Module 2: Priority Queues

Mark Petrick

Based on lecture notes by many previous cs240 instructors

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

Fall 2020

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
   - Towards the Selection Problem
Outline

1 Priority Queues
   • Abstract Data Types
   • ADT Priority Queue
   • Binary Heaps
   • Operations in Binary Heaps
   • \textit{PQ-sort} and \textit{Heapsort}
   • Towards 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**)

---

Petrick  (SCS, UW)  
CS240 – Module 2  
Fall 2020
Stack ADT

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

- **push:** inserting an item
- **pop:** removing (and typically returning) 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 (and typically returning) 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
   • Towards the Selection Problem
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 and returning the item of highest priority

There is also **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

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

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

Realization 2: sorted arrays

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

Using sorted linked lists is identical. 

*PQ-sort* with this realization yields *insertion sort*. 
Outline

1 Priority Queues
   • Abstract Data Types
   • ADT Priority Queue
   • Binary Heaps
   • Operations in Binary Heaps
   • $PQ$-sort and Heapsort
   • Towards the Selection Problem
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 $\log(n + 1) - 1 \in \Omega(\log n)$. 
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 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.
Heaps in Arrays – Navigation

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 $\left\lfloor \frac{i-1}{2} \right\rfloor$
- The last node is $n - 1$

We should hide implementation details using helper-functions!

- Functions $\text{root}()$, $\text{parent}(i)$, $\text{last}(n)$, etc.
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - PQ-sort and Heapsort
   - Towards the Selection Problem
**Insert in Heaps**

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

```
fix-up(A, i)
i: an index corresponding 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)
```

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

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

---

*Petrick (SCS, UW)  
CS240 – Module 2  
Fall 2020  
13 / 25*
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 \texttt{fix-down}:

\begin{center}
\begin{small}
\begin{verbatim}
def fix-down(A, n, 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 ← left child of i
        if (j is not last(n) and A[j+1].key > A[j].key)
            j ← j + 1
        if A[i].key ≥ A[j].key break
        swap A[j] and A[i]
        i ← j
\end{verbatim}
\end{small}
\end{center}

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

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

Petrick (SCS, UW)
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*
   • Towards the Selection Problem
Sorting using heaps

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

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

\[
PQsortWithHeaps(A)
\]

\begin{enumerate}
\item initialize \( H \) to an empty heap
\item \textbf{for} \( i \leftarrow 0 \) \textbf{to} \( n - 1 \) \textbf{do}
\item \( H.insert(A[i]) \)
\item \textbf{for} \( i \leftarrow n - 1 \) \textbf{down to} \( 0 \) \textbf{do}
\item \( A[i] \leftarrow H.deleteMax() \)
\end{enumerate}

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

\( \leadsto \) **PQ-Sort** using heaps takes \( O(n \log n) \) time.

- Can improve this with two simple tricks \( \rightarrow \) **Heapsort**
  \begin{enumerate}
  \item Heaps can be built faster if we know all input in advance.
  \item Can use the same array for input and heap. \( \leadsto O(1) \) auxiliary space!
  \end{enumerate}
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:

```
simpleHeapBuilding(A)
A: an array
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: $\Theta(n \log n)$. 
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:

```plaintext
heapify(A)
A: an array
1. $n \leftarrow A.size()$
2. for $i \leftarrow parent(last(n))$ downto 0 do
3. \hspace{1em} 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

```
10  80
 |
30  40  70
```

- Petrick (SCS, UW)
- CS240 – Module 2
- Fall 2020
HeapSort

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

\[
\text{HeapSort}(A, n)
\]

1. // heapify
2. $n \leftarrow A.size()$
3. for $i \leftarrow \text{parent}(\text{last}(n))$ 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.
Continue with the example from heapify:
Heap summary

- Binary heap: A binary tree that satisfies structural property and heap-order property.
  - *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 (⇝ *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.
Outline

1. Priority Queues
   - Abstract Data Types
   - ADT Priority Queue
   - Binary Heaps
   - Operations in Binary Heaps
   - PQ-sort and Heapsort
   - Towards the Selection Problem
Finding the largest items

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

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

**Solution 2:** Sort $A$, then return $A[n-k]$.
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:** Create a max-heap with $\text{heapify}(A)$. Call $\text{deleteMax}(A)$ $k$ times.
Complexity: $\Theta(n + k \log n)$. 