Heaps

EECS 214

November 4, 2015
Take-aways

- What is a priority queue is all about?
- How is the *heap property* defined?
- What does a binary heap look like?
- How do its operations work?
- What are their time complexities?
The *priority queue* ADT

\[
\begin{align*}
&\text{Empty}() : \text{PrioQ} \\
&\text{Empty?}(\text{PrioQ}) : \text{Bool} \\
&\text{Insert}(\text{PrioQ}, \text{Element}) \\
&\text{FindMin}(\text{PrioQ}) : \text{Element} \\
&\text{RemoveMin}(\text{PrioQ})
\end{align*}
\]

Note:

An Element has a *key*; keys are totally ordered
## The *priority queue* ADT

<table>
<thead>
<tr>
<th>representation</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Empty</strong>() : PrioQ</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><strong>Empty?</strong>(PrioQ) : Bool</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><strong>Insert</strong>(PrioQ, Element)</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><strong>FindMin</strong>(PrioQ) : Element</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td><strong>RemoveMin</strong>(PrioQ)</td>
<td>$\mathcal{O}(n)$</td>
</tr>
</tbody>
</table>

**Notes:**

1. An Element has a *key*; keys are totally ordered
2. $n$ is the number of elements
The *priority queue* ADT

<table>
<thead>
<tr>
<th>operation</th>
<th>representation</th>
<th>linked list</th>
<th>sorted array</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Empty()</code> : PrioQ</td>
<td></td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><code>Empty?</code> (PrioQ) : Bool</td>
<td></td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><code>Insert</code> (PrioQ, Element)</td>
<td></td>
<td>(O(1))</td>
<td>(O(n))</td>
</tr>
<tr>
<td><code>FindMin</code> (PrioQ) : Element</td>
<td></td>
<td>(O(n))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><code>RemoveMin</code> (PrioQ)</td>
<td></td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>

**Notes:**

1. An Element has a *key*; keys are totally ordered
2. \(n\) is the number of elements
The *priority queue* ADT

<table>
<thead>
<tr>
<th>operation</th>
<th>representation</th>
<th>linked</th>
<th>sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Empty()</code> : PrioQ</td>
<td></td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>Empty?(PrioQ) : Bool</code></td>
<td></td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>Insert(PrioQ, Element)</code></td>
<td></td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
<tr>
<td><code>FindMin(PrioQ) : Element</code></td>
<td></td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td><code>RemoveMin(PrioQ)</code></td>
<td></td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(n)$</td>
</tr>
</tbody>
</table>

Notes:

1. An Element has a *key*; keys are totally ordered
2. $n$ is the number of elements
# The priority queue ADT

<table>
<thead>
<tr>
<th>operation</th>
<th>representation</th>
<th>linked list</th>
<th>sorted ring buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Empty()</code> : PrioQ</td>
<td></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>Empty?</code>(PrioQ) : Bool</td>
<td></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>Insert</code>(PrioQ, Element)</td>
<td></td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>FindMin</code>(PrioQ) : Element</td>
<td></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>RemoveMin</code>(PrioQ)</td>
<td></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Notes:

1. An Element has a key; keys are totally ordered
2. $n$ is the number of elements
We can do better

A heap is a tree that satisfies the heap property
We can do better

A heap is a tree that satisfies the heap property: every element’s key is less than all of its descendants’ keys
We can do better

A *min-heap* is a tree that satisfies the *min-heap property*: every element’s key is less than all of its descendants’ keys.
We can do better

A max-heap is a tree that satisfies the max-heap property: every element’s key is greater than all of its descendants’ keys.
Heaps versus search trees

min-heap property:
for all nodes $n$,
- $n.key < n.left.key$, and
- $n.key < n.right.key$

BST property:
for all nodes $n$,
- for all of $n$’s left-descendants $\ell$,
  $\ell.key < n.key$, and
- for all of $n$’s right-descendants $r$,
  $r.key > n.key$
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.
A tree is *complete* if the levels are all filled in left-to-right
Like this:

```
    O
```
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right. Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right. Like this:

![Diagram of a complete tree](image)
Definition: complete tree

A tree is complete if the levels are all filled in left-to-right. Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.
Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.

Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.

Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.

Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.

Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.
Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right. Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right

Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.  
Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.

Like this:
Definition: *complete tree*

A tree is *complete* if the levels are all filled in left-to-right.

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.

Like this:

```
  3
```

```
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.

Like this:

```
  3
 / \
5   8
```
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:

```
  3
 /\  
5  8
/ \  /
17 6 9
```
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.

Like this:

```
      3
     / \
   5   8
  / \ / \  \
 17 6 9 60 \
 /  \    /  \
20 37  9  60
```
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property.

Like this:
Definition: *binary heap*

*A binary heap* is a complete binary tree satisfying the heap property.

Like this:
Definition: *binary heap*

A *binary heap* is a complete binary tree satisfying the heap property

Like this:
Operation *FindMin*

This one is easy:
Operation *FindMin*

This one is easy:

```
3
5
8
17
6
10
60
20
37
44
14
12
```

How long does this take?

$O(1)$
Operation \textit{FindMin}

This one is easy:

\begin{center}
\begin{tikzpicture}
  \node [circle,draw] (root) {3}
  \child {node [circle,draw] (5) {5}
    \child {node [circle,draw] (17) {17}
      \child {node [circle,draw] (20) {20}}
      \child {node [circle,draw] (37) {37}}
    }
    \child {node [circle,draw] (6) {6}
      \child {node [circle,draw] (44) {44}}
      \child {node [circle,draw] (14) {14}}
    }
  }
  \child {node [circle,draw] (8) {8}
    \child {node [circle,draw] (10) {10}}
    \child {node [circle,draw] (60) {60}}
  }
\end{tikzpicture}
\end{center}

How long does this take? 9:3
Operation \textit{FindMin}

This one is easy:

How long does this take? \(O(1)\)
Operation *Insert*

This one’s a bit harder. Let’s insert 11 into the heap.

