CS 240 – Data Structures and Data Management

Module 2E: Priority Queues - Enriched

A. Hunt A. Jamshidpey O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2023

version 2023–01–16 15:49

Outline

2. Merging heaps
   - More PQ operations
   - Meldable Heaps
   - Binomial Heaps

Outline

2. Merging heaps
   - More PQ operations
   - Meldable Heaps
   - Binomial Heaps

Merging Priority Queues

New operation: *merge*(P_1, P_2)

- Given: two priority queues P_1, P_2 of size n_1 and n_2.
- Want: One priority queue P that contains all their items

Merging Priority Queues

New operation: *merge*(P_1, P_2)

- Given: two priority queues P_1, P_2 of size n_1 and n_2.
- Want: One priority queue P that contains all their items

This will take time $\Omega(\min\{n_1, n_2\})$ if PQs stored as array.
Can we do it *faster* if PQs are stored as trees?

Merging Priority Queues

New operation: *merge*(P_1, P_2)

- Given: two priority queues P_1, P_2 of size n_1 and n_2.
- Want: One priority queue P that contains all their items

This will take time $\Omega(\min\{n_1, n_2\})$ if PQs stored as array.
Can we do it *faster* if PQs are stored as trees?

Three approaches (where $n = n_1 + n_2$):

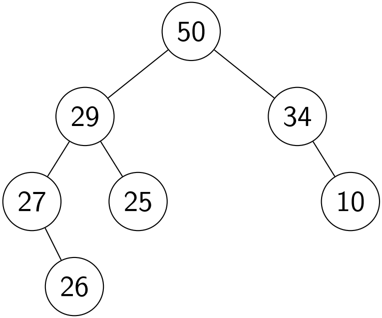
- Merge binary heaps (stored as trees).
  $O(\log^3 n)$ worst-case time (no details)
- Merge *meldable heaps* that have heap-property (but not structural property). $O(\log n)$ *expected* run-time.
- Merge *binomial heaps* that have a different structural property.
  $O(\log n)$ *worst-case* run-time.

Outline

2. Merging heaps
   - More PQ operations
   - Meldable Heaps
   - Binomial Heaps

Meldable Heaps

- Priority queue stored as binary tree
- Heap-order-property: Parent no smaller than child.
- No structural property; any binary tree allowed.
- *Tree-based:* Store nodes and references to *left*/*right*



PQ-operations in Meldable Heaps

Both *insert* and *deleteMax* can be done by *reduction* to *merge*.

P.*insert*(k, v):

- Create a 1-node meldable heap P' that stores (k, v).
- Merge P' with P.

P.*deleteMax*():

- Stash item that is at root.
- Let P_ℓ and P_r be left and right sub-heap of root.
- Update $P \leftarrow merge(P_\ell, P_r)$
- Return stashed item.

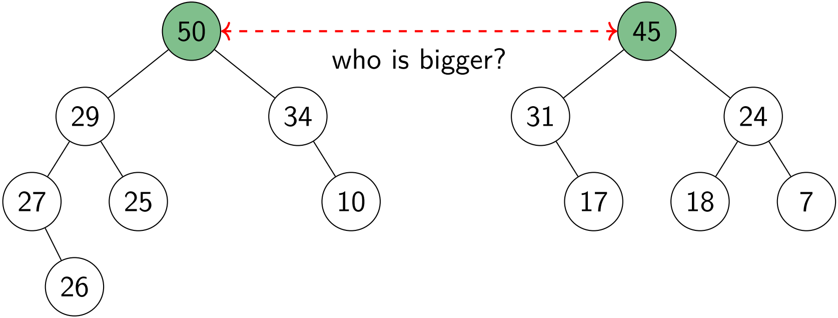
Both operations have run-time $O(merge)$.

