

CST 370 – Fall (B) 2021
Homework 5
Due: 12/07/2021 (Tuesday) (11:55 PM)

How to turn in: Write **three programs** in either **C++ or Java** and submit them on Canvas before the due.

- You can submit your programs **multiple times** before the due. However, the **last submission will be used for grading**.
- You have to **submit three programs together**, especially at your last submission. **If you submit, for example, only one program at the last submission, we are able to see only that program when we grade** your homework.
- Due time is 11:55(PM). Since there could be a long delay between your computer and Canvas, you should submit it early.
- When you submit your homework program, don't forget to include "Title", "Abstract", "ID", "Name", and "Date".

1. Write a C++ (or Java) program called **hw5_1.cpp (or hw5_1.java)** to conduct heap operations.

Input format: This is a sample input from a user.

```
5
10 20 30 40 70
5
displayMax
insert 50
insert 15
deleteMax
display
```

The first line (= 5 in the example) indicates that there are five numbers in the second line (= 10, 20, 30, 40, and 70 in the example). In the program, you can **assume that all numbers in the heap are unique (= no duplicates)**. Your program should read the numbers and display if it's a max heap or not. If it's not a max heap, your program should construct a heap with the numbers. Thus, this is the heap built with the input data.

70

40

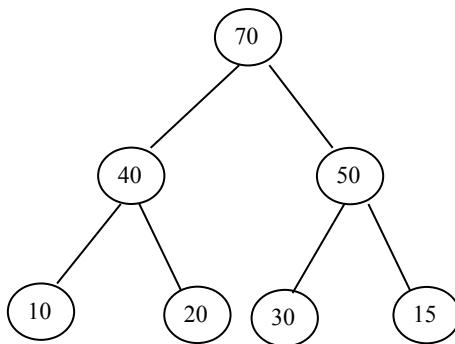
30

10

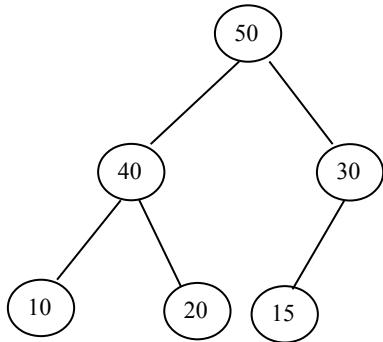
20

The third line (= 5 in the example) indicates the number of commands you have to conduct to the heap. The commands include “displayMax”, “insert” (= insert a number to the heap and do the “heapify” to adjust the heap), “deleteMax”, and “display” (= display all nodes in the heap on the screen).

For the “displayMax” command, your program should display “70” on the screen. After that, your program should insert “50” and “15” to the heap. This is the result after the two insert operations.



For the “deleteMax” command, your program should delete the max (= 70) from the heap. This is the result.



For the “display” command, your program should display the nodes in the heap (= 50 40 30 10 20 15)

Sample Run 0: Assume that the user typed the following lines

```
5
10 20 30 40 70
5
displayMax
insert 50
insert 15
deleteMax
display
```

This is the correct output. For the input data, your program should display that it's not a heap. Then, 70 is the current max of the heap for the command "displayMax". The result (= 50 40 30 10 20 15) is the result of "display" command.

```
This is NOT a heap.  
70  
50 40 30 10 20 15
```

Sample Run 1: Assume that the user typed the following lines

```
6  
20 10 8 1 3 5  
4  
display  
deleteMax  
displayMax  
display
```

This is the correct output.

```
This is a heap.  
20 10 8 1 3 5  
10  
10 5 8 1 3
```

Sample Run 2: Assume that the user typed the following lines

```
11  
99 55 88 44 33 66 77 22 11 5 3  
4  
insert 200  
display  
insert 100  
display
```

This is the correct output.

```
This is a heap.  
200 55 99 44 33 88 77 22 11 5 3 66  
200 55 100 44 33 99 77 22 11 5 3 66 88
```

2. Write a C++ (or Java) program named **hw5_2.cpp (or hw5_2.java)** that displays the performance of two different sorting algorithms (= **merge sort** and **quick sort**) on the screen.

Your program should read the input size from a user and generate random integer numbers of the input size. After that, your program should run the sorting algorithms for the input data and display execution time (= elapsed time) for each sorting algorithm.

For the homework, your program should be able to hold the input data of a large size such as 250,000, 500,000, up to 1,000,000. Of course, your program should also work well with the input data of a small size such as 1, 2, or 10.

For the homework, you can't use a library sorting function. **If your program uses a library sorting function**, you will get **zero without grading**. But you **can use sorting programs available on the Internet for the merge and quick sorts**. For the sorting programs available on the Internet, add the source program's website(s) in your head comment to indicate the original source of the programs.

