Unit 11: Heaps

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Unit 11 Overview

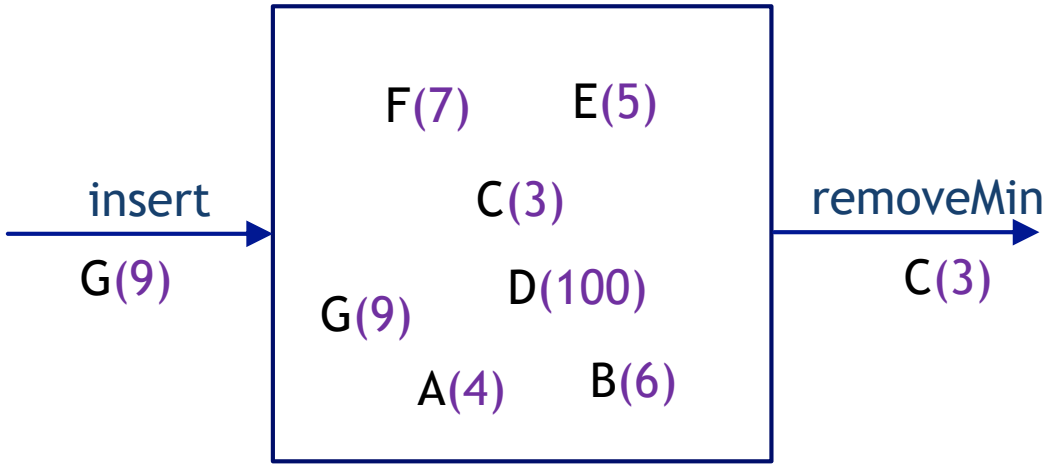
- Learning Objectives: (You should be able to…)
  - provide examples of appropriate applications for priority queues and heaps
  - describe how data is manipulated in heap when an element is inserted or removed
  - describe the benefits of a heap over a list to implement the priority queue ADT

Priority Queues - Motivation

- Suppose I have a made a to-do list (and given the things I need to do priority values, where 1 is most important, 10 is the least)
  - 7 – Buy groceries
  - 2 – Create next lecture's worksheet
  - 8 – Replace bicycle flat tire
  - 3 – Sleep
  - 4 – Write Assignment 5
  - 1 – Create next lecture's slides, videos, and quiz
- As I go through my to-do list, I need to quickly find the task with the highest priority (in this case with the *minimum* key value)

The Notion of a Priority Queue

- Collection of items
  - Characterized by the way it is organized to allow for fast access to and removal of the element with the smallest (or largest) key
  - Element with the highest priority is the first element to be removed
    - Although it is named a priority **queue**, it's not FIFO



The Priority Queue ADT

- The Priority Queue ADT specifies the following operations:
  - insert(*o*, *k*): Insert object *o* with key k into the collection
  - removeMin() or removeMax(): Removes the element with the minimum (or maximum) key from the collection
  - isEmpty(): Determines whether the collection is currently empty

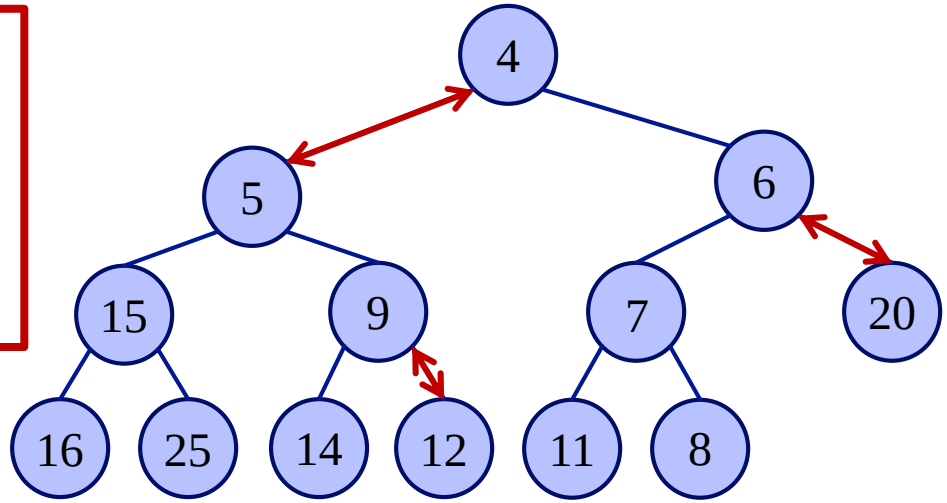
- In general, a single priority queue supports *only* removeMin or *only* removeMax, but not both

We can implement a Priority Queue with an array or a linked list, but either the insert or the removeMin operation would not be efficient

The Heap Data Structure

- A *heap* is a realization of a Priority Queue that is efficient for both insertion and removal of minimum or maximum values
- A *heap* satisfies the following properties:
  - **Heap-Shape Property:** A *heap* is a complete binary tree
  - ***Heap-Order Property***: for every node v other than the root, the priority of the key stored at v is less than or equal to the key stored at v's parent

