CS 240E – Data Structures and Data Management (Enriched)

Module 2: Priority Queues

Therese Biedl
Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version 2025–01–09 14:04

Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- Binomial Heaps

ADT Priority Queue

Priority Queue generalizes both ADT Stack and ADT Queue.

It is a collection of items (each having a **priority** or **key**) with operations

- *insert*: inserting an item tagged with a priority
- *delete-max*: removing and returning an item of *highest* priority.

We can have extra operations: *size*, *is-empty*, and *get-max*

This is a **maximum-oriented** priority queue. A **minimum-oriented** priority queue replaces operation *delete-max* by *delete-min*.

Applications:

- How would you simulate a stack with a priority queue?
- How would you simulate a queue with a priority queue?
- Other applications: typical todo-list, simulation systems, sorting

Using a Priority Queue to Sort

```
PQ-Sort($A[0..n-1]$)
1.  initialize $PQ$ to an empty priority queue
2.  for $i \leftarrow 0$ to $n-1$ do
3.      $PQ$.insert(an item with priority $A[i]$)
4.  for $i \leftarrow n-1$ down to $0$ do
5.      $A[i] \leftarrow$ priority of $PQ$.delete-max()
```

- Note: Run-time depends on how we implement the priority queue.
- Sometimes written as: $O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$

With two easy (but slow) realizations of priority queues:

- **Unsorted** array or list ($\rightsquigarrow$ *selection-sort*)
- **Sorted** array or list ($\rightsquigarrow$ *insertion-sort*)

Using a Priority Queue to Sort

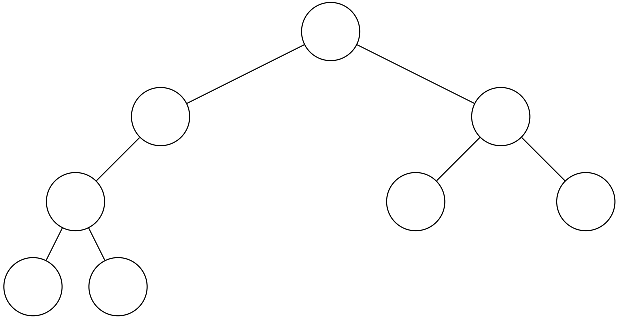
```
PQ-Sort($A[0..n-1]$)
    initialize $PQ$ to an empty priority queue
    for $i \leftarrow 0$ to $n-1$ do
        $PQ$.insert(an item with priority $A[i]$)
    for $i \leftarrow n-1$ down to $0$ do
        $A[i] \leftarrow$ priority of $PQ$.delete-max()
```

- Note: Run-time depends on how we implement the priority queue.
- Sometimes written as: $O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$

With two easy (but slow) realizations of priority queues:

- **Unsorted** array or list ($\rightsquigarrow$ *selection-sort*)
- **Sorted** array or list ($\rightsquigarrow$ *insertion-sort*)

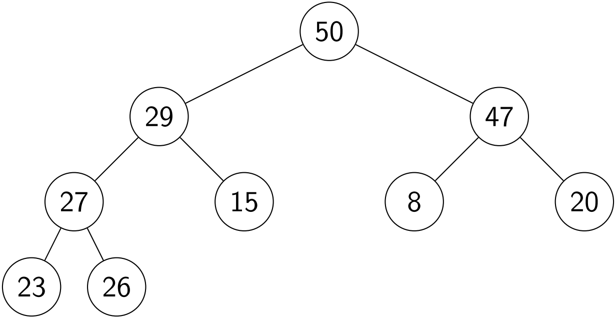
Better Realization: Binary Heap



Binary tree with

1. structural property and

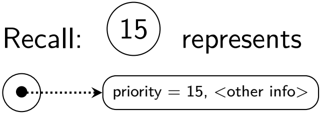
Better Realization: Binary Heap



Binary tree with

1. structural property and
2. heap-order property.

Recall:



Heaps – Definition

A **heap** is a binary tree with the following two properties:

1. **Structural Property:** All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified*.
2. **Heap-order Property:** For any node i, the key of the parent of i is larger than or equal to key of i.

Heaps – Definition

A **heap** is a binary tree with the following two properties:

1. **Structural Property:** All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified*.
2. **Heap-order Property:** For any node i, the key of the parent of i is larger than or equal to key of i.

The full name for this is *max-oriented binary heap*.

Heaps – Definition

A **heap** is a binary tree with the following two properties:

1. **Structural Property:** All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are *left-justified*.
2. **Heap-order Property:** For any node i, the key of the parent of i is larger than or equal to key of i.

The full name for this is *max-oriented binary heap*.

Lemma: The height of a heap with n nodes is $\Theta(\log n)$.

Storing Heaps in Arrays

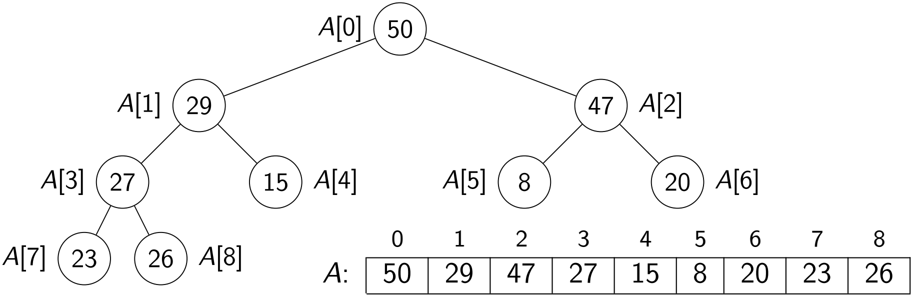
Heaps should *not* be stored as binary trees!

Let H be a heap of n items and let A be an array of size n. Store root in $A[0]$ and continue with elements *level-by-level* from top to bottom, in each level left-to-right.

Storing Heaps in Arrays

Heaps should *not* be stored as binary trees!

Let H be a heap of n items and let A be an array of size n. Store root in $A[0]$ and continue with elements *level-by-level* from top to bottom, in each level left-to-right.



