CMPT 225: Data Structures & Programming – Unit 15 – Priority Queues

Dr. Jack Thomas
Simon Fraser University

Spring 2021

Today’s Topics

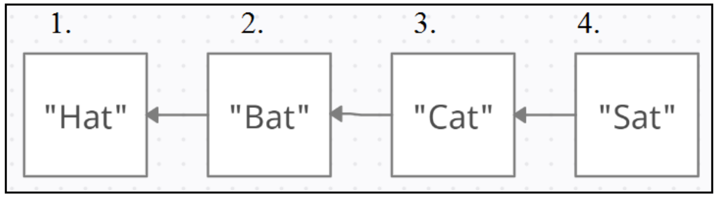
- Keys and Comparators
- The Priority Queue
- Priority Queue ADT
- Priority Queues in Java
- Implementing Priority Queues and Sorting

Positions vs. Keys

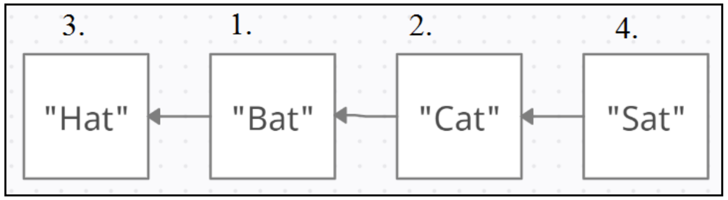
- When we introduced Trees, we discovered that all the data structures we'd covered so far were **linear**, whereas Trees were **non-linear**.
- Now we'll discover that all previous data structures were **position-based**, meaning the user is manipulating data via its position (using an index, addLast, head, tail, root…).
- **Priority Queues** are instead **key-based**, hiding exactly where the data is being stored inside itself and instead using a special key to retrieve it.

What is a Priority Queue?

- Exactly what it sounds like – a queue, except instead of giving you the element that's been in the queue the longest, it gives you the element that has the **highest associated "priority"**.
- Taking from a Queue:

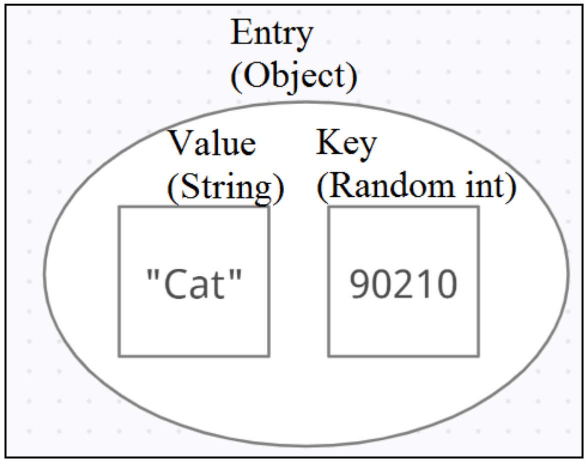


- Taking from a Priority Queue, prioritized alphabetically:



Okay, What is “Priority”?

- Priority is in quotes because **it doesn’t actually have to correlate to priority**. It could be alphabetical by name, or sorting a randomly generated ID number from smallest to largest.
- Whatever it’s based on, this priority score is the **key**, whereas the data it is paired to is the **value**. Together, they make up an **entry** into the Priority Queue.



Properties of Keys

- **They don’t have to be unique.**
  - The chosen key might be unique per element, but the broadest definition allows duplicate keys.
- **They don’t have to be one thing.**
  - Keys can be calculated from several attributes, or even from something not directly stored with the object, so long as the key is consistent each time it’s called.
- **For a PQ, they do have to achieve a total ordering.**
  - It must be possible to achieve a linear ordering from largest to smallest using the keys, without any contradictions (i.e. A greater than B, B greater than C, C greater than A).
  - Therefore, there will exist a definitive smallest key or keys.

Entries and Comparators

- Entries are **essentially nodes** which each store one value and one key.
- To find and return the entry with the **smallest key**, there must be a **rule for comparing keys**.
- These rules can’t be stored with every key, requiring special **Comparator objects** which can **take in two keys and return which one is smaller**.

