looks quite different.

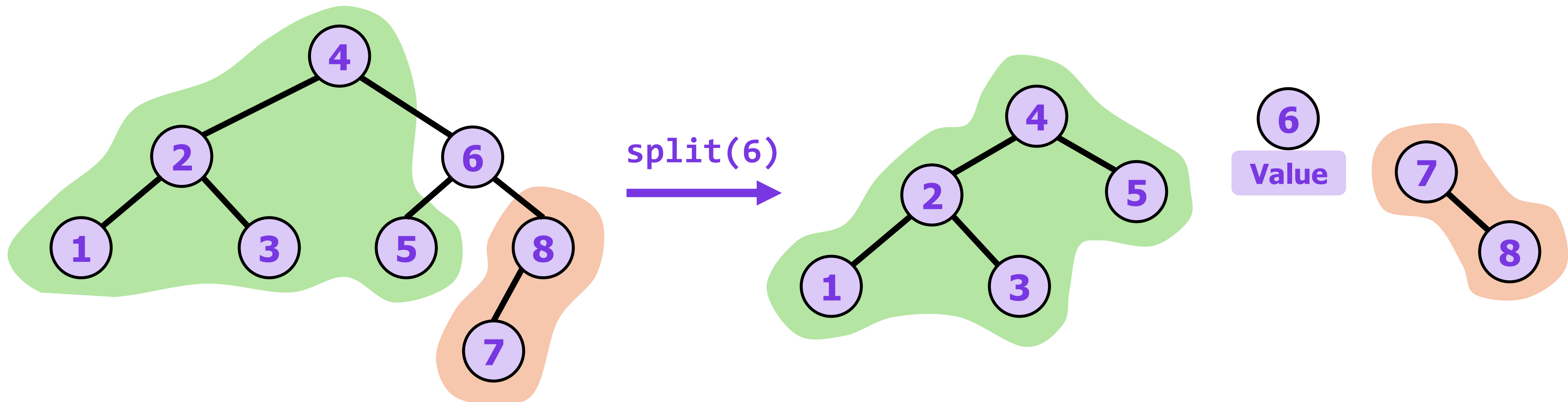
Review: Structural Operations on BSTs

- BSTs also support some interesting operations we have not seen yet
  - **`join(L, R)`** `-> BST<K,V>:`
    join the two trees `L` and `R`, where every key in `L` is less than every key in `R`
  - **`split(T, k)`** `-> (BST<K,V>, option<V>, BST<K,V>):`
    splits T into two trees, one containing the keys less than k, one containing the keys greater than k, and if k is present, its value.



Review: Structural Operations on BSTs

- BSTs also support some interesting operations we have not seen yet
  - **`join(L, R)`** `-> BST<K,V>:`
    join the two trees `L` and `R`, where every key in `L` is less than every key in `R`
  - **`split(T, k)`** `-> (BST<K,V>, option<V>, BST<K,V>):`
    splits T into two trees, one containing the keys less than k, one containing the keys greater than k, and if k is present, its value.

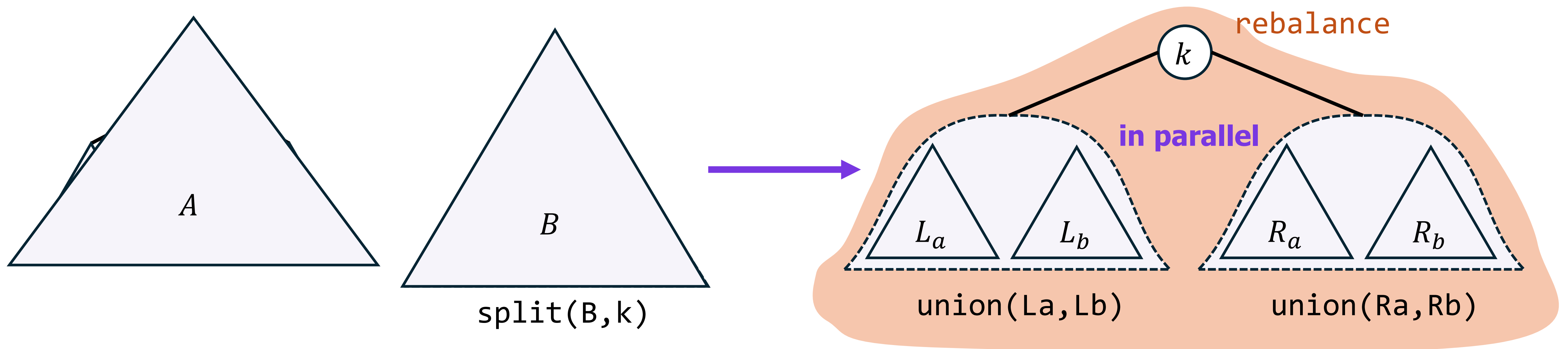


BST Algorithms

Union

- **`union(A, B)`** `-> BST<K,V>:`
  return a BST whose keys are the union of the keys of `A` and `B`

Caveat: Which **value** do we keep if a key is in both? For now, just take value from A.



Union

- **union(A, B) ->** BST<K,V>:
  return a BST whose keys are the union of the keys of A and B

```
fun union(A : BST<K,V>, B : BST<K,V>) -> BST:
  match (A, B) with:
    case (Empty, _): return B
    case (_, Empty): return A
    case (Node(La, k, v, _, Ra), _):
      Lb, _, Rb = split(B, k)
      L, R = parallel (union(La, Lb), union(Ra, Rb))
      return rebalance(makeNode(L,k,v,R))
```

Cost of Union

Theorem (Cost of Union): Given two balanced binary search trees A and B of size m and n respectively, where WLOG $m \leq n$, union costs

$$O\left(m + m\log\left(\frac{n}{m}\right)\right)$$

work and $O(\log m \log n)$ span.

