15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 25

Priority Queues

Synopsis

- Priority Queues
- Heaps
- Meldable Priority Queues
- Leftist Heaps

Priority Queues

- Abstract Data Type supporting
  - `deleteMin`/`deleteMax`
  - `insert`
- Used in many useful algorithms
  - Dijkstra' Algorithm
  - Prim's Algorithm for MST
  - Constructing Huffman Codes
  - *Heapsort*

HEAPSORT

```
1   fun sort S =
2   let
3       pq = iter Q.insert Q.empty S
4       fun sort' pq =
5       let
6           case (PQ.deleteMin pq) of
7               NONE ⇒ []
8             | SOME(v, pq') ⇒ v :: sort'(pq')
9   in
10      Seq.fromList(sort'pq)
11  end
```

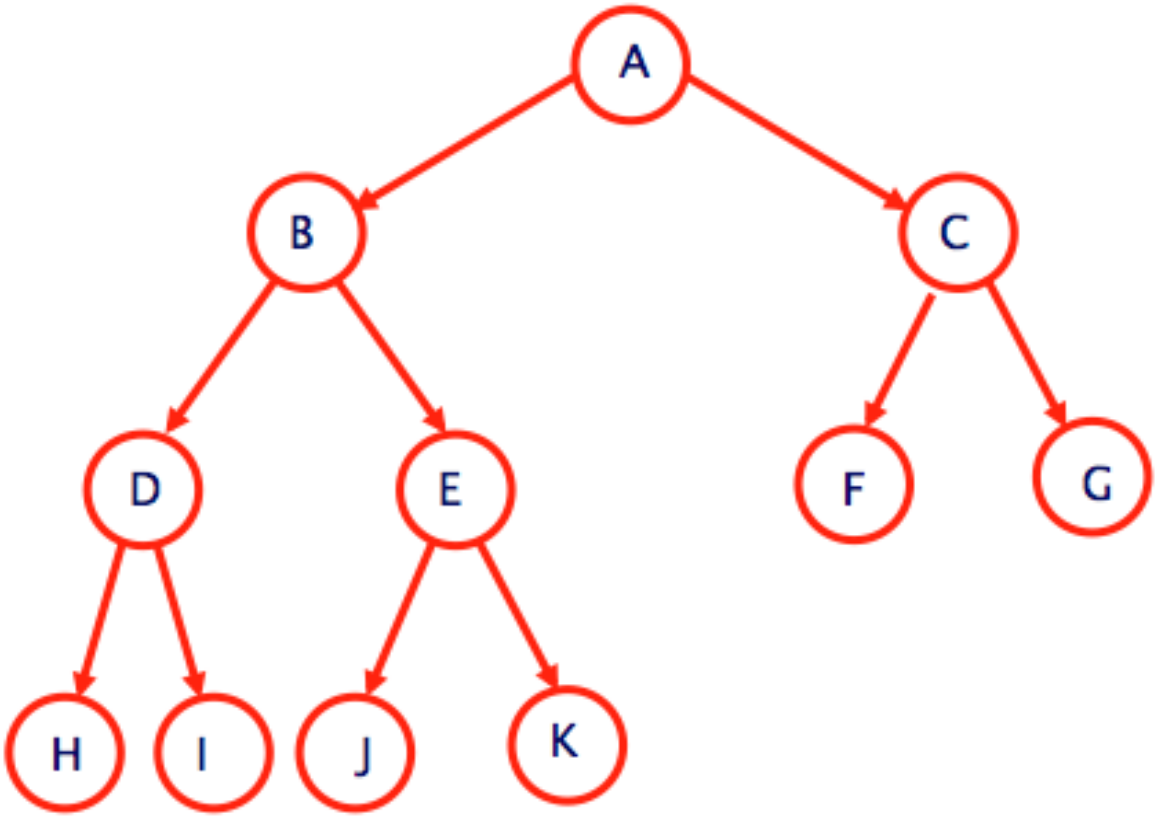
Underlying Implementations

- **Sorted and Unsorted Lists/Arrays**
  - One of `deleteMin` and `insert` is fast ($O(1)$)
  - The other is slow. $O(n)$
- **Balanced binary search trees**
  - Both operations have $O(\log n)$ work and span.
- **Binary heaps**
  - Both operations have $O(\log n)$ work and span.
  - But binary heaps provide a $O(1)$ work `findMin` operation.

