# Stack & Queue ADTs

Tiziana Ligorio
Hunter College of The City University of New York

# Today's Plan

ADT Recap

Stack ADT

Stack Applications

Queue ADT

Queue Applications

# ADT Recap

Abstract Data Types:

Bag (unordered)

List (ordered)

ADT operations

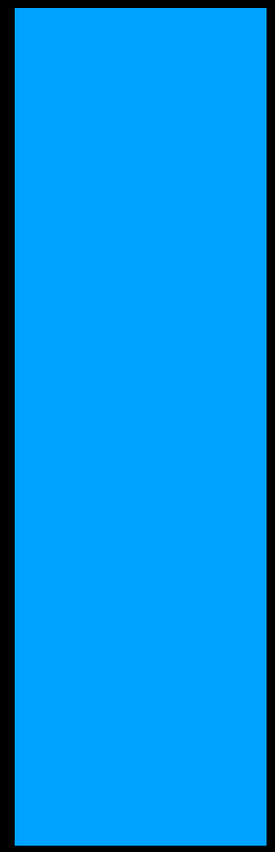
add/insert, remove, find

# Stack



An ADT representing a stack of items

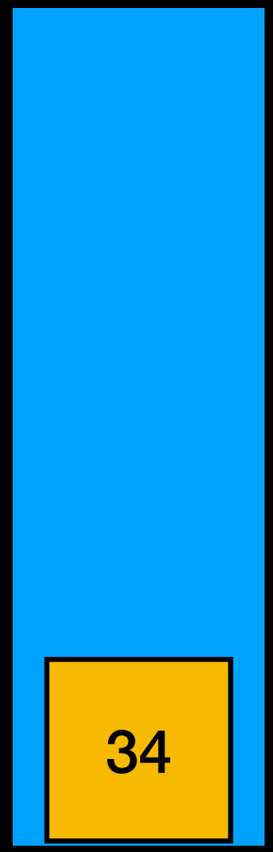
Objects can be pushed onto the stack or popped from the stack



# Stack

An ADT representing a stack of items

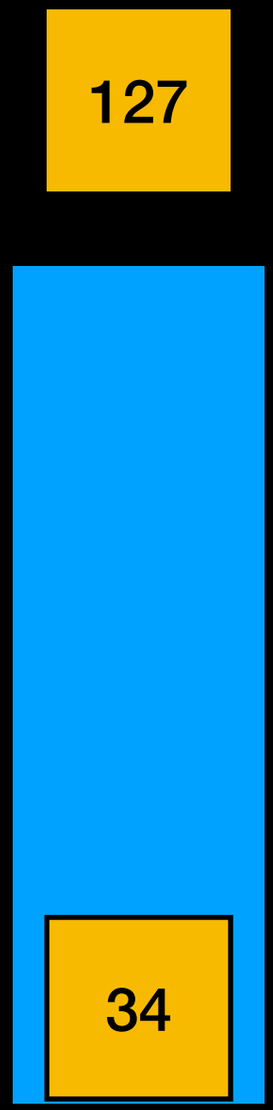
Objects can be pushed onto the stack or popped from the stack



# Stack

An ADT representing a stack of items

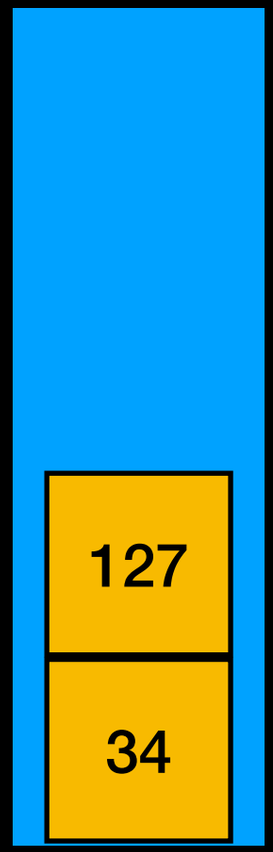
Objects can be pushed onto the stack or popped from the stack



# Stack

An ADT representing a stack of items

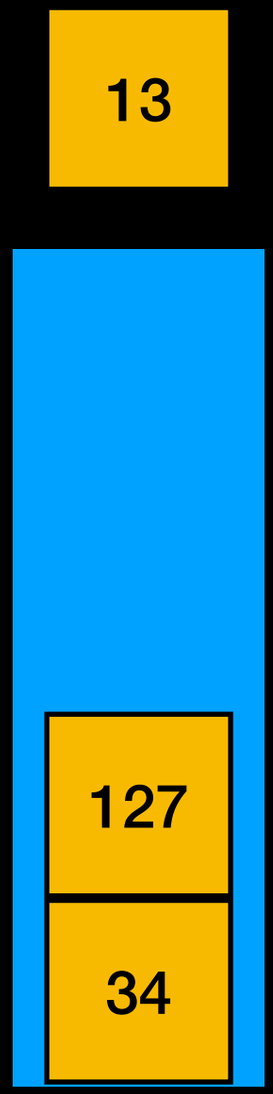
Objects can be pushed onto the stack or popped from the stack



# Stack

An ADT representing a stack of items

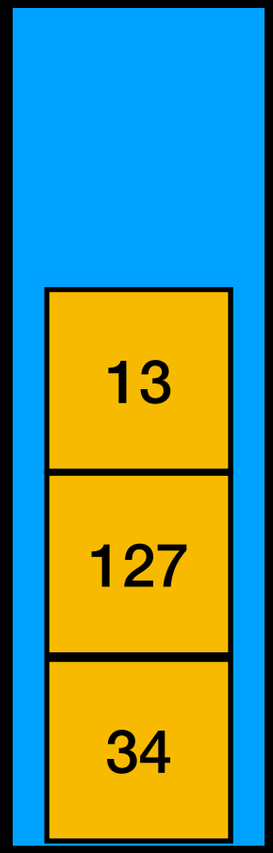
Objects can be pushed onto the stack or popped from the stack



# Stack

An ADT representing a stack of items

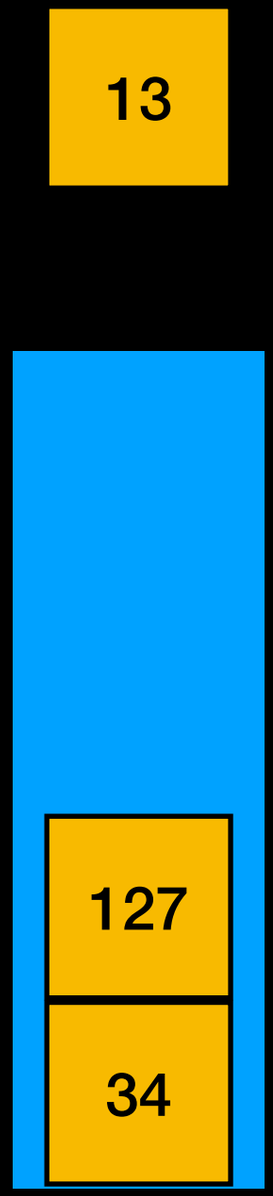
Objects can be pushed onto the stack or popped from the stack



# Stack

An ADT representing a stack of items

Objects can be pushed onto the stack or popped from the stack



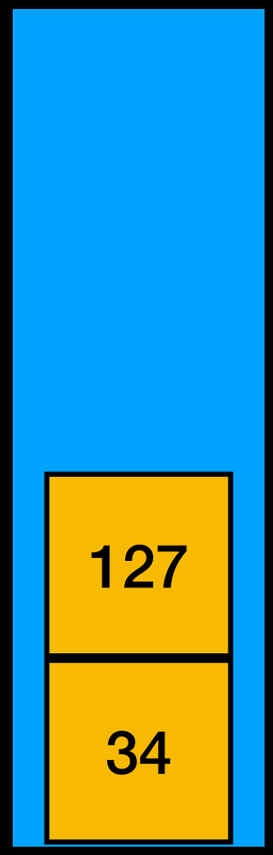
# Stack

An ADT representing a stack of items

Objects can be pushed onto the stack or popped from the stack

**LIFO:** Last In First Out

Only top of stack is accessible (top), no other objects on the stack are visible



# Applications

Very simple structure

Many applications:

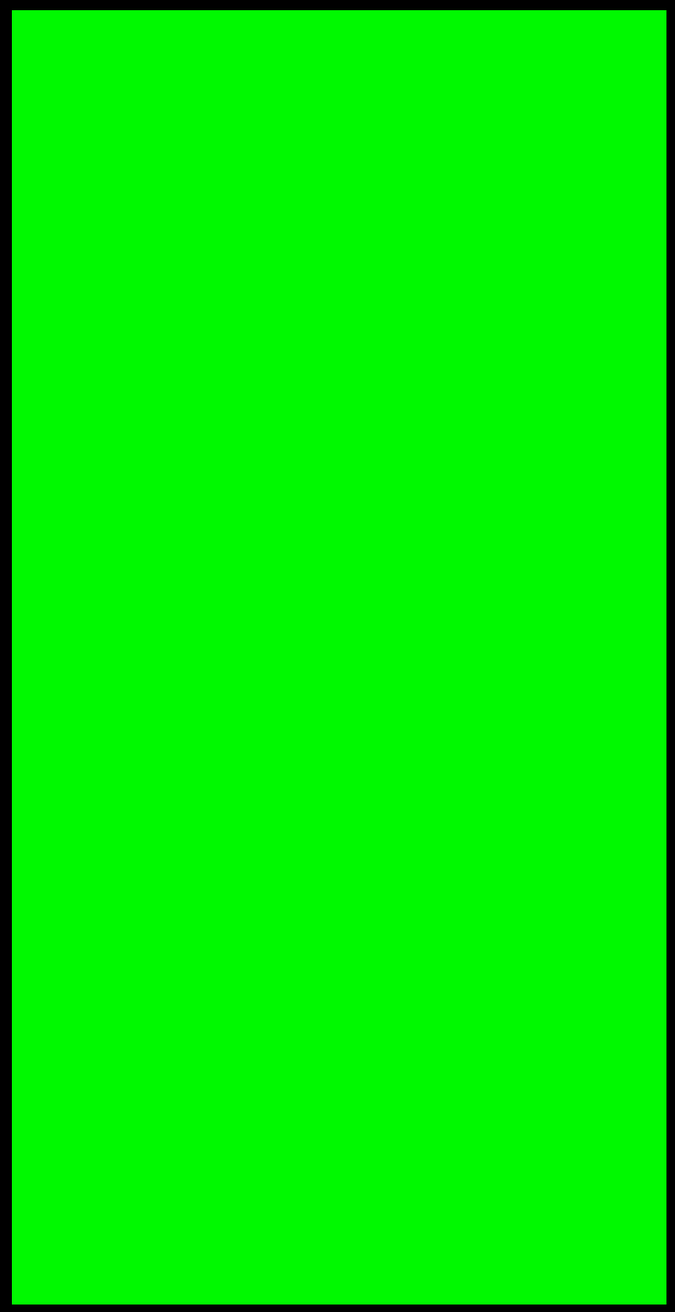
program stack
balancing parenthesis
evaluating postfix expressions
backtracking
. . . and more

# Program Stack

# Program Stack

```
1  void f(int x, int y)
2  {
3     int a;
4     // stuff here
5     if(a<13)
6        a = g(a);
7     // stuff here
8  }

9  int g(int z)
10 {
11    int p ,q;
12    // stuff here
13    return q;
14 }
```

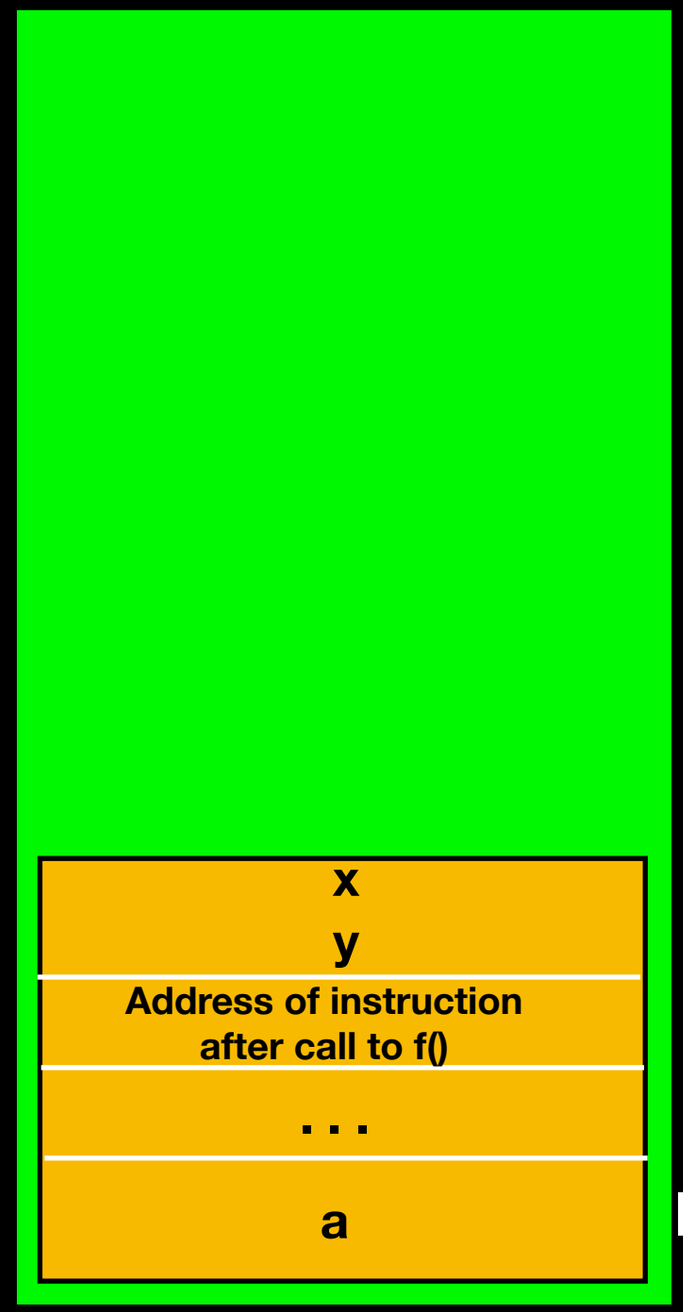


# Program Stack

```
1   void f(int x, int y)
2   {
3     int a;
4     // stuff here
5     if(a<13)
6       a = g(a);
7     // stuff here
8   }

9   int g(int z)
10  {
11    int p ,q;
12    // stuff here
13    return q;
14  }
```

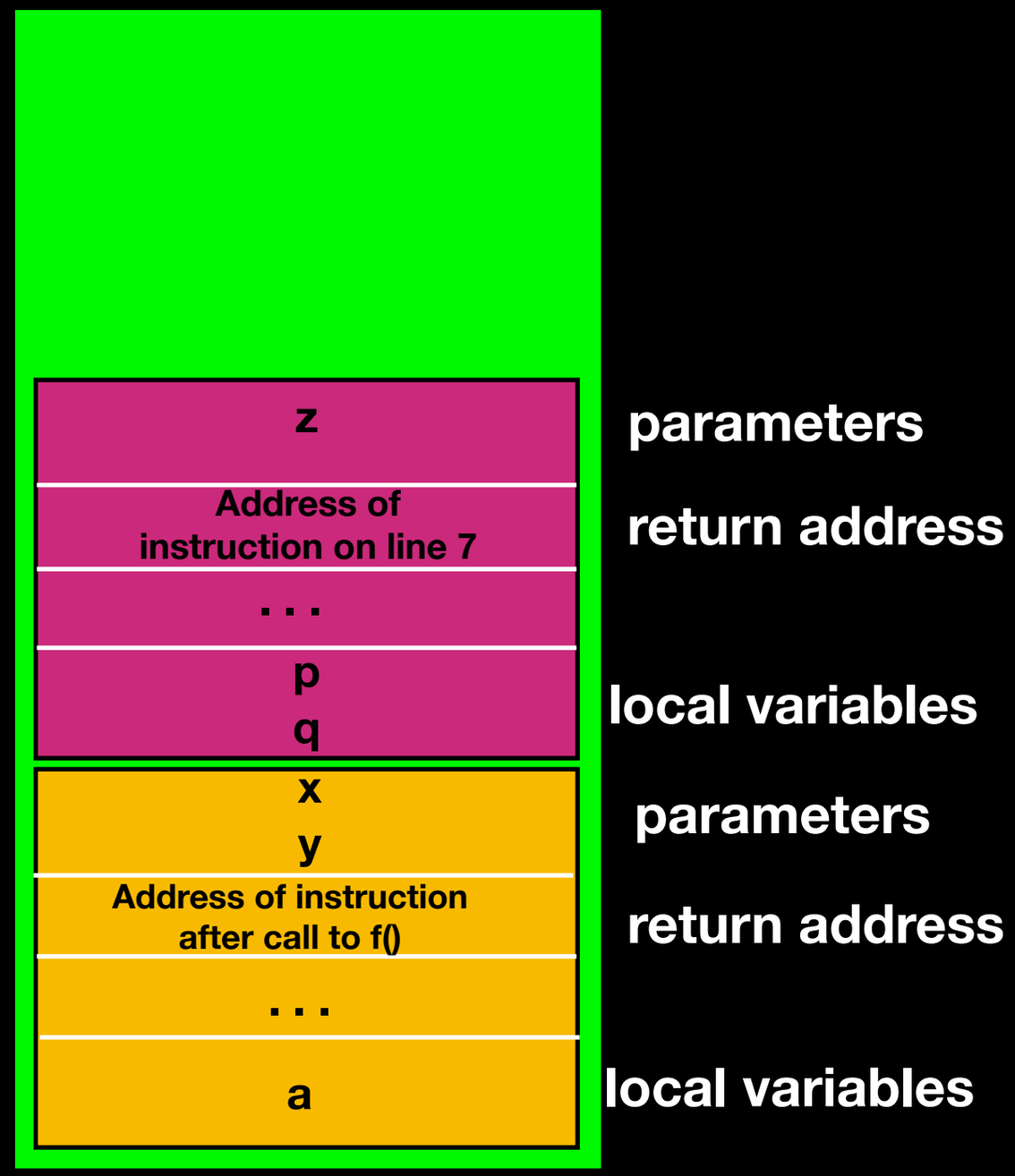
**Stack Frame for f()**



# Program Stack

```
1   void f(int x, int y)
2   {
3      int a;
4      // stuff here
5      if(a<13)
6          a = g(a);
7      // stuff here
8   }

9   int g(int z)
10  {
11     int p ,q;
12     // stuff here
13     return q;
14  }
```

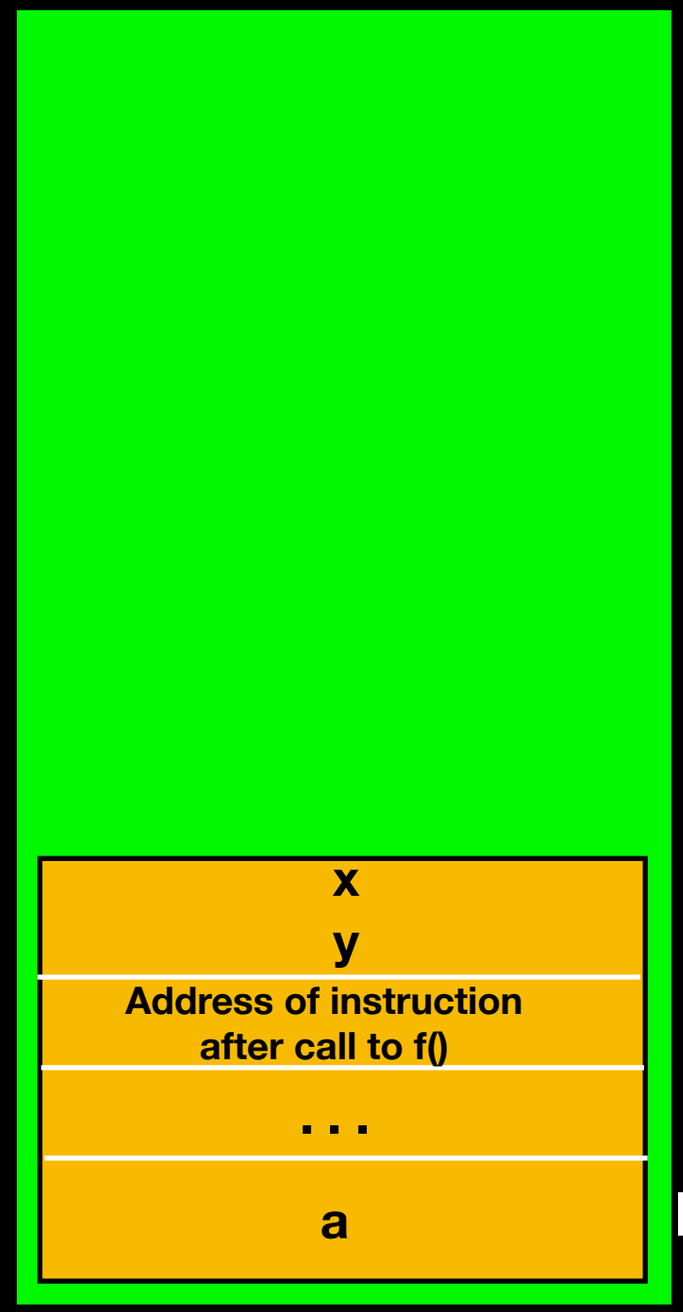


# Program Stack

```
1  void f(int x, int y)
2  {
3     int a;
4     // stuff here
5     if(a<13)
6        a = g(a);
7     // stuff here
8  }

9  int g(int z)
10 {
11    int p ,q;
12    // stuff here
13    return q;
14 }
```

**Stack Frame for f()**



How would you solve it?

# Balancing Parentheses

Given a string,  determine if parenthesis are balanced.

Parentheses can be { }, [ ] or ( ), and must be nested properly.

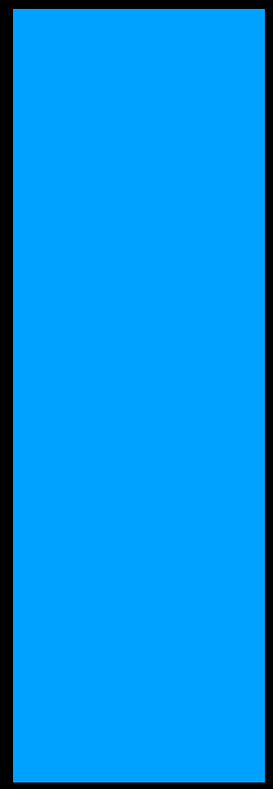
E.g. “ [ ( { } ) ] “ is balanced, while “ [ ( { }  ] “ or “ [ ( {  ) } ] “ are not.

Typical applications: parsers and compilers.

# Balancing Parentheses

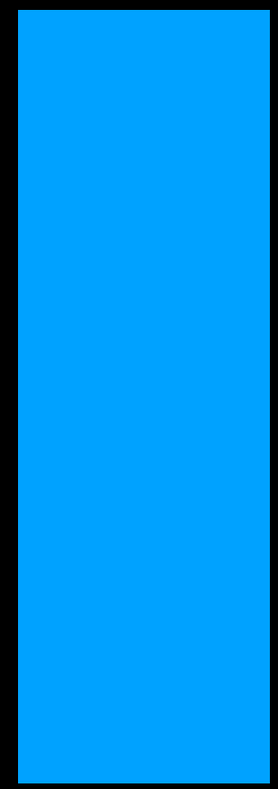
```
int f(){if(x*(y+z[i])<47){x += y}}
```





# Balancing Parentheses

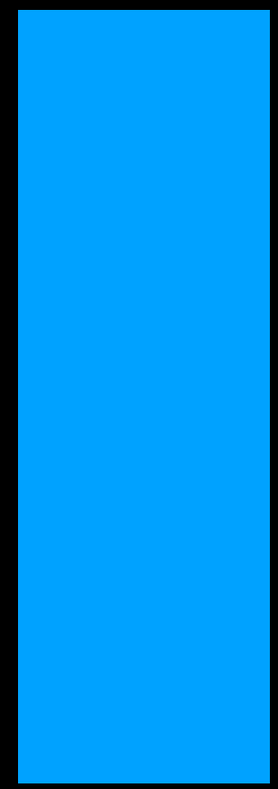
```
int f(){if(x*(y+z[i])<47){x += y}}
```



# Balancing Parentheses

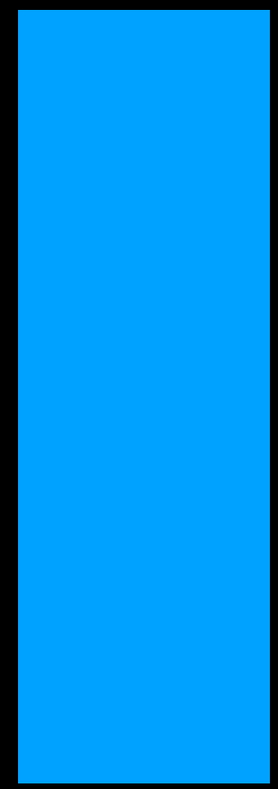
```
int f(){if(x*(y+z[i])<47){x += y}}
```





# Balancing Parentheses

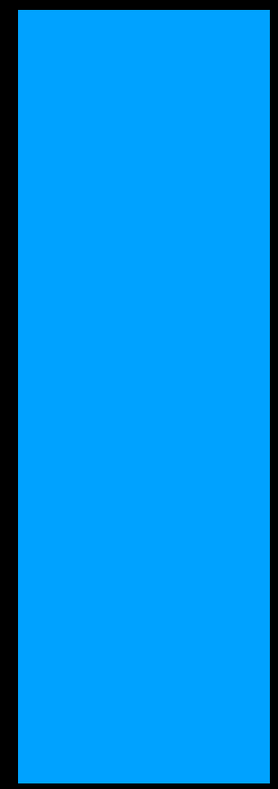
```
int f(){if(x*(y+z[i])<47){x += y}}
```



# Balancing Parentheses

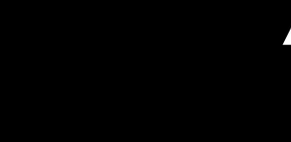
```
int f(){if(x*(y+z[i])<47){x += y}}
```

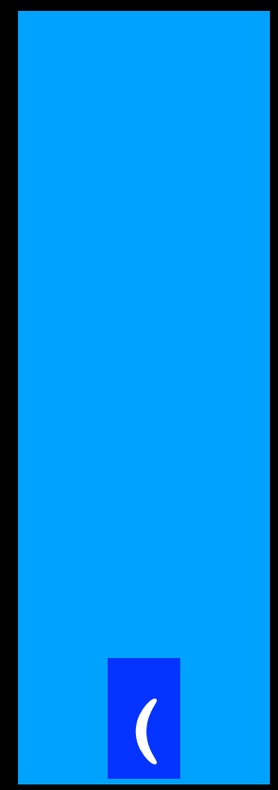




# Balancing Parentheses

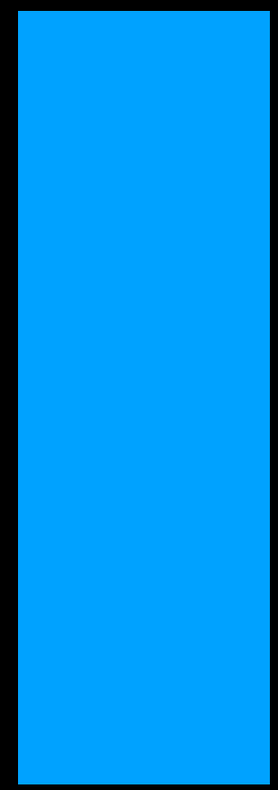
```
int f(){if(x*(y+z[i])<47){x += y}}
```





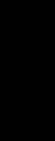
# Balancing Parentheses

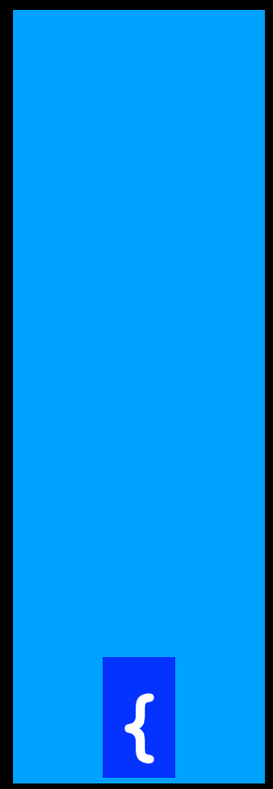
```
int f(){if(x*(y+z[i])<47){x += y}}
```



# Balancing Parentheses

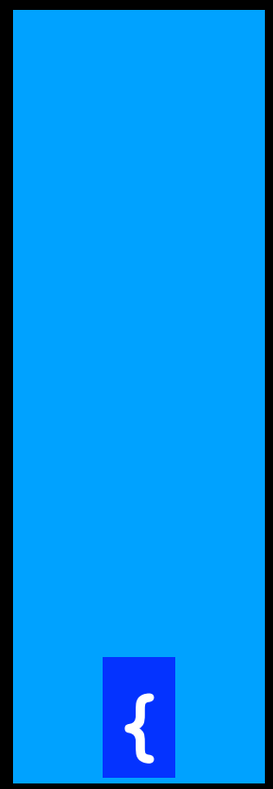
```
int f(){if(x*(y+z[i])<47){x += y}}
```





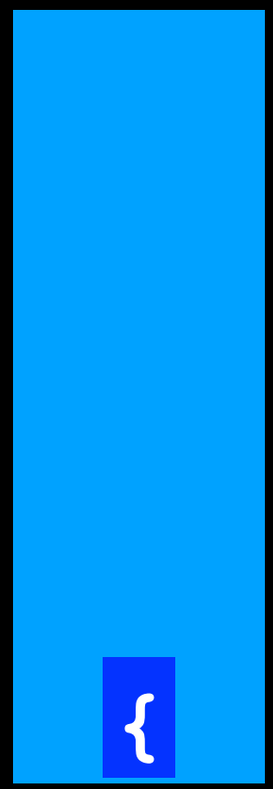
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



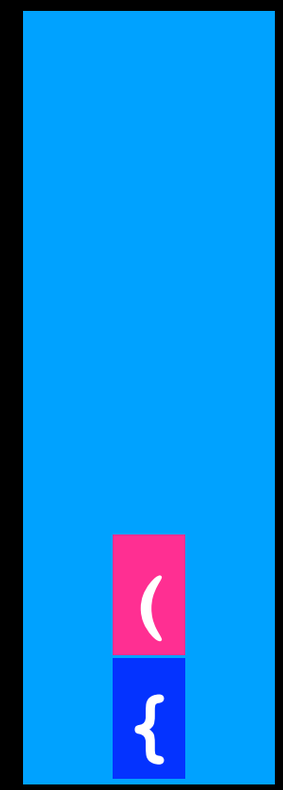
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



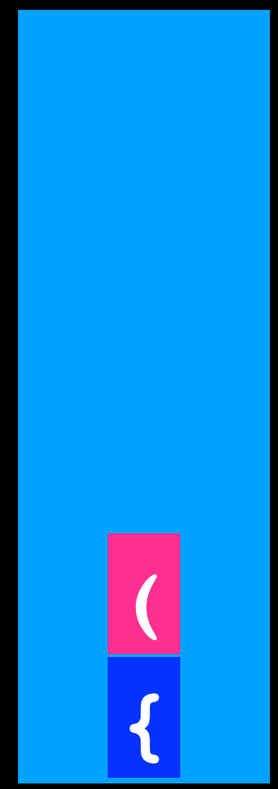
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



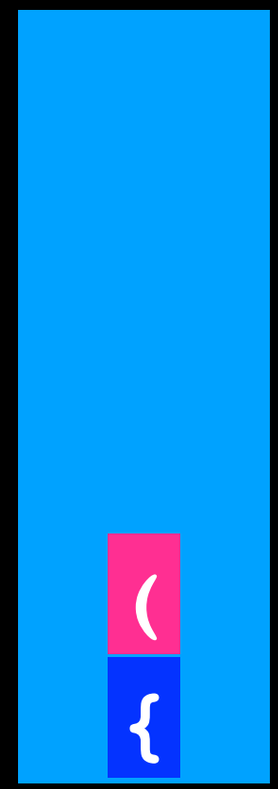
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



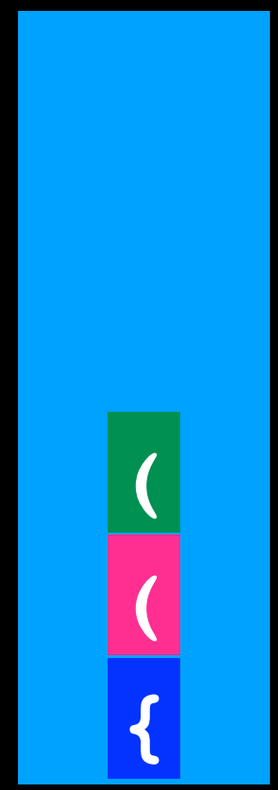
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



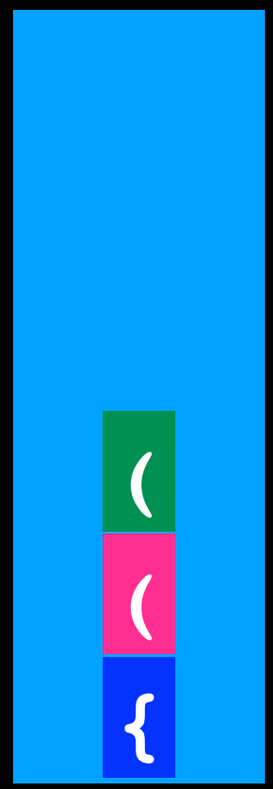
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



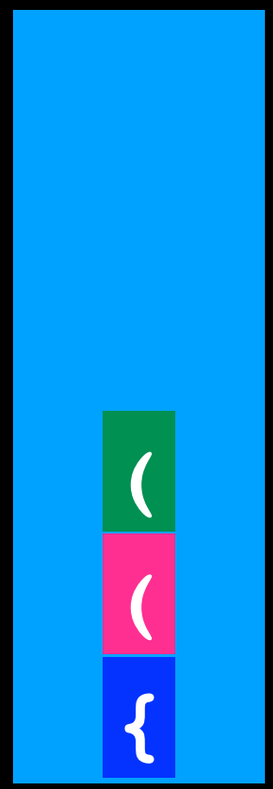
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



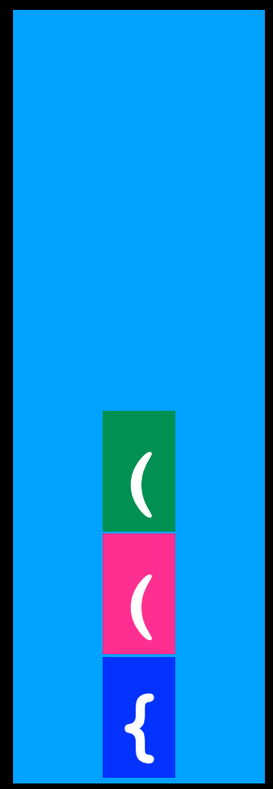
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



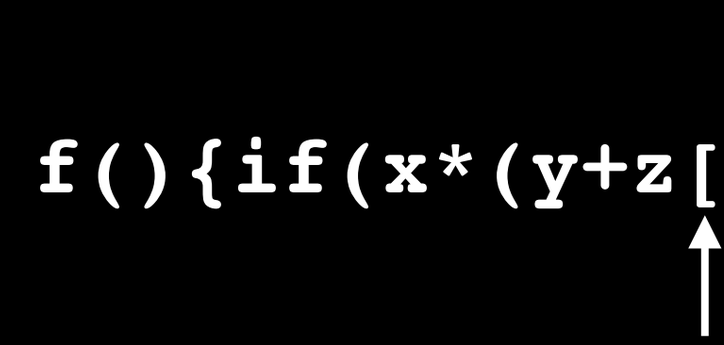
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```

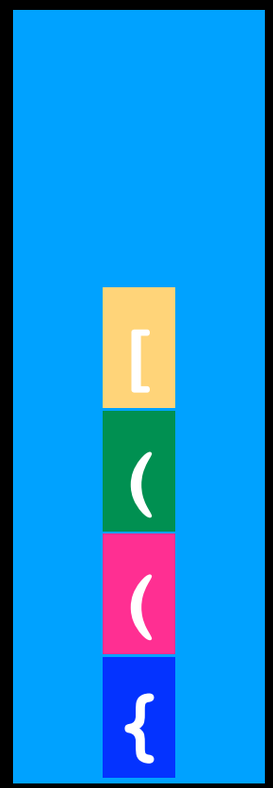


# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```

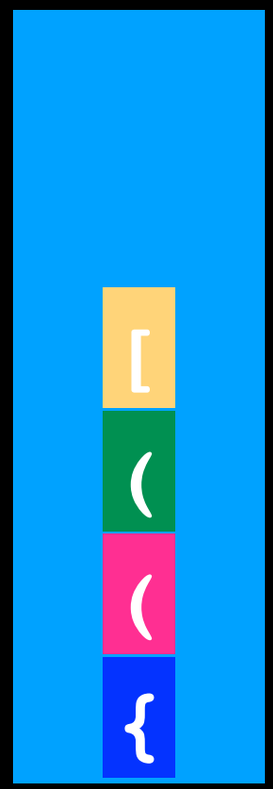


push



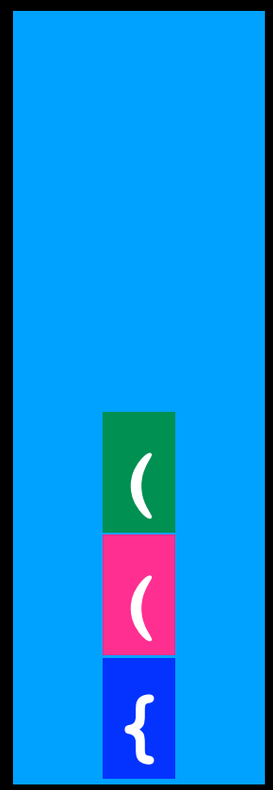
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



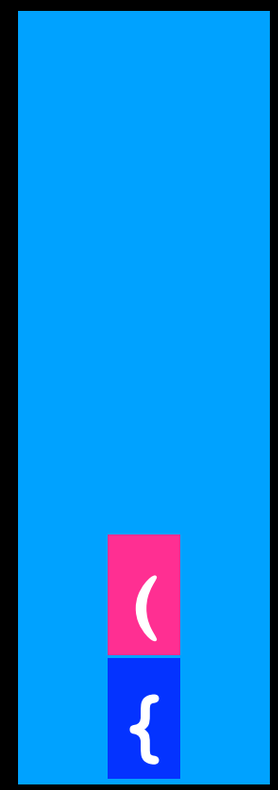
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



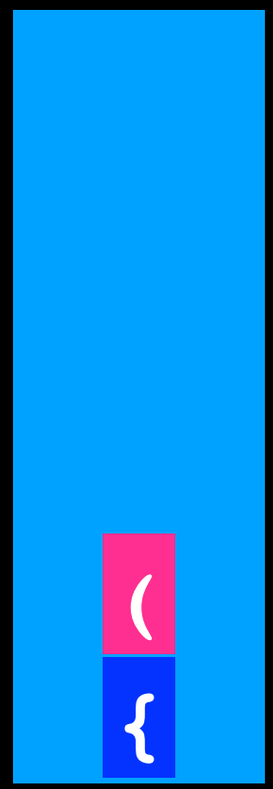
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



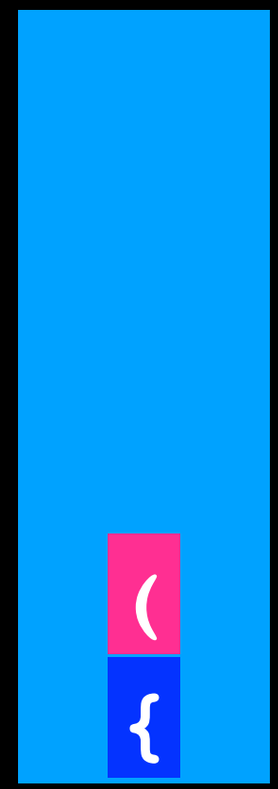
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



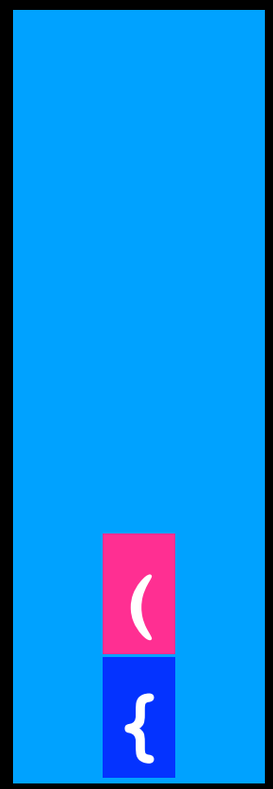
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



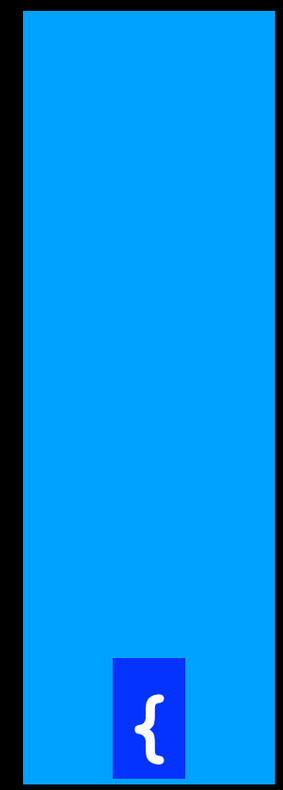
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



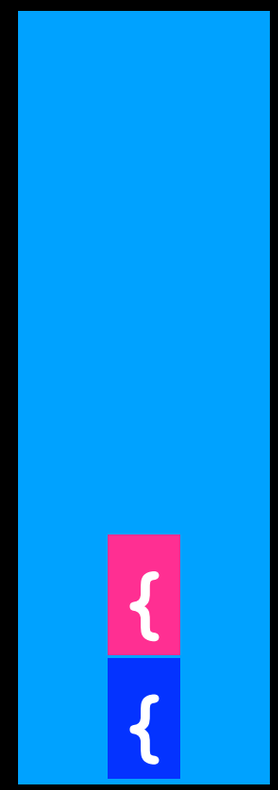
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



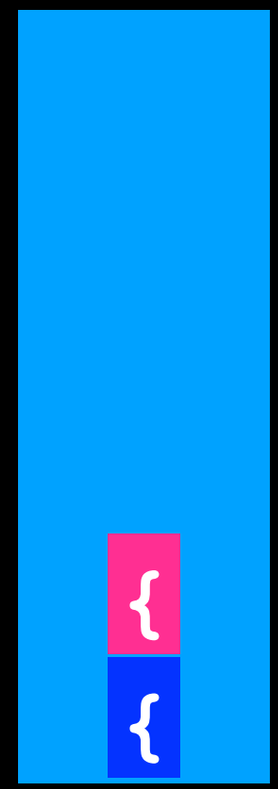
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



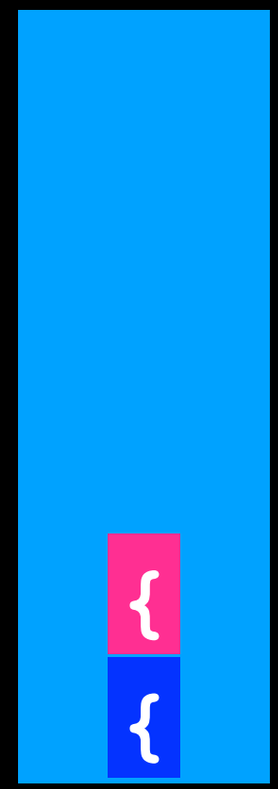
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



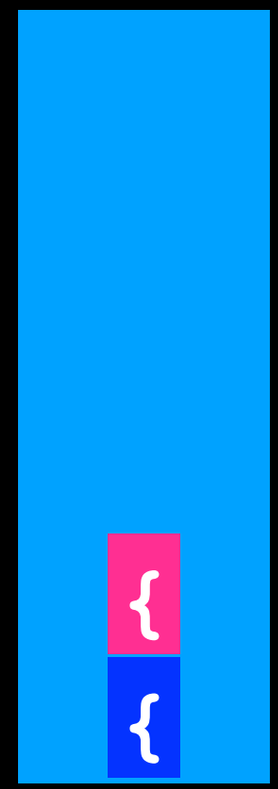
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



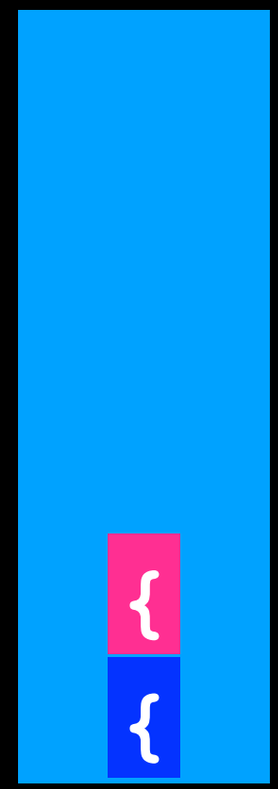
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



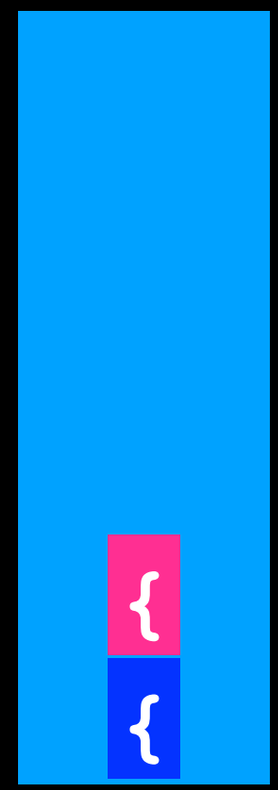
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



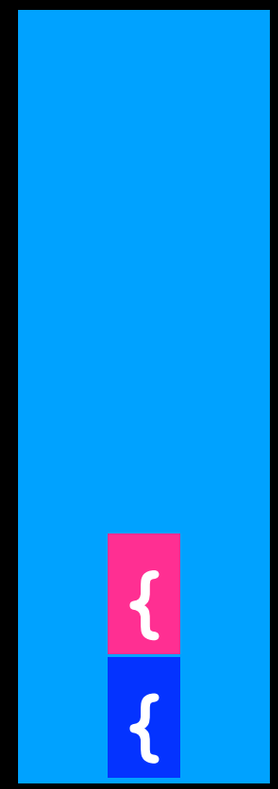
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



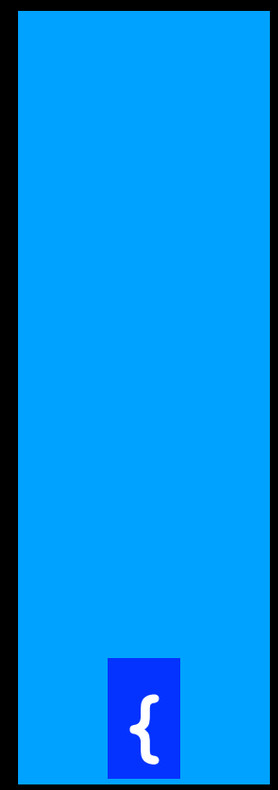
# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```



# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```

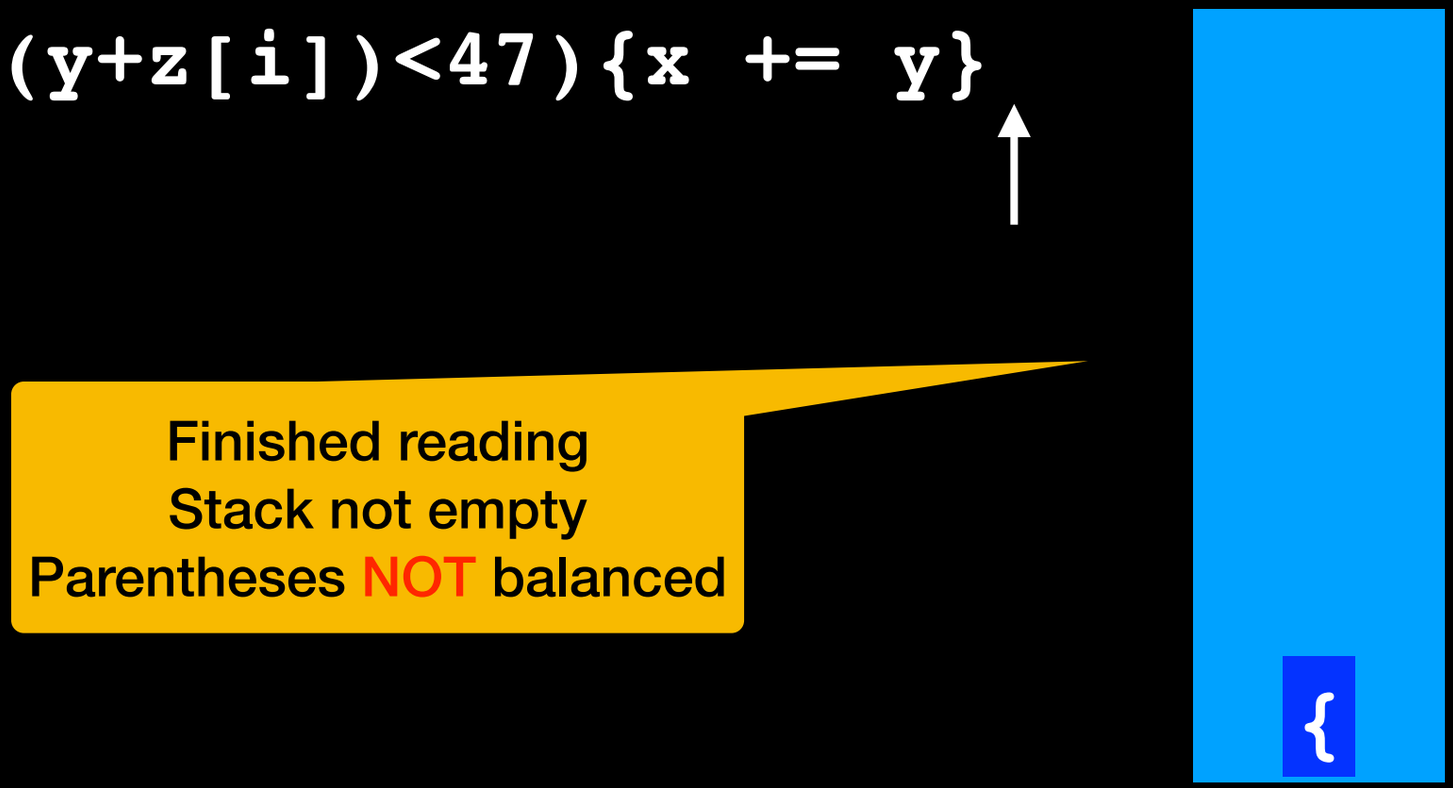


# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}}
```

# Balancing Parentheses

```
int f(){if(x*(y+z[i])<47){x += y}
```



# Balancing Parentheses

```
for(char ch : st)
{
   if ch is an open parenthesis character
       push it on the stack
   else if ch is a close parenthesis character
       if it matches the top of the stack
       pop the stack
       else
          return unbalanced
   // else it is not a parenthesis
}

if stack is empty
   return balanced
else
   return unbalanced
```

**O(n)**

# Evaluating Postfix Expressions

# Postfix Expressions

Operator applies to the two operands immediately preceding it

| Infix: | Postfix: |
| --- | --- |
| 2 * (3 + 4) | 2 3 4 + * |
| 2 * 3 + 4 | 2 3 * 4 + |

# Evaluating Postfix Expressions

Operator applies to the two operands immediately preceding it

**Postfix:**

**2 3 4 + ***

**Assumptions / simplifications:**

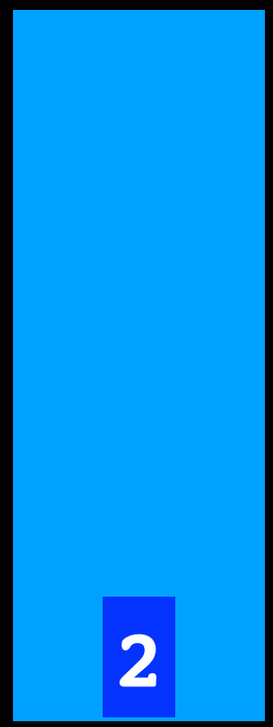
- String is syntactically correct postfix expression
- No unary operators
- No exponentiation operation
- Operands in string are single integer values

# Evaluating Postfix Expressions

**Postfix:**

**2 3 4 + ***

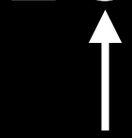


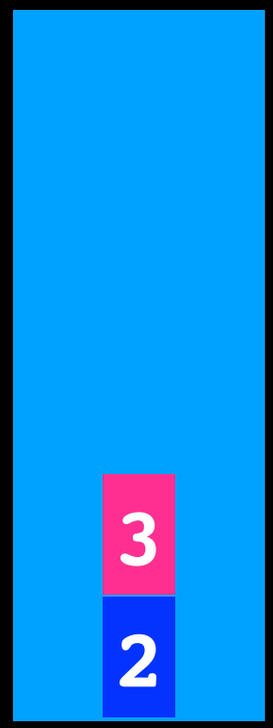


# Evaluating Postfix Expressions

**Postfix:**

2 3 4 + *

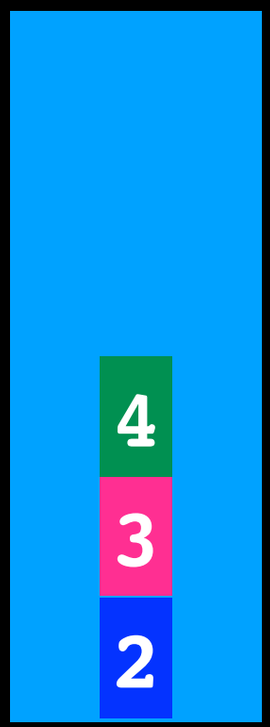




# Evaluating Postfix Expressions

**Postfix:**

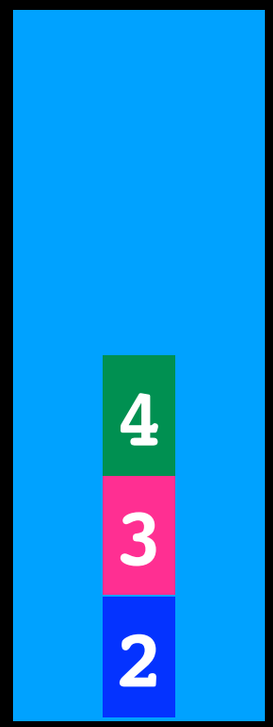
2 3 4 + *



# Evaluating Postfix Expressions

**Postfix:**

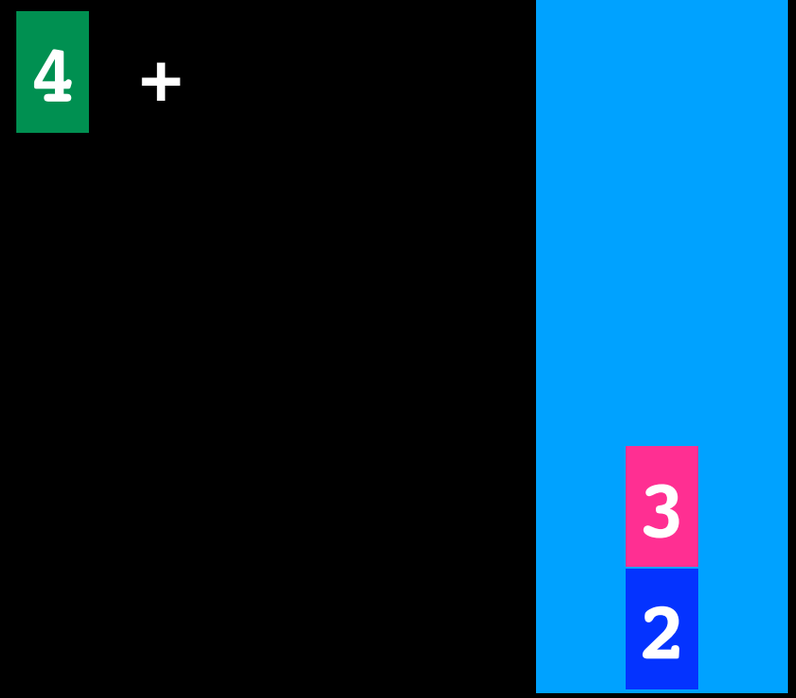
**2 3 4 + ***



# Evaluating Postfix Expressions

**Postfix:**

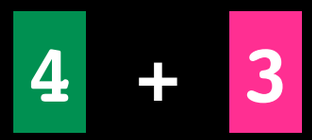
**2 3 4 + ***

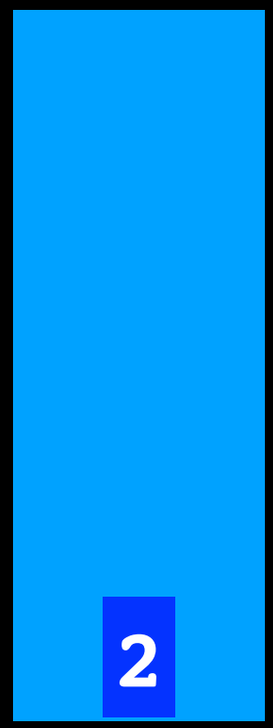


# Evaluating Postfix Expressions

**Postfix:**

2 3 4 + *

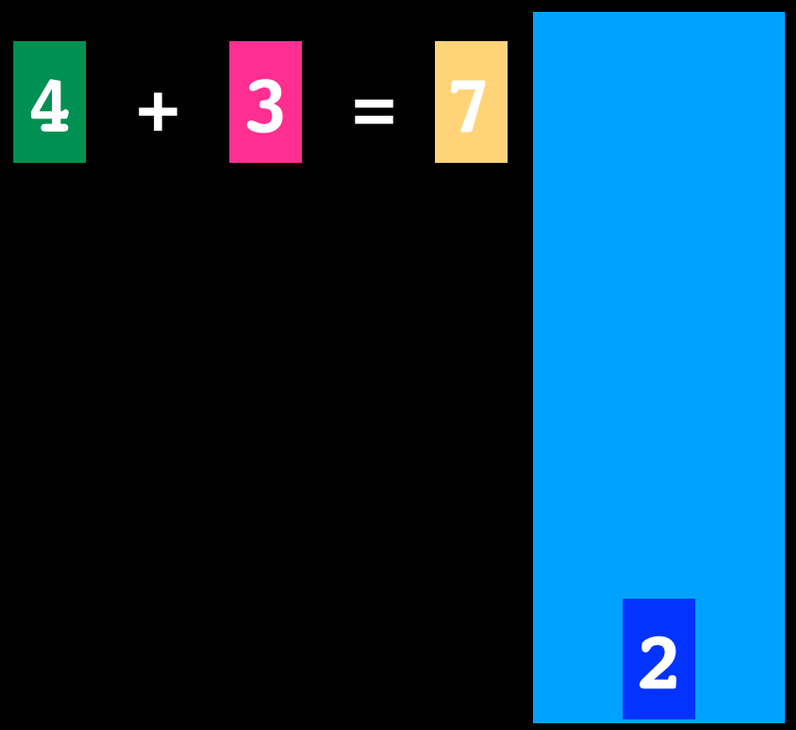


# Evaluating Postfix Expressions

**Postfix:**

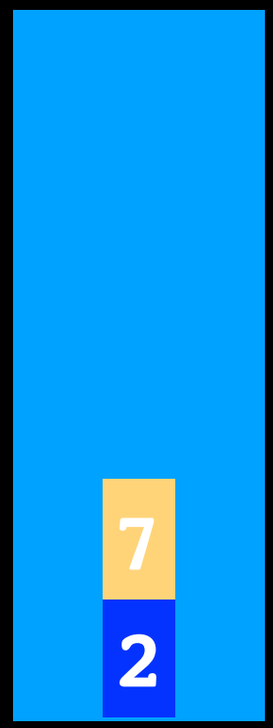
2 3 4 + *



# Evaluating Postfix Expressions

**Postfix:**

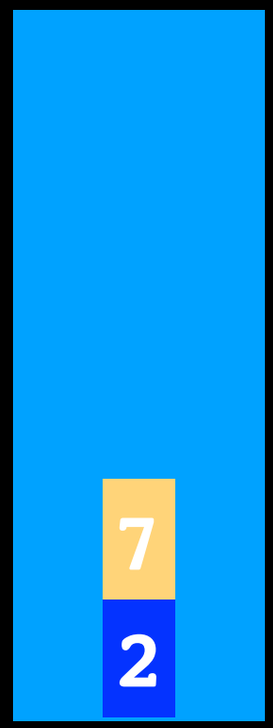
**2 3 4 + ***



# Evaluating Postfix Expressions

**Postfix:**

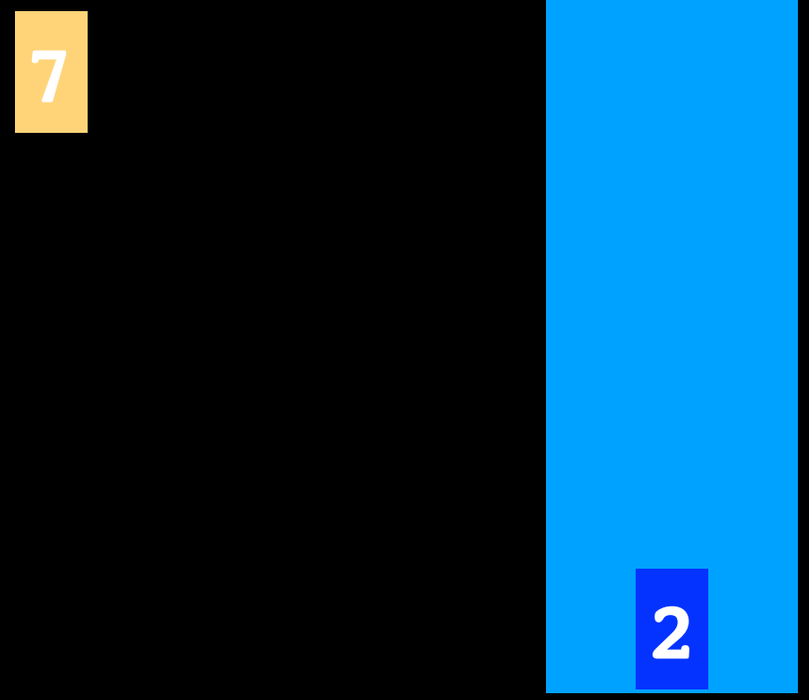
**2 3 4 + ***



# Evaluating Postfix Expressions

**Postfix:**

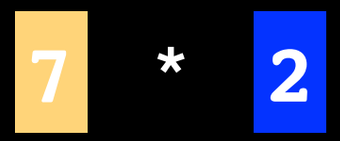
**2 3 4 + ***

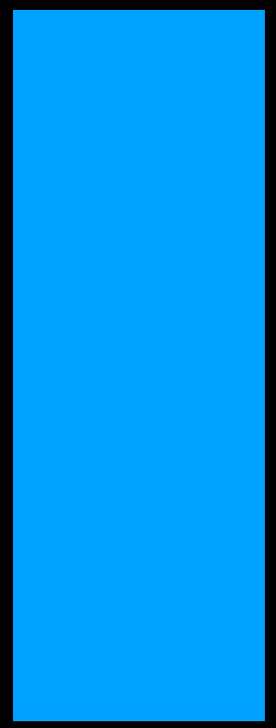


# Evaluating Postfix Expressions

**Postfix:**

2 3 4 + *

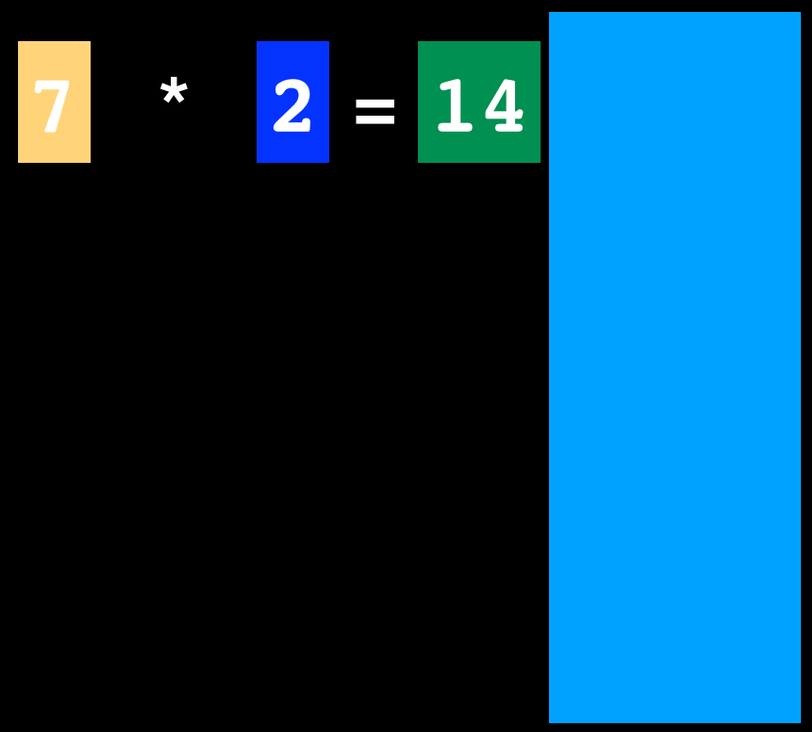


# Evaluating Postfix Expressions

**Postfix:**

2 3 4 + *

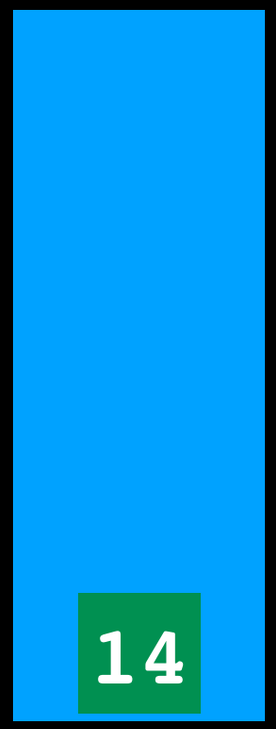


# Evaluating Postfix Expressions

**Postfix:**

**2 3 4 + ***

Done reading string
The top of the stack
is the result




# Evaluating Postfix Expressions

Operator applies to the two operands immediately preceding it

**Postfix:**

**2 3 * 4 +**

Assumptions / simplifications:

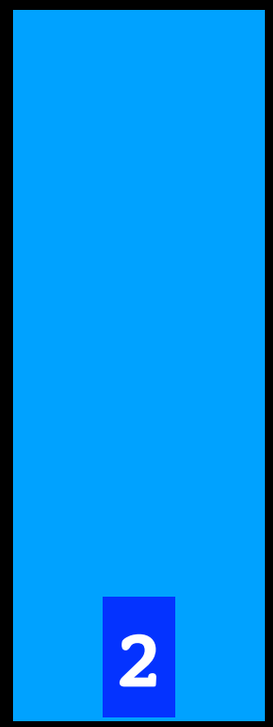
- string is syntactically correct postfix expression
- No unary operators
- No exponentiation operation
- Operands in string are single integer values

# Evaluating Postfix Expressions

**Postfix:**

**2 3 * 4 +**

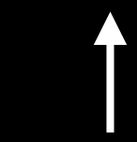


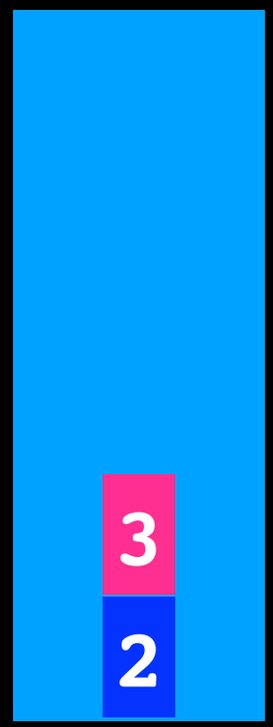


# Evaluating Postfix Expressions

**Postfix:**

**2 3 * 4 +**

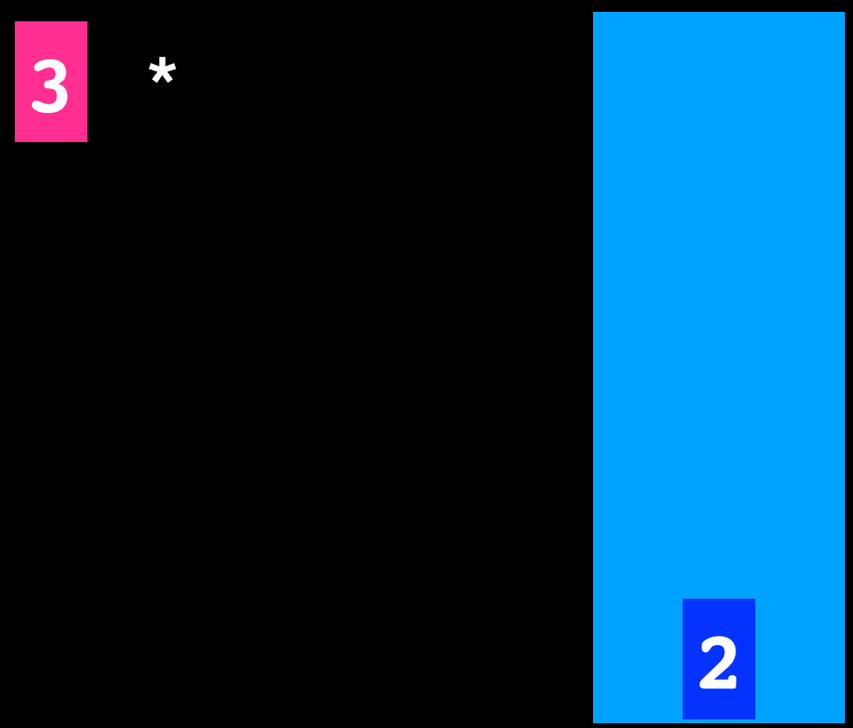




# Evaluating Postfix Expressions

**Postfix:**

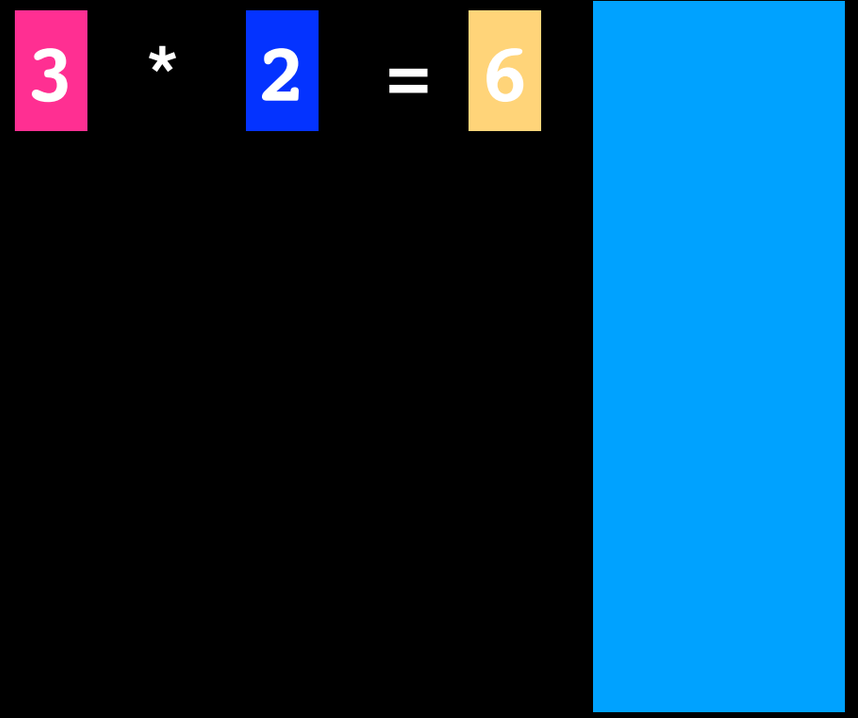
**2 3 * 4 +**



# Evaluating Postfix Expressions

**Postfix:**

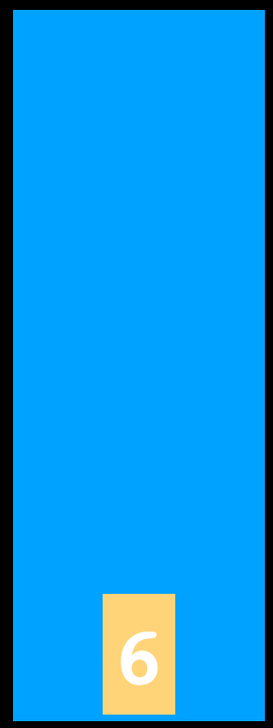
2 3 * 4 +



# Evaluating Postfix Expressions

**Postfix:**

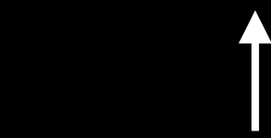
**2 3 * 4 +**

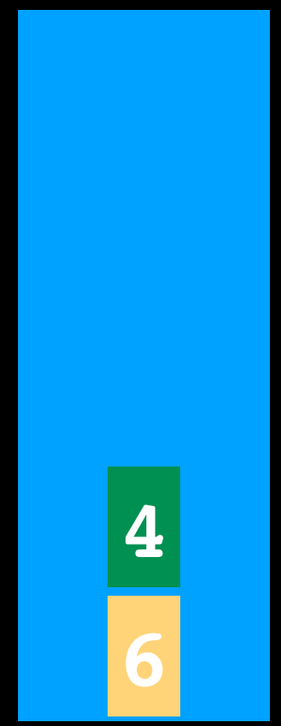


# Evaluating Postfix Expressions

**Postfix:**

**2 3 * 4 +**

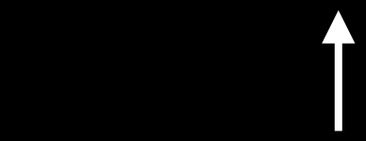


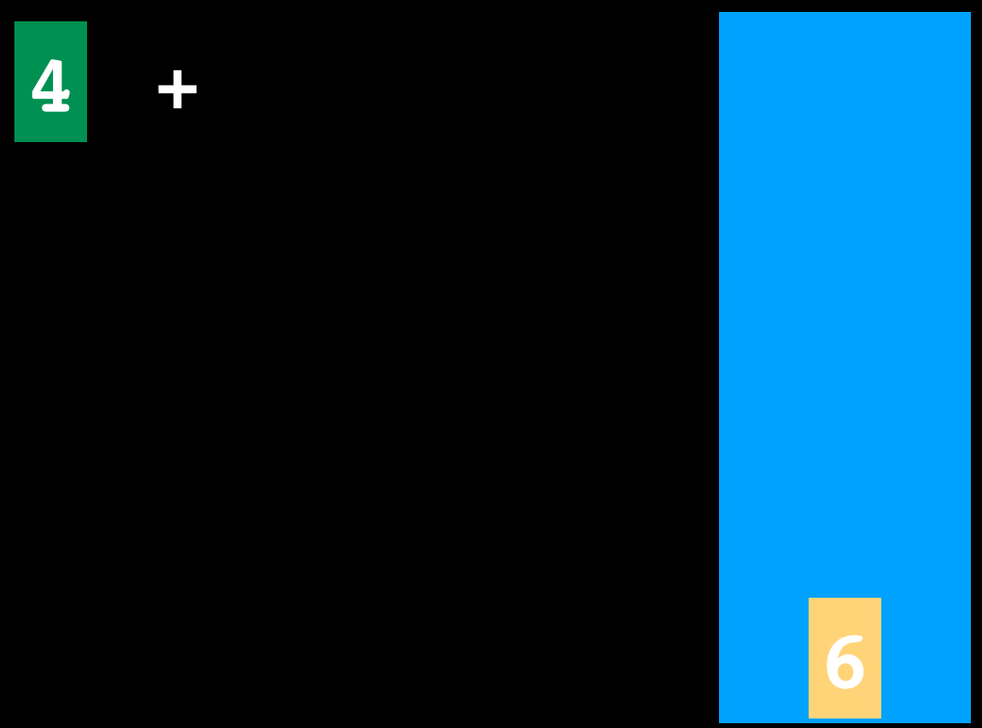


# Evaluating Postfix Expressions

**Postfix:**

**2 3 * 4 +**

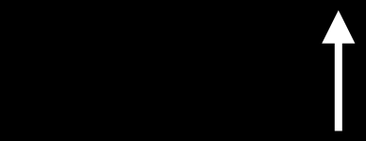




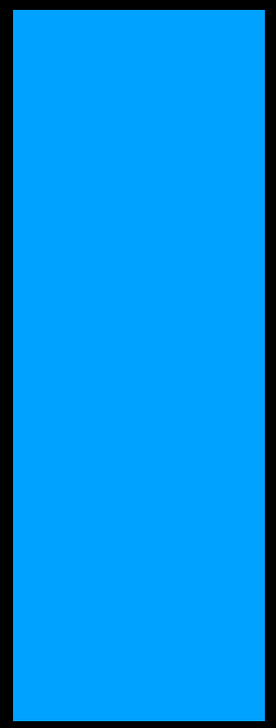
# Evaluating Postfix Expressions

**Postfix:**

2 3 * 4 +



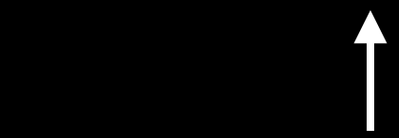
4 + 6 = 10



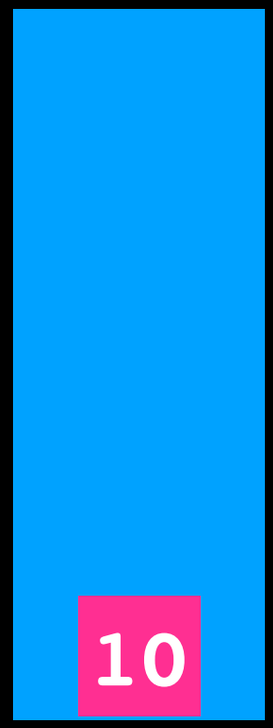
# Evaluating Postfix Expressions

**Postfix:**

**2 3 * 4 +**



Done reading string
The top of the stack
is the result




# Evaluating Postfix Expressions

