CSE 373
Data Structures and Algorithms

Lecture 14: Priority Queues (Heaps) II

Code for add method

```
public void add(int value) {
    // grow array if needed
    if (size >= array.length - 1) {
        array = resize();
    }

    // place element into heap at bottom
    size++;
    int index = size;
    array[index] = value;

    bubbleUp();
}
```

The bubbleUp helper

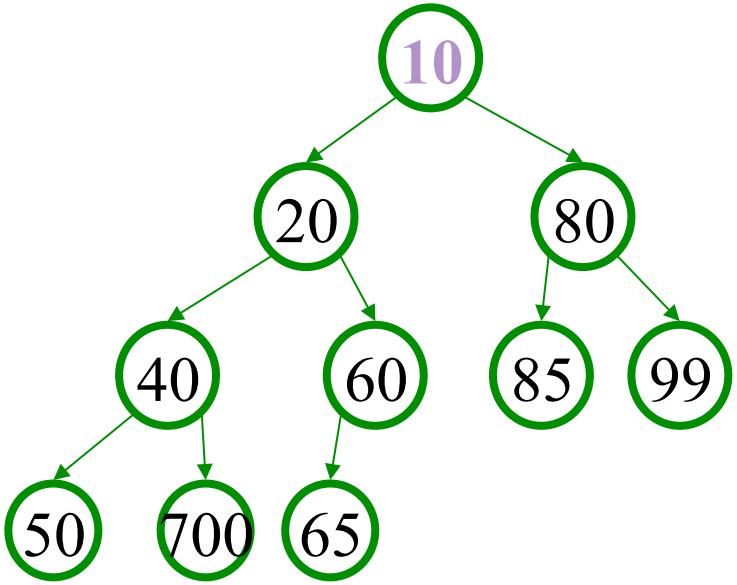
```
private void bubbleUp() {
    int index = size;

    while (hasParent(index)
           && (parent(index) > array[index])) {
        // parent/child are out of order; swap them
        swap(index, parentIndex(index));
        index = parentIndex(index);
    }
}

// helpers
private boolean hasParent(int i) { return i > 1; }
private int parentIndex(int i)   { return i / 2; }
private int parent(int i)  { return array[parentIndex(i)]; }
```

The peek operation

- peek on a min-heap is trivial; because of the heap properties, the minimum element is always the root
  - peek is O(1)
  - peek on a max-heap would be O(1) as well, but would return you the maximum element



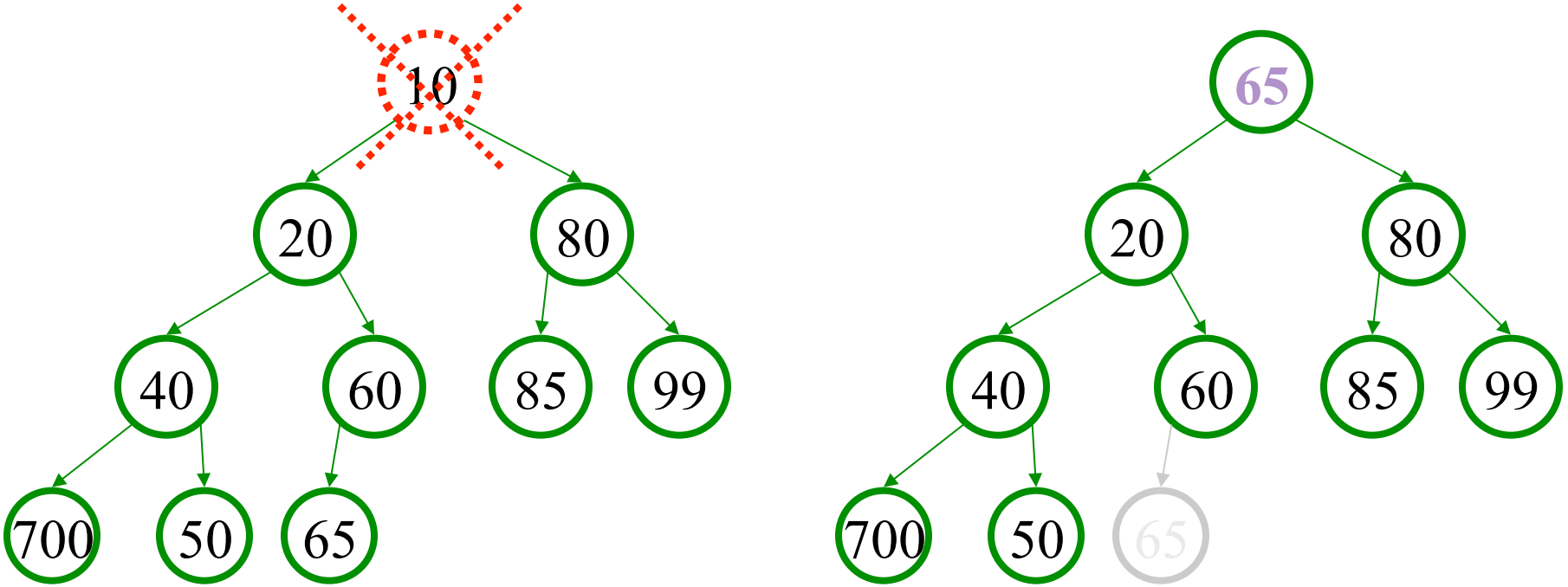
Code for peek method

```
public int peek() {
    if (isEmpty()) {
        throw new NoSuchElementException();
    }

    return array[1];
}
```

