Ranking Up (Project 3)

Before you begin:

1. May I present to you a little reading, so we're all caught up on nomenclature: https://en.wikipedia.org/wiki/Online_algorithm

That is to say, we'll be saying offline & online, but you should know they *have different meanings* in the context of algorithms. Just because something's online doesn't mean it involves the web!

2. **More importantly**, while C++ implements heaps via priority-queues, we *won't* be using them for memory efficiency. We'll be using the heap operations from `<algorithm>`, which works with raw vectors. See them here.

Task 1: Offline Algorithms

Anywho, let's begin. Both of our Offline algorithms will achieve the same end but in different ways; we'll select & sort the top 10% of players by level given some collection of players.

Mind you, I calculate the top 10% of players using:

```
std::floor(0.1 * players.size())
```

However, to make your life easier, we'll only test your code using inputs with sizes that are perfect multiples of 10, so you shouldn't get tripped up!

Part A: Stop Heaping Around

This one's simple. Like only-a-couple-lines simple.

You've seen Heapsort before. As a reminder, it builds a heap, then pops everything until the entire heap has been exhausted and suddenly we have a sorted vector. We'll be doing something similar here, except only popping what we need to.

Your function should perform in-place. Besides the extra RankingResult vector. Remember, in-place just means constant O(1) memory. You shouldn't be making a separate priority queue, etc. Once again, we're twisting things a bit, because we're making an exception for the RankingResult vector (so technically it's O(N) memory, but you know what I mean).

This means, hint, you should be using a certain kind of heap (min or max, which one?), if you weren't already aware (but you should be from class anyway).

Of course, sharing docs here for convenience once again.

Without further ago, get coding!

```
/**
 * @brief Uses an early-stopping version of heapsort to
 *        select and sort the top 10% of players in-place
 *        (excluding the returned RankingResult vector)
 *
 * @param players A reference to the vector of Player objects to be ranked
 * @return A Ranking Result object whose
 * - top_ vector -> Contains the top 10% of players from the input in sorted orde
 * - cutoffs_    -> Is empty
 * - elapsed_    -> Contains the duration (ms) of the selection/sorting operation
 *
 * @post The order of the parameter vector is modified.
 */
RankingResult heapRank(std::vector<Player>& players);
```

Part B: Quicktime Events

Quick, flick the joystick left! Then A! Ok, whatever this isn't getting us anywhere.

We'll be making a quickselect/quicksort hybrid. Remember, the only difference between the two is a decision on whether to recurse on one or both sides.

So for you, continue quick-selecting until your pivot is placed in the top 10%, then begin quick-sorting until you've finished sorting the top 10%. Done! I'll leave the implementation details to you.

Once again, however, this should be done with certain memory constraints (ignoring RankingResult). Since we're using quickselect/sort we'll use O(logN) memory since recursive calls are counted when considering memory. That is to say, don't make copies of input data, feel free to modify the reference.

You'll need helper functions for this, so feel free to declare them in Leaderboard.hpp. Make sure to keep them in the `Offline` namespace.

```
/**
 * @brief Uses a mixture of quickselect/quicksort to
 *        select and sort the top 10% of players in-place
 *        (excluding the returned RankingResult vector)
 *
 * @param players A reference to the vector of Player objects to be ranked
 * @return A Ranking Result object whose
 * - top_ vector -> Contains the top 10% of players from the input in sorted orde
 * - cutoffs_    -> Is empty
 * - elapsed_    -> Contains the duration (ms) of the selection/sorting operation
 *
 * @post The order of the parameter vector is modified.
 */
RankingResult quickSelectRank(std::vector<Player>& players);
```

Task 2: Going Online

Part A: "Live" Streaming

We'll start off by implementing a `VectorPlayerStream` which extends the `PlayerStream` interface. Think `stringstream`. You give it a string to "stream" (ie. read from) and then read in the next word, one at a time. The same will apply to our `VectorPlayerStream`.

Given some vector, calling `nextPlayer()` will give us the next player in the stream until there aren't any left, and in which case, it throws an exception if you try and fetch more. We'll also have a `remaining()` function that lets us know when to stop, before we hit that exception when using the stream (if you didn't already know, handling exceptions are slow, so we avoid try/catch blocks when we can).

