# CSCI203:Algorithms and Data Structures

# Assignment 2

Ivana Ozakovic, 4790339
Username: io447

## 1. A high-level description of the overall solution strategy

Since this assignment task involved creating discrete event simulation that involves queues and sorting elements based on some condition, initially I have observed the types of queues and efficiency of their standard algorithms, but also sorting algorithms in order to decide with data structures and operation to use.

I decided to use heap and circular queue as explained below.

**Common Data Structure Operations**

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|---|---|---|---|---|---|---|---|---|---|
| | Average | | | | Worst | | | | Worst |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Array | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| Stack | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Queue | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Singly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Doubly-Linked List | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| Skip List | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| Hash Table | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| Binary Search Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| Cartesian Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| B-Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Red-Black Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Splay Tree | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| AVL Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| KD Tree | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

Image reference: http://bigocheatsheet.com/

## Array Sorting Algorithms

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | θ(n+k) | O(n^2) | O(n) |
| Radix Sort | Ω(nk) | θ(nk) | O(nk) | O(n+k) |
| Counting Sort | Ω(n+k) | θ(n+k) | O(n+k) | O(k) |
| Cubesort | Ω(n) | θ(n log(n)) | O(n log(n)) | O(n) |

Image reference: http://bigocheatsheet.com/

## Circular Queue data structure (Customer processing)

For customers queue, the decision which data structure to use was fairly simple (FIFO queue as customers are served based on time they arrive), but to improve that decision, I have decided to use Circular Queue instead of regular Queue data structure.

In regular Queue we can insert elements only until queue becomes full, but in Circular Queue we can reuse the free space that was freed after dequeue operation.

Also, enqueue and dequeue operation Big-O complexity is O(1), which means it takes constant time.

## Heap (Servers and Events processing)

Since events are going to be processed based on priority of an event, depending on when customer arrived, and which server is used to serve the customer (based on server's efficiency), I have decided to implement heap as a priority queue for storing events.

As heap is implemented as a priority queue, each operation used (listed below) gives O(log n) performance. Furthermore, event queue is implemented as minimum heap, as events that happened before another have priority. First element will always give us next event that is supposed to be processed.

After implementation of heap for event priority queue, I have decided use heap for storing server indexes of of idle servers, as based on whether server is occupied or idle, we have decide which one is currently most efficient (fastest) free server that can serve the next customer.

I did not use Heapsort sorting algorithm as it gives O(n log n) performance, but instead I also used minimum heap for storing only idle server indexes, sorted by efficiency, as it is more efficient to add and delete server indexes on the heap when necessary, as each of those operations have O(log n) complexity. First element will also be the currently fastest idle server to use, which we can access in constant time when needed to find the most efficient currently idle server.

## 2. A list of all of the data structures used (reason for choosing explained in previous section)

1. Heap as priority queue for:
    - finding most efficient idle server (storing idle server's index only)
    - event priority queue
2. Circular queue for:
    - Customer queue

## 3. A list of any standard algorithms used (reasoning explained in section 1. )

1. Heap algorithms:
    - Insert
        - insert new event
        - insert new idle server index
    - Delete Minimum
        - remove next event to be processed
        - remove server index once server becomes occupied
    - Sift up - used in heap Insert algorithm
    - Sift down - used in heap Delete Minimum algorithm

2. Circular queue algorithms:
    - Enqueue - add customer to queue as they arrive
    - Dequeue - remove customer from queue once served

## References:


1. http://www.eecs.wsu.edu/~ananth/CptS223/Lectures/heaps.pdf
2. http://thejjjunk.ucoz.com/algorithms
3. http://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/