CS2400 - Data Structures and Advanced Programming
Module 9: Trees (II) - Heaps

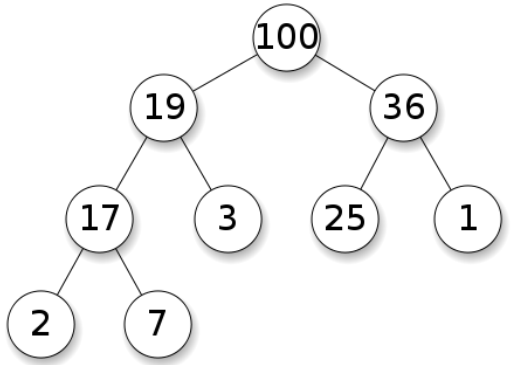
Hao Ji

Computer Science Department

Cal Poly Pomona

Today

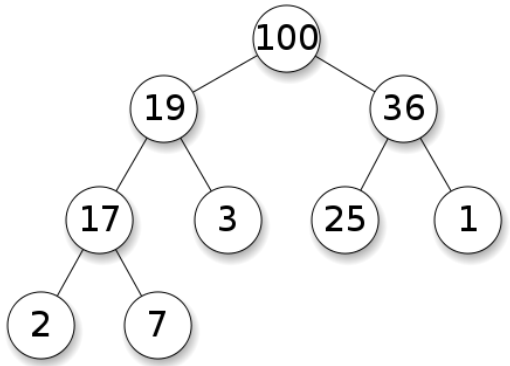
- This Class
  - Heap
    - Definition
    - Operations in Heap
    - Implementations
    - Efficiency of operations



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.

