Data Structure – Trie

Intro.

The term “trie” comes from the word “retrieval”, meaning the retrieval of data. In computer science, trie, aka prefix tree, is a type of tree data structure utilized to efficiently locate a key from a set. The key usually takes the form of `string`; however, not necessarily, it could also be a binary data like `int`, `float` or memory address. For simplicity, I shall introduce trie data structure that is keyed on only string of alphabetic characters in this paper. Nevertheless, it doesn’t affect the comprehension of the essential concept of trie, for whatever data type a trie is keyed on, the method is no other than associating the node that represents the element of key, and defining the key by the location of nodes.

Motivation.

It all starts when I was dealing with HW9, where the majority of problems is about tree data structure. In Problem A1, it requests the implementation of a dictionary, which stores string words and provides function `StartsWith(prefix)` to return all words starts with the prefix and is in the dictionary. At that time trie comes into my mind, I had had heard of its efficiency in searching string by prefix. Though I didn’t know much about it, it would be great to start learning trie to deal with HW9 and term report(this paper) at the same time, it’s two bird with one stone. The idea sounds quite time-efficient; and therefore, I choose trie as my subject.

Application - Trie Dictionary.

Here I will introduce the concept of trie with a trie dictionary, which is not only a basic application of trie but a good example used to demonstrate how trie works. Previously we mentioned that a trie is a tree data structure used for retrieving a certain key from within a set. So in the concept of trie dictionary, we can see a word as a “key”, the dictionary itself as a “set”, and trie as the method we use to arrange the words in this dictionary.

We start by building up the dictionary. Initially, just like most other data structrue, we create a root node representing nothing other than a root. The dictionary is empty at first, therefore I would like to insert some words. The first word is “chapel”. To insert the word into the dictionary trie tree, we firstly transform the string `chapel` into a linked list composed of 6 nodes representing character `c`, `h`, `a`, `p`, `e` and `l` respectively. The list would be like `c->h->a->p->e->l`, then we link the first node of the list to the root. Lastly, mark the last node `l` as the end of the word (**figure 1**). The insertion process doesn’t change if the word we proceed to insert doesn’t share the same prefix with any word exist in the dictionary, for

example, the insertion of the string `hello` (**figure 2**); otherwise, however, the process would be a little different. Suppose that the next word to be inserted be “champion”, because it shares the prefix “cha” with the word exist in the dictionary, “chapel”. In this case, because `c->h->a` list is already built from the insertion of “chapel”, so instead of building a 8-nodes list `c->h->a->m->p->i->o->n`, we could merely build a 6-nodes list `m->p->i->o->n`, then link the first node `m` to the end of prefix, which is the `a` from the list `c->h->a` (**figure 3**). Finally, mark the end of the word `n`. In a similar case, we want to add “chap” (“chapter” in short) into the dictionary. The word “chap” not only shares the prefix “chap” with “chapel”, but it is the prefix itself. In this case, we only have to mark the end of the word, which is node `p` from `c->h->a->p->e->l`.
`->l` (**figure 4**). Currently, we have 4 words in our trie dictionary.

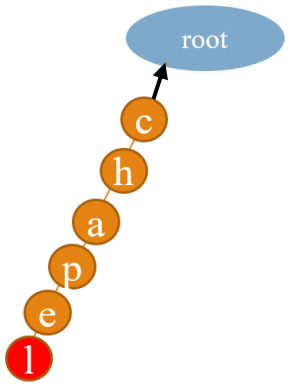

figure 1, insertion of “chapel”

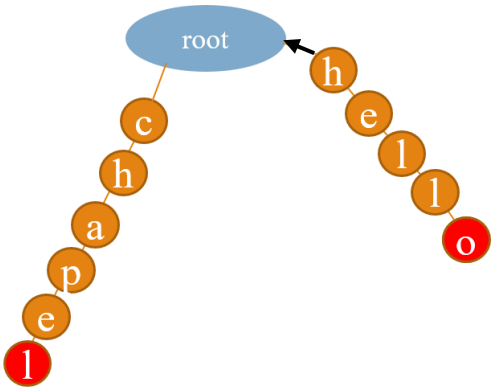

figure 2, insertion of “hello”

