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: An Interface for Binary Trees

A type **Tree** is a **BinaryTree**<Elem> if:

```
size(T : Tree) -> int
expose(T : Tree) -> optional<NodeView<Tree>>

empty : Tree
join_mid(L : Tree, e : Elem, R : Tree) -> Tree
```

expose asks to see the root element and its left and right subtrees.

*Irrelevant internal data such as balancing invariants (height, color, etc.) are **not** returned.*

join_mid creates a new tree with e at the root and L and R as its subtrees (**and then balances that tree**)

```
type NodeView<Tree : BinaryTree<Elem>> = {
  left :  Tree,
  e :     Elem,
  right : Tree
}
```

expose returns *only what the algorithm needs to see*

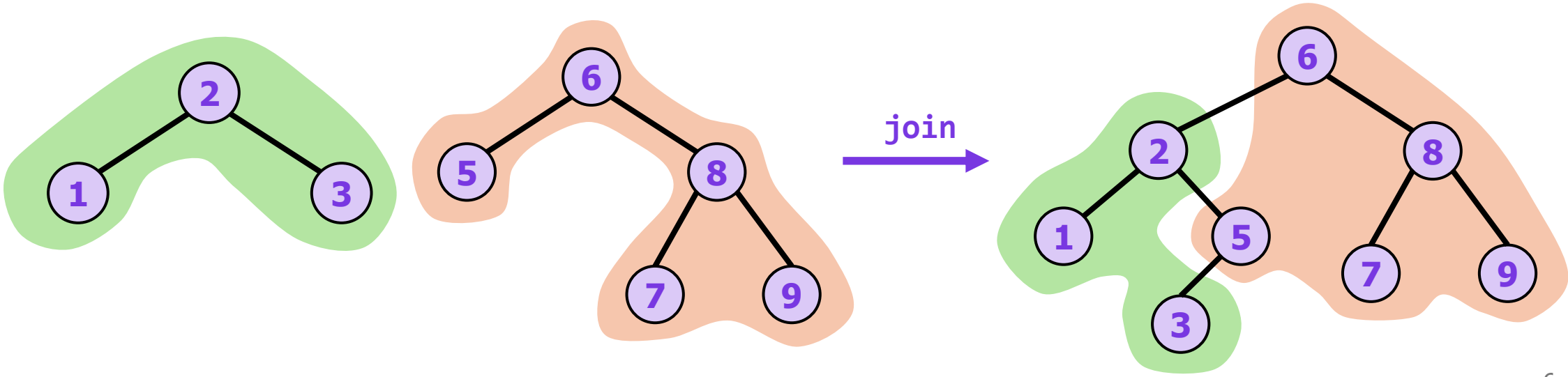
Review: Writing Generic Algorithms

```
fun contains(T : BinaryTree<K>, k : K) -> bool:
  match expose(T) with:
    case NONE: return False
    case SOME(NodeView(L, k', R)):
      if k < k': return contains(L, k)
      else if k > k': return contains(R, k)
      else: return True
```

Same algorithm, any binary-tree representation. `contains` now only depends on `expose`, so it works for AVL trees, red-black trees, treaps, etc, once we implement `expose` for each of them

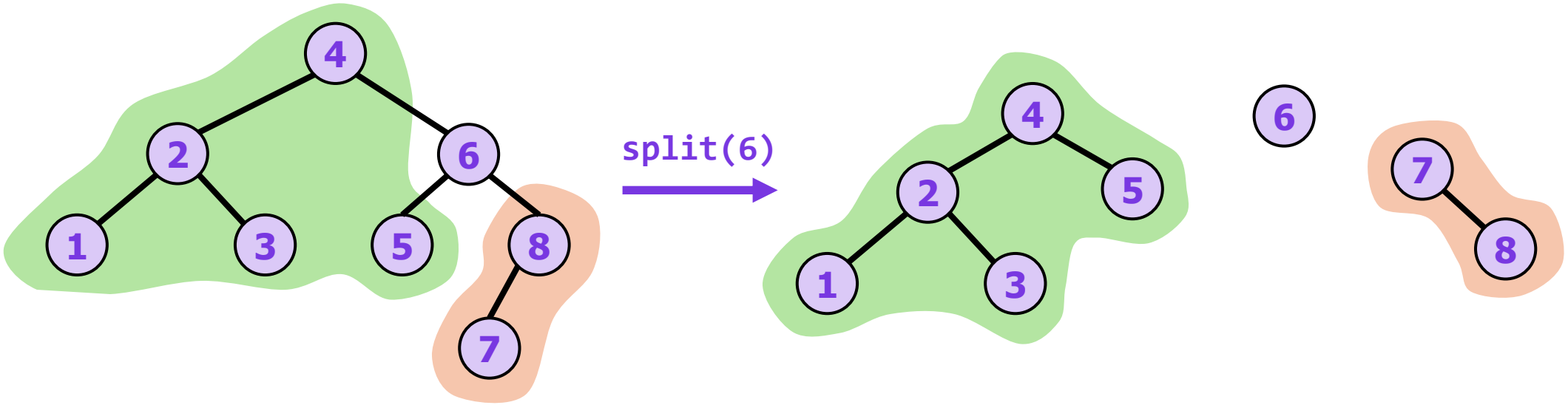
Review: Structural Operations on BSTs

- 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.



