COMP47500
Advanced Data Structures in Java - Assignment 1

Queue

School of Computer Science
University College Dublin

Assignment 1

Date: 26 / 2 / 2024

Contributors

Assignment Type of Submission:			
Group	Yes	**List all group members' details:**	**% Contribution Assignment Workload**
		Student Name Jie Wang Student ID 23205138	33%
		Student Name Yuhong He Student ID 19326053	33%
		Student Name Zihao Yu Student ID 22212389	33%

Contents

1. Introduction

In the realm of computer science and software engineering, efficient data processing and resource management are fundamental pillars for the development of robust and scalable systems. One critical aspect of this endeavour lies in the design and implementation of data structures that can handle prioritized data and resource allocation effectively. Among these structures, the Priority Queue stands out as a cornerstone for managing elements based on their priority levels, enabling streamlined operations and enhanced system performance.

This report delves into the theoretical foundations, design principles, implementation details, experiments, results, and conclusions of a custom Priority Queue data structure and its utilization in a resource scheduling system. By understanding the theoretical underpinnings of Priority Queues, including their relationship with concepts like heaps and max-heaps, as well as their advantages and disadvantages, we lay the groundwork for the subsequent exploration of their practical applications.

Furthermore, the report elucidates the design and implementation specifics of the Priority Queue, along with its integration into a customized ResourceDispatcher class aimed at efficient resource management in operating systems. The ResourceDispatcher class, leveraging the Priority Queue, facilitates fair allocation of resources among competing threads, thereby enhancing overall system efficiency and responsiveness.

Through a series of experiments and performance evaluations, we aim to quantify the effectiveness and scalability of our Priority Queue implementation and ResourceDispatcher system. By analysing the experimental results, we seek to draw insights into the practical efficiency of our solutions and identify potential areas for optimization and future research.

In conclusion, this report presents a comprehensive exploration of the Priority Queue data structure, its theoretical foundations, practical implementations, and real-world applications in resource management systems.

2. Problem Description

Our project aims to introduce a custom Priority Queue data structure tailored to support efficient insertion, deletion, and maximum value retrieval operations. The selection of a Priority Queue stems from its ability to offer an average time complexity of $O(\lg n)$, making it a suitable choice for scenarios where prioritized data processing is essential.

Building upon this objective, we have developed a comprehensive operating system resource scheduling system utilizing our custom Priority Queue implementation. This system incorporates a semaphore parameter (signal), facilitating P (acquire) and V (release) operations essential for managing resource allocation effectively.[1]

In our implementation, the ResourceDispatcher class contains an instance of the PriorityQueue class, which it utilizes to manage threads awaiting resource allocation based on their priority levels. The semaphore parameter governs resource availability, ensuring efficient resource utilization and fair allocation among competing threads.

With scalability and flexibility in mind, our Priority Queue is designed to accommodate future expansion and integration with other data structures seamlessly. This versatility enables the systems to be applied across diverse industries, including finance, healthcare, and retail, where efficient data management and processing are paramount.

In summary, our project offers a robust solution for prioritized data processing and resource management, providing organizations with the tools necessary to streamline operations and enhance overall efficiency.

[1] A semaphore is a binary variable acting like a lock. When it is 1, the semaphore is free. When it is 0, the semaphore is busy. It controls accesses to critical sections that are large enough to warrant the overhead of pre-empting a thread and re-scheduling it later. If the semaphore is busy on a P operation, the thread is suspended and its descriptor is inserted in a waiting queue associated with the semaphore. Another thread is then activated and runs in its place. On a V operation, the waiting queue associated with the semaphore is looked up. If it is empty, the semaphore is set to free. Otherwise the highest priority thread is picked from the waiting queue and is activated. [1, pp. 396–397]

3. Theoretical Foundations

3.1. Queue

Queue ADT defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the front of the queue, and element insertion is restricted to the back of the sequence, which according to the first-in, first-out (FIFO) principle [2, p. 239]. Any list implementation is suitable for a queue, for example, it can be implemented using linked lists or arrays [3, p. 92]. Both the time complexity of queue operations enqueue and dequeue are $O(1)$ [4, p. 235].

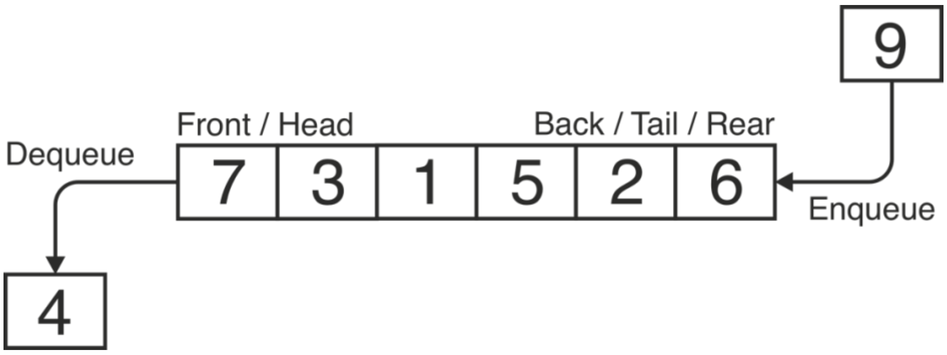


Figure: Queue Data Structure.

