Ternary Search Tree

Advance String tree data structure

What is ternary search tree?

- A ternary search tree (TST) is a prefix base tree-based data structure used for efficiently storing and searching strings. Unlike binary trees, each node in a TST has three children: one for characters less than the current node's character, one for characters equal to it, and one for characters greater than it. TSTs are particularly useful for tasks like spell checking, autocomplete, and symbol tables due to their efficient handling of string operations and memory usage.

Why we learn ternary search tree data structure?

- Efficient Searching
- Memory efficiency
- prefix searching
- ordered operations
- versatility

In summary

- We use ternary search trees when we need to efficiently store and search for strings, especially in scenarios where memory efficiency, fast searching, and prefix-based operations are crucial. Common applications include spell checkers, autocomplete features, symbol tables in compilers, IP routing tables, and more. The benefit of employing a ternary search tree lies in its balance between memory efficiency and fast searching capabilities, making it a versatile choice for a wide range of string-related tasks.

A simple visualization of ternary search tree

Added strings

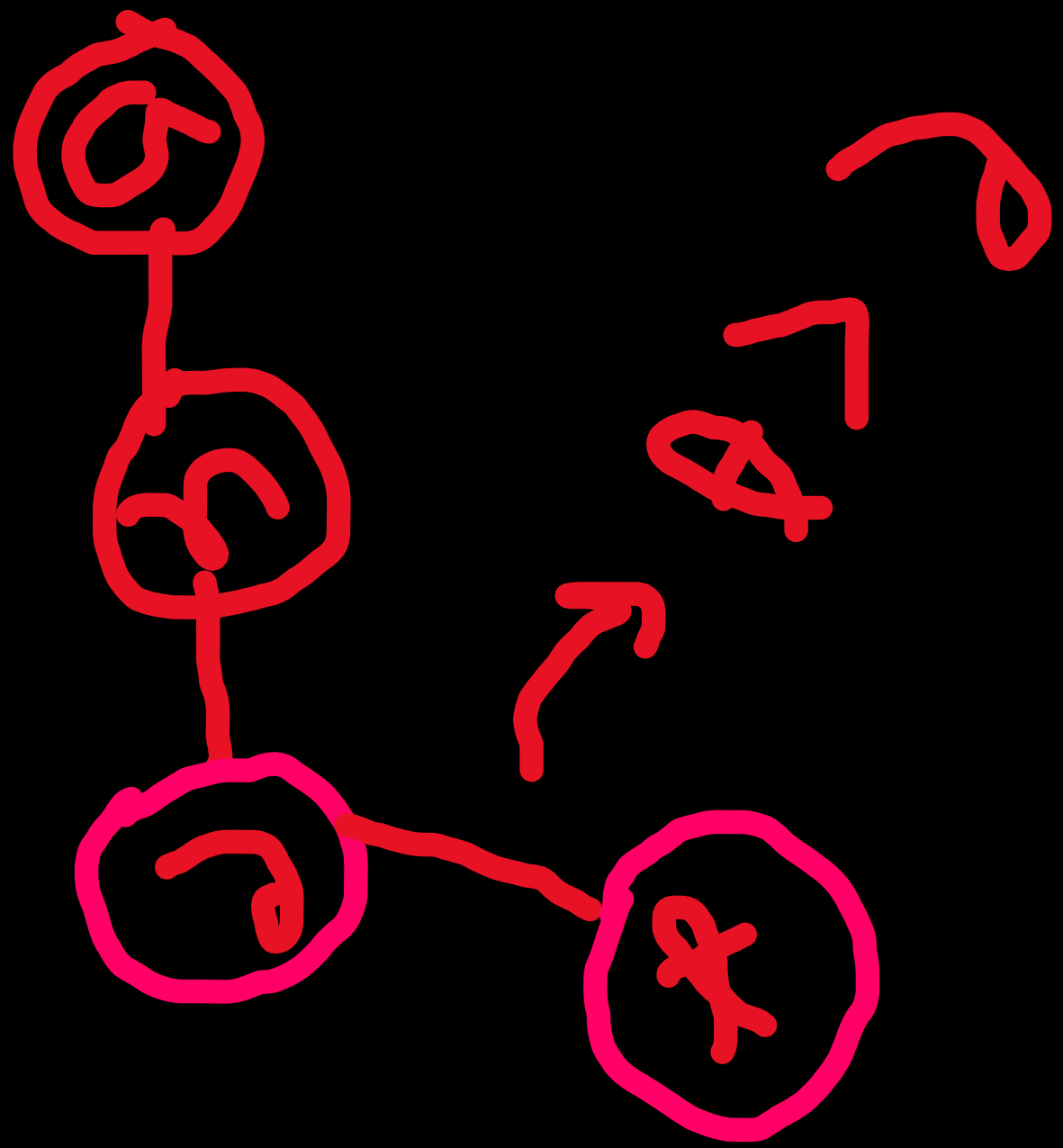
1. Ant
2. And

Node properties of ternary search tree

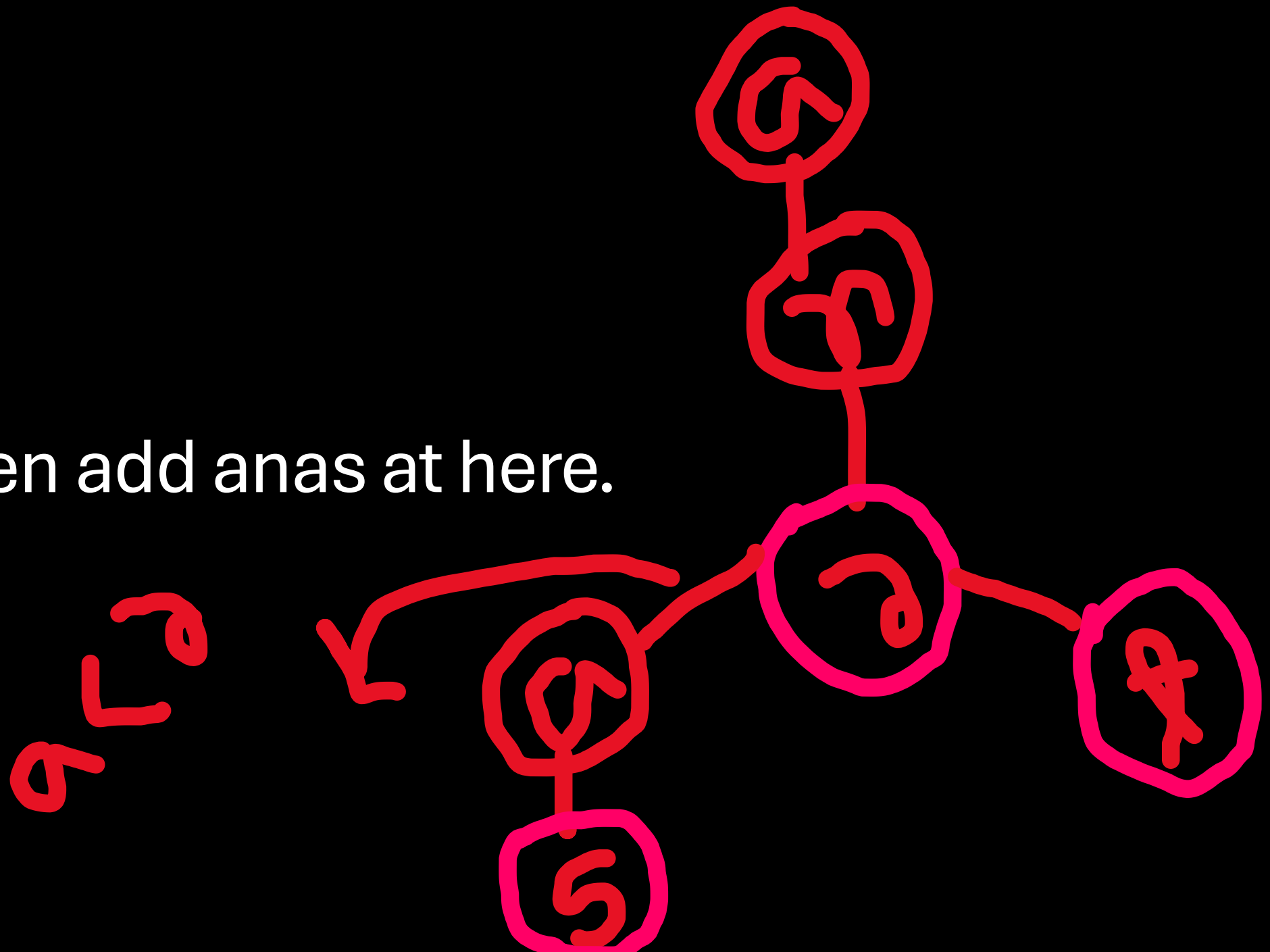
- Has a character data.
- Has a left child(smaller char than parent node).
- Has a right child(greater char than parent).
- Has a middle child(carry out the same string character).
- A marked up Boolean value to represent this is the end character of the string or not.

Same character
of a word set as the
middle node.

When try to add ant
at here.



When add anas at here.

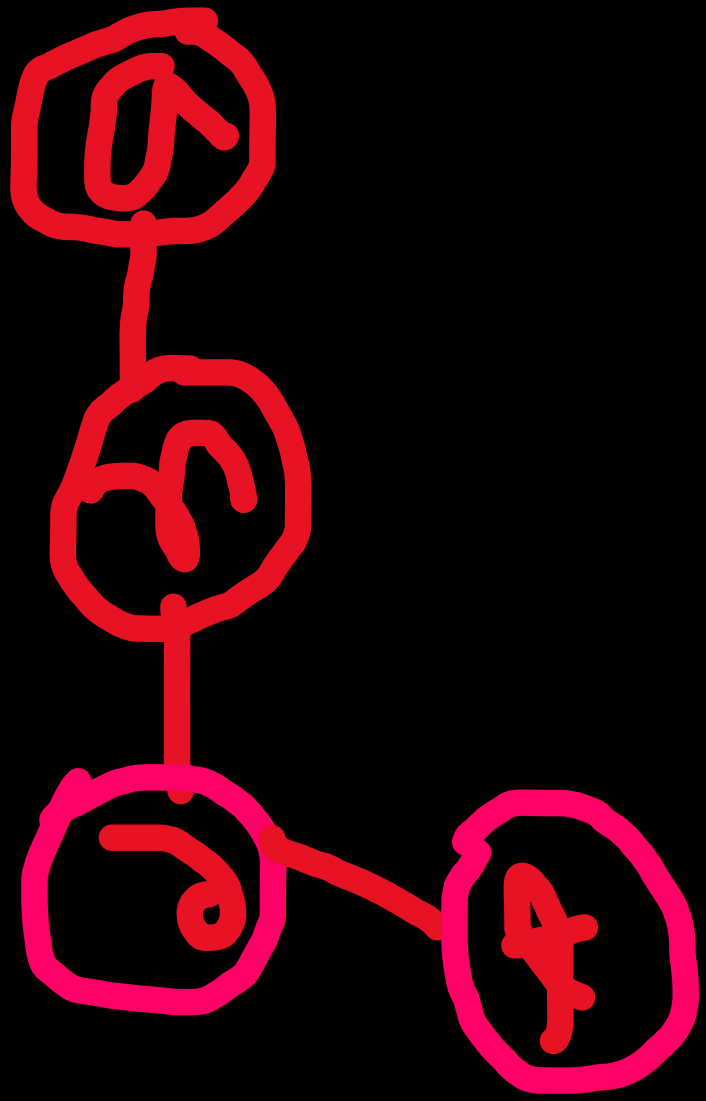


Let's solve an example to understand the insertion.

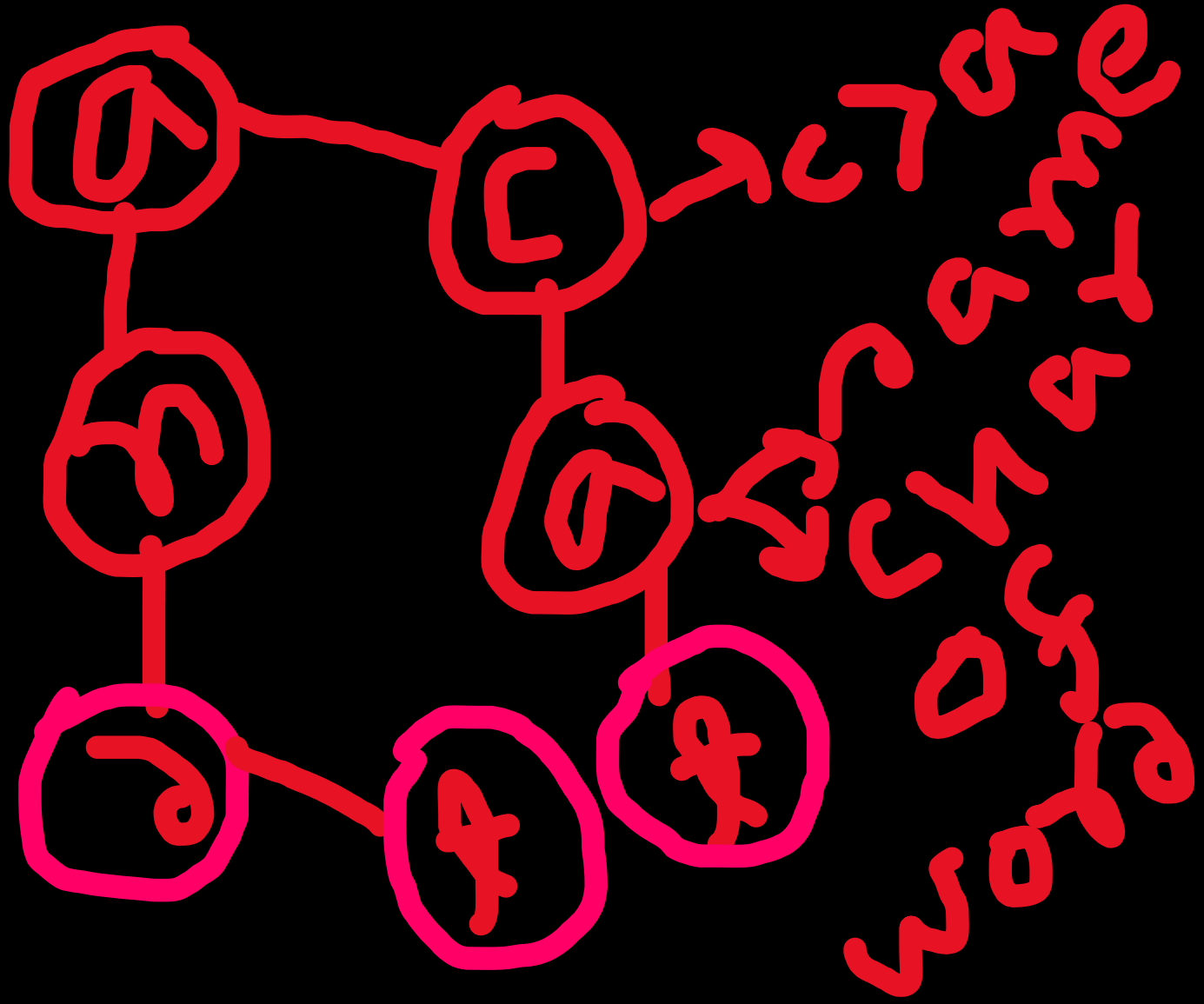
and, ant, cat, car, rat, ram, anas, anam, at, cow, tie, tide, am.

Add and at here.

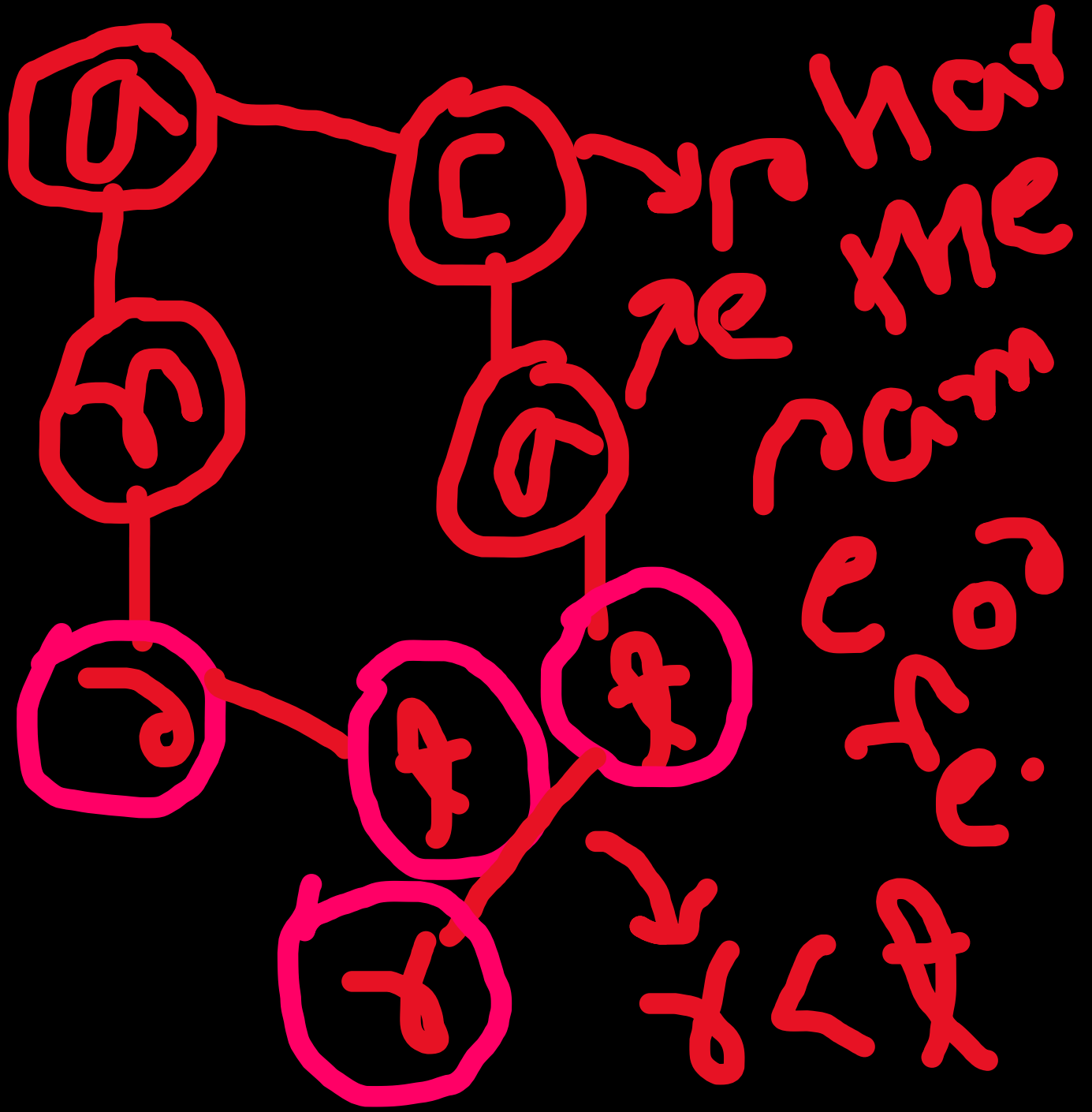
Add ant at here.



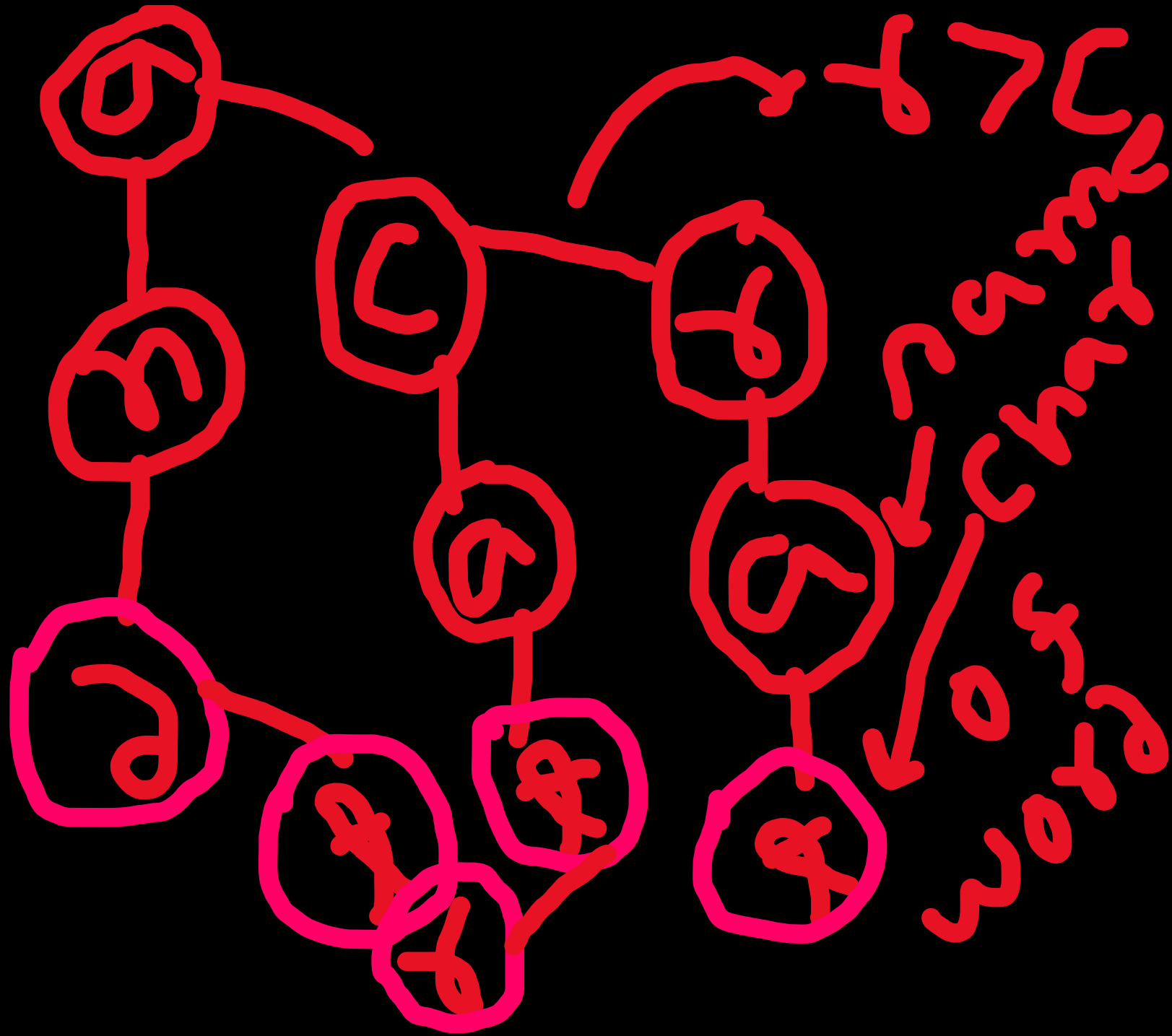
Add cat at here.



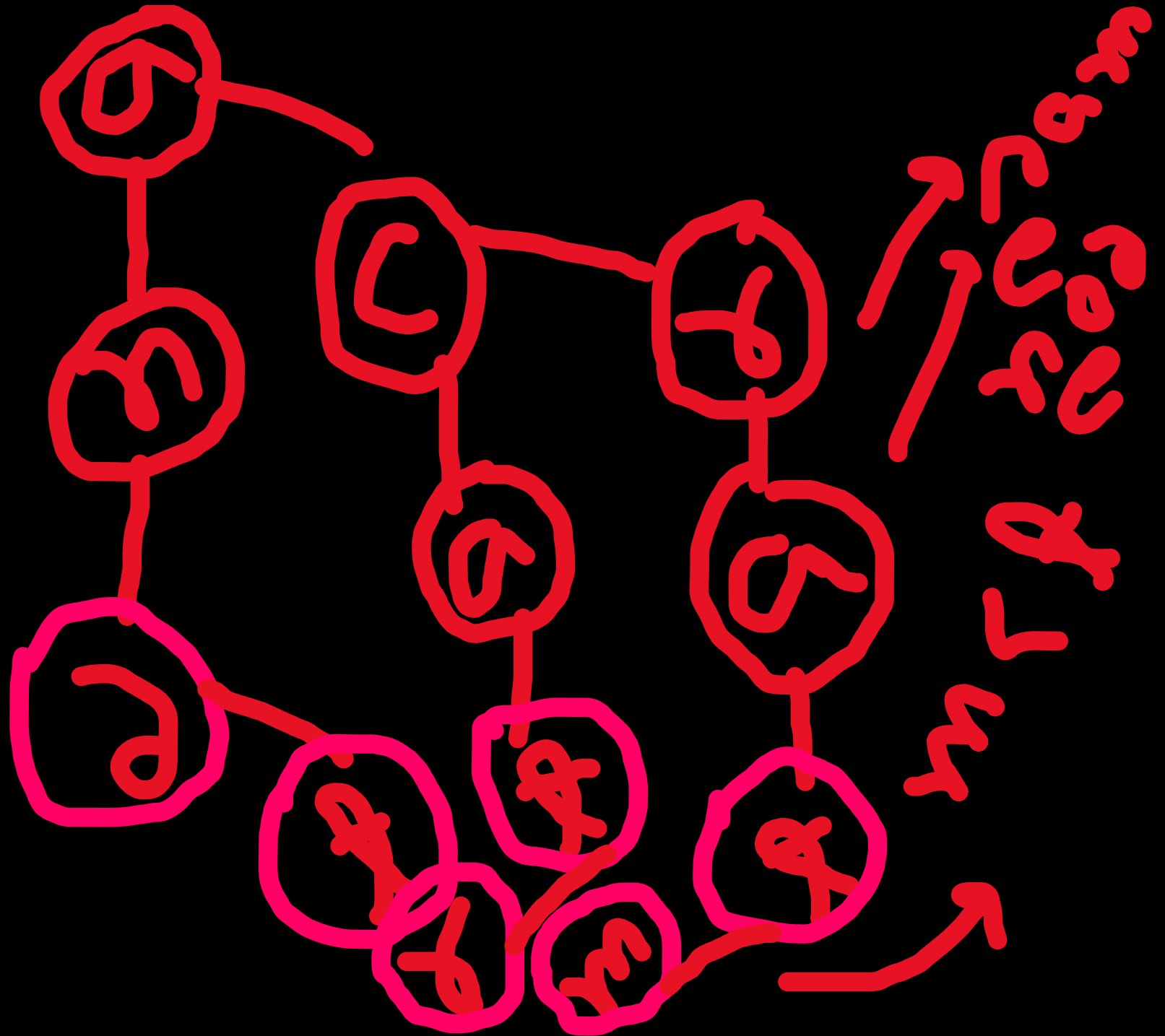
Add car at here.



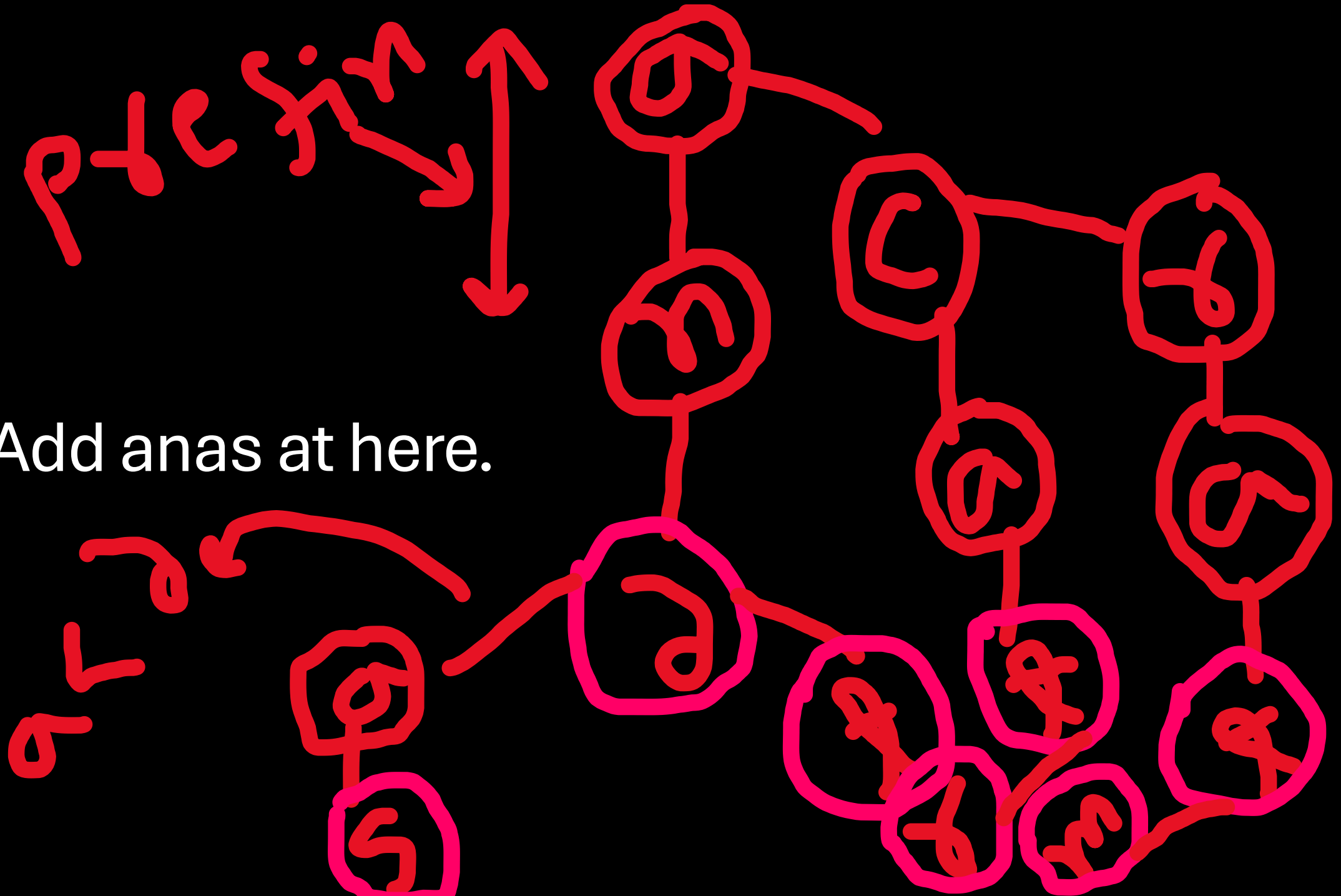
Add rat at here.



Add ram at here.

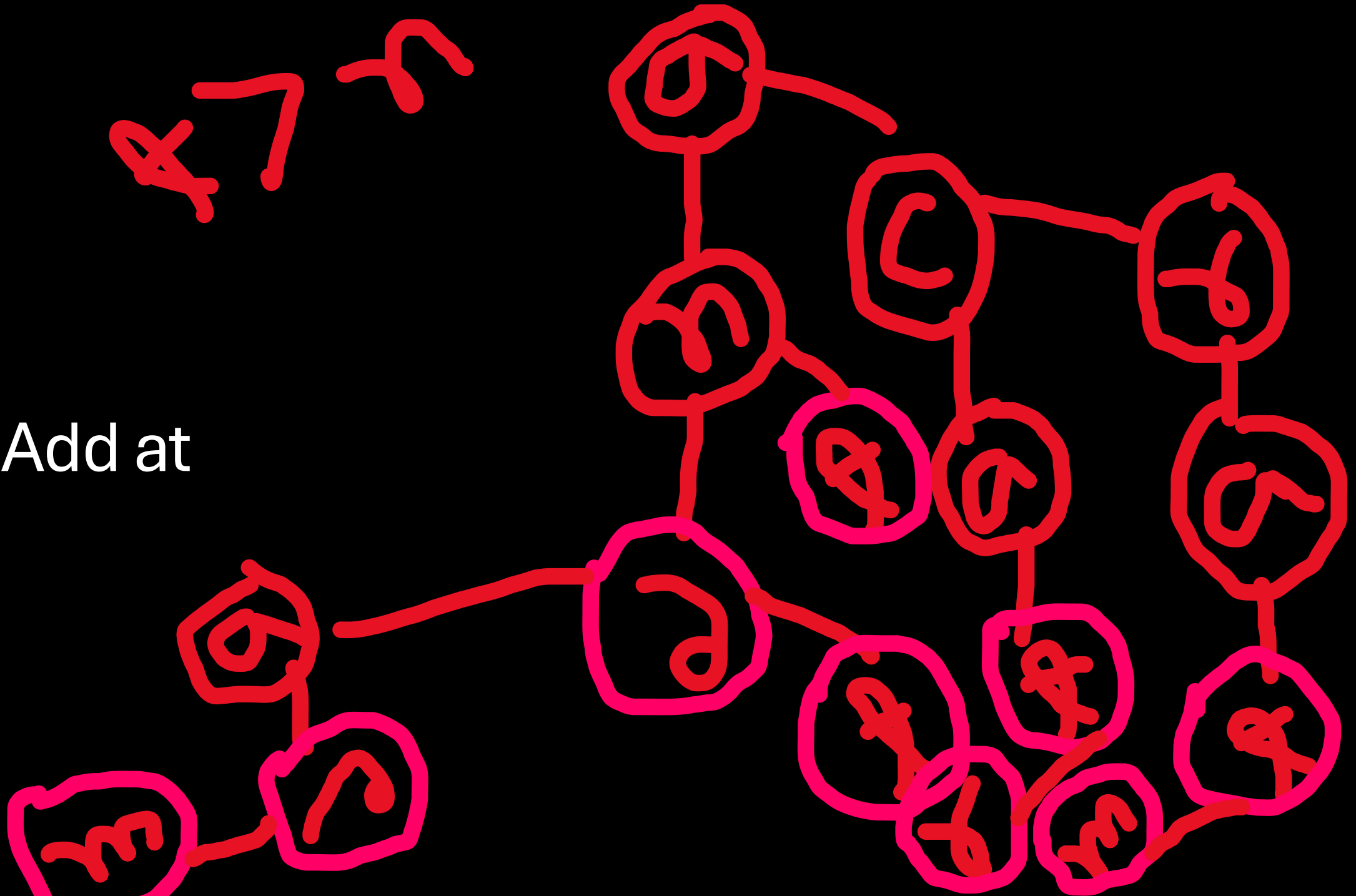


Add anas at here.



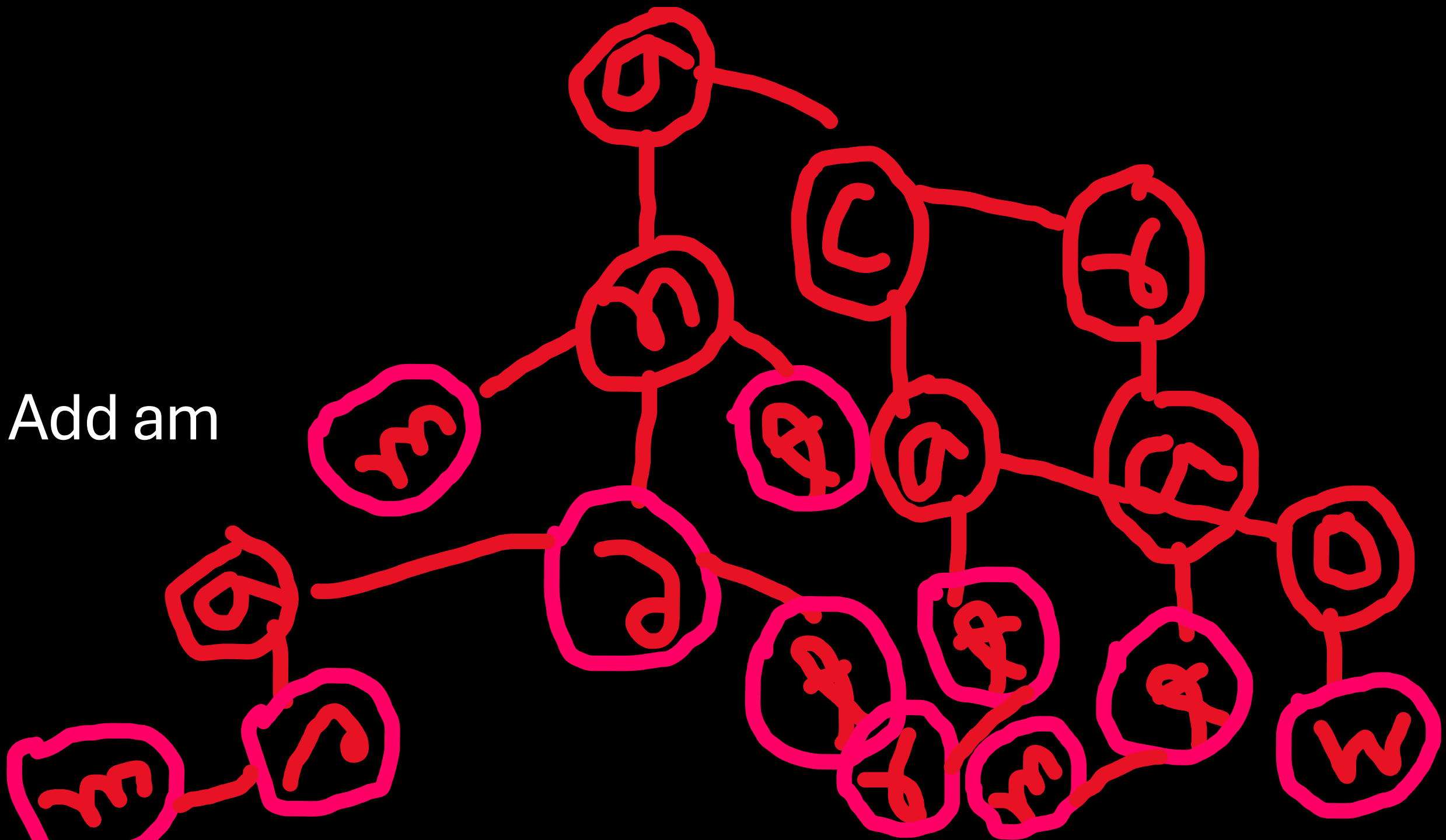
Add anam at here.

Add at

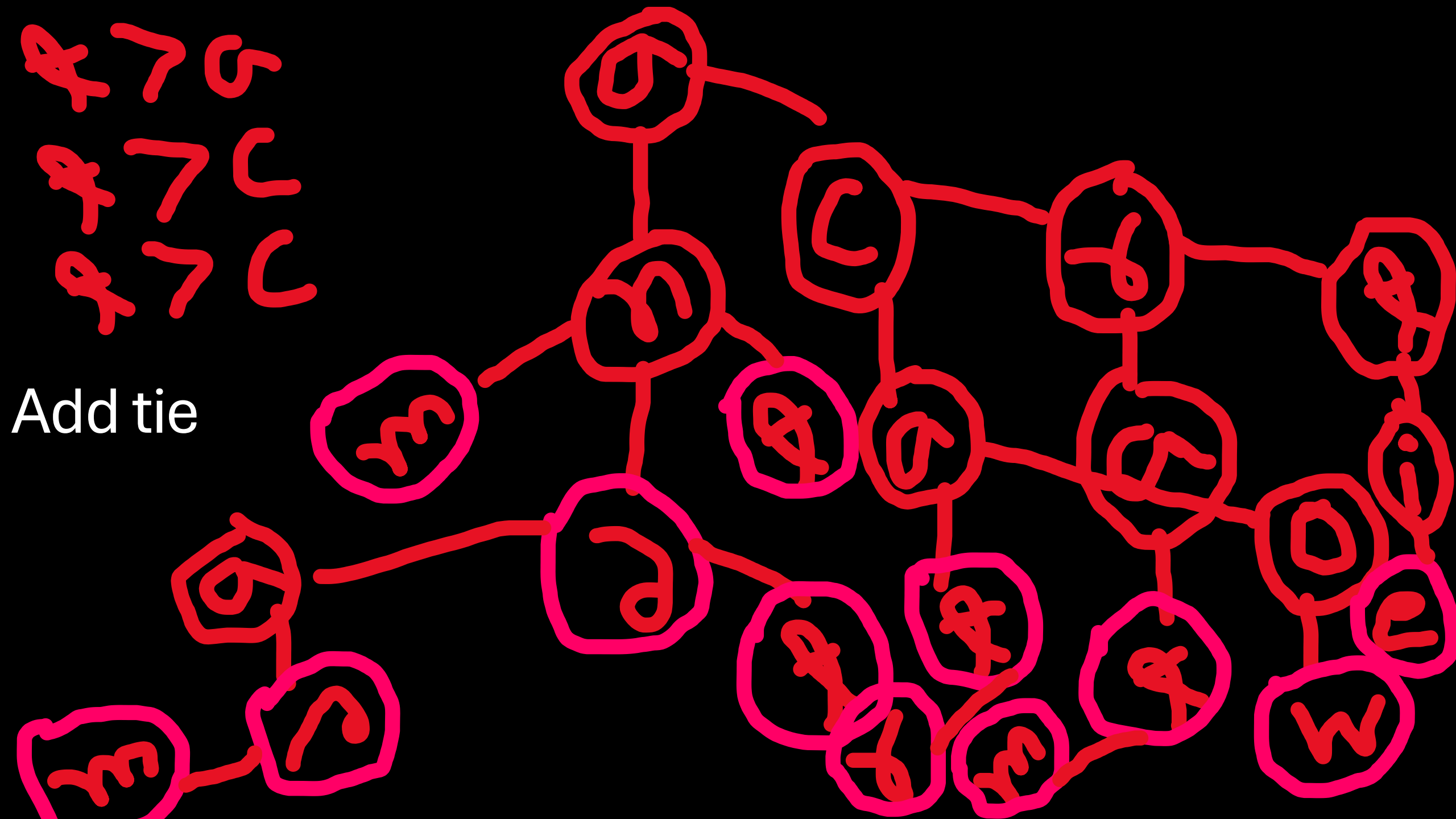


Add cow at here.

Add am



Add tie

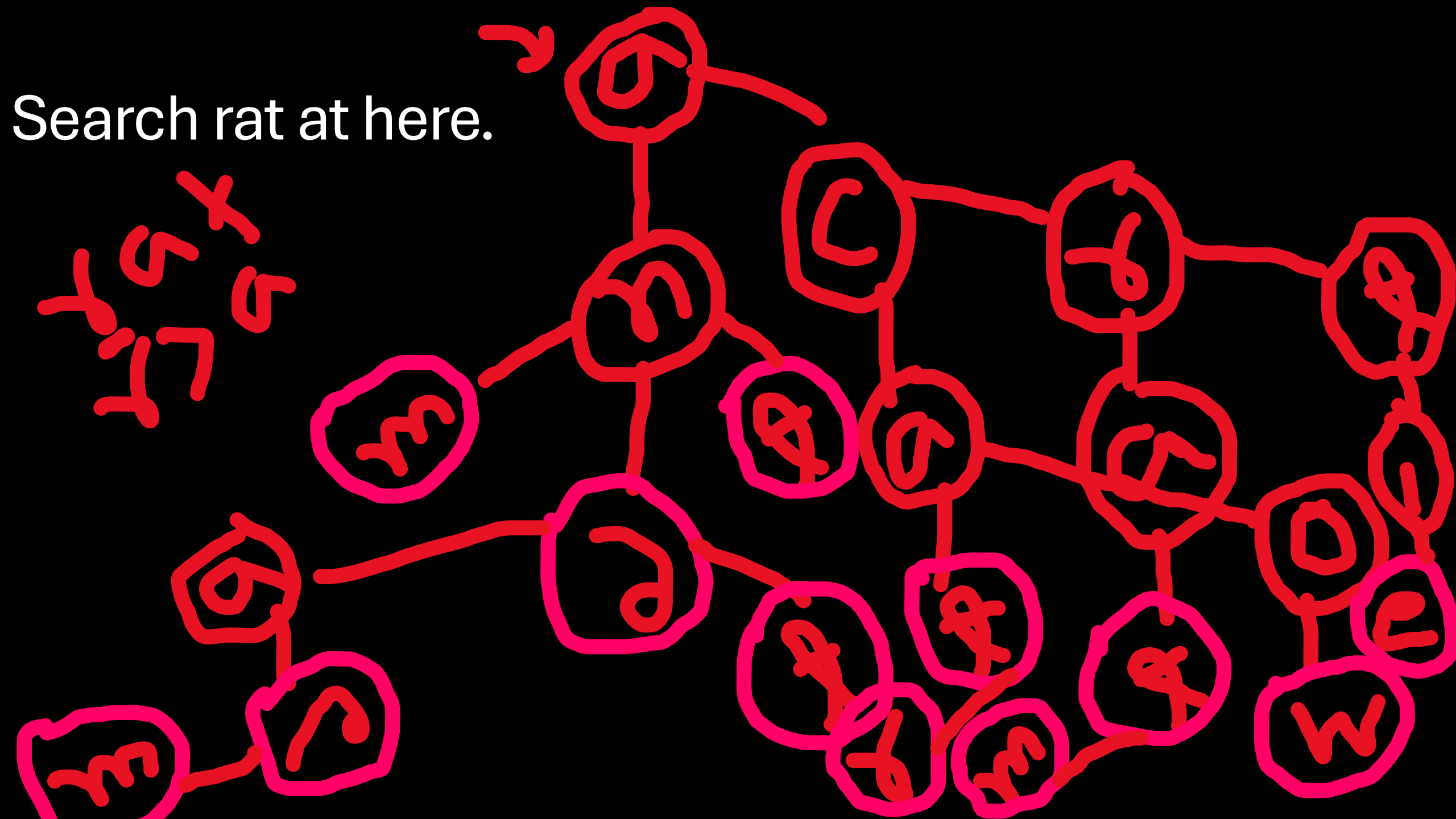


Please try to add tide at here.

Now come to see the operations of searching.

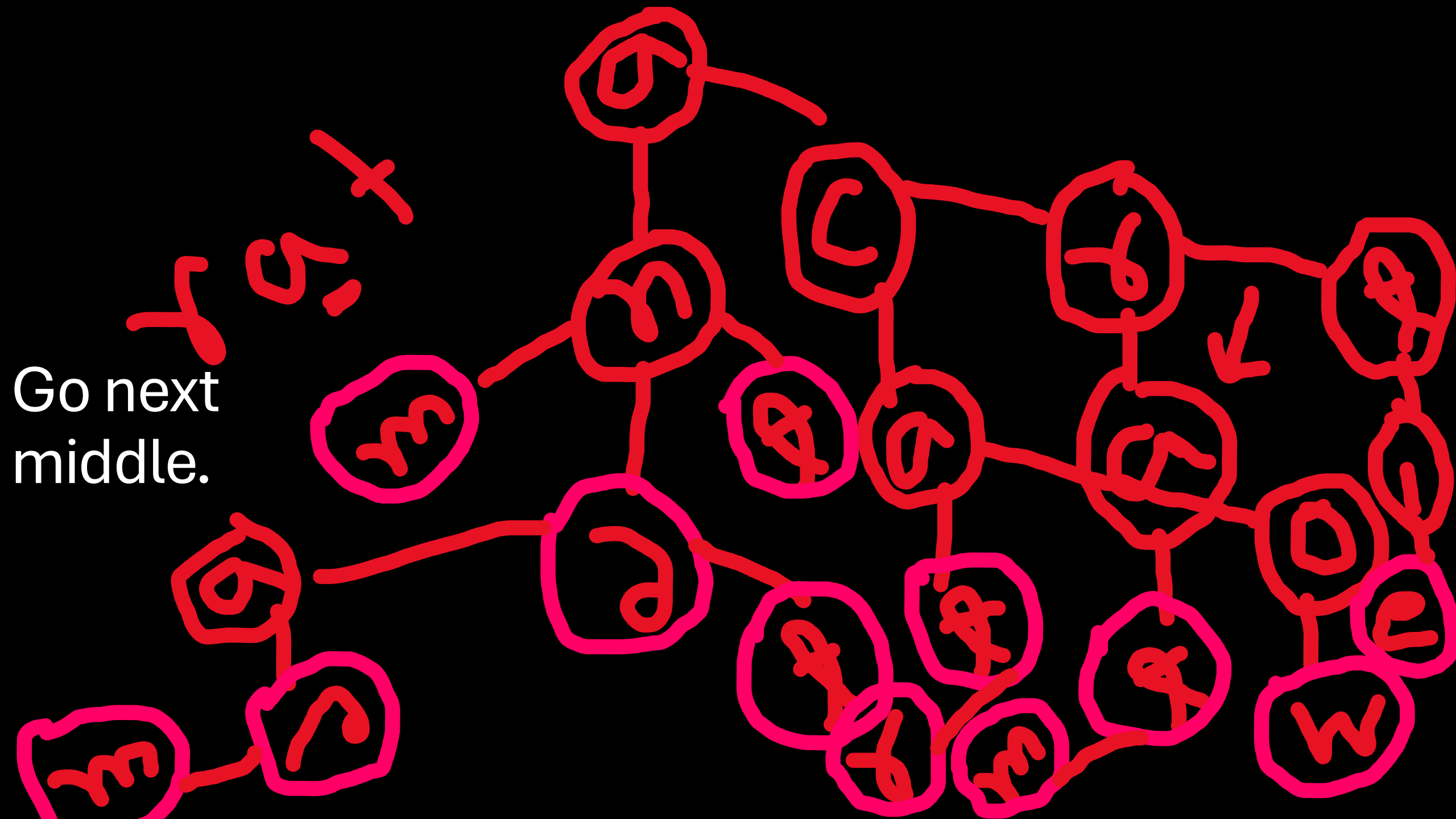
- For searching 🔎
  - we will follow the same rules as bst.
  - If our searching strings character is greater than the present node, then go right.
  - If our present string character is smaller than the present nod, go left.
  - If they are equal go to the middle and continue searching until find the end of the character of the string and that is marked as true at there.

Search rat at here.

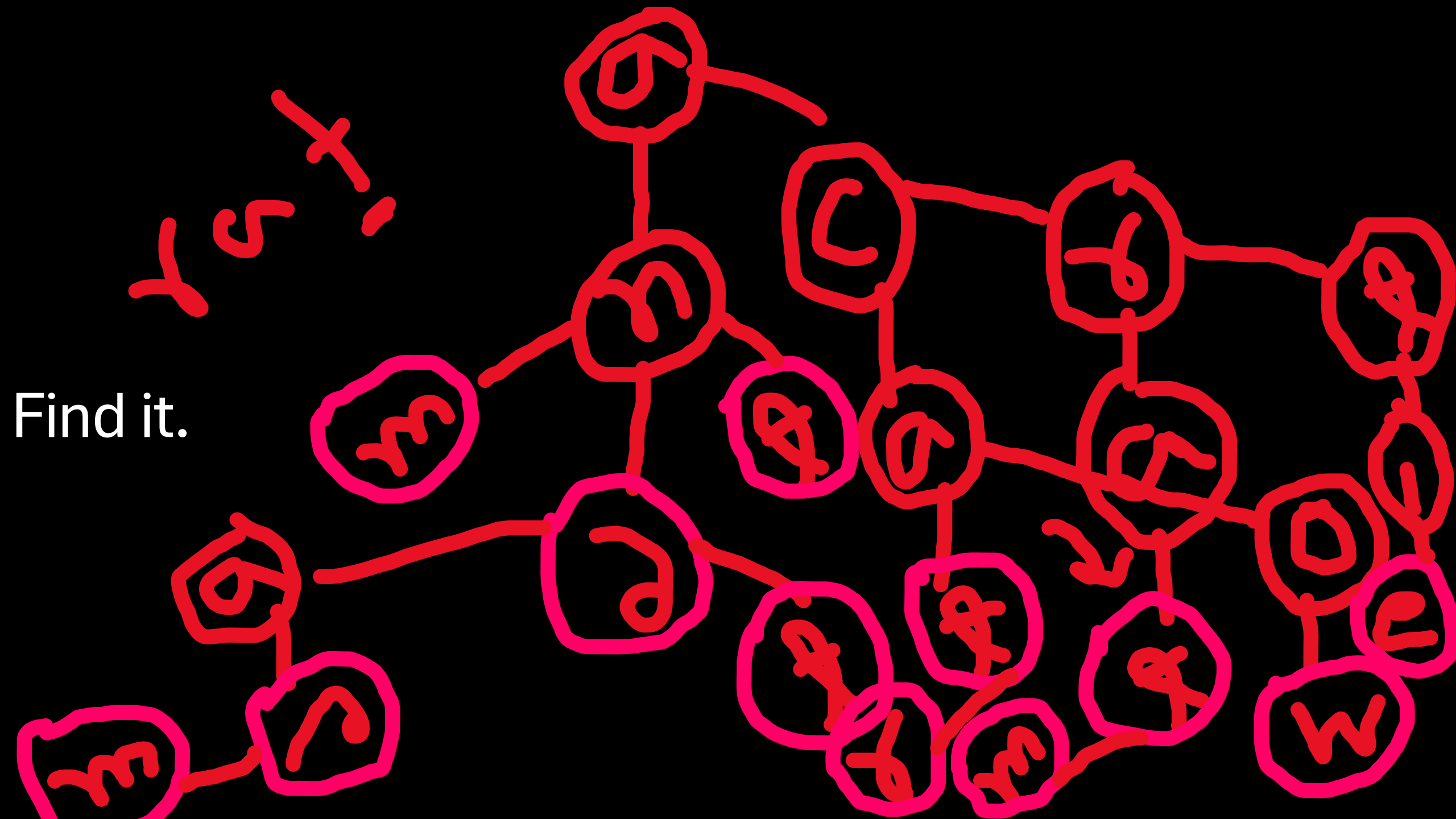


Both of them same

Go next
middle.



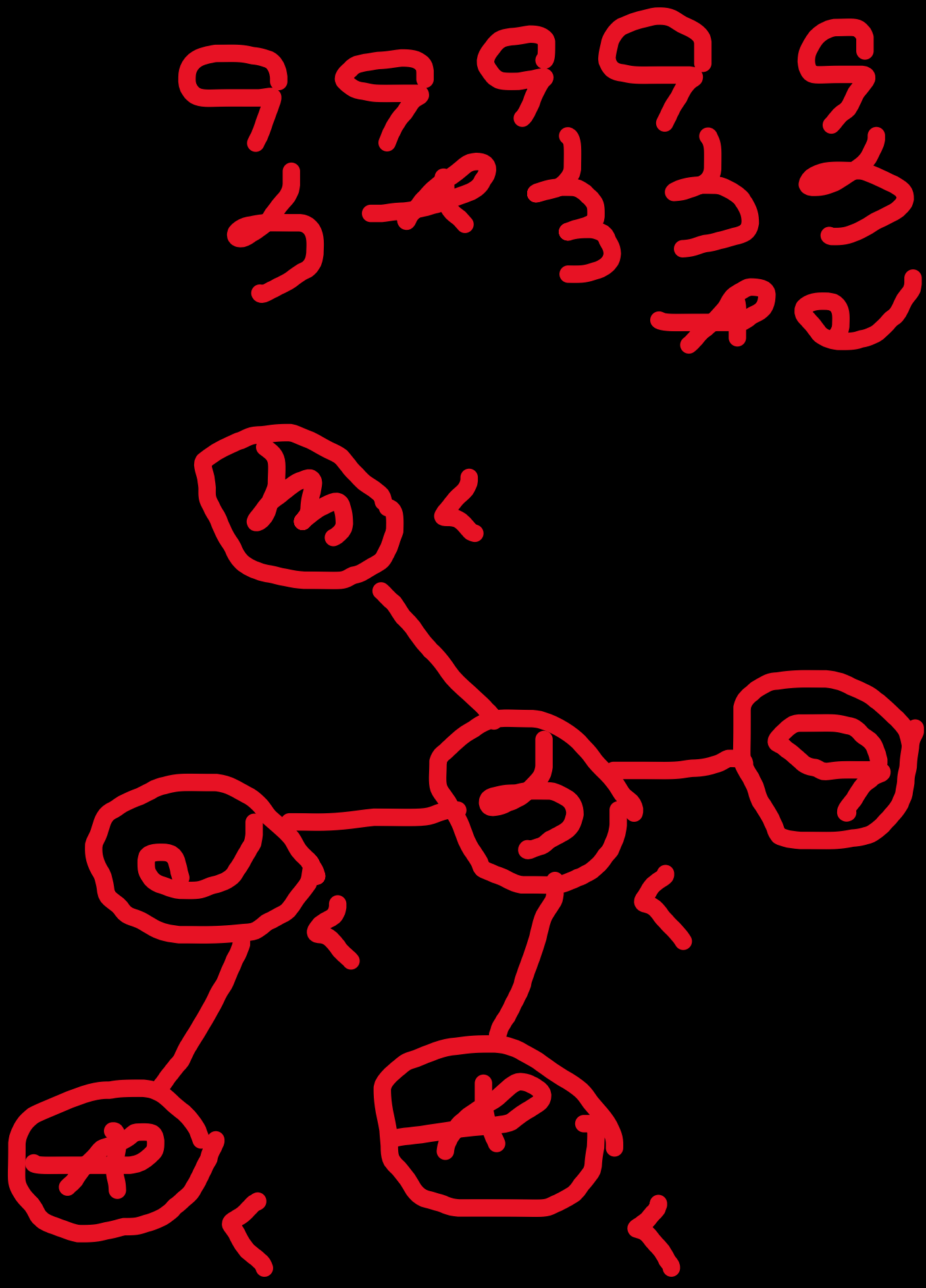
Find it.

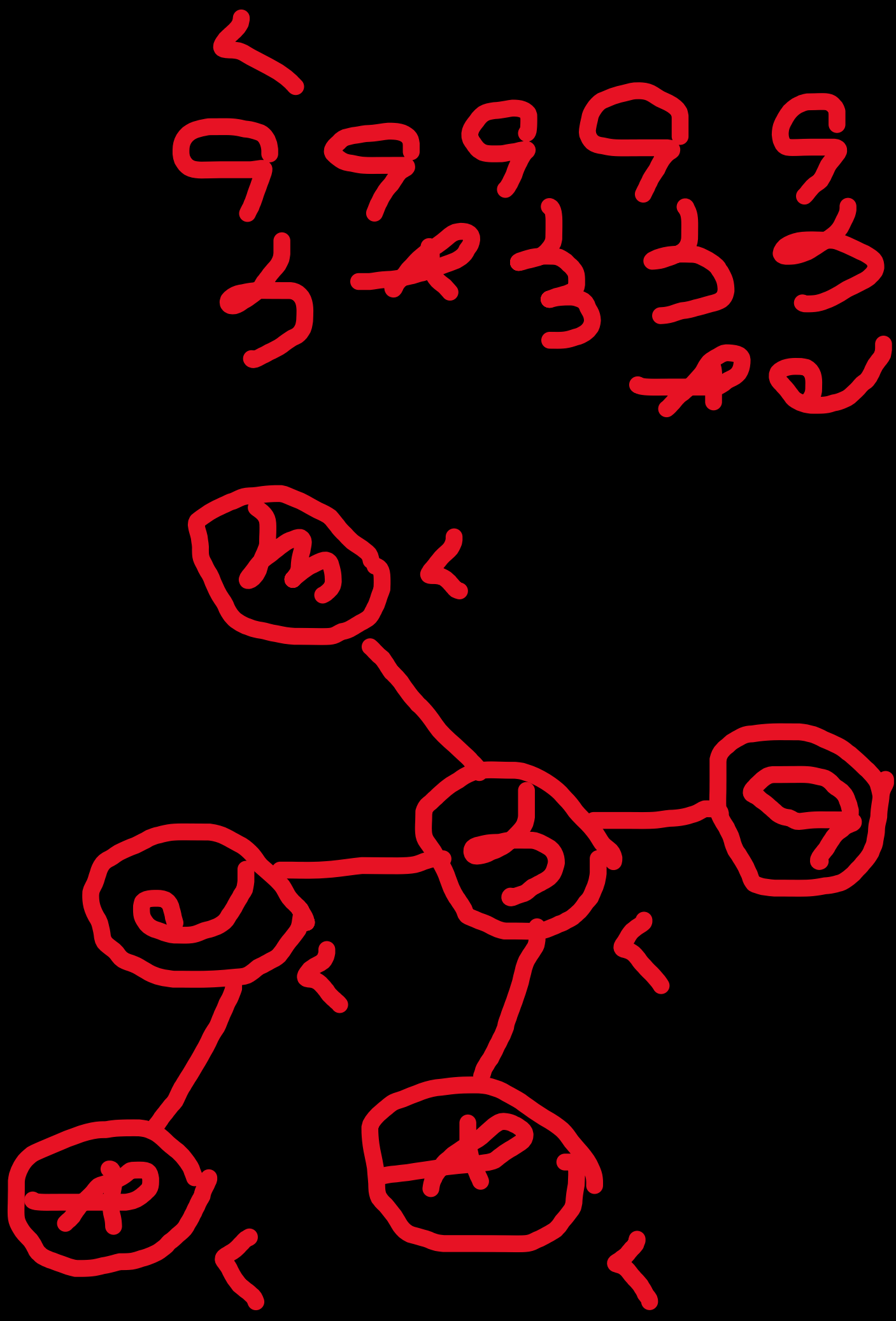


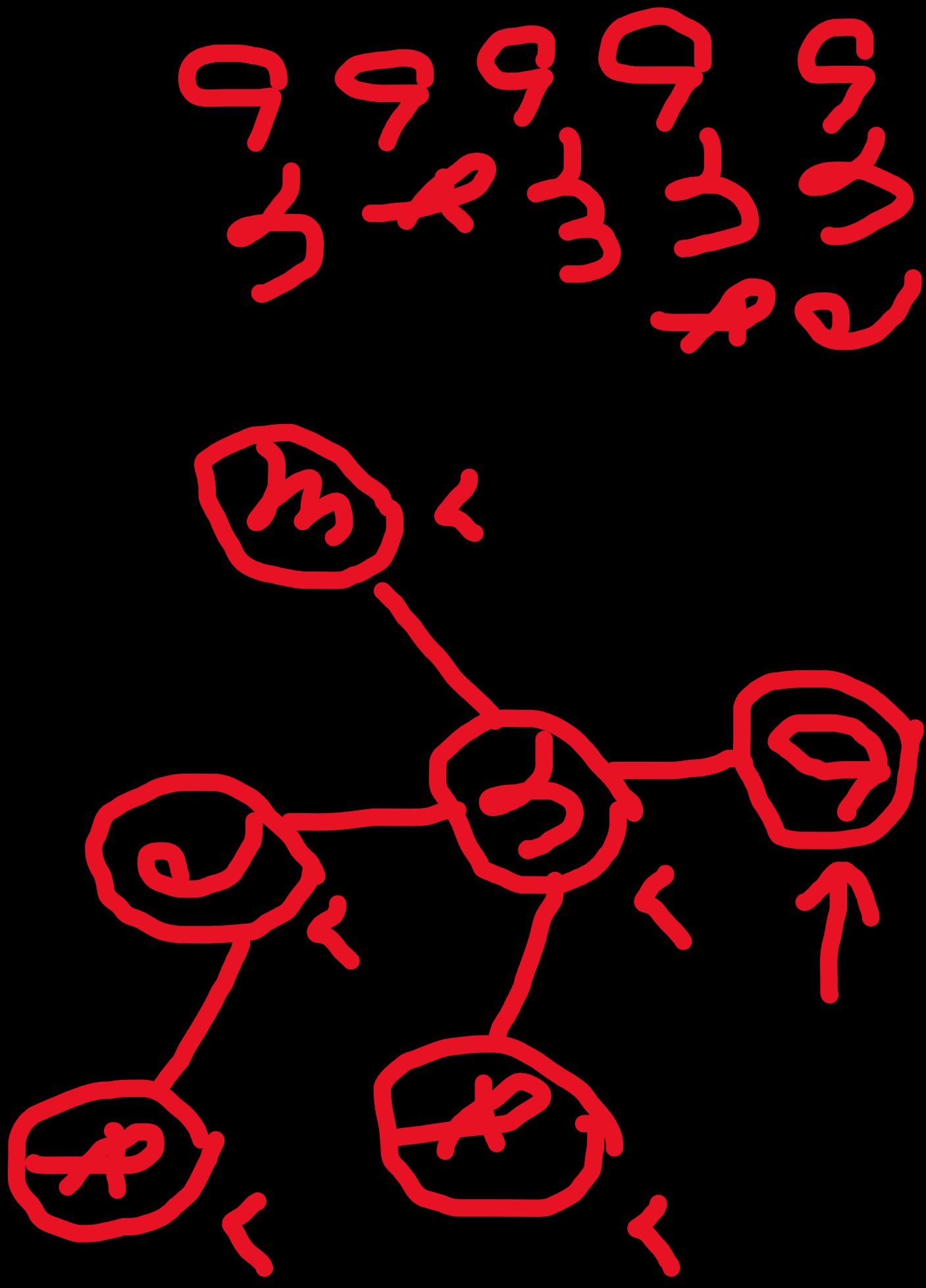
Now come for the delete operations.

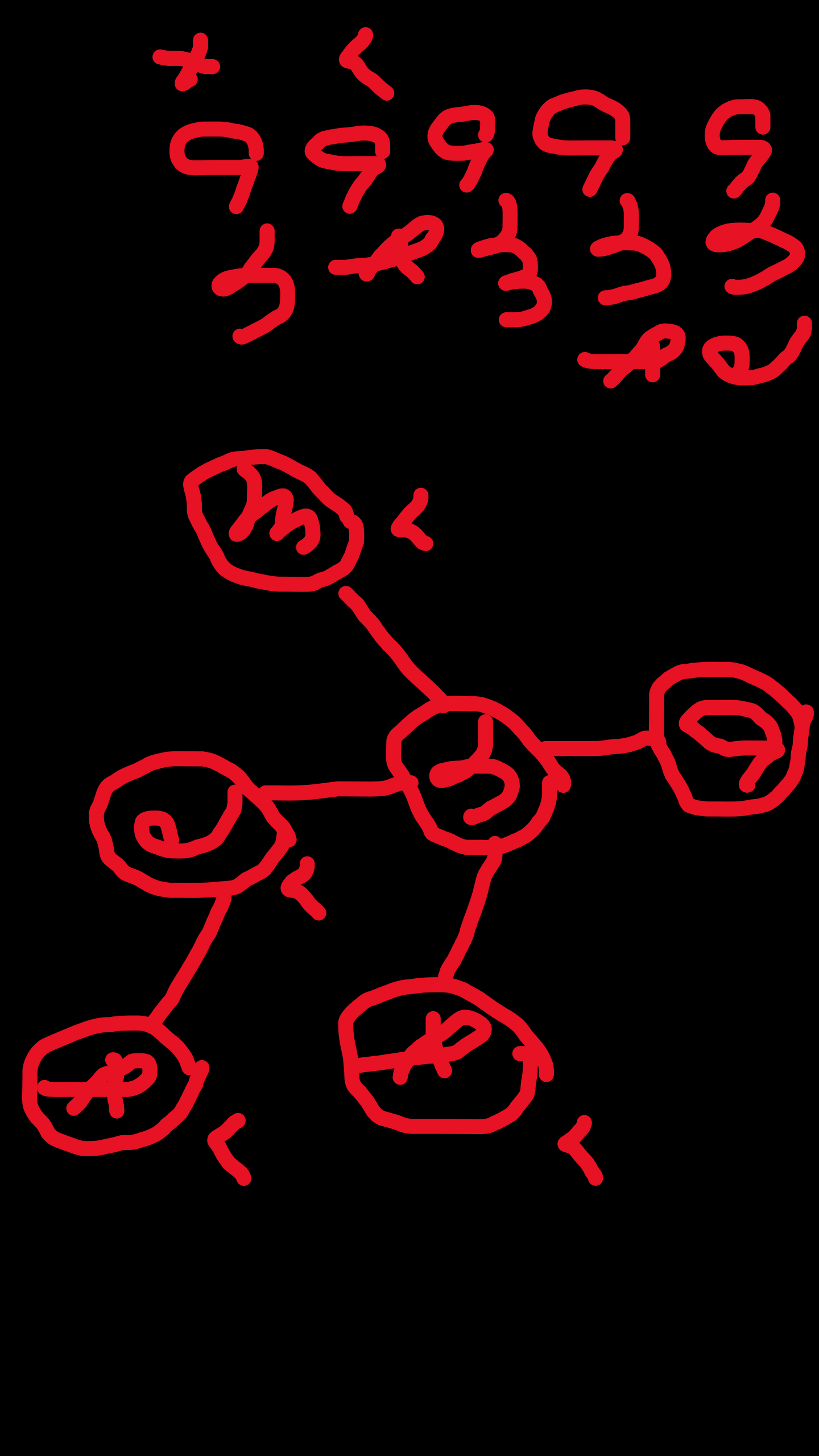
- This is as like as same as the bst and trie data structure.
  - First time find the word we want to delete. Then mark the end character as false if this is prefix of other string.
  - If no other string use that node, then just delete it.

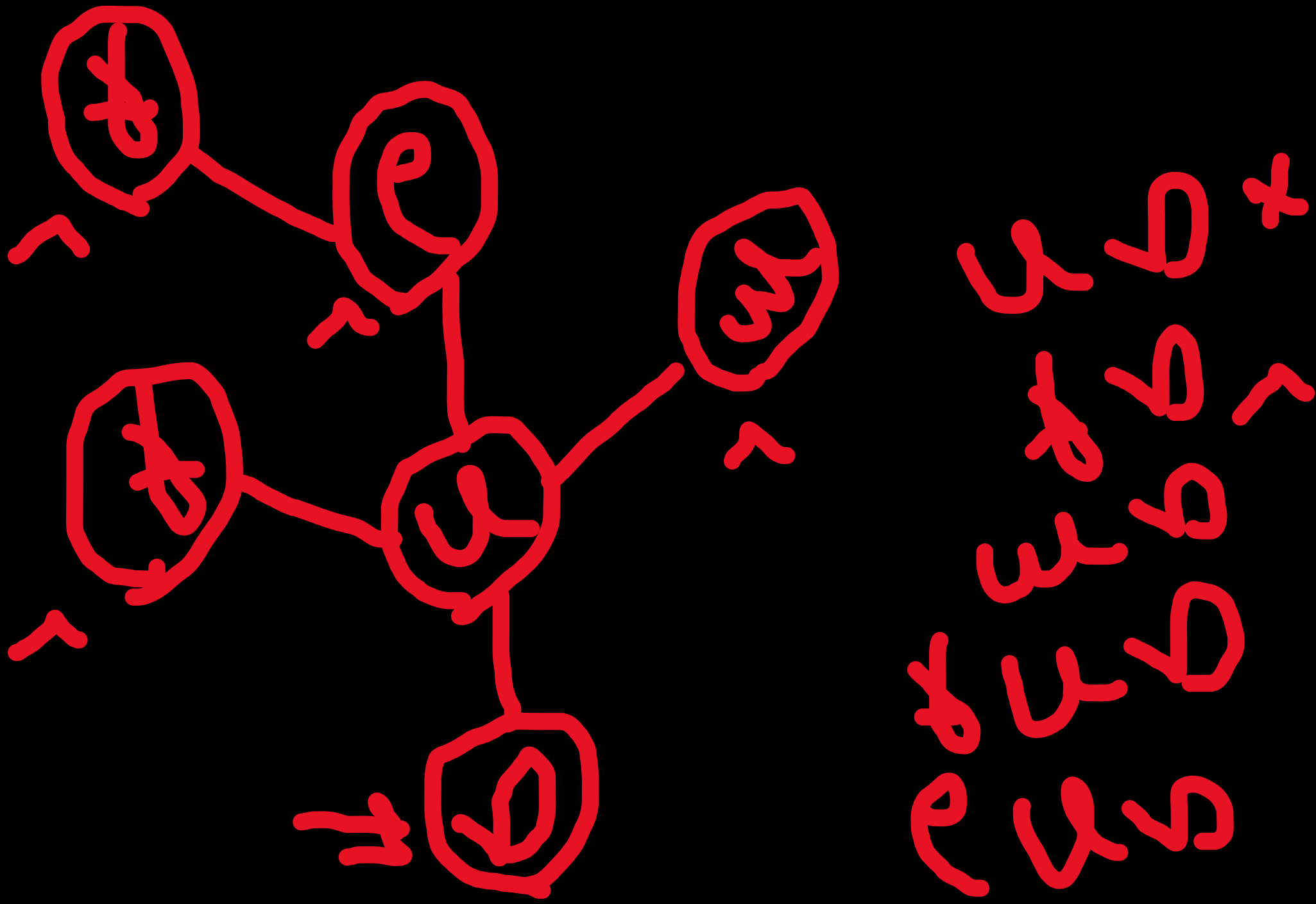
Let's solve an example.

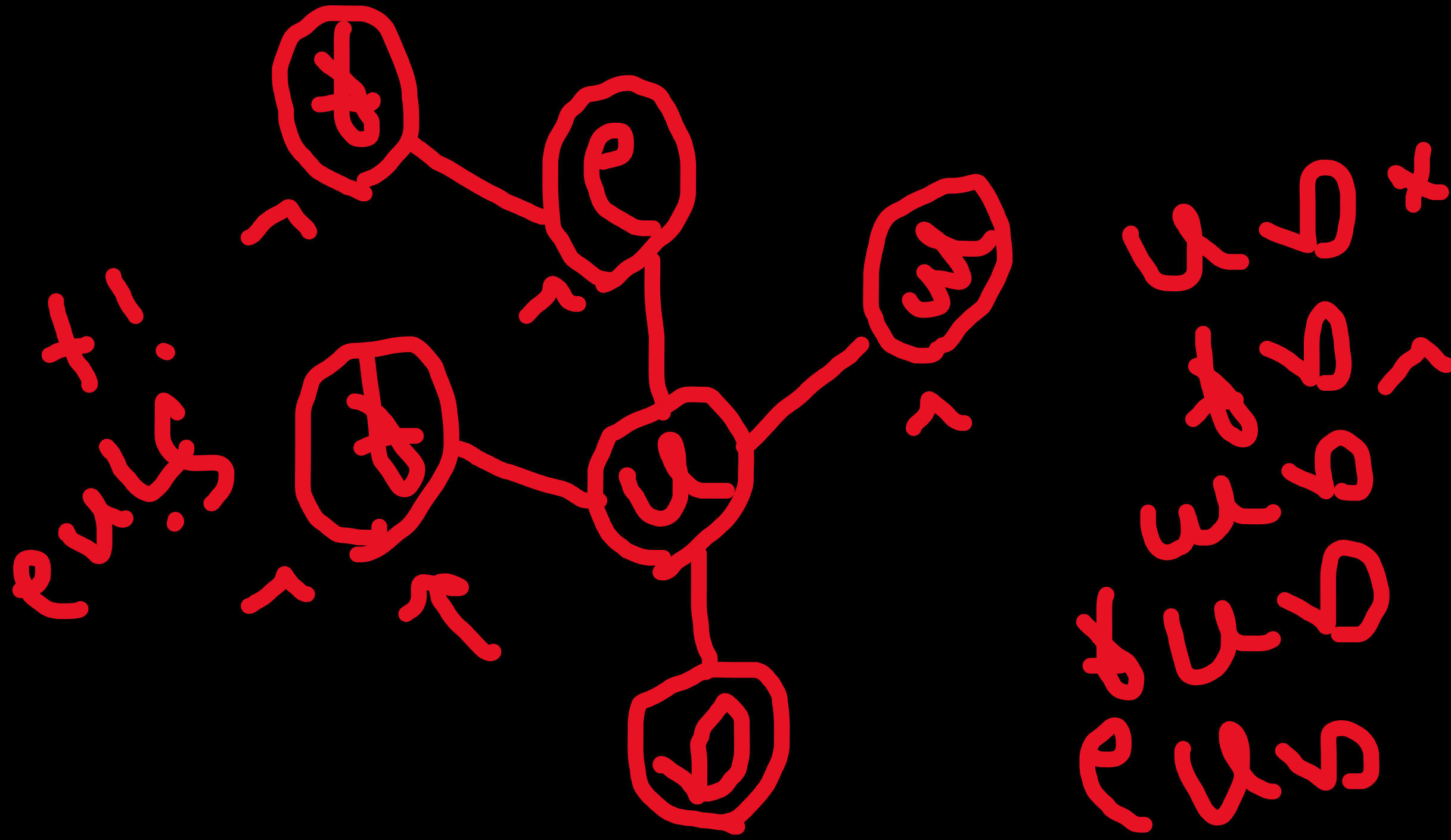


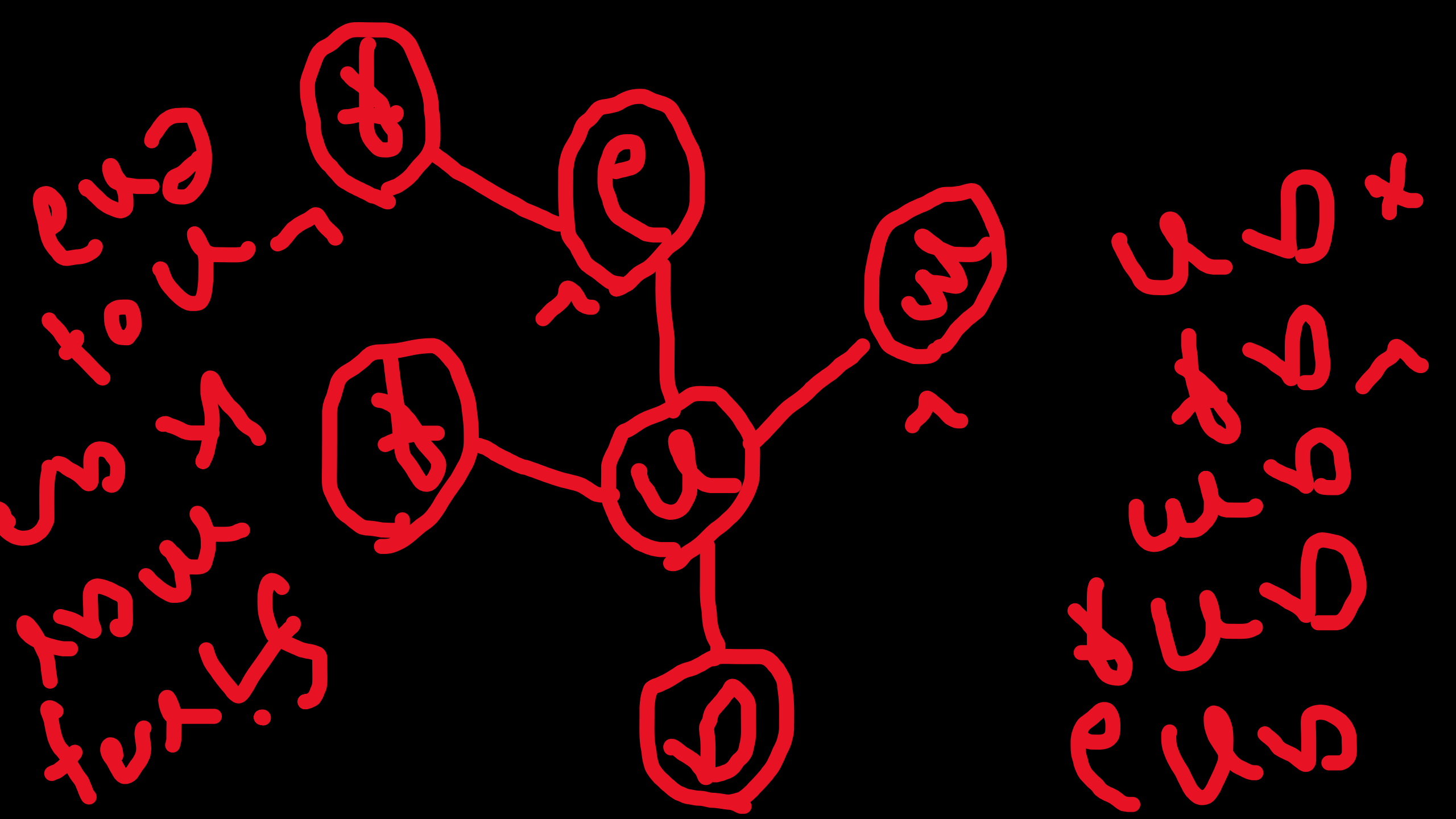












Now, let's go to the implementations