Stacks and Queues

This lesson talks about operations on stacks and queues.

Stack

Stack is a well-known data-structure, which follows *first in, last out* paradigm. It offers `push` and `pop` operations. We'll examine the complexity of these operations when stack is implemented using an array or a linked list.

Stack using Linked List

A stack can be implemented using a linked list. New items are always appended and removed at the head of the list. We discussed in the linked list section that appending an item to the head of a linked list takes constant time. Therefore, both `push` and `pop` operations take constant or O(1) time.

However, note that when we use a linked list we are still using an extra pointer in each node of the list to keep track of the next node in the chain. That additional cost allows us to theoretically have a stack with an infinite capacity. We can keep on pushing onto the stack for as long as the computer memory allows. Instead of ***O(n)*** space, we'll be using ***O(2n)*** space which is still ***O(n)***.

Stack using an Array

Stack can also be implemented using an array. We can keep an integer pointer ***top*** to keep track of the top of the stack and increment it whenever a new push occurs. Poping elements is also a constant time operation because we can easily return ***array[top]*** followed by a decrement of the ***top***.

While using an array, we are saving the cost of having an additional pointer. At the same time, the size of the stack is limited to the initial size of the array. If the stack is completely filled then we'll need to create a bigger array, and copy over the stack to the new array, and discard the old one. A dynamic array may be used in this scenario to amortize the cost.

Another disadvantage of using the array is that it ties up memory if the stack is lightly used. Assume you created a stack thinking in the worst case it could have a thousand elements. The backing array for the stack will also be of size 1000. Say the stack is only filled with 100 elements for 90% of the time, then 90% of the memory isn't being used 90% of the time. This scenario wouldn't happen in case the stack was implemented using a linked list.

Queue

The queue is a type of data-structure that lets the first element enqueued also be the first element dequeued or *first in, first out*. Again, we can implement a queue using either an array or a linked list. It offers two methods: `enqueue` and `dequeue`.

Queue using Linked List

We can implement a queue using a linked list, where elements are always dequeued from the head and enqueued at the tail. We'll need two pointers - one head and one tail - to keep track of the front and back of the queue. Both the enqueue and dequeue operations take constant time. However, the price paid is in terms of the extra *next* pointer variable that will be part of each node of the list.

Queue using Array

Implementing a queue using an array is slightly tricky, since it requires appropriate manipulation of the front and the end pointers to keep track of the head and the tail of the queue respectively. However, both the operations take constant time since accessing any array element takes constant time.

The same tradeoffs exist that we discussed when implementing a stack using an array or a linked list. The queue can grow without bounds if implemented via a linked list, but not with an array. Using a linked list will incur an additional cost of a pointer node but will give flexibility in growing the queue without copying.

Summary

In crux, arrays should be used to implement queues or stacks when the maximum size of each is known beforehand. Also, it would help to know the expected utilization of the data-structure so that one is aware of how much memory will be tied up and may not be used.

Linked list should be used to back an array or stack when the maximum size isn't known beforehand. It'll also make sense to use a linked list when maximum size is known in advance but is only expected as a spike (i.e. the stack or the queue will not hold the maximum or near maximum number of elements for most of the time.)