CMPT 225: Data Structures & Programming – Unit 17 – Adaptable Priority Queues & Maps

Dr. Jack Thomas

Simon Fraser University

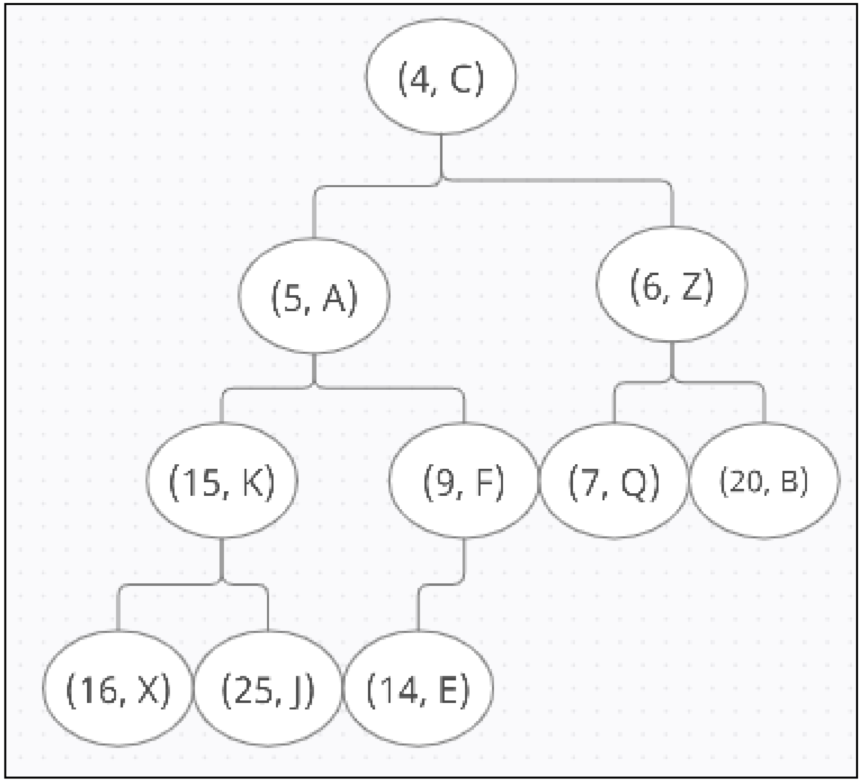
Spring 2021

Today’s Topics

- Refreshing Priority Queues
- Justifying Adaptation
- The Adaptable PQ ADT
- Locations, Positions, & Entries
- Maps

Post-Reading-Week Refresher

- Before the break, we introduced **Heap-based Priority Queues**, which organize data into binary trees sorted according to their keys.
- This combined the benefits of **nonlinear storage** with those of **nonpositional access**.



There Are Still Some Limitations

- Our Priority Queue is still a Queue, meaning we can only reliably **retrieve the entry stored at the root**.
- This also makes **changing an entry's key or value** more complicated, since we need to find it first – and making changes could require reordering the entire Priority Queue.
- The Heap structure makes this even more complicated, since now we need to care about **tree traversals and Heap sorting** to make any changes.

For Example: Consider An Airport

- At first, a normal Priority Queue might seem like enough to let us board a plane by priority.
  - What if a passenger pays to **upgrade their ticket**?
  - What if an existing ticket gets **transferred to a different passenger**?
  - What if a passenger **cancels last-minute** and re-books on a different flight?

Image credit: https://www.internationalairportreview.com/article/81738/smarter-way-cut-queue/

Making Priority Queues Adaptable

- We want to augment our existing Priority Queue with ways to **remove a given entry**, or **modify an entry's key or value**, and be confident that the Priority Queue is still in the correct order.
- This will **require reintroducing the idea of positions** to PQs, so we can start considering and acting on more than just the root or head.

