Explanations about the first assessment

I used Microsoft Visual Studio to implement my solution. I have also a **Demo.gif** in the base root of the github repo in order for you to see also the results I am getting in my machine. The code of the assessments was tested in both Debug-x64 and Release-x64 builds.

Exercise one:

- I downloaded the requested txt file and wrote a parser for that, picking random words from the .txt to serve as keys to the hash table (random generated dataset was not required by the assessment but I just implemented it to be fair on the gathering of words for the hash table).
- The hash table is actually a vector of elements (we of course do not access it sequentially due to the increased complexity of this task). To achieve lower complexity O(1) we are using the std::hash to get hashes for the keys.
- The insertion happens with these hashes. We use modular arithmetic of the hash (hash % hash_table_capacity) in order to limit the indices to the capacity of our table and not create segfaults and avoid out of bounds indexing.
- Using Linear Probing we solve the conflicts, as while the calculated index is already occupied we move onto the next one and when we find a free memory space we actually save the values there.
- I also have a check for deleted values as in case we insert, then delete and then try to find an element we could potentially give wrong results as the linear probing can point you to a previously deleted index and the actual value could be in another index due to a previous conflict while inserting.
- For the last and first inserted elements I have a vector that tracks all of the insertions in the hash table (this could be also useful in future scenarios for tracking the activity) and I simply pop the front or the back in case we want to acquire the first or last element.
- Inside the main.cpp you can see all the code for the first exercise in the runFirstExercise function.
- I had the following function calls that everything is working correctly:
    - Insert non listed elements
    - Insert an already existing element so we can check that we update the value
    - Remove an element
    - Get an element from the hash table
    - Get first and last inserted elements.
    - I also checked that the return codes are working when we try to insert a new key and the hash table is already full.

Exercise two:

- In order to get the information from your endpoints I am using a cpp library that is wrapping the curl c library. Specifically you can find the repo of this library here: https://github.com/JosephP91/curlcpp
- I linked this library to my project and started working on that.
- After reading your documentation I used this url in order to get the available data from your endpoints https://fapi.binance.com/fapi/v1/aggTrades?symbol=BTCUSDT
- I choose BTCUSDT totally randomly as I visited your site (https://testnet.binancefuture.com/) that was pointed in your documentation.
- After I ensured that the curl request was working and indeed I was getting the data I started working on the parser.
- Here I was a bit confused whether I should use a third-party library or write my own parser. As there were some questions regarding the complexity of the parser and I didn't want to base the implementation of my code on third-party libraries, so I wrote my own parser. Note that I do not like re-inventing the wheel and in production code maybe is better to use ready solutions optimized for performance (always based on the requirements of the product though, maybe there is not an appropriate implementation already in other libraries based on your requirements).
- The parser is really simple and uses std::substring to isolate the entries while reading the stream from { } to first isolate each Trade.
- Then I used std::sscanf to directly insert the data from the json buffer to my structures. There were some problems here so I will give you some information of how I solved them:
    - Initially the Aggregate tradeId, First tradeId and Last tradeId were not parsed correctly. I then noticed that as those ids were huge I specifically needed to use a long long type to store them correctly, as previously I was using a smaller data type.
    - The caveat that I had in mind for std::sscanf was that it does not directly support bool type for the m field contained. So I switched to a string approach for this specific element. The problem here is that "true" and "false" have different lengths and I needed to tell std::sscanf how many characters to read. I selected 5 as it is the superset of the lengths for the "true" and "false" strings. This was creating a problem when reading the true value as it was also capturing the next character which was the } as m is always in the end of the Trades in the json buffer. For this case I simply remove the extra character from the acquired value in the case of "true".

      Note that this can also work if the m key was not placed last. If we had placed it in an intermediate position and not last I would simply get a buffer like "true," instead of "true}" and I could handle it exactly as I am currently handling the extra character on the buffer.
- The complexity of this parser (and in general I believe of any parser) is going to be O(N) as we actually need to traverse our entire json buffers to get the values.
- I measured the performance of my parser (of course in Release build) using the std::chrono::high_resolution_clock from the c++ std:

- for the parsing of the total json buffer containing 500 trades the time needed was 1330μs or in the millisecond unit 1,33ms.
- for the parsing of each individual trade the time was 2-7 microseconds.
- Note that for the monitoring of the total json buffer parsing we need to remove the logging of each individual trade as the performance is slower if we log on std::cout channel each one of the 500 trades. So I monitored those two cases separately.
- Also I am including here my hardware specs for my experiments as they are related to the performance monitoring. I am including only the CPU as the GPU was not used in this experiment at all.
  CPU: 11th Gen Intel(R) Core(TM) i9-11900KF @ 3.50GHz (16 CPUs), ~3.5GHz
- Inside the main.cpp you can see all the code for the first exercise in the runSecondExercise function.

Thanks a lot for your time and if you find that any explanation is missing please feel free to contact me!

Warm regards,
Leonidas Saroglou