Project: Analyzing Stock Data
Due date: Sunday, April 28th

Spring 2024

1 Introduction

In this project, you will explore the intersection of finance and data structures, applying your programming skills to analyze stock market data. This project is designed to give you practical experience with data structures by manipulating, analyzing, and deriving insights from real-world financial data.

1.1 The Dataset

The dataset you will be working with contains historical stock data for various companies. This data includes the open, high, low, close prices, and volume of trades for each trading day:

- **Date**: The specific trading day for which the data is recorded.
- **Open**: The price at which the first trade was made at the beginning of the trading day.
- **High**: The highest price that the stock reached during the trading day.
- **Low**: The lowest price that the stock reached during the trading day.
- **Close**: The price at which the last trade was made at the end of the trading day.
- **Volume**: The number of shares of the stock that were traded during the trading day.

Understanding these terms is crucial for analyzing the stock data effectively. Throughout this project, you will learn to compute various metrics such as the moving average and stock performance, which are essential for making informed decisions in the stock market. An example of this data is shown in Table 1, and its format is shown in Listing 1. The stock history of each company is stored in a separate Comma Separated Values (CSV) file. The name of the file corresponds to the company code.

Table 1: Sample stock data for Google (company code: GOOGL)

Date	Open	High	Low	Close	Volume
2004-08-19	2.50	2.60	2.40	2.51	893181924
2004-08-20	2.53	2.73	2.52	2.71	456686856
2004-08-23	2.77	2.84	2.73	2.74	365122512
2004-08-24	2.78	2.79	2.59	2.62	304946748
...	...	...	...	...	...

Listing 1: Sample of Google stock data stored in a CSV file named ”GOOGL.csv”

```
Date,Open,High,Low,Close,Volume
2004-08-19,2.502502918243408,2.6041040420532227,2.4014010429382324,2.5110108852386475,893181924
2004-08-20,2.527777910232544,2.7297298908233643,2.515014886856079,2.7104599475860596,456686856
2004-08-23,2.771522045135498,2.8398399353027344,2.7289791107177734,2.7377378940582275,365122512
2004-08-24,2.7837839126586914,2.792793035507202,2.59184193611145,2.6243739128112793,304946748
```

You are given two sets of data. In the folder named `real`, you will find real data files. In the folder `examples`, you will find simple example data to debug your code.

1.2 Terminology

Below are definitions of terms you will frequently encounter in this project:

- **Data Point**: A single piece of data recorded at a specific time. In our dataset, a data point could be the open, high, low, and close prices and volume for a stock on a particular day or a combination of these.
- **Time Series**: A sequence of data points representing the same information over time. In the context of stock data, a time series could be the daily closing prices of a stock over a period.
- **Moving Average**: A calculation used to analyze time series by creating a series of averages of subsets of the full data set. It helps smooth out price data over a specified period, making it easier to identify trends.

  Example 1. *Consider a simple scenario where we have the daily closing prices of a stock over five days as follows:*

  - *Day 1: $10*
  - *Day 2: $12*
  - *Day 3: $11*
  - *Day 4: $13*
  - *Day 5: $14*

  To calculate a 3-day moving average (period = 3), we average the prices over three consecutive days, then move the window one day forward and repeat the process. The calculations are as follows:

  1. *First 3-day moving average (Days 1-3):* $\frac{\$10 + \$12 + \$11}{3} = \$11$
  2. *Second 3-day moving average (Days 2-4):* $\frac{\$12 + \$11 + \$13}{3} = \$12$
  3. *Third 3-day moving average (Days 3-5):* $\frac{\$11 + \$13 + \$14}{3} = \$12.67$

  The resulting time series will then be:

  - *Day 3: $11*
  - *Day 4: $12*
  - *Day 5: $12.67*

- **Price Increase**: Refers to a rise in the price of a stock from one time period to the next. A price increase between two different dates is defined as the difference between the closing prices on those two days. A *Single Day Price Increase* (SDPI) is defined as the difference between the closing and opening price on that day.
- **Stock Performance**: An evaluation of how the price of a particular stock has changed over time. Performance can be measured in various ways, but in our case, we will focus on the relative change in price.

