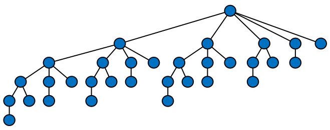
Lecture 3. Priority Queues

CpSc 8400: Algorithms and Data Structures

Brian C. Dean

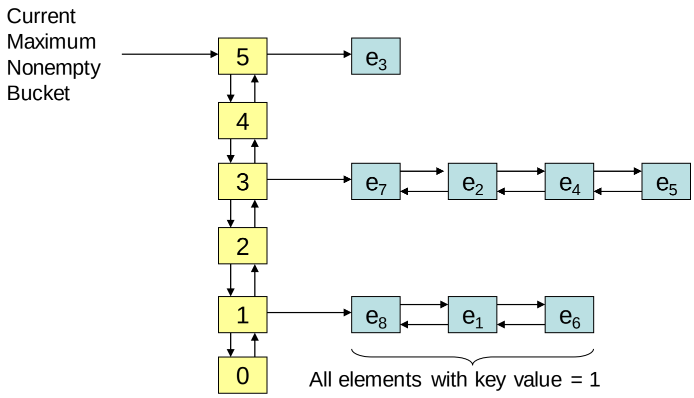


School of Computing
Clemson University
Spring, 2016

Warm-Up: Incremental Priority Queues

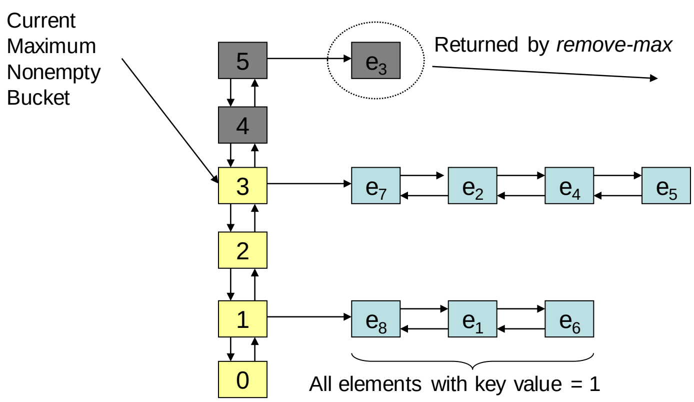
- Fundamental operations of a (max-) priority queue:
  – *Insert :* insert new element
  – *Remove-max* : remove element with maximum key
- We'll study general priority queues in a moment, but for now, consider the special case of an *incremental* priority queue:
  – Keys stored in the structure are nonnegative integers, initially zero.
  – We additionally support an *increment-priority(e)* operation that takes a pointer to an element and increases its key by 1.

Implementing an Incremental Priority Queue



3

The Remove-Max Operation



4

Analysis of Incremental Priority Queue

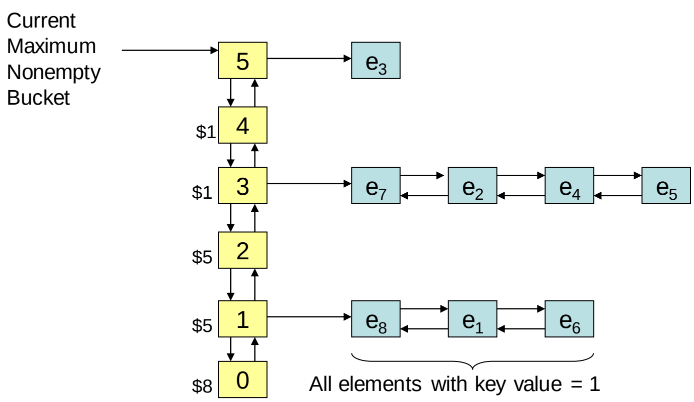
- Let M denote the amount by which the “current maximum bucket” pointer moves.

	Worst-Case Running Time	Amortized Running Time
insert	1	1
increment-priority	1	2
remove-max	1+M (not bounded!)	1

All operations have O(1) amortized running times!

5

Plenty of Credit to Spare…



6

General Priority Queues

- In a simple FIFO queue, elements exit in the same order as they enter.
- In a priority queue, the element with highest priority (usually defined as having *lowest* key) is always the first to exit.
- Many uses:
  - **Scheduling:** Manage a set of tasks, where you always perform the highest-priority task next.
  - **Sorting:** Insert n elements into a priority queue and they will emerge in sorted order.
  - **Complex Algorithms:** For example, Dijkstra's shortest path algorithm is built on top of a priority queue.

7

Priority Queues

- All priority queues support:

  Insert(e, k) : Insert a new element e with key k.

  Remove-Min : Remove and return the element with minimum key.
- In practice (mostly due to Dijsktra's algorithm), many support:

  Decrease-Key(e, Δk) : Given a pointer to element e within the heap, reduce e's key by Δk.
- Some priority queues also support:

  Increase-key(e, Δk) : Increase e's key by Δk.

