CSCE 155 - Java

Lab 13 - Searching & Sorting

Dr. Chris Bourke

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Review notes on Search & Sorting

Some additional resources that may help with this lab:

- Wikipedia Article on Selection Sort:
  http://en.wikipedia.org/wiki/Selection_sort
- A Java implementation of Selection Sort:
  http://en.literateprograms.org/Selection_sort_%28Java%29
- API Specification for `Arrays`: http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Understand basic searching and sorting algorithms
- Understand how comparators work and their purpose
- How to leverage standard search and sort algorithms built into a framework

2 Background

Recursion has been utilized in several searching and sorting algorithms. In particular, quicksort and binary search both can be implemented using recursive functions. The quicksort algorithm works by choosing a pivot element and dividing a list into two sub-lists consisting of elements smaller than the pivot and elements that are larger than the pivot (ties can be broken either way). A recursive quicksort function can then be recursively called on each sub-list until a base case is reached: when the list to be sorted is of size 1 or 0 which, by definition, is already sorted.

Binary search can also be implemented in a recursive manner. Given a sorted array and a key to be searched for in the array, we can examine the middle element in the array. If the key is less than the middle element, we can recursively call the binary search function on the sub-list of all elements to the left of the middle element. If the key is greater than the middle element, we recursively call the binary search function on the sub-list of all elements to the right of the middle element.

Searching and sorting are two solved problems. That is, most languages and frameworks offer built-in functionality or standard implementations in their standard libraries to facilitate searching and sorting through collections. These implementations have been optimized, debugged and tested beyond anything that a single person could ever hope to accomplish. Thus, there is rarely ever a good justification for implementing your own searching and sorting methods. Instead, it is far more important to understand how to leverage the framework and utilize the functionality that it already provides.

3 Activities

The activities in this lab will involve the same baseball data as used in a prior lab. A complete framework has been provided to you to load data on the 2011 National League teams. There is more data this time around and we have defined a structure to encapsulate team data as well as several functions to process the input file and construct instances of the `Team` structures for you.

Clone the project code for this lab from GitHub using the following URL: https://github.com/cbourke/CSCE155-Java-Lab13.

3.1 Sorting the Wrong Way

In this activity, you are to implement a selection sort algorithm to sort an array of `Team` objects. Refer to lecture notes, your text, or any online source on how to implement the selection sort algorithm. The order by which you will sort the array will be according to the total team payroll (in dollars) in increasing order.

Instructions

1. Familiarize yourself with the `Team` class and the methods provided to you (the `main` method automatically loads the data from the data file and provides an array of teams).
2. Implement the `selectionSortTeamsByPayroll` function in the `MLBTeamUtils.java` file as specified
3. Run your program

3.2 Slightly Better Sorting

The `Team` class has many different data fields; wins, losses (and win percentage), team name, city, state, payroll, average salary. Say that we wanted to re-sort teams according to one of these fields in either ascending or descending order (or even some combination of fields–state/city for example). Doing it the wrong way as above we would need to implement no less than 16 different sort functions!

Most of the code would be the same though; only the criteria on which we update the index of the minimum element would differ. Instead, it would be better to implement one sorting function and make it configurable: provide a means by which the function can determine the relative ordering of any two elements. The way of doing this in Java is to define a *comparator* class.

`Comparator` classes are simple. They are required to implement the interface `Comparator<T>`, which means specifying (coding) one method: `compare(T a, T b)`. The `T` in this declaration is a generic that we specify when we create a comparator. The method itself takes two arguments: a and b (of type `T`) and returns:

- A negative value if $a < b$
- Zero if a is equal to b
- A positive value if $a > b$

For example, in order to sort an array containing references to instances of `Team` by name, we would create a class that implements `Comparator<Team>` and implement the `compare()` method to compare two references to `Team`s passed in, and return an integer value based on the above rules.

Several completed comparator classes have been provided as examples. All the comparator classes take two `Team` arguments, and return an integer based on the specified comparison. Refer to the provided comparator classes for concrete examples.

Instructions

1. Implement the `PayrollDescendingComparator` class. Only one method, `compare()` needs to be coded.
2. Use the completed comparator classes provided with this lab an example on how to implement the `compare()` method
3. Implement the `selectionSortTeams()` method in `MLBTeamsUtils.java` source file
4. Use the comparator class you just created to call `selectionSortTeams()` to sort an array of `Team`s by payroll descending

3.3 Sorting the Right Way

The best way to sort is to leverage a sorting algorithm provided by a standard library. This eliminates the need to "reinvent" the wheel and avoids debugging, testing, etc. of our own code. Java provides several sorting methods in its standard developer kit (SDK). The one we will make use of is in the Arrays class. The method is able to sort arrays containing any type of object by making use of a provided comparator:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

where

- `T[]` a is the array of objects (of type `T`) to be sorted

- `Comparator<? super T> c` is the comparator class that is used to determine sort order of the object elements in the array. Note: the `? super T` in the comparator angle brackets indicate that the type of class being compared must be `T`, or any superclass (ancestor) of type `T`.

Instructions

1. Examine the source files and observe how `Comparator` classes are defined and how the `Arrays.sort` method is called
2. Implement two more comparators, `StateComparator` which sorts based on team home state, and `StateCityComparator` which sorts based first on state then by city
3. Use your method in the `main` method to re-sort the array and print out the results.

3.4 Searching

The `Arrays` class also offers a method to search an array. In this exercise, we will focus on two strategies for searching–linear search and binary search.

- `linearSearchMLB` - a linear search implementation (in the `MLBTeamUtils` class) that does not require the array to be sorted. The method works by iterating through the array and calling your comparator class on the provided key (an object instance whose state matches the criteria that you are searching for). It returns the index of the first instance such that the comparator returns zero for (indicating equality between the objects).
- `binarySearch` - a binary search implementation (in the `Arrays` class) that requires the array to be sorted according to the same comparator used to search.

Both methods require a comparator (as used in sorting) and a key upon which to search. A key is a "dummy" instance of the same type as the array that contains values of fields that you?re searching for.

Instructions

1. Examine the `linearSearchMLB` method in the `MLBTeamUtils` class and understand how it works
2. Answer the questions in your worksheet regarding this code segments
3. Based on your observations add code to search the array for the team representing the Chicago Cubs:

a) Create a dummy `Team` key for the Cubs by instantiating a dummy team and using empty strings and zero values except for the team name (which should be `"Cubs"`)

b) Sort the array by team name using the appropriate comparator

c) Call the `binarySearch` method using your key and the appropriate comparator to find the `Team` representing the Chicago Cubs

d) Print the team to the standard output

e) Demonstrate your working program to a lab instructor

4 Advanced Activities (Optional)

Selection sort is a quadratic sorting algorithm, thus doubling the input size (n elements to $2n$ elements) leads to a blowup in its execution time by a factor of 4. Quick sort requires only $n \log(n)$ operations on average, so doubling the input size would only lead (roughly) to a blowup in execution time by a factor of about 2. Verify this theoretical analysis by setting up an experiment to time each sorting algorithm on various input sizes of randomly generated integer arrays. Use `System.nanoTime()` to record the elapsed time of your algorithms.