COMP254 Project 1: Comparing Vectors and Linked Lists

Brief summary of the assignment:

Write a 3–5 page memo to your (fictitious) colleagues at a small software company, explaining and justifying your recommendation for using either `Vector`s or `LinkedList`s in the implementation of your company's massively multiplayer game.

Details of the assignment:

Imagine you are part of a team at a small software company that is developing a new massively multiplayer game. Every player of the game has an *avatar* (a virtual person) that participates in the game's virtual world. One important feature of the game is that it permits very large numbers of avatars to assemble in the same location in the virtual world; a collection of avatars in the same location is called a *crowd*. While some members of your team are working on creating state-of-the-art graphics for displaying crowds, and others are working on novel techniques for simulating the behavior of crowds, you have been given the specific task of determining which of two possible data structures will be most suitable for storing data about large crowds within the application. Although no one in the team really knows how popular the new game will be, it is hoped that the game will eventually have hundreds of thousands of avatars. Therefore, the chosen data structure needs to work efficiently for crowds of only a few avatars up to at least 100,000 avatars.

The application is being written in Java. The team has determined that crowds will be stored using either the `Vector` class from the `java.util` package, or the `LinkedList` class, also from the `java.util` package. It is your job to perform some experiments to investigate whether `Vector`s or `LinkedList`s are more suitable, and write a memo to the other members of your team explaining and justifying your results.

The code for the `Avatar` class is given in the following listing:

```
class Avatar {
	// the age of this avatar in years
	private double age;
	// the height of this avatar in meters
	private double height;

	/**
	 * @param age
	 *            the age of this avatar in years
	 * @param height
	 *            the height of this avatar in meters
	 */
	public Avatar(double age, double height) {
		this.age = age;
		this.height = height;
	}

	/**
	 * @return the age of this avatar in years
	 */
	public double getAge() {
		return age;
	}
```

```java
        /**
         * @return the height of this avatar in meters
         */
        public double getHeight() {
                return height;
        }

        /**
         * This method is intentionally empty at present. The
         * development team will fill it in later with some
         * useful computations that are used to "process" the
         * current avatar in order to simulate crowd
         * behavior, but for our purposes, it is fine to assume
         * that the "process" method does nothing.
         */
        public void process() {

        }

}
```

Note the `process()` method of the `Avatar` class: some other members of the team are working on this method, which will compute some interesting crowd behavior based on the avatar's fields. For your experiments, it is fine to assume that the `process()` method does nothing. However, the team has already determined that the avatars in a crowd will need to be processed in *FIFO order* (i.e. first in, first out). In particular, the data structure you recommend for storing a crowd will need to support the following sequence of operations:

1. Create a new crowd, and add a fixed number of avatars to it in a given order.
2. Remove the avatars one at a time from the crowd, in the same order in which they were added, calling the `process()` method on each avatar after it is removed from the crowd.

In other words, the Java code for processing a crowd will closely resemble the following method:

```java
// create a Vector with the given number of elements,
// and then remove all elements from the Vector one at a
// time, removing from the *head* of the Vector, and
// "processing" each element in turn.
public static void doVectorExperiment(int numElements) {
        // STEP 1. create the list and add elements to it
        Vector<Avatar> vectorCrowd = new Vector<Avatar>();
        for (int i = 0; i < numElements; i++) {
                // the values 22.5 and 1.6 are chosen
                // arbitrarily and have no particular
                // significance for this experiment.
                vectorCrowd.add(new Avatar(22.5, 1.6));
        }

        // STEP 2. remove elements from the head of the
        // list, one at a time, processing each one in
        // sequence
        for (int i = 0; i < numElements; i++) {
                Avatar avatar = vectorCrowd.remove(0);
                avatar.process();
        }
}
```

Note that the above method uses the `Vector` class; your experiments will need to employ an additional, very similar, method that uses the `LinkedList` class.

Your memo should be 3–5 pages in length. Any Java code you use for experiments should be included in the memo as an appendix, but this appendix does not count as part of the suggested length of 3–5 pages. The memo should start with a very clear and concise summary of the task you addressed, and your conclusions. (You need to describe the task, because not all members of the team will be aware of the job you were given. You need to summarize your conclusions at the start, because not all of your colleagues will have the time or inclination to read your entire memo.) Following this summary, you should give a complete description of your experiments and the reasoning that led to the stated conclusion. You must give sufficient details that anyone reading the memo could replicate the experiments. Any data that you use should be presented in a suitable form, such as a table or a graph (see the additional hints below). At the end of the memo, briefly summarize your conclusions again.

Any reasonable spacing and formatting conventions are acceptable.

To turn in the assignment, submit it to Moodle using any widely-accepted file type, such as PDF, OpenOffice, or Microsoft Word.

Additional hints:

- The following links may be useful: Java API, LinkedList API, Vector API, System.nanoTime() (useful for timing experiments), Tutorial on Java generics
- Any timing result should be averaged over enough repetitions to reduce uncertainty to a sensible level.
- Use a graph or graphs to present your results, and ensure that everything on the graph is labeled clearly. If appropriate, use a log scale on one or both axes of your graph(s), and explain the meaning of your results.
- Your document should include not only empirical results (i.e. the actual results of timing experiments), but also a theoretical analysis that explains the results. Big-O notation should be used if possible. (Easy challenge to check understanding: describe the performance of the two methods in BigODemo.java using Big-O notation.) A careful reading of the API links above for LinkedList and Vector will help with the theoretical analysis.
- You will almost certainly benefit from visiting the Dickinson Writing Center with a draft of your document. Make an appointment with the Writing Center by calling extension 1620. If possible, make your appointment with a writing tutor who has a background in the sciences, and is therefore familiar with the type of technical writing required for this assignment. The Writing Center has over a dozen tutors with scientific backgrounds, and will be happy to honor your request for one of these tutors if the schedule permits.