CMPSC 462: Data Structures

Project-1
Stack, Queue and Deque Applications

Tyler Thompson

Professor Vinayak Elangovan

Submitted On: 2/7/23

TABLE OF CONTENTS

Introduction

Create and develop multiple applications that will help better the understanding of abstract data structures including a queue, deque, and stack. The data structures will be created from scratch and will be used in the implementation of these projects. Each data structure class will provide a set of methods that allow them to function properly so they may be correctly used with each application. The first application will use multiple stacks to allow the famous Tower of Hanoi game to be solved recursively. The use of the stack functions as the structure of the game and proves very useful when it comes to the recursive algorithm used to solve the game. The next implementation of a data structure will be through a demonstration of a server with clients. It will use a queue to demonstrate a person joining the queue and will remove a person from the queue when needed. Although it is just an example it will prove useful for understanding the applications that a queue could be used for. The last use of a data structure will come through the use of deque to simulate a storage container. It provides ways to add items into the container from either side and allows the items to be removed and returned to the user.

The use of all these data structures will be shown through a series of tests. The output will be provided in each test to show how they were implemented and created. Further analysis can come from the code itself, which is heavily documented so any programmer may view it and easily understand how it runs.

Algorithm Theory

There is one specific algorithm used in the Tower of Hanoi implementation that allows it to solve in $O(2^n)$ since each disk that is added results in another set of operations that are more complex. In reality the algorithm actually runs in $2^n – 1$ time complexity, but since 2^n results in almost all the calls nothing else is needed to understand that as n grows sufficiently large so will the amount of time the algorithm takes. In this case n represents the number of disks in the game. A recursive function was used for an easy implementation of the algorithm, and it works perfectly by solving the game as instructed.

Design and Implementation

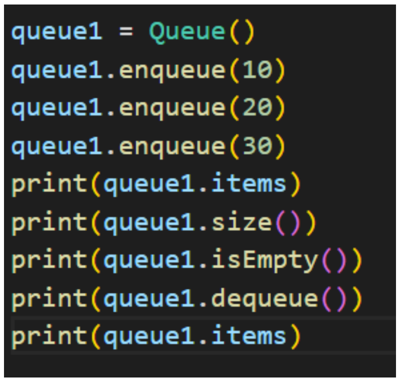
Each of the projects were carefully designed to create the desired result. The algorithm used in the Tower of Hanoi game allows the program to solve the game efficiently and correctly for 3 disks. Of course, with a few modifications the game can be allowed to support any number of disks as everything is in place for it to function properly except for the actual display of the game. For now though, the use of 3 disks will help better the understanding of this concept. Multiple classes were used which include a block class which compares the size of the blocks through comparison operations, a rod class that hold the blocks and uses the block methods to correctly display and format that output of each rod, and finally the TOH class which is the actual game itself. This class also includes the recursive function used in solving the game. The rod class uses a stack to store the blocks onto it, which is used for almost all the operations like printing what is on the rod at that moment. The TOH uses a list to hold each rod and they are all passed into the recursive function that solves the game correctly.

The next project consists of a server example which shows a basic way of clients connecting to a server and joining it. In the case of this example, it assumed that the server is full and there is a wait to get in so each person is added to a queue. The order in which they connect is determined by their place in the queue so clients who join first will be connected first. Once again it is a very basic example of a queue to show how powerful they can be. Methods of this class include adding and removing clients from the queue as well as checking the size of the queue in case someone wants to join and see the wait.

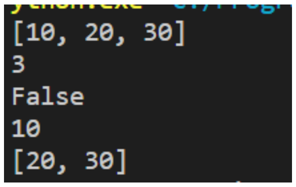
The last project uses a deque to allow a storage container to be filled with items. It is a simple example that allows the deque to be used correctly. Methods of this class include adding elements to each side of the container as well as removing from each side of the container. There is also a way to view the size of the container in case it is almost full. A very practical use case to once again show how these data structures can be used in applications.

