Parallel And Sequential Data Structures and Algorithms

Priority Queues and Leftist Heaps

Amruta

Cole

Learning Objectives

- Understand the definition of a **priority queue** and what operations it supports
- Analyze cost bounds for different PQ implementations
- Introduce the **meld** operation and discuss its uses
- Figure out how to implement **meld**
- Learn about **Leftist heaps** and prove that they give us logarithmic-time **meld**

What is a Priority Queue?

- Efficient access to minimum (or maximum) element of a set!!
  - Prim's, Dijkstra's
- Queue-like data structure: instead of access to first element, provide access to smallest element
  - Highest priority = smallest
  - Lowest priority = largest

Gives us a min-PQ…

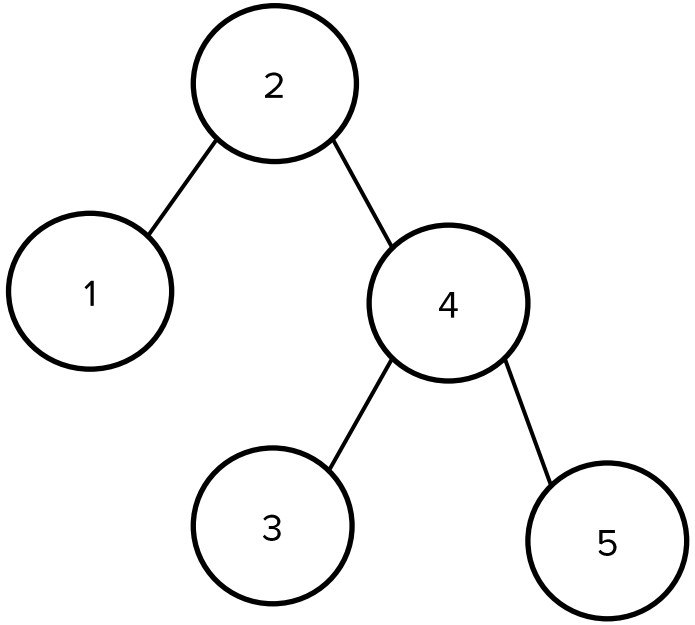
🚨 **IMPORTANT: Key type must be comparable/totally ordered**

Operations

- **PriorityQueue<K>** typically supports two operations:
  - `insert`: efficiently insert a new key
  - `deleteMin`: retrieve the smallest key and remove from PQ
- Costs for each depend on implementation
  - Balanced BST
  - Binary Heap
  - Sorted/Unsorted List

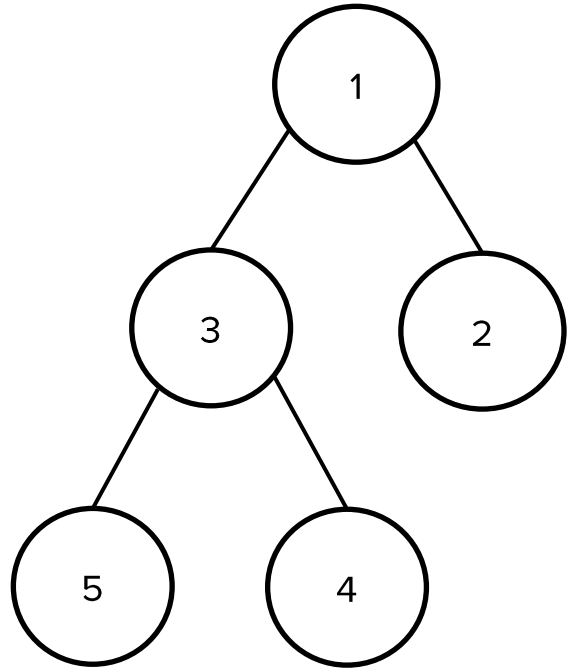
Implementation 1: Balanced BST

- Keys are already in placed sorted order
  - `insert`: $O(\log n)$
    - BST `insert` operation
  - `deleteMin`: find leftmost node of tree in $O(\log n)$
    - BST `first` operation
  - `fromSeq`: in $O(n \log n)$