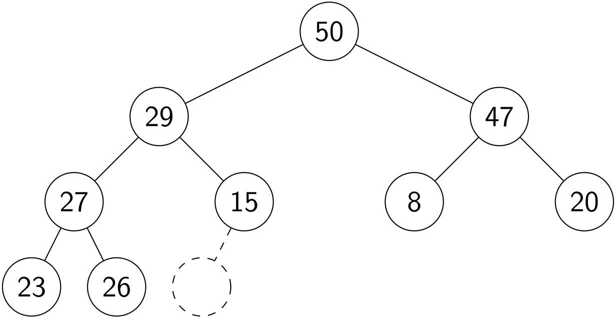
	0	1	2	3	4	5	6	7	8
A:	50	29	47	27	15	8	20	23	26

Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- Binomial Heaps

insert in Heaps

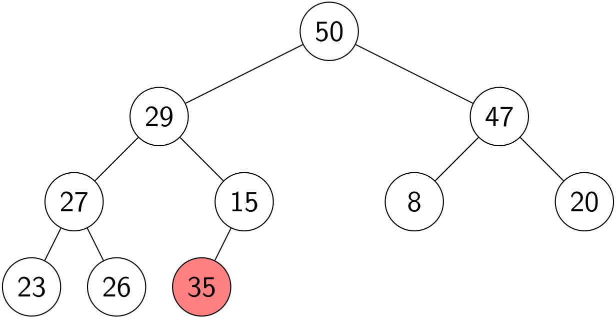
insert(35):



- By structural property: no choice where the new node can go.

insert in Heaps

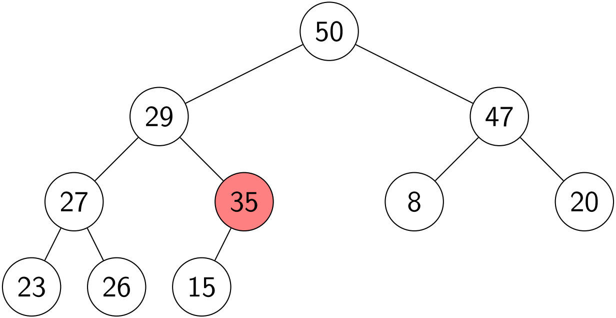
insert(35):



- By structural property: no choice where the new node can go.
- This may or may not lead to heap-order violations.

insert in Heaps

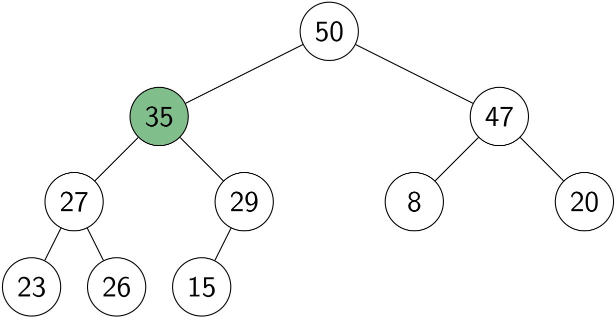
insert(35):



- By structural property: no choice where the new node can go.
- This may or may not lead to heap-order violations.
- Fix violations by "bubbling up" in the tree.

insert in Heaps

insert(35):



- By structural property: no choice where the new node can go.
- This may or may not lead to heap-order violations.
- Fix violations by “bubbling up” in the tree.

insert in Heaps

- Place the new key at the first free leaf
- Use *fix-up* to restore heap-order.

```
insert($x$)
    $\ell \leftarrow$ last()+1
    $A[\ell] \leftarrow x$          // assume dynamic array used
    increase size     // size:  stored by PQ
    fix-up($A, \ell$)
```

insert in Heaps

- Place the new key at the first free leaf
- Use *fix-up* to restore heap-order.

```
insert($x$)
    $\ell \leftarrow$ last()$+1$
    $A[\ell] \leftarrow x$          // assume dynamic array used
    increase size        // size:  stored by PQ
    fix-up($A, \ell$)
```

```
fix-up($A, i$)
i: an index corresponding to a node of the heap
    while parent($i$) exists and $A[parent(i)].key < A[i].key$ do
        swap $A[i]$ and $A[parent(i)]$
        $i \leftarrow parent(i)$
```

insert in Heaps

- Place the new key at the first free leaf
- Use *fix-up* to restore heap-order.

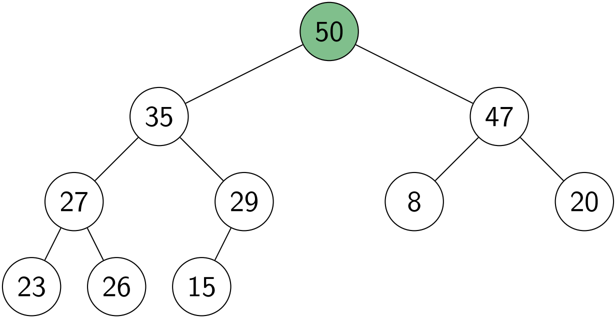
```
insert($x$)
    $\ell \leftarrow last()+1$
    $A[\ell] \leftarrow x$        // assume dynamic array used
    increase size        // size:  stored by PQ
    fix-up($A, \ell$)
```

```
fix-up($A, i$)
i: an index corresponding to a node of the heap
    while $parent(i)$ exists and $A[parent(i)].key < A[i].key$ do
        swap $A[i]$ and $A[parent(i)]$
        $i \leftarrow parent(i)$
```

Time: $O(\text{height of heap}) = O(\log n)$ (and this is tight).

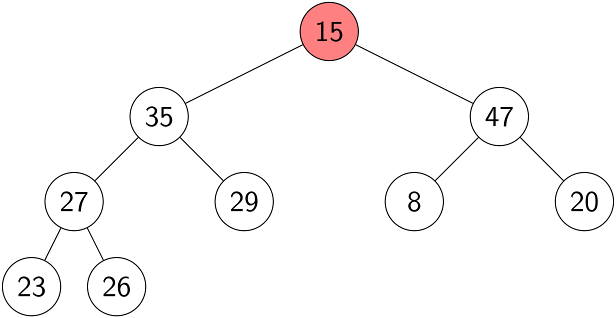
(Correctness may seem obvious, but is actually non-trivial.)

delete-max in Heaps



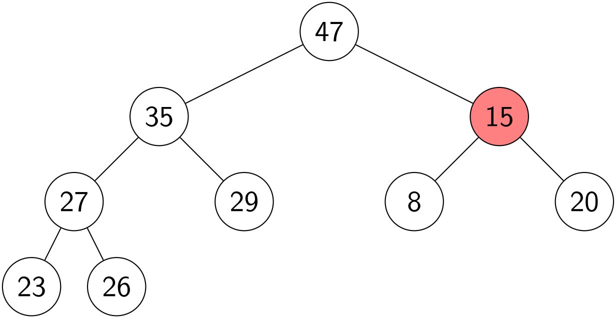
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

delete-max in Heaps



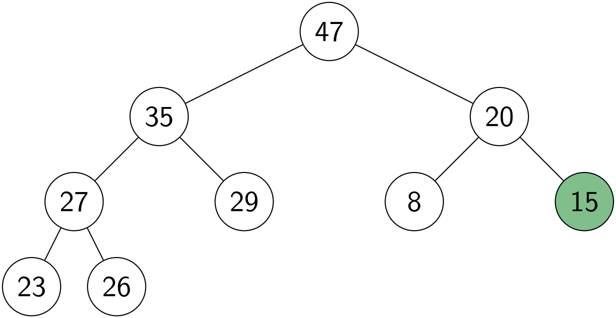
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

delete-max in Heaps



- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

delete-max in Heaps



- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

delete-max in Heaps

delete-max in Heaps

```
fix-down($A, i$)
A: an array that stores a heap of size $n$
i: an index corresponding to a node of the heap
while $i$ is not a leaf do
    $j \leftarrow$ left child of $i$        // find child with larger key
    if ($i$ has right child and $A$[right child of $i$].key $> A[j]$.key)
        $j \leftarrow$ right child of $i$
    if $A[i]$.key $\geq A[j]$.key break
    swap $A[j]$ and $A[i]$
    $i \leftarrow j$
```

Time: $O(\text{height of heap}) = O(\log n)$ (and this is tight).

Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- Binomial Heaps

Sorting using heaps

- Recall: Any priority queue can be used to *sort* in time

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$$

- Using the binary-heaps implementation of PQs, we obtain:

```
PQ-sort-with-heaps(A)
initialize H to an empty heap
for $i \leftarrow 0$ to $n-1$ do
    $H.insert(A[i])$
for $i \leftarrow n-1$ down to $0$ do
    $A[i] \leftarrow H.delete\text{-}max()$
```

Sorting using heaps

- Recall: Any priority queue can be used to *sort* in time

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$$

- Using the binary-heaps implementation of PQs, we obtain:

```
PQ-sort-with-heaps(A)
initialize H to an empty heap
for i ← 0 to n − 1 do
    H.insert(A[i])
for i ← n − 1 down to 0 do
    A[i] ← H.delete-max()
```

- both operations run in $O(\log n)$ time for heaps

⇝ *PQ-sort* using heaps takes $O(n \log n)$ time (and this is tight).

Sorting using heaps

- Recall: Any priority queue can be used to *sort* in time

$$O(\textit{initialization} + n \cdot \textit{insert} + n \cdot \textit{delete-max})$$

- Using the binary-heaps implementation of PQs, we obtain:

```
PQ-sort-with-heaps(A)
initialize H to an empty heap
for i ← 0 to n − 1 do
    H.insert(A[i])
for i ← n − 1 down to 0 do
    A[i] ← H.delete-max()
```

- both operations run in $O(\log n)$ time for heaps

⇝ *PQ-sort* using heaps takes $O(n \log n)$ time (and this is tight).

- Can improve this with two simple tricks → **heap-sort**
  1. Can use the same array for input and heap. ⇝ *$O(1)$ auxiliary space!*
  2. Heaps can be built faster if we know all input in advance.

Building Heaps with *fix-down*

Problem: Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

Building Heaps with *fix-down*

Problem: Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 2: Using *fix-downs* instead:

```
heapify(A)
A: an array
    $n \leftarrow A.size()$
    for $i \leftarrow parent(last())$ downto $root()$ do
        $fix\text{-}down(A, i, n)$
```

Building Heaps with *fix-down*

Problem: Given n items all at once (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 2: Using *fix-downs* instead:

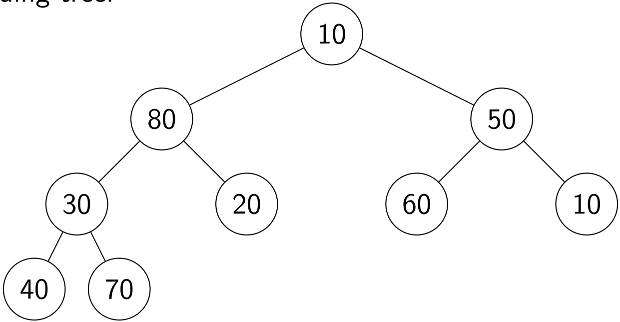
```
heapify(A)
A: an array
    $n \leftarrow A.size()$
    for $i \leftarrow parent(last())$ downto $root()$ do
        $fix\text{-}down(A, i, n)$
```

Show: *heapify* has run-time $\Theta(n)$.

heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	30	20	60	10	40	70

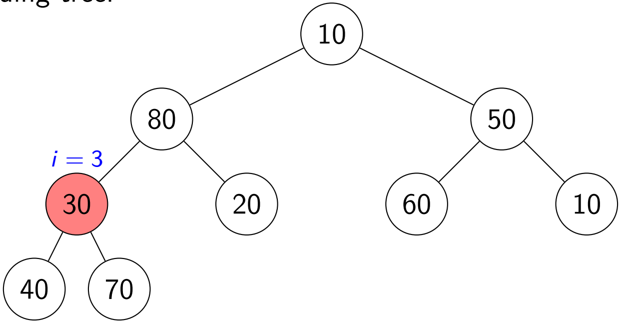
Corresponding tree:



heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	30	20	60	10	40	70

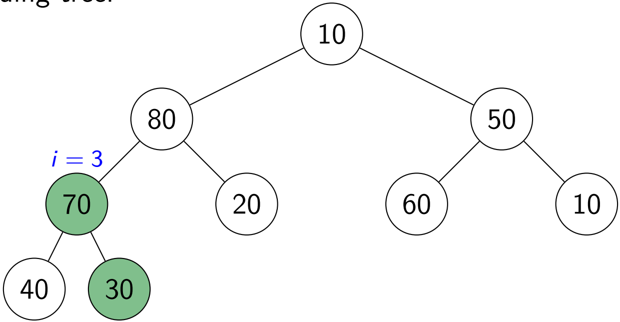
Corresponding tree:



heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	70	20	60	10	40	30

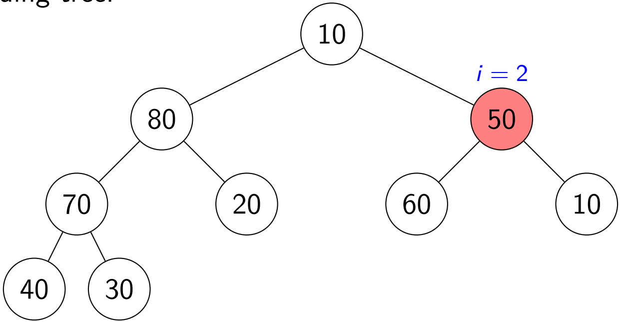
Corresponding tree:



heapify example

	0	1	2	3	4	5	6	7	8
A :	10	80	50	70	20	60	10	40	30

Corresponding tree:

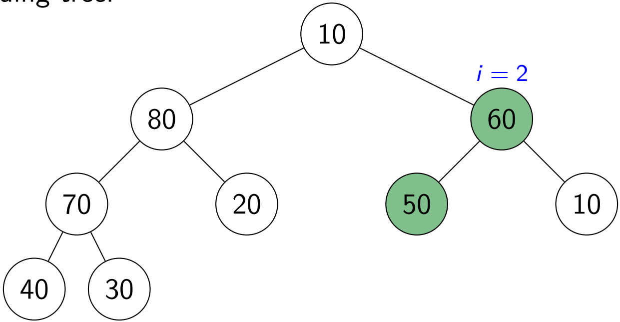


heapify example

A :

0	1	2	3	4	5	6	7	8
10	80	60	70	20	50	10	40	30

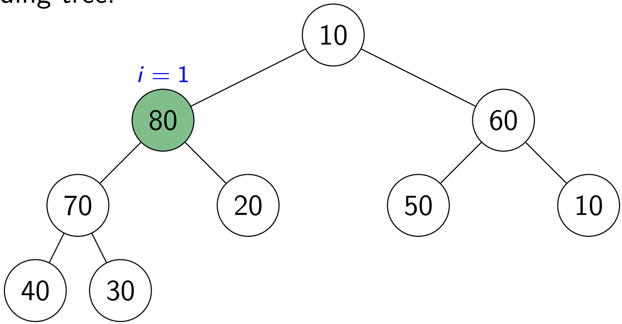
Corresponding tree:



heapify example

	0	1	2	3	4	6	7	8
A :	10	80	60	70	20	10	40	30

Corresponding tree:

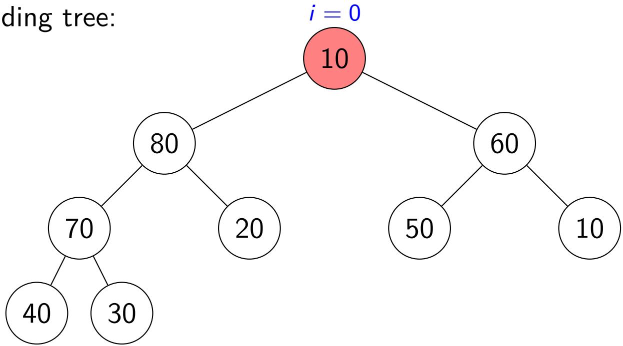


heapify example

A :

0	1	2	3	4	6	7	8
10	80	60	70	20	10	40	30

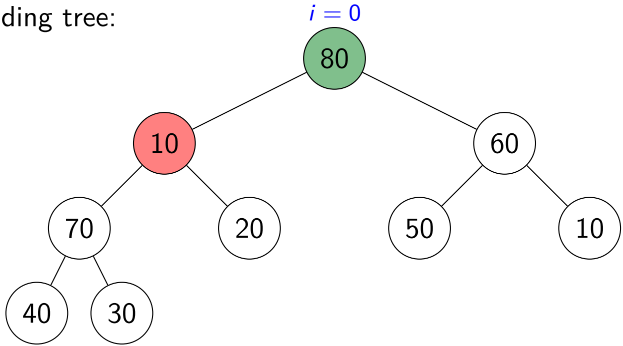
Corresponding tree:



heapify example

	0	1	2	3	4	6	7	8
A :	80	10	60	70	20	10	40	30

Corresponding tree:

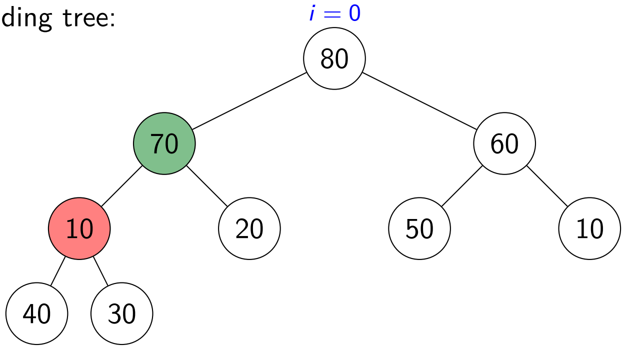


heapify example

A :

0	1	2	3	4	6	7	8
80	70	60	10	20	10	40	30

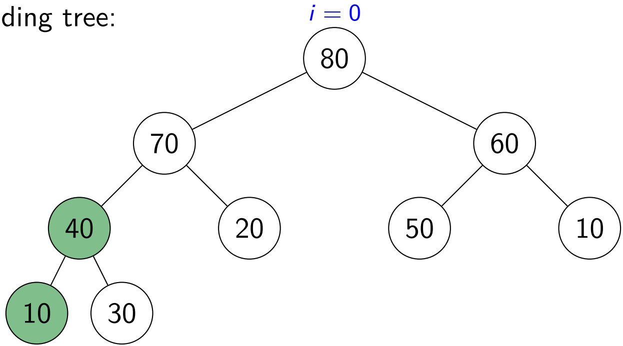
Corresponding tree:



heapify example

	0	2	3	4	6	7	8
A :	80	60	40	20	10	10	30

Corresponding tree:



heapify run-time: proof

Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- Binomial Heaps

More Priority Queues Operations

Binary Heaps are a good realization for *insert* and *delete-max*.
What if we want *more* operations for a priority queue (PQ)?

increase-key(v, k), *decrease-key*(v, k)

- Change key of node v to k if k is bigger/smaller than v.*key*
- Easy to do with *fix-up*/*fix-down*

More Priority Queues Operations

Binary Heaps are a good realization for *insert* and *delete-max*.
What if we want *more* operations for a priority queue (PQ)?

increase-key(v, k), *decrease-key*(v, k)

- Change key of node v to k if k is bigger/smaller than v.*key*
- Easy to do with *fix-up*/*fix-down*

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

More Priority Queues Operations

Binary Heaps are a good realization for *insert* and *delete-max*.
What if we want *more* operations for a priority queue (PQ)?

increase-key(v, k), *decrease-key*(v, k)

- Change key of node v to k if k is bigger/smaller than v.*key*
- Easy to do with *fix-up*/*fix-down*

merge(P_1, P_2)

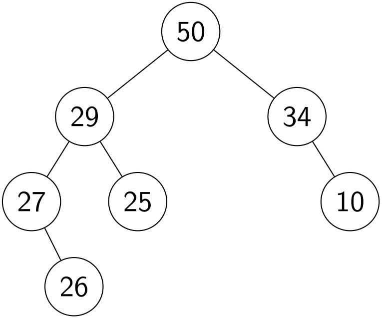
- Given: two priority queues P_1, P_2 of size n_1 and n_2.
- Want: One priority queue P that contains all their items
- **Outlook:** Three approaches (where $n = n_1 + n_2$):
  - Merge binary heaps. $O(\log^3 n)$ worst-case time (no details)
  - **Meldable heaps**: heap-order-property but no structural property. $O(\log n)$ *expected* run-time for all operations.
  - **Binomial heaps**: different structural and order property. $O(\log n)$ *worst-case* run-time for all operations.

Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- Binomial Heaps

Meldable Heaps

- Priority queue stored as a binary tree
- Heap-order-property: Parent no smaller than child.
- No structural property; any binary tree is allowed.



- *Tree-based:* Store items at nodes with references to *left*/*right* child

  (Array-based implementations must use $\Omega(n)$ time for merge—why?)

PQ-operations in Meldable Heaps

Both *insert* and *delete-max* can be done by *reduction* to *merge*.

$P.\mathit{insert}(x)$:

- Create a 1-node meldable heap P' that stores x.
- Merge P' with P.

$P.\mathit{delete\text{-}max}()$:

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

Both operations have run-time $O(\mathit{merge})$.

