COMP47500 – Advanced Data Structures in Java

Assignment No: 3
Date: 01 /04 /2023

Title: Implementing Priority Queue to handle orders at a Manufacturing Workstation

Code Link : https://github.com/COMP47500-Adv-Data-Structures-in-Java/Assignment_3

Execute: Main.java inside ie/ucd/csnl/comp47500

Assignment Type of Submission:			
Group	Yes/No	**List all of group members' details:**	**% Contribution** **Assignment Workload**
	Yes	Student Name : Prakash Jha Student ID : 2320077	25%
		Student Name: Satish Gaikwad Student ID: 23204142	25%
		Student Name: Chungman Lee Student ID: 23205535	25%
		Student Name: Abhijith Sathyendran Student ID: 23200124	25%

1. **Problem Domain Description:**

Imagine a manufacturing floor with various work stations, from clothing assembly to book printing. Tasks constantly arrive, each requiring a specific station and skilled worker. The challenge lies in efficiently managing this work queue, ensuring high-priority tasks are completed first while keeping workers engaged.

Prioritization is Key:
Not all tasks are created equal. Urgent orders or delicate steps in a larger process might require higher priority. Our system needs to effectively differentiate these tasks, pushing them to the forefront of the queue for quicker completion. Multiple factors can decide the priority of an order like premium subscription by customer etc.

Transparency is Essential:
Production managers and workers need real-time visibility into the work queue. Understanding what tasks are pending, their priorities, and assigned stations is crucial for informed decision-making and efficient workflow.

The Ideal Solution:
We envision a system that acts as a central hub for work management. It efficiently prioritizes tasks, considers worker expertise, and facilitates a balanced workload. Additionally, it provides a transparent view of the entire work queue, enabling everyone to stay informed and work collaboratively. By tackling these challenges, this system will contribute to smoother production processes, reduced delays, and a more optimized manufacturing environment.

Implementation:
We are considering 5 workstations that is "**clothing**", "**Card printing**", "**Photo making**", "**Mug building**", "**Pens printing**" which has constant influx of orders and the priority with which the orders are to be executed, the workers will process the orders and mark them as done after which they will be sent to ready to be delivered List. The processing of orders will be done parallel i.e we will use 5 different **Priority queue/sorted sequence/unsorted sequence**.

Key Features:

- **Prioritized Queue:**
  The system prioritizes tasks based on predefined levels (urgent, normal, low) or user-defined factors like customer subscriptions.
  This ensures high-priority tasks are completed first, addressing the challenge of efficiently managing the work queue.

- **Real-time Work Queue View:**
  Both production managers and workers have access to a real-time view of the work queue.
  This transparency allows everyone to see pending tasks, their priorities, and assigned stations, facilitating informed decision-making and efficient workflow.

- **Task Completion and Status Updates:**
  Workers can update task statuses as they progress, indicating "in progress" or "completed."
  This keeps everyone informed about task completion and allows managers to monitor progress.

2. Theoretical Foundations of the Data Structure(s) utilised

We aim to provide a solution to the above use case by adopting a priority queue in the following implementations.

I. SORTED SEQUENCE

Using a sorted list to implement a priority queue involves maintaining the elements in the list in sorted order according to their priority. When an element is added to the queue, it is inserted into the list in such a way that the order is preserved.

Implementation:

- Maintains a list of elements sorted based on priority (highest priority first). Insertion involves finding the correct position and inserting the element, potentially requiring a linear search (O(n)) in the worst case. Removal of the highest priority element is constant time (O(1)) as it's at the front.

Key Points

- Adding is slower (O(n)) because we might need to search the sequence to derive the exact location.
- Getting the higher priority element is extremely fast at (O(1)) because the element is right there at the front of the queue.

Analysis:

- **Addition**: O(n) in the worst case due to potential linear search for insertion.
- **Retrieval (Highest Priority)**: O(1) - fastest retrieval.
- **Practical Efficiency**: Not ideal for frequent insertions due to the overhead of maintaining sorted order.
- **Note:** Simple to understand and implement, guarantees retrieval of the highest priority element in constant time.

II. UNSORTED SEQUENCE

When an unsorted sequence is used to implement a priority queue, the elements are inserted on a first come first serve basis to the end of the queue.

Implementation:

- Elements are added to the end of the list in the order they arrive. Retrieval of the highest priority element requires iterating through the entire list (O(n)) to find the element with the highest priority. Removal of the highest priority element also involves searching and potentially shifting elements (O(n)).

Key Points

- Adding is super fast (O(1)) because the element just hops on the end of the line.
- Retrieving the most important element is slow (O(n)) because we might have to check each element to see who has the highest priority.

Analysis:

- **Addition**: O(1) - fastest addition as elements are appended.
- **Retrieval (Highest Priority)**: O(n) - slowest retrieval, requires iterating through all elements.

