Queue ADT

Tiziana Ligorio

Today's Plan

Announcements

Queue ADT

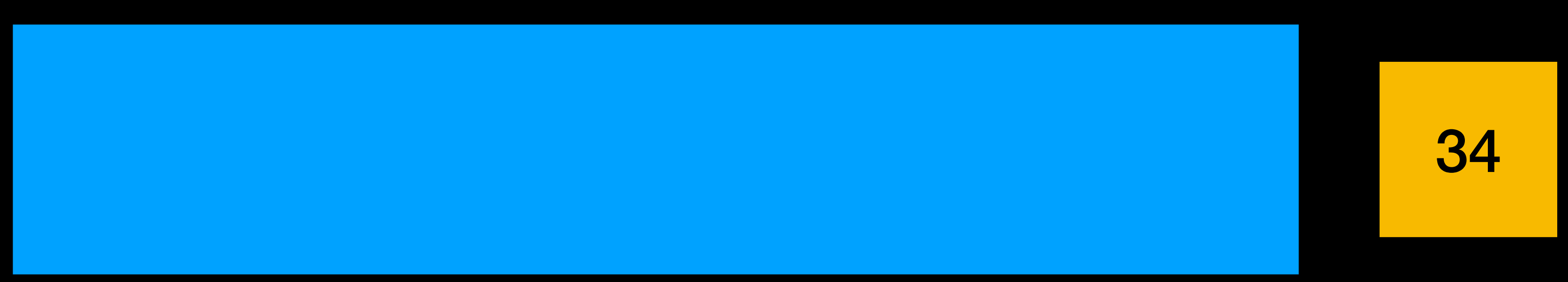
Applications

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

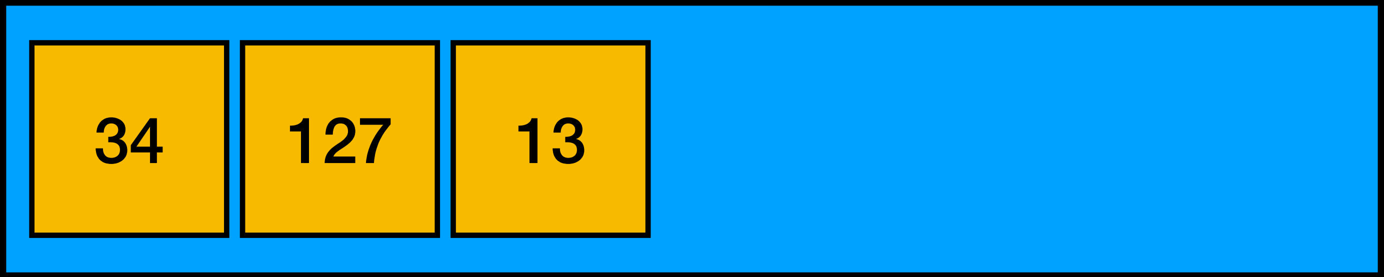
or dequeued from the front of the line

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

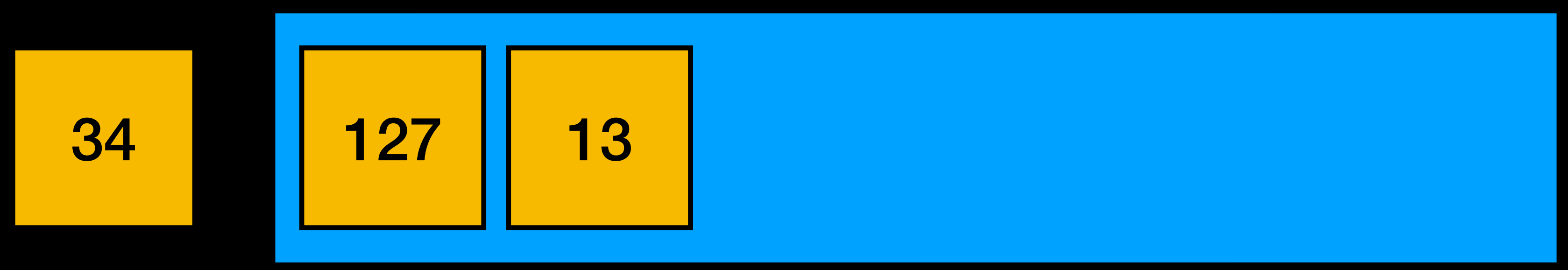


Queue

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

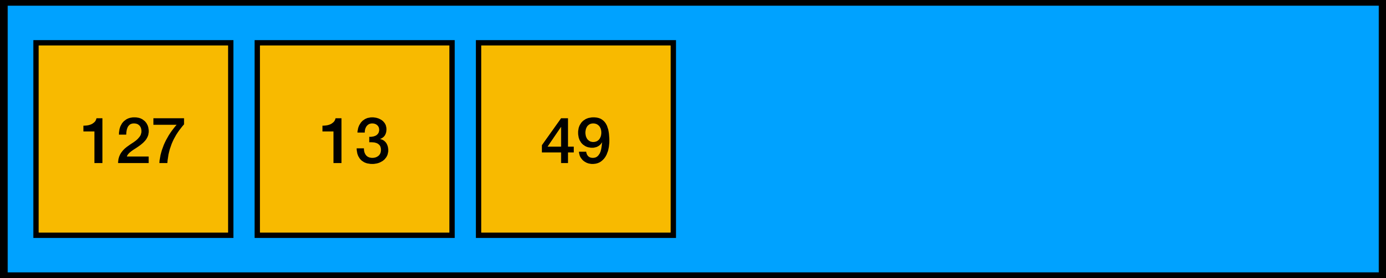
or dequeued from the front of the line

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

Any waiting queue

- Print jobs
- OS scheduling processes with equal priority
- Messages between asynchronous processes

. . .

Queue Applications

Generating all substrings

Any waiting queue

- Print jobs
- OS scheduling processes with equal priority
- Messages between asynchronous processes

. . .

Generating all substrings

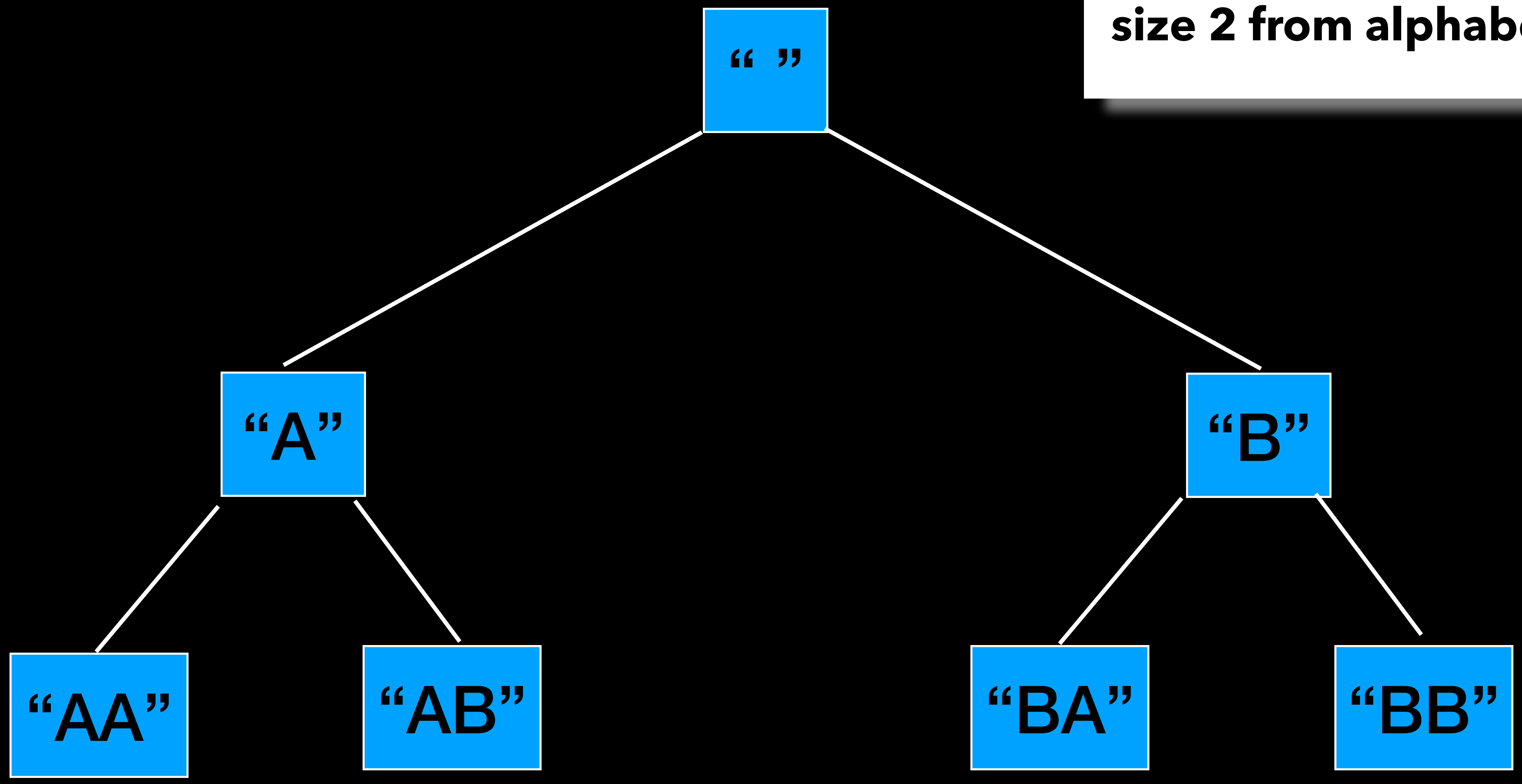
Generate all possible strings **up to** some fixed length **n with repetition (same character included multiple times)**

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

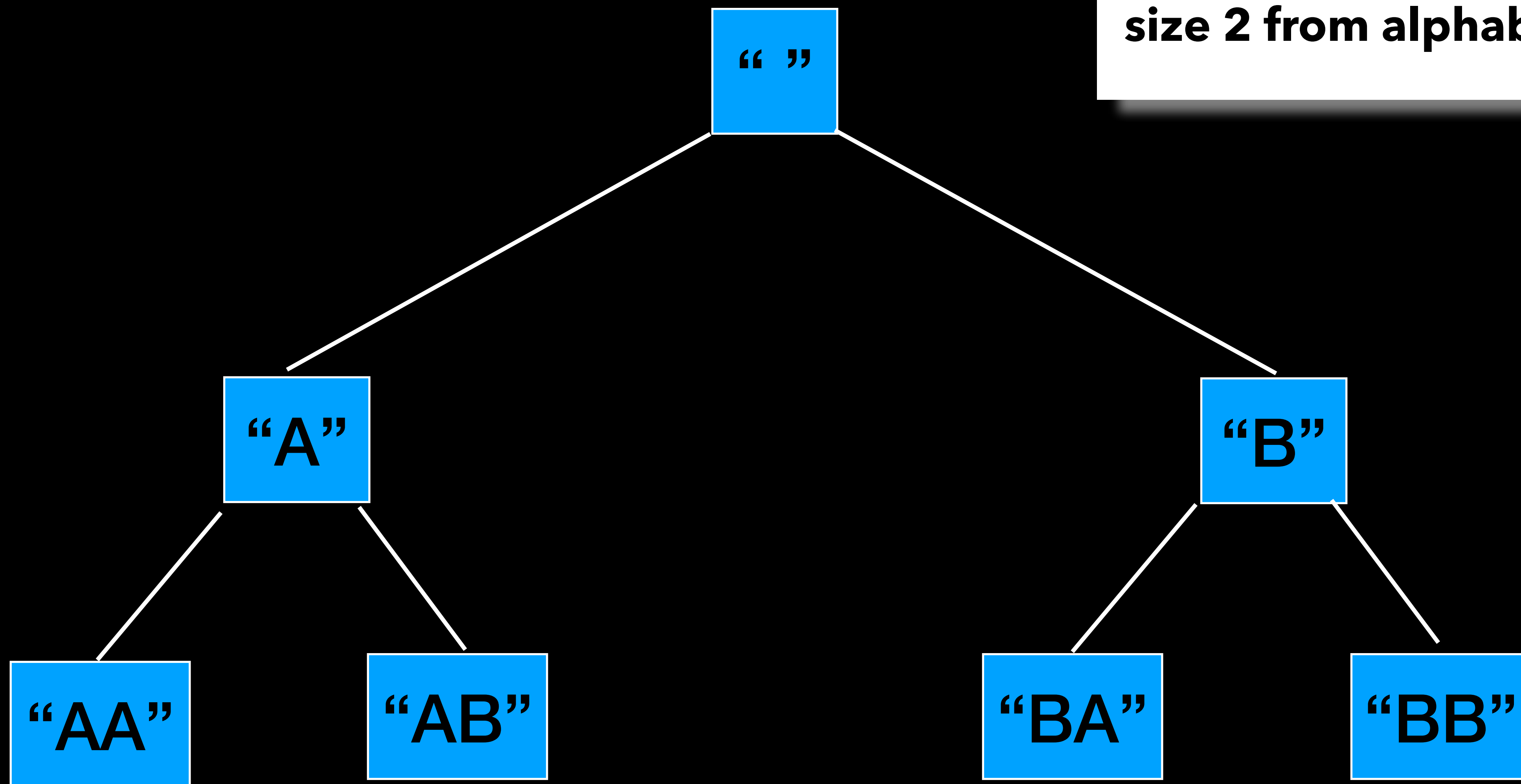
How might we do it with a queue?

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

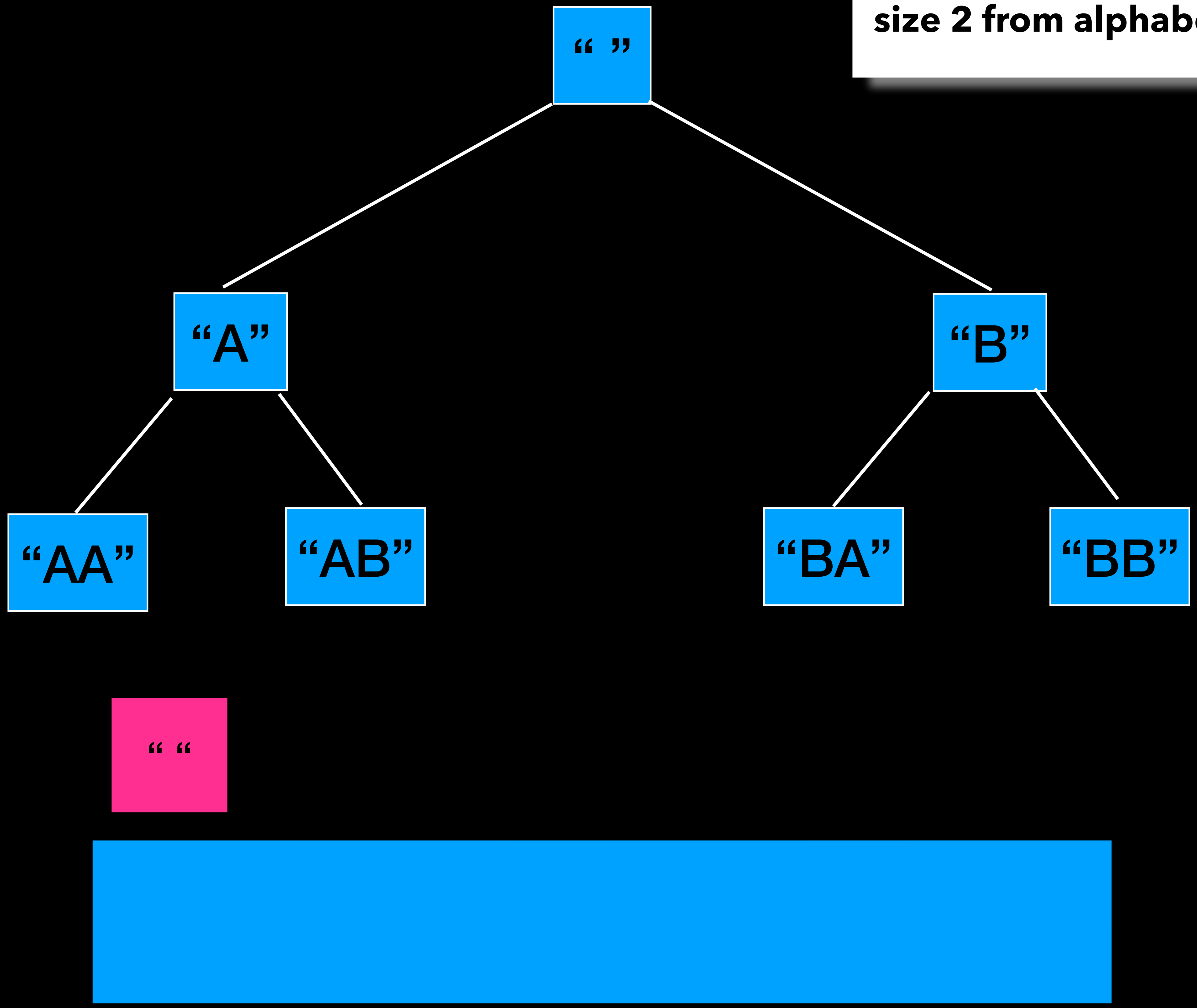
Generate all substrings of size 2 from alphabet {'A', 'B'}



