Program 3

Programs must be written in C++ and are to be submitted using handin on the CSIF by the due date (see Canvas) using the command:

[redacted] handin yafrid p2 file1 file 2

Your programs must compile and run on the CSIF. Use `handin` to submit *all* files that are required to compile (even if they come from the prompt). Programs that do not compile with `make` on the CSIF will lose points and possibly get a 0. Programs that have warnings during compilation will lose 10 out of 100 points.

An autograder along with examples and solutions will be available shortly.

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,
4. introduce greedy algorithms using a toy application (Volleyball team selection).

This program consists of three parts:

1. creating a priority queue with an underlying binary heap implementation (you will complete the files `priorityqueue.cpp` and `priorityqueue.h`),
2. build a binary heap given a sequence of instructions (you will complete the file `buildheap.cxx`),
3. use your priority queue to construct a *greedy algorithm* to pick "fair" 2-person teams from a list of Volleyball players.

2 PriorityQueue

Finish the (Minimum) `PriorityQueue` class in the files `priorityqueue.h` and `priorityqueue.cpp`. It must implemented as an array-based binary heap with the root node at index 1 of the array[1]. The keys are `double`s and the values are `std::pair`s of `int`s that represent player ids.
Some terminology: the $\texttt{i}^{\text{th}}$ node of the heap stores the `KeyValuePair` at index `i` in the array. That is, the $\texttt{1}^{\text{st}}$ node has the minimum `Key` in the heap since we are writing a minimum priority queue.
Implement the following member functions, which are marked in `priorityqueue.cpp` with `TODO`'s. I recommend implementing them in this order, as you should be using some of these functions when you implement the others.

`PriorityQueue::isEmpty()` : Returns `true` if and only if the heap is empty.

`PriorityQueue::size()` : Returns the number of `KeyValuePair`s in the heap.

`PriorityQueue::getKey(size_t i)` : Returns the key of the $\texttt{i}^{\text{th}}$ node in the heap.

`PriorityQueue::heapifyUp(size_t i)` : a.k.a. `swimup`, percolates the $\texttt{i}^{\text{th}}$ node *up* the heap to ensure the heap property is preserved.

`PriorityQueue::heapifyDown(size_t i)` : a.k.a. `sink`, percolates the $\texttt{i}^{\text{th}}$ node *down* the heap to ensure the heap property is preserved.

`PriorityQueue::removeNode(size_t i)` : Removes the $\texttt{i}^{\text{th}}$ node from the heap.

`PriorityQueue::min()` : Returns the `KeyValuePair` that has the minimum `Key` in the heap.

`PriorityQueue::removeMin()` : Returns and removes the `KeyValuePair` that has the minimum `Key` in the heap.

`PriorityQueue::insert(KeyValuePair kv)` : Inserts a key-value pair into the priority queue. The types `Key`, `Value`, and `KeyValuePair` are defined in `priorityqueue.h` using `typedef`s.

3 BuildHeap

Write a program that does the following:

1. reads a JSON file of heap operations (`insert` and `removeMin`),
2. executes the heap operations from the JSON file,
3. prints the priority queue as a JSON object to `stdout` (do not print output to a file).

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

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

```
{
  "Op01": {
    "key": 3019,
    "operation": "insert"
  },
  "Op02": {
    "key": 1500,
    "operation": "insert"
  },
  "Op03": {
    "key": 1334,
    "operation": "insert"
  },
  "Op04": {
    "key": 4119,
    "operation": "insert"
  },
  "Op05": {
    "operation": "removeMin"
  },
  "Op06": {
    "key": 2199,
    "operation": "insert"
  },
  "Op07": {
    "operation": "removeMin"
  },
  "Op08": {
    "operation": "removeMin"
  },
  "Op09": {
    "key": 4180,
    "operation": "insert"
  },
  "Op10": {
    "key": 4586,
    "operation": "insert"
  },
  "metadata": {
    "maxHeapSize": 4,
    "numOperations": 10
  }
}
```

After running build heap, my output using `std::cout << jsonObj.dump(2)` on the result is:


```
{
  "1": {
    "key": 3019.0,
    "leftChild": "2",
    "rightChild": "3",
    "value": [
      0,
      0
    ]
  },
  "2": {
    "key": 4119.0,
    "leftChild": "4",
    "parent": "1",
    "value": [
      0,
      0
    ]
  },
  "3": {
    "key": 4180.0,
    "parent": "1",
    "value": [
      0,
      0
    ]
  },
  "4": {
    "key": 4586.0,
    "parent": "2",
    "value": [
      0,
      0
    ]
  },
  "metadata": {
    "maxHeapSize": 4,
    "numOperations": 10,
    "size": 4
  }
}
```


Note that the values are ignored for `buildheap` so they default to a `std::pair` of 0's. This is the behavior coded in the `PriorityQueue::insert(Key k)` func-

tion, which has already been completed.
In the output, 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.