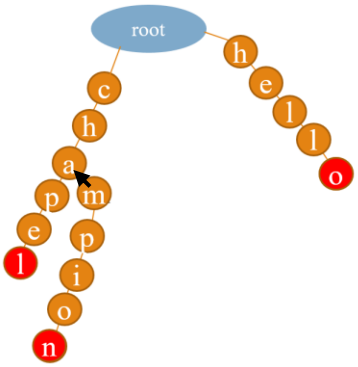

figure 3, insertion of “champion”

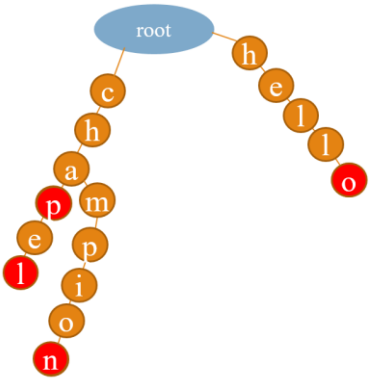

figure 4, insertion of “chap”

In conclusion, there are basically 3 different cases for the insertion process:

1. No prefix: Like the second word “hello” shares no prefix with the first word “chapel”, so we just build the list composed of all characters within the word: `h->e->l->l->o` and link the `h` to the root node.
2. Shares prefix: Like the third word “champion” shares prefix “chap” with the word “chapel”, so we build the list composed of all characters except the prefix: `m->p->i->o->n` and link the `m` to the `a` from `c->h->a->p->e->l`.
3. Is prefix: Like the forth word “chap” is the prefix of “chapel”. In this case we mark the `p`

node from `c->h->a->p->e->l` as the end of a word.

From above we see how the insertion of trie tree works, now let's consider how to search for a word to see if the word exists.

So far we have a few words in the dictionary: "chapel", "hello", "champion" and "chap". Suppose that we would like to search for the following string: 1. `chess` 2. `chap` 3. `hell`. The SOP of searching process are composed of two steps:

1. Check if the first character is associated to the root, and if the rest is associated to the previous character.
2. Check the last character has been marked as the end of a word.

Following the SOP, we search for the string `chess`:

`1. root->c(true), c->h(true), h->e(false)`

`e` does not linked after `h`, the searching ends at step one, and the result is the absence of the string `chess`.

Subsequently, we search for `chap`:

```
1. root->c(true), c->h(true), h->a(true), a->p(true)
2. p.isEnd == true
```

All the character nodes are associated by order and the last node is marked as the end of the word. The result is, therefore, the existence of `chap`.

Lastly, we search for the string `Hell`:

```
1. root->h(true), h->e(true), e->l(true), l->l(true)
2. l.isEnd == false
```

The searching ends at the last step. Although the word "hell" is contained in "hello", but the last character `l` is not marked as "the end of a word", which means "hell" has never been inserted into the dictionary, elsewise `l.isEnd` should return `true`.

Here lies an important concept: to search, we check if the last character node is marked as the end of a word, not if it is a terminal node. In other words, if the last character is a terminal node, the word must exist; if it is not, it's unsure if it does not exist.

Above we use trie dictionary as an example to illustrates how a trie structure inserts a key and searches for a key. In comparison with binary search tree and many other tree structure discussed in the DSAP classes, trie t3. utilizes a quite different way to store keys. Unlike most of the tree, the node of trie tree stores no key, instead, a key is defined by the location and the association of the node; and, most interestingly, two keys share the nodes by which they are defined if the keys share a prefix, this is the major reason why trie can locate a key in a time-efficient way.

Implementation.

In this section, we discuss the implementation strategy of basic trie class. We again take trie dictionary for example. The class provides two basic operations:

1. `void Insert(key)`, to insert a key (in the form of `string`) into the trie tree.
2. `bool Search(key)`, returning `true` if the key exists; `false`, otherwise.