- **Practical Efficiency**: Suitable for scenarios with fewer retrievals and a focus on fast insertions.
- **Note :** Simple to implement, efficient for adding elements.

III. HEAP

Heap represents a binary tree data structure where the key at each node is greater than or equal to the keys of its children. It can be implemented using an array or linked list. We use an arraylist in our implementation.

Implementation:

- Utilizes a binary tree structure where each parent node has a higher or equal priority compared to its children. Insertion involves adding the element to the end and then "bubbling" it up the tree (upheap) to maintain the heap property (O(log n)). Retrieval of the highest priority element involves swapping it with the last element and then "sifting" down the tree (downheap) to maintain the heap property (O(log n)).

Key Points

- Adding an element to the list involves adding it at the end of the queue and then shifting it up the heap structure to maintain the heap property. This process is called **upheap**. This process has a growth rate of (O(log n)) .
- When the highest priority element is removed from the heap, it involves switching the root element(highest priority) with the last element in the heap, removing the last element, and then bringing down the element at root to maintain the heap structure. This process called **downheap** will have a complexity of O(log n).

Analysis:

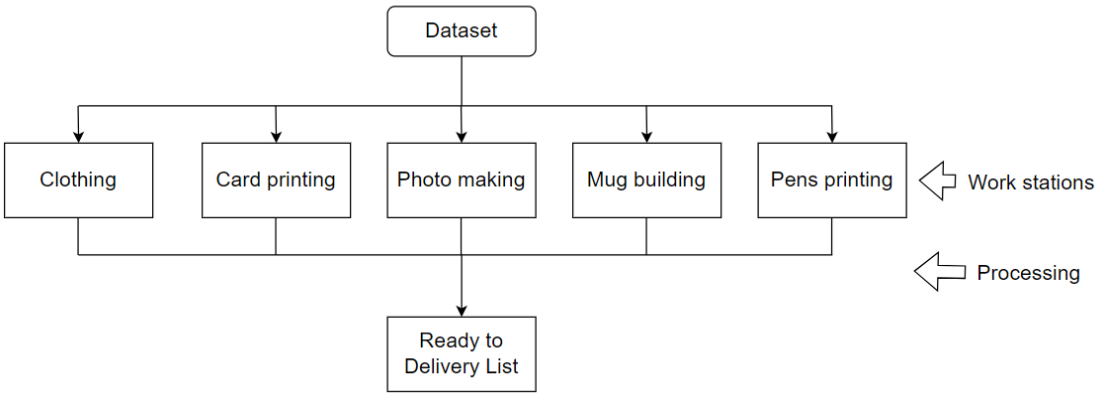
- **Addition**: O(log n) - efficient for adding elements with upheap operation.
- **Retrieval (Highest Priority**): O(log n) - efficient retrieval with downheap operation.
- **Practical Efficiency**: Ideal for scenarios with frequent insertions and retrievals due to its balanced performance.
- **Note :** Offers a balance between addition and retrieval efficiency with logarithmic complexity (O(log n)) for both operations.

COMPARISON OF OPERATIONS ACROSS DIFFERENT IMPLEMENTATIONS

OPERATION	SORTED SEQ	UNSORTED SEQ	HEAP
insert(key, value)	O(n)	O(1)	O(log n)
Retrieval min()	O(1)	O(n)	O(1)
removemin()	O(1)	O(n)	O(log n)
size()	O(1)	O(1)	O(1)