Merging Meldable Heaps

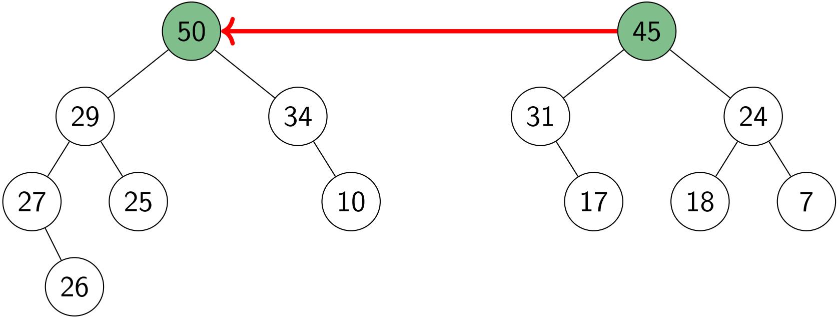
- Idea: Merge heap with smaller root into other one, *randomly* choose into which sub-heap to merge.
- Structural property not maintained

meldableHeap::merge(r_1, r_2)
r_1, r_2: *roots of two heaps (possibly* `NIL`*)*
returns root of merged heap
1. **if** r_1 is `NIL` **return** r_2
2. **if** r_2 is `NIL` **return** r_1
3. **if** $r_1.key < r_2.key$ *swap*(r_1, r_2)
4. `//` `now` r_1 `has max-key and becomes the root.`
5. *randomly* pick one child c of r_1
6. replace subheap at c by *heapMerge*(c, r_2)
7. **return** r_1

