CMPT 225: Data Structures & Programming – Unit 16 – Heaps

Dr. Jack Thomas
Simon Fraser University
Spring 2021

Today's Topics

- The Heap
- Complete Binary Trees & the Heap-Ordering Property
- Heap ADT
- Up-Heap & Down-Heap Bubbling
- Heap Sort
- Heaps in Java

The Drawback of Priority Queues: Efficiency

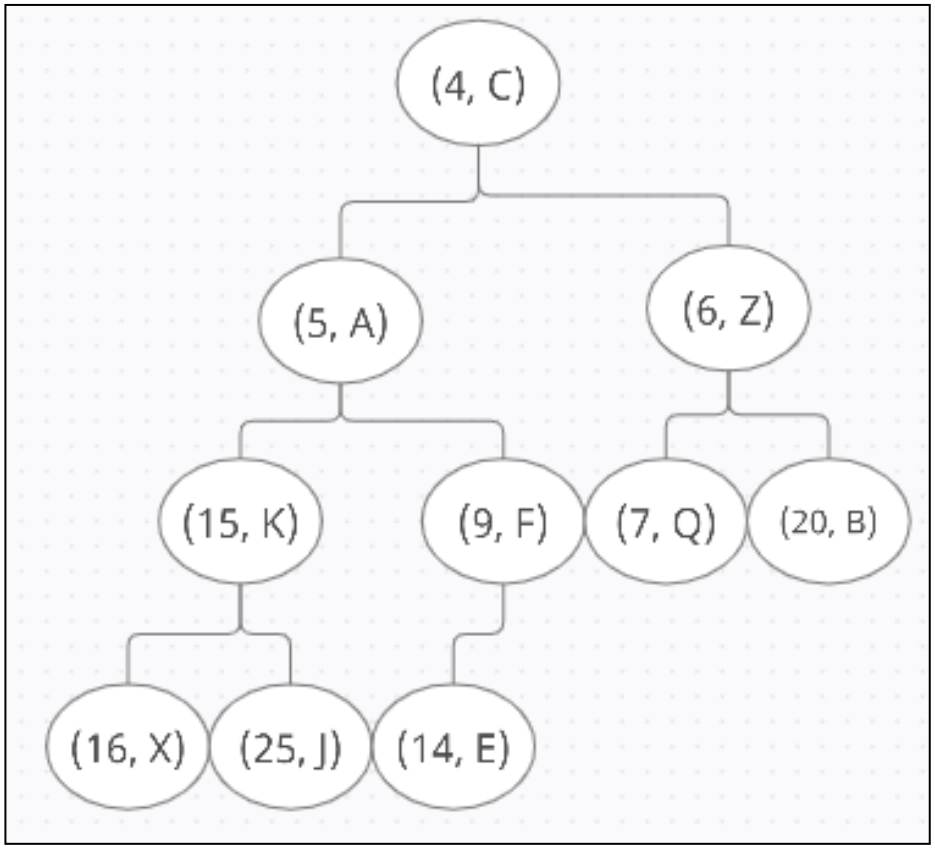
- As we learned previously, a naïve implementation for a Priority Queue must choose between **either keeping the data sorted** (O(n2) insertions and O(1) retrievals) or **unsorted** (O(1) insertions and O(n2) retrievals), meaning there's **always some inefficiency**.
- This is a result of the **slow speed of sorting an unsorted queue**, which is partly a consequence of the underlying structure.
- Our solution: **change the structure!**

Combining Non-Positional and Non-Linear Data Structures

- With **Priority Queues**, we learned about data structures that don't track elements using their position in the structure, but rather their key.
- With **Trees**, we learned about non-linear data structures, as well as some of the rules and constraints we can apply.
- It's time to… **COMBIIIIIINE**!

The Heap: Not Just a Disorganized Pile

- A Heap is a key-based data structure that stores the keys of its entries as a complete Binary Tree.
- The value of each child node's key in the Tree must be equal to or larger than that of its parent.



