Anthony Stem
CS 124 | Fall 2021
Project 2 Report

Dataset

This dataset is a subset of the dataset maintained by the Extrasolar Planets Encyclopaedia as of February 28, 2018. It contains data for 3732 confirmed exoplanets from various sources and missions such as Kepler, CoRoT, K2, Kelt, OGLE, and more.[1] This dataset was found on Kaggle and originally contained 97 different columns of data; however, only five columns from the original dataset were used for this project. In addition, a new column called "row" was introduced to create a unique identifier for each row in the dataset. The six fields used in the project are as follows:

• *row* - a unique integer; indicates the row in the database.
• *name* - a string; indicates the name of the exoplanet.
• *mass* - a double; represents the mass of an exoplanet (in terms of Jovian mass J_M).
• *planetaryRadius* - a double; represents the radius of the exoplanet (in terms of Jovian radii J_R).
• *orbitalPeriod* - a double; represents the orbital period (in Earth days) of the exoplanet around its host star.
• *discoveryYear* - an integer; represents the year in which the exoplanet was discovered.

The entries in this dataset are ordered numerically by *row* in ascending order.

The primary reason I chose this dataset is because I have always been fascinated by space, especially exoplanets and extrasolar objects. I wanted a dataset that would allow me to analyze and explore the various properties and attributes of different exoplanets.

Stack Class Complexity

The time complexity of each method in the Stack class are as follows:

Stack()
The Stack constructor has a time complexity of O(1).

~Stack()
The Stack destructor has a time complexity of O(n).

push()
The *push()* method has a time complexity of O(1).

---

[1] Original Source: http://exoplanet.eu/
Kaggle: https://www.kaggle.com/eduardowoj/exoplanets-database

pop()
The *pop()* method has a time complexity of O(1).

search()
The *search()* method has a time complexity of O(n).

peek()
The *peek()* method has a time complexity of O(1).

printStack()
The *printStack()* method has a time complexity of O(n).

Queue Class Complexity

The time complexity of each method in the Queue class is as follows:

Queue()
The Queue constructor has a time complexity of O(1).

~Queue()
The Queue destructor has a time complexity of O(n).

push()
The *push()* method has a time complexity of O(1).

pop()
The *pop()* method has a time complexity of O(1).

getFront()
The *getFront()* method has a time complexity of O(1).

getBack()
The *getBack()* method has a time complexity of O(1).

search()
The *search()* method has a time complexity of O(n).

printQueue()
The *printQueue()* method has a time complexity of O(n).

Discussion

What is the order of the objects before and after adding them to the Queue and Stack? When and why did it change?

Before adding objects to the queue, they were ordered in ascending order by their row number as follows:

1, 2, 3, 4, … , 8, 9, 10.

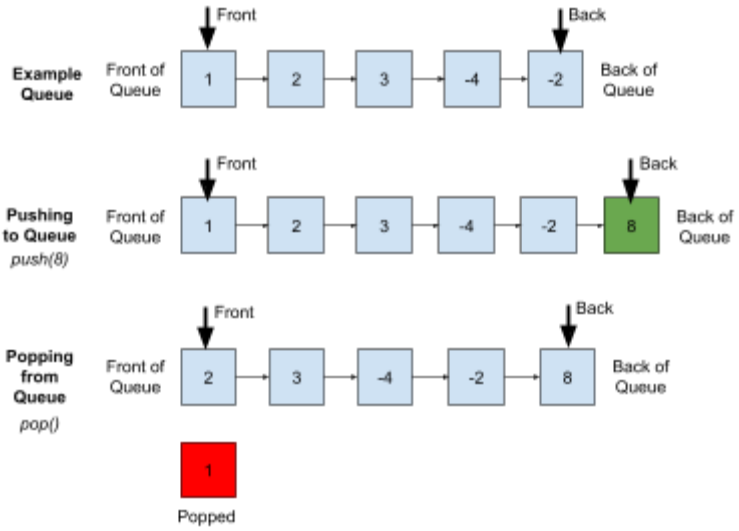
When they were added to the queue, the order remained the same. However, when the objects were popped off of the queue and added to the stack, the order changed to descending order as follows:

10, 9, 8, 7, … , 3, 2, 1.

This change in the order of the objects occurred when they were popped off the queue and added to the stack. This is due to the way objects are stored in a stack. When adding an object to a stack, it is put at the top of the stack, and when removing or popping an object from a stack, it is also removed from the top of the stack. So, when the objects were moved from the queue to the stack, they were continuously added to the top of the stack, which made the order of the objects in the stack be 10, 9, 8, 7, … , 3, 2, 1, where 10 is at the top of the stack and 1 is at the bottom of the stack.

