Analysis of fast searching techniques in dictionary with large data

Along with complexity and time based graphical analysis

Team 39
Yash Gupta - 191IT158
Sarthak Jain - 191IT145
Rishit - 191IT141

Introduction

In a digital dictionary, the main point of consideration is searching time. Words are stored in lexicographical order in a dictionary, so we have the advantage of using binary search to get our desired word. That is considerably faster than naive searching but we can still do better.

Trie is a data structure which can be used to do faster searching on large data in which the time complexity of searching is O(length of search word). Caching has also been implemented to increase the performance further.Caching algorithms can be really helpful where large databases are to be considered.

Problem Statement

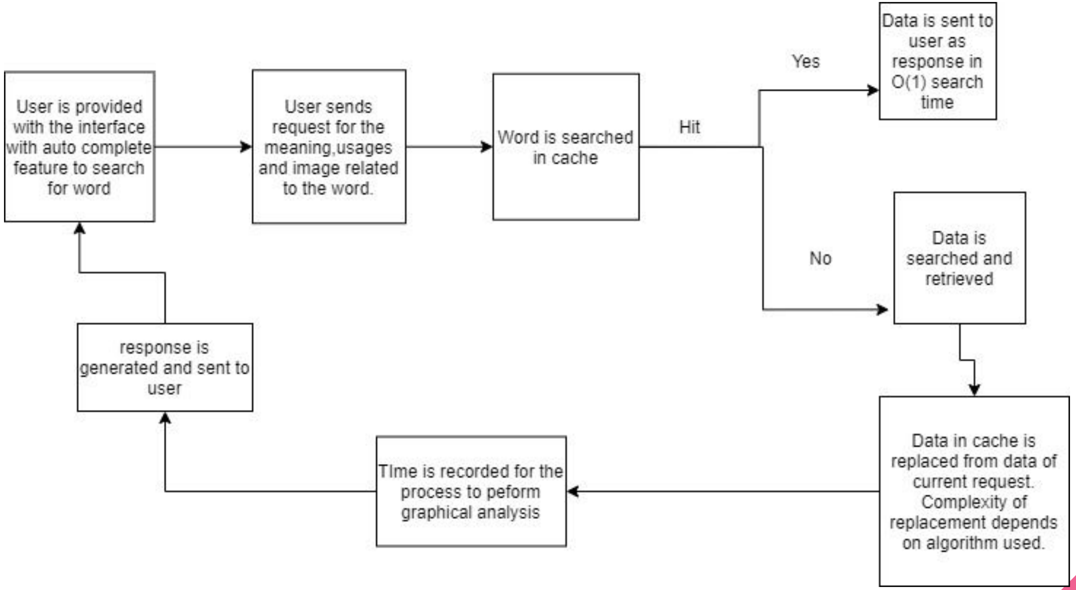
Comparison and analysis of fast searching techniques on English dictionary with large data

OBJECTIVES

1) Implementing LRU, LFU, FIFO, and RR caching algorithms for improving average search time in a data query.
2) Implement Trie based data storage and algorithms for fast searching and autocompletion based on prefix.
3) Performing Complexity analysis of implemented greedy algorithms.
4) Comparing and analyzing the run-times of searching in these algorithms and plotting the results.
5) Create a web app where user can enter a word to search and get the meaning , usage of word and picture related to word.

METHODOLOGY

Speedup using cache when data is stored using traditional storage methods :



Caching

Caching comes under **Dynamic Programming** paradigm as calculated results are memorised so that no need to calculate again

Hashmap is used to memorize(cache) the words along with their meaning.

When cache is full, for replacement, following algorithms were considered:

1. FIFO (First In First Out) - deletes the oldest item
2. LRU (Least Recently Used) - deletes least recently used item
3. LFU (Least Frequently Used) - deletes least frequently used item
4. RR (Random Replacement) - deletes a random item

FIFO

Paradigm - Greedy

Time Complexity of operations :

Insertion - O(1)

Deletion - O(1)

Searching - O(1)

Space Complexity = O(n), where

n is the size of cache.

Algorithm 2 FIFO Cache Replacement

Input: search word, result
Ensure: store the (word,result) pair in cache

```
    Initialisation: cache = hashmap(cache size)
if cache is full then
    Delete the first entry in cache
    store cache[word] = result
else
    Store cache[word] = result
end if
```

LRU

Paradigm - Greedy

Time Complexity of operations :

Insertion - O(1)

Deletion - O(1)

Searching - O(1)

Space Complexity = O(n), where

n is the size of cache.

Algorithm 3 LRU Cache

Input: search word, result
Ensure: store the (word,result) pair in cache

```
Initialisation:
Linked List - to store meaning and usages of word, hashmap(word,address) - to store the address of Node where details of word is stored

if word in hashmap then
    Get address of Linked List node using hashmap
    Move word's linked list node to the head of the linked list
else
    if cache is full then
        Find the Least Recently Used(present at the tail of linked lis)
        Delete that word from Linked List and hashmap
    end if
    Insert a new Node containing meaning of word at head of linked list
    Store the new node's adderss in the hashmap
end if
```

LFU

Paradigm - Greedy

Time Complexity of operations :

Insertion - O(1)

Deletion - O(1)

Searching - O(1)

Space Complexity = O(n), where

n is the size of cache.

Algorithm 4 LFU Cache

Input: search word, result

Ensure: store the (word,result) pair in cache

```
Initialisation: Linked List - to store meaning and usages
of word, hashmap(word, address) - to store the address of
Node where details of word is stored, hashmap(frequency,
address) - to store the head of linked list with given search
frequency

if word in hashmap then
    Get address of Linked List node using hashmap
    Move word's linked list node to the head of the linked
    list of the next frequency
else
    if cache is full then
        Find the Least Frequently Used (present at the tail of
        the lowest frequency of linked list )
        Delete that word from Linked List and hashmap
    end if
    Insert a new Node containing meaning of word at head
    of linked list with frequency 1
    Store the new node's address in the hashmap
end if
```

RR

Paradigm - Randomized

Time Complexity of operations :

Insertion - O(1)

Deletion - O(n), n is the cache size

Searching - O(1)

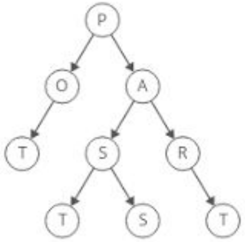
Space Complexity = O(n), where

n is the size of cache.

Algorithm 5 Random Replacement Cache

```
Input: search word, result
Ensure: store the (word,result) pair in cache
   Initialisation:
   hashmap(word,address) - to store the (word,meaning) pair
   as cache

   if word is not in cache then
      if cache is full then
         Select a random entry in cache and delete it
         Store the word meaning pair in cache
      else
         Store the word meaning pair in cache
      end if
   end if
```

Speedup using Trie Based storage of data :



Trie is a tree based data structure(prefix tree).

1. Create Trie class.
2. Implement search and auto-complete functions on the trie.
3. Store the dictionary data in trie.
4. Build a server to send the meaning and usages of word after searching in trie, whenever a request is made to the server.

Trie Build

Time Complexity :O(m*n),where m is the average word size and n is the number of words

Space Complexity = O(26*m*n), where m is the average word size and n is the number of words

Algorithm 6 Building trie of english words

Input: Dataset containing the english words in the oxford dictionary

Output: Trie built using the english words of the oxford dictionary

```
Initialisation
Node
(
  Node *child[26]
  bool End
)
Node *p = new Node;
Node.End = false;

for i = 0 to 26 do
  p.child[i] = NULL
end for

Node *p = root;

for i = 0 to word.length do
  index = key[i] - 'a';
  if (!p.child[index]) then
    p.child[index] = getNode
  end if
  p = p.child[index];
end for

p.End = true

return Trie
```

Trie Search

Paradigm - Greedy

Time Complexity :O(m),where m is the word size.

Space Complexity = O(1)

Algorithm 7 Searching a word in a trie

Input: Trie containing the english words
Output: Word searched and the corresponding meaning

```
    Initialisation
Node *p = root;
for i = 0 to word.length do
    index = key[i] - 'a';
    if (!p.child[index]) then
        return NOT FOUND
    end if
    p = p.child[index];
end for
Meaning of the word is saerched
return wordmeaning
```

Auto-Complete

Paradigm - Backtracking

Time Complexity :O(h^26),where h is the depth of the tree

Space Complexity = O(n),where n is the number of words in the dictionary.

Algorithm 8 Auto Complete using Tries

Input: Trie containing the english words

Output: Suggestion Words for the word searched

```
    Initialisation
Node *root = NULL
for i = 0 to word.length do
    if (!p.child[index]) then
        root = currnode
        break
    end if
    p = p.child[index];
end for
words = []
Node *p = root;

for i = 0 to 26 do
    index = key[i] - 'a';
    if (!p.child[index]) then
        words.push(currword)
        return to original word
    end if
    p = p.child[index];
end for


return words[]
```

Novelty

For Data stored in traditional Database :

- For data stored in traditional database, speedup is provided using the concept and algorithms of caching.
- Popular caching algorithms are analyzed and compared based on the cache hits and then best performing is chosen.

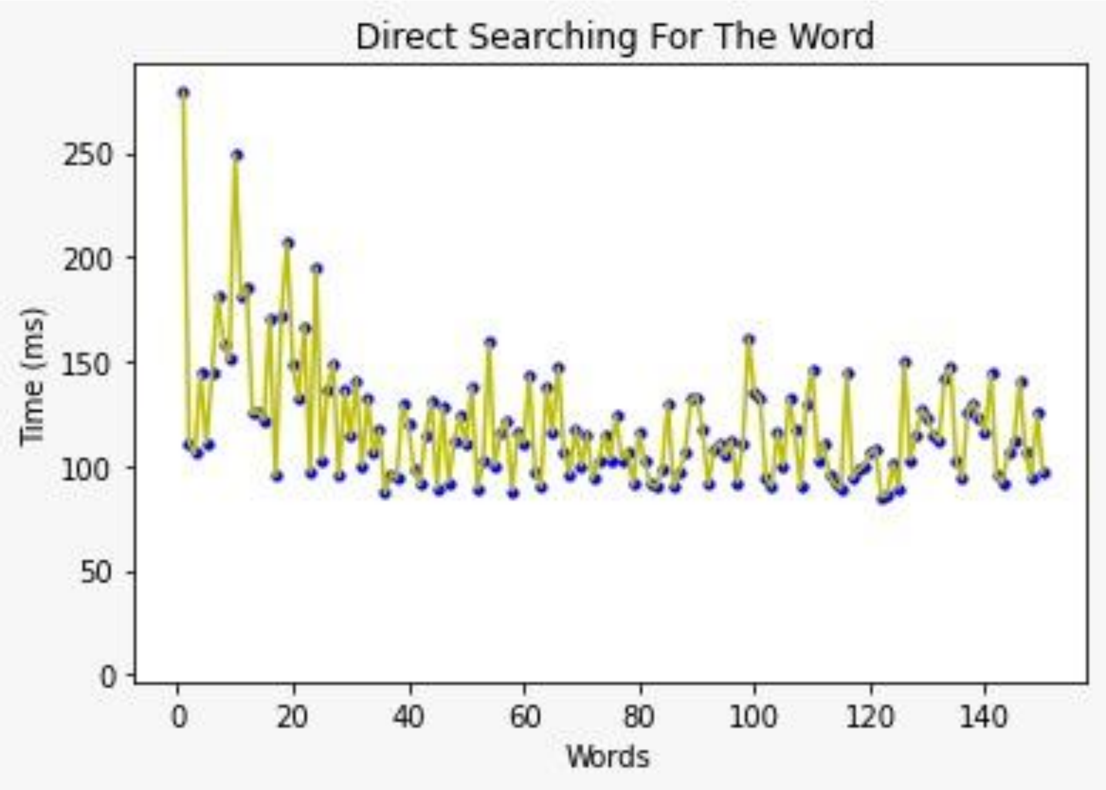
Data is stored in Trie Data Structure :

