Queue ADT

Tiziana Ligorio

tligorio@hunter.cuny.edu

Today’s Plan

Recap

Queue ADT

Applications

Announcements

Queue

A data structure representing a waiting line

Objects can be enqueued to the back of the line

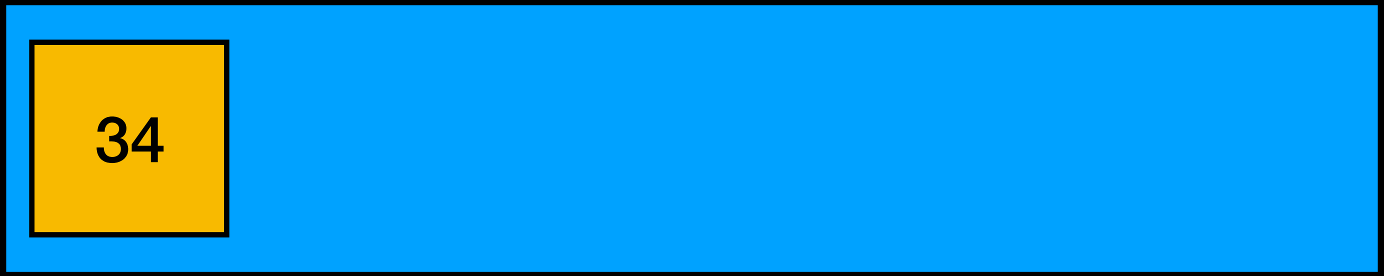
or dequeued from the front of the line

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

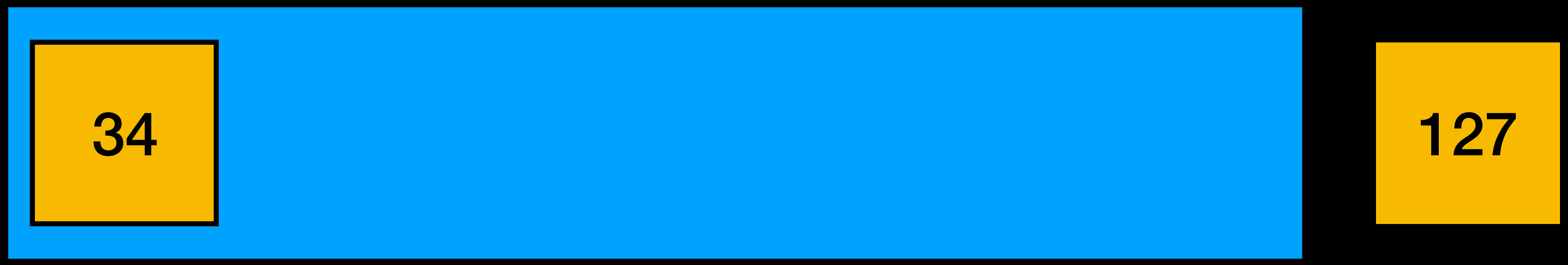


Queue

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

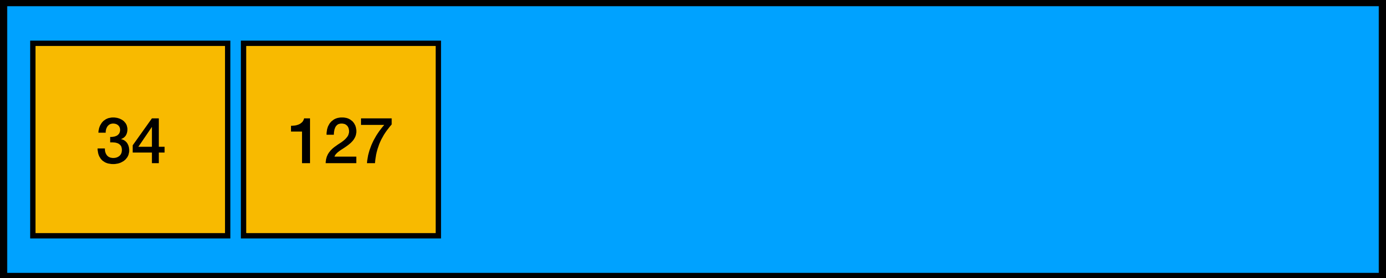


Queue

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

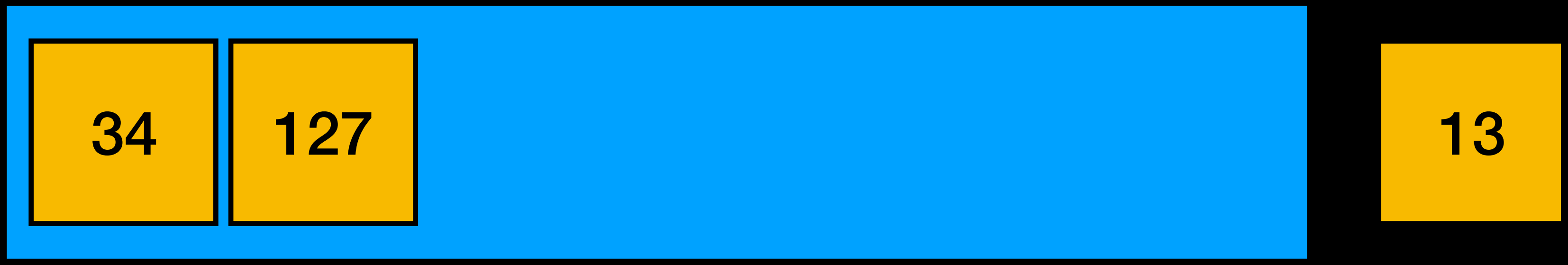


Queue

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

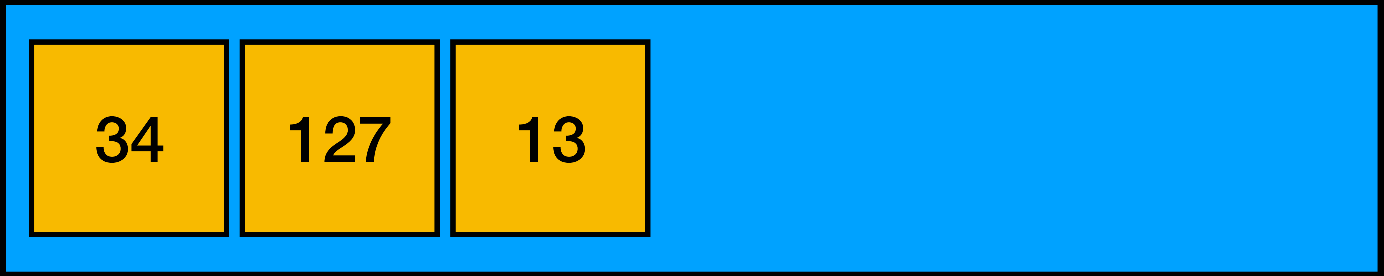


Queue

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

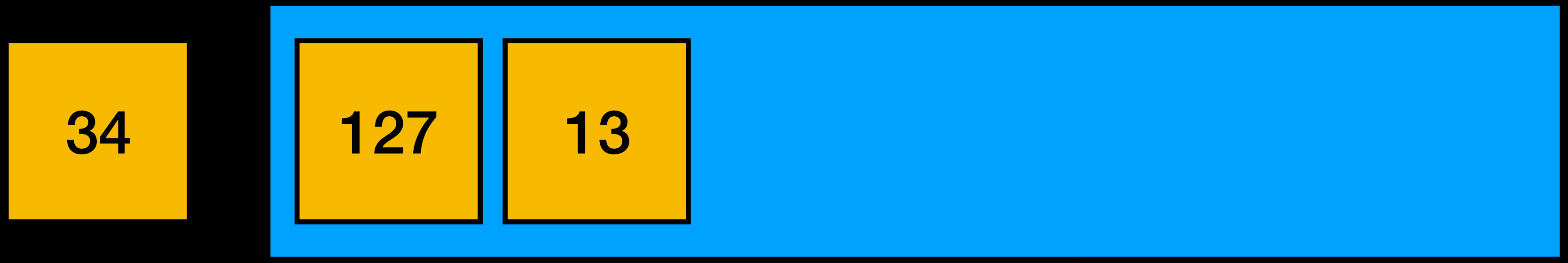


Queue

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

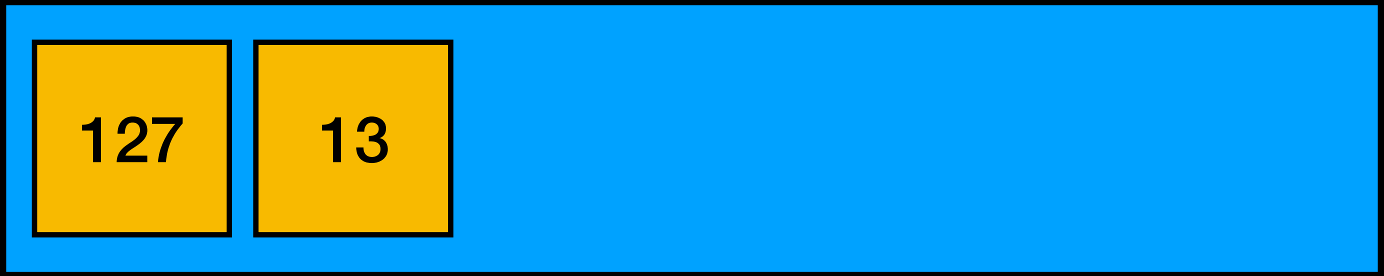


Queue

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

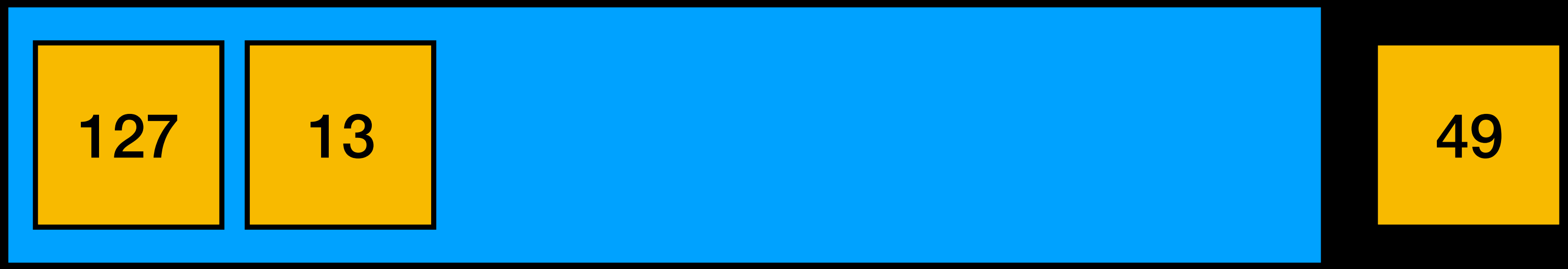


Queue

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

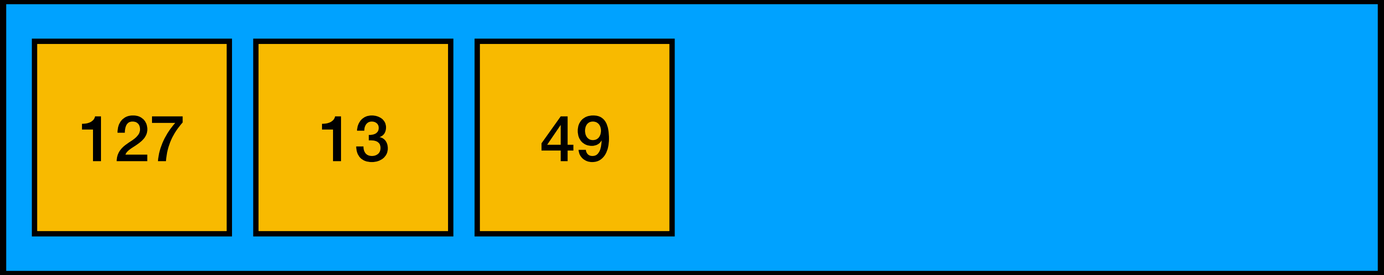


Queue

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

Recognizing Palindromes

Any waiting queue

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

. . .

Generating all substrings

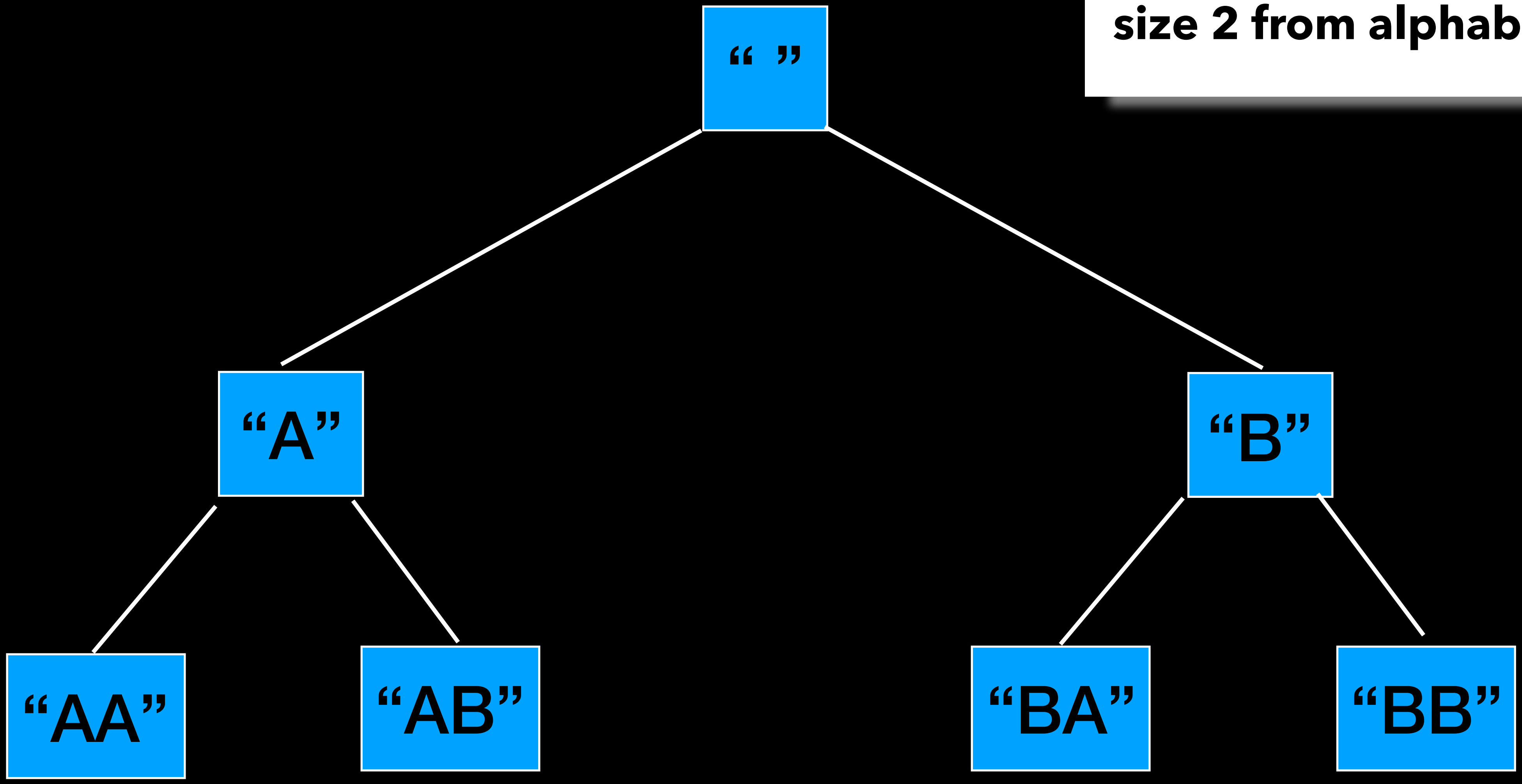
Generate all possible strings **up to** some fixed length **n with repetition (same character included multiple times)**

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'}



" "

"A"

"B"

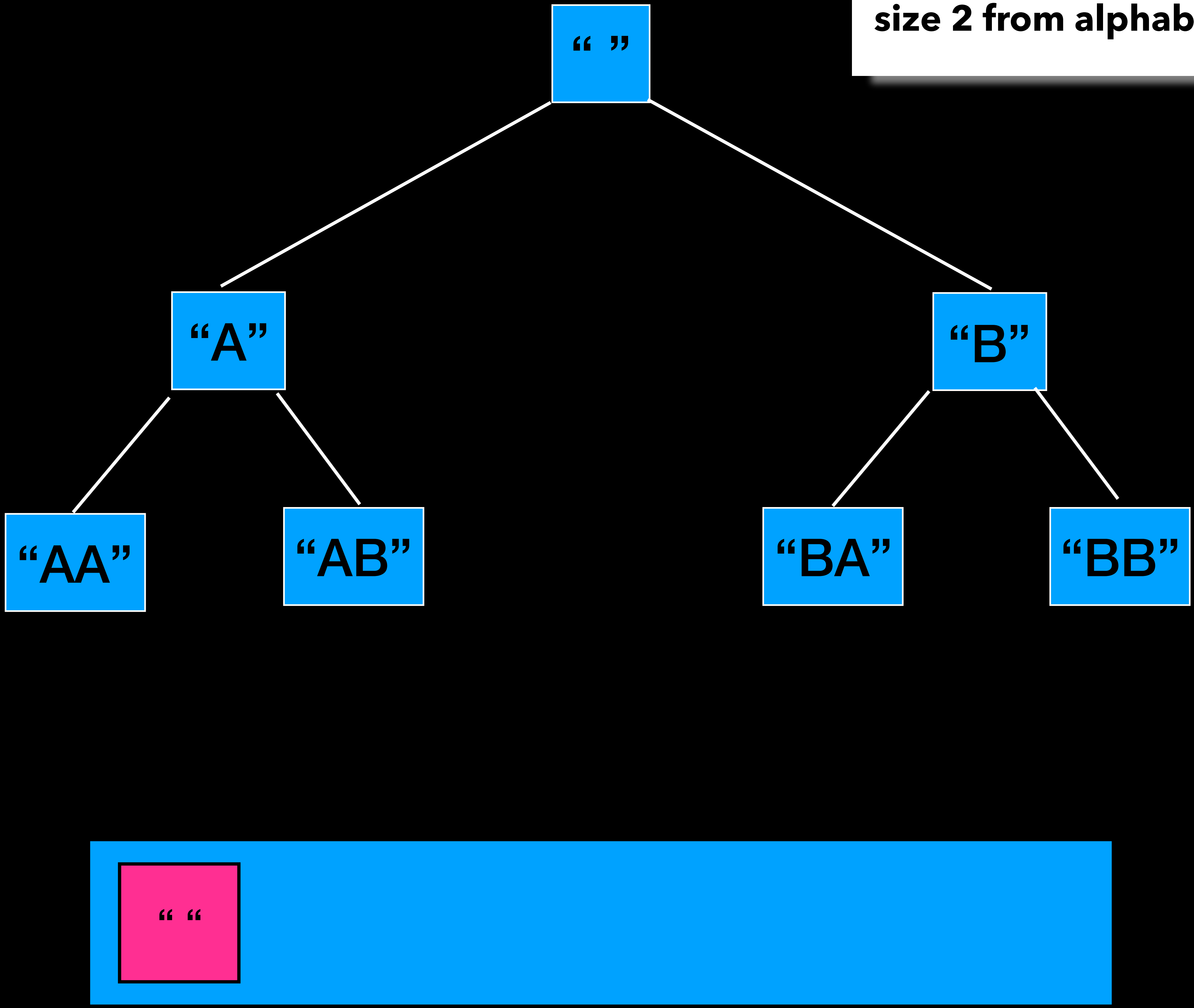
"AA"

"AB"

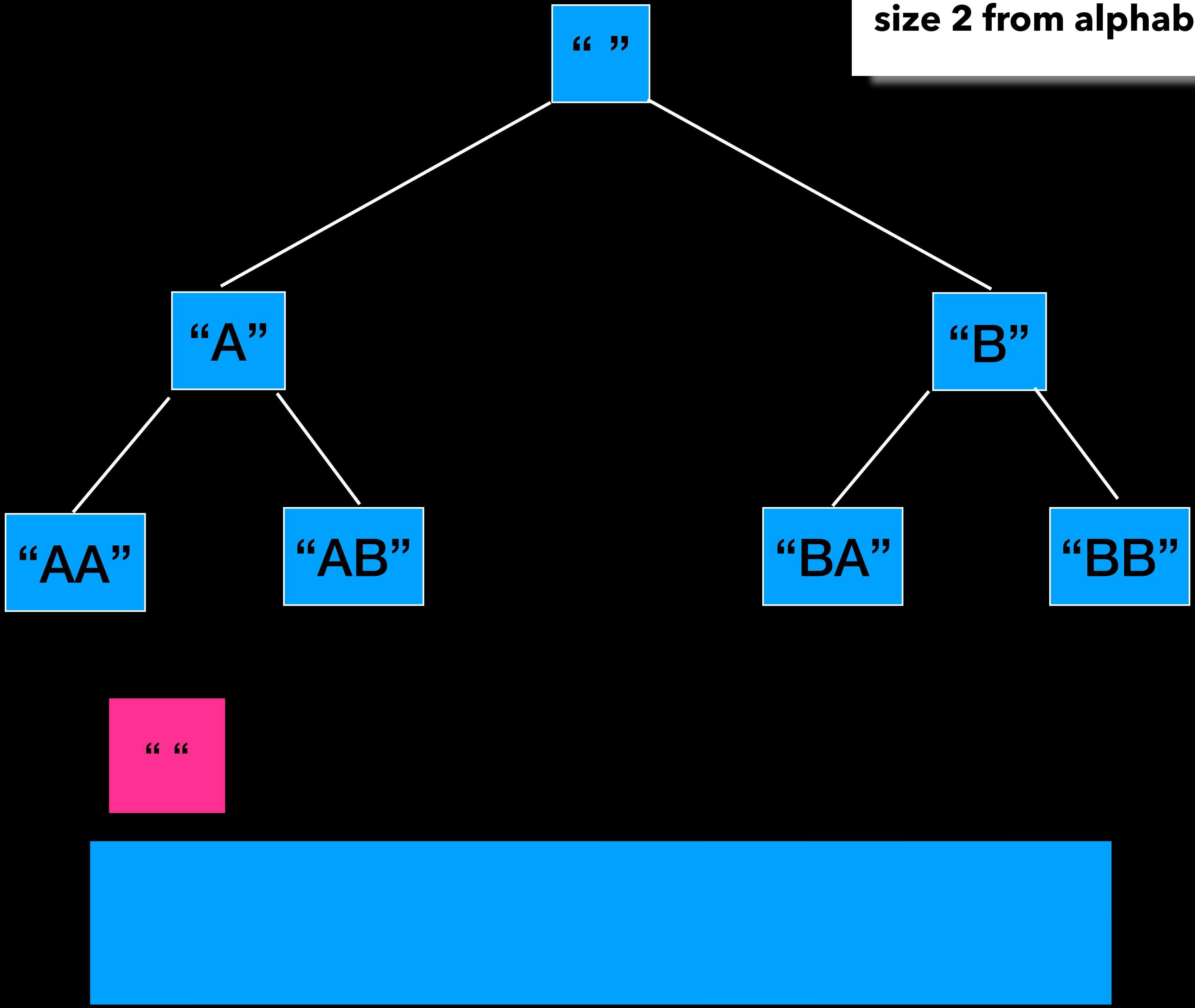
"BA"

"BB"

