Implementation of Compact Trie in React Autocomplete Input Component to Increase Memory Efficiency

Made Debby Almadea Putri - 13521153[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521153@mahasiswa.itb.ac.id

***Abstract*—Autocomplete Input Component is a commonly used component in building application where the user can get automatic suggestion based on their input. Autocomplete implementation use a tree-based data structure called trie. However, the basic implementation of trie brings new problem as it can take a lot of space. In order to increase memory efficiency of trie data structure, the trie is modified into compact trie. This modification of trie will be discussed in this paper along with its memory usage comparison with basic trie.**

***Keywords*—Autocomplete, Compact Trie, Tree, Trie.**

I. Introduction

While building an application, we often deal with form submission that requires the user to only input values within list of options. This can be dealt by using *HTML Element* called radio button for single choice and checkbox for multiple choice. However, that does not always solve the problem. There are some cases where there were more than 30 options that the user can choose. Using only *HTML Element* can break the interface of our application, as it takes a lot of space, and decrease the user experience, as the user needs to scroll down to see all the options. This is where a new component called *Autocomplete Input Component* came in handy.

Autocomplete, according to Cambridge Dictionary, is a computer program that automatically finishes a word that someone has started to type [1]. While you type in some input area, e.g., form input or google search engine, a list of prediction of words or sentences you wanted to type will be shown. Autocomplete Input Component is a component used to get user input where the user can type the prefix of the options they wanted to choose and the application will show options that matches the prefix the user typed.

The implementation of autocomplete commonly use tree-based data structure called *trie*. One of *trie* implementation, as explained by Jay Wengrow, is by using nodes containing a *HashMap* with characters as keys and other nodes of *trie* as values [2]. While it is simple and fast for list of short words, it can cause potentially dangerous memory issues if the options are long phrases, e.g., name of institution or suggestions in search engine, as it can only store one character in one node.

There are many alternates to solve this problem, such as *burst-trie* (Heinz, Zobel, and Williams 2002) [3], *PATRICIA-trie* (Donald R. Morrison 1968) [4], and *HAT-trie* (Askitis and Sinha 2007) [5]. However, the general idea behind those solution are similar which is using a *compressed* or *compact trie* data structure. Unlike the basic *trie* data structure, each nodes of *compact trie* can store more than one characters, making each nodes at least have two child.

In this paper, the *Autocomplete Input Component* will be implemented with a compressed *trie* data structure suggested by Jay Wengrow using *object-oriented programming*. The component will be made using *TypeScript* and JavaScript framework, *React,* with intention of good reusability and customization. The author will also compare the memory usage for both *basic trie* and *compact trie* with the help of chrome developer tools.

II. Basic Theory

A. Tree

Tree is a node-based data structure where each node is connected by an edge. Unlike graphs, tree nodes should not create a circuit. For example, if we have a simple undirected graph $G = (V, E)$ with n vertices, then G is a tree if G is a connected graph with $m = n - 1$ edges [6]. Based on this, the graphs in *Fig.1*, from left to right, can be classified as tree, tree, non-tree, and non-tree.

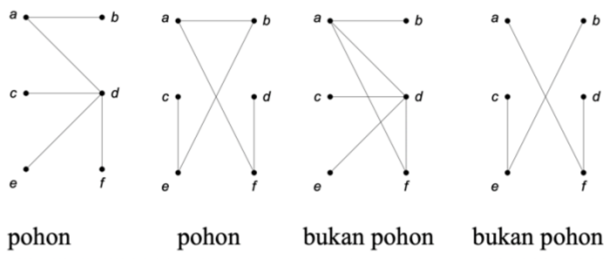


Fig. 1 Example of Tree and Non-Tree
(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf)

B. Rooted Tree

A rooted tree is a tree that represents a hierarchy. Given by its name, a rooted tree has one node that acts as a root and every edges were given a direction, making it a directed graph

[7].

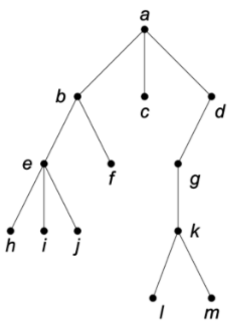


Fig. 2 Example of Rooted Tree
(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/Pohon-2021-Bag2.pdf)

Rooted tree consist of some terminologies to help us analyze its data structure. First, if N is a rooted tree node and N is connected with node M, then N is the parent of M and M is a child of N. For example, in Fig.2, node *h, i, j,* and *k* are the children of node *e* and *e* is the parent of *h, i, j,* and *k.* A parent node can have none to n-many children.

Second, because rooted tree is a graph, it also has a path. Path in rooted tree T with length n is a series of nodes and edges $v_0, e_1, v_1, \dots, v_{n-1}, e_{n-1}, v_n$ that connects one node v_0 to node v_n so $e_1 = (v_0, v_1), \dots, e_{n-1} = (v_{n-1}, v_n)$ is the edges of the rooted tree T [8].

Third, if N is a parent of node P and Q then P and Q are called siblings. For example, in Fig.2, node *e* and *f* are sibling because it has the same parent, *b*, while *f* and *g* are not sibling because node *g* parent is *d*.