Implementation 2: Binary Heap

- **Min-Heap** invariant: Node's parent has smaller key than node
- **Shape** invariant: Fill lowest level from left to right
  - `insert`: $O(\log n)$
    - Insert into lowest level, Sift upwards $O(\log n)$ levels
  - `deleteMin`: $O(\log n)$
    - Remove node at root, Restore shape invariant
  - `fromSeq`: $O(n)$



Inefficient Implementations: Lists

- Sorted List
  - `insert`: $O(n)$
  - `deleteMin`: $O(1)$
  - `fromSeq`: $O(nlogn)$
- Unsorted List
  - `insert`: $O(1)$
  - `deleteMin`: $O(n)$
  - `fromSeq`: $O(n)$

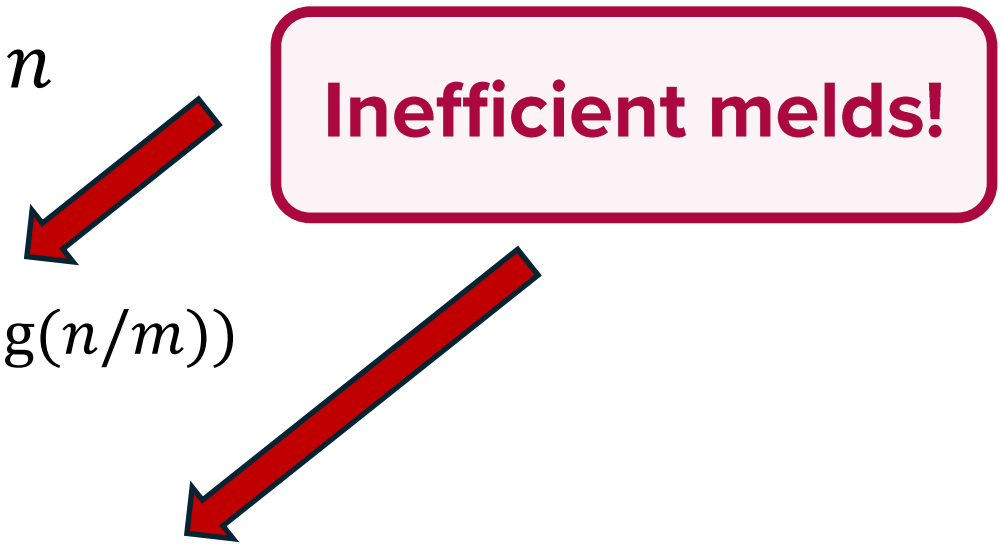
Linear time is inefficient!!!

The `meld` Operation

- `meld<Q1: PQ<K>, Q2: PQ<K>> → Q3: PQ<K>`
  - Acts as a Union of two PQs
- `Q1 = <1, 4, 8, 12>, Q2 = <3, 9, 12>`
  - `meld(Q1, Q2) = <1, 3, 4, 8, 9, 12, 12>`

`meld` Cost Bounds

- Assume |Q1| $= m$, |Q2| $= n$, $m \leq n$
- With BST
  - Use BST `Union` operation in $O(m + m \log(n/m))$
- With Binary Heap
  - Delete each element from `Q1` and add to `Q2` in $O(m \log n)$
- Can we do better? Can we implement `meld` in logarithmic time?



Stick to heap data structure!

- We will introduce leftist heaps shortly
- But assuming we ***do*** have access to a data structure with logarithmic time `meld` … what can we do?

```
type PQ<K: Ordered> =
    Empty
  | Node { left: PQ
           key: K
           right: PQ }
```

Priority Queue type definition: functional, assumes a heap implementation

Let's Implement Some Operations...

```
fun singleton(k: K) -> PQ<K>:
  return Node(Empty, k, Empty)

fun insert(Q: PQ<K>, k: K) -> PQ<K>:
  return meld(Q, singleton(k))

fun deleteMin(Q: PQ<K>) -> (option<K>, PQ<K>):
  match Q with:
    case Empty: return (NONE, Empty)
    case Node(L, k, R): return (SOME(k), meld(L, R))

fun fromSeq(S: sequence<K>) -> PQ<K>:
  return reduce(meld, Empty, map(singleton, S))
```

Note: `meld` preserves the heap invariant!