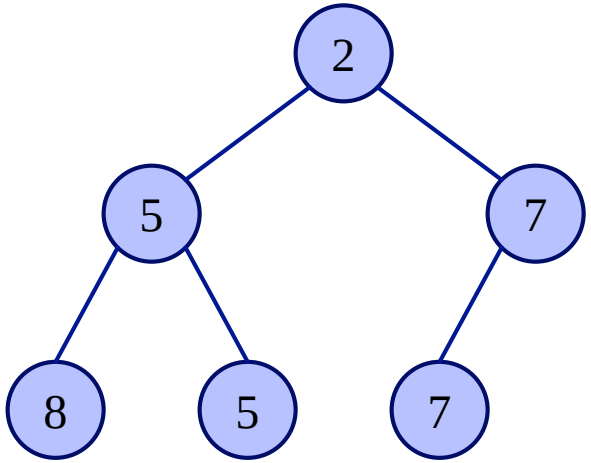
All levels are filled down to the bottom level, and it is filled from left to right

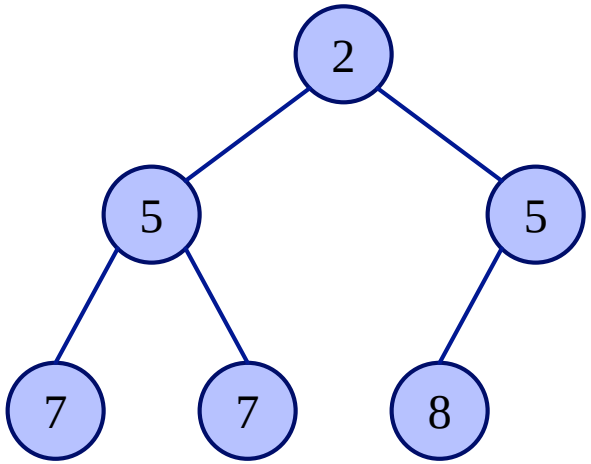


Heaps are *partially* ordered, they maintain a relationship between parent and child, but not across children

Duplicate Keys, Different Ordering

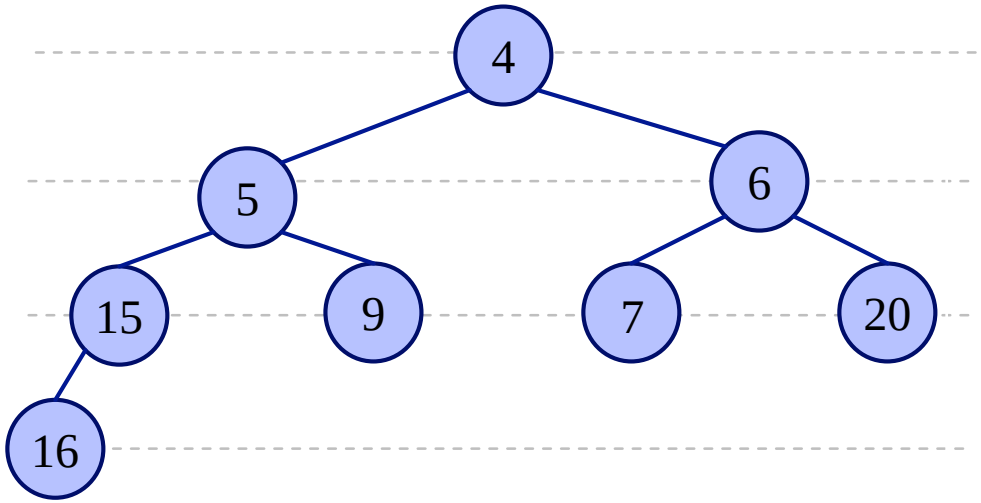
- We will need to consider the *Heap-Order Property* when we implement the insertion and removal methods
- **Observation:** two binary heaps can contain the same data, but the elements may appear at different positions within the structure
- Remember: *Heap-Order Property* looks at parent-child relationships





Height of a Heap

- The *Heap-Shape Property* is important for runtime analysis
- Theorem: A heap storing n keys has height $O(\log n)$.
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at level i for $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \cdots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



