CMP2003 Data Structures and Algorithms (C++) Term Project:

Log Analyzer

Project Overview:

We wrote a C++ console application that reads a text file "access_log" , a log file loaded with entries, and then prints out “top 10” most visited web pages. Our key element was implementing our own hash table as our main data structure and also use std::unordered_map to store the filenames and a count of how many times this page (filename) has been visited. Furthermore, to compare how efficient our hash table and std::unordered_map, measurements were made of the time it takes to read through “access_log ” followed by printing out top 10 pages most visited.

Group Members:

- Abdullah Hani Abdellatif Al-shobaki

Screenshot for the Elapsed Time Using our Custom Hash Table:

Screenshot for the Elapsed Time Using Std::Unordered_Map:

Screenshot for The Project Output:

```
/Users/diyasinnokrot/CLionProjects/Data-Structure/cmake-build-debug/Data_Structure
Top 10 most visited web pages:
index.html 139247 of total visits
3.gif 24001 of total visits
2.gif 23590 of total visits
4.gif 8014 of total visits
244.gif 5147 of total visits
5.html 5010 of total visits
4097.gif 4874 of total visits
8870.jpg 4492 of total visits
6733.gif 4278 of total visits
8472.gif 3843 of total visits

Total Elapsed Time for Custom Hash Table: 2.90277 seconds
Top 10 most visited web pages using unordered_map:
index.html 139247 of total visits
3.gif 24001 of total visits
2.gif 23590 of total visits
4.gif 8014 of total visits
244.gif 5147 of total visits
5.html 5010 of total visits
4097.gif 4874 of total visits
8870.jpg 4492 of total visits
6733.gif 4278 of total visits
8472.gif 3843 of total visits

Total Elapsed Time for Unordered Map: 2.00779 seconds

Process finished with exit code 0
```

Header Includes:

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <list>
#include <unordered_map>
#include <numeric>

using namespace std;
```

Initially, we just added all the C++ standard libraries necessary for input and output, file handling, algorithm functions and data structures.

Hash Table Implementation:

Data Structures:

```
class CustomHashTable {
private:
    struct Entry {
        string filename;
        int visits;
    };

    vector < list < Entry >> table;
    size_t size;
```

We used a vector of linked lists (list<Entry>) called table to implement the hash table where each list represents a bucket in the hash table and each element contains a filename and the number of visits. Additionally, size refers to the size of the hash table.

Hash Function:

```
    size_t hashFunction(const std::string& key) const {


        const size_t prime = 31;
        size_t hash = 0;

        for (char c : key) {
            hash = (hash * prime) + static_cast<size_t>(c);
        }

        return hash % size; // 'size' is the table size
    }
```

In the hash function part, we used a modulo-based hash function with a prime number 31 for a better distribution multiplier. We implemented it in which it takes a key (filename) and returns a hash value using a polynomial rolling hash with a prime number as the base.

Collision Resolution Method:

```
public:
    CustomHashTable(size_t tableSize) : size(tableSize), table( n: tableSize) {}

    void insert(const string& filename) {
        size_t index = hashFunction( key: filename) % size;
        auto& bucket : list<Entry> & = table[index];

        auto it : iterator<Entry, void *> = find_if( first: bucket.begin(), last: bucket.end(),
                            pred: [&](const Entry& entry) -> bool { return entry.filename == filename; });

        if (it != bucket.end()) {

            it->visits++;
        } else {

            bucket.push_back( x: {filename, .visits: 1});
        }
    }
```

Separate chaining is used for collision resolution. Our hash table consists of linked lists, and if there is a collision the entry is appended to whichever bucket it falls into.

Top 10 Most Visited Pages Method:

Data Structures and Algorithms:

```
void printTopVisitedPages(int topN) const {
    vector < pair < string, int >> allPages;

    for (const auto& bucket : const list<Entry> & : table)
        for (const auto& entry : const Entry & : bucket)
            allPages.emplace_back( t1: entry.filename, t2: entry.visits);

    partial_sort( first: allPages.begin(), middle: allPages.begin() + min(topN, static_cast<int>(allPages.size())), last: allPages.end(),
                  comp: [](const auto& a : const pair<string, int> & , const auto& b : const pair<string, int> & ) -> bool { return a.second > b.second; });

    cout << "Top " << topN << " most visited web pages:" << endl;

    for (int i = 0; i < topN && i < static_cast<int>(allPages.size()); ++i)
        cout << allPages[i].first << " " << allPages[i].second << " of total visits " << endl;
}

};
```

The PrintTopVisitedPages method collects all the entries in the hash table and puts them into a list of pairs (filename, visits). It then uses PartialSort to do a partial descending order sort of the vector by number of visits. Last, it prints the top N entries.

Conclusion:

In summary, the implementations of our custom hash table and std:unordered_map can track effectively web page visits. Both implementations are similar in performance, and the custom hash table has excellent collision handling.

Teacher initials:

Thanks to Cigdem Eris for valuable guidance. We welcome feedback for further improvement and exploration.