3. Methodology :

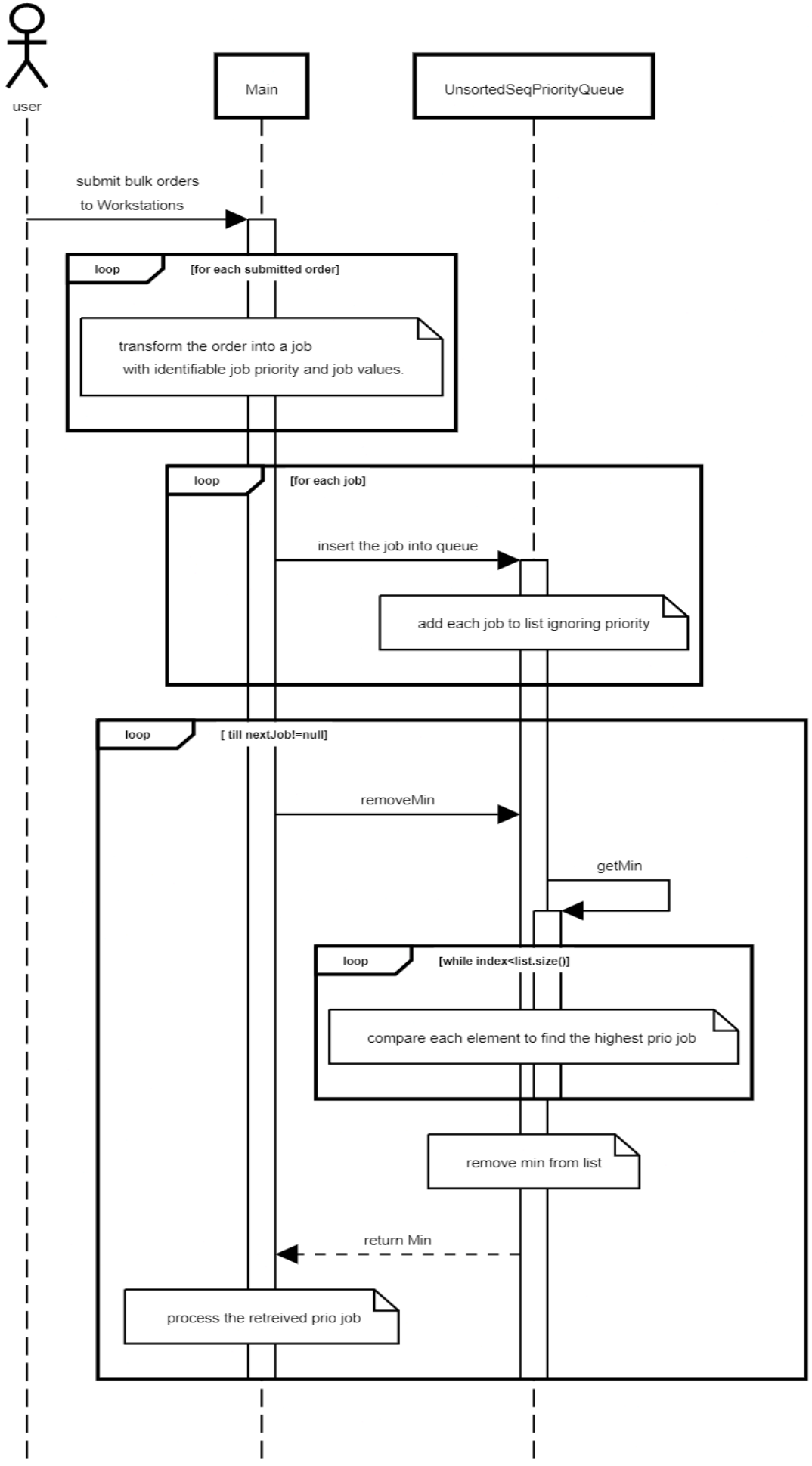
Below diagram showcases the code process, first we extract data from dataset and add the orders and their task to different workstation data structures **(Priority queue/sorted sequence/unsorted sequences)** based on their priority



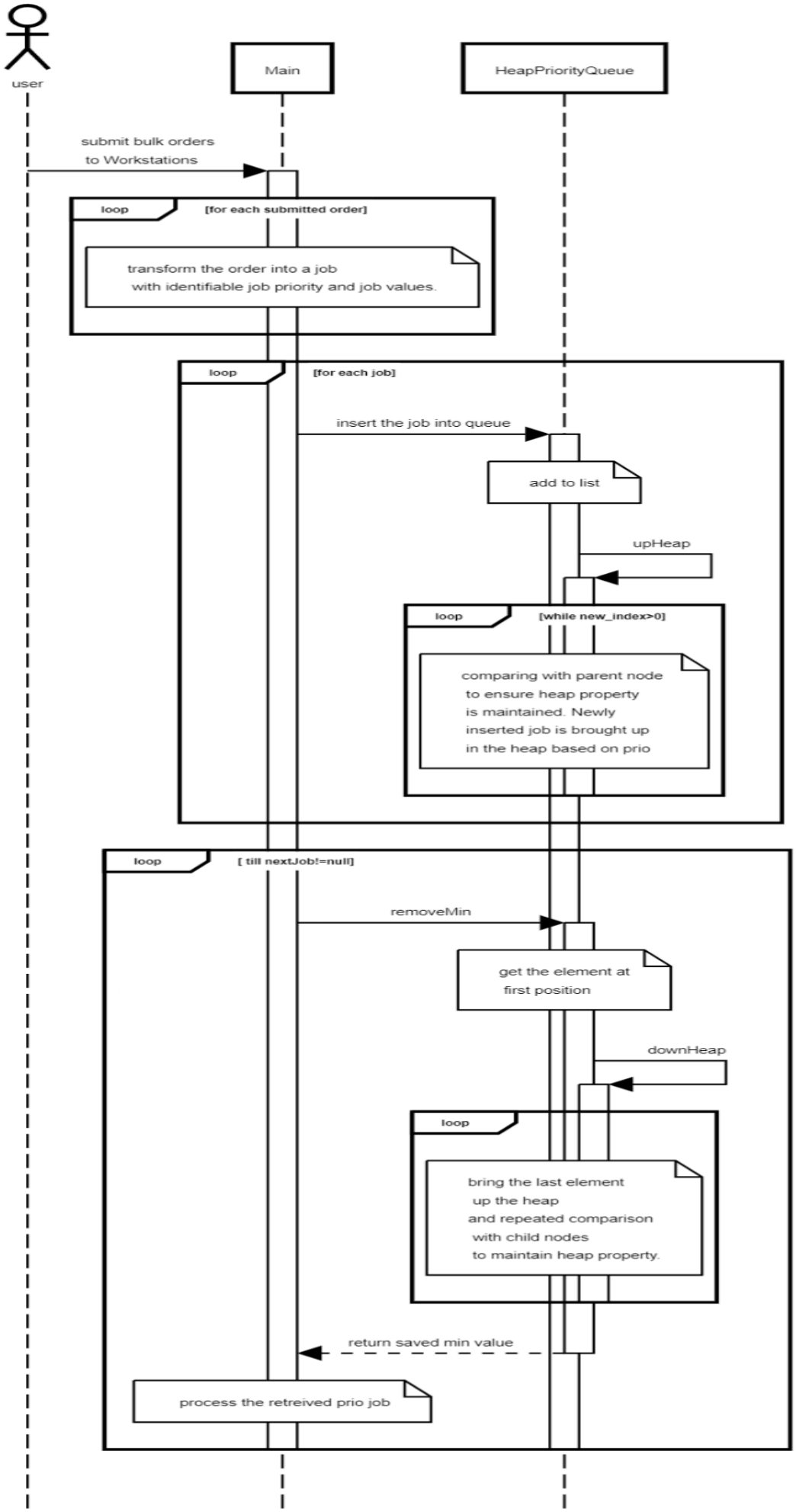
Analysis/Design (UML Diagram(s)):

