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)

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?)

Timeline

When are we getting project grades/our midterm back?

Timeline

When are we getting project grades/our midterm back?

Tuesday or Wednesday

Timeline

Do we have something due soon?

Timeline

Do we have something due soon?

- Project 3 will be released today or tomorrow

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)

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

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...

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

Today

Motivating question:

Suppose we have a collection of “items”.

We want to return whatever item has the smallest “priority”.

The Priority Queue ADT

Specifically, want to implement the **Priority Queue** ADT:

The Priority Queue ADT

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

The Priority Queue ADT

Specifically, want to implement the **Priority Queue** ADT:

The Priority Queue ADT

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

- **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

The Priority Queue ADT

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:

The Priority Queue ADT

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

Initial implementation ideas

Fill in this table with the worst-case runtimes:

Idea	**removeMin**	**peekMin**	**insert**
Unsorted array list			
Unsorted linked list			
Sorted array list			
Sorted linked list			
Binary tree			
AVL tree			

Initial implementation ideas

Fill in this table with the worst-case runtimes:

Idea	**removeMin**	**peekMin**	**insert**
Unsorted array list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Unsorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Sorted array list			
Sorted linked list			
Binary tree			
AVL tree			

Initial implementation ideas

Fill in this table with the worst-case runtimes:

Idea	removeMin	peekMin	insert
Unsorted array list			
Unsorted linked list			
Sorted array list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Sorted linked list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Binary tree			
AVL tree			

Initial implementation ideas

Fill in this table with the worst-case runtimes:

Idea	removeMin	peekMin	insert
Unsorted array list			
Unsorted linked list			
Sorted array list			
Sorted linked list			
Binary tree	$\Theta(n)$	$\Theta(n)$	$\Theta(\log(n))$
AVL tree			

Initial implementation ideas

Fill in this table with the worst-case runtimes:

Idea	**removeMin**	**peekMin**	**insert**
Unsorted array list			
Unsorted linked list			
Sorted array list			
Sorted linked list			
Binary tree			
AVL tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

Initial implementation ideas

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 $\mathcal{O}(1)$, and `insert` and `remove` are still $\mathcal{O}(\log(n))$ in the worst case.

However, `insert` is $\mathcal{O}(1)$ in the average case!

Binary heap invariants

Idea: adapt the tree-based method

Binary heap invariants

Idea: adapt the tree-based method

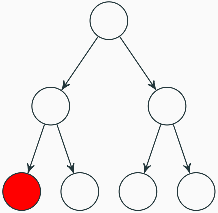
Insight: in a tree, finding the min is expensive! Rather then having it to the left, have it on the top!

Binary heap invariants

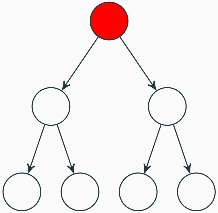
Idea: adapt the tree-based method

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



Binary heap invariants

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

Binary heap invariants

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 then its children

Binary heap invariants

We now need to change our invariants...

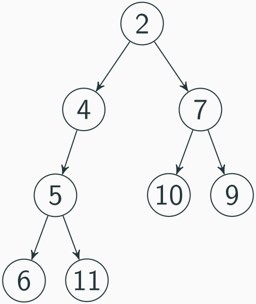
Binary heap invariants

A binary heap has three invariants:

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

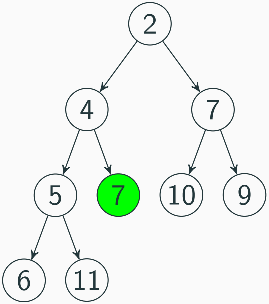
Example of a heap

A broken heap



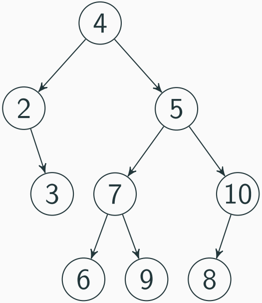
Example of a heap

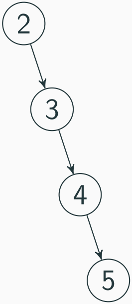
A fixed heap

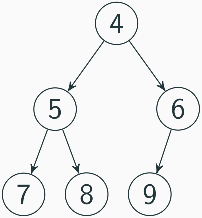


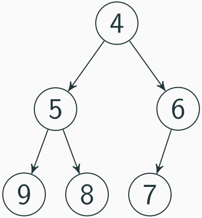
The heap invariant

Are these all heaps?



