Ignat Miagkov

CSCI 241

29 March 2020

Project 3: Queue the Stacking of the Deque

Project 3 built on what we created in project 2: our task this time around was to build stack, queue, and deque classes using our linked list implementation from the last project, as well as create an array deque implementation to create stack and queues to compare results between the two. Using our stack, queue, and deque implementations we were then asked to create a delimiter checker and a recursive solution to the Towers of Hanoi problem. This write-up will detail performance analysis and the testing process for the deque, queue, and stack implementation for both an array and a linked list environment, followed by my solution for the delimiter checker and the Towers of Hanoi.

Performance analysis for Deque (both list and array based)

__init__ : O(1), initializes attributes for array deque, creates a new linked list for the linked list deque. Both implementations do not depend on any other factors and will always create an instance with the same attributes, thus taking constant time.

__str__: O(n), prints out a string representation of the deque. Run time will depend on the amount of values in the deque, thus taking linear time proportional to number of values in list.

__len__: O(1). Returns the size of the list or array, which is kept track of by a single integer. Always takes constant time.

__grow (only for array structure): O(n), this function doubles the capacity of the array in the array-based structure, as well as shifting all values to the left by copying into a new array and replacing the old one. This method will take linear time, as the for loop that copies values into the new array will depend on the length of the old array.

push_front: O(1). In a linked list, references at the beginning of the list are updates and will always take constant time, does not depend on any other factors. In an array, since we have front and back as attributes of the array deque, it will always take constant time to put the value in the specified cell and update the back. However, in an array deque, if the size is equal to the capacity, we will have to call the grow function, which itself is linear time. Thus, push_front will normally take constant time, except in the case of an array deque in which the user is attempting to add to an array that is full, in which case push_front will take linear time, or O(n).

pop_front: O(1). In a linked list deque, references at the beginning of the list are updated and the removed value is returned. This does not depend on the length of the list, thus being constant time. In an array deque, we again keep track of the front and back, so we already have our index for the removal at the front. Value at front is removed, front and size are updated. This also takes constant time, since the removal from an array takes O(1), and the only thing we are doing is updating the front index.

peek_front: O(1). In a linked list deque, we are going to access the value at the very front of the list and return it without modifying the deque. This takes constant time since we are always going to move one

place over from the header no matter the contents of the list, thus always being constant time. For the array deque, since we keep track of our front, we do not need to find the index to access the value from, and since accessing a value in an array takes constant time, this method will also take constant time, or O(1).

push_back, pop_back, peek_back: All of these are the same as the previous three methods, except taken from the back of the deque. These three methods are all O(1), except in the case when the array is at full capacity and push_back is called: in this case, the array will have to grow, which takes linear time, thus the push_back function call will take linear time whenever the deque is at full capacity and the user tries to call push_back. Otherwise, the methods are the same as the front methods, except this time the back pointer will have to be updated every time, rather than the front pointer in the array. In the linked list, nothing changes, other than moving one node previous the trailer node, and updating the references there.

Performance Analysis for Stack

__init__: O(1). Initializes a deque, either array or linked list structure. Just creates the instance, always takes constant time.

__str__: O(n). Depends on the amount of values in the stack, takes linear time.

__len__: O(1). Returns length of the stack, which redirects to returning the size attribute. Method calls a method within the deque class that takes constant time; thus this will also take constant time.

push: O(1). Calls push_front from the deque implementation. If using a linked list, push in stack will always run in constant time, since adding to the front of the linked list also runs in constant time. If using an array deque, push will also take constant time, unless the capacity of the array is equal to the size when the push_front is called. If the array is not full, push to stack will be constant time, while if the grow function is necessary to push, then it will run in linear time since it will depend on the length of the stack as the grow functions is called.

pop: O(1). Calls pop_front from the deque implementation. An array-based stack and a linked list-based stack will both run in constant time. In the linked list structure, the outermost value will be removed, and references will be updated, thus always taking constant time. In an array-based stack, the value with which front is pointing to will be removed and the front pointer will then be updated. However, the index of removal does not have to be searched for, thus this removal will also take constant time.

peek: O(1). Calls peek_front from the deque implementation. An array-based and linked list-based stack will both run in constant time. In the linked list structure, the outermost value will be accessed, and since is it always one node over from the header, it will always take constant time to retrieve the value at that node. In an array-based structure, the value with which front is pointing to will be accessed and the value will be returned without altering the stack. Since we already have a pointer to the front value, the peek method will always take constant time.

Performance Analysis for Queue

__init__: O(1). Initializes a queue, either array or linked list structure. Just creates the instance, always takes constant time.