- An efficient way to store data for fast retrieval(searching).
- Huge decrease in searching time as compared to traditional storage methods

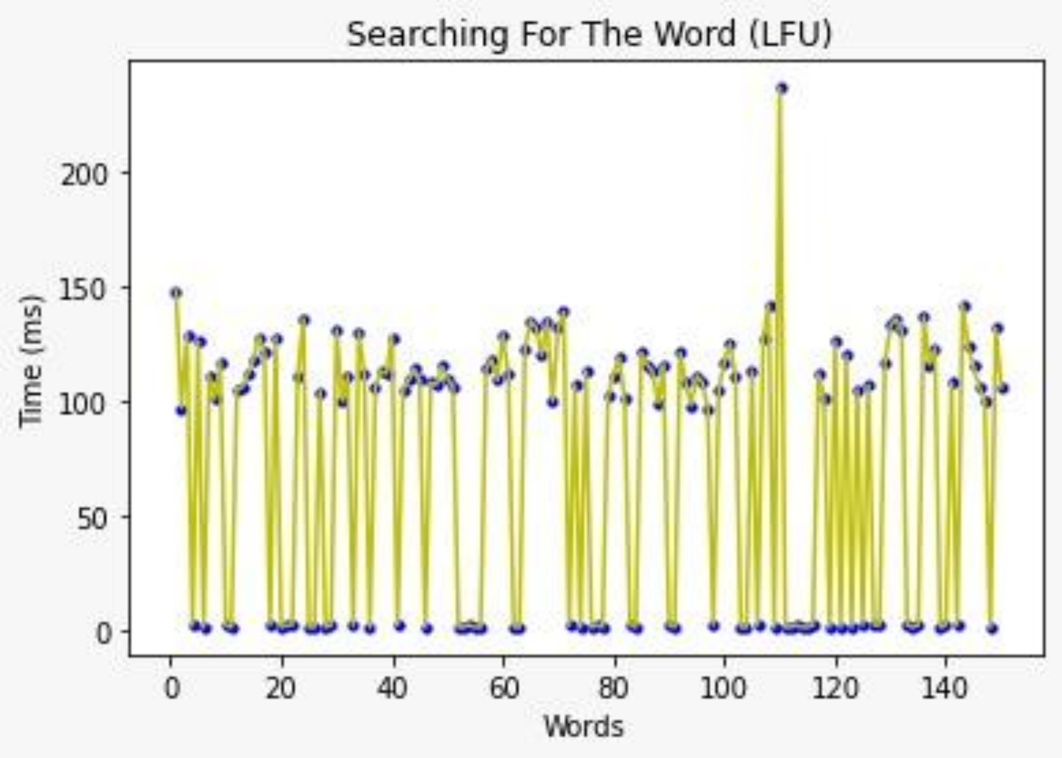
- For implementation of LFU cache algorithm, instead of using heaps based approach with O(logn) time complexity, Hashmap and linked list based approach is used which has O(1) time complexity for all the operations.
- Along with the meaning, usages and picture related to that word is also provided to user.