Results

Queue

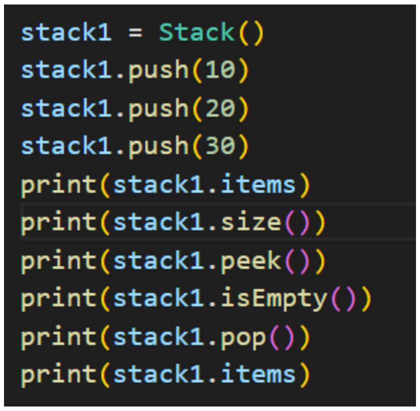


```
queue1 = Queue()
queue1.enqueue(10)
queue1.enqueue(20)
queue1.enqueue(30)
print(queue1.items)
print(queue1.size())
print(queue1.isEmpty())
print(queue1.dequeue())
print(queue1.items)
```

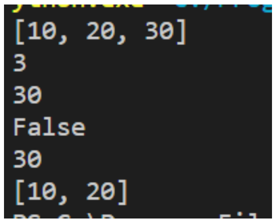


```
[10, 20, 30]
3
False
10
[20, 30]
```

Stack



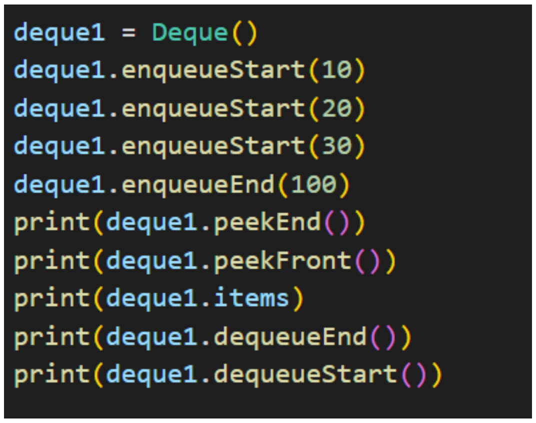
```
stack1 = Stack()
stack1.push(10)
stack1.push(20)
stack1.push(30)
print(stack1.items)
print(stack1.size())
print(stack1.peek())
print(stack1.isEmpty())
print(stack1.pop())
print(stack1.items)
```



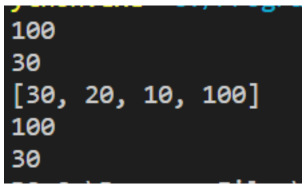
```
[10, 20, 30]
3
30
False
30
[10, 20]
```

Results

Deque



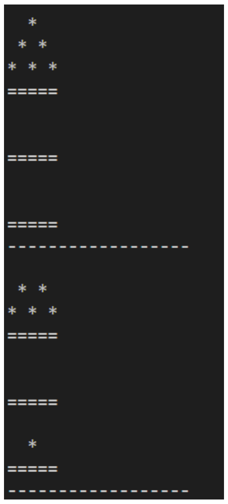
```
deque1 = Deque()
deque1.enqueueStart(10)
deque1.enqueueStart(20)
deque1.enqueueStart(30)
deque1.enqueueEnd(100)
print(deque1.peekEnd())
print(deque1.peekFront())
print(deque1.items)
print(deque1.dequeueEnd())
print(deque1.dequeueStart())
```



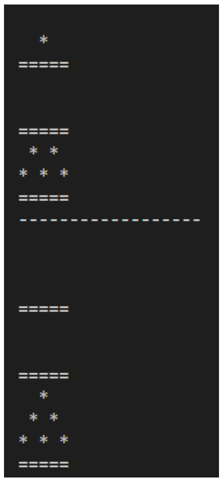
```
100
30
[30, 20, 10, 100]
100
30
```

Results

Tower of Hanoi



```
  *
 * *
* * *
=====


=====


=====
------------------

 * *
* * *
=====


=====

  *
=====
------------------
```

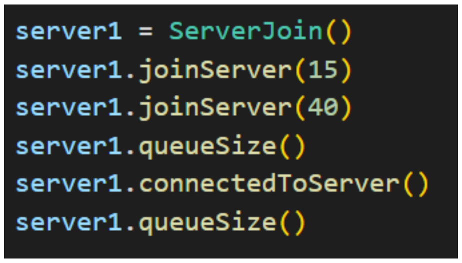


```
   *
 =====


 =====
  * *
 * * *
 =====
 ------------------



 =====


 =====
   *
  * *
 * * *
 =====
```

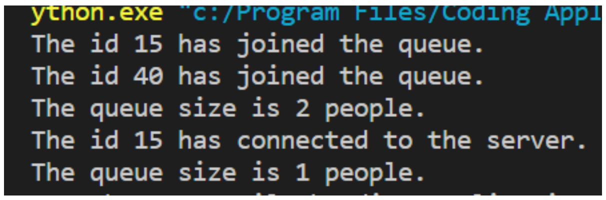
This output of course is a short version of the game itself. The full code can be found within the project and the full output is shown. Each call prints each step shown on how the game is solved.