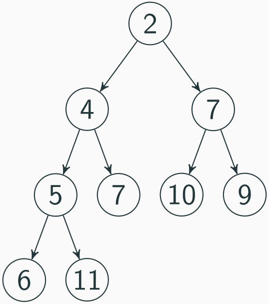






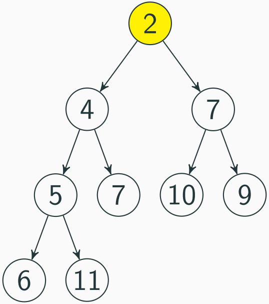
Implementing peekMin

How do we implement `peekMin`?



Implementing peekMin

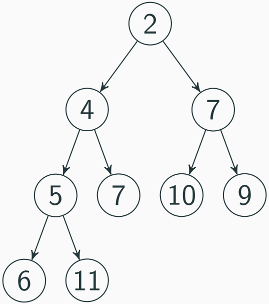
How do we implement `peekMin`?



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

Implementing removeMin

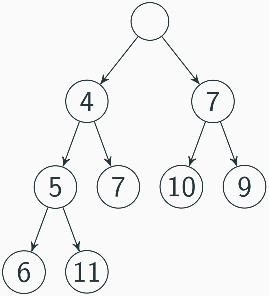
What about `removeMin`?



Implementing removeMin

What about `removeMin`?

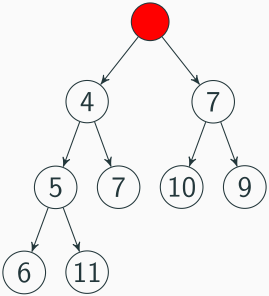
Step 1: Just remove it!



Implementing removeMin

What about `removeMin`?

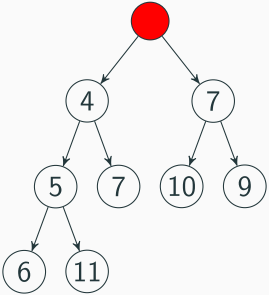
Step 1: Just remove it!



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

Implementing removeMin

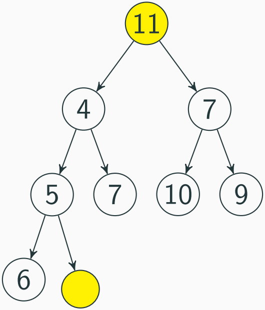
How do we fix the gap?



Implementing removeMin

How do we fix the gap?

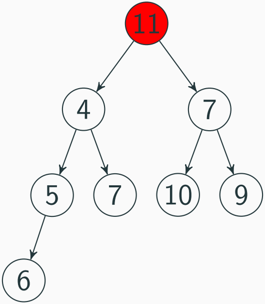
Step 2: Plug the gap by moving the last element to the top!



Implementing removeMin

How do we fix the gap?

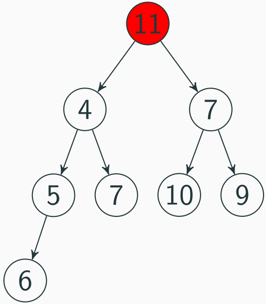
Step 2: Plug the gap by moving the last element to the top!



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

Implementing removeMin

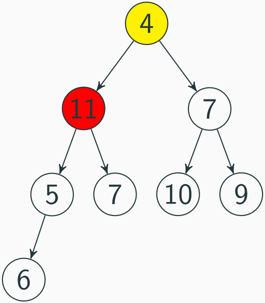
How do we fix the heap invariant?



Implementing removeMin

How do we fix the heap invariant?

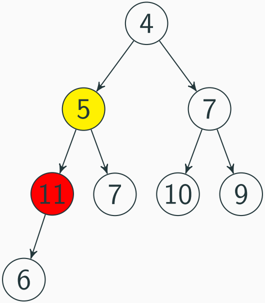
Step 3: "percolate down" – keep swapping node with smallest child



Implementing removeMin

How do we fix the heap invariant?

