Topic 5
Lecture 5
Heap and Priority Queue

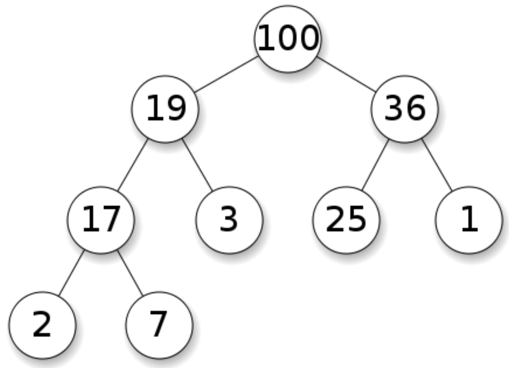
CSCI 240

Data Structures and Algorithms

Prof. Dominick Atanasio

Today

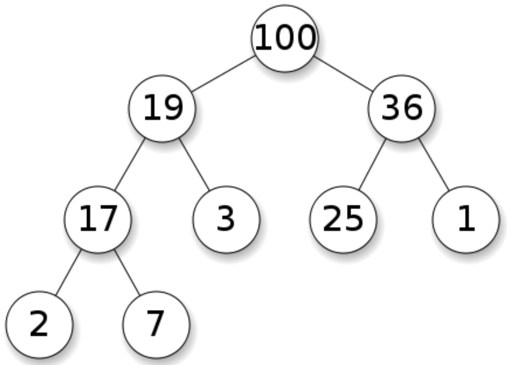
- This Class
  - Heap
    - Definition
    - Operations in Heap
    - Implementations
    - Efficiency of operations
  - Priority Queue



A max-heap

Heap

- A heap is a specialized tree-based data structure that satisfied the heap property
  - if B is a child node of A, then key(A) ≥ key(B).
- This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a max-heap.

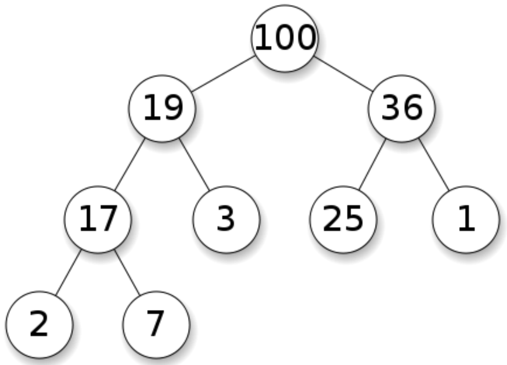


A max-heap

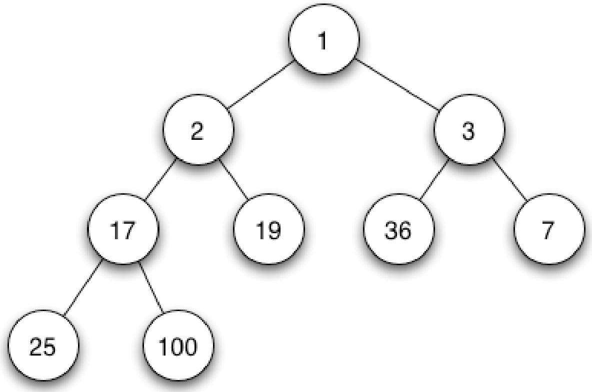
Heap

- A heap is a specialized tree-based data structure that satisfied the heap property
  - if B is a child node of A, then key(A) ≥ key(B).
- This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a max-heap.

- Of course, there's also a min-heap.