Source: Author

3.2. Heap & Max-Heap

The heap data structure is represented as an array object, which can be visualized as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is filled on all levels except possibly the lowest, which is filled from the left up to a point. Given an index i of a node, we can easily compute the indices of its parent ($i / 2$), left child ($2i$), and right child ($2i + 1$) [4, pp. 151–152].

A max-heap is a type of heap that satisfies the max-heap property, which means for every node i in an array A, it satisfies $A[Parent(i)] \geq A[i]$. The largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values less than or equal to the value contained at the node itself [4, pp. 152–153].

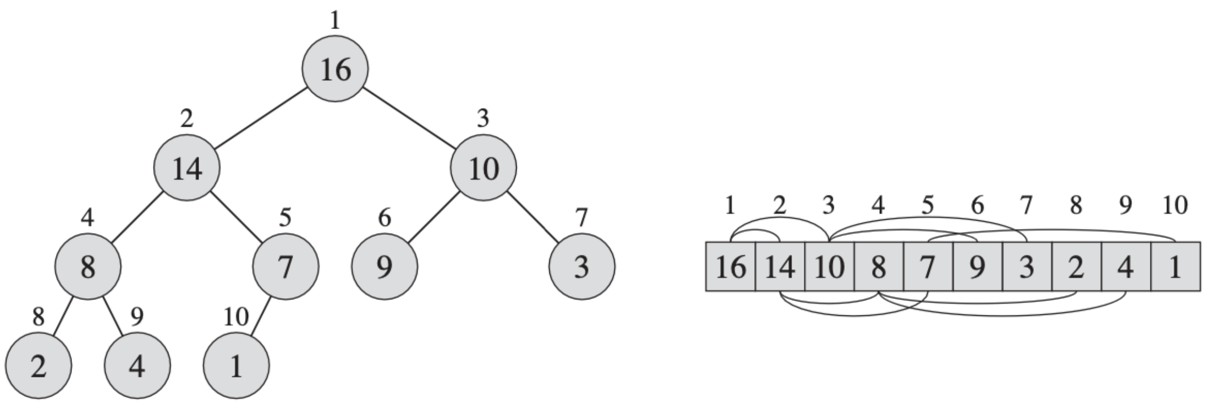


Figure: A max-heap viewed as a binary tree (left) and an array (right).

Source: [4, p. 152]

3.3. Priority Queue

Any data structure that supports the operation of search min (or max), insert, and delete min (or max, respectively) is called a Priority Queue [5, p. 91]. The operations of 'insert' and 'delete' are very similar to the 'enqueue' and 'dequeue' of a queue, the difference is that the Priority Queue ADT does not follow the FIFO approach satisfied by the Queue ADT, instead it always returns the highest-priority element from the current set of enqueued elements, regardless of when it was enqueued [6, p. 614]. There are several ways to implement the Priority Queues: Implicit Binary Heaps, Median Pointer Linked Lists, Skew Heaps, Calendar Queue, Henriksen's, Lazy Queue, SPEEDESQ, etc [7, p. 5]. The heap is the standard implementation of a Priority Queue [8, p. 126], which can support any priority-queue operation on a set of size n in $O(\lg n)$ time [4, p. 164].

3.3.1. Pros & Cons of Priority Queue

Advantages of Priority Queue: [9]

1. **Fast Access**: Elements in a Priority Queue are ordered by priority, allowing for quick retrieval of the highest priority element without searching through the entire queue.
2. **Dynamic Ordering**: Priority values of elements in a priority queue can be updated, enabling dynamic reordering of the queue as priorities change.
3. **Efficient Algorithms**: Priority Queues enhance the efficiency of algorithms like Dijkstra's algorithm for finding the shortest path and the A* search algorithm for pathfinding.
4. **Real-time Systems Integration**: Due to their ability to quickly retrieve the highest priority element, priority queues are commonly utilized in real-time systems where time sensitivity is critical.

Disadvantages of Priority Queue: [9]

1. **Complexity**: Priority Queues are more complex than basic data structures like arrays and linked lists, making them potentially more challenging to implement and manage.
2. **Memory Consumption**: Storing priority values for each element increases memory usage, which can be problematic in resource-constrained systems.
3. **Efficiency Limitations**: Alternative data structures like heaps or binary search trees may offer greater efficiency for specific operations, such as locating the minimum or maximum element.
4. **Predictability Concerns**: The order of elements in a priority queue is based on priority values, resulting in less predictable retrieval order compared to structures like stacks or queues, which follow FIFO or LIFO order.