  Example 2. *Given the following closing prices:*

  - *Day 1: $10*
  - *Day 2: $12*
  - *Day 3: $11*
  - *Day 4: $13*
  - *Day 5: $14*

  The performance of this stock between Day 2 and Day 4, for example, is $\frac{13 - 12}{12} = 0.083$*, or 8.3%, where 12 in the denominator is the price of Day 2.*

2 Interfaces and Classes

This section covers the various interfaces and classes used in this project. Interfaces should not be changed. Some classes are given to you, whereas you should implement other classes.

2.1 Interface DLL

This is the interface of a doubly linked list seen in class, augmented with the method `size`. **Implement** this interface in the class `DLLImp`.

```
// Interface representing a doubly linked list.
public interface DLL<T> {

  // Returns the number of elements in the list.
  int size();

  // Returns true if the list is empty, false otherwise.
  boolean empty();

  // Returns true if the current position is at the last element, false otherwise.
  boolean last();

  // Returns true if the current position is at the first element, false
  // otherwise.
  boolean first();

  // Sets the current position to the first element of the list (detailed
  // specification seen in class).
  void findFirst();

  // Advances the current position to the next element in the list (detailed
  // specification seen in class).
  void findNext();

  // Moves the current position to the previous element in the list (detailed
  // specification seen in class).
  void findPrevious();

  // Retrieves the element at the current position.
  T retrieve();

  // Updates the element at the current position with the provided value.
  void update(T val);

  // Inserts the provided element at the current position in the list (detailed
  // specification seen in class).
  void insert(T val);

  // Removes the element at the current position from the list (detailed
  // specification seen in class).
  void remove();
}
```

2.2 Interface DLLComp

This is the interface of a doubly linked list with comparable data. **Implement** this interface in the class `DLLCompImp` [**Requires Chapter on Sorting**].

```
// A DLL where data is comparable.
public interface DLLComp<T extends Comparable<T>> extends DLL<T> {
  // [Requires Chapter on Sorting] Sorts the list. The parameter "increasing"
  // indicates whether the sort is done in increasing or decreasing order.
  void sort(boolean increasing);

  // Returns the maximum element. The list must not be empty.
  T getMax();

  // Returns the maximum element. The list must not be empty.
  T getMin();
}
```

2.3 Class Pair

A generic class used to store a pair of objects (**do not modify**).

```
// Class used to store a pair of elements.
public class Pair<U, V> {
  public U first;
  public V second;

  public Pair(U first, V second) {
    this.first = first;
    this.second = second;
  }

  @Override
  public String toString() {
    return first.toString() + "," + second.toString();
  }
}
```

2.4 Class CompPair

A generic class used to store a pair of objects. The pair is comparable on the second element (**do not modify**).

```
// Class used to store a pair of elements that is comparable on the second element.
public class CompPair<U, V extends Comparable<V>> extends Pair<U, V> implements Comparable<
    CompPair<U, V>> {
  public CompPair(U first, V second) {
    super(first, second);
  }

  @Override
  public int compareTo(CompPair<U, V> other) {
    return this.second.compareTo(other.second);
  }
}
```

2.5 Class DataPoint

A generic class that represents a data point (**do not modify**).

```
import java.text.SimpleDateFormat;
import java.util.Date;

// Represents a single data point in a time series. Each data point consists of a date and
    a corresponding value.
public class DataPoint<T> {

  // The date of the data point.
  public Date date;

  // The value of the data point.
  public T value;

  public DataPoint(Date date, T value) {
    this.date = date;
    this.value = value;
  }

  @Override
  public String toString() {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    return dateFormat.format(date) + "␣:␣" + value.toString();
  }
}
```

