Shreyas Bharadwaj

UID: 705 342 716

CS 32 Winter 2020

Homework 4

1. See Map.h.
2. The call to Map<Coord, int>::insert causes the following compilation error: error: invalid operands to binary expression ('Coord' and 'const Coord'). Note that merely creating a Map of Coords to ints does not cause a compilation error, it is the act of trying to insert a pair that does. To understand why this happens, we may look at the implementation of the insert method. The insert method calls the auxiliary helper method doInsertOrUpdate(), which attempts to add a key-value pair to the map. The method doInsertOrUpdate() attempts to check if a key-value pair is in the Map and compares it to the head using the != operator. The != operator is a binary operator that, by default, takes only built-in types, such as int, long, double, bool, and so on. Therefore, trying to use the operator with a Coord, which does not have a definition for it, will seem to the compiler that the code is trying to compare two operands that cannot be compared with a binary operator. To fix this error, the programmer would need to overload the operator in the Coord class to an appropriate definition.
   1. Looking from the inside out of the function, the inner loop beginning with for (int k = 0; k < N; k++) makes N passes through the loop, and will thus run in time; that is, proportional to the number of items in the array. The loop just outside this inner loop, beginning with for (int j = 0; j < N; j++) makes N passes through the loop, and will thus run in time; that is, proportional to the number of items in the array. This means the total runtime of the inner two loops is now The outermost loop, beginning with for (int i = 0; i < N; i++) makes N passes through the loop, and will thus run in time; that is, proportional to the number of items in the array. This means the total runtime of three loops is now and the runtime of the algorithm as a whole is that is, proportional to the cube of the number of items in the array.
   2. Looking from the inside out of the function, the inner loop beginning with for (int k = 0; k < N; k++) makes N passes through the loop, and will thus run in time; that is, proportional to the number of items in the array. The loop just outside this inner loop, beginning with for (int j = 0; j < i; j++) makes i passes through the loop. However, since i grows linearly with N, this is no different than a linear time pass through the array (although on average, it will pass perhaps half as many times through.) Thus this loop also has runtime and so the inner two loops have runtime From here on, there is no change; the outermost loop, beginning with for (int i = 0; i < N; i++) makes N passes through the loop, and will thus run in time; that is, proportional to the number of items in the array. This means the total runtime of three loops is now and the runtime of the algorithm as a whole is that is, proportional to the cube of the number of items in the array. Thus, the change from N to i, while probably halving the runtime on average, will not do anything to change of runtime as N grows. That is to say, they are both equally bad algorithms.
   3. To analyze the time complexity of this method, the time complexities of each of the steps taken must be analyzed. Looking from the inside out, there are two methods called inside the loop, get() and insert(), which are both linear time algorithms; that is, with time complexity Looking outside from there, we see a loop beginning with for (int i = 1; i < m.size(); i++), which makes one pass through the loop and is therefore of time complexity Therefore, until this point, the algorithm has complexity Outside the for loop, we see get() and insert(), both of which are linear time algorithms. Therefore, the time remains . The final if statement is of constant time, so the overall is
   4. Unlike the previous algorithm, this one does not need to loop through the first Map’s nodes and reassign its values to a new temporary Map. Therefore, the inner loop needed in the previous implementation is entirely avoided. It thus needs to make just one pass through the linked list, shifting the value of the i+1th node down to the ith node, and finally setting the value of the last node to the former value of the first. Only one pass through the linked list implies a time complexity of so this algorithm has a faster runtime than the previous one. Thus, it is better than the one in part (a).