Firstly, we should define the node of trie. A trie nodes should store its children (the memory address of the node that represents the character associated after it). Therefore it should store 26 (alphabetic size) pointers to trie node, and it is a null pointer if in that position the child does not exist. Lastly, the node itself can be marked as "the end of a word" or as "not the end of a word". So the implementation of trie node should be like:

```
struct TrieNode {
    TrieNode* children[26];
    bool isEnd;
    ~TrieNode();
};

TrieNode::~TrieNode() {
    for(int i = 0; i < 26; i++)
        delete children[i];    //destroy all the 26 children
}
```

Now we return to trie tree itself. Just like most other tree structure, the private member contains the first created node – root node. So the trie class should look like this:

```
class Trie {
    public:
        Trie();
        void Insert(const std::string&);
        bool Search(const std::string&);
        ~Trie();

    private:
        TrieNode* root;
};
```

The constructor is quite simple, we simply need to create a new node, obtain its address, and assign it to the `root`:

```
TrieNode* CreateNode() {                //obtain address of a
    TrieNode* Node = new TrieNode;      //new created node
    for (int i = 0; i < 26; i++)
        Node->children[i] = nullptr;    //no children at first
    Node->isEnd = false;                //not end of a word
    return Node;                        //return address
}
Trie::Trie() {root = CreateNode();}     //assign address to root
```

Although in the aforementioned introduction we say the insertion has 3 cases to be considered, the implementation is rather, well, straightforward:

```
void Trie::Insert(const std::string& key) {
    TrieNode* curr = root;  //traverse from the root
    for (int i = 0; i < key.length(); i++) {  //traverse the
        int index = key[i] - 'a';             //tree with key
        if (curr->children[index] == nullptr)    //next node
            curr->children[index] = CreateNode(); //not exist
        curr = curr->children[index];
    }
    curr->isEnd = true;     //mark the end of the word
}
```

In the fifth line, if the next node was not built before, just create a new one.

The implementation of `Search()` is almost identical to that of `Insert()` because the traversal method remains unchanged. We only need to make two modification:

```
bool Trie::Search(const std::string& key) {
    TrieNode* curr = root;
    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (curr->children[index] == nullptr)
            return false;  //M1. Return false if child not exist
        curr = curr->children[index];
    }
    return curr->isEnd;   //M2. Check if the last node is marked
}                         //as the end of a word
```

The destructor is the easiest, but with the most fascinating technique. What we do is to simply delete the root node:

```
Trie::~Trie() {delete root;}
```

The magic here is, when we delete a root, we delete all the nodes below. Recall the destructor `~TrieNode`, in which we delete all children of the node. So when we delete a node, we call the destructor for its children nodes, in which we delete the grandchildren nodes, and again we call the destructor for the grandchildren nodes. In this way, we clean the "trie family tree" by eliminating the root.

Complexity Analysis.

1. Constructor: `Trie()`

In `Trie()`, we call another function `CreateNode()` to create a new node and obtain its address to assign it to `root`. In the `CreateNode()` function, we create node pointers for 26 times constantly. And, `Trie()` do nothing than assign the address returned by `CreateNode()` to `root`, so the time and space complexity is constant $\Theta(1)$.

2. Insertion: `Insert()`

In `Insert()`, we pass by the reference of the key, and traverse the string key until its last character in any case. So, let the length of the key be n, the time complexity is $\Theta(n)$ for all cases. But the space complexity is diverse. In best cases, we never call `CreateNode()` because the key is a prefix itself, so the complexity is constant.
In worst cases, we call `CreateNode()` for every character traversed because there is no prefix, so the complexity would be $\Theta(n)$ in this case.

3. Searching: `Search()`

In `Search()` we pass by the reference of the key, and traverse the string key until end or until the next character is not associated to the previous one. So the worst case time complexity is $\Theta(n)$ (traverse till the end) and the best case is constant $\Theta(1)$ (the first character is not linked after the root). As for space complexity, no new node is created, because we just simply do the "searching", so space complexity is constant $\Theta(1)$ for all cases.

4. Destructor: `~Trie()`

In the destructor, we call the destructors for all nodes in the tree and delete them. So let the size of the trie be m, because we call the destructor for trie node and delete them for m times, the time complexity is $\Theta(m)$ in any case. And, because we pass the pointer to nodes for m times, so the space complexity is $\Theta(m)$ for all cases as well.

END