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

Module 2: Priority Queues

Tom Iagovet

Based on lecture notes by many previous cs240 instructors

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

Spring 2026

version 2026−05−13 17:48

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. $\quad PQ.$*insert*(an item with priority $A[i]$)
4. **for** $i \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[i] \leftarrow$ priority of $PQ.$*delete-max*()

- Note: Run-time depends on how we realize ADT 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 ($\leadsto$ *selection-sort*)
- **Sorted** array or list ($\leadsto$ *insertion-sort*)

Using a priority queue to sort

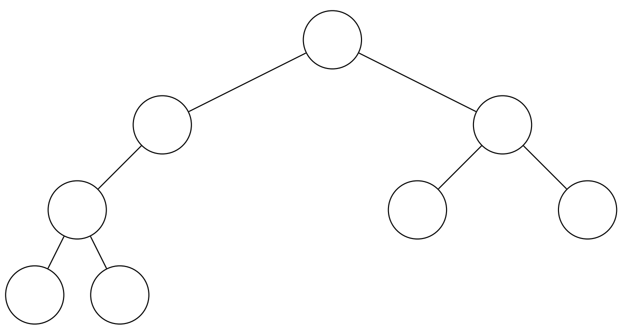
PQ-Sort$(A[0..n-1])$
1. initialize PQ to an empty priority queue
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad PQ.$*insert*(an item with priority $A[i]$)
4. **for** $i \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[i] \leftarrow$ priority of $PQ.$*delete-max*()

- Note: Run-time depends on how we realize ADT 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*)

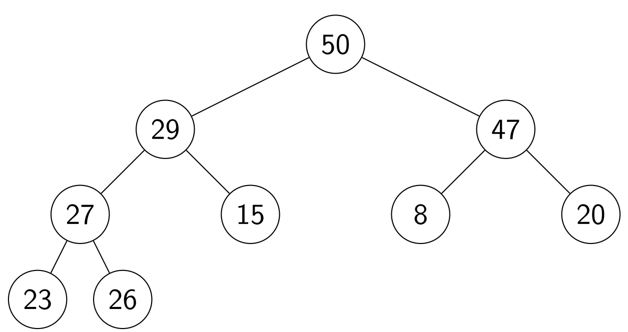
Better realization: Binary heap



Binary tree with

1. structural property and

Better realization: Binary heap



Binary tree with

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

Convention: (15) represents

priority = 15, <other info>

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

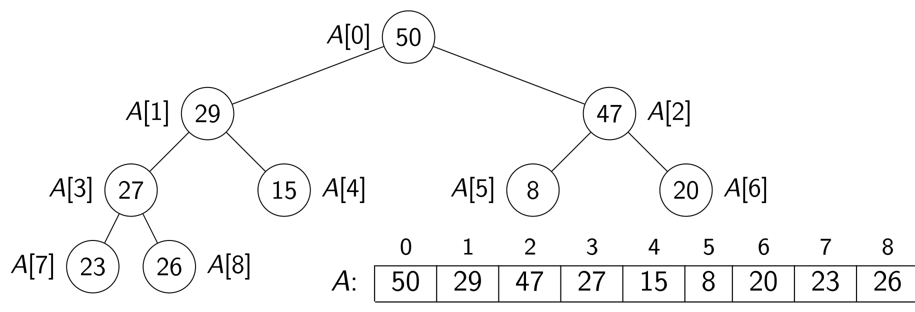
Heaps should *not* be stored as binary trees!

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

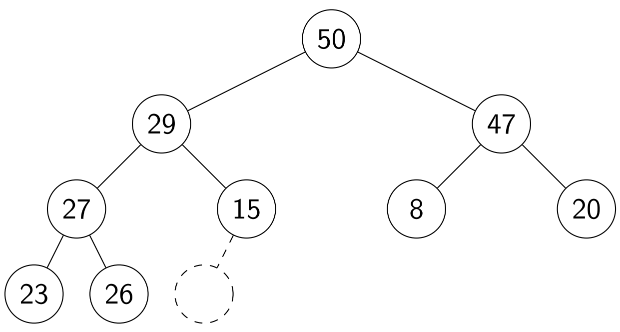
We hide indices behind *accessor-functions* such as *parent*, *left*, *right*, ...

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

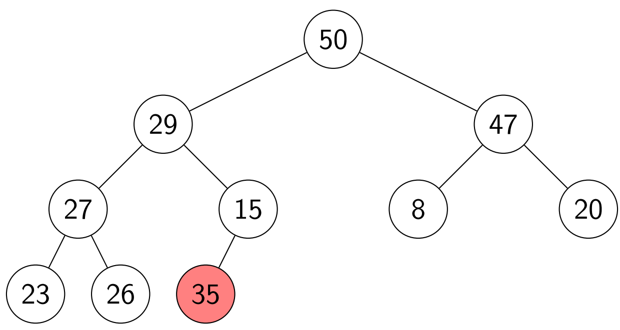
insert(35):



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

insert in Heaps

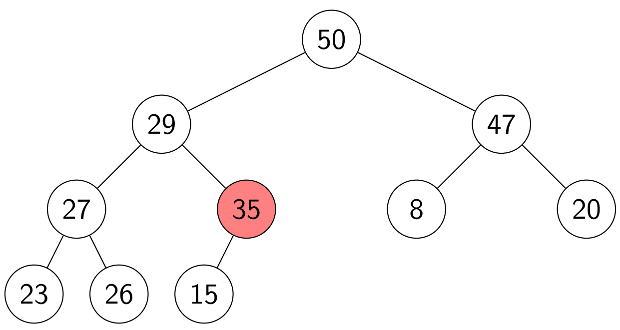
insert(35):



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

insert in Heaps

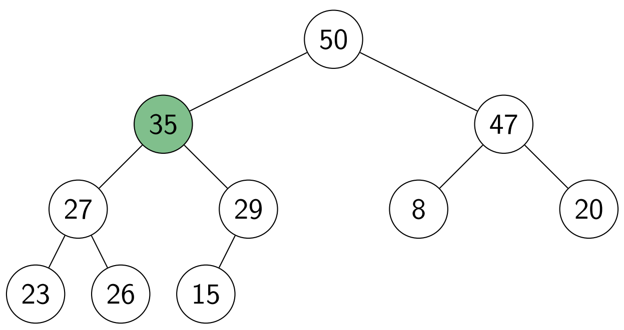
insert(35):