A max-heap



A min-heap

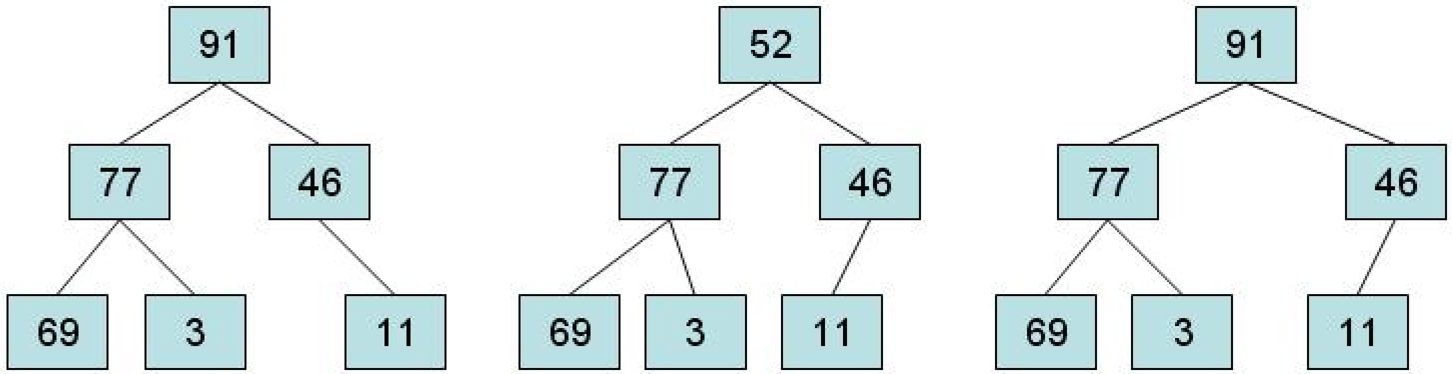
Heap

- A heap implemented with a binary tree in which the following two rules are followed:
  - The element contained by each node is greater than or equal to the elements of that node's children.
  - The tree is a complete binary tree

Heap

- A heap implemented with a binary tree in which the following two rules are followed:
  - The element contained by each node is greater than or equal to the elements of that node's children.
  - The tree is a complete binary tree

- Example: which one is a max-heap?



Operations in Heap

Interface for a max-heap

```
template<typename T>
struct MaxHeap
{
      virtual void add(T item) = 0;
      virtual T max() = 0;
      virtual T removeMax() = 0;
      virtual void clear() = 0;
      virtual bool empty() = 0;
      virtual size_t size() = 0;
};
```

Interface for a min-heap

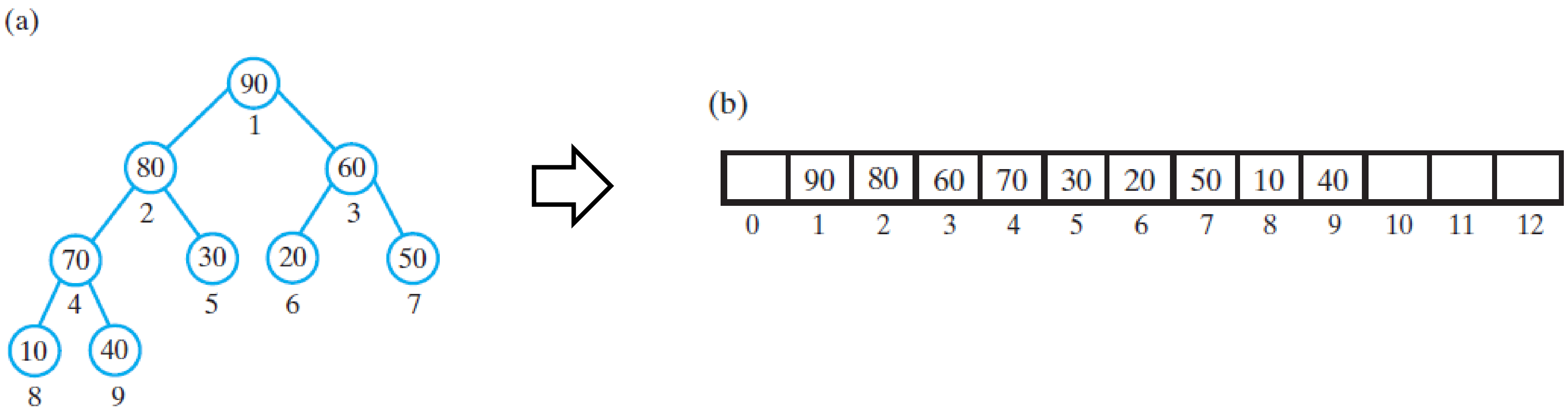
```
template<typename T>
struct MinHeap
{
      virtual void add(T item) = 0;
      virtual T min() = 0;
      virtual T removeMin() = 0;
      virtual void clear() = 0;
      virtual bool empty() = 0;
      virtual size_t size() = 0;
};
```

Heap Implementation

- A more common approach is to store the heap in an array.
  - Since heap is always a complete binary tree, it can be stored compactly.
  - The parent and children of each node can be found by simple arithmetic on array indices.
- A heap could be stored using linked Nodes
  - less efficient in terms of special locality and space consumption.

