# EE360C: Algorithms

## Priority Queues

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

# Recap

# Efficient Algorithms

- Seek algorithms that are quantitatively better than brute force search.
- Seek algorithms that are polynomial time.

Once we find this efficient algorithm, we can further improve runtime by taking care of the implementation details, sometimes through complex data structures.

# Motivation

# Motivation: Stable Marriage

The stable marriage algorithm needs a data structure that maintains the dynamically changing set of all free men. The algorithm needs to be able to:

- add elements to the set
- delete elements from the set
- select an element from the set, based on some assigned *priority*

# Priority Queues

A priority queue is a data structure that maintains a set of elements $S$, where each element $v \in S$ has an associated value $\mathtt{key}(v)$ that denotes the priority of the element $v$. Smaller keys represent higher priority.

Operations on a priority queue

- Adding an element.
- Deleting an element.
- Selection of an element with the smallest key.

# Example: Schedule Processes on a Computer

- Each process has a priority
- Processes do not arrive in order of priority
- When ready, we want to extract the process with the highest priority or key with lowest value.

# Motivation: Sort a List of Numbers

**Sort**

Sort a set of $n$ elements.

**Possible Algorithm**

- Set up a priority queue $H$, and insert each value into $H$ with it's value as the key.
- Repeatedly find the smallest number in $H$, and output it ("find minimum" operation).

- Sort array in $O(n)$ "find minimum" operations.
- Comparison sorting algorithms have $O(n \log n)$ running time. If we want to achieve this bound each "find minimum" step must take $O(\log n)$ time.

# Candidate Data Structures for Priority Queues

The data structure we select must support inserting a new element, finding the minimum element, and deleting the minimum element.

- **List:** Insertion and deletion take $O(1)$ time, but finding the minimum requires scanning the list and takes $\Omega(n)$ time
- **Sorted array:** Finding the minimum takes $O(1)$ time, but locating where to insert or delete element from would take $O(\log n)$, and then inserting/deleting would take $O(n)$ (move all elements).

None of these data structures give us "priority queue" operations of order $O(\log n)$

# Properties of Priority Queue

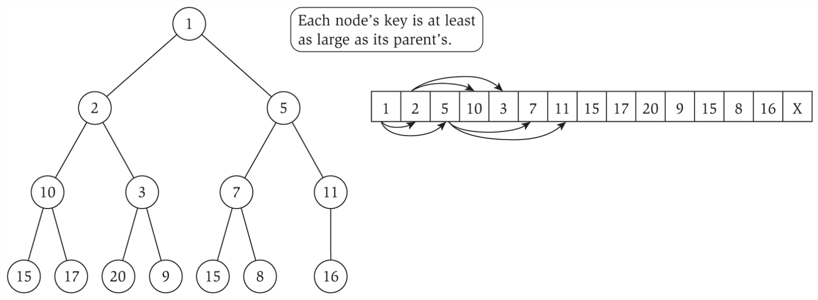
- Store a set $S$ of elements, where each element $v$ has a priority value $\texttt{key}(v)$
- Smaller key values denote higher priorities
- Operations supported:
  - find the element with the smallest key
  - remove the element with the smallest key
  - insert a new element
  - delete an element
- We would like to do these operations in $O(\log n)$.

# Heaps

# Heaps

- Combine the benefits of both lists and sorted arrays
- Conceptually, a heap is a balanced binary tree
- The tree has a root, and each node can have up to two children.
- Heap order: For every element $v$ at node $i$, the element $w$ at $i$'s parent satisfies $\texttt{key}(w) \leq \texttt{key}(v)$

# A Heap Example



# Heaps (contd.)

- We can implement a heap in a pointer-based data structure
- Alternatively, assume a maximum number $N$ of elements is known in advance
- Store nodes of the heap in an array
  - Node at index $i$ has children at indices $2i$ and $2i+1$ and parent at index $\lfloor i/2 \rfloor$
  - Index 1 is the root
  - How do you know that a node at index $i$ is a leaf? If $2i > N$, the number of elements in the heap.

