TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

MÔN THỰC HÀNH CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

BÁO CÁO ĐỒ ÁN ĐIỂM CỘNG MÔN HỌC

CẤU TRÚC DỮ LIỆU TRIE

Họ và tên: Tô Thành Long

MSSV: 23120143

Lớp: 23CTT2

TP.HCM, Tháng 12/2024

Mục lục

1. MÃ NGUỒN ĐỒ ÁN

1.1. Mô tả cấu trúc Trie

1.1.1. Giới thiệu sơ lược

Trie[1] (còn được gọi là prefix tree hoặc digital tree) là một cấu trúc dữ liệu dạng cây, được sử dụng để lưu trữ và truy xuất các chuỗi một cách hiệu quả, đặc biệt là khi làm việc với các chuỗi có chung tiền tố. Trie rất hữu ích trong nhiều ứng dụng, chẳng hạn như từ điển, tìm kiếm từ tự động, hoặc xử lý văn bản.

Về cấu trúc, Trie bao gồm:

- Node gốc (root): Đây là nút bắt đầu của Trie. Nó không chứa giá trị, chỉ đóng vai trò là điểm khởi đầu.
- Các nút con (children): Mỗi nút con đại diện cho một ký tự trong chuỗi. Các nút con của một nút có thể lưu trữ nhiều ký tự khác nhau.
- Đánh dấu kết thúc chuỗi (end-of-word marker): Một số nút được đánh dấu là nút kết thúc chuỗi, cho biết rằng một chuỗi hoàn chỉnh kết thúc tại đó.

Các thao tác cơ bản trên Trie có thể kể đến:

- Thêm một từ (Insert)
- Xoá một từ (Remove)
- Tìm kiếm một từ (Search)
- Tìm kiếm tiền tố (Prefix search)

1.1.2. Tổ chức mã nguồn

Trong đồ án, cấu trúc Trie và các chức năng của nó được tổ chức bởi hai phần chính là ***struct TrieNode*** và ***class Trie***. Ngoài ra, Trie còn được hỗ trợ ***Cache***[2], bao gồm ***struct CacheNode*** và ***class CacheManager*** để quản lí việc lưu trữ tiền tố đã được tìm kiếm, sẽ được mô tả ở phần sau.

Trong *struct TrieNode* bao gồm:

- ***TrieNode * children*[26]:** Mảng chứa 26 địa chỉ tương ứng với 26 kí tự trong bảng chữ cái. Khi được khởi tạo, 26 ô nhớ này là con trỏ rỗng.
- ***bool isEndOfWord***: Biến đánh dấu kí tự kết thúc của một từ, mặc định là *false*. Khi từ mới được thêm, biến này sẽ có thể được cập nhật.

Trong *class Trie* bao gồm:

- ***TrieNode * root***: Nút gốc của Trie, đóng vai trò là điểm khởi đầu.
- ***CacheManager * cache*:** Lớp quản lí các từ việc thêm, xoá, sửa tiến tố đã được tìm kiếm và các từ được đề xuất liên quan đến tiến tố đó. Đây là tính năng nâng cao và sẽ được mô tả chi tiết hơn ở phần sau.

- ***bool hasWildCard***: Hàm kiểm tra xem tiến tố có chứa kí tự đại diện hay không. Trong đồ án này, kí tự đại diện chỉ bao gồm "." và "[]". Hàm này là một phần của hàm nâng cao $void\ searchByRegex()$, sẽ được mô tả ở phần sau.
- ***void removeHelper***(): Hàm hỗ cho hàm $remove()$ trong việc xoá một từ trong Trie.
- ***void suggestHelper***(): Hàm hỗ trợ cho hàm $suggest()$ trong việc đề xuất các từ dựa trên tiến tố cho trước.
- ***TrieNode * searchPrefix***(): Hàm trả về con trỏ đại diện cho kí tự cuối cùng của tiến tố cho trước. Hàm này phục vụ cho hàm $vector < string > suggest()$.
- ***void fuzzySearchHelper*()**: Hàm hỗ trợ cho hàm $fuzzySearch()$ trong việc đề xuất các từ dựa trên một tiến tố không khớp hoàn toàn với từ trong Trie.
- ***void clearTrie*()**: Hàm xoá tất cả các từ trong Trie.
- ***void loadDictionary*()**: Hàm thêm tất cả các từ trong một file vào Trie.
- ***bool isEmpty*()**: Hàm kiểm tra một nhánh con của Trie có rỗng hay không.
- ***void isnert*()**: Hàm thêm một từ vào Trie và đồng thời cập nhật $CacheManager$.
- ***void remove*()**: Hàm xoá một từ khỏi Trie và đồng thời cập nhật $CacheManager$.
- ***vector < string > suggest*()**: Hàm trả về các từ được đề xuất dựa trên tiến tố người dùng nhập.
- ***vector < string > fuzzySearch***(): Hàm trả về các từ được đề xuất dựa trên một tiến tố không khớp hoàn toàn với từ trong Trie.

Đối với phần thực nghiệm, đồ án chứa hai phần chính ***class TrieUnitTests*** nhằm kiểm tra tính chính xác của các chức năng và ***class TriePerformanceTests*** nhằm đo thời gian chạy của các chức năng.

Trong $class\ TrieUnitTests$ bao gồm:

- ***void runAllTests***(): Hàm chạy tất cả các trường hợp kiểm tra.
- ***void testInsertion***(): Hàm kiểm tra thao tác chèn của một số từ vào Trie.
- ***void testSearchWithExistingWord***(): Hàm kiểm tra thao tác tìm kiếm một số từ tồn tại trong Trie.
- ***void testSearchWithNonExistingWord***(): Hàm kiểm tra thao tác tìm kiếm một số từ không tồn tại trong Trie.
- ***void testRemoval***(): Hàm kiểm tra thao tác xoá một số từ trong Trie.
- ***void testEmptiness***(): Hàm kiểm tra xem Trie có rỗng hay không.
- ***void testSuggestNoRegex***(): Hàm kiểm tra thao tác đề xuất với tiến tố không chứa kí tự đại diện.
- ***void testSuggestWithRegex***(): Hàm kiểm tra thao tác đề xuất với tiến tố chứa kí tự đại diện.
- ***void testFuzzySearch*()**: Hàm kiểm tra thao tác đề xuất với tiến tố gần khớp.

Trong $class\ TriePerformanceTests$ bao gồm:

- ***void runAllTest***(): Hàm chạy tất cả các trường hợp để đo.
- ***void testInsertion*()**: Hàm đo thời gian chạy của thao tác chèn trên số lượng từ cho trước.
- ***void testSuggest***(): Hàm đo thời gian chạy của thao tác đề xuất trên số lượng từ cho trước.
- ***void testRemoval***(): Hàm đo thời gian chạy của thao tác xoá trên số lượng từ cho trước.

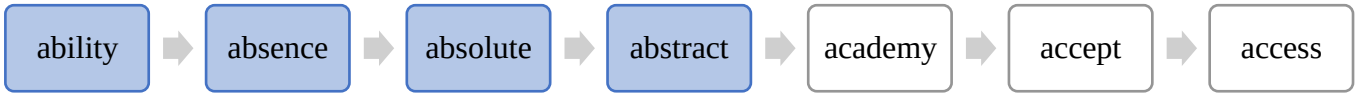
1.2. Mô tả cấu trúc Sorted Array

1.2.1. Giới thiệu sơ lược

Sorted Array (còn được gọi là mảng đã được sắp xếp) là một mảng chứa các từ được sắp xếp theo thứ tự từ điển (lexicographical order). Cấu trúc này sẽ được sử dụng trong đồ án để so sánh với Trie thông qua các chức năng cơ bản là thêm từ, xoá từ và đề xuất từ.

Mảng sắp xếp tăng dần

Quá trình tìm kiếm các từ bắt đầu bằng tiền tố cho trước đồng nghĩa với việc một khoảng (mảng con) mà các từ thuộc khoảng này phải bắt đầu bằng tiền tố. Việc tìm kiếm này có thể thực hiện bằng cách sử dụng **Binary Search** để tìm chặn trên và chặn dưới của khoảng, sau đó liệt kê các từ thuộc khoảng này.



Các từ bắt đầu bằng "ab"

1.2.2. Tổ chức mã nguồn

Cấu trúc Sorted Array và các chức năng của nó được tổ chức bởi $\boldsymbol{class\ SortedArray}$. Trong lớp này bao gồm:

- $\boldsymbol{vector < string > words}$: Mảng chứa các từ đã được sắp xếp theo thứ tự từ điển.
- $\boldsymbol{bool\ startsWith}()$**:** Hàm kiểm tra xem một từ có bắt đầu bằng tiền tố cho trước không.
- $\boldsymbol{pair < int, int > findPrefixRange}()$**:** Hàm trả về khoảng các từ trong mảng bắt đầu bằng tiền tố cho trước.
- $\boldsymbol{bool\ isEmpty}()$**:** Hàm kiểm tra mảng rỗng.
- $\boldsymbol{void\ loadDictionary}()$**:** Hàm thêm tất cả các từ trong file vào Sorted Array.
- $\boldsymbol{void\ insert}()$**:** Hàm thêm một từ vào Sorted Array.
- $\boldsymbol{void\ remove}()$**:** Hàm xoá một từ khỏi SortedArray.
- $\boldsymbol{vector < string > suggest}()$**:** Hàm trả về các từ được đề xuất dựa trên tiền tố người dùng nhập.

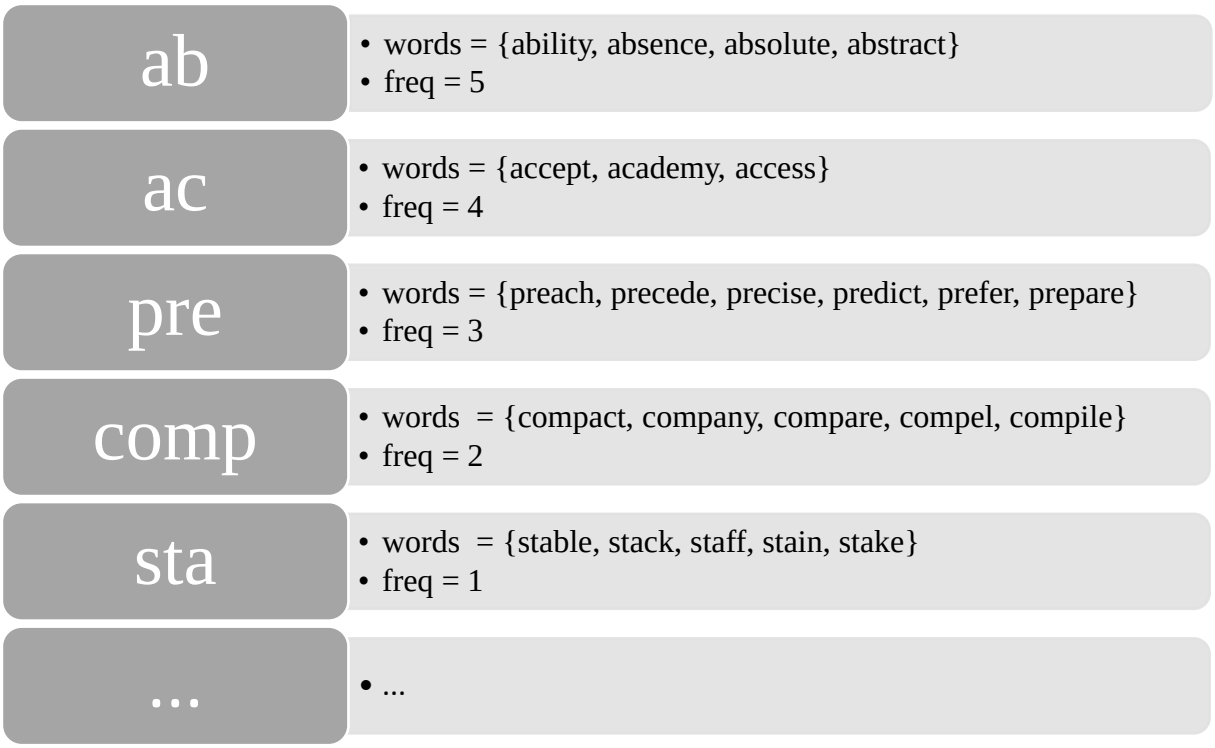
Đối với phần thực nghiệm, cấu trúc này có $\boldsymbol{class\ SortedArrayPerformanceTests}$ nhằm đo thời gian chạy của các chức năng. Tương tự với $TriePerformanceTests$, lớp này cũng bao gồm:

- $\boldsymbol{void\ runAllTest}()$: Hàm chạy tất cả các trường hợp để đo.
- $\boldsymbol{void\ testInsertion}()$**:** Hàm đo thời gian chạy của thao tác chèn trên số lượng từ cho trước.
- $\boldsymbol{void\ testSugges}()$: Hàm đo thời gian chạy của thao tác đề xuất trên số lượng từ cho trước.
- $\boldsymbol{void\ testRemoval}()$: Hàm đo thời gian chạy của thao tác xoá trên số lượng từ cho trước.

1.3. Mô tả cấu trúc Cache Manager

1.3.1. Giới thiệu sơ lược

Đây là cấu trúc được thiết kế để lưu trữ và quản lý các từ được đề xuất liên quan đến tiền tố mà người dùng đã tìm kiếm. Chức năng chính của nó là tối ưu hóa hiệu suất trong việc cung cấp các đề xuất, bằng cách duy trì một bộ nhớ đệm với các quy tắc ưu tiên dựa trên tần suất tìm kiếm.



Minh hoạ cấu trúc Cache được dùng

CacheManager sử dụng một **Hash Map** lưu trữ cặp $(prefix, \{suggestedWord, frequency\})$ để theo dõi tần suất tìm kiếm của tiền tố và một biến $capacity$ để giới hạn dung lượng trong cache. Các chức năng chính của cấu trúc này bao gồm:

- Thêm tiền tố và từ được đề xuất vào cache:
  - Nếu tiền tố đã tồn tại trong cache, tăng tần suất tìm kiếm ($frequency$).
  - Nếu tiền tố chưa tồn tại, thêm tiền tố và các từ được đề xuất vào Hash Map. Trong trường hợp cache vượt quá dung lượng giới hạn, loại bỏ tiền tố có tần suất thấp nhất.
- Lấy danh sách từ được đề xuất:
  - Nếu tiền tố đã tồn tại trong cache, trả về danh sách các từ đã được lưu.
  - Nếu không tồn tại, thực hiện truy vấn trong Trie để lấy danh sách đề xuất, sau đó thêm vào cache.
- Làm mới tiền tố:
  - Trong trường hợp một từ được thêm hoặc xoá trên Trie, điều này sẽ ảnh hưởng đến các từ đề xuất dựa trên tiền tố đã được lưu trong cache. Do đó, khi thêm hoặc xoá một từ, CacheManager sẽ tiến hành xoá các tiền tố có chứa từ này, sau đó tiến hành cập nhật lại khi người dùng tìm kiếm.

Bằng việc tích hợp CacheManager vào trong Trie, các tiền tố có tần suất tìm kiếm cao sẽ được lưu giữ lâu hơn trong cache, đảm bảo rằng các đề xuất phổ biến luôn sẵn sàng mà không cần truy vấn lại từ Trie. Điều này cải thiện hiệu suất cho các trường hợp người dùng liên tục tìm kiếm những tiền tố phổ biến.

1.3.2. Tổ chức mã nguồn

Cấu trúc này được tổ chức bởi hai phần chính là ***struct CacheNode*** và ***class CacheManager.***

Trong *struct CacheNode* bao gồm:

- ***vector < string > suggestions***: Mảng lưu trữ các từ được đề xuất.
- ***int frequency***: Tần suất tiền tố được tìm kiếm.

Trong *class CacheManager* bao gồm:

- ***unordered_map < string, CacheNode > cache***: Hash Map lưu trữ và theo dõi tần suất của tiền tố.
- ***int size***: Kích thước hiện tại của cache.
- ***int capacity***: Dung lượng tối đa của cache.
- ***int getSize***(): Hàm lấy kích thước hiện tại của cache.
- ***int getCapcity***(): Hàm lấy dung lượng tối đa của cache.
- ***void insert***(): Thêm một cặp $(prefix, CacheNode)$ mới.
- ***void update***(): Cập nhật một cặp $(prefix, CacheNode)$ trong cache.
- ***vector < string > get***(): Hàm trả về các từ được đề xuất của tiền tố có trong cache.
- ***void remove***(): Xoá một cặp $(prefix, CacheNode)$ trong cache.
- ***void removeItemByWord***(): Xoá tất cả các cặp $(prefix, CacheNode)$ có chứa từ được yêu cầu.
- ***void evict***(): Hàm xoá cặp $(prefix, CacheNode)$ có tần suất thấp sử dụng nhất.

1.4. Mô tả thuật toán Fuzzy Search

1.4.1. Giới thiệu sơ lược

Thuật toán tìm kiếm mờ (Fuzzy Search)[3] được sử dụng để tìm kiếm các từ gần khớp trong cấu trúc dữ liệu Trie mà có thể sai khác với từ khóa tìm kiếm một khoảng cách chỉnh sửa nhỏ nhất định. Thuật toán này áp dụng khoảng cách Levenshtein để tính số bước tối thiểu (thêm, xóa, hoặc thay thế ký tự) cần thiết để biến một chuỗi thành chuỗi khác.

Ví dụ: Với từ "apple", các từ khoá gần khớp mà người dùng có thể nhập là "apl", "appke", "apples",...

Khoảng cách Levenshtein[4] là một thước đo về độ tương đồng giữa hai chuỗi. Nó được định nghĩa là số lượng tối thiểu các phép chỉnh sửa đơn ký tự cần thiết để biến một chuỗi thành một chuỗi khác. Các phép chỉnh sửa bao gồm:

- Thêm một ký tự vào chuỗi $(insert)$.
- Xóa một ký tự khỏi chuỗi $(delete)$.
- Thay một ký tự này bằng một ký tự khác $(replace)$.

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Hình 1: Công thức tính khoảng các Levenshtein giữa 2 chuỗi a và b[6]

Có thể lấy ví dụ 2 từ "$kitten$" và "$sitting$" để biểu diễn thuật toán này bằng một ma trận 2 chiều như sau:

		k	**i**	**t**	**t**	**e**	**n**
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Ô dưới cùng góc phải chính là khoảng cách Levenshtein giữa 2 từ này, hoặc nói các khác là số bước chỉnh sửa tối thiểu để biến "$kitten$" thành "$sitting$".

1.4.2. Cách thức hoạt động

Cơ chế hoạt động của thuật toán trong hàm $fuzzySearchHelper()$ được mô tả như sau:

- Tính khoảng cách Levenshtein:
  - Dùng mảng quy hoạch động để tính khoảng cách Levenshtein giữa tiền tố của từ trong Trie ($currentWord$) và tiền tố của từ khoá tìm kiếm ($query$).
  - Mỗi ô trong mảng thể hiện số bước cần thiết để chuyển đổi tiền tố của $currentWord$ thành tiền tố của $query$.
- Kiểm tra kết quả:
  - Nếu khoảng cách Levenshtein tại ô cuối cùng của dòng hiện tại trong mảng 2 chiều nhỏ hơn hoặc bằng $maxDistance$ và kí tự hiện tại là kết thúc của một từ, từ này sẽ được lưu.
- Loại bỏ trường hợp:
  - Nếu ô có khoảng cách Levenshtein nhỏ nhất có giá trị lớn hơn $maxDistance$ thì kể từ kí tự này chắc chắn không có kết quả, nên ta sẽ loại bỏ nhánh này.
- Tiếp tục kiểm tra các kí tự tiếp theo:
  - Với nỗi nút (đại diện cho 1 kí tự), ta tiếp tục đệ quy xuống các nút con để tạo thành $currentWord$ mới và tiếp tục tiến hành kiểm tra.

Vì vậy, tìm kiếm mờ trong Trie rất hiệu quả vì chỉ cần duyệt các nhánh có khả năng chứa kết quả và cung cấp một thước đo chính xác về độ tương đồng giữa các chuỗi.

1.5. Mô tả thuật toán Search By Regex

1.5.1. Giới thiệu sơ lược

Biểu thức chính quy (Regular Expression)[5], thường được gọi tắt là $regex$, là một chuỗi các ký tự đặc biệt định nghĩa một mẫu tìm kiếm. Nó là một công cụ cực kỳ mạnh mẽ để làm việc với chuỗi, cho phép:

- Tìm kiếm sự xuất hiện của một mẫu cụ thể trong một chuỗi lớn.
- Thay thế các phần của chuỗi khớp với một mẫu bằng một chuỗi khác.
- Kiểm tra xem một chuỗi có tuân theo một định dạng nhất định hay không.
- Trích xuất các phần cụ thể của chuỗi khớp với một mẫu.

Regex được sử dụng rộng rãi trong nhiều ngôn ngữ lập trình, trình soạn thảo văn bản và công cụ dòng lệnh. $Regex$ hoạt động bằng cách sử dụng các ký tự đại diện và các ký tự thông thường để tạo thành một mẫu. Một công cụ sẽ so sánh mẫu này với chuỗi đầu vào để tìm ra các vị trí khớp. Trong đồ án này, hàm $searchByRegex()$ chỉ sử dụng kí tự đại diện "." (***dấu chấm***), "^" (***dấu mũ***) và "[]" (***dấu ngoặc vuông***). Lưu ý, kí tự *dấu mũ* sẽ luôn đi cùng với *dấu ngoặc vuông* trong đồ án này.