  Delete(e) : Remove e from the structure.

  Find-min : Return a pointer to the element with minimum key.

8

Redundancies Among Operations

- Given *insert* and *delete*, we can implement *increase-key* and *decrease-key*.
- Given *decrease-key* and *remove-min*, we can implement *delete*.
- Given *find-min* and *delete*, we can implement *remove-min*.
- Given *insert* and *remove-min*, we can implement *find-min.*

9

Priority Queue Implementations

- There are *many* simple ways to implement the abstract notion of a priority queue as a concrete data structure:

	insert	*remove-min*
Unsorted array or linked list	O(1)	O(n)
Sorted array or linked list	O(n)	O(1)
Binary heap	O(log n)	O(log n)
Balanced binary search tree	O(log n)	O(log n)
Skew heap	O(log n) am.	O(log n) am.

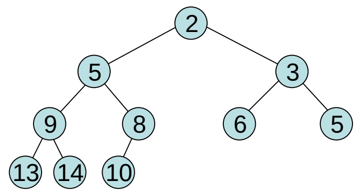
10

The Binary Heap

- An almost-complete binary tree (all levels full except the last, which is filled from the left side up to some point).
- Satisfies the **heap property**: for every element e, key(parent(e)) ≤ key(e).
  - Minimum element always resides at root.
- Physically stored in an array A[0...n-1].
- Easy to move around the array in a treelike fashion:
  - Parent(i) = floor((i-1)/2).
  - Left-child(i) = 2i + 1
  - Right-child(i) = 2i + 2.

	0	1	2	3	4	5	6	7	8	9
A:	2	5	3	9	8	6	5	13	14	10

Actual array representation in memory



Mental picture as a tree

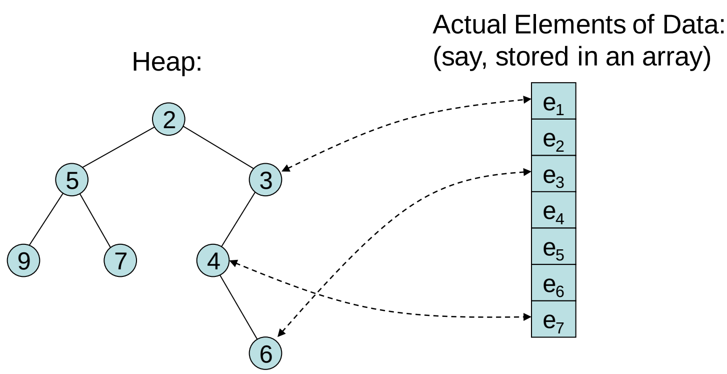
Heap Operations : sift-up and sift-down

- All binary heap operations are built from the two fundamental operations *sift-up* and *sift-down*:
  - *sift-up*(i) : Repeatedly swap element A[i] with its parent as long as A[i] violates the heap property with respect to its parent (i.e., as long as A[i] < A[parent(i)]).
  - *sift-down*(i) : As long as A[i] violates the heap property with one of its children, swap A[i] with its smallest child.
- Both operations run in O(log n) time since the height of an n-element heap is O(log n).
- In some other places, *sift-down* is called *heapify,* and *sift-up* is known as *up-heap*.

Implementing Heap Operations Using sift-up and sift-down

- The remaining operations are now easy to implement in terms of *sift-up* and sift-down:
  - insert : place new element in A[n+1], then sift-up(n+1).
  - *remove-min* : swap A[n] and A[1], then sift-down(1).
  - *decrease-key*(i, Δk) : decrease A[i] by Δk, then sift-up(i).
  - *increase-key*(i, Δk) : increase A[i] by Δk, then sift-down(i).
  - *delete*(i) : swap A[i] with A[n], then sift-up(i), sift-down(i).
- All of these clearly run in O(log n) time.
- General idea: modify the heap, then fix any violation of the heap property with one or two calls to *sift-up* or *sift-down*.

Caveat: You Can't Easily Find Elements In Heaps (Except the Min)



Each record in the data structure keeps a pointer to the physical element of data it represents, and each element of data maintains a pointer to its corresponding record in the data structure.

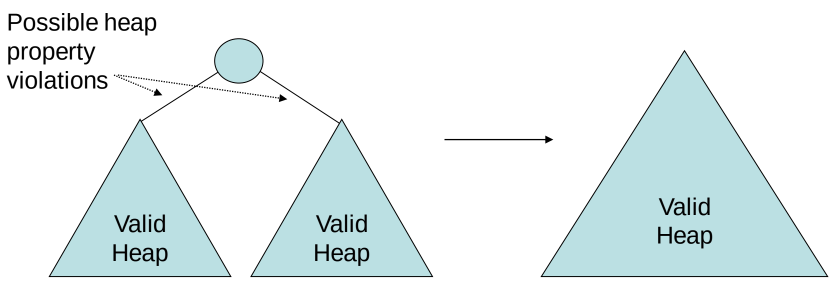
Building a Binary Heap

- We could build a binary heap in O(n log n) time using n successive calls to *insert*.
- Another way to build a heap: start with our n elements in arbitrary order in A[0..n-1], then call *sift-down(i)* for i = n-1 down to 0.
  – Remarkable fact #1: this builds a valid heap!
  – Remarkable fact #2: this runs in only O(n) time!