3.3.2. Technical Issues on Implementation

There are several technical issues involving the implementation of the Priority Queue ADT in Java: [2, pp. 362–363]

- One challenge in implementing a Priority Queue is the need to maintain both an element and its associated key, even as elements are rearranged within the data structure. One solution to this problem is to use composition to combine a key (k) and a value (v) into a single object.
- When defining the Priority Queue ADT, any type of object can serve as a key, but it's essential to ensure that keys can be compared to each other in a meaningful way, and the comparisons must not be contradictory. To ensure that a comparison rule for any keys $k1$, $k2$, $k3$, denoted by $\geq$, is self-consistent, it must satisfy the properties of Comparability ($k1 \geq k2$ or $k2 \geq k1$), Antisymmetric (if $k1 \geq k2$ and $k2 \geq k1$, then $k1 = k2$), Transitive (if $k1 \geq k2$ and $k2 \geq k3$, then $k1 \geq k3$), and Reflexive ($k1 \geq k1$). In Java, there are two commonly used interfaces for performing comparisons: java.lang.Comparable and java.util.Comparator. Properly implementing one of these interfaces for the key can make comparisons meaningful and logical.

4. Design & Implementation

Java includes a class called java.util.PriorityQueue, which implements the java.util.Queue interface. Elements are added and removed based on priority, with the 'front' of the queue always holding the minimum element [2, p. 384]. In our project, we will design and implement our own PriorityQueue class, where the 'front' of the queue will always hold the maximum element, achieved using a max-heap structure. Subsequently, we will utilize this PriorityQueue to develop our ResourceDispatcher, which will include a custom Thread implementation.

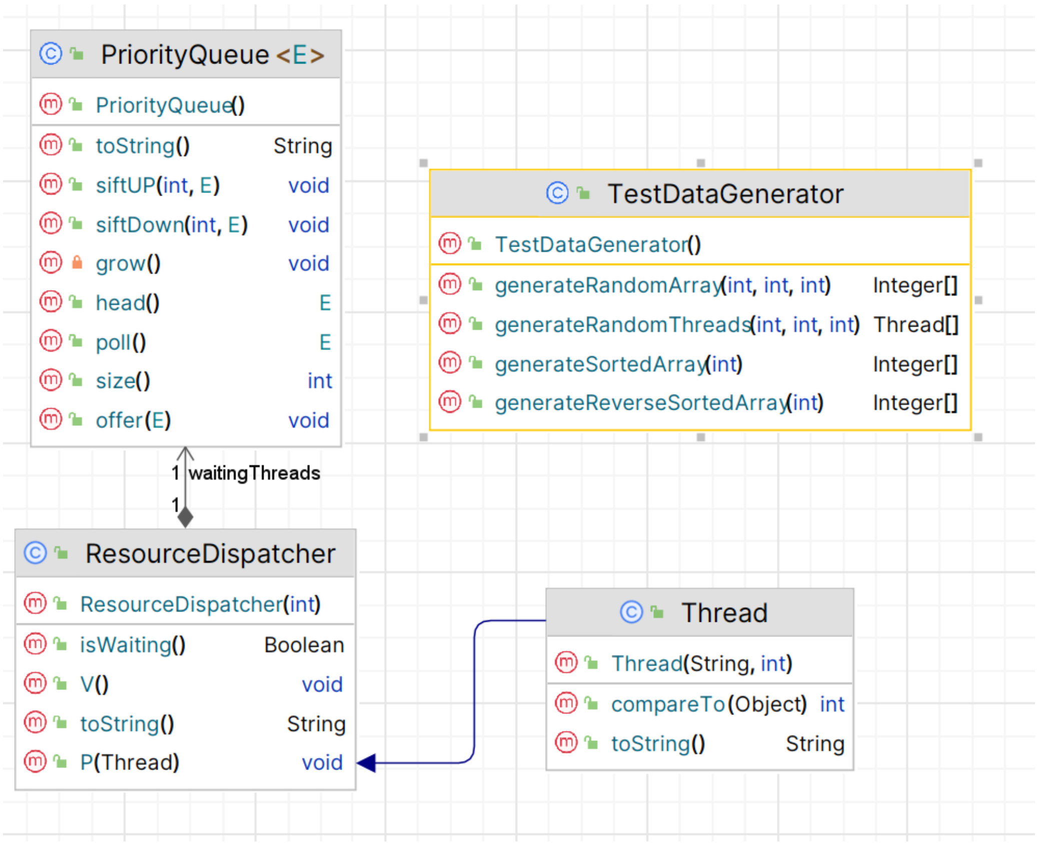


Figure: Project Class Diagram.

Hints: The GitHub repository and demo video link please refer to Appendix.

