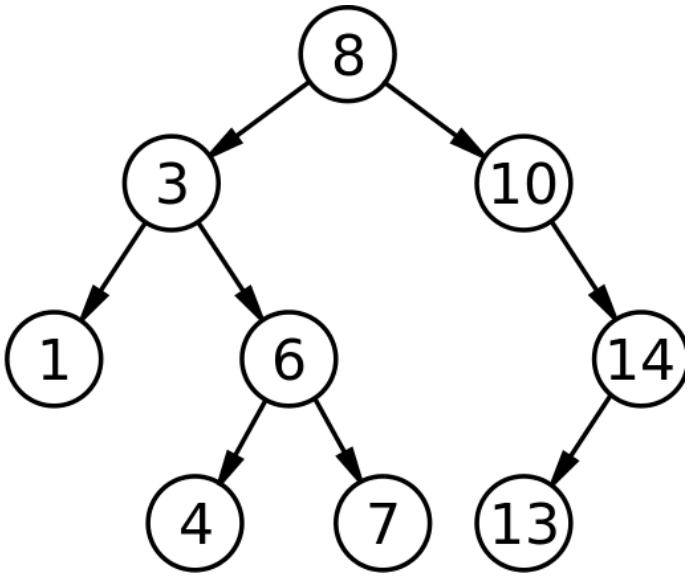
MINI SEARCH ENGINE –FINAL PROJECT IN CS 163

In this project, we will design and implement a mini search engine. Just like Google, Altavista or Yahoo, which are some of the most popular search engines that we can find on the Web. The task performed by a search engine is, as the name says, to search through a collection of documents. Given a set of texts and a query, the search engine will locate all documents that contain the keywords in the query. The problem may be therefore reduced to a search problem, which can be efficiently solved with the data structures we have studied in class.

BINARY SEARCH TREE



At the first time, we determine that we will not use the Binary Search Tree. Simply because it doesn't balance .So we must take a lot of effort to search for a word. The worst case for each searched query in the binary search tree has the height of h is O(h). In order to insert a piece of data into the BST, we have to search for a position to insert it in. With BST, we always insert the new data at the leaf node, that will make the situation worse and worse. If we have to insert k data (k is a very large number may be 100.000.000 or more) in to the BST. If we are out of luck, the worst situation is O(k*h). Of course there is no way that case happens … But in the whole, I believe that BST would be a very bad solution to come up with this search engine. Of course we also try to think of anything to cover its weakness in this situation but there is nothing that we can come up with to make it better.

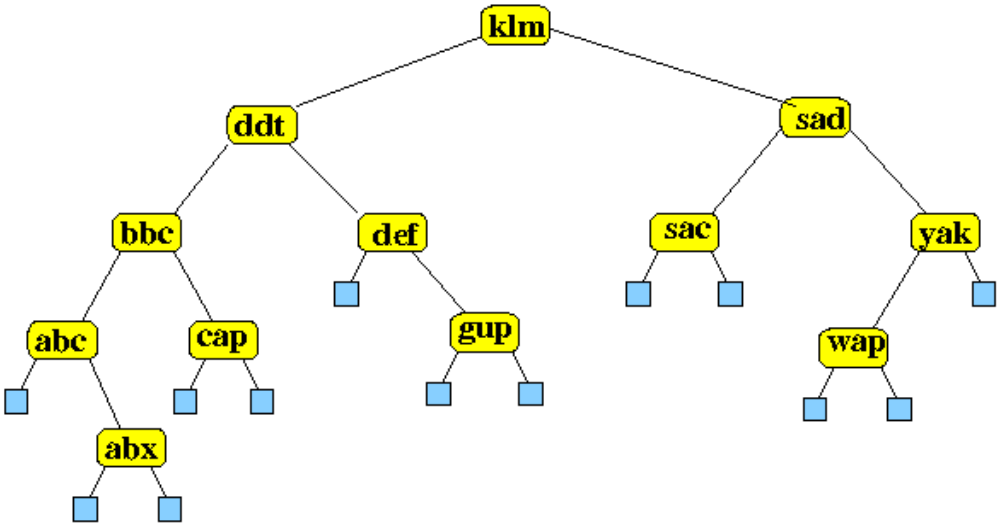
AVL TREE

So we consider about AVL Tree. It is a tree that is mostly balance. So that with a huge amount of input words, we will have a short tree. With the binary tree has the height h balanced, the efficiency of the search engine would be at most $O(\log_2(h))$. Even if h equal 10.000.000,if we use AVL Tree, it is just a small task.

So why we didn't choose this type of Tree? Because of its balance, we must do extra things for the rotations. The more data added in, the more rotation we must make in order to keep the tree balance. Moreover, each rotation will be done by the recursion. More rotations, more function is call, more time costs. But if we can do it without the recursion, it may be a good way to try. And in the real situation, even if the data structure at some time if very large, it doesn't mean that it will do many time of rotation. that is the factor that no one can predict. So what we gonna do next is trying to implement AVL tree in the fastest way as far as possible.