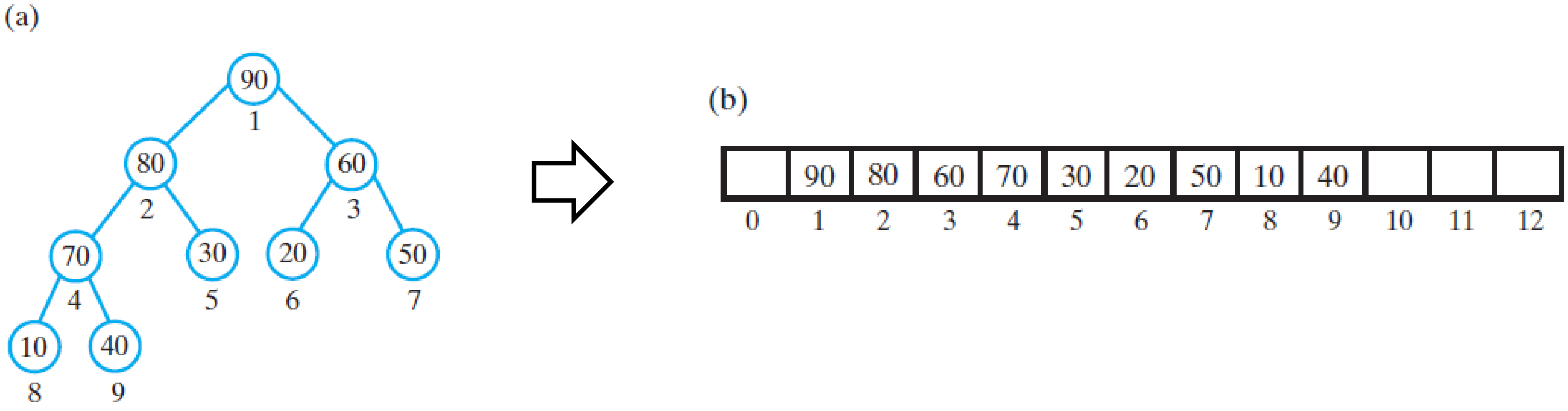
Heap Implementation

- The parent and children of each node can be found by simple arithmetic on array indices. (we begin the heap at index 1, to be consistent with textbook)



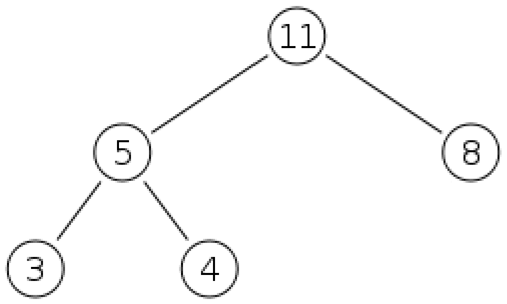
Heap Implementation

- The parent and children of each node can be found by simple arithmetic on array indices. (we begin the heap at index 1, to be consistent with textbook)



A complete binary tree with its nodes numbered in level order

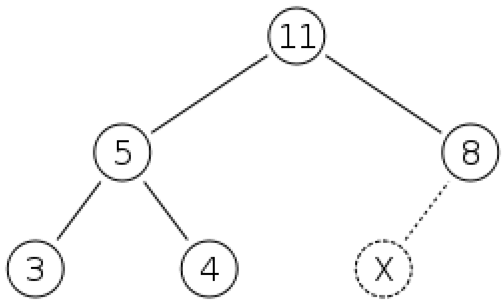
Adding an Entry to Heap



mheap.add(15)

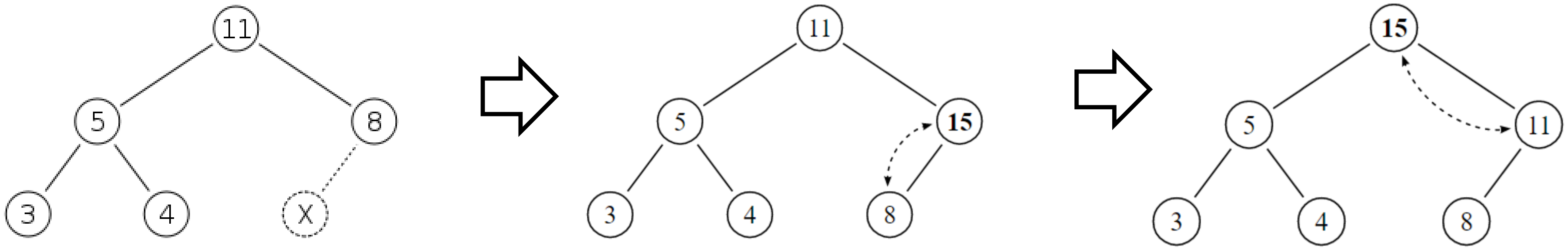
Adding an Entry to Heap

- Perform an up-heap operation
  - Add the element to the bottom level of the heap.
  - Compare the added element with its parent; if they are in the correct order, stop.
  - If not, swap the element with its parent and return to the previous step.