2.6 Interface TimeSeries

A generic interface that represents a time series. **Implement** this interface in the class `TimeSeriesImp`.

```
import java.util.Date;

// Interface representing a time series of data points.
public interface TimeSeries<T> {

  // Returns the number of elements in the time series.
  int size();

  // Returns true if the time series is empty, false otherwise.
  boolean empty();

  // Retrieves the data corresponding to a specific date. This method returns the
  // data point for the specified date, or null if no such data point exists.
  T getDataPoint(Date date);

  // Return all dates in increasing order.
  DLL<Date> getAllDates();

  // Returns min date. Time series must not be empty.
  Date getMinDate();

  // Returns max date. Time series must not be empty.
  Date getMaxDate();

  // Adds a new data point to the time series. If successful, the method returns
  // true. If date already exists, the method returns false.
  boolean addDataPoint(DataPoint<T> dataPoint);

  // Updates a data point. This method returns true if the date exists and the
  // update was successful, false otherwise.
  boolean updateDataPoint(DataPoint<T> dataPoint);

  // Removes a data point with given date from the time series. This method
  // returns true if the data point was successfully removed, false otherwise.
  boolean removeDataPoint(Date date);

  // Retrieves all data points in the time series as a DLL that is sorted in
  // increasing order of date.
  DLL<DataPoint<T>> getAllDataPoints();

  // Gets data points from startDate to endDate inclusive. If startDate is null,
  // fetches from the earliest date. If endDate is null, fetches to the latest
  // date. Returns sorted list in increasing date order.
  DLL<DataPoint<T>> getDataPointsInRange(Date startDate, Date endDate);
}
```

2.7 Interface NumericTimeSeries

An interface that specializes and extends `TimeSeries` for `Double` data. **Implement** this interface in the class `NumericTimeSeriesImp`.

```
// Time series of numeric (double) values.
public interface NumericTimeSeries extends TimeSeries<Double> {
  // Calculates and returns the moving average of the data points over a specified
  // period. The value period must be strictly positive. If the time series is
  // empty, the output must be empty.
  NumericTimeSeries calculateMovingAverage(int period);

  // Returns the maximum value in the time series. Time series must not be empty.
  DataPoint<Double> getMax();

  // Returns the minimum value in the time series. Time series must not be empty.
  DataPoint<Double> getMin();
}
```

2.8 Class StockData

A class used to store daily stock data (**do not modify**).

```
// Represent stock date at a given date.
public class StockData {
  public double open; // The opening price of the stock for that day
  public double close; // The closing price of the stock for that day
  public double high; // The highest price the stock reached during that day
  public double low; // The lowest price the stock reached during that day
  public long volume; // The number of shares that were traded during that day

  public StockData(double open, double close, double high, double low, long volume) {
    this.open = open;
    this.close = close;
    this.high = high;
    this.low = low;
    this.volume = volume;
  }
}
```

2.9 Interface StockHistory

An interface that represents the stock history of a single company. **Implement** this interface in the class `StockHistoryImp` (use `TimeSeries` to store data).

```
import java.util.Date;

// Interface representing the stock history of a given company.
public interface StockHistory {

  // Returns the number of elements in the history.
  int size();

  // Returns true if the history is empty, false otherwise.
  boolean empty();

  // Clears all data from the storage.
  void clear();

  // Returns company code.
  String getCompanyCode();

  // Sets company code
  void SetCompanyCode(String companyCode);

  // Returns stock history as a time series.
  TimeSeries<StockData> getTimeSeries();

  // Retrieves StockData for a specific date, or null if no data is found.
  StockData getStockData(Date date);

  // Adds a new StockData and returns true if the operation is successful, false
  // otherwise.
  boolean addStockData(Date date, StockData stockData);

  // Remove the StockData of a given date, and returns true if the operation is
  // successful, false otherwise.
  boolean removeStockData(Date date);
}
```

2.10 Class Map

A map interface that is generic in both key and data. **Implement** this interface in the class `BST` [**Requires Chapter on BST**].

```
public interface Map<K extends Comparable<K>, T> {

  // Returns the number of elements in the map.
  int size();

  // Return true if the tree is empty. Must be O(1).
  boolean empty();

  // Removes all elements in the map.
  void clear();

  // Return the key and data of the current element
  T retrieve();

  // Update the data of current element.
  void update(T e);

  // Search for element with key k and make it the current element if it exists.
  // If the element does not exist the current is unchanged and false is returned.
  // This method must be O(log(n)) in average.
  boolean find(K key);

  // Return the number of key comparisons needed to find key.
  int nbKeyComp(K key);
  // Insert a new element if does not exist and return true. The current points to
  // the new element. If the element already exists, current does not change and
  // false is returned. This method must be O(log(n)) in average.
  boolean insert(K key, T data);

  // Remove the element with key k if it exists and return true. If the element
  // does not exist false is returned (the position of current is unspecified
  // after calling this method). This method must be O(log(n)) in average.
  boolean remove(K key);

  // Return all keys in the map as a list sorted in increasing order.
  DLLComp<K> getKeys();
}
```