Generate all substrings of size 2 from alphabet {‘A’, ‘B’}

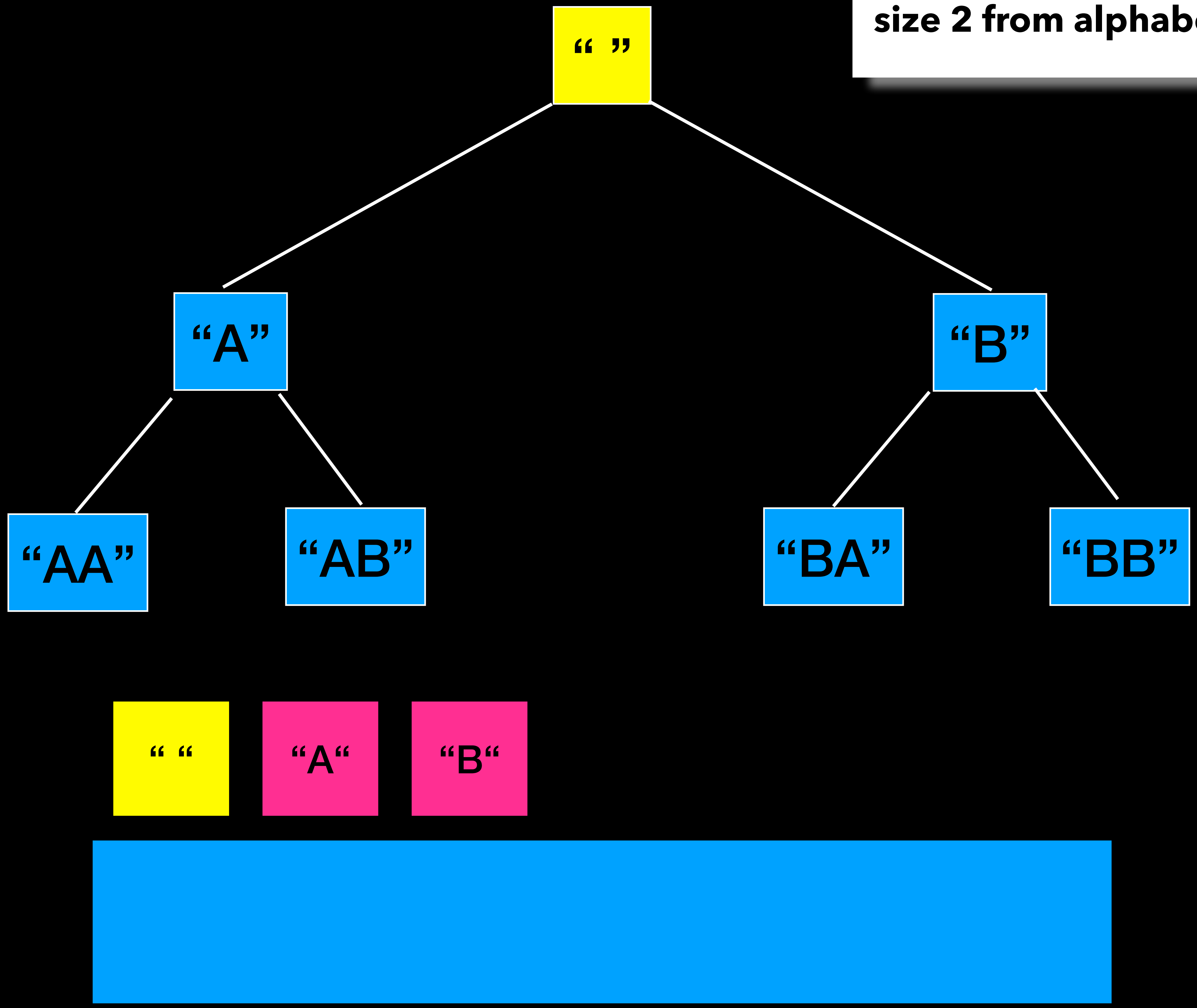


Generate all substrings of size 2 from alphabet {'A', 'B'}



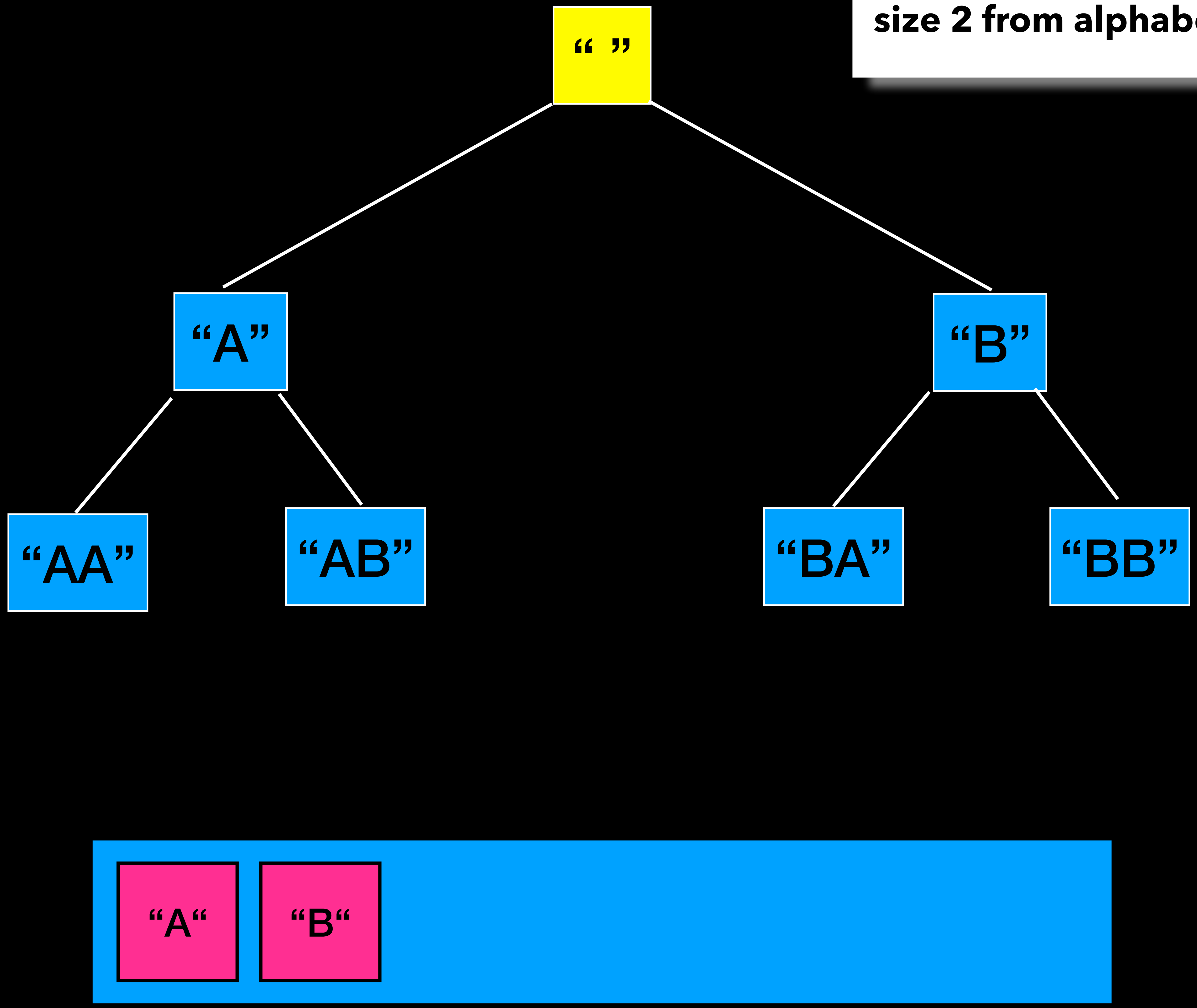
{ ""}

Generate all substrings of size 2 from alphabet {'A', 'B'}



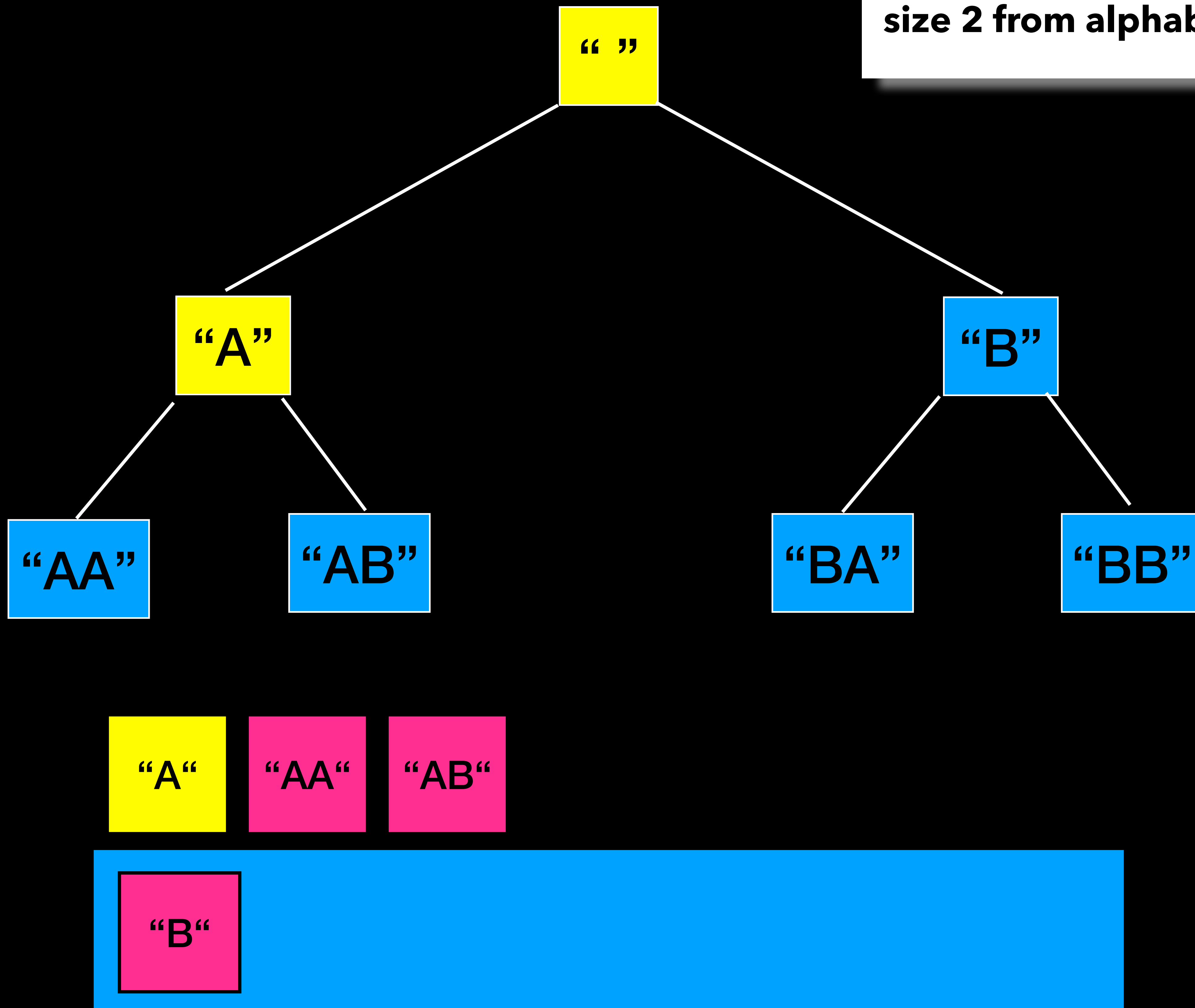
{ “”}

Generate all substrings of size 2 from alphabet {‘A’, ‘B’}



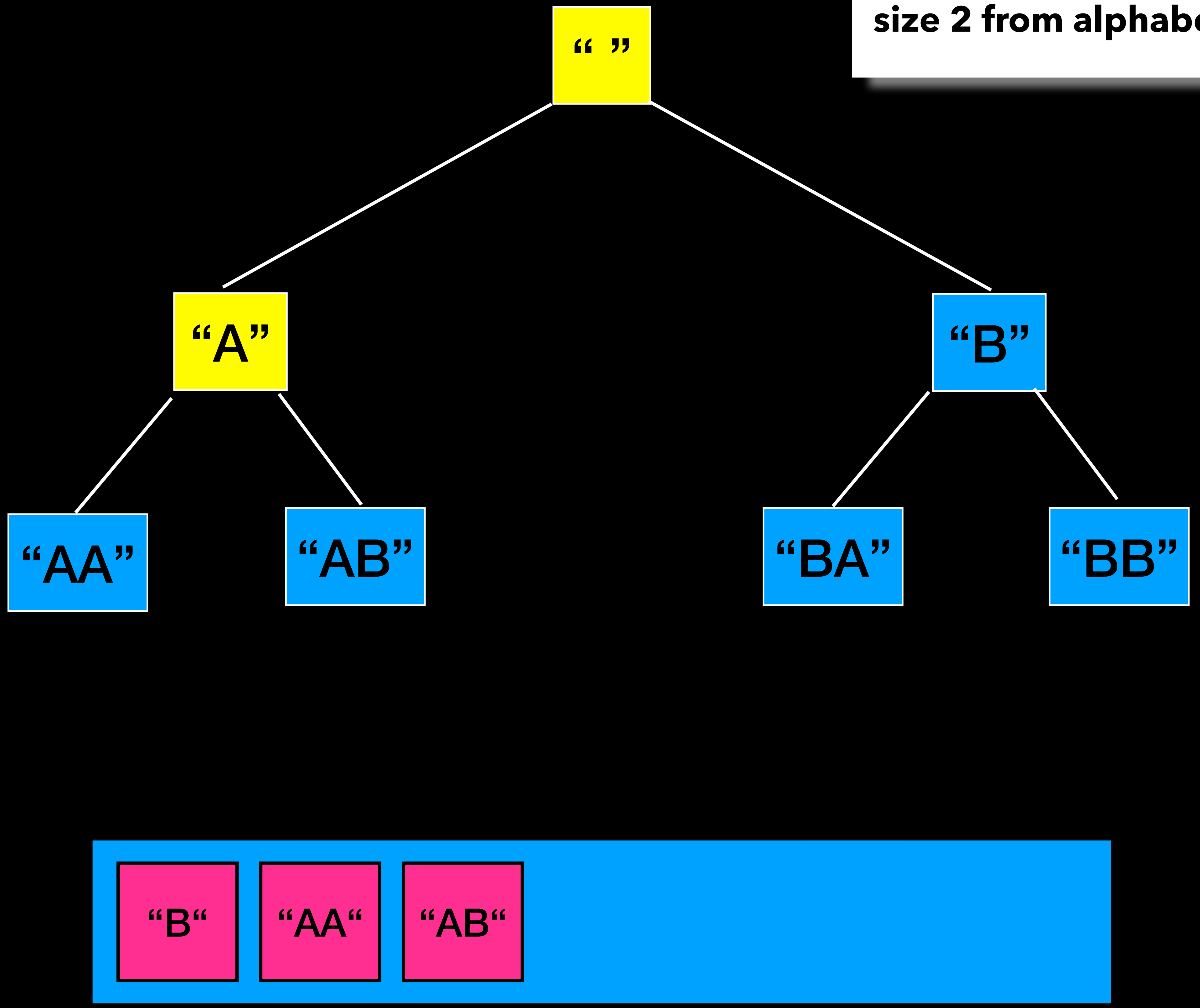
{ “”, “A”}

Generate all substrings of size 2 from alphabet {‘A’, ‘B’}



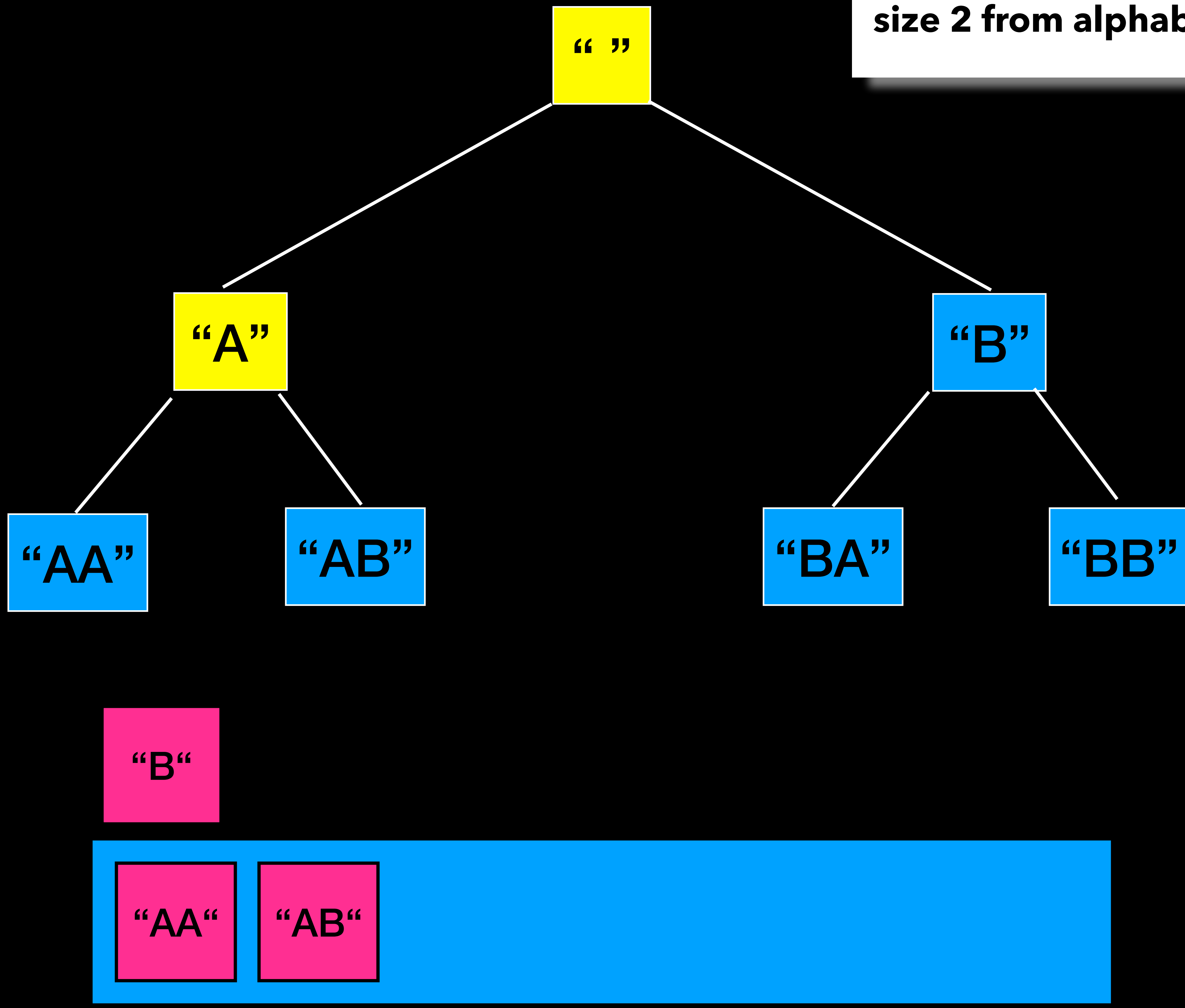
{ "", "A"}

Generate all substrings of size 2 from alphabet {'A', 'B'}



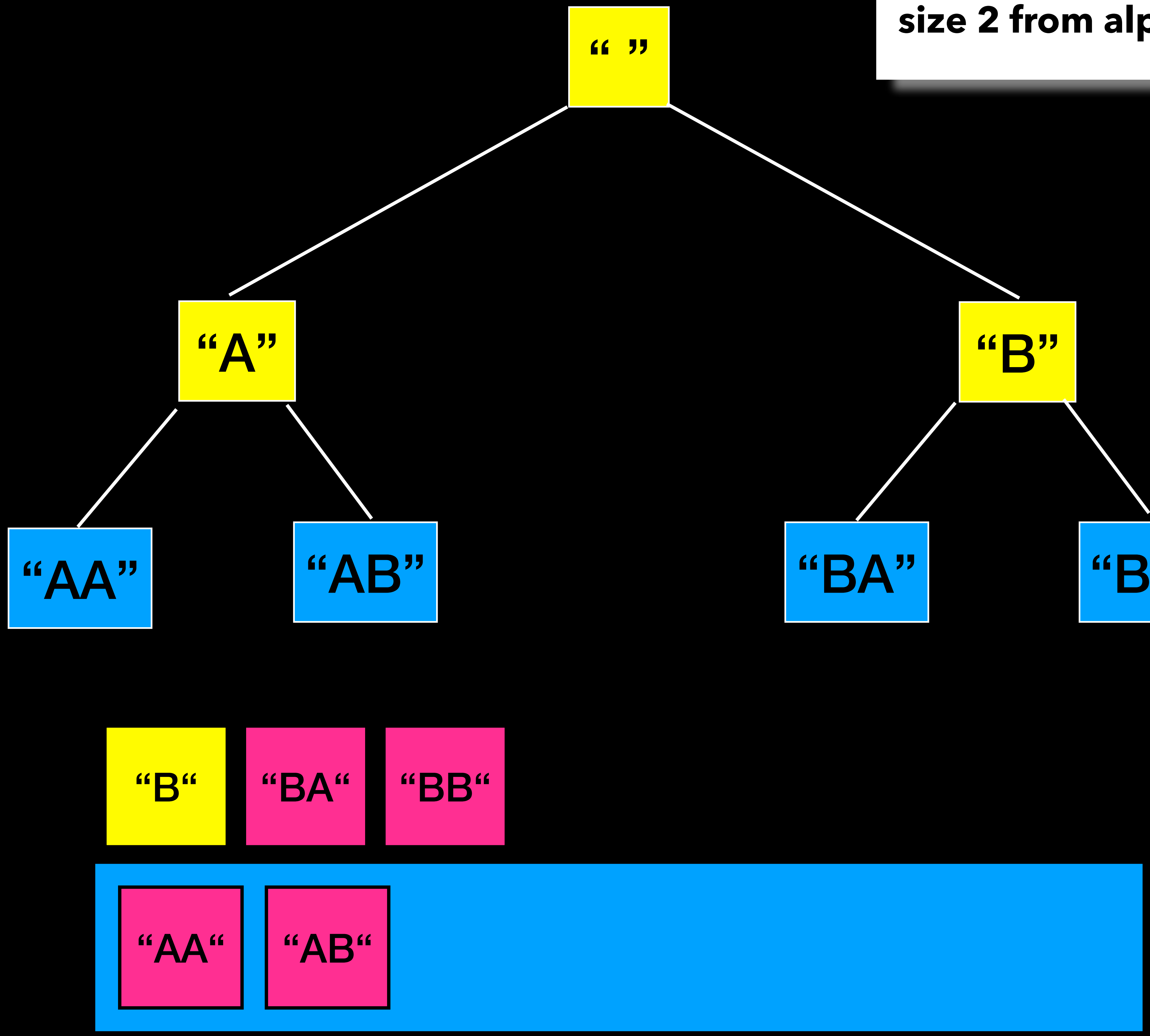
{ “”, “A”}

Generate all substrings of size 2 from alphabet {‘A’, ‘B’}



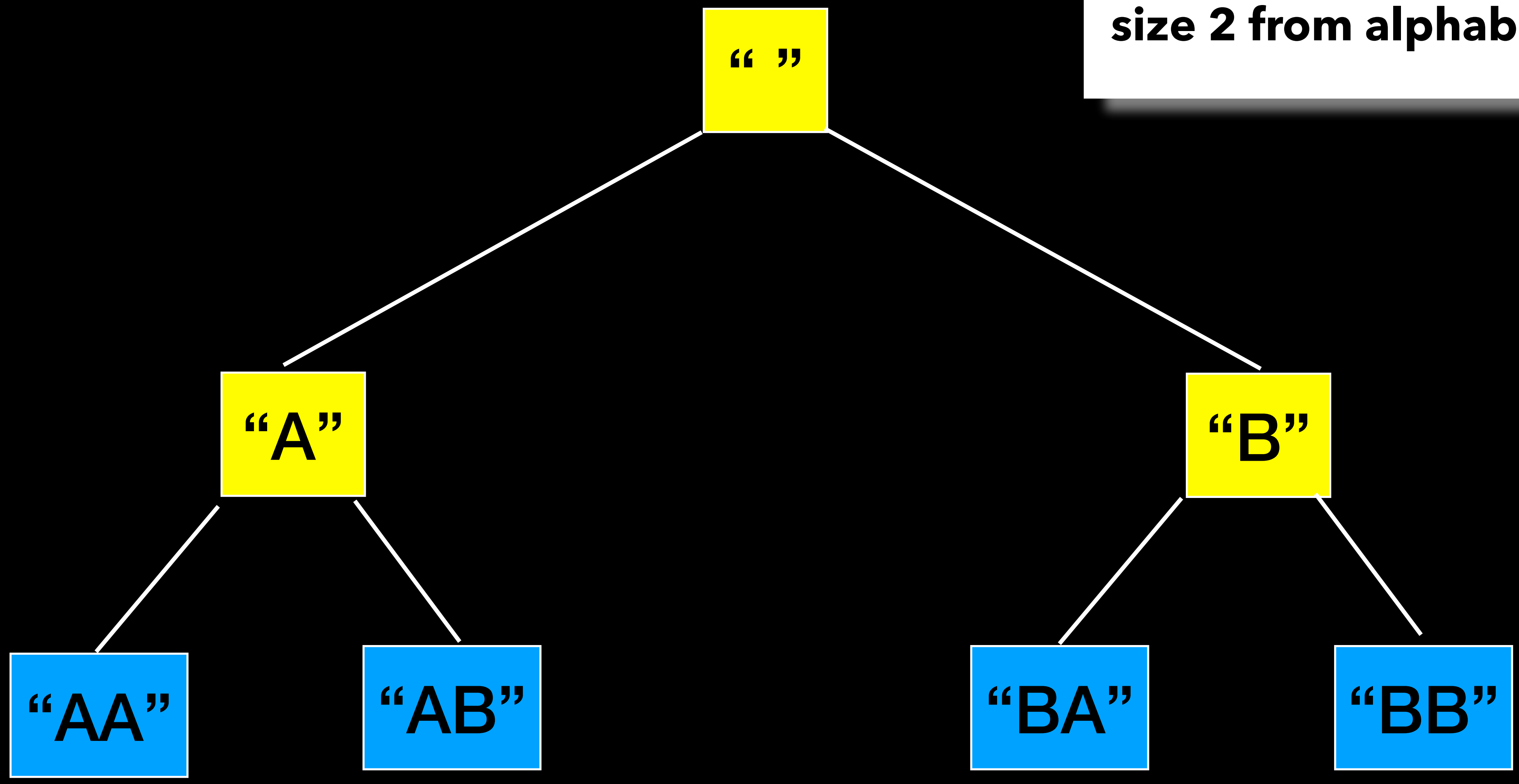
{ “”, “A”, “B”}

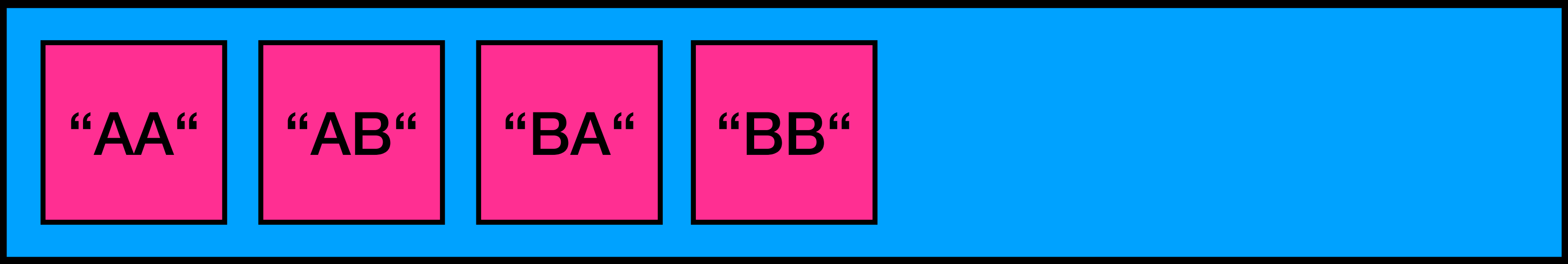
Generate all substrings of size 2 from alphabet {‘A’, ‘B’}



