CMSC 341
Lecture 14: Priority Queues, Heaps

Prof. John Park

Based on slides from previous iterations of this course

Today's Topics

- Priority Queues
  - Abstract Data Type
- Implementations of Priority Queues:
  - Lists
  - BSTs
  - Heaps
- Heaps
  - Properties
  - Insertion
  - Deletion

Priority Queues and Heaps

Priority Queue ADT

- A priority queue stores a collection of entries

- Typically, an entry is a pair
  `(key, value)`
  where the key indicates the priority
  - Smaller value, higher priority

- Keys in a priority queue can be arbitrary objects on which an order is defined

Priority Queue vs Queue

- Priority queue is a specific type of queue

- Queues are FIFO
  - The element in the queue for the longest time is the first one we take out
- Priority queues: most important, first out
  - The element in the priority queue with the highest priority is the first one we take out
  - Examples: emergency rooms, airline boarding

Implementing Priority Queues

- Priority queues are an Abstract Data Type
  - They are a concept, and hence there are many different ways to implement them
- Possible implementations include
  - A sorted list
  - An ordinary BST
  - A balanced BST
- Run time will vary based on implementation

Implementing a Priority Queue

Priority Queue: Unsorted List

- We can implement a priority queue with a simple unsorted list (array, vector, etc.)
- <u>Insertion</u> just adds element to end of list
  - Enqueuing new element takes O(1) time
  - However, to find the highest priority, must find MIN(entire list), which takes O(n) time

Priority Queue: Sorted List

- We can implement a priority queue with a sorted list (array, vector, etc.)
- Sorted by priority upon <u>insertion</u>
  - To find the highest priority, simply take the first element, in O(1) time

    ```
    findMin() --> list.front()
    ```
  - Insertion can take O(n) time, however

Priority Queue: BST

- A BST makes a bit more sense than a list

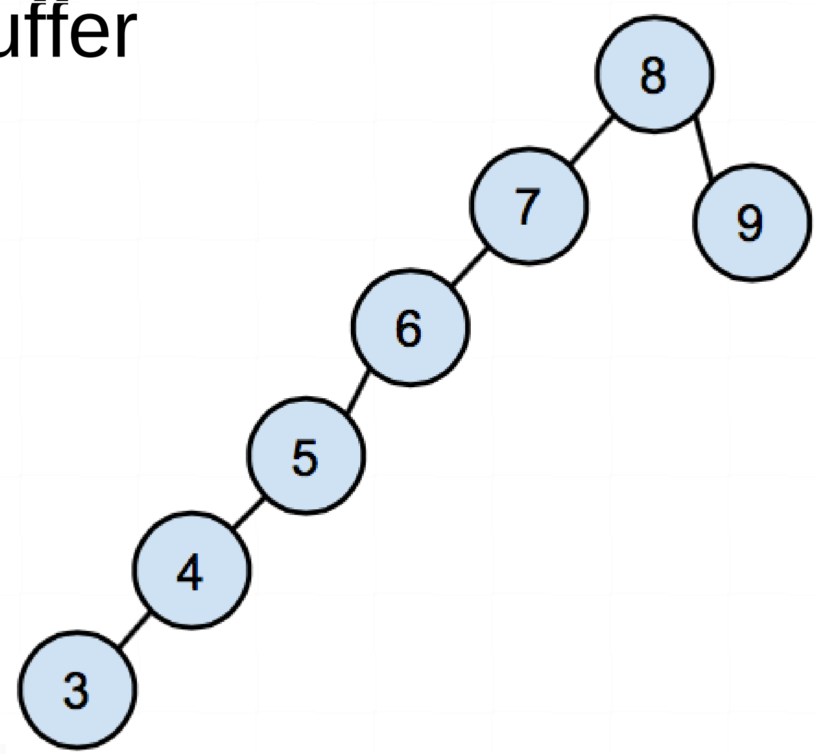
- Sorted like a regular BST upon insertion
  - To find the minimum, just go to the left

    call **`findMin()`**
    - And removal will be easy, because it will always be a leaf node!
  - Insertion should take no more than O(log n) time

    call **`Insert()`**

Priority Queue: BST Downsides

- Unfortunately, a BST Priority Queue can become unbalanced very easily, and the actual run time will suffer
  - If we have a low priority (high value) instance as our root, nearly everything will be to its left
- **`findMin()`** is now O(n) time ☹



