CSCE 155 - C

Lab 13.0 - Searching & Sorting

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 12 and 25 of the Computer Science I textbook
3. Watch Videos 13.1 thru 13.7 of the Computer Science I video series

Peer Programming Pair-Up

For students in the online section: you may complete the lab on your own if you wish or you may team up with a partner of your choosing, or, you may consult with a lab instructor to get teamed up online (via Zoom).

For students in the face-to-face section: your lab instructor will team you up with a partner.

To encourage collaboration and a team environment, labs are be structured in a *peer 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 comparator functions 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-C-Lab13`.

3.1 Sorting the Wrong Way

In this activity, you are to implement a selection sort algorithm to sort an array of `Team` structures. 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` structure and the functions provided to you (the `main` program automatically loads the data from the data file and provides an array of teams).
2. Implement the `selectionSortTeamsByPayroll` function in the `mlb.c` file as specified
3. Compile and run your program (use make which produces an executable called `mlbDriver`)

3.2 Slightly Better Sorting

The `Team` structure 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 C is to define a comparator function.

Comparator functions are simple: they take two arguments a, b (specified by generic void pointers) and return:

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

Several examples have been provided for comparing `Team` structures based on several different criteria. The usual pattern is to cast the arguments to the expected types, then to examine the relevant field(s) and return a result that orders the two teams appropriately.

Instructions

1. Implement the `selectionSortTeams` function by using the code in Activity 1 with appropriate modifications (use the provided `compar` function to find the minimum element each time)
2. Look to the comparator functions provided to you and to the bubble sort algorithm example on how you might use a comparator function
3. Implement your own comparator function that orders `Team`s according to the total payroll in decreasing order.
4. Use your comparator in the `mlbDemo.c` program to call your `selectionSortTeams` function.

3.3 Sorting the Right Way

The better way of doing this is to leverage the standard C library's `qsort` sorting function. The signature of this function is as follows:

```c
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

where

- `base` is the array to be sorted
- `nmemb` is the number of elements in the array
- `size` is the size of each element (use `sizeof()`)
- `compar` is a comparator function pointer

Instructions

1. Examine the source files and observe how comparator functions are implemented and how the `qsort` function is called.
2. Implement your own comparator function that orders `Team`s according to win percentage in increasing order.
3. Use your function in the `main` program to re-sort the array and print out the results.

3.4 Searching

The standard C library offers several search functions. The two that we will focus on are as follows.

- `lfind` - a linear search implementation that does not require the array to be sorted. The function works by iterating through the array and calling your comparator function on the provided key. It returns the first instance such that the comparator returns zero (equal).
- `bsearch` - a binary search implementation that requires the array to be sorted according to the same comparator used to search.

Both functions require a comparator function (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 searching code segment in the `mlbDriver.c` and understand how each function is called.
2. Answer the questions in your worksheet regarding this code segment.
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 calling the `createTeam` function using empty strings and zero values except for the team name (which should be `"Cubs"`).
   b) Sort the array by team name by calling `qsort` using the appropriate comparator function.
   c) Call the `bsearch` function using your key and the appropriate comparator function to find the `Team` representing the Chicago Cubs.

   d) Print out the team by using the `printTeam` function.

4. Answer the questions in your worksheet and demonstrate your working program to a lab instructor.

4 Handin/Grader Instructions

1. Hand in your completed files:
   - `mlb.c`
   - `mlb.h`
   - `mlbDriver.c`
   - `worksheet.md`

   through the webhandin (`https://cse-apps.unl.edu/handin`) using your cse login and password.
2. Even if you worked with a partner, you *both* should turn in all files.
3. Verify your program by grading yourself through the webgrader (`https://cse.unl.edu/~cse155e/grade/`) using the same credentials.
4. Recall that both expected output and your program's output will be displayed. The formatting may differ slightly which is fine. As long as your program successfully compiles, runs and outputs the *same values*, it is considered correct.

5 Advanced Activity (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. Make use of the time library?s functions to time the execution of your function calls.