fig1. Sequence Diagram for Unsorted Seq
fig2. Sequence Diagram for Heap
fig3. Sequence Diagram for Sorted Seq
fig4. Class Diagram.

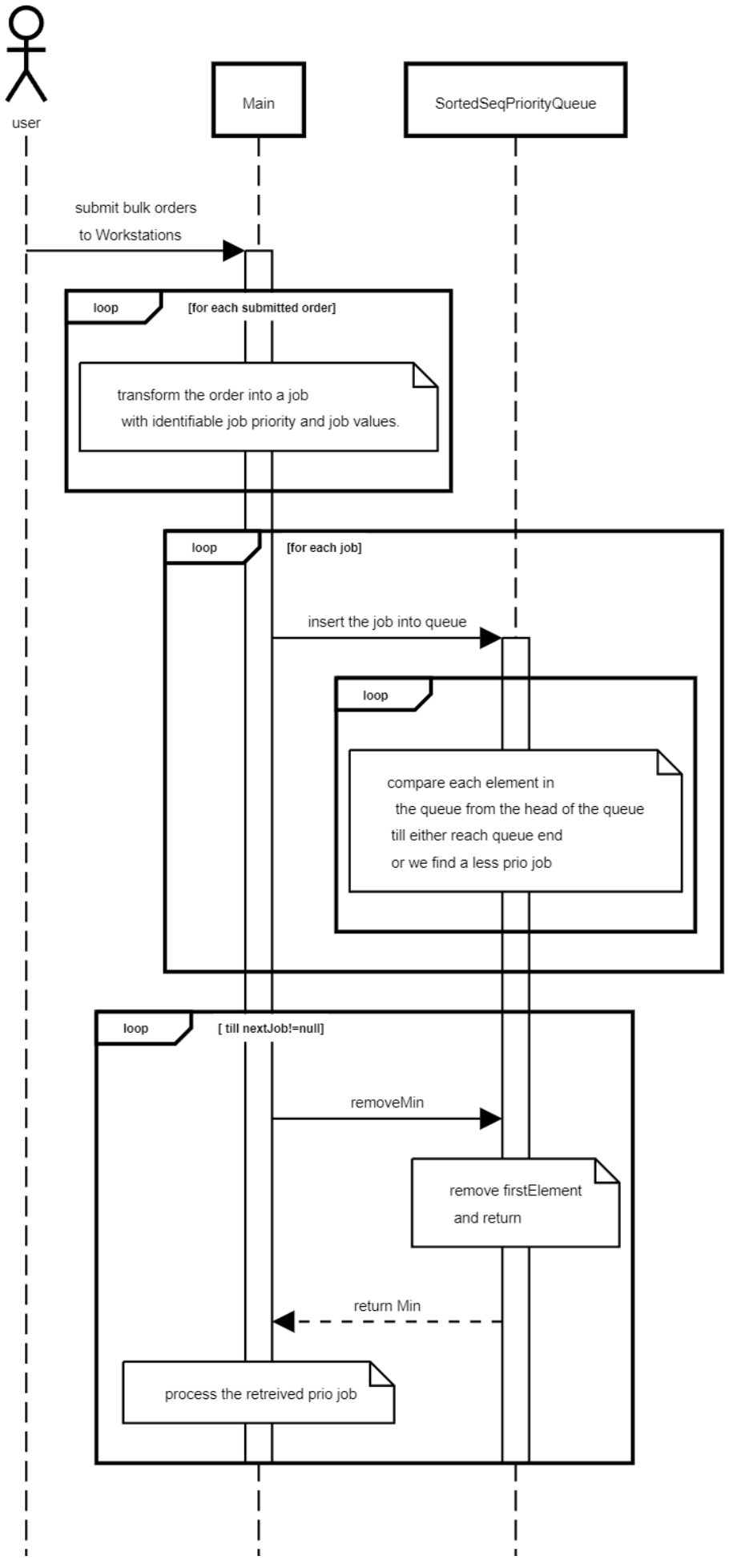
Priority Queue Unsorted Seq

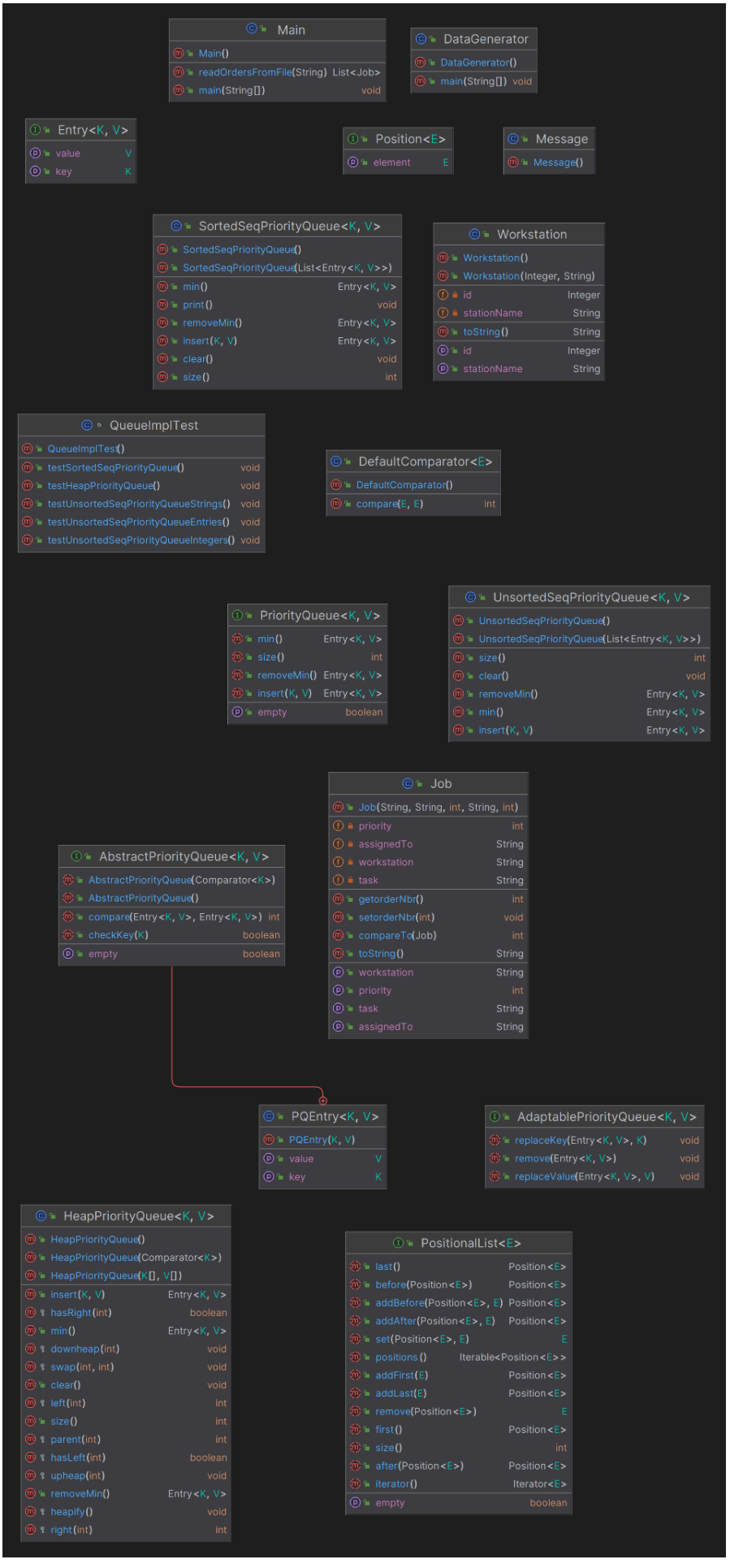


Priority Queue Using Heap



Priority Queue Sorted Seq

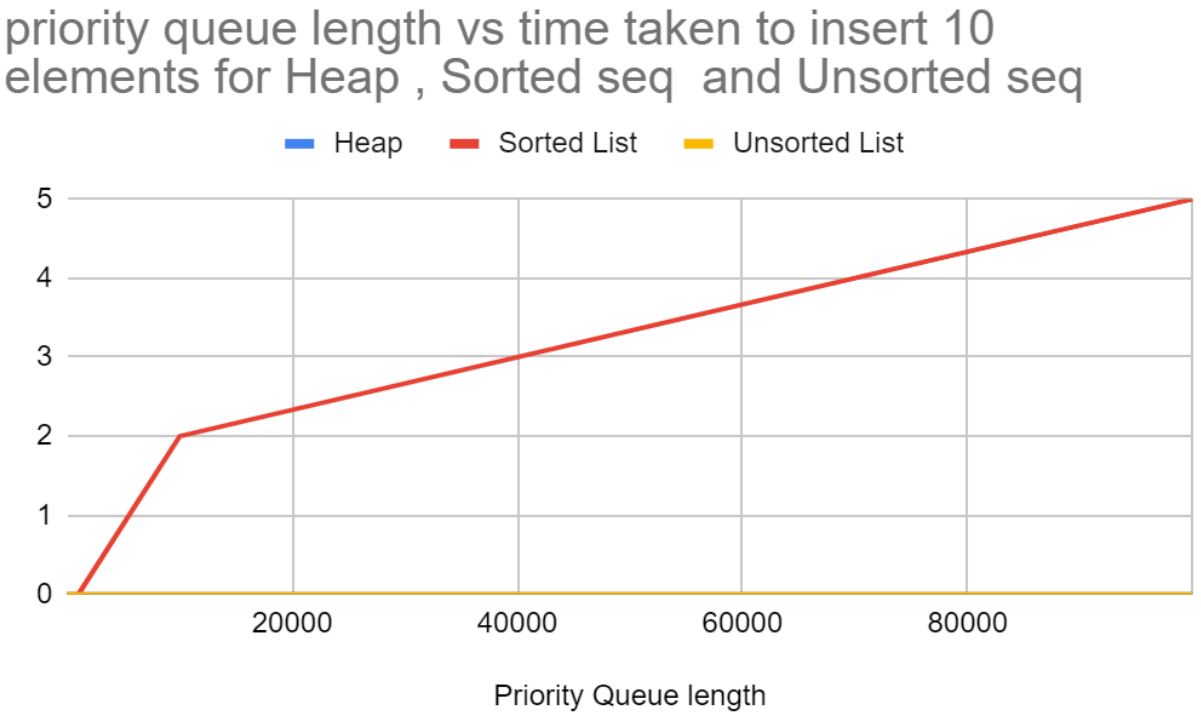



Main
Main()
readOrdersFromFile(String) List<Job>
main(String[]) void
DataGenerator
DataGenerator()
main(String[]) void
Entry<K, V>
value V
key K
Position<E>
element E
Message
Message()
SortedSeqPriorityQueue<K, V>
SortedSeqPriorityQueue()
SortedSeqPriorityQueue(List<Entry<K, V>>)
min() Entry<K, V>
print() void
removeMin() Entry<K, V>
insert(K, V) Entry<K, V>
clear() void
size() int
Workstation
Workstation()
Workstation(Integer, String)
id Integer
stationName String
toString() String
id Integer
stationName String
QueueImplTest
QueueImplTest()
testSortedSeqPriorityQueue() void
