COPYRIGHT © 2021 Angelo David Indre, All rights reserved.

This document is intended exclusively for use by The University of Akron's Department of Computer Science. Do not distribute document copies to the public in any way.

THIS PROGRAM'S CONTENT IS PROTECTED  
The document and associated content are either the property of, or used with permission by, The University of Akron, Department of Computer Science and may be protected by copyright and other restrictions. Copyrights in the site content may also be owned by individuals and entities other than, and in addition to, The University of Akron, Department of Computer Science. The use of this document by you, or anyone else authorized by you, is prohibited, except for purposes of completing the homework/project. Any unauthorized use of the document may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

**Linked List Based Queue**Angelo Indre ([adi19@uakron.edu](mailto:adi19@uakron.edu))

**Abstract**

This research paper explores the capabilities of my Abstract Data Type MyQueue, which is a linked list-based queue. It employs FIFO protocol and is implemented as a chain of nodes each containing an integer, and a pointer to the next node on the list. The list can be operated on via a head pointer and a tail pointer. Its methods include enqueue, dequeue, clear, empty, display, size, front, and back. It also of course has a user written default constructor and destructor.

**Introduction**

A queue’s main feature is a First In, First Out (FIFO) strategy for adding and removing data from a list of data entries. New values are added to the end of the queue, and old ones are removed from the front. Queues are not built for insertion or deletion from anywhere besides these positions. Considering this restriction of a Queue’s functionality, why would we want to implement it as a linked list—which can only outperform an array-based structure on these methods insert, delete, and find?

**Discussion**

Hypothesis: MyQueue will append any number of elements faster than the STL’s linked list because std::list is doubly linked and therefore every append will have one more assignment that wires the last element pointer to the previous element. Vector will out-perform both on data sets of any size as it is declared on the stack and assigning data that is already allocated is much faster than going to the heap over and over again for new nodes.

The very first trial for this test records the time it takes to fill each data structure with 50,000 values, restart, and then double the load 5 times over. For the queue and the list, this can be executed in Θ(1) time. As per the U1 ds04 Lists Stacks Queues power point slides. Also, because there are no loops in either of their append functions. They both rely on a tail pointer. Vector can perform in O(1) time, but if there is not enough space and a reallocation is required, it will perform in O(n) since every entry will need re-entered. But I anticipate Vector will still be faster because there is no need for next pointers at every entry and there is not an allocation at every call to push\_back as well.

The first trial testing MyQueue was to measure the time it takes to enqueue varying numbers of elements to the queue and compare it to doing a similar operation for the two standard template containers I chose to compare against. Each data structure has the index number of each element stored as an integer value so that they all contain the same data. They all contain the same number of elements each trial as well so that the only difference between columns is which data type is being used to store values. The table below display’s the measured completion times from the testing program.

|  |  |  |  |
| --- | --- | --- | --- |
| # of elements in  10,000’s | Enqueue  (Queue) | Push  (List) | Push\_back  (Vector) |
| 5 | 0.004s. | 0.004s. | 0.001s. |
| 10 | 0.006s. | 0.007s. | 0.001s. |
| 20 | 0.010s. | 0.015s. | 0.003s. |
| 40 | 0.020s. | 0.030s. | 0.004s. |
| 80 | 0.047s. | 0.061s. | 0.009s. |
| 160 | 0.099s. | 0.121s. | 0.017s. |

As anticipated, the List and My Queue’s Push\_back and Enqueue functions performed similarly and much slower than the Vector’s Push\_back function. The magnitude of difference between Vector and MyQueue is intense. If you wanted to be able to append values to a structure and do nothing else, Vector will do it faster than either of the other two structures.

These times were recorded on my personal laptop, a Microsoft surface pro 3 running Windows 10. It has an i5 clocked in at 1197MHz core speed. 64-bit operating system, 8 GB of ram installed.

Next up, I wanted to see if maybe the performance differences would be any different if I changed what data type I populated the structures with, but templating my class proved to be difficult, so I abandoned that idea. So now I must think of more attributes my Queue has that I want to compare to the other two structures from the STL.

Next up, I tried comparing my queue to the std::queue as I wanted to see how my performance compared to that of the STL’s. The data below shows that my queue performed appends much slower than the STL’s.

|  |  |  |
| --- | --- | --- |
| # of elements in  10,000’s | Enqueue  (MyQueue) | Push  (Queue) |
| 5 | 0.004s. | 0.001s. |
| 10 | 0.006s. | 0.001s. |
| 20 | 0.010s. | 0.003s. |
| 40 | 0.020s. | 0.005s. |
| 80 | 0.047s. | 0.010s. |
| 160 | 0.099s. | 0.024s. |

According to cplusplus.com, the STL’s queue is a “container adapter” which works like a std::deque by default, which is our case here. The std::deque is array-based instead of linked list based. This difference explains why the std::queue performed more similarly to the vector rather than its home-brewed linked list counterpart MyQueue.

