VNUHCM –

University of Science

Faculty of Information Technology

[CSC1004]

DATA STRUCTURES & ALGORITHMS

GROUP HOMEWORK 1

TRIE RESEARCH

December 19, 2020

INTRODUCTION

CLASS: 19CLC6 – GROUP 10

MEMBERS: 19127422 – NGUYỄN ĐỨC HUY

19127563 – NGUYỄN HOÀNG THÔNG

19127588 – NGUYỄN BẢO TRÂM

WORK IN PROGRESS

Completion level: 10/10. The work has been completed:

- Learn, research, and analyze the complexity of Insert, Deletion, Search of Trie Data Structure

Compare the advantages of Trie Data Structure with two existing data structures: Hash Table and Binary Search Tree (AVL Tree)

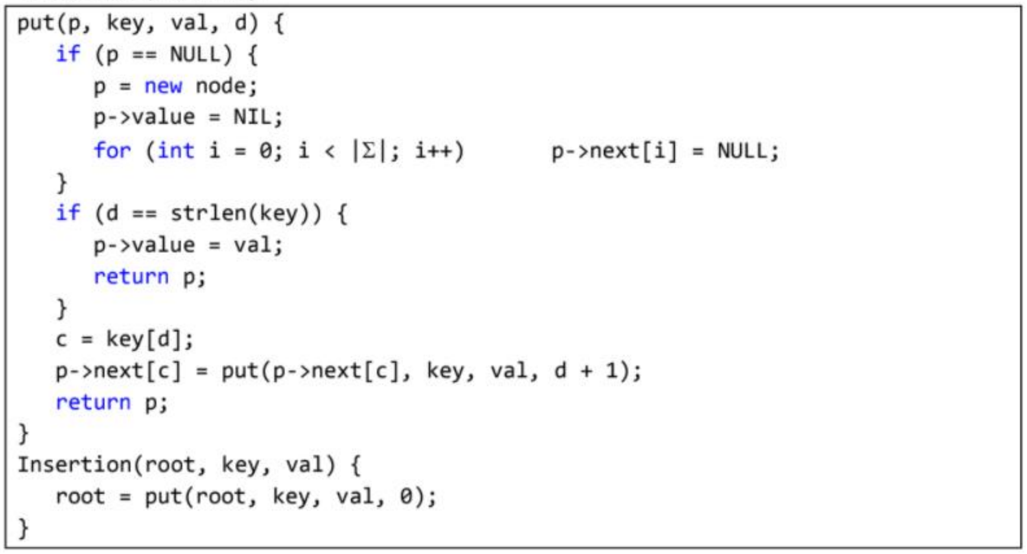
- Build Trie and implement a function that collects words with the same prefix.

Contents

A. RESEARCH

A.1 The time complexity of the Trie operations

A.1.a Adding a word – Insertion


Thêm khóa (Insertion)

```
put(p, key, val, d) {
   if (p == NULL) {
      p = new node;
      p->value = NIL;
      for (int i = 0; i < |Σ|; i++)       p->next[i] = NULL;
   }
   if (d == strlen(key)) {
      p->value = val;
      return p;
   }
   c = key[d];
   p->next[c] = put(p->next[c], key, val, d + 1);
   return p;
}
Insertion(root, key, val) {
   root = put(root, key, val, 0);
}
```

For *Insertion*, Worst case: `O(26 * k) = O(k)`, it has the same asymptotic complexity bound. Note that this does not change with the number of strings, only with its length. Even when the embedded trees are perfectly balanced, the constant factor decreases, but not the asymptotic complexity because of `O(log(26)∗k)=O(k)`, where k is the length of the string.

