Homework #1

Wenyi Liu

1 and 2.

Below is the C++ source code of this homework

#include <iostream>

#include <ctime>

#include <cstdlib>

#include <cstdio>

using namespace std;

// O(length) time overall for this insert function

int \*insert(int \*array, int length, int index, int value) {

int\* newArray = NULL; // O(1) time

// input array is empty, return a new array with length 1 and the target value

if (length == 0) { // O(1) time overall for if

newArray = new int[1]; // O(1) time

newArray[0] = value; // O(1) time

return newArray; // O(1) time

}

newArray = new int[length + 1]; // O(1) time

// copy array[0, index) to newArray[0, index)

for (int i = 0; i < index; ++i) { // O(index) time overall

newArray[i] = array[i]; // O(1) time

}

newArray[index] = value; // O(1) time

// copy array[index, length) to newArray[index + 1, length + 1)

for (int i = index; i < length; ++i) { // O(length - index) time overall

newArray[i + 1] = array[i]; // O(1) time

}

delete [] array; // free the memory of the old array, O(1) time

return newArray; // O(1) time

}

int main() {

const int INSERTS\_PER\_READING = 1000;

// start with an empty array

int\* array = NULL;

int length = 0;

// print the header

printf("%15s %20s\n", "Array length", "Seconds per insert");

// take 60 readings

for (int i = 0; i < 60; ++i) {

clock\_t startTime = clock();

// Each reading will be taken after INSERTS\_PER\_READING inserts

for (int j = 0; j < INSERTS\_PER\_READING; ++j) {

int index = rand() % (length + 1); // random index in [0, length]

int value = rand(); // random integer value

array = insert(array, length, index, value);

length++;

}

clock\_t stopTime = clock();

double timePerInsert = static\_cast<double>(stopTime - startTime)

/ CLOCKS\_PER\_SEC / INSERTS\_PER\_READING;

// Output reading in tabular format

printf("%15d %20.8f\n", (i + 1) \* INSERTS\_PER\_READING, timePerInsert);

}

// free the old array

delete [] array;

return 0;

}

And I compile the code with g++ using the following command:

g++ hw1.cpp -o hw1.exe

The output of the program is shown as follows.

Array length Seconds per insert

1000 0.00000138

2000 0.00000358

3000 0.00000576

4000 0.00000781

5000 0.00000972

6000 0.00001175

7000 0.00001393

8000 0.00001659

9000 0.00001873

10000 0.00002159

11000 0.00002306

12000 0.00002603

13000 0.00002720

14000 0.00002976

15000 0.00003200

16000 0.00003380

17000 0.00003685

18000 0.00003983

19000 0.00004063

20000 0.00004286

21000 0.00004626

22000 0.00004832

23000 0.00004958

24000 0.00005323

25000 0.00005383

26000 0.00005569

27000 0.00005815

28000 0.00006025

29000 0.00006473

30000 0.00006531

31000 0.00006769

32000 0.00006959

33000 0.00007223

34000 0.00007473

35000 0.00007634

36000 0.00007813

37000 0.00008010

38000 0.00008367

39000 0.00008449

40000 0.00008763

41000 0.00008921

42000 0.00009288

43000 0.00009401

44000 0.00009695

45000 0.00009960

46000 0.00010069

47000 0.00010303

48000 0.00010474

49000 0.00010677

50000 0.00011321

51000 0.00011189

52000 0.00011349

53000 0.00011664

54000 0.00011930

55000 0.00011933

56000 0.00012171

57000 0.00012526

58000 0.00012798

59000 0.00012982

60000 0.00013176

3. Using the profiling data from main, I plotted “Seconds per insert” (Y-axis) vs. “Array length” (X-axis).



4. A line-by-line Big-O analysis is shown by the inline comments in the source code, as can be seen from page 1. The overall time complexity of this insert function is O(n), where n is the length of the array. In this function, copying data from input array to the new array contributes most heavily to the overall time complexity.

5. From the plot, the seconds-per-insert scales linearly as the array length grows. This matches the Big-O analysis of O(n). In this respect, the performance roughly stays the same.

6. I followed the best practices when doing this homework.