The Heap-Order Property

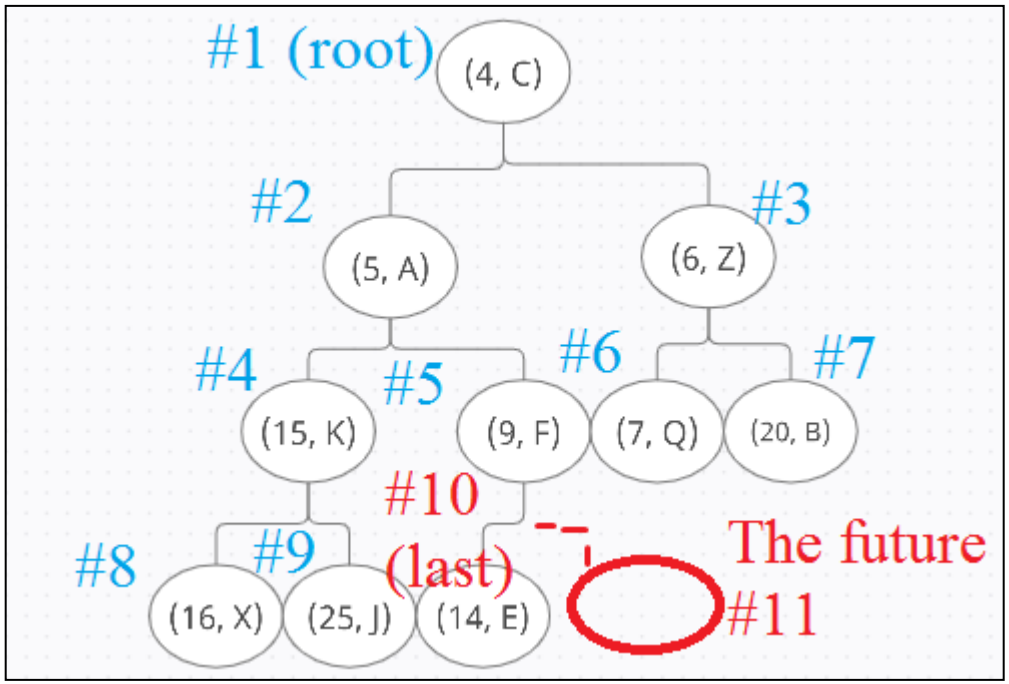
- The full and proper **definition** for this property states:
  - In the complete Binary Tree underlying the heap, for every node other than the root, the key stored by that node is **greater than or equal to** the key stored by their parent.

Complete Binary Trees

- The full and proper **definition** that makes a Binary Tree complete is:
  - A tree T with height h is a complete binary tree if levels 0,1,2…h-1 of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for 0 <= i <= h-1) and in level h-1, all the internal nodes are to the left of the external nodes, and there is at most one node with one child, which must be a left child.