level	keys
0	1
1	2
2^{h-2}	2^{h-2}
2^{h-1}	1

Heap Implementation

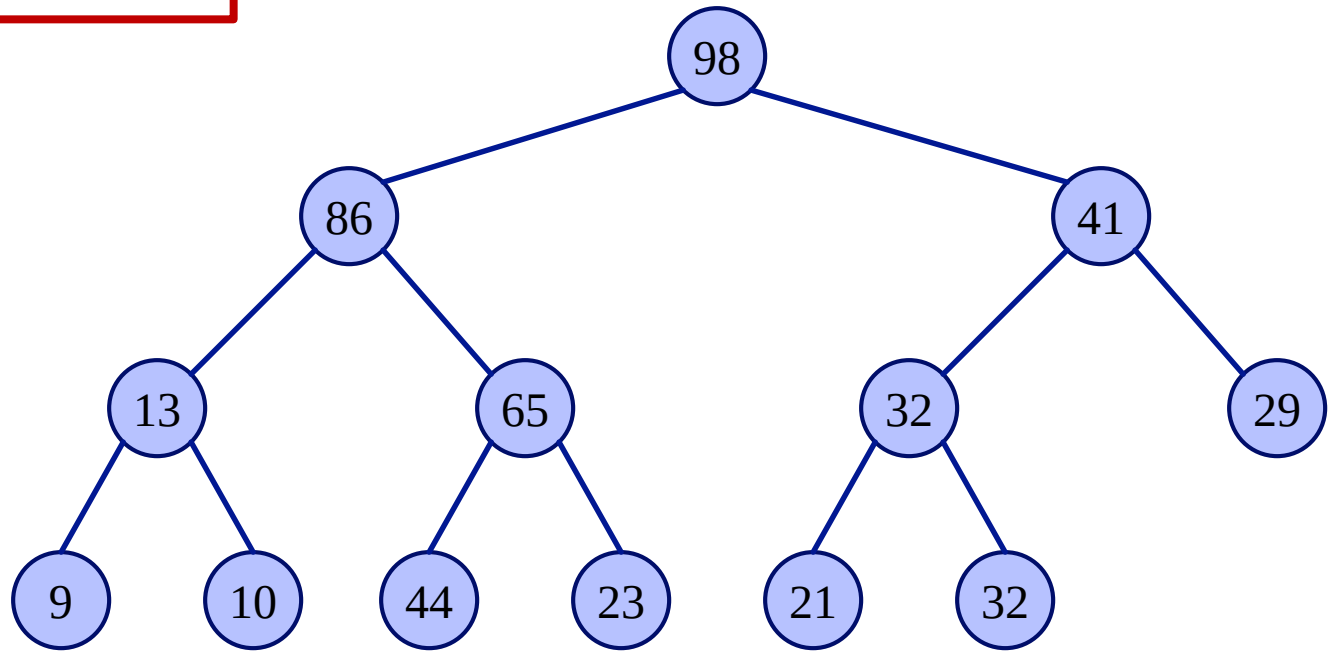
- The next video overviews the implementation of a Heap, where will see why these properties are important, and why we might choose to use a Heap to implement the Priority Queue ADT

Heap Insertion

- Heap properties need to be maintained
- *Heap-Shape Property:*
  - Insert the element at the first available spot (this equates to the left-most index at the bottom level of the tree
  - Therefore the heap remains a *complete binary tree*
- *Heap-Order Property*
  - “Bubble” the element up the tree until the heap-order property is restored
  - Repeatedly compare with parent and swap until the ordering is restored

Heap Insertion Example (Max Heap)