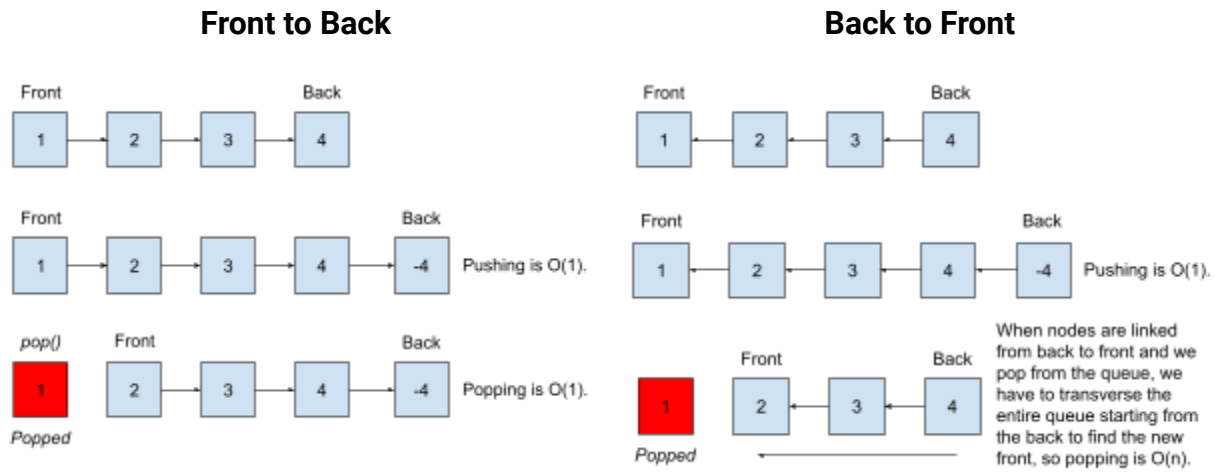
Using the Node class, will the links point from the front to the back of the Queue or from the back to the front? Which way will make the push and pop methods more efficient?

In the Queue class, nodes will be handled in a way such that pushing a node into the queue will store it on the back of a queue, and popping a node off of the queue will remove a node from the front of the queue as shown below:



This means that the nodes which are added to the queue first will also be the first nodes to be removed when the queue is popped; queues follow the FIFO (First-In, First-Out) principle.

The efficiency of the *push()* and *pop()* methods for the Queue class depend on the direction which nodes are linked in a queue (from front to back or back to front).



Queues normally work by linking nodes from front to back, which allows the complexity for both the *push()* and *pop()* methods to be O(1). If the nodes are linked from back to front, the *push()* method would still have a constant time complexity, but the complexity of the *pop()* method would be O(n) instead of O(1). When nodes are linked from front to back, the *pop()* method only needs to move the front pointer to the next node on the right, which is just an O(1) operation since the old front node was pointing directly to the new front node. However, when nodes are linked from back to front, the front pointer cannot access the rightmost node since the old front node wasn't pointing to the new

front node, so it has to transverse the entire queue, starting from the back, to find the new front node. This is why linking nodes from the front to the back makes the *push()* and *pop()* methods the most efficient.

How will you make sure there are no memory leaks?

```
~Queue() {
    while(front != nullptr) {
        pop();
    }
    cout << "Queue Deconstructed." << endl;
}
```

To ensure that there are no memory leaks in the Queue class, I added a destructor (shown above), which is called at the end of the lifetime of a Queue object. The destructor continuously removes the object at the front of the queue using the *pop()* method, which deletes each node from heap memory, until the queue is empty (front and back point to *nullptr*). Without a destructor in the Queue class, each Queue object instantiated in the program will result in more memory allocation without a way to deallocate memory and cleanup when the Queue object isn't needed anymore.

How will you print the objects in the main function? Should you overload an operator?

When printing the objects, we display them in a way such that they show each attribute—*row*, *name*, *mass*, *planetaryRadius*, *orbitalPeriod*, and *discoveryYear*—in a formatted row for each object in the queue. To ensure that the objects are displayed properly, I overloaded the stream insertion operator '<<' to format each object so that each attribute has its own separate column. The output format is shown below:

```
Front of Queue {
1        OGLE-2016-BLG-1469L b          13.6          -1          -1          2017
2        11 Com b               19.4          -1      326.03          2008
3        11 Oph b                 21          -1      730000          2007
4        11 UMi b               10.5          -1      516.22          2009
5        14 And b               5.33          -1      185.84          2008
6        14 Her b               4.64          -1      1773.4          2002
7        16 Cyg B b             1.68          -1       799.5          1996
8        18 Del b               10.3          -1       993.3          2008
9        1I/2017 U1               -1       2e-06          -1          2017
10       1RXS 1609 b              14         1.7          -1          2008
} End of Queue
```

Validation & Testing

How can you demonstrate in your code that your Queue class works correctly? How can you use the integer and string Queue objects to show this?

To demonstrate that the Queue class works correctly, I created a queue for type integer, string, and Exoplanet and tested each method, as well as demonstrated how items are pushed and popped from a queue in the *main()* method. I verified the Queue class's functionality by comparing the output of each method call for each queue to the expected results. To start, I filled the integer queue and string queue with 5 items, and the Exoplanet queue with 10 items. I used the *getBack()* method to check what happens if a queue is empty (*getBack()* should return the default object of the given type), and I used it to check that each item was being pushed to the back of the queue properly. Next, I used tested *printQueue()* method and used it to display the entire contents of each queue to see if the contents were ordered properly. I then tested the *getFront()* and *getBack()* methods to make sure they were getting the proper objects from each queue. After, I tested the functionality of the *search()* method by searching for an item that was and was not in each queue to see if it returns *true* if an item is in a queue and *false* if an item is not in the queue. Next, I tested the *pop()* method and removed every item from each queue object. I printed each item to the console as they were being deleted to show that the Queue class follows the LIFO principle and that the *pop()* method does indeed remove from the front of the queue. Also, *getBack()* and *getFront()* were called again to show that *pop()* has front and back point to *nullptr* when the queue is empty. Finally, each queue was printed to the console to show that the queues became empty. All of the method outputs are in agreement, as well as the order in which nodes are stored in each Queue object.