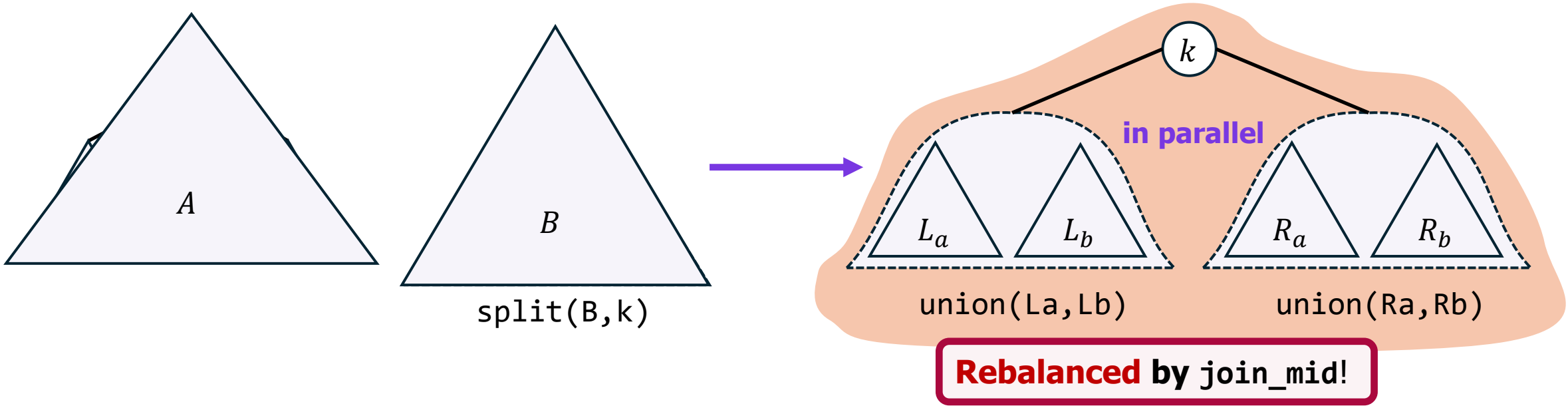
Dictionary Algorithms (Continued)

Union

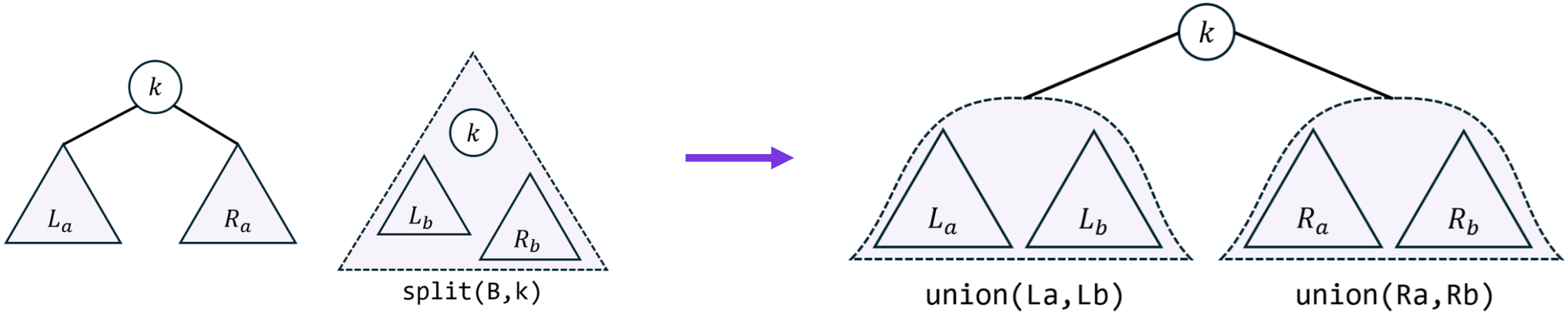
- **`union(A, B) ->`** `BST<K,V>`: return the union of `A` and `B`

Caveat: Which value do we keep if a key is in both? Doesn't matter for **sets**, but necessary for a dictionary.

Simplest choice: just use value from A.