Adding an Entry to Heap

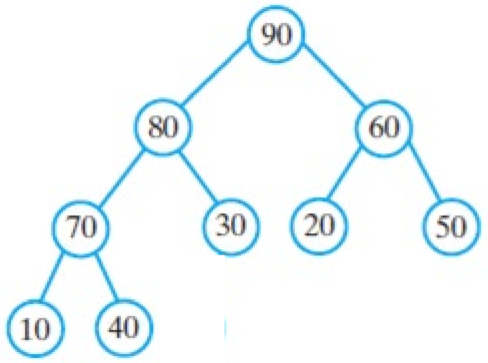
- Perform an up-heap operation
  - Add the element to the bottom level of the heap.
  - Compare the added element with its parent; if they are in the correct order, stop.
  - If not, swap the element with its parent and return to the previous step.



https://en.wikipedia.org/wiki/Binary_heap

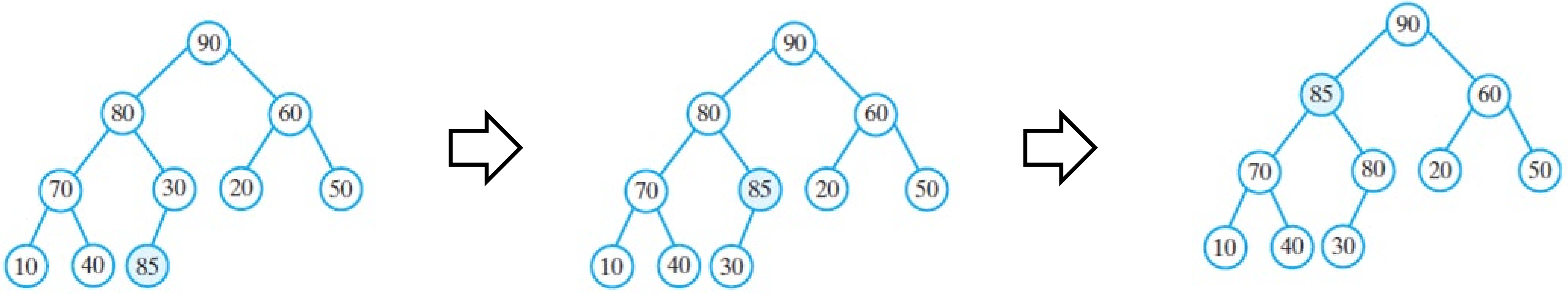
In-Class Exercises

- Add 85 to the following max-heap



In-Class Exercises

- Add 85 to the following max-heap



Adding an Entry to Heap

Add 85 to the max-heap

(a)

	90	80	60	70	**30**	20	50	10	40			
0	1	2	3	4	5 (10/2)	6	7	8	9	10	11	12

85 > 30

(b)

	90	80	60	70		20	50	10	40	**30**		
0	1	2	3	4	5	6	7	8	9	10	11	12

Move 30

(c)

	90	**80**	60	70		20	50	10	40	30		
0	1	2 (5/2)	3	4	5	6	7	8	9	10	11	12

85 > 80

(d)

	90		60	70	**80**	20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12

Move 80

(e)

	90		60	70	80	20	50	10	40	30		
0	1 (2/2)	2	3	4	5	6	7	8	9	10	11	12

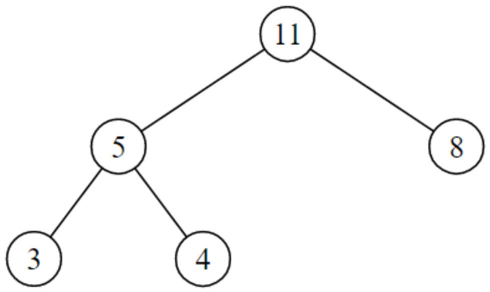
85 < 90

(f)

