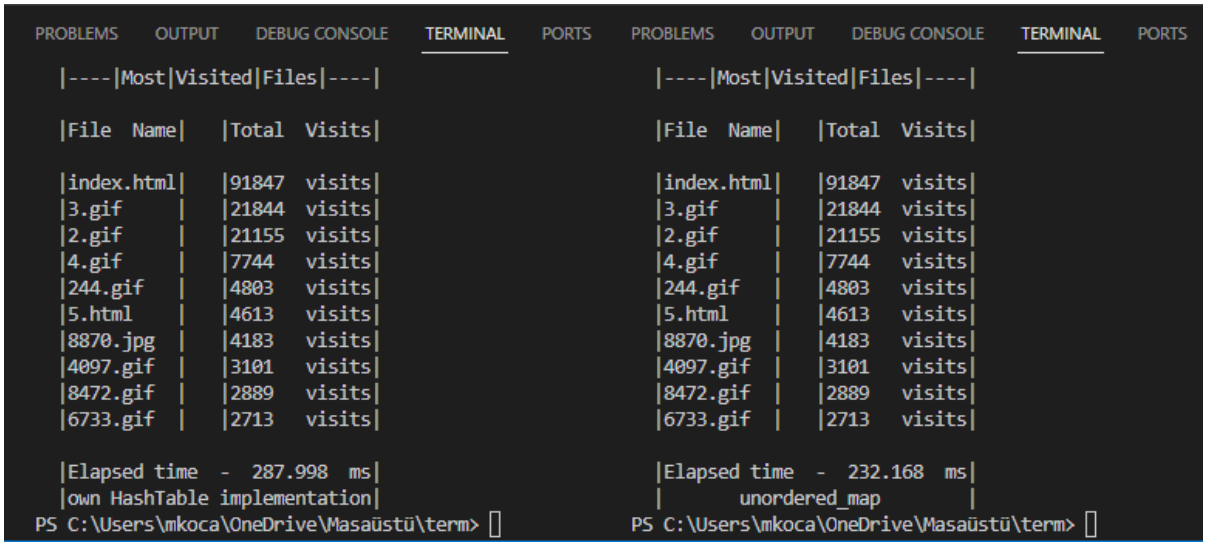
CMP2003 Data Structures and Algorithms (C++) Term Project Log Analyzer

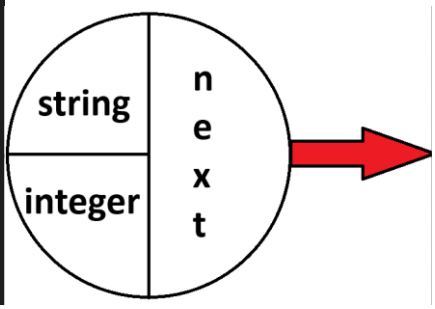


OWN HASH TABLE UNORDERED_MAP

Introduction: We have created a Hash Table class utilizing separate chaining as its collision resolution method. We used Max Heap data structure to sort the hash table in descending order. We took your hint as a guide to our method of finding the top ten most visited pages and utilized a heap data structure and we think that it is very efficient. For our hash table and heap classes we have not used vectors. Instead we used array structure with designated size.

Declaring Node Structure

```
6   struct Node {
7    std::string key;
8    int visitCount;
9    Node* next;
10
11       Node(const std::string& fname, int vcount)
12       {
13        key = fname;
14        visitCount = vcount;
15        next = nullptr;
16       }
17  };
```



Node structure is a must for us to be able to implement both the Hash Table and Heap classes. We started with declaring two variables to store information and the pointer for our node. We then declared the constructor for our node structure with a reference to the string type and an integer value as parameters and assigned those parameters to the corresponding members. Figure shows our node structure. (Red arrow resembles pointer.)

