CSE 373: Binary heaps

Michael Lee
Monday, Feb 5, 2018
Course overview

The course so far...

- Reviewing manipulating arrays and nodes
- Algorithm analysis
- Dictionaries (tree-based and hash-based)

Coming up next:

- Divide-and-conquer, sorting
- Graphs
- Misc topics (P vs NP, more?)
Course overview

The course so far...

- Reviewing manipulating arrays and nodes
- Algorithm analysis
- Dictionaries (tree-based and hash-based)

Coming up next:

- Divide-and-conquer, sorting
- Graphs
- Misc topics (P vs NP, more?)
When are we getting project grades/our midterm back?
When are we getting project grades/our midterm back?

Tuesday or Wednesday
Do we have something due soon?

Project 3 will be released today or tomorrow

- Part 1 due in two weeks (Fri, Feb 16)
- Full project due in three weeks (Fri, Feb 23)

Partner selection
- Selection form due Fri, Feb 9

You MUST find a new partner...
...unless both partners email me and petition to stay together
Do we have something due soon?

- Project 3 will be released today or tomorrow
Do we have something due soon?

- Project 3 will be released today or tomorrow
- Due dates:
  - Part 1 due in two weeks (Fri, Feb 16)
  - Full project due in three weeks (Fri, Feb 23)
Do we have something due soon?

- Project 3 will be released today or tomorrow
- Due dates:
  - Part 1 due in two weeks (Fri, Feb 16)
  - Full project due in three weeks (Fri, Feb 23)
- Partner selection
  - Selection form due Fri, Feb 9
Do we have something due soon?

- Project 3 will be released today or tomorrow
- Due dates:
  - Part 1 due in two weeks (Fri, Feb 16)
  - Full project due in three weeks (Fri, Feb 23)
- Partner selection
  - Selection form due Fri, Feb 9
  - You **MUST** find a new partner...
Timeline

Do we have something due soon?

- Project 3 will be released today or tomorrow
- Due dates:
  - Part 1 due in two weeks (Fri, Feb 16)
  - Full project due in three weeks (Fri, Feb 23)
- Partner selection
  - Selection form due Fri, Feb 9
  - You **MUST** find a new partner...
  - ...unless both partners email me and petition to stay together
Motivating question:

Suppose we have a collection of “items”.
We want to return whatever item has the smallest “priority”.
Specifically, want to implement the **Priority Queue** ADT:

<table>
<thead>
<tr>
<th>The Priority Queue ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A priority queue stores elements according to their “priority”. It supports the following operations:</td>
</tr>
</tbody>
</table>

RemoveMin: return the element with the smallest priority.

PeekMin: find (but do not return) the smallest element.

Insert: add a new element to the priority queue.
Specifically, want to implement the **Priority Queue** ADT:

<table>
<thead>
<tr>
<th>The Priority Queue ADT</th>
</tr>
</thead>
<tbody>
<tr>
<td>A priority queue stores elements according to their “priority”. It supports the following operations:</td>
</tr>
<tr>
<td>▶ <strong>removeMin</strong>: return the element with the <em>smallest</em> priority</td>
</tr>
<tr>
<td>▶ <strong>peekMin</strong>: find (but do not return) the <em>smallest</em> element</td>
</tr>
<tr>
<td>▶ <strong>insert</strong>: add a new element to the priority queue</td>
</tr>
</tbody>
</table>
An alternative definition: instead of yielding the element with the largest priority, yield the one with the *largest* priority:

<table>
<thead>
<tr>
<th>The Priority Queue ADT, alternative definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A priority queue stores elements according to their “priority”. It supports the following operations:</td>
</tr>
</tbody>
</table>
An alternative definition: instead of yielding the element with the largest priority, yield the one with the *largest* priority:

The Priority Queue ADT, alternative definition

A priority queue stores elements according to their “priority”. It supports the following operations:

- **removeMax:** return the element with the *largest* priority
- **peekMax:** find (but do not return) the *largest* element
- **insert:** add a new element to the priority queue

The way we implement both is almost identical – we just tweak how we compare elements

In this class, we will focus on implementing a “min” priority queue
Fill in this table with the worst-case runtimes:

<table>
<thead>
<tr>
<th>Idea</th>
<th>removeMin</th>
<th>peekMin</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted array list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td>AVL tree</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
</tr>
</tbody>
</table>
Fill in this table with the worst-case runtimes:

<table>
<thead>
<tr>
<th>Idea</th>
<th>removeMin</th>
<th>peekMin</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unssorted array list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Unssorted linked list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted array list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary tree</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>AVL tree</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
</tr>
</tbody>
</table>
Fill in this table with the worst-case runtimes:

<table>
<thead>
<tr>
<th>Idea</th>
<th>removeMin</th>
<th>peekMin</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted array list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL tree</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Initial implementation ideas

Fill in this table with the worst-case runtimes:

<table>
<thead>
<tr>
<th>Idea</th>
<th>removeMin</th>
<th>peekMin</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array list</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
</tr>
<tr>
<td>Sorted array list</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
<td>Θ (log(n))</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
<td>Θ (log(n))</td>
</tr>
<tr>
<td>Binary tree</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
<td>Θ (log(n))</td>
</tr>
<tr>
<td>AVL tree</td>
<td>Θ (n)</td>
<td>Θ (n)</td>
<td>Θ (log(n))</td>
</tr>
</tbody>
</table>
Initial implementation ideas

Fill in this table with the worst-case runtimes:

<table>
<thead>
<tr>
<th>Idea</th>
<th>removeMin</th>
<th>peekMin</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array list</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Sorted array list</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Binary tree</td>
<td>Θ(log(n))</td>
<td>Θ(log(n))</td>
<td>Θ(log(n))</td>
</tr>
<tr>
<td>AVL tree</td>
<td>Θ(log(n))</td>
<td>Θ(log(n))</td>
<td>Θ(log(n))</td>
</tr>
</tbody>
</table>
We want something optimized both frequent inserts and removes. An AVL tree (or some tree-ish thing) seems good enough... right?
Initial implementation ideas

We want something optimized both frequent inserts and removes. An AVL tree (or some tree-ish thing) seems good enough... right?

Today: learn how to implement a binary heap.

peekMin is $O(1)$, and insert and remove are still $O(\log(n))$ in the worst case.

However, insert is $O(1)$ in the average case!
Idea: adapt the tree-based method
**Idea:** adapt the tree-based method

**Insight:** in a tree, finding the min is expensive! Rather then having it to the left, have it on the top!
**Idea:** adapt the tree-based method

**Insight:** in a tree, finding the min is expensive! Rather then having it to the left, have it on the top!

A BST or AVL tree

A binary heap
We now need to change our invariants...

**Binary heap invariants**

A binary heap has three invariants:

- **Num children**: Every node has at most 2 children
We now need to change our invariants...

**Binary heap invariants**

A binary heap has three invariants:

- **Num children:** Every node has at most 2 children
- **Heap:** Every node is smaller than its children
We now need to change our invariants...

**Binary heap invariants**

A binary heap has three invariants:

- **Num children:** Every node has at most 2 children
- **Heap:** Every node is smaller than its children
- **Structure:** Every heap is a “complete” tree – it has no “gaps”
Example of a heap

A broken heap

```
2
/  \
4   7
/ \
5  10 9
/ \
6 11
```
Example of a heap

A fixed heap
The heap invariant

Are these all heaps?

1. 
   - 4
   - 2
   - 3
   - 5
   - 7
   - 6
   - 9
   - 10

2. 
   - 2
   - 3
   - 7
   - 6
   - 9
   - 8
   - 10

3. 
   - 5
   - 6
   - 7
   - 8
   - 9

4. 
   - 4
   - 5
   - 6
   - 7
   - 8
   - 9
   - 10
Implementing peekMin

How do we implement peekMin?

Easy: just return the root. Runtime: $\Theta(1)$.
Implementing peekMin

How do we implement peekMin?

Easy: just return the root. Runtime: $\Theta(1)$. 
Implementing removeMin

What about removeMin?

Problem: Structure invariant is broken – the tree has a gap!
Implementing removeMin

What about removeMin?