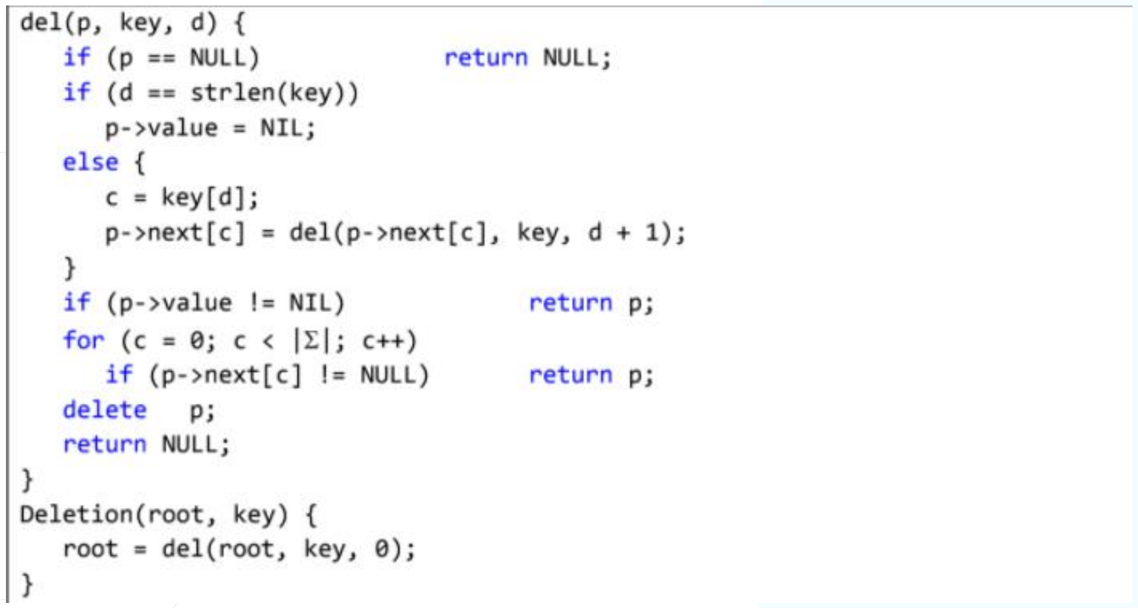
Average case: `O(k)`, in this case, k is an average length of the `string`

Best case: `O(1)`, sush as, your string is just 'a'.

According to this source, for inserting a word of length 'k' we need (k * 26) comparisons. By Applying the Big O notation it becomes O(k) which will be again O(1). Thus, insert operations are performed in constant time

irrespective of the length of the input string (this might look like an understatement, but if we make the length of the input string a worst-case maximum, this sentence holds true).

A.1.b Removing a word – Deletion



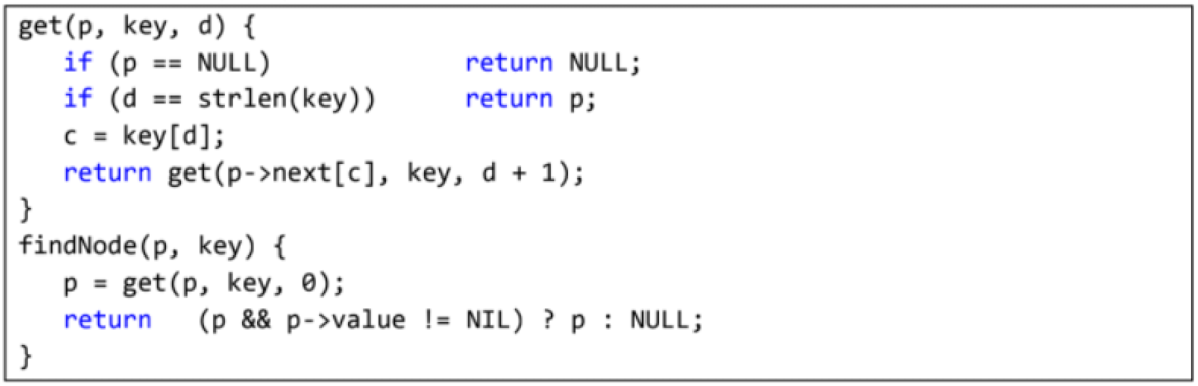
```
del(p, key, d) {
   if (p == NULL)            return NULL;
   if (d == strlen(key))
      p->value = NIL;
   else {
      c = key[d];
      p->next[c] = del(p->next[c], key, d + 1);
   }
   if (p->value != NIL)            return p;
   for (c = 0; c < |Σ|; c++)
      if (p->next[c] != NULL)       return p;
   delete   p;
   return NULL;
}
Deletion(root, key) {
   root = del(root, key, 0);
}
```

If we want to delete words from Trie tree, we first need to find out if the word is in Trie. This step takes at most L searches (where L is the length of the word) if the word is in Trie. The complexity is O (L).

Step 2, if the word is in Trie, we need to check whether the word to be deleted is a prefix of another word. If so, just reset the property to false to deduce the complexity is just O (1). If a word is not a prefix of any word, we must go from the last node to the first word to delete the O (L) complexity.

Therefore, the overall complexity of element deletion is O (L).

