Lecture 15: Priority Queues (Heaps) III
Generics and arrays

```java
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.
- Why not?
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 declare variables of that type, accept them as parameters, return them, or create arrays by casting Object[].
The **compareTo** method

- The standard way for a Java class to define a comparison function for its objects is to define a **compareTo** method.
  - Example: in the String class, there is a method:
    
    ```java
    public int compareTo(String other)
    ```

- A call of `A.compareTo(B)` will return:
  - a value `< 0` if A comes "before" B
  - a value `> 0` if A comes "after" B
  - or `0` if A and B are "equal"
Comparable

public interface Comparable<E> {
    public int compareTo(E other);
}

A class can implement the Comparable interface to define a natural ordering function for its objects.

A call to the compareTo method should return:

- a value < 0 if the other object comes "before" this one
- a value > 0 if the other object comes "after" this one
- 0 if the other object is considered "equal" to this one
Comparable template

```java
public class name implements Comparable<name> {
    ...

    public int compareTo(name other) {
        ...
    }
}
```

Exercise: Add a `compareTo` method to the `PrintJob` class such that `PrintJobs` are ordered according to their priority (ascending – lower priorities are more important than higher ones).
Comparable example

```java
public class PrintJob implements Comparable<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 int compareTo(PrintJob otherJob) {
        return priority - otherJob.priority;
    }

    public String toString() {
        return number + " (" + user + ")":" + priority;
    }
}
```
$d$-Heaps
Generalization: d-Heaps

- Each node has $d$ children
- Still can be represented by array
- Good choices for $d$ are a power of 2
  - Only because multiplying and dividing by powers of 2 is fast on a computer
- How does height compare to binary heap?
Operations on $d$-Heap

- **insert**: runtime = depth of tree decreases, $\Theta(\log_d n)$
- **remove**: runtime = bubbleDown requires more comparisons to find min, $\Theta(d \log_d n)$

Does this help insert or remove more?
Other Priority Queue Operations
More Min-Heap Operations

- **decreasePriority**: reduce the priority value of an element in the queue
  
  Solution: change priority and ________________________

- **increasePriority**: increase the priority value of an element in the queue
  
  Solution: change priority and _________________________

- How do we find the element in the queue? What about duplicates?
  
  Need a reference to the element!
More Min-Heap Operations

- **remove**: given a reference to an object in the queue, remove the object from the queue
  
  Solution: set priority to negative infinity, percolate up to root and **deleteMin**

- **findMax**: Can look at all leaves, but not really the point of a min-heap!
Building a Heap

- Given a list of numbers, how would you build a heap?
- At every point, the new item may need to percolate all the way through the heap
- Adding the items one at a time is $\Theta(n \log n)$ in the worst case
- A more sophisticated algorithm does it in $\Theta(n)$
$O(N)$ buildHeap

- First, add all elements arbitrarily maintaining the completeness property
- Then fix the heap order property by performing a "bubble down" operation on every node that is not a leaf, starting from the rightmost internal node and working back to the root
Each element in the list \([12, 5, 11, 3, 10, 6, 9, 4, 8, 1, 7, 2]\) has been inserted into a heap such that the completeness property has been maintained.

Now, fix the heap's order property by "bubbling down" every internal node.
Final State of the Heap
Different Heaps

Successive inserts $\Theta(n \log n)$:

But it doesn't matter because they are both heaps.

buildHeap $\Theta(n)$:
Heap Sort
Heap sort

- **heap sort**: an algorithm to sort an array of N elements by turning the array into a heap, then doing a `remove` N times.
  - The elements will come out in sorted order!

- What is the runtime?

- This algorithm is not very space-efficient. Why not?
Improved heap sort

- The heap sort shown requires a second array
- We can use a max-heap to implement an improved version of heap sort that needs no extra storage
  - Useful on low-memory devices
  - Still only $O(n \log n)$ runtime
  - Elegant
Improved heap sort 1

- Use an array heap, but with 0 as the root index
- max-heap state after buildHeap operation:

```
[97, 53, 59, 26, 41, 58, 31, 16, 21, 36, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```
Improved heap sort 2

- State after one **remove** operation:
  - Modified **remove** that moves element to end

![Heap Diagram]

```
<table>
<thead>
<tr>
<th>59</th>
<th>53</th>
<th>58</th>
<th>26</th>
<th>41</th>
<th>36</th>
<th>31</th>
<th>16</th>
<th>21</th>
<th>97</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Improved heap sort 3

- **State after two remove operations:**
  - Notice that the largest elements are at the end (becoming sorted!)
According to Knuth, the average growth rate of Insertion sort is about 0.9 times that of Selection sort and about 0.4 times that of Bubble Sort. The average growth rate of Quicksort is about 0.74 times that of Mergesort and about 0.5 times that of Heapsort.

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Average case (^{(†)})</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort</td>
<td>(n)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Selection sort</td>
<td>(n^2)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>(n)</td>
<td>(n^2)</td>
<td>(n^2)</td>
</tr>
<tr>
<td>Mergesort</td>
<td>(n \log_2 n)</td>
<td>(n \log_2 n)</td>
<td>(n \log_2 n)</td>
</tr>
<tr>
<td>Heapsort</td>
<td>(n \log_2 n)</td>
<td>(n \log_2 n)</td>
<td>(n \log_2 n)</td>
</tr>
<tr>
<td>Quicksort</td>
<td>(n \log_2 n)</td>
<td>(n \log_2 n)</td>
<td>(n^2)</td>
</tr>
</tbody>
</table>