Step 1: Just remove it!
What about removeMin?

Step 1: Just remove it!

\[ \begin{array}{c}
4 & \quad & 7 \\
5 & \quad & 7 & \quad & 10 & \quad & 9 \\
6 & \quad & 11 \\
\end{array} \]

**Problem:** Structure invariant is broken – the tree has a gap!
Implementing removeMin

How do we fix the gap?

Problem: Heap invariant is broken – 11 is not smaller than 4 or 7!
How do we fix the gap?

Step 2: Plug the gap by moving the last element to the top!
Implementing removeMin

How do we fix the gap?

Step 2: Plug the gap by moving the last element to the top!

Problem: Heap invariant is broken – 11 is not smaller then 4 or 7!
Implementing `removeMin`

How do we fix the heap invariant?
Implementing removeMin

How do we fix the heap invariant?

Step 3: “percolate down” – keep swapping node with smallest child

And we’re done!
Implementing removeMin

How do we fix the heap invariant?

Step 3: “percolate down” – keep swapping node with smallest child
Implementing removeMin

How do we fix the heap invariant?

Step 3: “percolate down” – keep swapping node with smallest child

And we’re done!
Practice: What happens if we call removeMin?
Practice: What happens if we call removeMin?

After removing min:
Analyzing removeMin

The percolateDown algorithm

```java
percolateDown(node) {
    while (node.data is bigger then children) {
        swap data with smaller child
    }
}
```
Analyzing removeMin

The percolateDown algorithm

```java
percolateDown(node) {
    while (node.data is bigger then children) {
        swap data with smaller child
    }
}
```

The runtime?

\[
\text{findLastNodeTime + removeRootTime + numSwaps \times swapTime}
\]

This ends up being:

\[
n + 1 + \log(n) \cdot 1
\]

...which is in \( \Theta(n) \).
Analyzing removeMin

The percolateDown algorithm

```java
percolateDown(node) {
    while (node.data is bigger than children) {
        swap data with smaller child
    }
}
```

The runtime?

```
findLastNodeTime + removeRootTime + numSwaps \times swapTime
```

This ends up being:

```
n + 1 + \log(n) \cdot 1
```

...which is in $\Theta(n)$.  

20
Implementing insert

What about insert? Suppose we insert 3 – what happens?

```
Step 1: insert at last available node
```

```
2 4 5 6
11 7 2 4 5 6
11 7 2 4 5 6 3
11 7 2 4 5 6 3 10 9
```

Problem: heap invariant broken! 7 is not smaller than 3!
Implementing insert

What about insert? Suppose we insert 3 – what happens?

Step 1: insert at last available node

![Heap Diagram]

Problem: heap invariant broken! 7 is not smaller than 3!
Implementing insert

What about insert? Suppose we insert 3 – what happens?

Step 1: insert at last available node

![Diagram of a heap with nodes 2, 4, 7, 5, 7, 10, 9, 6, 11, 3]

**Problem:** heap invariant broken! 7 is not smaller than 3!
Implementing insert

How do we fix the heap invariant?
Implementing insert

How do we fix the heap invariant?

Step 2: “percolate up” – keep swapping node with parent until heap invariant is fixed
Implementing insert

How do we fix the heap invariant?

Step 2: “percolate up” – keep swapping node with parent until heap invariant is fixed

All ok!
Implementing insert

How do we fix the heap invariant?

Step 2: “percolate up” – keep swapping node with parent until heap invariant is fixed

All ok!
Practice: What happens if we insert 3?
Practice: What happens if we insert 3?

After inserting 3:
Analyzing insert

The percolateUp algorithm

```java
percolateUp(node) {
    while (node.data is smaller then parent) {
        swap data with parent
    }
}
```

The runtime?

\[ \text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime} \]

This ends up being:

\[ n + 1 + \log(n) \times 1 \]

...which is in \( \Theta(n) \).
Analyzing insert

The `percolateUp` algorithm

```java
percolateUp(node) {
    while (node.data is smaller then parent) {
        swap data with parent
    }
}
```

The runtime?

\[
\text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}
\]

This ends up being:

\[
n + 1 + \log(n) \cdot 1
\]

...which is in \(\Theta(n)\).
Analyzing insert

The percolateUp algorithm

```java
percolateUp(node) {
    while (node.data is smaller then parent) {
        swap data with parent
    }
}
```

The runtime?

findLastNodeTime + addNodeToLastTime + numSwaps × swapTime

This ends up being:

\[ n + 1 + \log(n) \cdot 1 \]

...which is in \( \Theta(n) \).
**Problem:** But wait! I promised worst-case $\Theta(\log(n))$ insert and average-case $\Theta(1)$ insert.

