CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 34 Apr 12, 2013

Finite State Machines

Lecture Goals

1. Dijkstra Heartburn
2. Visual Studio
3. FSMs

Upcoming Homework Assignment

HW #9 **Due: May?**

Dijkstra

I removed the lateness cap. So you can turn this in whenever. Just don't put it off too long.

Upcoming Homework Assignment

HW #10 **Due: Friday, Apr 19**

Finite State Machines

This is all about making a finite state machine. There is absolutely zero chance that I will push this next one's deadline.

Questions about Dijkstra?

Is it the *algorithm?*

Or is it *C++ classes?*

Visual Studio and IDEs

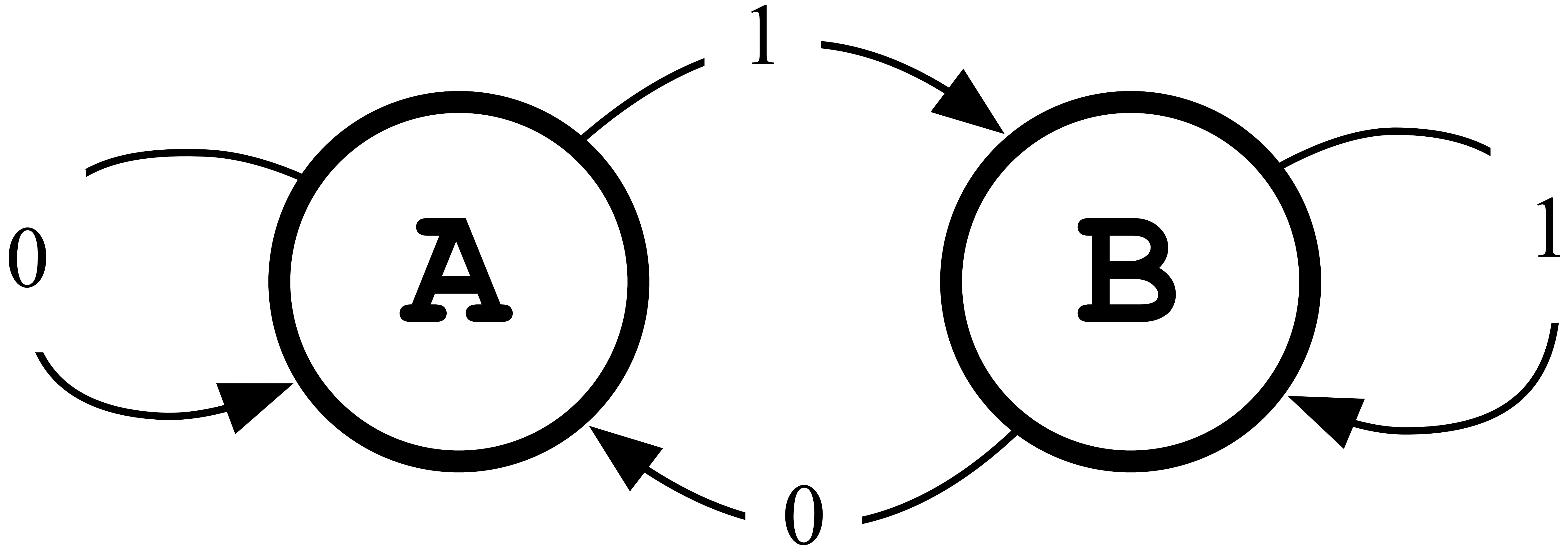
With your host, professor Alec...

Finite State Machines!

A Finite State Machine is an amazingly cool little thing that can do a lot of stuff:

* Model user interface widgets
* Model robot behavior
* Parse programming languages
* Search for regular expressions
* Model digital communication

What they look like

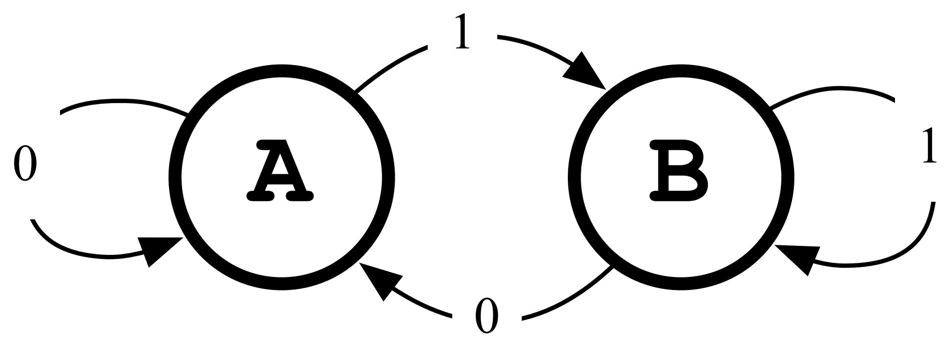

$$\begin{tikzcd} A \arrow[loop left, "0"] \arrow[r, bend left, "1"] & B \arrow[loop right, "1"] \arrow[l, bend left, "0"] \end{tikzcd}$$

Anatomy of a FSM

State: a node representing the machine's current status. A FSM *can only be in one state at a time*.

