CS 240 – Data Structures and Data Management

Module 2: Priority Queues

A. Storjohann

Based on lecture notes by many previous cs240 instructors

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

Fall 2018

References: Sedgewick 9.1-9.4

version 2018–09–24 10:05

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

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** $k \leftarrow 0$ **to** $n-1$ **do**
3. $\quad PQ.insert(A[k])$
4. **for** $k \leftarrow n-1$ **down to** 0 **do**
5. $\quad A[k] \leftarrow PQ.deleteMax()$

- 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*

- 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

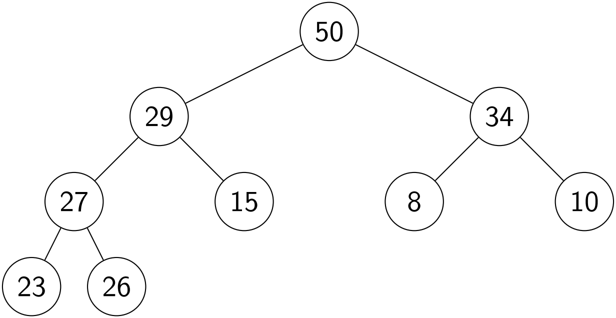
Third Realization: Heaps

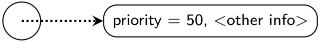
A *(binary) heap* is a certain type of binary tree.

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 $\log(n+1) - 1 \in \Omega(\log n)$.

Example Heap



(In our examples we only show the priorities, and we show them directly in the node. A more accurate picture would be 

)

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.

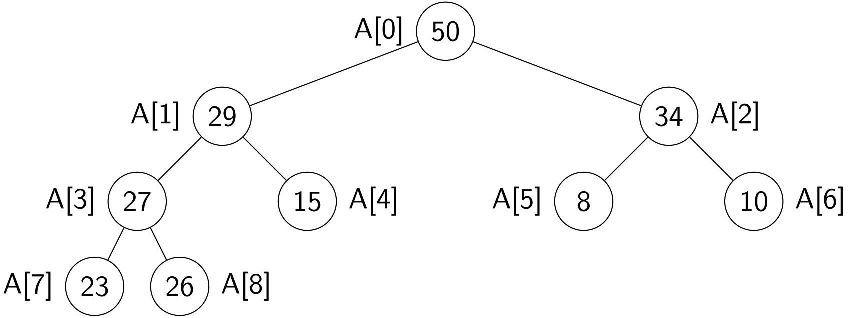
A *min-heap* is the same, but with opposite order property.

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 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 $A[n-1]$

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

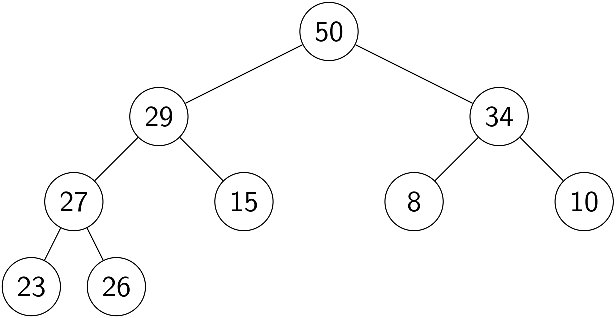
- Place the new key at the first free leaf
- The heap-order property might be violated: perform a *fix-up*:

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

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



deleteMax in Heaps

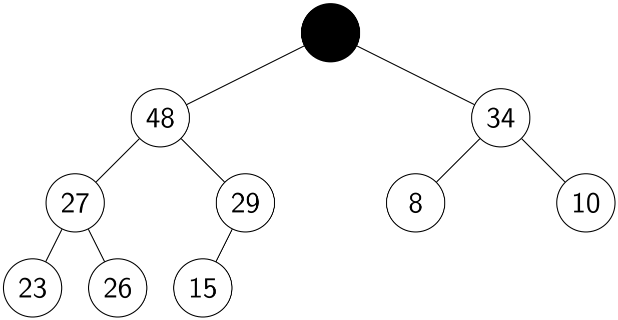
- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a *fix-down*:

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

    while k is not a leaf **do**
        // Find the child with the larger key
        $j \leftarrow$ left child of k
        if (j is not *last*(n) and $A[j+1] > A[j]$)
            $j \leftarrow j+1$
        if $A[k] \geq A[j]$ **break**
        swap $A[j]$ and $A[k]$
        $k \leftarrow j$

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

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. $\ell \leftarrow last(size)$
3. $A[\ell] \leftarrow x$
4. $\textit{fix-up}(A, \ell)$

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

$$O(n + n \cdot \mathit{insert} + n \cdot \mathit{deleteMax})$$

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

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

$\rightsquigarrow$ *PQ-Sort* using heaps takes $O(n \log n)$ 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. $\rightsquigarrow$ *$O(1)$ additional space!* $\rightarrow$ *Heapsort*

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:

```
simpleHeapBuilding(A)
A: an array
1.    initialize H as an empty heap
2.    for i ← 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

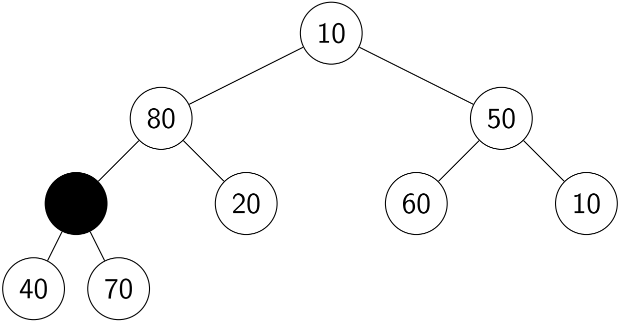
Problem statement: Given n items (in $A[0 \cdots n-1]$) build a heap containing all of them.

Solution 2: Using *fix-downs* instead:

heapify(A)
A: an array
1. $n \leftarrow A.size()$
2. **for** $i \leftarrow parent(last(n))$ **downto** 0 **do**
3. $\quad$ *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



HeapSort

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

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

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

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

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