2.11 Interface StockHistoryDataSet

An interface that represents the stock histories of several companies. **Implement** this interface in the class `StockHistoryDataSetImp` (use `Map` and `StockHistory` to store data) [**Requires Chapter on BST**].

```
// Interface representing stock history of several companies.
public interface StockHistoryDataSet {
  // Returns the number of companies for which data is stored.
  int size();

  // Returns true if there are no records, false otherwise.
  boolean empty();

  // Clears all data from the storage.
  void clear();

  // Returns the map of stock histories, where the key is the company code.
  Map<String, StockHistory> getStockHistoryMap();

  // Returns the list of all company codes stored in the dataset sorted in
  // increasing order.
  DLLComp<String> getAllCompanyCodes();

  // Retrieves the stock history for a specific company code. This method returns
  // null if no data is found.
  StockHistory getStockHistory(String companyCode);

  // Adds the stock history of a specific company. This method returns true if the
  // operation is successful, false otherwise (company code already exists).
  boolean addStockHistory(StockHistory StockHistory);

  // Removes the stock history of a specific company from the storage. This method
  // returns true if the operation is successful and false if the company code
  // does not exist.
  boolean removeStockHistory(String companyCode);
}
```

2.12 Interface StockDataLoader

An interface used to load stock data from a file. **Implement** this interface in the class `StockDataLoaderImp`.

```
public interface StockDataLoader {

  // Loads and adds stock history from the specified CSV file. The code of the
  // company is the basename of the file. This method returns null if the
  // operation is not successful. Errors include, non-existing file, incorrect
  // format, repeated dates, dates not sorted in increasing order, etc.
  StockHistory loadStockDataFile(String fileName);

  // Loads and returns stock history data from all CSV files in the specified
  // directory. This method returns null if the operation is not successful (see
  // possible errors in the method loadStockDataFile).
  StockHistoryDataSet loadStockDataDir(String directoryPath);
}
```

2.13 Interface StockDataSetAnalyzer

An interface used to analyze stock data. **Implement** this interface in the class `StockDataSetAnalyzerImp`.

```java
import java.util.Date;

//Interface for analyzing stock dataset.
public interface StockDataSetAnalyzer {

  // Returns dataset.
  StockHistoryDataSet getStockHistoryDataSet();

  // Sets dataset.
  void setStockHistoryDataSet(StockHistoryDataSet stockHistoryDataSet);

  // Returns the list of company codes sorted according to their stock performance
  // between startDate and endDate. It returns an empty list if either dates is
  // null.
  DLLComp<CompPair<String, Double>> getSortedByPerformance(Date startDate, Date endDate);

  // Returns the list of company codes sorted according to their total volume
  // between startDate and endDate inclusive. If startDate is null, fetches from
  // the earliest date. If endDate is null, fetches to the latest
  // date.
  DLLComp<CompPair<String, Long>> getSortedByVolume(Date startDate, Date endDate);

  // Returns the list of company codes sorted by the maximum single day price
  // increase between startDate and endDate inclusive. If startDate is null,
  // fetches from the earliest date. If endDate is null, fetches to the latest
  // date.
  DLLComp<CompPair<Pair<String, Date>, Double>> getSortedByMSDPI(Date startDate, Date
      endDate);
}
```

2.14 Class StockAnalyzerCLI [The use of this class is optional]

This class includes a `main` method that runs some examples (**not all code and cases are covered**). You may find the output of this program in the file: `resources/Output.txt`.

2.15 Class StockAnalyzerGUI [The use of this class is optional]

This is a GUI interface that uses other classes (**not all code and cases are covered**). You can find this class in the directory `extra`.

A screenshot of this GUI is shown in Figure 1. To use it, follow the following steps:

1. Click "Load Stock" and select the directory that contains the data.
2. You can change the plotted stock using the list.
3. You can also change the start and end dates.
4. You can sort the stocks alphabetically, or according to performance, volume, or Maximum Single Day Price Increase (MSDPI).
5. To plot the moving average of the current stock, change "Moving Average Period". Set this value to 0 to remove the moving average plot.

