COMP20230 Data Structures & Algorithms

Data Structures Report

Brian Manning

Student ID: 17324576

A report submitted as part of the degree of

MSc. in Computer Science (Conv.)

Lecturer: Professor Madhusanka Liyanage

UCD School of Computer Science

University College Dublin

April 30, 2021

Table of Contents

Chapter 1: **Introduction and Overview**

The following report will outline my work with regards to Assignment 2 for Data Structures & Algorithms. I will discuss the abstract data types (ADT's) of Linked Lists, Stacks & Queues. I will then discuss Real World examples of each data structure. Following this, I will discuss both my implementation of the data structures in Python as well as running time analysis which I have conducted using the data structures.

Below you can see an overview UML diagram of the implemented classes and how they work together.

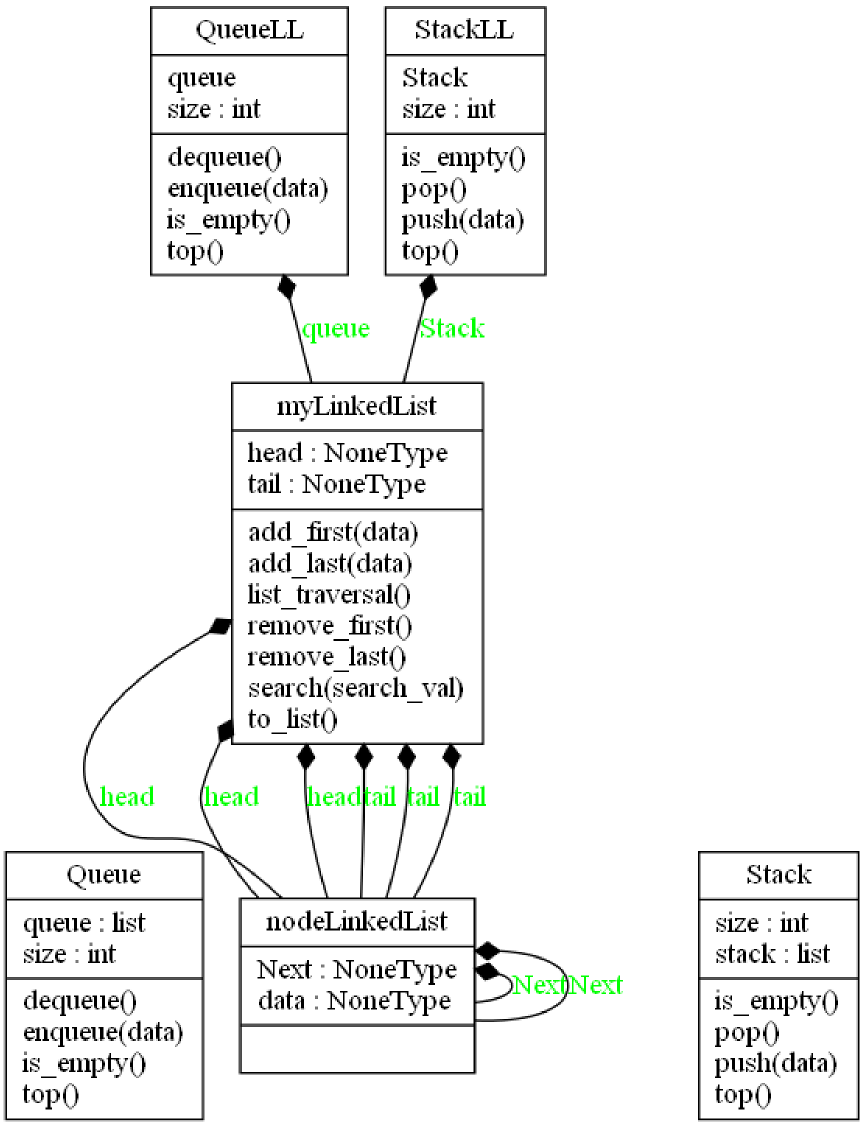


Figure 1.1: UML diagram

Chapter 2: Linked List

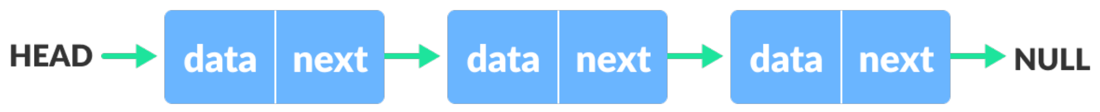


Figure 2.1: Linked List ADT[1]

2.0.1 Abstract Data Type

The Linked List Data Type stores a collection of Nodes which can only be accessed sequentially. The key characteristics of the linked list data type is that each node in the list is linked to its successor using a pointer.

