Queues ADT

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today’s Plan

Announcements and Syllabus Check

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

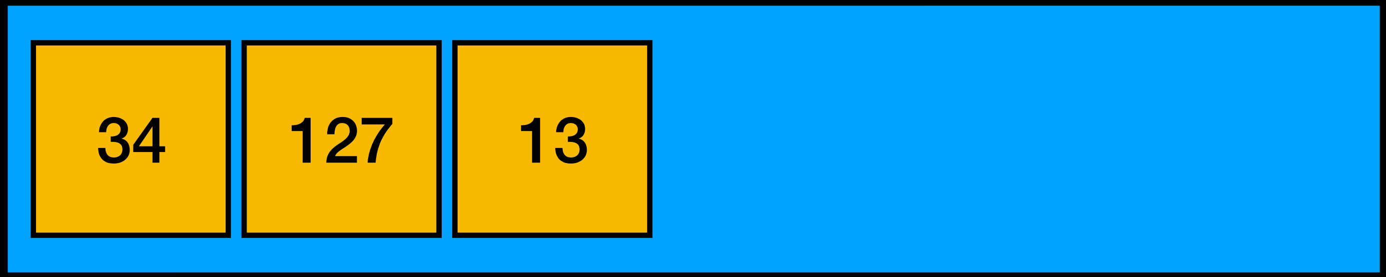
or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

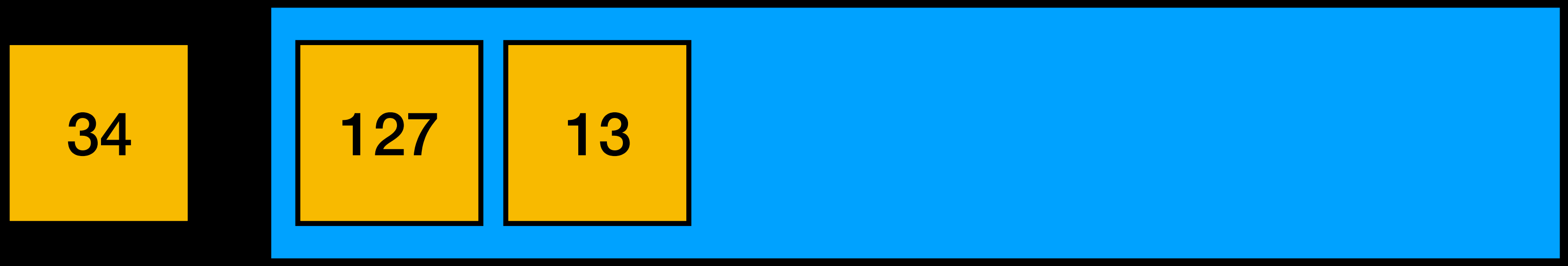


Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line



Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

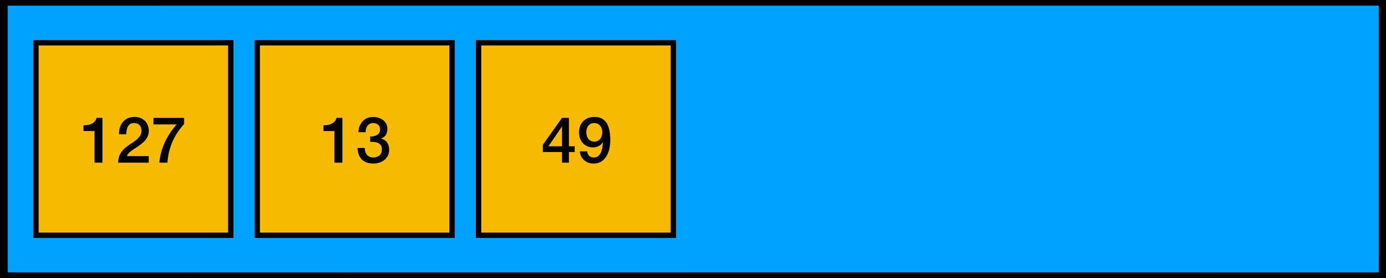
or dequeued from the front of the line

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line



Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

FIFO: First In First Out

Only front of queue is accessible (front), no other objects in the queue are visible

Queue Applications

Generating all substrings

Recognizing Palindromes

Print (or any other) queue

Genius Bar Simulation

- now we could implement it to be fair!!!

Queue Applications

Generating all substrings

Recognizing Palindromes

Print (or any other) queue

Genius Bar Simulation

- now we could implement it to be fair!!!

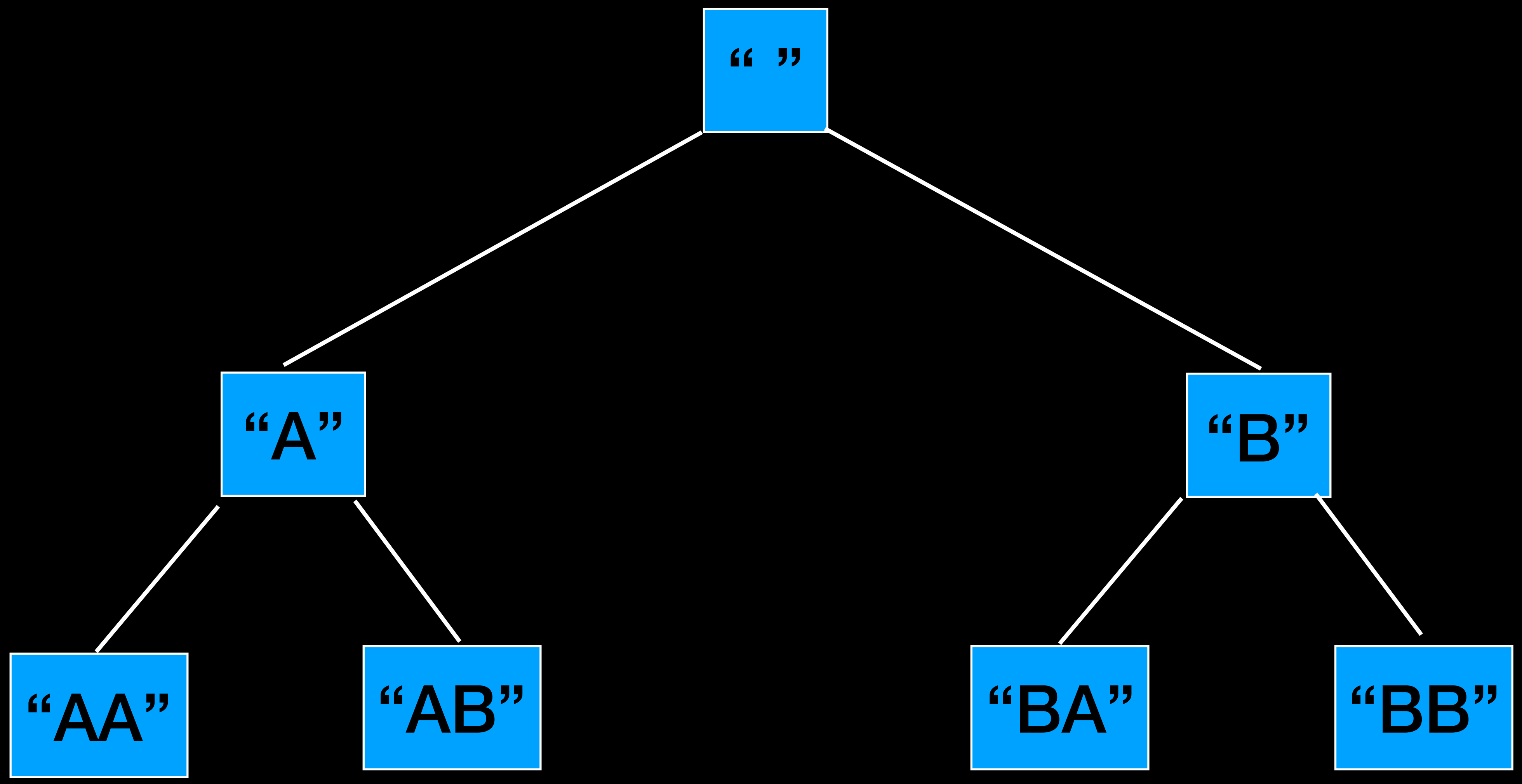
Generating all substrings

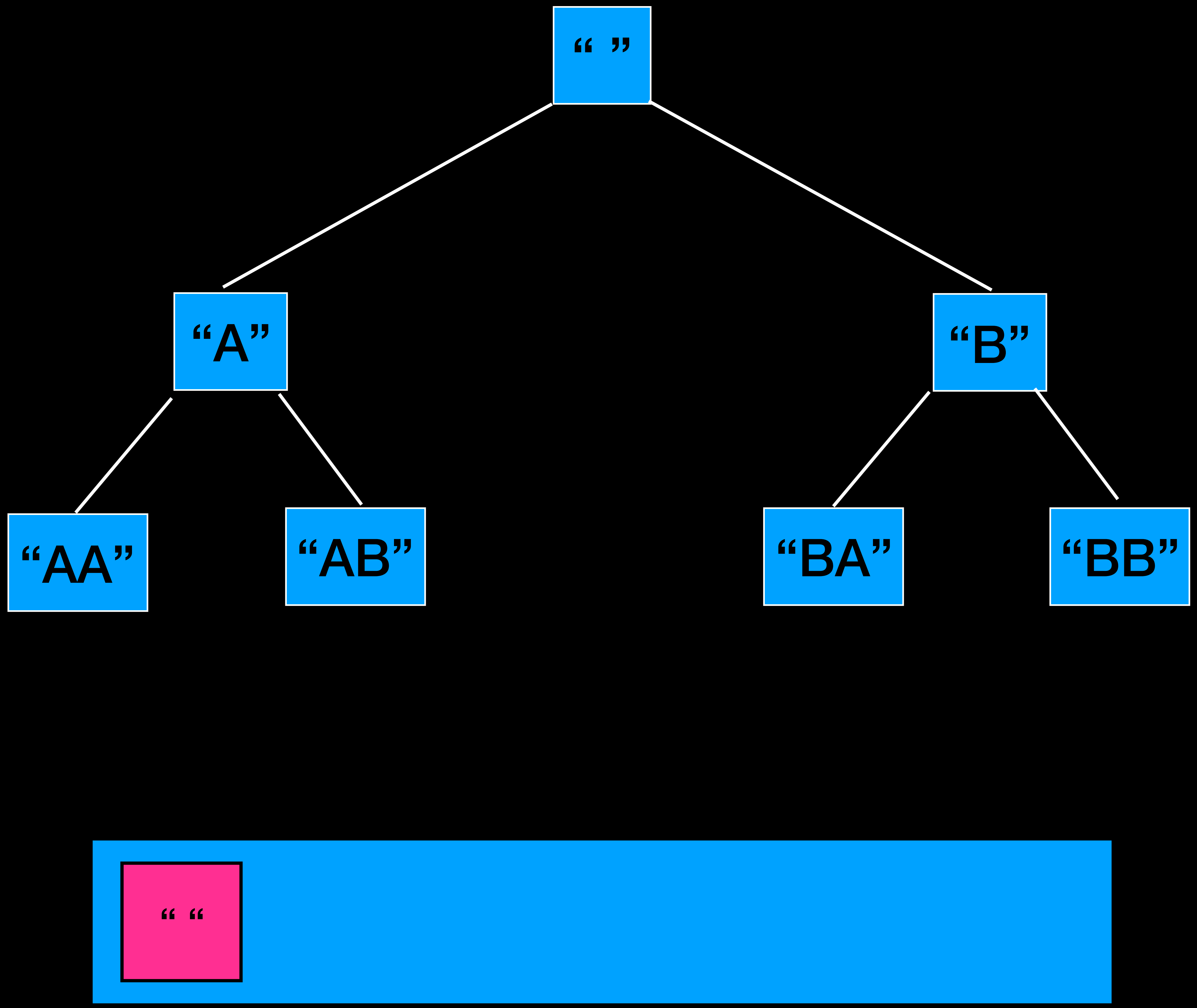
Generate all possible strings **up to** some fixed length **n with repetition**

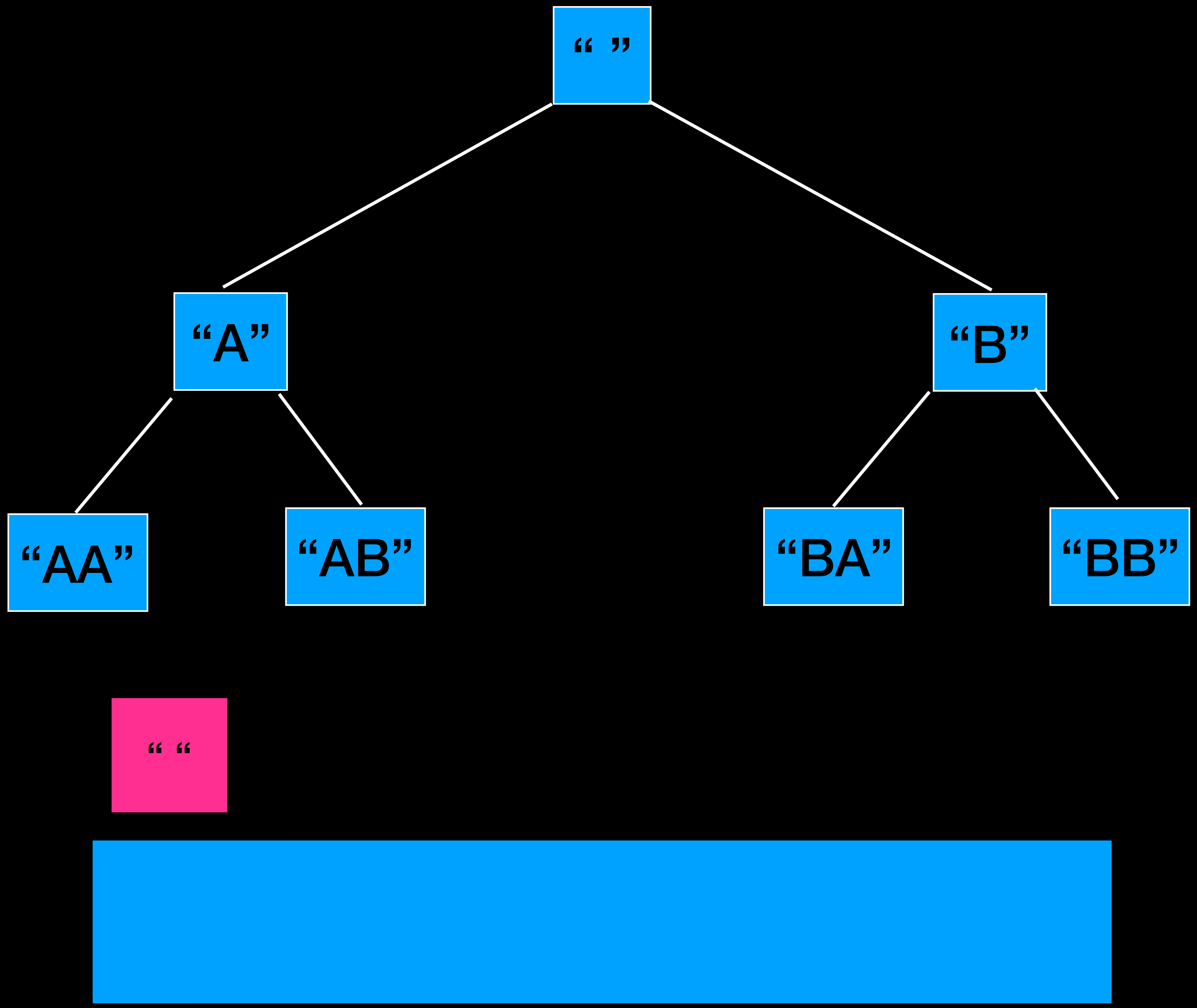
We saw how to do something similar recursively (generate permutations of fixed size n no repetition)

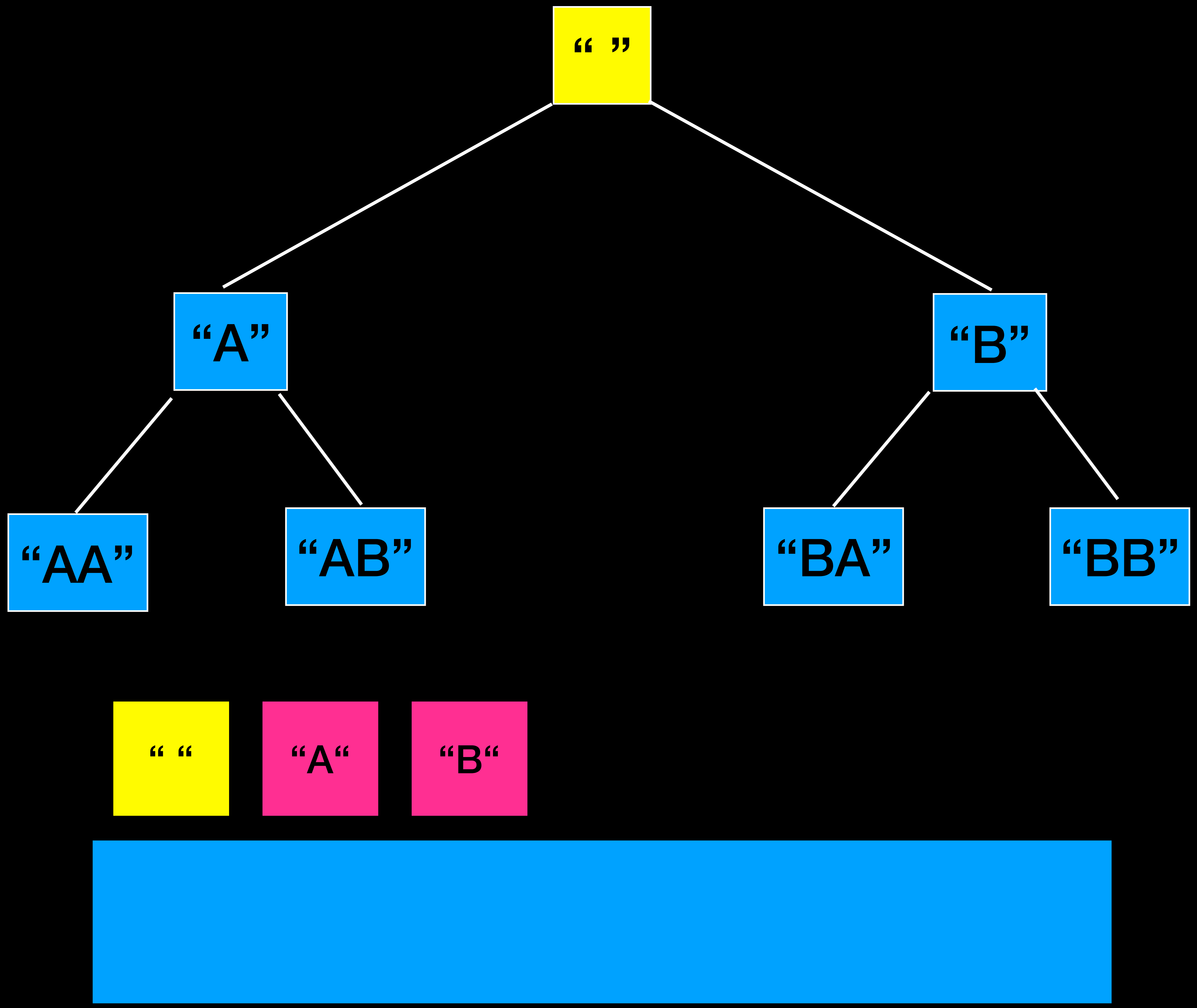
How might we do it with a queue?

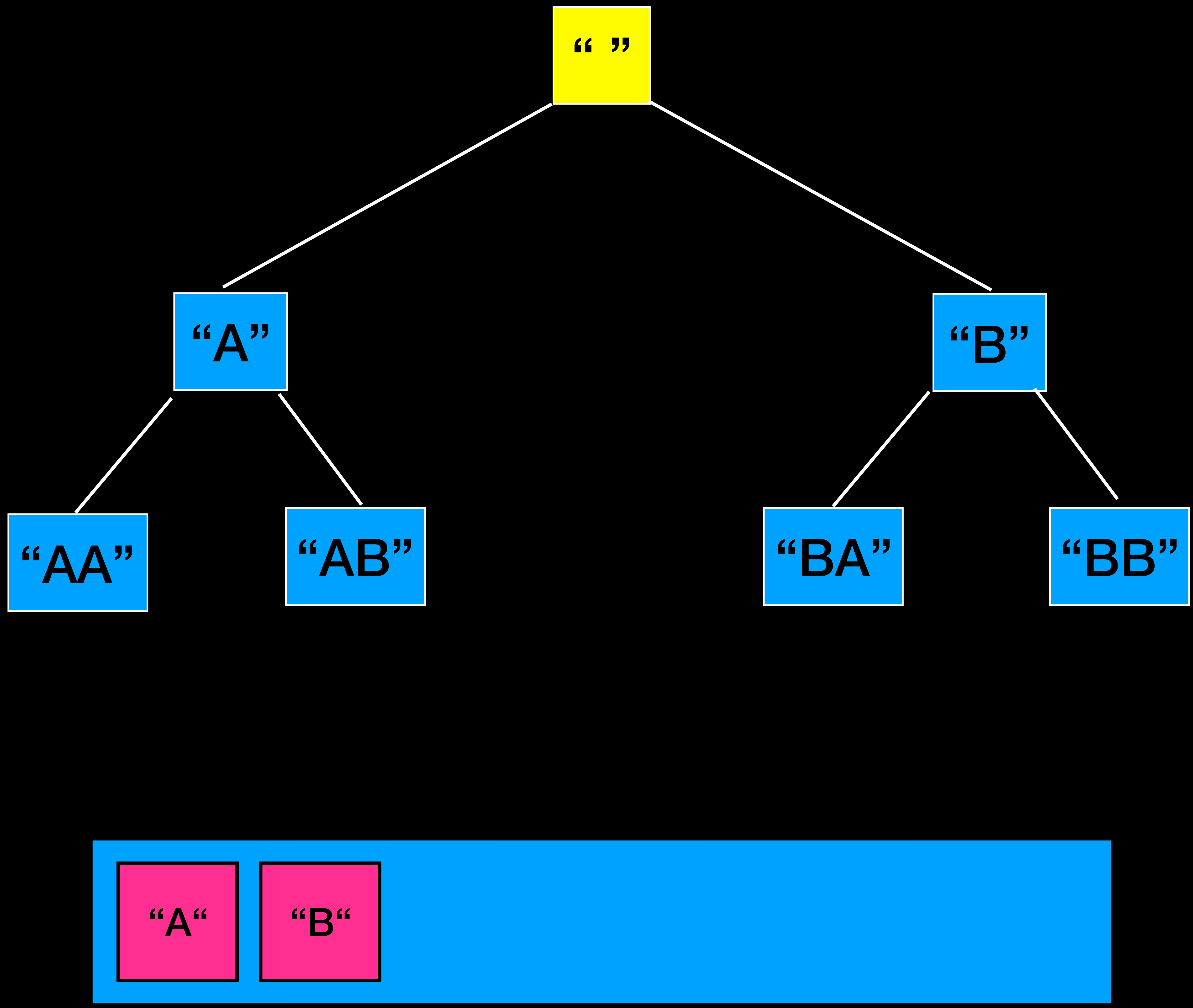
Example simplified to n = 2 and only letters A and B

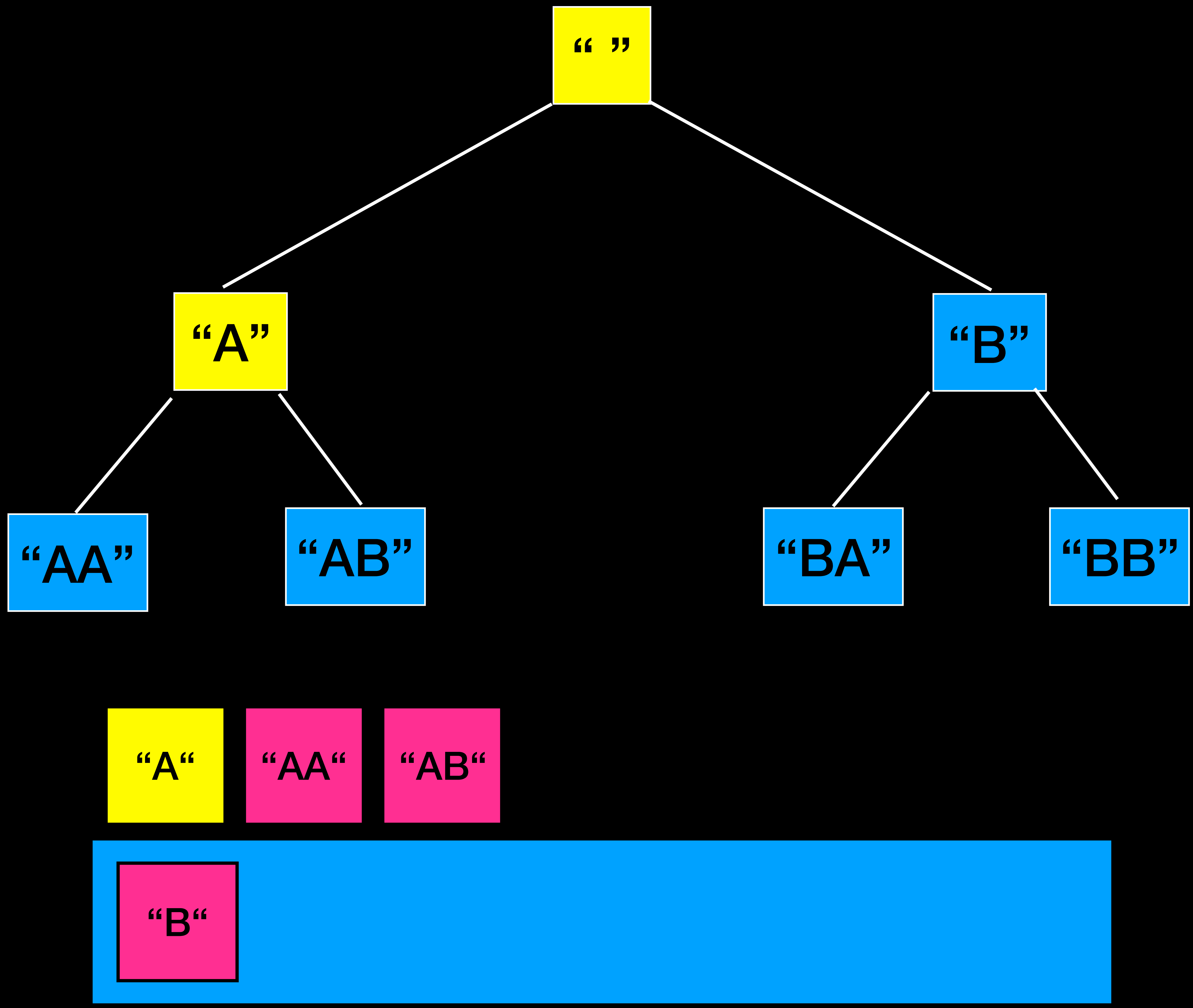

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

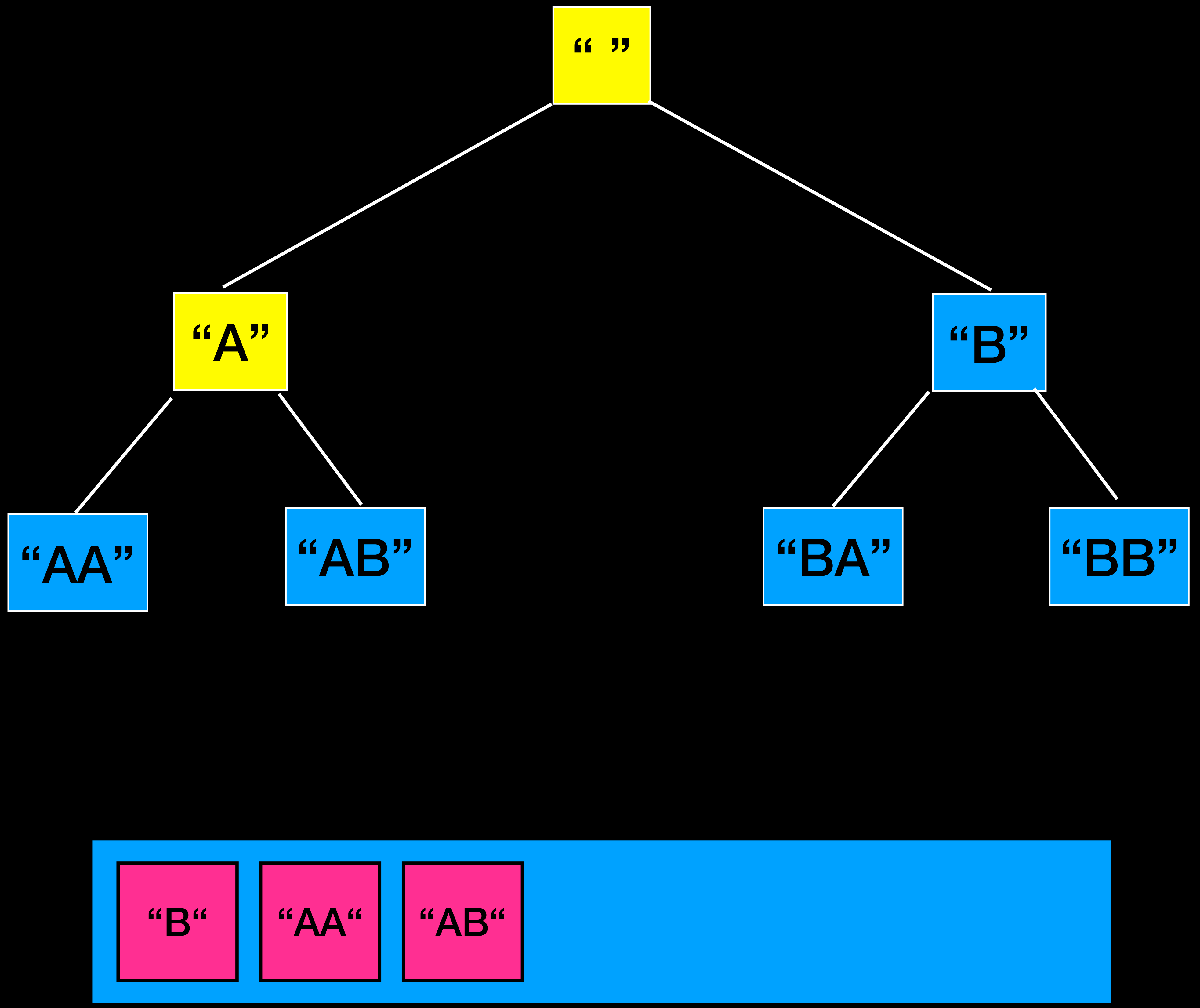

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“ “

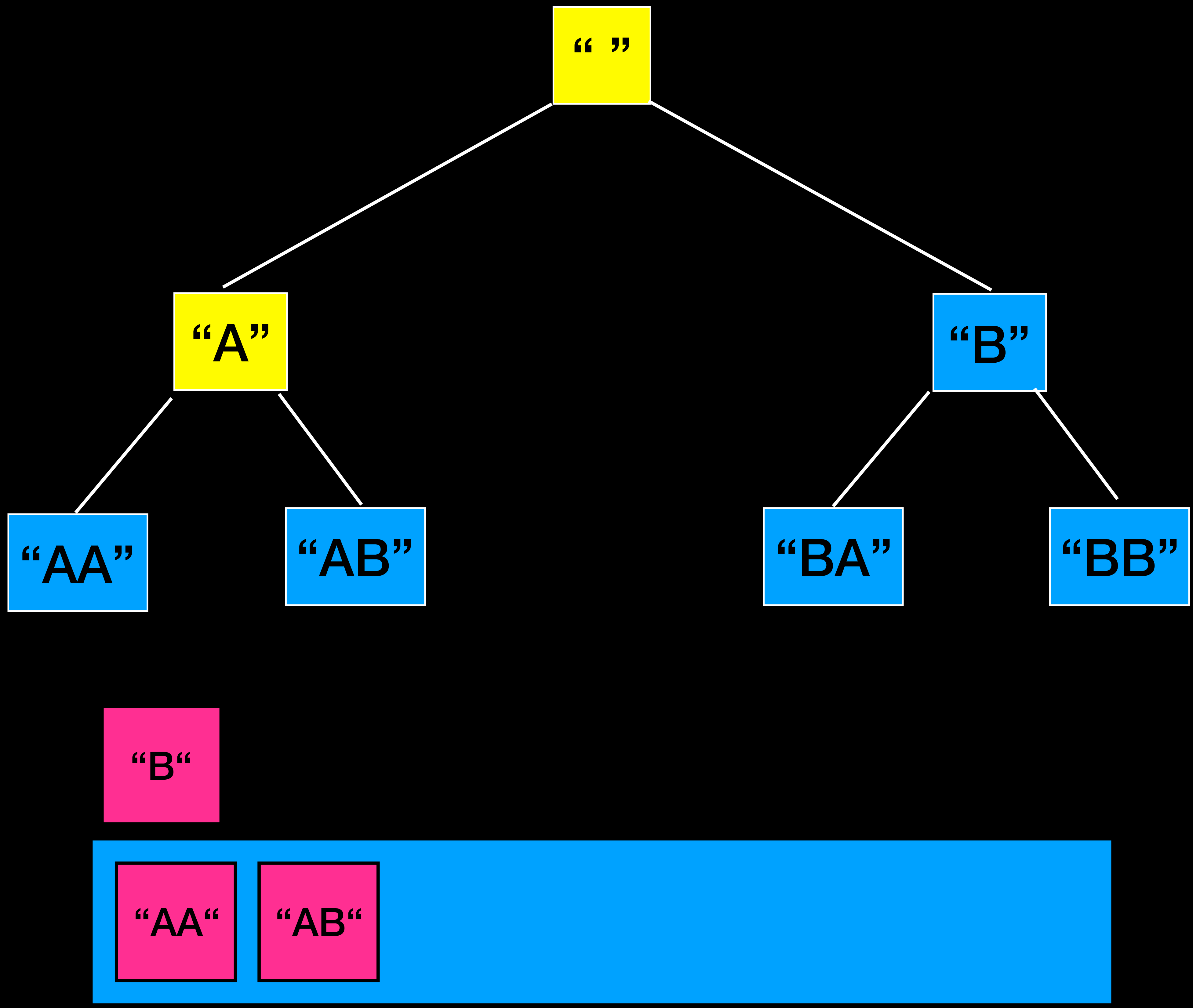

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“ “

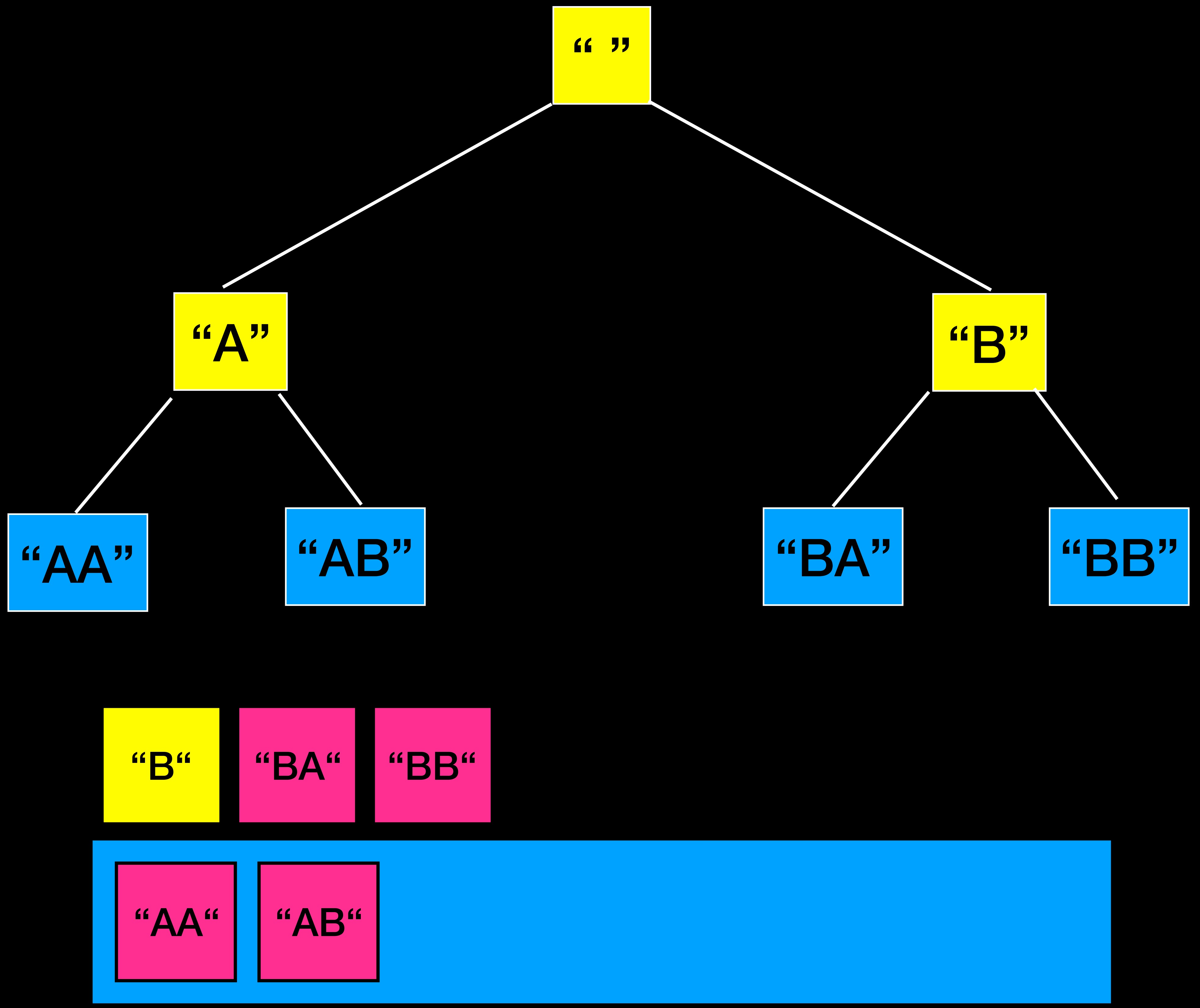

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“ “
“A“
“B“

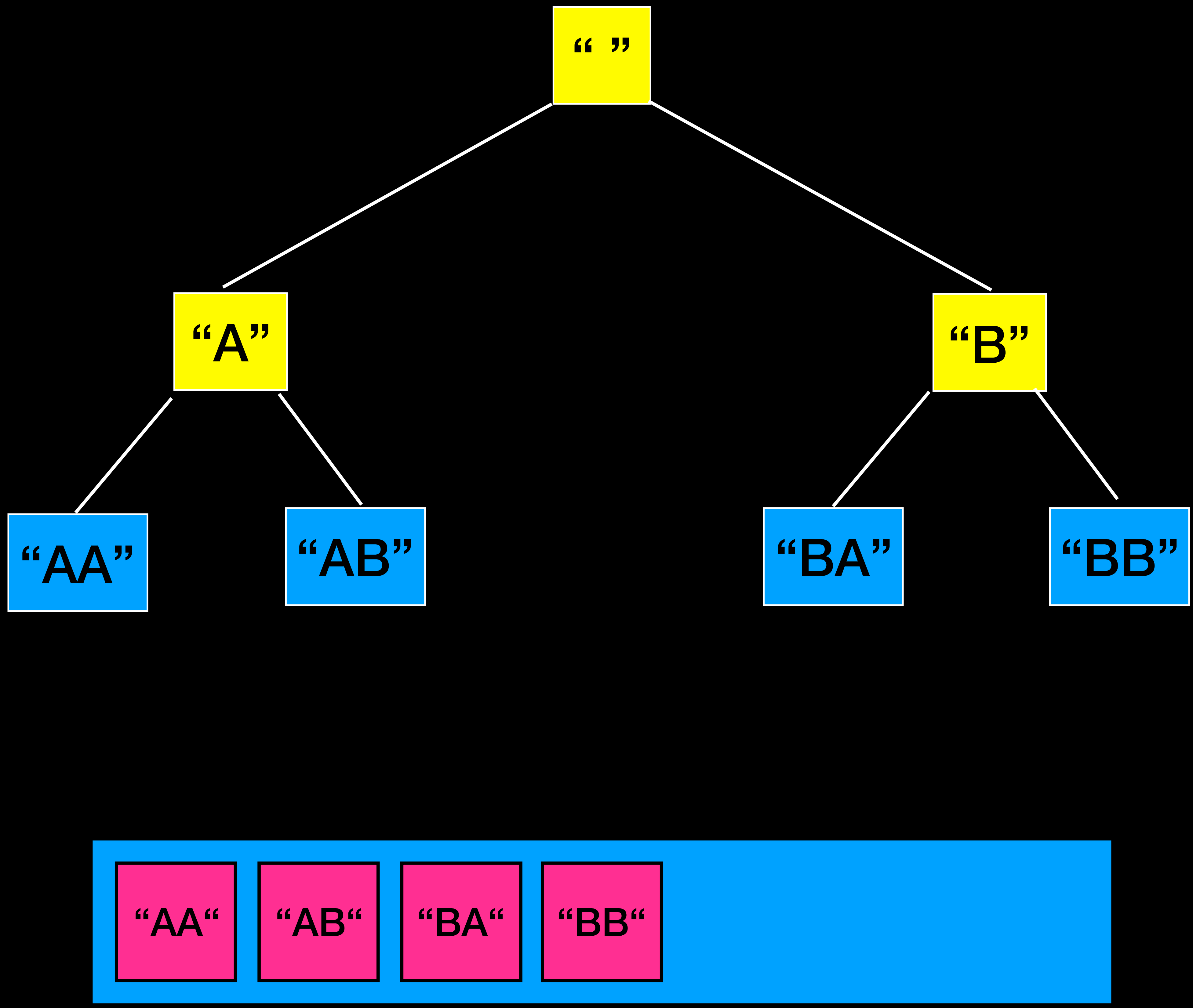

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“A“
“B“

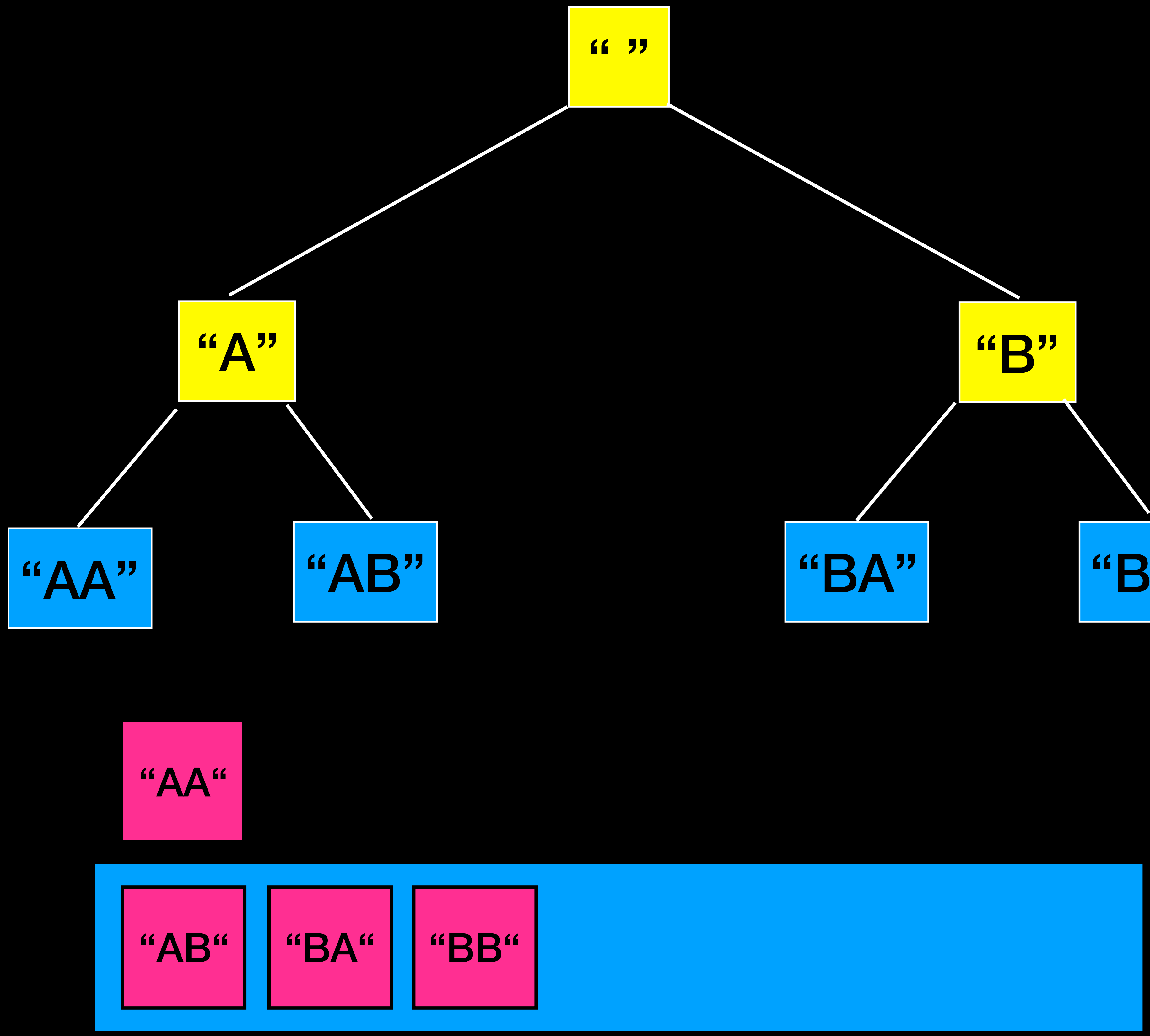

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“A“
“AA“
“AB“
“B“

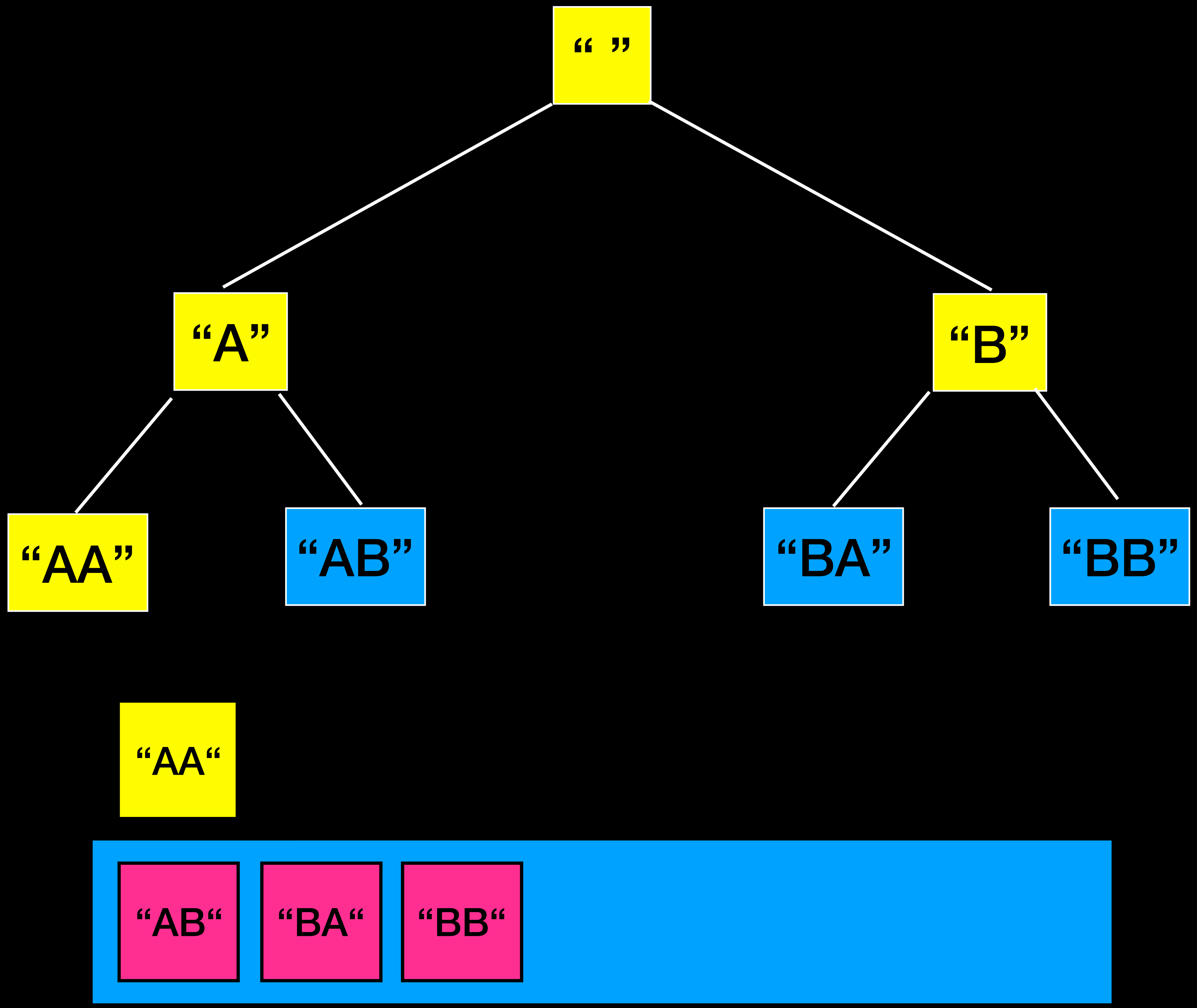

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“B“
“AA“
“AB“

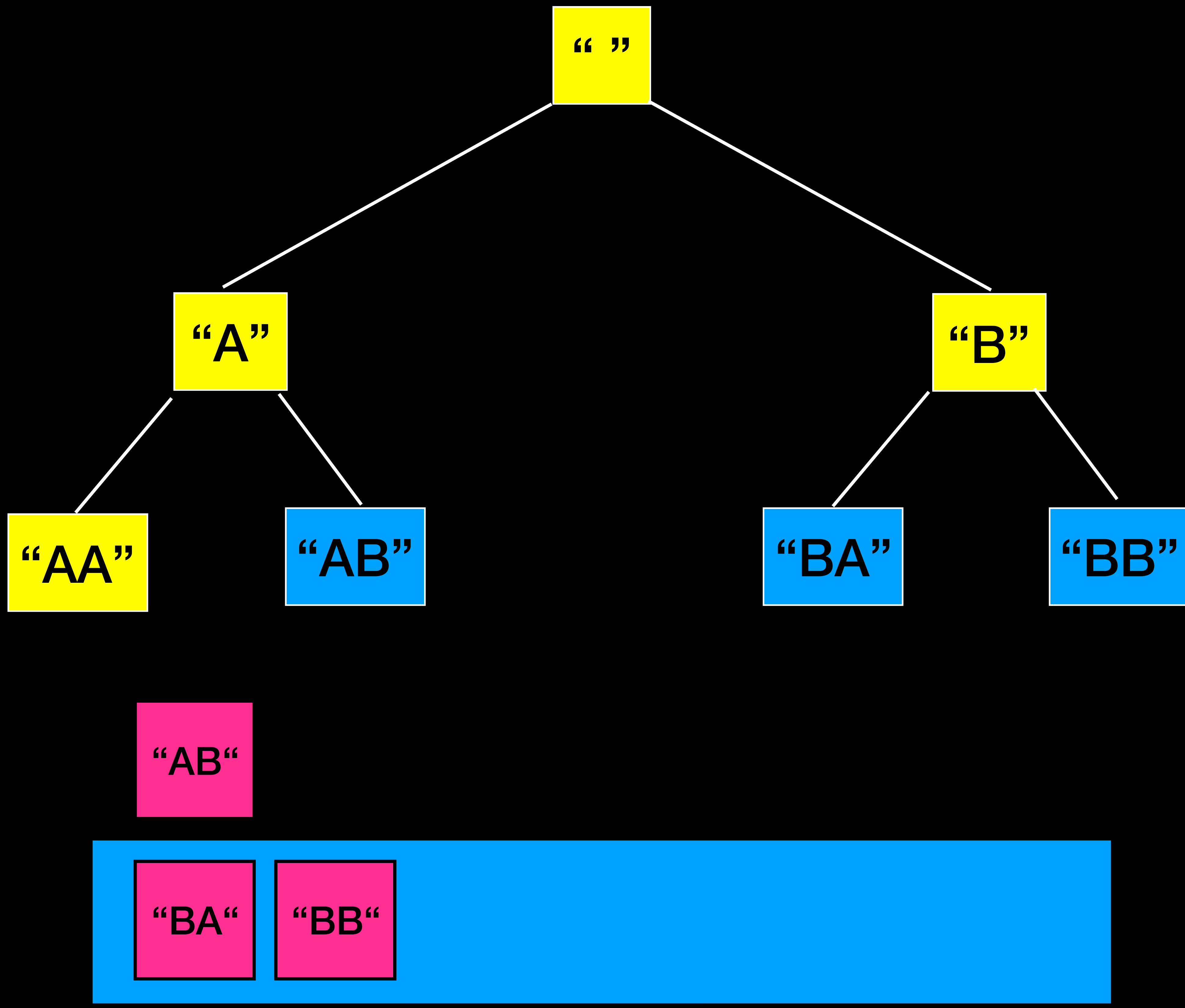

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“B“
“AA“
“AB“

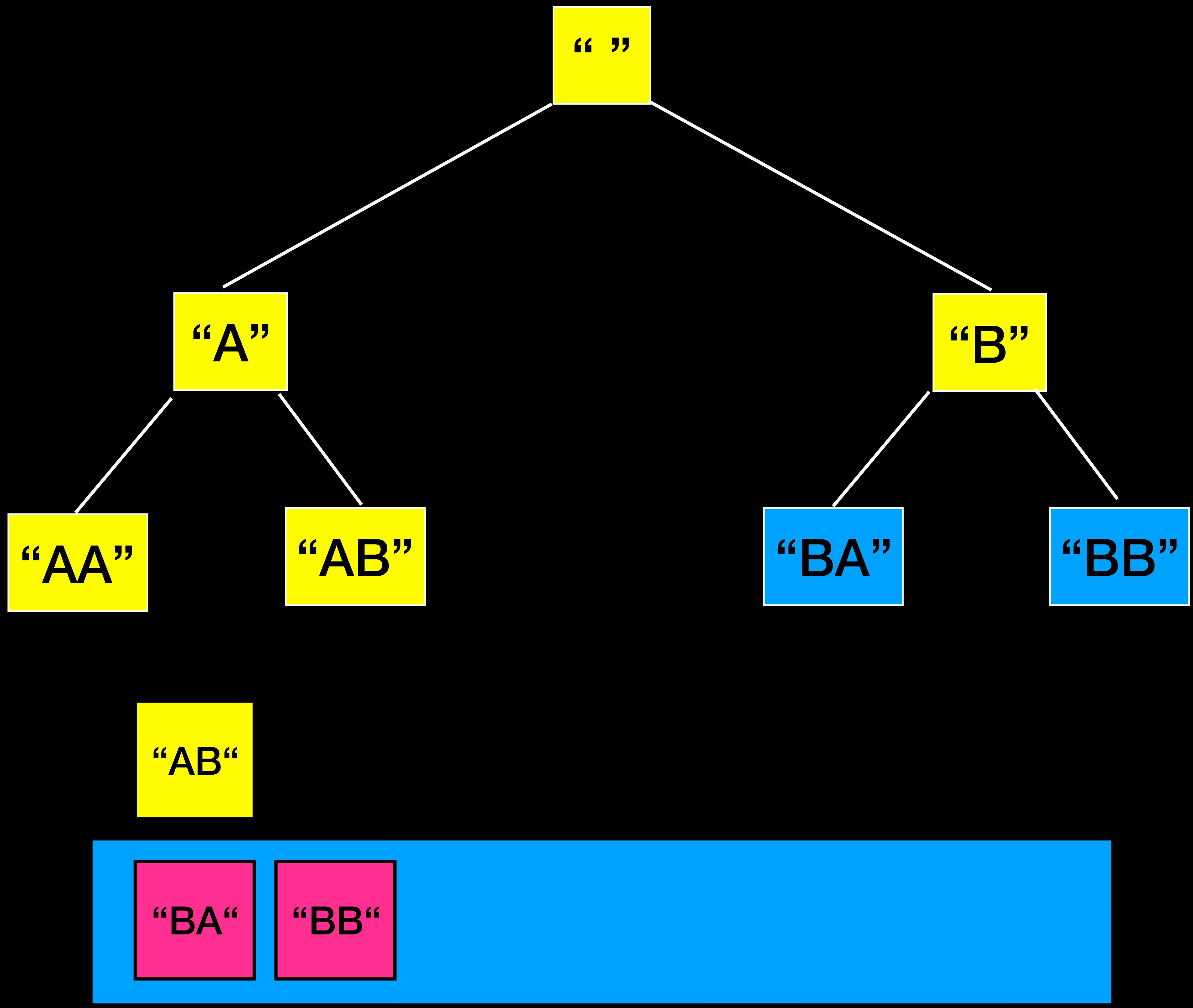

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“B“
“BA“
“BB“
“AA“
“AB“

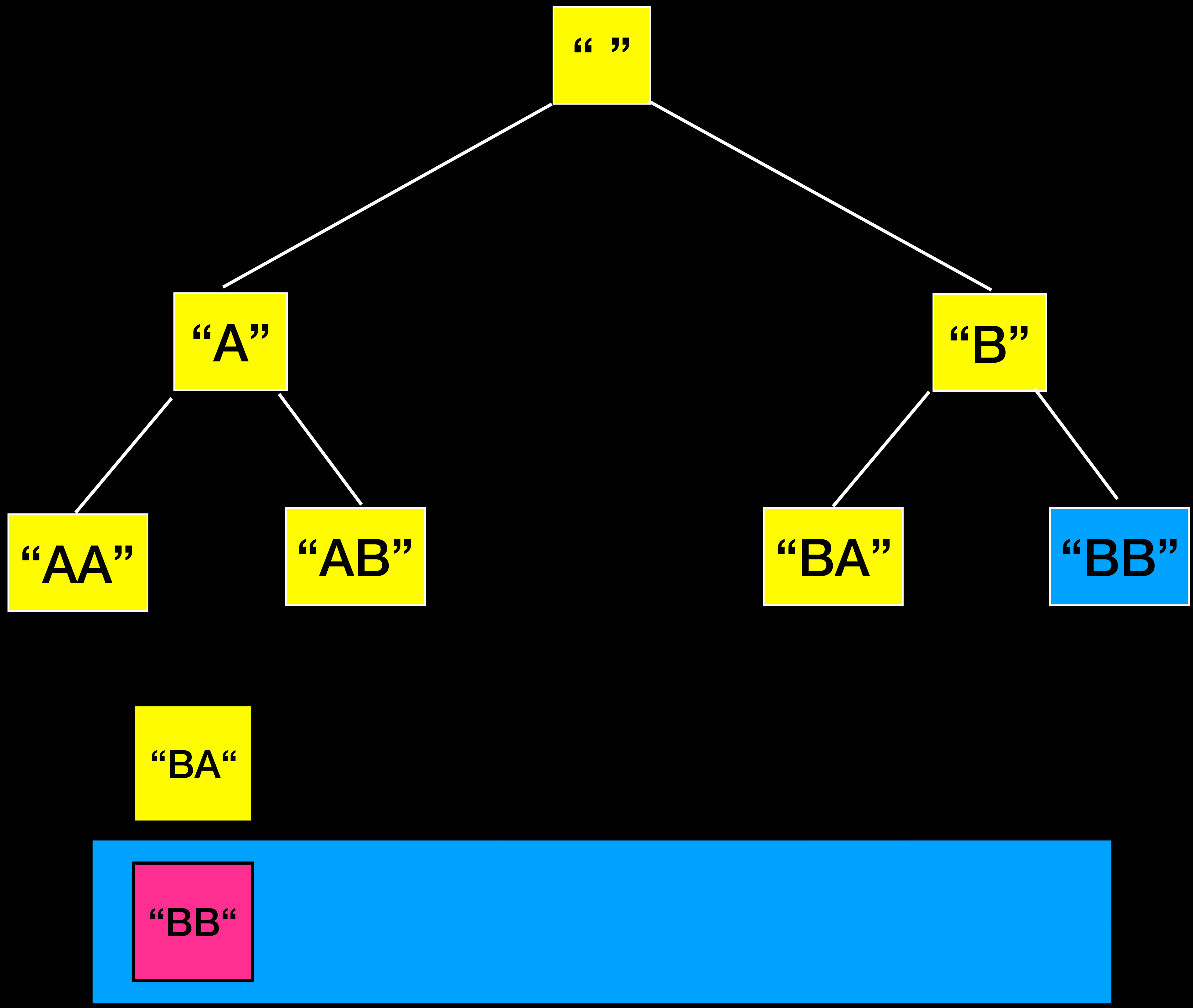

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AA“
“AB“
“BA“
“BB“

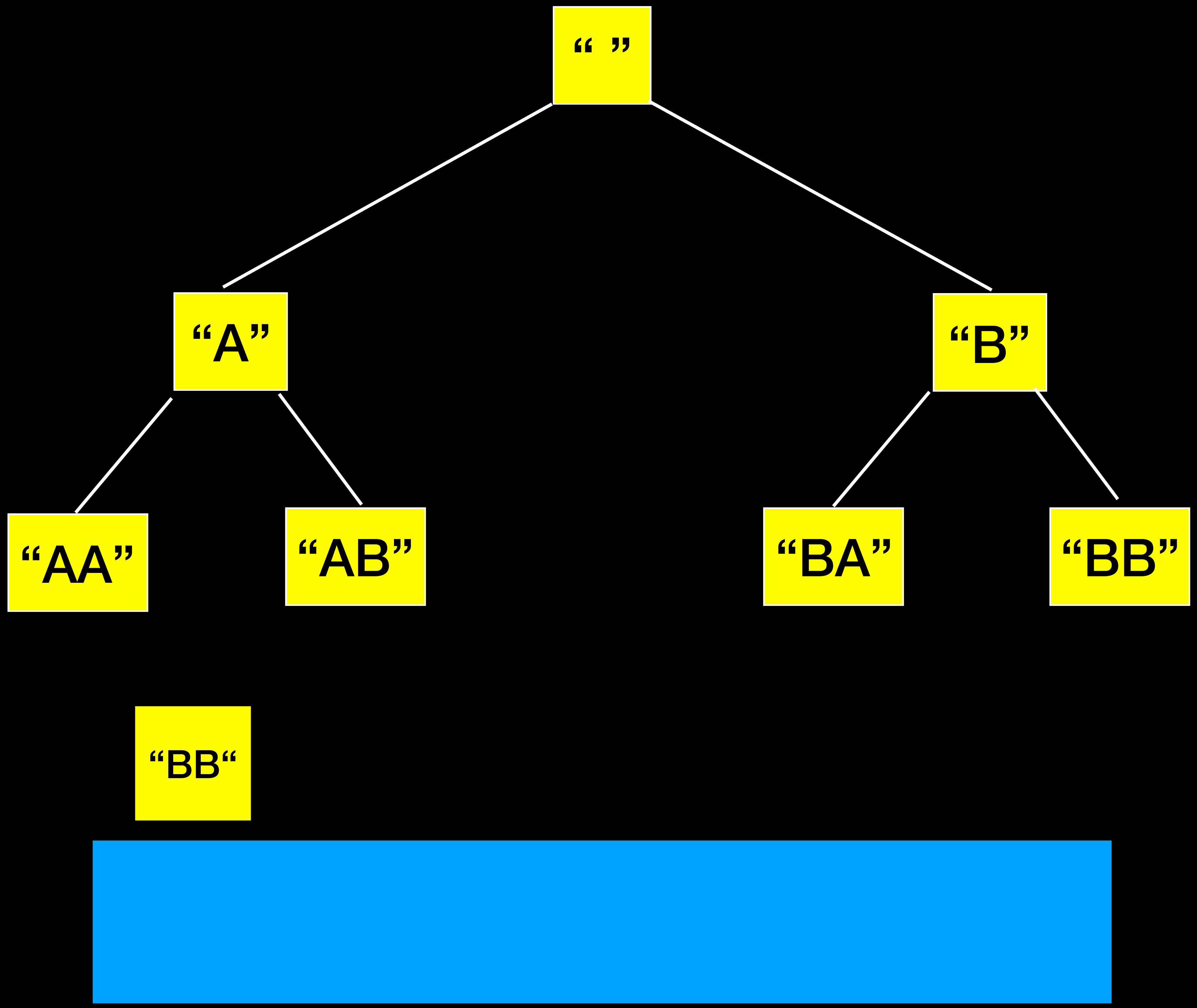

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AA“
“AB“
“BA“
“BB“

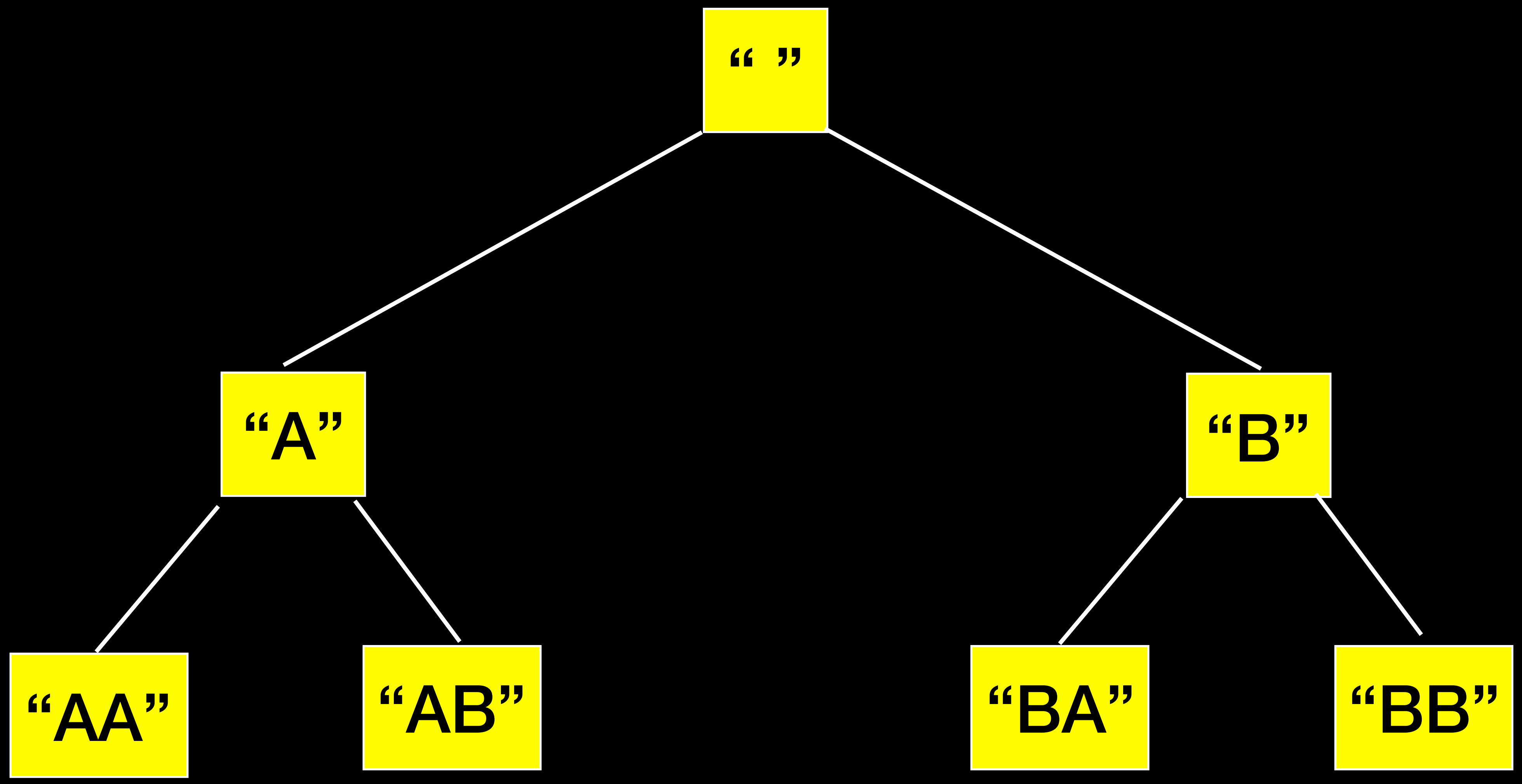

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AA“
“AB“
“BA“
“BB“


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AB“
“BA“
“BB“


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AB“
“BA“
“BB“


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“BA“
“BB“


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“BB“


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

Breadth-First Search

Applications

Find shortest path in graph

GPS navigation systems

Crawlers in search engines

. . .

Generally looks for the “shortest” or “best” way to do something => lists things in increasing order of “size” stopping at the “shortest” solution

```
findAllStrings(int n)
{
    put empty string on the queue

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result;
}
```

Analysis

Finding permutations of all strings of size **up to n**

Assume alphabet of size 26

The empty string = 1

All strings of size 1 = 26^1

All strings of size 2 = 26^2

. . .

All strings of size n = 26^n

With repetition: I have `26` options for each of the `n` characters

Exam Drill:

Analyze the worst-case time complexity of this algorithm

T(n) = ?

O(?)

```
findAllStrings(int n)
{
    put empty string on the queue

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result;
}
```

Removes 1 string from the queue

Adds 26 strings to the queue

```
findAllStrings(int n)
{
    put empty string on the queue

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result
}
```

Will stop when all strings have been removed from queue

```
findAllStrings(int n)
{
    put empty string on the queue

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result
}
```

Removes 1 string from the queue

Adds 26 strings to the queue

Will stop when all strings have been removed from queue

$$T(n) = 26^0 + 26^1 + 26^2 + \ldots 26^n$$

```
findAllStrings(int n)
{
    put empty string on the queue

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result
}
```

Removes 1 string from the queue

Adds 26 strings to the queue

Will stop when all strings have been removed from queue

$$T(n) = 26^0 + 26^1 + 26^2 + \ldots \boxed{26^n}$$

```
findAllStrings(int n)
{
    put empty string on the queue;

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result
}
```

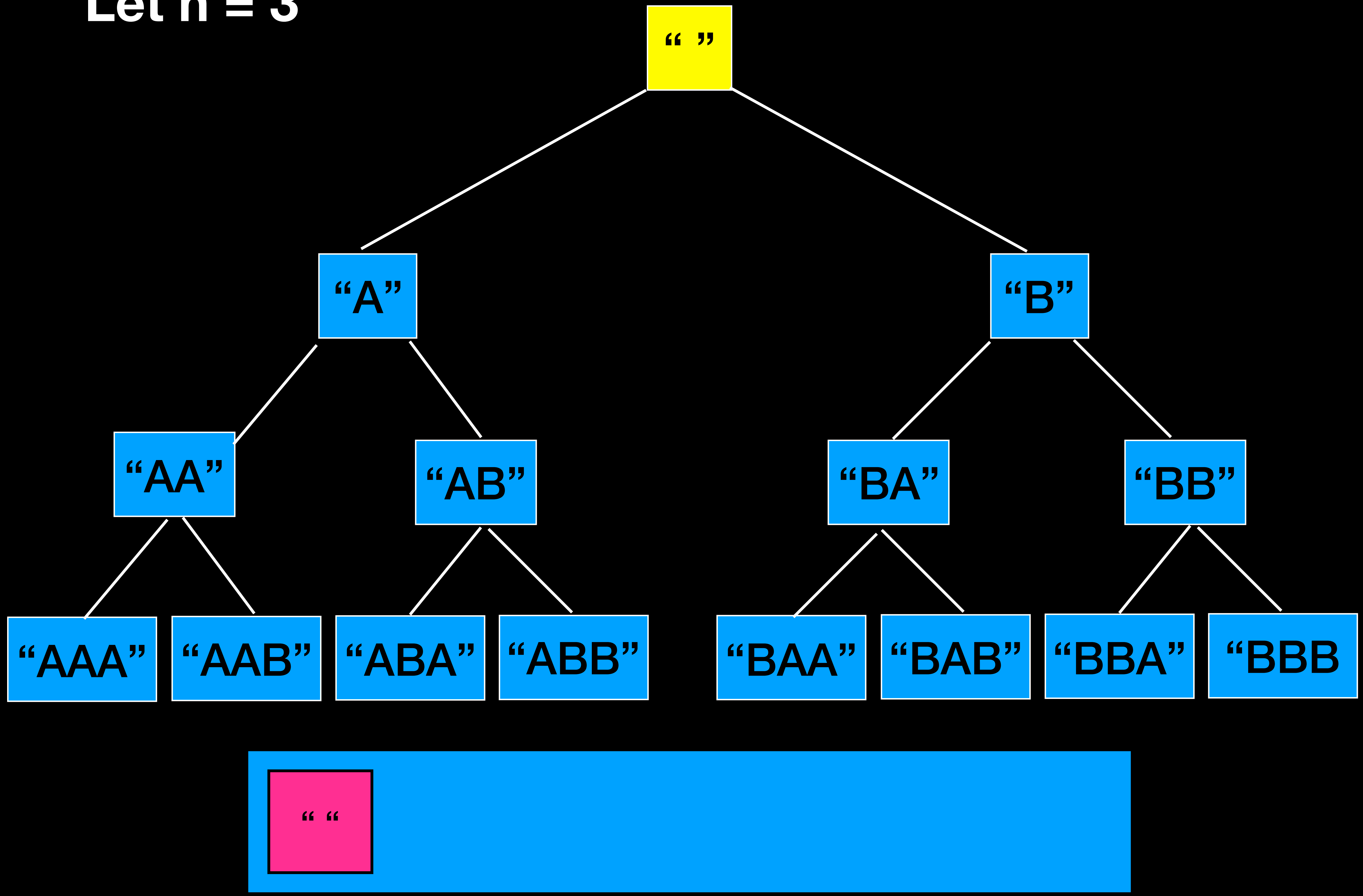
Removes 1 string from the queue

Adds k strings to the queue

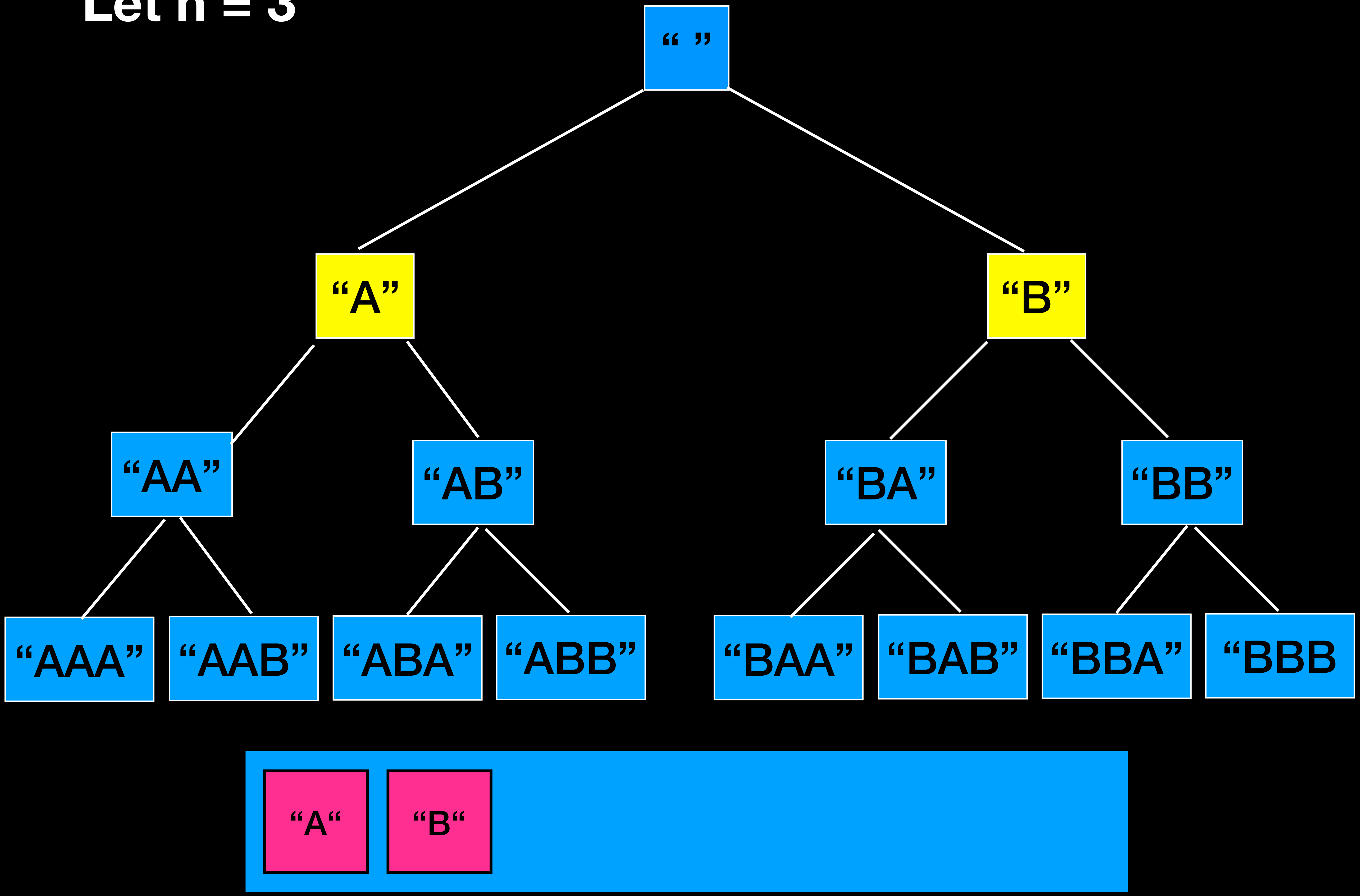
Will stop when all strings have been removed from queue

$O(26^n)$

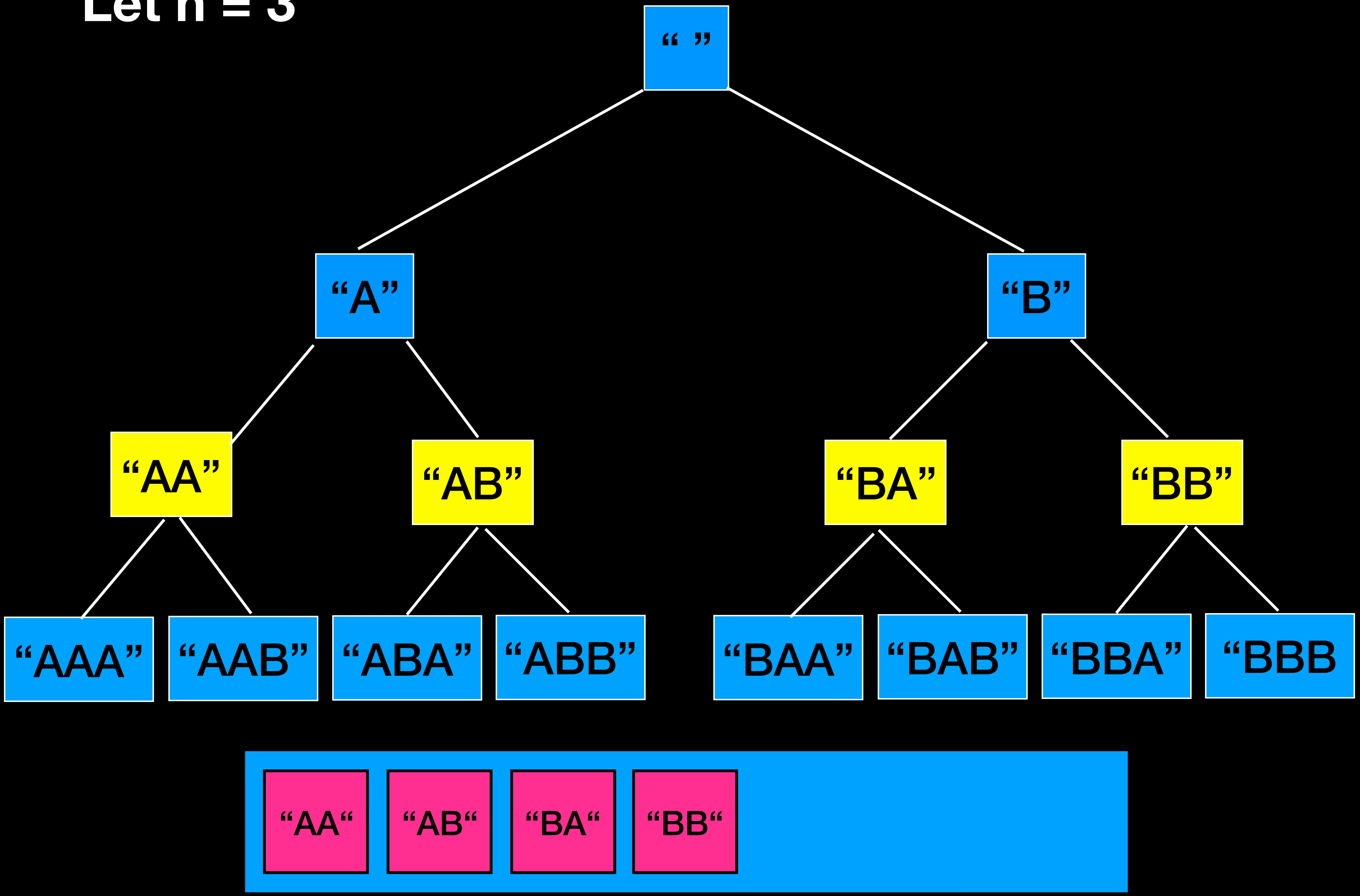
Let n = 3



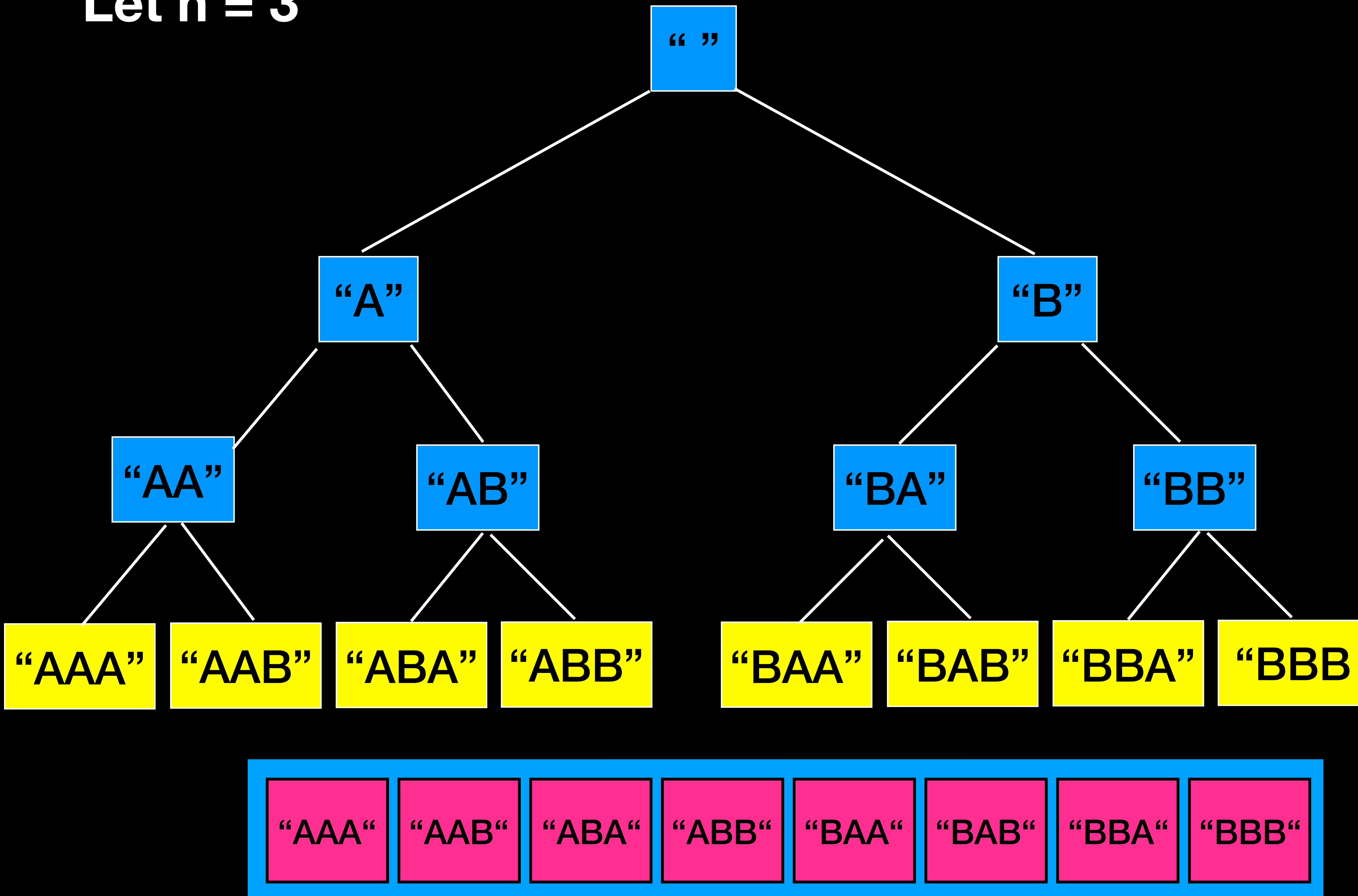
Let n = 3



Let n = 3



Let n = 3


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AAA”
“AAB”
“ABA”
“ABB”
“BAA”
“BAB”
“BBA”
“BBB
“AAA“
“AAB“
“ABA“
“ABB“
“BAA“
“BAB“
“BBA“
“BBB“

Memory Usage

At some point we end up with **26^n** strings in memory

Size of string on my machine = 24 bytes

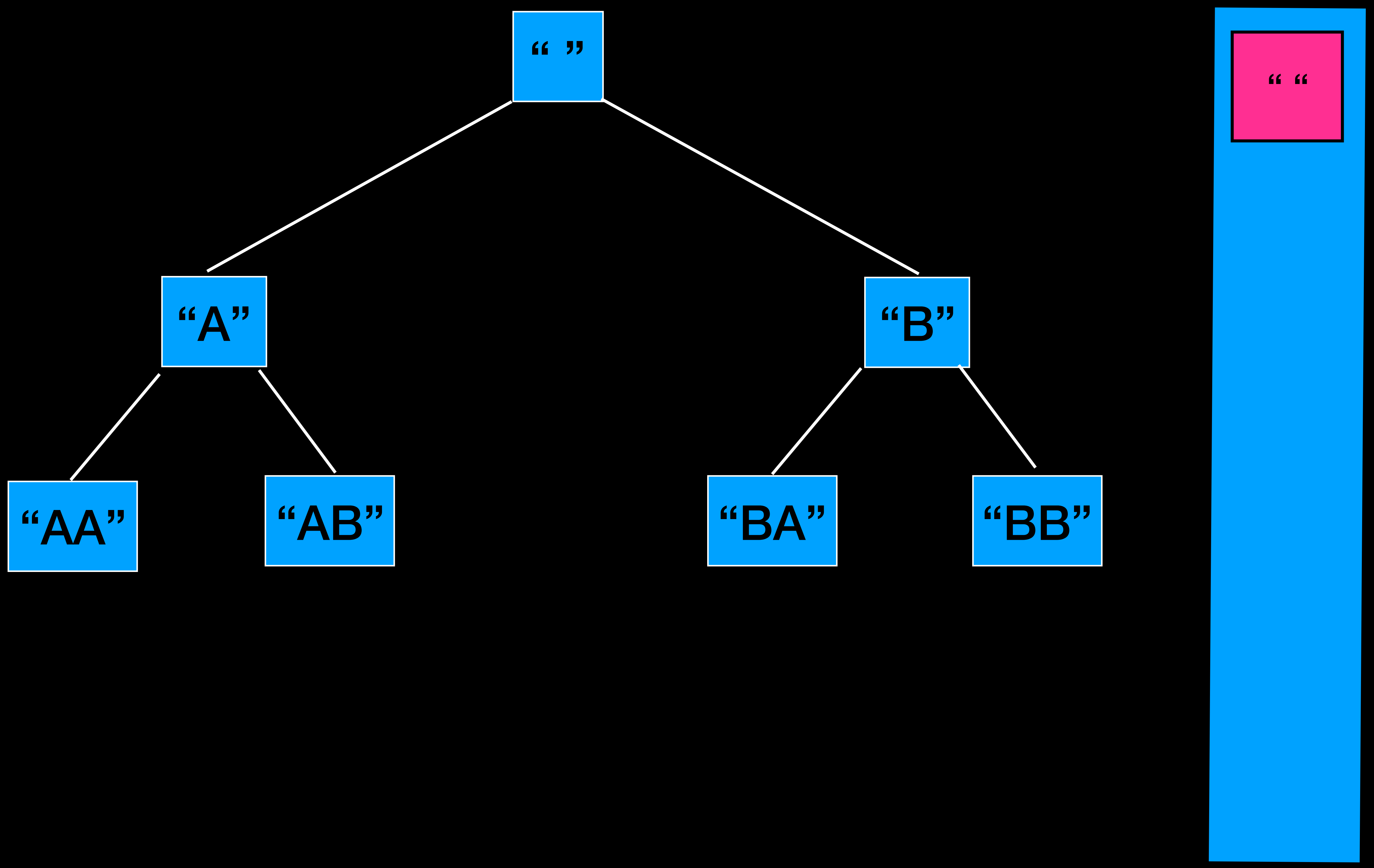
Running this algorithm for n = 7 (≈ 193GB) is the maximum that can be handled by a standard personal computer

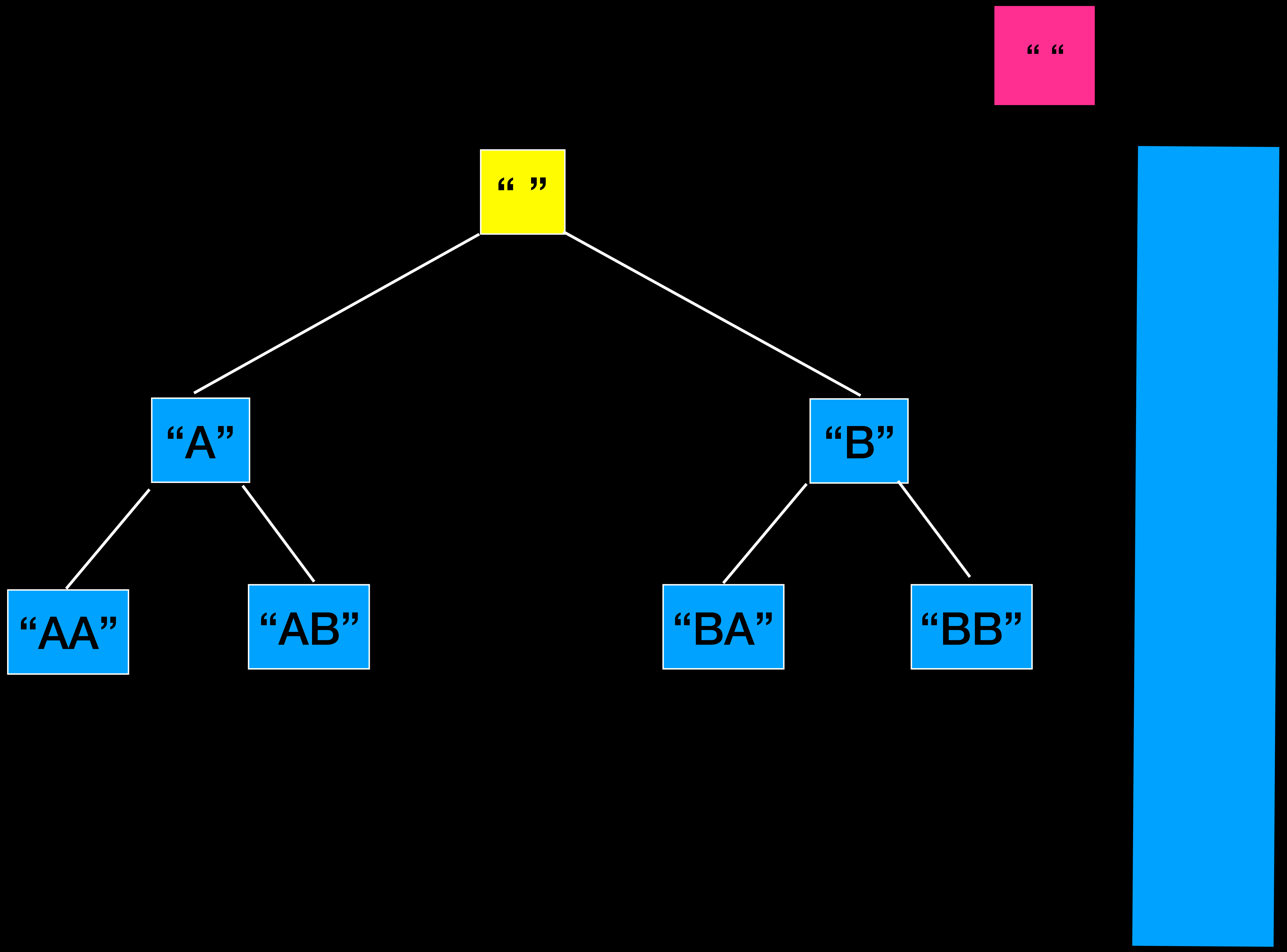
For n = 8 ≈ 5TB

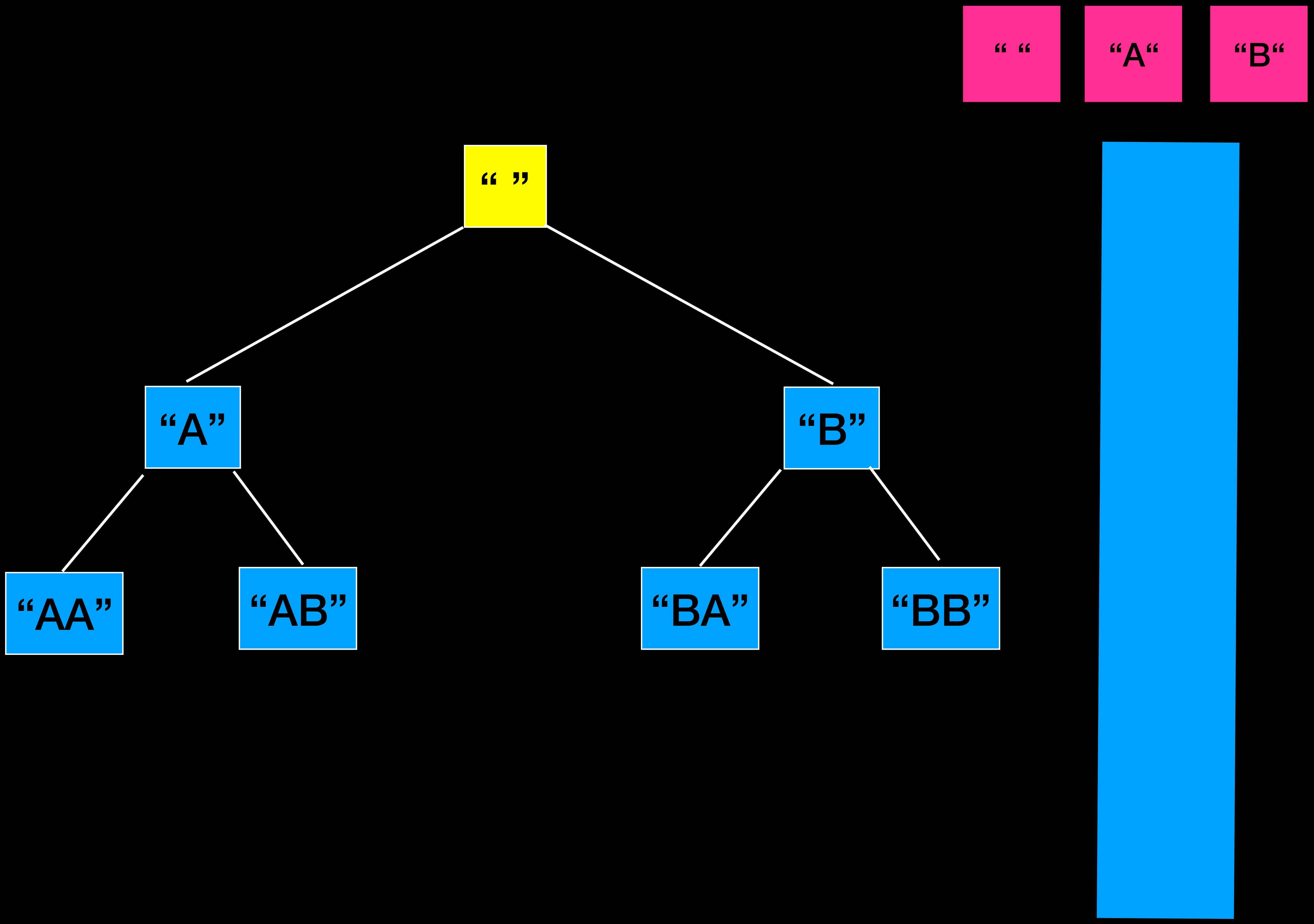
What if we use a stack?

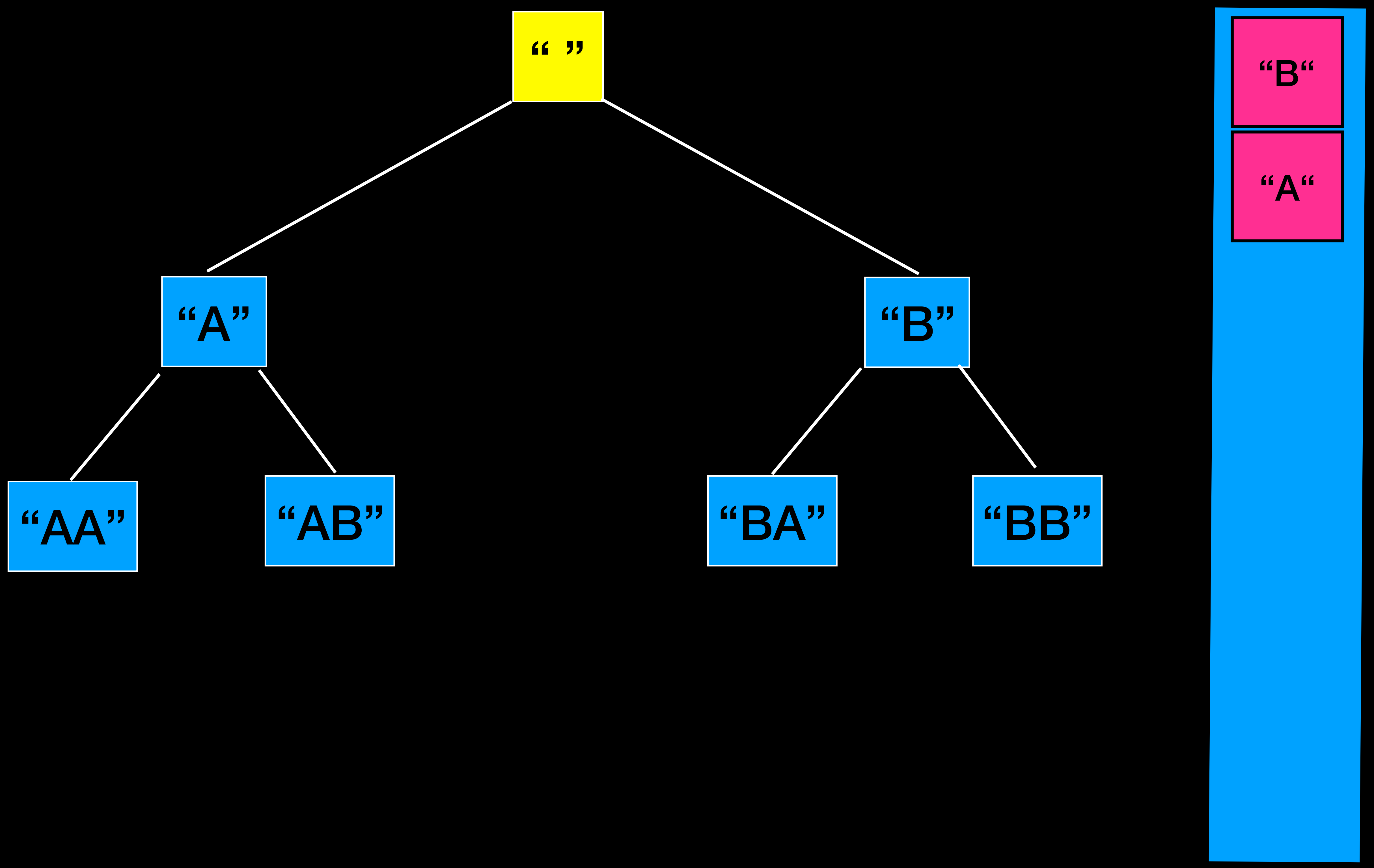
```
findAllStrings(int n)
{
    put empty string on the stack

    while(stack is not empty){
        let current_string = top of stack and add to result
        if(current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to stack
        }
    }
    return result
}
```

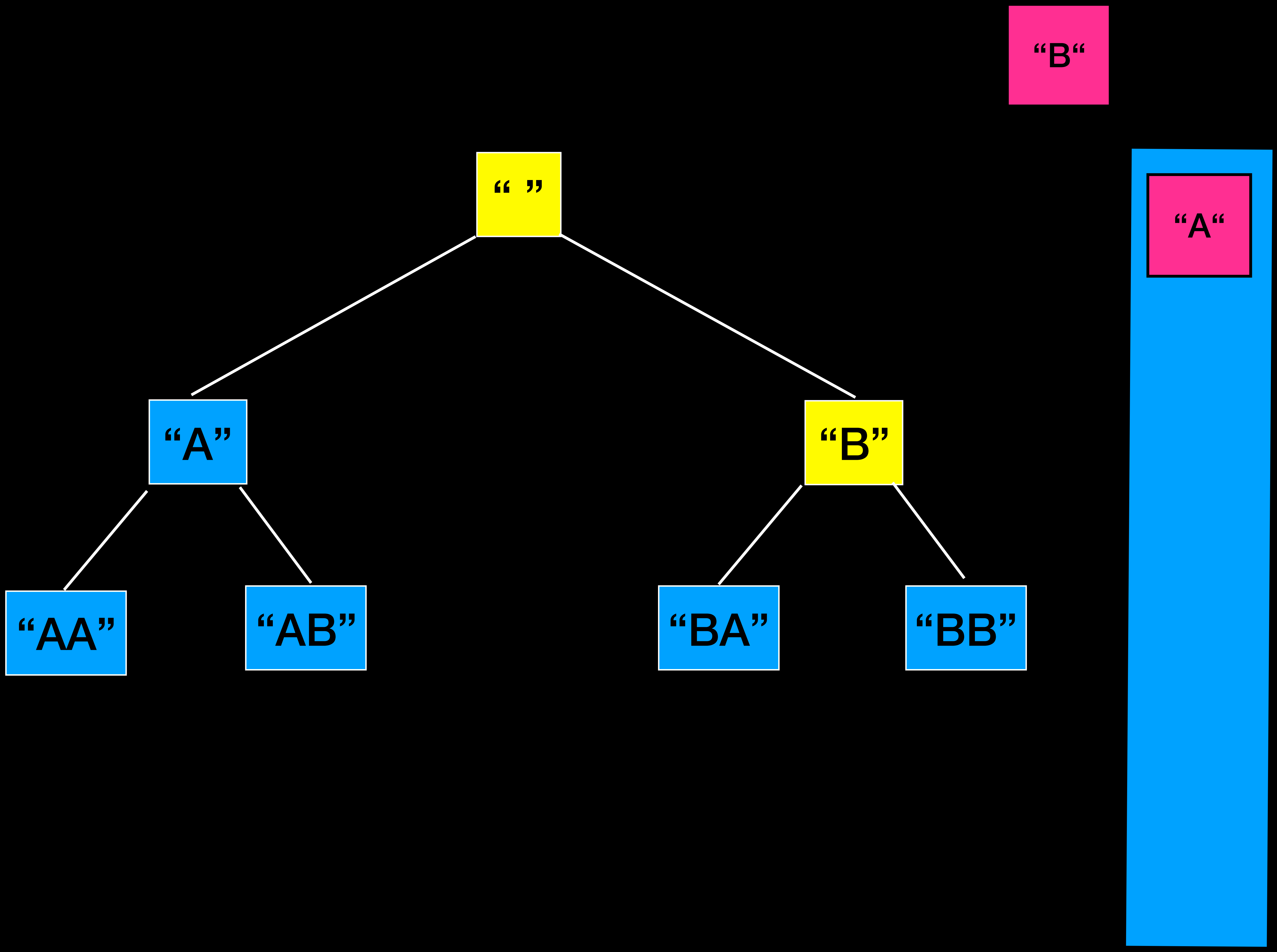
$O(26^n)$

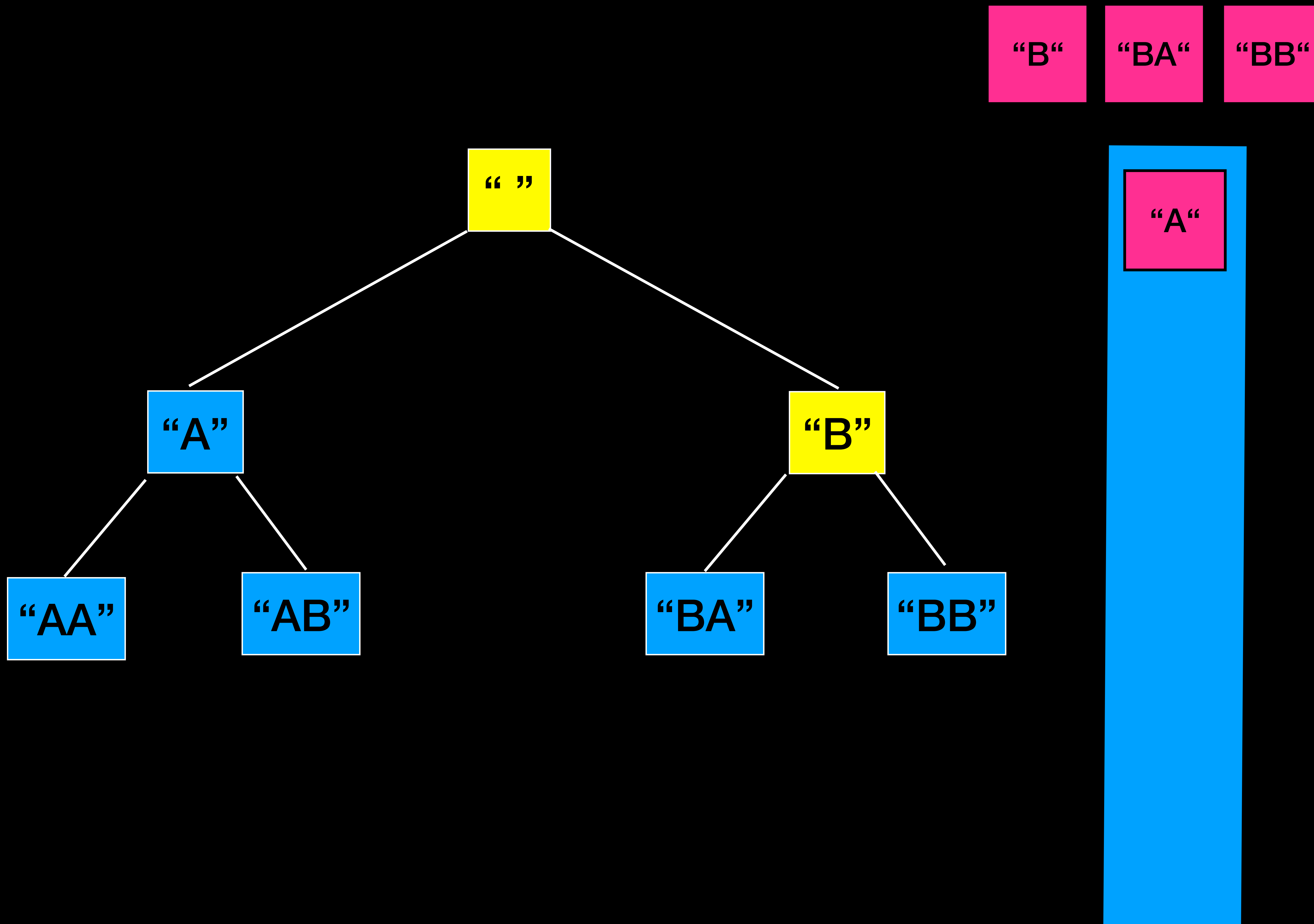

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“ “


“ “
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

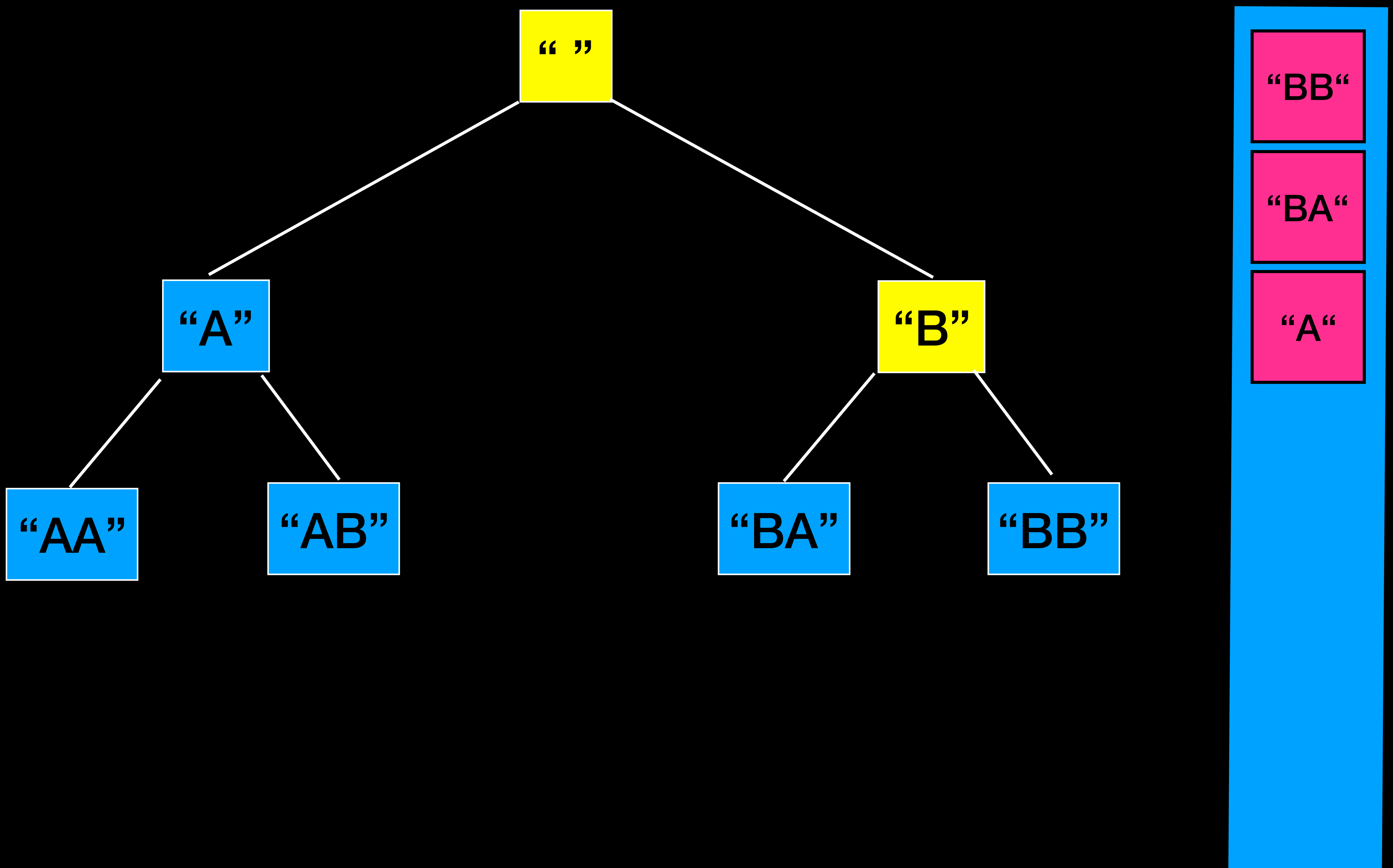

“ “
“A“
“B“
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

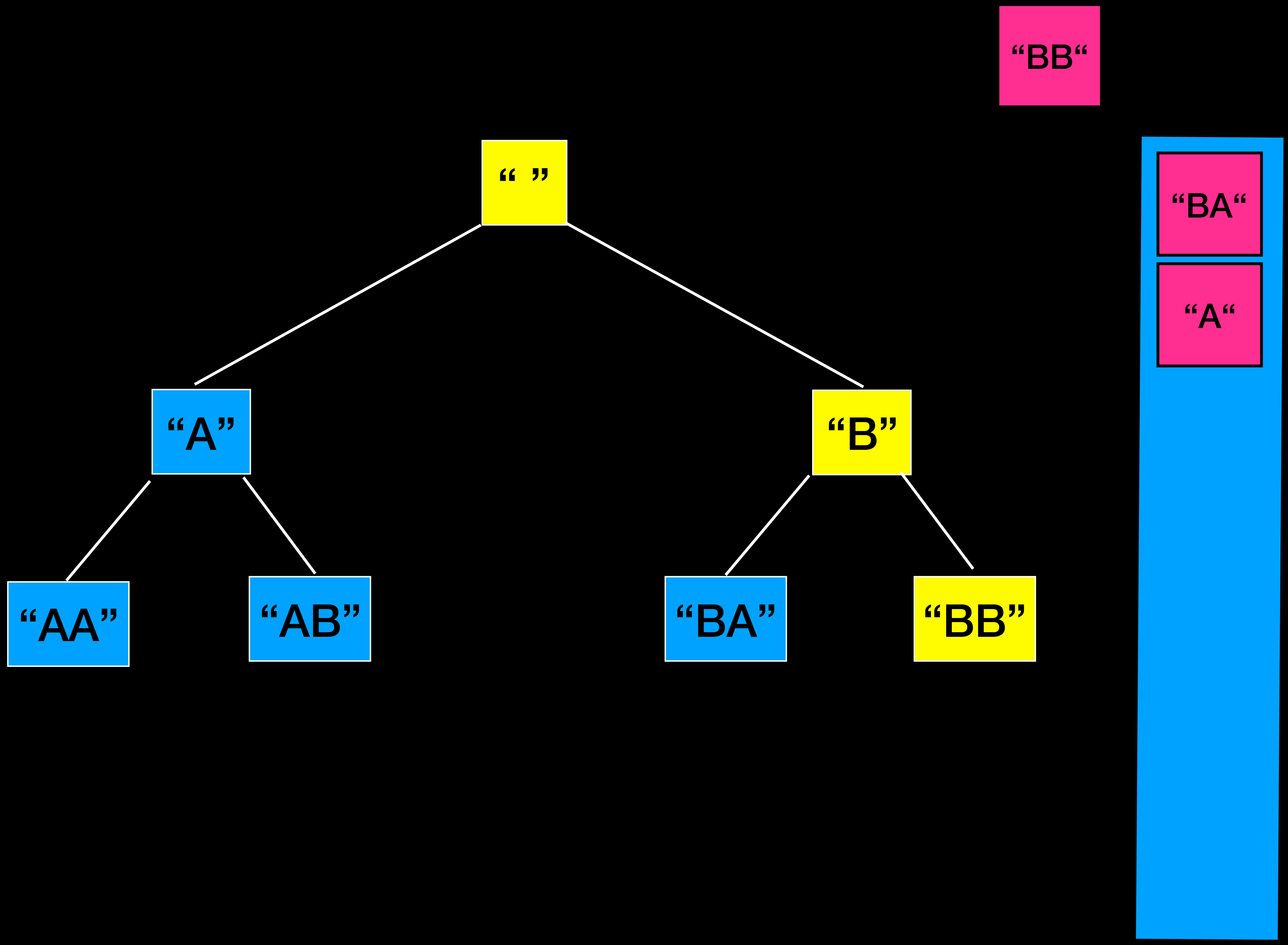

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“B“
“A“


“B“
“ ”
“A“
“A”
“B”
“AA”
“AB”
“BA”
“BB”

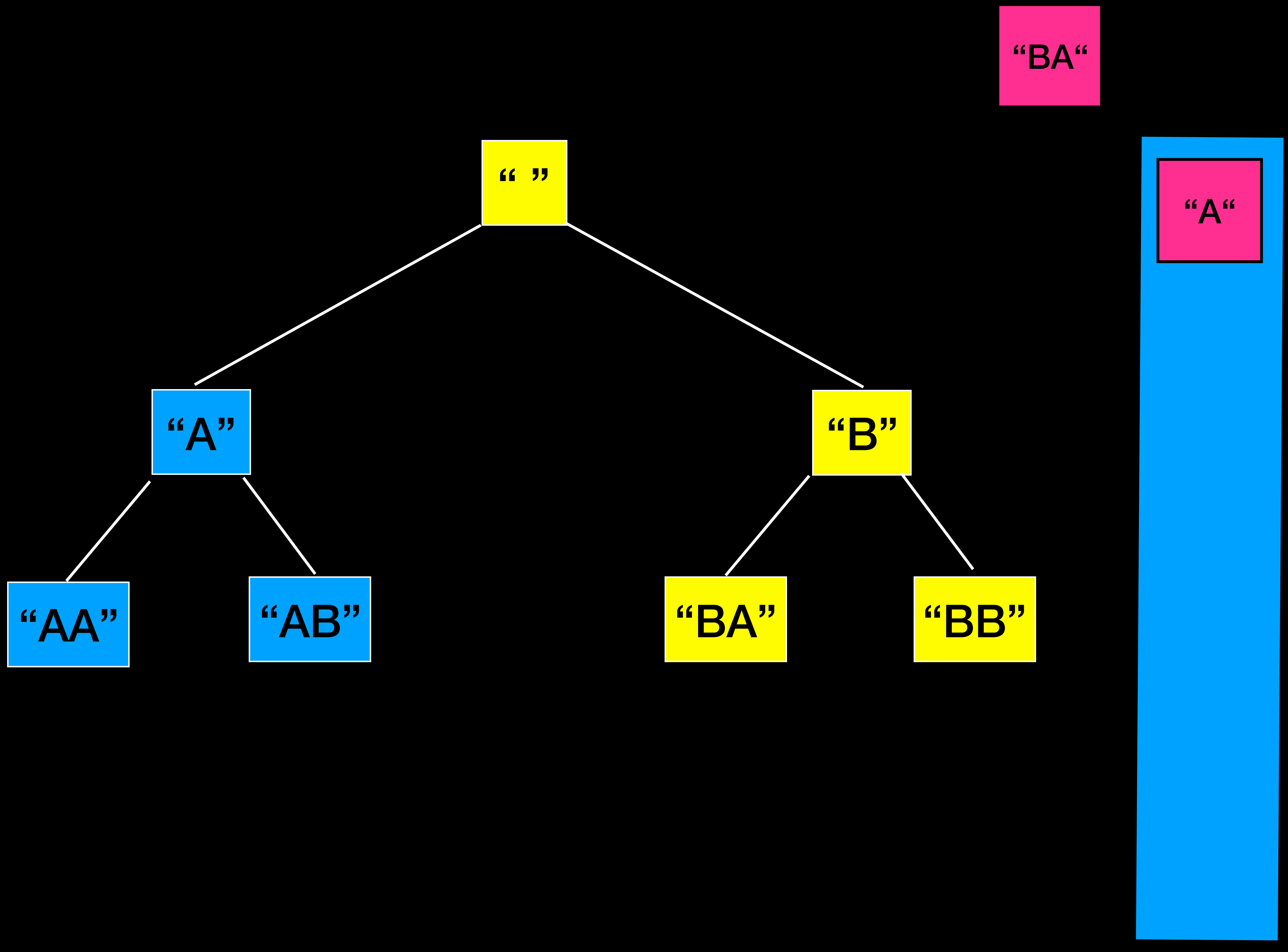


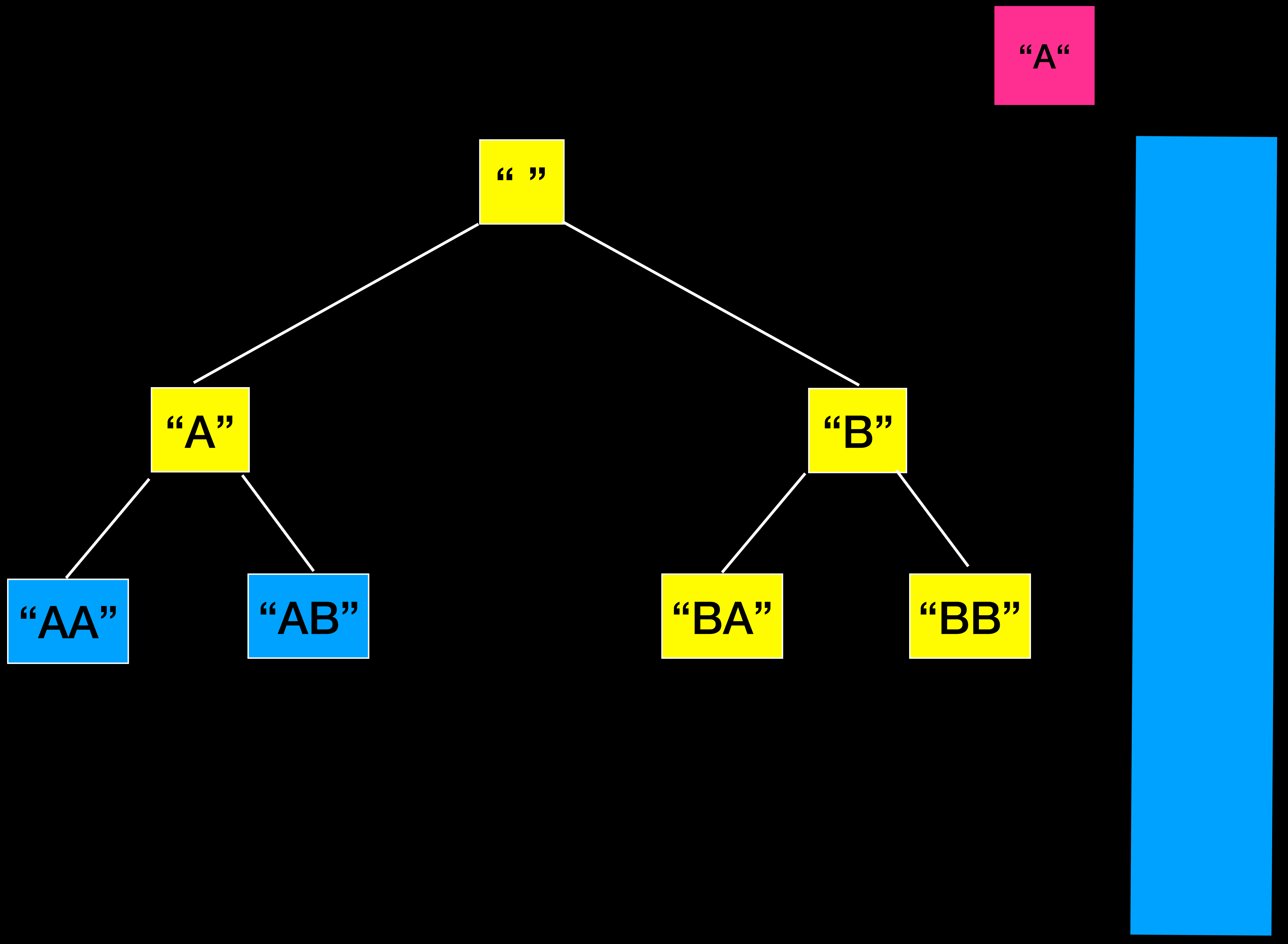
“B“
“BA“
“BB“
“ ”
“A“
“A”
“B”
“AA”
“AB”
“BA”
“BB”


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“BB“
“BA“
“A“

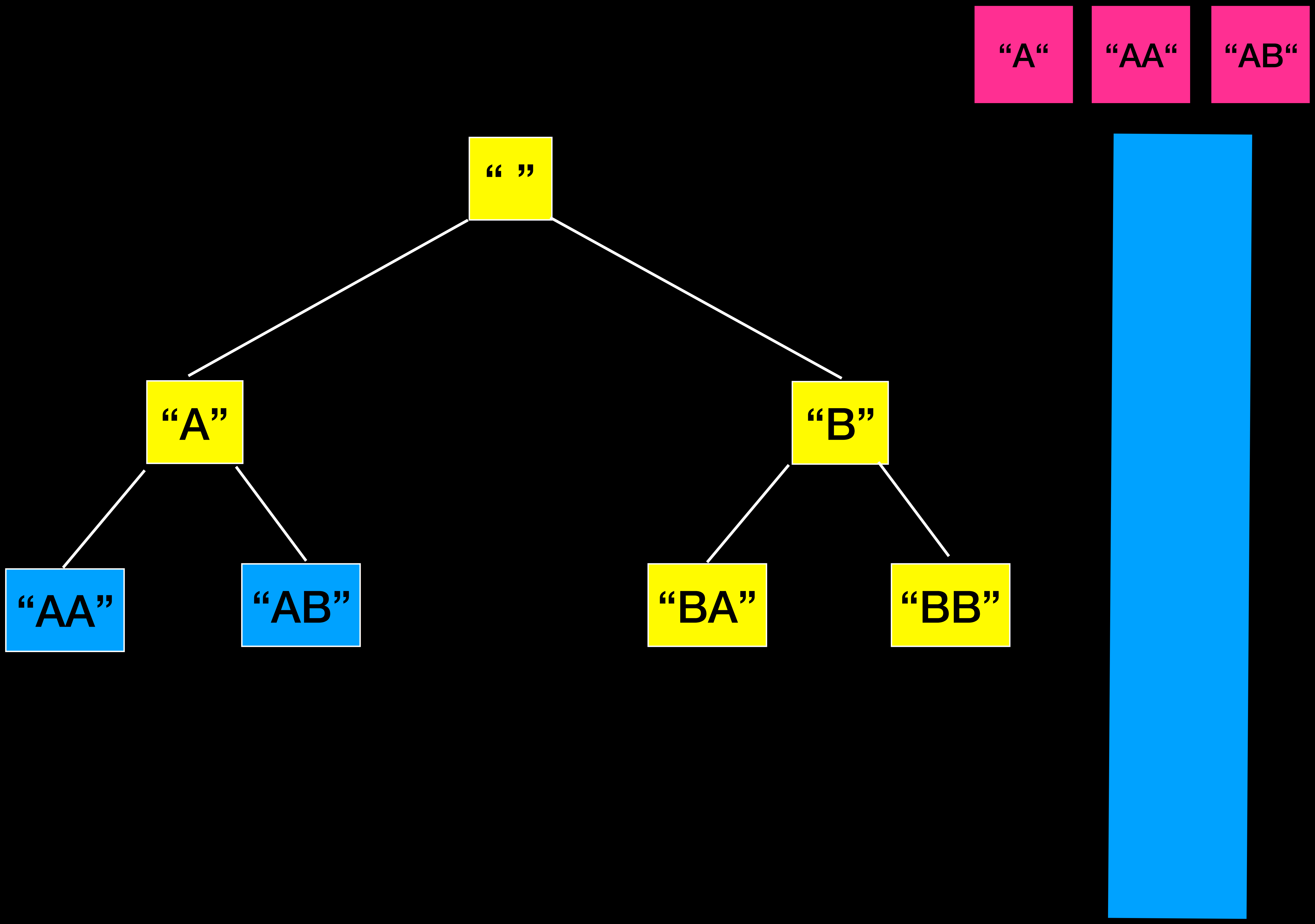


"BB"
" "
"A"
"B"
"AA"
"AB"
"BA"
"BB"
"BA"
"A"

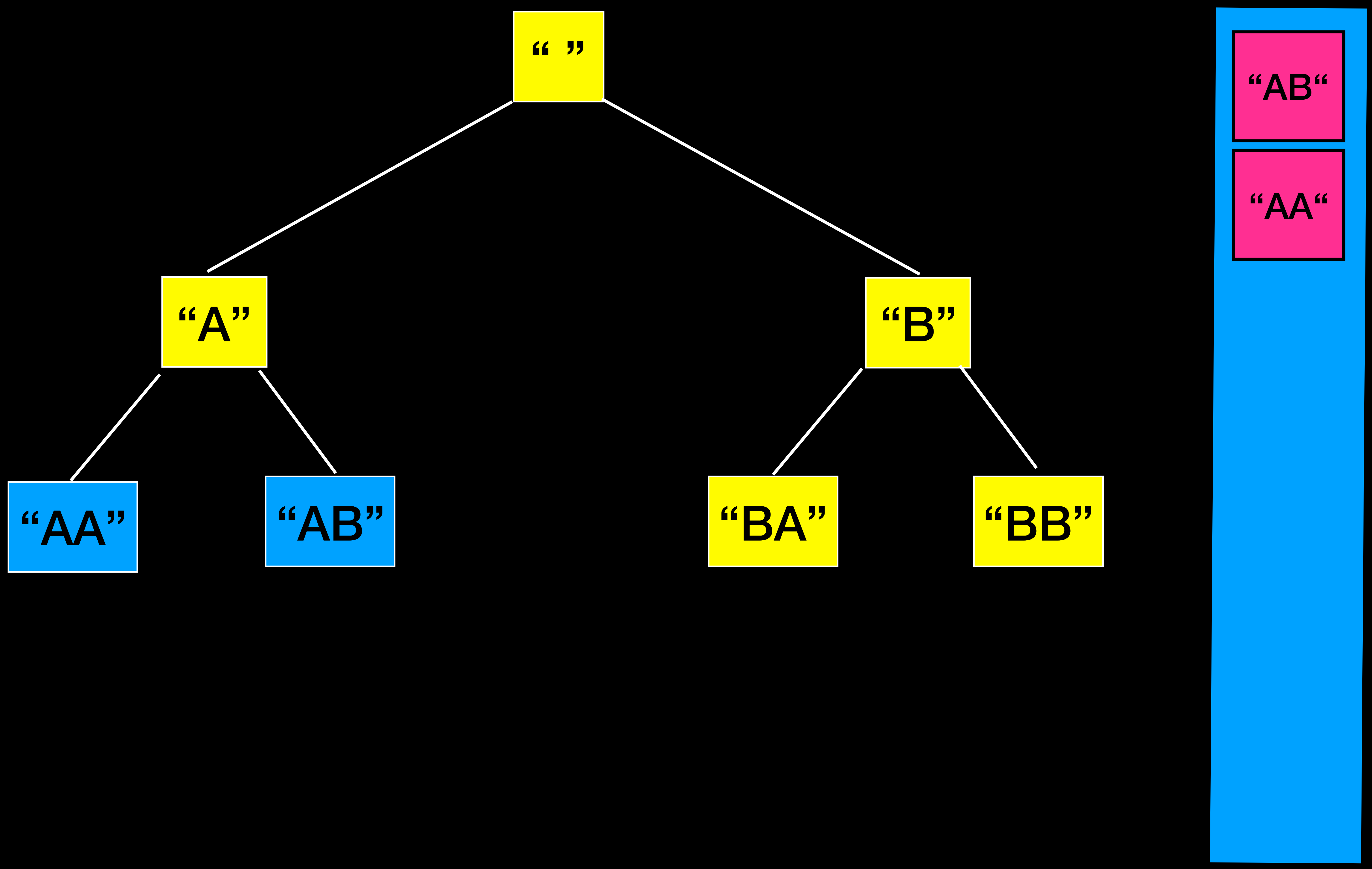

“BA“
“ ”
“A“
“A”
“B”
“AA”
“AB”
“BA”
“BB”

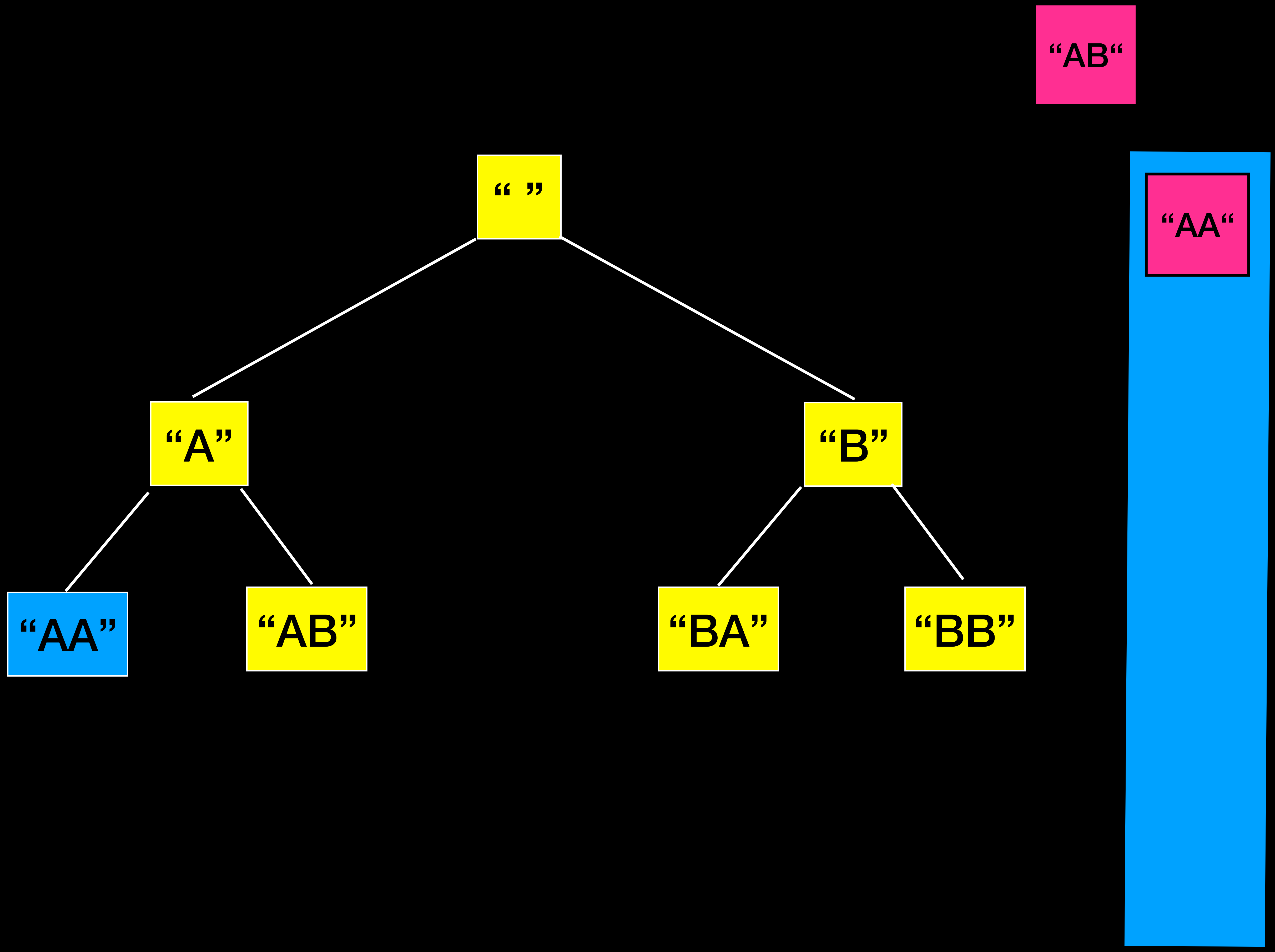


“A“
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

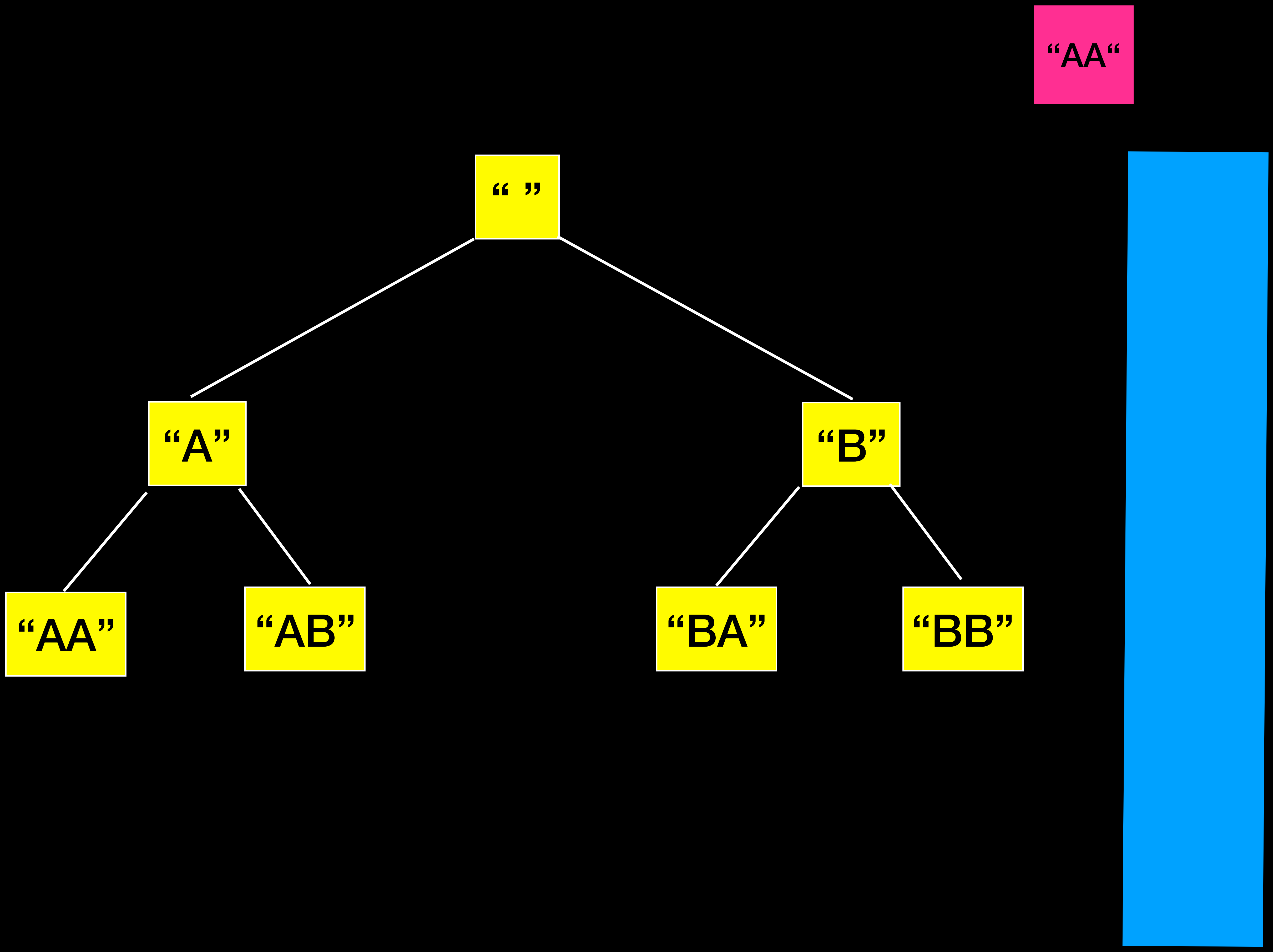


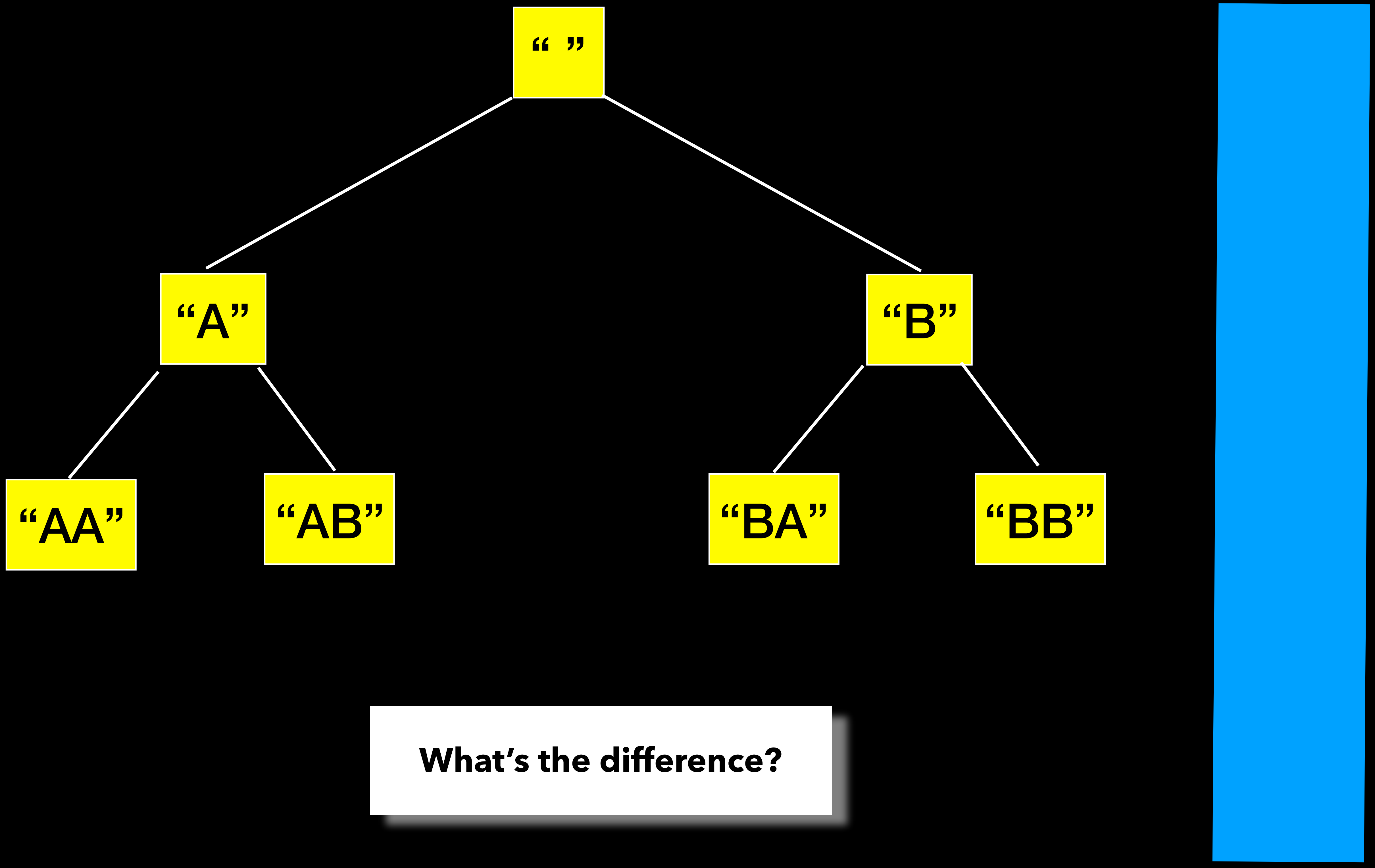
“A“
“AA“
“AB“
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“AB“
“AA“



“AB“
“ ”
“AA“
“A”
“B”
“AA”
“AB”
“BA”
“BB”


“AA“
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”



What’s the difference?

Depth-First Search

Applications

- Detecting cycles in graphs
- Topological Sorting
- Path finding
- Finding strongly connected components in graph
- . . .

More space efficient than previous approach

Does not explore options in increasing order of size

Comparison

Breadth-First Search
(using a queue)

Time $O(26^n)$

Space $O(26^n)$

Good for exploring options in increasing order of size when expecting to find "shallow" solution

Memory inefficient when must keep each "level" in memory

Depth-First Search
(using a stack)

Time $O(26^n)$

Space $O(n)$

Explores each option individually to max size - does NOT list options by increasing size

More memory efficient

Recognizing Palindromes

Palindrome: a string that reads the same in reverse order

Anna

Civic

Kayak

Noon

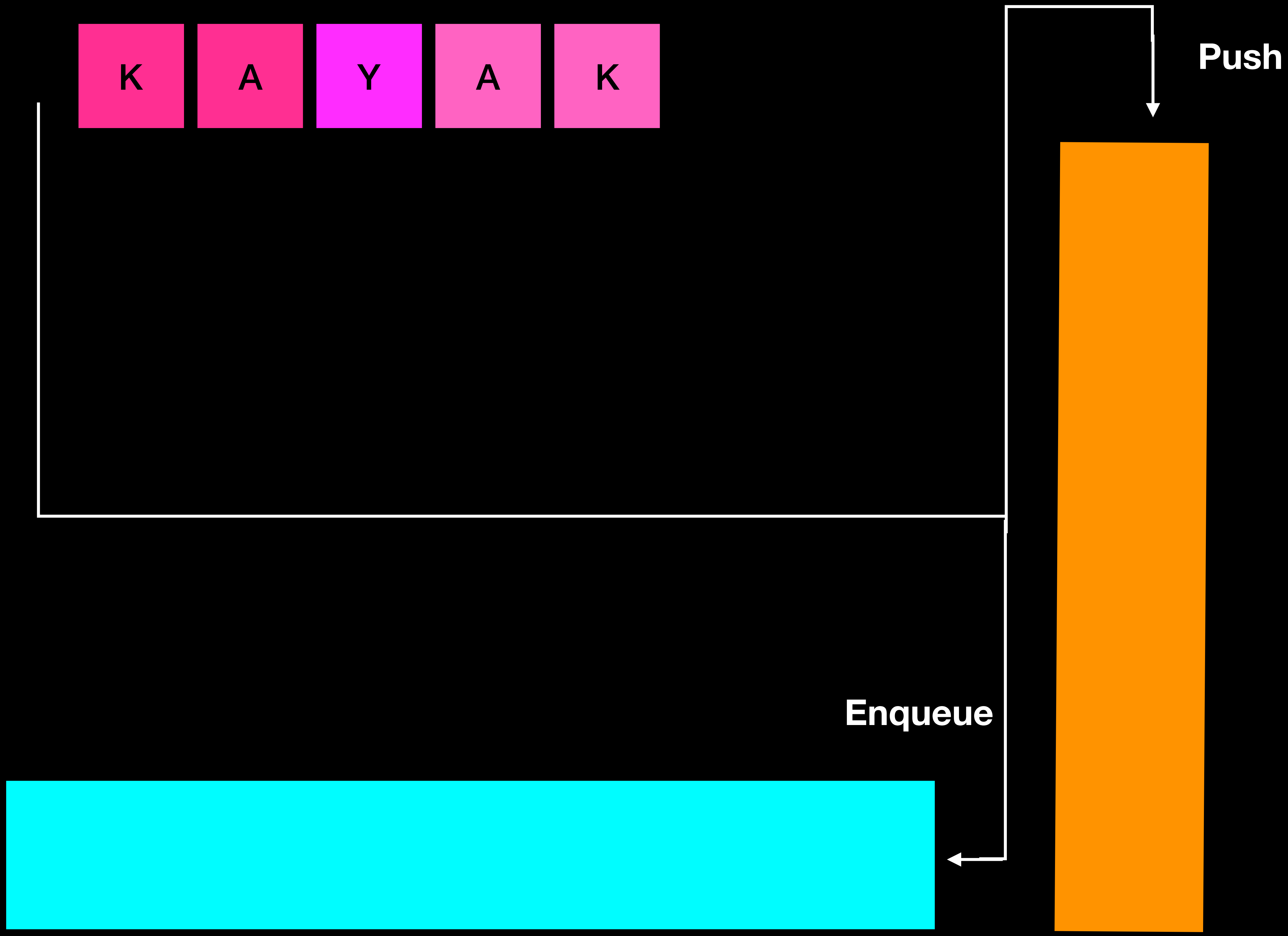
Radar

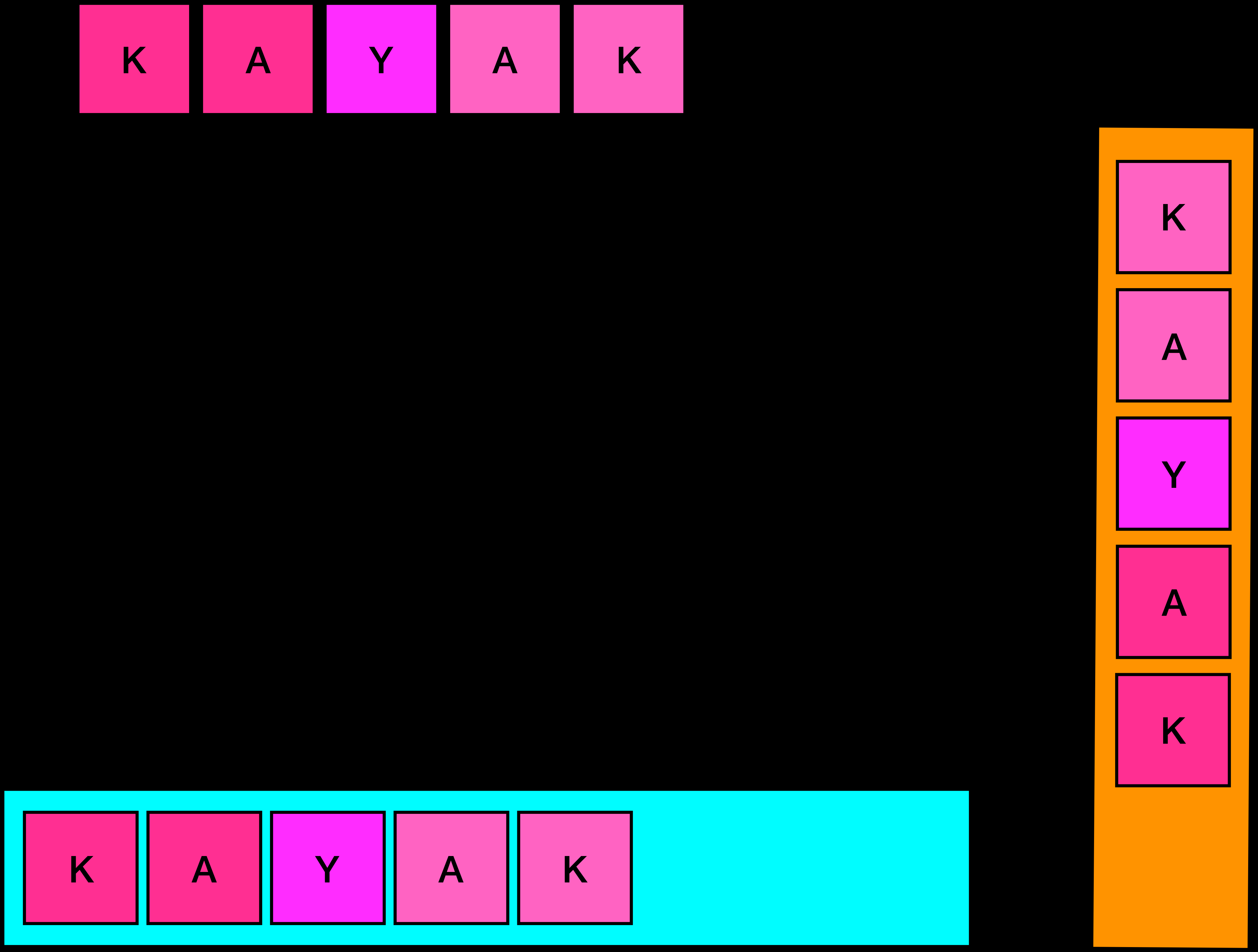
Notice

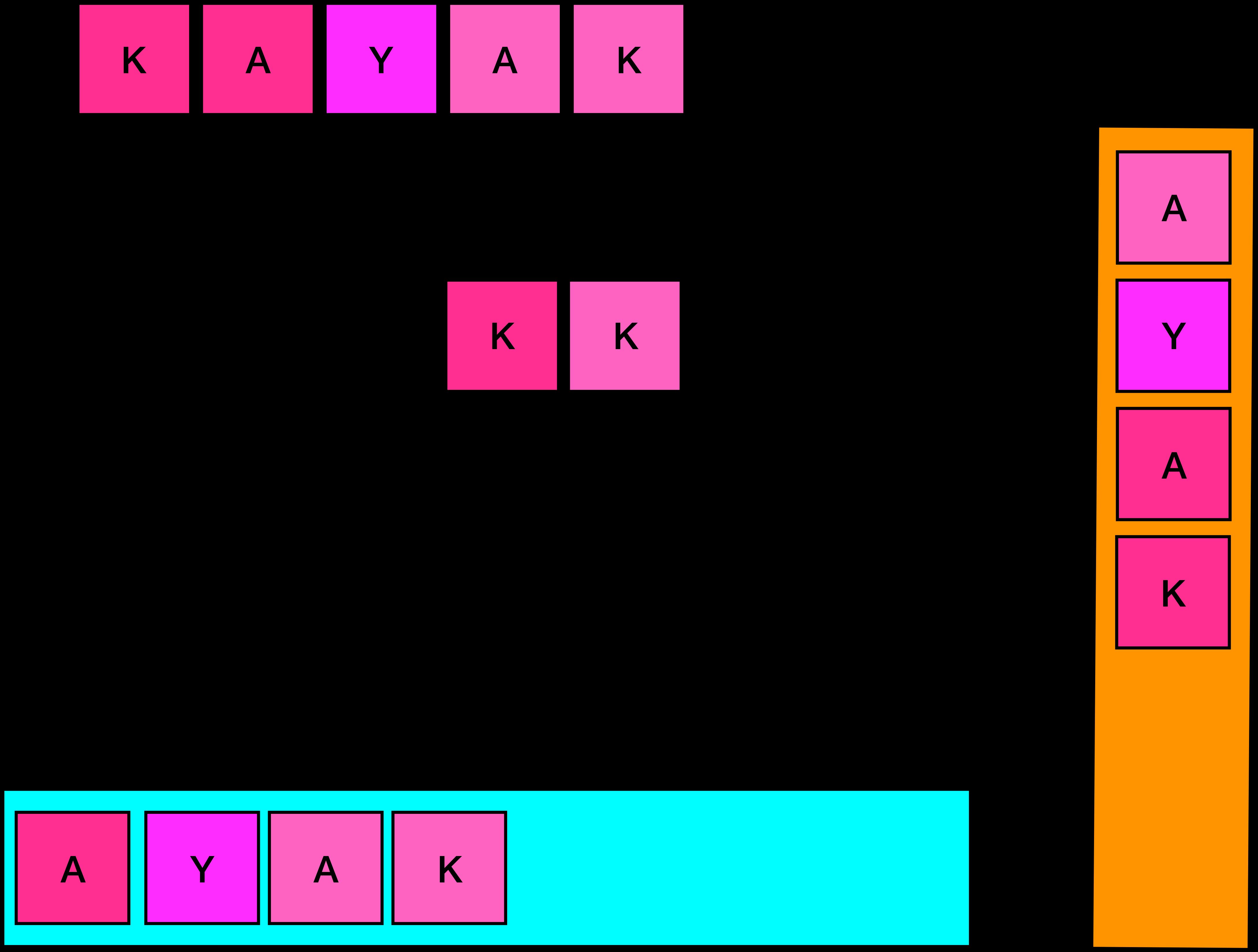
A stack can be used to reverse a string (LIFO)

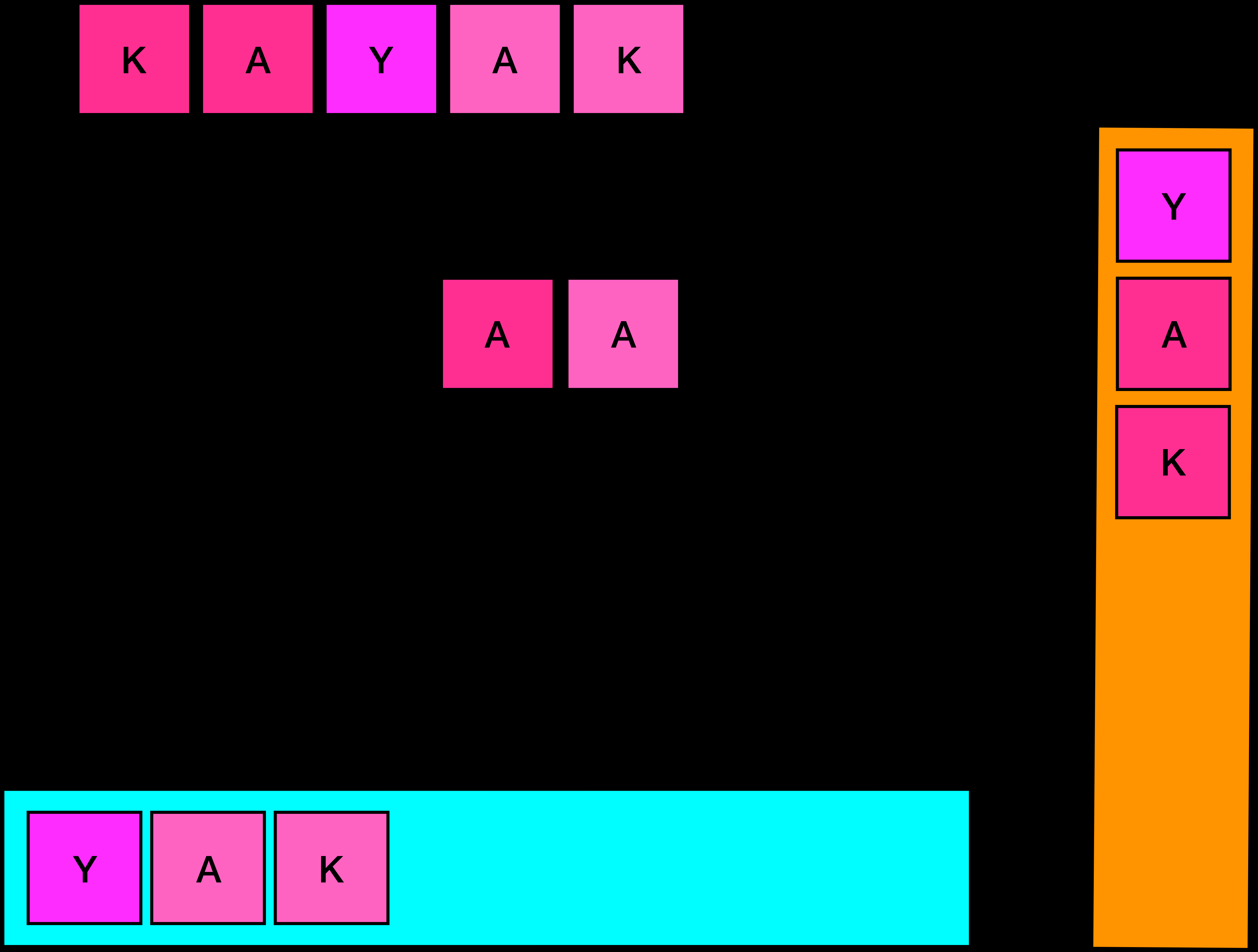
A queue can be used to preserve the original order of a string (FIFO)

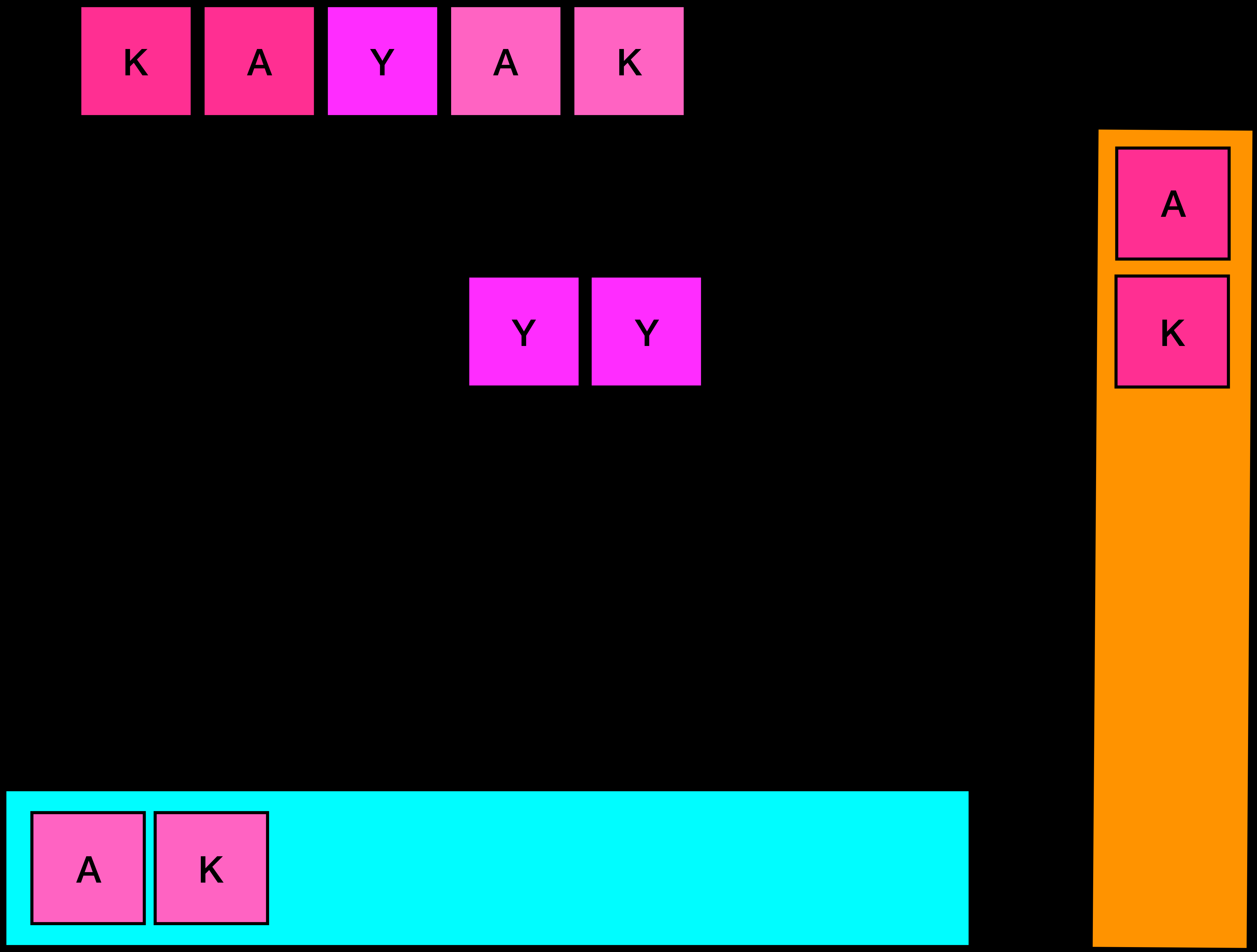
Algorithm: add string characters to both stack and queue and then compare to check if they are the same

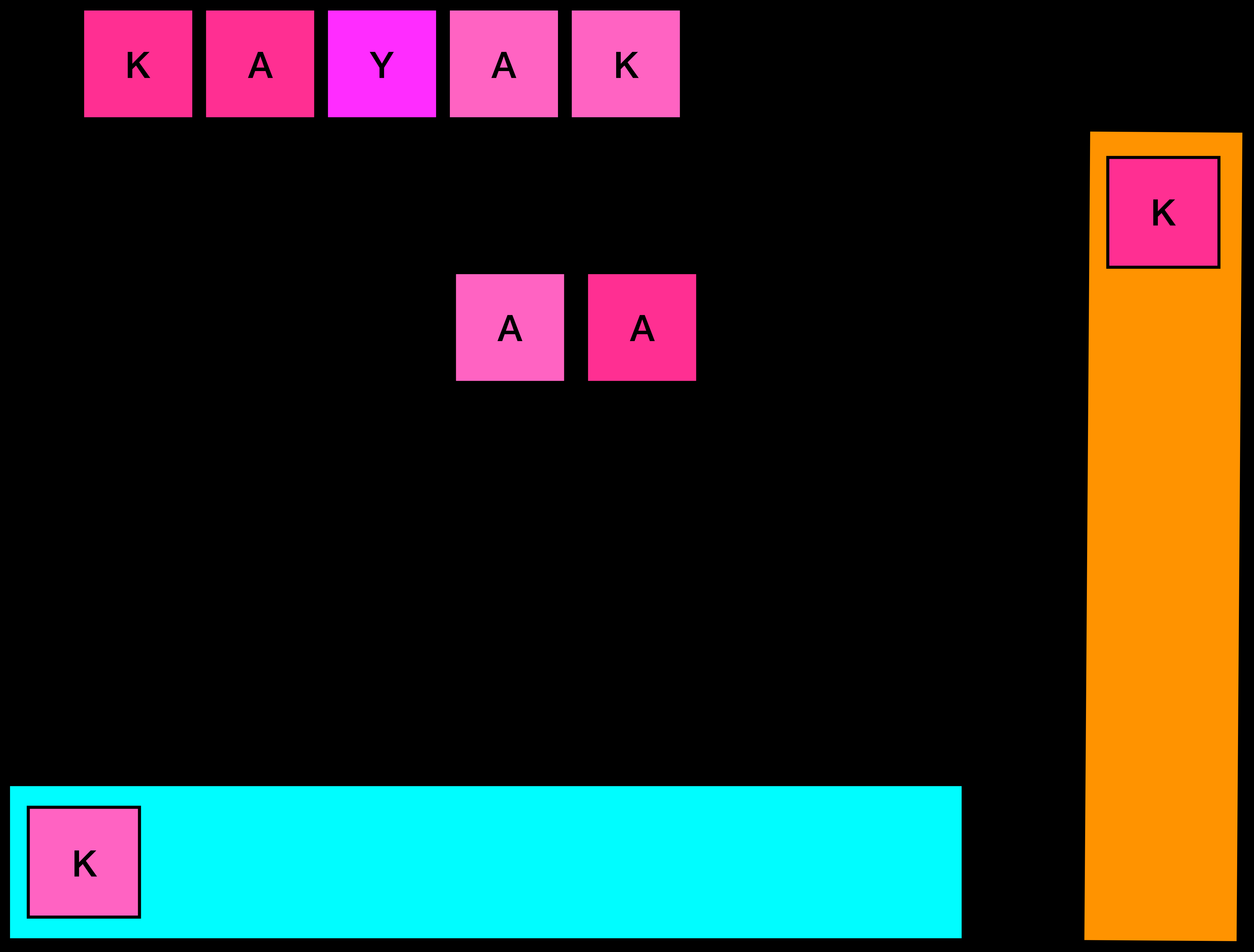

K
A
Y
A
K
Push
Enqueue

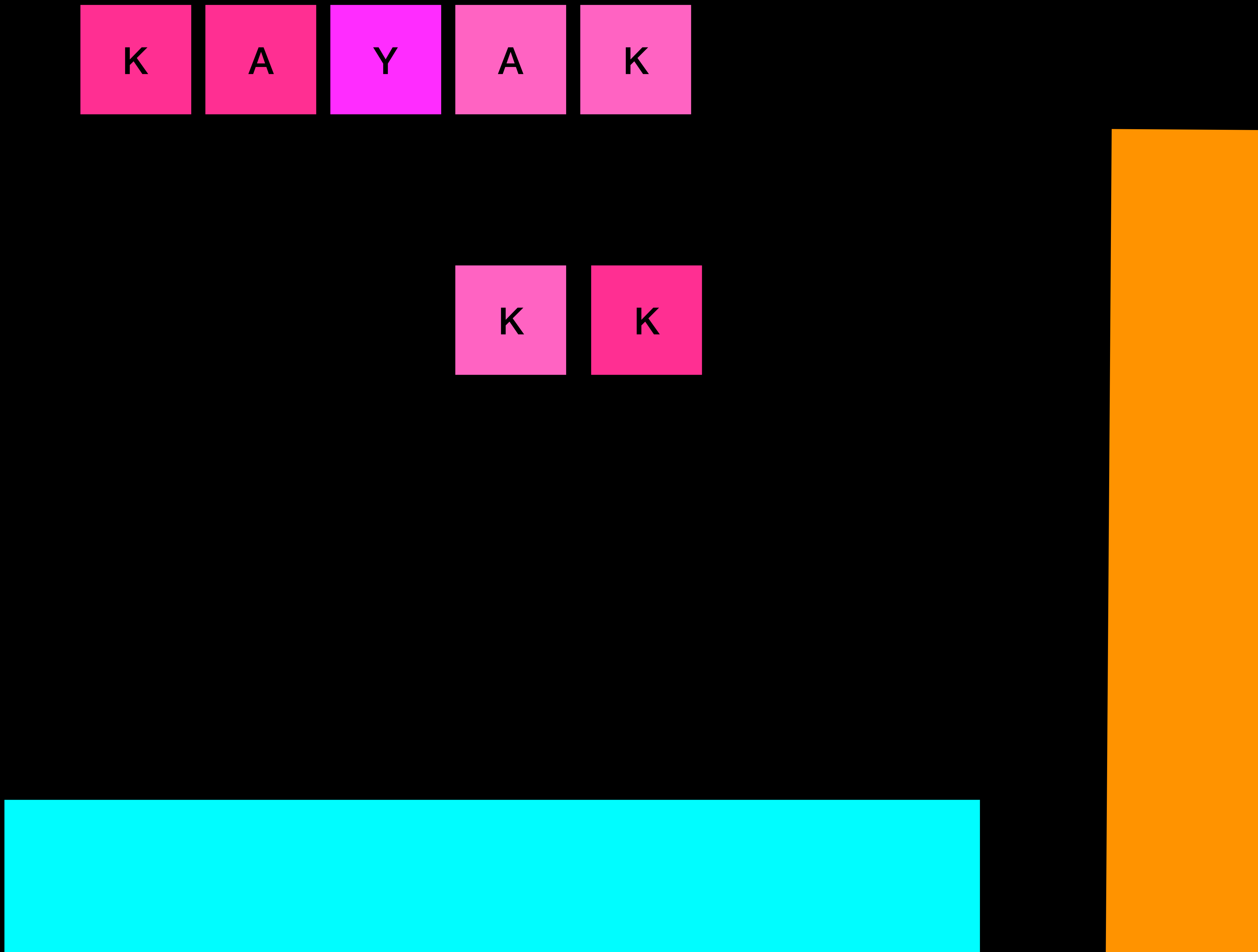

K A Y A K
K A Y A K
K A Y A K


K A Y A K
K K
A
Y
A
K
A Y A K


K A Y A K
A A
Y A K
Y A K


K A Y A K
A
K
Y Y
A K


K A Y A K
K
A A
K


K A Y A K
K K

```
bool isPalidrome(word)
{
    for(each character in word)
        add character to both stack and queue

    caractersAreEqual = true

    while(queue is not empty and charactersAreEqual){
        if(queue.front() = stack.top()){
            queue.dequeue()
            stack.pop()
        }
        else
            charactersAreEqual = false
    }
    return charactersAreEqual
}
```

Exam Drill:
Analyze the worst-case time complexity of this algorithm
T(n) = ?
O(?)

```
bool isPalidrome(word)
{
    for(each character in word)
        add character to both stack and queue

    caractersAreEqual = true

    while(queue is not empty and charactersAreEqual){
        if(queue.front() = stack.top()){
            queue.dequeue()
            stack.pop()
        }
        else
            charactersAreEqual = false
    }
    return charactersAreEqual
}
```

```
    bool isPalidrome(word)
    {
n       for(each character in word)
            add character to both stack and queue

        caractersAreEqual = true

n       while(queue is not empty and charactersAreEqual){
            if(queue.front() = stack.top()){
                queue.dequeue()
                stack.pop()
            }
            else
                charactersAreEqual = false
        }
        return charactersAreEqual
    }
```

$T(n) = 2n + k$ $O(n)$

In-Class Task

Write `isPalindome()` as a RECURSIVE function (without using stack and queue)

```
bool isPalindrome(string const& word, int first, int last)
{
    //base case: a string with 0 or 1 character is a palindrome
    if(last - first <= 1)
        return true;
    // first and last are different, it is not a palindrome
    if(word[first] != word[last])
        return false;
    // first = last so check if smaller word is a palindrome
    return isPalindrome(word, first+1, last-1);
}
```

Deque

Double ended queue (deque)

Can add and remove to front and back

Deque

Double ended queue (deque)

Can add and remove to front and back

Deque

Double ended queue (deque)

Can add and remove to front and back

Deque

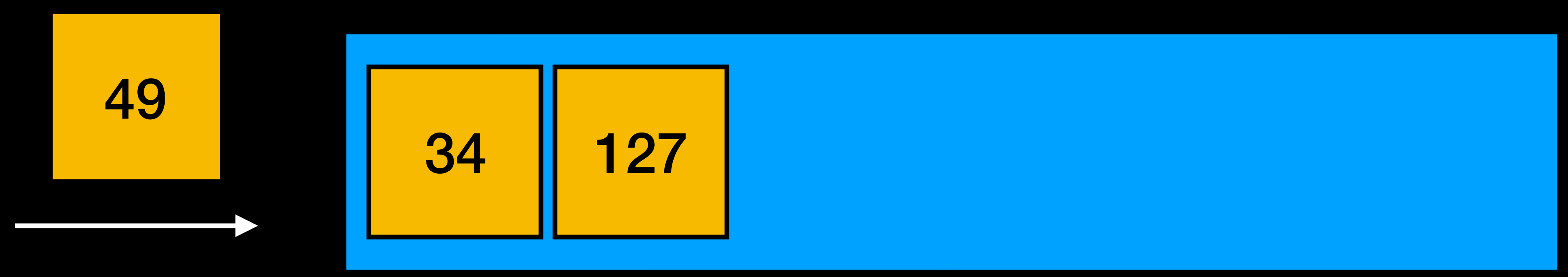
Double ended queue (deque)

Can add and remove to front and back

Deque

Double ended queue (deque)

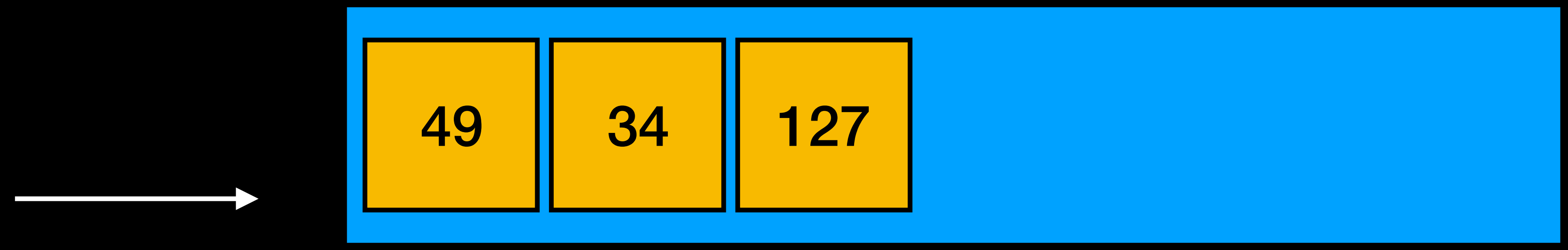
Can add and remove to front and back



Deque

Double ended queue (deque)

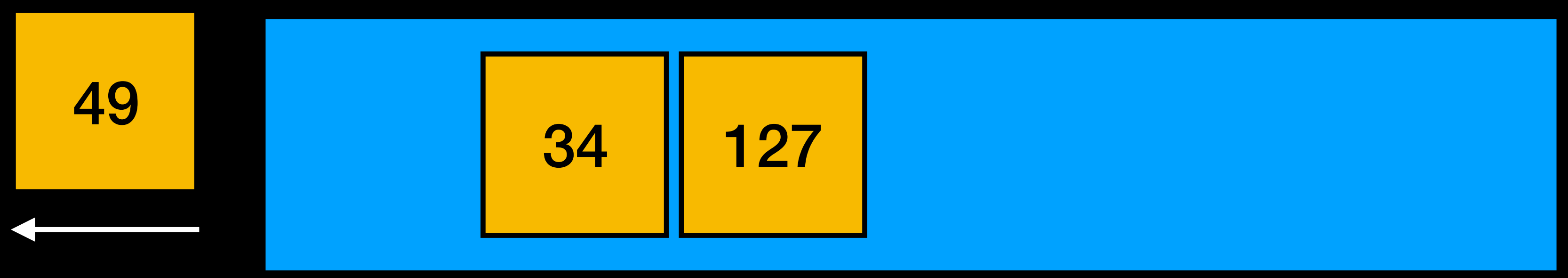
Can add and remove to front and back



Deque

Double ended queue (deque)

Can add and remove to front and back



Deque

Double ended queue (deque)

Can add and remove to front and back

Deque

Double ended queue (deque)

Can add and remove to front and back

Priority Queue

Orders elements by priority => removing an element will return the element with highest priority value

Elements with same priority kept in queue order (in some implementations)

Commonly (but not always) implemented with a Heap (we may cover Heaps if we have time after Trees, if so we will look at its implementation)