Module 2: Priority Queues

T. Biedl    M. Petrick    O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2019

References: Sedgewick 9.1-9.4
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - PQ-Sort and Heapsort
   - Intro for the Selection Problem
Outline

1 Priority Queues
   • Abstract Data Types
   • ADT Priority Queue
   • Binary Heaps
   • Operations in Binary Heaps
   • PQ-Sort and Heapsort
   • Intro for the Selection Problem
Abstract Data Types

**Abstract Data Type (ADT):** A description of *information* and a collection of *operations* on that information.

The information is accessed *only* through the operations.

We can have various *realizations* of an ADT, which specify:

- How the information is stored (*data structure*)
- How the operations are performed (*algorithms*)
Stack ADT

**Stack**: an ADT consisting of a collection of items with operations:

- *push*: inserting an item
- *pop*: removing the most recently inserted item

Items are removed in LIFO (*last-in first-out*) order.

We can have extra operations: *size*, *isEmpty*, and *top*

Applications: Addresses of recently visited sites in a Web browser, procedure calls

Realizations of Stack ADT

- using arrays
- using linked lists
Queue ADT

**Queue:** an ADT consisting of a collection of items with operations:

- **enqueue:** inserting an item
- **dequeue:** removing the least recently inserted item

Items are removed in FIFO (*first-in first-out*) order. Items enter the queue at the *rear* and are removed from the *front*.

We can have extra operations: *size*, *isEmpty*, and *front*

Realizations of Queue ADT

- using (circular) arrays
- using linked lists
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
     - Binary Heaps
     - Operations in Binary Heaps
     - PQ-Sort and Heapsort
     - Intro for the Selection Problem
Priority Queue ADT

**Priority Queue:** An ADT consisting of a collection of items (each having a *priority*) with operations

- **insert:** inserting an item tagged with a priority
- **deleteMax:** removing the item of *highest priority*

`deleteMax` is also called `extractMax` or `getmax`.

The priority is also called *key*.

The above definition is for a *maximum-oriented* priority queue. A *minimum-oriented* priority queue is defined in the natural way, by replacing the operation `deleteMax` by `deleteMin`.

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 k ← 0 to n − 1 do
3.    PQ.insert(A[k])
4. for k ← n − 1 down to 0 do
5.    A[k] ← PQ.deleteMax()
```

- Note: Run-time depends on how we implement the priority queue.
- Sometimes written as: $O(n + n \cdot \text{insert} + n \cdot \text{deleteMax})$
Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

\[ \text{insert: } \mathcal{O}(1) \]
\[ \text{deleteMax: } \mathcal{O}(n) \]

Note: We assume dynamic arrays, i.e., expand by doubling as needed. (Amortized over all insertions this takes \( \mathcal{O}(1) \) extra time.)

Using unsorted linked lists is identical.

This realization used for sorting yields *selection sort*.

Attempt 2: Use *sorted arrays*

\[ \text{insert: } \mathcal{O}(n) \]
\[ \text{deleteMax: } \mathcal{O}(1) \]

Using sorted linked-lists is identical.

This realization used for sorting yields *insertion sort*.
Realizations of Priority Queues

Attempt 1: Use unsorted arrays

- insert: $O(1)$
- deleteMax: $O(n)$

**Note:** We assume *dynamic arrays*, i.e., expand by doubling as needed. (Amortized over all insertions this takes $O(1)$ extra time.)
Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

- insert: $O(1)$
- deleteMax: $O(n)$

**Note:** We assume *dynamic arrays*, i.e., expand by doubling as needed. (Amortized over all insertions this takes $O(1)$ extra time.)

Using unsorted linked lists is identical.

This realization used for sorting yields *selection sort*.
Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

- insert: $O(1)$
- deleteMax: $O(n)$

**Note:** We assume *dynamic arrays*, i.e., expand by doubling as needed. (Amortized over all insertions this takes $O(1)$ extra time.)

Using unsorted linked lists is identical.
This realization used for sorting yields *selection sort*.

Attempt 2: Use *sorted arrays*
Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

- *insert:* $O(1)$
- *deleteMax:* $O(n)$

**Note:** We assume *dynamic arrays*, i.e., expand by doubling as needed. (Amortized over all insertions this takes $O(1)$ extra time.)

Using unsorted linked lists is identical.
This realization used for sorting yields *selection sort*.

Attempt 2: Use *sorted arrays*

- *insert:* $O(n)$
- *deleteMax:* $O(1)$

Using sorted linked-lists is identical.
This realization used for sorting yields *insertion sort*. 
Outline

1. **Priority Queues**
   - Abstract Data Types
   - ADT Priority Queue
   - **Binary Heaps**
   - Operations in Binary Heaps
   - PQ-Sort and Heapsort
   - Intro for the Selection Problem
Third Realization: Heaps

A \textit{(binary) heap} is a certain type of binary tree.

You should know:

- A \textit{binary tree} is either
  - empty, or
  - consists of three parts: a node and two binary trees (left subtree and right subtree).

- Terminology: root, leaf, parent, child, level, sibling, ancestor, descendant, etc.

- Any binary tree with \( n \) nodes has height at least \( \log(n + 1) - 1 \in \Omega(\log n) \).
In our examples we only show the priorities, and we show them directly in the node. A more accurate picture would be (priority = 50, <other info>).
Heaps – Definition

A **max-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 parent of $i$ is larger than or equal to key of $i$.

A **min-heap** is the same, but with opposite order property.
Heaps – Definition

A **max-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 parent of \( i \) is larger than or equal to key of \( i \).

A **min-heap** is the same, but with opposite order property.

**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.
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.