Each Node in the Linked List will contain both data and the location of the next Node.

Key Operations

- add_first: Add an element to the front of the Linked List. Worst Case $\mathcal{O}(1)$
- add_last: Add an element to the end of the Linked List. Worst Case $\mathcal{O}(1)$ when tail pointer is stored.
- remove_first: Remove the node at the front of the Linked List. Worst Case $\mathcal{O}(1)$
- list_traversal: Print every node's value in the Linked List. Worst Case $\mathcal{O}(N)$
- search: Find a given value in a Linked List. Worst Case $\mathcal{O}(N)$

Support Operations

- isEmpty: Returns true if the Linked List is empty, otherwise false Worst Case $\mathcal{O}(1)$

2.0.2 Real World Examples

File Systems on Operating Systems

Linked Allocation is a method for storing files using Linked Lists. It works by storing each file as the linked list of disk blocks enabling the disk blocks to be located non-contiguously throughout the disk. This means that a file will have a head disk block which will contain some data and point the next location in the list, where the next part of the file is stored.

This provides advantages to the file system:

- The file can grow as long as there is still free blocks available across the whole file system.[2]

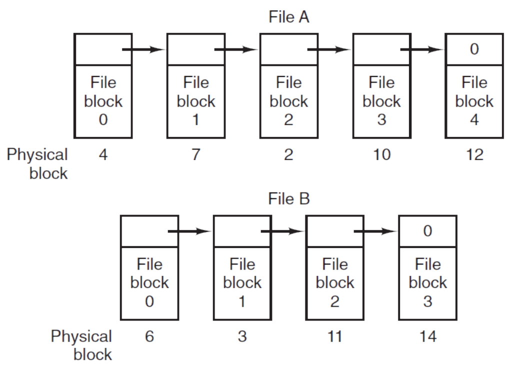


Figure 2.2: File System Linked List

- No external fragmentation: This is because the Operating System can use any free block it has to satisfy more data.[3]

Separate Chaining in the Hash Table Data Type

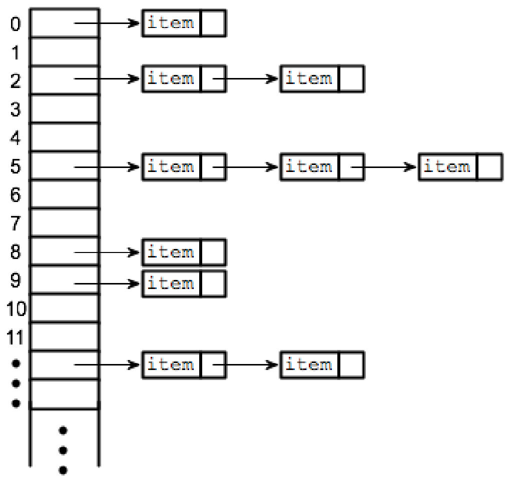


Figure 2.3: Separate Chaining Linked List

Separate Chaining is a strategy of hash table collision resolution. Here we will briefly discuss the use of Linked Lists in Separate Chaining, although other data structures may also be used.

In a Hash Table, when the hashing algorithm hashes more than one data point to the same value it is known as a collision. This collision can be resolved using Separate Chaining, where the data can still be stored at this value alongside the other data points, but it will be added to the end or start of the Linked List at this value.