Assumption: We are going to assume that A is **perfectly balanced**, i.e., every internal node has two children, and all the leaves have the same depth. Without this assumption, the proof is a 40-page research paper (read *Joinable Parallel Balanced Binary Trees* by **Blelloch** et al).

Cost of Union

Theorem (Cost of Union): Given two balanced binary search trees A and B of size m and n respectively, where WLOG $m \leq n$, union costs

$$O\left(m + m \log\left(\frac{n}{m}\right)\right)$$

work and $O(\log m \log n)$ span.

Proof: Let $W(m, n)$ *be the work to union trees of sizes* m *and* n

$$W(m, n) = \begin{cases} 1 + \log n, & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log n, & \text{otherwise} \end{cases}$$

Cost of Union: Brick Method

$$W(m,n) = W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log n$$

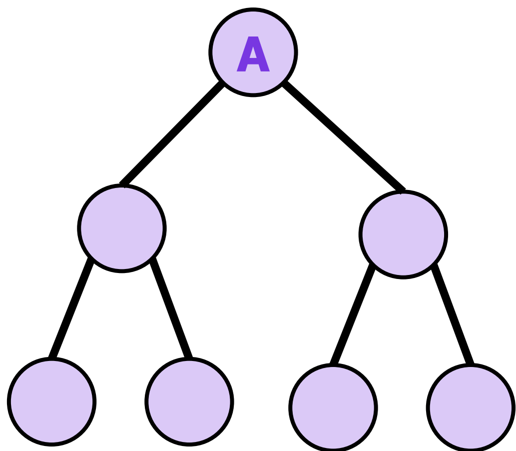
- **Parent work**: $\log n$
- **Child work**: $\log |L_b| + \log|R_b|$ where $|L_b| + |R_b| = n$
- **Child** $\geq$ **Parent** so it's not root dominated. But whether this is leaf dominated or balanced depends on $|L_b|$ and $|R_b|$…

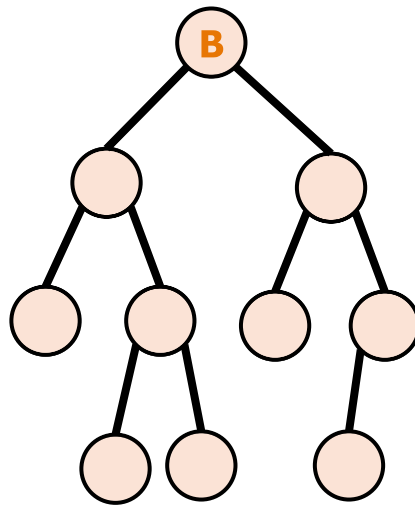
Key Observation: In the **worst case**, $|L_b| = |R_b| = n/2$ because $\log$ is a **concave** function. Formally, Jensen's inequality says:

$$\log |L_b| + \log|R_b| \leq 2 \cdot \log\left(\frac{n}{2}\right) = 2 \cdot \log n \ - 2$$

Leaf Dominated!

$$W(m,n) = \begin{cases} 1 + \log n, & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log n, & \text{otherwise} \end{cases}$$





Leaf Dominated!

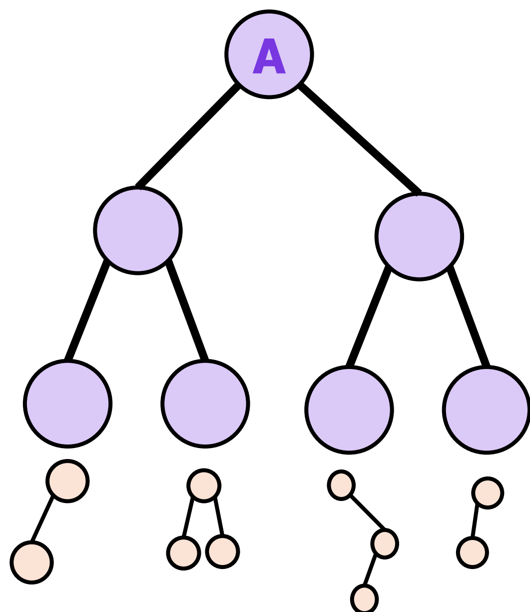
$$W(m,n) = \begin{cases} 1 + \log n, & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log n, & \text{otherwise} \end{cases}$$

A

B

Leaf Dominated!

$$W(m,n) = \begin{cases} 1 + \log n, & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log n, & \text{otherwise} \end{cases}$$



- The leaves occur when $m = 1$. Under our assumption that A is perfectly balanced, there are $m/2$ leaves
- The leaves cost $W(1, b_i)$ where b_i is the size of B at leaf i,

$$\text{Total leaf cost} = \sum (1 + \log b_i)$$

Total Cost of the Leaves

- The leaves cost $W(1, b_i)$ where b_i is the size of B at leaf i,

$$\text{Total leaf cost} = \sum (1 + \log b_i)$$

- The leaf sizes b_i are the result of splitting B, so $\sum b_i \leq n$
- Again, since $\log$ is a concave function, the maximum is when all the b_i values are equal at the $m/2$ leaves, so $b_i = 2n/m$

$$\text{Total leaf cost} = \sum \left(1 + \log\left(\frac{2n}{m}\right)\right) = O\left(m + m \log\left(\frac{n}{m}\right)\right)$$

Span

- There are $O(\log m)$ levels of recursion and each levels pays $O(\log n)$ local work, so the span is at most $O(\log m \log n)$

There exists a better algorithm with the same work but with just $O(\log n)$ span. So, we will say that the span cost of union is $O(\log n)$ but we won't cover that algorithm.