Fourth, if T is a rooted tree and S is a node in rooted tree T then we can view node S and its descendants as another rooted tree then S is a subtree of rooted tree T.

Fifth, degree of a node N is the amount of subgraph or children in node N. For example in Fig.2, node *a* has 3 degree while node *c* has 0 degree. Sixth, node with 0 degree are called *leaf*. In Fig. 2, node *h, i, j, f, c, l,* and *m* are *leaf*. Seventh, node with at least 1 degree are called *internal nodes*. Because root tree represents hierarchy, it also has *levels*, *heights* or *depth* as visualized by Fig. 3.

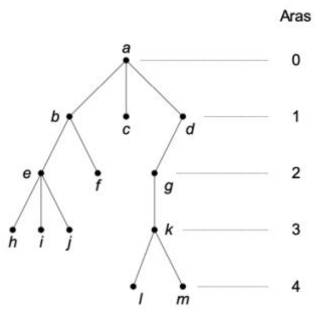


Fig. 3 Levels of Rooted Tree
(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/Pohon-2021-Bag2.pdf)

Based on the maximum degree or children each node have, a rooted tree are divided into two group, binary tree and n-ary tree. Binary tree is a tree which each node has no more than 2 degree. On the other hand, n-ary tree is a tree which each node can have more than 2 degree.

C. Trie

Trie, according to Jay Wengrow, is a tree-based data structure that is usually implemented in text-based structure such as autocomplete [2]. The word *trie* originated from the word *retrieval*. It is also often called as *digital tree* or *prefix tree*. Because a trie is a node-based data structure, it can retrieve collection of strings that match the given prefix faster and more feasible than hash table. Each node of trie contains a character and pointers to the next possible character in a dictionary. The trie data structure also includes an end of string symbol [9].

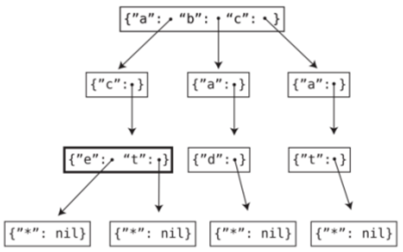


Fig. 4 Implementation of Trie with HashMap
(Source: [2] Wengrow, J., MacDonald, B. (2020). It Doesn't Hurt to Trie. In A common-sense guide to data structures and algorithms: Level up your core programming skills. essay, The Pragmatic Bookshelf.)

A simple implementation of trie is by using a HashMap to store characters and pointers in each node. The key is a character and the value is a pointer to all possible character after the key. In Fig. 4 is a trie which stores the word "ace", "act", "bad", and "cat". As you can see, there are nodes that has key '*' and value nil. This indicates a traversal from the root to those nodes will form a word. This end of word symbol is extremely important especially when there are word that is a substring of other word. For example, in Fig. 5, a node contains keys '*' and 't'. This means traversing to those nodes will form a word "bat" but it can still be traversed to form another word, "batter", which has "bat" as a substring.

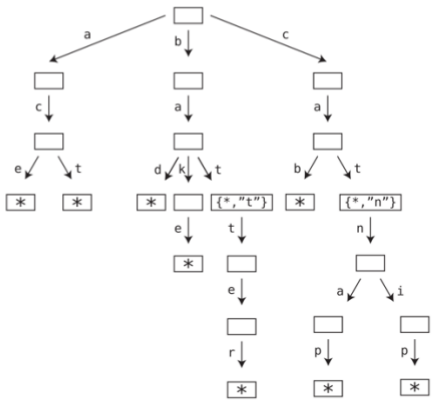


Fig. 5 Implementation of Trie with HashMap
(Source: [2] Wengrow, J., MacDonald, B. (2020). It Doesn't Hurt to Trie. In A common-sense guide to data structures and algorithms: Level up your core programming skills. essay, The Pragmatic Bookshelf.)

D. *Compact Trie*

Compact trie, often called *compressed trie*, *PATRICIA trie* or *radix trie*, is a modified trie which any path that has internal nodes with only one degree or child are compressed into one edge. In short, each node of a *compressed trie* are either a node with 0 degree (leaf) or at least 2 degree [10]. This alternate data structure is proposed because the usual trie data structure will eventually suffer in space as each node can only store up to one character.

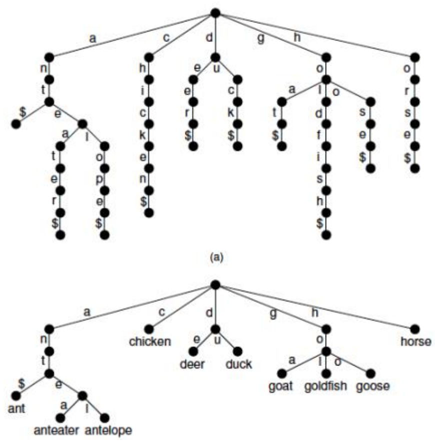


Fig. 6 Difference Between Trie (Top) and Compact Trie (Bottom)
(Source: https://cgi.luddy.indiana.edu/~yye/c343-2019/tries.php)

In Fig. 6, we can see the difference in the number of nodes needed to build the tree between trie and compact trie. Trie required more than 40 nodes to build a trie that only store 10 words, while a compact trie only require 18 nodes. Similar to trie, a compact trie also needs to store end of word symbol. The disadvantage of this data structure is looking up words that match given prefix require additional algorithm that is to traverse each character stored in a node before traversing to the next possible node [10]. Because we can store up to the whole string in one node, the "word" inserted to the *compact trie* can be a *phrases* or even a *sentence*. For consistency, the word "word" will still be used but it does not only reference a word but also a phrases or a sentence.