Linked Lists are a popular choice in collision resolution because they are relatively simple to implement, we can combine all the advantages of using Linked Lists into the Hash Table now e.g. the hash table will always have more storage available. [4]

2.0.3 Implementation

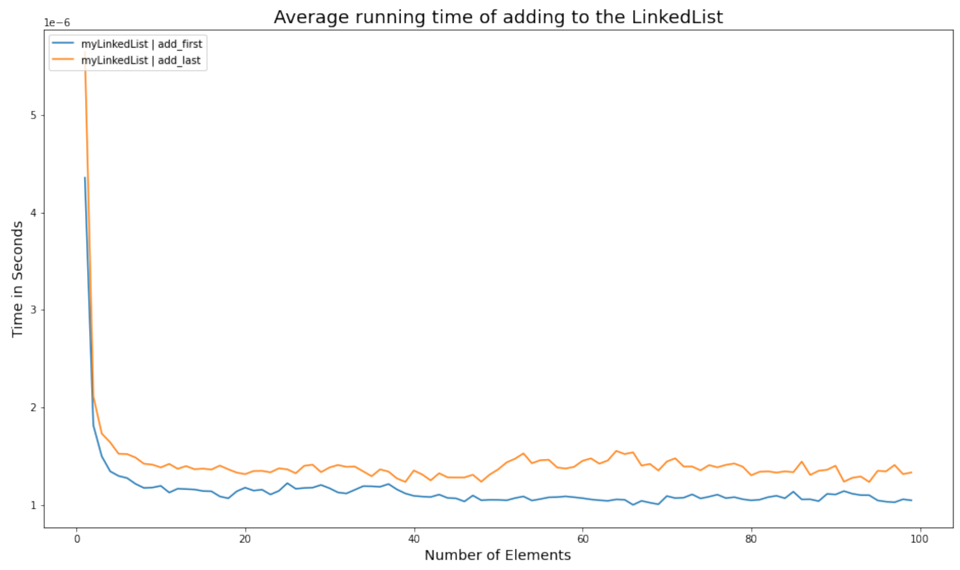
I have implemented the following methods in the Linked List class:

- add_first: adds a node to the start of the linked list
- add_last: adds a node to the end of the linked list
- remove_first: remove the first element of the linked list
- remove_last: remove the last element of the linked list
- list_traversal: print out all the data stored in the linked list
- search: search for a given value in the linked list
- to_list: return a list of all the values stored in the linked list

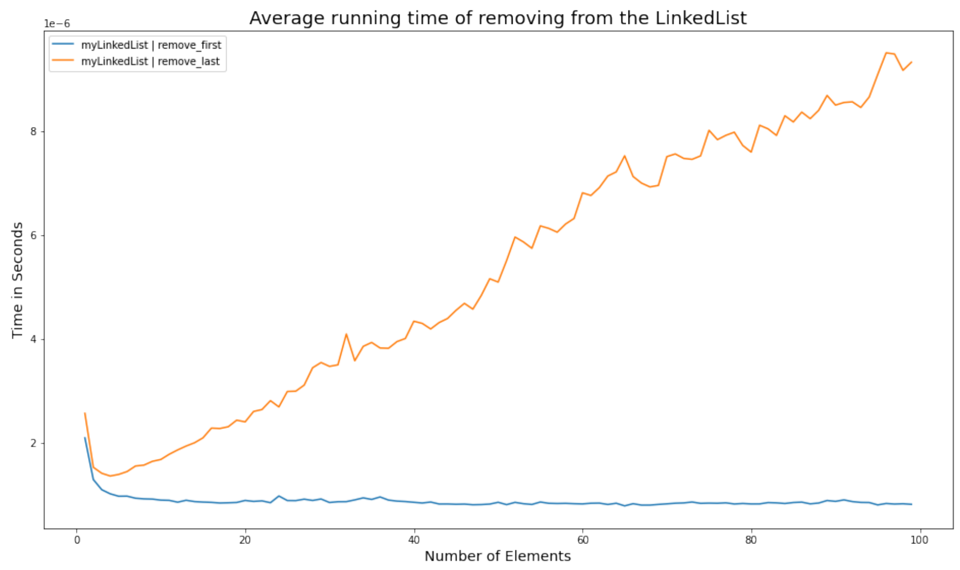
Space Complexity Analysis

The space complexity of this data structure is O(N) as it increases linearly with the number of elements stored in it.

Running Time Analysis



Here we see that for the first element when the Linked List is initially being created, the running time is very high. After the first node, the running time is as O(1) for both adding first and adding last elements to the linked list. The reason they are both O(1) is because I chose to store a pointer to the tail as well as the head of the linked list meaning it has direct access to both the tail and head pointers when inserting both to the start and end of the list.



