Project 3: Queue the Stacking of the Deque

Project three aims to combine knowledge of arrays, linked lists, deques, queues, and stacks. This project includes eight python files: a corrected Linked_List.py, Linked_List_Deque.py, Array_Deque.py, Stack.py, Queue.py, Hanoi.py, Delimiter_Check.py, and DSQ_Test.py.

Linked_List.py

This file was previously written in Project 2. However, many errors were found throughout the code.

Changes

1. **Header and Trailer Nodes:** The first change that was made was the relabeling of the Header and Trailer Nodes since they were previously labeled incorrectly.
2. **IndexError Implementation:** The second change that was made addressed the second largest error that led to many failed tests. For cases outside of the bounds of the linked list, IndexErrors needed to be raised. In the previous version, this was done by trying to return an error rather than raise. Thus, test cases for insert_element_at, remove_element_at, and get_element_at failed. These were all addressed.
3. **Bound Errors:** For insert_element_at, several errors were made when applying bounds. The first error caused several problems in the stack implementation. One of the bounds was set as index>=self.size. However, this does not account for when the size is 0, and thus was causing an error. In order to fix this, extra conditionals were added stating that self.size could not be zero. Although this fixed the function, it was later pointed out that insert_element_at should not be able to insert an element when the size is zero, as this should only be left to the append function. Thus, the conditional previously added can be deleted or another conditional can be added (this makes the code a bit more wordy but still functionally correct). In the previous version, an unnecessary bound was placed in the conditional that would restrict placement in certain indices. This was flagged as unnecessary and thus removed.
4. **Other errors** fixed were fairly trivial. For example, the floor function was replaced with // in order to increase clarity. Other smaller alterations were made to fix any failed test cases (Ex: making sure removed nodes were returned)

One of the major criticisms of the original implementation was that there were many unnecessary special cases. Although this is true due to the fact that the linked list is sentineled and thus every index has two nodes on either side, the way the code is formatted made it difficult to remove the test cases in the time frame we had to fix our linked list. The current node is set to the first available index, self.head.next, thus making it impossible to add in anything at index 0 without a special case. In order to fix this, current would need to be set initially to self.head, and the loop range would need to be altered. Unfortunately, this process takes a while to visualize for me, and thus since it does not affect runtime nor cause performance problems, I have chosen to leave them in.

Deques

Deques are double sided queues; they can push, peek, and pop from either side. This structure is more flexible than the ordinary queue. The Deque.py file was given in order to check any structure that claims itself to be a deque; it checks that certain functions are present. The deque generator will signal which type of deque is being used in the implementations.

<u>Array_Deque.py</u>

This file uses a circular array to create a deque. This particular array will have a grow function which increases the spaces available for the user to add in an element.

__init__(self)

The __init__ function creates five important variables. Our array will begin with a capacity of one; thus, a variable to track capacity was created. Because the contents of the array depend on capacity, another variable holding places for elements will be created. This is our array; it will have a certain number of spaces determined by capacity. Per usual, we also need a variable to hold the size of the array, which starts at zero. Because this is a circular array, the front and the back need to be tracked; this is set up in the constructor as None since the array is initially empty.

__str__

The string function is very similar in nature to the linked list string, but there is one major difference. Rather than just running through everything in self.__contents, the function must distinguish the starting point and add each element to a string. If we apply the __str__ function from linked lists, we see that a circular array might not be in the correct order from left to right. For example, in the setup output for Towers of Hanoi, the output would be [1,2, None, 0]. However, if we run through the array starting at element 0 and wrapping around to the front, we can reorder this in an easier way to read. In order to do this, the for loop needed to change from running through the contents to running through the indices. With self.__front as the starting index, the loop can add each thing into the string and then change the index using modular math. This function still remains at quadratic time.