A.1.c Searching a word



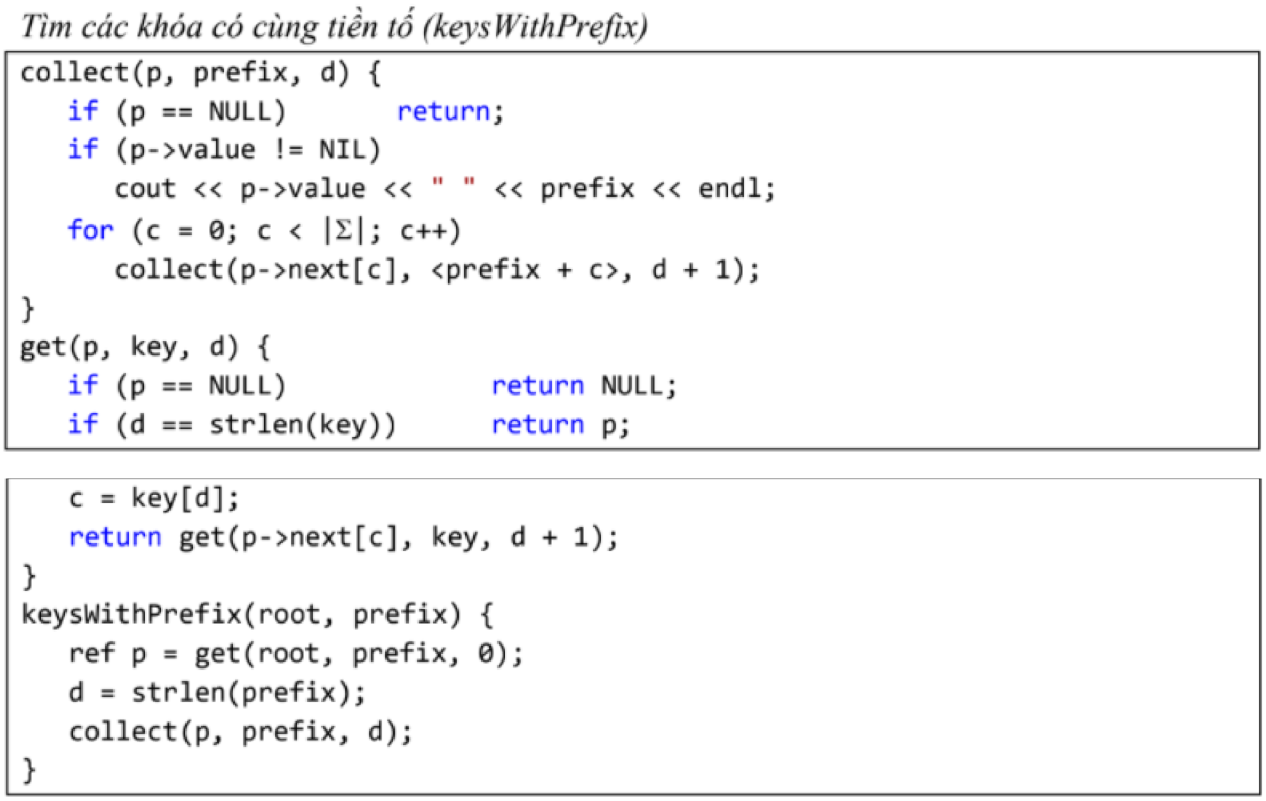
```
get(p, key, d) {
   if (p == NULL)            return NULL;
   if (d == strlen(key))     return p;
   c = key[d];
   return get(p->next[c], key, d + 1);
}
findNode(p, key) {
   p = get(p, key, 0);
   return   (p && p->value != NIL) ? p : NULL;
}
```

As you can see for the function `get()` in the photo above, you must find the last node (the last letter of the word) then you return this node, in this function, you must traversal all letter of the word (if each letter of the key in tries) so that you take `3*n - 1` (n is the length of the key) instructions for this function `get`().

In the `findNode` function, the first thing that you find the last node of the word so that take `3*n` (`3*n - 1`) instruction in the get function and 1 assignment in the `findNode`) then you return if the key exists in the tries, so in this function, it takes `3*n + 2`.

To conclude the function for Searching the word in the tries, the complexity for finding the word in the tries depends on the length of the key you want to find. Therefore, the complexity for Searching the key in the tries is O(n) with n is the length of the key.

A.1.d Searching words which has the same prefix with length i.