Comparator Example

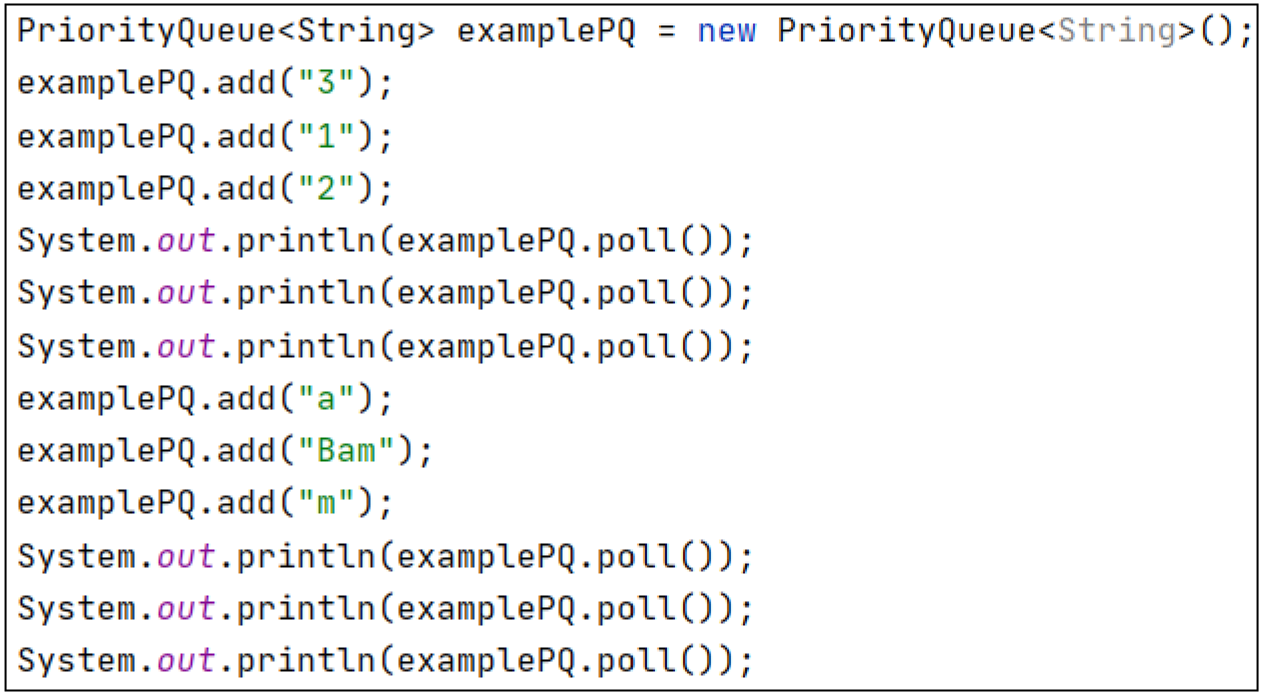
- Say I'm using the **day of the month** as my key.
- One entry has the 4th, while another has the 12th.
- If read as **integers**, then **4 < 12**, but if read as **Strings**, then **12 < 4** (because it starts with 1).
- What if I decided to include the month as part of the key? Then the **4th of February** would come before the **12th of February**, but after the **12th of January**.
- Defining these rules and deciding which key comes first is the job of a Comparator object.

The Priority Queue ADT

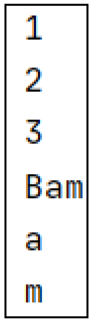
- A **data structure** for storing **entries** containing data **values** and **keys**.
- **Based on keys** included with each entry **rather than their positions** in the queue.
- Standard methods include:
  - **Insert**: Adds a given key and value to the Priority Queue, and returns their combined entry.
  - **removeMin**: Removes and returns an entry of P with the smallest key. (Sometimes called poll, from queue)
  - **Min**: Returns but does not remove an entry of P with the smallest key. (Sometimes called peek, from queue)
  - The usual generic methods from Queue as well, like isEmpty() and size().

