LinkedList & Queue & Stack

1. Array vs. LinkedList

Similar:

1. Both *Arrays* and *Linked List* can be used to store linear data.

Difference:

1. The Size of the *Array* is fixed, The Size of the *Linked List* is dynamic.
2. In *Array*, inserting or removing items from the beginning or from the middle of the *Array* is expensive, because the elements need to be shifted over. But in *Linked List* we do not need to shift.
3. *Array* have better cache locality that can make a pretty big difference in performance.
3. Unlike *Array*, in *Linked List*, the elements are not placed contiguously in memory:

- Each element consists of a node that stores the element itself and also a reference that points to the next elements.
- So it needs Extra memory space for a pointer is required with each element of the list.

4. In *Linked List*, random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.

2. ArrayList vs. LinkedList vs. Vector

Similar:

ArrayList and *LinkedList* both implements List interface and their methods and results are almost identical

Difference:

1. *ArrayList* is implemented as a resizable array. As more elements are added to *ArrayList*, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since *ArrayList* is essentially an array
2. *LinkedList* is implemented as a double linked list. Its performance on add and remove is better than *Arraylist*, but worse on get and set methods.
3. *Vector* is similar with *ArrayList*, but it is synchronized.

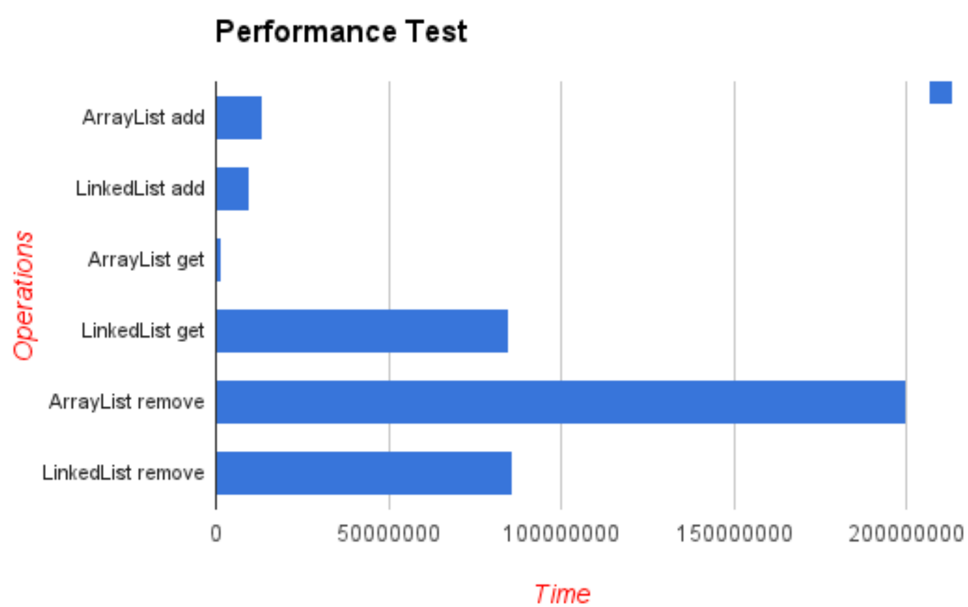
Comparison:

ArrayList is a better choice if your program is thread-safe. *Vector* and *ArrayList* require more space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50% of its size each time. *LinkedList*, however, also implements Queue interface which adds more methods than *ArrayList* and *Vector*, such as offer(), peek(), poll(), etc.

Performance:

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

- *ArrayList* has O(n) time complexity for arbitrary indices of add/remove, but O(1) for the operation at the end of the list.
- *LinkedList* has O(n) time complexity for arbitrary indices of add/remove, but O(1) for operations at end/beginning of the List.



3. LinkedList Basic

ListNode 1 —> ListNode 2 —> ListNode 3 —> ListNode 4 —> ListNode 5 —> null

1. Never loose your access to new and old head —> head is the only way to access LinkedList.
2. When you do de reference, make sure the null pointer.
3. Always add a null at the end of singly / doubly LinkedList.

Corner Case —> head === null || head.next === null;
Main Structure —> While loop (consider when to exit);

4. Question

Q1. Reverse LinkedList (LC206) / Reverse LinkedList by pair three k (LC24 LC25) / Reverse Linked List by range (LC92)

- Reverse LinkedList (LC206)
    - Solution 1: Iteration (95s)
    - Solution 2: Recursion (119s)
- Reverse LinkedList by pair three k (LC24 LC25)
    - LC24: Recursion(129s)
    - LC25:
        - Solution 1: Tail Recursion
        - Solution 2: Iteration (Head Recursion)

Q2. Find Middle Node (1/2) in Linked List —> 1/3, 1/5, 3/7

- Solution 1: Dummy node;

```
1  —>  2  —>  3  —>  4  —>  5  —>  6  —>  null
s  --------------------------------------------------
f  --------------------------------------------------
         s  -------------------------------------------
                  f  ----------------------------------
                  s  ----------------------------------
                                f  -------------------
                         s  -------------------------
```

```
                                                  f
       f !== null && f.next !== null;
       s stop at 4

       1  —>  2  —>  3  —>  4  —>  5  —>  null
       s  ------------------------------------------
       f   ------------------------------------------
            s  ----------------------------------
                 f  -------------------------
                 s  -------------------------
                           f  ----------
       f !== null && f.next !== null;
       s stop at 3
```

- Solution 2: f !== null && f.next.next !== null

Q3. Check whether a LinkedList has a cycle

- Intersection of Linked List (LC160)
- Return Size of Cycle
- Linked List Cycle 1 & 2 (LC141 & LC142)
- Return Kth position after enter node of cycle

Q4. Insert a node into Sorted Linked List

- Insertion Sort List (LC147) (固定一边，从前向后扫，找到合适的然后 swap)
- Sort List (LC148) (find mid —> merge)

Q5. Delete a node/value from Linked List & Remove duplicates

- Remove Linked List Elements (LC203)
  - Solution 1: With Dummy Node

    - Solution 2: ??? Without Dummy Node
- Remove Duplicates from Sorted List (LC083)
    - Same with Last Question, but without dummy, and the condition of while loop is different;
- Delete Node in a Linked List (LC237) (node.val => node.next.val, node.next = node.next.next)

Q6. Merge two Sorted Linked List (LC021)

- Solution 1: Recursion
- Solution 2: Iteration

Q7. Reorder List (LC143)

- Solution 1: Extra O(n) Space: Reverse the whole Linked List and pari merge.
- Solution 2: Find Middle —> Reverse Half part —> Merge
    - Merge Function:
        - fast = head;
        - slow = midHead;
        - while(fast && slow) {
            - next1 = fast.next;
            - next2 = slow.next;
            - slow.next = fast.next;
            - fast.next = slow;
            - fast = next1;
            - slow = next2;
        - }

Q8. Odd Even List (LC 328)

```
        head
===>   1   —>    2    —>    3    —>    4    —>    5    —>    6    —>    null
org    oHead     eHead
        oTail        eTail
                     node
===> while(node && node.next)
===>   1   —>    2    —>    3    —>    4    —>    5    —>    6    —>    null
        oHead
```

Q9. Partition List (LC 086)

Q10. Plus One Linked List (LC 369) & Add Two Numbers (LC 002)