Suyogya Rayamajhi

CS313-Spring 2021

Group 6

Problem 1:

Code Explanation:

The part a) and b) of the program firstly performs the basic insertion, deletion, search functions in a vector and a linked list. We then calculate the time taken to fill both the vector and the linked list with 1000 random numbers and 100 strings of size 5 using the **<time.h> and <chrono>** header files and **namespace::chrono**. The time of the clock is read before the task is performed and again read after the task has ended. Then the difference gives us the time taken to perform the task.

In the part c) we use move semantics to fill up the vector & list with 10000 random strings using 3 simple functions **getRandomString, VectorOfRandomStrings_Move and ListOfRandomString_Move**. All the functions are timed during the call using the method performed in part a) and b).

Analysis:

After compiling the programs multiple times, filling a vector and a list with 1000 random output showed that filling a vector with random integers is faster than filling a list with random integers.

Also, when the vector is filled with random 100 strings each of size 5, it is slower than filling the list with the same amount of data. Using move semantics, the results show that filling a vector with 1000 random strings is slower than filling the list with the same amount of data.

Problem 6:

Code Explanation:

The ciphertext created in The Vigenére cipher is extremely hard to decode because it does not work by one-one-substitution. I chose vector as my preferred data structure because of its dynamic functionality and random-access.

We use a **convertChar** and a **convertInt** functions which convert the provided string to char and int values, respectively.

Once we create a vector **vec** containing the integer representation of the string is constructed and a random key is generated. This vector

vec and the key is then sent to the encode function which iterates through the text and the key, adding elements mod 26, that generates the ciphertext and placed in **vec**.

Next, we create a list of keys that continually increases the threshold according to the key score. This uses the two new vectors **listKeys** & **tempVec**.

Once the encryption is finished a **scoreCalc** function uses the randomly generated key to decrypt the message and then analyze the result. The analysis focuses on the frequency of letters in the English language, which occurs over quite a broad range of 13% for ‘e’ to 0.074% for ‘z’. The function returns a double in the form of a key score to the main. The key score relies on the principle that, having two strings and wanting to take a sum of products of pairs of numbers from each string, the largest sum will be derived from the product of largest numbers in each string, plus the product of the second largest in each string, and so on. This sum is then divided by the number of letters in the text to generate a more broadly applicable score. The average random key returns a key score of approximately 3.4. For comparison, the actual

key, when tested, typically returns a value of 6.2 or so. Random keys with a score above 4.0 are relatively scarce.

Then the **scoreEle** function comes into play that returns a score based on the position of the key. A score is obtained for every element of the key and we use a loop to find the max score for every key and stores that letter as the result and moves on further.

As a result, particularly improved key is generated and given to the user.

Problem 7:

Code Explanation:

The same functions used in Program 1 of the project were used to find the time taken to fill up a vector, array and a linked list with 10000 random numbers.

Hypothesis:

I had assumed that Vector would take the least time, List would take the medium time & Array would take the most time.

Output:

Time taken to fill Vector with 10000 random numbers: 108 microseconds

Time taken to fill an Array with 10000 random numbers: 241 microseconds

Time taken to fill a List with 10000 random numbers: 531 microseconds

Explanation:

The array and the vector are respectively statically and dynamically allocated. Hence, both of them perform quite good but vector edges out arrays. Since the list allocates from the node and requires extra space for the element itself and the pointer to the previous and the next node, it takes significantly longer time to perform.

Problem 11:

Code Explanation:

Here we create a base class named **Restaurant** that has class variable, a member function and a pure virtual function. A pure virtual function can have implementation we must be overridden in the derived

class else the derived class is caused to be an abstract class. Since **Restaurant** has a pure virtual function, it is an abstract class.

A struct is also derived from the base class and multiple classes namely **Italian_Restaurant, Greek_Restaurant and Chinese_Restaurant** are also derived from it. We use pointers and references of the abstract class type to access the derived class.

Problem 12:

Code Explanation:

In this program a linked list for each appointment. We define functions that schedule, display, book and cancel the appointment using Dynamic memory allocation and by assigning data and addresses.

We use a number switch-case to access each function as per the user's needs. When the schedule function is called we ask the user for how many slots he wants to book for. The time constraints are printed when the user input is asked. Each input is store in the dynamically allocated memory.

We have created class that contains all the functions and a scope resolution operator (::) is used to define the functions outside of the class. A table is used to refine the user output.