{ "", "A", "B"}

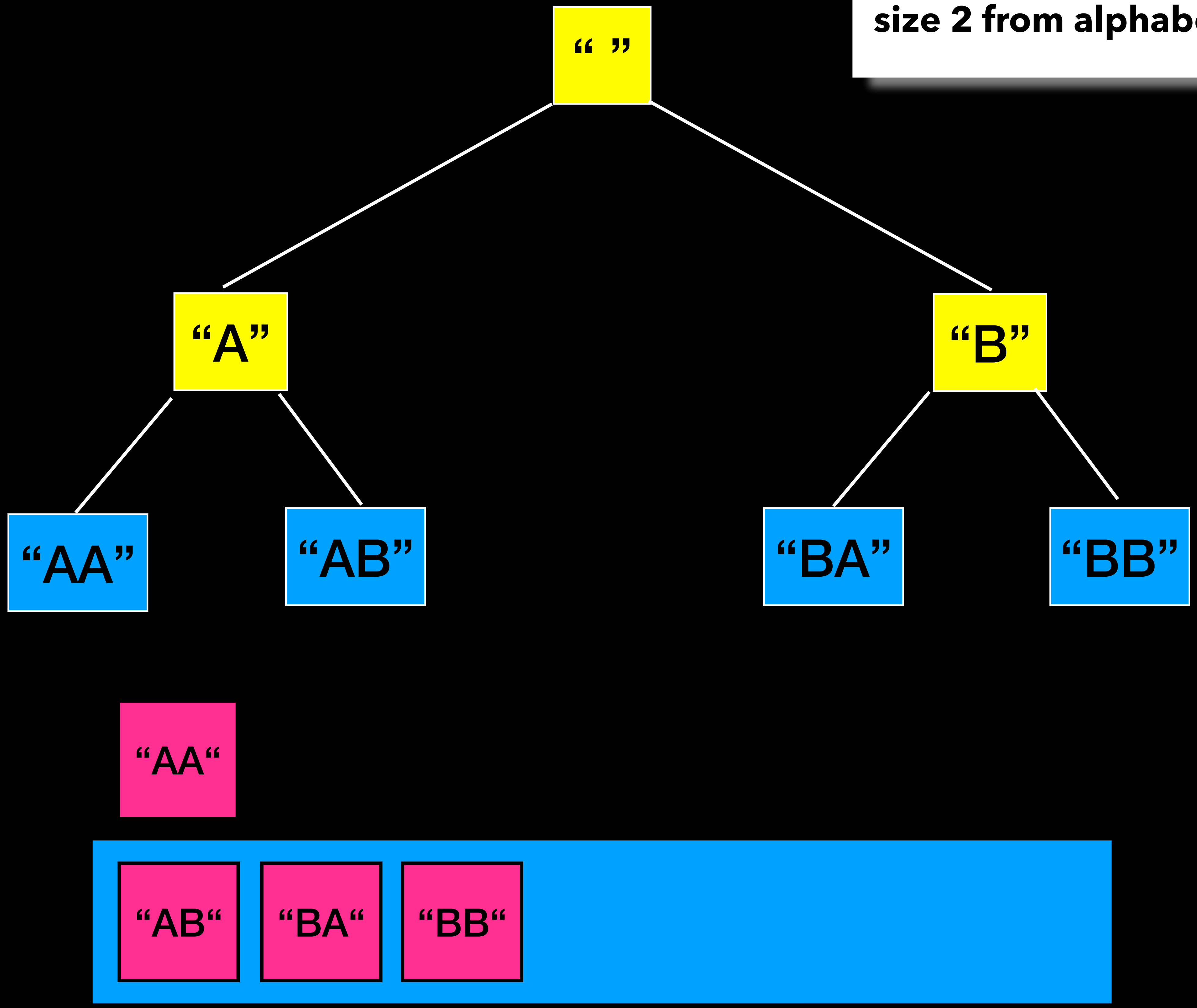
Generate all substrings of size 2 from alphabet {'A', 'B'}





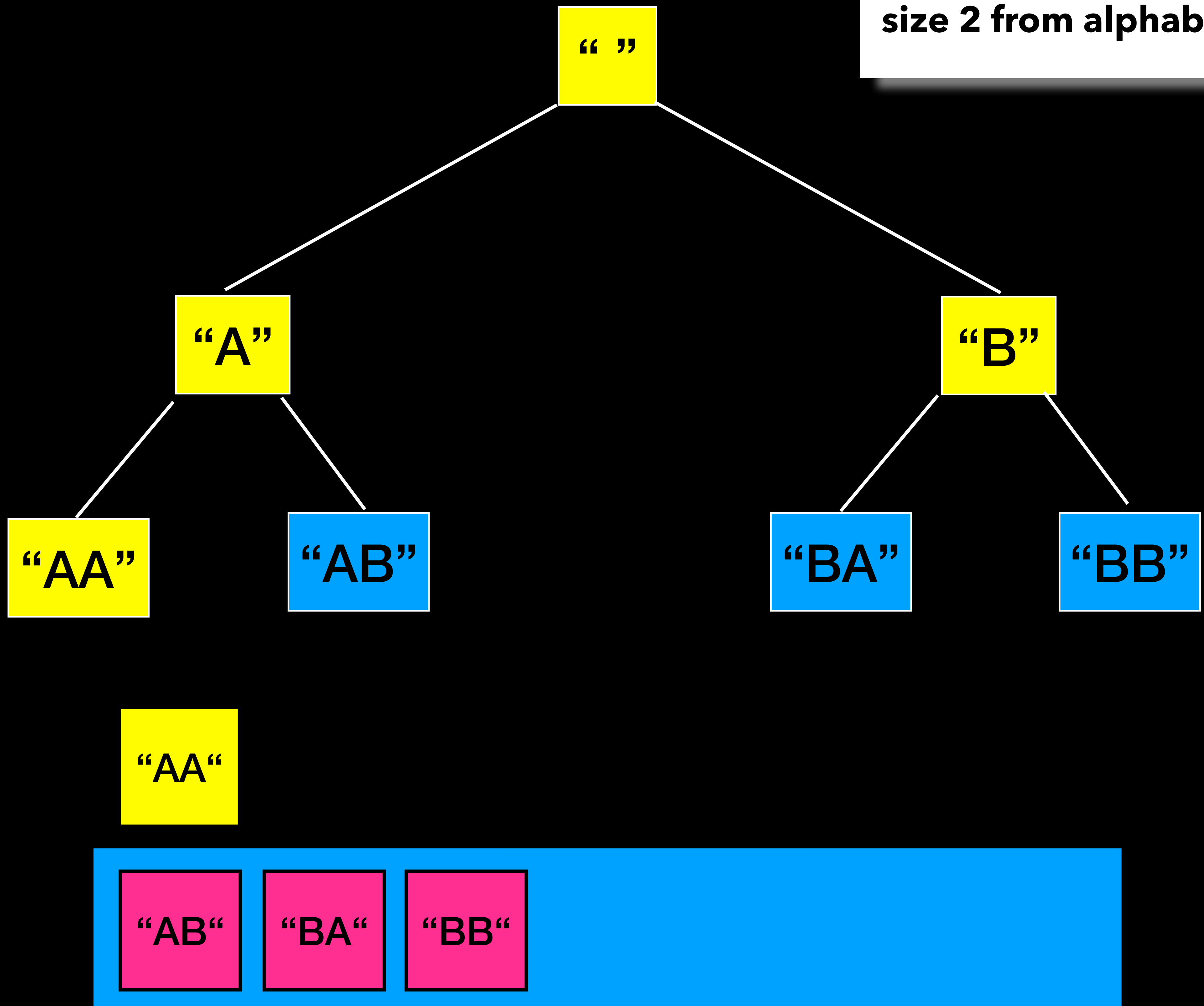
{ "", "A", "B"}

Generate all substrings of size 2 from alphabet {'A', 'B'}



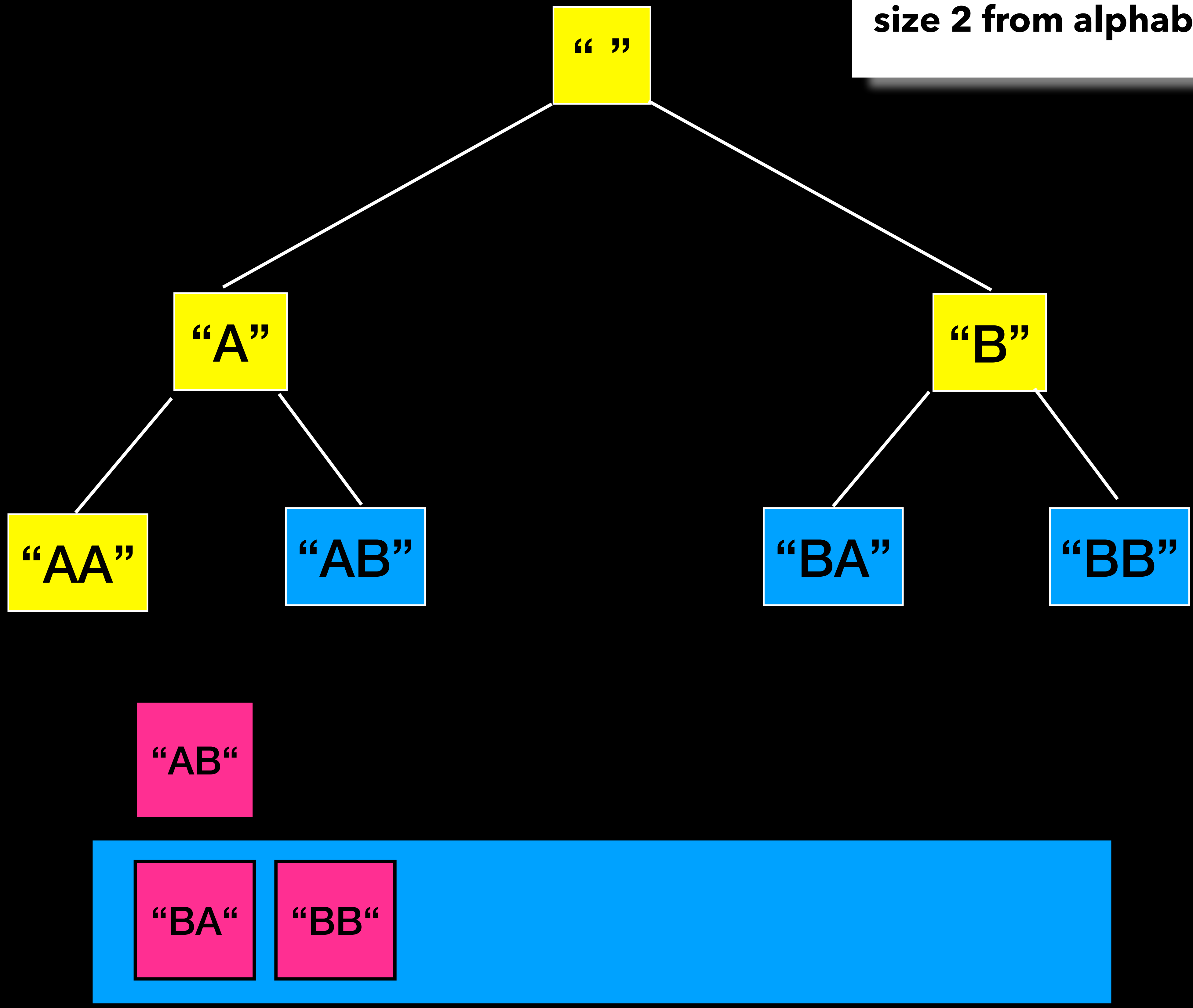
{ "", "A", "B", "AA"}

Generate all substrings of size 2 from alphabet {'A', 'B'}



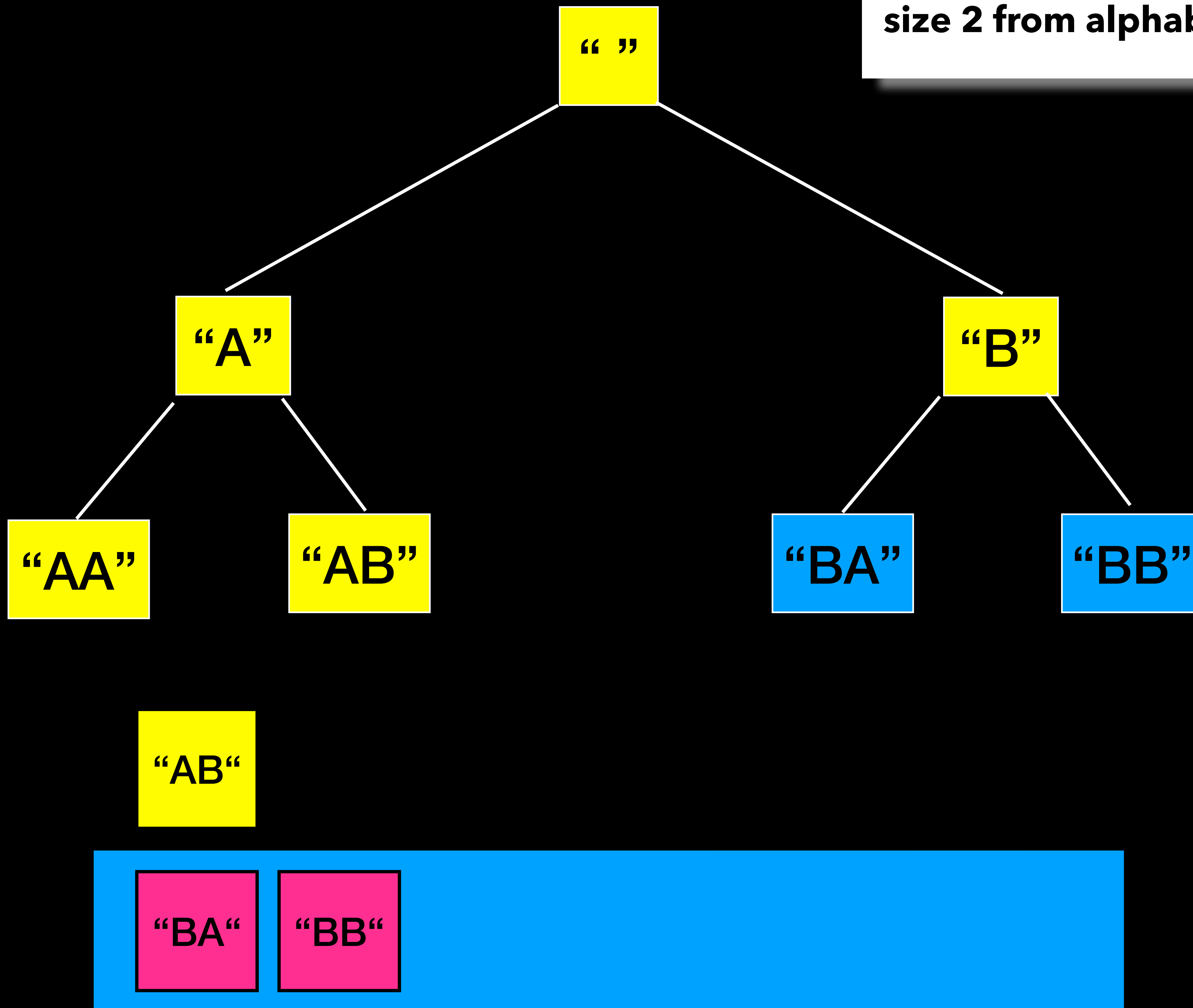
{ "", "A", "B", "AA"}

Generate all substrings of size 2 from alphabet {'A', 'B'}



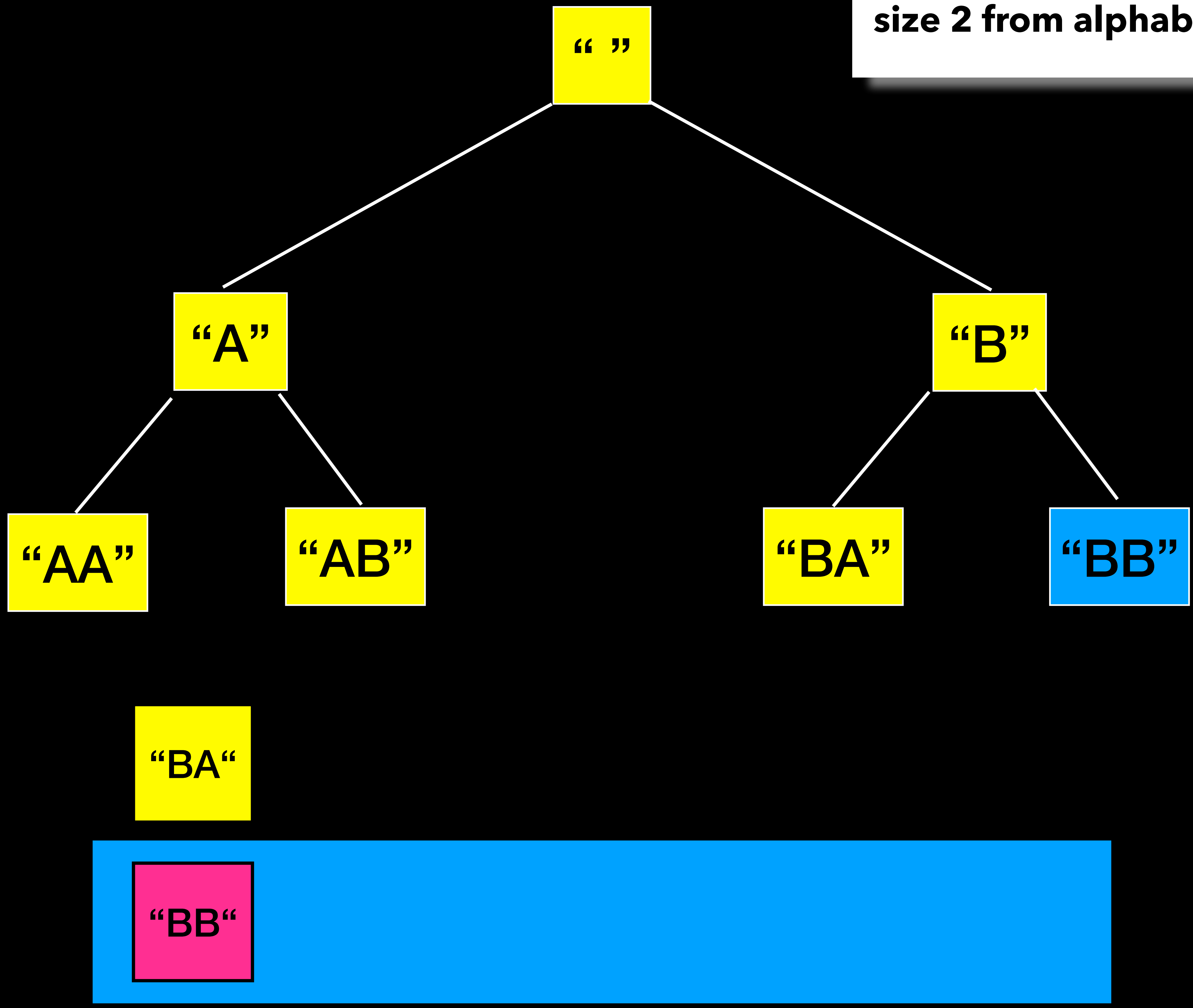
{ “”, “A”, “B”, “AA”, “AB”}

Generate all substrings of size 2 from alphabet {‘A’, ‘B’}



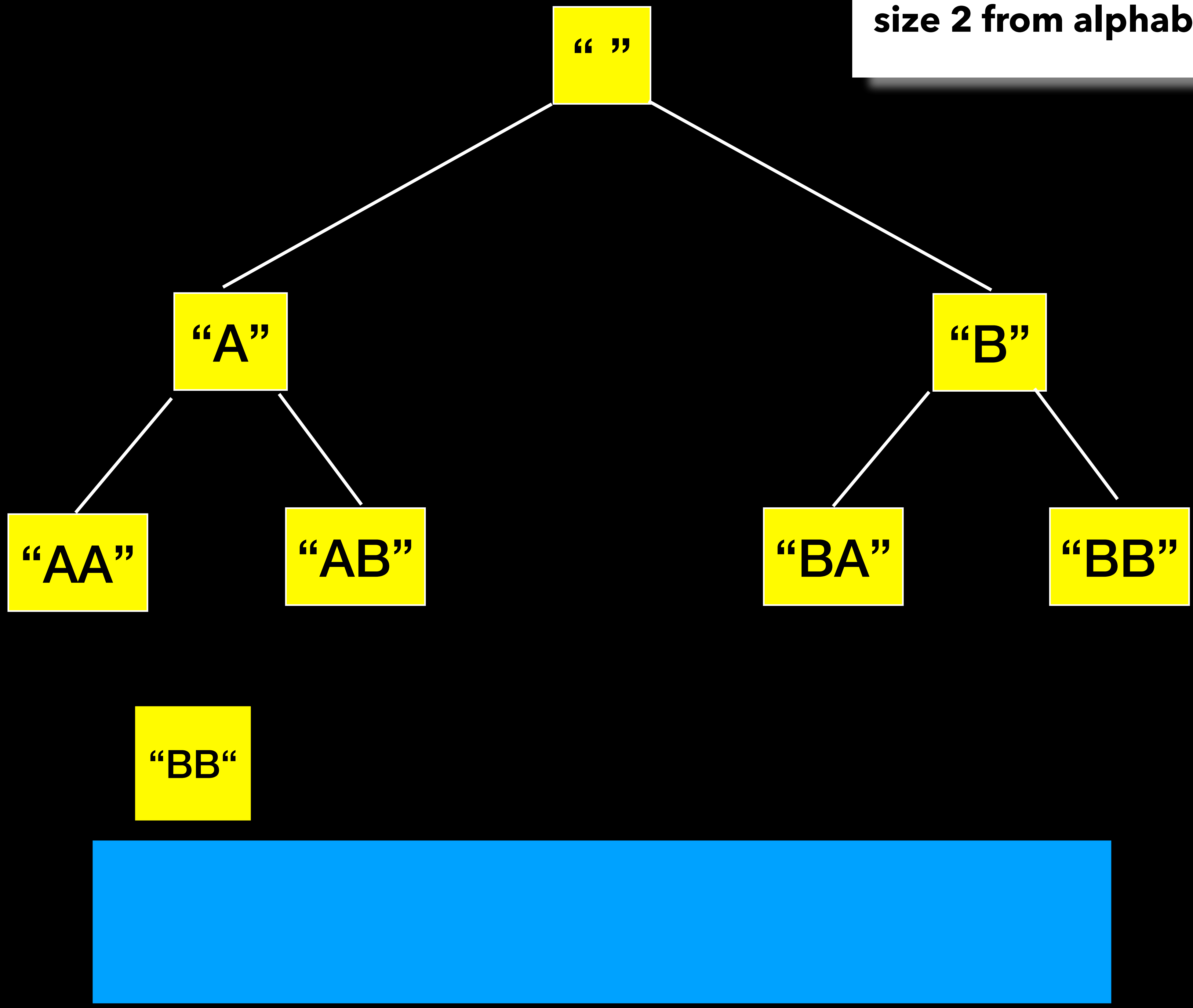
{ "", "A", "B", "AA", "AB", "BA"}

Generate all substrings of size 2 from alphabet {'A', 'B'}



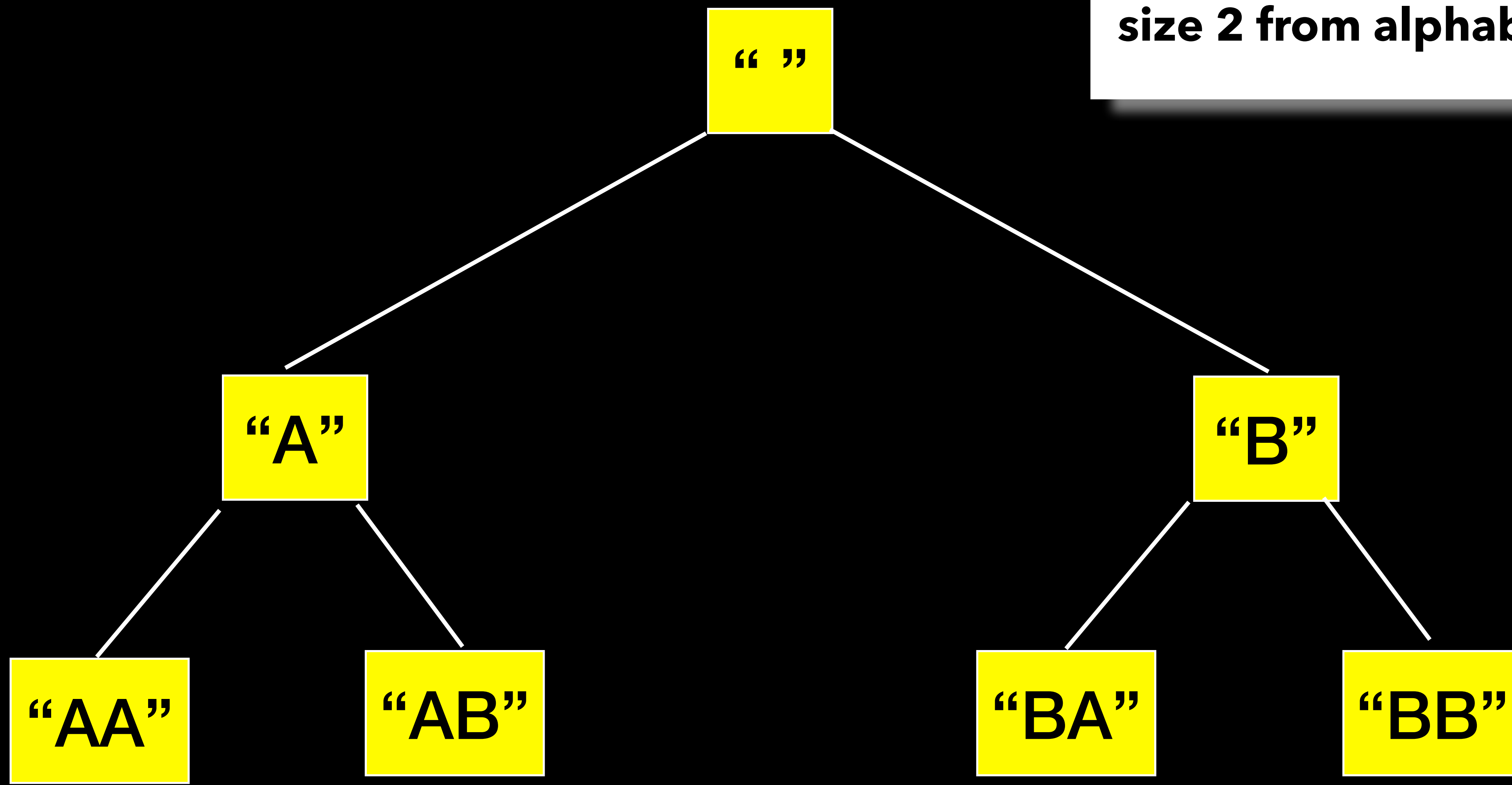
{ "", "A", "B", "AA", "AB", "BA", "BB" }