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

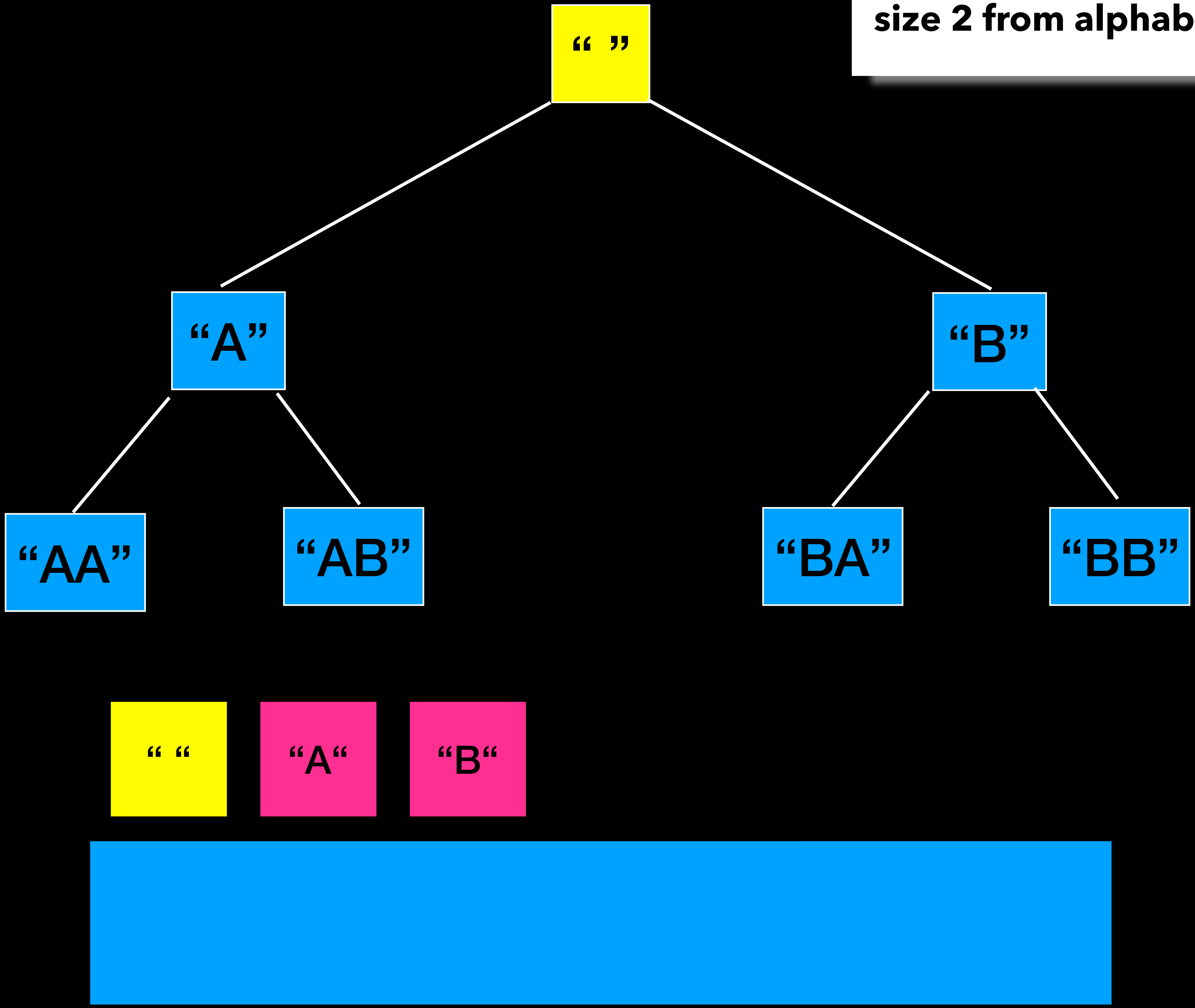


Generate all substrings of size 2 from alphabet {'A', 'B'}



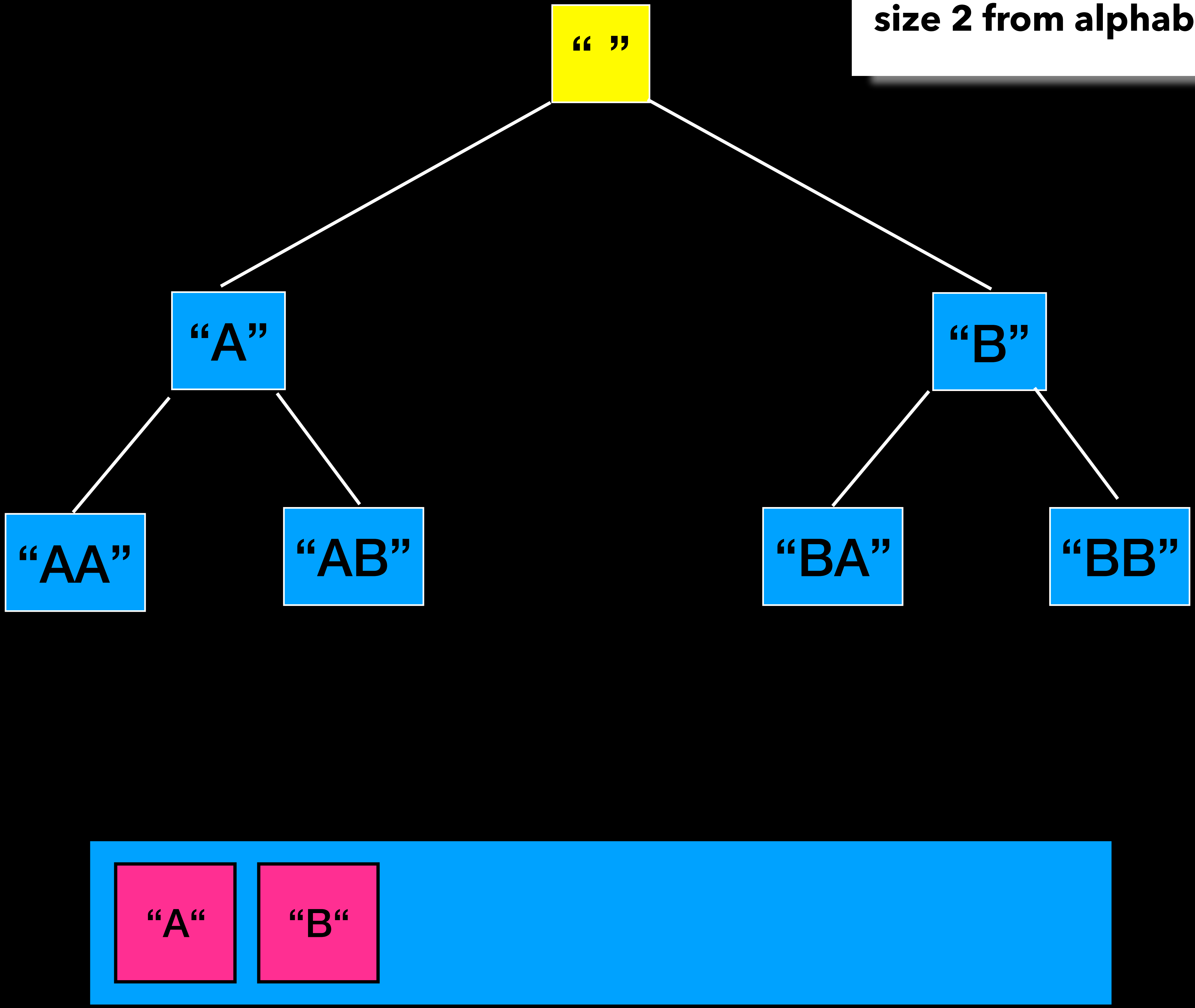
{ “”}

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



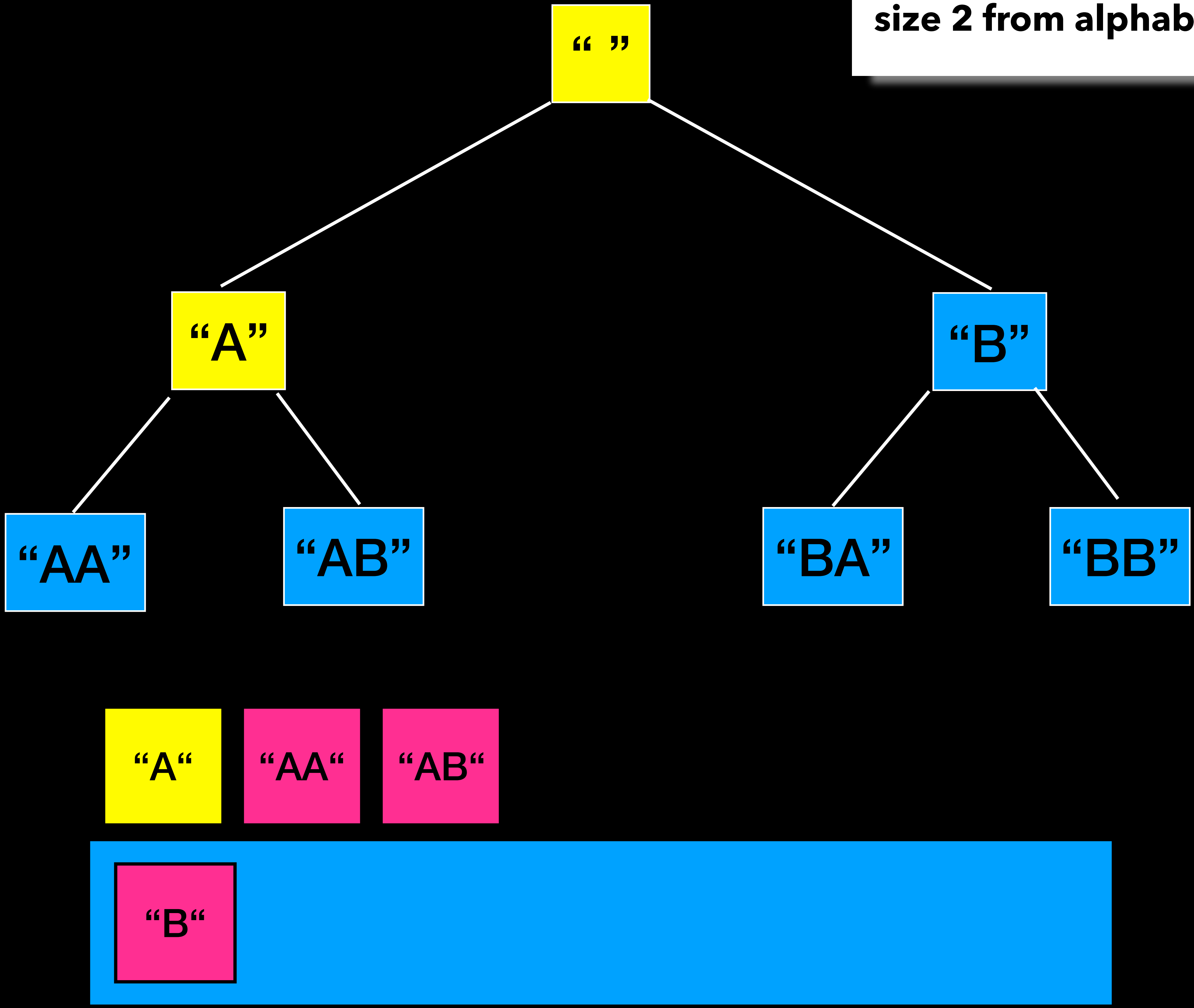
{ “”}

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



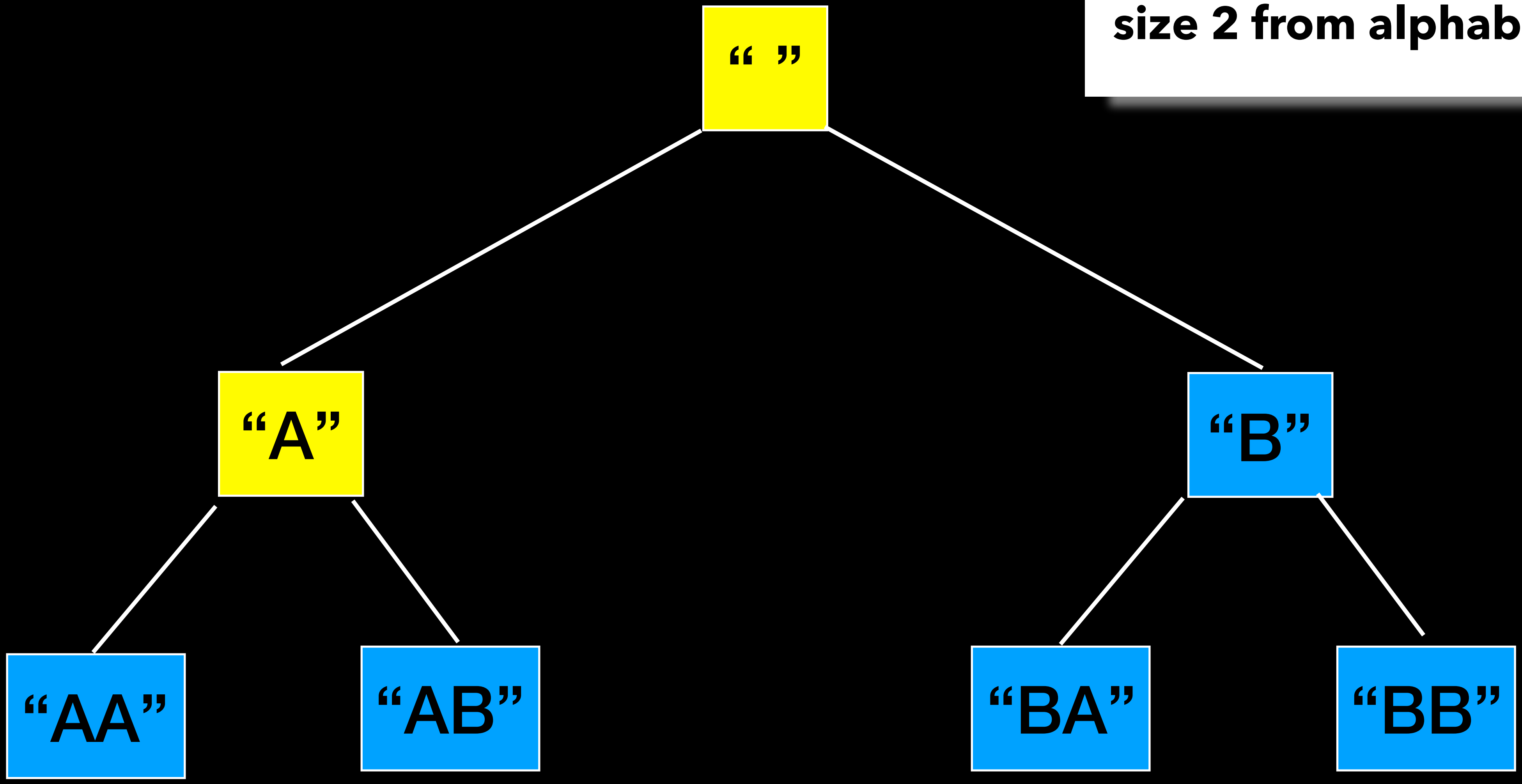
{ "", "A"}

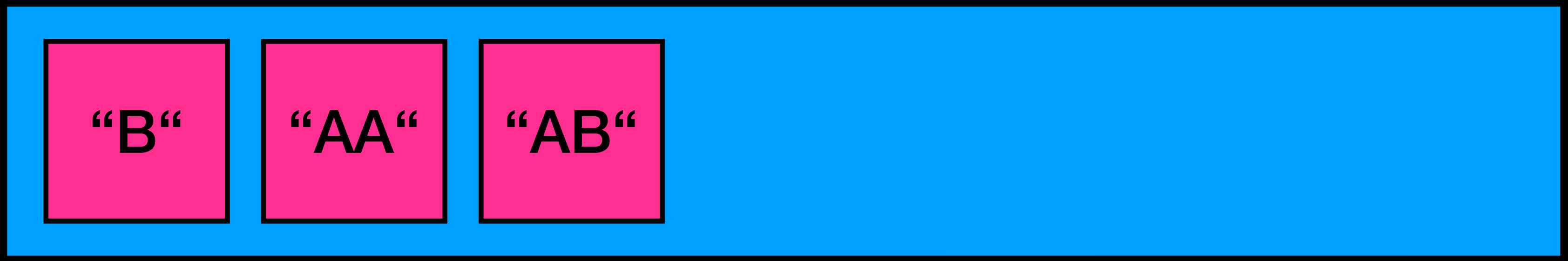
Generate all substrings of size 2 from alphabet {'A', 'B'}



{ “”, “A”}

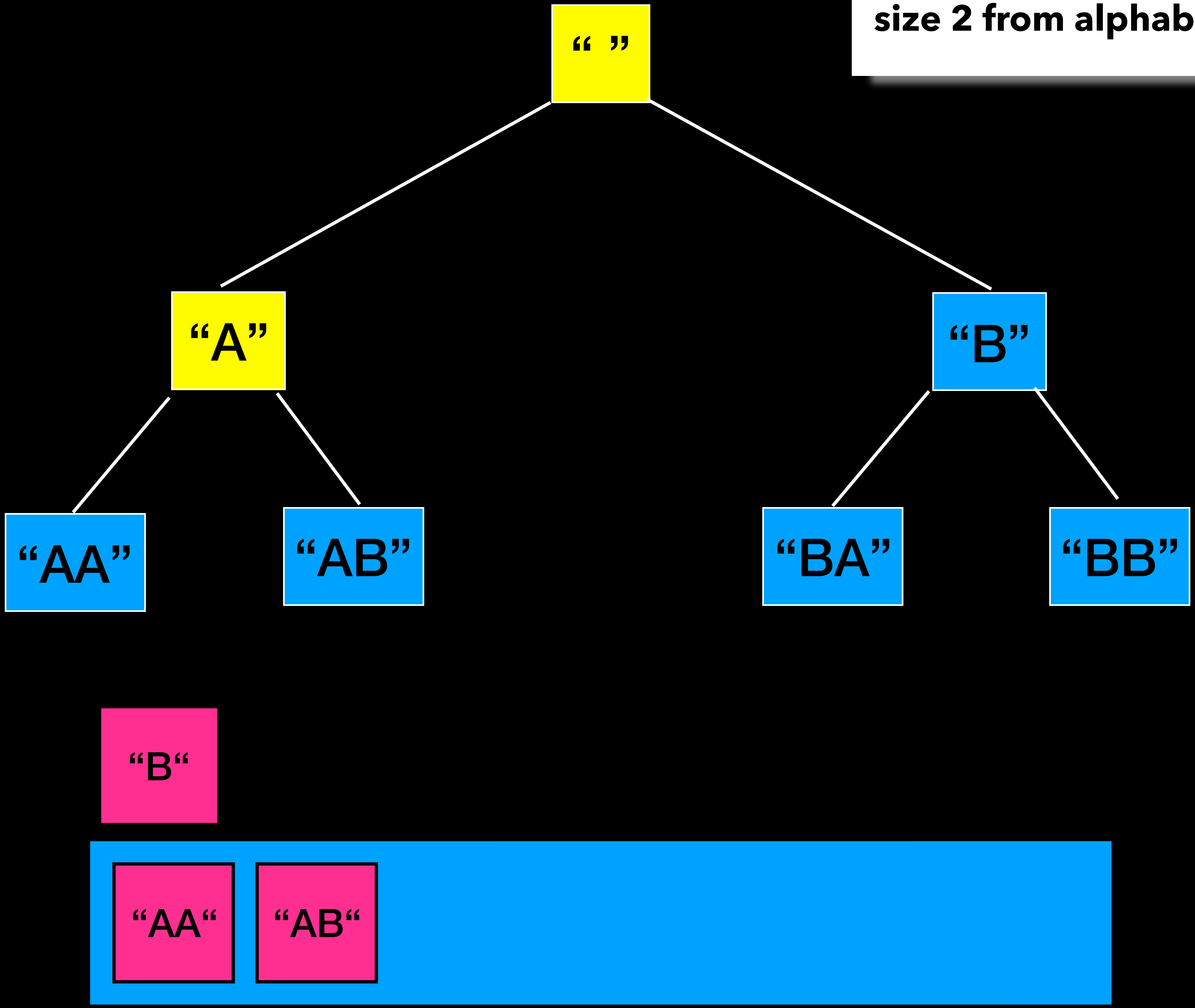
Generate all substrings of size 2 from alphabet {‘A’, ‘B’}





{ "", "A"}

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



" "

"A"

"B"

"AA"

"AB"

"BA"

"BB"

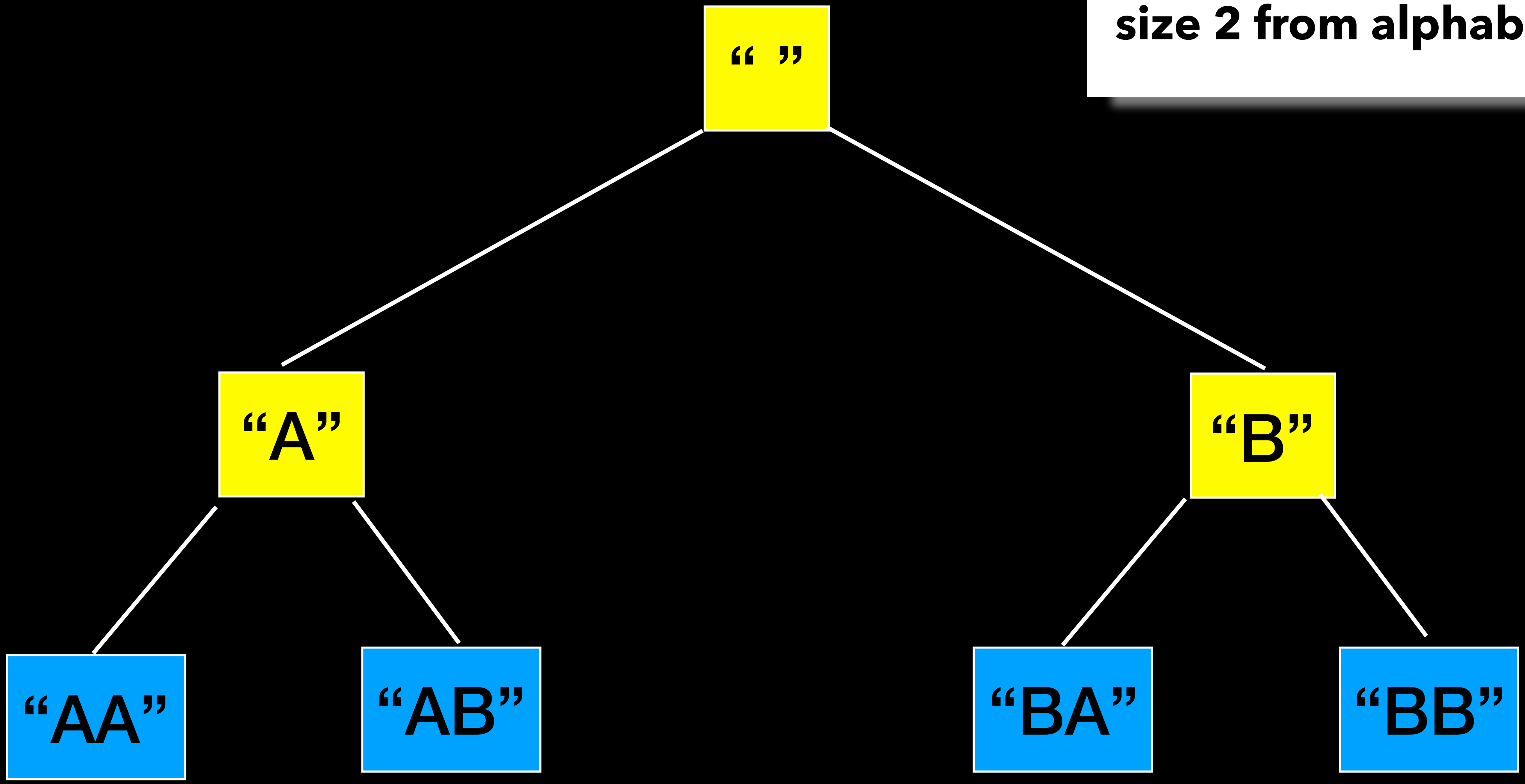
"B"

"AA"

"AB"

{ “”, “A”, “B”}

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


“ ”

“A”

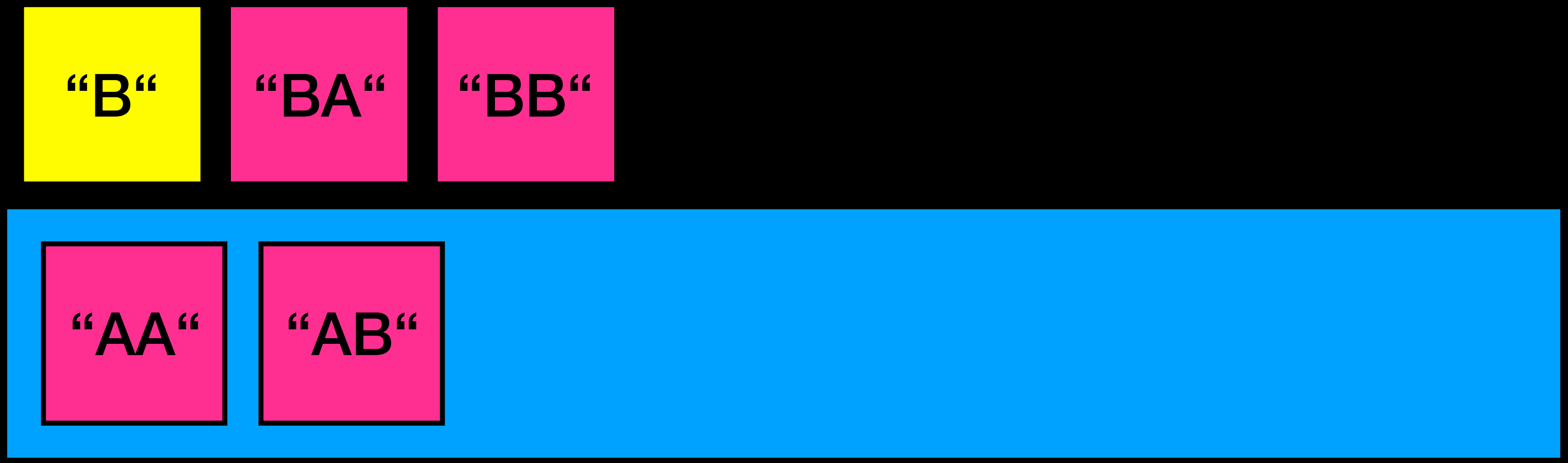
“B”

“AA”

“AB”

“BA”

“BB”


“B“

“BA“

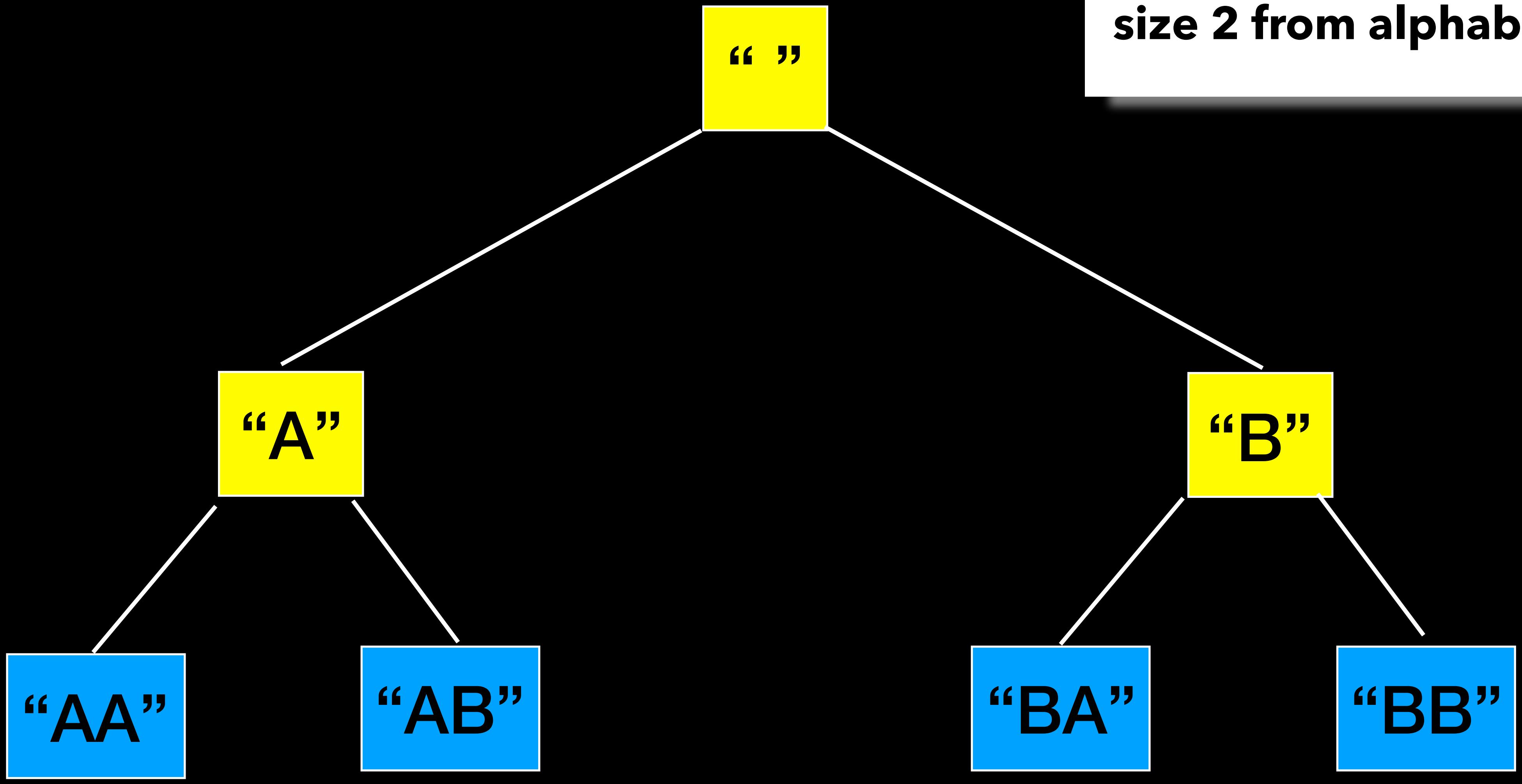
“BB“

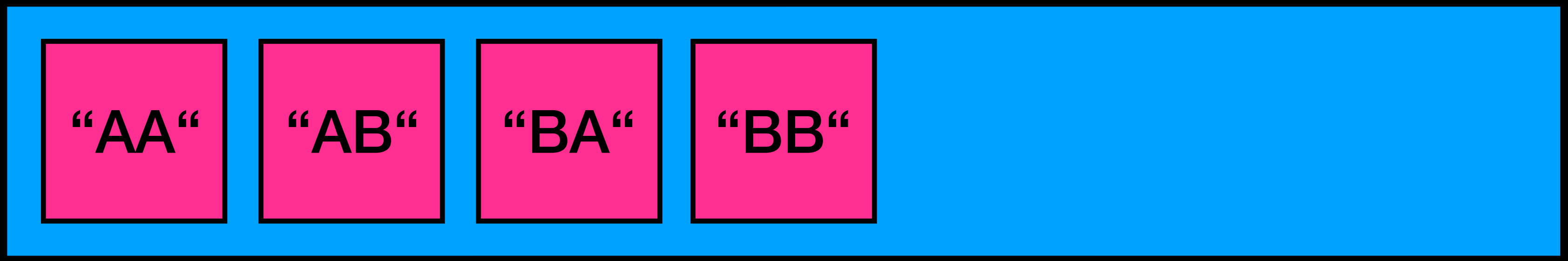
“AA“

“AB“

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

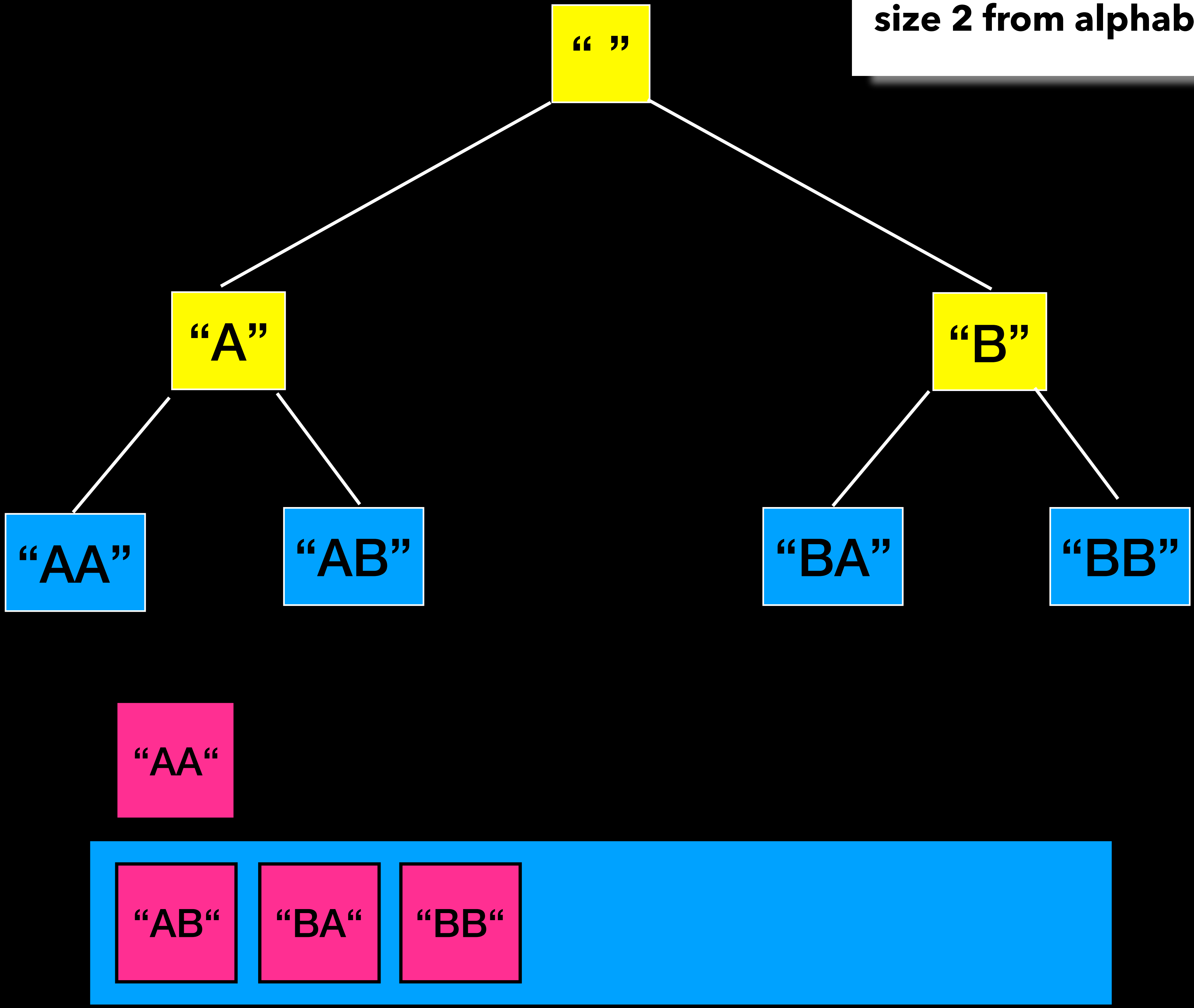
Generate all substrings of size 2 from alphabet {‘A’, ‘B’}





