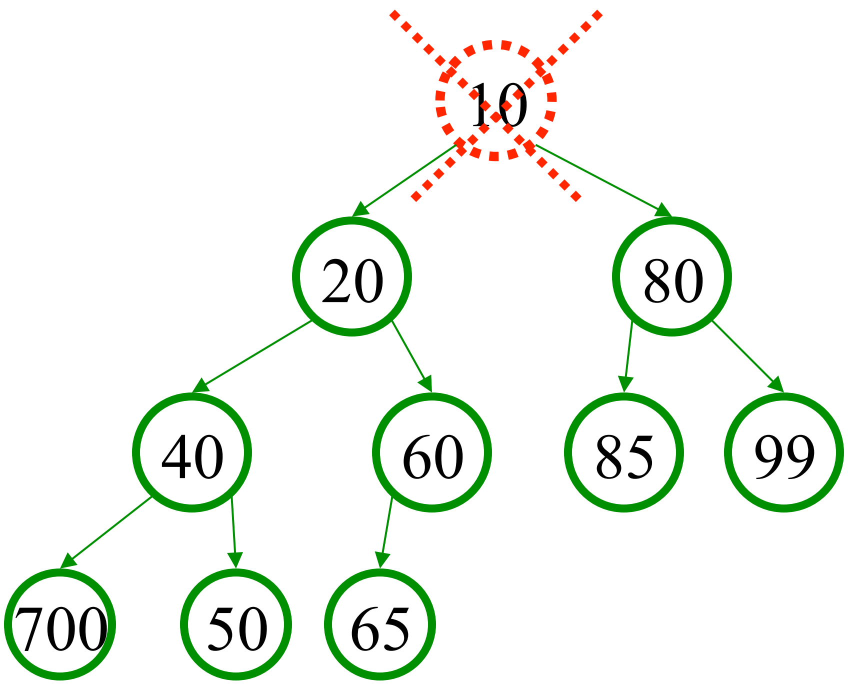
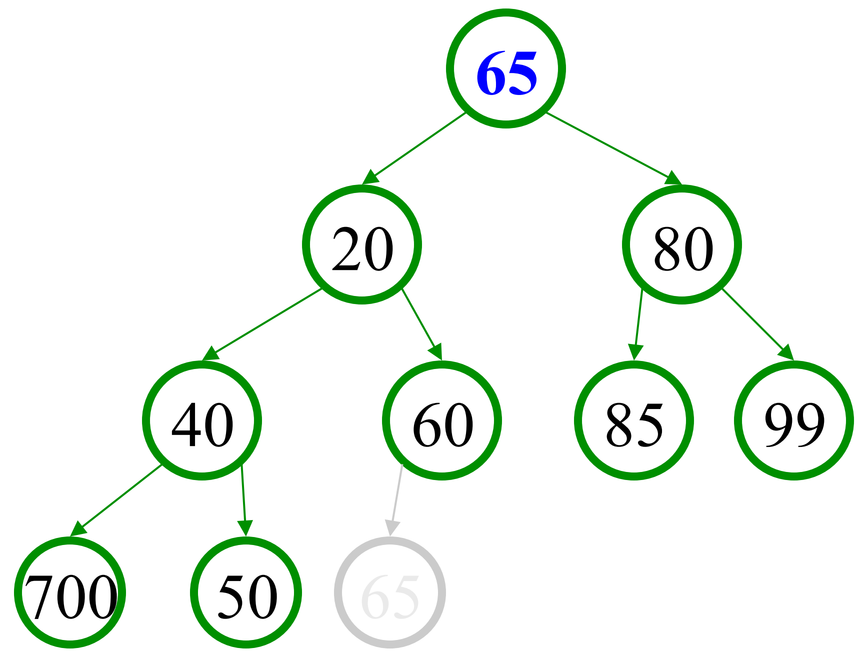
CSE 373: Data Structures and Algorithms

Lecture 12: Priority Queues (Heaps) II

Removing from a min-heap

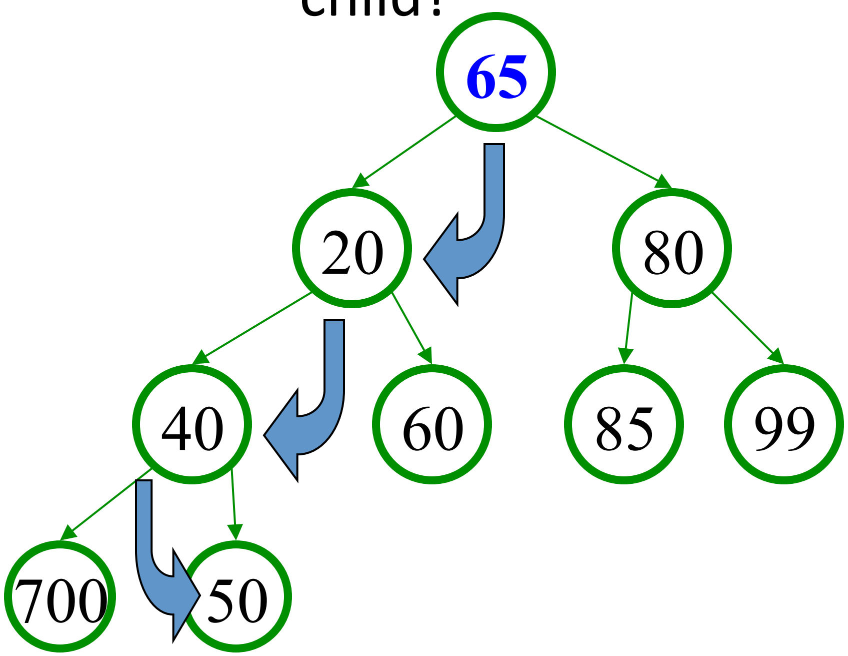
- min-heaps support `remove` of the min element (the root)
  - must remove the root while maintaining heap completeness and ordering properties
  - intuitively, the last leaf must disappear to keep it a heap
  - initially, just swap root with last leaf (we'll fix it)