```
for(char ch : st)
{
   if ch is an operand
       push it on the stack
   else // ch is an operator op
   {
      //evaluate and push the result
     operand2 = pop stack
     operand1 = pop stack
     result = operand1 op operand2
     push result on stack
    }
}
```

**O(n)**

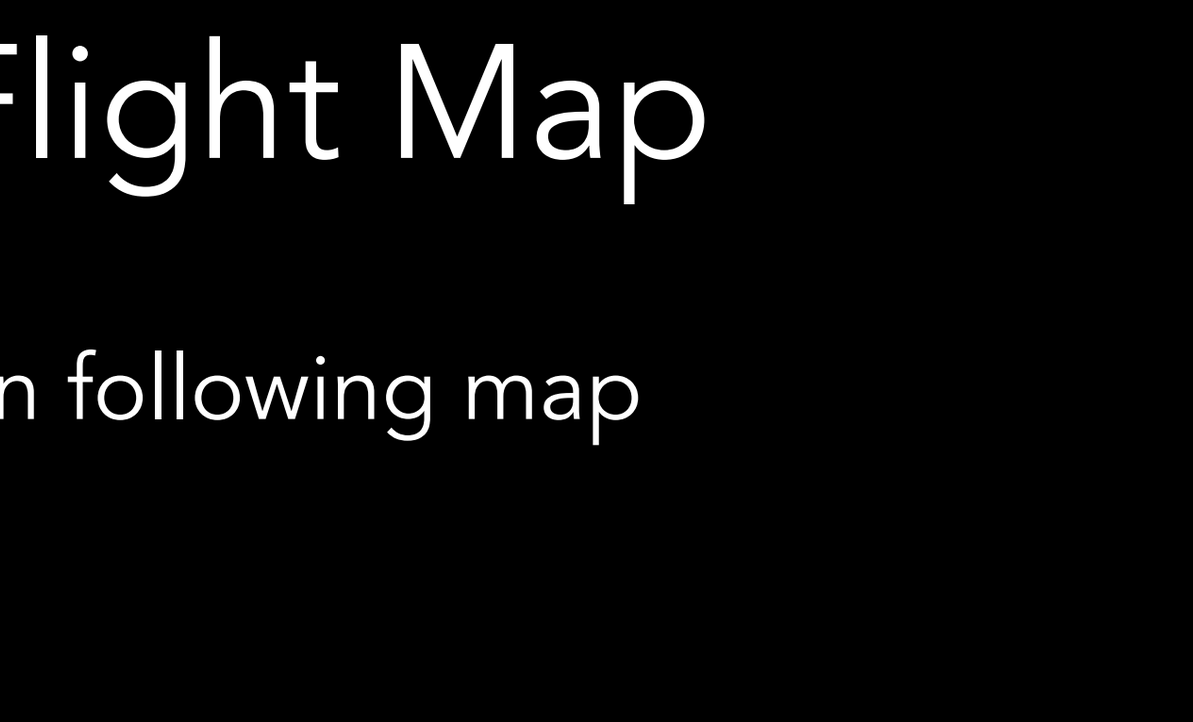
# Search a Flight Map

Fly from Origin to Destination following map

1. Reach destination

2. Reach city with no departing flights (dead end)

3. Go in circles forever



# Backtracking

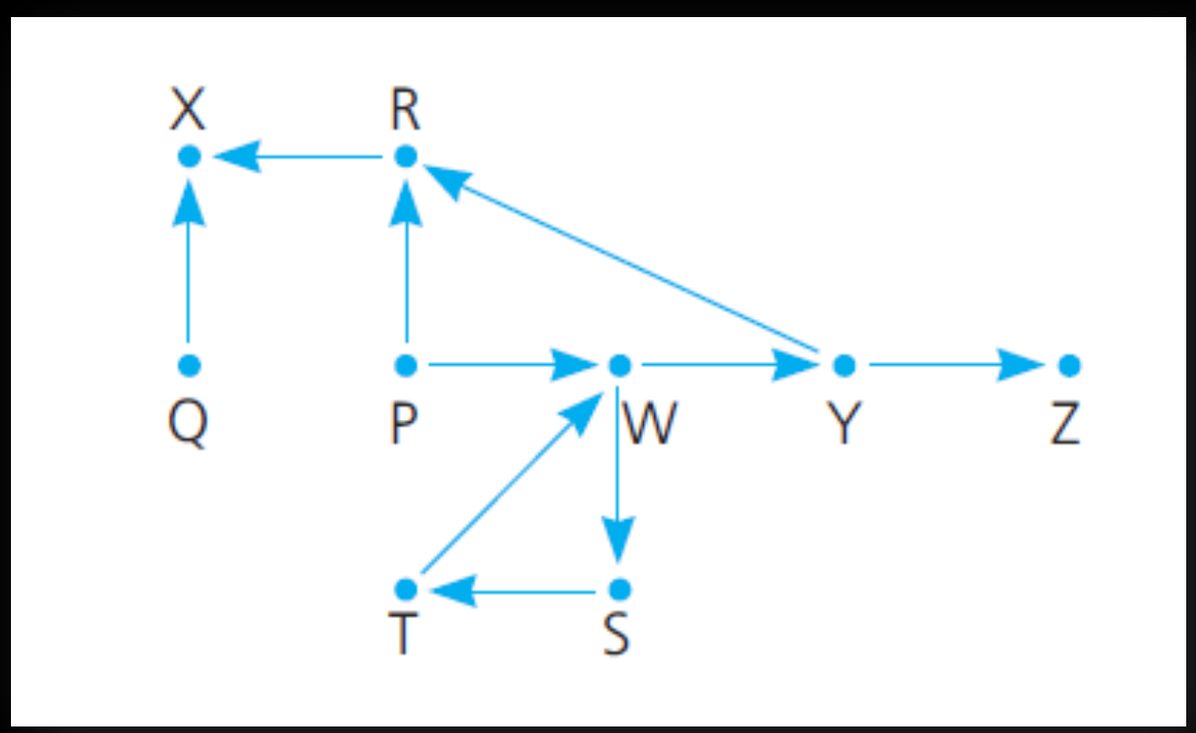
Avoid dead end by backtracking

**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

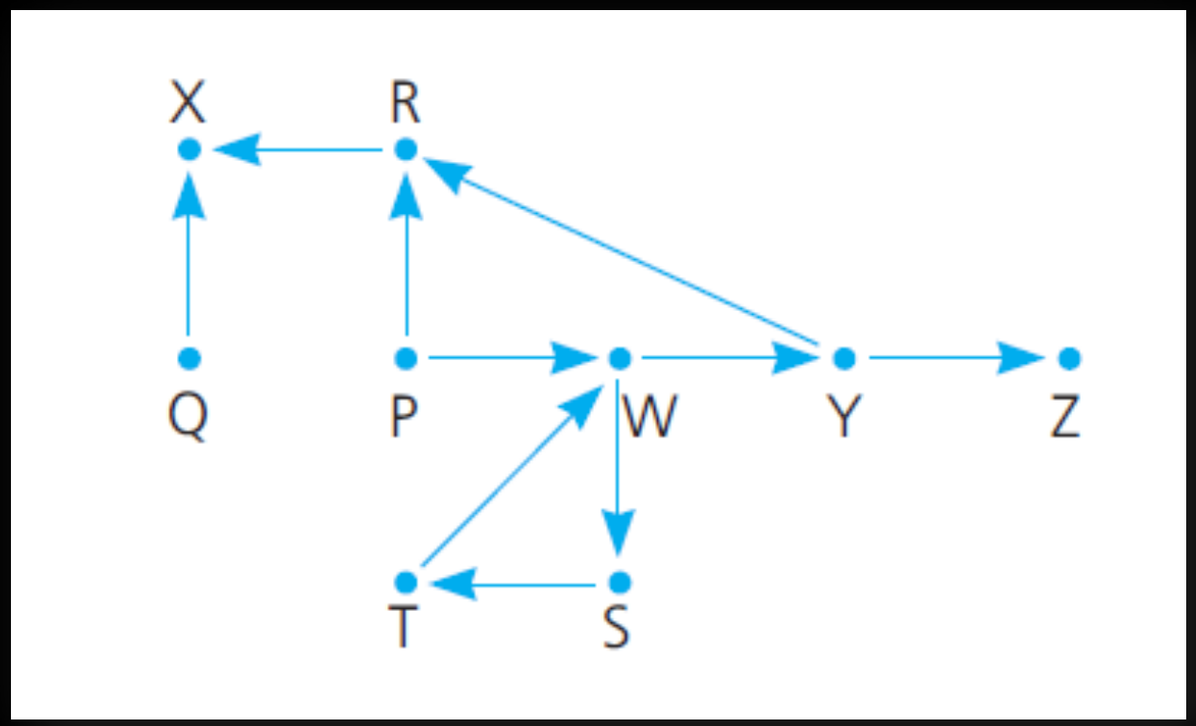
**C = visited**

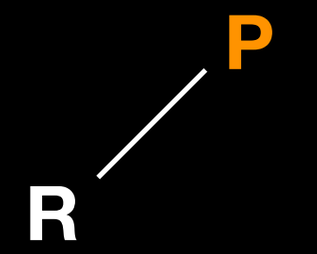
**C = backtracked**

Avoid dead end by backtracking

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

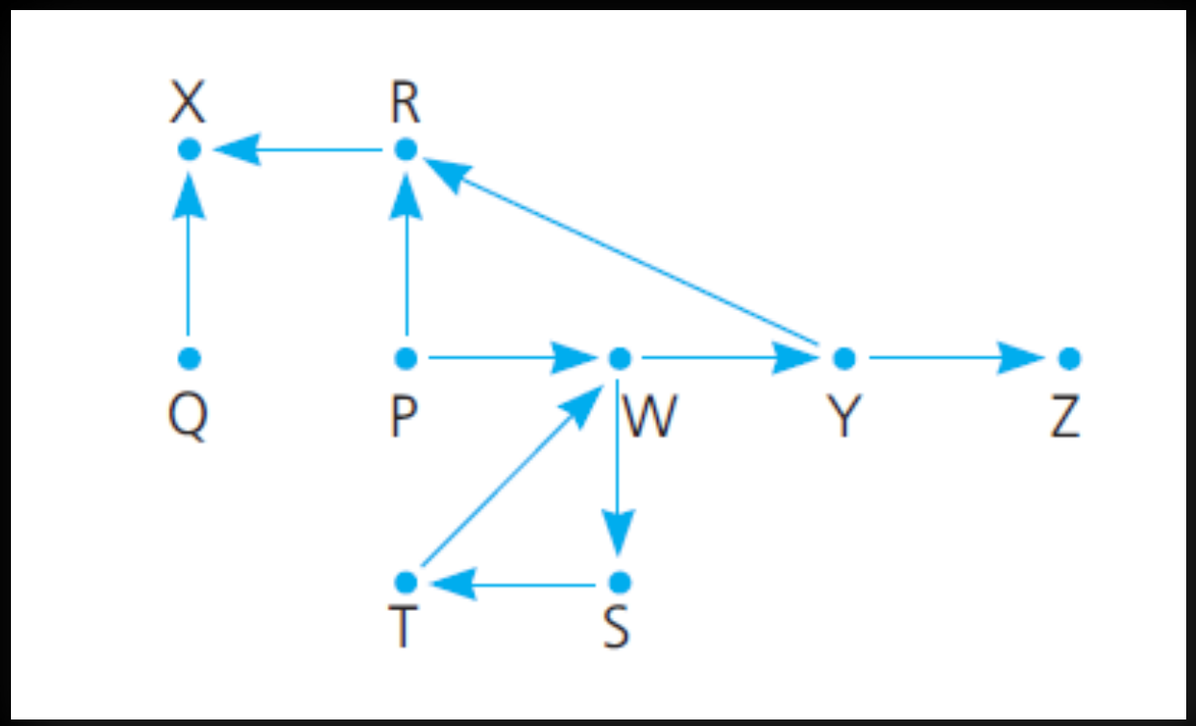
Avoid dead end by backtracking

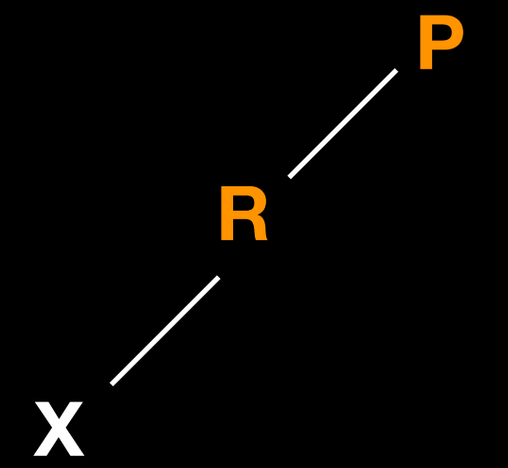
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

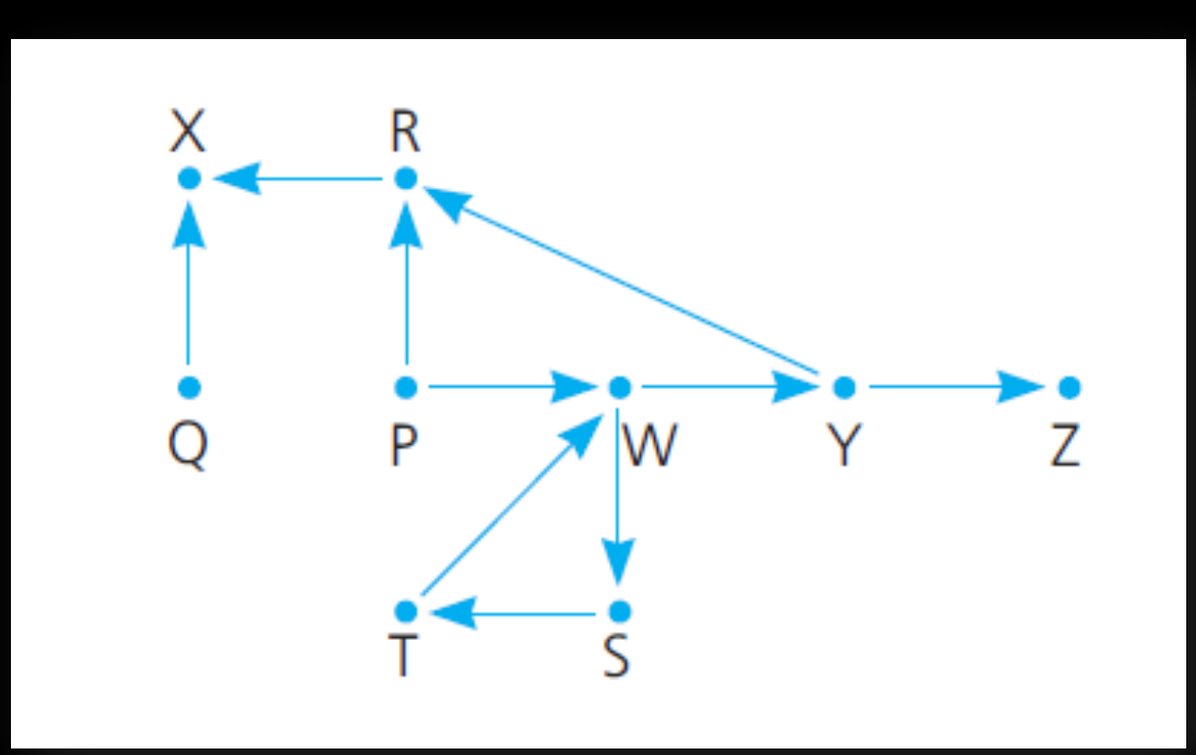
Avoid dead end by backtracking

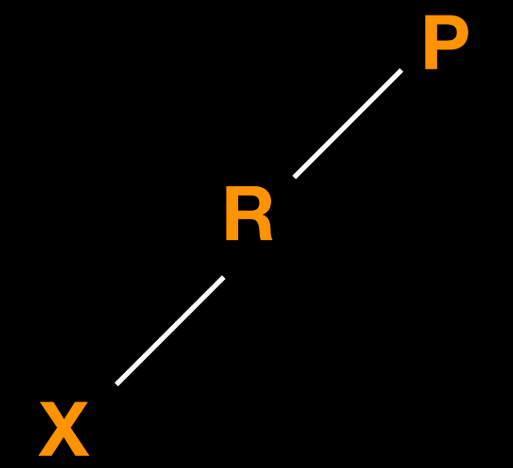
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

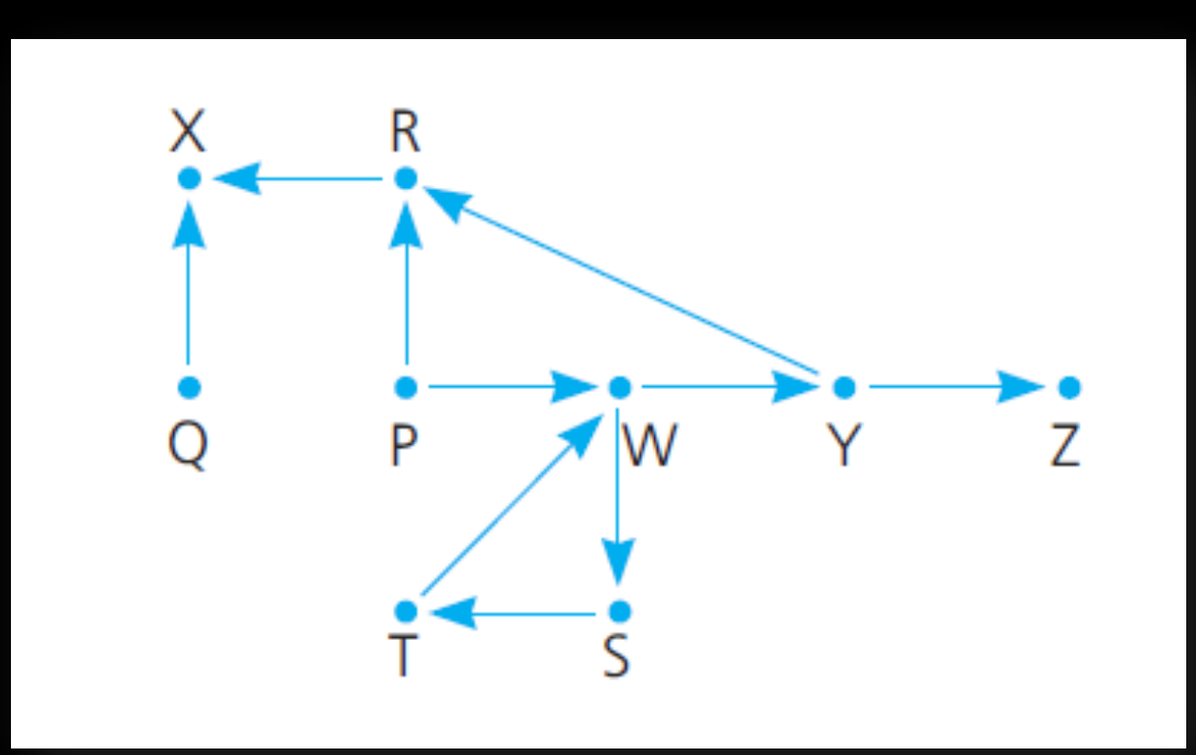
Avoid dead end by backtracking

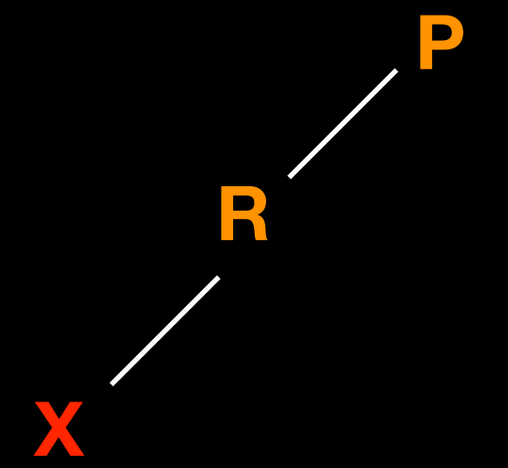
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

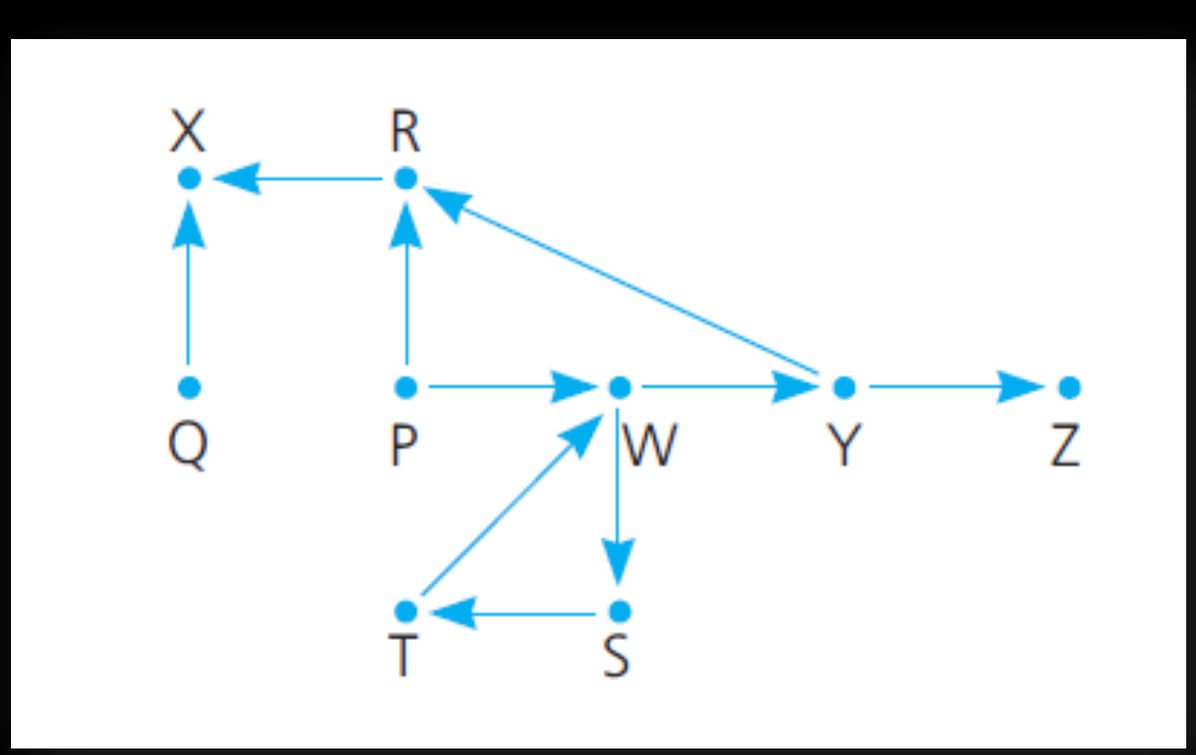
Avoid dead end by backtracking

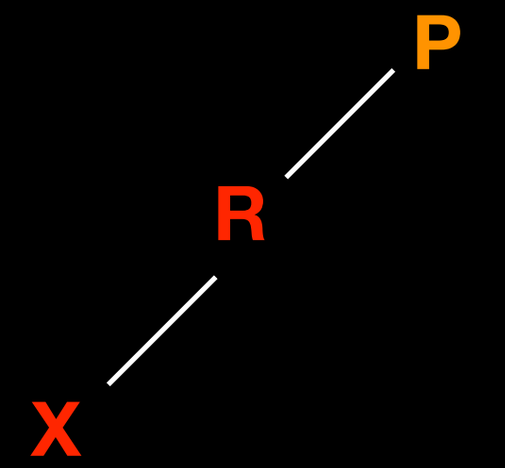
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

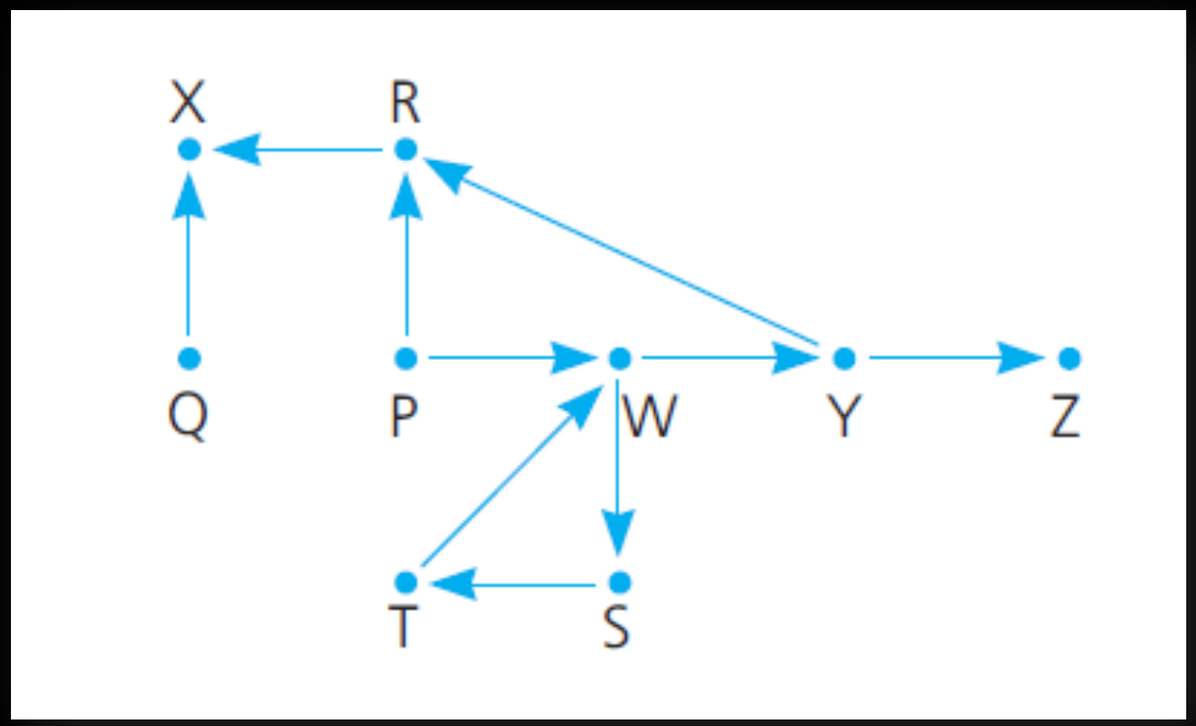
Avoid dead end by backtracking

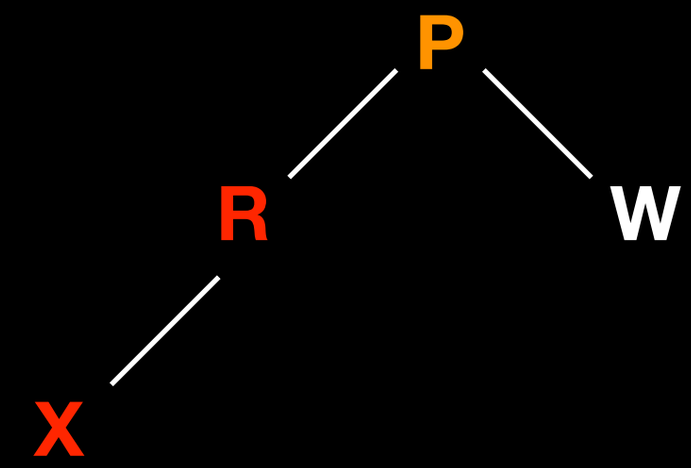
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

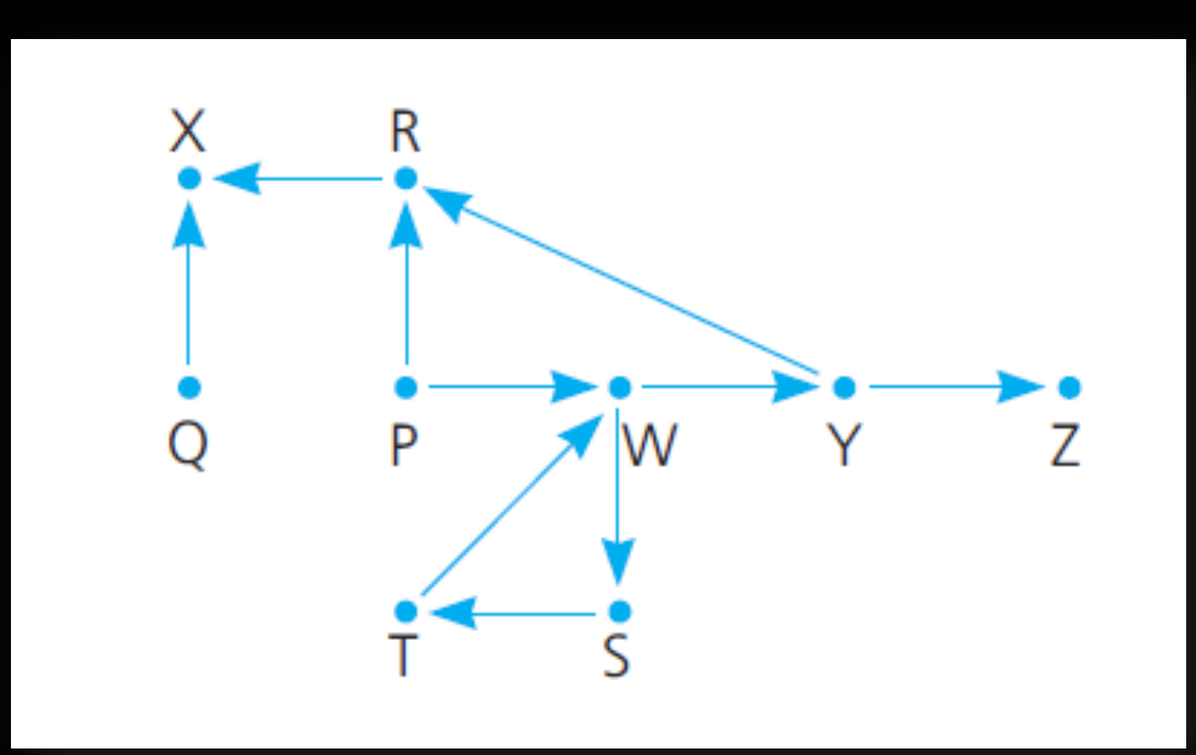
Avoid dead end by backtracking

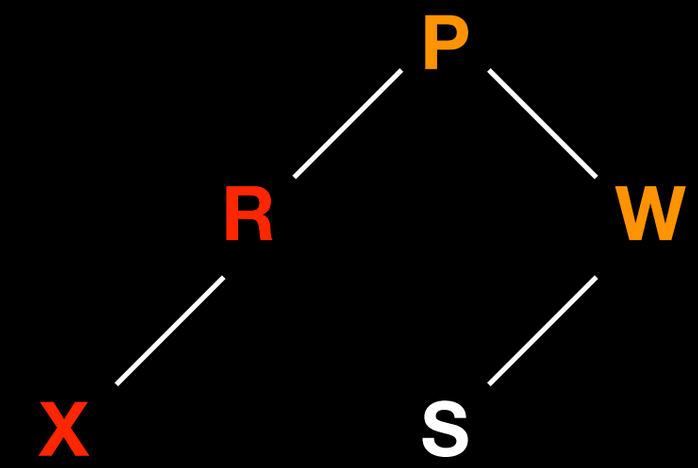
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

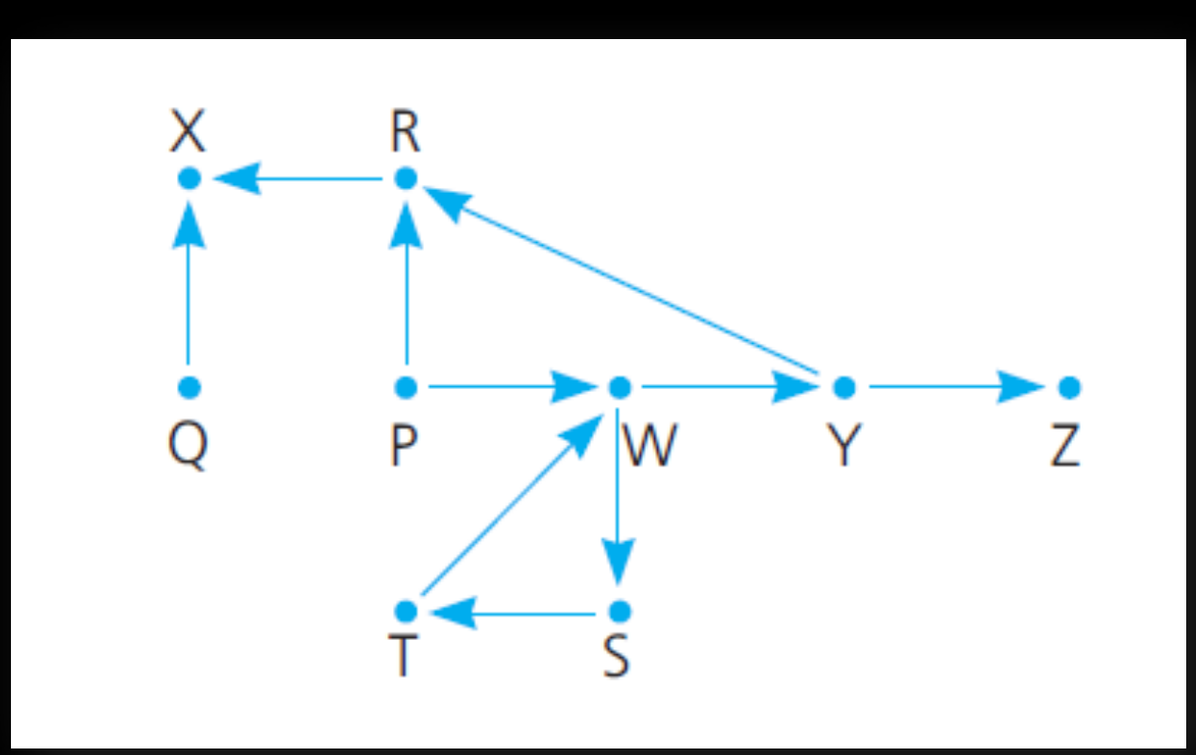
Avoid dead end by backtracking

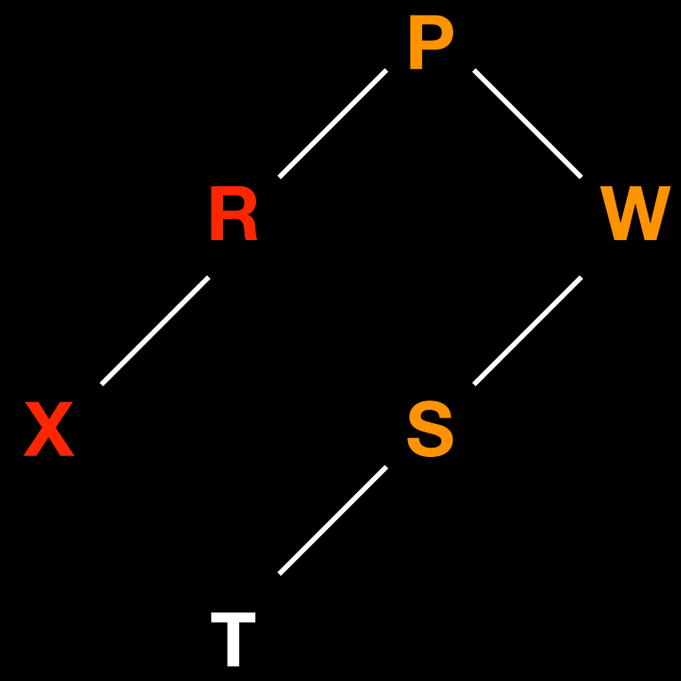
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

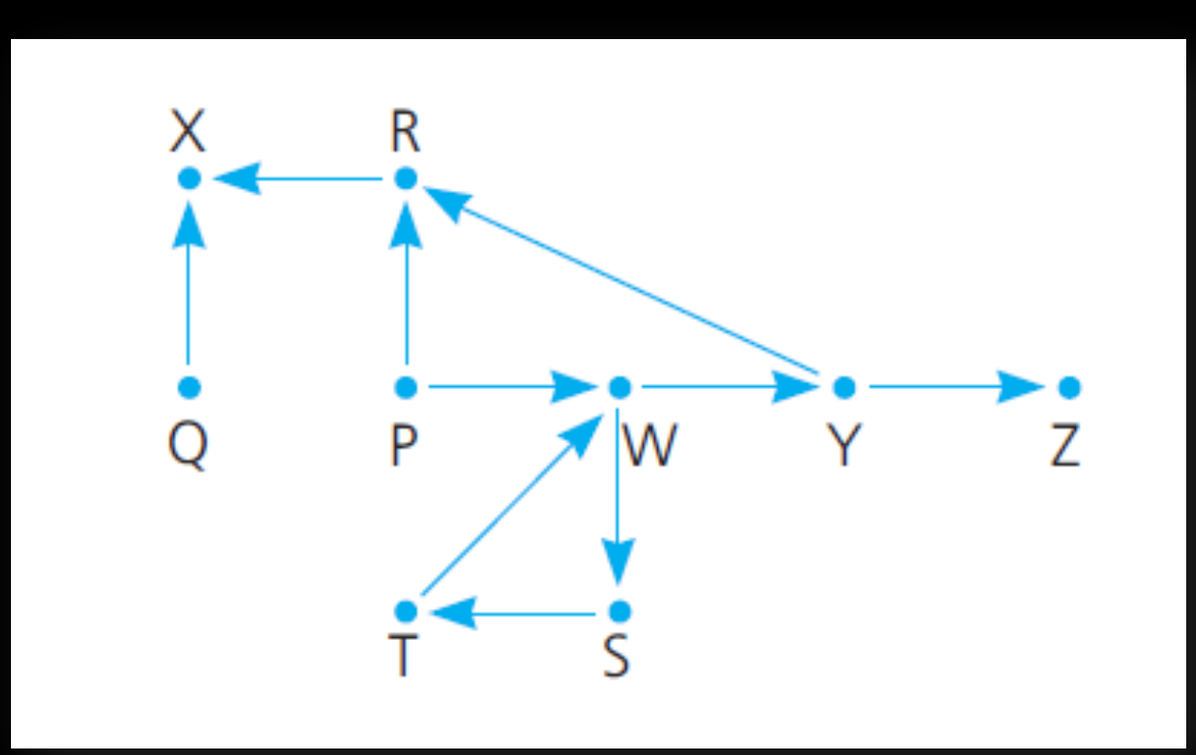
Avoid dead end by backtracking

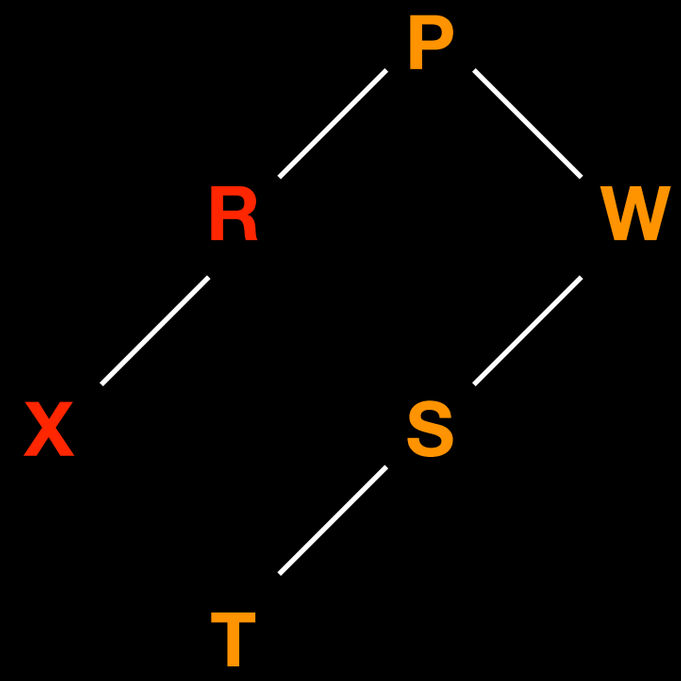
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

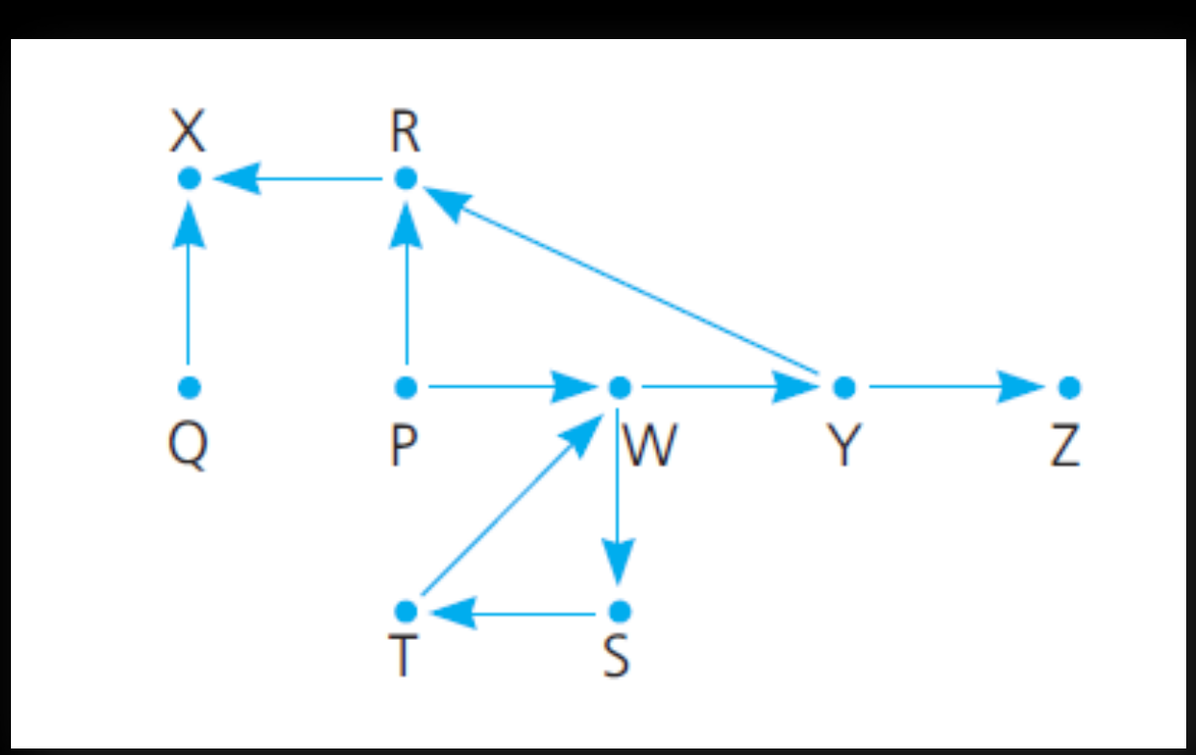
Avoid dead end by backtracking

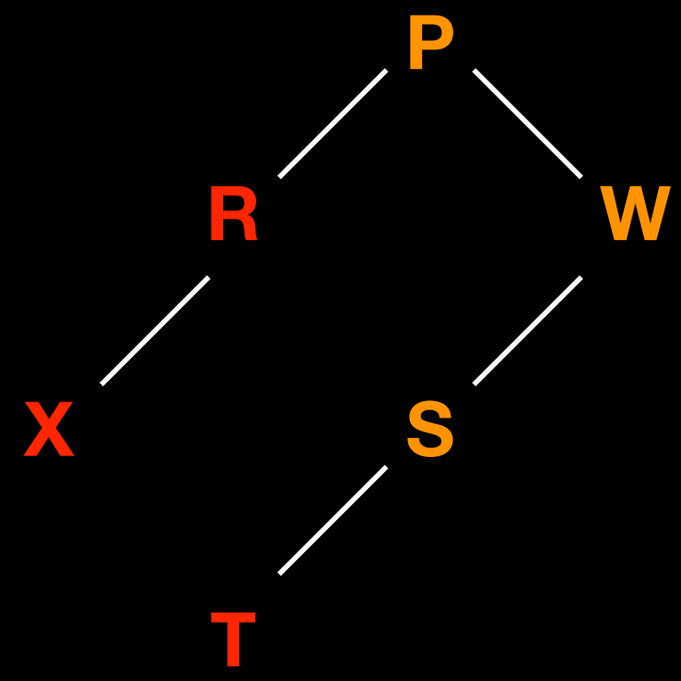
