Thomas Hudson

Professor James Pepe

CS401 Data Structures and Algorithms

Aug 9, 2020

Priority Queues and Heaps

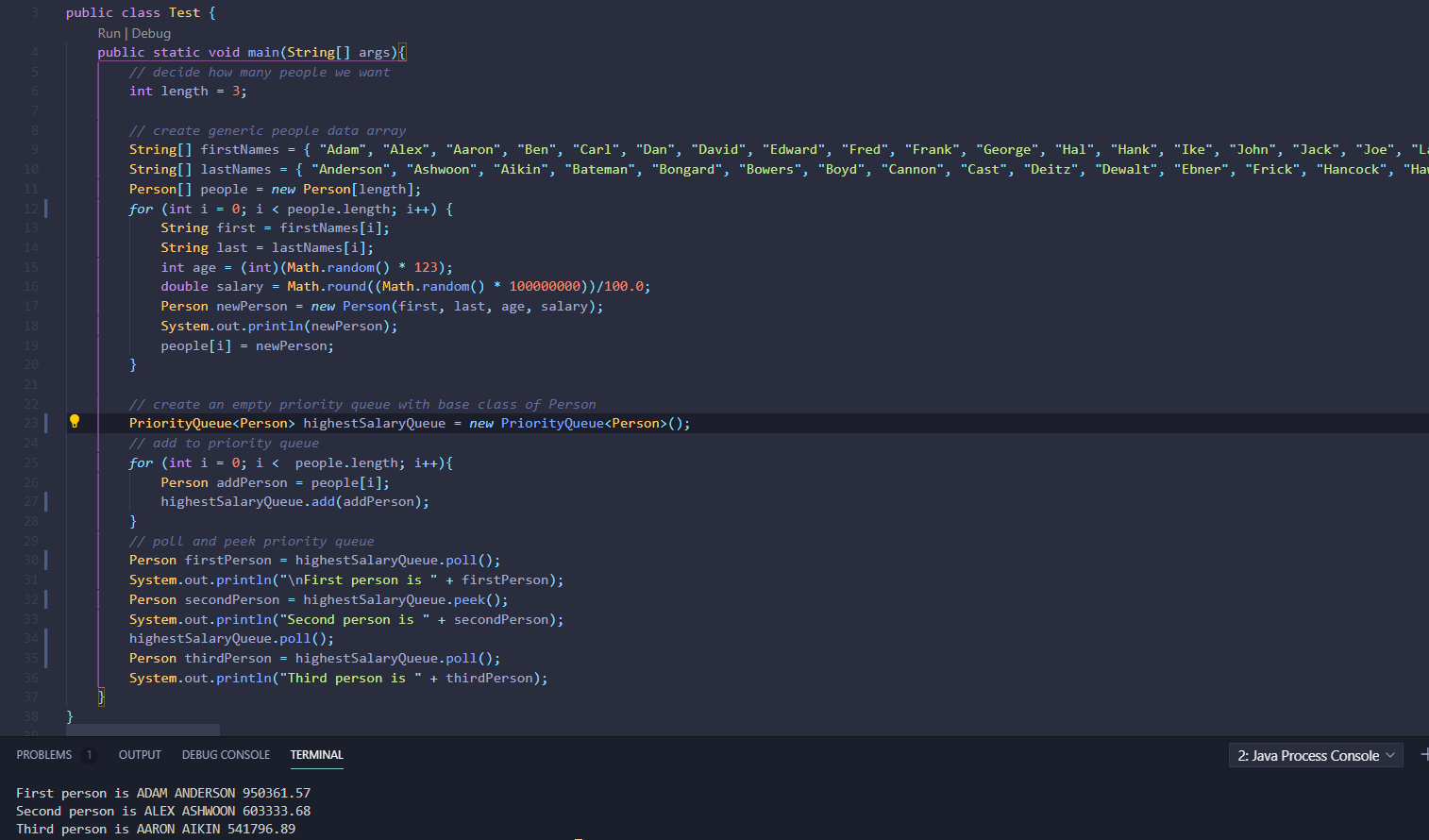
**Defining Priority Queues**

A priority queue is an abstract data structure that revolves around the concept of element priority. Each element within a priority queue has a given priority based on a comparison operator. This priority is what determines which elements of the queue are returned. As elements are returned from the queue, elements with the highest priority are returned first. Thus, as elements are removed from the queue, they are returned in descending order of priority. In order for a data structure to meet the requirements of a priority queue, it must be able to perform three operations, polling, adding, and check if empty (Skiena).

