Trie

Task

Let's imagine the following task: we have a set of strings, and we need to support three types of queries:

1) Add a string to the set
2) Remove a string from the set
3) Check if a string is in the set.

insert(abc)

s = {abc}

insert(aba)

s = {abc, aba}

insert(ba)

s = {abc, aba, ba}

insert(ba)

s = {abc, aba, ba}

remove(ba)

s = {abc, aba}

find(ba), no

s = {abc, aba}

Trie

Let's construct a tree where each edge contains a letter.

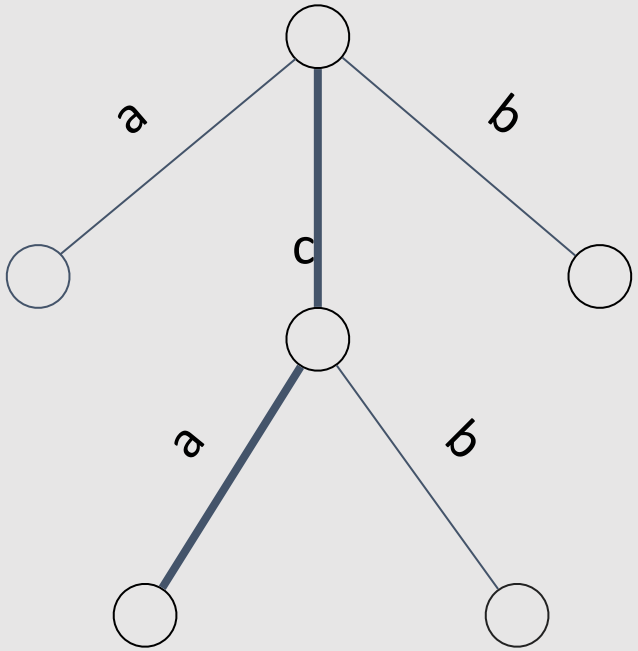
Moreover, let's agree that from any given vertex, there can only be one edge with each letter

a
b
c
a
b

Path

It can be noted that any path from the root to a certain vertex represents a word.

ca

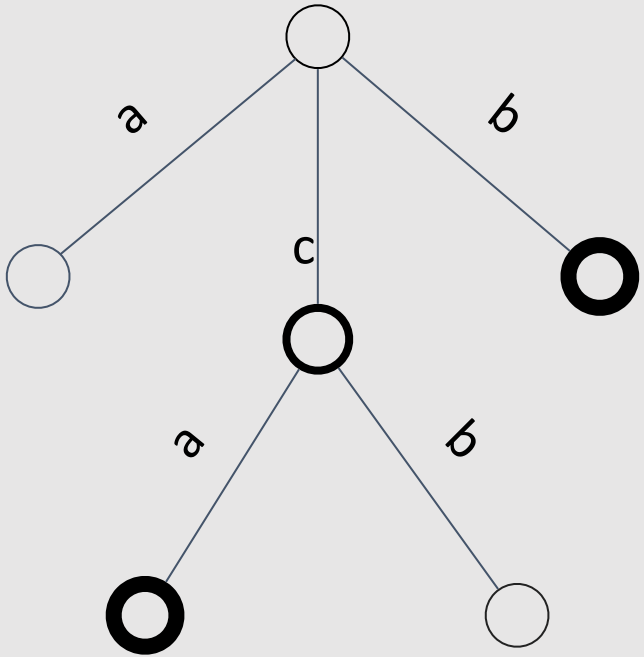


Terminal vertices

We will consider that some vertices are marked as terminal.

Words that end at terminal vertices will be considered words from the dictionary.

set = {b, ca, c}



What we can now?

Now we are able to check if a string is in the set, add it to the set, or remove it from the set in O(length(string)) time.

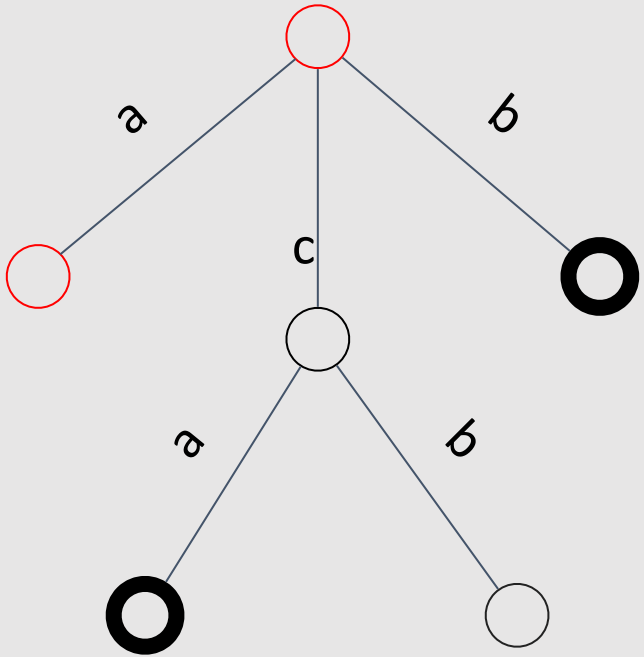
What we can now?

Also, for example, we can find all strings in the set by simply performing a DFS (depth-first search) on the trie.

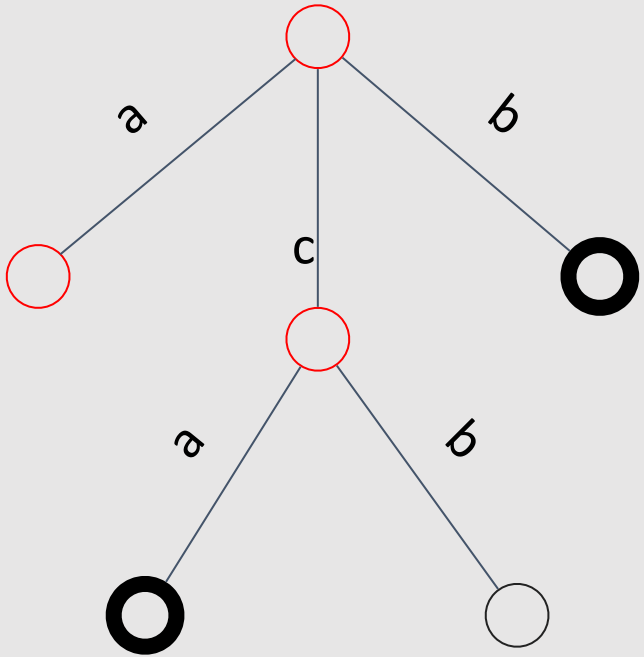
It will works for O(sum_length) time.

a
c
b
a
b

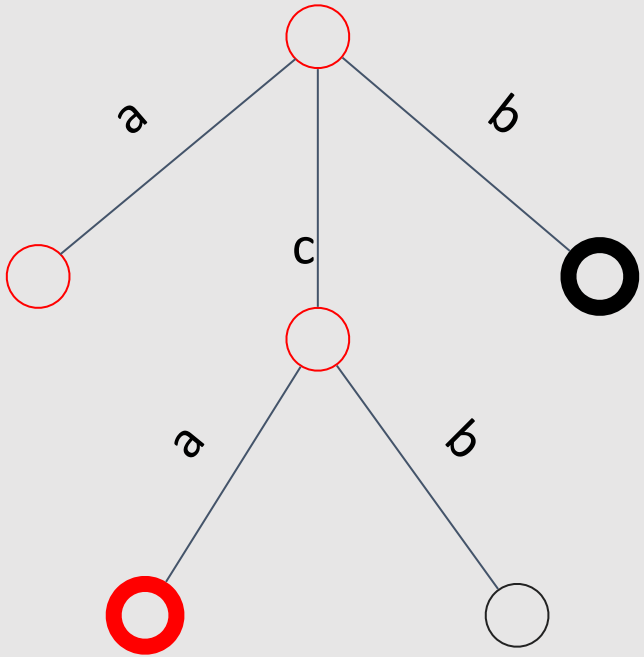
string = a



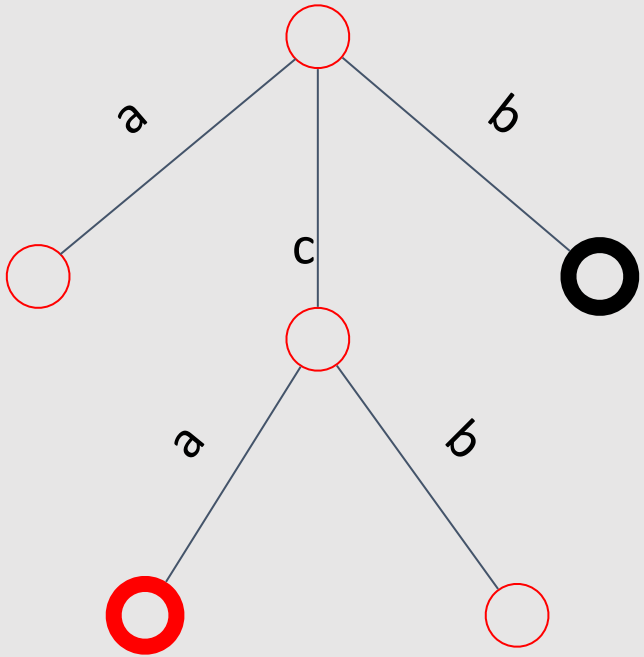
string = c



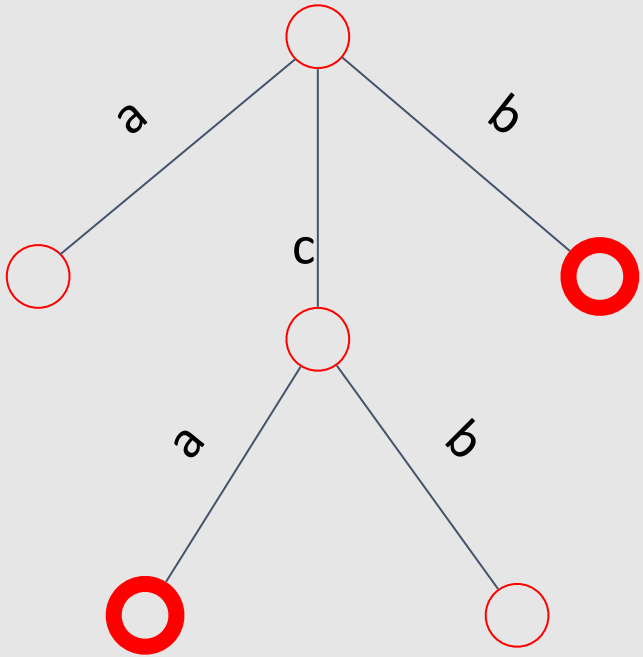
string = ca, cout << ca;



string = cb



string = b, cout << b



How to implement?

How to generally implement such a structure.

We will create a structure vertex, which in some form, will hold pairs {letter, reference to the vertex with an edge on that letter}.

How to implement?

That is, we have an Interface vertex, which has methods like add_edge, which adds a pair {letter, reference to the vertex with an edge on that letter}.

And some method go_by_letter, which takes the necessary pair and returns the vertex for that character.

How to implement?

We need to implement structure, which will storage pairs {letter, reference to the vertex with the edge by that letter}.

There are two ways to do this:

1) If the alphabet is small enough and we know it, then we can just make an array of edges.
2) Otherwise, we can create a dictionary {char, reference}.

How to implement?

For such structures representing tree structures, there are two main solutions:

1. Pointers or references in the case of higher-level languages (I honestly won't write a destructor in C++ code)
2. A vector of structures

We will consider each of them and then discuss their advantages and disadvantages.

```cpp
// If we have only english letters
const int ALPHA_LENGTH = 26;

struct Vertex {
    Vertex* step_by_letter[ALPHA_LENGTH] = {};
    bool is_terminal;
    Vertex() {
        for (int i = 0; i < ALPHA_LENGTH; i++) {
            step_by_letter[i] = nullptr;
        }
        is_terminal = false;
    }

    Vertex* add_edge(char letter) {
        auto new_node = new Vertex();
        step_by_letter[letter - 'a'] = new_node;
        return new_node;
    }

    Vertex* go_by_letter(char letter) {
        return step_by_letter[letter - 'a'];
    }
};
```

```
29. void add_string(Vertex* root, string s) {
30.     Vertex *cur_node = root;
31.     for (auto letter: s) {
32.         Vertex *next_node = cur_node->go_by_letter(letter);
33.         if (next_node) {
34.             cur_node = next_node;
35.         }
36.         else {
37.             cur_node = cur_node->add_edge(letter);
38.         }
39.     }
40.     cur_node->is_terminal = true;
41. }
42. 
43. void traverse_trie(Vertex *root, string &cur_string) {
44.     if (root->is_terminal) {
45.         cout << cur_string << "\n";
46.     }
47.     for (char i = 'a'; i <= 'z'; i++) {
48.         auto step = root->go_by_letter(i);
49.         if (step) {
50.             cur_string.push_back(i);
51.             traverse_trie(step, cur_string);
52.             cur_string.pop_back();
53.         }
54.     }
55. }
```

```
58. int main() {
59.     Vertex *root = new Vertex();
60.     string strings[4] = {"aaaba", "abaa", "aaaaa", "aa"};
61.     for (auto &s: strings) {
62.         add_string(root, s);
63.     }
64.     string cur_string = "";
65.     traverse_trie(root, cur_string);
66.     return 0;
67. }
```

Success #stdin #stdout 0.01s 5304KB

stdin

```
Standard input is empty
```

stdout

```
aa
aaaaa
aaaba
abaa
```

```
8.  struct Vertex {
9.      map<char, Vertex*> step_by_letter;
10.     bool is_terminal;
11.     Vertex() {
12.         is_terminal = false;
13.     }
14.
15.     Vertex* add_edge(char letter) {
16.         auto new_node = new Vertex();
17.         step_by_letter[letter] = new_node;
18.         return new_node;
19.     }
20.
21.     Vertex* go_by_letter(char letter) {
22.         // if (step_by_letter.contains(letter)) {
23.         if (step_by_letter.find(letter) != step_by_letter.end()) {
24.             return step_by_letter[letter];
25.         }
26.         return nullptr;
27.     }
28. };
```

```
30. void add_string(Vertex* root, string s) {
31.     Vertex *cur_node = root;
32.     for (auto letter: s) {
33.         Vertex *next_node = cur_node->go_by_letter(letter);
34.         if (next_node) {
35.             cur_node = next_node;
36.         }
37.         else {
38.             cur_node = cur_node->add_edge(letter);
39.         }
40.     }
41.     cur_node->is_terminal = true;
42. }
43. 
44. void traverse_trie(Vertex *root, string &cur_string) {
45.     if (root->is_terminal) {
46.         cout << cur_string << "\n";
47.     }
48.     for (auto [letter, step] : root->step_by_letter) {
49.         cur_string.push_back(letter);
50.         traverse_trie(step, cur_string);
51.         cur_string.pop_back();
52.     }
53. }
```

```
56.  int main() {
57.      Vertex *root = new Vertex();
58.      string strings[4] = {"aaaba", "abaa", "aaaaa", "aa"};
59.      for (auto &s: strings) {
60.          add_string(root, s);
61.      }
62.      string cur_string = "";
63.      traverse_trie(root, cur_string);
64.      return 0;
65.  }
```

Success #stdin #stdout 0s 5312KB

stdin

```
Standard input is empty
```

stdout

```
aa
aaaaa
aaaba
abaa
```

```
10. struct Vertex {
11.     int step_by_letter[ALPHA_LENGTH];
12.     bool is_terminal;
13.     Vertex() {
14.         for (int i = 0; i < ALPHA_LENGTH; i++) {
15.             step_by_letter[i] = 0;
16.         }
17.         is_terminal = false;
18.     }
19. 
20.     int add_edge(char letter) {
21.         step_by_letter[letter - 'a'] = TRIE_SIZE;
22.         TRIE_SIZE++;
23.         return step_by_letter[letter - 'a'];
24.     }
25. 
26.     int go_by_letter(char letter) {
27.         return step_by_letter[letter - 'a'];
28.     }
29. };
```

```
vector<Vertex> Trie(100);
void add_string(int root_index, string s) {
    int cur_node_id = root;
    for (auto letter: s) {
        int next_node = Trie[cur_node_id].go_by_letter(letter);
        if (next_node) {
            cur_node_id = next_node;
        }
        else {
            cur_node_id = Trie[cur_node_id].add_edge(letter);
        }
    }
    Trie[cur_node_id].is_terminal = true;
}

void traverse_trie(int root_index, string &cur_string) {
    if (Trie[root_index].is_terminal) {
        cout << cur_string << "\n";
    }
    for (char i = 'a'; i <= 'z'; i++) {
        auto step = Trie[root_index].go_by_letter(i);
        if (step) {
            cur_string.push_back(i);
            traverse_trie(step, cur_string);
            cur_string.pop_back();
        }
    }
}
```

```
61.  int main() {
62.      string strings[4] = {"aaaba", "abaa", "aaaaa", "aa"};
63.      for (auto &s: strings) {
64.          add_string(root, s);
65.      }
66.      string cur_string = "";
67.      traverse_trie(root, cur_string);
68.      return 0;
69.  }
```

Success #stdin #stdout 0.01s 5280KB

stdin

```
Standard input is empty
```

stdout

```
aa
aaaaa
aaaba
abaa
```

```
10.  struct Vertex {
11.      map<char, int> step_by_letter;
12.      bool is_terminal;
13.      Vertex() {
14.          is_terminal = false;
15.      }
16.
17.      int add_edge(char letter) {
18.          step_by_letter[letter] = TRIE_SIZE;
19.          TRIE_SIZE++;
20.          return step_by_letter[letter];
21.      }
22.
23.      int go_by_letter(char letter) {
24.          return step_by_letter[letter];
25.      }
26.  };
```

```
vector<Vertex> Trie(100);
void add_string(int root_index, string s) {
    int cur_node_id = root;
    for (auto letter: s) {
        int next_node = Trie[cur_node_id].go_by_letter(letter);
        if (next_node) {
            cur_node_id = next_node;
        }
        else {
            cur_node_id = Trie[cur_node_id].add_edge(letter);
        }
    }
    Trie[cur_node_id].is_terminal = true;
}

void traverse_trie(int root_index, string &cur_string) {
    if (Trie[root_index].is_terminal) {
        cout << cur_string << "\n";
    }
    for (auto [letter, step]: Trie[root_index].step_by_letter) {
        cur_string.push_back(letter);
        traverse_trie(step, cur_string);
        cur_string.pop_back();
    }
}
```

```
55. int main() {
56.     string strings[4] = {"aaaba", "abaa", "aaaaa", "aa"};
57.     for (auto &s: strings) {
58.         add_string(root, s);
59.     }
60.     string cur_string = "";
61.     traverse_trie(root, cur_string);
62.     return 0;
63. }
```

Success #stdin #stdout 0.01s 5292KB

stdin

```
Standard input is empty
```

stdout

```
aa
aaaaa
aaaba
abaa
```

Comparison

map/array - go by letter

map find - O(log(n))

un_map/dict find - O(1)

array find - O(1)

Comparison

map/array - memory

map find - O(length_of_text)

array find - O(length_of_text * alpha)

Comparison

references and pointer/list and vector

size of (references and pointer) depends

Trie traversal

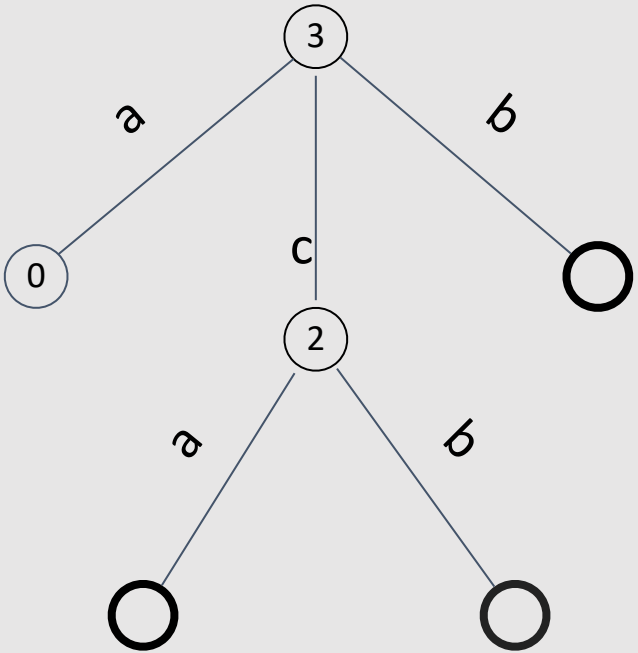
An interesting note is that if we traverse the trie and in that traversal we will check the edges from the smaller letter to the larger, we will pass all the words in lexicographical order. In total, we get a sort for n * alpha, where n is the total length of the text.

k-th string

In fact, now we can solve a huge range of problems. For example, let's find the k-th string in lexicographical order.

