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

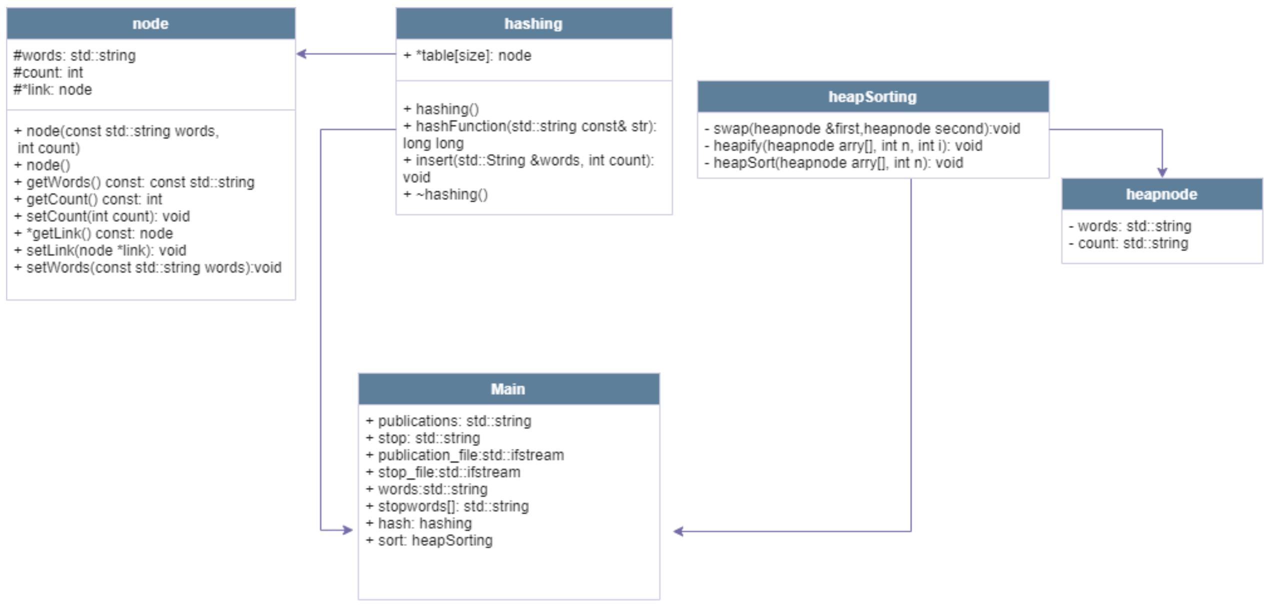
Top 10 Frequent Words

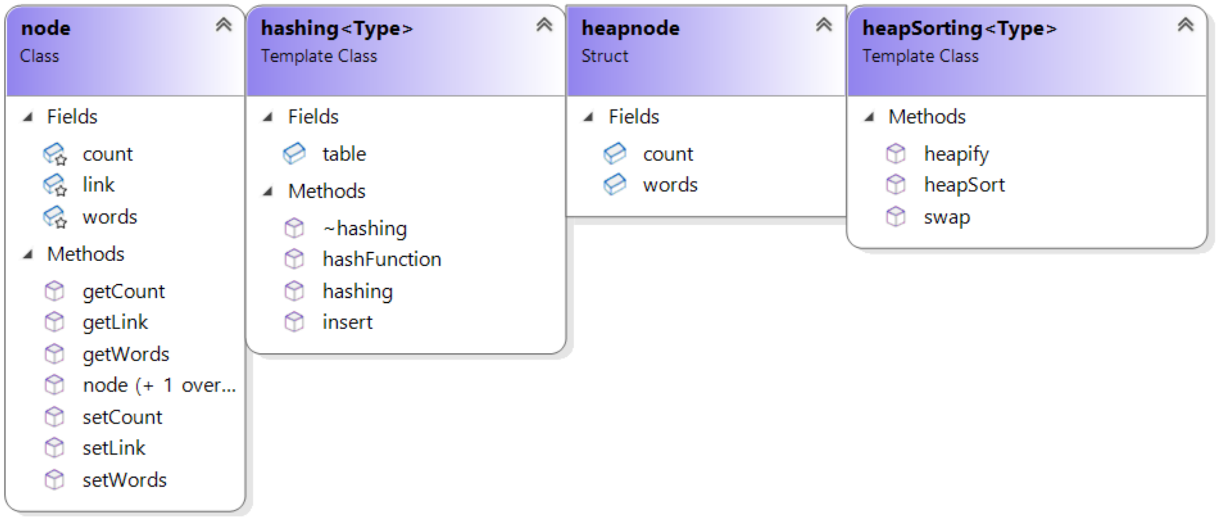
Student ID	Name-Surname
1900812	Umniyah Sameer Haitham Abbood
1901000	Jawad Ntefeh
1902293	Şükran Öykü Erdik

The purpose of the project is to employ the most efficient data structure and sorting algorithms to find the top ten most often occurring words in the fastest running time. Two datasets have been provided: the first is "PublicationsDataSet," from which we extracted the word and its count that appeared in the "unigramCount," and the second is "stopwords." We implement our code in such a way that if a stopword occurs, it is not inserted into our hash table.