The Adaptable Priority Queue ADT

- An **extension of the Priority Queue** data structure that allows for removing and editing arbitrary entries, not just the highest priority.
- Standard methods include all of the PQ ones, as well as:
  - **Remove**: Removes a given entry from the PQ, while ensuring it remains ordered.
  - **replaceKey**: Swaps the key of a given entry, then adjusting the ordering as needed.
  - **replaceValue**: Swaps the value of a given entry, which probably also requires re-checking the ordering.

Adaptable Priority Queues in Java

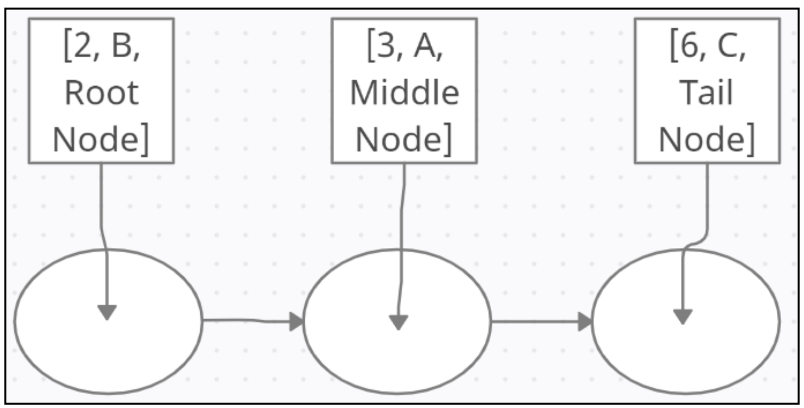
- In Java, the Adaptable Priority Queue is…
- …the **PriorityQueue** class! The same one you've already been using!
- It has a **remove() function built-in** that lets you remove any arbitrary entry, so long as you have a reference to it (which might be the hard part).
- However, it **doesn't have replaceKey() or replaceEntry()** functions, you'll just have to remove() the old object and then re-add it after modifying it.

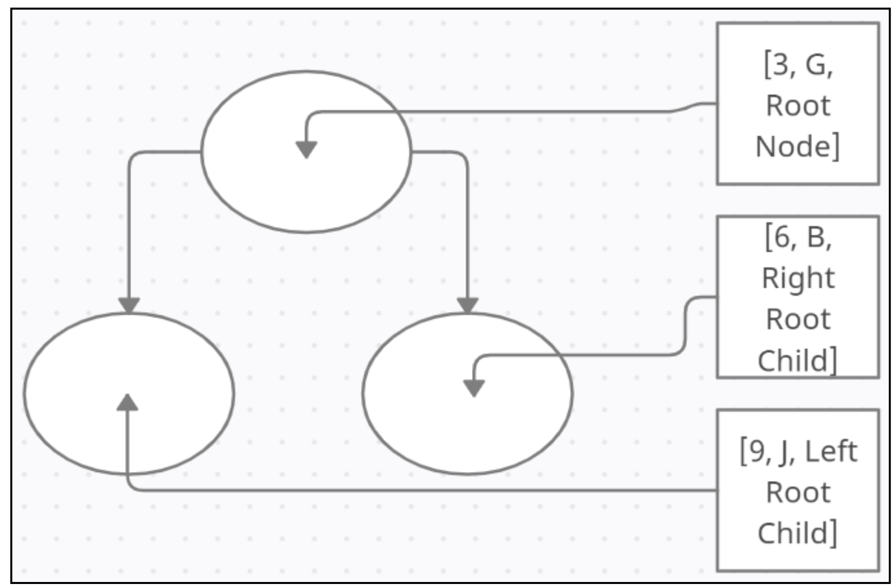
Location-Aware Entries

- When implementing our own Adaptable Priority Queues, we need to start storing each entry's **location** alongside the value and key.
- Depending on the underlying data structure, this location is stored in one of two ways:
  – In the **sorted list** implementation, the location is that entry's place in the linear sequence.
  – In the **heap** implementation, the location is that entry's node in the heap's tree structure.

Distinguishing Between Positions and Entries

- With both lists and heaps, the key is separating the **position** within the structure from the **entry** that occupies that position.