Cost Analysis

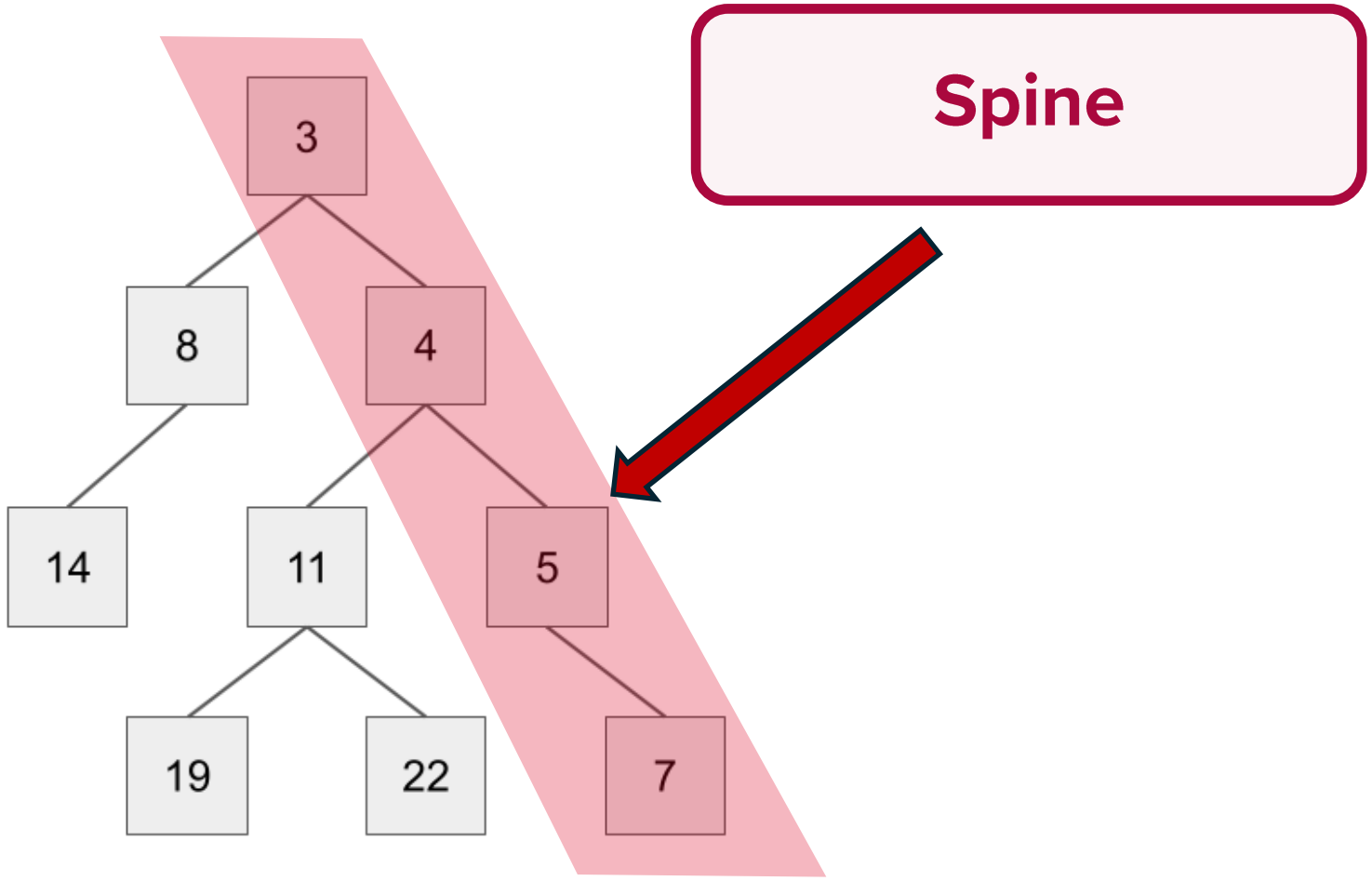
- Assume $O(\log(m+n))$ implementation of meld
  - `insert`: $O(\log(1+n)) = O(\log n)$
  - `deleteMin`: $O(\log(n/2 + n/2)) = O(\log n)$
  - `fromSeq`:
    - $W(n) = 2W\left(\frac{n}{2}\right) + O(\log n)$ Leaf Dominated $\Rightarrow W(n) = O(n)$
    - $S(n) = S\left(\frac{n}{2}\right) + O(\log n)$ Balanced $\Rightarrow S(n) = O(\log^2 n)$

So how do we achieve this bound on `meld`?