{ “”, “A”, “B”}

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



“ ”

“A”

“B”

“AA”

“AB”

“BA”

“BB”

“AA“

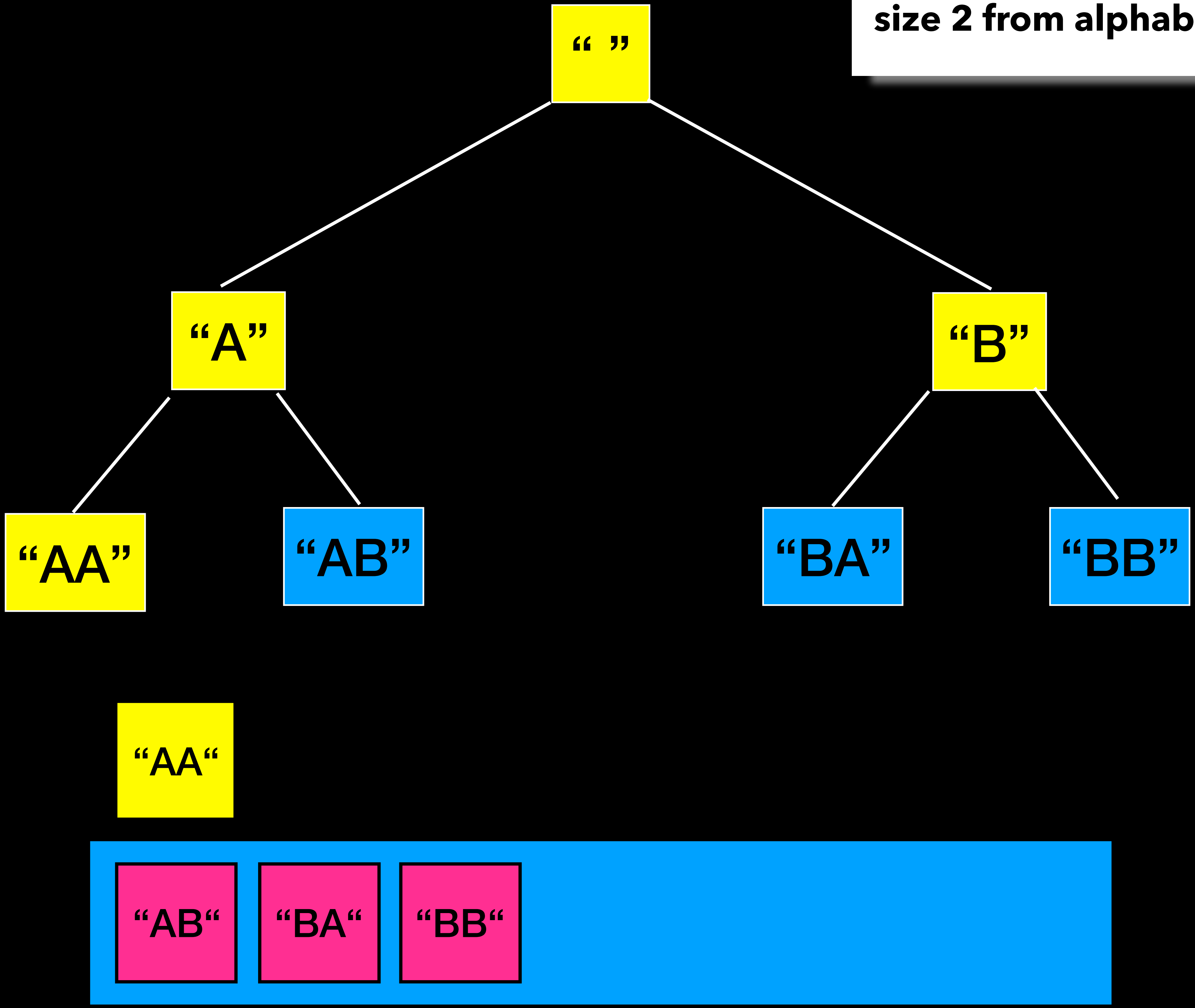
“AB“

“BA“

“BB“

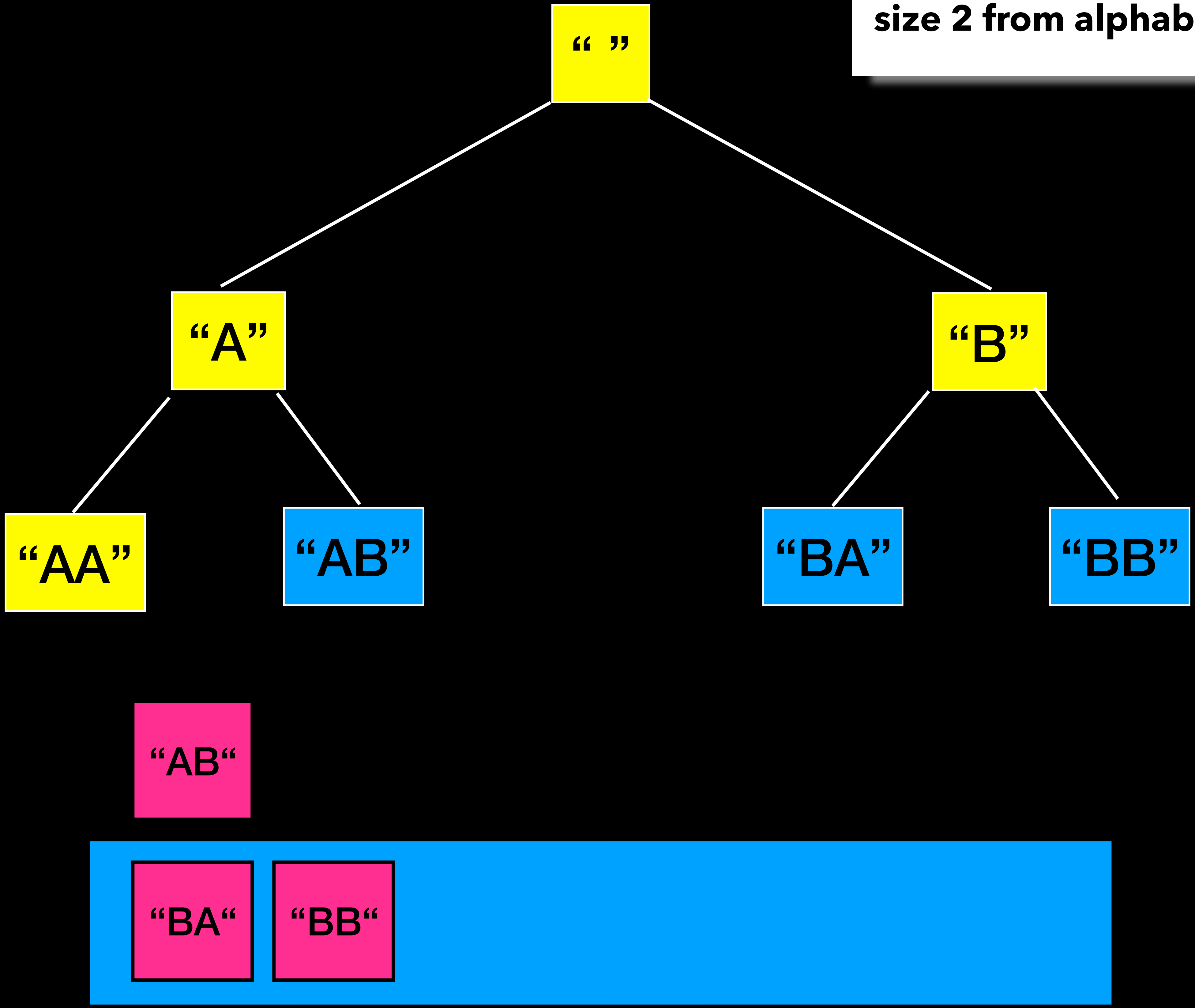
{ "", "A", "B", "AA"}

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



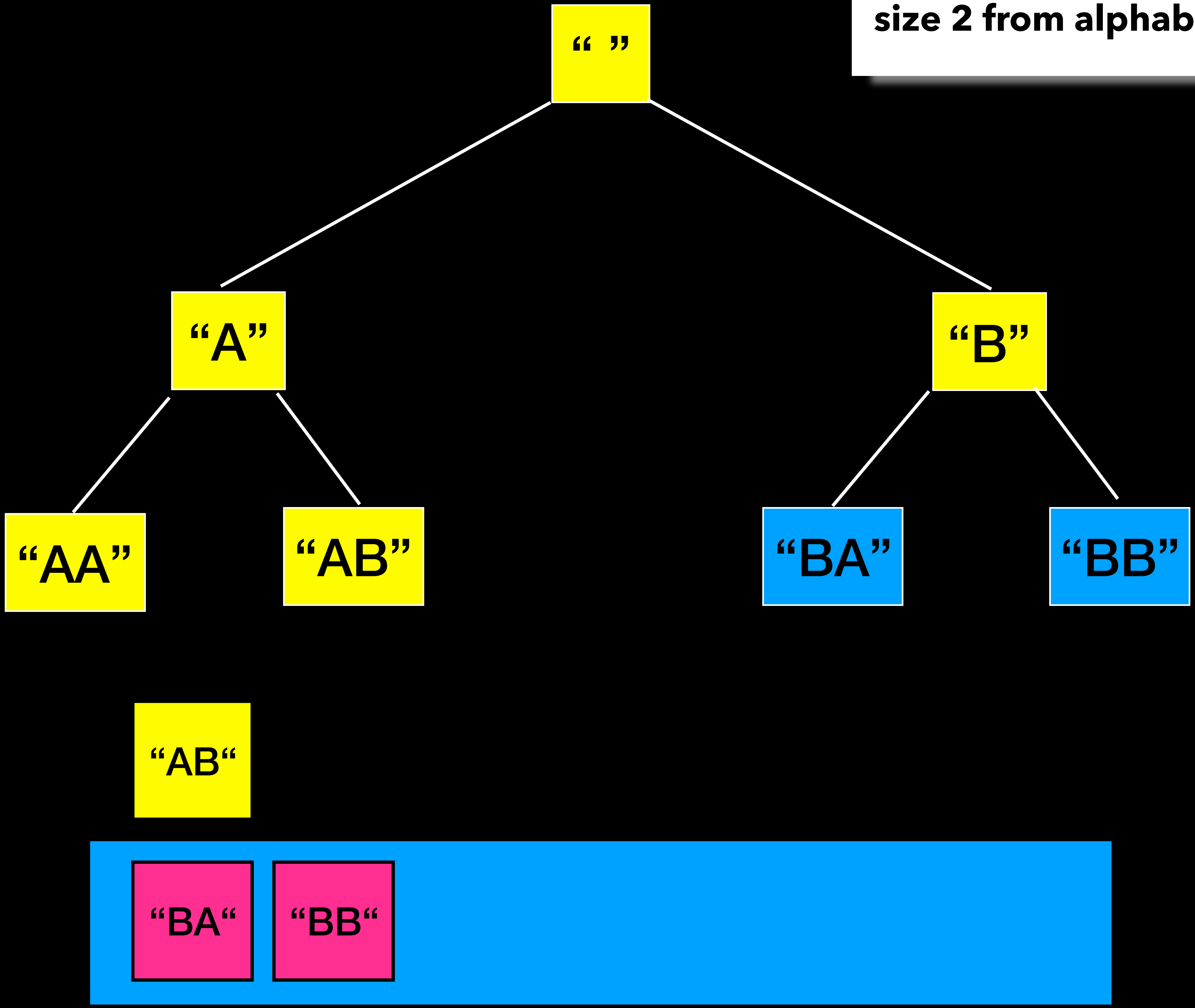
{ “”, “A”, “B”, “AA”}

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



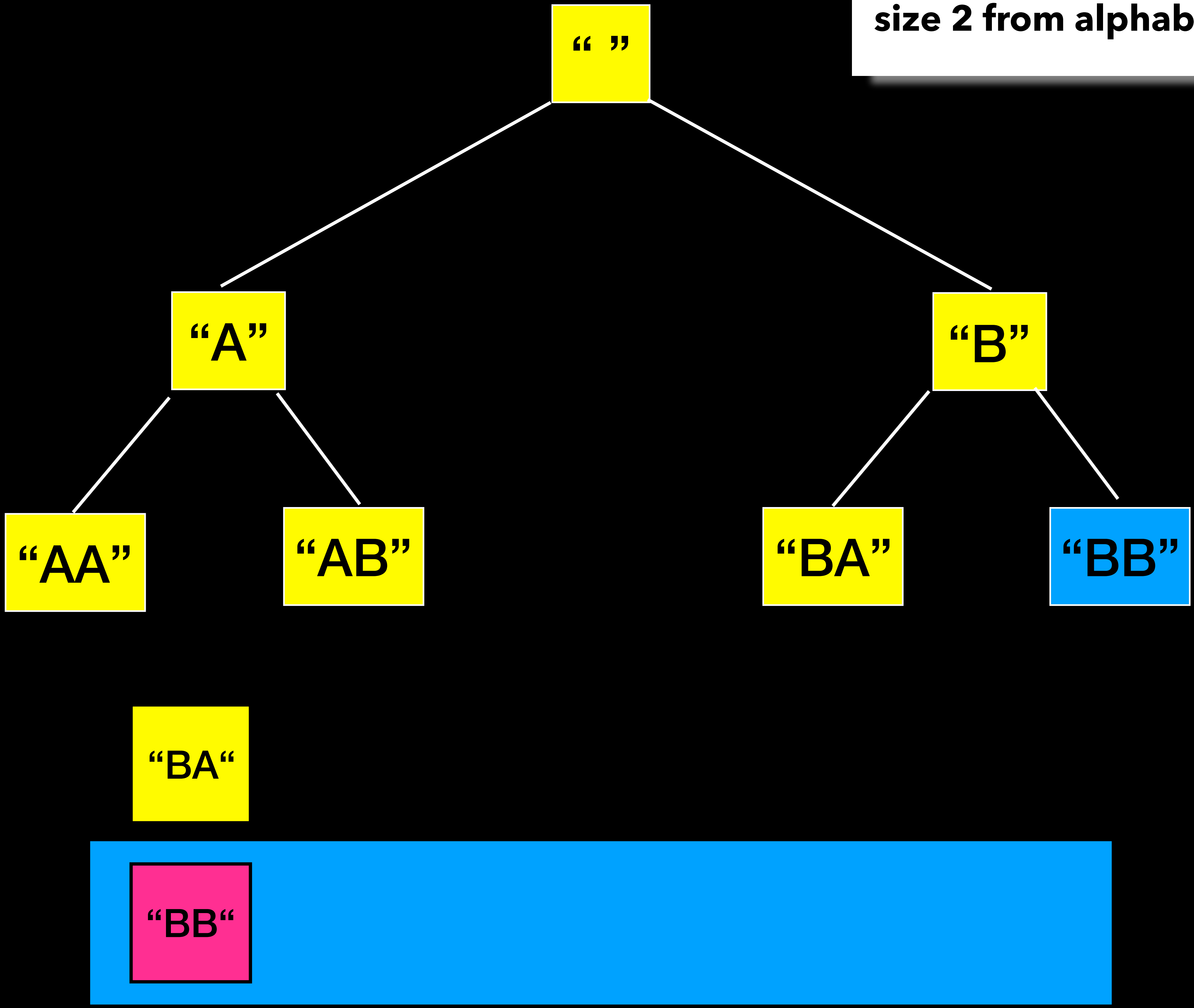
{ “”, “A”, “B”, “AA”, “AB”}

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



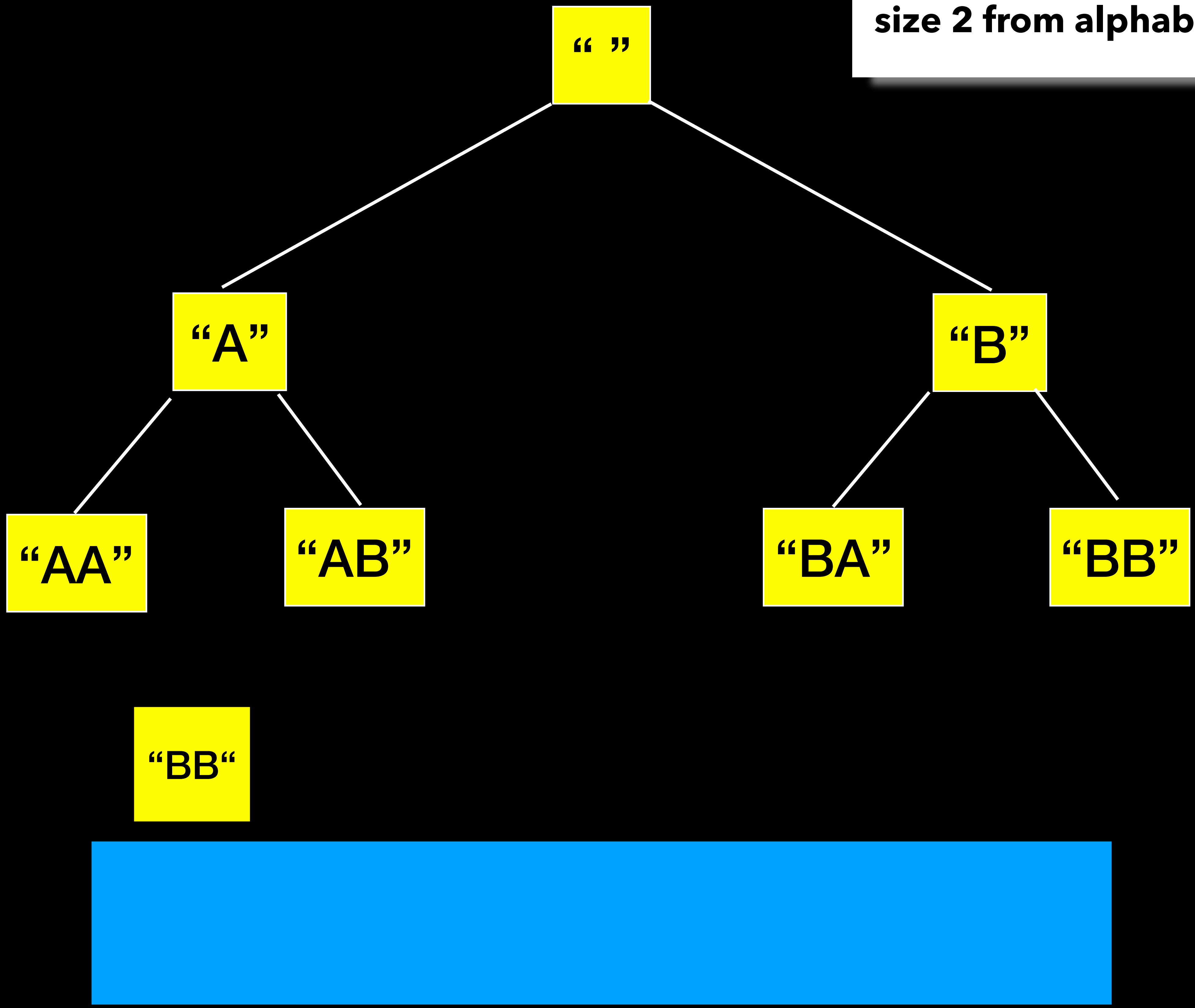
{ “”, “A”, “B”, “AA”, “AB”, “BA”}

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



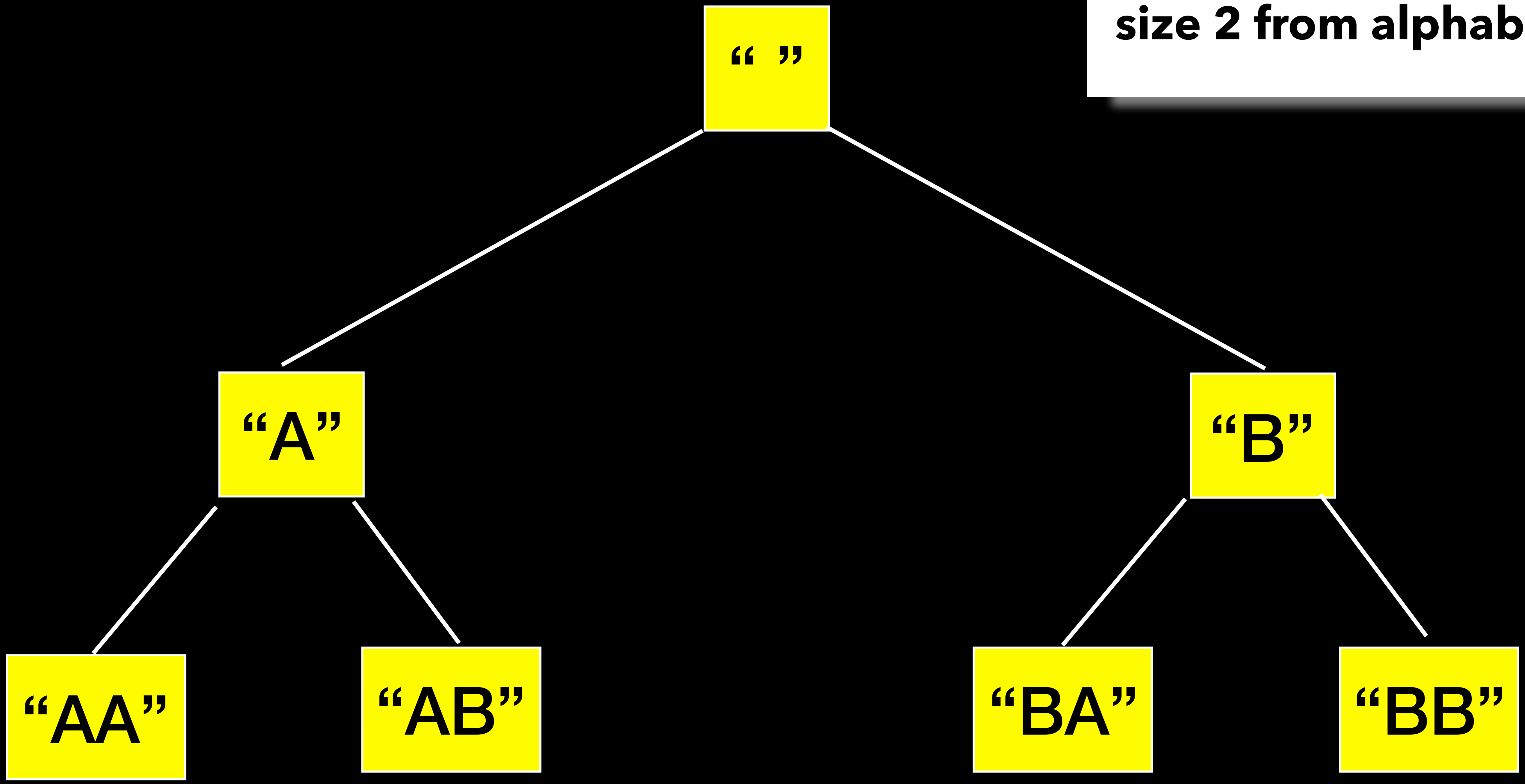
{ “”, “A”, “B”, “AA”, “AB”, “BA”, “BB” }

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'}



" "

"A"

"B"

"AA"

"AB"

"BA"

"BB"

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 = front of queue and add to result
        if(size of 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 all substrings (with repetition) of size **up to n**

Assume alphabet (A, B, … , Z) 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

Lecture Activity

Analyze the worst-case time complexity of this algorithm

T(n) = ?

O(?)

Size of Substring

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

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(size of 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

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

    while(queue is not empty){
        let current_string = front of queue and add to result
        if(size of 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

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 = front of queue and add to result
        if(size of 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

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 = front of queue and add to result
        if(size of current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    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 = front of queue and add to result
        if(size of current_string < n){
            for(each character ch)//every character in alphabet
                append ch to current_string and add it to queue
        }
    }
    return result;
}
```

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

Will stop when all strings have been removed from queue

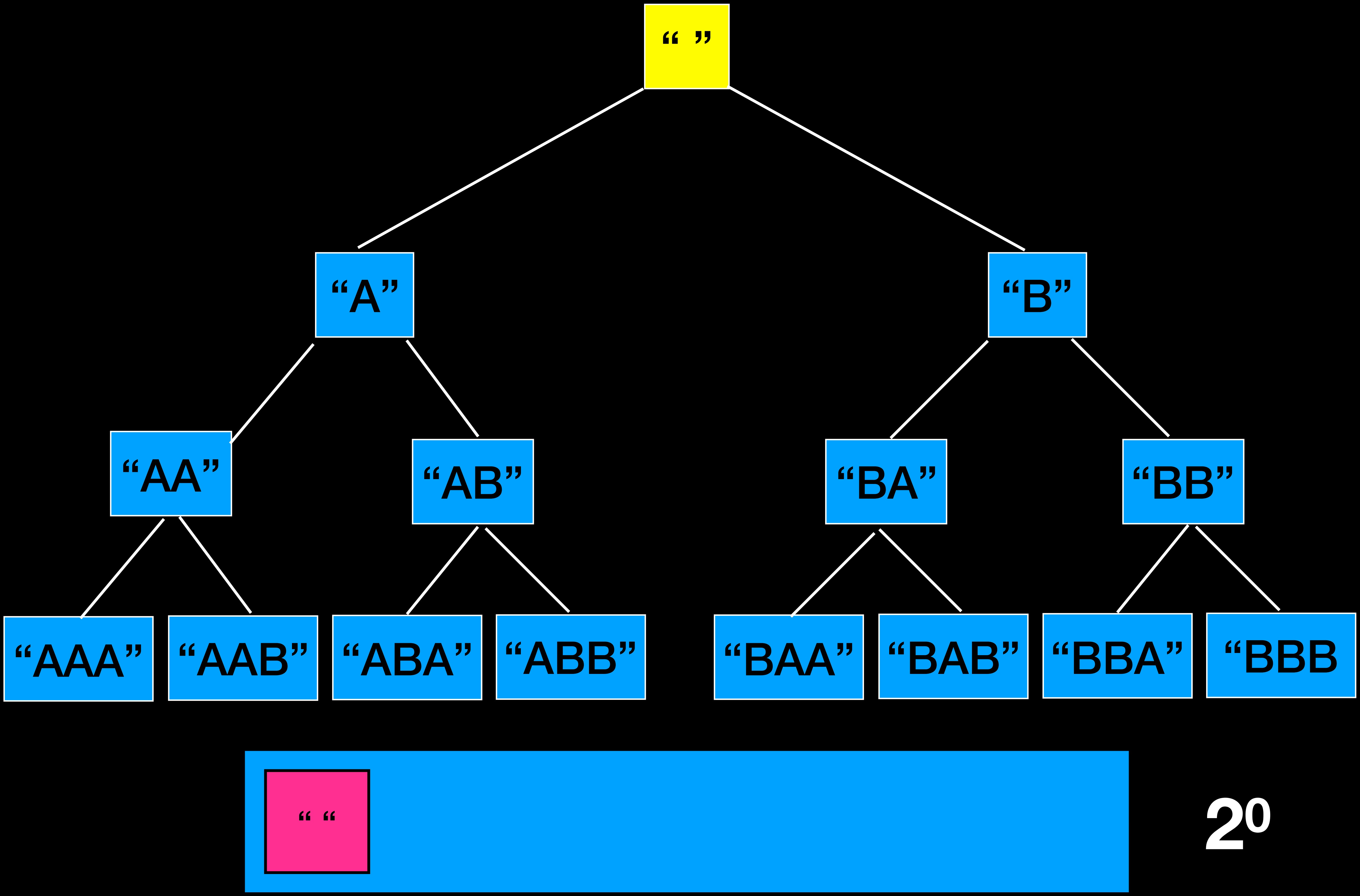
Removes 1 string from the queue

Adds 26 strings to the queue

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

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

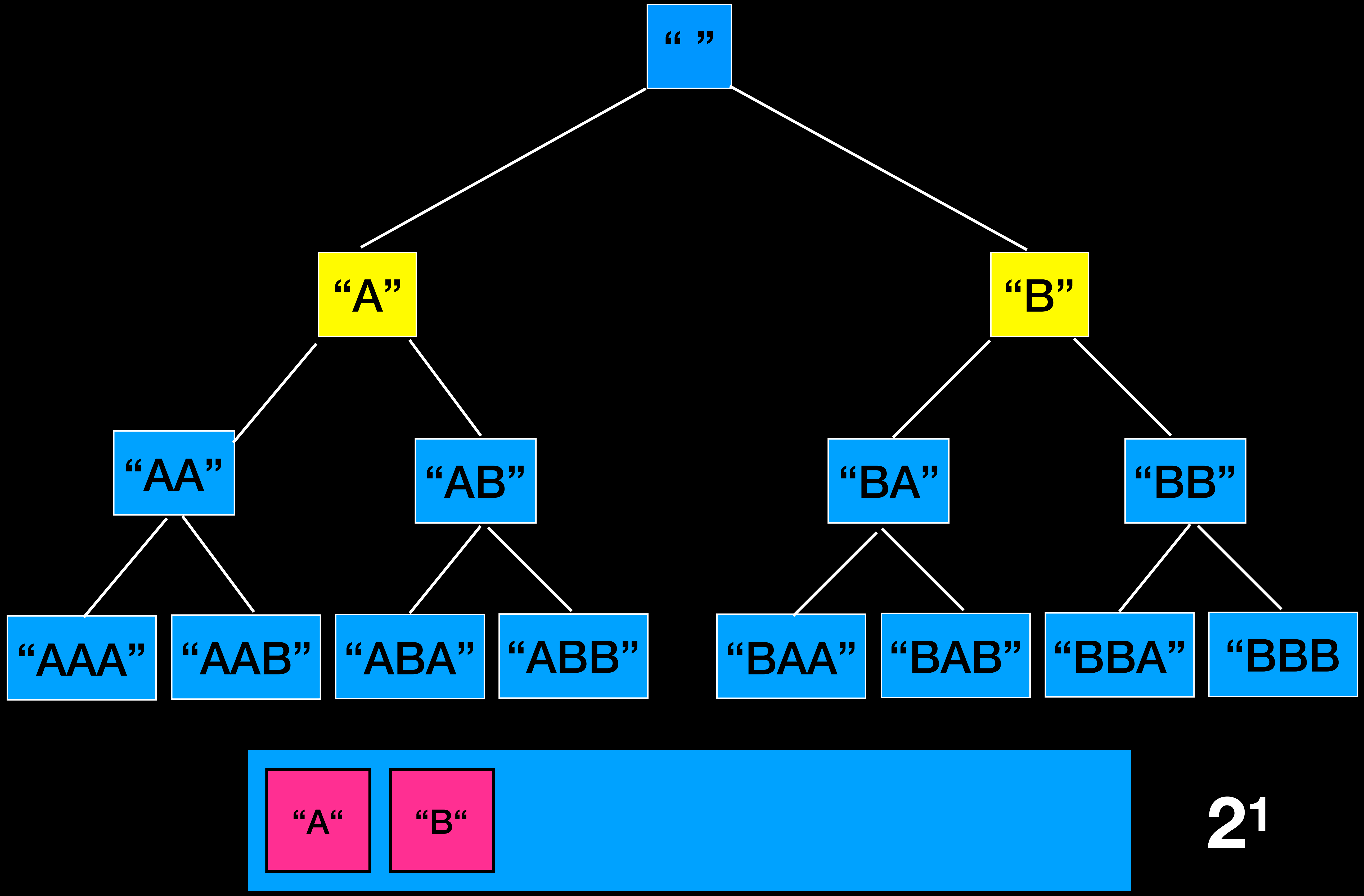
$O(26^n)$

Let n = 3, alphabet still {‘A’,’B’}

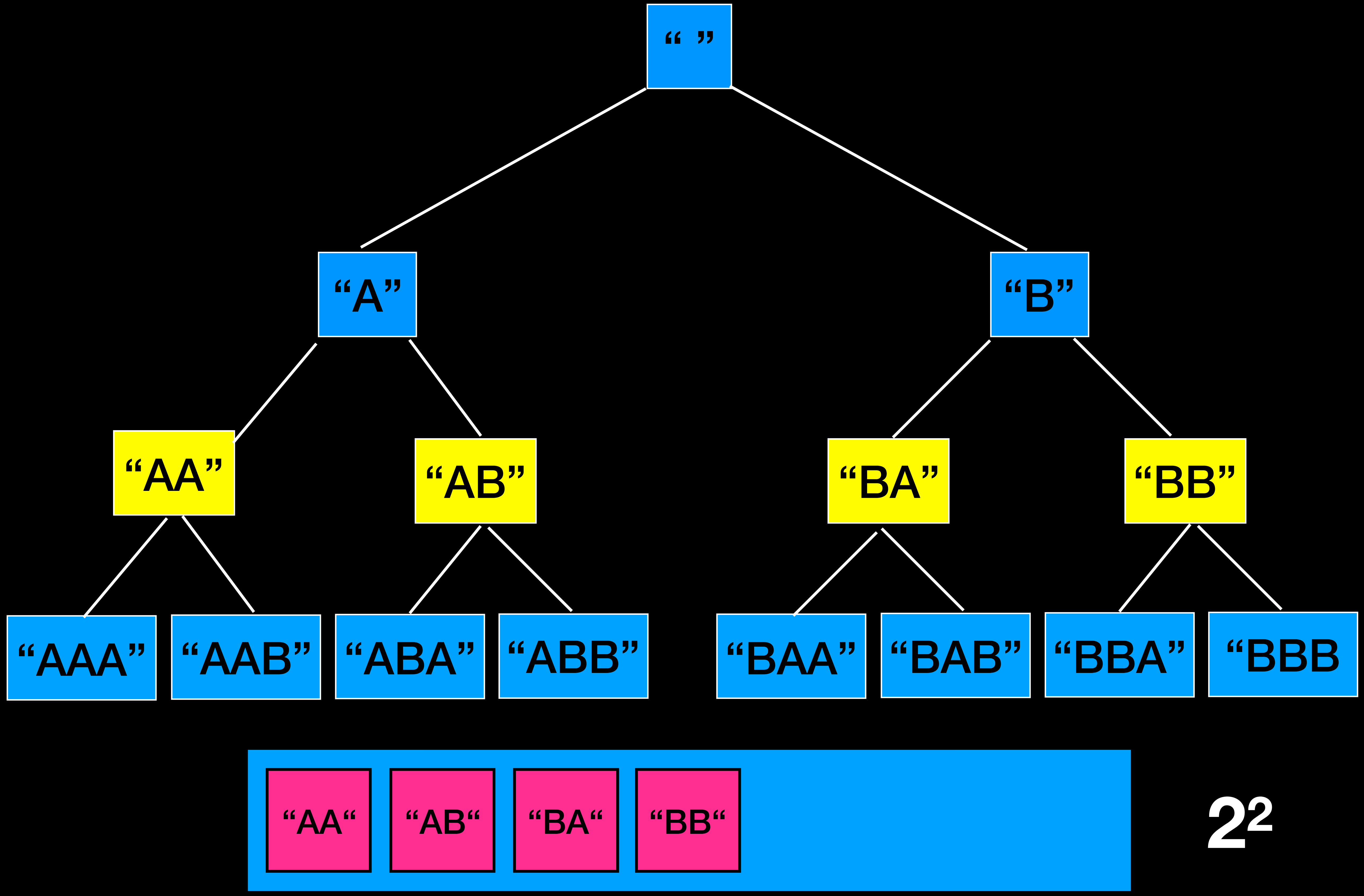


2^0

Let n = 3, alphabet still {‘A’,’B’}

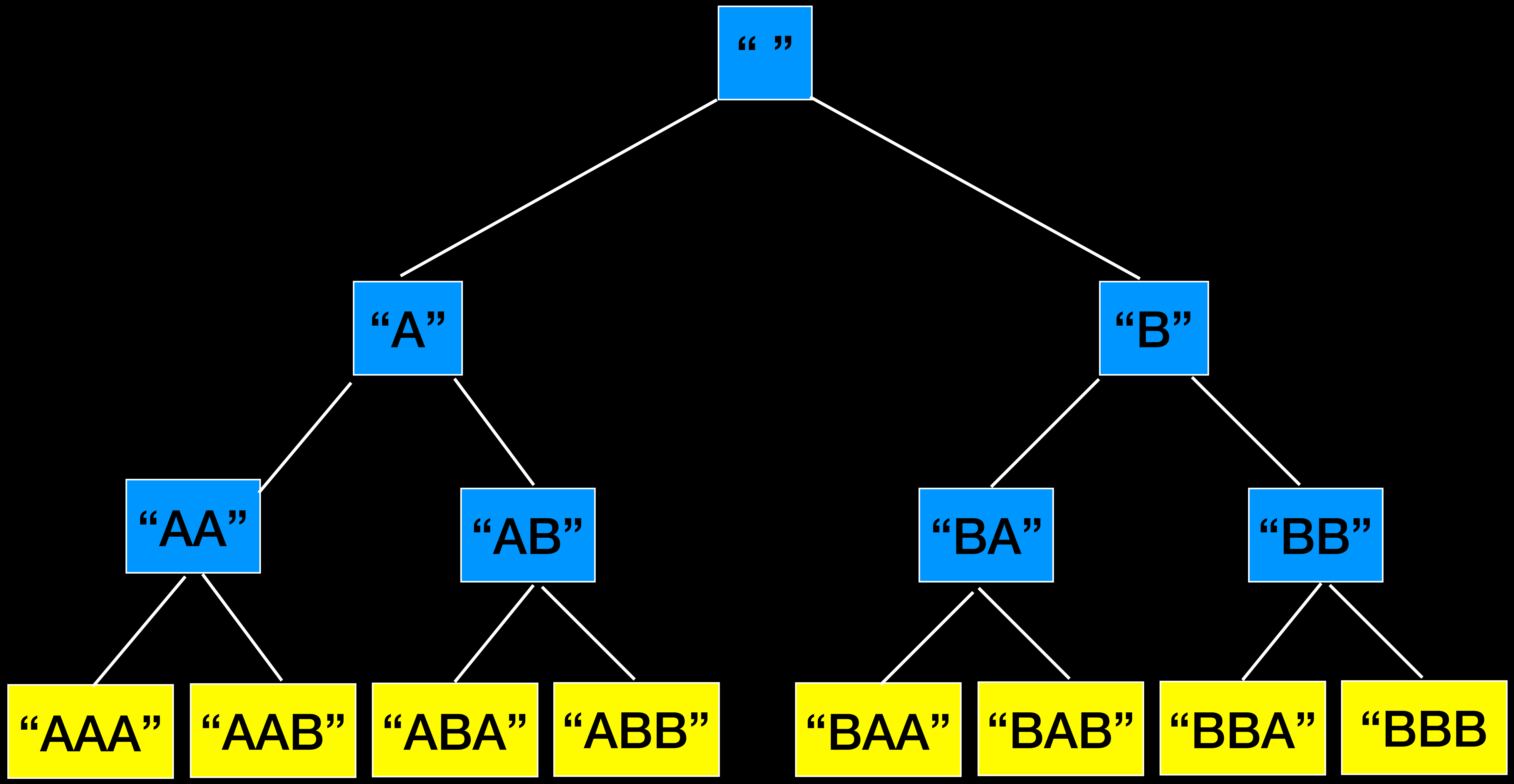


Let n = 3, alphabet still {‘A’,’B’}



2^2

Let n = 3, alphabet still {‘A’,’B’}



“ ”

“A” “B”

“AA” “AB” “BA” “BB”

“AAA” “AAB” “ABA” “ABB” “BAA” “BAB” “BBA” “BBB

