Collections, Part One

Organizing Data

- In order to model and solve problems, we have to have a way of representing structured data.
- We need ways of representing concepts like
  - sequences of elements,
  - sets of elements,
  - associations between elements,
  - etc.

Collections

- A ***collection class*** (or ***container class***) is a data type used to store and organize data in some form.
- Understanding and using collection classes is critical to good software engineering.
- Our next three lectures are dedicated to exploring different collections and how to harness them appropriately.
- Later in the quarter, we'll see how these types work and analyze their efficiencies. For now, let's just focus on how to use them.

Stack

Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

42

137

Stack

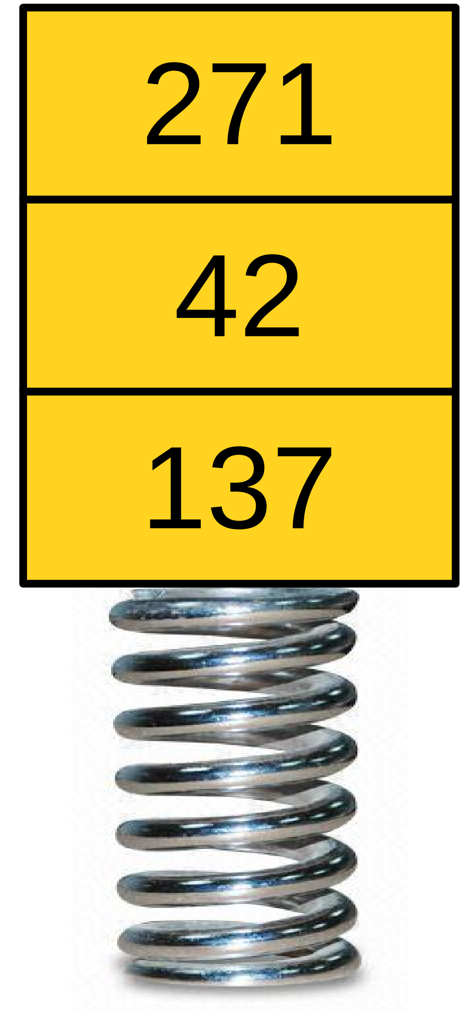
- A **`Stack`** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

Stack

- A `Stack` is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.



Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

271

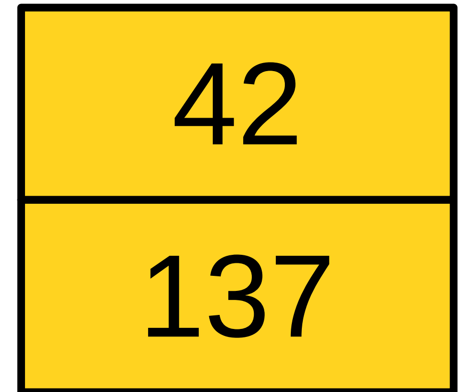
42

137

Stack

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

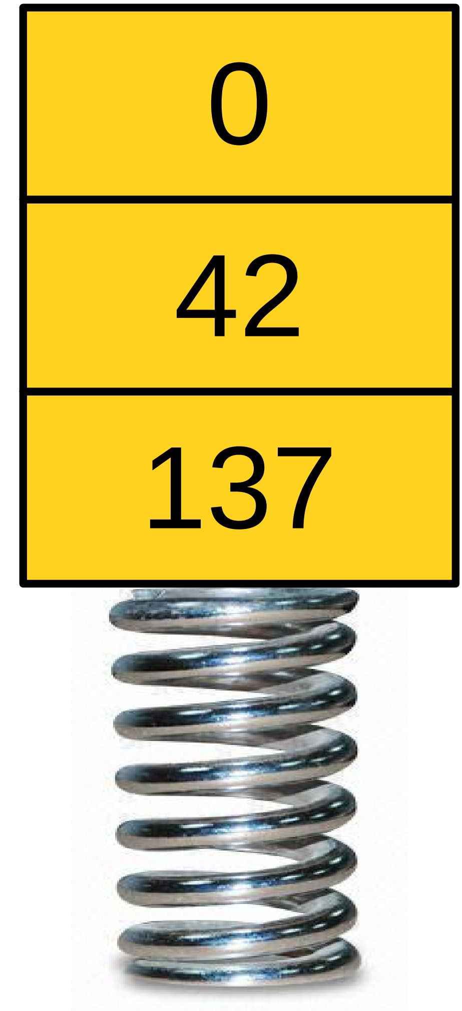
Stack



- A **`Stack`** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.

Stack

- A **`Stack`** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of the stack or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- This is why it's called the call *stack* and we talk about *stack* traces.



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
 ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
  ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
   ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
    ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
     ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
       ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
       ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
        ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
         ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
         ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
          ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
            ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
             ^
```

Balancing Parentheses

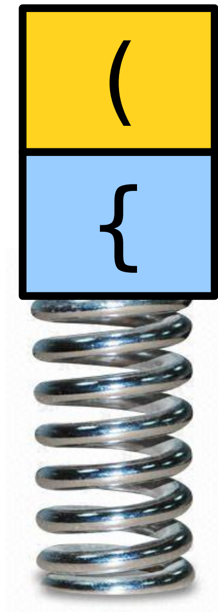
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
              ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
               ^
```

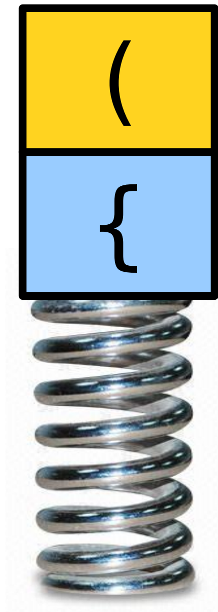
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
               ^
```



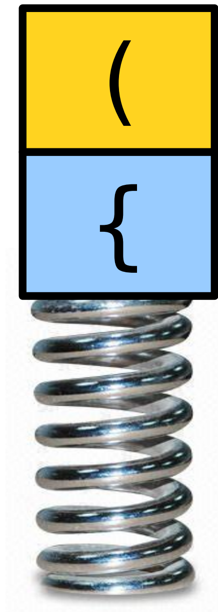
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                ^
```



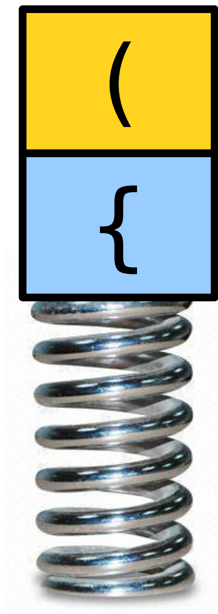
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                 ^
```



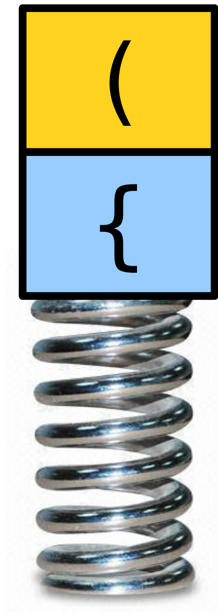
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                  ^
```



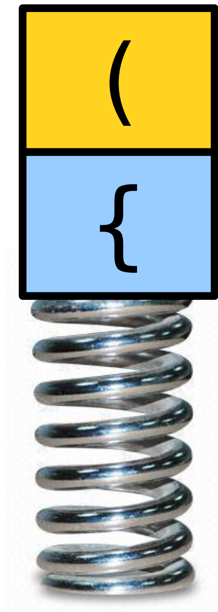
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                   ^
```



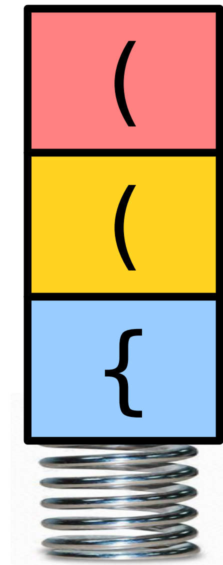
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



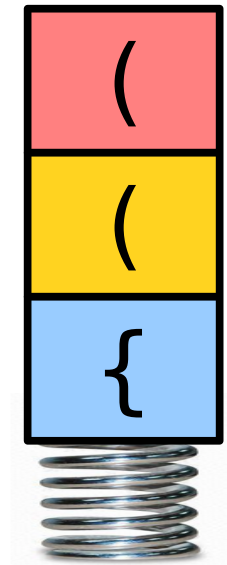
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



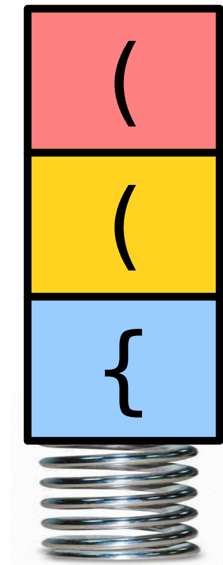
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                     ^
```



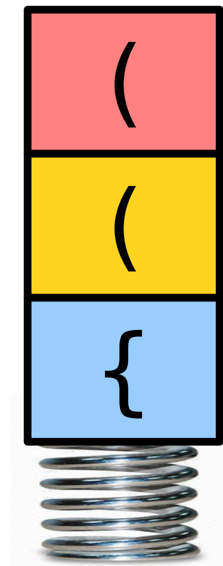
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                         ^
```



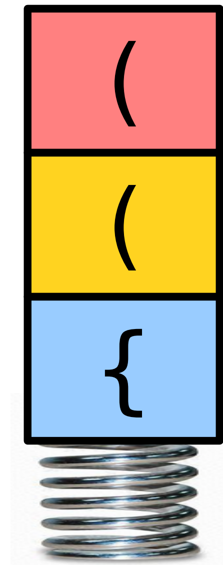
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                    ^
```



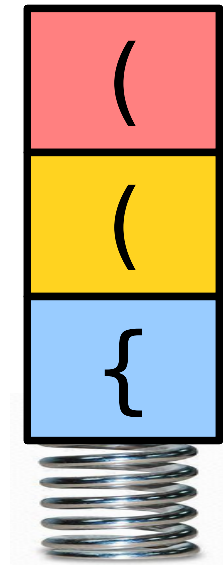
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                    ^
```



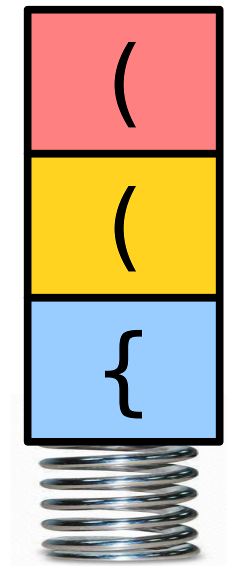
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                         ^
```



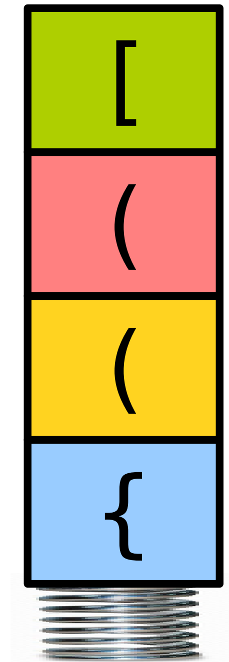
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                             ^
```



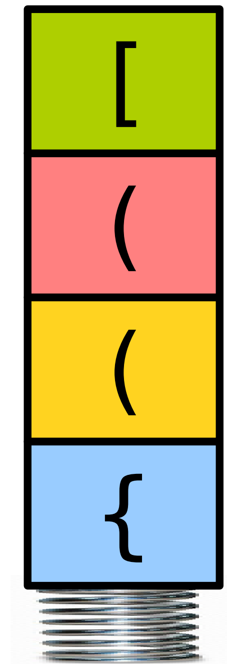
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



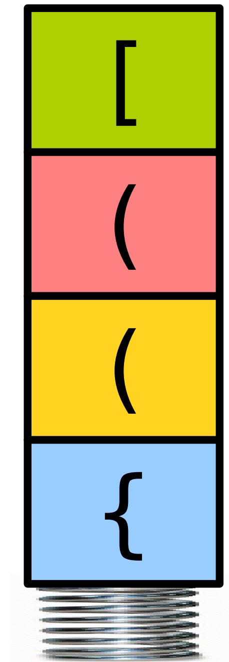
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                             ^
```



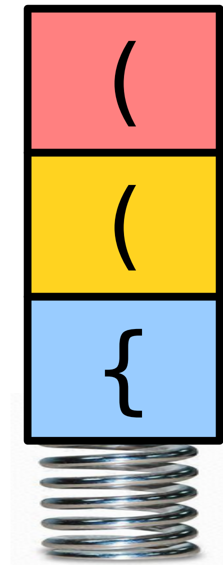
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                               ^
```



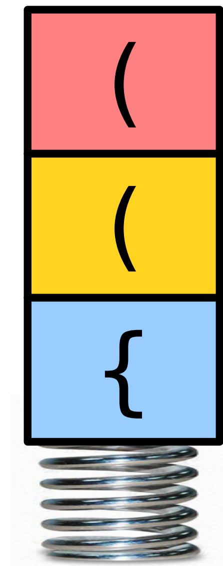
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



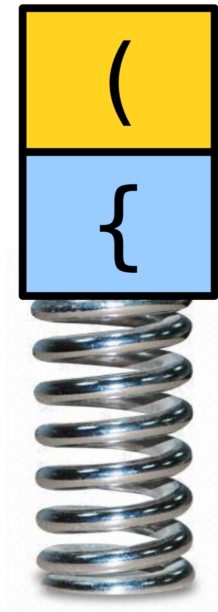
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                             ^
```



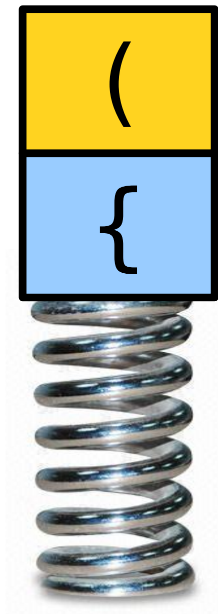
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                              ^
```



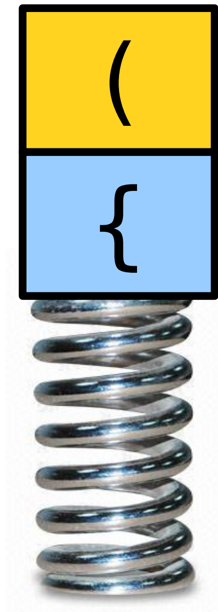
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                               ^
```



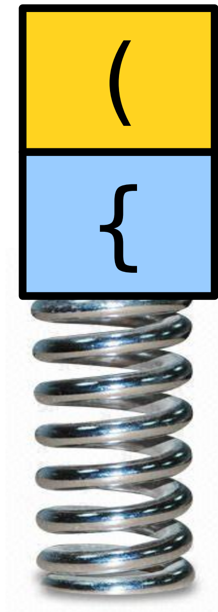
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



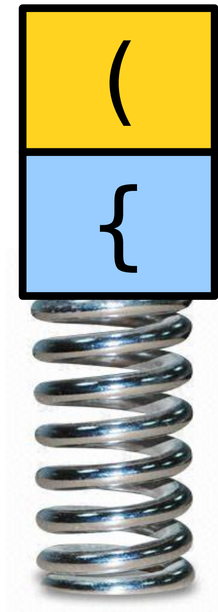
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                ^
```



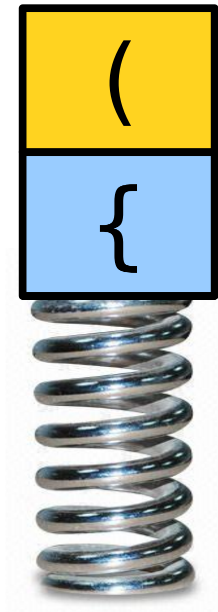
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                 ^
```



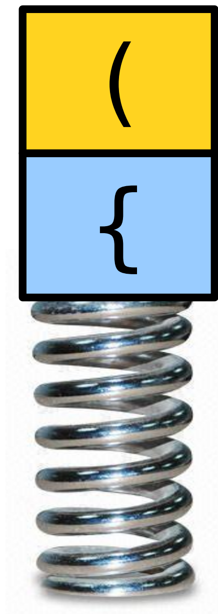
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                    ^
```



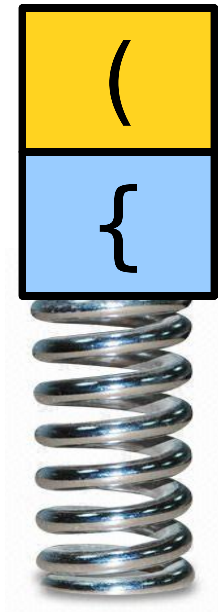
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                    ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                    ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                    ^
```