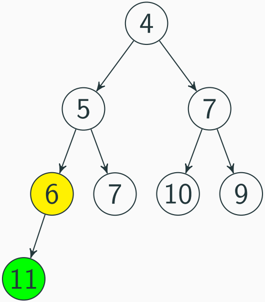
Step 3: “percolate down” – keep swapping node with smallest child



Implementing removeMin

How do we fix the heap invariant?

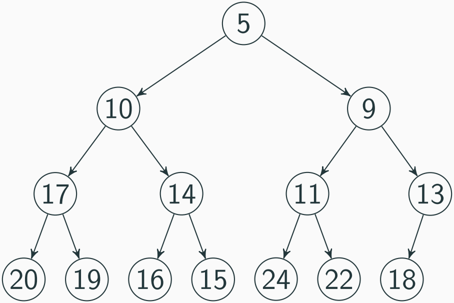
Step 3: “percolate down” – keep swapping node with smallest child



And we’re done!

Practice

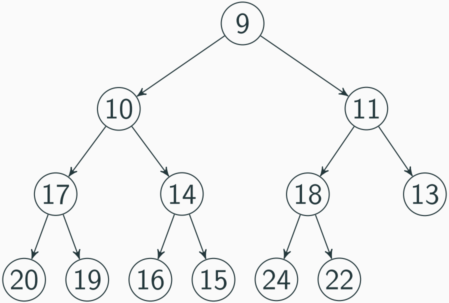
Practice: What happens if we call `removeMin`?



Practice

Practice: What happens if we call `removeMin`?

After removing min:



Analyzing removeMin

The percolateDown algorithm

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

Analyzing removeMin

The percolateDown algorithm

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

The runtime?

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

Analyzing removeMin

The percolateDown algorithm

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

The runtime?

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

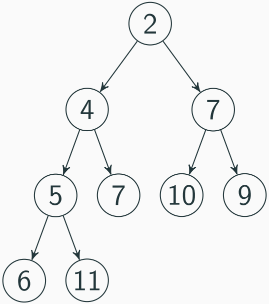
This ends up being:

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

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

Implementing insert

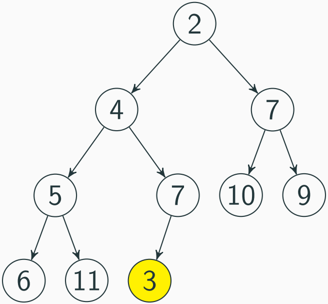
What about `insert`? Suppose we insert `3` – what happens?



Implementing insert

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

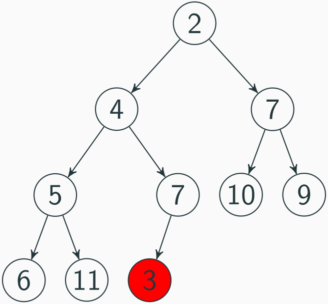
Step 1: insert at last available node



Implementing insert

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

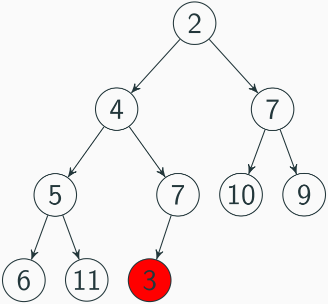
Step 1: insert at last available node



Problem: heap invariant broken! 7 is not smaller then 3!

Implementing insert

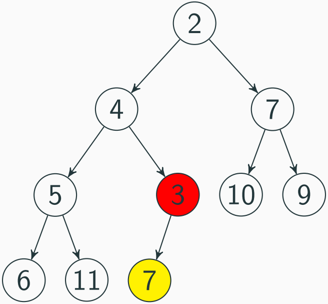
How do we fix the heap invariant?



Implementing insert

How do we fix the heap invariant?

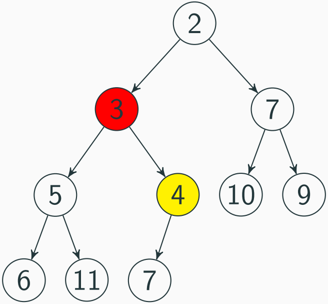
Step 2: “percolate up” – keep swapping node with parent until heap invariant is fixed



Implementing insert

How do we fix the heap invariant?

