DADSA Assignment 1

A tennis player ranking system.

Takes in the score of each match for a given tournament and updates each players position, calculates each players ranking points and produces a list of the players ranking in descending order.

Solutions

This system can be designed in two different ways, depending on how data is presented to the application: static, and streamed.

Stream

Data could be streamed into the application if games are still ongoing. Players in this state may still score points, thus changing their position in the current tournament. Therefore an algorithm that can quickly update individual positions in the tournament should be favoured.

A well-programmed sort algorithm will have an average sort time complexity of `O(n log n)`. However self balancing trees tend to have modification time complexities of `O(log n)`. Since we are optimising for sorting on individual element modifications, it would be very expensive to reorganize and verify the entire data structure. A self balancing tree would then be the most optimal route to take when streaming data.

We must take into account that third parties, whom are feeding us the data, may not understand nor wish to feed us our direct object references. Instead we will probably be given either player names or other types of identification. These can be used as keys which may then be used for indexing our own player profiles. Hash tables provide a speedy average lookup time of `O(1)` for this operation, so these will be used as a form of compatibility when interfacing with the data provider.

Red-black order statistic tree

- Space complexity: `O(n)`
- Best case access, search, insertion and deletion: `O(log n)`
- Average case access, search, insertion and deletion: `O(log n)`
- Worst case access and insertion: `O(log n)`
- Worst case search and deletion: `O(n)`
- The red-black self balancing method was chosen as it is guaranteed `O(log n)` lookup and modification time within the tree itself (forgetting node operations).

- Multiple players may have the same score, so each node in the tree must allow for storing multiple players. In this scenario, I have chosen to simply supply a linked list for holding the node values. This is why the search and deletion operations are worst case `O(n)`, unlike normal red-black trees.
- We may be required to find all players at a specified rank, or find the rank of a specified player. Each node is given a size, which can be used for performing these `O(log n)` operations.

Hash table

- Space complexity: `O(n)`
- Best case search, insertion and deletion: `O(1)`
- Average case search, insertion and deletion: `O(1)`
- Worst case search, insertion and deletion: `O(n)`

Static

Static (immutable) data may be provided to the application if all games are finished. We can still use the solution provided with streamed data. Though there are some downfalls that can be avoided since we are certain the data cannot change.

Some operations in our previous solution have a worst case time complexity of `O(n)`, therefore processing the entire data set will give us an overall worst case time complexity of `O(n^2)`. As stated before, there are sorting algorithms that provide a worst case scenario of `O(n log n)`. Sorting is also able to put the data into an array data structure. Arrays have a guaranteed lookup time of `O(1)`, which is more optimal than the trees `O(log n)`. Hence a sorting algorithm would be better than a tree for static data.

This then leaves the question "What sorting algorithm should be used?". There tons of different sorting algorithms, all with their own sets of advantages and disadvantages. Selecting an optimal sorting algorithm is generally down to what is to be expected of the input data. As this assignment is vague on the specifics on what to expect, it's probably best to use a general purpose sorting algorithm that has the best all-round statistics.

Tim sort was the first general purpose sorting algorithm that popped into my head. It is the main sorting algorithm used by Python and Java as it offers excellent benchmarks on sorting real world data. Tim sort achieves this by taking advantage of already sorted sections or "runs" of the provided data set. Though Tim sort is not the only algorithm that does this.

Block sort is a newer sorting algorithm that, like Tim sort, is a combination of merge sort and insertion sort and uses runs. Block sort is almost identical to Tim sort when ran on real world data, but it provides one extra advantage over

Tim sort, a space complexity of `O(1)`. For comparison, Tim sort has a space complexity of `O(n)`.

My original design was to use block sort. Though I've found while gradually loading player data from a file, instead of putting it into an array, it could be put into a better data structure that helps with the sorting process. Pipe sort is a sorting algorithm I designed which was inspired by Tim sort. It's similar to Tim sort in the way that it is able to detect runs, both forwards and backwards. Even if the entire data set supplied was in reverse, pipe sort should be able to sort it in `O(n)` time.

Pipe sort differs from Tim sort in two ways. When feeding data into pipe sort, it not only finds runs in the data but it also groups these runs by their relative sizes using the specialised tree data structure specified in the stream solution. Note that this tree is indexed by the size of the runs, not the data itself. The other difference is, insertion sort is now no longer necessary as it is more optimal to recursively merge sort all similar sized runs. We know all the data is sorted once there is only a single run remaining.

Pipe sort

- Space complexity: `O(n)`
- Time complexities:
- Best: `O(n)`
- Average: `O(n log n)`
- Worst: `O(n log n)`

User interface

The static solutions method of taking in data is via the default files as provided for the assignment. This makes sense as we do not expect this data to change at all during the player ranking and reward finding process. One change was made to the provided files, since I dislike having hardcoded values in my programs. I added the difficulty level of each tournament when they're defined in the `tournaments.csv` file as an extra column. Other than that, they are as they were provided.

Whereas in the solution for streamed data, I decided to take an entirely different route as we are expecting the data to come in at different times and want to update the leader boards live. I introduced a method of receiving user input via command line, using a state-based model to easily parse all the required information on tournaments. This means when each game is running, it's possible to update the player scores and each of the leader boards become automatically updated and sorted.

Both solutions offer the ability to pause and save all circuit data for another day by pressing `CTRL+C` at any point during the score creation process. To continue from the previous state, simply re-run the program. The streamed implementation offers the ability to discard and overwrite the previous circuit data upon re-run. The same affect can be achieved simply through manually deleting the programs relevant `output` sub-directory.

How to run

Requires Python 3. This project was tested using Python 3.6.3.

First, ensure you are in the source directory:

```
cd src
```

To run the static solution:

```
python solution_static.py
```

To run the stream solution:

```
python solution_stream.py
```

Other notes

- The library numpy was used in this project, not for ease of use but to emulate a proper C-style array of immutable size.
- I am aware that focusing on algorithms like this being optimal is futile in a language like Python, and that creating these in a lower level language will probably provide a performance gain. This assignment specified for them to be written in this language. If the intention was to get us to create native bindings, then I'd be happy to oblige.
- There are already many great libraries out there for Python that have similar behaviour to what I have written, and reinventing the wheel is usually something that should be avoided in software development. I made the exception here as I assume the assignment was designed to show an understanding of common data structures and algorithms.