1. **(30 points) Data Structures**

A medical center has given each patient a pedometer that reports how many steps the patient takes on every Date (month/day/year). The center plans to use the data to check (1) that patients are taking a similar number of steps each week, and (2) that the average steps per week is above 50,000.

Below, we propose three data structures for the steps data for a *single* patient. Comment on whether each is a good choice for the steps-per-week checks described above (ignore other possible uses of the data). Briefly explain why (a sentence or two is enough), indicating strengths and/or weaknesses as appropriate. There may be multiple good choices or no good choices; evaluate each independently.

Your answer should demonstrate that you understand the traits of the proposed data structure.

The first two proposals use the following class to store the number of steps for a given date:

```
class DailySteps {
  Date whichDay;
  int numSteps;
}
```

- `LinkedList<DailySteps>`, assuming the list is sorted by `Date` (from most recent to least recent).

~~Good~~ Okay choice. If the list is sorted, we can easily find the objects for a single week. Weakness is that we have to break into weeks each time we use the data

- `LinkedList<LinkedList<DailySteps>>`, where each inner list has `DailySteps` objects from the same week, but there is no specific order in either the inner or outer lists.

Very good choice. This problem only cares about computations on the same week. We can easily isolate one week of data through the nested list

- `HashMap<Date,Integer>` that maps each date to the number of steps taken that day.

Not a good choice. It's hard to isolate date for a single week since the keys are independent.

(exam continues next page)

2. **(35 points) Exceptions/Java Programming**

One day, someone at the medical center accidentally updates patients' LDL history with results from a different test whose numeric results are much smaller than valid LDL values. To guard against this happening again, the center wants to check that test results are in an expected range before updating LDL data in EHR objects.

The current (unsafe) updateLDLHistory method (for updating EHR objects with LDL test results, as described on the previous page) is as follows:

```
class MedicalCenter {

  HashMap<String, EHR> patientData; // map patient names to EHRs

  void updateLDLHistory(LinkedList<TestResult> newData) {
try {  checkrange(newData, 100, 250)
    for (TestResult tr : newData) {

      EHR patientEHR = patientData.get(tr.patientName);

      if (!(patientEHR == null)) {

        patientEHR.LDLHistory.add(tr.result);

      }

    } //end of for
  } catch (NotLDLException e) { . . . }
}
                                                   throws
                                                    NotLDLException
```

Here is a method that checks whether every TestResult in a list has a result within a given range.

```
// check whether all numeric results are between the low and high values
boolean checkRange(LinkedList<TestResult> data, double low, double high) {
  for (TestResult tr : data) {
    if ((low > tr.result) || (tr.result > high)) // value out of range
      ~~return false~~;
  }        throw new NotLDLException();
  return true;
}
```

(a) (30 points) Edit either or both of these methods as needed to store the test results only if **all of them** lie within the range low=100 to high=250. If even one TestResult is out of range, none should be stored and a NotLDLException should be thrown. Include **throws** declarations as needed. Assume both methods are in the same class.

(b) (5 points) Somewhere on this page, write the class definition for NotLDLException. Assume it has no fields. You do **NOT** need to write the constructor.

Class NotLDLException extends Exception { }

(exam continues next page)

3. **(40 points) Encapsulation/Program Design**

In this question, as the `updateLDLHistory` method processes `TestResults`, it also creates a "watchlist" of patients whose results have changed significantly since their last test. (The inner **`if`** statement and the `watchlist` variable are the differences from the version in question 2.)

```
class MedicalCenter {
  HashMap<String, EHR> patientData; // map patient names to EHRs

  LinkedList<String> updateLDLHistory(LinkedList<TestResult> newData) {
    // names of patients we should check on based on results
    LinkedList<String> watchlist = new LinkedList<String>();

    for (TestResult tr : newData) {
      EHR patientEHR = patientData.get(tr.patientName);
      if (!(patientEHR == null)) {
        // if difference to last LDL reading above 50, put patient in watchlist
(1)     if ((patientEHR.LDLHistory.size() > 0) &&
            ((tr.result - patientEHR.LDLHistory.get(0)) > 50)) {
          watchlist.add(tr.patientName);
        }
(2)     patientEHR.LDLHistory.add(tr.result);
      }
    }
    return watchlist;
  }
}
```

(a) (10 points) Draw boxes on the above code around computations that should be in a class other than `MedicalCenter` (which contains the `updateLDLHistory` method). Label each box with a unique number (1, 2, ...). You only have to draw and number boxes for this part.

(b) (30 points) Using the space on this page and the next (if needed), for each box:

- Write the method call (method name and arguments) that should replace the boxed-off code. This call can be to a new method (give it a descriptive name). **Write only the call, do not write the actual method.** Use the box numbers to label calls that are not immediately next to their boxes.
- Indicate which class each new method should be in. If the class exists, simply name it (i.e., "in class Dillo"). If the class does not exist, define it (giving the class name, fields, and field types).

(1) in EHR class
boolean largeReadingChange (TestResult tr) {...}

(2) in EHR class
void updateHistory (TestResult tr) {...}

(exam continues next page)