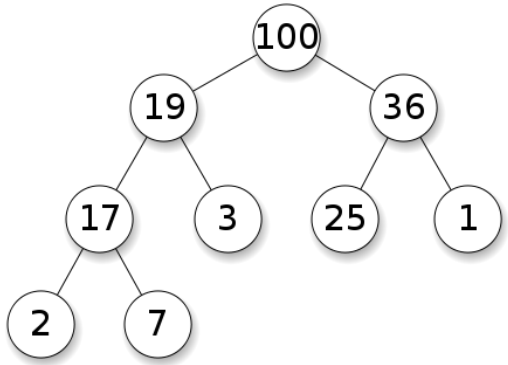


A max-heap

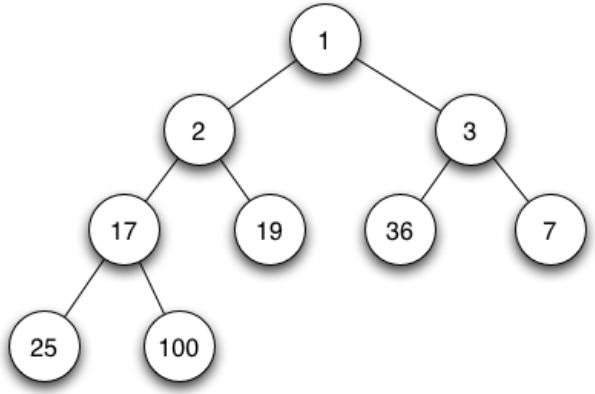
Heap

- 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

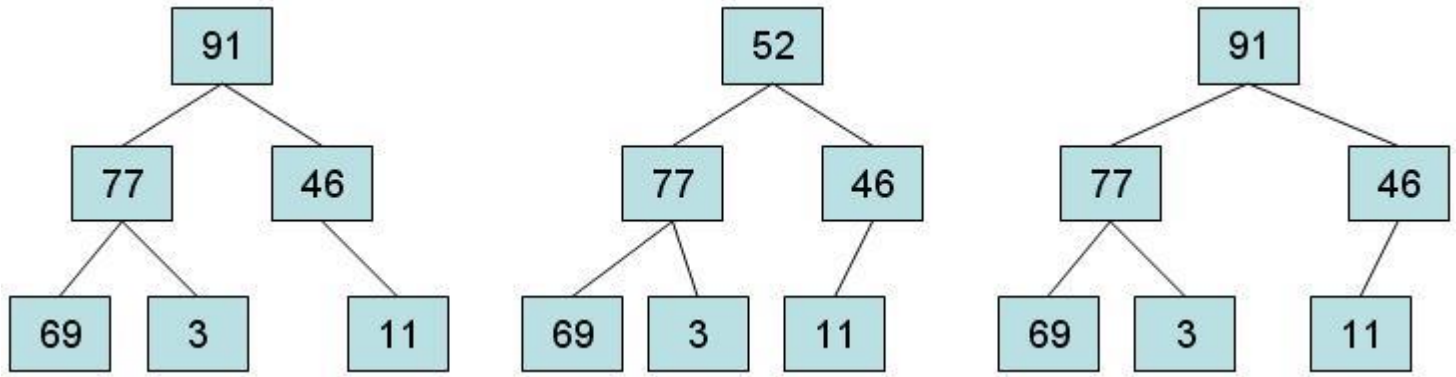
Heap

- 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

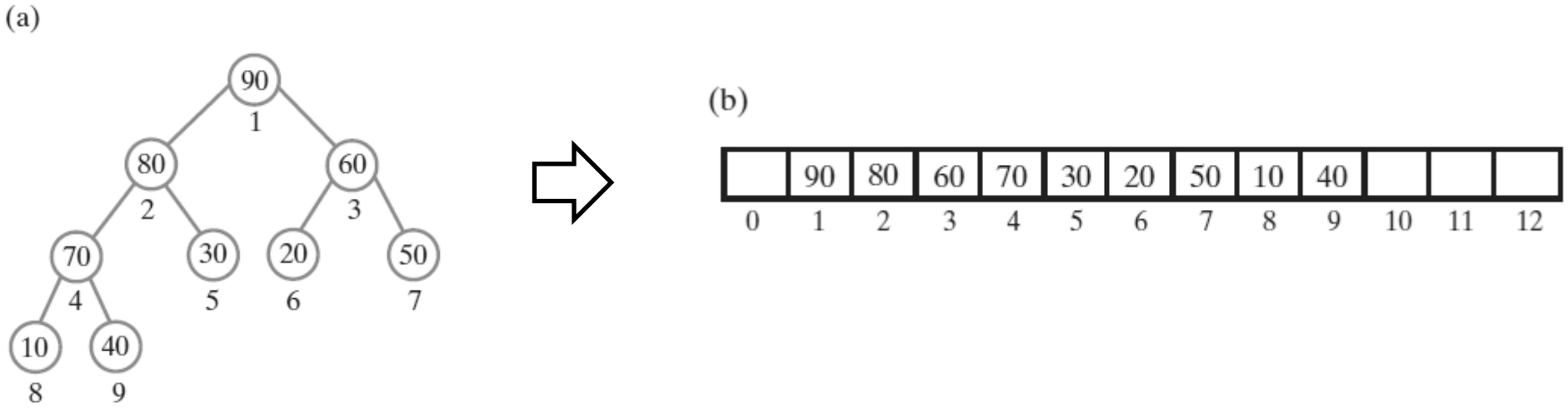
```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
   public void add(T newEntry);
   public T removeMax();
   public T getMax();
   public boolean isEmpty();
   public int getSize();
   public void clear();
} // end MaxHeapInterface
```

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.

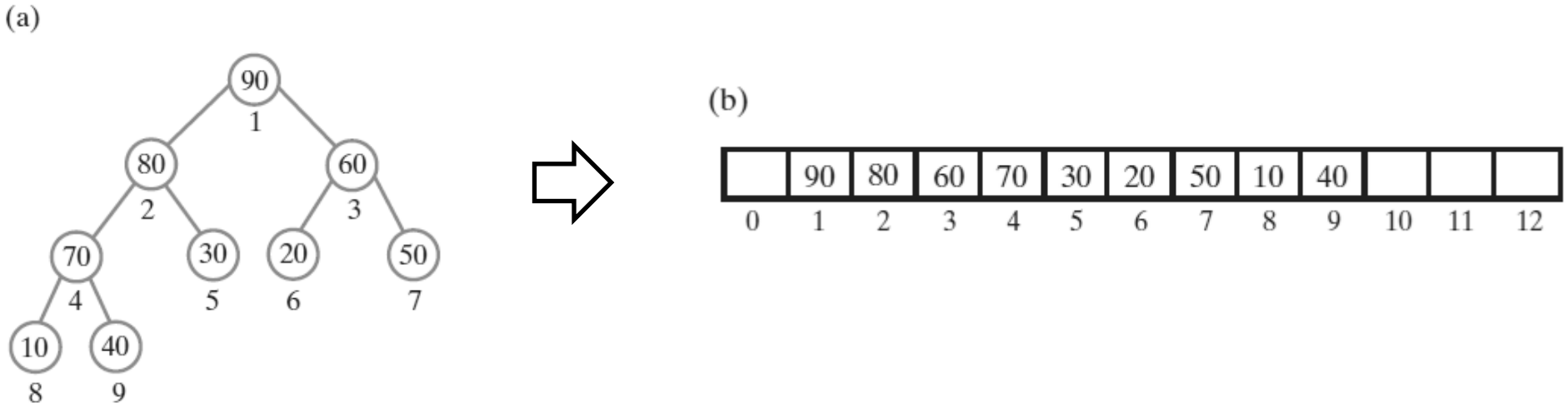
Heap Implementation

- 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)

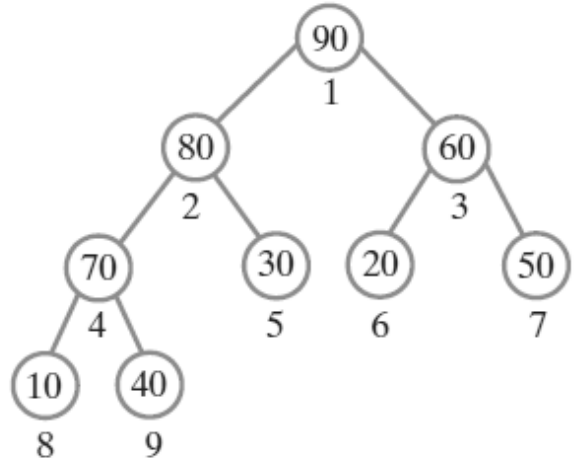


A complete binary tree with its nodes numbered in level order

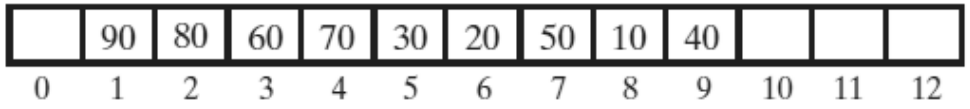
Heap Implementation

```java
import java.util.Arrays;
public final class MaxHeap<T extends Comparable<? super T>>
                implements MaxHeapInterface<T>
{
   private T[] heap;       // Array of heap entries
   private int lastIndex; // Index of last entry
   private boolean initialized = false;
   private static final int DEFAULT_CAPACITY = 25;
   private static final int MAX_CAPACITY = 10000;

   public MaxHeap()
   {
      this(DEFAULT_CAPACITY); // Call next constructor
   } // end default constructor
   public MaxHeap(int initialCapacity)
   {
      // Is initialCapacity too small?
      if (initialCapacity < DEFAULT_CAPACITY)
         initialCapacity = DEFAULT_CAPACITY;
      else // Is initialCapacity too big?
         checkCapacity(initialCapacity);

      // The cast is safe because the new array contains all null entries
      @SuppressWarnings("unchecked")
      T[] tempHeap = (T[]) new Comparable[initialCapacity + 1];
      heap = tempHeap;
      lastIndex = 0;
      initialized = true;
   } // end constructor
```