Generate all substrings of size 2 from alphabet {'A', 'B'}



{ "", "A", "B", "AA", "AB", "BA", "BB" }

Generate all substrings of size 2 from alphabet {'A', 'B'}



Breadth-First Search

Applications

- Find shortest path in graph
- GPS navigation systems
- Crawlers in search engines
- . . .

Generally good when looking for the "shortest" or "best" way to do something => lists things in increasing order of "size" stopping at the "shortest" solution

Size of Substring

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

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

Analysis

Finding all substrings (with repetition) of size **up to n**

Assume alphabet (A, B, … , Z) of size 26

The empty string= 1= 26^0 ""

All strings of size 1 = 26^1

A	B	C	. . .	Z

All strings of size 2 = 26^2

AA	BA	CA	. . .	ZA
AB	BC	CB	. . .	ZB
. . .				
AZ	BZ	CZ	. . .	ZZ

. . .

All strings of size n = 26^n

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

Lecture Activity

Analyze the worst-case time complexity of this algorithm assuming alphabet of size 26 and up to strings of length n

T(n) = ?

O(?)

Size of Substring

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

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

Will stop when all strings have been removed from queue

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

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

Will stop when all strings have been removed from queue

Removes 1 string from the queue

Adds 26 strings to the queue

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

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

Will stop when all strings have been removed from queue

Removes 1 string from the queue

Adds 26 strings to the queue

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

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

Loop until queue is empty and dequeue only 1 each time.

So the question becomes:

How many strings are enqueued in total?

Will stop when all strings have been removed from queue

Removes 1 string from the queue

Adds 26 strings to the queue

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

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

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

Will stop when all strings have been removed from queue

Removes 1 string from the queue

Adds 26 strings to the queue

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

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

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

Will stop when all strings have been removed from queue

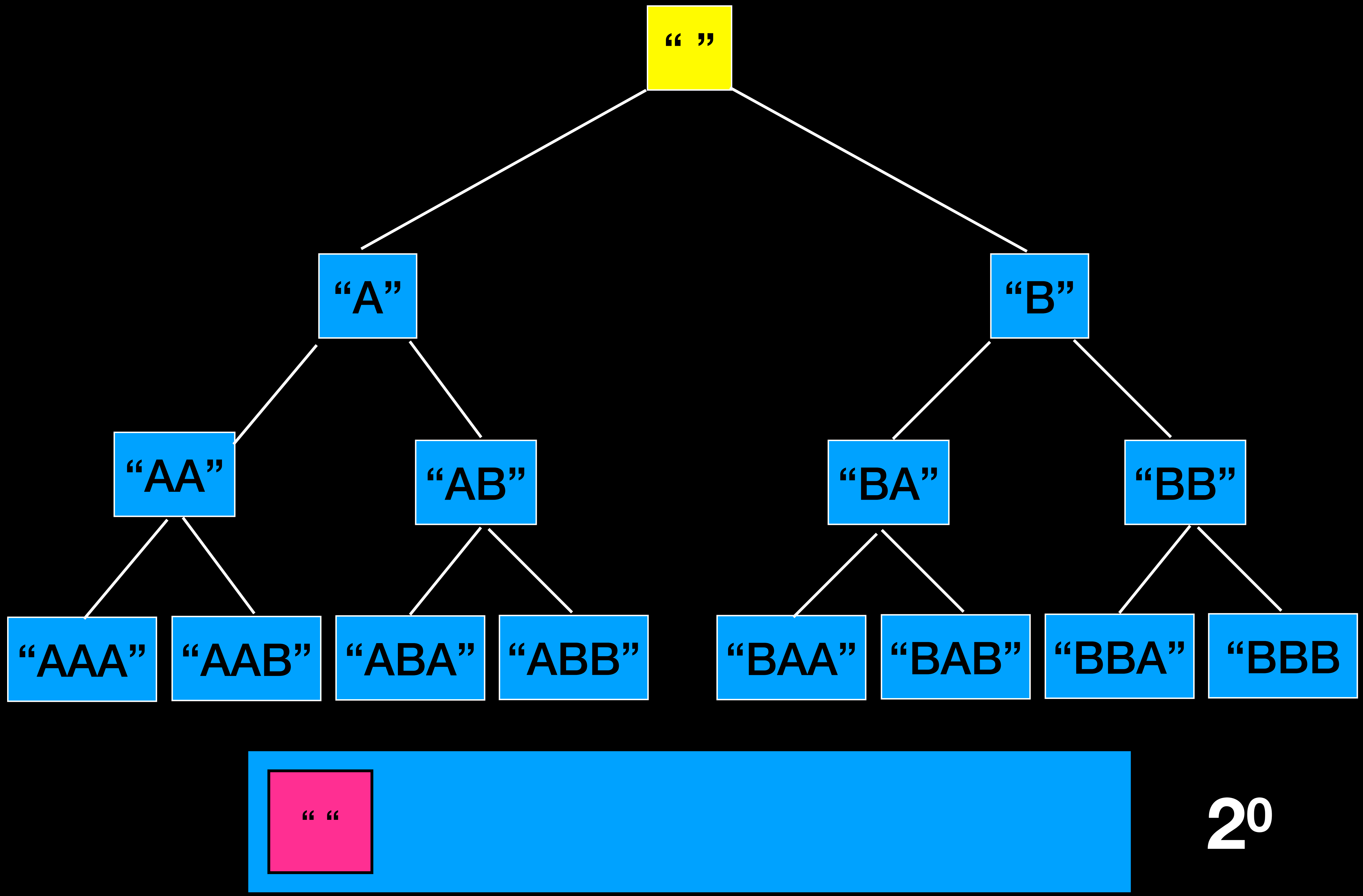
Removes 1 string from the queue

Adds 26 strings to the queue

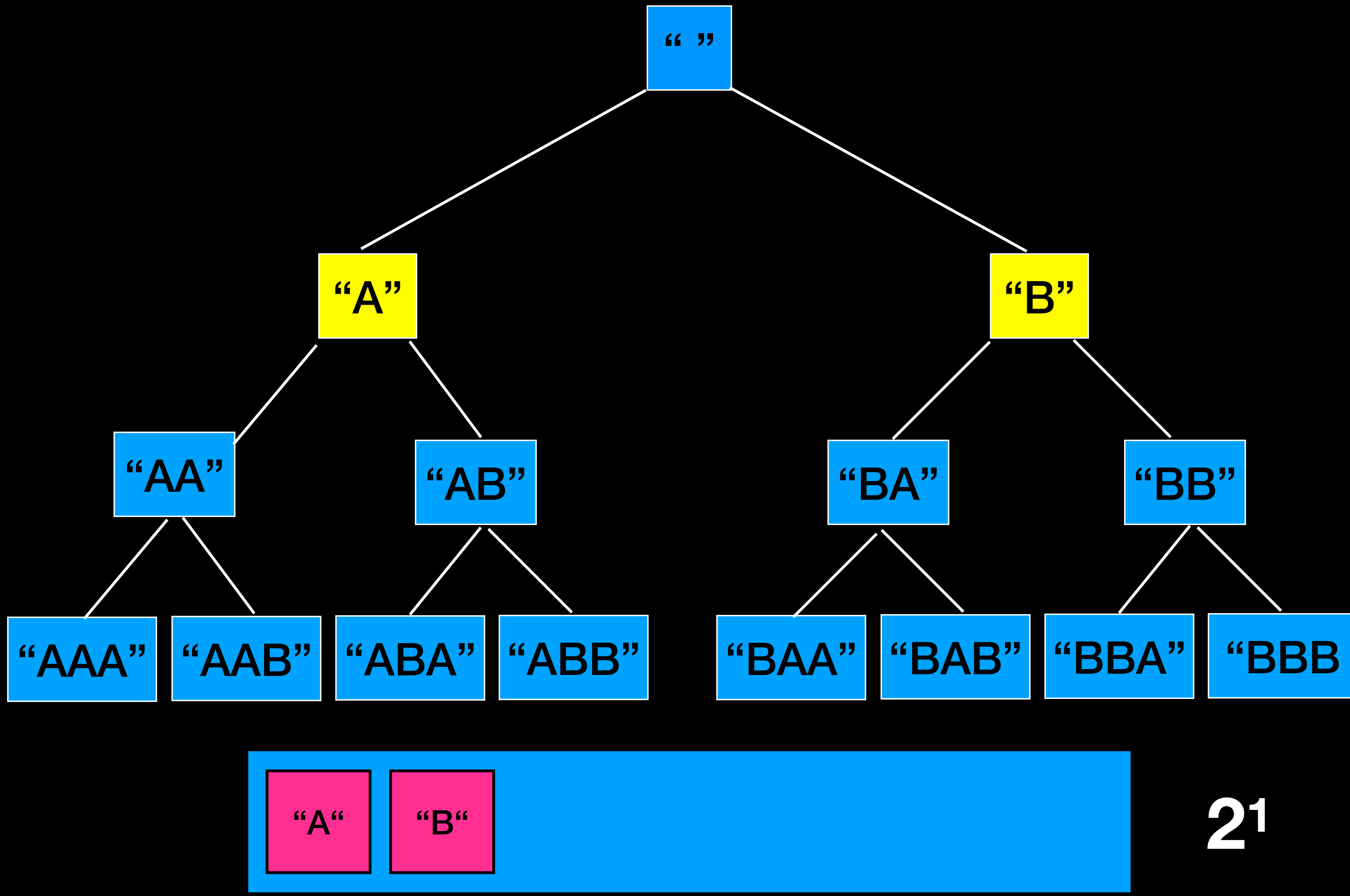
```
findAllSubstrings(int n)
{
    put empty string on the queue

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

$O(26^n)$

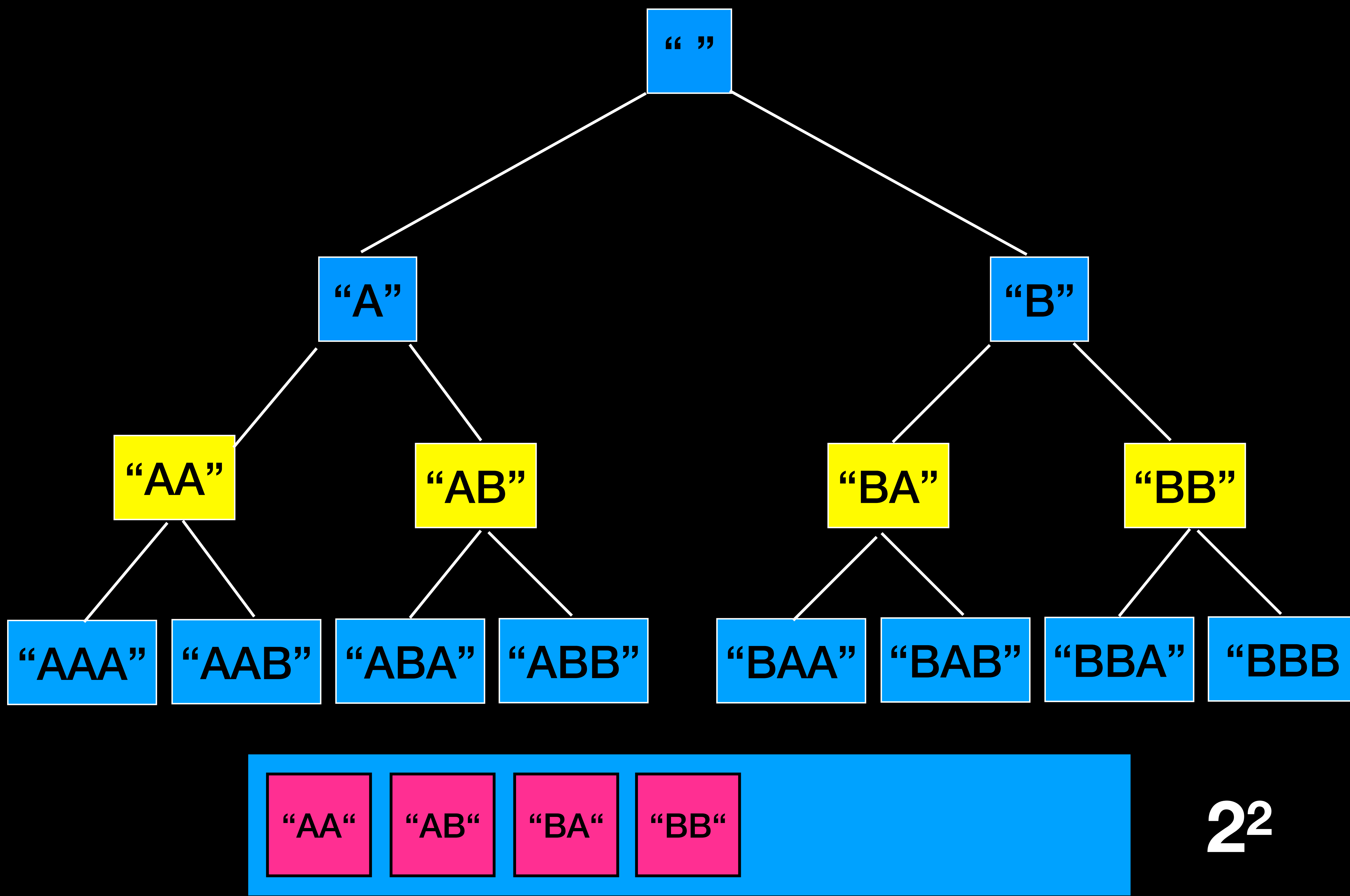
Let **n = 3**, alphabet still **{‘A’,’B’}**



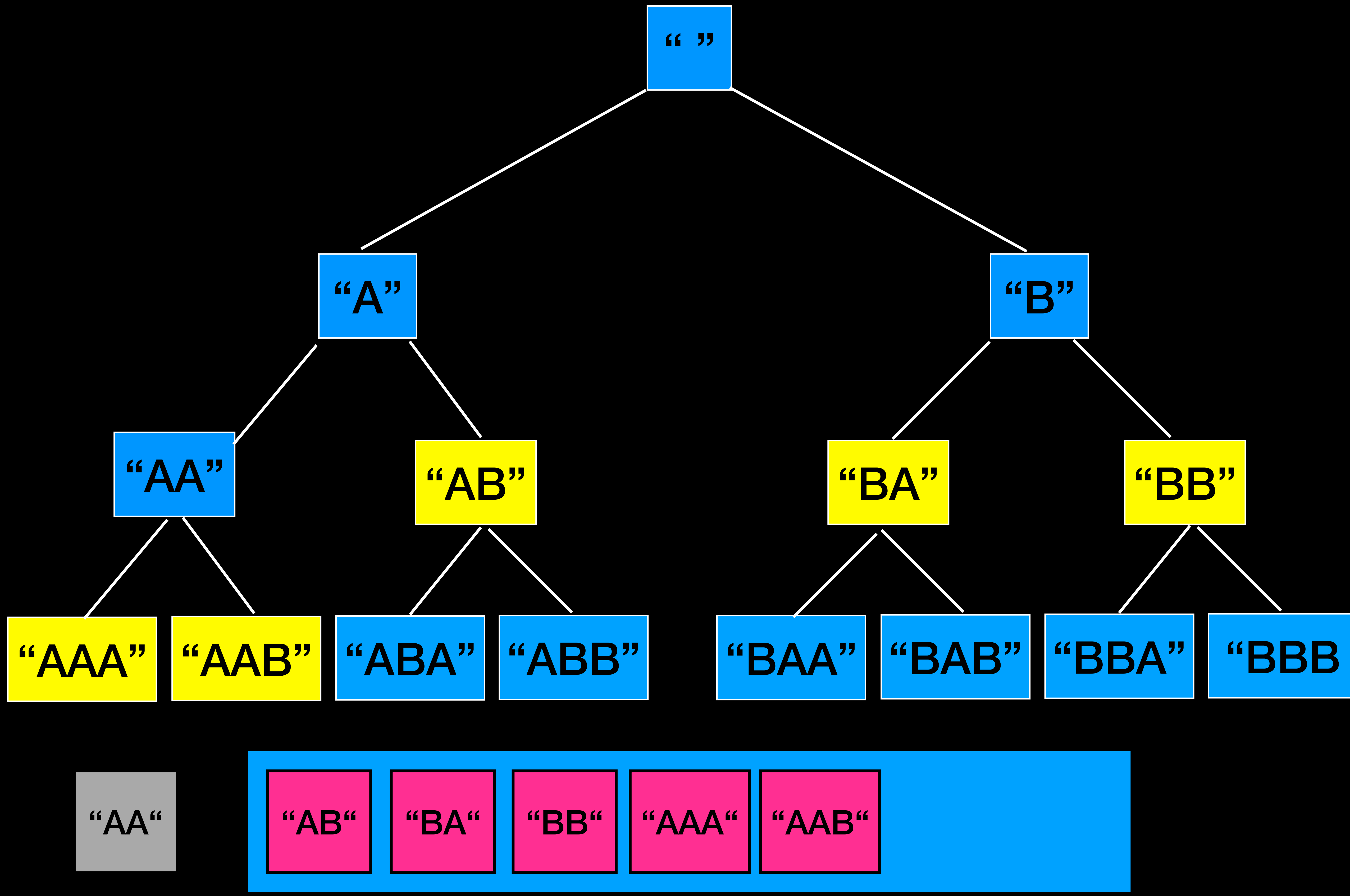
Let n = 3, alphabet still {‘A’,’B’}



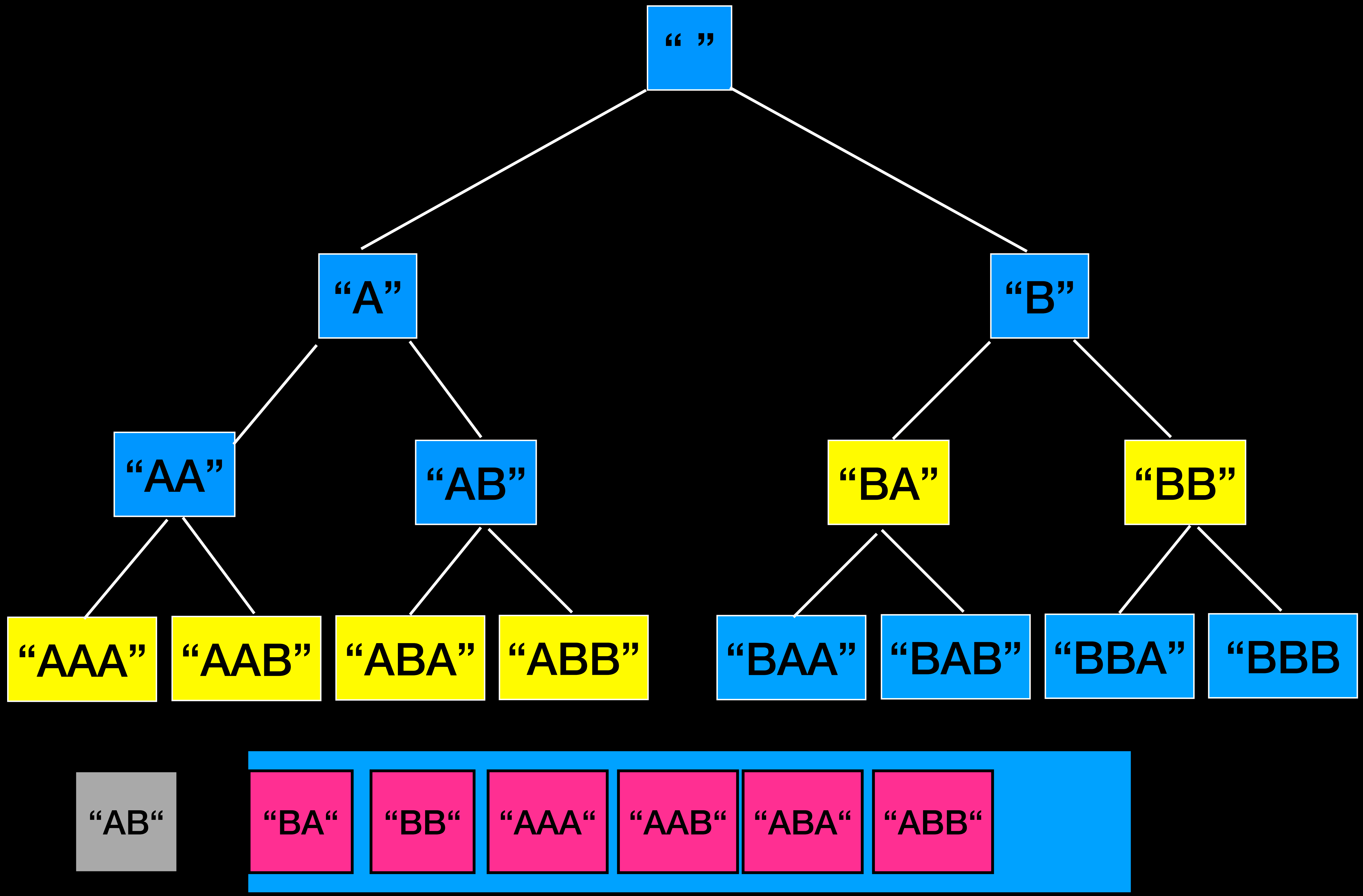
Let n = 3, alphabet still {‘A’,’B’}



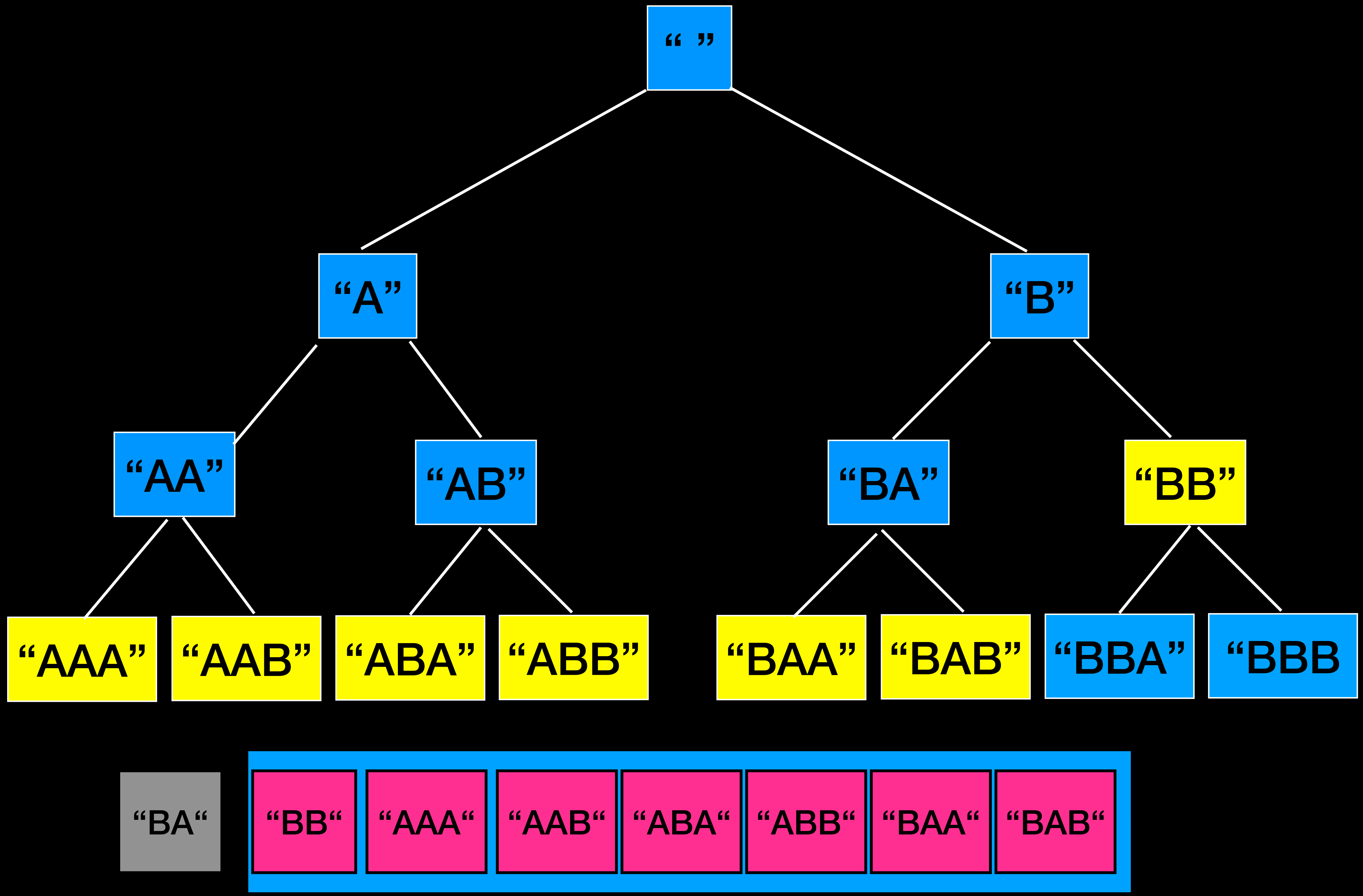
Let n = 3, alphabet still {‘A’,’B’}



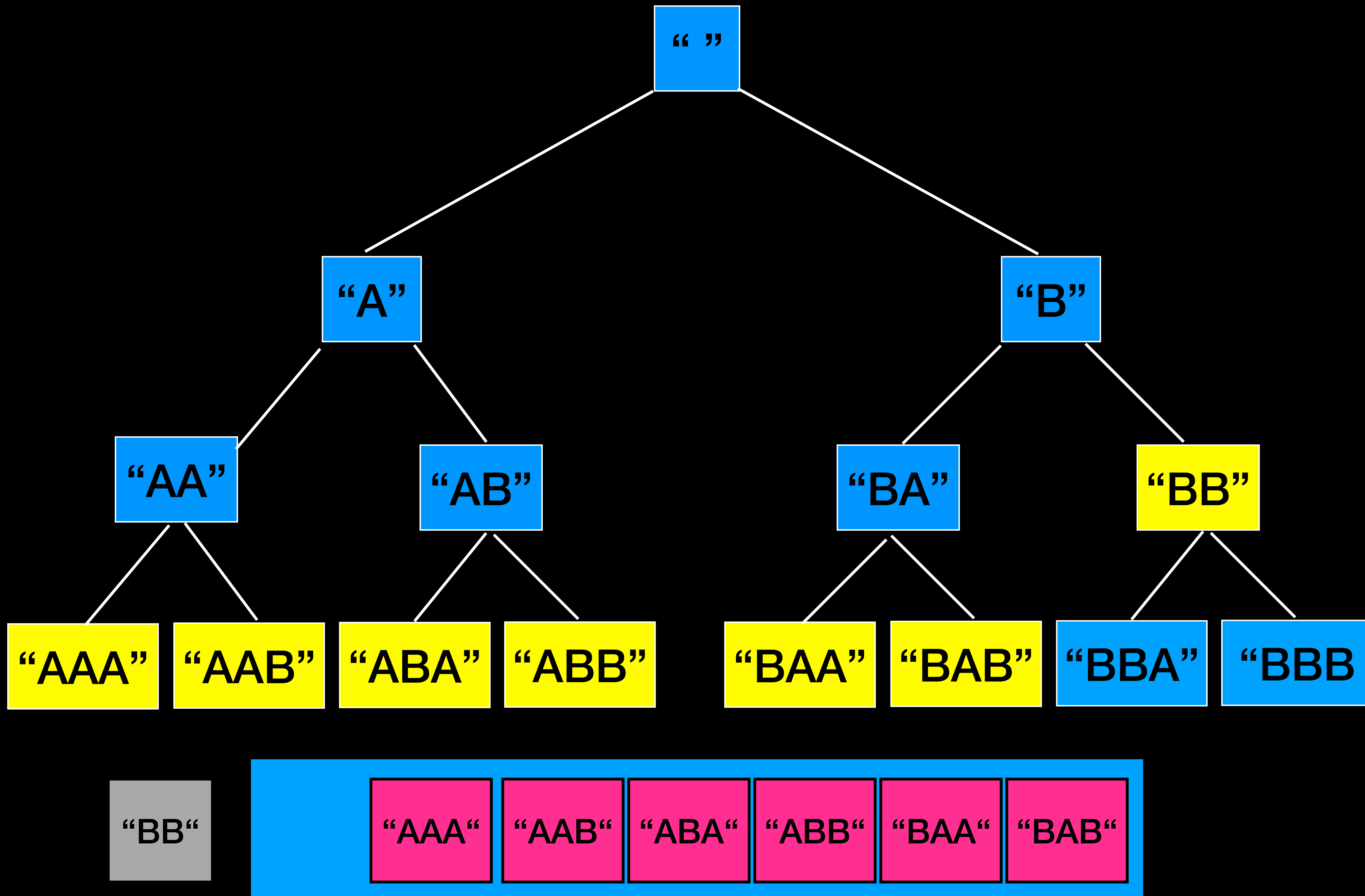
Let n = 3, alphabet still {‘A’,’B’}



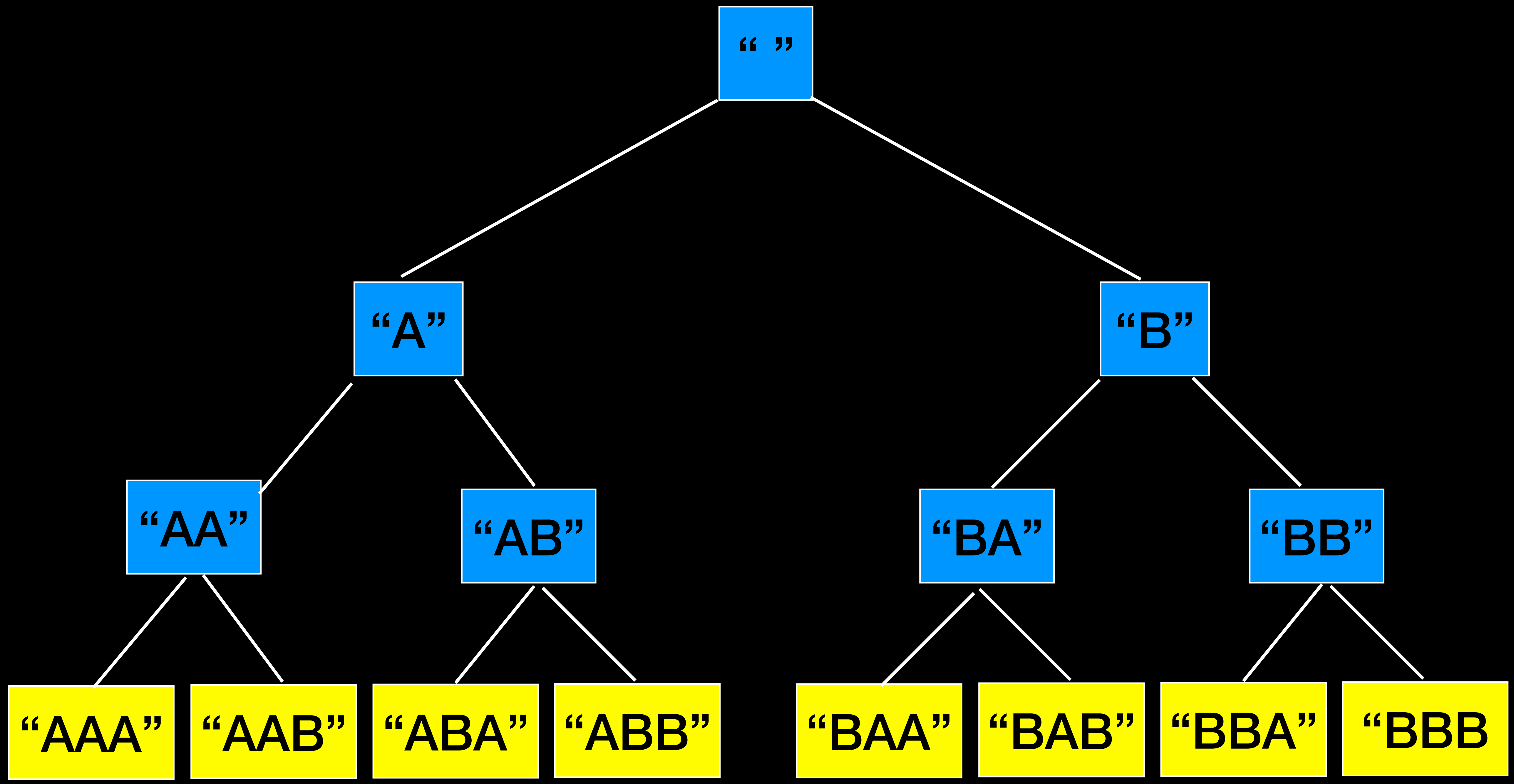
Let n = 3, alphabet still {‘A’,’B’}

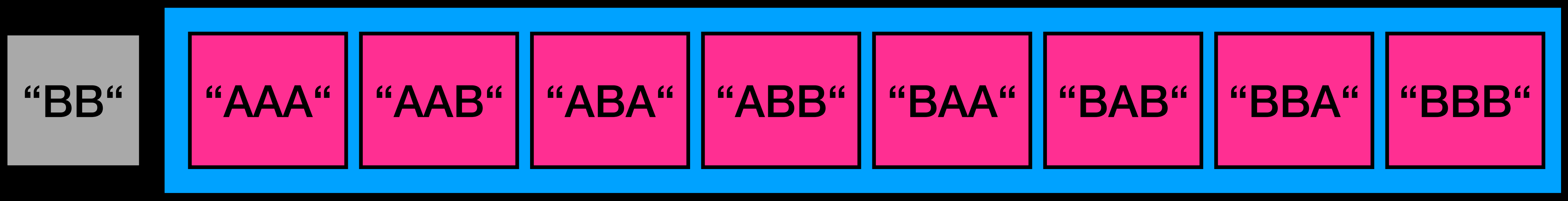


Let n = 3, alphabet still {‘A’,’B’}

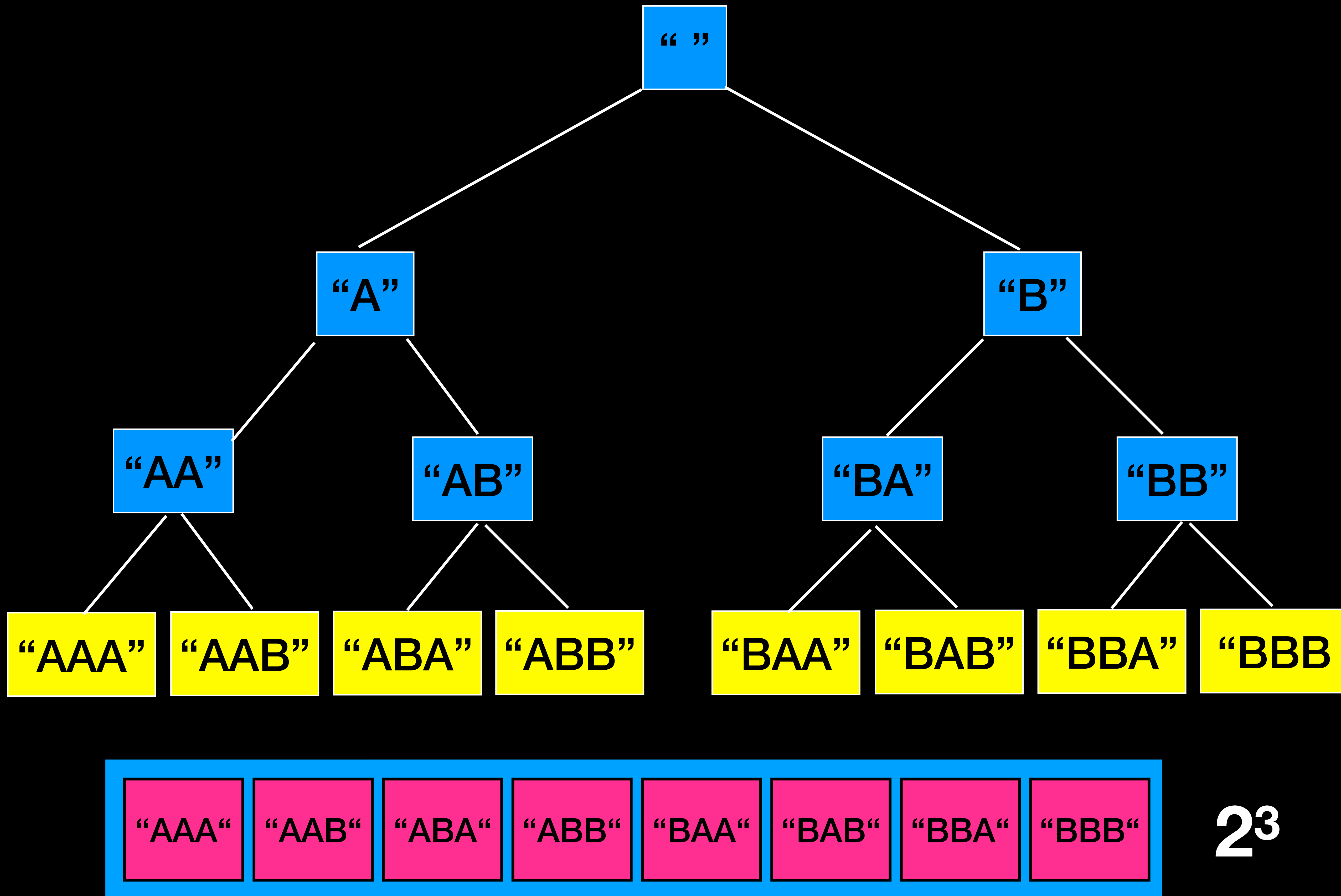


Let n = 3, alphabet still {‘A’,’B’}





Let n = 3, alphabet still {‘A’,’B’}



2^3

Memory Usage

With alphabet {'A', 'B', …, 'Z'}, 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

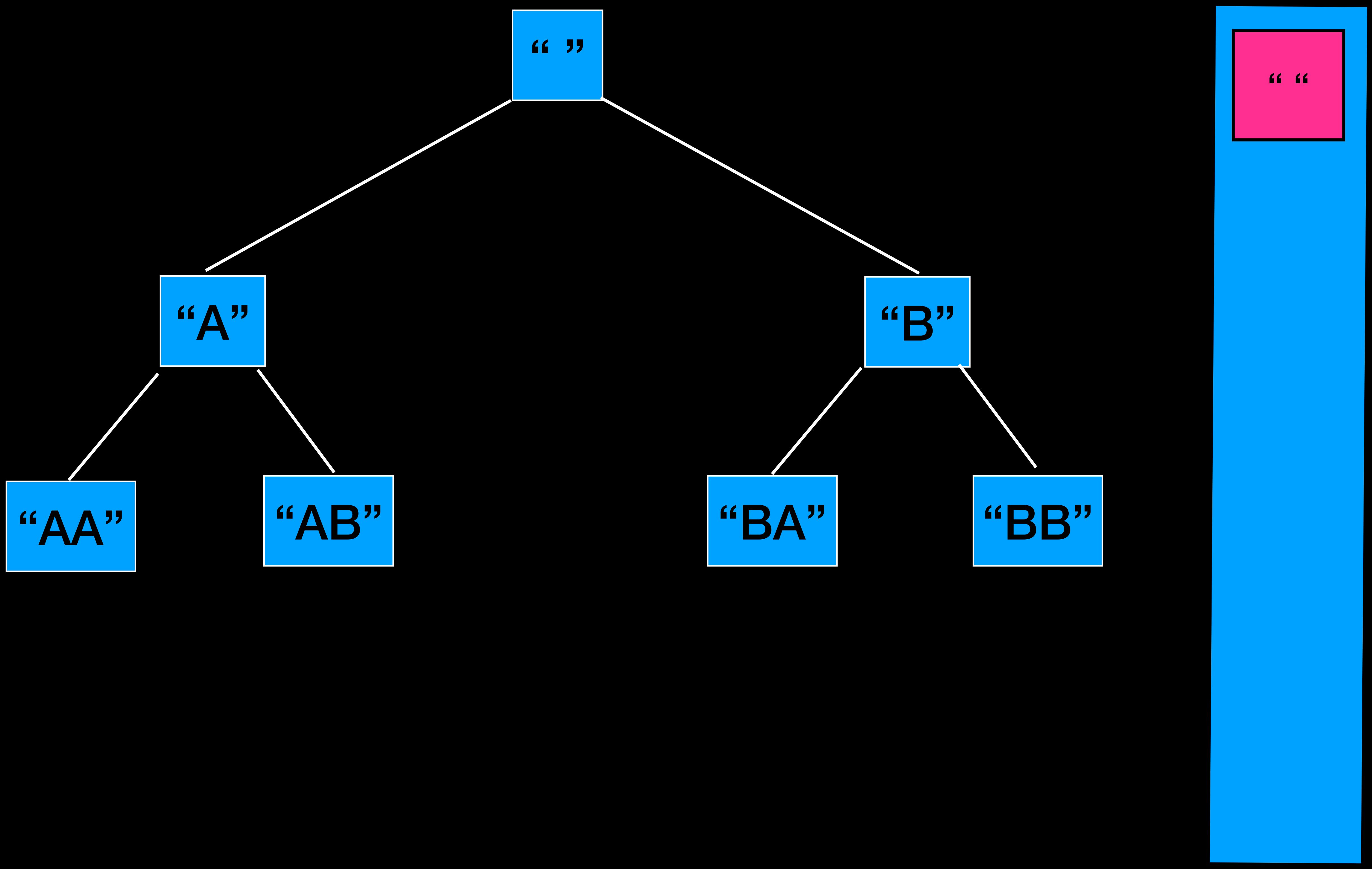
For n = 8 ≈ 5TB

Massive space requirement

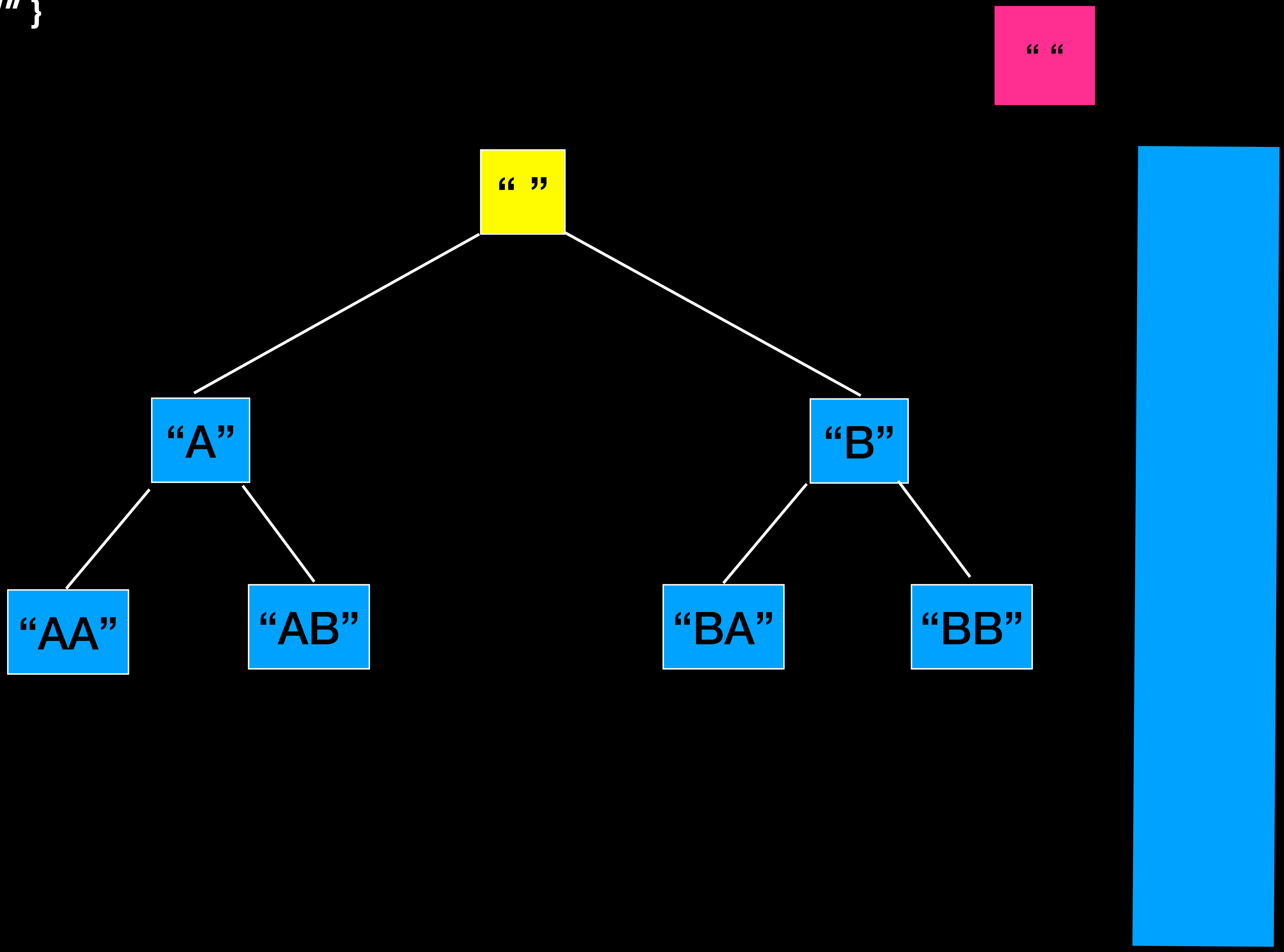
What if we use a stack?

```
findAllSubstrings(int n)
{
    push empty string on the stack

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

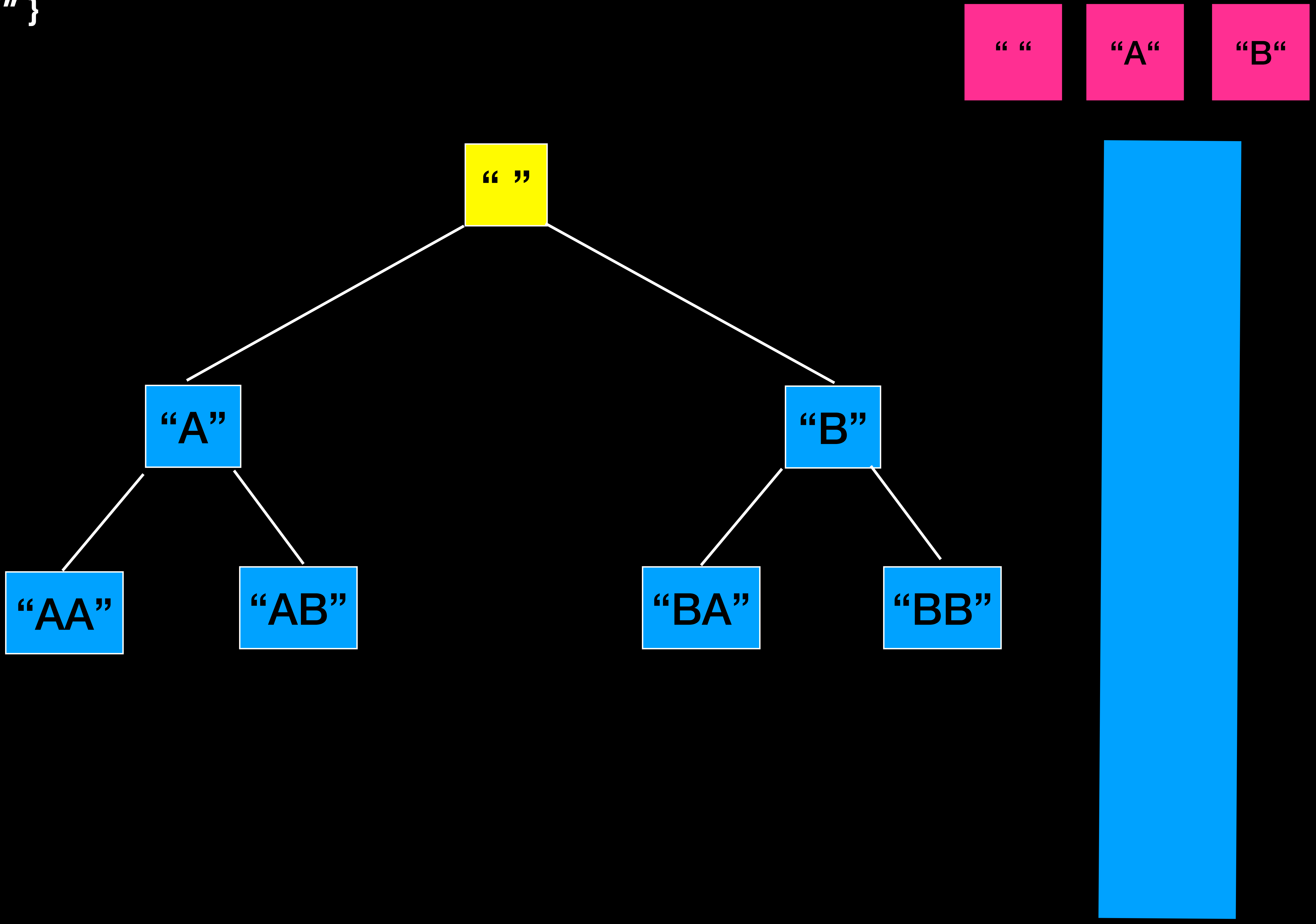
$O(26^n)$


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“ “

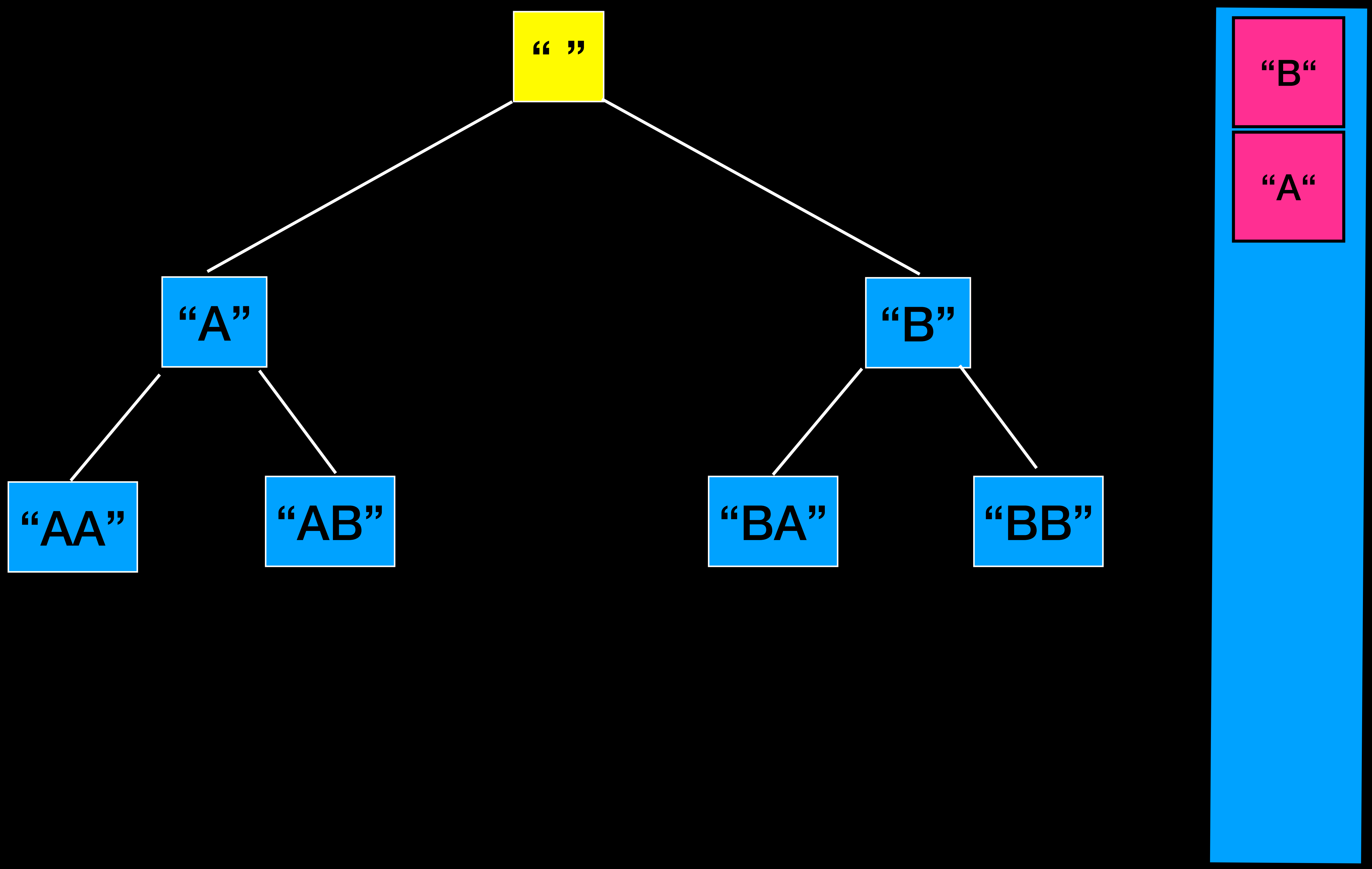
{ “” }


“ “
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

{ “” }

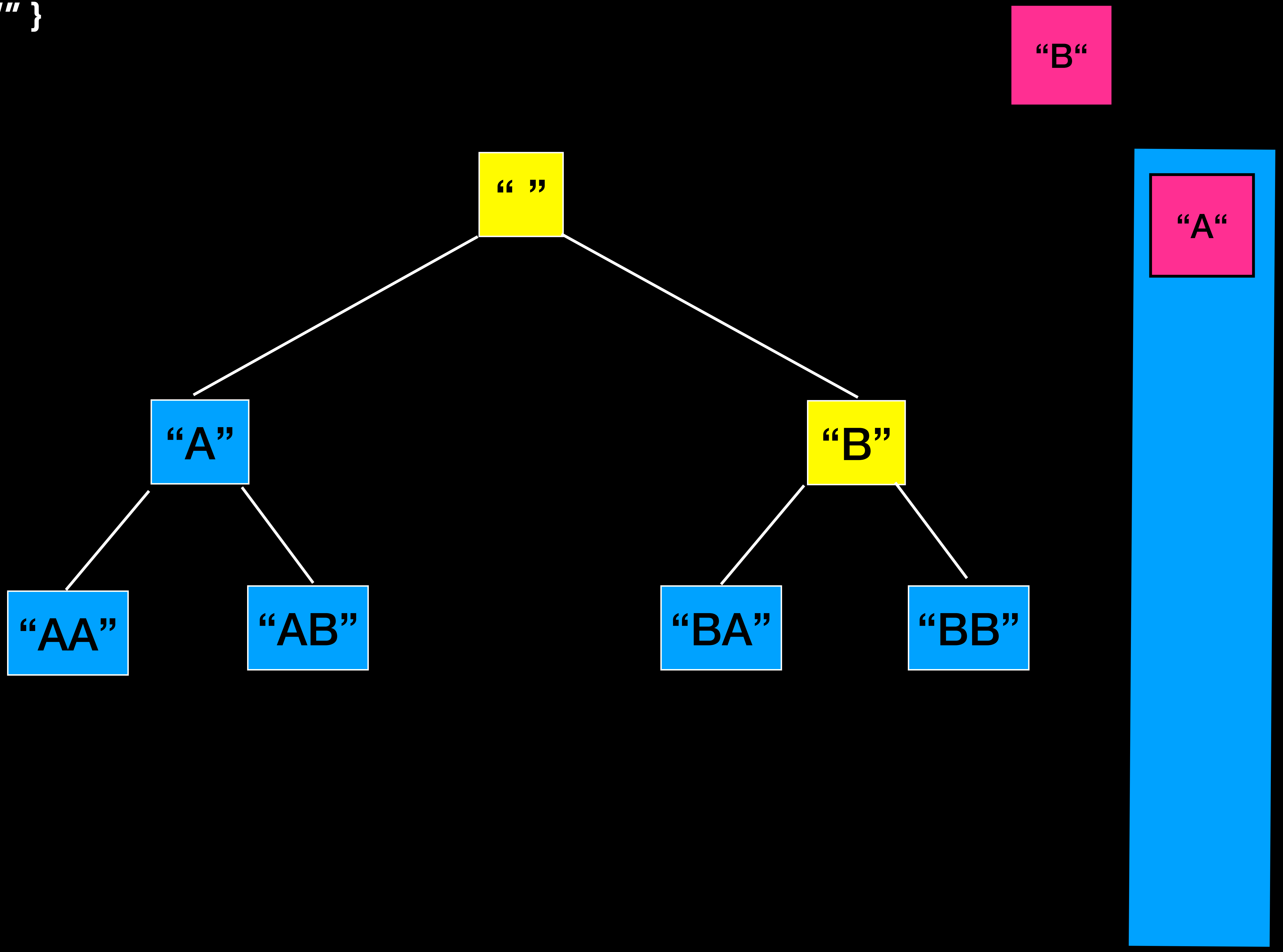

“ “
“A“
“B“
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”

{ "" }



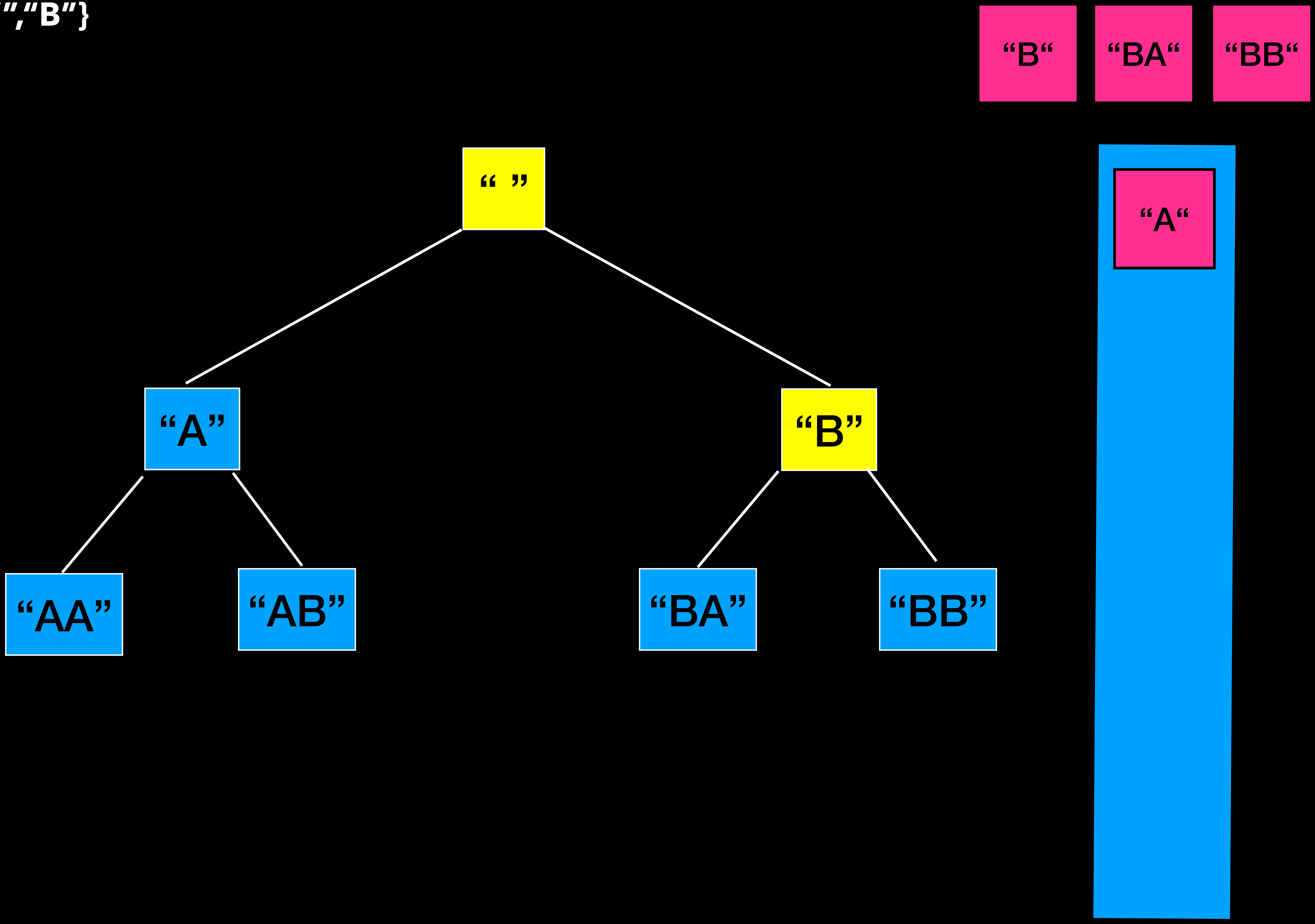
“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“B“
“A“

{ "" }

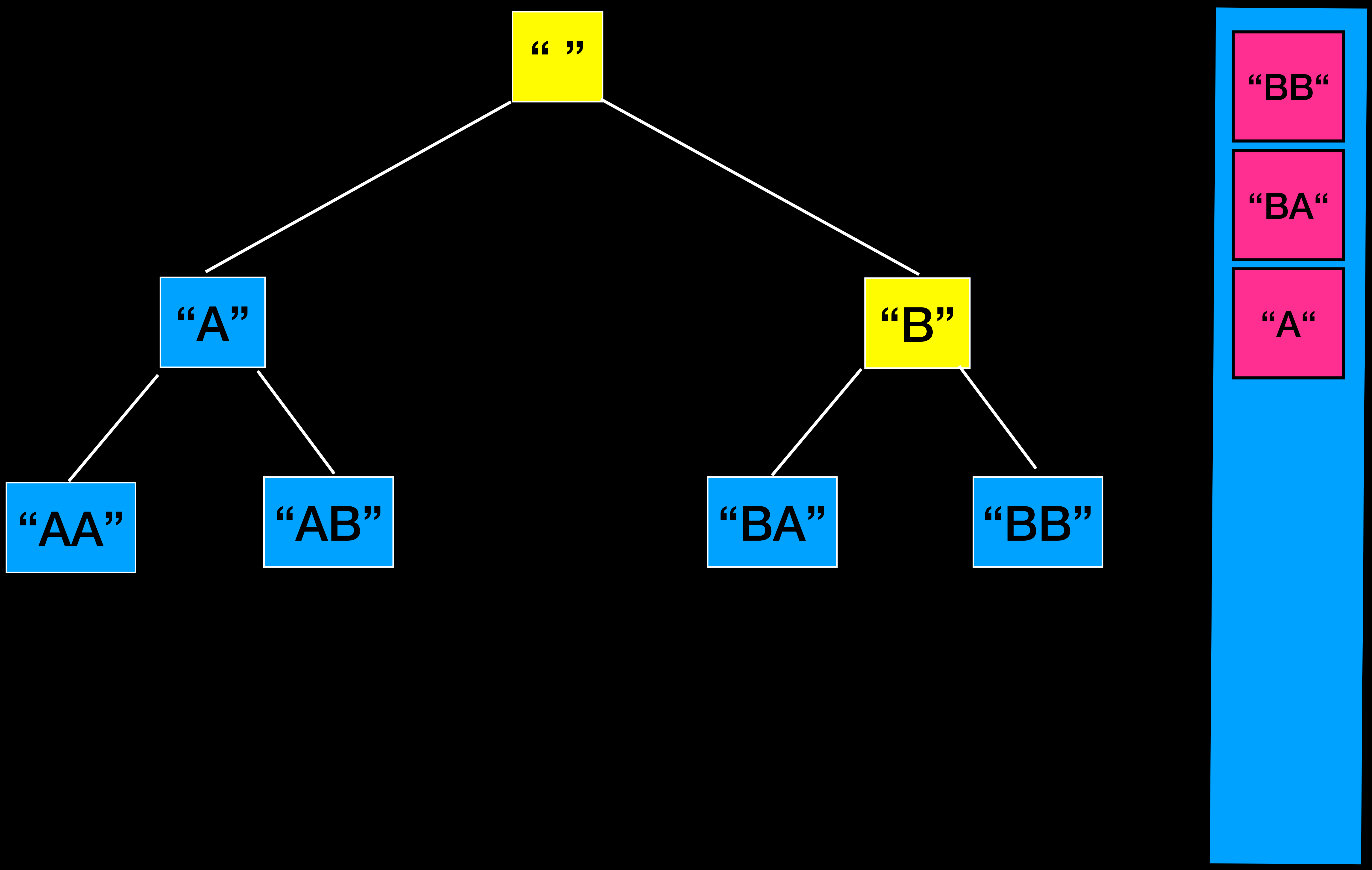


“B“
“ ”
“A“
“A”
“B”
“AA”
“AB”
“BA”
“BB”