Treaps

Treaps

- Inspired by **Quicksort**. Use randomness to balance a tree instead of relying on a deterministic balancing scheme

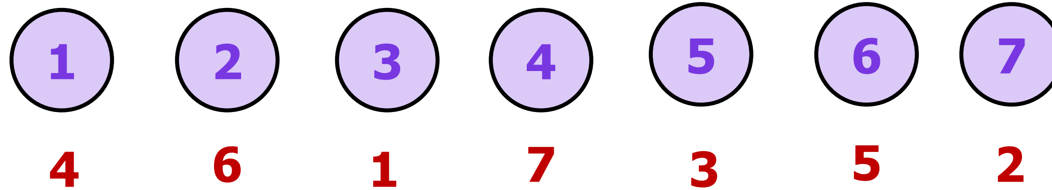
Definition (Treap): A **treap** is a binary search tree where each key k has an associated priority $p(k)$. The tree must satisfy:

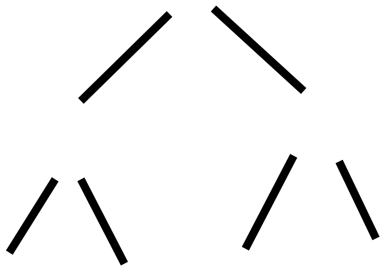
1. **BST Invariant**: For every node with key k, all keys in the left subtree are less than k. All keys in the right subtree are greater than k.
2. **Heap Invariant**: For every node with key k and priority $p(k)$, the priority $p(k)$ is greater than the priorities of its children

Treap = Tree + Heap

Priorities ⇒ Tree Shape

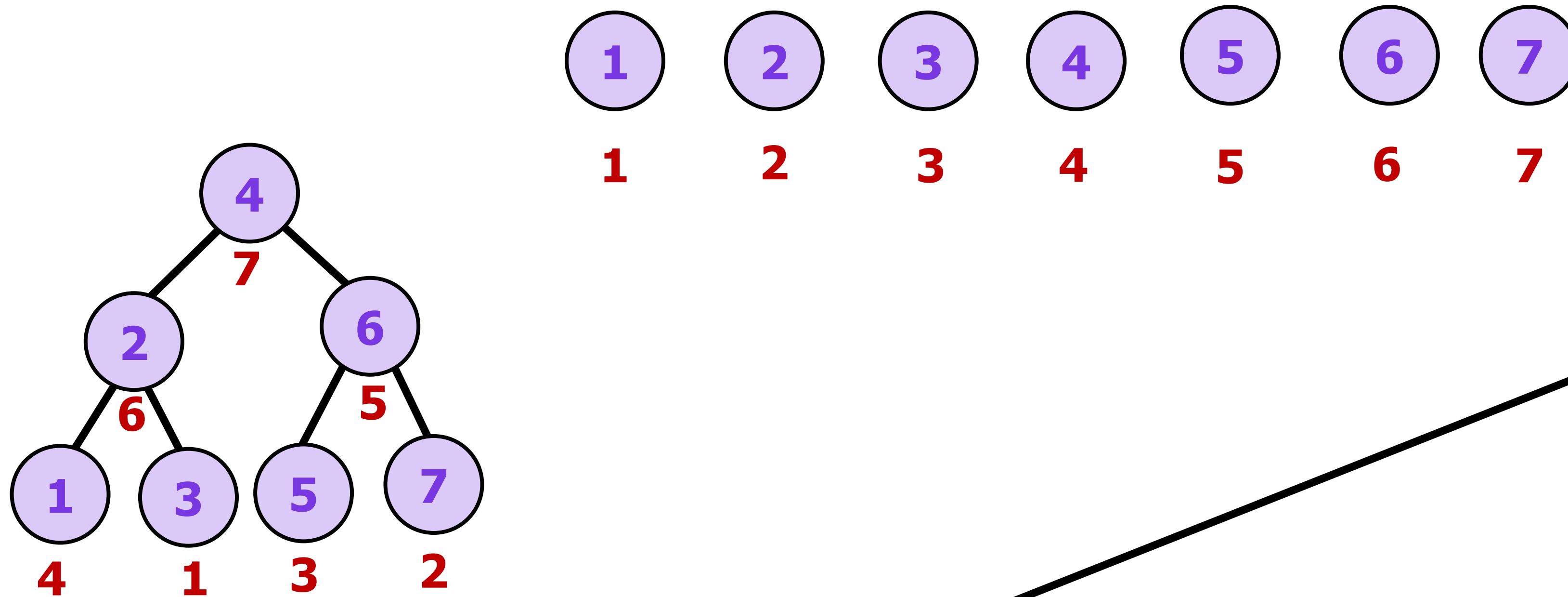
Claim (Priorities): Given a set of distinct keys with distinct priorities, there is a unique treap that satisfies the BST and heap invariants.





Priorities ⇒ Tree Shape

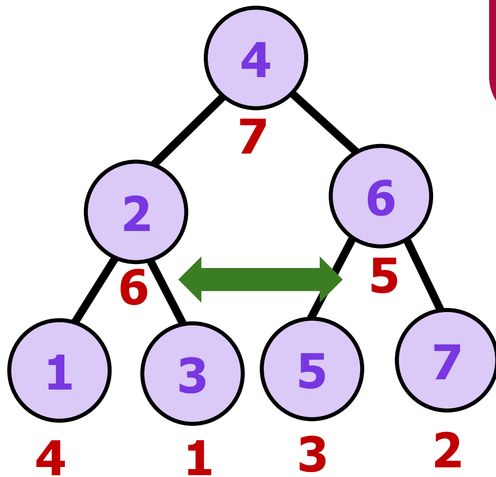
Claim (Priorities): Given a set of distinct keys with distinct priorities, there is a unique treap that satisfies the BST and heap invariants.