4.1. PriorityQueue

The PriorityQueue class offers a lightweight Priority Queue solution, leveraging binary heap operations for effective prioritization. It's simple interface makes it suitable for various applications where efficient handling of high-priority elements is essential. The following is a brief introduction to the methods in the PriorityQueue class:

- **size()**: Returns the current size of the Priority Queue.
- **offer(E element)**: Inserts an element into the Priority Queue.
- **poll()**: Retrieves and removes the highest-priority element from the queue.

- **head()**: Retrieves the highest-priority element without removal.
- **toString()**: Returns a string representation of the Priority Queue.
- **siftUp(int k, E x)**: Moves the element at index k up the heap to maintain the ordering.
- **siftDown(int k, E x)**: Moves the element at index k down the heap to maintain the ordering.
- **grow()**: Increases the capacity of the Priority Queue dynamically.

It is worth noting that methods of grow(), siftUp(), and siftDown() are the important internal implementation of the Priority Queue. The grow() method dynamically increases capacity following a growth strategy, while the siftUp() and siftDown() methods are responsible for maintaining the heap property during insertion and removal, respectively.

4.2. Thread

The Thread class implements the java.lang.Comparable interface to enable comparisons between Thread objects. Threads are prioritized based on the specified priority, with thread names utilized for further differentiation in case of a tie.

It contains two fields:

- **priority**: An integer representing the priority of the thread.
- **name**: A string representing the name of the thread.

The Constructor method Thread(String name, int priority) creates a Thread object with the specified name and priority.

Moreover, it contains two methods:

- **toString()**: Returns a string representation of the Thread object, including the thread name and priority.
- **compareTo(Object o)**: Compares two Thread objects based on priority and name. Returns a negative integer, zero, or a positive integer if the current thread is less than, equal to, or greater than the specified thread. This method ensures that the Priority Queue computes the size of the key in a meaningful and logical manner.

4.3. ResourceDispatcher

Internally, the ResourceDispatcher class employs a Priority Queue to effectively manage waiting threads according to their priority levels. This prioritization ensures that threads with higher priority gain access to resources in a timely manner. Additionally, the class implements atomic operations akin to the "P" (acquire) and "V" (release) operations for resource allocation and release, respectively. These operations ensure synchronized access to resources, preventing conflicts and race conditions among threads. Furthermore, the class handles resource availability through the signal field, which dynamically adjusts based on the number of available resources and the threads waiting for allocation. This signal mechanism facilitates efficient resource utilization and fair allocation among competing threads, enhancing the overall performance and effectiveness of the resource management system.

It contains two fields:

- **waitingThreads**: A Priority Queue that stores threads waiting for the resource, prioritized based on their assigned priority.
- **signal**: An integer representing the availability of the resource and functioning as a semaphore parameter. Positive values indicate the number of available resources, while negative values indicate the number of threads waiting for the resource, effectively representing the semaphore status.

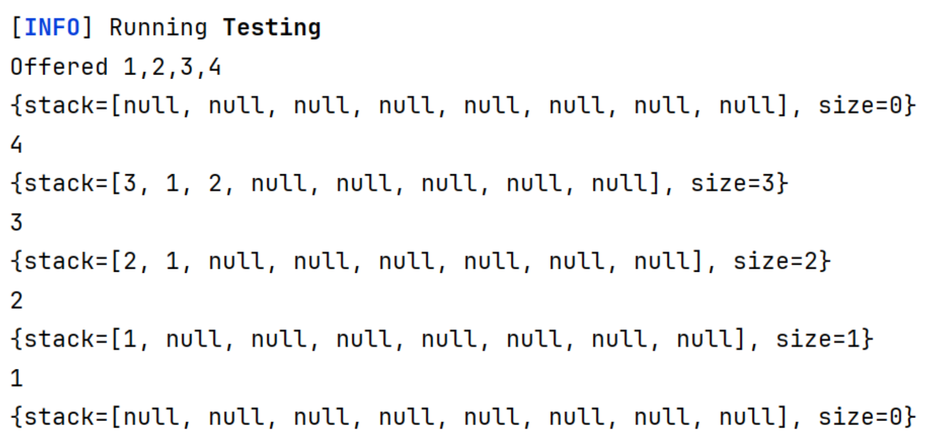
The Constructor method ResourceDispatcher(int s) initializes the ResourceDispatcher with an initial signal value s and an empty waiting thread queue.

