Time Performance of Different Queue Implementations

Alexander Alvarez

Fall 2024

Introduction

The time efficiency of adding and removing objects is to be measured on two different implementations of the queue data structure. The add and remove operations will be benchmarked and their implementations will be discussed.

The general behaviours of the queue data structure are described as well as how it was created and implemented for benchmarking.

Method

The benchmarks will be performed using 10 sizes `n` starting from 100 and doubling up to 51200 (`n =` {100, 200,..., 25400, 51200}).

The add (`enqueue`) and remove (`dequeue`) operations will be looped an amount (`loop`) of times. The `loop` variable is set to `1000` for the benchmarks. Due to consistency, only the minimum time out of `10` runs for each `n` will be measured.

The function for measuring the time runs the operation method in a `for` loop. A queue list is declared within the operation method in each iteration. The time is taken right before and after performing the operation for `n` times on said queue. The time then returns and accumulates in a variable `t` in the `for` loop which becomes the total runtime.

```
private static long queueBench(int n, int loop){
  long t = 0;
    for(int i = 0; i < loop; i++)
      t += operationBench(n, loop);
  return t;
}
```

The benchmark methods are run for a sufficient time before the actual benchmarks to enable the just-in-time compiler to make optimizations.

Queue

The queue data structure manages data in accordance with the first-in-first-out (FIFO) principle. In this repoort, the queue is implemented using linked lists. This implies that the first node added to a list is the first node to be accessed and removed.

The queue is initially created similarly to linked list by declaring a node with its data and reference variables. Focus lies mostly in the different `enqueue()` and `dequeue()` methods for the following two implementations.

First implementation

For the first implementation a "head" node (`queue`) is created and set to `null` in the constructor.

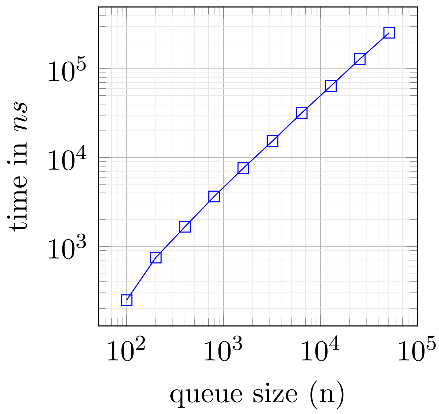
The `enqueue()` method creates a new node with its data being the `object` argument from the method parameter. The new node is made the new head node with the reference pointing to the previous head node.

```
public void enqueue(T object){
  this.queue = new Node(object, queue); // add to queue
}
```

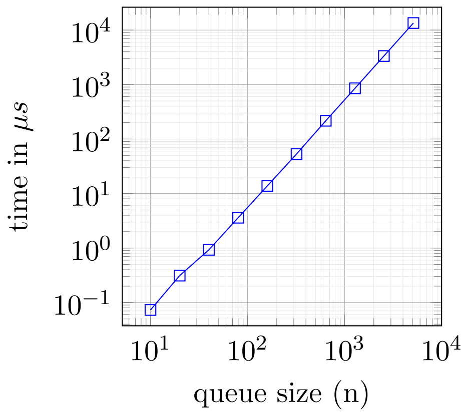
The `dequeue()` method first checks if the queue is empty. Two nodes is then created where one acts as the current node in the queue (`curr`) and the other acting as the previous node from the current (`prev`). The `curr` node loops to the first node added to the queue, that being the last node in the list. The head node is set to `null` if the queue only contained one node. Otherwise, the `prev` node removes the reference to the `curr` node from the queue. The data stored in the `curr` node is lastly returned.

```
public T dequeue(){
  if(this.queue == null) return null; // check empty
  Node prev = null; // previous from current
  Node curr = this.queue; // current node in queue
  while(curr.next != null){ // go to last node
      prev = curr;
      curr = curr.next;
  }
  if(prev == null) this.queue = null; // only one node
  else prev.next = null; // remove from queue
  return curr.object; // return data
}
```

Two benchmarks are run where the time to add and remove `n` objects to and from the queue is measured respectively. The benchmarks yields the following results displayed in figure 1.



(a) Time to add `n` objects



(b) Time to remove `n` objects

Figure 1: Benchmark for first implementation

The time to add `n` objects increases linearly as `n` grows. The time for one `enqueue()` operation is thus n/n which has the time complexity $O(1)$. This is due to the `enqueue()` method being able to add a node directly through the head node which is the same regardless of how big the queue is.