We see here that the running time of removing the last node increases with the number of elements in the list. This is because the linked list does not store a pointer to the second last element in it, meaning that if it needs to remove the 100th element of the linked list it must traverse all of the elements up till the second last element - this makes the running time of removing the last element O(N). The running time of removing the first element is O(1) as we store a pointer to the head of the linked list and can simply make the head equal to the current head.Next to resolve this.

Chapter 3: Stack

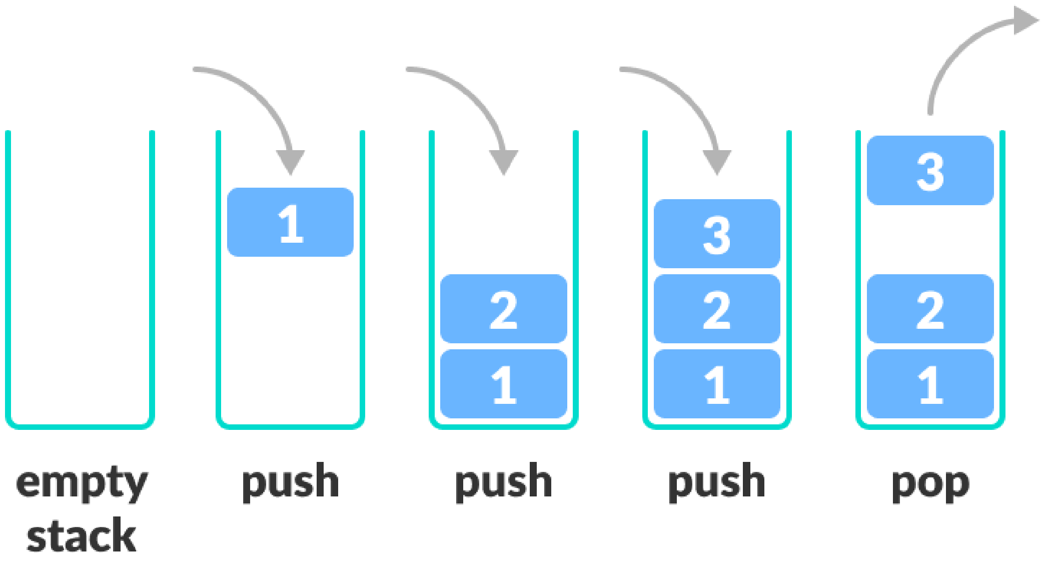


Figure 3.1: Stack ADT[5]

3.0.1 Abstract Data Type

The Stack Data Type stores a collections of elements which can only be manipulated from the top of the Stack.

The Stack is ordered LIFO (Last-In First-Out). This means that if an element is added to the Stack, the next element to be removed in the stack is this element. The top element of the stack is the only element which can be accessed at any given time, this is the most recently inserted element.

Key Operations

- Pop: This removes the top element of the Stack. Worst Case $\mathcal{O}(1)$
- Push: This adds an element to the top of the Stack. Worst Case $\mathcal{O}(1)$

Support Operations

- size: Returns the number of objects in the stack. Worst Case $\mathcal{O}(1)$
- Peek: Returns the element at the top of the Stack without removing it. Worst Case $\mathcal{O}(1)$
- isEmpty: Returns true if the Stack is empty, otherwise false. Worst Case $\mathcal{O}(1)$

3.0.2 Real World Examples

Command Pattern of Undo/Redo Operations

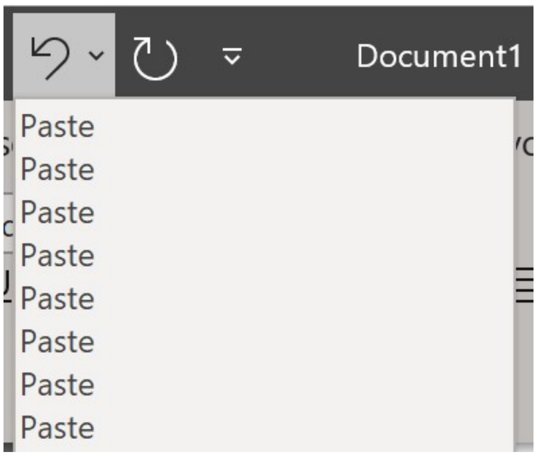


Figure 3.2: Undo Stack in Microsoft Word

The Undo/Redo operations are a fundamental tool in many desktop applications. They provide the functionality to undo an action a user has just taken or redo an action that the user has just undoed. The Stack data structure is perfect for this operation because it allows us to easily push to the top of it and then regain that top value quickly.

