Lecture 5. Priority Queues and Randomized Algorithm Analysis

CpSc 8400: Algorithms and Data Structures
Brian C. Dean

School of Computing
Clemson University
Spring, 2021

1

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.

2

2

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.

3

3

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.*

4

4

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.
Randomized mergeable binary heap	O(log n) whp.	O(log n) whp.
Binomial heap	O(1) am.	O(log n)
Fibonacci heap	O(1) am.	O(log n) am.

5

5

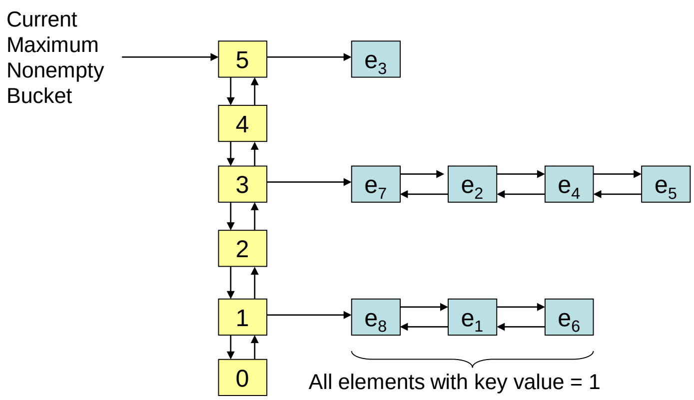
Warm-Up: Incremental Priority Queues

- 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.

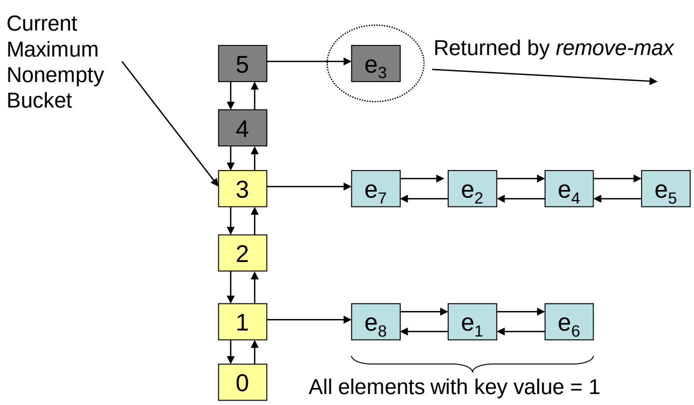
6

6

Implementing an Incremental Priority Queue



The Remove-Max Operation



Analysis of Incremental Priority Queue

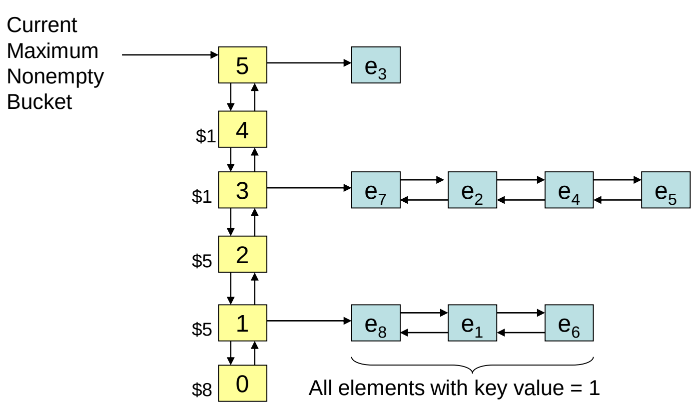
- Let M denote the amount by which the "current maximum bucket" pointer moves.

	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!

9

Plenty of Credit to Spare…



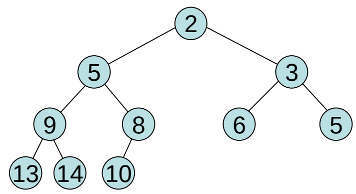
10

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

11

Heap Operations : sift-up and sift-down

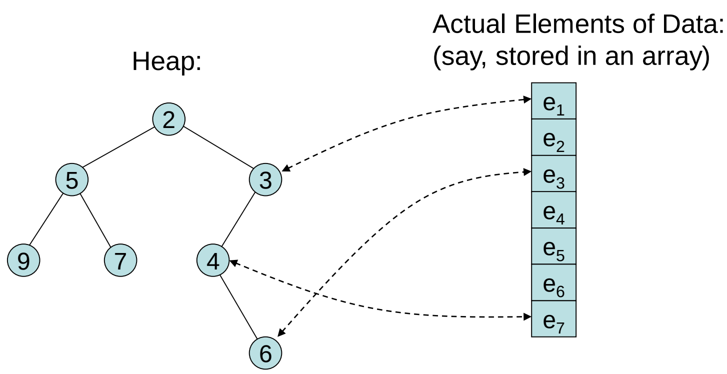
- All binary heap operations are built from the two fundamental operations *sift-up* and *sift-down*:
  - *sift-up*(i) : Repeatedly swap element A[i] with its parent as long as A[i] violates the heap property with respect to its parent (i.e., as long as A[i] < A[parent(i)]).
  - *sift-down*(i) : As long 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*.