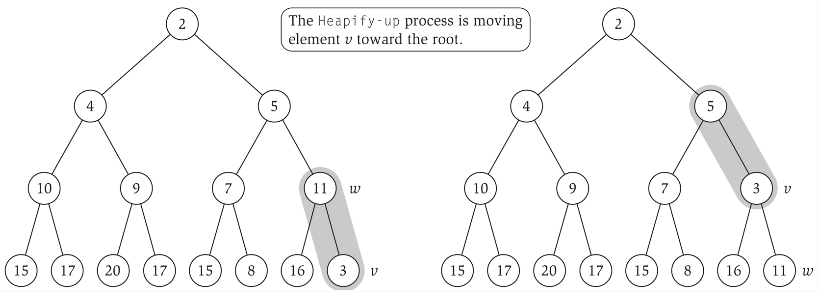
## Inserting an Element: `Heapify-up`

1. Heap $H$ has $n < N$ elements
2. Insert a new element at $i = n + 1$ by setting $H[i] = v$.
3. This may break the heap-order.
4. Fix the heap order using `Heapify-up`$(H, n + 1)$.

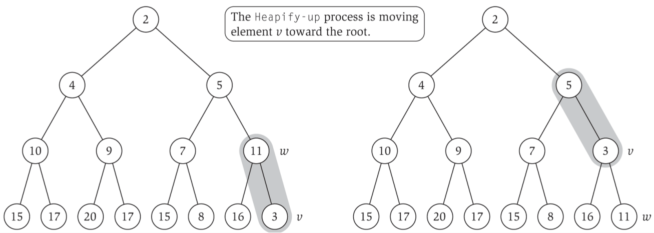
**Algorithm 2.8, page 61**

```
Heapify-up(H,i):
  If i > 1 then
    let j = parent(i) = ⌊i/2⌋
    If key[H[i]]<key[H[j]] then
      swap the array entries H[i] and H[j]
      Heapify-up(H,j)
    Endif
  Endif
```

# `Heapify-Up` **Example**



# Correctness of `Heapify-Up`




- *H* is almost a heap with key of $H[i]$ too small if there is a value $\alpha \geq$ `key`$(H[i])$ such that increasing `key`$(H[i])$ to $\alpha$ makes *H* a heap
- **Claim:** The procedure `Heapify-Up`$(H, i)$ fixes the heap property in $O(\log i)$ time, assuming that the array *H* is almost a heap with the key of $H[i]$ too small.
- **Corollary:** Using `Heapify-Up` we can insert a new element in a heap of *n* elements in $O(\log n)$ time. (Why?)

# Correctness of `Heapify-Up`

**Claim:** The procedure `Heapify-Up`$(H, i)$ fixes the heap property in $O(\log i)$ time, assuming that the array $H$ is almost a heap with the key of $H[i]$ too small.

**Proof:** Prove by induction on $i$.

- Base case: $i = 1$. H[1] is the root, so if it's too small, then H is already a heap.
- Inductive Hypothesis: `Heapify-Up`$(H, j)$, where $j = \lfloor \frac{i}{2} \rfloor$fixes the heap property in $O(\log j)$ time, assuming that the array $H$ is almost a heap with the key of $H[j]$ too small.
- Inductive step: $H$ is almost a heap with key of $H[i]$ too small. Let $j =$`parent`$(i) = \lfloor \frac{i}{2} \rfloor$ and $\beta$ be its key. Swapping the elements at $H[i]$ and $H[j]$ takes $O(1)$ time, and now $H[i] = \beta$. After the swap, $H$ is a heap or almost a heap with the key of $H[j]$ too small, since setting its key to $\beta$ would make $H$ a heap. Finally, by the inductive hypothesis, the recursive call to `Heapify-Up`$(H, j)$ fixes the heap property.