__grow

The grow function's algorithm works by taking the array at its current capacity and creating a new temporary variable for it (Line 39). Then, the capacity variable is changed; in this case it is multiplied by two(Line 40). In the following lines, we copy the old list to the new list with more empty slots. In other words, we are creating a new empty array and filling it up with old elements in its new corresponding order. This essentially is doing the same thing as is done in the __str__ method but with different types of collections of elements. The modulo operator assures that the array is circular; the remainder points at the index the element should be put in. For example, if we have an array of size ten with a front index of seven, the push functions will place elements up through index nine. Once it runs out of indices, the modular operator will kick in and compute that (9+1)%10 is 0, and thus the next index will be zero. Lastly, the array's self.__front variable is realigned in order to make up for the transfer of index. The grow function will only be used in cases when the capacity will equal the size of the array. When this occurs, the capacity will be doubled. The runtime is quadratic time.

_push_front

The push function's first concern is deciding whether or not there is enough space to add in another element. If it needs space, the grow function is applied. The second condition in the push function depends on the size. If the array is empty, self.__front and self.__back are none. Thus, when something wants to be pushed in when the array is empty, we must set self._front and self._back to index 0. The size also must be increased. If the element is being pushed from the front in any other case, the element must be placed in its corresponding index. Once again, modular math is necessary. Because it is being placed in the front, the modular math function will feature the index of the front subtracted by one. Then, the new front must be reestablished as the new element and the size must be increased. The back label must also be reassigned, as it will have increased by one. The runtime is constant, as it only needs to go to the front element regardless of the size of the array.

pop_front

This pop function will remove the first element of the list. It will take the front element (self.__front) and tag it as the item that needs to be removed. Then, it will reassign the value at this index to None. Lastly, it will reassign the value of the front element(self.__front) and return the popped item. Reassigning the front element involves pushing the label by one and then using modular math to find the location in the circular array. There is only one instance where the pop function should not work. If the size is zero, there are no elements to remove on the list. This function also has a runtime of O(1), as it will only run through one element in the worst case scenario.

peek_front

The peek function will return whatever value the front label holds. This will only fail when the array is empty; it will just return None. Because the peek function just returns an indexed element, it has a constant runtime.

push_back

The push back element will also first define whether a new element can be added or if the array is full. It will call on the grow method when necessary. The push_back algorithm works very similarly to the push_front algorithm; if the size is zero, the labels of front and back need to be set to the 0 index. Otherwise, the value needs to be added at the respective place it belongs in by using modular math. Because the back label is the only one that is affected, it is the only label that needs to be reassigned. This function will also work at constant time, as in its worst case, the back label will need to be found in one step regardless of size.

Pop_back

The pop back function will work similarly to the pop front function. It will find the back element that needs to be removed and then reassign it to None. Lastly, it will change the size and the back label. This also will run in constant time due to the fact that it only needs to find the value associated with self.__back.

Peek_back

The peek function will look at the last element in the array and return it. This will fail only when the array is empty due to the fact that there are no assignments to an element (everything will be None). Because it is just calling an element of the array at a particular index, it is constant time.

The array deque will be the basis for further implementations.

Linked List Deque

The second deque is made of linked lists; thus, one must format the deque using the previously made linked list class. This implementation is made only of calling functions from the linked list algorithm.

The runtimes for all methods depend on the linked list algorithms. However, there is a slight difference with some due to the fact that deques do not need to insert or remove anything from the middle. Thus, most of the runtimes should be constant. For example, the insert function is usually linear time. However, in this implementation in push_front, the linked list only needs to look to the right of its header or left of its tailer in its worst case. Thus, the runtime will more likely be constant. This will only be the case for the methods which call the front or back of the deque; anything that needs to run through the entirety of the list will not be constant. The implementation of the majority of these functions involves only one inherited function from linked_list.py with the exception of the push front function. Because the rules of linked lists state that the insert function cannot insert a node when the list is empty, an exception needs to be made. The append function is the only function with power to do this. Thus, there is one conditional statement in this sector of code. All other elements pushed to the front will be inserted with the insert function at index 0. The pop back and peek back functions also have an argument that is beyond the scope of an index number. These functions need to find the last node, and thus find the index by subtracting one from the length of the linked list. In the linked_list.py file, we raised several IndexErrors. Project three differs such that no errors are raised. Thus, we must account for this in Linked List Deque. For the functions that produce IndexErrors such as pop front, peek front, pop back, and peek back, we need to avoid any circumstances that may produce errors. Thus, conditionals were inserted in order to return None. These conditionals tended to occur when the list was empty.