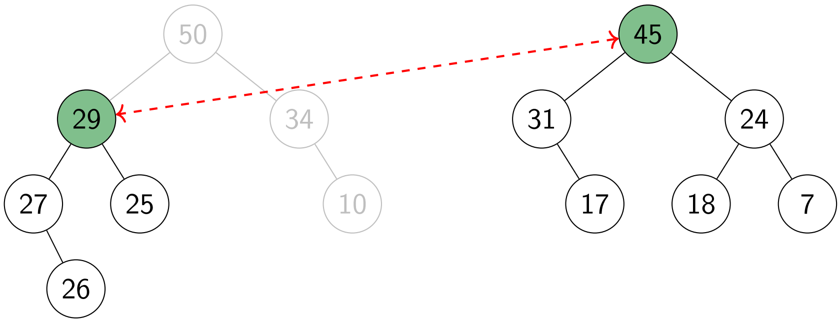
Merge Example



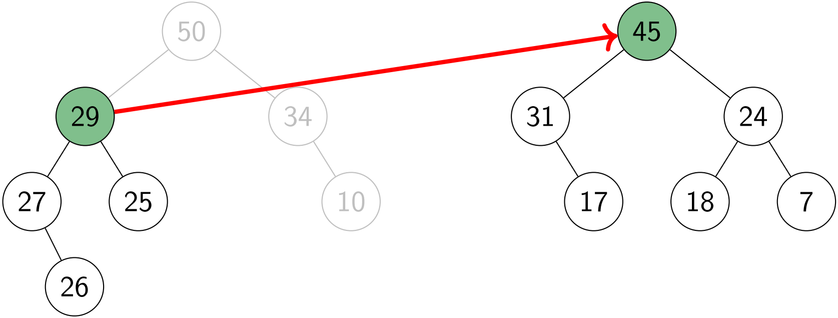
Merge Example



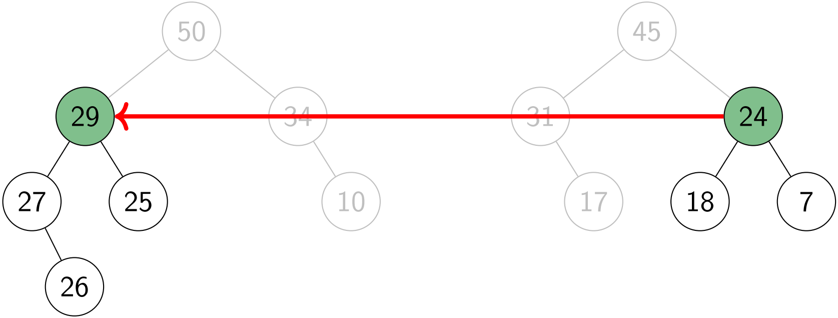
Merge Example



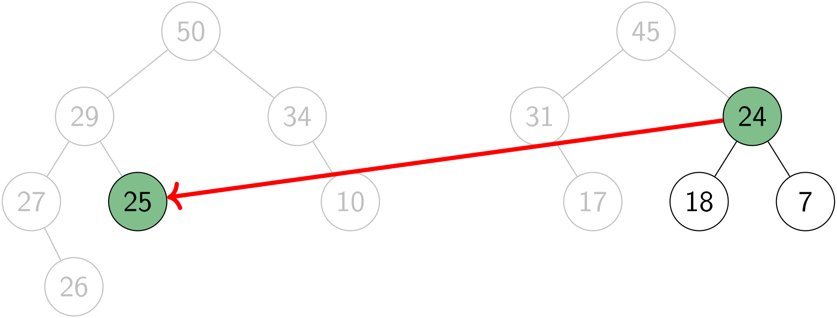
Merge Example



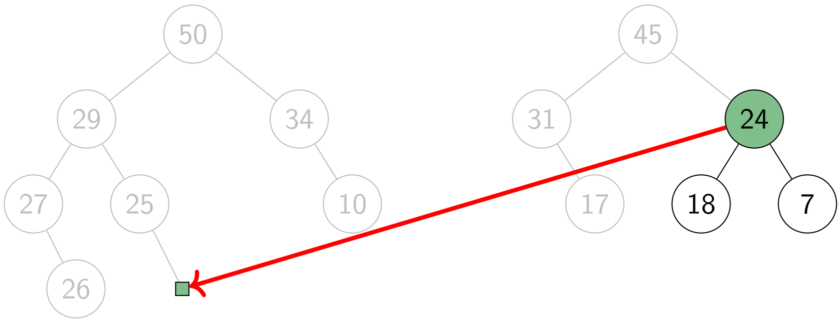
Merge Example



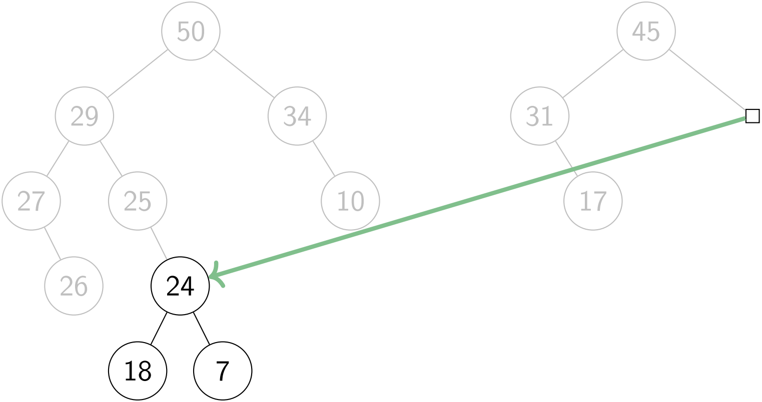
Merge Example



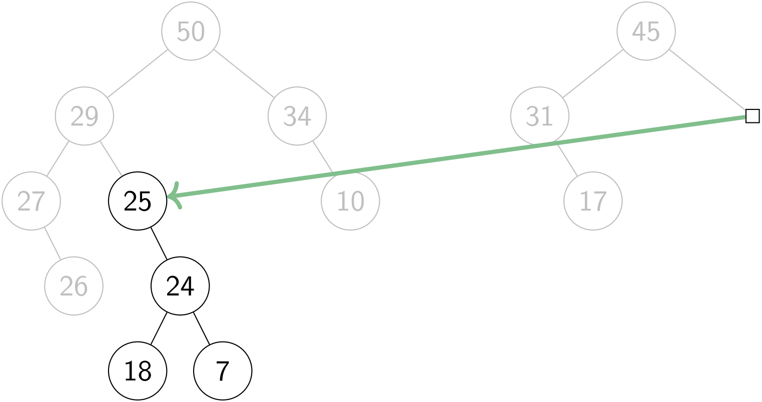
Merge Example



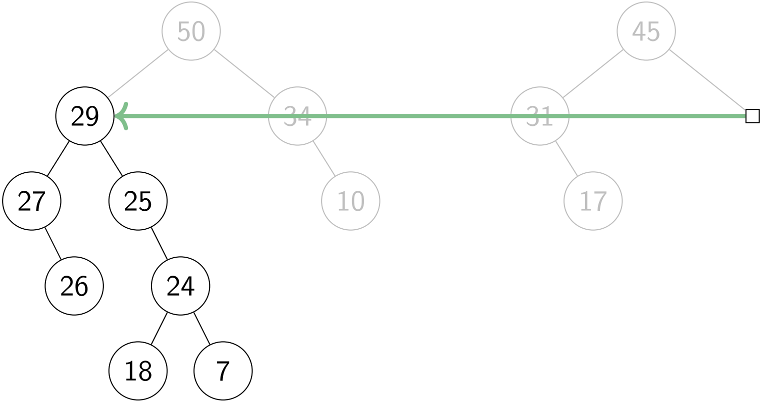
Merge Example



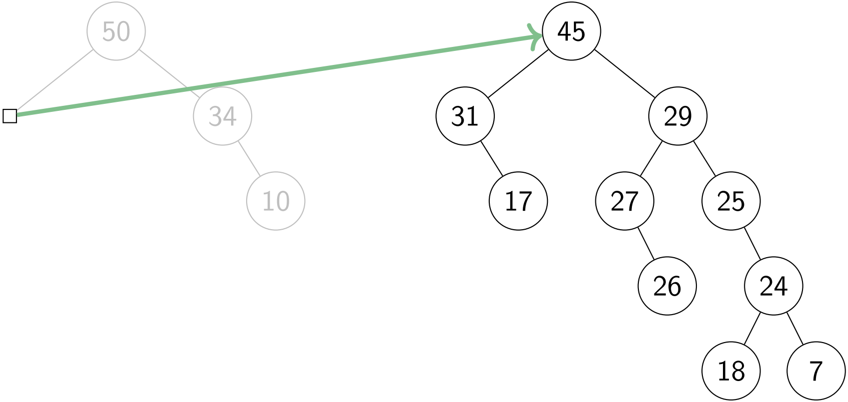
Merge Example



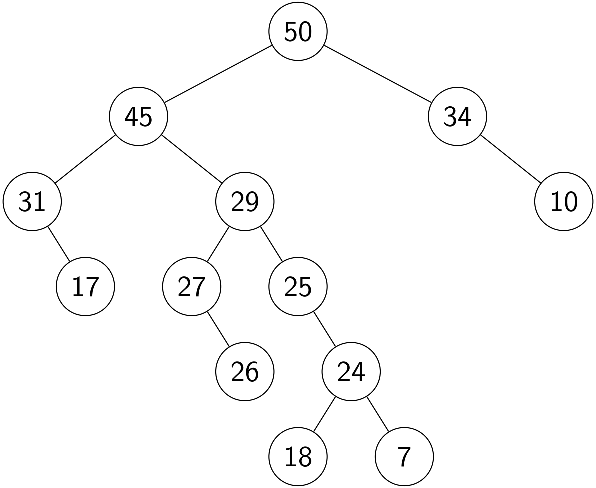
Merge Example



Merge Example



Merge Example



Merging meldable heaps

Run-time? Not more than two *random downward walks* in a binary tree.

Let $T(n)$ = expected length of a random downward walk.

Theorem: $T(n) \in O(\log n)$.
Proof:

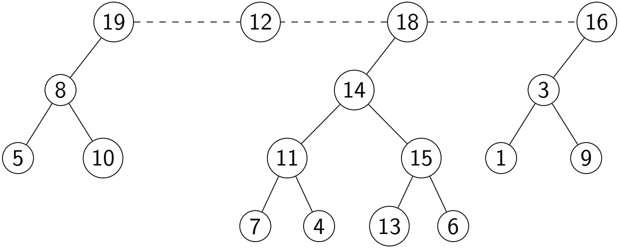
So *merge* (and also *insert* and *deleteMax*) takes $O(\log n)$ expected time.

Outline