To do this, let's calculate additional dp - how many strings end (or how many terminal vertices are in the subtree of vertex i).

set = {b, ca, cb}



k-th string

dp[i] - answer for vertex i.

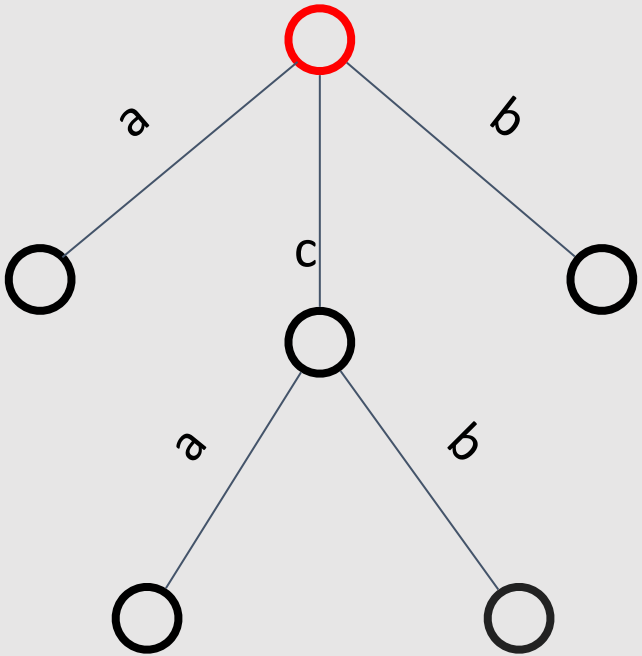
dp[leaf] = is_term(leaf).

calculate in dfs order.

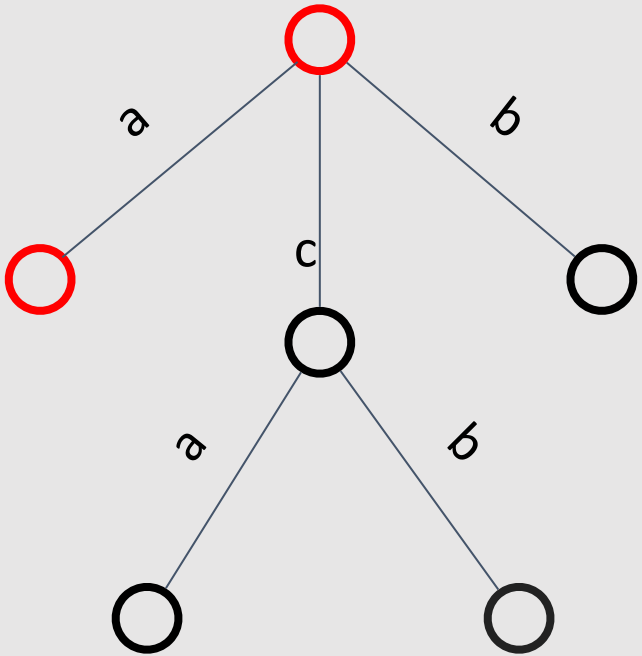
dp[v] = sum(dp[sons]) + is_term(v)

answer - dp[vertex]

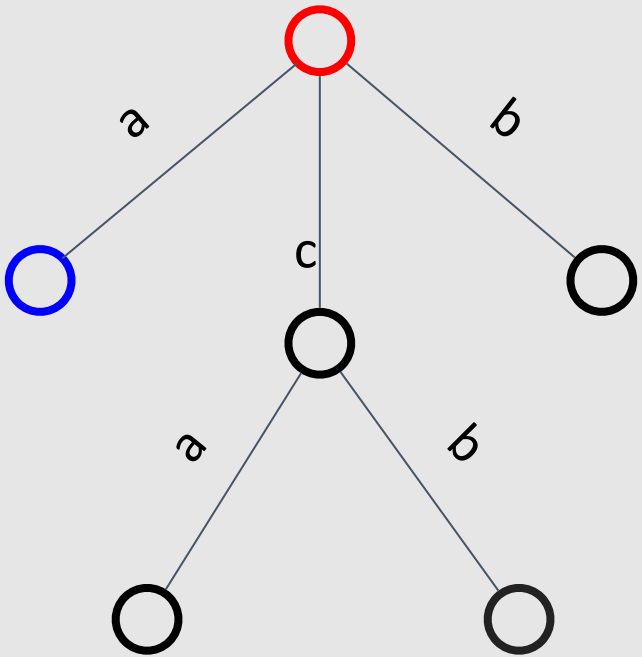
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



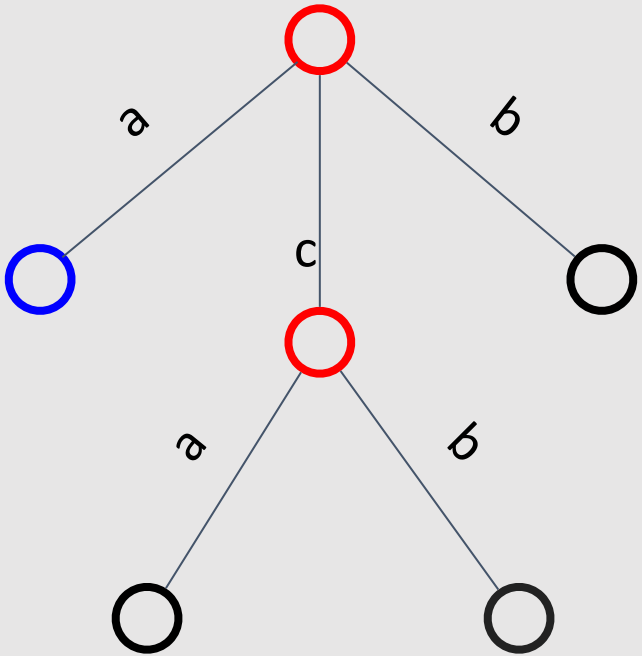
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



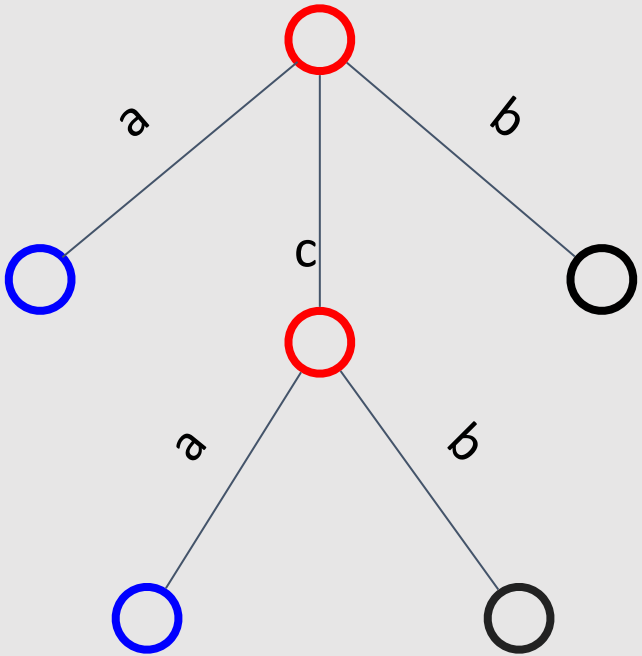
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



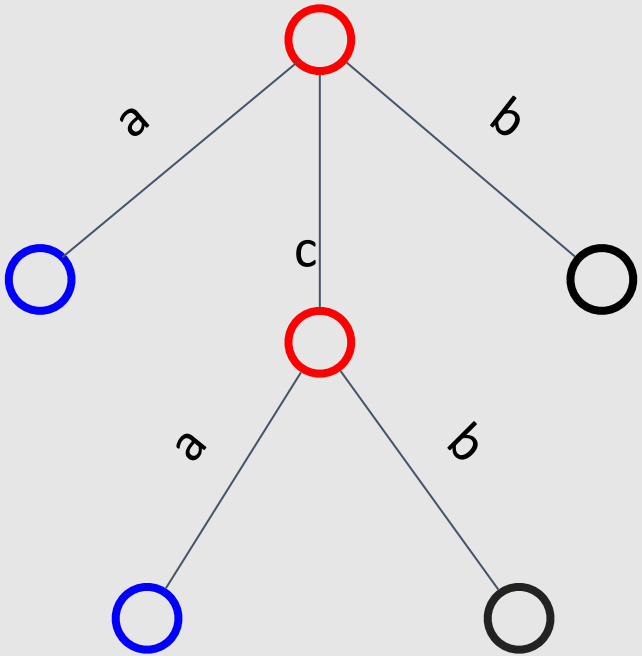
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



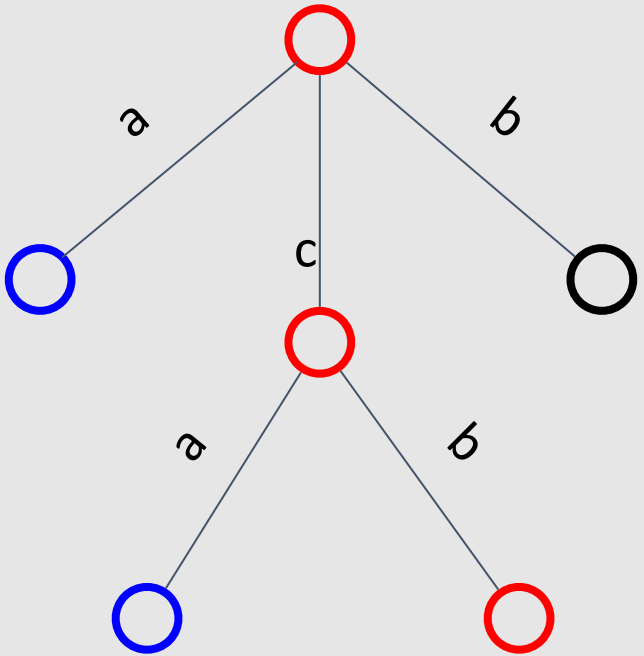
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



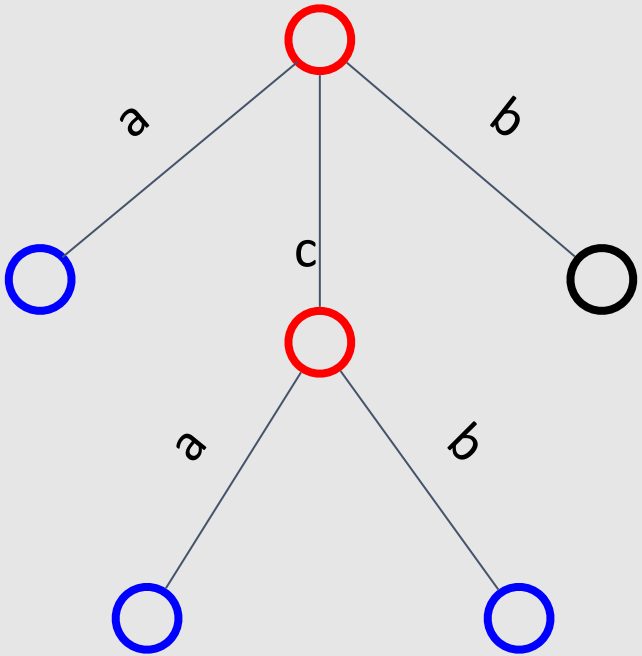
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



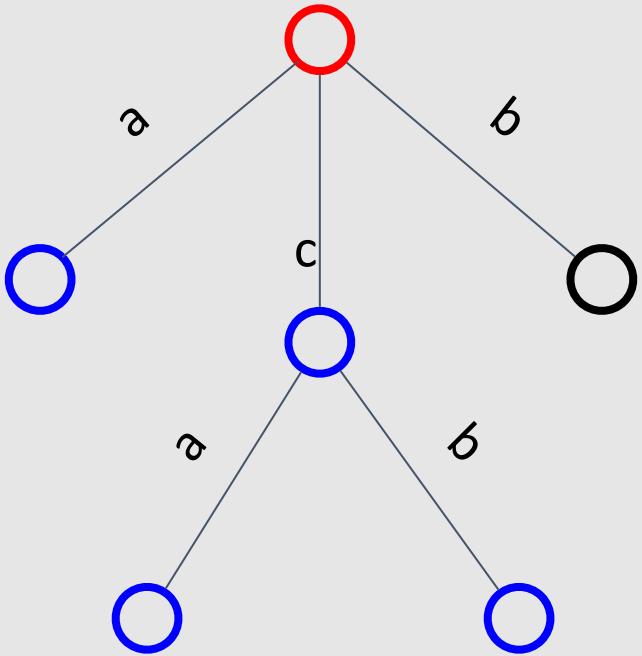
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



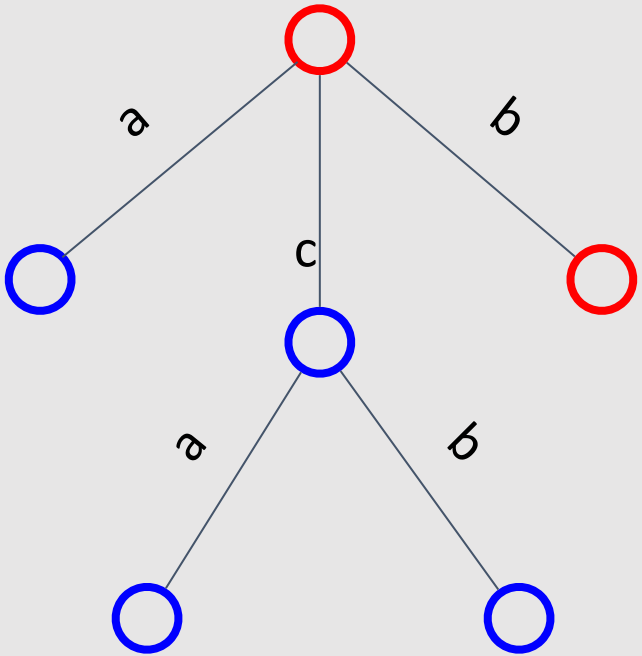
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



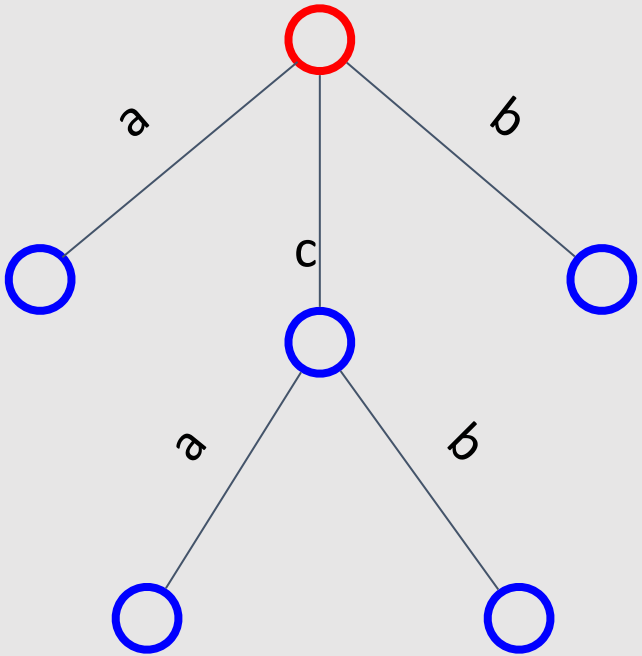
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



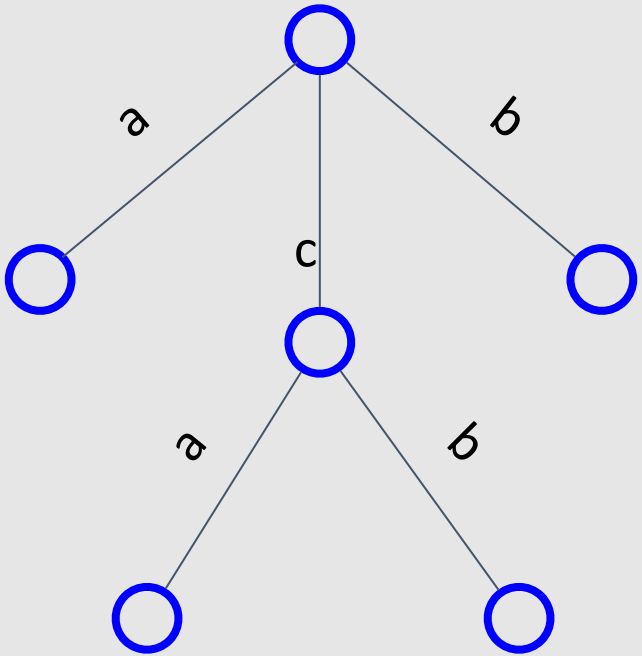
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



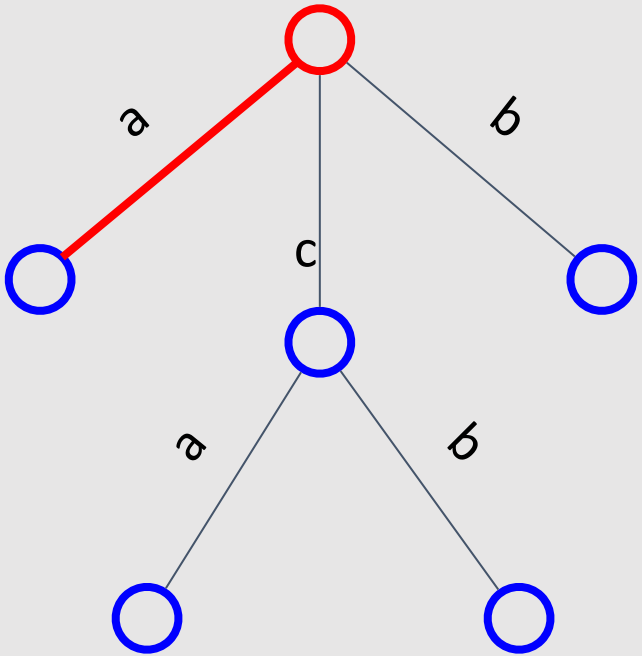
set = {b, ca, cb}, red - active_in_dfs, black - not visited yet, blue - finished



k-th string

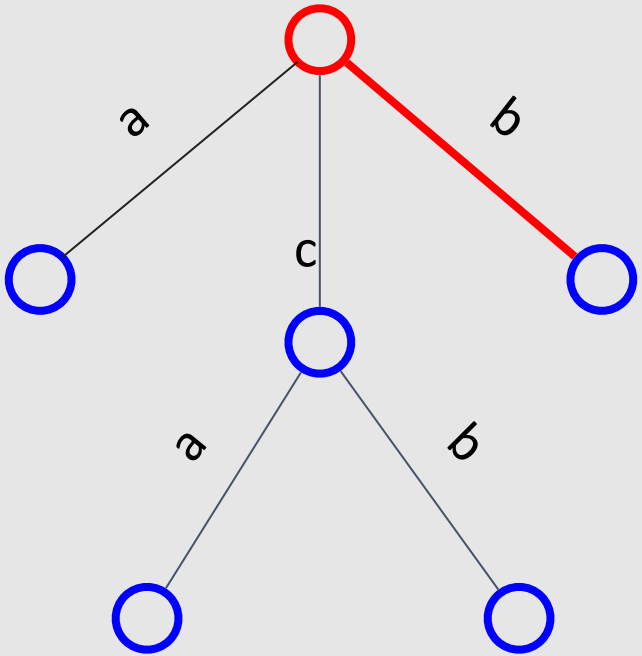
To find the k-th string now, let's check all the transitions at each step and choose the one inside which lies the k-th string. For example, if we need the 1st string (numbering from zero), then we will go to the transition by the letter 'b', as there lies 1 string, but if we need the third one, then we will go by the letter 'c'.

red - active_in_search, red edge, try to go by it, k = 3, amount = 0, 0 < 3 - ignore