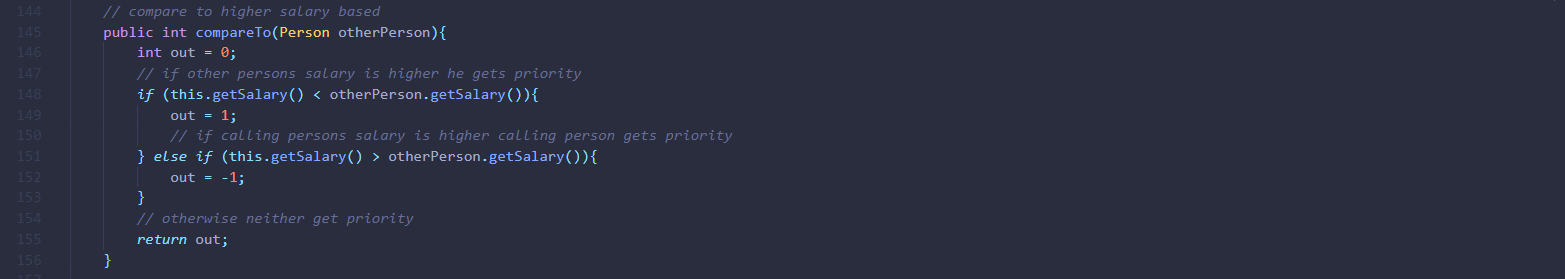
Polling a priority queue is the data structures most efficient action and is what gives the data structure value over others. When a priority queue is polled the element with the highest priority is returned and removed from the data structure in a highly efficient manner. The second method that a priority queue must implement is the add method. The add method inserts a new element into the priority queue in a location based on its individual priority. Lastly the priority queue must have a method to check if there are any items remaining within the queue. In addition, a common method that is used in priority queues but is not required within the abstract definition is a peek method (Boas et al.). A peek method allows for a copy of the next element that would be polled by the priority queue to be returned without removing the element from the queue.

In Java, priority queues are represented by the public class PriorityQueue<E> and are based on an underlying array data structure. Elements in a Java priority queue are ordered according to the elements “natural ordering”, or by a comparator provided at queue instruction time (Java Documentation). In Java, natural ordering refers to the order that can be derived from the inserted elements’ compare to method. Consequently, for elements to be added into a Java priority queue they must either extend the comparable interface (which requires the implementation of a compare to method) or, during instantiation of the priority queue, a comparator method must be defined with the type of element that is to be inserted.