Union




```
fun union(A : BST<K,V>, B : BST<K,V>) -> BST:
  match (expose(A), expose(B)) with:
    case (NONE, _): return B
    case (_, NONE): return A
    case (SOME(NodeView(La, (k,v), Ra), _)):
      Lb, _, Rb = split(B, k)
      L, R = parallel (union(La, Lb), union(Ra, Rb))
      return join_mid(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 paper (*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

$$W(m,n) = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = 0 \\ 1 + \log(1+n), & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log(n+m), & \text{otherwise} \end{cases}$$

Cost of Union: Brick Method

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

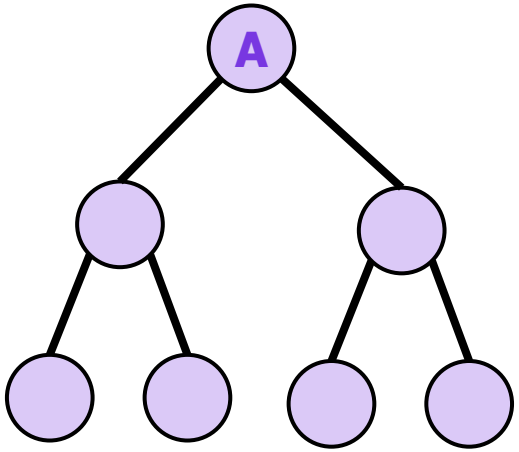
- **Parent work**: $\log(n+m)$
- **Child work**: $\log\left(\frac{m}{2} + |L_b|\right) + \log\left(\frac{m}{2} + |R_b|\right)$ where $|L_b| + |R_b| \leq n$
- **But:** the cost of the children depends on $|L_b|$ and $|R_b|$, which can vary Let's consider the **worst case**.

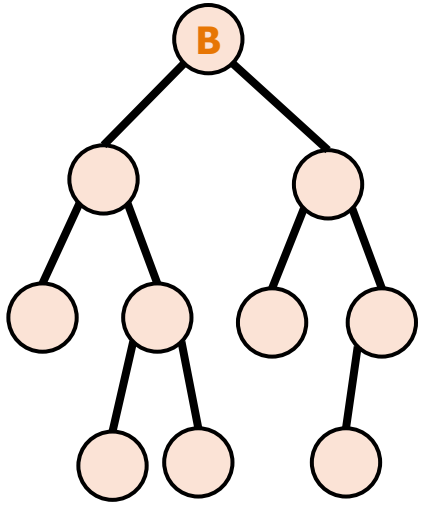
Key Observation: In the **worst case**, $|L_b| = |R_b| = n/2$ because $\log$ is a **concave** function. Formally, Jensen's inequality says:

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

Leaf Dominated!

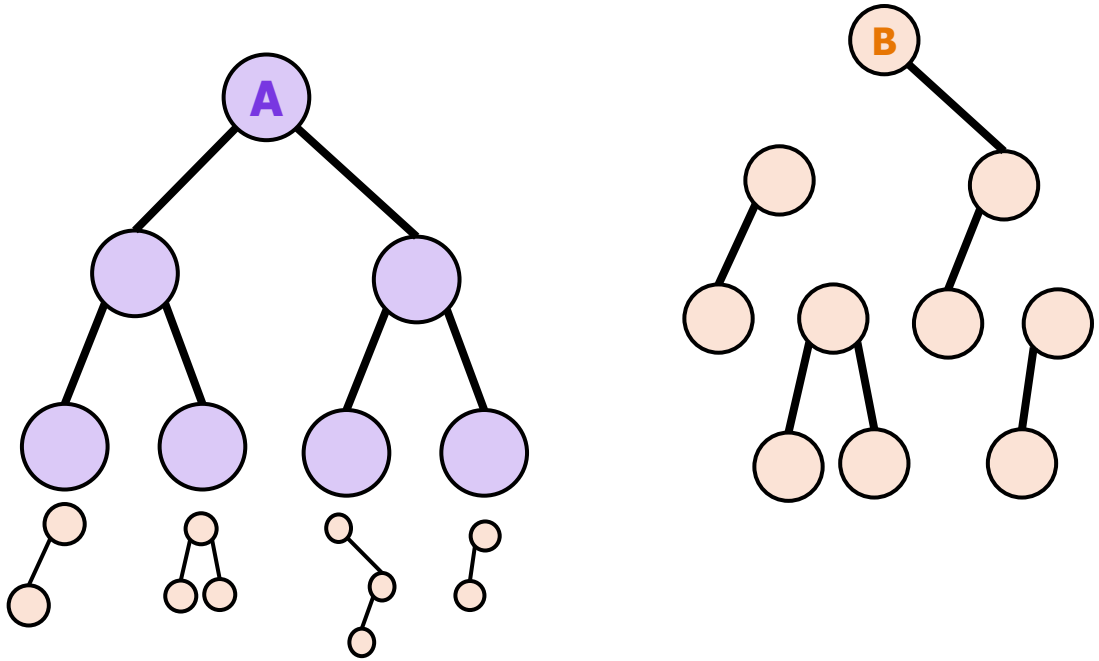
$$W(m,n) = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = 0 \\ 1 + \log(1+n), & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log(n+m), & \text{otherwise} \end{cases}$$





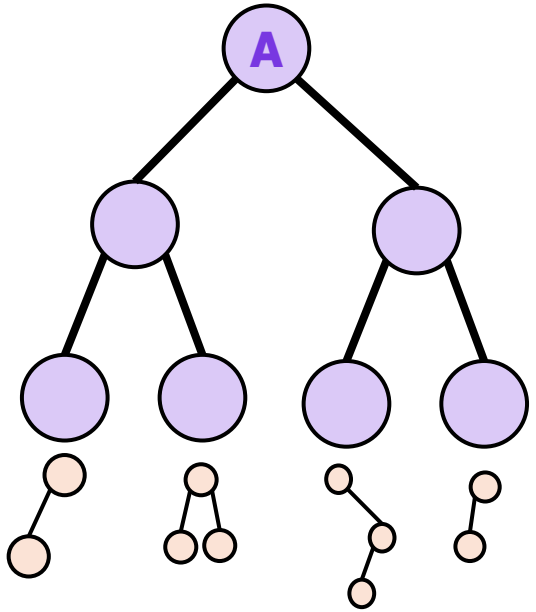
Leaf Dominated!

$$W(m,n) = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = 0 \\ 1 + \log(1+n), & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log(n+m), & \text{otherwise} \end{cases}$$



Leaf Dominated!

$$W(m,n) = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = 0 \\ 1 + \log(1+n), & \text{if } m = 1 \\ W\left(\frac{m}{2}, |L_b|\right) + W\left(\frac{m}{2}, |R_b|\right) + \log(n+m), & \text{otherwise} \end{cases}$$