2 Merging heaps
- More PQ operations
- Meldable Heaps
- Binomial Heaps

Binomial Heaps

Very different structure from binary heaps and meldable heaps:



- List L of binary trees.
- Each binary tree is a **flagged tree**:
  Complete binary tree T plus root r that has T as left subtree
  - Flagged tree of height h has 2^h nodes.
  - So $h \leq \log n$ for all flagged trees.
- Order-property: Nodes in *left* subtree have no-smaller keys.
  (No restrictions on nodes in the right subtree.)

Binomial Heap Operations

- *insert*: Reduce to *merge* as before.
- *findMax*:
  - At each flag tree, root contains the maximum.
  - Search roots in $L \Rightarrow O(|L|)$ time.
- We want L to be short.

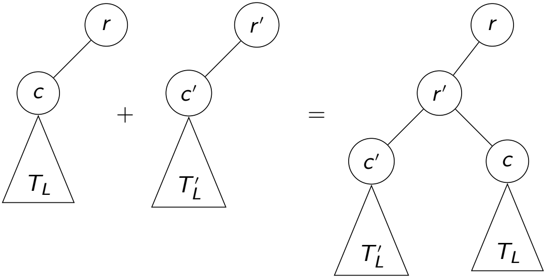
Binomial Heap Operations

- *insert*: Reduce to *merge* as before.
- *findMax*:
  - At each flag tree, root contains the maximum.
  - Search roots in $L \Rightarrow O(|L|)$ time.
- We want L to be short.