```
50
A[0]
```

```
29  A[1]
```

```
27  15  A[3]
```

```
```

```
34  A[2]
```

```
```

```
8  A[5]
```

Biedl, Petrick, Veksler (SCS, UW)  
CS240 – Module 2  
Winter 2019  11 / 23
Heaps in Arrays – Navigation

It is easy to navigate the heap using this array representation:

- the **root** node is $A[0]$

  The textbook puts it at $A[1]$ instead. This gives prettier formulas but more complicated heapsort code.

- the **left child** of $A[i]$ (if it exists) is $A[2i + 1]$,
- the **right child** of $A[i]$ (if it exists) is $A[2i + 2]$,
- the **parent** of $A[i]$ ($i \neq 0$) is $A[\lfloor \frac{i-1}{2} \rfloor]$
- the **last** node is $A[n - 1]$
It is easy to navigate the heap using this array representation:

- the *root* node is $A[0]$
- The textbook puts it at $A[1]$ instead. This gives prettier formulas but more complicated heapsort code.
- the *left child* of $A[i]$ (if it exists) is $A[2i + 1]$,
- the *right child* of $A[i]$ (if it exists) is $A[2i + 2]$,
- the *parent* of $A[i]$ ($i \neq 0$) is $A[\left\lfloor \frac{i-1}{2} \right\rfloor]$
- the *last* node is $A[n - 1]$

Should hide implementation details using helper-functions!

- functions `root()`, `parent(i)`, `last(n)`, etc.
Outline

1 Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - PQ-Sort and Heapsort
   - Intro for the Selection Problem
Insertion in Heaps

- Place the new key at the first free leaf
- The heap-order property might be violated: perform a fix-up:

\[
\text{fix-up}(A, k)
\]

1. while parent(k) exists and \(A[parent(k)] < A[k]\) do
2. swap \(A[k]\) and \(A[parent(k)]\)
3. \(k \leftarrow \text{parent}(k)\)

The new item bubbles up until it reaches its correct place in the heap.

Time: \(O(\text{height of heap}) = O(\log n)\).
Insertion in Heaps

- Place the new key at the first free leaf
- The heap-order property might be violated: perform a fix-up:

\[
\text{fix-up}(A, k) \\
\text{ }k: \text{ an index corresponding to a node of the heap} \\
1. \text{ while } \text{parent}(k) \text{ exists and } A[\text{parent}(k)] < A[k] \text{ do} \\
2. \text{ swap } A[k] \text{ and } A[\text{parent}(k)] \\
3. \text{ } k \leftarrow \text{parent}(k)
\]

The new item bubbles up until it reaches its correct place in the heap.
Insertion in Heaps

- Place the new key at the first free leaf
- The heap-order property might be violated: perform a \textit{fix-up}:

\begin{quote}
  \textit{fix-up}(A, k)  \\
  \text{\textit{k: an index corresponding to a node of the heap}}  \\
  1. \textbf{while} parent(k) exists \textbf{and} \ A[parent(k)] < A[k] \textbf{ do}  \\
  2. \textbf{swap} A[k] \textbf{ and } A[parent(k)]  \\
  3. \text{\textit{k ← parent(k)}}
\end{quote}

The new item bubbles up until it reaches its correct place in the heap.

Time: $O(\text{height of heap}) = O(\log n)$. 
fix-up example
fix-up example

