Tries and LCP

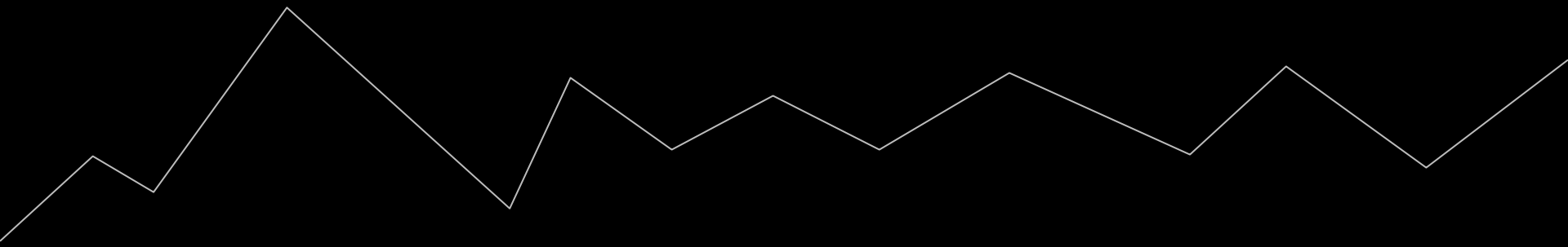
Authors: Prince, Muskan
19th Feb ‘23

The Problem

Numbers, Numbers, Numbers

- Dealt with numbers only!
- Strings, more practical
- No time for hashing collisions
- Faster, stronger, better (*insert big asterisk here)

The Variants

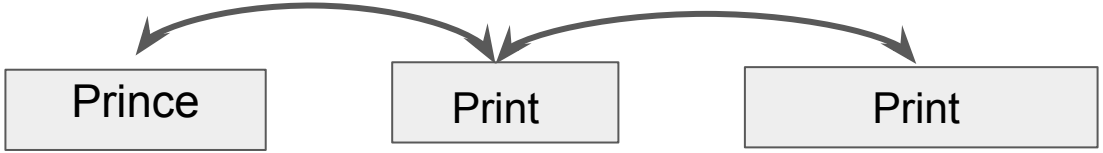


Linear Bros

- Stack : WE CAN DO BETTER!
- Queues : No different than stack

Big Linear Bros

Linked List : We don't want this!



Graph?

- Remember? Low Auxiliary space?
- Or low time complexity
- Or that graphs are complex structures?
- Or that I am not well versed in graphs!

Tree: The Messiah

The Tree Variants

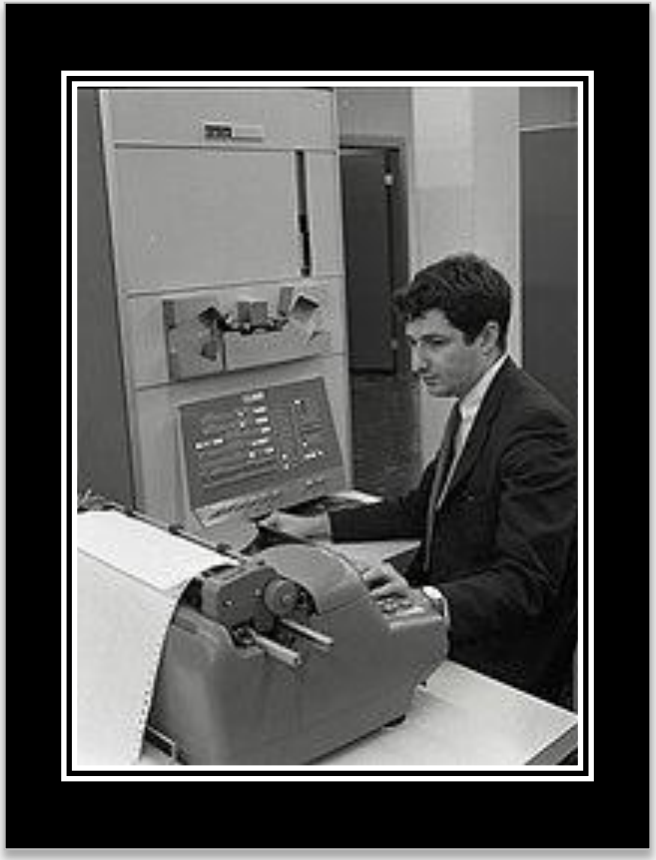
- Binary — Limited by the k in k-ary
- AVL — Same as binary trees
- RB Trees — Funny
- B-Trees — Repetitive
- B Search — Repetitive

Tries

Edward Fredkin

This Guy

- Trie from RETRIEVAL
- The stupidest name. /’tri:/
- Later changed to /’traɪ/



Basic Idea

Performance

—

- Olog(n*m) [n = no. of strings, m = avg. length of strings]
- No repetitions
- No fextra fields
- No collisions
- Best O(n) algo!

The Project

- Least Common Prefix using Tries
- Autocomplete using trie (*sort of)

Longest Common Prefix

Musk	Muskan	Muskellunge

ObiWan	Abhinawan	Swan

The Code

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

A Preview

```cpp

#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

/*
 *
 *
 *
 * LCP
 *
 *
 * */
struct trie_word
{
        bool wordState;
        unordered_map<char, trie_word*> character;

        trie_word() {
                wordState = 0;
        }
};

void insert(trie_word* root, std::string str)
{

        trie_word* temp = root;

        for (char char_val: str)
        {
```

```cpp
struct compl_trie* getNode(void)
{
        struct compl_trie* new_node = new compl_trie;
        new_node -> word_end = false;

        for (size_t i = 0; i < 52; ++i)
                new_node -> kids[i] = NULL;

        return new_node;
}

void ins_auto(struct compl_trie* root, const std::string key_val)
{
        struct compl_trie* traverse_node = root;

        for (size_t lvl = 0; lvl < key_val . length(); ++lvl) {
                int index = CHAR_TO_INDEX(key_val[lvl]);
                if (!traverse_node -> kids[index])
                        traverse_node -> kids[index] = getNode();

                traverse_node = traverse_node -> kids[index];
        }
        traverse_node -> word_end = true;
}
```

LCP Pseudo Explanation

1. Create original copy of root

2. Create copy of root

3. Loop through the sub nodes only if wordState is false or node has only 1 child

4. Add the current character to previous iteration

5. Assign iterator in map to go through each characters

Autocomplete Pseudo Explanation

1. Create original copy of root!

2. Calculate character index and iterate until array[index] is not reached!

3. If the index is leaf, exit!

Else, from index 1 or array, add current character to previous iteration

Guide to LCP make

Common Pitfalls And Hacks Implemented

Common Pitfalls

- Use of standard iterator for non-linear data structure
- Inserting more than specified!
- Not checking edge cases
- Substandard function implementations

Hacks

- Use of Regex for insertion files
- Use of public domain files for storage

FAQ

- Is it Regex?
- No, Regex is compiled! The search string is in no way compiled or parsed here!
- Recursive searching for each node and each hashmap key.
- How does LCP work?
- What is Auto I in your code?
- Auto automatically assigns an iterator or complex type
- This is not autocomplete?
- Autocomplete is a general sense but not a fully fledged system for completion!

Final Words

—

- Expect more documentation
- A fully fledged autocompletion on the way
- New Perl Implementation

Hold no bars!
Questions? Shoot 💪