Transition: a directed edge indicating a change from one state to another. *The destination state might actually be the same we started in.*

Event: a *signal* that causes the FSM to follow a transition, depending on what state it is in.


$$\begin{tikzcd} A \arrow[loop left, "0"] \arrow[r, bend left, "1"] & B \arrow[loop right, "1"] \arrow[l, bend left, "0"] \end{tikzcd}$$

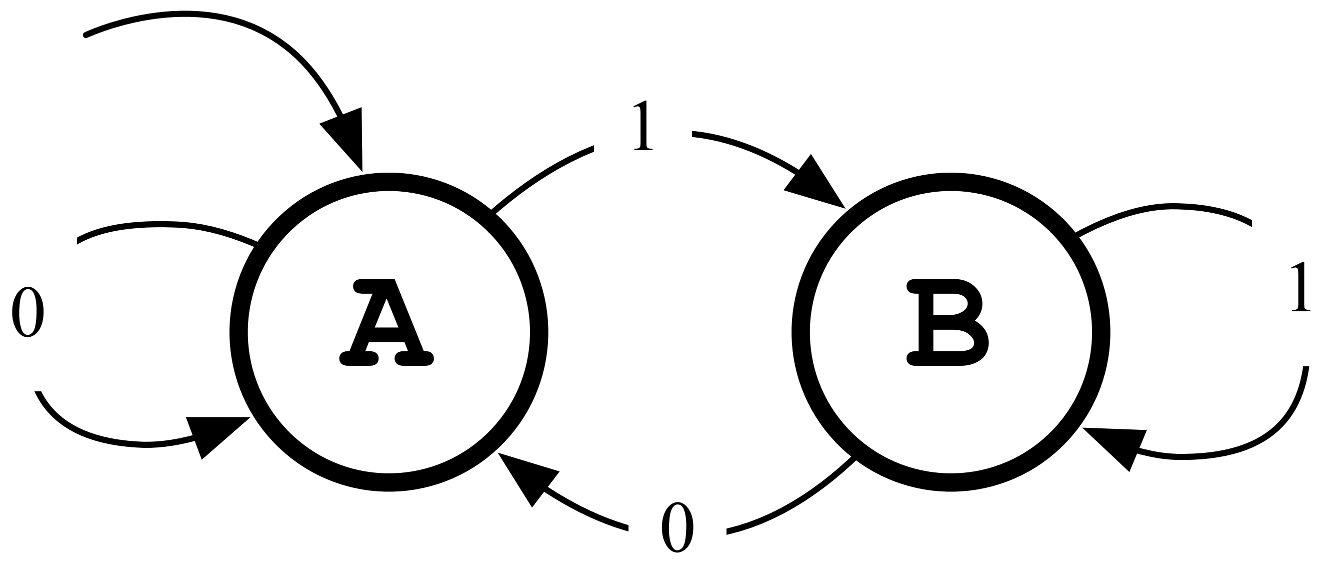
Anatomy of a FSM

State: A and B are states.

Transition: The lines with arrows are transitions.

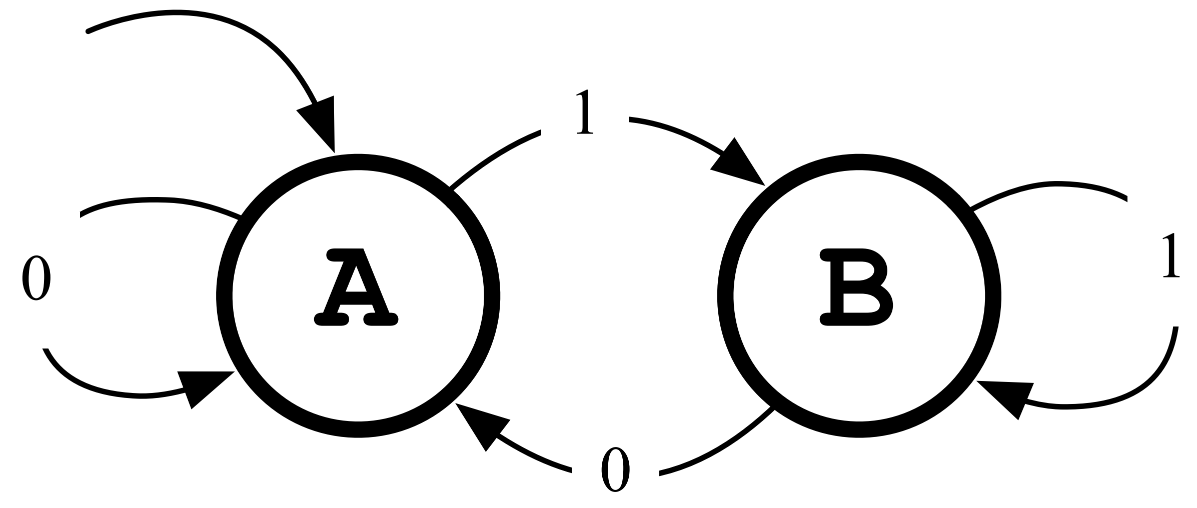
Event: This FSM can receive events *0* and *1*.