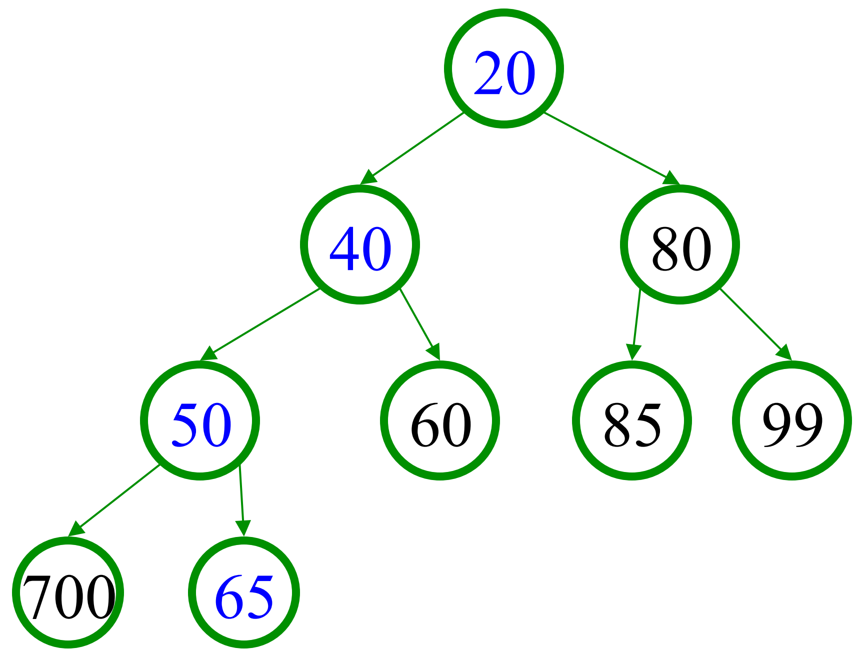




Removing from heap, cont'd.

- 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?

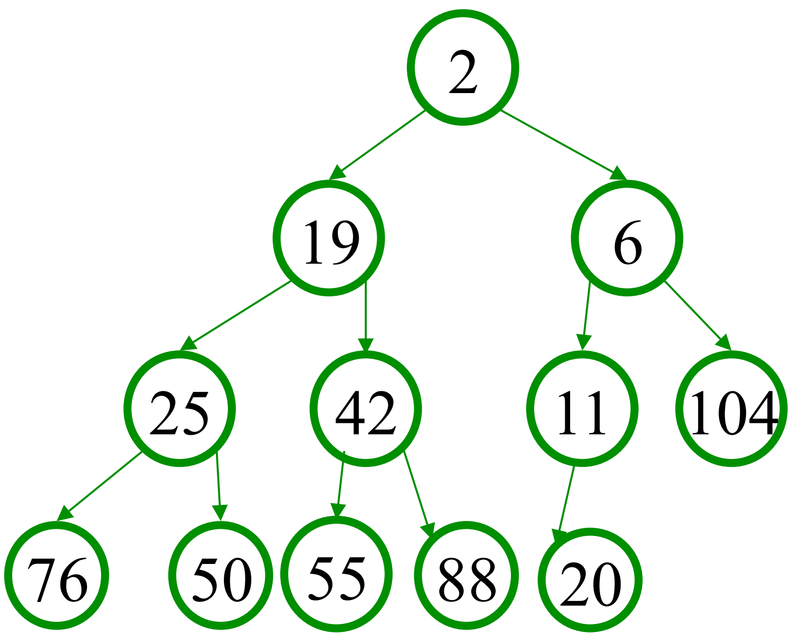




Heap practice problem

- The heap below is the min-heap built in the last heap practice problem.

- Now, show the state of the 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 = this.peek();

    // move last element of array up to root
    array[1] = array[size];
    array[size] = 0;
    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 usually 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)

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

- Recall: When constructing a `java.util.ArrayList`, you specify the type of elements it will contain between < and >.
  - We say that the `ArrayList` class accepts a **type parameter**, or that it is a **generic** class.

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

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.

Generics and arrays

```
public class Foo<T> {
    private T myField;                    // ok

    public void method1(T param) {
        myField = new T();                // error
        T[] a = new T[10];                // error
    }
}
```

– You cannot create objects or arrays of a parameterized type.

Generics/arrays, fixed

```
public class Foo<T> {
    private T myField;                      // ok

    public void method1(T param) {
        myField = param;                    // ok
        T[] a2 = (T[]) (new Object[10]);   // ok
    }
}
```

– But you can create variables of that type, accept them as parameters, return them, or create arrays by casting `Object[]`.