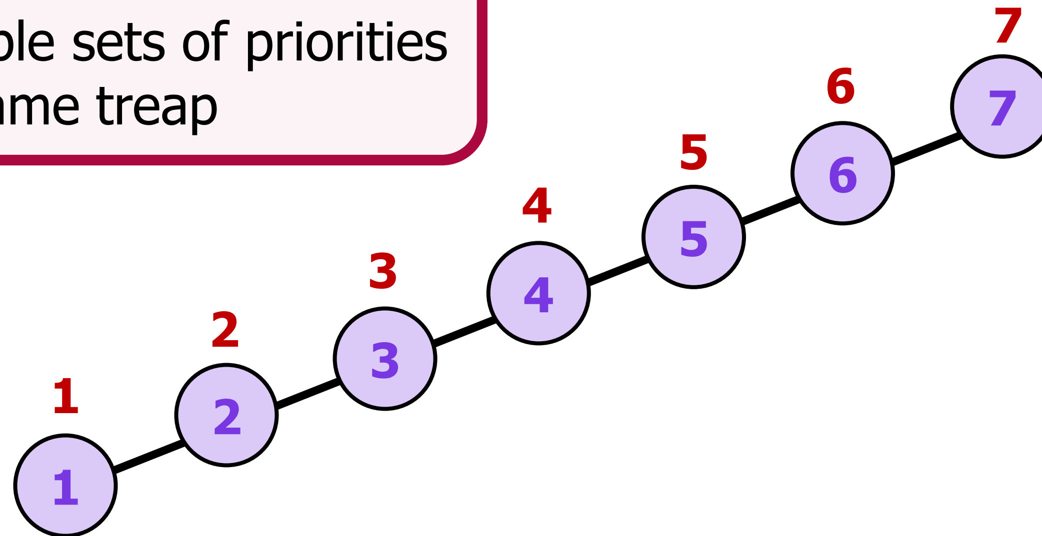


Priorities ⇒ Tree Shape

Claim (Priorities): Given a set of distinct keys with distinct priorities, there is a unique treap that satisfies the BST and heap invariants.

Importantly, the converse is not true. Multiple sets of priorities can give the same treap

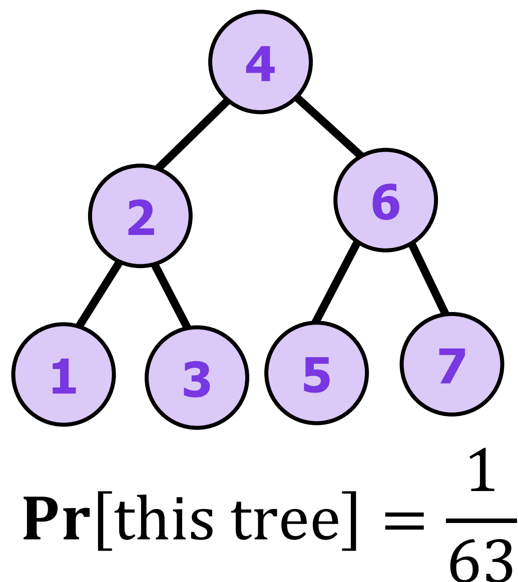


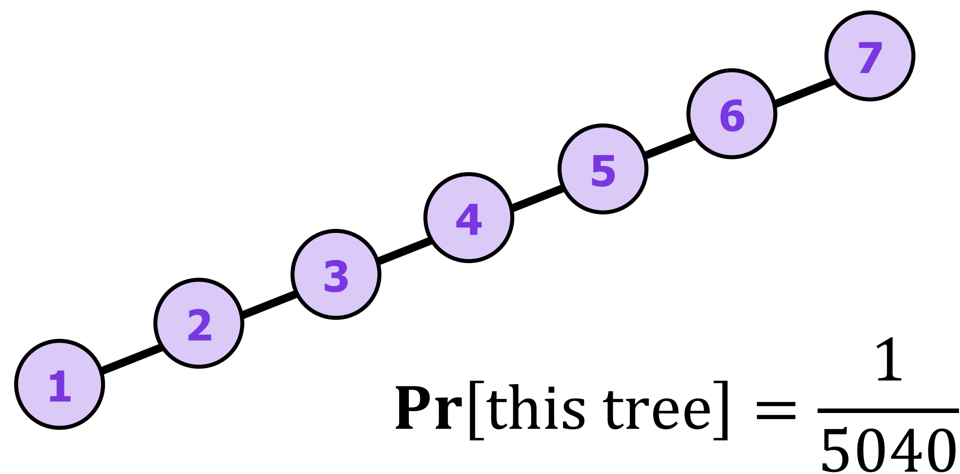


How to Choose Priorities?

In a treap, priorities are chosen **randomly**

- By choosing randomly, imbalanced trees are extremely unlikely
- In fact, we will prove that the resulting tree is balanced w.h.p.