We chose **open Hashing (Chaining)** for our project since keys are stored in linked lists associated with cells of a hash table in open hashing. As a result, we had a **node class** and a **hashing class**. The reason for using open hashing instead of other data structures is that it is more efficient; we can insert a node in O (1) in the best and average cases, however the worst-case O(n). Moreover, we selected **heap sorting** to determine the top ten most often occurring words; the best, average, and worst complexity is O(n*log(n)), which is superior to insertion, selection, and quicksort.

1. UML DIAGRAMS OF THE CODE


node
#words: std::string
#count: int
#*link: node
+ node(const std::string words, int count)
+ node()
+ getWords() const: const std::string
+ getCount() const: int
+ setCount(int count): void
+ *getLink() const: node
+ setLink(node *link): void
+ setWords(const std::string words):void
hashing
+ *table[size]: node
+ hashing()
+ hashFunction(std::string const& str): long long
+ insert(std::String &words, int count): void
+ ~hashing()
heapSorting
- swap(heapnode &first,heapnode second):void
- heapify(heapnode arry[], int n, int i): void
- heapSort(heapnode arry[], int n): void
heapnode
- words: std::string
- count: std::string
Main
+ publications: std::string
+ stop: std::string
+ publication_file:std::ifstream
+ stop_file:std::ifstream
+ words:std::string
+ stopwords[]: std::string
+ hash: hashing
+ sort: heapSorting



2. **Implementation**

Before starting implementation, we made four header files.

- **node.h:** implemented and added the related variables and functions.
- **hashing.h:** include ”node.h” used to connect to the node header and get the values from them. Also implemented related functions with the template classes.
- **heapnode.h:** heapSorting header needs to take heap node in some functions so heapnode header implemented with its's value
- **heapSorting.h:** include “hashing.h” used to get the variables from hashing header and implement related functions and structures with the template class.

After initializing the values and functions in the header files, we started coding into main.cpp

-main (): we have opened our txt files, initialized the instance of the class that we have such as **hashing, heapsorting**, and the other needed variables. Our first goal was to read “PublicationsDataSet” we read it by using getline inside a while loop using a **filter ()** function, after that we called **heapAll()** function to find and print the top ten words.

- class node:

- We use it to create a node that will take the word as a string, the count as an integer, and a node *link which will be used to link the node to each other.
- It has:
- **node ():** default construct
- **node (const std::string& words, int count):** used to get the word and the count
- **setters and getters:** as word, count, link is initialized as access modifier protected for encapsulation

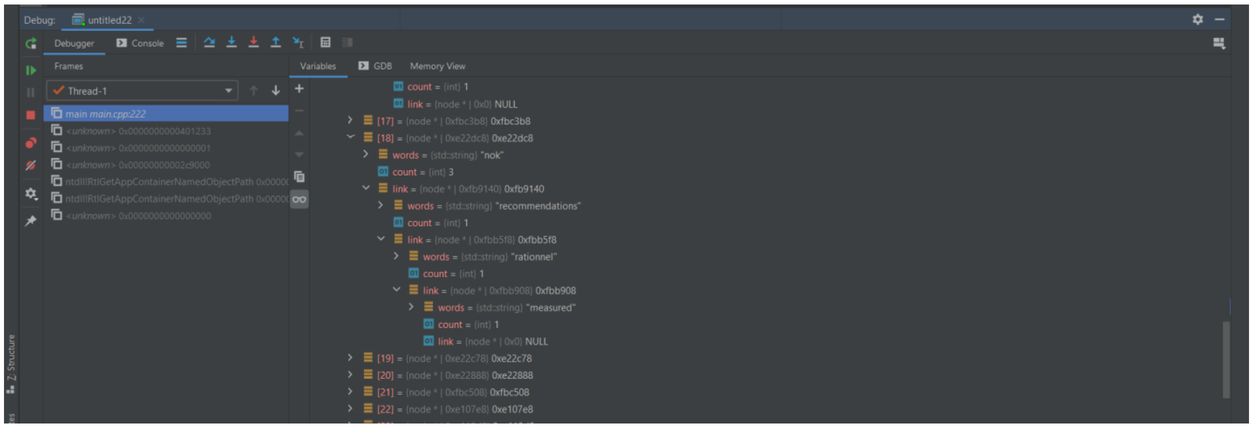
- class hashing:

- We use this class to create a hash table when we inserted the nodes into it
- It has:
- table which is a node type and takes the size from the globule variable size which is 6500.
- **hashing ():**default construct
- **long long hashFunction(std::string const& str):** which is a function that gets the string and calculates the key where the node will be inserted into the table, we borrowed this function from GitHub, however, we made a few changes we chose variable p to be 47 as it is a prime number and we wanted to make sure that we returned a positive key which is the index where the node will be inserted by using abs() function
- **~hashing ():** deconstructed to destroy the table to not have a memory leak
- **void insert (std::string& words, int count):** this function takes a word and count and insert them to the correct index in the hash table, it had 3 pointer nodes**, node* newWord** which will be used to hold the word and count vale, *** current, * trailCurrent,** are used to iterate over the nodes in the table. And it has 2 cases

    - **case 1: if** the index in the table is null, which means no node has been inserted yet, newWord node store the value, set its link to null, and then we insert it into the table
    - **case 2:** if the index in the table is not null, which means at least there is one node that has been inserted and the reason that we have more than one node in an index is the collusion that happening, and our way to resolve that is to create a link list of nodes in that index.

      First, we assign **current** to the index in the table and **trailCurrent** to null.

        - **Case2A:** if the inserted word has been inserted before, we add its count to the original word as it is the most crucial part of our project, we are trying to find the most frequent word, therefore, we cannot allow duplication. Here the current = table[key]; trailCurrent = NULL.
        - **Case2B:** if the word is new, we iterate over the nodes in that specific index
            - **Case2B1:** if current is null, insert the node, set newWord like to null and set trailCurrent link to the newWord
            - **Case2B2:** if current is not null, which means we have inserted the same word before, therefore we just add count to current which is the pointer that point to the original node that had the same word, it is like **Case2A,** However, here we are iterating inside the link list of a specific index



```
Frames
Thread-1
main main.cpp:222
<unknown> 0x0000000000401233
<unknown> 0x0000000000000001
<unknown> 0x00000000002c9000
ntdll!RtlGetAppContainerNamedObjectPath 0x00000
ntdll!RtlGetAppContainerNamedObjectPath 0x00000
<unknown> 0x0000000000000000
```

```
Variables
            count = {int} 1
            link = {node * | 0x0} NULL
  > [17] = {node * | 0xfbc3b8} 0xfbc3b8
  v [18] = {node * | 0xe22dc8} 0xe22dc8
      > words = {std::string} "nok"
        count = {int} 3
      v link = {node * | 0xfb9140} 0xfb9140
          > words = {std::string} "recommendations"
            count = {int} 1
          v link = {node * | 0xfbb5f8} 0xfbb5f8
              > words = {std::string} "rationnel"
                count = {int} 1
              v link = {node * | 0xfbb908} 0xfbb908
                  > words = {std::string} "measured"
                    count = {int} 1
                    link = {node * | 0x0} NULL
  > [19] = {node * | 0xe22c78} 0xe22c78
  > [20] = {node * | 0xe22888} 0xe22888
  > [21] = {node * | 0xfbc508} 0xfbc508
  > [22] = {node * | 0xe107e8} 0xe107e8
```

Here, we can see that we have collusions in the debug mood.

- **bool isStopWord(std::string stop[], std::string uwords):** Is a global function that takes a string array **stop** which contains the stop words taken from "stopwords.txt" and a string **uwords** then loops through stop to check if **uwords** is a stop word or not. In the for loop the value **571** is used because that is the size of **stop.**
- **void filter(std::string str, hashing& hash, std::string stop[]):** Is a global function that takes a string **str** which contains text taken from the "PublicationsDataSet.txt" file, hash table hash which is passed as a reference to allow us to insert "filtered" words with their counts into it using the **insert()** function, and a string array **stop** which contains the stop words taken from "stopwords.txt" then loops through the text and when a word which matches the criteria is found, it inputs that word and its count into **hash** and starts looking for the next suitable word. The function has four string variables, **searchStr** which is used to assist in making the for loop more efficient **unigramWord**, **wordsinUnigrams**, and **counts** are used as placeholders within the loop and are reset with every iteration. The function also has two integer variables, **startUnigramsIndex** which is set to the index where **searchStr** is found + **searchStr's** length, and **endUnigramsIndex** which is set to **str's** length -1 both of which are used in the for loop to make it more efficient. Finally, the function also has a bool **digitcame** which is used within the loop when the time comes to pull each word's count. The loop has 2 main cases:

  - **Case 1:** if the variable **wordsinUnigrams** does not include the characters ":, the character at **str[i]** is added to **wordsinUnigrams** then if that character turns out to be a capital letter, it is converted using a nested **if** statement. Otherwise, an **if** statement and two **else if** statements are checked:

    - **if** the character at **str[i]** is a punctuation but is not an apostrophe, that character is separated out of the word.
    - **else if** the character at **str[i]** is a digit, it is removed from the word and the bool **digitcame** is set to true.
    - **else if** the variable **digitcame** is still false, the character at **str[i]** is added to the variable **unigramWord**.

- **Case 2:** an else block is called where a while loop continues to add **str[i++]** to **counts** as long as the character at **str[i]** is not a comma, the while loop contains an if statement which breaks the while loop **if** the variable counts contains }}. Then an **if** statement checks if the variable **unigramWord** is not empty and if it is not:

    - A string variable **uwords** and a stringstream **iss** which takes **unigramWord** as an argument are made and a while loop goes through iss and feeds that data into uwords as long as no blank space is encountered. The while loop has 3 if statements as follows:

        - **If** the last character in **uwords** is an apostrophe, that character is popped.
        - **If** the first character in **uwords** is an apostrophe, **uwords** is assigned a new value from the second character in **uwords** to the last character in it.
        - If the variable **uwords** has more than 1 character **and** is not considered a stop word, the word and it's count are inserted into hash as follows **hash.insert(uwords,atoi(counts.c_ctr()));** and the variable **counter** is incremented by one. Finally, the variables **unigramWord**, **wordsinUnigrams**, **counts**, and **digitcame** are reset to their original values and the loop starts to look for the next word**.**

- struct heapnode: it has string word and integer count, we needed it in the **heapSorting class**

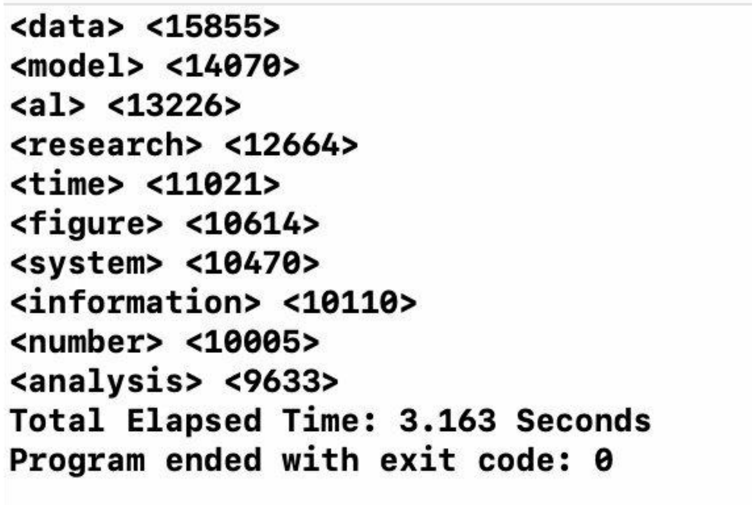
- class heapSorting: the idea behind the heap sort is getting an array, transferee it to complete binary tree, which fillies every level, with the possible exception of the last level completely from right to the far left