This algorithm is $\Theta(\log(n))$ in both the worst and average case!
Analyzing removeMin, part 2

**Problem:** But wait! I promised worst-case $\Theta(\log(n))$ insert and average-case $\Theta(1)$ insert.

This algorithm is $\Theta(\log(n))$ in both the worst and average case!

**Why:** Finding and modifying the last node is slow: requires traversal!

Can we speed it up?
Remember this slide?

<table>
<thead>
<tr>
<th>Idea</th>
<th>removeMax</th>
<th>peekMax</th>
<th>insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Unsorted linked list</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted array list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td>AVL tree</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
</tr>
</tbody>
</table>
Observation:

- Arrays let us find and append to the end quickly
- Trees let us have nice $\log(n)$ traversal behavior
Analyzing removeMin, part 2

Observation:

- Arrays let us find and append to the end quickly
- Trees let us have nice $\log(n)$ traversal behavior

The trick: Why pick one or the other? Let’s do both!
The array-based representation of binary heaps

Take a tree:

A
  /   \
B     C
 /\   /\ \
D E   F G
 /\   /\    /\ \
H I J K L
The array-based representation of binary heaps

Take a tree:

```
A
 / \
B   C
 /   /\
D   E  F
|   |   |
H   I   J
```

And fill an array in the **level-order** of the tree:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The array-based representation of binary heaps

Take a tree:

How do we find parent?

\[ \text{parent}(i) = \left\lfloor \frac{i - 1}{2} \right\rfloor \]

The left child?

\[ \text{leftChild}(i) = 2i + 1 \]

The right child?

\[ \text{leftChild}(i) = 2i + 2 \]

And fill an array in the level-order of the tree:

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finding the last node

If our tree is represented using an array, what’s the time needed to find the last node now?
Finding the last node

If our tree is represented using an array, what’s the time needed to find the last node now?

Θ(1): just use `this.array[this.size - 1]`. 
If our tree is represented using an array, what’s the time needed to find the last node now?

$\Theta(1)$: just use `this.array[this.size - 1]`.

...assuming array has no 'gaps'. (Hey, it looks like the structure invariant was useful after all)
Re-analyzing insert

How does this change runtime of insert?
Re-analyzing insert

How does this change runtime of insert?

Runtime of insert:

\[ \text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime} \]

...which is:

\[ 1 + 1 + \text{numSwaps} \times 1 \]
Re-analyzing insert

How does this change runtime of insert?

Runtime of insert:

\[ \text{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime} \]

...which is:

\[ 1 + 1 + \text{numSwaps} \times 1 \]

Observation: when percolating, we usually need to percolate up a few times! So, \( \text{numSwaps} \approx 1 \) in the average case, and \( \text{numSwaps} \approx \text{height} = \log(n) \) in the worst case!
Re-analyzing removeMin

How does this change runtime of removeMin?
Re-analyzing removeMin

How does this change runtime of removeMin?

Runtime of removeMin:

\[\text{findLastNodeTime} + \text{removeRootTime} + \text{numSwaps} \times \text{swapTime}\]

...which is:

\[1 + 1 + \text{numSwaps} \times 1\]
Re-analyzing removeMin

How does this change runtime of removeMin?

Runtime of removeMin:

$$\text{findLastNodeTime} + \text{removeRootTime} + \text{numSwaps} \times \text{swapTime}$$

...which is:

$$1 + 1 + \text{numSwaps} \times 1$$

**Observation:** unfortunately, in practice, usually must percolate all the way down. So $\text{numSwaps} \approx \text{height} \approx \log(n)$ on average.