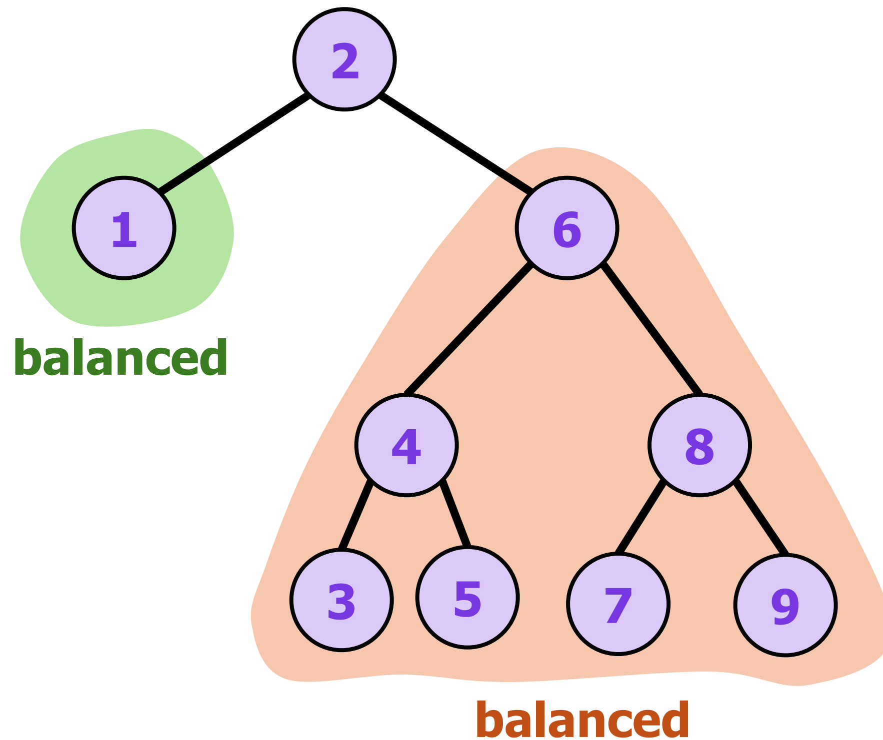
Implementing Rebalance For Treaps

Review: The `rebalance` primitive

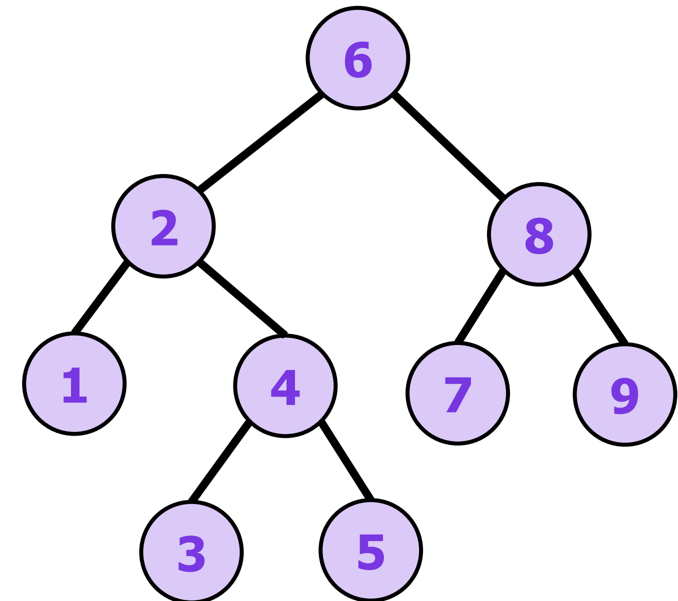
- To implement our BST algorithms so that they keep the tree balance, we will assume we have a function **rebalance**
- This lecture, we will implement rebalance for a particular class of balanced BSTs called **treaps**

Definition (Rebalance): Rebalance takes a BST whose left and right children are balanced (but the root might not be) and returns a new BST that is balanced (now including at the root node).

Review: Rebalance example



rebalance



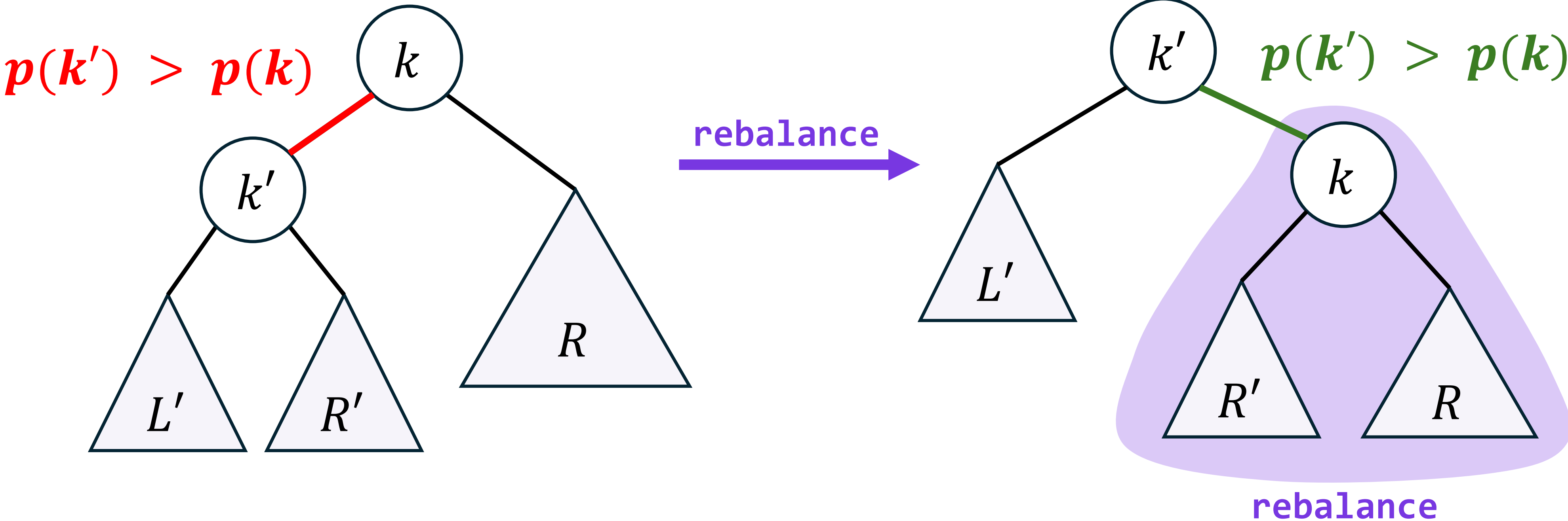
rebalance can be implemented using AVL-Tree rotations, Red-Black Tree rotations (with a color field), or ~~next lecture~~ right now, treaps!