It has

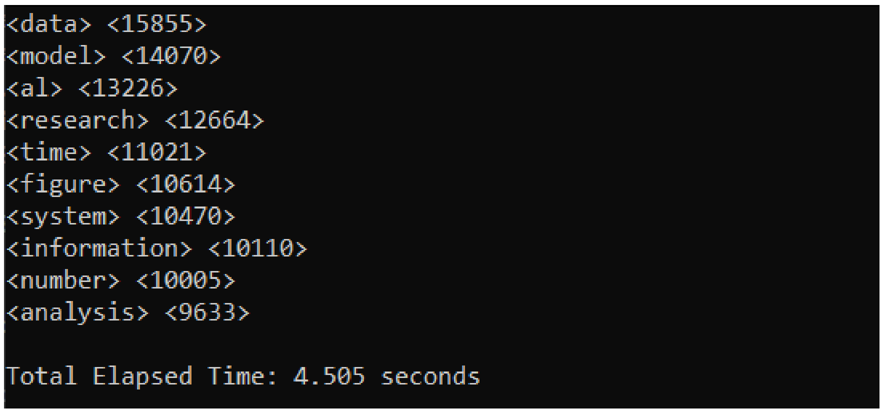
- **void swap(heapnode& first, heapnode& second):** it initialized tempword and temp count as we need a third varable to swap two values. This function has been used in heapify() and heapSort() functions
- **void heapify(heapnode arry[], int n, int i):** this function is re-heaping the tree to make sure it is a max tree, which means the root is the largest node, and each parent node is larger than its left and right child. We can get left child by (2* partentIndex+1) and we can calculate right child by (2* partentIndex+2), therefore we check if the left child larger than the root, we swap them using **swap(heapnode& first, heapnode& second)** function, same with the right child. And keep doing that in a recursive manner
- **void heapSort(heapnode arry[], int n) :** in this function we are taking an array and its size, we have two for loops
    - the first for loop **calls heapify(arry, n, i)** to build the max tree
    - the second for loop calls the **sawp()** function to swap the root with the last index, then call **heapify(arry, n-1, 0),** which is re-heap the tree by reducing it

– **void heapAll(hashing& heapHash):** Is a global function that takes the hash table and transfers it to arraymby creating **heapnode* arry** which take **counter** as a size, which is a globule variable that initially was zero, however it increases every time a node inserted to a table. To do that we need a variable (k) that will keep track on unique word and a **node* current** to iterate over the link list using for loop which has a while loop inside it. After that we call **heapSort(arry, k)** and then we print the top ten word and their count, then delete *** arry.**

3. **output**
   Best time:3.163 seconds
   Average time: 4.505 seconds
   Worse time: 4.858 seconds

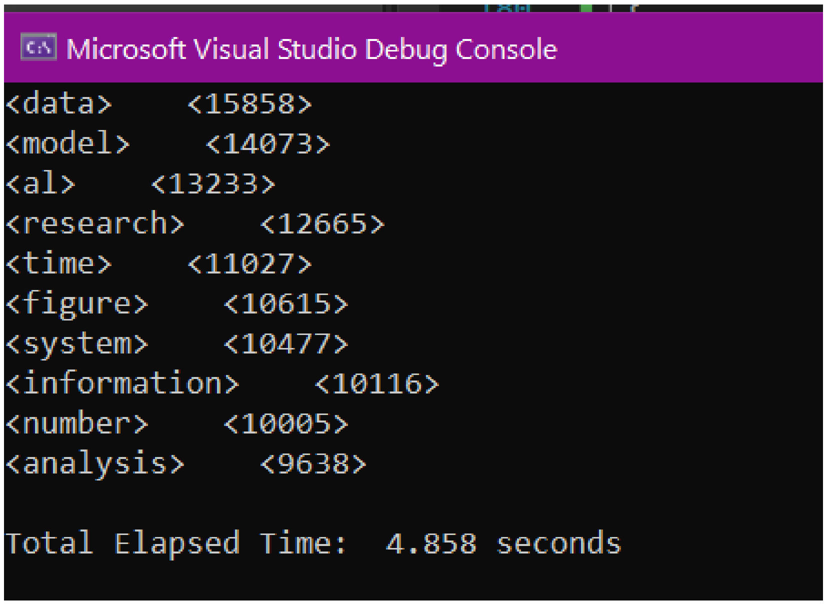


```
<data> <15855>
<model> <14070>
<al> <13226>
<research> <12664>
<time> <11021>
<figure> <10614>
<system> <10470>
<information> <10110>
<number> <10005>
<analysis> <9633>
Total Elapsed Time: 3.163 Seconds
Program ended with exit code: 0
```



```
<data> <15855>
<model> <14070>
<al> <13226>
<research> <12664>
<time> <11021>
<figure> <10614>
<system> <10470>
<information> <10110>
<number> <10005>
<analysis> <9633>

Total Elapsed Time: 4.505 seconds
```



```
<data>    <15858>
<model>    <14073>
<al>    <13233>
<research>    <12665>
<time>    <11027>
<figure>    <10615>
<system>    <10477>
<information>    <10116>
<number>    <10005>
<analysis>    <9638>

Total Elapsed Time:  4.858 seconds
```

https://www.geeksforgeeks.org/heap-sort/ (we get the main idea and implementation from there)
https://github.com/hiamsevval/Top-10-Frequent-Words (we get the how to calculated the key from there with few changes)