

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

INTRODUCTION

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

INTRODUCTION

- ▶ **Introduction**
- ▶ **Why study algorithms?**
- ▶ **Coursework**
- ▶ **Resources**
- ▶ **Outline**

Instructor and Course Schedule

- Section I- Dr.Aykut ERDEM
- aykut@cs.hacettepe.edu.tr
- Office: 111

- Section II- Dr. Erkut ERDEM
- erkut@cs.hacettepe.edu.tr
- Office: 112

- Section III- Dr. Adnan Ozsoy
- adnan.ozsoy@hacettepe.edu.tr
- Office: 114

- Lectures: Monday, 09:00 - 09:45 @D1-D2-D3
Tuesday, 09:00 - 10:45 @D1-D2-D3
- Practicum (BBM204): Friday, 14:00-16:50@D3-D4-D8

Instructor and Course Schedule

- Teaching Assistants
 - Selim Yilmaz selimy@cs.hacettepe.edu.tr
 - Levent Karacan karakan@cs.hacettepe.edu.tr
 - Merve Özdes merve@cs.hacettepe.edu.tr
- Practicum (BBM204): Friday, 14:00-16:50@D3-D4-D8

About BBM202-204

- This course concerns programming and problem solving, with applications.
- The aim is to teach student how to develop algorithms in order to solve the complex problems in the most efficient way.
- The students are expected to develop a foundational understanding and knowledge of key concepts that underly important algorithms in use on computers today.
- The students are also be expected to gain hand-on experience via a set of programming assignments supplied in the complementary BBM 204 Software Practicum.
- Grading for BBM204 will be based on a set of quizzes (20%), and 4 programming assignments (done individually) (80%).

Why study algorithms?

Their impact is broad and far-reaching.

Internet. Web search, packet routing, distributed file sharing, ...

Biology. Human genome project, protein folding, ...

Computers. Circuit layout, file system, compilers, ...

Computer graphics. Movies, video games, virtual reality, ...

Security. Cell phones, e-commerce, voting machines, ...

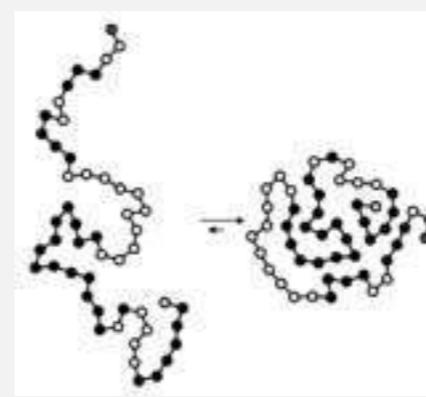
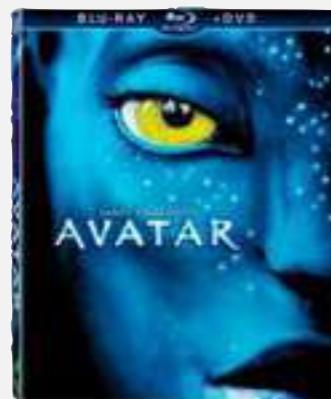
Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

Social networks. Recommendations, news feeds, advertisements, ...

Physics. N-body simulation, particle collision simulation, ...

:

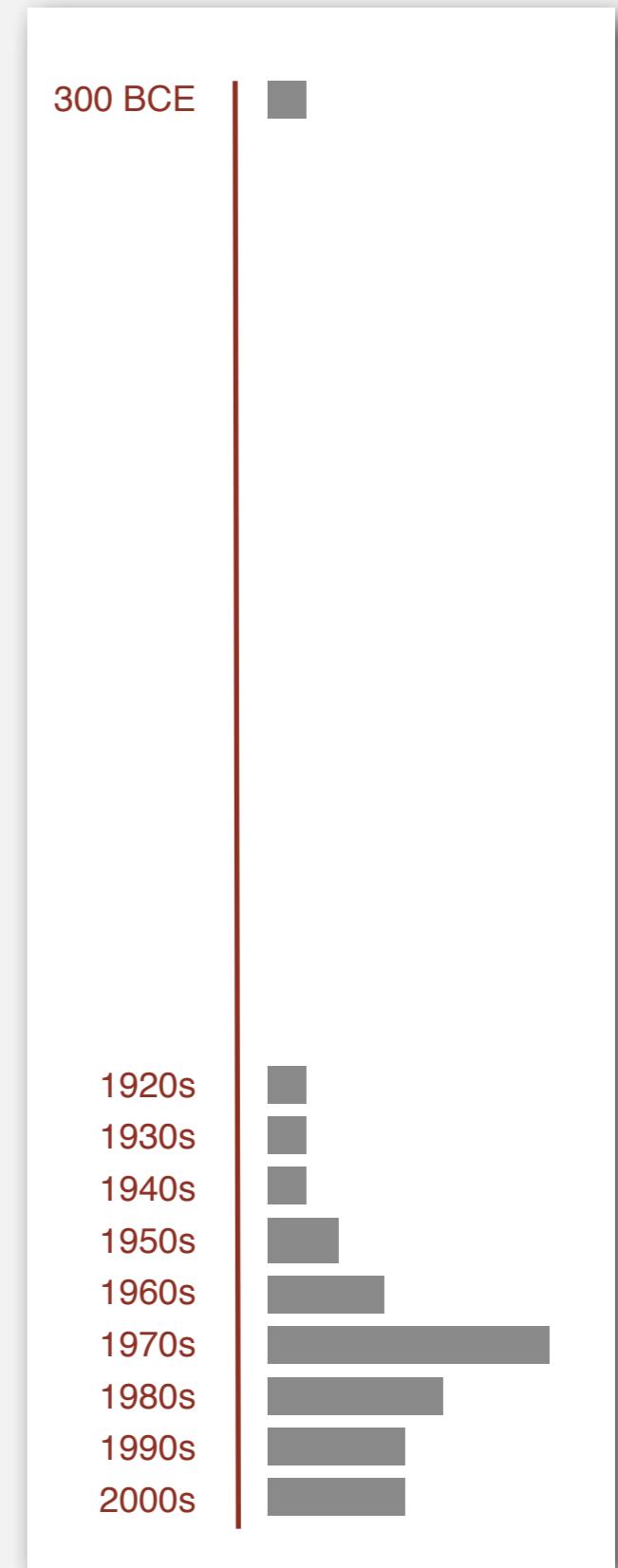
Google
YAHOO!
bing



Why study algorithms?

Old roots, new opportunities.

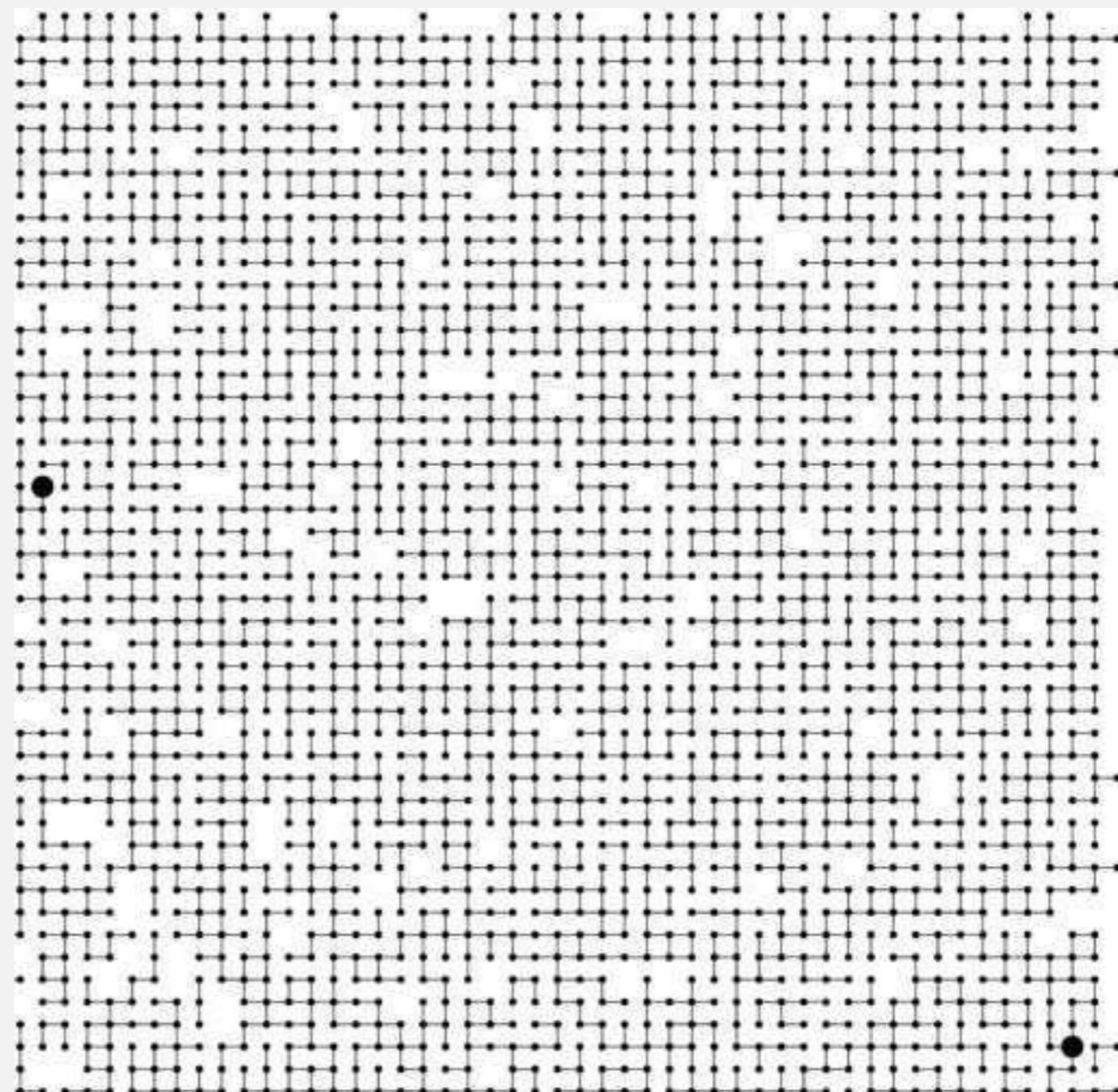
- Study of algorithms dates at least to Euclid.
- Formalized by Church and Turing in 1930s.
- Some important algorithms were discovered by undergraduates in a course like this!



Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity.



Why study algorithms?

For intellectual stimulation.

“For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious.

But once unlocked, they cast a brilliant new light on some aspect of computing. ” — Francis Sullivan

FROM THE EDITORS

THE JOY OF ALGORITHMS

Francis Sullivan, Associate Editor-in-Chief



THE THEME OF THIS FIRST-OF-THE-CENTURY ISSUE OF COMPUTING IN SCIENCE & ENGINEERING IS ALGORITHMS. IN FACT, WE WERE BOLD ENOUGH—AND PERHAPS FOOLISH ENOUGH—to call the 10 examples we've selected “THE TOP ALGORITHMS OF THE CENTURY.”

Computational algorithms are probably as old as civilization. Some of the most important early algorithms, consisting earthly of algorithms for reckoning in trade, were probably developed by the Babylonians around 60. And I suppose we could claim that the Droid algorithm for estimating the state of summer is embodied in Stonehenge. (That's the theory of the 100-year-old British archaeologist.)

Like so many other things that technology affects, algorithms have changed over time. In fact, they've changed a lot. At least it looks that way to us now. The algorithms we chose for this issue have been essential for progress in numerous fields, from space exploration to medical imaging, weather prediction, defense, and fundamental science. Conversely, progress in these areas has stimulated the search for ever-newer and more powerful algorithms. For example, in 1969, the Maryland Shore when someone asked, “Who first ate a crab? After all, they don't look very appetizing.” After the usual speculation, the person who had asked the question said, “I don't care what must be the right answer—sample.” A very hungry person like me are a crab.”

The algorithm is the mother of invention: “An invention creates its own necessity.” Our need for powerful machines always exceeds their availability. Each significant computational advance has opened up new possibilities, but also how to extract information from extremely large masses of data is still almost untouched. There are still very big challenges ahead. For example, in medicine, we want to predict the onset of a disease, or to detect cancerous cells. For example, we need efficient methods to tell when the result of a large floating-point calculation is likely to be correct. Think of the difference between a 10-cent error in a bank account and one that is very small, but the added confidence in the answer is large.

Is there an analog for things such as huge, multidisciplinary teams? I think so. We have to learn how to work together to find better ways to attack them?

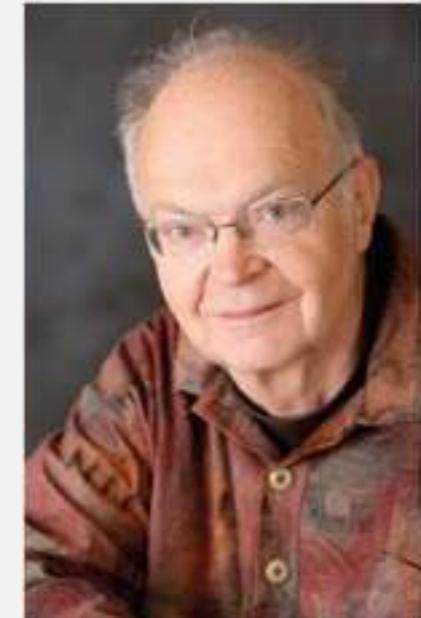
I suspect that in the 21st century, there will be ripe for algorithmic breakthroughs in the foundations of the fields of computational theory. Questions already arising from quantum computing, for example, are forcing us to re-examine what random numbers seem to require that we somehow tie together theories of computing, logic, and the nature of the physical world.

For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even

inventive. I hope you'll agree.

Francis Sullivan
Associate Editor-in-Chief

“It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e. express it as an algorithm. The attempt to formalise things as algorithms lead to a much deeper understanding than if we simply try to comprehend things in the traditional way. algorithm must be seen to be believed. ” — Donald Knuth



Why study algorithms?

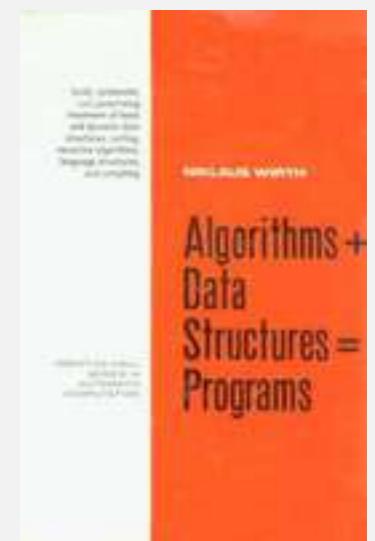
To become a proficient programmer.

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. ”

— Linus Torvalds (creator of Linux)



“Algorithms + Data Structures = Programs. ” — Niklaus Wirth



Why study algorithms?

They may unlock the secrets of life and of the universe.

Computational models are replacing mathematical models in scientific inquiry.

$$E = mc^2$$

$$F = ma$$

$$\left[-\frac{\hbar^2}{2m} \nabla^2 + V(r) \right] \Psi(r) = E \Psi(r)$$

$$F = \frac{Gm_1 m_2}{r^2}$$

20th century science

(formula based)

```
for (double t = 0.0; true; t = t + dt)
    for (int i = 0; i < N; i++)
    {
        bodies[i].resetForce();
        for (int j = 0; j < N; j++)
            if (i != j)
                bodies[i].addForce(bodies[j]);
    }
```

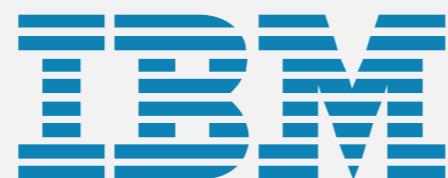
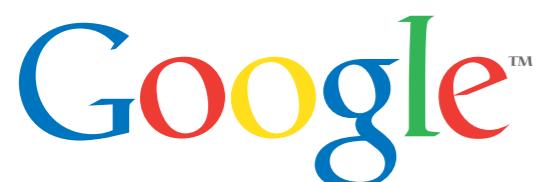
21st century science

(algorithm based)

“Algorithms: a common language for nature, human, and computer.” — Avi Wigderson

Why study algorithms?

For fun and profit.



Why study algorithms?

- Their impact is broad and far-reaching.
- Old roots, new opportunities.
- To solve problems that could not otherwise be addressed.
- For intellectual stimulation.
- To become a proficient programmer.
- They may unlock the secrets of life and of the universe.
- For fun and profit.

Why study anything else?



Communication

- The course webpage will be updated regularly throughout the semester with lecture notes, programming assignments and important deadlines.
- <http://web.cs.hacettepe.edu.tr/~bbm202>
- <https://piazza.com/configure-classes/spring2018/bbm202>

Getting help

- Office Hours
- BBM204 Software Practicum
 - Course related recitations, practice with algorithms, etc.
- Communication
 - Announcements and course related discussions
 - through **piazza** : <https://piazza.com/configure-classes/spring2018/bbm202>



The image contains two side-by-side screenshots of the Piazza mobile application. The left screenshot shows the main "Classes" screen with a grid of nine course icons. The right screenshot shows a specific course page for "BBM 202" with a note titled "Merhabalar". The note text reads:

Bu dönem BBM202 (2. Sube) ve BBM204 (3.-5. Subeler) dersleri ile ilgili duyuru ve tartışmalar için Piazza'yi kullanıyor olacağız. Lütfen dersle alakalı sorularınızı e-mail atmak yerine Piazza'nın sağladığı yine bu ortamı kullanınız.

Umarım hep birlikte güzel bir dönem geçirez.
Hepinize başarılar dilerim,
-Erkut

Coursework and grading

Class participation/Attendance 5%

- Contribute to Piazza discussions.
- Attend and participate in lecture.

Midterm exams 55% (30+25)

- Two closed-book exams
 - in class on March 20 and April 17

Final exam. 40%

- Closed-book
- Scheduled by Registrar.

BBM204 Software Practicum

Programming assignments (PAs)

- Four assignments throughout the semester.
- Each assignment has a well-defined goal such as solving a specific problem.
- You must work alone on all assignments stated unless otherwise.

Important Dates

- Programming Assignment 1 27 February
- Programming Assignment 2 20 March
- Programming Assignment 3 17 April
- Programming Assignment 4 1 May

Cheating

What is cheating?

- Sharing code: by copying, retyping, looking at, or supplying a file
- Coaching: helping your friend to write a programming assignment, line by line
- Copying code from previous course or from elsewhere on WWW

What is NOT cheating?

- Explaining how to use systems or tools
- Helping others with high-level design issues

Penalty for cheating:

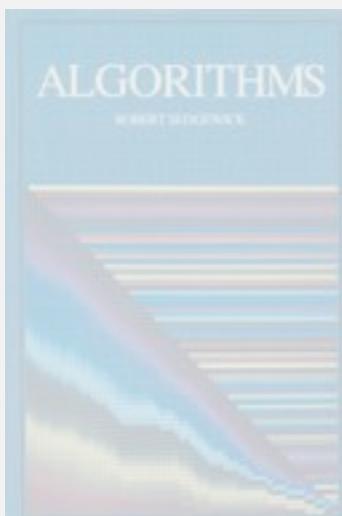
- Helping others with high-level design issues
- A violation of academic integrity, disciplinary action

Detection of cheating:

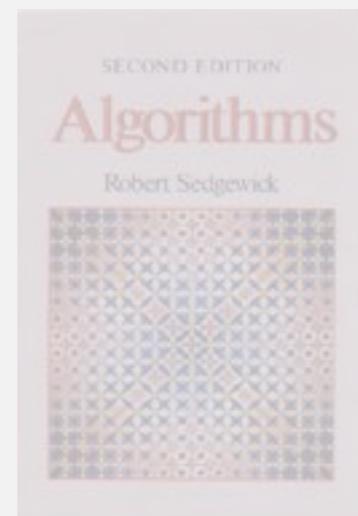
- We do check
- Our tools for doing this are much better than most cheaters think!

Resources (textbook)

Required reading. Algorithms 4th edition by R. Sedgewick and K. Wayne, Addison-Wesley Professional, 2011, ISBN 0-321-57351-X.



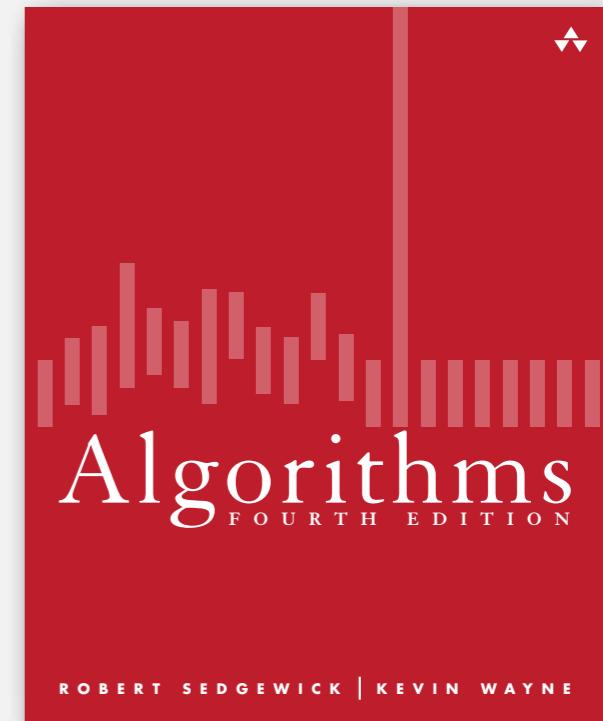
1st edition (1982)



2nd edition (1988)



3rd edition (1997)



Booksite.

- Brief summary of content.
- Download code from book.

A screenshot of the Booksite for the 4th edition of "Algorithms". The page includes the book cover, a brief description, and a table of contents. The description highlights that the book surveys important algorithms and data structures. The table of contents lists chapters 1 through 6.

<https://algs4.cs.princeton.edu/home/>

Course outline

Introduction

Analysis of Algorithms

- Computational Complexity

Sorting

- Elementary Sorting Algorithms,
- Mergesort,
- Quicksort,
- Priority Queues and HeapSort

Searching

- Sequential Search
- Binary Search Trees
- Balanced Trees
- Hashing,
- Search Applications

Course outline

Graphs

- Undirected Graphs,
- Directed Graphs,
- Minimum Spanning Trees,
- Shortest Path

Strings

- String Sorts, Tries,
- Substring Search,
- Regular Expressions,
- Data Compression

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ANALYSIS OF ALGORITHMS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ **Analysis of Algorithms**
- ▶ **Observations**
- ▶ **Mathematical models**
- ▶ **Order-of-growth classifications**
- ▶ **Dependencies on inputs**
- ▶ **Memory**

Cast of characters



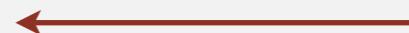
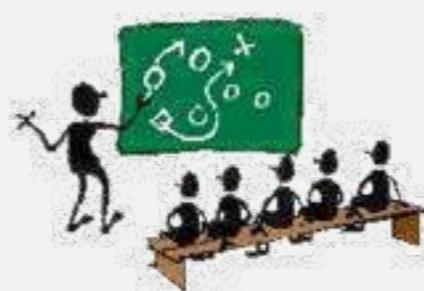
Programmer needs to develop
a working solution.



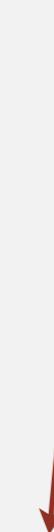
Client wants to solve
problem efficiently.



Theoretician wants
to understand.



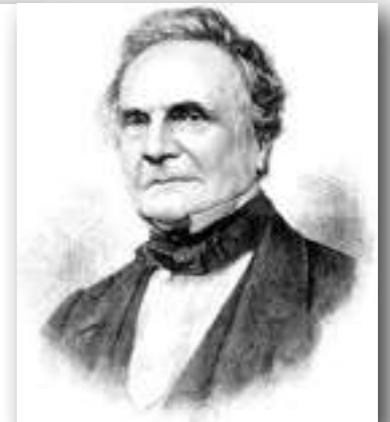
Student might play
any or all of these
roles someday.



Basic blocking and tackling
is sometimes necessary.
[this lecture]

Running time

“As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? ” — Charles Babbage (1864)



how many times do you have to turn the crank?

Analytic Engine

Reasons to analyse algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (BBM 202)

Analysis of algorithms (BBM 408)

Primary practical reason: avoid performance bugs.



**client gets poor performance because programmer
did not understand performance characteristics**



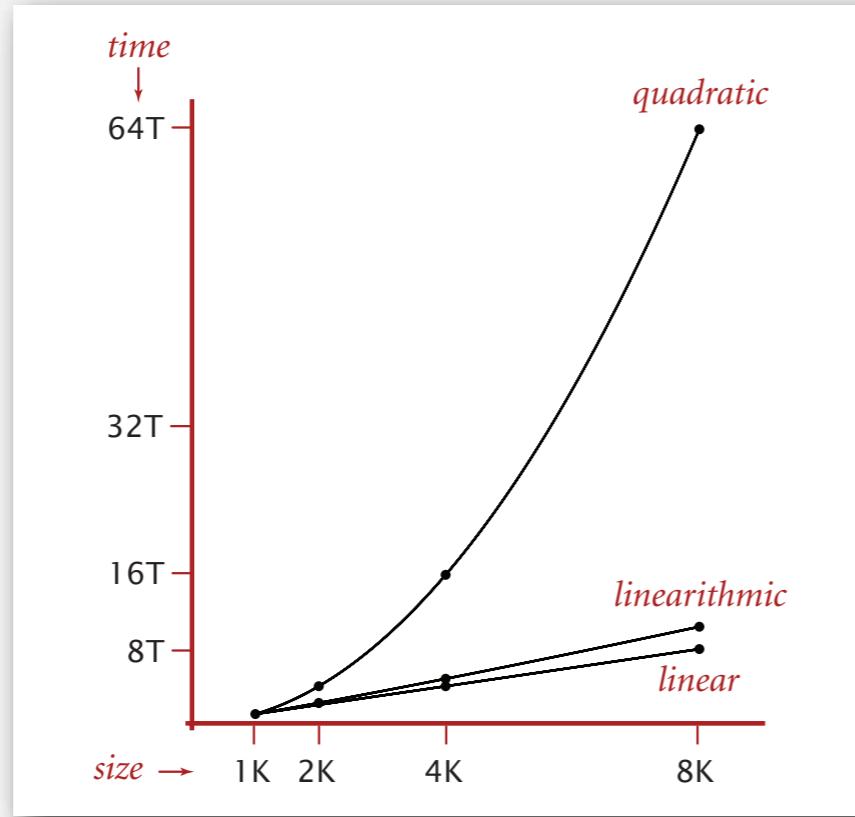
Some algorithmic successes

Discrete Fourier transform.

- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics,
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, enables new technology.



Friedrich Gauss
1805



- sFFT: Sparse Fast Fourier Transform algorithm (Hassanieh et al., 2012)
 - A faster Fourier Transform: $k \log N$ steps (with k sparse coefficients)

Some algorithmic successes

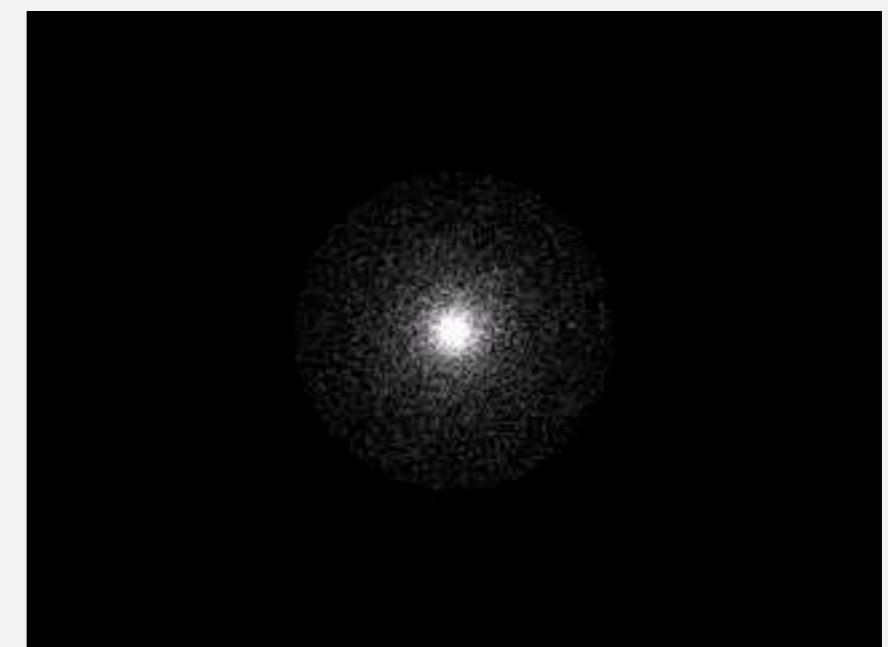
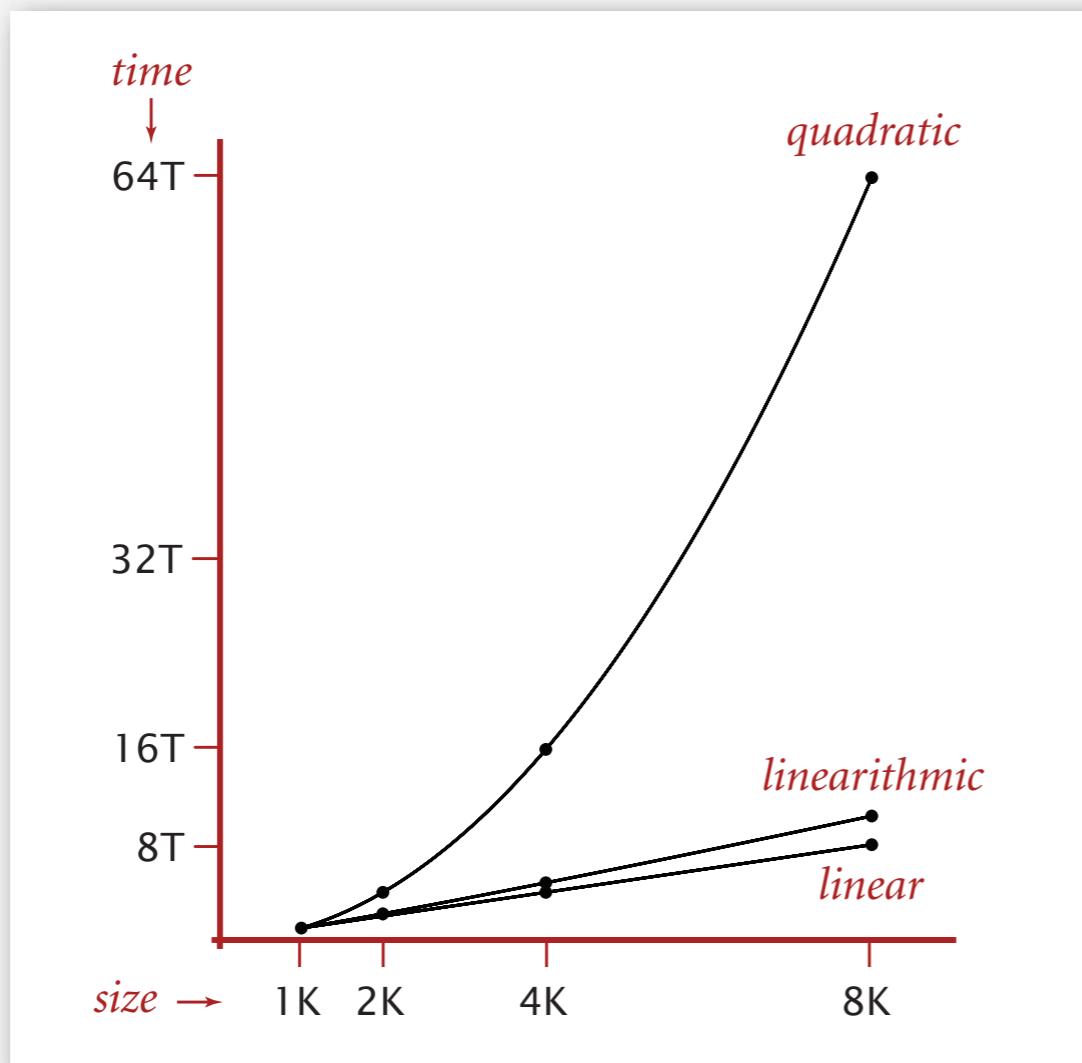
N-body simulation.

- Simulate gravitational interactions among N bodies.
- Brute force: N^2 steps.
- Barnes-Hut algorithm: $N \log N$ steps, enables new research.



Andrew Appel

PU '81



The challenge

Q. Will my program be able to solve a large practical input?

Why is my program so slow ?

Why does it run out of memory ?



Key insight. [Knuth 1970s] Use scientific method to understand performance.

Scientific method applied to analysis of algorithms

A framework for predicting performance and comparing algorithms.

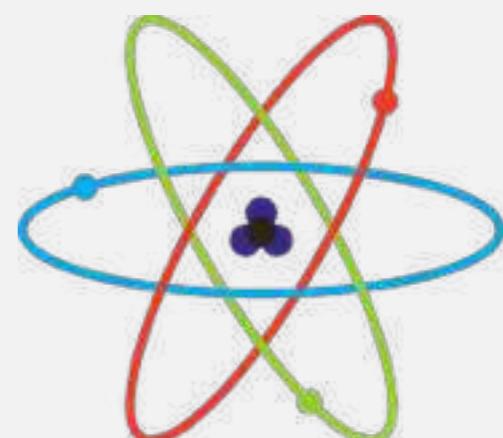
Scientific method.

- **Observe** some feature of the natural world.
- **Hypothesize** a model that is consistent with the observations.
- **Predict** events using the hypothesis.
- **Verify** the predictions by making further observations.
- **Validate** by repeating until the hypothesis and observations agree.

Principles.

Experiments must be **reproducible**.

Hypotheses must be **falsifiable**.



Feature of the natural world = computer itself.

ANALYSIS OF ALGORITHMS

- ▶ **Observations**
- ▶ **Mathematical models**
- ▶ **Order-of-growth classifications**
- ▶ **Dependencies on inputs**
- ▶ **Memory**

Example: 3-sum

3-sum. Given N distinct integers, how many triples sum to exactly zero?

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5

% java ThreeSum 8ints.txt
4
```

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0

Context. Deeply related to problems in computational geometry.

3-sum: brute-force algorithm

```
public class ThreeSum
{
    public static int count(int[] a)
    {
        int N = a.length;
        int count = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                for (int k = j+1; k < N; k++)
                    if (a[i] + a[j] + a[k] == 0)
                        count++;
        return count;
    }

    public static void main(String[] args)
    {
        int[] a = In.readInts(args[0]);
        StdOut.println(count(a));
    }
}
```

← check each triple
← for simplicity, ignore
integer overflow

Measuring the running time

Q. How to time a program?

A. Manual.



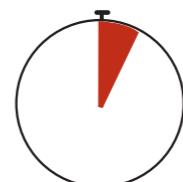
```
% java ThreeSum 1Kints.txt
```



tick tick tick

70

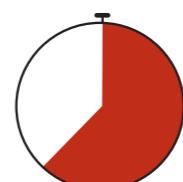
```
% java ThreeSum 2Kints.txt
```



*tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick
tick tick tick tick tick tick tick tick*

528

```
% java ThreeSum 4Kints.txt
```



4039

Measuring the running time

Q. How to time a program?

A. Automatic.

```
public class Stopwatch (part of stdlib.jar)
```

Stopwatch()	<i>create a new stopwatch</i>
--------------------	-------------------------------

double elapsedTime()	<i>time since creation (in seconds)</i>
-----------------------------	---

```
public static void main(String[] args)
{
    int[] a = In.readInts(args[0]);
    Stopwatch stopwatch = new Stopwatch();
    StdOut.println(ThreeSum.count(a));
    double time = stopwatch.elapsedTime();
}
```

Measuring the running time

Q. How to time a program?

A. Automatic.

```
public class Stopwatch (part of stdlib.jar)
```

Stopwatch()	<i>create a new stopwatch</i>
--------------------	-------------------------------

double elapsedTime()	<i>time since creation (in seconds)</i>
-----------------------------	---

```
public class Stopwatch
{
    private final long start = System.currentTimeMillis();

    public double elapsedTime()
    {
        long now = System.currentTimeMillis();
        return (now - start) / 1000.0;
    }
}
```

implementation (part of stdlib.jar)

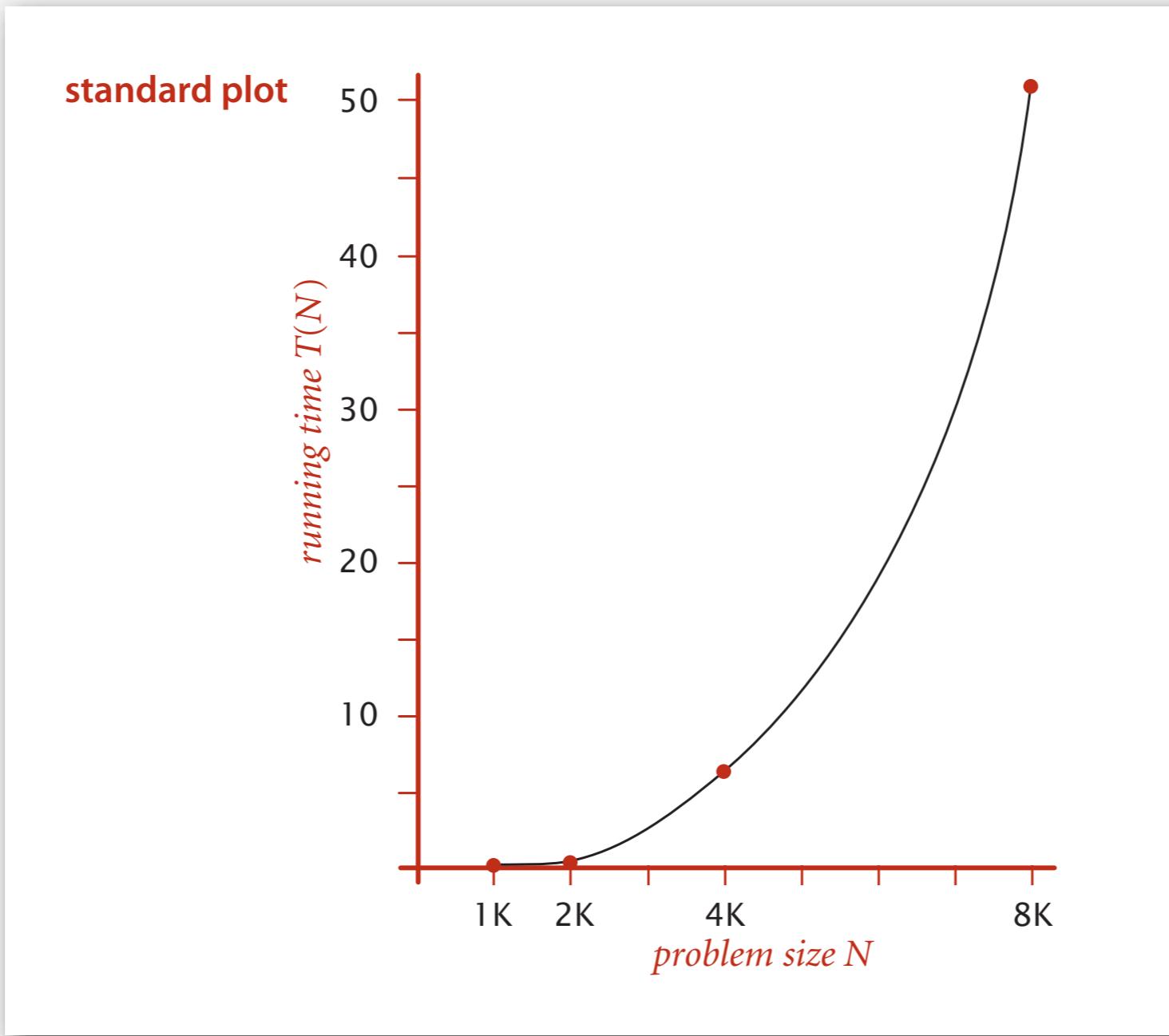
Empirical analysis

Run the program for various input sizes and measure running time.

N	time (seconds) t
250	0
500	0
1.000	0,1
2.000	0,8
4.000	6,4
8.000	51,1
16.000	?

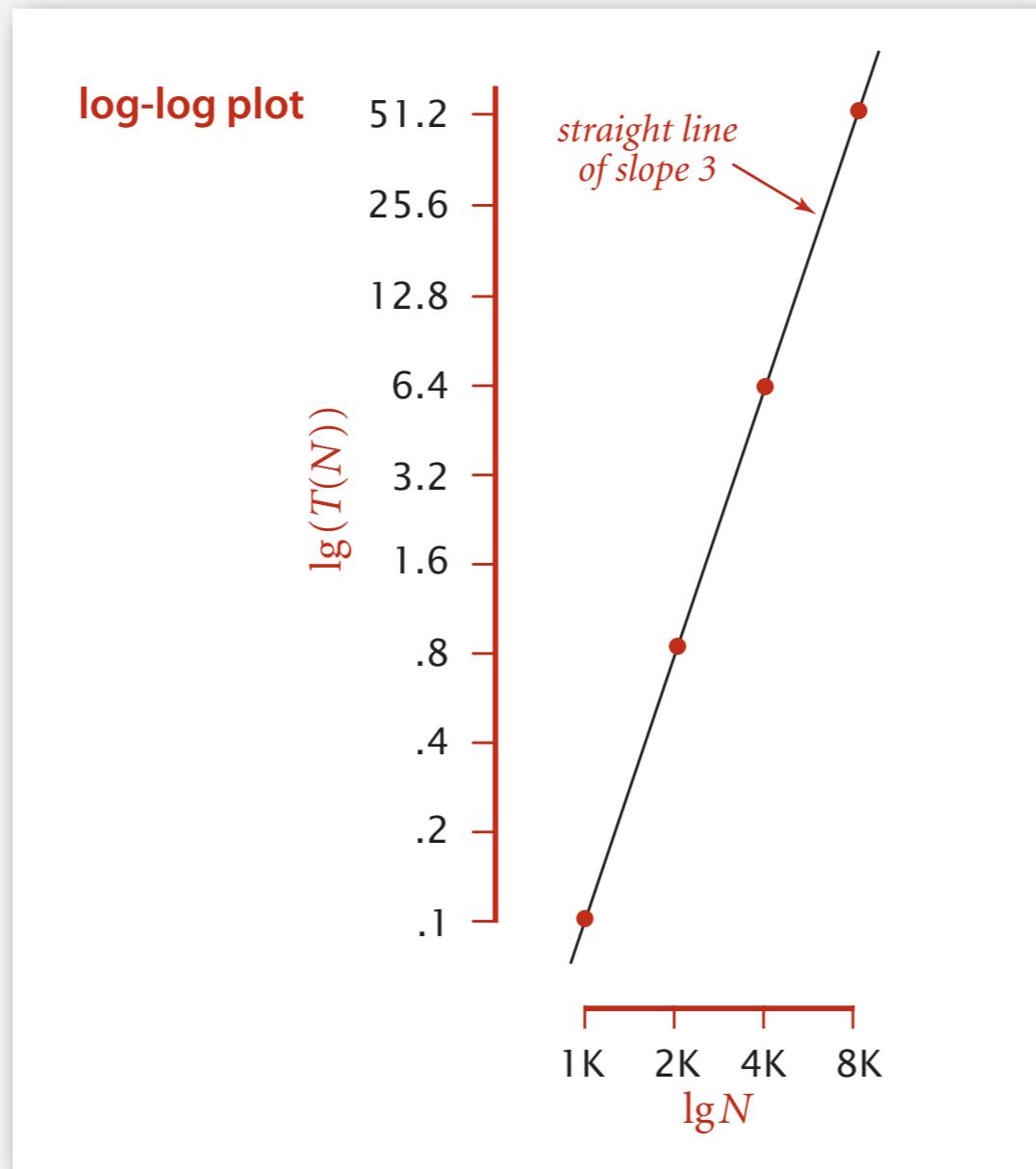
Data analysis

Standard plot. Plot running time $T(N)$ vs. input size N .



Data analysis

Log-log plot. Plot running time $T(N)$ vs. input size N using log-log scale.



$$\lg(T(N)) = b \lg N + c$$

$$b = 2.999$$

$$c = -33.2103$$

$$T(N) = a N^b, \text{ where } a = 2^c$$

Regression. Fit straight line through data points: $a N^b$.

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

power law

slope

Prediction and validation

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.



"order of growth" of running time is about N^3 [stay tuned]

Predictions.

- 51.0 seconds for $N = 8,000$.
- 408.1 seconds for $N = 16,000$.

Observations.

N	time (seconds) †
8.000	51,1
8.000	51
8.000	51,1
16.000	410,8

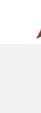
validates hypothesis!

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law relationship.

Run program, **doubling** the size of the input.

N	time (seconds) †	ratio	lg ratio
250	0		—
500	0	4,8	2,3
1.000	0,1	6,9	2,8
2.000	0,8	7,7	2,9
4.000	6,4	8	3
8.000	51,1	8	3



seems to converge to a constant $b \approx 3$

Hypothesis. Running time is about $a N^b$ with $b = \lg \text{ratio}$.

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

Doubling hypothesis. Quick way to estimate b in a power-law hypothesis.

Q. How to estimate a (assuming we know b) ?

A. Run the program (for a sufficient large value of N) and solve for a .

N	time (seconds) †
8.000	51,1
8.000	51
8.000	51,1

$$51.1 = a \times 8000^3$$

$$\Rightarrow a = 0.998 \times 10^{-10}$$

Hypothesis. Running time is about $0.998 \times 10^{-10} \times N^3$ seconds.



almost identical hypothesis

to one obtained via linear regression

Experimental algorithmics

System independent effects.

- Algorithm.
 - Input data.
- 
- determines exponent b
in power law

System dependent effects.

- Hardware: CPU, memory, cache, ...
 - Software: compiler, interpreter, garbage collector, ...
 - System: operating system, network, other applications, ...
- 
- determines constant a
in power law

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.



e.g., can run huge number of experiments

In practice, constant factors matter too!

Q. How long does this program take as a function of N ?

```
String s = StdIn.readString();
int N = s.length();
...
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        distance[i][j] = ...
...
```

N	time
1.000	0,11
2.000	0,35
4.000	1,6
8.000	6,5

Jenny $\sim c_1 N^2$ seconds

N	time
250	0,5
500	1,1
1.000	1,9
2.000	3,9

Kenny $\sim c_2 N$ seconds

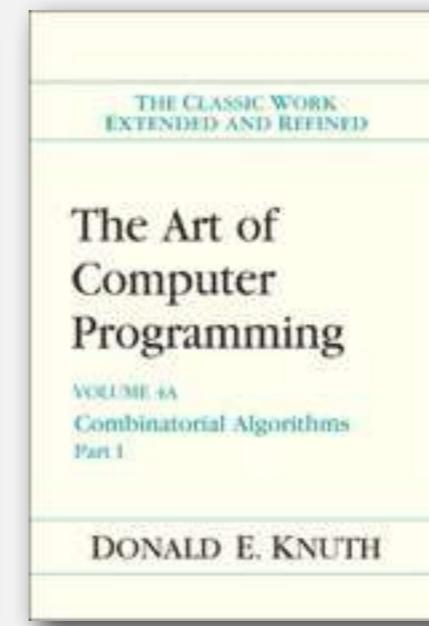
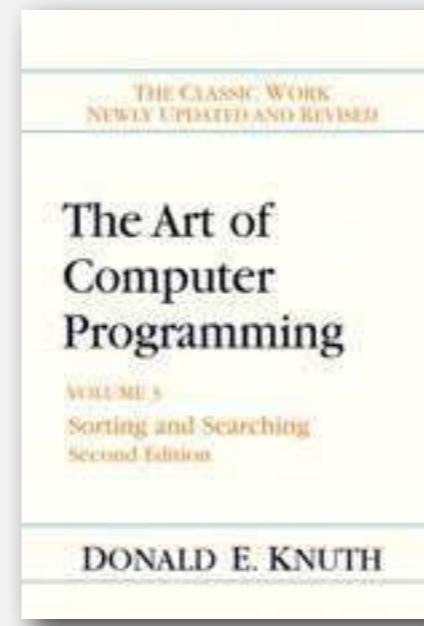
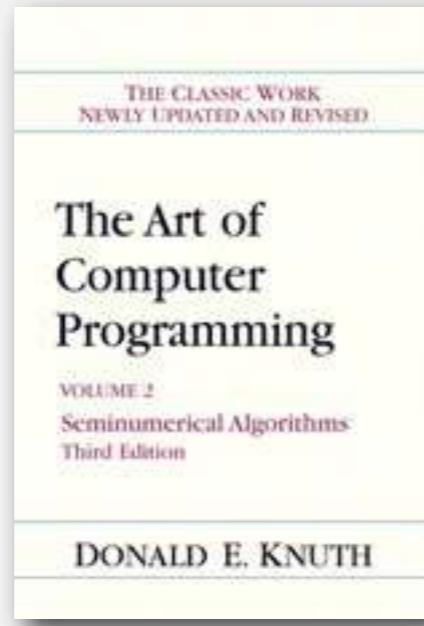
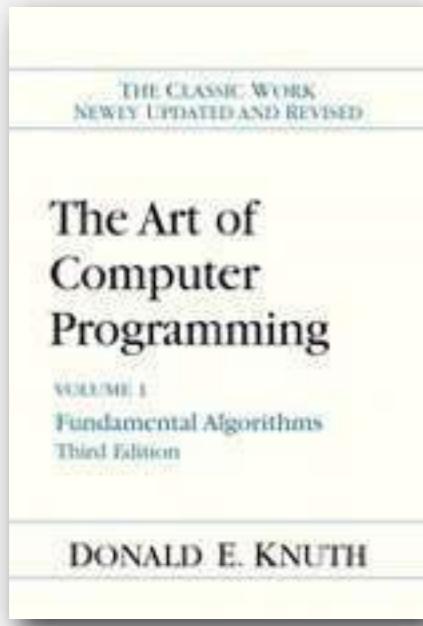
ANALYSIS OF ALGORITHMS

- ▶ **Observations**
- ▶ **Mathematical models**
- ▶ **Order-of-growth classifications**
- ▶ **Dependencies on inputs**
- ▶ **Memory**

Mathematical models for running time

Total running time: sum of cost \times frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

operation	example	nanoseconds †
integer add	<code>a + b</code>	2,1
integer multiply	<code>a * b</code>	2,4
integer divide	<code>a / b</code>	5,4
floating-point add	<code>a + b</code>	4,6
floating-point multiply	<code>a * b</code>	4,2
floating-point divide	<code>a / b</code>	13,5
sine	<code>Math.sin(theta)</code>	91,3
arctangent	<code>Math.atan2(y, x)</code>	129
...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

operation	example	nanoseconds [†]
variable declaration	<code>int a</code>	c_1
assignment statement	<code>a = b</code>	c_2
integer compare	<code>a < b</code>	c_3
array element access	<code>a[i]</code>	c_4
array length	<code>a.length</code>	c_5
1D array allocation	<code>new int[N]</code>	$c_6 N$
2D array allocation	<code>new int[N][N]</code>	$c_7 N^2$
string length	<code>s.length()</code>	c_8
substring extraction	<code>s.substring(N/2, N)</code>	c_9
string concatenation	<code>s + t</code>	$c_{10} N$

Novice mistake. Abusive string concatenation.

Example: I-sum

Q. How many instructions as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    if (a[i] == 0)  
        count++;
```

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2N$

Example: 2-sum

Q. How many instructions as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$
equal to compare	$\frac{1}{2} N (N - 1)$
array access	$N (N - 1)$
increment	$\frac{1}{2} N (N - 1)$ to $N (N - 1)$

$$0 + 1 + 2 + \dots + (N - 1) = \frac{1}{2} N (N - 1) \\ = \binom{N}{2}$$

tedious to count exactly

Simplifying the calculations

*“ It is convenient to have a **measure of the amount of work involved in a computing process**, even though it be a very **crude** one. We may count up the number of times that various elementary operations are applied in the whole process and then give them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. ” — Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(National Physical Laboratory, Teddington, Middlesex)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known ‘Gauss elimination process’, it is found that the errors are normally quite moderate: no exponential build-up need occur.



Simplification I: cost model

Cost model. Use some basic operation as a proxy for running time.

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

$$0 + 1 + 2 + \dots + (N - 1) = \frac{1}{2}N(N - 1) \\ = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N + 1)(N + 2)$
equal to compare	$\frac{1}{2}N(N - 1)$
array access	$N(N - 1)$
increment	$\frac{1}{2}N(N - 1)$ to $N(N - 1)$

cost model = array accesses

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N .
- Ignore lower order terms.
 - when N is large, terms are negligible
 - when N is small, we don't care

Ex 1. $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$

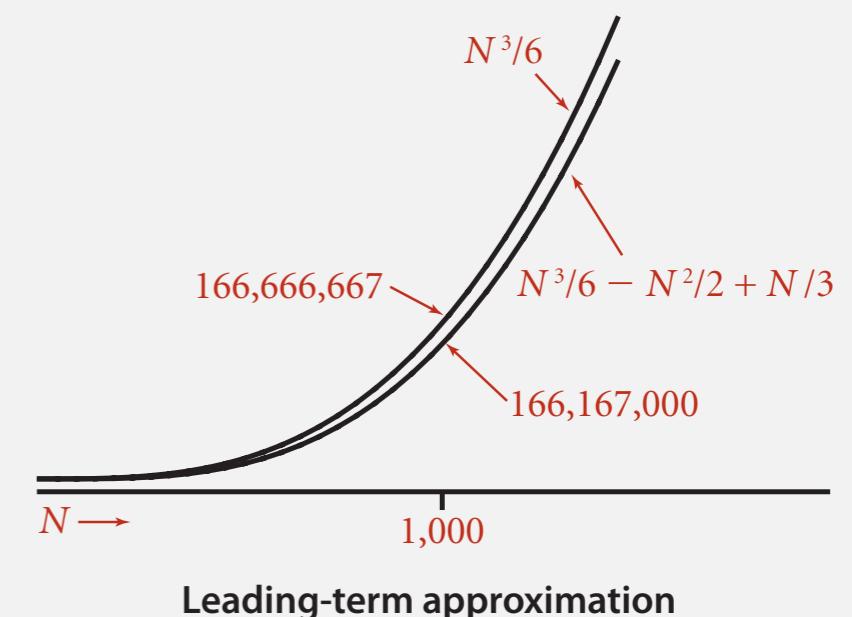
Ex 2. $\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$

Ex 3. $\frac{1}{6}N^3 - \frac{1}{2}N^2 + \frac{1}{3}N \sim \frac{1}{6}N^3$



discard lower-order terms

(e.g., $N = 1000$: 500 thousand vs. 166 million)



Technical definition. $f(N) \sim g(N)$ means $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1$

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N .
- Ignore lower order terms.
 - when N is large, terms are negligible
 - when N is small, we don't care

operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2} (N + 1) (N + 2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N (N - 1)$	$\sim \frac{1}{2} N^2$
array access	$N (N - 1)$	$\sim N^2$
increment	$\frac{1}{2} N (N - 1)$ to $N (N - 1)$	$\sim \frac{1}{2} N^2$ to $\sim N^2$

Example: 2-sum

Q. Approximately how many array accesses as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        if (a[i] + a[j] == 0)  
            count++;
```

"inner loop"

A. $\sim N^2$ array accesses.

$$\begin{aligned}0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2} N(N - 1) \\&= \binom{N}{2}\end{aligned}$$

Bottom line. Use cost model and tilde notation to simplify frequency counts.

Example: 3-sum

Q. Approximately how many array accesses as a function of input size N ?

```
int count = 0;  
for (int i = 0; i < N; i++)  
    for (int j = i+1; j < N; j++)  
        for (int k = j+1; k < N; k++)  
            if (a[i] + a[j] + a[k] == 0) ← "inner loop"  
                count++;
```

A. $\sim \frac{1}{2} N^3$ array accesses.

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \\ \sim \frac{1}{6} N^3$$

Bottom line. Use cost model and tilde notation to simplify frequency counts.

Estimating a discrete sum

Q. How to estimate a discrete sum?

A1. Take discrete mathematics course.

A2. Replace the sum with an integral, and use calculus!

Ex 1. $1 + 2 + \dots + N$.

$$\sum_{i=1}^N i \sim \int_{x=1}^N x dx \sim \frac{1}{2} N^2$$

Ex 2. $1 + 1/2 + 1/3 + \dots + 1/N$.

$$\sum_{i=1}^N \frac{1}{i} \sim \int_{x=1}^N \frac{1}{x} dx = \ln N$$

Ex 3. 3-sum triple loop.

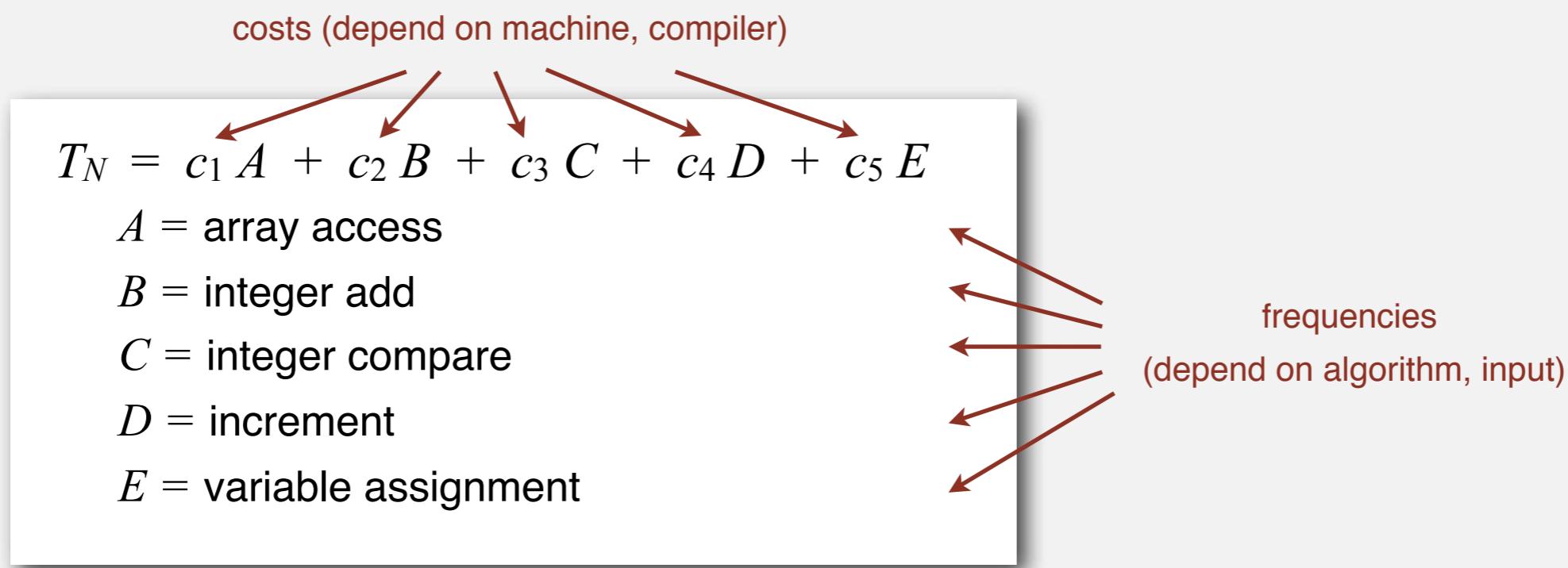
$$\sum_{i=1}^N \sum_{j=i}^N \sum_{k=j}^N 1 \sim \int_{x=1}^N \int_{y=x}^N \int_{z=y}^N dz dy dx \sim \frac{1}{6} N^3$$

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.



Bottom line. We use approximate models in this course: $T(N) \sim c N^3$.

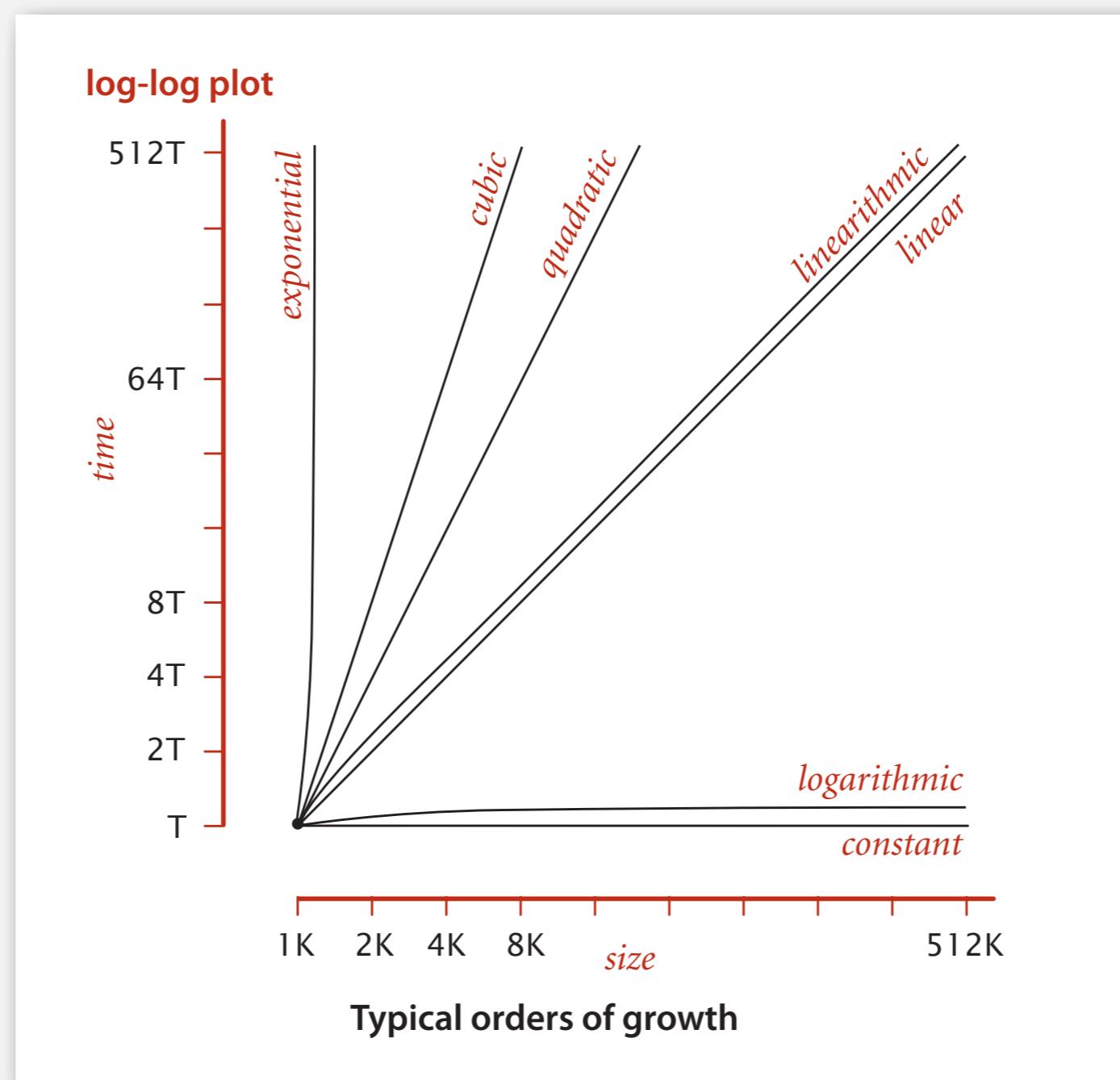
ANALYSIS OF ALGORITHMS

- ▶ **Observations**
- ▶ **Mathematical models**
- ▶ **Order-of-growth classifications**
- ▶ **Dependencies on inputs**
- ▶ **Memory**

Common order-of-growth classifications

Good news. the small set of functions
1, $\log N$, N , $N \log N$, N^2 , N^3 , and 2^N
suffices to describe order-of-growth of typical algorithms.

order of growth discards
leading coefficient



Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	constant	$a = b + c;$	statement	add two numbers	1
$\log N$	logarithmic	{ while ($N > 1$) $N = N / 2;$... }	divide in half	binary search	~ 1
N	linear	for ($int i = 0; i < N; i++$) { ... }	loop	find the maximum	2
$N \log N$	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	for ($int i = 0; i < N; i++$) for ($int j = 0; j < N; j++$) { ... }	double loop	check all pairs	4
N^3	cubic	for ($int i = 0; i < N; i++$) for ($int j = 0; j < N; j++$) for ($int k = 0; k < N; k++$) { ... }	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
$\log N$	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
$N \log N$	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Practical implications of order-of-growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
$\log N$	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
$N \log N$	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millennia

Practical implications of order-of-growth

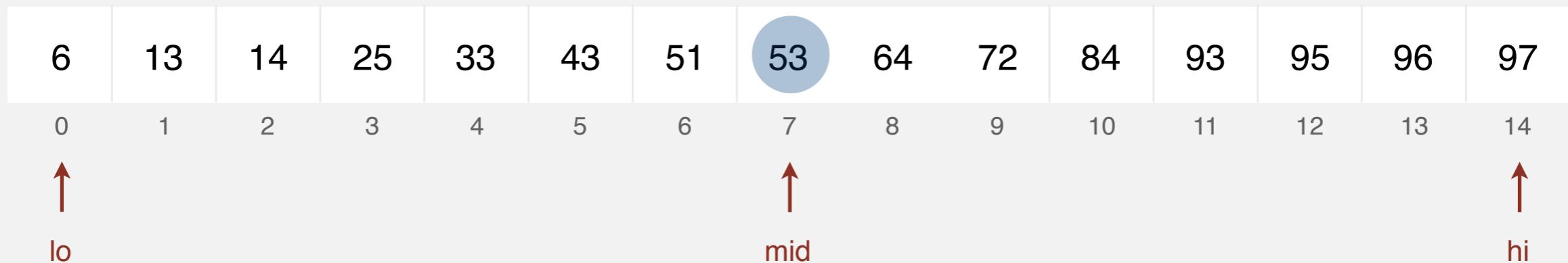
growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	—	—
$\log N$	logarithmic	nearly independent of input size	—	—
N	linear	optimal for N inputs	a few minutes	100x
$N \log N$	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4–5x
2^N	exponential	useful only for tiny problems	forever	1x

Binary search

Goal. Given a sorted array and a key, find index of the key in the array?

Binary search. Compare key against middle entry.

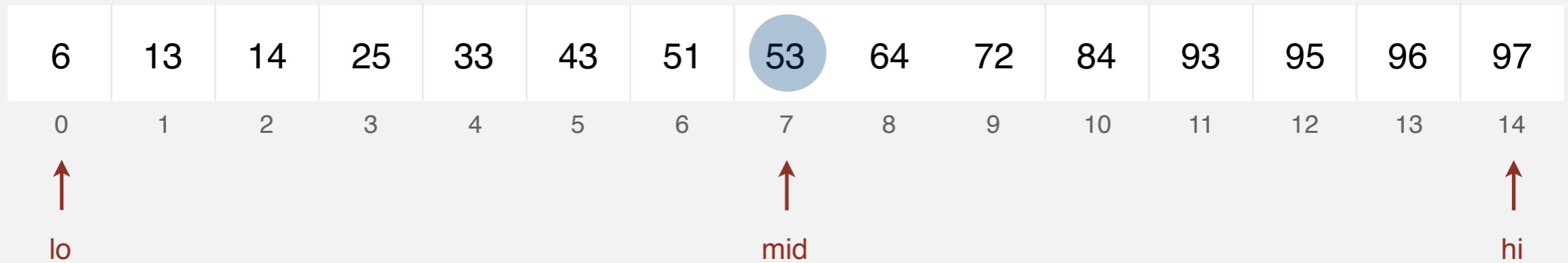
- Too small, go left.
- Too big, go right.
- Equal, found.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

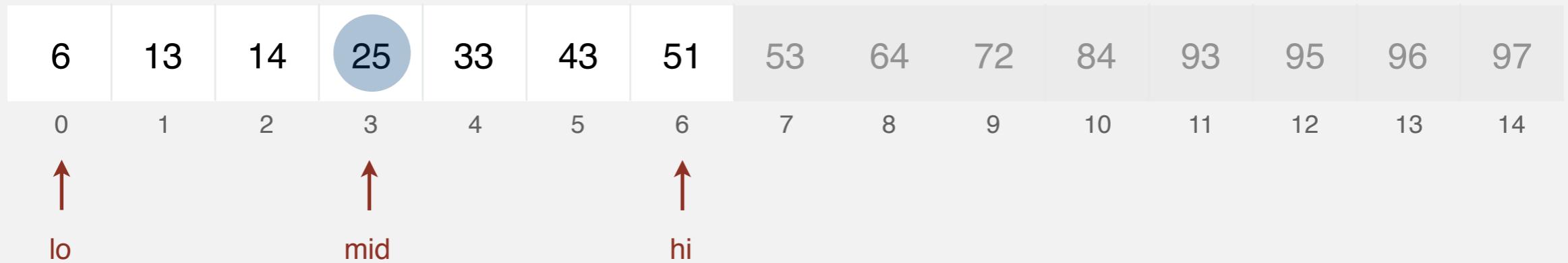
Successful search. Binary search for 33.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

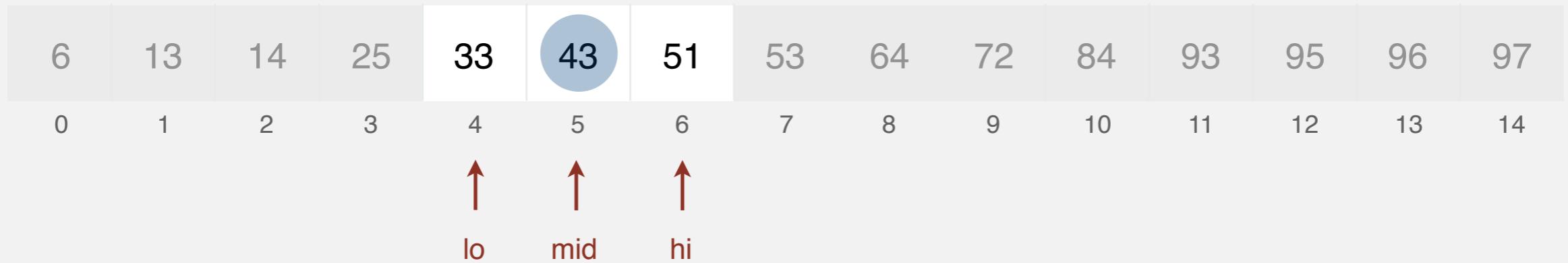
Successful search. Binary search for 33.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

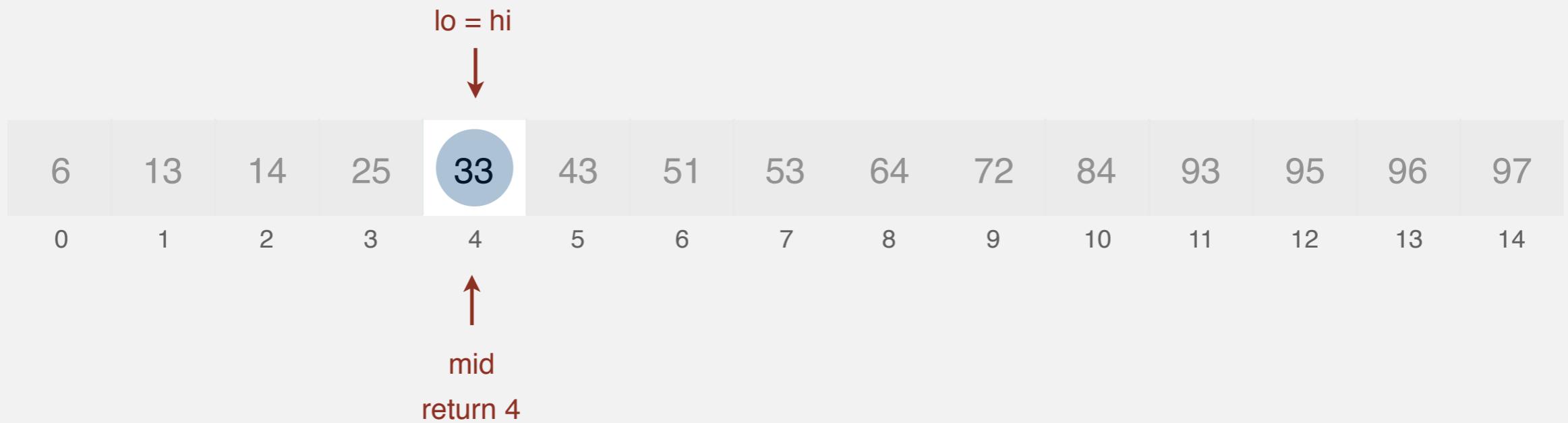
Successful search. Binary search for 33.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Successful search. Binary search for 33.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

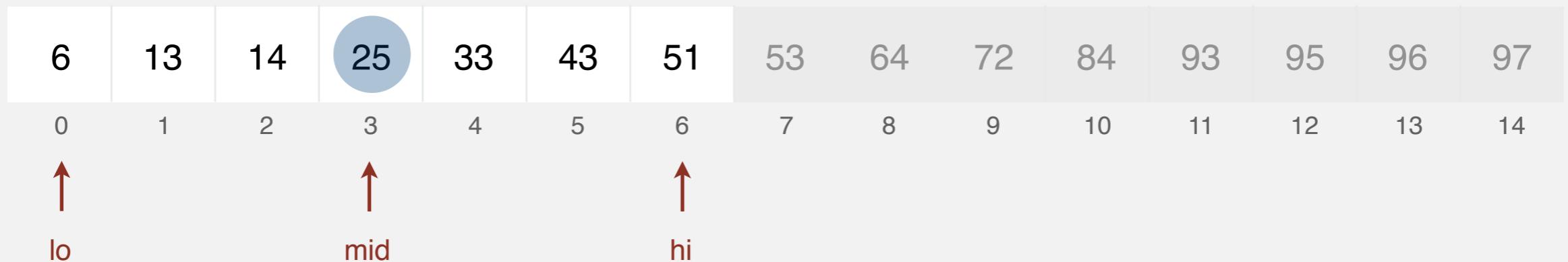
Unsuccessful search. Binary search for 34.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

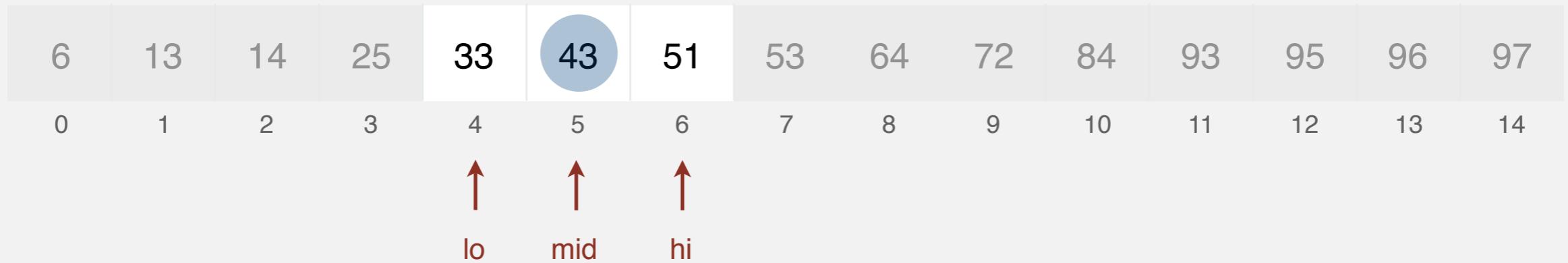
Unsuccessful search. Binary search for 34.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

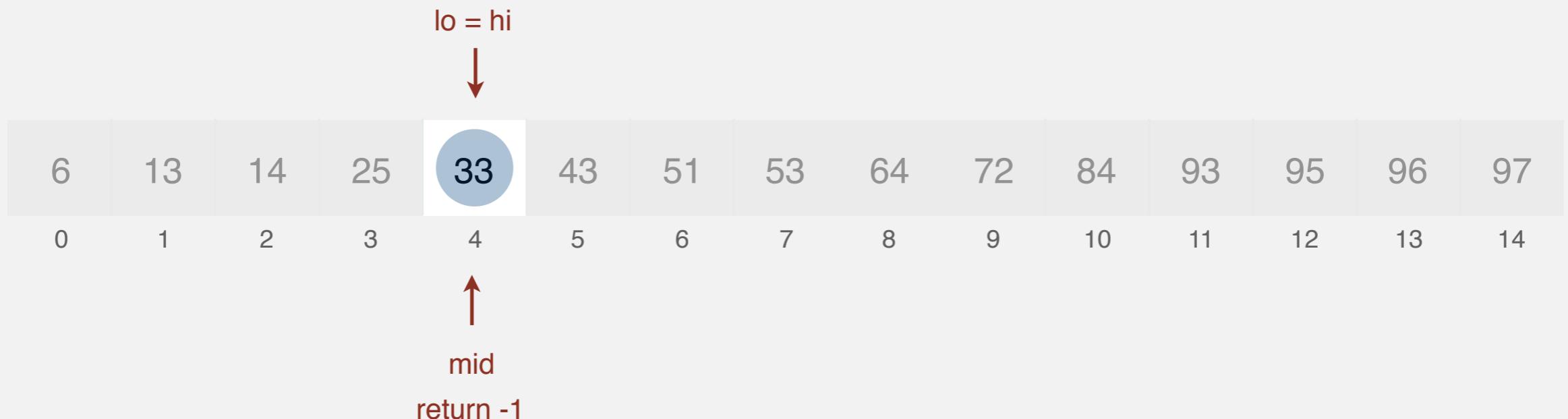
Unsuccessful search. Binary search for 34.



Binary search demo

Goal. Given a sorted array and a key, find index of the key in the array?

Unsuccessful search. Binary search for 34.



Binary search: Java implementation

Trivial to implement?

- First binary search published in 1946; first bug-free one published in 1962.
- Bug in Java's `Arrays.binarySearch()` discovered in 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

one "3-way compare"

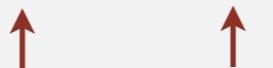
Invariant. If `key` appears in the array `a[]`, then `a[lo] ≤ key ≤ a[hi]`.

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ compares to search in a sorted array of size N .

Def. $T(N) \equiv \# \text{ compares to binary search in a sorted subarray of size at most } N$.

Binary search recurrence. $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.



left or right half

possible to implement with one
2-way compare (instead of 3-way)

Pf sketch.

$$\begin{aligned} T(N) &\leq T(N/2) + 1 \\ &\leq T(N/4) + 1 + 1 \\ &\leq T(N/8) + 1 + 1 + 1 \\ &\dots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 \\ &= 1 + \lg N \end{aligned}$$

given

apply recurrence to first term

apply recurrence to first term

stop applying, $T(1) = 1$

Binary search: mathematical analysis

Proposition. Binary search uses at most $1 + \lg N$ compares to search in a sorted array of size N .

Def. $T(N) \equiv \# \text{ compares to binary search in a sorted subarray of size at most } N$.

Binary search recurrence. $T(N) \leq T(\lfloor N/2 \rfloor) + 1$ for $N > 1$, with $T(0) = 0$.

For simplicity, we prove when $N = 2^n - 1$ for some n , so $\lfloor N/2 \rfloor = 2^{n-1} - 1$.

$$\begin{aligned} T(2^n - 1) &\leq T(2^{n-1} - 1) + 1 \\ &\leq T(2^{n-2} - 1) + 1 + 1 \\ &\leq T(2^{n-3} - 1) + 1 + 1 + 1 \\ &\quad \dots \\ &\leq T(2^0 - 1) + 1 + 1 + \dots + 1 \\ &= n \end{aligned}$$

given

apply recurrence to first term

apply recurrence to first term

stop applying, $T(0) = 1$

An $N^2 \log N$ algorithm for 3-sum

Algorithm.

- Sort the N (distinct) numbers.
- For each pair of numbers $a[i]$ and $a[j]$,
binary search for $-(a[i] + a[j])$.

Analysis. Order of growth is $N^2 \log N$.

- Step 1: N^2 with insertion sort.
- Step 2: $N^2 \log N$ with binary search.

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10 0 5 10 30 40

binary search

(-40, -20)	60
(-40, -10)	50
(-40, 0)	40
(-40, 5)	35
(-40, 10)	30
⋮	⋮
(-40, 40)	0
⋮	⋮
(-10, 0)	10
⋮	⋮
(-20, 10)	10
⋮	⋮
(10, 30)	-40
(10, 40)	-50
(30, 40)	-70

only count if
 $a[i] < a[j] < a[k]$
to avoid
double counting

Comparing programs

Hypothesis. The $N^2 \log N$ three-sum algorithm is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1.000	0,1
2.000	0,8
4.000	6,4
8.000	51,1

ThreeSum.java

N	time (seconds)
1.000	0,14
2.000	0,18
4.000	0,34
8.000	0,96
16.000	3,67
32.000	14,88
64.000	59,16

ThreeSumDeluxe.java

Guiding principle. Typically, better order of growth \Rightarrow faster in practice.

ANALYSIS OF ALGORITHMS

- ▶ **Observations**
- ▶ **Mathematical models**
- ▶ **Order-of-growth classifications**
- ▶ **Dependencies on inputs**
- ▶ **Memory**

Types of analyses

Best case. Lower bound on cost.

- Determined by “easiest” input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

- Need a model for “random” input.
- Provides a way to predict performance.

Ex 1. Array accesses for brute-force 3 sum.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Comparisons for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Types of analyses

Best case. Lower bound on cost.

Worst case. Upper bound on cost.

Average case. “Expected” cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.

Theory of Algorithms

Goals.

- Establish “difficulty” of a problem.
- Develop “optimal” algorithms.

Approach.

- Suppress details in analysis: analyze “to within a constant factor”.
- Eliminate variability in input model by focusing on the worst case.

Optimal algorithm.

- Performance guarantee (to within a constant factor) for any input.
- No algorithm can provide a better performance guarantee.

Commonly-used notations

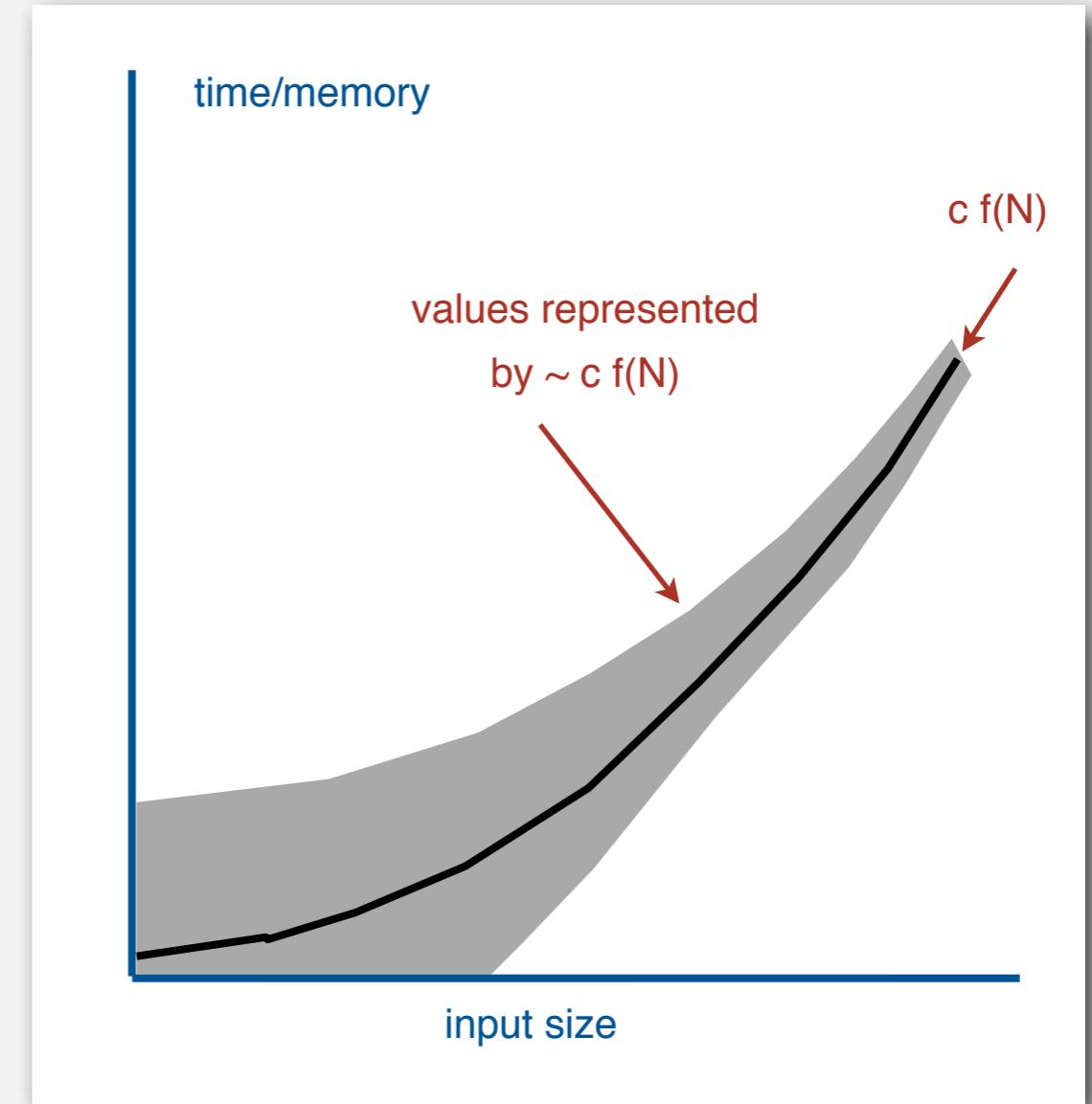
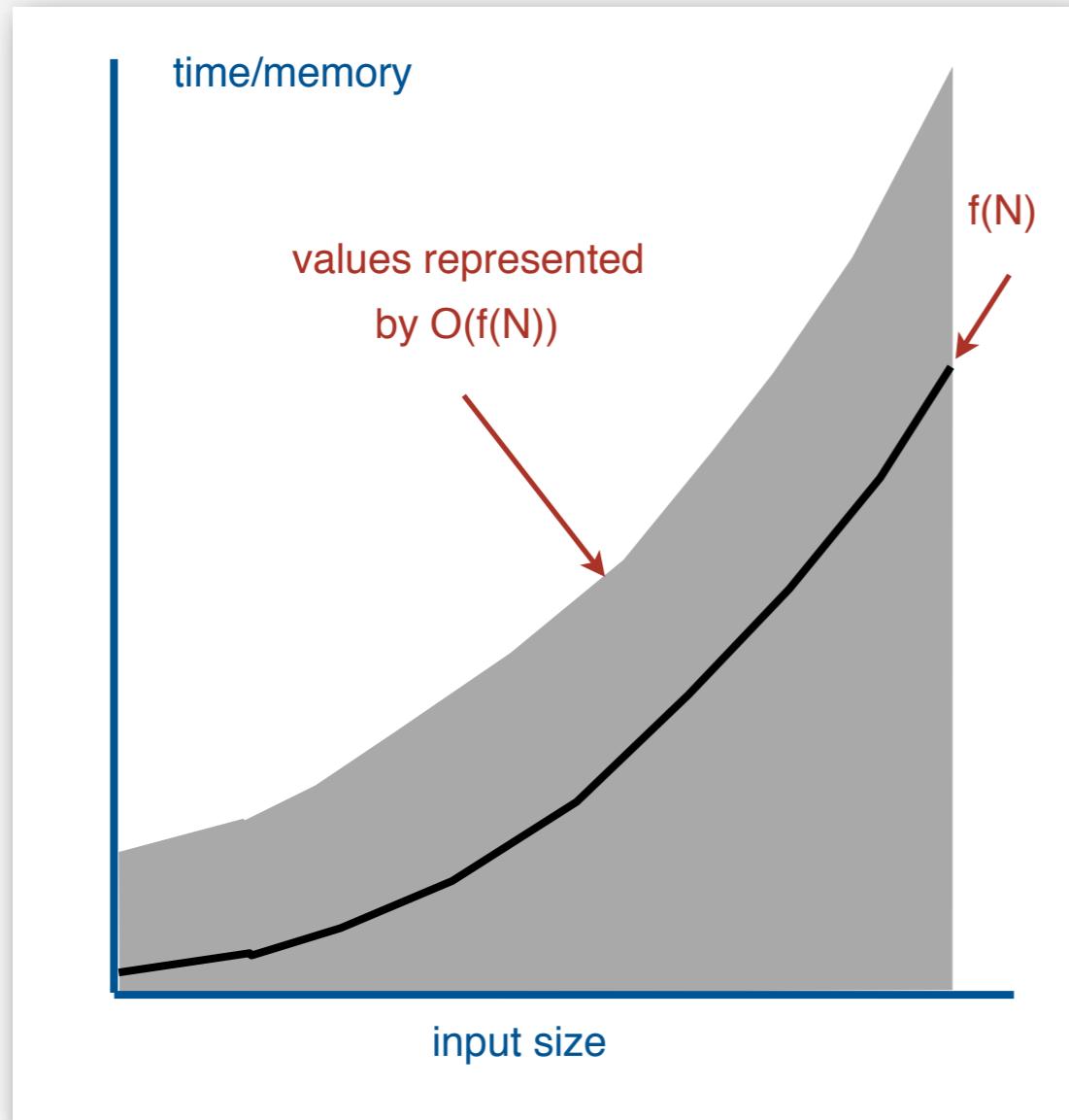
notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10 N^2$	$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$	develop lower bounds

Common mistake. Interpreting big-Oh as an approximate model.

Tilde notation vs. big-Oh notation

We use tilde notation whenever possible.

- Big-Oh notation suppresses leading constant.
- Big-Oh notation only provides upper bound (not lower bound).



Theory of algorithms: example I

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. I-SUM = “Is there a 0 in the array? ”

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for I-SUM: Look at every array entry.
- Running time of the optimal algorithm for I-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for I-SUM is $\Omega(N)$.

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for I-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM

Upper bound. A specific algorithm.

- Ex. Brute-force algorithm for 3-SUM
- Running time of the optimal algorithm for 3-SUM is $O(N^3)$.

Theory of algorithms: example 2

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM

Upper bound. A specific algorithm.

- Ex. Improved algorithm for 3-SUM
- Running time of the optimal algorithm for 3-SUM is $\mathcal{O}(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.

- Ex. Have to examine all N entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.

- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

- Lower the upper bound (discover a new algorithm).
- Raise the lower bound (more difficult).

Golden Age of Algorithm Design.

- 1970s-.
- Steadily decreasing upper bounds for many important problems.
- Many known optimal algorithms.

Caveats.

- Overly pessimistic to focus on worst case?
- Need better than “to within a constant factor” to predict performance.

ANALYSIS OF ALGORITHMS

- ▶ **Observations**
- ▶ **Mathematical models**
- ▶ **Order-of-growth classifications**
- ▶ **Dependencies on inputs**
- ▶ **Memory**

Basics

Bit. 0 or 1.

NIST

Byte. 8 bits.



most computer scientists

Megabyte (MB). 1 million or 2^{20} bytes.

Gigabyte (GB). 1 billion or 2^{30} bytes.



Old machine. We used to assume a 32-bit machine with 4 byte pointers.

Modern machine. We now assume a 64-bit machine with 8 byte pointers.

- Can address more memory.
- Pointers use more space.



some JVMs "compress" ordinary object
pointers to 4 bytes to avoid this cost

Typical memory usage for primitive types and arrays

Primitive types.

type	bytes
boolean	1
byte	1
char	2
int	4
float	4
long	8
double	8

for primitive types

Array overhead. 24 bytes.

type	bytes
char []	$2N + 24$
int []	$4N + 24$
double []	$8N + 24$

for one-dimensional arrays

type	bytes
char [] []	$\sim 2 MN$
int [] []	$\sim 4 MN$
double [] []	$\sim 8 MN$

for two-dimensional arrays

Typical memory usage for objects in Java

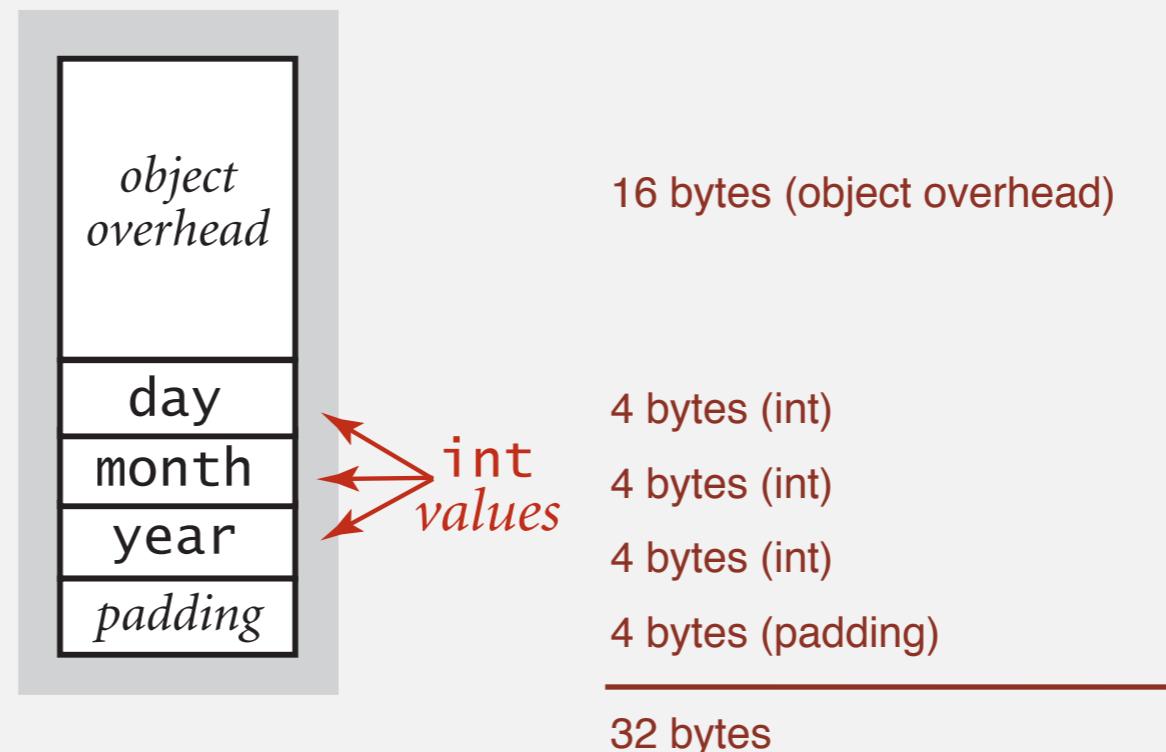
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 1. A `Date` object uses 32 bytes of memory.

