CSE 373

Advanced heap implementation; ordering/Comparator
read: Weiss Ch. 6

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Int PQ ADT interface

• Let's write our own implementation of a priority queue.
  ▪ To simplify the problem, we only store ints in our set for now.
  ▪ As is (usually) done in the Java Collection Framework, we will define sets as an ADT by creating a Set interface.
  ▪ Core operations are: add, peek (at min), remove (min).

```java
public interface IntPriorityQueue {
    void add(int value);
    void clear();
    boolean isEmpty();
    int peek();         // return min element
    int remove();       // remove/return min element
    int size();
}
```
Generic PQ ADT

- Let's modify our priority queue so it can store any type of data.
  - As with past collections, we will use Java generics (a type parameter).

```java
public interface PriorityQueue<E> {
    void add(E value);
    void clear();
    boolean isEmpty();
    E peek(); // return min element
    E remove(); // remove/return min element
    int size();
}
```
Generic HeapPQ class

- We can modify our heap priority class to use generics as usual...

```java
public class HeapPriorityQueue<E>
    implements PriorityQueue<E> {  
    private E[] elements;
    private int size;

    // constructs a new empty priority queue
    public HeapPriorityQueue() {  
        elements = (E[]) new Object[10];
        size = 0;
    }

    ...
    }
```
Problem: ordering elements

// Adds the given value to this priority queue in order.
public void add(E value) {
    ...
    int index = size + 1;
    boolean found = false;
    while (!found && hasParent(index)) {
        int parent = parent(index);
        if (elements[index] < elements[parent]) { // error
            swap(elements, index, parent(index));
            index = parent(index);
        } else {
            found = true; // found proper location; stop
        }
    }
}

- Even changing the < to a compareTo call does not work.
  - Java cannot be sure that type E has a compareTo method.
Comparing objects

- Heaps rely on being able to order their elements.
- Operators like `<` and `>` do not work with objects in Java.
  - But we do think of some types as having an ordering (e.g. Dates).
  - (In other languages, we can enable `<, >` with operator overloading.)
- **natural ordering**: Rules governing the relative placement of all values of a given type.
  - Implies a notion of equality (like equals) but also `< and >`.
- **total ordering**: All elements can be arranged in \( A \leq B \leq C \leq \ldots \) order.
- The Comparable interface provides a natural ordering.
The Comparable interface

The standard way for a Java class to define a comparison function for its objects is to implement the `Comparable` interface.

```java
public interface Comparable<T> {
    public int compareTo(T other);
}
```

A call of `A.compareTo(B)` should return:
- a value < 0 if `A` comes "before" `B` in the ordering,
- a value > 0 if `A` comes "after" `B` in the ordering,
- or exactly 0 if `A` and `B` are considered "equal" in the ordering.

**Effective Java Tip #12**: Consider implementing `Comparable`.
Bounded type parameters

\[ <\text{Type} \text{ extends } \text{SuperType}> \]

- An upper bound; accepts the given supertype or any of its subtypes.
- Works for multiple superclass/interfaces with &:
  \[ <\text{Type} \text{ extends } \text{ClassA} \text{ & InterfaceB} \text{ & InterfaceC} \text{ & } ...> \]

\[ <\text{Type} \text{ super } \text{SuperType}> \]

- A lower bound; accepts the given supertype or any of its supertypes.

• Example:

```
// can be instantiated with any animal type
public class Nest<T extends Animal> {
    ...
}
...
Nest<Bluebird> nest = new Nest<Bluebird>();
```
Corrected HeapPQ class

```java
public class HeapPriorityQueue<E extends Comparable<E>> implements PriorityQueue<E> {
    private E[] elements;
    private int size;

    // constructs a new empty priority queue
    public HeapPriorityQueue() {
        elements = (E[]) new Object[10];
        size = 0;
    }

    public void add(E value) {
        ... while (...) {
           if (elements[index].compareTo(elements[parent]) < 0) {
                swap(...);
           }
        }
    }
}
```
Other heap operations

- Java collections support these methods in addition to the ones we listed. How would we implement them in our heap PQ?
  - (What would be each method's Big-Oh?)

