CSE101: Discussion #04
Agenda

1. Diversity, Equity, and Inclusion: Today’s Guest
2. Review:
   a. Priority Queues
   b. Heaps
3. Discussion Problem
Today’s Guest: Héctor García-Molina

DR. HÉCTOR GARCÍA MOLINA
—
1954 – 2019

- Born in Monterrey, Nuevo León, Mexico in 1954
- Studied E.E. at Monterrey Institute of Technology and Higher Studies
- Chairman of Stanford Computer Science Department
DR. HÉCTOR GARCÍA MOLINA

—

1954 – 2019
Today’s Guest: Héctor García-Molina

Dr. Héctor García Molina
1954 – 2019
Priority Queues: What are they?

**Priority Queue** is a data structure which stores a collection of items (priority, info)

1. **Priority**: a.k.a. “Keys” are arbitrary objects on which an order is defined i.e. a total order relation exists (ex. A, B, C or Red, Blue, Green, or 1, 2, 3)
2. **Non-unique priorities/keys**: Multiple items can have same priority/key
Priority Queues: Types?

Priority Queues can be implemented in many ways (including):

1. **Array**: sorted or unsorted
2. **Linked List**: ordered or unordered
3. **Binary Search Tree**
4. **Binary Heap**: (to be discussed later)
Priority Queues: What operations?

Priority Queue supports:

1. **Add**: add item (priority, info) to queue with specified priority/key and info
2. **Peek Max/Min**: find item in queue with min/max priority
3. **Remove Max/Min**: remove item in queue with min/max priority
Priority Queues: Operation run-times?

Priority Queue supports:

1. **Add**: add item (priority, info) to queue with specified priority/key and info
2. **Peek Max/Min**: find item in queue with min/max priority
3. **Remove Max/Min**: remove item in queue with min/max priority

<table>
<thead>
<tr>
<th></th>
<th>Unordered Array</th>
<th>Sorted Array</th>
<th>Unordered Linked List</th>
<th>Sorted Linked List</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(\log N)^*$</td>
</tr>
<tr>
<td>peekMax</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)^*$</td>
</tr>
<tr>
<td>removeMax</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)^*$</td>
</tr>
</tbody>
</table>

* The BST efficiencies assume a balanced tree, which is not guaranteed. Actual efficiencies may be worse than stated.
### Priority Queues: Operation run-times?

**Priority Queue** supports:

1. **Add**: add item (priority, info) to queue with specified priority/key and info
2. **Peek Max/Min**: find item in queue with min/max priority
3. **Remove Max/Min**: remove item in queue with min/max priority

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unordered Array</th>
<th>Sorted Array</th>
<th>Unordered Linked List</th>
<th>Sorted Linked List</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(\log N)^*$</td>
</tr>
<tr>
<td><strong>peekMax</strong></td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)^*$</td>
</tr>
<tr>
<td><strong>removeMax</strong></td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)^*$</td>
</tr>
</tbody>
</table>

*The BST efficiencies assume a balanced tree, which is not guaranteed. Actual efficiencies may be worse than stated.*

**Can we do better?**
Heaps: What are they?

**Binary Heap** is a Binary Tree (but not a Binary Search Tree) that satisfies:

1. **Complete**: A “complete” binary tree is a tree where every level except the last one is completely filled, and the last level has all leaves as far left as possible.
Heaps: What are they?

Binary Heap is a Binary Tree (but not a Binary Search Tree) that satisfies:

1. **Complete**: A “complete” binary tree is a tree where every level except the last one is completely filled, and the last level has all leaves as far left as possible.
Heaps: What are they?

Binary Heap is a Binary Tree (but not a Binary Search Tree) that satisfies:

1. **Complete**: A “complete” binary tree is a tree where every level except the last one is completely filled, and the last level has all leaves as far left as possible.
   a. **Depth of a node**: # of edges from the root to the node.
   b. **Height of a node**: # of edges from the node to the deepest leaf
   c. **Height of a tree**: = height of the root
      (see above: # of edges from the node to the deepest leaf)
Heaps: What are they?

