Comparing List Implementations

How to compare algorithms runtime?

1. Time the running of the algorithm, as in Asg 5. One complication is the speed of the computer.
2. Do a formal analysis of the algorithm. Recall big-O notation?

We will informally classify algorithms using the runtime categories. For discussing the efficiency of our list operations, we measure the algorithms in terms of the number of elements in the collection.

Classifications

Algorithm runtimes can be classified with the following categories:

1. Constant. A code "segment" that does not loop over our collection (O(1)). Ex: for n=100 elements, the algorithm takes a single step in list.

1. Linear. A code "segment" that loops over the collection but without any nested loops (O(n)). Ex: for n=100, elements it would take something like 100 steps through the collection.
2. Quadratic. A code "segment" that loops over the collection with a nested loop (bit only a single nesting) ($O(n^2)$). Ex: for n=100 elements it would take something like 100*100 = 10000 steps through the collection.

Other classifications are: $O(\log_2(n))$ (ex: binary search), $O(n * \log_2(n))$ (ex: sorting) and really large runtimes like $O(2^n)$ and O(n!)

Classifications of List operations for LinkedList and ArrayList

For basic list operations:

Operation/Algorithm	LinkedList	ArrayList
add(x)	constant	constant*
add(pos, x)	linear	linear
remove(pos)	linear	linear
get(pos)	linear	constant
set(pos, x)	linear	constant
clear()	constant**	linear

* expand doens't run each time…. for a decent expand factor this will be constant overall.

** maybe, what about garbage collection?

For some simple list algorithms (see below)

Operation/Algorithm	LinkedList	ArrayList
initializeWithAppend(list)	linear	linear*
initializeAddAt0(list)	linear	quadratic
initializeAddAtMiddle(list)	quadratic	quadratic
initializeAddAtSizeMinus2(list)	quadratic	linear?
replaceFrontToBack(list)	linear	quadratic
replaceBackToFront(list)	quadratic	quadratic
maxUsingForLoop(list)	quadratic	linear
maxUsingTraversal(list)	linear	linear

Conclusions:

1. If random access is important then use `ArrayList`.
2. Always use traversal (iterator) when visited each item.
3. Sometimes linkedlist is better when add/remove at 0 for example.
4. Add during traversal for linked-list is efficient

Algorithms:

```
private static void initializeWithAppend(List<String> list) {
       for(int i = 0; i< SAMPLE_SIZE; i++)
           list.add("abc");
   }

   private static void initializeAddAt0(List<String> list) {
       for(int i=0; i<SAMPLE_SIZE; i++)
           list.add(0,"abc");
   }

   private static void initializeAddAtMiddle(List<String> list) {
       for(int i=0; i<SAMPLE_SIZE; i++) {
           int pos = list.size() / 2;
           list.add(pos, "abc");
       }
   }
```

```
private static void initializeAddAtSizeMinus2(List<String> list) {
    list.add("a");
    list.add("b");
    for(int i=2; i<SAMPLE_SIZE; i++) {
        int pos = list.size()-1;
        list.add(pos, "abc");
    }
}

private static void replaceFrontToBack(List<String> list) {
    for(int i=0; i<list.size(); i++)
        list.add(list.remove(0));
}

private static void replaceBackToFront(List<String> list) {
    for(int i=0; i<list.size(); i++)
        list.add(0, list.remove(list.size()-1));
}

private static String maxUsingForLoop(List<String> list) {
    if(list.size() == 0)
        throw new RuntimeException("List can't be empty");

    String max = list.get(0);
    for(int i=1; i<list.size(); i++) {
        String current = list.get(i);
        if(current.compareTo(max) > 0)
            max = current;
    }
    return max;
}

private static String maxUsingTraversal(List<String> list) {
    if(list.size() == 0)
        throw new RuntimeException("List can't be empty");

    list.reset();
    String max = list.next();
    while(list.hasNext()) {
        String current = list.next();
        if(current.compareTo(max) > 0)
```

```
            max = current;
    }

    return max;
}
```