When a user performs an action, for instance pasting some text in Microsoft Word, the paste operation will be added to the undo stack. If the user decides they no longer want that pasted value, they can simply press the undo button, the program will then pop the top of value of the stack, run it through the reversing algorithm of the program and the document will be back to its previous state. This operation can then be added to the redo stack. This is the case in other applications, like games[6], also.

Most Recently Closed Tabs in Web Browsers

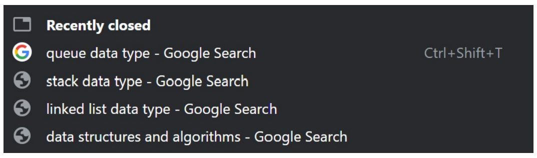

Recently closed
queue data type - Google Search
Ctrl+Shift+T
stack data type - Google Search
linked list data type - Google Search
data structures and algorithms - Google Search

Figure 3.3: Recently Closed Tabs in Chrome

The most recently closed tabs in web browsers is also a similar feature to the undo/redo stacks above. When a user closes a tab in their browser, the browser will add this tab to the web browser's recently closed tabs stack. The user can then use shortcuts, like Ctrl-Shift-T on Chrome, to pop this tab from the top of the stack and reopen the tab. They can continue to use this to keep reopening closed tabs. This data structure allows for very quick access to their most recently closed tabs.

3.0.3 Implementation

I have implemented the following methods in the Stack class:

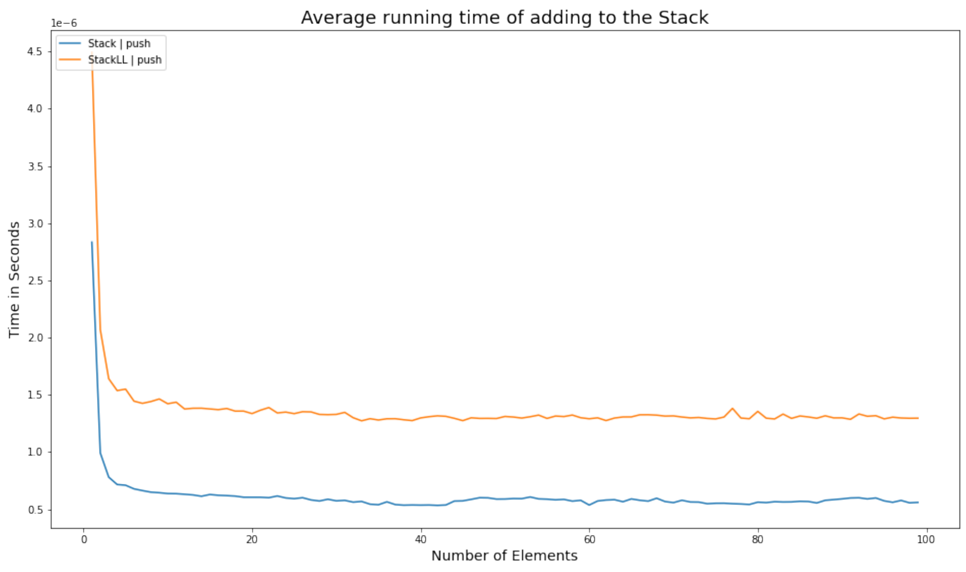
- push: add element to end of stack
- pop: remove element from end of stack and return its value
- top: return the element at the end of the stack without removing
- is_empty: return whether the stack is empty or not
- get_size: return the current size of the stack

I have also implemented this data structure using both lists (class Stack) and Linked Lists (class StackLL). Note, for the list based implementations of both the Stack and Queue data structures, these could be improved by using circular array indexing rather than using certain Python list operations, like del.