A general meld implementation:

```
fun meld(A: PQ<K>, B: PQ<K>) -> PQ<K>:
  match (A, B) with:
    case (Empty, _): return B
    case (_, Empty): return A
    case (Node(L_a, k_a, R_a), Node(L_b, k_b, R_b)):
      if (k_a < k_b):
        return makeNode(L_a, k_a, meld(R_a, B))
      else:
        return makeNode(L_b, k_b, meld(R_b, A))
```

A meld example



How much time does this take?

- We always meld down the right spine
- In the worst case, the spine is n nodes, and so we take $O(n)$ time

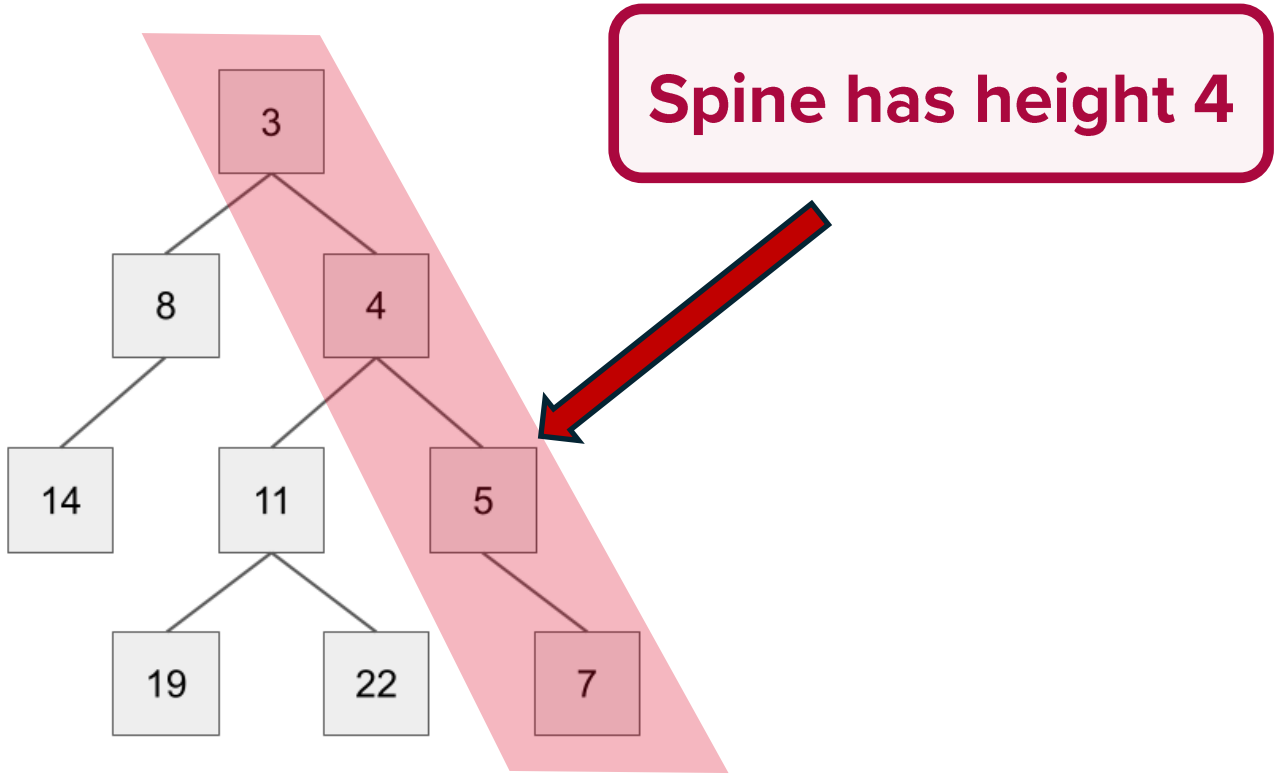
This is bad.

How can we make this better?

- We always meld down the right spine
- If we can bound the height of that, we can get a **faster** meld!