When finishing AVL Tree, we were surprised… It takes more than we think, 500ms for 50.000 words. So we try to implement more and more. We made our code as simple as we can. We brought any of local variable if possible to the global scope. We even created a function to initialize a before-handed Tree with full of Node at some height (we tried it at h equals 5 at most, h equals 6 is taking to much time), then we would mark it in what files after read data form files.

So far, we take the time down to about 350 ms with 50.000 word.



PROBLEM?

And day after day, testing, we see some important problems in our code. There are three main problems to mention. We found it a threat if we continue to use our data structure building up an search engine.

1. AVL tree take a piece of data as a single Node. So if I have 1.000.000 different word, it will store all data in the RAM. It will take over many memory in RAM and may be crashed (I use my PC 2GB RAM to test so it can be crashed).
2. AVL tree comparison between data which is an array of characters is very slow. Because if it is a string, so when comparing, it will takes O(m) in worst case to have a result.
3. With AVL tree, incomplete matches become very slow (if it runs too long we cannot call it successful).
4. Using linked list to build up the tree with the huge data may cause fragment memory.

Now let me explain clearer about those above.

SUCK UP TOO MUCH MEMORY IN RAM

A Node in our data structure will have these things like the following code:

```
struct Node{
        char* str;
        bool  nFile[300];
        Node*Left, *Right;
}
```

I would like to use String from C++ but string C++ operators is cost much time then String-Terminated-By-NULL in C.

In general, for each Node, it costs 300 bytes sum up with the length of the string str (len)*1 byte and the size off two pointer 4 byte. So each Node costs (300 +len+8) Bytes. If we have to insert 10.000.000 different Word in to the Tree, it would be nightmare!

We could use vector<int> nFile in order to decrease the memory cost but it will make it more harder for the Search engine (may be the search engine would be much slow).

Supposed with a very large set of data, the average len of each word is about 6 so one Node costs 314 Bytes. If we run a program in the computer 2 GB Ram. We have at most 7 millions Nodes. Just with 1.670.000 Nodes we have used up about 500 MB!

COMPARISON TAKING TOO LONG

The comparison occurs when we insert a new Data to a Tree. We arrange words in Tree in alphabetical order. So the comparison will be directly between those string. But unfortunately, the comparison with strings has the efficiency O (m*n), in which n is the length of the string and m is the length of another string. Thanks for the great support of C language library, we have a function to do the comparison on string. But the complexity is still worst case in O (n*m). And unfortunately, if we do not invent a algorithm with complexity O(m), we cannot do better than those founders of C language.

To fix this up, we thought of the hash function to change an array of characters into a simple number. We found the hash function very promised. It has a very low possibility of occurring collision. But then we thought that, the integer has the largest size in our system is long long (32 bytes). But there is still a little of change that collision occurring. So we will not want to take any risks. And if we used hash, another problem will be arise. How can we form an array of string from a number. Yeah you know… Our best friend, Recursion… but it takes to much long too. Then we quit following this idea.

INCOMPLETE MATCHES

With AVL trees, incomplete Matches will take many work effort to do. It is clearly to see, One query with incomplete Matches has to search all the nodes of the tree and search for the Node has the part of the query. If the AVL tree has the height h, so we has nearly $(2^h -1)$ Node, then to search all Nodes along with checking if is there exist a query in those strings, we have the complexity O ($(2^h -1)*m*n$) for each query. So it is very expensive.

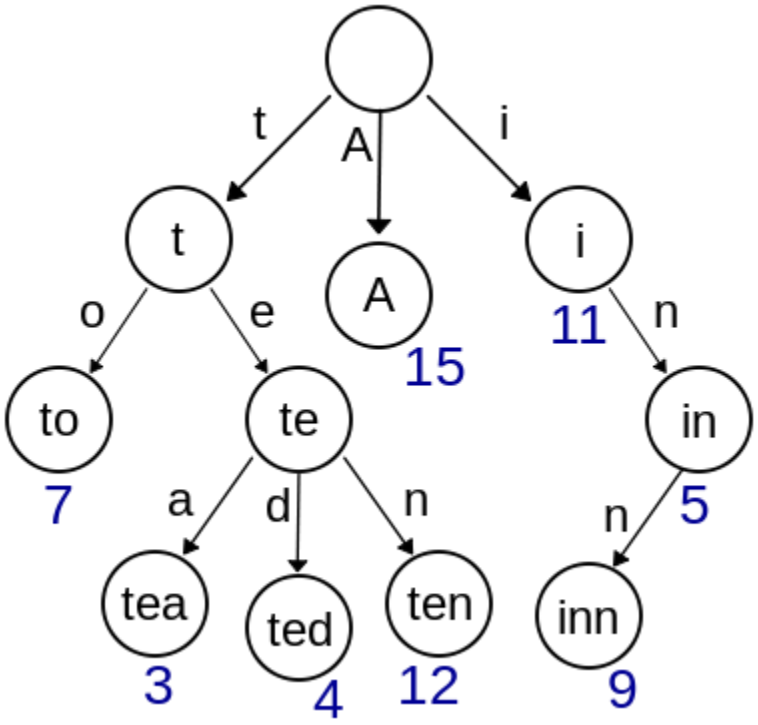
FRAGMENT IN MEMORY

Due to the task we need to do, we need to create many Nodes. So we must have them on the Heap. But the memory we assign for is at some random address. Imagining it like the Tetris game, if you arrange it well, you can build it up very high (although it means more dangerous, but high score !). The memory does not let us arrange as our own will, so who know what will happen ?