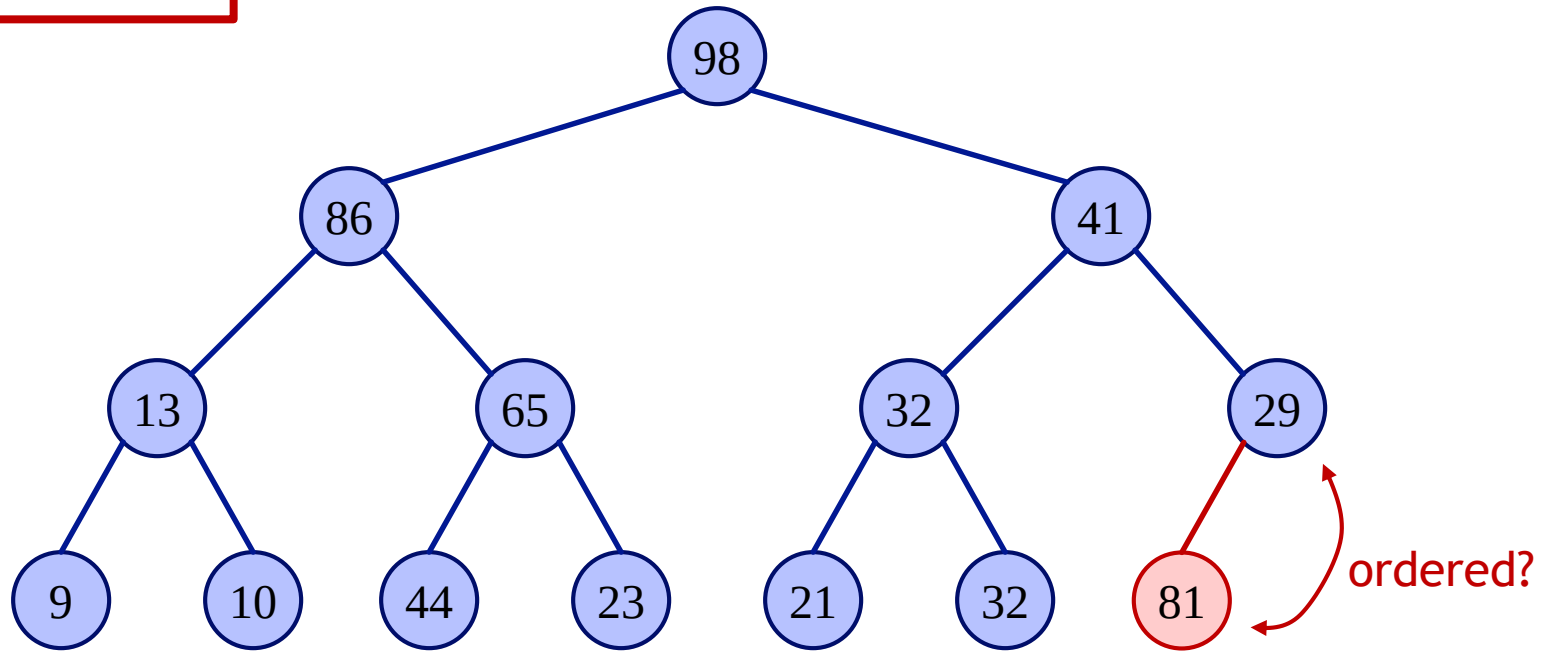
Insert 81



value	--	98	86	41	13	65	32	29	9	10	44	23	21	32	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Insertion Example (Max Heap)

Insert 81

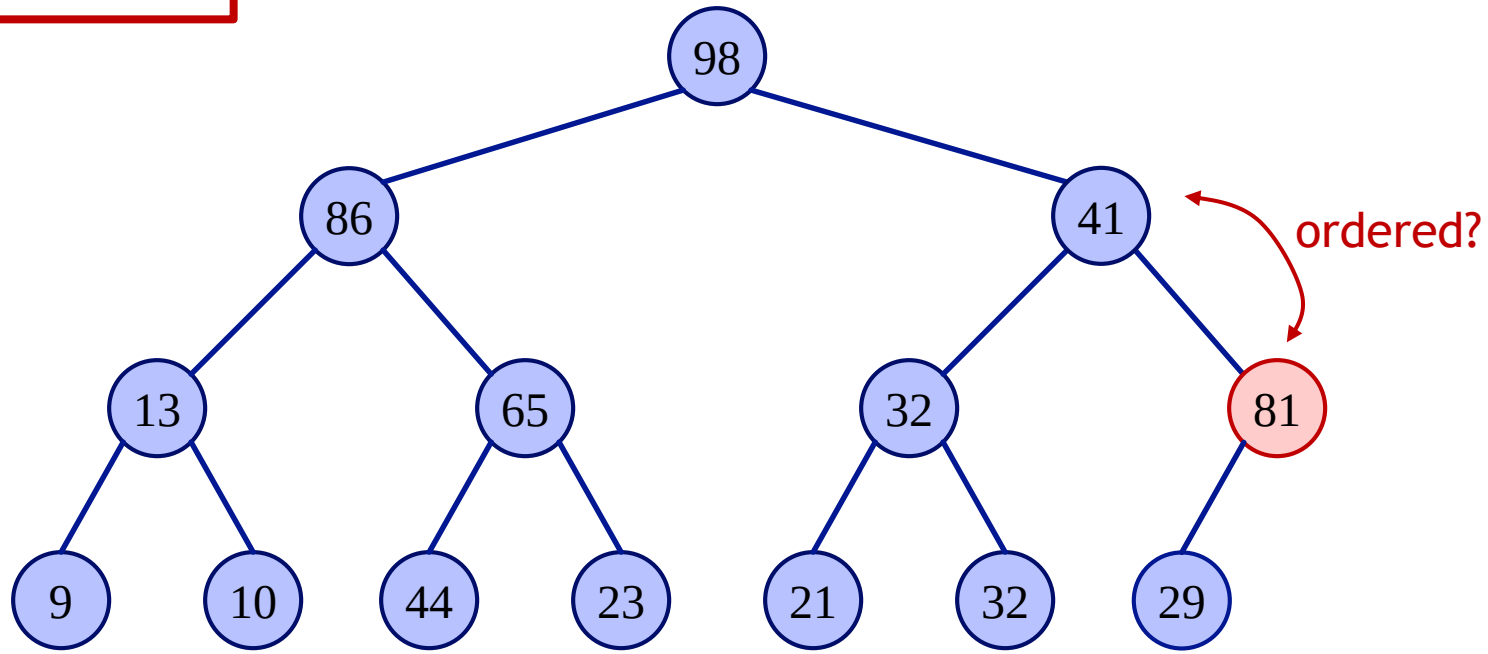


parent index $= \lfloor i/2 \rfloor = \lfloor (14)/2 \rfloor = 7$

value	--	98	86	41	13	65	32	29	9	10	44	23	21	32	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Insertion Example (Max Heap)