	90	**85**	60	70	80	20	50	10	40	30		
0	1	2	3	4	5	6	7	8	9	10	11	12

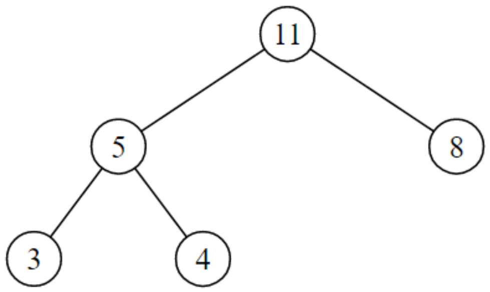
Insert 85

Removing the Root



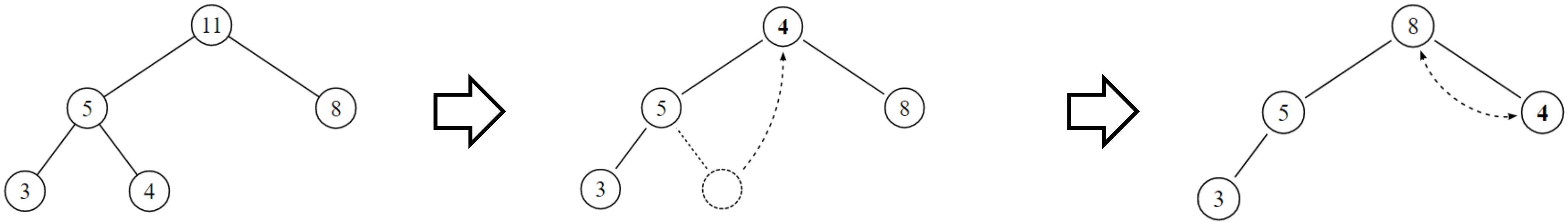
Removing the Root

- Perform a down-heap operation
  - Replace the root of the heap with the last element on the last level.
  - Compare the new root with its children; if they are in the correct order, stop.
  - If not, swap the element with one of its children and return to the previous step.
    - Swap with its smaller child in a min-heap and its larger child in a max-heap.



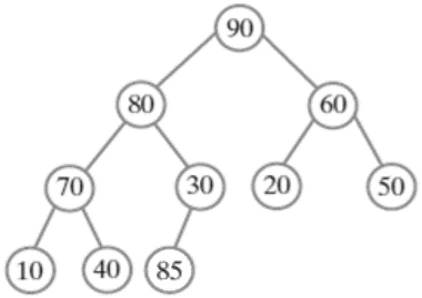
Removing the Root

- Perform a heapify operation
  - Replace the root of the heap with the last element on the last level.
  - Compare the new root with its children; if they are in the correct order, stop.
  - If not, swap the element with one of its children and return to the previous step.
    - Swap with its smaller child in a min-heap and its larger child in a max-heap.



