1. [List Performance] In the lecture notes, we have learned two ways to implement the list, namely list implementation via array and list via linked lists. Try to recall the time efficiency and space efficiency for both ways. In the following example, we want to test the performance of the two types of Java built-in List, namely **ArrayList** and **LinkedList**.

   a. Please try the following code and explain why.

   ```java
   import java.util.*;
   class ListTester {
       List<String> list;
       public ListTester(List<String> list) {
           this.list = list;
       }
       public void TestAdd(int iteration) {
           // Pre-Processing (un-timed)
           long sTime = System.currentTimeMillis(); // Starting Time
           // Put Your Experiment Here:
           for (int i = 0; i < iteration; i++)
               list.add(i + "");
           long eTime = System.currentTimeMillis(); // Ending Time
           // Post-Processing (if any)
           // Timing:
           System.out.println(eTime - sTime);
       }
       public static void main(String[] args) {
           ListTester lTest1 = new ListTester(new LinkedList<String>());
           ListTester lTest2 = new ListTester(new ArrayList<String>());
           lTest1.TestAdd(100000); // Value might need to be changed
           lTest2.TestAdd(100000);
       }
   }
   ```

   **Suggested Solution:** The value depends on the machine you are running it from. However, some clear trend is seen. After certain value, ArrayList is slower in the adding (even when the adding is at the end; i.e. no re-ordering).

   The reason why adding in ArrayList is slower, it is because of the doubling (or adding by 50% size) when the ArrayList is full. Since we are adding a lot of data, eventually, the copying during the doubling (or adding by 50% size) will be a significant overhead in adding element into the ArrayList.
b. The code given now only has the test for “Add”. Create functions similar to `TestAdd()` for the following test to compare the results for `ArrayList` and `LinkedList`:

i. Add to Head: add only to the head of the List (index = 0)

ii. Remove from Head: remove only from the head of the List (index = 0)

iii. Get from Head: get (without removing) only from the head of the List (index = 0)

iv. Get: get (without removing) from `index=0` to `index = size-1`

Explain the differences in the running time. Pre-processing is for initializing the List (if needed), and Post-processing is for collecting value or removal of values from List (if needed).

Suggested Solution:

```java
public void TestAddHead(int iteration)
{
    // Pre-Processing (un-timed)
    long sTime = System.currentTimeMillis(); // Starting Time
    for(int i=0; i<iteration; i++) {
        list.add(0, i++); // Add to Index 0 only
    }
    long eTime = System.currentTimeMillis(); // Ending Time
    // Timing:
    System.out.println(eTime-sTime);
}

ArrayList perform even worse in this case due to re-ordering (i.e. shifting the element by 1 to the right, or higher index).

public void TestRmHead(int iteration)
{
    // Pre-Processing (un-timed)
    for(int i=0; i<iteration; i++) {
        list.add(0, i++); 
    }
    long sTime = System.currentTimeMillis(); // Starting Time
    for(int i=0; i<iteration; i++) {
        String tmp = list.remove(0);
    }
    long eTime = System.currentTimeMillis(); // Ending Time
    // Timing:
    System.out.println(eTime-sTime);
}

LinkedList saw an improvement in performance as creating new memory space is slower than re-assigning pointer (this probably hasn’t gone through memory reclamation by garbage collector). However, the ArrayList is still as bad due to re-ordering.
```
public void TestGetHead(int iteration)
{
    // Pre-Processing (un-timed)
    for(int i=0; i<iteration; i++) {
        list.add(0, i+"".signIn);
    }
    long sTime = System.currentTimeMillis(); // Starting Time
    // Put Your Experiment Here:
    for(int i=0; i<iteration; i++) {
        String tmp = list.get(0);
    }
    long eTime = System.currentTimeMillis(); // Ending Time
    // Post-Processing (if any)
    // Timing:
    System.out.println(eTime-sTime);
}

Both ArrayList and LinkedList perform approximately the same, although ArrayList might be slightly faster. It is reasonable to be faster as the memory location of index 0 is explicitly known by the variable in ArrayList. On LinkedList, we might have to load the head value first although we will be loading the same value over and over again.