**Priority Queue Implementation (Natural Ordering)**

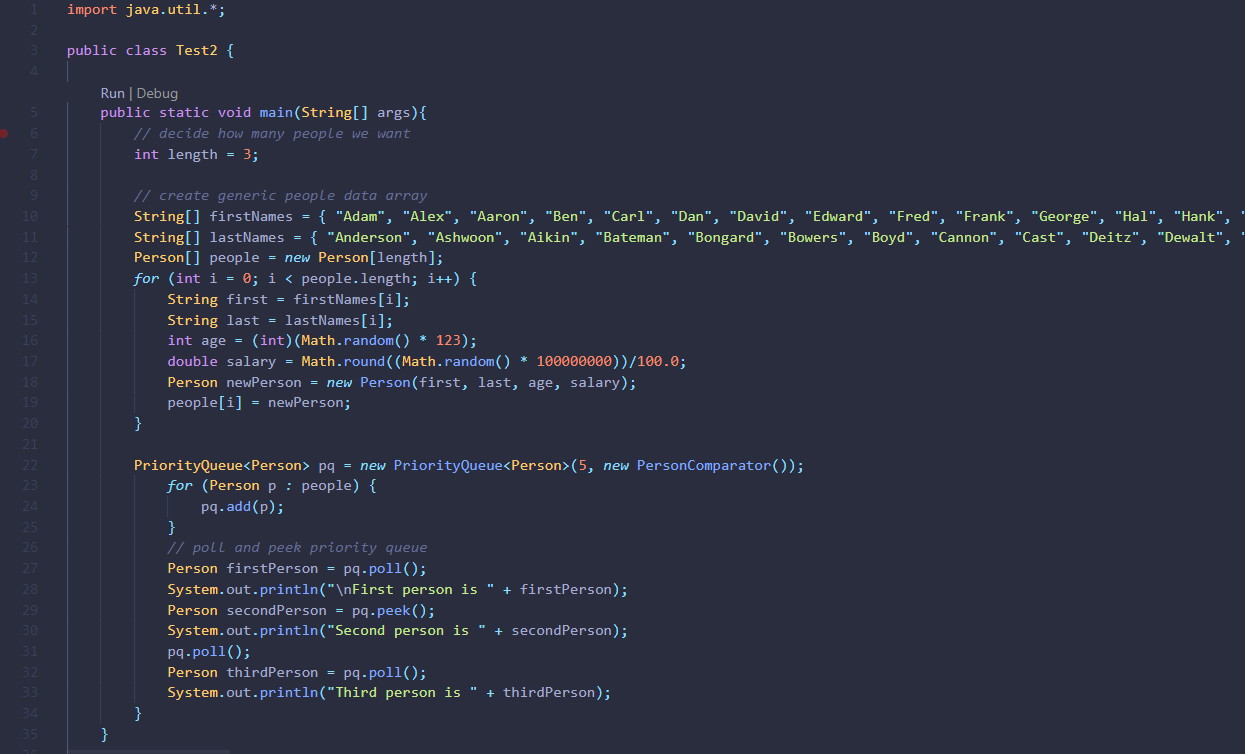


In the above example, using the Java priority queue class, a queue with a type of the person class is created. Three person objects are created and then inserted into the priority queue using the add method. The add method then compares the new person to other elements in the priority queue using the compare to method to determine its position in the array. The method in the person class is shown below.

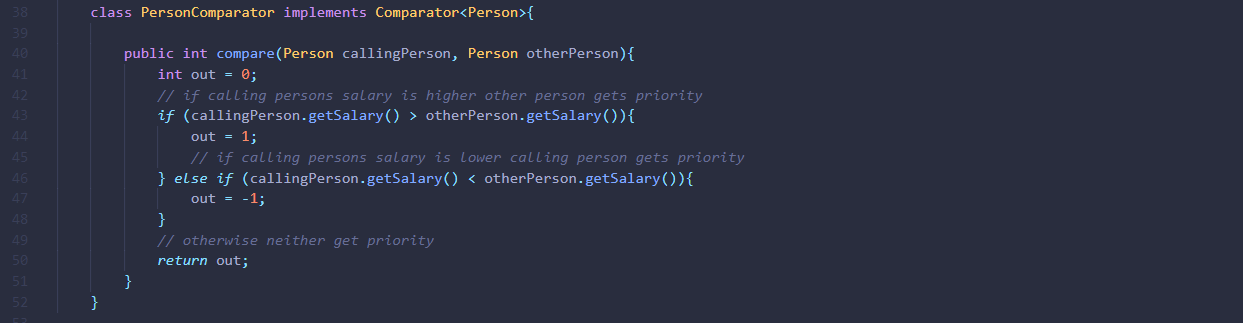


When polling is performed, Java simply removes the first element from the array and shifts other items’ positions forward.

**Priority Queue Implementation (Comparator Ordering)**



In this implementation, the natural ordering of elements is overwritten by creating a class that implements the comparable interface. Within instantiation of the priority queue, a new person comparator class is also instantiated. This person comparator class overrides the default compare to method within the person class and is used in the add method of the priority queue. In order for this to happen, the person comparator class must implement the comparator interface which requires that the class have a method called compare. This method is referenced within the priority queue class (Java Docs). The person comparator class is shown below.