Merging Meldable Heaps

- Idea: Merge heap with smaller root into other one, *randomly* choose into which sub-heap to merge.

```
meldableHeap::merge($r_1, r_2$)
$r_1, r_2$: roots of two heaps (possibly NULL)
returns root of merged heap
    if $r_1$ is NULL return $r_2$
    if $r_2$ is NULL  return $r_1$
    if $r_1.key < r_2.key$   swap($r_1, r_2$)
    // now $r_1$ has max-key and becomes the root.
    randomly pick one child $c$ of $r_1$
    replace subheap at $c$ by merge($c, r_2$)
    return $r_1$
```

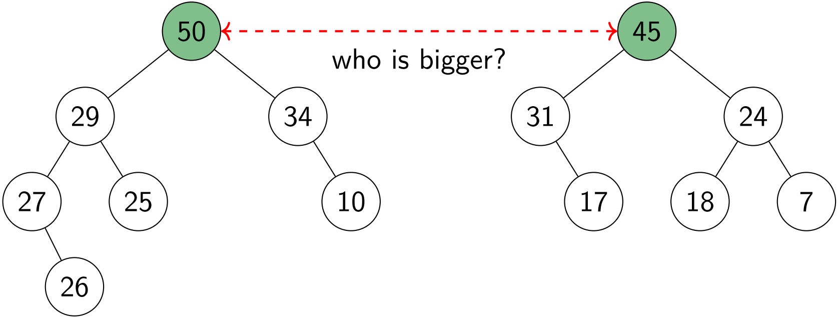
Merging Meldable Heaps

- Idea: Merge heap with smaller root into other one, *randomly* choose into which sub-heap to merge.

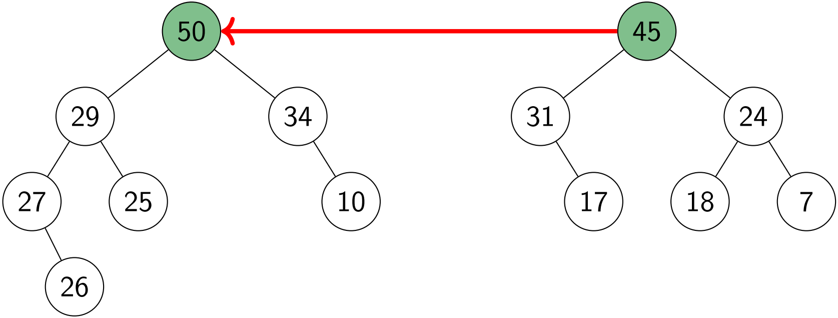
```
meldableHeap::merge($r_1, r_2$)
$r_1, r_2$: roots of two heaps (possibly NULL)
returns root of merged heap
if $r_1$ is NULL return $r_2$
if $r_2$ is NULL return $r_1$
if $r_1.key < r_2.key$   swap($r_1, r_2$)
// now $r_1$ has max-key and becomes the root.
randomly pick one child $c$ of $r_1$
replace subheap at $c$ by merge($c, r_2$)
return $r_1$
```

We will see: The **expected run-time** is $O(\log n)$.