Heaps

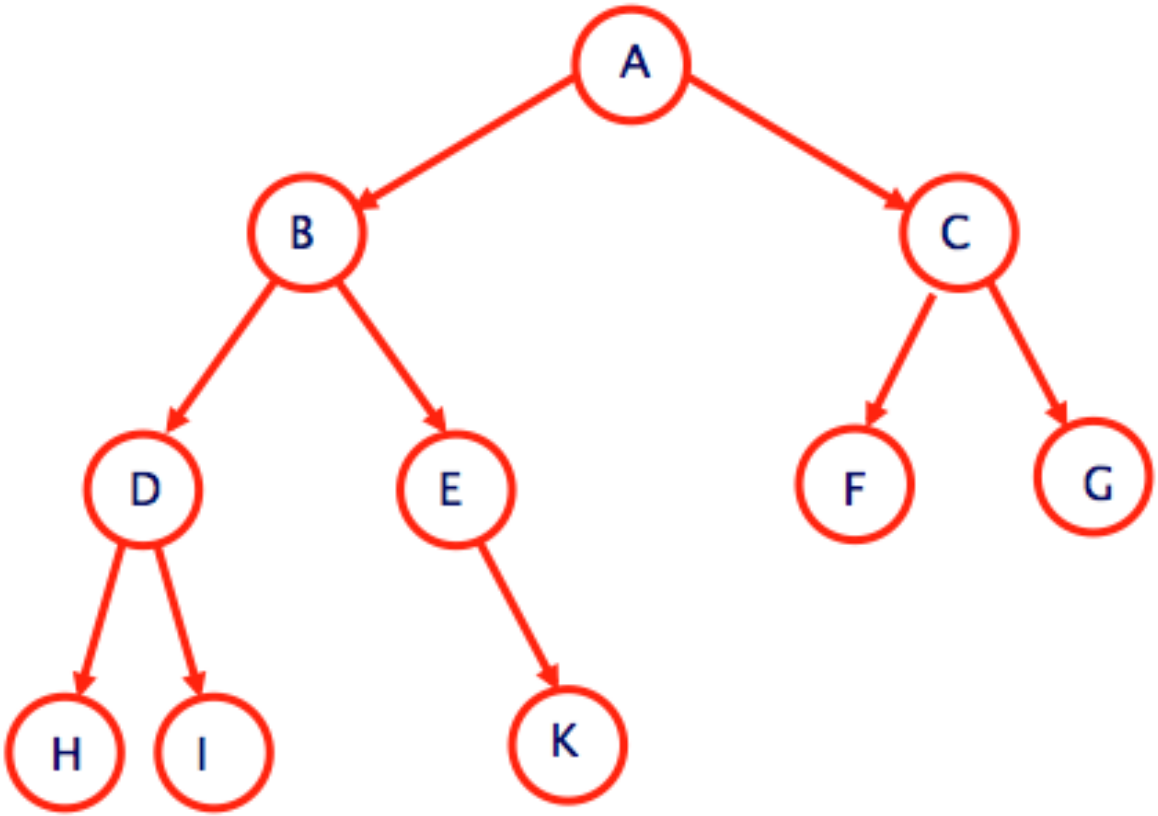
- A *min-heap* (*max-heap*) is a rooted tree
- Key at every node is $\leq$ ($\geq$) all descendants.
- A *binary heap* is heap which has
  - *Shape property*: The tree is a complete binary tree
    - All levels of the tree are completely filled except the bottom level, which is filled from the left
  - *Heap Property*