Insert 81

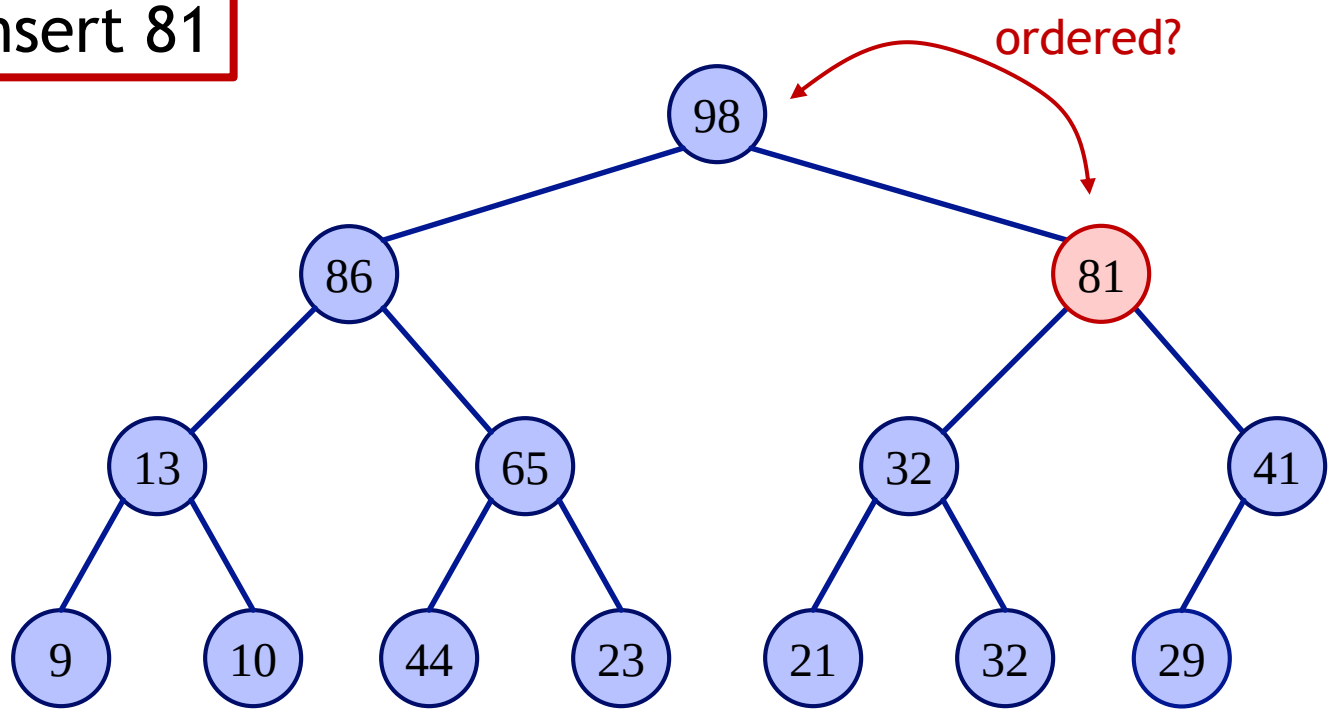


parent index = $\lfloor (7)/2 \rfloor = 3$

value	--	98	86	41	13	65	32	**81**	9	10	44	23	21	32	29
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Insertion Example (Max Heap)

Insert 81



81 is less than 98, so the insertion is complete!

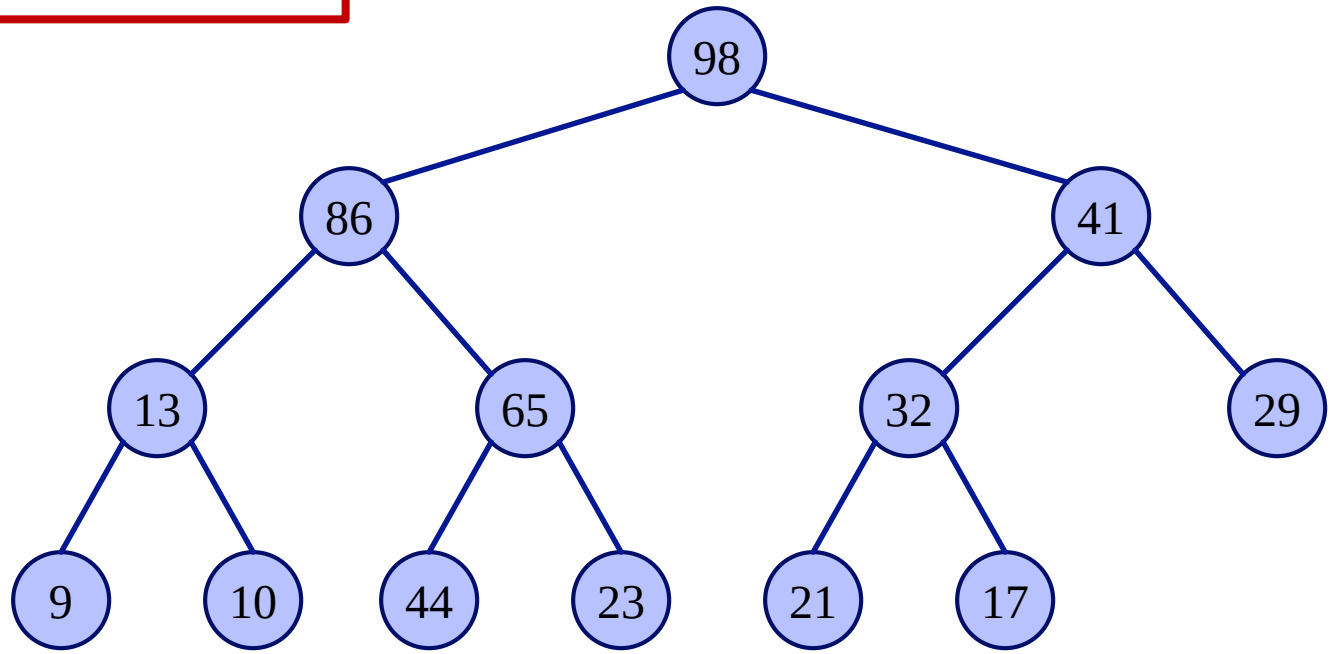
value	--	98	86	**81**	13	65	32	41	9	10	44	23	21	32	29
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Removal