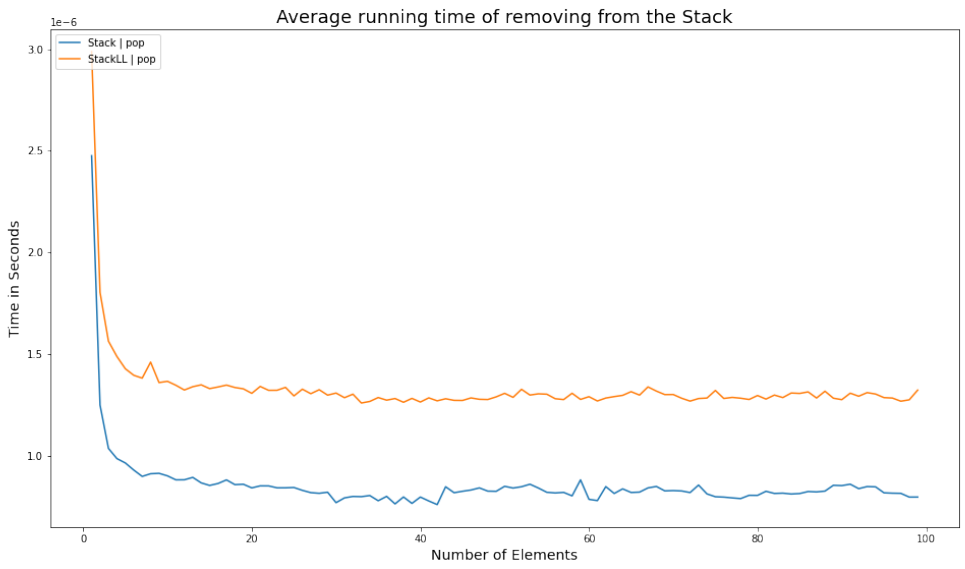
Space Complexity Analysis

The space complexity of this data structure is O(N) as it increases linearly with the number of elements stored in it.

Running Time Analysis



Here it is clear to see that the running time of both implementations of the stack are O(1). This is because pushing to the stack is a single operation which does not change with the size of the underlying stack. The Linked List based implementation takes more time to run than the Python list implementation. This trend is seen for all other operations shown below also. I would assume this is due to Python optimisation of list operations.



The same trend as above for pushing to the stack can be seen here.

Chapter 4: **Queue**

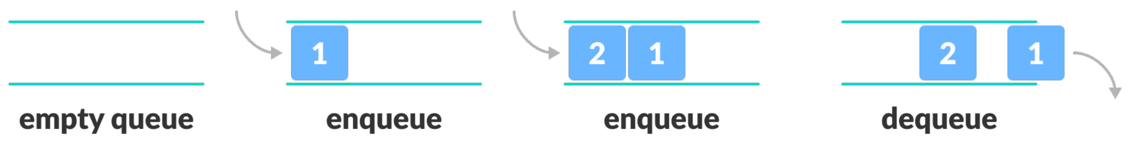


Figure 4.1: Queue ADT[7]

4.0.1 Abstract Data Type

The Queue Data Type stores a collections of elements, which can only be added to at the back of the Queue and removed from the front.

The Stack is ordered FIFO (First-In First-Out). This means that if an element is the first element added to the Queue, no matter how many more elements are added to the Queue, that first element will be the first to be removed.

Key Operations

- Enqueue: This adds a new item to the back of the Queue. It takes the item to be added as input and has no return value. Worst Case $\mathcal{O}(1)$
- Dequeue: This removes the item at the front of the Queue. It takes no value as input and returns the removed item. Worst Case $\mathcal{O}(1)$

Support Operations

- Front: Returns but does not remove the front object in the queue, error if the queue is empty. Worst Case $\mathcal{O}(1)$
- Size: Returns the number of items in the Queue. Worst Case $\mathcal{O}(1)$
- isEmpty: Returns true if the Stack is empty, otherwise false Worst Case $\mathcal{O}(1)$

4.0.2 Real World Examples

Process Scheduling (Operating Systems)

When a process is added to the system, it enters the job queue. Once it enters the ready state, it is then dequeued from the front of the job queue and enqueued to the end of the ready queue[8]. Queues are a good data structure to use in this case as we want to follow a defined order. Note that other scheduling methods may actually be used in practice as the above methods alone will not account for priority of the task.