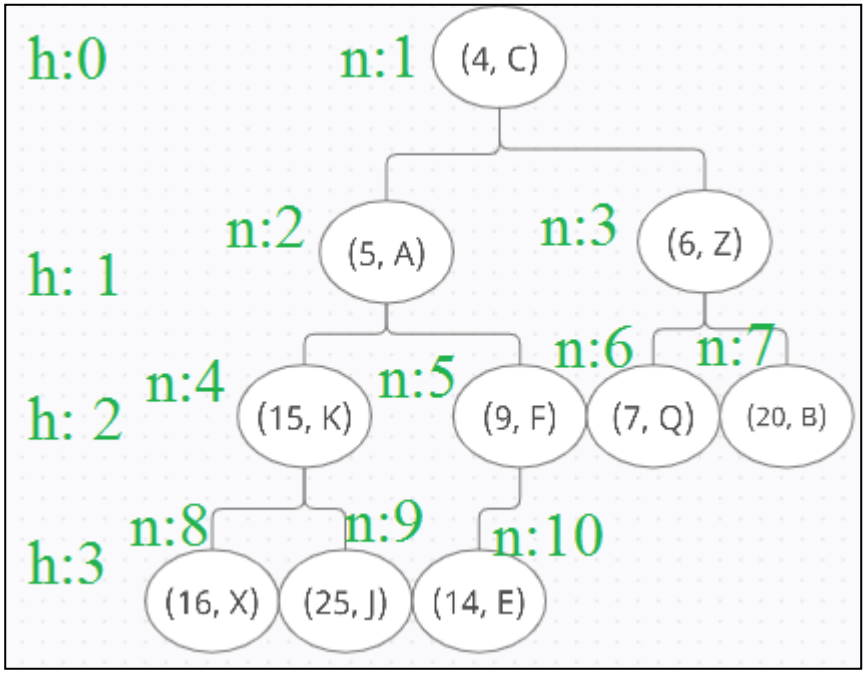
The Last Node

- Along with the **root** at the start, we now have the concept of the **last node**, which is the latest one to be added, and which is placed in the one spot the rules defining a Heap allow it to be placed.
- Another definition is **it is the node on level h** such that all the other nodes on that level **are to its left**.



The Height of a Heap

- By building our Heap according to the rules of a full Binary Tree and the Heap-Order, we can claim that Tree's height (h) = floor($\log_2 n$)
- Therefore, we can traverse the Tree from the root to any one leaf in O(log n).
- This can help us solve the insertion-sort vs. selection sort problem!



Where's the Heap ADT?

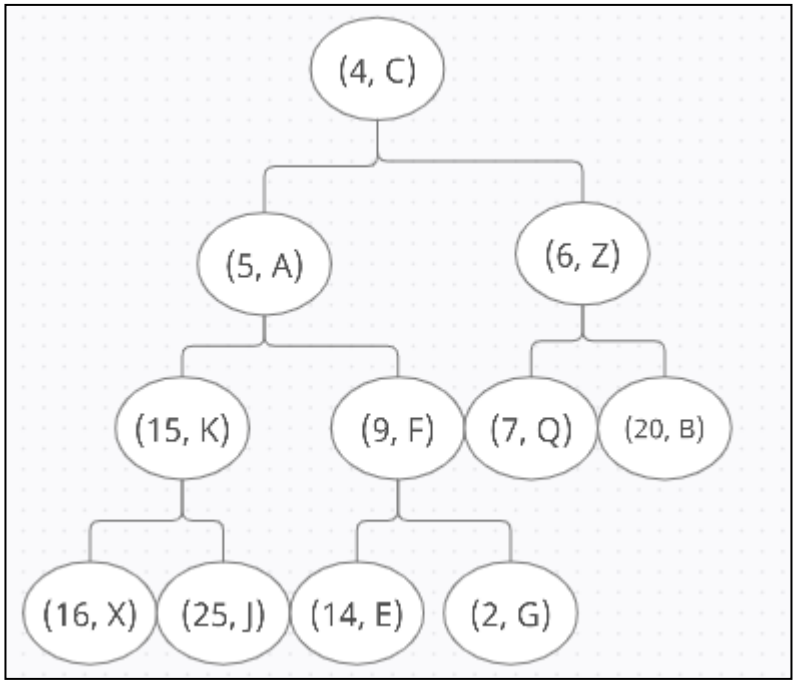
- Heaps, oddly enough, are **not considered true ADTs**.
- Instead, **complete Binary Trees that also adhere to the Heap-Ordering property** are called Heaps.
- We can **replace** the linear **Queue** structure within a **Priority Queue** with a **Heap** to gain the benefits of its more efficient layout.

Insertion in Heaps

- Since we always know where the current "last" node is, we can also tell where the next node to be added should go – this is called the **insertion node**.
- However, the new node may have violated the Heap-Order property if its key is smaller than its new parent.
- Our new add method must therefore restore this ordering via **up-heap bubbling**.