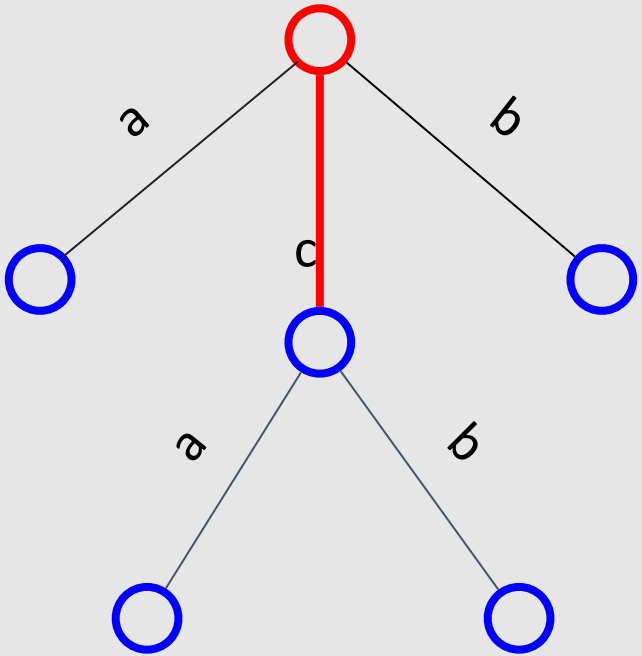


red - active_in_search, red edge, try to go by it, k = 3, amount = 1, 1 < 3 - ignore

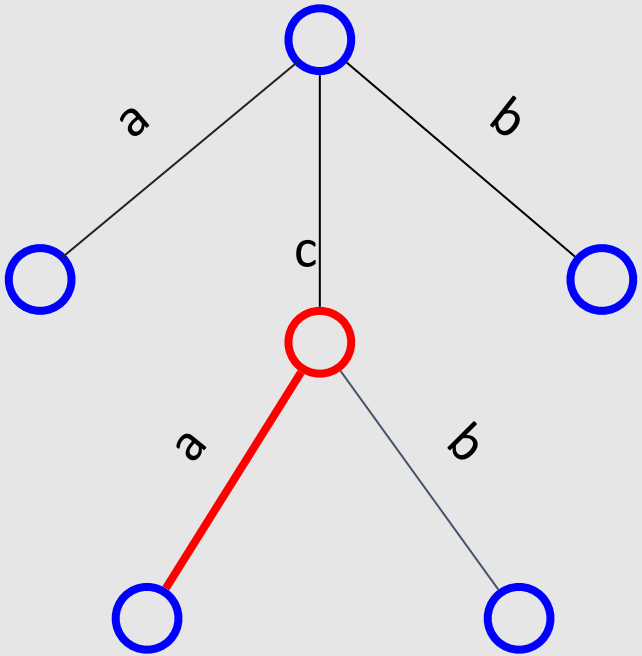
3 - 1 = 2, need to find in next tree



red - active_in_search, red edge, try to go by it, k = 2, amount = 2, 2 >= 2 - go by it



red - active_in_search, red edge, try to go by it, k = 2, amount = 1, 1 < 2
k - 1 = 1, check next tree



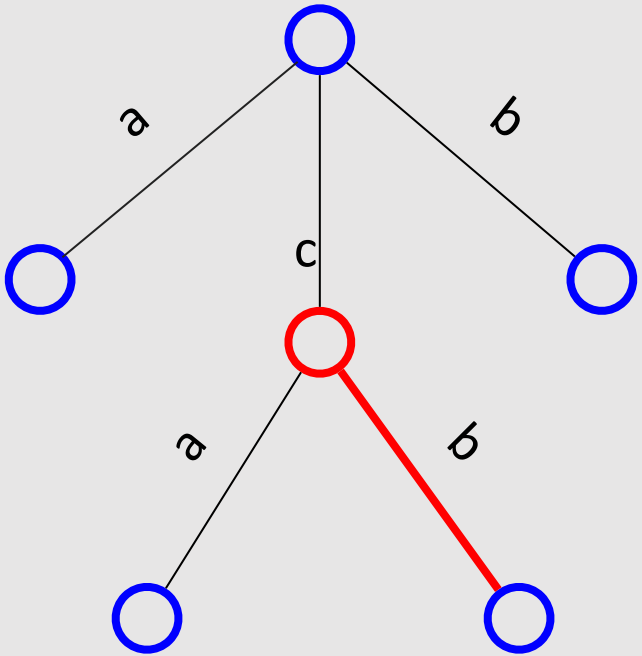
Explanation

so we fant to find second string and we know that it starts from c.

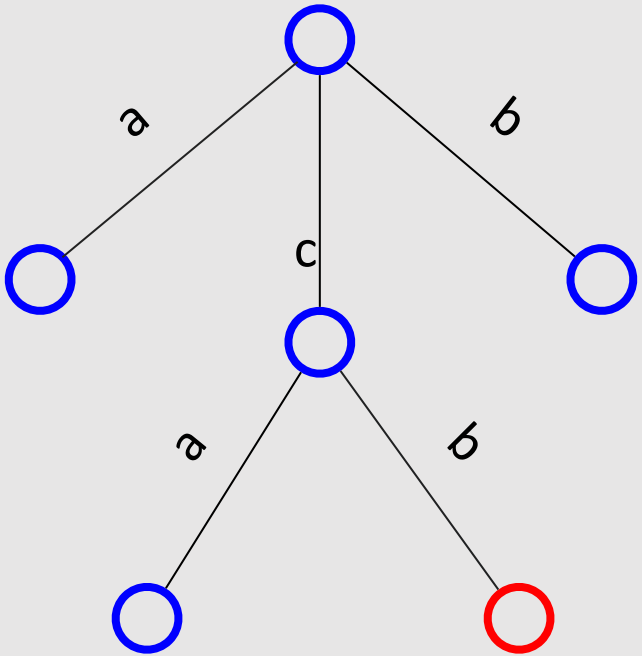
We now that with prefix “ca” we have only one string.

The string we need > “ca” and now we search not for the second string, but for the 2 - 1 string, for the first string, because we have one string with pref “ca” and it’s less.

red - active_in_search, red edge, try to go by it, k = 1, amount = 1, 1 >= 1, go inside



final vertex, answer = cb

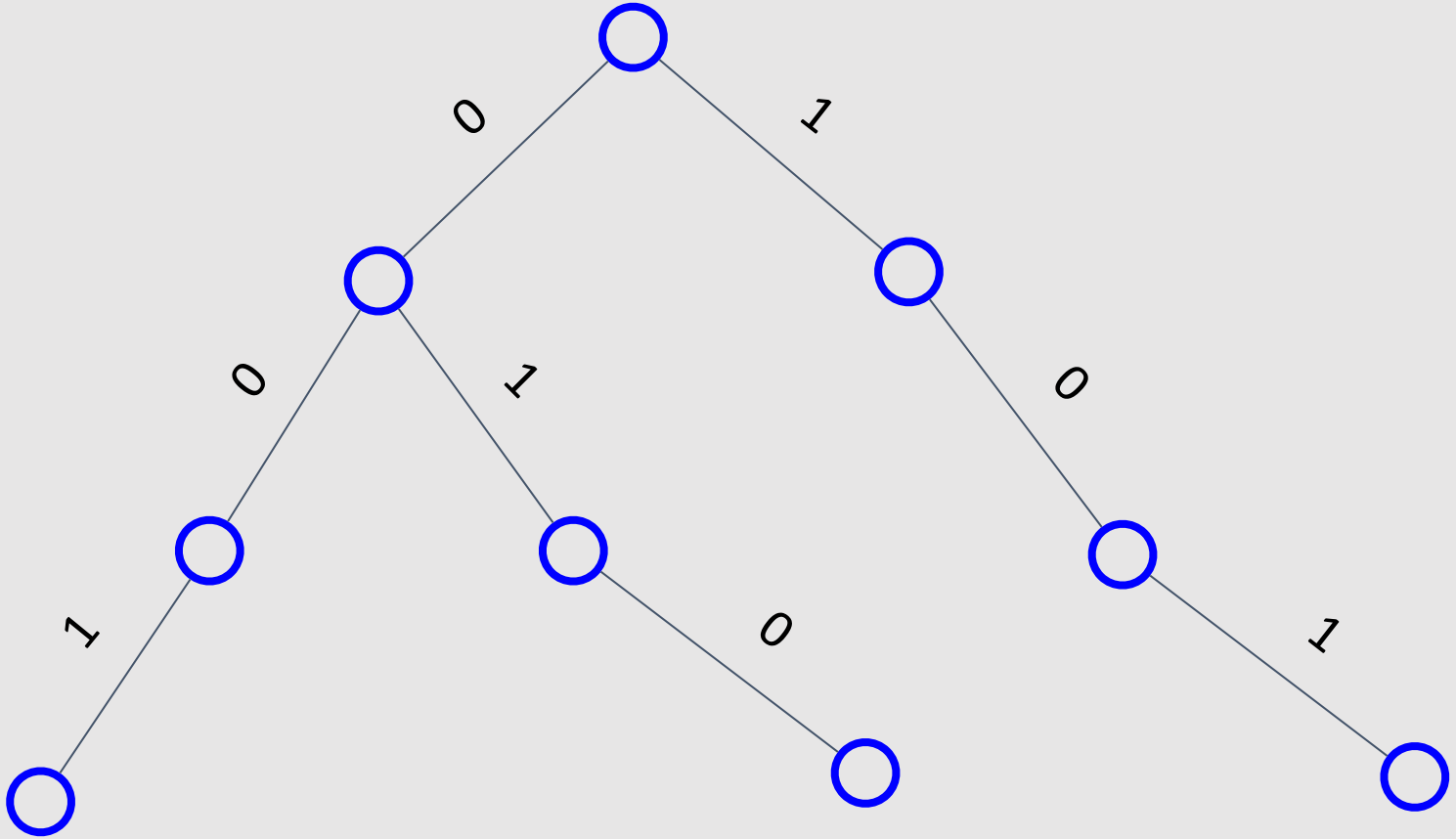


Digit trie

Instead of storing strings in the trie, we can store numbers, and then we can perform very interesting operations with numbers. Usually, in a trie, either numbers in the decimal system or in binary are stored.

Also, it is usually necessary for the numbers to be of equal length, and for this purpose, they are filled with leading zeros.

set = {10, 101, 1}



Task

What kind of problems can be solved in this way?

Usually, it's the problem of searching for some optimal number.

For example - there is a set of numbers, and it is required to respond to two types of queries:

1) Find a number from the set, the xor of which with x_i is maximal
2) Add the number x_i to the set

Example

set = {}

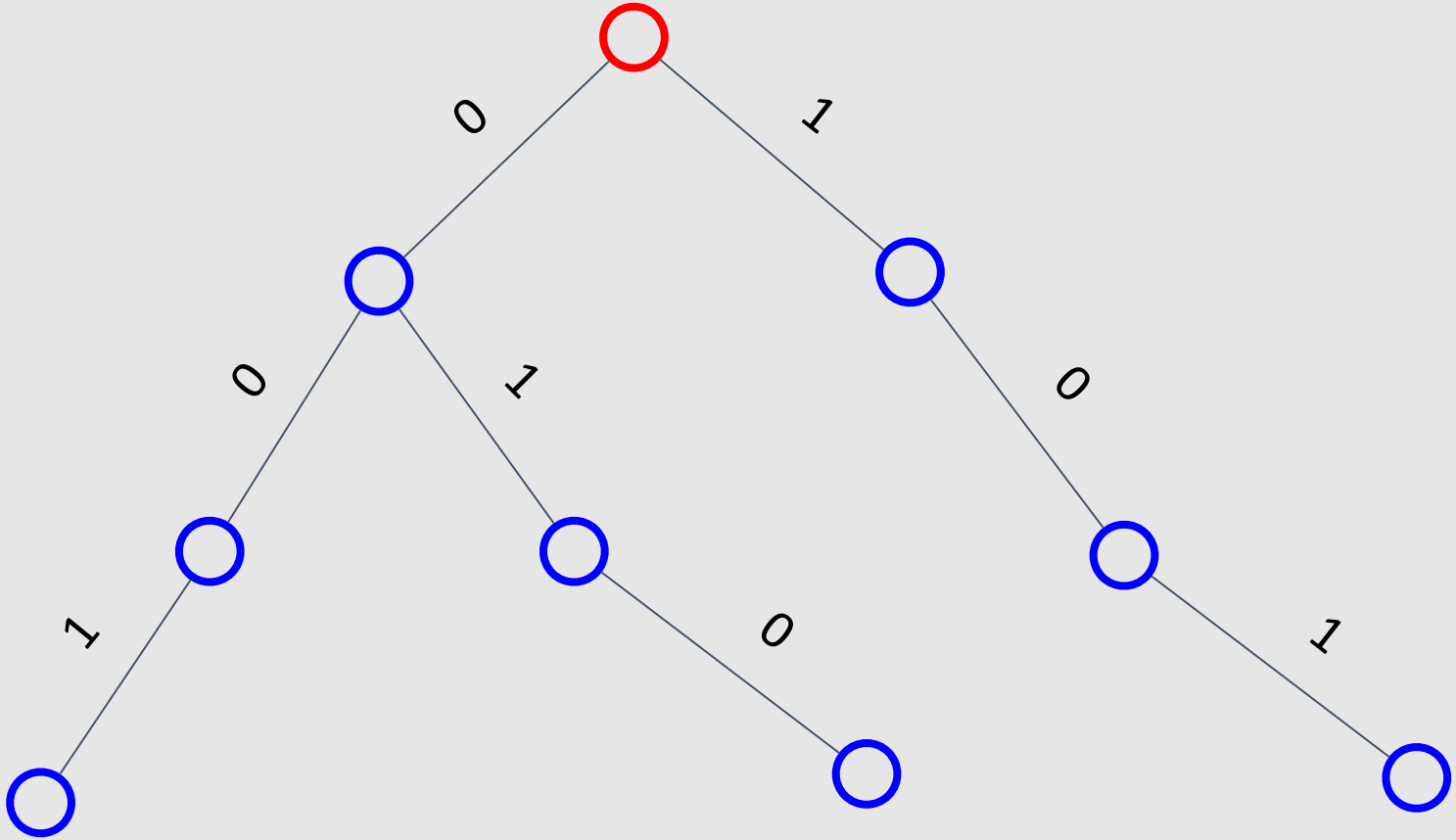
1) add 3 -> set = {3}
2) add 4 -> set ={3, 4}
3) find 5, 4 xor 5 = 1, 3 xor 5 = 6, so 6 is answer
4) add 6 -> set = {3, 4, 6}

Solution

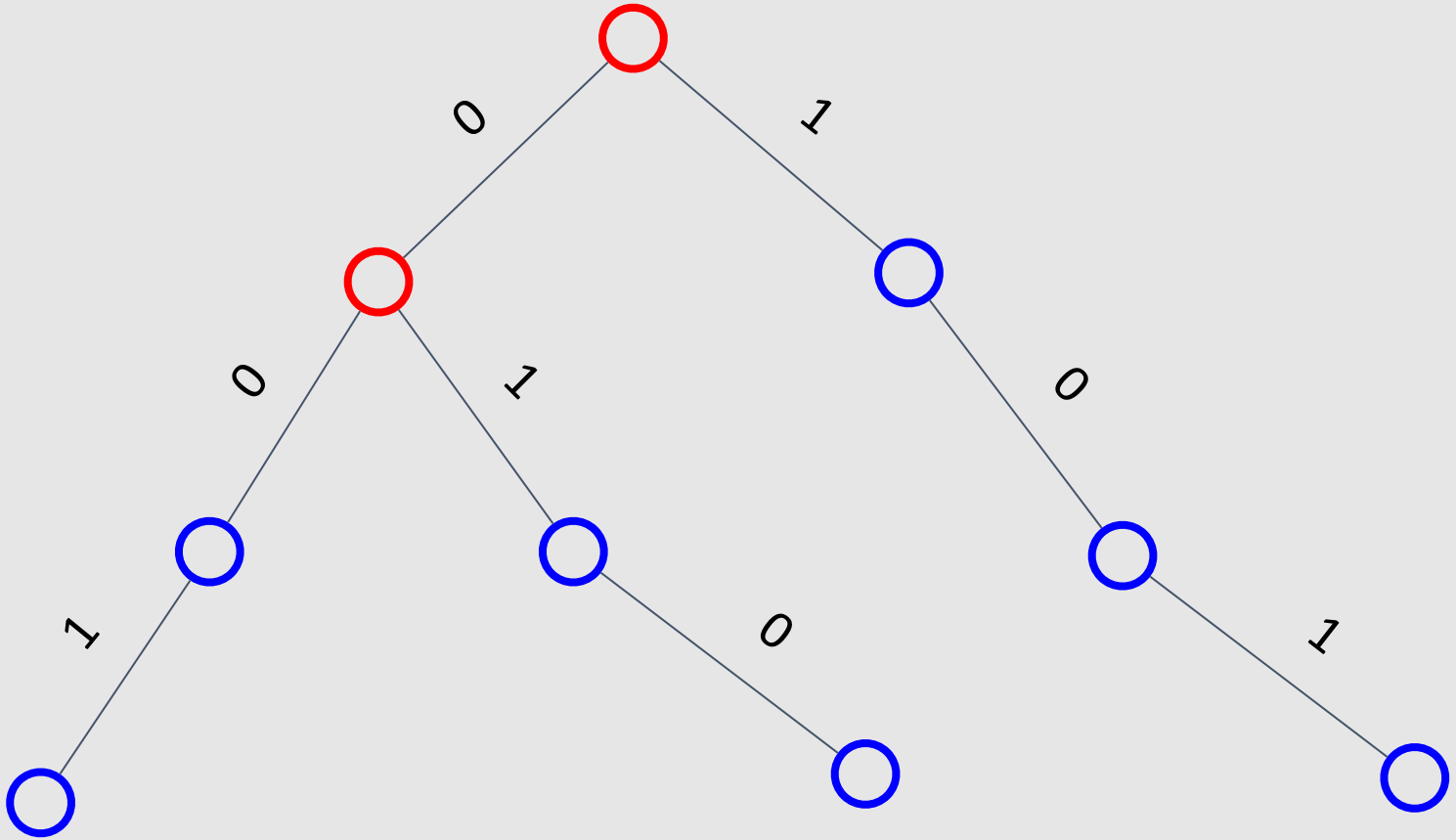
Let's build a trie from a set of numbers. The insertion of a number will work just like the insertion of a string into a trie.

For searching, we can proceed greedily, let's go through the digits of the number x_i and each time try to take the opposite value. If it's not possible, then we take the same value.

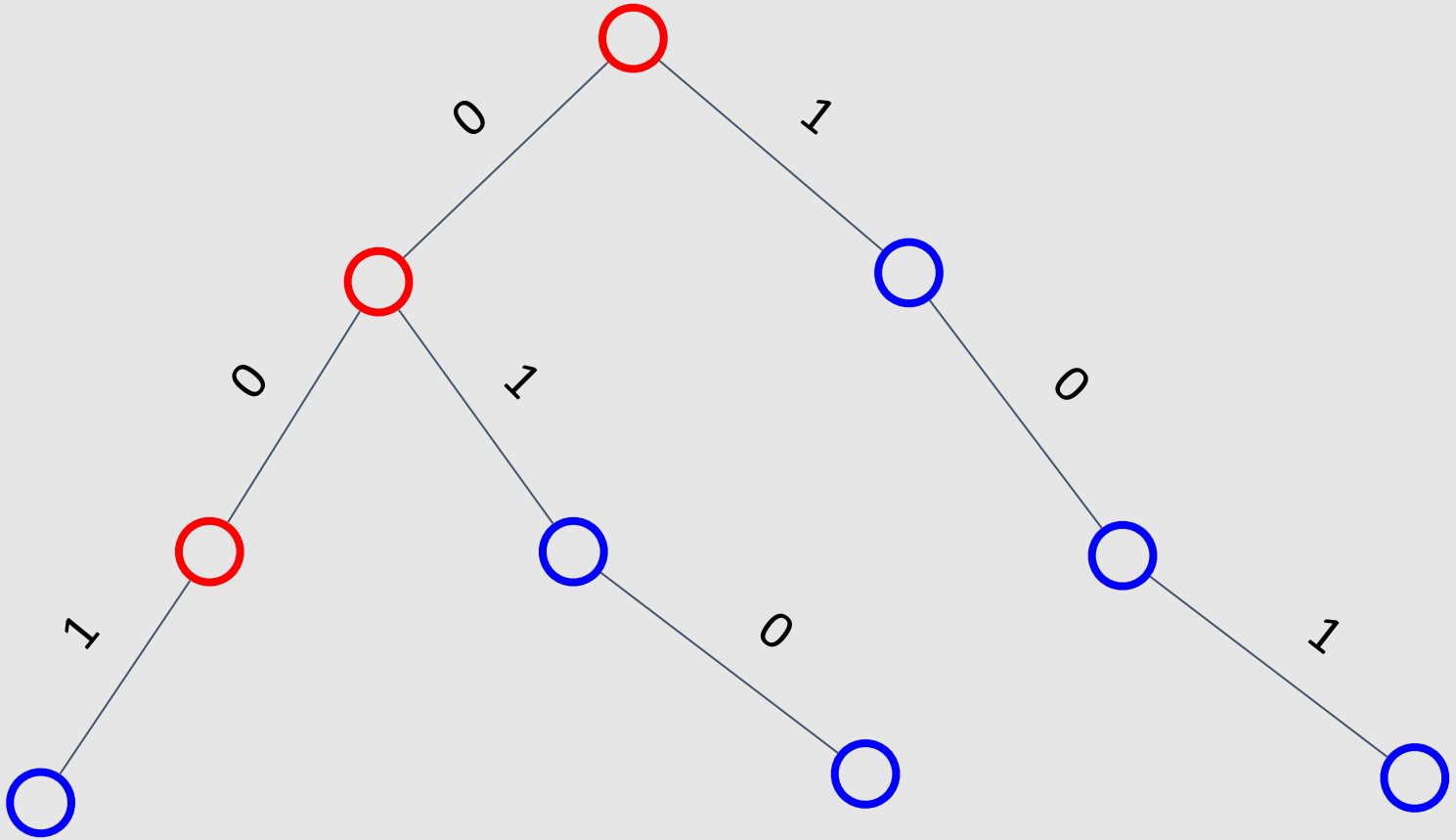
set = {10, 101, 1}, x_i = 111, x_i[0] = 1, try to go by 0



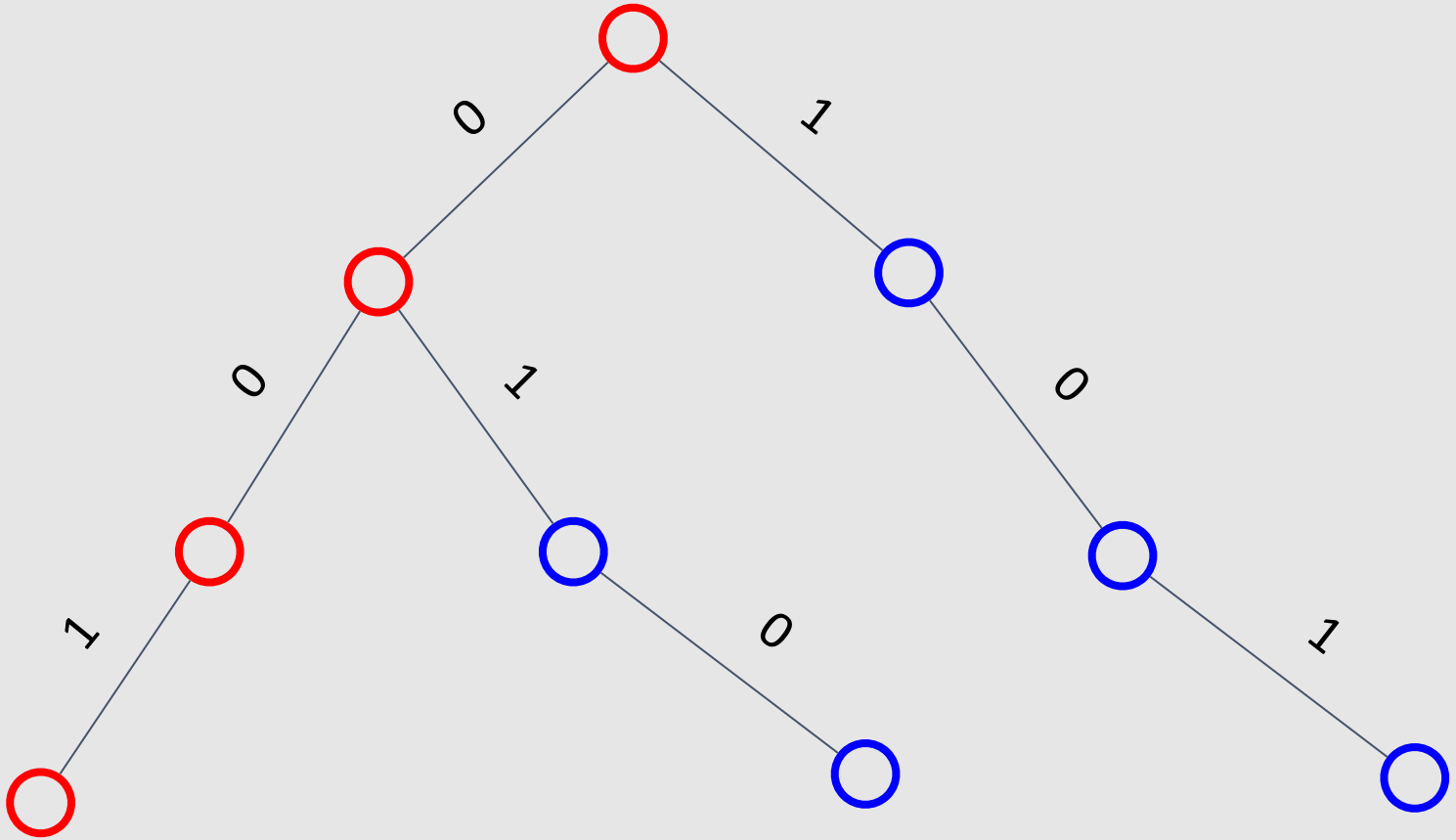
set = {10, 101, 1}, x_i = 111, x_i[1] = 1, try to go by 0



set = {10, 101, 1}, x_i = 111, x_i[2] = 1, try to go by 0, no such edge, go by 1



answer = 001 = 1



Why we can do greedy?

At each step, it is advantageous for us to choose greedily because if we select the opposite value at the i-th step, then we will add 2^(n - i) to the answer. However, if we take all the good values at the following steps, we can only get (2^0 + 2^1 + 2^2 + .. + 2^(n - i - 1)) = 2^(n - i) - 1.

What also?

mex(minimal excluding) - least natural number, that is not in set.

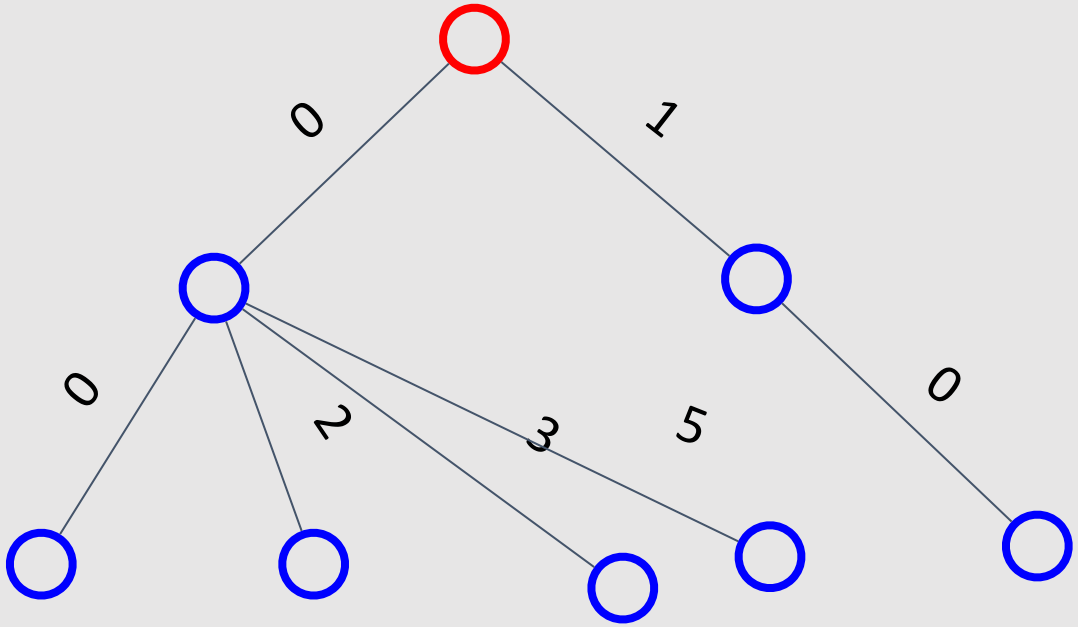
mex({1, 2}) = 0

mex({0, 3}) = 1

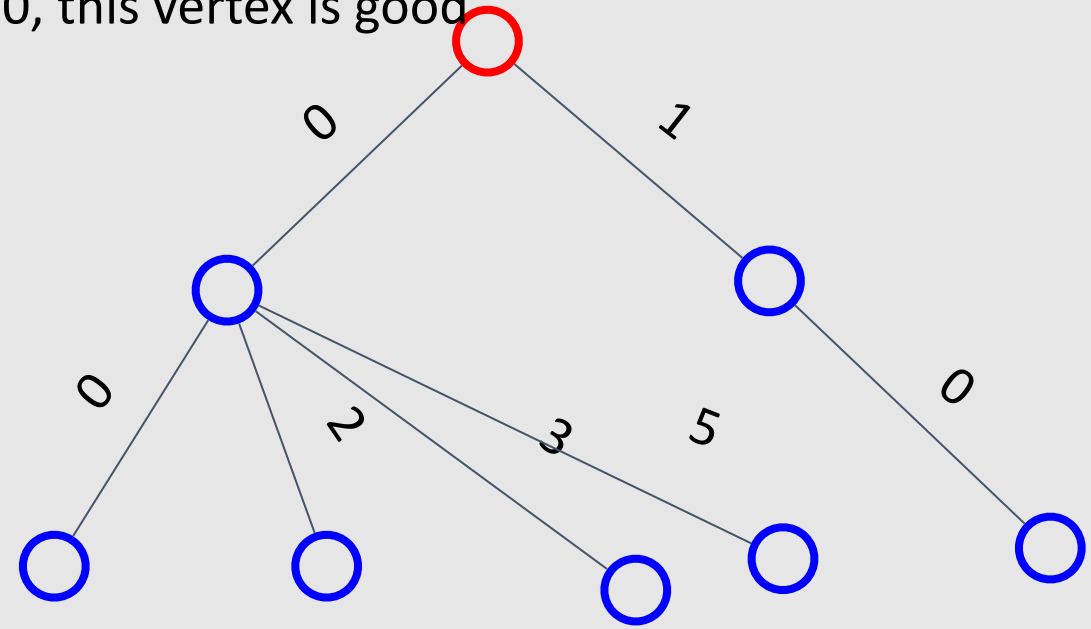
There are given n numbers, you need to find MEX.

Mex <= n

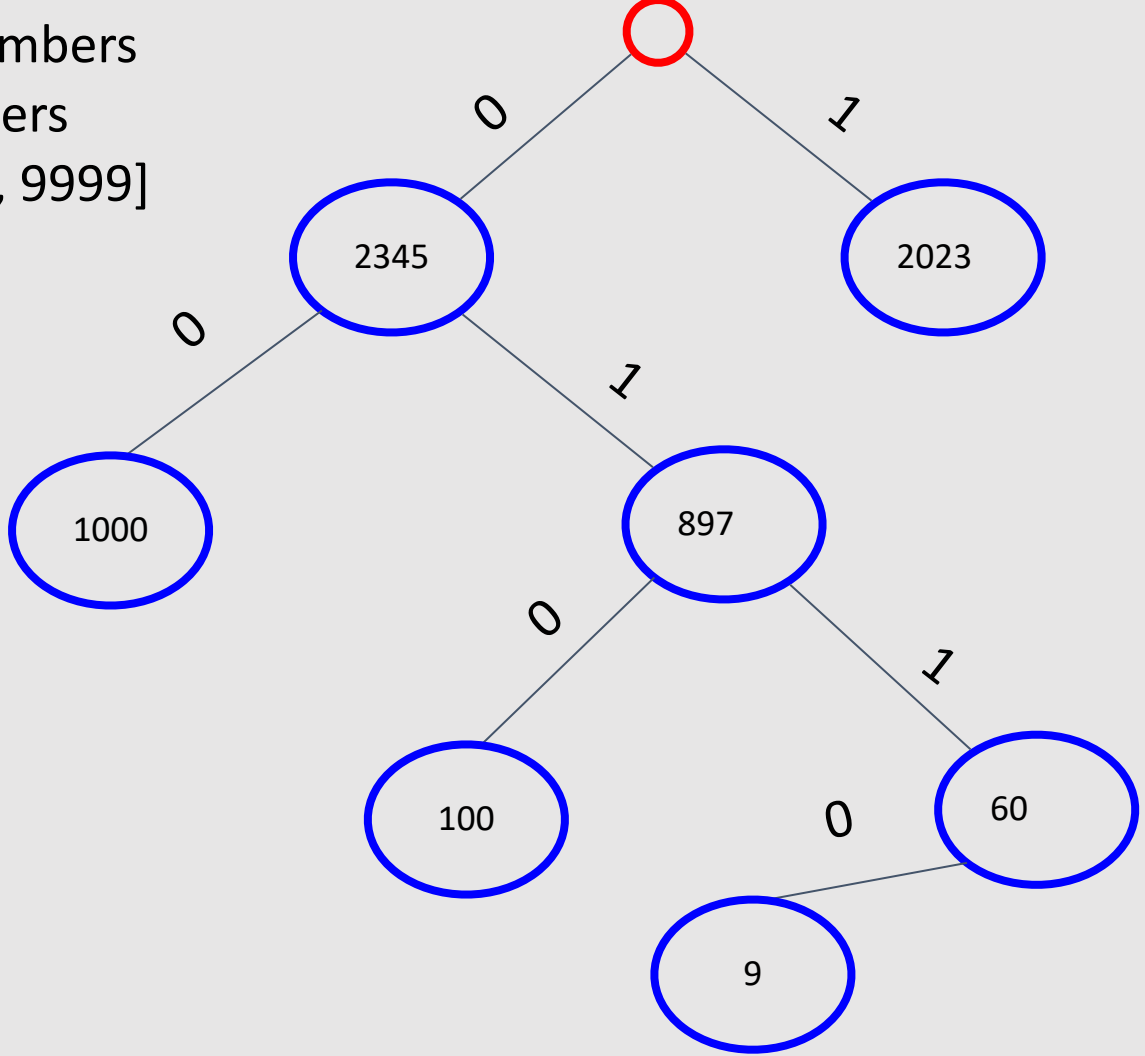
set = {03, 02, 05, 00, 10), i need to find mex, i know that mex is less then 15



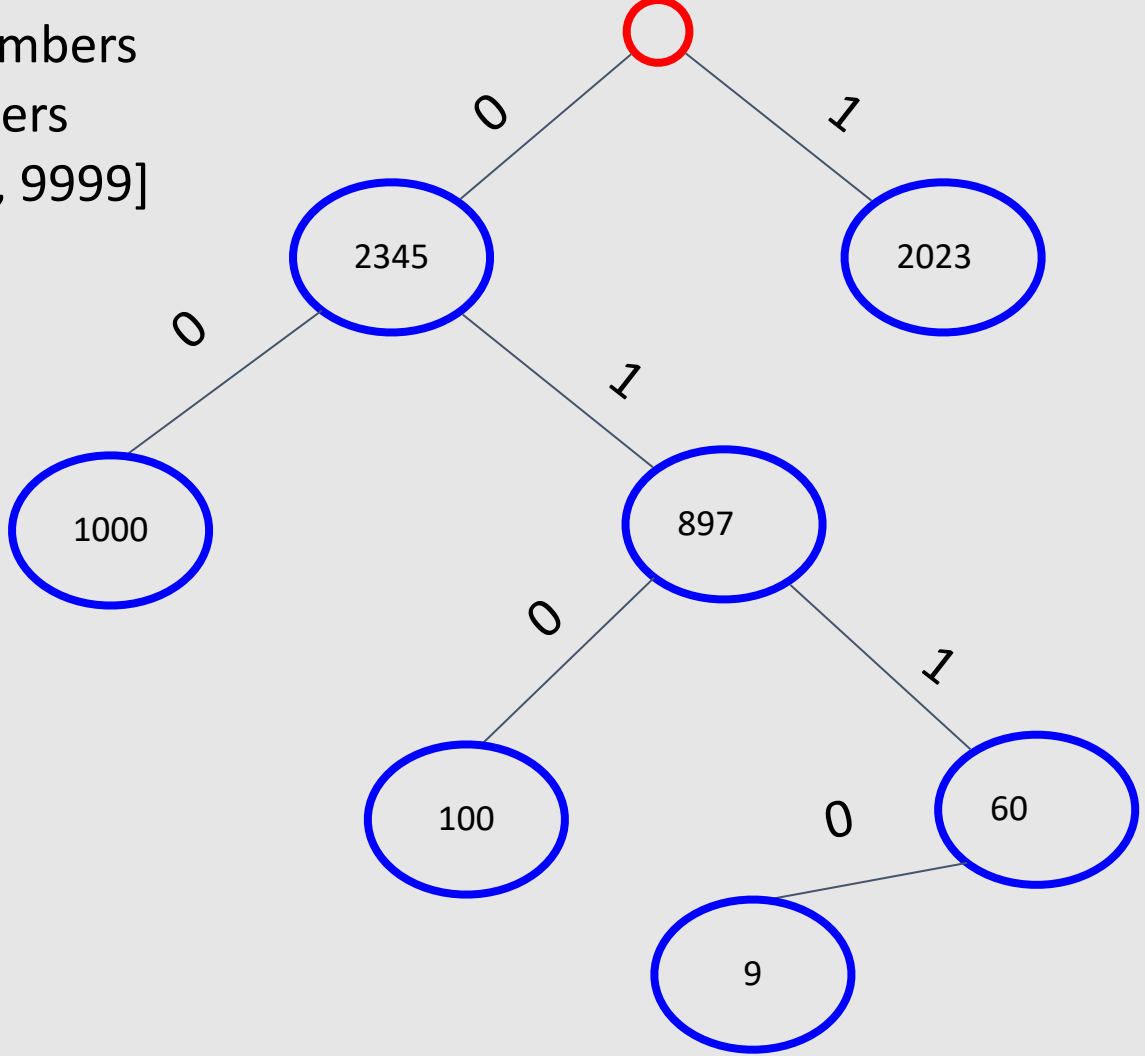
set = {03, 02, 05, 00, 10), i check for the digit (/100 % 10), if the amount of sons in the vertex = 10, this vertex is good
so we can skip



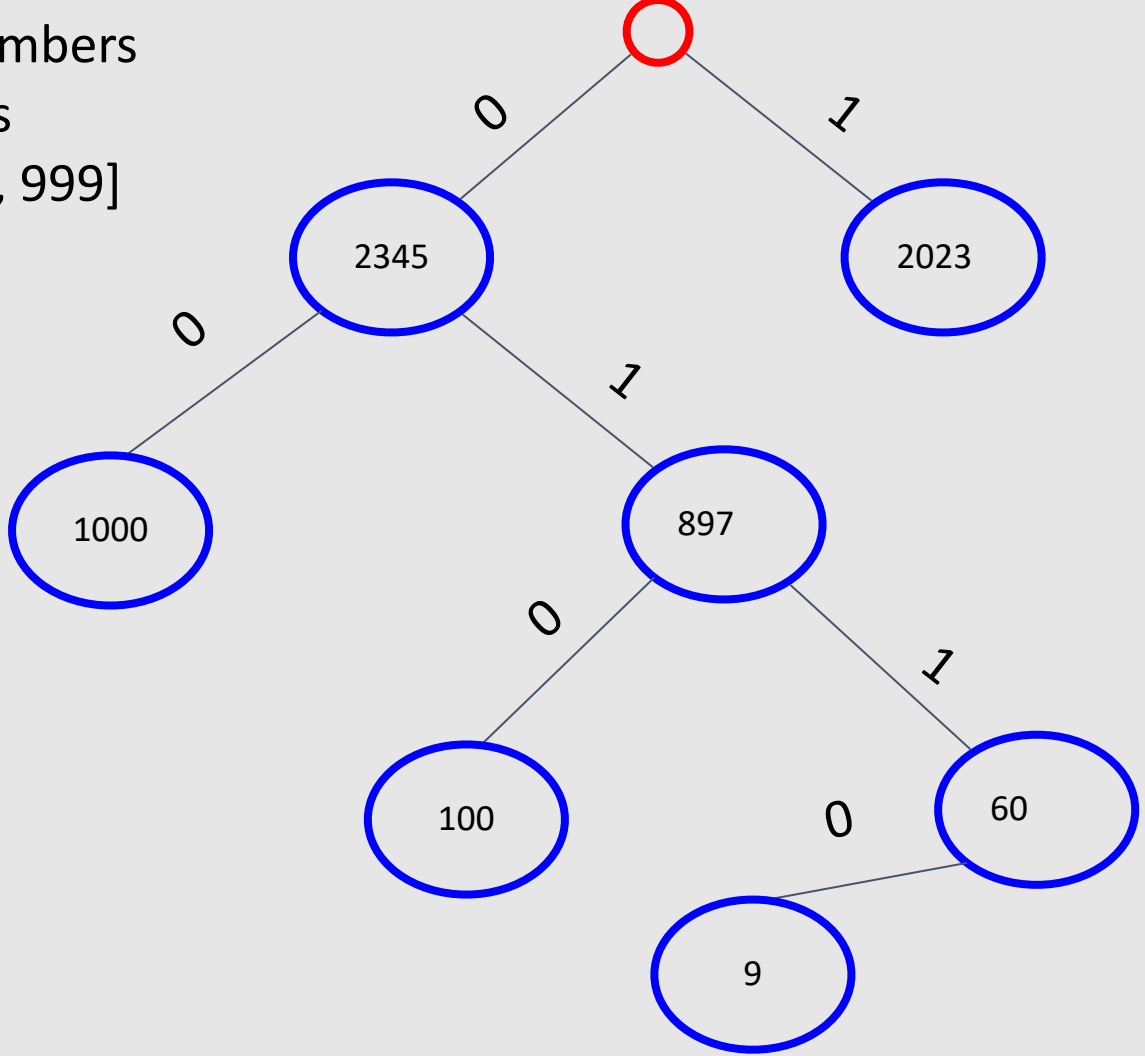
we have 2 * 10^5 numbers
00000 - 10000 numbers
because it’s range [0, 9999]
10000
20000
30000



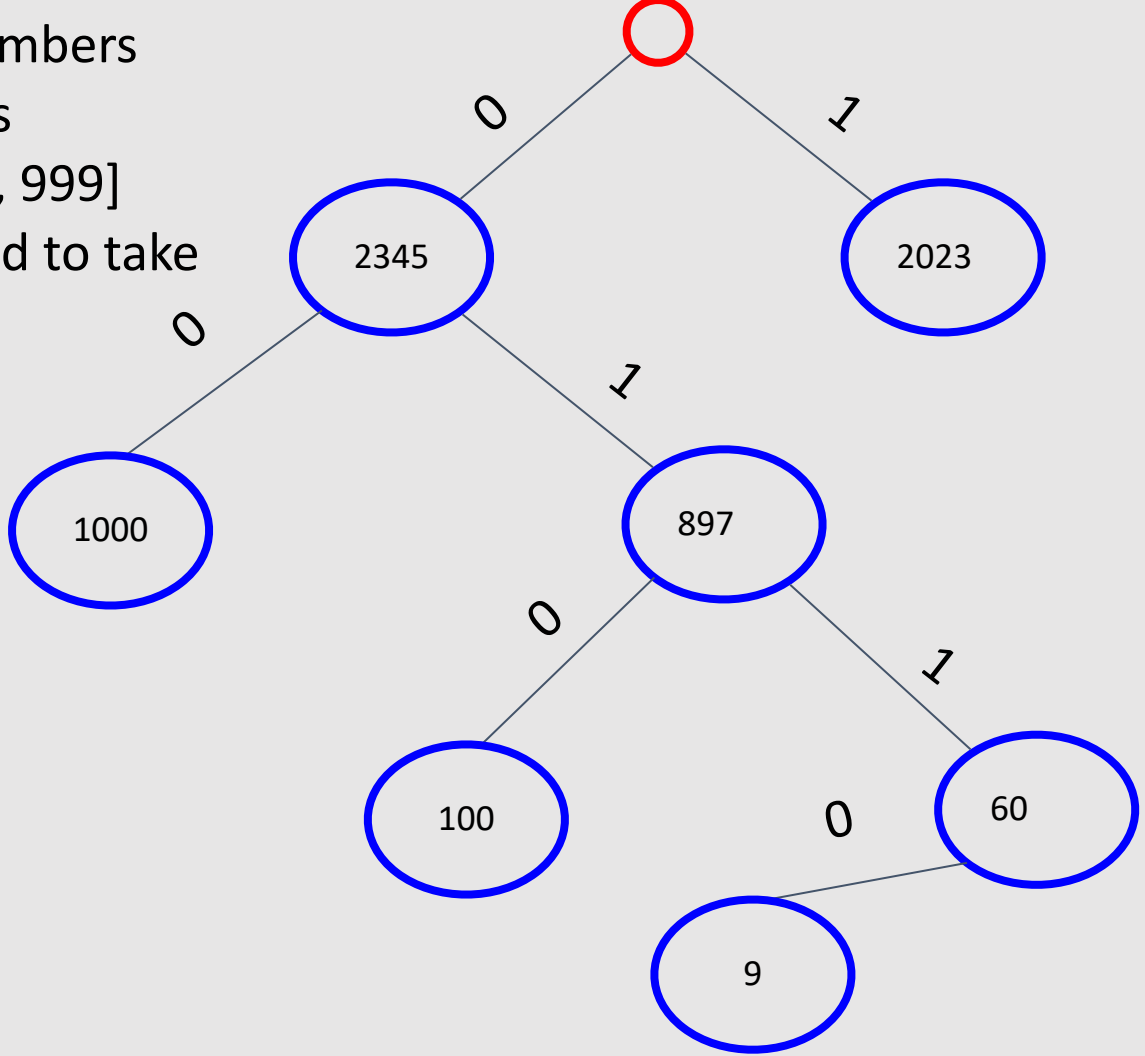
we have 2 * 10^5 numbers
00000 - 10000 numbers
because it’s range [0, 9999]
2345 < 10000
first digit is 0
mex < 10000



we have 2 * 10^5 numbers
0000 - 1000 numbers
because it’s range [0, 999]
1000
2000
3000



we have 2 * 10^5 numbers
0000 - 1000 numbers
because it’s range [0, 999]
1000 = 1000, we need to take bigger
1000
2000
3000

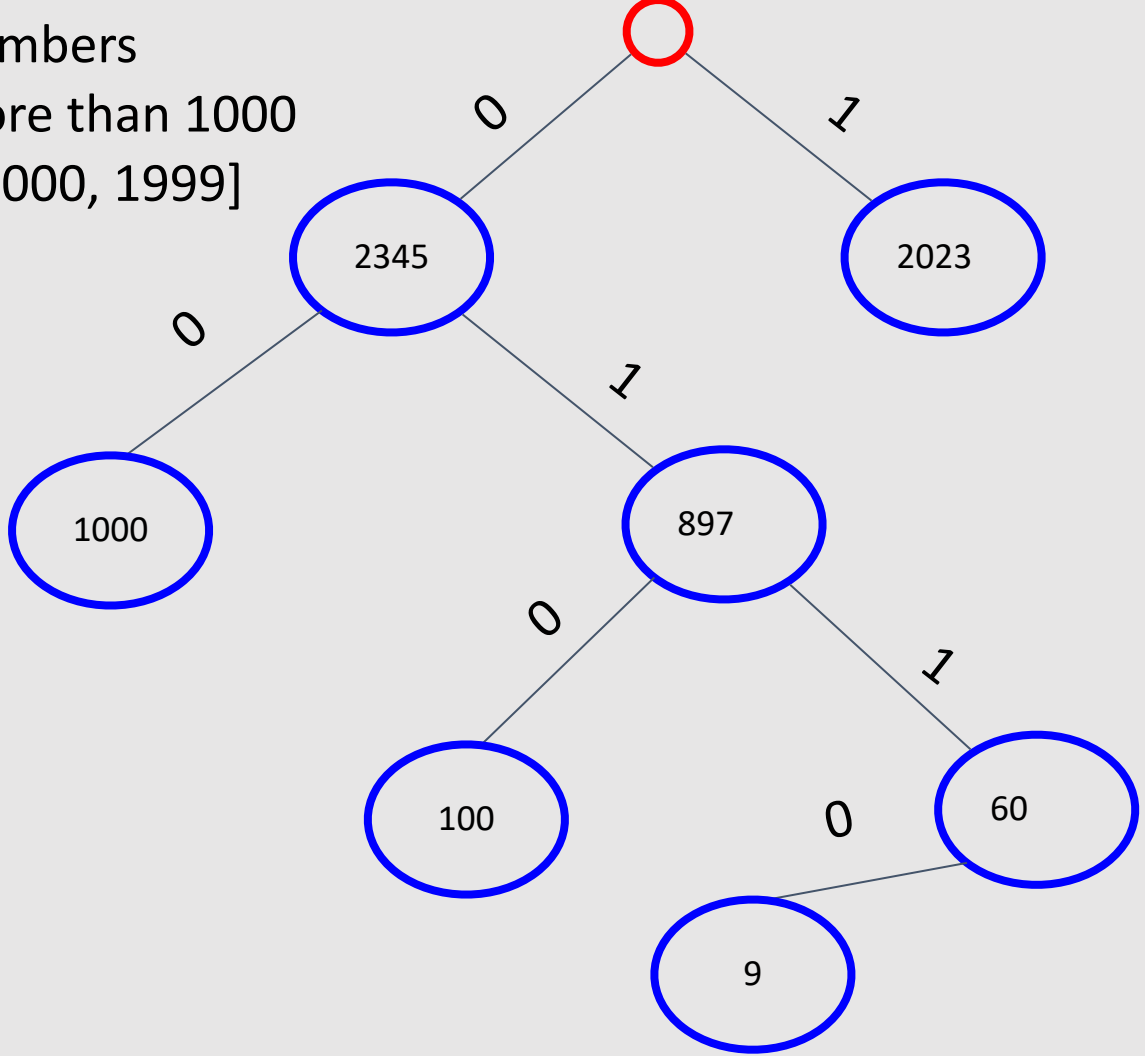


we have 2 * 10^5 numbers

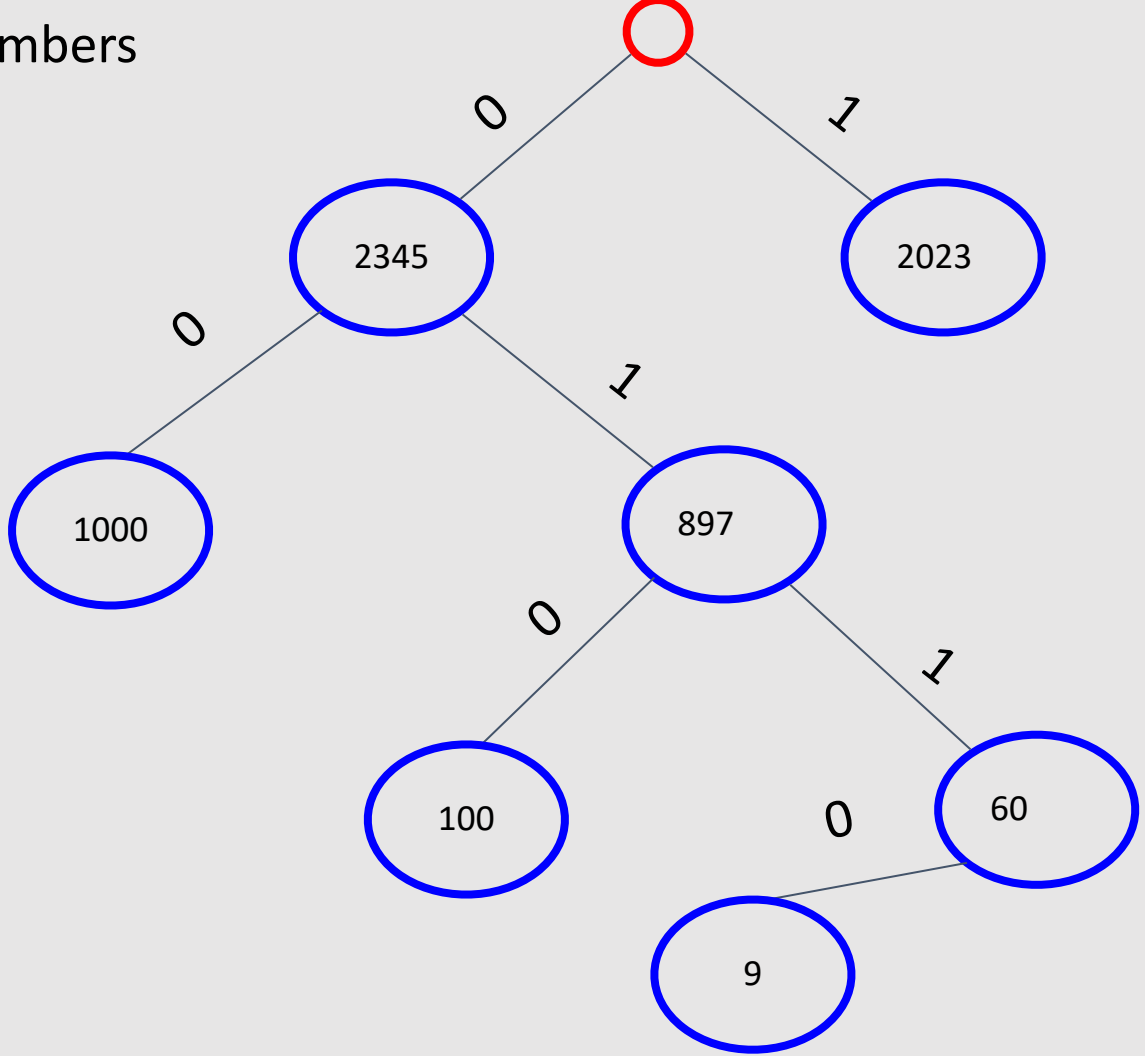
1000 - must keep more than 1000 numbers, because [1000, 1999]

2000

3000



we have 2 * 10^5 numbers
answer start from 01
100
60 < 100
200
300



we have 2 * 10^5 numbers

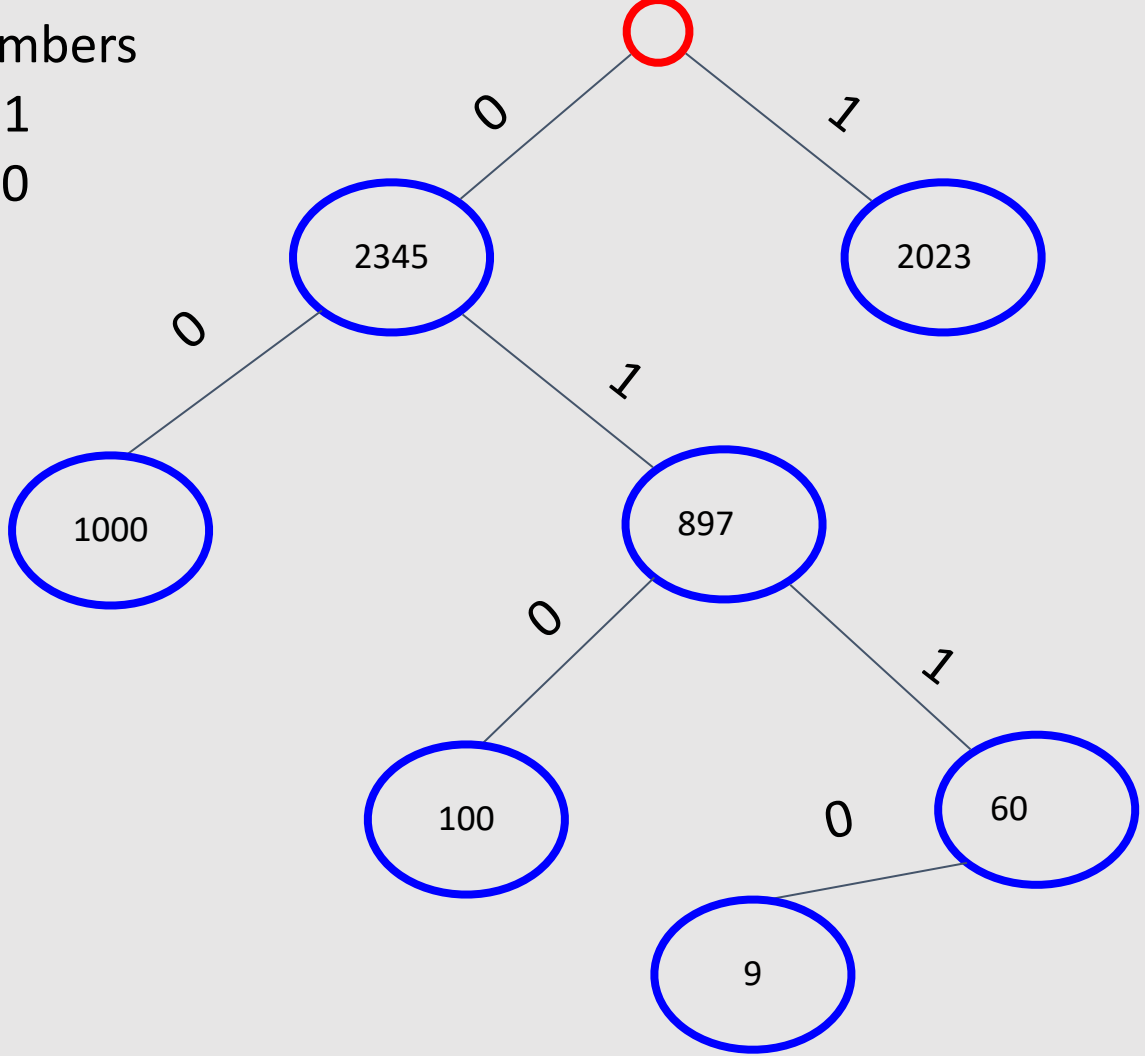
answer start from 011

00 - 9 < 10 -> you go 0

10

20

30



we have 2 * 10^5 numbers

answer start from 0110

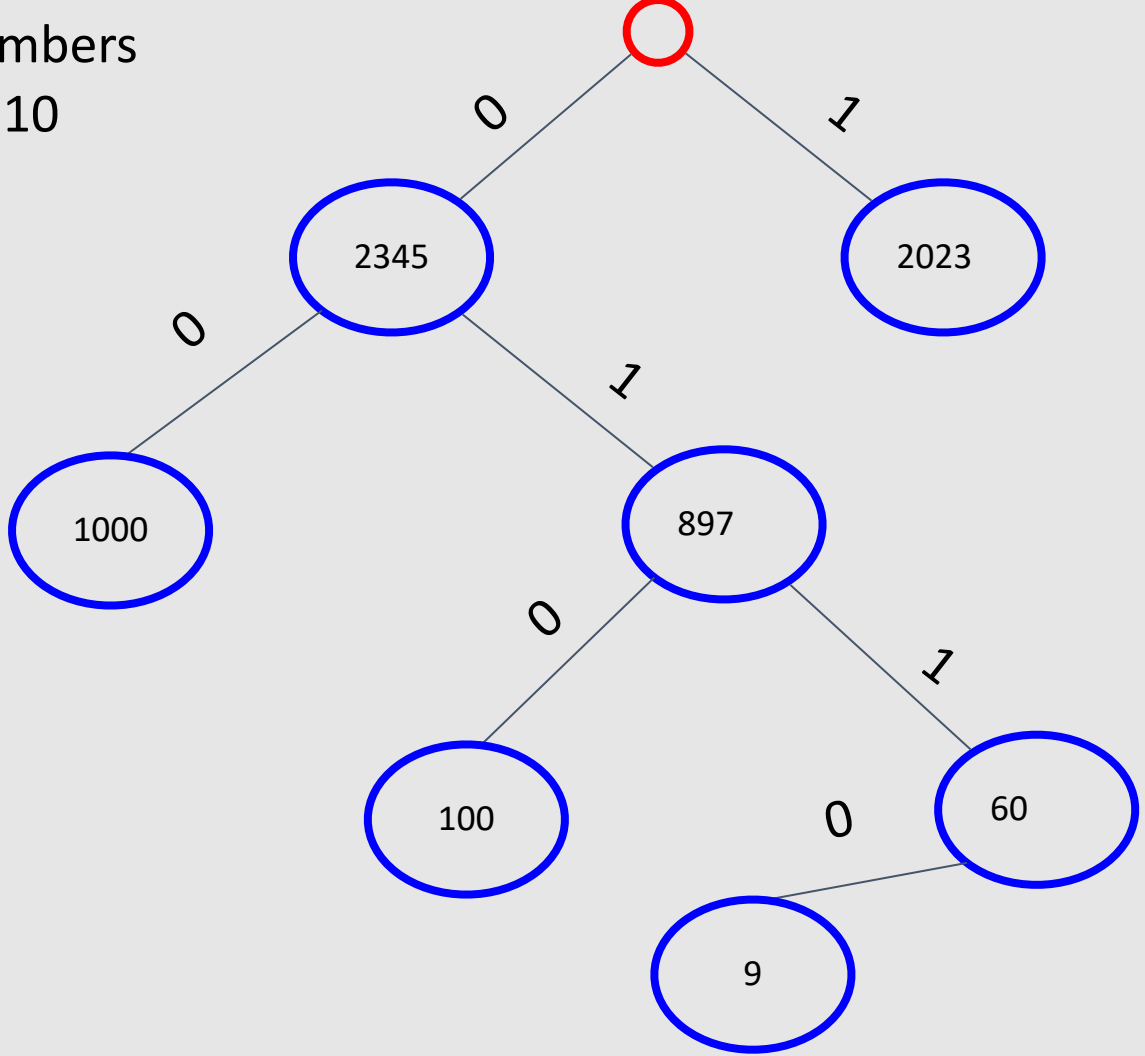
0

1

2

3

4



Aho–Corasick algorithm

Let there be a set of strings {s_1, .. s_n}, called a dictionary, and a large text t. It is necessary to find all positions where the dictionary strings enter the text. For simplicity, let's additionally assume that the strings from the dictionary are not substrings of each other (later we will see that this requirement is redundant).

Example

{s_1, .. s_n} = {“aba”, “da”, “ac”}

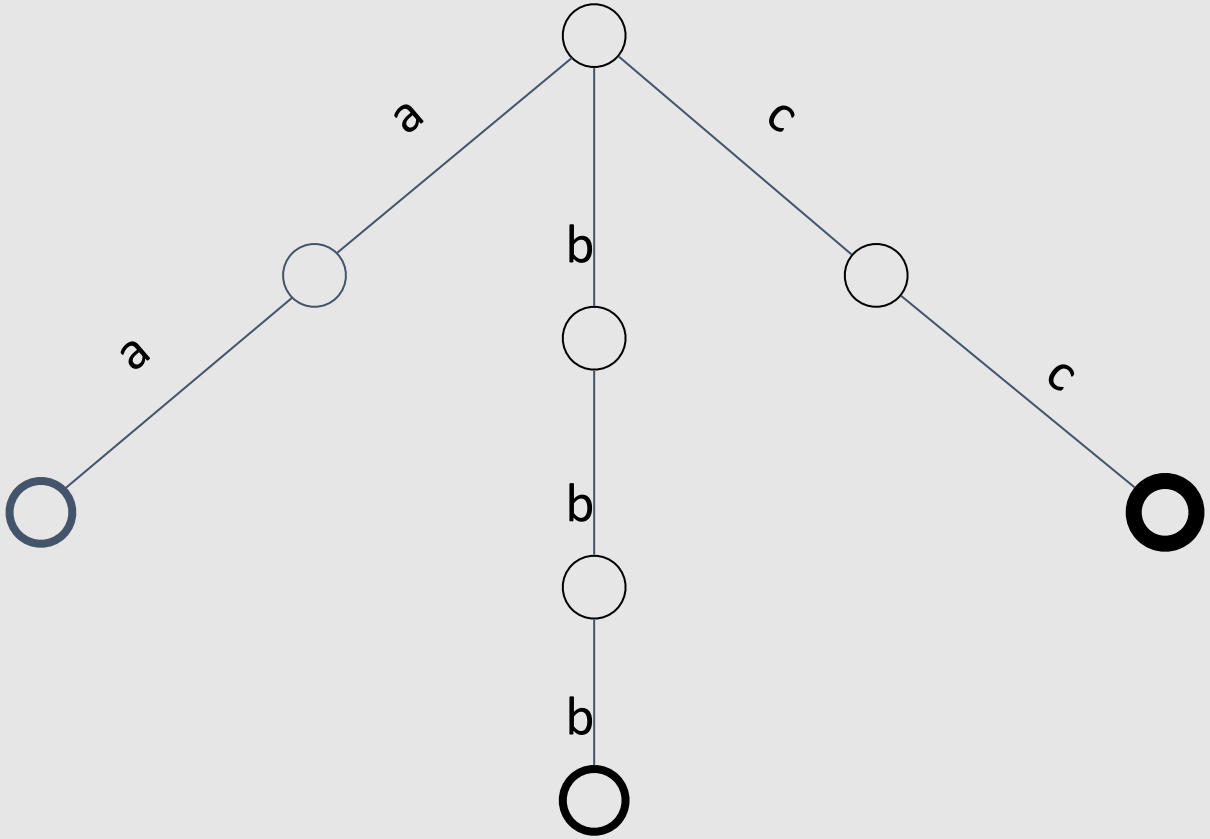
t = dabac

answer = {0, 1, 3}

Solution

First, let's consider a super trivial example where the strings do not contain any common characters at all.

set = {aa, bbb, cc}

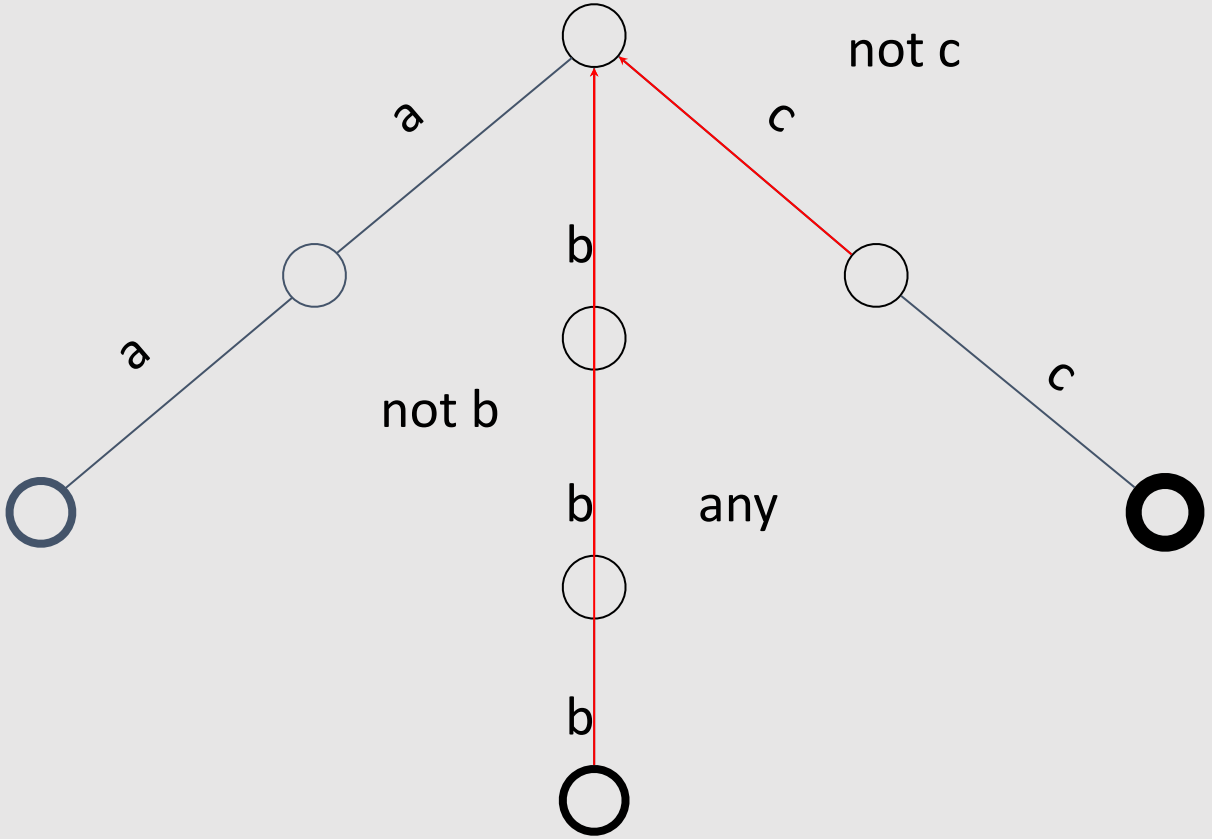


Solution

Then let's simply go through our trie if there is an edge, and if there isn't, then return to the root vertex.

set = {aa, bbb, cc}

word - abbccc



Automaton

Such a structure is generally called an automaton.

First, let's discuss informally what an automaton is.

An automaton is a set of states and transitions between them. In the case of strings, we will consider that a transition is a letter. If we can reach some terminal vertex via the automaton, it means that the automaton accepts the word obtained in such a way.

Automaton

We will consider that a Deterministic Finite Automaton (DFA) A is <Q, ∑, δ, S_0, F>.

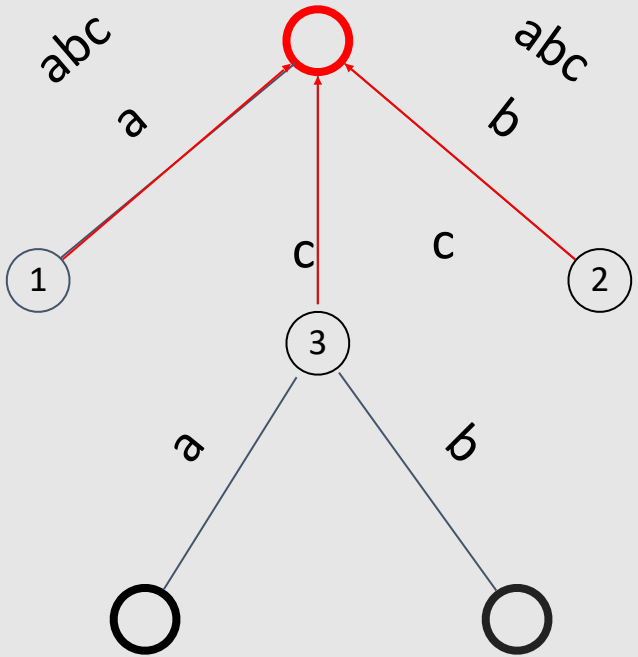
Q — set of automation's states

∑ - alphabet

δ - function of transition, such as Q * ∑ -> Q

S_0 ∈ Q — starting state

F ⊆ Q - set of finite states


abc
a
abc
b
c
c
1
2
3
a
b

Example

A = <Q, ∑, δ, S_0, F>.

Q — {0, 1, 2, 3, 4, 5}

∑ - {a, b, c}

δ - {(0, a) -> 1, (0, b) -> 2, (0, c) -> 3, (1/2, a/b/c) -> 0, (3, c) -> 0, (3, a) -> 4, (3, b) -> 5}

S_0 = 0

F = {4, 5}

Initial task

Let's return to the original problem and solve it again, but now with weaker constraints - let's agree that no string s_i is a substring of s_j.

We need to understand where to transition from any vertex by any letter, and then we will obtain a complete automaton.

Steps

What would we do if there is no edge by letter in the trie.

Let's think, for example, we are currently standing in some vertex of the trie, which means the string 'aba', and we want to make a step by the letter 'c'.

Steps

Ideally, we would like to make a step to the word 'abac', but if it doesn't exist, we would want to make a step to 'bac', 'ac', or 'c', that is, the longest suffix of the string, because we clearly want to find some terminal string in the trie while no string is a substring of any other.

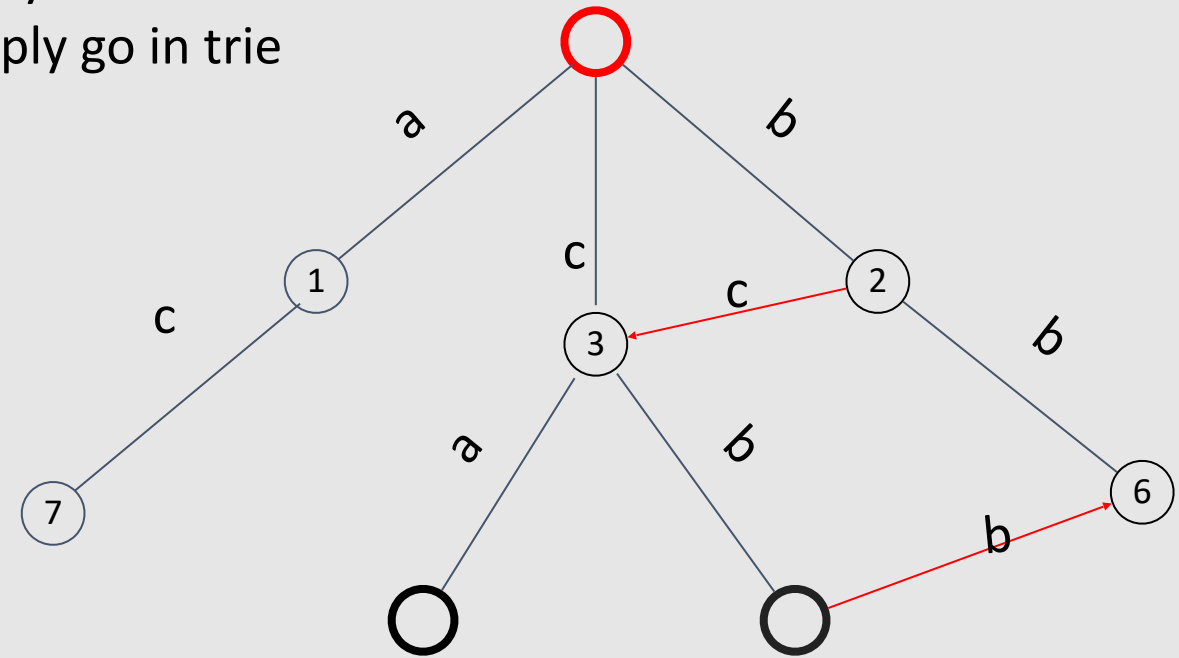
Such a reference will be called 'suffix_by_letter'.

we are staying in the first vertex

string that will be if we go from root to first vertex is s = “a”

we want to go by letter c = ‘c’

ac - we will simply go in trie



Steps

s = {aaaaab, aaca}

aaaaaca

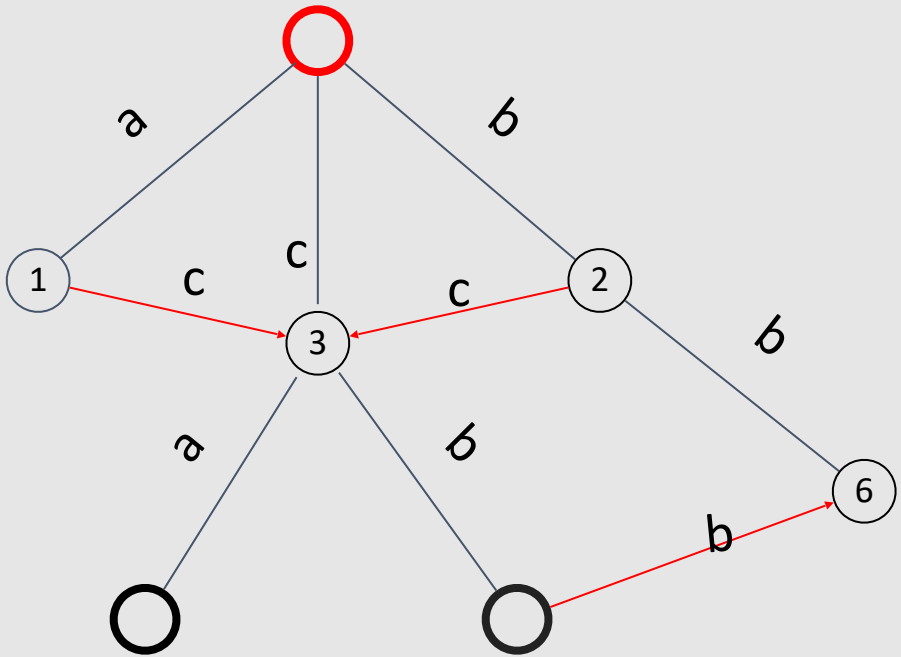
aaaaa

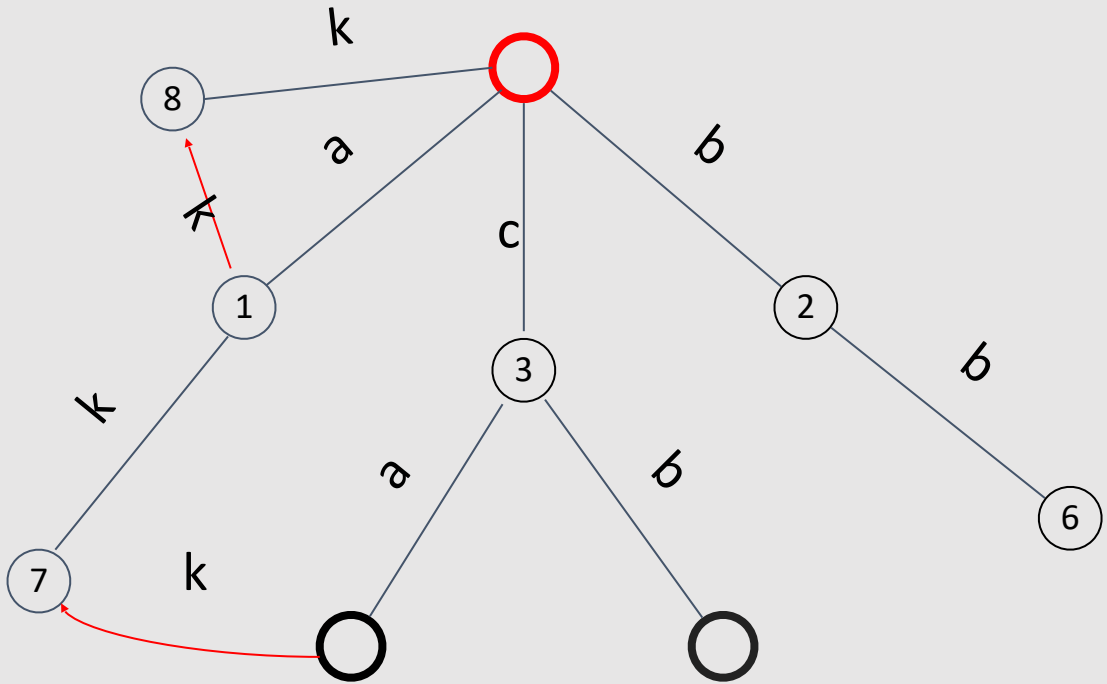
we are staying in the first vertex

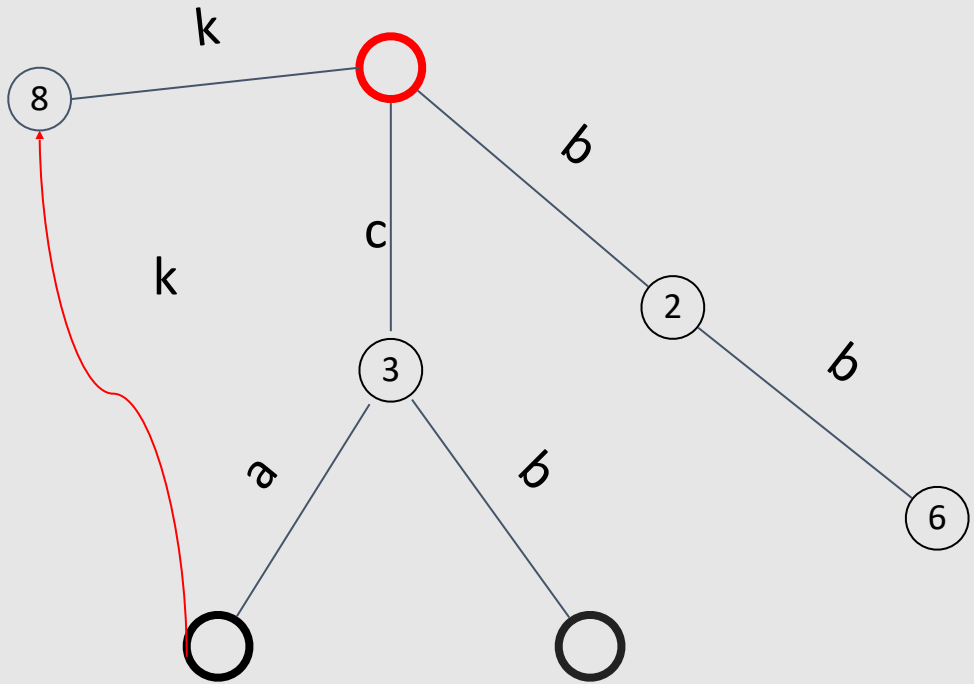
string that will be if we go from root to first vertex is s = “a”

we want to go by letter c = ‘c’

ac




k
8
a
b
k
c
1
2
3
b
k
a
b
6
7
k

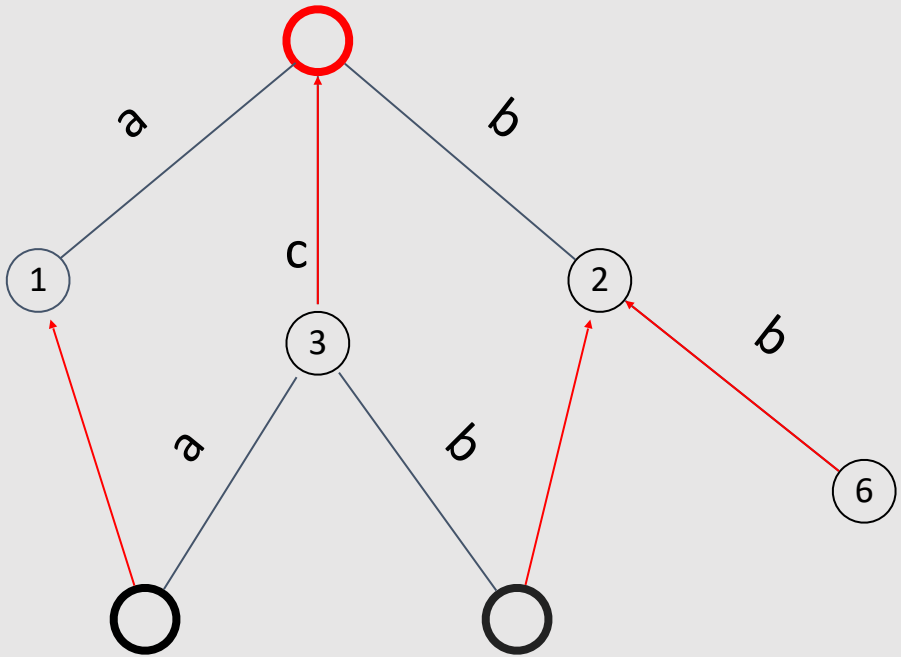

k
8
b
c
k
2
3
b
a
b
6

Steps

Moreover, for the sake of simplicity of implementation, a simple suffix link is also defined. This is a link that leads to the longest suffix of the current string that exists in the trie.