  - public boolean contains(E element)
    - returns true if the priority queue contains the given value

  - public void remove(E element)
    - deletes an arbitrary element in the priority queue, if it is found

  - public String toString()
    - returns a string representation of the priority queue's elements
The contains operation

- Though there is ordering to the heap, it is not easy to take advantage of the ordering to optimize `contains`.
  - Why not?
  - What elements *must* be examined to see if the heap contains:
    - 11?
    - 19?
    - 31?
  - In practice we usually just loop over the heap array linearly.

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | ...
|-------|---|---|---|---|---|---|---|---|---|---|---|---|---|...
| value | 0 | 10| 15| 80| 40| 20| 85| 99| 50| 77| 65| 60| 0 | ...
| size  | 11|    |    |    |    |    |    |    |    |    |    |    |    |
Removing arbitrary element

• Similar to `contains`, removing an arbitrary element from a heap is not easy to optimize because you must first `find` the value.
  - Suppose the client wants to remove 40.
  - How can we remove it safely without disturbing the heap?

```plaintext
<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>10</td>
<td>15</td>
<td>80</td>
<td>40</td>
<td>20</td>
<td>85</td>
<td>99</td>
<td>50</td>
<td>77</td>
<td>65</td>
<td>60</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>size</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Implementing remove

queue.remove(40);

- **Step 1:** Pretend 40's value is $-\infty$ (very small)
  - Bubble 40 all the way up to the root.
- **Step 2:** Perform a remove-min on 40, which is currently the root.
  - Do it the same as usual:
    Swap up the rightmost leaf (60), then bubble that leaf down.
The `toString` operation

- A typical heap PQ implementation does "the simple thing" and produces a `toString` with the elements in the heap order.
  - e.g. `toString` on the heap shown would return 
    `"[10, 15, 80, 40, 20, 85, 99, 50, 77, 65, 60]"

- Why not output the elements in their sorted order?
  - Wouldn't that make more sense to the client?
Ordering and Comparators
public class Rectangle implements Comparable<Rectangle> {
    private int x, y, width, height;
    public int compareTo(Rectangle other) {
        // ...?
    }
}

• What is the "natural ordering" of rectangles?
  ▪ By x, breaking ties by y?
  ▪ By width, breaking ties by height?
  ▪ By area? By perimeter?

• Do rectangles have any "natural" ordering?
  ▪ Might we want to arrange rectangles into some order anyway?
Comparator interface

```java
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

• **Interface Comparator** is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering(s) for a type even if there is no obvious "natural" ordering for that type.
  - Allows you to externally define an ordering for a class that, for whatever reason, you are not able to modify to make it Comparable:
    - a class that is part of the Java class libraries
    - a class that is final and can't be extended
    - a class from another library or author, that you don't control
    - ...
Comparator examples

```java
public class RectangleAreaComparator implements Comparator<Rectangle> {
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}

public class RectangleXYComparator implements Comparator<Rectangle> {
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```
Using Comparators

• TreeSet, TreeMap, PriorityQueue can use Comparator:

   Comparator<Rectangle> comp = new RectangleAreaComparator();
   Set<Rectangle> set = new TreeSet<Rectangle>(comp);
   Queue<Rectangle> pq = new PriorityQueue<Rectangle>(10, comp);

• Searching and sorting methods can accept Comparators.

   Arrays.binarySearch(array, value, comparator)
   Arrays.sort(array, comparator)
   Collections.binarySearch(list, comparator)
   Collections.max(collection, comparator)
   Collections.min(collection, comparator)
   Collections.sort(list, comparator)

• Methods are provided to reverse a Comparator's ordering:

   public static Comparator Collections.reverseOrder()
   public static Comparator Collections.reverseOrder(comparator)
PQ and Comparator

• Our heap priority queue currently relies on the Comparable natural ordering of its elements:

```java
class HeapPriorityQueue<E extends Comparable<E>> implements PriorityQueue<E> {
    ...
    public HeapPriorityQueue() { ... }
}
```

• To allow other orderings, we can add a constructor that accepts a Comparator so clients can arrange elements in any order:

```java
public HeapPriorityQueue(Comparator<E> comp) { ... }
```
PQ Comparator exercise

- Write code that stores strings in a priority queue and reads them back out in ascending order *by length*.
  - If two strings are the same length, break the tie *by ABC order*.