Rank

Definition (Rank): The **rank** of a priority queue is the height of its **right** spine

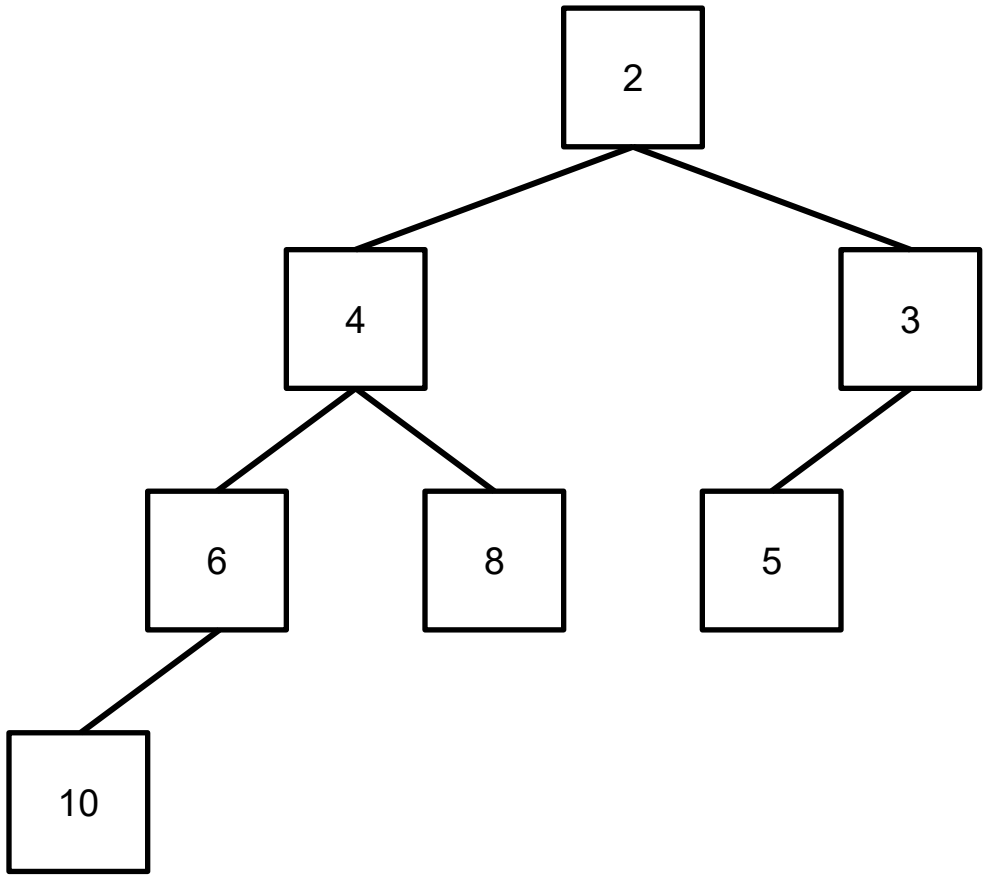


Leftist Heaps

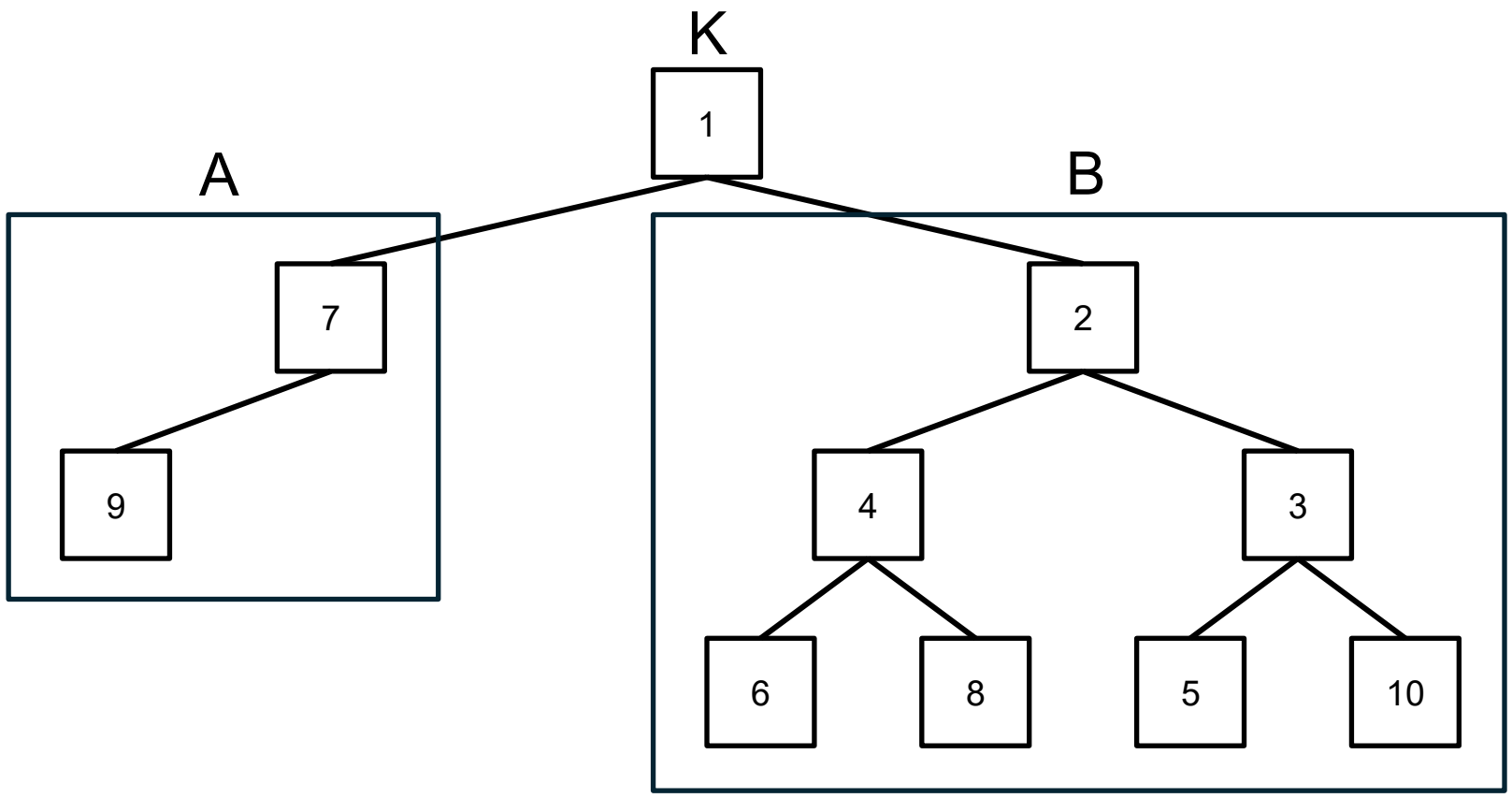
Definition (Leftist Heaps): A **leftist heap** is one where for every Node(*L, k, R*), the rank of *L* is ≥ the rank of *R*

```
type PQ<K: Ordered> =
    Empty
  | Node { left: PQ
           key: K,
           rank: int,
           right: PQ }
```