(a)



(b)

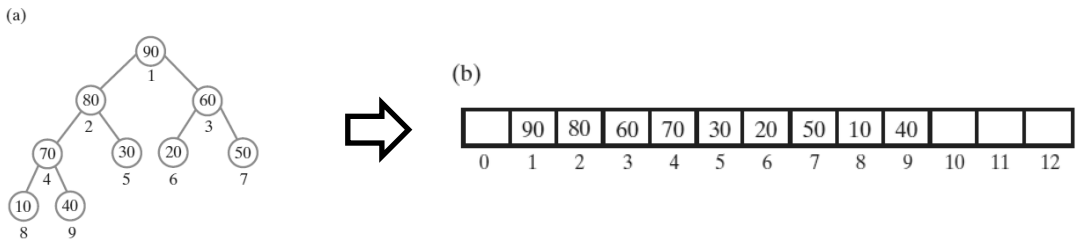


Methods in Heap

- Implement the methods: getMax(), isEmpty(), getSize(); clear();

```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
   public void add(T newEntry);
   public T removeMax();
   public T getMax();
   public boolean isEmpty();
   public int getSize();
   public void clear();
} // end MaxHeapInterface
```

Interface




```
1  import java.util.Arrays;
2  public final class MaxHeap<T extends Comparable<? super T>>
3                implements MaxHeapInterface<T>
4  {
5     private T[] heap;      // Array of heap entries
6     private int lastIndex; // Index of last entry
7     private boolean initialized = false;
8     private static final int DEFAULT_CAPACITY = 25;
9     private static final int MAX_CAPACITY = 10000;
```

Private fields in the class

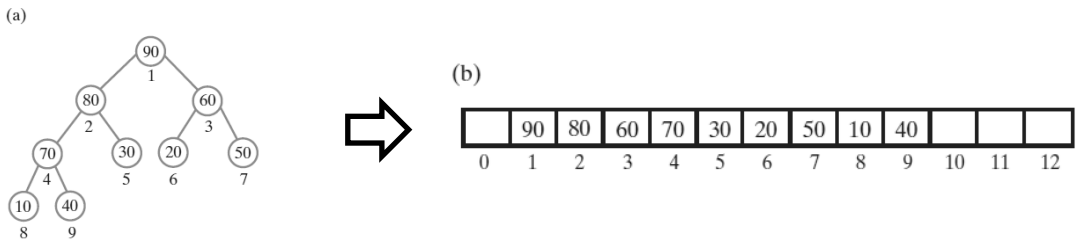
Methods in Heap

- Implement the methods: getMax(), isEmpty(), getSize(); clear();

```
42    public T getMax()
43    {
44       checkInitialization();
45       T root = null;
46       if (!isEmpty())
47          root = heap[1];
48       return root;
49    } // end getMax
50
51    public boolean isEmpty()
52    {
53       return lastIndex < 1;
54    } // end isEmpty
55
56    public int getSize()
57    {
58       return lastIndex;
59    } // end getSize
60
61    public void clear()
62    {
63       checkInitialization();
64       while (lastIndex > -1)
65       {
66          heap[lastIndex] = null;
67          lastIndex--;
68       } // end while
69       lastIndex = 0;
70    } // end clear
```

```
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
   public void add(T newEntry);
   public T removeMax();
   public T getMax();
   public boolean isEmpty();
   public int getSize();
   public void clear();
} // end MaxHeapInterface
```

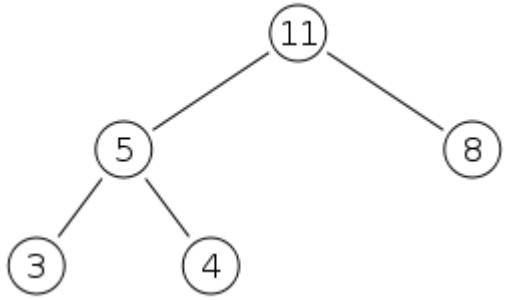
Interface




```
1  import java.util.Arrays;
2  public final class MaxHeap<T extends Comparable<? super T>>
3                implements MaxHeapInterface<T>
4  {
5     private T[] heap;       // Array of heap entries
6     private int lastIndex; // Index of last entry
7     private boolean initialized = false;
8     private static final int DEFAULT_CAPACITY = 25;
9     private static final int MAX_CAPACITY = 10000;
```

Private fields in the class

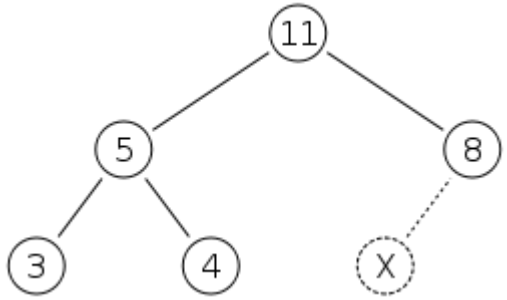
Adding an Entry to Heap



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

Adding an Entry to Heap

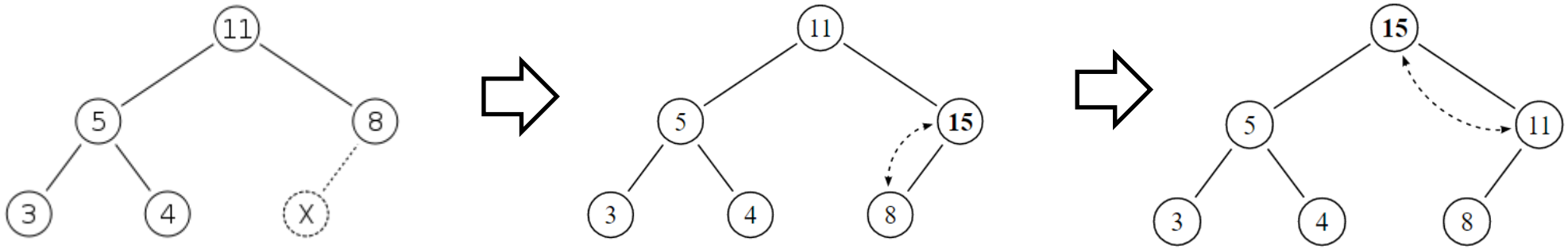
- Perform an *up-heap* operation
  - Add the element to the bottom level of the heap.
  - Compare the added element with its parent; if they are in the correct order, stop.
  - If not, swap the element with its parent and return to the previous step.