Priority Queues in Java

- There is a **standard Priority Queue class in Java** which automates and hides a lot of the work of setting up most default comparators.



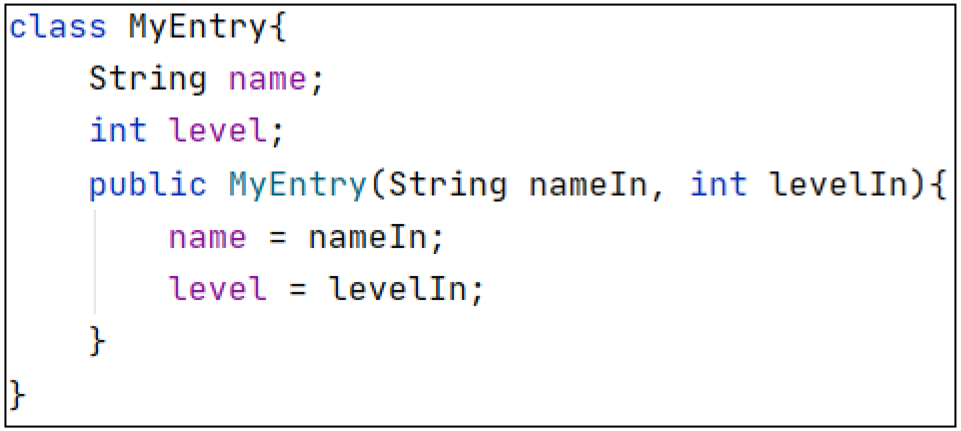
```java
PriorityQueue<String> examplePQ = new PriorityQueue<String>();
examplePQ.add("3");
examplePQ.add("1");
examplePQ.add("2");
System.out.println(examplePQ.poll());
System.out.println(examplePQ.poll());
System.out.println(examplePQ.poll());
examplePQ.add("a");
examplePQ.add("Bam");
examplePQ.add("m");
System.out.println(examplePQ.poll());
System.out.println(examplePQ.poll());
System.out.println(examplePQ.poll());
```