*Ví dụ với regex **"a..le"**, các từ khớp là "apple", "arole"; với regex **"b[aeiou]g"**, các từ khớp là "bag", "beg", "big", "bog", "bug".*

1.5.2. Cách thức hoạt động

Ba kí tự đại diện được sử dụng sẽ thực hiện các chức năng tương ứng như sau:

- Đối với kí tự *dấu chấm*, nó sẽ so khớp với tất cả các kí tự, tức là hàm sẽ thử hết 26 kí tự trong mảng $children$.
- Đối với kí tự *dấu ngoặc vuông*, nó sẽ so khớp với từng kí tự bên trong dấu ngoặc vuông.
- Đối với kí tự *dấu mũ*, nó sẽ loại trừ các kí tự bên trong ngoặc vuông, tức là so khớp với các kí tự còn lại trong 26 kí tự.
- Đối với kí tự khác, nó sẽ duyệt như bình thường.

Regex là một công cụ mạnh mẽ để làm việc với chuỗi, và ký tự *dấu chấm, dấu ngoặc vuông* và *dấu mũ* là những thành phần cơ bản của nó. Trong Trie, việc sử dụng 3 kí tự cho phép người dùng thực hiện tìm kiếm một cách linh hoạt và hiệu quả hơn.

1.6. Mô tả giao diện

Các chức năng chính của giao diện trong lớp ***class UI*** bao gồm:

- ***void userMode***(): Hàm chạy giao diện để người dùng tra cứu bằng tiền tố sử dụng Trie và các chức năng nêu trên.
- ***void trieTestMode*****()**: Hàm chạy $TrieUnitTests$ và $TriePerformanceTests$.
- ***void sortedArrayTestMode***(): Hàm chạy $SortedArrayPerformanceTests$.
- ***void run*****()**: Hàm khởi tạo giao diện cho người dùng lựa chọn chạy một trong ba tính năng trên.

1.7. Dữ liệu và thư viện được sử dụng

Đối với từ điển, đồ án sử dụng file $words\_alpha.txt$ với 370104 từ (danh sách các từ tiếng Anh chỉ gồm các chữ Latin) được lấy từ repository https://github.com/dwyl/english-words.

Đối với các thư viện C++, đồ án sử dụng:

- Đối với quá trình xây dựng các cấu trúc:
  - $< iostream >$
  - $< fstream >$
  - $< vector >$
  - $< algorithm >$ (chỉ sử dụng hàm $sort()$ để sắp xếp kết quả trả về theo sai số các từ trong $fuzzySearch()$ và sắp xếp mảng các từ trong $loadDictionary()$ thuộc $class\ SortedArray$).
  - $< unordered\_map >$
  - $< climits >$
  - $< string >$
- Đối với quá trình thực nghiệm và đo đạc:
  - $< cassert >$
  - $< chrono >$
- Đối với quá trình xây dựng giao diện:
  - $< conio.h >$

2. PHÂN TÍCH VÀ THỰC NGHIỆM

2.1. Phân tích thuật toán trên lý thuyết

2.1.1. Độ phức tạp và phép so sánh của hàm $insert()$ với từ có n kí tự

- Đối với cấu trúc Trie, đoạn code này có:
  - Độ phức tạp: $\boldsymbol{O(n)}$
  - Với mỗi kí tự, hàm có 1 phép so sánh $\boldsymbol{if}$ ($!current->children[idx]$), nên tổng số phép so sánh là $\boldsymbol{n}$.

```
void insert(const string& word) {
    TrieNode* current = root;

    for (char c : word) {
        int idx = c - 'a';

        if (!current->children[idx]) {
            current->children[idx] = new TrieNode();
        }

        current = current->children[idx];
    }

    current->isEndOfWord = true;
}
```

Hàm thêm một từ vào Trie

- Đối với cấu trúc Sorted Array, đoạn code này có:
  - Độ phức tạp của hàm $lower\_bound()$ thực hiện tìm kiếm nhị phân là $\boldsymbol{O(log\ H)}$, với H là kích thước mảng, và của hàm $words.insert()$, trong trường hợp tệ nhất là phần tử được chèn ở đầu vector, tức phải dời vector qua phải H lần, là $\boldsymbol{O(H)}$. Do đó, tổng độ phức tạp là $\boldsymbol{O(logH + H)} \approx \boldsymbol{O(H)}$.
  - Số phép so sánh của hàm $lower\_bound()$ tương tự như hàm tìm kiếm nhị phân, tức là bao gồm phép so sánh 2 biên $while\ (left <= right)$ và so sánh từ cần chèn với từ ở một vị trí trong vector $word <= words\_list[mid]$ là $\boldsymbol{2 * \log H}$. Trong hàm $insert()$, trường hợp tệ nhất là phần từ cần chèn nằm ở đầu vector, ta phải dời vector qua bên phải H lần, tức là phải duyệt và sao chép

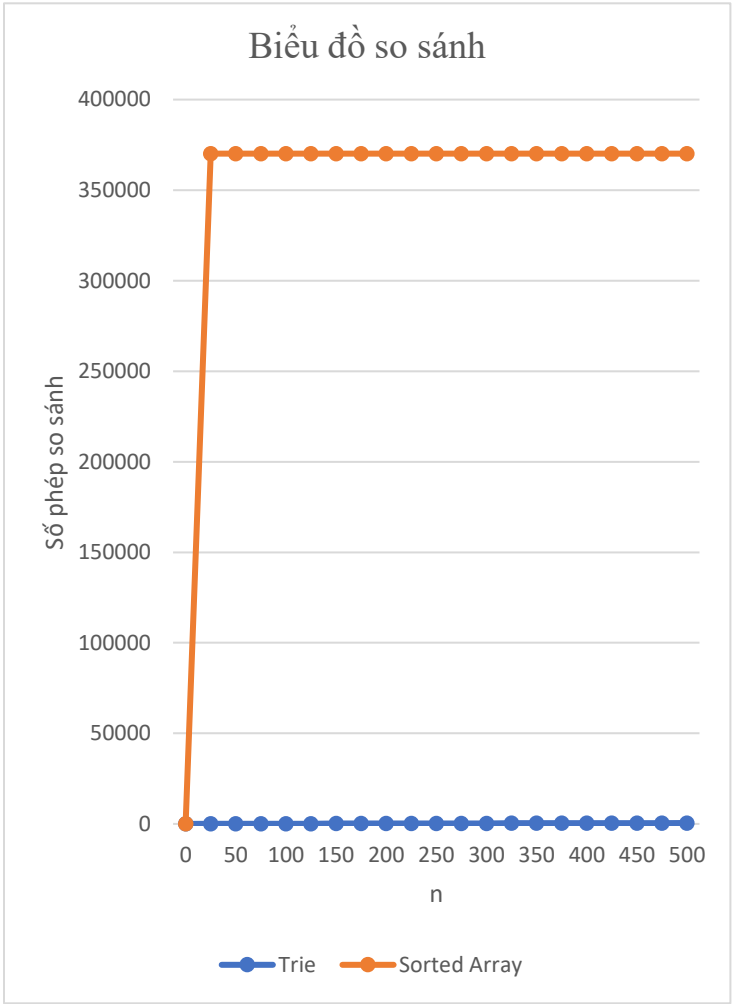
từ phần tử 1 đến $H + 1$, tương ứng với H phép so sánh. Ngoài ra, còn 2 phép so sánh để kiểm tra từ đó đã tồn tại trong vector. Vậy tổng số phép so sánh là $\mathbf{2 * \log H + H + 2.}$

```
void insert(const string& word) {
    int idx = lower_bound(words.begin(), words.end(), word) -
words.begin();

    if (idx < words.size() && words[idx] == word) {
        return;
    }

    words.insert(words.begin() + idx, word);
}
```

Hàm thêm một từ vào Sorted Array

Dưới đây là bảng và biểu đồ thể hiện số phép so sánh với từ có n kí tự

n	Trie	Sorted Array
0	0	21
25	25	370143
50	50	370143
75	75	370143
100	100	370143
125	125	370143
150	150	370143
175	175	370143
200	200	370143
225	225	370143
250	250	370143
275	275	370143
300	300	370143
325	325	370143
350	350	370143
375	375	370143
400	400	370143
425	425	370143
450	450	370143
475	475	370143
500	500	370143



2.1.2. Độ phức tạp và phép so sánh của hàm $remove()$ với từ có n kí tự

- Đối với cấu trúc Trie, đoạn code này có:
  - Với mỗi lần đệ quy, hàm sẽ gọi $isEmpty(current)$ một lần (trừ trường hợp nút cuối cùng thì gọi 2 lần). Hàm này trong trường hợp tệ nhất sẽ duyệt qua hết 26 phần tử trong danh sách nên độ phức tạp là $\boldsymbol{O(K)}$ $\boldsymbol{với}$ $K = 26$. Ngoài ra, khi đệ quy qua n kí tự trong 1 từ, độ phức tạp là $\boldsymbol{O(n)}$. Do đó, tổng độ phức tạp là $\boldsymbol{O(26 * n) \approx O(n)}$.
  - Với mỗi nút khác nút cuối cùng, có 1 phép so sánh $\boldsymbol{if}$ $(!current)$, 1 phép so sánh $\boldsymbol{if}$ $(idx == word.size())$, 26 phép so sánh $\boldsymbol{if}$ $(isEmpty(current))$ (ở dòng sau đệ quy), nên có $\boldsymbol{(n-1) * (1+1+26) = 28 * (n-1)}$ $\boldsymbol{phép\ so\ sánh}$. Với nút cuối cùng, ta có thêm 1 phép so sánh $\boldsymbol{if}$ $(current{-}{>}isEndOfWord)$, 26 phép so sánh $\boldsymbol{if}$ $(isEmpty(current)$ (ngay sau), 1 phép so sánh $!current{-}{>}isEndOfWord$, nên có $\boldsymbol{1+1+26+1+26+1 = 56}$ $\boldsymbol{phép\ so\ sánh}$. Do đó, tổng số phép so sánh là $\boldsymbol{28 * (n-1) + 56 = 28 * n - 27}$.

```
void removeHelper(const string& word, TrieNode* &current, int
idx) {
    if (!current) return;

    if (idx == word.size()) {
        if (current->isEndOfWord) {
            current->isEndOfWord = false;
        }

        if (isEmpty(current)) {
            delete current;
            current = nullptr;
        }

        return;
    }

    int i = word[idx] - 'a';

    removeHelper(word, current->children[i], idx + 1);

    if (isEmpty(current) && !current->isEndOfWord) {
        delete current;
        current = nullptr;
    }
}
```

```
void remove(const string& word) {
    removeHelper(word, root, 0);
}
```

