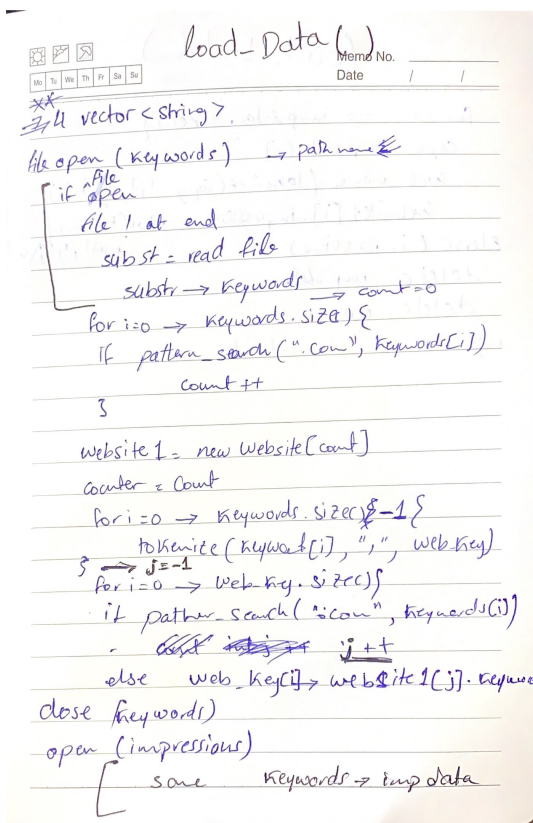


i. Indexing and Ranking algorithms pseudo-code:

a) Indexing algorithms:

1. load_website_data():

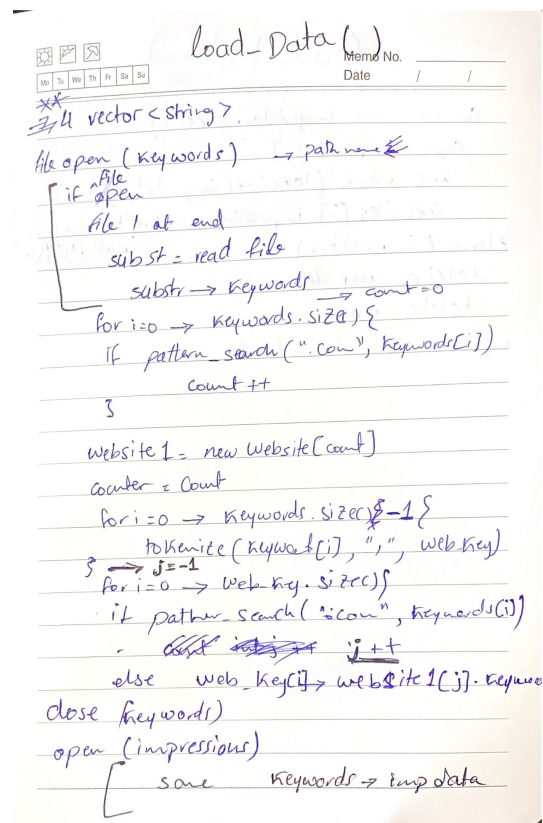
The function reads from the CSV files containing website data, such as URLs, impressions, and click-through rates (CTR), and extracts the relevant information. It then stores this data in the websites1 array, which is used throughout the program for indexing and ranking purposes.



```

load_Data()
Memo No.
Date / /

**
vector < string >
file open (Keywords) → path name
if open
  file ! at end
  substr = read file
  substr → Keywords → count = 0
  for i = 0 → Keywords.size() {
    if pattern_search(".com", Keywords[i])
      count ++
  }
  website1 = new Website(count)
  counter = count
  for i = 0 → Keywords.size() - 1 {
    tokwrite(Keywords[i], ",", web_key)
  }
  for i = 0 → web_key.size() {
    if pattern_search(".com", Keywords[i])
      count ++ j ++
    else web_key[i] → website1[j].key
  }
  close (Keywords)
  open (impressions)
  same Keywords → imp data
  