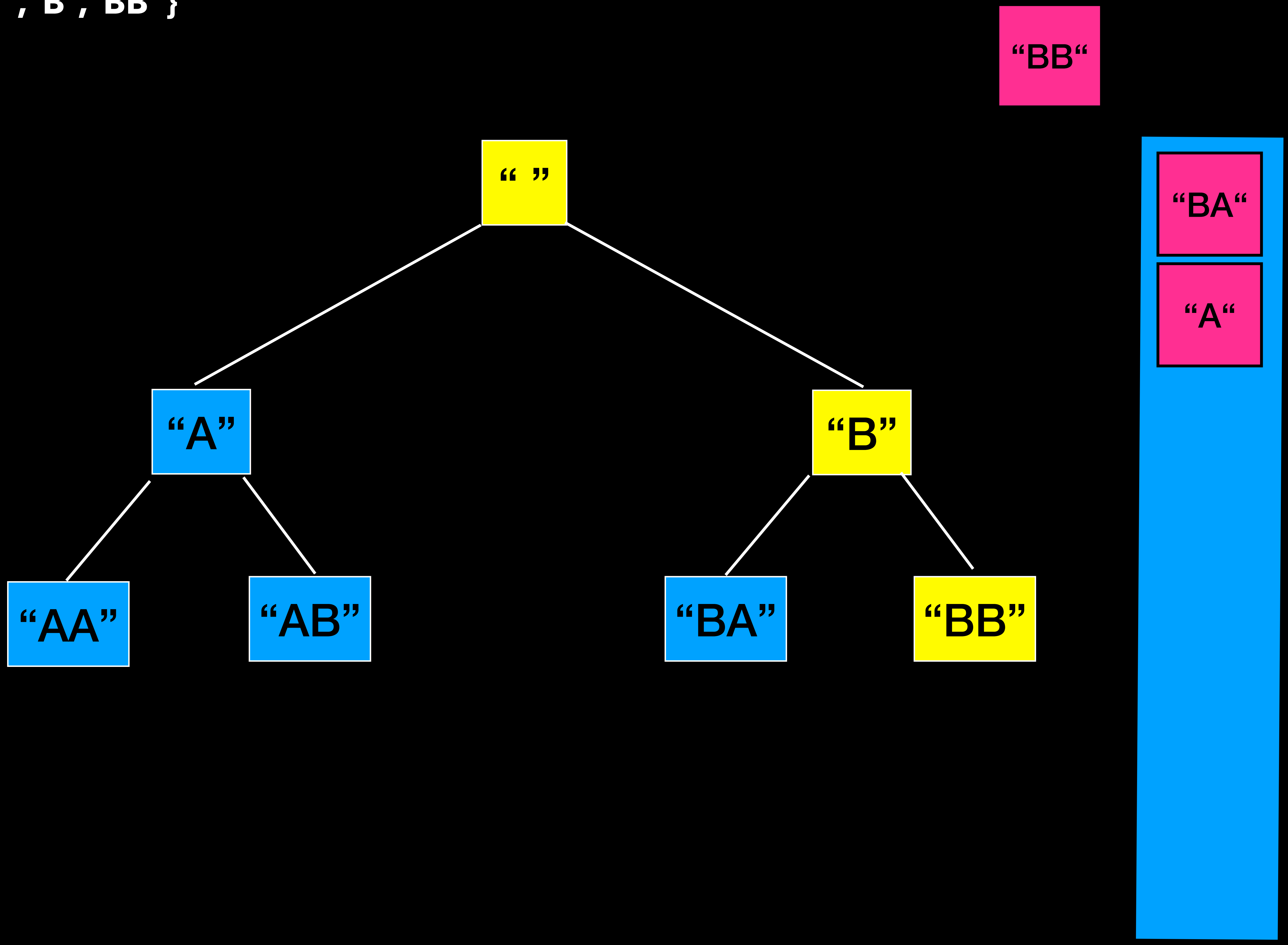
{ "","B"}



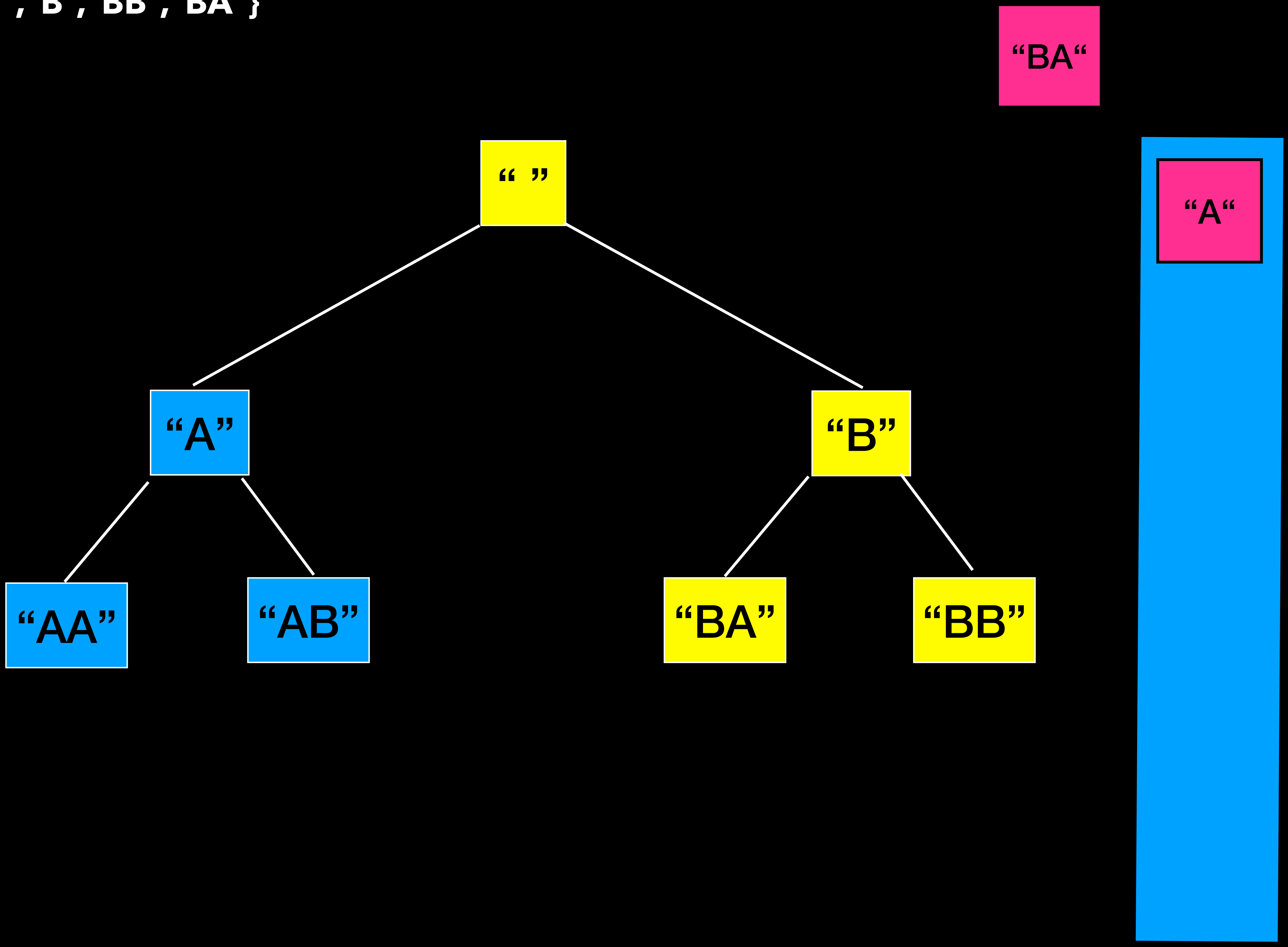
{ "","B"}



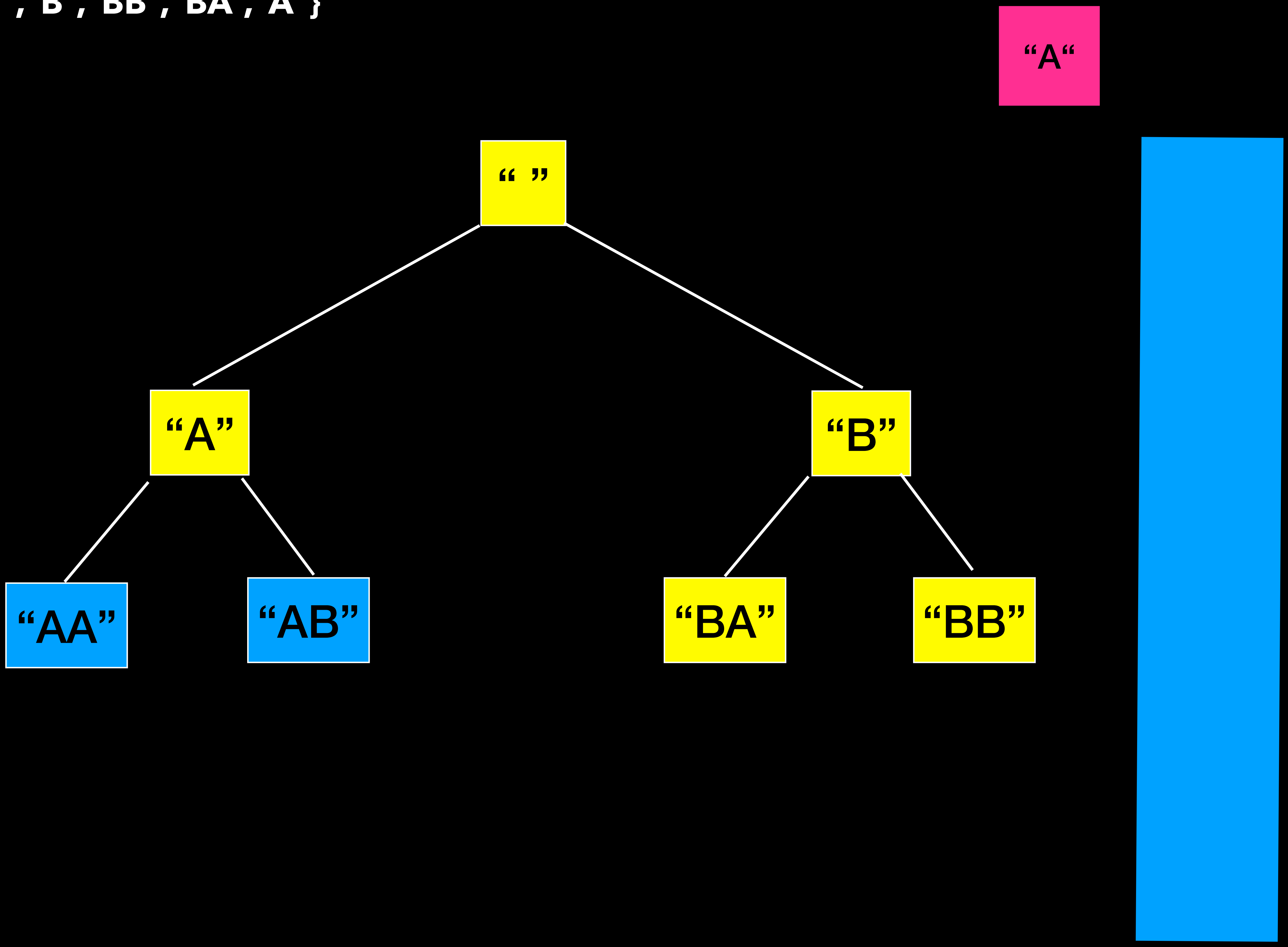
{ "","B","BB"}



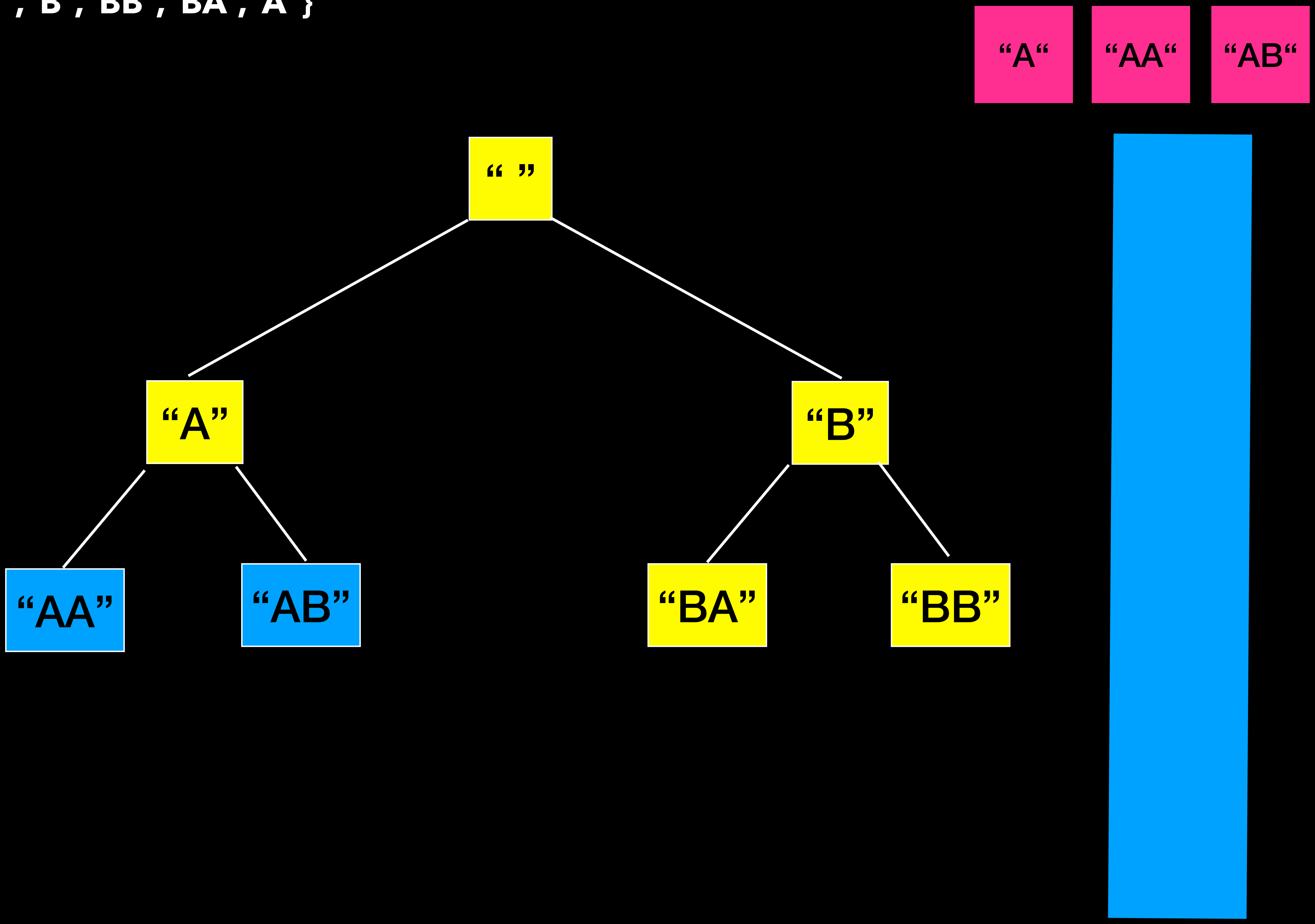
{ "","B","BB","BA"}



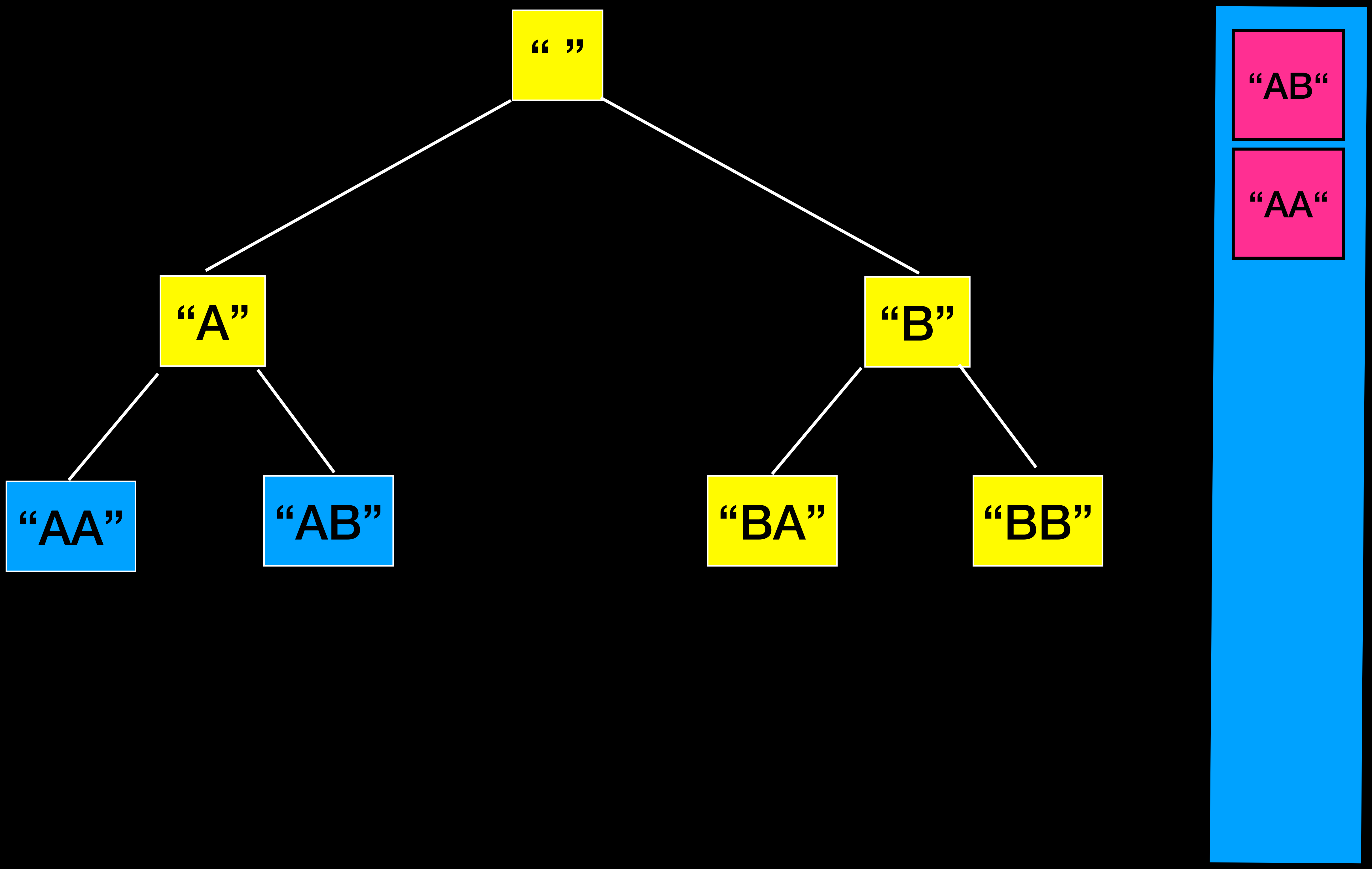
{ "","B","BB","BA","A"}



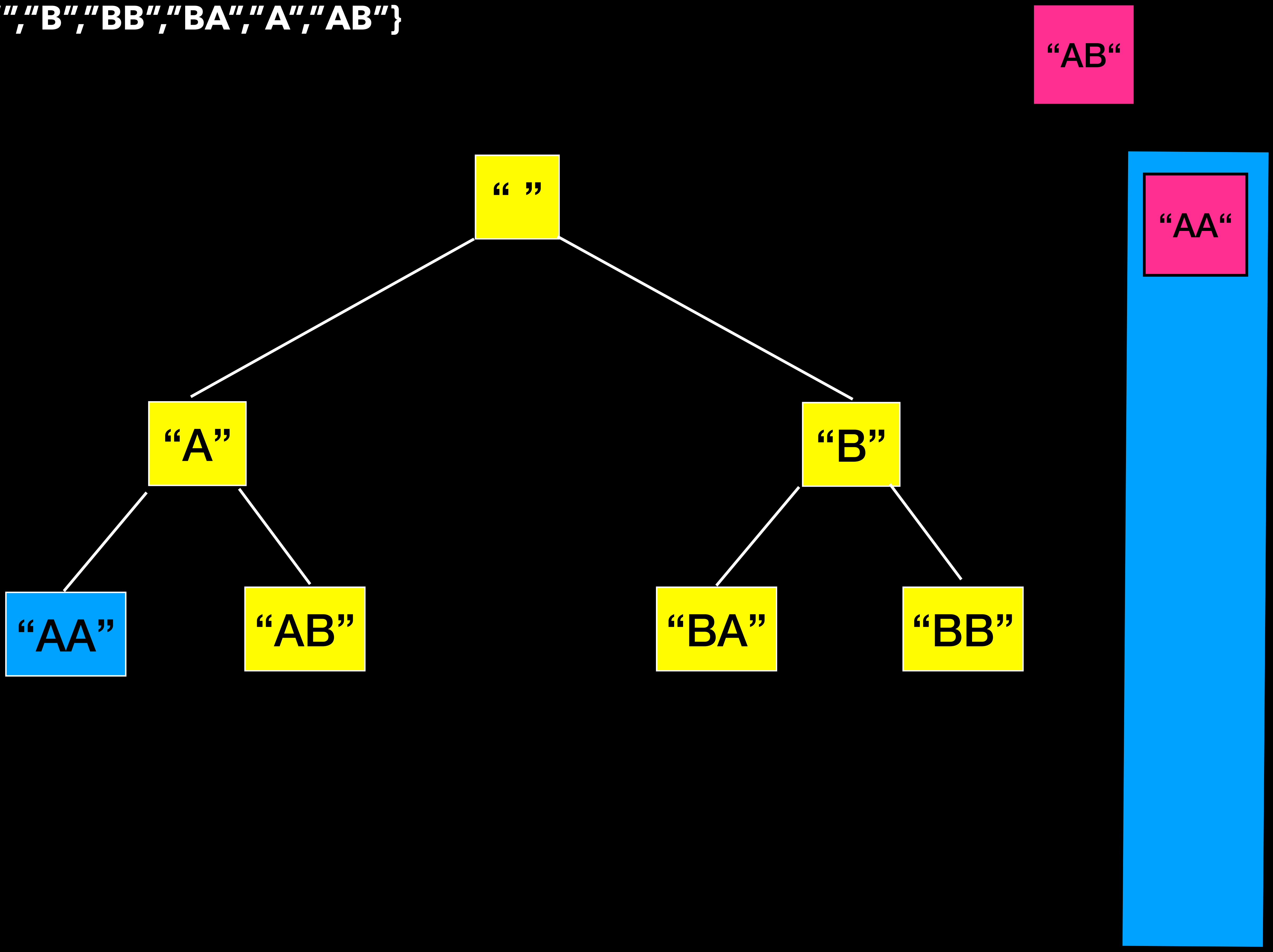
{ "","B","BB","BA","A"}



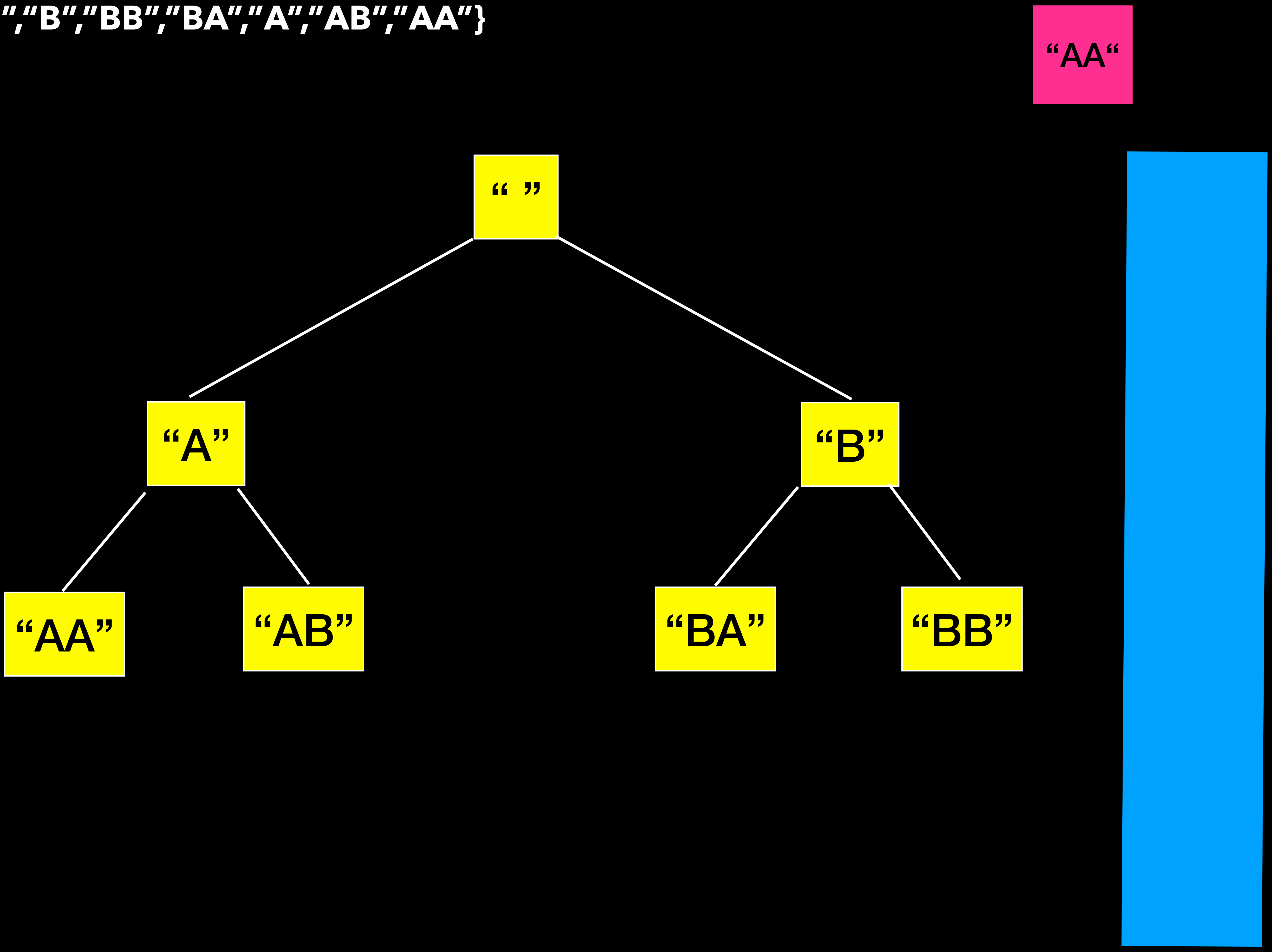
{ “”,”B”,”BB”,”BA”,”A”}



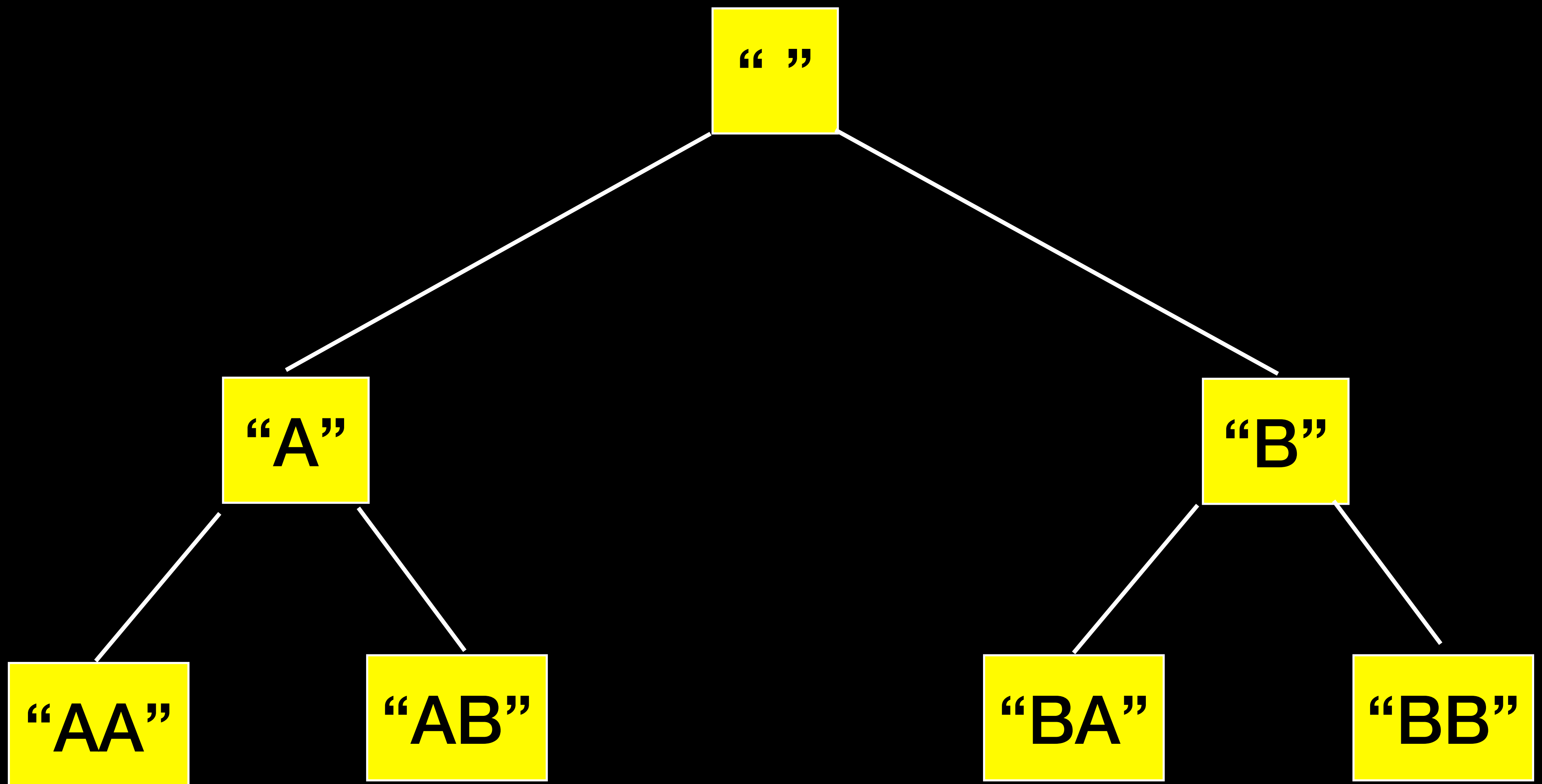
{ "","B","BB","BA","A","AB"}



{ “”,”B”,”BB”,”BA”,”A”,”AB”,”AA”}



{ "","B","BB","BA","A","AB","AA"}



What’s the difference?

Depth-First Search

Applications

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

Same worst-case runtime analysis

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" or "short" 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

Other ADTs

Deque

Double ended queue (deque)

Can add and remove to/from front and back

Deque

Double ended queue (deque)

Can add and remove to/from front and back

Deque

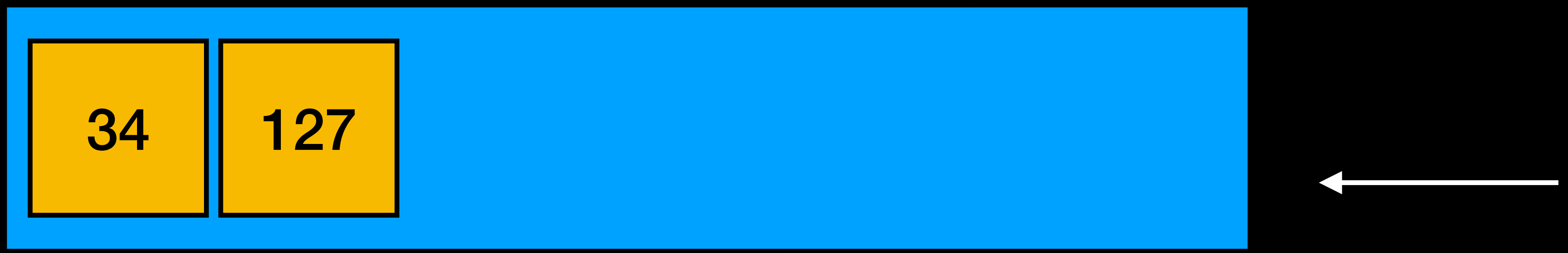
Double ended queue (deque)

Can add and remove to/from front and back

Deque

Double ended queue (deque)

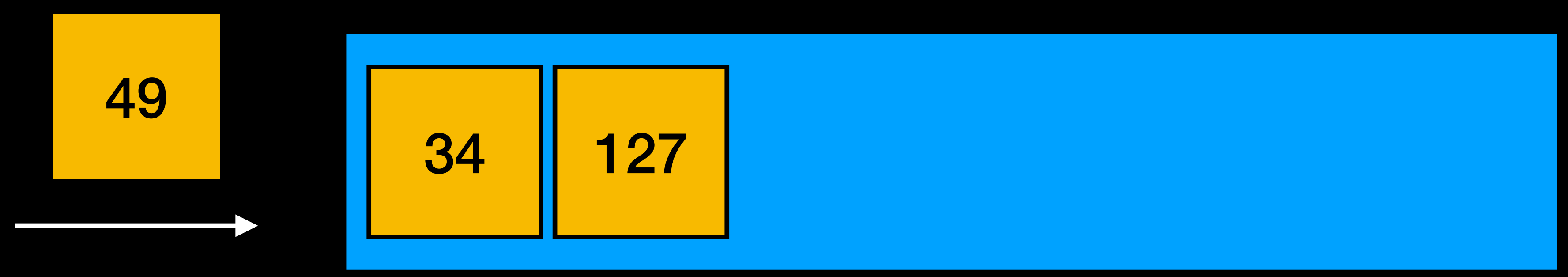
Can add and remove to/from front and back



Deque

Double ended queue (deque)

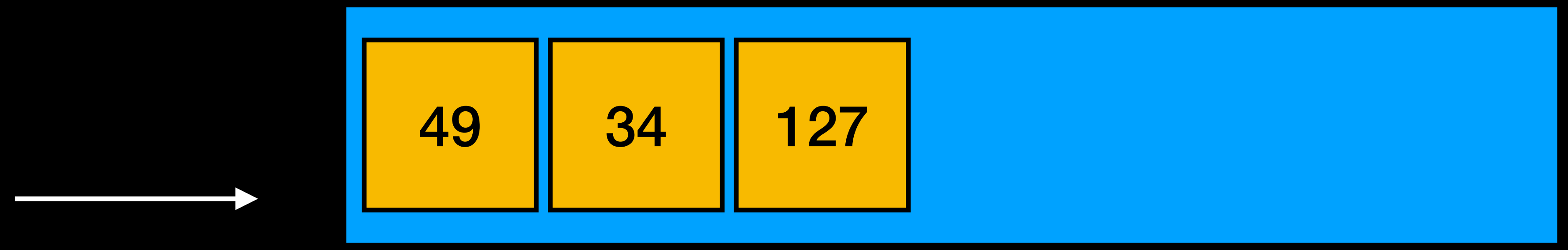
Can add and remove to/from front and back



Deque

Double ended queue (deque)

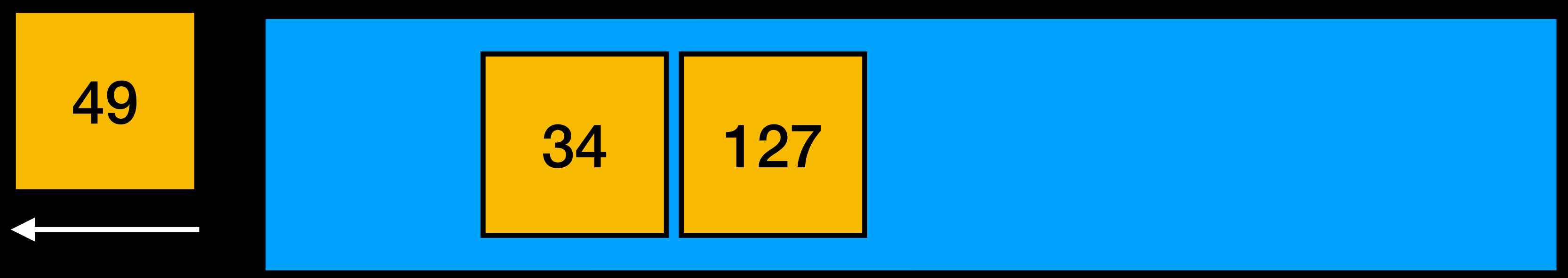
Can add and remove to/from front and back



Deque

Double ended queue (deque)

Can add and remove to/from front and back



Deque

Double ended queue (deque)

Can add and remove to/from front and back

Deque

Double ended queue (deque)

Can add and remove to/from front and back

Deque

In STL :