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

Adding an Entry to Heap

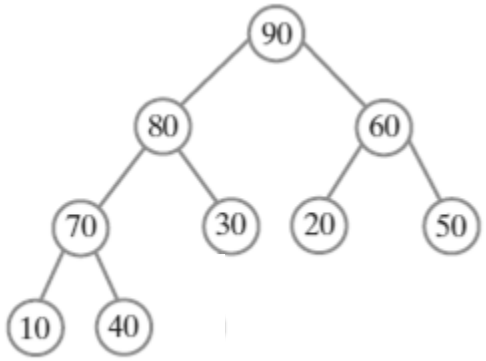
- Perform an *up-heap* operation
  - Add the element to the bottom level of the heap.
  - Compare the added element with its parent; if they are in the correct order, stop.
  - If not, swap the element with its parent and return to the previous step.



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

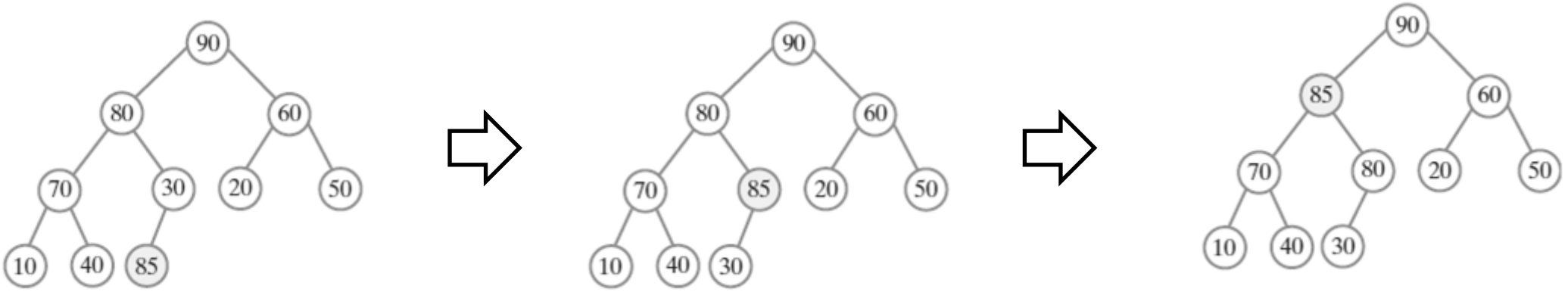
In-Class Exercises

- Add 85 to the following max-heap

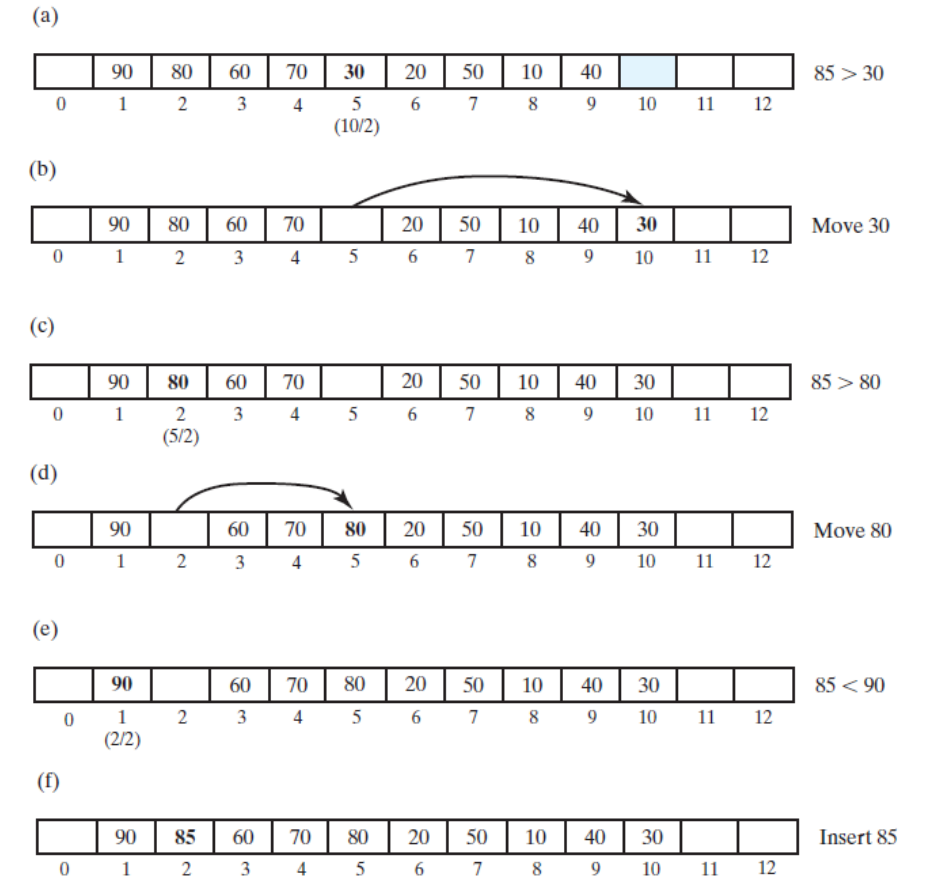


In-Class Exercises

- Add 85 to the following max-heap



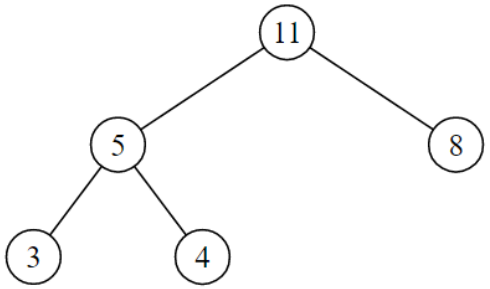
Adding an Entry to Heap



Add 85 to the max-heap

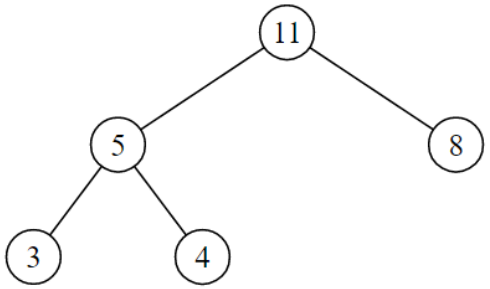
```
public void add(T newEntry)
{
   checkInitialization();         // Ensure initialization of data fields
   int newIndex = lastIndex + 1;
   int parentIndex = newIndex / 2;
   while ( (parentIndex > 0) && newEntry.compareTo(heap[parentIndex]) > 0)
   {
      heap[newIndex] = heap[parentIndex];