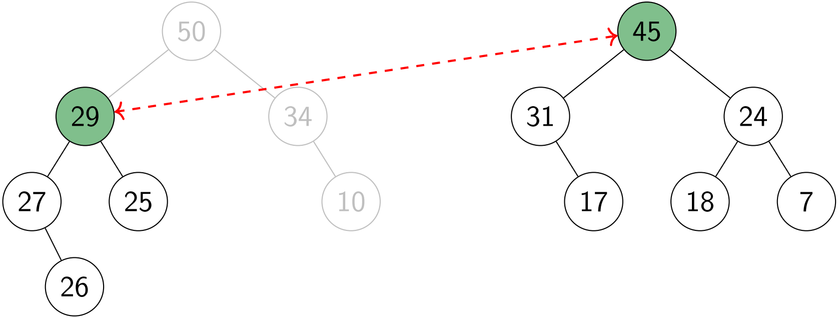
Merge Example



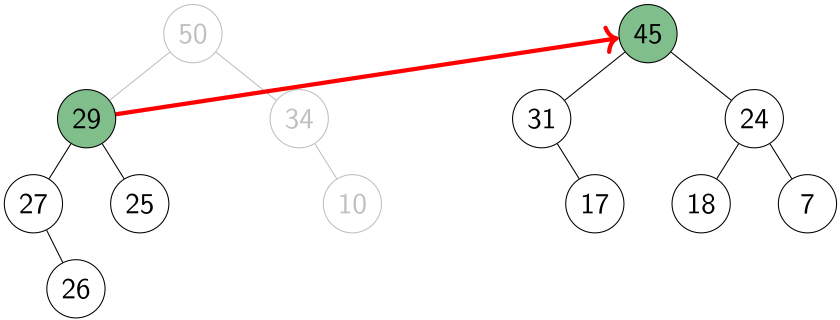
Merge Example



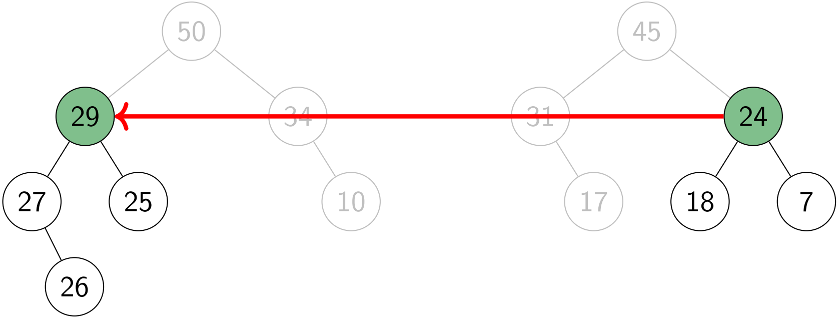
Merge Example



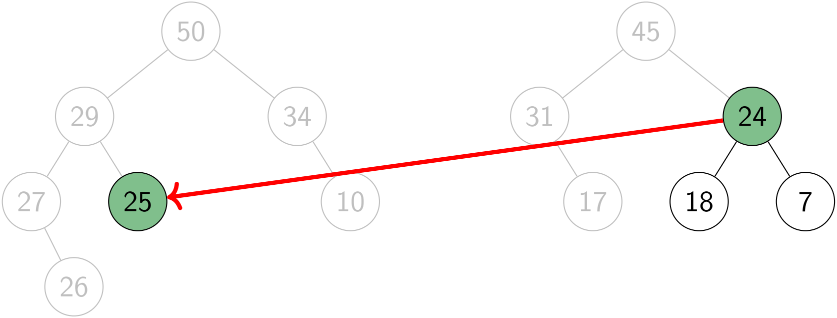
Merge Example



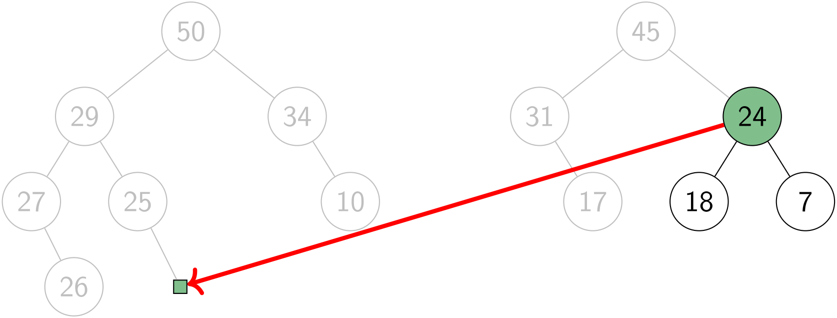
Merge Example



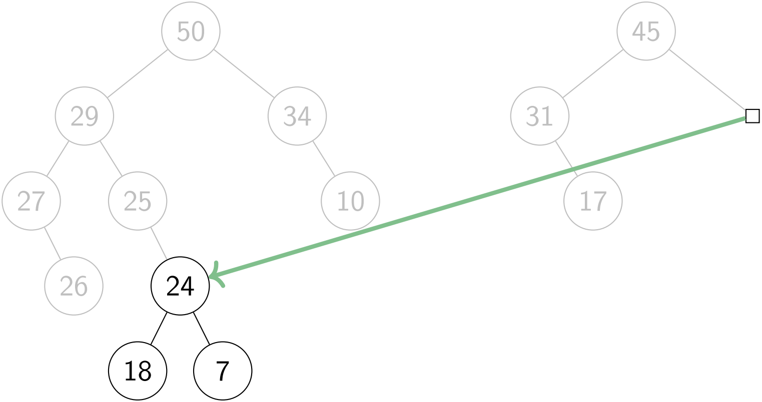
Merge Example



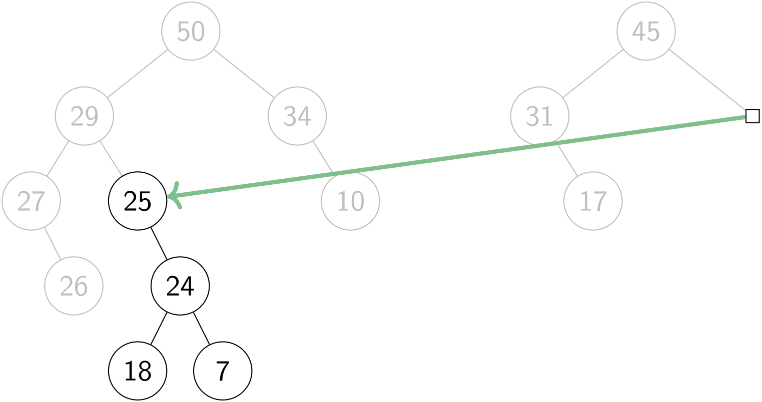
Merge Example



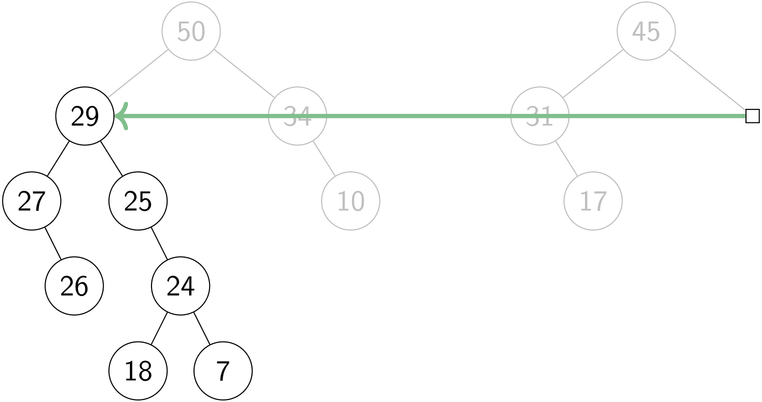
Merge Example



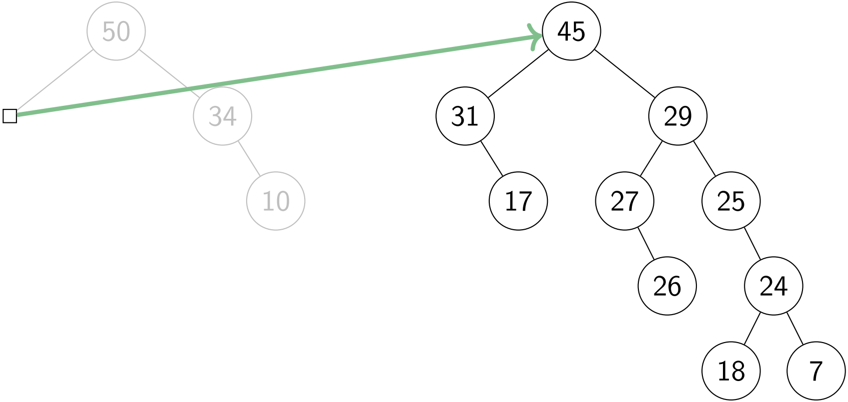
Merge Example



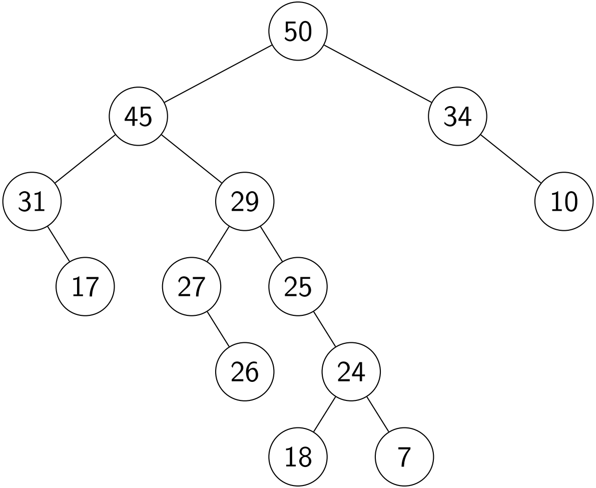
Merge Example



Merge Example



Merge Example



Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- Binomial Heaps

Randomized algorithms

- A **randomized algorithm** is one which relies on some random numbers in addition to the input.

  $\left(\right.$Computers cannot generate randomness. We assume that there exists a *pseudo-random number generator (PRNG)*, a deterministic program that uses an initial value or *seed* to generate a sequence of seemingly random numbers. The quality of randomized algorithms depends on the quality of the PRNG!$\left.\right)$

- Doing randomization is often a good idea if an algorithm has bad worst-case time but seems to perform much better on most instances.

- **Goal:** Shift the dependency of run-time from what we can't control (the input) to what we *can* control (the random numbers).

  No more bad instances, just unlucky numbers.

Expected run-time

The run-time of the algorithm now depends on the random numbers.

Define $T_{\mathcal{A}}(I, R)$ to be the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.

The **expected run-time $T^{exp}(I)$ for instance I** is the expected value:

$$T^{exp}(I) = \mathbf{E}[T(I,R)] = \sum_{R} T(I,R) \cdot \Pr(R)$$

Expected run-time

The run-time of the algorithm now depends on the random numbers.

Define $T_{\mathcal{A}}(I, R)$ to be the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.

The **expected run-time $T^{exp}(I)$ for instance I** is the expected value:

$$T^{exp}(I) = \mathbf{E}[T(I, R)] = \sum_{R} T(I, R) \cdot \Pr(R)$$

Now take the *maximum* over all instances of size n to define the **expected run-time** (or formally: *worst-instance expected-luck run-time*) **of $\mathcal{A}$**.

$$T^{exp}(n) := \max_{I \in \mathcal{I}_n} T^{exp}(I)$$

Expected run-time

The run-time of the algorithm now depends on the random numbers.