12

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

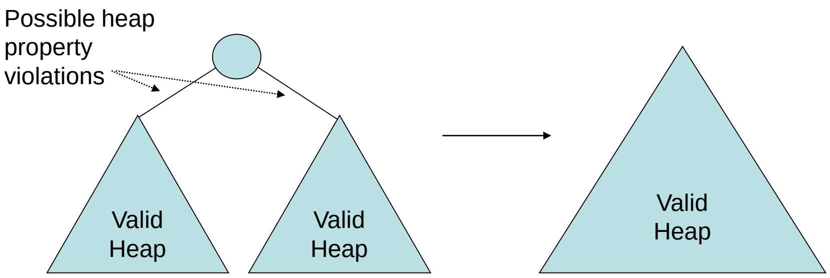
- We could build a binary heap in O(n log n) time using n successive calls to *insert*.
- Another way to build a heap: start with our n elements in arbitrary order in A[0..n-1], then call *sift-down(i)* for i = n-1 down to 0.
  - Remarkable fact #1: this builds a valid heap!
  - Remarkable fact #2: this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

15

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

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 2^{nd} lowest level, and at most 2 units of work are done to each of them.
  - At most n/4 elements reside in the 3^{rd} 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

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

Applying the same trick again:

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

18

Heapsort

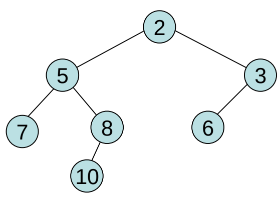
- Any priority queue can be used to sort. Just use n *inserts* followed by n *remove-mins.*
- The binary heap gives us a particularly nice way to sort in O(n log n) time, known as **heapsort**:
  - Start with an array A[0..n-1] of elements to sort.
  - Build a heap (bottom up) on A in O(n) time.
  - 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)

19

19

Recall: An Alternative Method With Simpler(?) Structure…

- 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.



20

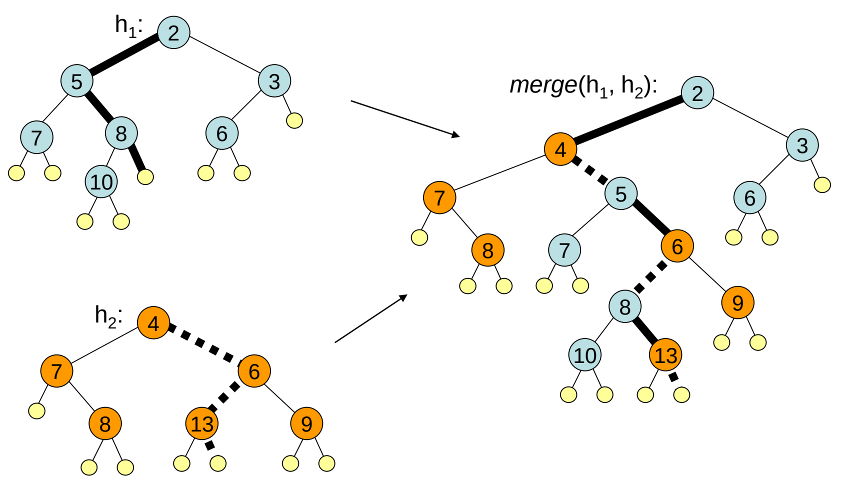
20

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.
  - *decrease-key & increase-key:* delete + re-insert
  - *delete:* replace with merge of two child subtrees

21

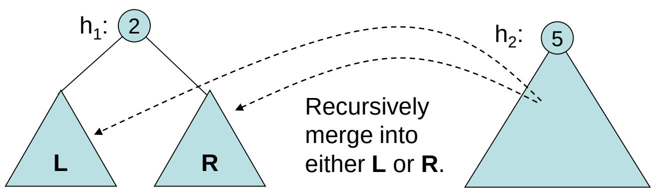
Merging Two Heap-Ordered Trees (Null Path Merging Viewpoint)



22

Merging Two Heap-Ordered Trees (Recursive Viewpoint)

- 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.