      newIndex = parentIndex;
      parentIndex = newIndex / 2;
   } // end while
   heap[newIndex] = newEntry;
   lastIndex++;
   ensureCapacity();
} // end add
```

Removing the Root



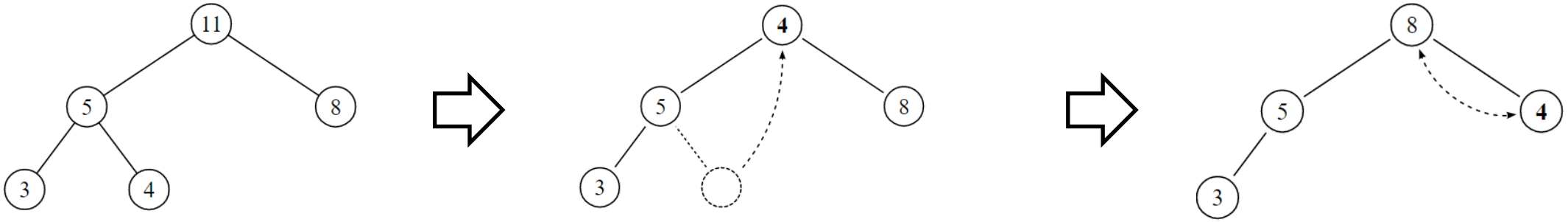
Removing the Root

- 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.)



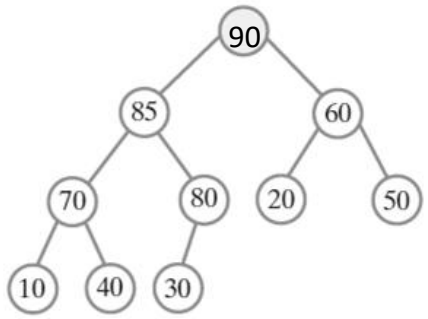
Removing the Root

- Perform a *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.)



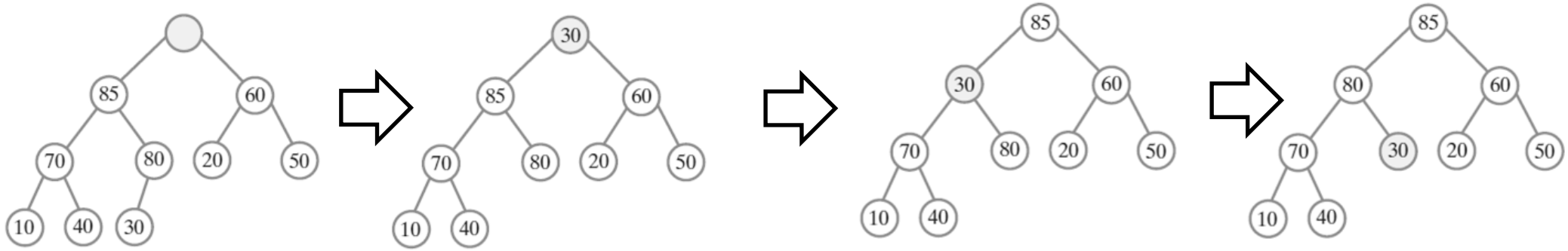
In-Class Exercises

- Remove 90 from the following max-heap



In-Class Exercises

- Remove 90 from the following max-heap



Removing the Root

```
public T removeMax()
{
   checkInitialization();        // Ensure initialization of data fields
   T root = null;

   if (!isEmpty())
   {
      root = heap[1];            // Return value
      heap[1] = heap[lastIndex]; // Form a semiheap
      lastIndex--;               // Decrease size
      reheap(1);                 // Transform to a heap
   } // end if

   return root;
} // end removeMax
```