## Correctness of `Heapify-Up` (contd.)

Cost of `Heapify-Up` $(H, i)$

$$
\begin{aligned}
&= \log j + 1 \\
&= \log(\lfloor \tfrac{i}{2} \rfloor) + \log 2 \\
&= \log(2\lfloor \tfrac{i}{2} \rfloor) \\
&= \log i
\end{aligned}
$$

# **Deleting an Element:** `Heapify-down`

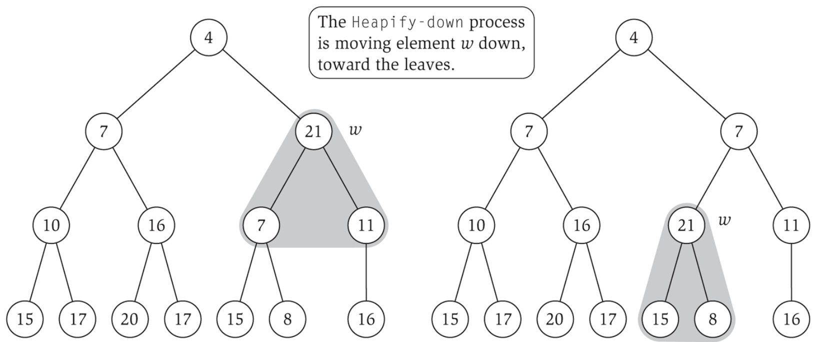
Suppose $H$ has $n+1$ elements

1. Delete element at $H[i]$ by moving element at $H[n+1]$ to $H[i]$
2. If element at $H[i]$ is too small, fix heap order using `Heapify-up`$(H, i)$
3. If element at $H[i]$ is too large, fix heap order using `Heapify-down`$(H, i)$

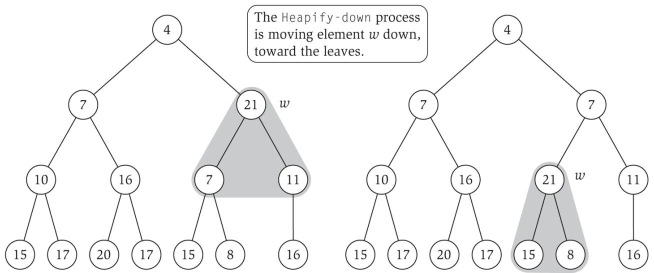
---

```
Heapify-down(H,i):
  Let n = length(H)
  If 2i > n then
    Terminate with H unchanged
  Else if 2i < n then
    Let left = 2i, and right = 2i + 1
    Let j be the index that minimizes key[H[left]] and key[H[right]]
  Else if 2i = n then
    Let j = 2i
  Endif
  If key[H[j]] < key[H[i]] then
     swap the array entries H[i] and H[j]
     Heapify-down(H, j)
```

# `Heapify-down` **Example**



# Heapify-down Correctness




- $H$ is almost a heap with key of $H[i]$ too big if there is a value $\alpha \leq$`key`$(H[i])$ s.t. decreasing `key`$(H[i])$ to $\alpha$ makes $H$ a heap
- **Claim:** The procedure `Heapify-Down`$(H, i)$ fixes the heap property in $O(\log i)$ time, assuming that the array $H$ is almost a heap with the key of $H[i]$ too big.
- **Corollary:** Using `Heapify-Down` we can delete an element from a heap of $n$ elements in $O(\log n)$ time.

# `Heapify-down` **Correctness**

The procedure `Heapify-Down`($H, i$) fixes the heap property in $O(\log n)$ time, assuming that the array $H$ is almost a heap with the key of $H[i]$ too big. **Proof:** Proof by reverse induction on $i$. Suppose $H$ has $n$ elements.

- Base case: $2i > n$. Then $i$ is a leaf, hence $H$ is a heap.
- Inductive step: Let $j$ be the child of $i$ with smaller key value and denote its key value $\beta$. Swapping the elements at $H[i]$ and $H[j]$ takes $O(1)$ time. The resulting array is a heap or almost a heap with $H[j]$ too big, since setting its key to $\beta$ makes it a heap. Since $j \geq 2i$, by the inductive hypothesis, the recursive call to `Heapify-Down` fixes the heap property.