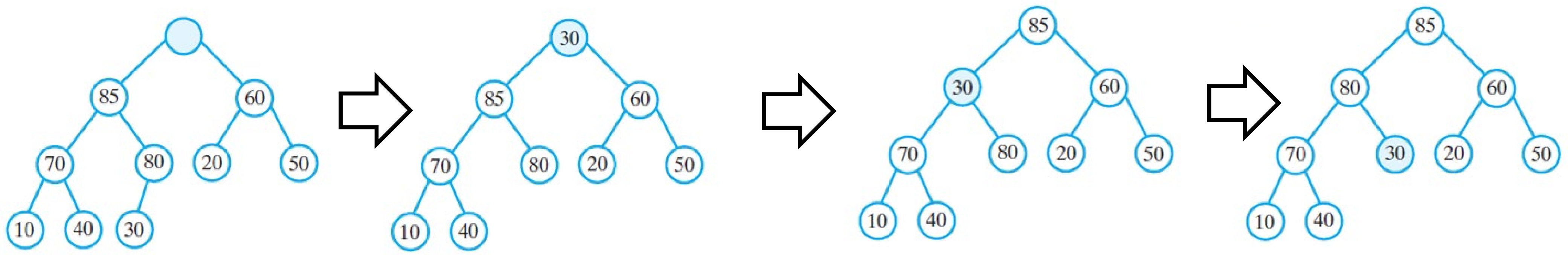
In-Class Exercises

- Remove 90 from the following max-heap



In-Class Exercises

- Remove 90 from the following max-heap

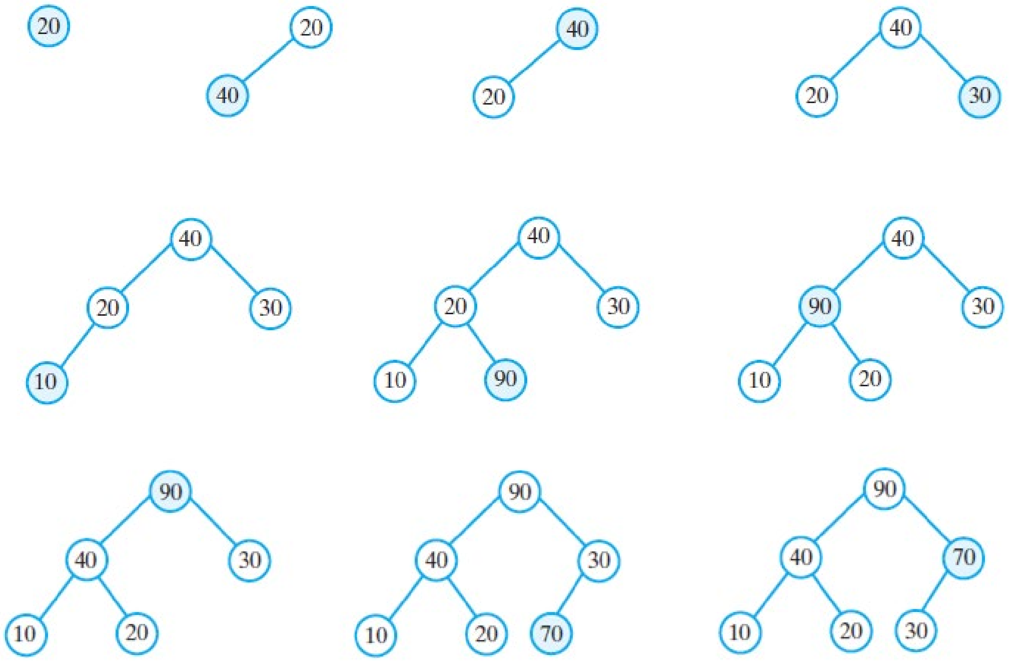


Creating a Heap

- Using the add method to add each object to an initially empty heap
  - Consider integers: 20, 40, 30, 10, 90, 70

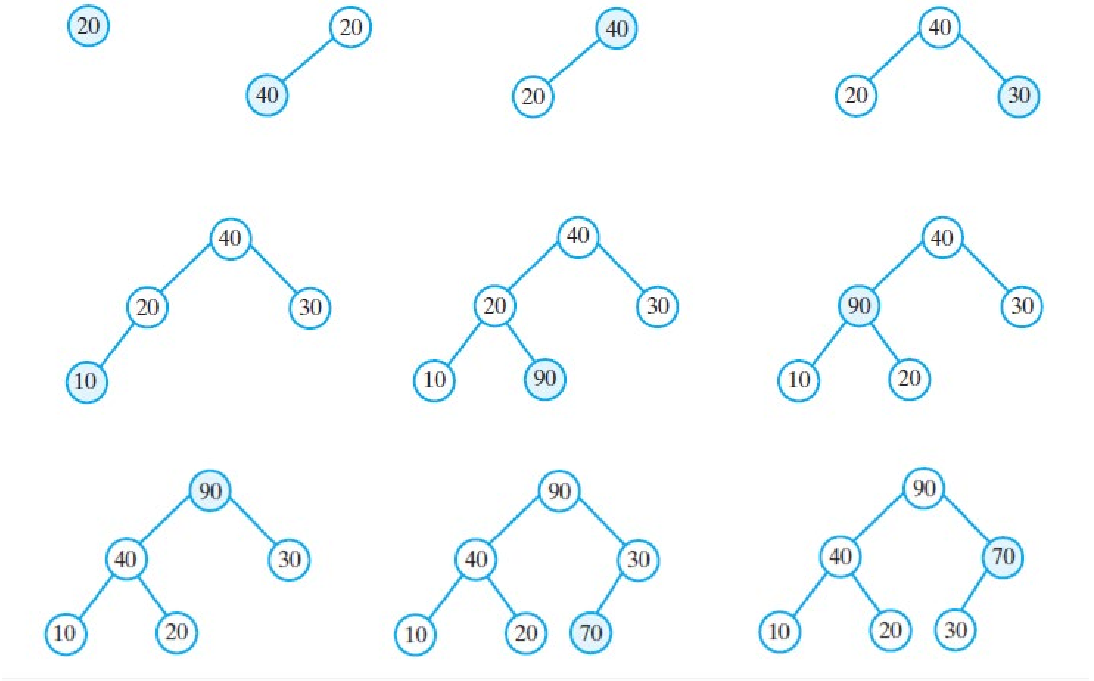
Creating a Heap

- Using the add method to add each object to an initially empty heap
  - Consider integers: 20, 40, 30, 10, 90, 70