```
3
5
17
20

6
37
44
14

8
10
12
60
```

How long does this take?

How tall is the tree?

\[ O(\log n) \]
Operation *Insert*

This one’s a bit harder. Let’s insert 11 into the heap.

Step 1: Add it at the end of the heap

How tall is the tree?

$O(\log n)$
Operation *Insert*

This one’s a bit harder. Let’s insert 11 into the heap.

Step 1: Add it at the end of the heap
Step 2: Check if the heap condition is (locally!) preserved
Operation **Insert**

This one’s a bit harder. Let’s insert 11 into the heap.

Step 1: Add it at the end of the heap
Step 2: Check if the heap condition is (locally!) preserved

It is, so we’re done! Why is the local check sufficient?
Okay, let’s try inserting 9 instead.

The local invariant is broken! How can we fix it?

Swap the troublesome node with its parent.

Now we check 9’s new parent.

How long does this take?

How tall is the tree?

$O(\log n)$
Okay, let’s try inserting 9 instead.
The local invariant is broken! How can we fix it?

How long does this take?
How tall is the tree?
$O(\log n)$
Okay, let’s try inserting 9 instead.
The local invariant is broken! How can we fix it?
Swap the troublesome node with its parent.

Operation *Insert*
Operation *Insert*

Okay, let’s try inserting 9 instead.
The local invariant is broken! How can we fix it? Swap the troublesome node with its parent.
Now we check 9’s new parent.
Operation *Insert*

Okay, let’s try inserting 9 instead.
The local invariant is broken! How can we fix it?
Swap the troublesome node with its parent.
Now we check 9’s new parent. Looks good.
Operation *Insert*

Okay, now let’s insert 2.
Operation *Insert*

Okay, now let’s insert 2.
Check the local invariant.
Operation *Insert*

Okay, now let’s insert 2.
Check the local invariant. It’s broken!
Operation *Insert*

Okay, now let’s insert 2.
Check the local invariant. It’s broken!
So swap with the parent. Still broken!
Operation *Insert*

Okay, now let’s insert 2.
Check the local invariant. It’s broken!
So swap with the parent. Still broken!
So “bubble up” until the invariant is restored.
Operation *Insert*

Okay, now let’s insert 2.

Check the local invariant. It’s broken!
So swap with the parent. Still broken!
So “bubble up” until the invariant is restored.

![Binary Search Tree Diagram]

How long does this take?

How tall is the tree?

$O(\log n)$
When bubbling up, why didn’t we compare the node against its other child? (*E.g.*: comparing 2 to 5)

Operation *Insert*

How long does this take?

How tall is the tree? $O(\log n)$
Operation Insert

When bubbling up, why didn’t we compare the node against its other child? (E.g.: comparing 2 to 5)

Before swap, node < parent, but also parent < other child (by heap condition). Transitivity of < tells us that node < other child!

How long does this take?

10:17
Operation *Insert*

When bubbling up, why didn’t we compare the node against its other child? (*E.g.*: comparing 2 to 5)

Before swap, node < parent, but also parent < other child (by heap condition). Transitivity of < tells us that node < other child!

![Binary Heap Diagram]

How long does this take? How tall is the tree?

10:18
Operation *Insert*

When bubbling up, why didn’t we compare the node against its other child? (*E.g.*: comparing 2 to 5)

Before swap, node < parent, but also parent < other child (by heap condition). Transitivity of < tells us that node < other child!

How long does this take? How tall is the tree? $O(\log n)$
Operation *RemoveMin*

Step 1: Replace the root with the last node, and remove the last node. (This preserves tree completeness.)

Step 2: Restore the invariant by "percolating down": swap new node with its smaller child until invariant is restored.
Operation \textit{RemoveMin}

Step 1: Replace the root with the \textit{last} node, and remove the last node.
Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)
Operation \textit{RemoveMin}

Step 1: Replace the root with the \textit{last} node, and remove the last node. (This preserves tree completeness.)

Step 2: Restore the invariant by “percolating down”: swap new node with its \textit{smaller} child until invariant is restored.
Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)

Step 2: Restore the invariant by “percolating down”: swap new node with its *smaller* child until invariant is restored.
Operation *RemoveMin*

Step 1: Replace the root with the *last* node, and remove the last node. (This preserves tree completeness.)

Step 2: Restore the invariant by “percolating down”: swap new node with its *smaller* child until invariant is restored.
Operation \textit{RemoveMin}

Step 1: Replace the root with the \textit{last} node, and remove the last node. (This preserves tree completeness.)
Step 2: Restore the invariant by “percolating down”: swap new node with its \textit{smaller} child until invariant is restored.
Operation *RemoveMin*

Why do we swap with the smaller child?
Operation *RemoveMin*

Why do we swap with the smaller child? Transitivity again!
Operation \textit{RemoveMin}

Why do we swap with the smaller child? Transitivity again!

How long does this take?
Operation \textit{RemoveMin}

Why do we swap with the smaller child? Transitivity again!

How long does this take? How tall is the tree?
Operation *RemoveMin*

Why do we swap with the smaller child? Transitivity again!

How long does this take? How tall is the tree? \( \mathcal{O}(\log n) \)
Take-aways

- What is a priority queue is all about?
- How is the heap property defined?
- What does a binary heap look like?
- How do its operations work?
- What are their time complexities?