Balancing Parentheses

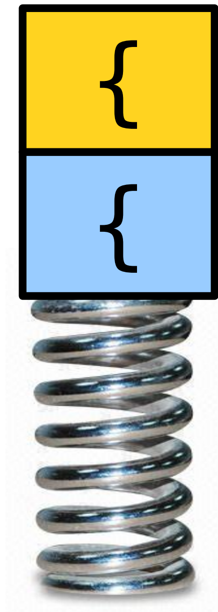
```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```

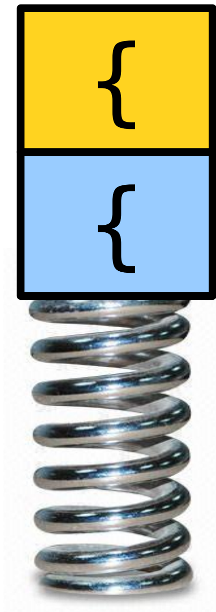
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                      ^
```



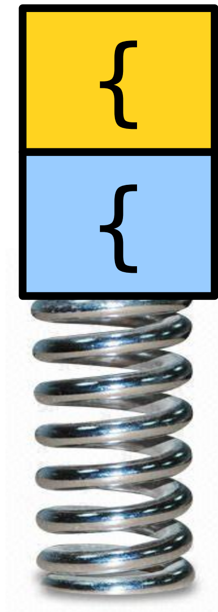
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



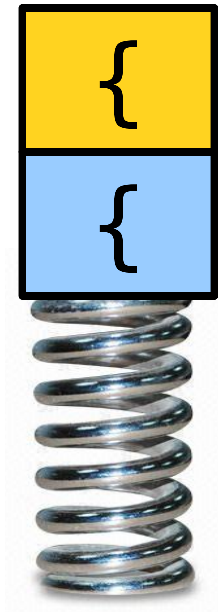
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                        ^
```



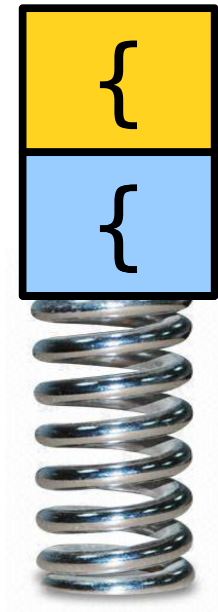
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



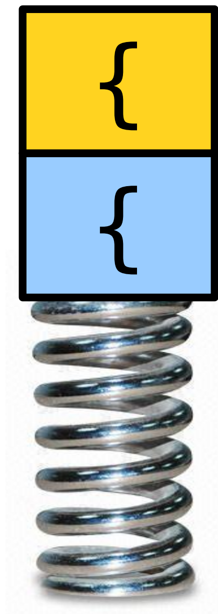
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



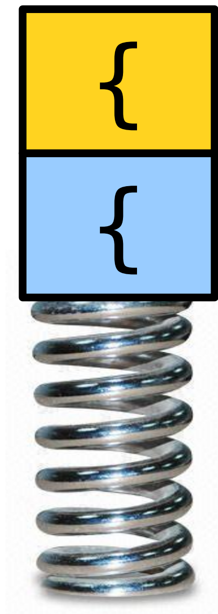
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                          ^
```



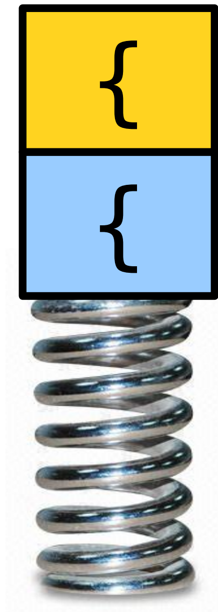
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                         ^
```



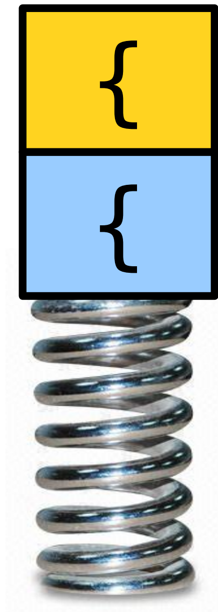
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                              ^
```



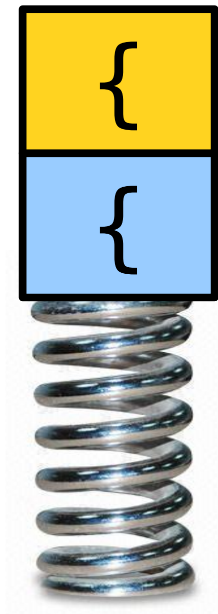
Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                               ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```



Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                 ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                 ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                 ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
                                                  ^
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Balancing Parentheses

```
int foo() { if (x * (y + z[1]) < 137) { x = 1; } }
```

Let's go code this up!

Objects in C++

- In Java, if you declare a variable like

```
Stack<String> myStack;
```

then `myStack` is *not* the actual stack; it's a reference to it.

- As a result, you have to both declare the variable and initialize it, like this:

```
Stack<String> myStack = new Stack<String>(); // Okay in Java
```

- In C++, if you write

```
Stack<string> myStack;
```

you get back an *honest to goodness* `Stack<string>`. It's not a reference to a `Stack<string>` or a pointer to one.

- As a result, in C++, you should *not* write

```
Stack<string> myStack = new Stack<string>(); // Error in C++!
```

Range-Based For Loops

- In C++, you can iterate over all the characters of a string, in order, by writing

```
for (char ch: str) {
    // do something with ch
}
```

- This same syntax can be used to iterate over a bunch of other collections as well.

Our Algorithm

- For each character:
  - If it’s an open parenthesis of some sort, push it onto the stack.
  - If it’s a close parenthesis of some sort:
    - If the stack is empty, report an error.
    - If the character doesn’t pair with the character on top of the stack, report an error.
- At the end, return whether the stack is empty (nothing was left unmatched.)

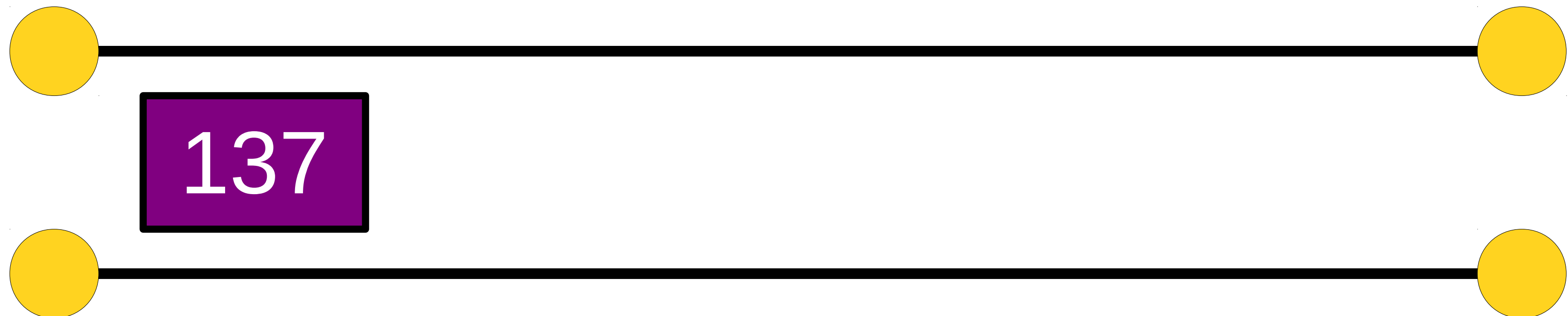
Other `Stack` Applications

- Stacks show up all the time in ***parsing***, recovering the structure in a piece of text.
  - Often used in natural language processing; take CS224N for details!
  - Used all the time in compilers – take CS143 for details!
- They're also used as building blocks in larger algorithms for doing things like
  - making sure a city's road networks are navigable (finding *strongly connected components*; take CS161 for details!) and
  - searching for the best solution to a problem (stay tuned!)

Queue

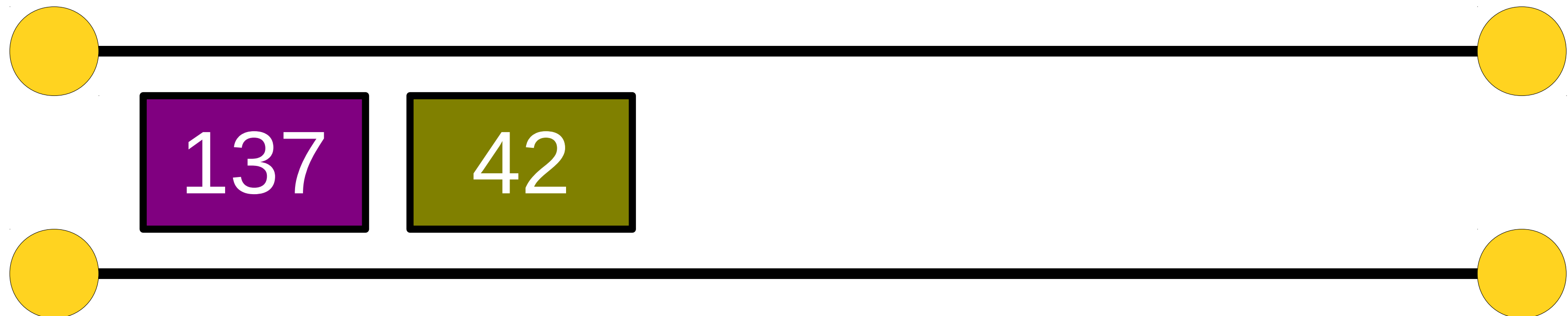
Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.



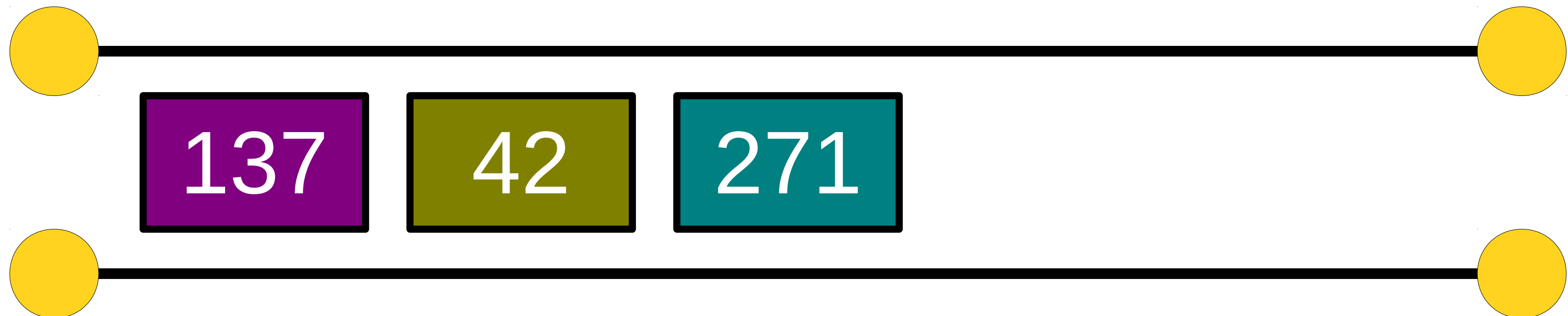
Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.



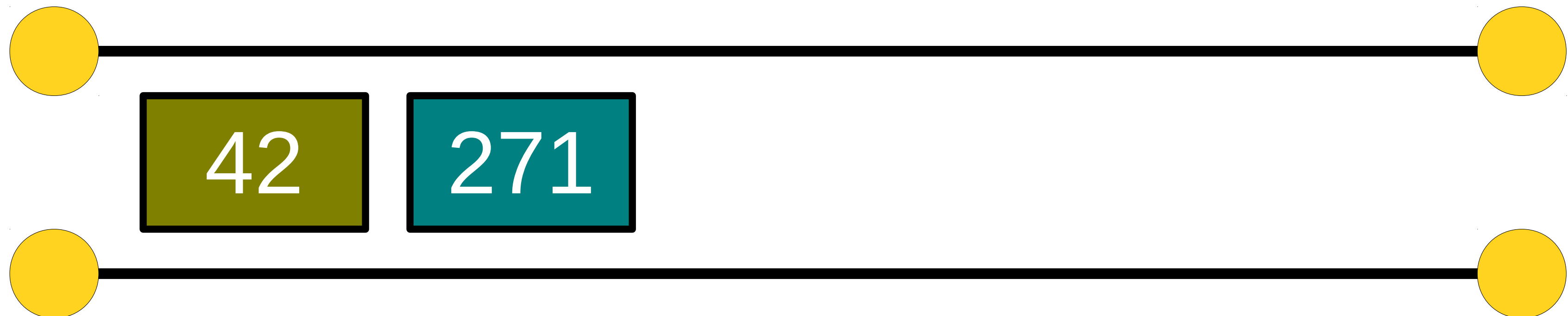
Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.



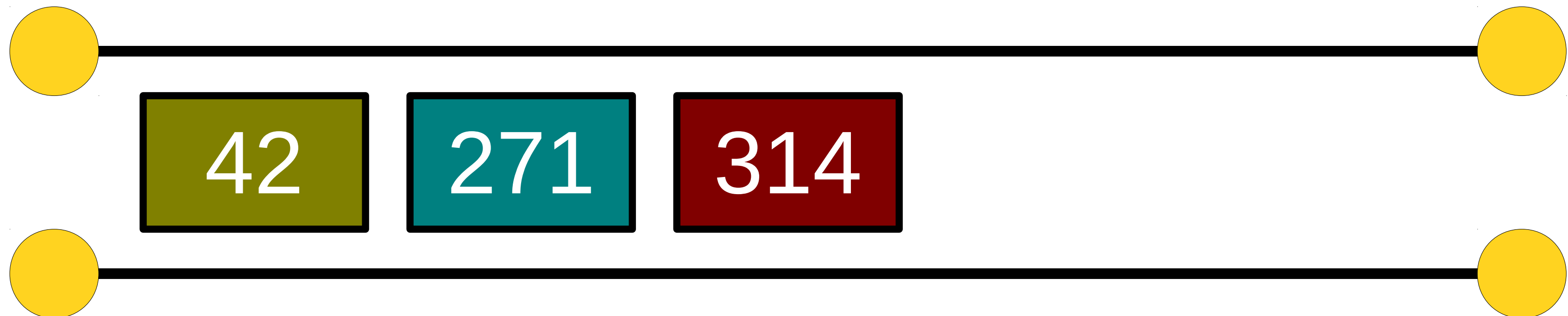
Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.



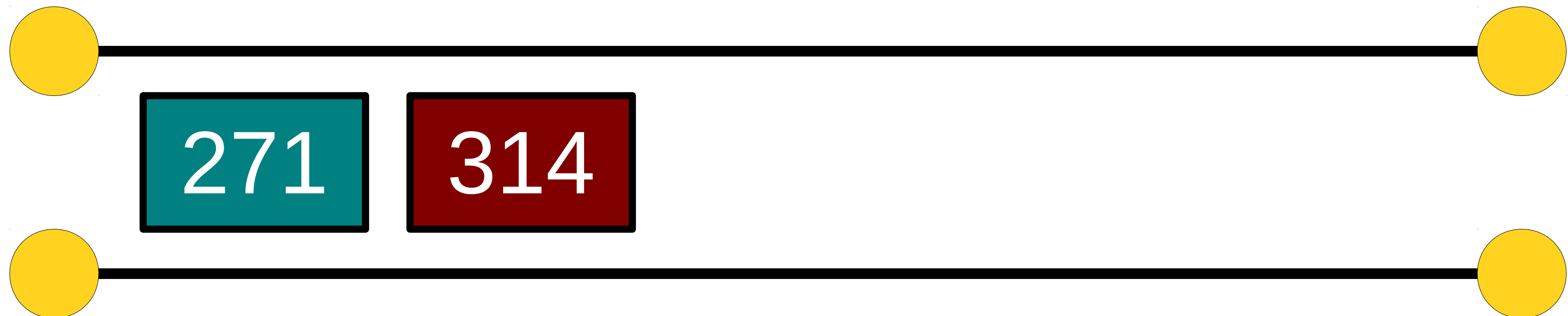
Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.



Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.



Queue

- A `Queue` is 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.
- No other objects in the queue are visible.
- Example: A checkout counter.

An Application: ***The Library of Babel***

“[T]he Library is total and that its shelves register all the possible combinations of the twenty-odd orthographical symbols (a number which, though extremely vast, is not infinite): Everything: the minutely detailed history of the future, the archangels' autobiographies, the faithful catalogues of the Library, thousands and thousands of false catalogues, the demonstration of the fallacy of those catalogues, the demonstration of the fallacy of the true catalogue, the Gnostic gospel of Basilides, the commentary on that gospel, the commentary on the commentary on that gospel, the true story of your death, the translation of every book in all languages, the interpolations of every book in all books.”