As mentioned previously, the runtimes for this document can be a little confusing. The __len__ function is constant time. Push_front contains two functions from linked list: append_element and insert_element. The append element will have constant runtime, while the insert_element function will deviate from its linked_list.py runtime. As we mentioned earlier, this will have a constant runtime because it is inserting something to the node next to the header. Overall, push_front will thus have a constant runtime in its worst case. Pop_front will also deviate from its linked_list runtime because it is removing the element next to the header in every single case. Thus, it doesn't matter how many nodes are in the linked list, and this will have a constant runtime. Push back is the inverse of push front; it will be finding the trailer node and appending an element to its left. Since the trailer nodes are signaled, it will find the index to insert an element at a constant runtime across all lengths. This function does not deviate in runtime from the linked list implementation. The same logic can be applied to pop back and peep back. Although their runtimes in linked_list.py are linear, here we only need to look at the element next to the trailer node for every case. Thus, in this deque file, the pop back and peek back runtimes are constant. The __str__ function's runtime will depend on the runtime in its worst case in the linked_list.py. Thus, this is quadratic time.

Stack.py

Stacks are structures which only have one opening for push, pop, and peek; in other words, the algorithm must signal either the left or right side as the opening, and elements can only be manipulated from that side. Because Stacks depends on the Deques, the six deque functions will be applied to form Stack functions. In this code, the left side of the deque was chosen. Thus, we must push, pop, and peek everything from the front. The size is increased or decreased for push and pop respectively; everything else corresponds to its deque function. Because Stack is calling back to previous classes, the runtimes will once again depend on the deques. There are no deviations from the runtimes of the inherited functions. Logically speaking, the functions in stack only refer to the first node of the deque. Thus, no matter the size of the stack, the worst case will be looking at one element only. All functions in stack are constant time with the exception of __str__. This will run with the worst case runtime of the deques; for both array deques and linked list deques, this function is quadratic time.

Queue.py

Queues have two openings but restrict which functions can be applied to these ends. These structures have one front and a back. The enqueue function will add the element to the back; thus, the function is comparable to the push back function in the deques. The dequeue function will remove the first element in the front, thus comparing to the pop front function in the deques. The peek function will only look at the front element, thus comparing to the peek front function in the deques. These will all have a constant runtime because it depends on the runtimes of the functions from the deques and only pertain to the end indices. All other functions depend on the constructor or inherited classes. Thus, the __str__ will be the only function with quadratic time.

DSQ_Test.py

The DSQ Testing file runs unit testing for deques, queues, and stacks. There are a total of 103 test cases; each runs several scenarios for the files.