Hash Table Class

Our hash table consists of the following items:

a) Hash table size declaration

b) Hash function

c) Hash table constructor

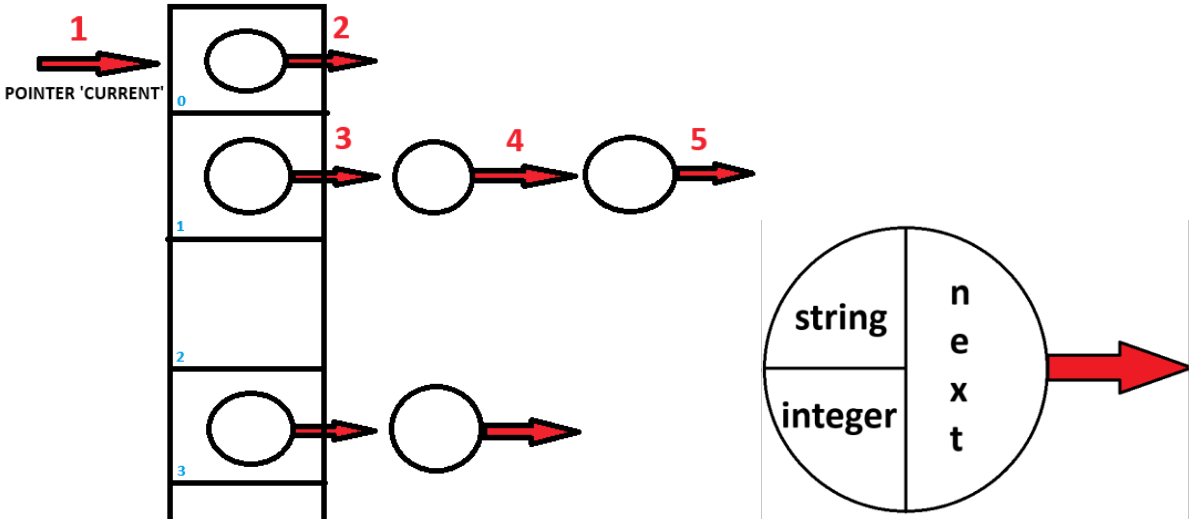
d) Array of pointers to point to our nodes

e) Function to insert file visit requests

f) Function that returns hash table size

g) Hash table destructor

We wanted to visualize our implementation of hash table class. Below figure shows an imaginary implementation of our table.



In the above figure red arrows are pointers. Blue numbers on the bottom left corner of the array boxes are index numbers of the pointer array. Our hash table uses separate chaining for collision handling method. This method utilizes nodes to create linked lists on each array index. When there is a collision the head of the index becomes the new node inserted. Number 1 is the ‘current’ pointer representation we commonly use in our loops.

We can not really show this in the figure but this array is not holding the first node it is holding the pointer that points to the adress of the head of a linked list.

Hash Function

```
19  class HashTable {
20
21  private:
22  static const int hashTableSize = 1100;
23
24      int hashValue(const std::string& key)
25      {
26       int keySum = 0;
27       for (int i = 0; i < key.length(); i++)
28          {
29           int asciiValue = key[i];
30           keySum = keySum + asciiValue ;
31          }
32       return (keySum % hashTableSize);
33      }
```

We start by declaring our HashTable class. Inside the private part, we declared the size of our hash table. (Actually, this is going to be the size of the array of pointers, all of which pointers to a head of a linked list of nodes containing file information.)

Following the hash table size declaration, we have declared our hash function, which takes a reference to a string. (This is going to be the file names that we extract from the "access_log"). We declared an integer value to hold the ASCII summation of characters that form the file name, which is fed to the function. Then the for loop iterates over the number of characters that string holds. Inside the for loop, we turn the string into an array and ask for the corresponding character of index=i of that iteration to be assigned to the integer type variable 'asciiValue'. For each iteration, we make sure we are taking the sum of each character. Finally, in return, we are taking the modulus of the total ASCII value of that string as the hash table size.

Hash Table Constructor & Array of pointers

```
35  public:
36
37      HashTable()
38      {
39          for (int i = 0; i < hashTableSize; i++)
40          {
41           pointerArray[i] = nullptr;
42          }
43      }
44
45      Node* pointerArray[hashTableSize];
```

Inside the public part, we start with declaring the constructor for our class. We iterate over the size of our hash table and set all pointers of pointerArray to null. Creating an array of pointers called pointerArray that points to Node objects.