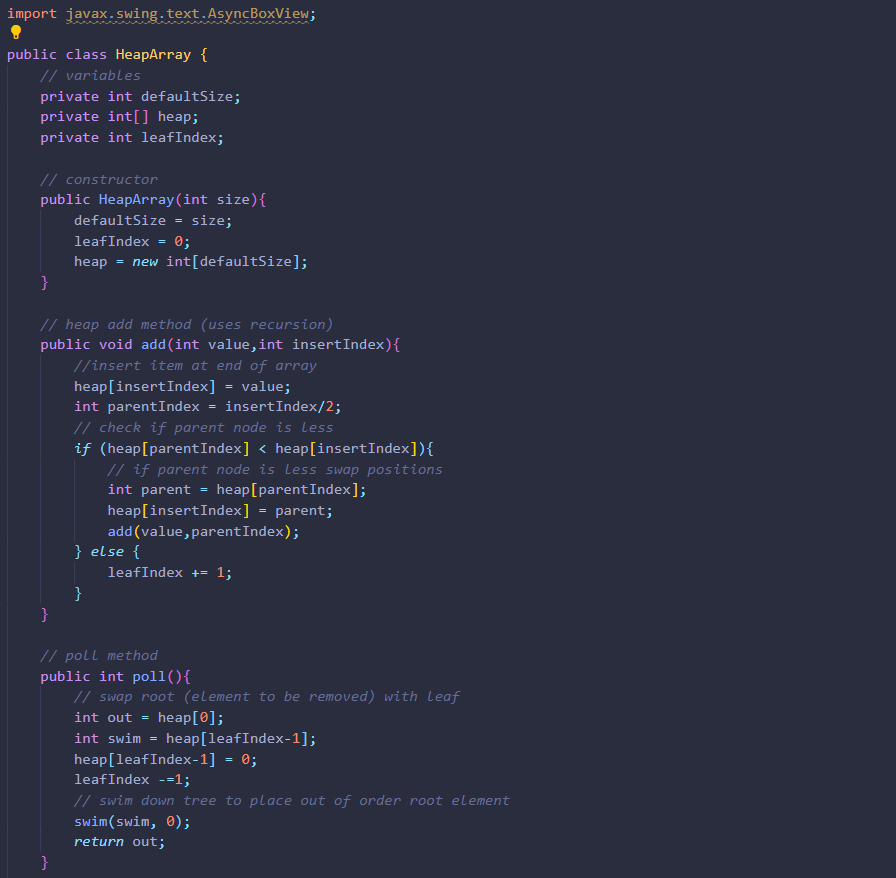
**Considerations for Java’s Priority Queue**

When implementing a priority queue using Java’s built in priority queue class it is important to consider the relative efficiency of the underlying data structure. Especially because Java’s priority queue defies the pure abstract definition of a priority queue. According to the definition of a priority queue the methods which define the utility of a priority queue are enqueing and dequeing elements to and from the priority queue in sorted order. However, Java uses a binary heap as its implementation of a priority queue offers a O(log(n)) time for these methods and a time complexity of O(n) for retrieval methods like peek (Java Docs). This means that for problems that require adding and removing often, the process can be inefficient. The solution is to instead use a more efficient version of a heap, the Fibonacci heap data structure. Before discussing this we must describe the construction of a heap.