Deque Test Cases

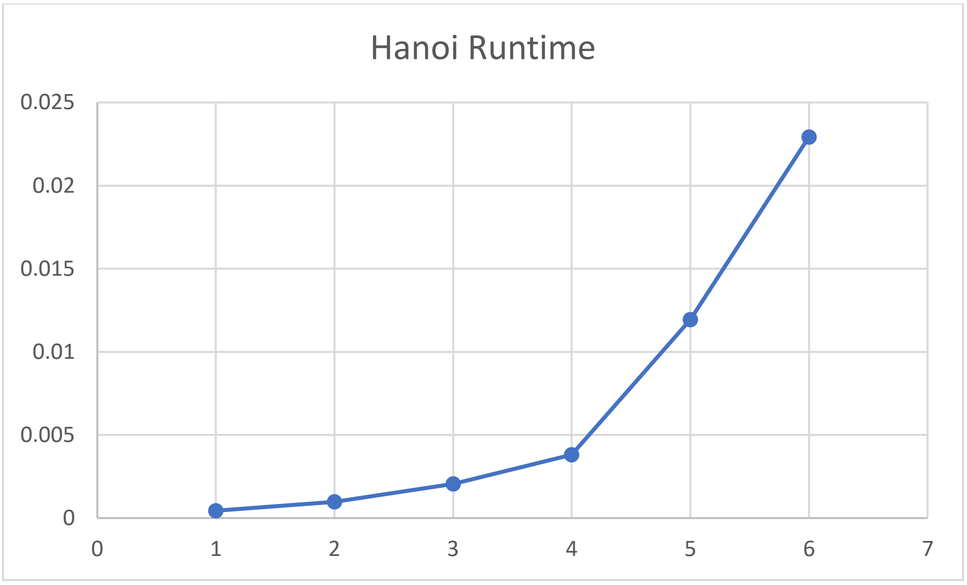
The Deque Test Cases can be sorted into a few categories: number of elements currently in the deque, the type of input the deque was receiving, and functionality of each function. The code is mostly sorted by number of elements in the deque. Test Cases 1-6 all deal with an empty deque. Each function in the deque is tested at least once, as well as the empty deque in string form and its length. All of these functions must either return None and return that there are zero elements in the deque. Tests 7-22 deal with a deque with one element in it. In my implementation, it is very important to not just test none and some, but to test none, one, two, and some elements. Why? Array Deque has special cases for when the size is zero. My linked list implementation (although it shouldn't have these special cases) depends on several special cases such as when we insert at index zero. Because the nodes in tests with 1 and two elements are attached to header and trailers, the bond it forms between these nodes is slightly different (in my code at least). Secondly, many errors in my linked list implementation dealt with applying a function at 0,1, and 2, so these must be checked more thoroughly. Once we reach greater than two nodes, the nodes will have links between themselves and thus, differentiation is not necessary. Some of these may be unnecessary, but again, due to past errors in my code, each one was necessary to check that the bugs were fixed. Looking at the actual functionality of the code, each function is tested at least once. It can be noted that for every push and pop function in the entirety of the file, it is usually followed by an almost identical test that runs through every single function but tests/returns length. This is because some functions need to be check not only for functionality in its output, but functionality in its reassignment of self.__size. Had we been writing test code for array deques alone, we could have also run tests that return the positions of the labels front and back. However, for flexibility across different structures, these were left out. For Project two, I arbitrarily tested different combinations. However, it wasn't until a later implementation that I realized that the code did not work in every order. Thus, in my code for one element, there are different combinations of functions. For example, push_front and push_back do the same thing when adding one element to the deque, but because they are different functions entirely, both needed to be checked in concurrence with something like a peek or pop function. Thus, there are several similar combinations that have the same output but do not entirely work the same in their logic. This section of code essentially 1) checks every single function when there is one element already in the deque and 2)Checks different combinations that can be made with functions when one element is in the deque.

The third section of deque tests are for when there are two elements in the deque. The same logic was applied from the previous section. Each pop and push function test has a secondary test to check the size reassignments. The code is mainly split up between adding in items from the back and front, and then checking combinations of these with other functions. The fourth section of the deque test cases addresses when many elements are placed into the deque. Along with the logic from the last couple test sections, this one mixes insertions from the back and the front in different combinations and applies other deque functions to them. The chosen combinations must pass for each of these four size categories. The last section in deques is under tested; it tests the ability to add in non-integers into the deques. Although the previous lines could have all been repeated once more with the non-integers, after a few tests, a vote of confidence is placed in the ability of each function to recognize the element as separate. For the future, I would try a combination of integers with non integers. The Deque class is the class with the most functions; thus, many more test cases were required. The breakdown of the overall test cases is about 50-25-25 with Deque being 50 and Stacks and Queues being 25. Mathematically, this seems like a fair distribution since 1)Stacks and Queues depend on Dequeues and 2)Dequeues have almost twice the functions.

Moving on to Stack and Queue Tests, it can be seen that the logic is done in an identical manner to the Deque cases. They both begin with the empty cases and test structures with 1,2, or many elements. While coding stacks and queues, there were errors that were derived from the deques that did not show up in the few original cases tried with deques. Because each structure depends on each other, the logic for deques should logically follow into stacks and queues.