Default State: An FSM also has a default state that is represented by an arrow from nowhere.



FSMs: How do they work

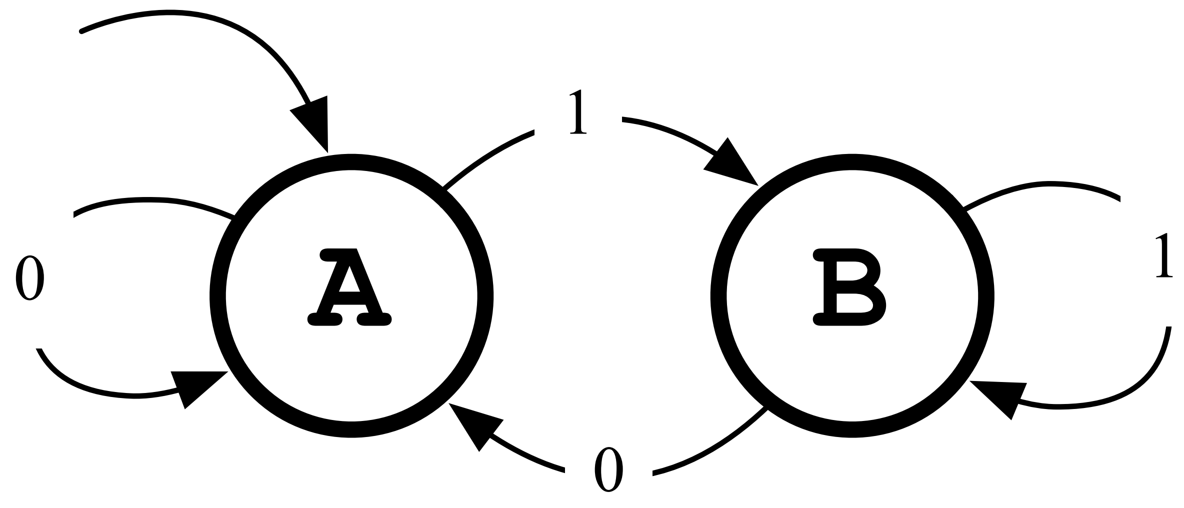
State	Signal	Next


$$\begin{tikzcd} \arrow[r] & A \arrow[loop left, "0"] \arrow[r, bend left, "1"] & B \arrow[loop right, "1"] \arrow[l, bend left, "0"] \end{tikzcd}$$

Say we use this simple FSM to parse the bitstring “001011101”.

FSMs: How do they work

State	Signal	Next
A		

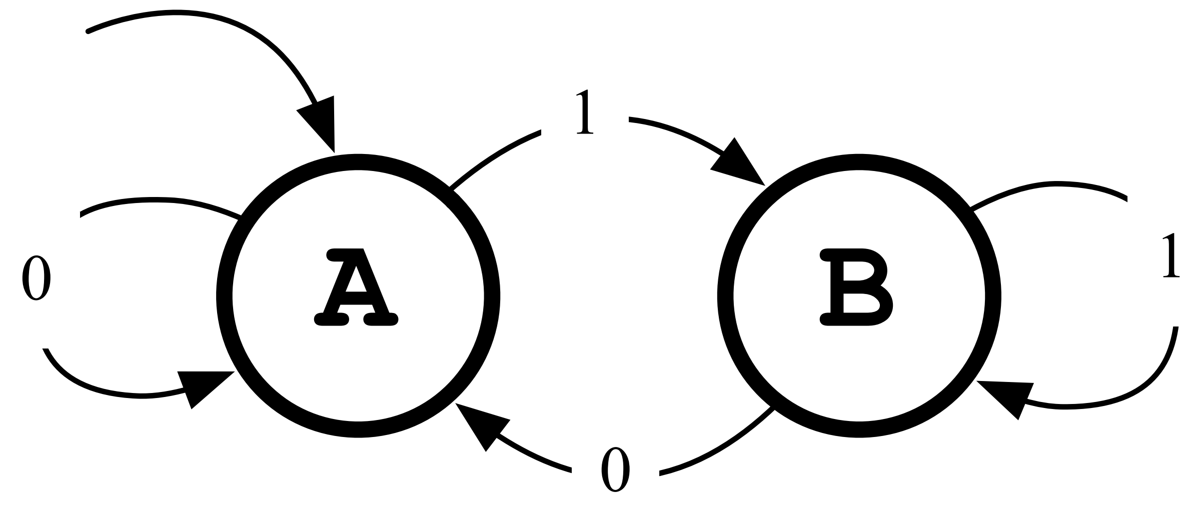


We start out in state A because that is the default state (arrow from nowhere).