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

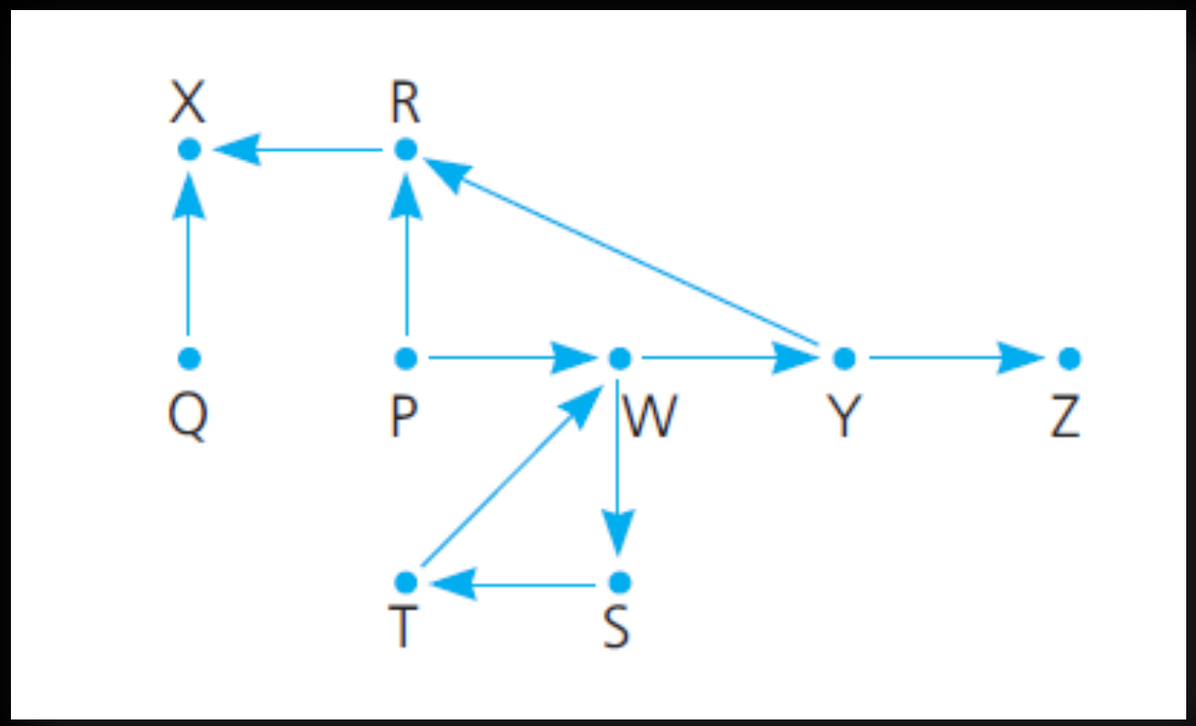
Avoid dead end by backtracking

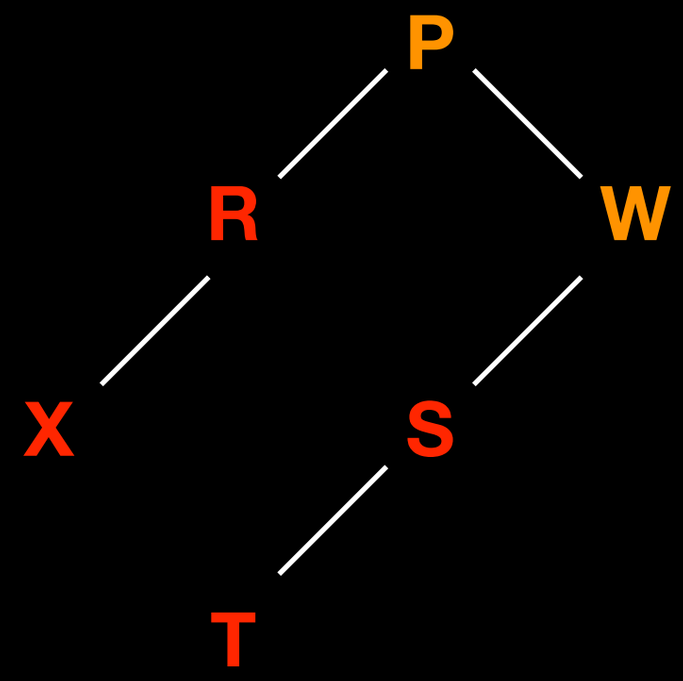
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

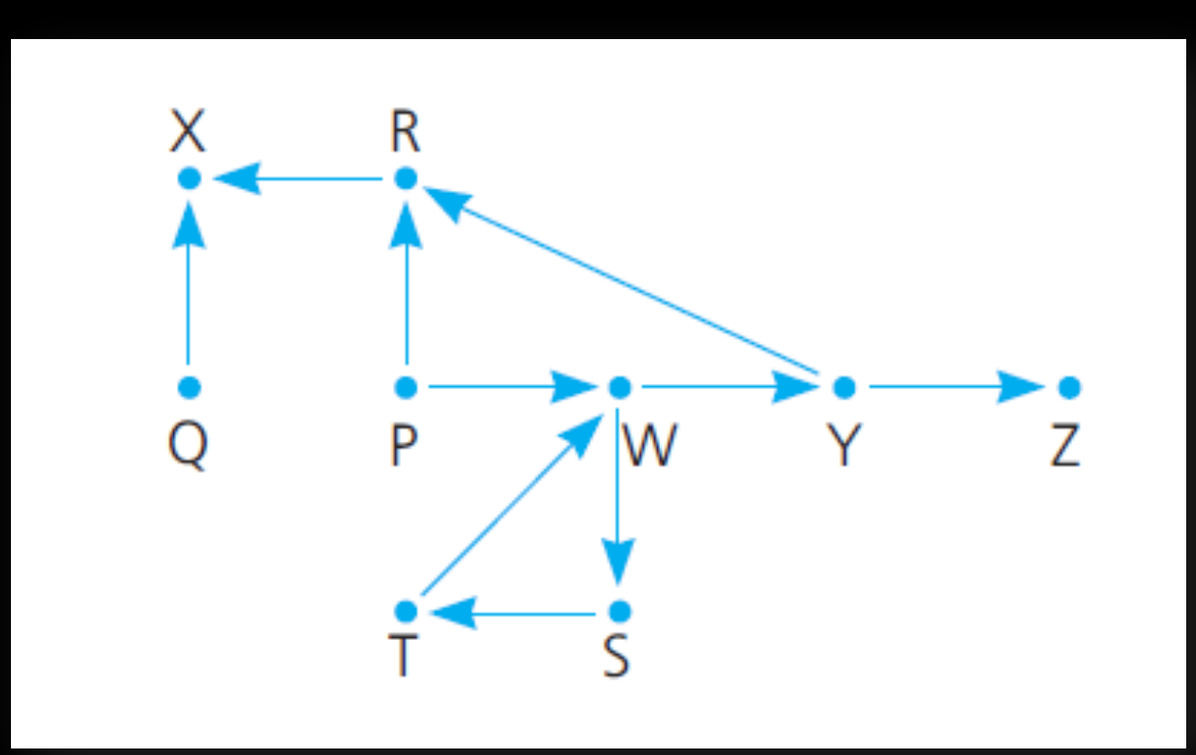
Avoid dead end by backtracking

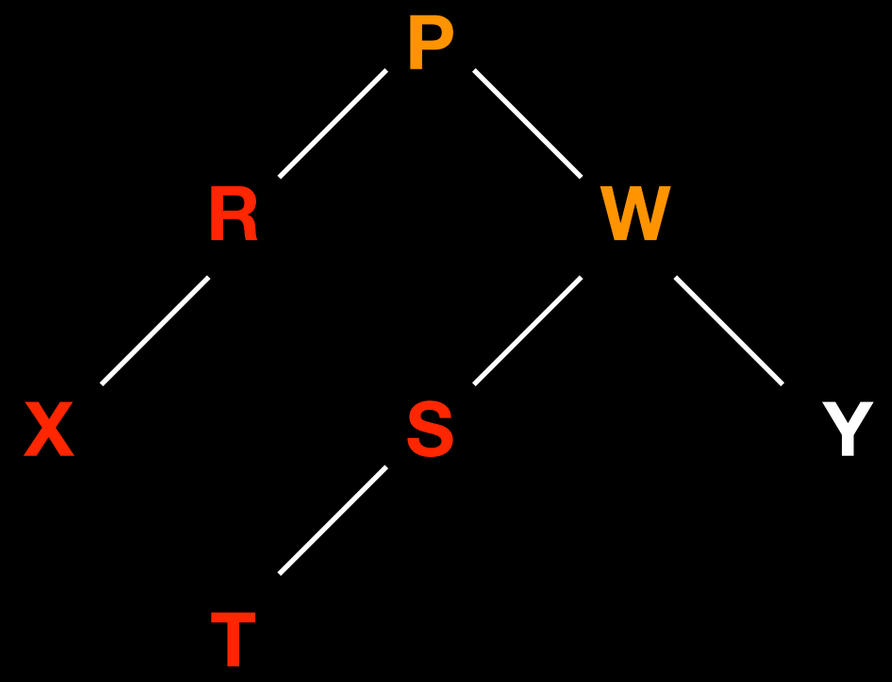
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

**Origin = P , Destination = Z**



# Backtracking

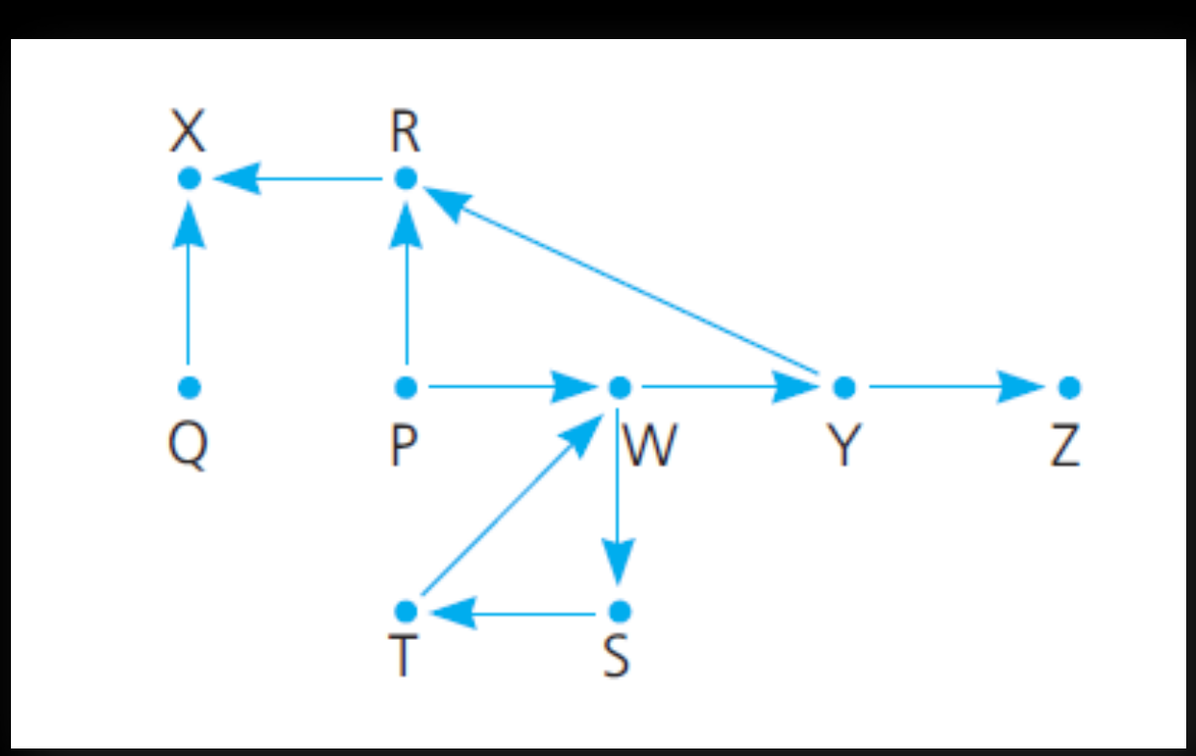
Avoid dead end by backtracking

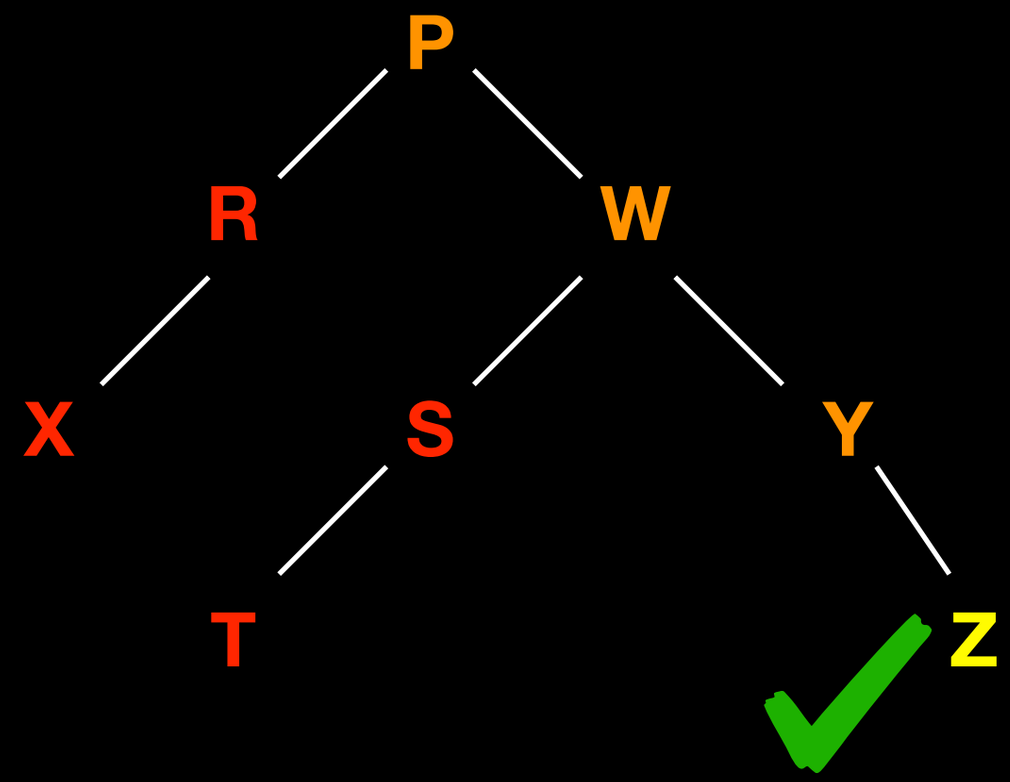
**C = visited**

**C = backtracked**

Avoid traveling in circles by marking visited cities

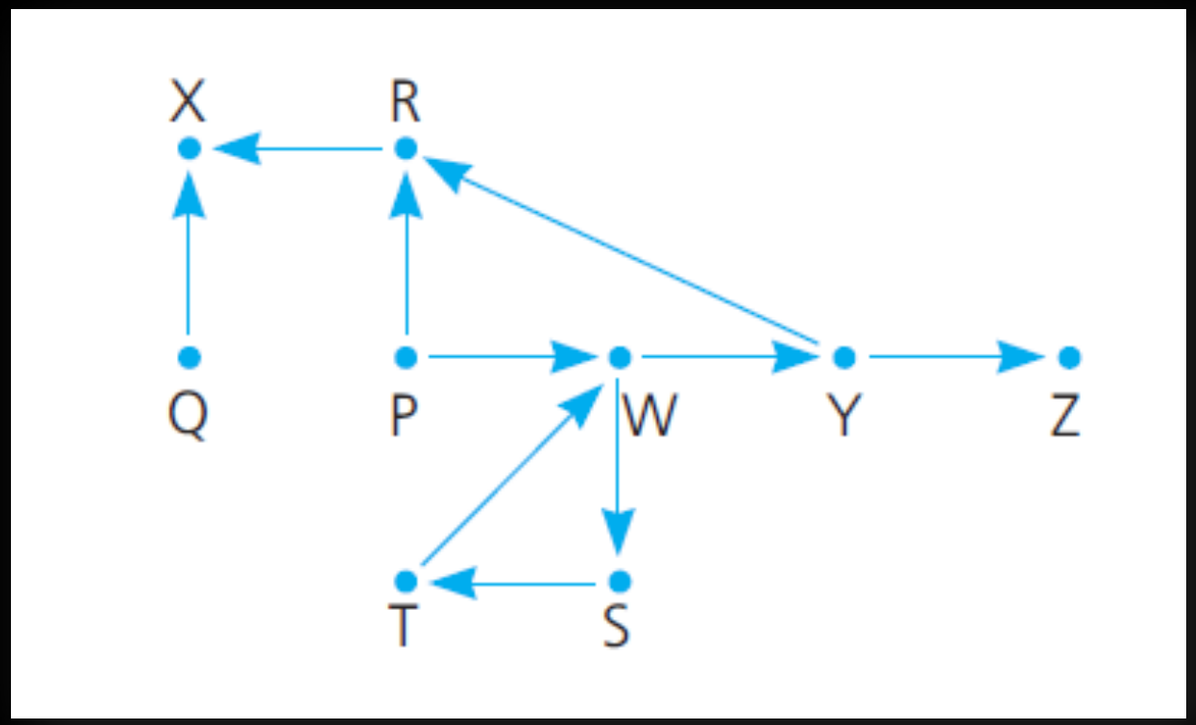
**Origin = P , Destination = Z**

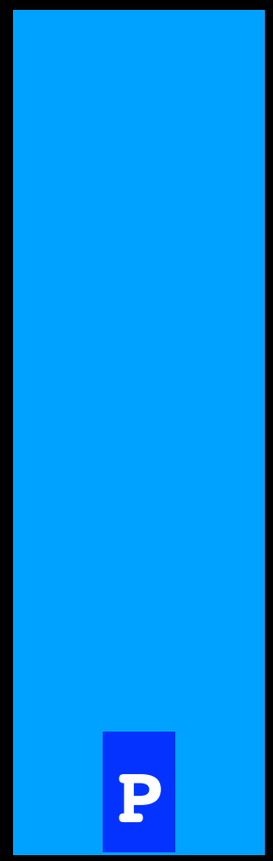


# Backtracking

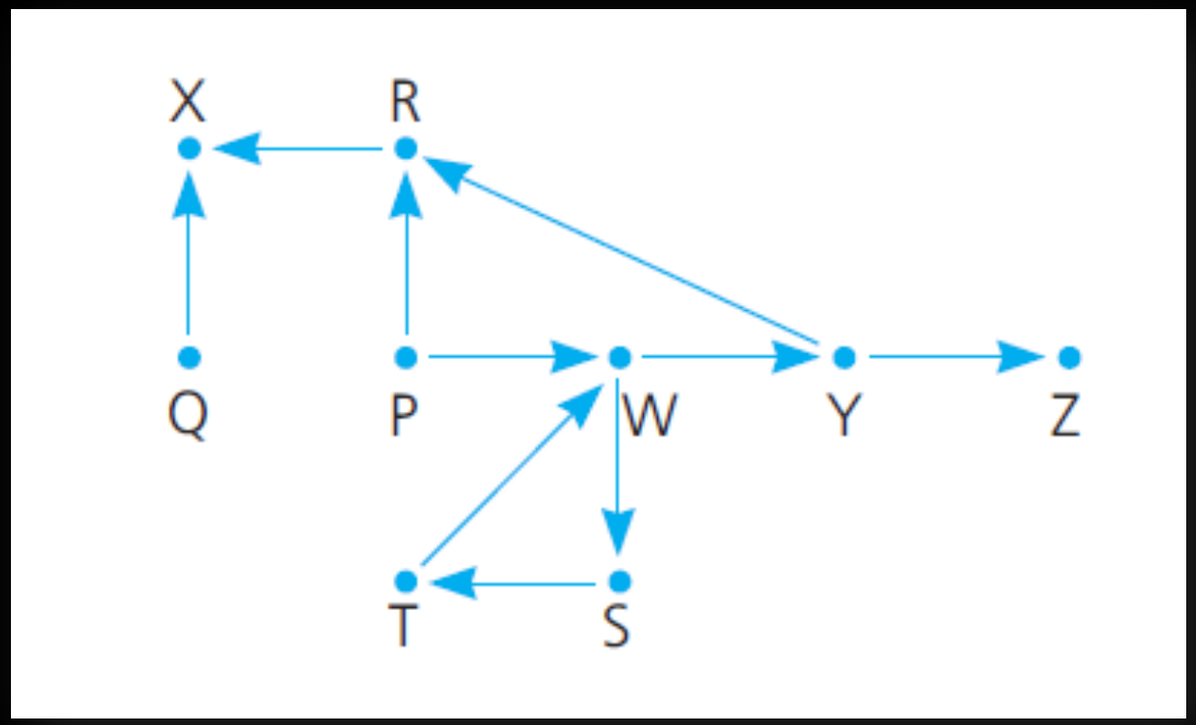
**Origin = P , Destination = Z**

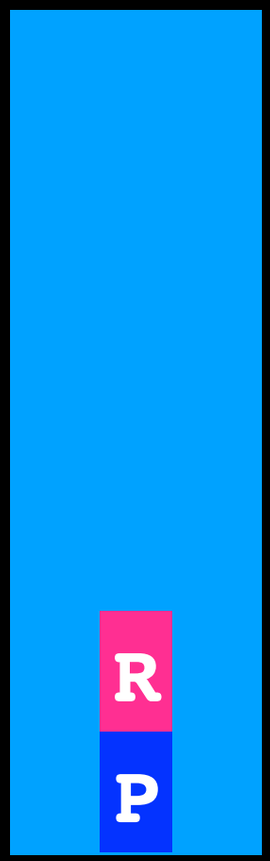


# Backtracking

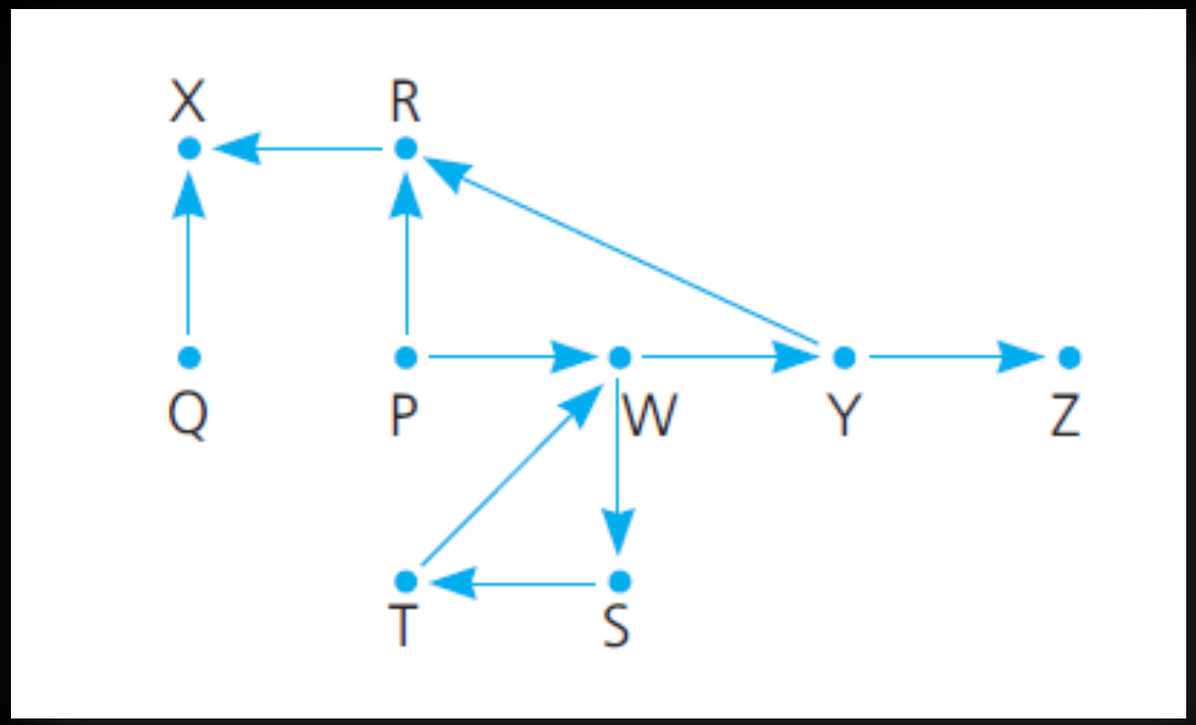
**Origin = P , Destination = Z**

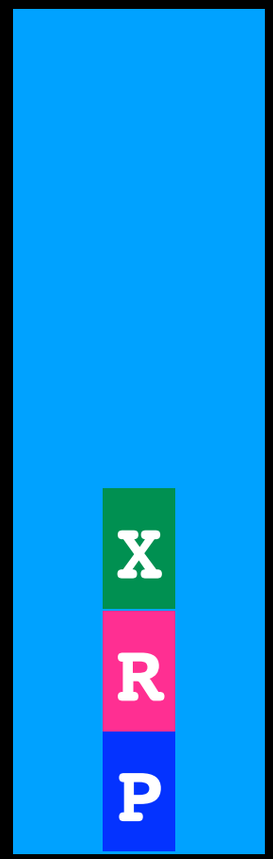


# Backtracking

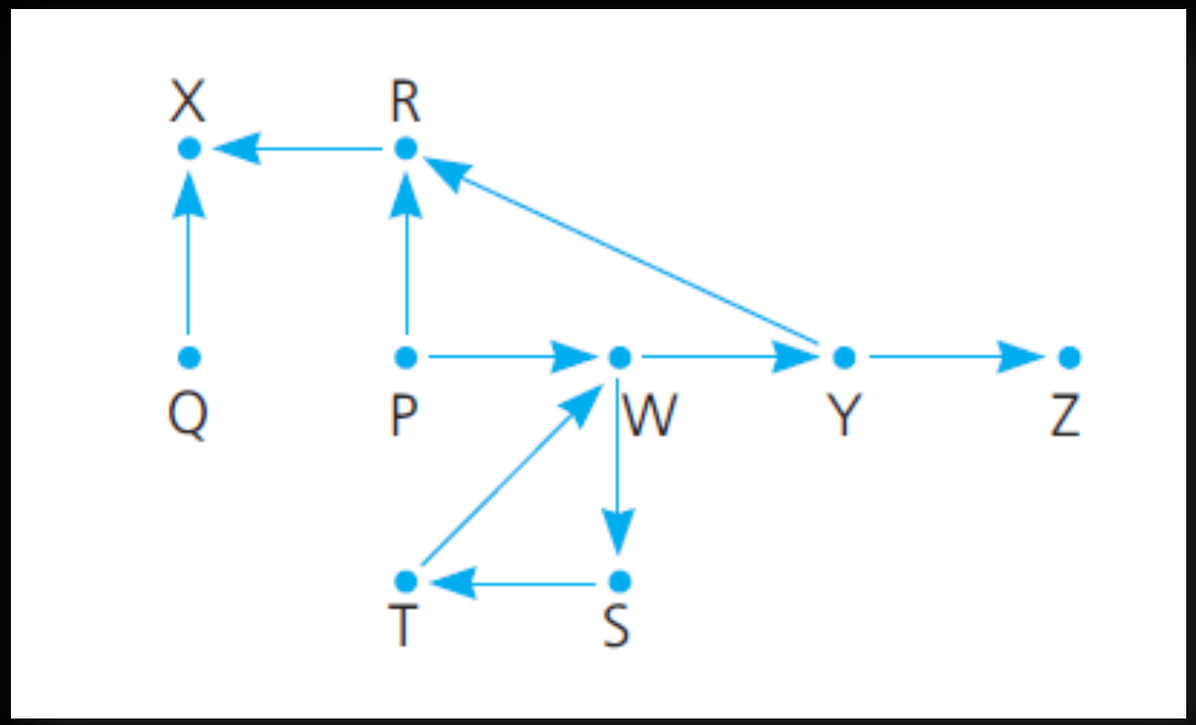
**Origin = P , Destination = Z**

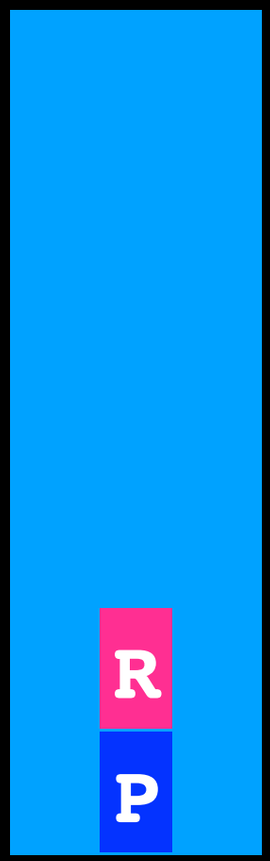


# Backtracking

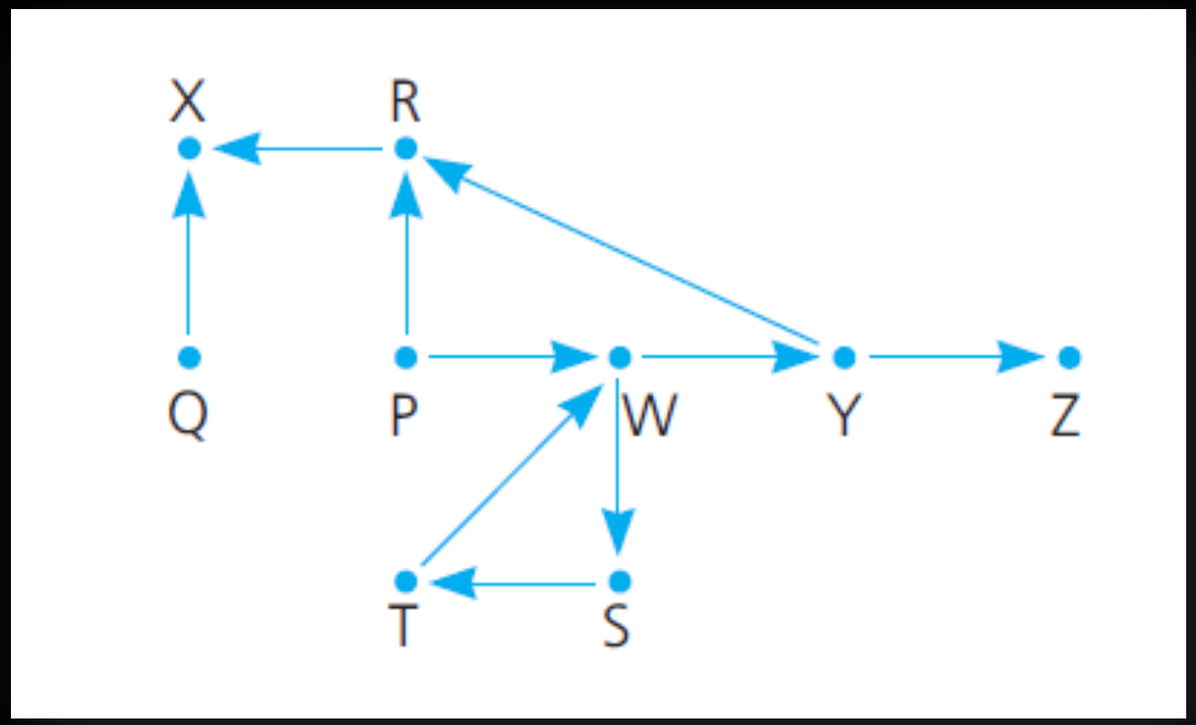
**Origin = P , Destination = Z**

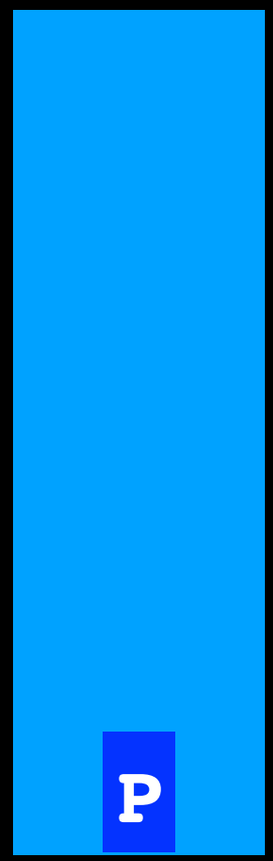


# Backtracking

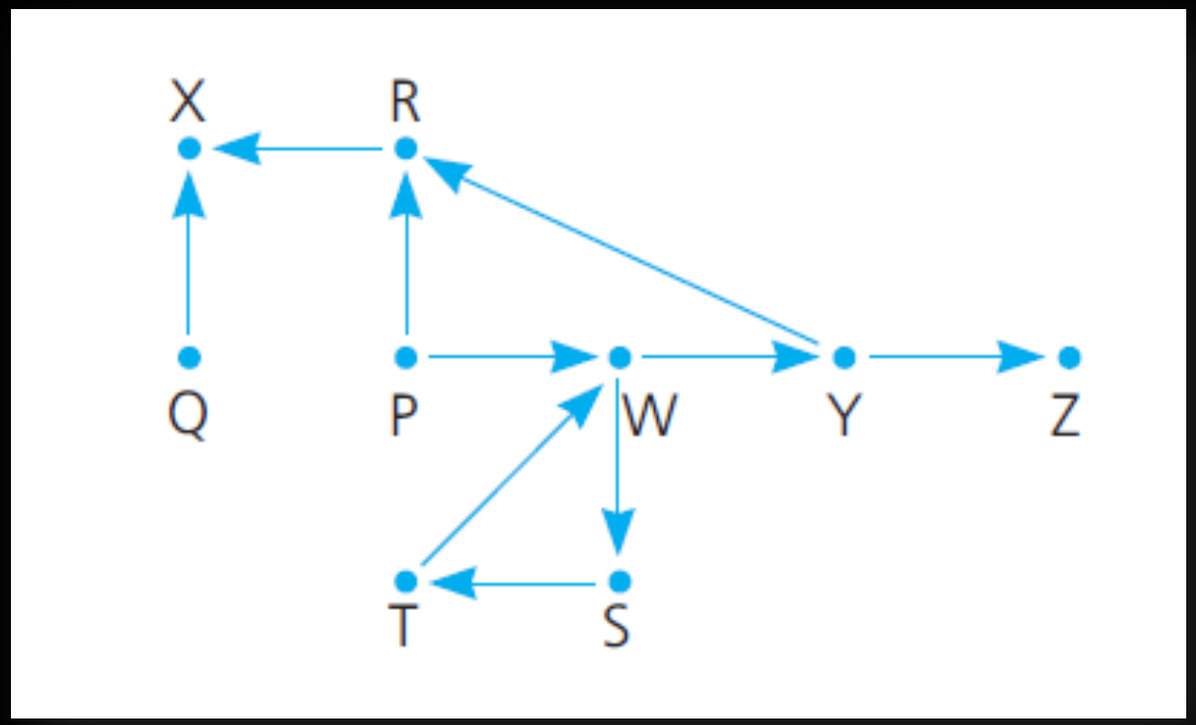
**Origin = P , Destination = Z**

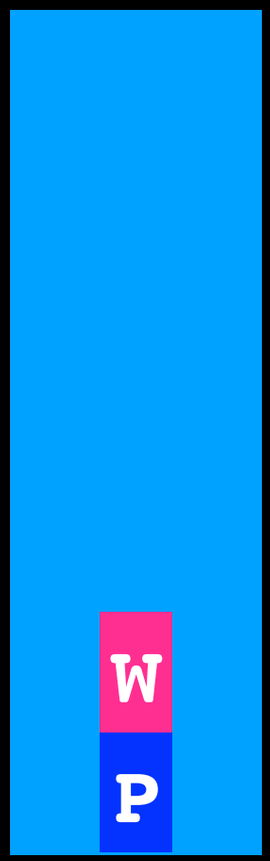


# Backtracking

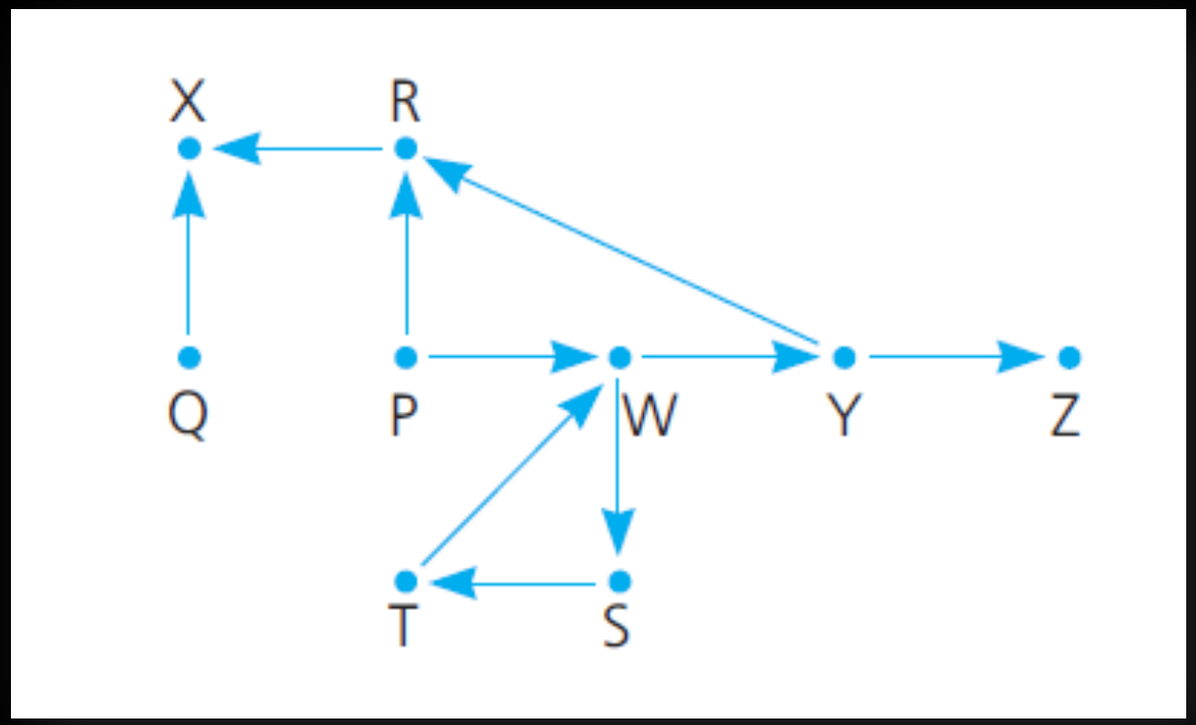
**Origin = P , Destination = Z**

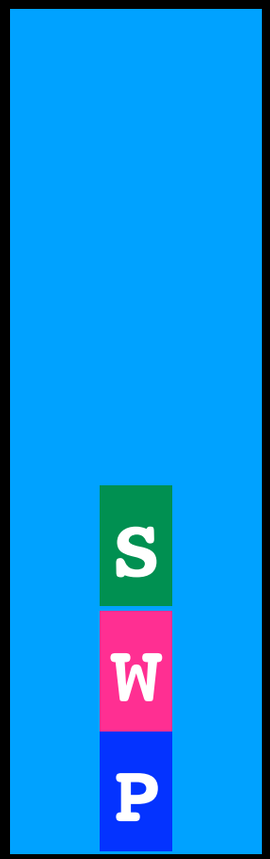


# Backtracking

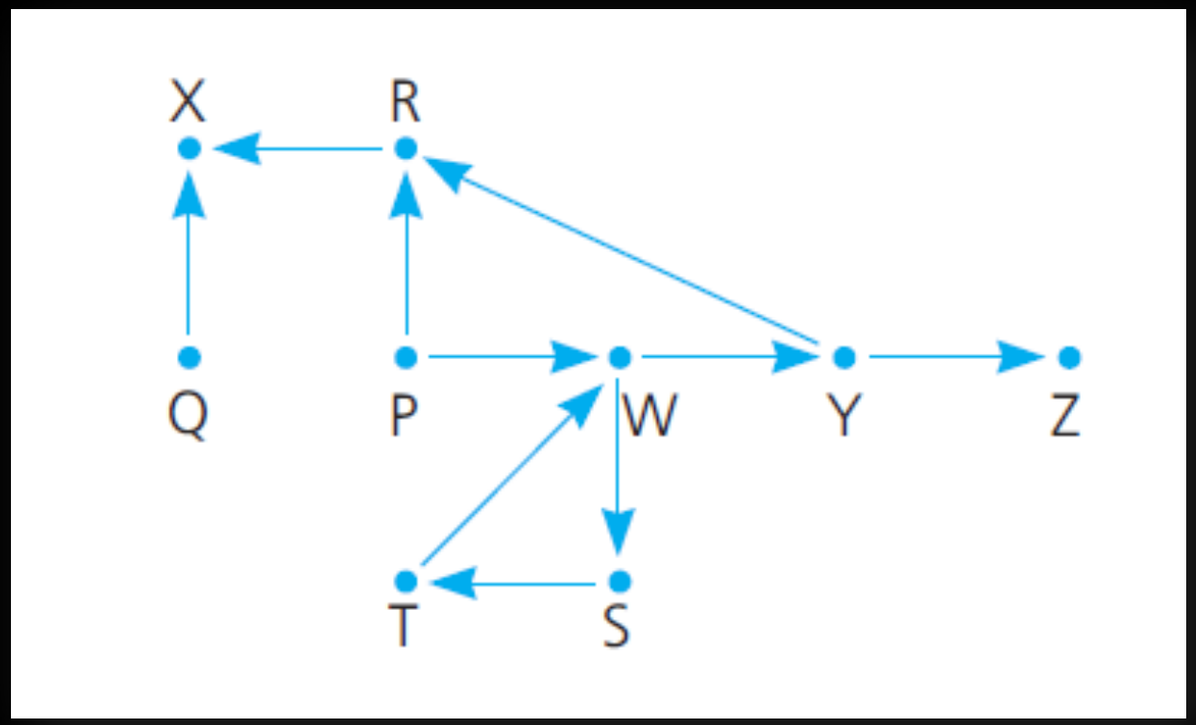
**Origin = P , Destination = Z**

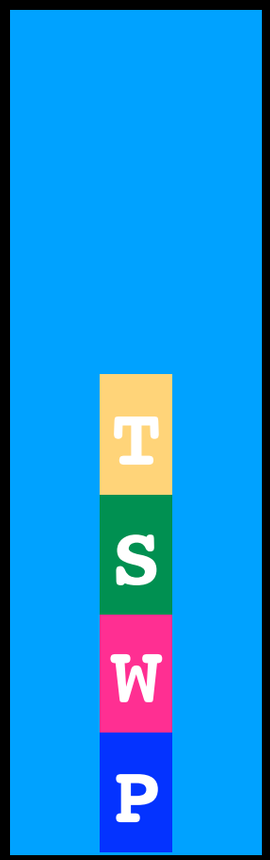


# Backtracking

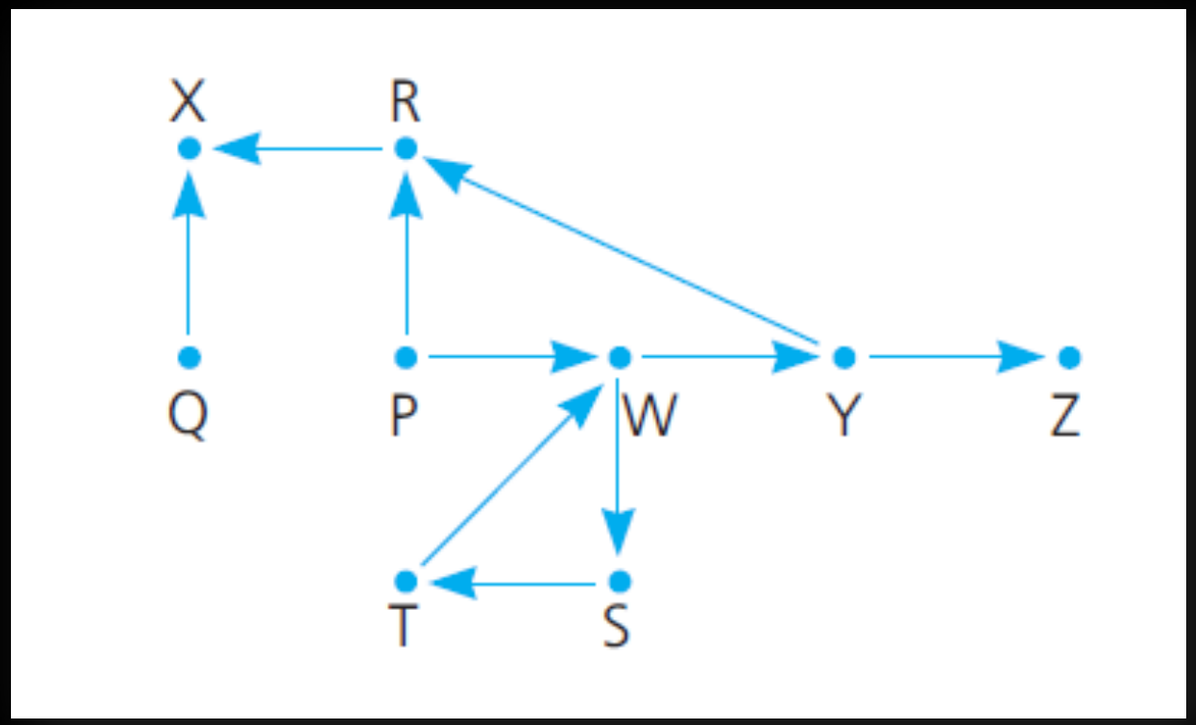
**Origin = P , Destination = Z**

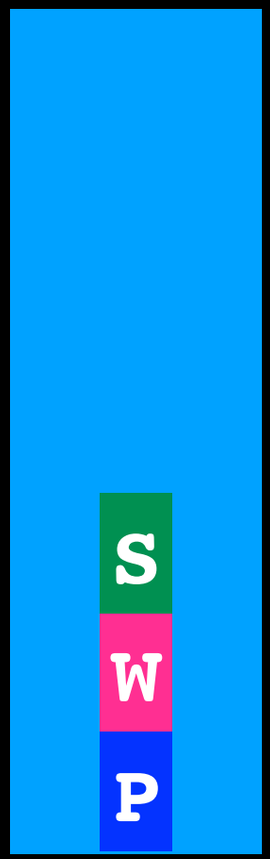


# Backtracking

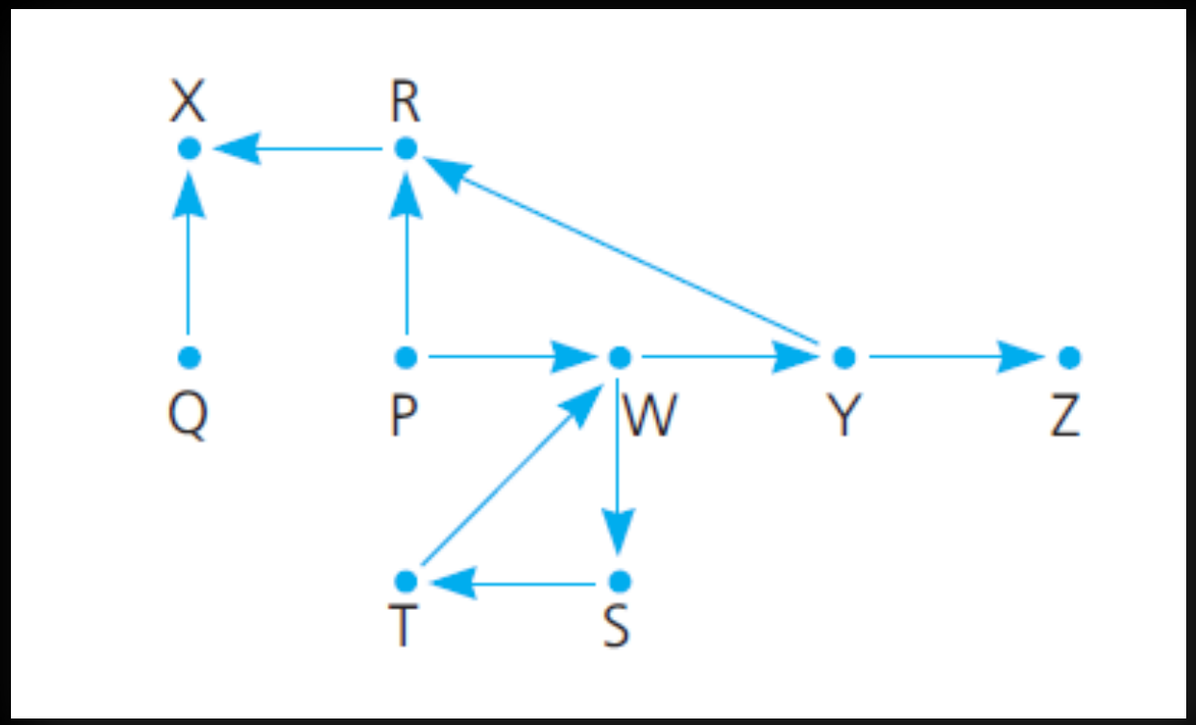
**Origin = P , Destination = Z**

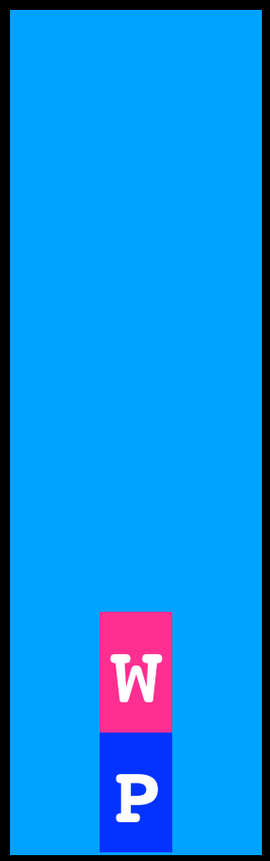


# Backtracking

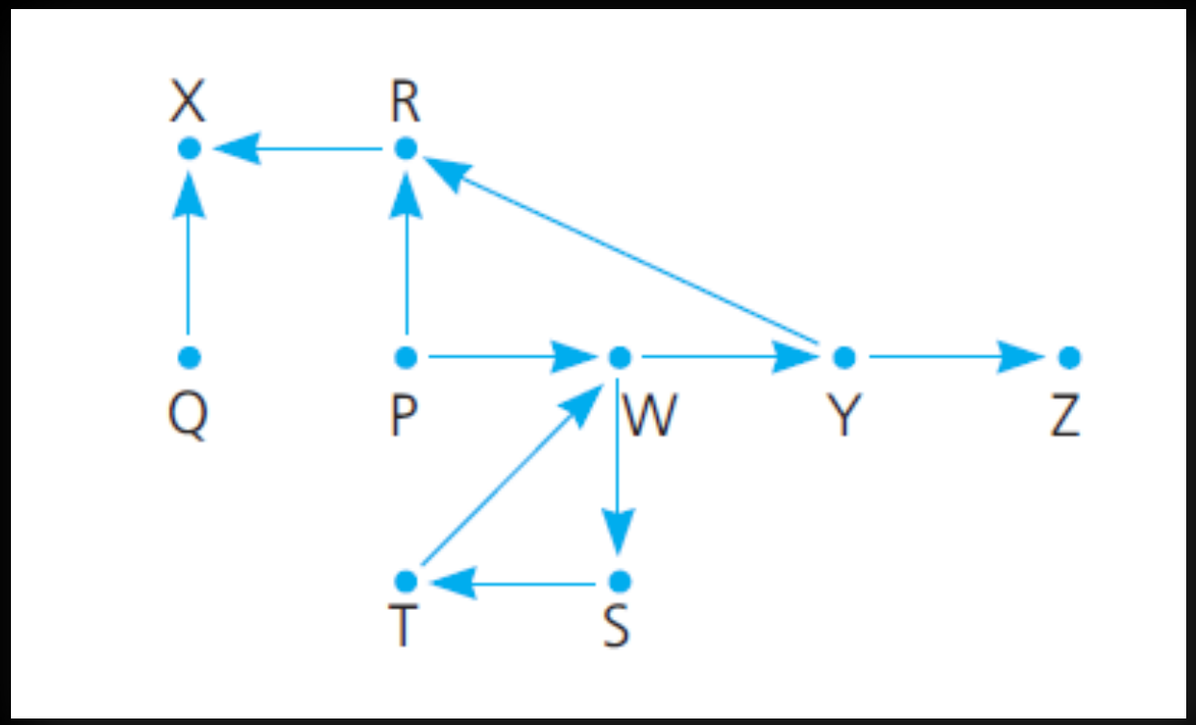
**Origin = P , Destination = Z**

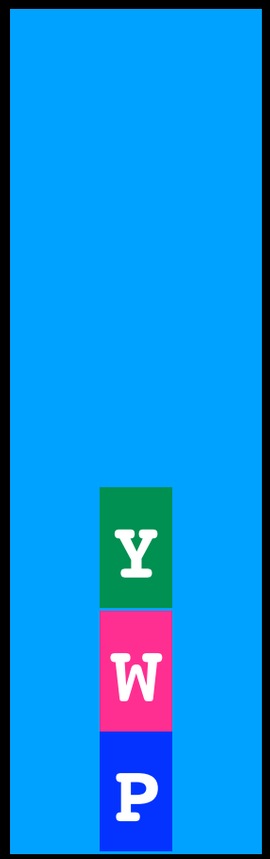


# Backtracking

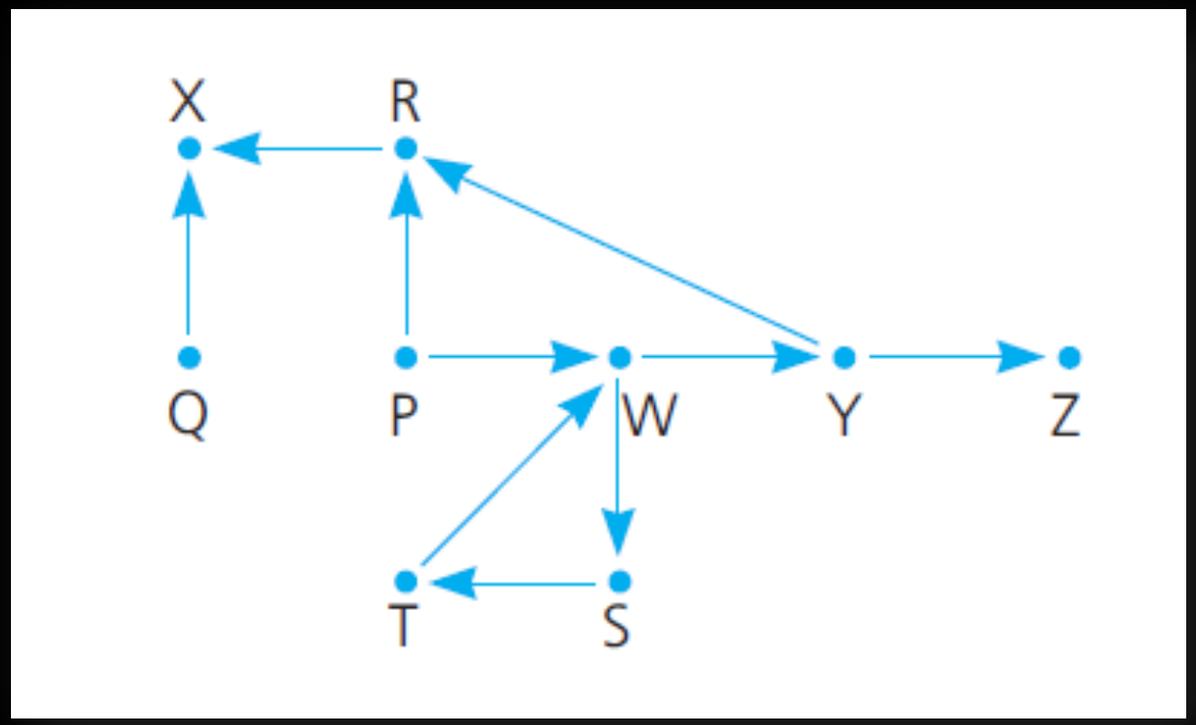
**Origin = P , Destination = Z**

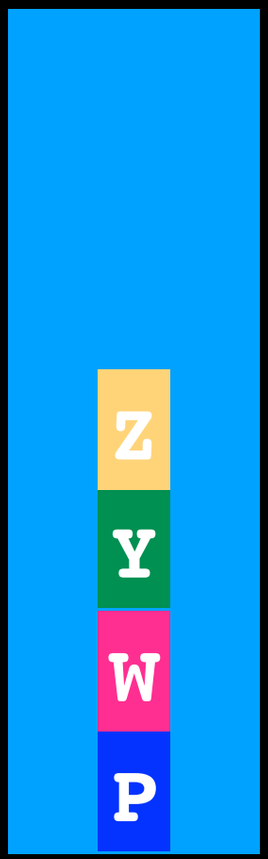


# Backtracking

**Origin = P , Destination = Z**



# Backtracking