Towers of Hanoi

The Towers of Hanoi code is split into two main sections. The function Hanoi(n) creates three stacks. A while loop then pushes each ring onto the source stack in corresponding order from biggest to smallest. This process recurs until it reaches the base case. Movement occurs with the Hanoi_rec function. We first introduce the base case. In order to determine the base case, we need to look for the smallest element for which we know a direct answer. In this case, the base case occurs when n=0 because when there is one ring, we know that the ring simply needs to be placed from the source to the destination. For all other cases, we need to move smaller rings out of the way. In the first recursion, we do exactly this; we recur all the rings to the middle with the exception of the biggest ring in that recursive cycle. Then, we can move the biggest ring to the destination. In order to repeat this across all n's, we need another recursion to repeat this process until all rings are at the end. The order in which one places s,a, and d depends on the order in which things need to be moved onto the stacks. Three recursions occur throughout the entire Hanoi file along with one while loop. This calls for a more complex runtime than we have analyzed; below are the runtimes from n=1 to n=6:



From the graph, we can see that the runtimes rapidly escalate with the rate of change becoming larger and larger as more n's are added. Thus, we can at least assume the runtime must be greater than linear time.

Delimiter Check

Delimiter Checks can be implemented with stacks. In order to determine whether a file is balanced or not, the function must run through every element in the file. When it finds a symbol that is one of the three delimiters specified, [], (), or {}, it is added to a created stack. When pairs match, they are popped off. The algorithm for this begins with the creation of a stack. A for loop then runs through each symbol in the file. A nested conditional was used in order to save space; had this not been used, several if statements would need to be added, making the code longer. The first conditional filters out symbols that are not the desired delimiters. The second conditional if filters out if it is a left hand delimiter. If it is, the delimiter is pushed onto the stack. The elif conditional finds pairs; it states that if the top symbol on the stack is a left hand delimiter and the current symbol is its matching right hand pair, the top delimiter can be popped from the stack. If it is a delimiter but does not follow the previous two conditionals, it returns False. This loop continues throughout every single symbol. If the length of the stack is zero at the end of the loop, then the file is balanced so it returns true. If it is not empty, the file is not balanced. This file has linear runtime. In the worst case, the file will run through every single symbol in the file, thus making it depend on the number of symbols in a file.

Raise vs Return

In project two, we raised errors in order to notify the user that the input they had put in was out of bounds. In project three, we curbed that function by simply returning an unchanged structure back to the user. In order to analyze which one is better or worse, we need to first understand what would happen if unacceptable inputs were not accounted for at all. For linked lists, one would receive an Attribute Error such as

```
AttributeError: 'NoneType' object has no attribute 'prev'
```

This is due to the fact that python is looking for the node with those values, and fails to find a node with any next and previous connections. In fact, “None” is returned to python at unacceptable indices, and thus it fails. For arrays, one may get an error such as

```
TypeError: list indices must be integers or slices, not NoneType
```

Once again, we can see that the out of bounds index returns “None”. Thus, python will fail. So which one is better? As we can see, returning None at out of bounds indices is exactly what python is doing when neither statement is called. Thus, using the return statement simply is allowing python to do this without causing the program to crash. The raise statement will bring up an IndexError, which is most commonly known to mean that something was referenced out of range in the program. This could be very useful information for the person using the program. Let’s assume I am building a program such as the Towers of Hanoi that is an application of one or several of our files. Because my program will be inheriting elements of the other codes, raising an IndexError that points out exactly what condition my code breaks can help me change my program to fit with the requirements. Although in this project we coded all nine files, in a collaborative project or work experience, I may not have coded one of the files and may not know the exact specifications. Thus, in a setting where the user can read IndexErrors and understand them, it may be beneficial to use raise IndexError. If I were implementing some program such as a Towers of Hanoi videogame, it might be more useful for me to return None and add a print statement saying out of bounds. For example, if a little boy was playing the game and tried to insert -5 rings, an IndexError might make him assume the game crashed and would drive him away from using it. However, if the game does not add any rings or states that he must put in a number higher than 0, the boy might have a better understanding. Thus, there is no right or wrong answer for whether one is better than the other. Raise and return should be chosen based on their respective implementations/programs.