Results

Server Example



```
server1 = ServerJoin()
server1.joinServer(15)
server1.joinServer(40)
server1.queueSize()
server1.connectedToServer()
server1.queueSize()
```

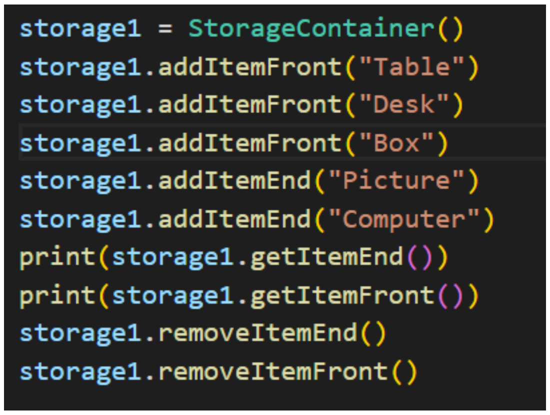


```
ython.exe "c:/Program Files/Coding Appl
The id 15 has joined the queue.
The id 40 has joined the queue.
The queue size is 2 people.
The id 15 has connected to the server.
The queue size is 1 people.
```

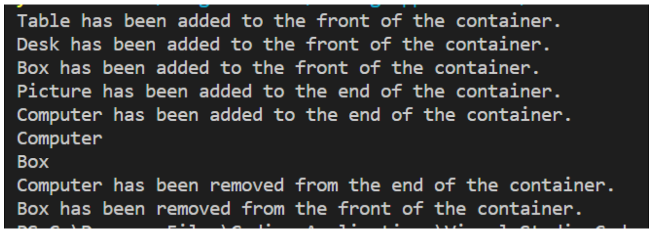
This is the server that uses a queue to run properly. A person is added to the queue each time someone joins. The queue grows for each person. A person is then removed from the queue if they connect to the server which decreases the size of the queue.

Results

Storage Container



```
storage1 = StorageContainer()
storage1.addItemFront("Table")
storage1.addItemFront("Desk")
storage1.addItemFront("Box")
storage1.addItemEnd("Picture")
storage1.addItemEnd("Computer")
print(storage1.getItemEnd())
print(storage1.getItemFront())
storage1.removeItemEnd()
storage1.removeItemFront()
```



```
Table has been added to the front of the container.
Desk has been added to the front of the container.
Box has been added to the front of the container.
Picture has been added to the end of the container.
Computer has been added to the end of the container.
Computer
Box
Computer has been removed from the end of the container.
Box has been removed from the front of the container.
```

This example of a deque shows a practical use that can be used in any sort of simulation to represent an object. Each item that is added to the corresponding side may also be removed if need be. It is also possible and useful to retrieve the item that is the tail and head without removing them from it. The use of this application is to only demonstrate and to provide useful examples.

Conclusion

These projects have proved extremely useful when it comes to understanding and developing data structures. Using each of these data structures shows how far a programmer can go when using them and how many use cases they have in applications. It is apparent that in the programming world this types of data structures will be used frequently and the need to optimize algorithms that utilize them is going to be something that is important. The use of recursion in the Tower of Hanoi game shows how these data structures can be used with so many different applications to solve problems. The most complex problems can be solved by breaking it down into pieces and building it back up with the results of these smaller problems. The recursive algorithm in the project included parameters that accepted objects of the class rod which of course is a stack and this is only the start on how far each of these data structures can be used to solve any problem that calls for them.

There are countless ways to expand and further develop each of these applications including a feature to incorporate any amount of disks in the Tower of Hanoi game. This once again is mostly implemented besides the animation for the rods being printed. For the server example this can be implemented into an actual website that could provide useful data on how long the expected queue time will be if a client were to join at that moment and much more. The last project featuring the storage container can be implemented into some sort simulation that can provide ways to store objects and retrieve them as necessary. There are many video games that use objects like this and it could prove useful. Expanding these projects depends on what is needed in the application they are running in, so for all of these examples the possibilities are up to the needs of the programmer.

References

Tower of Hanoi Explanation

[Tower of Hanoi recursion game algorithm explained - HackerEarth Blog]

Python Documentation

[3.11.1 Documentation (python.org)]