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

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

Total Cost of the Leaves

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

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

- 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 $(m+1)/2$ leaves, so $b_i = 2n/(m+1)$

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

Span

- There are $O(\log m)$ levels
- Each level pays $O\big(\log (n+m)\big)$ local work
- $m \leq n$, so the span is at most $O(\log m \log n)$

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.

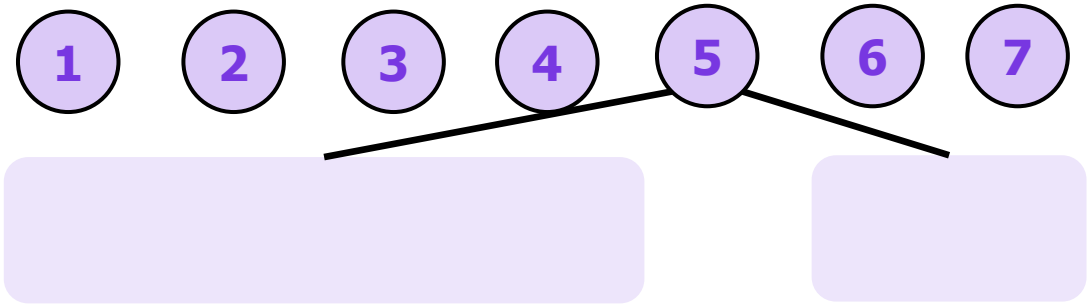
Treaps

Shaping a BST

- Given a set of keys, there are **many** valid BSTs on them

Observe: The left→right (***inorder***) order is unique determined by the keys. The thing that is free to choose is the ***depths*** of the nodes.

Idea: **choose the root**, then recursively build the left and right BSTs



?

Which algorithm does this remind you of?

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

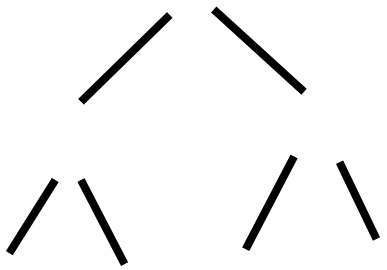
Treap = Tree + Heap

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.

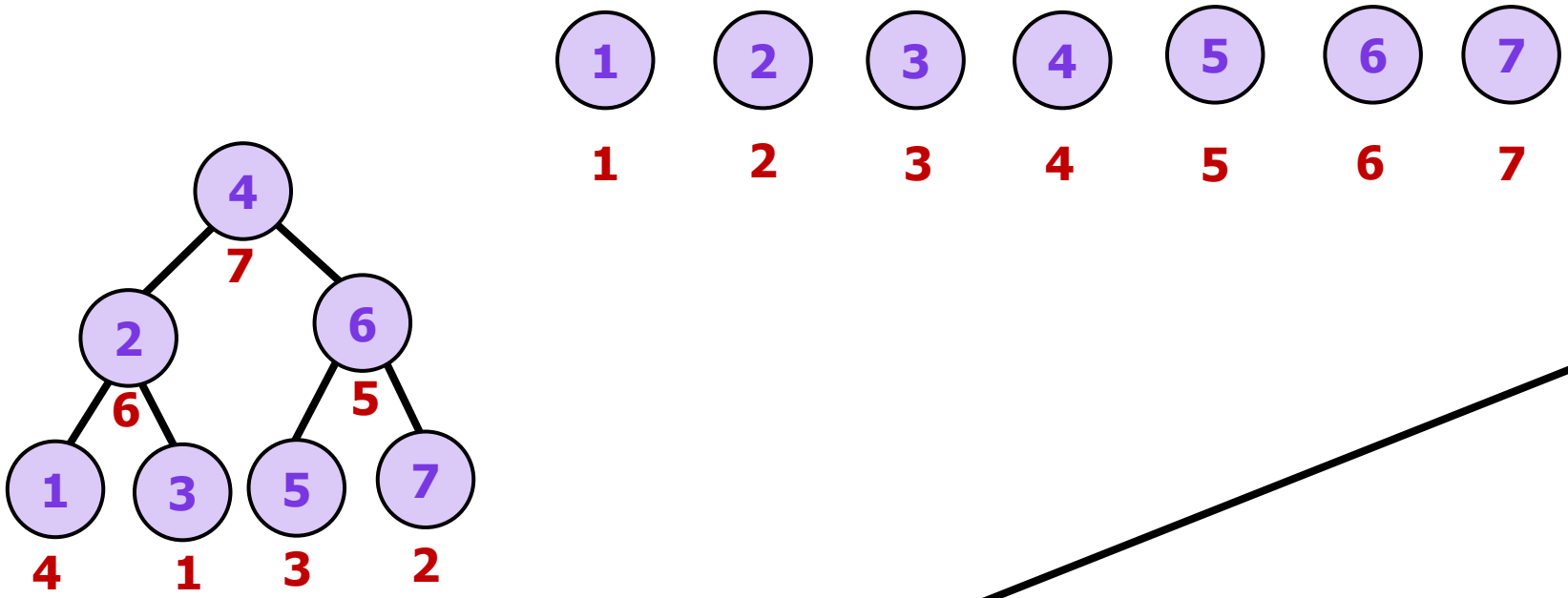
1	2	3	4	5	6	7
4	6	1	7	3	5	2

Recall: Max at root



Priorities ⇒ Tree Shape

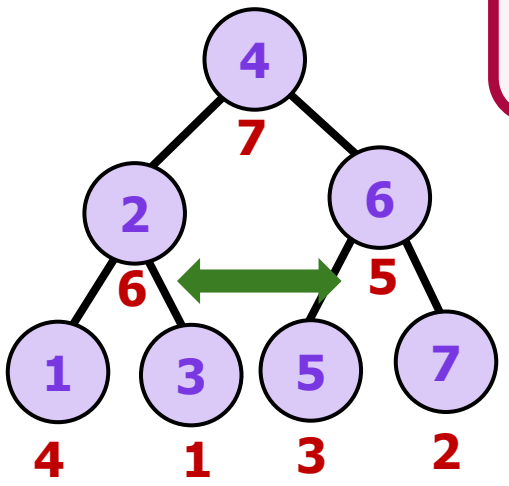
Claim (Priorities): Given a set of distinct keys with distinct priorities, there is a unique treap that satisfies the BST and heap invariants.

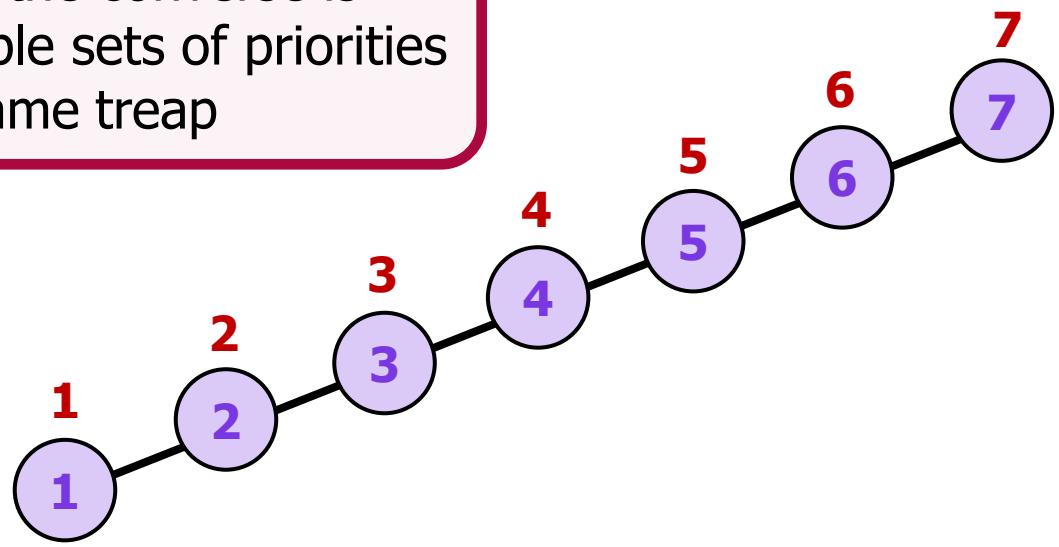


Priorities ⇒ Tree Shape

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

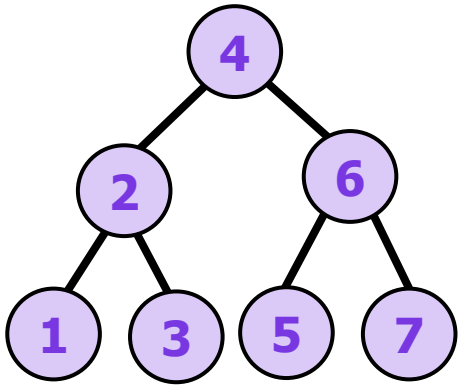




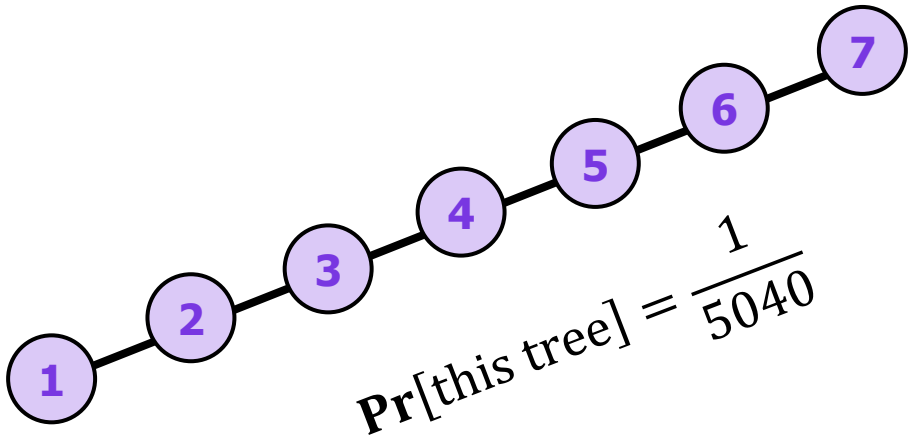
How to Choose Priorities?

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.



$$\mathbf{Pr}[\text{this tree}] = \frac{1}{63}$$



$$\mathbf{Pr}[\text{this tree}] = \frac{1}{5040}$$

Recall Quicksort

Algorithm (QuickSort):