- **Proper binomial heap:** No two flagged trees have the same height.
- **Observation:** A proper binomial heap has $|L| \leq \log n + 1$.
  - The flagged tree of largest height h has $h \leq \log n$.
  - Can have only one flagged tree of each height in $\{0, \ldots, h\}$.

Making Binomial Heaps Proper

- Goal: Given a binomial heap, make it proper.
- Need subroutine: combine two flagged trees of the same height. This can be done in constant time. If $r.key \geq r'.key$:

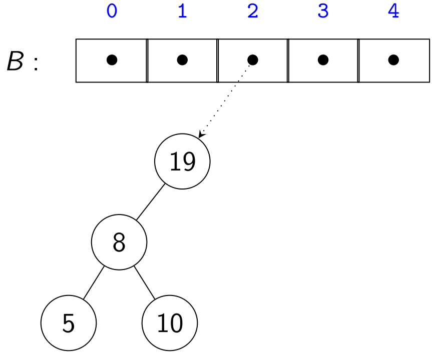


- Idea: Do this whenever two flagged trees have same height.
- Run-time to make proper: $O(|L| + \log n)$ if implemented suitably.

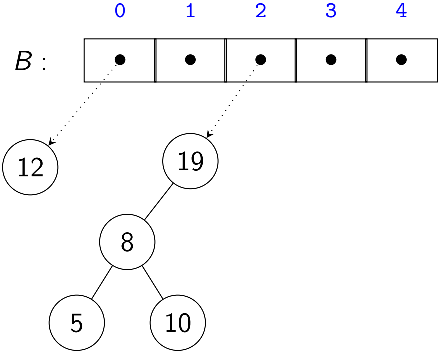
Making Binomial Heaps Proper

	0	1	2	3	4
B :	•	•	•	•	•

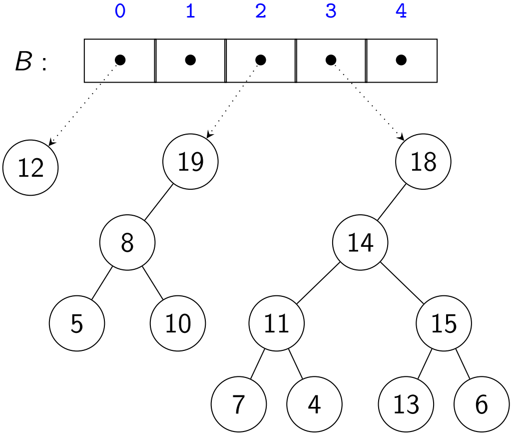
Making Binomial Heaps Proper



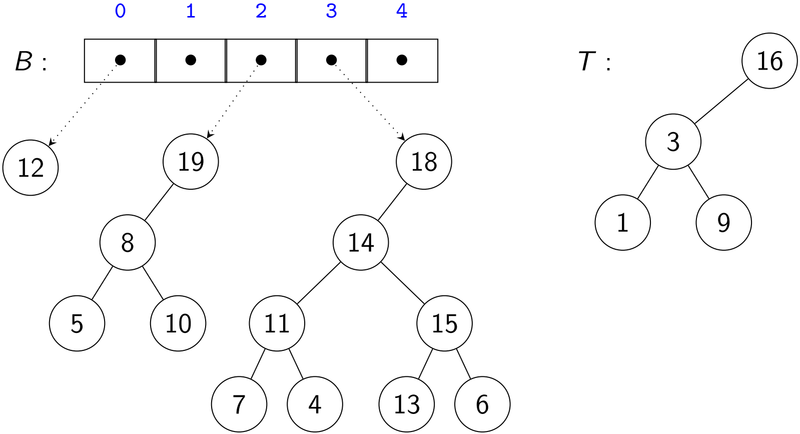
Making Binomial Heaps Proper



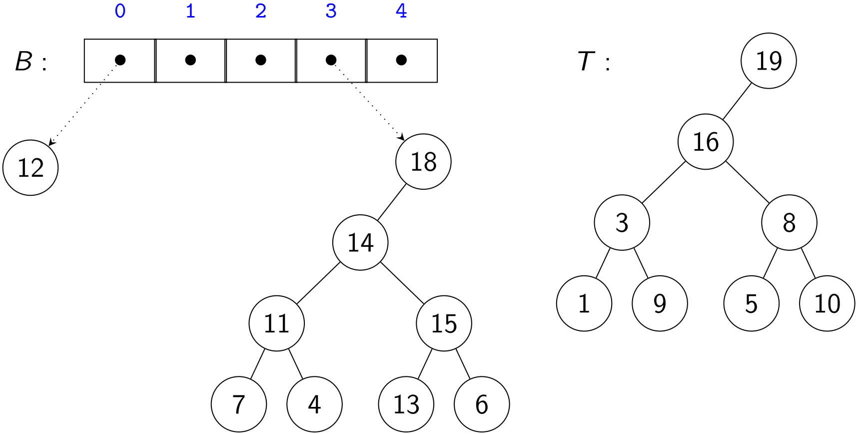
Making Binomial Heaps Proper



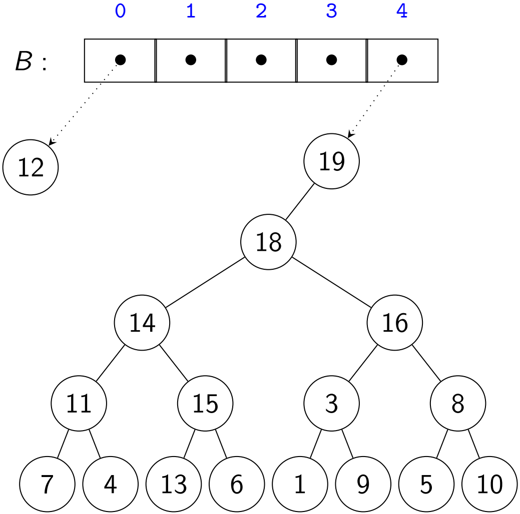
Making Binomial Heaps Proper



Making Binomial Heaps Proper



Making Binomial Heaps Proper



Making Binomial Heaps Proper

binomialHeap::makeProper()

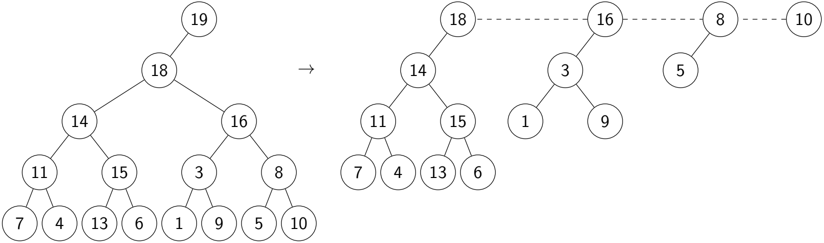
```
