GALWAY-MAYO INSTITUTE OF TECHNOLOGY

Department of Computer Science & Applied Physics

SW2 – Data Structures & Algorithms
Collections and Lists

As software developers, we make extensive use of different types of collections when writing applications. Collections include, *inter alia*, array, list, set, queue, stack, tree, map and graph data structures. Each of these collections are implemented in their own way, typically using either an array-based or node-based approach, and have their own suite of algorithms or operations that can be used to add, remove, modify and search for elements.

The *Java Collections Framework* has implementations of all of these data structures and many others in the *java.util* package. Each of these main collection types is ***described by an interface***, e.g. *Collection<E>*, and implemented by one or more concrete classes, e.g. *ArrayList<E>* and *TreeSet<E>*. All the collections are ***generic*** and use angled brackets (<…>) to declare the collection to be of a specific type. In this lab, we will focus on collections and lists and see how these can be declared and used in a Java programme. We will also consider the time complexity of each of the operations that we use.

- Download the Zip archive ***dsaLabCollectionsAndLists.zip*** from Moodle under the heading ***Source Code for Collections and Lists Lab*** and extract the files into a folder on your hard disk. As the classes involved are all packaged, ensure that you create the correct directory structure and study the source code before you compile.
- Open the class *Runner.java* and complete the exercises below inside the *main()* method **after** the statement *long startTime = System.nanoTime();* on Line 12.

Declaring Lists and Collections

A *collection is a grouping together of elements of the same type* and a *list is an ordered collection*. Declare each of the following collections and use the intelli-sense of the IDE to see the set of methods available to you using ***list.*** (dot):

- **ArrayList**<String> list = new ArrayList<String>(); //Okay but too specific and verbose
- **ArrayList**<String> list = new ArrayList<>(); //Okay but still too specific
- **List**<String> list = new ArrayList<>(); //Much better. Can be assigned any list
- **Collection**<String> col = new ArrayList<>(); //Most flexible, but maybe too general
- **var** list = new ArrayList<String>(); //Can use var as type can be inferred (but it's fixed!)

For each of the statement above, add the following declarations and see if the code still compiles:

- list = new LinkedList<String>(); //Swap the ArrayList for a LinkedList
- list = new Vector<String>(); //Swap the LinkedList for a Vector

Adding Elements

The easiest way to add elements to a collection is to call the ***add()*** method parameterised with the element that you want to add. Add the following statements under your declaration **List**<String> list = new ArrayList<>(); from above:

```
for (String word : words) {
  list.add(word); //Adds the element word to the end of the collection called list.
}
```

- Execute the application and note the time. Change the implementation of the collection from an *ArrayList* to a *LinkedList* and a *Vector*, execute them again and note the time. Was there much difference? *When we call **add(element)** on an ArrayList, LinkedList or Vector the element is **always added to the end of the list in O(1) constant time**.*

Change the implementation of the loop as follows and repeat the process. Make sure to note the times carefully:

```
for (int i = 0; i < words.length; i++) {
  list.add(0, words[i]); //Adds the ith element in words to the start of list.
}
```

This time we are using the ***add(index, element)*** method to add an element to index 0, or the start of the list. You should notice that the *LinkedList* executes in a time similar to the ***add(element)*** method but the *ArrayList* and *Vector* classes are very slow in comparison. Adding to the start of a *LinkedList* is done in the same time as adding to the end, i.e. O(1). However, adding to the front of an array-based structure like *ArrayList* or a *Vector* requires all of the following elements to shuffle up to make space. As a result, the running time for adding to any index other than the end is $O(n^2)$.

Removing Elements

We can remove elements from a list by either ***specifying the value*** that we want to remove or ***specifying the index*** that we want to remove from. Make sure that you use the original ***for-in*** loop above to populate an *ArrayList* and then execute the following:

```
Collections.shuffle(list); //Randomly move the elements in the list around
for (String word : words) {
  list.remove(word); //Search for an remove word from list.
}
```

The statement *Collections.shuffle(list)* shuffles all the elements in the list using the Fisher-Yeats algorithm or something similar. For each element in the array, the method ***remove(element)*** is invoked. Each call to r*emove(element)* requires O(n) time because the element that we are looking for could be anywhere in the list (the worst case is at the end) because the shuffling. As we are calling *remove(element)* ***n*** times, this means that the overall running time is $O(n * n) = O(n^2)$, i.e. very slow. Changing the list to a *LinkedList* or a *Vector* won't change the poor running time.

