Heap Coding Practice for Midterm 1 (CSCI 104)

You have a 5-ary Pokémon MinHeap that uses a vector container of std::pair based on 0-indexing. The std::pair has a .first of rarity (double) and a .second of name (std::string). The heap property is based on the rarity of a Pokémon. Assume that you have working implementations of trickleUp() and trickleDown() if you need it.

Here's the class you will be using (incomplete but it's enough to do the problem):

```
class Pokemon_MinHeap {
public:
    void updateRarity(std::string target_name, double new_rarity);
    void defeat();
    void multi_defeat(int x);

private:
    std::vector< std::pair<double, std::string> > pokemons;
    void trickleDown(int x);
    void trickleUp(int x);
};
```

PROBLEM 1.1:

A Pokémon was found to be more common than originally anticipated. We want to update our data structure to reflect that. You can assume that the value of new_rarity will always be greater than the Pokémon's current rarity. To do this, implement:

```
void Pokemon_MinHeap::updateRarity(std::string target_name, double new_rarity)
```

More specifically, you should:

1. Search the MinHeap for a Pokémon name that matches the target_name parameter. If a matching name cannot be found, throw `std::invalid_argument()`.

2. If a matching name is found, update the correct Pokémon's rarity and make sure you maintain the heap property (remember that rarity can only increase in this problem).

PROBLEM 1.2:

What is the runtime complexity of `Pokemon_MinHeap::updateRarity()` ? Justify your answer.

Answer:

PROBLEM 2.1:

We want to hunt down the rarest Pokémon possible.

Implement `void Pokemon_MinHeap::defeat()` to defeat the rarest Pokémon.

More specifically, you should:

1. Throw an `std::underflow_error()` if there is nothing to remove.
2. If there is something to remove, remove the rarest Pokémon (the Pokémon with the lowest rarity value) while maintaining the heap property.

PROBLEM 2.2:

What is the runtime complexity of `Pokemon_MinHeap::defeat()` ?

Answer:

PROBLEM 3.1:

Now that you have a hopefully working defeat implementation, we now want to defeat the **x** rarest Pokémon based on user inputs. To do this, implement

`void Pokemon_MinHeap::multi_defeat(int x)`.

More specifically, you should:

1. Check if there are enough Pokémon to defeat based on x and check if x is at least 1. If either check fails, throw `std::underflow_error()`.
2. If the checks are successful, defeat **x** amount of Pokémon by updating the MinHeap and maintaining the heap property. You are also allowed to use your coded implementations from previous problems (assume they work properly).

PROBLEM 3.2:

What is the runtime complexity of `Pokemon_MinHeap::multi_defeat()` ?

Answer:

PROBLEM 4.1 (unrelated to previous problems):

When is trickleUp() normally used?

Answer: