Module 2: Priority Queues

CS 240 - Data Structures and Data Management

Shahin Kamali, Yakov Nekrich, Olga Zorin

Based on lecture notes by many previous cs240 instructors

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

Spring 2015

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*)

Dynamic Arrays

Linked lists support $O(1)$ insertion/deletion,
but element access costs $O(n)$.

Arrays support $O(1)$ element access, but insertion/deletion cost $O(n)$.

Dynamic Arrays

Linked lists support $O(1)$ insertion/deletion,
but element access costs $O(n)$.

Arrays support $O(1)$ element access, but insertion/deletion cost $O(n)$.

Dynamic arrays offer a compromise:
$O(1)$ element access, and $O(1)$ insertion/deletion *at the end*.

Two realizations of dynamic arrays:

- Allocate one HUGE array, and only use the first part of it.
- Allocate a small array initially, and double its size as needed. (*Amortized analysis* is required to justify the $O(1)$ cost for insertion/deletion at the end — take CS 341/466!)

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

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*.

Applications: typical "todo" list, simulation systems

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*.

Using a Priority Queue to Sort

$PQ - Sort(A)$
1. initialize PQ to an empty priority queue
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad PQ.insert(A[i], A[i])$
4. **for** $i \leftarrow 0$ **to** $n-1$ **do**
5. $\quad A[n-1-i] \leftarrow PQ.deleteMax()$

Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

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

Using unsorted linked lists is identical.

Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

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

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: $O(n)$

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: $O(n)$

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.

Realizations of Priority Queues

Attempt 1: Use *unsorted arrays*

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

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*.

Third Realization: Heaps

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

Recall binary trees:
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.* .

Heaps

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, *key* (priority) 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

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, *key* (priority) 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: Height of a heap with n nodes is $\Theta(\log n)$.

Storing Heaps in Arrays

Let H be a heap (binary tree) 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

Let H be a heap (binary tree) 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.

It is easy to find parents and children using this array representation:

- 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 $A[\lfloor \frac{i-1}{2} \rfloor]$ ($A[0]$ is the root node).

Insertion in Heaps

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

Insertion in Heaps

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

> *bubble-up*(v)
> *v: a node of the heap*
> 1. **while** $parent(v)$ exists **and** $key(parent(v)) < key(v)$ **do**
> 2. swap v and $parent(v)$
> 3. $v \leftarrow parent(v)$

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 *bubble-up*:

bubble-up(v)
v: a node of the heap

while $parent(v)$ exists **and** $key(parent(v)) < key(v)$ **do**
    swap v and $parent(v)$
    $v \leftarrow parent(v)$

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

Time: O(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 *bubble-down*:

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 *bubble-down*:

bubble-down(v)
v: a node of the heap

```
while v is not a leaf do
    u ← child of v with largest key
    if key(u) > key(v) then
        swap v and u
        v ← u
    else
        break
```

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

Priority Queue Realization Using Heaps

heapInsert(A, x)
A: an array-based heap, x: a new item
1. $size(A) \leftarrow size(A) + 1$
2. $A[size(A) - 1] \leftarrow x$
3. $bubble\!-\!up(A, size(A) - 1)$

heapDeleteMax(A)
A: an array-based heap
1. $max \leftarrow A[0]$
2. $swap(A[0], A[size(A) - 1])$
3. $size(A) \leftarrow size(A) - 1$
4. $bubble\!-\!down(A, 0)$
5. **return** *max*

Insert and deleteMax: $O(\log n)$

Building Heaps

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

Building Heaps

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:

heapify1(A)
A: an array
1. initialize H as an empty heap
2. **for** $i \leftarrow 0$ **to** *size*$(A) - 1$ **do**
3. *heapInsert*$(H, A[i])$

Building Heaps

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:

heapify1(A)
A: an array
1. initialize H as an empty heap
2. **for** $i \leftarrow 0$ **to** *size*$(A) - 1$ **do**
3. *heapInsert*$(H, A[i])$

This corresponds to going from $0 \cdots n-1$ in A and doing *bubble-ups*
Worst-case running time: $\Theta(n \log n)$.

Building Heaps

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

Building Heaps

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

Solution 2: Using *bubble-downs* instead:

heapify(A)
A: an array
1. $n \leftarrow size(A) - 1$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 0 **do**
3. $\quad bubble{-}down(A, i)$

Building Heaps

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

Solution 2: Using *bubble-downs* instead:

heapify(A)
A: an array
1. $n \leftarrow size(A) - 1$
2. **for** $i \leftarrow \lfloor n/2 \rfloor$ **downto** 0 **do**
3. $\quad bubble\text{-}down(A, i)$

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

HeapSort

```
HeapSort(A)
    initialize H to an empty heap
    for i ← 0 to n − 1 do
        heapInsert(H, A[i])
    for i ← 0 to n − 1 do
        A[n − 1 − i] ← heapDeleteMax(H)
```

HeapSort

```
HeapSort(A)
    initialize H to an empty heap
    for i ← 0 to n − 1 do
        heapInsert(H, A[i])
    for i ← 0 to n − 1 do
        A[n − 1 − i] ← heapDeleteMax(H)
```

```
HeapSort(A)
    heapify(A)
    for i ← 0 to n − 1 do
        A[n − 1 − i] ← heapDeleteMax(A)
```

HeapSort

HeapSort(A)
1. initialize H to an empty heap
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad$ $heapInsert(H, A[i])$
4. **for** $i \leftarrow 0$ **to** $n-1$ **do**
5. $\quad$ $A[n-1-i] \leftarrow heapDeleteMax(H)$

HeapSort(A)
1. $heapify(A)$
2. **for** $i \leftarrow 0$ **to** $n-1$ **do**
3. $\quad$ $A[n-1-i] \leftarrow heapDeleteMax(A)$

Running time of HeapSort: $O(n \log n)$

Selection

Problem Statement: The kth-max problem asks to find the *kth 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 kth 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 *heapify*(A). Call *deleteMax*(A) k times.
Complexity: $\Theta(n + k \log n)$.