Moreover, it contains four methods:

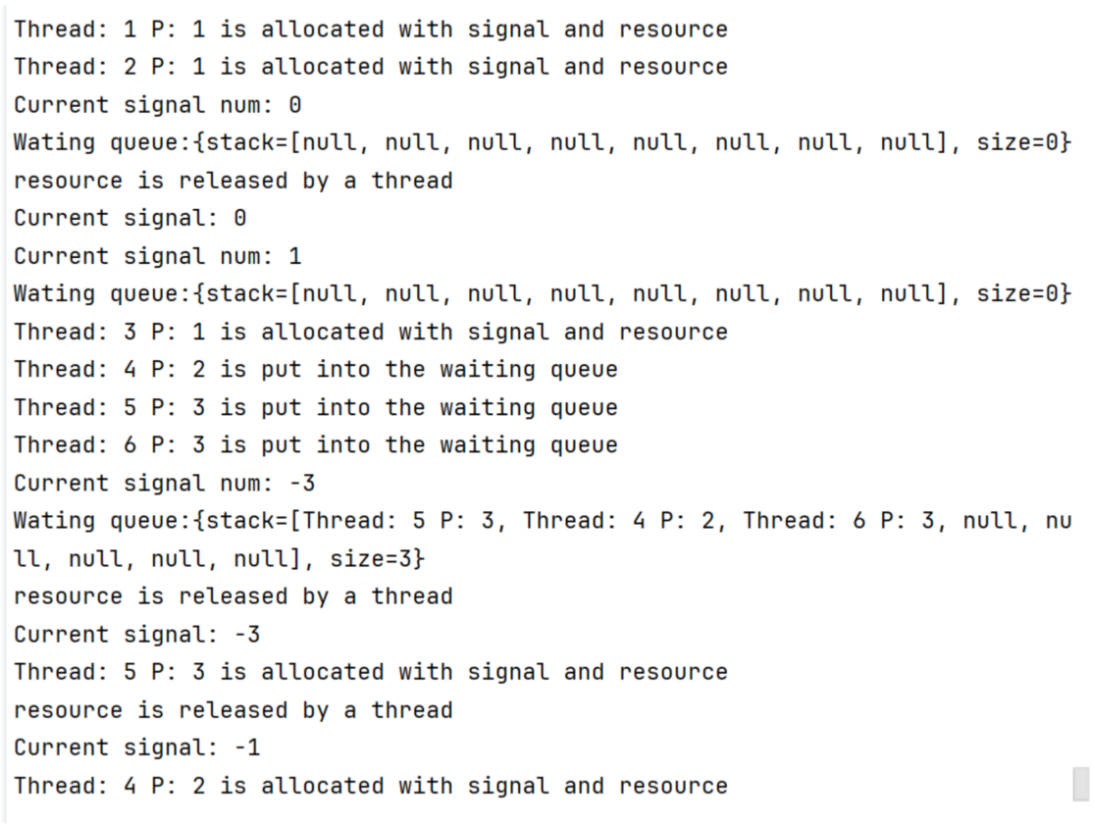
- **P(Thread thread)**: Represents the request from a thread to be allocated with a resource. If the signal is less than 0, the thread gets into the waiting queue. Otherwise, the thread gets the resource.
- **V()**: Represents the release of a resource by a thread. If the signal is less than or equal to 0, meaning there are threads waiting for the resource, allocate the resource to the highest-priority waiting thread.
- **isWaiting()**: Returns a Boolean indicating whether there are threads waiting for the resource.
- **toString()**: Returns a string representation of the ResourceDispatcher object, including the current signal value and the waiting queue.

4.4. Running Examples

Following are the console output of running example.



```
[INFO] Running Testing
Offered 1,2,3,4
{stack=[null, null, null, null, null, null, null, null], size=0}
4
{stack=[3, 1, 2, null, null, null, null, null], size=3}
3
{stack=[2, 1, null, null, null, null, null, null], size=2}
2
{stack=[1, null, null, null, null, null, null, null], size=1}
1
{stack=[null, null, null, null, null, null, null, null], size=0}
```



```
Thread: 1 P: 1 is allocated with signal and resource
Thread: 2 P: 1 is allocated with signal and resource
Current signal num: 0
Wating queue:{stack=[null, null, null, null, null, null, null, null], size=0}
resource is released by a thread
Current signal: 0
Current signal num: 1
Wating queue:{stack=[null, null, null, null, null, null, null, null], size=0}
Thread: 3 P: 1 is allocated with signal and resource
Thread: 4 P: 2 is put into the waiting queue
Thread: 5 P: 3 is put into the waiting queue
Thread: 6 P: 3 is put into the waiting queue
Current signal num: -3
Wating queue:{stack=[Thread: 5 P: 3, Thread: 4 P: 2, Thread: 6 P: 3, null, nu
ll, null, null, null], size=3}
resource is released by a thread
Current signal: -3
Thread: 5 P: 3 is allocated with signal and resource
resource is released by a thread
Current signal: -1
Thread: 4 P: 2 is allocated with signal and resource
```

Figure: Console output of running example.

5. Experiments & Results

Our experiments study and test the performance of the basic Priority Queue and our Resource Dispatcher. The following aspects were considered for this experiment:

- By controlling the initial capacity of the queue and keeping the input data as integer, we can keep the variants as size and reduce noise.
- The Priority Queue performance test controls the size of the input data and compares the average run time of each single operation. And analyse how their performance changes as the data size increases.
- The Resource Dispatcher performance test is to evaluate the performance of our implementation of the resource dispatcher.