- Heap properties need to be maintained
- *Heap-Shape Property:*
  - Swap the root with last element in the array (the right-most element on the bottom level of the tree)
  - The last element is removed, and the heap remains a *complete binary tree*
- *Heap-Order Property*
  - “Bubble” the element down the tree until the order property is restored
  - Repeatedly compare with children and swap until the ordering is restored

Heap Removal Example

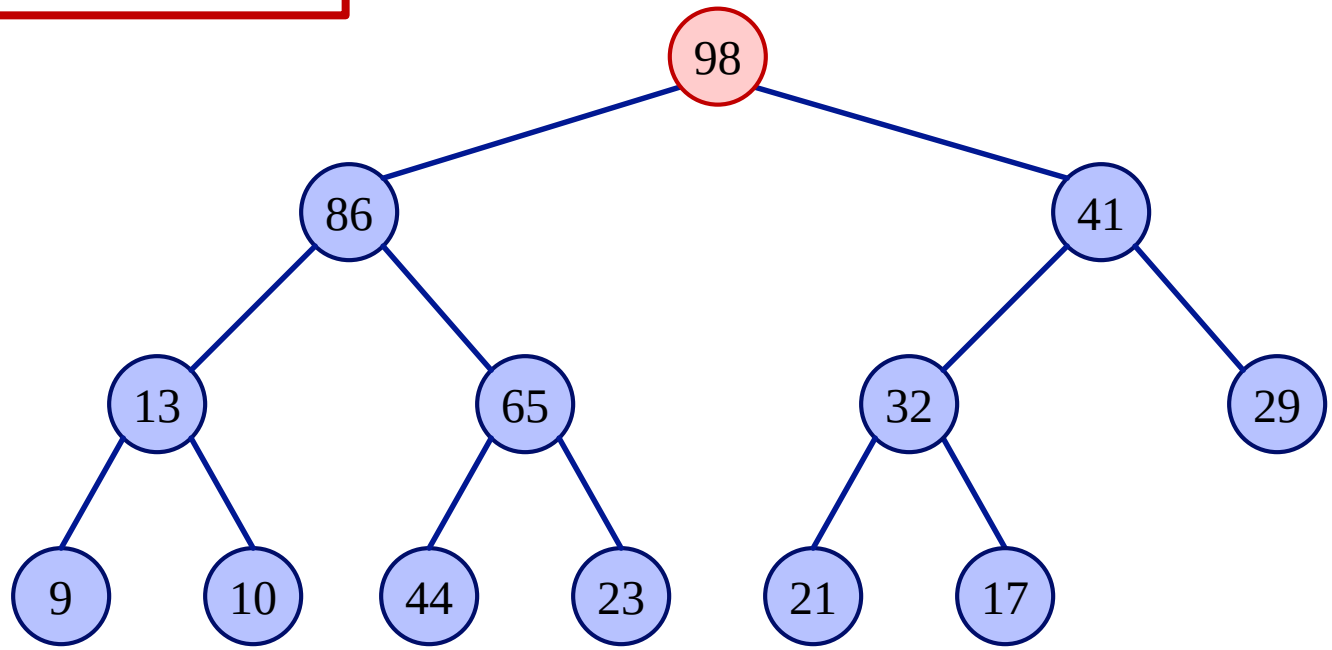
Remove max



value	--	98	86	41	13	65	32	29	9	10	44	23	21	17
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap Removal Example