StockAnalyzerGUI uses the JFreeChart library for plotting. To compile it, you need to add the JAR files jcommon-1.0.23.jar and jfreechart-1.0.19.jar (located in the directory `extra`) to your Java build path in Eclipse:

1. Open Eclipse IDE and navigate to your project in the Project Explorer or Package Explorer view.
2. Right-click on your project's name to open the context menu.
3. Select Properties at the bottom of the context menu. This opens the Properties dialog for your project.
4. In the Properties dialog, select Java Build Path from the list on the left side.
5. Switch to the Libraries tab in the middle pane.
6. Click the Add External JARs... button on the right side of the dialog.

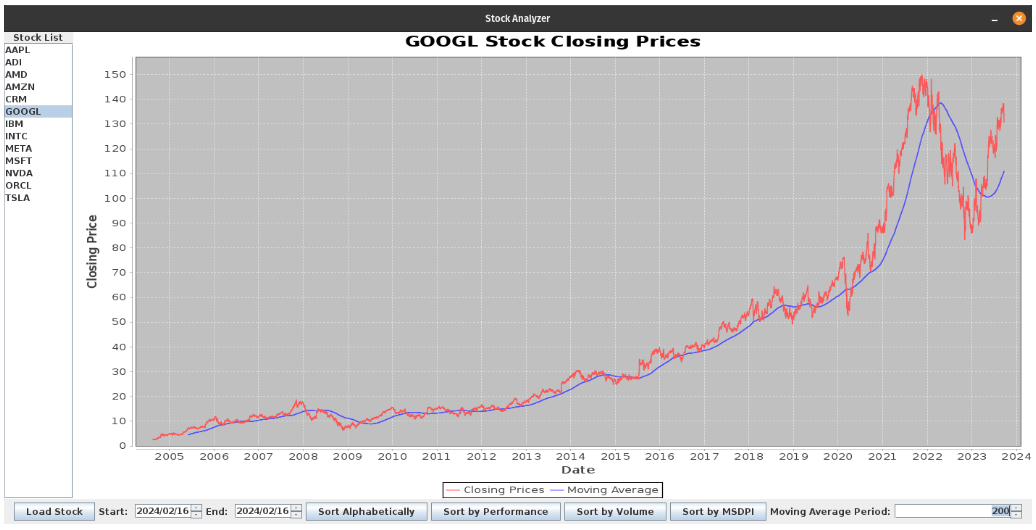

Figure 1: Screenshot of StockAnalyzerGUI.

7. Navigate to the location where you have saved jcommon-1.0.23.jar and jfreechart-1.0.19.jar on your computer.
8. Click Open to add them to your project's build path.
9. You will see the JAR files listed under the Libraries tab. Click Apply and Close to save the changes.

3 Deliverable and Rules

You must deliver:

1. You have to submit the following classes (and any other classes you need) in a zipped file:
   - `DLLImp.java`
   - `DLLCompImp.java`
   - `TimeSeriesImp.java`
   - `NumericTimeSeriesImp.java`
   - `StockHistoryImp.java`
   - `BST.java`
   - `StockHistoryDataSetImp.java`
   - `StockDataLoaderImp.java`
   - `StockDataSetAnalyzerImp.java`

   Notice that you should **not upload** the interfaces and classes given to you.

You have to read and follow the following rules:

1. The specification given in the assignment (**class and interface names, and method signatures**) must not be modified. Any change to the specification results in compilation errors and consequently the mark zero.
2. All data structures used in this assignment **must be implemented** by the student. The use of Java collections or any other data structures library is strictly forbidden.
3. Posting the code of the assignment or a link to it on public servers, social platforms or any communication media including but not limited to Facebook, Twitter or WhatsApp will result in disciplinary measures against any involved parties.
4. The submitted software will be evaluated automatically.
5. All submitted code will be automatically checked for similarity, and if plagiarism is confirmed penalties will apply.
6. You may be selected for discussing your code with an examiner at the discretion of the teaching team. If the examiner concludes plagiarism has taken place, penalties will apply.