CSE310 Project 2: Priority Queues

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. If you compile your project on `general.asu.edu` using the compiler commands provided in the sample `Makefile` for the first project, you should expect the same behavior of your project on both `general.asu.edu` and Gradescope. Therefore, you are highly recommended to develop your project on `general.asu.edu`.

For Project 1, I provided you with many help files for you to learn the right modular design style, as well as many examples for memory management. For Project 2, no help files are provided. It is assumed that you understand everything in the help files provided to you in Project 1, and can modify them for use in this project.

1 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.

1.1 Data Types

The project description will make reference to the following two data types. You may use them with modifications.

```
typedef struct TAG_ELEMENT{
    int    index;  // index of the element
    double key;    // key value of the element
    int    pos;    // index in the heap-array, 0 if not in the heap
    // Other fields as you see fit
}ELEMENT;

typedef struct TAG_HEAP{
    int  capacity;  // capacity of the heap
```

```
    int  size;      // current size of the heap
//    int *H;         // array of pointers to indices of the ELEMENT array
    int *H;         // array of indices to the ELEMENT array
    // Other fields as you see fit
}HEAP;
```

With the declaration

```
ELEMENT **V;
HEAP     *pHeap;
```

you declare `V` as a pointer to pointer to `ELEMENT`, and `pHeap` as a pointer to `HEAP`. With properly dynamic memory allocation, you can use `V` as an array of pointers to `ELEMENT`. Then `V[i]` is a pointer to `ELEMENT`.

In this project, you need to work with both `V` and `pHeap`. If `V[i]->pos` is 0, the element pointed to by `V[i]` is not on `pHeap`. Otherwise, the element pointed to by `V[i]` is on the `pHeap`. More importantly, for any `i` such that `V[i]->pos > 0` we have
`pHeap->H[V[i]->pos] == i`.

For any `k` between 1 and `pHeap->size`, we have

`V[pHeap->H[k]]->pos == k.`

In class, for ease of presentation, we presented the functions for **max heap** under the assumption that `A[i]` is of type `int` for each valid index `i`. `A[i]` is also used as the key in the heap operations. In this project, the key of the `k`-th element in the heap array is `V[pHeap->H[k]]->key`.

2 Modular Design

You will continue to do modular design, using the provided `Makefile` to compile various modules to generate the executable file named `PJ2`. You need to have exactly the following modules:

1. `data_structures.h`

   which defines the data types described in Section 1.1 or their equivalent.

2. `main.h` and `main.cpp`, which coordinate all modules;

3. `util.h` and `util.cpp`, which provide utility services including the recognition of various instructions;

4. `heap.h` and `heap.cpp`, which implement the functions for min heap.

3 Flow of the Project

3.1 Valid Execution

A valid execution of your project has the following form:

```
./PJ2 <I-File> <O-File>
```

where `./PJ2` tells the system to search for the executable file `PJ2` in the current directory, `<I-File>` is the input file, and `<O-File>` is the output file.

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 <I-File> <O-File>
<I-File> is the input file
<O-File> is the output file
```

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 should write the following to `stdout` and stop.

  ```
  Instruction: Stop
  ```

- `Read`:
  On reading `Read`, the program should do the following.

  1. Write the following line to `stdout`:

     ```
     Instruction: Read
     ```

  2. Open the file `argv[1]` in read mode. We will call this file the *input file*.
     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.

  3. Read in the first integer, `n`, from the input file.
     Perform proper memory allocation so that `V` can be used as an array name, and allocate memory for `V[i]` so that we can access the fields of `V[i]` for `i=1, 2, ..., n`. For each value of `i=1, 2, ..., n`, read in `key_i` from the input file; set