“AAA“	“AAB“	“ABA“	“ABB“	“BAA“	“BAB“	“BBA“	“BBB“

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

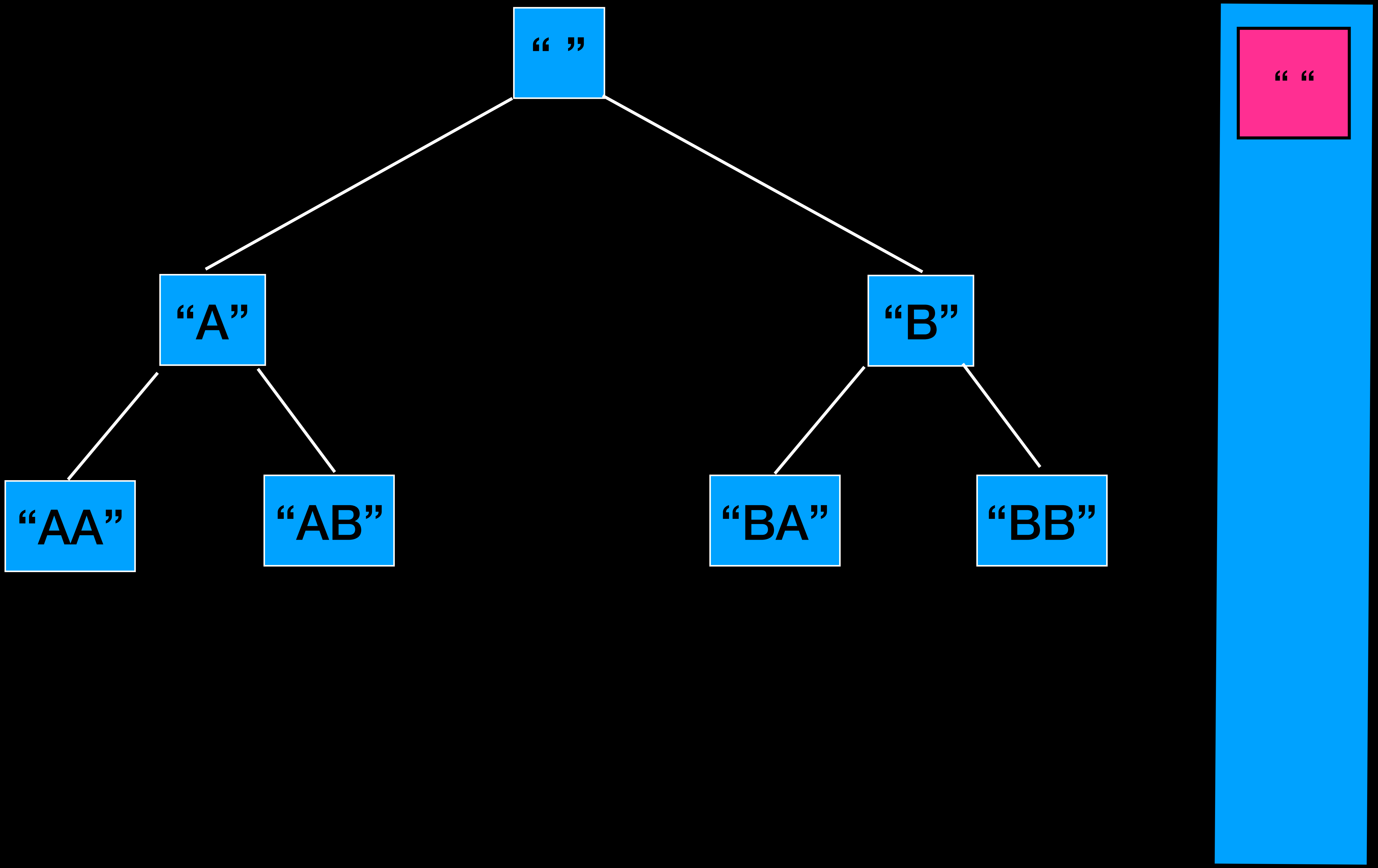
For n = 8 ≈ 5TB

Massive space requirement

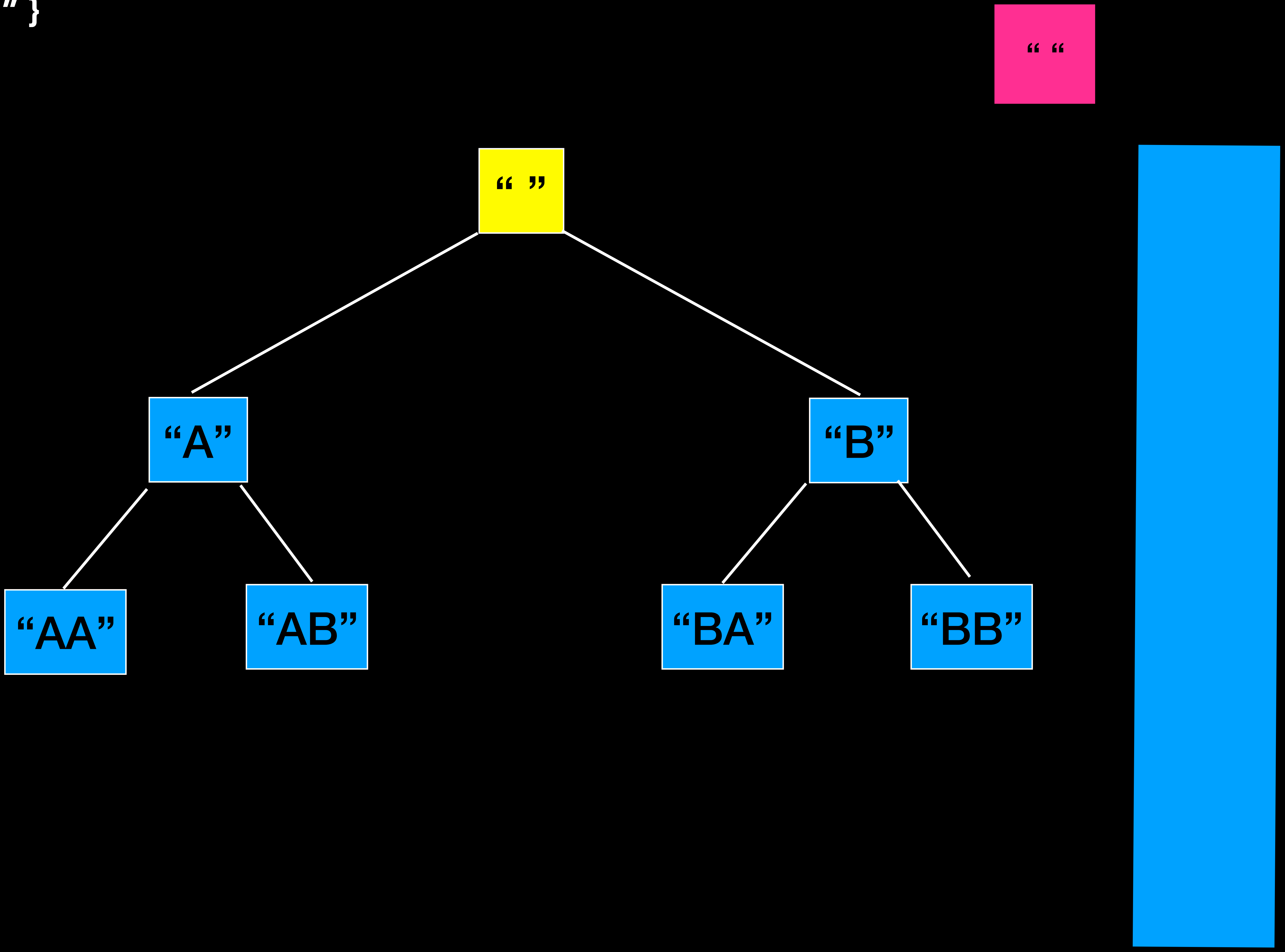
What if we use a stack?

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

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

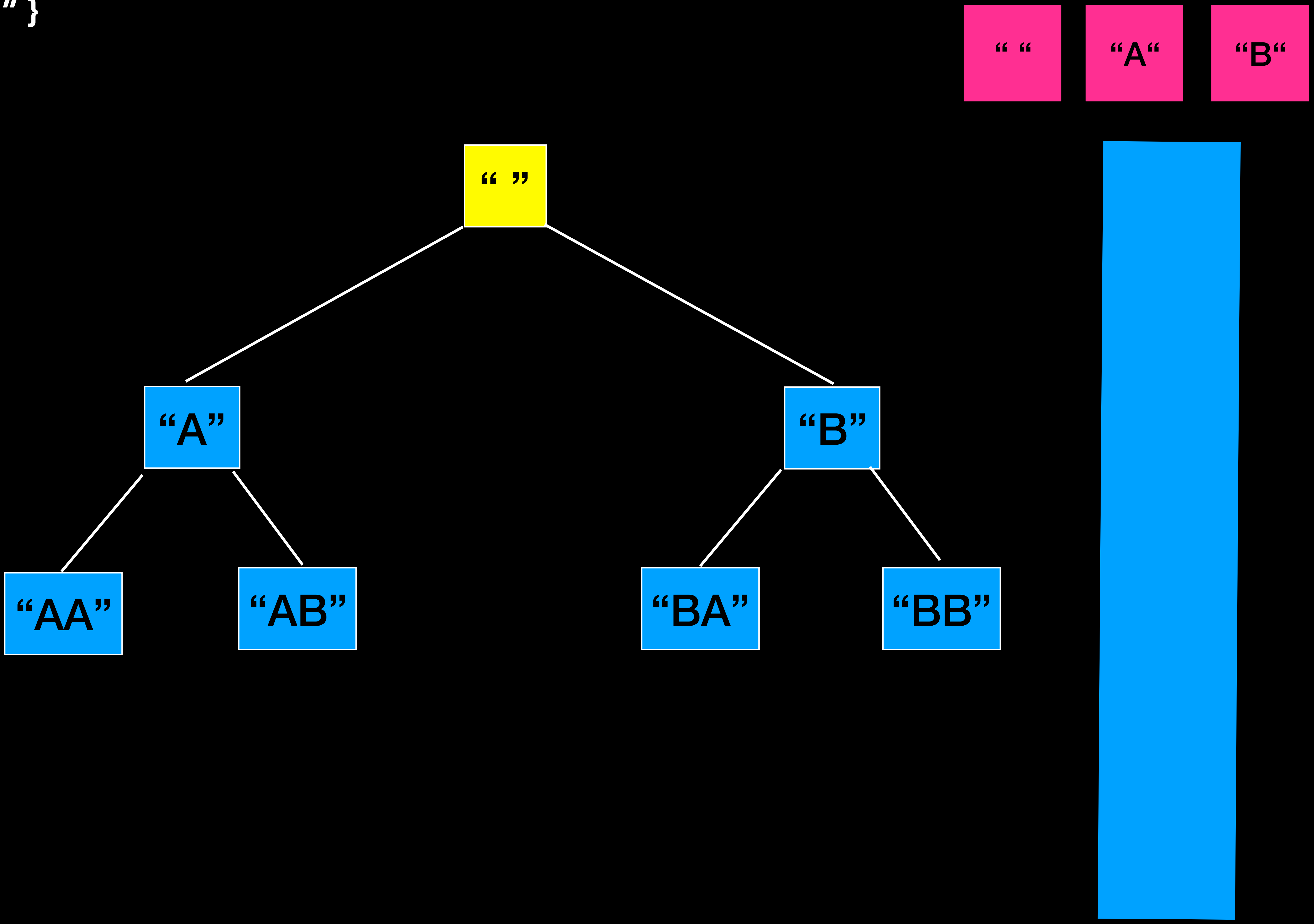
$O(26^n)$


“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“ “

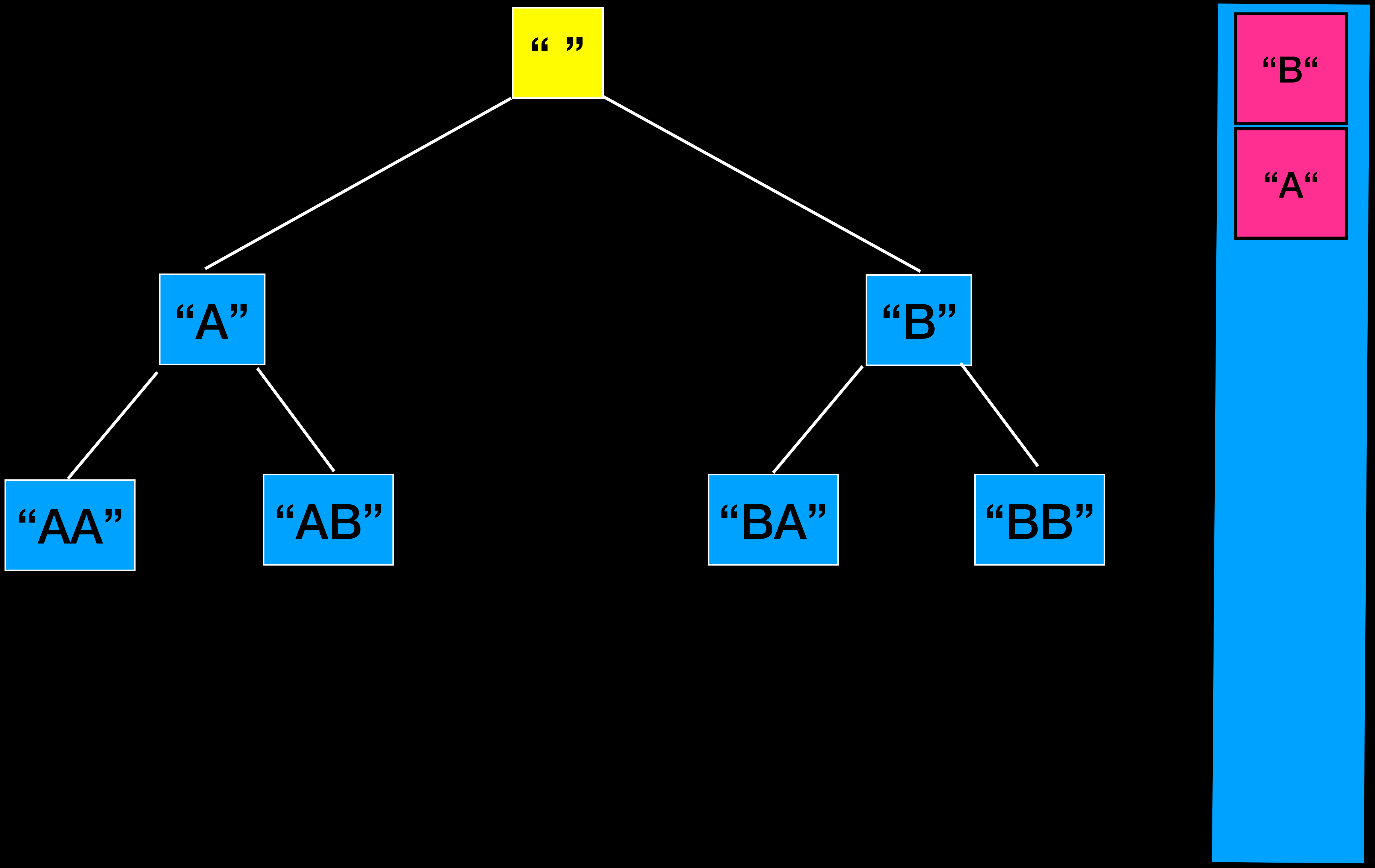
{ "" }


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

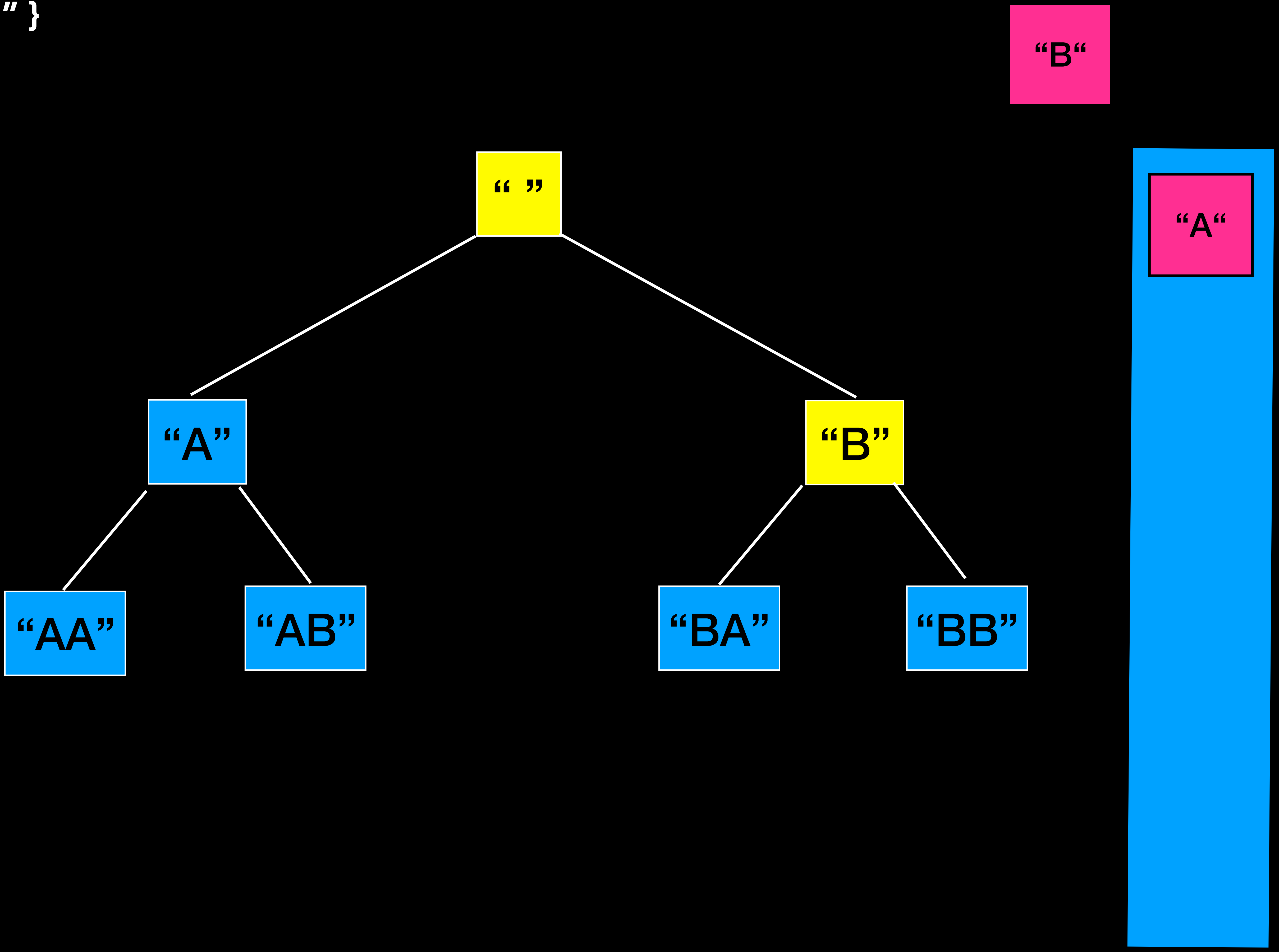
{ ”” }


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

{ "" }

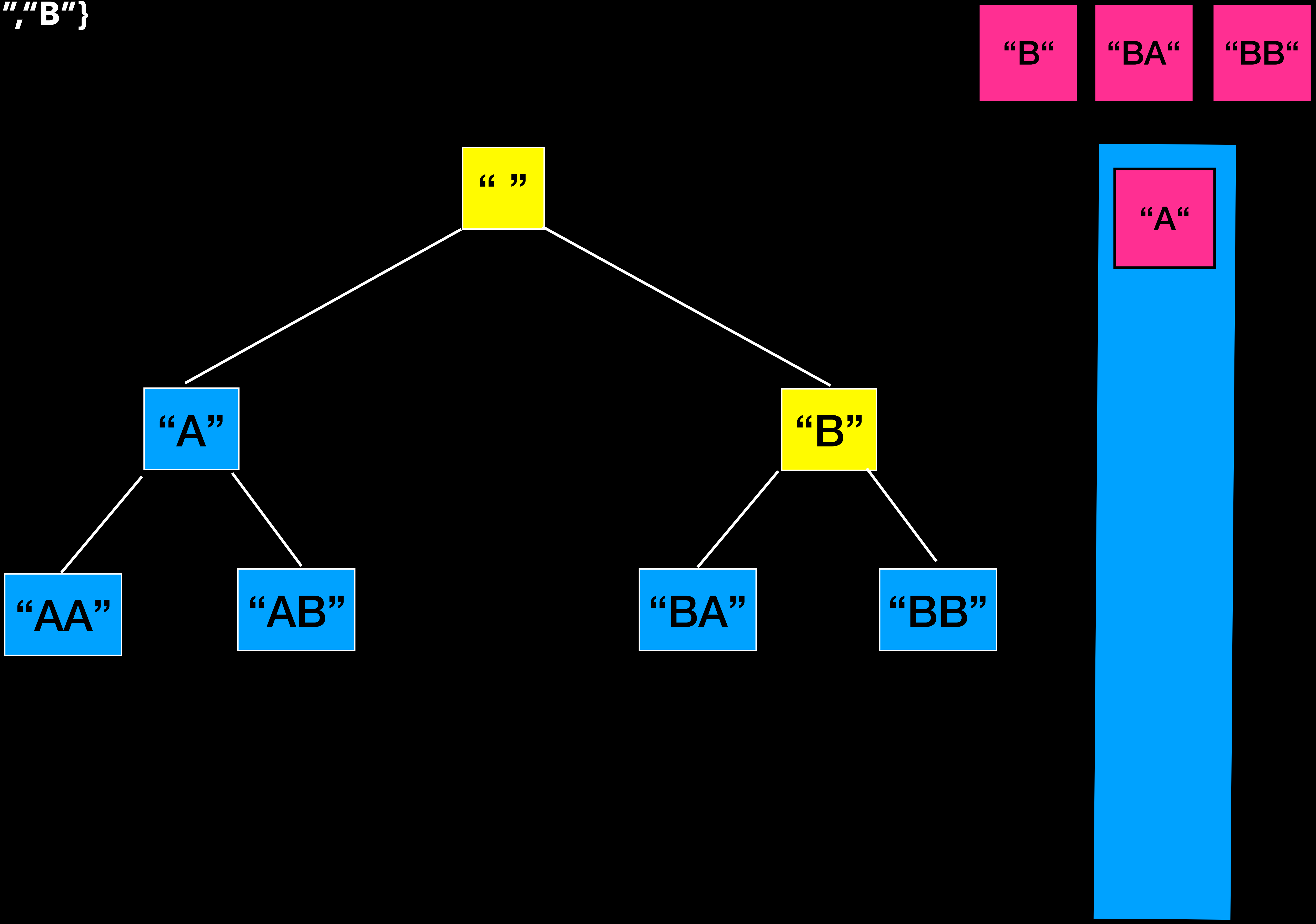

“ ”
“A”
“B”
“AA”
“AB”
“BA”
“BB”
“B“
“A“

{ "" }

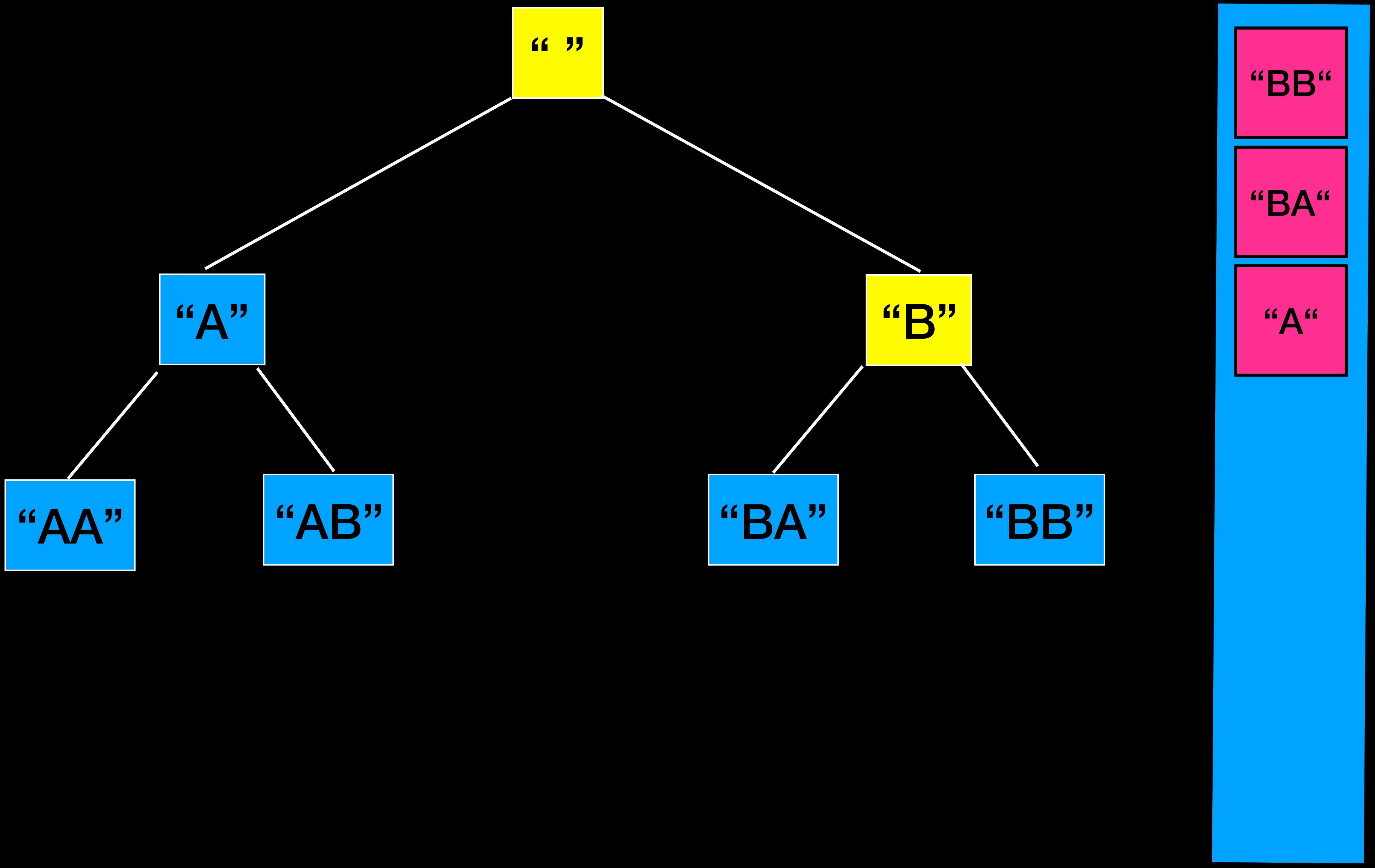


"B"
" "
"A"
"B"
"AA"
"AB"
"BA"
"BB"
"A"

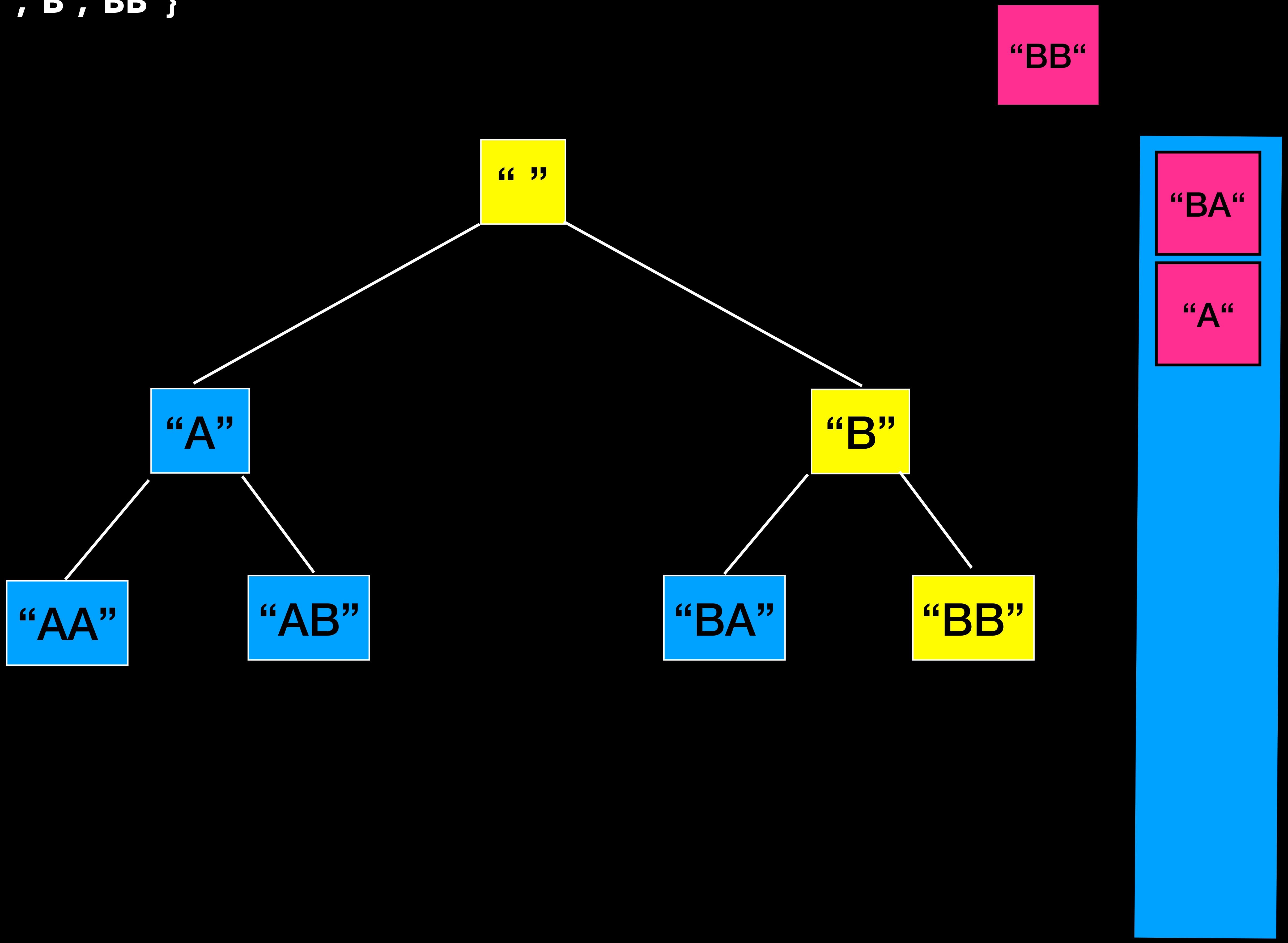
{ “”,”B”}



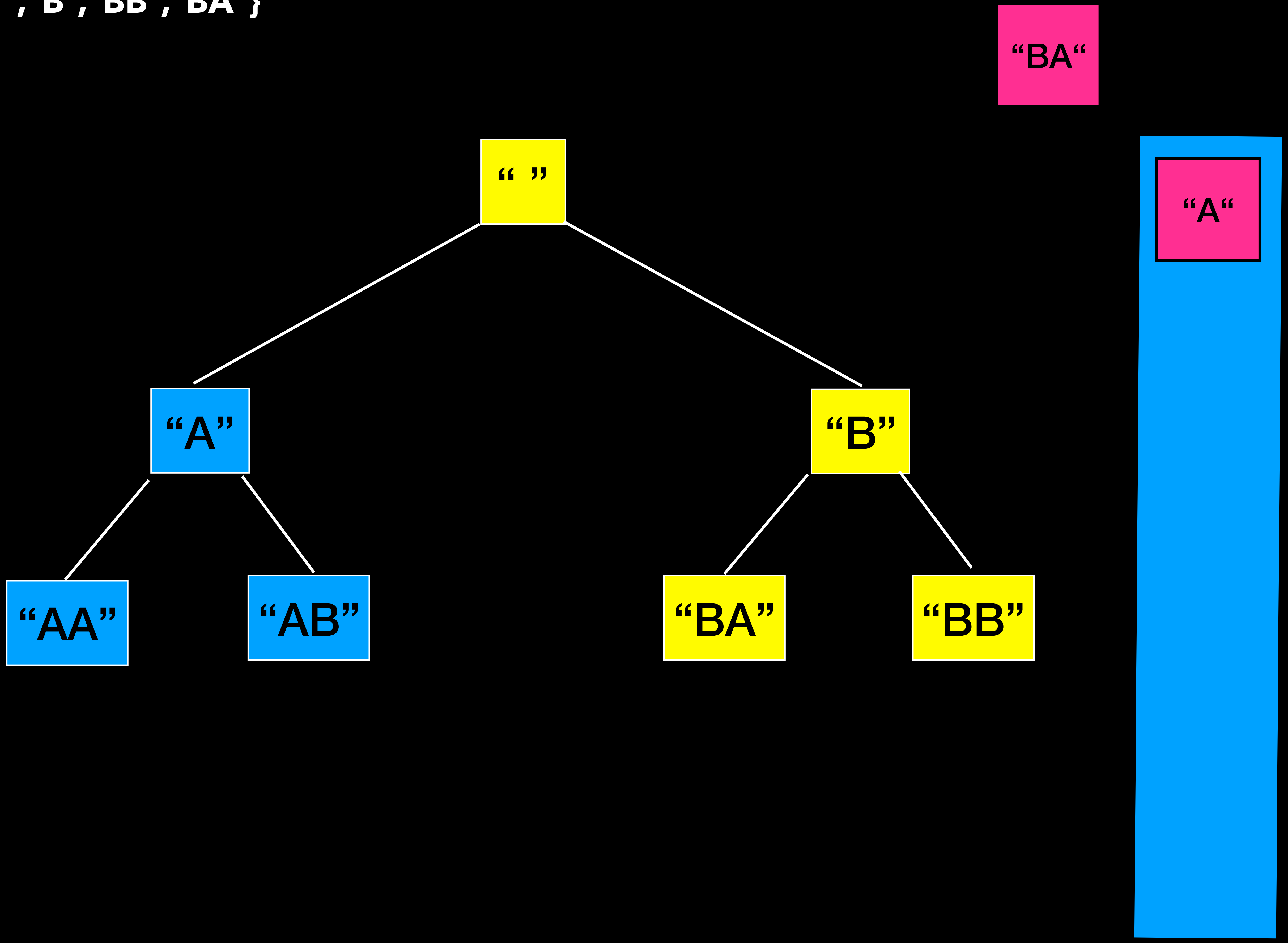
{ “”,”B”}



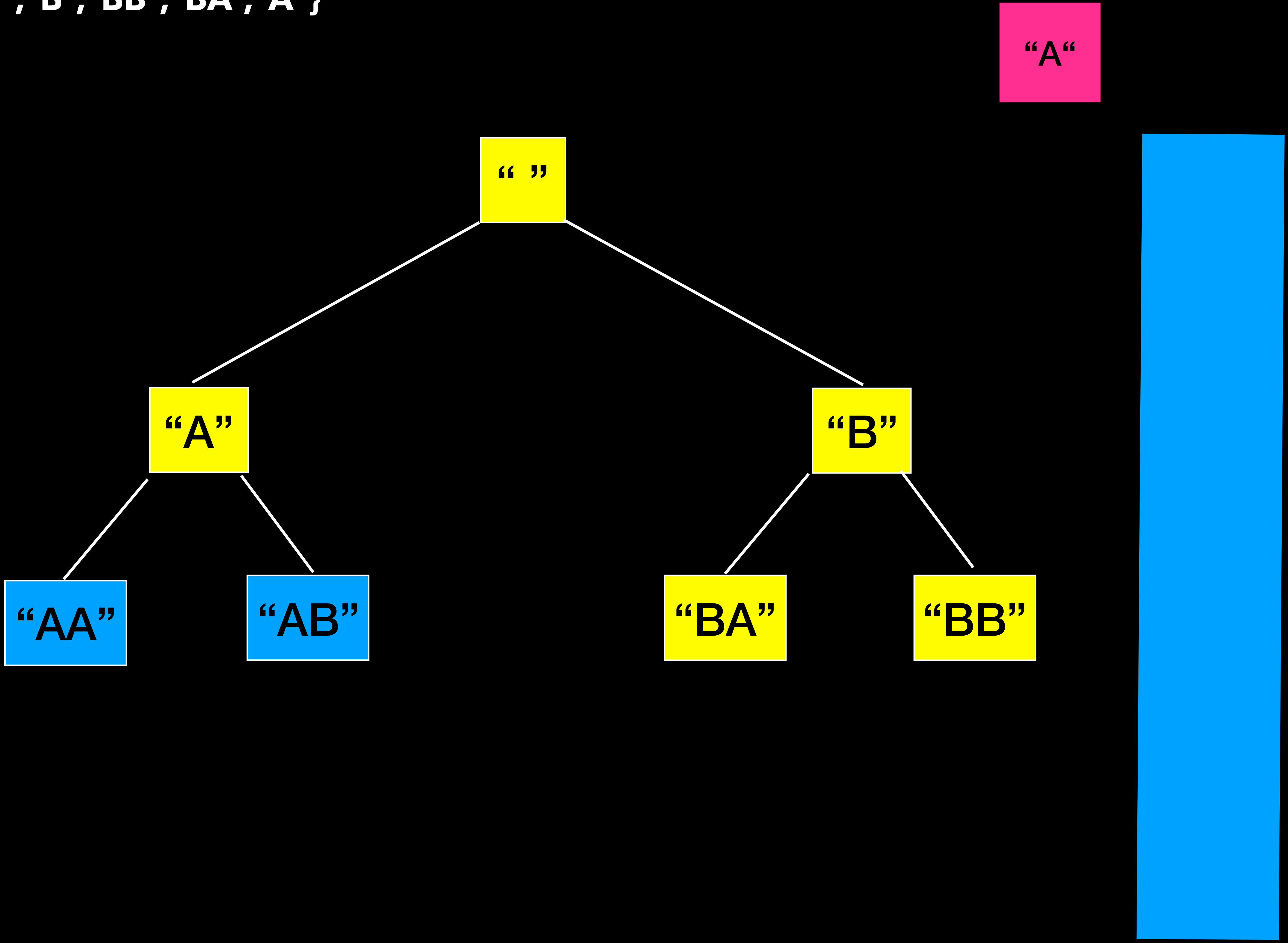
{ “”,”B”,”BB”}



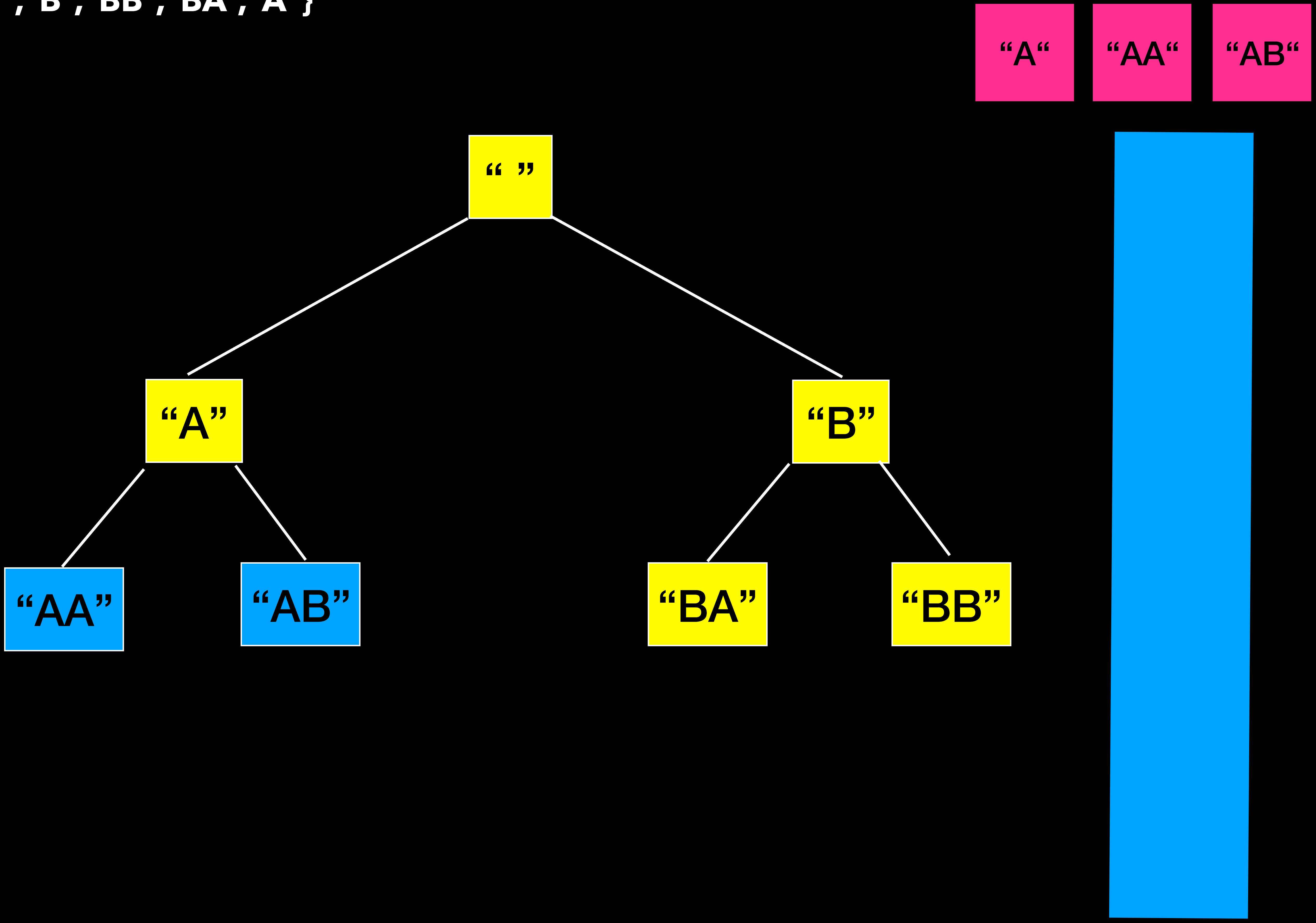
{ “”,”B”,”BB”,”BA”}



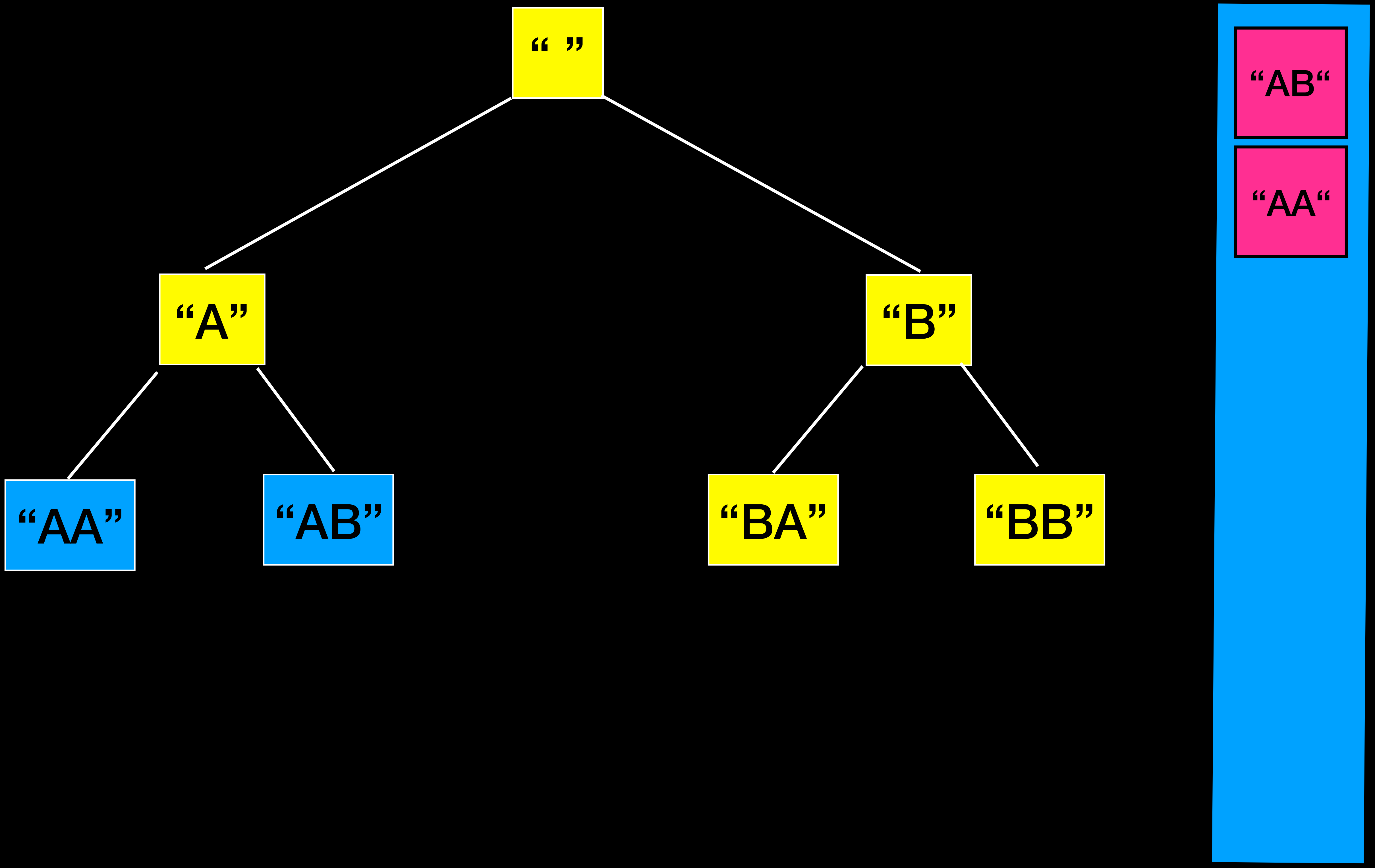
{ “”,”B”,”BB”,”BA”,”A”}



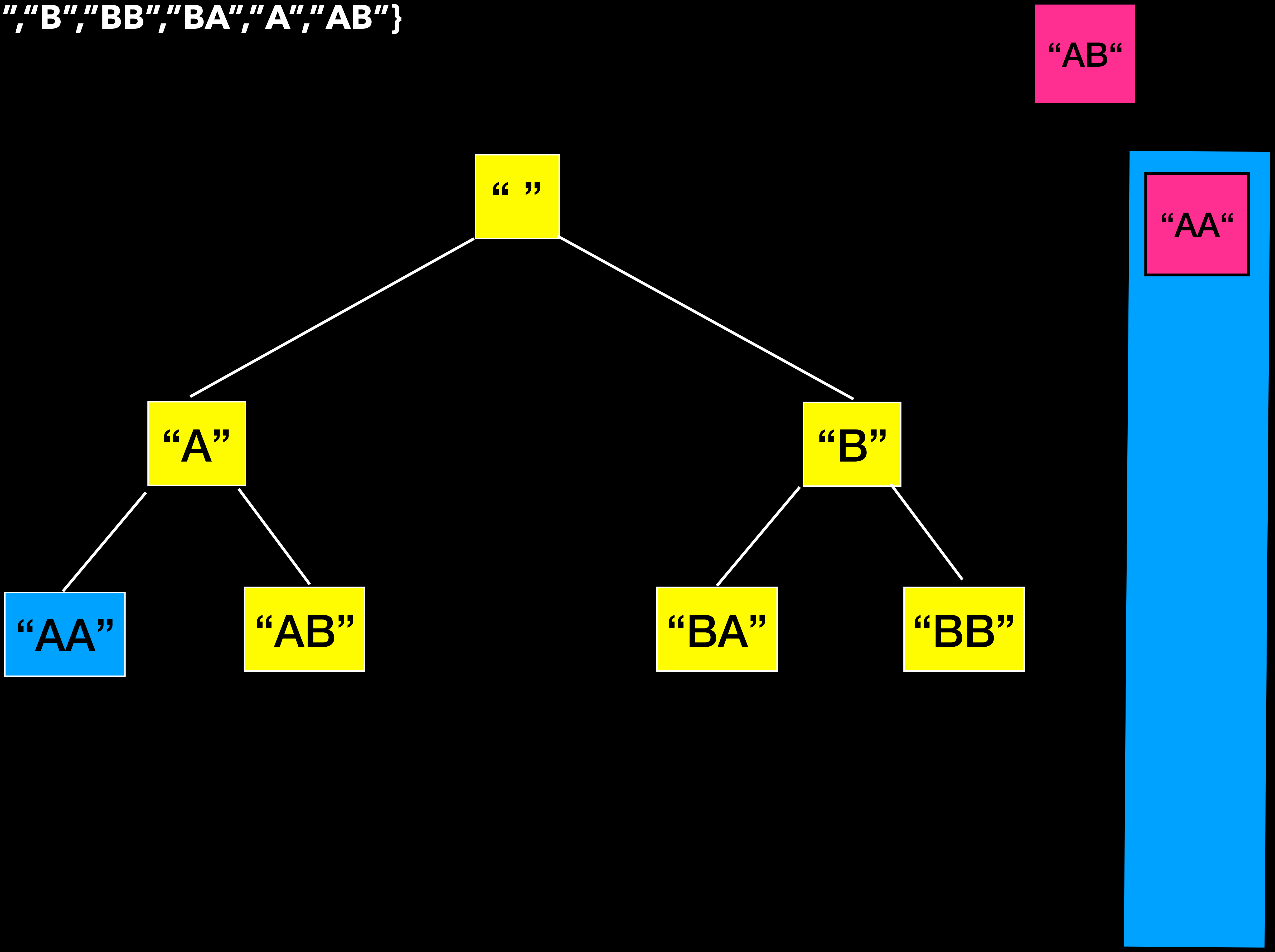
{ “”,”B”,”BB”,”BA”,”A”}



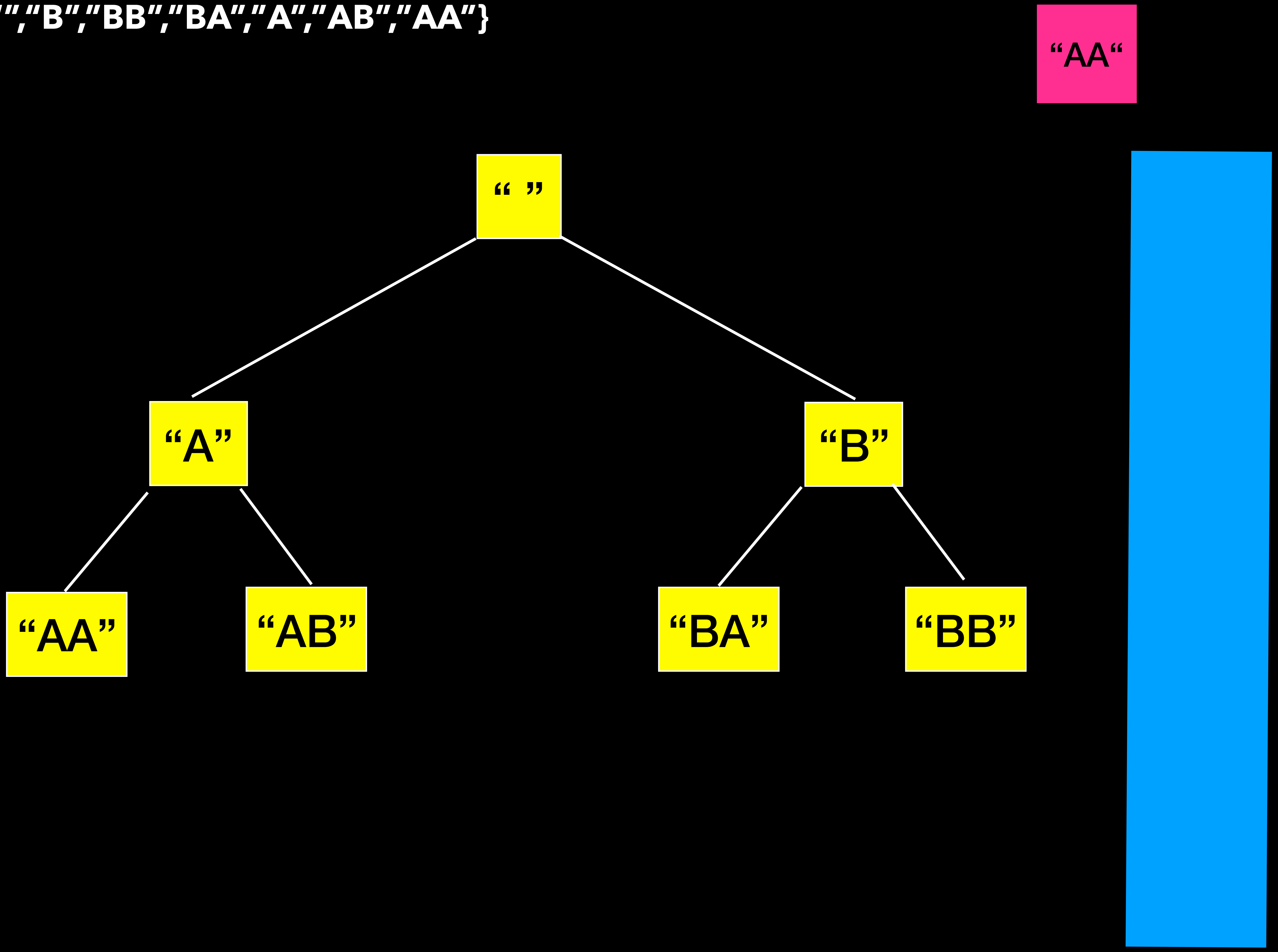
{ “”,”B”,”BB”,”BA”,”A”}



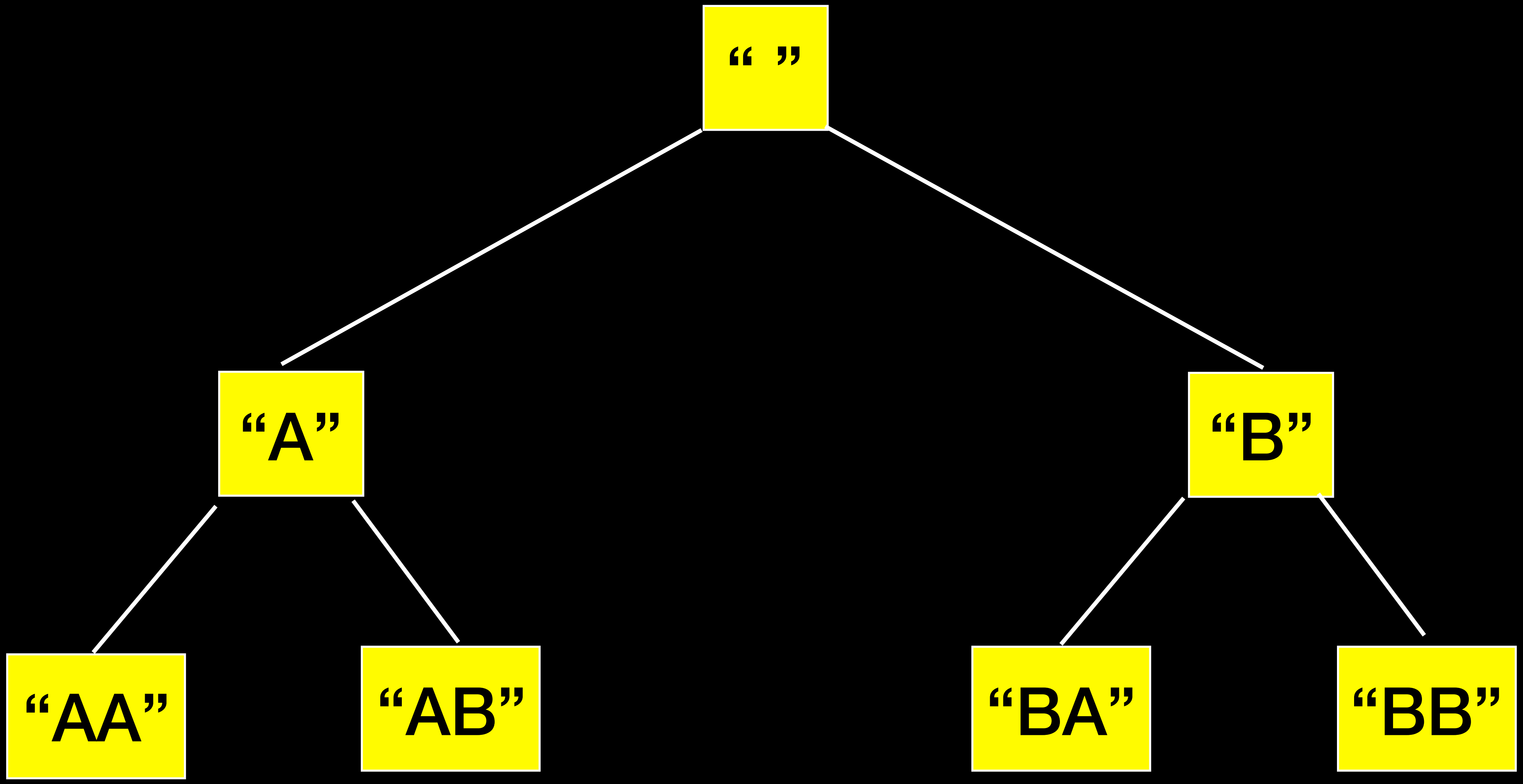
{ "","B","BB","BA","A","AB"}



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



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



“ ”

“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
- . . .

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

Recognizing Palindromes

Palindrome: a string that reads the same in reverse order

Anna

Civic

Kayak

Noon

Radar

Lecture Activity

Write **C++** for