public void TestGet(int iteration)
{
    // Pre-Processing (un-timed)
    for(int i=0; i<iteration; i++) {
        list.add(0, i+"".signIn);
    }
    long sTime = System.currentTimeMillis(); // Starting Time
    // Put Your Experiment Here:
    for(int i=0; i<iteration; i++) {
        String tmp = list.get(i);
    }
    long eTime = System.currentTimeMillis(); // Ending Time
    // Post-Processing (if any)
    // Timing:
    System.out.println(eTime-sTime);
}

ArrayList performs the same as TestGetHead while LinkedList performance suffers considerably. This is because getting element in the middle of a LinkedList requires traversal of the LinkedList to the specified index. Thus, the average traversal required is half of size of the LinkedList. Multiply this by the number of iterations (which is equal to the size of LinkedList), we have a very large number.

ArrayList retrieval only require a single arithmetic operation which is to calculate the memory location of the data. Because ArrayList uses Array which store data sequentially in memory, retrieval of an element only involve calculating the displacement of that element from the starting index. Thus, it is much faster in comparison to LinkedList.
2. [LinkedList Operations]

Write a function to split the Linked List into halves. In other words, split the list in the middle. For example,

Linked List before split: [1 2 3 4 5 6 7 8 9]

Two Linked Lists after split: [1 2 3 4 5] and [6 7 8 9]

Take note of the following restrictions:

1. You are not allowed to create new node;
2. You can traverse the list once only;
3. You cannot count or use the size() method to find the number of nodes in the list. You are not allowed to use linked list API.

```java
// Pre : backSublist are initialized
// Post: head points to backSublist
BasicLinkedList <E> split() {
    // Your code here.
}
```

Suggested Solution:

The idea is to use multiple pointers to traverse the list at the same time but with different speed.

```java
// Pre : backSublist are initialized
// Post: head points to backSublist
BasicLinkedList <E> split() {
    ListNode <E> temp1 = head, temp2 = head;
    BasicLinkedList <E> backSublist = new BasicLinkedList <E> ();
    while(temp1.next != null && temp1.next.next != null){
        temp1 = temp1.next.next;
        temp2 = temp2.next;
    }
    // Remainder is backSublist
    // set next of temp2 to NULL so that
    // frontEnd do not join with backEnd
    if(temp2.next != null){
        backSublist.head = temp2.next;
        temp2.next = null;
    }
    return backSublist;
}
```
3. **[Set implementation via Linked List]**

Write a Set class implemented via LinkedList. A set is a list that contains no duplicate elements. For example

1  2  2  3  4  is not a set.
1  2  3  4  is a set.

Sets with the same elements but in different orders in the lists are considered to be equal.
For example, 1  3  4  5 and 5  3  1  4 are equal.

Write the **mySet** class to include following operations:

**Operations involve only one set:**
1. Set insertion: add one element to this set if it is not already present.
2. Set remove: remove certain element to this set if it is currently in the set.
3. Set clear: remove all the elements from the set;

**Operations involve two sets:**
4. Set addition: merge two sets together. (Duplicates must be removed.)
5. Set difference: return set s after removing elements that are in set t.
6. Set intersection: return set s keeping only elements that are in t.

Suggest Solution: See Tut04ex3.java

---

4. **[Application of Linked List]**

Simple graph contains nodes and edges. An edge exits if there is direct connection between two nodes. Assume each node in the graph has at least one incident edge.

We can use edge list to represent this graph:

1,2
1,4
2,3
3,4
3,5
5,6

Implement a graph class using Linked list. Input the edge list to instantiate object (e.g. 1  2  1  4 2  3  3  4  3  5  5  6). Create a method called isEdgeExist() which checks whether there is an
edge between two nodes. It returns true if two nodes queried have an edge between them, otherwise
return false. You may implement other methods if necessary.

Suggest Solution: See Tut04ex4.java

The graph can be represented as follows.
The nodes form a linked list (the first column on the left, which we may refer to as the main list).
For each node, it also points to a list which contains all the nodes connected with it (which we may
refer to as adjacent list). To check whether two nodes i, j are connected, we can search for one of
the nodes, i, in the main column, and then search for the other node, j, in i’s adjacent list.