```
fun quicksort(S : sequence<T>) -> sequence<T>:
  if length(S) <= 1: return S
  p = uniform_random_element(S)
  L = filter(fn x => (x < p), S)
  R = filter(fn x => (x > p), S)
  M = filter(fn x => (x == p), S)
  sortedL, sortedR = parallel (quicksort(L), quicksort(R))
  return sortedL + M + sortedR
```

How can we change this to build a treap?

Quicktreap

Algorithm ("Quicktreap"):

```
fun quicktreap(S : sequence<T>) -> Treap<T>:
  if length(S) <= 1: return Node(empty, S[0], empty)      // was return S
  p = key_with_max_priority(S)                 // was uniform_random_element(S)
  L = filter(fn x => (x < p), S)
  R = filter(fn x => (x > p), S)
  M = filter(fn x => (x == p), S)
  left_treap, right_treap = parallel (quicktreap(L), quicktreap(R))
  return Node(left_treap, p, right_treap)   // was  sortedL + M' + sortedR
```

Cool Relationship:

Theorem (Treap/Quicksort Distribution): 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 roots 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, distribution of each subtree is the same.*

Implementing Treaps

Review: `BinaryTree` and `join_mid`

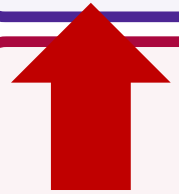
A type **`Tree`** is a `BinaryTree<Elem>` if it supports:

```
size(T : Tree) -> int
expose(T : Tree) -> optional<NodeView<Tree>>

empty : Tree
join_mid(L : Tree, e : Elem, R : Tree) -> Tree
```

`join_mid` creates a new tree with `e` at the root and `L` and `R` as its subtrees (**and then balances that tree**)

```
fun join_mid(L : MyTree<Elem>, e : Elem, R : MyTree<Elem>) -> MyTree<Elem>:
  return Node(L, e, 1 + size(L) + size(R), R)  // no balancing!
```



We must **replace this** with a proper implementation of `join_mid` that **rebalances** and preserves any other required invariant(s).

Implementing Treaps

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