Here's what to implement:

```
/**
 * @brief The interface for a PlayerStream created using the contents of a vector
 *
 * @example Let's cover a brief example:
 * Given a vector of Player objects v = {
 *      Player("Rykard", 23),
 *      Player("Malenia", 99)
 *  }
 *
 * Our stream's behavior would be someting akin to:
 *
 * stream.remaining() -> 2
 * stream.nextPlayer() -> Player("Rykard", 23)
 * stream.nextPlayer() -> Player("Malenia", 99)
 * stream.remaining() -> 0
 * stream.nextPlayer() -> throws std::runtime_error()
 */
class VectorPlayerStream : public PlayerStream {
private:
    // Your private members here. You're the designer now!

public:
    /**
    * @brief Constructs a VectorPlayerStream from a vector of Players.
    *
    * Initializes the stream with a sequence of Player objects matching the
    * contents of the given vector.
    *
    * @param players The vector of Player objects to stream.
    */
    VectorPlayerStream(const std::vector<Player>& players);

    /**
    * @brief Retrieves the next Player in the stream.
    *
    * @return The next Player object in the sequence.
    * @post Updates members so a subsequent call to nextPlayer() yields the Playe
    * following that which is returned.

    * @throws std::runtime_error If there are no more players remaining in the st
    */
    Player nextPlayer() override;

    /**
    * @brief Returns the number of players remaining in the stream.
    *
    * @return The count of players left to be read.
    */
```

```
    size_t remaining() const override; // see how many instances remaining to be
};
```

Part B:

Before we move on to selecting & sorting, we'll implement a helper function: `replaceMin()`.

It's a modified `percolateDown()` routine that replaces the top (ie. overrides the minimum value) of a min-heap with a new element, then percolates it down to maintain the heap-property. To keep in line with the syntax of the STL heap methods we've been using, this uses iterator parameters to specify the beginning and end of our heap.

Things to bear in mind:

1. Iterators can be tricky, familiarize yourself with the STL, particular iterator operations.
2. The textbook's `percolateDown()` is great to use as a starting point. In fact, the bulk of this exercise is just using the iterator methods to adapt that code. However, note the indexing difference. In the textbook, the 0th (ie. first) element of the heap is used to store temporary values. In our implementation, **the first element of the heap (ie. the `start` parameter) is a valid element of the heap.** In fact, it'll be considered the root.

By the by, since it's a reference, feel free to move the inserted Player object instead of copying. Not stricly necessary, but it's just up to you as a designer.

```
/**
 * @brief A helper method that replaces the minimum element
 * in a min-heap with a target value & preserves the heap
 * by percolating the new value down to its correct position.
 *
 * Performs in O(log N) time.
 *
 * @pre The range [first, last) is a min-heap.
 *
 * @param first An iterator to a vector of Player objects
 *      denoting the beginning of a min-heap
 *      NOTE: Unlike the textbook, this is *not* an empty slot
 *      used to store temporary values. It is the root of the heap.
 *
 * @param last An iterator to a vector of Player objects
 *      denoting one past the end of a min-heap
 *      (i.e. it is not considering a valid index of the heap)
 *
 * @param target A reference to a Player object to be inserted into the heap
 * @post
 * - The vector slice denoted from [first,last) is a min-heap
 *   into which `target` has been inserted.
 * - The contents of `target` is not guaranteed to match its original state
 *   (ie. you may move it).
 */
void replaceMin(PlayerIt first, PlayerIt last, Player& target);
```

Part C:

We've finally reached the coup de' grace: our online algorithm.

Mind you, I'll use `r` *instead of* `reporting_interval`*, because it's easier to read.*

Here's the gist of it: we'll read in the contents of a PlayerStream one at a time until there's none left. With each read, we'll maintain a min-heap of size `r` containing the highest leveled players we've seen thus far.

Some questions & tips to guide you (it's not fun if I just tell you the answer).

Suppose we have `r=100`. If we haven't read in `r=100` players yet, what should the Leaderboard look like? Then, consider the case where we do have `r=100` players AND the newly read player is higher level than the lowest-level player on the leaderboard. Should that lowest-level player stay on the leaderboard? That's your hint! The rest is on you to figure out.

To take advantage of our online algorithm, with every `r` reads we do, we'll record the minimum Player level needed to be in that collection.

Theoretically, if our stream was instead hooked up to some actual **live** feed, then we could continuously update our Leaderboard as new info comes in, instead of having to recompute over the entire set of Players.

