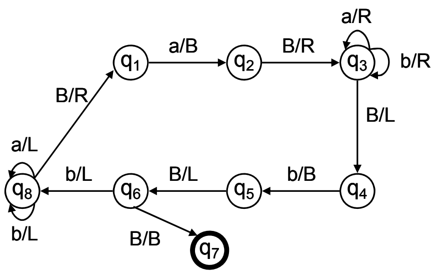
Languages Accepted by Turing Machines

Example 2:

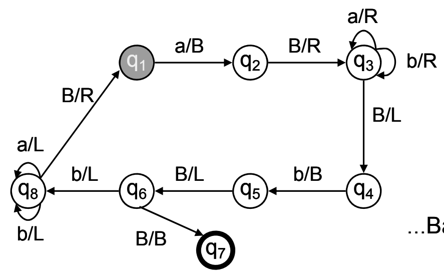
Language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

Language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.

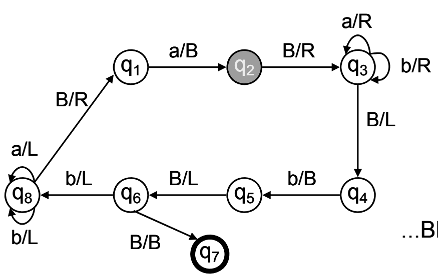


…BaabbB…
↑

Languages Accepted by Turing Machines

Example 2:

Language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.

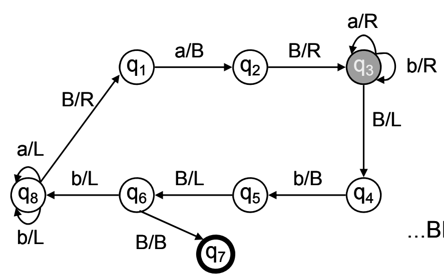


…BBabbB…
↑

Languages Accepted by Turing Machines

Example 2:

Language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.

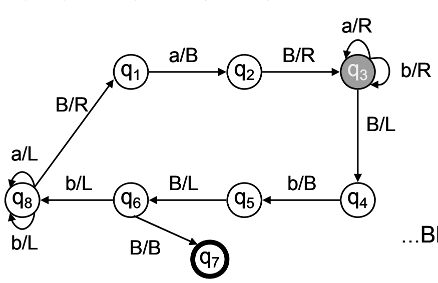


…BBabbB…
↑

Languages Accepted by Turing Machines

Example 2:

Language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.

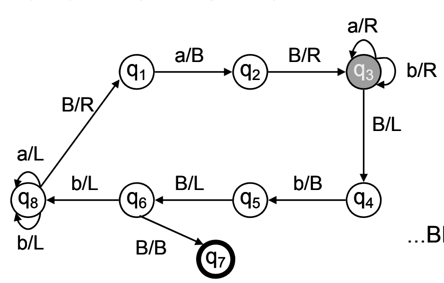


…BBabbB…
↑

Languages Accepted by Turing Machines

Example 2:

Language $L = \{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.

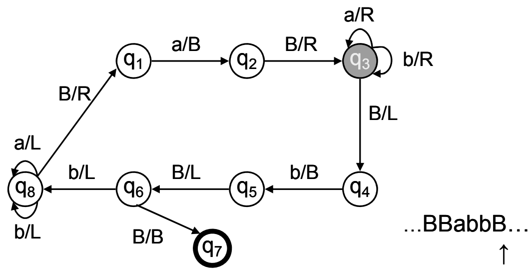


…BBabbB…
↑

Languages Accepted by Turing Machines

Example 2:

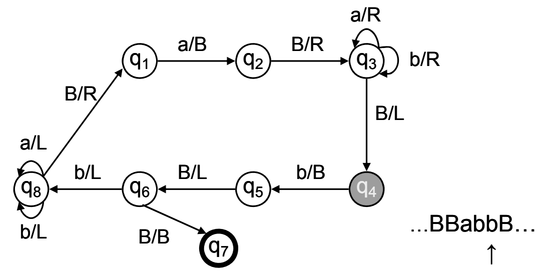
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