Binary Heaps



A complete tree

Binary Heaps

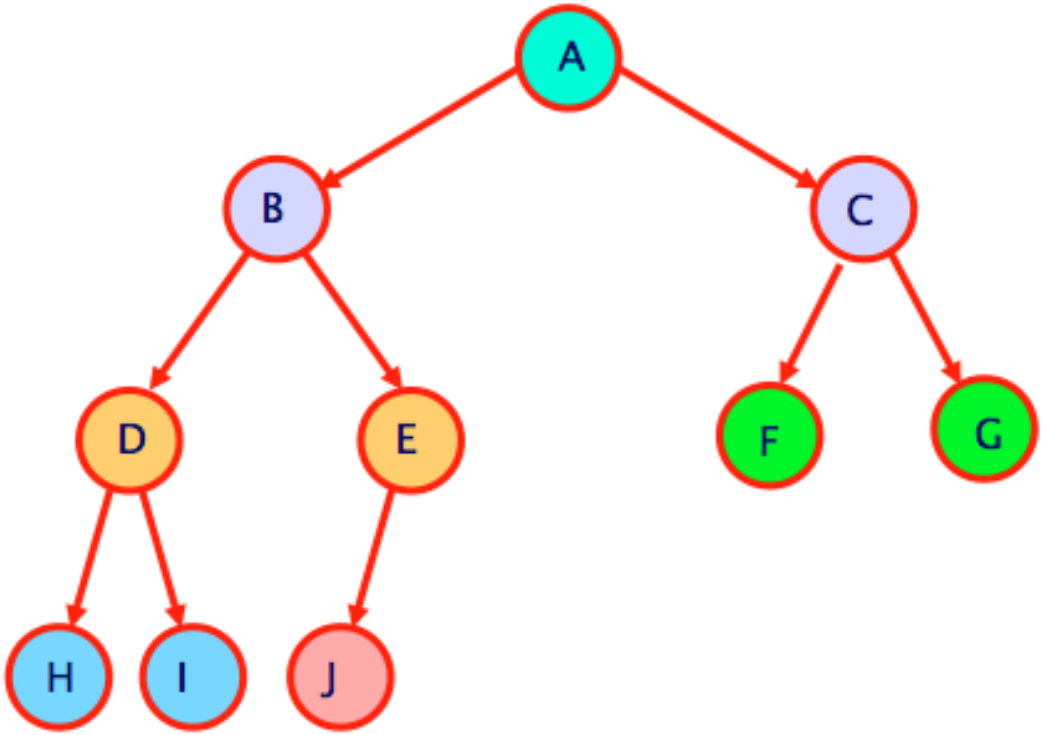


An incomplete tree

Binary Heaps

- Shape Property $\Rightarrow$ binary heap can be maintained in an array.
- Index of a parent or a child is very easy to compute
- Operations first restore shape property, then heap property.

Binary Heaps and Arrays



Building Priority Queues

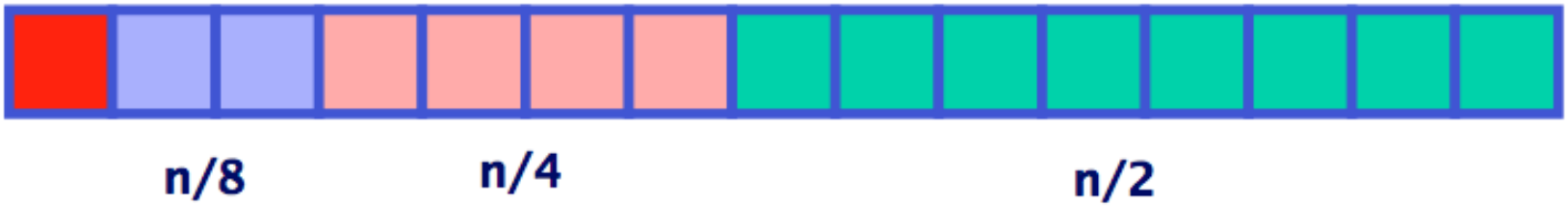
- We can insert elements one-by-one
  - With balanced binary trees and binary heaps, work is $O(n \log n)$
  - Can we do better?
- Build the heap recursively
  - If left and right sides are already heaps, just *shift down* the root element.

Building Heaps Directly

```
1  fun sequentialFromSeq S =
2  let
3     fun heapify(S, i) =
4        if (i >= |S|/2) then S
5        else let
6                S' = heapify(S, 2 * i + 1)
7                S'' = heapify(S', 2 * i + 2)
8             in shiftDown(S'', i) end
9  in heapify(S, 0) end
```

Cost Analysis

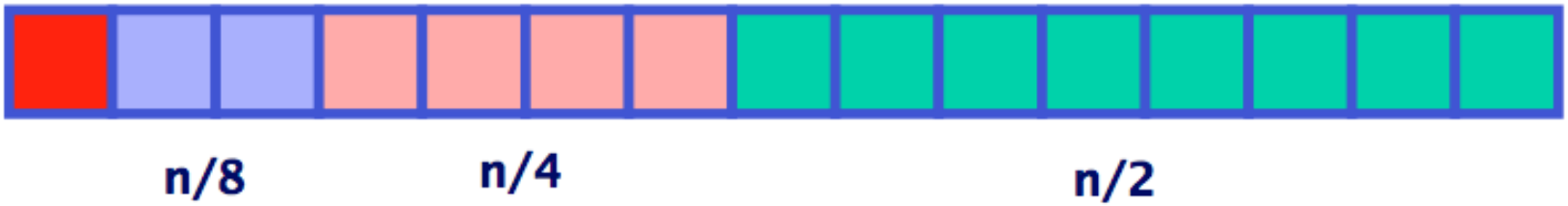
- `shiftDown` does $O(\log n)$ work on subtree of size n
- $W(n) = 2W(n/2) + O(\log n) \in O(n)$
- Opportunities for parallelism?