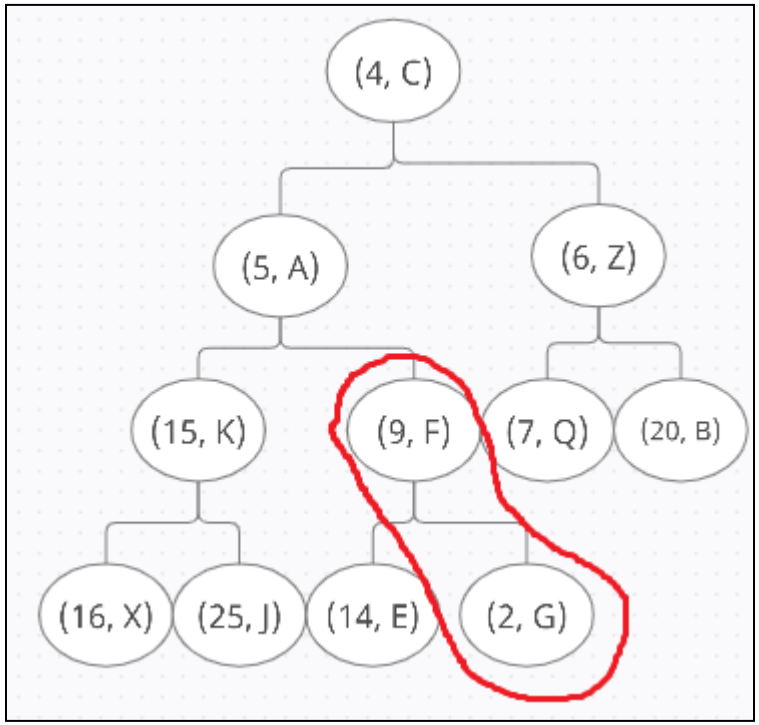
Up-Heap Bubbling

- First, we add our new node into the insertion node position of the Heap.
- Since we keep track of our last node, this should be easy to find.



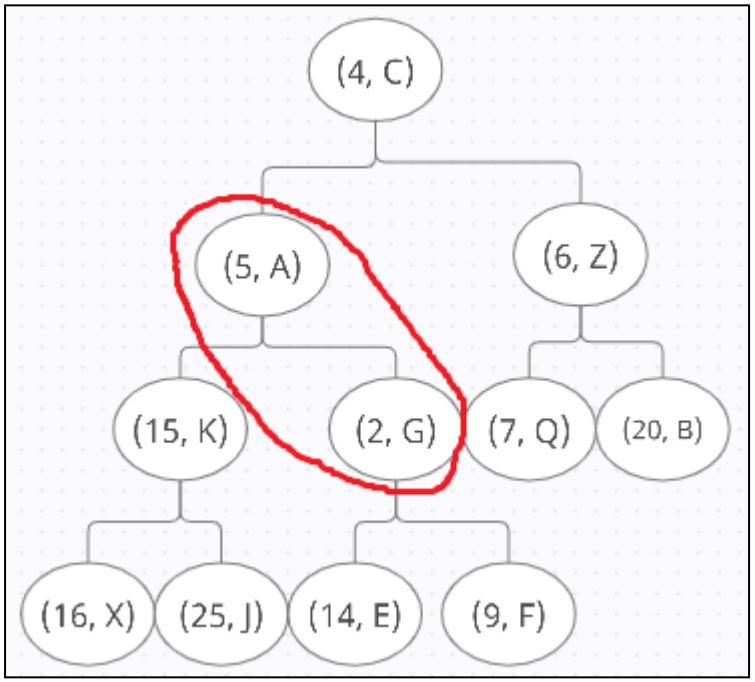
Up-Heap Bubbling

- Then, we begin comparing the key value of the newly-added child to that of its parent.
- If the child has a smaller key, then the two nodes are swapped.