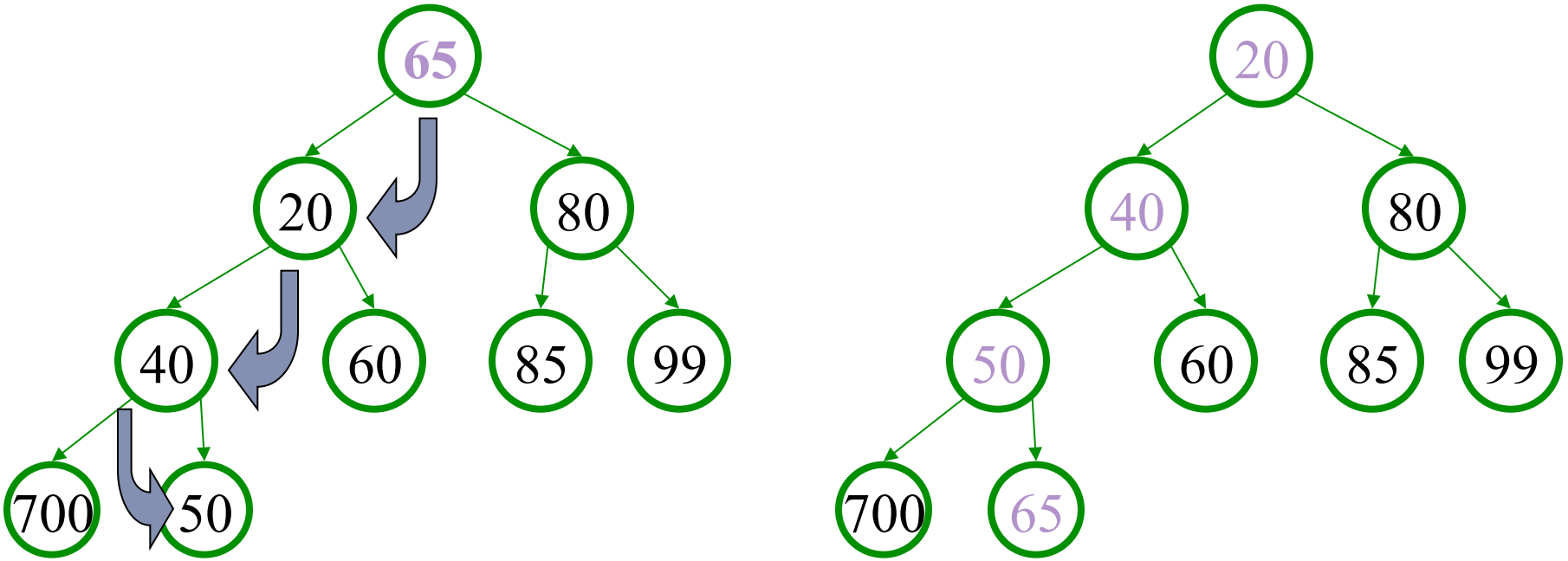
Removing from a min-heap

- min-heaps support `remove` of the min element
  - must remove the root while maintaining heap properties
  - intuitively, the last leaf must disappear to keep it a heap
  - initially, just swap root with last leaf (we'll fix it)