Tìm các khóa có cùng tiền tố (keysWithPrefix)

```
collect(p, prefix, d) {
   if (p == NULL)       return;
   if (p->value != NIL)
      cout << p->value << " " << prefix << endl;
   for (c = 0; c < |Σ|; c++)
      collect(p->next[c], <prefix + c>, d + 1);
}
get(p, key, d) {
   if (p == NULL)              return NULL;
   if (d == strlen(key))       return p;
   c = key[d];
   return get(p->next[c], key, d + 1);
}
keysWithPrefix(root, prefix) {
   ref p = get(root, prefix, 0);
   d = strlen(prefix);
   collect(p, prefix, d);
}
```

Reporting all words beginning with that prefix requires you to do a complete search on the trie, which will take time O(n), where n is the total number of nodes in the trie.

The challenge here is that searching for all words in a trie starting with some prefix might require you to search a number of nodes unrelated to the length of the prefix. It depends on what nodes are in the trie.

According to this source, the function `get()` has 3 * n – 1 (n is the length of the key) instructions; the function `collect()` has (2*n – 2*d + 1)(n - d) (n is the length of the key, d is the length of the prefix) instructions and the function `keysWithPrefix()` has f(get()) + f(collect()) + 1.

Finding all keys with a common prefix. We have to traverse all keys in hash table, which can be O(n) (n is the number of keys inserted). However, trie takes *O(k)* (k is the length of the prefix).

A.2 The advantages of Trie comparing to other data structures

Trie	AVL Tree	Hash Table
Predictable O(k) lookup time where k is the size of the key. Lookup can take less than k time if it's not there	the key values stored in the Tree are of basic data types	Your keys need not have any special structure
Looking up keys is faster. Looking up a key of length m takes worst-case O(m) time.	Hence in the worst case, a BST takes O(m log n) time. Moreover, in the worst case log(n) will approach m.	in the worst case, the search complexity is O(n)
Easily print all words in alphabetical order which is not easily possible with hashing	This traversal can be used both in the insertion and in the searching function	HashTable totally disturbs the lexographical order
Tries are more space-efficient when they contain a large number of short keys, because nodes are shared between keys with common initial subsequences.	The space complexity of an AVL tree is O(n) in both the average and the worst case.	More space-efficient than the obvious linked trie structure
There is no need to provide a hash function or to change hash functions as more keys are added to a trie.		Your system will already have a nice well – optimized implementation, faster than tries for most purposes.(with a good hash function)
The number of internal nodes from root to leaf equals the length of the key. Balancing the tree is therefore no concern	After each modification in the AVL tree, we must rebalance the tree	

B. PROGRAMMING

```
const int ALPHABET_SIZE = 26;

struct NODE {
    struct NODE* children[ALPHABET_SIZE];
    char key ;
    bool isEndOfWord;
};
```

Tries. [from retrieval, but pronounced "try"]

- Store characters in nodes (not keys).
- Each node has R children, one for each possible letter in the alphabet, R = 26 (letters)

B. 1 Implement and build a *Trie*

```
void insert(NODE* &proot, string key) {
    NODE* p = proot;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - 'a';
        if (p->children[index] == NULL) {
            p->children[index] = createNODE(key[i]);
        }

        p = p->children[index];
    }

    p->isEndOfWord = true;
}
```

Follow links corresponding to each character in the key.

- Encounter a null link: **create a new node**.
- Encounter the last character of the key: **set value in that node**.

B. 2 Implement an autocomplete feature

```
NODE* search(NODE* proot, string key) {
    NODE* p = proot;

    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';

        if (p->children[index] == NULL)
            return NULL;

        p = p->children[index];
    }

    if (p)
        return p;
    else
        return NULL;
}
```

The function **`search()`** returns the cursor position on the keyword if found, but otherwise returns NULL.

