The Binary Heap

CS 214, Fall 2019

Implementing a priority queue

A (min-)priority queue provides these operations:

- `insert`: adds an element
- `remove_min`: removes the smallest element

Some implementation complexities

	`insert`	`remove_min`
sorted list	$\mathcal{O}(n)$	$\mathcal{O}(1)$
unsorted list	$\mathcal{O}(1)$	$\mathcal{O}(n)$

Some implementation complexities

	insert	remove_min
sorted list	$\mathcal{O}(n)$	$\mathcal{O}(1)$
unsorted list	$\mathcal{O}(1)$	$\mathcal{O}(n)$
binary heap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Introducing the binary heap

A *binary heap* is complete binary tree that is *heap-ordered*

A tree is heap-ordered if every element is *less than or equal* to its children

Introducing the binary heap

A *binary heap* is complete binary tree that is *heap-ordered*

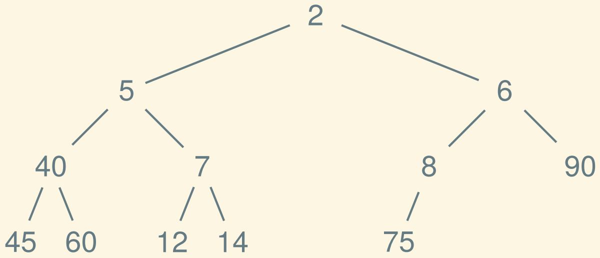
A tree is heap-ordered if every element is *less than or equal* to its children

Which of these is a binary heap?:

```
        2                 2                 5
      /   \             /   \             /   \
     5     97          5     97          2     97
    / \      \        / \    /          / \    /
  40   7      99    40   7  99        40   7  99
```

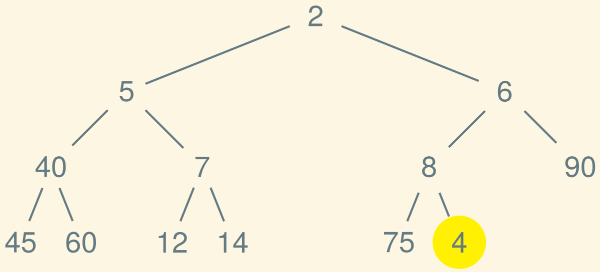
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



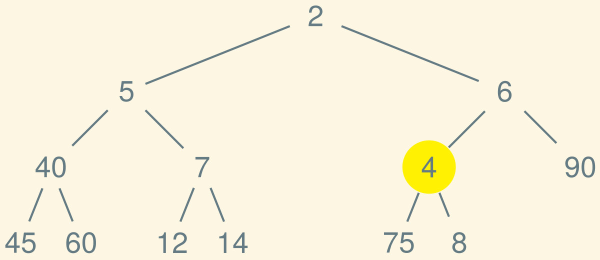
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



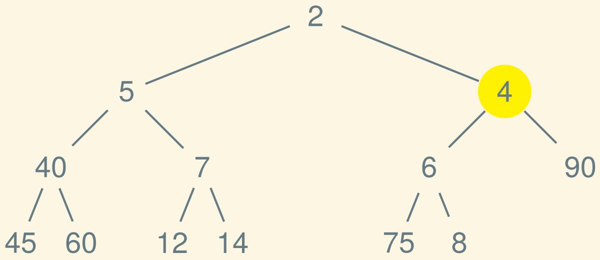
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



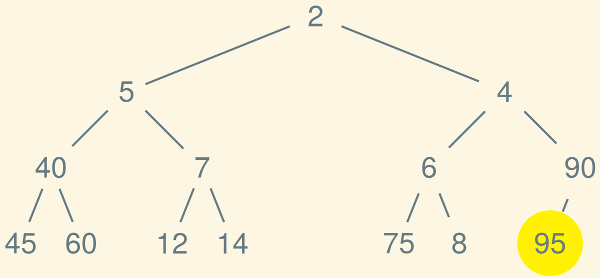
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



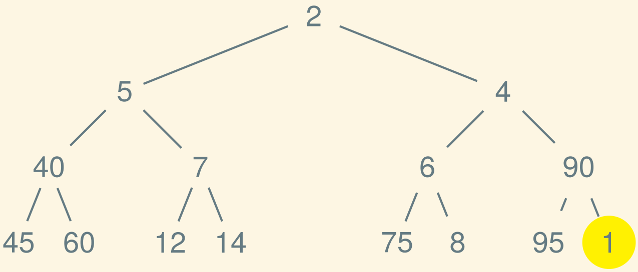
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



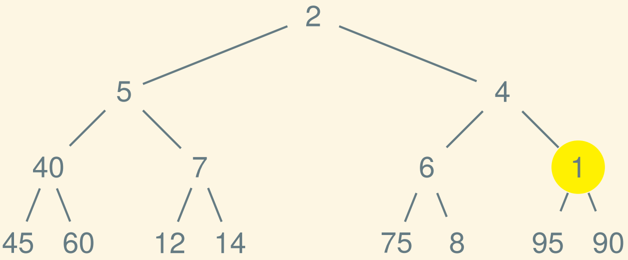
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



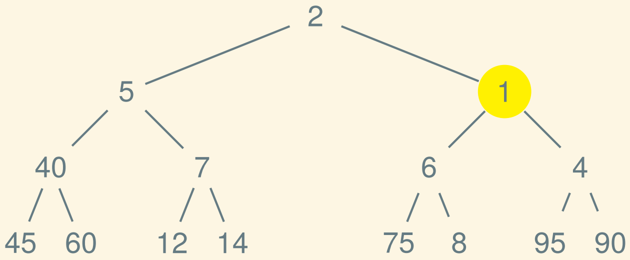
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



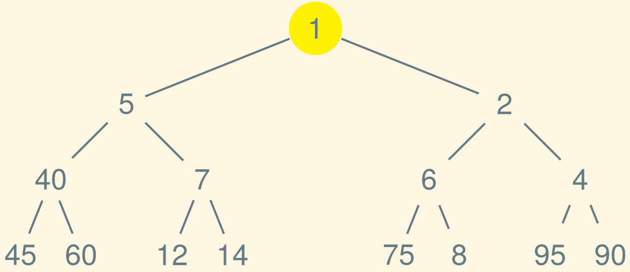
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



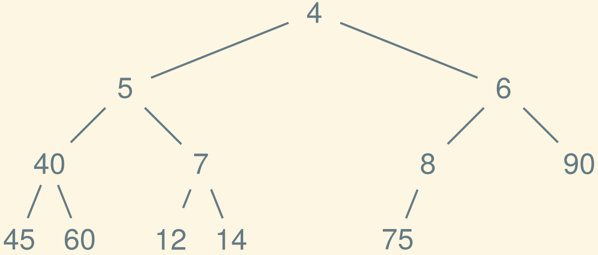
Binary heap insertion

1. Add the new element at the end
2. Bubble up to restore invariant



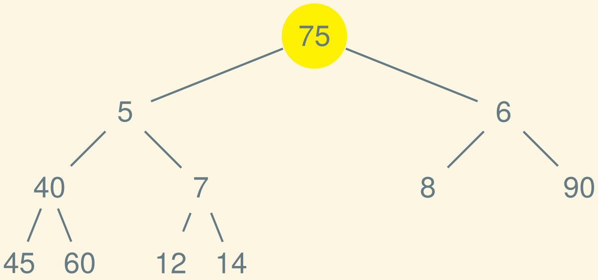
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



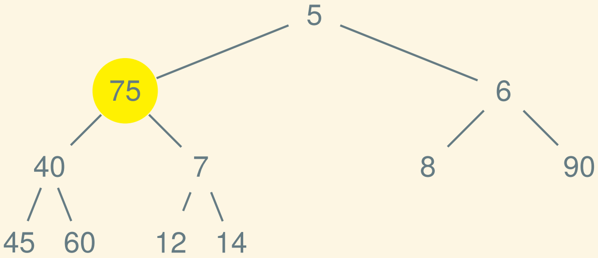
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



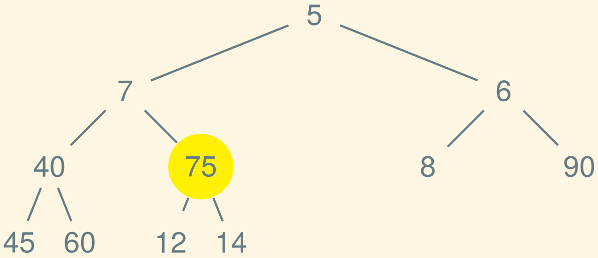
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



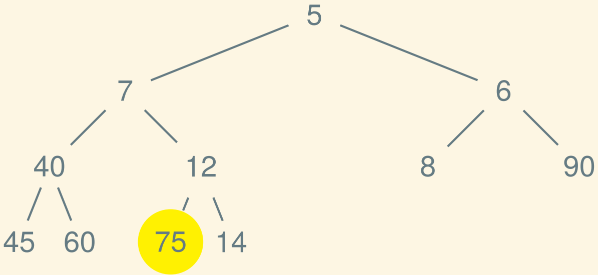
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



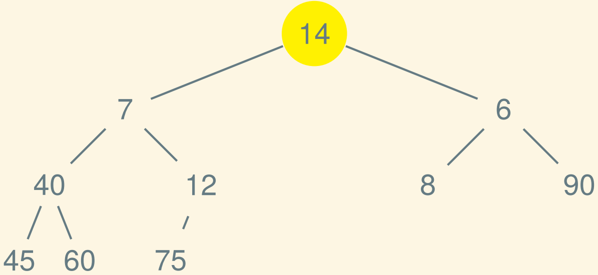
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



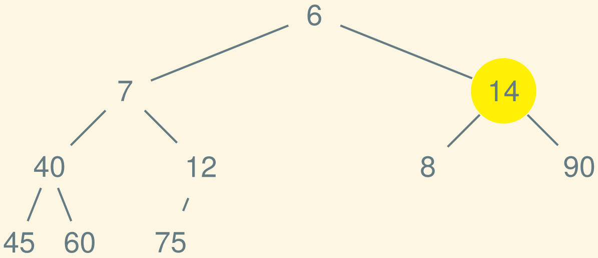
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



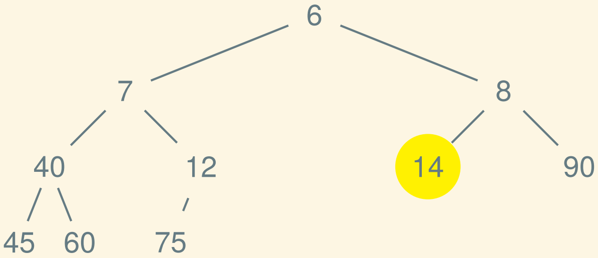
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



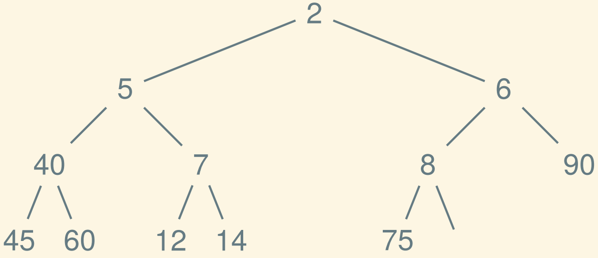
Binary heap removal

1. Replace the root with the last element of the heap
2. Sink down to restore invariant



The super cool thing about binary heaps

Instead of storing it as an actual tree with pointers:

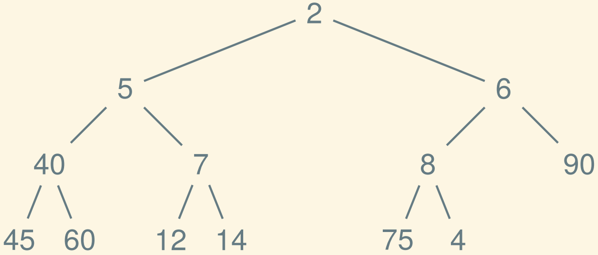


a binary heap is stored in level-order in an array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2	5	6	40	7	8	90	45	60	12	14	75												

The super cool thing about binary heaps

Instead of storing it as an actual tree with pointers:

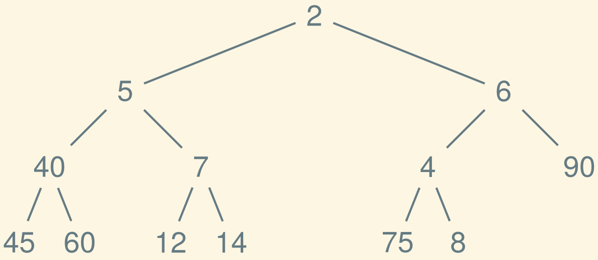


a binary heap is stored in level-order in an array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2	5	6	40	7	8	90	45	60	12	14	75	4											

The super cool thing about binary heaps

Instead of storing it as an actual tree with pointers:

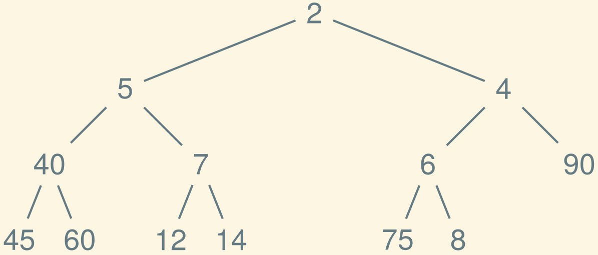


a binary heap is stored in level-order in an array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2	5	6	40	7	4	90	45	60	12	14	75	8											

The super cool thing about binary heaps

Instead of storing it as an actual tree with pointers:



a binary heap is stored in level-order in an array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
2	5	4	40	7	6	90	45	60	12	14	75	8											

Finding parents and children

Because the structure is *implicit*, we can't just follow pointers

Suppose i is the index of a node:

- How can we find its parent (if any)?
- How can we find its children (if any)?

Next time: another graph algorithm and another data structure to go with it