```
public class Date
{
    private int day;
    private int month;
    private int year;
    ...
}
```



Typical memory usage for objects in Java

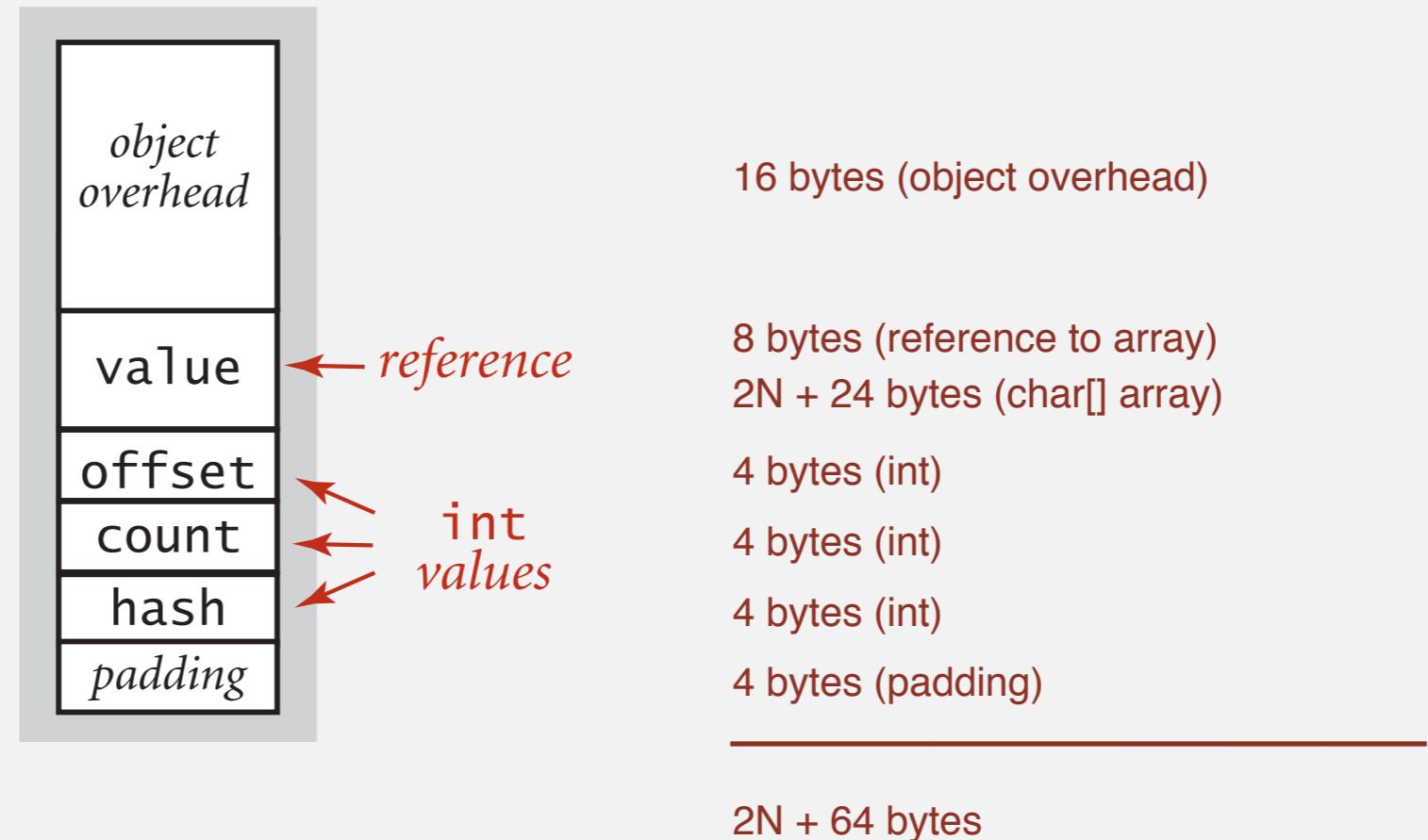
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

Ex 2. A virgin `String` of length N uses $\sim 2N$ bytes of memory.

```
public class String
{
    private char[] value;
    private int offset;
    private int count;
    private int hash;
    ...
}
```



Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable + 8 if inner class.

padding: round up
to multiple of 8

extra pointer to
enclosing class

Shallow memory usage: Don't count referenced objects.

Deep memory usage: If array entry or instance variable is a reference,
add memory (recursively) for referenced object.

Memory profiler

Classmexer library. Measure memory usage of a Java object by querying JVM.

<http://www.javamex.com/classmexer>

```
import com.javamex.classmexer.MemoryUtil;

public class Memory {
    public static void main(String[] args) {
        Date date = new Date(12, 31, 1999);
        StdOut.println(MemoryUtil.memoryUsageOf(date));
        String s = "Hello, World";
        StdOut.println(MemoryUtil.memoryUsageOf(s));
        StdOut.println(MemoryUtil.deepMemoryUsageOf(s))
    }
}
```

```
% javac -cp .:classmexer.jar Memory.java  
% java -cp .:classmexer.jar -javaagent:classmexer.jar Memory  
32  
40 ← don't count char[]  
88 ← 2N + 64
```

use `-XX:-UseCompressedOops`
on OS X to match our model

Turning the crank: summary

Empirical analysis.

- Execute program to perform experiments.
- Assume power law and formulate a hypothesis for running time.
- Model enables us to **make predictions**.

Mathematical analysis.

- Analyze algorithm to count frequency of operations.
- Use tilde notation to simplify analysis.
- Model enables us to **explain behaviour**.



Scientific method.

- Mathematical model is independent of a particular system; applies to machines not yet built.
- Empirical analysis is necessary to validate mathematical models and to make predictions.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ELEMENTARY SORTING ALGORITHMS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

ELEMENTARY SORTING ALGORITHMS

- ▶ **Sorting review**
- ▶ **Rules of the game**
- ▶ **Selection sort**
- ▶ **Insertion sort**
- ▶ **Shellsort**

ELEMENTARY SORTING ALGORITHMS

- ▶ **Sorting review**
- ▶ **Rules of the game**
- ▶ Selection sort
- ▶ Insertion sort
- ▶ Shellsort

Sorting problem

Ex. Student records in a university.

Chen	3	A	991-878-4944	308 Blair
Rohde	2	A	232-343-5555	343 Forbes
Gazsi	4	B	766-093-9873	101 Brown
Furia	1	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman

Sort. Rearrange array of N items into ascending order.

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

Sample sort client

Goal. Sort any type of data.

Ex 1. Sort random real numbers in ascending order.

seems artificial, but stay tuned for an application

```
public class Experiment
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Double[] a = new Double[N];
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
        Insertion.sort(a);
        for (int i = 0; i < N; i++)
            StdOut.println(a[i]);
    }
}
```

```
% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
```

Sample sort client

Goal. Sort **any** type of data.

Ex 2. Sort strings from file in alphabetical order.

```
public class StringSorter
{
    public static void main(String[] args)
    {
        String[] a = In.readStrings(args[0]);
        Insertion.sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.println(a[i]);
    }
}
```

```
% more words3.txt
bed bug dad yet zoo ... all bad yes
```

```
% java StringSorter words3.txt
all bad bed bug dad ... yes yet zoo
```

Sample sort client

Goal. Sort **any** type of data.

Ex 3. Sort the files in a given directory by filename.

```
import java.io.File;
public class FileSorter
{
    public static void main(String[] args)
    {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOut.println(files[i].getName());
    }
}
```

```
% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
```

Callbacks

Goal. Sort **any** type of data.

Q. How can `sort()` know how to compare data of type `Double`, `String`, and `java.io.File` without any information about the type of an item's key?

Callback = reference to executable code.

- Client passes array of objects to `sort()` function.
- The `sort()` function calls back object's `compareTo()` method as needed.

Implementing callbacks.

- Java: `interfaces`.
- C: function pointers.
- C++: class-type functors.
- C#: delegates.
- Python, Perl, ML, Javascript: first-class functions.

Callbacks: roadmap

client

```
import java.io.File;
public class FileSorter
{
    public static void main(String[] args)
    {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOut.println(files[i].getName());
    }
}
```

object implementation

```
public class File
    implements Comparable<File>
{
    ...
    public int compareTo(File b)
    {
        ...
        return -1;
        ...
        return +1;
        ...
        return 0;
    }
}
```

Comparable interface (built in to Java)

```
public interface Comparable<Item>
{
    public int compareTo(Item that);
}
```

key point: no dependence
on `File` data type

sort implementation

```
public static void sort(Comparable[] a)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0; j--)
            if (a[j].compareTo(a[j-1]) < 0)
                exch(a, j, j-1);
            else break;
}
```



Total order

A **total order** is a binary relation \leq that satisfies

- Antisymmetry: if $v \leq w$ and $w \leq v$, then $v = w$.
- Transitivity: if $v \leq w$ and $w \leq x$, then $v \leq x$.
- Totality: either $v \leq w$ or $w \leq v$ or both.

Ex.

- Standard order for natural and real numbers.
- Alphabetical order for strings.
- Chronological order for dates.
- ...

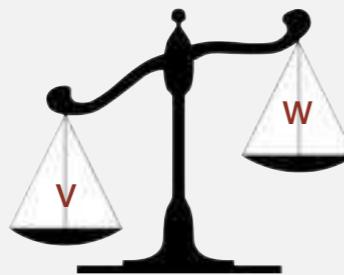


an intransitive relation

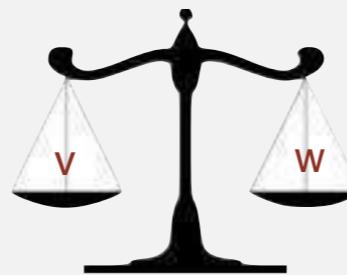
Comparable API

Implement `compareTo()` so that `v.compareTo(w)`

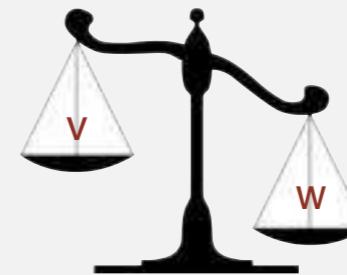
- Is a total order.
- Returns a negative integer, zero, or positive integer if v is less than, equal to, or greater than w , respectively.
- Throws an exception if incompatible types (or either is `null`).



less than (return -1)



equal to (return 0)



greater than (return +1)

Built-in comparable types. `Integer`, `Double`, `String`, `Date`, `File`, ...

User-defined comparable types. Implement the `Comparable` interface.

Implementing the Comparable interface

Date data type. Simplified version of `java.util.Date`.

```
public class Date implements Comparable<Date>
{
    private final int month, day, year;

    public Date(int m, int d, int y)
    {
        month = m;
        day   = d;
        year  = y;
    }

    public int compareTo(Date that)
    {
        if (this.year < that.year) return -1;
        if (this.year > that.year) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day < that.day) return -1;
        if (this.day > that.day) return +1;
        return 0;
    }
}
```

only compare dates
to other dates

Two useful sorting abstractions

Helper functions. Refer to data through compares and exchanges.

Less. Is item v less than w ?

```
private static boolean less(Comparable v, Comparable w)
{   return v.compareTo(w) < 0; }
```

Exchange. Swap item in array $a[]$ at index i with the one at index j .

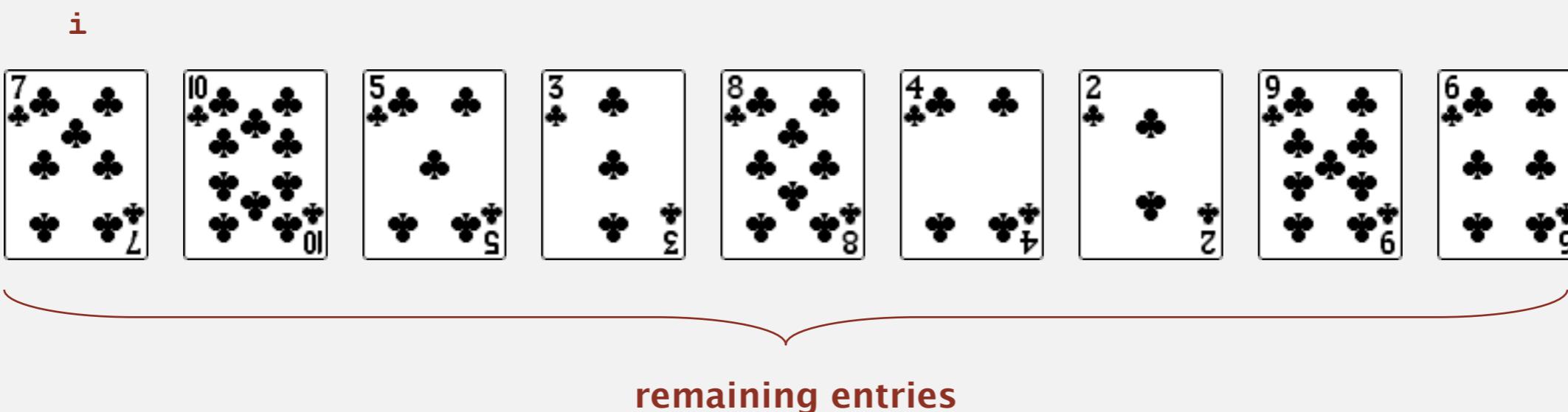
```
private static void exch(Comparable[] a, int i, int j)
{
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

ELEMENTARY SORTING ALGORITHMS

- ▶ **Sorting review**
- ▶ Rules of the game
- ▶ Selection sort
- ▶ Insertion sort
- ▶ Shellsort

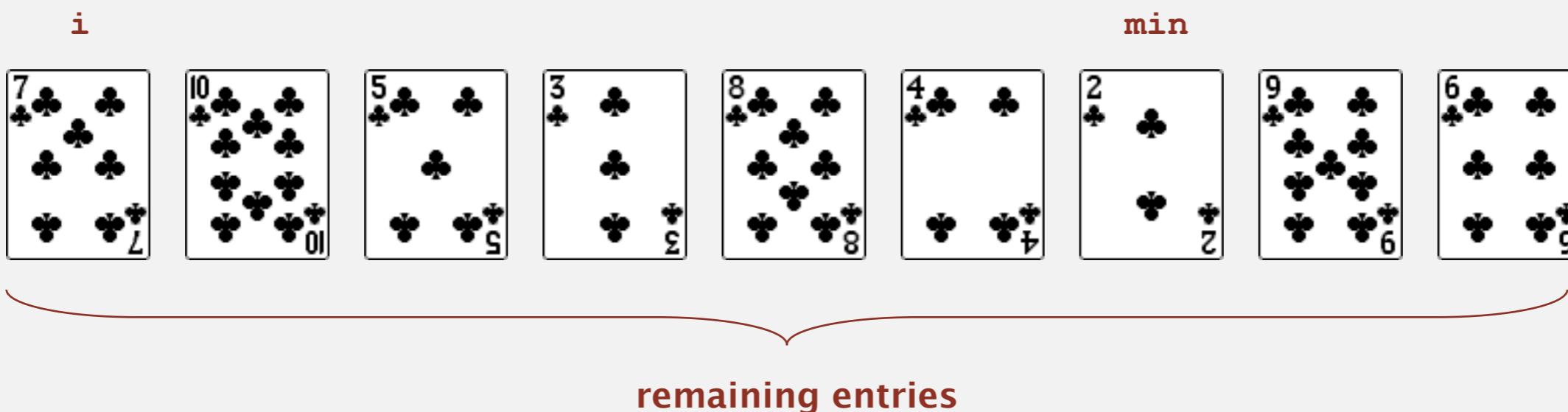
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



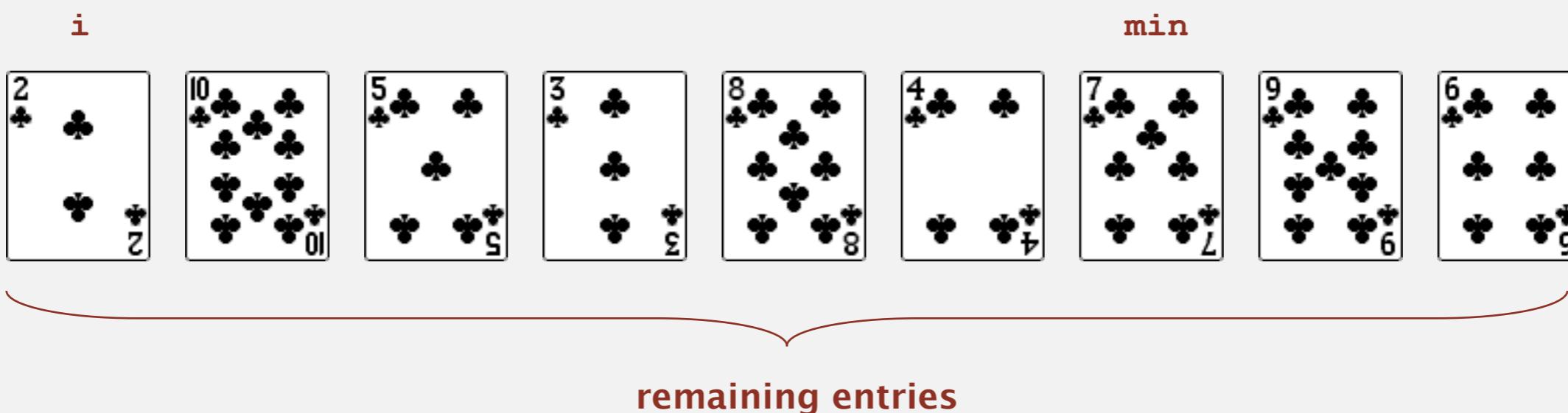
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



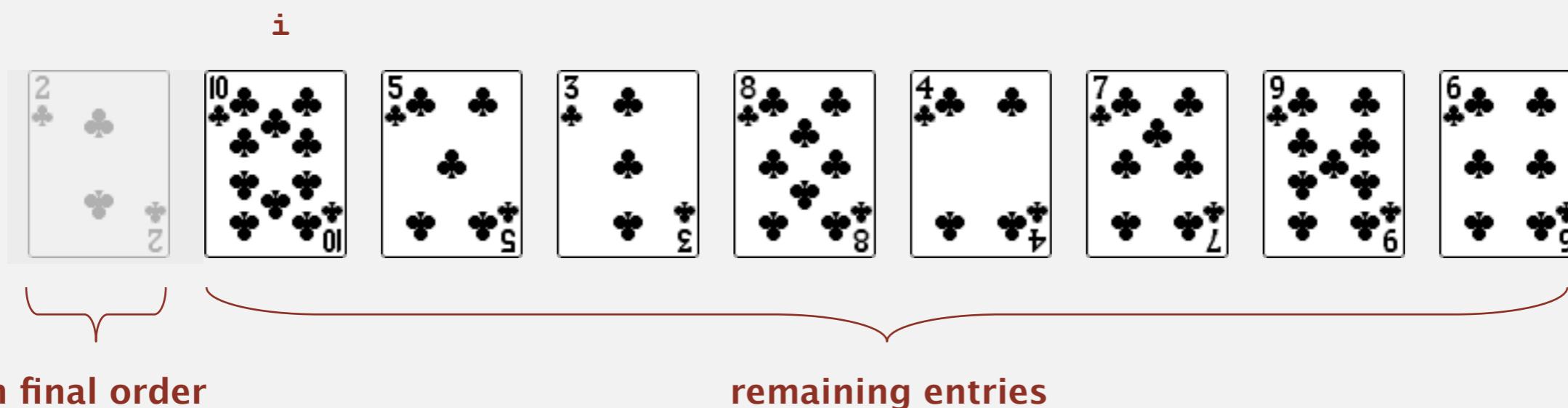
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



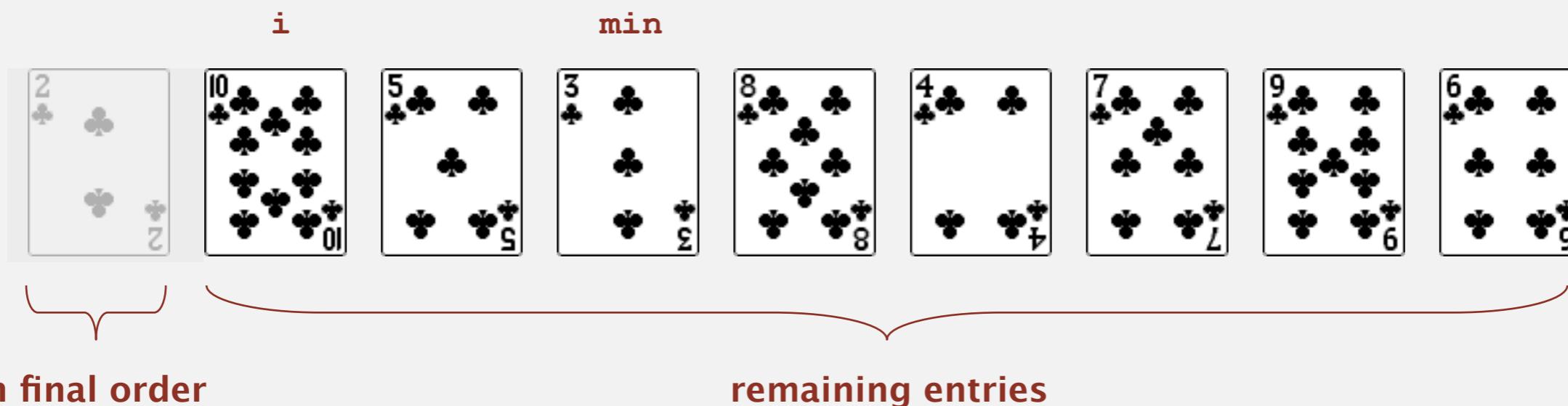
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



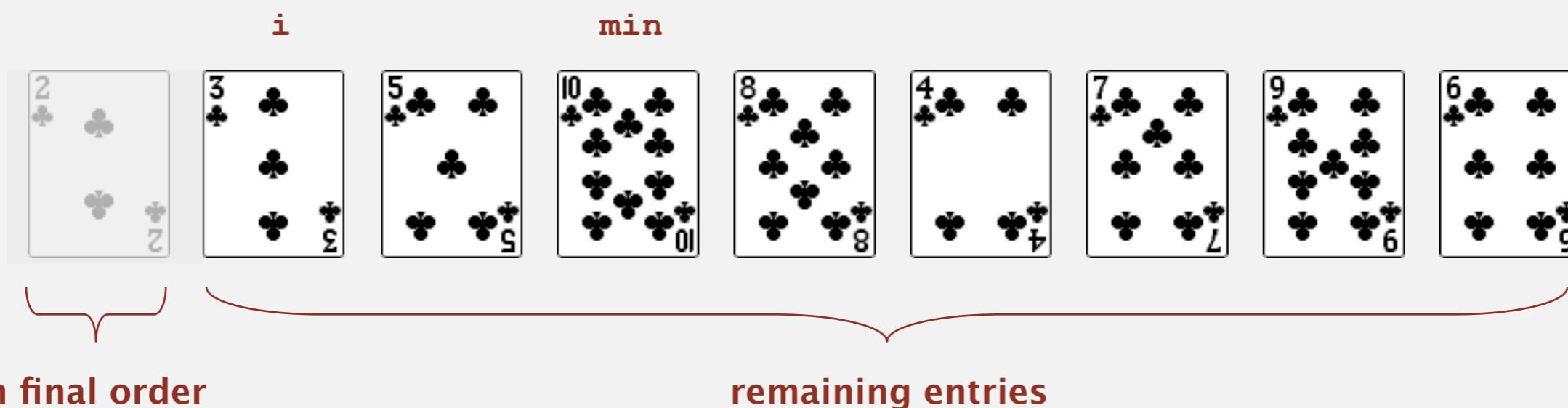
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



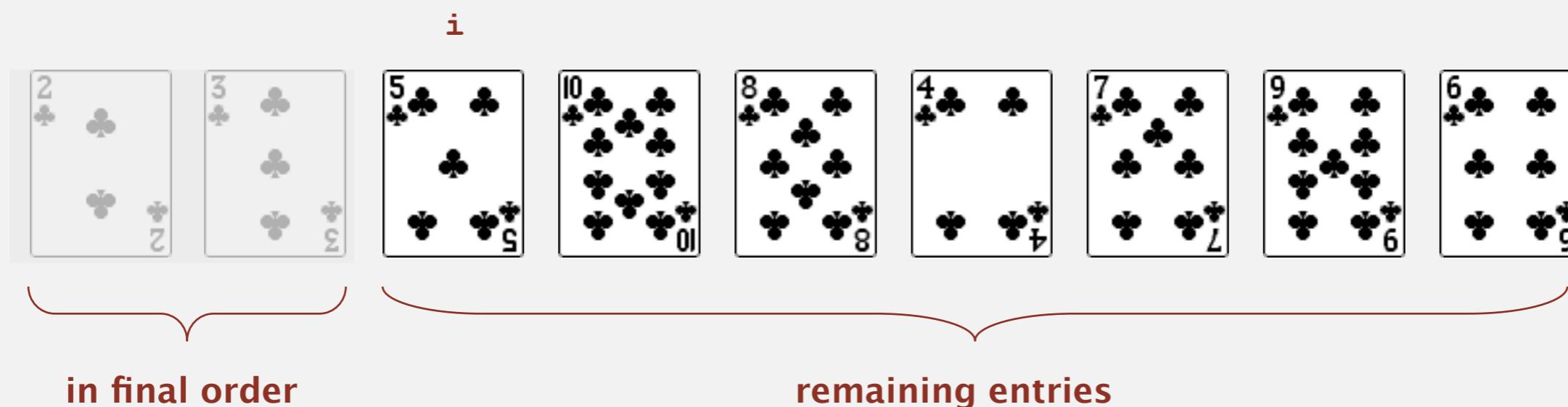
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



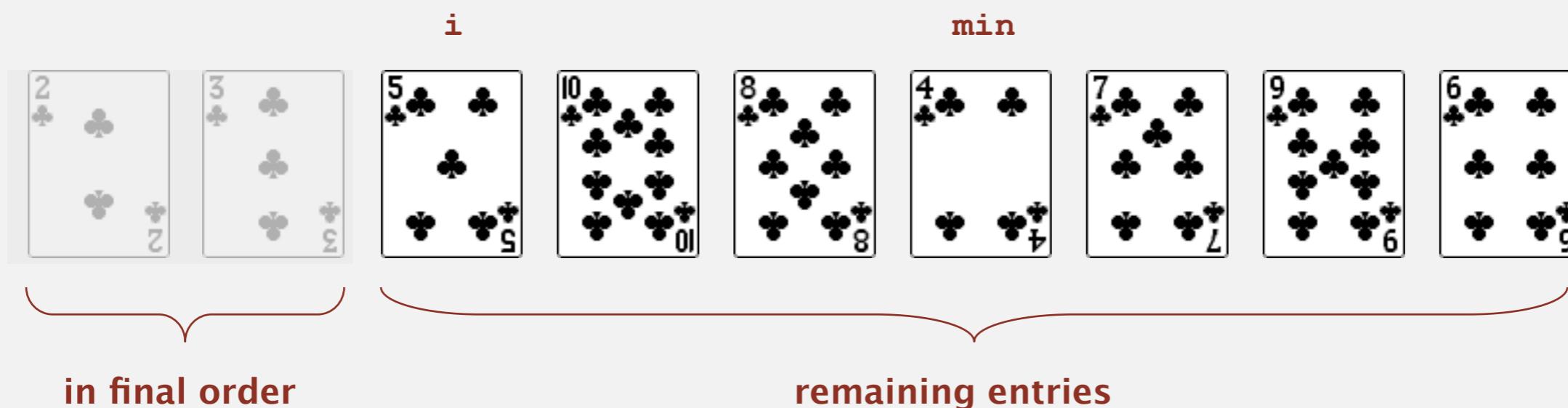
Selection sort

- In iteration i , find index min of smallest remaining entry.
 - Swap $a[i]$ and $a[\text{min}]$.



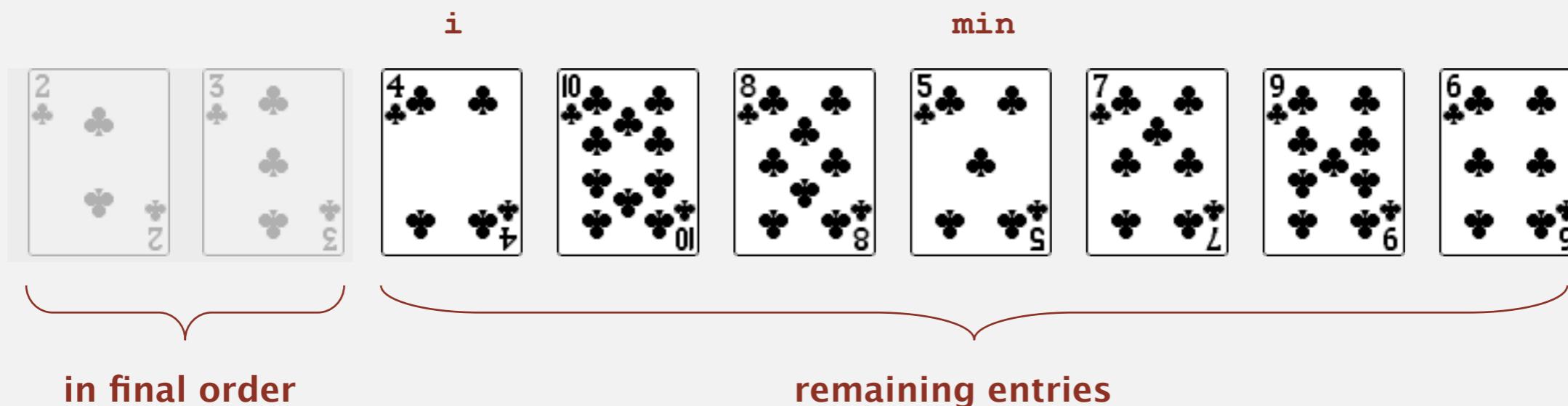
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



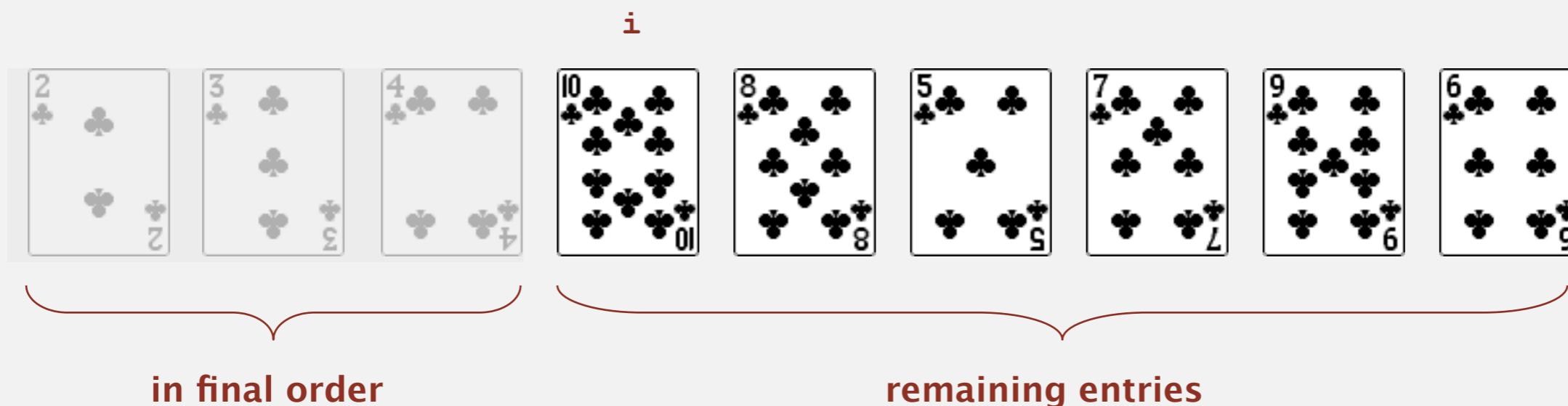
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



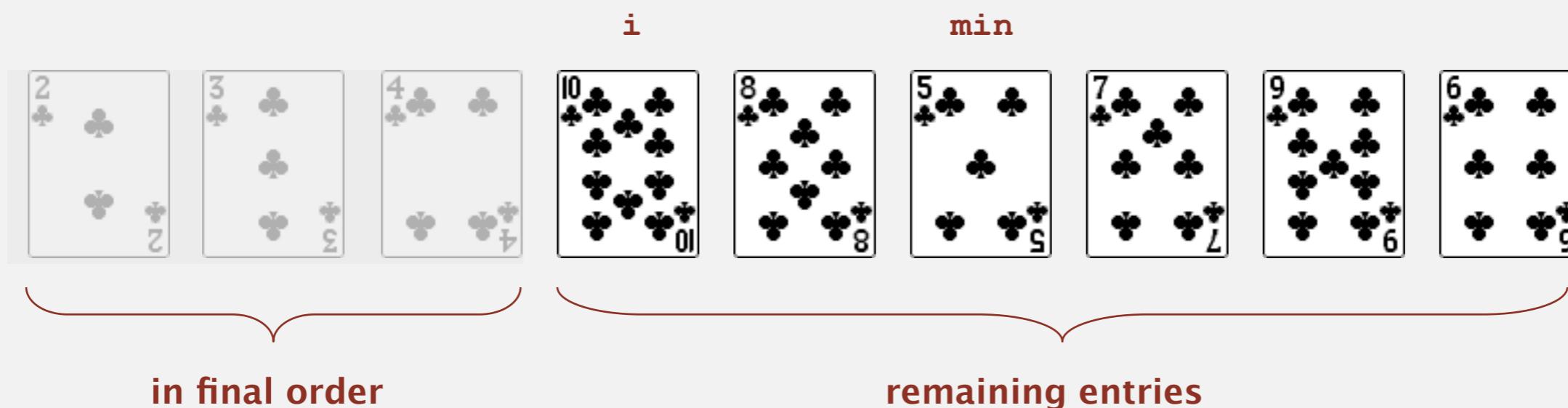
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



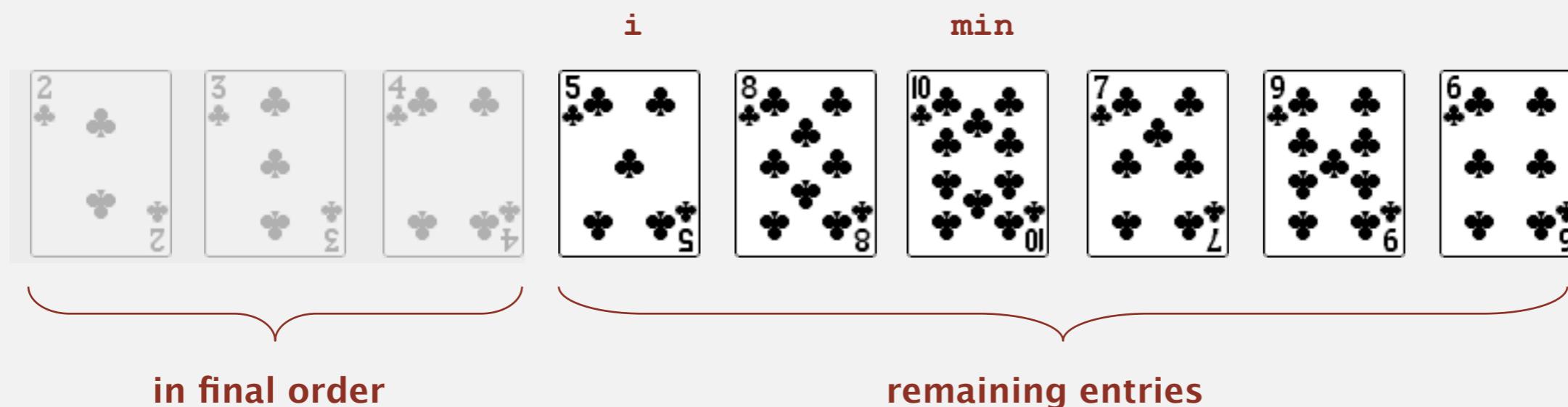
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



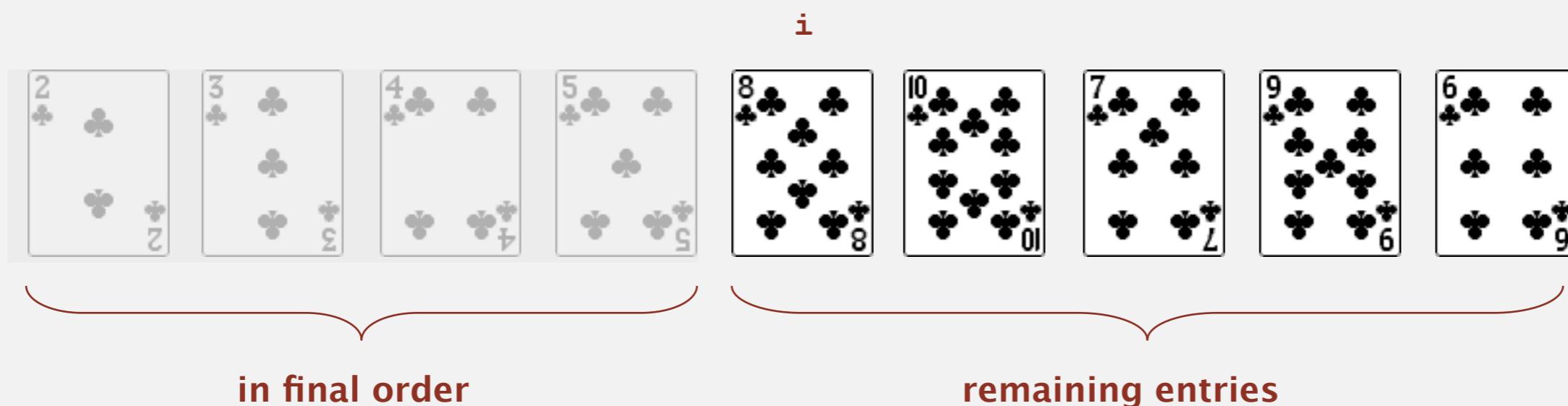
Selection sort

- In iteration i , find index min of smallest remaining entry.
 - Swap $a[i]$ and $a[\text{min}]$.



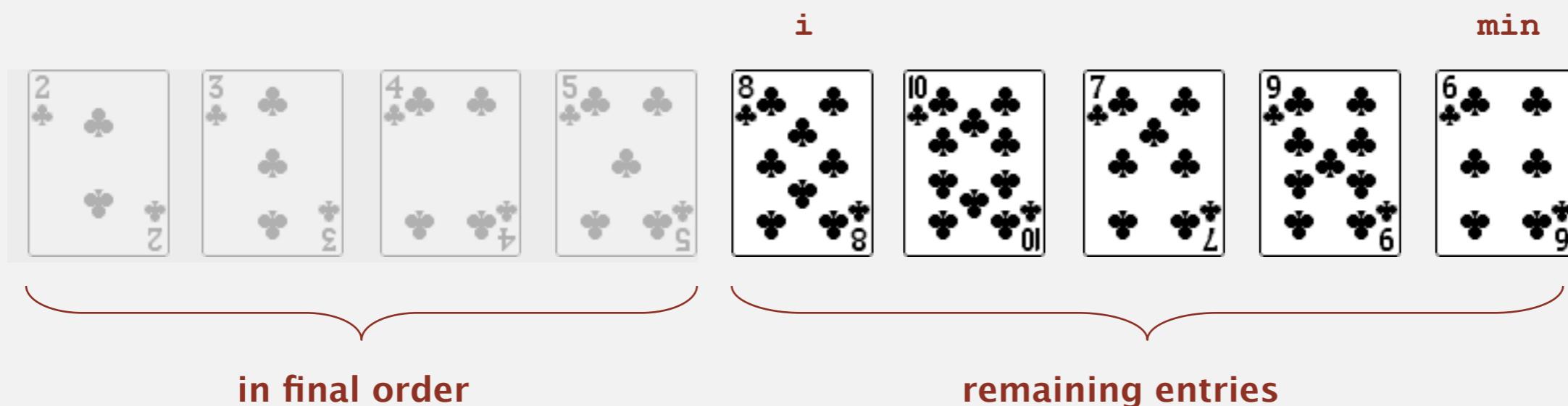
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



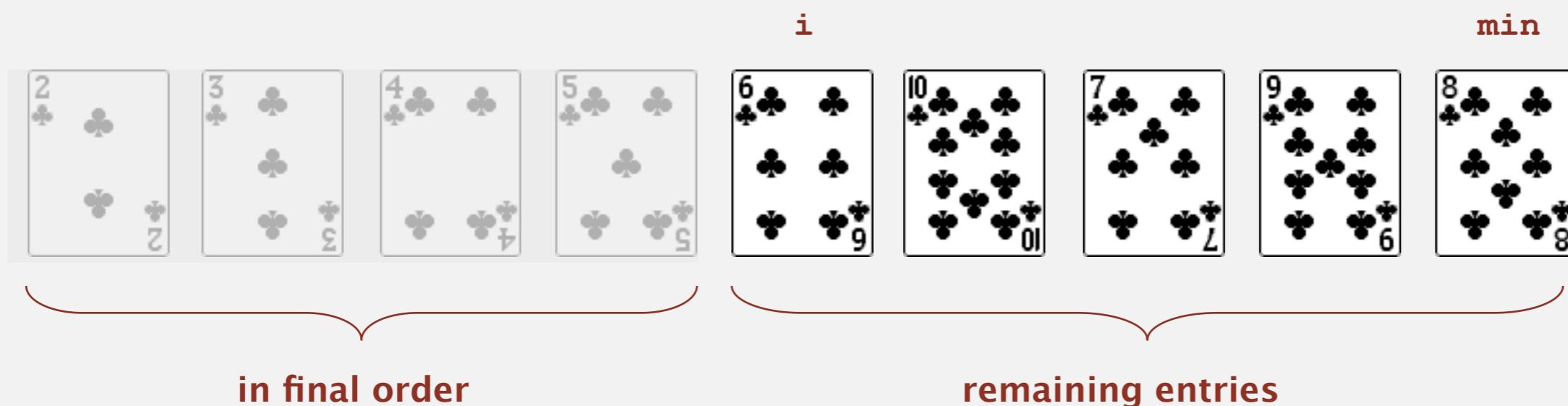
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



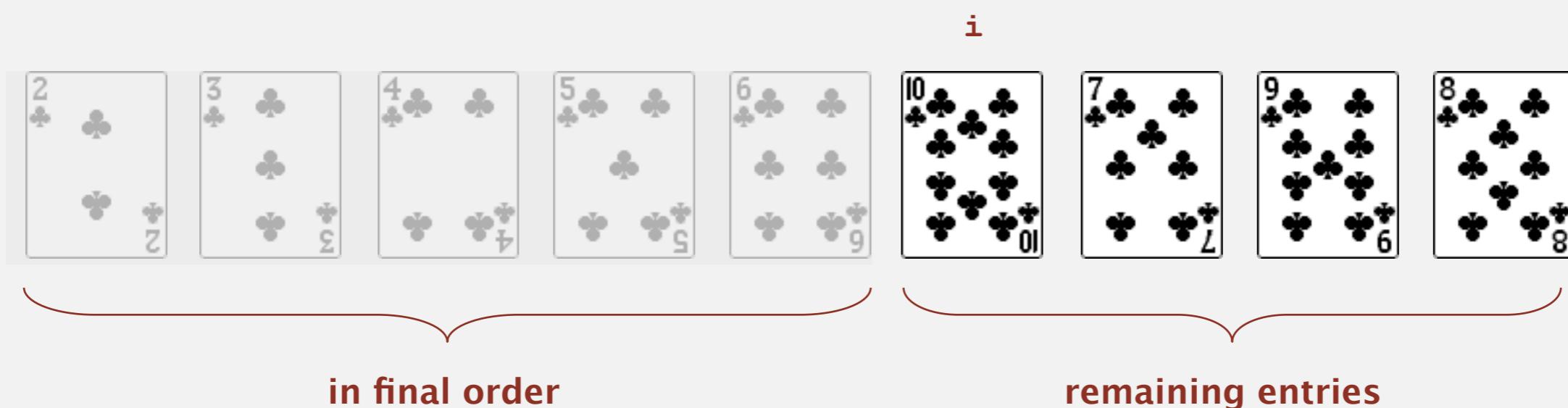
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



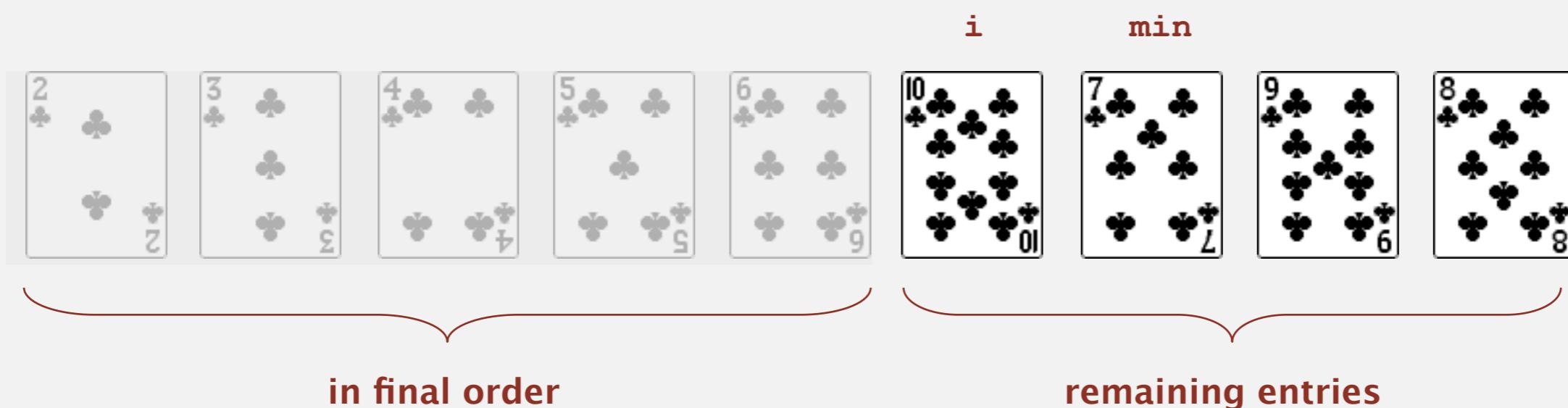
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



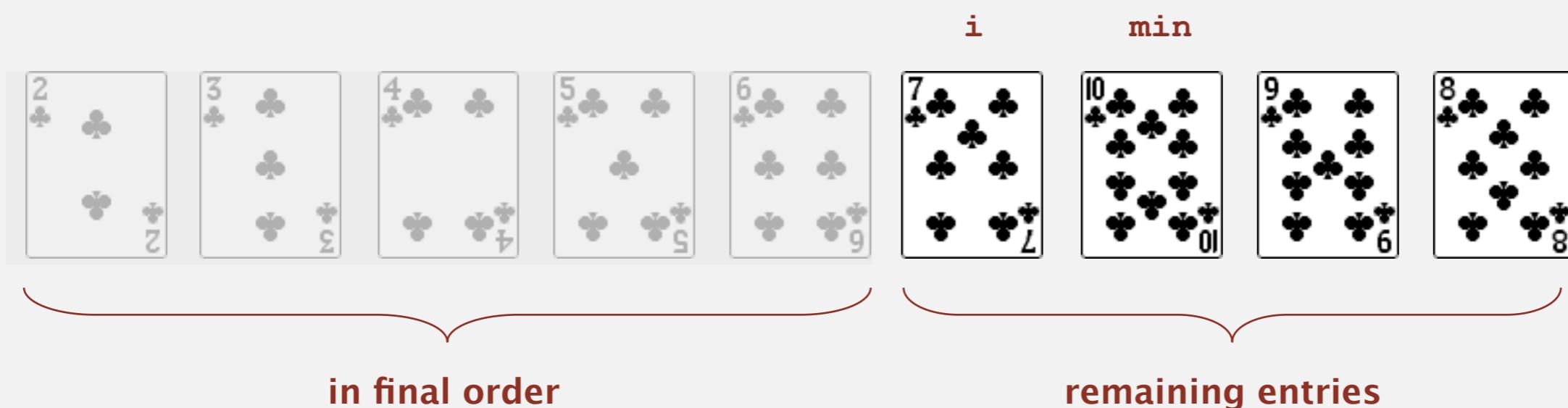
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



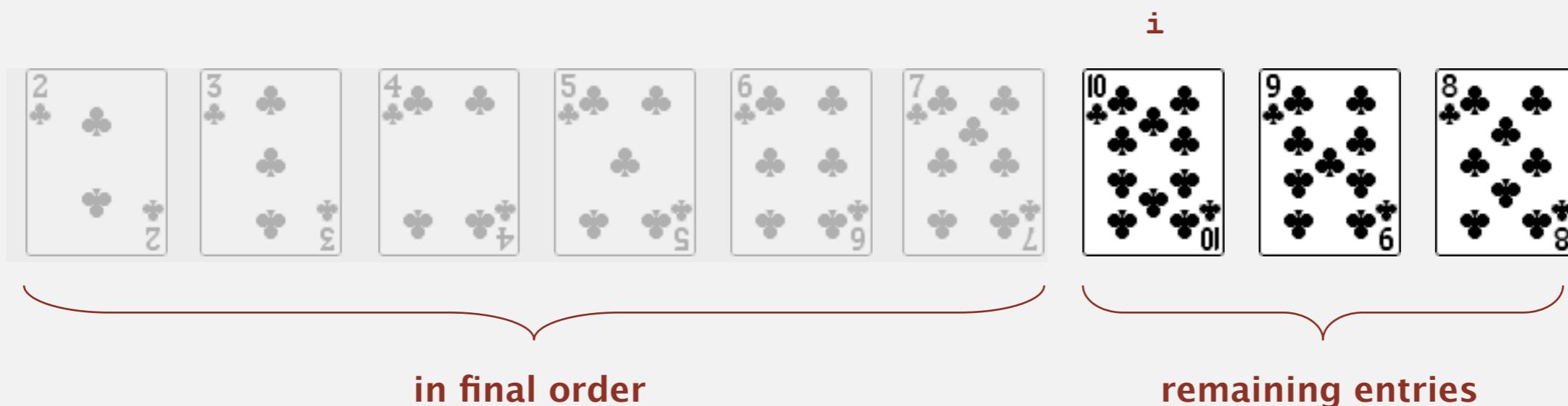
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



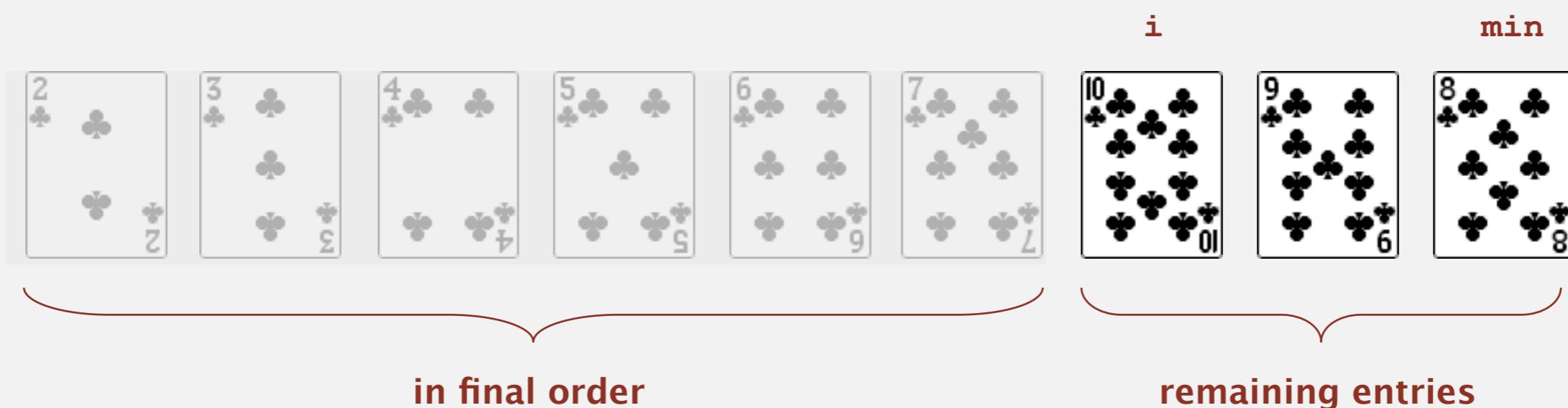
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



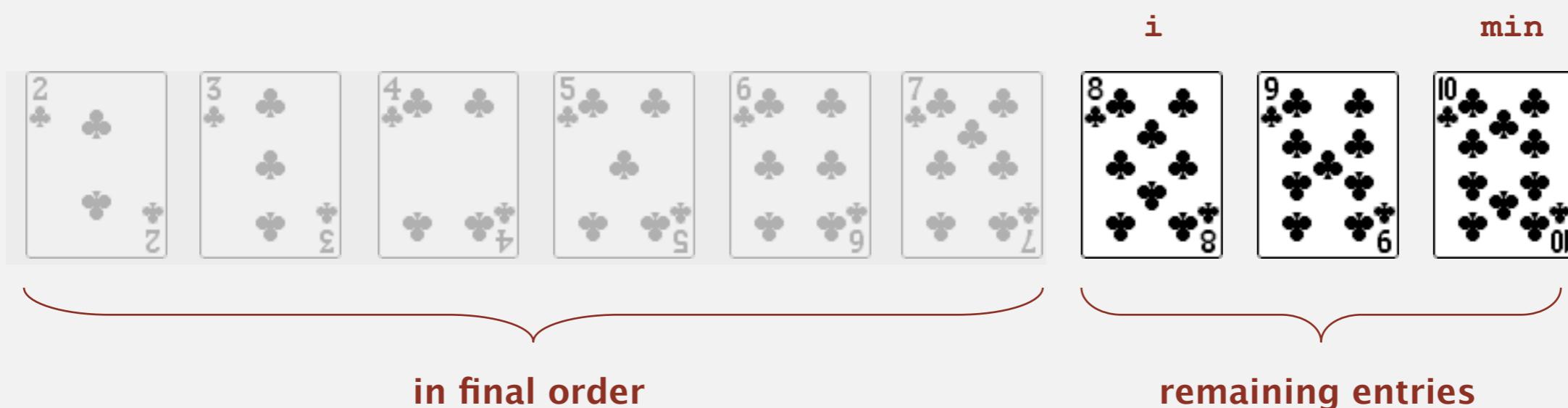
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



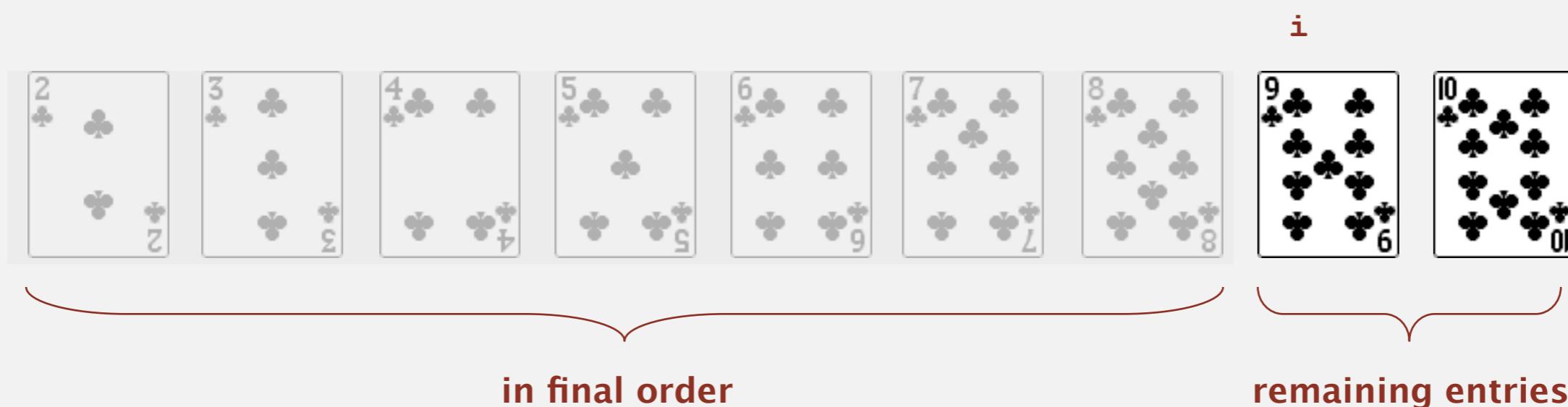
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



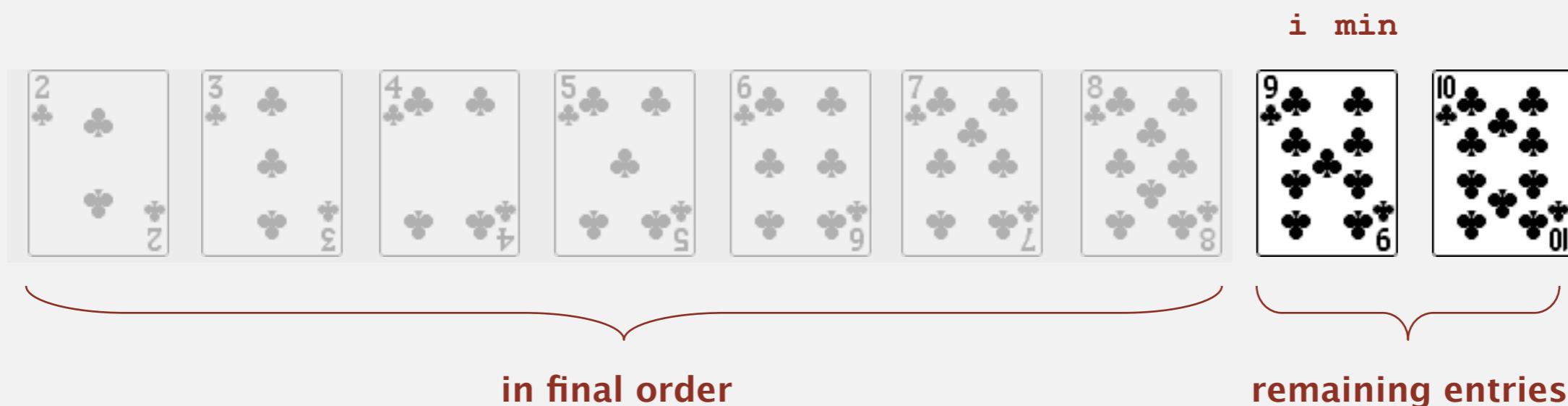
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



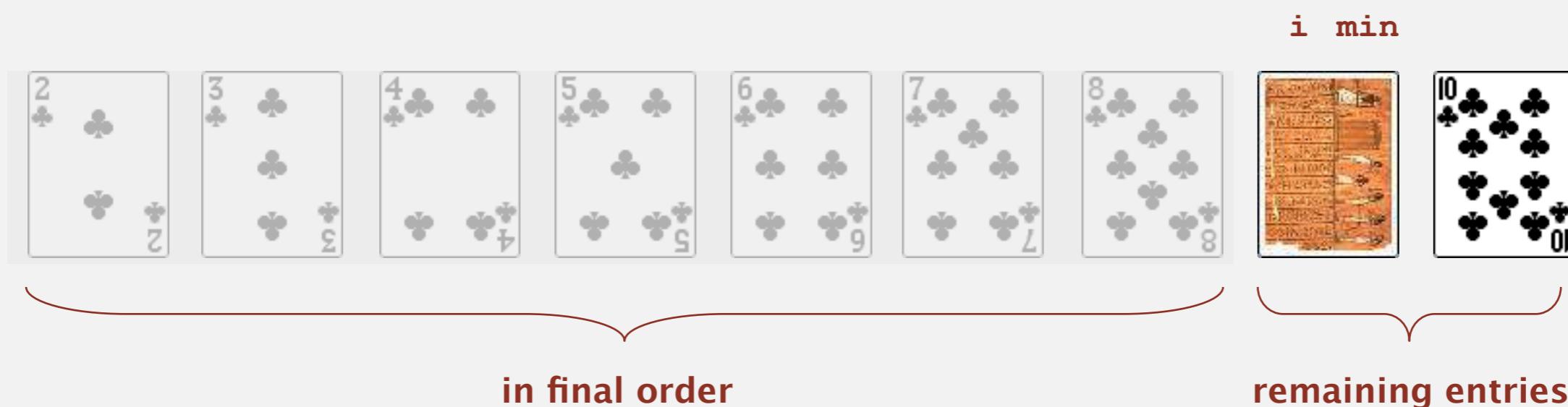
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



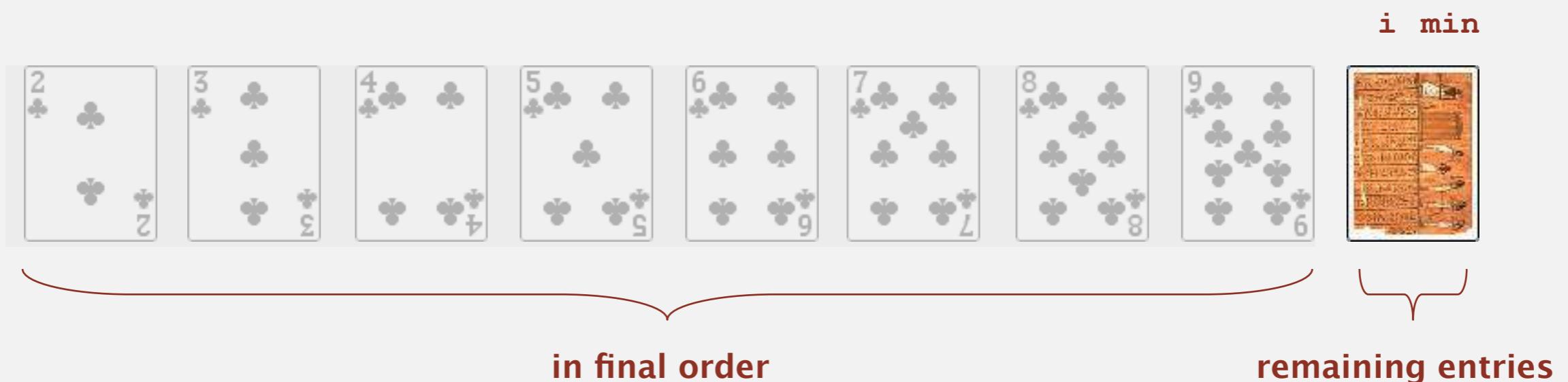
Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



Selection sort

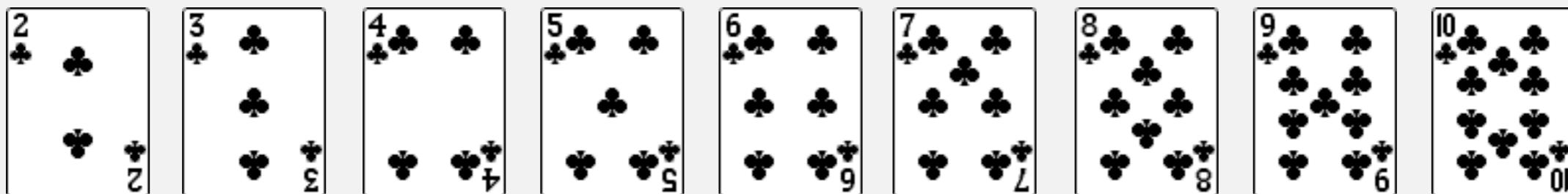
- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



in final order

Selection sort

- In iteration i , find index min of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.



sorted

Selection sort: Java implementation

```
public class Selection
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
        {
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
        }
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
```

Selection sort: mathematical analysis

Proposition. Selection sort uses $(N-1) + (N-2) + \dots + 1 + 0 \sim N^2/2$ compares and N exchanges.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

Trace of selection sort (array contents just after each exchange)

entries in black are examined to find the minimum

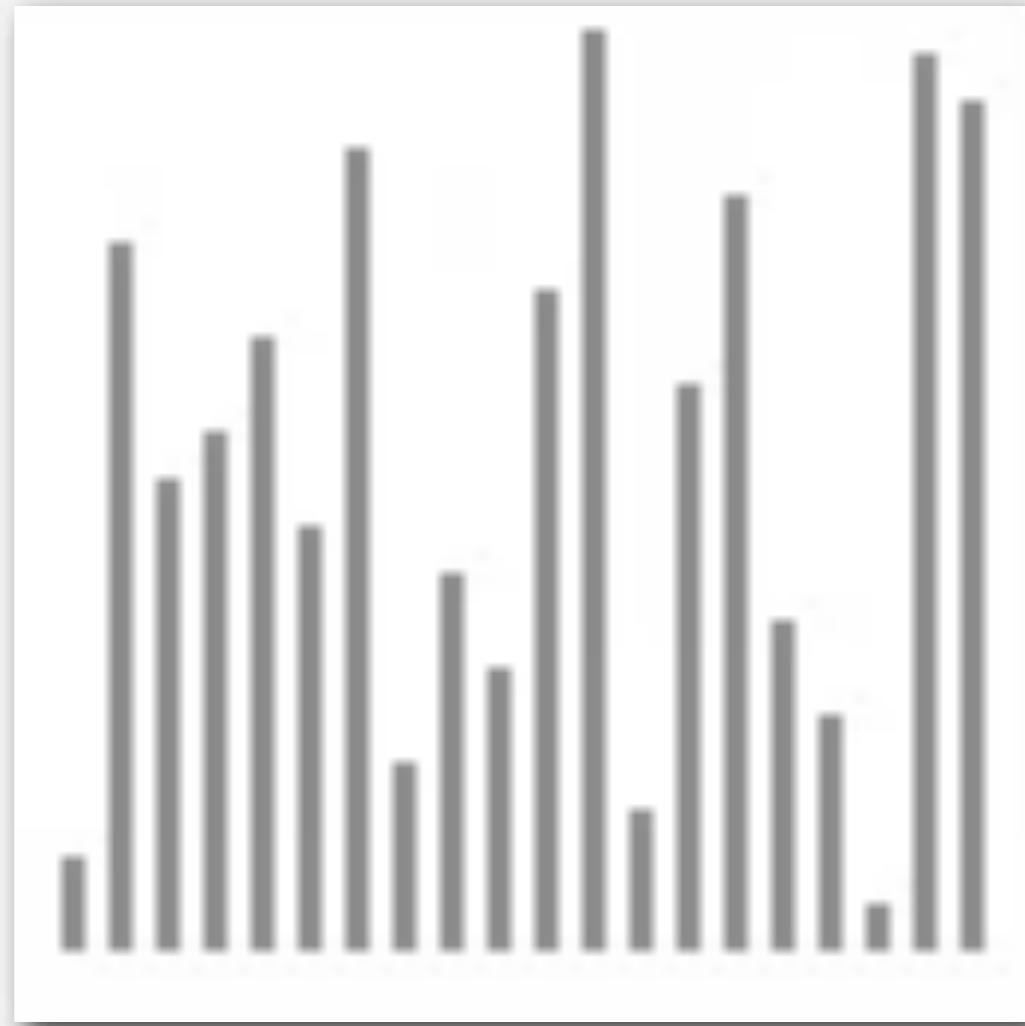
entries in red are $a[min]$

entries in gray are in final position

Running time insensitive to input. Quadratic time, even if input array is sorted.
Data movement is minimal. Linear number of exchanges.