-Jorge Luis Borges, *The Library of Babel*

Generating All Possible Strings

- Suppose we want to generate all possible strings up to some fixed length.
- How might we do this?

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

" "

"A"

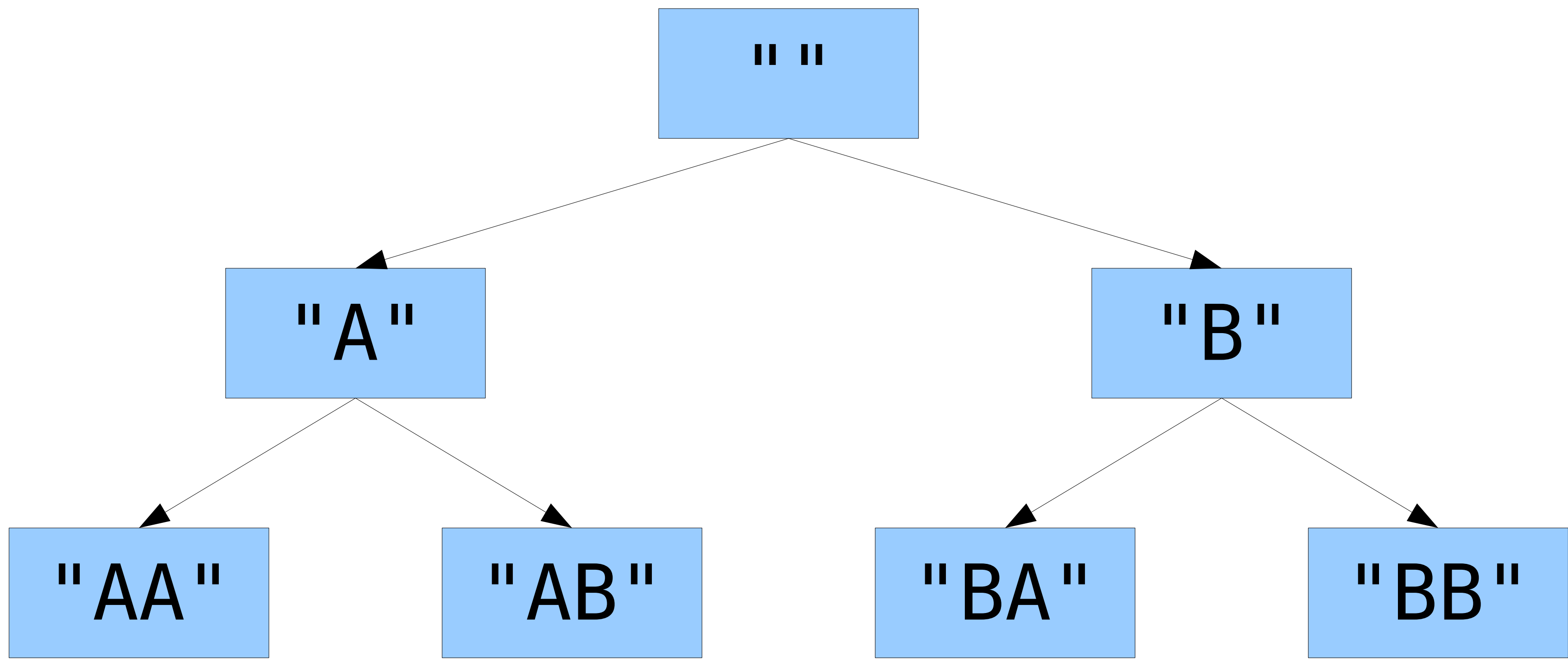
"B"

"AA"

"AB"

"BA"

"BB"


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

" "

"A"

"B"

"AA"

"AB"

"BA"

"BB"

" "

"A"

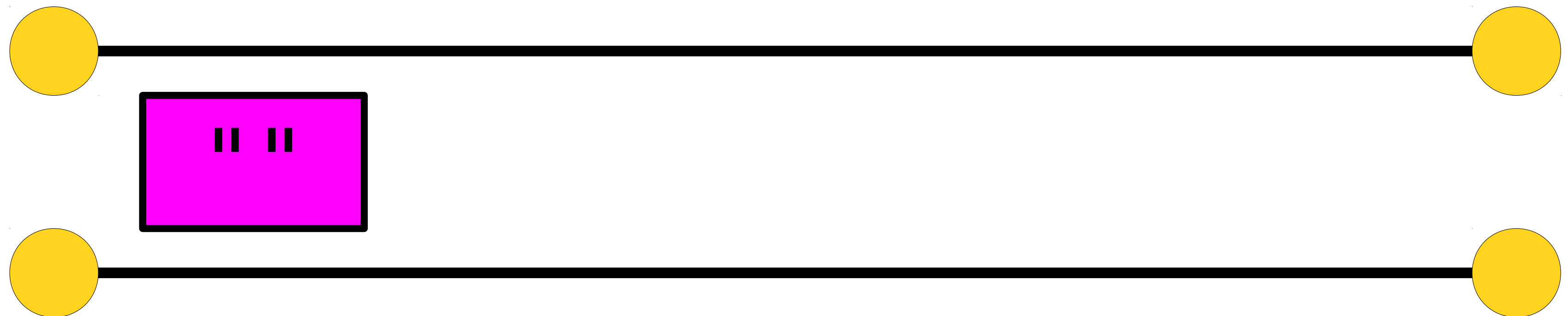
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

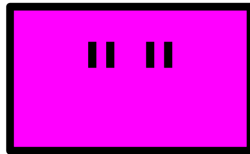
"B"

"AA"

"AB"

"BA"

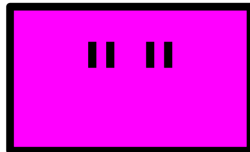
"BB"



" "

"A" "B"

"AA" "AB" "BA" "BB"



" "

"A"

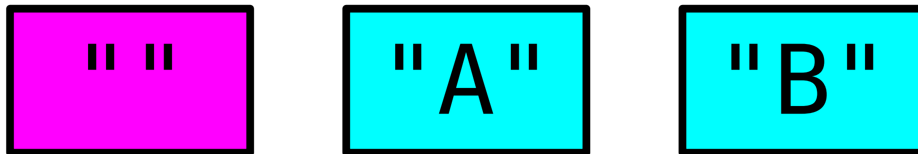
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

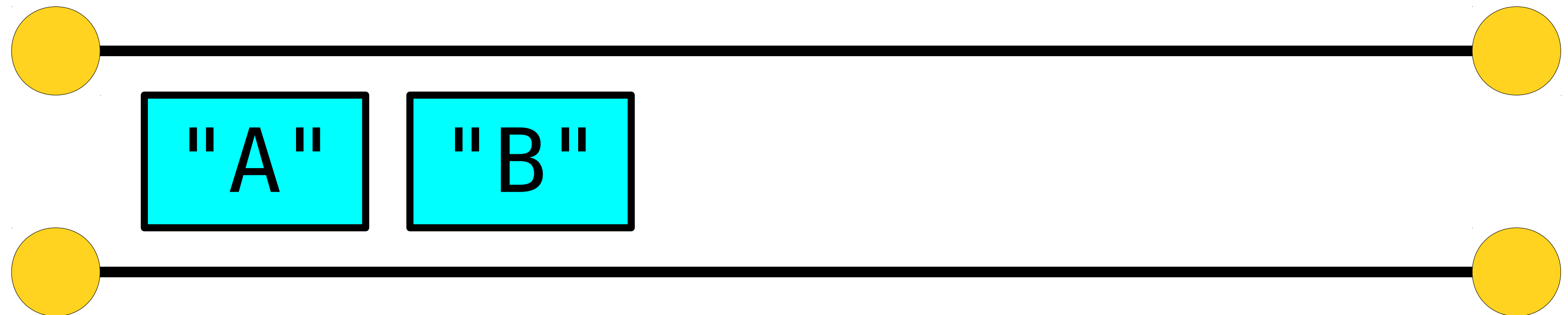
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

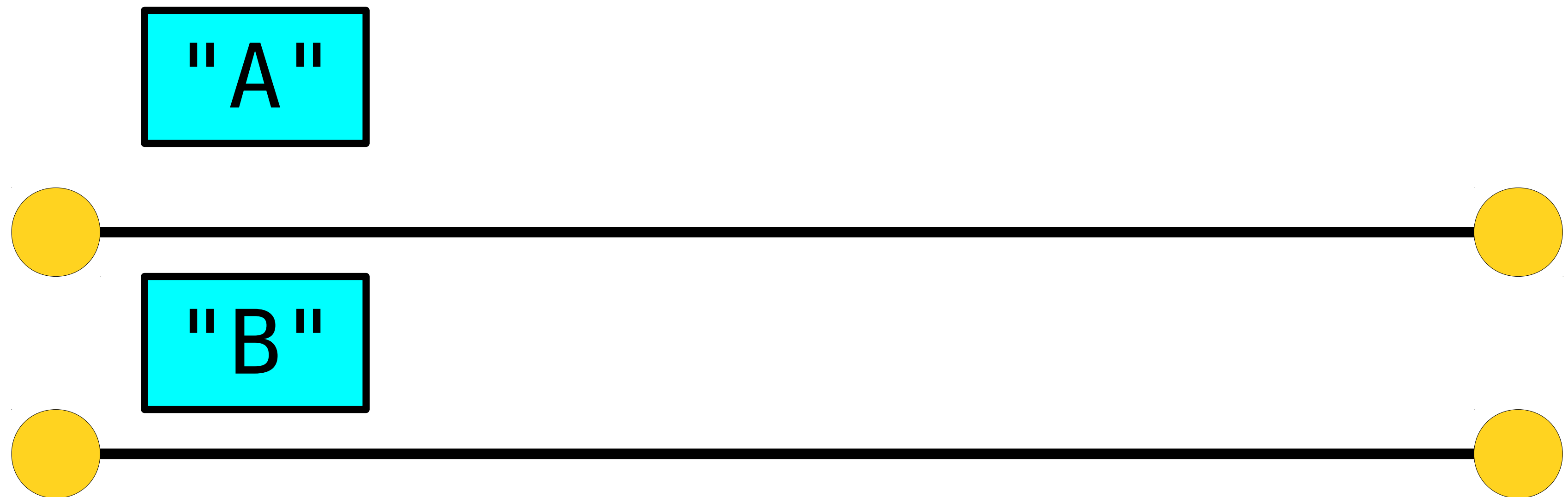
"B"

"AA"

"AB"

"BA"

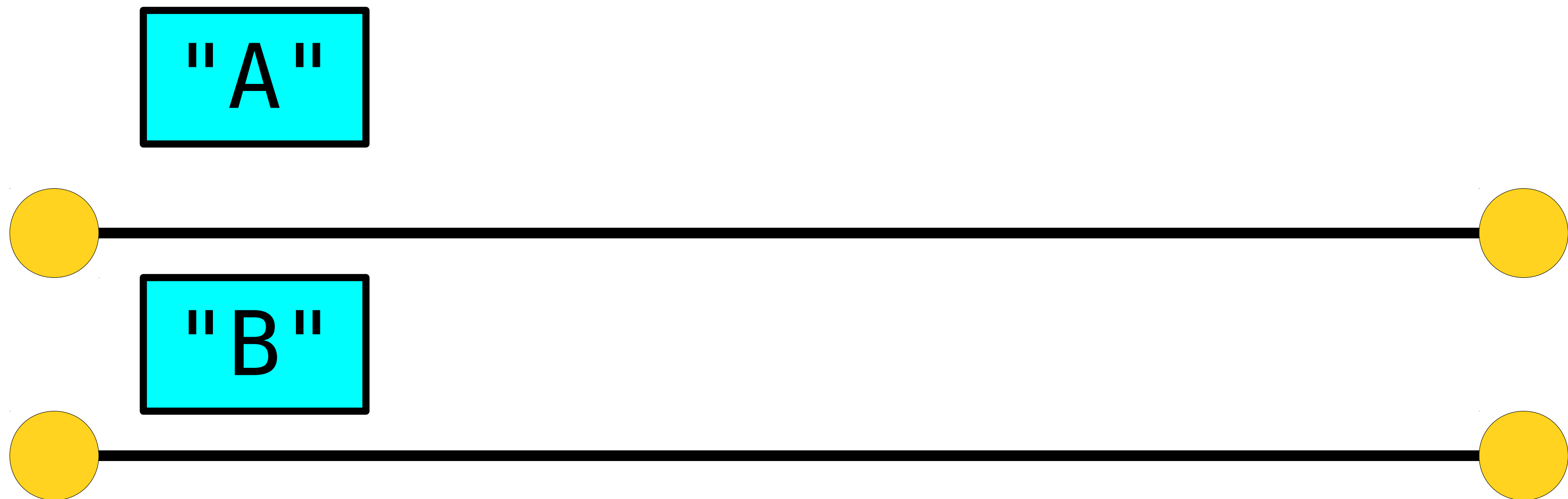
"BB"



" "

"A" "B"

"AA" "AB" "BA" "BB"



" "

"A"

"B"

"AA"

"AB"

"BA"

"BB"

" "

"A"

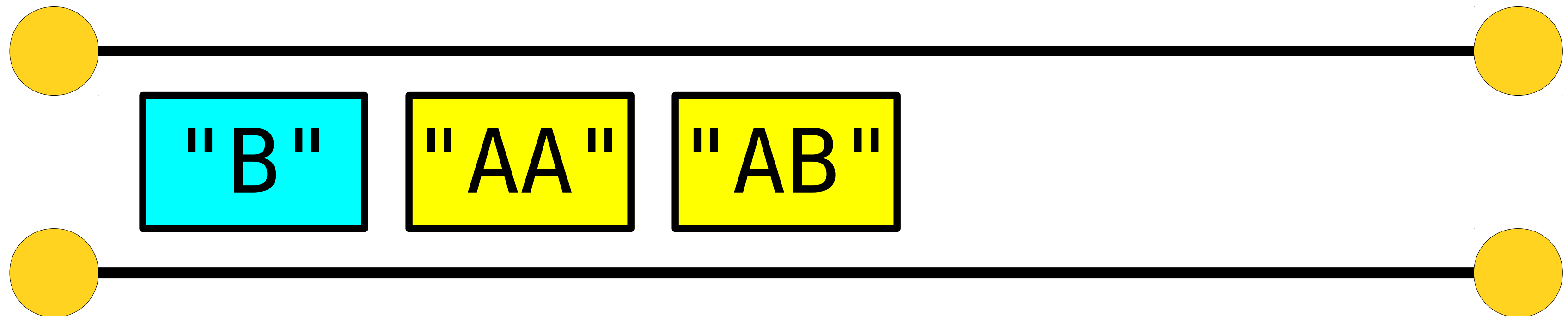
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

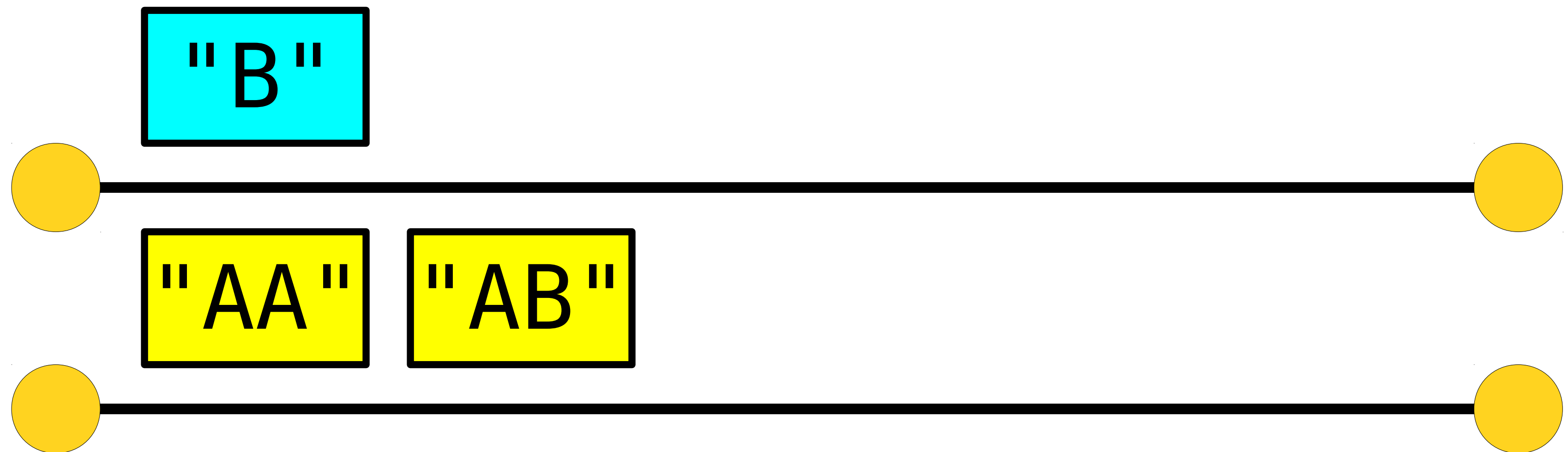
"B"