Function to Insert File Visit Requests

```
47      void insertVisit(const std::string& key)
48      {
49       int i = hashValue(key);
50       Node* current = pointerArray[i];
51
52          while (current != nullptr)
53          {
54              if (current->key == key)
55              {
56               current->visitCount++;
57               return;
58              }
59              else
60              {
61               current = current->next;
62              }
63          }
64
65          if (current == nullptr)
66          {
67           Node* newNode = new Node(key, 1);
68           newNode->next = pointerArray[i];
69           pointerArray[i] = newNode;
70          }
71      }
```

We need a function to insert new file names and update visit counts for existing ones. To hold new file names, we need to create new nodes taking that file name as their string value and attaching a visit count of one as they are recently created. Using new operation to create Nodes takes dynamically allocated memory. This memory then needs to be deallocated within the destructor; otherwise, it may cause a memory leak.

“insertVisit” function takes reference to a key that would be found in a node. We use the hash function .hashValue() to determine the index. Then we create a node pointer and point to the pointer on the corresponding index of "pointerArray". (This part is shown in the drawing above marked as number 1.)

While loop iterates if the index is populated by a node. If the file name fed to the function is the same as the file name stored in the string variable of the node, which is pointed by the pointer ‘current’ visitCount is incremented by one, and the return statement terminates .insertVisit() function. If the file name is not the same, else iterates, and the pointer ‘current’ is set to the pointed Nodes ‘next’ pointer(number 2 or 3). If there exists a

linked list at the index, this line goes through all the nodes, (number 3,4,5) starting from the head. If at some point, pointed nodes file name string is equal to the file name fed to the function, the visit count gets incremented. Otherwise, if we reach the end of the list without finding the Node with the same file name, while loop ends its iteration as the last node on the list has its pointer 'next' set to nullptr.

Hash Table Destructor and Size Function

```
73      const int htsize()
74      {
75       return hashTableSize;
76      }
77
78      ~HashTable()
79      {
80          for (int i = 0; i < hashTableSize; i++)
81          {
82           Node* current = pointerArray[i];
83              while (current != nullptr)
84              {
85               Node* temp = current;
86               current = current->next;
87               delete temp;
88              }
89           pointerArray[i] = nullptr;
90          }
91      }
92  };
```

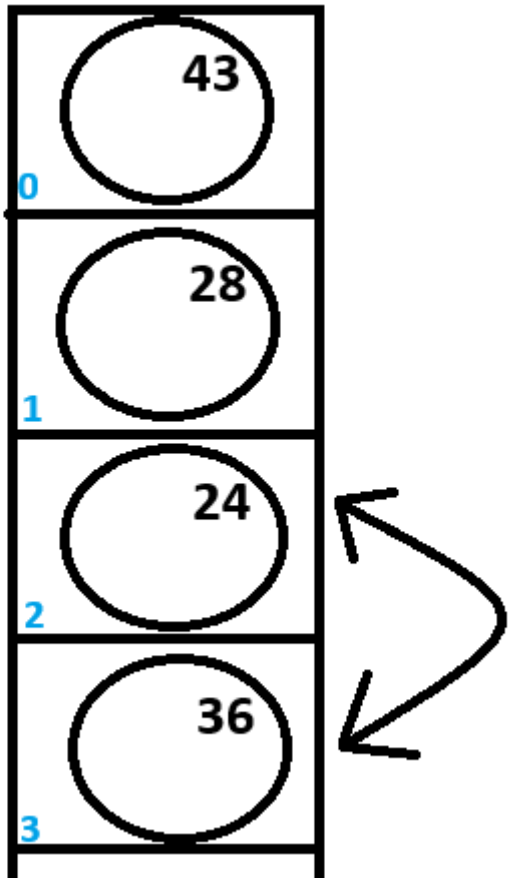
As we have already stated, we are using new to create new Nodes. This method involves dynamically allocating memory. It means that deallocating that memory is our responsibility to prevent memory leaks.So instead of relying on the default destructor, we have implemented a destructor to get rid of the Nodes and deallocate memory once we are done. We iterate the size of the hash table, deleting Nodes. This is done by pointing to a Node, with a pointer named 'current' and declaring another one in the while loop which is temporary, pointer 'temp'. Essentially, at the creation of temp, temp and current are pointing to the same node. Then we make current point to the node's next pointer, meaning current now points to the next node in the linked list on the pointer array index. This is done until the pointer array index is clear of nodes. (deleting the whole linked list at that array index)

.htsize() is present, so we can access the hash table size from inside the main function.

