

CS 10:
Problem solving via Object Oriented
Programming
Winter 2017

Tim Pierson
260 (255) Sudikoff

Day 13 – Prioritizing

Agenda



1. Priority queues
2. Java's priority queue
3. Reading from a file
4. ArrayList implementations

We can model airplanes landing as a queue

Airplanes queued to land



Each airplane assigned a priority to land in order of arrival

First in pattern, first to land

Sometimes higher priority issues arise and we need to change order

Airplanes queued to land



Suddenly one aircraft has an in flight emergency, and needs to land now!

Need a way to go to front of queue

Enter the priority queue

Priority queues add the ability to extract the highest priority item

Min Priority Queue Overview

- Lowest priority number are removed first (you are number 1 for landing)
- Can be used for sorting (put everything in, then extract lowest priority number, one at a time, until queue empty)
- Used extensively in simulations and scheduling
 - Minute 1, factory machine starts job, will end at current time + 10 mins (minute 11)
 - Minute 2, another job starts and will end at current time + 3 mins (minute 5)
 - Priority queue tells us job 2 will finish first, at minute 5, no need to check minute 2,3,4...

Priority queues have operations for adding and removing items

Min Priority Queue Operations

- *isEmpty()* – true if no items stores
- *insert()* – insert an element in priority queue
- *minimum()* – return element with smallest key, but leaves the element in priority queue
- *extractMin()* – return and remove element with smallest key
- *decreaseKey()* – reduce priority value of item so it is chosen more quickly

Interface is specified in MinPriorityQueue.java

MinPriorityQueue.java

- Interface for min priority queue
- Each element has value
 - Called key, but used to evaluate priority
 - Will be used to remove lowest key
- Element type must extend Comparable<E> so we can tell which key is lowest using compareTo() function (built in for Strings, Integers, etc, otherwise need to implement ourselves)
- Can also have max priority queue, just reverse the compareTo()

Agenda

1. Priority queues

 2. Java's priority queue

3. Reading from a file

4. ArrayList implementations

Java implements a PriorityQueue, but with non-standard names

Java's Min Priority Queue Operations

- *insert == add*
- *minimum == peek*
- *extractMin == remove*
- *isEmpty == isEmpty*

If we use our own PriorityQueue, we need to provide way to compare objects

Student.java

- Three ways to compare objects
- **Method 1: provide a compareTo method**
- Students have name and year
- @Override compareTo in Student class
 - Compare this student's name with param Student
 - Use String's built in compareTo() to compare names and return -1, 0, +1
 - name.compareTo(s2.name)
- Run
 - First demo ArrayList
 - Then addAll students to PriorityQueue
 - Remove one at a time (essentially sorting)

There are several ways to implement the comparator

Student.java

Method 2

Create a custom `Comparator` class that implements `Comparator`, instantiate, and pass to PQ constructor:

```
class NameLengthComparator implements
Comparator<Student> {
    Public int compare(Student s1, Student s2) {
        return s1.name.length() - s2.name.length()
    }
}

Comparator<Student> lenCompare = new
NameLengthComparator();
pq = new PriorityQueue<Student>(lenCompare)
```

There are several ways to implement the comparator

Student.java

Method 3


Use anonymous function

- Allows function in middle of code without giving it a name

```
pq = new PriorityQueue<Student>((Student s1,  
Student s2) -> s1.year - s2.year);
```

- Method body comes after ->
- Anonymous functions sometimes called Lambda expressions

Agenda

1. Priority queues
2. Java's priority queue
-  3. Reading from a file
4. ArrayList implementations

Use a BufferedReader to read a file line by line until reaching the end of file

Reading from a file

```
BufferedReader input = new BufferedReader(new FileReader(fileName));
String line;
int lineNum = 0;
while ((line = input.readLine()) != null) {
    System.out.println("read @" + lineNum + "`" + line + "'");
    lineNum++;
}
```


- *BufferedReader* opens file with name *filename*
- Reading will start at beginning of file
- Each line from file stored in *line* in while loop
- *input.readLine* will return null at end of file
- Here we are just printing each line

When reading files, we need to be ready to handle many different exceptions

Roster.java

- We try-catch the attempt to open the file, in case it's not there. Note that this requires splitting up the declaration of the variable from its initial assignment, so that the declaration sits outside of the try-catch.
- We try-catch the loop to read from the file, in case something goes wrong during reading.
- We test the number of comma-separate pieces in a line, and we try-catch the extraction of an integer from the second piece.
- We try-catch the closing of the file.
- One note: if we didn't want to print out the message for IO error while reading, we could have dropped the "catch" clause and instead put a "finally" clause that would be executed in either case (exception or not). This clause would wrap up the whole block of code to close the file, to ensure an attempt at closing in normal or exceptional circumstances. But then the method would pass the IO exception on up the line. That's illustrated in readRoster2.