Creating a Heap

- Using the add function to add each object to an initially empty heap

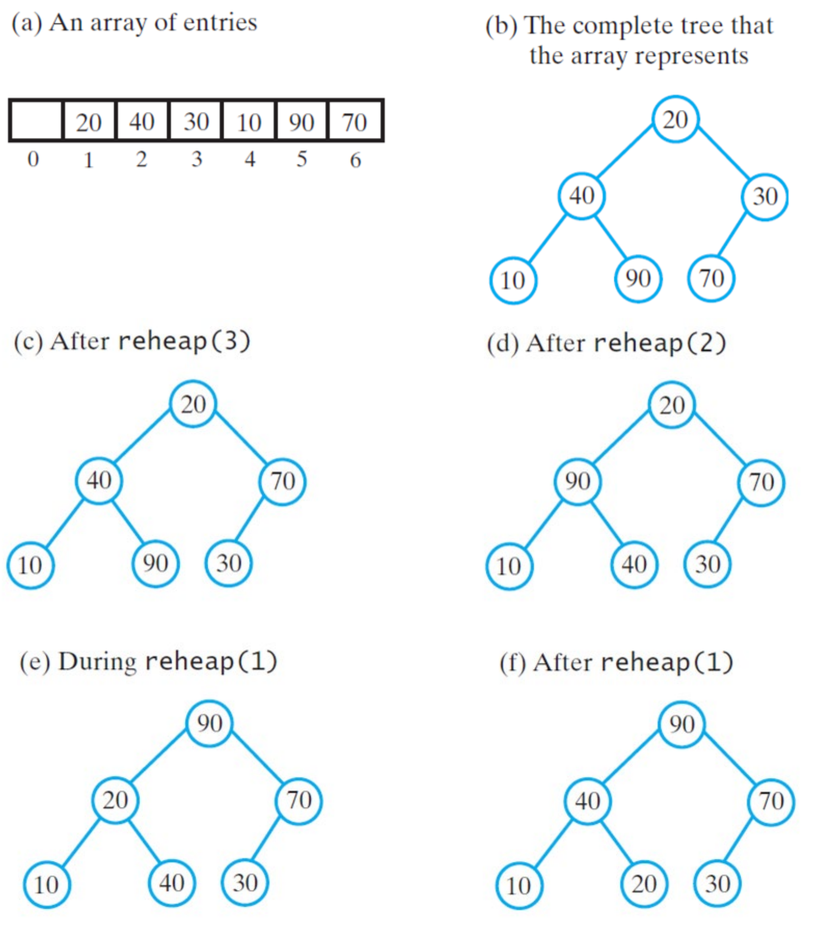


Time complexity:

- Method add is an O(log n) operation,
- Using add to create the heap is O(n log n)

Creating a Heap

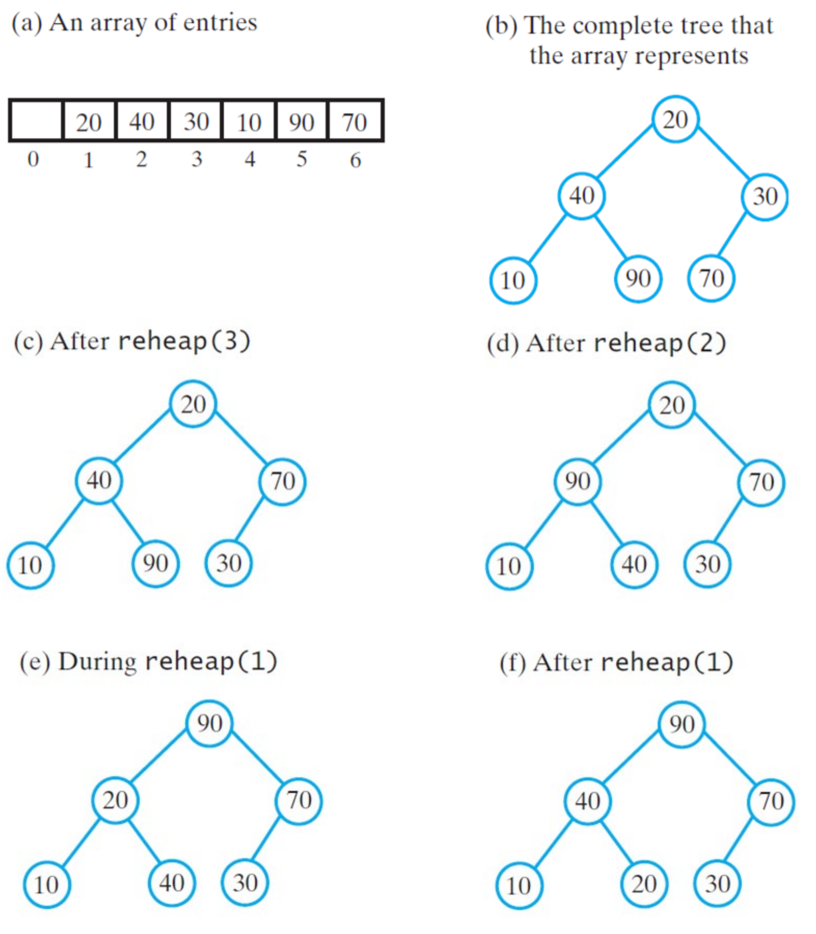
- A “Smart” way: create a heap uses the method reheap (heapify)
- Call upheap on the last non-leaf node, then the second to last and so on.