"AA"

"AB"

"BA"

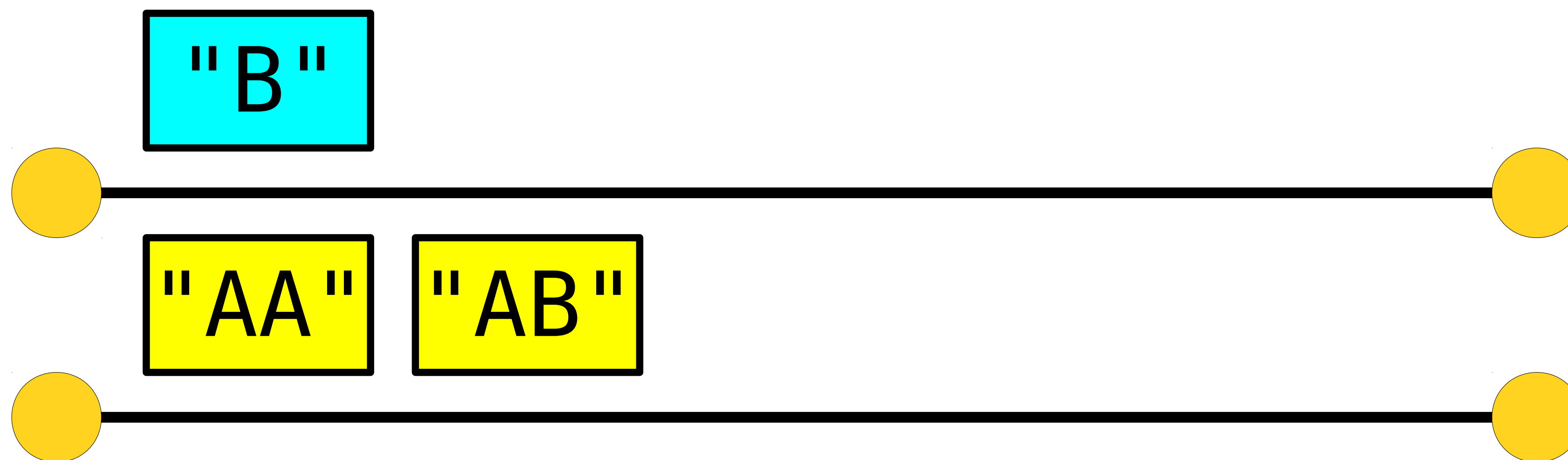
"BB"



" "

"A" "B"

"AA" "AB" "BA" "BB"



" "

"A"

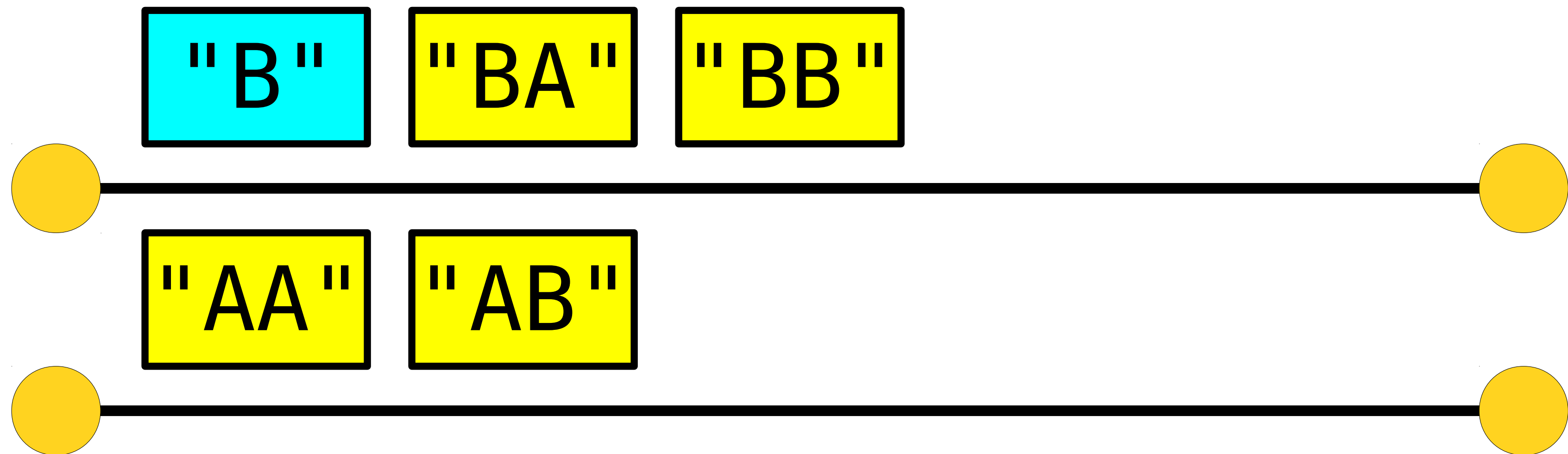
"B"

"AA"

"AB"

"BA"

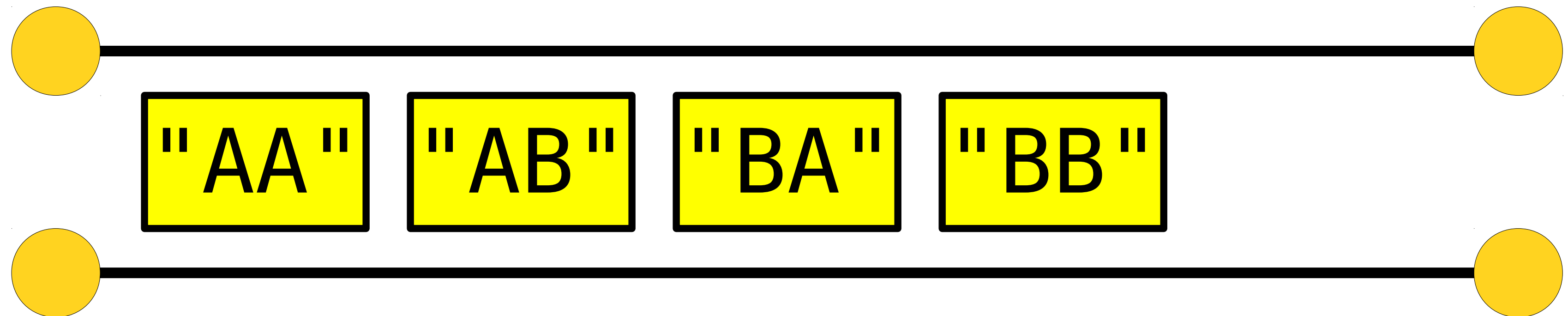
"BB"



" "

"A" "B"

"AA" "AB" "BA" "BB"



" "

"A"

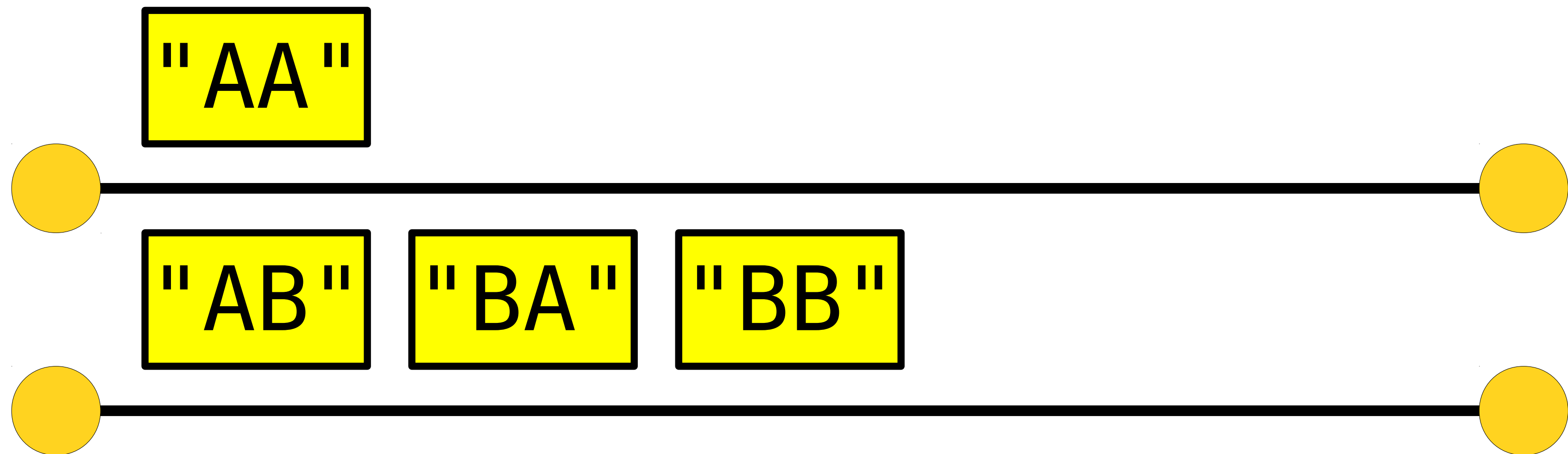
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

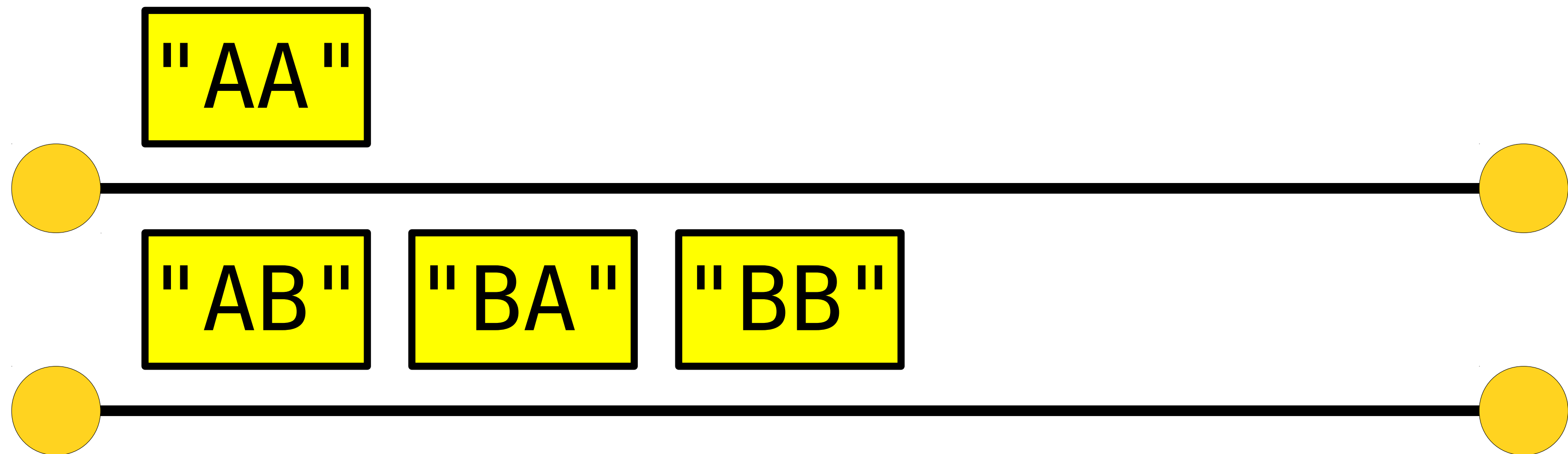
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

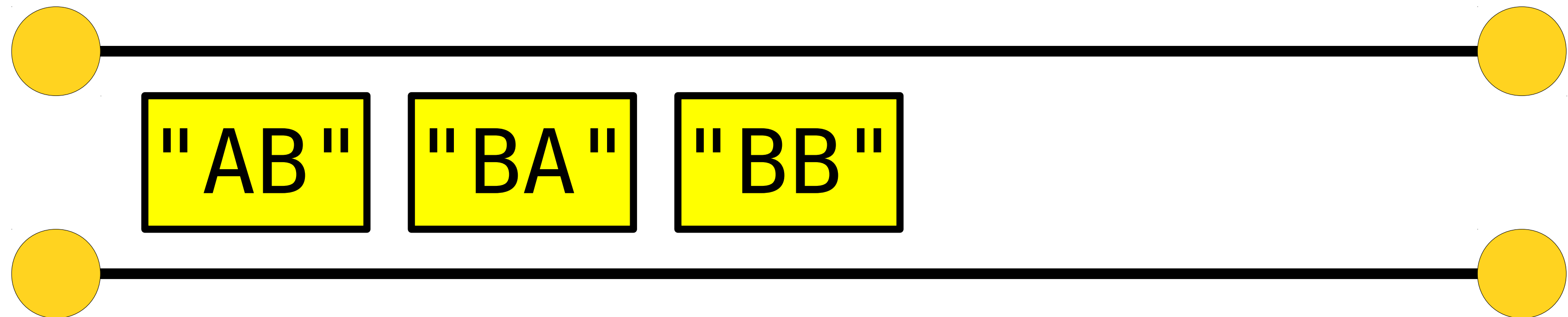
"B"

"AA"

"AB"

"BA"

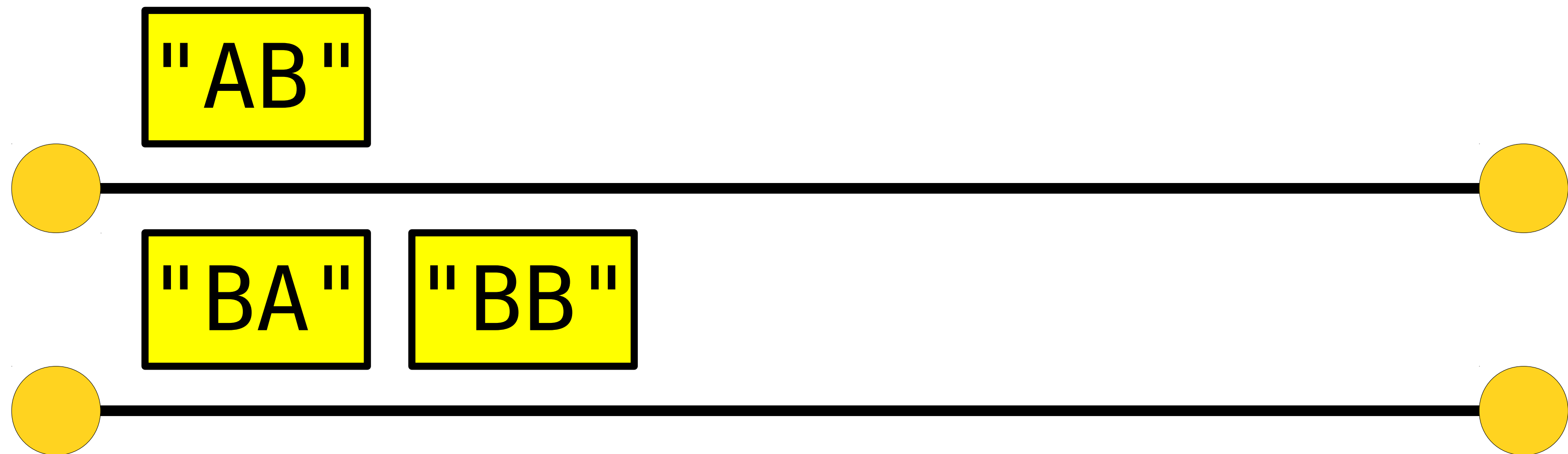
"BB"



" "

"A" "B"

"AA" "AB" "BA" "BB"



" "

"A"

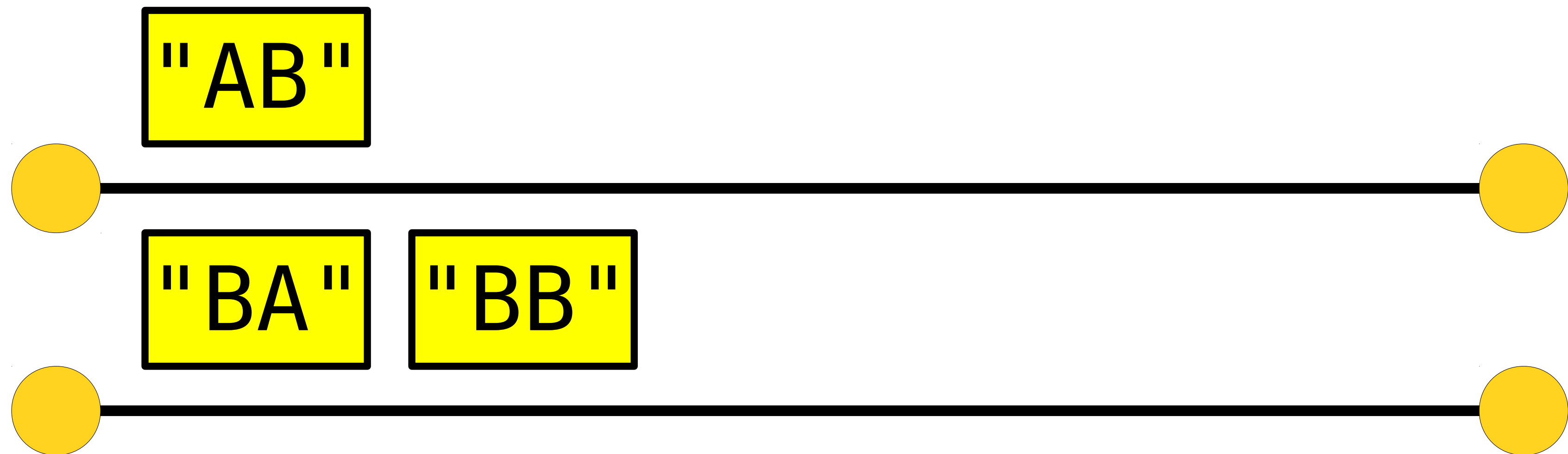
"B"

"AA"

"AB"

"BA"

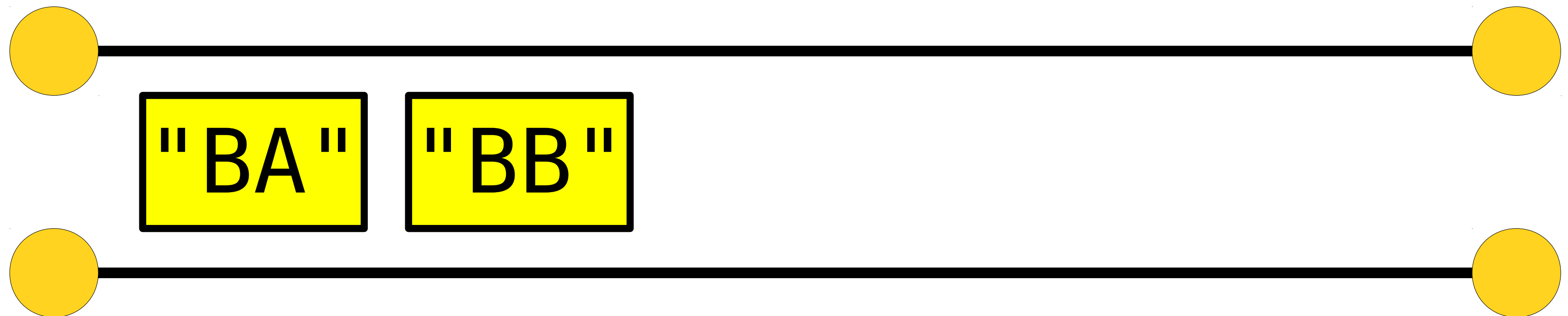
"BB"



" "

"A" "B"

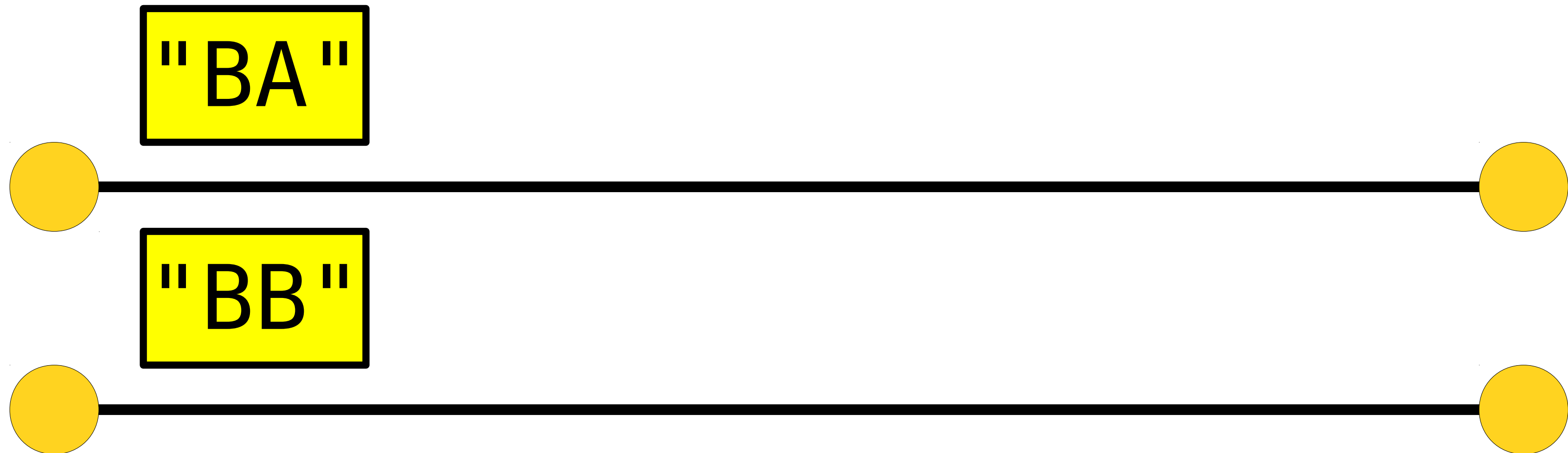
"AA" "AB" "BA" "BB"



" "

"A" "B"

"AA" "AB" "BA" "BB"



" "

"A"

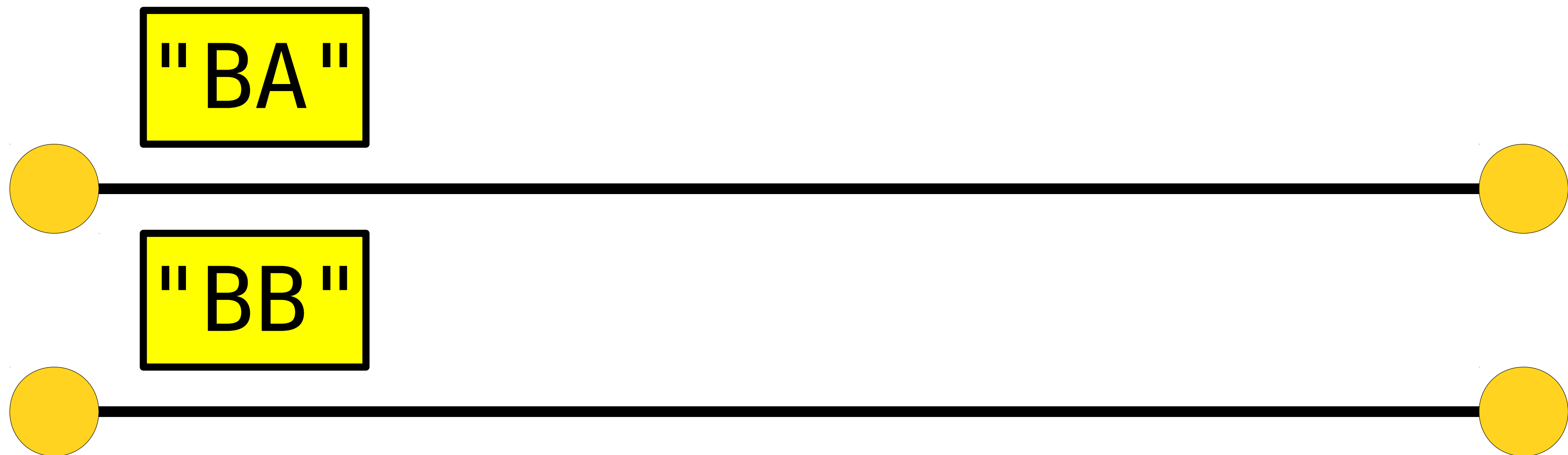
"B"

"AA"

"AB"

"BA"

"BB"



" "

"A"

"B"

"AA"

"AB"

"BA"

"BB"

" "

"A"

"B"

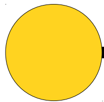
"AA"

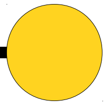
"AB"

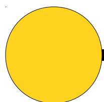
"BA"

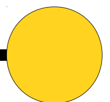
"BB"

"BB"









" "

"A" "B"

"AA" "AB" "BA" "BB"

"BB"

" "

"A"

"B"

"AA"

"AB"

"BA"

"BB"

Let’s code it up!

Our Algorithm

- Start with the empty string in a queue.
- While there are still things in the queue:
  - Pull one off.
  - Print it.
  - If it isn’t at the maximum length:
    - For each possible next character:
      - Put the string formed by appending that character back into the queue.

Breadth-First Search

- This general algorithm is called ***breadth-first search***.
- It's often used to find the fastest or best way to do something, since it lists objects in increasing order of "size."
- The algorithm that Google Maps uses is closely related to this algorithm. Stay tuned for details!
- You'll see some other applications of this algorithm in Assignment 2.

Time-Out for Announcements!

Assignment 1

- Assignment 1 is due this upcoming Monday at the start of class (11:30AM).
- Need help?
  - Stop by the LaIR! 6PM – Midnight, Sundays through Thursdays, on the ground floor of Tresidder.
  - Stop by Anton or Keith’s office hours!
  - Ask questions on Piazza!
  - Ask your section leader!

WiCS Casual CS Dinner

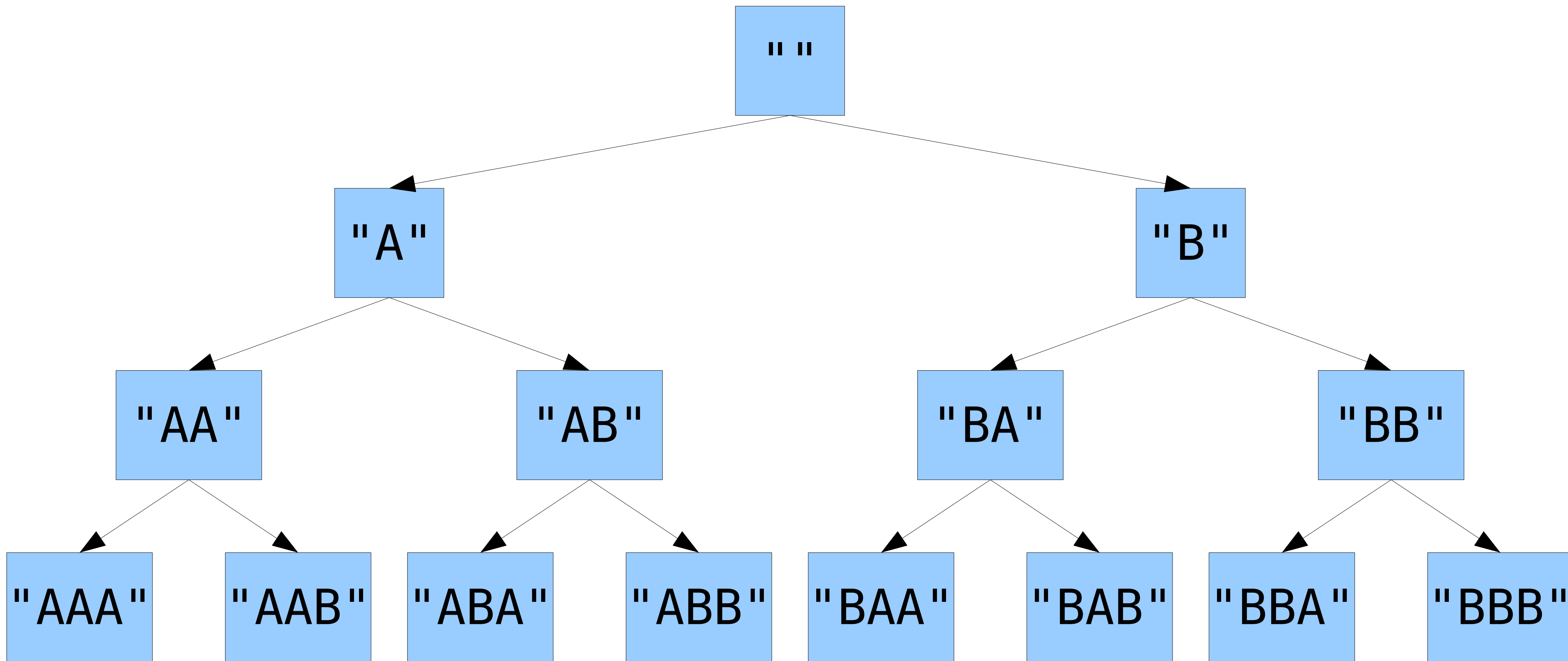
- WiCS is holding its first of its biquarter Casual CS Dinners this upcoming ***Monday, February 23*** at 6PM in the WCC.
- Highly recommended – these dinners are a real highlight of the quarter.
- Interested? RSVP at

https://goo.gl/forms/TaiRja7K2L5bcocG3

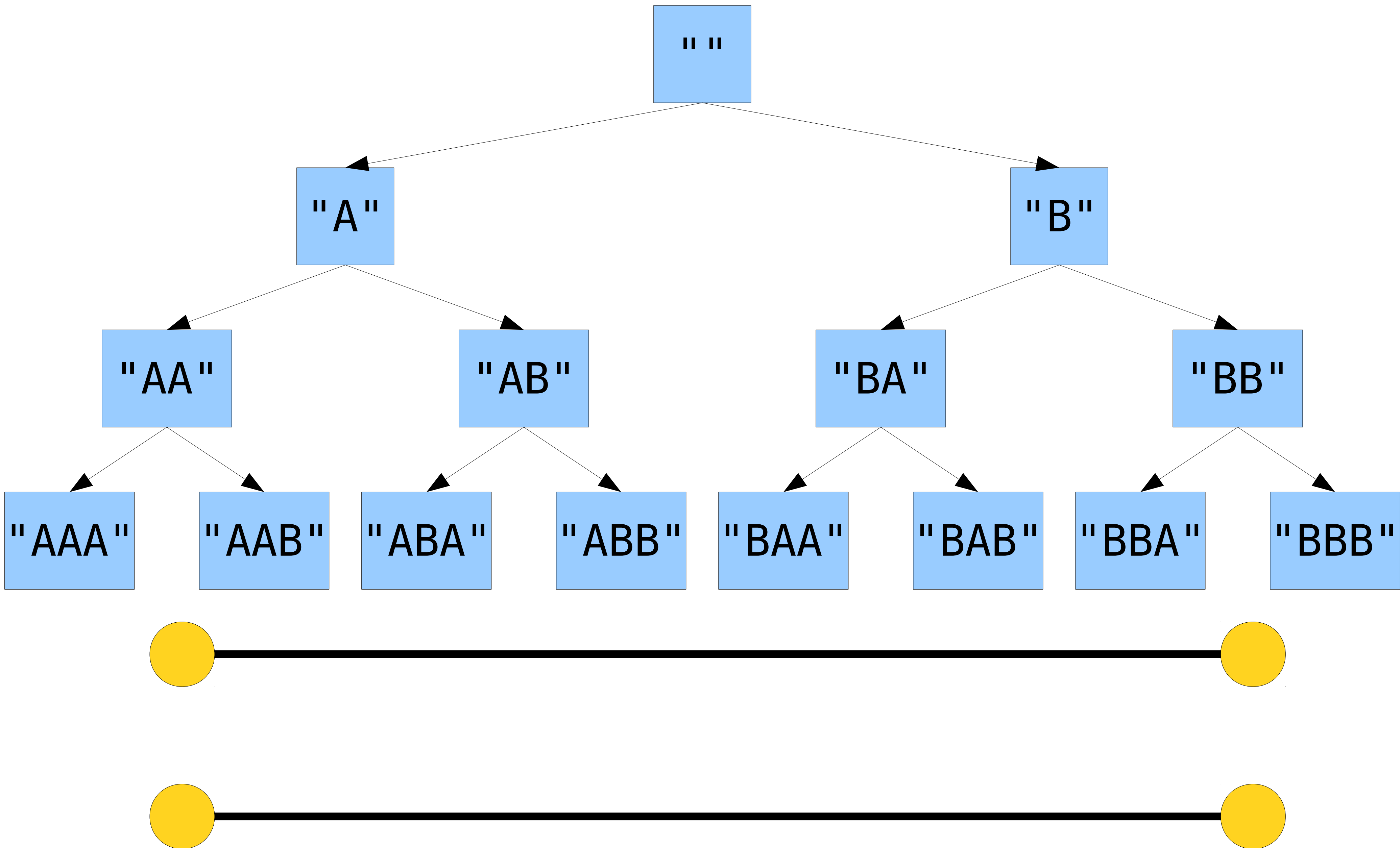
Many Happy Returns!