5.1. PriorityQueue

This experiment evaluates the time complexity of the Priority Queue by measuring the time taken to offer() and poll() the elements with highest priority. Specifically, the experiments were completed by using different sized datasets and measuring the average time for each implementation.

In order to test the performance of the algorithm, three ranges of data were selected for the performance test of this experiment:

- The first group is data-size: 10.
- The second group is data-size: 100.
- The third group is data-size: 1000.
- The fourth group is data-size: 10000.
- The fifth group is data-size: 100000.

In the generation of experimental data using a random generation method, but also to ensure the randomness of the elements, so each set of data were five times randomly generated and tested and recorded the time used each time.

Following are the charts of the result of the test.

Data-size	10		100		1000		10000		100000	
Method	offer()	poll()	offer()	poll()	offer()	poll()	offer()	poll()	offer()	poll()
Test 1	3752300	23900	855800	337200	523700	786200	4413900	3246000	16811700	38896800
Test 2	24400	11200	168600	122700	355500	553500	3172000	3622600	8813700	48484800
Test 3	23300	14300	131500	111800	369900	465900	2184100	2694300	823900	38494700
Test 4	21300	11200	140300	194900	301100	510900	1087700	1689600	6889900	37448900
Test 5	39800	8600	129700	80000	519000	614500	1691100	2368800	7270700	41232300
average	772220	13840	285180	169320	413840	586200	2509760	2724260	9605000	40911500

Figure: Running times (unit: ns) of offer() and poll() operations under various data sizes.

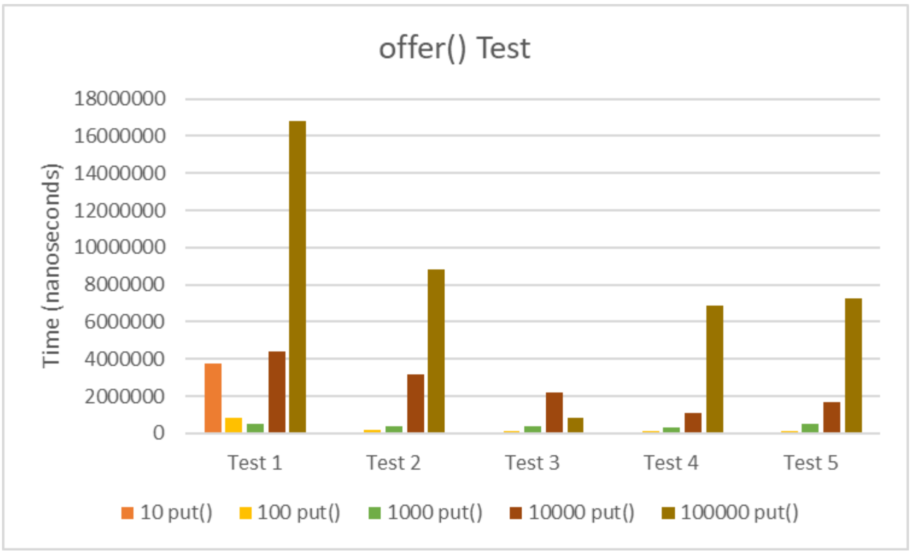


Figure: The comparison of the time consumption of offer() operations in each test under different data sizes.

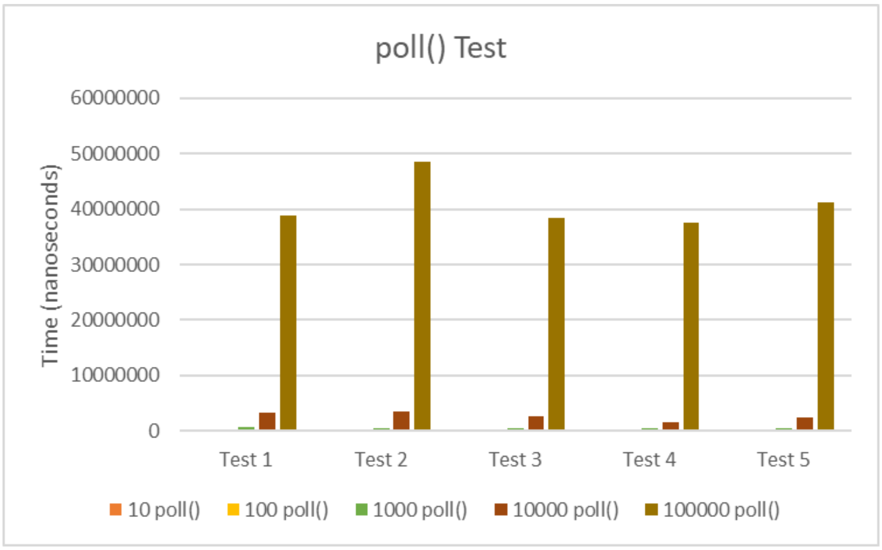


Figure: The comparison of the time consumption of poll() operations in each test under different data sizes.

