Technical Report

A dictionary is a popular application to help users look up the meanings of words. In dictionary app, there are some several basics tasks such as searching a word, adding a new word, updating a word, … . With a developer, in small data sets, there are many ways to solve the basic tasks above, for example using vector or array to save all words and search them. However, in a big data sets, it might compose lots of memories to save all word and lots of time to search our keyword. Or we can use trie to save the lists in a tree with memory: **256 childrens for a node** which is better than using vector. But it will contains too much spaces to save the tree. To address this problems, our team will use a new data structure to save all words in a tree with only **3 childrens for a node**: **Tenary Search Tree.**

A **ternary search tree** is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

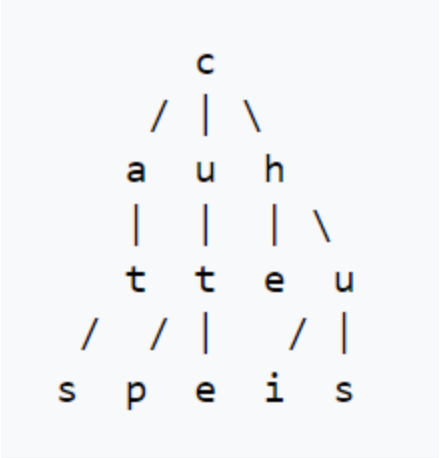
- The left pointer points to the node whose value is less than the value in the current node.
- The equal pointer points to the node whose value is equal to the value in the current node.
- The right pointer points to the node whose value is greater than the value in the current node.

The equal pointer points to the next character in the word.

Each node has a field to indicate data and another field to mark end of our word.

Each node is designed as this:

```
struct Node {
    char data; // the character in a node
    int EOS = 1; // is end of string or not
    Node* left, * middle, * right; // 3 children of a node
    vector <string> listDef; // list of definitions if this node is end of a keyword
}
```



For instance, the image above show the visualization of a tenary search tree with string: "cute","cup","at","as","he","us" and "i".

The advantage of using Tenary Search Tree is more space efficient (only 3 pointers per nodes compare to 256 in Trie).

Now I will explain in detail my algorithms in 5 basic functions in my dictionary application

- Loading a data set
- Searching a keyword
- Adding a new word
- Updating a word
- Delete a word

1. **Loading a data set**

   Firstly, let's mention a little bit about my saving data function. I will save the tree in DFS format or in another word, traversing the tree by DFS then save it into a file. When we completely traverse all subtree of a node, I will write -1 to my file to know if it completes its traversing.

   My saving tree in file will have the format:

   - If the current node is root of the tree: # 5 root's data root's isEndOfStringroot's Definition list'size list Definition
   - If the current node is not the root: # node's next state(0, 1, 2) node's isEndOfStringroot's Definition list'size list Definition
   - If we completely traverse all subtree of current node: # -1

   After saving my tree to a file, I will try to load my tree instead of loading all data set each time we open the app

   - If the current line has state -1 -> we will stop our DFS in current node

- If the current line has state 0 -> we will go to the left children
- If the current line has state 1-> we will go to the middle children
- If the current line has state 2 -> we will go to the right children
- If the current line has state 5 -> it is our root

```
void traverseToLoad(Node*& root) {
    if (posTreeNode >= listTreeNode.size()) return; // when we finish our DFS

    if (posTreeNode == 0) { // if my node is root of the tree
        root = listTreeNode[posTreeNode].nextNode;
        posTreeNode++;
        traverseToLoad(root);
    }
    else {
        for (int i = 1; i <= 4; i++) {
            if (listTreeNode[posTreeNode].nextState == -1) { // if my node completely traverses
                posTreeNode++;
                return;
            }

            if (listTreeNode[posTreeNode].nextState == 0) { // go to the left children
                root->left = listTreeNode[posTreeNode].nextNode;
                posTreeNode++;
                traverseToLoad(root->left);
            }
            else {
                if (listTreeNode[posTreeNode].nextState == 1) { // go to the middle children
                    root->middle = listTreeNode[posTreeNode].nextNode;
                    posTreeNode++;
                    traverseToLoad(root->middle);
```

```
            }
            else {
                if (listTreeNode[posTreeNode].nextState == 2) { // go to right children
                    root->right = listTreeNode[posTreeNode].nextNode;
                    posTreeNode++;
                    traverseToLoad(root->right);
                }
            }
        }
    }
  }
}
```

2. Searching a keyword

The most important function of dictionary is definitely searching a keyword and it is extremely efficient to solve by **Tenary Search Tree.**