n ← size of the binomial heap
compute ℓ ← ⌊log n⌋
B ← array of size ℓ + 1, initialized all-NIL
L ← list of flagged trees
while L is non-empty do
    T ← L.pop(), h ← T.height
    while T' ← B[h] is not NIL do
        if T.root.key < T'.root.key do swap T and T'
        // combine T with T'
        T'.right ← T.left, T.left ← T', T.height ← h+1
        B[h] ← NIL, h++
    B[h] ← T
// copy B back to list
for (h = 0; h ≤ ℓ; h++) do
    if B[h] ≠ NIL do L.append(B[h])
```

Binomial Heap Operations

- **Idea:** Make binomial heap proper after *every* opration.
  - $\Rightarrow$ L always has length $O(\log n)$
  - $\Rightarrow$ Each *makeProper* takes $O(\log n)$ time
- *findMax*: $O(\log n)$ worst-case time.
- *merge*: $O(\log n)$ worst-case time.
  - Concatenate the two lists.
  - Call *makeProper*.
- *insert*: $O(\log n)$ worst-case time via *merge*.
- *deleteMax*?

deleteMax in a binomial heap

- Search for maximum among roots, say it is in tree T
- Split $T \setminus \{\text{root}\}$ into into flagged trees $T_1, \ldots, T_k$



- Merge $L \setminus T$ with $\{T_1, \ldots, T_k\}$
- Have $k \leq \log n \Rightarrow O(\log n)$ worst-case time.

Summary: All operations have $O(\log n)$ worst-case run-time.