```



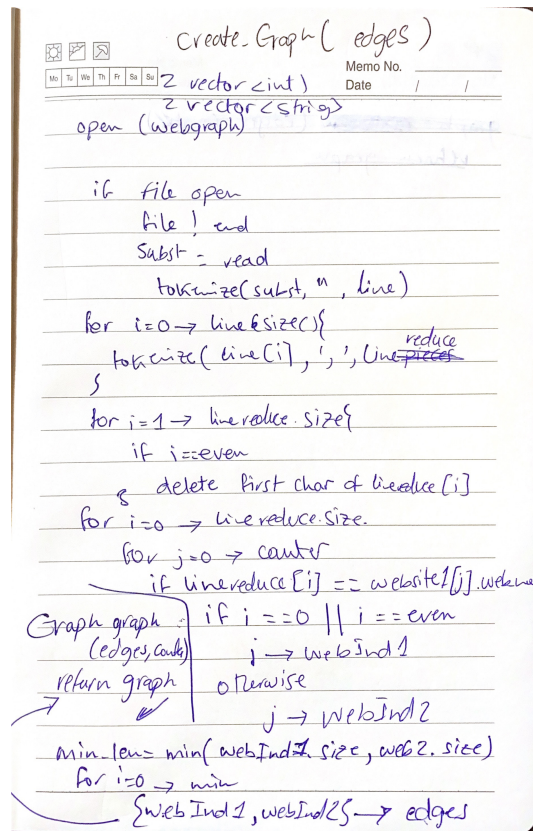
```

load_Data()
Memo No.
Date / /

**
vector < string >
file open (Keywords) → path name
if open
  file ! at end
  substr = read file
  substr → Keywords → count = 0
  for i = 0 → Keywords.size() {
    if pattern_search(".com", Keywords[i])
      count ++
  }
  website1 = new Website(count)
  counter = count
  for i = 0 → Keywords.size() - 1 {
    tokwrite(Keywords[i], ",", web_key)
  }
  for i = 0 → web_key.size() {
    if pattern_search(".com", Keywords[i])
      count ++ j ++
    else web_key[i] → website1[j].key
  }
  close (Keywords)
  open (impressions)
  same Keywords → imp data
  
```

2. Create_Graph():

This function plays a crucial role in indexing by establishing the relationships between different web pages and representing them in the form of a graph data structure. The graph captures the connectivity between web pages, allowing search engines to navigate and analyze the interlinking patterns of websites.



b) Ranking algorithms:

Similar to the YouTube video linked in the presentation, I distributed the transferred page rank from a specific page to its linked destinations in the subsequent iteration using the same division method for all the outgoing links. Essentially, the page rank attributed by an outgoing link corresponds to the website's own page score divided by the count of its outgoing links.

I used these 4 functions to calculate the page rank.

1. initialize_PR():

This function initializes the PageRank values for each website in the graph. It sets an initial PR value for each website, which is later refined through iterative calculations in the 'calculate_PR()' function.

counter2 = counter

outgoinglinks = create float array of size counter

```
for i = 0 to counter: outgoinglinks[i] = 0
for i = 0 to counter:
    // Initialize PageRank for each website
    websites[i].PR = 1 / counter2
for j = 0 to counter:
    for each link in graph.adjList[j]:
        // Count the outgoing links for each website
        outgoinglinks[j]++
for i = 0 to 2:
    // Perform PageRank calculation iterations
    calculate_PR(graph, websites, outgoinglinks)
2. calculate_PR():
```