TRIE, THE SUITABLE DATA STRUCTURE FOR THIS TASK

So in the end, we decide to use the Trie.

As you have already known: In computer science, a trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.



I believe Trie has many advantages over AVL tree in solving this task:

1. Trie do not need to store any string in each Nodes. So we can decrease the amount of memory wasted
2. Trie is designed that can avoid making new Node meaninglessly
3. Insertion, in the whole is faster than AVL tree
4. Searching is easier, faster
5. We do not need complex comparison
6. In memory, it performs with real words better than AVL

MEMORY

At the first glance, we obviously see that trie take more Node than the AVL Tree. Looking at the picture above, trie performs 6 word “to, tea, ted, ten, inn, A” in 11 Nodes. But with AVL, we just need 6 Nodes. But when we divide word into small pieces, there would be a lot of common character in those string. And that will make the trie advantage over AVL Tree. If we have some words “cooperatively”, “cooperation”,”cooperate”… , AVL will store the sub string “cooperat” 3 times. But with AVL Tree, they only store once. Imagining that we have a dictionary with many word like this to store in to the Tree, trie has to perform the work better than the AVL.

In real case, we run some experiments and see that, with 90 MB data, over 15.000.000 words, trie just needs 1.400.000 Nodes to stores. And with these small data you give us, they just about 90.000 Nodes has been built.

```
Struct Node{
        bool nFile[300];
        Node* mChild[MAXCHAR];
}
```

So each Node is about 300+(MAXCHAR)*4 bytes.We have about 26+9=35 character for word and number. In total, each Node costs 450 (Bytes)

Because I know the number of Node will not exceed 2.000.000 Nodes so we decide to emphasize the speed of the algorilhm.

Note: We can just used a vector for both nFile and mChild, make the complexity of Searching to O (MAXCHAR*k).

So I believe Trie will win.

INSERTION AND SEARCHING

Inserting a word in to Trie cost at most O (k*MAXCHAR), in which k is the length of the word and MAXCHAR is the maximum key to which the current key is connecting. If we

decide to emphasize on the speed, like what we just say as the memory part, it cost the complexity O (k) ⇔ O(1)

So does searching, it will have the complexity at most O(k) if we want.

It is easier to search for the incomplete Matches too. If we want to search by the suffix (EX: "*an"), we just go to each Node and Search for "an". With prefix (EX: "an*") we just need to search for an and print out all the sub-tree below. About many incomplete part in string is hard, (Ex: "*an*s*d"), but we can still do it. With AVL, we would say "No!!".

Real Time Through Optimization Code

Using Trie with 1 file data, total 3MB 365000 words:

- Trie with pure recursion, using string to read file: 6000ms
- Trie without recursion, using string, using many function: 1800ms
- Trie without recursion, using char*, remove unneeded function: 500m

Using Trie with 50 file data, total 100MB, about 16.000.000 words:

- without recursion, using char*, remove unneeded function: 2000ms
- remove unneeded allocated data, change the loop backward, using bitwise: 1600 - 1700ms
- using Data pool: 1155-1200ms

Testing with Desktop Core Duo, Ram 2GB

DATA POOL

After testing over and over again, we stuck with the time 1600-1700ms with 100MB data files. And then we realize that most the time consumed is for creating new Node on insertion. So we were thought of the way to creating a tree full with Node before-handed.

We thought about creating a Tree. Simply using recursion to create the full tree, of course, with just the height of 5 it initialize very slow, I increase it to the 6, then I receive a Crash. But for the tiny tree of that height, it did not do any help.

Then we come up with the idea. This project task is for the real case document, so all the test case must be real. So we made it to have a dictionary initialized first, after that inserting all the data on files later. And it really did a little help, we now reach the speed 1400ms -1500ms. That's very well.

And we thought of a data pool. Pool likes swimming pool, but it contain data. We create an amount of data enough. We create a pool of Nodes. Exactly, we will create an array of Nodes .Every time we need the new Node, we will call the pool to give us a little Node.

This gave us many advantages. At first I just used an array of pointer Node, you know, for keeping the links that each Node receives. If we do not do any initializations to that array, they only keep a Trash. But at that time, we were using a trick to make sure that the address the pointer keeping is the real one. We used two integer numbers, long long type, which has 32 bit. Each of these bit will keep the state 1 or 0. If that is bit 1, the pointer keep the real one. Otherwise, that address is just trash.

If we have created every Node from the start, and set all links it holding to NULL, there's been no need for the two long long number exists, no need to change the state of the bit from 0 to 1 or some relevant things.

And that make us running with 100 MB data in 1100-1200ms with Core Duo Operator System.