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

insert in heaps

fix-up(A, i) `// i corresponds to a node of the heap`
1. **while** *parent*(i) exists **and** $A[\mathit{parent}(i)].\mathit{key} < A[i].\mathit{key}$ **do**
2. swap $A[i]$ and $A[\mathit{parent}(i)]$
3. $i \leftarrow \mathit{parent}(i)$

insert(x)
1. $A[\ell \leftarrow \mathit{last}() + 1] \leftarrow x$
2. increase *size* `// size: stored by PQ`
3. *fix-up*(A, ℓ)

insert in heaps

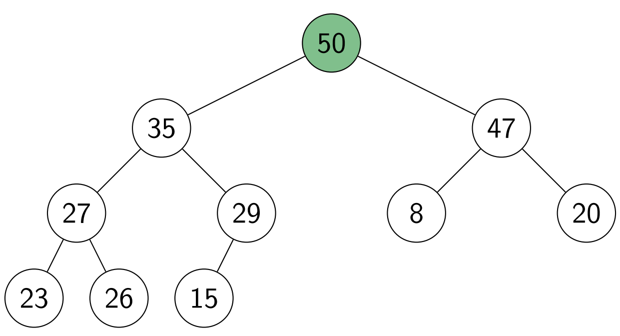
```
fix-up(A, i)    // i corresponds to a node of the heap
1.  while parent(i) exists and A[parent(i)].key < A[i].key do
2.      swap A[i] and A[parent(i)]
3.      i ← parent(i)
```

```
insert(x)
1.  A[ℓ ← last() + 1] ← x
2.  increase size         // size:  stored by PQ
3.  fix-up(A, ℓ)
```

Note: We assume **dynamic arrays** (= `std::vector`)

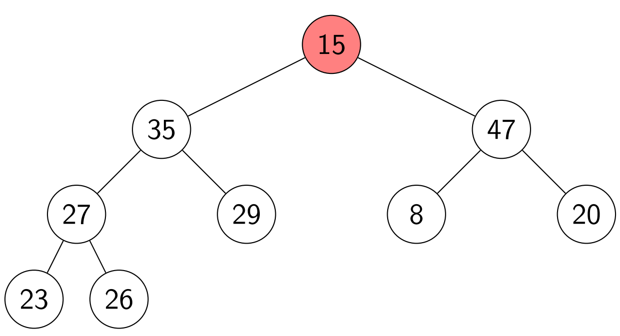
- Keep track of *size* and *capacity* of array.
- If *size* = *capacity*, copy items over to new array (twice as big). This takes $\Theta(n)$ time but happens only after $\Theta(n)$ "cheap" insertions.
- *insert* takes $O(1)$ time when "amortized" (averaged over operations)

delete-max in heaps



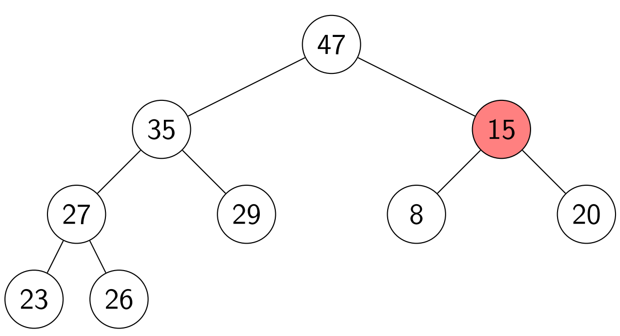
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix*-*down*:

delete-max in heaps



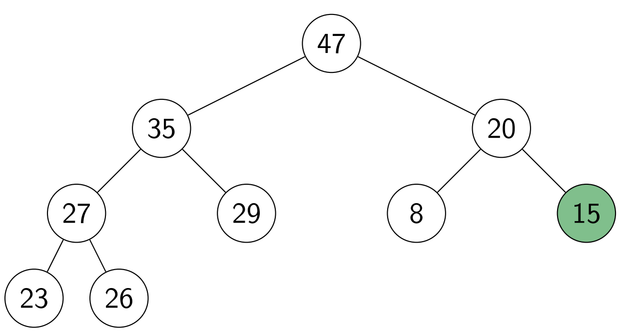
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

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

1. **while** i is not a leaf **do**
2. $\quad j \leftarrow$ left child of i `// find child with larger key`
3. $\quad$ if (i has right child and A[right child of i].*key* $> A[j]$.*key*)
4. $\quad\quad j \leftarrow$ right child of i
5. $\quad$ **if** $A[i]$.*key* $\geq A[j]$.*key* **break**
6. $\quad$ swap $A[j]$ and $A[i]$
7. $\quad i \leftarrow j$

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)
1. initialize H to an empty heap
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad H.\textit{insert}(A[i])$
4. **for** $i \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[i] \leftarrow H.\textit{delete-max}()$

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)
1.  initialize H to an empty heap
2.  for i ← 0 to n − 1 do
3.        H.insert(A[i])
4.  for i ← n − 1 down to 0 do
5.        A[i] ← H.delete-max()
```

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

$\rightsquigarrow$ *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)
1.   initialize H to an empty heap
2.   for i ← 0 to n − 1 do
3.        H.insert(A[i])
4.   for i ← n − 1 down to 0 do
5.        A[i] ← H.delete-max()
```

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

$\rightsquigarrow$ *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. $\rightsquigarrow$ *$O(1)$ 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 1: Use repeated *insert*. $\Theta(n \log n)$.
Solution 2: Using *fix-downs* instead:

```
heapify(A)
A: an array
1.   n ← A.size()
2.   for i ← parent(last()) downto root() do
3.         fix-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 1: Use repeated *insert*. $\Theta(n \log n)$.
Solution 2: Using *fix-downs* instead:

```
heapify(A)
A: an array
 1.  n ← A.size()
 2.  for i ← parent(last()) downto root() do
 3.        fix-down(A, i, n)
```

- Correctness via loop-invariant: For all $j > i$, sub-tree at j is a heap.
- Run-time: Clearly $O(n \log n)$, but this is not tight!

Building heaps with *fix-down*

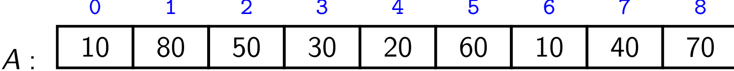
Problem: Given n items all at once (in $A[0 \cdots n-1]$), build a heap containing all of them.

Solution 1: Use repeated *insert*. $\Theta(n \log n)$.
Solution 2: Using *fix-downs* instead:

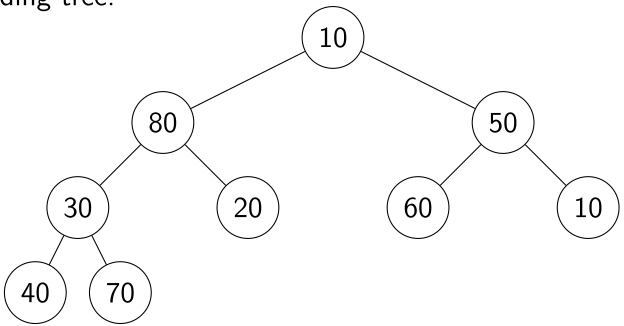
```
heapify(A)
A: an array
 1.   n ← A.size()
 2.   for i ← parent(last()) downto root() do
 3.        fix-down(A, i, n)
```

- Correctness via loop-invariant: For all $j > i$, sub-tree at j is a heap.
- Run-time: Clearly $O(n \log n)$, but this is not tight!
- Goal: *heapify* has run-time $O(n)$ (this is clearly tight).

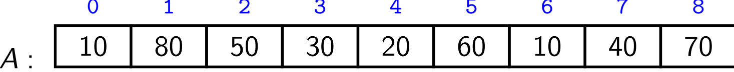
heapify example



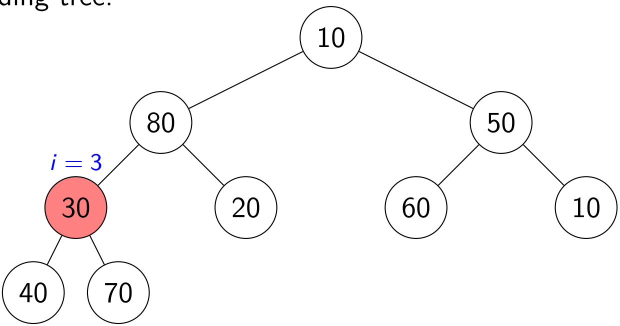
Corresponding tree:



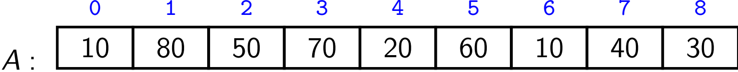
heapify example



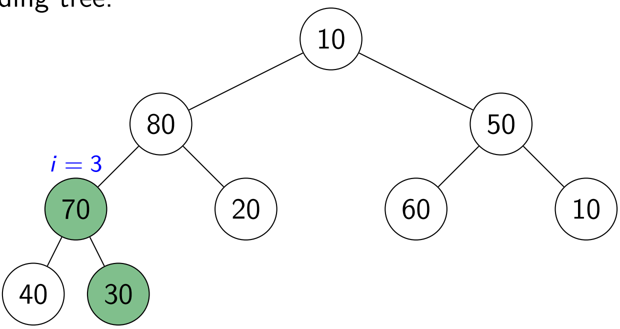
Corresponding tree:



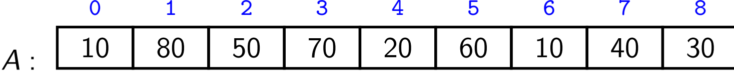
heapify example



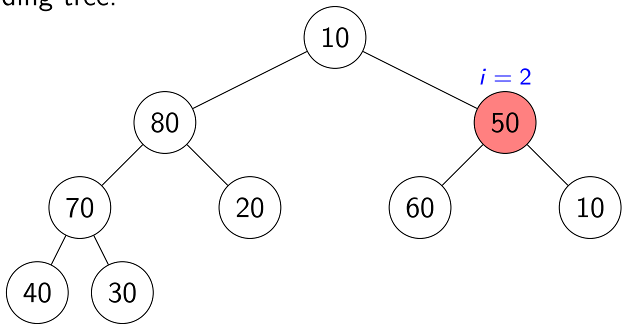
Corresponding tree:



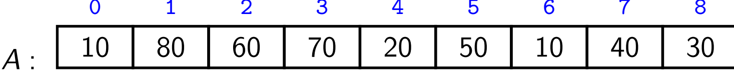
heapify example



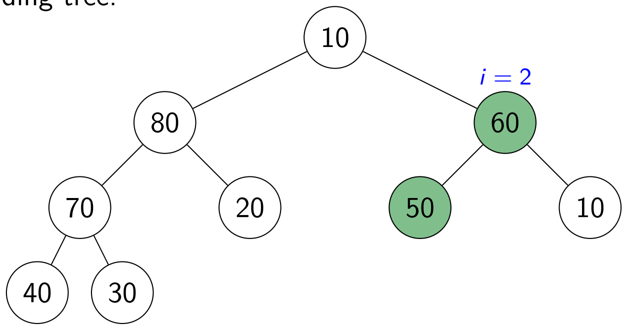
Corresponding tree:



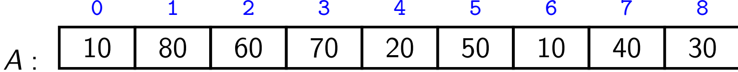
heapify example



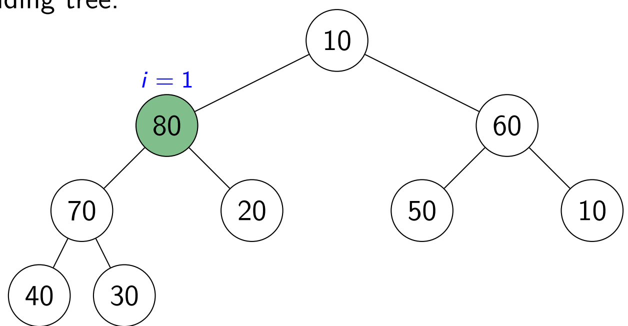
Corresponding tree:



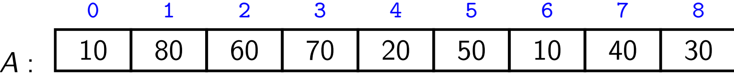
heapify example



Corresponding tree:



heapify example



Corresponding tree:

$i = 0$

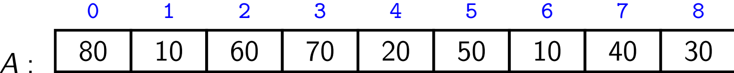
10

80 60

70 20 50 10

40 30

heapify example

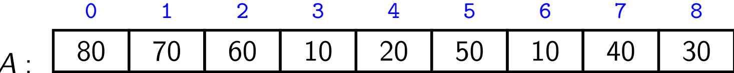


Corresponding tree:

$i = 0$

80
10 60
70 20 50 10
40 30

heapify example



Corresponding tree:

$i = 0$

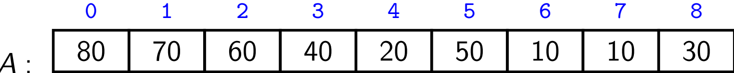
80

70 60

10 20 50 10

40 30

heapify example



Corresponding tree:

$i = 0$

80
70 60
40 20 50 10
10 30

heapify run-time: proof

heapify run-time: proof

Efficient sorting with heaps

- Idea: *PQ-sort* with heaps.
- $O(1)$ auxiliary space: Use same input-array A for storing heap.

heap-sort(A)
1. `// heapify`
2. $n \leftarrow A.size()$
3. **for** $i \leftarrow$ *parent*(*last*()) **downto** 0 **do**
4. *fix-down*(A, i, n)