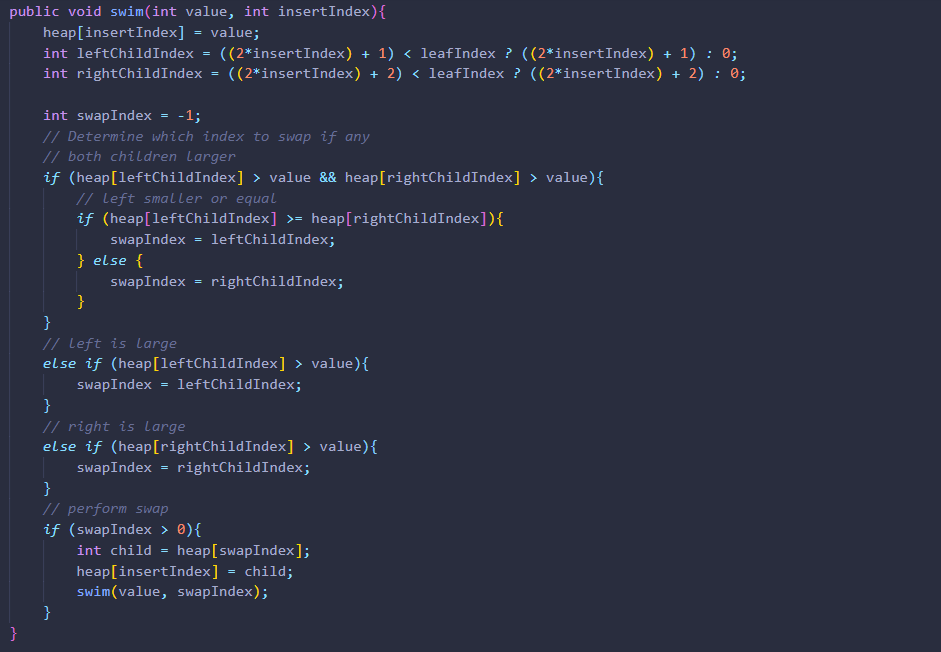
**Defining a Heap**

A heap is a tree-based data structure which definition stems from the principle that each parent node within the tree is of greater priority than all of its child nodes. To say it properly, “in a *max heap*, for any given node C, if P is a parent node of C, then the *key* (the *value*) of P is greater than or equal to the key of C. In a *min heap*, the key of P is less than or equal to the key of C.” (Black et al.). This also means that the item at the top of the heap is the item with the greatest priority, similar to priority queue. The heap is the most efficient form of a priority queue and is thus the data structure used by Java. To better understand how a heap works the following is an example of a simple heap based around integers.

**Heap Implementation**



The most common process for adding an element to a binary heap is called "bubbling up" the process begins by adding an item to tree at its max depth in the farthest left branch possible (if a row is not completely full do not make a new row). Following this, a comparison between the new node and the parent node is made. If the parent node is larger than the new node. The position is swapped. This will be repeated until the new node is in the correct position in the tree, moving upward until it has found its ideal position.



The polling process relies on similar but slightly more complex method. Firstly, the element with the highest priority is swapped with the element in the highest depth and farther right position in the tree. The high priority element is then cut from the tree and can be returned asynchronously. The element that now lies at the “top” of the tree then goes through a process called swimming (method shown above) which is in essence bubbling up but in reverse. The element is compared to each of its children, if either of them is of greater priority, the elements positions are swapped. If both children are of greater priority, we favor the left child which will make sure that the tree retains its structure. The above implementation matches the time and space complexity of Java’s priority queue.

**Alternatives to Java’s Priority Queue**

Java’s underlying data structure behind a priority queue is a binary heap as defined earlier. However as shown above there are “better” versions of the heap data structure, the most elegant but also lazy version of these is the Fibonacci heap. The Fibonacci heap achieves its efficiency by procrastinating the reordering of elements to only when a poll of the heap occurs. Instead of using a strict binary tree like Java’s heap implementation, a Fibonacci heap is constructed of a linked list of root nodes of individual trees. When an element is added, it is merely tacked onto the linked list. If an element is removed, it is broken off from its tree and if it has child nodes they are added top linked list as well. This makes these operations far more efficient. However, the poll method uses a complex and costly procedure to repair the damage to the trees organization that is caused by the add and remove methods. If a Fibonacci heap were to be used such that the poll method was never used, the time complexity of the Fibonacci heap degrades to a level of O(n). Furthermore, if the Fibonacci heap were only to use the poll method it is less efficient then a binary heap after repeated operations. The inner workings of a Fibonacci heap are beyond the scope of this paper, however the Fibonacci heap has seen uses in the Dijkstra’s algorithm and the traveling salesman problem (Ahuja et al.).

**Conclusion**

To conclude, priority queues are a unique data structure that work to balance outright organization and efficiency. For example, in comparison to a sorted list, priority queues offer a more efficient manner of adding and removing elements, at the cost of being unable to index an element. They are ideal for problems that require running maximums or minimums of groups of items that will not be removed. They are also comparable to other options when it comes to sorting when used in Heapsort or Dijkstra’s algorithm. Priority queues should not be used for data storage the requires repetitive insertion and deletion.

Works Cited

Java Documentation (2020, June 24). PriorityQueue. Retrieved August 11, 2020, from

https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

Skiena, S. S. (1998). *The algorithm design manual: Text* (Vol. 1). Springer Science & Business

Media.

van Emde Boas, P., Kaas, R., & Zijlstra, E. (1976). Design and implementation of an efficient

priority queue. *Mathematical systems theory*, *10*(1), 99-127.

Black (ed.), Paul E. (2004-12-14). Entry for *heap* in [*Dictionary of Algorithms and Data Structures*](https://en.wikipedia.org/wiki/Dictionary_of_Algorithms_and_Data_Structures). Online version. U.S. [National Institute of Standards and Technology](https://en.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology), 14

December 2004. Retrieved on 2017-10-08 from <https://xlinux.nist.gov/dads/HTML/heap.html>.

Ahuja, R. K., Mehlhorn, K., Orlin, J., & Tarjan, R. E. (1990). Faster algorithms for the shortest

path problem. *Journal of the ACM (JACM)*, *37*(2), 213-223.