```
while(not found flights from origin to destination)
{
   if no flight exists from city on top of stack to
   unvisited destination
       pop the stack //BACKTRACK
   else
   {
       select an unvisited city C accessible from city
       currently at top of stack
       push C on stack
       mark C as visited
    }
}
```

# Program Stack and Recursion

Recursion works because function waining for result/ return from recursive call are on program stack

Order of execution determined by **stack**

# More Applications

**Balancing** anything!

- html tags (e.g  matches  )
- parsers in general

**Reverse** characters in a word or words in a sentence

**Undo** mechanism for editors or backups

**Traversals** (graphs / trees)

. . .

# Stack ADT

```
#ifndef STACK_H_
#define STACK_H_

template<class T>
class Stack
{

public:
    Stack();
    void push(const T& new_entry); // adds an element to top of stack
    void pop(); // removes element from top of stack
    T top() const; // returns a copy of element at top of stack
    int size() const; // returns the number of elements in the stack
    bool isEmpty() const; // returns true if no elements on stack false otherwise

private:
          //implementation details here


};    //end Stack

#include "Stack.cpp"
#endif // STACK_H_`
```

# Abstract Data Types

Bag

List

Stack

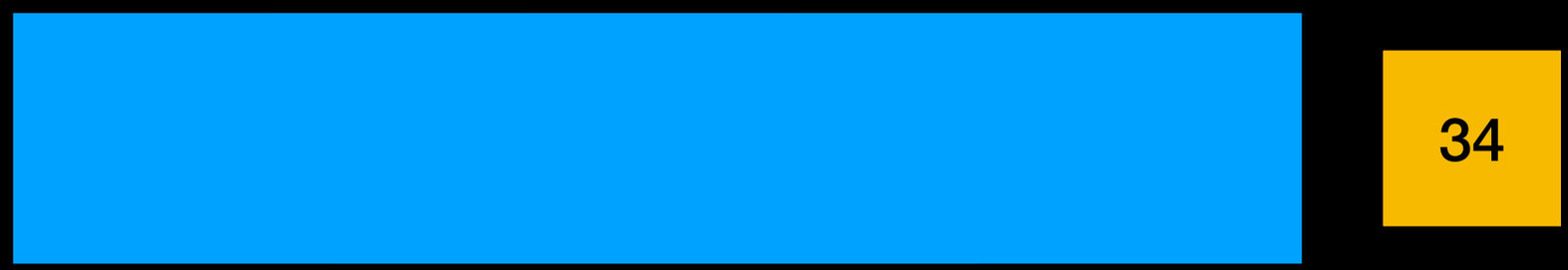
Queue

# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

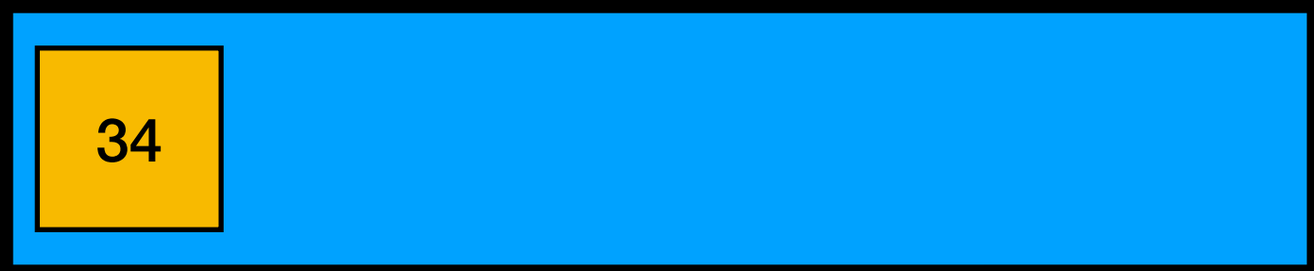


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

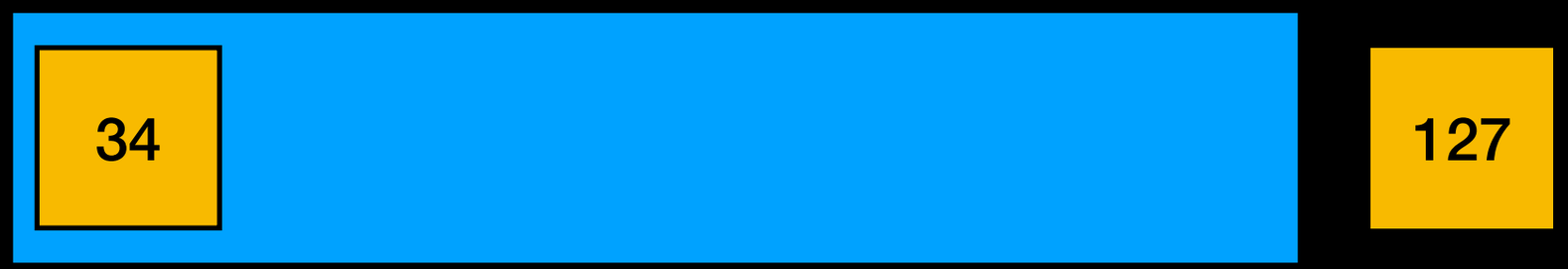


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

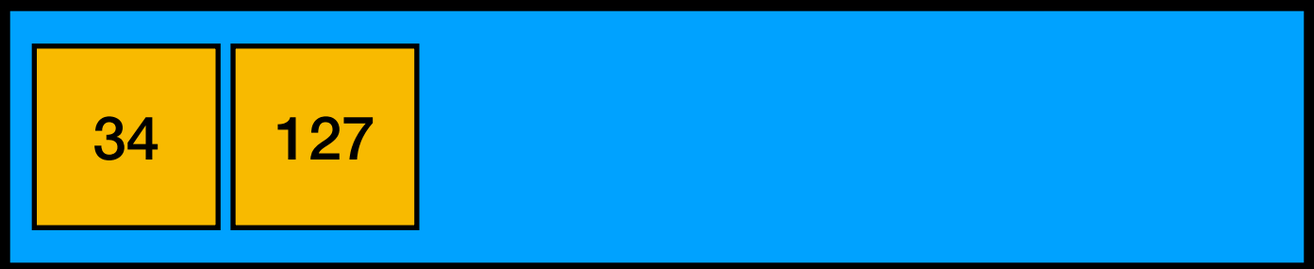


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

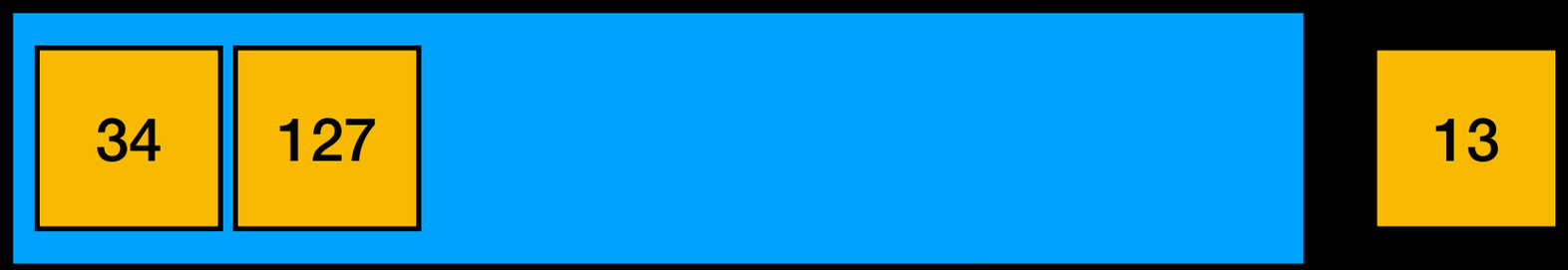


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

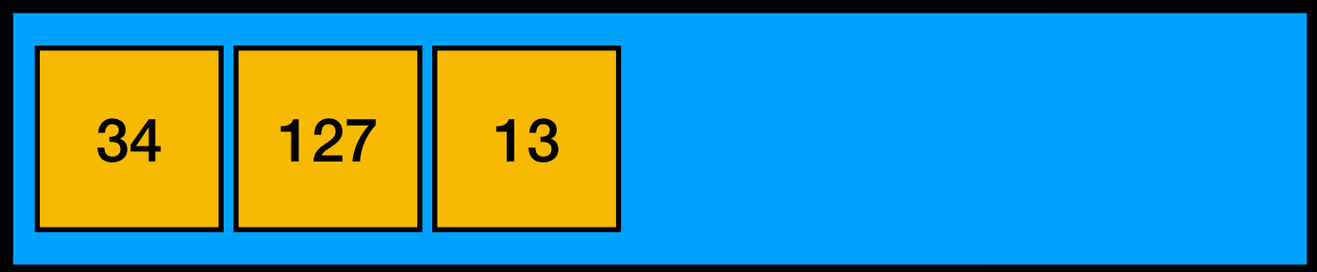


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

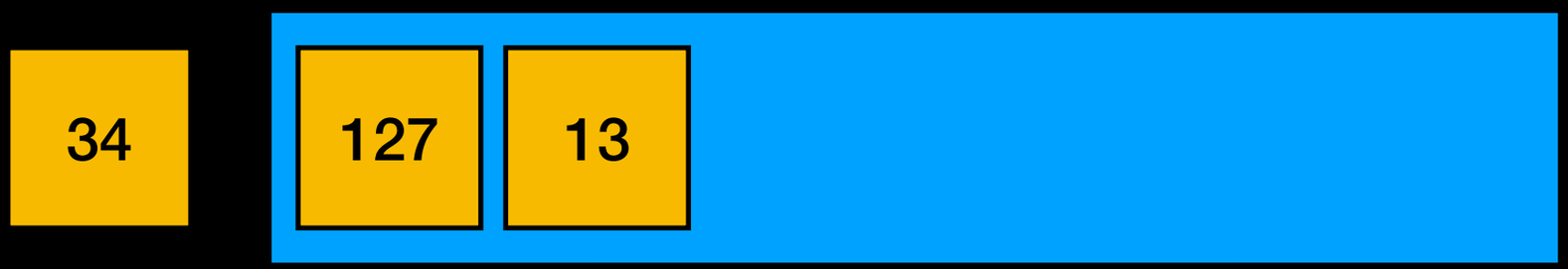


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

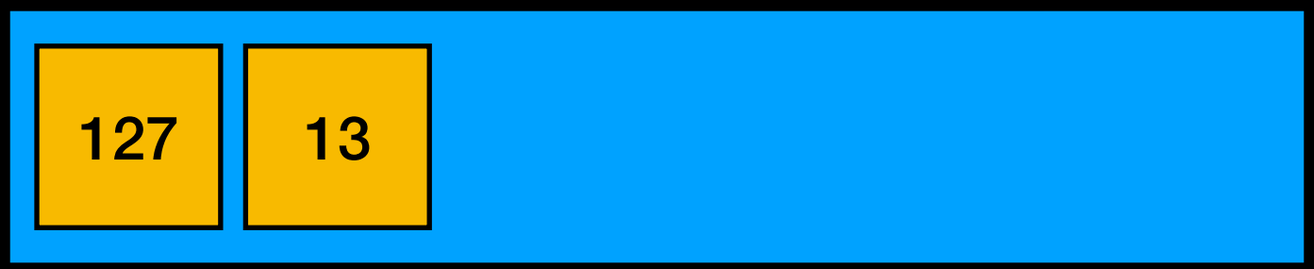


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

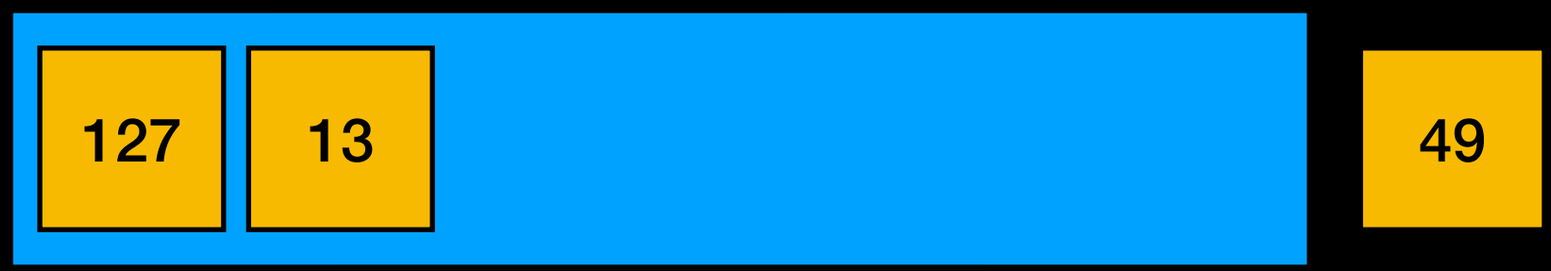


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line

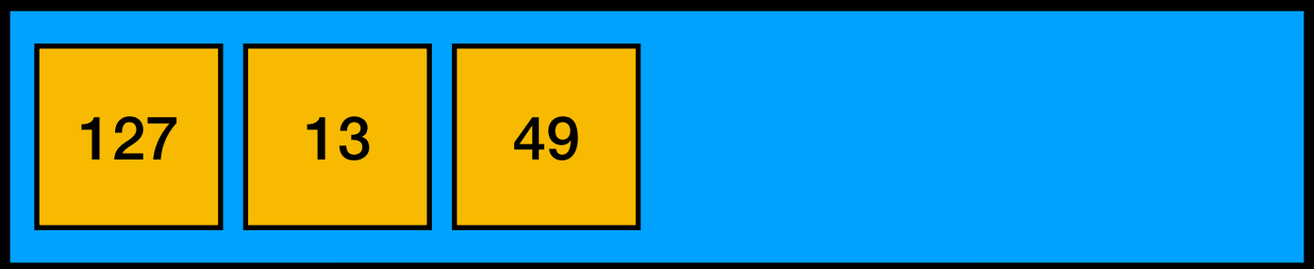


# Queue

An ADT representing a waiting line

Objects can be enqueued to the back of the line

or dequeued from the front of the line



# Queue

An ADT 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

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

. . .

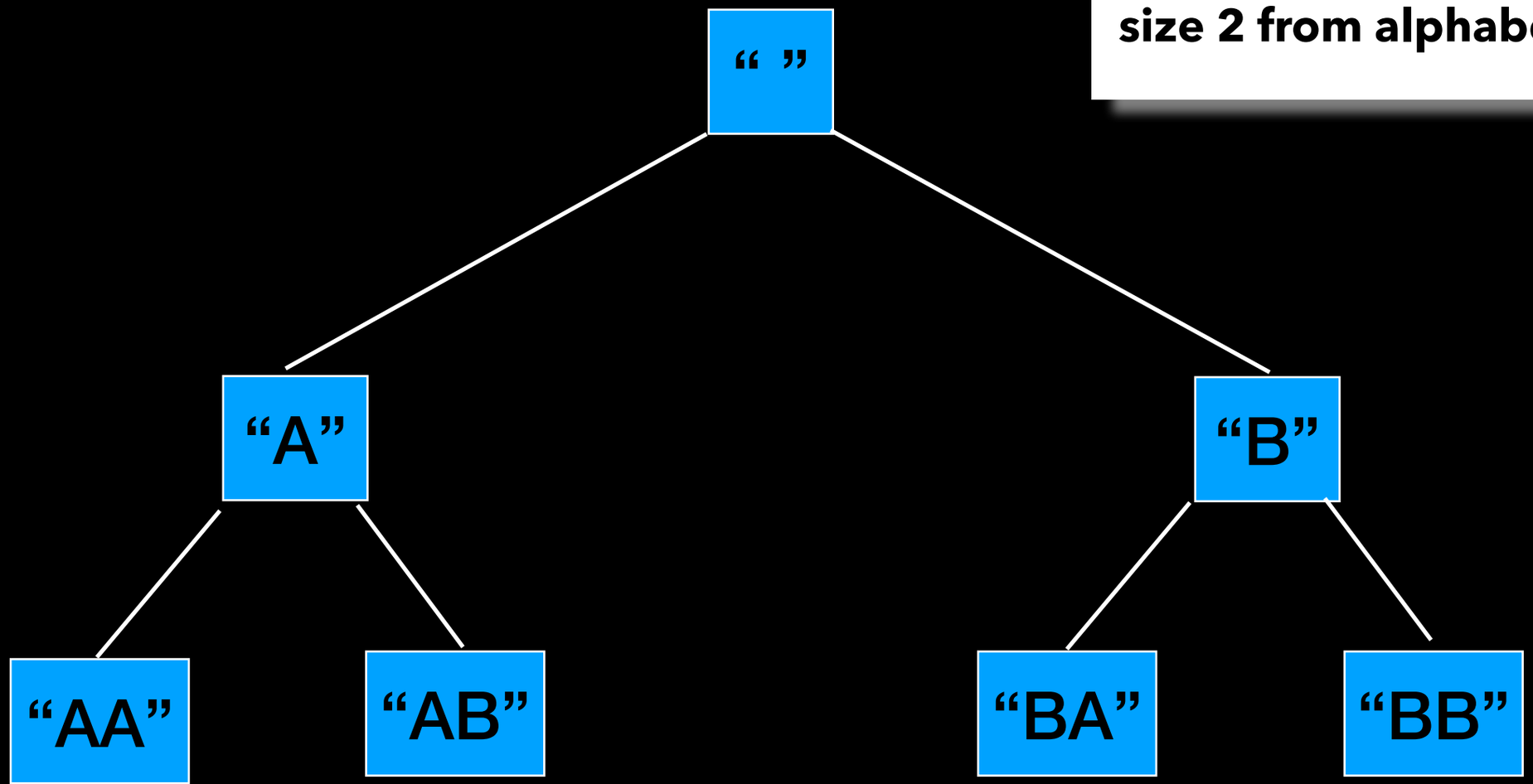
# Generating all substrings

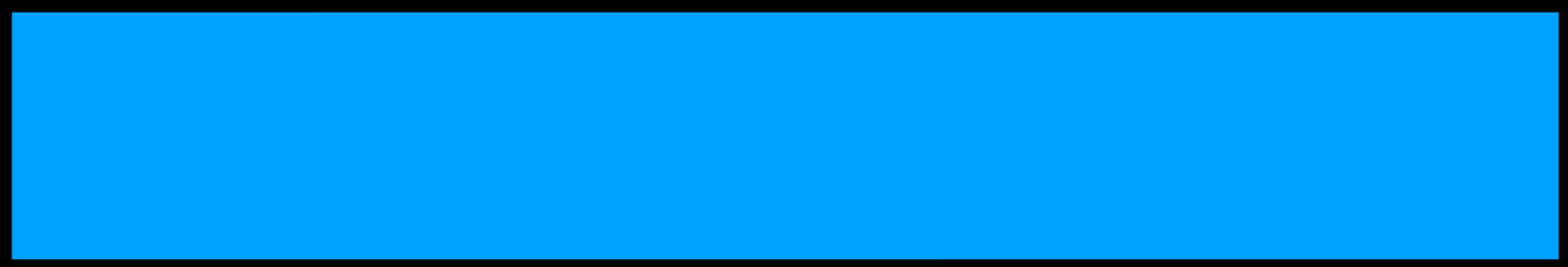
Generate all possible strings **up to** some fixed length **n with repetition (same character included multiple times)**

How might we do it with a queue?

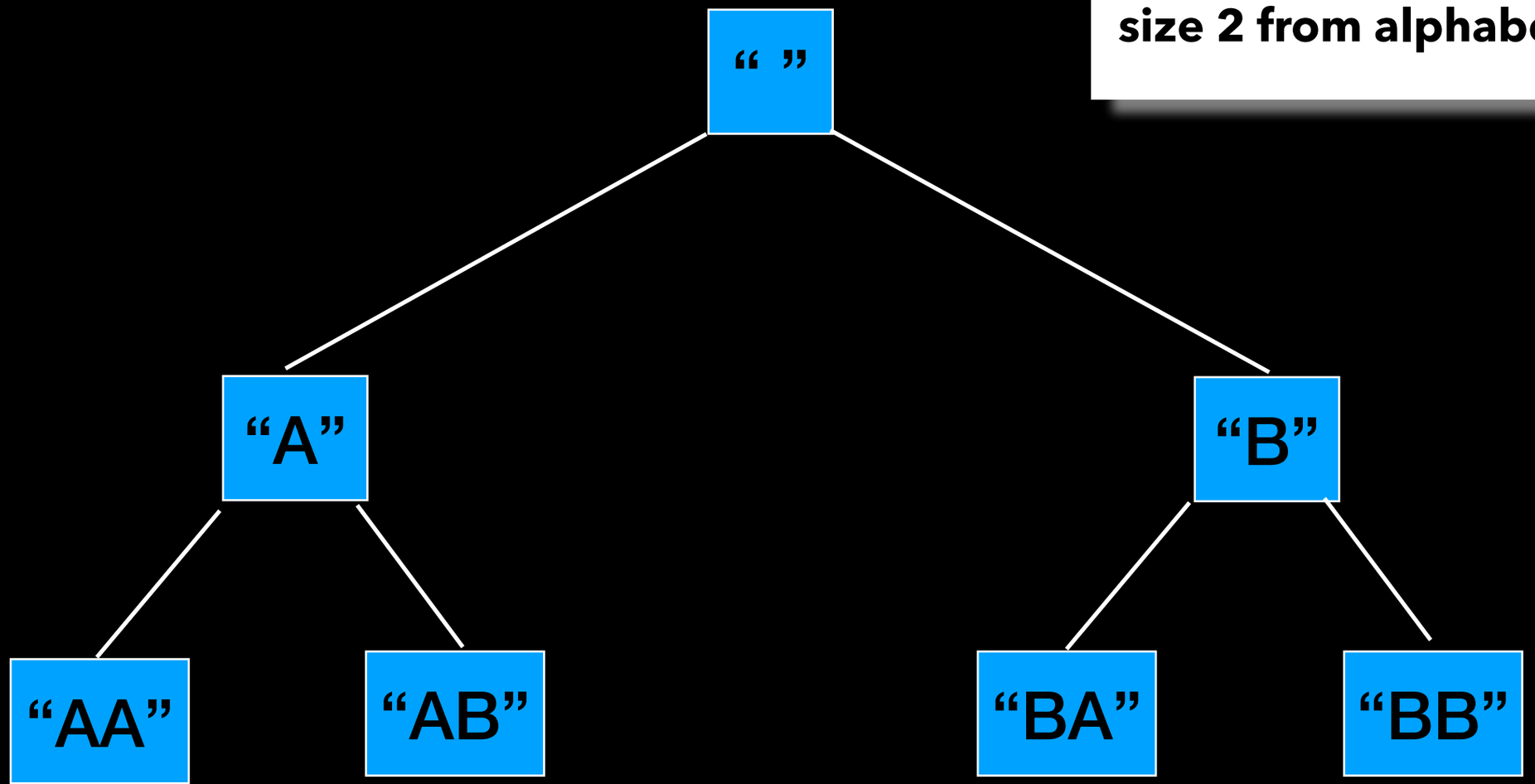
Example simplified to n = 2 and only letters A and B

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

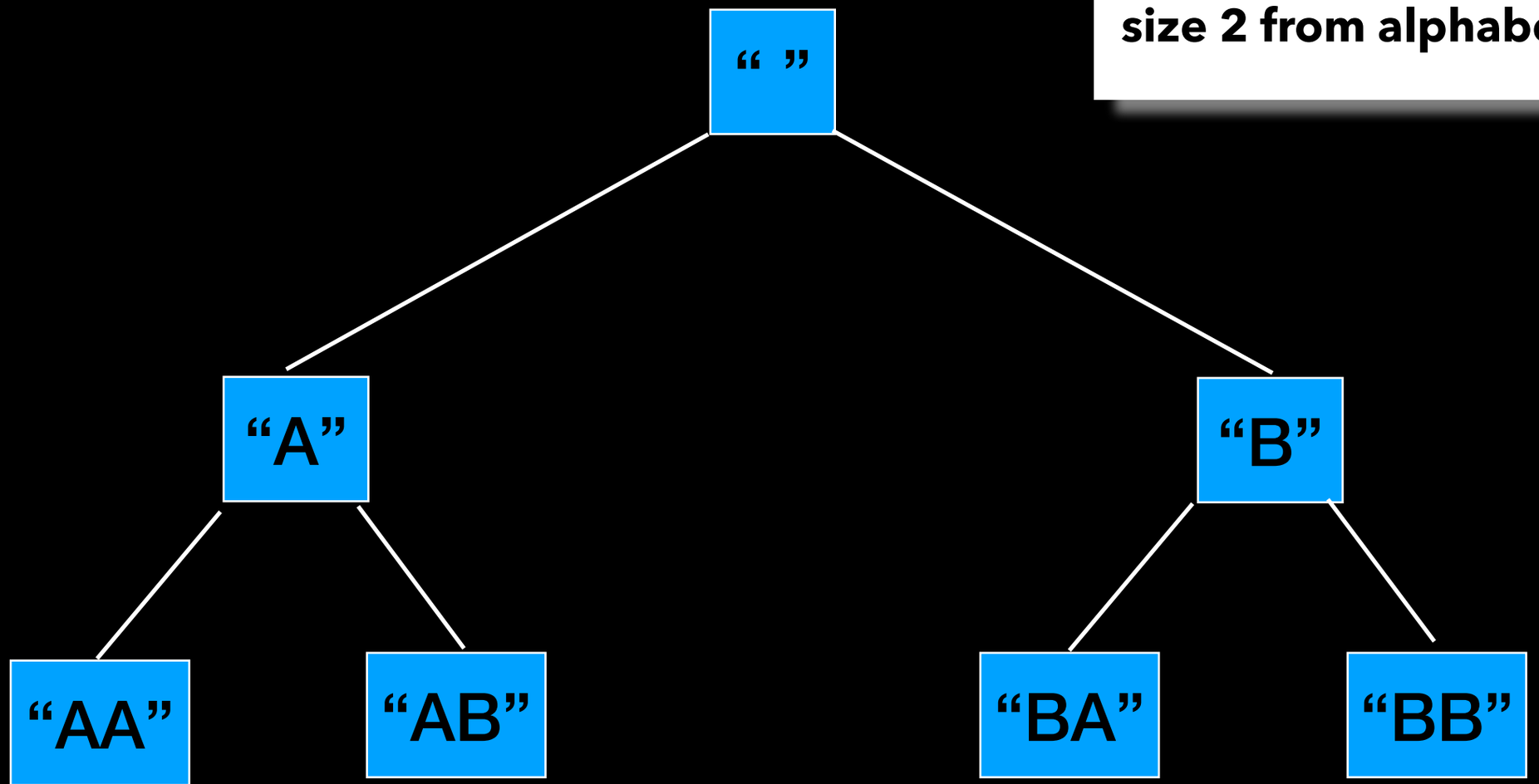


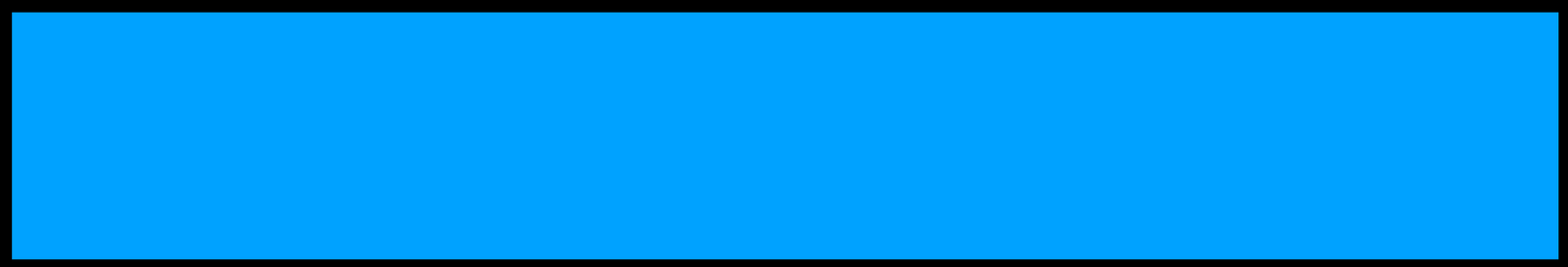
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**




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

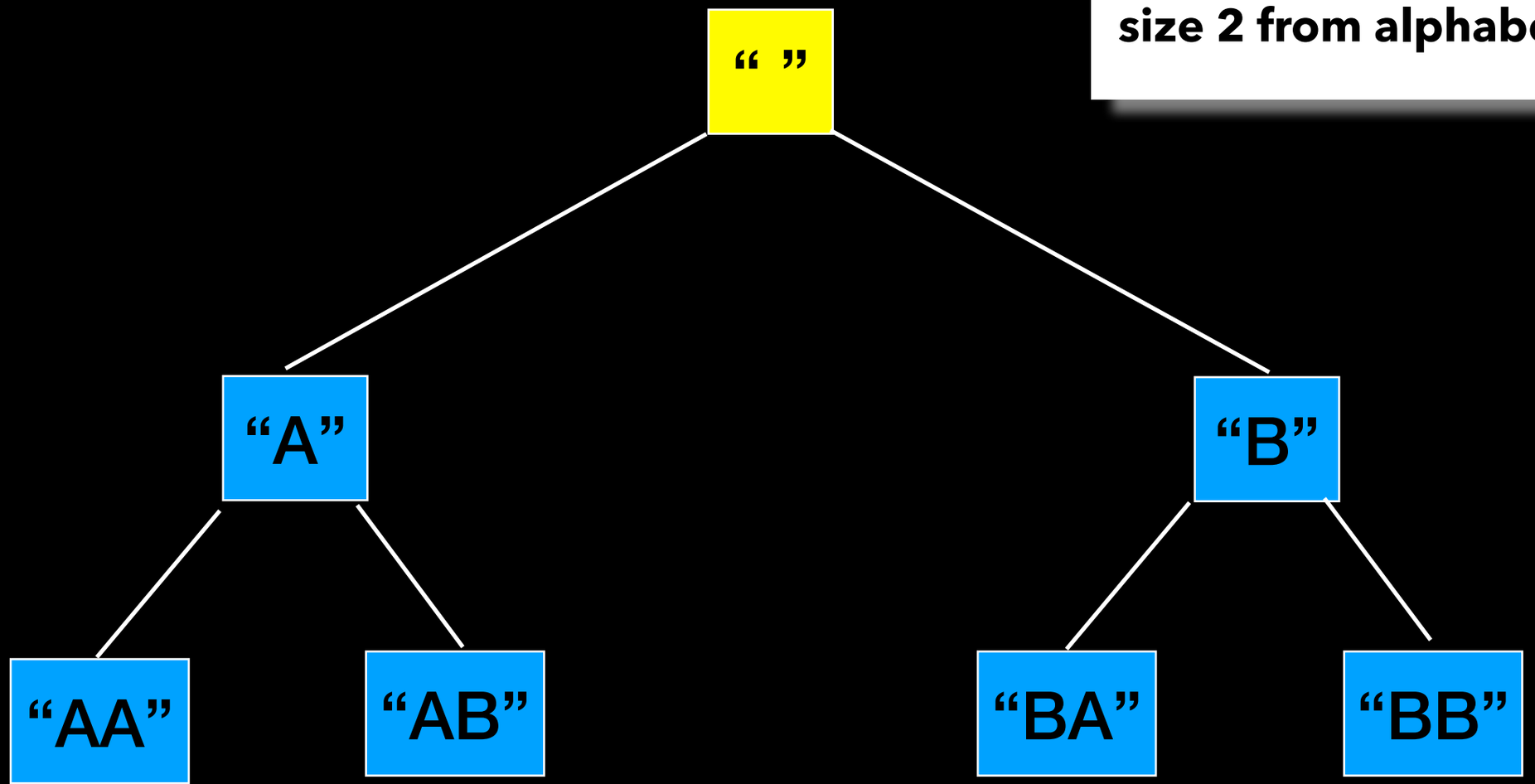


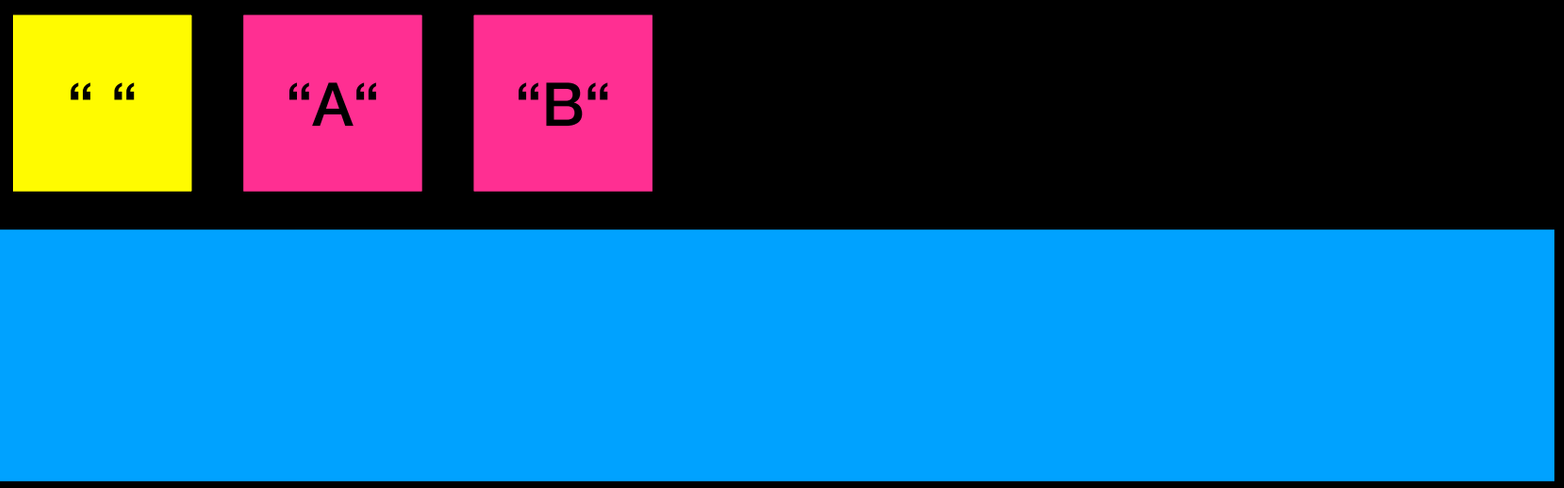




{ “”}

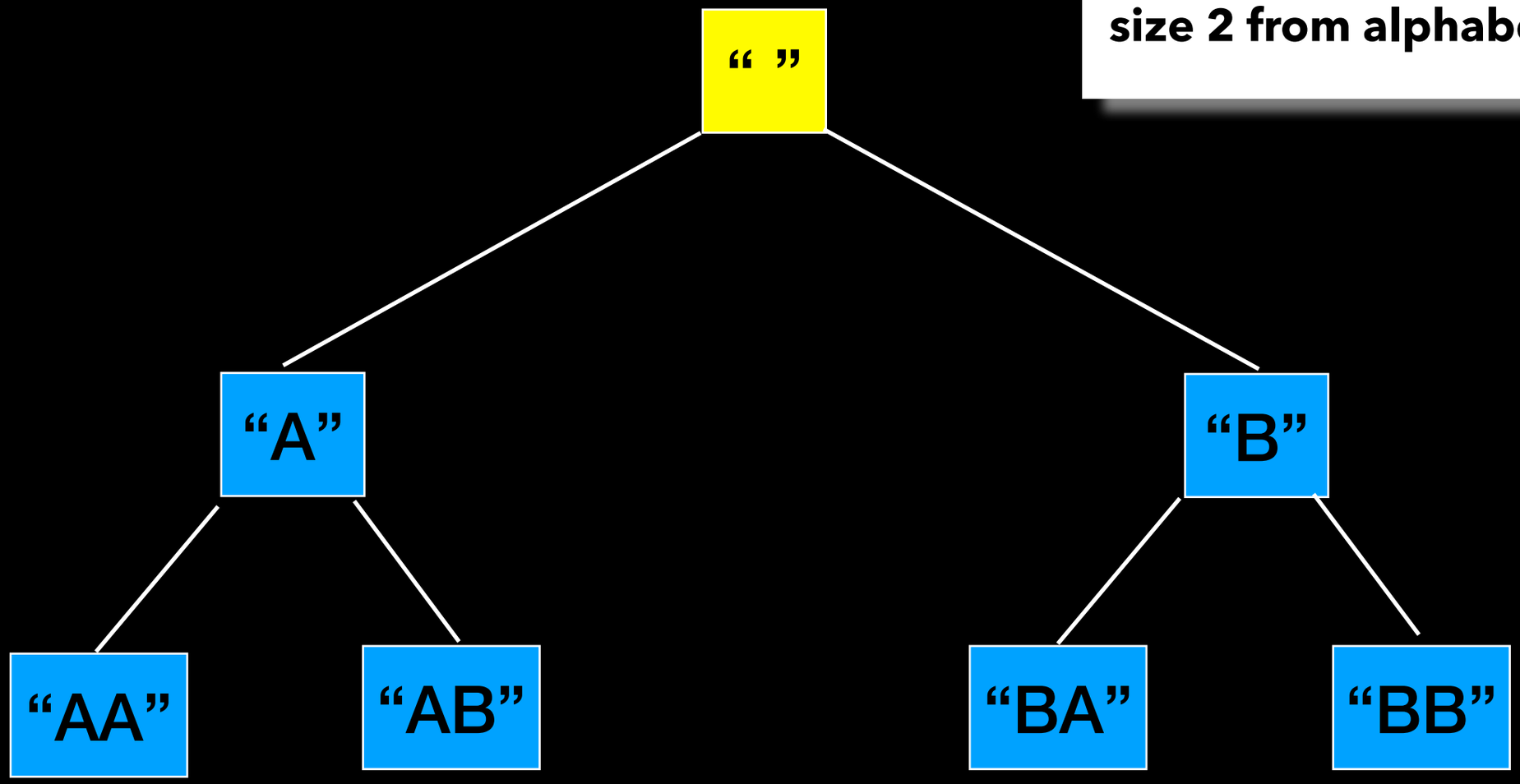
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**

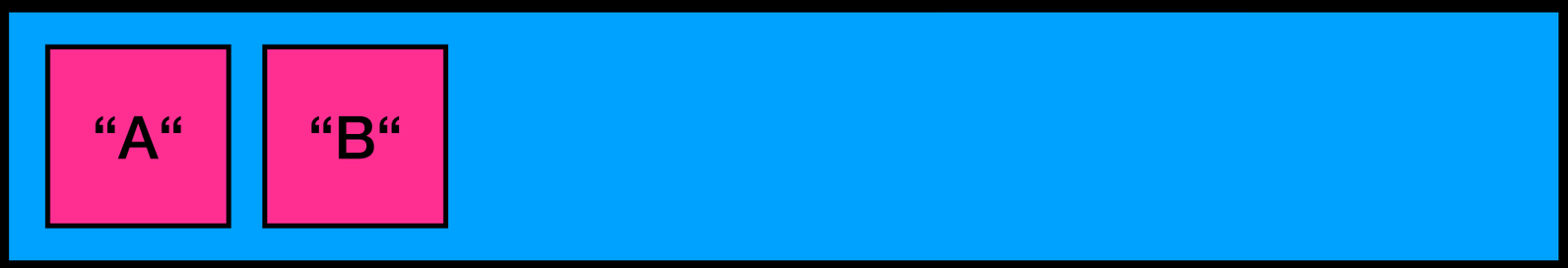


{ “”}

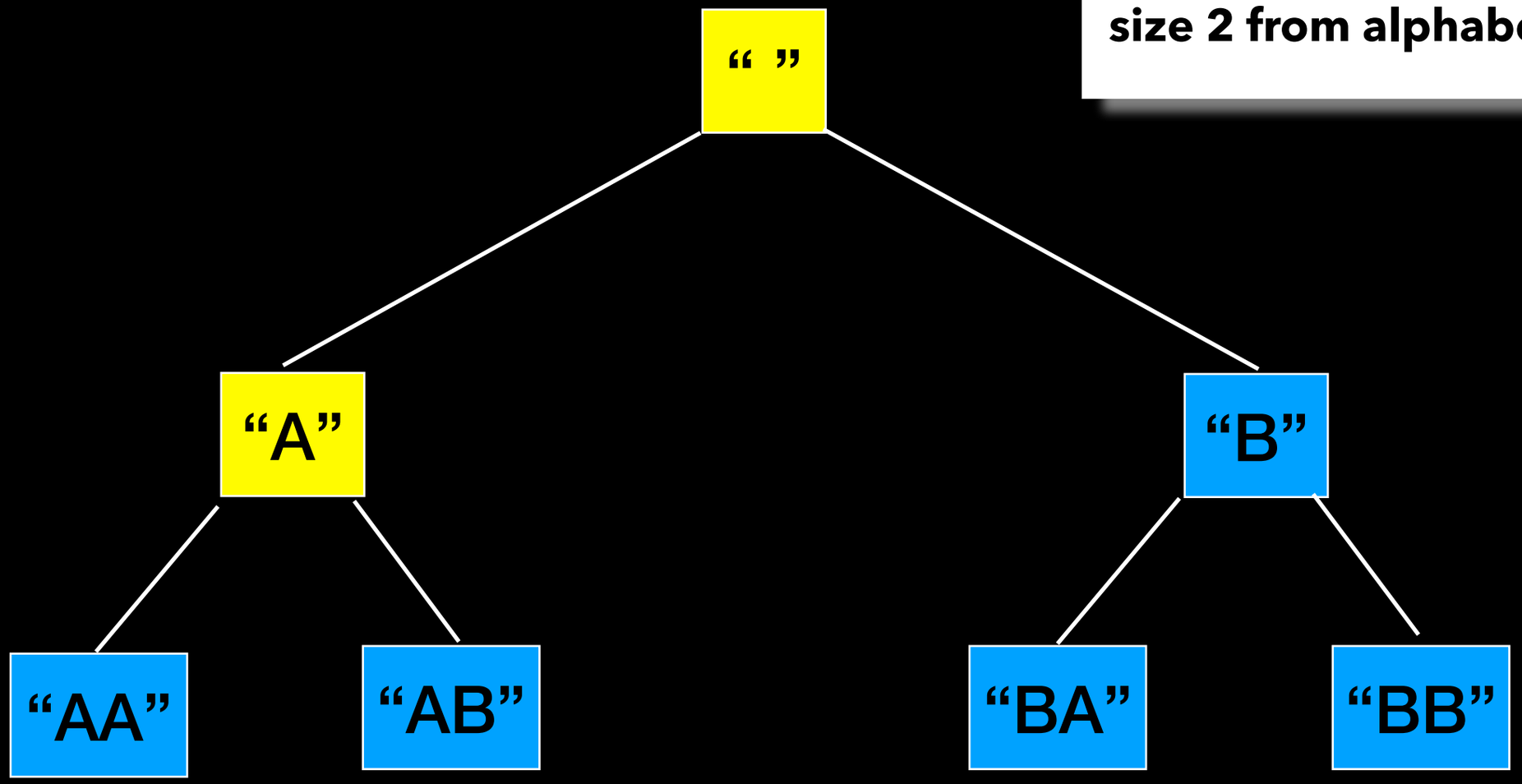
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**

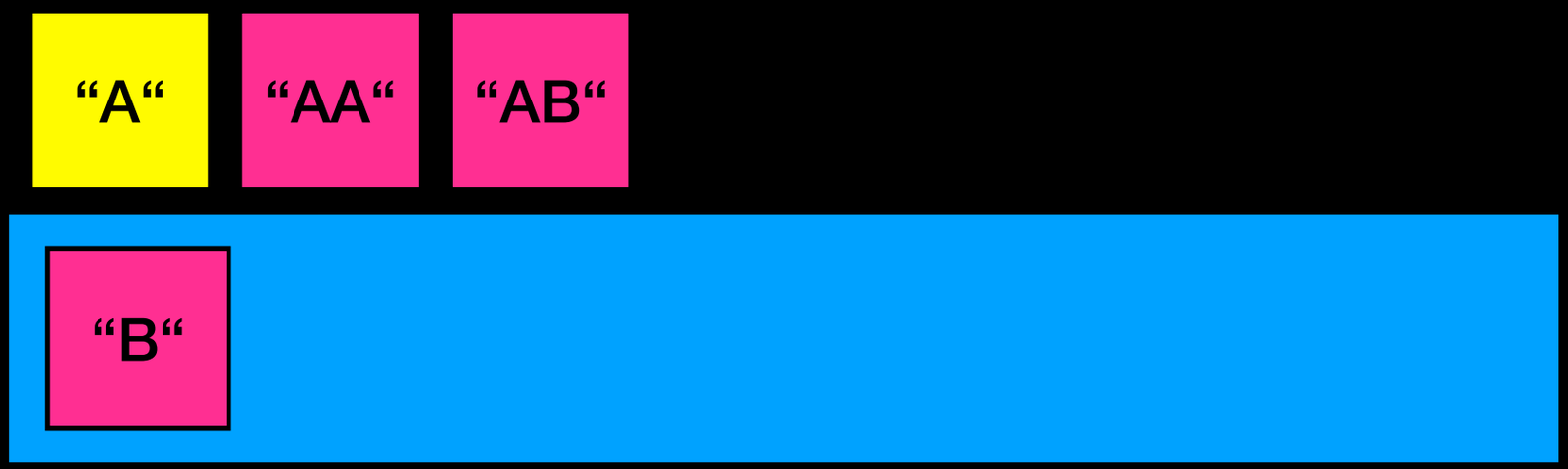


{ “”, “A”}

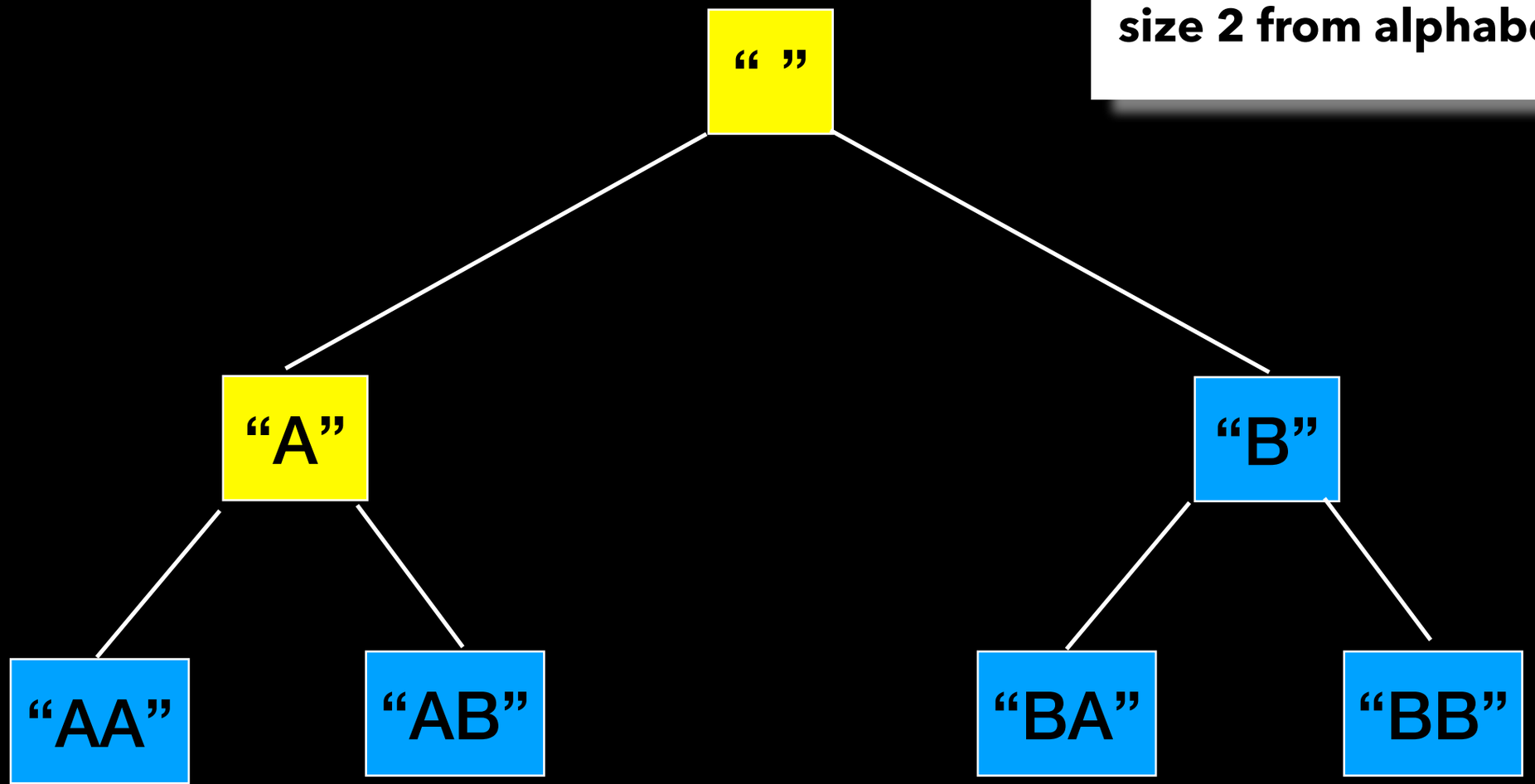
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**

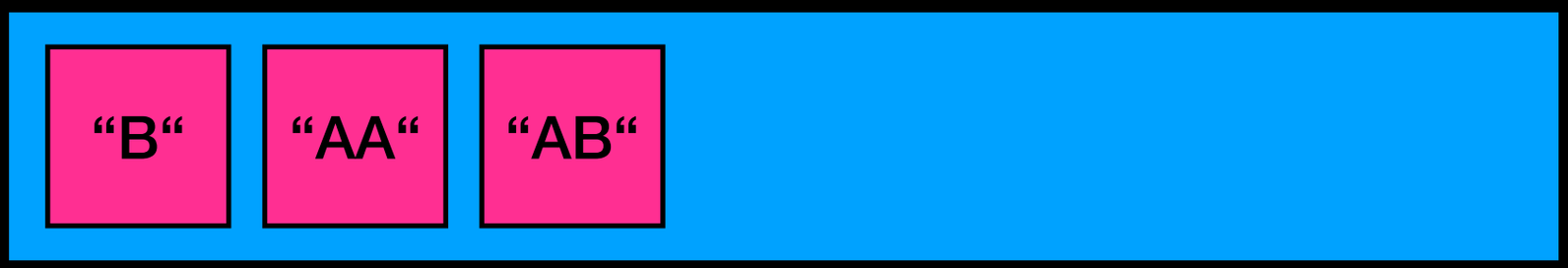


{ “”, “A”}

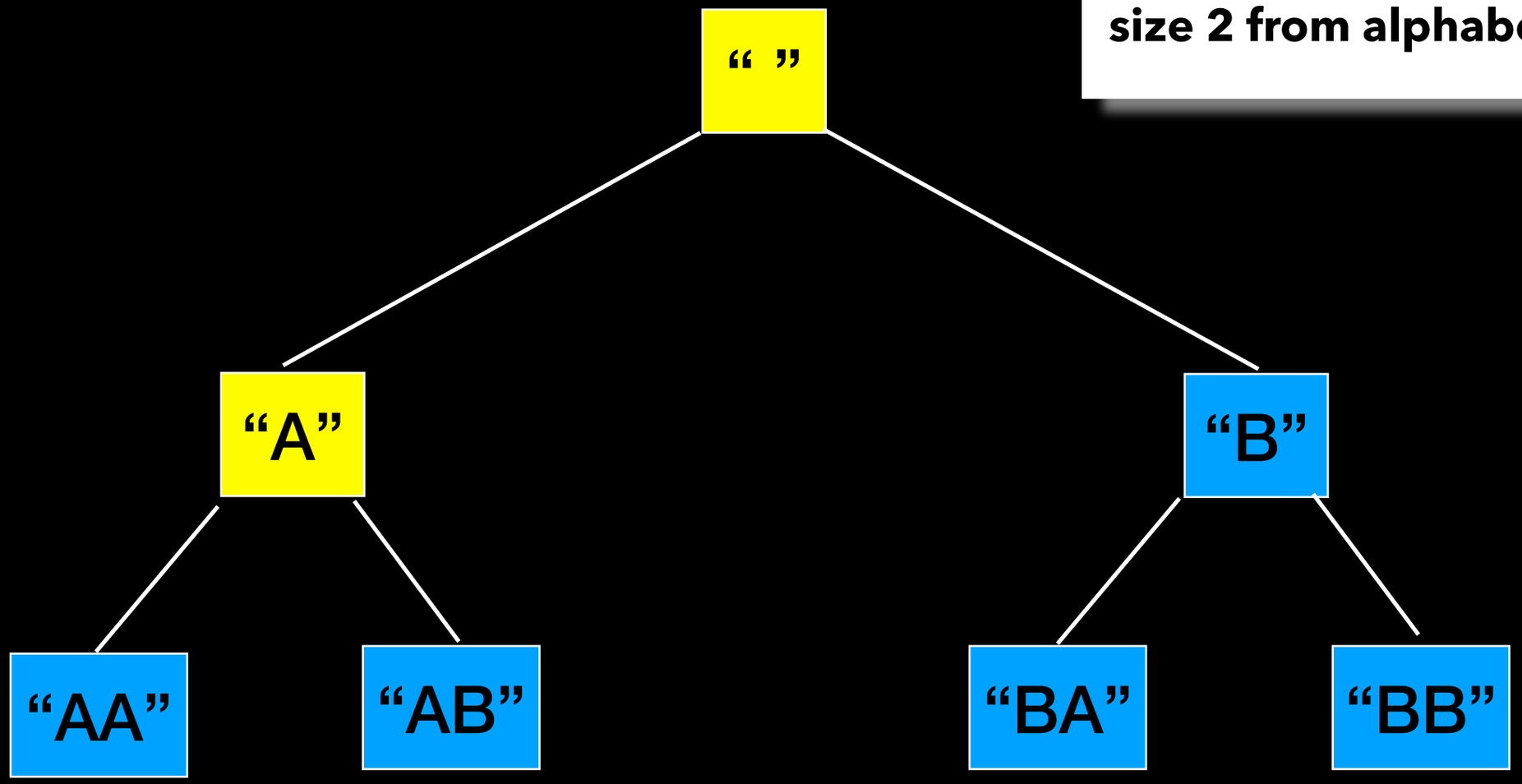
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**



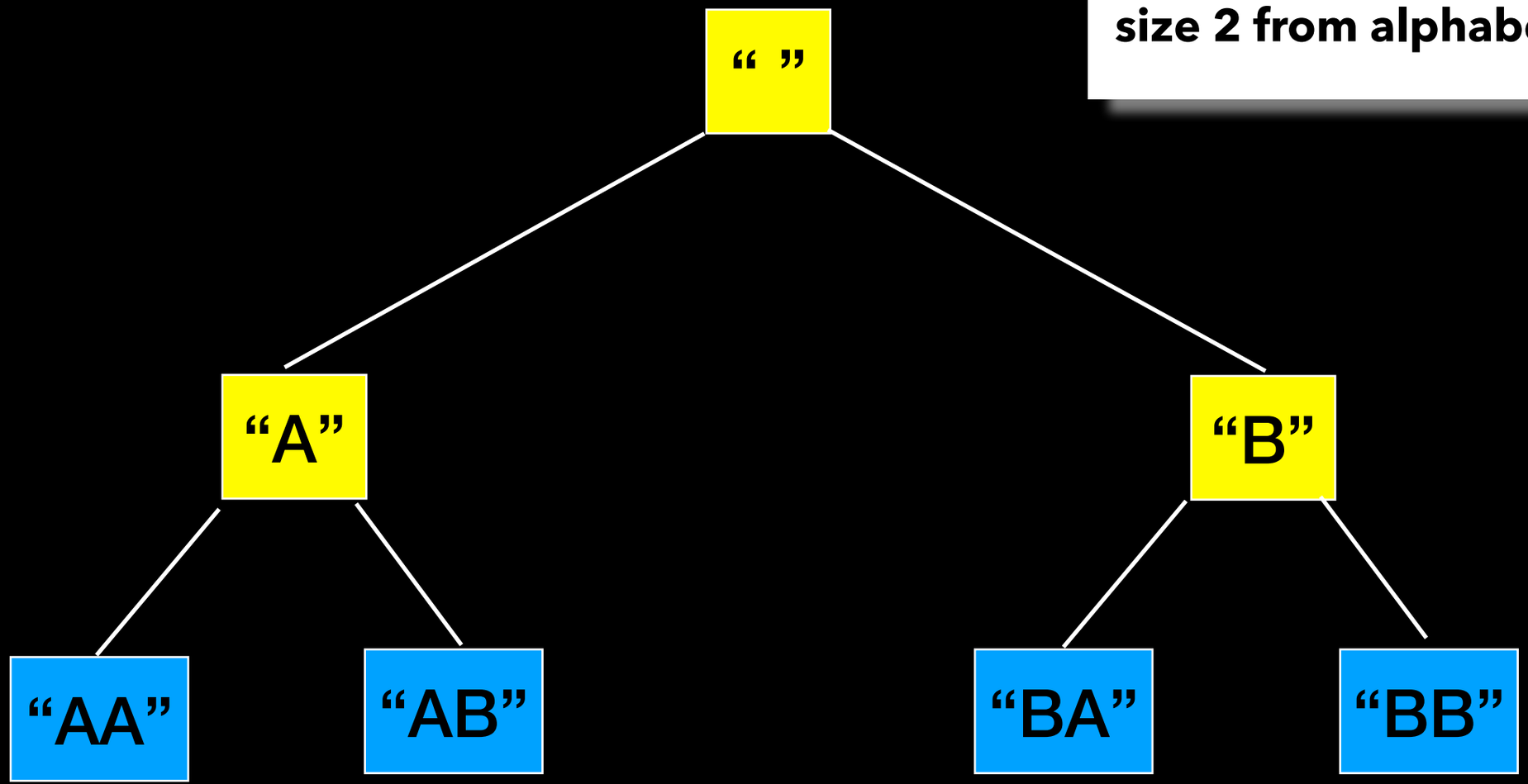
{ “”, “A”}

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




{ “”, “A”, “B”}

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

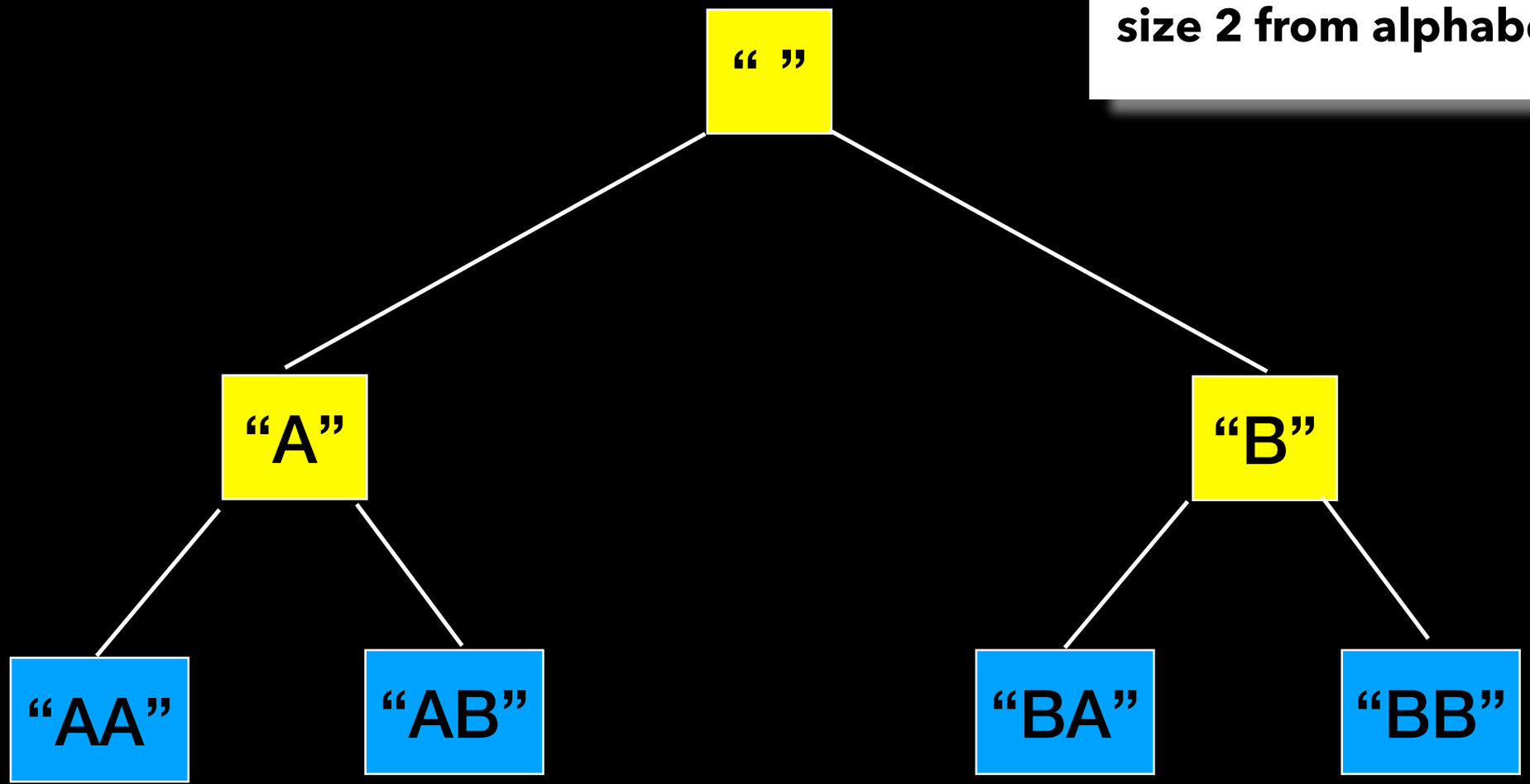


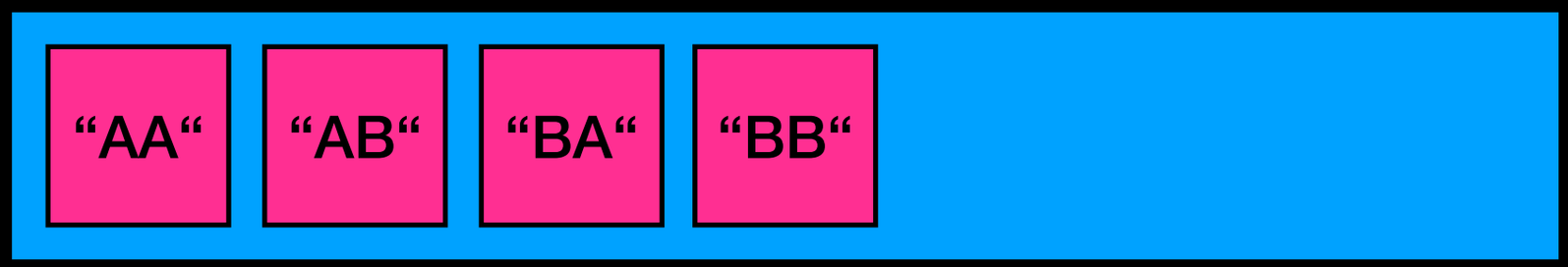

“B“ “BA“ “BB“

“AA“ “AB“

{ “”, “A”, “B”}

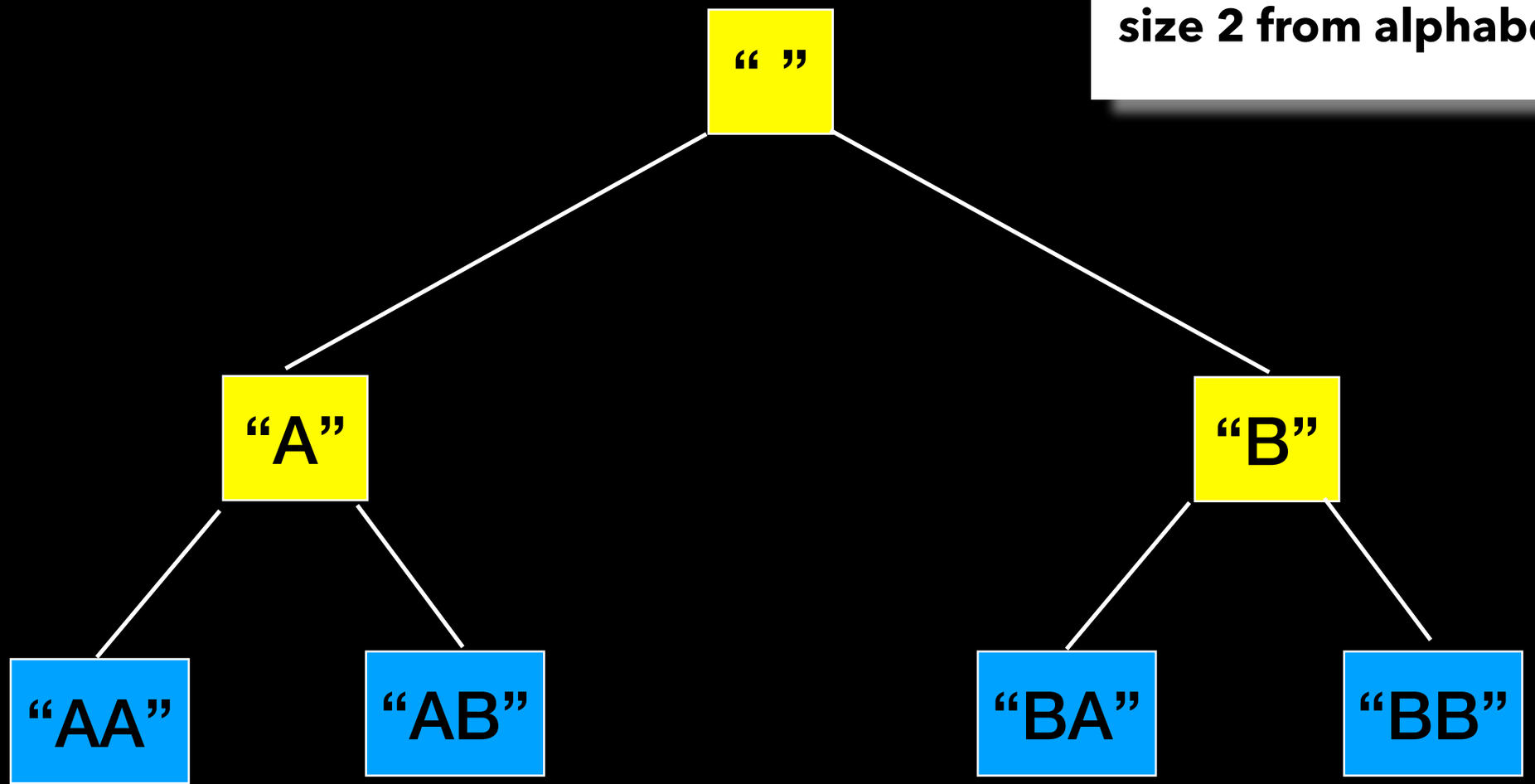
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**

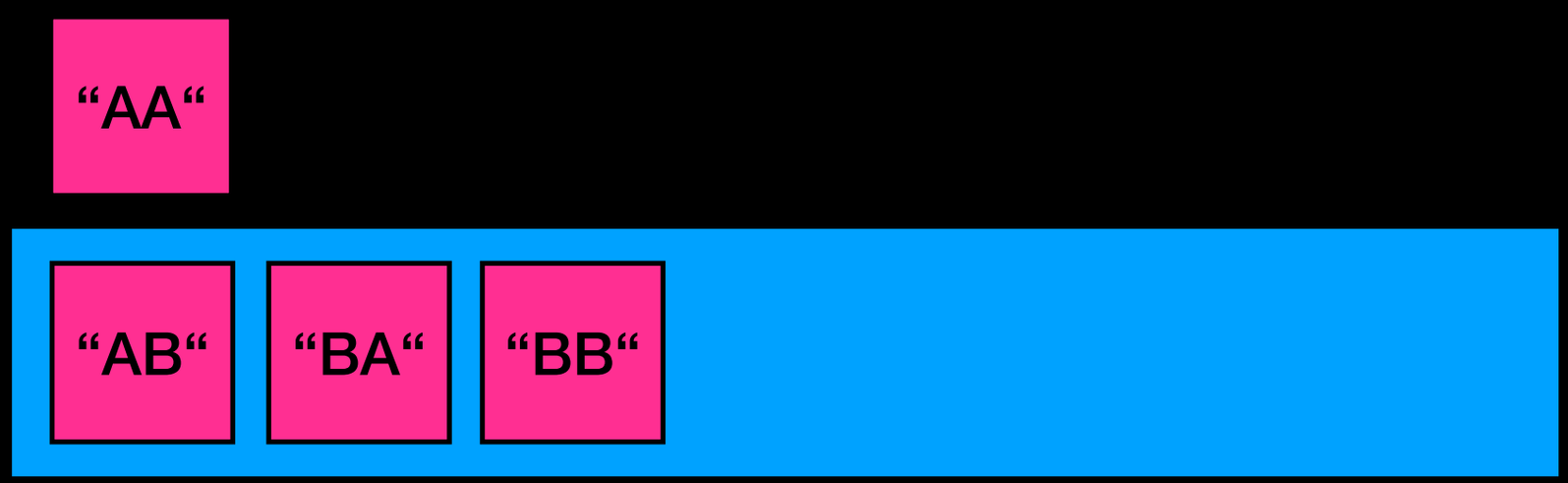


{ “”, “A”, “B”}

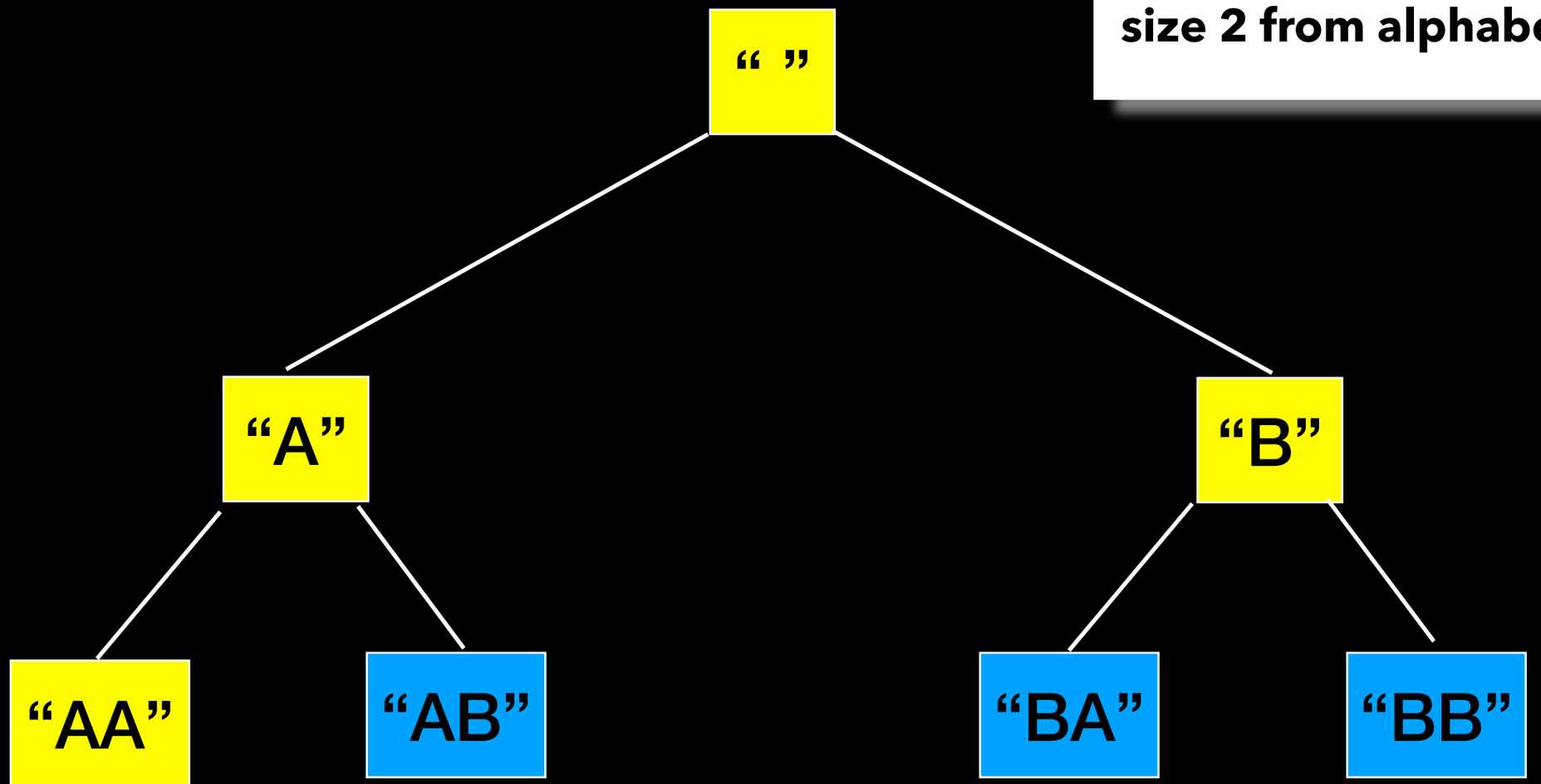
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**

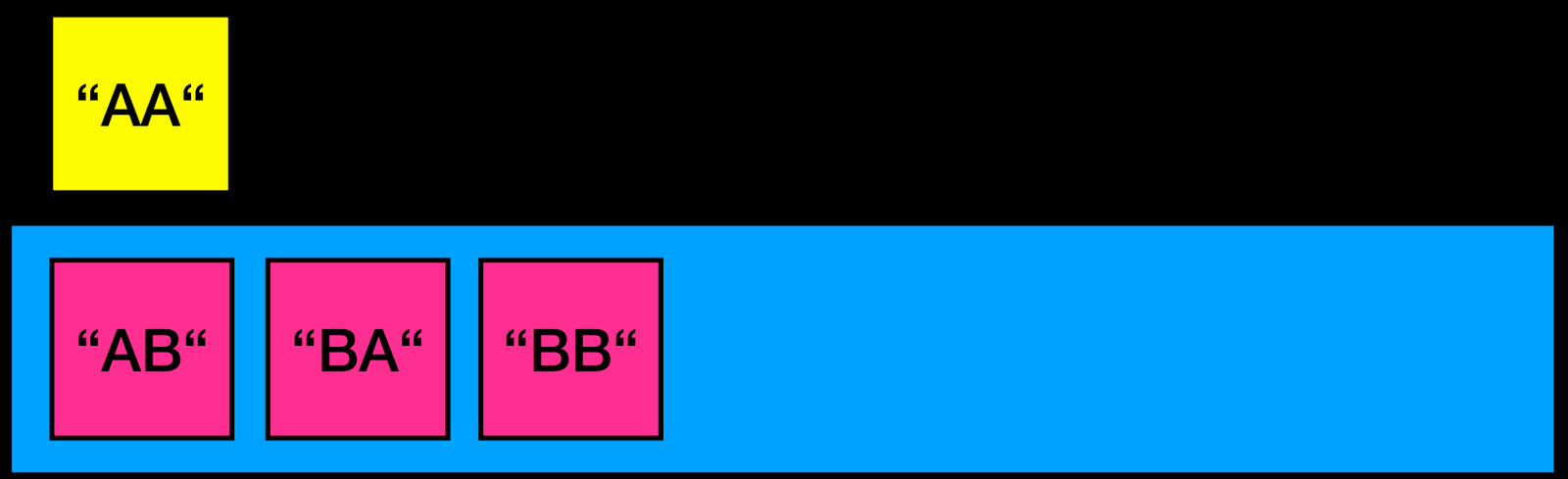


{ “”, “A”, “B”, “AA”}

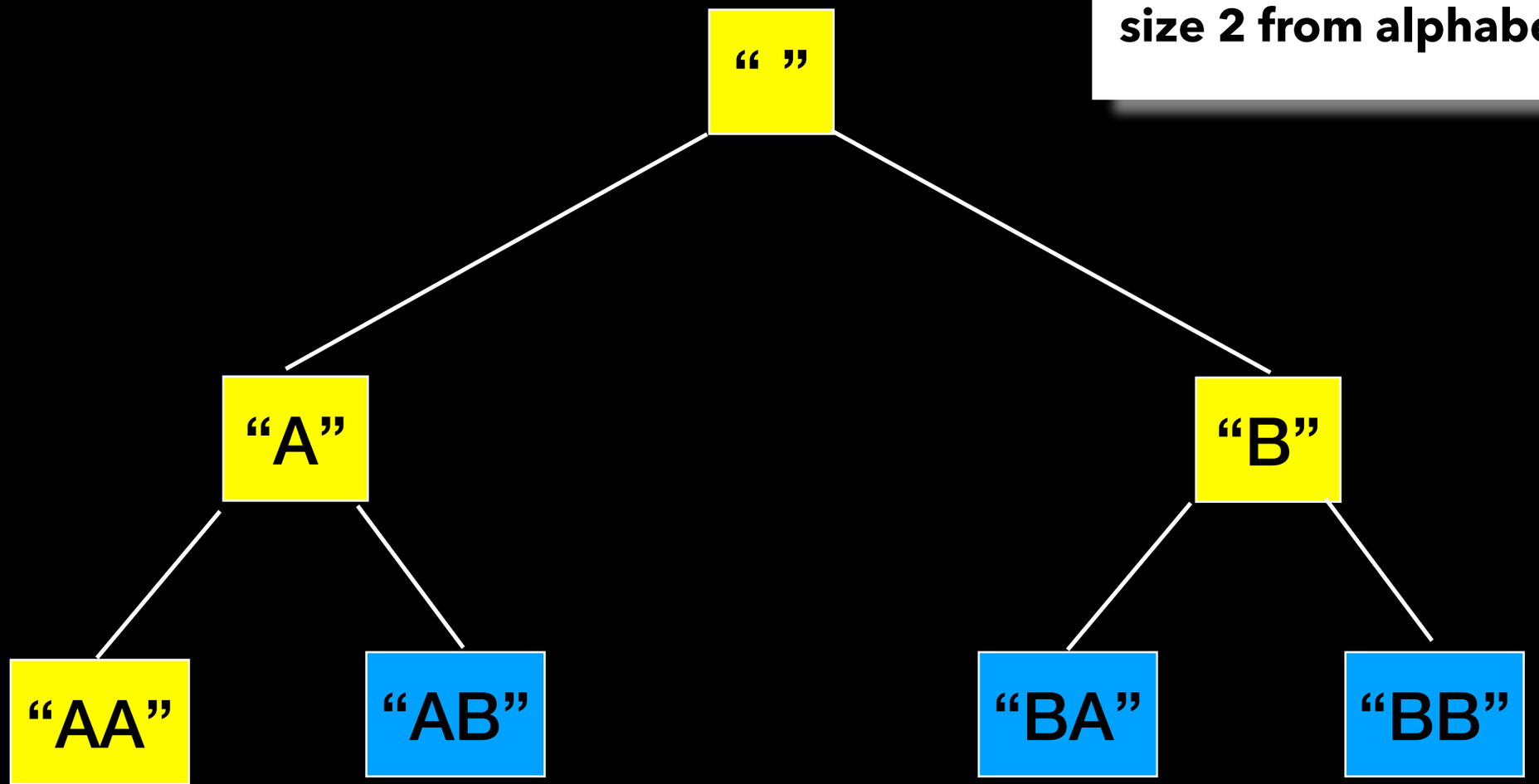
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**



{ “”, “A”, “B”, “AA”}

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

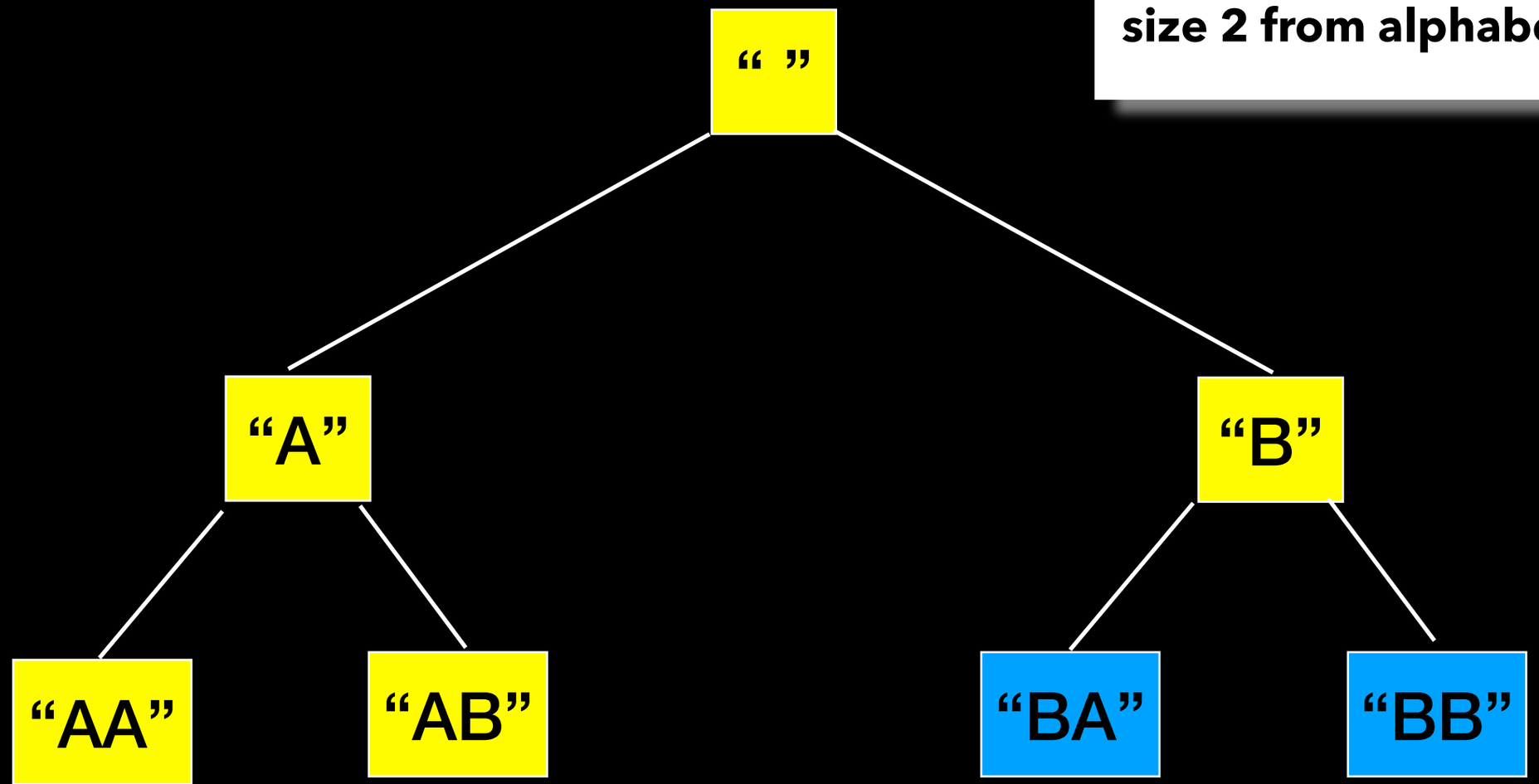



“AB“

“BA“ “BB“

{ “”, “A”, “B”, “AA”, “AB”}

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

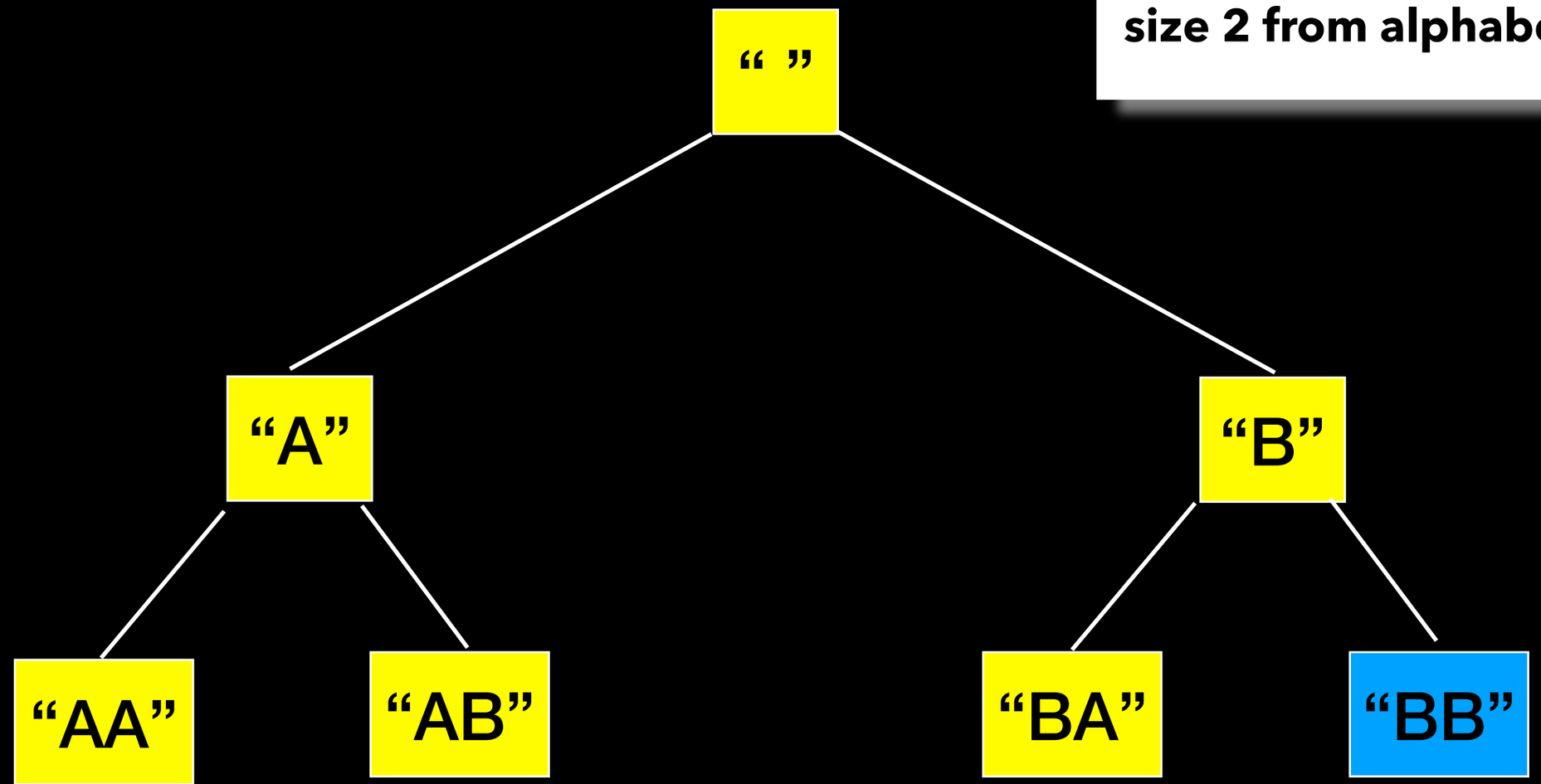



“AB“

“BA“ “BB“

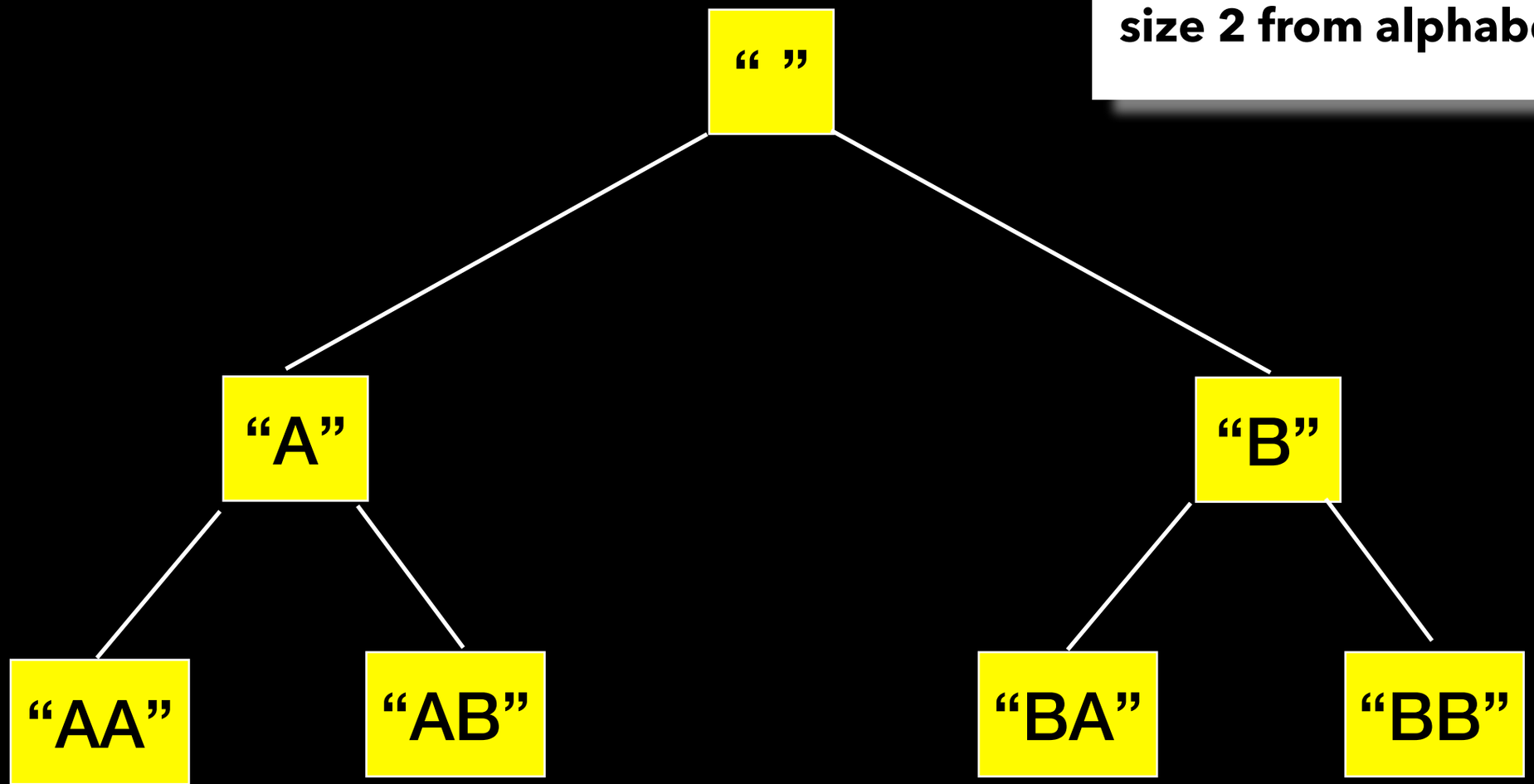
{ “”, “A”, “B”, “AA”, “AB”, “BA”}

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




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

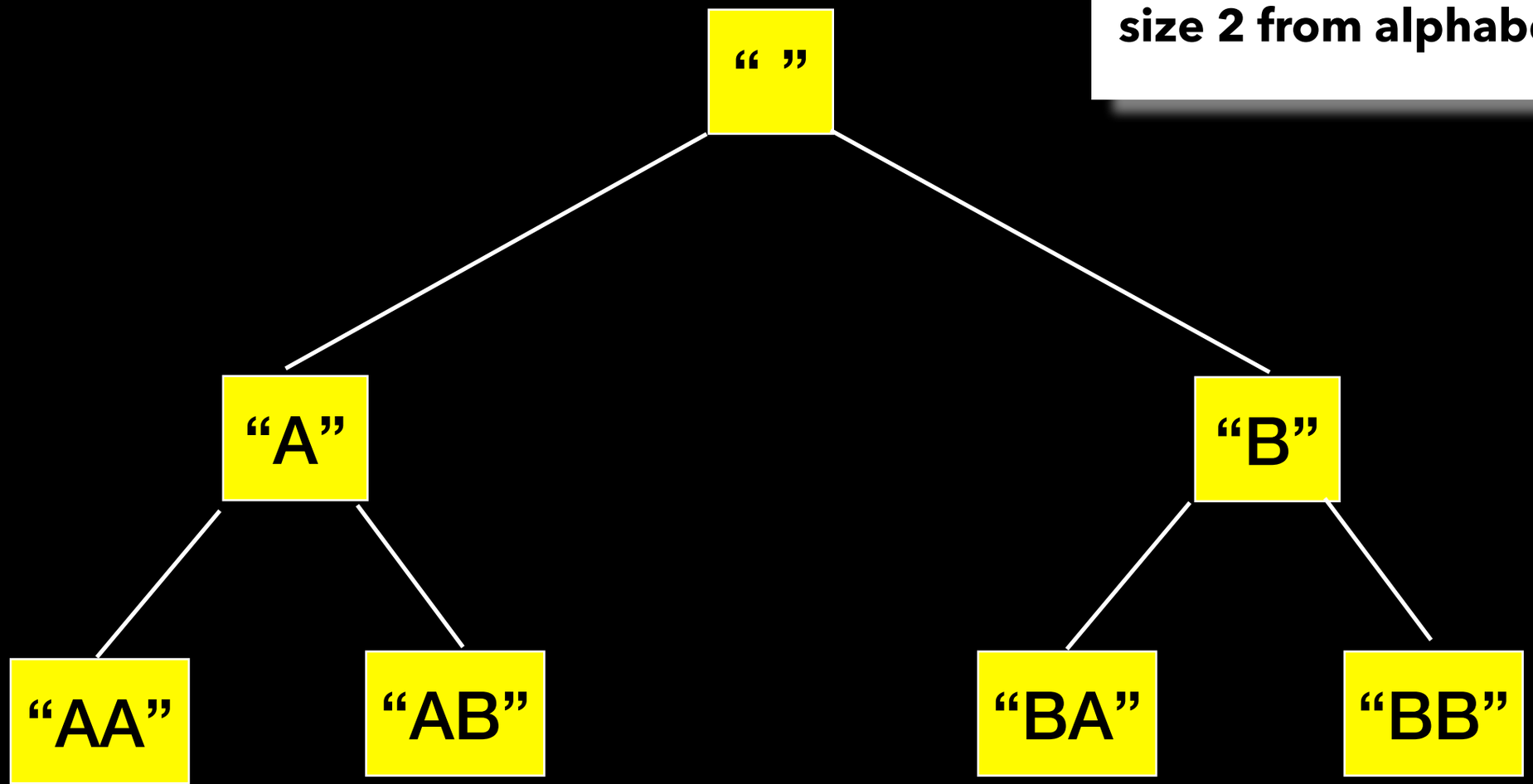
**Generate all substrings of size 2 from alphabet {‘A’, ‘B’}**




“BB“

{ “”, “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

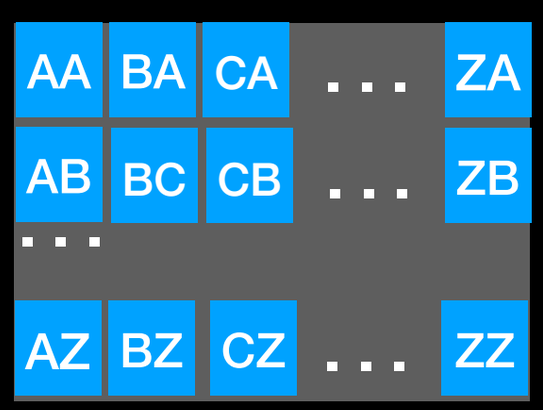
Finding all substrings (with repetition) of size **up to n**

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

The empty string= 1= $26^0$ ””

All strings of size 1 = $26^1$


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

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;
}
```

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

**T(n) = ?**

**O(?)**

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;
}
```

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

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;
}
```

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

Will stop when all strings have been removed from queue

Removes 1 string from the queue

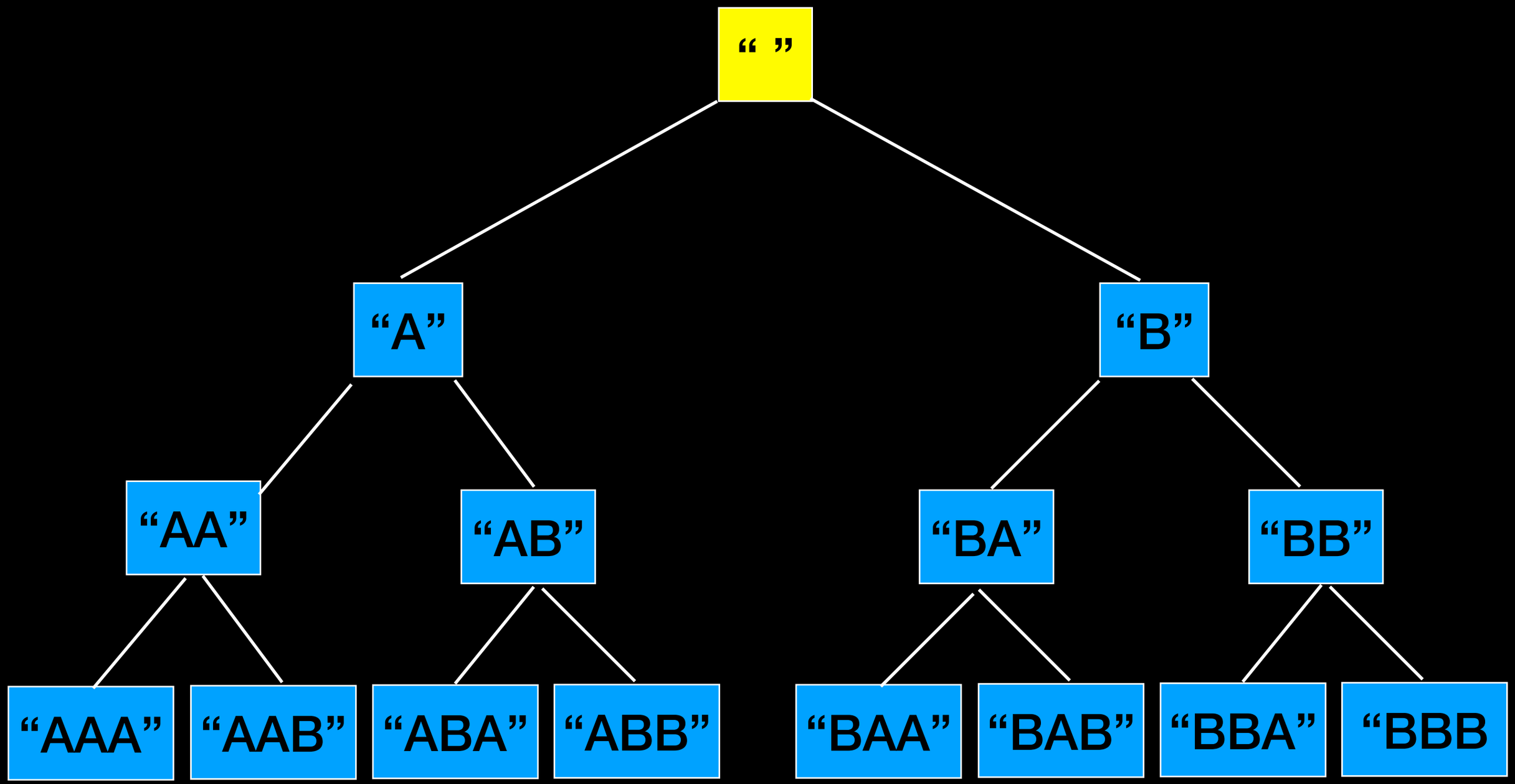
Adds 26 strings to the queue

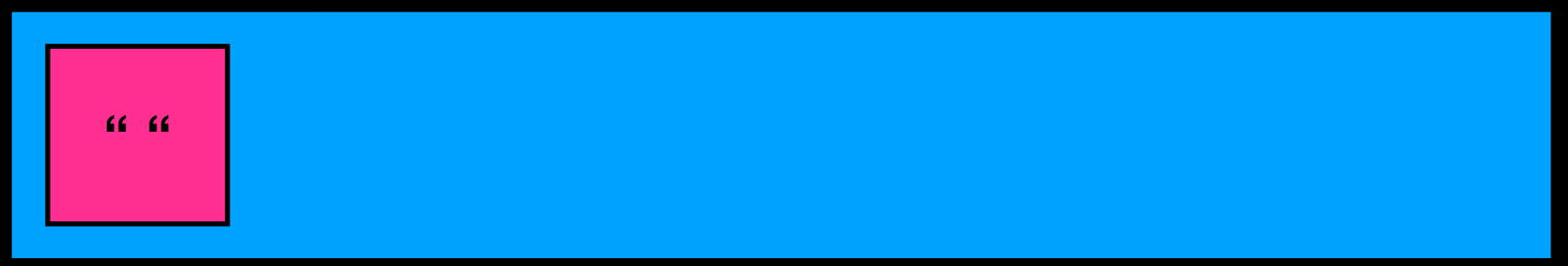
```
findAllSubstrings(int n)
{
    put empty string on the queue

    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;
}
```

