The goal of our project is to create a simple web search engine wherein we identified the following tasks to be performed in order to achieve the goal.

1. Write a program that takes in ‘n’ web pages as an input. The program should figure out the part of speech using NLP and only consider storing nouns, adjectives, verbs and adverbs.
2. Write a program to store the words obtained from step (1) in Quadratic Probing Hash Table and sort the values of hash table in the order of decreasing frequency.
3. Write a program to suggest the words using edit distance algorithm.
4. Write a program that allows the user to view/delete his/her history.

**Description of the System**:

Our system is divided into two parts:

1. ***Pre-processing phase:***

* In this phase, firstly we read 100 html web pages content. The web pages content are tokenized and we figure out the part of speech using **Natural Language Processing (NLP)** for which we have used Stanford core NLP packages**.** Since it does not make sense to store words like “of”, “for”,” in” in the hash table because the user won’t search for these words. Hence, we only consider storing nouns, adjectives, verbs and adverbs. This is done by in the program **HTMLToText.java**.
* Next, we store these words in the **Quadratic Probing Hash Table**. The structure of Quadratic Probing Hash table is **<word:<html\_file\_name: frequency>>.** So, we have a hash table inside another hash table. The first hash table consist of the NLP filtered words as the key and second hash table as the value. The second hash table consist of html file name as key and frequency of that word in the html file as the value. Then we sort the second hash table values based on the decreasing order of the frequency using **merge sort algorithm**. This allows us to display the websites having the highest frequency, which is more likely the website the user wants to visit. The above concept is implemented in the program **HashmapCreation.java.**
* In the best case, the search time complexity of any hashing algorithm is O(1). However, there are several factors that affects the performance of hash tables. That is the reason we didn’t use linear probing in order to prevent the performance drop due to the primary clustering. Similarly, we didn’t use separate chaining hash table as it maintains a linked list during collision, and this can take the search complexity of O(n) in the worse case. For, Insertion of elements in quadratic probing hash table the time complexity is O(1/1- α), where α is the number of keys divided by hash table size. Hence, we decided to go with Quadratic Probing Hash Table.

Similarly, the reason we chose merge sort algorithm to sort the hash table is because it has an average time complexity of O(nlogn) where n is the length of hash table. We could have used other algorithms like quick sort and heap sort, but the merge sort algorithm proved to have the least CPU time for sorting the elements as compared to quick sort and heap sort.

1. ***Application ready to use phase:***

* Once the pre-processing phase is completed, the application is ready to use.
* The user can search for a word using our search engine. When the user looks for a word and if the word is not available in the hash table(websites), then we suggest top 5 words to the user using **edit distance algorithm**. We have set the threshold value to 2, which means the if the user entered word and the word in the hash table has edit distance less than or equals to 2, we suggest such words to the user. The time complexity of edit distance algorithm is O(mn) where m and n are the length of strings for which we calculate edit distance.
* If the word that user is looking for is found in the hash table(websites), then we display the top 10 websites to the user on the basis of decreasing frequency of that word. When the user enters any one of the URL, then that URL, the current timestamp and an auto-incremental id is stored in the **linked list** object. So, for any URL that the user opens, that information is stored in the linked list object. The user can also view or delete the history. The time complexity to view the history is traversing through the entire list which is O(n) where n is the length of the linked list. Similarly, the user can delete the history by id. The time complexity for this is O(n) where n is the position of the node which contains the id that the user wants to delete.

So, the linked list here acts as a browser history. The reason we chose linked list data structure instead of any other algorithms was because for every node insertion the time complexity is always O(1). We are always inserting the nodes at the beginning of the linked list object, because when the user visits any website, that website appears at the top of the browser history. Hence, we always insert the new node at the beginning of the linked list, which takes the time complexity of O(1).

This application can be run by using the java program **FinalRun.java**