```
V[i]->index = i;
V[i]->key = key_i;
V[i]->pos = 0;
```

4. Close the file `argv[1]`.
5. Perform proper memory allocation so that `pHeap` points to a `HEAP` with capacity equal to `n` and size equal to `0`.
6. Wait for the next instruction from `stdin`.

- `PrintArray`:
  On reading `PrintArray`, the program should do the following.

  1. Write the following line to `stdout`:

     `Instruction: PrintArray`

  2. If the array `V` is `NULL`, print the following error message to `stderr`:

     `Error: array is NULL`

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

  3. If the array `V` is not `NULL`, print the current state of the array to `stdout` then wait for the next instruction from `stdin`.

  When the array `V` is not `NULL`, the program writes to `stdout` the `index` of the array, followed by the `key`, followed by the `pos` (refer to posted test cases for output format).

- `PrintHeap`:
  On reading `PrintHeap`, the program should do the following.

  1. Write the following line to `stdout`:

     `Instruction: PrintHeap`

  2. 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).

  3. 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 the heap information to `stdout` (refer to posted test cases for output format).

- `BuildHeap`:
  On reading `BuildHeap`, the program should do the following.

  1. Write the following line to `stdout`:

     `Instruction: BuildHeap`

  2. Set the size of the heap pointed to by `pHeap` to the number of elements in array `V`. For $i = 1$ `to` n `do` the following

     ```
     V[i]->pos = i;
     pHeap->H[i] = i;
     ```

     Then apply the linear-time BuildHeap algorithm taught in class.

  3. Wait for the next instruction from `stdin`.

  When the heap is not `NULL`, the program writes the heap information to `stdout` (refer to posted test cases for output format).

- `Insert <index>`:
  On reading `Insert <index>`, the program should do the following.

  1. Write the following line to `stdout`:

     `Instruction: Insert <index>`

     where `int` is printed using the "%d" format.

  2. If `index` is not within `1` and `n`, write the following to `stderr`:

     `Error: index out of bound`

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

  3. If `V[index]->pos != 0`, write the following to `stderr`:

     `Error: V[index] already in the heap`

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

  4. Insert `V[index]` into the heap. You start with the following statements:

```
pHeap->size = pHeap->size+1;
pHeap->H[pHeap->size] = index;
V[index]->pos = pHeap->size;
```

then make proper heap operations related to insertion. Write the following line to `stdout`:

```
Element V[index] inserted into the heap
```

where <`index`> (the index of the heap array where the element is inserted) is printed using the "%d" format.

5. Wait for the next instruction from `stdin`.

- `ExtractMin`:
  On reading `ExtractMin`, the program should do the following.

  1. Write the following line to `stdout`:

     ```
     Instruction: ExtractMin
     ```

  2. 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.

  3. 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.

  4. Perform the `ExtractMin` operation. Before the `ExtractMin` operation, we should have `V[pHeap->H[1]]->pos == 1`. Therefore `V[pHeap->H[1]]` is the element to be extracted from the heap.

  5. Wait for the next instruction from `stdin`.

- `DecreaseKey <index> <NewKey>`:
  On reading `DecreaseKey <index> <NewKey>`, the program should do the following.

  1. Write the following line to `stdout`:

     ```
     Instruction: DecreaseKey <index> <NewKey>
     ```

     where `int` is printed using the "%d" format and `double` is printed using the "%lf" format.

2. If `<index>` is out of range or `<NewKey>` is not smaller than the current 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.

3. If `V[index]->pos == 0`, write the following to `stderr`:

   ```
   Error: V[index] not in the heap
   ```

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

4. Decrease the `key` field of the corresponding object to `<NewKey>` and perform the corresponding operations on the min heap.

5. Wait for the next instruction from `stdin`.

- `Write`:
  On reading `Write`, the program should do the following.

  1. Write the following line to `stdout`:

     ```
     Instruction: Write
     ```

  2. 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.

  3. Write the array information (about `V`) to the output file. It should write `n` lines to the output file, where the `i`-th line contains

     ```
     i V[i]->key V[i]->pos
     ```

     where `int` is printed using the "%d" format and `double` is printed using the "%lf" format, with exactly one white space between values (refer to posted test cases for format).

  4. Close the output file `argv[2]`.

  5. 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 `stdout`

`Warning: Invalid instruction`

2. Wait for the next instruction from `stdin`.

4 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 is either a *space* (with ascii value 32), a *tab* (with ASCII value 9), or a *newline* (with ASCII value 10). In other words, it is one of the three characters in the set

`{' ', '\t', '\n'}`.

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