We will call it 'suffix_link'.

Let’s think about its connection with the prefix function.


a
b
c
1
2
3
a
b
b
6

Dynamic programming?

suffix_link[v] - link to max suffix for string (root->v)

suffix_link[root] = root

edge (p, v, c) : suffix_link[v] = suffix_by_letter[p][c]

dfs order

Example

root->v_1->v_2->v_3

a b a

suffix_link[v3] = suffix(aba) = suffix_by_letter[v2][‘a’]

Proof of formula

suffix_link[v] = suffix of path(root->v) = suffix of path((root->v) + edge(p->v))

suffix_by_letter[p][c] = suffix of string(suffix of path(root->p) + c) = suffix of string(suffix of path(root->p) + edge(p->v))

Dynamic programming?

suffix_by_letter[v][c] - link to max suffix for string (root->v + c)

suffix_by_letter[root][any letter] = root

if trie contains edge (v, X, c) suffix_by_letter[v][c] = X

else suffix_by_letter[v][c] = suffix_by_letter[suffix_link[v]][c]

Example

root->v_1->v_2->v_3

a b a

suffix_by_letter[v3, b] = suffix(abab) =

suffix_by_letter[suffix_link[v3]][b] =

suffix_by_letter[suffix_link(aba)][b] = suffix_by_letter[a][b] = ab

Proof of formula

if trie contains edge (v, X, c) suffix_by_letter[v][c] = X

suffix_by_letter[v][c] = X, because we add c to path (root->v)

and result = X

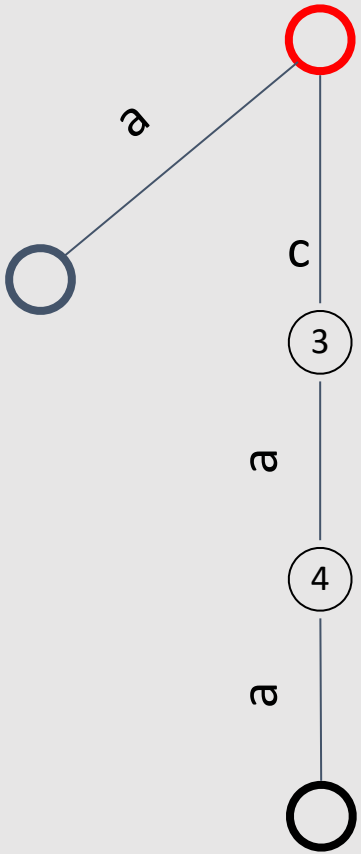
Proof of formula

suffix_by_letter[v][c] = suffix_by_letter[suffix_link[v]][c]

if no edge -> we go to string with less length and take the same symbol.

And what about substrings?

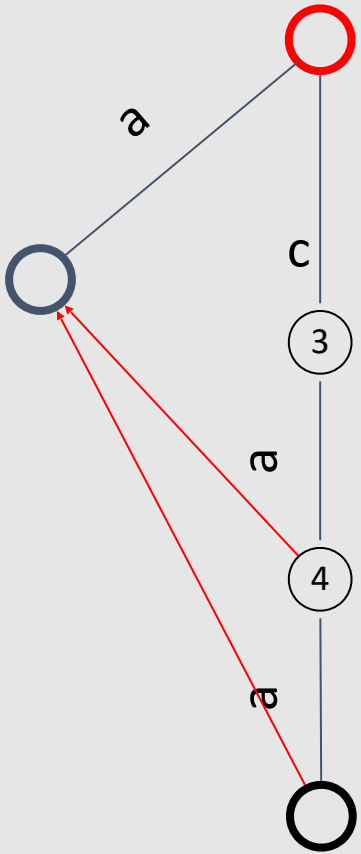
The only problem, if we allow s[i] to be a substring of s[j], is that we might find a path (root->v), and some part of the string s_path, formed by this path, is also in the trie, and we might miss it.


a
c
3
a
4
a

And what about substrings?

Note that such a string s# is a suffix of our string s.

Consequently, the vertex corresponding to it in the trie is the suffix link of the vertex corresponding to s, or the suffix link of its suffix link, and so on up to the root.


a
c
3
a
4
a

Again dynamic programming?

count[v] - dp, which means amount of terminal suffix links.

count[root] = is_term[root]

in dfs order

count[v] = count[suff_link[v]] + is_term[v]

Final result

Let's look again at how the Aho-Corasick algorithm works, it is also called the prefix automaton.

Final result

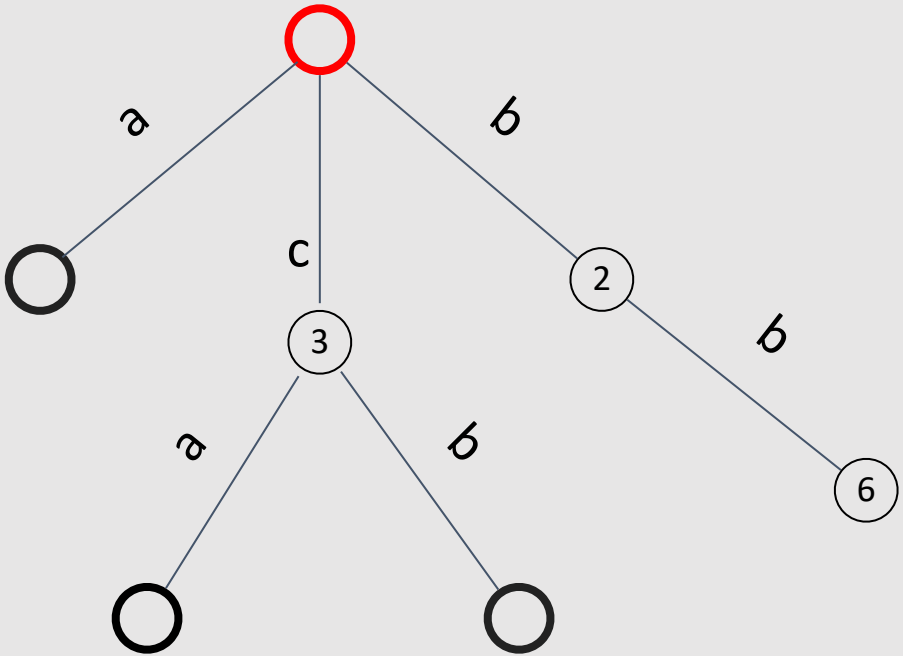
First, we build a trie based on the strings s.

Example

t = cabcb

s = {a, ca, cb, b}

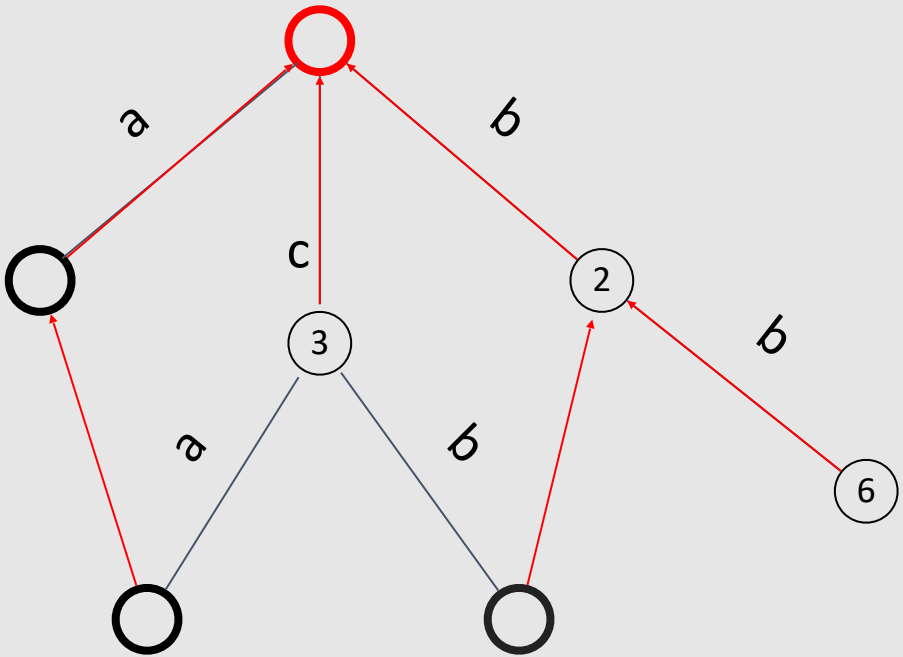
answer = 1 + 1 + 1 + 2 = 5


a
c
b
2
3
b
a
b
6

Final result

Then we need to build suffix_links and suffix_by_letter.

suffix_link



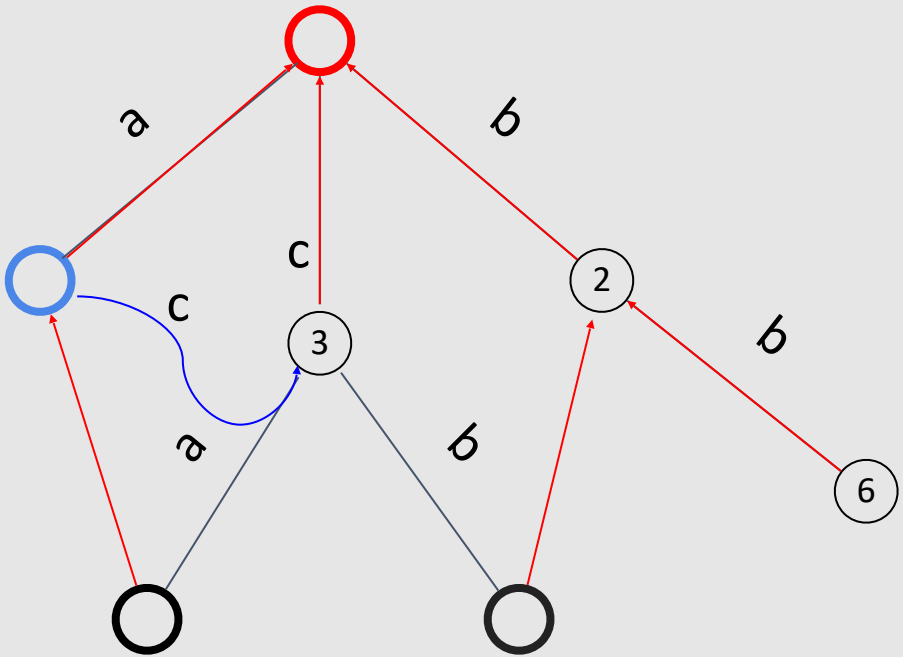
suffix_by_letter[“a”][‘c’]

we dont have edge

“a”->”ac”

suffix_by_letter

[suffix[“a”]][‘c’]



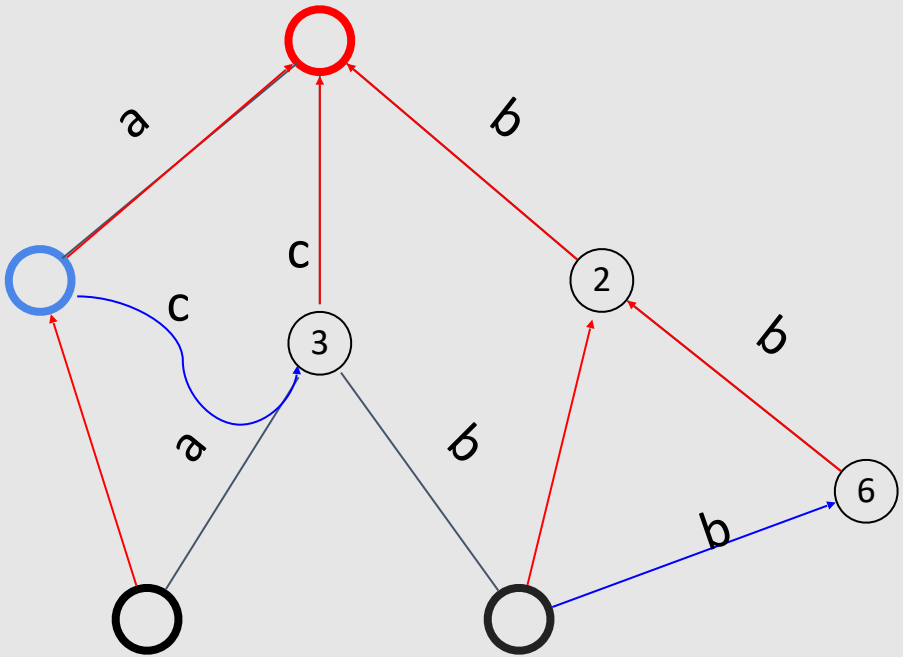
suffix_by_letter[“cb”][‘b’]

we dont have edge

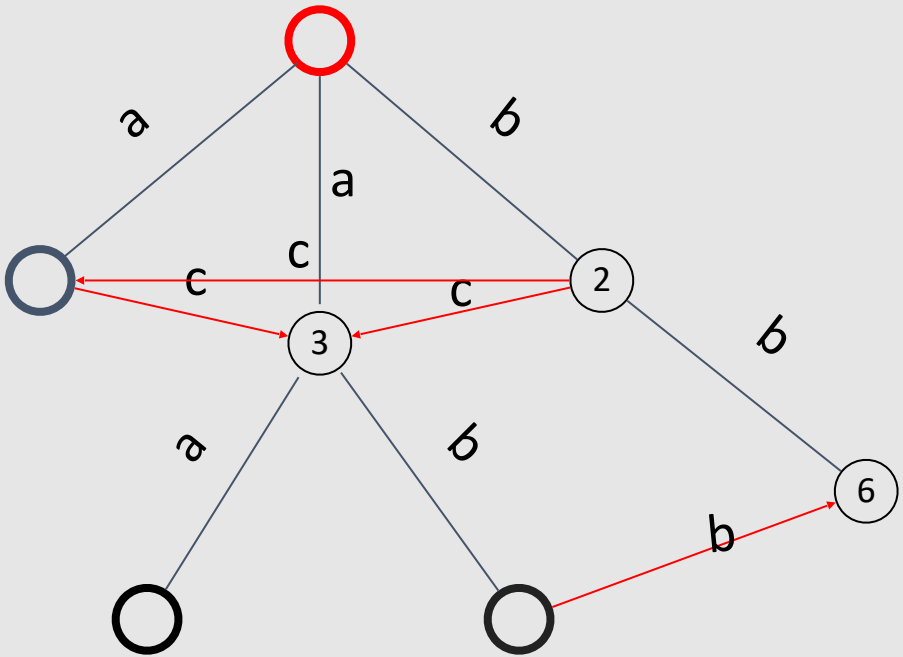
“cb”->”cbb”

suffix_by_letter

[suffix[“cb”]][‘b’]

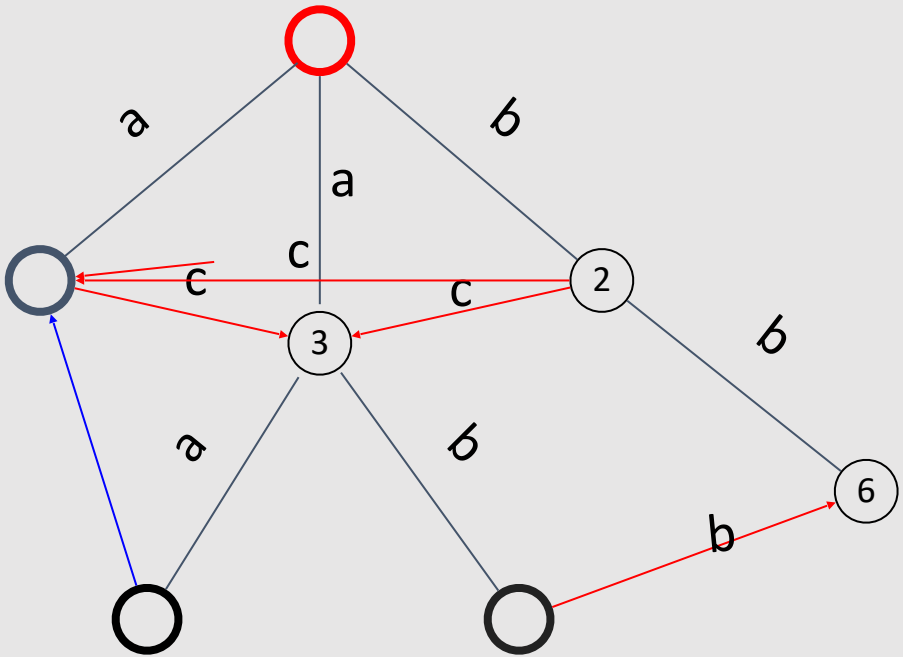


suffix_by_letter



suffix_link[4] = suffix_link[“ca”]

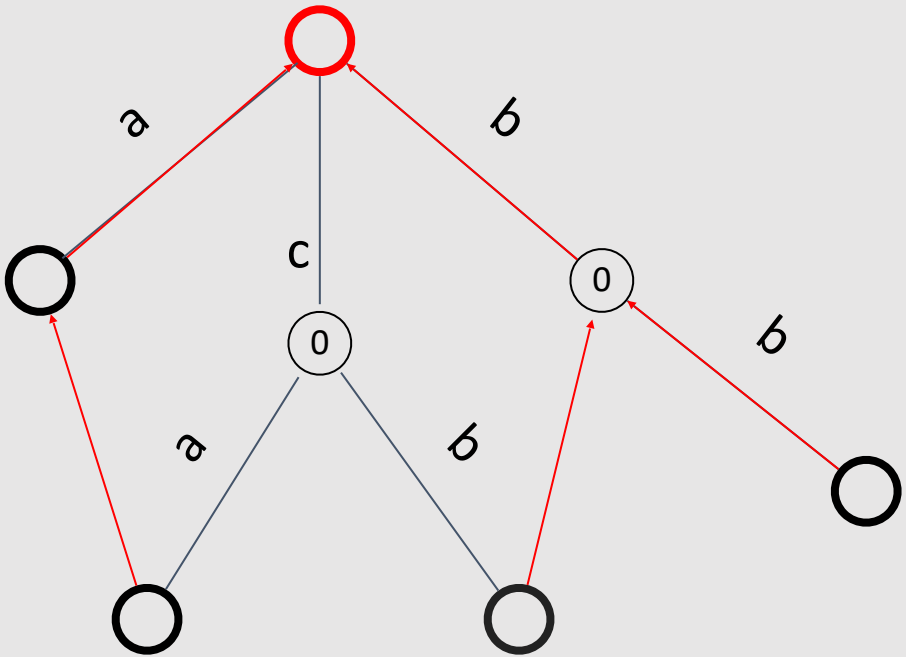
suffix_by_letter[3][‘a’] = 1



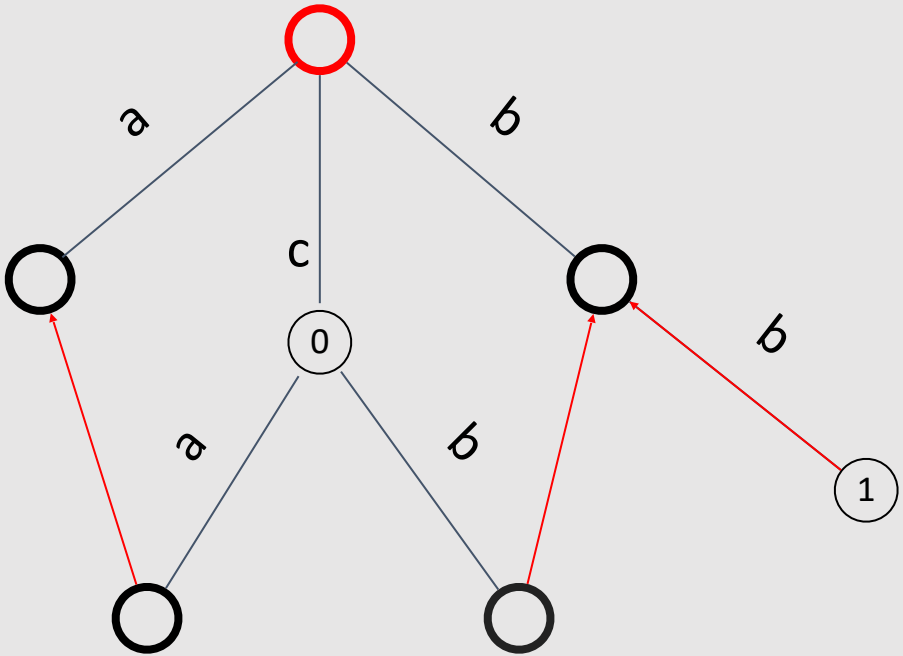
Final result

Then we need to calculate count[u], which means amount of terminal links, that are reachable by suffix_links.

count_dp



count_dp



Final result

To find the number of occurrences of strings from s as substrings in t, we move through the automaton and the string t simultaneously, and each time we add count[u] if we are standing in the vertex u of the automaton.

