CS2040S Semester 1 2023/2024

Data Structures and Algorithms

Tutorial+Lab 03

Linked List, Stack, Queue, Deque

For Week 05

Document is last modified on: September 6, 2023

MODAL ANSWER IS FOR OUR CLASS ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

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 all its variations (compact array (or vector/ArrayList-)based List - from tut01.pdf, SLL, Stack, Queue, DLL, Deque) as they will be the focus of today's tutorial.

2 Tutorial 03 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 → ↓ 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.

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.

Stack, Queue, or Deque

Q3). In the Lisp programming language, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expressions are enclosed in parentheses. There is only one operator in a pair of parentheses. The operators behave as follows:

- `( + a b c )` returns the sum of all the operands, and `( + )` returns `0`.
- `( - a b c )` returns `a - b - c` and `( - a )` returns `0 - a`,
  i.e., the minus operator must have at least one operand.
- `( * a b c )` returns the product of all the operands, and `( * )` returns `1`.
- `( / a b c )` returns `a / b / c` and `( / a )` returns `1 / a`, using double division.
  The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation, e.g. the following is a valid Lisp expression: `( + ( - 6 ) ( * 2 3 4 ) )`. The expression is evaluated successively as follows: `( + -6.0 ( * 2.0 3.0 4.0 ) )`, then we have `( + -6.0 24.0 )`, and we finally have `18.0`.

Design and implement an algorithm that uses up to 2 stacks to evaluate a legal Lisp expression composed of the four basic operators, integer operands, and parentheses. The expression is well formed (i.e., no syntax error), there will always be a space between 2 tokens, and we will not divide by zero. Output the result, which will be a double-precision floating point number.

This algorithm uses exactly two stacks: The first is used to store the tokens read from the expression one by one until the operator ')'.

The second stack is used to perform a simple operation on the operands in the innermost expression already in the first stack. The tokens are pushed into the second stack in reverse order. Therefore, tokens from the second stack are popped in the order of input. The calculated result is then pushed back into the first stack.

Tutor will demonstrate a few working examples and/or show the working code for this problem. Familiarity with this stack-based solution is important to solve PS3B - bungeebuilder.

Hands-on 3

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

- Do a short debrief of PS2 (after grading),
- Very quick review of Java LinkedList, Stack classes and Queue, Deque interfaces,
- 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 list ADT.

1-2 slides for PS2 debrief (in high level only).
1-2 slides for Java API demonstrations, just show a few techniques or refer students to relevant references, e.g., `https://github.com/stevenhalim/cpbook-code/tree/master/ch2/lineards`.
Show off in 5m :), this module is still very easy...
Kattis /joinstrings, the easiest is to use `splice` method of C++ std::list.
But in Java, this will take a bit of more work...

Problem Set 3

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

PS3 A (/sim) is an easy/medium Doubly Linked List problem. For the first week (early Week 05) of this PS3, at least get PS3 A subtask 1 done (there is no '['). This subtask 1 is just a Stack simulation (pop the last character from the stack if we see a '¡', do nothing if we see a ']', or push into stack for any other character). We print the content of the stack backwards at the end (as the content of a LIFO stack is reversed). The presence of '[' causes issue as a simple stack is not sufficient. We need a ... DLL (alternative ways exist).

For PS3 B (/janeeyre), the initial troublesome part is the input parsing (for other programming languages). Afterwards, this becomes a (min) Priority Queue simulation problem. Just read the problem description carefully to simulate the required book reading process until you read 'Jane Eyre' book. By the time we reach this tutorial, students have not studied Priority Queue ADT, so at least work on the I/O part.