laboratory C

Performance Evaluation

Objectives

In this laboratory you will:

- Implement a Timer class that you can use to measure the length time between two events—when a function starts and when it finishes, for instance
- Compare the performance of a set of searching routines
- Compare the performance of a set of sorting routines
- Compare the performance of your array and linked list implementations of the Stack ADT

Overview

A routine's performance can be judged in many ways and on many levels. In other laboratories, you describe performance using order-of-magnitude estimates of a routine's execution time. You develop these estimates by analyzing how the routine performs its task, paying particular attention to how it uses iteration and recursion. You then express the routine's projected execution time as a function of the number of data items (N) that it manipulates as it performs its task. The results are estimates of the form O(N), O(LogN), and so on.

These order-of-magnitude estimates allow you to group routines based on their projected performance under different conditions (best case, worst case, and so forth). As important as these order-of-magnitude estimates are, they are by their very nature only estimates. They do not take into account factors specific to a particular environment, such as how a routine is implemented, the type of computer system on which it is being run, and the kind of data being processed. If you are to accurately determine how well or poorly a given routine will perform in a particular environment, you need to evaluate the routine in that environment.

In this laboratory, you measure the performance of a variety of routines. You begin by developing a set of tools that allow you to measure execution time. Then you use these tools to measure the execution times of the routines.

You can determine a routine's execution time in a number of ways. The timings performed in this laboratory will be generated using the approach summarized below.

Get the current system time (call this *startTime*).

Execute the routine.

Get the current system time (call this *stopTime*).

The routine's execution time = *startTime* − *stopTime*.

If the routine executes very rapidly, then the difference between `startTime` and `stopTime` may be too small for your computer system to measure. Should this be the case, you need to execute the routine several times and divide the length of the resulting time interval by the number of repetitions, as follows:

Get the current system time (call this *startTime*).

Execute the routine *m* times.

Get the current system time (call this *stopTime*).

The routine's execution time = (*startTime* − *stopTime*) / *m*.

To use this approach, you must have some method for getting and storing the "current system time". How the current system time is defined and how it is accessed varies from system to system. Two common methods are outlined below.

Method 1

Use a function call to get the amount of processor time that your program (or process) has used. Typically, the processor time is measured in clock ticks or fractions of a second. Store this information in a variable of the following type:

```
typedef long SystemTime;
```

You can use this method on most systems. You must use it on multiuser or multiprocess systems, where the routine you are timing is not the only program running.

Method 2

Use a function call to get the current time of day. Store this information in a variable of the following type:

```
struct SystemTime
{
    int hour,       // Hour   0-23
        minute,     // Minute 0-59
        second,     // Second 0-59
        fraction;   // Fraction of a second
};
```

The range of values for the fraction field depends on the resolution of the system clock. Common ranges are 0–99 (hundredths of a second) and 0–999 (thousandths of a second). This method is effective only on single-user/single-process systems where the routine you are timing is the only program running.

In addition to acquiring and storing a point in time, you also need a convenient mechanism for measuring time intervals. The Timer ADT described below uses the familiar stopwatch metaphor to describe the timing process.

Start the timer.

...

Stop the timer.

Read the elapsed time.

Timer ADT

Data Items

A pair of times that denote the beginning and end of a time interval.

Structure

None

Operations

`void start ()`

Requirements
None

Results
Marks the beginning of a time interval (starts the timer).

```
void stop ()
```

Requirements
The beginning of a time interval has been marked.

Results
Marks the end of a time interval (stops the timer).

```
double getElapsedTime ()
```

Requirements
The beginning and end of a time interval have been marked.

Results
Returns the length of the time interval in seconds.

Laboratory C: Cover Sheet

Name _____ Date _____

Section _____

Place a check mark in the *Assigned* column next to the exercises your instructor has assigned to you. Attach this cover sheet to the front of the packet of materials you submit following the laboratory.