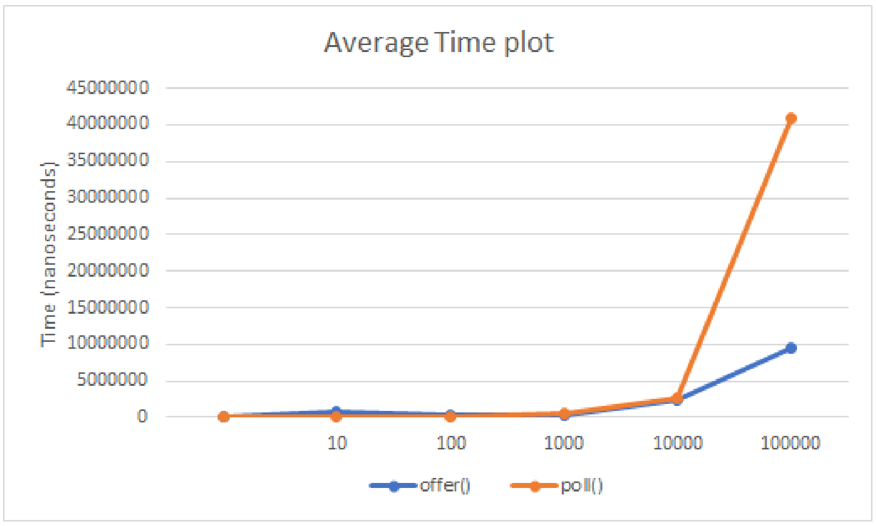


Figure: The trend of the time consumption for offer() and poll() operations as the data volume increases.

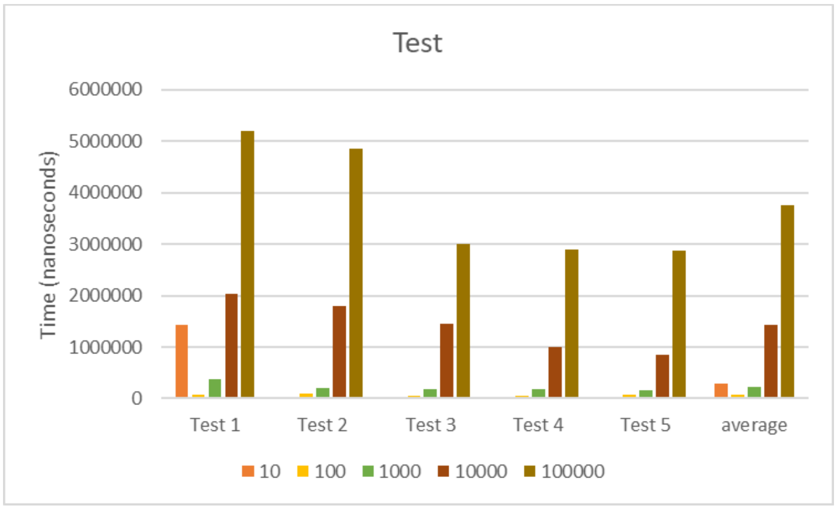


Figure: The comparison of the time consumption of poll() + offer() operations in each test under different data sizes, concluding with the average time consumption of the tests.

5.2. ResourceDispatcher

This experiment tested the performance of ResourceDispatcher for different sizes of five types of data models and obtained the respective times (nanoseconds).

```
+index---------Test Resource Dispatcher--time-------------+
| 1    average time for size: 10     is 1243400    nanoseconds|
| 2    average time for size: 100    is 4870300    nanoseconds|
| 3    average time for size: 1000   is 29444920   nanoseconds|
| 4    average time for size: 10000  is 199361060  nanoseconds|
| 5    average time for size: 100000 is 1933219460 nanoseconds|
+---------------------------------------------------------------+
```

Figure: The time consumption of PV operations in the ResourceDispatcher by different size of data volume.

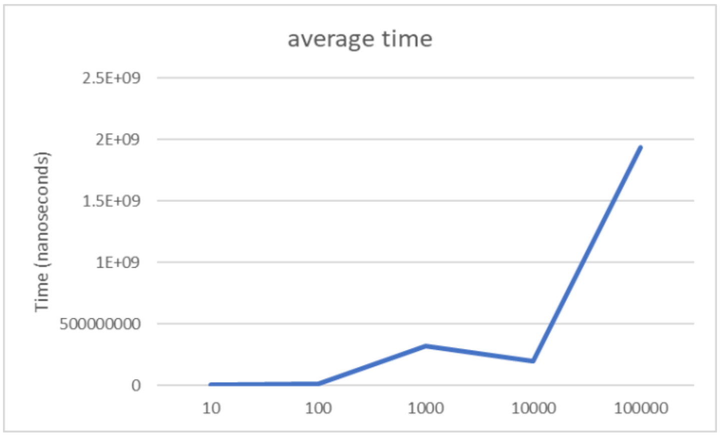


Figure: The trend of the time consumption for PV operations in the ResourceDispatcher as the data volume increases.