Example

t = cabcb

s = {a, ca, cb, b}

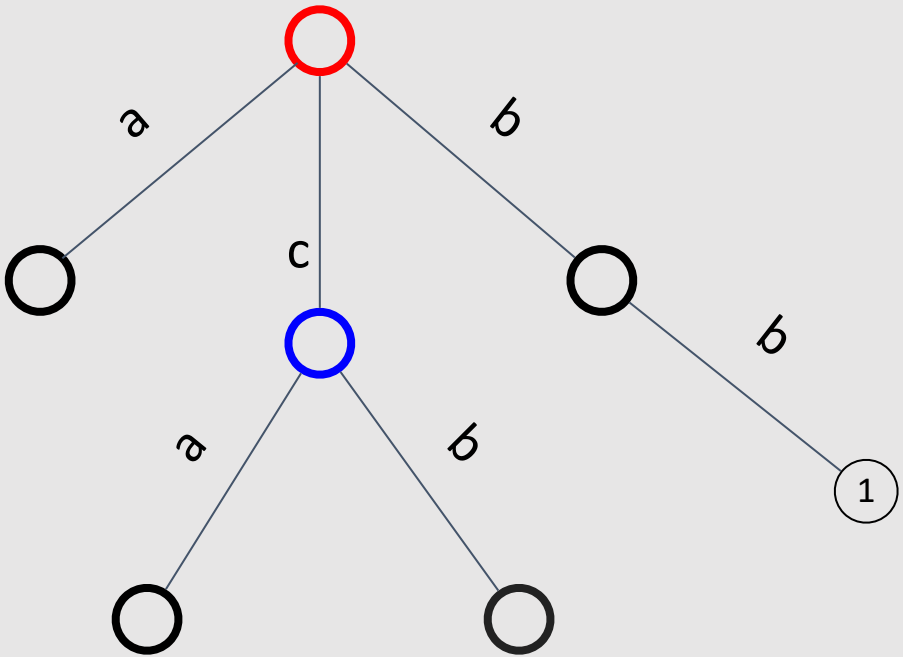
answer = 1 + 1 + 1 + 2 = 5

And now we will calculate the answer using automation.

i - current symbol, red - current vertex, blue - suffix_by_letter[red][s[i]]

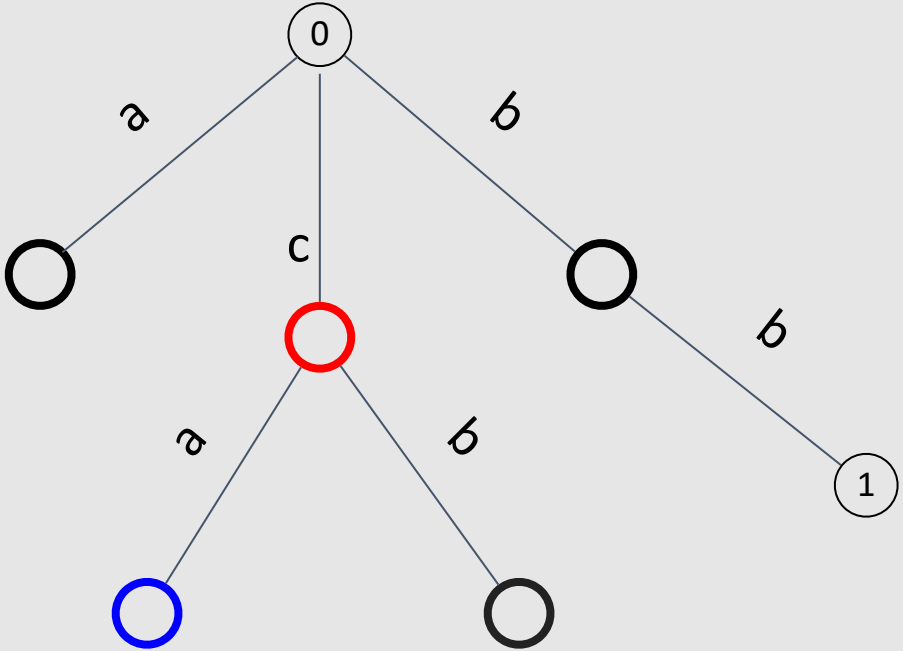
we checked t = “”, i = 0, t[i] = ‘c’, suffix_by_letter[“”][s[i] = ‘c’] = blue

answer = 0



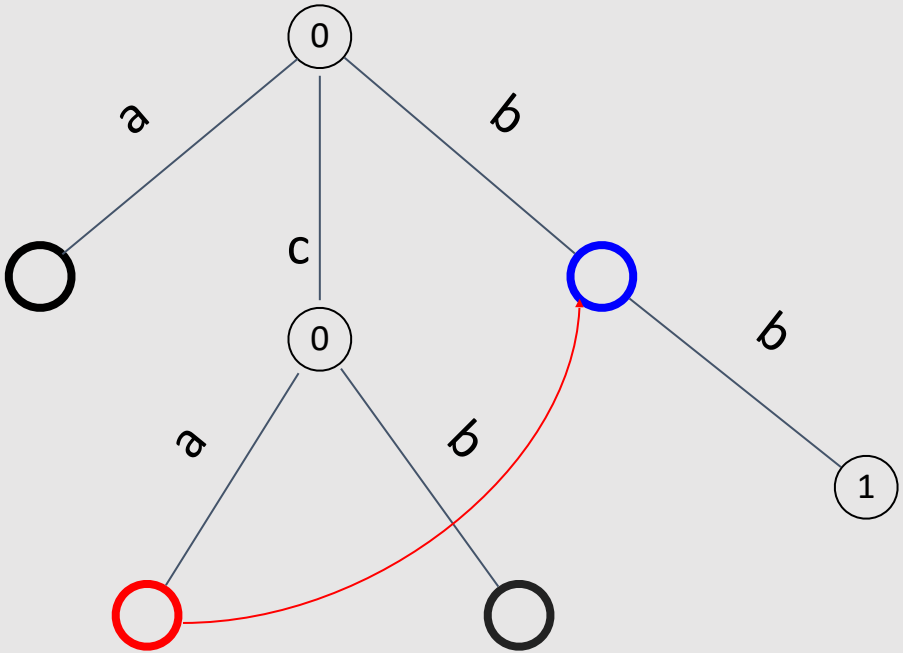
we checked t = “c”, s[i] = ‘a’, suffix_by_letter[“c”][s[i] = ‘a’] = blue

answer = 0 + 2 = 2



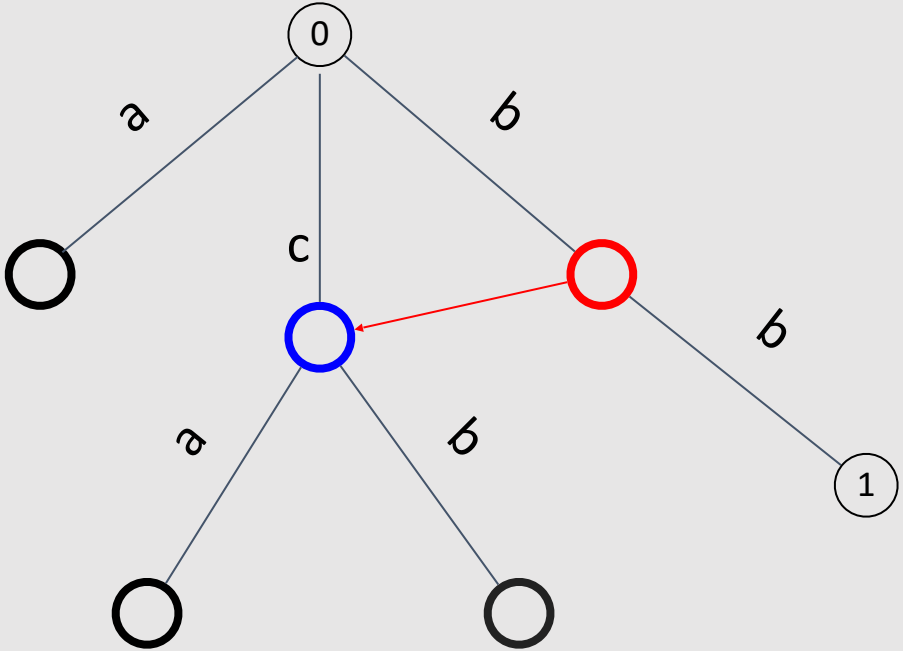
we checked t = “ca”, s[i] = ‘b’, suffix_by_letter[0][s[i] = ‘b’] = blue

answer = 2 + 1 = 3



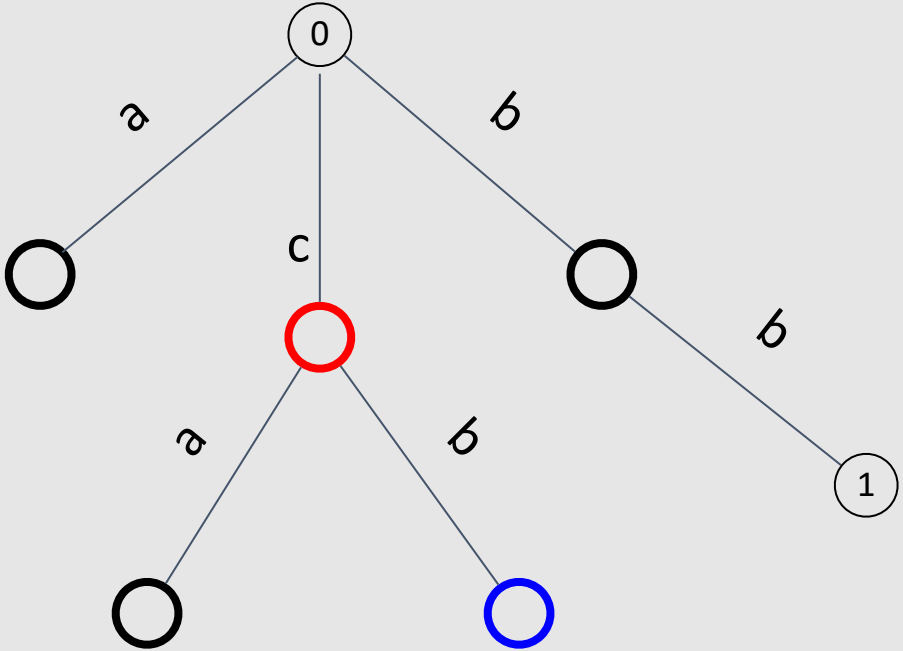
we checked t = “cab”, s[i] = ‘c’, suffix_by_letter[0][s[i] = ‘c’] = blue

answer = 3 + 0



we checked t = “cabc”, s[i] = ‘b’, suffix_by_letter[0][s[i] = ‘b’] = blue

answer = 3 + 2 = 5



Realisation

It turns out I didn't deceive you and it works.

However, this is a quite complex algorithm to implement, and there are many practical improvements to it (for example, you can quickly calculate all the links using BFS).

I will focus on the simplest possible implementation.

```
6.  const int ALPHA_LENGTH = 26;
7.
8.  struct Vertex {
9.      //common info
10.     int id; // id of vertex
11.     bool is_terminal;
12.     int p; // id of parent
13.     char letter; // letter from edge <p, id>
14.
15.     // Trie info
16.     map<char, int> steps;
17.
18.     // Aho info
19.     map<char, int> links;
20.     int link;
21.     int count_term_links;
22.
23.     Vertex(int u, int v, char c) {
24.         is_terminal = false;
25.         count_term_links = -1;
26.         link = -1;
27.         p = u;
28.         id = v;
29.         letter = c;
30.     }
31.
32.     int add_edge(char letter, vector<Vertex>& Trie) {
33.         steps[letter] = Trie.size();
34.         Vertex son(id, Trie.size(), letter);
35.         Trie.push_back(son);
36.         return steps[letter];
37.     }
```

```
    int step_by_letter(char letter) {
        if (steps.count(letter)) {
            return steps[letter];
        }
        else {
            return 0;
        }
    }

    int get(vector<Vertex>& Trie) {
        if(link == -1) {
            if (id == 0 || p == 0) {
                link = 0;
            }
            else {
                link = Trie[Trie[p].get(Trie)].go(letter, Trie);
            }
        }
        return link;
    }

    int go(int next_letter, vector<Vertex>& Trie) {
        // if !go.contains(next_letter)
        if(links.count(next_letter) == 0) {
            if (steps.count(next_letter)) {
                links[next_letter] = steps[next_letter];
            }
            else if (id == 0) {
                links[next_letter] = 0;
            }
            else {
                links[next_letter] = Trie[link].go(next_letter, Trie);
            }
        }
        return links[next_letter];
    }
```

```
76.     int count(vector<Vertex>& Trie) {
77.         if (count_term_links != -1) {
78.             return count_term_links;
79.         }
80.         if (id == 0) {
81.             return count_term_links = 0;
82.         }
83.         return count_term_links = is_terminal + Trie[get(Trie)].count(Trie);
84.     }
85. };
86. 
87. void add_string(int cur_node_id, string s, vector<Vertex>& Trie) {
88.     for (auto letter: s) {
89.         int next_node = Trie[cur_node_id].step_by_letter(letter);
90.         if (!next_node) {
91.             next_node = Trie[cur_node_id].add_edge(letter, Trie);
92.         }
93.         cur_node_id = next_node;
94.     }
95.     Trie[cur_node_id].is_terminal = true;
96. }
97. 
98. int make_step(int &vertex, char symbol, vector<Vertex>& Trie) {
99.     vertex = Trie[vertex].go(symbol, Trie);
100.     return Trie[vertex].count(Trie);
101. }
```

```
103.  int main() {
104.      vector<Vertex> Trie;
105.      Trie.reserve(100000);
106.      Vertex root({-1, 0, ' '});
107.      Trie.push_back(root);
108.      string strings[4] = {"a", "ca", "cb", "b"};
109.      string t = "cabcb";
110.      for (auto &s: strings) {
111.          add_string(root.id, s, Trie);
112.      }
113.      int answer = 0;
114.      int cur_id = root.id;
115.      for (char symbol: t) {
116.          answer += make_step(cur_id, symbol, Trie);
117.      }
118.      cout << answer;
119.      return 0;
120.  }
```

Success #stdin #stdout 0.01s 5260KB

stdin

```
Standard input is empty
```

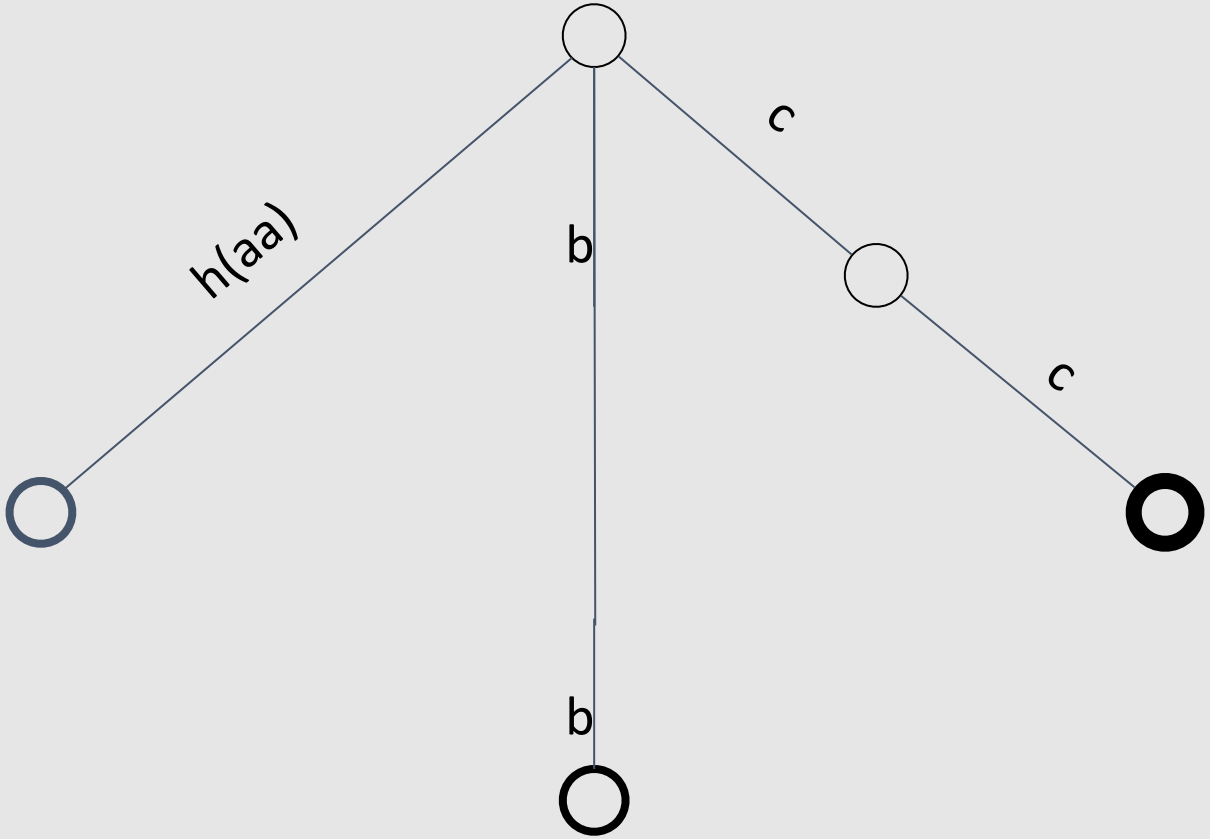
stdout

```
5
```

Realisation

Why this code doesnt work, if i remove string with reserve.

set = {aa, bbb, cc}



All codes

Trie, pointers, map - https://ideone.com/RpnxtT

Trie, pointers, array - https://ideone.com/1lOBT8

Trie, vector, array - https://ideone.com/MyaFAC

Trie, vector, map - https://ideone.com/eUJ6sD

aho - https://ideone.com/Skfp23