```
  50
 /   \
29    34
 / \  /  \  /  \
27  29  34  8  10
 /     /   /  /
23    26  48  8  10
```

Biedl, Petrick, Veksler (SCS, UW)
fix-up example
fix-up example
deleteMax 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*:

```
A: an array that stores a heap of size n
k: an index corresponding to a node of the heap

while k is not a leaf do
  // Find the child with the larger key
  j ← left child of k
  if (j is not last (n) and A[j+1] > A[j])
    j ← j + 1
  if A[k] ≥ A[j]
    break
  swap A[j] and A[k]
  k ← j
```

Time: $O(\text{height of heap}) = O(\log n)$. 
deleteMax 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 \textit{fix-down}:

\begin{verbatim}
fix-down(A, n, k)
A: an array that stores a heap of size n
k: an index corresponding to a node of the heap
1. while k is not a leaf do
2. // Find the child with the larger key
3. j ← left child of k
4. if (j is not last(n) and A[j + 1] > A[j])
5. j ← j + 1
7. swap A[j] and A[k]
8. k ← j
\end{verbatim}

Time: $O(\text{height of heap}) = O(\log n)$. 
**fix-down example**

```
50
/   \
48   34
|    /   |
27   29   8   10
|    |
23   26   15
```

Biedl, Petrick, Veksler (SCS, UW)
fix-down example
fix-down example
fix-down example
Priority Queue Realization Using Heaps

- Store items in priority queue in array $A$ and keep track of $size$

**insert($x$)**
1. increase $size$
2. $\ell \leftarrow \text{last}(size)$
3. $A[\ell] \leftarrow x$
4. $\text{fix-up}(A, \ell)$

**deleteMax()**
1. $\ell \leftarrow \text{last}(size)$
2. swap $A[root()]$ and $A[\ell]$
3. decrease $size$
4. $\text{fix-down}(A, size, root())$
5. return $(A[\ell])$

*insert* and *deleteMax*: $O(\log n)$
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - PQ-Sort and Heapsort
   - Intro for the Selection Problem
Sorting using heaps

- Recall: Any priority queue can be used to sort in time
  \[ O(n + n \cdot \text{insert} + n \cdot \text{deleteMax}) \]

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

  ```
  PQ-SortWithHeaps(A)
  1. initialize H to an empty heap
  2. for k ← 0 to n − 1 do
  3. \hspace{1em} H.insert(A[k])
  4. for k ← n − 1 down to 0 do
  5. \hspace{1em} A[k] ← H.deleteMax()
  ```
Sorting using heaps

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

\[ O(n + n \cdot \text{insert} + n \cdot \text{deleteMax}) \]

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

\[
\text{PQ-SortWithHeaps}(A) \\
1. \text{initialize } H \text{ to an empty heap} \\
2. \text{for } k \leftarrow 0 \text{ to } n - 1 \text{ do} \\
3. \quad H.\text{insert}(A[k]) \\
4. \quad \text{for } k \leftarrow n - 1 \text{ down to } 0 \text{ do} \\
5. \quad A[k] \leftarrow H.\text{deleteMax}() \\
\]

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

\[ \leadsto \text{PQ-Sort using heaps takes } O(n \log n) \text{ time.} \]

- Can improve this with two simple tricks:
  1. Heaps can be built faster if we know all input in advance.
  2. Can use the same array for input and heap. \[ \leadsto O(1) \text{ additional space!} \]

\[ \rightarrow \text{Heapsort} \]
Problem statement: Given $n$ items (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 1:

1. Initialize $H$ as an empty heap
2. for $i \leftarrow 0$ to $\text{size}(A) - 1$ do

This corresponds to doing fix-ups.

Worst-case running time: $\Theta(n \log n)$. 

Biedl, Petrick, Veksler (SCS, UW)
Building Heaps by Bubble-up

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

**Solution 1:** Start with an empty heap and insert items one at a time:

\[
\text{simpleHeapBuilding}(A)
\]

\[
A: \text{an array}
\]

1. initialize \( H \) as an empty heap
2. \textbf{for } \( i \leftarrow 0 \) \textbf{to } \text{size}(A) - 1 \textbf{ do}
3. \( H.insert(A[i]) \)
Building Heaps by Bubble-up

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

**Solution 1:** Start with an empty heap and insert items one at a time:

```plaintext
simpleHeapBuilding(A)
A: an array
1. initialize $H$ as an empty heap
2. for $i \leftarrow 0$ to $size(A) - 1$ do
3.     $H.insert(A[i])$
```

This corresponds to doing *fix-ups*

Worst-case running time: $\Theta(n \log n)$. 
Building Heaps by Bubble-down

**Problem statement:** Given $n$ items (in $A[0 \cdots n - 1]$) build a heap containing all of them.
Building Heaps by Bubble-down

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

Solution 2: Using \textit{fix-downs} instead:

\begin{verbatim}
heapify(A)
A: an array
1. \( n \leftarrow A.size() \)
2. \( \text{for } i \leftarrow \text{parent} (\text{last}(n)) \text{ downto } 0 \text{ do} \)
3. \( \text{fix-down}(A, n, i) \)
\end{verbatim}

A careful analysis yields a worst-case complexity of \( \Theta(n) \).

A heap can be built in linear time.
Building Heaps by Bubble-down

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

**Solution 2:** Using *fix-downs* instead:

\[
\text{heapify}(A) \\
A: \text{an array} \\
1. \quad n \leftarrow A.\text{size}() \\
2. \quad \text{for } i \leftarrow \text{parent}(\text{last}(n)) \text{ downto } 0 \text{ do} \\
3. \quad \quad \text{fix-down}(A, n, i)
\]

A careful analysis yields a worst-case complexity of \( \Theta(n) \).
A heap can be built in linear time.
heapify example
heapify example
heapify example
heapify example
heapify example
heapify example

10

80

70

40 30

20

50

60

10
heapify example
heapify example
heapify example
HeapSort

- Idea: **PQ-Sort** with heaps.
- But: Use same input-array $A$ for storing heap.

```
HeapSort(A, n)
1.   // heapify
2.   n ← A.size()
3.   for i ← parent(last(n)) downto 0 do
4.       fix-down(A, n, i)
5.   // repeatedly find maximum
6.   while n > 1
7.       // do deleteMax
8.       swap items at A[root()] and A[last(n)]
9.       decrease n
10.      fix-down(A, n, root())
```

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

Priority Queues
- Abstract Data Types
- ADT Priority Queue
- Binary Heaps
- Operations in Binary Heaps
- PQ-Sort and Heapsort
- Intro for the Selection Problem
Selection

**Problem Statement:** The $k$th-max problem asks to find the $k$th largest item in an array $A$ of $n$ numbers.

**Solution 1:** Make $k$ passes through the array, deleting the maximum number each time.

**Complexity:** $\Theta(kn)$.

**Solution 2:** First sort the numbers. Then return the $k$th largest number.

**Complexity:** $\Theta(n \log n)$.

**Solution 3:** Scan the array and maintain the $k$ largest numbers seen so far in a min-heap.

**Complexity:** $\Theta(n \log k)$.

**Solution 4:** Make a max-heap by calling $\text{heapify}(A)$. Call $\text{deleteMax}(A)$ $k$ times.

**Complexity:** $\Theta(n + k \log n)$. 