This function calculates the PageRank values for each website in the graph. PageRank is a ranking algorithm that assigns scores to websites based on the structure of the web graph. The higher the PageRank value, the higher the ranking of the website.

```
webs = create empty array of vectors with size 4
webs2 = create empty array of vectors with size 4
for i = 0 to counter:
    for j = 0 to counter:
        for each link in graph.adjList[j]:
            if link is equal to i:
                // Store pairs of PageRank and outgoing links for each website
                webs[i].push_back(make_pair(websites[j].PR, outgoinglinks[j]))
for i = 0 to counter:
    for z = 0 to length of webs[i]:
        if z is 0:
            // Calculate PageRank for each website based on the stored pairs
            websites[i].PR = webs[i][z].first / webs[i][z].second
```

else if $z > 0$: websites[i].PR += webs[i][z].first / webs[i][z].second

3. calculate_CTR():

This function calculates the Click-Through Rate (CTR) for a given website. CTR is a metric that measures the ratio of users who clicked on a website's link to the total number of users who viewed the website's link. The higher the CTR value, the higher the ranking of the website.

// Calculate Click-through Rate (CTR)

*CTR = (website.clicked / website.impressions) * 100*

return CTR

4. website_score():

This function calculates the overall score for a given website based on its PageRank, impressions, and CTR values. It combines these factors to determine the ranking score for each website.

// Calculate the website score

CTR = calculate_CTR(website)

*score = (0.4 * website.PR) + ((1 - ((0.1 * website.impressions) / (1 + 0.1 * website.impressions))) * website.PR + ((0.1 * website.impressions) / (1 + 0.1 * website.impressions))) * CTR * 0.6*

return score

ii. Indexing and Ranking algorithms complexity analysis:

a. *Indexing algorithms:*

1. Time complexity:

The time complexity for the *load_website_data()* function is $O(n*m)$, where n is the number of websites and m is the total length of keywords.

The time complexity for the *Create_Graph()* function is $O(E+n)$, where E is the number edges and n is the length of the web graph data.

So the total complexity can be written as $O(n*m + E + n)$.

2. Space complexity:

The space complexity for the *load_website_data()* function is $O(n)$, where n is the number of keywords of all websites.

The space complexity for the *Create_Graph()* function is $O(E+V)$, where E is the number edges and V is the number of vertices which are the websites in our case.

So the total complexity can be written as $O(n+ E+V)$.

b. Ranking algorithms:

1. Time complexity:

calculate_PR() and *initialize_PR()* have a time complexity of $O(N^2)$, where N is the number of websites.

calculate_CTR() and *website_score()* have a time complexity of $O(1)$, as they involve only a few arithmetic operations.

So the total complexity can be written as $O(N^2)$.

2. Space complexity:

calculate_PR() and *initialize_PR()* have a time complexity of $O(N)$, where N is the number of websites.

calculate_CTR() and *website_score()* have a time complexity of $O(1)$, as they involve only a few arithmetic operations.

So the total complexity can be written as $O(N)$.

NOTE: in the ranking algorithms the space complexity of the code is primarily determined by the size of the Website array *websites1* and the data stored in the STL containers. The complexity can vary depending on the input data and the number of websites.

iii. *Main data structures:*

I used different data structures like:

Structures: for the graphs and websites.

Vector: Used to store lists of objects dynamically. It is used for various purposes, such as storing graph edges, website keywords, search words, website indices, and more.

Map: Used to store key-value pairs, providing fast lookup based on the key. They are used for mapping keywords to websites and for storing website scores.

Priority queue: Used to create a priority queue of pairs. It is used for ranking websites based on their scores.

List: Used to store a list of integers. It is used for iterating over adjacency lists in the graph.

iv. Design trade-offs:

There are not many design trade-offs in my program; however, one thing that can count as so, is the tokenize function that I implemented as it is more custom than the standard default tokenization function, my function worked for some specific cases I had however, I noticed that it didn't work for all cases and I had to use the default one "strtok" in the instances that my function didn't work as required. My program identifies the "and" and "or" only if they are written as "&" or "|" so it is easier to write and to have a unified way of typing a search query. I also have the program clear the screen after every input as to make it cleaner for the user, it is not a trade-off really but more aesthetically pleasing.