Remove max

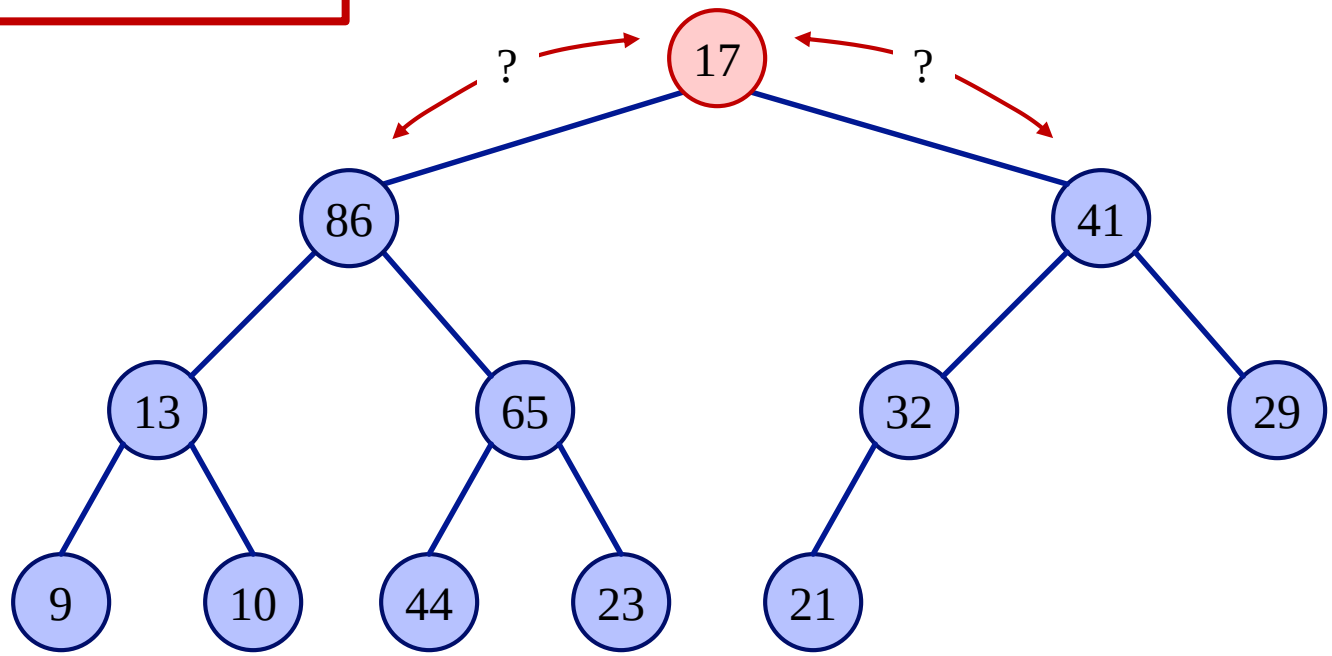


1. Replace root with rightmost leaf

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	--	98	86	41	13	65	32	29	9	10	44	23	21	17

Heap Removal Example

Remove max



1. Replace root with rightmost leaf
2. Swap with largest child

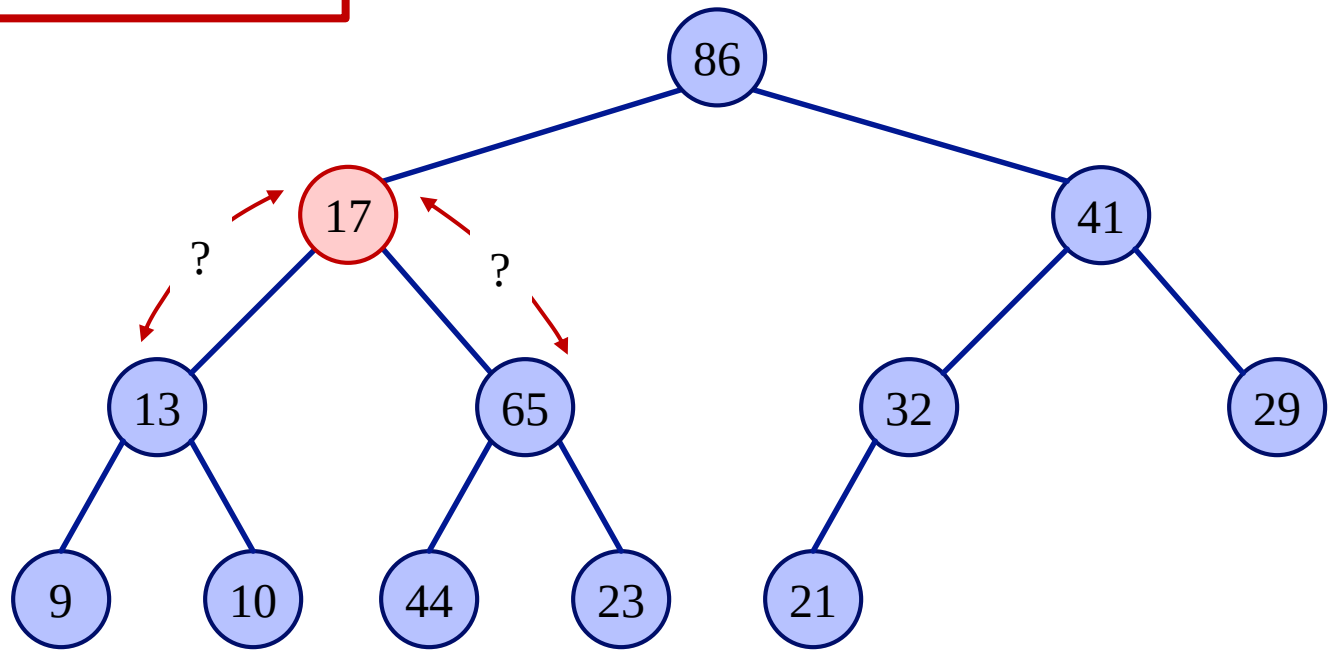
children of root (index 1): $2(1) = 1$, and $2(1) + 1 = 3$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	--	17	86	41	13	65	32	29	9	10	44	23	21	98