`001011101`

FSMs: How do they work

State	Signal	Next
A	0	

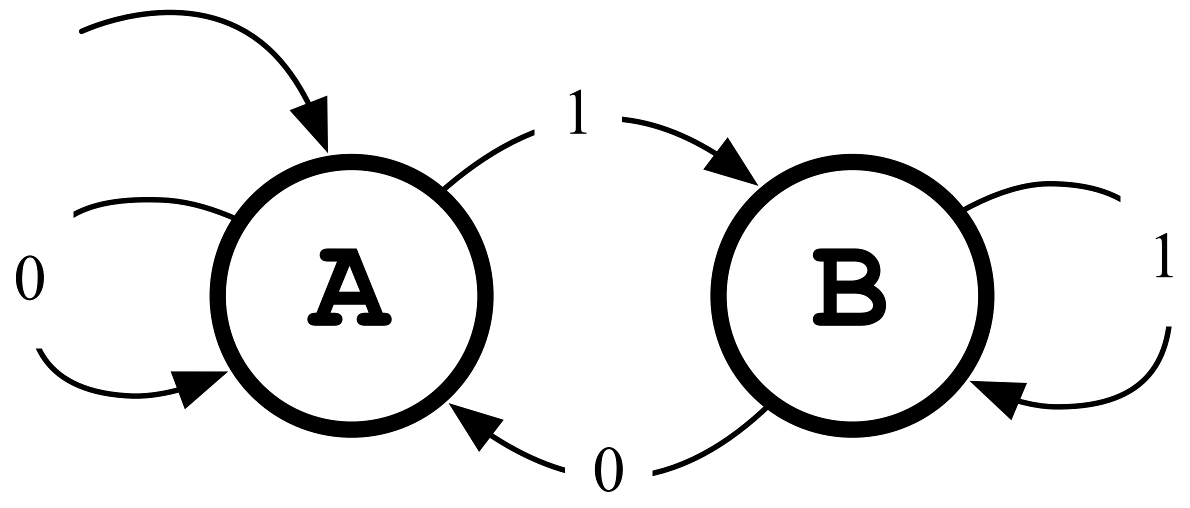


First signal is 0. Current state (A) has two possible transitions: one for 0, another for 1.

001011101

FSMs: How do they work

State	Signal	Next
A	0	A

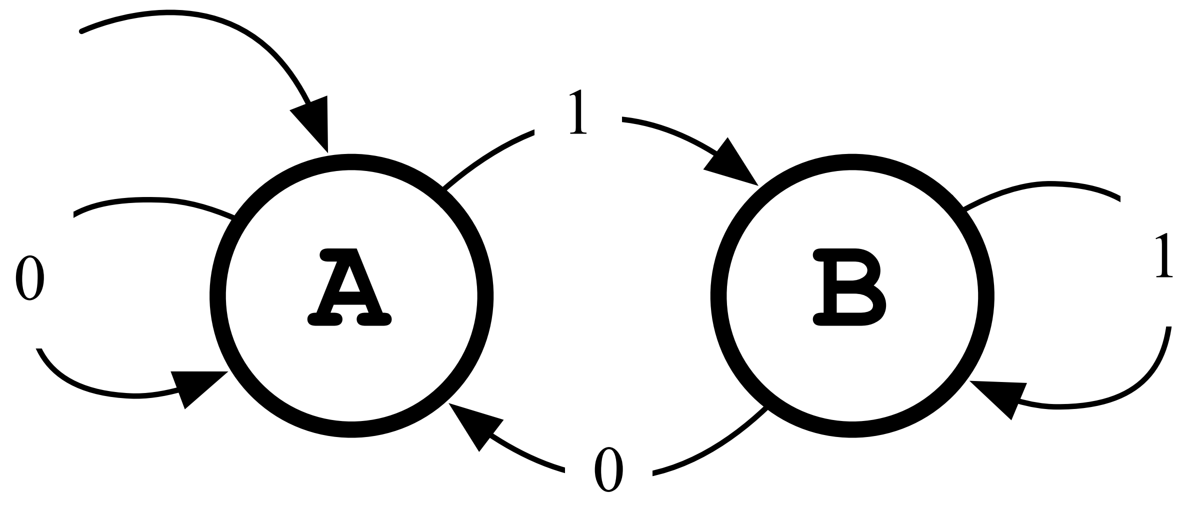


The transition for signal 0 loops back on ourself, so the next state is also A.