```
empty : Treap<Elem> = Empty
```

```
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))
```

All of these are **identical** to our *unbalanced tree*. Only **join_mid** is going to differ!

Implementing Treaps

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

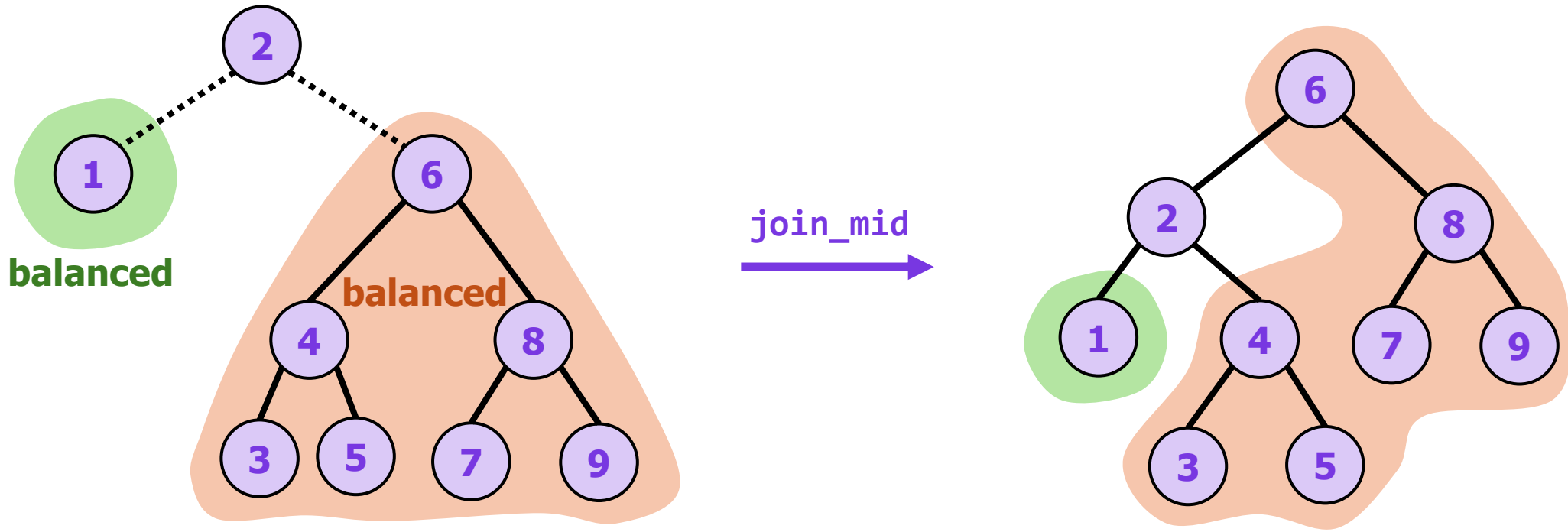
- The following helper function will be useful for `join_mid`
- **`make_node`** creates a `Node` without balancing it, assuming the caller has already done the necessary balancing

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

`make_node` is useful because it automatically handles calculating the size.

Remember: `join_mid` rebalances!

`join_mid(L,e,R)` gets to assume as a precondition that **L and R are already balanced**, but it is responsible for **making the result balanced**

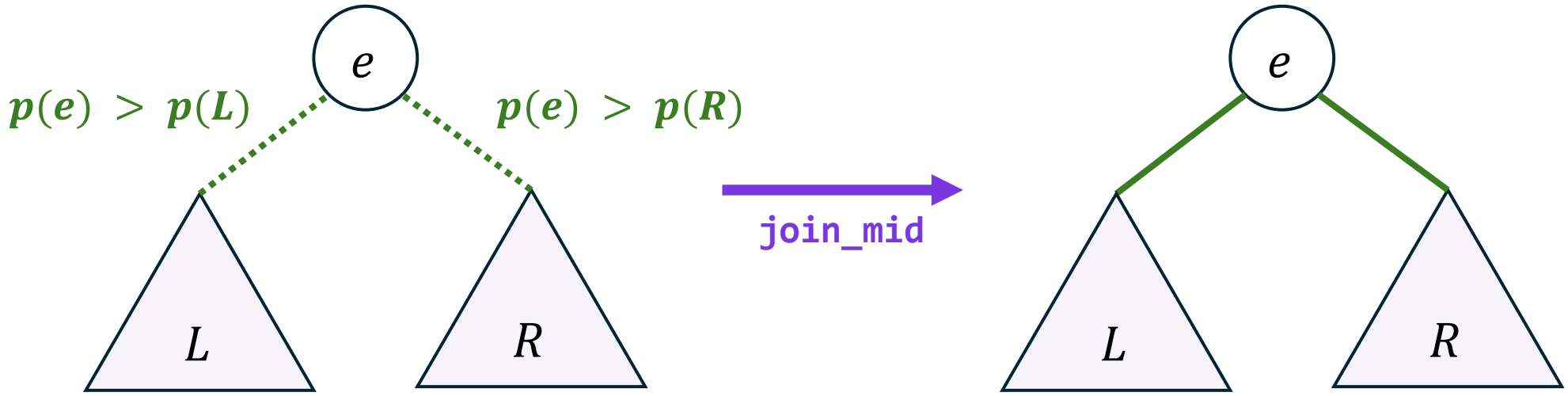


Priorities

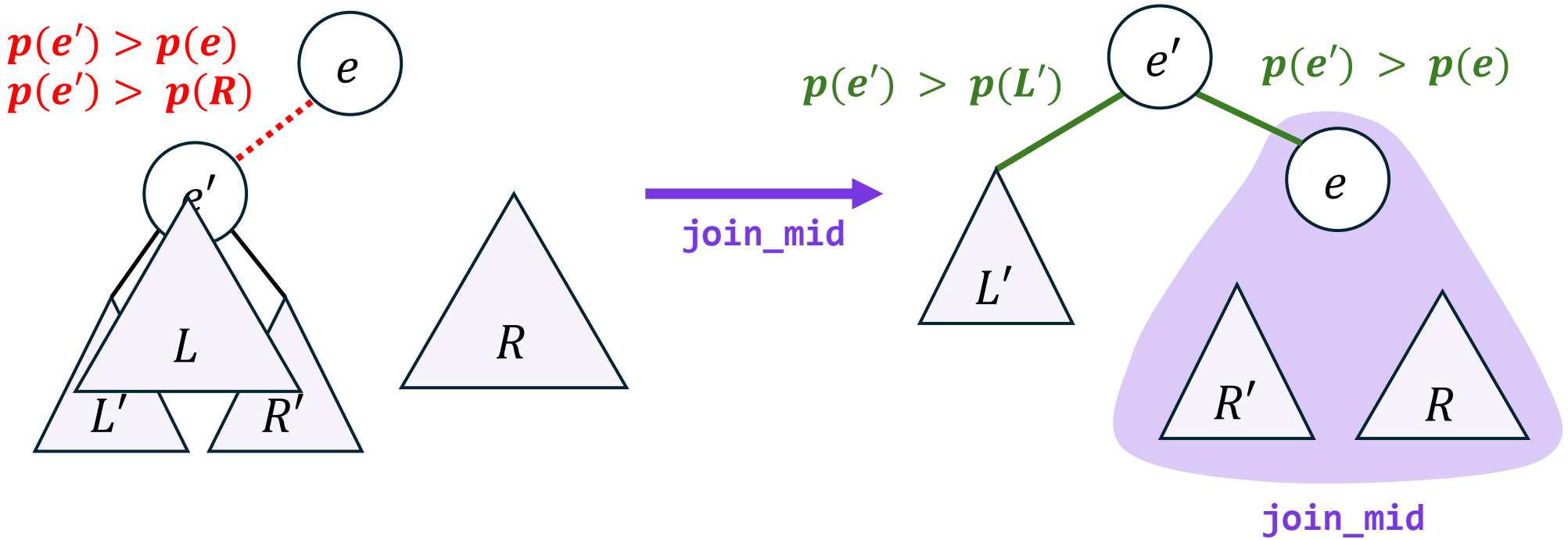
- Priorities can be implemented in many ways. One way is to use hashing to generate pseudorandom numbers
- We won't go into detail on this and just assume we have access to random priorities for each element
- We define a function that returns the priority of a tree from the root element, and assigns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