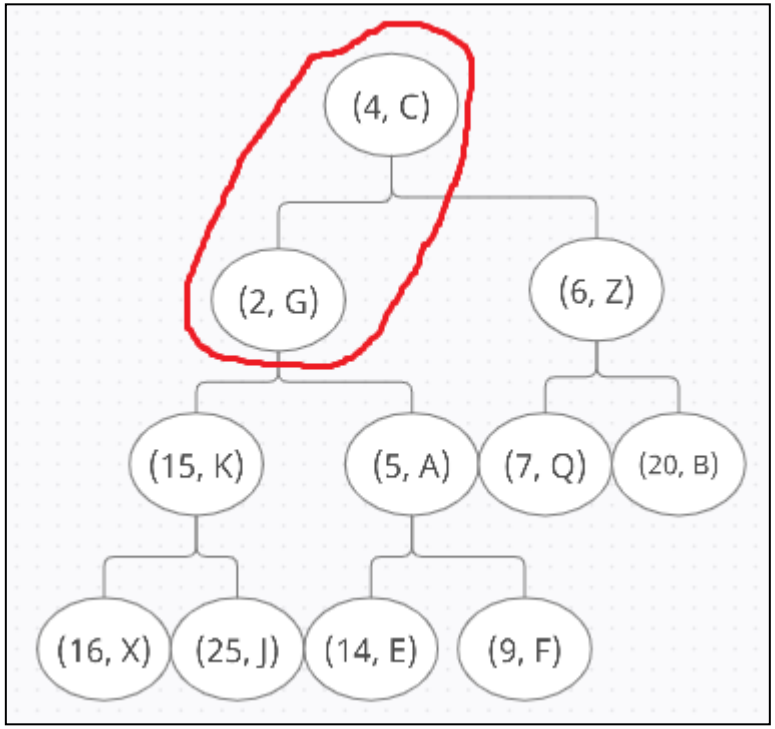
Up-Heap Bubbling

- This process then repeats with the node's new parent, and if their key is still smaller, then they also swap.



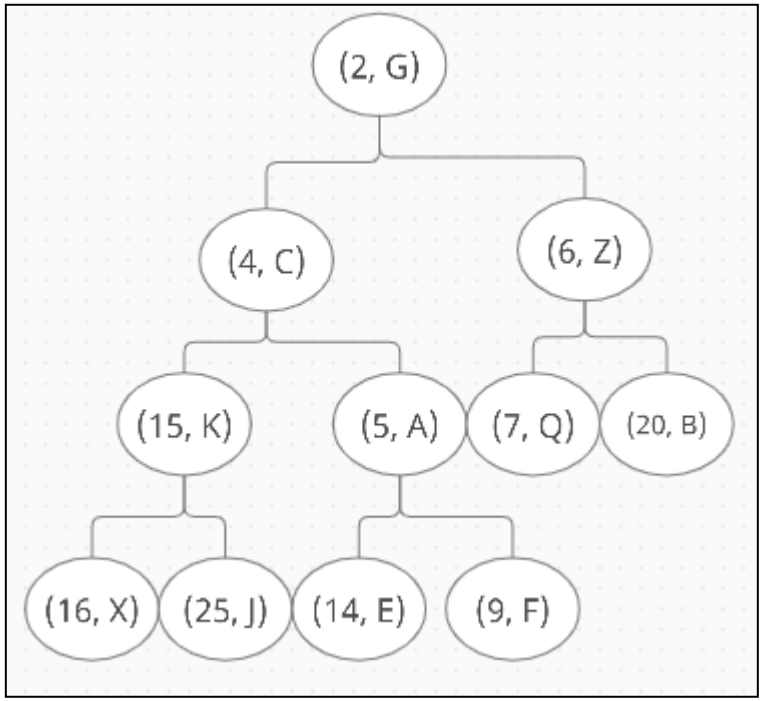
Up-Heap Bubbling

- The process continues until the node either finds a parent with a smaller key, or else swaps with the root and becomes the new root.