l child index: $2(i)$
r child index: $2(i) + 1$

Heap Removal Example

Remove max



1. Replace root with rightmost leaf
2. Swap with largest child
3. Repeat step 2 until ordered.

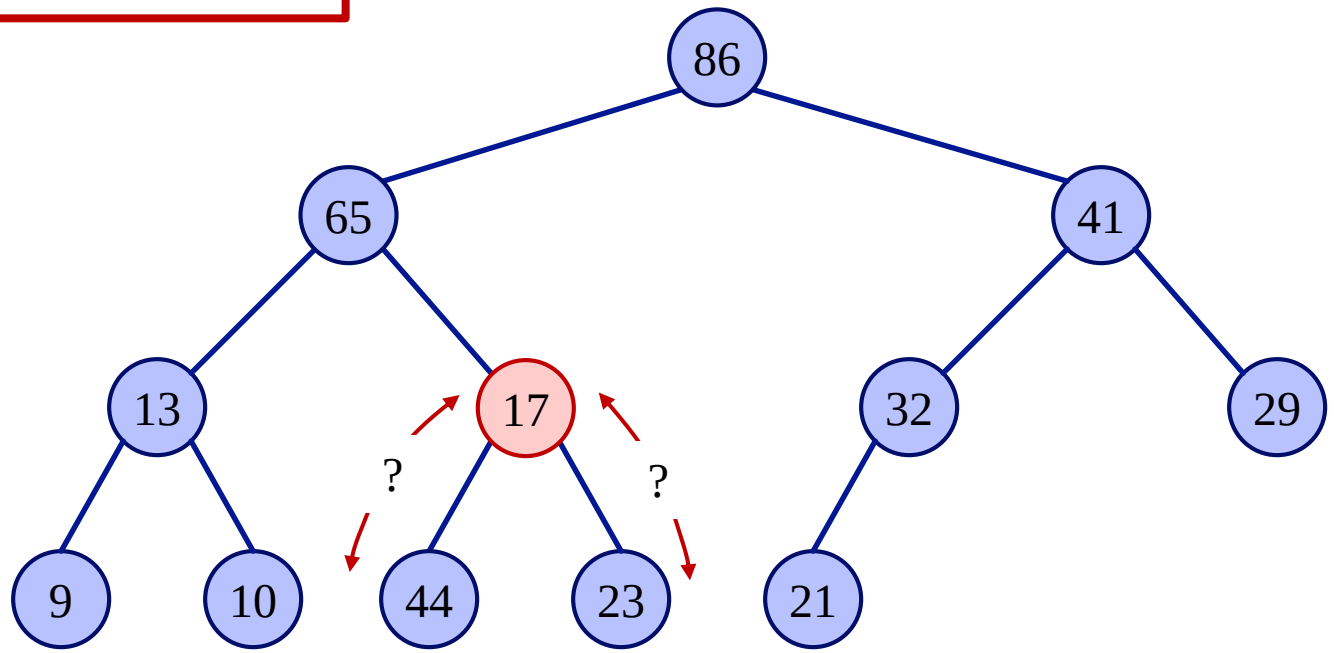
children of node (index 2): $2(2) = 4$, and $2(2) + 1 = 5$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	--	86	17	41	13	65	32	29	9	10	44	23	21	98

l child index: $2(i)$
r child index: $2(i) + 1$

Heap Removal Example

Remove max



1. Replace root with rightmost leaf
2. Swap with largest child
3. Repeat step 2 until ordered.

Done!

children of node (index 5): $2(5) = 10$, and $2(5) + 1 = 11$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
value	--	86	65	41	13	17	32	29	9	10	44	23	21	98

l child index: $2(i)$
r child index: $2(i) + 1$

Heaps - Recap

- We have seen that the height of a balanced binary tree is $O(\log n)$
- So the number of times the **Bubble Up** and **Bubble Down** operations will occur during an insertion or removal are $O(\log n)$
- Which means that Heap insertion and removal are $O(\log n)$ operations