Leftist Heaps



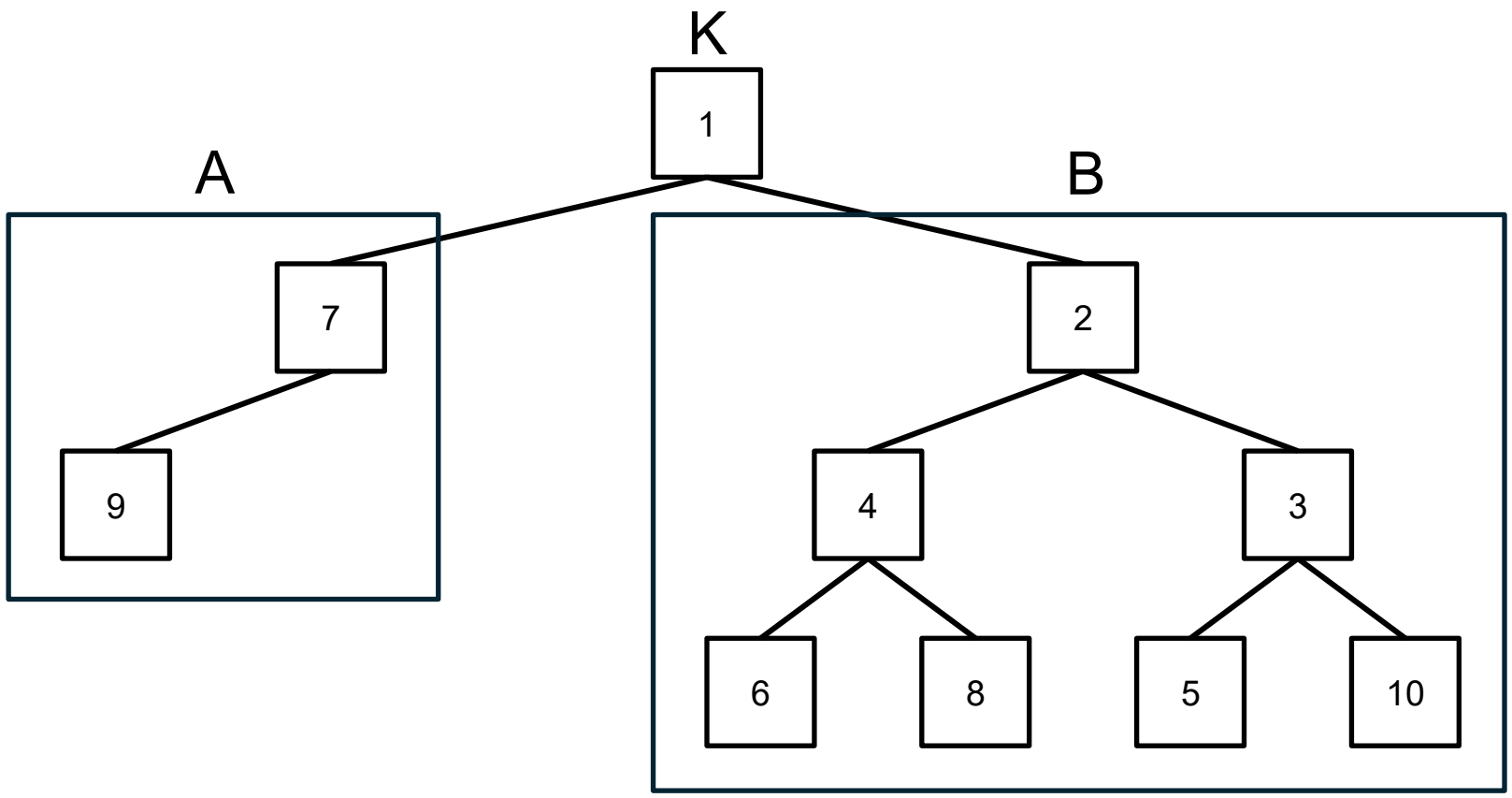
Making a Leftist node



makeNode

```
fun makeNode(A: PQ<K>, k: K, B: PQ<K>) -> PQ<K>:
  (_, _, r_a, _) = A
  (_, _, r_b, _) = B
  if (r_b < r_a):
    return Node(A, k, r_b + 1, B)
  else:
    return Node(B, k, r_a + 1, A)
```

Making a Leftist node (part 2)



Proving the meld bound

Lemma (Minimum Size of Leftist Heaps): Define a function $m(r)$ to be equal to the **minimum number of nodes** in a leftist heap with rank r. Our claim is that:

$$m(r) = 2^r - 1.$$

Proof (by induction)

- **Base case:** If $r = 0$, then there can be no nodes in our tree and

$$2^0 - 1 = 0$$

  as desired
- **Inductive case:** Our inductive hypothesis is that $m(r) = 2^r - 1$. To construct a tree of rank $r + 1$ we need a root key, as well as left and right subtrees of rank r. Then the size of our resulting tree is:

$$(2^r - 1) + (2^r - 1) + 1 = 2^{r+1} - 1$$

  where the first two terms come from our IH, and the last is the new root. This proves our desired result.

Bounding the rank of a fixed-size queue

Corollary (Maximum Rank of a PQ): If we have a PQ of size $|Q|$ and rank r, then:

$$2^r - 1 \leq |Q|$$

and so

$$r \leq \log_2(|Q| + 1)$$

This implies that merge is also $O(\log m + \log n)$!

Summary

- We've learned what a priority queue is and what operations we can use
- Different implementations have different bounds
- Meld merges two priority queues into one larger one
- Leftist heaps guarantee that the rank of the left subtree will be g.e.q. the rank of the right subtree
- This guarantees $O(\log(m + n))$ span for meld.