5. `// repeatedly find maximum`
6. **while** $n > 1$
7. `// 'delete' maximum by moving to end and decreasing n`
8. *swap* items at A[*root*()] and A[*last*()]
9. decrease n
10. *fix-down*(A, *root*(), n)

The for-loop takes $\Theta(n)$ time and the while-loop takes $\Theta(n \log n)$ 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

More operations for priority queues

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

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

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

More operations for priority queues

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

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

- Change key of node z to k if k is bigger/smaller than z.*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 operations for priority queues

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

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

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

merge(P_1, P_2)

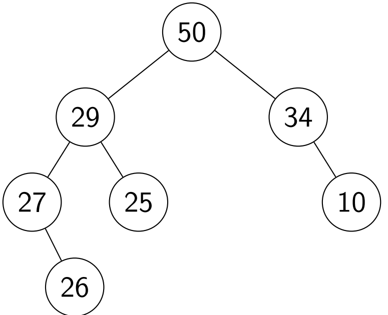
- Given: two priority queues P_1, P_2 of size n_1 and n_2.
- Want: One priority queue P that contains all their items
- **Outlook:** Three approaches (where $n = n_1 + n_2$):
  - Merge binary heaps. $O(\log^3 n)$ worst-case time (no details)
  - **Meldable heaps**: heap-order-property but no structural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

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

P.*delete-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

1. **if** r_1 is `NULL` **return** r_2
2. **if** r_2 is `NULL` **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 *merge*(c, r_2)
7. **return** r_1

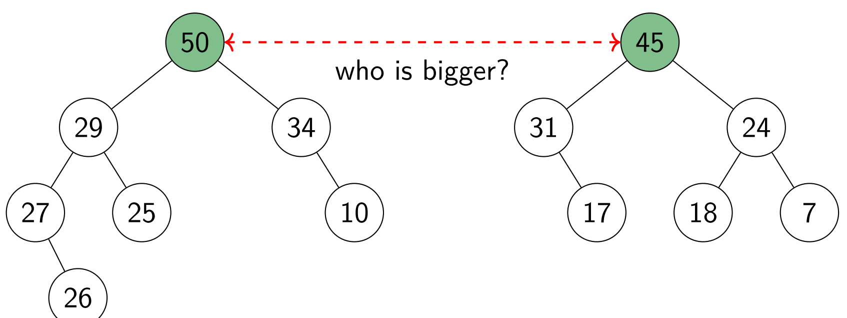
Merging meldable heaps

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

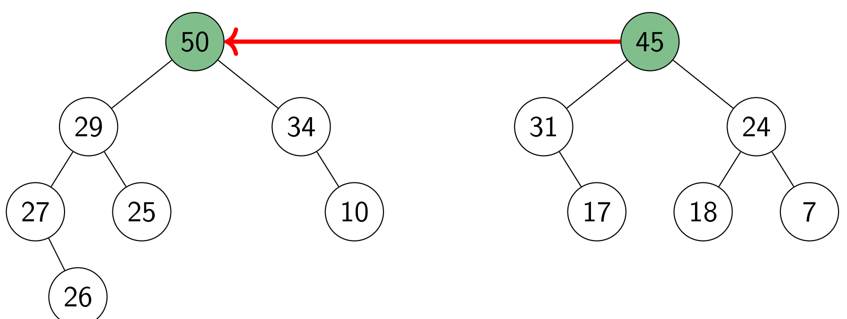
meldableHeap::merge(r_1, r_2)
r_1, r_2: *roots of two heaps (possibly* `NULL`*)*
returns root of merged heap
1. **if** r_1 is `NULL` **return** r_2
2. **if** r_2 is `NULL` **return** r_1
3. **if** $r_1.key < r_2.key$ *swap*(r_1, r_2)
4. `// now` r_1 `has max-key and becomes the root.`
5. *randomly* pick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

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

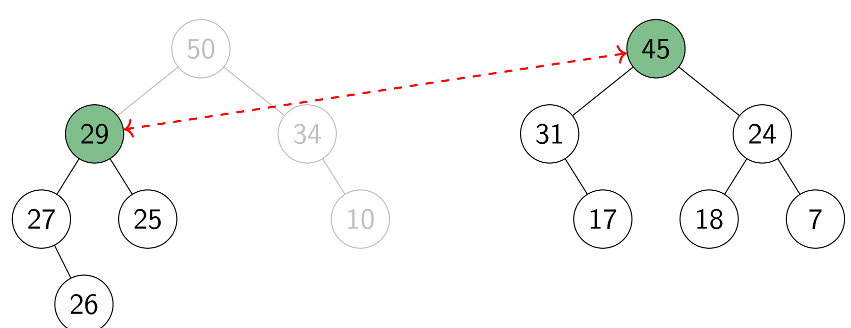
Merge: Example



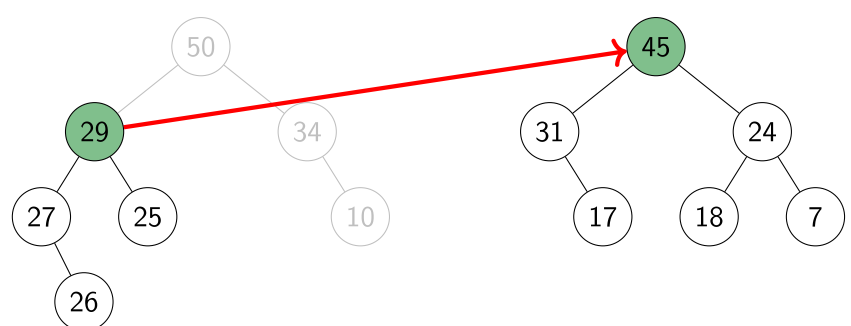