The time to remove `n` objects increases exponentially as `n` grows. One `dequeue()` operation has the time n^2/n which gives the time complexity $O(n)$. The reason is that the `dequeue()` operation loops to the end of the queue to remove a node which only increases in time as `n` grows. The queue size and amount of loops was decreased by a factor of 10 for this benchmark to save time.

Second implementation

The second implementation has two nodes created, `front` and `back`, which are set to `null` in the constructor. The purpose is to access the head node of the queue through the `front` node and to access the tail node of the queue through the `back` node.

The second `enqueue()` method, similarly to the first, creates a new node with its data being the `object` argument from the method parameter. However, this new node has a reference pointing to `null` and is stored in a variable (`add`) instead of directly in the head node. Both the `front` and `back` nodes is set to the new `add` node if the queue is empty. The `add` node is otherwise referenced to by the `back` node, thus adding it to the end of the queue. The `back` node is lastly set to the `add` node which makes `back` the node in the end of the queue.

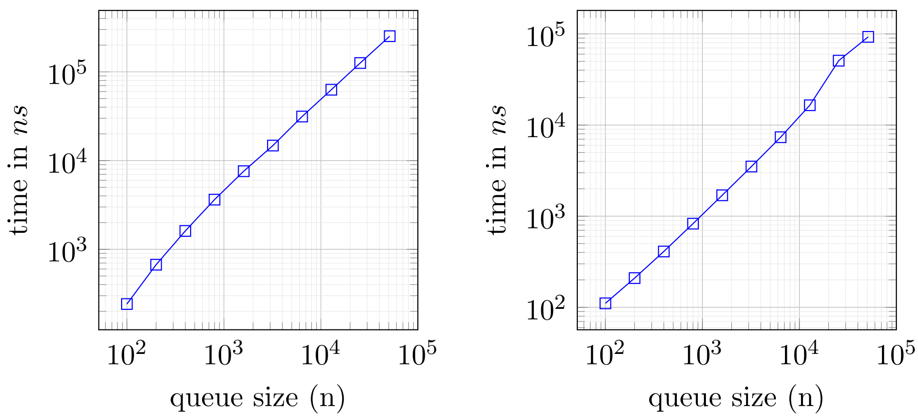
```
public void enqueue(T object){
  Node add = new Node(object, null); // creates new node
```

```java
    if(this.back == null){ // check empty
      this.front = add; // front is new node
      this.back = add; // back is new node
      return; // exit method
    }
    this.back.next = add; // add new node to end of queue
    this.back = add; // back is node at the end of queue
  }
```

The second `dequeue()` method first checks if the queue is empty. The data stored in the `front` node is saved in a variable (`obj`). The `front` node will then be the next node in the queue that first got added before returning the data in `obj`.

```java
  public T dequeue(){
    if(this.front == null) return null; // check empty
    T obj = this.front.object; // save data
    this.front = this.front.next; // front is the next node added
    return obj; // return data
  }
```

Two benchmarks are again run where the time to add and remove `n` objects to and from the queue is measured respectively. The benchmarks yields the following results displayed in figure 2.



(a) Time to add `n` objects

(b) Time to remove `n` objects

Figure 2: Benchmark for second implementation

The performance of the second `enqueue()` method is roughly the same as the first. In this case, the `enqueue()` method is able to add a node directly through the `back` node instead of the head. The result is nonetheless the same with the time complexity $O(1)$.

On the other hand, the second `dequeue()` method displayed a significantly greater performance than the first. Graph (b) in figure 2 shows a similar time increase as in (a) which gives the time complexity $O(1)$. The cause for this is the second `dequeue()` method's ability to remove nodes through the `front` node as opposed to looping to the end of the queue. Thus making both operations have the same time complexity.

Discussion

The first implementation has obvious drawbacks. One being the `dequeue()` method having to go through the whole list to remove a node which is the reason for its poor time performance. A secondary way to implement it would be to make it so that the `enqueue()` method loops to the end of the queue to add a node. This enables the `dequeue()` method to remove the oldest node through the head node but this also produces the same problem with the loop. Either way, the major flaw is that there is hardly any simple way to circumvent this problem with only one property that is the head node.

The second implementation is an improvement of the first. The major flaw of the first implementation is solved using two properties instead of one. These properties enables direct access to the front and back of the queue, allowing for instant insertion and removal. There are overall fewer compare operations as well which makes a small impact.

Advantages of the queue data structure are mainly the order preservation and performance. The queue data structure is best applied where efficient adding and removal of objects in an order following the FIFO principle is most needed. The disadvantages is mostly the limited functionality. There are other data structures that are better for cases where more functionalities and/or another order is required.