Selection sort: animations

20 random items



algorithm position

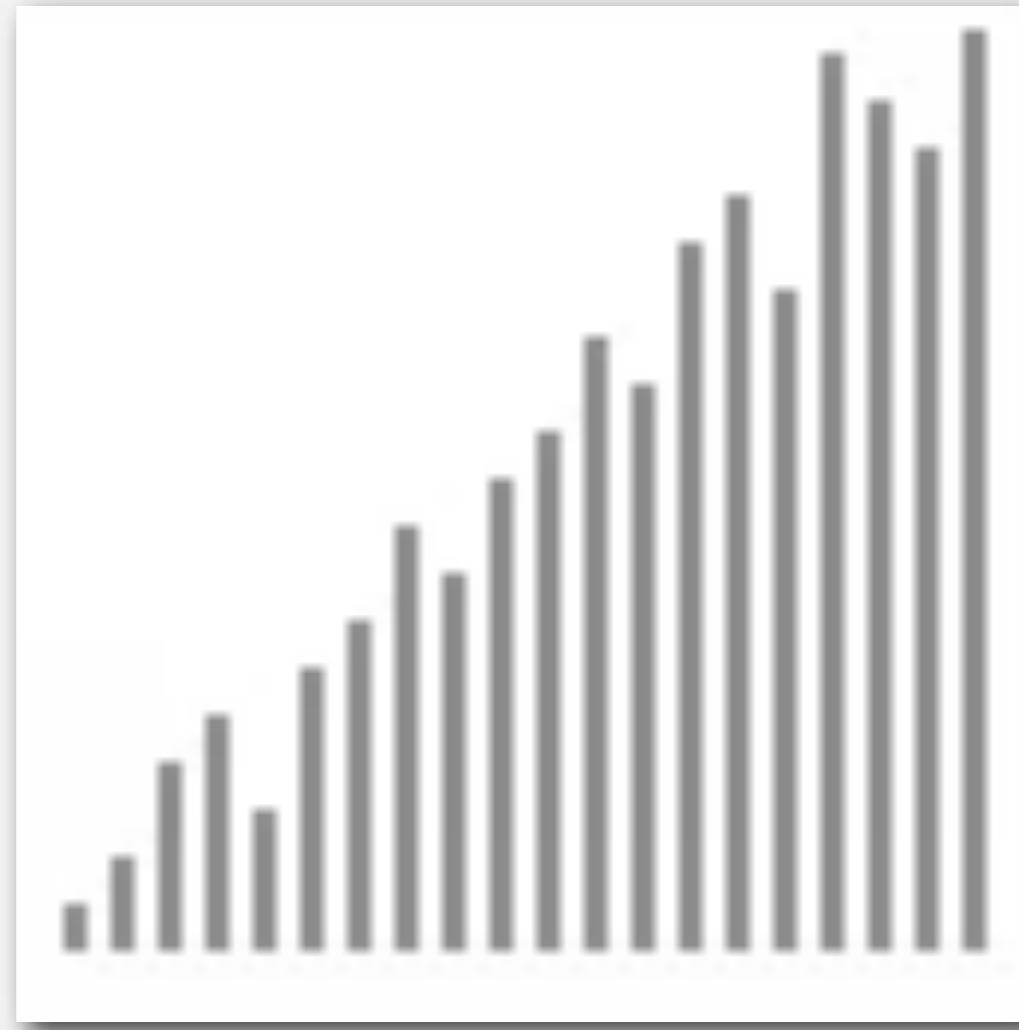
in final order

not in final order

<http://www.sorting-algorithms.com/selection-sort>

Selection sort: animations

20 partially-sorted items



algorithm position

in final order

not in final order

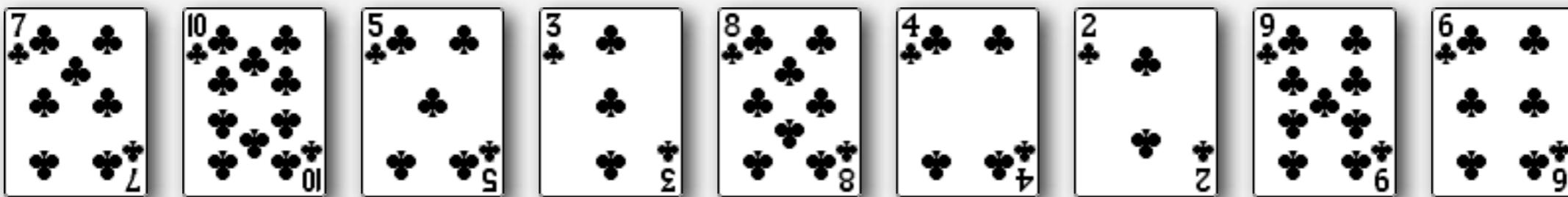
<http://www.sorting-algorithms.com/selection-sort>

ELEMENTARY SORTING ALGORITHMS

- ▶ **Sorting review**
- ▶ **Rules of the game**
- ▶ **Selection sort**
- ▶ **Insertion sort**
- ▶ **Shellsort**

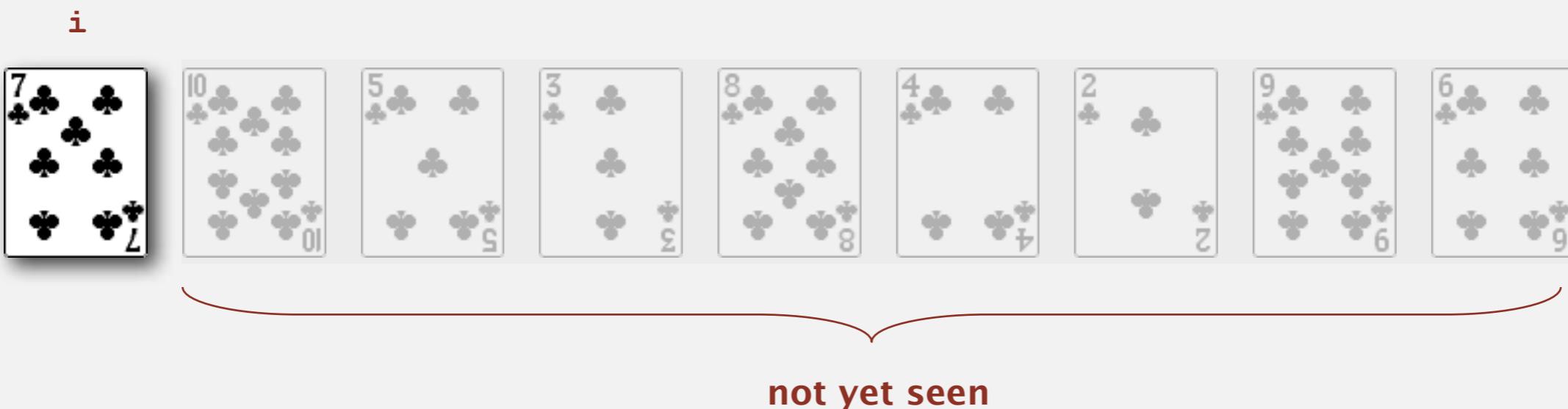
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



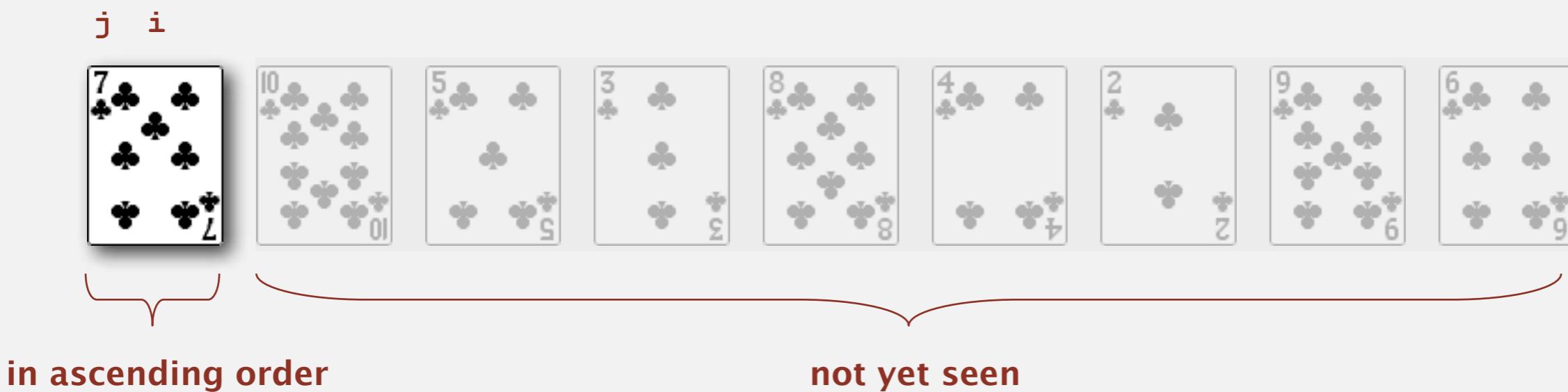
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



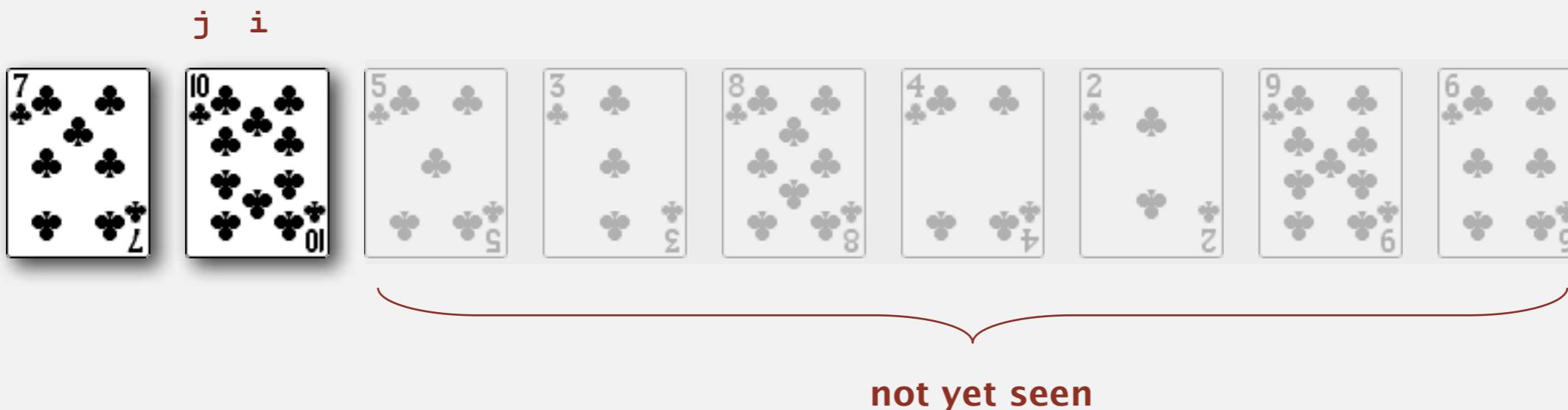
Selection sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



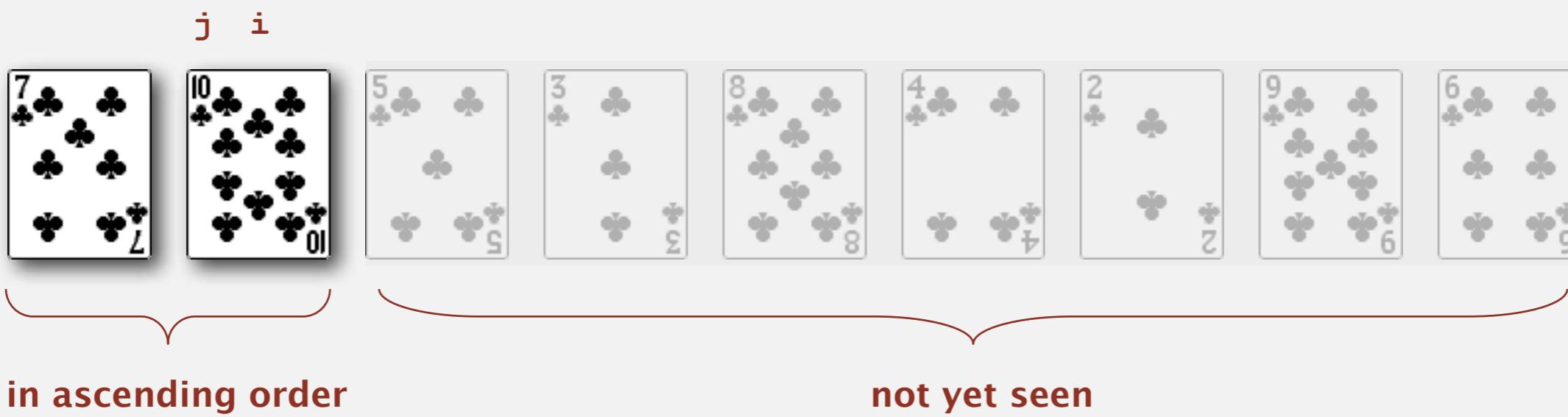
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



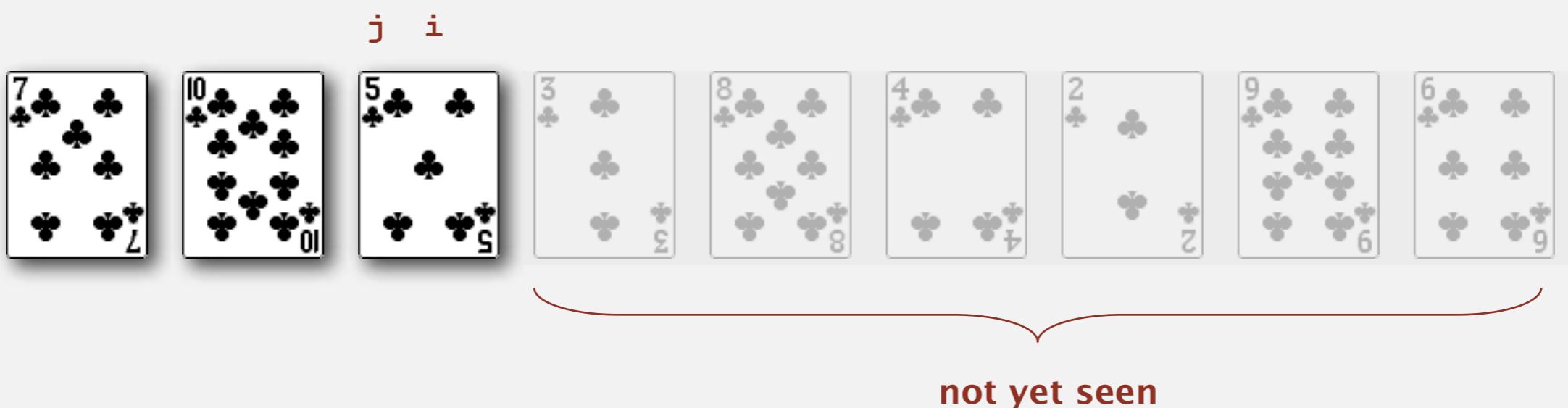
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



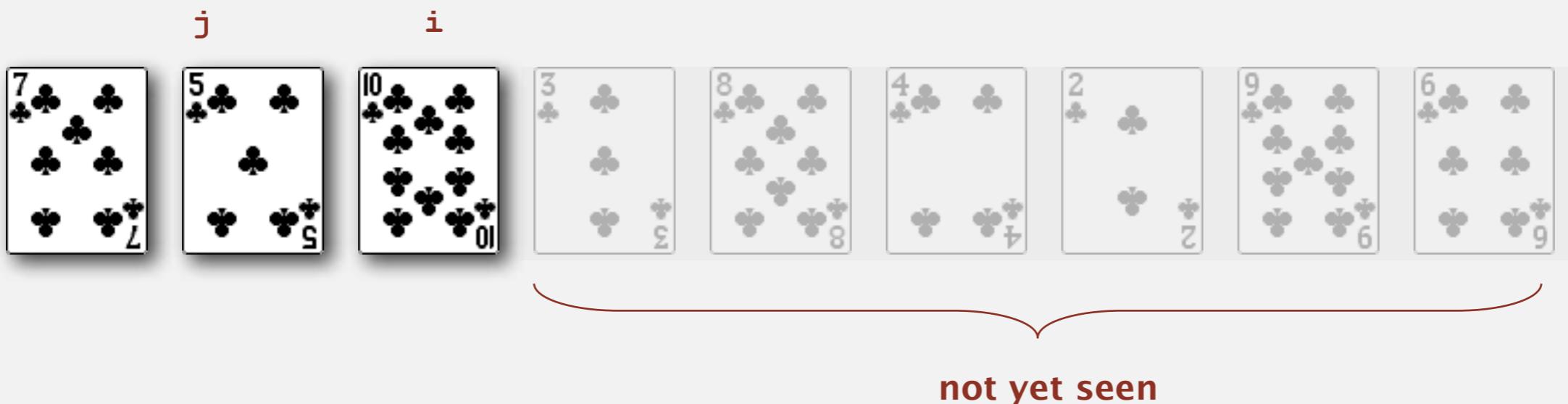
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



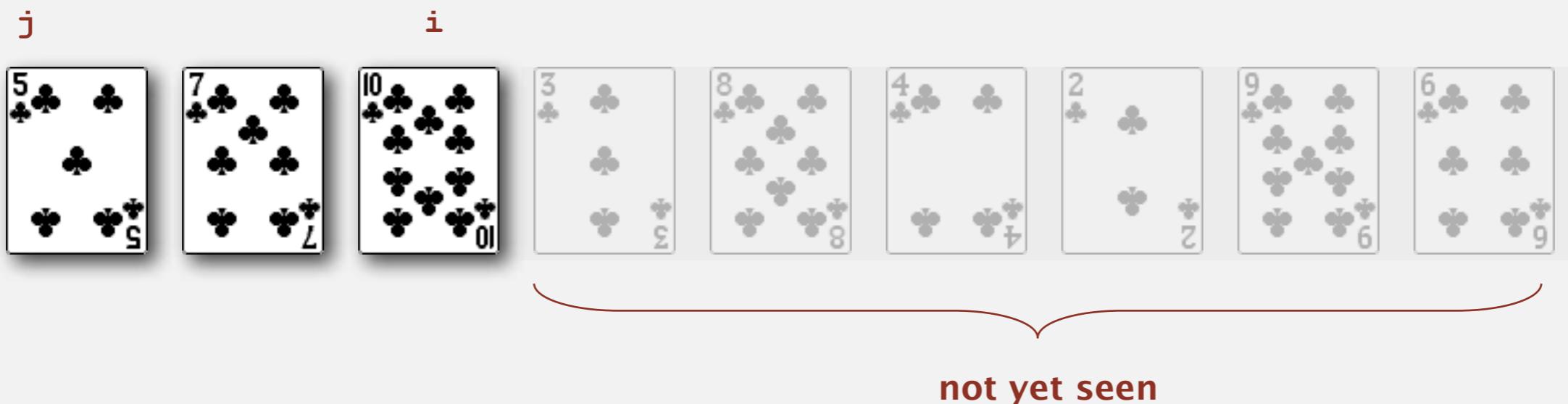
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



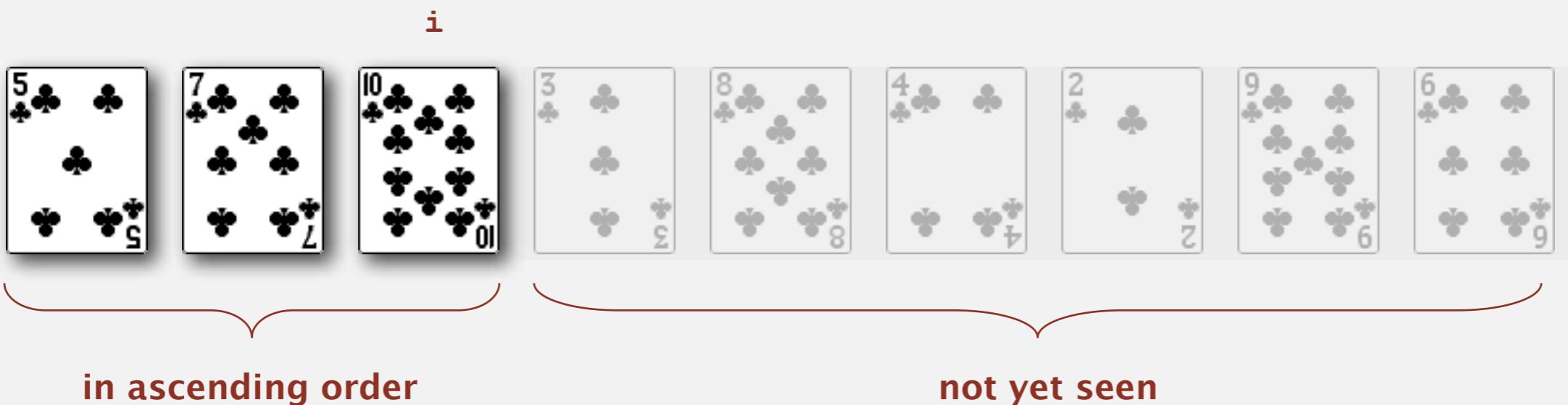
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



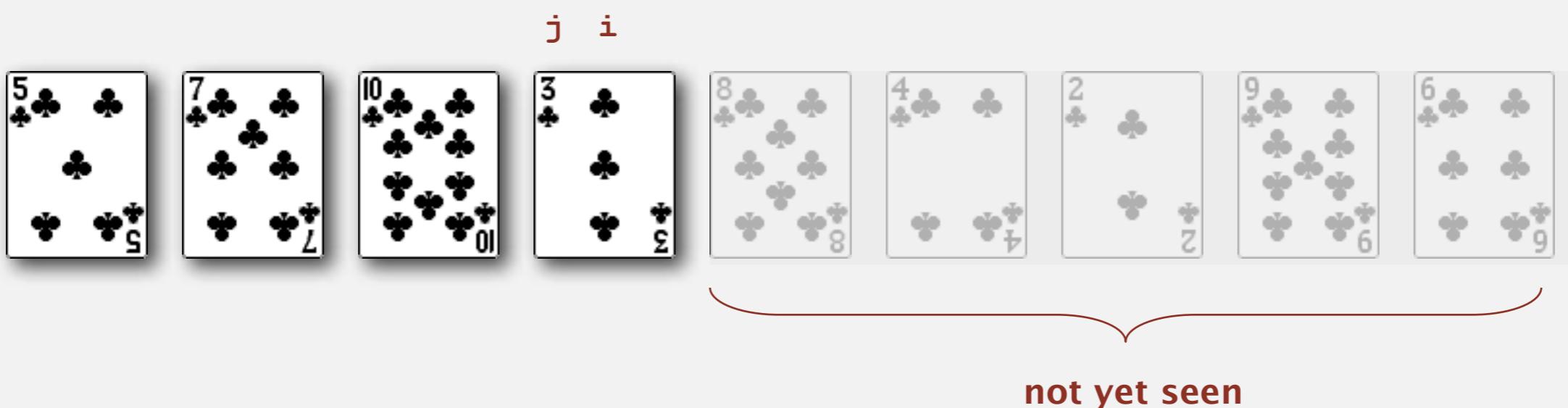
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



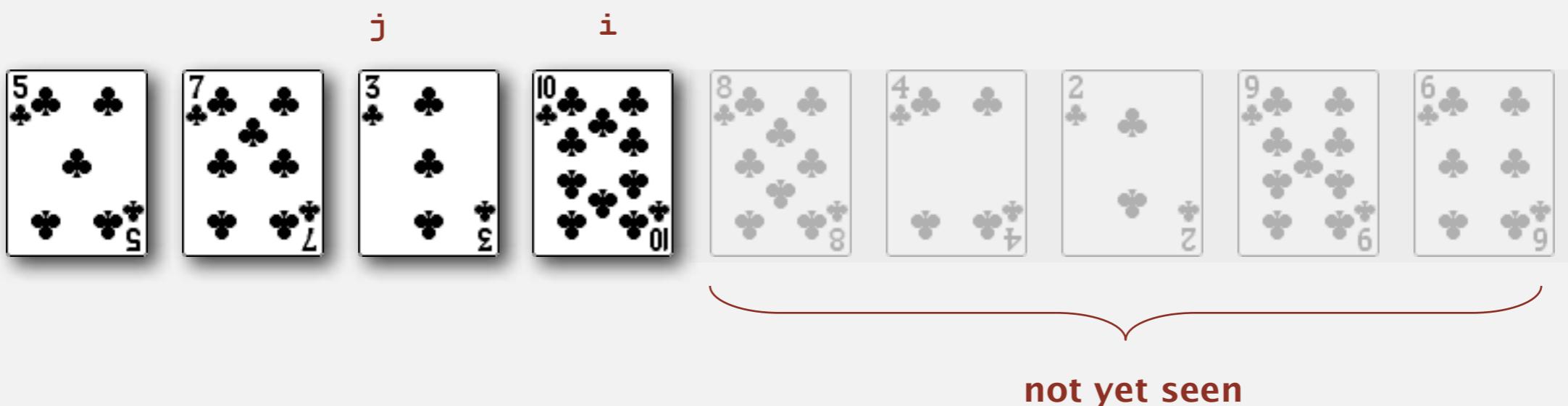
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



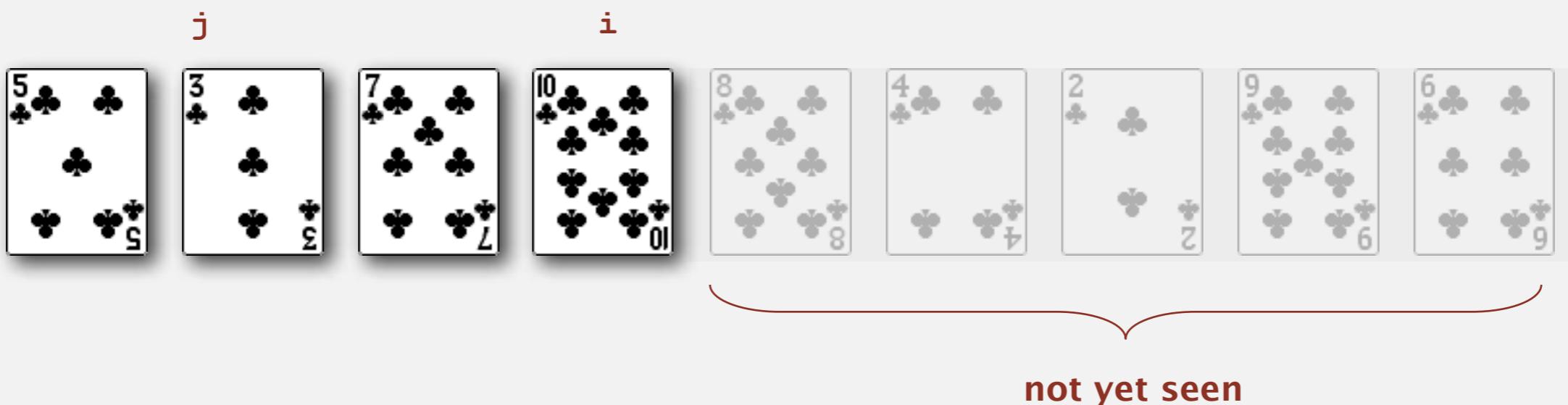
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



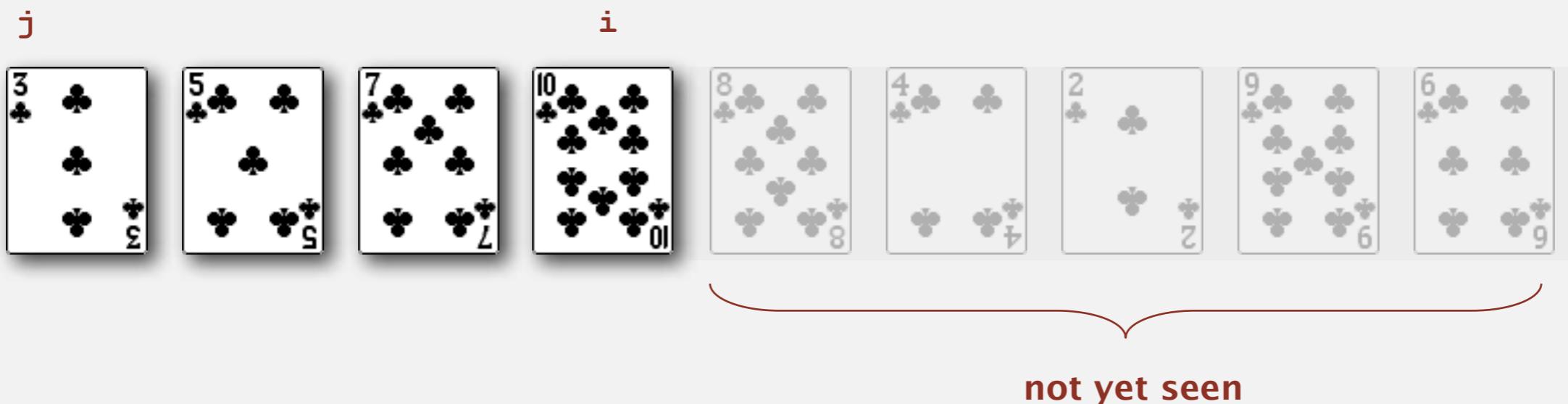
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



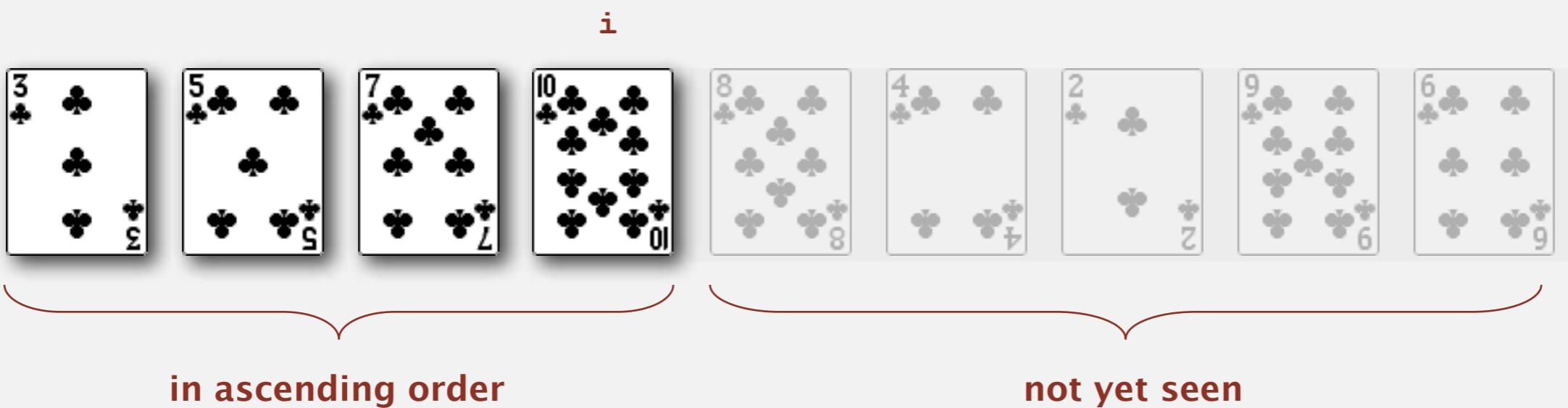
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



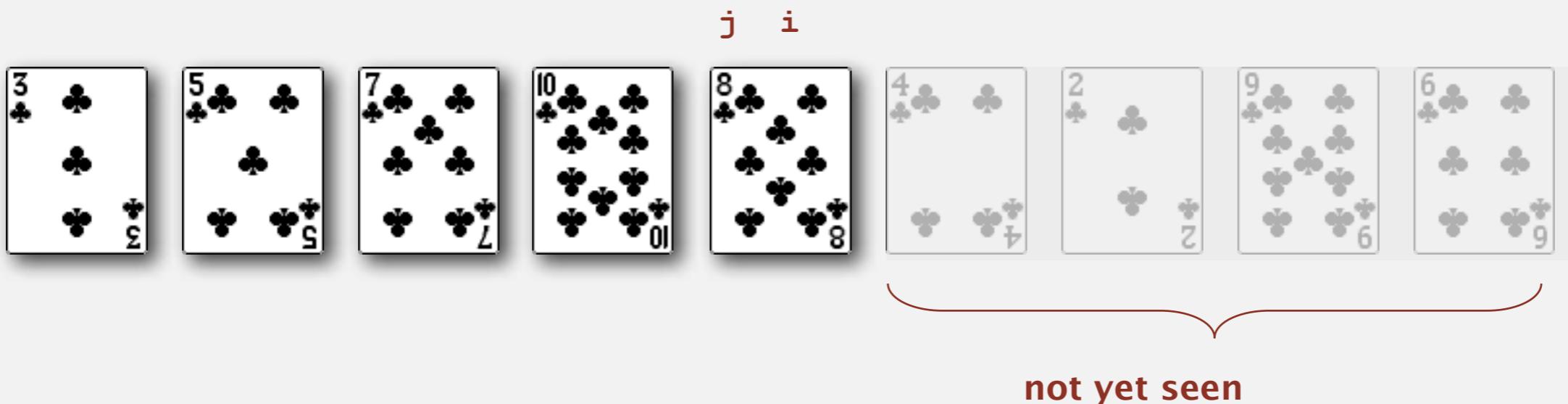
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



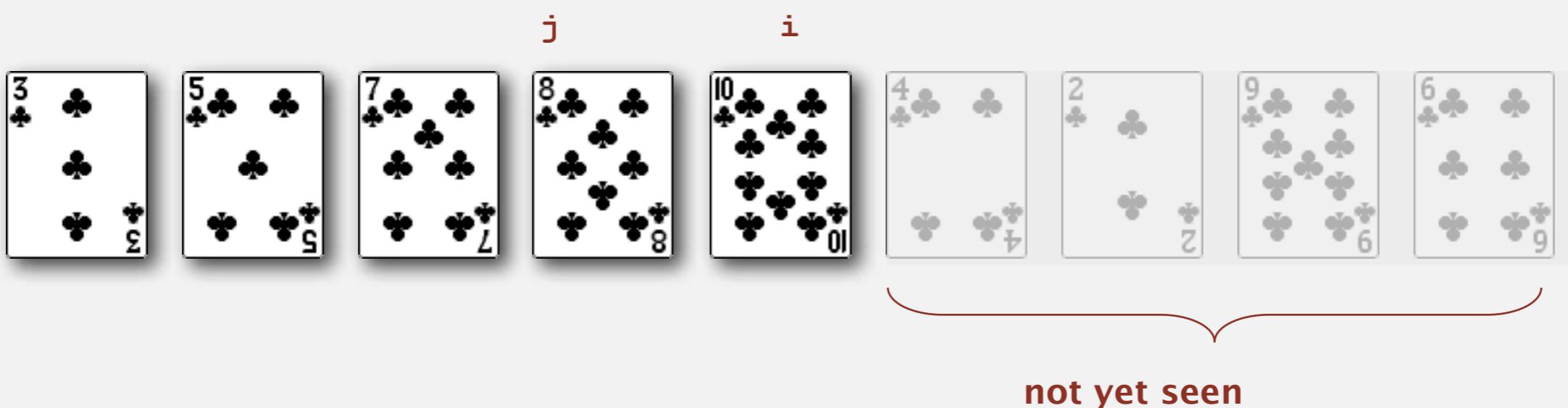
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



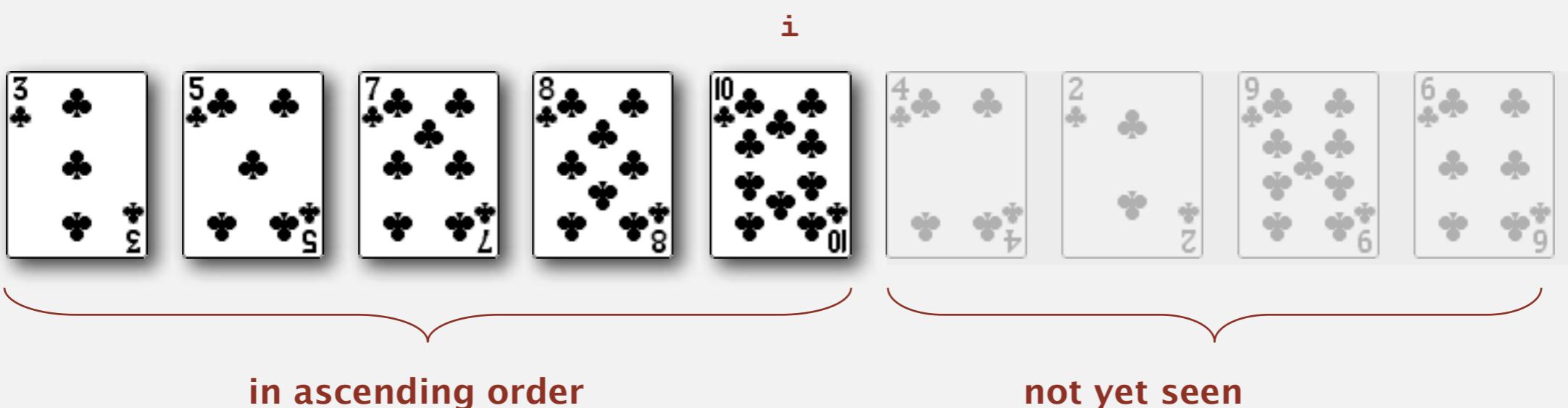
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



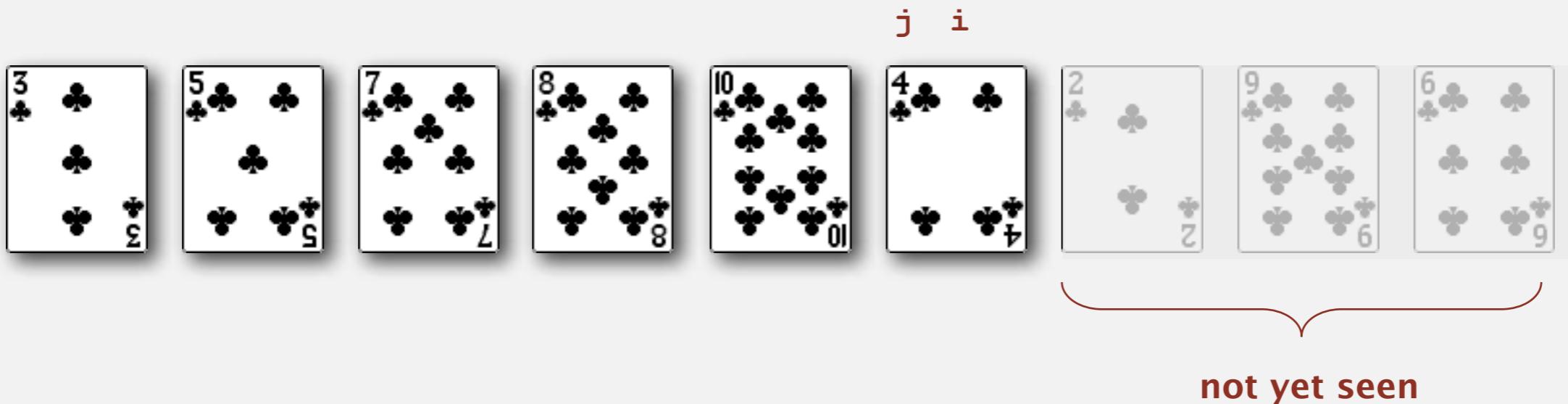
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



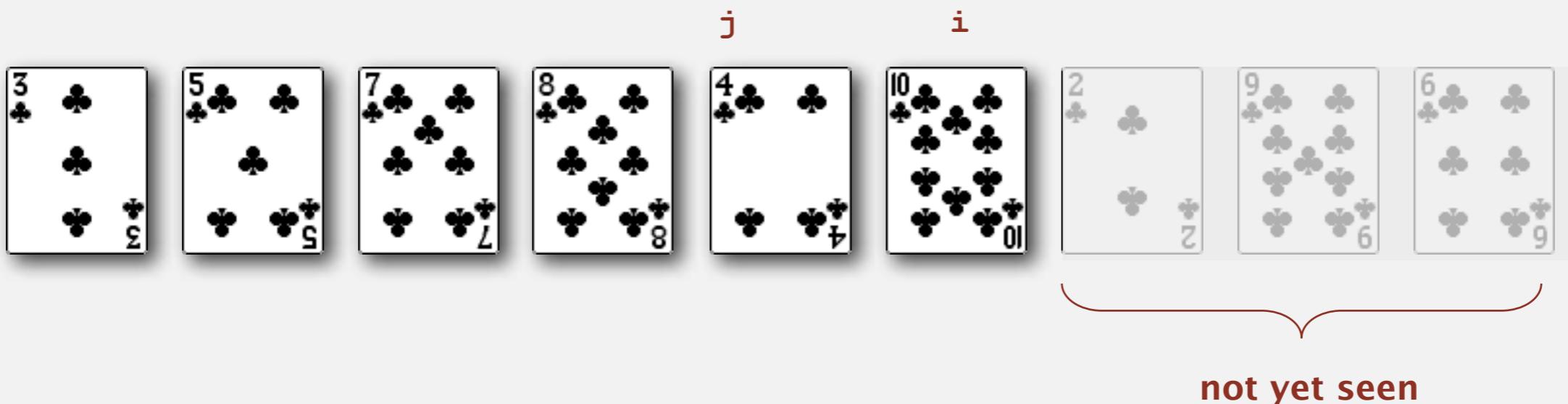
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



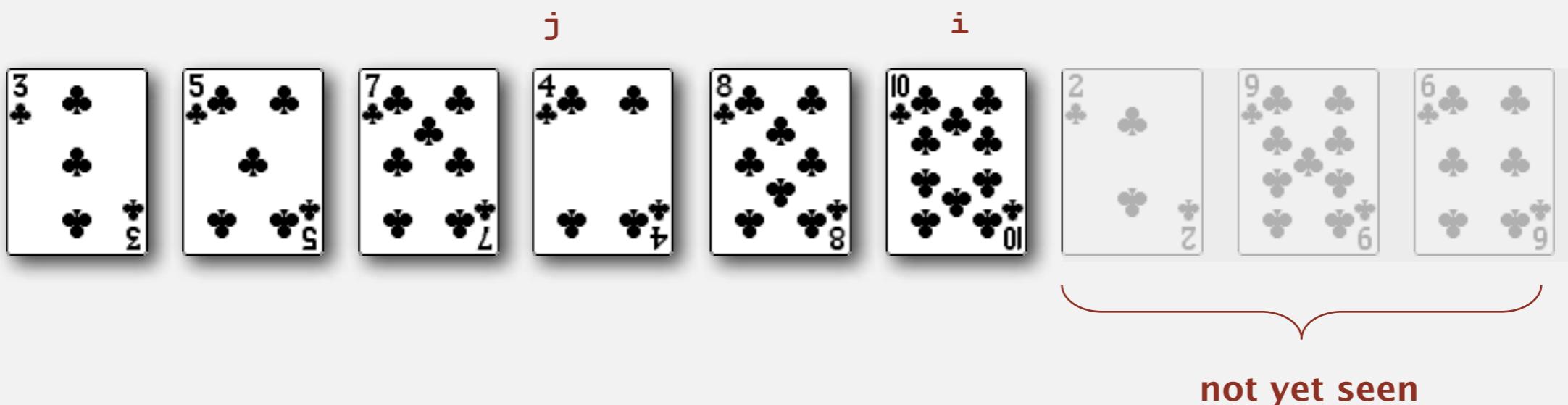
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



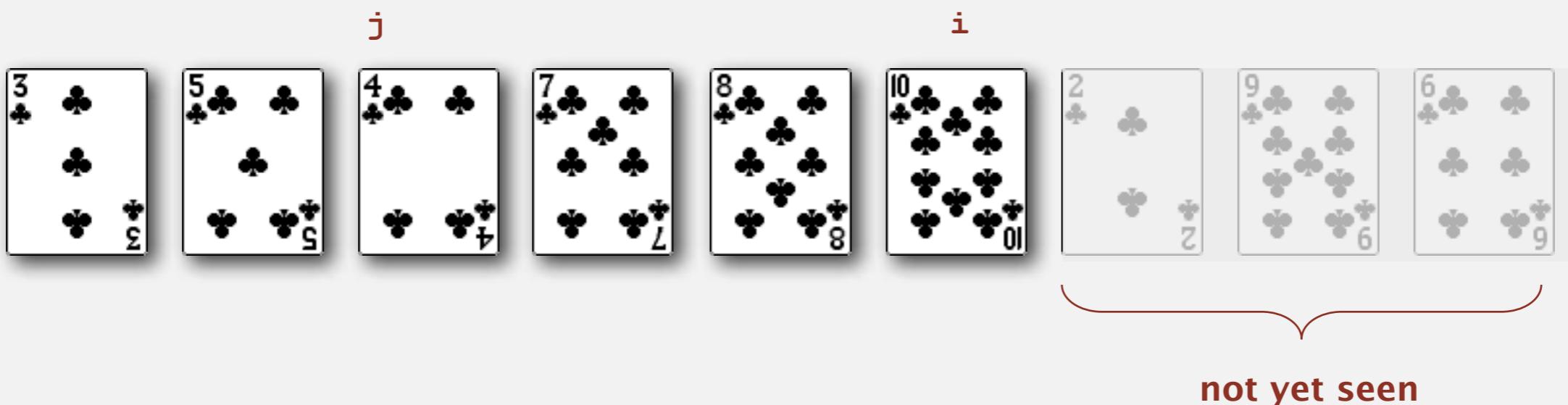
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



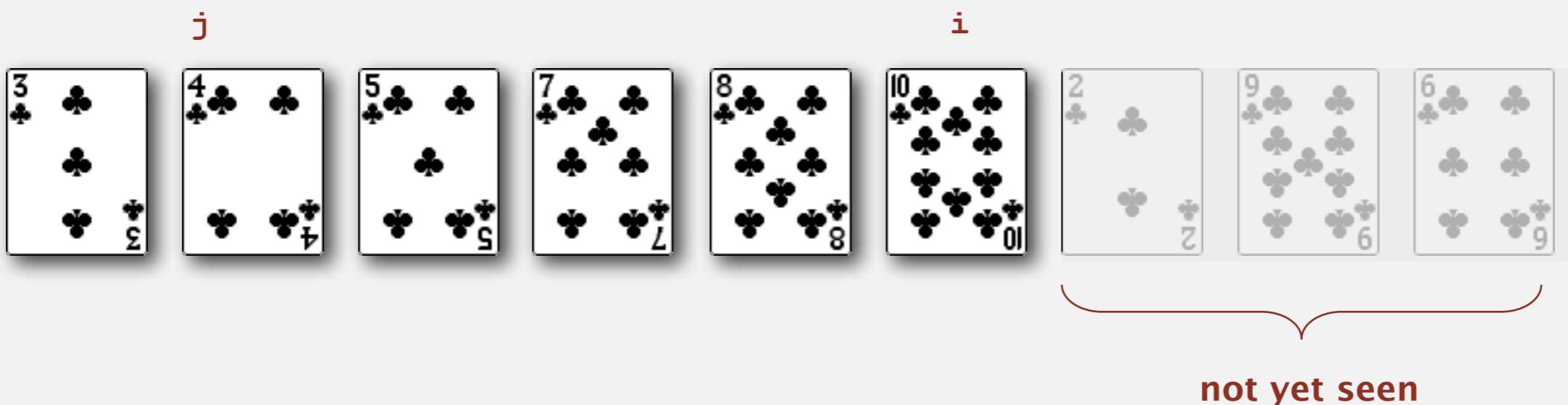
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



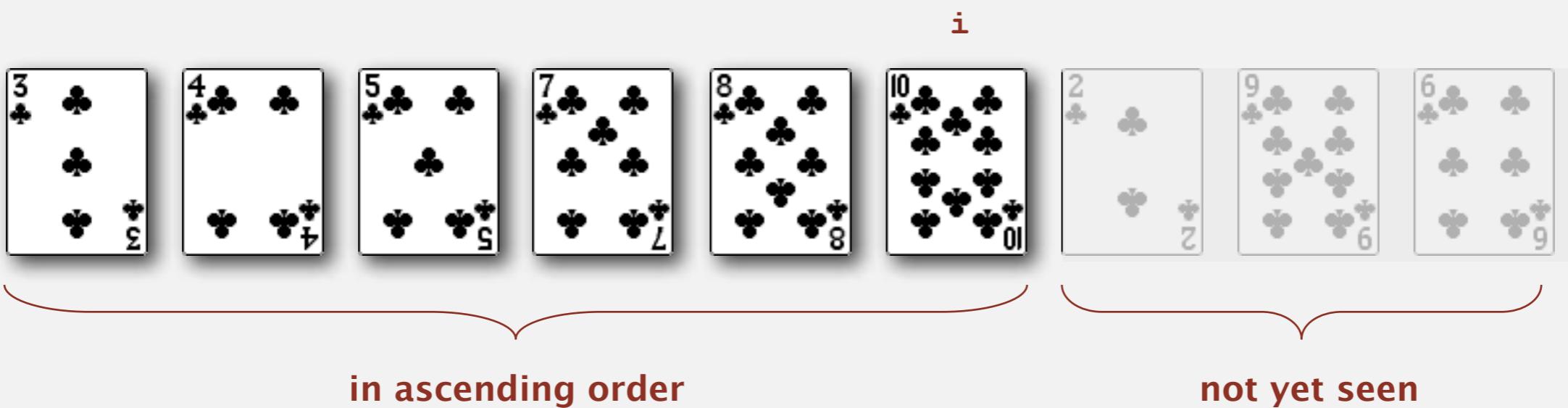
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



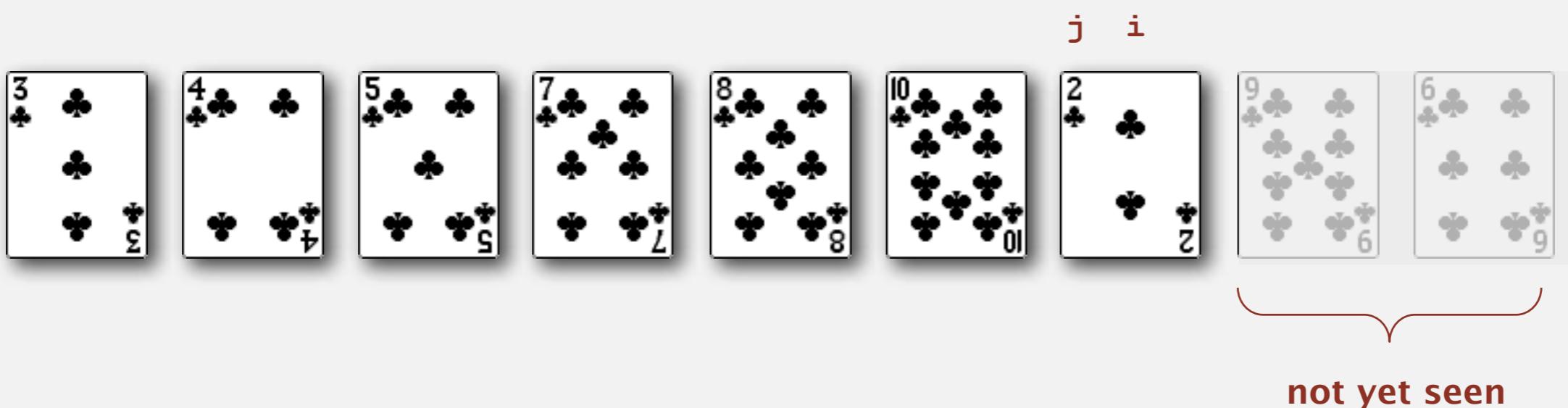
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



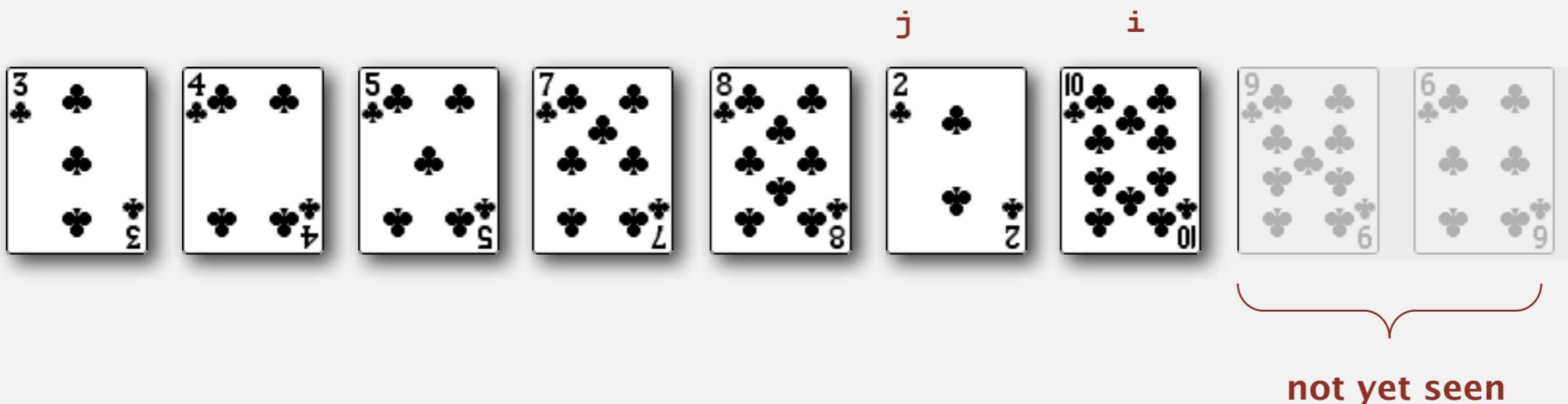
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



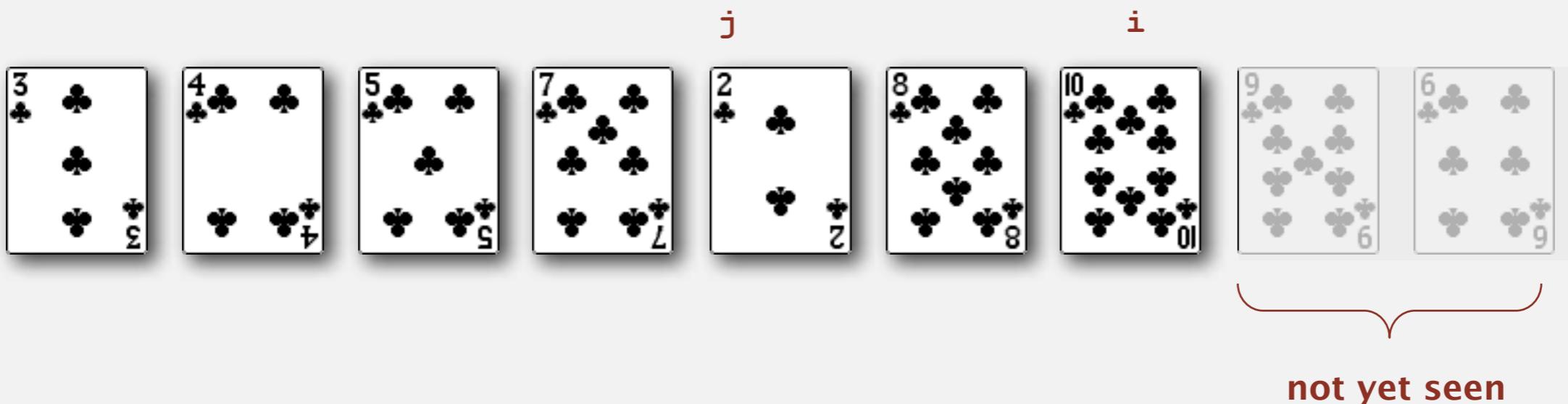
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



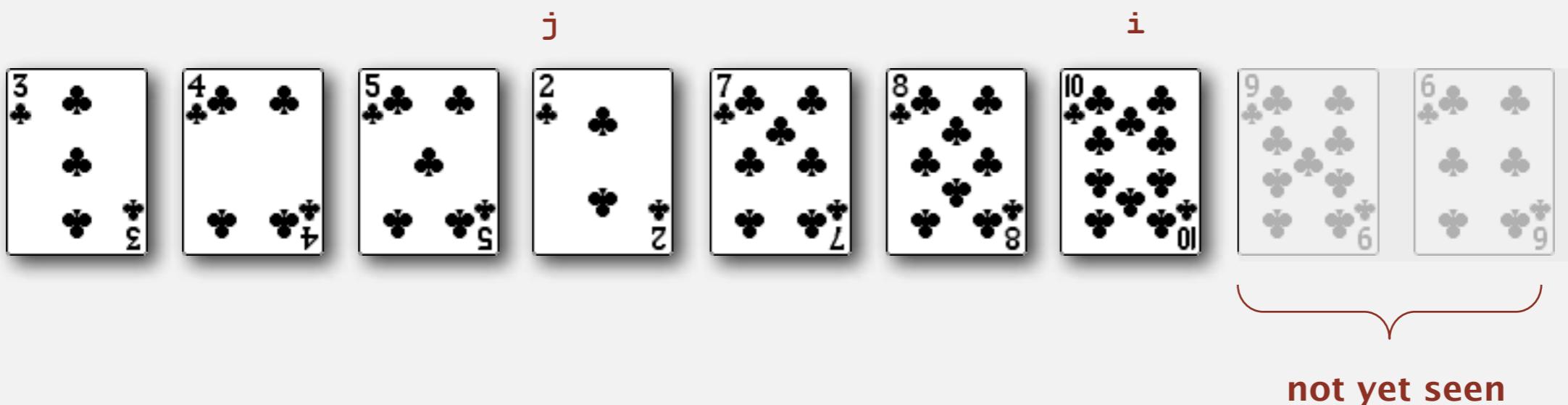
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



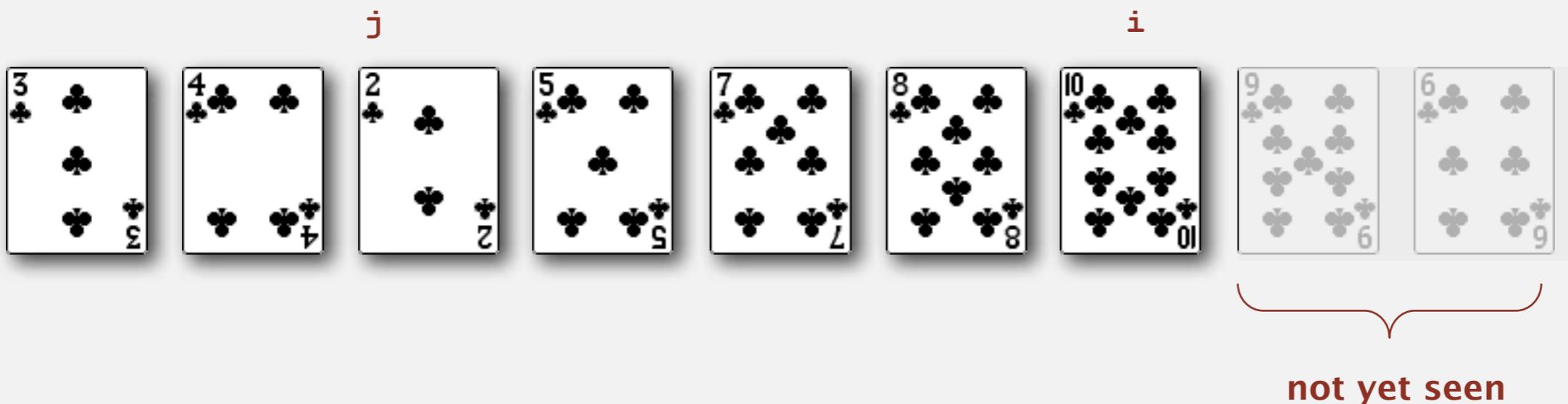
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



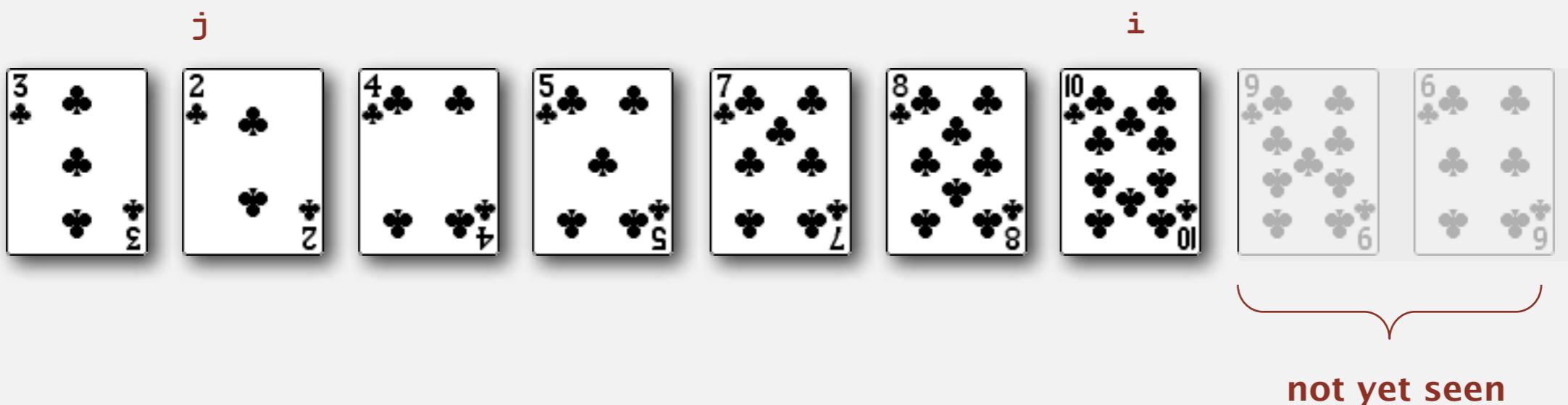
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



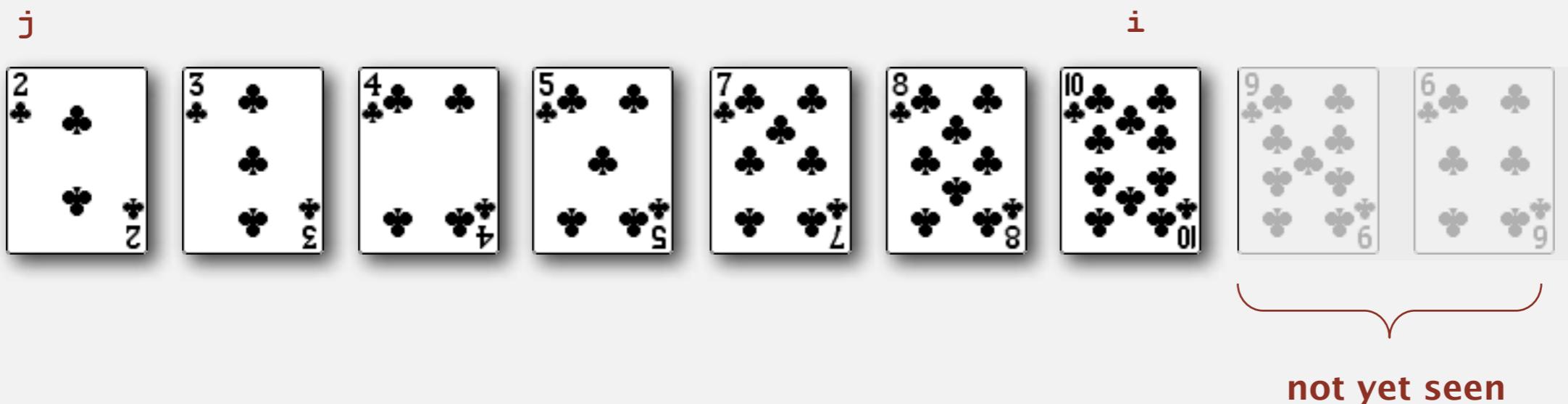
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



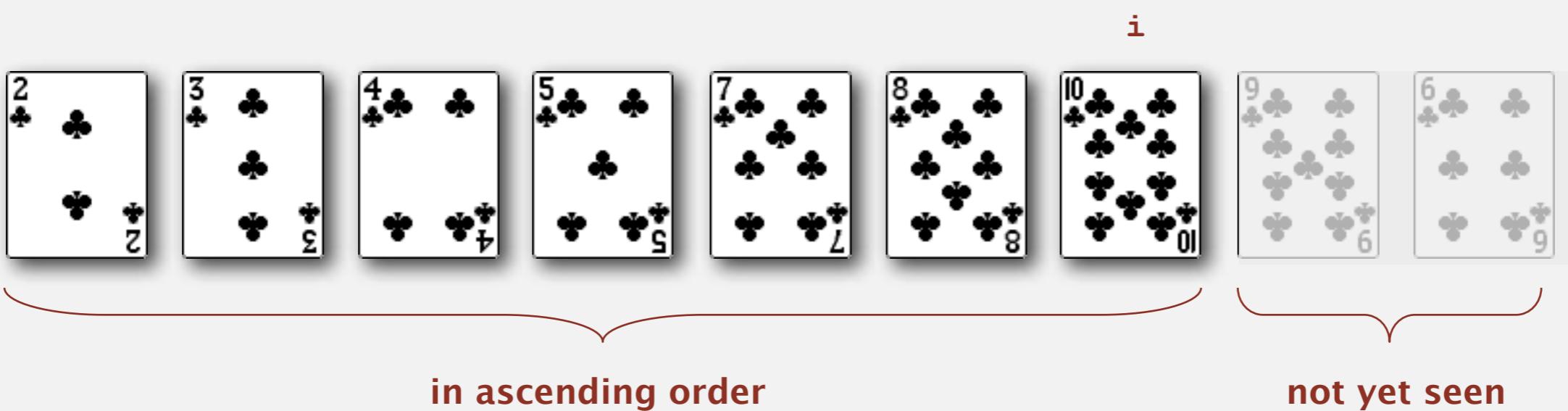
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



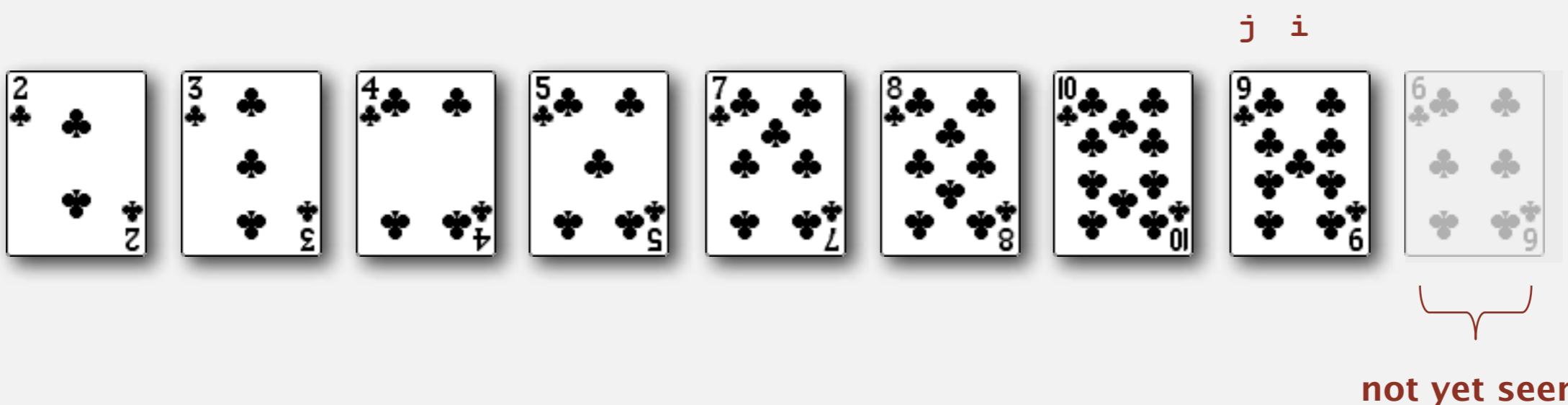
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



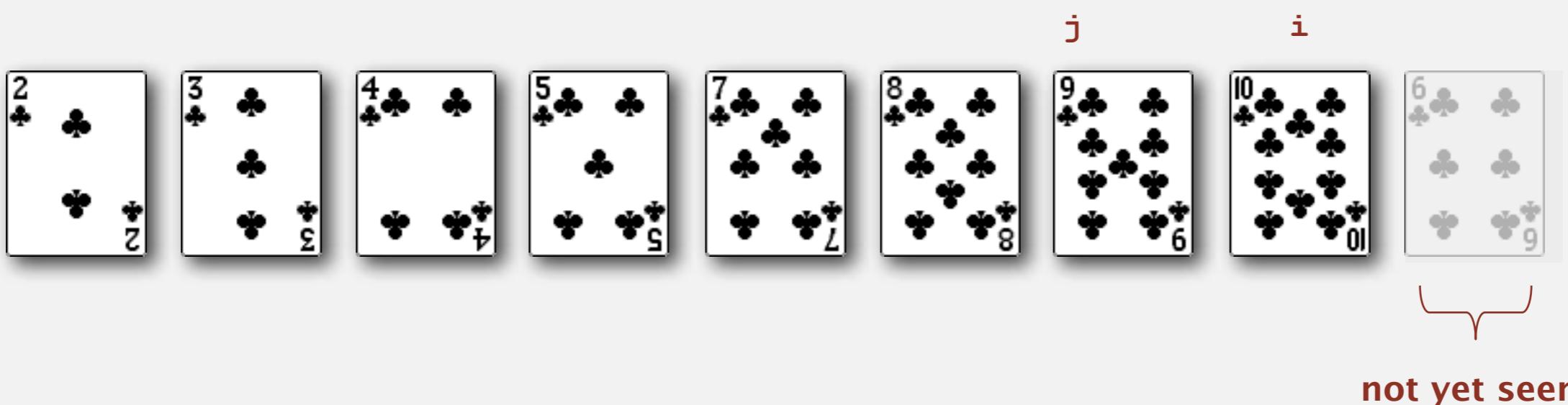
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



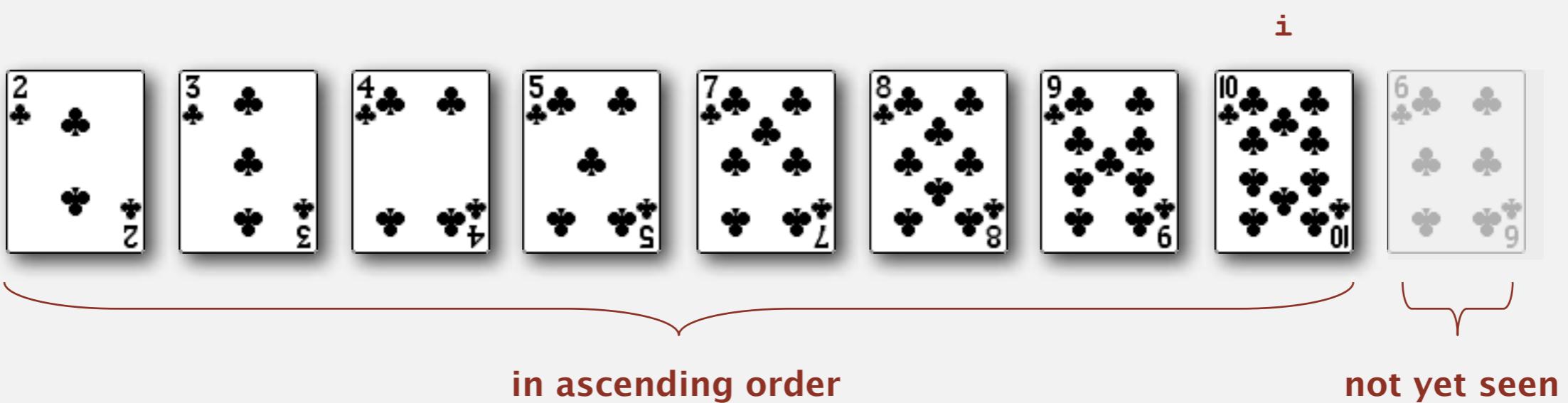
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



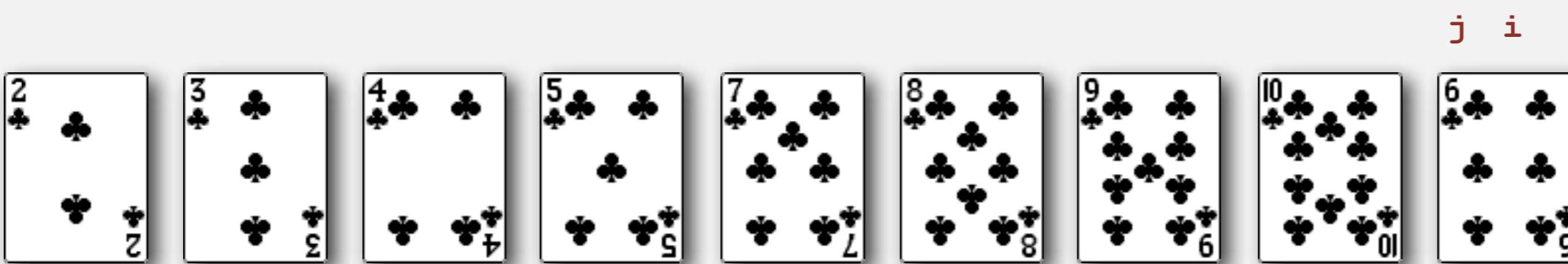
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



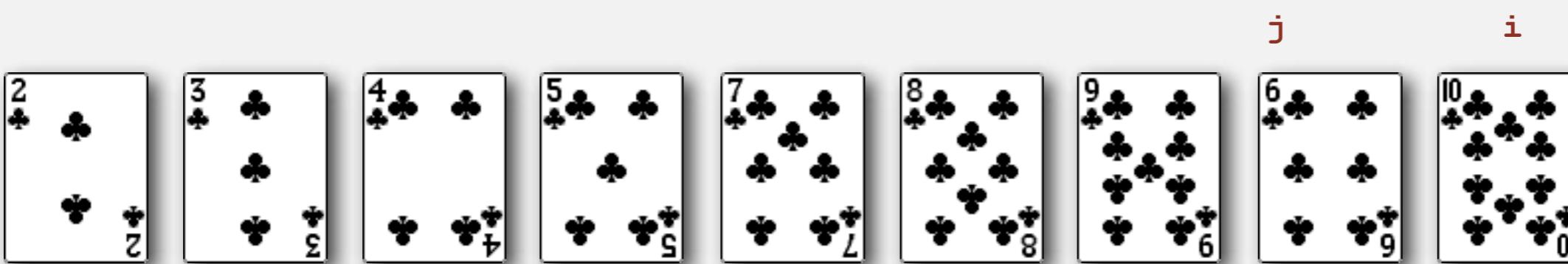
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



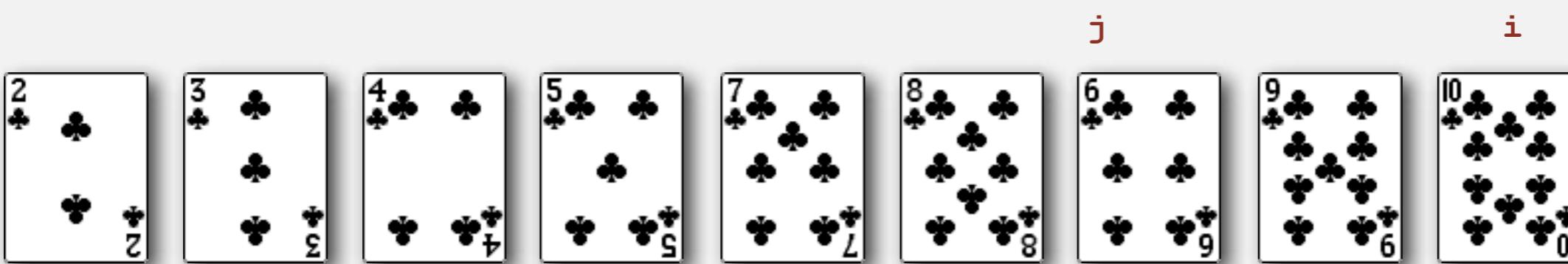
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



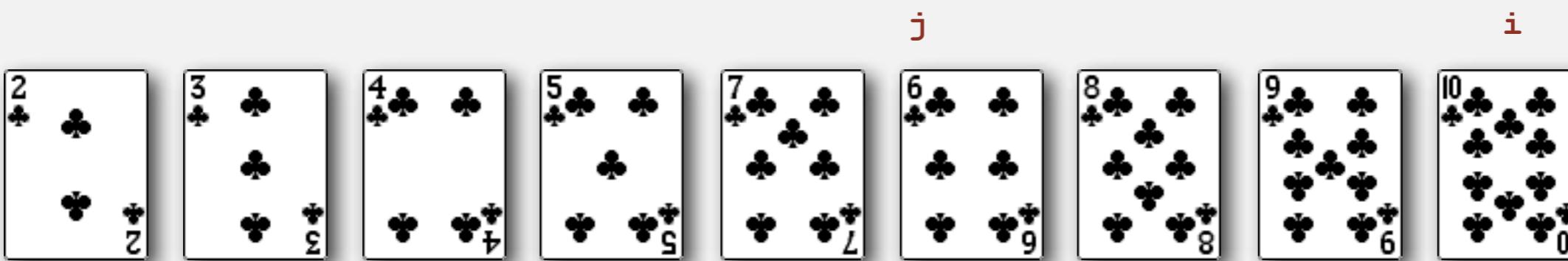
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



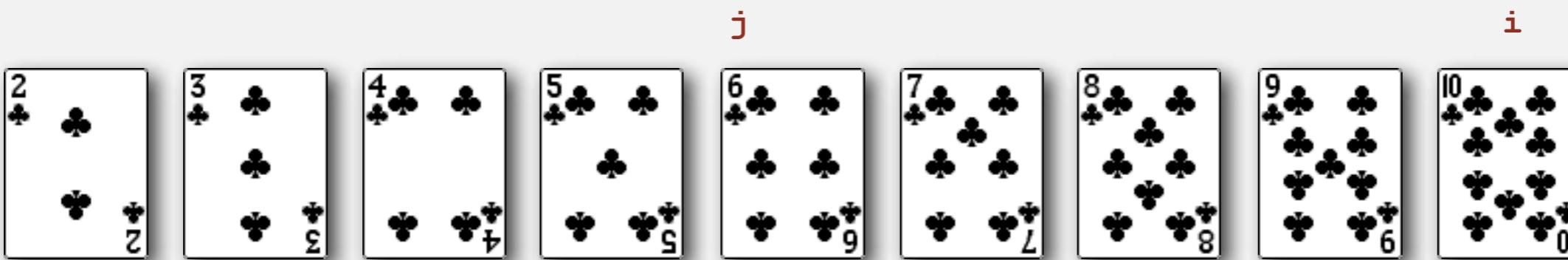
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



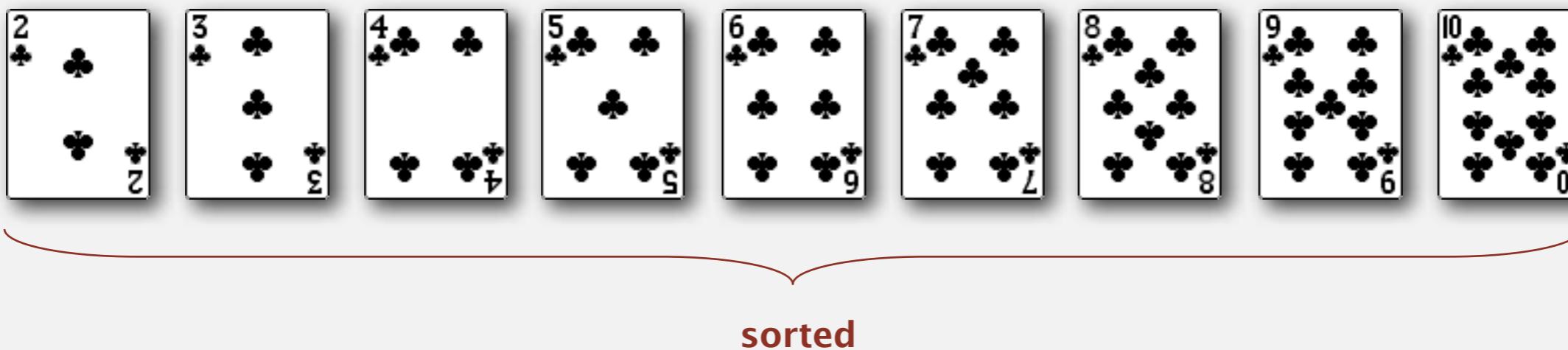
Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



Insertion sort

- In iteration i , swap $a[i]$ with each larger entry to its left.



Insertion sort: Java implementation

```
public class Insertion
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
            for (int j = i; j > 0; j--)
                if (less(a[j], a[j-1]))
                    exch(a, j, j-1);
                else break;
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
```

Insertion sort: mathematical analysis

Proposition. To sort a randomly-ordered array with distinct keys, insertion sort uses $\sim \frac{1}{4} N^2$ compares and $\sim \frac{1}{4} N^2$ exchanges on average.

Pf. Expect each entry to move halfway back.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	0	S	R	T	E	X	A	M	P	L	E
2	1	0	R	S	T	E	X	A	M	P	L	E
3	3	0	R	S	T	E	X	A	M	P	L	E
4	0	E	0	R	S	T	X	A	M	P	L	E
5	5	E	0	R	S	T	X	A	M	P	L	E
6	0	A	E	0	R	S	T	X	M	P	L	E
7	2	A	E	M	0	R	S	T	X	P	L	E
8	4	A	E	M	0	P	R	S	T	X	L	E
9	2	A	E	L	M	0	P	R	S	T	X	E
10	2	A	E	E	L	M	0	P	R	S	T	X
		A	E	E	L	M	0	P	R	S	T	X

Trace of insertion sort (array contents just after each insertion)

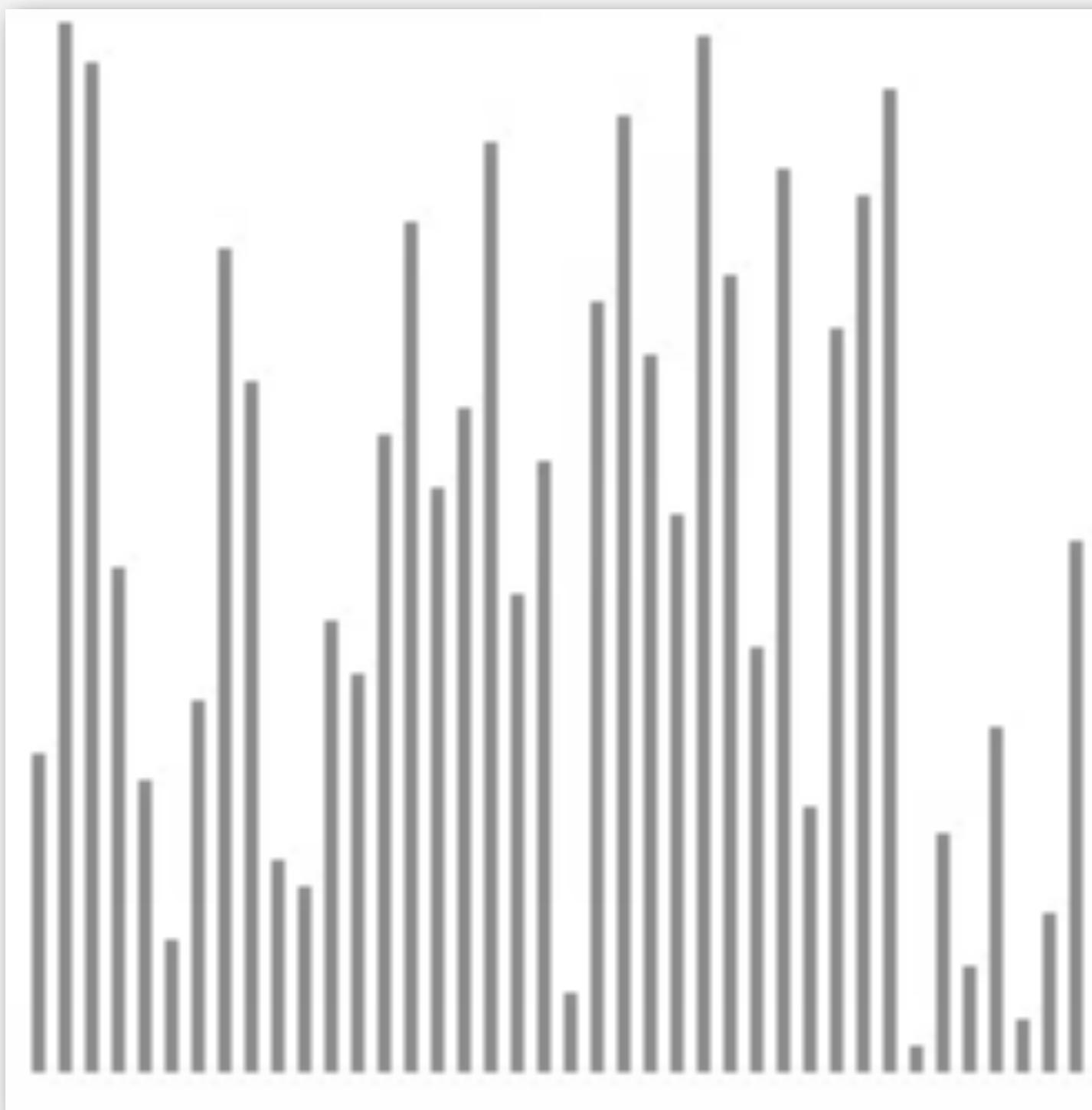
entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

Insertion sort: animation

40 random items



<http://www.sorting-algorithms.com/insertion-sort>

Insertion sort: best and worst case

Best case. If the array is in ascending order, insertion sort makes $N - 1$ compares and 0 exchanges.

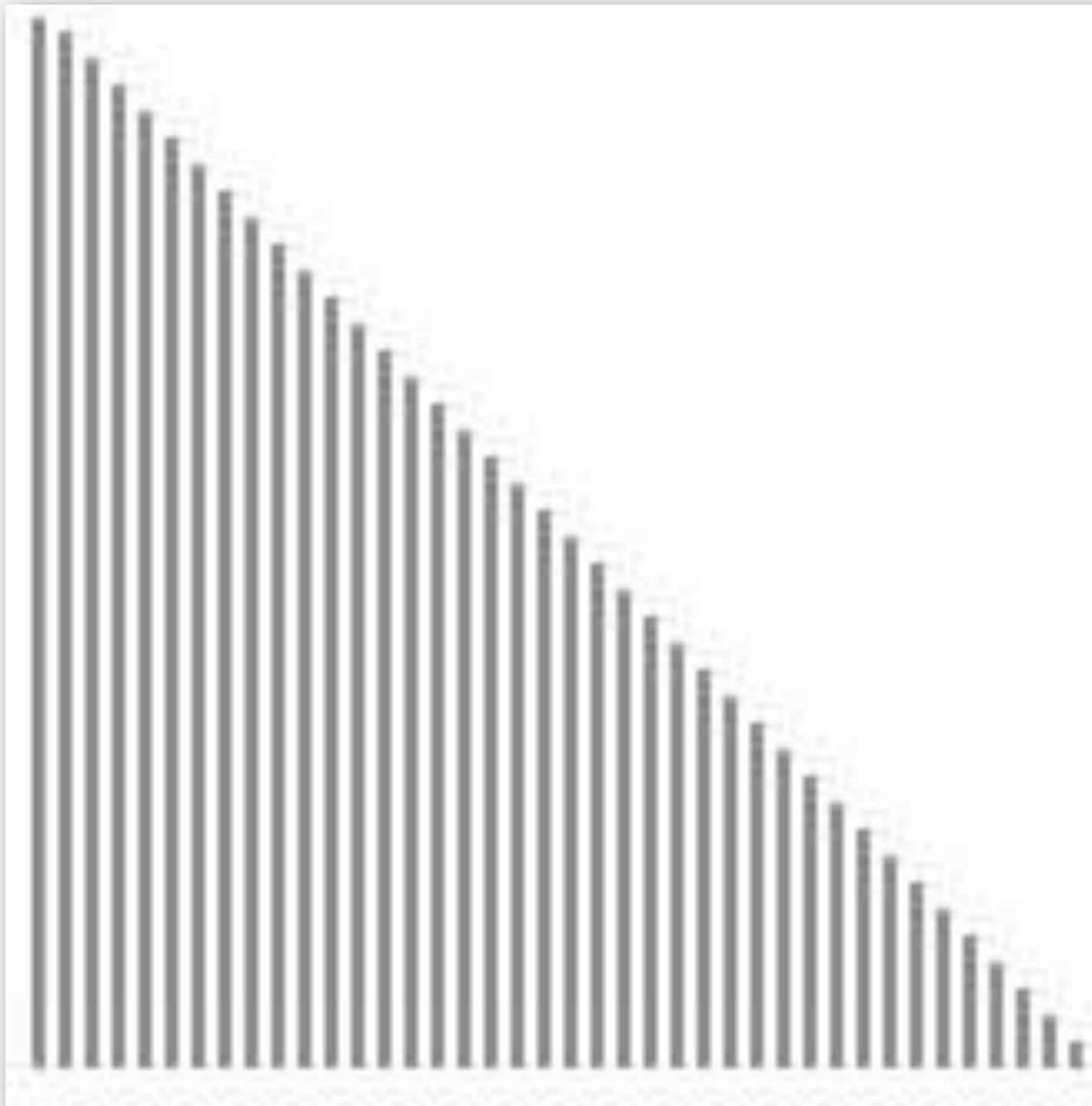
```
A E E L M O P R S T X
```

Worst case. If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2} N^2$ compares and $\sim \frac{1}{2} N^2$ exchanges.

```
X T S R P O M L E E A
```

Insertion sort: animation

40 reverse-sorted items



<http://www.sorting-algorithms.com/insertion-sort>

Insertion sort: partially-sorted arrays

Def. An **inversion** is a pair of keys that are out of order.

A E E L M O T R X P S

.T-R T-P T-S R-P X-P X-S

(6 inversions)

Def. An array is **partially sorted** if the number of inversions is $\leq c N$.

- Ex 1. A subarray of size 10 appended to a sorted subarray of size N .
- Ex 2. An array of size N with only 10 entries out of place.

Proposition. For partially-sorted arrays, insertion sort runs in linear time.

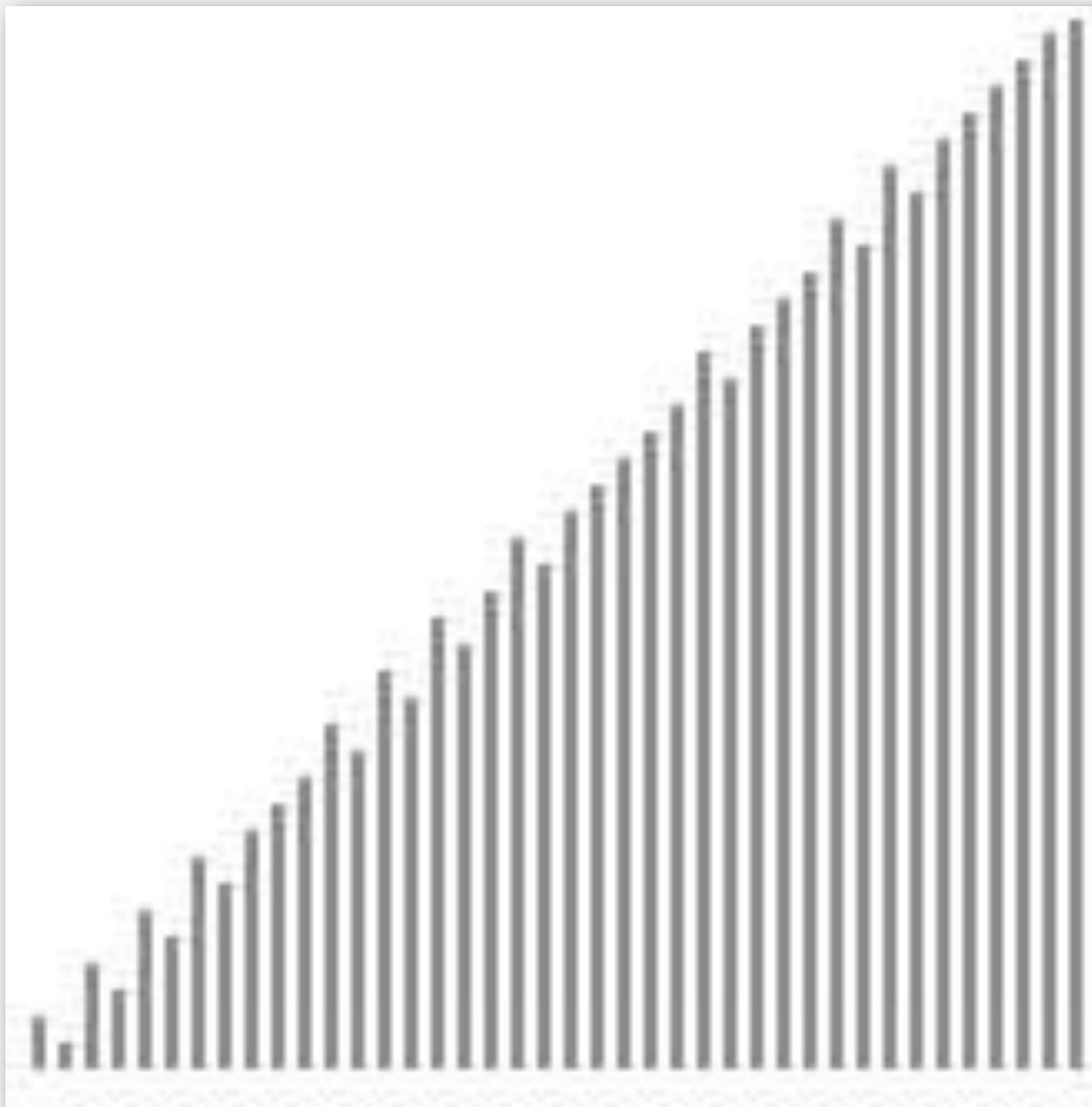
Pf. Number of exchanges equals the number of inversions.



number of compares = exchanges + $(N - 1)$

Insertion sort: animation

40 partially-sorted items



<http://www.sorting-algorithms.com/insertion-sort>

ELEMENTARY SORTING ALGORITHMS

- ▶ **Sorting review**
- ▶ Rules of the game
- ▶ Selection sort
- ▶ Insertion sort
- ▶ Shellsort

Shellsort overview

Idea. Move entries more than one position at a time by *h*-sorting the array.

an h-sorted array is h interleaved sorted subsequences

$h=4$



Shellsort. [Shell 1959] *h*-sort the array for decreasing seq. of values of *h*.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

h-sorting

How to h -sort an array? Insertion sort, with stride length h .