One Caveat

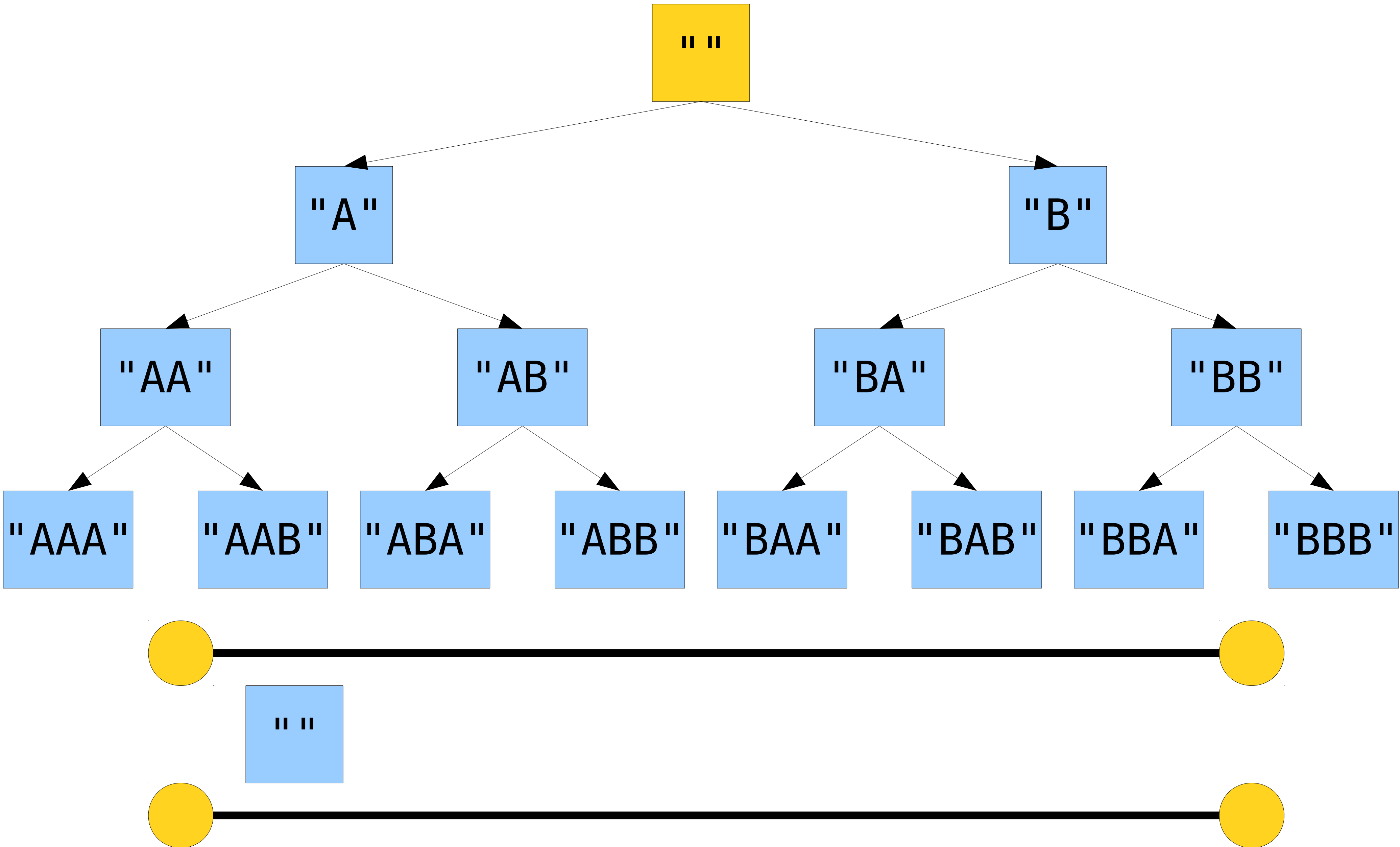
Memory Usage



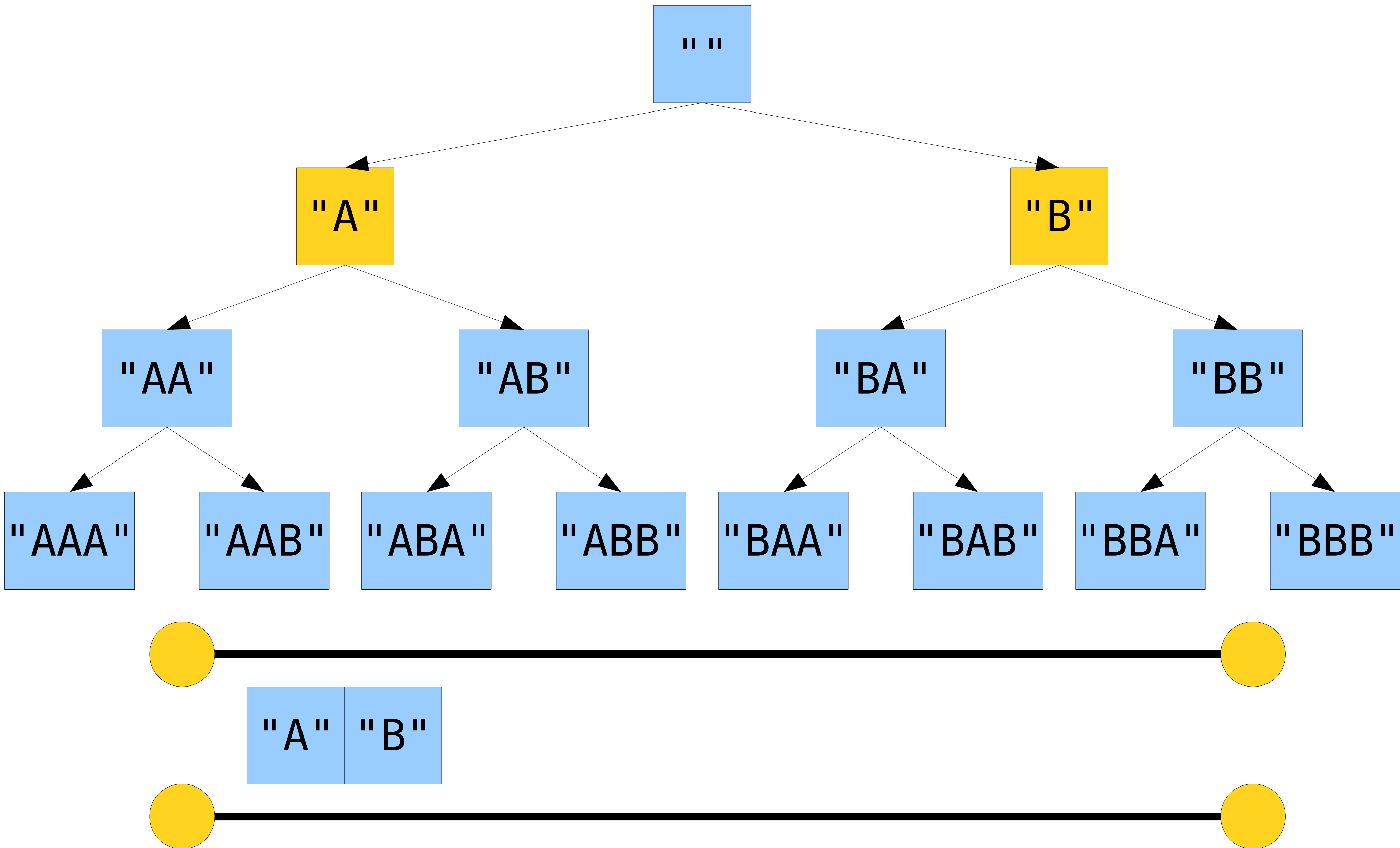
Memory Usage



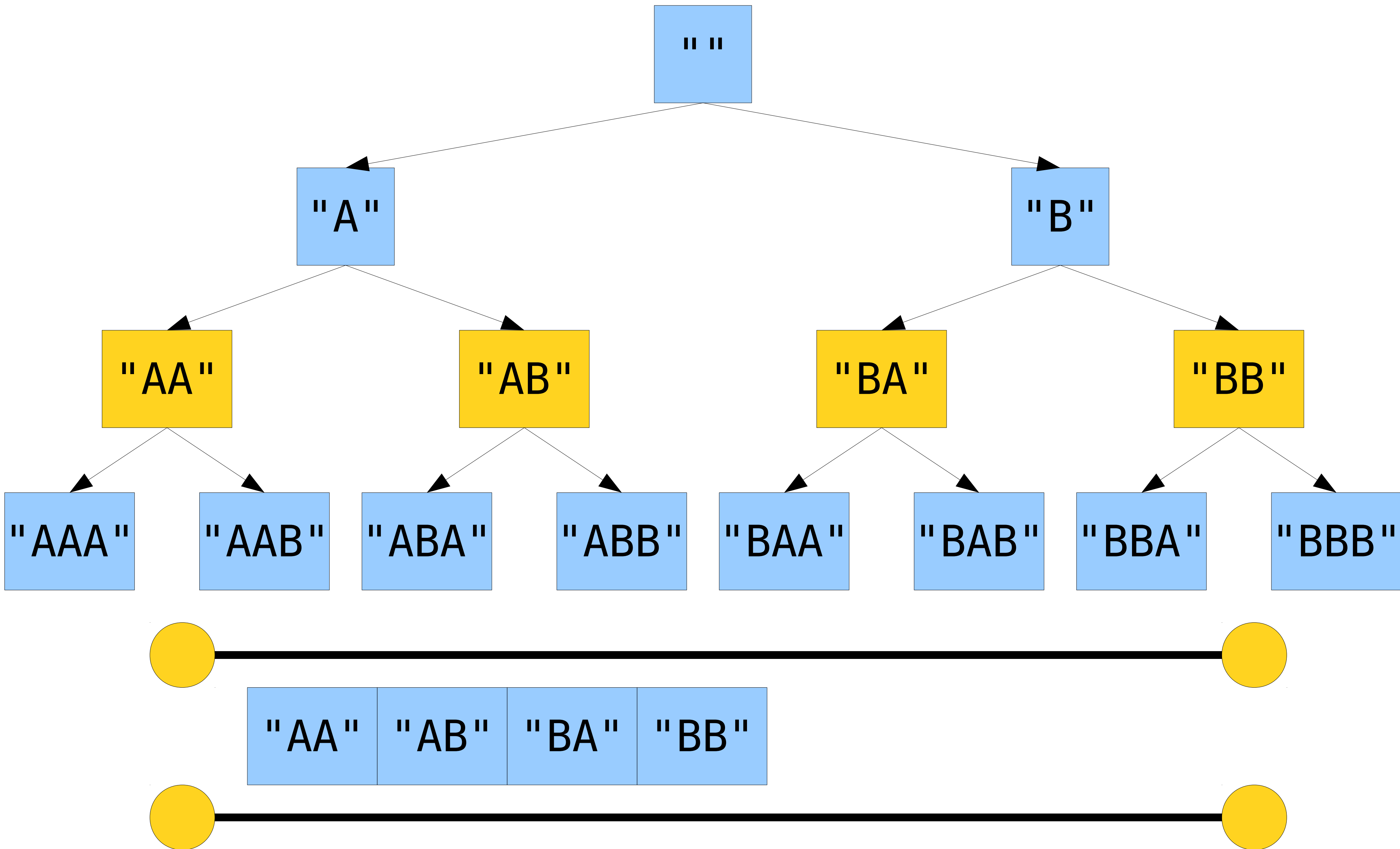
Memory Usage



Memory Usage



Memory Usage



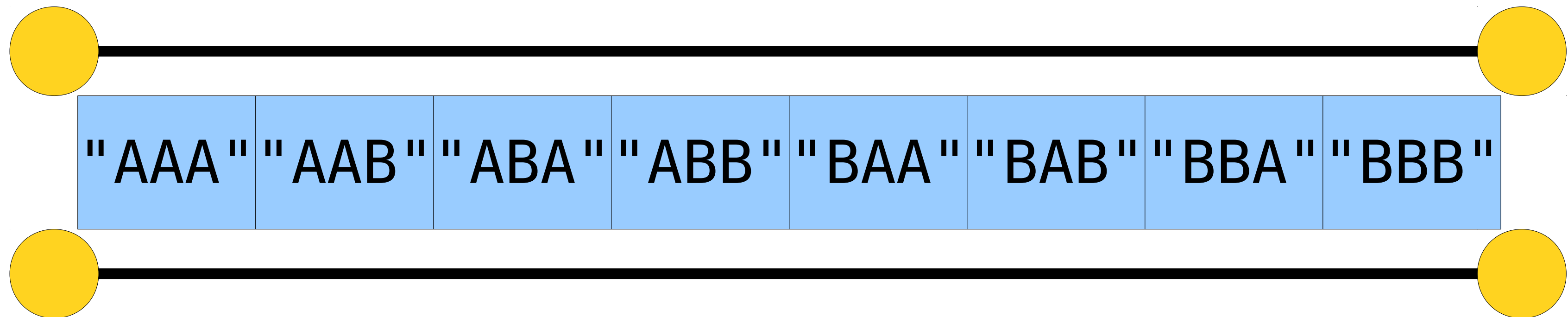
Memory Usage

""

"A" "B"

"AA" "AB" "BA" "BB"

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

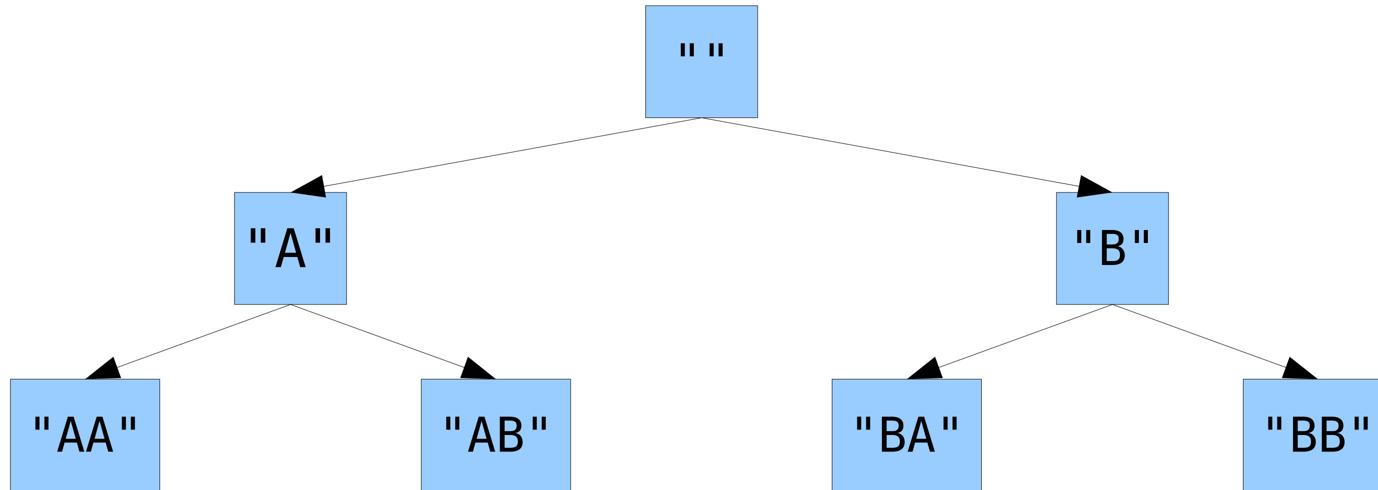


Memory Usage

- Currently, to generate all strings of lengths 0, 1, 2, …, *n*, we might end up having to hold roughly 26^n strings in memory.
- The size of a short string (on my machine, at least) is 32 bytes.
- This means that trying to list all strings of length seven should consume basically all free resources on my computer.
- Watch what happens when we try that!

