**CSE310-2023Fall-78596 Project 2 (Updated: 10/14/2023)**

**Due: 11/5/2023 11:59pm**

**This should be your individual work. While you can use the internet to aid your work, you should write your own code.** As in the case of your first programming project, It should be written using the standard C++ programming language, and compiled using the g++ compiler on a Linux platform. Your project will be graded on Gradescope,which uses the Ubuntu 22.04 version of Linux. If you compile your project on general.asu.edu using the compiler commands provided in the sample Makefile, you should expect the same behavior of your project on Gradescope. This is a project, you will not be given any starter code except for the Makefile and data\_structures.h, which is also a suggestion. The Makefile will give you hints on what files are required. But you have the choice to make your own Makefile, and need to submit it with the rest of the code.

# **Data Structures and Functions**

In class, we studied the **max heap** data structure and the basic max heap functions Heapify, BuildHeap, ExtractMax, IncreaseKey, and Insertion. Symmetrically, we have the **min heap** data structure (in which the key value at a node cannot be smaller than the value at its parent) and the corresponding basic min heap functions Heapify, BuildHeap, ExtractMin, DecreaseKey, and Insertion. In this project, you will implement the **min heap** data structure.

During the class presentation, we simplified our explanation by assuming that A[i] is of type int for all valid indices. However, for this project, A[i] should be either of type ELEMENT\* or ELEMENT, depending on your specific implementation. ELEMENT is a struct that includes a field named 'key' of type double, along with other fields. Consequently, the heap order is determined by either A[i]->key or A[i].key, depending on your chosen implementation.

## **1.1** **Data Types**

The project description will make reference to the following two data types. You may use them with modifications, but you do not have to use them.

| **typedef** **struct** **TAG\_ELEMENT**{  **double** key;  // other fields as you see fit }ELEMENT;   **typedef** **struct** **TAG\_HEAP**{  **int** capacity; /\* capacity of the heap \*/  **int** size; /\* current size of the heap \*/  ELEMENT \*\*A; /\* array of pointers to ELEMENT \*/  // other fields as you see fit }HEAP; |
| --- |

# **Modular Design**

In your project, you'll continue to follow a modular design approach. You should provide a **Makefile** to compile various modules in order to generate an executable file named **PJ2**. Your project should include, at a minimum, the following modules:

1. **main.cpp**: This module coordinates all other modules.
2. **util.h** and **util.cpp**: These modules offer utility services, including command line interpretation.
3. **heap.h** and **heap.cpp**: These modules are responsible for implementing the functions related to the min heap.

For each module, except the main program, you should have:

* A header file specifying the data structures and function prototypes within the module.
* An implementation file that implements all the functions defined in the header file.

It is recommended that you define all data types in a header file named **data structures.h** to keep your project well-organized.

# **Flow of the Project**

## **3.1** **Valid Execution**

A valid execution of your project has the following form:

| ./PJ2 <ifile> <ofile> flag |
| --- |

where **./PJ2** tells the system to search for the executable file **PJ2** in the current directory, **<ifile>** is the input file, **<ofile>** is the output file, and **flag** is either 0 or 1. When the value of **flag** is 1, your program counts the total number of **Heapify** calls for each **BuildHeap** or **ExtractMin** operation. When the value of **flag** is 0, your program does not count the total number of **Heapify** calls.

Your program should check whether the execution is valid. If the execution is not valid, your program should print out the following message to **stderr** and stop.

| Usage: ./PJ2 <ifile> <ofile> flag |
| --- |

Note that your program should not crash when the execution is not valid. Upon a valid execution, your program should expect the following instructions from **stdin** and act accordingly:

**Stop:**

On reading Stop, the program stops.

**Init <Capacity>**:

On reading Init <Capacity>, the program should do the following.

1. Allocate memory for an object of type HEAP, and return a pointer to this object. Please note that your program should also allocate memory for an heap array of size <Capacity>. You should also set the size of this HEAP object to 0.
2. Wait for the next instruction from stdin.

In case of a memory allocation failure during the above processes, your program should adhere to the following:

1. If the first memory allocation fails, return a null pointer and write an error message to stderr. Importantly, the program should not proceed to the second memory allocation.
2. However, if the second memory allocation fails, your program should free the object allocated in the first memory allocation and also write an error message to stderr.

**Print:**

On reading Print, the program should do the following.

1. If the heap is NULL, print the following error message to stderr:

| Error: heap is NUll |
| --- |

then wait for the next instruction from stdin, skipping the following action(s).

1. If the heap is not NULL, print the current state of the heap to stdout then wait for the next instruction from stdin.

When the heap is not NULL, the program writes to stdout the size of the heap (not the capacity), followed by the key values of the elements in the heap (refer to posted test cases for output format).