**Optimization**

At this point in my paper, I’m looking at my ADT and thinking that it really isn’t very applicable. You don’t get to exercise the benefits of easy removal and insertion in the middle of your structure like you would using a regular linked list because the queue follows FIFO protocol. Nor do you get the benefit of faster pre-allocation like you would if this queue were implemented array-based. So where does the linked list based queue fit in? What is the tradeoff we took? Where does it outperform these other data types?

**Discussion**

In researching my queue, I also wanted to change a different variable to see how it would affect the runtimes of the append methods under different case loads. The data in the tables above was acquired by running my program in the IDE CodeBlocks. I wanted to work on space complexity (which is coming up later in the paper) so I tried taking the advice of the project artifacts and instead of using PuTTY to connect to the university of Akron’s Linux servers, I used Windows Subsystem for Linux (WSL) and installed Valgrind in order to measure the memory used by each different data structure. But once I had my project properly put together for a linux distribution, I ran it and noticed differences in my runtimes which are recorded below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # of elements in  10,000’s | Enqueue  (MyQueue) | Push  (List) | Push\_back  (Vector) | Push  (std::queue) |
| 5 | 0.003s. | 0.003s. | 0.012s. | 0.011s. |
| 10 | 0.004s. | 0.005s. | 0.001s. | 0.002s. |
| 20 | 0.008s. | 0.011s. | 0.002s. | 0.003s. |
| 40 | 0.015s. | 0.022s. | 0.005s. | 0.006s. |
| 80 | 0.028s. | 0.045s. | 0.010s. | 0.012s. |
| 160 | 0.034s. | 0.089s. | 0.026s. | 0.025s. |

Interestingly, linked list based structures performed faster on WSL than they did in CodeBlocks while array based structures performed slower on WSL than in CodeBlocks. At the highest load of 1,600,000 elements, the disparity in runtime between my queue and the std::queue went down from .075s to .009s. So if you’re working on WSL instead of codeblocks, the opportunity cost of a linked list based queue is much lower.

Also, for some reason at 50,000 elements, both the array based data structures had an outlier value .010 seconds above what I was expecting. I would attribute this to needing to resize many times when appending values 0-50,000. The vector’s capacity doubling strategy would reallocate at 4, 8, 16, 32, 64… and so on for a grand total of log(base 2)(50,000) ~16 reallocations. Since no size is specified at the declaration. From then on, you’d think that each size increase would only need one reallocation since the size is doubling, but that is not the case because my testing code is built to declare a new vector or std::queue for each different case load. So what explains the unusually high first entry in WSL?

Truthfully, it seems like one should choose either a linked list or an array-based queue. If you need FIFO, it makes sense to go with the array-based queue because it is faster than the linked-list based. But if for your task it made sense to use FIFO protocol, choose a queue over a doubly linked list for faster runtimes. But leading into my next question to be answered: Why would you ever choose a linked list-based queue over the STL’s array-based one?

Hypothesis: The linked list-based queue has lower/more efficient space complexity than the array-based queue.

Below is the code that I used to test allocation for the case of 50,000 elements in type my queue. For the data I have below this code, I manipulated the number of elements, and the data type to see how memory allocation changed when these variables were manipulated. Below that is the output from 8 different trials.

int main()

{

    MyQueue quesef;

    for (int i = 0; i < 50,000; i++)

    {

        quesef.enqueue(i);

    }

    return 0;

}

872,704 bytes : 50,000 allocs |for 50,000 elements in my queue.  
8,072,704 bytes : 500,000 allocs |for 500,000 elements in my queue.

282,944 bytes : 399 allocs |for 50,000 elements in std::queue.  
2,236,672 bytes : 3,919 allocs |for 500,000 elements in std::queue.

1,272,704 bytes : 50,000 allocs |for 50,000 elements in std::list.  
12, 072,704 bytes : 500,000 allocs |for 500,000 elements in std::list.

596,998 bytes : 18 allocs |for 50,000 elements in std::vector.  
4,247,004 bytes : 21 allocs |for 500,000 elements in std::vector.

From the statistics above, you can see that in order from least to most bytes used, we have vector, std::queue, MyQueue, and list. It is also notable that an increase in magnitude for vector capacity only required 3 extra allocations. This output came from the console program called Valgrind’s memcheck tool. My testing programs were run on my personal laptop’s installed WSL.