Priorities

- Priorities can be implemented in many ways. One way is to use hashing to generate pseudorandom numbers
- We won't go into detail on this and just assume we have access to random priorities for each key
- We define a function that returns the priority of a tree from the key of the root, and assigns $-\infty$ as the priority of an empty tree

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

Treap Rebalance



Treap Rebalance (Code)

```
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'))
```

Comparison to AVL Trees (15-122)

```
tree* rebalance_right(tree* T) {
  if (height(T->right) - height(T->left) == 2) {
    if (height(T->right->right) > height(T->right->left)) {
      T = rotate_left(T);
    } else {
      T->right = rotate_right(T->right);
      T = rotate_left(T);
    }
  } else {
    fix_height(T);
  }
  return T;
}
```

```
tree* rotate_left(tree* T) {
  tree* R = T->right;
  T->right = T->right->left;
  R->left = T;
  fix_height(T);
  fix_height(R);
  return R;
}
```

```
void fix_height(tree* T) {
  int hl = height(T->left);
  int hr = height(T->right);
  T->height = (hl > hr ? hl+1 : hr+1);
  return;
}
```

This is only half the code. You need symmetric `rebalance_left` and `rotate_right` as well

Depth Analysis

Treap Node Depth

Theorem (Node Depth): Given a set of n keys and a particular key k, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the **expected depth** of node k is $O(\log n)$.

- *Let* $S = \{k_0, k_1, \dots, k_{n-1}\}$ *be the set of keys in sorted order*
- *Define* ***indicator random variables****:*

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

Treap Node Depth

- *Let* $S = \{k_0, k_1, \ldots, k_{n-1}\}$ *be the set of keys in sorted order*
- *Define* ***indicator random variables****:*

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

- *Let* $\text{depth}(j)$ *denote the depth of key* k_j

$$\text{depth}(j) = \sum_{0 \le i < n} A_j^i$$

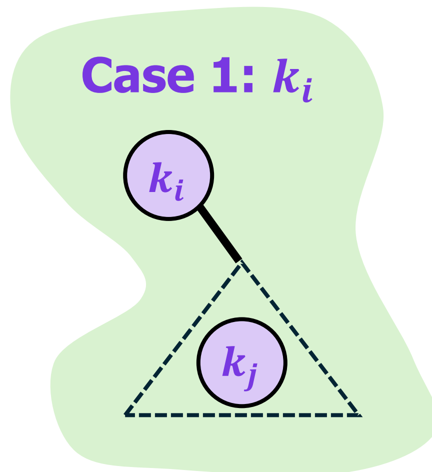
Treap Node Depth

Lemma (Ancestor Probability): For $i \neq j$,

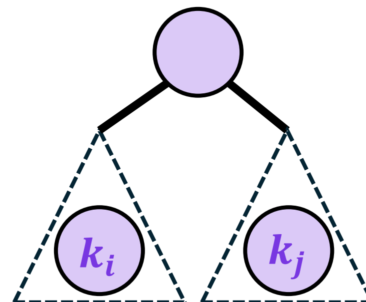
$$\mathbf{E}\left[A_j^i\right] = \mathbf{Pr}\left[k_i \text{ is an ancestor of } k_j\right] = \frac{1}{|i-j|+1}$$

- *Consider the keys between k_i and k_j inclusive*
- *Case: Which of these $|i-j|+1$ keys has the highest priority?*

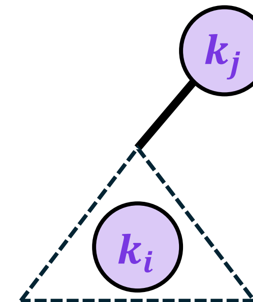
Case 1: k_i



Case 2: Some key between



Case 3: k_j



Expected Depth Calculation

$$\mathrm{depth}(j) = \sum_{0 \le i < n} A_j^i$$

- *Using **linearity of expectation***

$$\mathbf{E}[\mathrm{depth}(j)] = \sum_{0 \le i < n} \mathbf{E}[A_j^i]$$

$$= \sum_{0 \le i < n} \frac{1}{|i - j| + 1}$$

Expected Depth Calculation

$$= \sum_{0 \le i < n} \frac{1}{|i - j| + 1}$$

$$= \sum_{0 \le i < j} \frac{1}{j - i + 1} + \sum_{j \le i < n} \frac{1}{i - j + 1}$$

$$= \left(\frac{1}{j + 1} + \frac{1}{j} + \cdots + \frac{1}{2} \right) + \left(1 + \frac{1}{2} + \cdots + \frac{1}{n - j} \right)$$

$$= (H_{j+1} - 1) + H_{n-j} \quad = \Theta(\log n)$$

Depth ⇒ Height?

Theorem (Node Depth): Given a set of n keys and a particular key k, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the **expected depth** of node k is $O(\log n)$.

- Given this, what can we say about the expected height?

TRUE or FALSE: The theorem above implies that the height of a treap is $O(\log n)$ in expectation

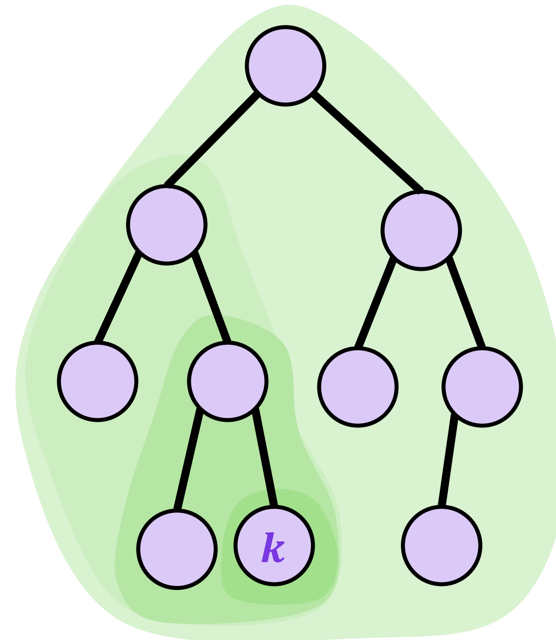
$$\mathbf{E}[\text{height}] = \mathbf{E}\left[\max_{0 \le j < n} \text{depth}(j)\right] \neq \max_{0 \le j < n} \mathbf{E}[\text{depth}(j)]$$