**Write:**

On reading Write, the program should do the following.

1. If the heap is NULL, print the following error message to stderr:

| Error: heap is NULL |
| --- |

then wait for the next instruction from stdin, skipping the following action(s).

1. Opens the file argv[2] in write mode.

If the file is not opened successfully, write an error message to stderr and wait for the next instruction from stdin, skipping the following actions.

1. Write the heap information to the file argv[2] (refer to posted test cases for format). Close the file argv[2].
2. Wait for the next instruction from stdin.

**Read:**

On reading Read, the program should do the following.

1. If the heap is NULL, print the following error message to stderr:

| Error: heap is NULL |
| --- |

then wait for the next instruction from stdin, skipping the following action(s).

1. Open the file argv[1] in read mode.

If the file is not opened successfully, write an error message to stderr and wait for the next instruction from stdin, skipping the following actions.

1. Read in the first integer, n, from the file opened.

If heap->capacity is smaller than n, write an error message to stderr, close the file argv[1], and wait for the next instruction from stdin, skip the following actions.

1. For each value of *j* = 1*,* 2*, . . . , n* read in *keyj* from the input file; dynamically allocate memory for an object of type ELEMENT; sets the key field of this object to *keyj*; let the *j*-th element of the corresponding array in the heap point to this object. Close the file argv[1].
2. Apply the *O*(*n*) time BuildHeap function to build the min heap.
3. Wait for the next instruction from stdin.

**Insert <Key>:**

On reading Insert <Key>, the program should do the following.

1. If the heap is NULL, print the following error message to stderr:

| Error: heap is NULL |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. If the heap is full, print the following error message to stderr:

| Error: heap is full |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. Dynamically allocate memory for an object of type ELEMENT. Set the key field of this object to <Key>. Insert this object to the min heap.
2. Wait for the next instruction from stdin.

**ExtractMin**:

On reading ExtractMin, the program should do the following.

1. If the heap is NULL, print the following error message to stderr:

| Error: heap is NULL |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. If the heap is empty, print the following error message to stderr:

| Error: heap is empty |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. Perform the ExtractMin operation. Write the key value of the extracted object to stdout using the format

| fprintf(stdout, "ExtractMin: %lf\n", Key); |
| --- |

where Key is the value of the key field of the extracted object.

1. Wait for the next instruction from stdin.

**DecreaseKey <Postion> <NewKey>:**

On reading DecreaseKey <Position> <NewKey>, the program should do the following. Here <Position> is the index to the array for the **min heap** in the given data structure, and <NewKey> is the new value of the key field of the object pointed to by the corresponding array at index <Position>.

1. If the heap is NULL, print the following error message to stderr:

| Error: heap is NULL |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. If the heap is empty, print the following error message to stderr:

| Error: heap is empty |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. If <Position> if out of range or <NewKey> is not smaller than the corrent key, print the following error message to stderr:

| Error: invalid call to DecreaseKey |
| --- |

then wait for the next instruction from stdin, skipping the following actions.

1. Decrease the key field of the corresponding object to <NewKey> and perform the corresponding operations on the min heap.
2. Wait for the next instruction from stdin.

**Unknown Instructions:**

If your program reads in an unknown instruction (other than the ones listed in the above), the program should do the following.

1. Write the following message to stderr

| Warning: Invalid instruction |
| --- |

1. Wait for the next instruction from stdin.

# **Format of the Input File**

The input file specified by argv[1] is an ascii file. It contains an integer n, followed by *n* real numbers, where two numbers are separated by one or more white spaces. Here a white space one of the three characters in the set {’ ’, ’\t’, ’\n’}.

The first integer, call it *n*, is the number of key values. The next *n* real numbers are the first key value, the second key value, . . . , the n-th key value. An example input file input has the following content.

| 7 17 16 15 14 13 12 11 |
| --- |

It contains the same information as the following file:

| 7 17.0 16 15 14.00 13 12 11 |
| --- |

# **Submission**

You should submit your project to Gradescope via the link on Canvas. Submit your Makefile along with all header files and implementation files. You should put your name and ASU ID at the top of each of the header files and the implementation files, as a comment.

Submissions are always due before 11:59pm on the deadline date. Do not expect the clock on your machine to be synchronized with the one on Canvas/Gradescope. This project is due on Sunday, 11/05/2023. It is your responsibility to submit your project well before the deadline. **Since you have about three weeks to work on this project, no extension request (too busy, sick, need more time accommodations) is a valid one.**

**Grading**

**All programs will be compiled and graded on Gradescope**. **If your program does not compile and work on Gradescope, you will receive** 0 **on this project**. If your program works well on general.asu.edu, there should not be much problems. The maximum possible points for this project is 100. The following shows how you can have points deducted.