Heap Data Structure Implementation

Our heap implementation is a max heap meaning index 0 will posses the greatest visit count value. Max heap will be sorted in a descending order. On our heap we did not implement left and right child for each parent which means it's kind of a linear tree. On our implementation we only have a single child for each parent Node. We did this because for this task this implementation is perfectly fine and simple. In our heap class we defined a integer named size, this integer increments by one every time a new node is inserted. We are inserting pointers to Nodes in array indexes starting from index 0. When we insert a pointer to a node we are also comparing it to its parent node in terms of visitCount integer value they hold. If the newly inserted child has a greater visitCount it's position is swapped with its parent by a swap function we declared. This operation iterates until the parent of that said Node is finally greater than it. We can then use this heap to extract top ten most visited pages as we can be sure that the index 0 to index 9 will hold the ten greatest visit counts.

Let us show our heap implementation in a figure and explain from there.



As you can see in the figure newly inserted node at index 3 is getting swapped with its parent. It will then be compared to its new parent which is at index 1 and swapped. Lastly it will be compared to the node at index 0 but since it doesn't hold a greater value it will stay at index 1 and the program will move on to insert the new node at index 4. Program will compare the newly inserted node with its parent and so on.

Heap Class Declaration and Member Functioon Swap

```
94   class MaxHeap {
95   private:
96
97       static const int heapSize = 12000;
98       Node* heap[heapSize];
99      int size = 0;
100
101     void swap(Node*& child, Node*& parent)
102     {
103      Node* temp = child;
104      child = parent;
105      parent = temp;
106     }
```

We declare the class MaxHeap. Heapsize is declared as a constant integer along with integer size. While they seem like they are both the same thing they are not. Heapsize is a constant and already valued at 12000. This is the size of the array of pointers which will be pointing to our Nodes that we will be inserting from the Hash table. (Array of nodes is also declared here in the private part.) On the other hand integer size is the amount of Nodes we have at a given time. This integer size is getting incremented by one everytime a new node is inserted from the hash table.

We need the swap function to be able to sort the heap. Once the program decides that the child Node have a greater visitCount value swap operation is called to change indexes of these said Nodes.

.swap() function takes two references to two Node pointers from the pointer array. Then a node pointer named ‘temp’ is declared and initially this pointer is set to be the same pointer that points to the child. Then we set the pointer which belongs to the parameter child to point to the parent node. Then the parameter parent is equalized to temp which is the array pointer at the index where the child was. *Although parameters are named child and parent this has nothing to do with the real parent and child. The swap function would take any given two Node pointers and swap their indexes. And although parameters are called child and parent they are actually refences to the pointers from the original array of pointers which is heap[] in this class.*

Function HeapifyUP

```
108     void heapifyUp(int index)
109     {
110         if (index == 0)
111         {
112          return;
113         }
114
115         int parent = (index - 1);
116         if (heap[parent]->visitCount < heap[index]->visitCount)
117         {
118          swap(heap[parent], heap[index]);
119          heapifyUp(parent);
120         }
121     }
```

This function is responsible for sorting nodes in the heap. This function will be used in another function in this class. This function takes an index and it declares index-1 as the parent index. It then access the index of the parent Node and child Node to compare their visitCount member variables. If the child's visitCount variable is greater than the parents this function swaps indexes and it calls itself again. This call creates a recursive function which returns at either index 0 or the if statement fails.

Heap Insert Function and Construtctor for Heap Class

```
123 public:
124     MaxHeap()
125     {
126         for (int i = 0; i < heapSize; i++)
127         {
128          heap[i] = nullptr;
129         }
130     }
131
132     void heapInsert(Node* node)
133     {
134         if (size == heapSize)
135         {
136          std::cerr << "Heap is full, insertion failed." << std::endl;
137          return;
138         }
139
140      heap[size] = node;
141      heapifyUp(size++);
142     }
```

Constructor for our MaxHeap class was defined it iterates for the heapSize and sets all pointers of pointer array to null.

Heap insert function takes in a node pointer. For our case this is going to be the pointers from our hash table. If the size is equal to the size of the pointer array an error message is displayed. Otherwise node pointer is inserted at the array index of size at that iteration. After inserting the node pointer in the pointer array heapifyUp is used to determine if the node needs to be on a lower index than it currently is depending on its visit count. For the first iteration size is 0 as there are no nodes present. After the iteration size is incremented by one and the function continues.