23

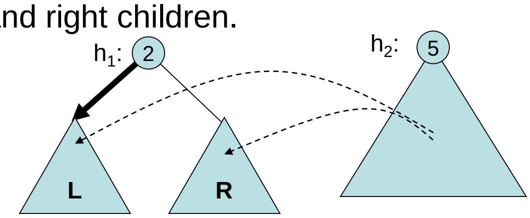
Running Time Analysis

- The time required to merge two heaps along null paths is proportional to the combined lengths of these paths.
- So all we need is a method to find "short" null paths and we will have an efficient merging algorithm.
- Note that every n-node binary tree has a null path of length O(log n).
- There are many ways to find short null paths, each of which leads us to a different mergeable heap data structure…

24

Recall: Skew Heaps

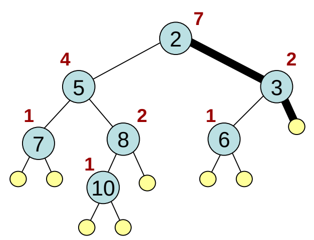
- Merge towards h_1's "preferred child", then toggle preferred child pointer (so alternate merging into left and right subtrees).
  - Equivalently: Always merge into R, but afterwards just swap h_1's children.
  - Equivalently (but more confusing…): Merge two heaps along their "right spines", then walk back up the right spine of the result, and for each element except the lowest, swap left and right children.
- Remarkably, this makes merge run in just O(log n) amortized time!



25

Size-Augmented Mergeable Heaps

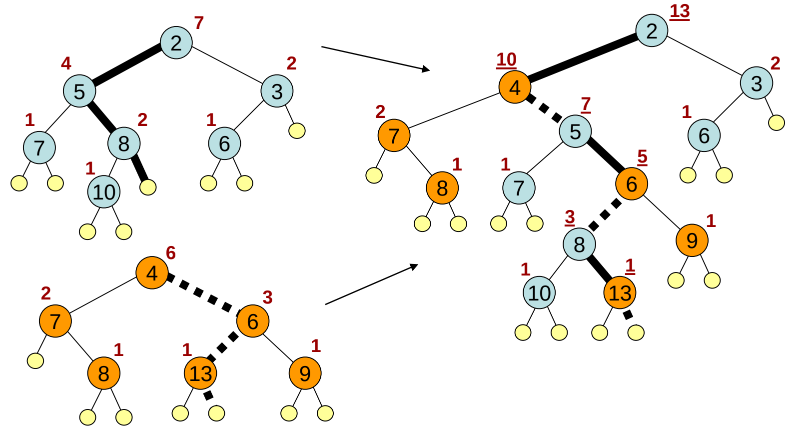
- Let's try to remove the randomness from our preceding approach…
  - Augment each element with the size of its subtree.
  - Now we can find a null path of length O(log n) by repeatedly stepping to whichever child has smaller size.
  - Each step reduces the size of our current subtree by at least a factor of 2.



26

Updating Augmented Information After Merging

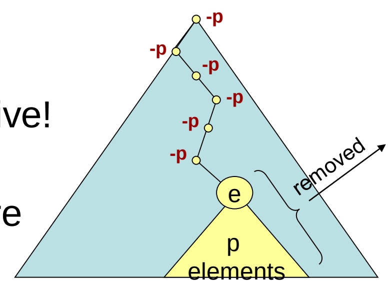
- When we merge two heaps, we walk back up the merge path and update augmented information.



27

Trouble with Decrease-Key Motivating Lazy Delete / Garbage Collection

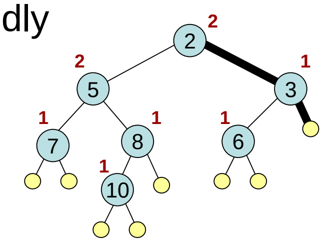
- Recall: *decrease-key(e)* removes the subtree rooted at element e, decreases e's key, and then merges the result back into the main tree.
- When we remove e's subtree, we need to update the augmented size information along the path from the root down to e.
- However, if e is deep in the tree, this can be too expensive!
- This also affects *delete* and *increase-key,* since these are built using *decrease-key*.



28

Augmenting with Null Path Lengths

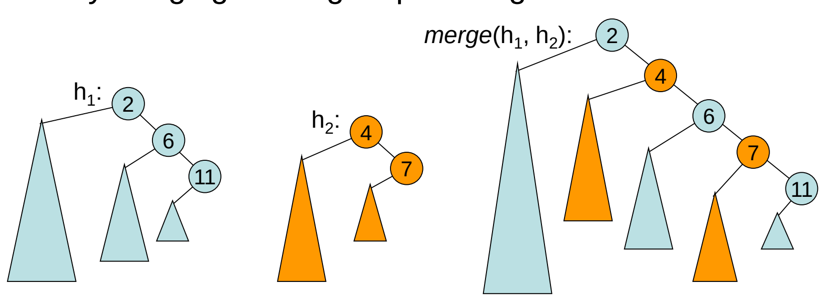
- The **null path length** of element e, npl(e), is the shortest distance from e down to an empty space at the bottom of of e's subtree.
- Suppose we augment every element e in our heap with npl(e).
- Since npl(root) = O(log n), we can find a null path of length O(log n) by repeatedly stepping to a child with the smaller null path length.
- This allows us to *merge* in O(log n) time.