Result And Analysis

Graphical Analysis

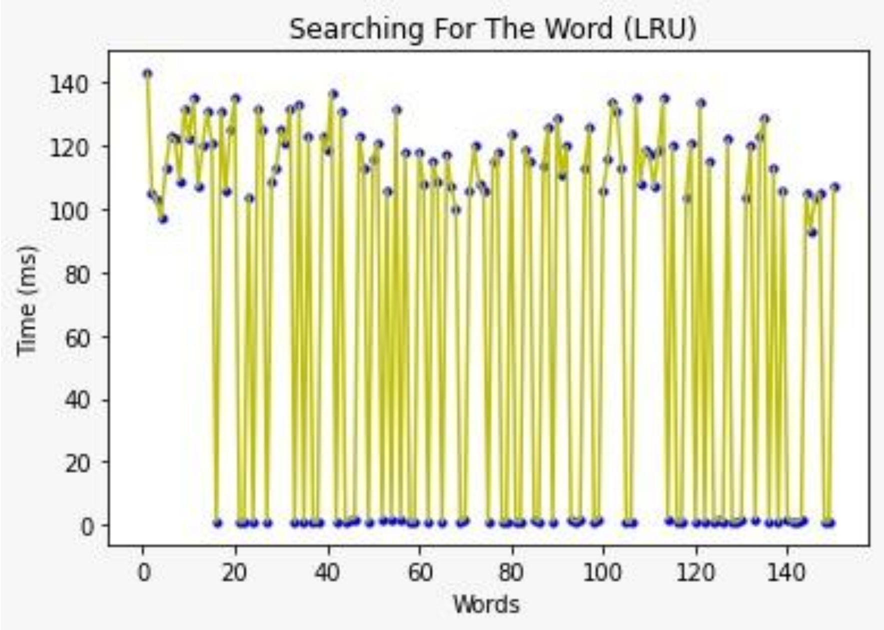


The average lookup time is higher since word is looked in main storage structure every time.

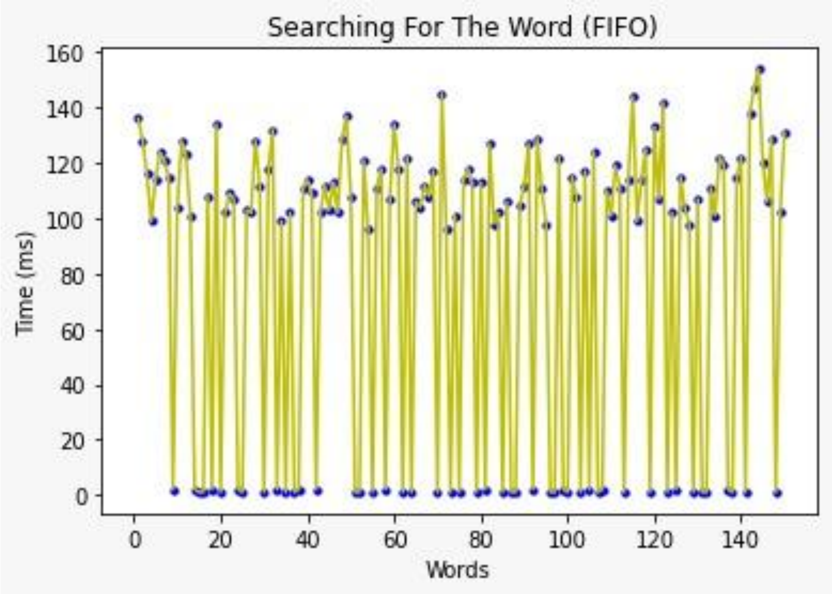


After caching, the search time during cache hit is 1-3ms, thus average search time is reduced.

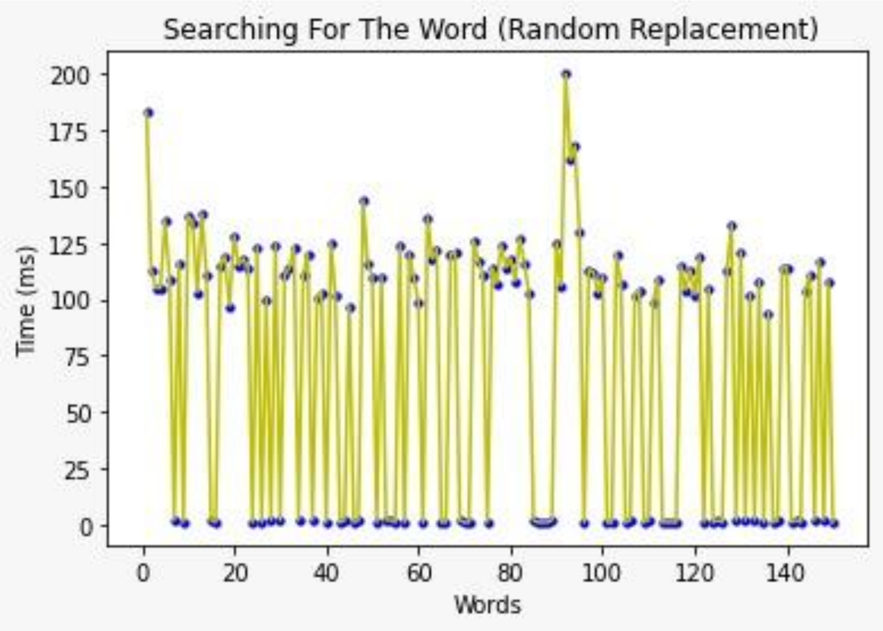
LFU Cache eviction algorithm.
Cache Hit - 58/150



Using LRU cache eviction algorithm
Cache Hits - 56/150



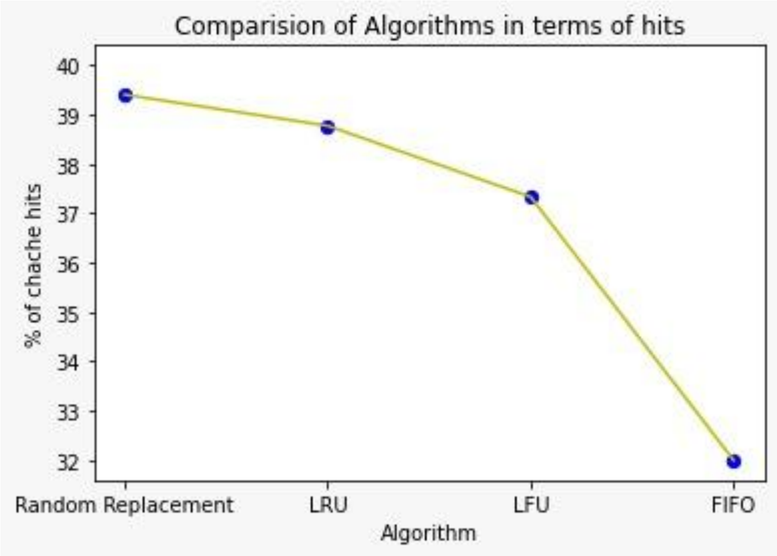
Using FIFO cache eviction algorithm
Cache Hits - 48/150

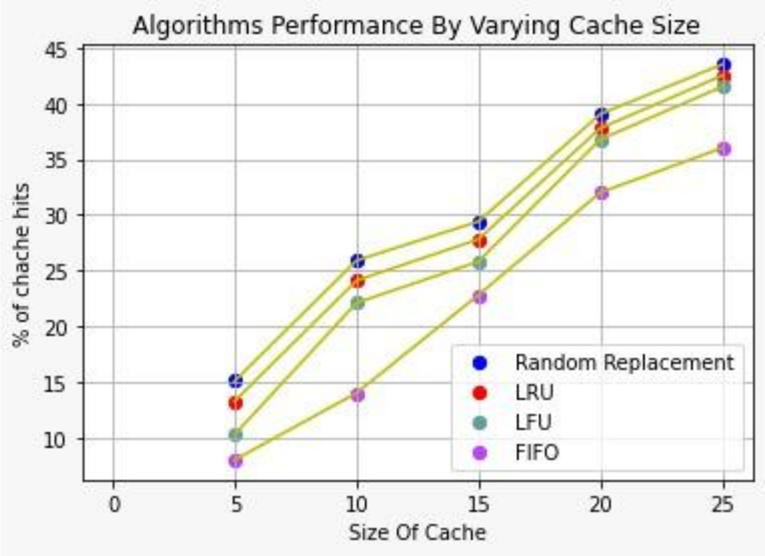


Using RR cache eviction algorithm

Cache Hits - 59/150

Comparison among algorithms in terms of hits and by varying size of the cache structure