Here's the function signature:

```
/**
 * @brief Exhausts a stream of Players (ie. until there are none left) such that u
 * 1) Maintain a running collection of the <reporting_interval> highest leveled pl
 * 2) Record the Player level after reading every <reporting_interval> players
 *    representing the minimum level required to be in the leaderboard at that po
 *
 * @note You should use NOT use a priority-queue.
 *       Instead, use a vector, the STL heap operations, & `replaceMin()`
 *
 * @param stream A stream providing Player objects
 * @param reporting_interval The frequency at which to record cutoff levels
 * @return A RankingResult in which:
 * - top_       -> Contains the top <reporting_interval> Players read in the strea
 *                 sorted (least to greatest) order
 * - cutoffs_   -> Maps player count milestones to minimum level required at that
 *                 including the minimum level after ALL players have been read, i
 *                 of being a multiple of the reporting interval
 * - elapsed_   -> Contains the duration (ms) of the selection/sorting operation
 *                 excluding fetching the next player in the stream
 *
 * @post All elements of the stream are read until there are none remaining.
 *
 * @example Suppose we have:
 * 1) A stream with 132 players
 * 2) A reporting interval of 50
 *
 * Then our resulting RankingResult might contain something like:
 * top_ = { Player("RECLUSE", 994), Player("WYLDER", 1002), ..., Player("DUCHESS",
 * cutoffs_ = { 50: 239, 100: 992, 132: 994 } (see RankingResult explanation)
 * elapsed_ = 0.003 (Your runtime will vary based on hardware)
 */
```

Extra Credit: Going Online (but this time, literally)

Before you begin, you should note: As this is *extra-credit*, the TAs won't be providing assistance for it (unless there is some glaring error, in which case show *proof* why it's an error on our end & flag it to us).

That's not to like, scare you. It's not a particularly difficult task, but it is additional work for your own pedagogy. Thus, the burden lies on your shoulders!

Anywho to begin, define the `API_ENABLED` compiler directive in `PlayerStream.hpp`, to flag to us that you'll be implementing `APIPlayerStream`.

A Brief Intro to CMake & Ninja

You've been using `make`. Well, here's its more mature sibling, `CMake`. In effect, this is the thing that *makes* the `Makefile`. I know, crazy right? What's more is that this lets us install external modules for our projects. That means we're not just limited to the STL. You've probably even heard of some of them, like Boost. For our intents and purposes, we'll be using the `<nlohmann/json>` and `<cpr>` libraries (optionally included in your `PlayerStream` file)

We're not going too in-depth in it, but we *have* provided a `CMakeLists.txt`.

To run it, you'll need to install CMake and then run (in the directory with the `CMakeLists.txt`):

```
cmake .
```

You'll notice that it is *obscenely* slow. This will always happen for the first run of it, but subsequent runs will be faster.

Similarly, you'll produce a new `Makefile`. If you make changes to your code, you can use `make` like you always have been.

But even `make` is pretty slow. So, instead, you should install ninja-build, which is a build tool specifically made to be fast.

After you've done this, you can run:

```
cmake -G Ninja -S . -B build
cd build
ninja
```

Which does the following:

1. Runs CMake in the current directory and generates all the build files in a new `build` directory
2. Navigate to that new directory
3. Compiles/builds your program using Ninja

Once you make your build files with CMake, generally you should just run `ninja` if you make changes.

Calling the API

Part A: Running a Sample server

Since we're making something that queries from an API, we've provided a sample Python server for you to run.

You may notice files like `python-version` & `pyproject.toml`. These are files associated with uv, a faster Python package manager (think `pip` but better). They detail the exact versions of python & dependencies we'll need to run our server.

To minimize compabatibility errors, I'd advise you to use `uv`. Ironically, we'll install it using `pip`, or see the site for more installation details.

```
pip install uv
```

Read the docs & once you're to roll, run the following in the `server` directory:

```
uv venv                # Makes a virtual environment
.\.venv\Scripts\Activate # Note, this may vary based on OS, but just launch the v
uv sync                # Install all dependencies
python server.py       # Runs the server
```

You should get something like:

```
 * Serving Flask app 'server'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://<YOUR_LOCAL_IP_ADDRESS>:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 520-643-247
```

Part B: Fetching from the Sample Server

Now, it's time to make the `APIPlayerStream`.

NOTE: When submitting your program, you should fetch the API from Host `127.0.0.1` at port `5000`. However, for some reason (eg. sometimes with WSL), you might be fetching from `127.0.0.1` using the `cpr` library, but the request fails to go through. In that case, try using the IP address listed after running the server (it may look something like `192.XXX.X.XXX.`).