Repeatedly apply heapify operation on all non-leaf nodes

Creating a Heap

- A “Smart” way: create a heap uses the method reheap (heapify)
- Call upheap on the last non-leaf node, then the second to last and so on.



Since reheap is an O(h_i) operation, where h_i is the height of the subtree rooted at index i.

The time complexity of using add to create the heap is O(2^h), such that O(n), where h is the height of the full tree with n nodes.

When is heapify useful?

- Often, the implementation of a min-heap or max-heap interface includes a constructor that accepts an array as an argument.
- The constructor copies the elements of the array into the heap then calls heapify on the heap.
- The last non-leaf node can be found with: $n_p = ceil(\frac{(n-1)}{2})$

In-Class Exercises

- Creating a max-heap with 27, 35, 23, 22, 4, 45, 21, 5, 42 and 19.
- The last non-leaf node can be found with: $n_p = ceil(\frac{(n-1)}{2})$

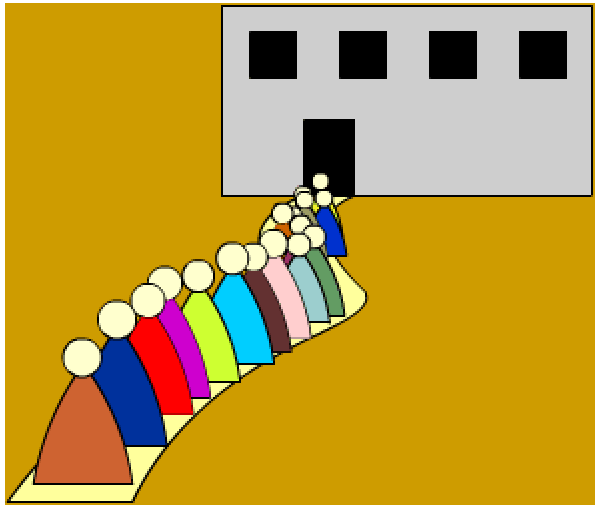
In-Class Exercises

- Start with an empty max-heap and enter 10 elements with priories 1 through 8. Draw the resulting max-heap. (Note: A higher priority is a larger number)
- Remove three elements from the heap you created in the above heap. Draw the resulting heap.

Priority Queues

- In general, a queue is:
  - A linear data structure with two access points: front and rear
  - Items can only be inserted to the rear (enqueue) and retrieved from the front (dequeue)
  - Dynamic length
  - The First-In, First-Out rule (FIFO)

In some situations, some tasks may be more important than others. We need to consider their priority.



Priority Queues

- A priority queue behaves much like an ordinary queue:
  - Elements are placed in the queue and later taken out.
  - But each element in a priority queue has an associated number called its priority.
    - This could be accomplished with a <key, value> pair
    - The key is used for the priority
  - When elements leave a priority queue, the highest priority element always leaves first.
    - Note, the key with the lowest value could have the highest priority.
- A heap is the most efficient implementation of priority queues.
  - It is possible to pass in a function that determines the priority of the key.
  - Could be passed in on instantiation of the priority queue.
  - This will make the implementation more generalized.

Implementation of Priority Queues

- Using a Heap
  - Each node of the heap contains one element along with the element's priority,
  - The tree is maintained so that it follows the heap storage rules using the element's priorities to compare nodes:
    - The element contained by each node has a priority that is greater than or equal to the priorities of the elements of that node's children.
    - The tree is a complete binary tree.
    - The heap is most often implemented with an array of key/value pairs where the key is the priority, and the value is that which is prioritized
    - Of course, it can hold single vales that act as both the priority and prioritized.