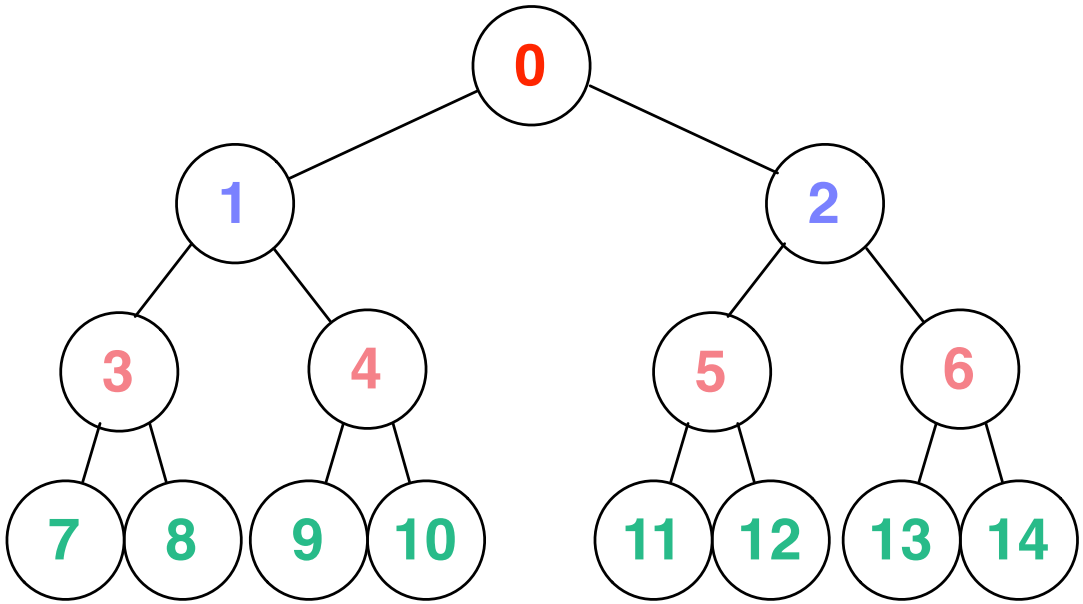


Parallel Heapify

- Green cells are OK
- All the pinks cells can be shifted down in parallel
- Then all purple cells can be shifted down in parallel
- (All) Red cell(s) can be shifted down in parallel

Parallel Heapify





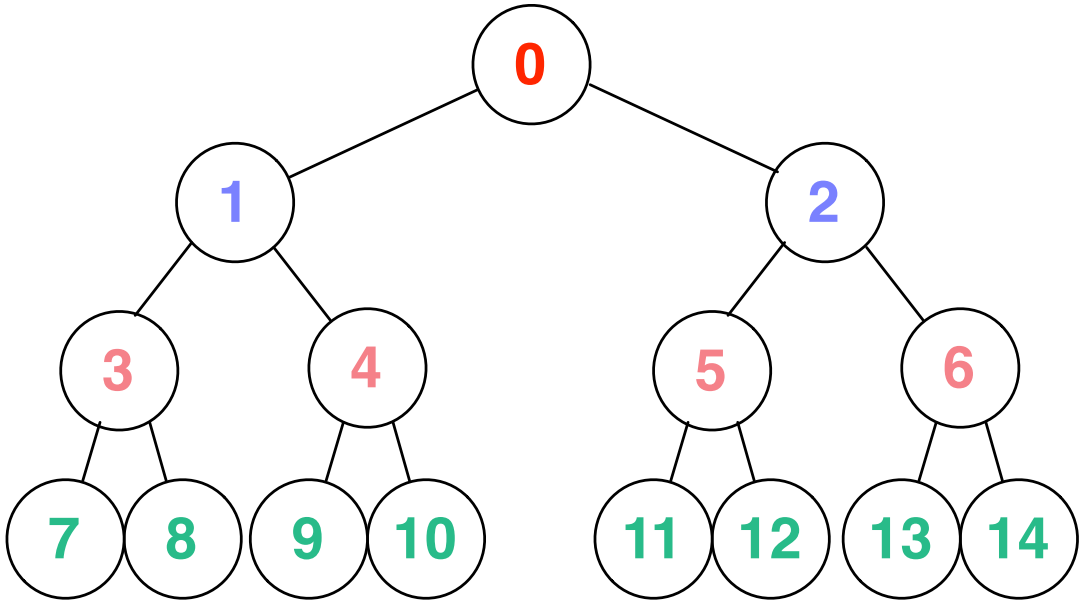
Parallel Heapify

- We use Single-threaded sequences

```
1  fun fromSeq S : 'a seq =
2  let
3      fun heapify (S, d) =
4      let
5          S' = shiftDown (S, ⟨2^d − 1, ..., 2^(d+1) − 2⟩, d)
6      in
7        if (d = 0) then S'
8        else heapify (S', d − 1)
9  in heapify (S, ⌊log2 n⌋ − 1) end
```

- $S(n) = S(n/2) + O(\log n) \in O(\log^2 n)$

Parallel Heapify



- $d = 2 \Rightarrow$ `shiftDown` (S, <3, 4, 5, 6>, 2)
- $d = 1 \Rightarrow$ `shiftDown` (S, <1, 2>, 1)
- $d = 0 \Rightarrow$ `shiftDown` (S, <0>, 0)