E. *Memory Terminology*

The size of an object is classified into two types, *shallow size* and *retained size*. This classification is based on the way an object hold its memory. *Shallow size* is the size of memory by the object itself [11]. *Retained size* is the size of memory by the object itself plus all objects referenced by this object [12]. In modern JavaScript, unneeded memory are automatically deallocated by *garbage collection* (GC). However, GC does not deallocate memory of objects referenced by another object that is still needed by the program.

III. Compact Trie Algorithm and Implementation

A. Data Structure

Compact trie will be implemented using JavaScript Map, similar to the one in trie implementation, with a character as key and pointer to other node as value. There is also additional data in each node which is value that contain the string stored in each node. The diagram of *compact trie* data structure are shown in Fig. 7.

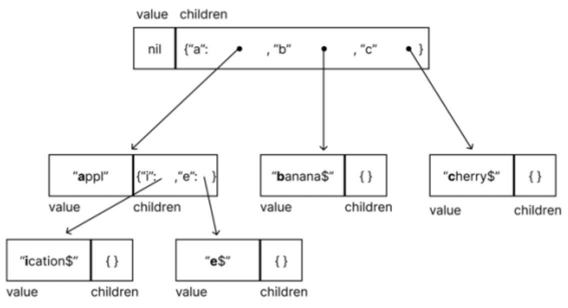
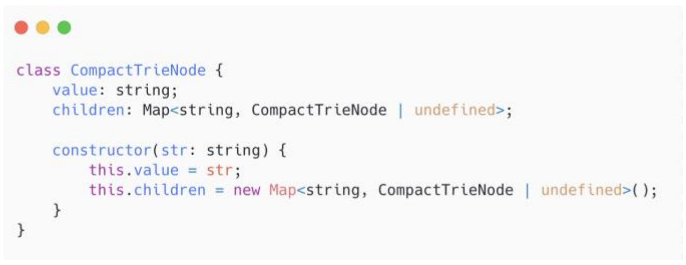


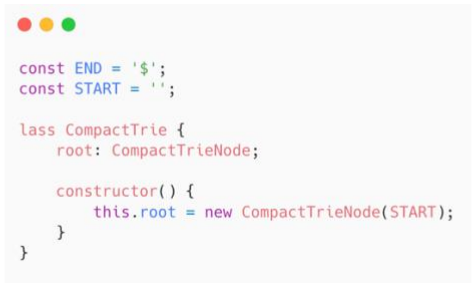
Fig. 7 Data Structure of Compact Trie (Source: Personal Library)

For convenience, the Map for children will be referenced as children map. In Fig.7, the root contains a value nil because there are no words stored in the trie at the beginning. The root also contains a children map with keys 'a', 'b', and 'c' which are the first unique character for all stored word. The children map value is a pointer that points to the child node that store string value beginning with the character stored in their (pointer) associated key. The child node will also have a children map that has the first character of each string value in their children node as keys and their children node as pointer. At the end of every word, there is a symbol '$' that indicates the end of a word.



```
class CompactTrieNode {
    value: string;
    children: Map<string, CompactTrieNode | undefined>;

    constructor(str: string) {
        this.value = str;
        this.children = new Map<string, CompactTrieNode | undefined>();
    }
}
```

Fig. 8 Data Structure of Compact Trie Node (Source: Personal Library)



```
const END = '$';
const START = '';

lass CompactTrie {
    root: CompactTrieNode;

    constructor() {
        this.root = new CompactTrieNode(START);
    }
}
```

Fig. 9 Data Structure of Compact Trie (Source: Personal Library)

B. Word Insertion

The insertion of our *compact trie* is categorized into three cases. The first case is the best case, which is when the words stored in the trie does not have the same first character as the word that will be inserted. For example, we wants to insert a new word in already constructed *trie* in Fig. 7. Before we start to move to the algorithm, we first convert the word to a lowercase so it still will match even when the user inputs a mixed type of lowercase and uppercase. After we convert it to a lowercase letter, then the algorithm of insertion are as follows:

1. Take the first character of the new word and use it as a key to check if there exist a child with that key in the root children map
2. If there is no child then this would be the best case of insertion as no more traversal is needed to insert the new word. Create a new trie node with the whole word as the value and insert a new child node to the root node with key the first character of the new word and value the new trie node that just been made.
3. If there exist a child with the key then we begin to traverse the string value in the child node. While traversing, we compare each character of the new word and the string value of child node. The traversal is terminated when the index is out of range of either the new word or the string value of the child node.
4. If the character at index *i* is not equal for both string, then we move to the next step which is splitting the both string at index *i*. For example, the word "channel" will be inserted to the trie on Fig. 7. The first character, 'c' already exist as a key to a child node in the root node. However, the third character of the new word, 'a' does not match the third character of the value stored in the child node, 'e'. Both string then split at the third character (or index 2) resulting in two string, "ch" and "erry$" for the value in the child node, "ch" and "annel" for the new word. The first substring for both words, "ch", will became the new value of the child node. Two new trie nodes then created with value of the later half of the split string, "erry$" and "annel$" with each has key of the respective first character, "e" and "a". These two nodes then inserted as a new child of the child node "ch". Because this *compact trie* use Map to store the key and not node, there is no need to calculate the order of insertion. In every insertion, the new word is concatenated with the end of word symbol '$'.