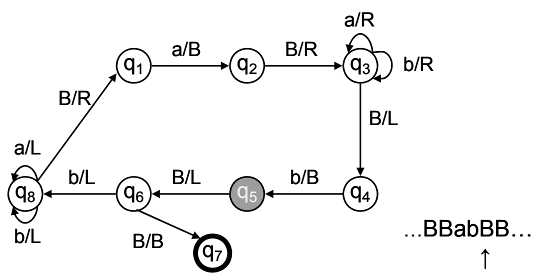
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

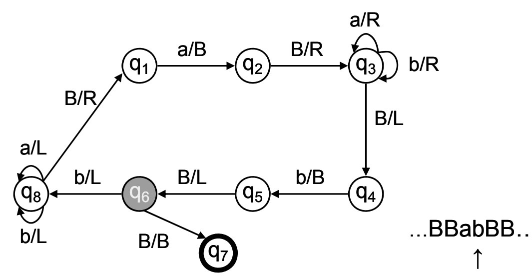
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

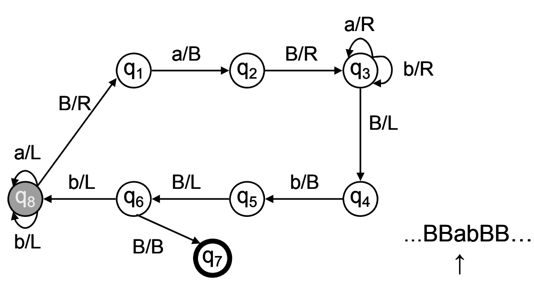
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

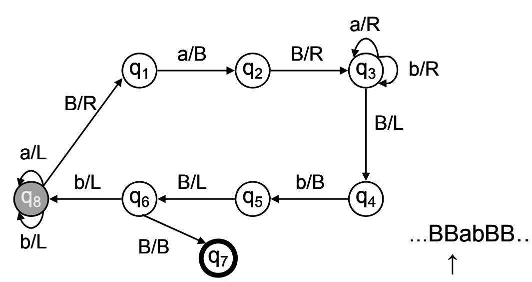
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

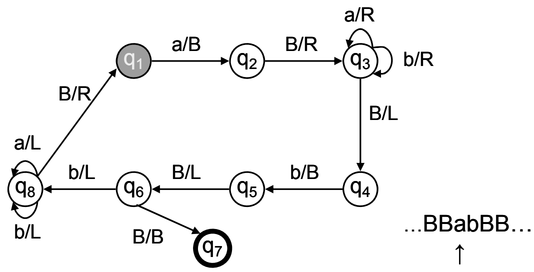
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

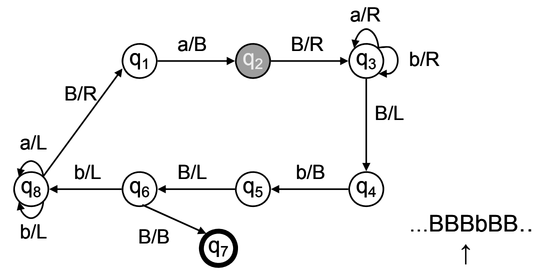
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

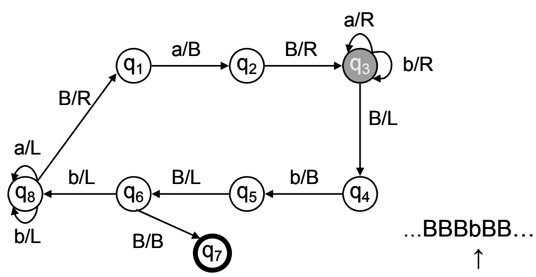
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

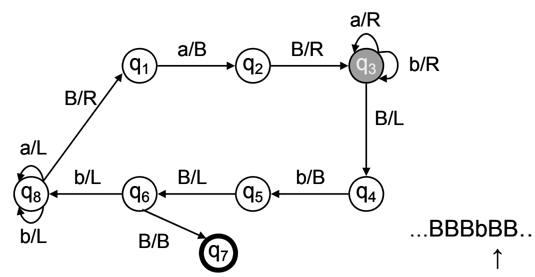
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

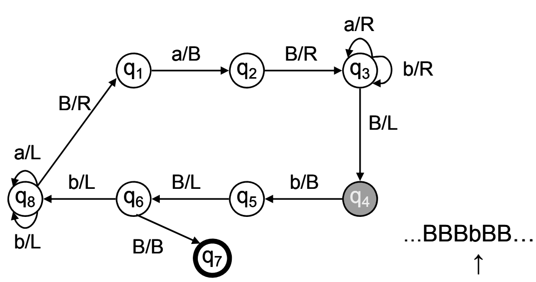
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