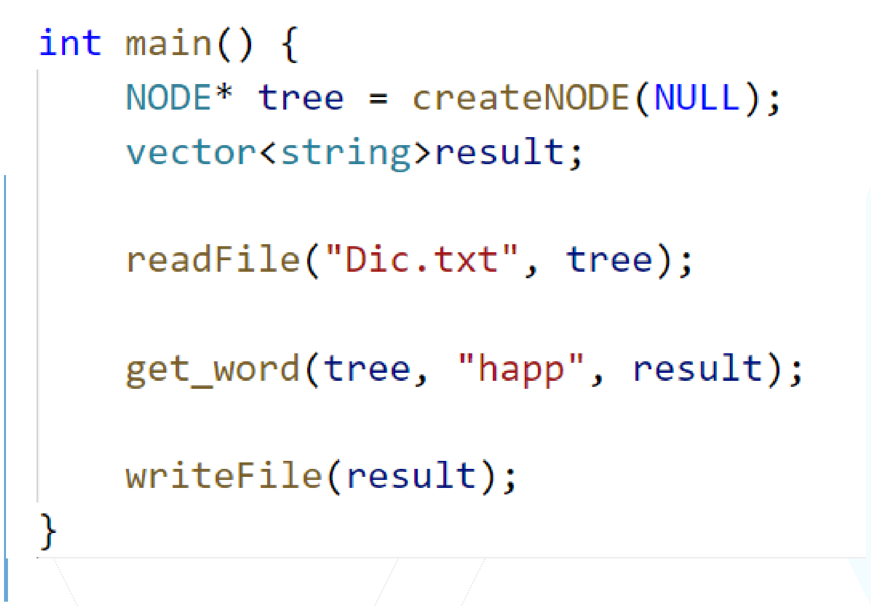
Use the function **`collect()`** to collect words with the same word prefix as the keyword.

Then, we browse the tree according to the added keyword according to the following rule: key + "X", where X is the letters in the alphabet X = {a, b, ..., x, y, z}. And store the found words into a vector.

Finally, the function **`get_word()`** will call **`search()`** to determine the location of the key, and then use **`collect()`** to collect all the words with the same word prefix as the key.

```
void collect(NODE* p, string key,vector<string>&result) {
    if (p == NULL) return;
    if (p->key != NULL && p->isEndOfWord)
        result.push_back(key);
    for (int c = 0; c < 26; c++) {
        char temp = 'a' + c;
        collect(p->children[c], key + temp, result);
    }
}
void get_word(NODE* proot, string key, vector<string>& result) {
    NODE* p = search(proot, key);
    collect(p, key, result);
}
```

EXAMPLE

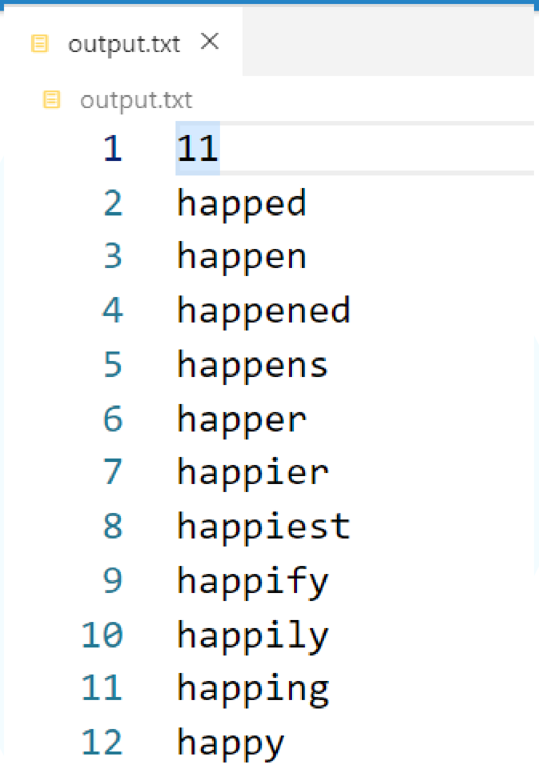


```cpp
int main() {
    NODE* tree = createNODE(NULL);
    vector<string>result;

    readFile("Dic.txt", tree);

    get_word(tree, "happ", result);

    writeFile(result);
}
```



```
11
happed
happen
happened
happens
happer
happier
happiest
happify
happily
happing
happy
```

REFERENCES

[1] Ternary Search Trees by Jon Bentley and Bob Sedgewick, April 01, 1998

[2] Fast Algorithms for Sorting and Searching Strings – Jon L. Bentley & Robert Sedgewick

[3] Laboratory Module D TRIE TREES

[4] Lecture Notes on Tries – 15 -122: Principles of Imperative Computation Frank Pfenning

[5] Trying to Understand Tries. In every installment of this series… | by Vaidehi Joshi | basecs | Medium

[6] Burst Tries: A Fast, Efficient Data Structure for String Keys – Steffen Heinz

[7] Trie vs BST vs HashTable

[8] An Efficient Implementation of Trie Structures

[9] Tries and suffix tries – Ben Langmead