Merge: Example



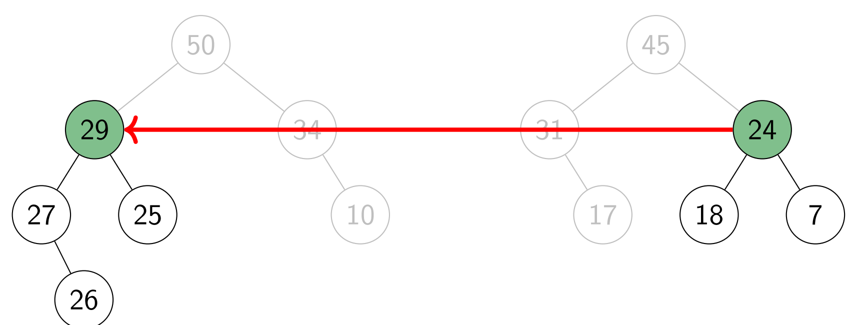
Merge: Example



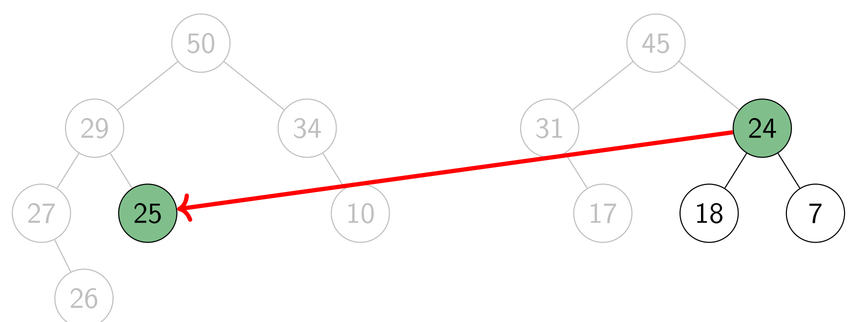
Merge: Example



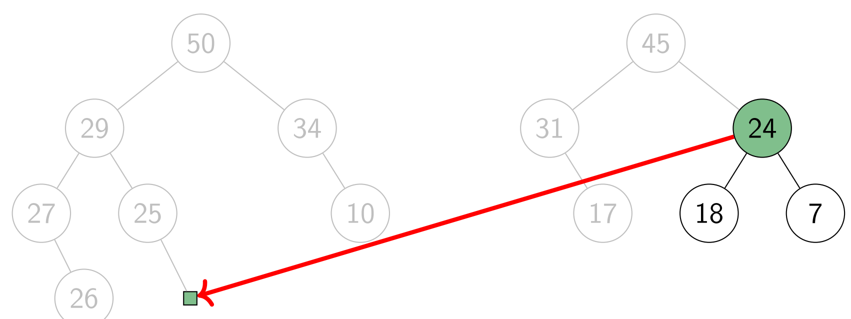
Merge: Example



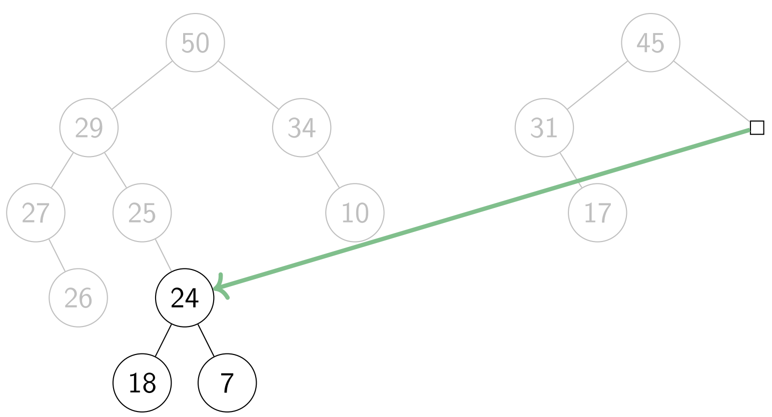
Merge: Example



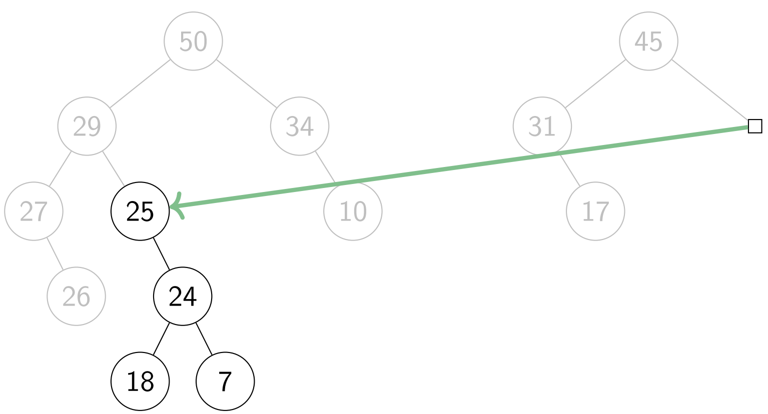
Merge: Example



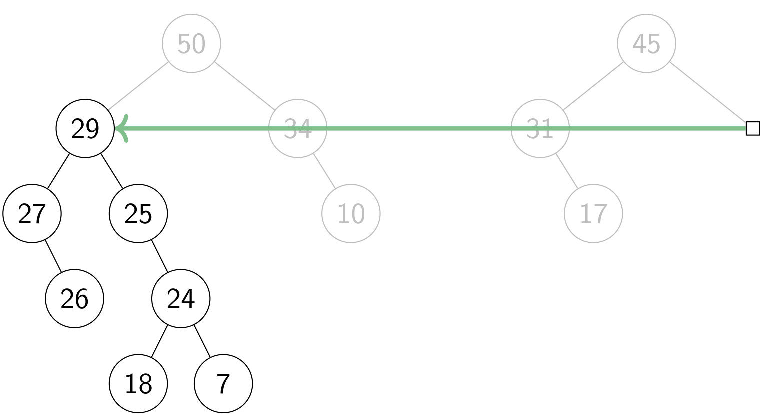
Merge: Example



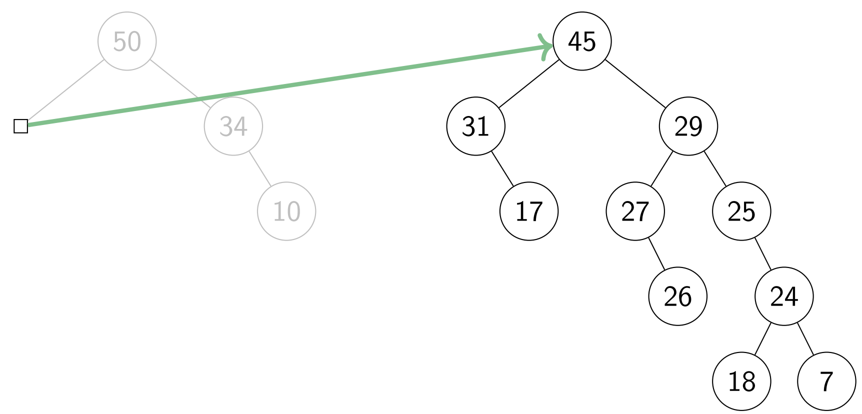
Merge: Example



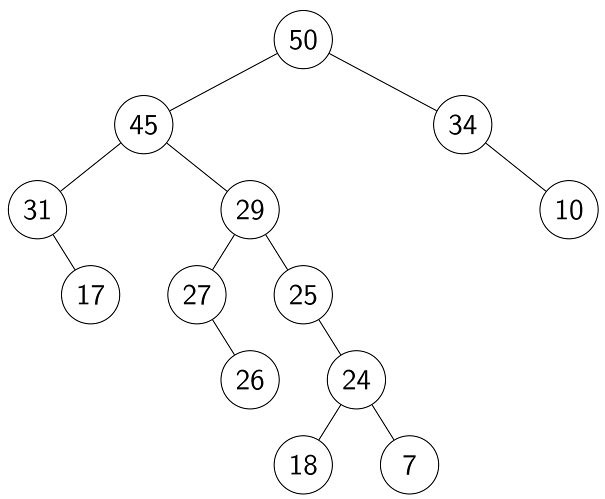
Merge: Example



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.

  (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!)

- We assume a constant-time method $\mathit{random}(N)$, which returns $i \in \{0, \ldots, N-1\}$ chosen uniformly and independently.

Randomized algorithms

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

  (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!)

- We assume a constant-time method *random*(N), which returns $i \in \{0, \ldots, N-1\}$ chosen uniformly and independently.

- **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:

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

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:

$$T^{\exp}(I) = E[T(I,R)] = \sum_R T(I,R) \cdot \mathbb{P}(R)$$

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:

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

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 I.

random-downward-walk(I)
1. **if** I is an empty subtree **then return**
2. $x \leftarrow$ *random*(2) `// random coin toss`
3. **if** $x = 0$ **then** *random-downward-walk*(I.*left*)
4. **else** *random-downward-walk*(I.*right*)

Analysis of *merge* in a meldable heap

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

random-downward-walk(I)
1. **if** I is an empty subtree **then return**
2. $x \leftarrow$ *random*(2) `// random coin toss`
3. **if** $x = 0$ **then** *random-downward-walk*($I.left$)
4. **else** *random-downward-walk*($I.right$)

- Define $T^{\exp}(n)$ = expected length of a random downward walk in a binary tree with n nodes.
- Goal: Develop a recursive formula for $T^{\exp}(n)$

Analysis of *merge* in a meldable heap

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

random-downward-walk(I)
1. **if** I is an empty subtree **then return**
2. $x \leftarrow$ *random*(2) `// random coin toss`