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A

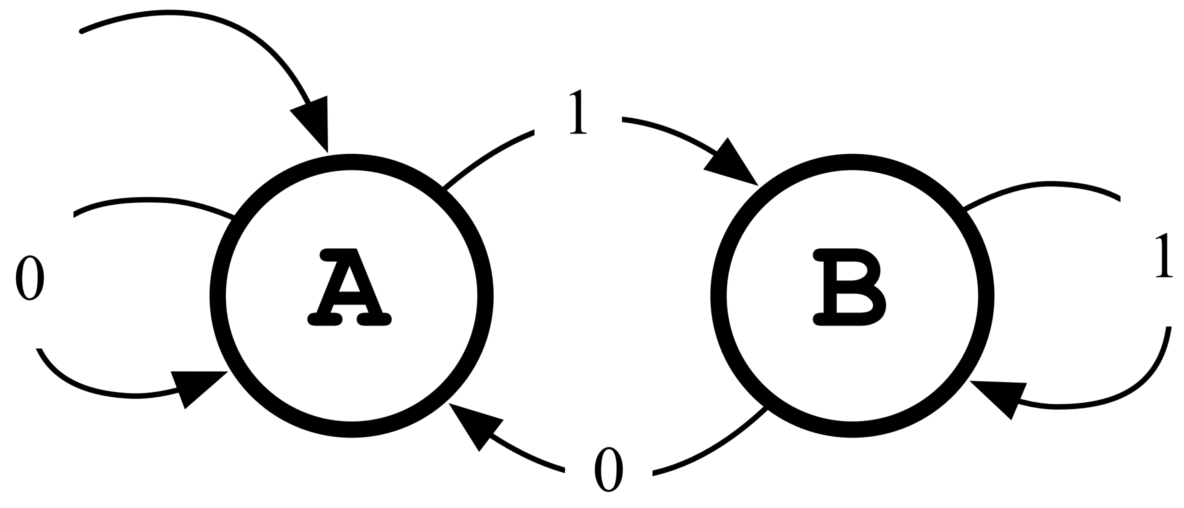


This continues for a while.

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B

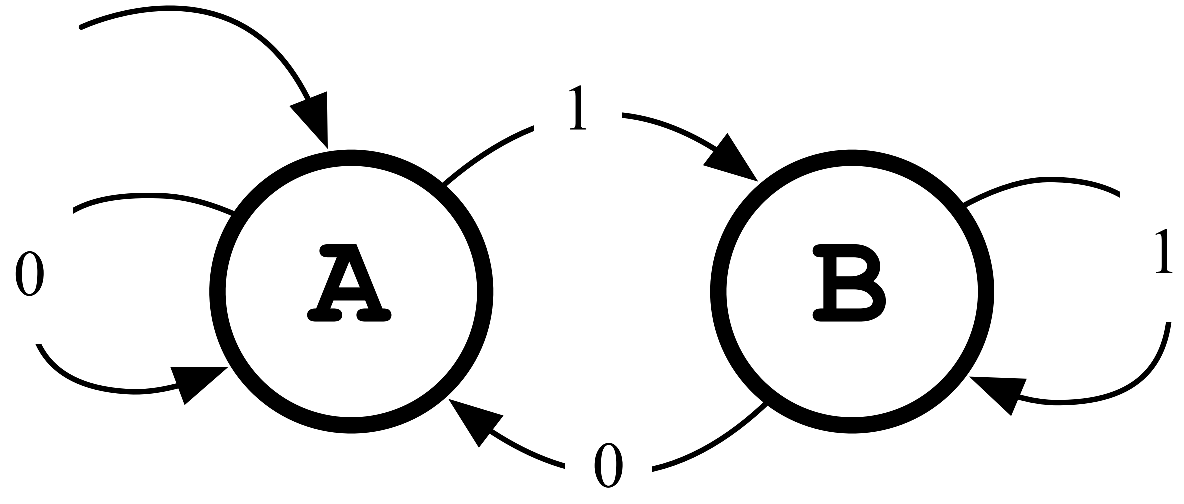


Now in state B!

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B
B	0	A

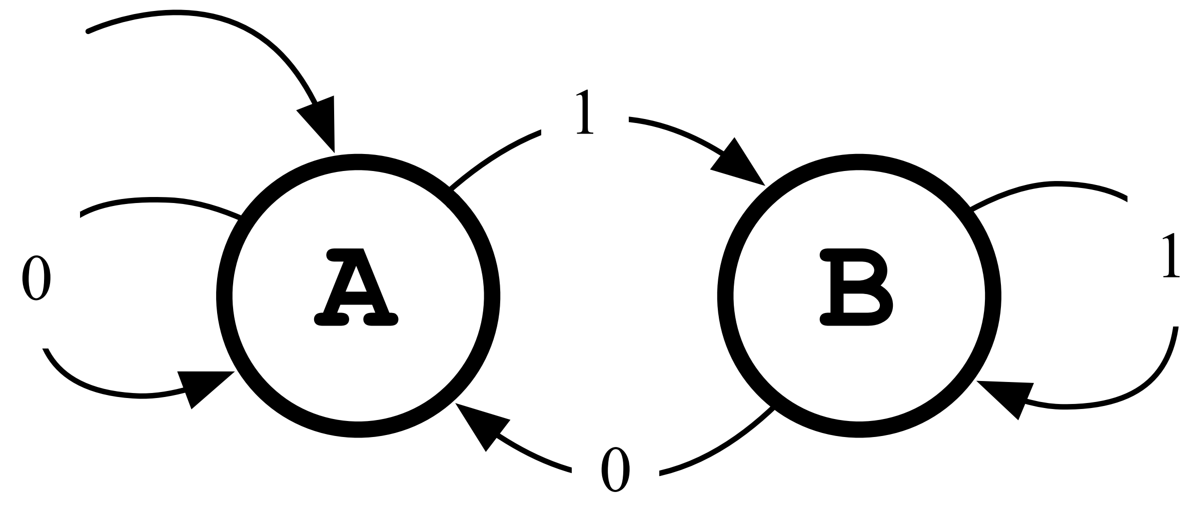


...

