CSE 373: Data Structures and Algorithms

Lecture 13: Priority Queues (Heaps) III
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 in the ordering,
  - a value > 0 if A comes "after" B in the ordering,
  - 0 if A and B are considered "equal" in the ordering.
Comparable

```java
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, or
  0 if the `other` object is considered "equal" to this.
Comparable template

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 this.priority - otherJob.priority;
    }

    public String toString() {
        return this.number + " (" + user + "):" + this.priority;
    }
}
```
Other Priority Queue Operations
More Min-Heap Operations

• `decreasePriority`
  – given a reference of an element in the queue, reduce its priority value

  Solution: change priority and _________________

• `increasePriority`
  – given a reference of an element in the queue, increase its priority value

  Solution: change priority and _________________

Why do we need a `reference`? Why not simply data value?
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**
Building 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 (what is the average case?)

• A more sophisticated algorithm does it in $\Theta(n)$
O(N) buildHeap

• Algorithm idea
  – 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
    o why does this buildHeap operation work?
buildHeap practice problem

• 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, starting from the rightmost internal node working back to the root.
Final State of the Heap
Different Heaps

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

buildHeap $\Theta(n)$:

But it doesn't matter because they are both heaps.
$d$-Heaps
Generalization: \(d\)-Heaps

- Each node has \(d\) children
- Still can be represented by array
- Good choices for \(d\) are a power of 2
  - How does height compare to binary heap?
Operations on $d$-Heap

• insert: runtime =

• remove: runtime =

Does this help insert or remove more?
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!
  - we can put them into a new sorted array
A max-heap

- the heaps shown have been minimum heaps because the elements come out in ascending order
- a *max-heap* is the same thing, but with the elements in descending order
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
  – $O(n \log n)$ runtime
  – no external storage required!
  – useful on low-memory devices
  – elegant
Improved heap sort 1

• use an array heap, but with 0 as the root index
• max-heap state after `buildHeap` operation:
Improved heap sort 2

- state after one `remove` operation:
  - modified `remove` that moves element to end

```
  59
 /   \
53     58
|     |
26 41 36 31
|   |
16 21 97
```

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

• state after two `remove` operations:
  – notice that the largest elements are at the end (becoming sorted!)
## Sorting algorithms review

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Average case (†)</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble sort</td>
<td>$n$</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>Treesort</td>
<td>$n \log_2 n$</td>
<td>$n \log_2 n$</td>
<td>$n^2$</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>

† 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. Also, the **average growth rate** of Quicksort is about 0.74 times that of Mergesort and about 0.5 times that of Heapsort.