Hàm xoá một từ trong Trie

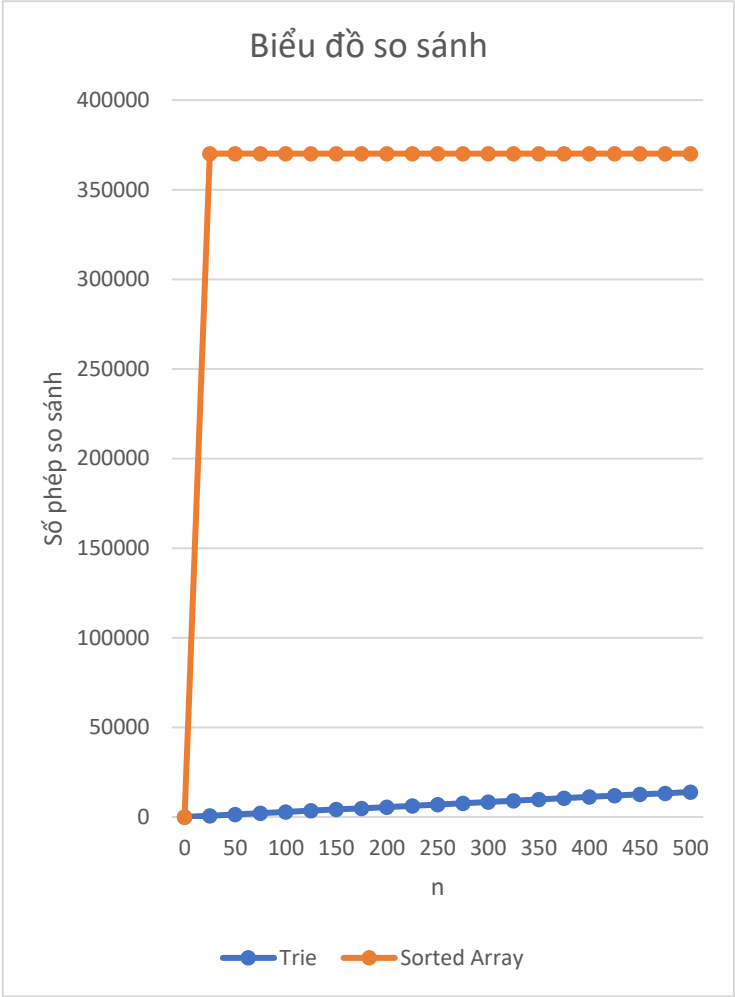
- Đối với cấu trúc Sorted Array, đoạn code này có:
  - Độ phức tạp của hàm $lower\_bound()$ thực hiện tìm kiếm nhị phân là $\boldsymbol{O(log\ H)}$, và của hàm $words.erase()$, trong trường hợp tệ nhất là phần tử được xoá ở đầu vector, tức phải dời vector qua trái H lần, là $\boldsymbol{O(H)}$. Do đó, tổng độ phức tạp là $\boldsymbol{O(logH + H) \approx O(H)}$.
  - Số phép so sánh của hàm $lower\_bound()$ tương tự như hàm tìm kiếm nhị phân, tức là bao gồm phép so sánh 2 biên $while\ (left <= right)$ và so sánh từ cần chèn với từ ở một vị trí trong vector $word <= words\_list[mid]$ là $\boldsymbol{2 * log\ H}$. Trong hàm $erase()$, trường hợp tệ nhất là phần từ cần xoá nằm ở đầu vector, ta phải dời vector qua bên trái n lần, tức là phải duyệt và sao chép từ phần tử 1 đến $H + 1$, tương ứng với $\boldsymbol{H}$ phép so sánh. Ngoài ra, còn 2 phép so sánh để kiểm tra từ đó không tồn tại trong vector. Vậy tổng số phép so sánh là $\boldsymbol{2 * log\ H + H + 2}$.

```
void remove(const string& word) {
    int idx = lower_bound(words.begin(), words.end(), word) -
words.begin();

    if (idx >= words.size() || words[idx] != word) {
        return;
    }

    words.erase(words.begin() + idx);
}
```

Hàm xoá một từ trong Sorted Array

Dưới đây là bảng và biểu đồ thể hiện số phép so sánh với từ có n kí tự.

n	Trie	Sorted Array
0	0	21
25	673	370143
50	1373	370143
75	2073	370143
100	2773	370143
125	3473	370143
150	4173	370143
175	4873	370143
200	5573	370143
225	6273	370143
250	6973	370143
275	7673	370143
300	8373	370143
325	9073	370143
350	9773	370143
375	10473	370143
400	11173	370143
425	11873	370143
450	12573	370143
475	13273	370143
500	13973	370143



2.1.3. Độ phức tạp và phép so sánh của hàm $suggest()$ đề xuất $\boldsymbol{wordLimit}$ từ, với $prefix$ có m kí tự.

- Đối với cấu trúc Trie, đoạn code này có:
  - Độ phức tạp của $searchPrefix()$ là **O(m)** nếu $prefix$ khớp với toàn bộ 1 từ trong Trie. Với $suggestHelper()$, trong trường hợp mảng có đủ các nút, hàm sẽ phải duyệt qua 26 phần tử và tiến hành đệ quy L lần, với L là chiều dài trung bình của 1 từ trong Trie, nên có độ phức tạp là $\boldsymbol{O(26^L)}$. Nhưng trên thực tế, ta thường giới hạn số từ được đề xuất bằng $int\ wordLimit$. Đối với mỗi từ được tìm kiếm, trong trường hợp xấu nhất, ta phải duyệt hết 26 kí tự và với chiều dài L, mỗi từ sẽ mất $O(26 * L)$. Tuy nhiên, ta cần tìm $wordLimit$ từ nên độ phức tạp của $suggestHelper()$ là $\boldsymbol{O(26 * L * wordLimit)}$. Vậy tổng độ phức tạp là $\boldsymbol{O(m + 26 * L * wordLimit)}$.
  - Đối với hàm $searchPrefix()$, có thể xảy ra tối đa $\boldsymbol{2m}$ phép so sánh nếu $prefix$ khớp với toàn bộ 1 từ trong Trie. Tiếp theo, ta có 1 phép so sánh $\boldsymbol{if}\ (currentNode)$, để kiểm tra xem có tồn tại $prefix$ hay không. Đối với hàm $suggestHelper()$, trước tiên ta có 2 phép so sánh $\boldsymbol{if}\ (!currentNode\ ||\ results.size()\ >=\ wordLimit)$ và 1 phép so sánh $\boldsymbol{if}\ (currentNode->$

$isEndOfWord$). Đối với mỗi nút, ta có tối đa 26 phép so sánh giá trị i và kiểm tra nút tồn tại trong vòng lặp. Như vậy, số phép so sánh tối đa trong 1 lần chạy hàm $suggestHelper()$ là $\mathbf{1 + 2 + 26 * 2 = 55}$. Vì hàm này sẽ được đệ quy L lần, với L là chiều dài trung bình của 1 từ trong Trie, và giới hạn với $wordLimit$ từ nên hàm $suggestHelper()$ có số phép so sánh là $\mathbf{55 * L * wordLimit}$. Cuối cùng, hàm $suggest()$, có tổng số phép so sánh là $\mathbf{2m + 55 * L * wordLimit}$.

```
TrieNode* searchPrefix(const string& word) {
    TrieNode* current = root;

    for (auto& c : word) {
        if (!current->children[c - 'a']) {
            return nullptr;
        }

        current = current->children[c - 'a'];
    }

    return current;
}

void suggestHelper(vector<string> &results, TrieNode*
currentNode, string currentWord, int wordLimit) {
    if (!currentNode || results.size() >= wordLimit) return;

    if (currentNode->isEndOfWord) {
        results.push_back(currentWord);
    }

    for (int i=0; i<26; i++) {
        if (currentNode->children[i]) {
            currentWord.push_back('a' + i);
            suggestHelper(results, currentNode->children[i],
currentWord, wordLimit);
            currentWord.pop_back();
        }
    }
}

vector<string> suggest(const string& prefix, int wordLimit = 10)
{
```

```
    vector<string> results;
    TrieNode* currentNode = searchPrefix(prefix);

    if (currentNode) {
        string currentWord = prefix;
        suggestHelper(results, currentNode, currentWord,
wordLimit);
    }

    return results;
}
```

Hàm đề xuất các từ bắt đầu bằng $prefix$ trong Trie

- Đối với cấu trúc Sorted Array, đoạn code này có:
  - Đối với hàm $findPrefixRange()$, vòng lặp while đầu tiên có độ phức tạp $O(\log H)$, với H là kích thước mảng. Trong vòng lặp while thứ 2, có hàm $startsWith()$ để kiểm tra từ có bắt đầu bằng $prefix$ hay không, nên độ phức tạp là $O(m * \log H)$ với m là chiều dài của $prefix$. Do đó, $findPrefixRange()$ có $\boldsymbol{O(\log H + m * \log H) \approx O(m * \log H)}$. Đối với hàm $suggest()$, ta gọi hàm $findPrefixRange()$ sau đó duyệt qua $wordLimit$ phần tử trong vector trả về, nên có tổng độ phức tạp là $\boldsymbol{O(m * \log H + wordLimit)}$.
  - Trong hàm $findPrefixRange()$, vòng lặp while đầu tiên có $\boldsymbol{2 * \log H + 1}$ phép so sánh. Sau đó, ta có 2 phép so sánh để kiểm tra có tồn tại từ nào chưa $prefix$ hay không. Tiếp đến vòng lặp while thứ 2, có hàm $startsWith()$ chứa $1 + 2m + 1 = 2m + 2$ phép so sánh. Do đó, vòng while này có $(1 + 2m + 2) * \log H + 1 = (2m + 3) * \log H + 1$ phép so sánh. Vậy tổng số phép so sánh của hàm $findPrefixRange()$ là $2 * \log H + 1 + 2 + (2m + 3) * \log H + 1 = \boldsymbol{(2m + 5) * \log H + 4}$. Đối với hàm $suggest()$, ta gọi hàm $findPrefixRange()$ sau đó thực hiện vòng lặp for để chọn ra $wordLimit$ từ, tức có thêm $\boldsymbol{(wordLimit + 1) * 2}$ phép so sánh nữa. Vậy tổng số phép so sánh của hàm $suggest()$ là $\boldsymbol{(2m + 5) * \log H + 4 + (wordLimit + 1) * 2}$ phép so sánh.