Execution Time in Cache Miss and Cache Hit

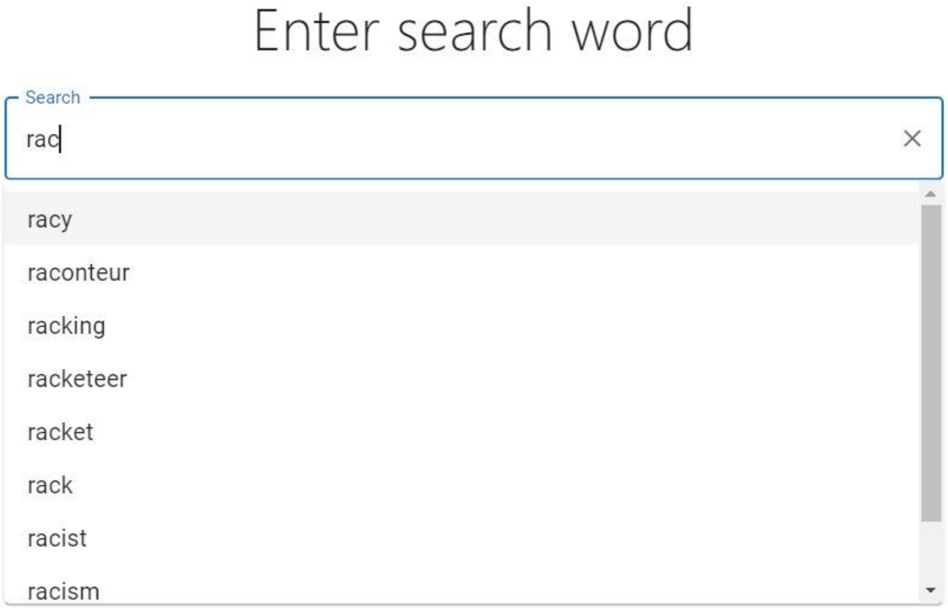
```
PS E:\study\5th sem\IT300\Project> node DBserver.js

This is the Database Dictionary server
Listening at http://localhost:3000

Word Searched = Catastrophe
Cache Miss
Execution time: 166 ms

Word Searched = Catastrophe
Cache Hit
Execution time: 1 ms
```

Comparison of Searching algorithms based on average search time

Database query : 119.10666666666667 ms

FIFO : 78.7 ms
LFU : 74.24666666666667 ms
LRU : 72.79333333333334 ms
RR : 71.55333333333333 ms

Trie : 12.1067 ms



Auto complete feature. While user will be typing a word, at each keypress, a request will be sent to backend using that prefix and in response, an array of words will be received which can be formed from that prefix.

Application:

Enter search word

Search

divide

Submit

Meaning : to split or separate something into two or more parts or groups

Usage-1 : I am in love with books and have to divide my day into house chores, time to write and time to read.

Enter search word

Search

play

Submit

Meaning : act in a manner such as one has fun

Usage-1 : “Can I go out and play, now that the clouds have gone away?”

REFERENCES

- S. K. Pradhan and A. Negi, "An Improved Approach of Dictionary Based Syntactic PR Using Trie," 2018 International Conference on Electronic Systems, Signal Processing and Computing Technologies, 2018, pp. 386-391, doi: 10.1109/ICESC.2014.76.

- S. Jiang, X. Ding, F. Chen, E. Tan, and D. Zhang, “an effective buffercache management scheme to exploit both temporal and spatial locality,”*Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*.

Thank You