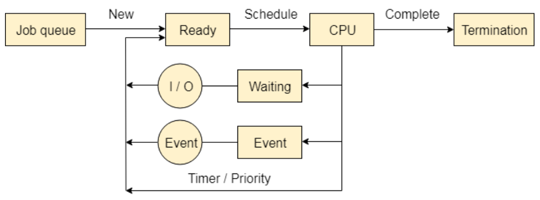


Figure 4.2: Semaphore Queue

Packet Queuing in Routers (Networking)

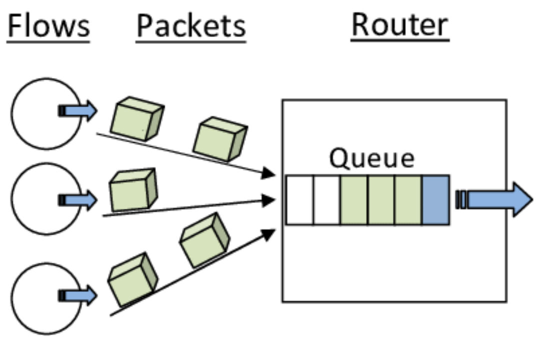


Figure 4.3: Queue in Routers

A router can only process one packet at a time, this means that if packets arrive faster than the router can process them they will need to be stored in the router. The router uses a queue (also known as a data buffer) to store these packets. As the router processes packets, the packets at the front of the queue are dequeued and processed next.

The delay that is caused by this queuing system is known as the Queuing Delay, which is the time the packet will wait in the queue until it is processed[9].

4.0.3 Implementation

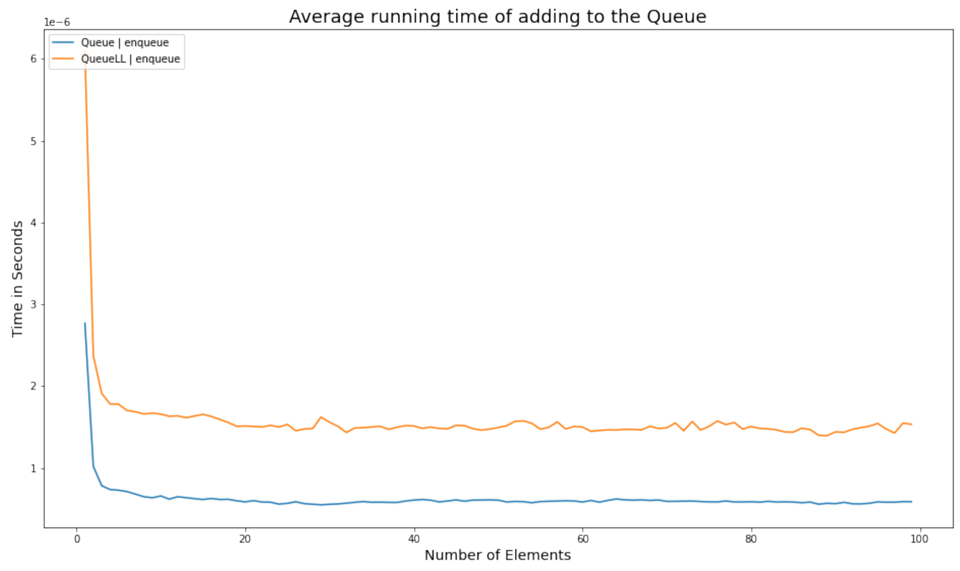
I have implemented the following methods in the Queue class:

- enqueue: add element to the end of the queue
- dequeue: remove element from the start of the queue and return its value
- is_empty: check if the queue is empty or not
- get_size: return the current size of the queue
- top: return the value at the start of the queue without removing it.