High Probability Bounds

High Probability Depth Bound

Theorem (Node Depth w.h.p.): Given a set of n keys and a particular key k, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the depth of node k is $O(\log n)$ **w.h.p.**

- *We can re-use the* ***Skittles Lemma*** *that we used to analyze Quickselect and Quicksort!*
- *Let* X_r *be the number of nodes in the subtree when searching for the key* k *at depth* r. $X_0 = n$



$X_0 = 10$

$X_1 = 5$

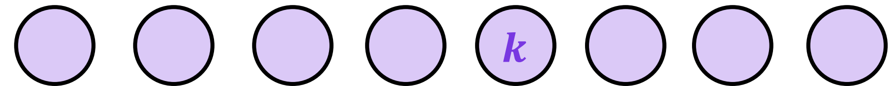
$X_2 = 3$

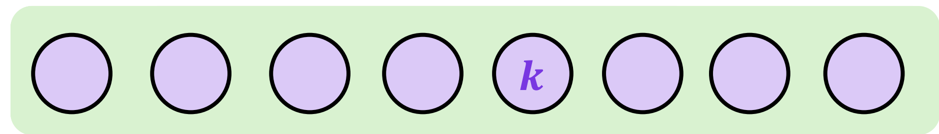
$X_3 = 1$

High Probability Depth Bound

Theorem (Node Depth w.h.p.): Given a set of n keys and a particular key k, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the depth of node k is $O(\log n)$ **w.h.p.**

- *Suppose* $X_r = n'$ *at some depth* r
- *Each of the* n' *keys is equally likely to be the highest priority (root)*
- *The size of the larger side is uniformly in the range* $[n'/2, n'-1]$
- *The average such size is* $\leq \frac{3}{4}n'$



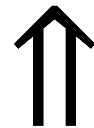


Recall: This is the same math we did for Quickselect!

High Probability Depth Bound

Theorem (Node Depth w.h.p.): Given a set of n keys and a particular key k, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the depth of node k is $O(\log n)$ **w.h.p.**

- *We therefore have* $\mathbf{E}[X_{r+1}|X_r] \leq \frac{3}{4} X_r \Rightarrow \mathbf{E}[X_{r+1}] \leq \frac{3}{4} \mathbf{E}[X_r]$

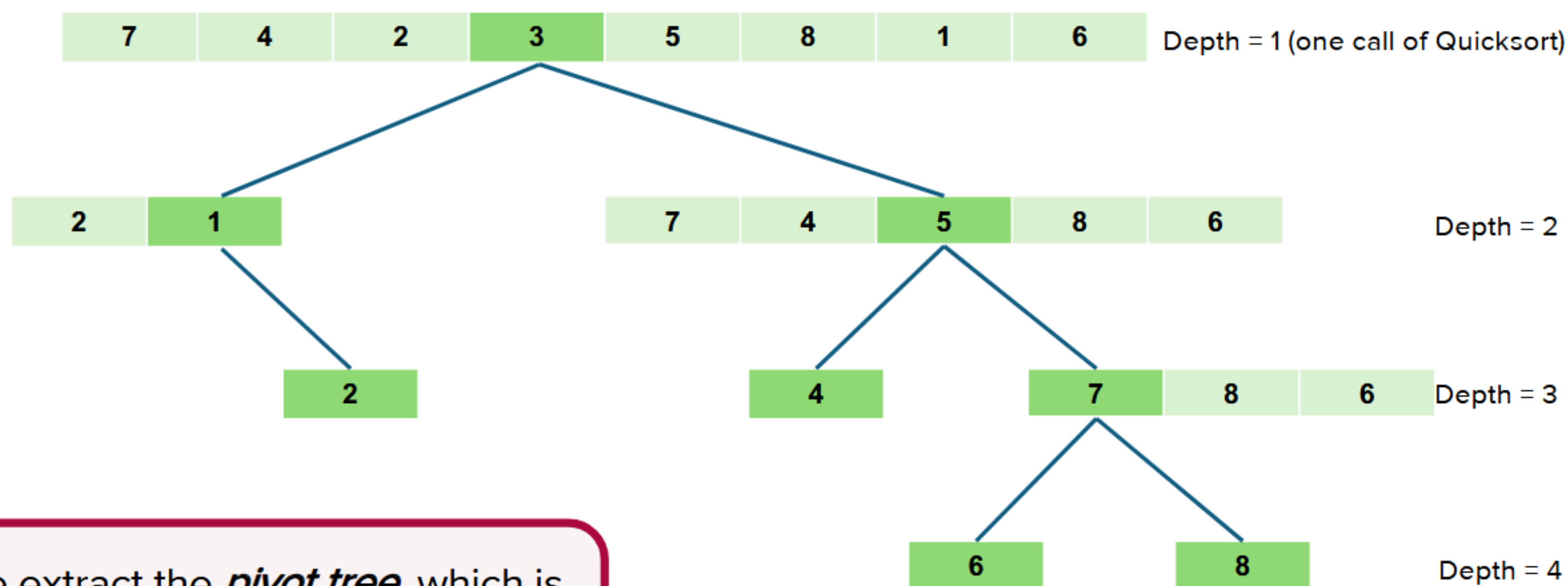


Lemma (Skittles Lemma): Let $X_0 = n$ and let X_{r+1} be computed by a random process from X_r that guarantees that:

- $0 \leq X_{r+1} \leq X_r$,
- $\mathbb{E}[X_{r+1}] \leq \alpha \mathbb{E}[X_r]$ where $0 < \alpha < 1$ is a constant.