**Binary Heap** is a Binary Tree (but not a Binary Search Tree) that satisfies:

1. **Complete**: A “complete” binary tree is a tree where every level except the last one is completely filled, and the last level has all leaves as far left as possible.
2. **Order**: For every internal node \( n \) i.e. not the root:
   a. **Max Heap**: \( \text{key}(\text{parent}(n)) \geq \text{key}(n) \) i.e. value at any node is **at most** the value of its parent
   b. **Min Heap**: \( \text{key}(\text{parent}(n)) \leq \text{key}(n) \) i.e. value at any node is **at least** the value of its parent
Heaps: What are they?

Binary Heap is a Binary Tree (but not a Binary Search Tree) that satisfies:

1. **Complete**: A “complete” binary tree is a tree where every level except the last one is completely filled, and the last level has all leaves as far left as possible.
2. **Order**: For every internal node (n) i.e. not the root:
   a. **Max Heap**: $\text{key(parent(n))} \geq \text{key(n)}$ i.e. value at any node is **at most** the value of its parent
   b. **Min Heap**: $\text{key(parent(n))} \leq \text{key(n)}$ i.e. value at any node is **at least** the value of its parent
Heaps: What operations?

**Binary Heap** supports:

1. **Add**: add a node to the heap
2. **“Heapify”**: after changing heap, ensure satisfies heap properties (see above: completeness and order)
3. **Peek Max/Min**: find largest or smallest node in heap
4. **Remove Max/Min**: remove largest or smallest node in heap
Heaps: Operation run-times?

Binary Heap supports:

1. **Add**: add a node to the heap $O(1)$
2. **“Heapify”**: after changing heap, ensure satisfies heap properties (see above: completeness and order) $O(\log N)$
3. **Peek Max/Min**: find largest or smallest node in heap $O(1)$
4. **Remove Max/Min**: remove largest or smallest node in heap $O(\log N)$

<table>
<thead>
<tr>
<th></th>
<th>Unordered Array</th>
<th>Sorted Array</th>
<th>Unordered Linked List</th>
<th>Sorted Linked List</th>
<th>Binary Search Tree</th>
<th>Binary Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(\log N)^*$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td><strong>peekMax</strong></td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)^*$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>removeMax</strong></td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)^*$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

* The BST efficiencies assume a balanced tree, which is not guaranteed. Actual efficiencies may be worse than stated.
Discussion Problem
CSE 101 Discussion 4

Part 2
Problem statement

Given: a max-heap containing $n$ elements, and a positive integer $k$.
Want: the $k$th largest element of the heap.

Heap operations

- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$

Given: $k = 4$: outputs 4

Naive: call removeMax $k$ times
Heap has $n$ elements, so $O(k \log n)$
inefficient if $n \approx k$ (large heaps)
Problem statement

Given: a max-heap containing $n$ elements, and a positive integer $k$.
Want: the $k$th largest element of the heap.

$k = 4$: outputs 4

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Given: a max-heap containing $n$ elements, and a positive integer $k$. Want: the $k$th largest element of the heap.

Naive: call removeMax $k$ times
heap has $n$ elements, so $O(k \log n)$

$k = 4$: outputs 4
Problem statement

Given: a max-heap containing $n$ elements, and a positive integer $k$.
Want: the $k$th largest element of the heap.

Naive: call removeMax $k$ times
heap has $n$ elements, so $O(k \log n)$
inefficient if $n \gg k$ (large heaps)

$k = 4$: outputs 4

Heap operations
add: $O(\log n)$
peekMax: $O(1)$
removeMax: $O(\log n)$
Leave input heap unchanged?

We know that the value of any node is at most the value of its parent.

Heap operations

- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Leave input heap unchanged?

We know that the value of any node is at most the value of its parent.

Idea: Keep an updating list of candidates for \( k \)th largest element by looking at children.

Heap operations
- add: \( O(\log n) \)
- peekMax: \( O(1) \)
- removeMax: \( O(\log n) \)
Leave input heap unchanged?

We know that the value of any node is at most the value of its parent.

Idea: Keep an updating list of candidates for $k$th largest element by looking at children.
Leave input heap unchanged?

We know that the value of any node is at most the value of its parent.

Idea: Keep an updating list of *candidates* for *k*th largest element by looking at children.

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Leave input heap unchanged?

We know that the value of any node is at most the value of its parent.

Idea: Keep an updating list of candidates for kth largest element by looking at children.