Up-Heap Bubbling

- Conveniently, since we know the height of the tree is no more than log n, and since the processes of adding the insertion node, comparing keys, and swapping parent and child nodes are all constant, this whole process is O(log n)



Returning and Removing the Smallest Key

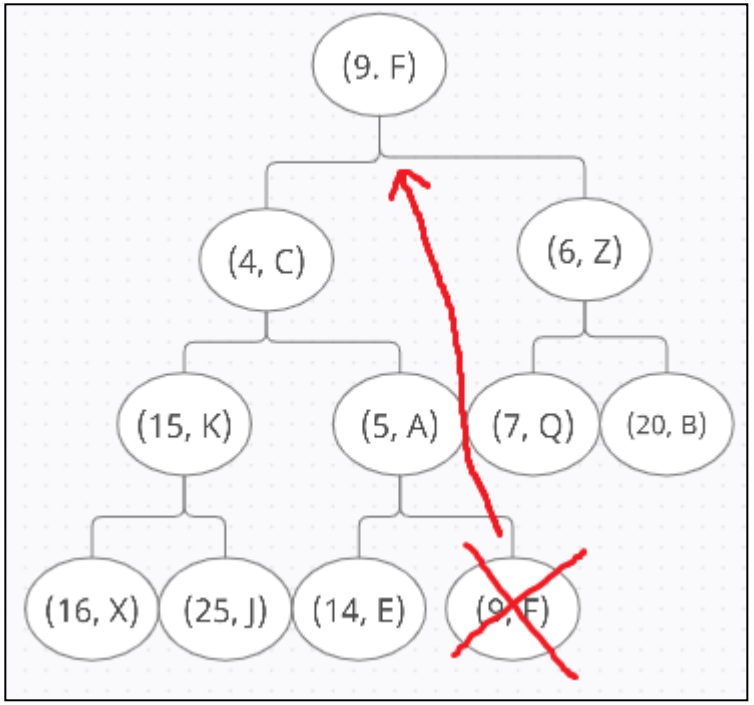
- Simply **returning** the highest-priority entry is easy, since that will be the **root**.
- Like adding, however, **removing the root** may cause the **Heap-Order property to be violated**.
- Therefore, our removeMin method from the Priority Queue must be modified to restore this property, through a process we call **Rootin-Tootin-Heap-Rebootin**.
- Nah I'm just kidding it's **down-heap bubbling**.

Down-Heap Bubbling

- You might think you already know how this will go, but it's a little more complicated than it sounds.
- First you should return (or copy to return later) the contents of the current root node, which will be the (or at least, a) smallest key.
- Then, you overwrite the current root with the current last node, and use the regular remove function to remove the current last node.

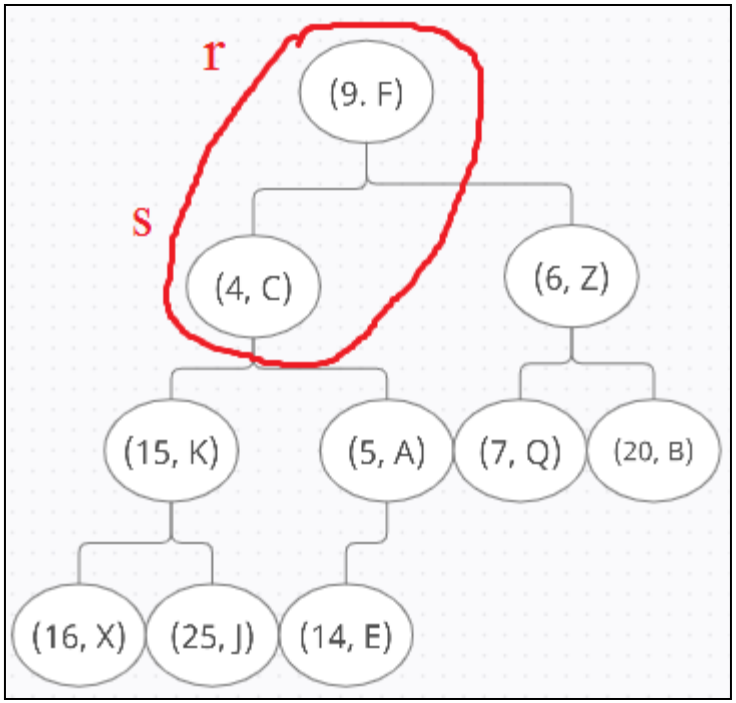
Down-Heap Bubbling

- We do this for a variety of reasons, like how it helps satisfy edge cases where there's only two nodes (root and last), but it almost certainly means we need to reorder.