$O(26^n)$

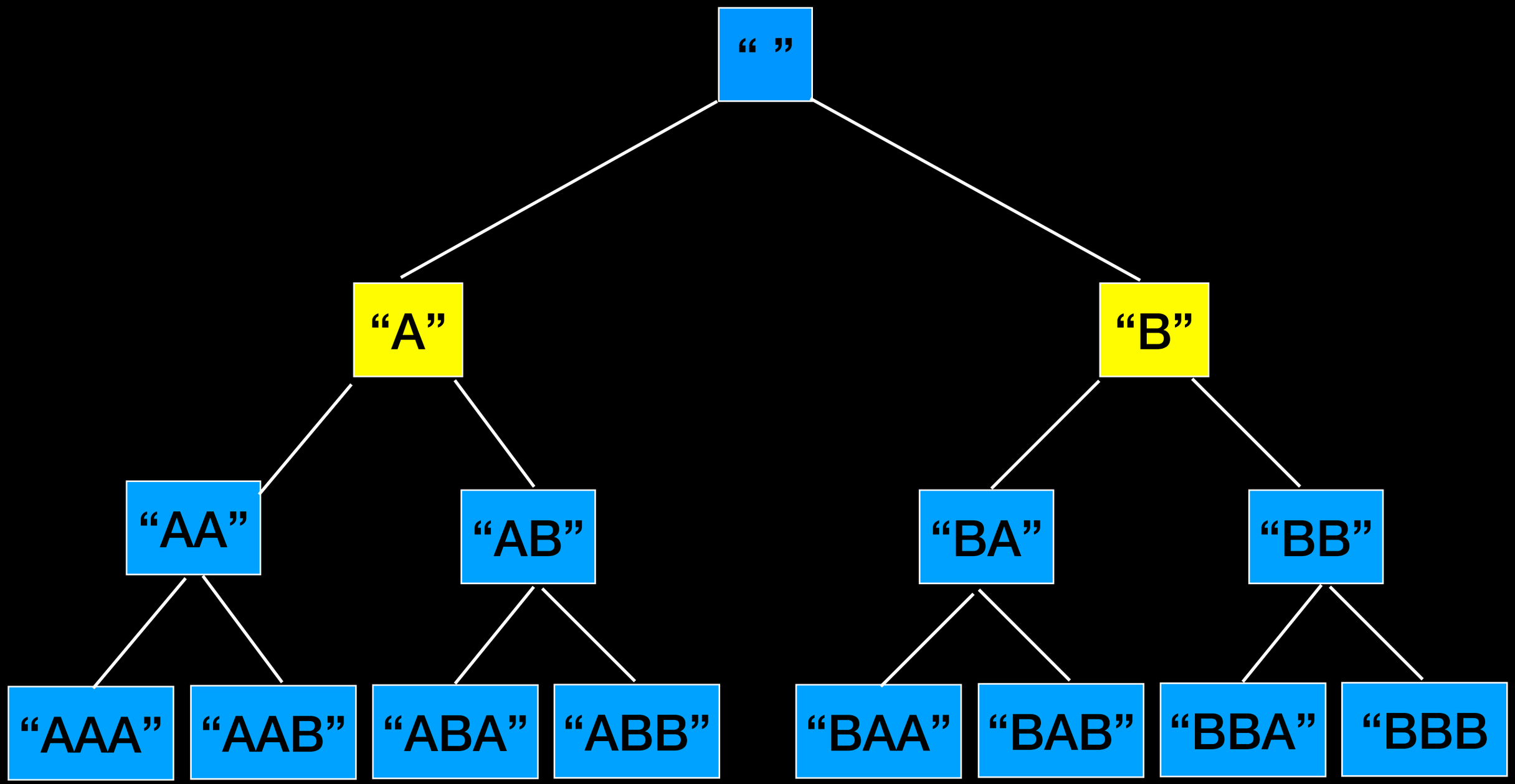
# Let n = 3, alphabet still {‘A’,’B’}

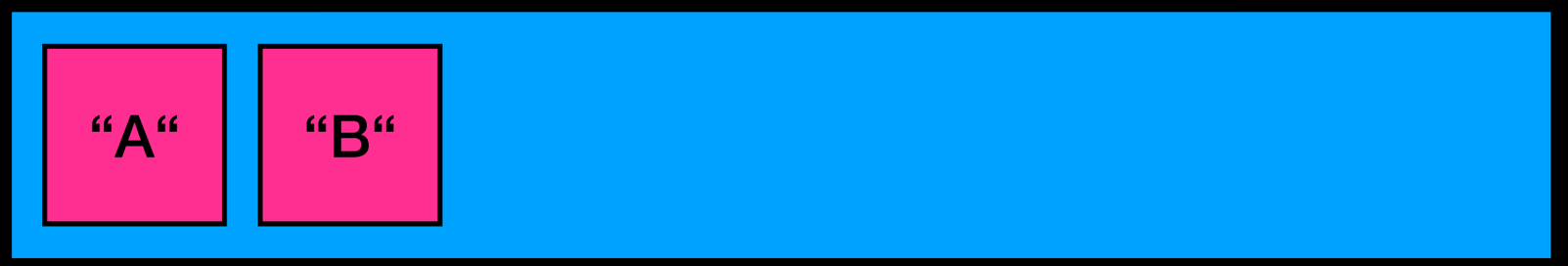



$2^0$

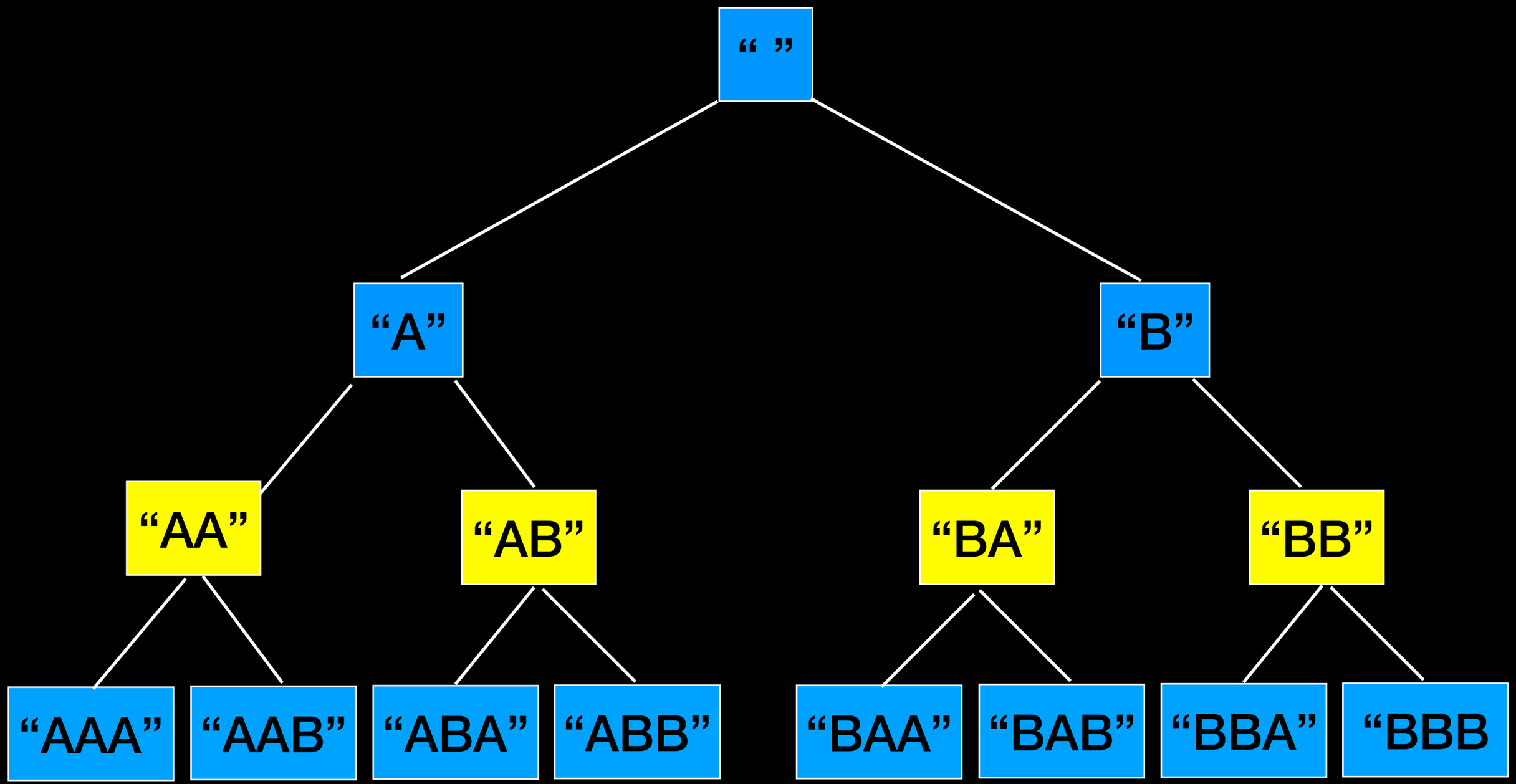
# Let n = 3, alphabet still {‘A’,’B’}

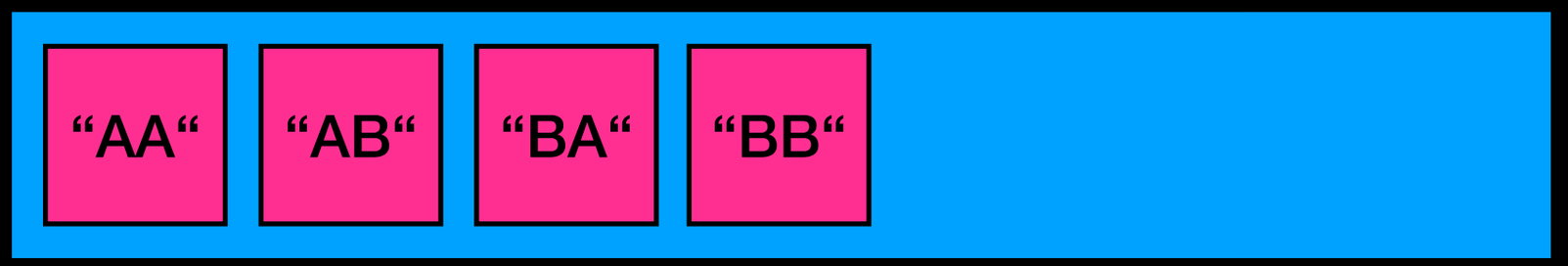



$2^1$

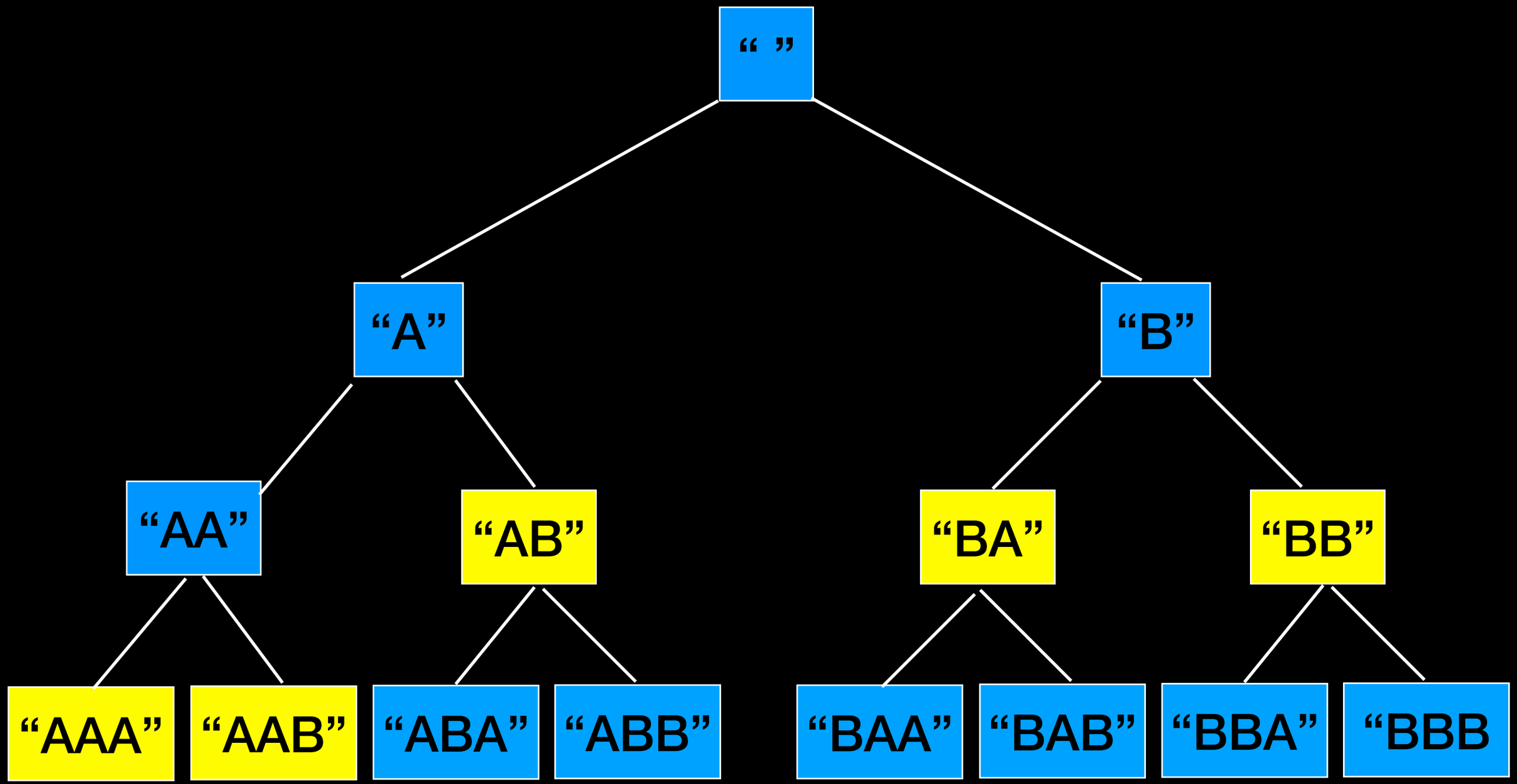
# Let n = 3, alphabet still {‘A’,’B’}




$2^2$

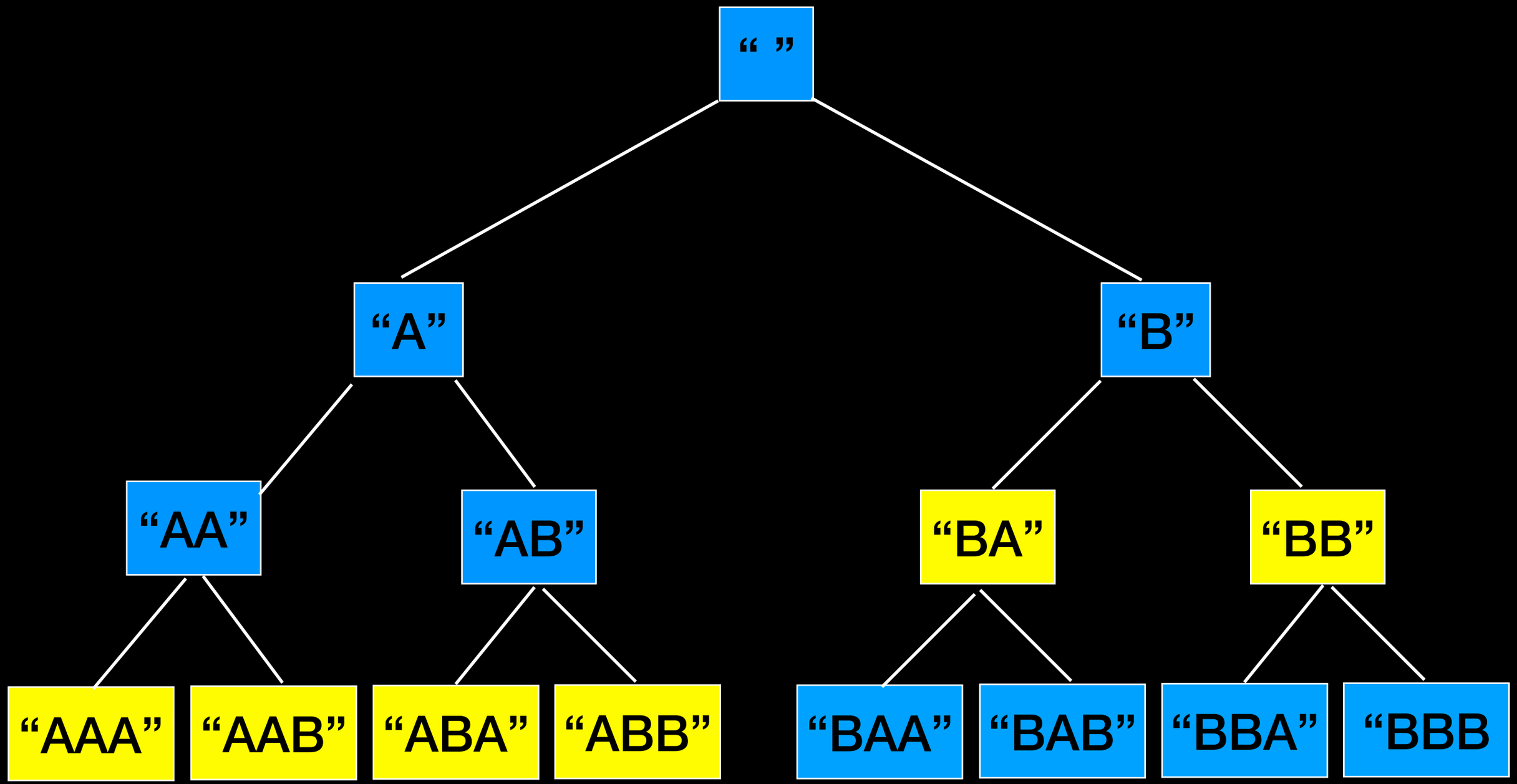
# Let n = 3, alphabet still {‘A’,’B’}




“AA“

“AB“ “BA“ “BB“ “AAA“ “AAB“

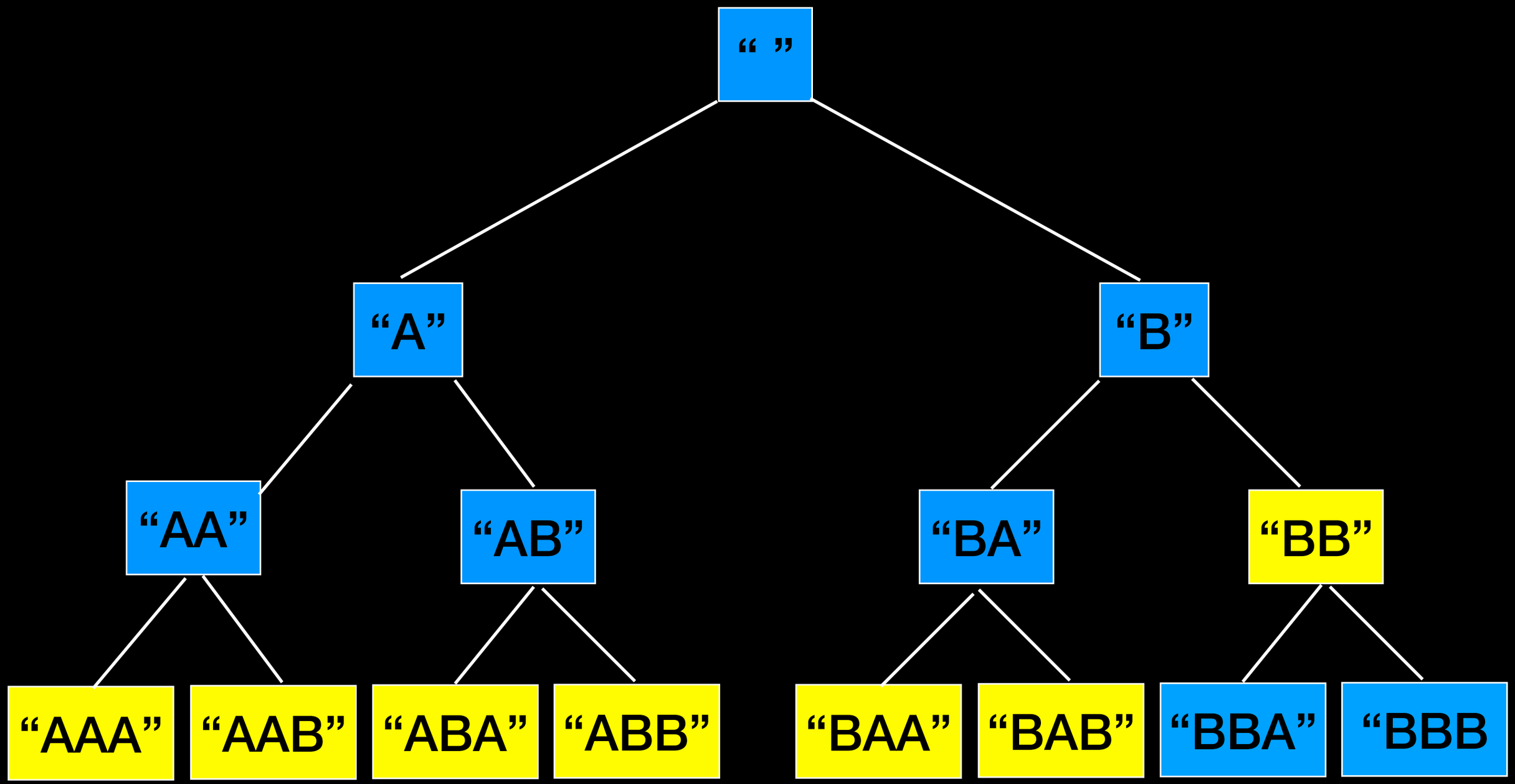
# Let n = 3, alphabet still {‘A’,’B’}

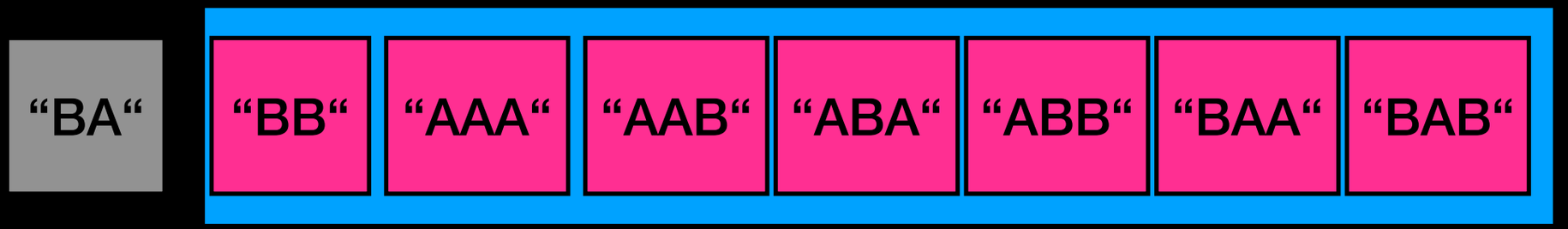



“AB“

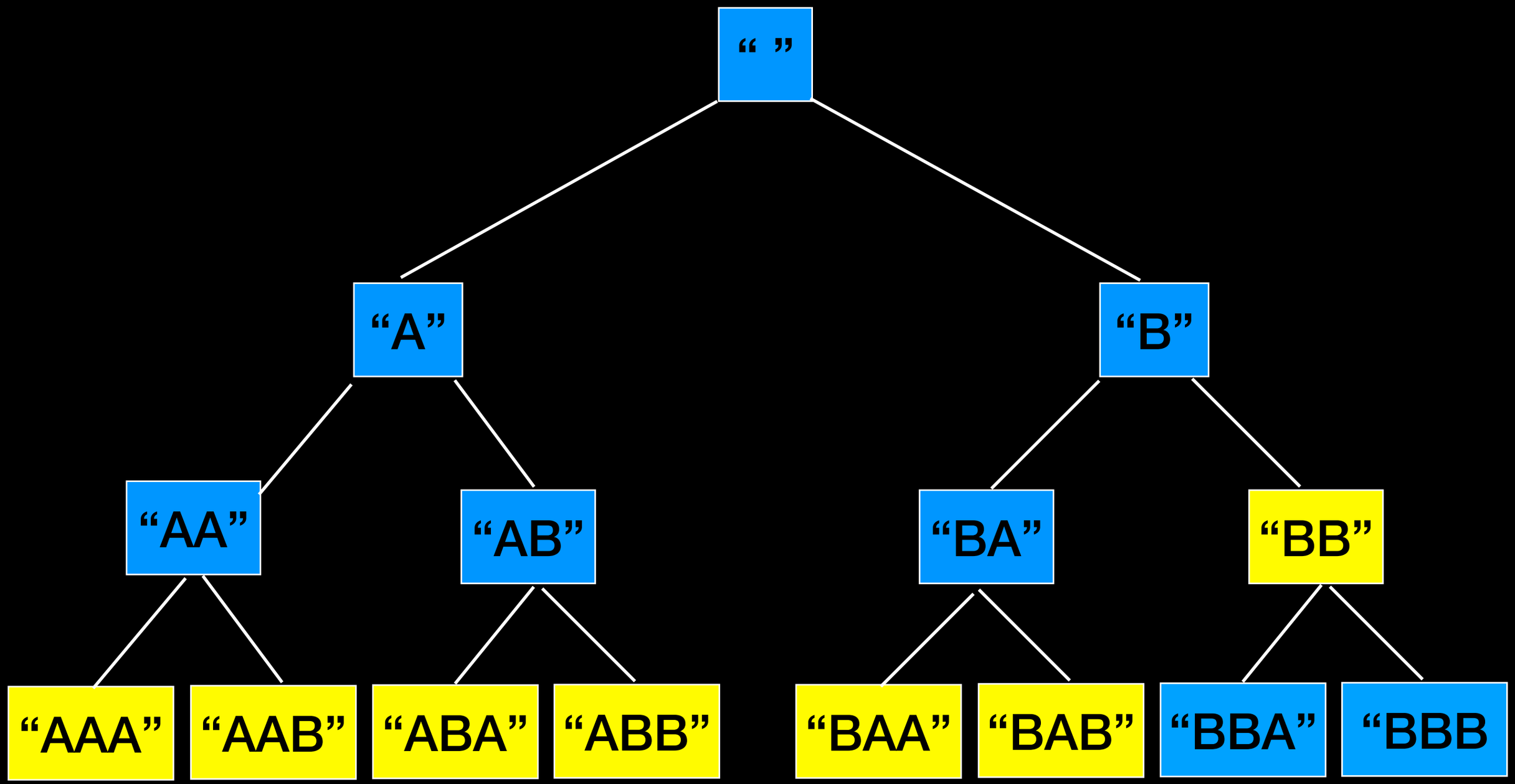
“BA“ “BB“ “AAA“ “AAB“ “ABA“ “ABB“

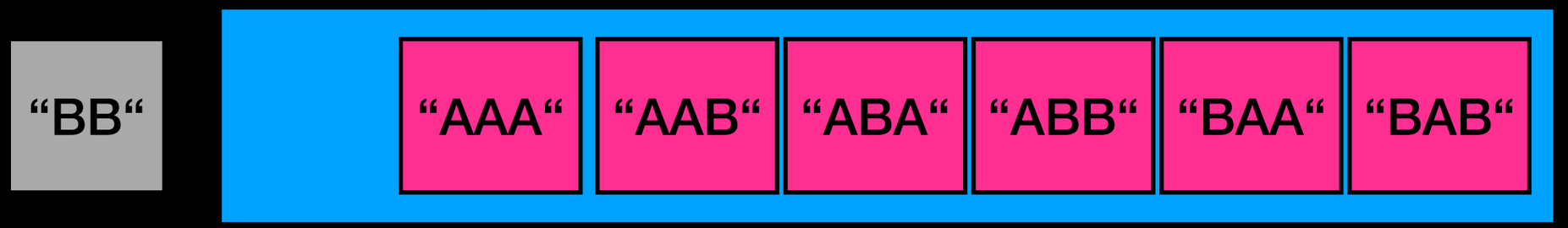
# Let n = 3, alphabet still {‘A’,’B’}



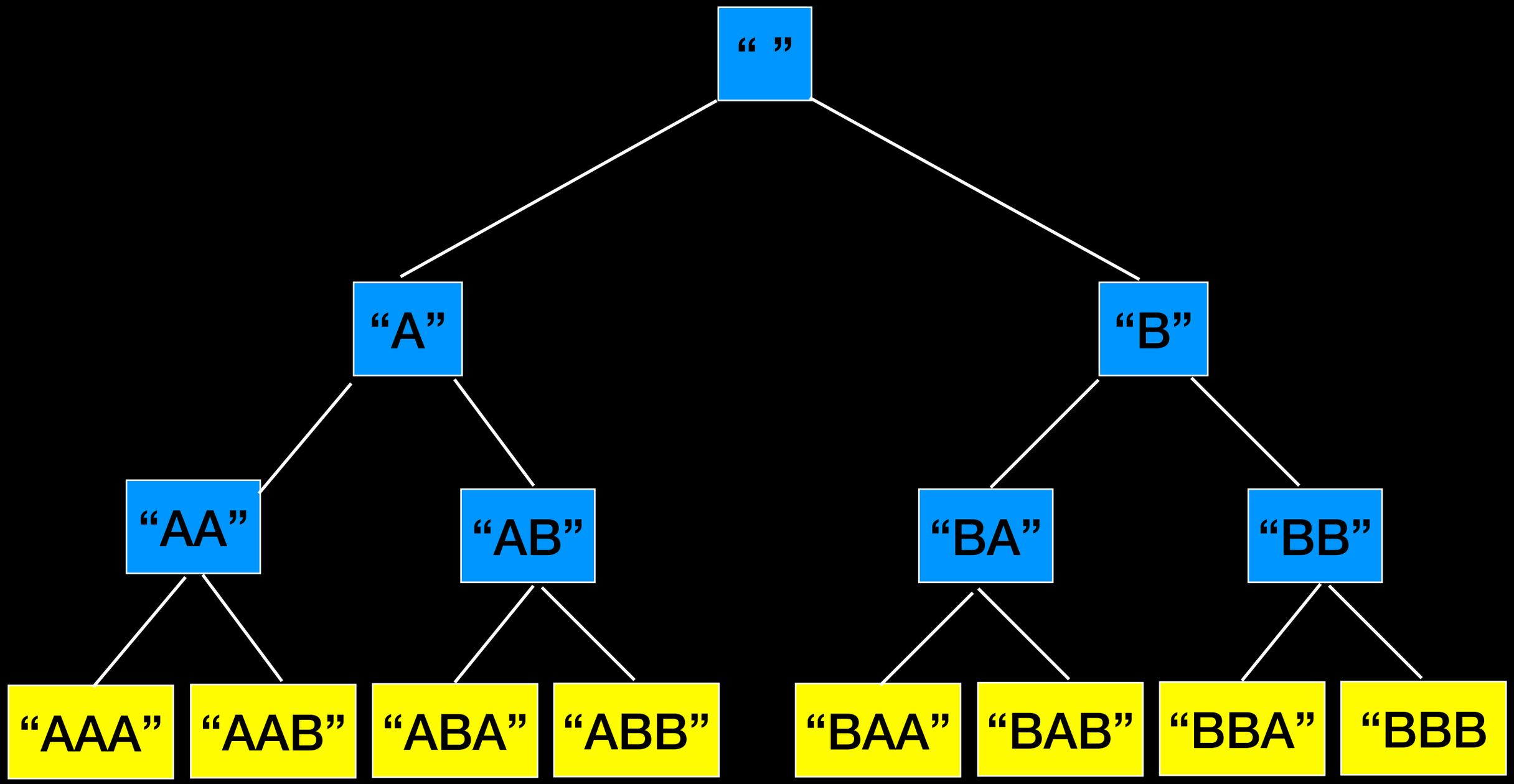



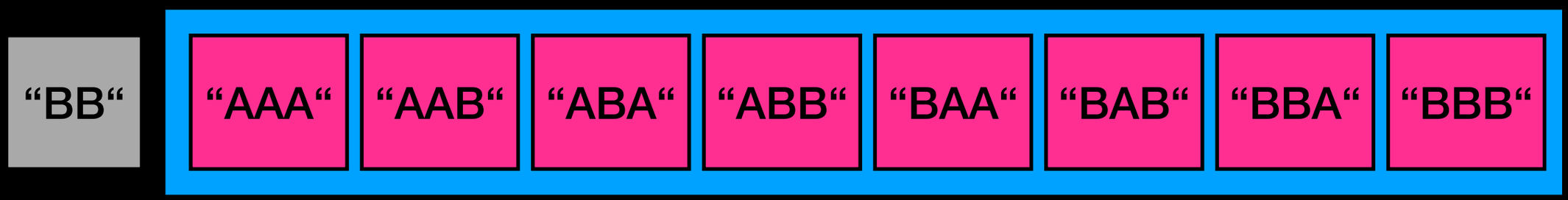
# Let n = 3, alphabet still {‘A’,’B’}



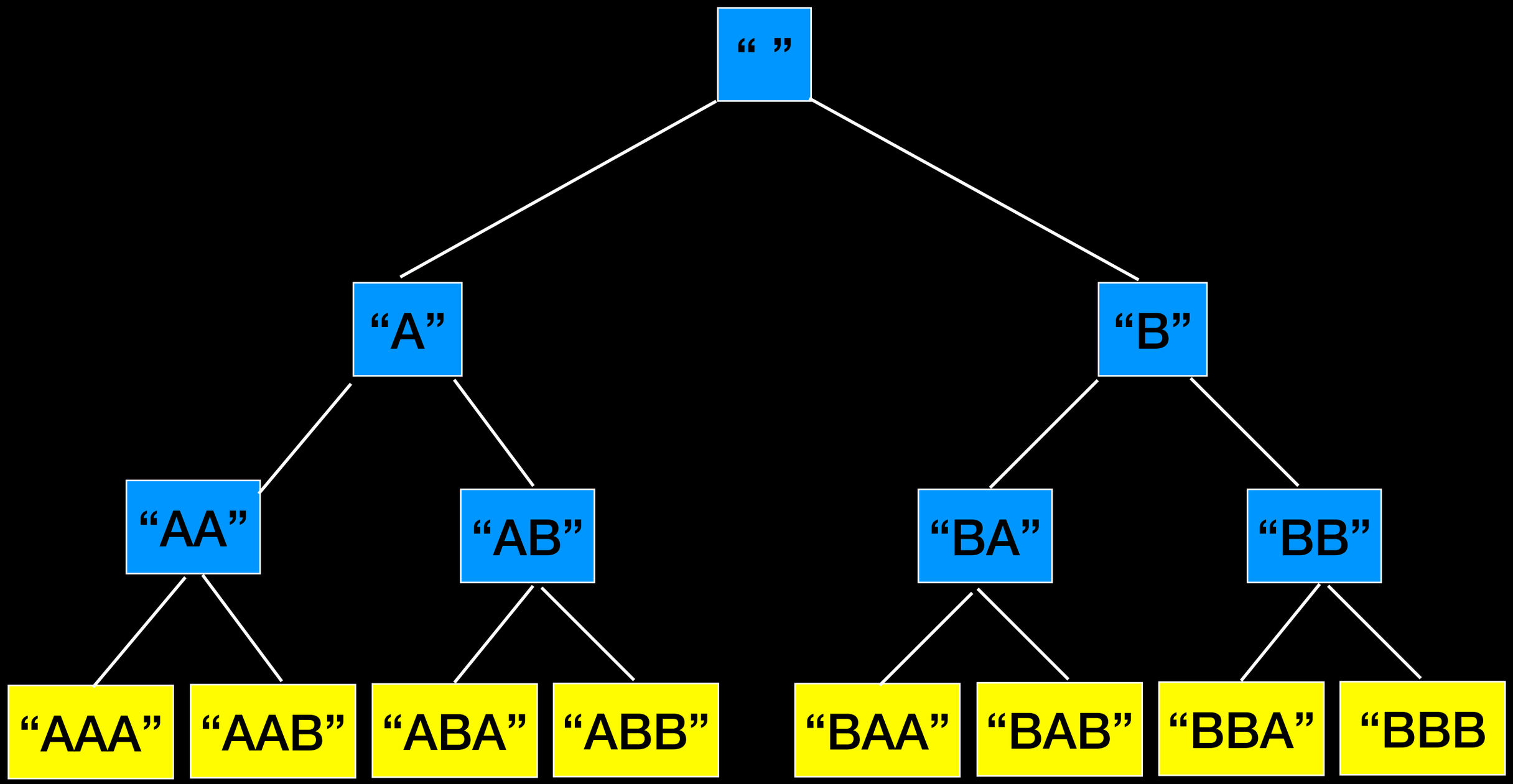



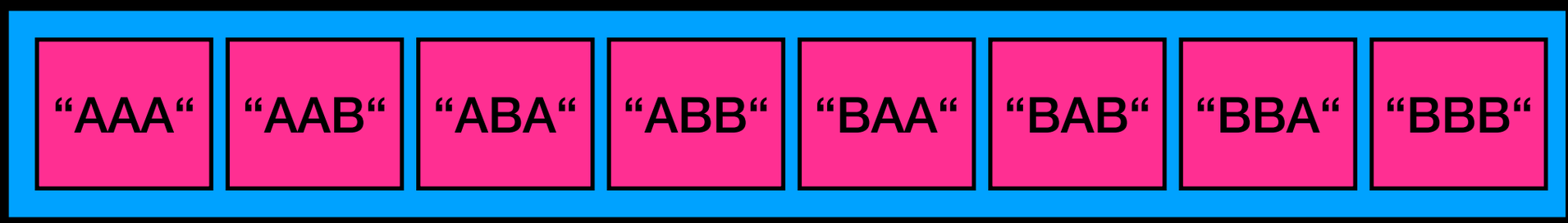
# Let n = 3, alphabet still {‘A’,’B’}



# 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

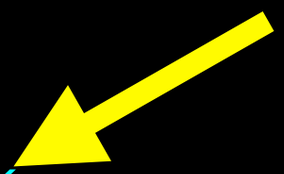
Running this algorithm for n = 7 (≈ 193GB) is the maximum that can be handled by a standard personal computer