Priority Queue: Heap

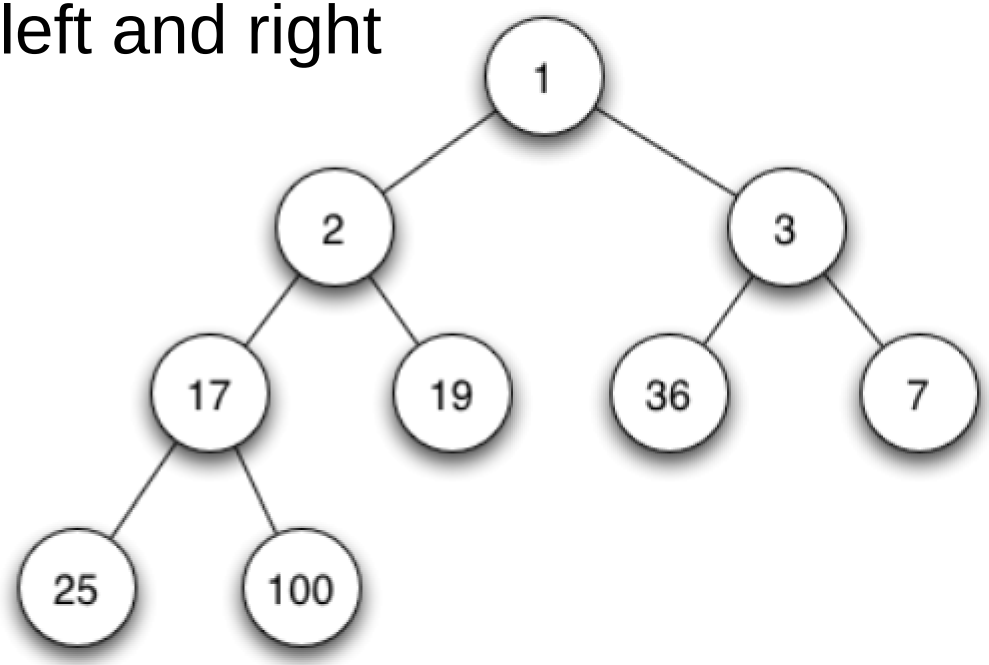
- The most common way to implement a priority queue is using a heap
- A heap is a binary tree (<u>not</u> a BST!!!) that satisfies the "heap condition":
  - Nodes in the tree are sorted based in relation to their parent's value, such that if A is a parent node of B, then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap
- Additionally, the tree must be <u>complete</u>

Heaps

Min Binary Heap

- A **min binary heap** is a…
  - Complete binary tree
  - Neither child is smaller than the value in the parent
  - No order between left and right

- In other words, smaller items go above larger ones



Min Binary Heap

- This property is called a **partial ordering**
  - There is no set relation between siblings, cousins, etc. – only that the values grow as we increase our distance from the root

- As a result of this partial ordering, every path from the root to a leaf visits nodes in a non-decreasing order