So the question we wanted to answer was “Why should we use my linked list queue over the STL’s array-based queue?” and the answer is definitely not “Because mine takes up less space.” It actually takes up 4 times as many bytes as the STL’s queue and almost twice as much as the STL’s vector. It is, however, 2/3 as expensive as the STL’s list.

These stats are in the same order as the speed at which each container’s appending function operates. This wouldn’t have been my guess as to what the results were because of the trend that faster programs use more memory. I suppose that just because the number of bytes are lower does not necessarily mean that the container is the cheaper option because array based implementation requires consecutive chunks of memory to keep your data together unlike a linked list.

So the million dollar question still stands: Why would you use a linked list queue over an array-based queue? There is one more trial showing how long it takes to wind and unwind our containers. It is similar to the first trial, except this time, we will time how long it takes to append our values to the back, and pop them off the front.

|  |  |  |  |
| --- | --- | --- | --- |
| # of elements in  10,000’s | Dequeue  (MyQueue) | Pop\_Front  (List) | Pop  (std::queue) |
| 5 | 0.006s. | 0.004s. | 0.001s. |
| 10 | 0.011s. | 0.008s. | 0.001s. |
| 20 | 0.022s. | 0.018s. | 0.003s. |
| 40 | 0.025s. | 0.035s. | 0.006s. |
| 80 | 0.044s. | 0.086s. | 0.014s. |
| 160 | 0.089s. | 0.137s. | 0.033s. |

All of the output from the trial above matches that of the first trial, but scaled a little bit higher to account for the extra operations of popping all the values out of the structure. In short, there wasn’t any new information from this trial, except now we know that dequeuing values is much faster than enqueuing them.

Notice that std::vector is not included in the table above. This is because vector does not have a pop\_front method. Another piece of information that is relevant to the current trial and our big question is that of how an array based queue deals with popping values and running out of storage. The linked list based queue cleans up after itself deleting each node as it is popped off. The array based does not do that.

**Array Based**

Empty Queue has space allocated  
[ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] ->

Some Enqueuing  
[o]-> [o]-> [o]-> [o]-> [o]-> [o]-> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [ ] ->

Some Enqueuing and Dequeuing  
[ ] -> [ ] -> [ ] -> [ ] -> [ ] -> [o]-> [o]-> [o]-> [o]-> [ ]-> [ ] -> [ ] -> [ ] ->

**Linked List Based**

Empty Queue has no space allocated  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Some Enqueuing  
[o]-> [o]-> [o]-> [o]-> [o]-> [o]-> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Some Enqueuing and Dequeuing  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ [o]-> [o]-> [o]-> [o]-> \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The above text/graphic displays how enqueuing and dequeuing affects the entire list for array based and linked list based implementation. This map highlights some of the biggest advantages of linked list based queue over array based.

Array based has limited space as we know, which requires reallocation if our queue is too big. But going to capacity is the only way to reallocate the queue. Hypothetically, if you queued 1000 items to work with, and worked with 990 of them, those who were dequeued would no longer be in the queue, but their space would remain allocated. Your program would have space for 1000 items and only be holding 10. There are options to amend this drawback like having a drifting queue or a circular queue in which as your queue inches along its allocated memory, it is repositioned at the front upon reallocation, or just wraps around back to the front.

With our linked list based queue, all these worries about how much space is allocated, cleanup, and where your data is located are non-existent. The graphic above does not properly display the arrangement of a linked list. The physical location of your data is arbitrary. The entries do not have to be in line with each other like they are in the array based queue.

**Conclusion**

The Linked list based queue has decisive advantages over the standard vector and list. It can exercise FIFO faster than the standard list while taking up less memory. While the standard Vector is incapable of FIFO altogether.

The main competitor against my homebrewed queue is the STL’s array based queue. The STL’s queue outperforms mine on push and pop operations on both WSL and in CodeBlocks. But I would like to note again that the disparity was not nearly as strong when tested on WSL. It is faster and requires less memory.

The redeeming quality of my queue is the way its structure makes every case into the common case. Maybe the common case is a little slower than that of the array-based queue, but there’s no reallocation, repositioning, and no need to find consecutive chunks of storage that can support the full size of your structure. There are fewer special cases and it is easier to maintain.

**References**

“The C++ Resources Network.” *Cplusplus.com*, https://www.cplusplus.com/.

“U1 ds04 Lists Stacks Queues.” Crissey, Will, Data Structures.

“Complexity\_Timer.hpp” Rensselaer Polytechnic Institute.

“Complexity\_Recorder.hpp” Rensselaer Polytechnic Institute.