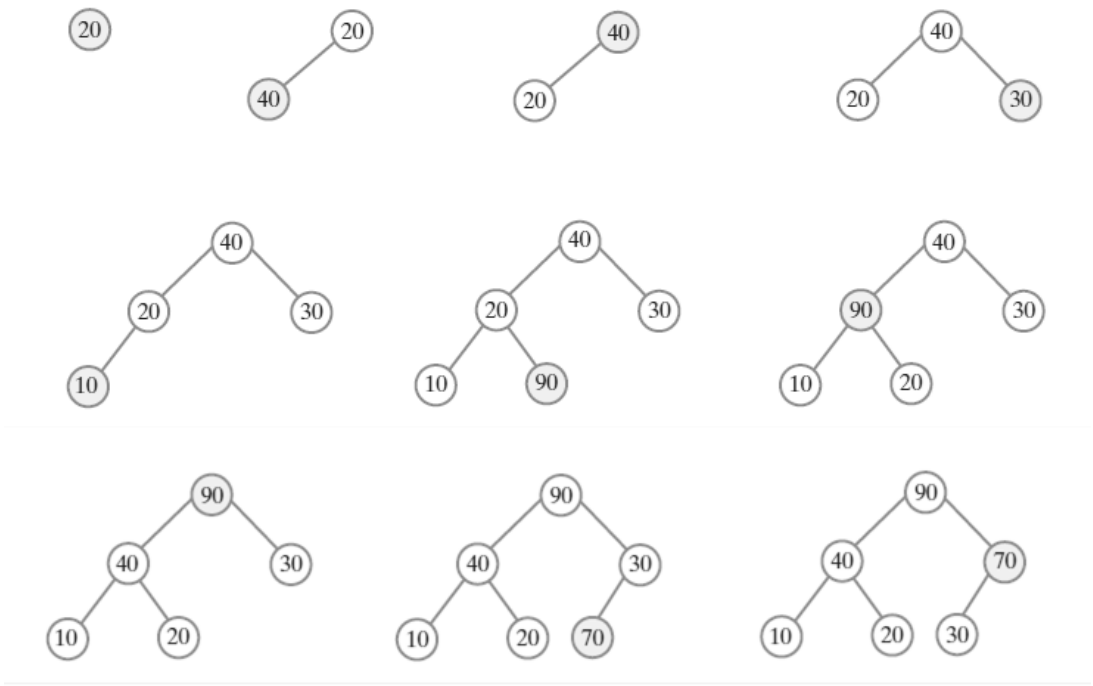
```
private void reheap(int rootIndex)
{
   boolean done = false;
   T orphan = heap[rootIndex];
   int leftChildIndex = 2 * rootIndex;

   while (!done && (leftChildIndex <= lastIndex) )
   {
      int largerChildIndex = leftChildIndex; // Assume larger
      int rightChildIndex = leftChildIndex + 1;
      if ( (rightChildIndex <= lastIndex) &&
           heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0)
      {
         largerChildIndex = rightChildIndex;
      } // end if

      if (orphan.compareTo(heap[largerChildIndex]) < 0)
      {
         heap[rootIndex] = heap[largerChildIndex];
         rootIndex = largerChildIndex;
         leftChildIndex = 2 * rootIndex;
      }
      else
         done = true;
   } // end while

   heap[rootIndex] = orphan;
} // end reheap
```