3. **if** $x = 0$ **then** *random-downward-walk*(I.*left*)
4. **else** *random-downward-walk*(I.*right*)

- Define $T^{\exp}(n)$ = expected length of a random downward walk in a binary tree with n nodes.
- Goal: Develop a recursive formula for $T^{\exp}(n)$
- Outline:
  1. Write a formula for $T(I, R)$ = length on input I if the random outcomes are R.
  2. Derive a formula for $T^{\exp}(I) = \sum_R \mathbb{P}(R) T(I, R)$
  3. Derive the recursive formula for $T^{\exp}(n) = \max_{I \in \mathcal{I}_n} T^{\exp}(I)$

Expected run-time of *random-downward-walk*

$T(I, R)$ = length on input I if the random outcomes are R.

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\mathbb{P}(R) = \mathbb{P}(x) \cdot \mathbb{P}(R')$ (random choices are independent).

Expected run-time of *random-downward-walk*

$T(I, R)$ = length on input I if the random outcomes are R.

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\mathbb{P}(R) = \mathbb{P}(x) \cdot \mathbb{P}(R')$ (random choices are independent).

- Recursive formula for one instance I (with left/right subtree I_ℓ / I_r):

$$T(I, R) = T(I, \langle x, R' \rangle) = \begin{cases} 1 + T(I_\ell) & \text{if } x = 0 \\ 1 + T(I_r) & \text{if } x = 1 \end{cases}$$

Expected run-time of *random-downward-walk*

$T(I, R)$ = length on input I if the random outcomes are R.

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\mathbb{P}(R) = \mathbb{P}(x) \cdot \mathbb{P}(R')$ (random choices are independent).

- Recursive formula for one instance I (with left/right subtree I_ℓ / I_r):

$$T(I, R) = T(I, \langle x, R' \rangle) = \begin{cases} 1 + T(I_\ell) & \text{if } x = 0 \\ 1 + T(I_r) & \text{if } x = 1 \end{cases}$$

$T^{\exp}(I) = \sum_R \mathbb{P}(R) T(I, R) = \dots$

$$= 1 + \tfrac{1}{2} T^{\exp}(I_\ell) + \tfrac{1}{2} T^{\exp}(I_r)$$

Expected run-time of *random-downward-walk*

$T(I, R)$ = length on input I if the random outcomes are R.

- The random outcomes R consist of two parts $R = \langle x, R' \rangle$:
  - x: outcome of first coin toss
  - R': random outcomes (if any) for the recursions

  We have $\mathbb{P}(R) = \mathbb{P}(x) \cdot \mathbb{P}(R')$ (random choices are independent).

- Recursive formula for one instance I (with left/right subtree I_ℓ / I_r):

$$T(I, R) = T(I, \langle x, R' \rangle) = \begin{cases} 1 + T(I_\ell) & \text{if } x = 0 \\ 1 + T(I_r) & \text{if } x = 1 \end{cases}$$