Agenda

1. Priority queues
2. Java's priority queue
3. Reading from a file
-  4. ArrayList implementations

We can implement a PriorityQueue with an unsorted ArrayList

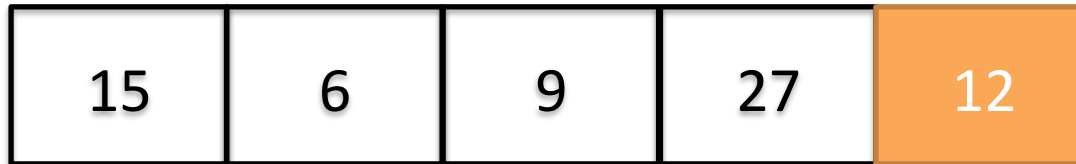
Unsorted ArrayList implementation

15	6	9	27
----	---	---	----

Keep elements unsorted in ArrayList

We can implement a PriorityQueue with an unsorted ArrayList

Unsorted ArrayList implementation

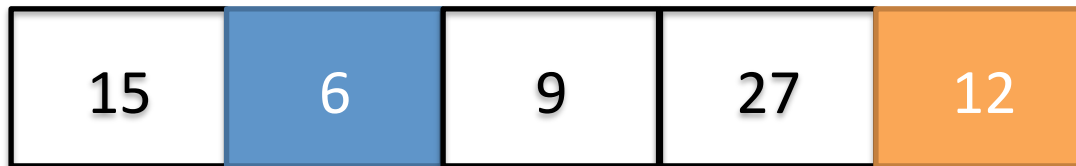


Inserting new items – just tack on to end

Operation	Run time	Notes
<code>isEmpty</code>	$O(1)$	Return size
<code>insert</code>	$O(1)$	Add on to end (amortized)
<code>minimum</code>	$O(n)$	Must loop through all elements to find
<code>extractMin</code>	$O(n)$	Loop through all elements and move to fill hole
<code>decreaseKey</code>	$O(1)$	Just update value

We can implement a PriorityQueue with an unsorted ArrayList

Unsorted ArrayList implementation



extractMin – get smallest and move last item to smallest index

Operation	Run time	Notes
<code>isEmpty</code>	$O(1)$	Return size
<code>insert</code>	$O(1)$	Add on to end (amortized)
<code>minimum</code>	$O(n)$	Must loop through all elements to find
<code>extractMin</code>	$O(n)$	Loop through all elements and move to fill hole
<code>decreaseKey</code>	$O(1)$	Just update value

We can implement a PriorityQueue with an unsorted ArrayList

Unsorted ArrayList implementation

15	12	9	27
----	----	---	----

extractMin – get smallest and move last item to smallest index

Operation	Run time	Notes
<code>isEmpty</code>	$O(1)$	Return size
<code>insert</code>	$O(1)$	Add on to end (amortized)
<code>minimum</code>	$O(n)$	Must loop through all elements to find
<code>extractMin</code>	$O(n)$	Loop through all elements and move to fill hole

We can implement a PriorityQueue with an unsorted ArrayList

ArrayListMinPriorityQueue.java

- Class Implements MinPriorityQueue interface
- Element type must provide Comparable
- Elements kept in *ArrayList list* of type E
- *indexOfMinimum()* returns index of smallest item in *list*
 - Loop through all elements compare with smallest so far
 - If smaller, then update index of smallest
 - Return smallest index
- *minimum()* uses *indexOfMinimum* with *get* to return smallest element
- *extractMin()*
 - Get smallest item
 - Move last item into hole created by removing smallest
 - Return smallest

We can improve extractMin by using a sorted List, but inserts take more time

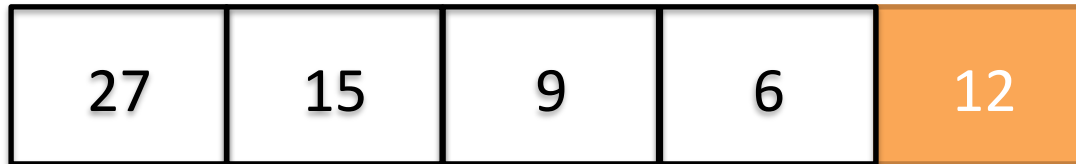
Sorted ArrayList implementation

27	15	9	6
----	----	---	---

Keep elements sorted in ArrayList with smallest at end

We can improve extractMin by using a sorted List, but inserts take more time

Sorted ArrayList implementation



Keep elements sorted in ArrayList with smallest at end

We can improve extractMin by using a sorted List, but inserts take more time

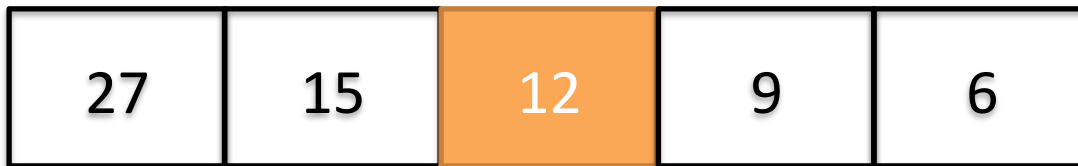
Sorted ArrayList implementation



Keep elements sorted in ArrayList with smallest at end

We can improve extractMin by using a sorted List, but inserts take more time

Sorted ArrayList implementation



Keep elements sorted in ArrayList with smallest at end

Operation	Run time	Notes
<code>isEmpty</code>	$O(1)$	Return size, same as unsorted
<code>insert</code>	$O(n)$	Insert in place and move other items right
<code>minimum</code>	$O(1)$	Get last element
<code>extractMin</code>	$O(1)$	Get last element, no need to move items

We can improve `extractMin` by using a sorted List, but inserts take more time

SortedArrayListMinPriorityQueue.java

- *insert()* – store in decreasing order so last item is the smallest
 - $O(n)$ to find place to insert into list with `add(index, element)`
- *minium()* and *extractMin()* now just get the last item
 - $O(1)$
- Generally the same number of inserts as extracts, so no real gain, unless just looking for min without extracting
- We will do better next class!