15

Bottom-Up Heap Construction

- The key property of *sift-down* is that it fixes an isolated violation of the heap property at the root:



- Using induction, it is now easy to prove that our “bottom-up” construction yields a valid heap.

16

Bottom-Up Heap Construction

- To analyze the running time of bottom-up construction, note that:
  - At most n elements reside in the bottom level of the heap. Only 1 unit of work done to them by *sift-down*.
  - At most n/2 elements reside in the 2nd lowest level, and at most 2 units of work are done to each of them.
  - At most n/4 elements reside in the 3rd lowest level, and at most 3 units of work are done to them.
- So total time ≤ T = n + 2(n/2) + 3(n/4) + 4(n/8) + …
  (for simplicity, we carry the sum out to infinity, as this will certainly give us an upper bound).
- Claim: T = 4n = O(n)

17

"Shifting" Technique for Sums

$$
\begin{array}{rll}
 & T & = n + 2(n/2) + 3(n/4) + 4(n/8) + \ldots \\
- & T/2 & = \quad\; n/2 + 2(n/4) + 3(n/8) + \ldots \\
\hline
 & T/2 & = n + n/2 + n/4 + n/8 + \ldots
\end{array}
$$

Applying the same trick again:

$$
\begin{array}{rll}
 & T & = 2n + n + (n/2) + (n/4) + \ldots \\
- & T/2 & = \quad\; n + (n/2) + (n/4) + \ldots \\
\hline
 & T/2 & = 2n
\end{array}
$$

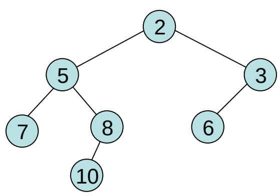
18

Heapsort

- Any priority queue can be used to sort. Just use n *inserts* followed by n *remove-mins.*
- The binary heap gives us a particularly nice way to sort in O(n log n) time, known as **heapsort**:
  - Start with an array A[0..n-1] of elements to sort.
  - Build a heap (bottom up) on A in O(n) time.
  - Call *remove-min* n times.
  - Afterwards, A will end up reverse-sorted (it would be forward-sorted if we had started with a "max" heap)

Another Simple Way to Implement Priority Queues…

- Suppose we store our priority queue in a "heap-ordered" binary tree.
  - Heap property: parent ≤ child.
  - Each node maintains a pointer to its left child and right child.
  - The tree is not necessarily "balanced". It could conceivably be nothing more than a single sorted path.
  - No longer easily mapped to an array, as with a binary heap.

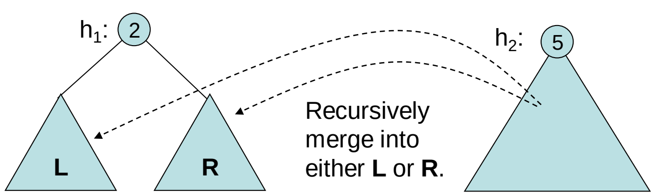


All You Need is Merge…

- Suppose we can *merge* two heap-ordered trees in O(log n) time.
- All priority queue operations now easy to implement in O(log n) time!
  - *insert*: merge with a new 1-element tree.
  - *remove-min*: remove root, merge left & right subtrees.

Merging Two Heap-Ordered Trees

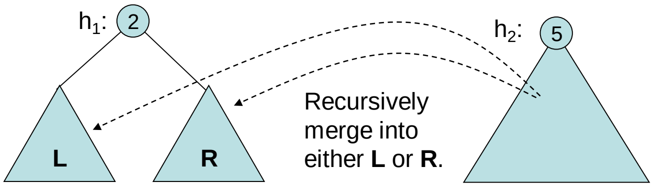
- Take two heap-ordered trees h_1 and h_2, where h_1 has the smaller root.
- Clearly, h_1's root must become the root of the merged tree.
- To complete the merge, recursively merge h_2 into either the left or right subtree of h_1:



- As a base case, the process ends when we merge a heap h_1 with an empty heap, the result being just h_1.

Skew Heaps

- Always merge into R, but after merging h_2 into h_1, just swap h_1's children (so we really alternate between L and R).
- Remarkably, this makes merge (and therefore all other major operations) run in just O(log n) amortized time!



What About Delete / Decrease-Key?

- In a skew heap, *insert* and *remove-min* are based on merging, so they run in O(log n) amortized time.
- For *decrease-key* (and *increase-key*), simply delete an element and re-insert it with a new key.
- Subtle question: how do we implement *delete* efficiently? (so that it doesn't interfere with the amortized analysis of other operations…)