Heap Retrieve Function

```
145     Node* heapRetrieve(int x)
146     {
147         if (size == 0)
148         {
149          std::cerr << "Heap is empty, nothing to retrieve" << std::endl;
150          return nullptr;
151         }
152
153      Node* retrievedNode = heap[x];
154      return retrievedNode;
155     }
```

This function is needed to access a spesific index in the pointerArray. It takes is in integer value and uses it as the index and returns the pointer in that index. Upon reaching the pointer we can then reach the node it points to and then the elements inside that Node. If our size integer is equal to zero, error message gets displayed. Otherwise node pointer retrievedNode is initialized the adress of pointer at given pointer array index. Since we know that we need to get the top 10 visits we know that we need to access index 0 to 9.

Main Function

```
183   int main() {
184    Timer timer;
185    timer.markBegin();
186
187    HashTable fileVisits;
188    std::ifstream logFile("access_log");
189
190       if (logFile.is_open())
191       {
192           std::string line;
193           while (std::getline(logFile, line))
194           {
195            int GETPos = line.find("GET");
196            int HTTPPos = line.find("HTTP");
197            int successCode = line.find("\" 200");
198               if (GETPos != std::string::npos && HTTPPos != std::string::npos && successCode != std::string::npos)
199               {
200                std::string fileName = line.substr(GETPos+4, HTTPPos-(GETPos+5));
201                fileVisits.insertVisit(fileName);
202               }
203           }
204        logFile.close();
205       }
```

Timer is explained after the main function.

Our main function starts with declaring a object of class HashTable named fileVisits. Then using input file stream we access the “access_log” file.Then we give the if statement the boolen value of the operation ‘logfile.isopen()’ this operation returns true if the file exists and can be opened. Inside the if statement we declare a string type variable named line.This line is used to store each line of the log file on each incrementation.Inside the while loop getline function takes in the file name and returns those lines and equals them as our already declared line variable.Then we declare an integer type variable named GETPos this will store the position of ‘GET’ inside the current line that is being processed. .find() function searches for “GET” and returns its position.Similarly this is done for HTTPPos and successCode integers.

Inside the while loop an if statement checks if the three integers we declared as position holders are holding a position. If one of them could not be found in the line their value is equaled to npos by the .find() function. We check if any of them are npos by a simple

and operation. Inside the if statement we use the positions we have found to extract the file name in that line. The .substr() function takes the starting position and the length then returns what ever it finds on that position interval. We add +4 while determining our starting position because don't want to include GET and a space between the file name and GET. Similarly we subtract 5 from the length because we don't want to include HTTP and a space.

We then use .insertVisit function we declared in the Hash Table class to insert the file name in the hash table named fileVisits.

Once all the lines are read .getline() function returns boolean value false thus if statement ends iteration and by line 204 .close() function is called upon the logFile.

Inserting Into Max Heap From the Hash Table

```
207    MaxHeap maxHeap;
208
209       for (int i = 0; i < fileVisits.htsize(); i++)
210       {
211        Node* current = fileVisits.pointerArray[i];
212           while (current != nullptr)
213           {
214            maxHeap.heapInsert(current);
215            current = current->next;
216           }
217       }
```

We declare a object of class MaxHeap called maxHeap. We iterate for the size of hash table size. The program first sets a pointer named current to adress of the hash table index i.

While loop checks if current is null this is a check if we have reached the end of the linked list on that pointer array index.

As long as there are Nodes left on the linked list, heapInsert() function inserts the Node pointers into the pointer array in the max heap. Then pointer current is adressed to the next pointer of the Node it points to which translates to, program traverses to the next Node on the list until the end of the linked list is reached and nullptr ends the while loop.

When there are no Nodes left in the linked list for loop iterates for the next index of the pointerArray of the HashTable.