Creating a Heap

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

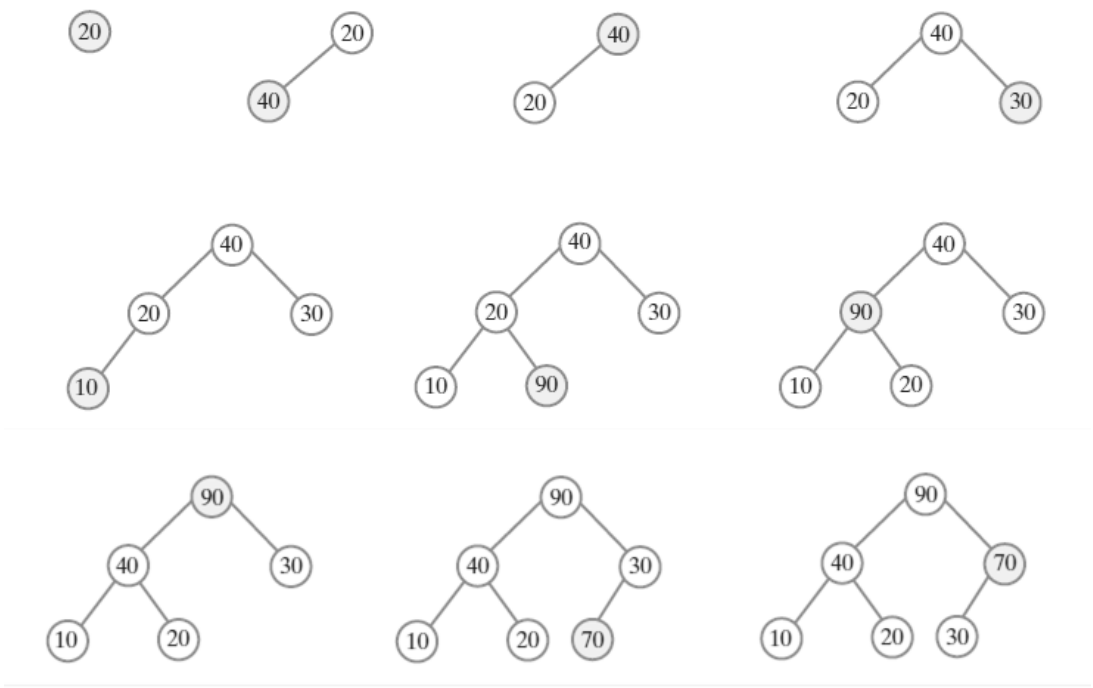
Creating a Heap

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



Creating a Heap

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

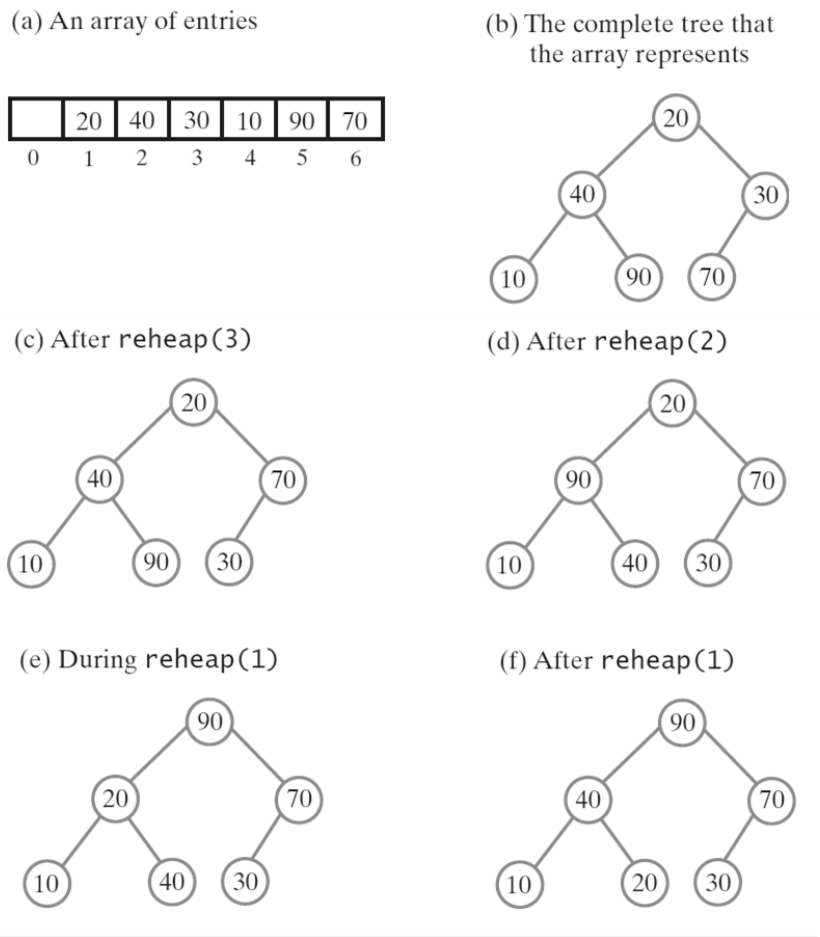


Time complexity:

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

Creating a Heap

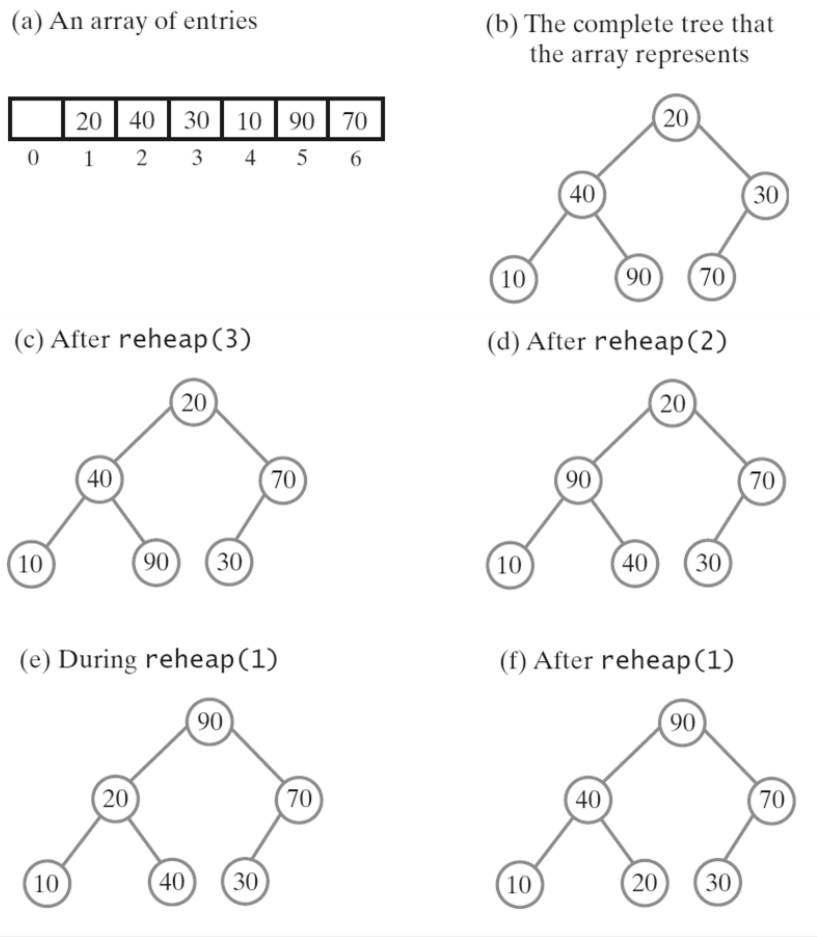
- A “Smart” way: create a heap uses the method reheap



Repeatedly apply reheap operation on all non-leaf nodes

Creating a Heap

- A “Smart” way: create a heap uses the method reheap



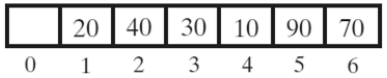
Since reheap is an O(h_i) operation, where h_i is the height of the subtree rooted at index i.

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.

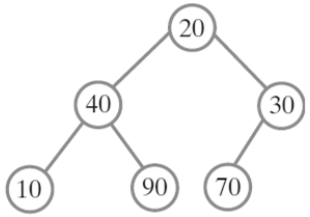
Creating a Heap

- A “Smart” way: create a heap uses the method reheap

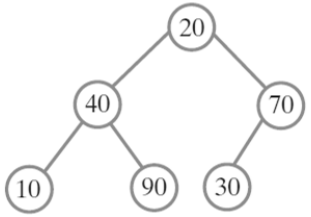
(a) An array of entries



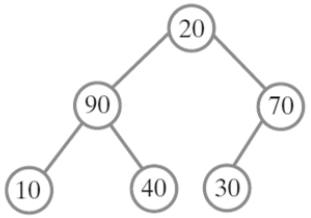
(b) The complete tree that the array represents



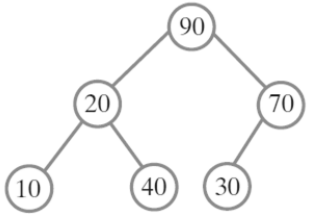
(c) After `reheap(3)`



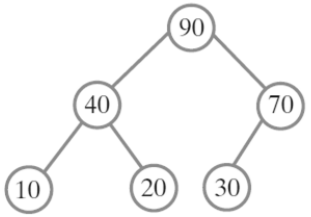
(d) After `reheap(2)`



(e) During `reheap(1)`



(f) After `reheap(1)`



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.