Activities	**Assigned:** Check or list exercise numbers	**Completed**
Prelab Exercise		
Bridge Exercise		
In-lab Exercise 1		
In-lab Exercise 2		
In-lab Exercise 3		
Postlab Exercise 1		
Postlab Exercise 2		
Total		

Laboratory C: Prelab Exercise

Name _____ Date _____

Section _____

Step 1: Select one of the two methods for acquiring and representing a point in time and use this method to create an implementation of the Timer ADT. Base your implementation on the following class declaration from the file *timer.hs*.

```
// Insert a declaration for SystemTime here.

class Timer
{
  public:

    // Start and stop the timer
    void start ();
    void stop ();

    // Compute the elapsed time (in seconds)
    double getElapsedTime ();

  private:

    SystemTime startTime,   // Time that the timer was started
               stopTime;    // Time that the timer was stopped
};
```

Step 2: Add the appropriate declaration for `SystemTime` to the beginning of the file and save the resulting header file as *timer.h*. Save your implementation of the Timer ADT in the file *time.cpp*.

Step 3: What is the resolution of your implementation—that is, what is the shortest time interval it can accurately measure?

Laboratory C: Bridge Exercise

Name _____ Date _____

Section _____

Check with your instructor whether you are to complete this exercise prior to your lab period or during lab.

The test program in the program shell file *testc.cs* allows you to test the accuracy of your implementation of the Timer ADT by measuring time intervals of known duration.

```
#include <iostream>
#include <iomanip>
#include <ctime>

#include "timer.h"

using namespace std;

// wait() is cross platform and works well but is not efficient.
// Feel free to replace it with a routine that works better in
// your environment.
void wait(int secs)
{
    int start = clock();
    while (clock() - start < CLOCKS_PER_SEC * secs);
}

void main()
{
    Timer checkTimer;       // Timer
    clock_t timeInterval;   // Time interval to pause

    // Get the time interval.

    // Measure the specified time interval.

    checkTimer.start();          // Start the timer
                                 // Pause for the specified time interval
    checkTimer.stop();           // Stop the timer

    cout << "Measured time interval ( in seconds ) : "
         << checkTimer.getElapsedTime() << endl;
}
```

Step 1: Two data items are left incomplete in this program: the call to the function that pauses the program and the string that prompts the user to enter a time interval. Complete the program by specifying the name of a "pause" function supported by your system. Common names for this function include `sleep()`, `delay()`, and `pause()`. Or you can use the provided `wait()` function. Add the time unit used by this function to the prompt string. Save the resulting program as *testc.cpp*.

Step 2: Prepare a test plan for your implementation of the Timer ADT. Your test plan should cover intervals of various lengths, including intervals at or near the resolution of your implementation. A test plan form follows.

Step 3: Execute your test plan. If you discover mistakes in your implementation, correct them and execute your test plan again.

Test Plan for the Operations in the Timer ADT

Test Case	*Actual Time Period (in seconds)*	*Measured time period (in seconds)*	*Checked*

Laboratory C: In-lab Exercise 1

Name _____ Date _____

Section _____

In this exercise you will examine the performance of the searching routines in the file *search.cpp*.

Step 1: Use the program in the file *timesrch.cpp* to measure the execution times of the `linearSearch()`, `binarySearch()`, and `unknownSearch()` routines. This program begins by generating an ordered list of integer keys (`keyList`) and a set of keys to search for in this list (`searchSet`). It then measures the amount of time it takes to search for the keys using the specified routines and computes the average time per search.

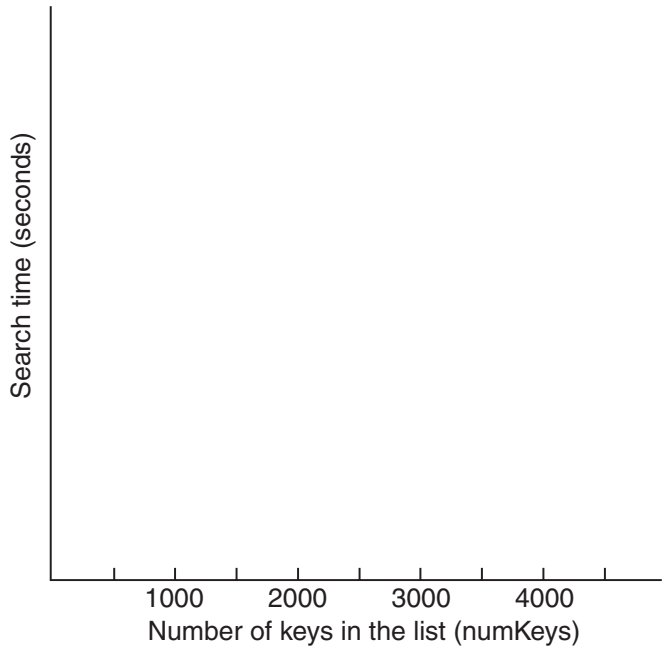
The constant `numRepetitions` controls how many times each search is executed. Depending on the speed of your system, you may need to use a value of `numRepetitions` that differs from the value given in the test program. **Before continuing, check with your instructor regarding what value of `numRepetitions` you should use.**