001**0**11101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B
B	0	A
A	1	B

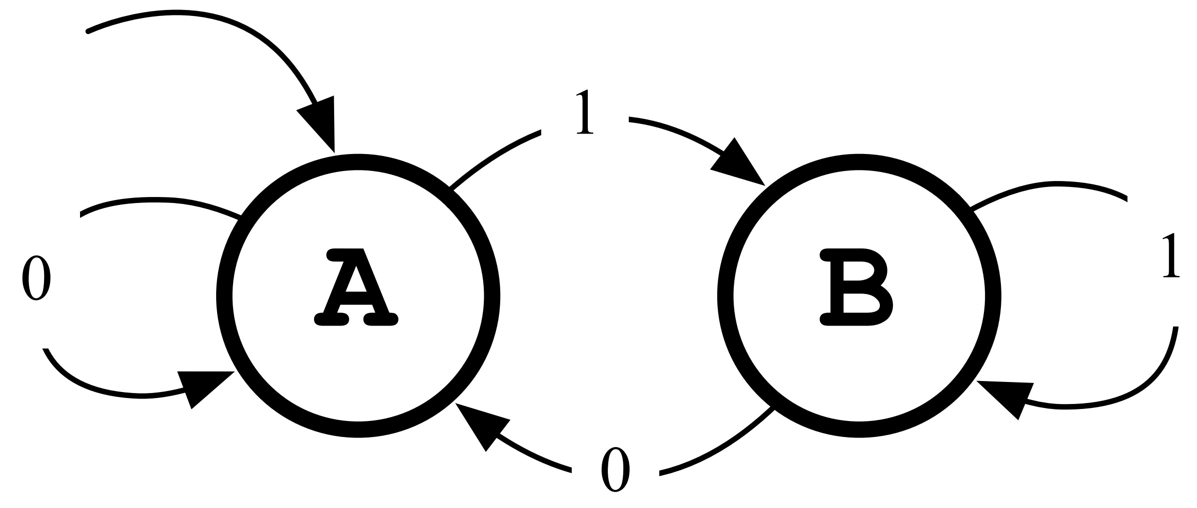


...

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B
B	0	A
A	1	B
B	1	B

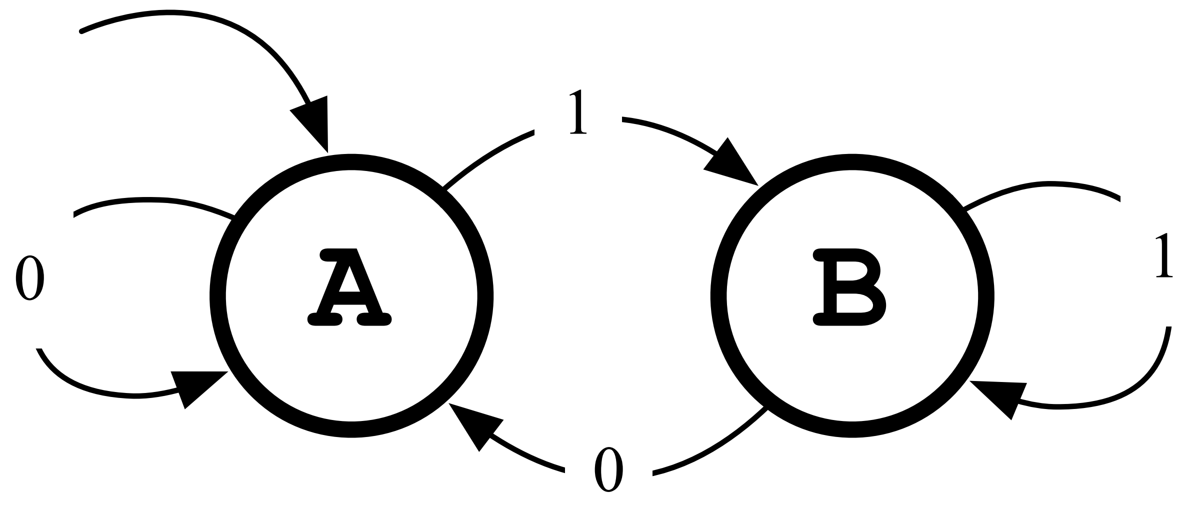


...