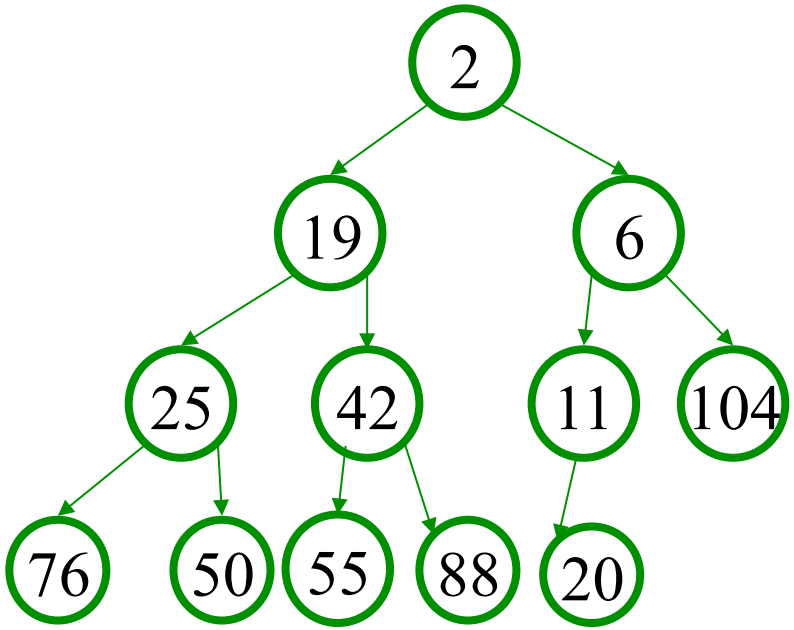
Removing from heap, cont'd.

- must fix heap-ordering property; root is out of order
  - shift the root downward ("bubble down") until it's in place
  - swap it with its smaller child each time
    - What happens if we don't always swap with the smaller child?



Heap practice problem

- Show the state of the following heap after `remove` has been executed on it 3 times, and state which elements are returned by the removal.



Code for remove method

```
public int remove() {
    int result = peek();

    // move last element of array up to root
    array[1] = array[size];
    size--;

    bubbleDown();

    return result;
}
```

The bubbleDown helper

```
private void bubbleDown() {
    int index = 1;
    while (hasLeftChild(index)) {
        int childIndex = leftIndex(index);

        if (hasRightChild(index)
            && (array[rightIndex(index)] < array[leftIndex(index)])) {
            childIndex = rightIndex(index);
        }

        if (array[childIndex] < array[index]) {
            swap(childIndex, index);
            index = childIndex;
        } else {
            break;
        }
    }
}

// helpers
private int leftIndex(int i)  { return i * 2; }
private int rightIndex(int i) { return i * 2 + 1; }
private boolean hasLeftChild(int i)  { return leftIndex(i) <= size; }
private boolean hasRightChild(int i) { return rightIndex(i) <= size; }
```

Advantages of array heap

- the "implicit representation" of a heap in an array makes several operations very fast
  - add a new node at the end (O(1))
  - from a node, find its parent (O(1))
  - swap parent and child (O(1))
  - a lot of dynamic memory allocation of tree nodes is avoided
  - the algorithms shown have elegant solutions

Generic Collection Implementation

PrintJob Class

```
public class PrintJob {
    private String user;
    private int number;
    private int priority;

    public PrintJob(int number, String user, int priority) {
        this.number = number;
        this.user = user;
        this.priority = priority;
    }

    public String toString() {
        return this.number + " (" + user + "):" + this.priority;
    }
}
```

Type Parameters (Generics)

- Recall: When constructing an ArrayList, you specify the type of elements it will contain between < and >.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Kona");
names.add("Daisy");
```

- We say that the ArrayList class accepts a type parameter, or that it is a generic class.

```
ArrayList<Type> name = new ArrayList<Type>();
```

Implementing generics

```
// a parameterized (generic) class
public class name<Type> {
    ...
}
```

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter.
  - The rest of your class's code can refer to that type by name.

- Exercise: Convert our priority queue classes to use generics.