```
bool isPalindrome(string s)
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);
}
```

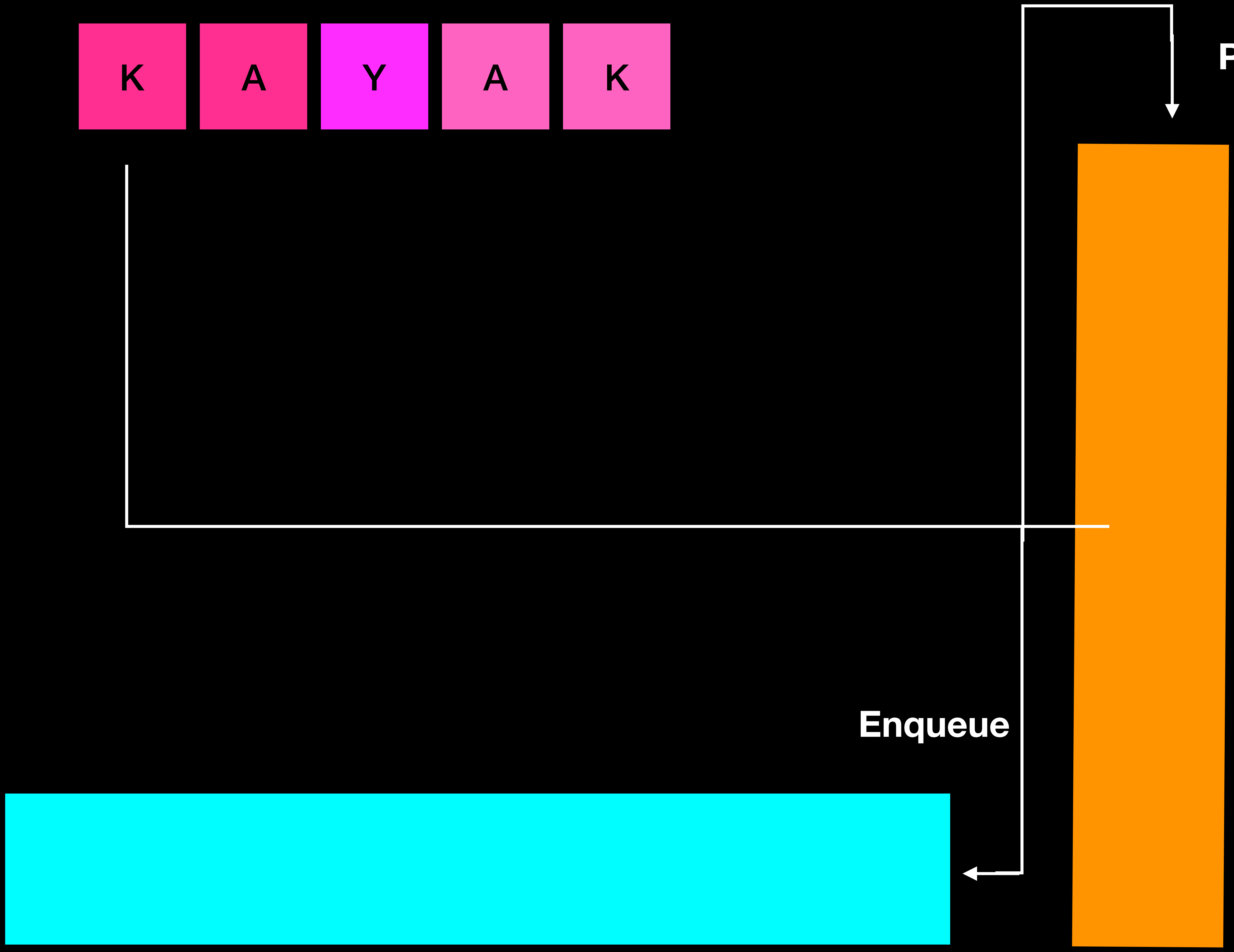
What if you have an incoming stream of characters, one at a time?

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)

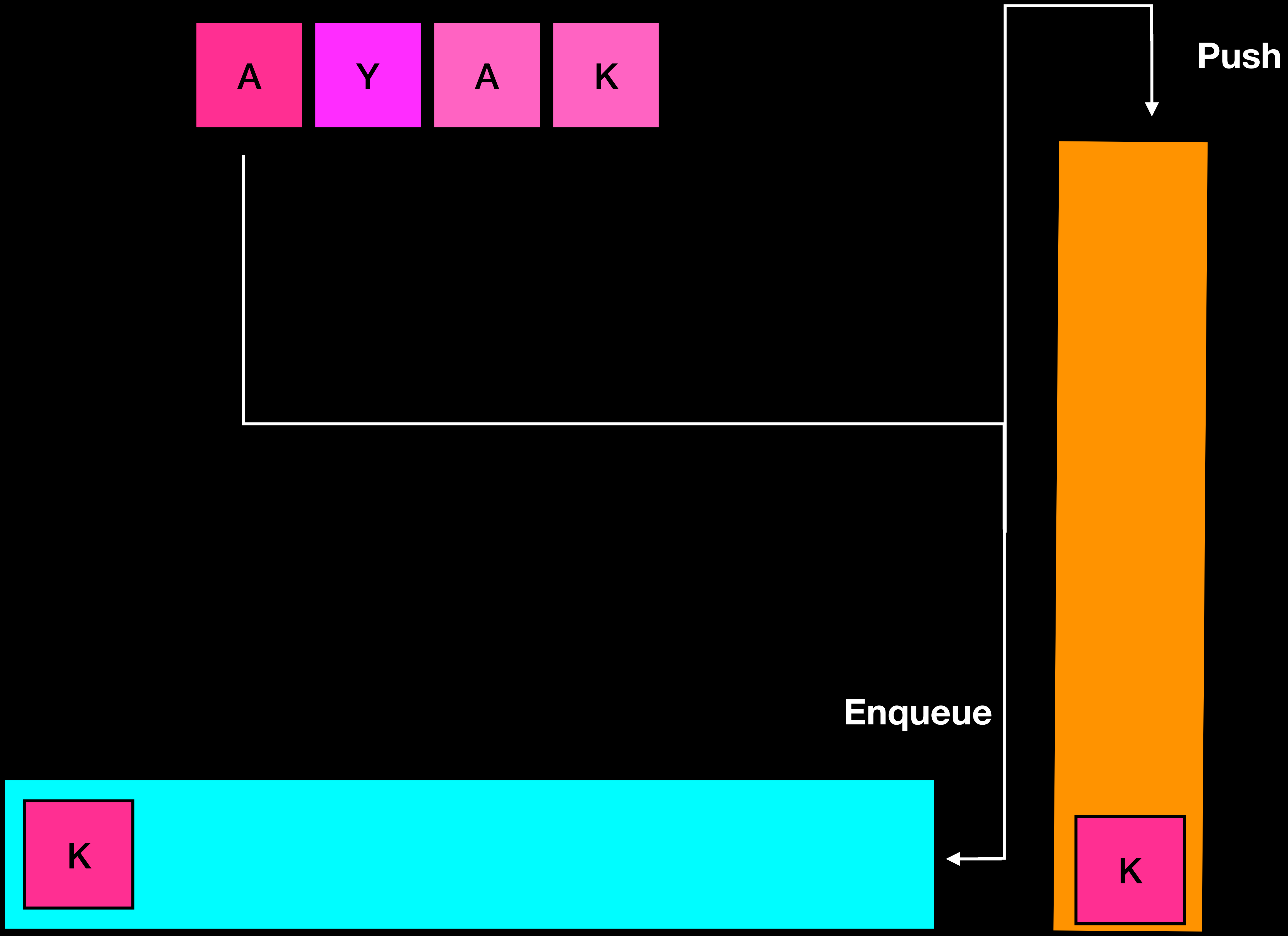
Algorithm: add incoming characters to both stack and queue and then compare to check if they are the same (palindrome!)



K A Y A K

Push

Enqueue



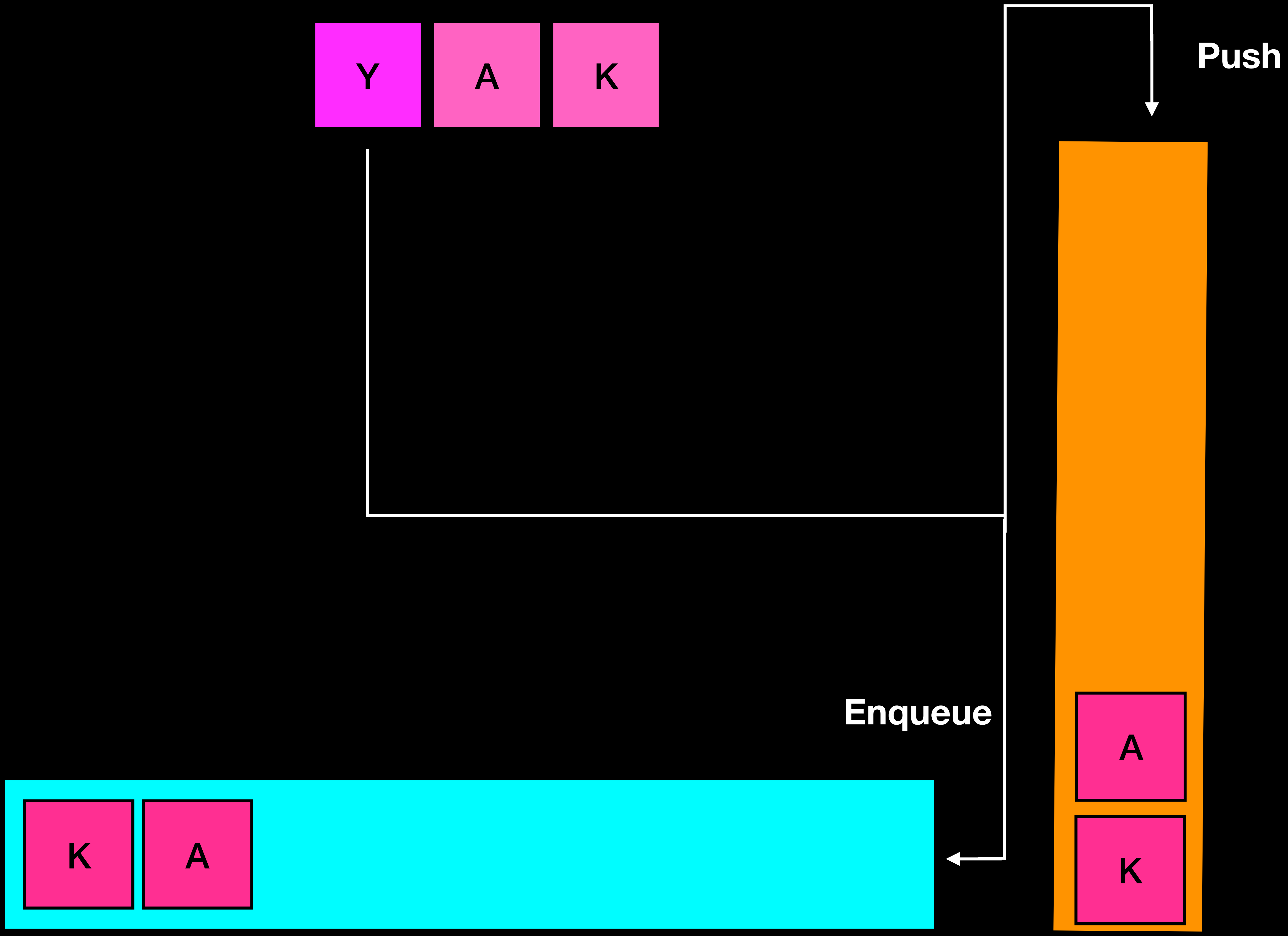
A Y A K

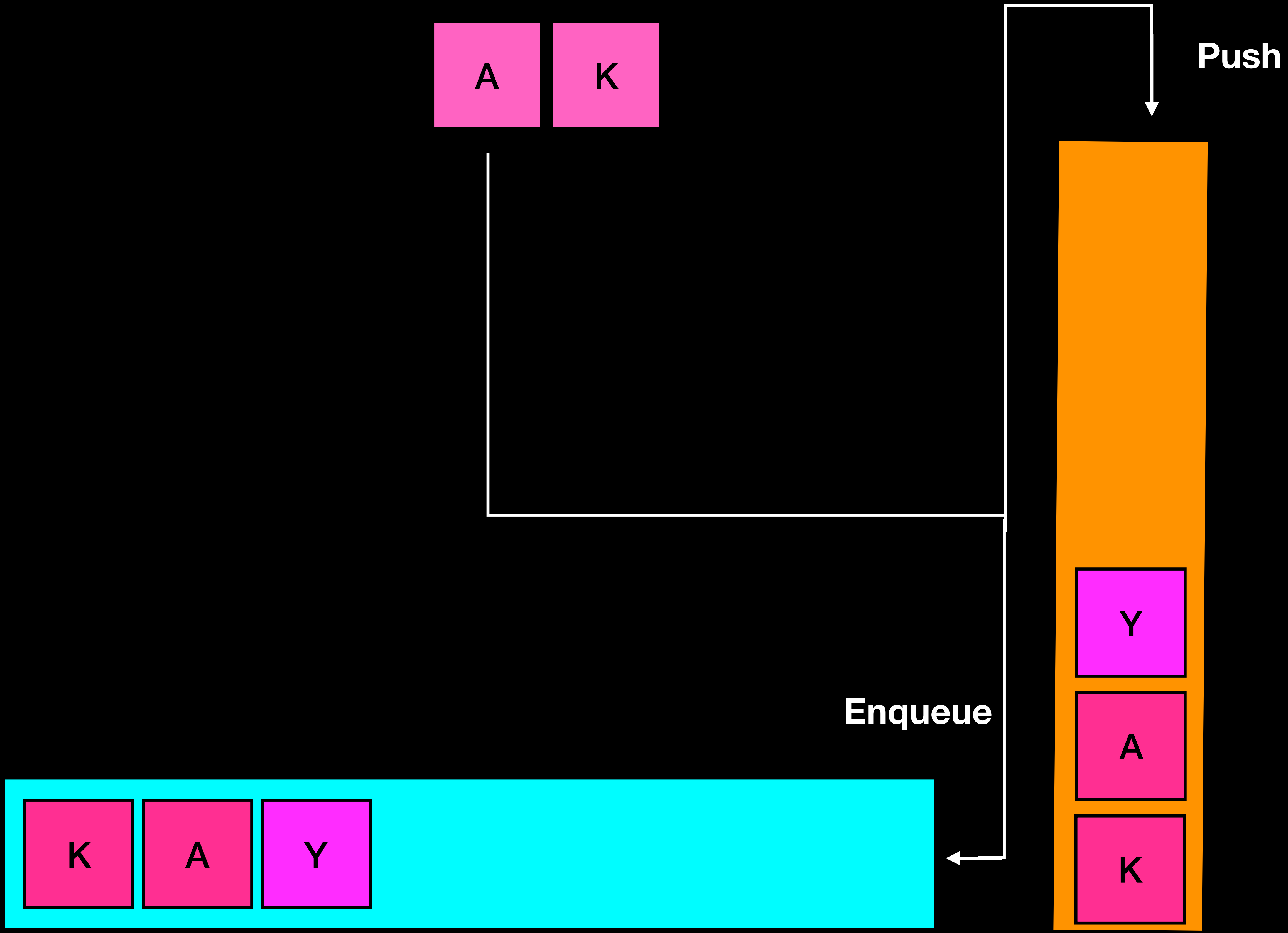