Define $T_{\mathcal{A}}(I, R)$ to be the run-time of a randomized algorithm $\mathcal{A}$ for an instance I and the sequence R of outcomes of random trials.

The **expected run-time $T^{exp}(I)$ for instance I** is the expected value:

$$T^{exp}(I) = \mathbf{E}[T(I,R)] = \sum_{R} T(I,R) \cdot \Pr(R)$$

Now take the *maximum* over all instances of size n to define the **expected run-time** (or formally: *worst-instance expected-luck run-time*) **of $\mathcal{A}$**.

$$T^{exp}(n) := \max_{I \in \mathcal{I}_n} T^{exp}(I)$$

We can still have good luck or bad luck, so occasionally we also discuss the very worst that could happen, i.e., $\max_I \max_R T(I,R)$.

Analysis of *merge* in a meldable heap

Observe: *merge* does two *random downward walks* in a binary tree.

- Let $T(I, R)$ = length of random downward walk in tree I when random outcomes are R.
- As usual: $T^{\exp}(n) = \max_{|I|=n} \sum_R Pr(R)T(I, R)$.

Theorem: $T^{\exp}(n) \in O(\log n)$.

Proof:

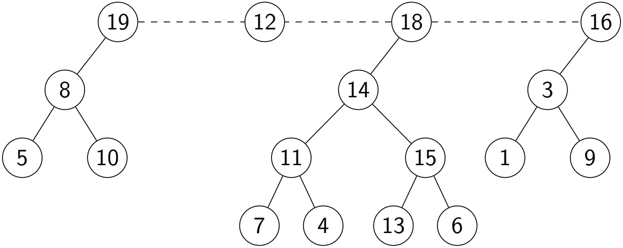
So *merge* (and also *insert* and *delete-max*) takes $O(\log n)$ expected time.

Outline

- ADT Priority Queue
- Binary Heaps as PQ realization
- *PQ-sort* and *heap-sort*
- More PQ operations
- Meldable Heaps
- Detour: Randomized algorithms and their analysis
- 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.
- *delete-max*: Bottleneck is *finding* the maximum.
    - At each flagged tree, root contains the maximum of tree.
    - Search roots in $L \Rightarrow O(|L|)$ time.
    - (Removal also will be non-trivial $\rightsquigarrow$ later)
- We want L to be short.

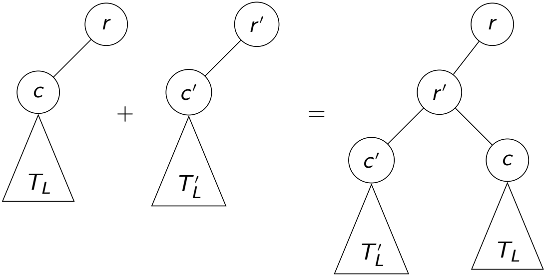
Binomial Heap Operations

- *insert*: Reduce to *merge* as before.
- *delete-max*: Bottleneck is *finding* the maximum.
  - At each flagged tree, root contains the maximum of tree.
  - Search roots in $L \Rightarrow O(|L|)$ time.
  - (Removal also will be non-trivial $\rightsquigarrow$ later)
- 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 $\{0, \ldots, h\}$.

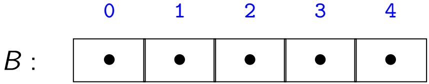
Making Binomial Heaps Proper

- 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 $r.\mathit{key} \geq r'.\mathit{key}$:

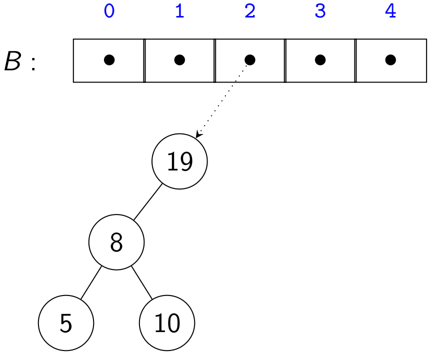


- Idea: Do this whenever two flagged trees have same height.
- With this, *make-proper* can be implemented in $O(|L| + \log n)$ time.