By the by, we'll be using a similar server to test your code against. So if it doens't work on your local server & machine, it probably won't work on Gradescope.

Without further ado, here's the specifications for `APIPlayerStream`.

```
/**
 * @brief A PlayerStream implementation that fetches Player objects from an API i
 *
 * Specifically, this retrieves Player objects localhost (127.0.0.1) at port 5000
 * to fetch subequent batches of Players of specified size.
 *
 * Calls fetch *batches* of players--it'd be inefficient to keep calling the API
 * when we could possibly have hundreds or thousands!
 *
 * Make calls by querying the following endpoint, while running the sample server
 * http://127.0.0.1:5000/api?seed={POSITIVE NUMBER}&cursor={CURSOR}&batch={BATCH_
 *
 * So an example might be :
 * http://127.0.0.1:5000/api?seed=1&cursor=1&batch=5
 *
 * Which might return a JSON whose body is of the form (note the numbers won't ma
 * {
 *  cursor: 6,
 *  levels: [ 10, 20, 23, 1, 389 ]
 * }
 *
 * NOTE:
 * 1) You'll have to use the cpr library to call the API
 *    via cpr::Get & cpr::Url
 * 2) And parse the JSON response using the nlohmann::json library.
 *    via json::parse & `.template get<size_t>()`
 *
 * See:
 * 1) https://docs.libcpr.org/introduction.html
 * 2) https://json.nlohmann.me/api/basic_json/get/#examples
 *
 * On you to read the docs! It's extra credit after all.
 * We've also included a sample server that will match the
 * format of the server of what you'll be calling.
 *
 * The numbers will be different though, so don't even think of hard-coding it :)
 */
class APIPlayerStream : public PlayerStream {
protected:
    const std::string PORT = "5000";
    const std::string HOSTNAME = 'http://127.0.0.1';
    const std::string SOCKET = HOSTNAME + ":" + PORT;

private:
    /**
     * @brief Imagine you're querying from a database and there's millions of resu
     * Instead, of returning all million of them, we'll periodically fetch a batc
     * and store where in that sequence of million results we're at.
```

```
     * `cursor_` is *exactly* this. Think of it like the page number on Google se
     */
    size_t cursor_;

    /**
     * @brief Since we're not working with an actual database,
     * we'll use seeds to pseudo-randomly generate contents.
     * Make sure to include this in your calls.
     */
    size_t seed_;

    // Your additional private members here
public:
    /**
     * @brief Constructs an APIPlayerStream that fetches Players from an API and
     *
     * @pre All parameters are positive (ie. > 0). Also, for simplicitly assume t
     *
     * @param expected_length The total number of Player objects expected from th
     * @param seed A seed value used for API requests to ensure consistent result
     * @param batch_size The number of Player objects to fetch in each API reques
     *
     * @post
     * a) `cursor_` is initialized to 1
     * b) `seed_` is initialized to the provided seed value.
     * c) You're the designer for the rest, create private member variables
     *    to achieve the outlined functionality.
     */
    APIPlayerStream(const size_t& expected_length, const size_t& seed, const size

    /**
    * @brief Retrieves the next Player in the stream.
    *
    * @details Performs either of the following:
    * If the currently stored batched has players that have not been read yet,
    * returns the next Player object from the current batch.
    *
    * OR
    *
    * If the current batch has been exhausted,
    *   1) Issues a new API request
    *       using 'seed_' for the seed query parameter
    *       the current `cursor_` value for the cursor query parameter
    *       the batch size specified in the constructor for the batch query param
    *   2) Records the fetched batch contents;
    *       since the task at hand is only related to ranking levels,
    *       the API will ONLY return a number representing the level of a player.
    *       When recording the contents, you can name Player objects whatever you
    *
```

```
    *   3) Returns the first Player Object in the newly retrieved batch.

    * @post If a new API call is made, `cursor_` is
    *   set to the updated cursor value returned by the API call.
    *
    * @return The next Player object in the sequence.
    * @throws std::runtime_error If there are no more players remaining or if the

    */
    Player nextPlayer() override;

    /**
     * @brief Returns the number of players remaining in the stream.
     *
     * @return The count of players left to be read.
     * @example If our stream is initialized with expected length 5,
     *   after calling nextPlayer() twice, we'll have 3 players remaining.
     */
    size_t remaining() const override;
};
```