```
function priority(T: Treap<Elem>) -> int:
  match T with:
    case Empty: return -∞
    case Node(_,e,_,_): return p(e)  // something (pseudo)random
```

join_mid(L, e, R)



join_mid(L, e, R)



`join_mid` for treaps (Code)

```
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:  // priority(L) < priority(R)
    Node(L', e', _, R') = R
    return make_node(join_mid(L, e, L'), e', R')
```

Under 10 lines of code to balance a binary search tree!

Cost: at most $O(h(L) + h(R))$ since we go down 1 level on each recursive call

Comparison

join_mid (AVL Tree), 34 lines

```
fun join_mid(L : AVLTree<Elem>, e : Elem, R : AVLTree<Elem>) -> AVLTree<Elem>:
  if h(L) > h(R) + 1:           // Left subtree too tall
    return join_right(L, e, R)
  else if h(R) > h(L) + 1:      // Right subtree too tall
    return join_left(L, e, R)
  else:                         // Invariant already true: can join directly
    return make_node(L, e, R)

fun join_right(L : AVLTree<Elem>, e : Elem, R : AVLTree<Elem>) -> AVLTree<Elem>:
  if h(L) <= h(R) + 1:
    return make_node(L, e, R)
  Node(LL, eL, _, LR) = L
  T = join_right(LR, e, R)        // Continue down the right spine
  if h(T) <= h(LL) + 1:           // Already balanced
    return make_node(LL, eL, T)
  else:
    Node(TL, eT, _, TR) = T
    if h(TR) >= h(TL):            // Single left rotation
      return make_node(make_node(LL, eL, TL), eT, TR)
    else:                         // Double rotation (right -> left)
      Node(TLL, eTL, _, TLR) = TL
      return make_node(make_node(LL, eL, TLL), eTL, make_node(TLR, eT, TR))

fun join_left(L : AVLTree<Elem>, e : Elem, R : AVLTree<Elem>) -> AVLTree<Elem>:
  if h(R) <= h(L) + 1:
    return make_node(L, e, R)
  Node(RL, eR, _, RR) = R
  T = join_left(L, e, RL)         // Continue down the left spine
  if h(T) <= h(RR) + 1:           // Already balanced
    return make_node(T, eR, RR)
  else:
    Node(TL, eT, _, TR) = T
    if h(TL) >= h(TR):            // Single right rotation
      return make_node(TL, eT, make_node(TR, eR, RR))
    else:                         // Double rotation (left -> right)
      Node(TRL, eTR, _, TRR) = TR
      return make_node(make_node(TL, eT, TRL), eTR, make_node(TRR, eR, RR))
```

join_mid (Red-Black Tree), 35 lines

```
fun join_mid(L : RBTree<Elem>, e : Elem, R : RBTree<Elem>) -> RBTree<Elem>:
  L = blacken(L)
  R = blacken(R)
  if bh(L) == bh(R):              // Invariant already true: can join directly
    return make_node(L, e, R, BLACK)
  else if bh(L) > bh(R):          // Left subtree has greater black height
    return blacken(join_right(L, e, R))
  else:                           // Right subtree has greater black height
    return blacken(join_left(L, e, R))

fun join_right(L : RBTree<Elem>, e : Elem, R : RBTree<Elem>) -> RBTree<Elem>:
  if bh(L) == bh(R) and color(L) == BLACK:
    return make_node(L, e, R, RED)  // Attach here
  Node(LL, eL, cL, _, LR) = L
  T = join_right(LR, e, R)          // Continue down the right spine
  if cL == BLACK and color(T) == RED and color(T.right) == RED:
    // Repair red-red violation
    Node(TL, eT, _, _, TR) = T
    return make_node(make_node(LL, eL, TL, BLACK), eT, blacken(TR), RED)
  else:
    return make_node(LL, eL, T, cL)

fun join_left(L : RBTree<Elem>, e : Elem, R : RBTree<Elem>) -> RBTree<Elem>:
  if bh(R) == bh(L) and color(R) == BLACK:
    return make_node(L, e, R, RED)  // Attach here
  Node(RL, eR, cR, _, RR) = R
  T = join_left(L, e, RL)           // Continue down the left spine
  if cR == BLACK and color(T) == RED and color(T.left) == RED:
    // Repair red-red violation
    Node(TL, eT, _, _, TR) = T
    return make_node(blacken(TL), eT, make_node(TR, eR, RR, BLACK), RED)
  else:
    return make_node(T, eR, RR, cR)