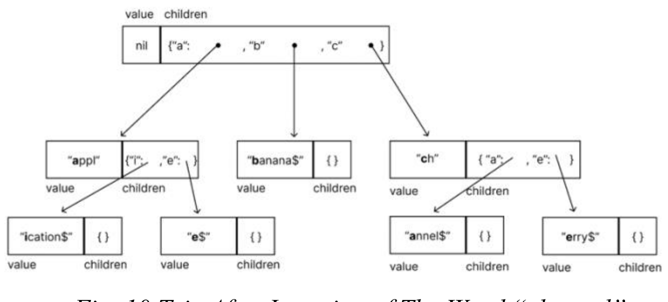


Fig. 10 Trie After Insertion of The Word "channel"
(Source: Personal Library)

5. If the given word has length less than the compared string value and all character at the start index to length of given word – 1 are equals then we do the same thing as step 4, but instead of splitting the given word, the new node has a value of symbol '$' or the end of word symbol. The key will also be the symbol '$'. For example the word "ban" will be inserted to the *trie* in Fig. 10. After we did step 2, we arrived at the child node containing the value of "banana$". Because "ban" is the prefix of "banana$", then the comparison from the starting index 0 to the minimum index possible for the two strings, in this case length of given word – 1, will give equal result. Similar to step 4, the value in child node is split at index of the length of given word, in this case index 3, resulting in substring "anana". Because there are no more character left in "ban", the second node will have the value of "$" as shown in the Fig. 11.

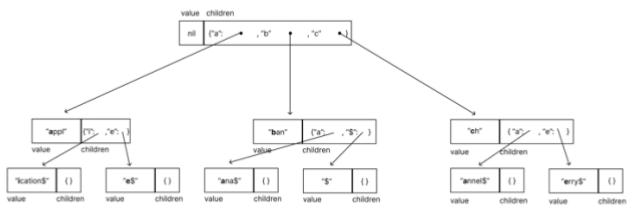


Fig. 11 Trie After Insertion of The Word "ban"
(Source: Personal Library)

6. If the *trie* level is more than 2 and the given word is longer than the value in the child node and the traversal is terminated at index *i* then execute step 1 – 6 except the root node, or the parent node, is the current child node and the starting index is not 0 but index *i*. For example the word "channeling energy" will be inserted to the trie in Fig. 12. After executing step 3, the traversal terminated while there are still characters left in the given word. However we cannot insert a new node to the child node immediately because there is a possibility the child node have children with the same prefix as the given word. Now, to make referencing more simple, the current child node became the "root" node or the parent node and the start index is not 0 but *i*. For example the traversal in node with value "annel" will terminated after letter 'l' so the remaining "ing energy" part of the given word remains unchecked. By stating index *i* as the new index 0 is the same as making "ing energy" as the word that will be inserted. After executing step 2 – 5 with this new rules, the resulting trie is shown at Fig. 13.

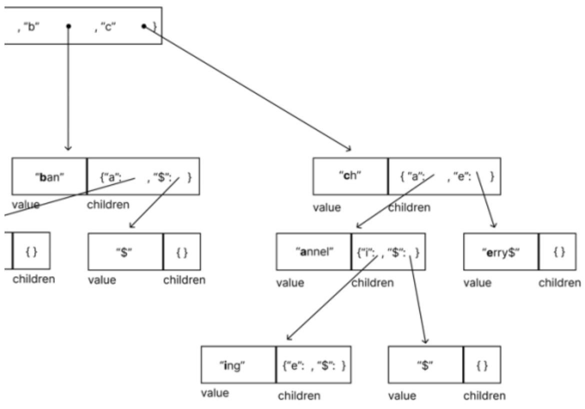


Fig. 12 Trie After Insertion of The Word "channeling"
(Source: Personal Library)