Priority Queues – Summary

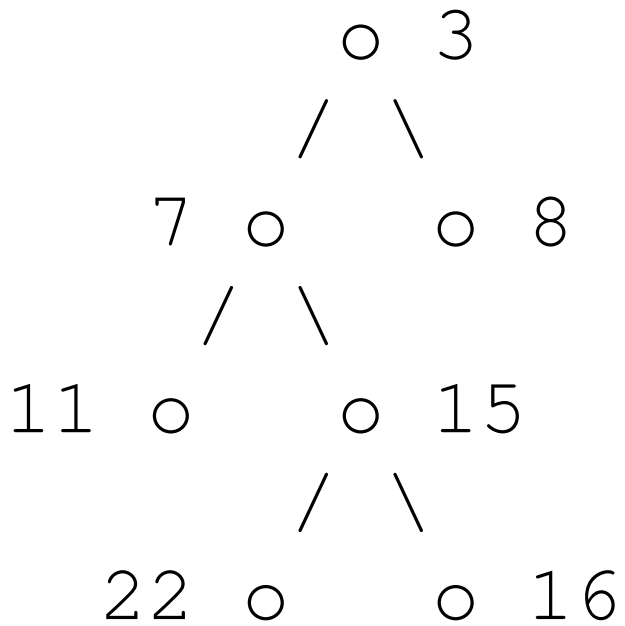
Data. Str.	`findMin`	`deleteMin`	`insert`	`fromSeq`
sorted linked list	$O(1)$	$O(1)$	$O(n)$	$O(n \log n)$
unsorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(n)$
balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$
binary heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$

Meldable Priority Queues

- Priority Queues with an additional *meld* operation
  - Just like the union in BSTs
  - Takes two meldable PQs and returns the union as a meldable PQ
- Implementation uses *leftist heaps*
  - Same work and span as binary heaps for insert, deletemin
  - Meld has $O(\log n + \log m)$ work and span where m and n are the heap sizes

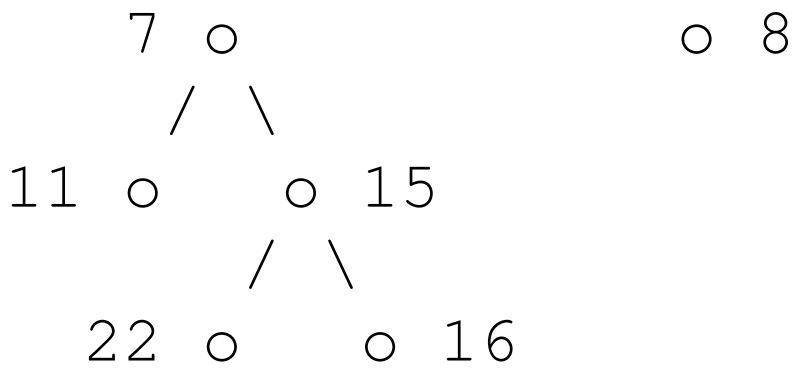
Min Heaps

- Binary tree
- Maintains the heap property
- But does *not* maintain the complete binary tree property
- Here is an example



Min Heaps

- To implement `deleteMin`
  - Remove the root



- We can then use `meld` to union the heaps.

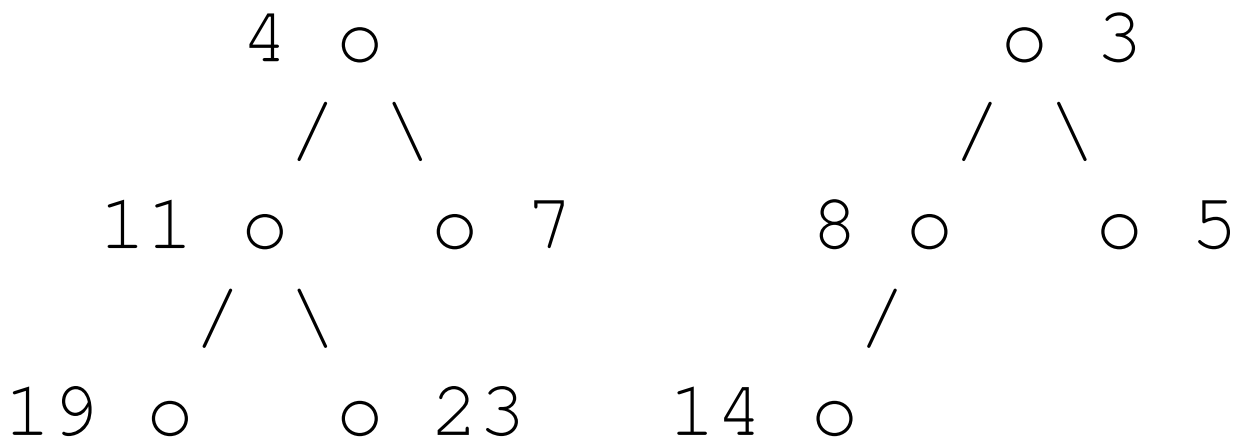
Min Heaps

- To implement `insert`
  - We create a single node heap
  - `meld` it with the original heap
- `fromSeq` is also easy using reduce

```
val pq = Seq.reduce Q.meld Q.empty
                    (Seq.map Q.singleton S)
```