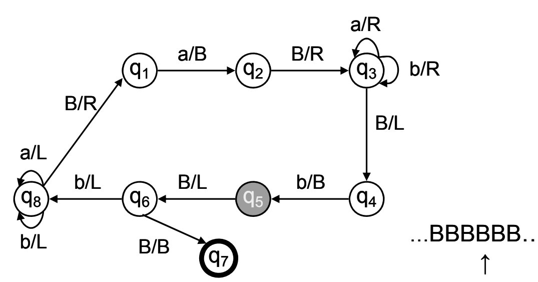
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

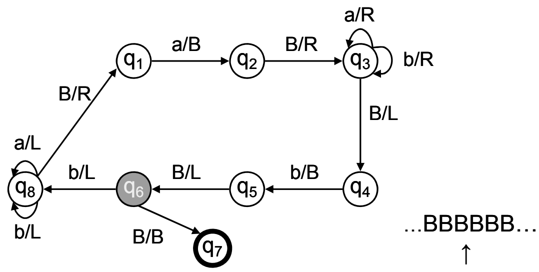
Language L = $\{a^{[n]}b^{[n]} \mid n > 0\}$ on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

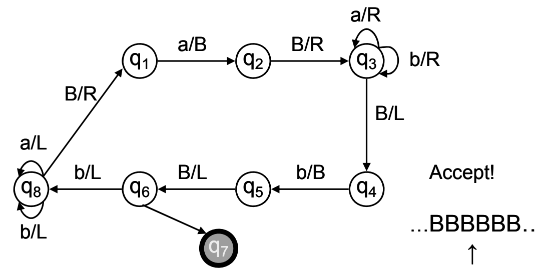
Language L = {$a^{[n]}b^{[n]}$ | n > 0} on the alphabet {a, b}.



Languages Accepted by Turing Machines

Example 2:

Language L = {$a^{[n]}b^{[n]}$ | n > 0} on the alphabet {a, b}.



TMs as Computers

So what languages can Turing machines accept?

This class of languages is called the **recursively enumerable languages**.

For example, it includes the language L = {$a^{[n]}b^{[n]}$ | n > 0}, which is not a regular language.

This shows that the unlimited memory leads to **increased computational capabilities**.

But what about the **functions** that Turing machines can compute?

(After the TM halts, the symbols on the tape can be regarded as the output of the TM, or the result of the function it computes.)

Church's Thesis

Church's thesis:

Any function that can be algorithmically computed can also be computed by a Turing machine.

Why is it only a **thesis**?

There is no general mathematical definition of algorithm – it is always specific to a particular scheme, i.e., a programming language.

Therefore, Church's thesis cannot be proved.

However, no counterexample has been found, and no scientist seriously doubts that Church's thesis is correct.

The Halting Problem

Are there any functions that cannot be computed?

Yes!

The most famous of those is the halting problem.

Let us enumerate all possible Turing machines, i.e., give them numbers 1, 2, 3, …

Then the function HALT(x, y) is true if Turing machine y terminates when given input x, and is false otherwise.

Here comes a surprise:

HALT(x, y) is not a computable function.

The Halting Problem

Because of Church's thesis, we can state the halting problem in a more general form:

There exists no algorithm that can decide for any other algorithm whether its computation will ever halt for a given input.

The Halting Problem

It may surprise you that there is no algorithm for solving the halting problem.

But did we not assume that the Theory of Computation applies to **all** things that compute, including our **brains**?

And is it not possible for a **computer scientist** to analyze a given program and find out whether it will terminate for a particular input or not (even if this analysis takes a very long time)?

No, actually we can devise a very simple Turing machine or computer program for which to date **nobody** is able to tell whether it will ever terminate.

The Halting Problem

This program is based on **Goldbach's conjecture**, which assumes that every even number ≥ 4 is the sum of two prime numbers.

For example, 4 = 2 + 2, 6 = 3 + 3, 48 = 41 + 7.

It was first stated by Goldbach in 1742, and despite great efforts, nobody has ever been able to prove or disprove it.

It would be easy to build a Turing machine or write a C, Java, or Basic program that searches for a counterexample to this conjecture.

Nobody knows whether this program will ever halt.