National University of Singapore
School of Computing
IT5003: Data Structure and Algorithm
Semester I, 2019/2020

Tutorial x Lab 3 Selected Solution
List ADT & Linked List

About selected solution: We will released solution to selected question only. Mainly for questions that are hard / information heavy. For the rest of questions, you should take notes during the discussion.

1. [Using List ADT] In scientific experiments, we frequently have to deal with data of the form (X, Y), i.e. a point on a 2-D plane. Let's designed a new **dataset** ADT to store and manipulate these data. Below are a few design decisions:
   - Each data point is represented by a **Python `tuple`** for simplicity, e.g. (1.23, -4.56).
   - The **`DataSet`** class has minimally the following two operations:

```
class DataSet():
    """
    DataSet ADT implementation using List ADT (any implementation)
    """
    def addPoint(self, newPoint ):
        """ Add a new point [newPoint] to the data set.
        [newPoint] should be a 2-tuple (x, y)
        """
        pass
    def findNearestPoint(self, refPoint ):
        """ Return the point nearest to [refPoint] in the data set.
        Return None if the data set is empty.
        """
        pass
```

   - In the **`DataSet`** class, we have decided to use **List ADT** as the internal data structure, i.e. we are using a simple ADT to build something more specialized. You can use either the **`ListArray`** or the **`ListLinkedList`** implementations.

   Questions to discuss:

   a. Give pseudo-code (or actual code) for the two operations and the constructor of the **`DataSet`** class.
   b. Is **`List`** ADT a good choice to implement the **`DataSet`** class in this case?
   c. Doe the specific version of **`List`** ADT implementation impacts the performance of **`DataSet`** class?

ANS:

a. <Omitted>

b. It is reasonable, but with slight mismatch in term of functionality. The order in which the points added is not important to the DataSet (as it is designed). So, using an ADT where position matters cause some unnecessary coding overhead (e.g. always need to get the size of list in order to insert at the end etc).

c. Yes. (How?)

2. [Linked List Implementation of List ADT] Analyze the efficiency of linked list implementation of List ADT in term of time and space used.

ANS:

Time efficiency of Singly Linked List implementation

Retrieval:

Only the first node (the head node) is directly accessible, all other nodes require traversal ("hops" along the list to reach the target node).
(What is the big-O time complexity for the best case and worst case?)

Insertion:

Operations needed: Change 2 pointers + Traversal to the right node
(What is the big-O time complexity for the best case and worst case?)

Deletion:

Operations needed: Change 2 pointers + Traversal to the right node
(What is the big-O time complexity for the best case and worst case?)

Comments: Traversal is usually more costly than shifting item in term of time, when the number of item to be shifted is small and/or the item itself is very simple (e.g. a single integer). However, if the item is complicated (e.g. big structure with many fields) and/or the number of items is big, then hopping is much more economical.

Space efficiency of Singly Linked List implementation

In most systems, the size of a pointer is the same as a single integer. Hence a linked list of 50 integers takes up the same space as 100 integers array (i.e. the overhead of using linked list is double that of array). Note that this ratio will decrease if we store bigger item in the list (i.e. the overhead of linked list will decrease compared to array implementation).

However, there is **no wasted space** for linked list (allocate new node only when required). Shifting elements due to expansion in size is also **not needed** for linked list.

4. [**Variations of Linked List**] Linked list, with its many variations, can be quite confusing. One way to understand the differences between the variations is to make a comparison table. Try to fill in the following table, the columns are:

- **Maximum "hops"**: Which node in a **N-nodes** list is the hardest to reach? What is the number of "hops" required to reach this node?
- **Memory space**: What is the memory requirement for N-nodes of list? You can assume that each item takes **X** amount of space and each pointer takes **Y** amount of space.

You can assume all linked list variations has a **head pointer** pointing to the first node in the list **unless otherwise specified**.

ANS:

The analysis for the simpler types of linked lists are given.

Linked List Variation	Maximum "Hops"	Memory Space
Singly linked list	*Last Node. Requires N-1 hops.*	*N*(X+Y) + Y (for the head pointer)*
Singly linked list with head and tail pointers	The 2nd Last node. Requires N – 2 hops.	N*(X+Y) + 2Y
…	…	…

Other than the above points, you should also consider other requirements:

- What is the most common location to operate on? E.g. if the last node is being accessed repeatedly, then doubly linked list wouldn't help.
- Ease of coding, some variations provide ease of coding rather than actual time efficiency. E.g. singly linked list with dummy head code simplify the coding but does not improve the efficiency over normal singly linked list.
- The difference in memory space is not huge, so it is usually not a point of consideration when choosing the appropriate variation.

5. **[Doubly Linked List**) We have covered the general insertion and removal in doubly linked list in lecture. Let us now use it as an alternative implementation of the List ADT. Define a new class **ListDLL** as the new implementation class for List ADT. You are free to try out the complete implementation (could be helpful for your lab submission questions ☺). In the tutorial, we will only cover the following in details:
    - The **ListDLL** class constructor
    - The **insert**() method
    - Is doubly linked list helpful in List ADT? Why?

ANS:

In general, to figure out the code for a linked list implementation (any variations), we can take the following steps:

i. Figure out the logic for the general case (e.g. insertion/deletion in the middle of the list)

ii. With the logic in (i), pay attention to **when the code will fail**. A common culprit is that the code attempt to access a None reference, e.g.

```
ptr.something
```

This will fails (runtime error) when ptr is a None. So, you can concentrate on checking **when will ptr be ill defined?**

Using this approach, we can discover several cases to be handled separately in the insertion, namely ***empty list***, ***first location*** and ***end of list***.