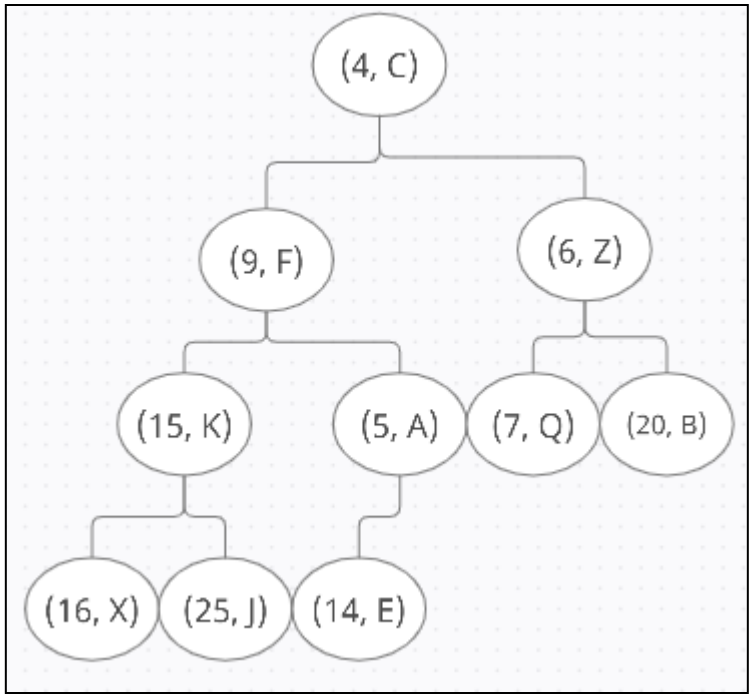
Down-Heap Bubbling

- If the new root had no children, we'd be done.
- Otherwise, if it had only one (left) child, we'd designate that child s.
- Otherwise, s is whichever of the two children of root has the smaller key.



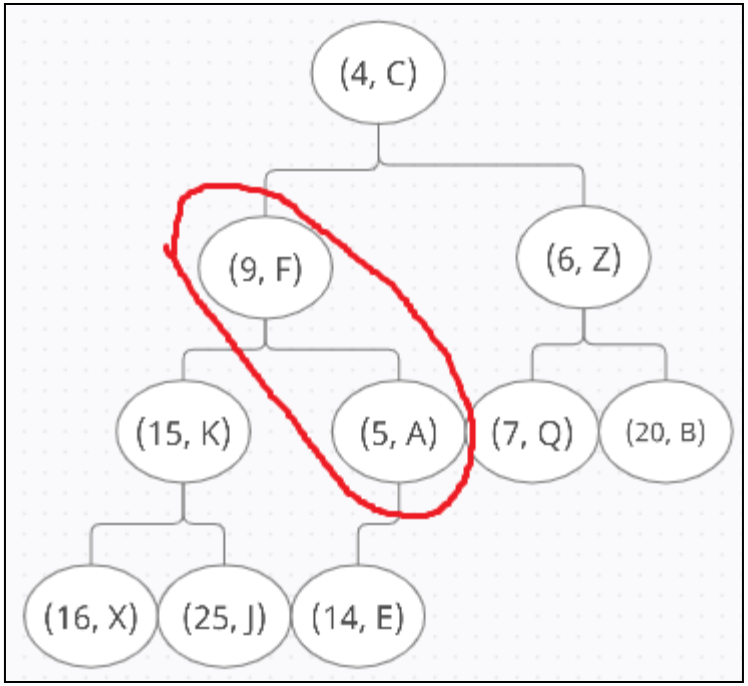
Down-Heap Bubbling

- If r < s, then the root and the child with the smallest key swap positions (since we chose the child with the smallest key, we don't have to worry about the new root and their former sibling)



