EEE2020-03 **Spring 2021**

Assignment 7: Sorting

Due: Sunday, June 20, 2021, 11:59PM

1 Introduction

- The objective of this assignment is to implement a sorting algorithm.
- In particular, you will have to write the *heap sort* based on the lecture slides of `7_sorting.pdf`.
- In Ubuntu or Mac terminal, use `wget` to download a copy of skeleton code compressed into `sorting.tar`.

```
$ wget https://icsl.yonsei.ac.kr/wp-content/uploads/sorting.tar
```

- Decompress the `tar` file, and go to the `sorting/` directory. Typing `ls` should show the following list of files.

```
$ tar xf sorting.tar
$ cd sorting/
$ ls
data.h  input  input.h  main.cc  Makefile  sort.cc  sort.h  tar.sh
```

- From the listed files, you will only have to work on the `sort.cc` file. Other files are all complete.
- You are allowed to change other files for your own testing and validation, but they will revert to the original state when your assignment is graded.
  - `data.h` defines the data type of numbers to sort. It is defined as `uint32_t` (i.e., `unsigned int`).
  - `sort.*` files define `heapsort()` function that is supposed to perform the heap sort.
  - `input.h` defines `class input_t`, and it loads two million random integer numbers from an input file named `input`.
  - `main.cc` has the `main()` function.
- To compile the code, type `make` in the terminal. `Makefile` that comes along with the skeleton code has automated all the compiling scripts for you.

```
$ make
g++ -Wall -Werror -g -o sort.o -c sort.cc
g++ -Wall -Werror -g -o main.o -c main.cc
g++ -o sort sort.o main.o
```

- Executing the skeleton code prints the following output.

```
$ ./sort
array = [ 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 ]
array = [ 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 ]
```

2 Implementation

- The following shows the `heapsort()` function in `sort.cc`. Fill in the function to sort numbers in the input array based on the heap sort algorithm.
- The function takes two input arguments, `array` and `size`. The first argument is a pointer to the data array, and the second argument indicates the number of elements in the array.

- All cells in the data array are filled with numbers including the index #0.

```
/* sort.cc */

#include "sort.h"

// Heap sort
void heapsort(data_t* array, const size_t size) {
    /* Assignment */
}
```

- The next shows the `main()` function that is organized in a very similar way as those of prior assignments.
- In the preliminary testing part of `main()` between the lines of `* TEST YOUR CODE HERE FOR VALIDATION *` and `* END OF TESTING *`, it creates a 20-element array and fills it with random integers between 0 and 99.
- Then, the `sort()` function is called to sort the numbers in the array. `sort()` is linked to (or an alias of) `heapsort()` at the beginning of `main()`.
- You may use this sample test and modify it as necessary to validate your heap sort implementation.
- The later part of `main()` loads two million random integers from `input` to grade your assignment.

```
/* main.cc */

#include <string>
#include "input.h"
#include "sort.h"

int main(int argc, char **argv) {
    // Function pointer to heapsort
    void (*sort)(data_t*, const size_t) = &heapsort;
    // Data array and size
    data_t *array = 0;
    size_t array_size = 0;







    /************************************
     * TEST YOUR CODE HERE FOR VALIDATION *
     ************************************/

    // Allocate a small test array.
    array_size = 20;
    array = new data_t[array_size];


    // Generate random integers between 0 and 99.
    std::cout << "array = [ ";
    for(size_t i = 0; i < array_size; i++) {
        array[i] = rand() % 100;
        std::cout << array[i] << " ";
    }   std::cout << "]" << std::endl;


    // Sort the numbers.
    sort(array, array_size);
```

```
    // Print the array elements.
    std::cout << "array = [ ";
    for(size_t i = 0; i < array_size; i++) {
        std::cout << array[i] << " ";
    }   std::cout << "]" << std::endl;


    // Deallocate the array.
    delete [] array;

    /******************
     * END OF TESTING *
     ******************/




    /* ************************************************
     * WARNING: DO NOT MODIFY THE CODE BELOW THIS LINE *
     ************************************************/

    // Proceed?
    if(argc != 2) { return 0; }

    // Load numbers from an input file.
    input_t input(argv[1]);
    array = input.get_array();
    array_size = input.size();

    std::cout << std::endl << "Sorting " << array_size << " numbers: ";

    // Sort the numbers.
    sort(array, array_size);

    // Check if the array is sorted.
    std::cout << "Array is " << (is_sorted(array, array_size) ? "" : "NOT ")
              << "sorted "   << std::endl;

    return 0;
}
```