6. Conclusions & Future Work

6.1. Conclusions

Through our experiments conducted on each group member's respective laptop, we observed variations in response times for the same algorithm and data structure across different computers. This led us to speculate that the algorithm's performance is closely tied to the specific configurations of the computers used. In future endeavours, we aim to further investigate this correlation by conducting tests across a variety of machine configurations.

Moreover, our simulations utilized random data, which may have resulted in less precise resizing of data ranges and reference values. To address this limitation, future research could focus on refining the resizing function to ensure more rigorous testing methodologies.

Additionally, our analysis of Priority Queue experiments revealed notable differences in the average execution times between the first and subsequent executions of functions such as offer() and poll(). For instance, in the case of offer(), the initial test cases consistently exhibited longer execution times compared to subsequent tests across various input sizes. We hypothesize that this discrepancy may be attributed to the JVM's memory allocation mechanism and low-level system processes, which may incur longer durations during the initial memory allocation phase. Furthermore, there may be underlying cache mechanisms that warrant further exploration to better understand their impact on performance.

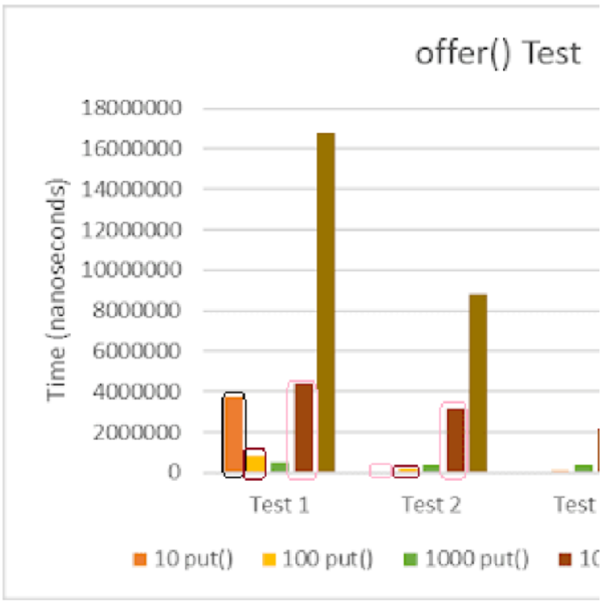


Figure: Comparison of time consumption on offer() operation test, the first test always exhibited the longest execution time compared to the subsequent tests.

6.2. Future Work

In future investigations, we plan to delve deeper into understanding the influence of computer configurations on algorithm performance by conducting tests on a wider range of machines with varying specifications. This will provide valuable insights into how different hardware configurations affect algorithmic efficiency and response times.

Furthermore, we intend to refine our testing methodologies by developing more robust resizing functions to ensure greater accuracy and reliability in data range adjustments. By optimizing the resizing process, we aim to enhance the precision of our performance testing procedures and obtain more accurate results.

Additionally, future research endeavours will involve exploring the underlying cache mechanisms and memory allocation processes to uncover their roles in determining algorithm performance. By gaining a deeper understanding of these underlying mechanisms, we can develop strategies to optimize algorithm execution and improve overall system efficiency.

References

[1] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge: Cambridge University Press, 2012.

[2] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed. Singapore: Wiley, 2015.

[3] M. A. Weiss, *Data Structures and Algorithm Analysis in Java*, 3rd ed. Upper Saddle River, N.J: Pearson, 2012.

[4] T. H. Cormen, Ed., *Introduction to Algorithms*, 3rd ed. Cambridge, Mass: MIT Press, 2009.

[5] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*. New York: Computer Science Press, 1998.

[6] N. Dale, D. T. Joyce, and C. Weems, *Object-oriented Data Structures Using Java*, 3rd ed. Sudbury, MA: Jones & Bartlett Learning, 2012.

[7] D. Wei, 'Comparative Study of Priority Queues Implementation', Oklahoma State University, 1999.

[8] J. Humpherys, T. J. Jarvis, and E. J. Evans, *Foundations of Applied Mathematics*. in Other Titles in Applied Mathematics, no. 152. Philadelphia: SIAM, Society for Industrial and Applied Mathematics, 2017.

[9] 'What is Priority Queue | Introduction to Priority Queue', GeeksforGeeks. Accessed: Feb. 26, 2024. [Online]. Available: https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

Appendix

Code Implementation Link

GitHub: https://github.com/Jay-Wang2000/COMP47500_assignments/tree/main/assignment1

Video of Implementation Running

- Zoom: https://ucd-ie.zoom.us/rec/share/k1hXh5D2L5r_SrXa-5kOpubBGoZeCppyT6QOceG9deMt5aOWziGdnNPFC8MTiA12.QueUhsPfLjQIn_dD
- Password: kTr+B7RW

This video briefly explains the main content of this project and shows the results of its implementation and experiments.