IT5003 Oct-Dec 2023
Data Structures and Algorithms

Tutorial+Lab 03
Linked List, Stack, Queue, Deque

Document is last modified on: July 18, 2023

1 Introduction and Objective

For this tutorial, you will need to (re-)review `https://visualgo.net/en/list?slide=1` (to last slide 9-6) about List ADT and most of its variations (SLL, Stack, Queue, DLL, and Deque) as they will be the focus of today's tutorial.

2 Questions

Linked List, Mini Experiment

Q1). Please use the 'Exploration Mode' of `https://visualgo.net/en/list` to complete the following table (some cells are already filled as illustration). You can use use the mode selector at the top to change between (Singly) Linked List (LL), Stack, Queue, Doubly Linked List (DLL), or Deque mode. You can use 'Create' menu to create input list of various types.

Mode $\rightarrow$ $\downarrow$ Action	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	not allowed	not allowed	$O(N)$	not allowed
peek-front()	$O(1)$				
peek-back()					$O(1)$
insert(0, new-v)				$O(1)$	
insert(N, new-v)					$O(1)$
insert(i, new-v), $i \in$[1..N-1]		not allowed			
remove(0)			$O(1)$		
remove(N-1)		not allowed			
remove(i), $i \in$[1..N-2]				$O(N)$	

You will need to fully understand the individual strengths and weaknesses of each Linked List variations discussed in class in order to be able to complete this mini experiment properly. You can assume that all Linked List implementations have head and tail pointers, have next pointers, and only for DLL and Deque: have prev pointers.

Points to be QUICKLY discussed in class (no need to be long-winded here):

- The comparison of these 9 possible actions in 5 modes of Linked List
- The fact that the specialized ADTs: stack, queue, and deque take advantage of the fastest $O(1)$ operations of the underlying data structure:
  - stack uses `insert(0, new-v)/remove(0)` of Singly Linked list for `push(new-v)/pop()` respectively,
  - queue uses `insert(N, new-v)/remove(0)` of Singly Linked List for `enqueue(new-v)/dequeue()` respectively,
  - deque uses `insert(0, new-v)/insert(N, new-v)/remove(0)/remove(N-1)` of Doubly Linked List for `push-front(new-v)/push-back(new-v)/pop-front()/pop-back()`, respectively.

Mode → ↓ Action	Singly Linked List	Stack	Queue	Doubly Linked List	Deque
search(any-v)	$O(N)$	not allowed	not allowed	$O(N)$	not allowed
peek-front()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
peek-back()	$O(1)$	not allowed	$O(1)$	$O(1)$	$O(1)$
insert(0, new-v)	$O(1)$	$O(1)$	not allowed	$O(1)$	$O(1)$
insert(N, new-v)	$O(1)$	not allowed	$O(1)$	$O(1)$	$O(1)$
insert(i, new-v), $i \in$[1..N-1]	$O(N)$	not allowed	not allowed	$O(N)$	not allowed
remove(0)	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
remove(N-1)	$O(N)$	not allowed	not allowed	$O(1)$	$O(1)$
remove(i), $i \in$[1..N-2]	$O(N)$	not allowed	not allowed	$O(N)$	not allowed

Q2). Assuming that we have a List ADT that is implemented using a Singly Linked List with both head and tail pointers. Show how to implement two additional operation:

1. `reverseList()` that takes in the current list of N items $\{a_0, a_1, ..., a_{N-2}, a_{N-1}\}$ and reverse it so that we have the reverse content $\{a_{N-1}, a_{N-2}, ..., a_1, a_0\}$. What is the time complexity of your implementation? Can you do this faster than $O(N)$?
2. `sortList()` that takes in the current list of N items and sort them so that $a_0 \leq a_1 \leq ... \leq a_{N-2} \leq a_{N-1}$. What is the time complexity of your implementation? Can you do this faster than $O(N \log N)$?

For `reverseList()`, we can use either one of these possible answers. Note to tutor: If a student answers one version in class, the tutor will challenge him/her to think of the other way:

Option 1: We iterate through the N items of the list in $O(N)$ time and insert them into the head of a **new** list (initially empty), $O(1)$ each time. Finally we return the new list, which is the reverse of the original list(, and erasing the original list). This is $O(N)$ overall.

Option 2: We can also use three adjacent pointer methods: pre, cur, aft that goes together from head to tail, along the way, we make cur.next points back to pre, then set pre, cur, aft to cur, aft, aft.next, respectively. Stop when aft = None. Then, we swap the head and tail pointers. This is $O(N)$ overall.

Option 3: We can also use recursion to go deep from the head element to the tail element in $O(N)$. Then, as the recursion unwinds, we can reverse the link from $u \rightarrow v$ into $v \rightarrow u$. Lastly, we swap the head and tail pointers.

It is not possible to do better than $\Omega(N)$ as there are N items that will need to change their (next) pointers in the `reverseList()` operation. In this module, we do not discuss this Ω (lower bound) notation in details.