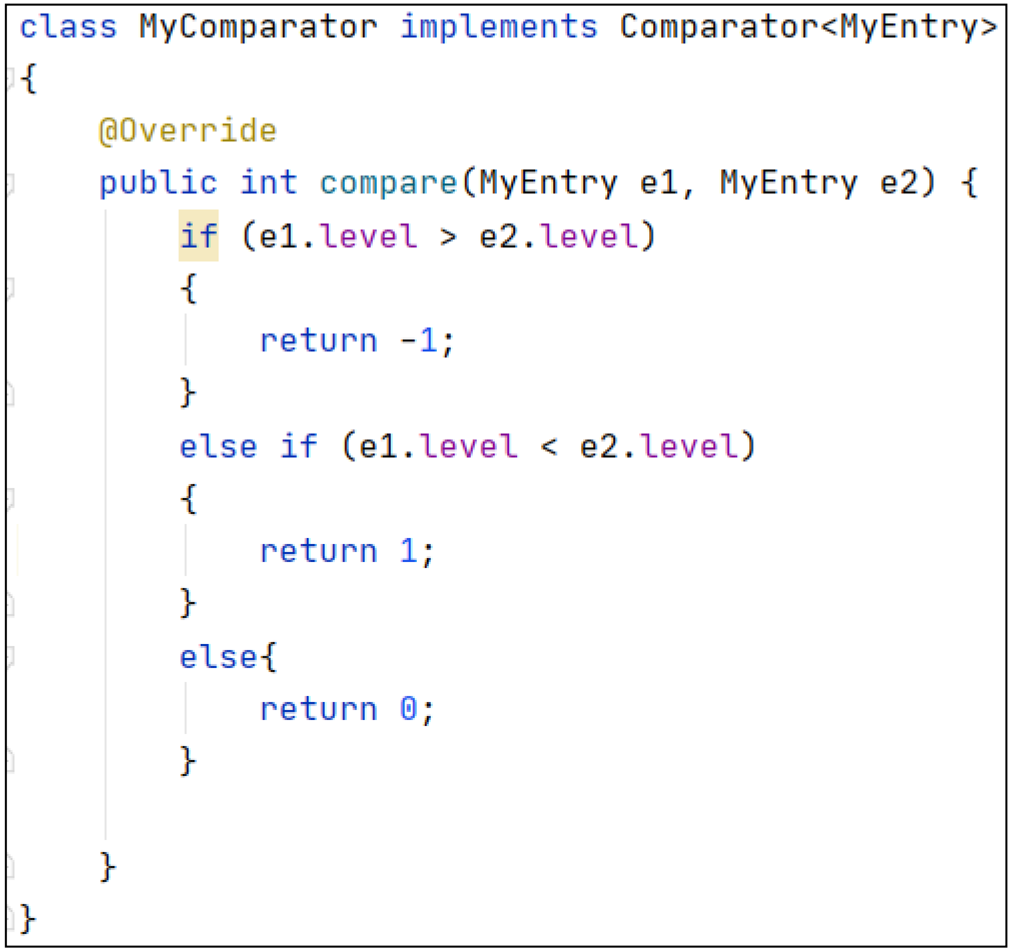
```
1
2
3
Bam
a
m
```

Comparators in Java

- PriorityQueue lets us **set a Comparator** of our own for prioritizing objects.
- **Comparator** is a **standard interface** that lets us define a compare(entry 1, entry 2) function to decide which is “smaller”.
- Say we have entries representing characters in a video game with a String name and an int level, and we want the Priority Queue to prioritize whoever has the **highest level**, not the lowest.

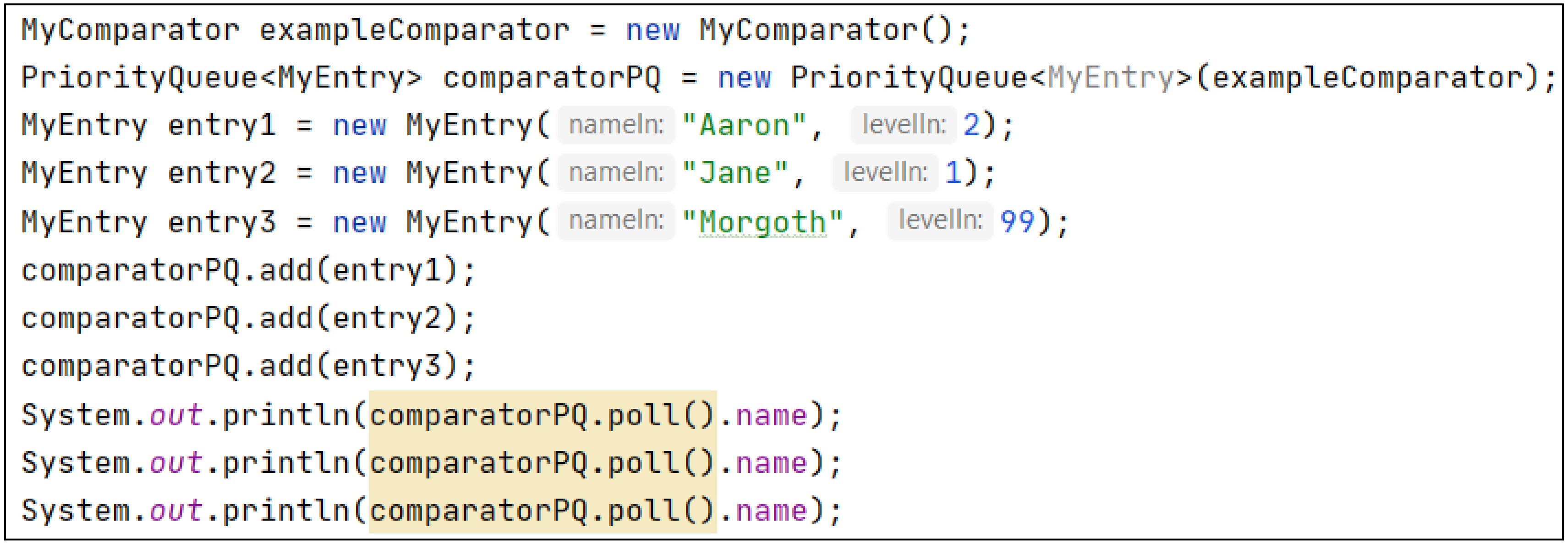


```java
class MyEntry{
    String name;
    int level;
    public MyEntry(String nameIn, int levelIn){
        name = nameIn;
        level = levelIn;
    }
}
```