For n = 8 ≈ 5TB

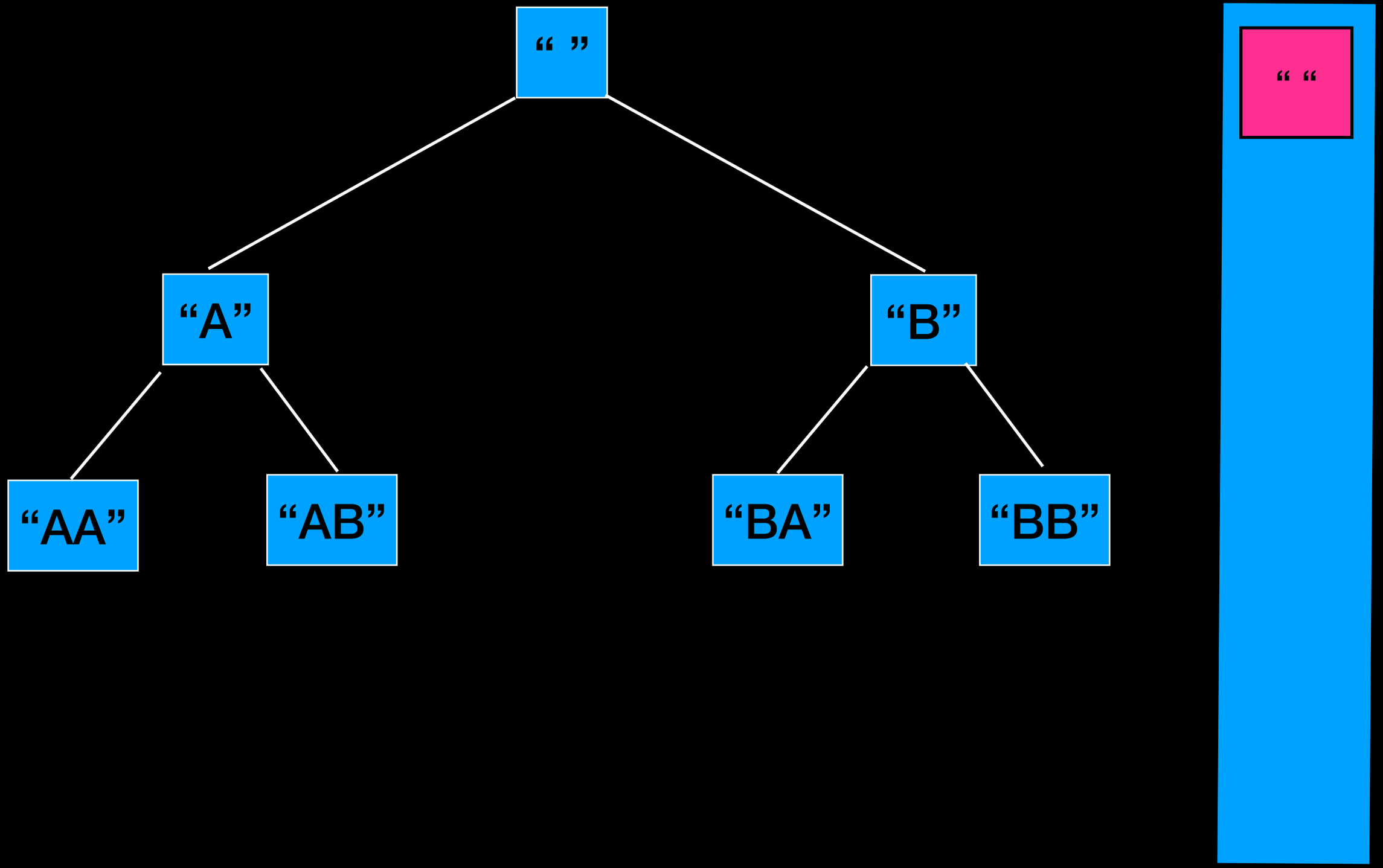
# What if we use a stack?

```
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”

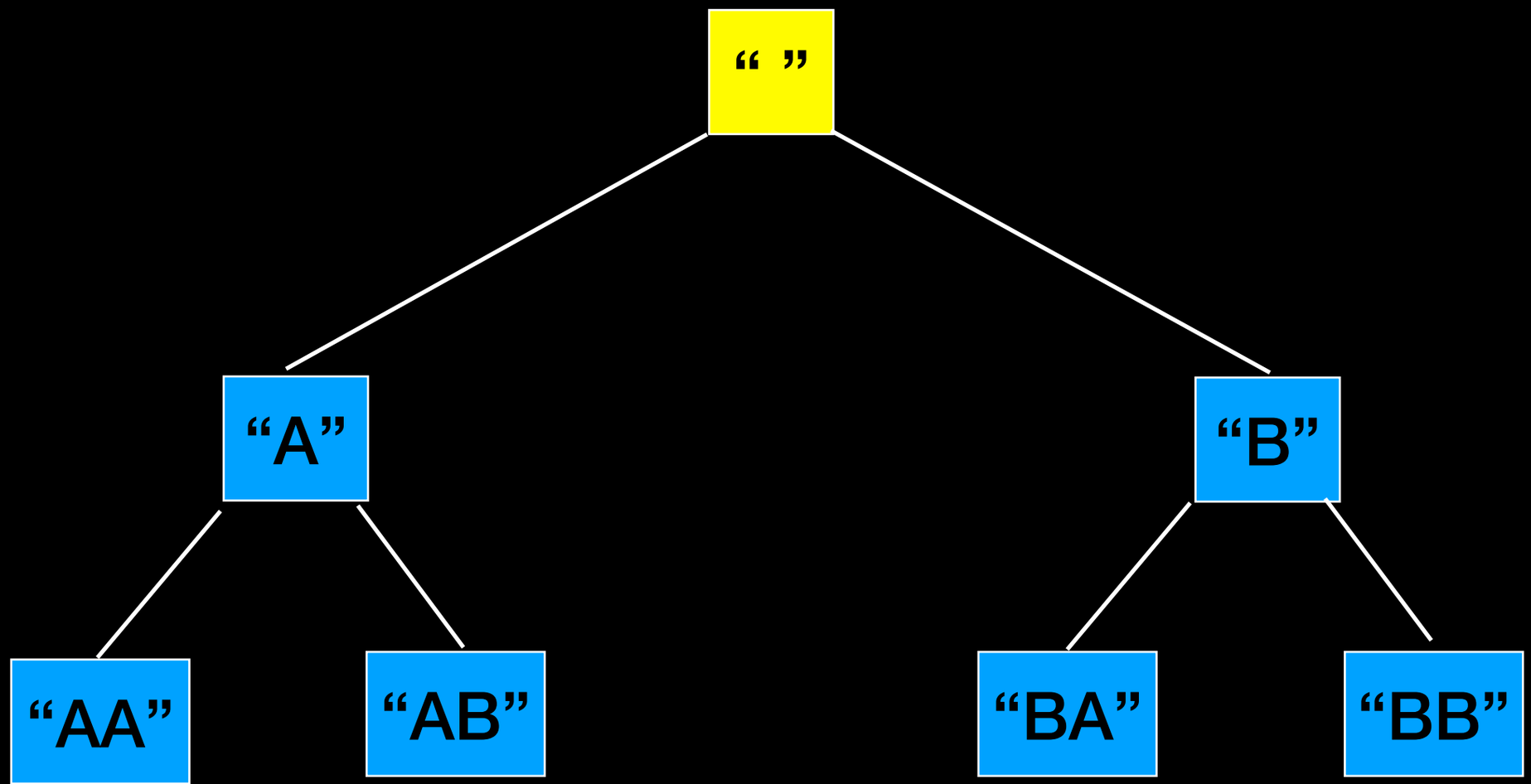
“AB”

“BA”

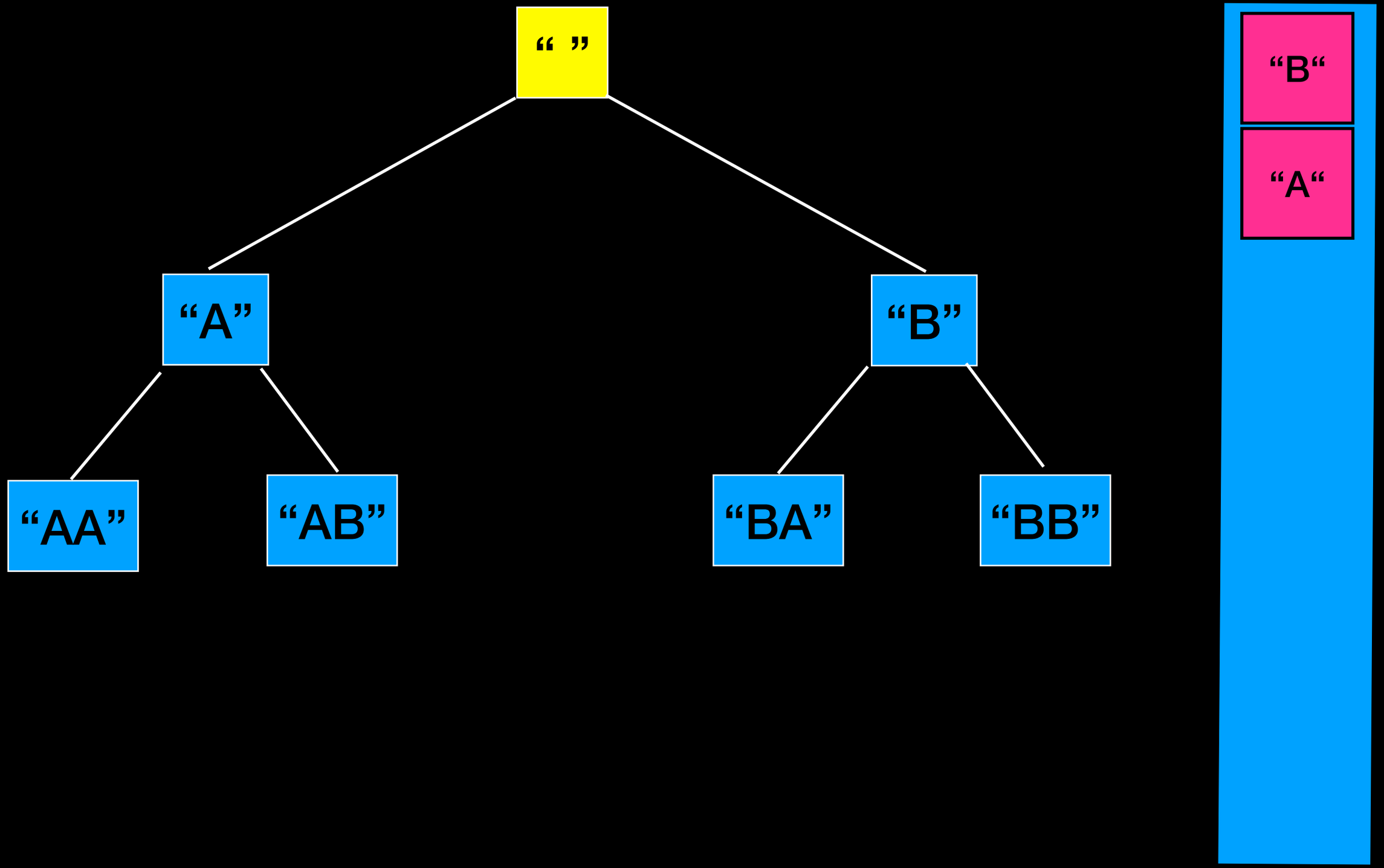
“BB”

{ “” }

“ “ “A“ “B“



{ “” }


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

{ “” }

“B“

“A“

“ ”

“A”

“B”

“AA”

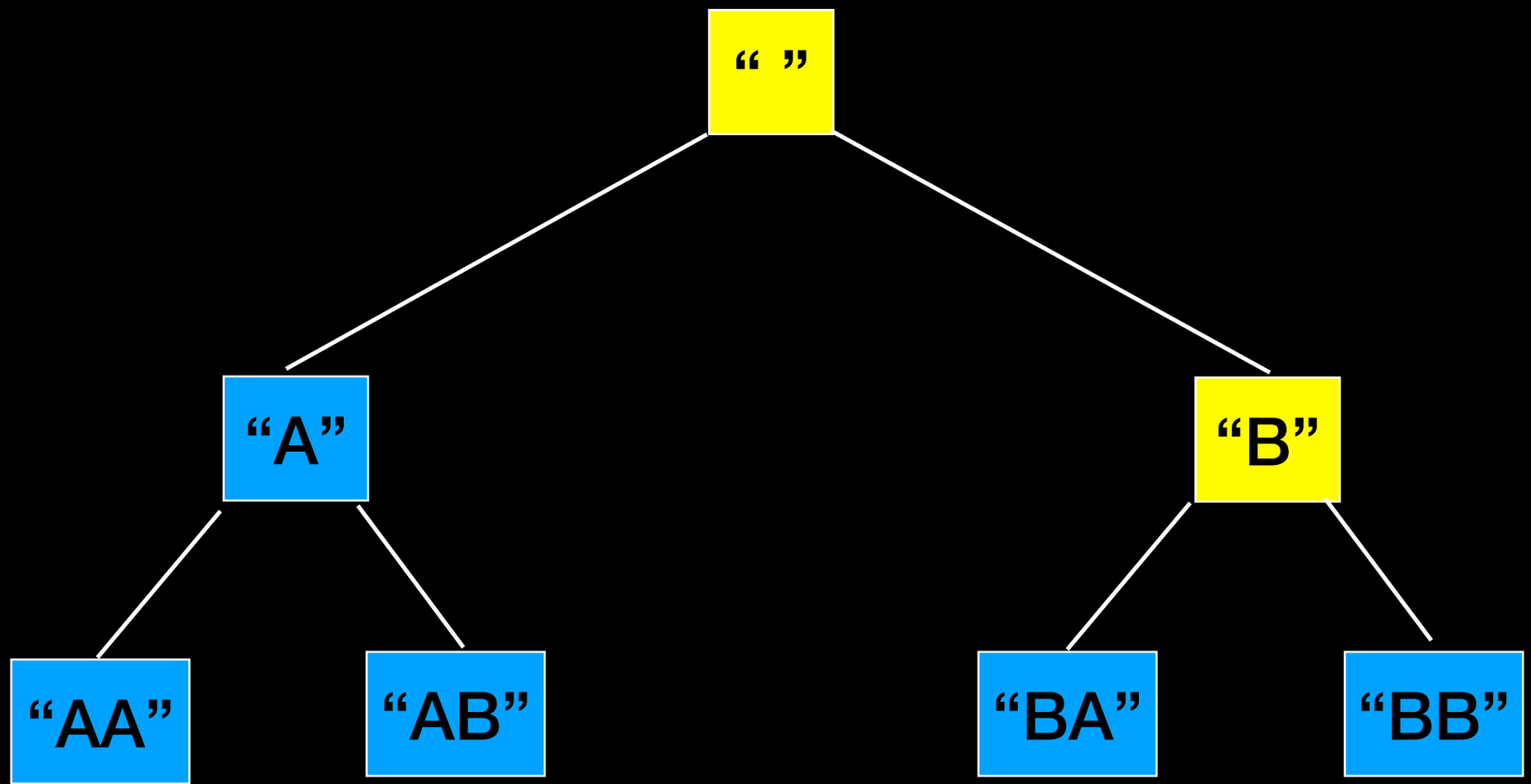
“AB”

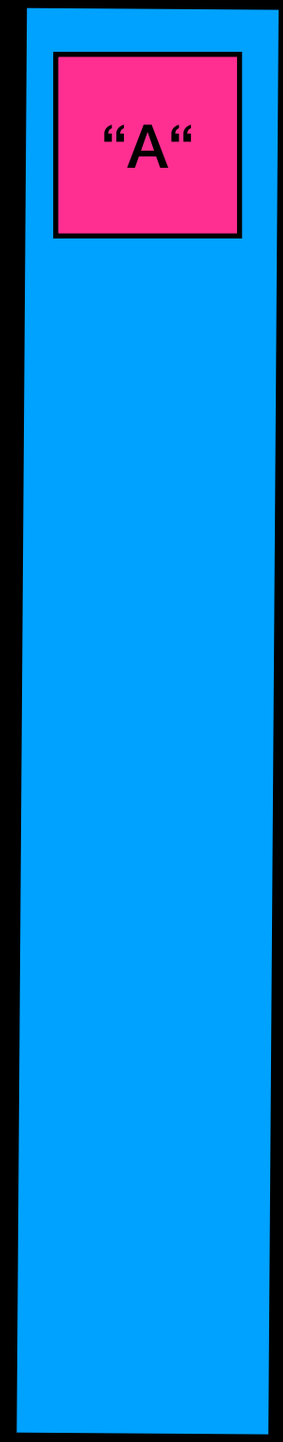
“BA”

“BB”

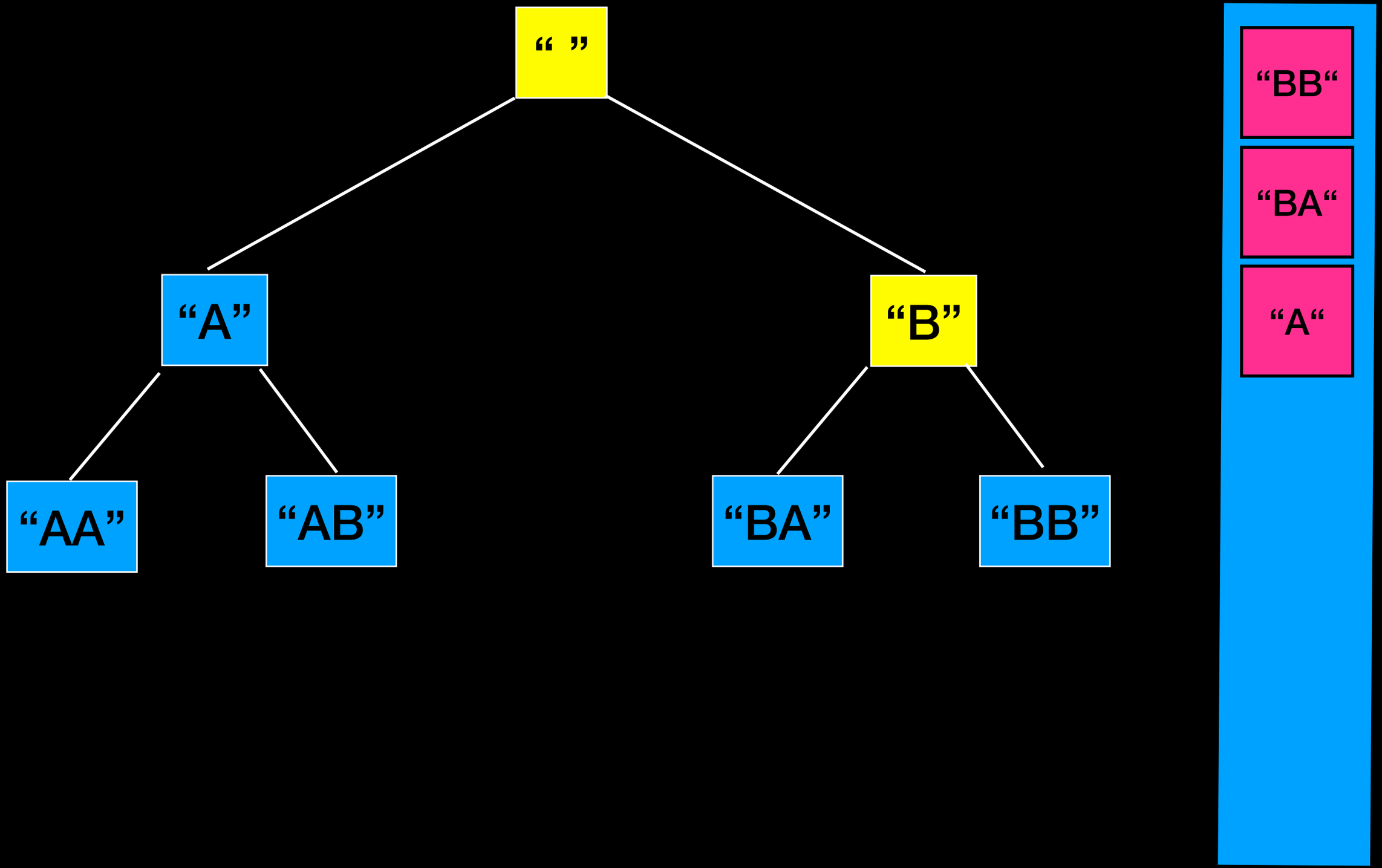
{ “”,“B”}

“B“ “BA“ “BB“



{ “”,“B”}

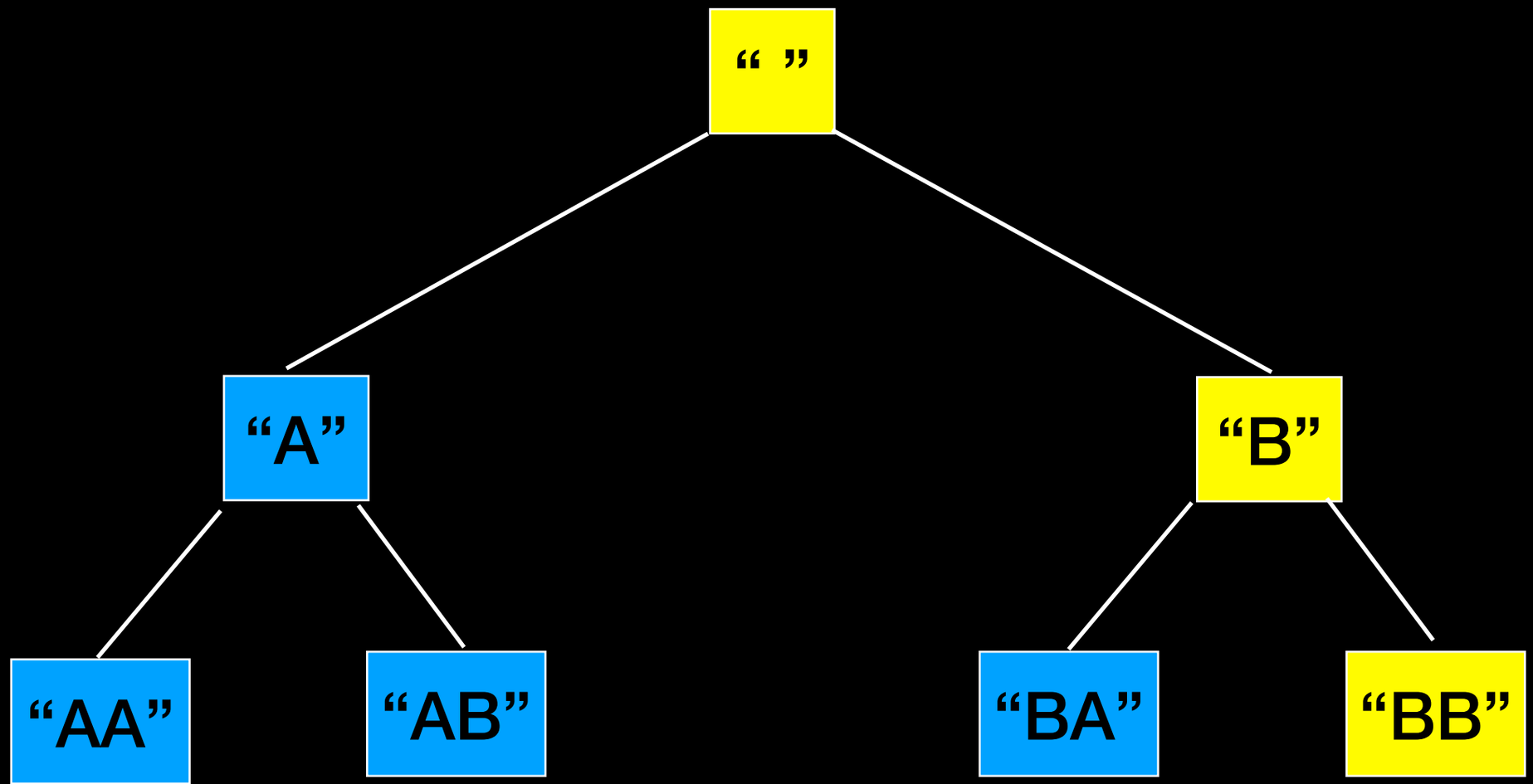


{ “”,“B”,“BB”}

“BB“

“BA“

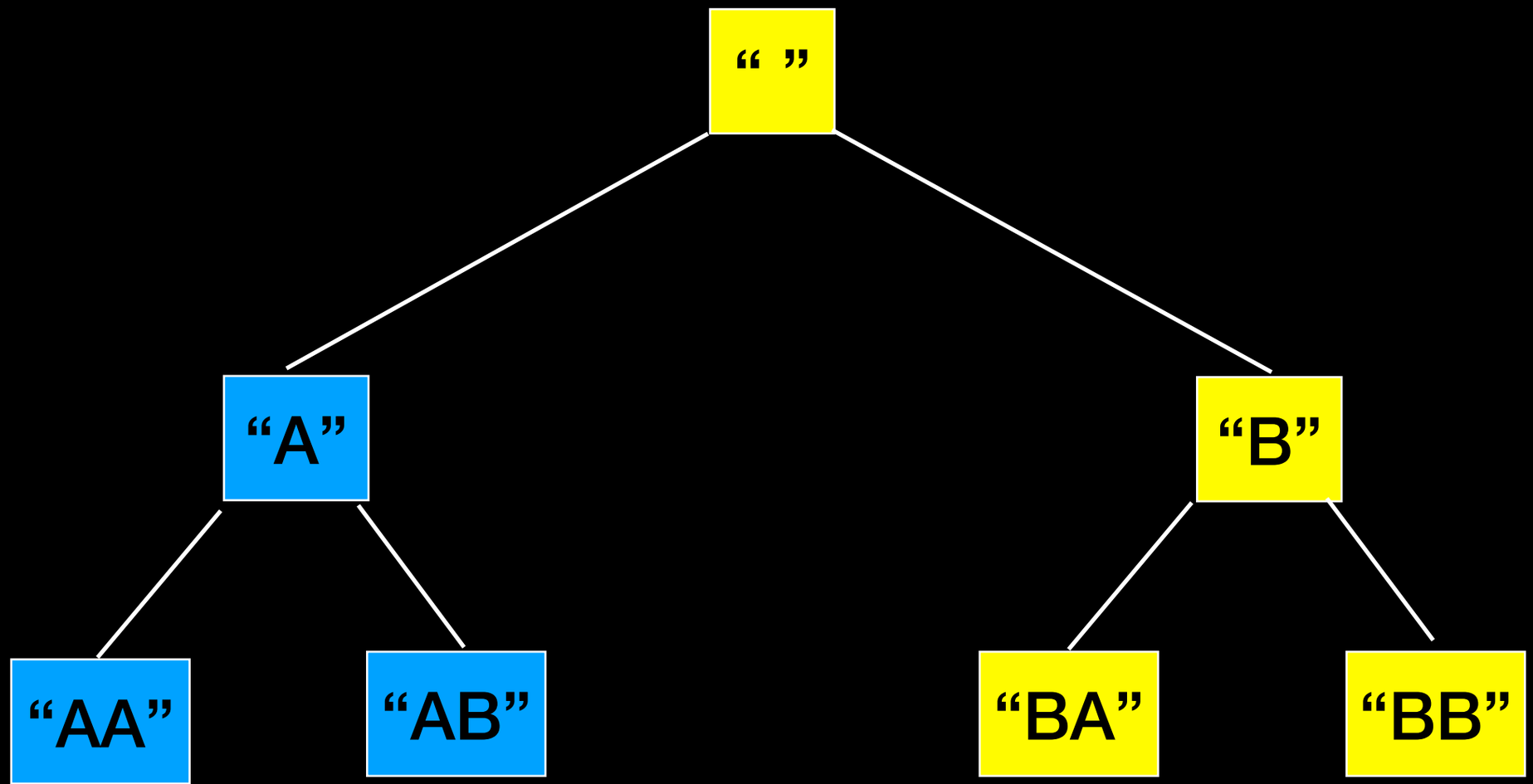
“A“



{ “”,“B”,“BB”,“BA”}

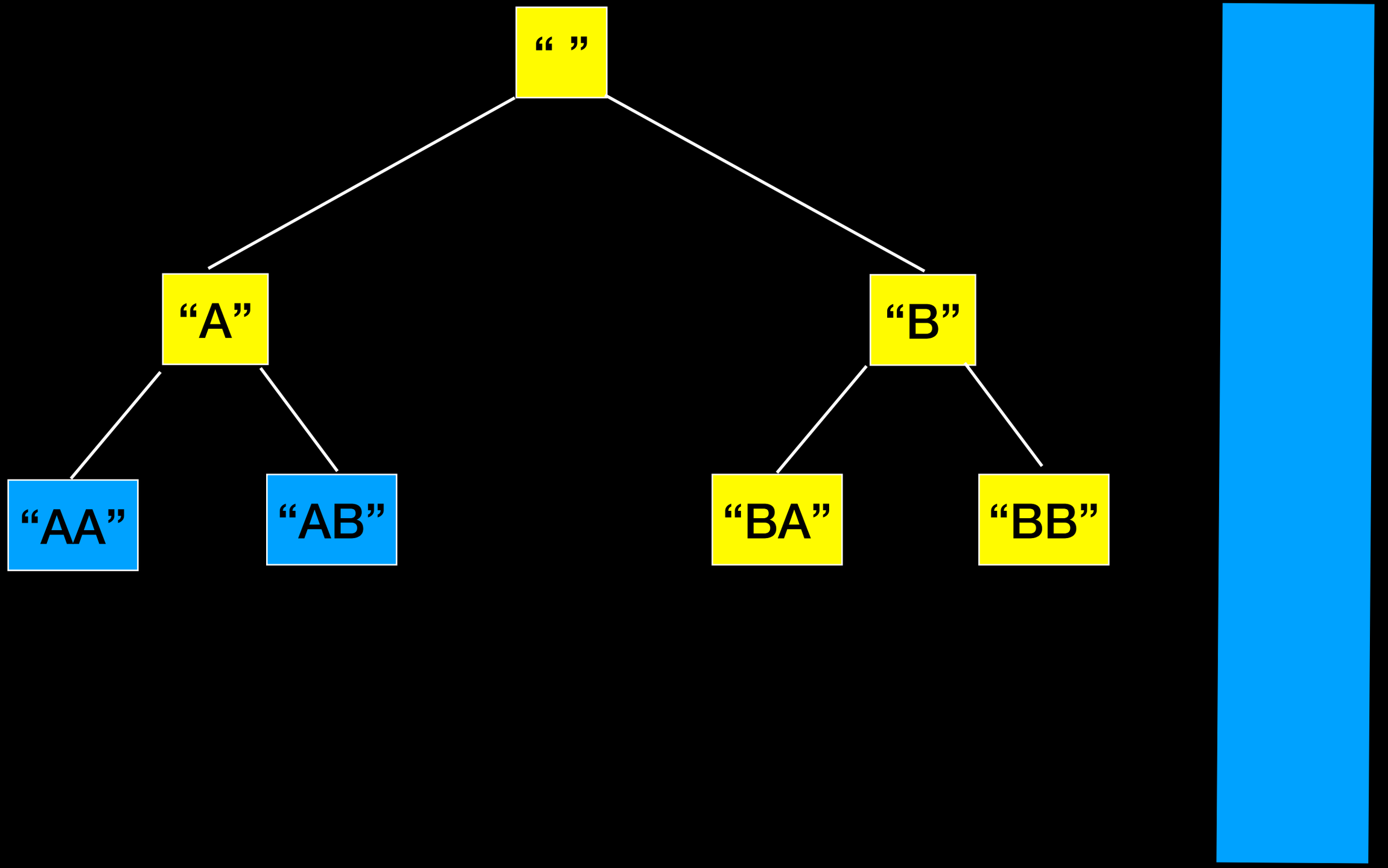
“BA“

“A“



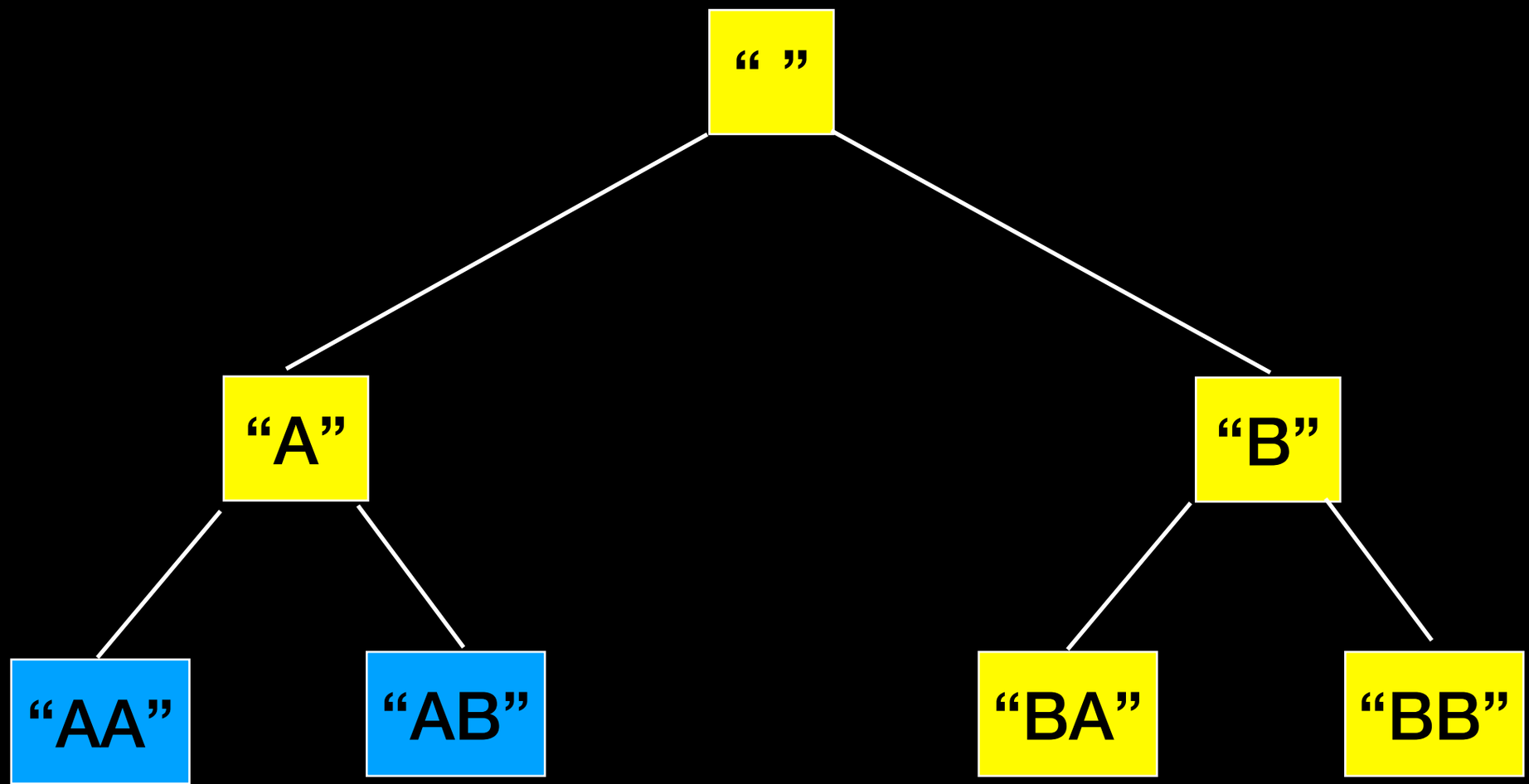
{ “”,“B”,“BB”,“BA”,“A”}

“A“

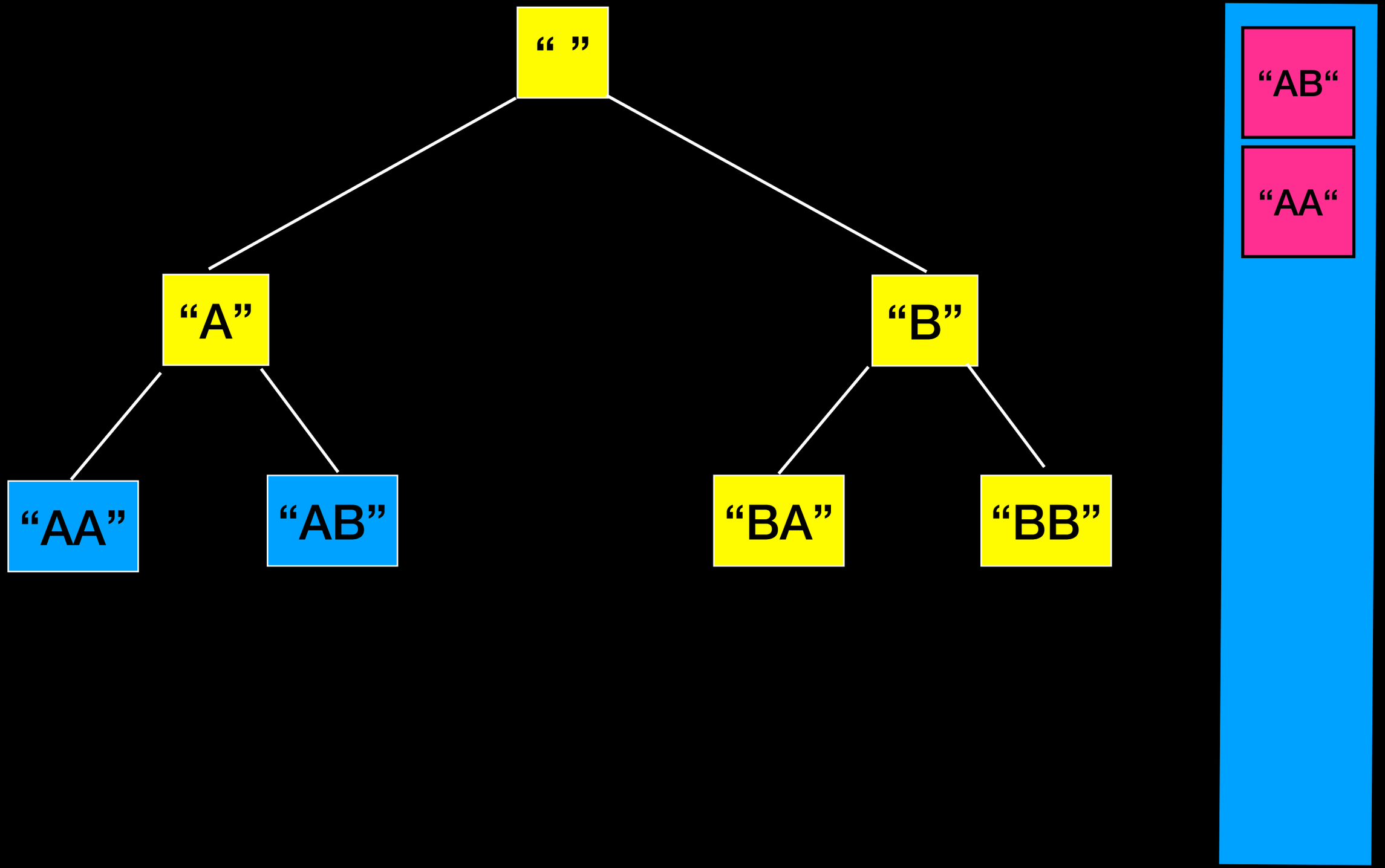


{ “”,“B”,“BB”,“BA”,“A”}

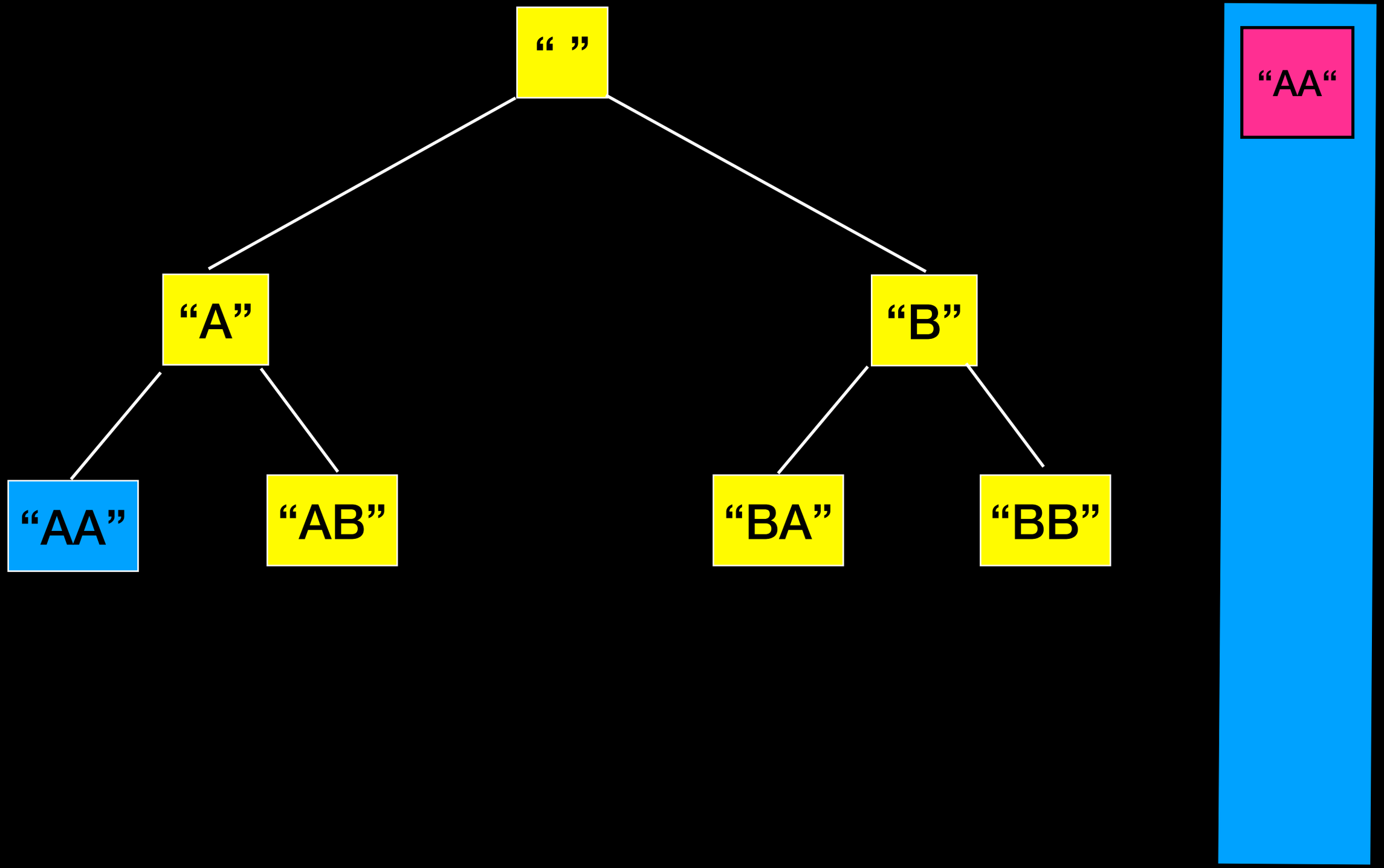
“A“ “AA“ “AB“



{ “”,“B”,“BB”,“BA”,“A”}

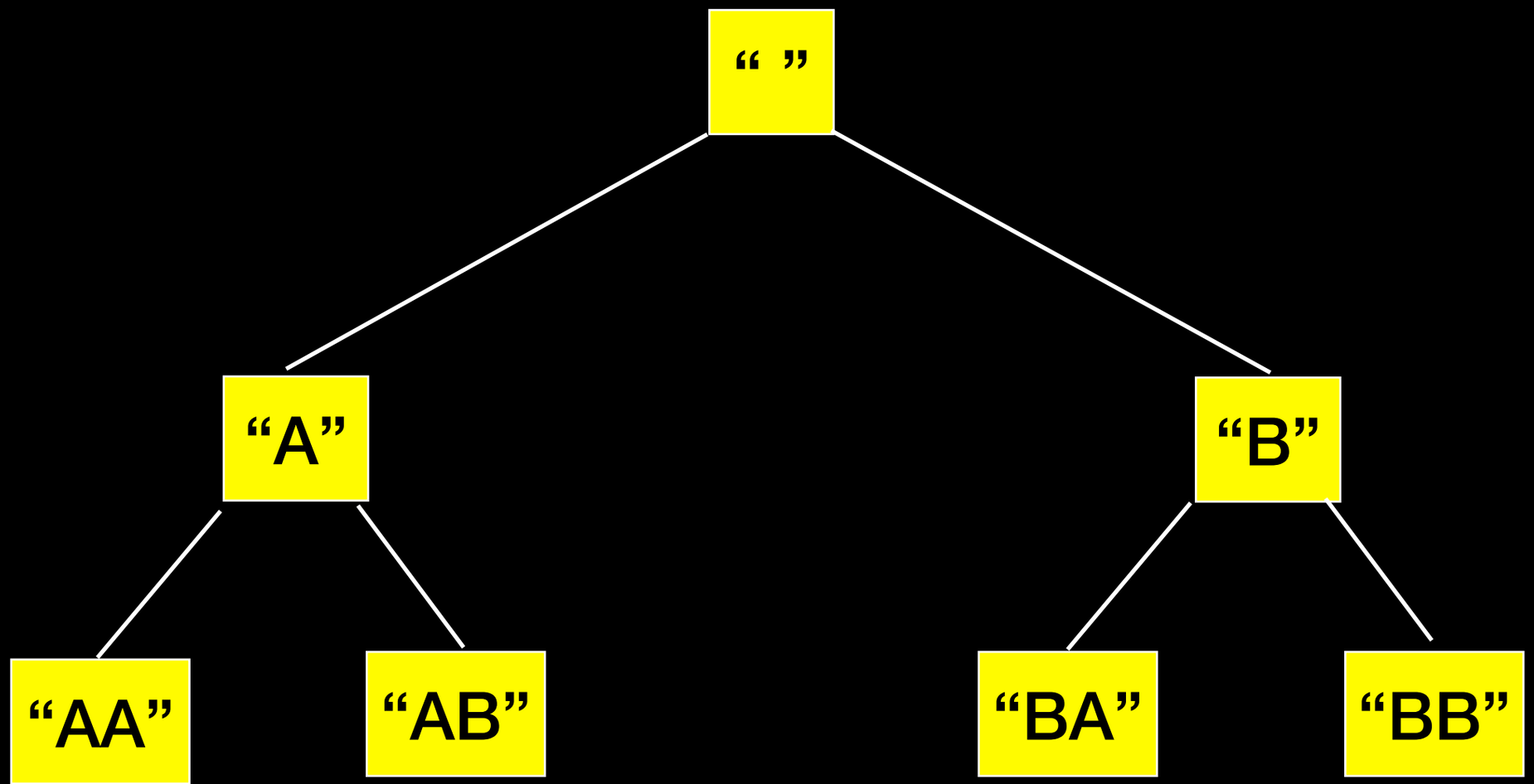


{ “”,“B”,“BB”,“BA”,“A”,“AB”}

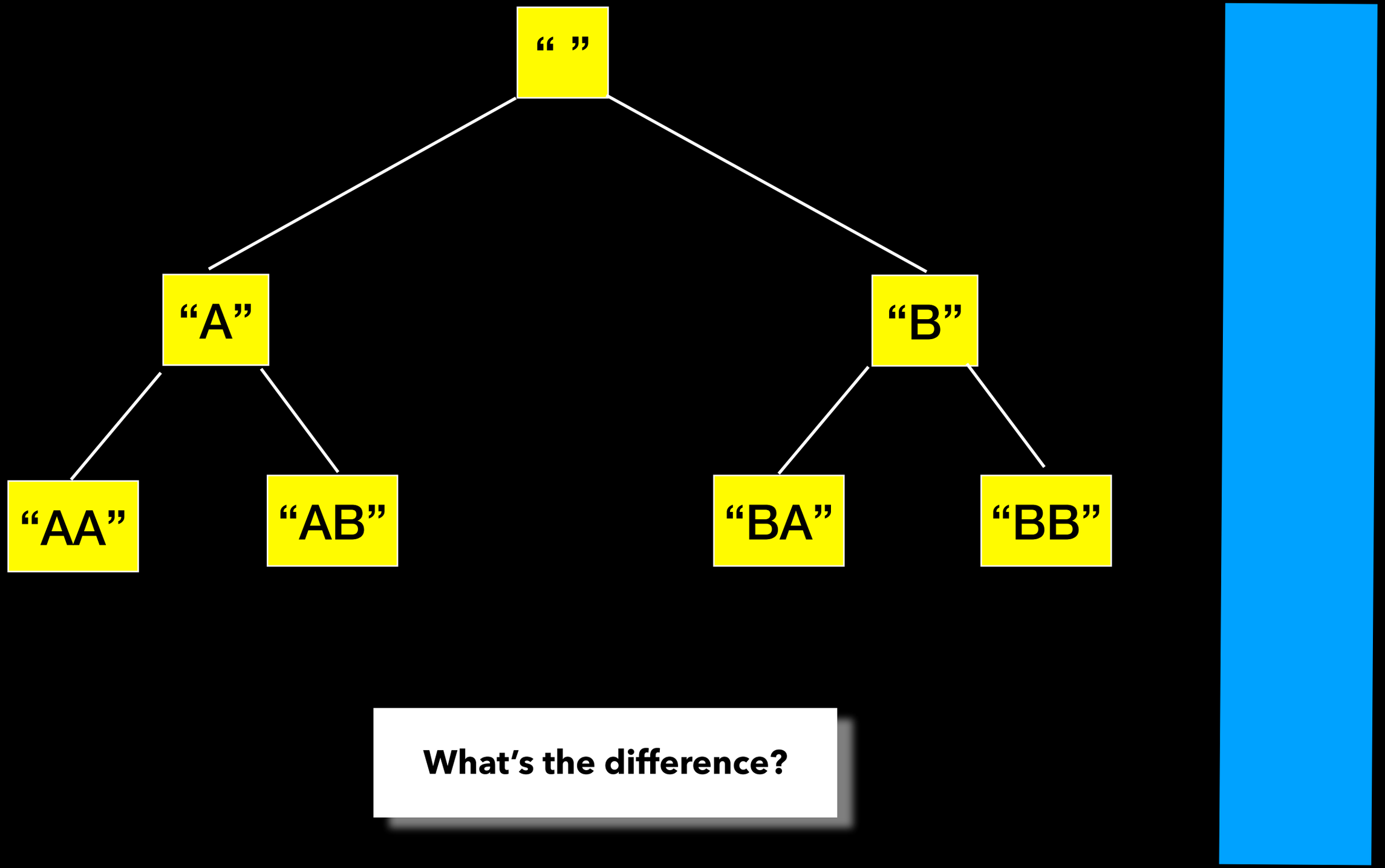


{ “”,“B”,“BB”,“BA”,“A”,“AB”,“AA”}

“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

# Queue ADT

```
#ifndef QUEUE_H_
#define QUEUE_H_

template<class T>
class Queue
{

public:
    Queue();
    void enqueue(const T& new_entry); // adds an element to back queue
    void dequeue(); // removes element from front of queue
    T front() const; // returns a copy of element at the front of queue
    int size() const; // returns the number of elements in the queue
    bool isEmpty() const; // returns true if no elements in queue, false otherwise

private:
        //implementation details here


};    //end Queue

#include "Queue.cpp"
#endif // QUEUE_H_ `
```

# Other ADTs

# Deque

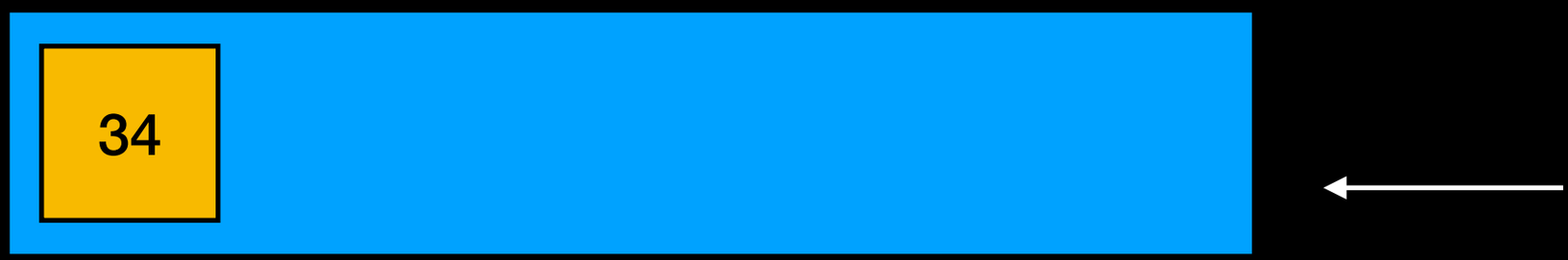
Double ended queue (deque)

Can add and remove to/from front and back

# Deque

Double ended queue (deque)

Can add and remove to/from front and back



# Deque

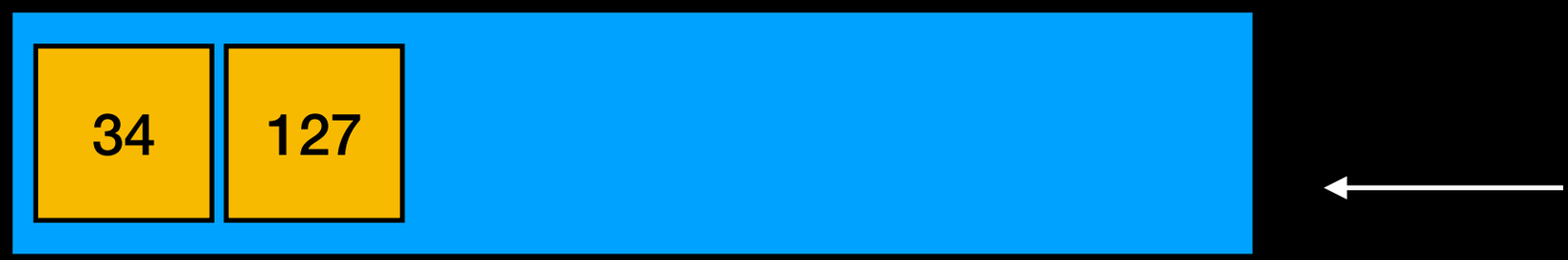
Double ended queue (deque)

Can add and remove to/from front and back

# Deque

Double ended queue (deque)

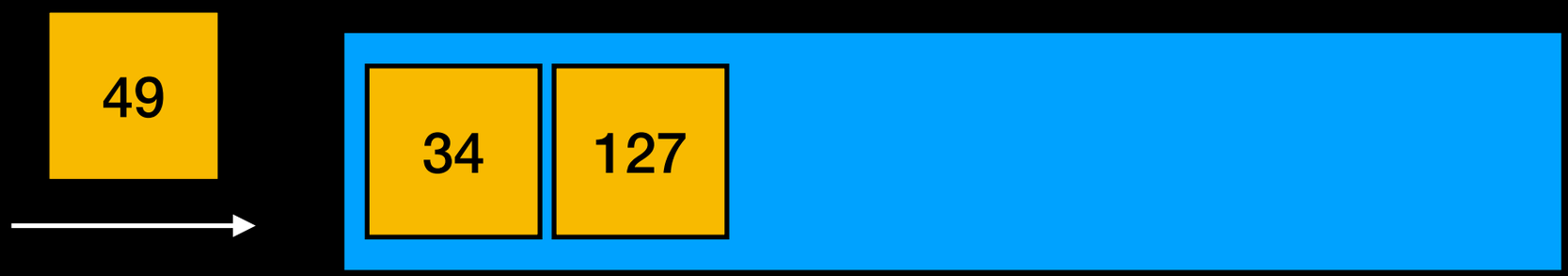
Can add and remove to/from front and back



# Deque

Double ended queue (deque)

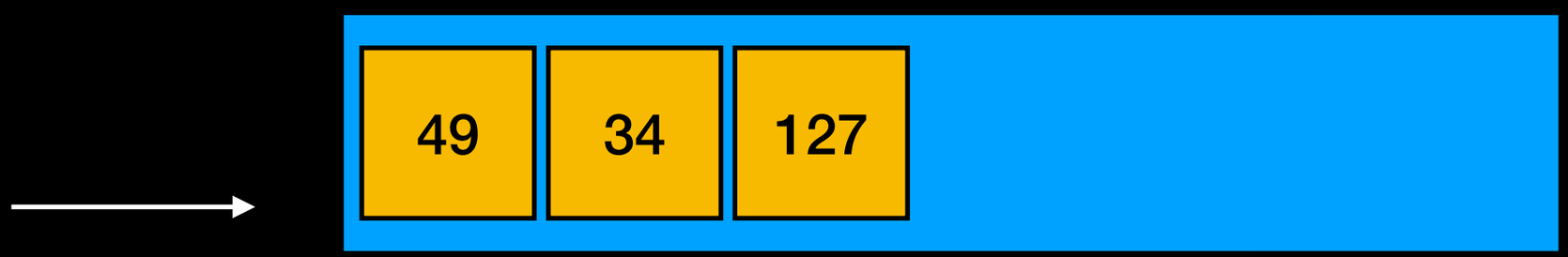
Can add and remove to/from front and back



# Deque

Double ended queue (deque)

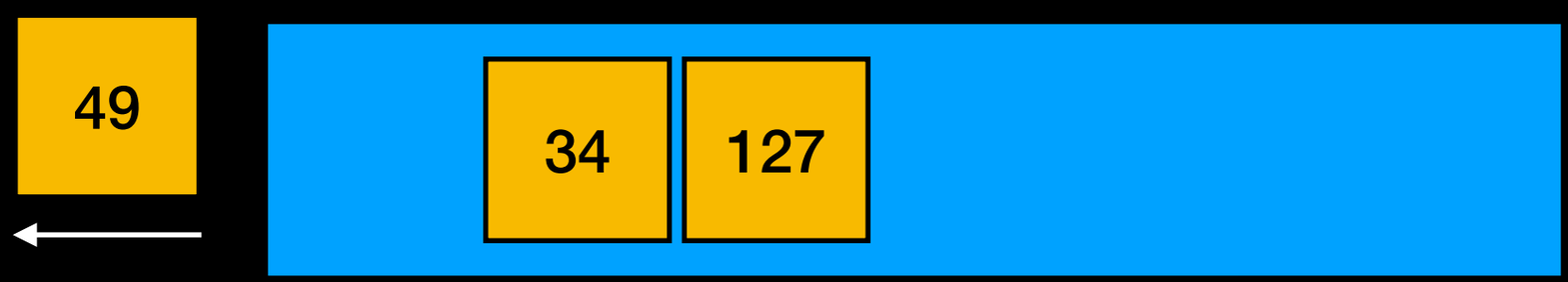
Can add and remove to/from front and back



# Deque

Double ended queue (deque)

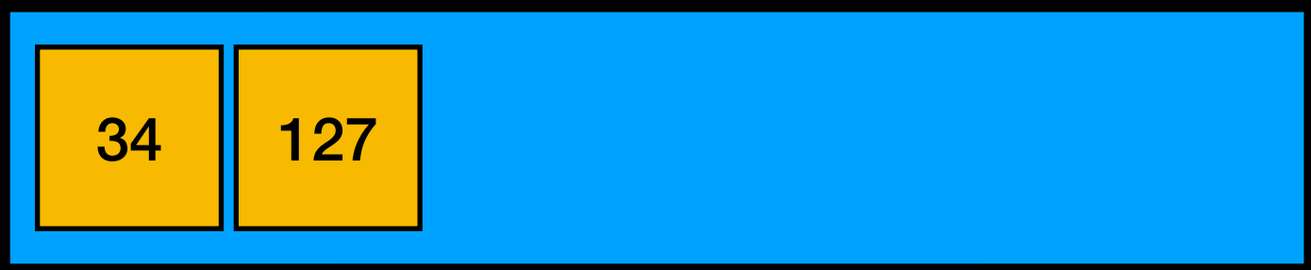
Can add and remove to/from front and back



# 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 deque

# 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 language “push” and “pop”, adapting with “push_back”, “push_front” etc. for all containers

# Priority Queue

A queue of items “sorted” by priority

**Low Priority**

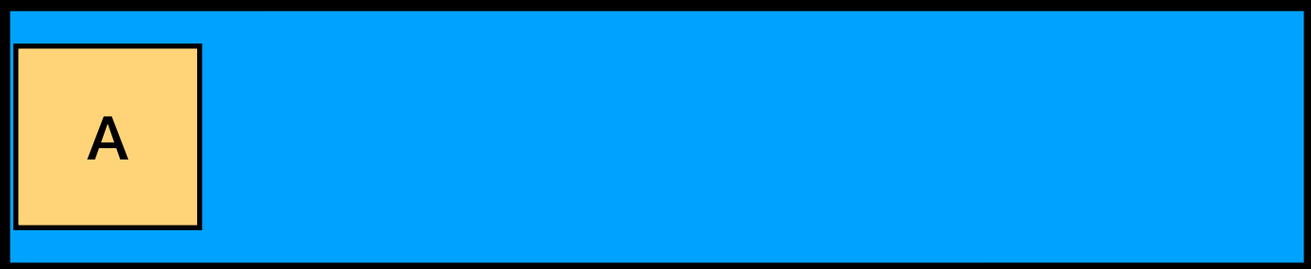
**High Priority**

# Priority Queue

A queue of items “sorted” by priority

**Low Priority**

**High Priority**



# Priority Queue

A queue of items “sorted” by priority

**Low Priority**

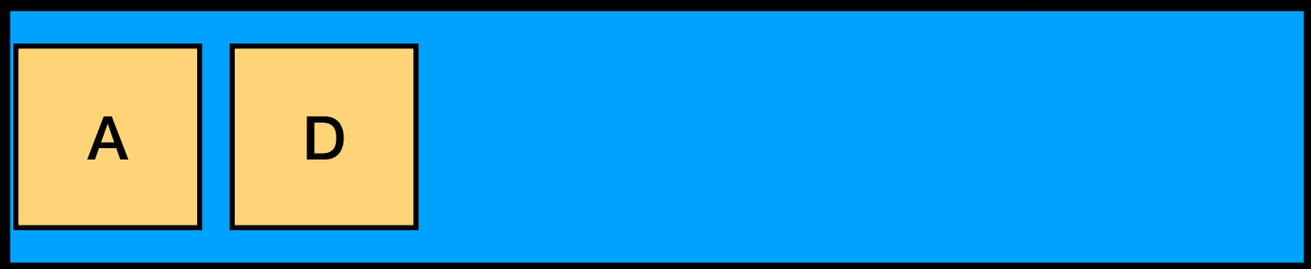
**High Priority**

# Priority Queue

A queue of items “sorted” by priority

**Low Priority**

**High Priority**



# Priority Queue

A queue of items “sorted” by priority

**Low Priority**

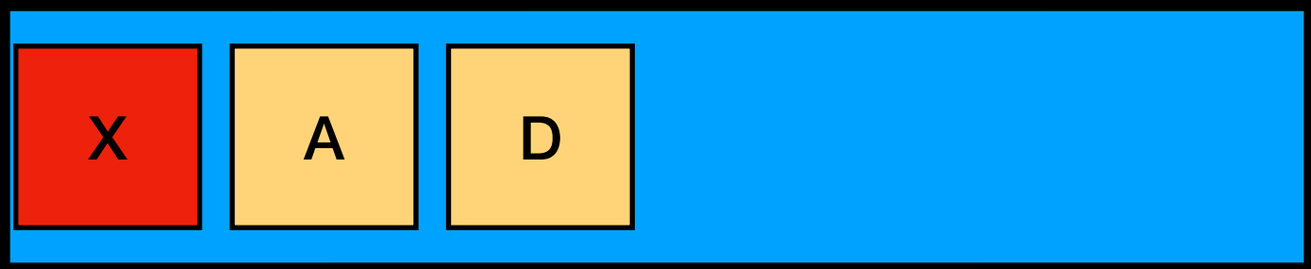
**High Priority**

# Priority Queue

A queue of items “sorted” by priority

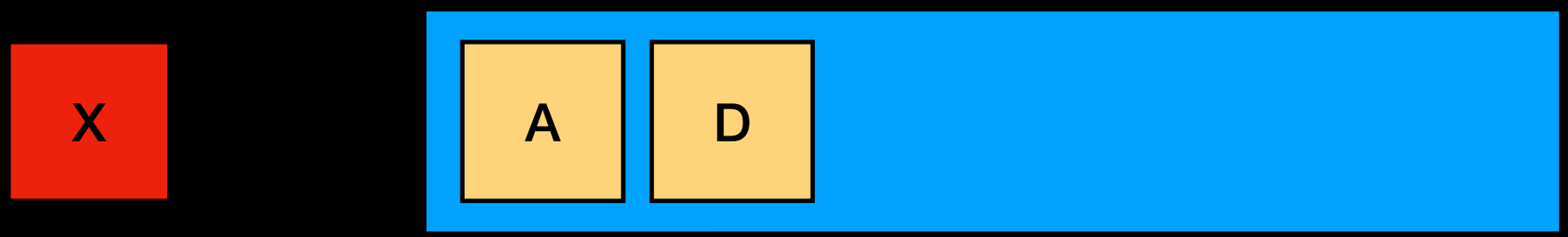
**Low Priority**

**High Priority**



# Priority Queue

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

# Explore the STL

It’s time to get to know it!!!

C++ Interlude 8 in your textbook

https://en.cppreference.com/w/cpp/standard_library

https://en.cppreference.com/w/cpp/container

https://en.cppreference.com/w/cpp/algorithm

You should use STL stack and queue for Project 6

Explore as you learn about new ADTs and algorithms.

# Main Components

Containers

Algorithms

Functions

Iterators

# Main Components

Containers

Algorithms

Functions

Iterators

# Sequence Containers

`array`

`vector`

`deque`

`forward_list` (singly linked)

`list` (doubly linked)

Contiguous memory

Fixed size

Grow and Shrink

Also allows access to any location with [ ]

Non-contiguous memory
Growing and shrinking more efficient

# Container Adaptors

Impose a different interface for the underlying container

`stack`

`queue`

`priority_queue`

# Algorithms

```
#include <algorithm>
```

Search and Compare Algorithms examples

```
    for_each() // applies a function to a range in container
    count() // counts the occurrences of an item within a range
    max_element() // returns the max value within a range
```

Sequence Modification Algorithms examples

```
    copy() //copies items within a range starting at given position
           within same or different container
    fill() // sets all entries within a range to give value
```

Sorting and Searching Algorithms examples

```
    sort() // sorts entries within a range in ascending order –
              typically some variation of QuickSort
    stable_sort() // "" – typically MergeSort may vary
    binary_search() // determines if an item exist in a given range
                       in a sorted container
```

. . . much more!!!