- After the implementation is done, executing the code will produce the following output.

```
$ ./sort
array = [ 7 49 73 58 30 72 44 78 23 9 40 65 92 42 87 3 27 29 40 12 ]
array = [ 3 7 9 12 23 27 29 30 40 40 42 44 49 58 65 72 73 78 87 92 ]
```

- To test the program with the `input` file, add `input` to the run command as follows.

```
$ ./sort input
array = [ 7 49 73 58 30 72 44 78 23 9 40 65 92 42 87 3 27 29 40 12 ]
array = [ 3 7 9 12 23 27 29 30 40 40 42 44 49 58 65 72 73 78 87 92 ]

Sorting 2000000 numbers: Array is sorted
```

- If your heap sort algorithm well follows the $O(N\log N)$ performance, sorting two million numbers should not take more than a few seconds.
- To measure the runtime of the program, you can prepend `time` to the run command as follows. For instance, executing the solution code takes about 0.3 seconds to sort the two million numbers.

```
$ time ./sort input
array = [ 83 86 77 15 93 35 86 92 49 21 62 27 90 59 63 26 40 26 72 36 ]
array = [ 15 21 26 26 27 35 36 40 49 59 62 63 72 77 83 86 86 90 92 93 ]

Sorting 2000000 numbers: Array is sorted

real    0m0.318s
user    0m0.306s
sys     0m0.013s
```

- Even if your code compiles and runs to completion with correct results, you need to double-check the code with `valgrind` to confirm no memory leaks.

```
$ valgrind ./sort input

...

==2020== All heap blocks were freed -- no leaks are possible

...
```

- In Mac OS, `valgrind` is not well supported. You may use `leaks` instead to test memory leaks.

```
$ leaks --atExit -- ./sort input

...

Process 2020: 0 leaks for 0 total leaked bytes.

...
```

3 Submission

- When the assignment is done, execute the `tar.sh` script in the `sorting/` directory.
- It will compress the `sorting/` directory except the 8MB `input` file into a `tar` file named after your student ID such as `2020140000.tar`,

```
$ ./tar.sh
rm -f sort.o main.o sort
sorting/
sorting/main.cc
sorting/Makefile
sorting/sort.h
sorting/sort.cc
sorting/data.h
sorting/tar.sh
sorting/input.h

$ ls
2020140000.tar  input    main.cc   sort.cc  tar.sh
data.h          input.h  Makefile  sort.h
```

- Upload the tar file (e.g., `2020140000.tar`) on LearnUs. Do not rename the `tar` file or C++ files included in it.

4 Grading Rules

- The following is the general guideline for grading. 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change, and the grader may add a few extra rules for fair evaluation of students' efforts.

  -3 points: A submitted `tar` file is renamed and includes redundant tags such as `hw7`, student name, etc.

  -5 points: A program code does not have sufficient amount of comments. Comments in the skeleton code do not count. You must make an effort to clearly explain what each part of your code intends to do.

  -15 points: The code compiles and runs, but it has memory leaks or crashes at the end with errors.

  -25 points: The code does not compile or fails to run (e.g., no or completely wrong results), but it shows substantial efforts to complete the assignment.

  -25 points: The code takes unbearably long to run. The execution time of a program strongly depends on the speed of executing hardware, so a runtime limit is not given as a hard bound. However, the code should not take more than a few seconds even if your computer is terribly slow. If the code does not complete within several seconds, the TA will not be able to grade your assignment unless the grader waits several hours for your ugly code to finish. Such a code will be regarded as a failure.

  -30 points: No or late submission. The following cases will also be regarded as no submissions.

  * Little to no efforts in the code (e.g., submitting nearly the same version of code as the skeleton code) will be regarded as no submission. Even if the code is incomplete, you must make substantial amount of efforts to earn partial credits.
  * Fake codes will be regarded as cheating attempts and thus not graded. Examples of fake codes are i) hard-coding a program to print expected outputs to deceive the grader as if the program is correctly running, ii) copying and pasting random stuff found on the Internet to make the code look as if some efforts are made. More serious penalties may be considered if students abuse the grading rules.

  Final grade = F: The submitted code is copied from someone else. All students involved in the incident will be penalized and given F for the final grades irrespective of assignments, attendance, etc.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect for any reasons, feel free to discuss your concerns with the TA. In case no agreement is made between you and the TA, elevate the case to the instructor to review your assignment. Refer to the course website for the contact information of TA and instructor: `https://icsl.yonsei.ac.kr/eee2020`

- Begging partial credits for no valid reasons will be treated as a cheating attempt, and such a student will lose all scores of the assignment.