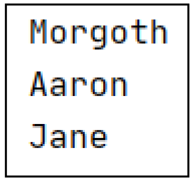


```java
class MyComparator implements Comparator<MyEntry>
{
    @Override
    public int compare(MyEntry e1, MyEntry e2) {
        if (e1.level > e2.level)
        {
            return -1;
        }
        else if (e1.level < e2.level)
        {
            return 1;
        }
        else{
            return 0;
        }

    }
}
```



```java
MyComparator exampleComparator = new MyComparator();
PriorityQueue<MyEntry> comparatorPQ = new PriorityQueue<MyEntry>(exampleComparator);
MyEntry entry1 = new MyEntry( nameIn: "Aaron",  levelIn: 2);
MyEntry entry2 = new MyEntry( nameIn: "Jane",  levelIn: 1);
MyEntry entry3 = new MyEntry( nameIn: "Morgoth",  levelIn: 99);
comparatorPQ.add(entry1);
comparatorPQ.add(entry2);
comparatorPQ.add(entry3);
System.out.println(comparatorPQ.poll().name);
System.out.println(comparatorPQ.poll().name);
System.out.println(comparatorPQ.poll().name);
```



```
Morgoth
Aaron
Jane
```

Implementing a Priority Queue and the Importance of Sorting

- The big implementation question involved in a Priority Queue is **how you keep track of the highest-priority entries**.
  1. **Option one** is to **keep the queue sorted according to priority**. You can poll the next highest priority entry right off the front of the queue, but every time you add an entry you need to sort the queue again.
  2. The **alternative** is to just **add each entry to the back of the queue** as normal, but now every time you poll the queue you'll have to **search the whole thing for the highest priority element**.

Insertion Sort and Selection Sort

- Using a **sorted list** as the basis for our Priority Queue requires an **insertion sort**, which triggers **when a new entry is added**.
- If we use an **unsorted list**, then the **add function remains unchanged**. Now when you **poll**, you'll need to trigger a **selection sort**.
- The details of these functions will depend on our choice of **underlying data structure** (in this case, a list) and **matching algorithm**, but **will be constrained by their run-time efficiency**.

Comparing Run-Times

- In a naive sequence (list) based implementation, we essentially need to **rebuild the queue every time** as part of insertion or selection sorting, resulting in O(n) for the rebuilding and O(n) for the sorting per element.
- The Priority Queue based on an **unsorted list** can **add elements at O(1)**, but **retrieves them at $O(n^2)$**.
- The one based on a **sorted list** can **retrieve entries at O(1)**, but **adds them at $O(n^2)$**.
- As such, our choice of implementation should reflect the needs of our situation!

Recap – The Highest Priority Lecture Points

- **Priority Queues** are our first data structure that's **key-based** rather than **position-based**.
- **Keys** are paired with data **values** in **entries**, and can be **compared** with one another using **comparators** to generate an **ordering**.
- The **Priority Queue ADT** adds entries to the queue and then retrieves the entry with the **smallest key** (highest priority).
- Java has a **standard Priority Queue class** and **Comparator interface** we can use to define our own rules for priority.
- The implementation of the Priority Queue **depends on when the queue is sorted**, and will significantly impact the run time.