fun blacken(T : RBTree<Elem>) -> RBTree<Elem>:
  if T == empty or color(T) == BLACK: return T
  Node(L, e, _, _, R) = T
  return make_node(L, e, R, BLACK)
```

Cost of Split and Join

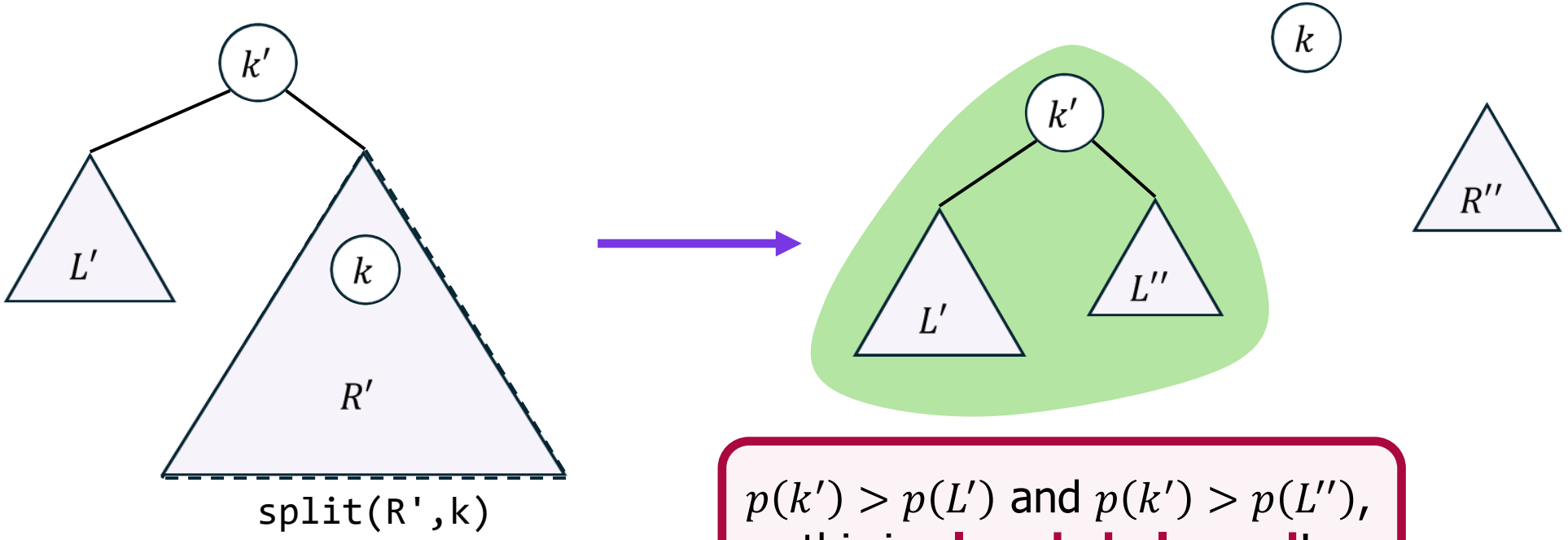
Split and Join

```
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')
```

- We call `join_mid` at every level of the tree, so this costs $O(h(T))$ per level for $O(h(T))$ levels ⇒ costs $O((h(T)^2))$?
- But we want `split` to cost $O(h(T))$!

Splitting a Treap

Theorem (Split Cost): Splitting a treap T costs $\Theta\big(h(T)\big)$ in the worst case



$p(k') > p(L')$ and $p(k') > p(L'')$,
so this is **already balanced**!

Joining A Treap

Corollary (Join Cost): `join(A,B)` on treaps costs $\Theta\big(h(T)\big)$ in the worst case

(where T is the *resulting* tree)

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

Proof: `join` *just calls* `split` *and* `join_mid`*, costing* $O(h(R))$ *and* $O(h(L))$

Height Analysis of Treaps

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[1], the **expected depth** of node k is $O(\log n)$.

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

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

[1] assume we pick from a large enough range so there are no repeats

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

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

(for math simplicity, a node is considered an ancestor of itself)

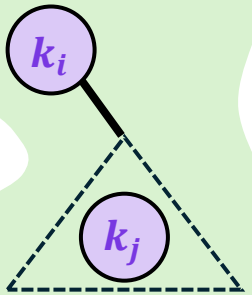
Treap Node Depth

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

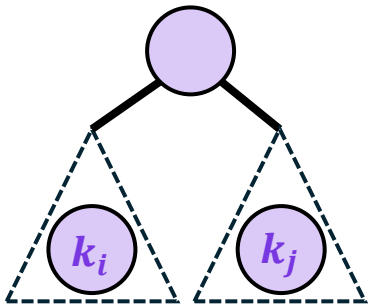
$$\mathbf{E}\left[A_j^i\right] = \mathbf{Pr}\left[k_i \text{ is an ancestor of } k_j\right] = \frac{1}{|i-j|+1}$$

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

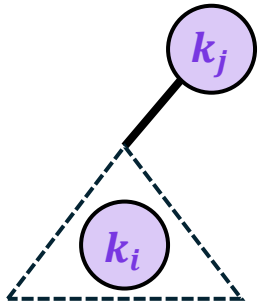
Case 1: k_i



Case 2: Some key between



Case 3: k_j



Expected Depth Calculation

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

- *Using **linearity of expectation***

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

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

Expected Depth Calculation

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

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

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

$$= \left(H_{j+1} - 1\right) + H_{n-j} \qquad = \Theta(\log n)$$

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

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

Recall: Cool Relationship

Theorem (Treap/Quicksort Distribution): The distribution of tree shapes for the randomized quicksort **recursion tree**, and for **treaps** with random distinct priorities is identical (assume |S| > 0).



Corollary (Tree Height w.h.p.): Given a set of n keys, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the height of the tree is $O(\log n)$ **w.h.p.**

Proof: Follows directly since we know the recursion depth of quicksort is $O(\log n)$ *w.h.p.*

High-Probability Cost Bounds

Corollary (Tree Height w.h.p.): Given a set of n keys, if a treap is built on those n keys with priorities chosen uniformly and independently at random, the height of the tree is $O(\log n)$ **w.h.p.**

Theorem (Split Cost): Splitting a treap T costs $\Theta\big(h(T)\big)$ in the worst case

+

Corollary (Join Cost): `join(A,B)` on treaps costs $\Theta\big(h(T)\big)$ in the worst case

⇓

Corollary (Cost of Split and Join): When invoked on a treap, the worst-case cost of the generic `split` and `join` algorithms is $O(\log n)$ **w.h.p.**

Summary

- **Union** on balanced binary search trees costs $O\left(m + m \log\left(\frac{n}{m}\right)\right)$ work and $O(\log n \log m)$ span.
- **Treaps** are a **randomized** balanced binary search tree:
  - based on assigning random priorities to keys,
  - higher priority ⇒ higher in the tree.
- Our core function for balancing binary search trees, **`join_mid`**, is under 10 lines of code for a treap!
- Treaps have $O(\log n)$ height w.h.p.
- **split** and **join** cost $O(\log n)$ w.h.p. on treaps.