By the observation at the end of Segment 26.11, reheap is O(h_i), where h_i is the height of the subtree rooted at index i. In the worst case, the heap would be a full tree of height h. Each node in level $l < h$ is the root of a subtree whose height is $h - l + 1$. Moreover, level l contains 2^{l-1} nodes. Thus, the sum of the heights of the subtrees rooted at level l of a full heap is $(h - l + 1) \times 2^{l-1}$.

Since the loop in the previous segment ignores the nodes in the last level—level h—of the heap, its complexity is

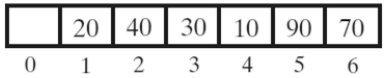
$$O\left(\sum_{l=1}^{h-1} (h - l + 1) \cdot 2^{l-1}\right)$$

Exercise 6 at the end of this chapter asks you to show that this expression is equivalent to O(2^h), which is O(n).

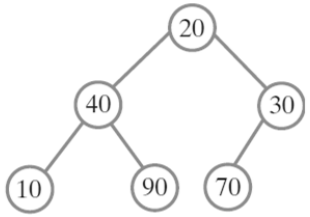
Creating a Heap

- A “Smart” way: create a heap uses the method reheap

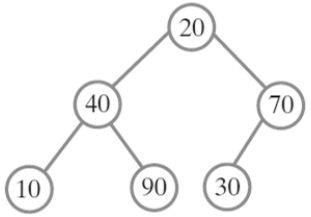
(a) An array of entries



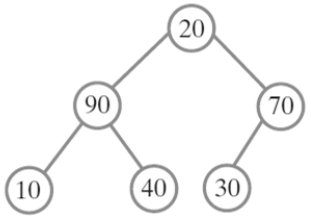
(b) The complete tree that the array represents



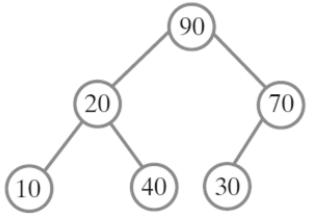
(c) After `reheap(3)`



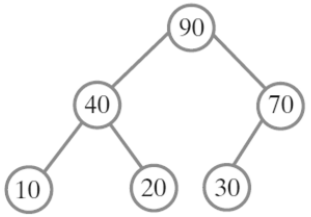
(d) After `reheap(2)`



(e) During `reheap(1)`



(f) After `reheap(1)`



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.

```
public MaxHeap(T[] entries)
{
   this(entries.length); // Call other constructor
   assert initialized = true;

   // Copy given array to data field
   for (int index = 0; index < entries.length; index++)
      heap[index + 1] = entries[index];

   // Create heap
   for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)
      reheap(rootIndex);
} // end constructor
```

In-Class Exercises

- Creating a max-heap with 27, 35, 23, 22, 4, 45, 21, 5, 42 and 19.

In-Class Exercises

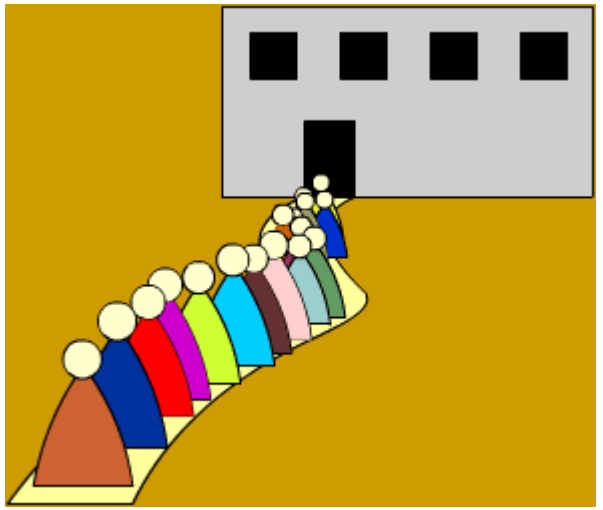
- Start with an empty heap and enter 10 elements with priories 1 through 10. Draw the resulting max-heap. (Note: A higher priority is a larger number)

- Remove three elements from the heap you created in the above heap. Draw the resulting heap.

Priority Queues

- A queue
  - 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.
  - When elements leave a priority queue, the highest priority element always leaves first.

- A heap is the most efficient implementation of priority queues.

https://www.cpp.edu/~ftang/courses/CS241/notes/heap.htm

Implementation of Priority Queues

- Using 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.

https://www.cpp.edu/~ftang/courses/CS241/notes/heap.htm

Implementation of Priority Queues

- Using ordinary queue,
  - Define an array of ordinary queues, called queues[].
  - The items with priority 0 are stored in queues[0]. Items with priority 1 are stored in queues[1]. And so on, up to queues[highest].
  - When an item with priority *i* needs to be added, we insert it to the end of queues[*i*].
  - When an item needs to be removed, we move down through the ordinary queues, starting with the highest priority, until we find a nonempty queue. We then remove the front item from this nonempty queue. For efficiency, we could keep a variable to remember the current highest priority.

https://www.cpp.edu/~ftang/courses/CS241/notes/heap.htm

Summary

- Heap
  - Definition
  - Operations in Heap
  - Implementations
  - Efficiency of operations

What I Want You to Do

- Review Class Slides
- Review Chapter 27