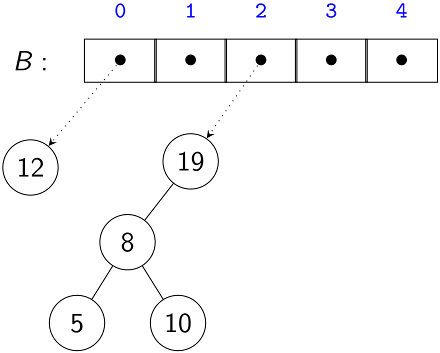
Making Binomial Heaps Proper



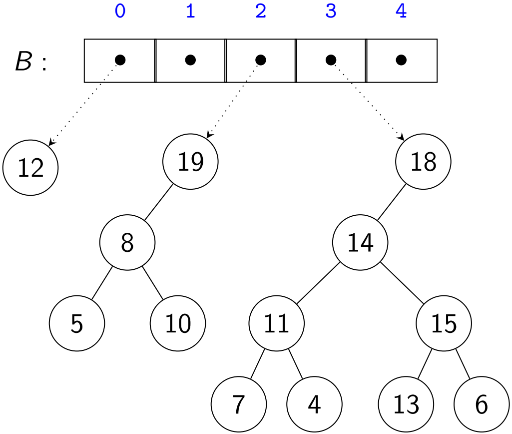
Making Binomial Heaps Proper



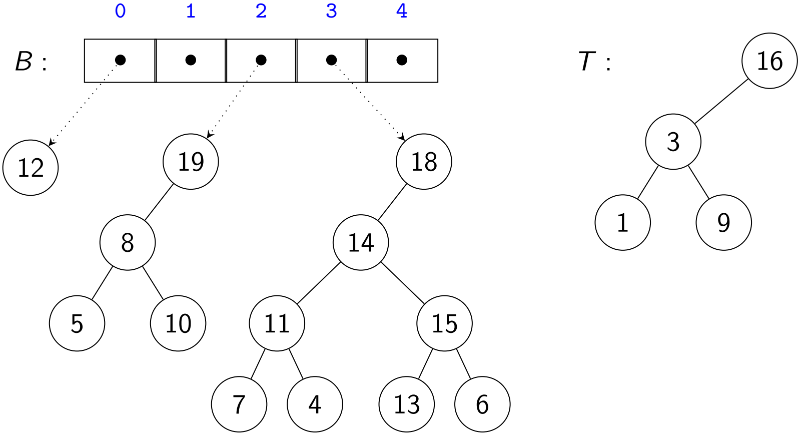
Making Binomial Heaps Proper



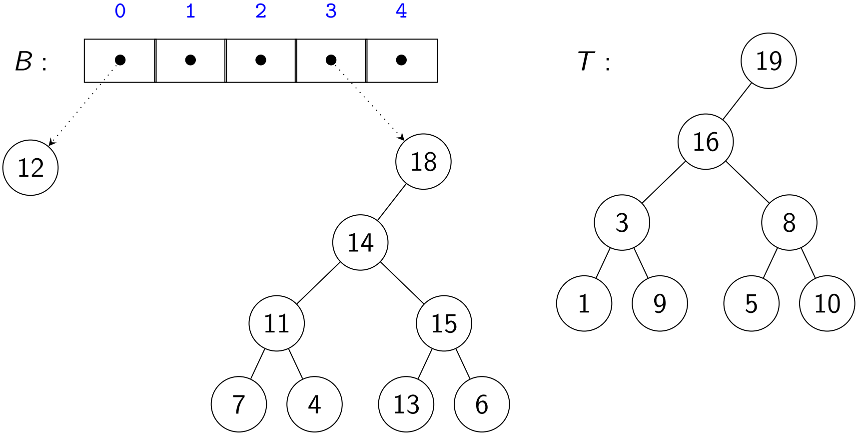
Making Binomial Heaps Proper



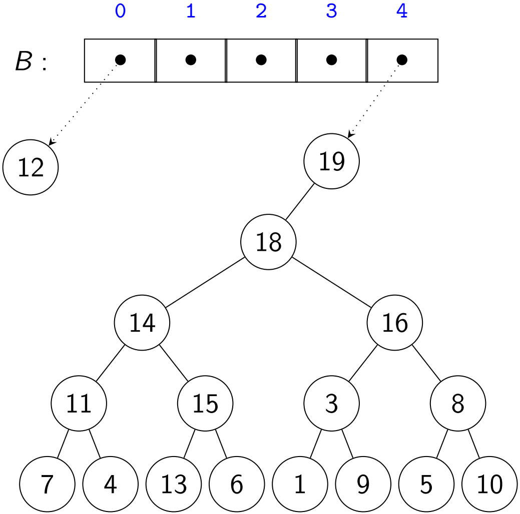
Making Binomial Heaps Proper



Making Binomial Heaps Proper



Making Binomial Heaps Proper



Making Binomial Heaps Proper

```
binomialHeap::make-proper()
n ← size of the binomial heap
compute $\ell \leftarrow \lfloor \log n \rfloor$
$B \leftarrow$ array of size $\ell + 1$, initialized all-NULL
$L \leftarrow$ list of flagged trees
while $L$ is non-empty do
    $T \leftarrow L.pop()$, $h \leftarrow T.height$
    while $T' \leftarrow B[h]$ is not NULL do
        if $T.root.key < T'.root.key$ do swap $T$ and $T'$
        // combine $T$ with $T'$
        $T'.right \leftarrow T.left$, $T.left \leftarrow T'$, $T.height \leftarrow h+1$
        $B[h] \leftarrow$ NULL, $h$++
    $B[h] \leftarrow T$
// copy $B$ back to list
for ($h = 0$; $h \leq \ell$; $h$++) do
    if $B[h] \neq$ NULL do $L.append(B[h])$
```

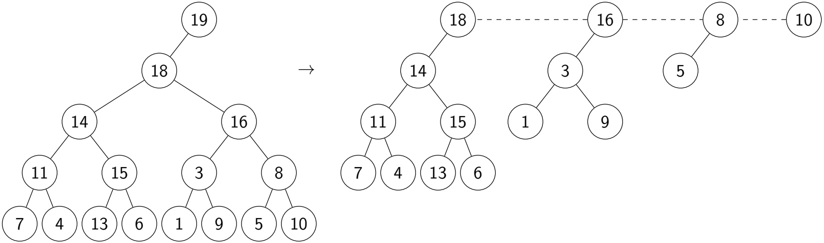
Binomial Heap Operations

- **Idea:** Make binomial heap proper after *every* operation.
  - ⇒ L *always* has length $O(\log n)$
  - ⇒ Each *make-proper* takes $O(\log n)$ time

- *merge*: $O(\log n)$ worst-case time.
  - Concatenate the two lists.
  - Call *make-proper*.
- *find-max*: $O(\log n)$ worst-case time.
  - Find maximum root in $O(|L|)$ time
- *insert*: $O(\log n)$ worst-case time
  - Create new flagged tree with one node, add to L.
  - Call *make-proper*.
- *delete-max*
  - Find maximum as in *find-max*
  - Now how do we remove it?

delete-max in a binomial heap

- Say the maximum key is at root of flagged tree T
- Split $T \setminus \{\text{root}\}$ into into flagged trees $T_1, \ldots, T_k$

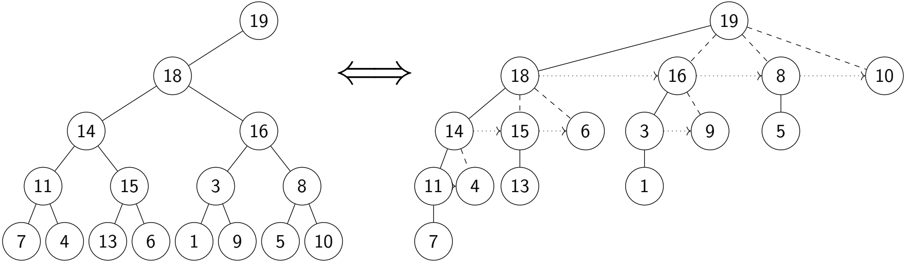


- Remove T from L, create new binomial heap M' with $\{T_1, \ldots, T_k\}$
- Have $k \leq \log n \Rightarrow O(\log n)$ worst-case time.
- Apply *merge* to M' and existing binomial heap

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

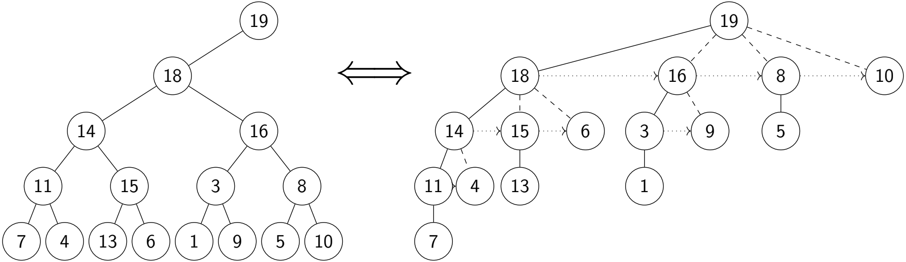
Why these weird conventions?

- Flagged trees, heap-order property, *delete-max* seem unintuitive.
- These are actually very intuitive if one knows **left-child-right-sibling conversion** from binary trees to multi-way trees.



Why these weird conventions?

- Flagged trees, heap-order property, *delete-max* seem unintuitive.
- These are actually very intuitive if one knows **left-child-right-sibling conversion** from binary trees to multi-way trees.



- Flagged tree of height d ⇔ d subtrees, of heights $d-1, d-2, \ldots, 0$
- Binomial-heap order property ⇔ standard heap-order property
- *delete-max* ⇔ create new heap with all children of the root