PS: Python list (which is not really a Singly Linked List), has indexing operation, e.g., $l[:: -1]$ will reverse Python list l. There is also LIST(REVERSED(L)) operation. Both are $O(N)$ too as discussed above. However, later we will see that for some rare application (e.g., Kattis - integerlists), we can use Doubly Linked List and $O(1)$ 'virtual reverse'.

For `sortList()`, although not that intuitive and we are unlikely to go this route if we ever need to sort a list (we will likely use an array/a vector instead), we can actually implement almost all? sorting algorithms that we have learned so far: Bubble/Insertion/Selection/Merge/Quick sort (be careful of SLL vs DLL, e.g., Insertion sort needs to go back and cannot be done using SLL). To achieve $O(N \log N)$ performance, we will rely on either Merge sort or (Randomized) Quick sort on Linked List. Note to tutor: Pick just one (anything that your student in your group doesn't say, e.g., if they say: implement Merge sort, you challenge him/her to use the other version).

Option 1: Merge sort on Linked List: simple, during the merge process, we create a new linked list to merge two sorted shorter lists. Also $O(N)$ for this merge of two sorted linked lists.

Option 2: (Randomized) Quick sort on Linked List: not that hard either (assuming no duplicate first), we can pick a random element as pivot (probably in $O(N)$, not in $O(1)$) and create a new linked list with just that pivot. We then grow to the left (add Head) if the next other element is smaller than the pivot or grow to the right (add Tail) otherwise. Also $O(N)$ for this partition on linked list.

Same argument as with the lower bound of sorting: It is not possible to do better than $\Omega(N \log N)$ for comparison-based sort, otherwise if we can, we will then use Linked List to do 'faster than $\Omega(N \log N)$'

comparison-based sorting instead of using the usual array/vector.

Python Implementations: list, stack, and queue (deque)

Q3). To strengthen the discussions in the lecture, answer the following sub-questions:
a). What is Python list really is? Is it a Singly Linked List? A Doubly Linked List? or?
See `https://docs.python.org/3/tutorial/datastructures.html#more-on-lists`
b). How to implement an efficient Stack in Python?
See `https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-stacks`
c). How to implement an efficient Queue in Python?
See `https://docs.python.org/3/tutorial/datastructures.html#using-lists-as-queues`

Python list is not really a Singly Linked List. It is more like a resizeable array. It only has efficient $O(1)$ append (to the back/insert after tail) and $O(1)$ pop (from the back/delete tail). Python list can be (single) indexed in $O(1)$ and can be sliced in $O(\textit{range-of-indices})$. Remember all these time complexity details.

We can use Python list (as above) to implement an efficient Stack ADT but we need to use its back side as the top so that we can do $O(1)$ append/push into stack and $O(1)$ pop/pop from stack.

If we use Python list to implement the Queue ADT we will have efficient $O(1)$ append (enqueue to the back of the queue) but slow $O(N)$ pop(0) (dequeue from the front of the queue). Swapping the front/back orientation of the queue does not help. For IT5003 level, the easiest to have efficient Queue is to actually use Python deque (`from collections import deque`) that has both efficient $O(1)$ append (enqueue to the back of the deque) and efficient $O(1)$ popleft() (dequeue from the front of the queue). The details of Python deque is not really discussed in IT5003 to keep the course manageable (NEW: maybe discussed in the additional recitation class), but it is good to know this differences to avoid Time Limit Exceeded (TLE) for queue-based questions.

Hands-on 3

TA will run the second half of this session with a few to do list:

- PS2 Quick Debrief,
- Do a sample speed run of VisuAlgo online quiz that are applicable so far, e.g., `https://visualgo.net/training?diff=Medium&n=5&tl=5&module=list`.
- Finally, live solve another chosen Kattis problem involving a List.

Do PS2 quick debrief according to the context of your lab group.

Do VisuAlgo Online Quiz sample run in medium setting.

`https://nus.kattis.com/problems/integerlists`.

It is a ‘medium-level’ (deque, link this from Q3 discussion above) problem. Notice that reversing a list is ‘slow’, $O(N)$, link this from Q2 above. So what if we ‘virtually simulate the reverse process (without actually reversing the entire list)’? Then, ‘Drop’ first element becomes Drop last element if the list is actually reversed. Now notice that if we do this, we need efficient way to pop elements from EITHER side of the list, hence deque is the best as it supports $O(1)$. See short integerlists.py for the solution.

Problem Set 3

We will end the tutorial with high level discussion of PS3 A+B.

PS3 A (/bracketmatching) is a classic ‘bracket matching’ using a Stack that is discussed in class (see `https://visualgo.net/en/list?slide=4-4`). It is so classic and almost everyone are expected to be able to solve this without much problem. This is because the second task is going to be quite challenging.

PS3 B (/congaline) is going to stress-test students knowledge about Linked List. All operations should be $O(1)$ to pass all subtasks. Attempt this task gradually, i.e., do subtask 1 first (very simple if you understand what it means), then subtask 2 (concentrate on getting P-operations correct), then subtask 3 (concentrate on getting R-operations correct, even with slower solution). The hardest operation type is the C-operations. The key part is that we need to quickly know where anyone’s partner is located. What is the best way to do this?