testHeapPriorityQueue() void
testUnsortedSeqPriorityQueueStrings() void
testUnsortedSeqPriorityQueueEntries() void
testUnsortedSeqPriorityQueueIntegers() void
DefaultComparator<E>
DefaultComparator()
compare(E, E) int
PriorityQueue<K, V>
min() Entry<K, V>
size() int
removeMin() Entry<K, V>
insert(K, V) Entry<K, V>
empty boolean
UnsortedSeqPriorityQueue<K, V>
UnsortedSeqPriorityQueue()
UnsortedSeqPriorityQueue(List<Entry<K, V>>)
size() int
clear() void
removeMin() Entry<K, V>
min() Entry<K, V>
insert(K, V) Entry<K, V>
Job
Job(String, String, int, String, int)
priority int
assignedTo String
workstation String
task String
getorderNbr() int
setorderNbr(int) void
compareTo(Job) int
toString() String
workstation String
priority int
task String
assignedTo String
AbstractPriorityQueue<K, V>
AbstractPriorityQueue(Comparator<K>)
AbstractPriorityQueue()
compare(Entry<K, V>, Entry<K, V>) int
checkKey(K) boolean
empty boolean
PQEntry<K, V>
PQEntry(K, V)
value V
key K
AdaptablePriorityQueue<K, V>
replaceKey(Entry<K, V>, K) void
remove(Entry<K, V>) void
replaceValue(Entry<K, V>, V) void
HeapPriorityQueue<K, V>
HeapPriorityQueue()
HeapPriorityQueue(Comparator<K>)
HeapPriorityQueue(K[], V[])
insert(K, V) Entry<K, V>
hasRight(int) boolean
min() Entry<K, V>
downheap(int) void
swap(int, int) void
clear() void
left(int) int
size() int
parent(int) int
hasLeft(int) boolean
upheap(int) void
removeMin() Entry<K, V>
heapify() void
right(int) int
PositionalList<E>
last() Position<E>
before(Position<E>) Position<E>
addBefore(Position<E>, E) Position<E>
addAfter(Position<E>, E) Position<E>
set(Position<E>, E) E
positions() Iterable<Position<E>>
addFirst(E) Position<E>
addLast(E) Position<E>
remove(Position<E>) E
first() Position<E>
size() int
after(Position<E>) Position<E>
iterator() Iterator<E>
empty boolean