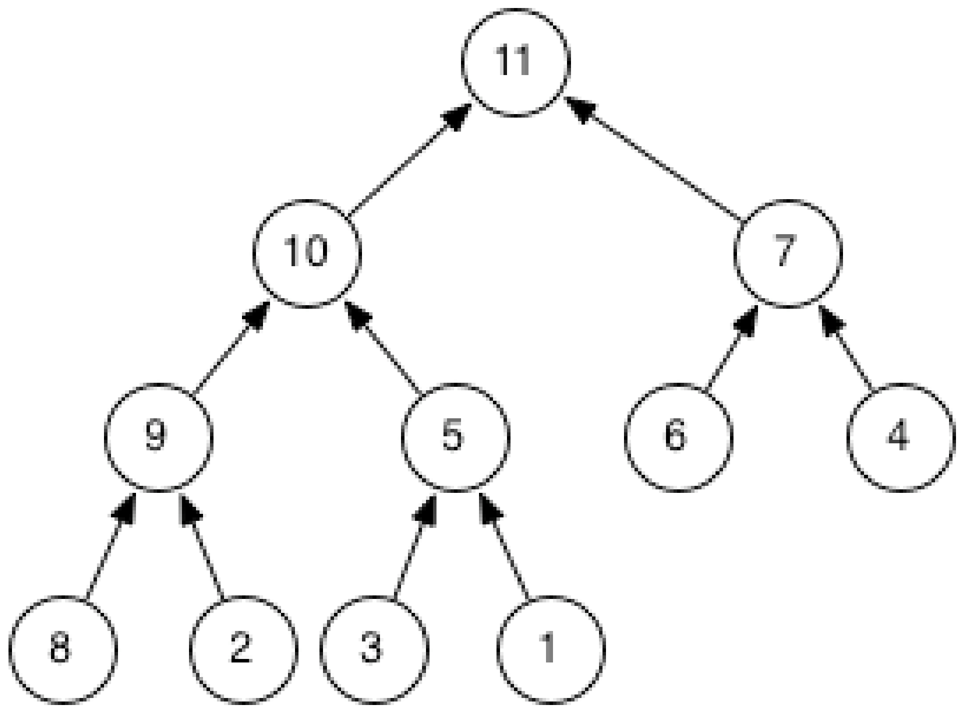
Min Binary Heap Performance

- Performance
  - (n is the number of elements in the heap)

- construction O(n)
- **`findMin()`** O(1)
- **`insert()`** O(lg n)
- **`deleteMin()`** O(lg n)

Convert a Heap to an Array

- Level-order traversal



11	10	7	9	5	6	4	8	2	3	1

Min Binary Heap Performance

- Heap efficiency results, in part, from the implementation
  - Conceptually a complete binary tree
  - But implemented by using an array/vector (in level order) with the root at index 1

To Find	Formula
Parent index	`floor((index)/2)`
Left Child index	`2(index)`
Right Child index	`2(index) + 1`

Min Binary Heap Performance

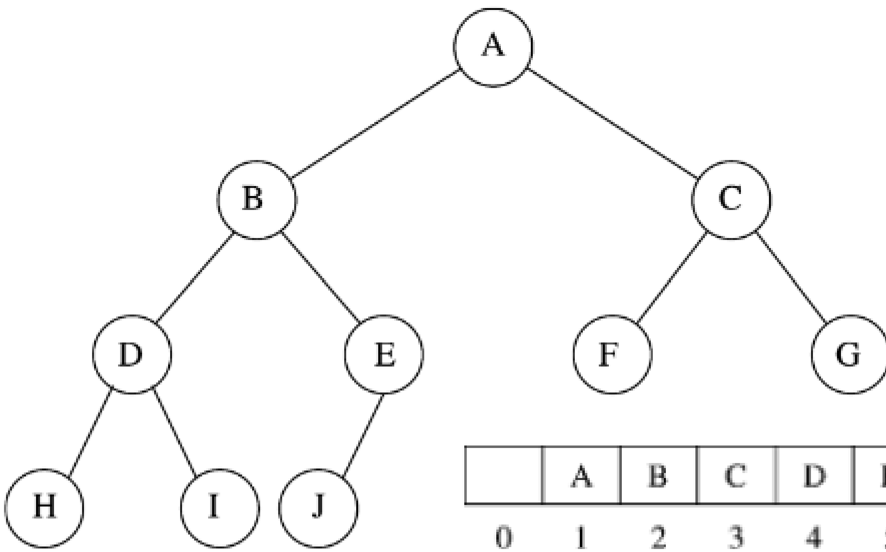
- For a node at index **`i`**
  - Its left child is at index **`2i`**
  - Its right child is at index **`2i+1`**
  - Its parent is at index $\lfloor$**`i/2`**$\rfloor$

- No pointer storage

- Fast computation of **`2i`** and $\lfloor$**`i/2`**$\rfloor$ by bit shifting
  - **`i << 1 = 2i`**
  - **`i >> 1 =`** $\lfloor$**`i/2`**$\rfloor$

Min Binary Heap: Exercises

- How to find the parent of E?
- The left child of D?
- The right child of A?