Let $R(n)$ be a random variable which is the first round r when $X_r < 1$. Then $R(n)$ is $O(\log n)$ w.h.p.

Same Math as QuickSelect/Sort?



From this we extract the ***pivot tree***, which is a binary tree in which every element of the input sequence occurs as a node.

A **pivot tree** is the same thing as a **treap**!

High Probability Height

- We proved that each individual node has height $O(\log n)$ w.h.p.

Theorem (Node Depth w.h.p.): Given a set of n keys and a particular key k, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the depth of node k is $O(\log n)$ **w.h.p.**

- The height of the tree is the maximum depth over all nodes
- Therefore, by a **union bound** over all n nodes

Theorem (Treap Height w.h.p.): Given a set of n keys, a treap built on those n keys with priorities chosen uniformly and independently at random has height $O(\log n)$ **w.h.p.**

Cost of Split and Join

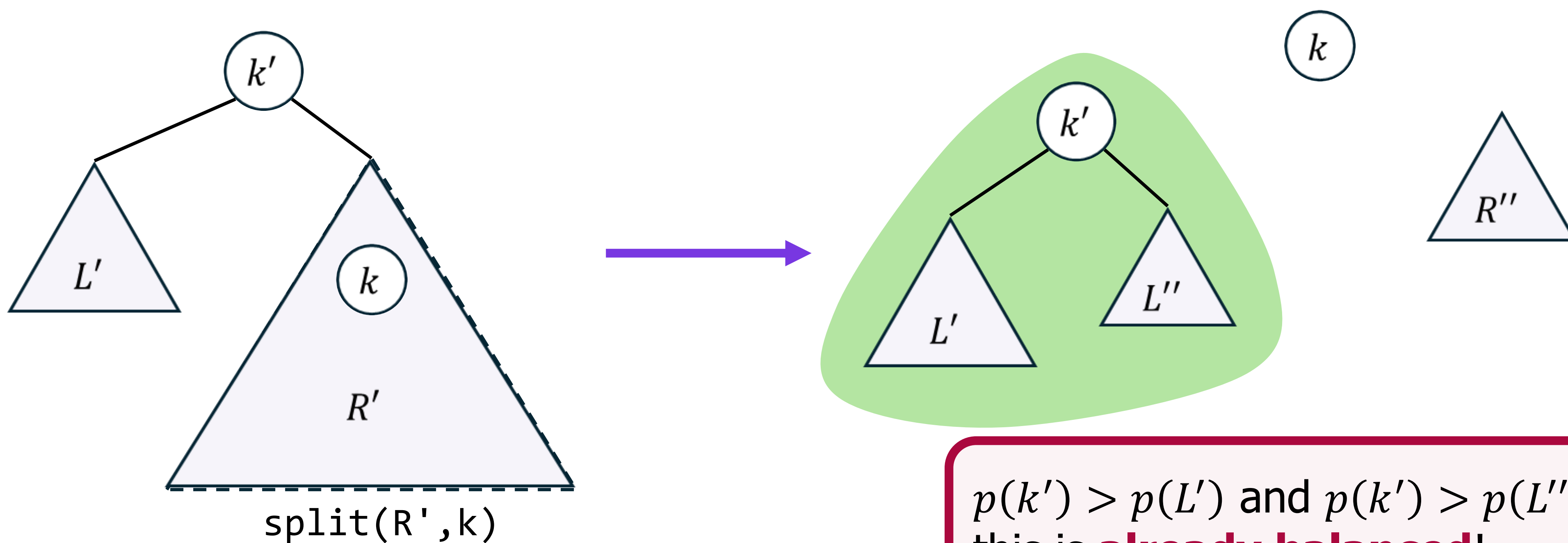
Split and Join

```
fun split(T : BST<K,V>, k : K) -> (BST, option<V>, BST):
  match T with:
    case Empty: return (Empty, NONE, Empty)
    case Node(L’,k’,v’,_,R’):
      if k < k’:
        L’’, optV, R’’ = split(L’, k)
        return (L’’, optV, rebalance(makeNode(R’’, k’, v’, R’)))
      else if k > k’:
        L’’, optV, R’’ = split(R’, k)
        return (rebalance(makeNode(L’, k’, v’, L’’)), optV, R’’)
      else: return (L’, SOME(v’), R’)
```

- We call rebalance at every level of the tree, so this costs $O(\log \mathrm{n})$ per level for $O(\log n)$ levels $\Rightarrow$ costs $O(\log^2 n)$
- But we have been saying split should cost $O(\log n)$!

Splitting a Treap

Theorem (Split Cost): Splitting a treap of n keys costs $O(\log n)$ w.h.p.



Joining A Treap

Corollary (Join Cost): `join(A,B)` on treaps costs $O\left(\log\left(|A| + |B|\right)\right)$ w.h.p.

```
fun join(L: BST<K,V>, R: BST<K,V>) -> BST:
  match first(R) with:
    case NONE: return L
    case SOME((k,v)):
      _, _, R' = split(R, k)
      return rebalance(makeNode(L, k, v, R'))
```

`join` *just calls split and rebalance, costing* $O(\log |B|)$ *and* $O\left(\log\left(|A| + |B|\right)\right)$

Summary

- **Union** on balanced binary search trees costs $O\left(m + m\log\left(\frac{n}{m}\right)\right)$
- **Treaps** are a **randomized** balanced binary search tree based on assigning random priorities and enforcing that nodes are heap ordered by priority (higher priority ⇒ higher in the tree)
- Treaps have $O(\log n)$ height w.h.p.
- **split** and **join** cost $O(\log n)$ w.h.p. on treaps