CS 240 – Data Structures and Data Management

Module 2: Priority Queues

T. Biedl    É. Schost    O. Veksler
Based on lecture notes by many previous cs240 instructors

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

Winter 2021

References: Sedgewick 9.1-9.4
Outline

1 Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - \textit{PQ-sort} and \textit{Heapsort}
   - Summary
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - PQ-sort and Heapsort
   - Summary
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.

Items enter the stack at the *top* and are removed from the *top*.

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

Applications: Addresses of recently visited web sites, 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*

Applications: Waiting lines, printer queues

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
  - Summary
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, replacing operation *deleteMax* by *deleteMin*,

Applications: typical “todo” list, simulation systems, sorting
Using a Priority Queue to Sort

1. initialize PQ to an empty priority queue
2. for \( k \leftarrow 0 \) to \( n - 1 \) do
3. \( PQ\).insert(\( A[k], A[k] \)) (priority and item are equal to \( A[k] \))
4. for \( k \leftarrow n - 1 \) down to 0 do
5. \( A[k] \leftarrow PQ\).deleteMax()

run-time \( O(\sum_{0 \leq i < n} \text{insert}(i) + \sum_{0 \leq i < n} \text{deleteMax}(i)) \)
depends on how we implement the priority queue
Using a Priority Queue to Sort

\[ \text{PQ-Sort}(A[0..n - 1]) \]
1. initialize \( PQ \) to an empty priority queue
2. for \( k \leftarrow 0 \) to \( n - 1 \) do
3. \( PQ.insert(A[k], A[k]) \) (priority and item are equal to \( A[k] \))
4. for \( k \leftarrow n - 1 \) down to 0 do
5. \( A[k] \leftarrow PQ.deleteMax() \)

- run-time \( O(\sum_{0 \leq i < n} \text{insert}(i) + \sum_{0 \leq i < n} \text{deleteMax}(i)) \)
- depends on how we implement the priority queue
Realizations of Priority Queues

**Realization 1**: unsorted arrays

- **insert**: $O(1)$
- **deleteMax**: $O(n)$
Realizations of Priority Queues

**Realization 1**: 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

**Realization 1**: 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.

*PQ-sort* with this realization yields *selection sort*, so runtime is

$$O\left(\sum_{i<n} i\right) = O(n^2)$$
Realizations of Priority Queues

**Realization 2**: sorted arrays

- **insert**: $O(n)$
- **deleteMax**: $O(1)$
Realizations of Priority Queues

**Realization 2**: sorted arrays

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

Using sorted linked lists is identical.

*PQ-sort* with this realization yields *insertion sort*, runtime is

$$O\left(\sum_{i<n} i\right) = O(n^2)$$
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - *PQ-sort* and *Heapsort*
   - Summary
Realization 3: Heaps

A **(binary) heap** is a certain type of binary tree.

You should know:

- A **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
  \[ \log(n + 1) - 1 \in \Omega(\log n). \]
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 $\Theta(\log n)$. 

---

Biedl, Schost, Veksler (SCS, UW) 
CS240 – Module 2 
Winter 2021 
10 / 26
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 $\Theta(\log n)$.
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>)
```
Storing Heaps in Arrays

Heaps should \textit{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 \textit{level-by-level} from top to bottom, in each level left-to-right.
It is easy to navigate the heap using this array representation:

- the *root* node is at index 0
  (We use “node” and “index” interchangeably in this implementation.)
- the *left child* of node $i$ (if it exists) is node $2i + 1$
- the *right child* of node $i$ (if it exists) is node $2i + 2$
- the *parent* of node $i$ (if it exists) is node $\lfloor \frac{i-1}{2} \rfloor$
- the *last* node is $n - 1$
It is easy to navigate the heap using this array representation:

- the *root* node is at index 0
  
  (We use “node” and “index” interchangeably in this implementation.)
- the *left child* of node $i$ (if it exists) is node $2i + 1$
- the *right child* of node $i$ (if it exists) is node $2i + 2$
- the *parent* of node $i$ (if it exists) is node $\lfloor \frac{i - 1}{2} \rfloor$
- the *last* node is $n - 1$

We 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*
   • Summary
Insert in Heaps

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

\[
\text{fix-up}(A, k) \quad \text{where } k \text{ is an index corresponding to a node of the heap}.
\]

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

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

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

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

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

\[k: \text{an index corresponding to a node of the heap}\]

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

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

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

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

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

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

```
50
/   \
/    /
29   34
|    |
27   15
|    |
23   26
|    |
8    10
```
fix-up example
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:

```python
def fix_down(A, n, k):
    while k is not a leaf:
        j = left child of k
        if (j is not last 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 *fix-down*:

```
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
```

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

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

Priority Queue Realization Using Heaps

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

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

**deleteMax()**
1. $\ell \leftarrow last(size)$
2. swap $A[root()]$ and $A[\ell]$
3. decrease $size$
4. $fix-down(A, size, root())$
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*
   - Summary
Sorting using heaps

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

\[
PQ\text{sortWithHeaps}(A)
\]