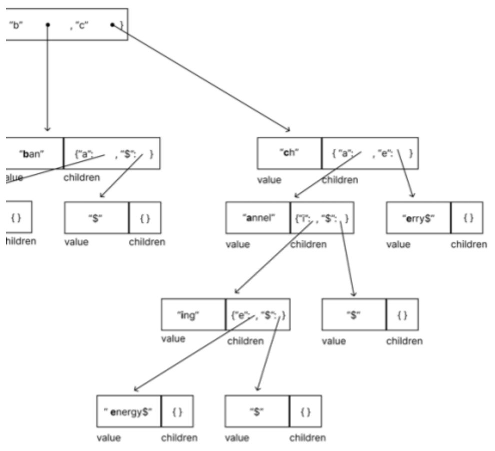
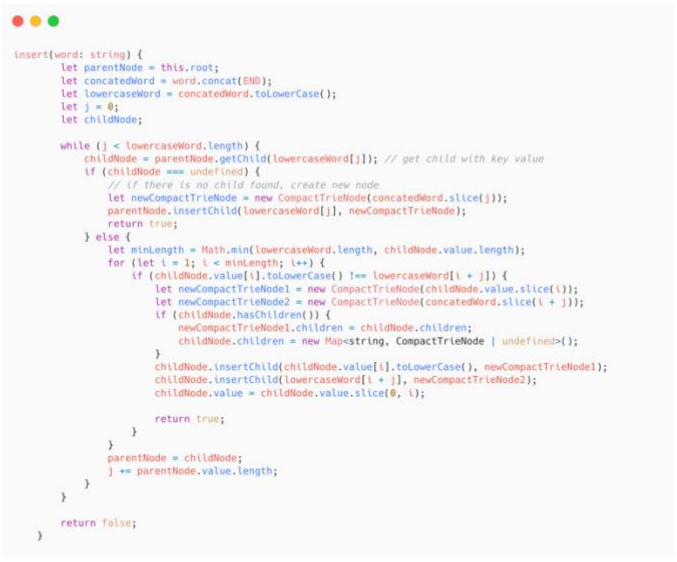


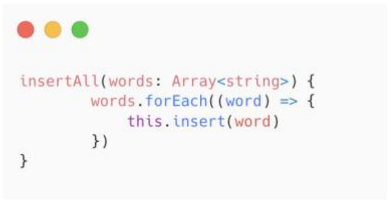
Fig. 13 Trie After Insertion of The Word "channeling energy"
(Source: Personal Library)



```
insert(word: string) {
        let parentNode = this.root;
        let concatedWord = word.concat(END);
        let lowercaseWord = concatedWord.toLowerCase();
        let j = 0;
        let childNode;

        while (j < lowercaseWord.length) {
            childNode = parentNode.getChild(lowercaseWord[j]); // get child with key value
            if (childNode === undefined) {
                // if there is no child found, create new node
                let newCompactTrieNode = new CompactTrieNode(concatedWord.slice(j));
                parentNode.insertChild(lowercaseWord[j], newCompactTrieNode);
                return true;
            } else {
                let minLength = Math.min(lowercaseWord.length, childNode.value.length);
                for (let i = 1; i < minLength; i++) {
                    if (childNode.value[i].toLowerCase() !== lowercaseWord[i + j]) {
                        let newCompactTrieNode1 = new CompactTrieNode(childNode.value.slice(i));
                        let newCompactTrieNode2 = new CompactTrieNode(concatedWord.slice(i + j));
                        if (childNode.hasChildren()) {
                            newCompactTrieNode1.children = childNode.children;
                            childNode.children = new Map<string, CompactTrieNode | undefined>();
                        }
                        childNode.insertChild(childNode.value[i].toLowerCase(), newCompactTrieNode1);
                        childNode.insertChild(lowercaseWord[i + j], newCompactTrieNode2);
                        childNode.value = childNode.value.slice(0, i);

                        return true;
                    }
                }
                parentNode = childNode;
                j += parentNode.value.length;
            }
        }

        return false;
    }
```

Fig. 14 Implementation of Insertion Algorithm in TypeScript
(Source: Personal Library)

To make it simpler for the component to insert words in a batch, the *compact trie* class also has *insertAll* method shown in Fig. 15.



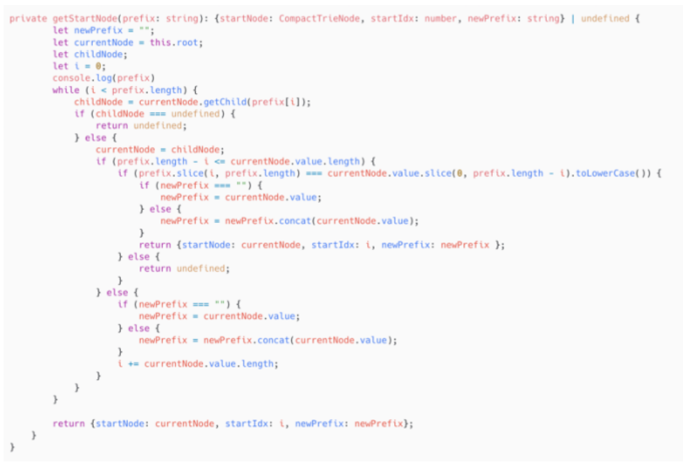
```
insertAll(words: Array<string>) {
        words.forEach((word) => {
            this.insert(word)
        })
}
```

Fig. 15 Implementation of InsertAll Method in TypeScript
(Source: Personal Library)

C. Looking up Words

The algorithm to look up for all the words matching the given prefix is similar to the insertion algorithm except for additional recursive algorithm to get all possibility. The recursive algorithm is similar to the one suggested by Jay Wengrow [2] with some modification to fit the data structure of *compact trie*. The recursive part does not immediately performed, but there are some pre-lookup algorithm needed to perform the recursion. For example, given a prefix "cha" then the algorithm for looking up words are:

1. Search for the node with maximum depth that has the same prefix as the given prefix. This is the pre-recursion part and the node is called the starting node. Searching for the starting node is similar to the one in insertion. In Fig. 16. searching for the start node also involved searching the child node and matching the value in the child node with the given string. If when searching for the child node by key, it returns *undefined* value then there is no need to perform a recursion because it means there is no words stored in *compact trie* that has the same prefix as the given prefix. If there is a child then the algorithm checks if the given prefix is also a prefix for the value in a child. If it is, then that child node is the start node of the recursion process. For given prefix "cha" then the node with maximum depth that has the prefix "cha" is node with value "annel", because the path from the root node to the start node formed a word "channel" with prefix "cha".