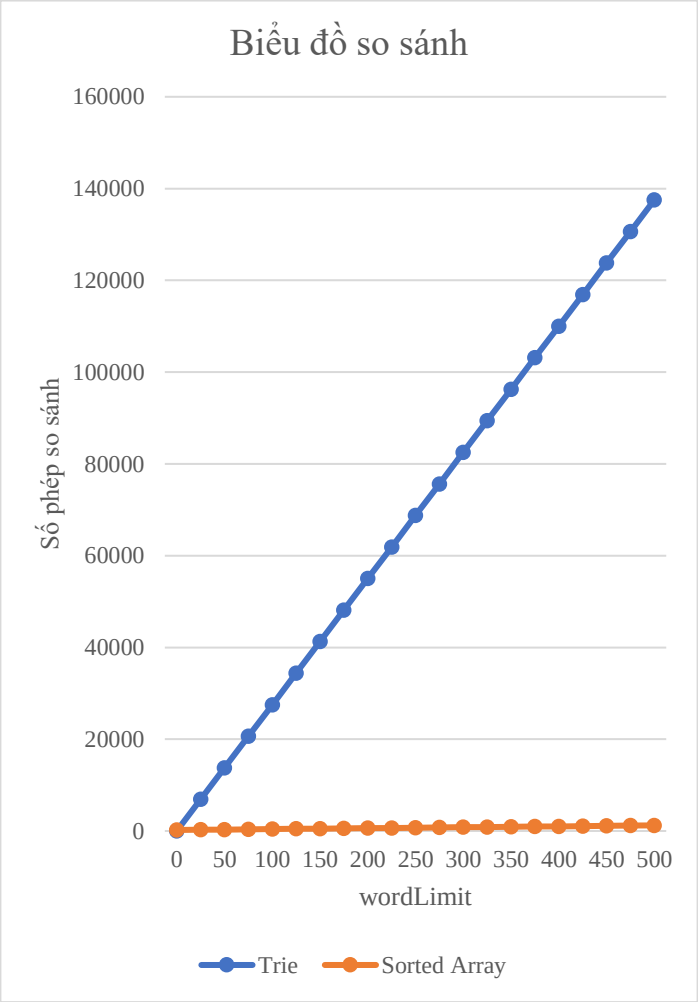
```
bool startsWith(const string &word, const string &prefix) {
    int n = word.size();
    int m = prefix.size();

    if (m > n) {
        return false;
    }

    for (int i = 0; i < m; i++) {
        if (word[i] != prefix[i]) {
            return false;
        }
    }

    return true;
}

pair<int, int> findPrefixRange(const string &prefix) {
    int n = words.size();
    int m = prefix.size();

    int low = 0, high = n - 1, start = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (words[mid] >= prefix) {
            high = mid - 1;
            start = mid;
        } else {
            low = mid + 1;
        }
    }

    if (start == -1 || !startsWith(words[start], prefix)) {
        return {-1, -1};
    }

    low = start;
    high = n - 1;
    int end = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
```

```
        if (startsWith(words[mid], prefix)) {
            low = mid + 1;
            end = mid;
        } else {
            high = mid - 1;
        }
    }

    return {start, end};
}

vector<string> suggest(const string &prefix, int wordLimit = 10)
{
    vector<string> results;
    pair<int, int> range = findPrefixRange(prefix);

    if (range.first == -1) {
        return results;
    }

    for (int i = range.first; i <= range.second && results.size()
< wordLimit; i++) {
        results.push_back(words[i]);
    }

    return results;
}
```

Hàm đề xuất các từ bắt đầu bằng `prefix` trong Sorted Array

Dưới đây là bảng và biểu đồ thể hiện số phép so sánh để đề xuất ***wordLimit*** từ trong từ điển có $\boldsymbol{H = 370104}$ từ, với $prefix$ có $\boldsymbol{m = 3}$ kí tự và một từ trong từ điển có độ dài trung bình $\boldsymbol{L \approx 5}$ kí tự.

wordLimit	Trie	Sorted Array
0	6	215
25	6881	265
50	13756	315
75	20631	365
100	27506	415
125	34381	465
150	41256	515
175	48131	565
200	55006	615
225	61881	665
250	68756	715
275	75631	765
300	82506	815
325	89381	865
350	96256	915
375	103131	965
400	110006	1015
425	116881	1065
450	123756	1115
475	130631	1165
500	137506	1215



2.2. Đánh giá thuật toán trên thực nghiệm

2.2.1. Đánh giá tổng quát

Độ phức tạp của các thao tác được biểu thị dưới bảng sau:

Thao tác	Trie	Sorted Array
Chèn (insert)	$O(n)$	$O(logH + H) \approx O(H)$
Xoá (remove)	$O(26 * n) \approx O(n)$	$O(logH + H) \approx O(H)$
Đề xuất (suggest)	$O(m + 26 * L * wordLimit)$	$O(m * log\ H + wordLimit)$

Số phép so sánh của các thao tác được biểu thị dưới bảng sau:

Thao tác	**Trie**	**Sorted Array**
Chèn (insert)	n	$2 * \log H + H + 2$
Xoá (remove)	$28 * n - 27$	$2 * \log H + H + 2$
Đề xuất (suggest)	$2m + 55 * L * wordLimit$	$(2m + 5) * \log H + 4$ $+ (wordLimit + 1) * 2$

Lưu ý:

- $\boldsymbol{n}$ là số kí tự của một từ được thêm/xoá
- $\boldsymbol{H}$ là kích thước mảng (tức số lượng từ trong từ điển)
- $\boldsymbol{m}$ là chiều dài của $prefix$
- $\boldsymbol{L}$ là số kí tự trung bình của 1 từ trong từ điển
- $\boldsymbol{wordLimit}$ là số lượng từ giới hạn được đề xuất

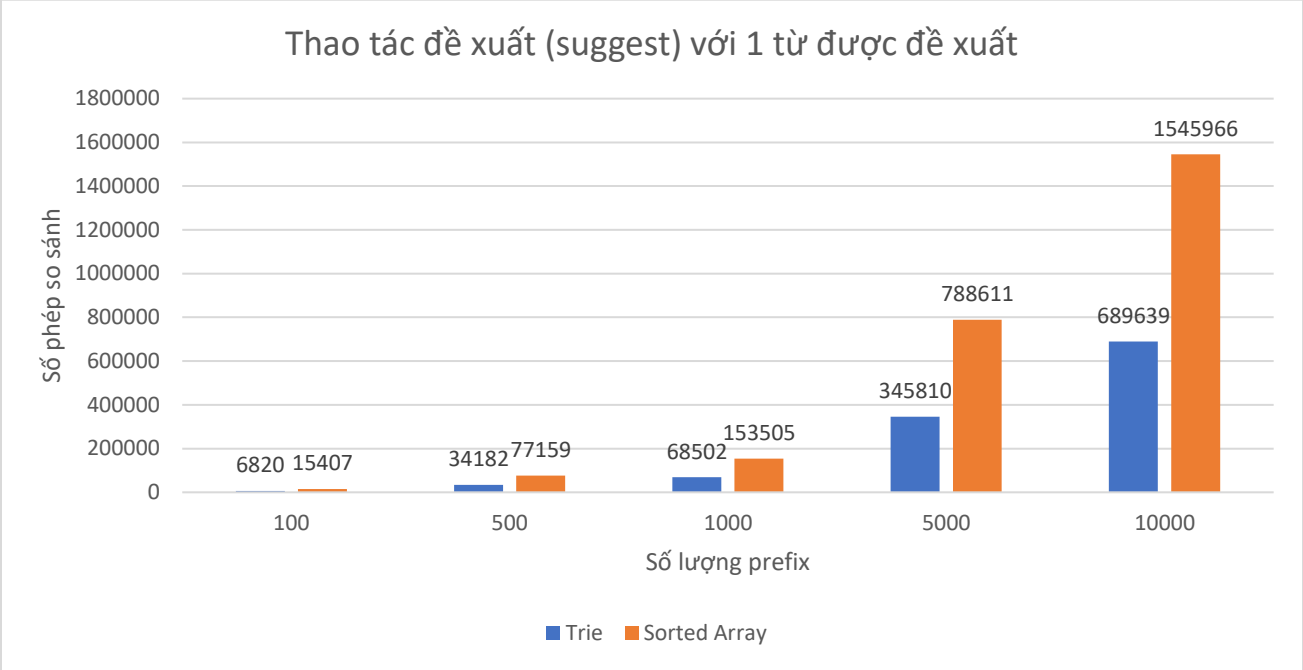
Nhận xét:

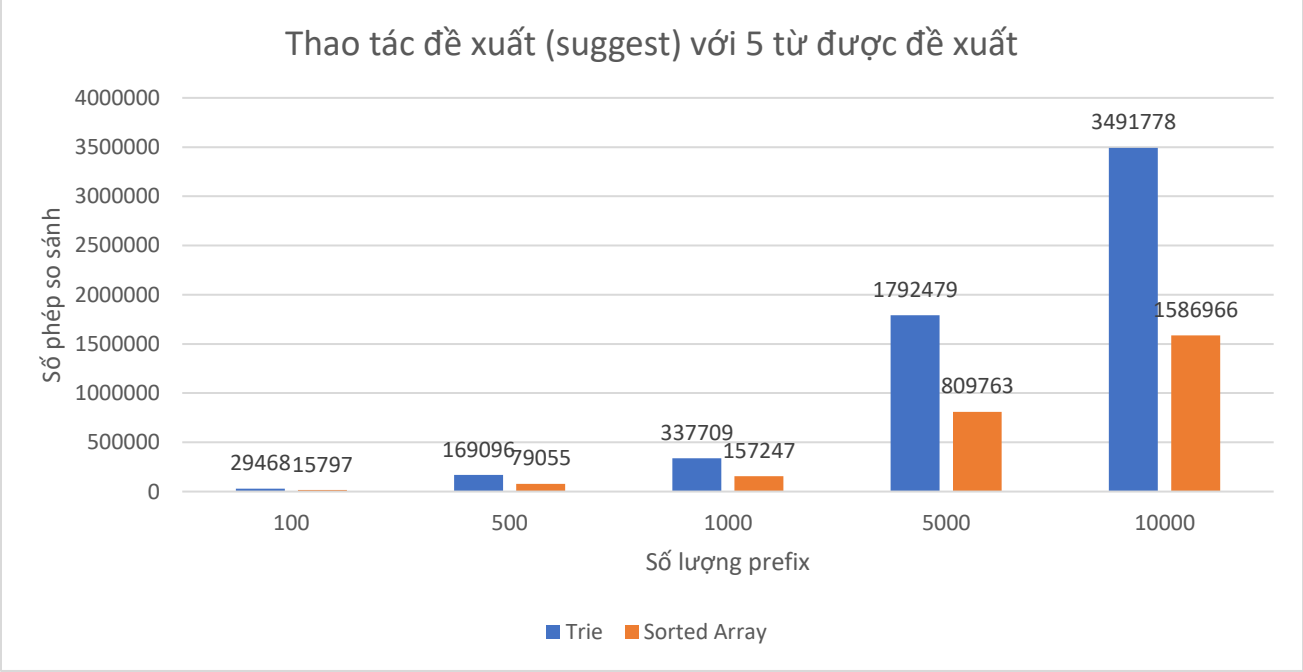
- Thao tác chèn của Trie **nhanh hơn rất nhiều** so với Sorted Array vì giá trị H là rất lớn.
- Thao tác xoá của Trie **nhanh hơn rất nhiều** so với Sorted Array vì giá trị H là rất lớn.
- Thao tác đề xuất của Trie **chậm hơn rất nhiều** so với Sorted Array khi $wordLimit$ càng lớn.