- does not use contiguous memory
- more complex to implement (keep track of memory blocks)
- grows more efficiently than vector

Deque

In STL :
- does not use contiguous memory
- more complex to implement (keep track of memory blocks)
- grows more efficiently than vector

In STL stack and queue are *adapters* of deck

Deque

In STL :
- does not use contiguous memory
- more complex to implement (keep track of memory blocks)
- grows more efficiently than vector

In STL stack and queue are *adapters* of deque

STL standardized the use of “pus” and “pop”, adapting with “push_back”, “push_front” etc. for all containers

Priority Queue

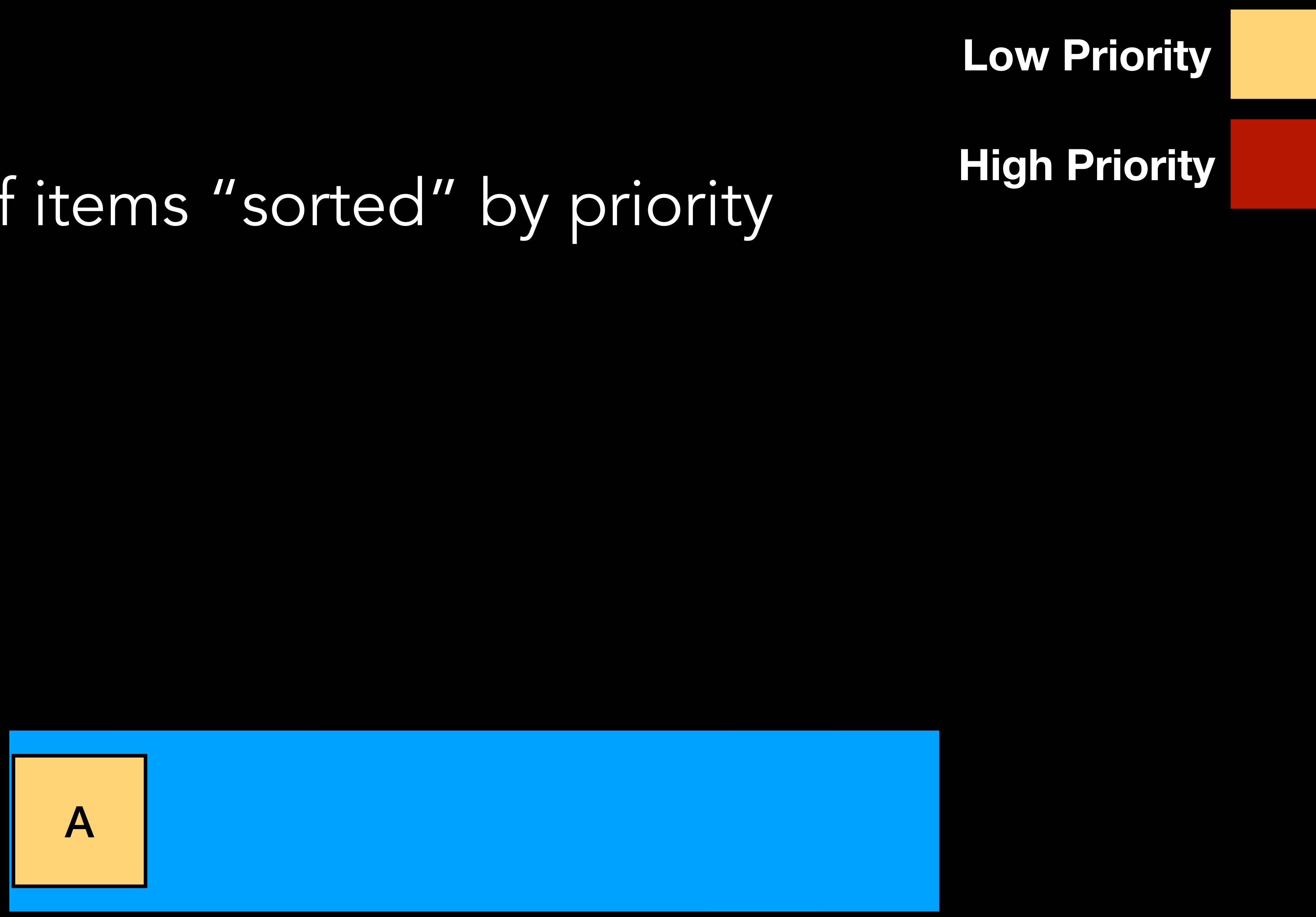
Low Priority

High Priority

A queue of items "sorted" by priority

Priority Queue

A queue of items "sorted" by priority



Priority Queue

Low Priority

High Priority

A queue of items "sorted" by priority

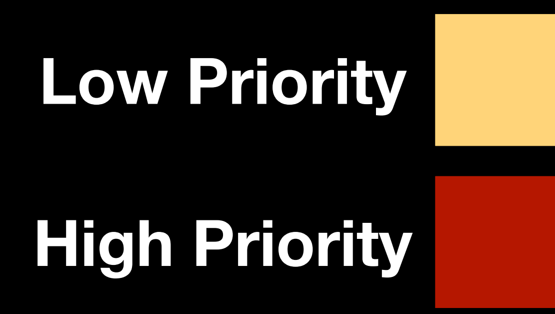
Priority Queue

Low Priority

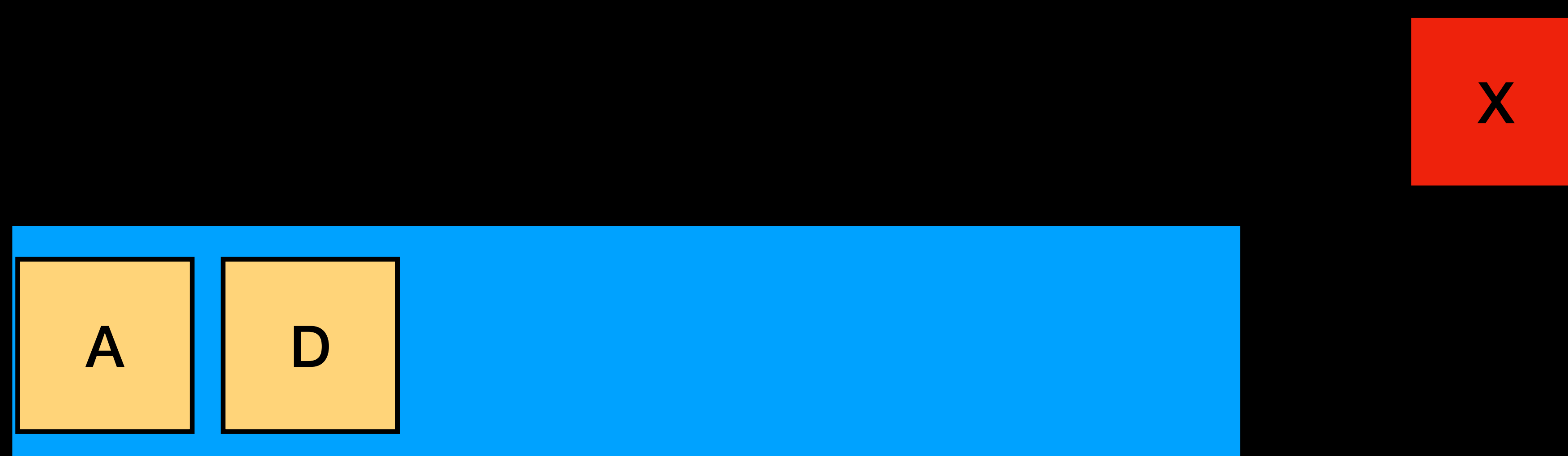
High Priority

A queue of items "sorted" by priority

Priority Queue

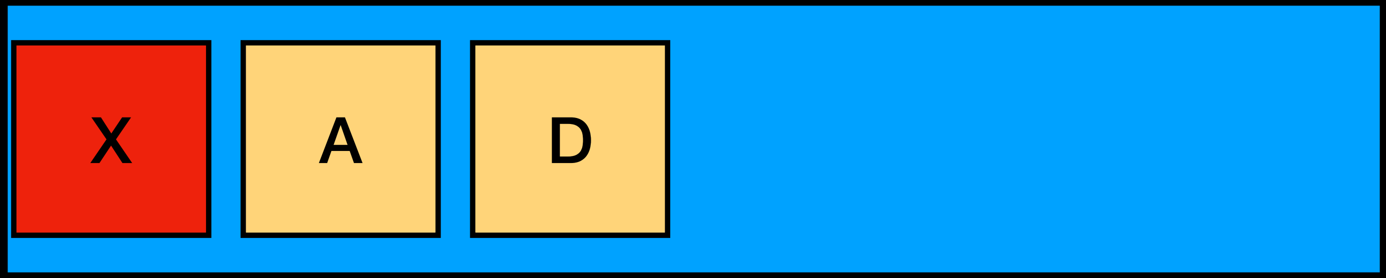


A queue of items “sorted” by priority



Priority Queue

A queue of items “sorted” by priority

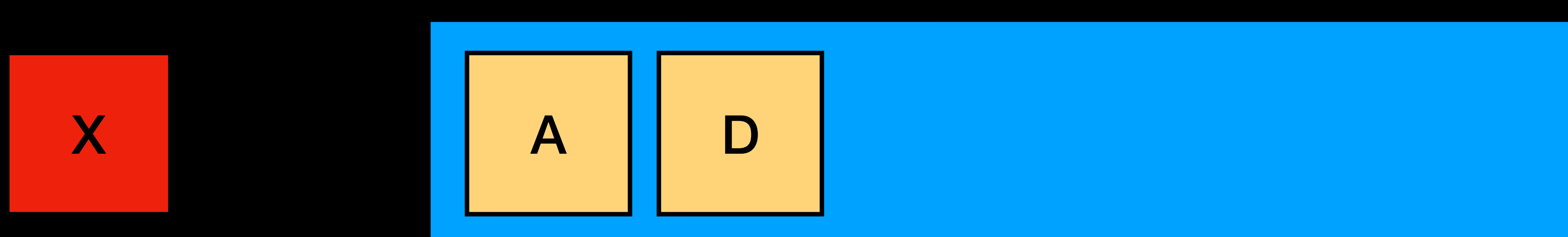


Priority Queue

Low Priority

High Priority

A queue of items “sorted” by priority



Priority Queue

Low Priority

High Priority

A queue of items “sorted” by priority

If value indicates priority, it amounts to a sorted list that accesses/removes the “highest” items first

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)

Priority Queue

Spoiler Alert!!!!

Often implemented with a Heap

Will tell you what it is in a few weeks… but here is another example of ADT vs data structure