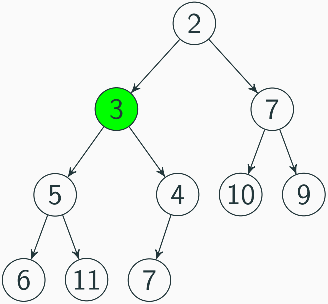
Step 2: "percolate up" – keep swapping node with parent until heap invariant is fixed



Implementing insert

How do we fix the heap invariant?

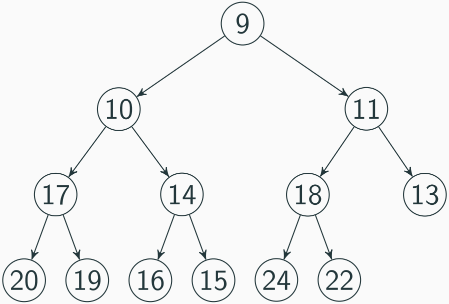
Step 2: "percolate up" – keep swapping node with parent until heap invariant is fixed



All ok!

Practice

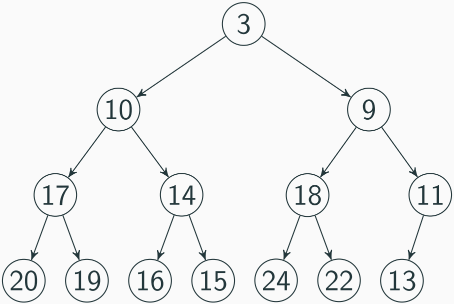
Practice: What happens if we insert 3?



Practice

Practice: What happens if we insert 3?

After inserting 3:



Analyzing insert

The `percolateUp` algorithm

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

Analyzing insert

The percolateUp algorithm

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

The runtime?

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

Analyzing insert

The percolateUp algorithm

```
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 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!

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?

Analyzing removeMin, part 2

Remember this slide?

Idea	**removeMax**	**peekMax**	**insert**
Unsorted array list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Unsorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Sorted array list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Sorted linked list	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Binary tree	$\Theta(n)$	$\Theta(n)$	$\Theta(\log(n))$
AVL tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$

Analyzing removeMin, part 2

Observation:

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

Analyzing removeMin, part 2

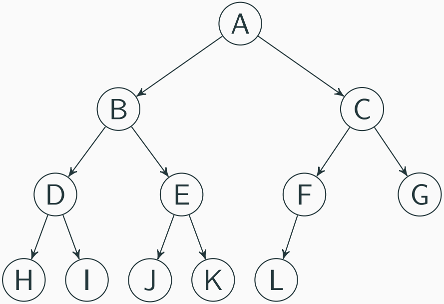
Observation:

- 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!

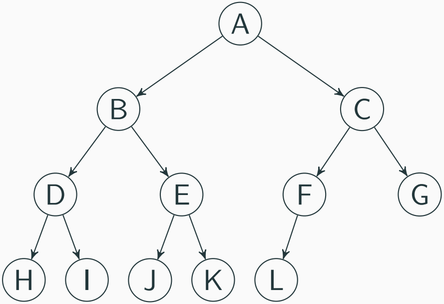
The array-based representation of binary heaps

Take a tree:



The array-based representation of binary heaps

Take a tree:

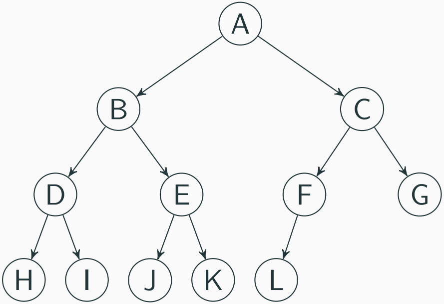


And fill an array in the **level-order** of the tree:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E	F	G	H	I	J	K	L			

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:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	D	E	F	G	H	I	J	K	L			

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?

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

Finding the last node

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:

$$\texttt{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}$$

...which is:

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

Re-analyzing insert

How does this change runtime of `insert`?

Runtime of insert:

$$\texttt{findLastNodeTime} + \text{addNodeToLastTime} + \text{numSwaps} \times \text{swapTime}$$

...which is:

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

Observation: when percolating, we usually need to percolate up a few times! So, numSwaps ≈ 1 in the average case, and numSwaps $\approx$ 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 numSwaps $\approx$ height $\approx \log(n)$ on average.