CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2019
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MW 2:30-3:45
Handout 9: Lab #6 — Daniel 313

1 Sort and Scan Practice

This lab involves coding up a "sweep line" or "sort and scan" algorithm, to illustrate how sorting the input can dramatically simplify the remainder of an algorithm. Not to fear, we will also revisit hashing and binary search trees.

Our problem of study is the longest line of sight problem, variants of which have been discussed in lecture and also appear on homework #2. The input to the problem consists of the locations $x_1 \ldots x_n$ (along a one-dimensional number line) and heights $h_1 \ldots h_n$ of n people standing in a row. All of the x_i's will be distinct (no duplicates), and all of the h_i's will also be distinct.

Input to your code will come from a file with n being on the first line, and each of the successive n lines describing a person in terms of two nonnegative integers x and h. An example input file is given in the directory:

`/group/course/cpsc212/f19/lab06/`

2 Goal One: Sorting Using the Standard Library

If you `#include <algorithm>` in your code, then you can sort using the quicksort function that is part of the C++ standard library. The running time is $\Theta(n \log n)$ in most cases, although this depends on selection of pivots, as described in lecture, so for particularly contrived inputs it could be as bad as $\Theta(n^2)$. To sort an array, simply pass pointers to the beginning and end (or rather, one position just after the end) to the sort function, like this:

```
int A[4] = { 10, 6, 2, 5 };
sort(A, A+4);
// Now A contains 2, 5, 6, 10
// Note that A+3 points to A[3], the last integer in A,
// and hence A+4 actually points to one position past the end of A
```

Make sure you have `using namespace std;` near the top of your file, or else you need to write the full name `std::sort` instead of just `sort`.

The built-in sorting function can sort any type of object for which the "less than" operator is defined (e.g., ints, strings, doubles). For sorting more exotic data types, you may need to define your own "less than" operator first. For example:

```
#include <algorithm>
#include <string>
using namespace std;

struct Student {
  string name;
  int GPA;
};

// const just means you can't modify A or B in this function
bool operator< (const Student &A, const Student &B)
{
  // Return whether A's GPA is less than B's GPA, unless the student
  // is Marianna which case always treat as the smaller of the two
  // just for fun...
  if (A.name == "Marianna") return true;
  if (B.name == "Marianna") return false;
  return A.GPA < B.GPA;
}

int main(void)
{
  Student *S = new Student[100];
  // Fill in Student array...
  sort (S, S+100);
}
```

Your task here is to read in the input file into an array of structs, each representing a person (so it should have a location and height field). You should then sort this array in reverse order of height, so the first person in the array will have the tallest height.

3 Finding Long Lines of Sight

For each person p in the input, print out the distance to the closest person taller than p on p's left, and the distance to the closest person taller than p on p's right. Your output should therefore have n lines each with 2 integers. If there is nobody taller on the left or right, output -1 for this corresponding distance. Your output should be *in order in which the people originally appeared in the input file*. Therefore, you may need to remember in your Person structure the original index of each person in the input array before sorting, so you can build an output array with the results in the right order.

To compute long lines of sight quickly, we use a "sweep line" or "sort and scan" approach. We process each person in decreasing order of height, adding their locations to a balanced binary search

tree (please feel welcome to import the code we previously wrote for balanced BSTs here). The BST will only store the locations (not heights or any other data), and it will therefore be ordered from left to right by location.

Right before we add each person's location x to the BST, we determine the closest neighbors on the left and right who are taller by looking up the predecessor and successor of x in the tree. To do this, you will need to implement two functions:

```
Node *predfind(Node *root, int x)
{
   // returns a pointer to the node with key x
   // if x does not exist in the tree, returns a pointer to
   // the node with the next-smallest key than x, or NULL
   // if there is no key smaller than x.
}

Node *succfind(Node *root, int x)
{
   // same, only for successor instead of predecessor
}
```

Please implement these functions in a simple recursive manner, and use them to complete solving the longest line of sight problem.

4 Generating Large Inputs

To stress test your code, write a second program that generates a large random input of some size n provided on the command line (see the code provided for homework #2 for an example of how to parse an argument on the command line).

Each location and each height should be chosen to be a random integer in the range $0 \ldots 10n$. However, since we want to guarantee that the locations are distinct and that the heights are distinct, you will may need to re-choose some of these random numbers if you happen to generate a number that has been used before. I.e., to pick each random number, you keep choosing random numbers in the appropriate range until you find a number that hasn't been used previously. You should know the right data structure to use for the task of remembering the random numbers you have already used. Please feel encouraged to import code for this data structure that we have already written during previous assignments / labs.

5 Grading

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Saturday, October 5. No late submissions will be accepted.