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

size: the number of elements in the priority queue.

To test your code, compile `createheapoperationdata.cxx` to create a JSON file with a few operations and then run `buildheap.exe` on this file and examine its output. `createheapoperationdata.cxx` 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. If you see errors, start again with small examples (10 operations) to see if you can reproduce the error. If the error is not reproduced with small examples, increase your operations by one order of magnitude (ex: 100 operations) and try to reproduce the error. Continue in this way and attempt to find the smallest number of operations required before trying to debug your errors. Once you have fixed your bug, return to testing with a few million operations.

4 CreateTeams

For the final part of this project, imagine that you are working for a summer camp and there are $2n$ kids. Your boss tells you to split the $2n$ children up into n teams of 2 so that everyone can play volleyball. Your boss then emphasizes that teams must be *fair*.
Your boss has been quite vague, and can't seem to clarify what "fair" means. You have data on all the past 2-person teams *win percentages*, that is, the

percent of games a 2-person team has won in the past[2]. You decide that "fair" can be interpreted as picking teams with win percentages as close to 50% as possible, and take it upon yourself to use the following reasonable strategy, which is known as a *greedy algorithm*:

1. Pick a 2-person team with a win percentage as close to 50% as possible.
2. For each of the two players in the 2-person team picked in part 1, remove all data of other 2-person teams that contain at least one of these players. That is, every player must be assigned to exactly one team.
3. Go to step 1 if fewer than n 2-person teams have been created.

Use your priority queue to implement the above greedy algorithm. An algorithm is called *greedy* if it makes a *locally optimal* choice. That is, the algorithm may not necessary give the *best* result[3], but it is going to give a *good* result. The *greedy choice* is made on line 1, where we arbitrarily pick a 2-person team that seems to be the best choice without considering the consequences of this action further down the line. Think of this as something like playing chess without thinking about any moves beyond the move you are making.

Your input files will be look like this, which is the contents of `CreateTeamsExample.json`:

```
{
  "metadata": {
    "numPlayers": 4
  },
  "teamStats": [
    {
      "playerOne": 0,
      "playerTwo": 1,
      "winPercentage": 55.11
    },
    {
      "playerOne": 0,
      "playerTwo": 2,
      "winPercentage": 29.748
    },
    {
      "playerOne": 0,
      "playerTwo": 3,
      "winPercentage": 14.872
```

[2]All children have played with all other children, and they have done this enough to have reasonable win percentage estimates. You're right, that's a lot of volleyball.

[3]We haven't even defined what "best" means for this problem! Does "best" minimize the sum of |win percentage minus 50%| for all teams? Does "best" minimize the maximum of |win percentage minus 50%| over all 2-person teams selected? These are two different *optimization criteria*, which lead to different algorithms and different "optimal" solutions.

```
    },
    {
      "playerOne": 1,
      "playerTwo": 2,
      "winPercentage": 53.724
    },
    {
      "playerOne": 1,
      "playerTwo": 3,
      "winPercentage": 95.86
    },
    {
      "playerOne": 2,
      "playerTwo": 3,
      "winPercentage": 65.498
    }
  ]
}
```

Here, the closest team to 50% has players 1 and 2 with a win percentage of 53.724. Once this team has been picked, the only players not currently assigned are players 0 and 3, who have a win percentage of 14.872. The final teams are stored in a JSON object and printed to `stdout` (do not print to a file):

```
{
  "teams": [
    [
      1,
      2
    ],
    [
      0,
      3
    ]
  ]
}
```

The `teams` key is an array that has two 2-person teams, (1,2) and (0,3). When you insert an `std::pair<int, int>` into an array in a `nlohmann::json` object, the result will be an array of size 2, so the teams are represented by the arrays [1, 2] and [0, 3].

Notice that, by almost any metric, the team assignments (0,1) and (2,3) seem more fair than the above assignments (at least on paper). This is okay, and this is how your algorithm should function. Greedy algorithms are generally fast, and while some greedy algorithms return optimal solutions (you will learn

about this in ECS122a), even those that don't can be useful if we are looking for a quick, "good enough" answer.

5 Compilation

A `Makefile` has been provided for you. If you add new files to your program, you will need to edit this `Makefile`. If you only edit the files given to you, you should not edit this `Makefile` in any way.
You may add more files to your program if you wish, but be sure that they are incorporated into your `Makefile`. I found it useful to extract the logic of reading the team statistics into an object called `TeamData`, so I have placed skeleton `teamdata.h` and `teamdata.cpp` files for this purpose. Using these files for this purpose is strongly suggested but is not required.