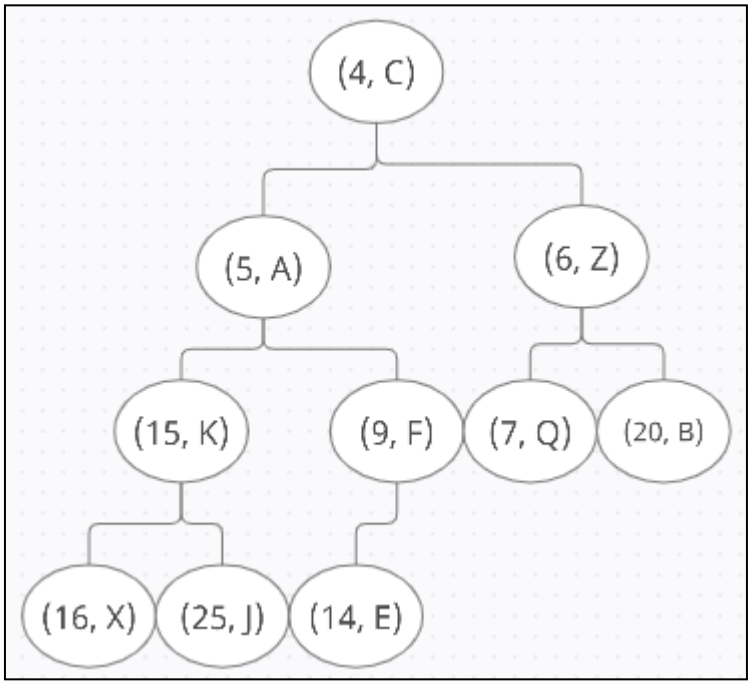
Down-Heap Bubbling

- This process then repeats for the newly swapped node, and will continue until it reaches a place where it is smaller than all of its children or it becomes an external node.



Down-Heap Bubbling

- Because this process only involves constant-time operations, and can only take as many iterations as the Tree has height, it too is **O(log n)**.



Analysis of Heap Methods for Priority Queues

- If we use a Heap and Heap Sorting for our **add** and **removeMin** methods, we can get both down to **O(log n)** in our Priority Queue.
- A full **Heap Sort** of an unsorted set of n elements, therefore, will take **O(n log n)**.
- This is, **in general**, a huge improvement over having to pick between **Insertion Sort's O(n2)/O(1)**, or **Selection Sort's O(1)/O(n2**).
- Be aware there are niche situations where one or the other could still be beneficial, **constant time can be very powerful!**

Heaps in Java

- Good news! The **PriorityQueue** class is actually **based on a Heap**!
- A fully-default PriorityQueue of integers (or one with a comparator that fishes an integer key out of a node object) will **act exactly like a min-Heap**, with the lowest-numbered (highest priority) key at the root.
- You can define a custom Comparator that flips the result to get a **max-Heap**, with the highest-numbered key at the root instead, or any other rule you want to use that gives a total ordering.

A Final Note on Why Heaps Work

- It turns out that to find the **next highest priority entry**, we don't actually need to keep the entire set ordered, just that **every family line** descending from the current root **is in order**.
- The **growth rate** of the time it takes to order any one family line is dependent on **the growth rate of the height** of the tree, not the number of inputs directly.
- This does mean that the Tree as a whole **isn't in a linear order** at any one time, which means pulling out arbitrary keys (like the fifth-highest priority) is more complicated – more on that later!

Recap – A Heaping Helping Of Slides

- A **Heap** is a combination of a **complete Binary Tree** and the **Heap-Ordering Property**.
- This binds the **growth-rate of the height of the tree**, and ensures each path from the **root** to a **leaf** is **ordered from smallest to largest**.
- We can use a Heap as the basis for a **Priority Queue** to improve and balance the run-times of the **add** and **removeMin** methods, by adding **up-heap** and **down-heap bubbling**.
- In **Java**, the default PriorityQueue is **based on a Heap** already.