Output

```
219   std::cout << "  |----|Most|Visited|Files|----|" << std::endl << std::endl;
220   std::cout << "  |File  Name|   |Total  Visits|"<< std::endl<< std::endl;
221      for (int i = 0; i < 10; i++)
222      {
223       Node* fileInfo = maxHeap.heapRetrieve(i);
224       std::string fileNameOut = fileInfo->key;
225       int visitCountOut = fileInfo->visitCount;
226       std::string numOfDigits = std::to_string(visitCountOut);
227       std::cout << "  |" << fileNameOut << std::string(10 - fileNameOut.length(), ' ') <<"|   |"<<
228      }
229
230   timer.markEnd();
231   std::cout << std::endl << "  |Elapsed time  -  " << timer.elapsedTime() << "  ms|" << std::endl;
232   std::cout << "  |own HashTable implementation|";
233   return 0;
234   }
```

Line 227 is cut because it didn't fit the screenshot but the cut part is not important, just aesthetics for the table like output.

For loop iterates exactly 10 times with index values 'i' to be used for heapRetrieve function. This function returns pointers for the given index. Node pointer fileInfo is adressed to the Node pointer on the index of max heap. Then for aesthetic purposes, we take the variables stored in the Node and declare them as string and integer values to be printed. We also use to_string() function to get the visit count integer into a string type so we can perform .lenght() function on it. Again these are only for determining how many spaces will be printed so we can have a clean and neat looking output.

Timer Class And Implementation

```
161   class Timer {
162   private:
163    std::chrono::steady_clock::time_point timerBegin;
164    std::chrono::steady_clock::time_point timerEnd;
165   public:
166
167       void markBegin()
168       {
169        timerBegin = std::chrono::steady_clock::now();
170       }
171       void markEnd()
172       {
173        timerEnd = std::chrono::steady_clock::now();
174       }
175       double elapsedTime()
176       {
177        std::chrono::duration<double, std::milli> elapsedTime = timerEnd - timerBegin;
178        return elapsedTime.count();
179       }
180   };
```

This is the class declaration for the Timer class. We have used chrono library to keep track of time and find the duration of which this code takes to compile. We start by declaring time_points in 'steady_clock' clock type from the chrono library. We declare two

time_points named timerBegin and timerEnd to be used in the class. We are using steady_clock because we did our research and we come up with that it will be the best type of clock for duration calculations of compile time.

We declare two functions that both marks the current time they are called as time_points. We declare a third function to calculate and return the duration as known as elapsed time. Duration is in namespace chrono and is used to explicitly calculate duration. We are stating milli in its definition to get the duration in milliseconds. The duration type variable elapsedTime can be turned into an double value using .count() function on the duration.

```
183   int main() {
184     Timer timer;
185     timer.markBegin();
```

```
230     timer.markEnd();
231     std::cout << std::endl << "  |Elapsed time  -  " << timer.elapsedTime() << "  ms|" << std::endl;
232     std::cout << "  |own HashTable implementation|";
233     return 0;
234   }
```

We create an object of class Timer named timer. We call on the .markBegin() to mark the beginning time of the main function. We then do the operations we want to do and at the end of the main function we mark the end time point with .markEnd() function. We then use .elapsedTime() function to get the duration in a double type and in milliseconds. And print the elapsed time.

Unordered Map Implementation

```
127                 fileVisits[fileName]++;
128               }
129           }
130       logFile.close();
131       }
132
133
134   MaxHeap maxHeap;
135
136       for (std::pair<const std::string, int>& pair : fileVisits)
137       {
138         std::string fileName = pair.first;
139         int visitCount = pair.second;
140
141         Node* newNode = new Node(fileName, visitCount);
142         maxHeap.heapInsert(newNode);
143       }
```

Unordered map replaces our HashTable class. It does the same job as our hash table but it is a little more efficient. The only difference in the code is that the hash table class is now not utilized and the part that is shown on the screenshot is a little different. As you can see now we need to insert into the unordered_map using its own template. Unordered map takes a string and if it exists it increments its corresponding integer value by one and if it doesn't exist it creates it on the map and gives it the value one.

A pair of a string and an integer type is declared and the range based for loop iterates over unordered_map container. String type variable fileName initialized the first variable of the pair and integer type variable visitCount is initialized the second variable.

A node pointer new node is adressed to a new Node created by the operation new. This new node takes in the fileName and visitCount as its variables. Afterwards .heapInsert() function is called to insert this Node into the heap.

As you can notice for our hash table implementation we have created nodes with our hash table class but while using unordered_map we created the Nodes while insertion into the heap. This is because our heap class was declared on taking and dealing with Nodes.

Libraries Used

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   #include <chrono>
```

Mehmet Kocabağ 2200588

Ali Emre Yaman 2102410