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B
B	0	A
A	1	B
B	1	B
B	1	B

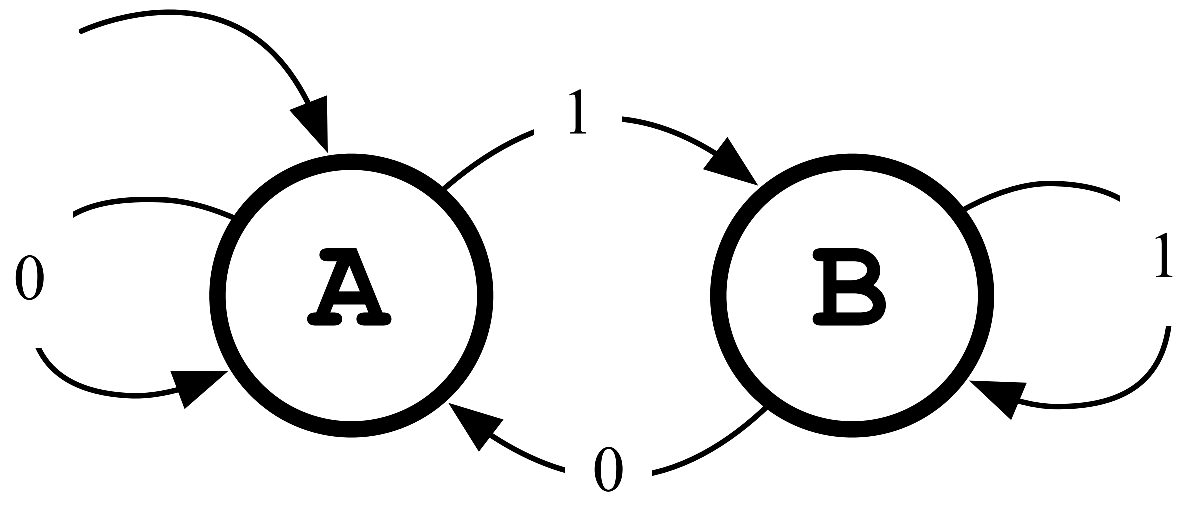


...

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B
B	0	A
A	1	B
B	1	B
B	1	B
B	0	A

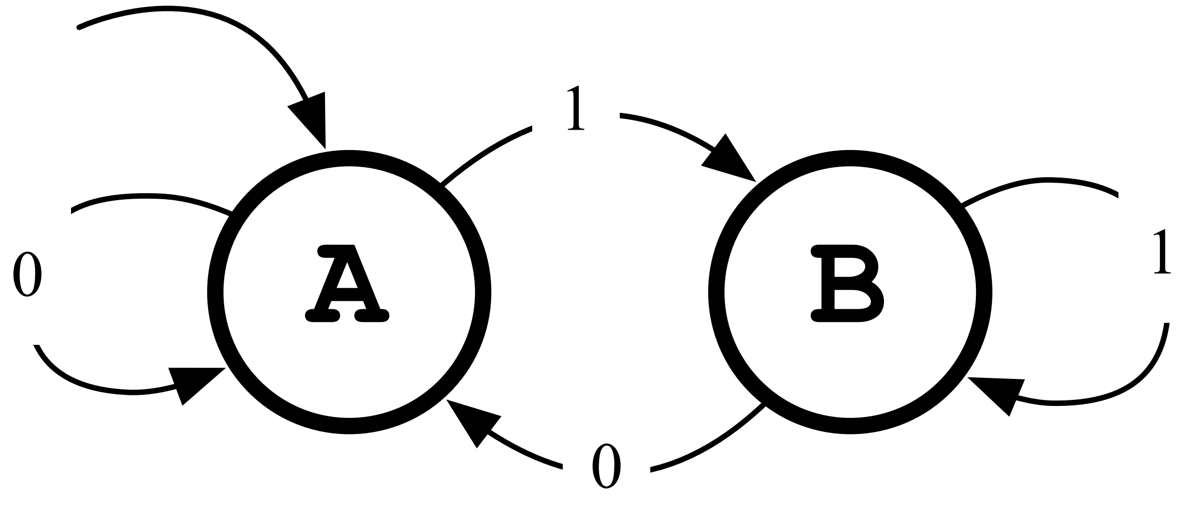


...

001011101

FSMs: How do they work

State	Signal	Next
A	0	A
A	0	A
A	1	B
B	0	A
A	1	B
B	1	B
B	1	B
B	0	A
A	1	B


$$\begin{tikzcd} \arrow[r] & A \arrow[loop left, "0"] \arrow[r, bend left, "1"] & B \arrow[l, bend left, "0"] \arrow[loop right, "1"] \end{tikzcd}$$