The Meld Operation

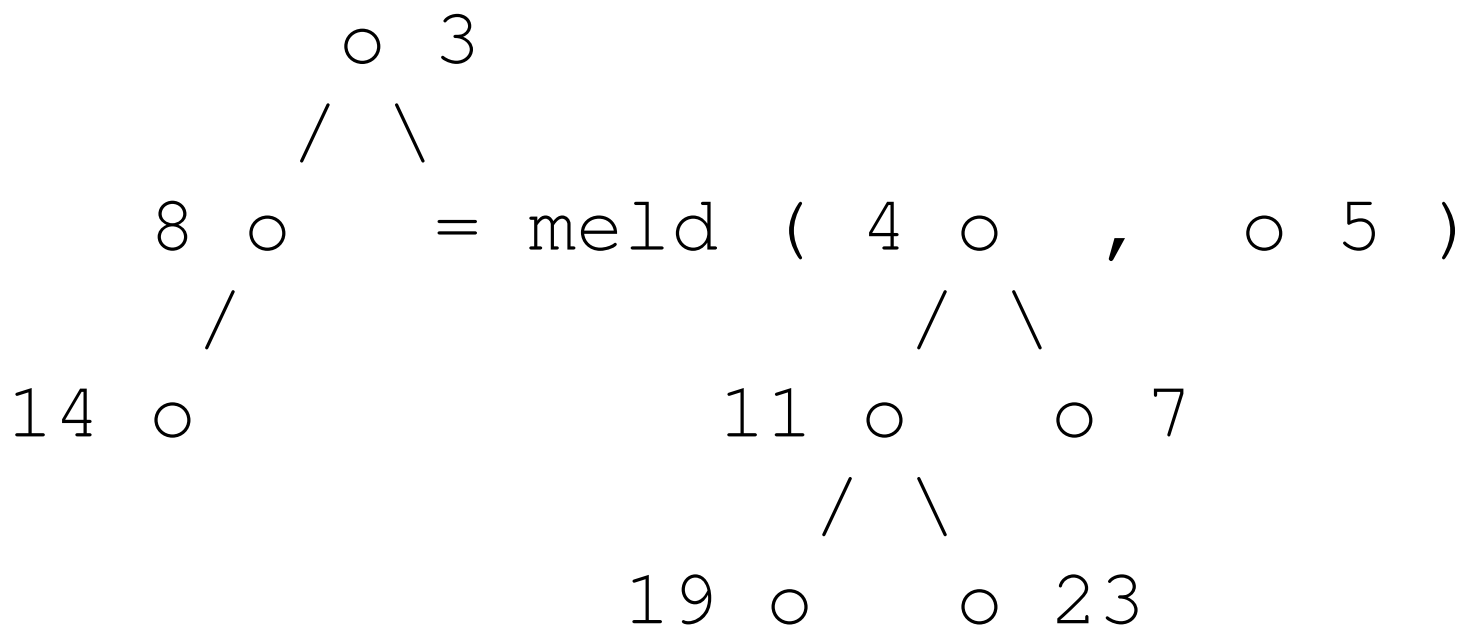
- So we only need the `meld` operation
- Consider



- Which element goes to the root?