To Find	Formula
Parent index	`floor((index)/2)`
Left Child index	`2(index)`
Right Child index	`2(index) + 1`

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

Building a Heap

Insert Operation

- Must maintain
  - Heap shape:
    - Easy, just insert new element at “the end” of the array
  - Min heap order:
    1. Could be wrong after insertion if new element is smaller than its ancestors
    2. Continuously swap the new element with its parent until parent is not greater than it (“percolate up”)
- Performance of insert is **`O(log n)`** in the worst case because the height of a complete binary tree (CBT) is at most **`log n`**

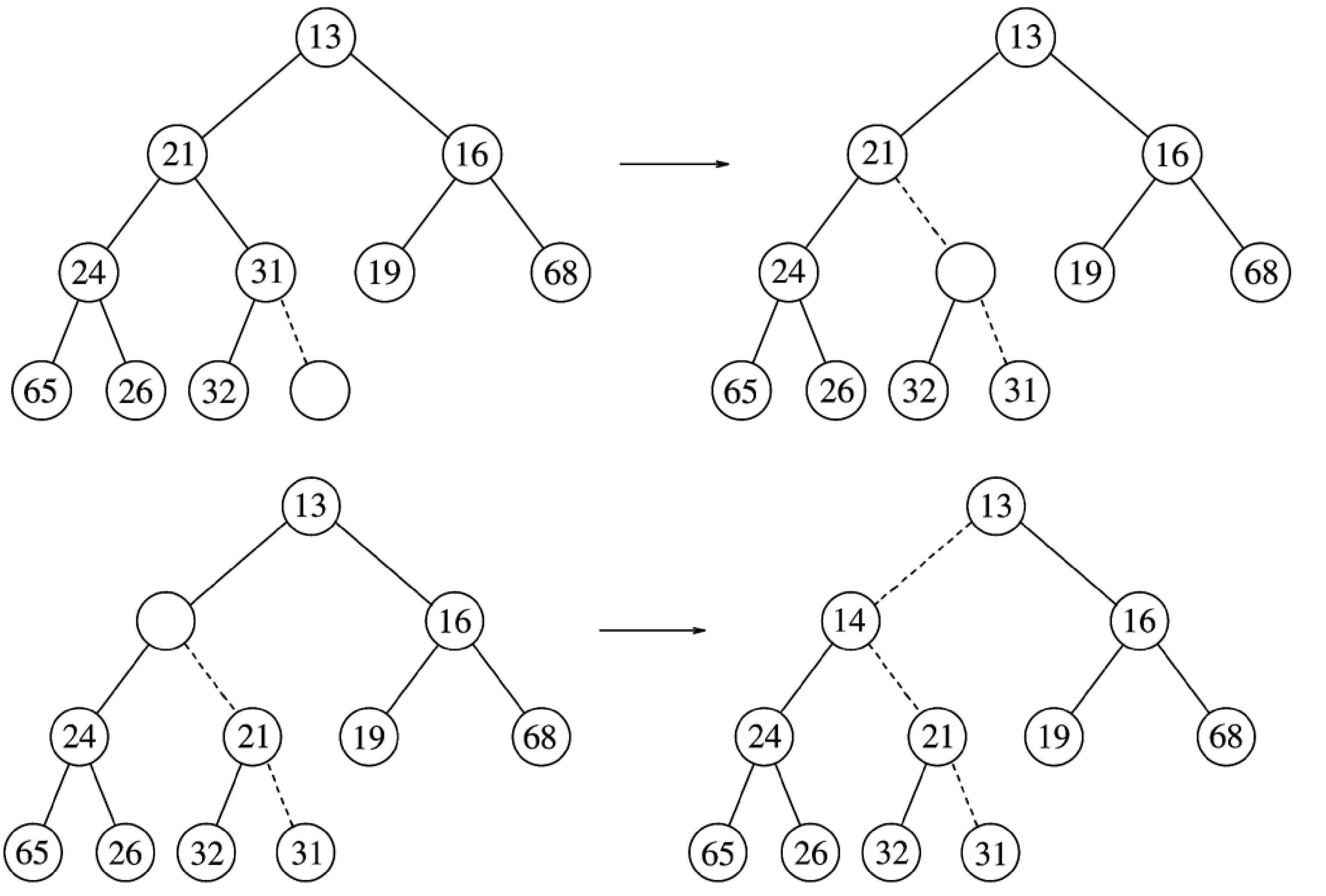
Insert Code

```
void insert(const Comparable &x) {
  /* First, check we are not overflowing array
     (code not included here) */

  // percolate up
  Comparable tmp;
  int hole = ++currentSize;
  array[hole] = x;
  for( ; hole > 1 && x < array[hole/2]; hole /= 2) {
     // swap, from child to parent
     tmp = array[hole];
     array[hole] = array[hole / 2];
     array[hole / 2] = tmp;
  }
}
```