Changing our Data Structure

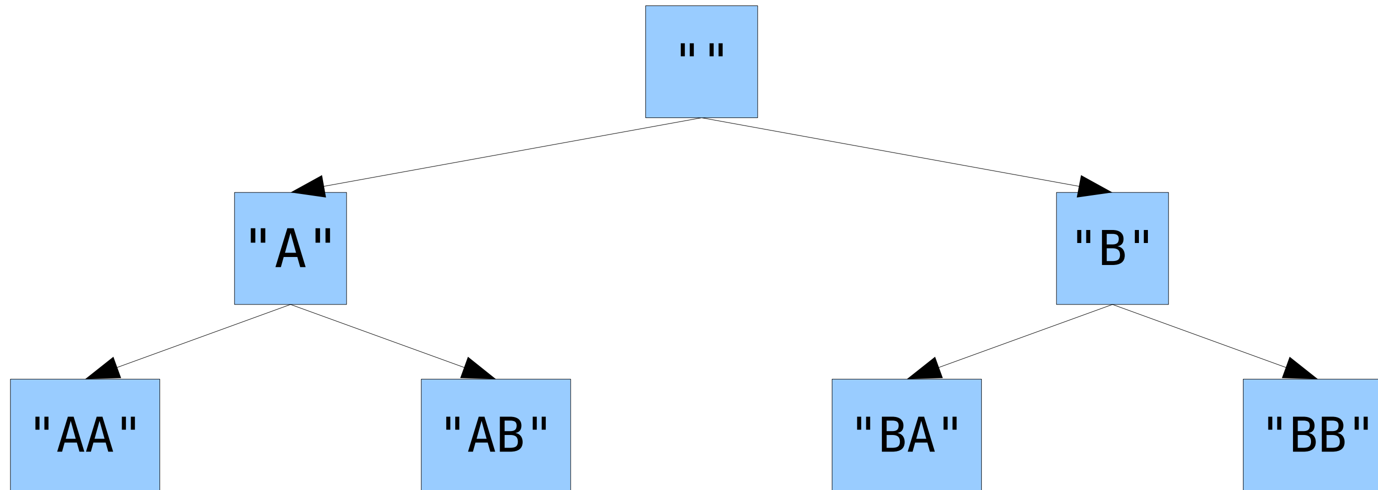
Back to the Stack




```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

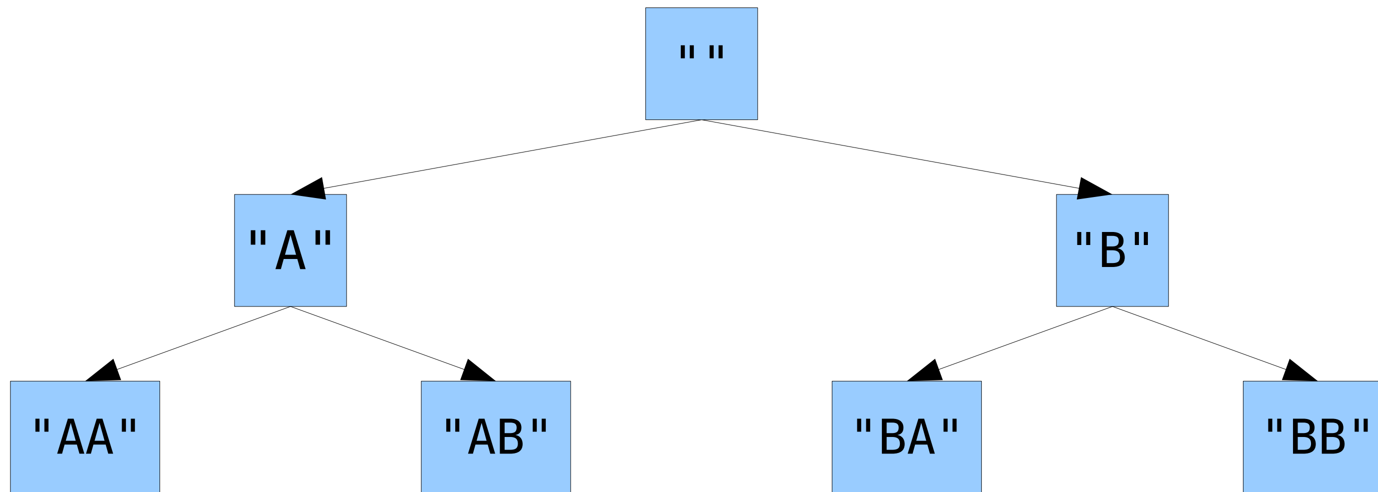
Back to the Stack

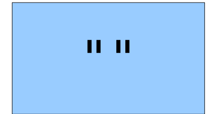



```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
        add that string
      }
   }
}
```

Back to the Stack

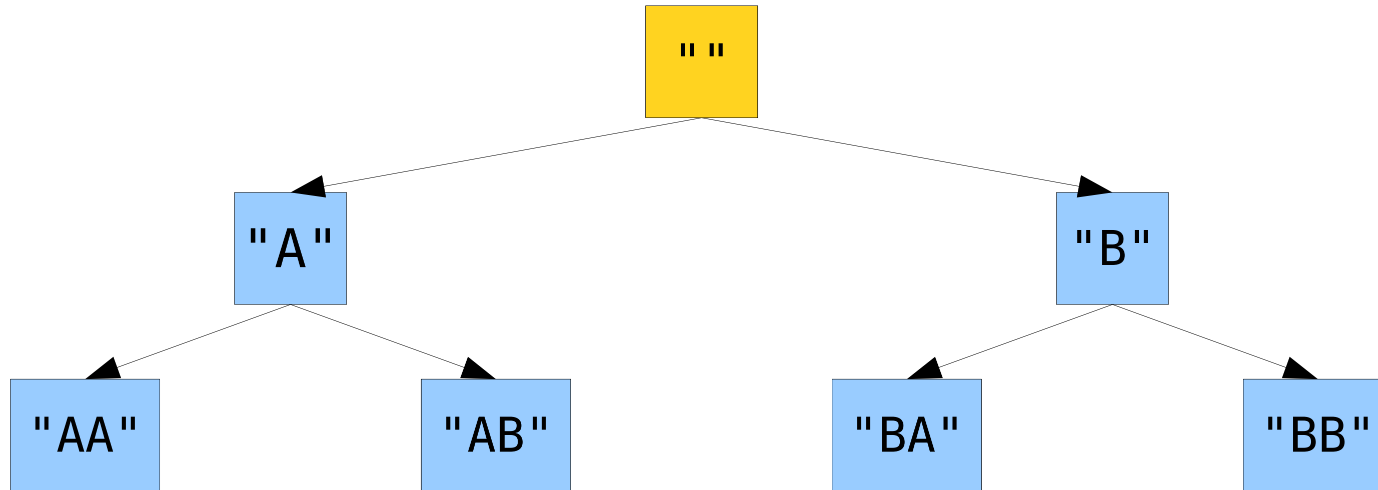




```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
        add that string
      }
   }
}
```

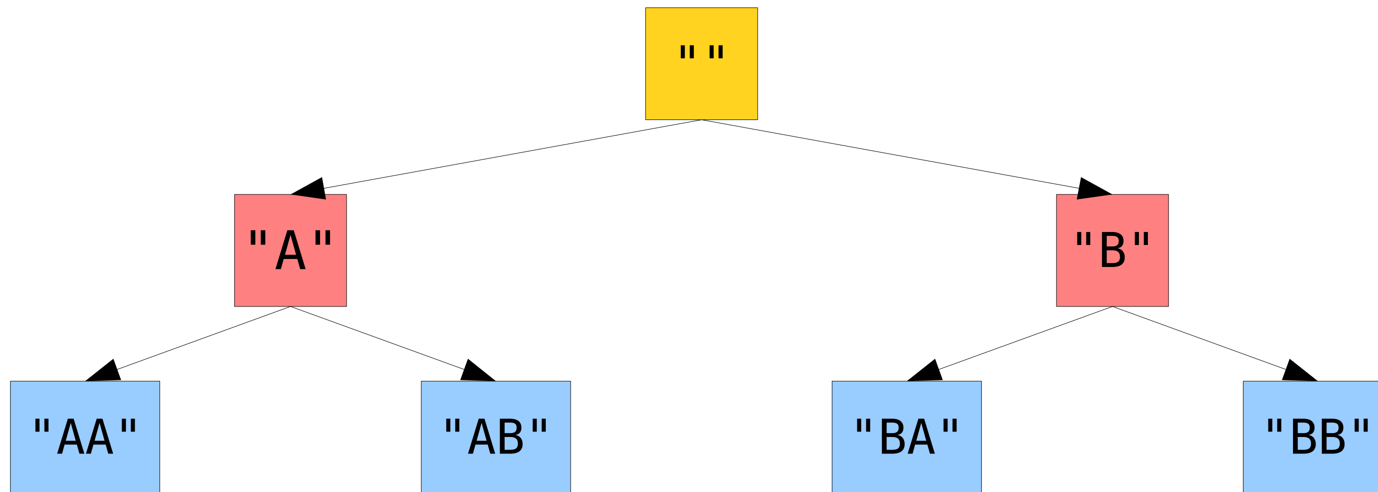
Back to the Stack

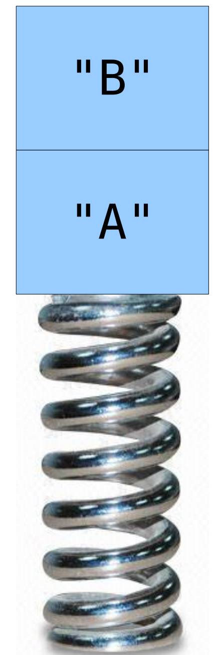



```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
        add that string
      }
   }
}
```

Back to the Stack

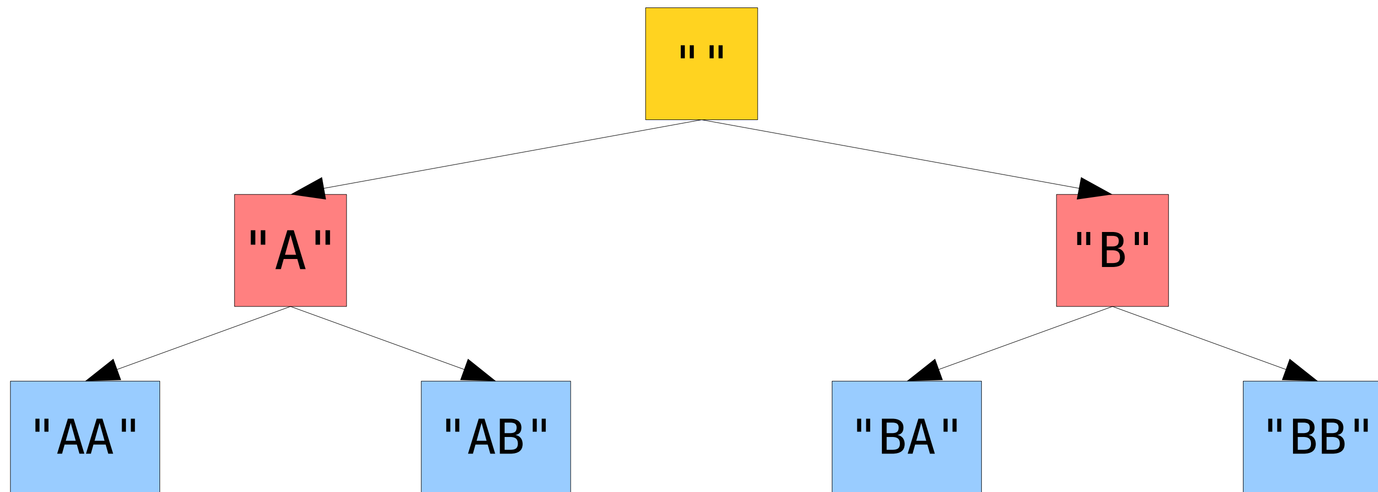





```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

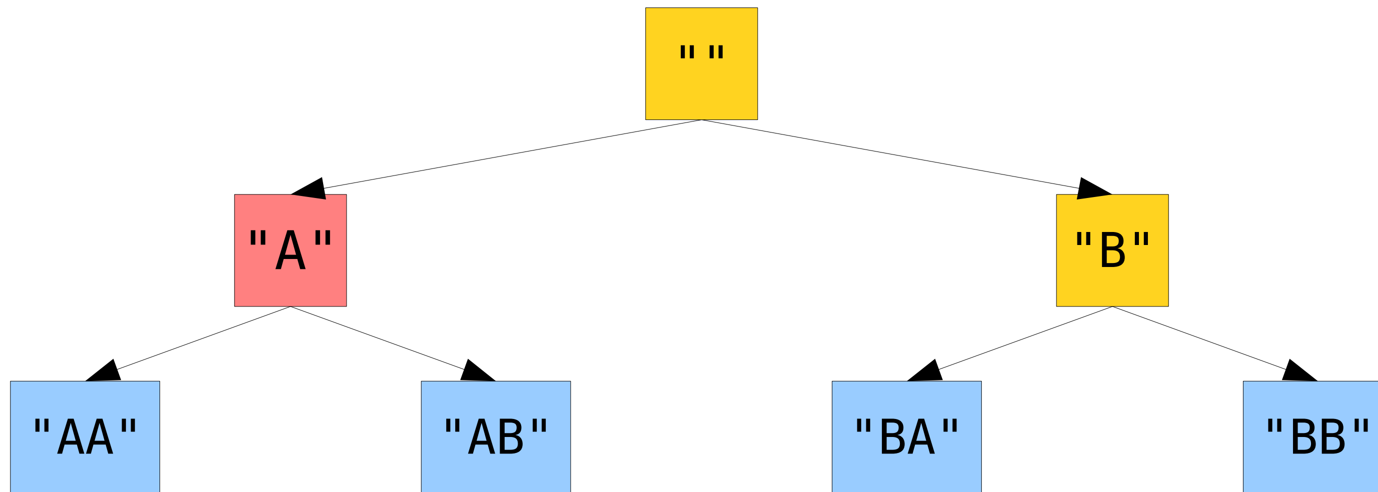
Back to the Stack




```
add the empty string
while (data structure not empty) {
    string curr = get next string;
    process curr;

    if (curr isn't too long) {
        for (each next possible string) {
            add that string
        }
    }
}
```

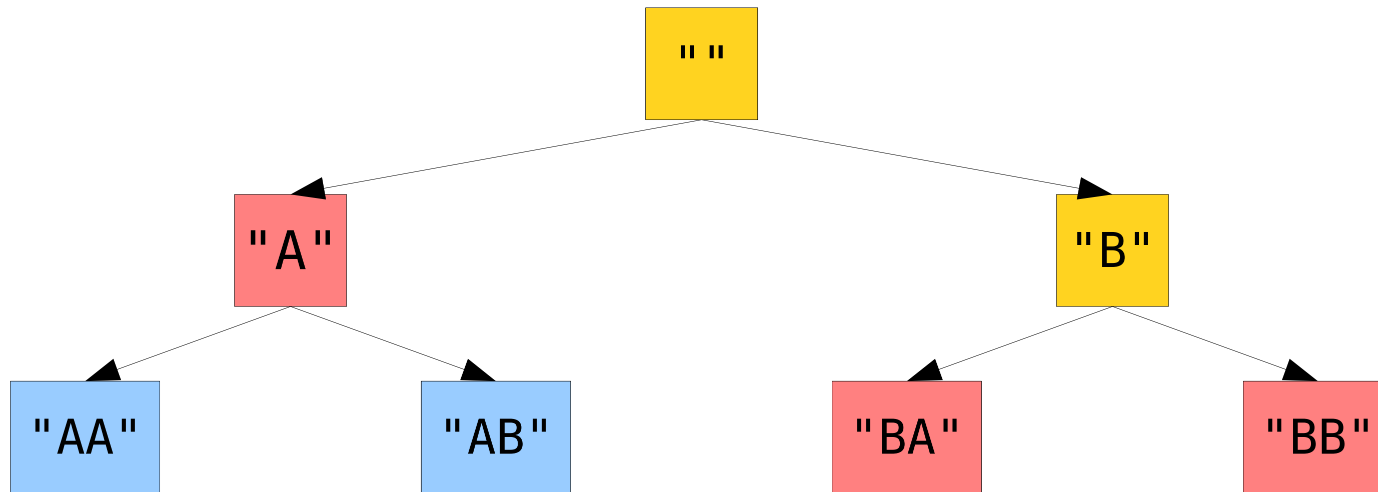
Back to the Stack

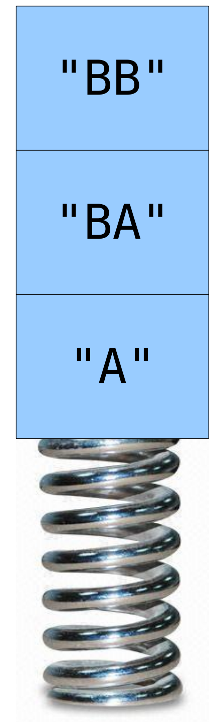



```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

Back to the Stack

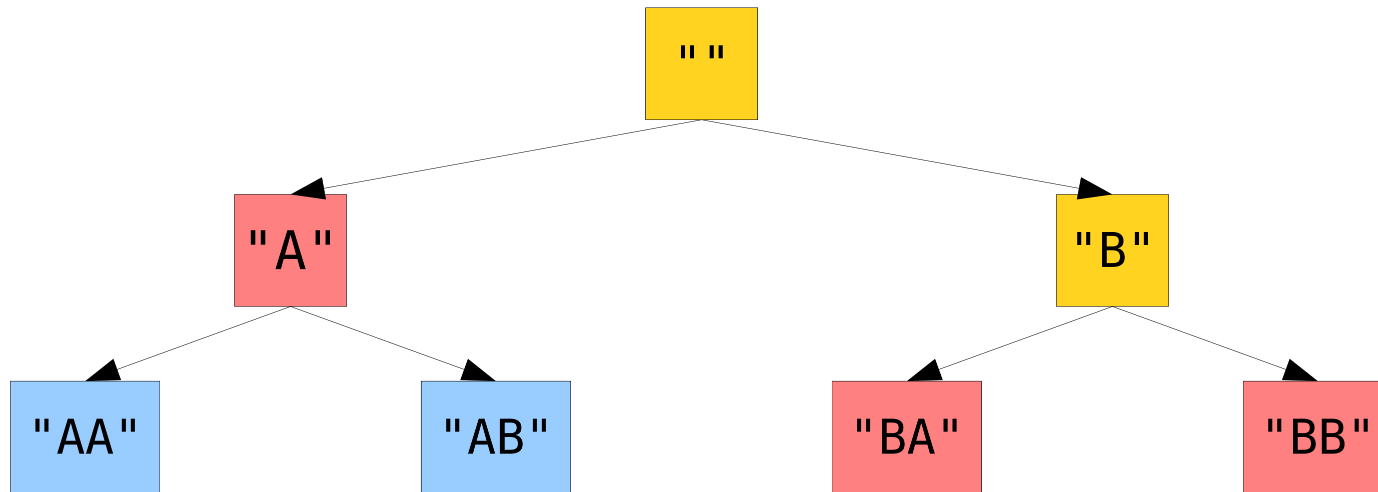