$T^{\exp}(I) = \sum_R \mathbb{P}(R) T(I, R) = \ldots$

$$= 1 + \tfrac{1}{2} T^{\exp}(I_\ell) + \tfrac{1}{2} T^{\exp}(I_r) \leq 1 + \tfrac{1}{2} T^{\exp}(|I_\ell|) + \tfrac{1}{2} T^{\exp}(|I_r|)$$

Analysis of *merge* in a meldable heap

Crucial insight: The right-hand side of

$$T^{\exp}(I) \leq 1 + \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r)$$

does *not* depend on I, only on the sizes n_ℓ, n_r of the subtrees.

Analysis of *merge* in a meldable heap

Crucial insight: The right-hand side of

$$T^{\exp}(I) \le 1 + \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r)$$

does *not* depend on I, only on the sizes n_ℓ, n_r of the subtrees.

So

$$T^{\exp}(n) = \max_{I \in \mathcal{I}_n} T^{\exp}(I) \le 1 + \max_{n_\ell + n_r = n-1} \Big\{ \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r) \Big\}.$$

Analysis of *merge* in a meldable heap

Crucial insight: The right-hand side of

$$T^{\exp}(I) \leq 1 + \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r)$$

does *not* depend on I, only on the sizes n_ℓ, n_r of the subtrees.
So

$$T^{\exp}(n) = \max_{I \in \mathcal{I}_n} T^{\exp}(I) \leq 1 + \max_{n_\ell + n_r = n-1} \Big\{ \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r) \Big\}.$$

This gives us a recursive formula (with $T(0) = 0$) that we can analyze.

Analysis of *merge* in a meldable heap

Crucial insight: The right-hand side of

$$T^{\exp}(I) \le 1 + \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r)$$

does *not* depend on I, only on the sizes n_ℓ, n_r of the subtrees.
So

$$T^{\exp}(n) = \max_{I \in \mathcal{I}_n} T^{\exp}(I) \le 1 + \max_{n_\ell + n_r = n-1} \Big\{ \tfrac{1}{2}\,T^{\exp}(n_\ell) + \tfrac{1}{2}\,T^{\exp}(n_r) \Big\}.$$

This gives us a recursive formula (with $T(0) = 0$) that we can analyze.

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

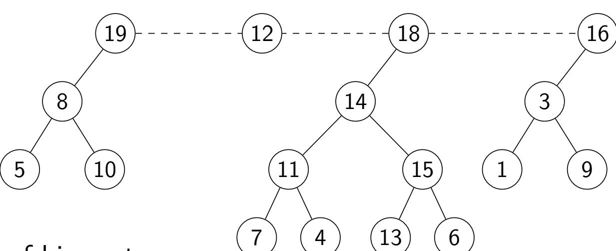
So *merge* (and also *insert* and *delete-max*) takes $O(\log n)$ expected time.

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.)

(These strange conventions make sense if we convert each binary tree to a multiway-tree. See course notes for details.)

Binomial heaps: 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 heaps: Operations

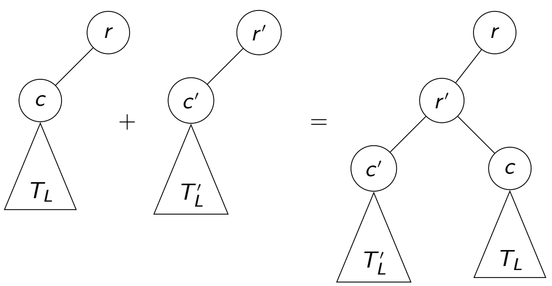
- *insert*: Reduce to *merge* as before.
- *delete-max*: Bottleneck is *finding* the maximum.