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Leave input heap unchanged?

We know that the value of any node is at most the value of its parent.

Idea: Keep an updating list of *candidates* for $k$th largest element by looking at children.

We can use a new max-heap to keep track of these candidates (a priority queue also works)
What would this look like?

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$

Input heap, $k = 4$

New heap
Heap operations
add: $O(\log n)$
peekMax: $O(1)$
removeMax: $O(\log n)$

Input heap, $k = 4$

New heap
What would this look like?

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$

Input heap, $k = 4$

New heap
What would this look like?

Input heap, $k = 4$

New heap

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
What would this look like?

Heap operations
add: $O(\log n)$
peekMax: $O(1)$
removeMax: $O(\log n)$

Input heap, $k = 4$

New heap
What would this look like?

Heap operations
add: $O(\log n)$
peekMax: $O(1)$
removeMax: $O(\log n)$

Input heap, $k = 4$

New heap
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.

Heap operations

- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps $(k - 1)$ times:

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$

Runtime analysis
- Maximum size of our new heap: each loop iteration removes 1, adds at most 2
- Heap operations now take $O(\log k)$ so total runtime: $O(k \log k)$
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps $(k - 1)$ times:
  - Remove the maximum element of our new heap. Call it $\text{max}$.
  - Copy over $\text{max}$'s 2 children from the input heap, and add them both to our new heap (we can find these by keeping and incrementing pointers to the input heap).
  - Remove the maximum element of our heap one more time, and return that element.

Runtime analysis
Maximum size of our new heap: each loop iteration removes 1, adds at most 2
Heap operations now take $O(\log k)$ so total runtime: $O(k \log k)$

Heap operations
  - add: $O(\log n)$
  - peekMax: $O(1)$
  - removeMax: $O(\log n)$
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps ($k - 1$) times:
  - Remove the maximum element of our new heap. Call it $\text{max}$.
  - Copy over $\text{max}$’s 2 children from the input heap, and add them both to our new heap (we can find these by keeping and incrementing pointers to the input heap).

Heap operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>peekMax</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeMax</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps ($k - 1$) times:
  - Remove the maximum element of our new heap. Call it $\text{max}$.  
  - Copy over $\text{max}$’s 2 children from the input heap, and add them both to our new heap (we can find these by keeping and incrementing pointers to the input heap)
- Remove the maximum element of our heap one more time, and return that element.

Heap operations

- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps ($k - 1$) times:
  - Remove the maximum element of our new heap. Call it $\text{max}$.
  - Copy over $\text{max}$’s 2 children from the input heap, and add them both to our new heap (we can find these by keeping and incrementing pointers to the input heap)
- Remove the maximum element of our heap one more time, and return that element.

Runtime analysis  Maximum size of our new heap:

Heap operations
- add: $O(\log n)$
- peekMax: $O(1)$
- removeMax: $O(\log n)$
Our algorithm

Given: input heap, an integer $k$
Returns: $k$th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps $(k - 1)$ times:
  - Remove the maximum element of our new heap. Call it $\text{max}$.
  - Copy over $\text{max}$’s 2 children from the input heap, and add them both to our new heap (we can find these by keeping and incrementing pointers to the input heap)
- Remove the maximum element of our heap one more time, and return that element.

**Runtime analysis**  Maximum size of our new heap: each loop iteration removes 1, adds at most 2 $\rightarrow O(k)$
Our algorithm

Given: input heap, an integer \( k \)
Returns: \( k \)th largest element of input heap

- Create a new heap. Copy over the root value of the input heap to the new heap.
- Repeat the following steps \( (k - 1) \) times:
  - Remove the maximum element of our new heap. Call it \( \text{max} \).
  - Copy over \( \text{max} \)'s 2 children from the input heap, and add them both to our new heap (we can find these by keeping and incrementing pointers to the input heap)
- Remove the maximum element of our heap one more time, and return that element.

Runtime analysis

Maximum size of our new heap:
each loop iteration removes 1, adds at most 2 \( \rightarrow O(k) \)
heap operations now take \( O(\log k) \) so total runtime: \( O(k \log k) \)

Heap operations
add: \( O(\log n) \)
peekMax: \( O(1) \)
removeMax: \( O(\log n) \)