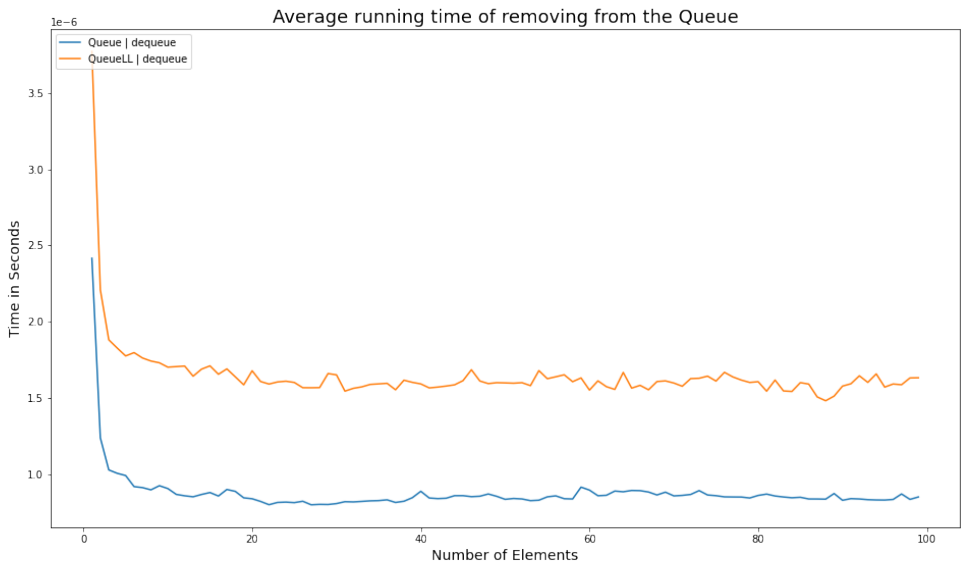
Space Complexity Analysis

The space complexity of this data structure is O(N) as it increases linearly with the number of elements stored in it.

Running Time Analysis



Here we can see the running time of adding to the queue is O(1). This is because adding elements to the queue is a single operation and doesn't change with the size of the queue.



The running time of dequeuing here is also O(1). In the case of QueueLL, this is because the Queue works by adding elements to the end of the linked list and then when we want to dequeue elements we simple remove the first node of the linked list, which in itself is O(1). I have previously implemented the QueueLL the opposite way around so that removing from the Queue was actually O(N) because the whole linked list had to be traversed before removing the last element, but I changed the implementation after noticing this in the running time plot.

Chapter 5: Questions

Please note I have included code in the notebook to illustrate these questions also. Please note for the examples below that for the push() and enqueue() operations, I have taken those methods to return the value of the data which has just been added. This can also be implemented without a return value.

Part II Question 2

During the given series of operations, the following values are returned:

- push(5): 5
- push(3): 3
- pop(): 3
- push(2): 2
- push(8): 8
- pop(): 8
- pop(): 2
- push(9): 9
- push(1): 1
- pop(): 1
- push(7): 7
- push(6): 6
- pop(): 6
- pop(): 7
- push(4): 4
- pop(): 4
- pop(): 9

Part II Question 3

The current size of S is 28.

Part III Question 2

During the given series of operations, the following values are returned:

- enqueue(5): 5
- enqueue(3): 3
- dequeue(): 5
- enqueue(2): 2
- enqueue(8): 8
- dequeue(): 3
- dequeue(): 2
- enqueue(9): 9
- enqueue(1): 1
- dequeue(): 8
- enqueue(7): 7
- enqueue(6): 6
- dequeue(): 9
- dequeue(): 1
- enqueue(4): 4
- dequeue(): 7
- dequeue(): 6

Part III Question 3

The current size of Q is 40.

Bibliography

1. *Linked List Data Structure* https://www.programiz.com/dsa/linked-list. (Accessed on 04/26/2021).
2. *OS Linked List Allocation - javatpoint* https://www.javatpoint.com/os-linked-list-allocation. (Accessed on 04/26/2021).
3. *Difference between Internal and External fragmentation - GeeksforGeeks* https://www.geeksforgeeks.org/difference-between-internal-and-external-fragmentation/. (Accessed on 04/26/2021).
4. *Hashing | Set 2 (Separate Chaining) - GeeksforGeeks* https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/. (Accessed on 04/26/2021).
5. *Stack Data Structure and Implementation in Python, Java and C/C++* https://www.programiz.com/dsa/stack. (Accessed on 04/26/2021).
6. *Undo and Redo: Yes you have to implement it – The Danger Zone* https://mynameismjp.wordpress.com/2008/12/19/undo-and-redo/. (Accessed on 04/26/2021).
7. *Queue Data Structure and Implementation in Java, Python and C/C++* https://www.programiz.com/dsa/queue. (Accessed on 04/26/2021).
8. *Process Scheduling and Operations in Operating System | Studytonight* https://www.studytonight.com/operating-system/process-scheduling. (Accessed on 04/26/2021).
9. *Definition: queuing delay* https://web.archive.org/web/20121219004715/http://www.its.bldrdoc.gov/fs-1037/dir-029/_4318.htm. (Accessed on 04/26/2021).