Insert Code (v2)

```
/* More efficient version, where instead of swapping
   pairs, we just shift values down until right spot
 */
void insert(const Comparable &x) {
  /* First check we are not overflowing array
     (code not included here) */

  // percolate up
  int hole = ++currentSize;
  for( ; hole > 1 && x < array[hole/2]; hole /= 2) {
      // swap, from child to parent
     array[hole] = array[hole / 2];
  }
  array[hole] = x;
}
```

Insert Example: 14



Delete Operation

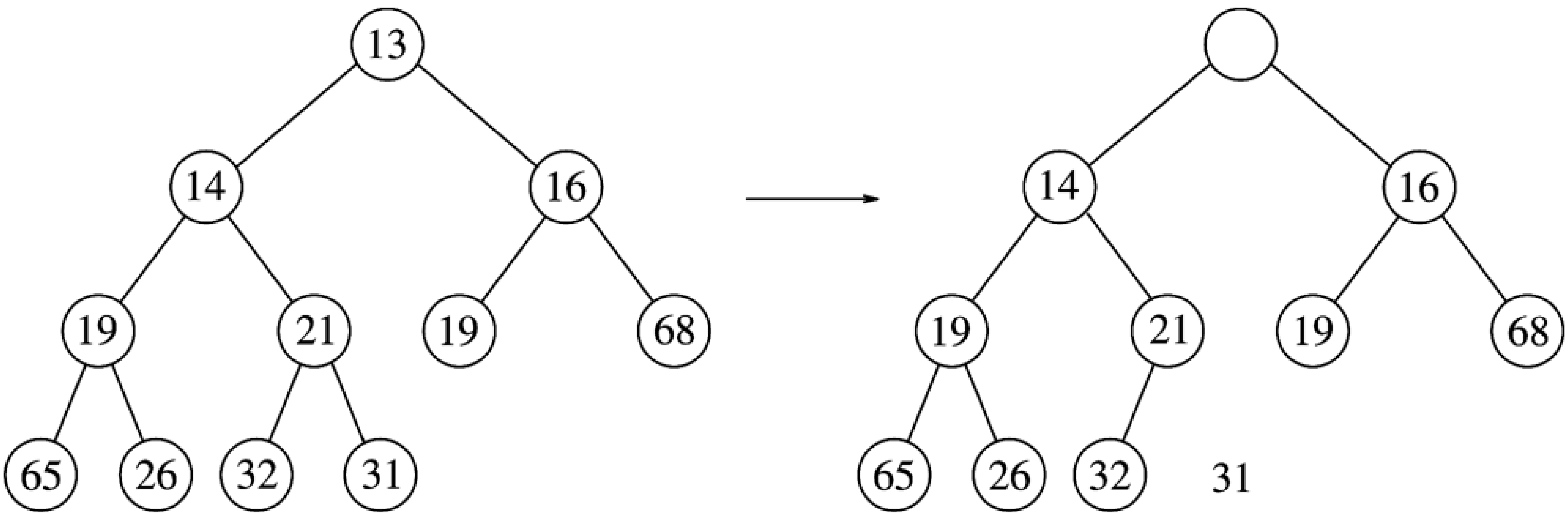
- Steps
  - Remove min element (the root)
  - Maintain heap shape
  - Maintain min heap order

- To maintain heap shape, actual node removed is “last one” in the array
  - Replace root value with value from last node and delete last node
  - Sift-down the new root value
    - Continually exchange value with the smaller child until no child is smaller.