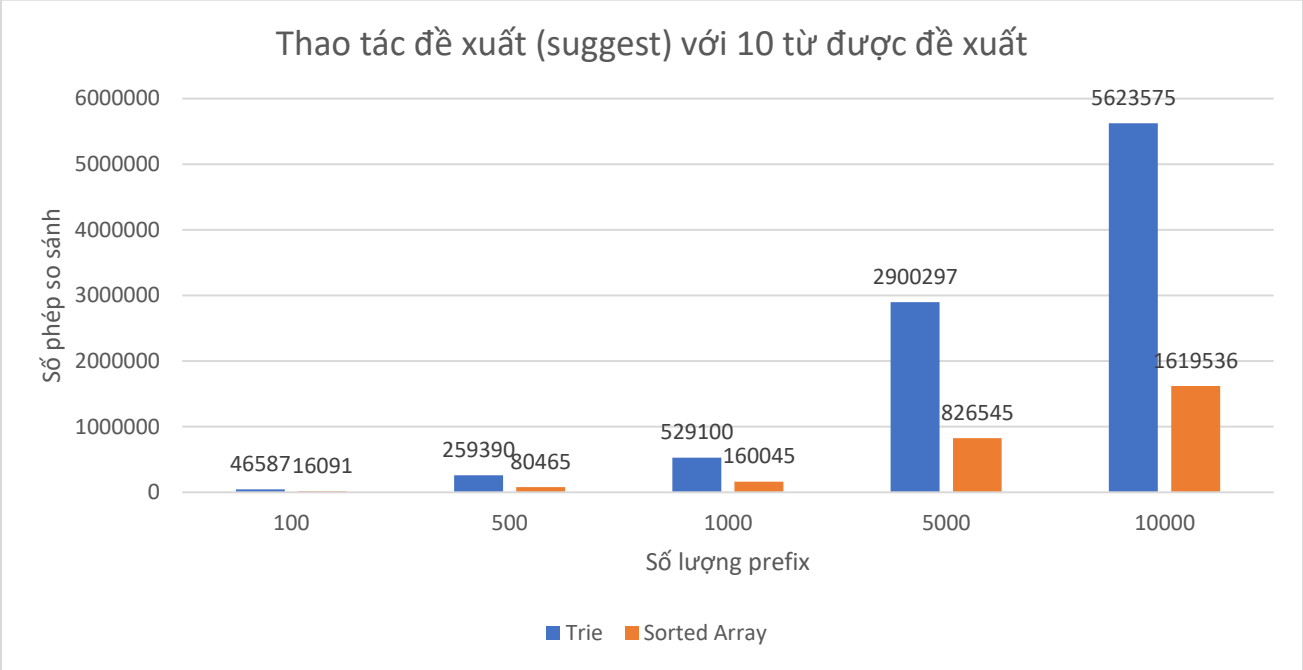
2.2.2. Số liệu thực nghiệm của số phép so sánh

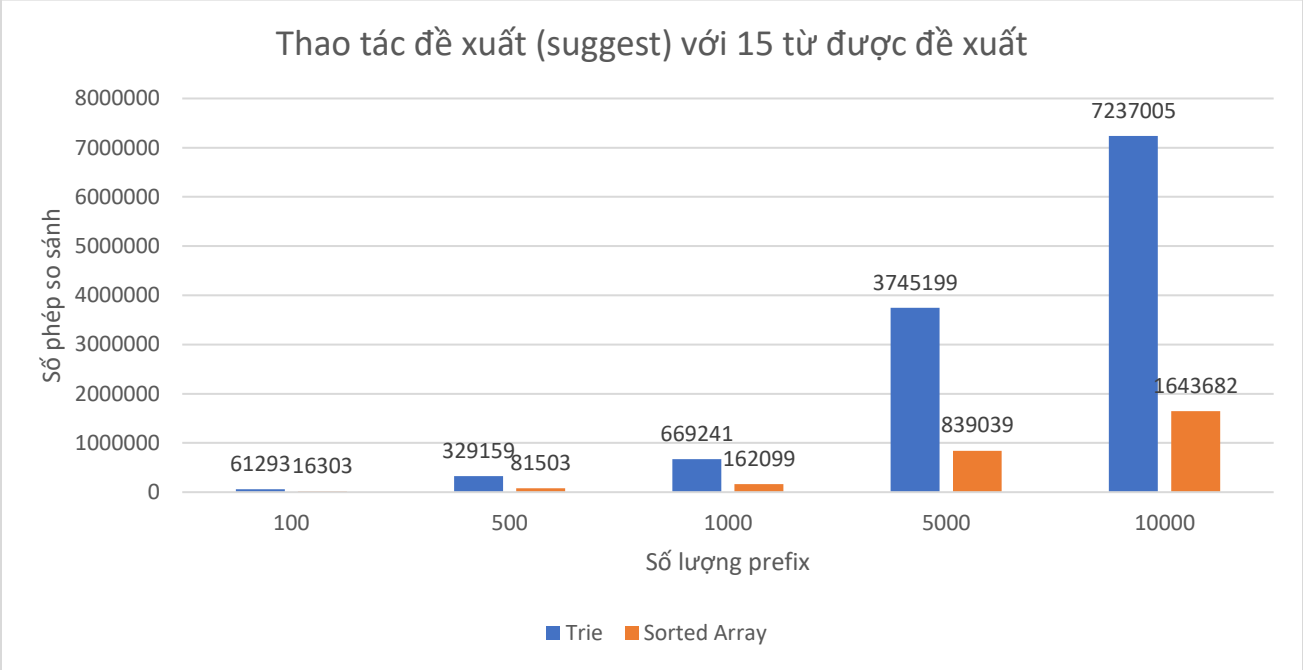
Ta thực hiện chạy hàm với số lượng từ/prefix lần lượt là **100, 500, 1000, 5000, 10000**. Đối với hàm suggest(), ta tiến hành đề xuất số từ lần lượt là **1, 5, 10, 15, 20.**

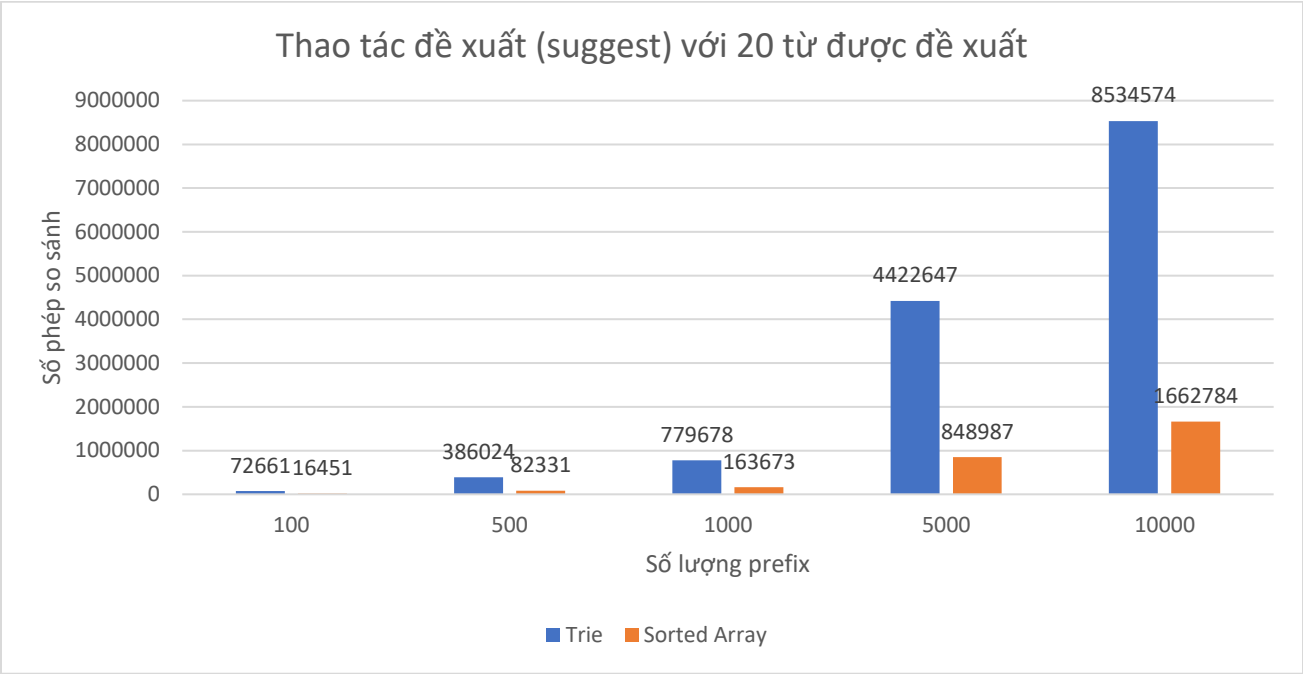
- Với thao tác đề xuất (suggest), ở đây các $prefix$ được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh số phép so sánh giữa Trie với Sorted Array như sau:


Thao tác đề xuất (suggest) với 1 từ được đề xuất
Số phép so sánh
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
6820
15407
34182
77159
68502
153505
345810
788611
689639
1545966
100
500
1000
5000
10000
Số lượng prefix
Trie
Sorted Array


Thao tác đề xuất (suggest) với 5 từ được đề xuất
Số phép so sánh
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
29468
15797
169096
79055
337709
157247
1792479
809763
3491778
1586966
100
500
1000
5000
10000
Số lượng prefix
Trie
Sorted Array


Thao tác đề xuất (suggest) với 10 từ được đề xuất
Số phép so sánh
6000000
5000000
4000000
3000000
2000000
1000000
0
4658
16091
259390
80465
529100
160045
2900297
826545
5623575
1619536
100
500
1000
5000
10000
Số lượng prefix
Trie
Sorted Array

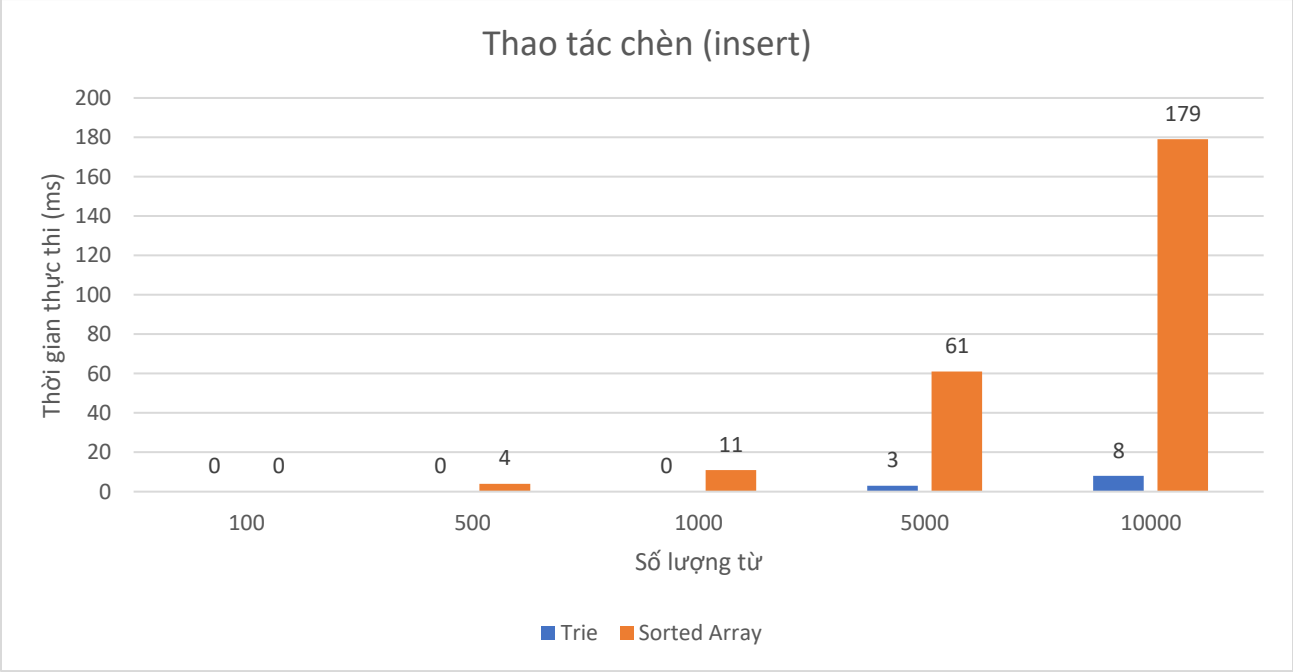

Thao tác đề xuất (suggest) với 15 từ được đề xuất
Số phép so sánh
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
6129
16303
329159
81503
669241
162099
3745199
839039
7237005
1643682
100
500
1000
5000
10000
Số lượng prefix
Trie
Sorted Array