Step 2: Complete the following table by measuring the execution times of the `linearSearch()`, `binarySearch()`, and `unknownSearch()` routines for each of the values of `numKeys` listed in the table.

Execution Times of a Set of Searching Routines

Routine		*Number of keys in the list (`numKeys`)*		
		1000	2000	4000
`linearSearch()`	O(N)			
`binarySearch()`	O(LogN)			
`unknownSearch()`	O( )			

Step 3: Plot your results below.



Step 4: How well do your measured times conform with the order-of-magnitude estimates given for the `linearSearch()` and `binarySearch()` routines?

Step 5: Using the code in the file *search.cpp* and your measured execution times as a basis, develop an order-of-magnitude estimate of the execution time of the `unknownSearch()` routine. Briefly explain your reasoning behind this estimate.

Laboratory C: In-lab Exercise 2

Name _____ Date _____

Section _____

In this exercise you will examine the performance of the set of sorting routines in the file *sort.cpp*.

Step 1: Use the program in the file *timesort.cpp* to measure the execution times of the `selectionSort()`, `quickSort()`, and `unknownSort()` routines. This program begins by generating a list of integer keys (`keyList`). It then measures the amount of time it takes to sort this list into ascending order using the specified routine.

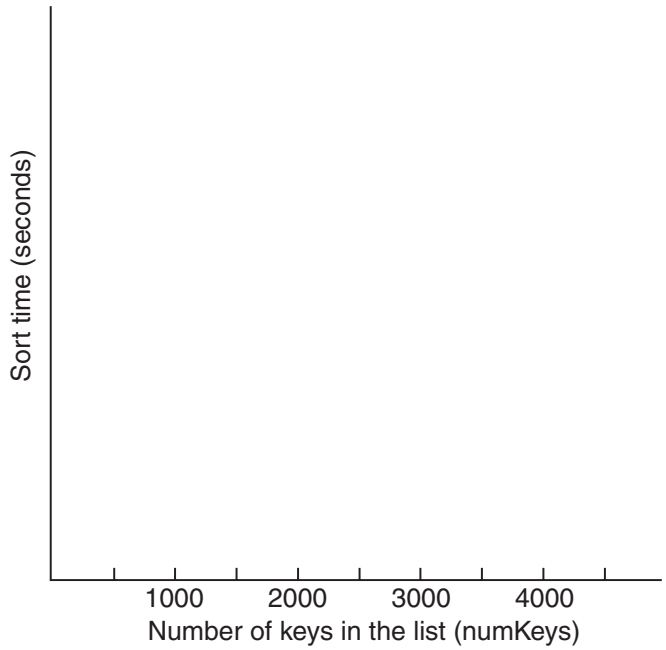
The constant `numRepetitions` controls how many times each search is executed. Depending on the speed of your system, you may need to use a value of `numRepetitions` that differs from the value given in the test program. **Before continuing, check with your instructor regarding what value of `numRepetitions` you should use.**

Step 2: Complete the following table by measuring the execution times of the `selectionSort()`, `quickSort()`, and `unknownSort()` routines for each combination of the three test categories and the three values of `numKeys` listed in the table.

Execution Times of a Set of Sorting Routines

Routine	*Number of keys in the list (`numKeys`)*		
	1000	2000	4000
`selectionSort()` $O(N^2)$			
`quickSort()` O(NLogN)			
`unknownSort()` O( )			

Step 3: Plot your results below.



Step 4: How well do your measured times conform with the order-of-magnitude estimates given for the `selectionSort()` and `quickSort()` routines?

Step 5: Using the code in the file *sort.cpp* and your measured execution times as a basis, develop an order-of-magnitude estimate of the execution time of the `unknownSort()` routine. Briefly explain your reasoning behind this estimate.