```java
Queue<String> pq = new PriorityQueue<>(...);
pq.add("you");
pq.add("meet");
pq.add("madam");
pq.add("sir");
pq.add("hello");
pq.add("goodbye");
while (!pq.isEmpty()) {
    System.out.print(pq.remove() + " ");
}

// sir you meet hello madam goodbye
```
PQ Comparator answer

- Use the following comparator class to organize the strings:

```java
public class LengthComparator
    implements Comparator<String> {
    public int compare(String s1, String s2) {
        if (s1.length() != s2.length()) {
            // if lengths are unequal, compare by length
            return s1.length() - s2.length();
        } else {
            // break ties by ABC order
            return s1.compareTo(s2);
        }
    }

    }

    ...

    Queue<String> pq = new PriorityQueue<String>(100,
            new LengthComparator());
```
$d$-heaps; heap sort
Generalization: d-Heaps

- **d-heap**: one where each node has $d$ children ($d \geq 2$)
  - Can still be represented by an array.
  - How does its height compare to that of a binary ($d = 2$) heap?
  - Example, a 3-heap:
d-heap runtime

• What is the effect on runtime of using a \( d \)-heap?
  
  - add: \( O(\log_d N) \) - fewer parents to examine; faster.
  - peek: \( O(1) \)
  - remove: \( O(d \log_d N) \) - must look at all \( d \) children each time; slower.

• Adding is slightly faster; removing is slightly slower.
Heap sort

- **heap sort**: An algorithm to sort an array of *N* elements by turning the array into a heap, then calling `remove` *N* times.
  - The elements will come out in sorted order.
  - We can put them into a new sorted array.
  - What is the runtime?
public static void heapSort(int[] a) {
    PriorityQueue<Integer> pq =
        new HeapPriorityQueue<Integer>();
    for (int n : a) {
        pq.add(a);
    }
    for (int i = 0; i < a.length; i++) {
        a[i] = pq.remove();
    }
}

- This code is correct and runs in $O(N \log N)$ time but wastes memory.
- It makes an entire copy of the array $a$ into the internal heap of the priority queue.
- Can we perform a heap sort without making a copy of $a$?
Improving the code

• **Idea:** Treat `a` itself as a max-heap, whose data starts at 0 (not 1).
  - `a` is not actually in heap order.
  - But if you repeatedly "bubble down" each non-leaf node, starting from the last one, you will eventually have a proper heap.

• Now that `a` is a valid max-heap:
  - Call `remove` repeatedly until the heap is empty.
  - But make it so that when an element is "removed", it is moved to the end of the array instead of completely evicted from the array.
  - When you are done, voila! The array is sorted.
Step 1: Build heap in-place

- "Bubble" down non-leaf nodes until the array is a max-heap:
  - int[] a = {21, 66, 40, 10, 70, 81, 30, 22, 45, 95, 88, 38};
  - Swap each node with its larger child as needed.
Build heap in-place answer

- 30: nothing to do
- 81: nothing to do
- 70: swap with 95
- 10: swap with 45
- 40: swap with 81
- 66: swap with 95, then 88
- 21: swap with 95, then 88, then 70

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 0 |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| value | 95 | 88 | 81 | 45 | 70 | 40 | 30 | 22 | 10 | 66 | 21 | 38 | 0  |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| size  | 12 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Remove to sort

- Now that we have a max-heap, remove elements repeatedly until we have a sorted array.
  - Move each removed element to the end, rather than tossing it.
Remove to sort answer

- 95: move 38 up, swap with 88, 70, 66
- 88: move 21 up, swap with 81, 40
- 81: move 38 up, swap with 70, 66
- 70: move 10 up, swap with 66, 45, 22
- ...

(Notice that after 4 removes, the last 4 elements in the array are sorted. If we remove every element, the entire array will be sorted.)