29

Leftist Heaps

- **Leftist property:** npl(left(e)) ≥ npl(right(e)) for all elements e.
- A **leftist heap** is a heap-ordered leftist tree Each element in a leftist heap is augmented with its null path length.
- The shortest null path in a leftist tree (of length O(log n)) is its "right spine", so we can merge two leftist heaps in O(log n) time by merging their right spines together:



30

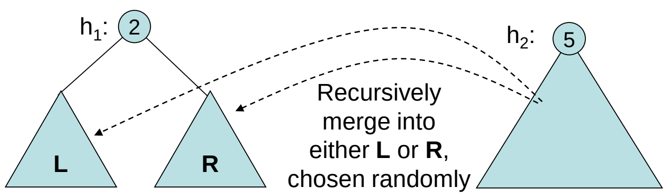
Restoring the Leftist Property

- After merging, we walk back up the right spine of the merged heap…
  - … recalculating npl(e) for each element e.
  - … and swapping the left and right subtrees of any element that now violates the leftist property.
- Therefore, *merge* (hence *insert* and *remove-min*) all take O(log n) time on a leftist heap.
- The same problem exists with *decrease-key, delete*, and *increase-key* as with size-augmented and npl-augmented heaps though.
- **In fact, the leftist heap is really nothing more than an npl-augmented heap where we always treat the child of smaller npl as the left child.**

31

The Randomized Mergeable Binary Heap

- Perhaps the simplest possible idea: choose null paths at random! (i.e., starting from root, repeatedly step left or right, each with probability ½.
- In terms of our recursive outlook for merging h_1 and h_2 (h_1 having the smaller root) this corresponds to the following simple procedure:



- Remarkably, this trivial procedure merges any two heaps in O(log n) time with high probability!

32

Definition : “With High Probability”

- We say an algorithm with input size n runs in O(log n) time **with high probability** if we can find a constant k such that

  Pr[running time > k log n] ≤ X,

  where X is a sufficiently small number.

  but how small should X be?...

Definition : “With High Probability”

- We say an algorithm with input size n runs in O(log n) time **with high probability** if we can find a constant k such that

  Pr[running time > k log n] ≤ X,

  where X is a sufficiently small number.

  but how small should X be?...

  Pr[Struck by lightning in a year] ≈ 10^{-6}

Definition : “With High Probability”

- We say an algorithm with input size n runs in O(log n) time **with high probability** if we can find a constant k such that

  Pr[running time > k log n] ≤ X,

  where X is a sufficiently small number.

  but how small should X be?...

  Pr[Struck by lightning in a year] ≈ 10^{-6}

  Pr[Bitten by shark in a year] ≈ 10^{-7}

35

35

Definition : “With High Probability”

- We say an algorithm with input size n runs in O(log n) time **with high probability** if we can find a constant k such that

  Pr[running time > k log n] ≤ X,

  where X is a sufficiently small number.

  but how small should X be?...

  Pr[Struck by lightning in a year] ≈ 10^{-6}

  Pr[Bitten by shark in a year] ≈ 10^{-7}

  Pr[Hit by a meteor in a year] ≈ 10^{-10}

36

36

Definition : "With High Probability"

- We say an algorithm with input size n runs in O(log n) time **with high probability** if we can find a constant k such that

  Pr[running time > k log n] ≤ X,

  where X is a sufficiently small number.

  but how small should X be?...

  Pr[Struck by lightning in a year] ≈ 10^{-6}

  Pr[Bitten by shark in a year] ≈ 10^{-7}

  Pr[Hit by a meteor in a year] ≈ 10^{-10}

  Pr[All 3 in the same year!] ≈ 10^{-23}

37

37

Definition : "With High Probability"

- We say an algorithm with input size n runs in O(log n) time **with high probability** if, for any constant c, we can find another constant k such that

  Pr[running time > k log n] ≤ 1 / n^c.

- That is, the probability we fail to run in O(log n) time is at most 1 / n^c, for any constant c of our choosing (as long as we choose a sufficiently large hidden constant in the O(log n) notation).
- We'll discuss and motivate this definition in more detail later in the course.

38

38

Example: Boosting Success Probability via Independent Repetition

- Take a randomized algorithm that fails with probability ≤ ½. (and that we can detect failures).
- Run it k times: probability of failure drops to $\leq 1/2^k$.
- Run it $k = c \log n$ times: the probability of failure drops to $\leq 1/n^c$ (i.e., the algorithm succeeds with high probability!)

39