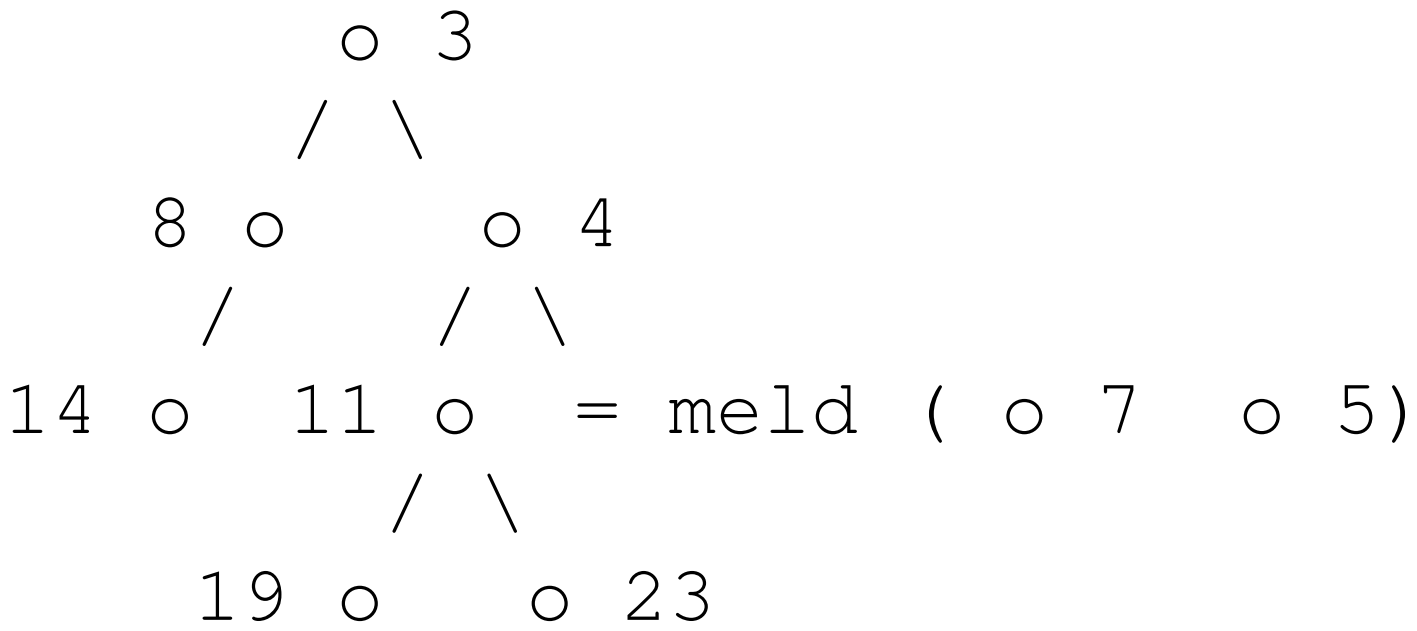
The Meld Operation

- Select the tree with the smaller root and recursively `meld` with one of its children



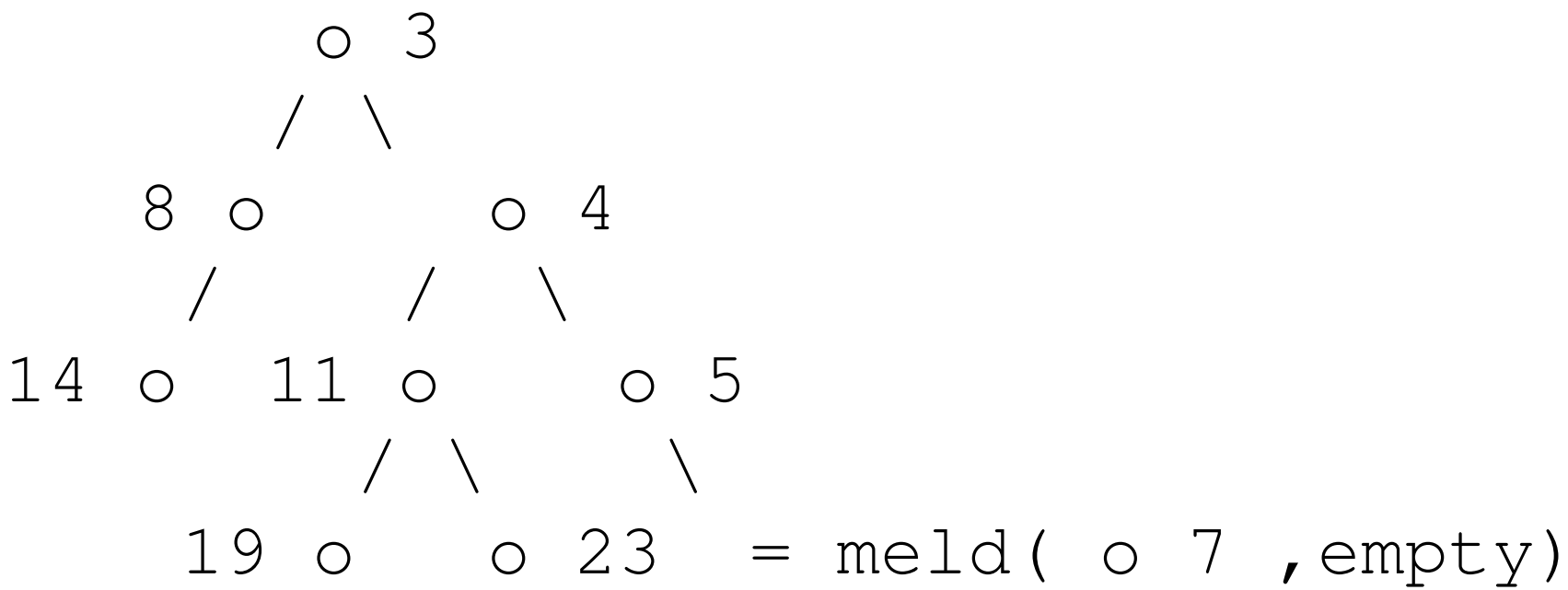
The Meld Operation

- Applying recursively



The Meld Operation

- Applying recursively



- Melding *A* with an empty heap gives *A*

The Meld Operation

```
1  datatype PQ = Leaf | Node of (key × PQ × PQ)

2  fun meld(A, B) =
3    case (A, B) of
4      (_, Leaf) ⇒ A
5    | (Leaf, _) ⇒ B
6    | (Node(k_a, L_a, R_a), Node(k_b, L_b, R_b)) ⇒
7        case Key.compare (k_a, k_b) of
8          LESS ⇒ Node(k_a, L_a, meld(R_a, B))
9        | _ ⇒ Node(k_b, L_b, meld(A, R_b))
```

- Traverses the right spines of the trees
- Could be $\Theta(|A| + |B|)$ in the worst case.