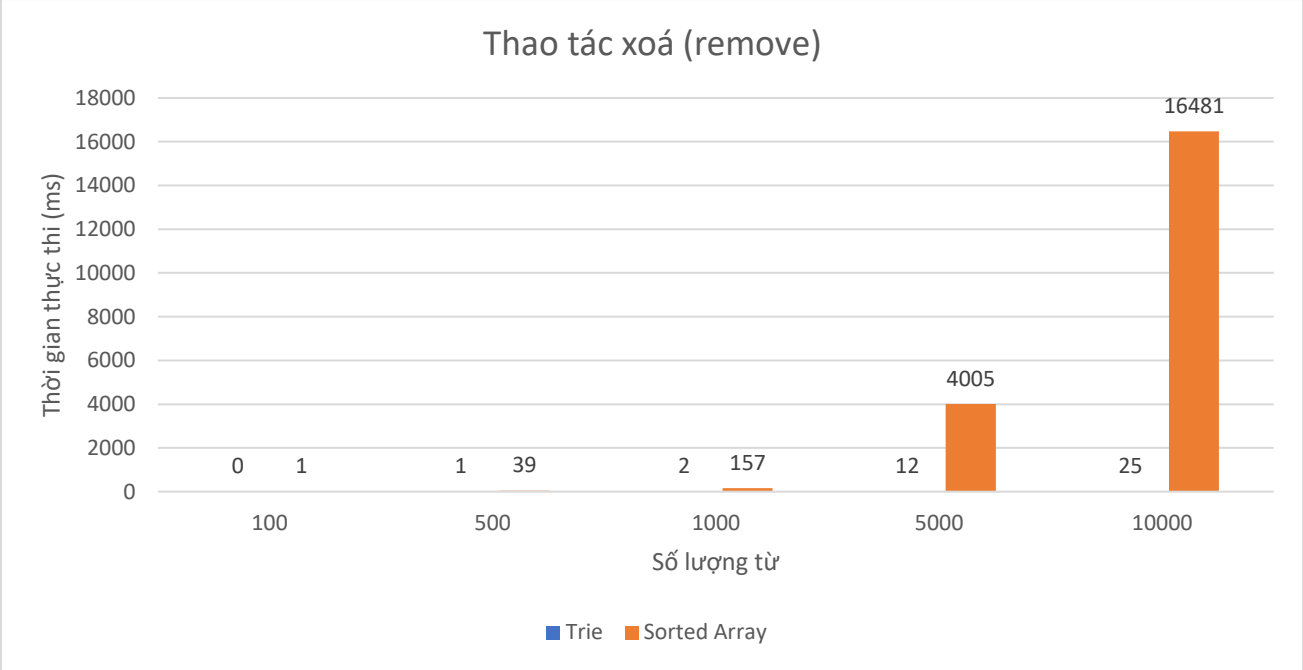
2.2.3. Số liệu thực nghiệm của thời gian thực thi

Để tính chính xác hơn thời gian thực thi của các hàm, ta sẽ thực tiến hành chạy hàm **10 lần** với số lượng từ/prefix lần lượt là **100, 500, 1000, 5000, 10000**. Sau đó ta tính trung bình thời gian chạy 10 lần đó để lấy kết quả thực nghiệm. Đối với hàm suggest(), ta tiến hành đề xuất số từ lần lượt là **1, 5, 10, 15, 20.**

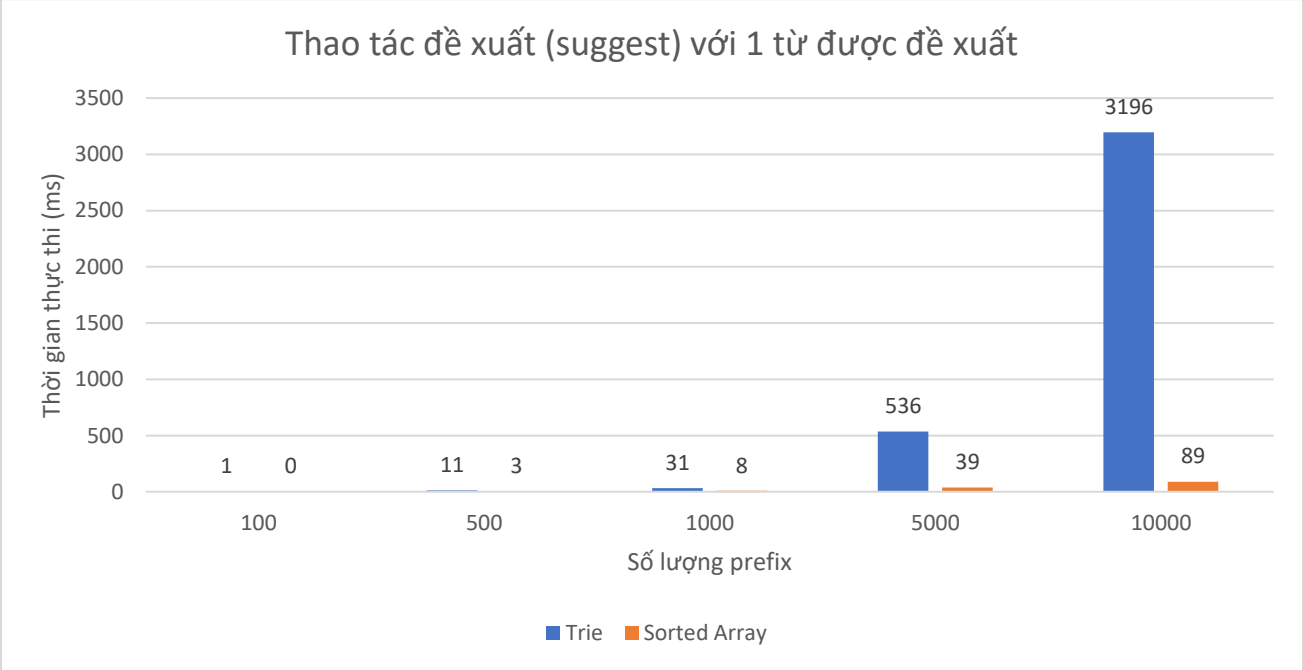
- Với thao tác chèn (insert), biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau (tham khảo):

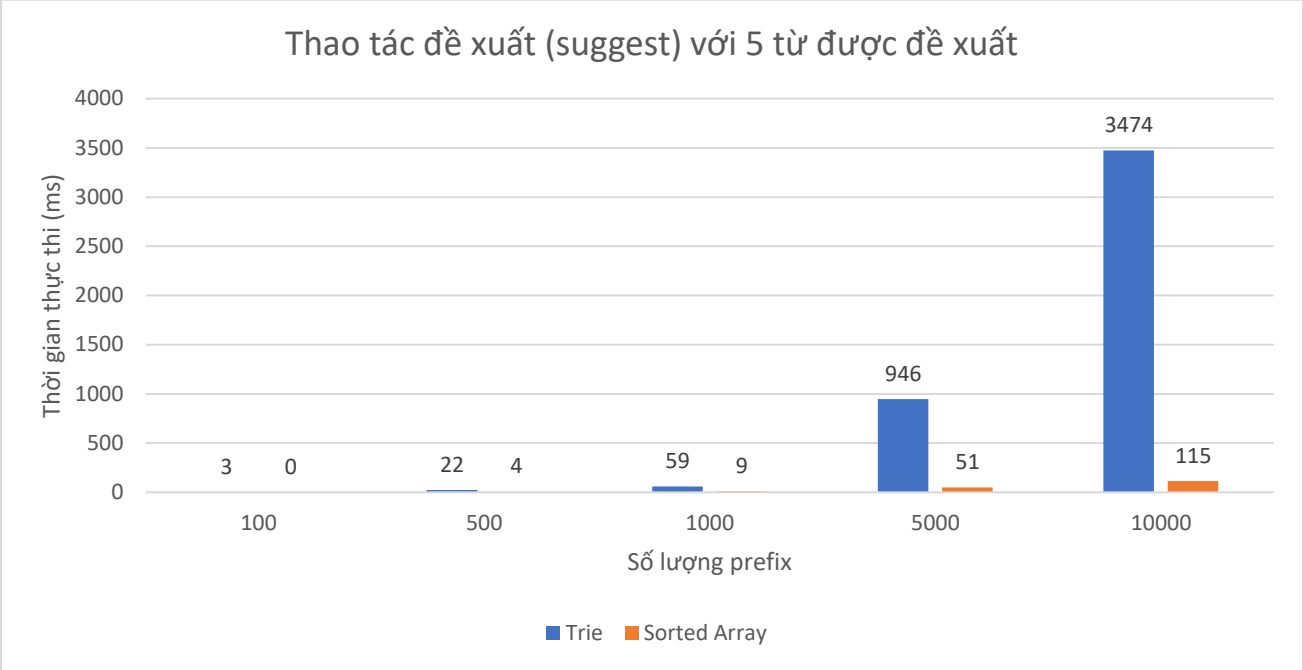


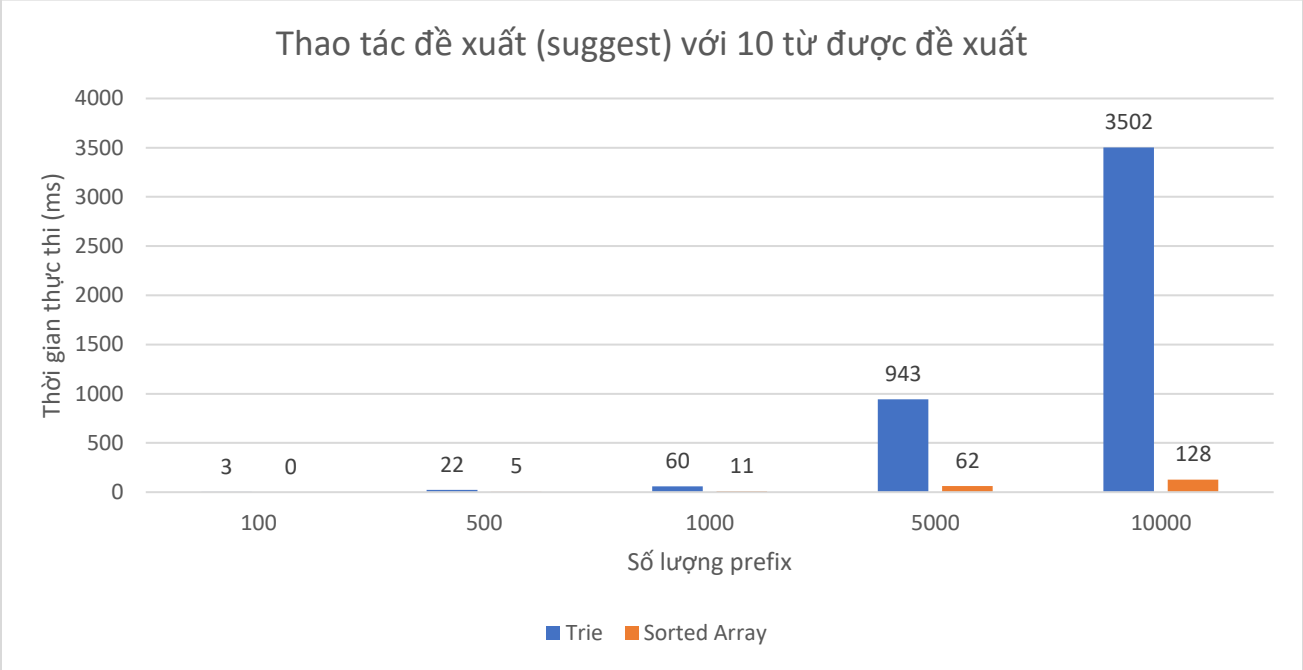
- Với thao tác xoá (remove), biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau (tham khảo):

