Dept. of Computer Science, University of California, Davis
ECS60 Instructor: Rob Gysel 2/4/18

Programming Assignment #2*
Due date: 2/15/18 11:59pm

Programs are to be submitted using handin on the CSIF by the due date using the command:

```
handin rsgysel 60-Program2 file1 file2 ... fileN
```

1 Overview & Learning Objectives

In this program you will implement a priority queue with an underlying binary heap implementation. There are multiple objectives of this assignment:

1. strengthen your knowledge of JSON,
2. strengthen your understanding of automated testing,
3. understand and implement a binary heap.

This program consists of three parts:

1. creating a priority queue with an underlying binary heap implementation (in C++ I implemented this as `priorityqueue.cpp` and `priorityqueue.h`),
2. use a priority queue to implement HeapSort (`heapsort.sh`; in C++ I implemented this as `heapsort.cxx`),
3. build a binary heap given a sequence of instructions (`buildheap.sh`; in C++ I implemented this as `buildheap.cxx`).

Example inputs with expected outputs are in the directory

```
~rsgysel/public/60-Program2-Examples
```

You can copy all of them to your current directory using:

```
cp ~rsgysel/public/60-Program2-Examples/* .
```

*Last updated February 4, 2018

You can also create your own examples for the two executables you are writing. For `heapsort.sh`, use `createsortingdata.exe`. For `buildheap.sh`, use `createheapoperationdata.exe`. Refer to the syllabus for group work policies. You may use Rust, Java, or C++ for your code.
Students that work alone for all programs a receive a B- or better on the programs will receive 1% extra credit at the end of the quarter. Programs submitted up to 24 hours late will still be accepted but incur a 10% grade penalty.

2 PriorityQueue

Create a Max-PriorityQueue as a C++ `class`, a Java `class`, or Rust `struct` called `PriorityQueue`. It must implemented as an array-based binary heap with the root node at index 1 of the array[1]. The keys are non-negative integers. You must implement:

Construction : When your priority queue is created, an integer `max_size` must be passed to the priority queue. It is *full* if the number of items in the priority queue is equal to `max_size`.

`insert(Key k)` : Insert the key `k` into the priority queue. If the priority queue is full and `insert` is called, print the following error and then exit:

```
PriorityQueue::insert called on full priority queue
```

`removeMax()` : Remove the maximum key from the priority queue. If the priority queue is empty and `removeMax` is called, print the following error and then exit:

```
PriorityQueue::removeMax called on an empty priority queue
```

`removeKey(Key k)` : Remove the key `k` from the priority queue. If this key does not exist in the priority queue, print the following error and then exit (in this example, `k = 45`):

```
PriorityQueue::removeKey key 45 not found
```

`change(Key k, Key newK)` : Change the key `k` to the key `newK`. If this key does not exist in the priority queue, print the following error and then exit (in this example, `k = 63`):

```
PriorityQueue::change key 63 not found
```

You may implement any other functions that you find helpful. Amongst others, I implemented `heapifyUp`, `heapifyDown`, and `JSON` (which prints a JSON object representing the priority queue; see part 4 for details on its structure).

[1]This is just to simplify the math. The 0th element in your array is present but unused.

3 HeapSort

The HeapSort algorithm works as follows, given an input array `A`.

1. Put all elements of `A` into a binary heap.
2. Extract the maximum element from the heap and place it at the last position of `A` that has not yet had a heap element placed in it. Repeat this until the heap is empty.

Implement `HeapSort` which reads a JSON file of integer arrays to be sorted. The format of this file is identical to that of Program 1. Write a shell script called `heapsort.sh` that runs your program on an input file. For example, my executable is called `heapsort.exe`, so the shell script I wrote is:

```
#/bin/bash
./heapsort.exe inputFile
```

To make sure that your code works, I suggest you use the code & testing procedures detailed in Program 1 to verify that your sorting algorithm is correct.

4 BuildHeap

Write a program that does the following:

1. reads a JSON file of heap operations,
2. executes the heap operations from the JSON file,
3. prints the priority queue as a JSON object to `stdout`.

The contents of `BuildExample.json`, an example of a JSON file of operations, is as follows:

```
{
  "Op01": {
    "key": 3804,
    "operation": "insert"
  },
  "Op02": {
    "key": 4035,
    "operation": "insert"
  },
  "Op03": {
    "key": 1755,
    "operation": "insert"
  },
```

```
  "Op04": {
    "operation": "removeMax"
  },
  "Op05": {
    "key": 2109,
    "operation": "insert"
  },
  "Op06": {
    "key": 3333,
    "operation": "insert"
  },
  "Op07": {
    "key": 105,
    "operation": "insert"
  },
  "Op08": {
    "operation": "removeMax"
  },
  "Op09": {
    "key": 1755,
    "newKey": 2634,
    "operation": "change"
  },
  "Op10": {
    "operation": "removeMax"
  },
  "metadata": {
    "maxHeapSize": 5,
    "numOperations": 10
  }
}
```

You can create these files using the executable `createheapoperationdata.exe`. After running build heap, my output[2] is:

```
{
  "1": {
    "key": 2634,
    "leftChild": "2",
    "rightChild": "3"
  },
  "2": {
```

[2]If you are using the `nlohmann::json` library, you can use `jsonObj.dump(2)` on an object `jsonObj` of type `nlohmann::json` to print a human-readable version of the json object.

```
    "key": 105,
    "parent": "1"
  },
  "3": {
    "key": 2109,
    "parent": "1"
  },
  "metadata": {
    "maxHeapSize": 5,
    "max_size": 5,
    "numOperations": 10,
    "size": 3
  }
}
```

Here, the top-level keys are either node data or metadata. The root node, a.k.a. node 1, has key 2634, its left child has key 105 (node with index 2), and its right child has key 2109 (node with index 3). Each node must contain the following key value pairs:

key: the key the node contains.

parent: the index of its parent node, if it exists (i.e. if it is not the root).

leftChild: the index of its left child, if it exists. Otherwise, this field must be omitted.

rightChild: the index of its right child, if it exists. Otherwise, this field must be omitted.

The metadata must contain the following key value pairs:

maxHeapSize: defined from the input file, this is the maximum heap size possible.

max_size: the `max_size` of the priority queue passed during construction.

numOperations: defined from the input file, this is the maximum heap size possible.

size: the number of elements in the priority queue.

To test of your code, use `createheapoperationdata.exe` to create a few operations and then run your `buildheap.sh` and examine its output. `createheapoperationdata.exe` will ensure that the heap operations it produces will not produce errors in a properly working implementation. For final testing, consider testing with a few million operations and verifying that no errors are produced.

5 Compilation

Submit a script called `compile.sh` that will compile all of your source code once it is run. For example, in the skeleton code I have provided, I have used `compile.sh` which uses a `Makefile` to build my code.