Position-Entry-Location Walkthrough

- Picture a **double-linked-list-based Adaptable Priority Queue**.
- The actual list is made up of **Positions**, a kind of **node** that store links to the nodes ahead and behind in the list, and an **Entry**.
- The **Entry** contains the **value**, **key**, and **location**, which is a link to the **Position** currently storing it.
- Now you can use the **Location** links between the **Position** and **Entry** to track each **Entry** for the sake of functions like **remove()**.

Efficiency

- If you'll recall, our options for how to organize the underlying data structure for a PQ are an **unsorted list** (constant adds, O(n) removes), **sorted list** (constant removes, O(n) adds), or **heap** (log n for both).
- Adding position-awareness makes our remove and replace functions constant for the **unsorted list**, O(log n) for the **heap**, and constant for remove/O(n) for replace for **sorted lists**.

Next Up: Maps

- Another detail about key-based structures so far is that keys don't have to be unique.
- What if they were?
- In a **Map**, **every key** stored is **unique**.
- Imagine a filing cabinet, where every folder (**entry**) has a unique label (**key**), and they're kept in order (**position**) inside a cabinet drawer (the **map** itself).

Image credit: https://www.amazon.ca/Hirsh-SOHO-Drawer-Cabinet-Charcoal/dp/B01ASUWBQM

Grouping Together Related Content

- It can help to remember that the value attached to each key **can be more than a single piece of data** – it can be a whole object, or a collection of objects.
- In a filing cabinet, the label of each folder could be a **student's ID number**, while inside the folder could be **any and all documents** having to do with that student.
- Maps are therefore sometimes called **associative stores**, because each entry stores everything associated with that unique key.
- It can also help to think of keys in a Map as functioning like an **index**.

The Map ADT

- A unique-key-based data structure, storing a set of key-value pairs called entries.
- Standard methods include:
  - **Get**: Return the value associated with the given key.
  - **Put**: If a given key doesn't exist in the map yet, add it and the given value, otherwise replace the existing value of the given key with the given value.
  - **Remove**: Removes and returns the entry associated with a given key.
  - **keySet**: Returns a collection of all the keys stored in the entries.
  - **Values**: Returns a collection of all the values stored in the entries.
  - **entrySet**: Returns a collection of all entries.

The Map in Java

- **Java** has a standard **Interface for a Map**, but no standard class.
- Be aware that implementing this interface will **require you to use an Entry class that implements the Interface for java.util.Map.Entry**, in order to satisfy the entrySet() interface function.

Get Algorithm

- Given a Map M based on a List S, we want to get the value associated with a key k.

```
Algorithm get(k):
        Input: A key k.
        Output: The value for key k in map M, or null
                if there is no key k in M.
        for each position p in S.positions() do
                if p.element().getKey() = k then
                        return p.element().getValue()
                return null
```

Put Algorithm

```
Algorithm put(k, v):
        Input: A key-value pair (k, v).
        Output: The old value associated with key k in M,
                or null if k is new.
        for each position p in S.positions() do
                if p.element().getKey() = k then
                        t <- p.element().getValue()
                        B.set(p, (k,v))
                        return t
        S.addLast((k,v))
        n <- n + 1
```

Remove Algorithm

```
Algorithm remove(k):
        Input: A key k.
        Output: The (removed) value for key k in M, or null
                if k is not in M.
        for each position p in S.positions() do
                if p.element().getKey() = k then
                        t <- p.element().getValue()
                        S.remove(p)
                        n <- n - 1
                        return t
        return null
```

Recap – Adapting the Lecture

- **Adaptable Priority Queues** extend PQs by adding the ability to remove arbitrary entries and adjust an entry's key or value.
- To do this, we **reintroduce the notion of positions and locations** to key-based data structures.
- The standard **Java PriorityQueue is adaptable** already, with some minor workarounds.
- **Maps** are a key-based data structure where every key is unique.
- Java doesn't provide a Map class, but it does provide a **Map interface**.