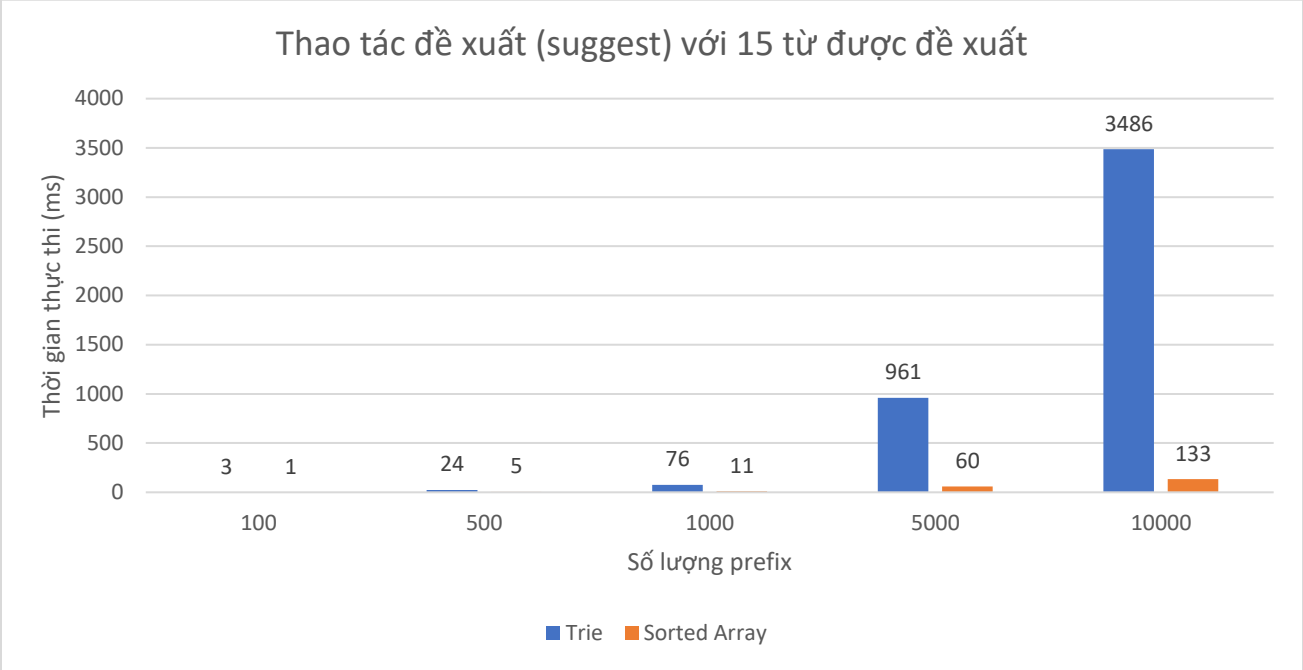


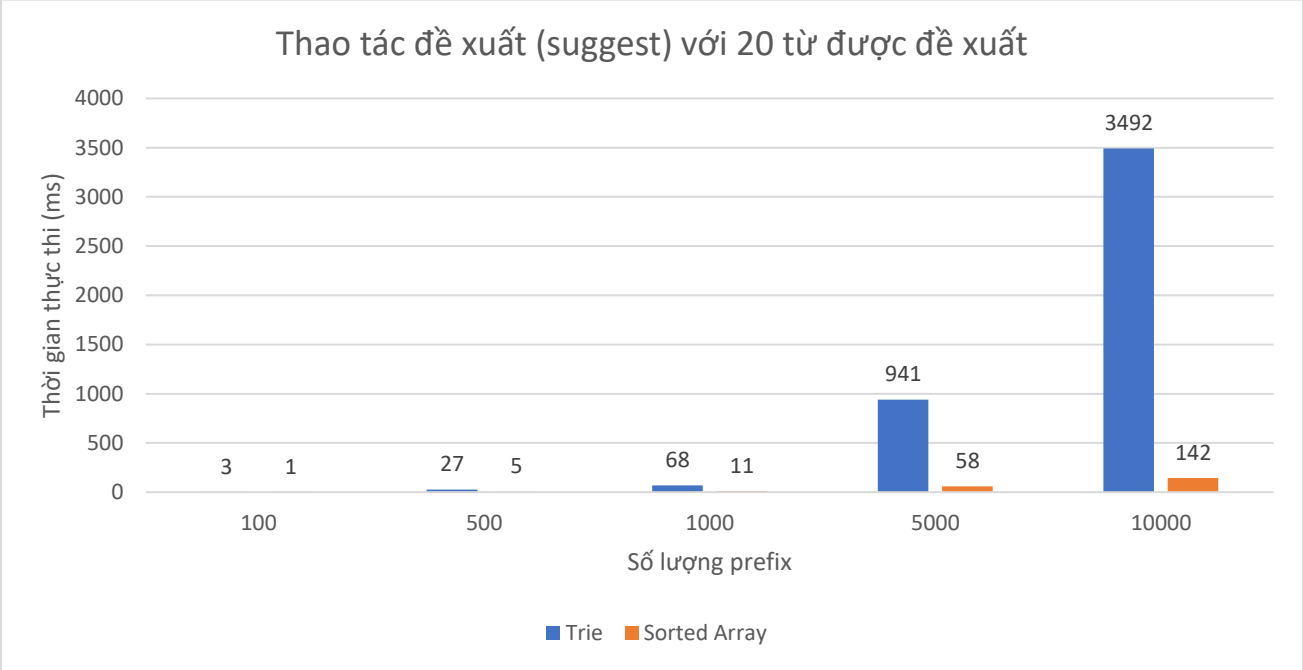
- Với thao tác đề xuất (suggest), ở đây các $prefix$ được sử dụng có độ dài từ 2 đến 5 kí tự, biểu đồ so sánh thời gian thực hiện trung bình giữa Trie với Sorted Array như sau:




Thao tác đề xuất (suggest) với 5 từ được đề xuất
Thời gian thực thi (ms)
4000
3500
3000
2500
2000
1500
1000
500
0
3
0
22
4
59
9
946
51
3474
115
100
500
1000
5000
10000
Số lượng prefix
Trie
Sorted Array


Thao tác đề xuất (suggest) với 10 từ được đề xuất
Thời gian thực thi (ms)
4000
3500
3000
2500
2000
1500
1000
500
0
3
0
22
5
60
11
943
62
3502
128
100
500
1000
5000
10000
Số lượng prefix
Trie
Sorted Array





2.2.4. Kết luận

Từ số liệu thực nghiệm trên, ta có thể dễ dàng thấy rằng quá trình thêm hay xoá một lượng lớn các từ vào Trie nhanh hơn rất nhiều so với Sorted Array. Lí do là vì các ô nhớ, đại diện cho 1 kí tự, chỉ được tạo ra và xoá đi khi cần thiết. Đồng thời việc tạo ra hay xoá đi này cũng không ảnh hưởng đến toàn bộ cấu trúc. Tuy nhiên đối với Sorted Array, việc tạo ra hay xoá đi một từ đồng nghĩa với việc phải di dời và phân bổ lại toàn bộ các phần tử trong mảng con (trong trường hợp xấu nhất là toàn bộ mảng). Do đó, 2 quá trình này tốn rất nhiều thời gian.

Đối với quá trình đề xuất từ, Sorted Array đã thể hiện khả năng vượt trội hơn so với Trie. Lí do là vì việc đề xuất từ trong Sorted Array dựa trên thuật toán Binary Search để tìm kiếm vị trí bắt đầu và kết thúc của các từ bắt đầu $prefix$ trong mảng. Do đó, đối với số lượng từ cần đề xuất lớn thì Sorted Array vẫn duy trì ổn đỉnh hơn. Quá trình này sẽ nhanh hơn và tiết kiệm bộ nhớ hơn là duyệt qua từng kí tự của từ trong Trie, vốn phải sử dụng nhiều stack cho kĩ thuật đệ quy và quay lui.

Như vậy, Trie thường vượt trội hơn khi thao tác với dữ liệu có tính di động, thường xuyên phải cập nhật. Còn Sorted Array vượt trội hơn khi thực hiện chức năng đề xuất lượng lớn từ với bộ dữ liệu cố định.

3. TÀI LIỆU THAM KHẢO

3.1. Tài liệu văn bản

[1] Knuth, Donald (1997). "6.3: Digital Searching". The Art of Computer Programming Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley. p. 492. ISBN 0-201-89685-0.

[2] Alan Jay Smith. 1982. Cache Memories. ACM Comput. Surv. 14, 3 (Sept. 1982), 473–530. https://doi.org/10.1145/356887.356892

[3] Baeza-Yates, R., & Navarro, G. (1998). *Fast approximate string matching in a dictionary*. Proceedings of SPIRE'98, IEEE CS Press, 14–22. http://www.dcc.uchile.cl/~gnavarro/ps/spire98.2.pdf

[4] В. И. Левенштейн (1965). Двоичные коды с исправлением выпадений, вставок и замещений символов [Binary codes capable of correcting deletions, insertions, and reversals]. Доклады Академии Наук СССР (in Russian). 163 (4): 845–848. Appeared in English as: Levenshtein, Vladimir I. (February 1966). "Binary codes capable of correcting deletions, insertions, and reversals". Soviet Physics Doklady. 10 (8): 707–710. https://ui.adsabs.harvard.edu/abs/1966SPhD...10..707L

[5] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). Introduction to automata theory, languages, and computation (3rd ed., pp. 83–122). Pearson.

3.2. Tài liệu ảnh

[6] Ethan Nam. (2019, February 27). Levenshtein distance formula [Image]. In Medium. Retrieved December 20, 2024, from https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0