```
9
9 8 7
6
5
4
3 2 1
```

It contains the same information as the following file:

```
9


9.00
8
7
6
5
4
3
2
1
```

5 Output Format

When writing an `int`, use the "%d" format. When writing a `double`, use the "%lf" format. When writing multiple values in the same line, use exactly one *space* between values, here *space* has ASCII value 32.

6 Submission

You should submit your project to Gradescope via the link on Canvas. Submit 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. It is your responsibility to submit your project well before the deadline.
The instructor and the TAs will offer more help to this project early on, and will not answer emails/questions near the project due date that are clearly in the very early stage of the project. So, please manage your time, and start working on this project immediately. **You are requested to submit a version of your project on Gradescope on each of the Check Point dates.**

7 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.

8 Check Points

While you have a lot of time to work on this project, time goes very fast. You are required to submit a version of your project on Gradescope on each of the given Check Point days.

9 Examples

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

```
9


9
8
7
6
5
4
3
2
1
```

9.1 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 <I-File> <O-File>
<I-File> is the input file
<O-File> is the output file
```

9.2 Example 2

Execution line is the following:

```
./PJ2 I-File my-O-File
```

The instructions from `stdin` are as follows:

```
Read
PrintArray
PrintHeap
BuildHeap
PrintHeap
Write
Stop
```

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

```
Instruction: Read
Instruction: PrintArray
1 9.000000 0
2 8.000000 0
3 7.000000 0
4 6.000000 0
5 5.000000 0
6 4.000000 0
7 3.000000 0
8 2.000000 0
9 1.000000 0
Instruction: PrintHeap
Capacity = 9, size = 0
Instruction: BuildHeap
Instruction: PrintHeap
Capacity = 9, size = 9
H[1] = 9
H[2] = 8
H[3] = 7
H[4] = 4
H[5] = 5
H[6] = 6
H[7] = 3
```

```
H[8] = 2
H[9] = 1
Instruction: Write
Instruction: Stop
```

writes the following to the file `my-O-File`:

```
1 9.000000 9
2 8.000000 8
3 7.000000 7
4 6.000000 4
5 5.000000 5
6 4.000000 6
7 3.000000 3
8 2.000000 2
9 1.000000 1
```

and terminates.

9.3 Example 3

Execution line is the following:

```
./PJ2 I-File my-O-File
```

The instructions from `stdin` are as follows:

```
Read
BuildHeap
PrintHeap
ExtractMin
PrintHeap
DecreaseKey 1 -1
PrintArray
PrintHeap
Write
Stop
```

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

```
Instruction: Read
Instruction: BuildHeap
Instruction: PrintHeap
Capacity = 9, size = 9
H[1] = 9
H[2] = 8
H[3] = 7
H[4] = 4
H[5] = 5
H[6] = 6
H[7] = 3
H[8] = 2
H[9] = 1
Instruction: ExtractMin
Instruction: PrintHeap
Capacity = 9, size = 8
H[1] = 8
H[2] = 5
H[3] = 7
H[4] = 4
H[5] = 1
H[6] = 6
H[7] = 3
H[8] = 2
Instruction: DecreaseKey 1 -1.000000
Instruction: PrintArray
1 -1.000000 1
2 8.000000 8
3 7.000000 7
4 6.000000 4
5 5.000000 5
6 4.000000 6
7 3.000000 3
8 2.000000 2
9 1.000000 0
```

```
Instruction: PrintHeap
Capacity = 9, size = 8
H[1] = 1
H[2] = 8
H[3] = 7
H[4] = 4
H[5] = 5
H[6] = 6
H[7] = 3
H[8] = 2
Instruction: Write
Instruction: Stop
```

writes the following to the file `my-O-File`:

```
1 -1.000000 1
2 8.000000 8
3 7.000000 7
4 6.000000 4
5 5.000000 5
6 4.000000 6
7 3.000000 3
8 2.000000 2
9 1.000000 0
```

and terminates.

10 Test Cases and Autograder

There are 20 posted test cases and 5 hidden test cases. The autograder is also set up on Gradescope.