__str__: O(n). Depends on the amount of values in the queue, takes linear time.

__len__: O(1). Returns length of the queue, which redirects to returning the size attribute. Method calls a method within the dequeue class that takes constant time; thus, this will also take constant time.

enqueue: O(1). Calls push_back from the deque implementation. If using a linked list, enqueue in queue will always run in constant time, as the new node is added at the back of the list and references are updated, without any dependence on length or other factors. If using an array deque, enqueue will also take constant time, since the index for the back is kept track of within the array implementation, thus the insertion at a known index will take constant time. However, if the size of the array is equal to the capacity of the array, then the grow function must be called, and that will take linear time. So, if the array is full with values, the next enqueue operation will take linear time.

dequeue: O(1). Calls pop_front from the deque implementation. An array-based queue and a linked list-based queue will both run in constant time. In the linked list structure, the outermost value will be removed, and references will be updated, thus always taking constant time. In an array-based queue, the value with which front is pointing to will be removed and the front pointer will then be updated. However, the index of removal does not have to be searched for, thus this removal will also take constant time.

peek: O(1). Calls peek_front from the deque implementation. An array-based and linked list-based queue will both run in constant time. In the linked list structure, the outermost value will be accessed, and since is it always one node over from the header, it will always take constant time to retrieve the value at that node. In an array-based structure, the value with which front is pointing to will be accessed and the value will be returned without altering the queue. Since we already have a pointer to the front value, the peek method will always take constant time.

Test Cases

Stack: First we check the str method by printing the empty stack out. Then we test the stack by pushing to an empty stack, stack of size one, stack of size two, and stack of size three. We then test the pop function by checking what was returned, the remaining stack, and the length. We then test the peek function by checking what was returned, the remaining stack, and that the length. The main difference between the peek and the pop should be the unaltered state of the stack during a peek, as well as the length not changing. After all of this, we check to make sure that the user can still call pop and peek off an empty stack without an exception being raised.

Queue: First we check the str method by printing the empty queue out. Then we test by queueing to an empty queue, queue of size one, queue of size two, and queue of size three. We then test the dequeue function by checking what was returned, the remaining queue, and the length. We then test the peek function by checking what was returned, the remaining queue, and the length. The main difference between dequeue and peek is the unaltered state of the queue during a peek, as well as the length not changing. After all of this, we check to make sure that the user can still dequeue and peek off an empty queue without an exception being raised.

Deque: First we check the str method by printing the empty deque. Then we test pushing to the front for and empty deque, deque of size one, deque of size two, deque of size three, and a deque of size

four. We repeat everything just mentioned but by pushing back to deques of those sizes to check the push back functions. We then test popping from the front by checking the returned value, the remaining deque, and the length of the deque. Afterwards, we test peeking from the front by checking the returned value, the remaining deque, and the length of the deque. The main difference between peeking and popping should be the unaltered state of the deque after a peek, as well as the length remaining unchanged. This whole process is then repeated for popping from the back and peeking from the back, checking for the same behaviors. Finally, we check that a user can peek and pop from an empty deque without raising an exception, from both the front and the back.

Delimiter Checker

For the delimiter checker, I used a stack to aid me in the process. Initially, I created two lists: one with the opening delimiters and one with the closing delimiters. Additionally, I created a dictionary to redirect the opening delimiters to its respective closing delimiter. For each character in the file being checked, if a character is an opening delimiter (so is in the list of opening delimiters), it will be pushed onto the stack. If the character is a closing delimiter, we then check if the closing delimiter is associated with the correct opening delimiter on the top of the stack. We do this by checking if the closing delimiter redirects to the opening delimiter that is popped off the stack. If it does not match, then false is automatically returned. If it does match, the function proceeds as normal. The final check is if the stack length is equal to zero, then it will return true, since all the delimiters matched to its respective delimiter, thus containing balanced delimiters. Otherwise, if the stack is not empty, somewhere the delimiters did not find a match, thus returning false. This function runs in linear time: it depends on the length of the filename being passed along to be checked. However, if the array stack implementation is used, then it will run in quadratic time if a very large file is used, as the array will have to keep on calling the grow function, causing the stack to have to be copied many times. Using the linked list implementation would be optimal for this situation, as it will always run in linear time.

Towers of Hanoi

For my solution of the Hanoi problem, I used a base case and a recursive case. The base case is that if we are on the bottom most disk, which in this case was labeled as zero, then it should be popped off the source stack and pushed onto the destination stack. Otherwise, call the Hanoi_rec function again, but use n-1 disks and swapping the locations of the aux and the destination stacks in the call. Then, pop the top disk off source and push it onto destination. Afterwards, call Hanoi_rec again, but this time use the aux peg as the source peg, the source peg as the aux peg, and the destination remains the same. This else section simulates the solution to the part of the problem to get to the state in which the bottom disk is alone on one of the pegs, while the rest of the disks are stacked on the aux peg. These calls will then continue until they reach the base case and will start pushing and popping disks all over the place and printing the result with each step. The run time of this problem is exponential. In fact, it takes 2^N - 1 to solve the Towers of Hanoi problem with N disks.

Raising Exceptions

I feel that the incentive to not raise exceptions for removing and peeking an empty structure in this case is valid. It is very well possible that the fact that a peek or a remove is a None could be very important to the user or whatever product is implementing such a structure. For example, a stack could be used to

track the remaining left in a warehouse and in what order they should be sold. The person in charge of the structure would rather have a None be returned to them, rather than crash the entire program and be raised an exception. As humans, it is much better to interpret the None as a valid removal or peek of the structure, rather than crash the entire thing just because nothing was present.