Push

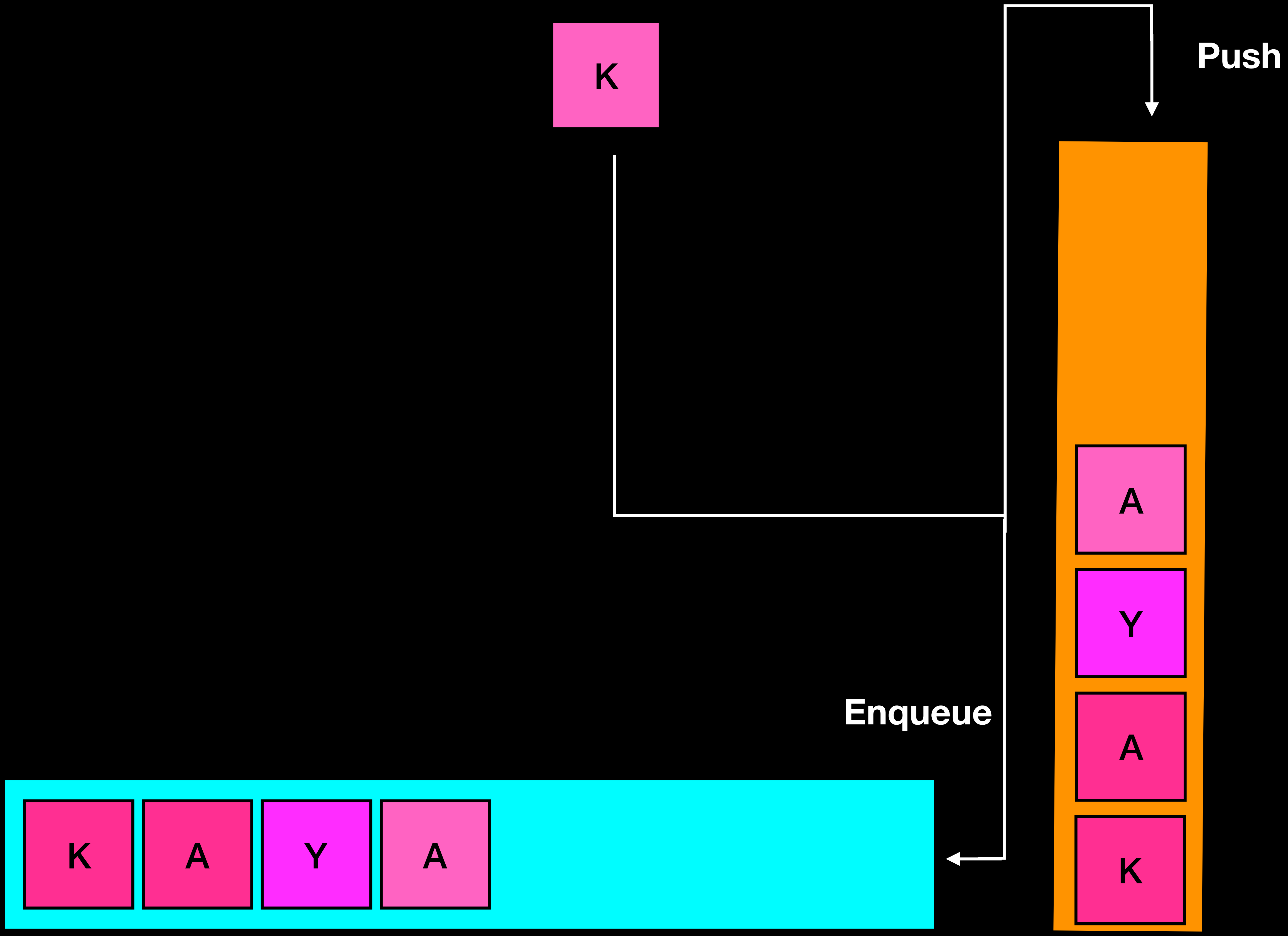
Enqueue

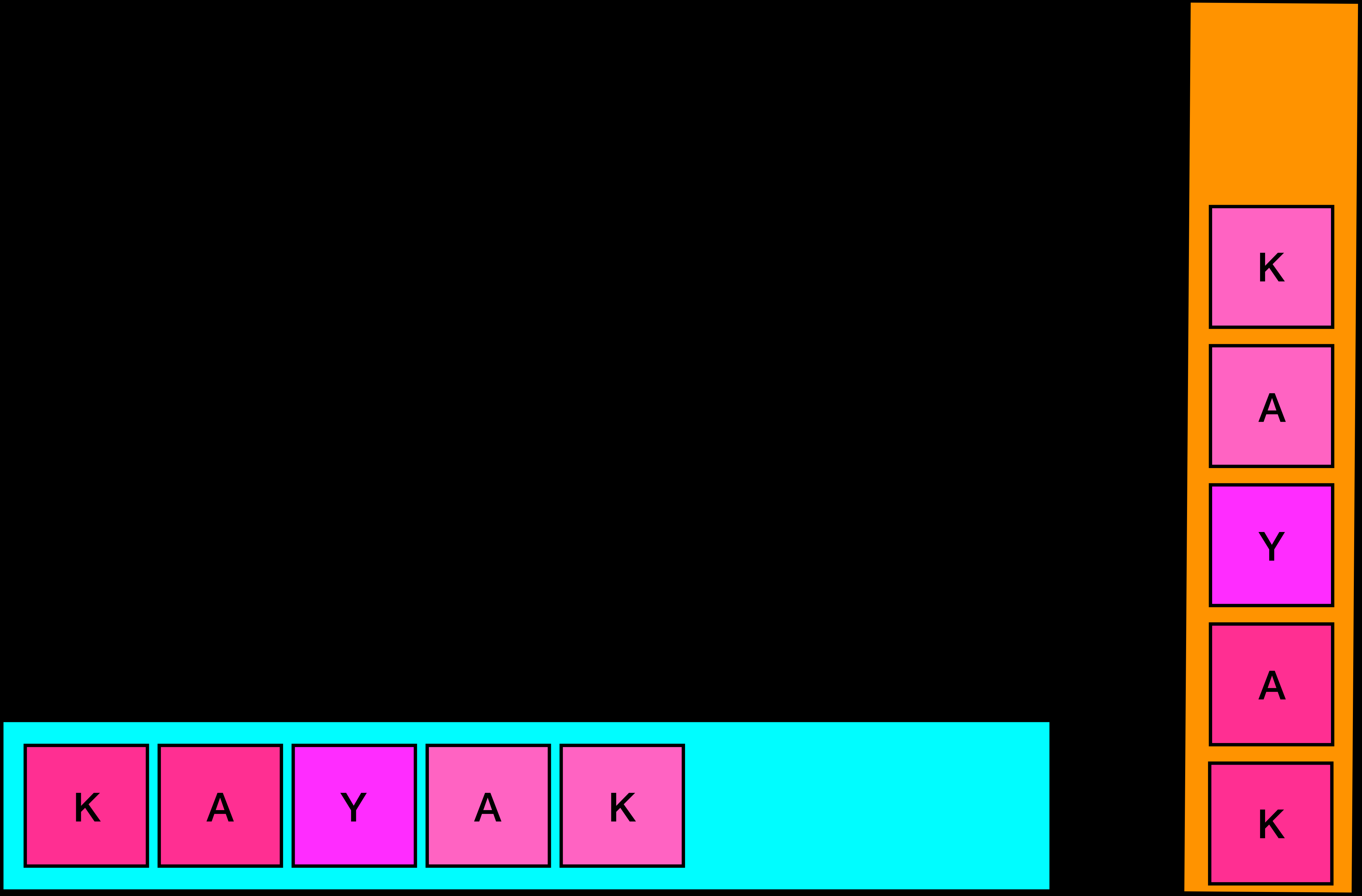
K

K


Y
A
K
Push
Enqueue
A
K
K
A

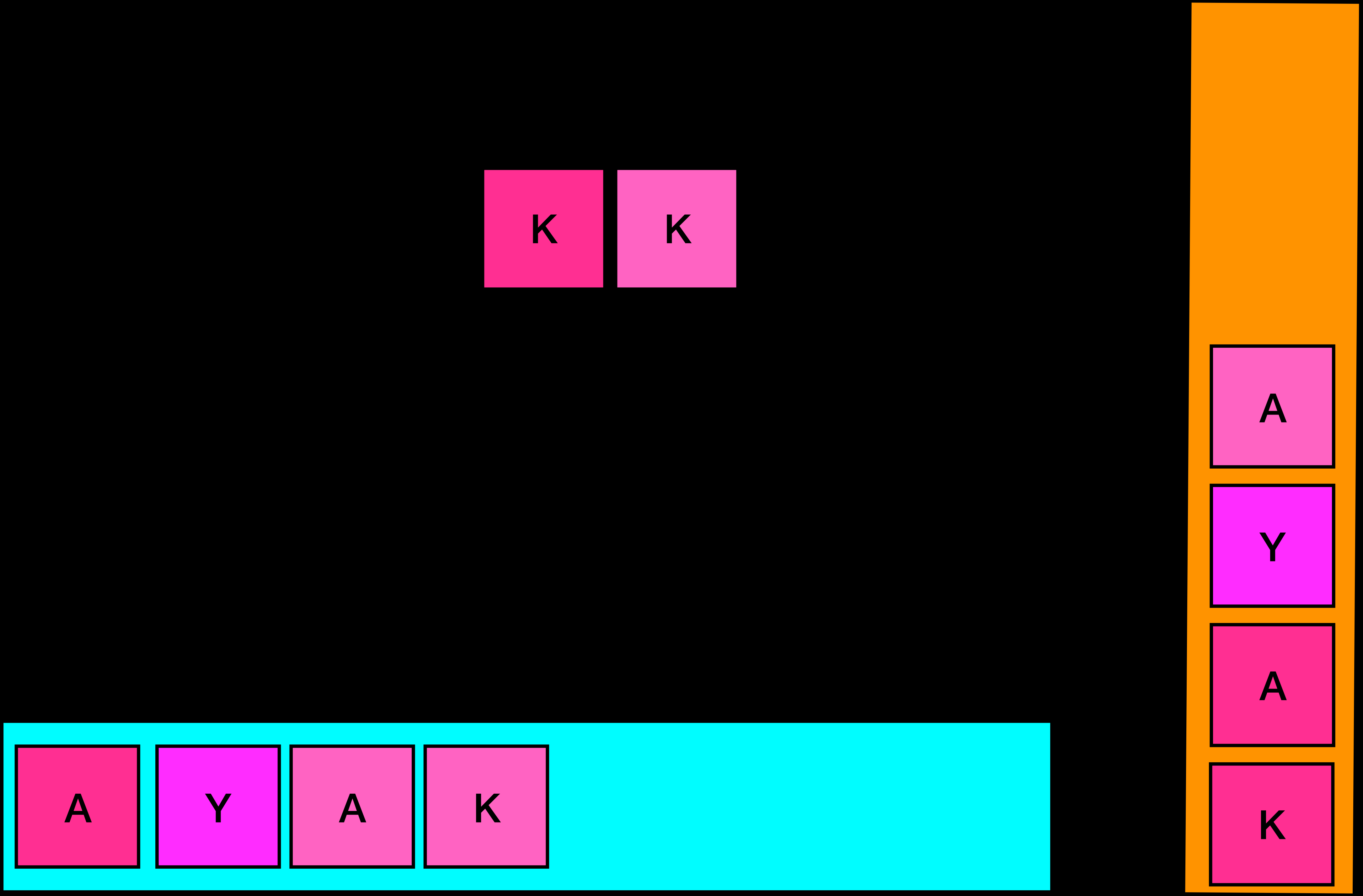

A
K
Push
Y
A
K
Enqueue
K
A
Y

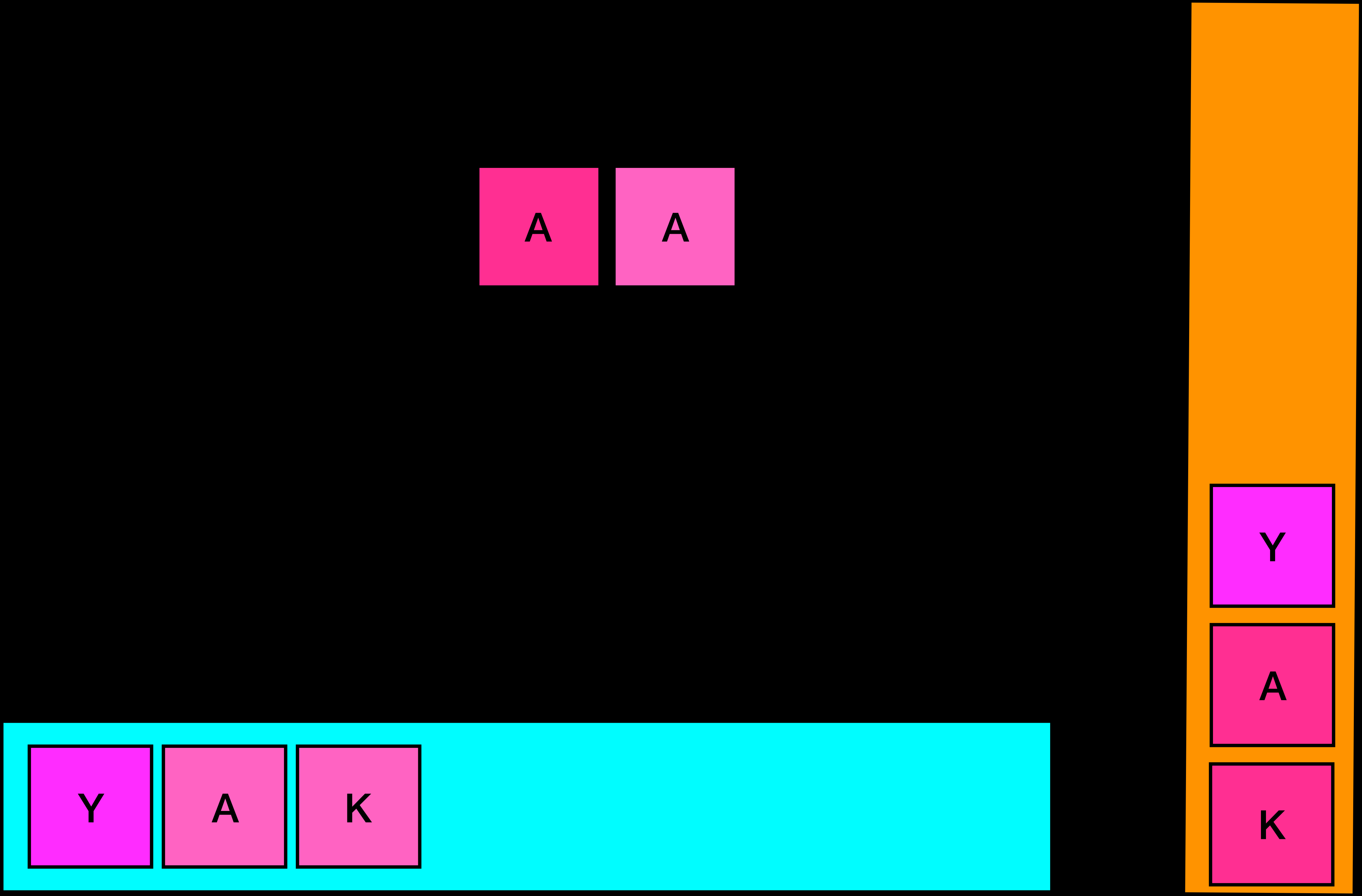

K
Push
A
Y
A
K
Enqueue
K
A
Y
A

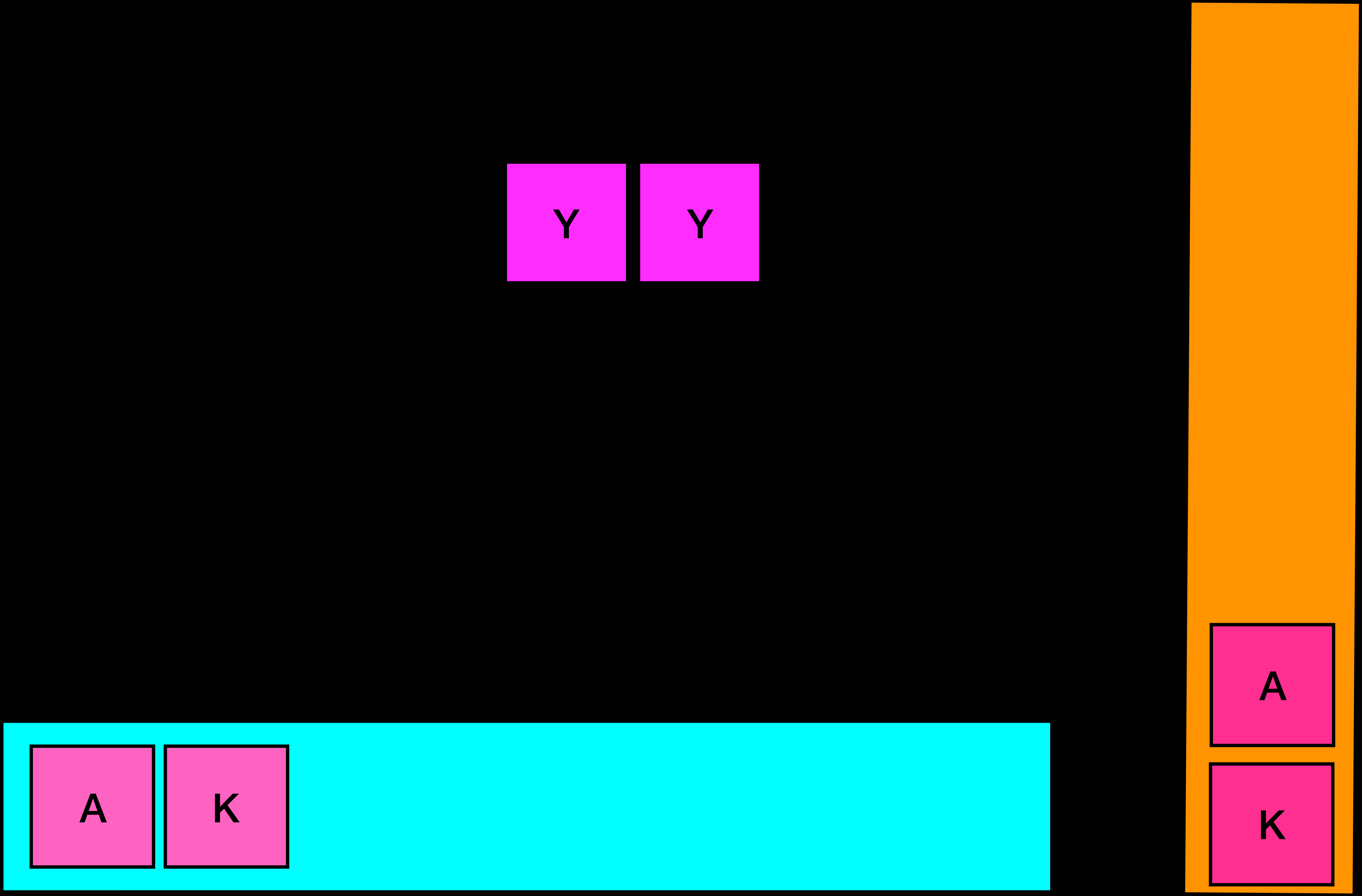


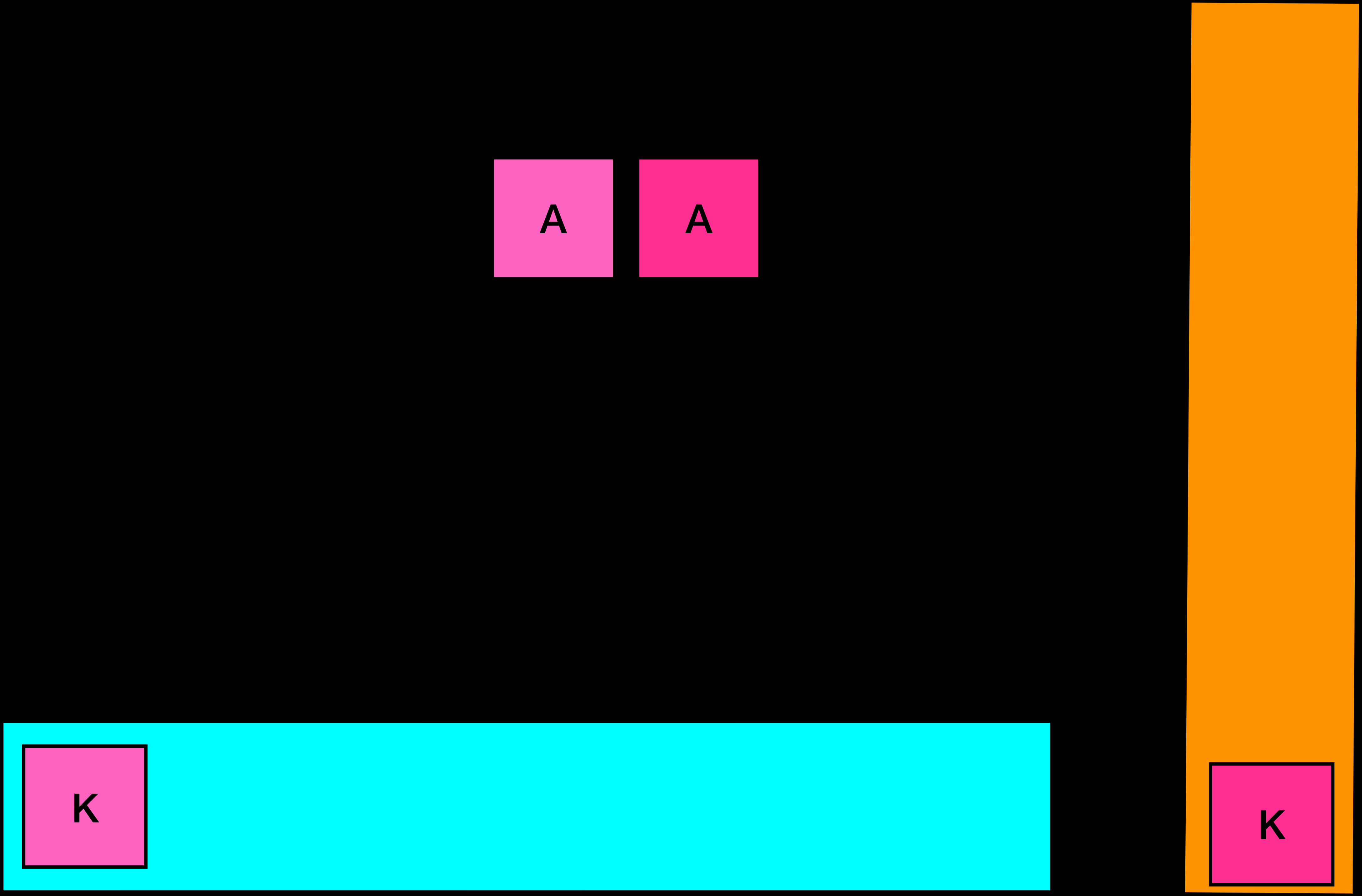
K A Y A K

K
A
Y
A
K


K
K
A
Y
A
K
A
Y
A
K


A
A
Y
A
K
Y
A
K

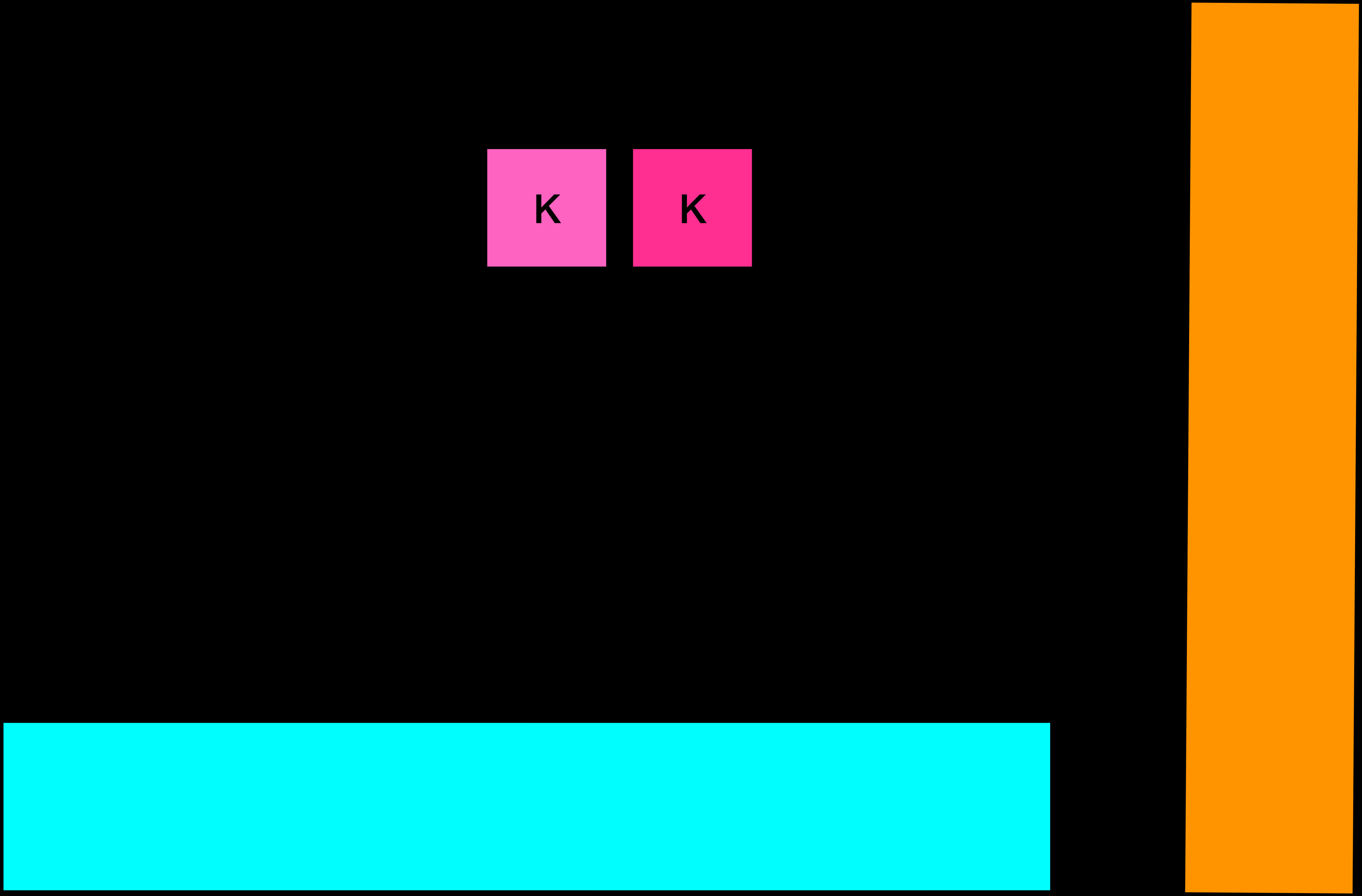

Y
Y
A
A
K
K

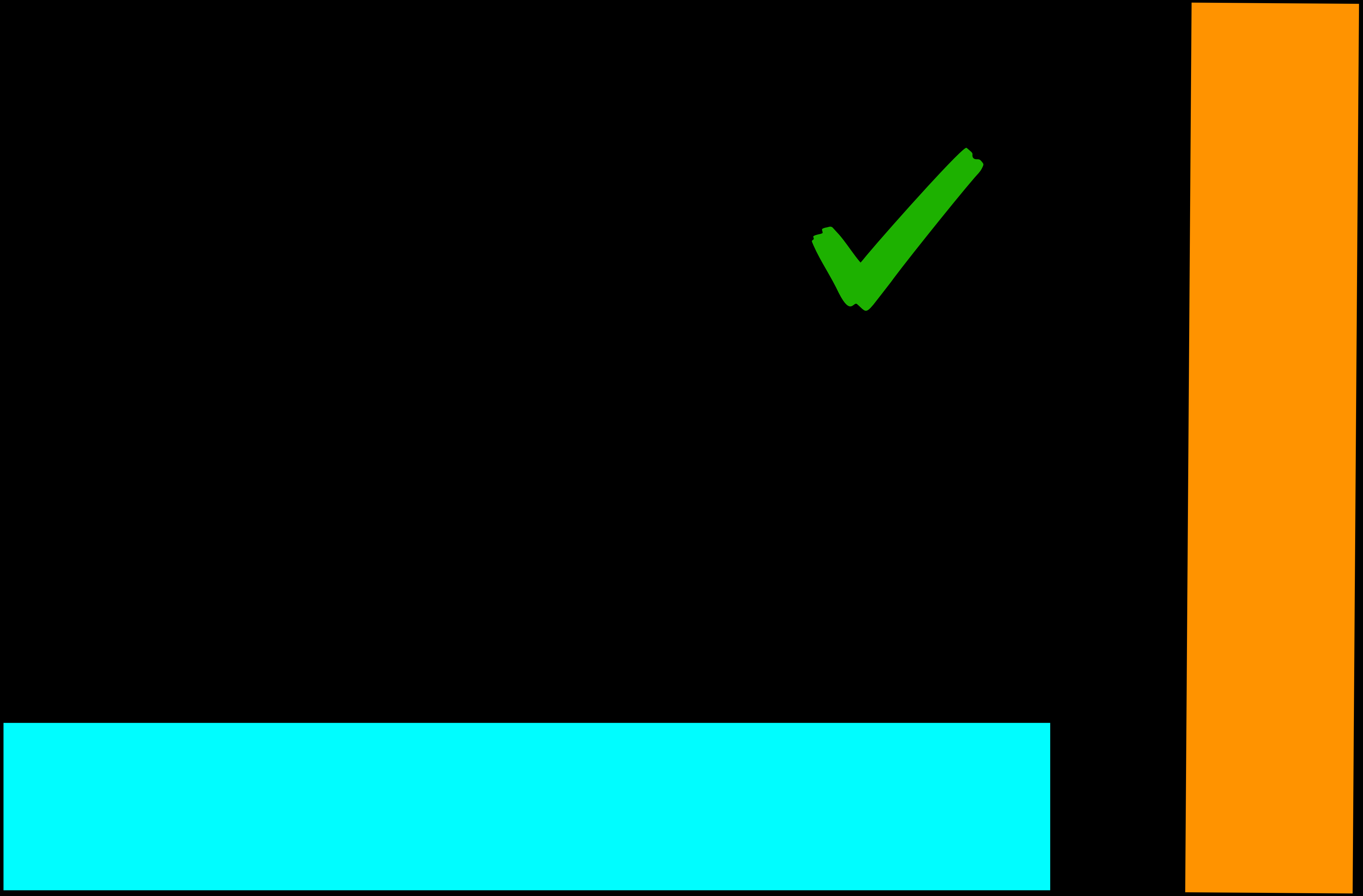


A A

K

K


K
K