```
add the empty string
while (data structure not empty) {
    string curr = get next string;
    process curr;

    if (curr isn't too long) {
        for (each next possible string) {
            add that string
        }
    }
}
```

Back to the Stack



"BB"

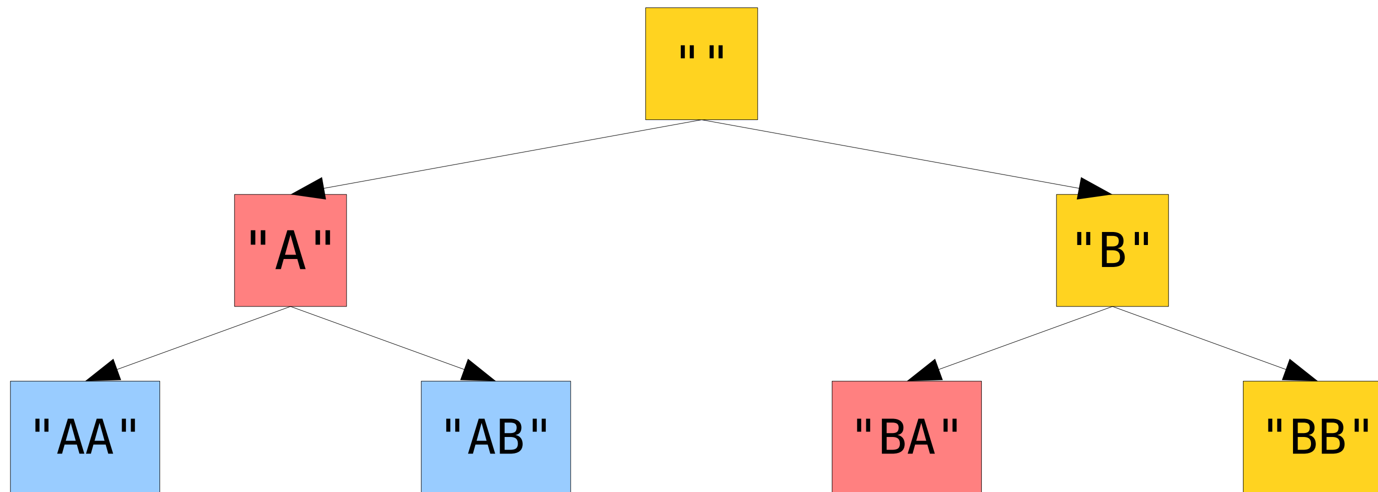
"BA"

"A"

```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

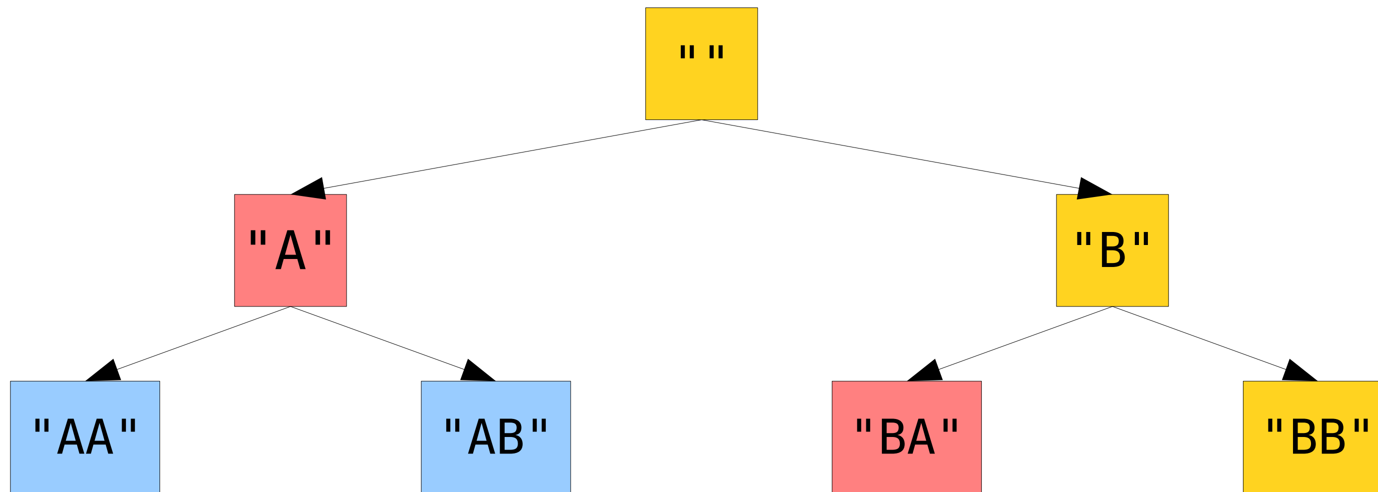
Back to the Stack




```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

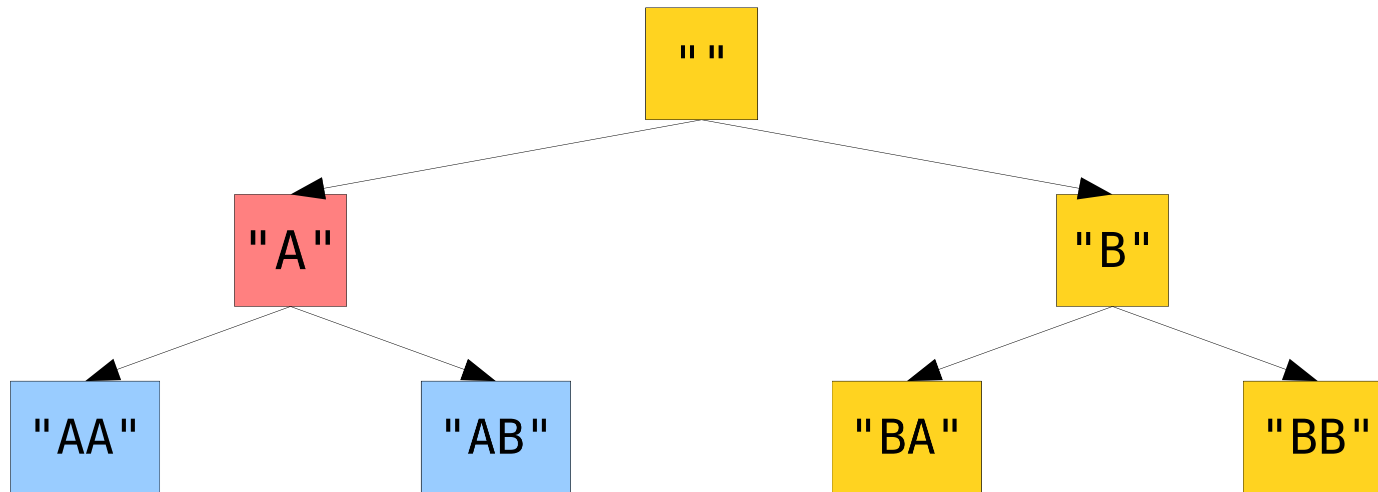
Back to the Stack



```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

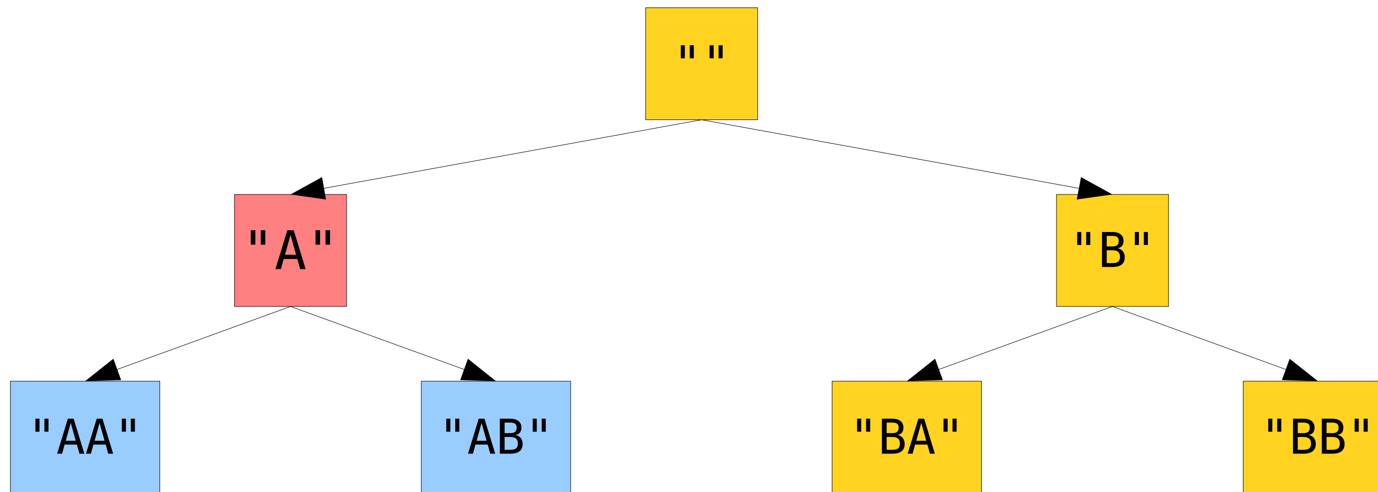
Back to the Stack




```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
        add that string
      }
   }
}
```

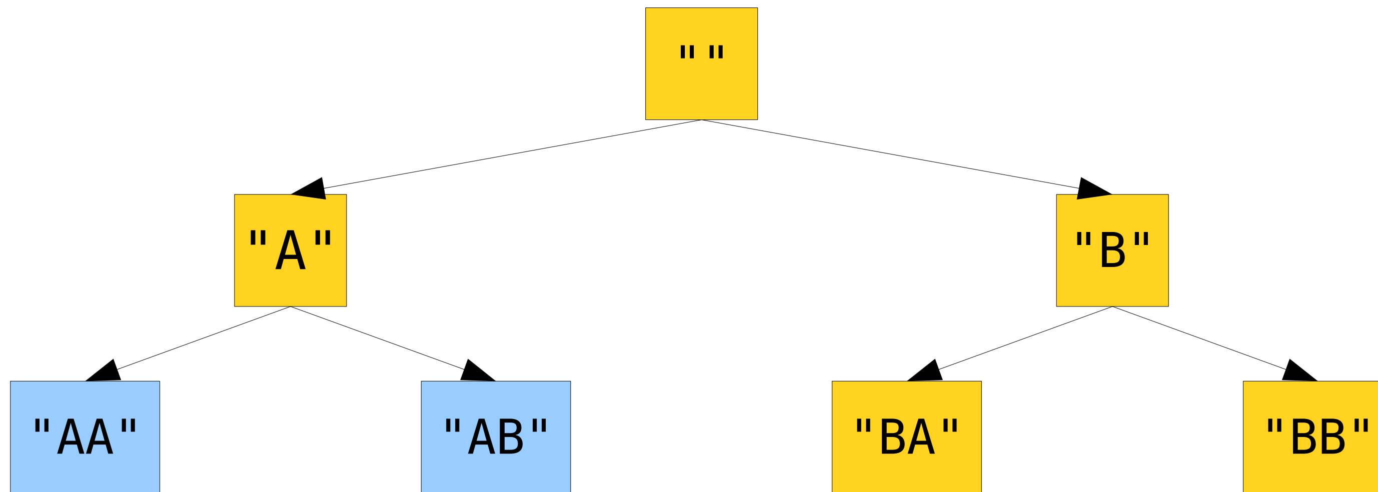
Back to the Stack




```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
        add that string
      }
   }
}
```

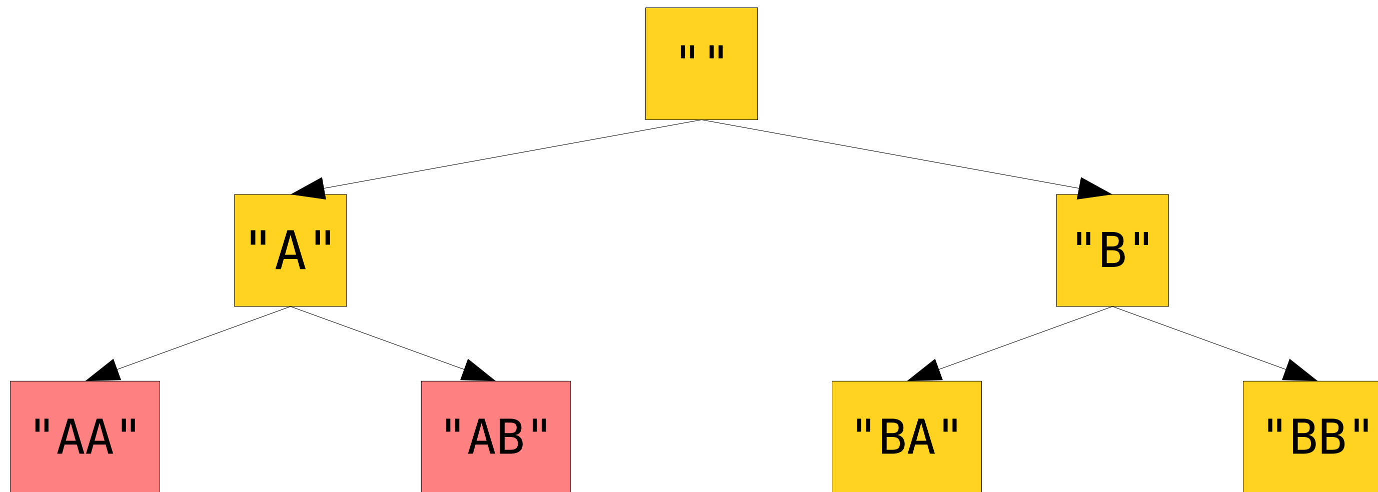
Back to the Stack



```
add the empty string
while (data structure not empty) {
    string curr = get next string;
    process curr;

    if (curr isn't too long) {
        for (each next possible string) {
            add that string
        }
    }
}
```

Back to the Stack



```
add the empty string
while (data structure not empty) {
   string curr = get next string;
   process curr;

   if (curr isn't too long) {
      for (each next possible string) {
         add that string
      }
   }
}
```

Let’s Code it Up!

Depth-First Search

- This algorithm is called ***depth-first search*** and shows up everywhere in computer science:
  - Checking resiliency of road networks to closures and outages (*2-edge connectivity*)
  - Ordering tasks while satisfying prerequisites (*topological sorting*)
- ***Claim:*** The memory used by this algorithm is dramatically lower than breadth-first search.
- Curious? Stay tuned in CS106B, and take a look at CS161!

A Comparison

- ***Breadth-first search*** (use a `Queue`):
  - Time: $\approx 26^n$
  - Space: $\approx 26^n$
- Lists everything in increasing order of length.
- But it's a memory hog!

- ***Depth-first search*** (use a `Stack`)
  - Time: $\approx 26^n$
  - Space: $\approx n$
- Very memory-efficient.
- But doesn't list things in increasing size order.

Your Action Items

- Read Chapter 5 of the textbook, which talks about container classes.
- Keep working on Assignment 1. Aim to start working on that last part soon.
- Read the style guide up on the course website for more information about good programming style.

Next Time

- ***The `Vector` Type***
  - How do we store and manipulate sequences in C++?