# In Class Exercise 1

**Problem**

Naively, we can build a heap out of an arbitrary array using successive calls to HEAPIFY-DOWN, starting at element $\lfloor \text{length}[H]/2 \rfloor$ and going down to 1. If each call to HEAPIFY-DOWN takes $O(\log n)$ time and we have $O(n/2)$ such calls, we can build a heap in $O(n \log n)$ time. Prove that this process is actually faster than $O(n \log n)$ (i.e., provide a *tighter* bound on the process's running time). Starters:

- What is the height of an $n$-element heap?
- How many nodes are there at height $h$ of an $n$-element heap?

# In Class Exercise 1: continued

What is the height of an $n$-element heap?

$O(\log n)$ (it's a (nearly) complete binary tree).

# In Class Exercise 1: continued

How many nodes are there at height $h$ of an $n$-element heap?

## Key Observation

The number of leaves in a complete binary tree is $\lceil n/2 \rceil$.

## Proposition

In an $n$-element heap, there are $\lceil n/2^{h+1} \rceil$ nodes at height $h$.

## Proof (by induction on $h$)

**Base case:** $h = 0$ (the leaves). This is trivially true from the observation above.

**Inductive step:** Suppose that the claim is true for $h - 1$. Let $N_h$ be the number of nodes at height $h$ in an $n$-node tree $T$. Consider $T'$ formed by removing the leaves of $T$. $T'$ has $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ nodes. Nodes at height $h$ in $T$ are at height $h - 1$ in $T'$ (because $T'$ is missing the bottom level of $T$). Let $N'_{h-1}$ denote the number of nodes at height $h - 1$ in $T'$.
$N_h = N'_{h-1} = \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \leq \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$.

# In Class Exercise 1: continued

**Problem**

Naively, we can build a heap out of an arbitrary array using successive calls to HEAPIFY-DOWN, starting at element $\lfloor \text{length}[H]/2 \rfloor$ and going down to 1. If each call to HEAPIFY-DOWN takes $O(\log n)$ time and we have $O(n/2)$ such calls, we can build a heap in $O(n \log n)$ time. Prove that this process is actually faster than $O(n \log n)$ (i.e., provide a *tighter* bound on the process's running time). Starters:

- What is the height of an $n$-element heap? $O(\log n)$
- How many nodes are there at height $h$ of an $n$-element heap? $\lceil n/2^{h+1} \rceil$

# In Class Exercise 1: Solution

## Problem

Naively, we can build a heap out of an arbitrary array using successive calls to HEAPIFY-DOWN, starting at element $\lfloor \text{length}[H]/2 \rfloor$ and going down to 1. If each call to HEAPIFY-DOWN takes $O(\log n)$ time and we have $O(n/2)$ such calls, we can build a heap in $O(n \log n)$ time. Prove that this process is actually faster than $O(n \log n)$ (i.e., provide a *tighter* bound on the process's running time).

## Solution

The time required by HEAPIFY-DOWN, when called on a node at height $h$ is $O(h)$. The total cost of building a heap is bounded above by:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n)$$

The last step is because (looking up the summation):

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

# HeapSort

## Sorting with a Priority Queue

### Sort

**Instance:** Nonempty list $x_1, x_2, \ldots, x_n$ of integers

**Solution:** A permutation $y_1, y_2, \ldots y_n$ of $x_1, x_2, \ldots, x_n$ such that $y_i \leq y_{i+1}$ for all $1 \leq i < n$

### Final Algorithm

- Insert each number in a priority queue $H$
- Repeatedly find the smallest number in $H$, output it, and delete it from $H$

Each insertion and deletion takes $O(\log n)$ time for a total running time of $O(n \log n)$

# In Class Exercise 2

**Problem**

One of your classmates claims that he built an alternative data structure (other than a heap) for representing a priority queue. He claims that, using his new data structure, INSERT, MAX, and EXTRACTMAX all take constant ($O(1)$) time in the worst case. Give a very simple proof that he is mistaken.

**Solution**

If this were true, we could comparison sort in $O(n)$ time. But we've already proven that this is not possible.

# Questions