1. **Non-working program**: If your program does not compile or does not execute on Gradescope, you will receive a 0 on this project. Do not claim “my program works perfectly on my PC, but I do not know how to use Gradescope.”
2. **Posted test cases**: For each of the 20 posted test cases that your program fails, 4 points will be marked off.
3. **UN-posted test cases**: For each of the 5 un-posted test cases that your program fails, 4 points will be marked off.

**Examples**

In this section, I provide some examples. All examples assume that the input file is named input.txt and has the following content:

| 7 17.0 16 15 14.00 13 12 11 |
| --- |

## **Example 1**

Execution line is the following:

| ./PJ2 |
| --- |

This is an invalid execution. The program writes the following error message to stderr and terminates.

Usage: PJ2 <ifile> <ofile> flag

## **Example 2**

Execution line is the following:

| ./PJ2 input.txt out 0 |
| --- |

The instructions from stdin are as follows:

| Init 20  Read  Print  Write  Stop |
| --- |

This is a valid execution. The program writes the following to stdout:

| 7 11.000000 13.000000 12.000000 14.000000 16.000000 17.000000 15.000000 |
| --- |

writes the following to the file out:

| 7 11.000000 13.000000 12.000000 14.000000 16.000000 17.000000 15.000000 |
| --- |

and terminates.

## **Example 3**

Execution line is the following:

| ./PJ2 input.txt out 1 |
| --- |

The instructions from stdin are as follows:

| Init 20  Read  Print  Write  Stop |
| --- |

This is a valid execution. The program writes the following to stdout:

| Number of Heapify calls: 7 7 11.000000 13.000000 12.000000 14.000000 16.000000 17.000000 15.000000 |
| --- |

writes the following to the file out:

| 7 11.000000 13.000000 12.000000 14.000000 16.000000 17.000000 15.000000 |
| --- |

and terminates.

## **Example 4**

The instructions from stdin are as follows:

| Init 20  Read  Print  Insert 1  Print  Stop |
| --- |

The execution line is the following:

| ./PJ2 input.txt out 0 |
| --- |

This is a valid execution. The program writes the following to stdout and terminates.

| 7 11.000000 13.000000 12.000000 14.000000 16.000000 17.000000 15.000000 8 1.000000 11.000000 12.000000 13.000000 16.000000 17.000000 15.000000 14.000000 |
| --- |

and terminates.

# **Test Cases**

Test cases will be in the test\_cases\_test\_script.zip in the shared drive. Unzip it, and add the posted\_test\_cases folder in the root directory of your project. So should the other files. Below is the suggested folder structure.

| Project2/  ├─ posted\_test\_cases/  │ ├─ test1/  │ │ ├─ ifile  │ │ ├─ ofile  │ │ ├─ Execution  │ │ ├─ Commands  ├─ run\_posted\_test\_cases.sh  ├─ Makefile  ├─ PJ2  ├─ make.cpp  ├─ util.cpp  ├─ util.h  ├─ heap.cpp  ├─ heap.h  ├─ data\_structure.h |
| --- |

To run all the test cases:

| Project2> make #this just builds your code Project2> chmod +x run\_posted\_test\_cases.sh # this allows you to execute the script Project2> ./run\_posted\_test\_cases.sh |
| --- |

To run one specific test cases

| Project2> ./PJ2 posted\_test\_cases/test01/ifile posted\_test\_cases/test01/myoutput 0 < posted\_test\_cases/test01/Commands > posted\_test\_cases/test01/myscreenoutput  Project2> diff posted\_test\_cases/test01/ofile posted\_test\_cases/test01/myoutput |
| --- |

You can find the flags from the execution file in that test folder, it tells you what flag that test case was run against.

# **Suggested Steps**

We suggest the following steps.

## **Read in Instructions from** stdin

For this project, you are required to make modifications to the util module that was provided for Project 1. These modifications should ensure that the util module can cater to the specific instructions of this project. Additionally, you should update the corresponding section of the main program to facilitate the reading of these instructions. Rather than executing the corresponding actions, your main program should simply print out the instruction to stdout and then wait for the next instruction. However, if the instruction is Stop, your program should terminate.

## **Implement the Function for the** Init <Capacity> **Instruction**

You can write a function in heap.cpp that creates a HEAP object with the desired capacity and 0 size.

## **Implement the Function for the** Print **Instruction**

You can write a function in heap.cpp that prints the information of the HEAP object to stdout. With this function, you can print out the content of the heap object as needed in the debugging process.

## **Implement Other Functions**

Now you can implement other functions as needed. One function at a time. As you are implementing more and more functions, you can modify your main.cpp accordingly to take corresponding actions, instead of just printing out the next instruction to stdout.