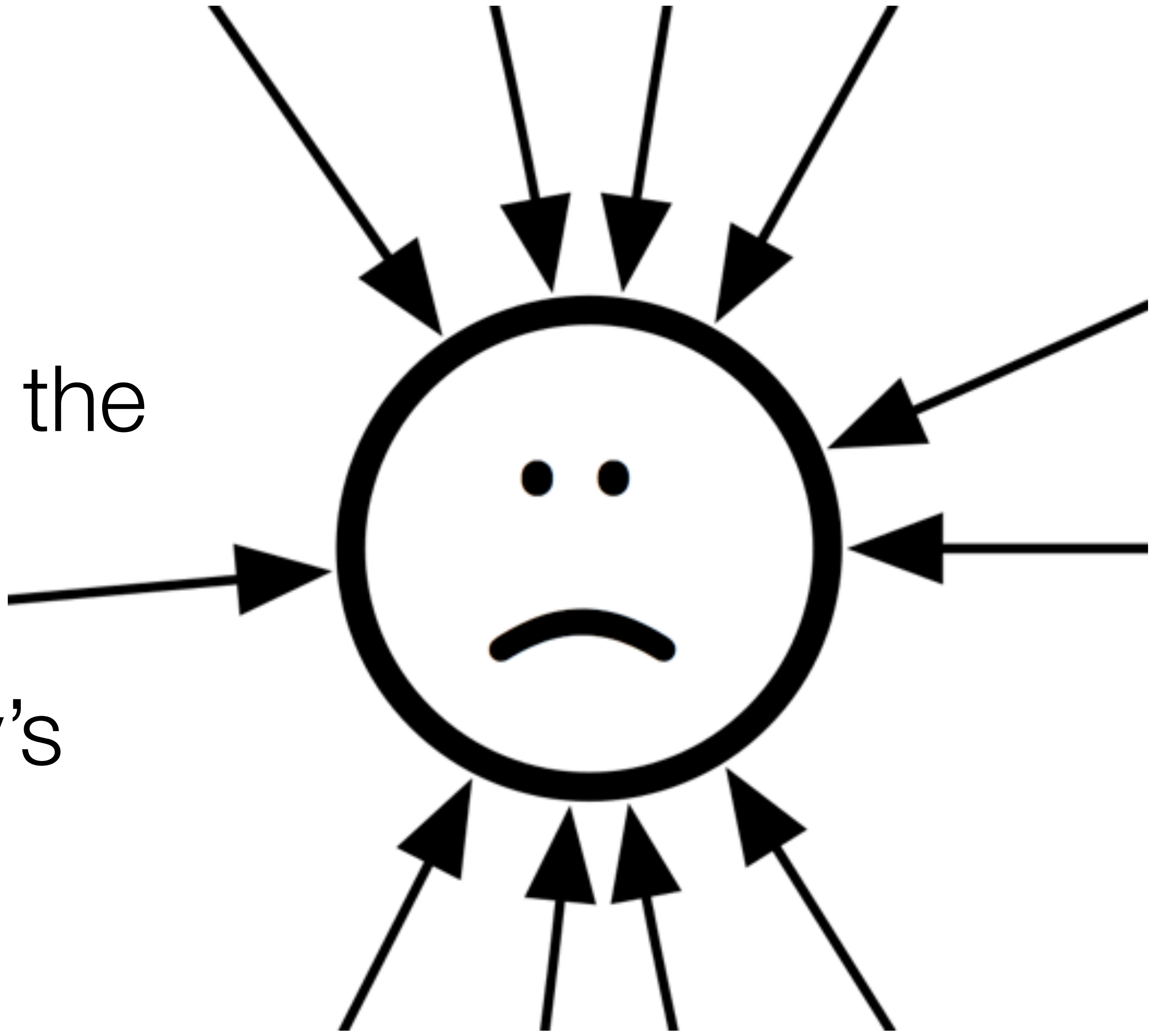
At the end of this bitstring we are in state B. So what's the FSM doing for us?

001011101

Longer Examples

Check out FSM.pdf in the HW directory.

Find out what this guy’s deal is.



FSM Implementation

```
class FSM {
private:
  vector<State*> states;
  vector<Transition*> transitions;
  int state;
  int default_state;
public:
  // next slide
};
```

FSM Implementation

```
class FSM {
private:
  // previous slide
public:
   FSM();
   int addState(string label, bool is_accept_state);
   int addState(string label);
   int addTransition(int stateA, int stateB,
   int signal, string transLabel);
   int countStates();
   int countTransitions();
   int getCurrentState();
   bool isAcceptState();
   State* getState(int id);
   Transition* getTransition(int id);
   int getDefaultState();
   void setState(int id);
   bool handleSignal(int signal);
};
```

1 or 2 line function defs

```
class FSM {
private:
  // previous slide
public:
   FSM();
   int addState(string label, bool is_accept_state);
   int addState(string label);
   int addTransition( int stateA, int stateB,
                      int signal, string transLabel);
   int countStates();
   int countTransitions();
   int getCurrentState();
   bool isAcceptState();
   State* getState(int id);
   Transition* getTransition(int id);
   int getDefaultState();
   void setState(int id);
   bool handleSignal(int signal);
};
```

2 < lines < 10

```
class FSM {
private:
  // previous slide
public:
   FSM();
   int addState(string label, bool is_accept_state);
   int addState(string label);
   int addTransition( int stateA, int stateB,
                      int signal, string transLabel);
   int countStates();
   int countTransitions();
   int getCurrentState();
   bool isAcceptState();
   State* getState(int id);
   Transition* getTransition(int id);
   int getDefaultState();
   void setState(int id);
   bool handleSignal(int signal);
};
```

Trickier Functions

```
class FSM {
private:
  // previous slide
public:
   FSM();
   int addState(string label, bool is_accept_state);
   int addState(string label);
   int addTransition( int stateA, int stateB,
                      int signal, string transLabel);
   int countStates();
   int countTransitions();
   int getCurrentState();
   bool isAcceptState();
   State* getState(int id);
   Transition* getTransition(int id);
   int getDefaultState();
   void setState(int id);
   bool handleSignal(int signal);
};
```

Read the Header File

There’s no recursion or pointer work or any of that here. Everything is described in the header file, including what valid data ranges are (e.g. IDs are non-negative), and how you should handle various edge cases.

For those of you who need more exposure to writing objects, this is your garden-variety class.