Change the ***for*** loop to the following and execute the programme again for an *ArrayList*, *LinkedList* and *Vector*. This time, the programming will execute quickly as we are removing the from the last index in the list. The *ArrayList* and *Vector* should execute more quickly than the *LinkedList*.

```
for (int i = 0; i < words.length; i++) { //O(n)
  list.remove(list.size() - 1); //O(1) => Accessing an element at a known index
}
```

Change the ***remove(index)*** method to remove from index 0, i.e. the start of the list, as follows:

```
for (int i = 0; i < words.length; i++) { //O(n)
  list.remove(0); //O(1) or O(n²)  depending on the type of list…
}
```

The *LinkedList* executes much faster than the array-based *ArrayList* and *Vector* because adding or removing from the front or end of a doubly-linked list can be done on O(1) time. The total time for the *LinkedList* is O(n * 1) or O(n) to remove the n elements from the original array. The time for the *ArrayList* and *Vector* degrades to O(n * n^2) = O(n^3), which is really bad.

Searching for Elements

The generic method for searching a collection is the ***contains(element)*** method which returns true if *element* is located in the collection using the given implementation of *equals()*. As the ***contains(element)*** method needs to iterate over all elements in a linear collection in the worst case, the running time for ***contains(element)*** is O(n) for a *ArrayList*, *LinkedList* or *Vector*. Delete the *remove()* statements from *Runner.java* and execute the following:

```
boolean found = list.contains("Galway"); //Returns false as Galway is not in the list!
System.out.println(found);
```

We can also use the ***indexOf(element)*** operation to search a list, but not a collection. This method executes in O(n) time and returns the index of the element in the list or a -1 if it is not found:

```
System.out.println(list.indexOf("Galway")); // Returns -1 as Galway is not in the list!
```

Note that if we want to ensure a match for "galway" and "Galway", assuming that one of these element is in the list, then we need to change the implementation of *equals()* to be case-insensitive.

If we are using a **list**, we can access an element in O(1) time if we already know the index. This can be done with the ***get(index)*** method:

```
System.out.println(list.get(777));
```

This returns the 778th element in the list in constant time, i.e. it will execute in the same time if there are 1,000 or 1,000,000,000 elements in the list. However, if the index is greater than the array size minus one, then an *ArrayIndexOutOfBoundsException* will be thrown and your programme will more than likely crash.

Dynamically Expanding Lists

A linked list is created by chaining a set of nodes together either singly or doubly. Expanding the capacity of a linked list is therefore only a matter of adding a new node to the end of the chain and can be done in O(1) time. However, dynamically expanding collections that are array-based, like *ArrayList* and *Vector*, is not so straightforward, as array size is fixed when declared and cannot be dynamically altered.

The only way to dynamically change the size of an array is to create a new array of the necessary size and then copy all of the elements from the original array into the larger new one. This is exactly the approach used in the classes *ArrayList* and *Vector*, but care must be taken to ensure that this is done efficiently.

Examine the class Dictionary.java and look at the method *getSortedWords(boolean slow)*. This method copies all of the elements from the ArrayList into a String array and re-sizes the array if additional capacity is needed. As the initial capacity of the array is just 8 and the dictionary file contains 234,936 words, if we increase the array size by just one extra index at a time, we will need to create 234936 – 8 = 234,929 expanded arrays and copy a total of 8+9+10+11+12…234,936 = 27,597,579,508 words! This is very slow and has a running time of exactly $(n\ (n + 1) / 2) - 8$ or $O(n^2)$.

We can significantly reduce the running time of an expansion by applying the following simple heuristic (rule of thumb) when re-sizing the array. Instead of expanding the String array by one index at a time, we'll ***expand it by some constant factor greater than 1*** like 1.5. While this does not appear at first glance to make much of a difference, it will actually reduce the running time of the operation from $O(n^2)$ to **amortized** O(1). The term **amortized** means that the time appears to be significant at the start, but becomes insignificant as the size of the array increases. Consider the following table of results from my own computer(s) using a 2010 i3 processor and a 2020 i7 processor:

Resizing Method	**Expansions**	**Time(ms) on 3.2 GHz Intel Core i3 processor with Java 6.**	**Time(ms) on 1.2 GHz Quad-Core Intel Core i7 with Java 15.**
(a) 1 index at a time	234,929	15,090	4,092
(b) Increase by 1.5 each time	26	22	12
(c) None. Fixed size of 234936 indices.	0	4	15

Note: *c < b << a, proving that the cost of expanding the array with the (length * 3) / 2 heuristic is an amortized O(1) operation.*

Add the following statements to *Runner.java* under the statement *long startTime = System.nanoTime();* on Line 12:

```
boolean slow = true;
String[] words = d.getSortedWords(slow);
System.out.println("Time: " + (System.currentTimeMillis() - start));
System.out.println("Array contains " + words.length + " words.");
System.out.println("Amortized Time: " + slow);
```

- Execute the programme and check the running time. Change the value of the variable ***slow*** to ***true*** and run the programme again. You should a significant difference in the running time.