3-sorting an array

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

Why insertion sort?

- Big increments \Rightarrow small subarray.
- Small increments \Rightarrow nearly in order. [stay tuned]

Shellsort example: increments 7, 3, 1

input

S	O	R	T	E	X	A	M	P	L	E
---	---	---	---	---	---	---	---	---	---	---

7-sort

S	O	R	T	E	X	A	M	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T

3-sort

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

1-sort

A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	R	S	X	T
A	E	E	L	M	O	P	R	S	T	X

result

A	E	E	L	M	O	P	R	S	T	X
---	---	---	---	---	---	---	---	---	---	---

Shellsort: intuition

Proposition. A g -sorted array remains g -sorted after h -sorting it.

7-sort

M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T
M	O	L	E	E	X	A	S	P	R	T

3-sort

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

still 7-sorted

Shellsort: which increment sequence to use?

Powers of two. 1, 2, 4, 8, 16, 32, ...

No.

Powers of two minus one. 1, 3, 7, 15, 31, 63, ...

Maybe.

→ $3x + 1$. 1, 4, 13, 40, 121, 364, ...

OK. Easy to compute.

merging of $(9 \times 4^i) - (9 \times 2^i) + 1$ and $4^i - (3 \times 2^i) + 1$



Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

Good. Tough to beat in empirical studies.

=

Interested in learning more?

- See Section 6.8 of Algs, 3rd edition or Volume 3 of Knuth for details.
- Do a JP on the topic.

Shellsort: Java implementation

```
public class Shell
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;

        int h = 1;
        while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...

        while (h >= 1)
        { // h-sort the array.
            for (int i = h; i < N; i++)
            {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }

            h = h/3;
        }
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

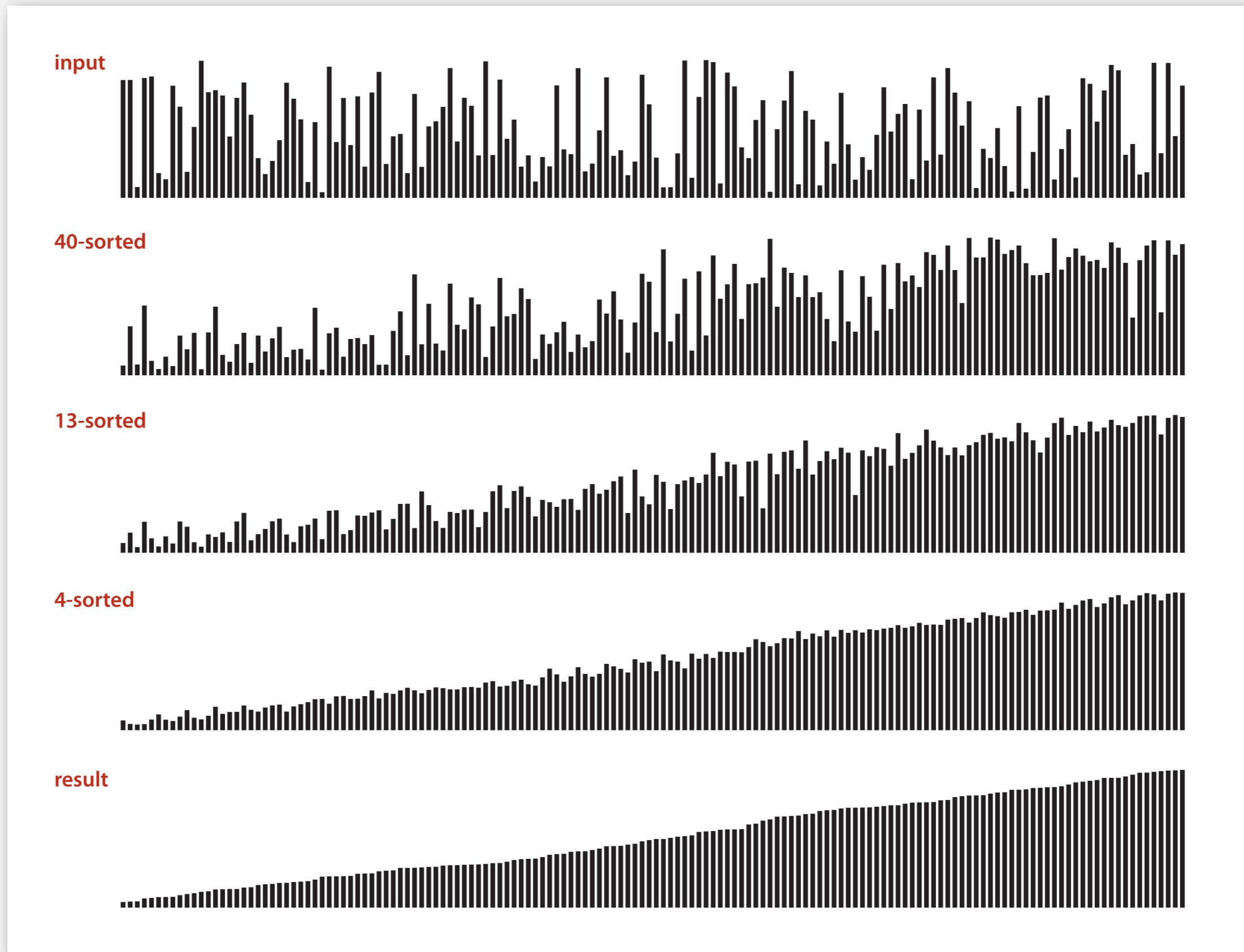
    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
```

3x+1 increment sequence

insertion sort

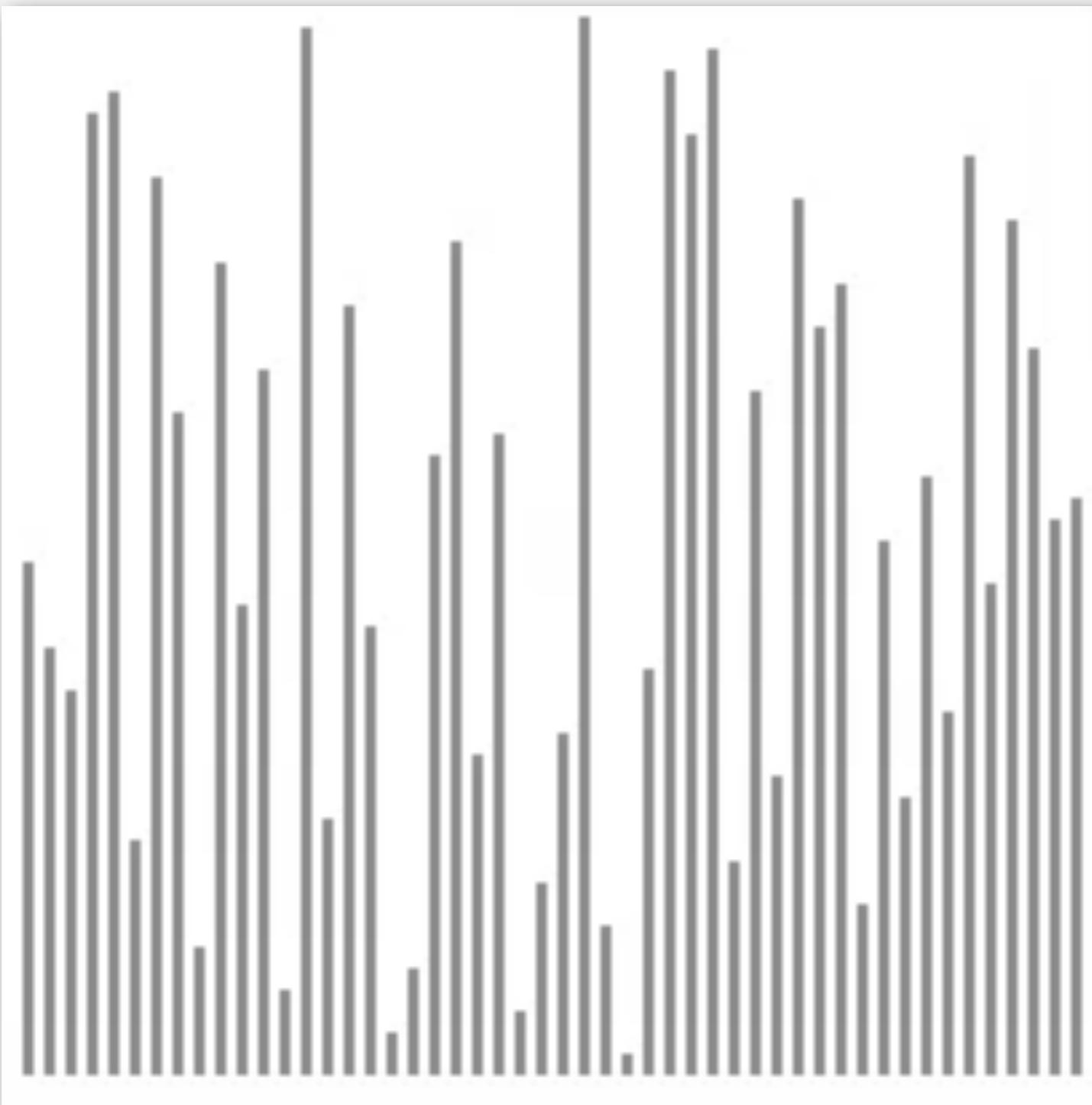
move to next increment

Shellsort: visual trace



Shellsort: animation

50 random items

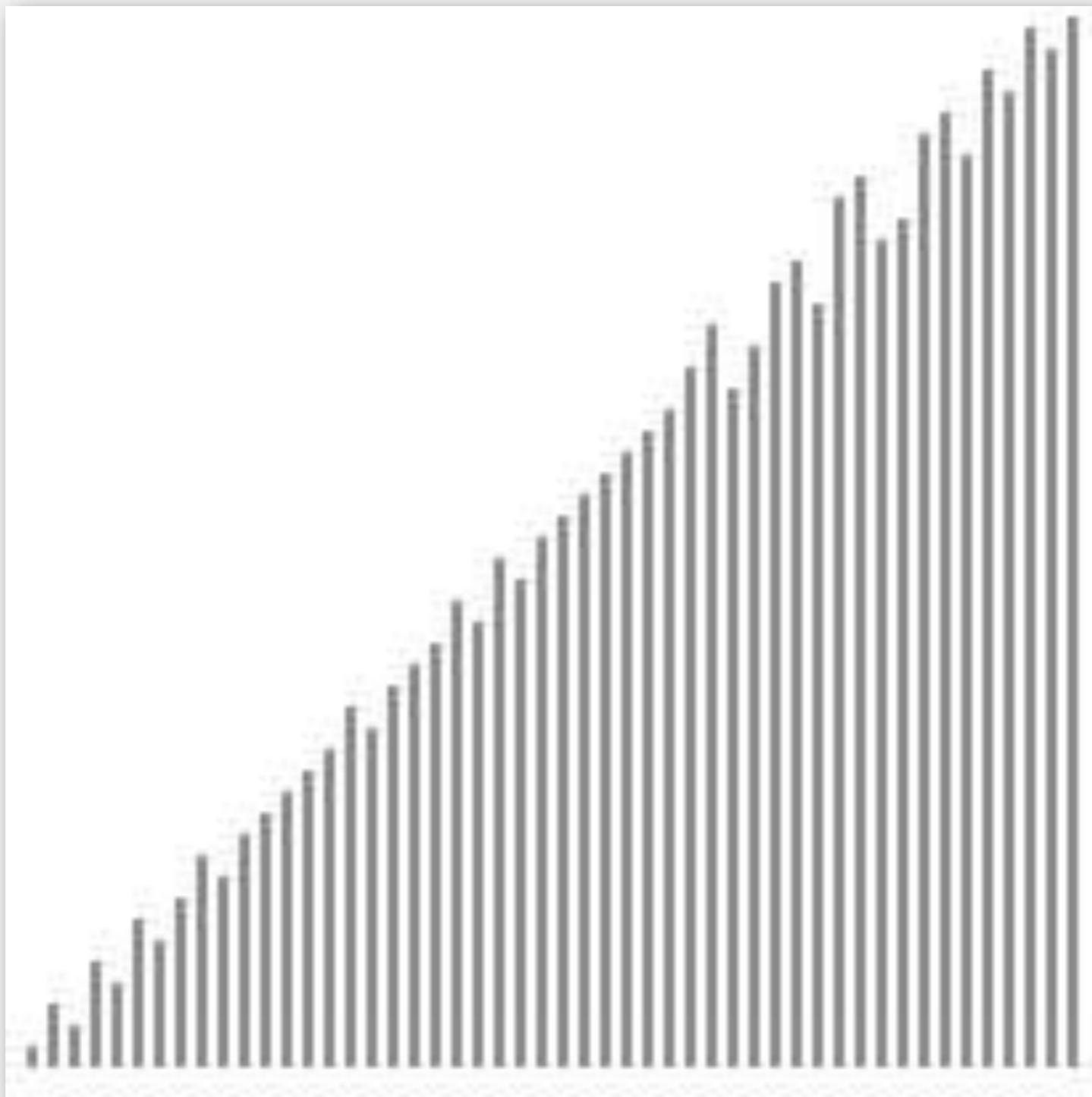


<http://www.sorting-algorithms.com/shell-sort>

- ▲ algorithm position
 - █ h-sorted
 - █ current subsequence
 - █ other elements

Shellsort: animation

50 partially-sorted items



<http://www.sorting-algorithms.com/shell-sort>

- ▲ algorithm position
 - █ h-sorted
 - █ current subsequence
 - █ other elements

Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains.

Useful in practice.

- Fast unless array size is huge.
- Tiny, fixed footprint for code (used in embedded systems).
- Hardware sort prototype.

Simple algorithm, nontrivial performance, interesting questions.

- Asymptotic growth rate?
- Best sequence of increments? ← open problem: find a better increment sequence
- Average-case performance?

Lesson. Some good algorithms are still waiting discovery.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

MERGESORT

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

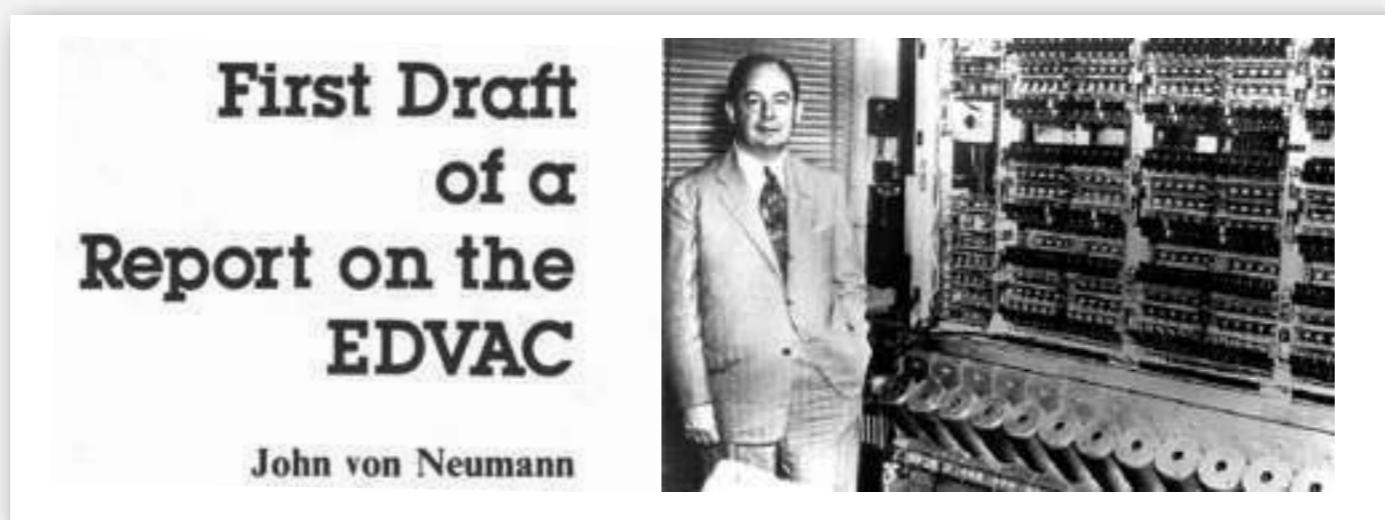
Mergesort

Basic plan.

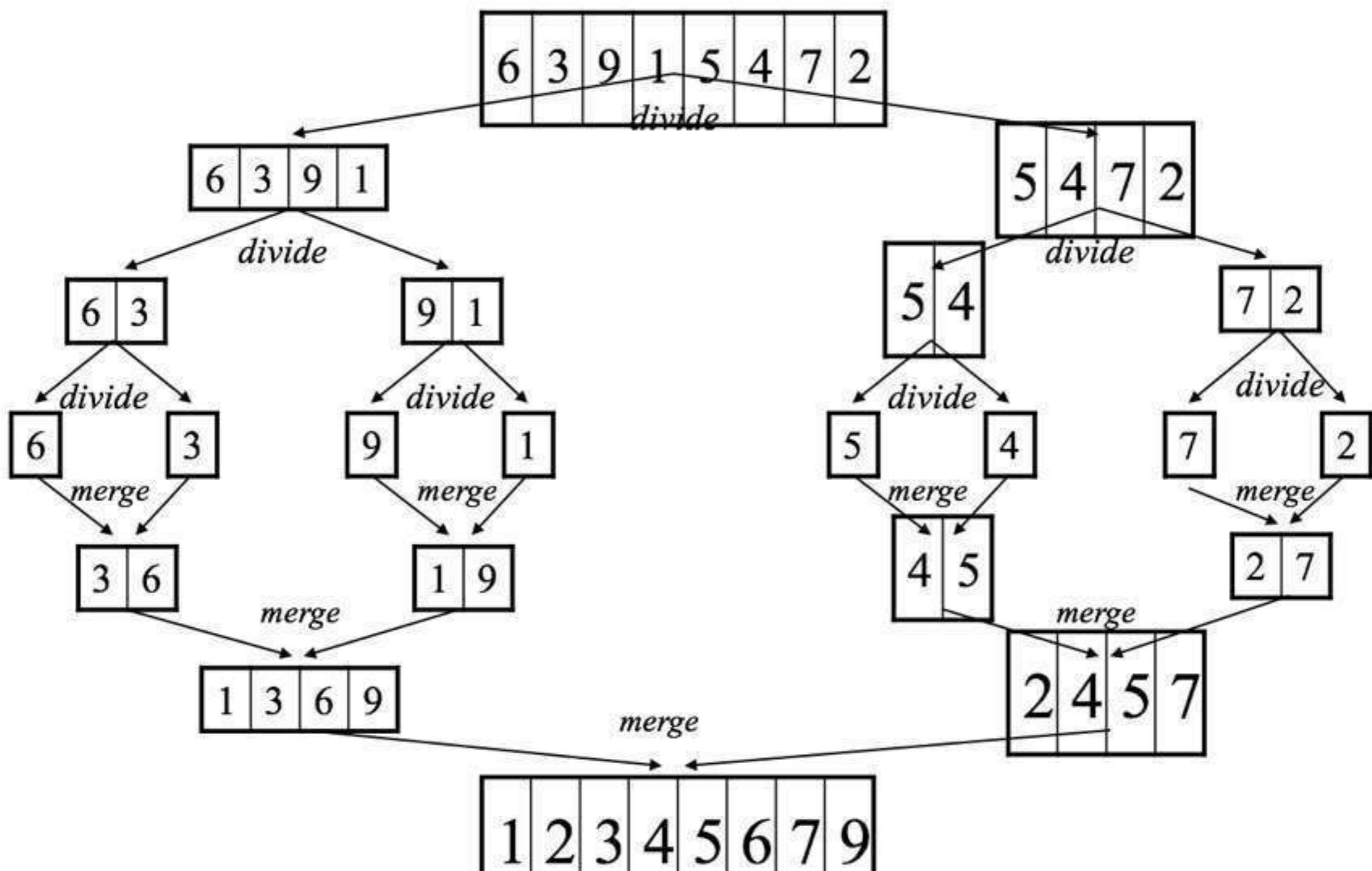
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

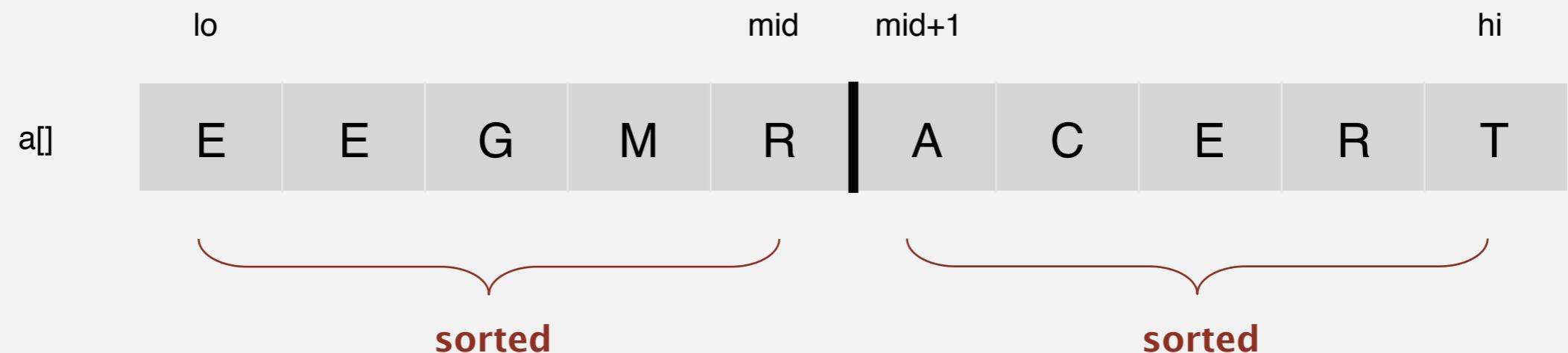


Mergesort - Example



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

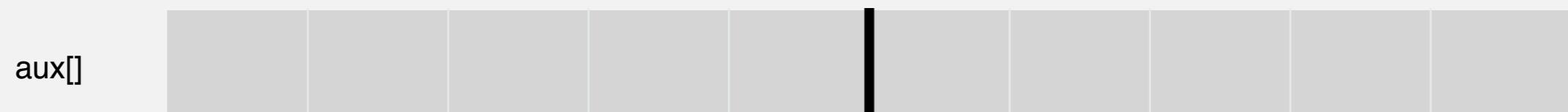


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

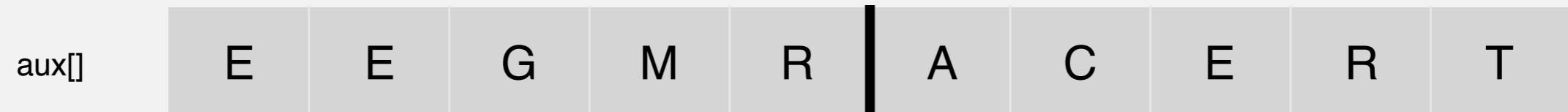
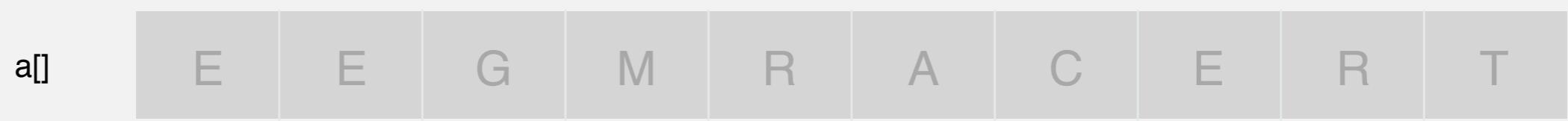


copy to auxiliary array



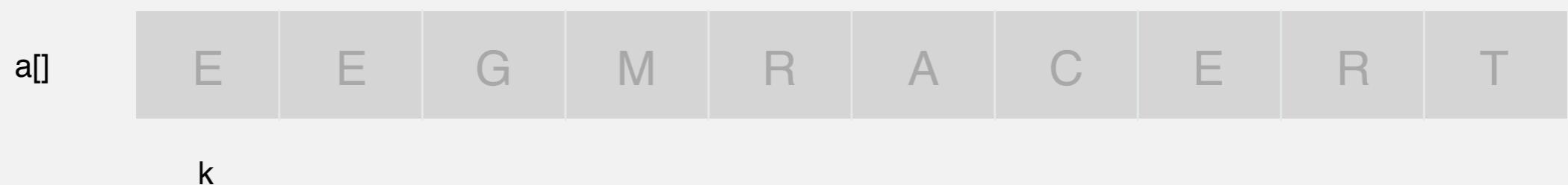
Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

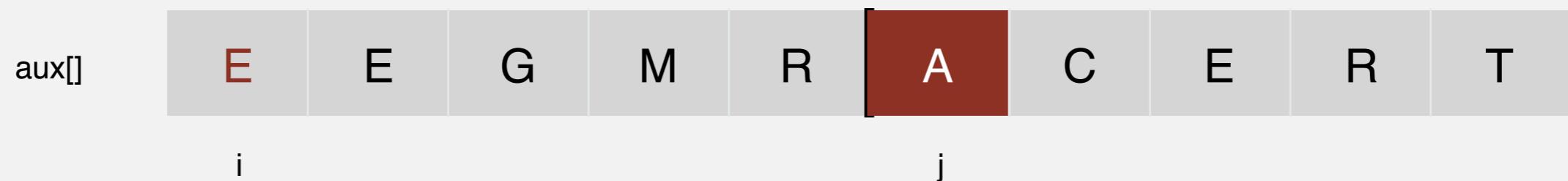


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

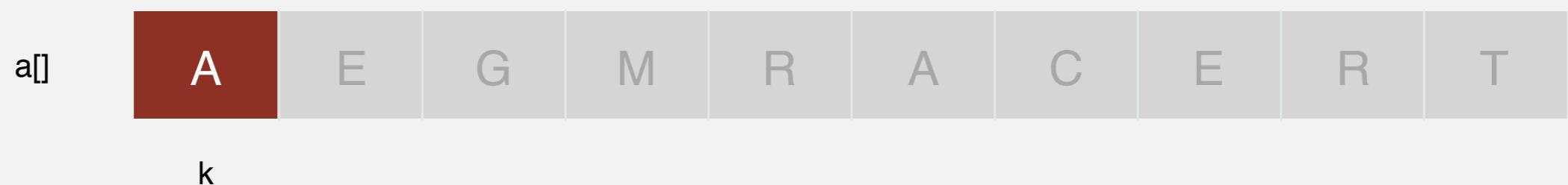


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

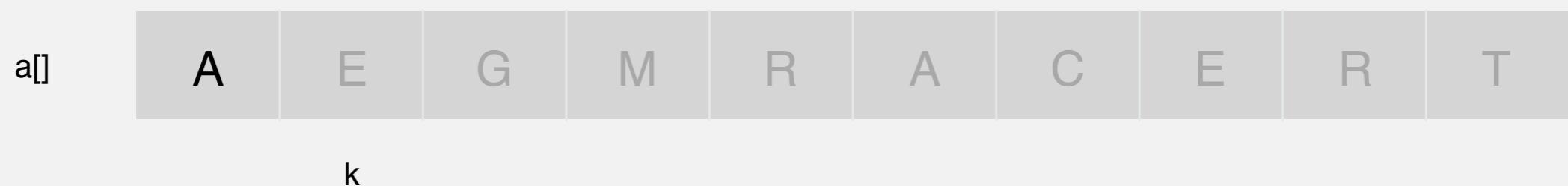


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

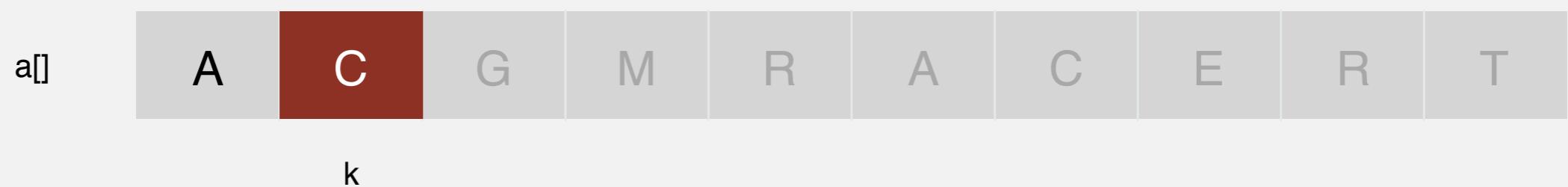


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

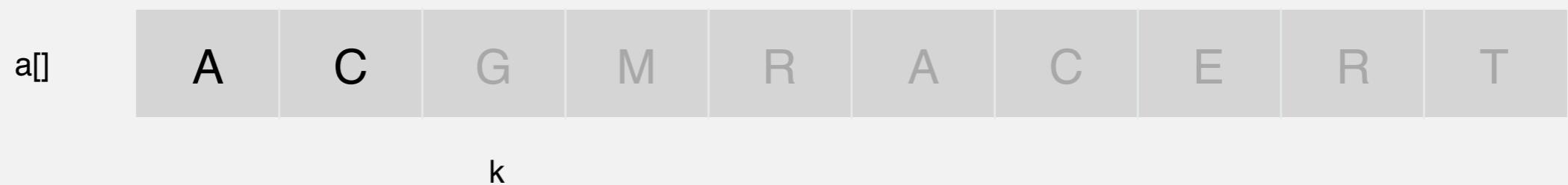


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

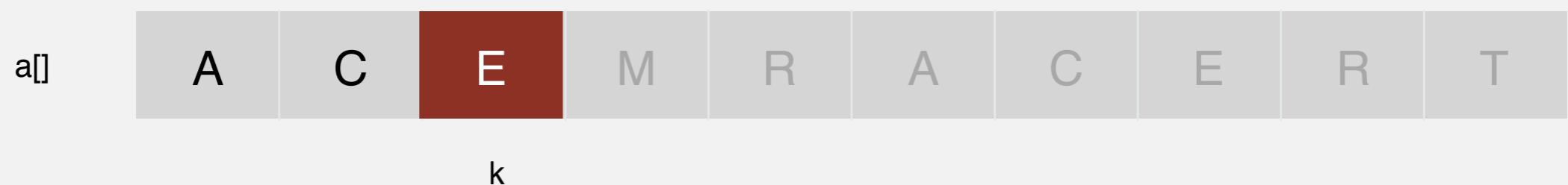


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

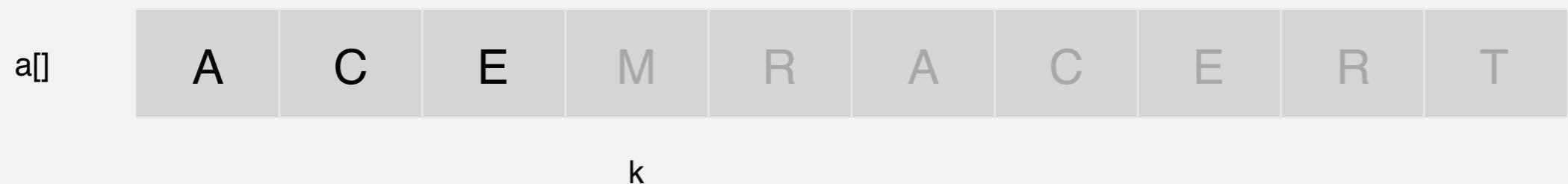


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

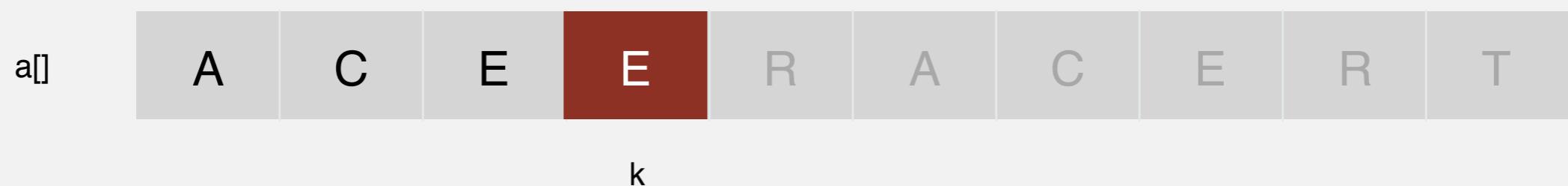


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

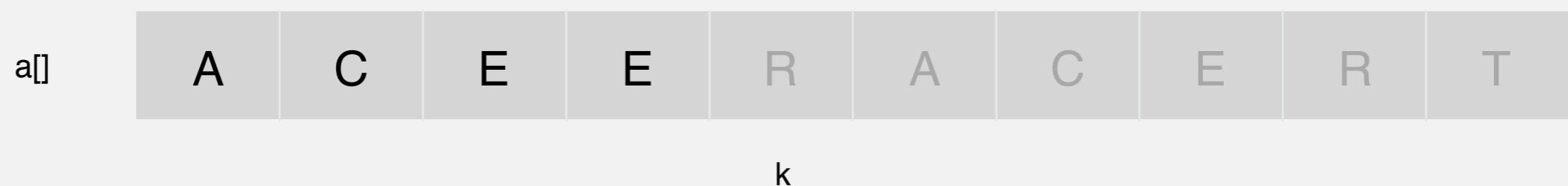


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

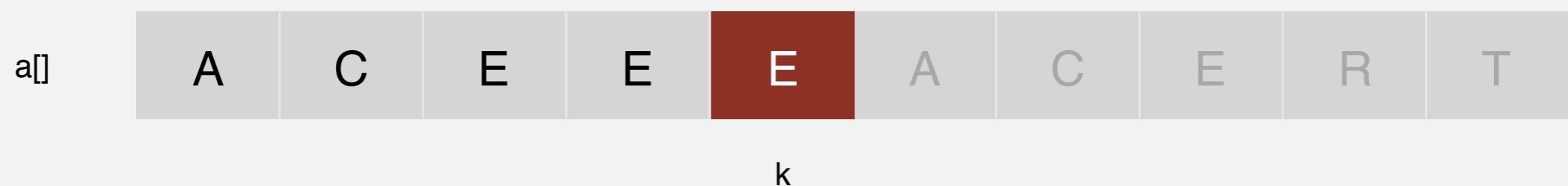


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

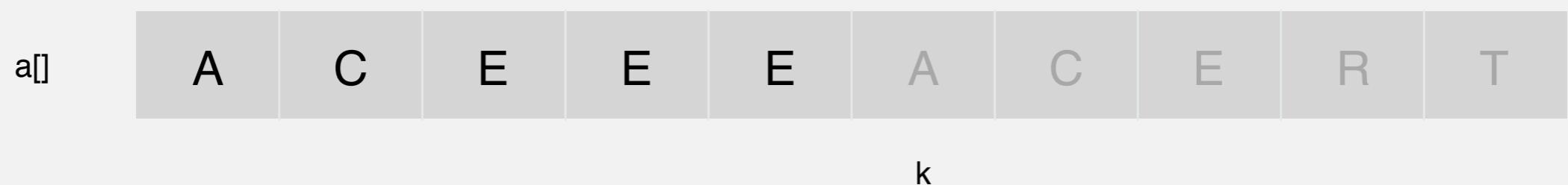


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.



compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

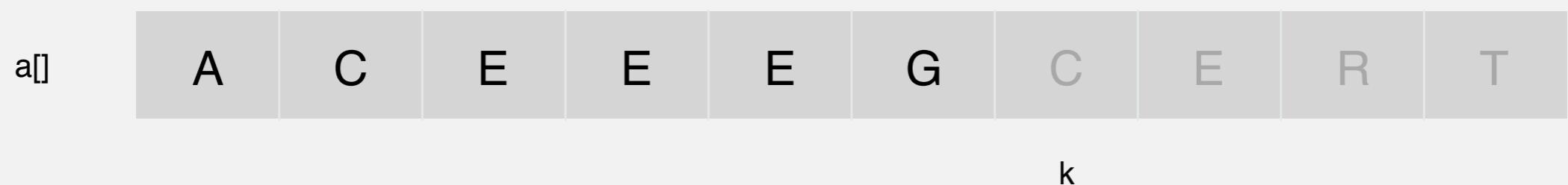


compare minimum in each subarray

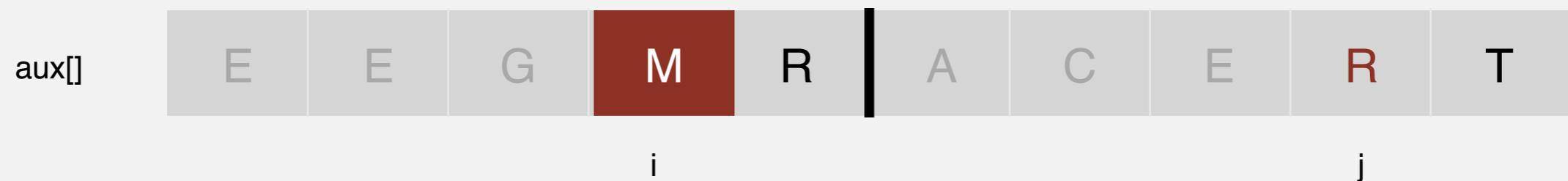


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

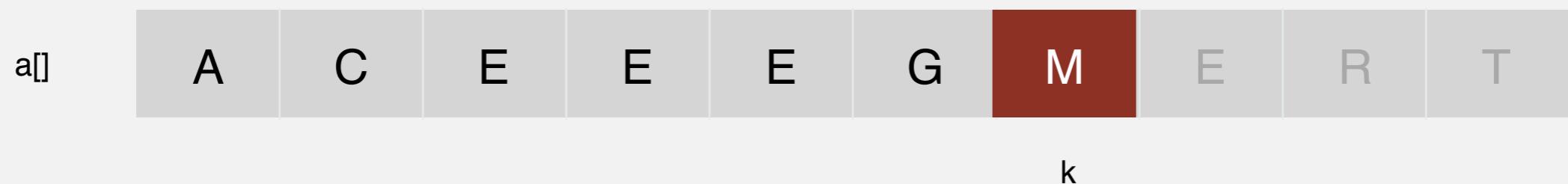


compare minimum in each subarray



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

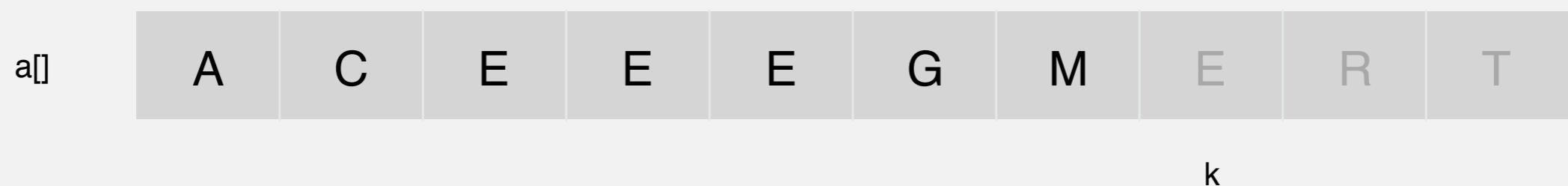


compare minimum in each subarray

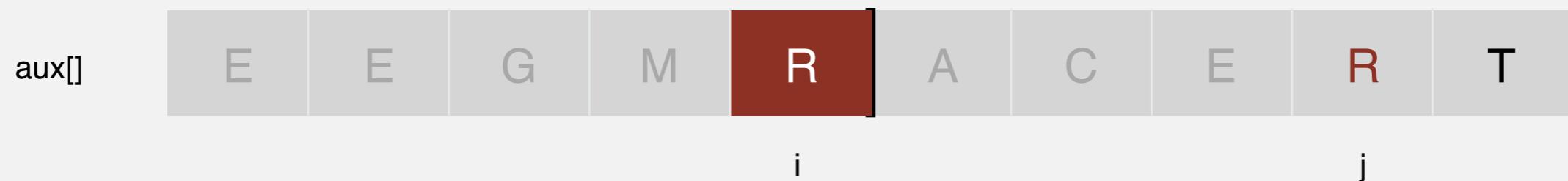


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

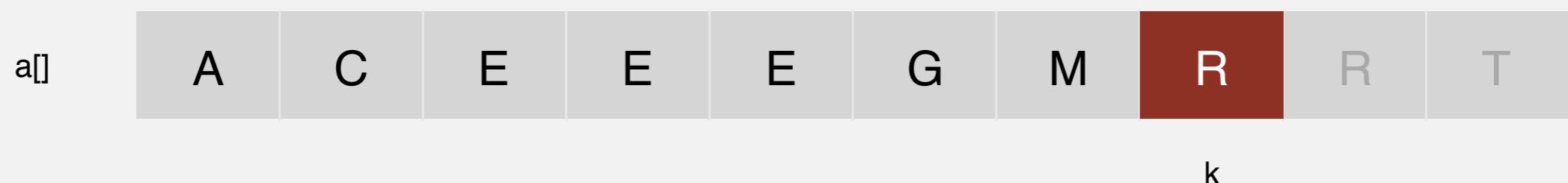


compare minimum in each subarray

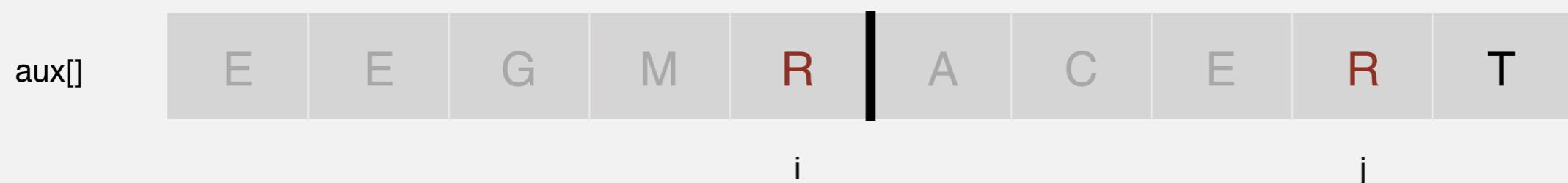


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

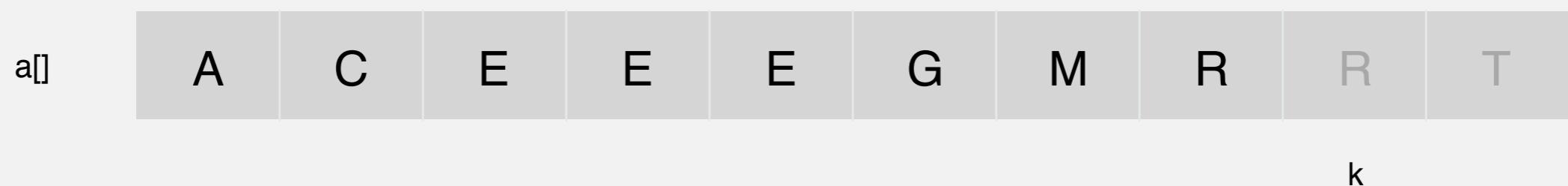


compare minimum in each subarray

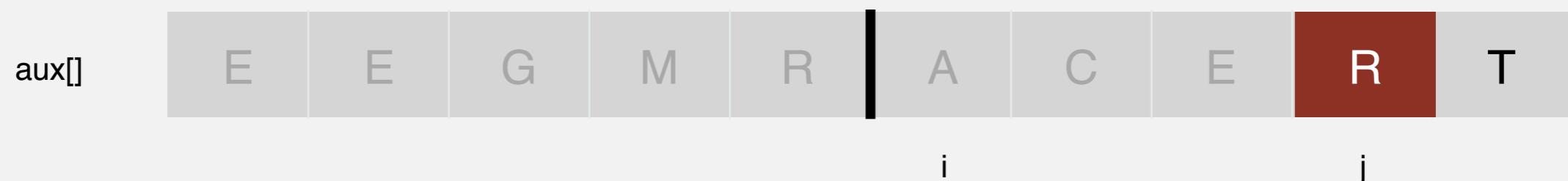


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

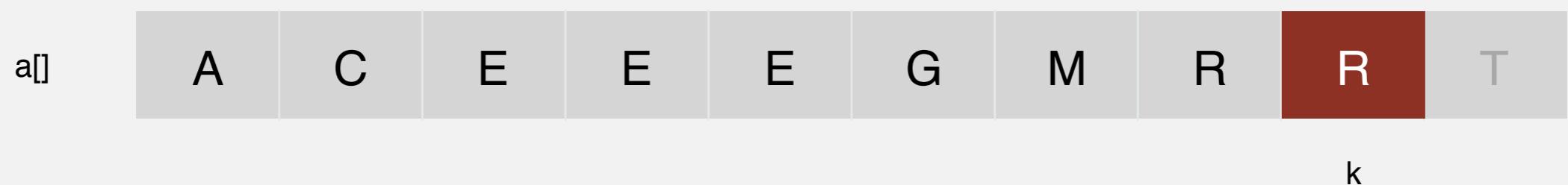


one subarray exhausted, take from other

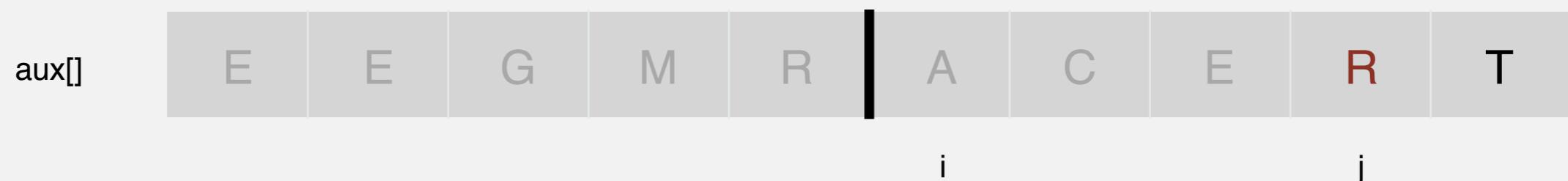


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

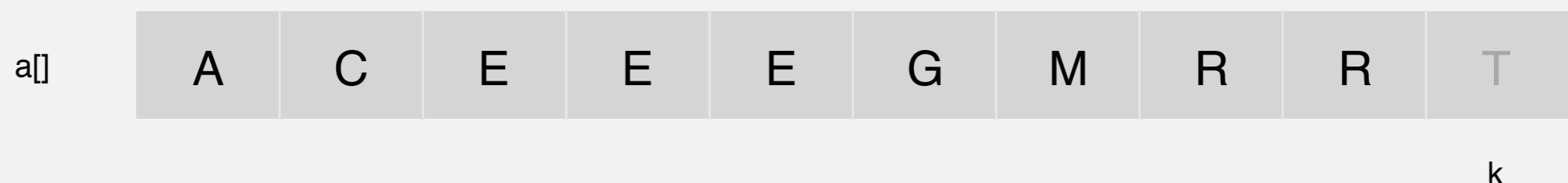


one subarray exhausted, take from other

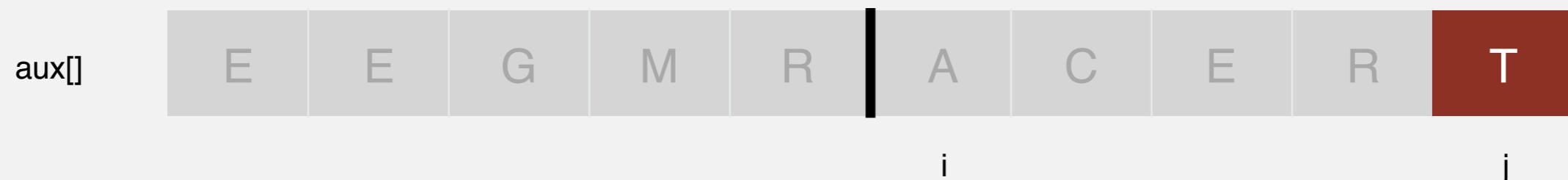


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

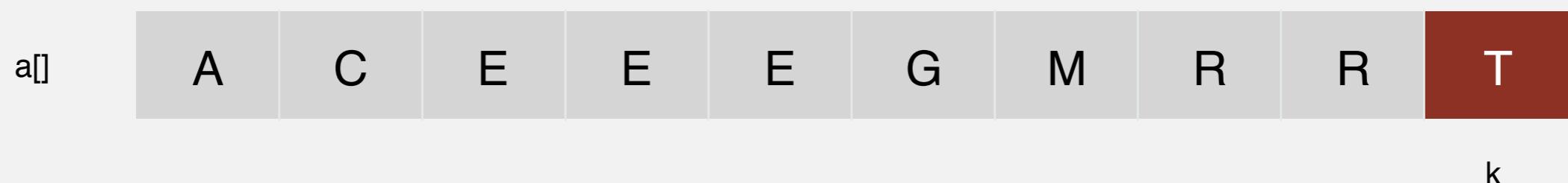


one subarray exhausted, take from other

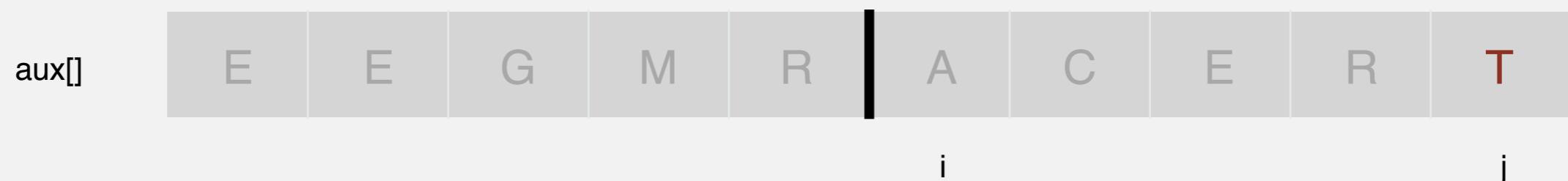


Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

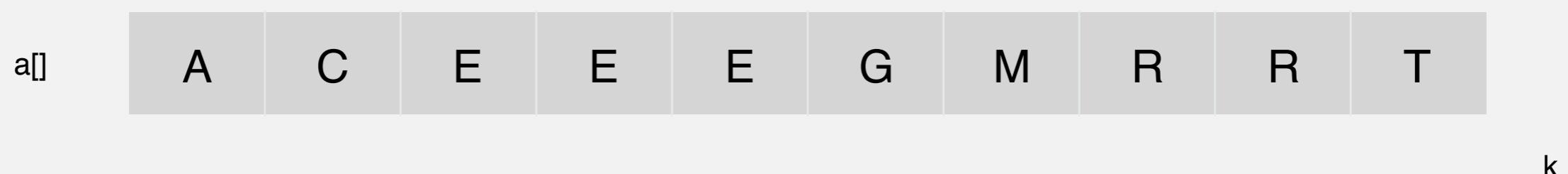


one subarray exhausted, take from other



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

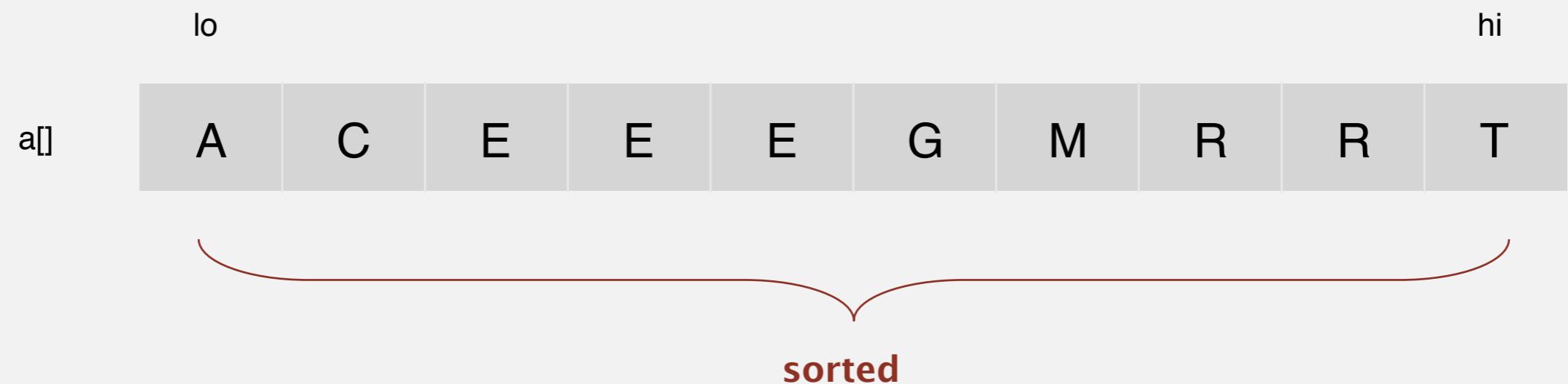


both subarrays exhausted, done



Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.



Merging

Q. How to combine two sorted subarrays into a sorted whole.

A. Use an auxiliary array.

a[]											aux[]											
k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
input	E	E	G	M	R	A	C	E	R	T	-	-	-	-	-	-	-	-	-	-	-	
copy	E	E	G	M	R	A	C	E	R	T	E	E	G	M	R	A	C	E	R	T	-	
0	A										0	5										
1	A	C									0	6	E	E	G	M	R	A	C	E	R	T
2	A	C	E								0	7	E	E	G	M	R	C	E	R	T	-
3	A	C	E	E	E						1	7	E	E	G	M	R		E	R	T	-
4	A	C	E	E	E	E					2	7		E	G	M	R		E	R	T	-
5	A	C	E	E	E	E	G				2	8			G	M	R		E	R	T	-
6	A	C	E	E	E	E	G	M			3	8			G	M	R		R	T	-	-
7	A	C	E	E	E	E	G	M	R		4	8				M	R		R	T	-	-
8	A	C	E	E	E	E	G	M	R	R	5	8					R		R	T	-	-
9	A	C	E	E	E	E	G	M	R	R	T	6	10							T		
merged result	A	C	E	E	E	E	G	M	R	R	T											

Abstract in-place merge trace

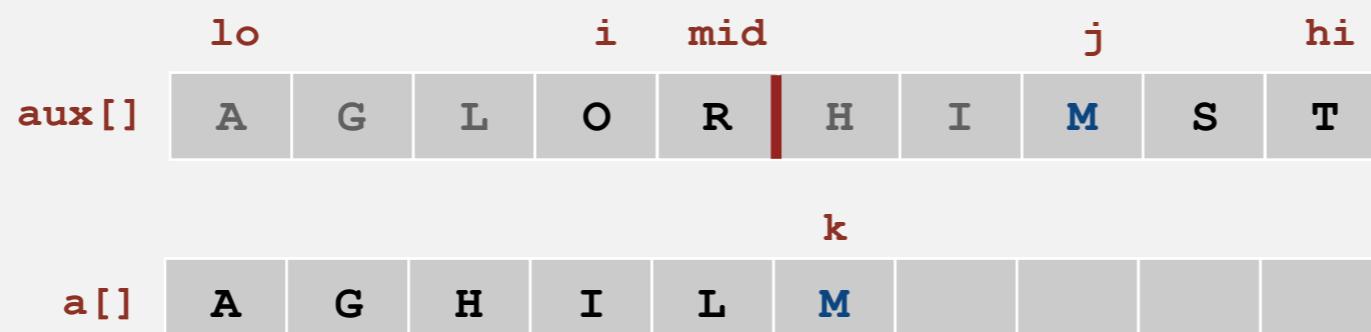
Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    assert isSorted(a, lo, mid);      // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi);   // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++)                                copy
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)                                merge
    {
        if (i > mid)                      a[k] = aux[j++];
        else if (j > hi)                  a[k] = aux[i++];
        else if (less(aux[j], aux[i]))    a[k] = aux[j++];
        else                            a[k] = aux[i++];
    }

    assert isSorted(a, lo, hi);      // postcondition: a[lo..hi] sorted
}
```

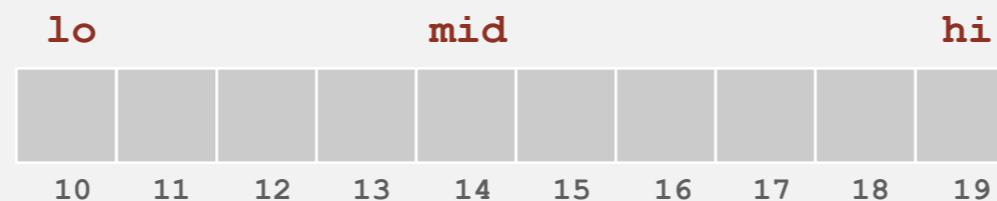


Mergesort: Java implementation

```
public class Merge
{
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
    { /* as before */ }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a)
    {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```



Mergesort: trace

	a[]																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
lo	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
hi	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 0, 0, 1)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 2, 2, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E	
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E	
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E	
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E	
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E	
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E	
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L	
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P	
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
merge(a, 0, 7, 15)	A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

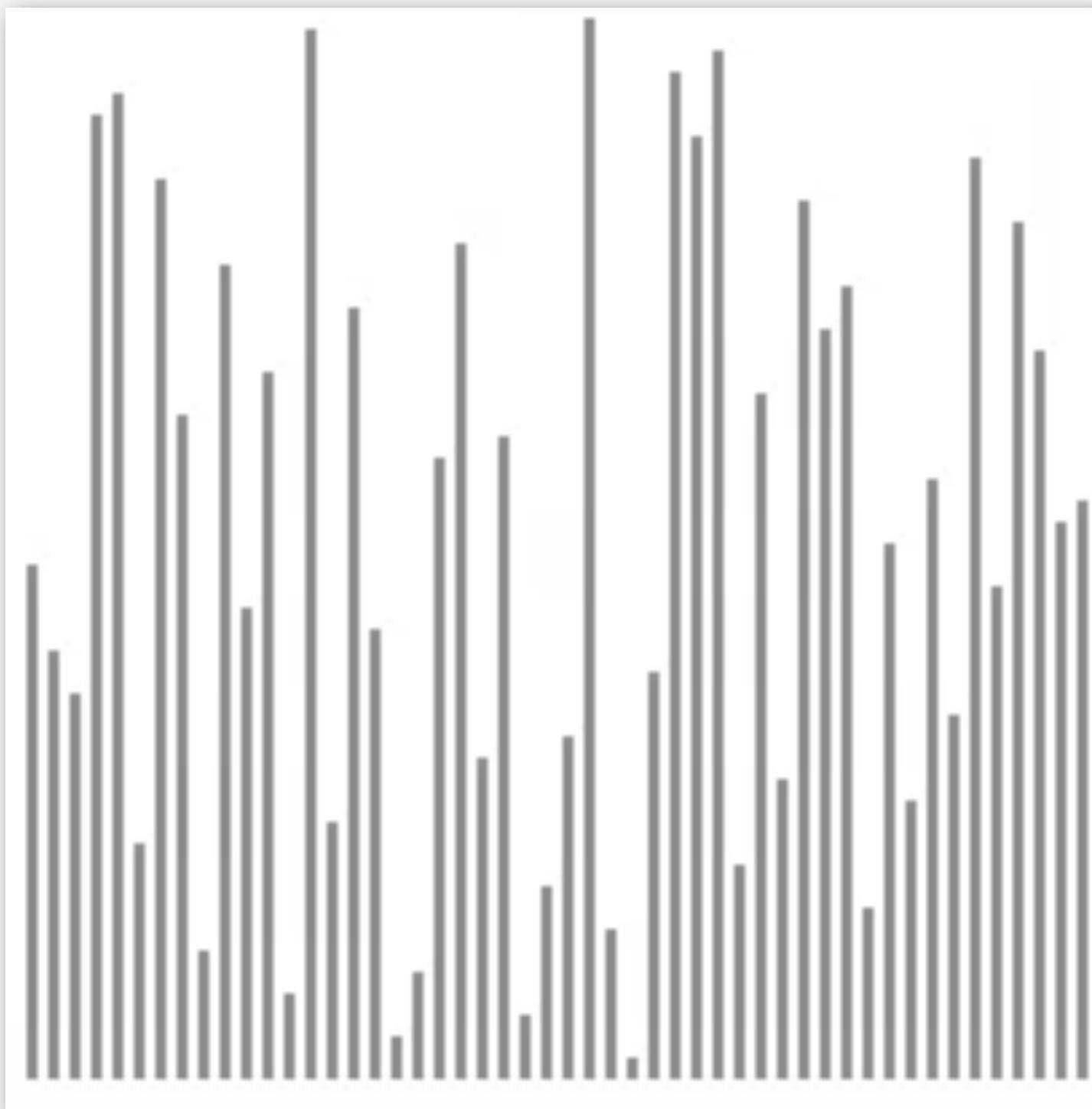
Trace of merge results for top-down mergesort



result after recursive call

Mergesort: animation

50 random items

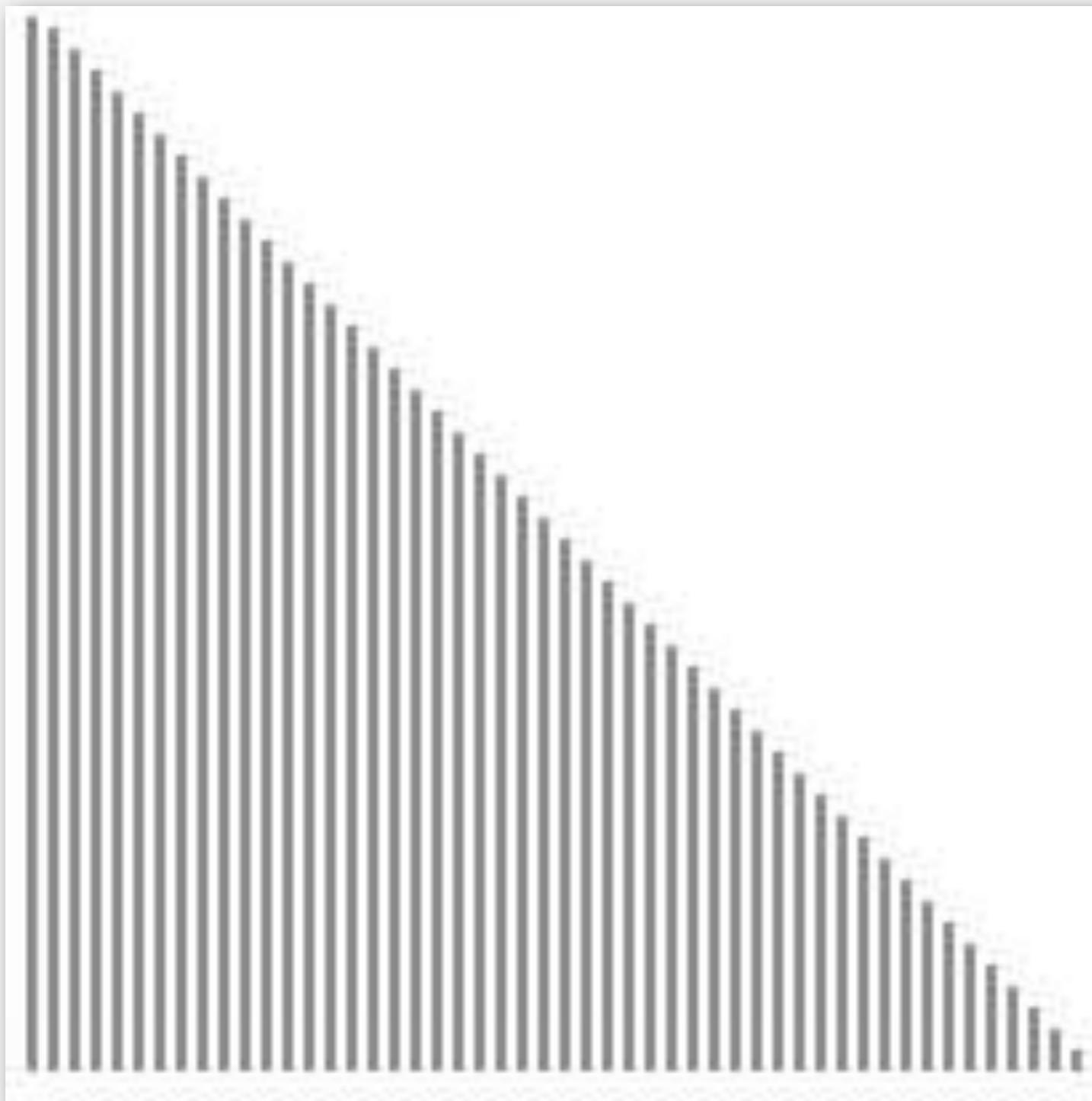


<http://www.sorting-algorithms.com/merge-sort>

- ▲ algorithm position
- █ in order
- ▒ current subarray
- ░ not in order

Mergesort: animation

50 reverse-sorted items



- ▲ algorithm position
- █ in order
- █ current subarray
- █ not in order

<http://www.sorting-algorithms.com/merge-sort>

Mergesort: empirical analysis

Running time estimates:

- Laptop executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of compares and array accesses

Proposition. Mergesort uses at most $N \lg N$ compares and $6N \lg N$ array accesses to sort any array of size N .

Pf sketch. The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size N satisfy the recurrences:

$$C(N) \leq C(\lceil N/2 \rceil) + C(\lfloor N/2 \rfloor) + N \text{ for } N > 1, \text{ with } C(1) = 0.$$



$$A(N) \leq A(\lceil N/2 \rceil) + A(\lfloor N/2 \rfloor) + 6N \text{ for } N > 1, \text{ with } A(1) = 0.$$

We solve the recurrence when N is a power of 2.

$$D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    assert isSorted(a, lo, mid);      // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi);   // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];           ← copy 2N

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if         (i > mid)          a[k] = aux[j++];
        else if (j > hi)            a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                          a[k] = aux[i++];   ← merge 2N
    }
}

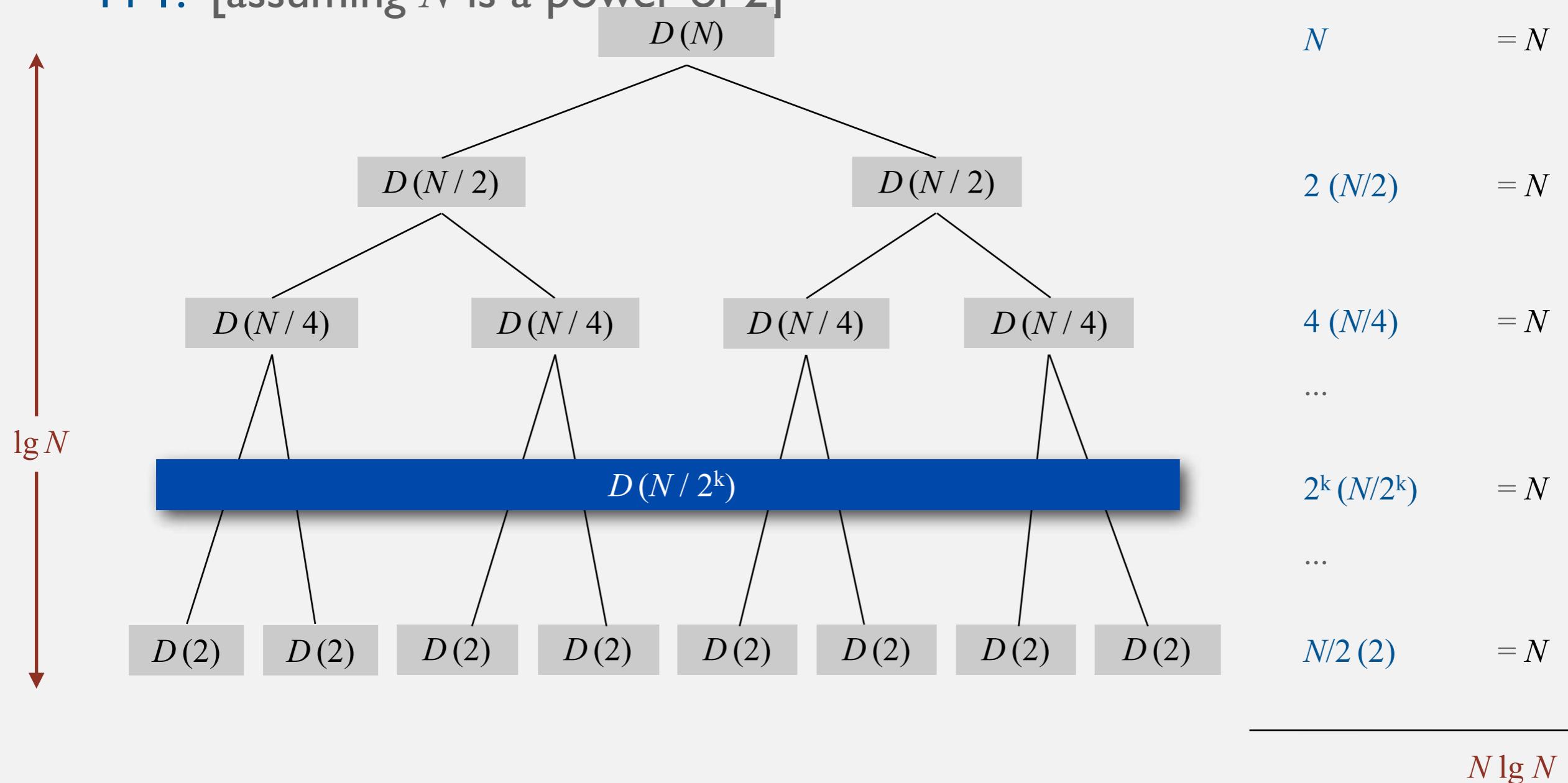
assert isSorted(a, lo, hi);      // postcondition: a[lo..hi] sorted
}
```

Proof: Each merge uses at most $6N$ array accesses (2N for the copy, 2N for the move back, and at most 2N for compares). The result follows from the same argument as for PROPOSITION F.

Divide-and-conquer recurrence: proof by picture

Proposition. If $D(N)$ satisfies $D(N) = 2 D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf I. [assuming N is a power of 2]



Divide-and-conquer recurrence: proof by expansion

Proposition. If $D(N)$ satisfies $D(N) = 2 D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 2. [assuming N is a power of 2]

$$D(N) = 2 D(N/2) + N$$

given

$$D(N)/N = 2 D(N/2)/N + 1$$

divide both sides by N

$$= D(N/2)/(N/2) + 1$$

algebra

$$= D(N/4)/(N/4) + 1 + 1$$

apply to first term

$$= D(N/8)/(N/8) + 1 + 1 + 1$$

apply to first term again

...

$$= D(N/N)/(N/N) + 1 + 1 + \dots + 1$$

$$= \lg N$$

stop applying, $D(1) = 0$

Divide-and-conquer recurrence: proof by induction

Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 3. [assuming N is a power of 2]

- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N \lg N$.
- Goal: show that $D(2N) = (2N) \lg (2N)$.

$$\begin{aligned} D(2N) &= 2D(N) + 2N \\ &= 2N \lg N + 2N \\ &= 2N(\lg(2N) - 1) + 2N \\ &= 2N \lg(2N) \end{aligned}$$

given

inductive hypothesis

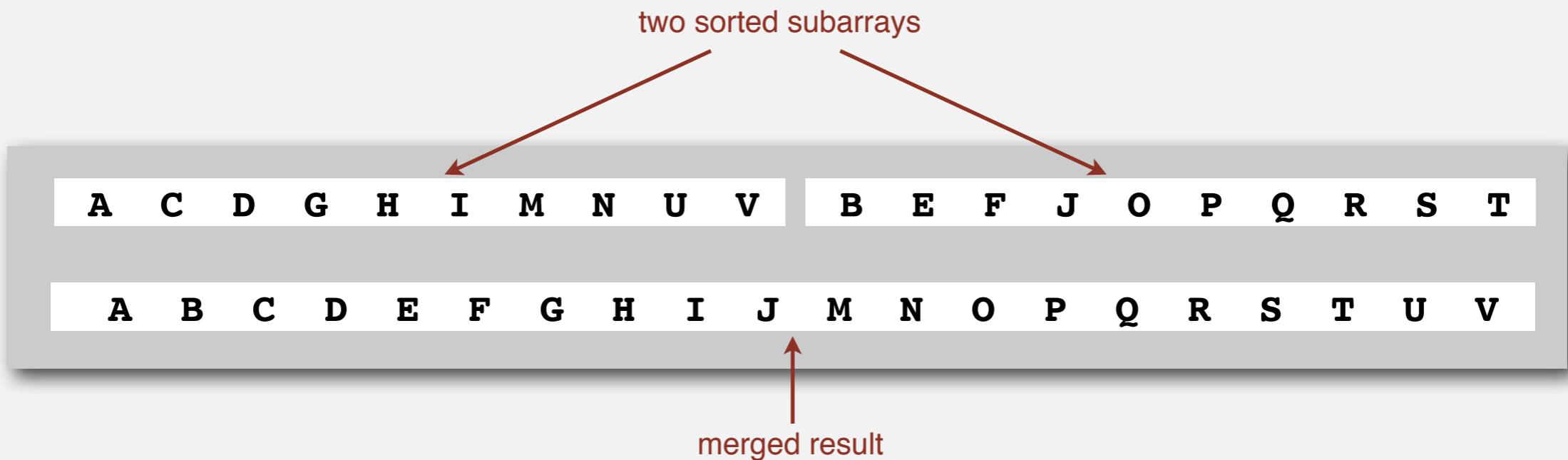
algebra

QED

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to N .

Pf. The array `aux[]` needs to be of size N for the last merge.



Def. A sorting algorithm is **in-place** if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

Challenge for the bored. In-place merge. [Kronrod, 1969]

Mergesort: practical improvements

Use insertion sort for small subarrays.

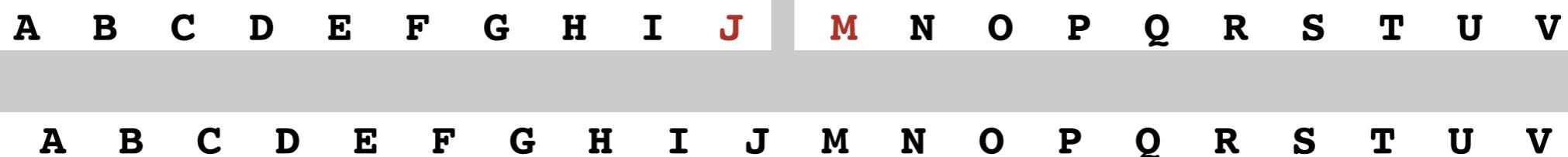
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1) Insertion.sort(a, lo, hi);
    int mid = lo + (hi - lo) / 2;
    sort (a, aux, lo, mid);
    sort (a, aux, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Mergesort: practical improvements

Stop if already sorted.

- Is biggest item in first half \leq smallest item in second half?
- Helps for partially-ordered arrays.



```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid+1, hi);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```

Mergesort: practical improvements

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid) aux[k] = a[j++];
        else if (j > hi) aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else aux[k] = a[i++];
    }
}

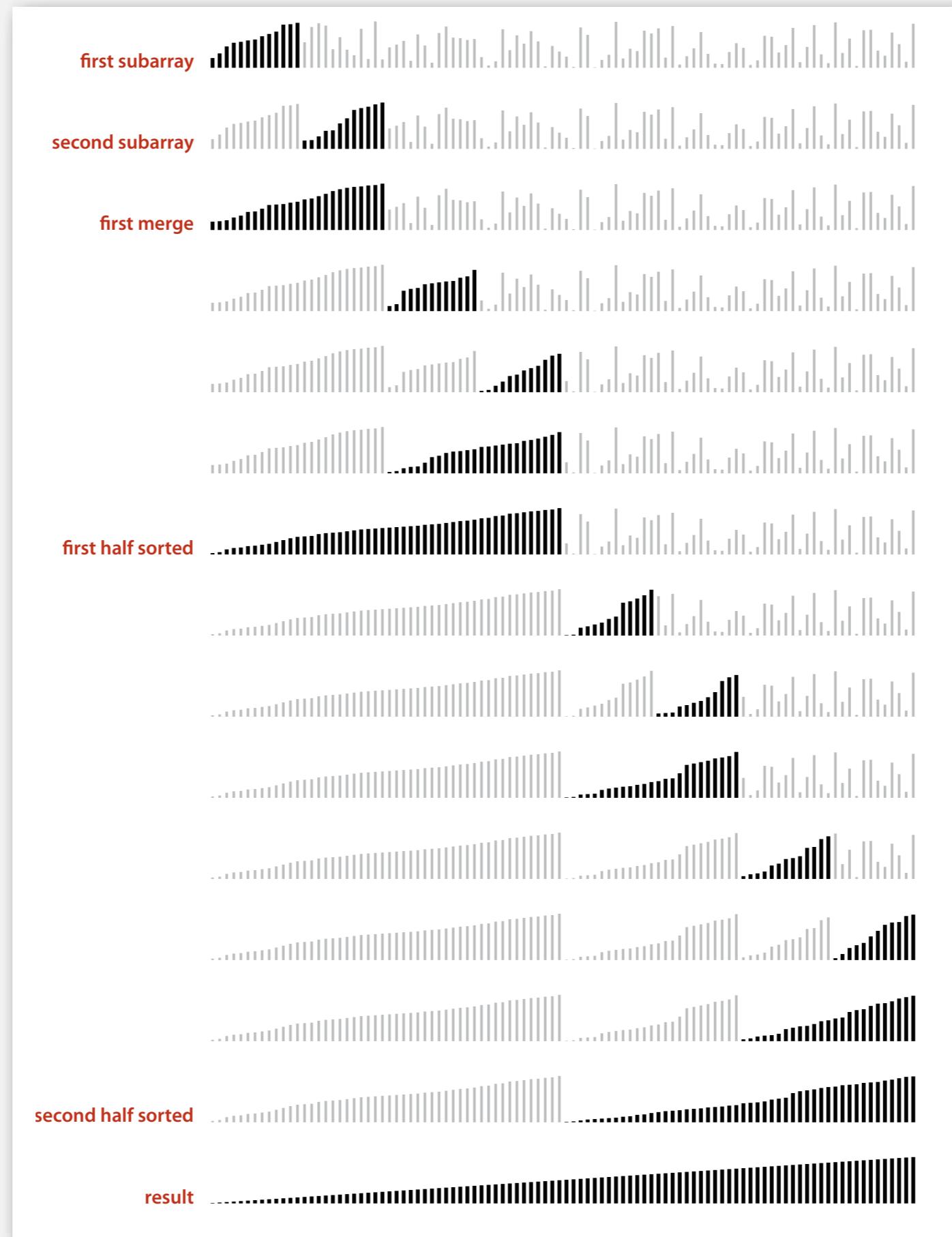
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(aux, a, lo, mid);
    sort(aux, a, mid+1, hi);
    merge(aux, a, lo, mid, hi);
}
```

switch roles of aux[] and a[]



← merge from a[] to aux[]

Mergesort: visualization



Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16,

	a[i]																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
sz = 1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E	
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E	
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E	
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E	
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L	
sz = 2	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L	
merge(a, 0, 1, 3)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L	
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L	
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L	
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P	
sz = 4	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P	
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
sz = 8	A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Bottom line. No recursion needed!

Bottom-up mergesort: Java implementation

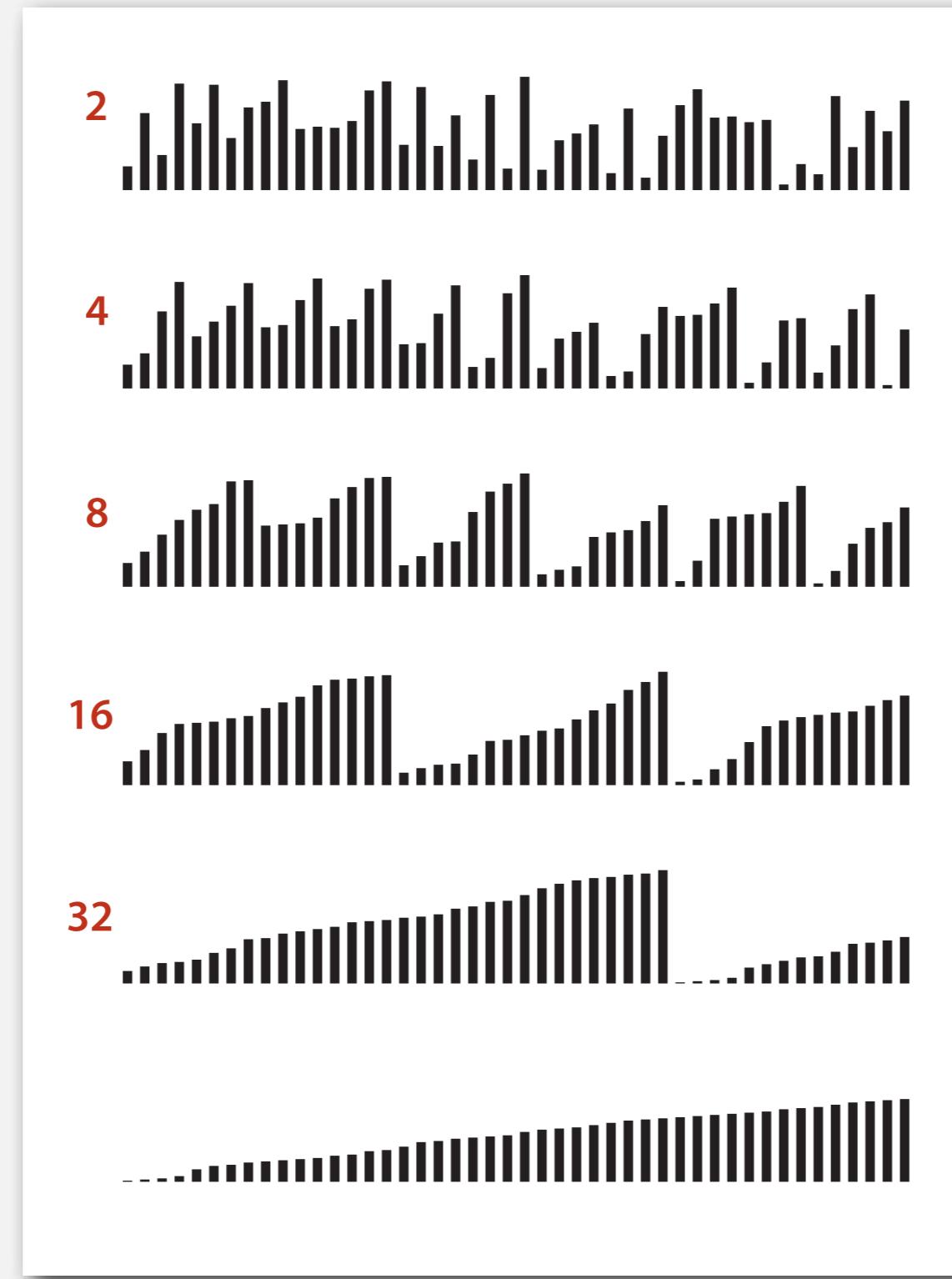
```
public class MergeBU
{
    private static Comparable[] aux;

    private static void merge(Comparable[] a, int lo, int mid, int hi)
    { /* as before */ }

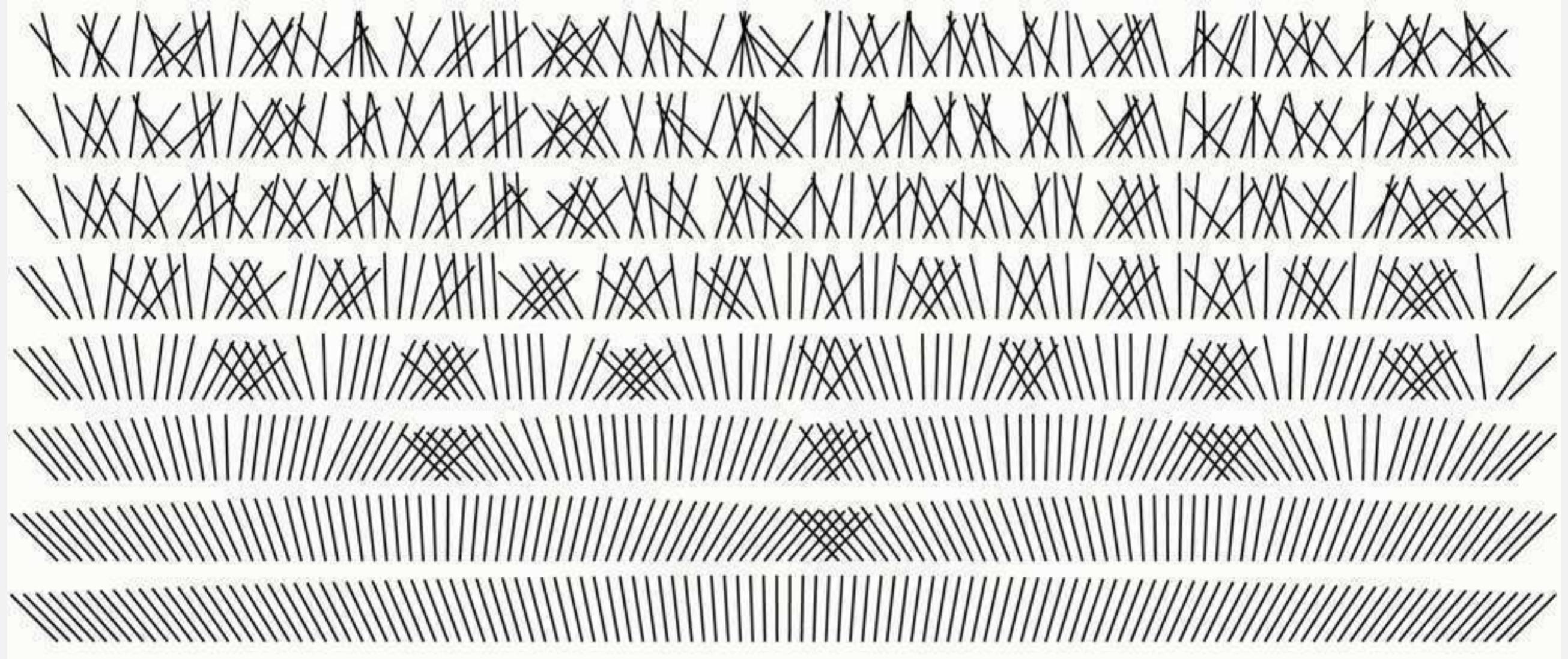
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo += sz+sz)
                merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
```

Bottom line. Concise industrial-strength code, if you have the space.

Bottom-up mergesort: visual trace

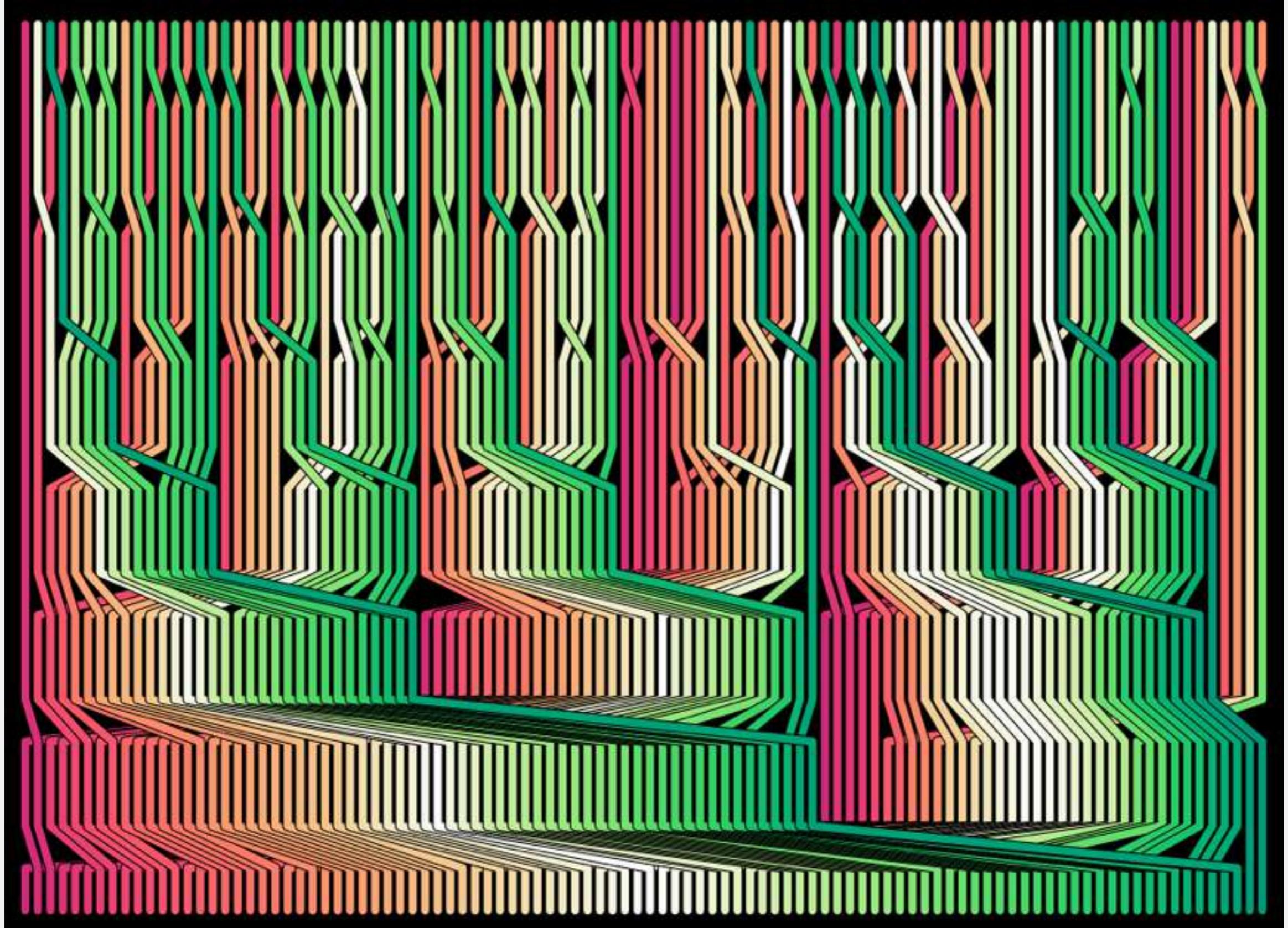


Bottom-up mergesort: visual trace



<http://bl.ocks.org/mbostock/39566aca95eb03ddd526>

Bottom-up mergesort: visual trace



<http://bl.ocks.org/mbostock/e65d9895da07c57e94bd>

Complexity of sorting

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem X .

Model of computation. Allowable operations.

Cost model. Operation count(s).

Upper bound. Cost guarantee provided by **some algorithm** for X .

Lower bound. Proven limit on cost guarantee of **all algorithms** for X .

Optimal algorithm. Algorithm with best possible cost guarantee for X .

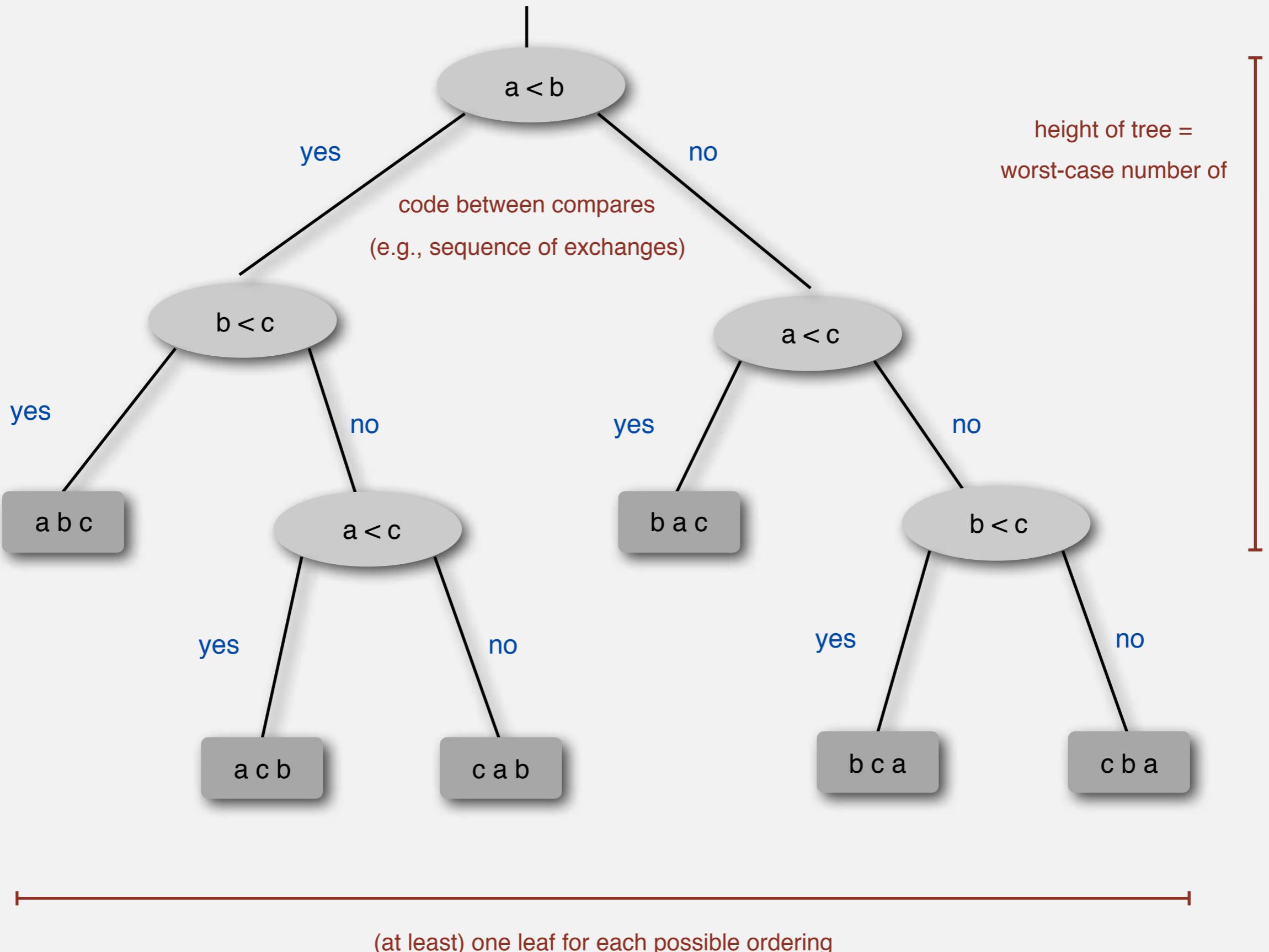
Example: sorting.

- Model of computation: decision tree. ←
can access information
only through compares
(e.g., Java Comparable framework)
- Cost model: # compares.
- Upper bound: $\sim N \lg N$ from mergesort.
- Lower bound: ?
- Optimal algorithm: ?



lower bound \sim upper bound

Decision tree (for 3 distinct items a, b, and c)

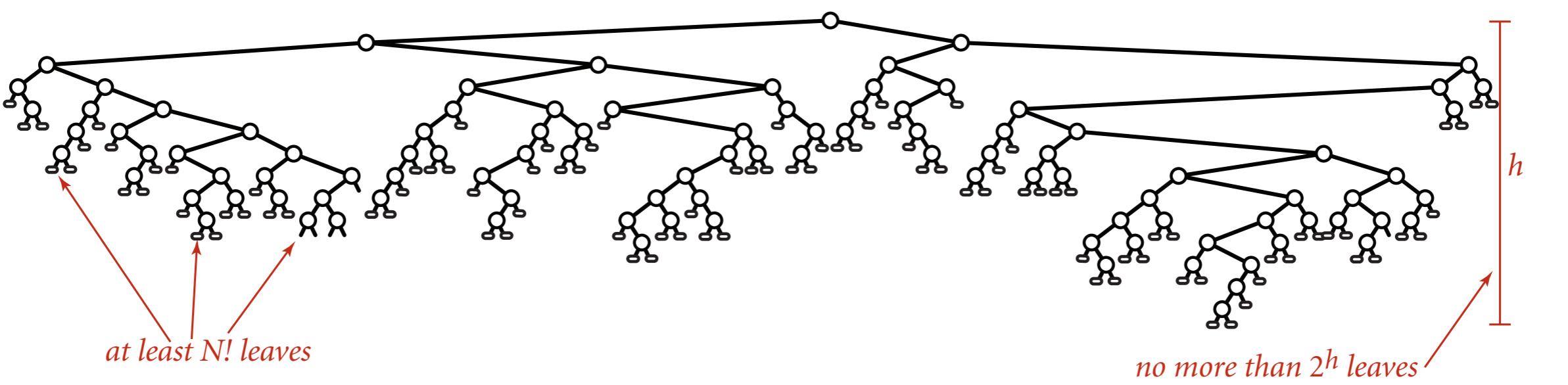


Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg(N!) \sim N \lg N$ compares in the worst-case.

Pf.

- Assume array consists of N distinct values a_1 through a_N .
- Worst case dictated by **height h** of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings \Rightarrow at least $N!$ leaves.



Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use at least $\lg(N!) \sim N \lg N$ compares in the worst-case.

Pf.

- Assume array consists of N distinct values a_1 through a_N .
- Worst case dictated by **height h** of decision tree.
- Binary tree of height h has at most 2^h leaves.
- $N!$ different orderings \Rightarrow at least $N!$ leaves.

$$\begin{aligned} 2^h &\geq \# \text{leaves} \geq N! \\ \Rightarrow h &\geq \lg(N!) \sim N \lg N \end{aligned}$$



Stirling's formula

Complexity of sorting

Model of computation. Allowable operations.

Cost model. Operation count(s).

Upper bound. Cost guarantee provided by some algorithm for X .

Lower bound. Proven limit on cost guarantee of all algorithms for X .

Optimal algorithm. Algorithm with best possible cost guarantee for X .

Example: sorting.

- Model of computation: decision tree.
- Cost model: # compares.
- Upper bound: $\sim N \lg N$ from mergesort.
- Lower bound: $\sim N \lg N$.
- Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.

Complexity results in context

Other operations? Mergesort is optimal with respect to number of compares (e.g., but not with respect to number of array accesses).

Space?

- Mergesort is **not optimal** with respect to space usage.
- Insertion sort, selection sort, and shellsort are space-optimal.

Challenge. Find an algorithm that is both time- and space-optimal.

[stay tuned]

Lessons. Use theory as a guide.

Ex. Don't try to design sorting algorithm that guarantees $\frac{1}{2} N \lg N$ compares.

Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about:

- The initial order of the input.
- The distribution of key values.
- The representation of the keys.

Partially-ordered arrays. Depending on the initial order of the input, we may not need $N \lg N$ compares.

insertion sort requires only $N-1$ compares if input array is sorted

Duplicate keys. Depending on the input distribution of duplicates, we may not need $N \lg N$ compares.

stay tuned for 3-way quicksort

Digital properties of keys. We can use digit/character compares instead of key compares for numbers and strings.

stay tuned for radix sorts

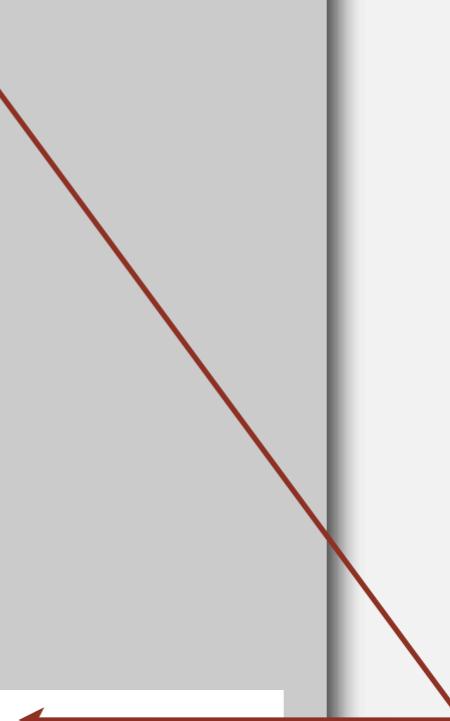
Comparable interface: review

Comparable interface: sort using a type's natural order.

```
public class Date implements Comparable<Date>
{
    private final int month, day, year;

    public Date(int m, int d, int y)
    {
        month = m;
        day   = d;
        year  = y;
    }

    ...
    public int compareTo(Date that)
    {
        if (this.year < that.year) return -1;
        if (this.year > that.year) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day   < that.day ) return -1;
        if (this.day   > that.day ) return +1;
        return 0;
    }
}
```



natural order

Comparator interface

Comparator interface: sort using an **alternate order**.

```
public interface Comparator<Key>
```

```
    int compare(Key v, Key w)
```

compare keys v and w

Required property. Must be a **total order**.

Ex. Sort strings by:

- Natural order.
- Case insensitive.
- Spanish.
- British phone book.
- ...

Now is the time

is Now the time

café cafetero cuarto churro nube ñoño

McKinley Mackintosh

pre-1994 order for
digraphs ch and ll and rr

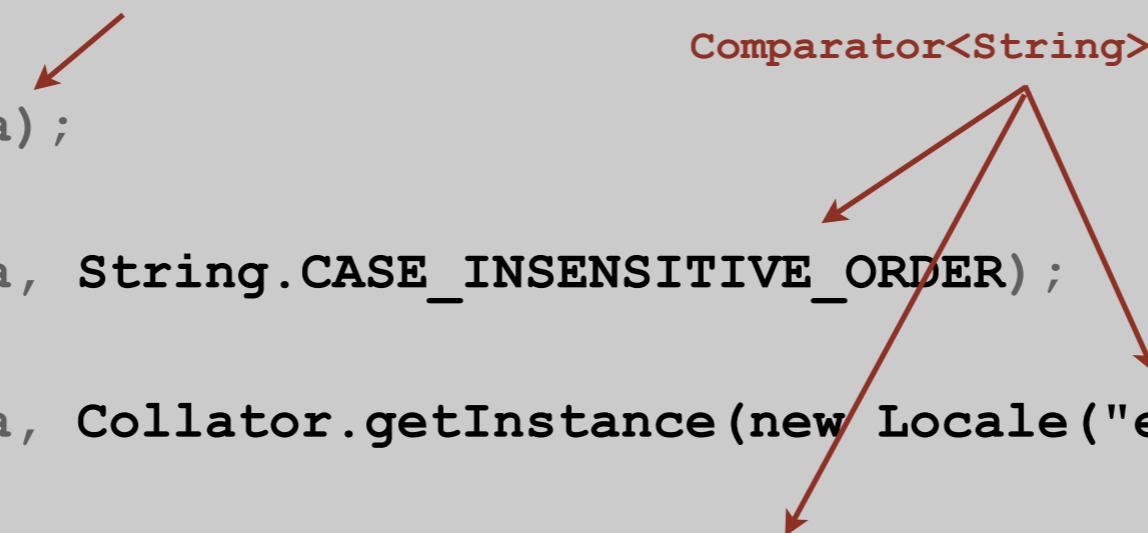


Comparator interface: system sort

To use with Java system sort:

- Create **Comparator** object.
- Pass as second argument to **Arrays.sort()**.

```
String[] a;  
...  
Arrays.sort(a);           uses natural order  
...  
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);    uses alternate order defined by  
...  
Arrays.sort(a, Collator.getInstance(new Locale("es")));  
...  
Arrays.sort(a, new BritishPhoneBookOrder());  
...
```



Bottom line. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.

Comparator interface: using with our sorting libraries

To support comparators in our sort implementations:

- Use `Object` instead of `Comparable`.
- Pass `Comparator` to `sort()` and `less()` and use it in `less()`.

insertion sort using a Comparator

```
public static void sort(Object[] a, Comparator comparator)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
            exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{ return c.compare(v, w) < 0; }

private static void exch(Object[] a, int i, int j)
{ Object swap = a[i]; a[i] = a[j]; a[j] = swap; }
```

Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
public class Student
{
    public static final Comparator<Student> BY_NAME      = new ByName();
    public static final Comparator<Student> BY_SECTION = new BySection();
    private final String name;
    private final int section;
    ...
    private static class ByName implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        { return v.name.compareTo(w.name); }
    }

    private static class BySection implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        { return v.section - w.section; }
    }
}
```

one Comparator for the class

this technique works here since no danger of overflow

Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the **Comparator** interface.
- Implement the **compare()** method.

```
Arrays.sort(a, Student.BY_NAME);
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Arrays.sort(a, Student.BY_SECTION);
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

Stability

A typical application. First, sort by name; then sort by section.

`Selection.sort(a, Student.BY_NAME);`

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

`Selection.sort(a, Student.BY_SECTION);`

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Andrews	3	A	664-480-0023	097 Little
Kanaga	3	B	898-122-9643	22 Brown
Gazsi	4	B	766-093-9873	101 Brown
Battle	4	C	874-088-1212	121 Whitman

@#%&@! Students in section 3 no longer sorted by name.

A **stable** sort preserves the relative order of items with equal keys.

Stability

Q. Which sorts are stable?

A. Insertion sort and mergesort (but not selection sort or shellsort).

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

Note. Need to carefully check code ("less than" vs "less than or equal to").

Stability: insertion sort

Proposition. Insertion sort is **stable**.

```
public class Insertion
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
            for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
                exch(a, j, j-1);
    }
}
```

i	j	0	1	2	3	4
0	0	B ₁	A ₁	A ₂	A ₃	B ₂
1	0	A ₁	B ₁	A ₂	A ₃	B ₂
2	1	A ₁	A ₂	B ₁	A ₃	B ₂
3	2	A ₁	A ₂	A ₃	B ₁	B ₂
4	4	A ₁	A ₂	A ₃	B ₁	B ₂
		A ₁	A ₂	A ₃	B ₁	B ₂

Pf. Equal items never move past each other.

Stability: selection sort

Proposition. Selection sort is **not** stable.

```
public class Selection
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i < N; i++)
        {
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
        }
    }
}
```

i	min	0	1	2
0	2	B ₁	B ₂	A
1	1	A	B ₂	B ₁
2	2	A	B ₂	B ₁
		A	B ₂	B ₁

Pf by counterexample. Long-distance exchange might move an item past some equal item.

Stability: shellsort

Proposition. Shellsort sort is **not** stable.

```
public class Shell
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        int h = 1;
        while (h < N/3) h = 3*h + 1;
        while (h >= 1)
        {
            for (int i = h; i < N; i++)
            {
                for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h/3;
        }
    }
}
```

h	0	1	2	3	4
	B ₁	B ₂	B ₃	B ₄	A ₁
4	A ₁	B ₂	B ₃	B ₄	B ₁
1	A ₁	B ₂	B ₃	B ₄	B ₁
	A ₁	B ₂	B ₃	B ₄	B ₁

Pf by counterexample. Long-distance exchanges.

Stability: mergesort

Proposition. Mergesort is **stable**.

```
public class Merge
{
    private static Comparable[] aux;
    private static void merge(Comparable[] a, int lo, int mid, int hi)
    { /* as before */ }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid+1, hi);
        merge(a, lo, mid, hi);
    }

    public static void sort(Comparable[] a)
    { /* as before */ }
}
```

Pf. Suffices to verify that merge operation is stable.

Stability: mergesort

Proposition. Merge operation is stable.

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if (i > mid)                      a[k] = aux[j++];
        else if (j > hi)                  a[k] = aux[i++];
        else if (less(aux[j], aux[i]))    a[k] = aux[j++];
        else                                a[k] = aux[i++];
    }
}
```

0	1	2	3	4		5	6	7	8	9	10
A ₁	A ₂	A ₃	B	D		A ₄	A ₅	C	E	F	G

Pf. Takes from left subarray if equal keys.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

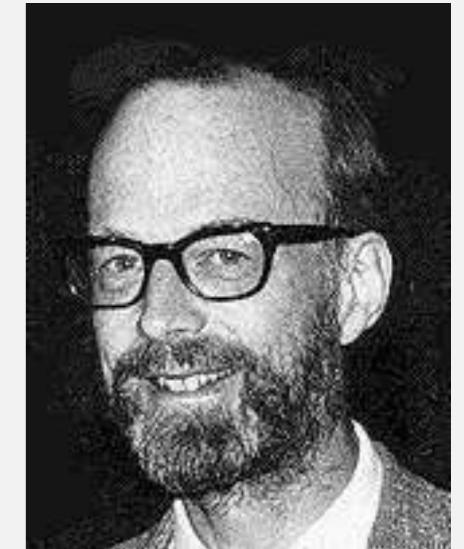
QUICKSORT

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

Quicksort

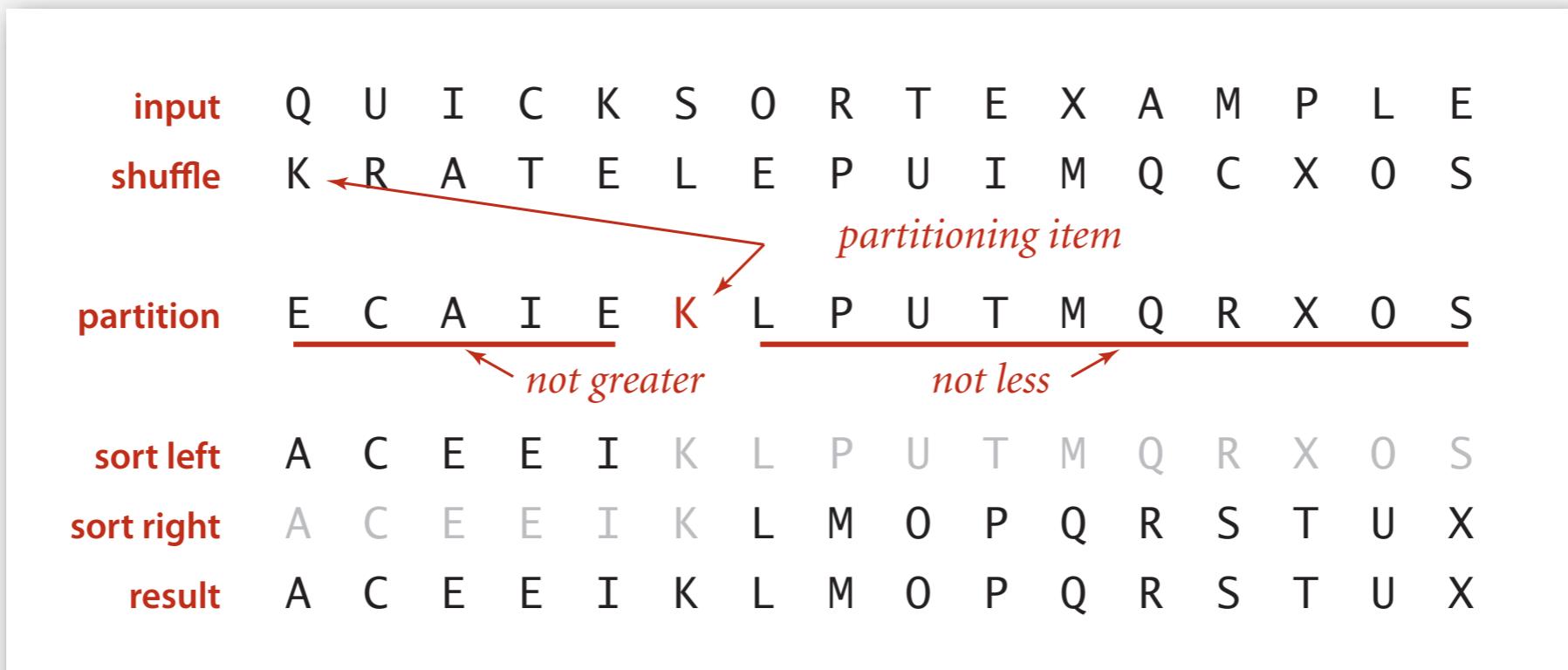
Basic plan.

- **Shuffle** the array.
- **Partition** so that, for some j
 - entry $a[j]$ is in place
 - no larger entry to the left of j
 - no smaller entry to the right of j
- **Sort** each piece recursively.



Sir Charles Antony Richard Hoare

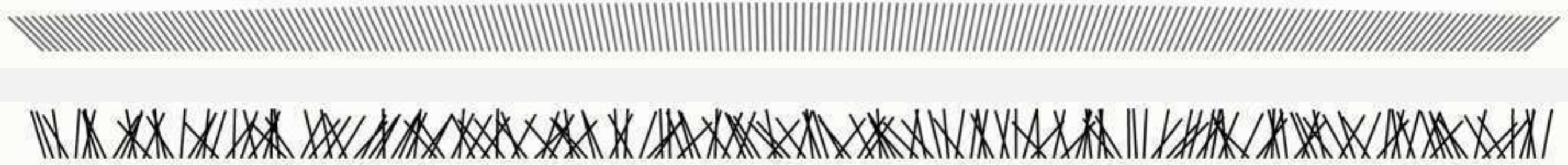
1980 Turing Award



Shuffling

Shuffling

- Shuffling is the process of rearranging an array of elements randomly.
- A good shuffling algorithm is unbiased, where every ordering is equally likely.
- e.g. the Fisher–Yates shuffle (aka. the Knuth shuffle)

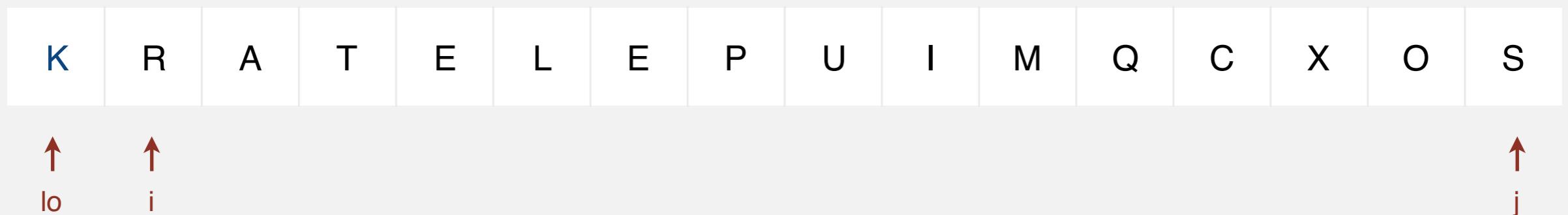


<http://bl.ocks.org/mbostock/39566aca95eb03ddd526>

Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

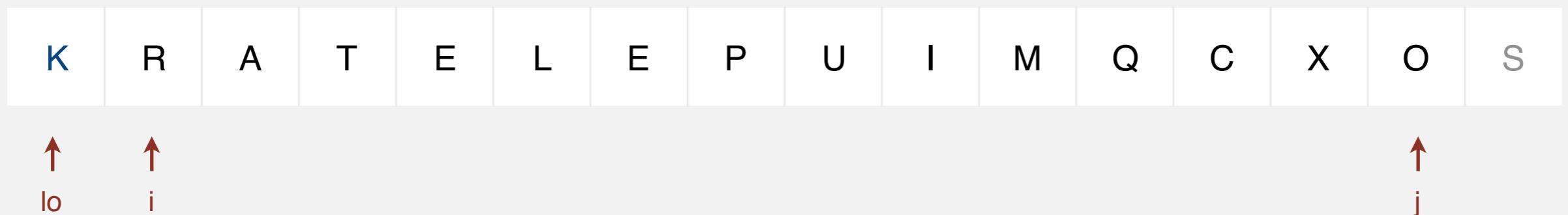


stop i scan because $a[i] \geq a[lo]$

Quicksort partitioning

Repeat until i and j pointers cross.

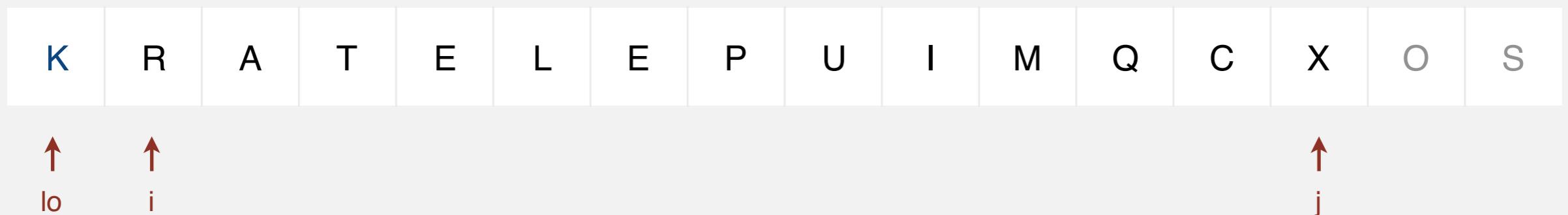
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

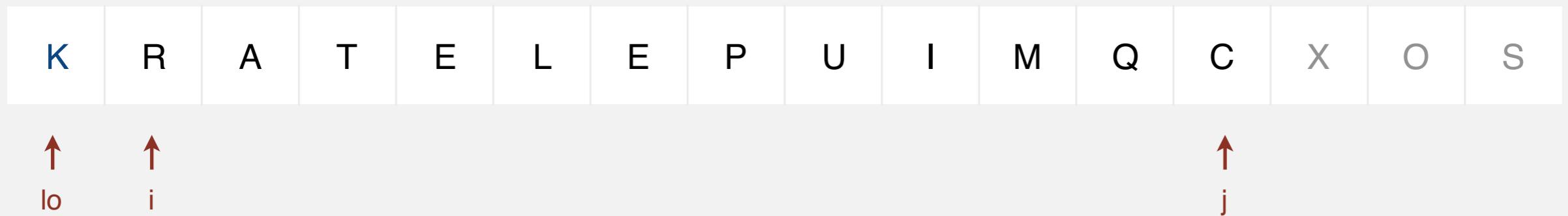
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
 - Scan j from right to left so long as $a[j] > a[lo]$.
 - Exchange $a[i]$ with $a[j]$.

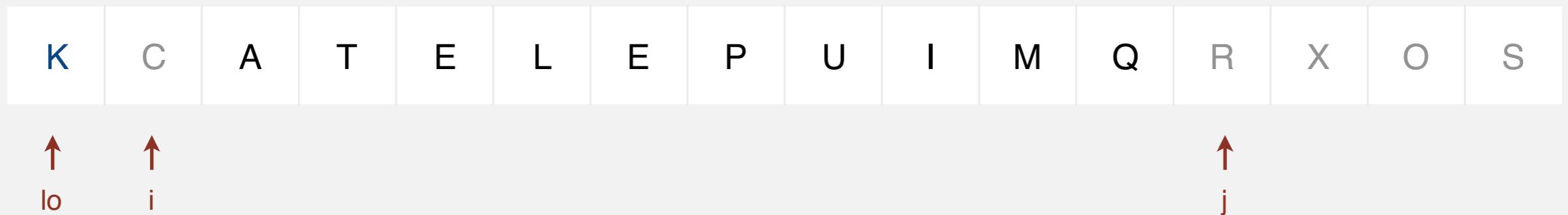


stop j scan and exchange $a[i]$ with $a[j]$

Quicksort partitioning

Repeat until i and j pointers cross.

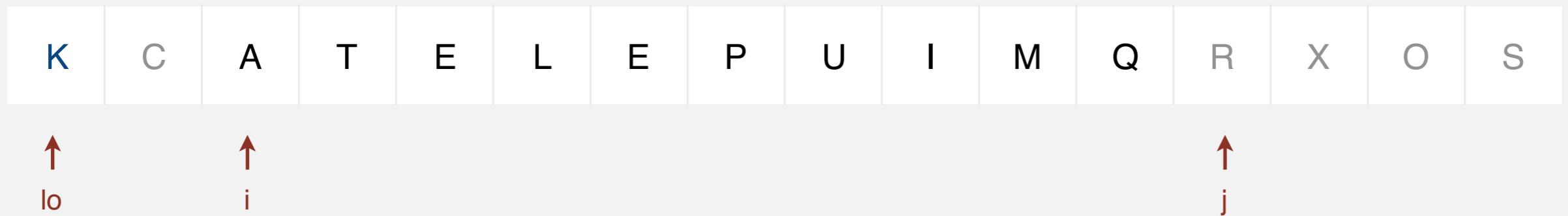
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

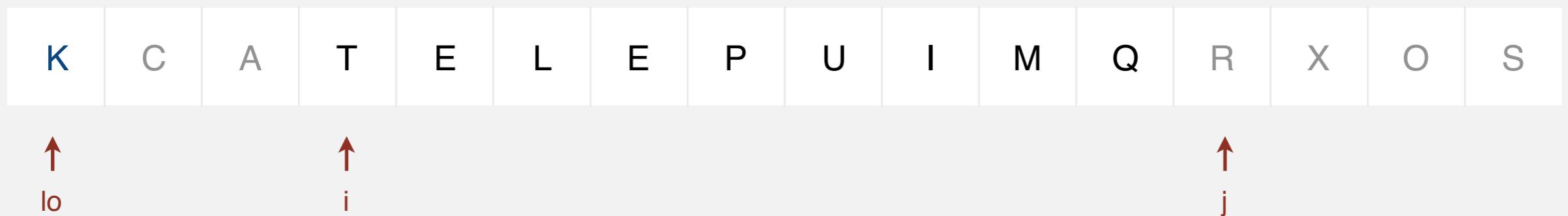
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

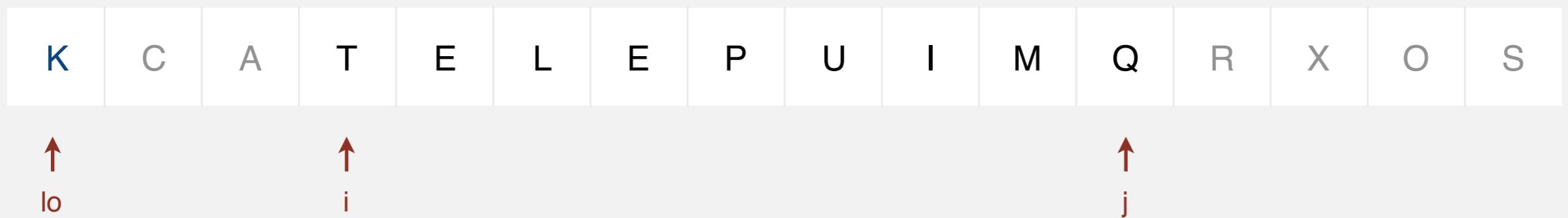


stop i scan because $a[i] \geq a[lo]$

Quicksort partitioning

Repeat until i and j pointers cross.

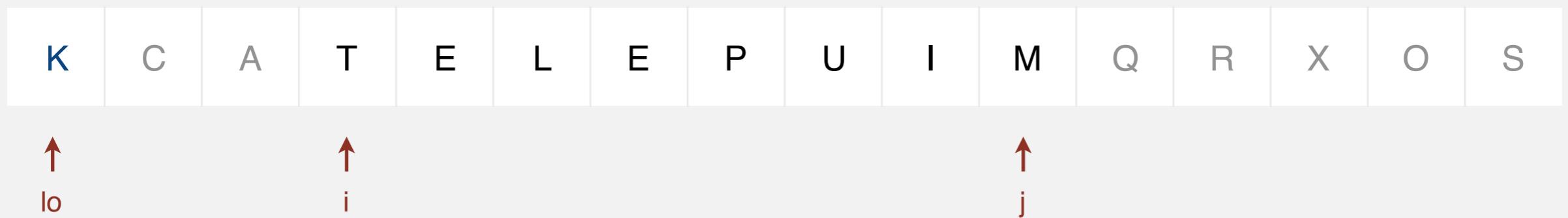
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

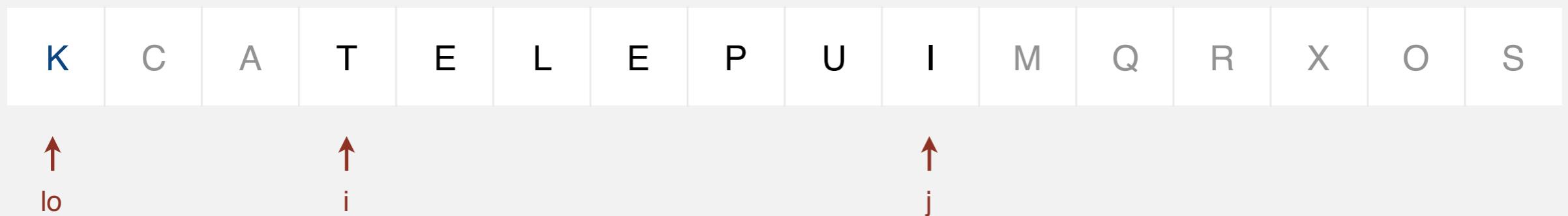
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

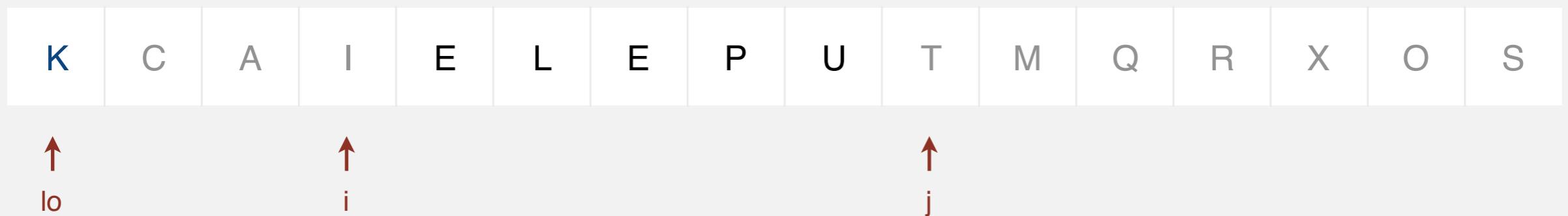


stop j scan and exchange $a[i]$ with $a[j]$

Quicksort partitioning

Repeat until i and j pointers cross.

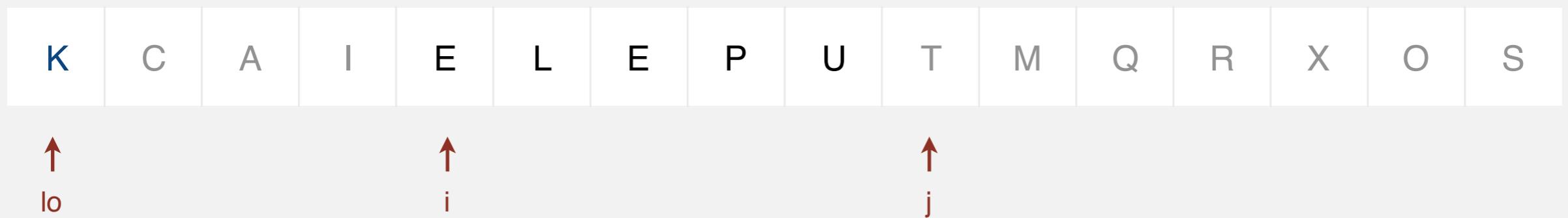
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

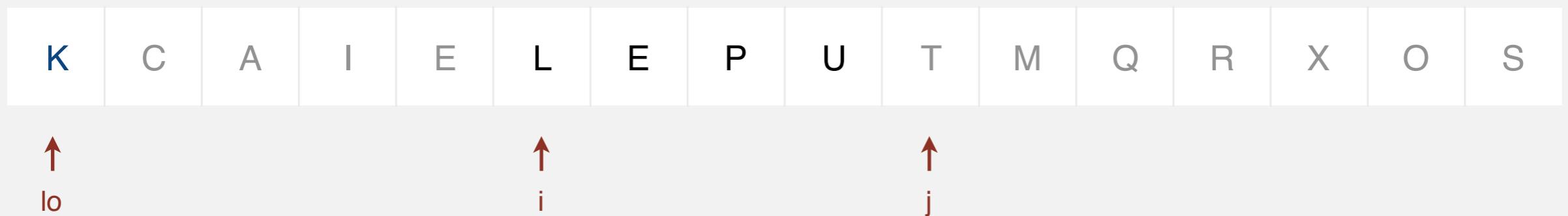
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

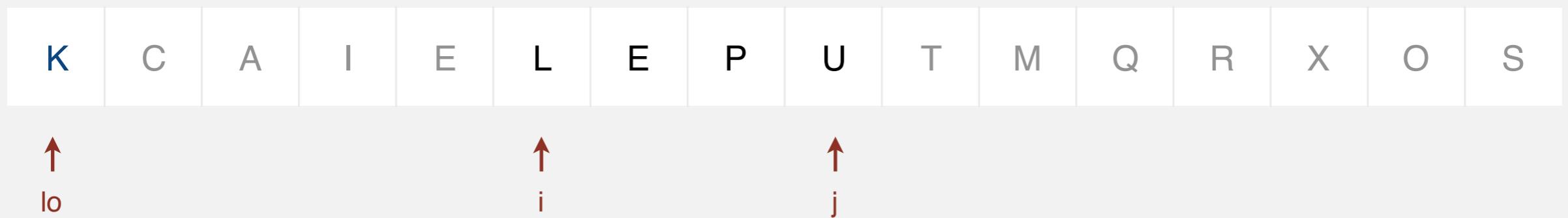


stop i scan because $a[i] \geq a[lo]$

Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

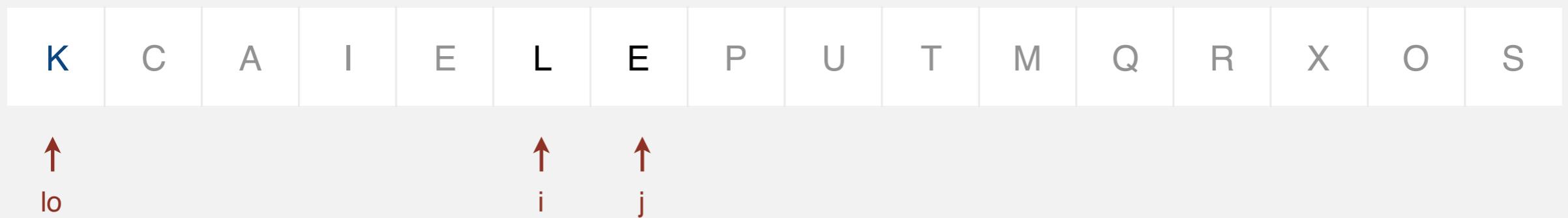
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

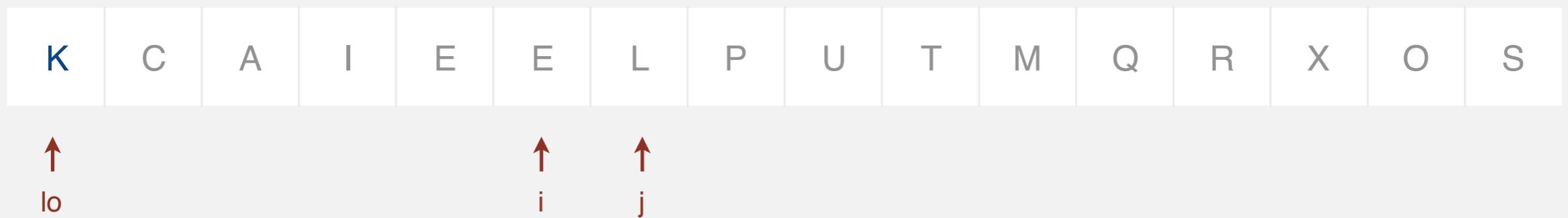


stop j scan and exchange $a[i]$ with $a[j]$

Quicksort partitioning

Repeat until i and j pointers cross.

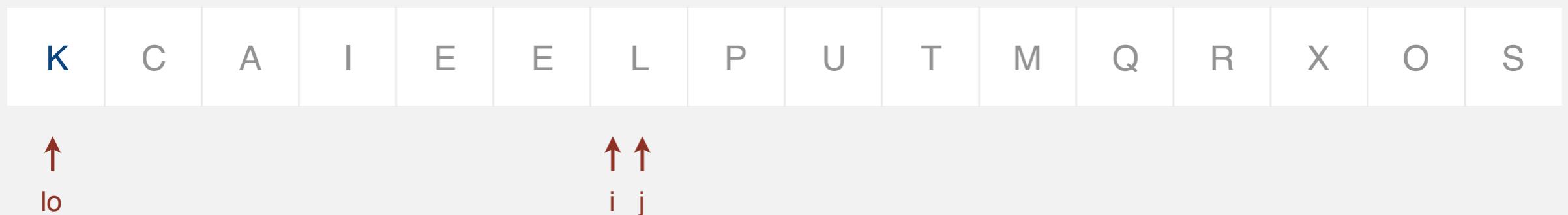
- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

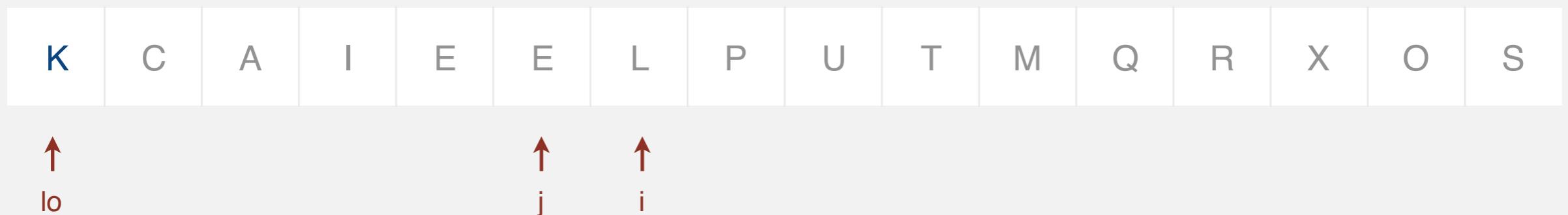


stop i scan because $a[i] \geq a[lo]$

Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.



stop j scan because $a[j] \leq a[lo]$

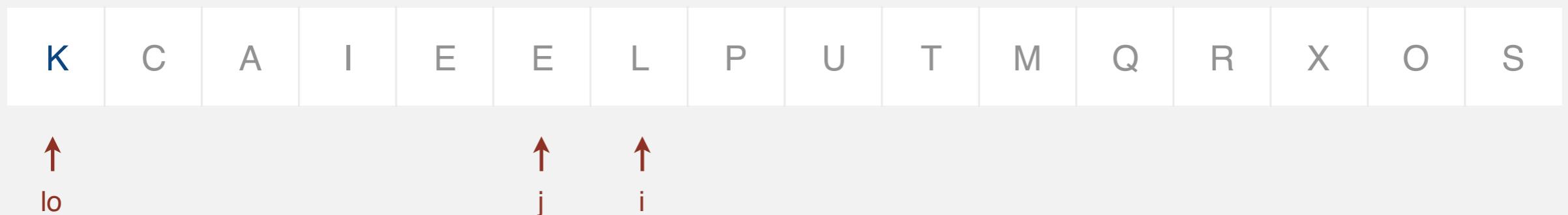
Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

When pointers cross.

- Exchange $a[lo]$ with $a[j]$.



pointers cross: exchange $a[lo]$ with $a[j]$

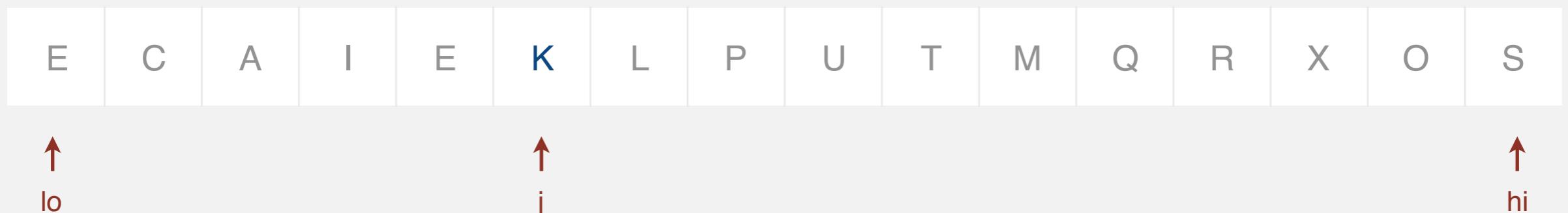
Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

When pointers cross.

- Exchange $a[lo]$ with $a[j]$.



partitioned!

Quicksort partitioning

Basic plan.

- Scan i from left for an item that belongs on the right.
- Scan j from right for an item that belongs on the left.
- Exchange $a[i]$ and $a[j]$.
- Repeat until pointers cross.

	$a[i]$																
	i	j	$a[i]$														
initial values	0	16	$a[i]$														
scan left, scan right	1	12	$a[i]$														
exchange	1	12	$a[i]$														
scan left, scan right	3	9	$a[i]$														
exchange	3	9	$a[i]$														
scan left, scan right	5	6	$a[i]$														
exchange	5	6	$a[i]$														
scan left, scan right	6	5	$a[i]$														
final exchange	6	5	$a[i]$														
result	6	5	$a[i]$														
Partitioning trace (array contents before and after each exchange)																	

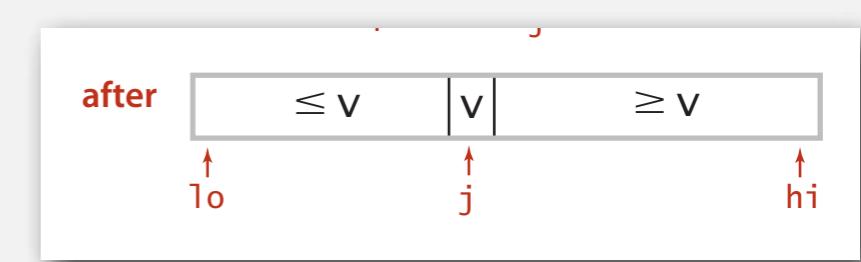
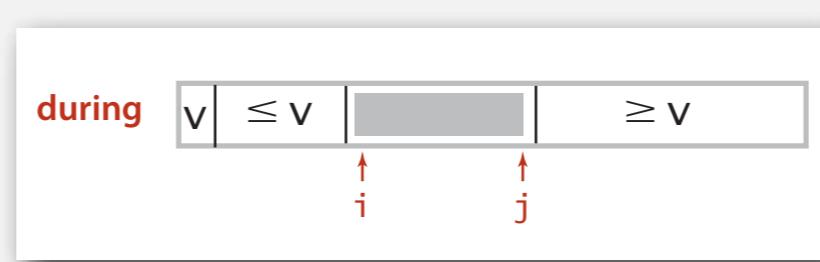
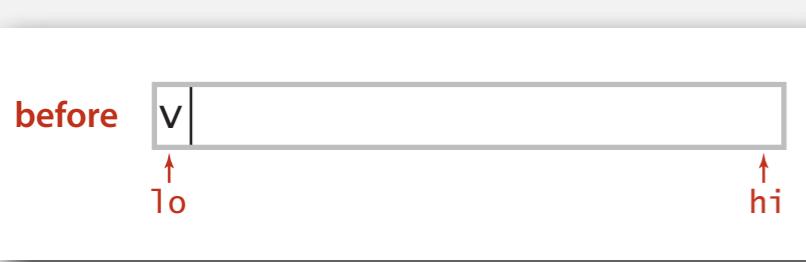
Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi+1;
    while (true)
    {
        while (less(a[++i], a[lo]))           find item on left to swap
            if (i == hi) break;

        while (less(a[lo], a[--j]))           find item on right to swap
            if (j == lo) break;

        if (i >= j) break;                  check if pointers cross
        exch(a, i, j);                   swap
    }

    exch(a, lo, j);                  swap with partitioning item
    return j;                        return index of item now known to be in place
}
```



Quicksort: Java implementation

```
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    { /* see previous slide */ }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
```

shuffle needed for
performance guarantee
(stay tuned)



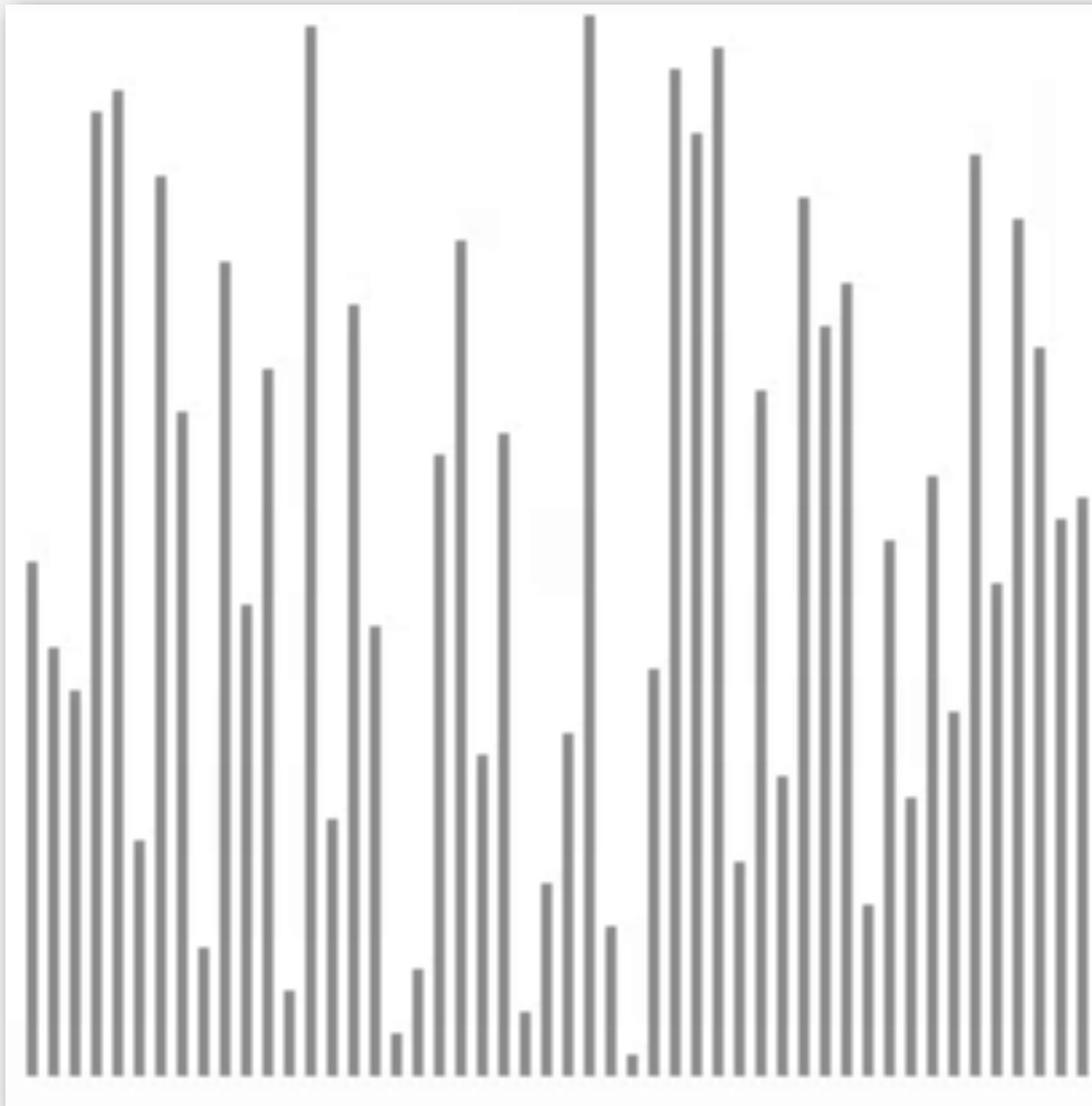
Quicksort trace

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	0	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	0	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	0	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	1	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S	
	4	4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S	
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	0	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S	
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10	10	10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X	
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

Quicksort animation

50 random items



<http://www.sorting-algorithms.com/quick-sort>

- ▲ algorithm position
- in order
- current subarray
- not in order

Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The `(j == lo)` test is redundant (why?), but the `(i == hi)` test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop on keys equal to the partitioning item's key.

Quicksort: empirical analysis

Running time estimates:

- Home PC executes 10^8 compares/second.
- Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort ($N \log N$)			quicksort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

Quicksort: best-case analysis

Best case. Number of compares is $\sim N \lg N$.

Each partitioning process splits the array exactly in half.

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0	0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O	
2	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O	
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4	4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O	
6	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O	
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8	8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O	
10	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O	
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: worst-case analysis

Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

One of the subarrays is empty for every partition.

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \dots + 1 \sim \frac{1}{2} N^2$.
- More likely that your computer is struck by lightning bolt.

Average case. Number of compares is $\sim N \lg N$.

- more compares than mergesort.
- **But** faster than mergesort in practice because of less data movement.

Random shuffle.

- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

Caveat emptor. Many textbook implementations go **quadratic** if array

- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)

Quicksort properties

Proposition. Quicksort is an **in-place** sorting algorithm.

Pf.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

can guarantee logarithmic depth by
recurring on smaller subarray
before larger subarray

Proposition. Quicksort is **not stable**.

Pf.

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.
- Note: could delay insertion sort until one pass at end.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo + CUTOFF - 1)
    {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

Quicksort: practical improvements

Median of sample.

- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.



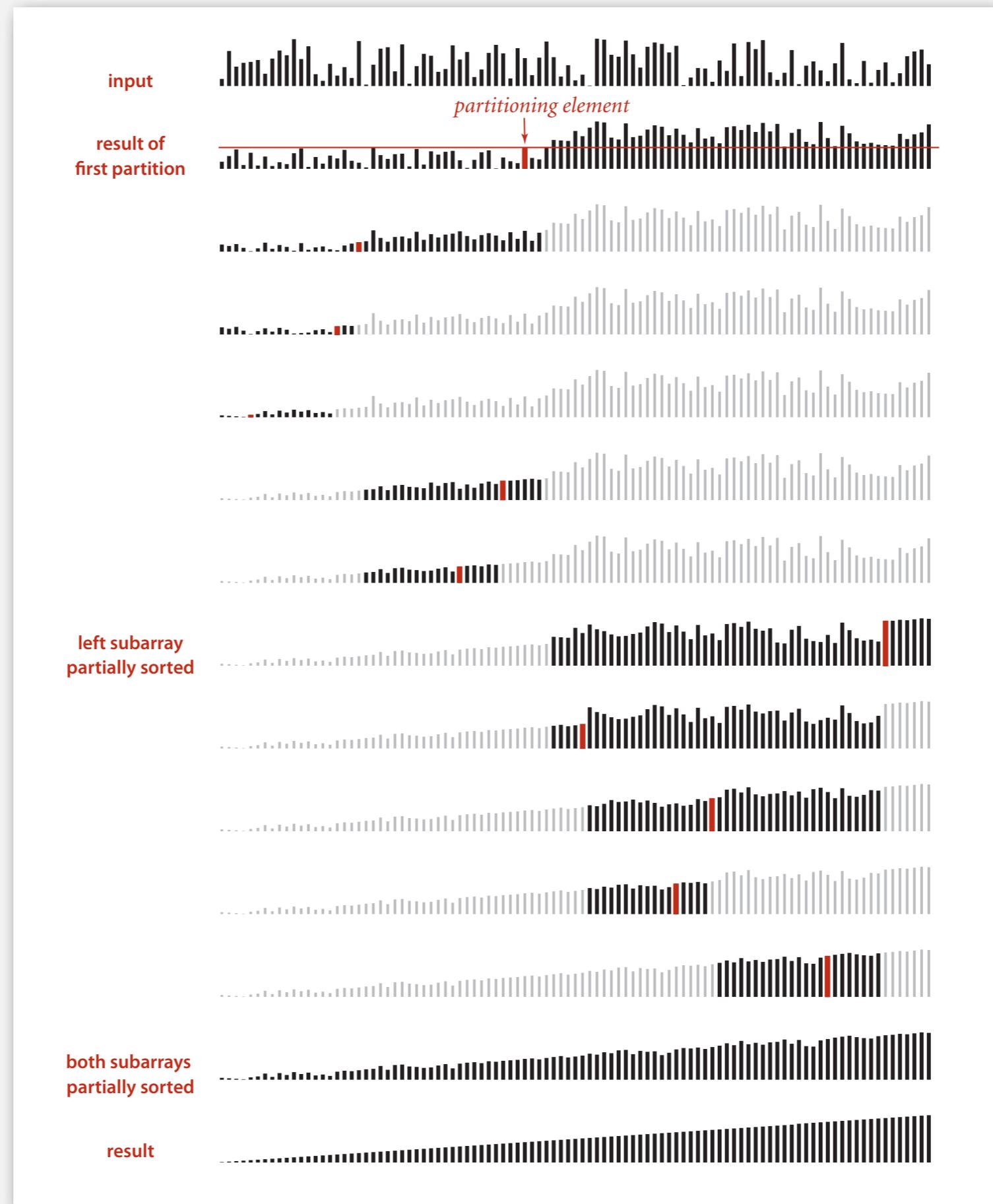
~ $12/7 N \ln N$ compares (slightly fewer)
~ $12/35 N \ln N$ exchanges (slightly more)

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;

    int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, m);

    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```

Quicksort with median-of-3 and cutoff to insertion sort: visualization



Selection

Goal. Given an array of N items, find the k^{th} largest.

Ex. Min ($k = 0$), max ($k = N - 1$), median ($k = N/2$).

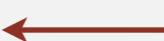
Applications.

- Order statistics.
- Find the "top k ."

Use theory as a guide.

- Easy $N \log N$ upper bound. How?
- Easy N upper bound for $k = 1, 2, 3$. How?
- Easy N lower bound. Why?

Which is true?

- $N \log N$ lower bound?  is selection as hard as sorting?
- N upper bound?  is there a linear-time algorithm for each k ?

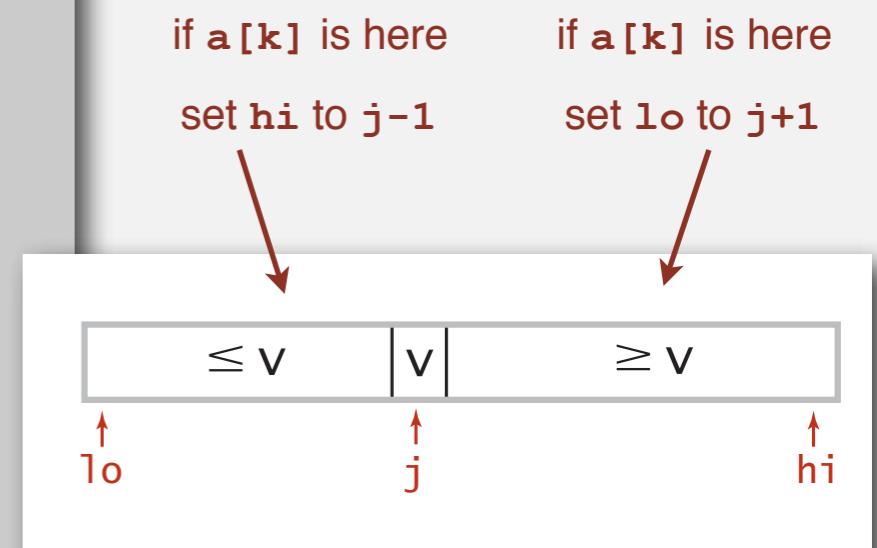
Quick-select

Partition array so that:

- Entry $a[j]$ is in place.
- No larger entry to the left of j .
- No smaller entry to the right of j .

Repeat in **one** subarray, depending on j ; finished when j equals k .

```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
        int j = partition(a, lo, hi);
        if      (j < k) lo = j + 1;
        else if (j > k) hi = j - 1;
        else            return a[k];
    }
    return a[k];
}
```



Quick-select: mathematical analysis

Proposition. Quick-select takes linear time on average.

Pf sketch.

- Intuitively, each partitioning step splits array approximately in half:
 $N + N/2 + N/4 + \dots + 1 \sim 2N$ compares.
- Formal analysis similar to quicksort analysis yields:

$$C_N = 2N + k \ln(N/k) + (N-k) \ln(N/(N-k))$$



(2 + 2 $\ln 2$) N to find the median

Remark. Quick-select uses $\sim \frac{1}{2} N^2$ compares in the worst case, but (as with quicksort) the random shuffle provides a probabilistic guarantee.

Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Find collinear points.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

Chicago	09:25:52
Chicago	09:03:13
Chicago	09:21:05
Chicago	09:19:46
Chicago	09:19:32
Chicago	09:00:00
Chicago	09:35:21
Chicago	09:00:59
Houston	09:01:10
Houston	09:00:13
Phoenix	09:37:44
Phoenix	09:00:03
Phoenix	09:14:25
Seattle	09:10:25
Seattle	09:36:14
Seattle	09:22:43
Seattle	09:10:11
Seattle	09:22:54

↑
key

Duplicate keys

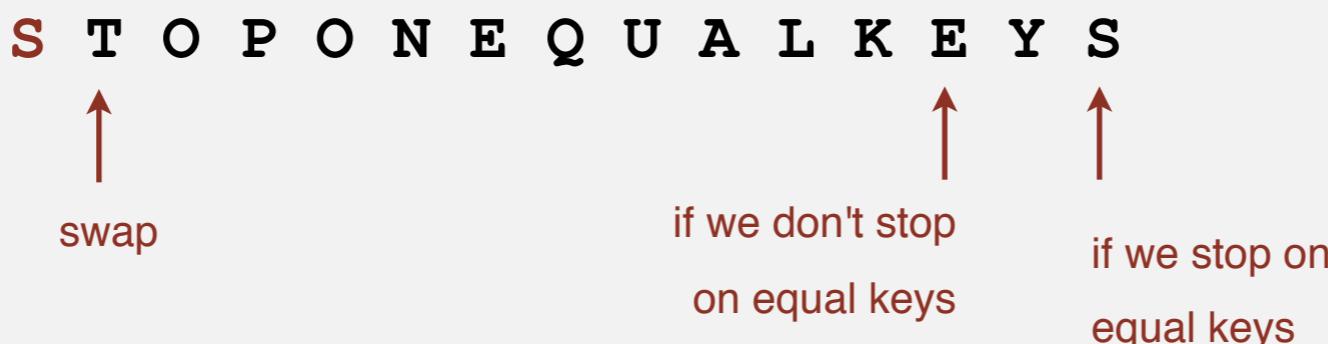
Mergesort with duplicate keys.

Always between $\frac{1}{2} N \lg N$ and $N \lg N$ compares.

Quicksort with duplicate keys.

- Algorithm goes quadratic unless partitioning stops on equal keys!
- 1990s C user found this defect in `qsort()`.

several textbook and system
implementation also have this defect



Duplicate keys: the problem

Mistake. Put all items equal to the partitioning item on one side.

Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

B A A B A B B B C C C

A A A A A A A A A A A

Recommended. Stop scans on items equal to the partitioning item.

Consequence. $\sim N \lg N$ compares when all keys equal.

B A A B A B C C B C B

A A A A A A A A A A A

Desirable. Put all items equal to the partitioning item in place.

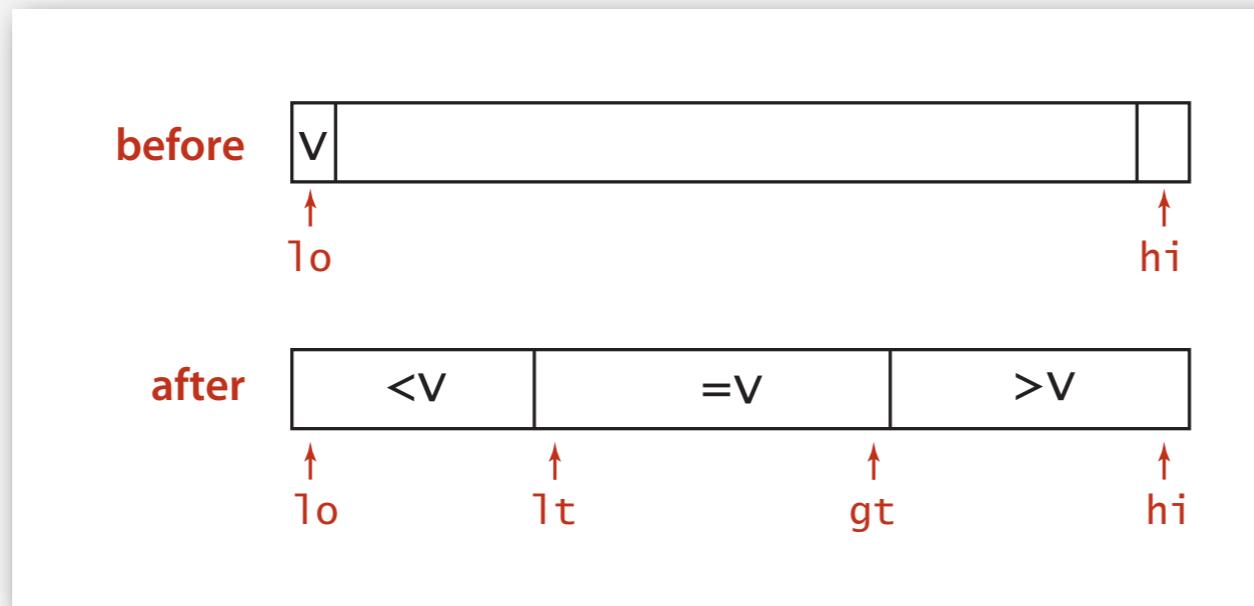
A A A B B B B C C C

A A A A A A A A A A A

3-way partitioning

Goal. Partition array into 3 parts so that:

- Entries between lt and gt equal to partition item v .
- No larger entries to left of lt .
- No smaller entries to right of gt .

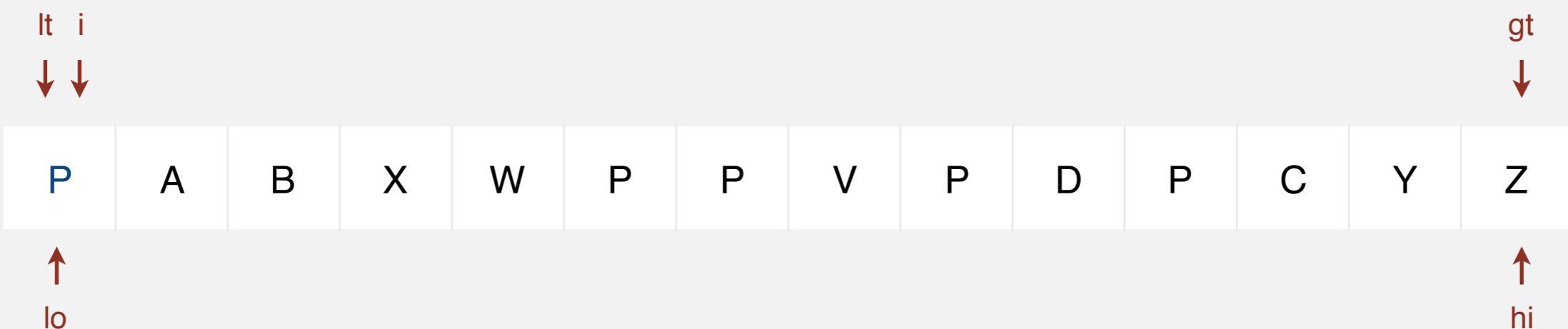


Dutch national flag problem. [Edsger Dijkstra]

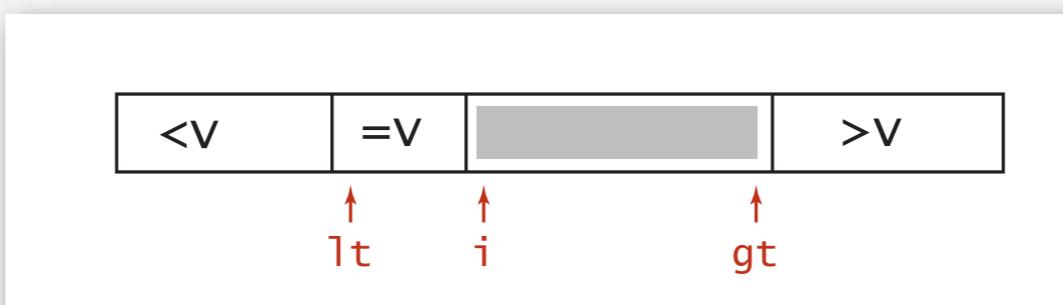
- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.

Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i

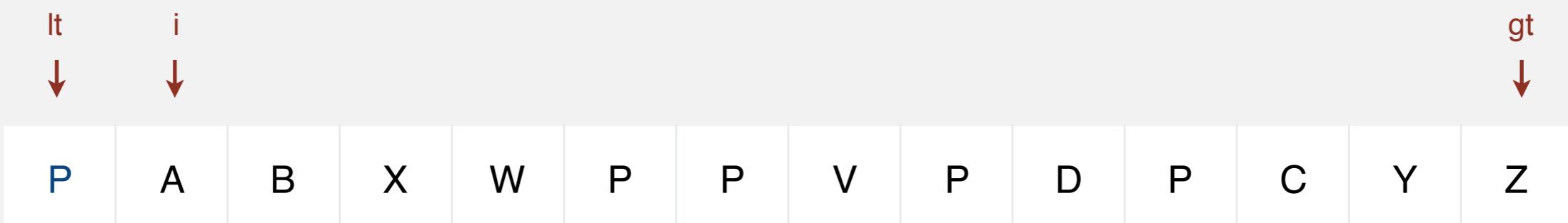


invariant

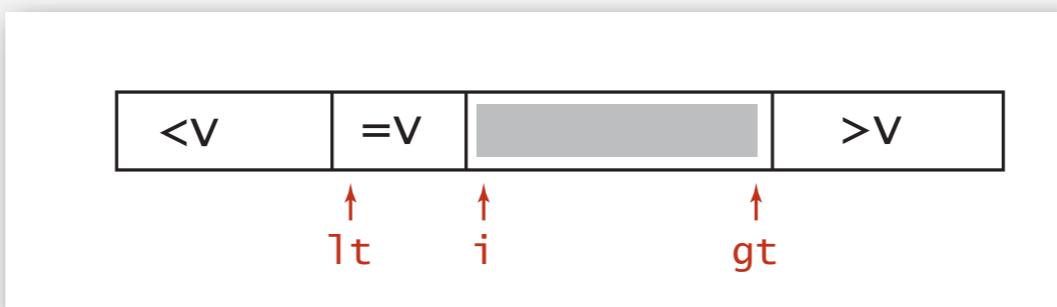


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$ and increment both l_t and i
 - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$ and decrement g_t
 - $(a[i] == v)$: increment i

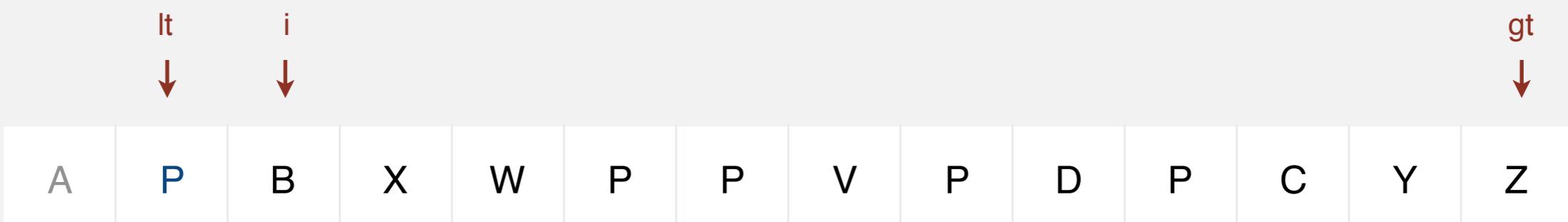


invariant

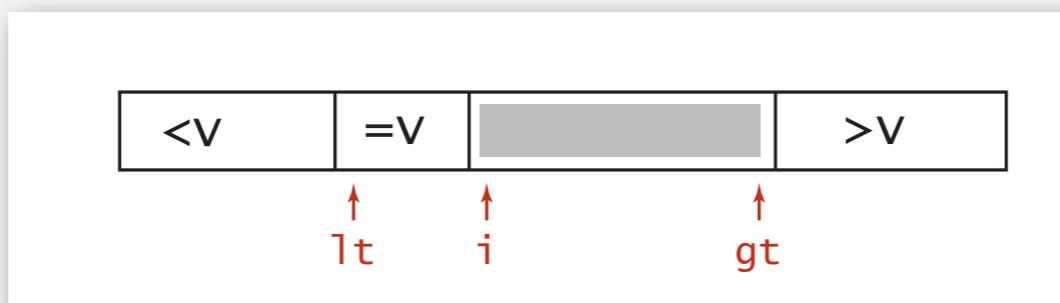


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$ and increment both l_t and i
 - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$ and decrement g_t
 - $(a[i] == v)$: increment i

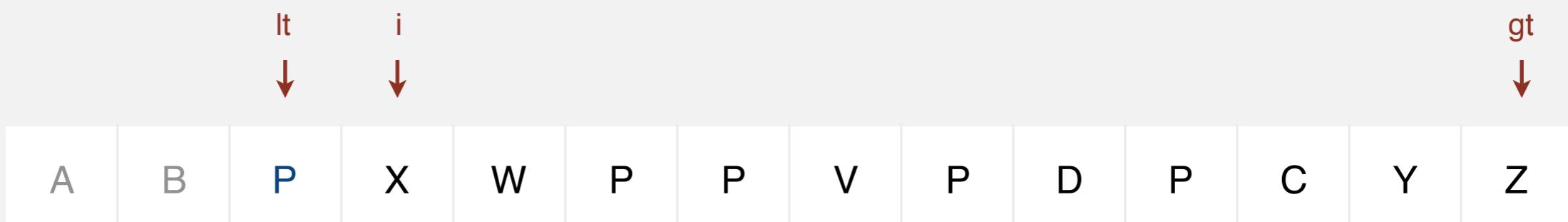


invariant

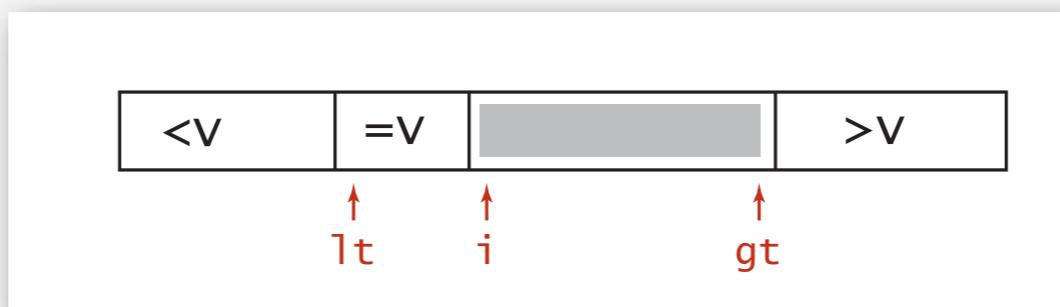


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i

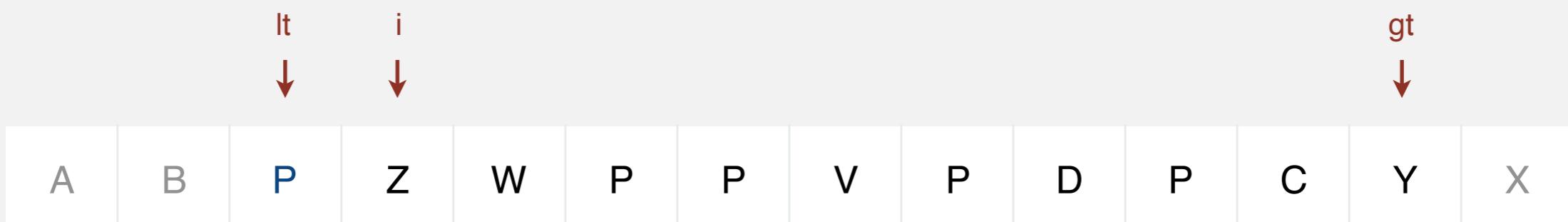


invariant

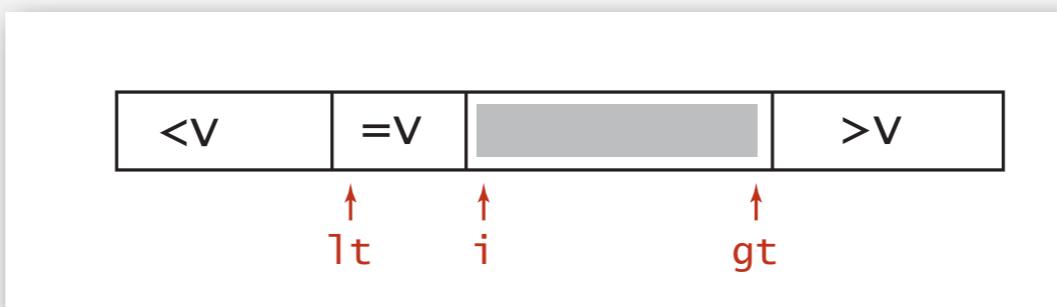


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$ and increment both l_t and i
 - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$ and decrement g_t
 - $(a[i] == v)$: increment i

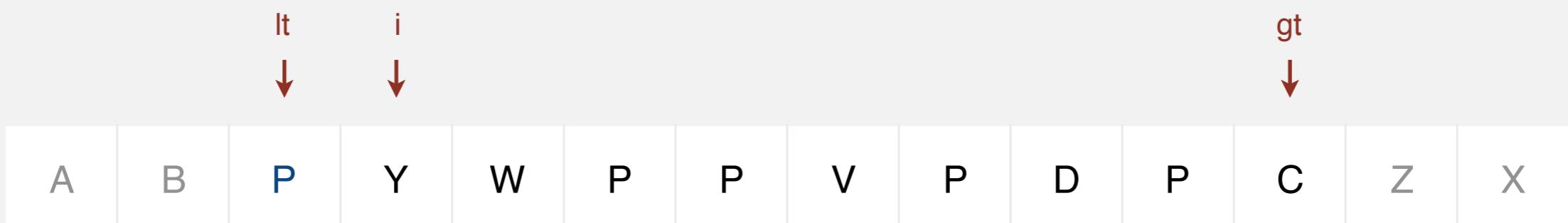


invariant

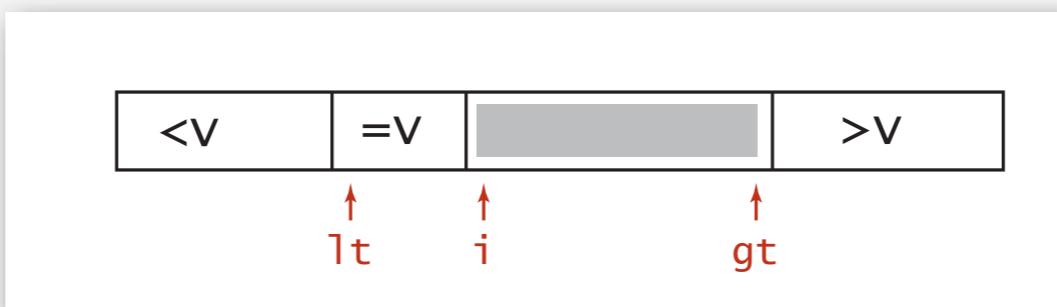


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$ and increment both l_t and i
 - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$ and decrement g_t
 - $(a[i] == v)$: increment i

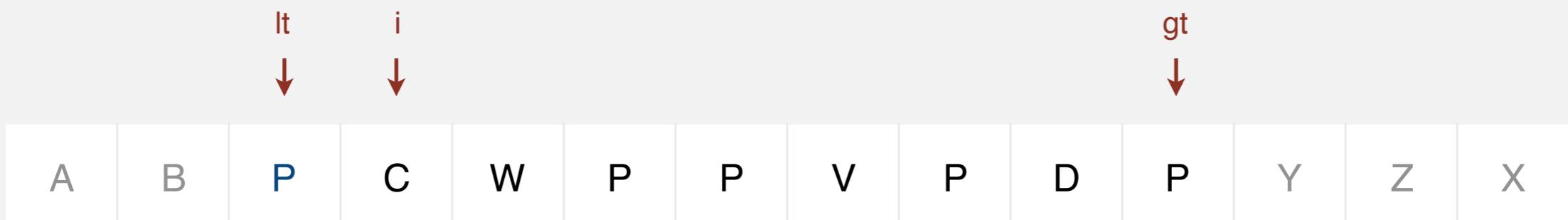


invariant

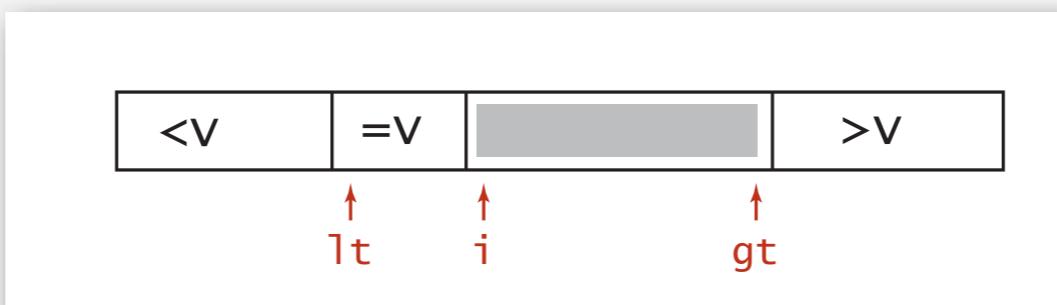


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i

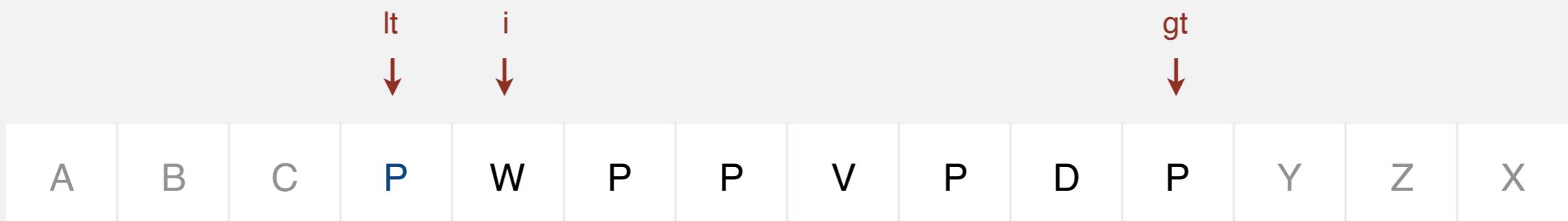


invariant

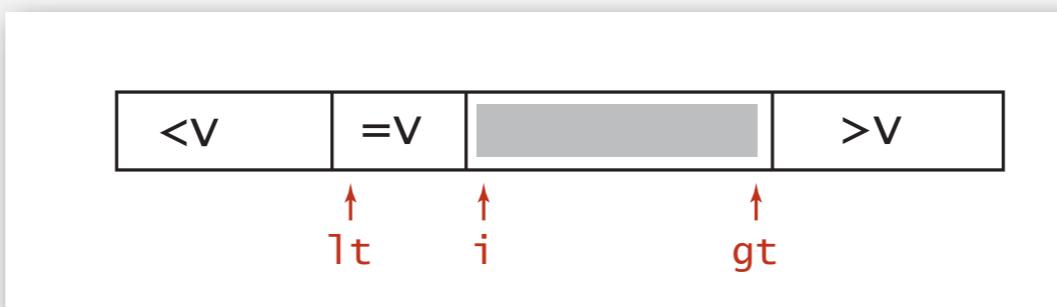


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$ and increment both l_t and i
 - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$ and decrement g_t
 - $(a[i] == v)$: increment i

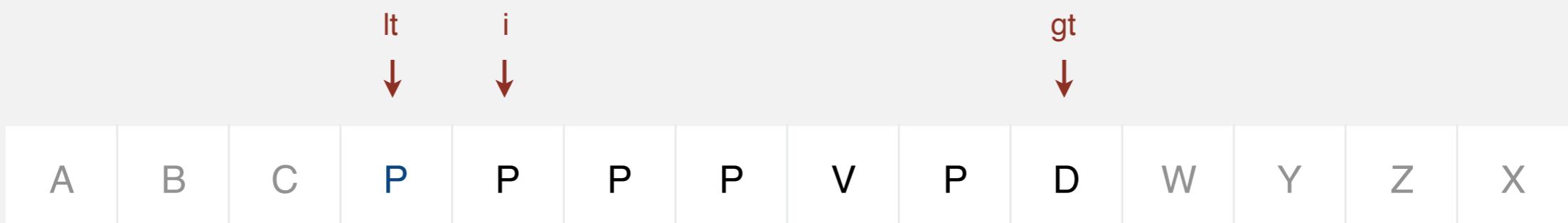


invariant

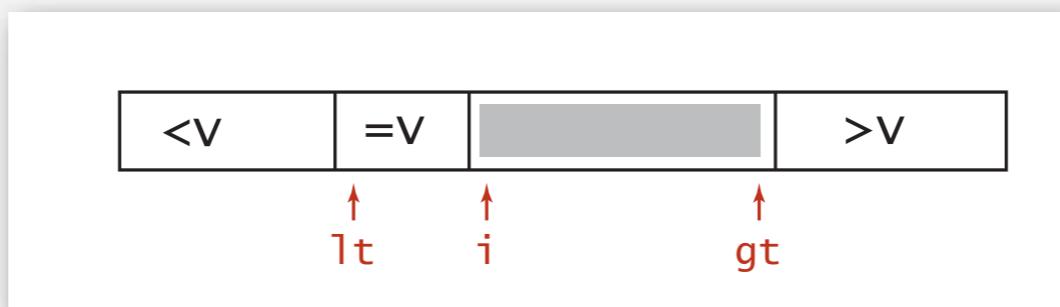


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i

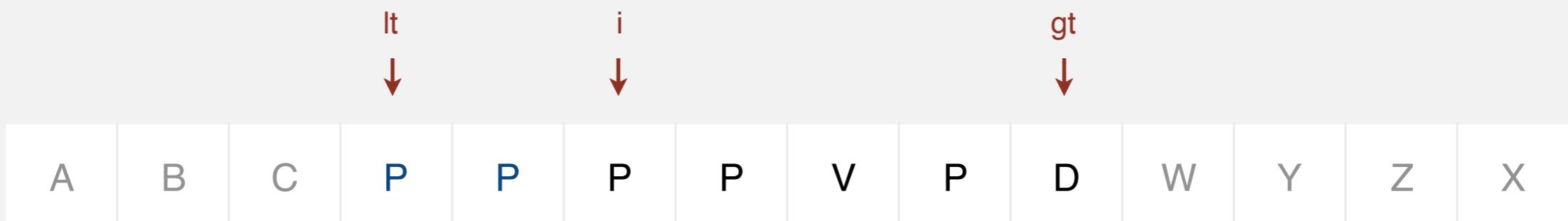


invariant

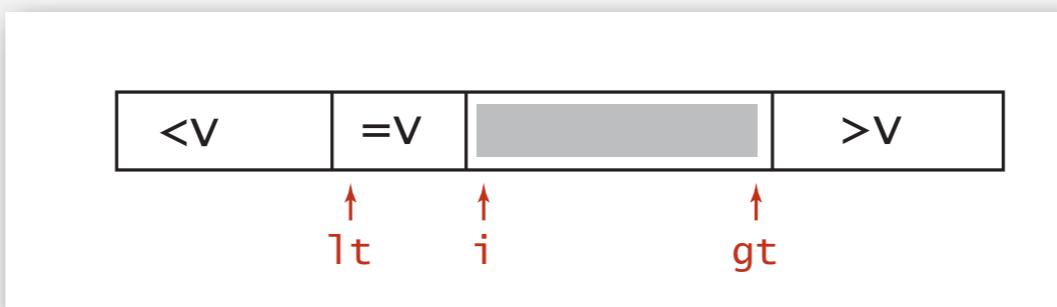


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i

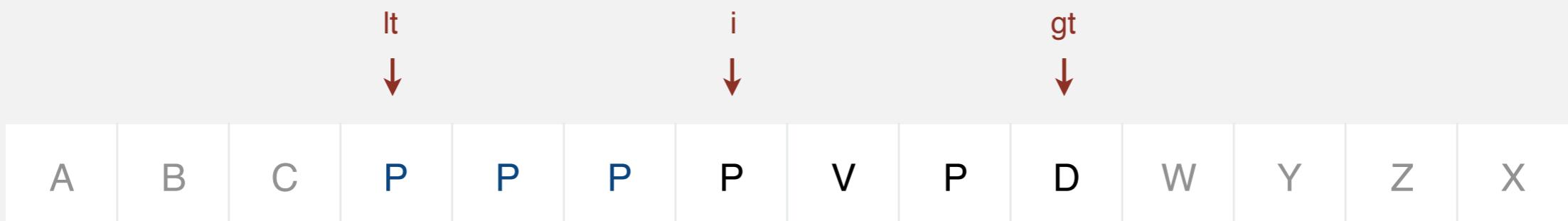


invariant

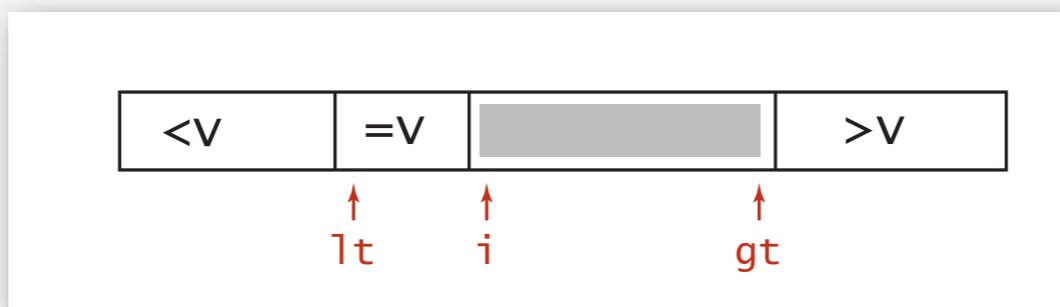


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i



invariant

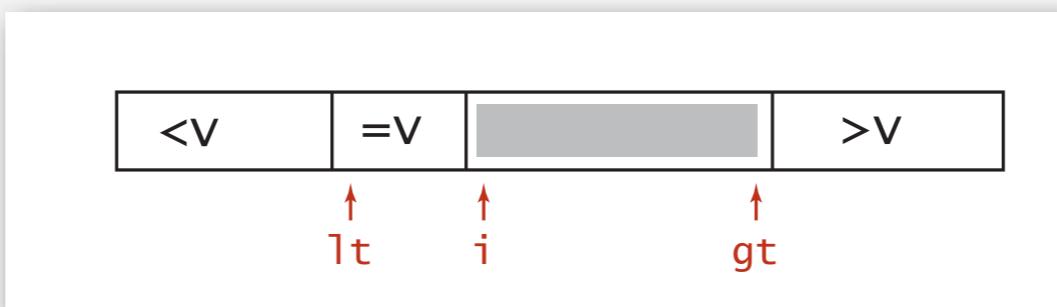


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i



invariant

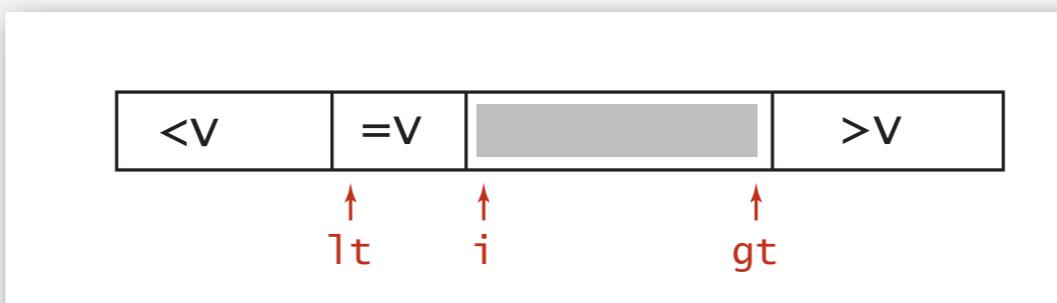


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[l_t]$ with $a[i]$ and increment both l_t and i
 - $(a[i] > v)$: exchange $a[g_t]$ with $a[i]$ and decrement g_t
 - $(a[i] == v)$: increment i

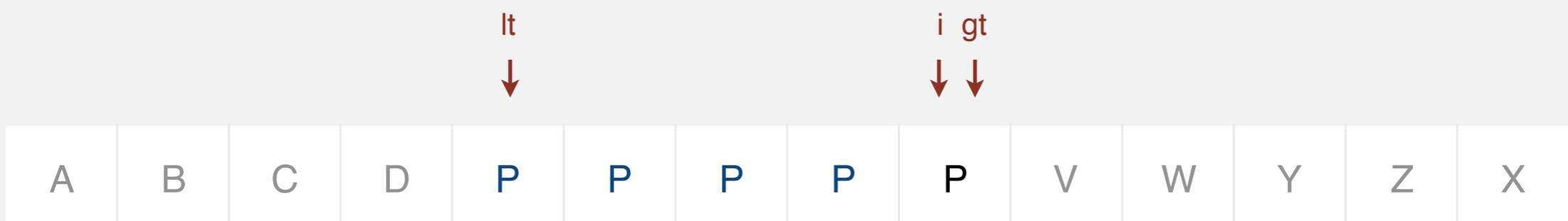


invariant

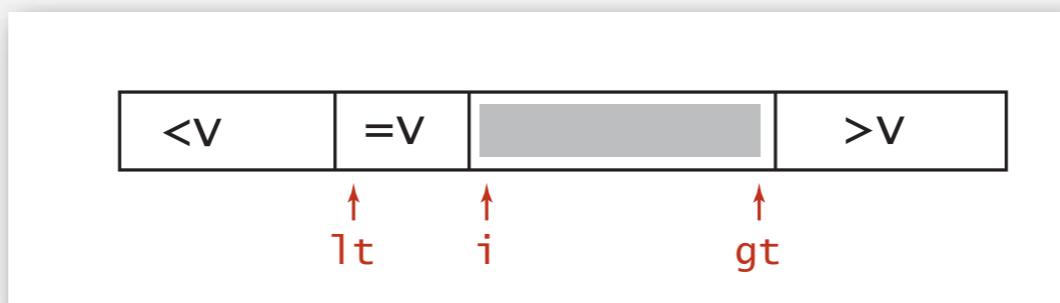


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i

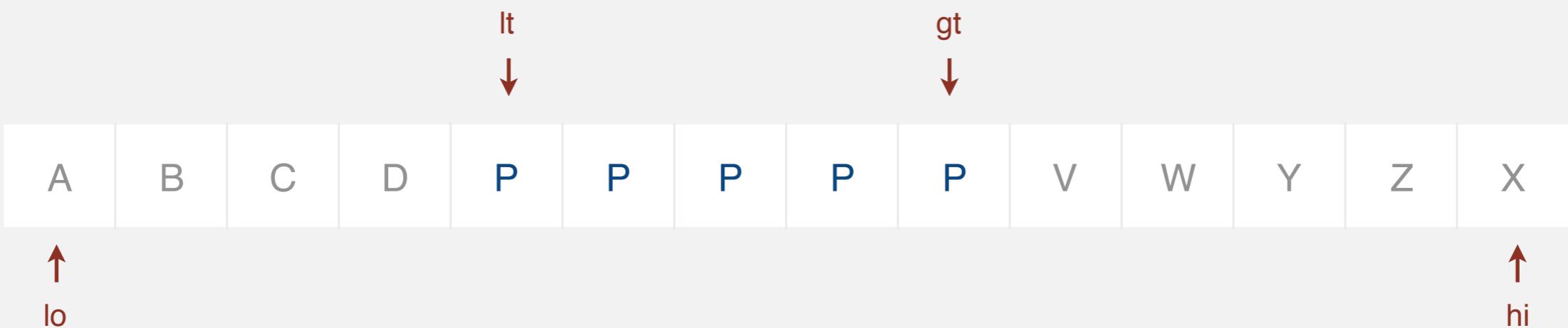


invariant

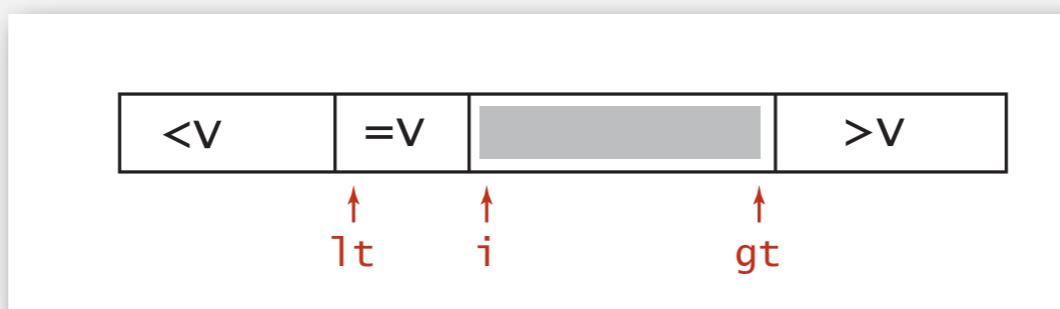


Dijkstra 3-way partitioning

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $(a[i] < v)$: exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $(a[i] > v)$: exchange $a[gt]$ with $a[i]$ and decrement gt
 - $(a[i] == v)$: increment i



invariant



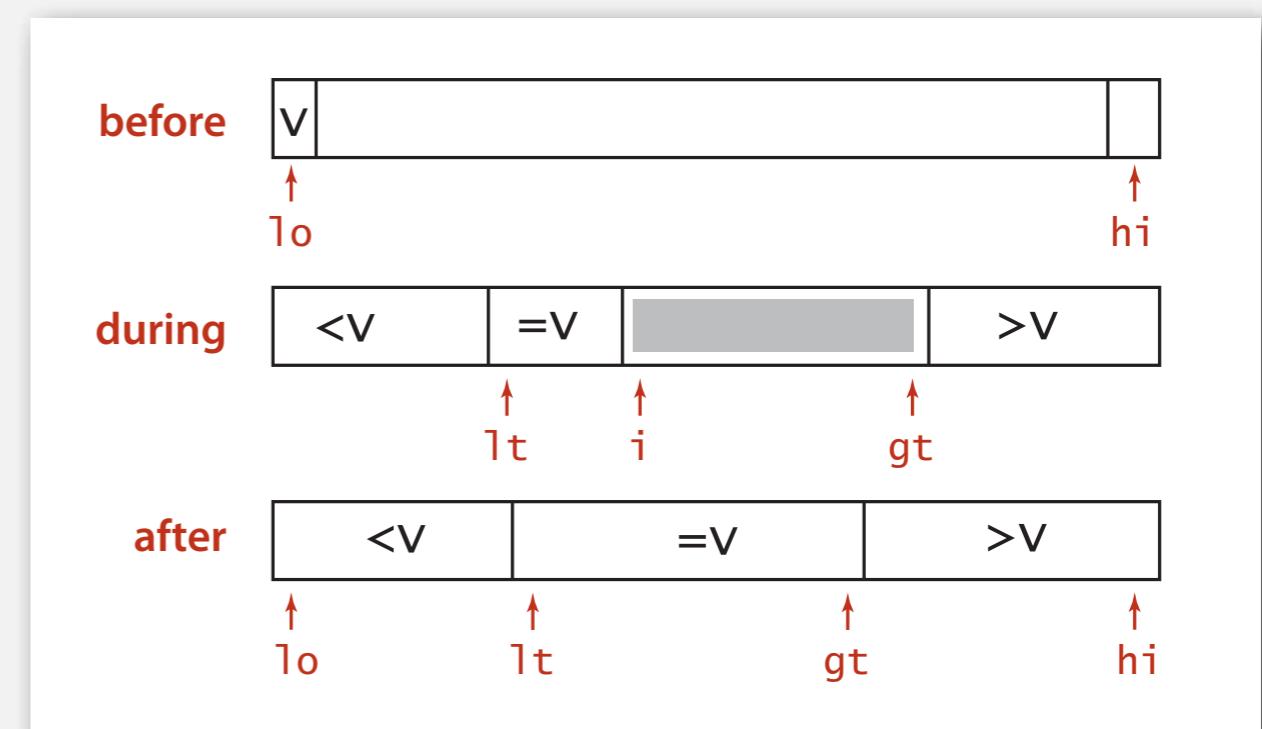
Dijkstra 3-way partitioning algorithm

3-way partitioning.

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $a[i]$ less than v : exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $a[i]$ greater than v : exchange $a[gt]$ with $a[i]$ and decrement gt
 - $a[i]$ equal to v : increment i

Most of the right properties.

- In-place.
- Not much code.
- Linear time if keys are all equal.



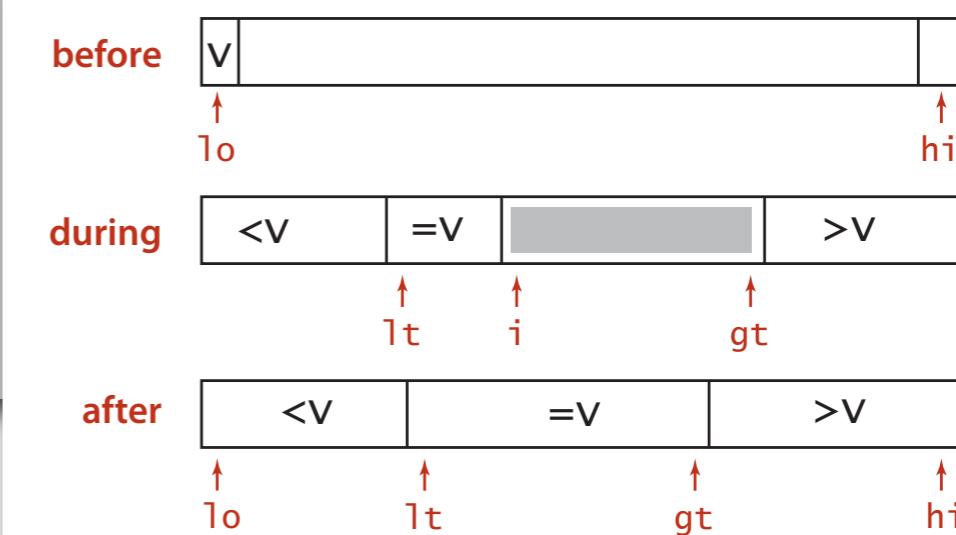
Dijkstra's 3-way partitioning: trace

1t	i	gt	v	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	R	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	R	W
1	3	9	B	R	R	R	B	R	W	B	R	R	W	R	W
2	4	9	B	B	R	R	R								
2	5	9	B	B	R	R	R	R							
2	5	8	B	B	R	R	R	R							
2	5	7	B	B	R	R	R	R							
2	6	7	B	B	R	R	R	R							
3	7	7	B	B	B	R									
3	8	7	B	B	B	R	R	R	R	R	R	R	R	R	R
3	8	7	B	B	B	R									
3-way partitioning trace (array contents after each loop iteration)															

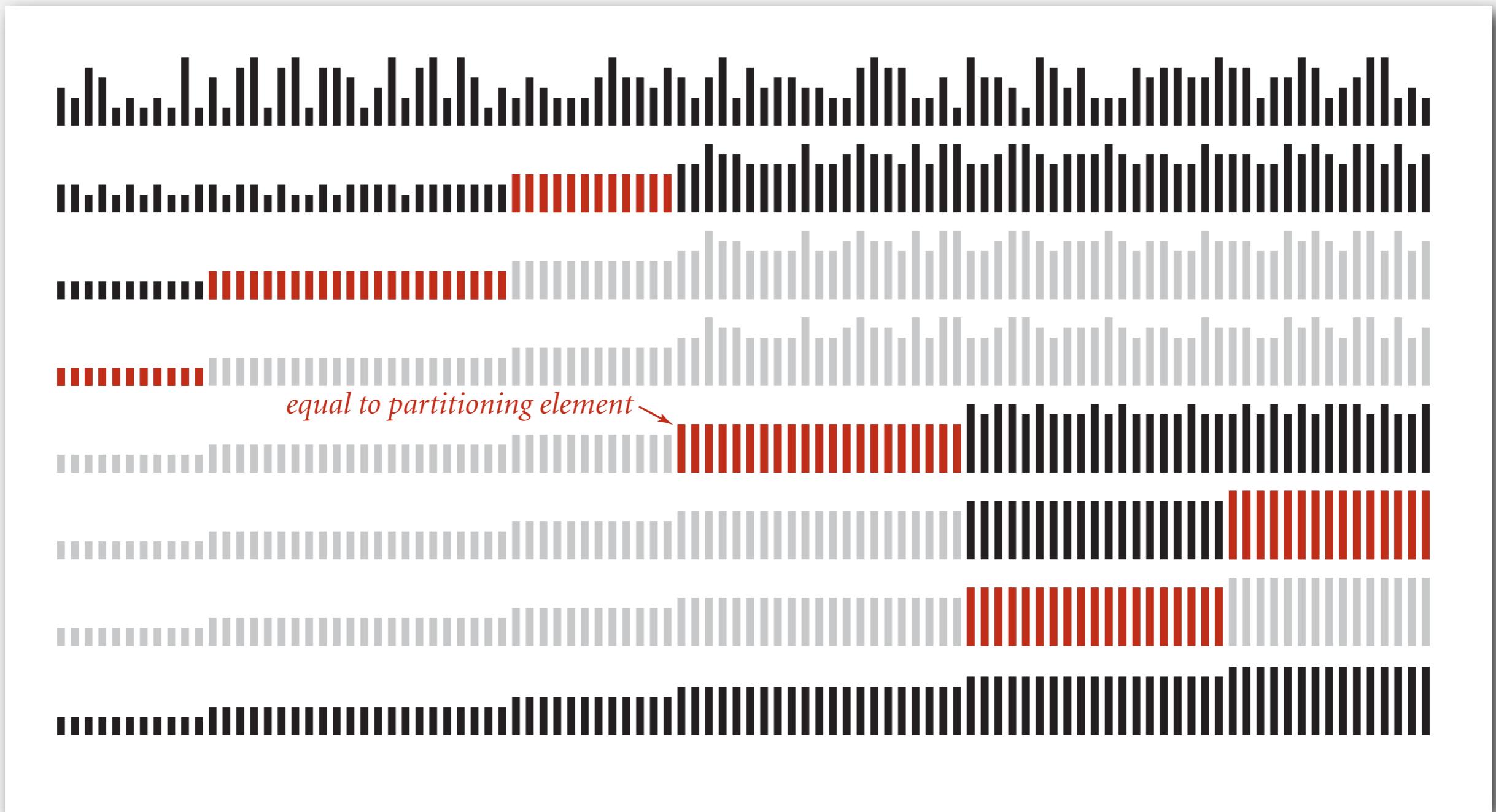
3-way quicksort: Java implementation

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    Comparable v = a[lo];
    int i = lo;
    while (i <= gt)
    {
        int cmp = a[i].compareTo(v);
        if (cmp < 0) exch(a, lt++, i++);
        else if (cmp > 0) exch(a, i, gt--);
        else i++;
    }

    sort(a, lo, lt - 1);
    sort(a, gt + 1, hi);
}
```



3-way quicksort: visual trace



Sorting summary

	inplace?	stable?	worst	average	best	remarks
selection	✓		$N^2/2$	$N^2/2$	$N^2/2$	N exchanges
insertion	✓	✓	$N^2/2$	$N^2/4$	N	use for small N or partially ordered
shell	✓		?	?	N	tight code, subquadratic
merge		✓	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
quick	✓		$N^2/2$	$N \lg N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	✓		$N^2/2$	$N \lg N$	N	improves quicksort in presence of duplicate keys
???	✓	✓	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

PRIORITY QUEUES AND HEAPSORT

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ Heapsort
- ▶ API
- ▶ Elementary implementations
- ▶ Binary heaps
- ▶ Heapsort

Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

Priority queue. Remove the largest (or smallest) item.

<i>operation</i>	<i>argument</i>	<i>return value</i>
<i>insert</i>	P	
<i>insert</i>	Q	
<i>insert</i>	E	
<i>remove max</i>		Q
<i>insert</i>	X	
<i>insert</i>	A	
<i>insert</i>	M	
<i>remove max</i>		X
<i>insert</i>	P	
<i>insert</i>	L	
<i>insert</i>	E	
<i>remove max</i>		P

Priority queue API

Requirement. Generic items are Comparable.

Key must be Comparable (bounded type parameter)	
<code>public class MaxPQ<Key extends Comparable<Key>></code>	
<code>MaxPQ()</code>	<i>create an empty priority queue</i>
<code>MaxPQ(Key[] a)</code>	<i>create a priority queue with given keys</i>
<code>void insert(Key v)</code>	<i>insert a key into the priority queue</i>
<code>Key delMax()</code>	<i>return and remove the largest key</i>
<code>boolean isEmpty()</code>	<i>is the priority queue empty?</i>
<code>Key max()</code>	<i>return the largest key</i>
<code>int size()</code>	<i>number of entries in the priority queue</i>

Priority queue applications

- Event-driven simulation.
[customers in a line, colliding particles]
- Numerical computation.
[reducing roundoff error]
- Data compression.
[Huffman codes]
- Graph searching.
[Dijkstra's algorithm, Prim's algorithm]
- Computational number theory.
[sum of powers]
- Artificial intelligence.
[A* search]
- Statistics.
[maintain largest M values in a sequence]
- Operating systems.
[load balancing, interrupt handling]
- Discrete optimization.
[bin packing, scheduling]
- Spam filtering.
[Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

- Fraud detection: isolate \$\$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N items.

```
% more tinyBatch.txt
Turing      6/17/1990   644.08
vonNeumann  3/26/2002  4121.85
Dijkstra    8/22/2007  2678.40
vonNeumann  1/11/1999  4409.74
Dijkstra    11/18/1995  837.42
Hoare       5/10/1993  3229.27
vonNeumann  2/12/1994  4732.35
Hoare       8/18/1992  4381.21
Turing      1/11/2002   66.10
Thompson    2/27/2000  4747.08
Turing      2/11/1991  2156.86
Hoare       8/12/2003  1025.70
vonNeumann  10/13/1993 2520.97
Dijkstra    9/10/2000  708.95
Turing      10/12/1993  3532.36
Hoare       2/10/2005  4050.20
```

```
% java TopM 5 < tinyBatch.txt
Thompson    2/27/2000  4747.08
vonNeumann  2/12/1994  4732.35
vonNeumann  1/11/1999  4409.74
Hoare       8/18/1992  4381.21
vonNeumann  3/26/2002  4121.85
```

sort key

Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

use a min-oriented pq

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();  
  
while (StdIn.hasNextLine())  
{  
    String line = StdIn.readLine();  
    Transaction item = new Transaction(line);  
    pq.insert(item);  
    if (pq.size() > M)  
        pq.delMin(); ← pq contains  
}                                largest M items
```

Transaction data
type is Comparable
(ordered by \$\$)

order of growth of finding the largest M in a stream of N items

implementation	time	space
sort	$N \log N$	N
elementary PQ	$M N$	M
binary heap	$N \log M$	M
best in theory	N	M

PRIORITY QUEUES AND HEAPSORT

- ▶ Heapsort
- ▶ API
- ▶ Elementary implementations
- ▶ Binary heaps
- ▶ Heapsort

Priority queue: unordered and ordered array implementation

<i>operation</i>	<i>argument</i>	<i>return value</i>	<i>size</i>	<i>contents (unordered)</i>	<i>contents (ordered)</i>
<i>insert</i>	P		1	P	P
<i>insert</i>	Q		2	P Q	P Q
<i>insert</i>	E		3	P Q E	E P Q
<i>remove max</i>		Q	2	P E	E P
<i>insert</i>	X		3	P E X	E P X
<i>insert</i>	A		4	P E X A	A E P X
<i>insert</i>	M		5	P E X A M	A E M P X
<i>remove max</i>		X	4	P E M A	A E M P
<i>insert</i>	P		5	P E M A P	A E M P P
<i>insert</i>	L		6	P E M A P L	A E L M P P
<i>insert</i>	E		7	P E M A P L E	A E E L M P P
<i>remove max</i>		P	6	E M A P L E	A E E L M P P

A sequence of operations on a priority queue

Priority queue: unordered array implementation

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;      // pq[i] = ith element on pq
    private int N;          // number of elements on pq

    public UnorderedMaxPQ(int capacity)
    {   pq = (Key[]) new Comparable[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void insert(Key x)
    {   pq[N++] = x;   }

    public Key delMax()
    {
        int max = 0;
        for (int i = 1; i < N; i++)
            if (less(max, i)) max = i;
        exch(max, N-1);
        return pq[--N];
    }
}
```

no generic
array creation

less() and exch()
similar to sorting methods

null out entry
to prevent loitering

Priority queue elementary implementations

Challenge. Implement **all** operations efficiently.

order-of-growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

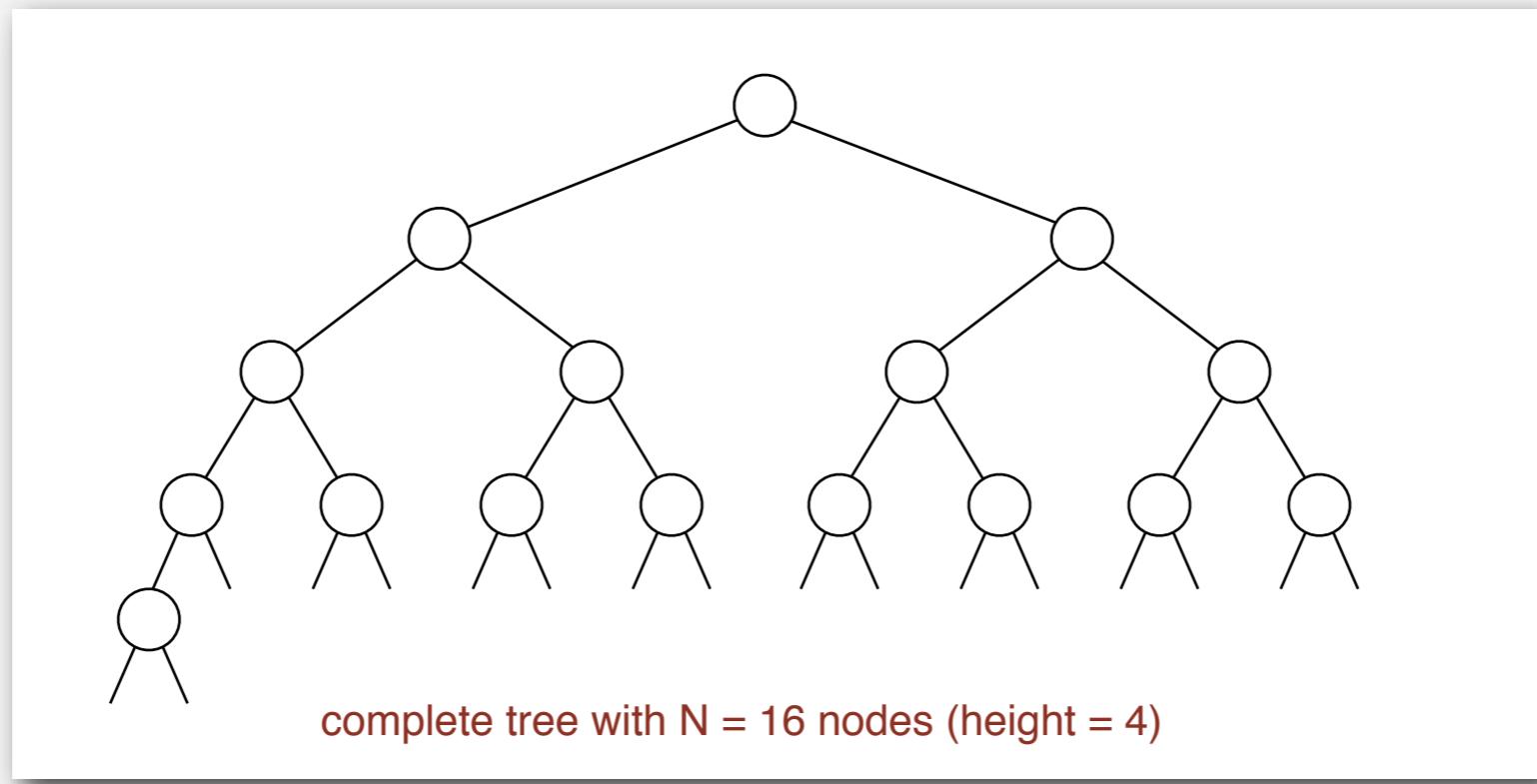
PRIORITY QUEUES AND HEAPSORT

- ▶ Heapsort
- ▶ API
- ▶ Elementary implementations
- ▶ Binary heaps
- ▶ Heapsort

Binary tree

Binary tree. Empty or node with links to left and right binary trees.

Complete tree. Perfectly balanced, except for bottom level.



Property. Height of complete tree with N nodes is $\lfloor \lg N \rfloor$.

Pf. Height only increases when N is a power of 2.

A complete binary tree in nature



Hyphaene Compressa - Doum Palm

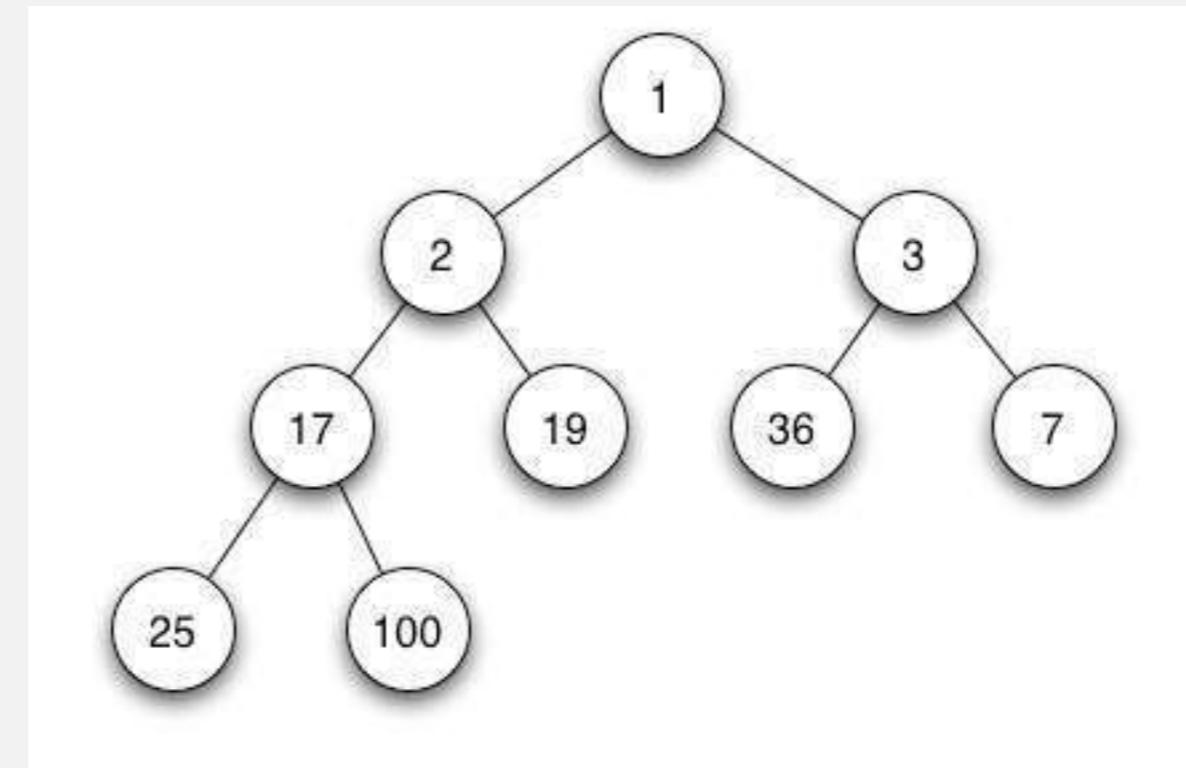
© Shlomit Pinter

Heap

Heap: a heap is a specialised tree-based data structure that satisfies the heap property.

Heap Property:

min-heap property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.



max-heap property: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

Binary heap representations

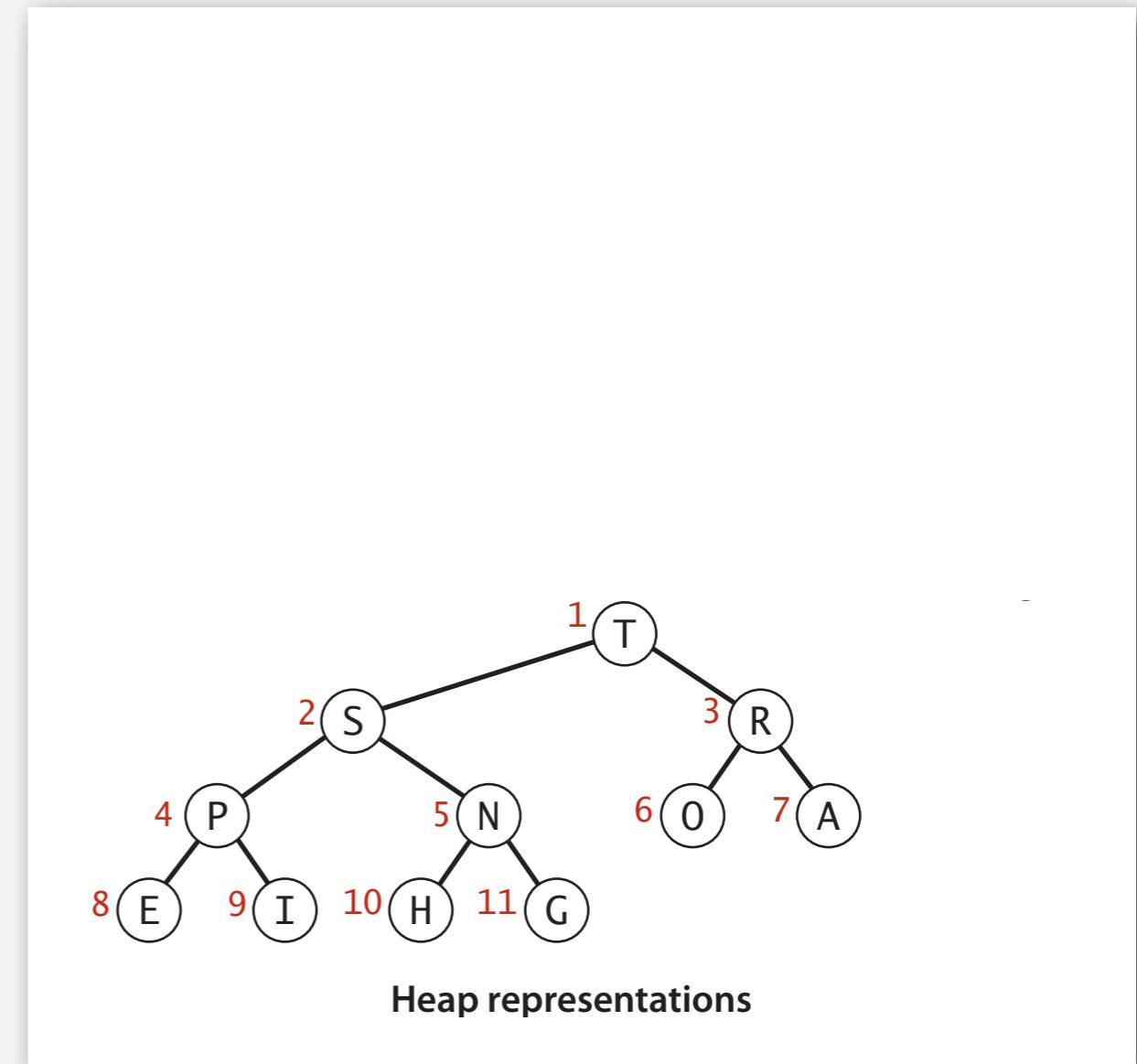
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- Parent's key no smaller than children's keys.

Array representation.

- Indices start at 1.
- Take nodes in **level** order.
- No explicit links needed!

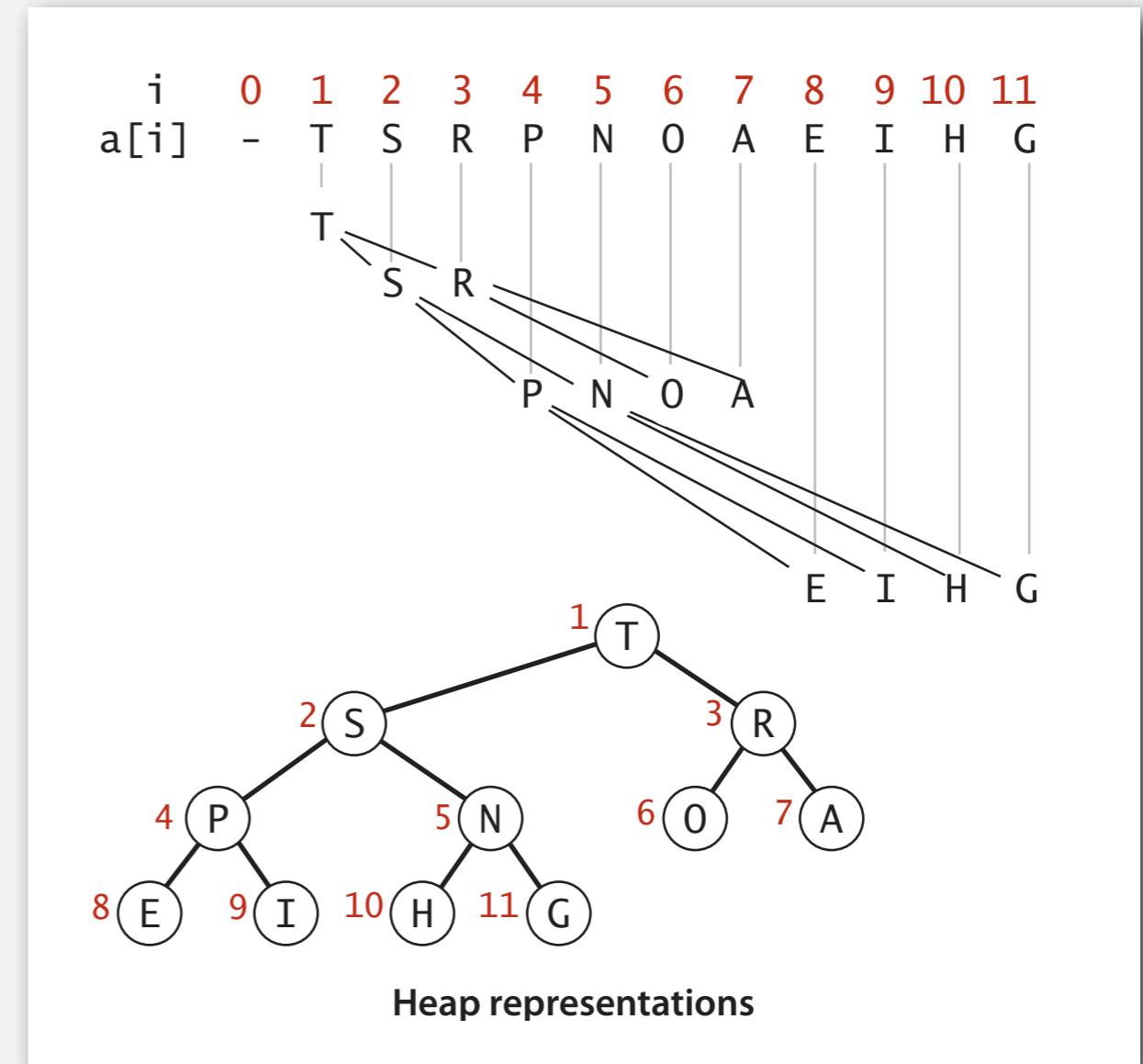


Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of node at k is at $k/2$.
- Children of node at k are at $2k$ and $2k+1$.



Promotion in a heap

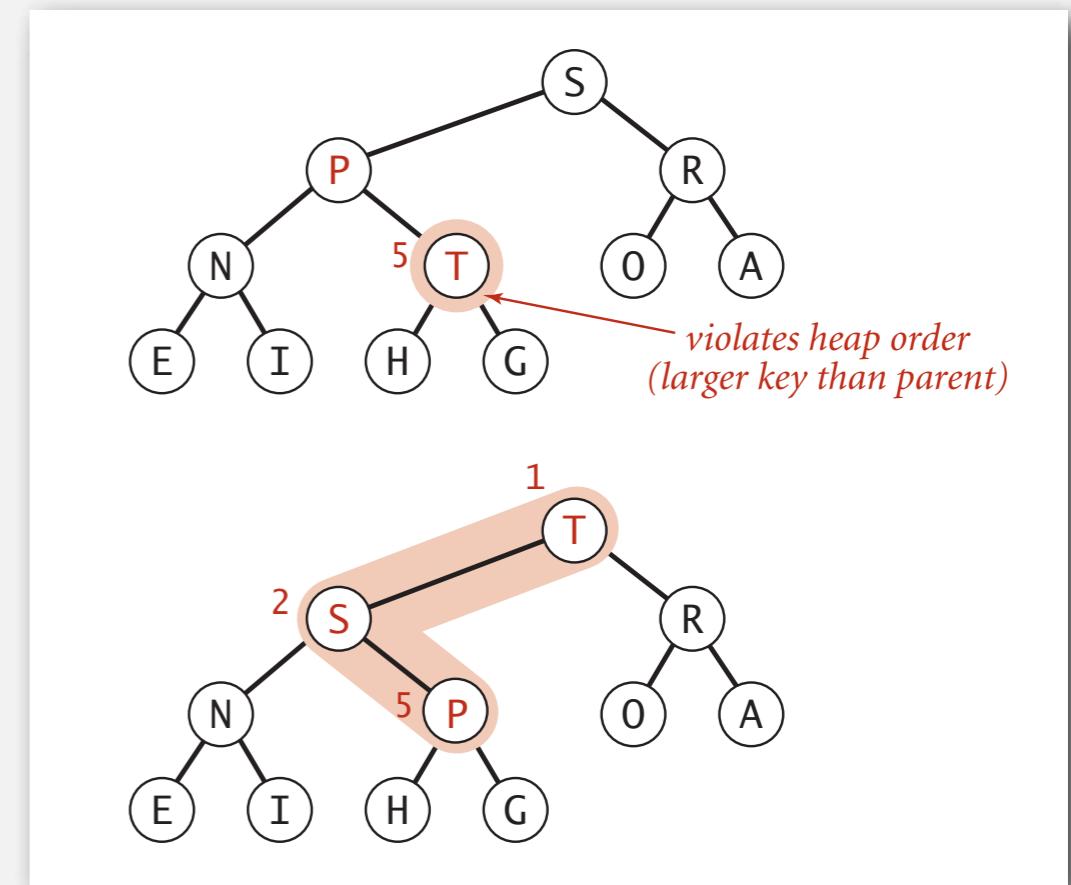
Scenario. Child's key becomes **larger** key than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

parent of node at k is at k/2



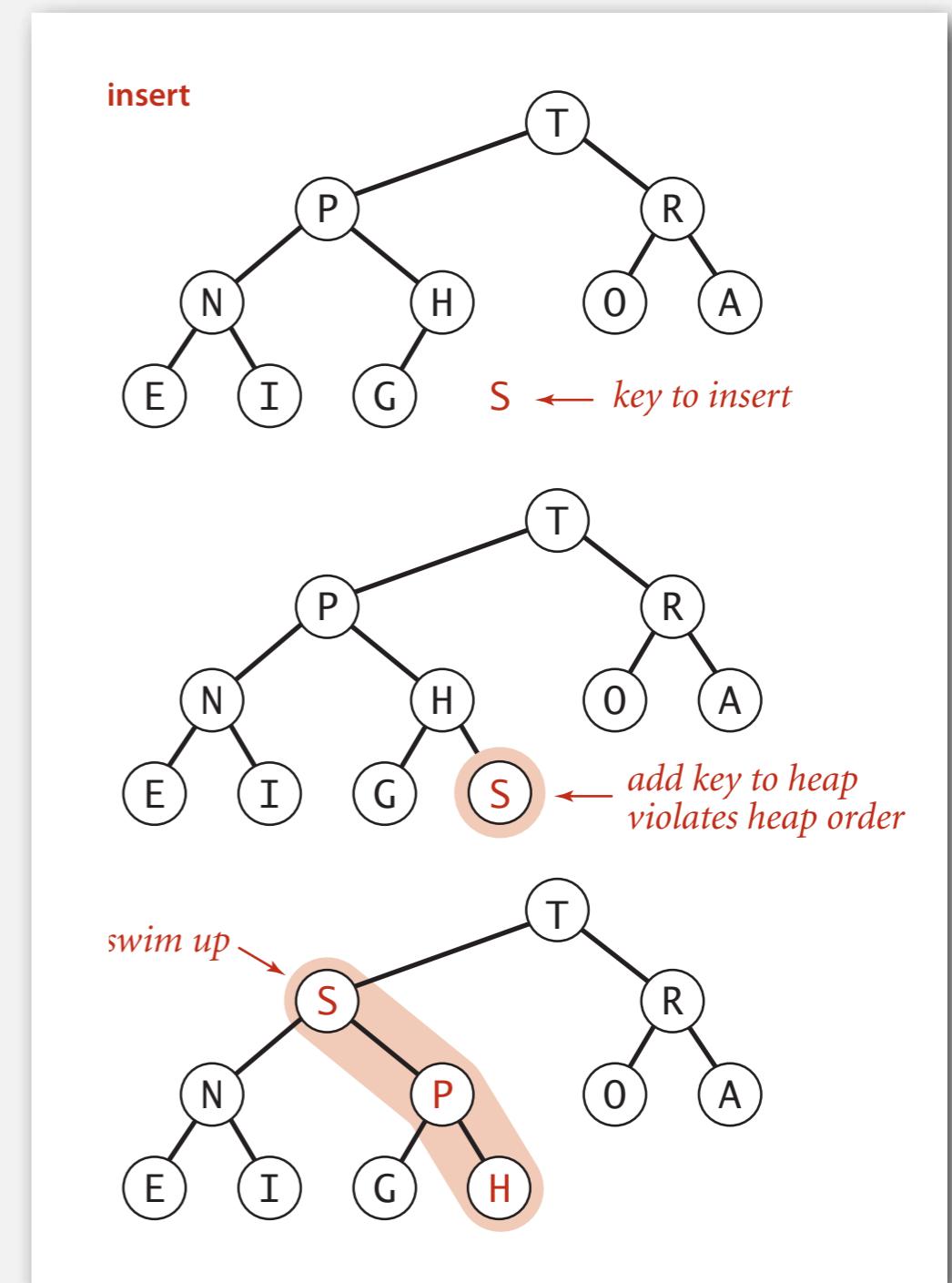
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then swim it up.

Cost. At most $1 + \lg N$ compares.

```
public void insert(Key x)
{
    pq[++N] = x;
    swim(N);
}
```



Demotion in a heap

Scenario. Parent's key becomes **smaller** than one (or both) of its children's keys.

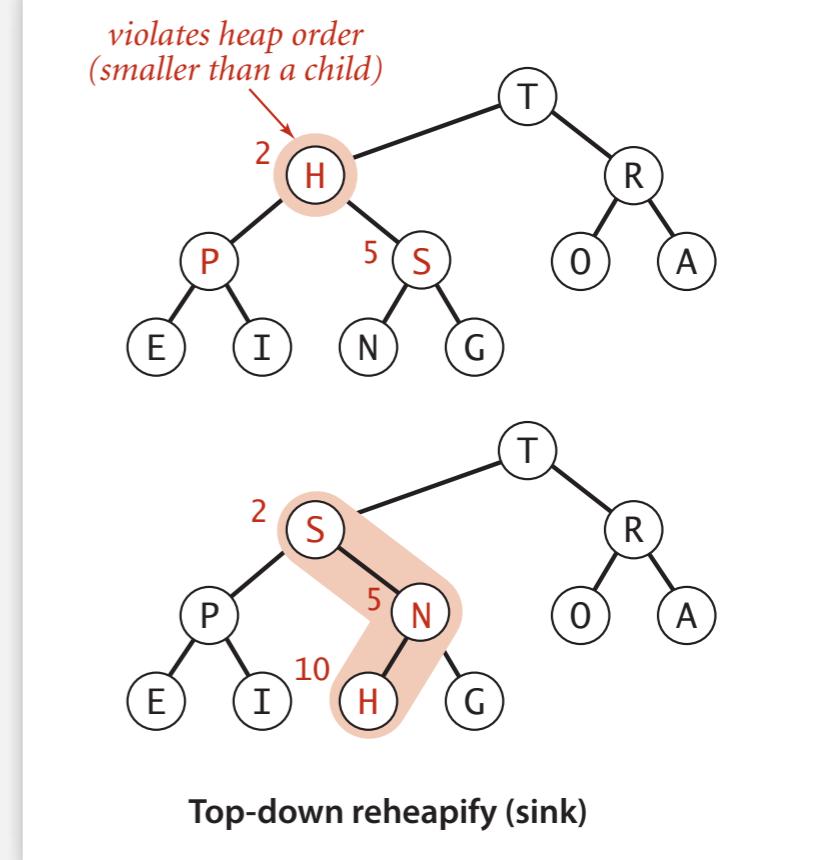
To eliminate the violation:

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

why not smaller child?

```
private void sink(int k)
{
    while (2*k <= N)
    {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

children of node
at k are $2k$ and $2k+1$



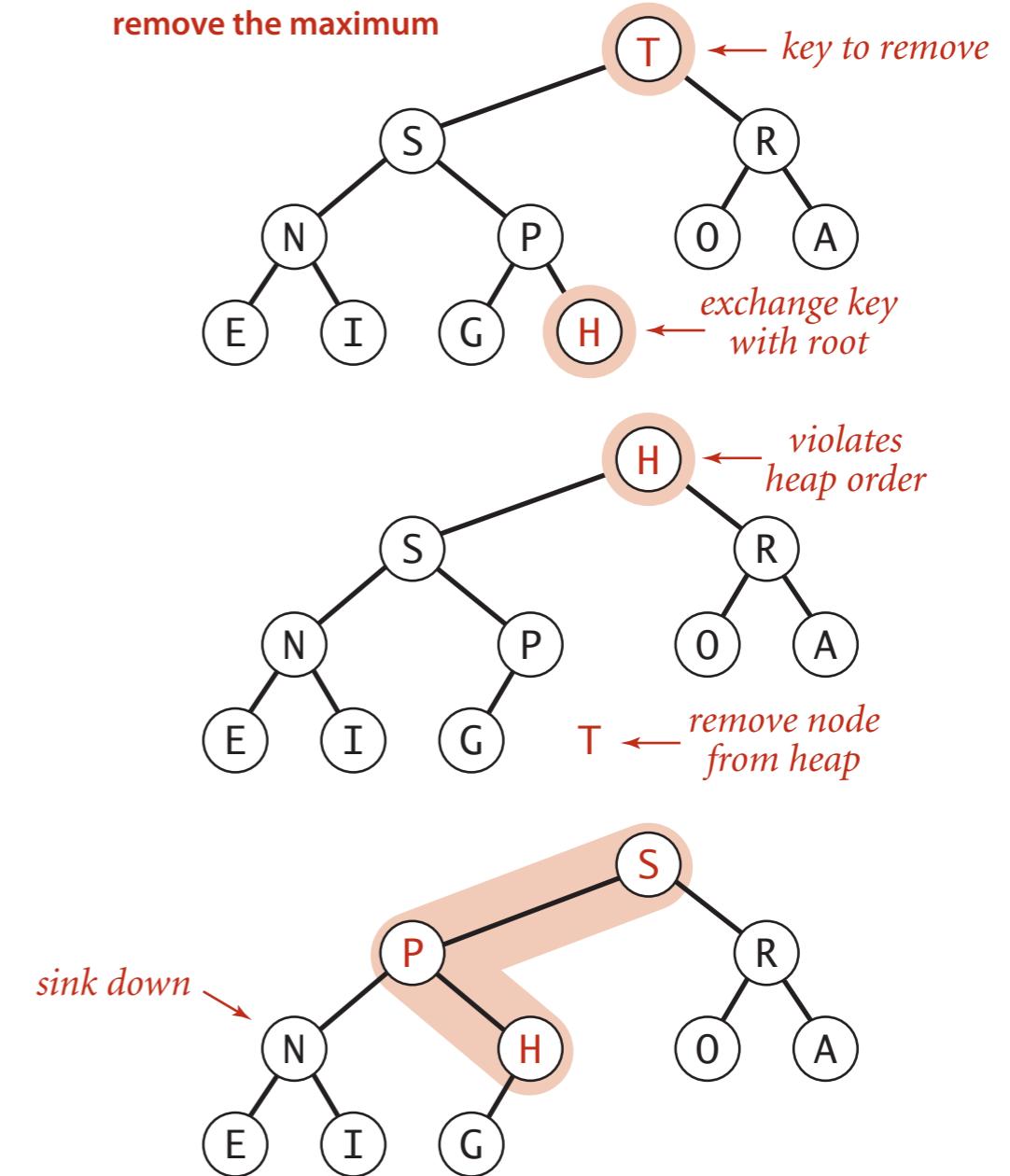
Power struggle. Better subordinate promoted.

Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