4. set of experiments and results

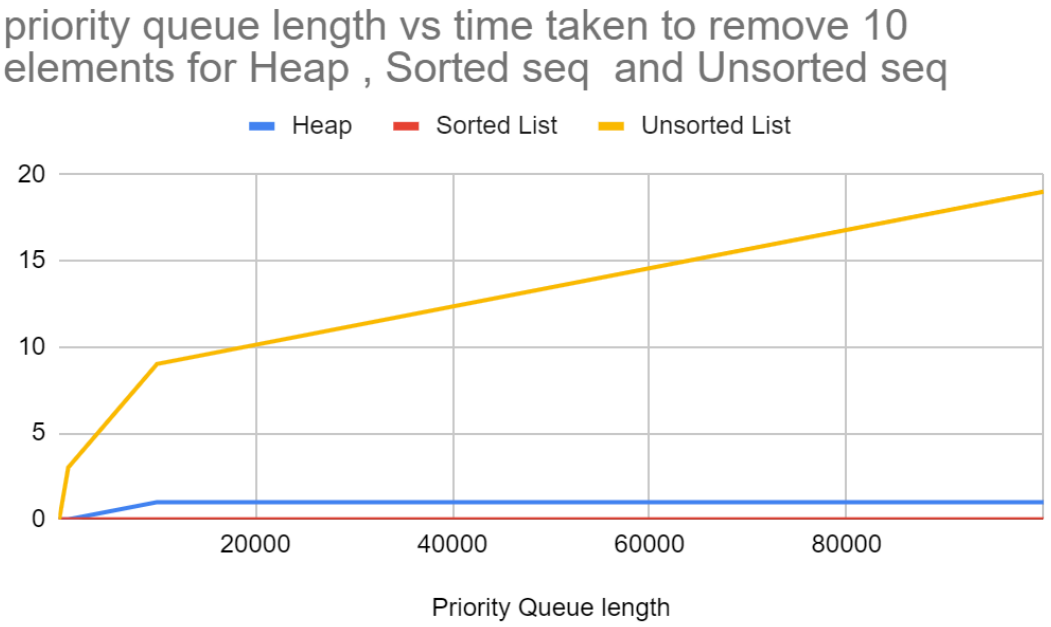
Priority Queue le	Heap	Sorted List	Unsorted List
10	7	5	4
100	7	2	1
1000	21	14	7
10000	53	156	35
100000	183	18995	179

Comments : as seen from above diagram the execution time increases linearly except for unsorted list where there is a noticeable difference



Priority Queue le	Heap	Sorted List	Unsorted List
10	0	0	0
100	0	0	0
1000	0	0	0
10000	0	2	0
100000	0	5	0

Comments : as seen from above diagram the execution time to insert 10 elements is almost 0 except for sorted sequence



Priority Queue le	Heap	Sorted List	Unsorted List
10	0	0	0
100	0	0	0
1000	0	0	3
10000	1	0	9
100000	1	0	19

Comments : as seen from above diagram the execution time to remove 10 elements is almost 0 except for unsorted sequence where it increases linearly