  - At each flagged tree, root contains the maximum of tree.
  - Search roots in $L \Rightarrow O(|L|)$ time.
  - (Removal also will be non-trivial $\rightsquigarrow$ later)
- We want L to be short.

- **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 $\{0, \ldots, h\}$.

Binomial heaps: Make 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 $r.key \geq r'.key$:

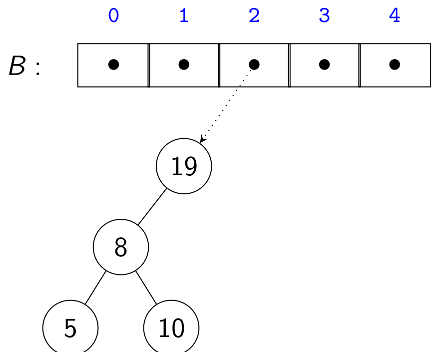


- Idea: Do this whenever two flagged trees have same height.
- To find such pairs: Temporarily put flagged trees into an array B ("buckets") indexed by height of flagged tree.

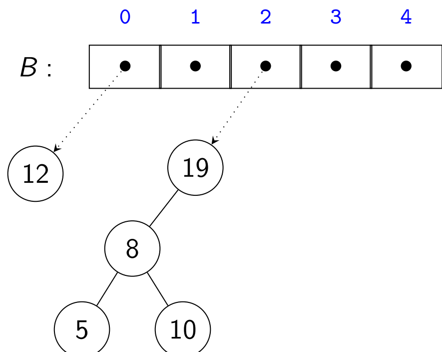
Binomial heaps: Make proper

	0	1	2	3	4
B :	•	•	•	•	•

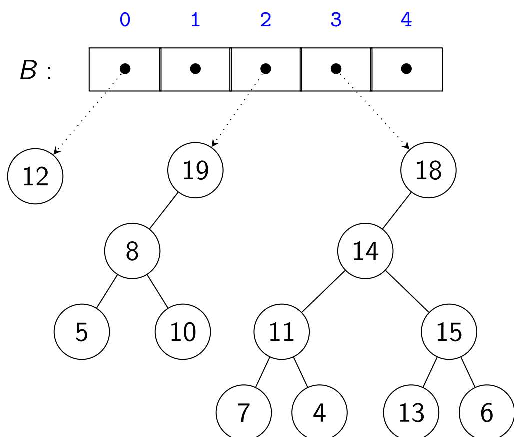
Binomial heaps: Make proper



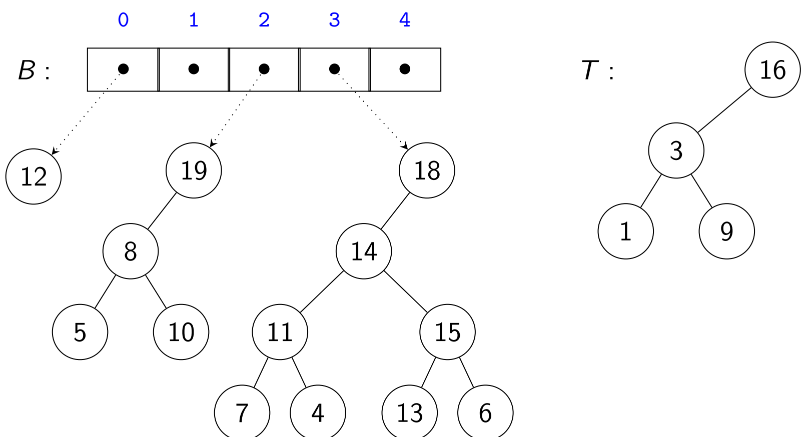
Binomial heaps: Make proper



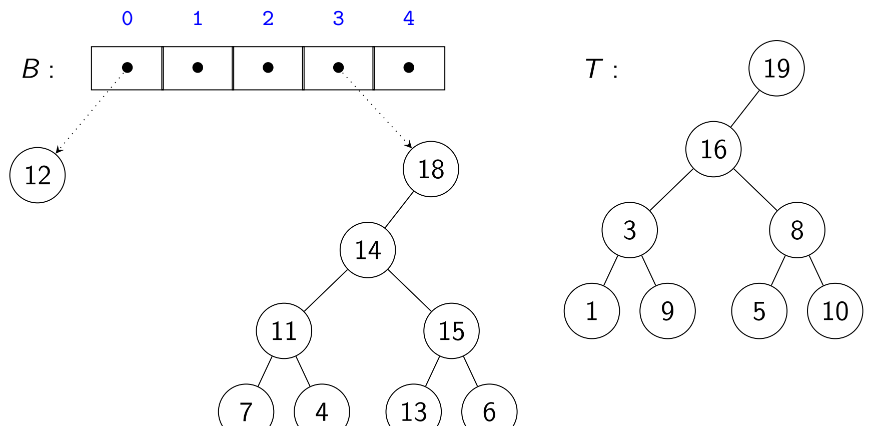
Binomial heaps: Make proper



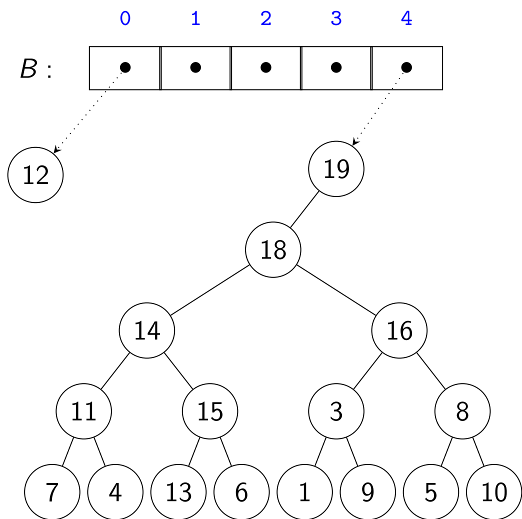
Binomial heaps: Make proper



Binomial heaps: Make proper



Binomial heaps: Make proper



Binomial heaps: Make proper

binomialHeap::make-proper()
1. $n \leftarrow$ total number of stored keys
2. **for** $(\ell \leftarrow 0; n > 1; \ell{+}{+})$ **do** $n \leftarrow n/2$ `// compute` $\lfloor \log n \rfloor$
3. $B \leftarrow$ array of size $\ell + 1$, initialized all-`NULL`
4. $L \leftarrow$ list of flagged trees
5. **while** L is non-empty **do**
6. $\quad T \leftarrow L.pop()$, $h \leftarrow T.height$
7. $\quad$ **while** $T' \leftarrow B[h]$ is not `NULL` **do**
8. $\quad\quad$ **if** $T.root.key < T'.root.key$ **do** swap T and T'
9. $\quad\quad$ `// combine` T `with` T'
10. $\quad\quad T'.right \leftarrow T.left$, $T.left \leftarrow T'$, $T.height \leftarrow h{+}1$
11. $\quad\quad B[h] \leftarrow$ `NULL`, $h{+}{+}$
12. $\quad B[h] \leftarrow T$
13. `// copy` B `back to list`
14. **for** $(h = 0;\ h \leq \ell;\ h{+}{+})$ **do**
15. $\quad$ **if** $B[h] \neq$ `NULL` **do** $L.append(B[h])$

Run-time: $O(|L| + \log n)$

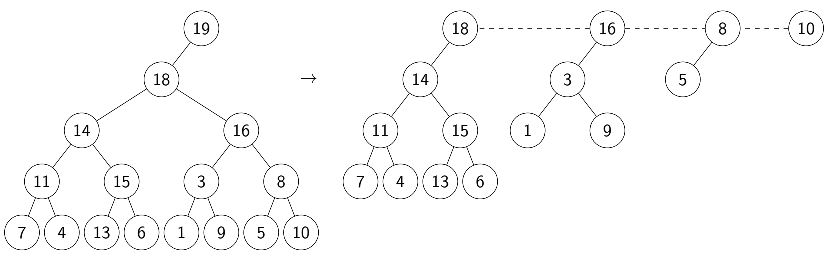
Binomial heap: Operations

- **Idea:** Make binomial heap proper after *every* operation.
  - $\Rightarrow$ L *always* has length $O(\log n)$
  - $\Rightarrow$ 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?

Binomial heap: *delete-max*

- 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.