**NETWORK BROWSER HISTORY KEEPER**

**Submitted by:**

Ayush Saxena (17103108)

Shivank Agarwal (9917102262)

Harshit Kumar (17103019)

Shrey Gupta (17103126)

**Submitted to:**

Dr. Mohit Kumar



Jaypee Institute of Information Technology

Sector 62, Noida

**Introduction:**

Suppose we have learned something from internet but we still are unsure of the topic. By searching again the same term we see that the browser automatically completes the string searched before. The browser uses trie to complete the string. So this is what are we trying to implement. A network browser keeps a history of the URLs of sites

that you have visited. By organizing this history as a trie, the user need only type the prefix of a previously used URL and the

browser can complete the URL. The technique we will try to use

is maintains trie of strings and auto completes the word by searching for a string with the same prefix and having maximum frequency count.

**Data Structures**

• **Trie**

• **Multi List**

**The Trie Data Structure:**

The word trie is an infix of the word “retrieval” because the trie can find a single word in a dictionary with only a prefix of the word. So, a Trie is nothing but a tree and each node in it contains the number of pointers equal to the number of characters of the alphabet. For example, if we assume that all the strings are formed with English alphabet characters 'a' to 'z' then each node of the trie can contain maximum all possible character. The standard trie for a set of string S is an ordered tree such that

•The root represents an empty string("")

•Each node but the root is labeled with a character

•The children of a node is alphabetically ordered

•The paths from the external nodes to the root yields the strings of S

**Time Complexity of Trie**

• Time Complexity of trie creation: O(Length of longest string \* Number of strings)

• Time Complexity of searching: O(Length of string)

**Applications of Trie:**

• **Auto Complete:** Auto Complete functionality is used widely in mobile apps and text editors. Trie is an efficient data structure widely used for its implementation. Trie provides an easy way to search for the possible dictionary words to complete word

because of the following reasons

• **Spell Checkers:** Spell checking is a three-step process. Check if a word is in a dictionary, generate potential suggestions, and then sort the suggestions–hopefully with the intended word on top.

Tries can be used to store that dictionary and by searching the words over the data structure one can easily implement a spell checker in the most efficient way. Using trie not only the lookup for a word into the dictionary becomes easy but an algorithm to provide the list of valid words or suggestions can be easily constructed.

• **Longest Prefix Matching:** Also called Maximum prefix length match refers to an algorithm used by routers in Internet protocol(IP) networking to select an entry from a routing table.

One of the first IP lookup techniques to employ tries is the radix trie implementation in the BSD kernel. Optimizations requiring contiguous masks bound the worst case lookup time to O(W) where W is the length of the address in bits. In order to speed up the lookup process, multi bit trie schemes were developed which perform a search using

multiple bits of the address at a time.

**The MultiList Data Structure:**

We will use this data structure to print the history of the URLs. A multi-linked list is a more general linked list with multiple links from nodes. In a general multi-linked list

each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer. Multi-lists are essentially the technique of embedding multiple lists into a single data structure. A multi-list has more than one next pointer, like a doubly linked list, but the pointers create separate lists**.**

**Implementation:**

As we know from the previous text that a Trie is a tree like data structure which stores words such that the search for a word is proportional to the length of the word. Imagine a situation where a user is typing a word and he is mid way through. If you want to display all possible suggestions which he could have meant, then you need to find all the  words which starts with that prefix.

This is one class of problems which Trie is meant to solve. When the user starts typing, we initialize a character buffer which serves as a prefix. And every time there is a new character we append it to the prefix and search our Trie for the prefix. If the prefix exists, we return all the words followed by that.

**Adding a word to the Trie:**

Here is the algorithm to add a word to the Trie

* Start from the root of the Trie
* For the first character in the word, find if there exist a child node to the root containing that character.
* If no node is present, then add a node with the first character of the word, and subsequently add child nodes for the following characters.
* If the node is found, move to the next character in the word and the child node in the Trie.
* Repeat this process until either the word is exhausted or we reach to a null child node.
* If reached a null child and there is still part of the word remaining, just add it child nodes  one after the other

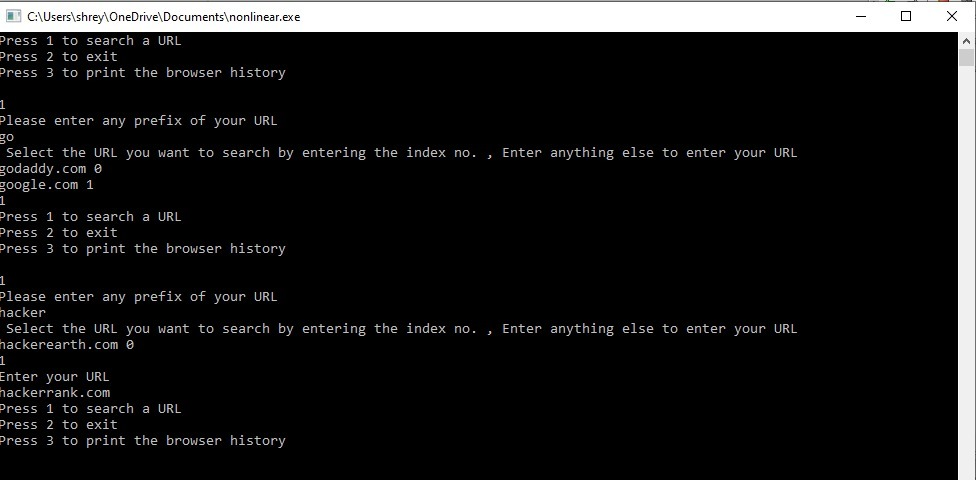
### Fetching word list for a prefix:

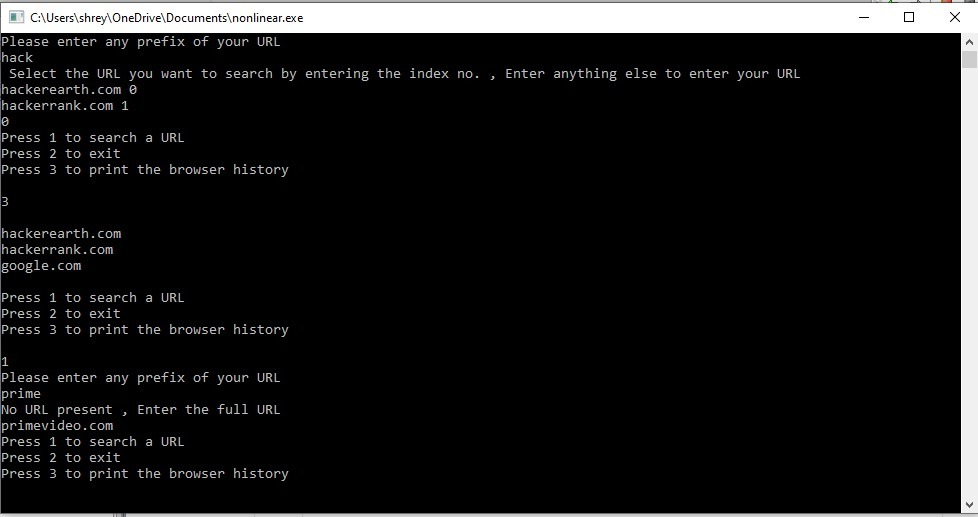
* Traverse through the Trie to find the node which stores the last character of the prefix.
* Two cases may occur
  + Either the prefix gets exhausted
  + We reach a null node.
* If we reach a null node then there are no words with this prefix so return
* If the prefix gets exhausted at a node terminal, then the whole sub tree of terminal qualifies for the result.
* Print all the paths to all leaves from this node. (You can use any technique to print all paths from a given node, here we will use a queue.

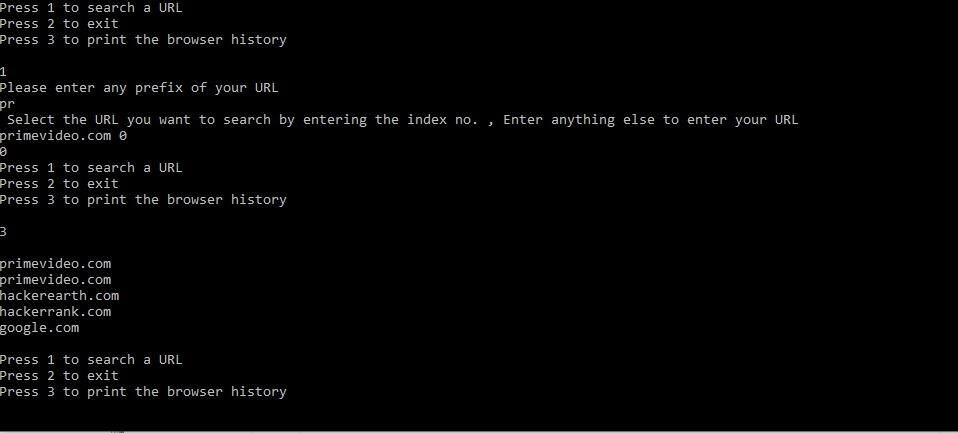
**Functions:**

* **Addword():** Function takes a string as a parameter and add the characters of the string to the existing trie.
* **printURL():** Function takes a string as a parameter and iterate over the string to check if string is present in the trie or not.It return 0 if string is not present, return -1 is if string is present and last character of string is terminal. Otherwise, function calls on suggestURL().
* **suggestURL():** Function prints all the string whose one of the prefix is given string.
* **Createlist():** Function creates the Multilist and store the strings.
* **Printmultilist() :** Function prints all the string contains in multilist.

**Screenshots:**

****

****

****

**Conclusion:**

We see that trie is the most efficient data-structure used to solve the auto-complete problem as there are varying length keys and all the descendent of a node have a common prefix of the string associated with that node