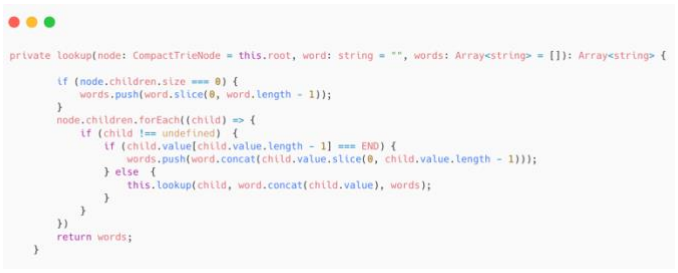


```
private getStartNode(prefix: string): {startNode: CompactTrieNode, startIdx: number, newPrefix: string} | undefined {
        let newPrefix = "";
        let currentNode = this.root;
        let childNode;
        let i = 0;
        console.log(prefix)
        while (i < prefix.length) {
            childNode = currentNode.getChild(prefix[i]);
            if (childNode === undefined) {
                return undefined;
            } else {
                currentNode = childNode;
                if (prefix.length - i <= currentNode.value.length) {
                    if (prefix.slice(i, prefix.length) === currentNode.value.slice(0, prefix.length - i).toLowerCase()) {
                        if (newPrefix === "") {
                            newPrefix = currentNode.value;
                        } else {
                            newPrefix = newPrefix.concat(currentNode.value);
                        }
                        return {startNode: currentNode, startIdx: i, newPrefix: newPrefix };
                    } else {
                        return undefined;
                    }
                } else {
                    if (newPrefix === "") {
                        newPrefix = currentNode.value;
                    } else {
                        newPrefix = newPrefix.concat(currentNode.value);
                    }
                    i += currentNode.value.length;
                }
            }
        }

        return {startNode: currentNode, startIdx: i, newPrefix: newPrefix};
    }
}
```

Fig. 16 Implementation of Searching for Start Node in TypeScript
(Source: Personal Library)

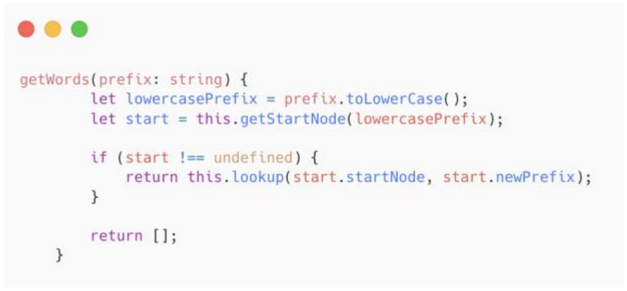
2. The starting node is now viewed as the root node making it and its descendant node a subtree of a *compact trie*. Because the starting node is the node that has maximum depth with the same prefix as the given prefix then all of its descendant also has the same prefix as the given prefix.
3. The first step is to check if the start node is a leaf node. If it is a leaf node (base 1) then the path from the original root node to that node must formed a word. This word then stored inside an array of string that will be returned at the end [2]. While pushing a new word to the array, we need to remove the end of word symbol '$' so the array only contains the pure word without the mark.
4. If the start node has children then for each child, if the child value is ended with an end of word symbol (base

2) then the path from the original root node to the child node also formed a word. Similar to step 3, this word also stored in an array of string, with the end of word symbol removed. If the child value is not ended with an end of word symbol then we perform the recursive part with new prefix which is the given prefix concatenated with the value in the child node and a new start node which is the child node.



```
private lookup(node: CompactTrieNode = this.root, word: string = "", words: Array<string> = []): Array<string> {

        if (node.children.size === 0) {
            words.push(word.slice(0, word.length - 1));
        }
        node.children.forEach((child) => {
            if (child !== undefined)  {
                if (child.value[child.value.length - 1] === END) {
                    words.push(word.concat(child.value.slice(0, child.value.length - 1)));
                } else  {
                    this.lookup(child, word.concat(child.value), words);
                }
            }
        })
        return words;
    }
```

Fig. 17 Implementation of Looking up Words Algorithm in TypeScript inspired by reference [2]
(Source: Personal Library)



```
getWords(prefix: string) {
        let lowercasePrefix = prefix.toLowerCase();
        let start = this.getStartNode(lowercasePrefix);

        if (start !== undefined) {
            return this.lookup(start.startNode, start.newPrefix);
        }

        return [];
    }
```

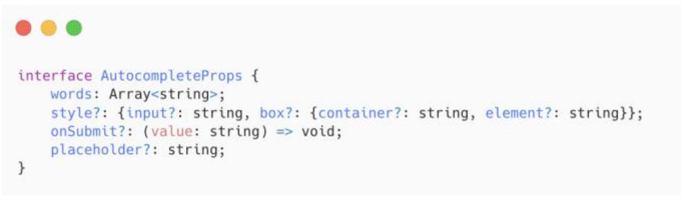
Fig. 18 Method to Retrieve All Words
(Source: Personal Library)

For given prefix "cha", after executing step 1 – 4, we will obtain an array of string containing "channel" and "channeling". The method to retrieve all words is shown in Fig. 18. First, it execute the first step which is getting the start node and if the start node is undefined then it returns empty array indicating there is no match. If there is a start node then it calls the lookup method with new prefix equals to the word formed from the root up to the start node as returned by the *getStartNode* method in Fig. 16.