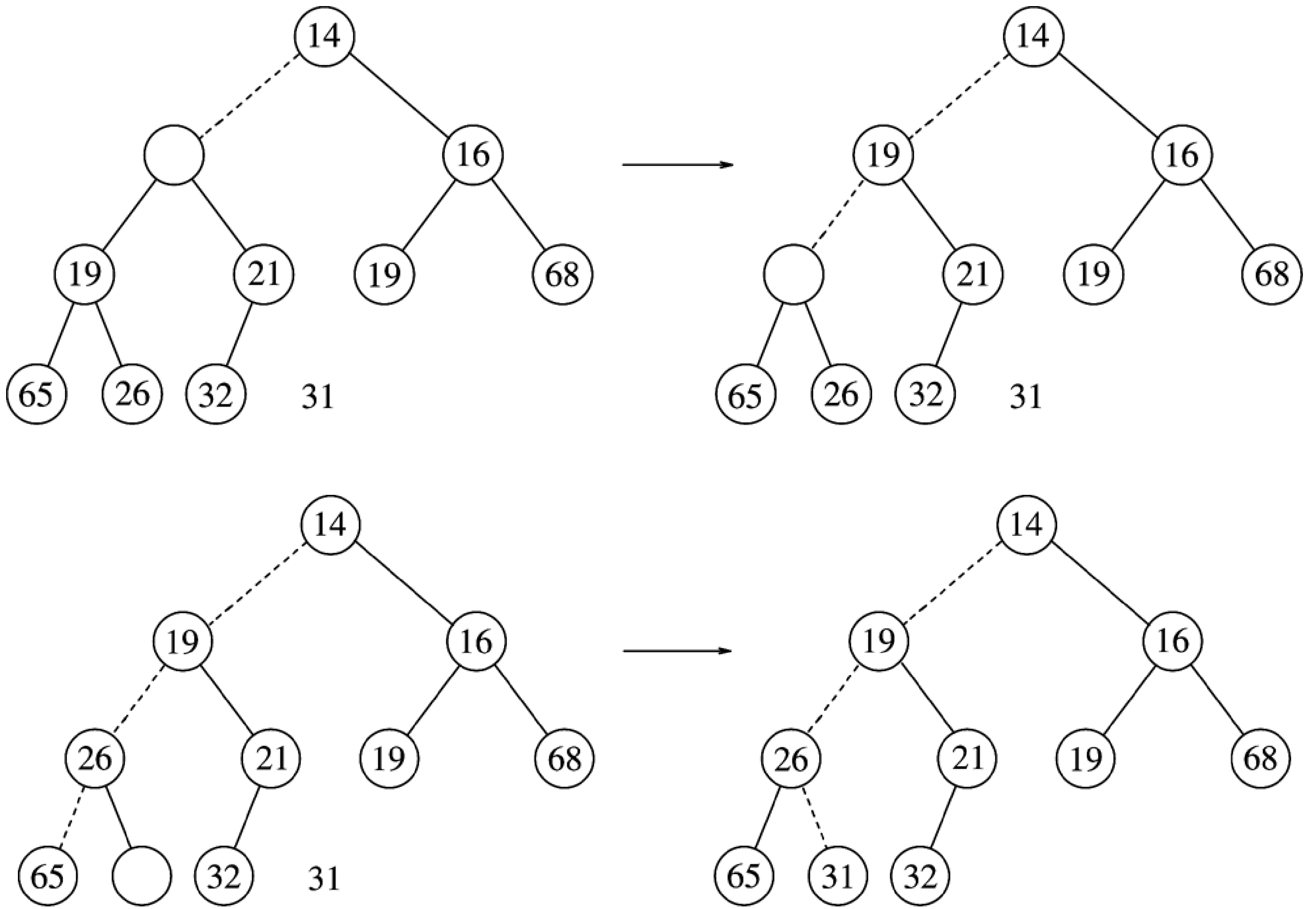
Delete Code

```
void deleteMin() {
  /* First, check for empty queue (code not included here) */

  int hole, child;
  Comparable tmp = array[currentSize--];

  for (hole = 1, child = 2; child <= currentSize;
      hole = child, child *= 2) {
    /* find smaller of siblings (if there is one) */
    if (child < currentSize && array[child+1] < array[child])
      child++;
    if (array[child] < tmp)
      array[hole] = array[child];
    else
      break;
  }
  array[hole] = tmp;
}
```

Example: Delete Min



Example: Delete Min



Visualization

- This visualization of a minimum heap may be helpful in your understanding of the different properties of a heap, as well as the exact steps taken for the operations of insertion, deletion, etc.

- http://www.cs.usfca.edu/~galles/JavascriptVisual/Heap.html