```
bool isPalidrome()
{
    while(there are incoming characters)
        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
}
```

Analyze the worst-case time complexity of this algorithm

T(n) = ?

O(?)

```
bool isPalidrome()
{
    while(there are incoming characters)
        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
}
```

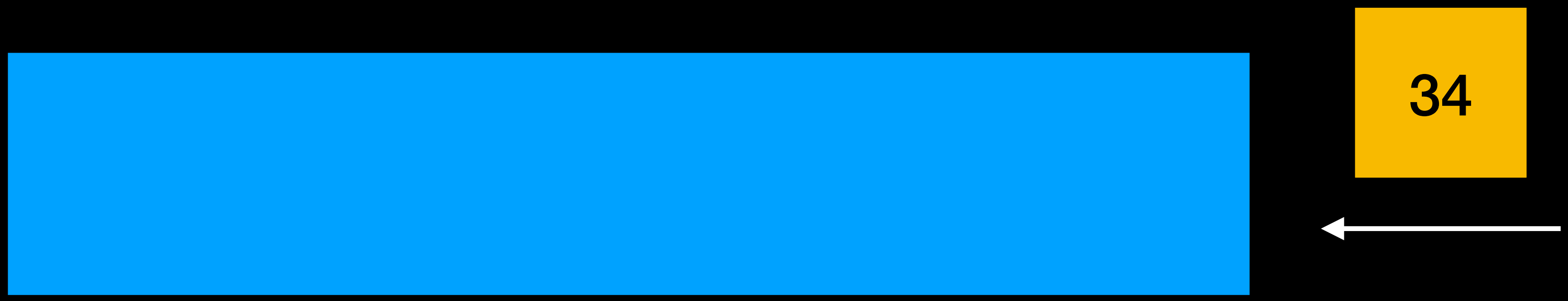
```
   bool isPalidrome()
   {
n      while(there are incoming characters)
           add character to both stack and queue

       caractersAreEqual = true

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

$T(n) = K_1 n + K_2$ $O(n)$

Deque

Double ended queue (deque)

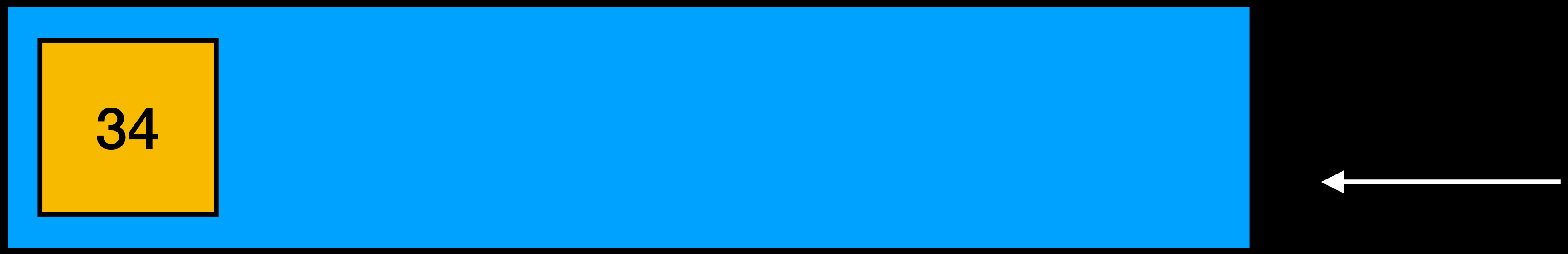
Can add and remove to/from front and back



Deque

Double ended queue (deque)

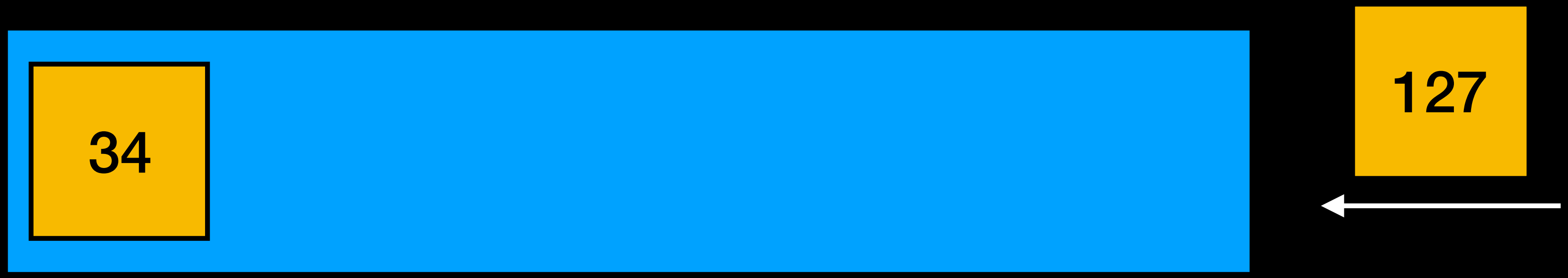
Can add and remove to/from front and back



Deque

Double ended queue (deque)

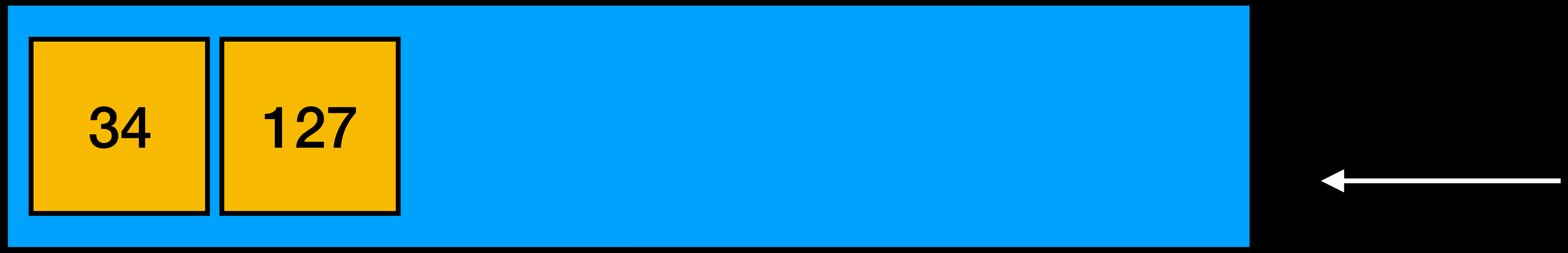
Can add and remove to/from front and back



Deque

Double ended queue (deque)

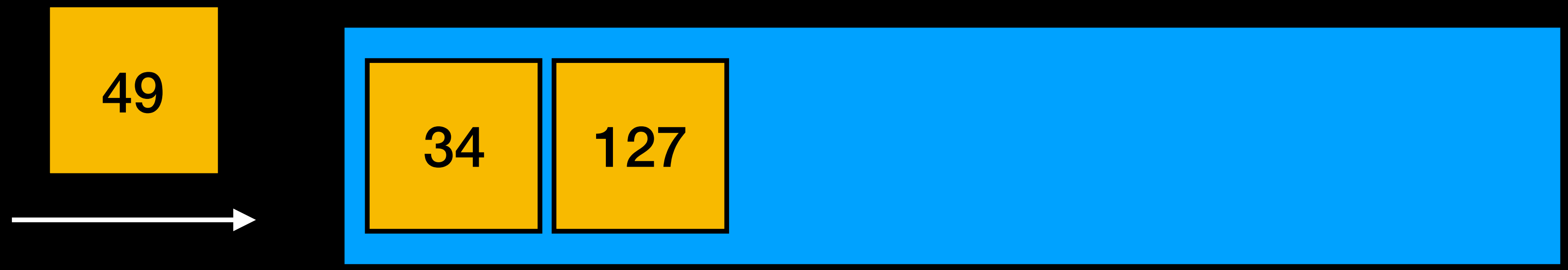
Can add and remove to/from front and back



Deque

Double ended queue (deque)

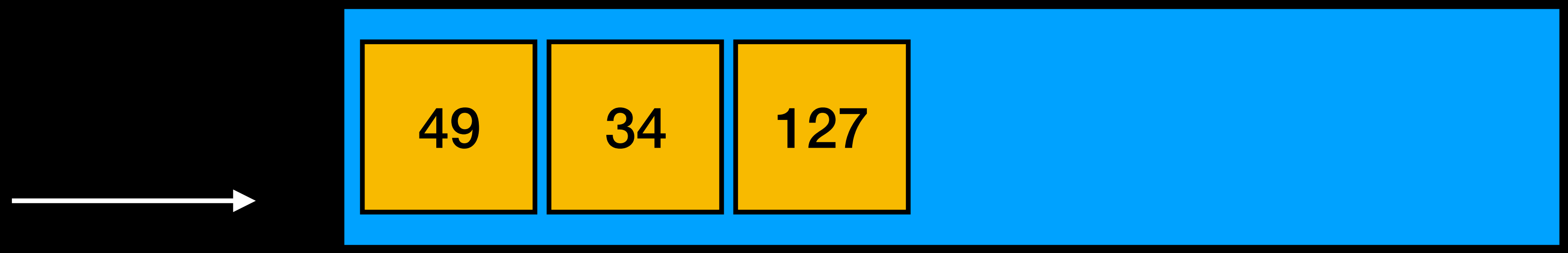
Can add and remove to/from front and back



Deque

Double ended queue (deque)

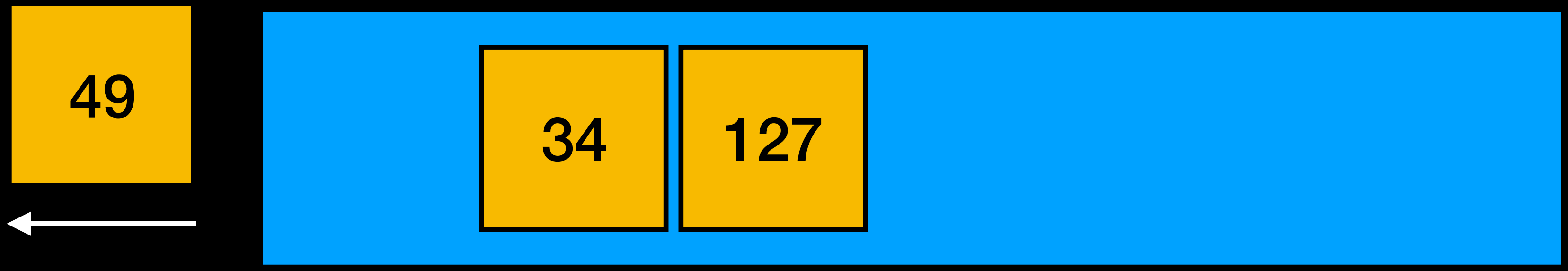
Can add and remove to/from front and back



Deque

Double ended queue (deque)

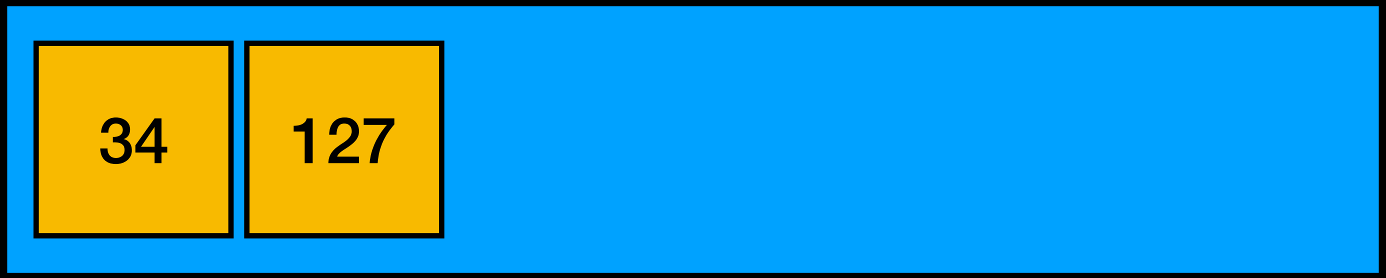
Can add and remove to/from front and back



Deque

Double ended queue (deque)

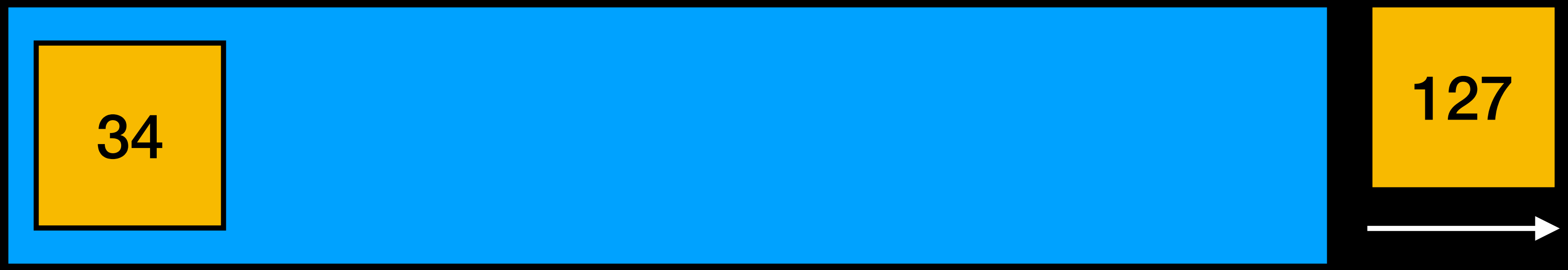
Can add and remove to/from front and back



Deque

Double ended queue (deque)

Can add and remove to/from front and back

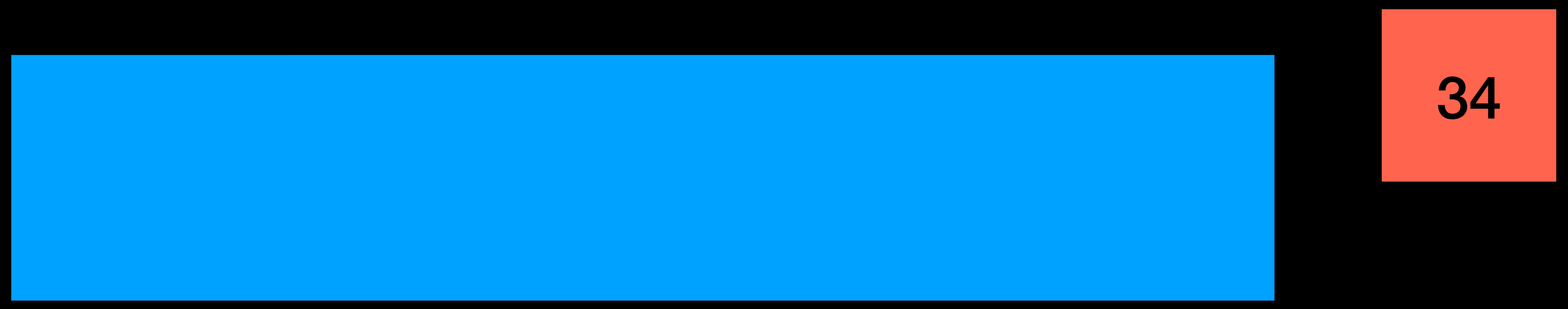


Priority Queue

Low Priority 🟥

High Priority 🟥

A queue of items “sorted” by priority

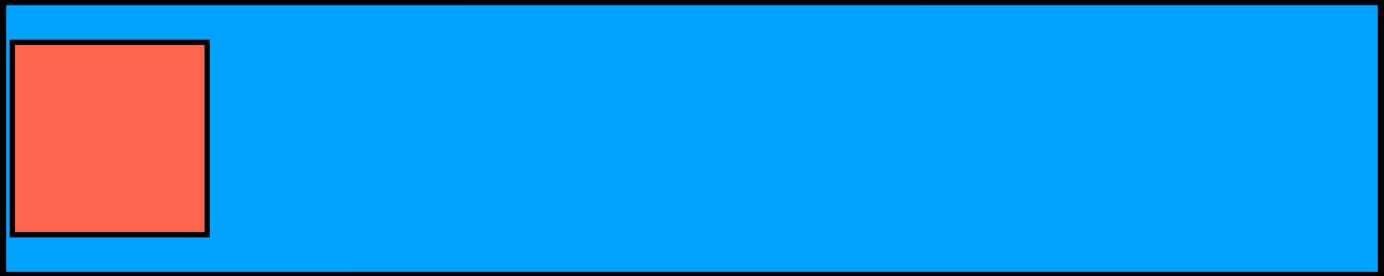


Priority Queue

Low Priority 🟥

High Priority 🟥

A queue of items “sorted” by priority

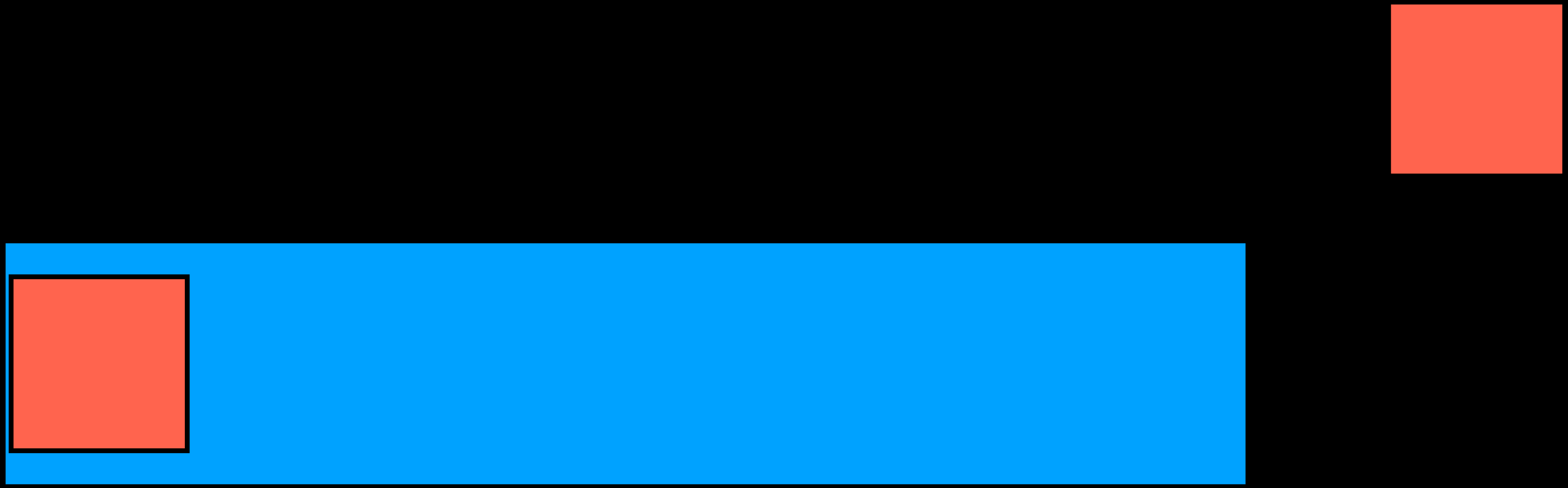


Priority Queue

Low Priority 🟧

High Priority 🟥

A queue of items “sorted” by priority

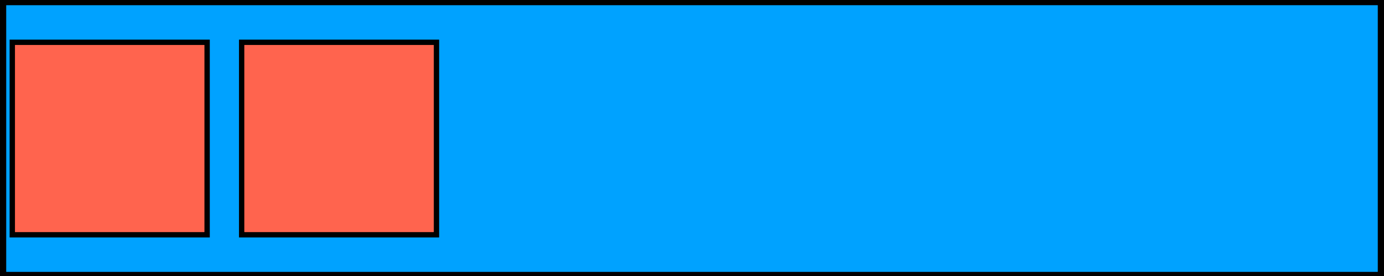


Priority Queue

Low Priority 🟧

High Priority 🟥

A queue of items “sorted” by priority

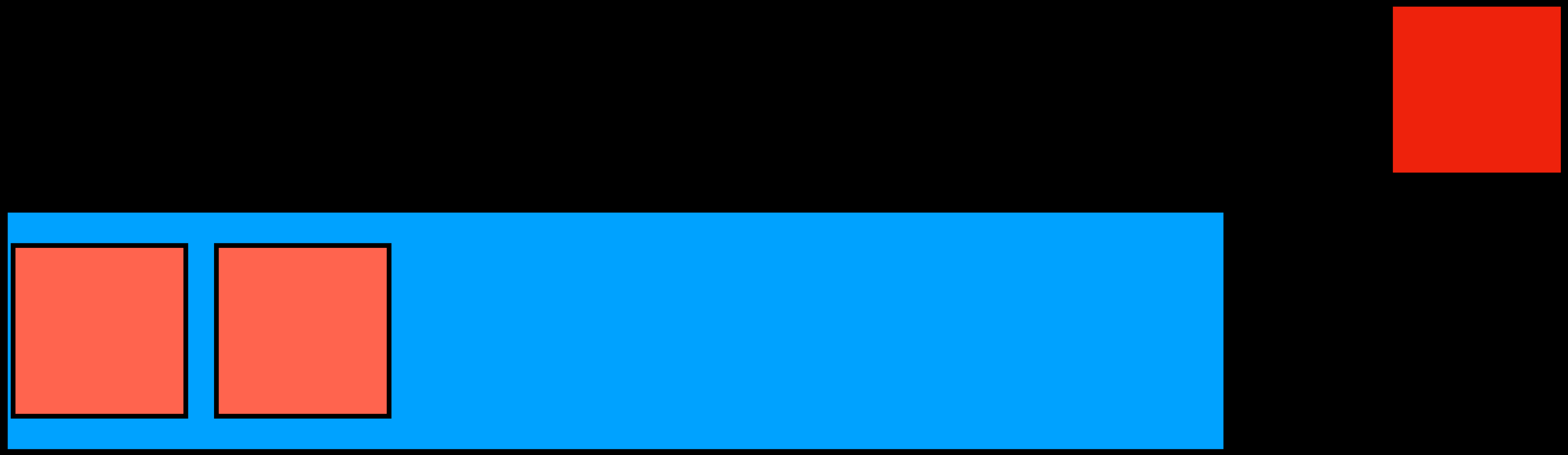


Priority Queue

Low Priority 🟥

High Priority 🟥

A queue of items “sorted” by priority

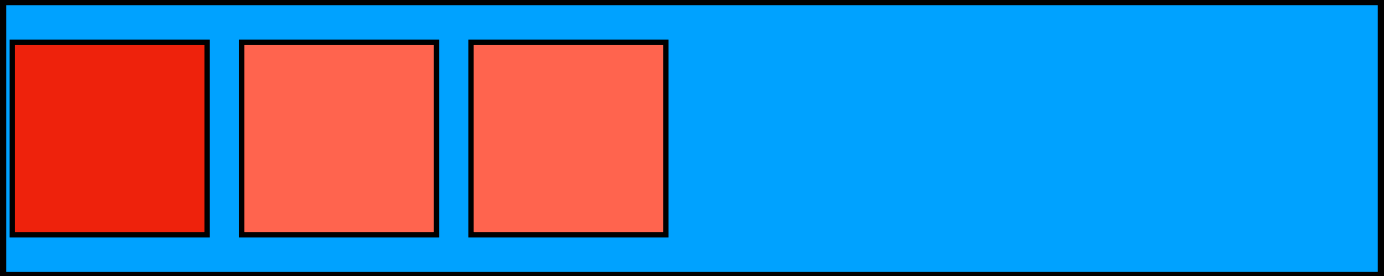


Priority Queue

Low Priority 🟥

High Priority 🟥

A queue of items “sorted” by priority

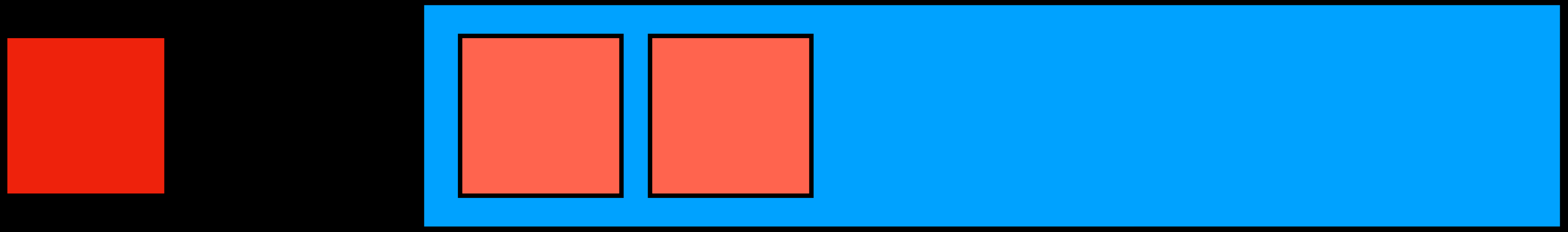


Priority Queue

Low Priority 🟥

High Priority 🟥

A queue of items “sorted” by priority



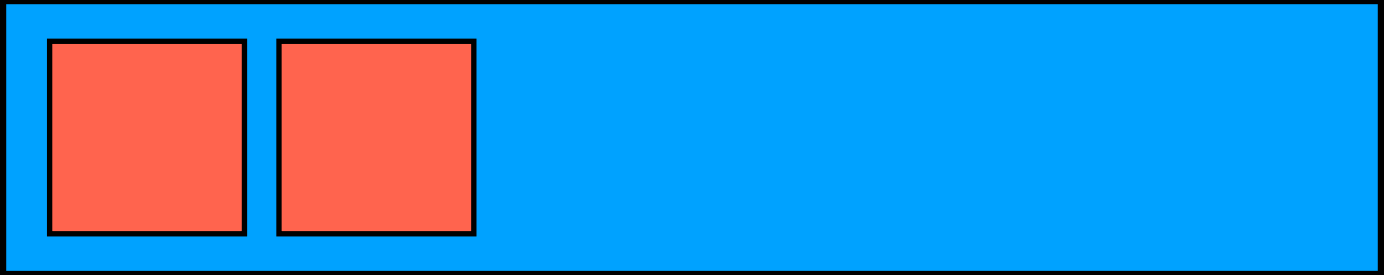
Priority Queue

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)