D. Implementation in React Autocomplete Input Component

The *Autocomplete Input Component* in React is divided into two sub-component, the input component and the box component. The input component is the component that is responsible for obtaining the user input and calling the method in class *compact trie* to retrieve all words that has prefix equals to the string from the input. In this implementation, the autocomplete input will be restricted, meaning the user can only submit values within the options provided by the application. The box component will show all matching words based on user input.

The *Autocomplete Input Component* has four properties; words is an array of string that will be stored inside the *compact trie*; *style* is an Object for the style of the component, the string is in a format of *tailwind* styling; *onSubmit* is a method that has the value the user choose as the parameter; *placeholder* is a placeholder for the input component. The interface of this properties is shown in Fig. 19.



```
interface AutocompleteProps {
    words: Array<string>;
    style?: {input?: string, box?: {container?: string, element?: string}};
    onSubmit?: (value: string) => void;
    placeholder?: string;
}
```

Fig. 19 Interface for Autocomplete Input Component Props
(Source: Personal Library)

At first render, the component will create a new compact trie object and call the insertAll method from compact trie class. Every time the user type in the prefix in the input area, the component will call getWords method. The array returned by the method is stored in a state and rendered via box component. The website preview for *Autocomplete Input Component* is shown in Fig. 20.

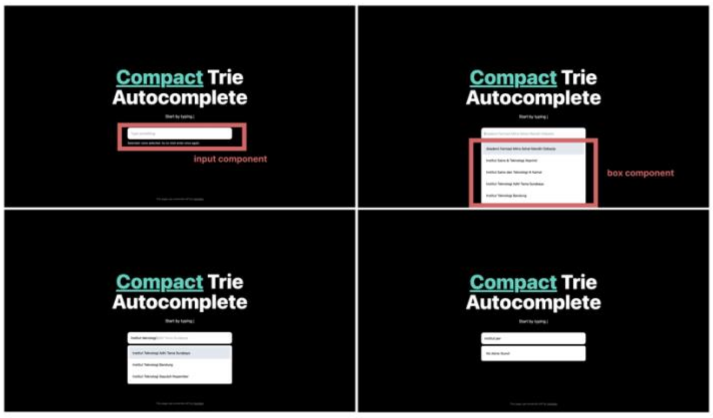


Fig. 20 Preview for Autocomplete Input Component
(Source: Personal Library)

IV. Memory Usage Comparison

In order to see the difference clearly, the words used in the comparison experiment are phrases with length more than 20. The words are retrieved from international university API by *hipolabs*. This experiment only use up to 160 universities in Indonesia.

Table 1 Example of Words Used in Experiment

Word	Length
Akademi Farmasi Mitra Sehat Mandiri Sidoarjo	44
Institut Sains & Teknologi Akprind	34
STMIK AMIKOM Yogyakarta	23
STIKES RS Anwar Medika	22
Universitas Katolik Indonesia Atma Jaya	39

(Source:

http://universities.hipolabs.com/search?country=indonesia)

Table 2 Memory Usage of Trie and Compact Trie (bytes)

Total Words	Avg Length	Trie		Trie Node			Compact Trie		Compact Trie Node		
		Shallow Size	Retained Size	Total Node	Shallow Size	Retained Size	Shallow Size	Retained Size	Total Node	Shallow Size	Retained Size
5	32	60	17592	161	2576	17456	60	1484	7	140	1348
10	27	60	25420	232	3712	25292	60	2760	15	300	2624
20	28	60	42972	394	6304	42844	60	5252	31	620	5124
40	26	60	72460	666	10656	72332	60	9804	62	1240	9676
80	26	60	125060	1152	18432	124932	60	18352	121	2420	18224
160	26	60	206420	1898	30368	206264	60	34320	234	4680	34192

(Source: Google Heap Memory Snapshot)

Computation of the memory usage by each trie were done in real-time by using google developer tools, heap memory snapshot. The snapshot showed the *shallow size* and *retained size* of an object. As shown in Table 2 and Fig. 21, the *shallow size* of both trie object are equals meaning the object have the same direct use of memory. A very significant difference is shown by the *retained size*. The retained size of compact trie is almost 10 times smaller than the retained size of basic trie (trie). The cause of this is related to the total node used by each trie. Basic trie use a total of 151 nodes to store 5 words, shown by Table 1, while compact trie only use 7 of them. This bloated node use by basic trie is because the similarity of the prefix store inside the trie is minimum. The graph in Fig. 22 shown a very fast increase in total size of the trie nodes as the total words increased.