1. initialize \( H \) to an empty heap
2. for \( k \leftarrow 0 \) to \( n - 1 \) do
3. \( H.\text{insert}(A[k]) \) (we just insert keys, no items)
4. for \( k \leftarrow n - 1 \) down to 0 do
5. \( A[k] \leftarrow H.\text{deleteMax}() \)
Sorting using heaps

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

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

  1. initialize \( H \) to an empty heap
  2. for \( k \leftarrow 0 \) to \( n - 1 \) do
  3. \( H.insert(A[k]) \) (we just insert keys, no items)
  4. for \( k \leftarrow n - 1 \) down to \( 0 \) do
  5. \( A[k] \leftarrow H.deleteMax() \)

- Recall: runtime is

  \[
  O\left( \sum_{0 \leq i < n} \text{insert}(i) + \sum_{0 \leq i < n} \text{deleteMax}(i) \right)
  \]

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

  \(\Rightarrow\) \( \text{PQ-Ssort} \) using heaps takes \( O(n \log n) \) time.

- Can improve this with two simple tricks \(\rightarrow\) \textbf{Heapsort}

  1. Heaps can be built faster if we know all input in advance.
  2. Can use the same array for input and heap. \(\Rightarrow\) \( O(1) \) auxiliary space!
Building Heaps with Fix-up

**Problem:** Given \( n \) items all at once (in \( A[0 \cdots n - 1] \)) build a heap containing all of them.
Building Heaps with Fix-up

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

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

```
simpleHeapBuilding(A)
A: an array
1. initialize $H$ as an empty heap
2. for $i \leftarrow 0$ to size($A$) – 1 do
```

Worst-case running time: $\Theta(n \log n)$ (we proved $O()$, $\Omega()$ is an exercise).
Building Heaps with Fix-up

**Problem:** Given $n$ items all at once (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
```

This corresponds to doing *fix-ups*
Worst-case running time: $\Theta(n \log n)$ (we proved $O(\ )$, $\Omega(\ )$ is an exercise)
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
1.   $n \leftarrow A.size()$
2.   for $i \leftarrow \text{parent}(\text{last}(n))$ downto 0 do
3.       fix-down(A, n, i)
```

A careful analysis yields a worst-case complexity of $\Theta(n)$. A heap can be built in linear time.
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
1. \( n \leftarrow A.size() \)
2. for \( i \leftarrow \text{parent}(\text{last}(n)) \) downto 0 do
3. \( \text{fix-down}(A, n, i) \)
```

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

```
        10
       / \
     80   60
    /    /  \
  70    50  10
   / \
40  30
```
heapify example

```
    80
   / \  
  70 20
 |   /  
40 30 50
    /  
   60 10
      /
     10

Biedl, Schost, Veksler (SCS, UW)
CS240 – Module 2
Winter 2021
```
heapify example
heapify example
heapify example
heapify example
HeapSort

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

```plaintext
HeapSort(A, n)
1. // heapify
2. $n \leftarrow A.size()$
3. for $i \leftarrow parent(last(n))$ downto 0 do
4.   fix-down(A, n, i)
5. // repeatedly find maximum
6. while $n > 1$
7. // delete the maximum
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.
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:

```
70 40 30 10 80 20 60 50 10
```

The array (i.e., the heap in level-by-level order) is now in sorted order.
Heapsort example

Continue with the example from heapify:
Continue with the example from heapify:

The array (i.e., the heap in level-by-level order) is now in sorted order.
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:
Heapsort example

Continue with the example from heapify:

The array (i.e., the heap in level-by-level order) is now in sorted order.
Heapsort example

Continue with the example from heapify:

![Heap diagram]

The array (i.e., the heap in level-by-level order) is now in sorted order.
Outline

Priority Queues
- Abstract Data Types
- ADT Priority Queue
- Binary Heaps
- Operations in Binary Heaps
- \textit{PQ-sort} and \textit{Heapsort}
- Summary
Heap summary

- **Binary heap**: A binary tree that satisfies structural property and heap-order property.
- Heaps are one possible realization of ADT PriorityQueue:
  - `insert` takes time \(O(\log n)\)
  - `deleteMax` takes time \(O(\log n)\)
  - Also supports `findMax` in time \(O(1)\)
- A binary heap can be built in linear time.
- **PQ-sort** with binary heaps leads to a sorting algorithm with \(O(n \log n)\) worst-case run-time (\(\leadsto\) HeapSort)
- We have seen here the *max-oriented version* of heaps (the maximum priority is at the root).
- There exists a symmetric *min-oriented version* that supports *insert* and *deleteMin* with the same run-times.
Finding the smallest item

**Problem:** Find the *kth smallest item* in an array $A$ of $n$ distinct numbers.

**Solution 1:** Make $k$ passes through the array, deleting the minimum number each time.
Complexity: $\Theta(kn)$.

**Solution 2:** Sort $A$, then return $A[k-1]$.
Complexity: $\Theta(n \log n)$.

**Solution 3:** Scan the array and maintain the $k$ smallest numbers seen so far in a max-heap
Complexity: $\Theta(n \log k)$.

**Solution 4:** Create a min-heap with $\text{heapify}(A)$. Call $\text{deleteMin}(A)$ $k$ times.
Complexity: $\Theta(n + k \log n)$.