There are 3 cases in this tasks **(assume my keyword is variable word, current position of my word is pos)**

- If my node's character > word[pos] then we will move to the left children
- If my node's character < word[pos] then we will move to the right children
- If my node's character = word[pos] then
    - If the current node is not end of a word, we will move to the middle children and add 1 to pos
    - Else return the list definition of our keyword

We stop our searching if the current node is **NULL** or we reach to suitable word with isEOS = true or we will return our list definition of the keyword

```
// return a vector contains all definition of my searching word
vector <string> searchDefinition(Node* root, string word, int pos = 0) {
    if (root == nullptr) { // if root is NULL then the word is not exist
        vector <string> res;
        res.clear();
```

```
        return res;
    }

    if (root->data > word[pos]) // move to the left
        return searchDefinition(root->left, word, pos);
    else {
        if (root->data < word[pos]) // move to the right
            return searchDefinition(root->right, word, pos);
        else {
            if (pos + 1 == int(word.size())) { // return our list
                return root->listDef;
            }
            return searchDefinition(root->middle, word, pos + 1);
        }
    }
}
```

3. Adding a new word

Another function of the dictionary is adding a new word to our data. There are 3 cases in this tasks **(assume my keyword is variable word, current position of my word is pos)**

- If my node's character > word[pos] then we will move to the left children
- If my node's character < word[pos] then we will move to the right children
- If my node's character = word[pos] then
    - If the current node is not end of a word, we will go to the middle children and add 1 to pos
    - Else add the new word and its definition
- While inserting if we reach a null node or leaf node, we just allocate memory and add a new one there and repeat.

```
// to insert a new word
    void insert(Node*& root, string word, string definition, int pos = 0) {
```

```
    if (root == nullptr) {
        root = new Node(word[pos]);
    }

    if (word[pos] < root->data) {
        insert(root->left, word, definition, pos);
    }
    else if (word[pos] > root->data)
    {
        insert(root->right, word, definition, pos);
    }
    else {
        if (pos < word.size() - 1) {
            insert(root->middle, word, definition, pos + 1);
        }
        else {
            root->EOS = 1;
            root->listDef.push_back(definition);
        }
    }
}
```

4. Update a word

We will do similar traversing as 2 functions above and change our information if we find the word.

```
// to edit a definition of a word
void edit(Node* root, string word, string newDef, int index, int pos = 0) {
    if (!root) return;

    if (root->data > word[pos])
```

```
        edit(root->left, word, newDef, pos);
    else {
        if (root->data < word[pos])
            edit(root->right, word, newDef, pos);
        else {
            if (pos + 1 == int(word.size())) {
                if (!root->listDef.empty()) {
                    root->listDef[index] = newDef;
                }
                return;
            }
            edit(root->middle, word, newDef, pos + 1);
        }
    }
}
```

5. **Delete a word**

We will traverse similar as the searching and adding a word. But in the last step when we go to the end of our word, we will delete all definition of the word and assign its isEndOfString state to 0

```
// to remove a word
    void remove(Node*& root, string word, int pos = 0) {
        if (root == nullptr) return;


        if (word[pos] < root->data) remove(root->left, word, pos);
        else {
            if (word[pos] > root->data) remove(root->right, word, pos);
            else {
                if (pos < word.size() - 1) {
                    remove(root->middle, word, pos + 1);
```

```
            }
            else {
                root->listDef.clear();
                root->EOS = 0;
                return;
            }
        }
    }
}
```

The complexity for 5 basics function above: (Let's say that the length of current word is k and that there are n total strings.)

	Average-case running time	**Worst-case running time**
Loading data sets	O(n * k)	O(n * k)
Searching a keyword	O(log n + k)	O(n + k)
Adding a new word	O(log n + k)	O(n + k)
Updating a word	O(log n + k)	O(n + k)
Deleting a word	O(log n + k)	O(n + k)

The running time of each functions with this data: 60000 words with 3-4 definitions for each word

Word	**Loading data sets**	**Searching a keyword**	**Adding a new word**	**Updating a word**	**Deleting a word**
dog	**0.478246 s**	**0.000001 s**	**0.000001 s**	**0.000001 s**	**0.000001 s**
cat	**0.474703 s**	**0.000001 s**	**0.000000 s**	**0.000001 s**	**0.000001 s**
mouse	**0.475233 s**	**0.000001 s**	**0.000001 s**	**0.000001 s**	**0.000001 s**
tiger	**0.481734 s**	**0.000001 s**	**0.000001 s**	**0.000001 s**	**0.000001 s**