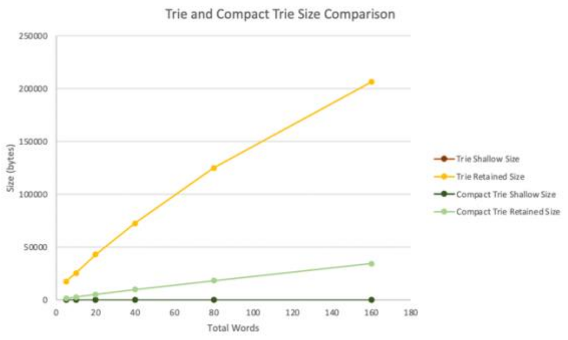


Fig. 20 Trie and Compact Trie Comparison (Source: Personal Library

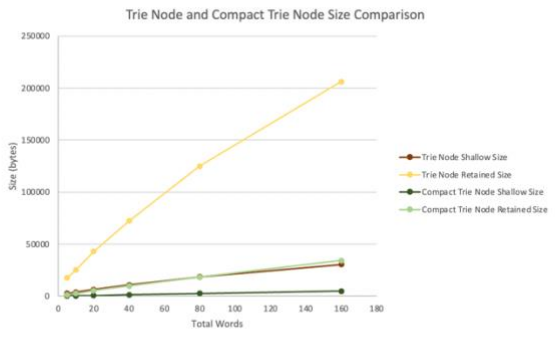


Fig. 21 Trie Node and Compact Trie Node Comparison (Source: Personal Library)

V. Conclusion

Autocomplete Input Component is an important component to improve user experience of our application. The most common implementation is a tree-based implementation called trie. The data structure of a basic trie, however, only store up to one character each node. This caused a problem in memory usage. This problem can be solved by using compressed trie or compact trie. Compact trie can reduce the size of the trie up to 10 times smaller. This is due to the node of compact trie have at least two children if they are not a leaf node and each node does not only store a character but also a string value.

VI. Appendix

The website preview along with the code implementation of *compact trie* and the basic implementation of *trie* based on reference [2] can be accessed via https://github.com/debbyalmadea/autocomplete-trie. The repository for API used in section IV can be accessed via https://github.com/Hipo/university-domains-list.

VII. Acknowledgment

I want to express my gratitude to God, who has given me knowledge and strength to finish this paper.

I am also very grateful to Ms. Fariska Zakhralativa Ruzkanda, S.T. M.T., who introduced me to the world of graphs and trees. The idea and the research behind this paper would not be possible without her guidance and knowledge.

Last, I also want to express my appreciation towards previous researcher in trie data structure and its modification. Without their insight and determination, it would not be possible for me to have the base idea for trie and this paper.

REFERENCES

[1] Autocomplete. Cambridge Dictionary. (n.d.). Retrieved December 10, 2022, from https://dictionary.cambridge.org/dictionary/english/autocomplete
[2] Wengrow, J., MacDonald, B. (2020). It Doesn't Hurt to Trie. In A common-sense guide to data structures and algorithms: Level up your core programming skills. essay, The Pragmatic Bookshelf.
[3] Heinz, S., Zobel, J., Williams, H. E. (2002). Burst tries. ACM Transactions on Information Systems, 20(2), 192–223. https://doi.org/10.1145/506309.506312
[4] Morrison, D. R. (1968). Patricia—practical algorithm to Retrieve Information Coded in alphanumeric. Journal of the ACM, 15(4), 514–534. https://doi.org/10.1145/321479.321481
[5] Askitis, N., &amp; Sinha, R. (2007). HAT-trie: A Cache-conscious Trie-based Data Structure for Strings. Retrieved December 10, 2022, from https://www.researchgate.net/publication/262410440_HAT-Trie_A_Cache-Conscious_Trie-Based_Data_Structure_For_Strings.
[6] Munir, R. (2021). Pohon 2020 Bag 1. Homepage Rinaldi Munir. Retrieved December 10, 2022, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf
[7] Munir, R. (2021). Pohon 2020 Bag 2. Homepage Rinaldi Munir. Retrieved December 10, 2022, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf
[8] Munir, R. (2021). Graf 2020 Bag 1. Homepage Rinaldi Munir. Retrieved December 10, 2022, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf
[9] Trie data structure - javatpoint. www.javatpoint.com. (n.d.). Retrieved December 12, 2022, from https://www.javatpoint.com/trie-data-structure#:~:text=Trie%20is%20a%20sorted%20tree,help%20of%20the%20word's%20prefix.
[10] CGLAB. (n.d.). Chapter 7 Data Structures for Strings - cglab.ca. Computational Geometry Lab. Retrieved December 10, 2022, from https://cglab.ca/~morin/teaching/5408/notes/strings.pdf
[11] news, M. K. O. latest. (n.d.). Memory terminology. Chrome Developers. Retrieved December 11, 2022, from https://developer.chrome.com/docs/devtools/memory-problems/memory-101/
[12] Dominators¶. Dominators - Firefox Source Docs documentation. (n.d.). Retrieved December 11, 2022, from https://firefox-source-docs.mozilla.org/devtools-user/memory/dominators/index.html#:~:text=shallow%20size%3A%20the%20size%20of,kept%20alive%20by%20this%20object

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Desember 2022

[illegible]

Made Debby Almadea Putri - 13521153