Execution Output:

```
--------------------Executing for Heaps-------------------------
Initial Size: 10
Size After Removal: 1
Min - Key: 866112 value: Job{workstation=clothing, task='Strategic Planning',
priority=866112, assignedTo='Tyler', orderNbr=600327}
Time taken to insert 10 random elements in 10 dataset 0ms
total execution Time taken for length 10 is 48ms

Initial Size: 100
Size After Removal: 91
Min - Key: 137146 value: Job{workstation=card printing, task='Research and Development',
priority=137146, assignedTo='Kerry', orderNbr=973188}
Time taken to insert 10 random elements in 100 dataset 0ms
total execution Time taken for length 100 is 2ms

Initial Size: 1000
Size After Removal: 991
Min - Key: 11258 value: Job{workstation=mug building, task='Customer Service',
priority=11258, assignedTo='Jordan', orderNbr=802168}
```

```
Time taken to insert 10 random elements in 1000 dataset 0ms
total execution Time taken for length 1000 is 7ms

Initial Size: 10000
Size After Removal: 9991
Min - Key: 733 value: Job{workstation=photo making, task='Environmental Compliance',
priority=733, assignedTo='Brett', orderNbr=130946}
Time taken to insert 10 random elements in 10000 dataset 0ms
total execution Time taken for length 10000 is 26ms

Initial Size: 100000
Size After Removal: 99991
Min - Key: 52 value: Job{workstation=mug building, task='Project Management',
priority=52, assignedTo='Brook', orderNbr=811230}
Time taken to insert 10 random elements in 100000 dataset 0ms
total execution Time taken for length 100000 is 106ms

--------------------Executing for SortedSeq -------------------------
Initial Size: 10
Size After Removal: 1
Min- Key: 866112 value: Job{workstation=clothing, task='Strategic Planning',
priority=866112, assignedTo='Tyler', orderNbr=600327}
Time taken to insert 10 random elements in 10 dataset 0ms
Time taken for length 10 is 3ms

Initial Size: 100
Size After Removal: 91
Min- Key: 137146 value: Job{workstation=card printing, task='Research and Development',
priority=137146, assignedTo='Kerry', orderNbr=973188}
Time taken to insert 10 random elements in 100 dataset 0ms
Time taken for length 100 is 1ms

Initial Size: 1000
Size After Removal: 991
Min- Key: 11258 value: Job{workstation=mug building, task='Customer Service',
priority=11258, assignedTo='Jordan', orderNbr=802168}
Time taken to insert 10 random elements in 1000 dataset 0ms
Time taken for length 1000 is 8ms

Initial Size: 10000
Size After Removal: 9991
Min- Key: 733 value: Job{workstation=photo making, task='Environmental Compliance',
priority=733, assignedTo='Brett', orderNbr=130946}
Time taken to insert 10 random elements in 10000 dataset 0ms
Time taken for length 10000 is 67ms

Initial Size: 100000
Size After Removal: 99991
Min- Key: 52 value: Job{workstation=mug building, task='Project Management', priority=52,
assignedTo='Brook', orderNbr=811230}
Time taken to insert 10 random elements in 100000 dataset 0ms
Time taken for length 100000 is 6058ms

--------------------Executing for UnsortedSeq-------------------------
Initial Size: 10
Size After Removal: 1
Min- Key: 866112 value: Job{workstation=clothing, task='Strategic Planning',
priority=866112, assignedTo='Tyler', orderNbr=600327}
Time taken to insert 10 random elements in 10 dataset 0ms
Time taken for length 10 is 2ms
```

```
Initial Size: 100
Size After Removal: 91
Min- Key: 137146 value: Job{workstation=card printing, task='Research and Development',
priority=137146, assignedTo='Kerry', orderNbr=973188}
Time taken to insert 10 random elements in 100 dataset 0ms
Time taken for length 100 is 0ms

Initial Size: 1000
Size After Removal: 991
Min- Key: 11258 value: Job{workstation=mug building, task='Customer Service',
priority=11258, assignedTo='Jordan', orderNbr=802168}
Time taken to insert 10 random elements in 1000 dataset 0ms
Time taken for length 1000 is 1ms

Initial Size: 10000
Size After Removal: 9991
Min- Key: 733 value: Job{workstation=photo making, task='Environmental Compliance',
priority=733, assignedTo='Brett', orderNbr=130946}
Time taken to insert 10 random elements in 10000 dataset 0ms
Time taken for length 10000 is 8ms

Initial Size: 100000
Size After Removal: 99991
Min- Key: 52 value: Job{workstation=mug building, task='Project Management', priority=52,
assignedTo='Brook', orderNbr=811230}
Time taken to insert 10 random elements in 100000 dataset 0ms
Time taken for length 100000 is 79ms
```

5. Code Implementation

GitHub (link): https://github.com/COMP47500-Adv-Data-Structures-in-Java/Assignment_3

6. Video of the Implementation running

Recording link: https://drive.google.com/file/d/1WB0nC6mgpe-6t1Td09prMaaEod9inco0/view?usp=sharing
Comments: uploaded to google drive