```
public Key delMax()
{
    Key max = pq[1];
    exch(1, N--);
    sink(1);
    pq[N+1] = null; ← prevent loitering
    return max;
}
```

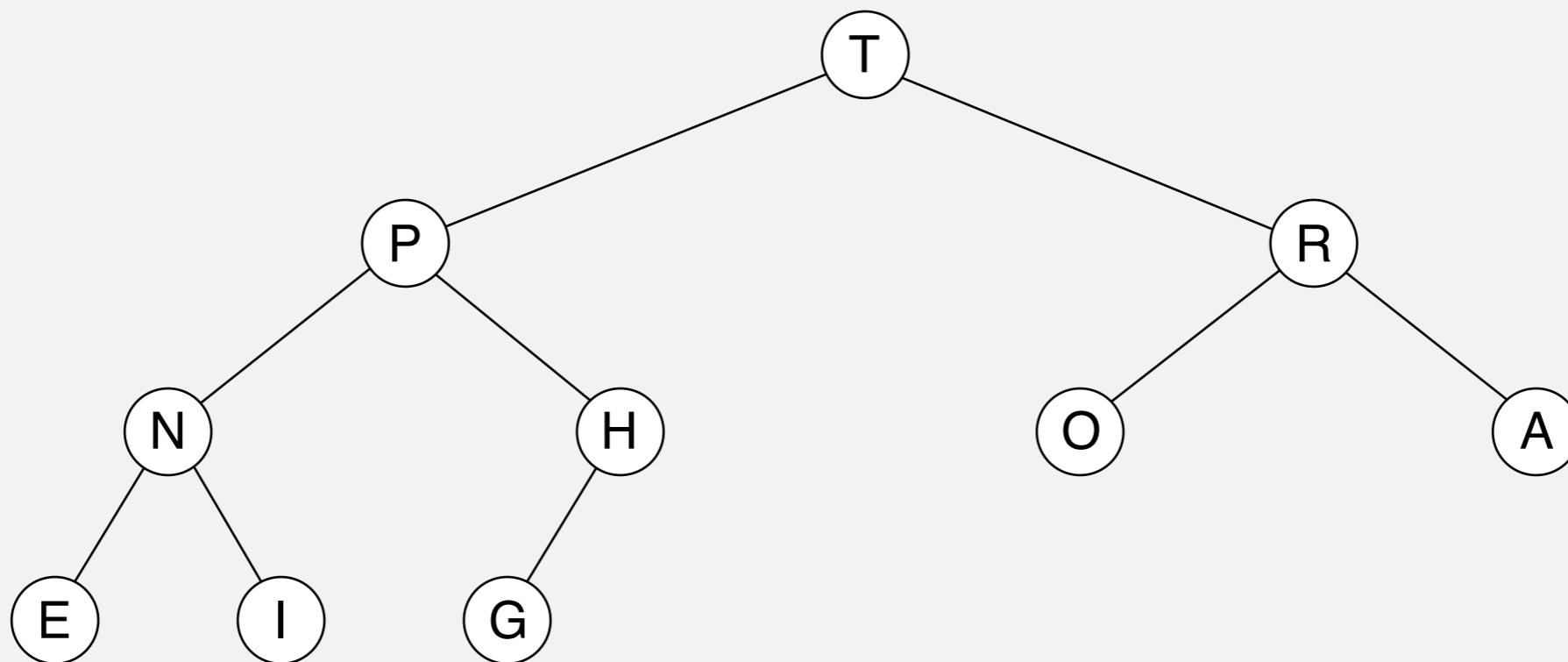


Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



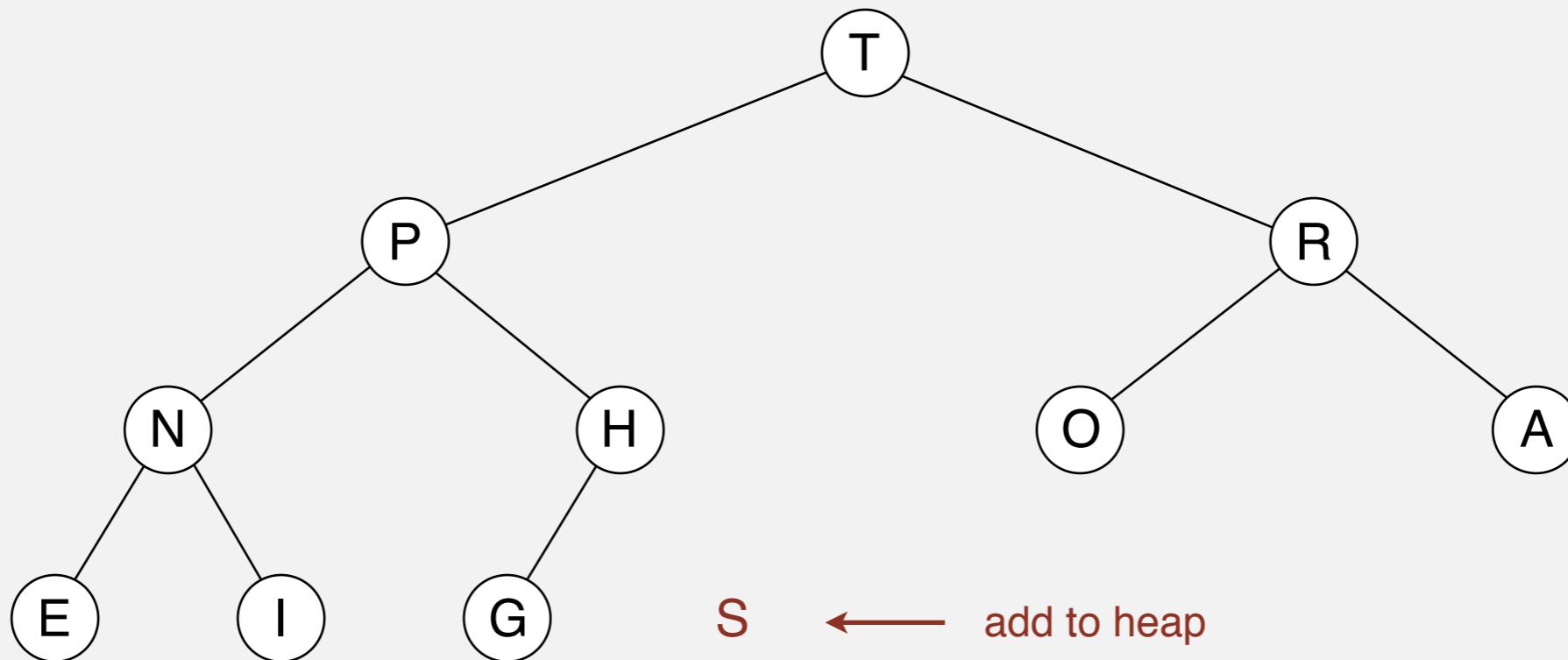
T	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



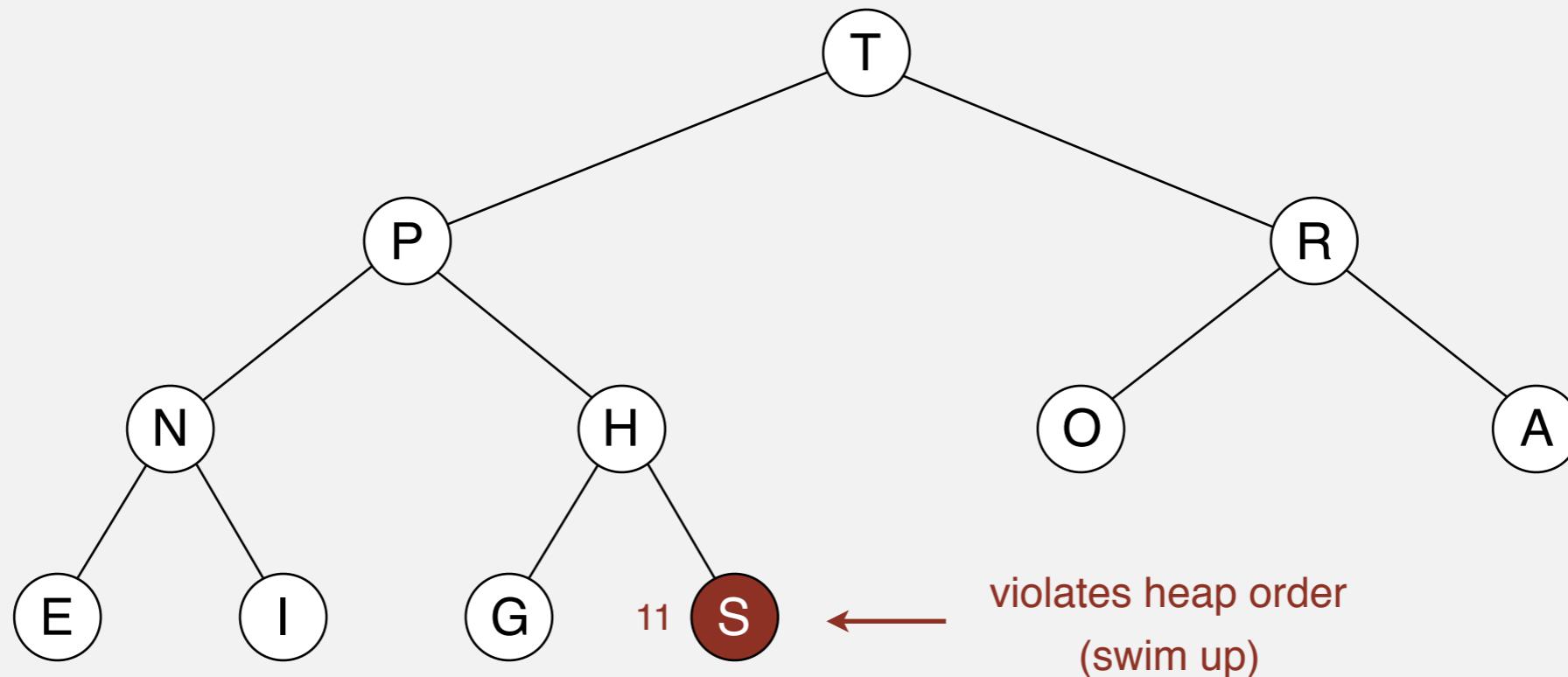
T	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



T	P	R	N	H	O	A	E	I	G	S
---	---	---	---	---	---	---	---	---	---	---

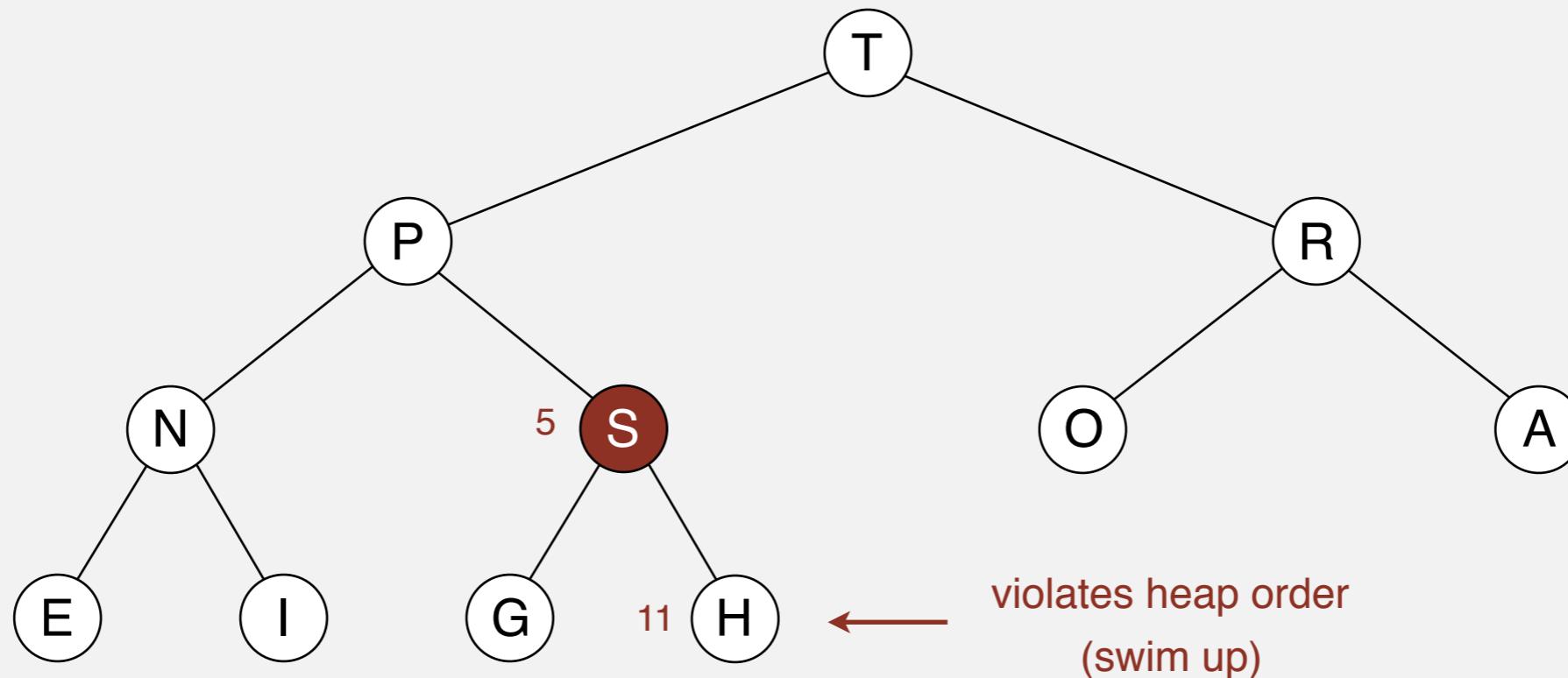
11

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



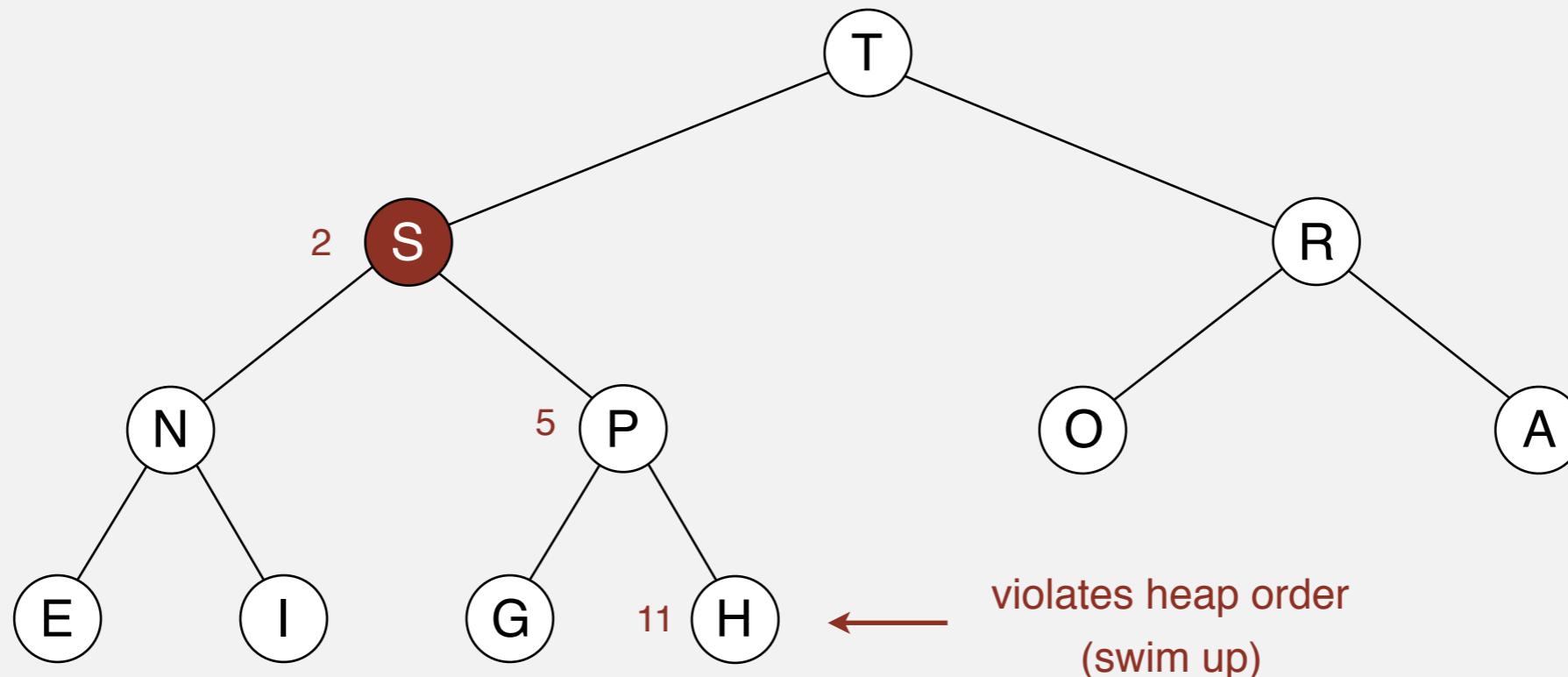
T	P	R	N	S	O	A	E	I	G	H
5									11	

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



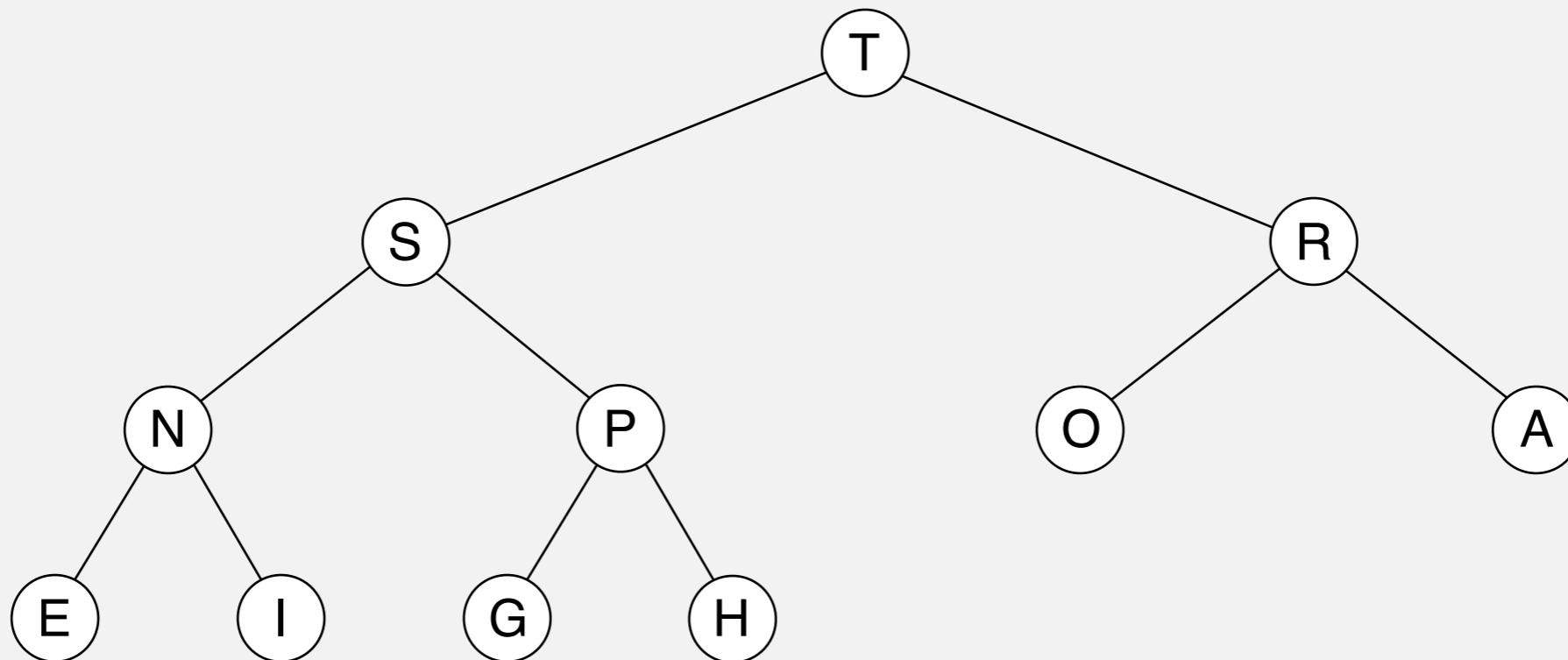
T	S	R	N	P	O	A	E	I	G	H
2	2	5	5	5					11	11

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



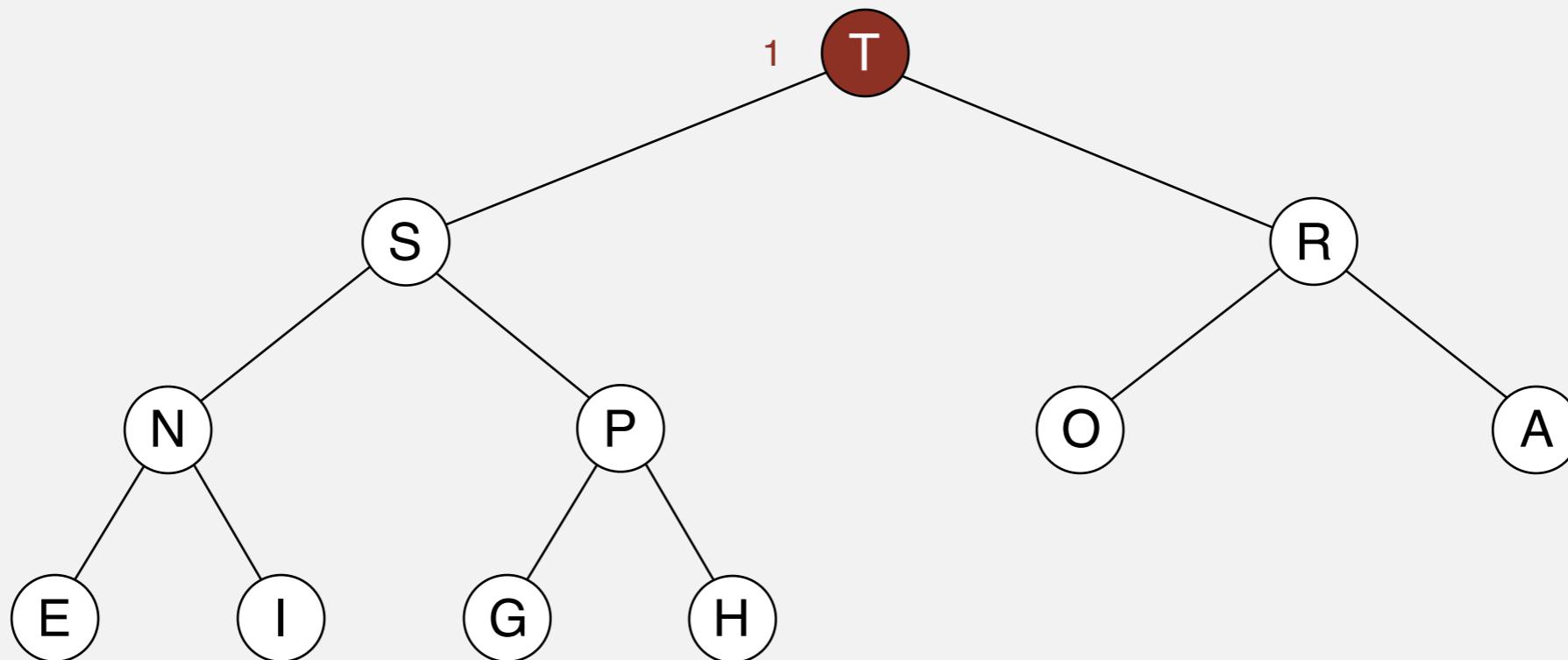
T	S	R	N	P	O	A	E	I	G	H
---	---	---	---	---	---	---	---	---	---	---

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



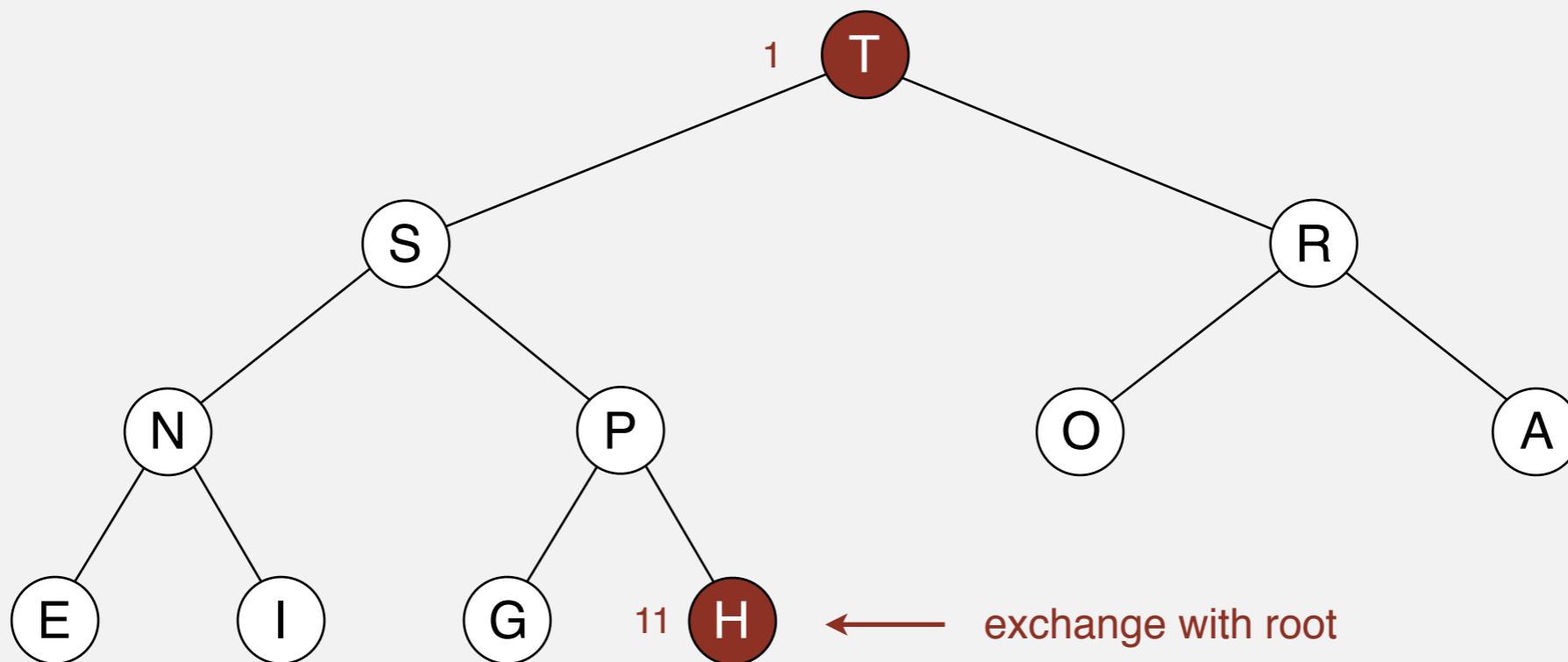
T	S	R	N	P	O	A	E	I	G	H
---	---	---	---	---	---	---	---	---	---	---

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



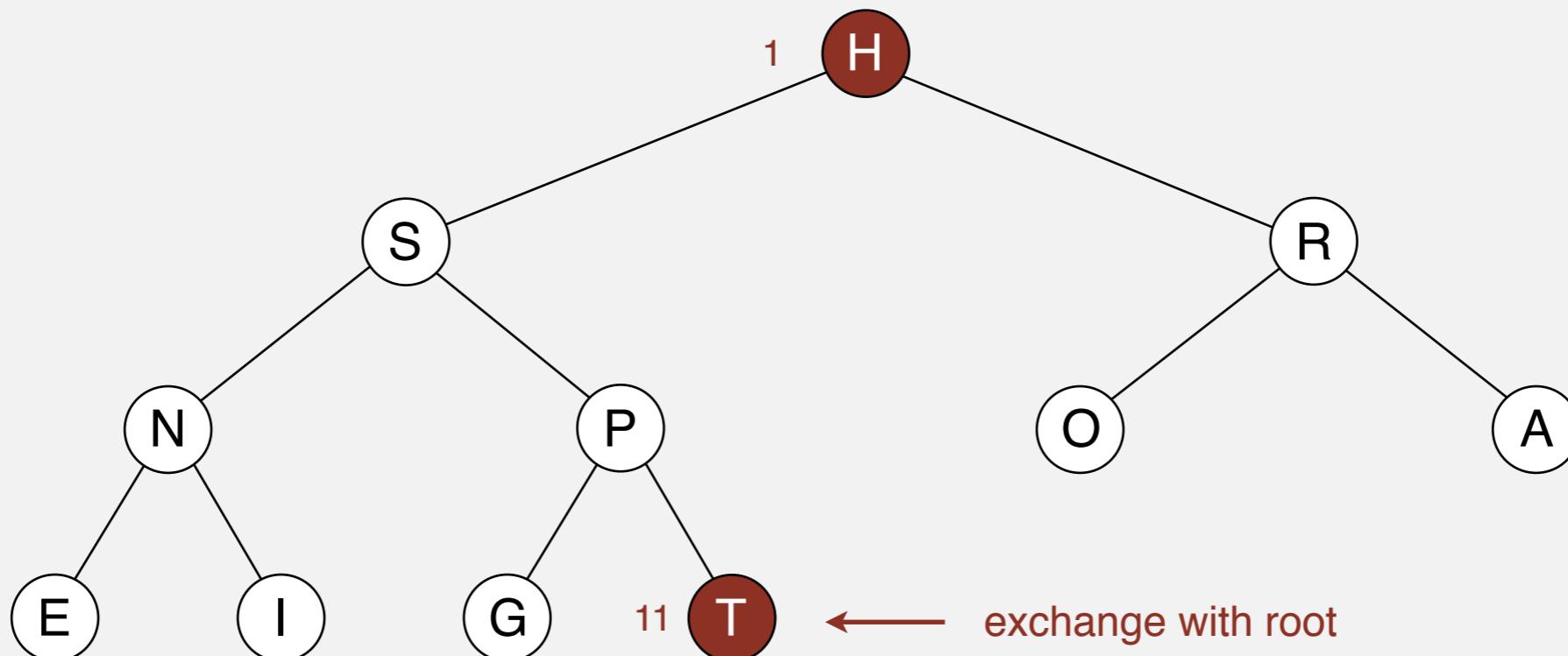
T	S	R	N	P	O	A	E	I	G	H
1										11

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



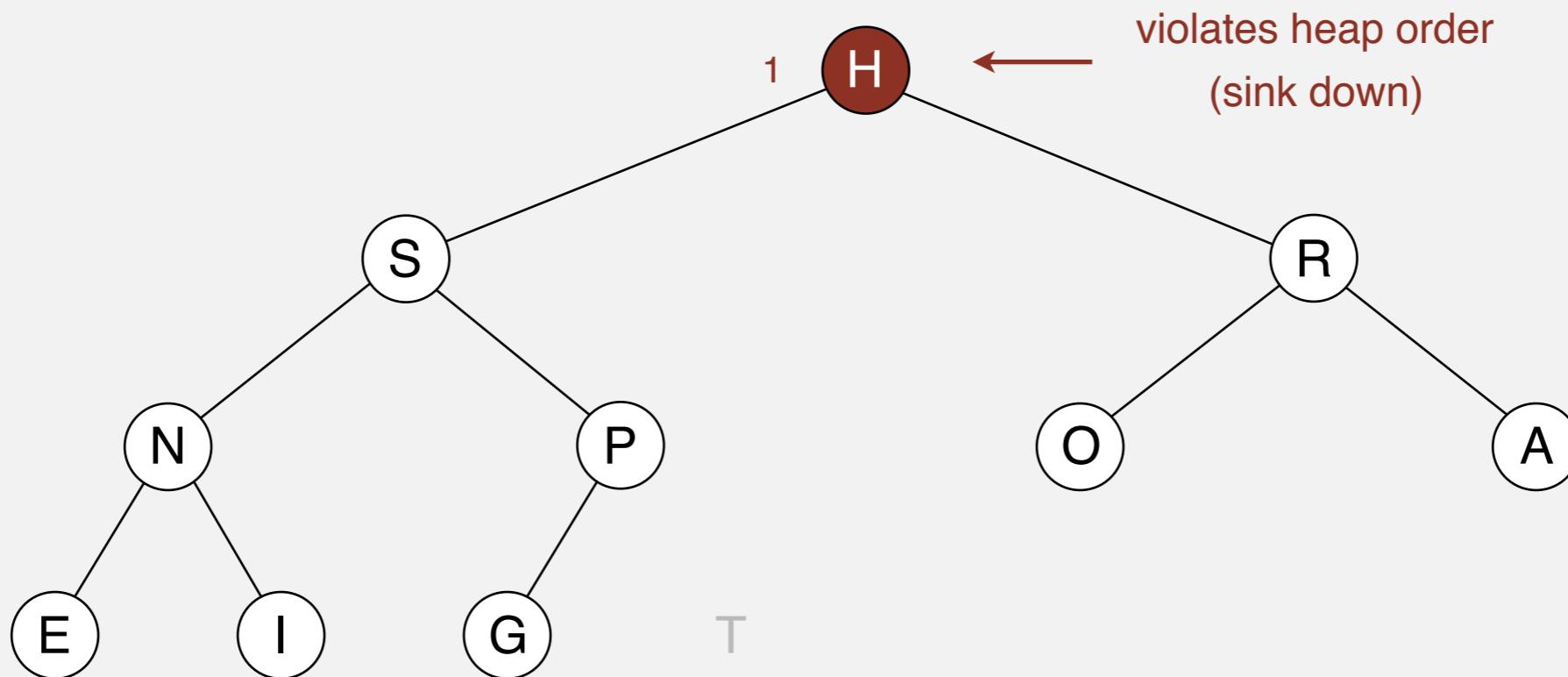
H	S	R	N	P	O	A	E	I	G	T
1										11

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



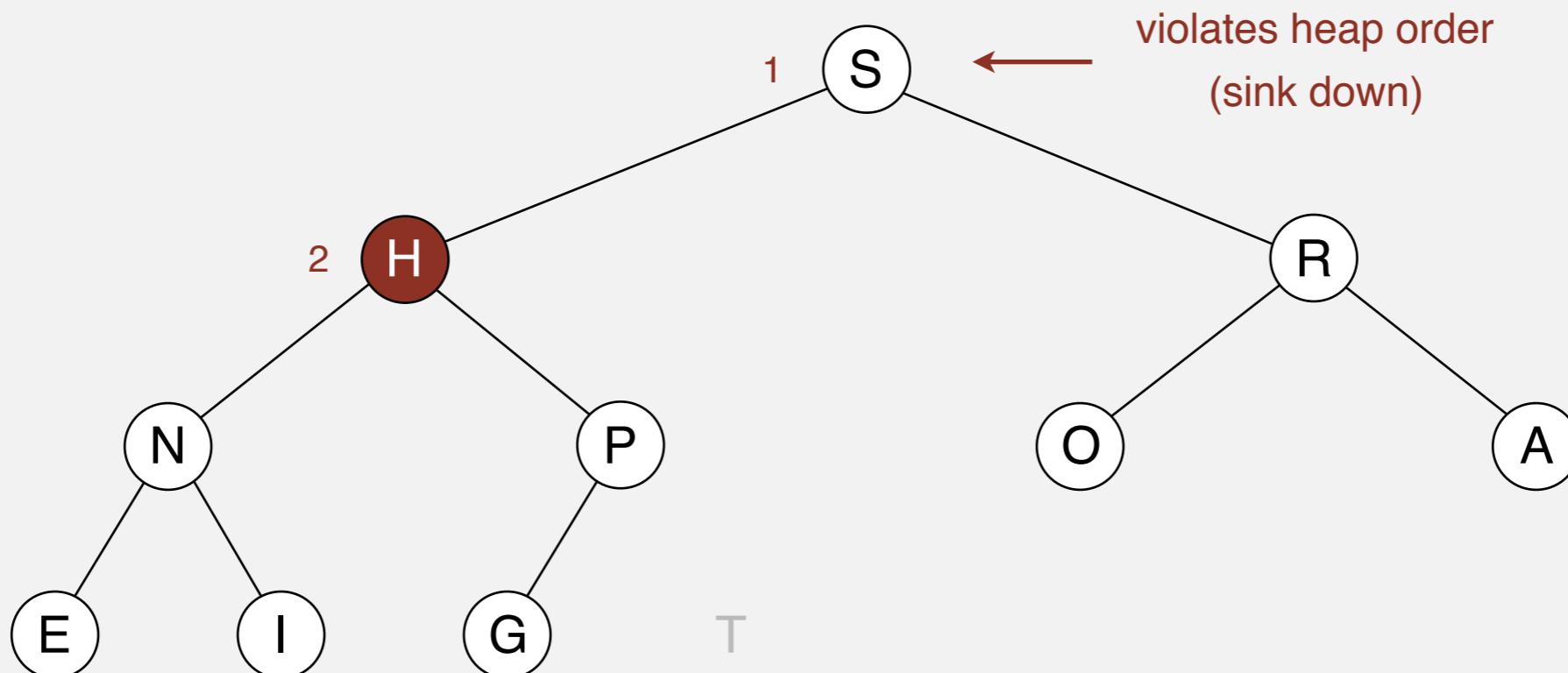
H	S	R	N	P	O	A	E	I	G	T
---	---	---	---	---	---	---	---	---	---	---

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



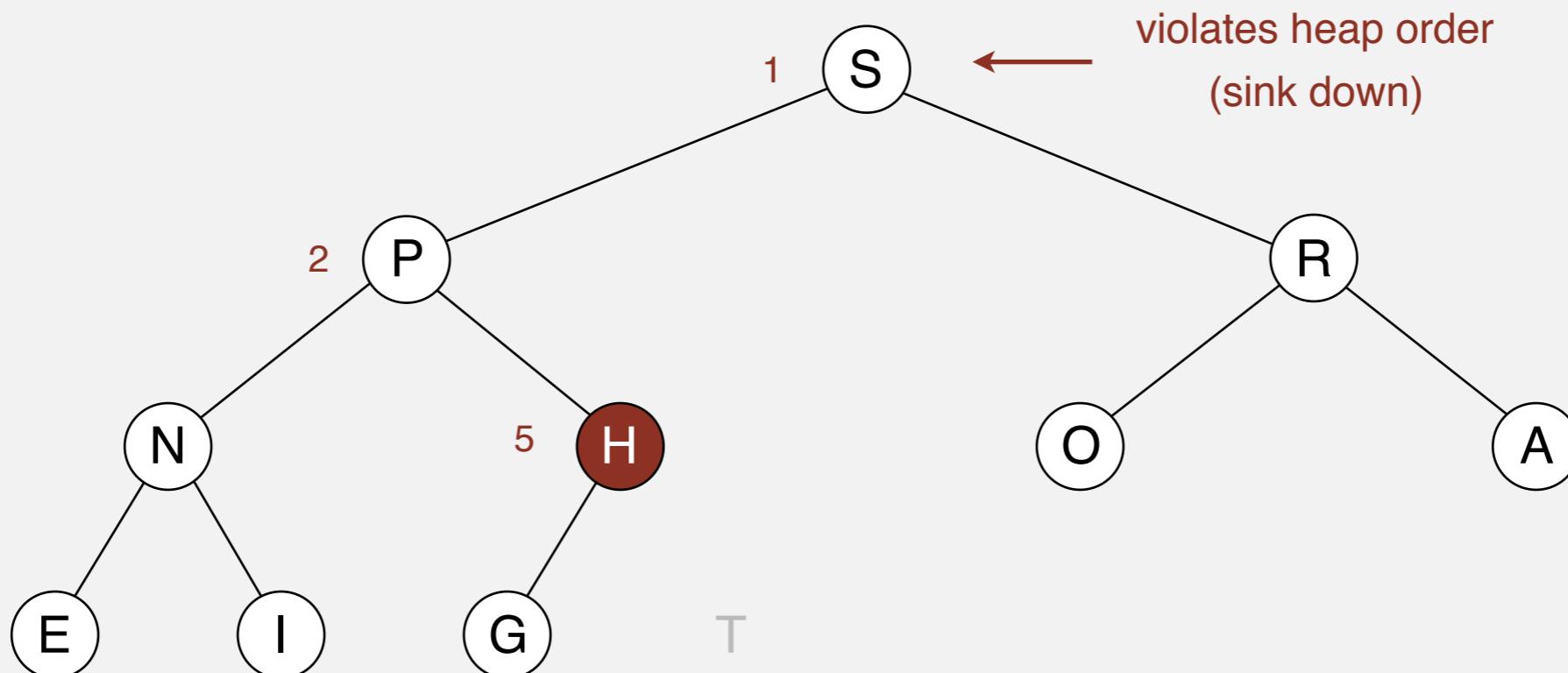
S	H	R	N	P	O	A	E	I	G	T
1	2									

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



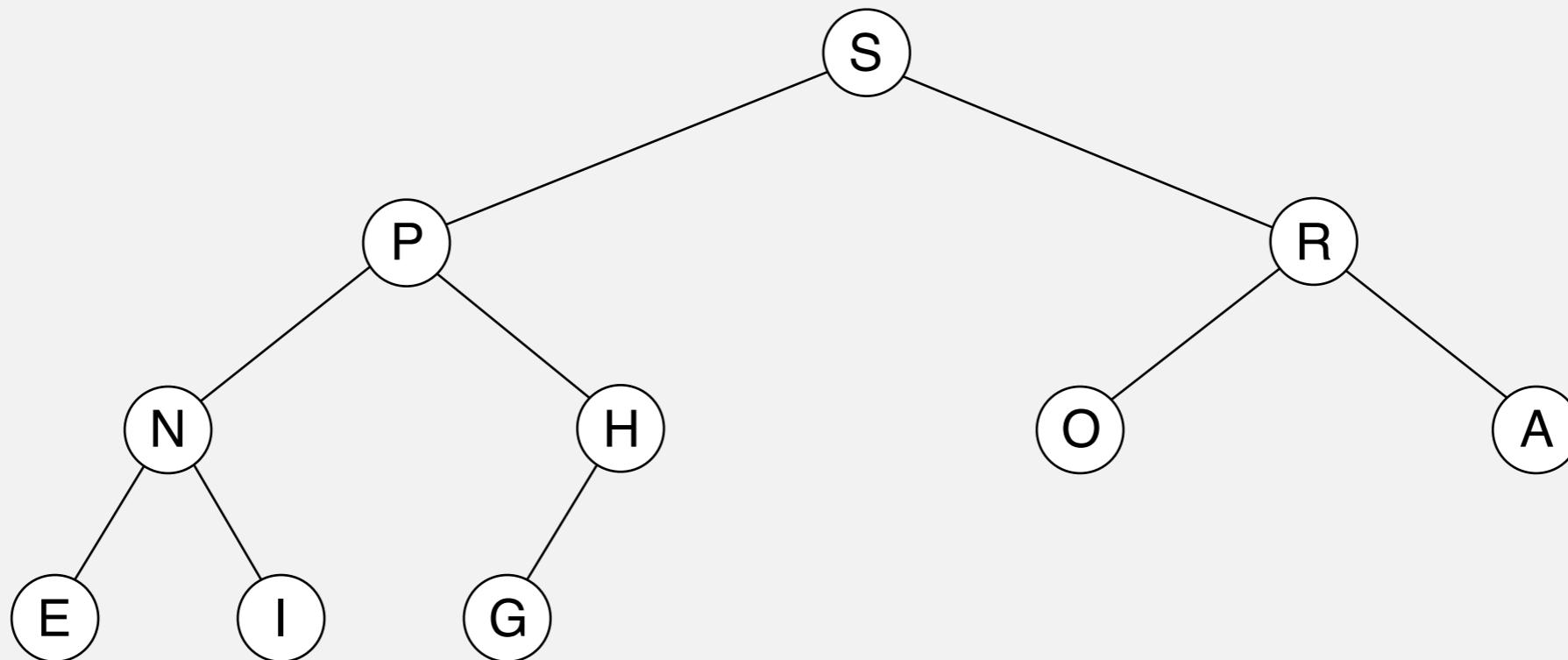
S	P	R	N	H	O	A	E	I	G	T
1	2			5						

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



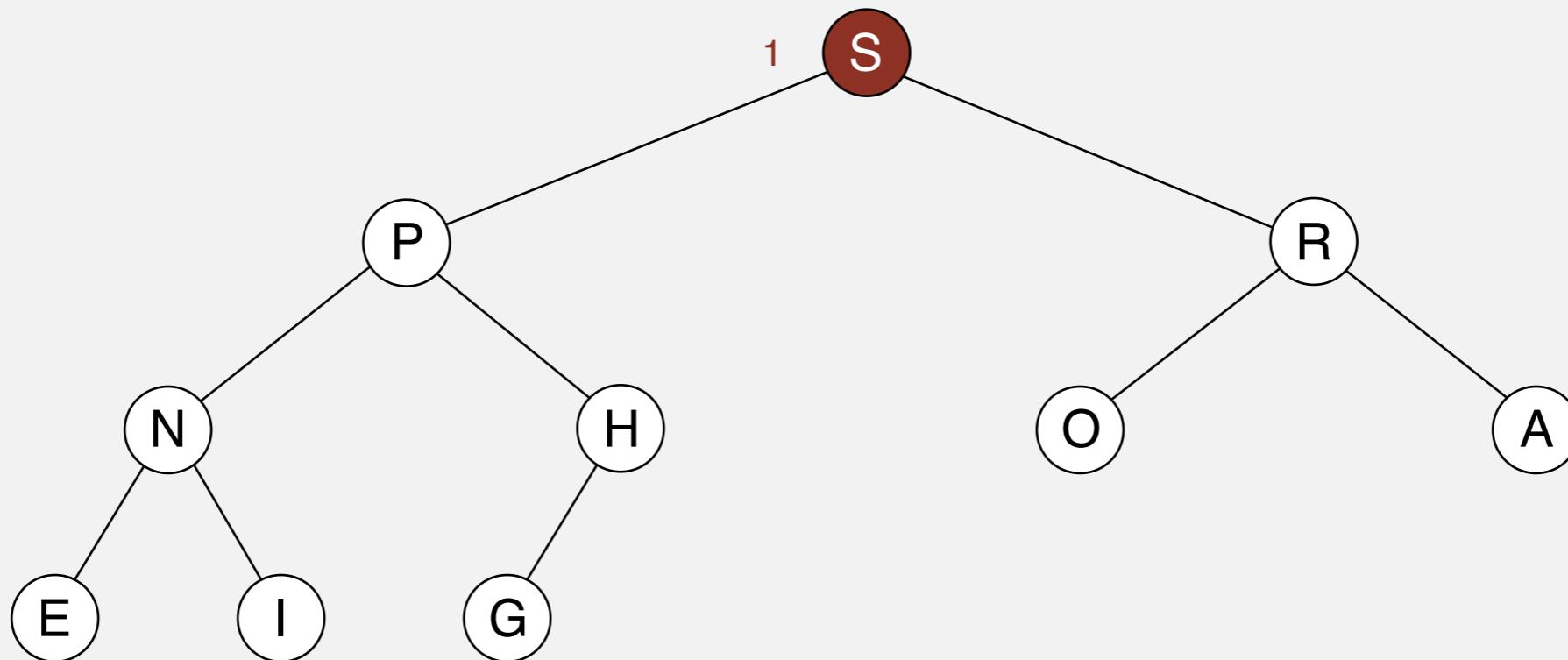
S	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



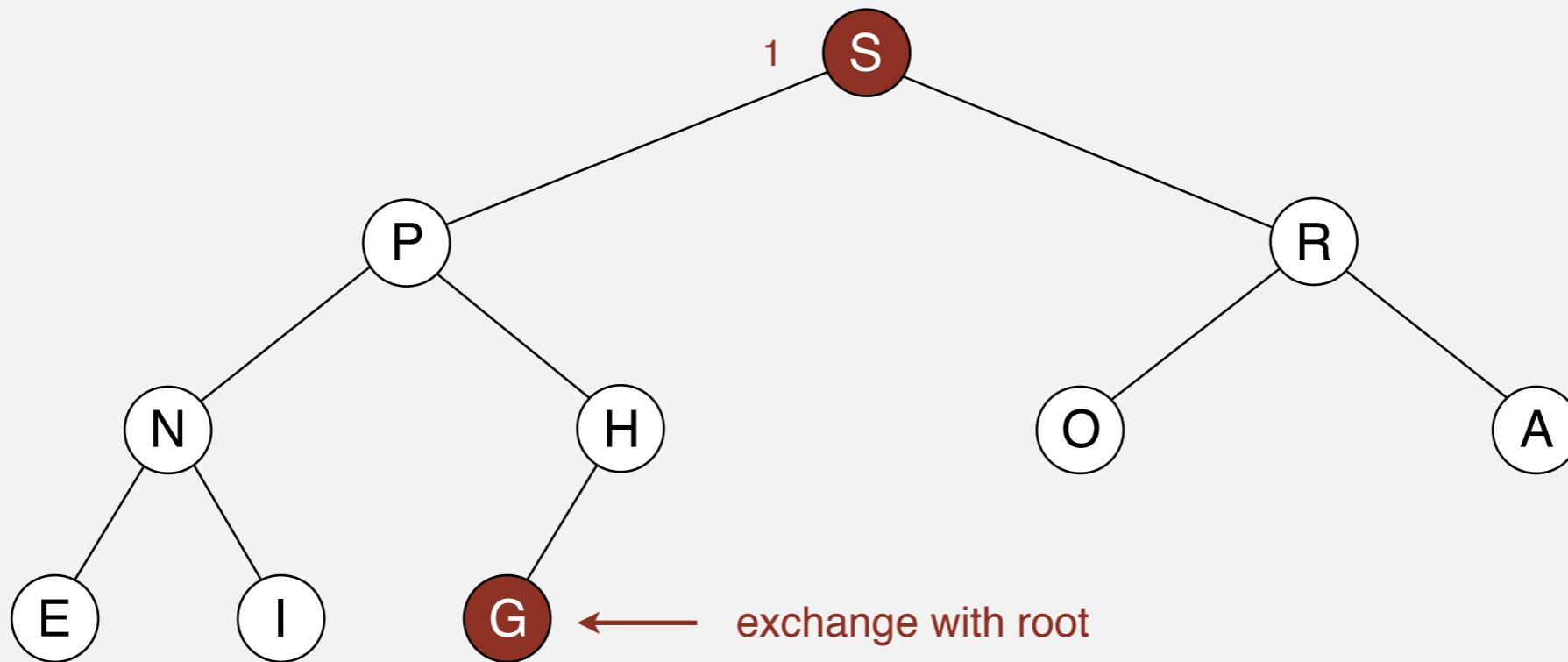
S | P | R | N | H | O | A | E | I | G

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



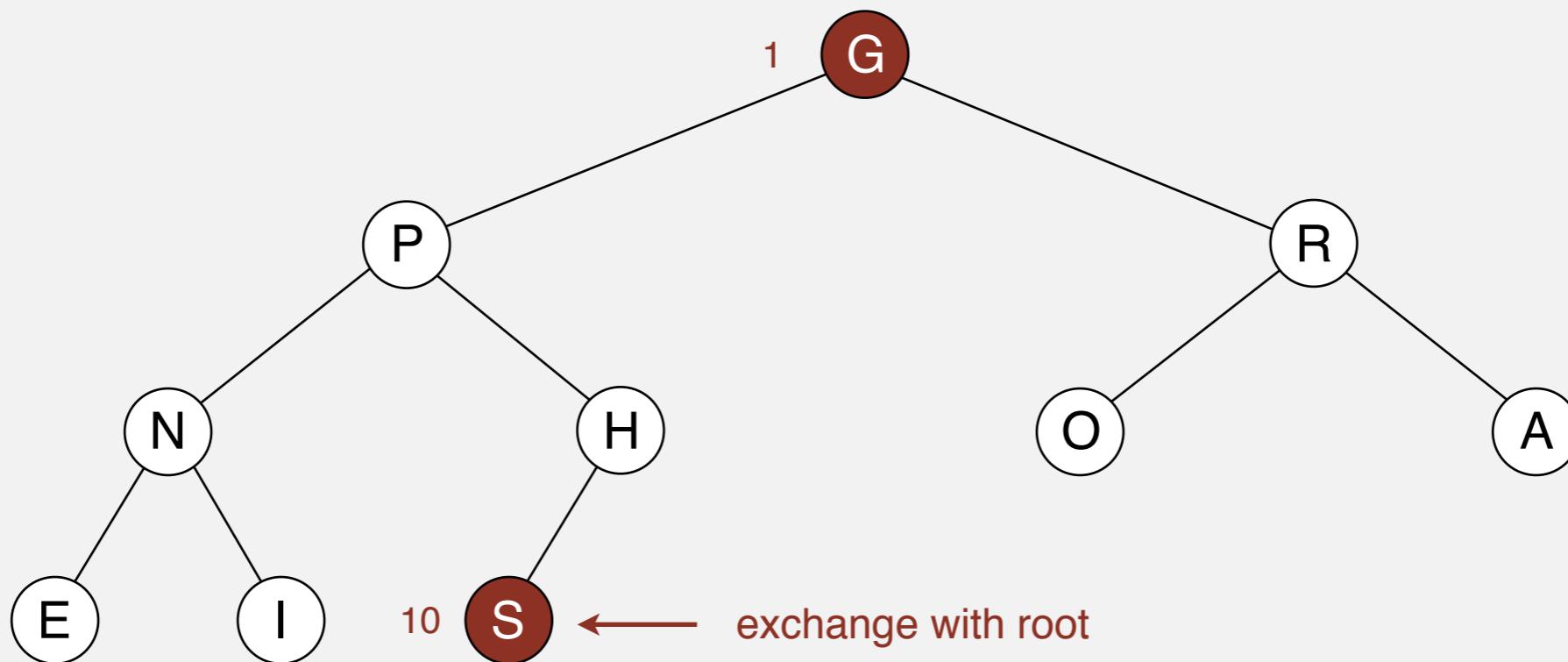
S	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



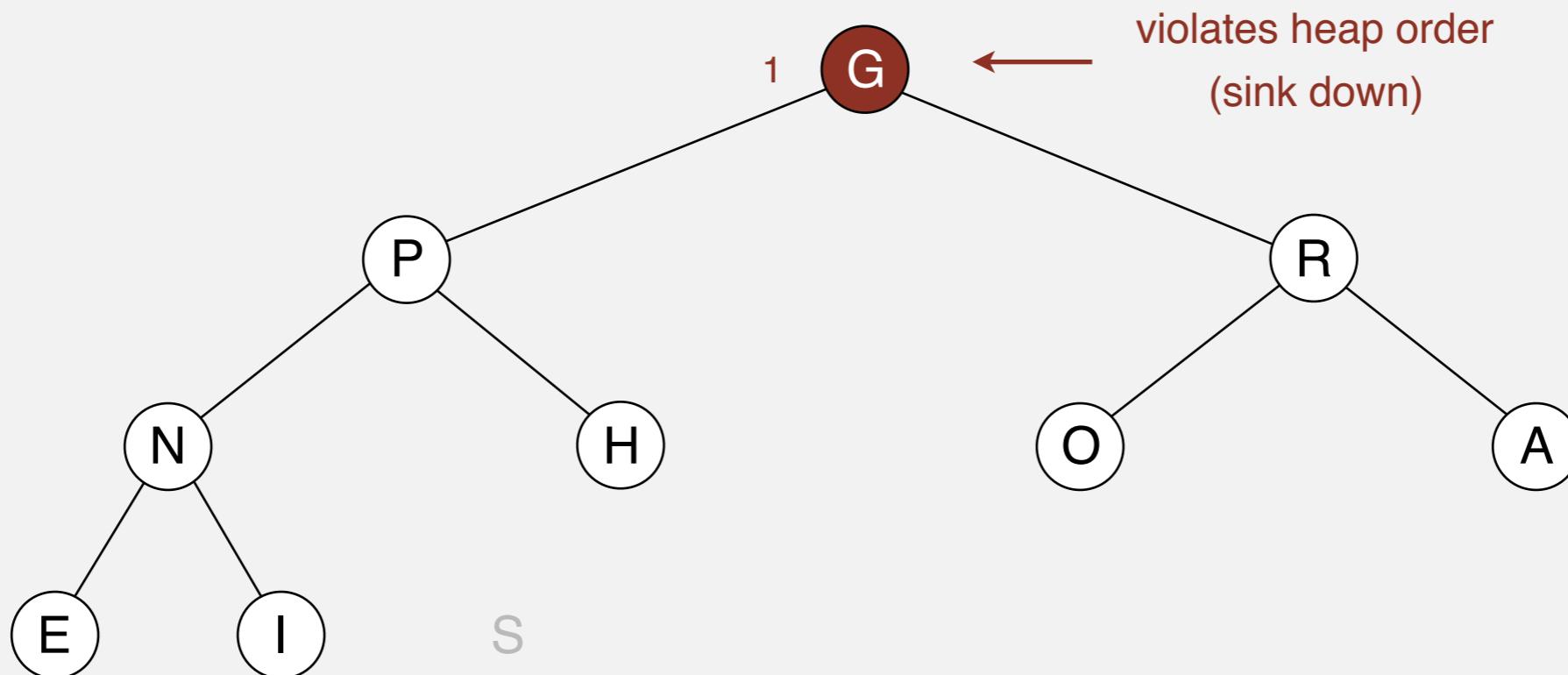
G	P	R	N	H	O	A	E	I	S	
1									10	

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



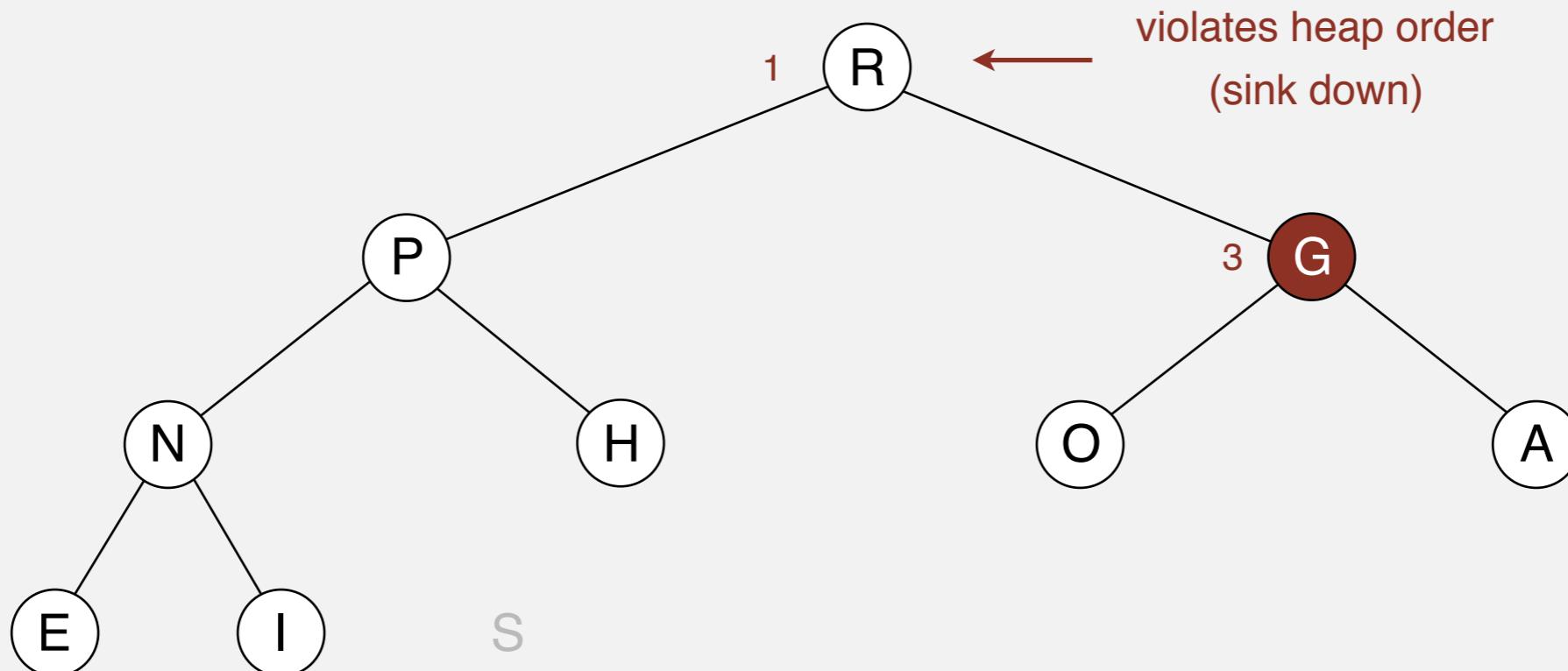
G	P	R	N	H	O	A	E	I	S
---	---	---	---	---	---	---	---	---	---

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



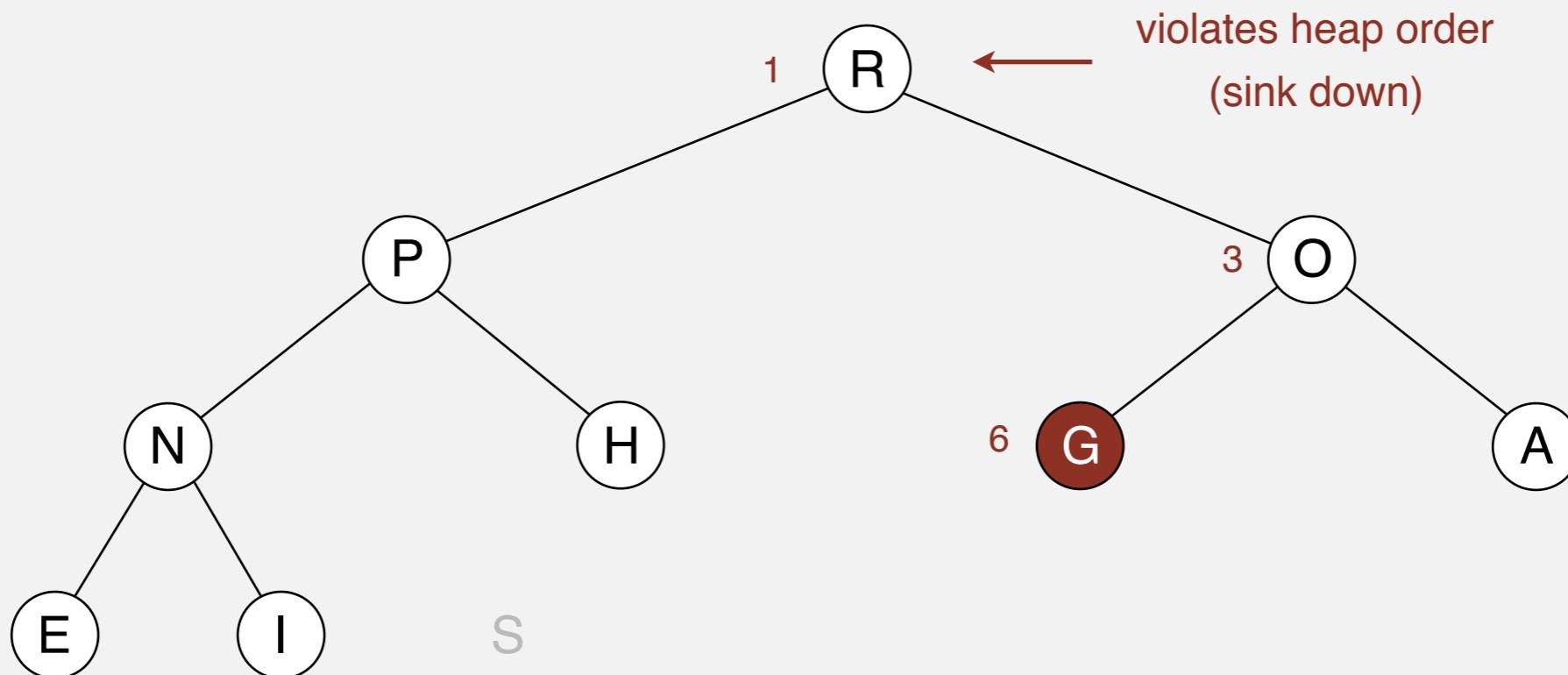
R	P	G	N	H	O	A	E	I	S
1		3							

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum



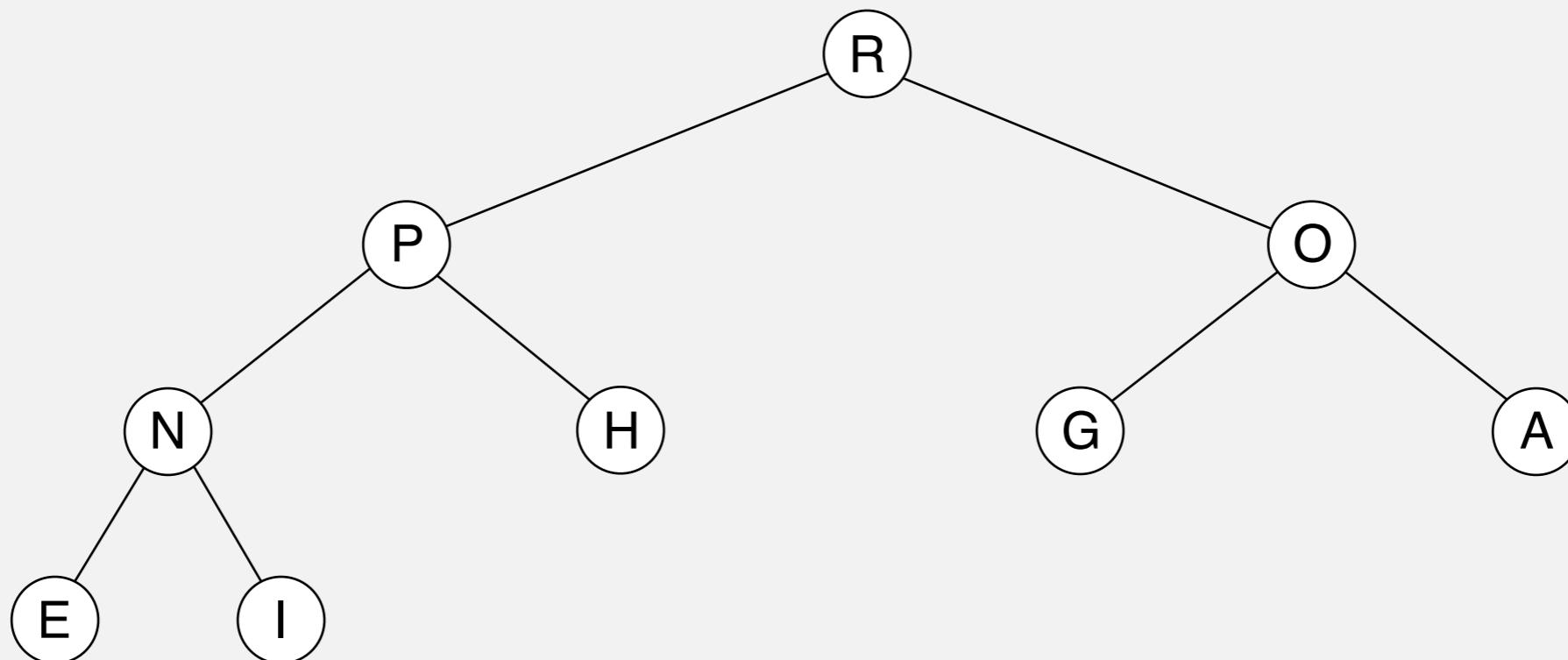
R	P	O	N	H	G	A	E	I	S	
1	3	3	3	3	6	3	3	3	3	

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



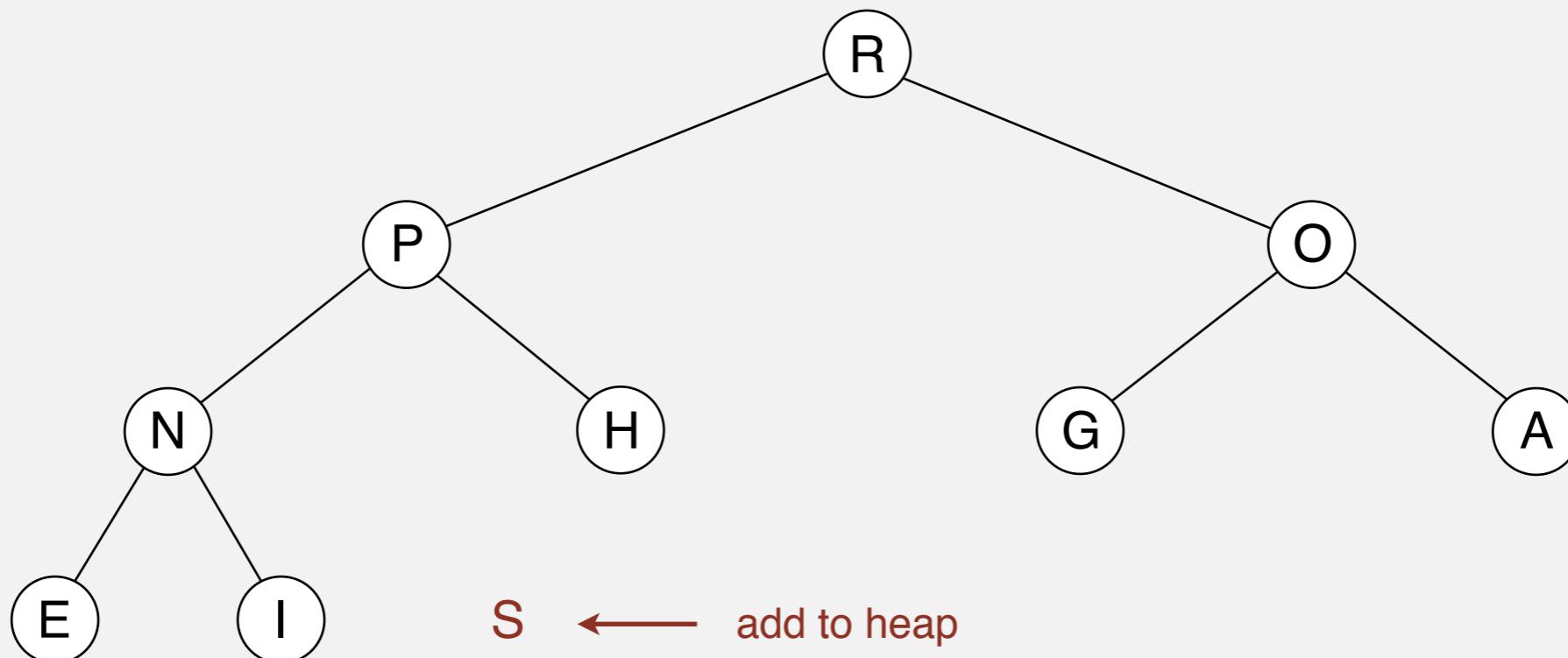
R	P	O	N	H	G	A	E	I		
---	---	---	---	---	---	---	---	---	--	--

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



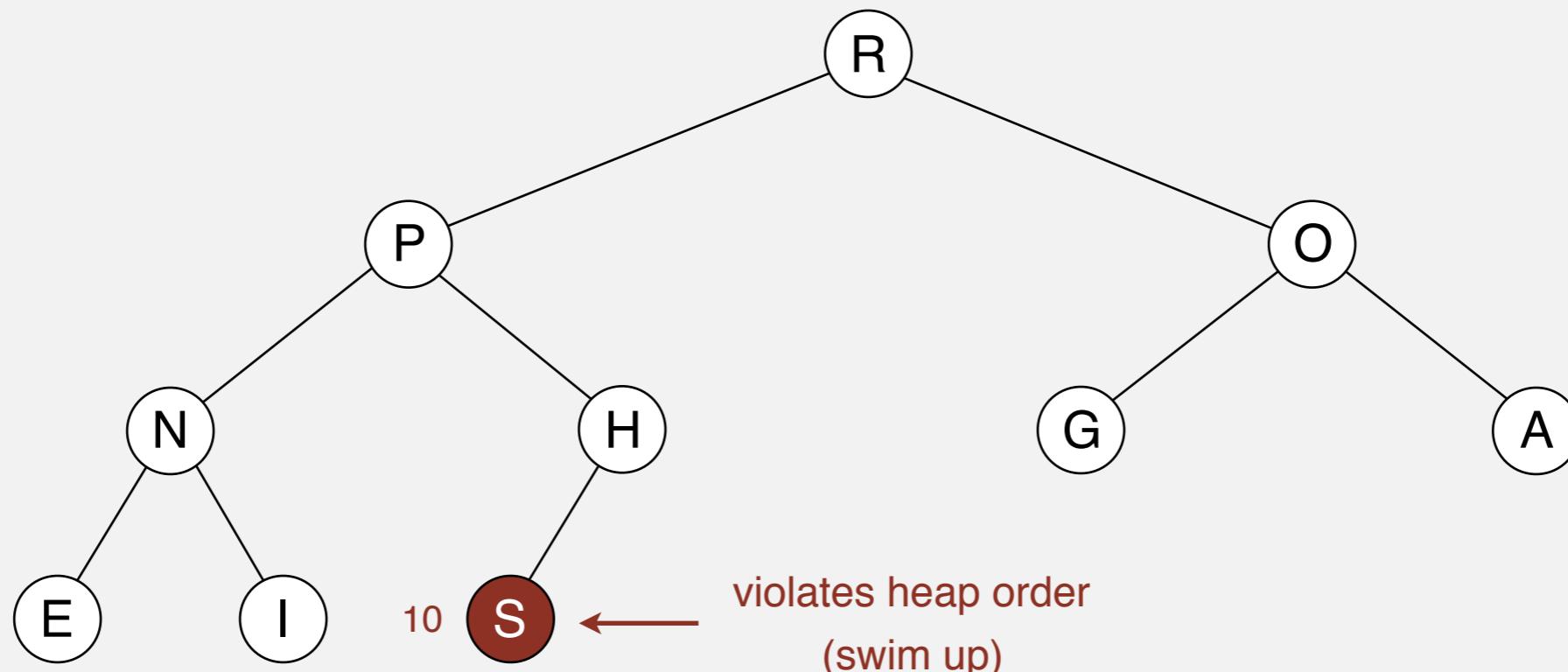
R	P	O	N	H	G	A	E	I	S	
---	---	---	---	---	---	---	---	---	---	--

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



R	P	O	N	H	G	A	E	I	S	
---	---	---	---	---	---	---	---	---	---	--

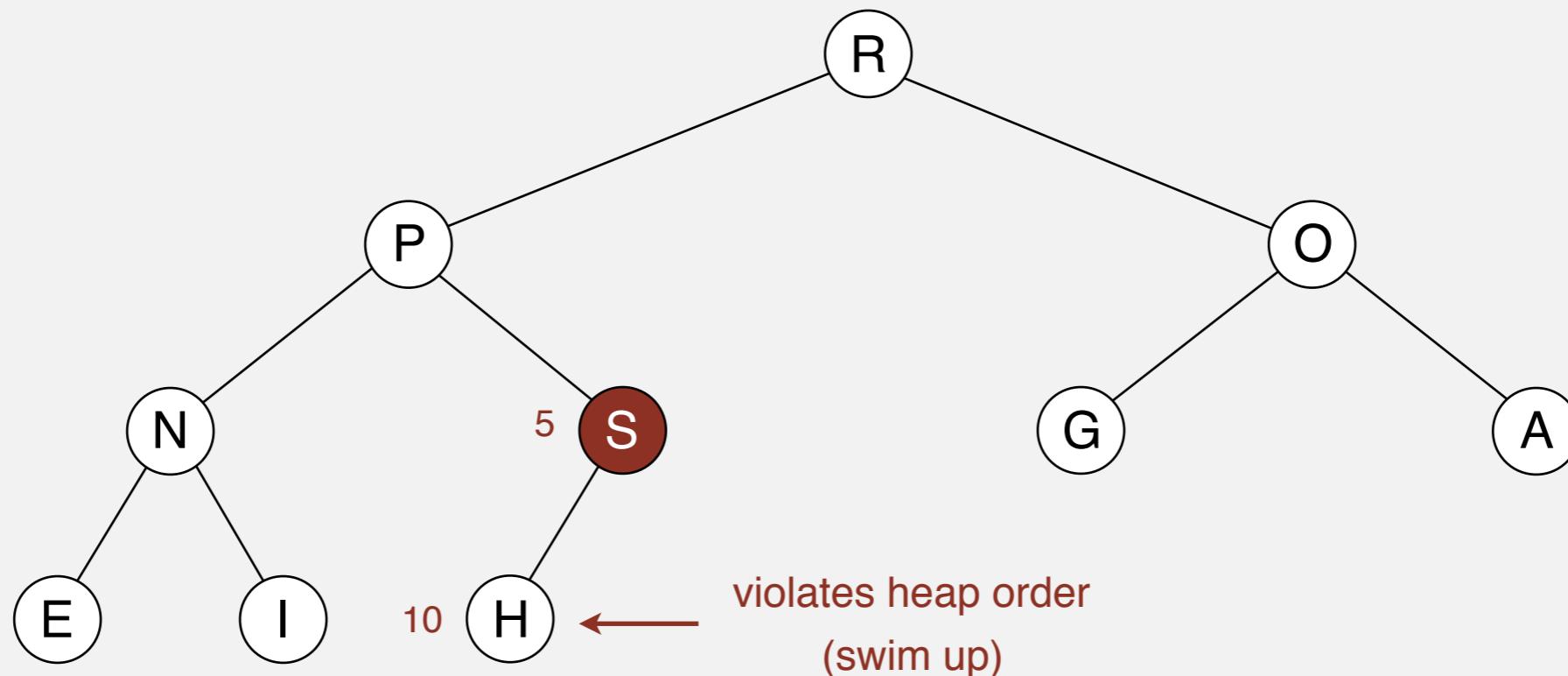
10

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



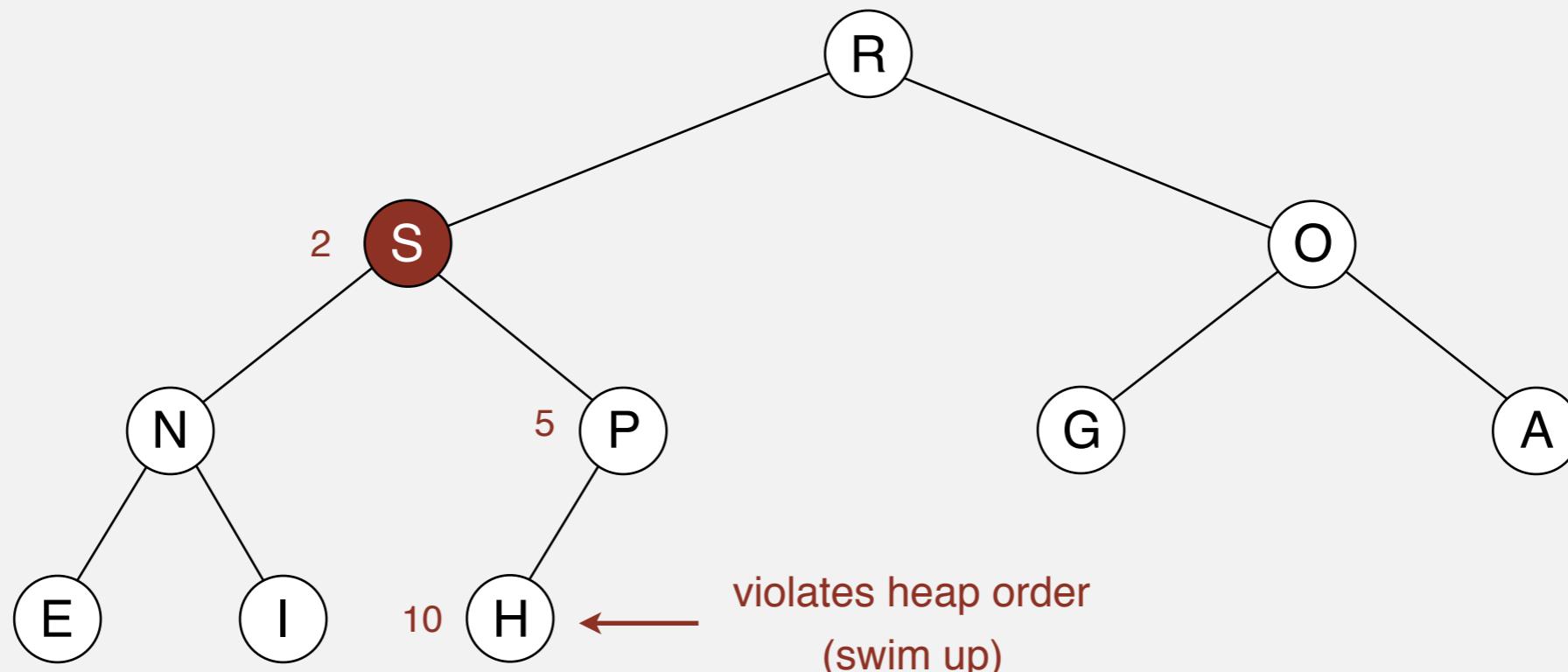
R	P	O	N	S	G	A	E	I	H	
5							10			

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



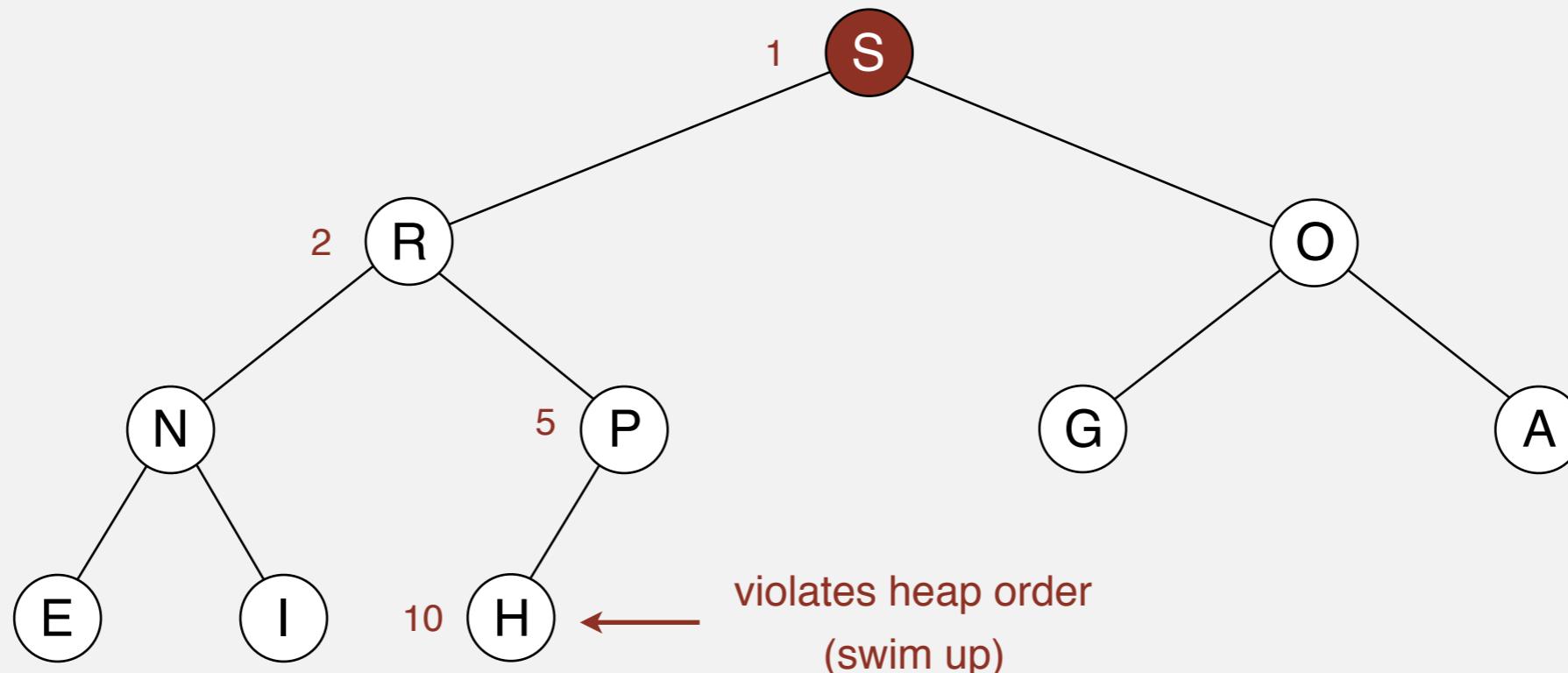
R	S	O	N	P	G	A	E	I	H	
2	2			5					10	

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

insert S



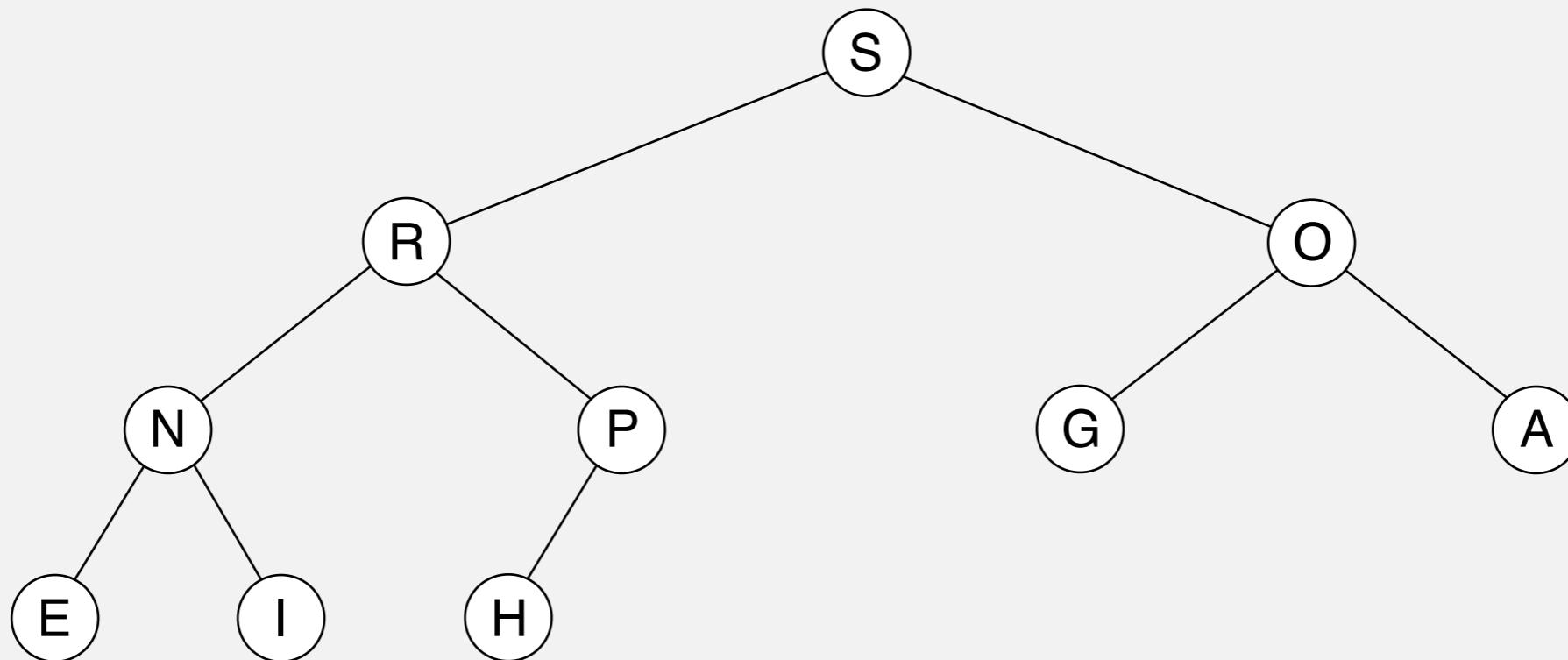
S	R	O	N	P	G	A	E	I	H	
1	2			5					10	

Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered



S	R	O	N	P	G	A	E	I	H	
---	---	---	---	---	---	---	---	---	---	--

Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;
    private int N;

    public MaxPQ(int capacity)
    {   pq = (Key[]) new Comparable[capacity+1]; }

    public boolean isEmpty()
    {   return N == 0; }

    public void insert(Key key)
    {   /* see previous code */ }

    public Key delMax()
    {   /* see previous code */ }

    private void swim(int k)
    {   /* see previous code */ }

    private void sink(int k)
    {   /* see previous code */ }

    private boolean less(int i, int j)
    {   return pq[i].compareTo(pq[j]) < 0; }

    private void exch(int i, int j)
    {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t; }

}
```

← PQ ops

← heap helper functions

← array helper functions

Priority queues implementation cost summary

order-of-growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log N$	$\log N$	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	$\log N^\dagger$	1
impossible	1	1	1

← why impossible?

† amortized

Binary heap considerations

Immutability of keys.

- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N

amortized time per op

(how to make worst case?)

Minimum-oriented priority queue.

- Replace `less()` with `greater()`.
- Implement `greater()`.

Other operations.

- Remove an arbitrary item.
- Change the priority of an item.

can implement with `sink()` and `swim()` [stay tuned]

Immutability: implementing in Java

Data type. Set of values and operations on those values.

Immutable data type. Can't change the data type value once created.

```
public final class Vector {  
    private final int N;  
    private final double[] data;  
  
    public Vector(double[] data) {  
        this.N = data.length;  
        this.data = new double[N];  
        for (int i = 0; i < N; i++)  
            this.data[i] = data[i];  
    }  
  
    ...  
}
```

The diagram shows a Java code snippet for a `Vector` class. Annotations with arrows point to specific parts of the code:

- An arrow points to the `final` keyword in the class declaration with the text "can't override instance methods".
- An arrow points to the `private final` declarations for `N` and `data` with the text "all instance variables private and final".
- An arrow points to the assignment of `this.data` in the constructor with the text "defensive copy of mutable instance variables".
- An arrow points to the assignment in the constructor loop with the text "instance methods don't change instance variables".

Immutable. `String`, `Integer`, `Double`, `Color`, `Vector`, `Transaction`, `Point2D`.

Mutable. `StringBuilder`, `Stack`, `Counter`, `Java array`.

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can't change the data type value once created.

Advantages.

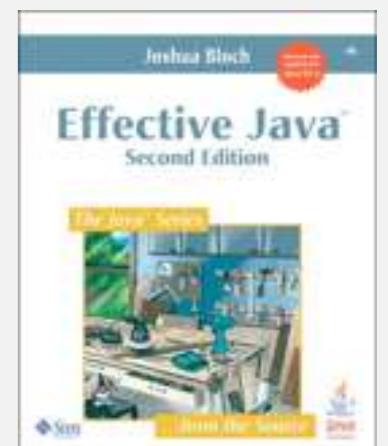
- Simplifies debugging.
- Safer in presence of hostile code.
- Simplifies concurrent programming.
- Safe to use as key in priority queue or symbol table.



Disadvantage. Must create new object for each data type value.

“Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible.”

— Joshua Bloch (Java architect)



PRIORITY QUEUES AND HEAPSORT

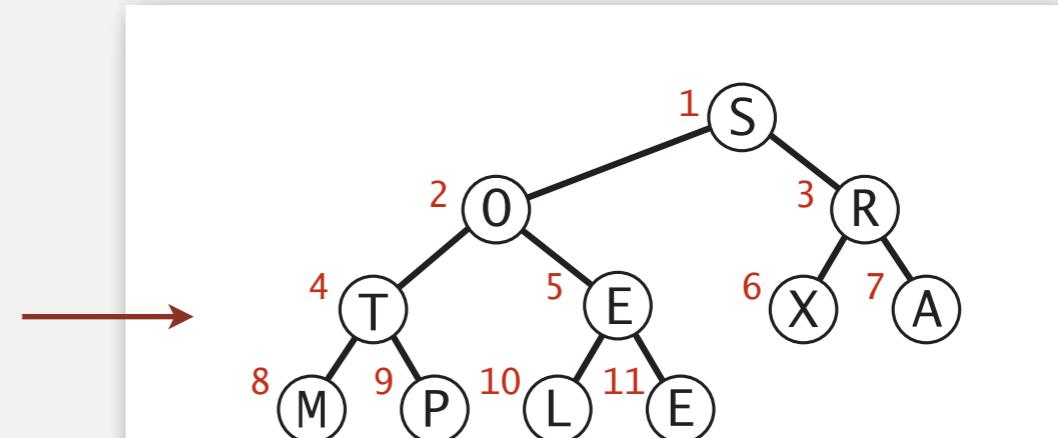
- ▶ Heapsort
- ▶ API
- ▶ Elementary implementations
- ▶ Binary heaps
- ▶ Heapsort

Heapsort

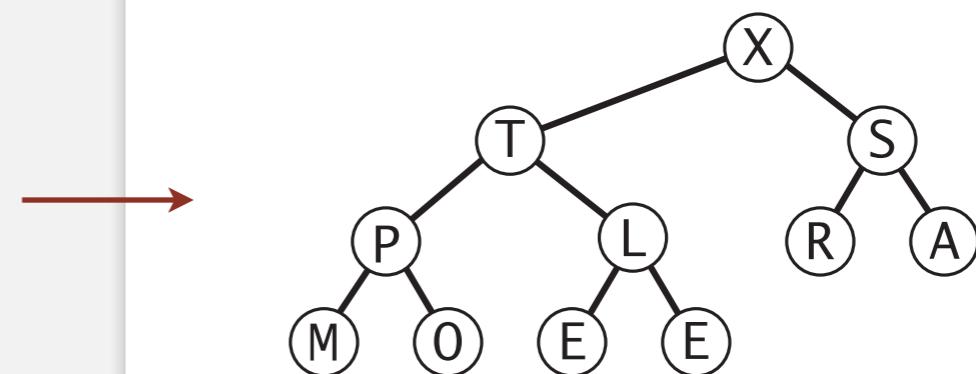
Basic plan for in-place sort.

- Create max-heap with all N keys.
- Repeatedly remove the maximum key.

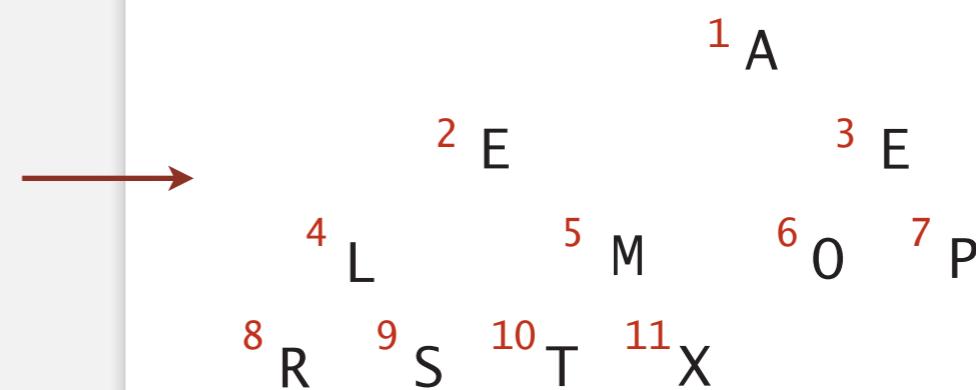
start with array of keys
in arbitrary order



build a max-heap
(in place)



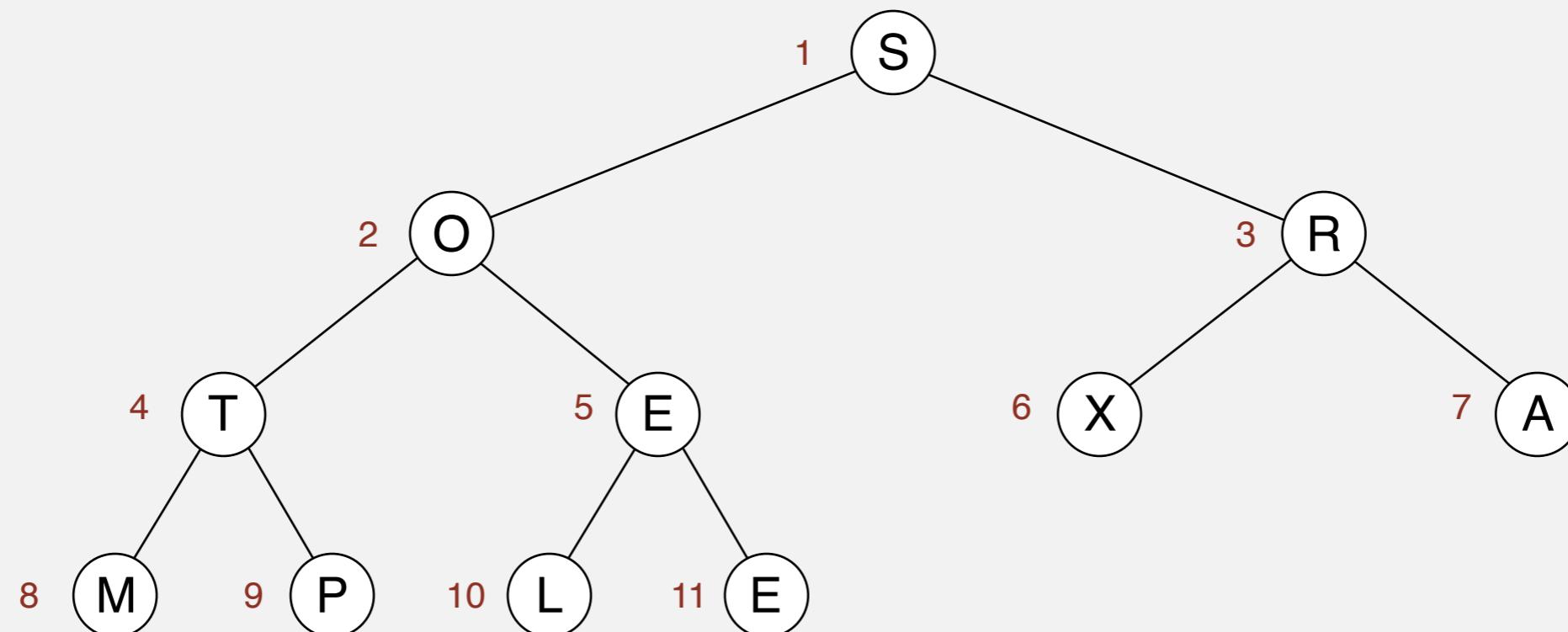
sorted result
(in place)



Starting point. Array in arbitrary order.



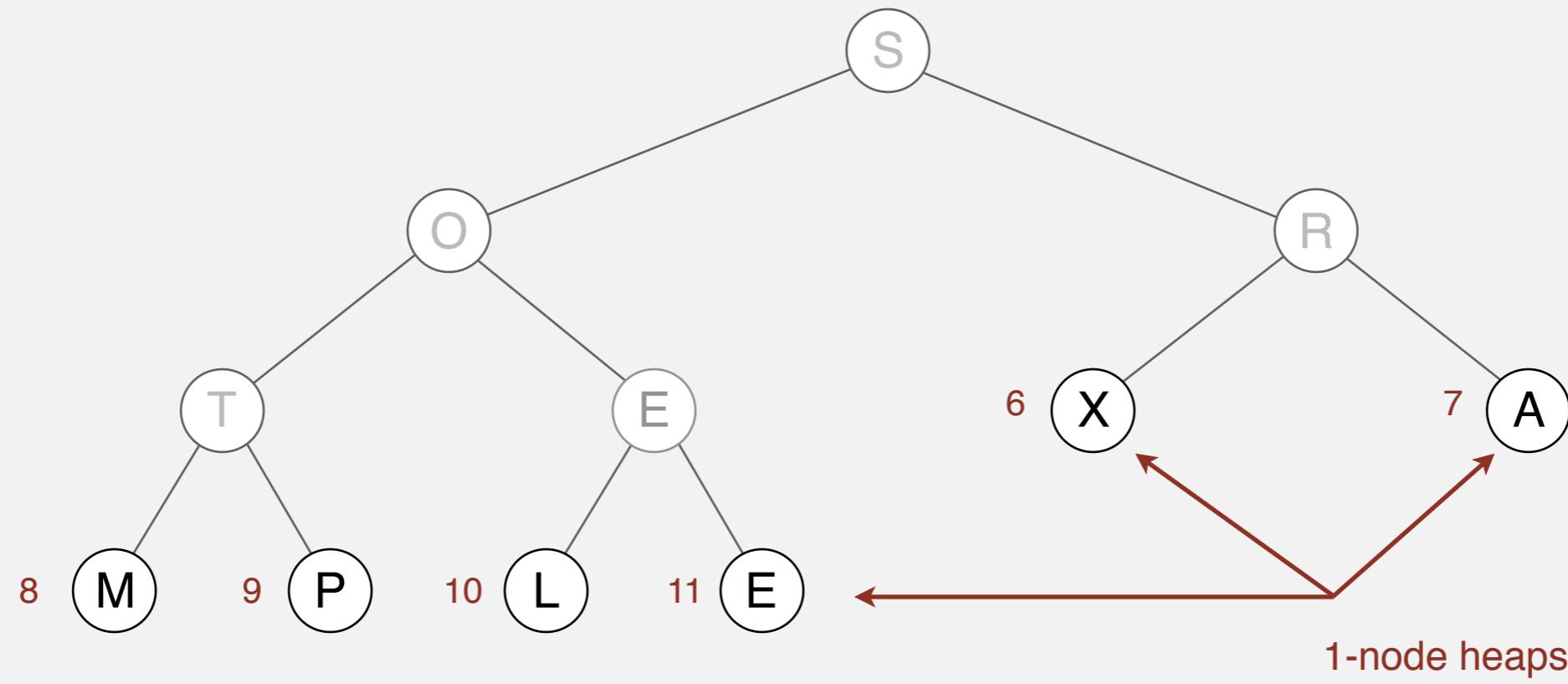
we assume array entries are indexed 1 to N



S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort

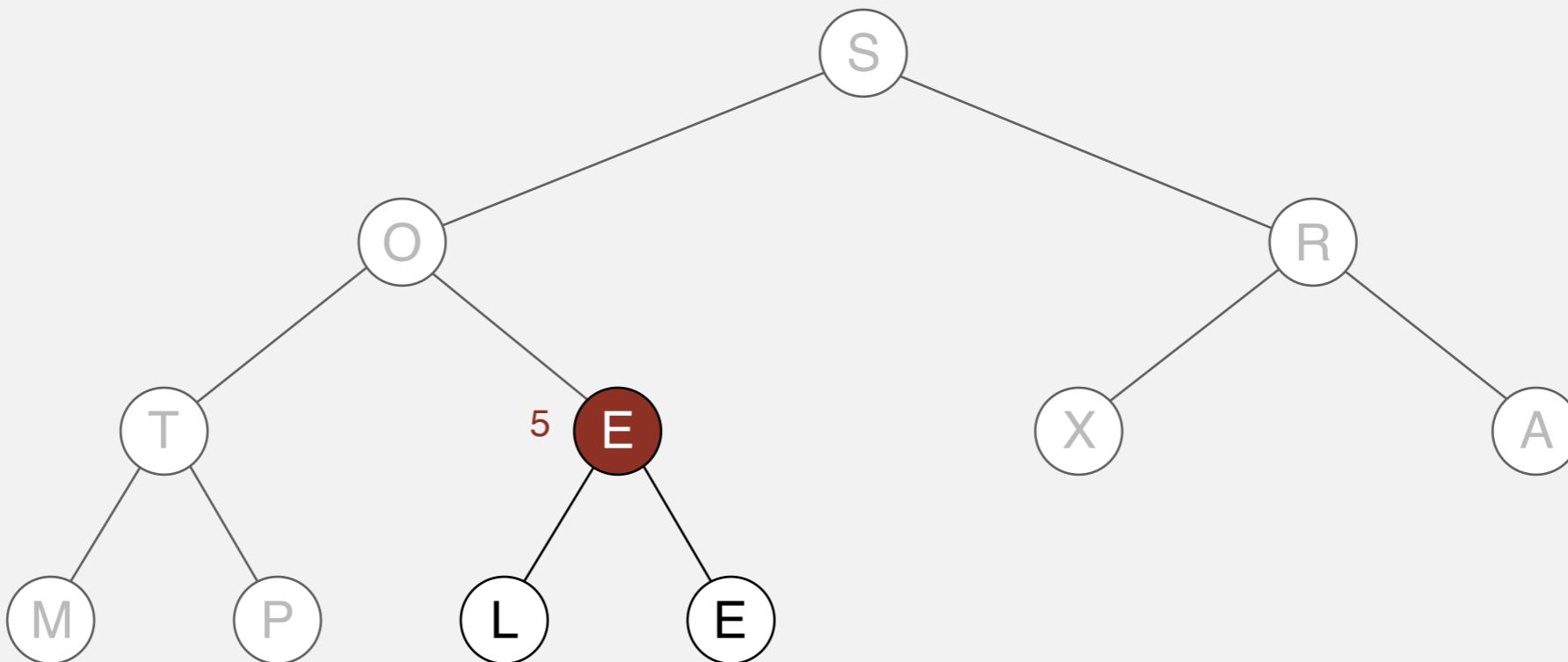
Heap construction. Build max heap using bottom-up method.



S	O	R	T	E	X	A	M	P	L	E
6	7	8	9	10	11					

Heap construction. Build max heap using bottom-up method.