Leftist Heaps

- When melding, keep trees *deeper* on the left.
- Define

$$\text{rank}(x) = \#\text{ of nodes on the right}$$

$$\text{spine of the subtree rooted at } x,$$

- For all nodes, rank can be inductively defined

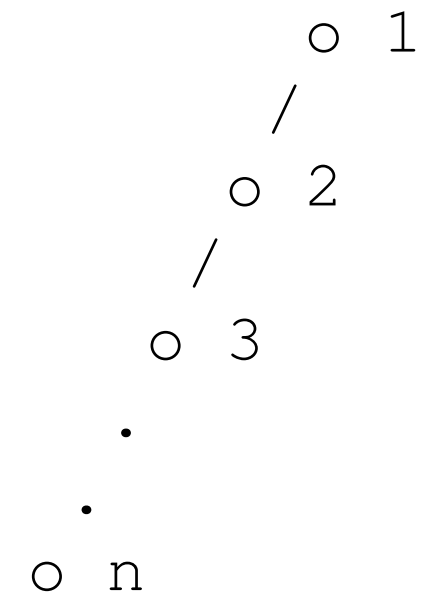
$$\text{rank}(\text{leaf}) = 0$$
$$\text{rank}(\text{node}(\_, \_, R) = 1 + \text{rank}(R)$$

Leftist Property

- For all node x in a leftist heap,

$$\text{rank}(L(x)) \geq \text{rank}(R(x))$$

  - $L(x)$ and $R(x)$ are the left and child children of x
- Allows for



- But this is OK (Why?)

Leftist Heaps

- Most items pile to the left
- Right spine is relatively short!

Lemma

In a leftist heap with n entries, the rank of the root node is at most $\log_2(n+1)$.

Leftist Heaps

```
1  datatype PQ = Leaf | Node of (int × key × PQ × PQ)
2  fun rank Leaf = 0
3    | rank (Node(r, _, _, _)) = r
4  fun makeLeftistNode (v, L, R) =
5    if (rank(L) < rank(R))
6    then Node(1 + rank(L), v, R, L)
7    else Node(1 + rank(R), v, L, R)
```

- Puts lower rank subtree to the right!

Leftist Heaps

```
1  fun meld (A, B) =
2    case (A, B) of
3       (_, Leaf) ⇒ A
4     | (Leaf, _) ⇒ B
5     | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
6         case Key.compare(k_a, k_b) of
7            LESS ⇒ makeLeftistNode (k_a, L_a, meld(R_a, B))
8          | _ ⇒ makeLeftistNode (k_b, L_b, meld(A, R_b))
```

Leftist Heaps

Theorem

If A and B are leftist heaps then

- the `meld`(A, B) algorithm runs in $O(\log(|A|) + \log(|B|))$ work, and
- returns a leftist heap containing the union of A and B.

- Code traverses the right spines, one node at a time
  - so needs at most $\text{rank}(A) + \text{rank}(B)$ steps
  - Each step needs constant work
- `makeLeftistNode` guarantees leftist result

Proving the Lemma

Claim

If a heap has rank r, it contains at least $2^r - 1$ entries.

- $n(r) \equiv$ nodes in the smallest heap of rank r
  - Monotone: if $r' \geq r$, then $n(r') \geq n(r)$
  - $n(0) = 0$
- $\text{rank}(L(x)) \geq \text{rank}(R(x)) = r - 1$

$$\begin{aligned} n(r) &= 1 + n(\text{rank}(L(x))) + n(\text{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

- $n(r) \geq 2^r - 1$

Proving the Lemma

- Apply the claim
- Suppose leftist heap of n nodes has rank r
- $n \geq n(r) \geq 2^r - 1$
- $2^r \leq n + 1 \Rightarrow r \leq \log_2(n+1)$
- Rank of a leftist node of n nodes is at most $\log_2(n+1)$

Summary of Priority Queues

Implementation	`insert`	`findMin`	`deleteMin`	`meld`
(Unsorted) Sequence	$O(n)$	$O(n)$	$O(n)$	$O(m+n)$
Sorted Sequence	$O(n)$	$O(1)$	$O(n)$	$O(m+n)$
Balanced Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log(1+\frac{n}{m}))$
Leftist Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log m + \log n)$