For the grading, we **will not test with more than 1,000,000 input numbers**. For the input data of a large size such as 500,000 or 1,000,000, **the quick sort should present better performance than the merge sort, if your program is developed properly**.

This is a sample result of your program:

```
Enter input size: 350
=====
Execution Time =====
Merge sort: 0.001234 milliseconds
Quick sort: 0.001234 milliseconds
=====
```

In the sample execution, your program generates 350 random integer numbers. For the homework, you have to display the “**elapsed time**”. Note that the elapsed time in **the sample run (= 0.001234 milliseconds) is not an actual execution time**. It is a meaningless number entered by the instructor. In your case, your program should display actual execution time in milliseconds or seconds. When you display the elapsed time, **exclude the time to generate the input data**. You have to **measure only the sorting time**. Also, the two sorting programs should use the same input data.

Note that **we will test your program on AWS**. Thus, you have to **make sure that your program runs well on AWS**.

This is another sample result:

```
Enter input size: 500000
=====
Execution Time =====
Merge sort: 1234.56789 milliseconds
Quick sort: 1234.56789 milliseconds
=====
```

To help your development, we will use the following test cases for the hw5_2 grading.

Test case 1:

Testing item:

Check if the program uses a library sorting function or not.

If it uses a library sorting function, the program will get zero without further testing.

Test case 2:

Input size: 1

Testing item:

The program should display the execution result.

Program should not crash.

Execution should finish immediately.

Test case 3:

Input size: 5

Testing item:

The program should display the execution result.

Program should not crash.

Execution should finish immediately.

Test case 4:

Input size: 1,000

The program should display the execution result.

Program should not crash.

Execution should finish immediately.

Test case 5:

Input size: 50,000

The program should display the execution result.

Program should not crash.

Execution should not take more than 10 seconds.

Test case 6:

Input size: 250,000

The program should display the execution result.

Program should not crash.

Execution should not take more than 10 seconds.

Test case 7:

Input size: 1,000,000

Testing item:

Quicksort should be faster than merge sort.

Program should not crash.

Execution should not take more than 20 seconds.

3. Write a C++ (or Java) program called **hw5_3.cpp (or hw5_3.java)** to simulate the operations of **linear probing** covered in the lecture. For the sample operation, watch <https://youtu.be/yAaUTDuiaqY>

Input format: This is a sample input from a user.

```
5
10
insert 17
insert 12
displayStatus 2
tableSize
insert 20
tableSize
search 20
search 15
displayStatus 1
displayStatus 2
```

The first line (= 5 in the example) is the initial size of the hash table. The size will be always a prime number. The second line (= 10 in the example) indicates the number of commands you have to conduct to the hash table. The commands include “insert” (= insert a key to the table), “displayStatus” (= display the status of an entry in the table), “tableSize” (= display the size of the table), and “search” (= search a key in the table).

For the first two “insert” commands, the table will be like below. For the homework, you can **assume that the “insert” numbers are non-negative**. They will be zero or greater than zero.

| Index | Key Value |
|-------|-----------|
| 0 | |
| 1 | |
| 2 | 17 |
| 3 | 12 |
| 4 | |

Note that if the load factor becomes **greater than 0.5** after a new insert, you have to conduct the **rehashing**. In other words, you have to find the first prime number after doubling the current table size and move the keys in the current table to the new table. After that, you have to insert the new key value. The following presents the result after the “insert 20” command. For this homework, you **can assume that the table size is always less than 200**. In other words, we will not test the case which requires a table size with more than or equal to 200.

| Index | Key Value |
|-------|-----------|
| 0 | |
| 1 | 12 |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | 17 |
| 7 | |
| 8 | |
| 9 | 20 |
| 10 | |

Sample Run 0: Assume that the user typed the following lines

```
5
10
insert 17
insert 12
displayStatus 2
tableSize
insert 20
tableSize
search 20
search 15
displayStatus 1
displayStatus 2
```

This is the correct output. For the “displayStatus” command, your program should display the status of an entry of the table. For example, your program should display “17” for the first “displayStatus 2” command. For the second “displayStatus 2” command, it should display “Empty”.

```
17
5
11
20 Found
15 Not found
12
Empty
```

Sample Run 1: Assume that the user typed the following lines

```
7
7
insert 100
insert 16
insert 37
displayStatus 3
displayStatus 2
search 37
tableSize
```

This is the correct output.

```
16
100
37 Found
7
```

Sample Run 2: Assume that the user typed the following lines

```
8
insert 97
tableSize
insert 1000
insert 2000
insert 3000
insert 4000
displayStatus 0
displayStatus 1
```

This is the correct output.

```
97
97
Empty
```