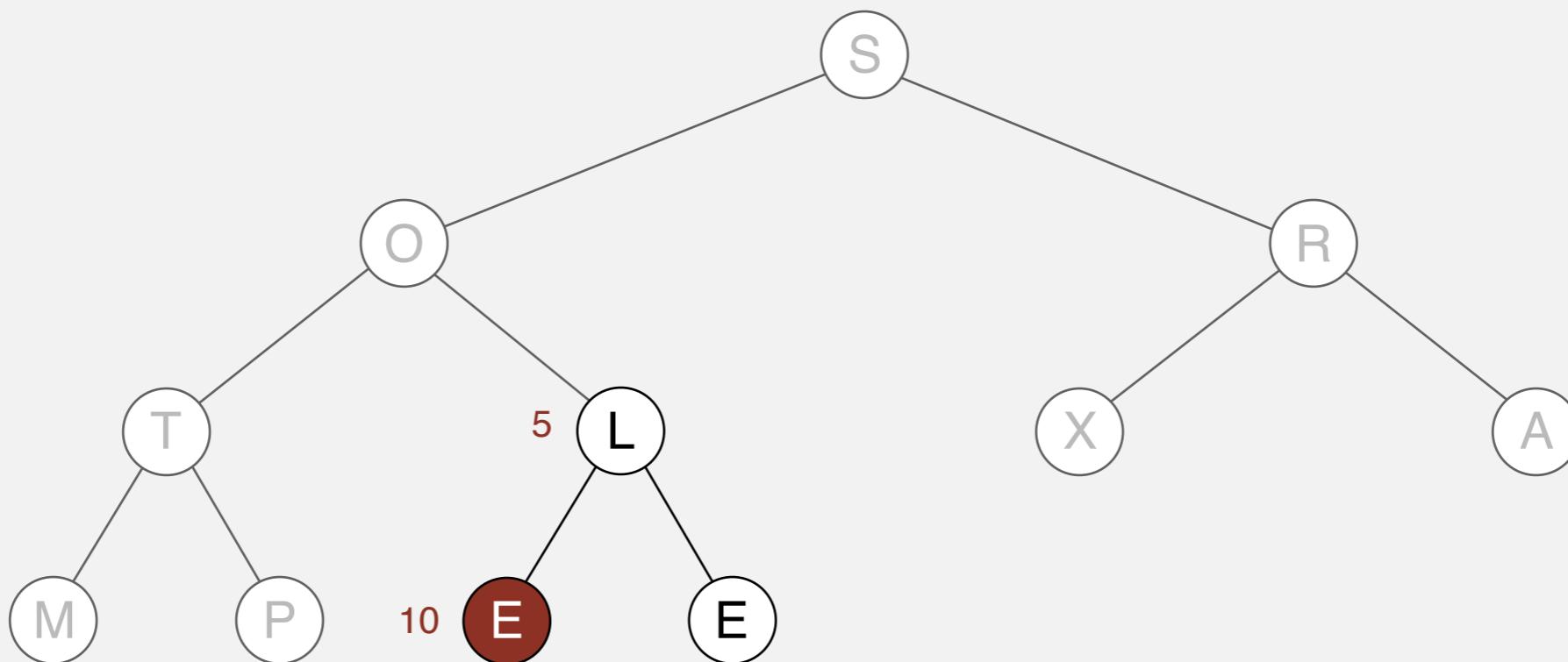
sink 5



S	O	R	T	E	X	A	M	P	L	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

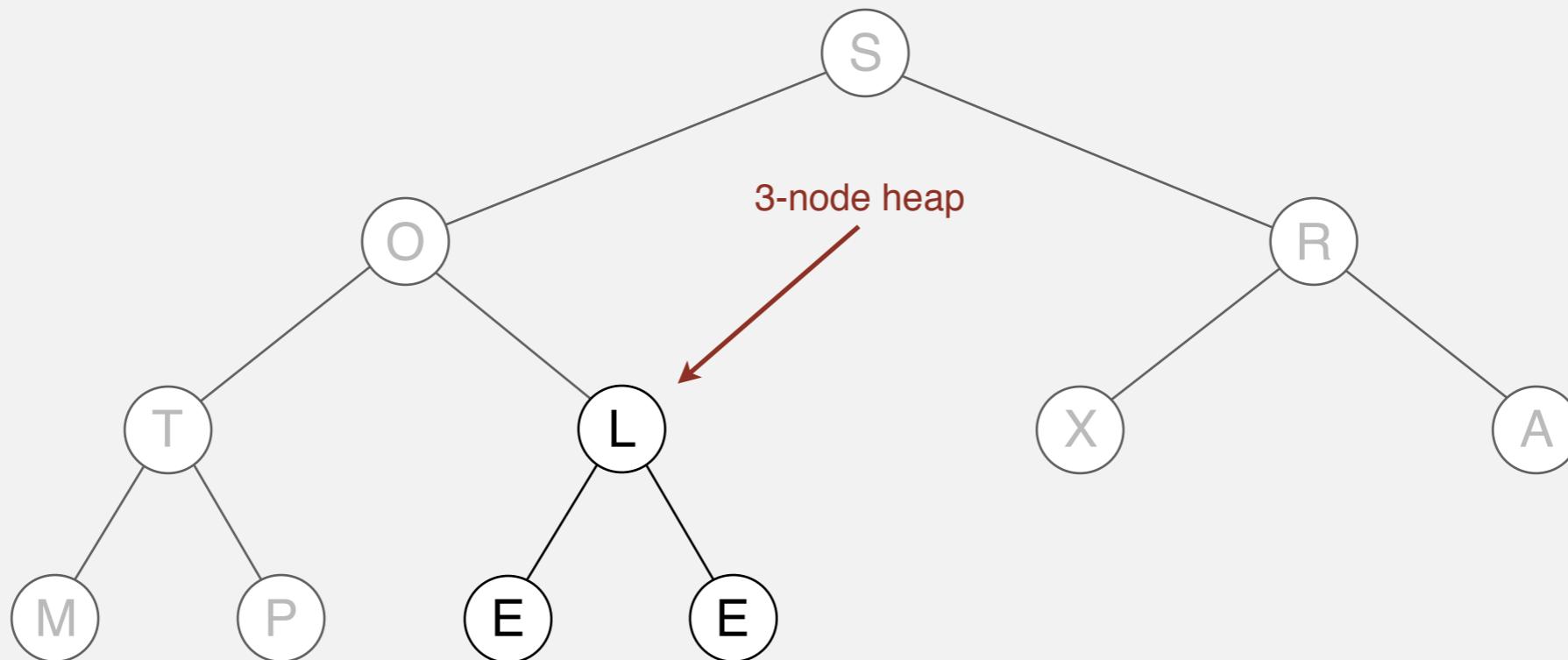
sink 5



S	O	R	T	L	X	A	M	P	E	E
5									10	

Heap construction. Build max heap using bottom-up method.

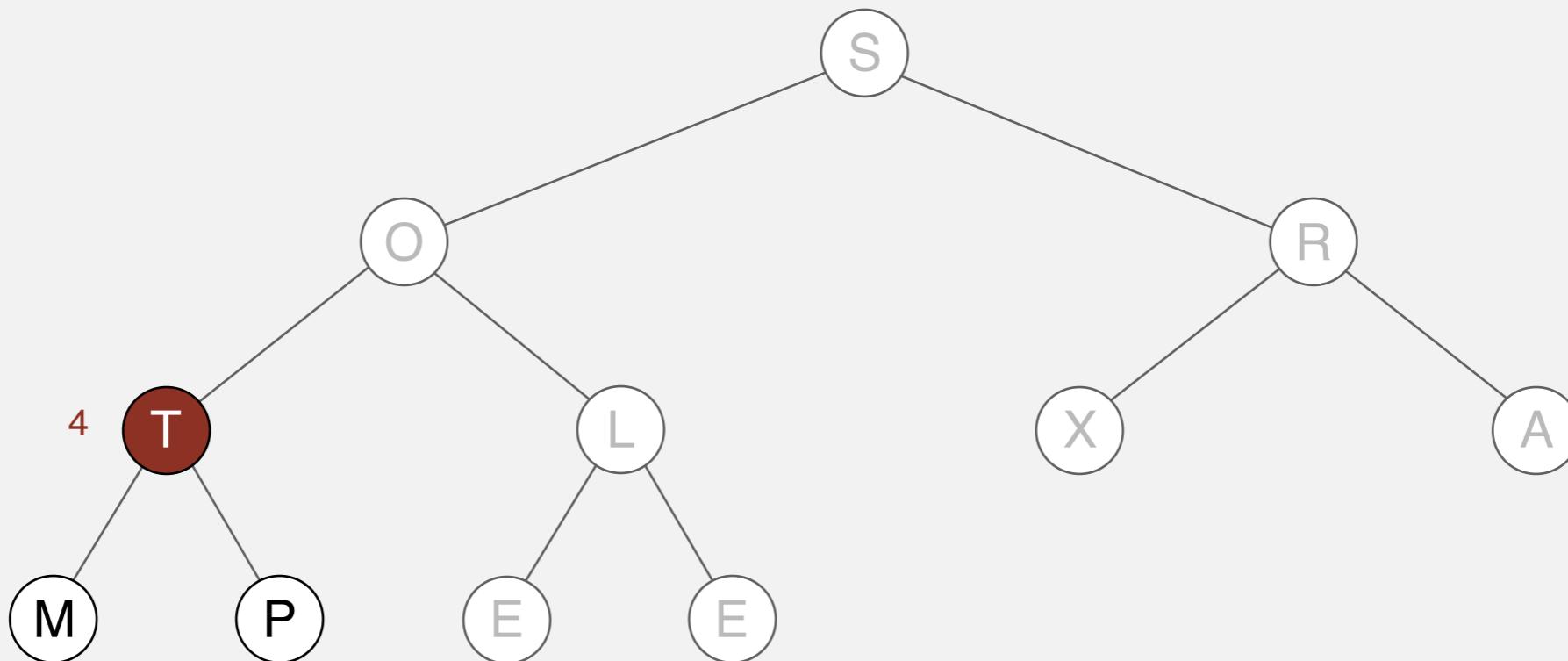
sink 5



S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

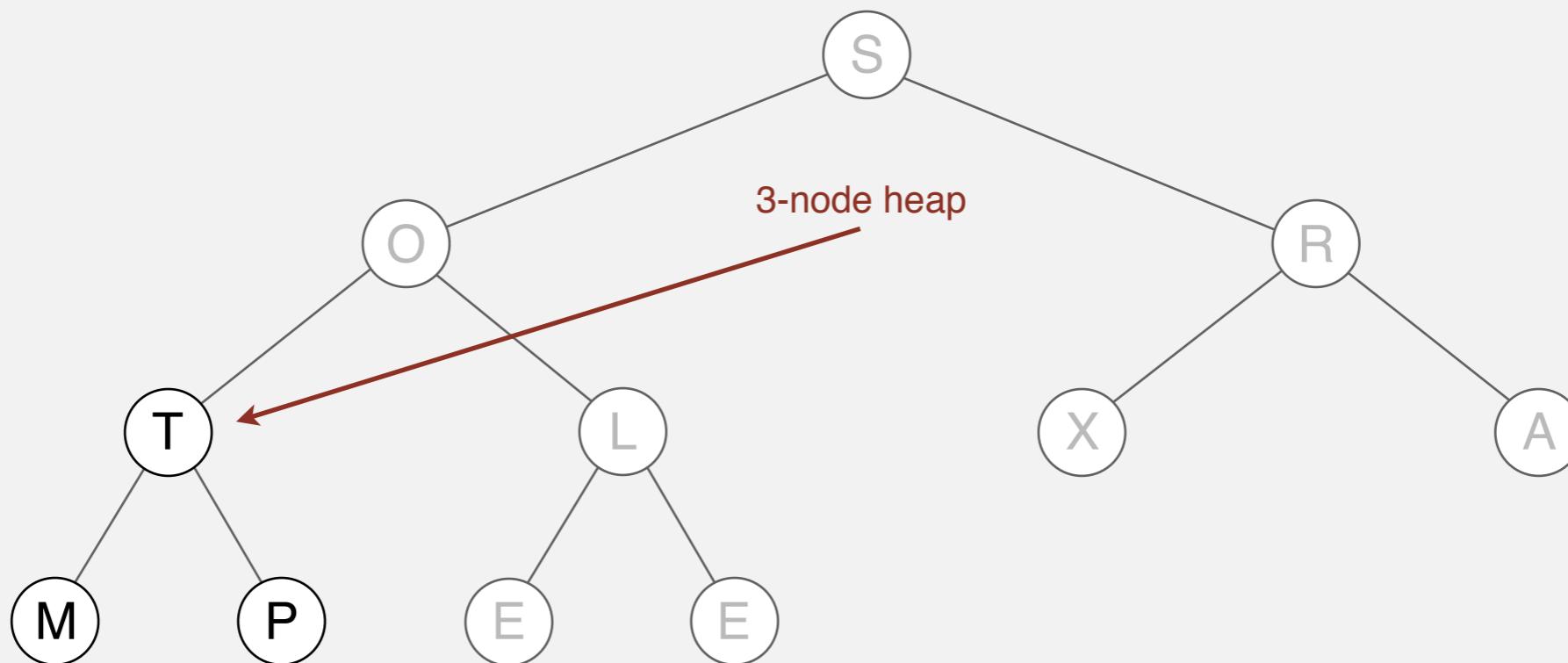
sink 4



S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

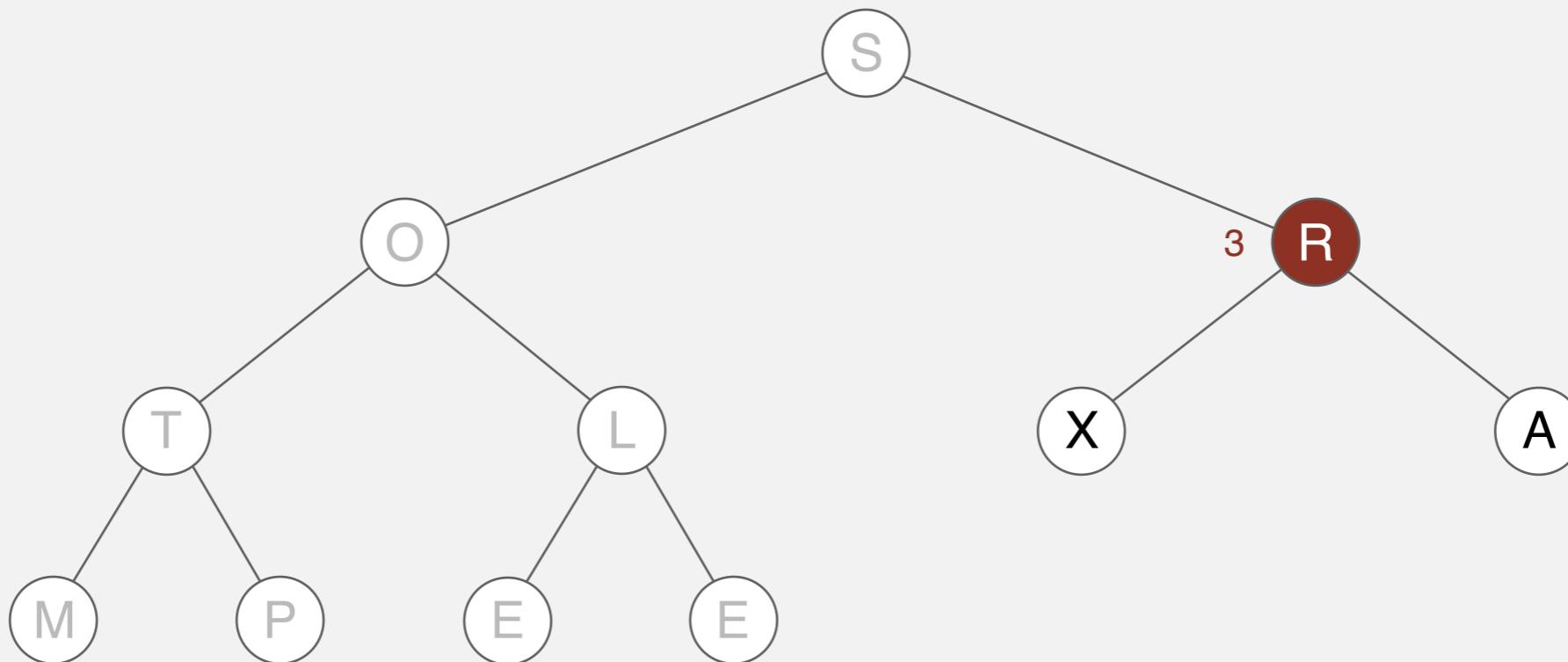
sink 4



S	O	R	T	L	X	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

sink 3

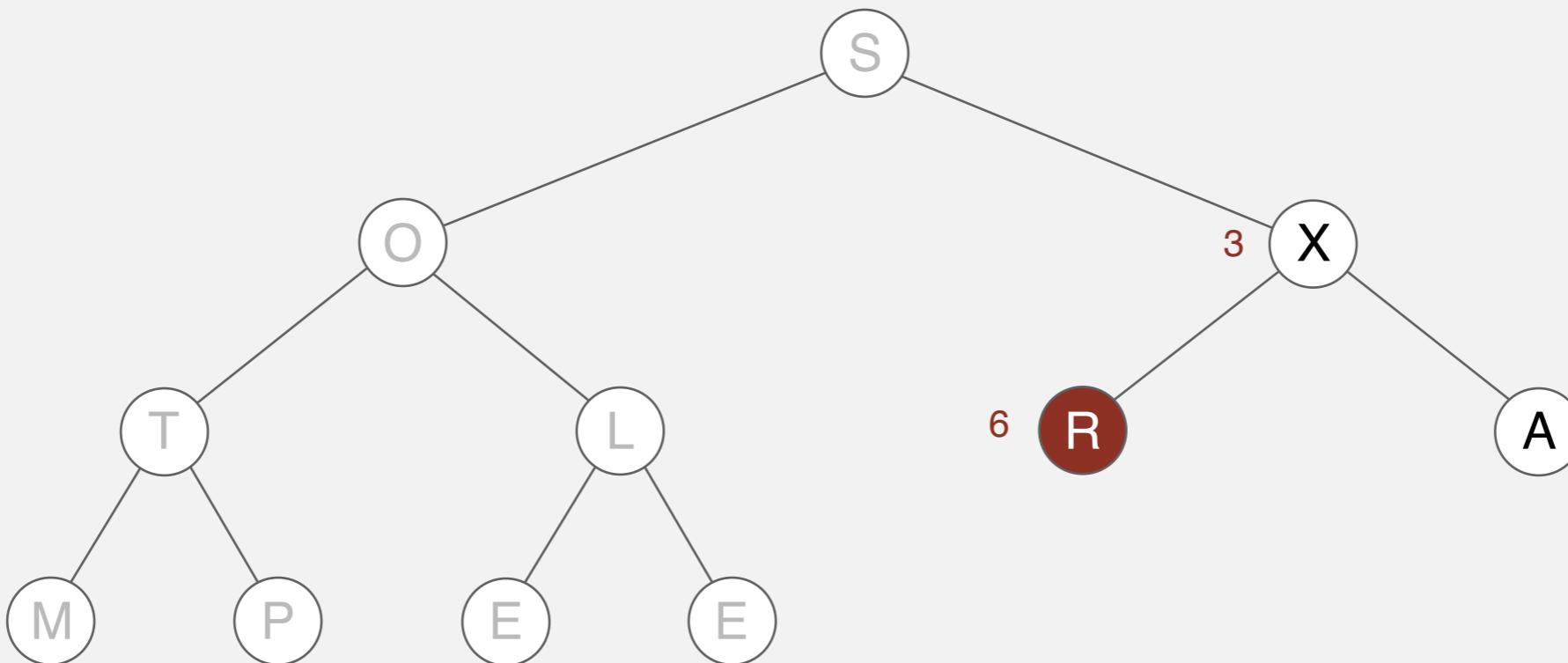


S	O	R	T	L	X	A	M	P	E	E
		3								

Heapsort

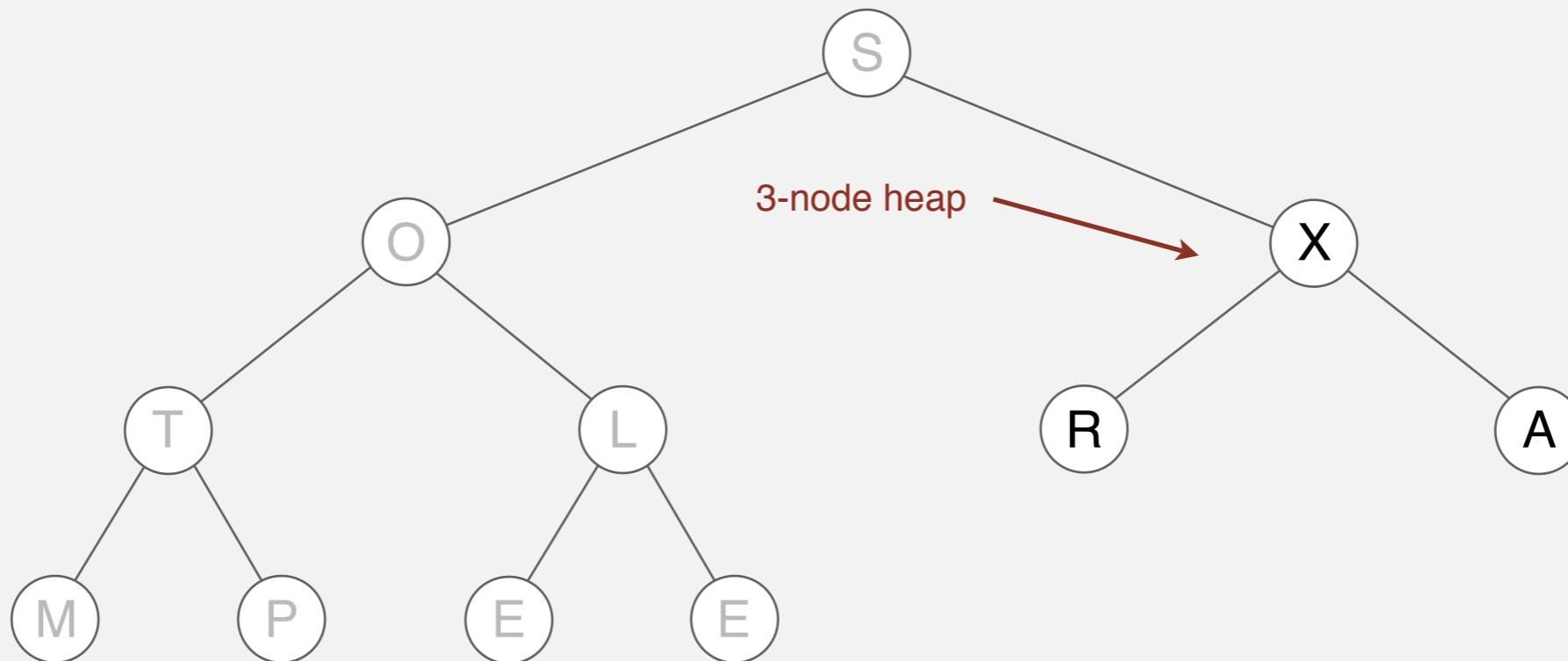
Heap construction. Build max heap using bottom-up method.

sink 3



Heap construction. Build max heap using bottom-up method.

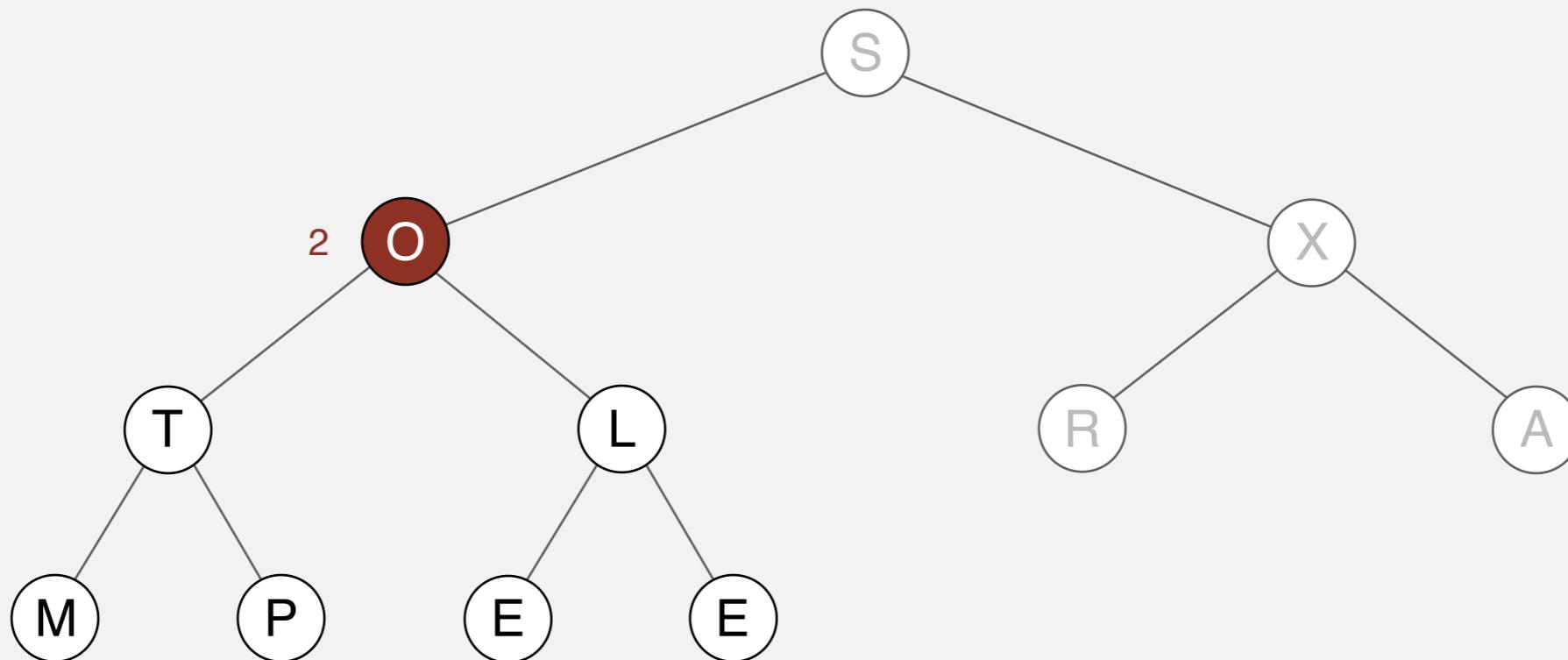
sink 3



S	O	X	T	L	A	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

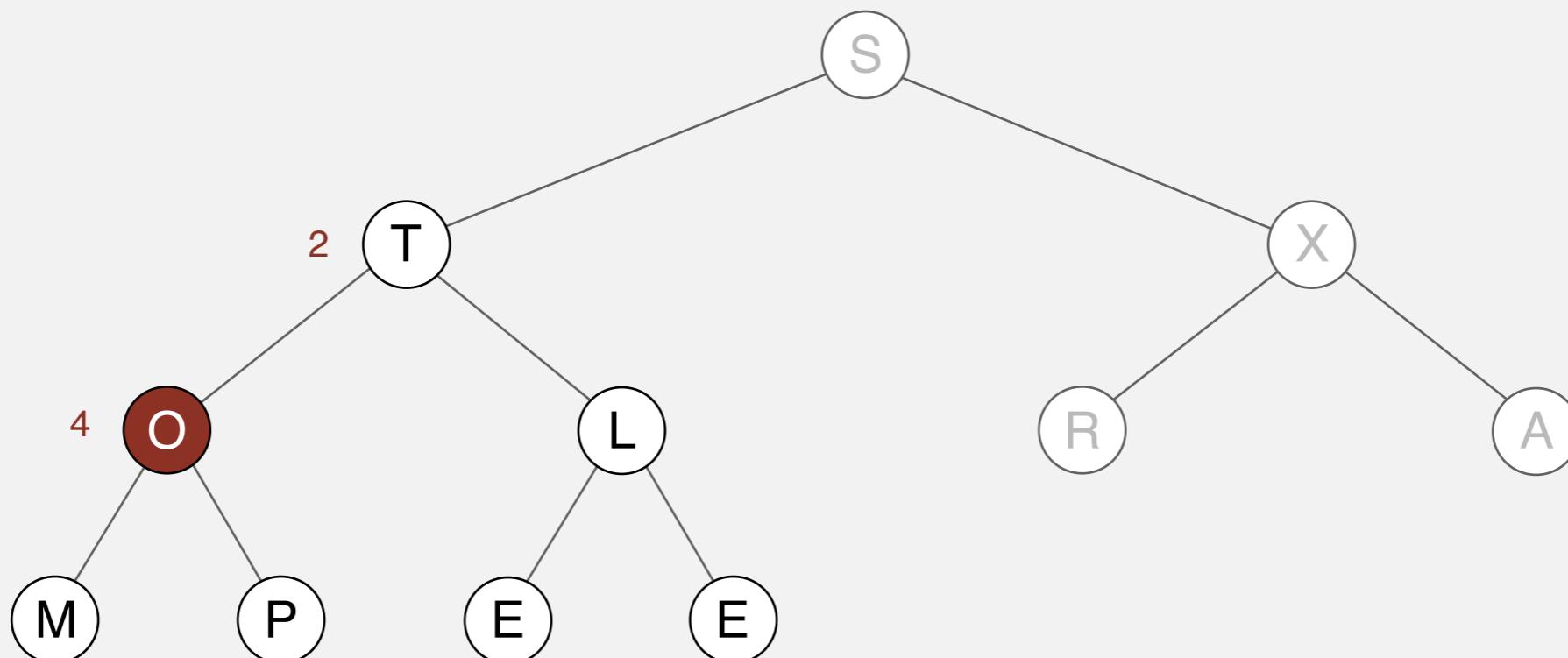
sink 2



S	O	X	T	L	R	A	M	P	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

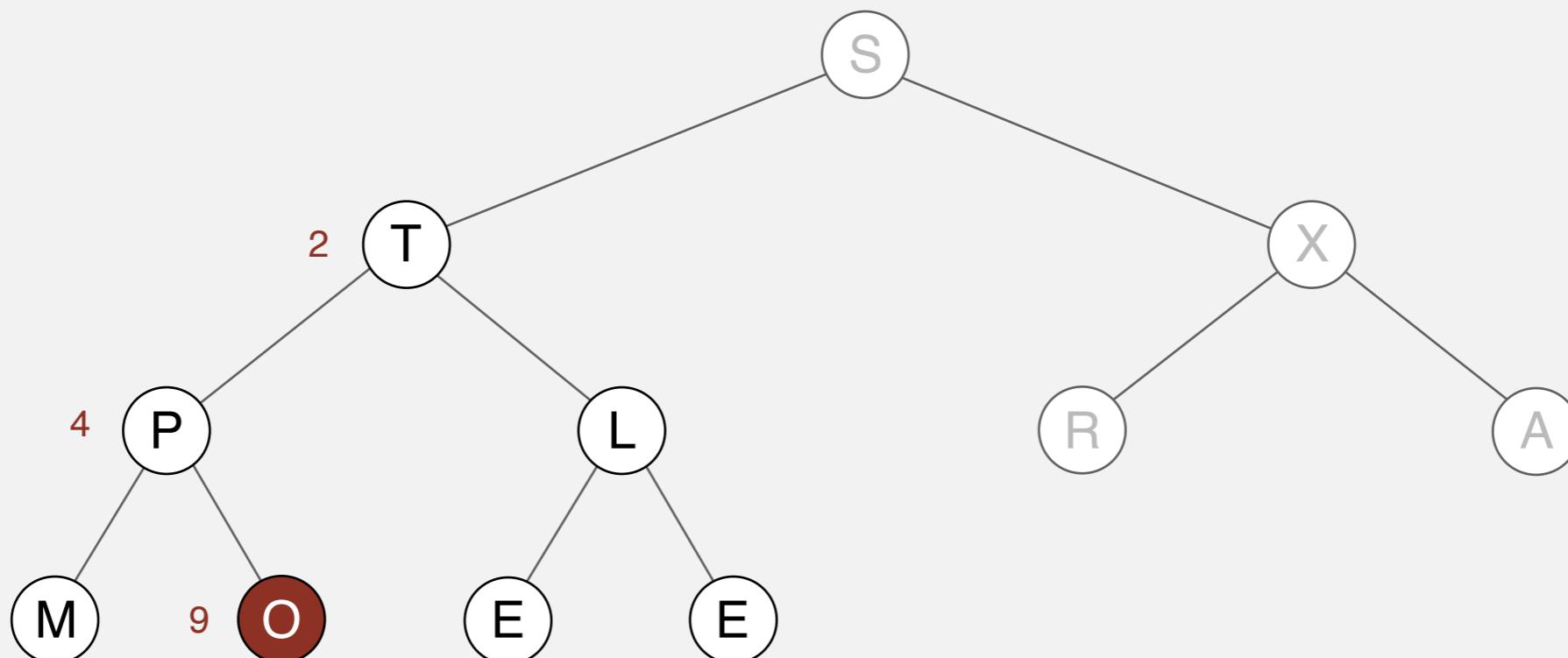
sink 2



S	T	X	O	L	R	A	M	P	E	E
2	4									

Heap construction. Build max heap using bottom-up method.

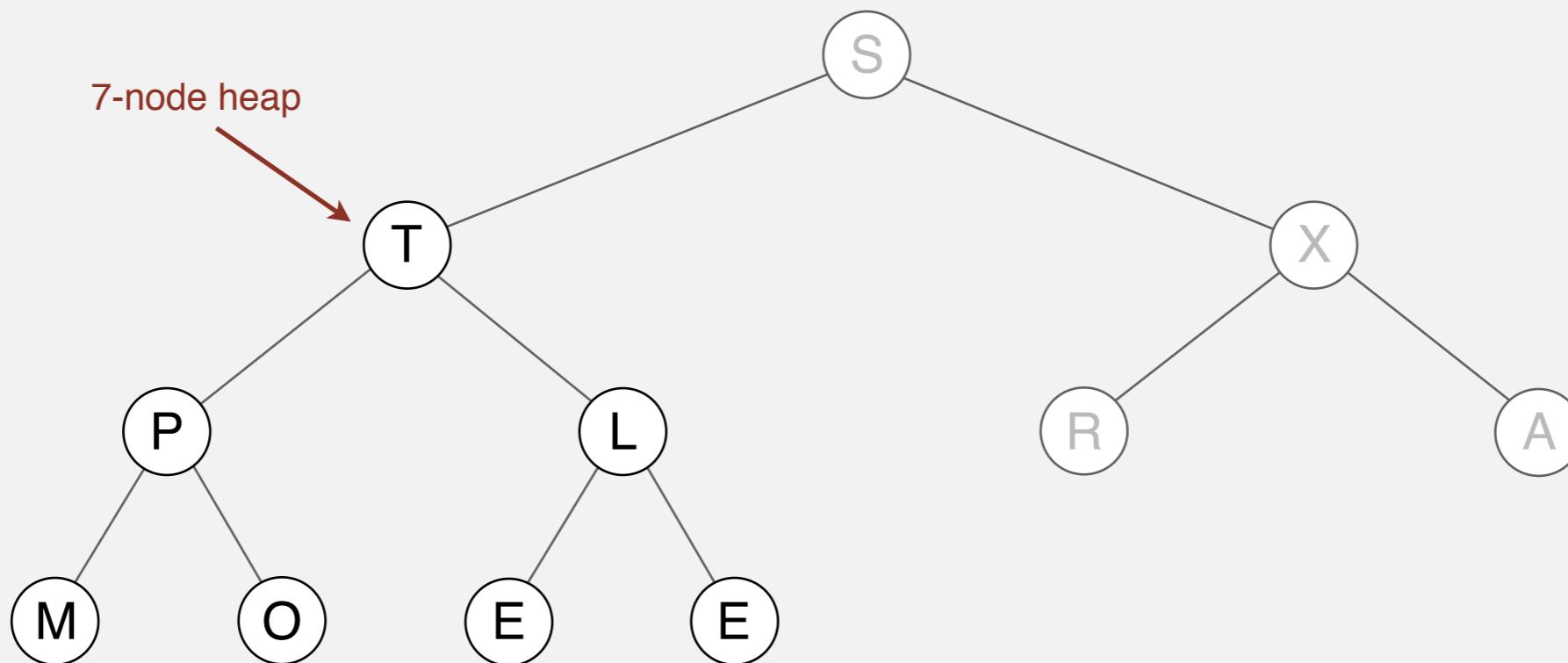
sink 2



S	T	X	P	L	R	A	M	O	E	E
2	4							9		

Heap construction. Build max heap using bottom-up method.

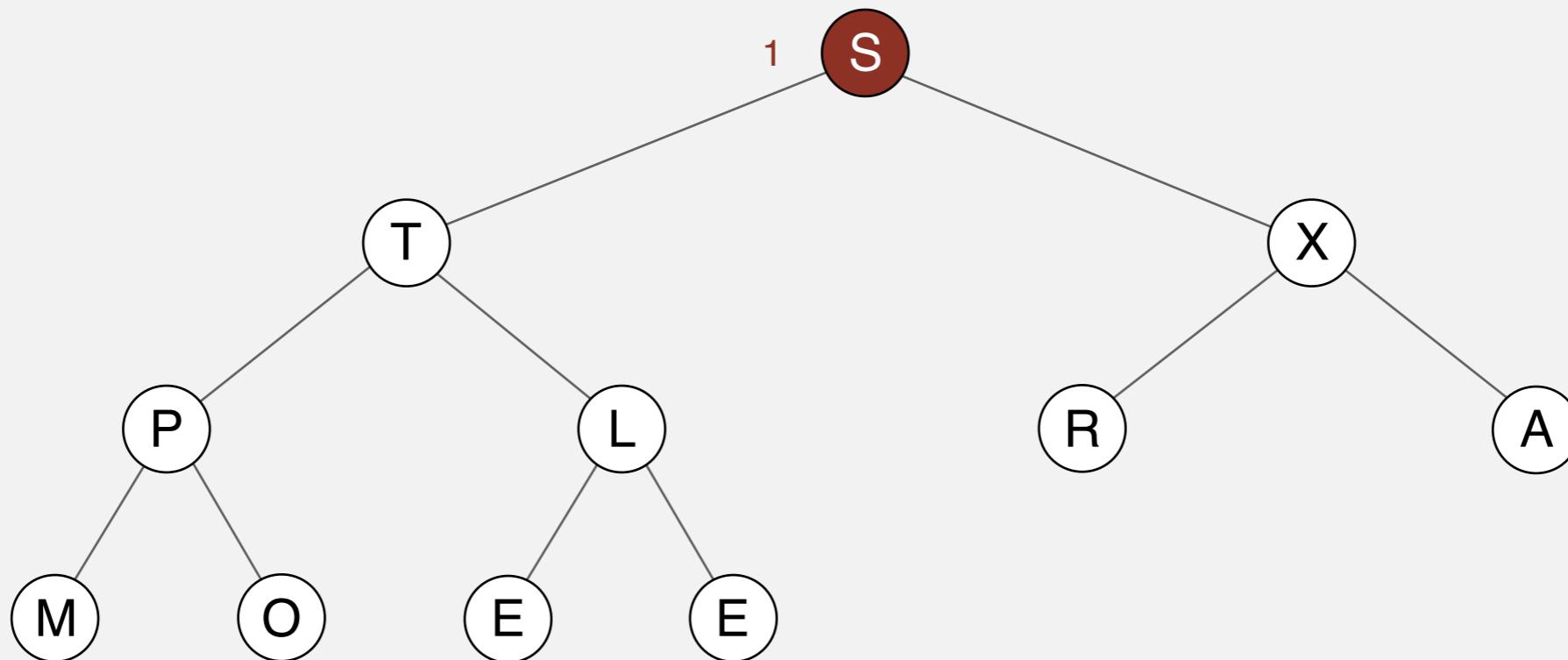
sink 2



S	T	X	P	L	R	A	M	O	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

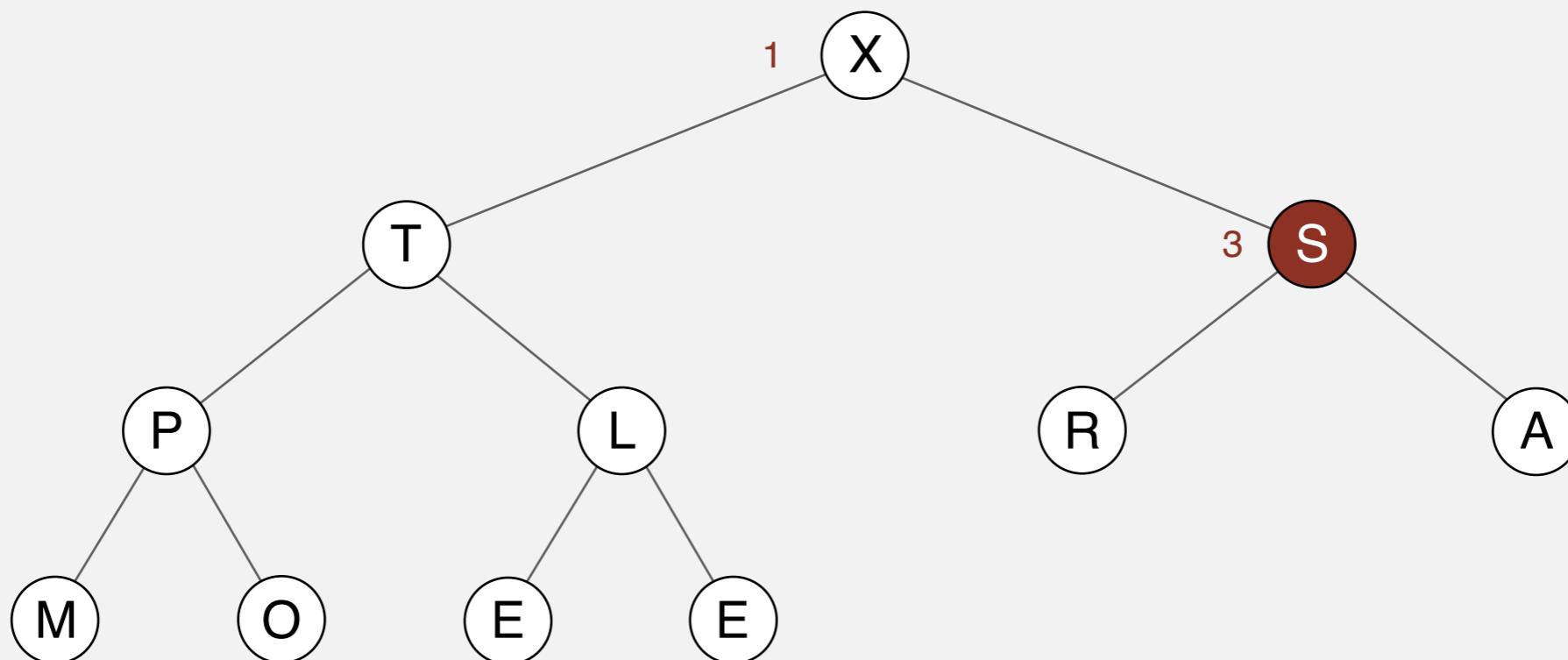
sink 1



S	T	X	P	L	R	A	M	O	E	E
---	---	---	---	---	---	---	---	---	---	---

Heap construction. Build max heap using bottom-up method.

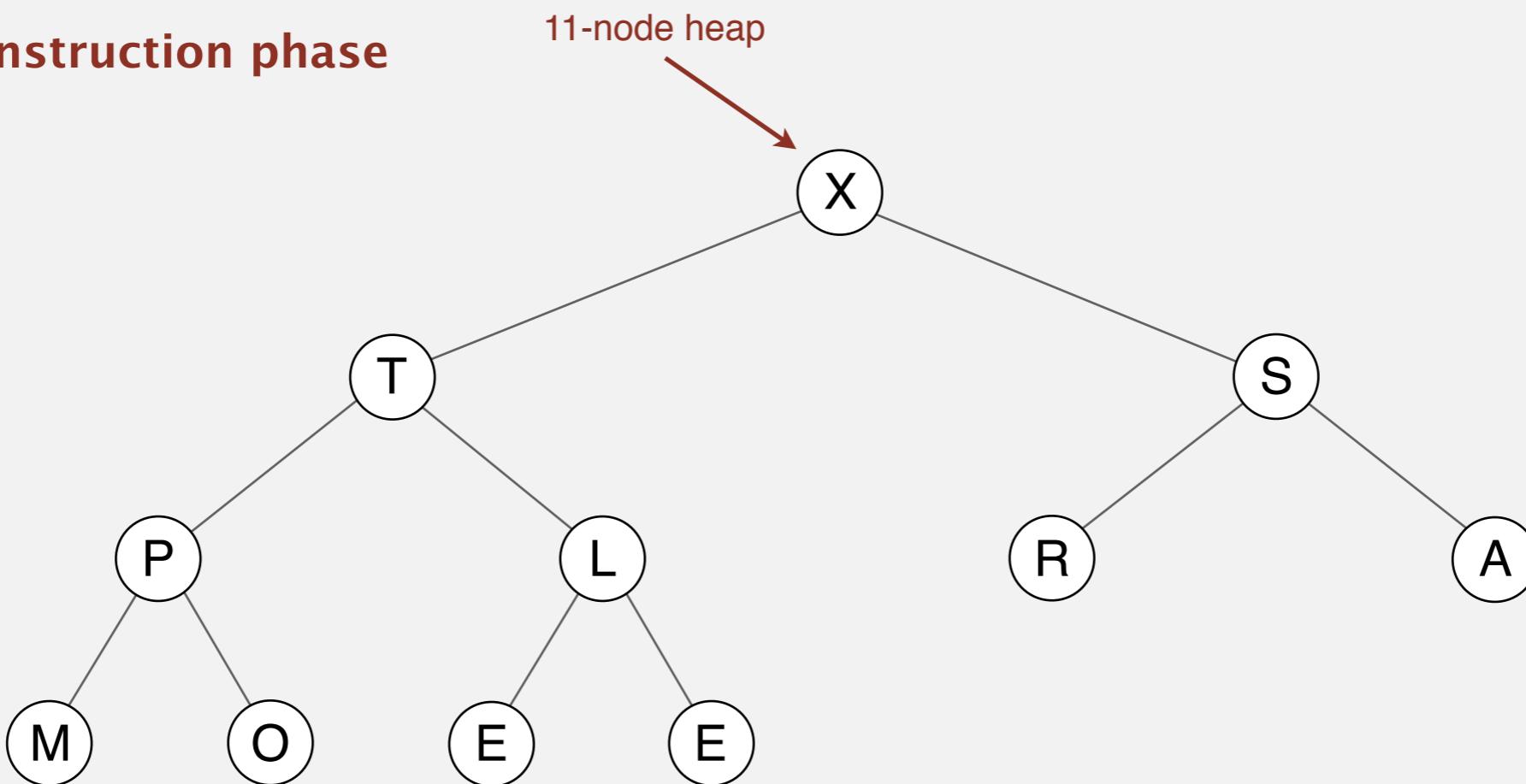
sink 1



X	T	S	P	L	R	A	M	O	E	E
1		3								

Heap construction. Build max heap using bottom-up method.

end of construction phase

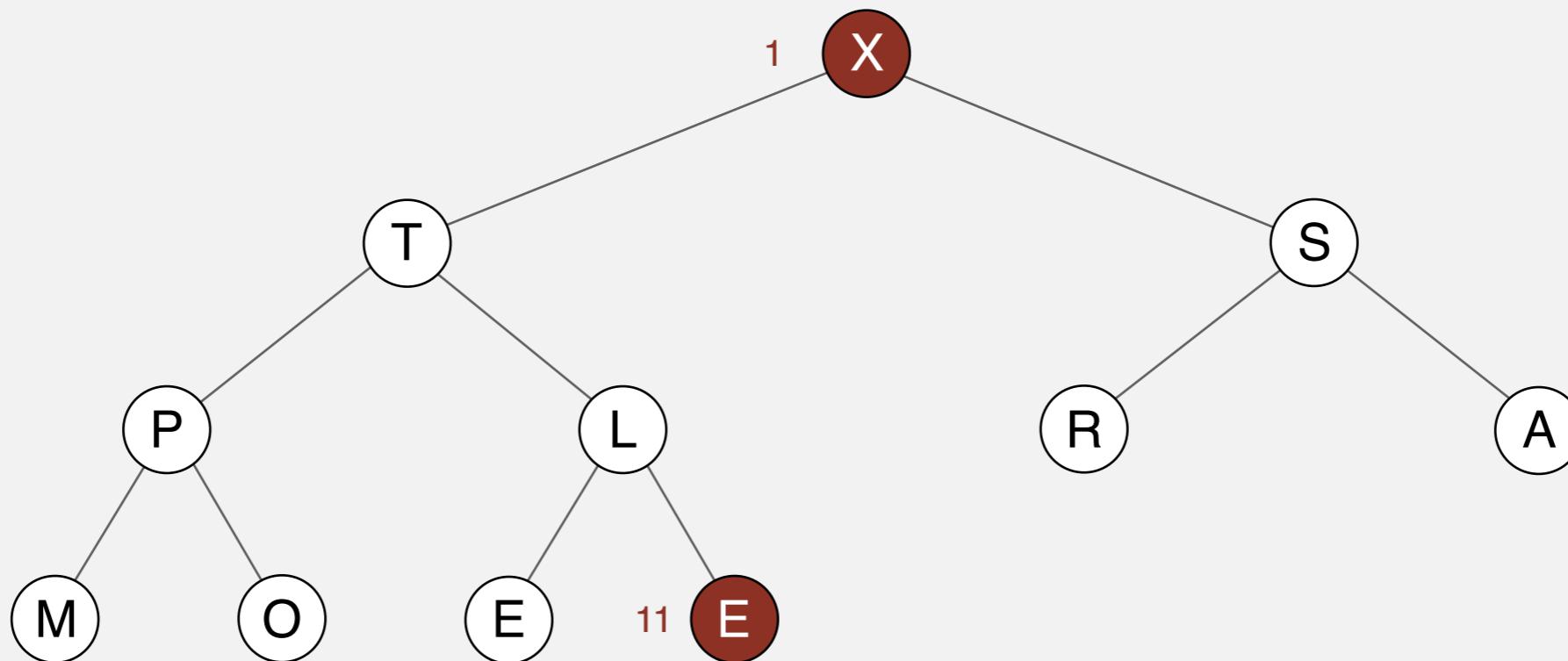


X	T	S	P	L	R	A	M	O	E	E
---	---	---	---	---	---	---	---	---	---	---

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

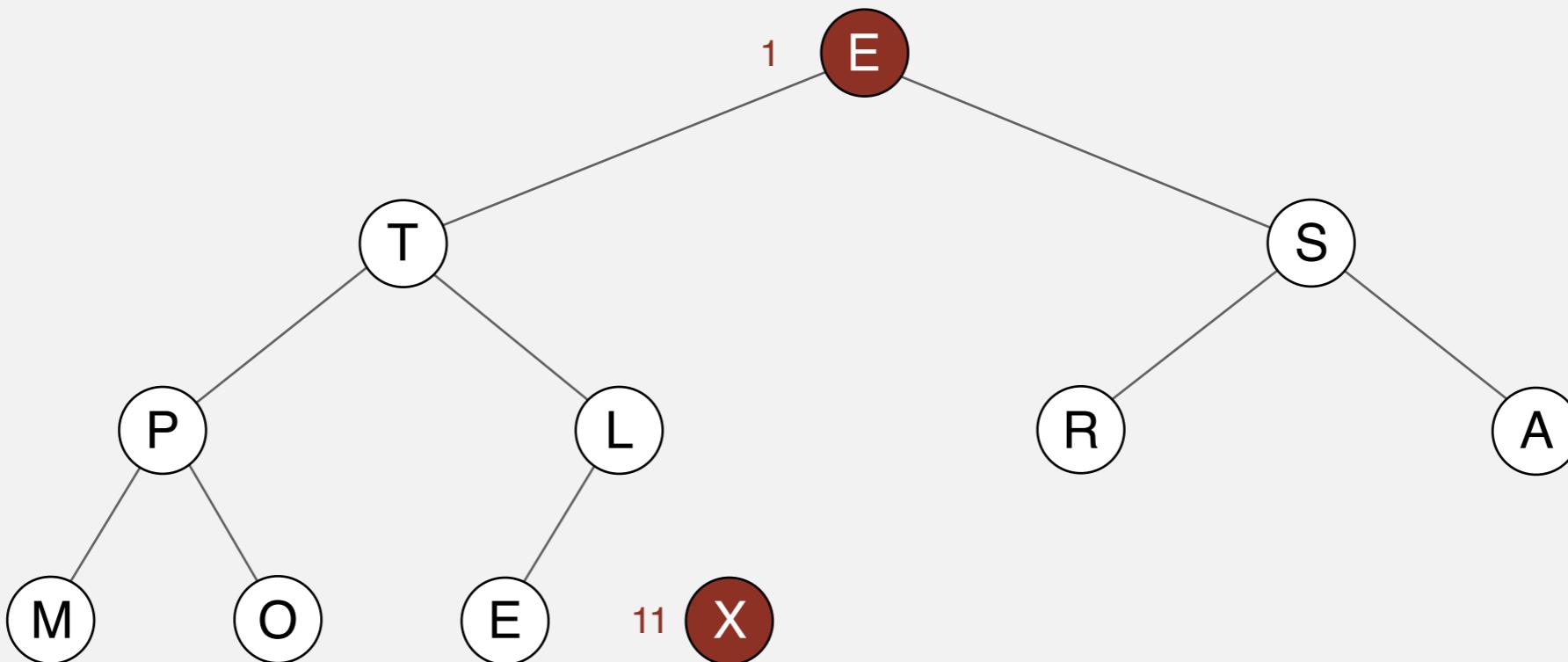


X	T	S	P	L	R	A	M	O	E	E
1										11

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

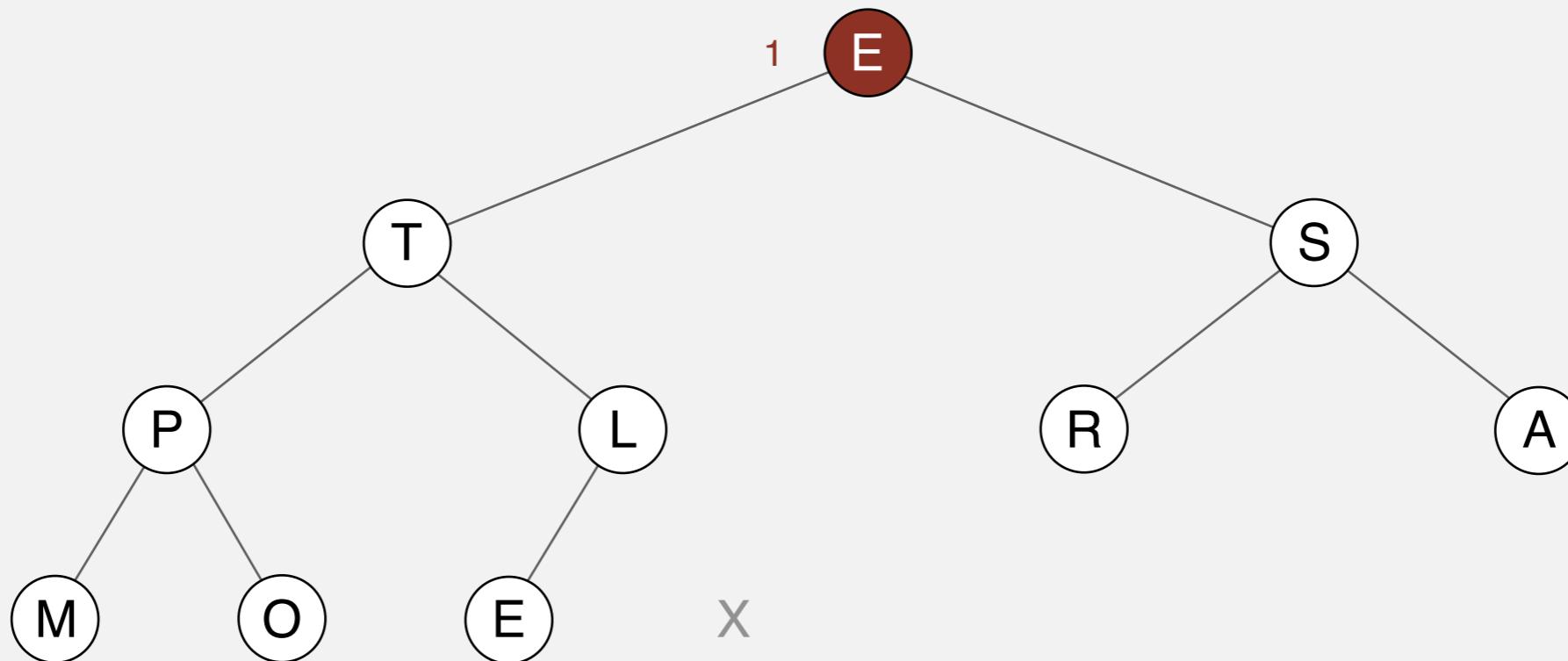


E	T	S	P	L	R	A	M	O	E	X
1										11

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1



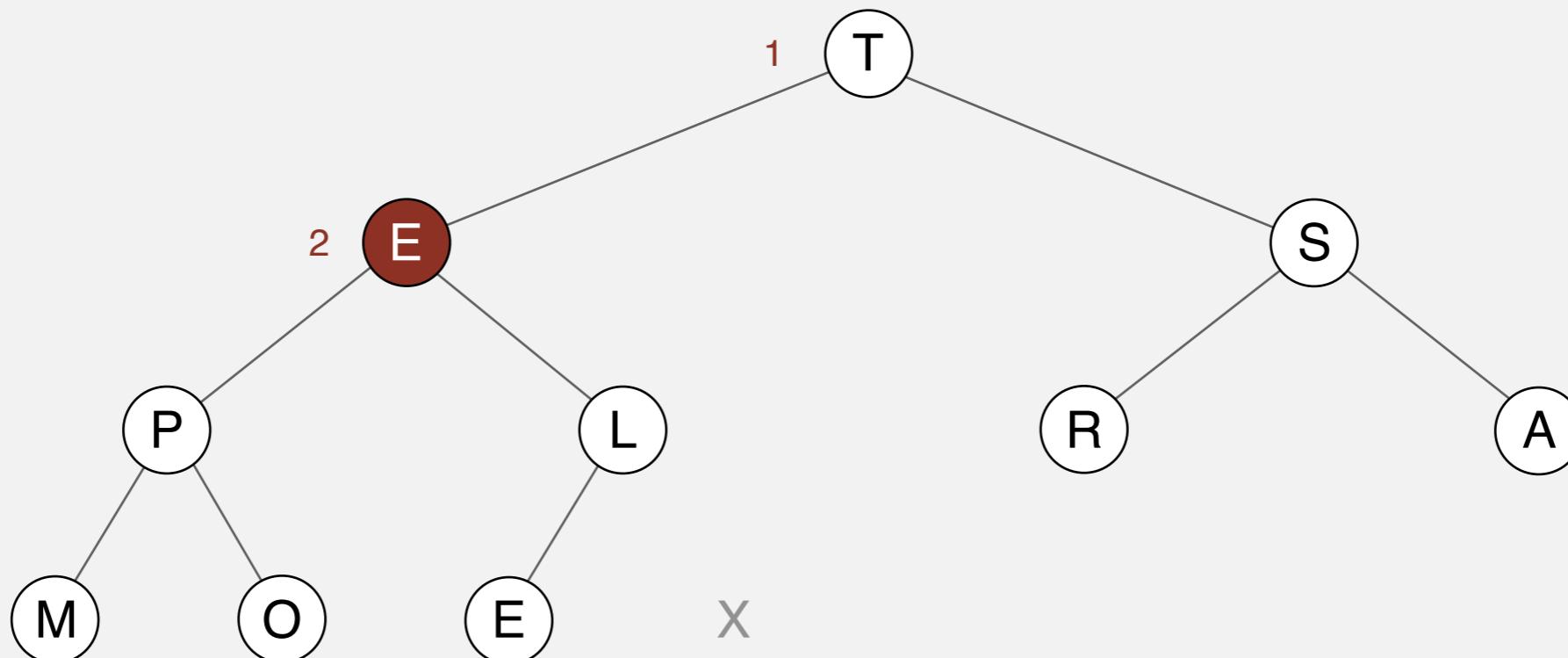
E	T	S	P	L	R	A	M	O	E	X
---	---	---	---	---	---	---	---	---	---	---

1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

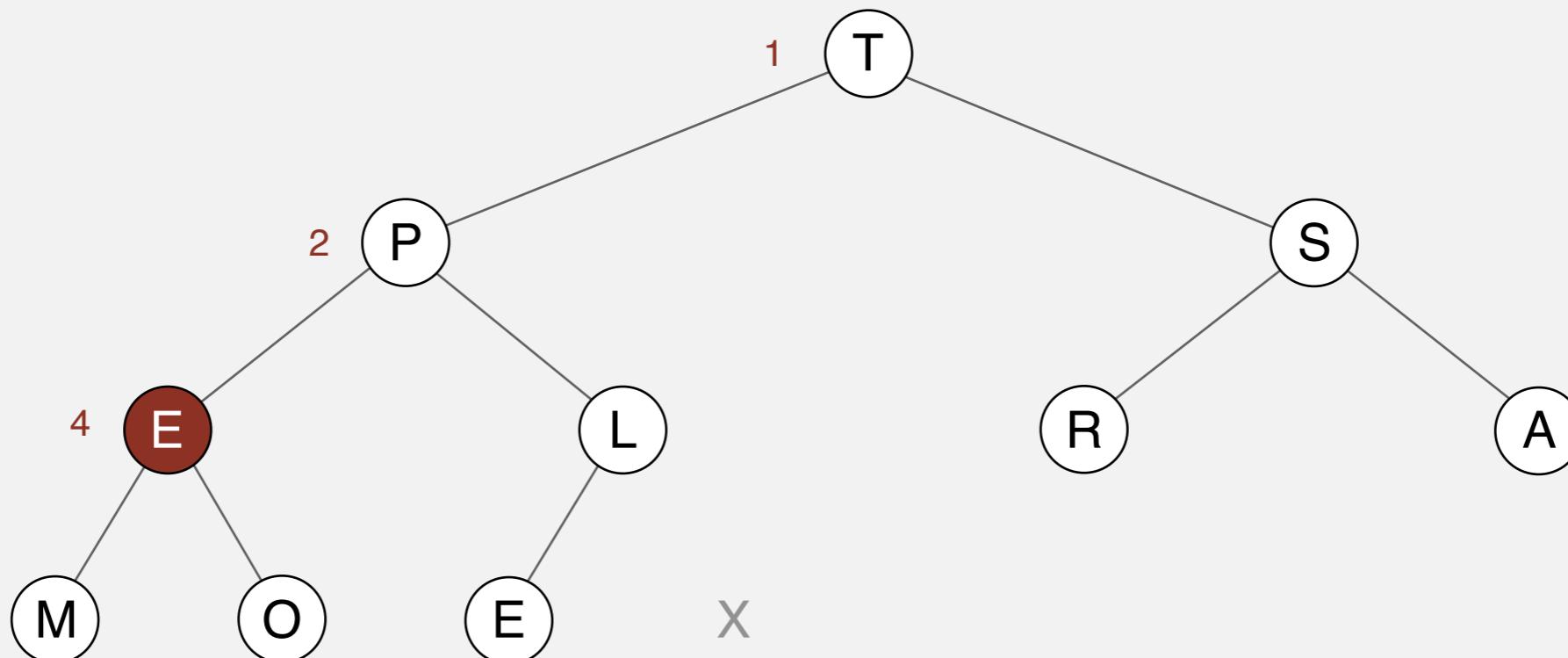


T	E	S	P	L	R	A	M	O	E	X
1	2									

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

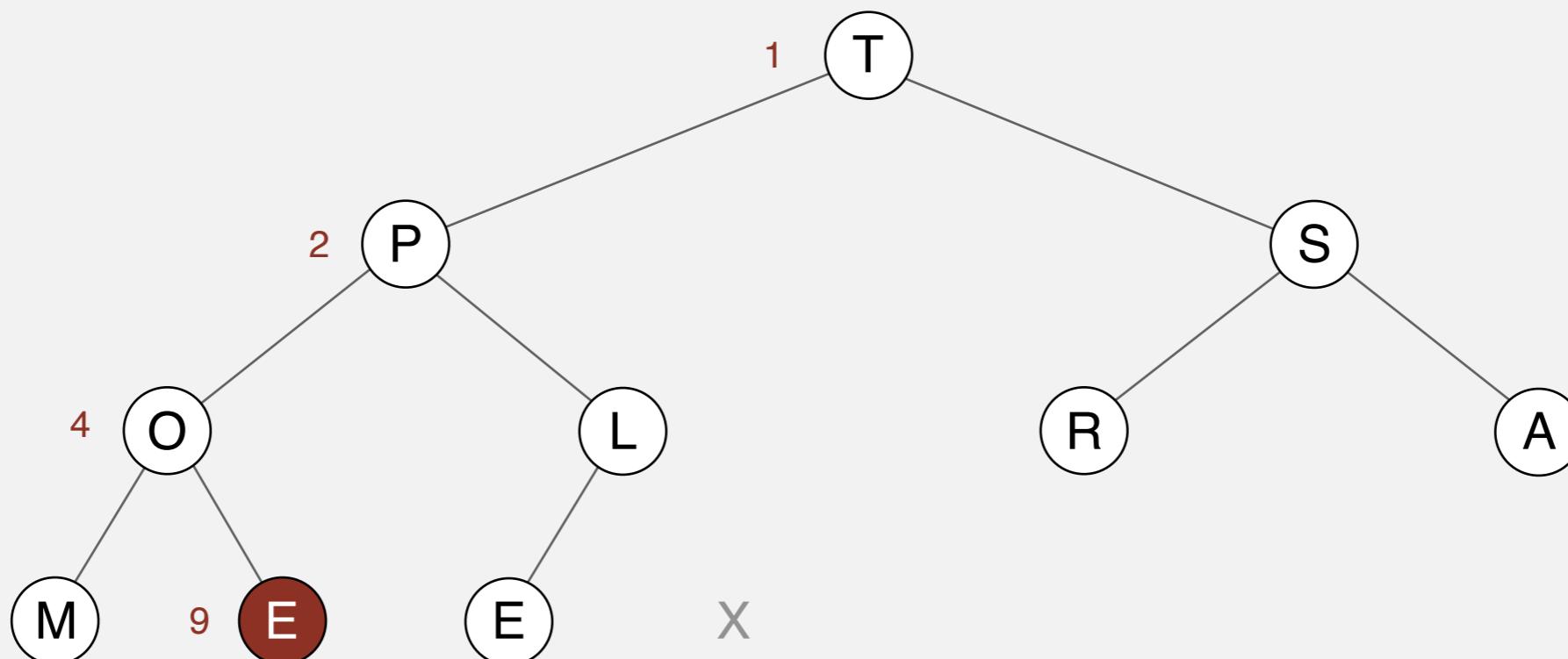


T	P	S	E	L	R	A	M	O	E	X
1	2	4								

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

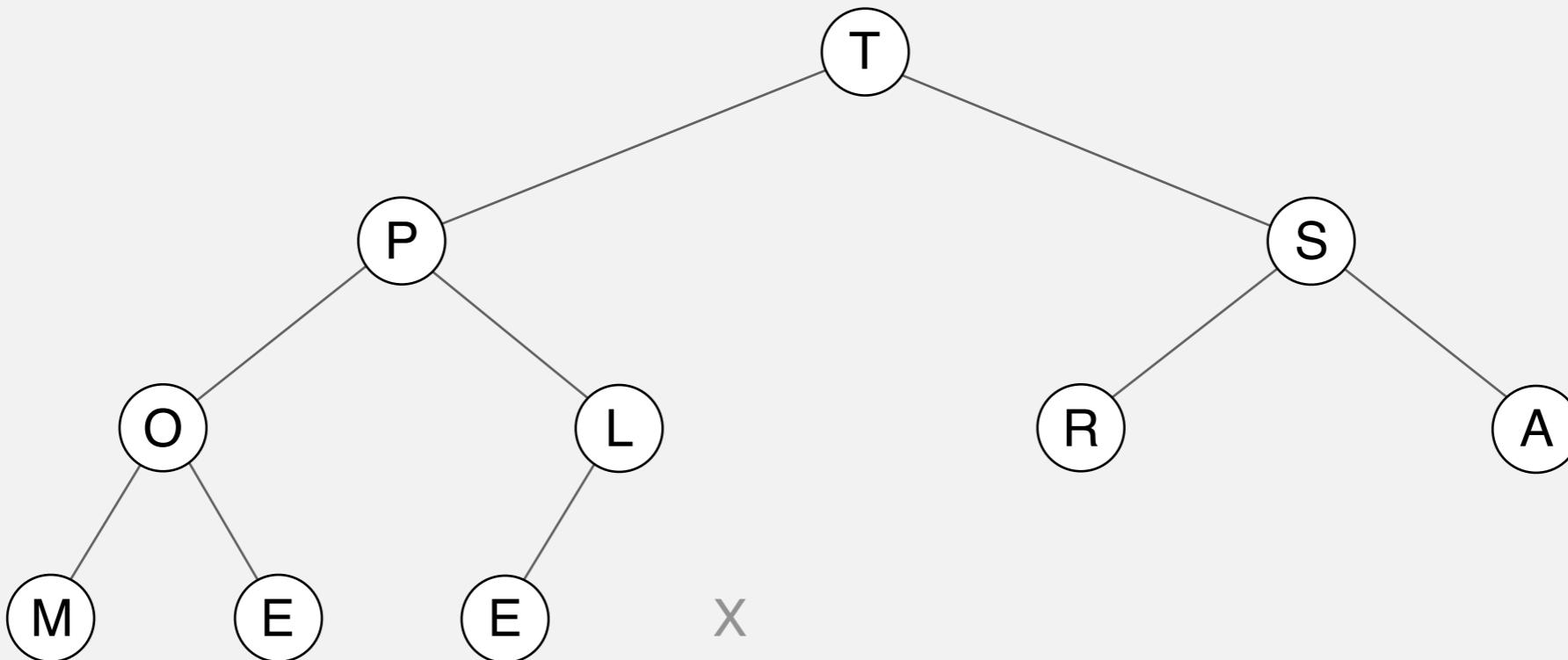
sink 1



T	P	S	O	L	R	A	M	E	E	X
1	2		4					9		

Heapsort

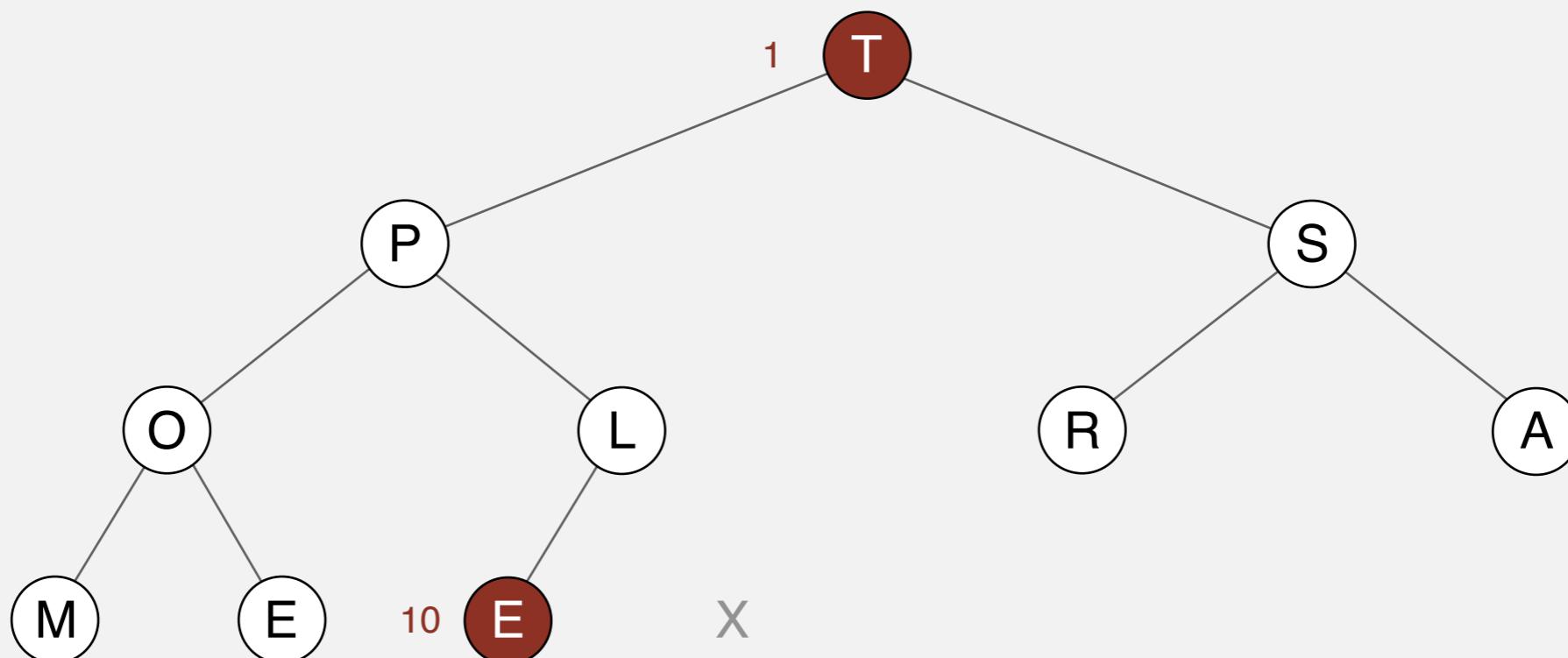
Sortdown. Repeatedly delete the largest remaining item.



T | P | S | O | L | R | A | M | E | E | X

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10

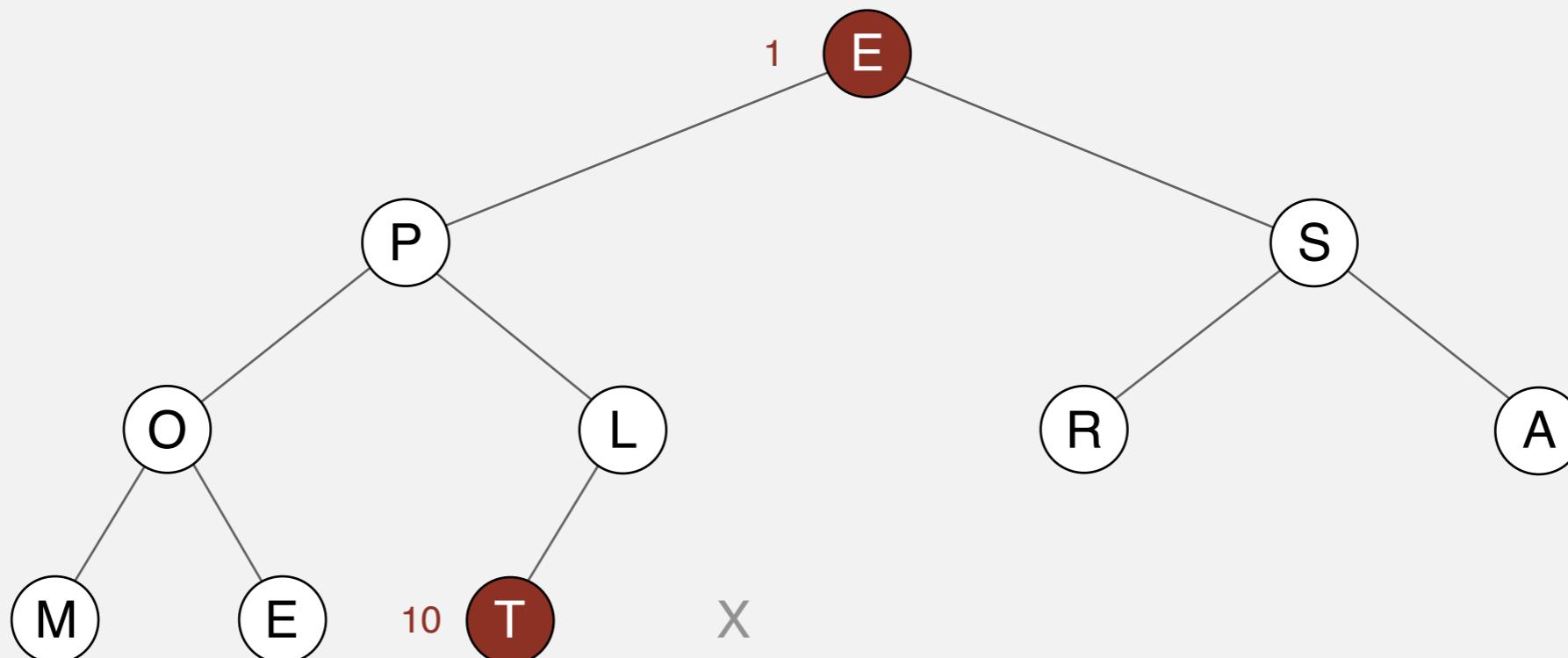


T	P	S	O	L	R	A	M	E	E	X
1									10	

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

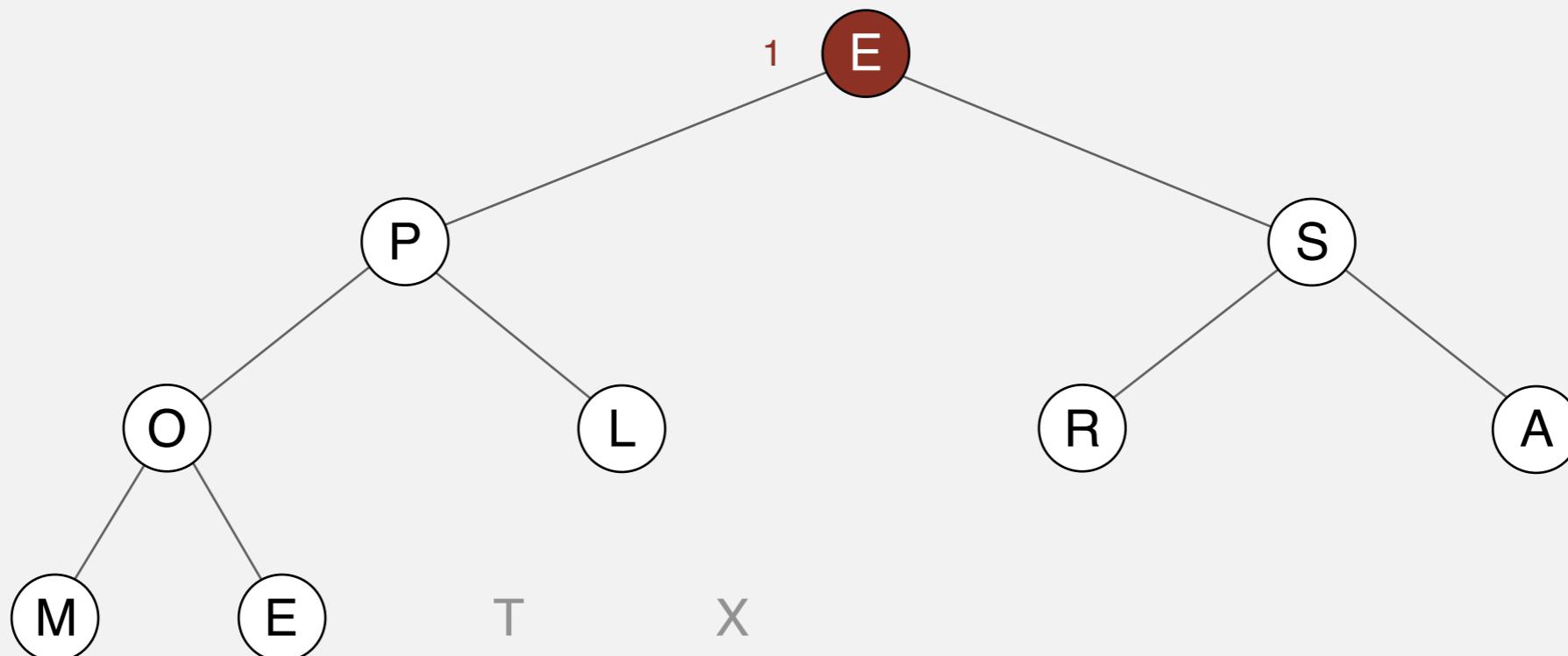
exchange 1 and 10



E	P	S	O	L	R	A	M	E	T	X
1									10	

Sortdown. Repeatedly delete the largest remaining item.

sink 1

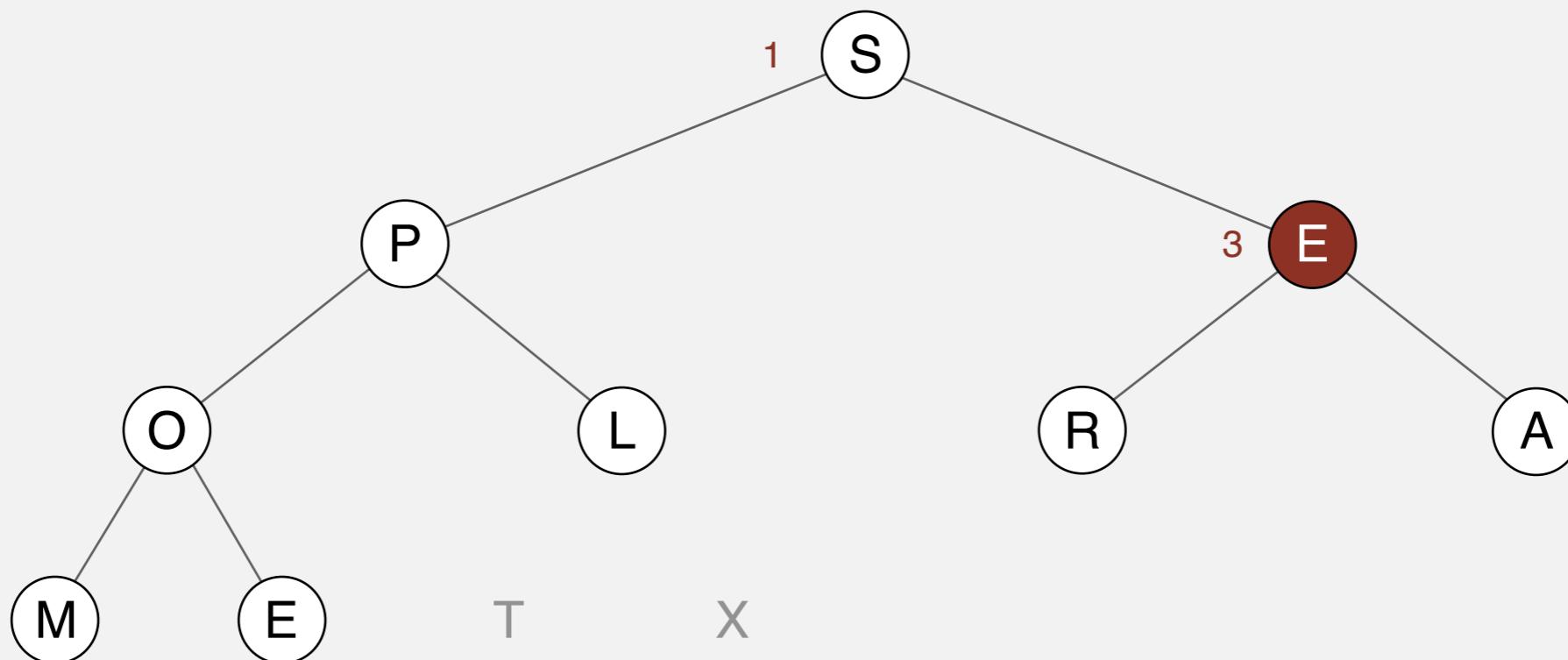


E	P	S	O	L	R	A	M	E	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

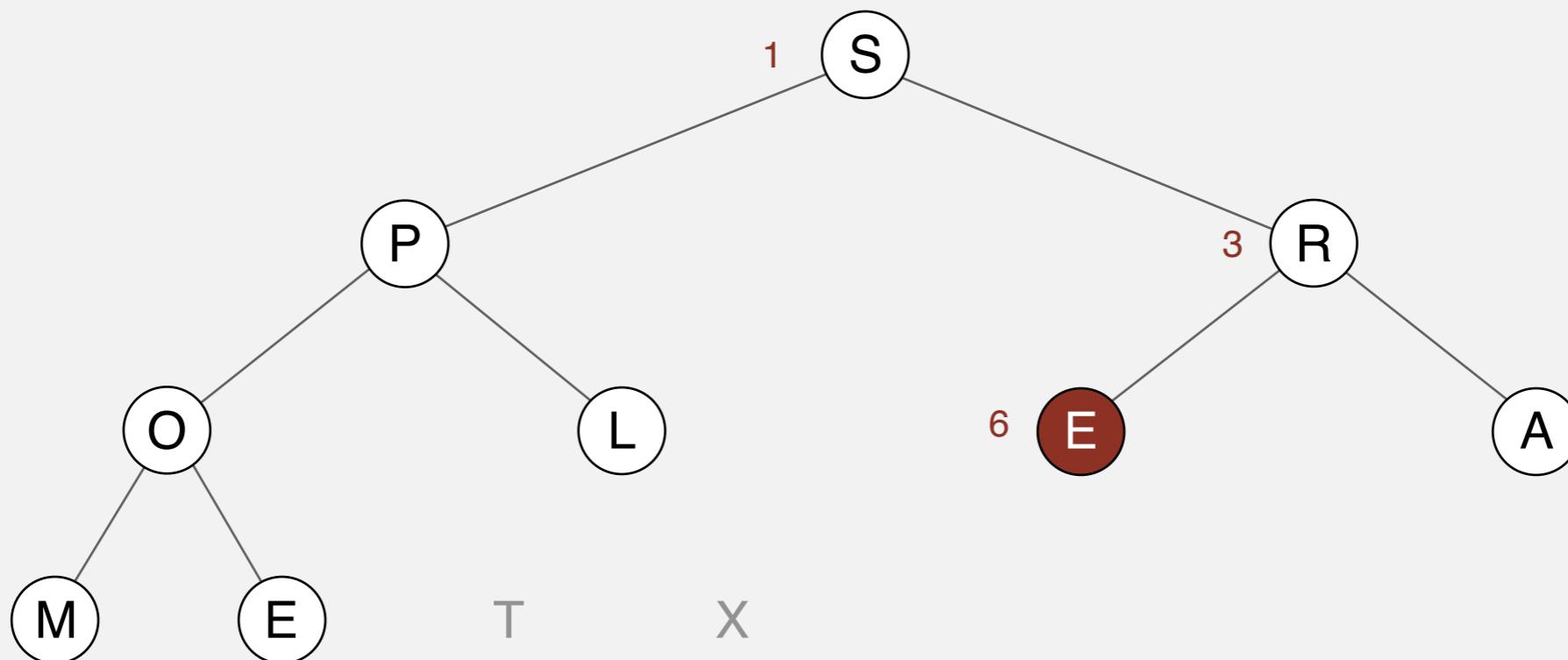


S	P	E	O	L	R	A	M	E	T	X
1		3								

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

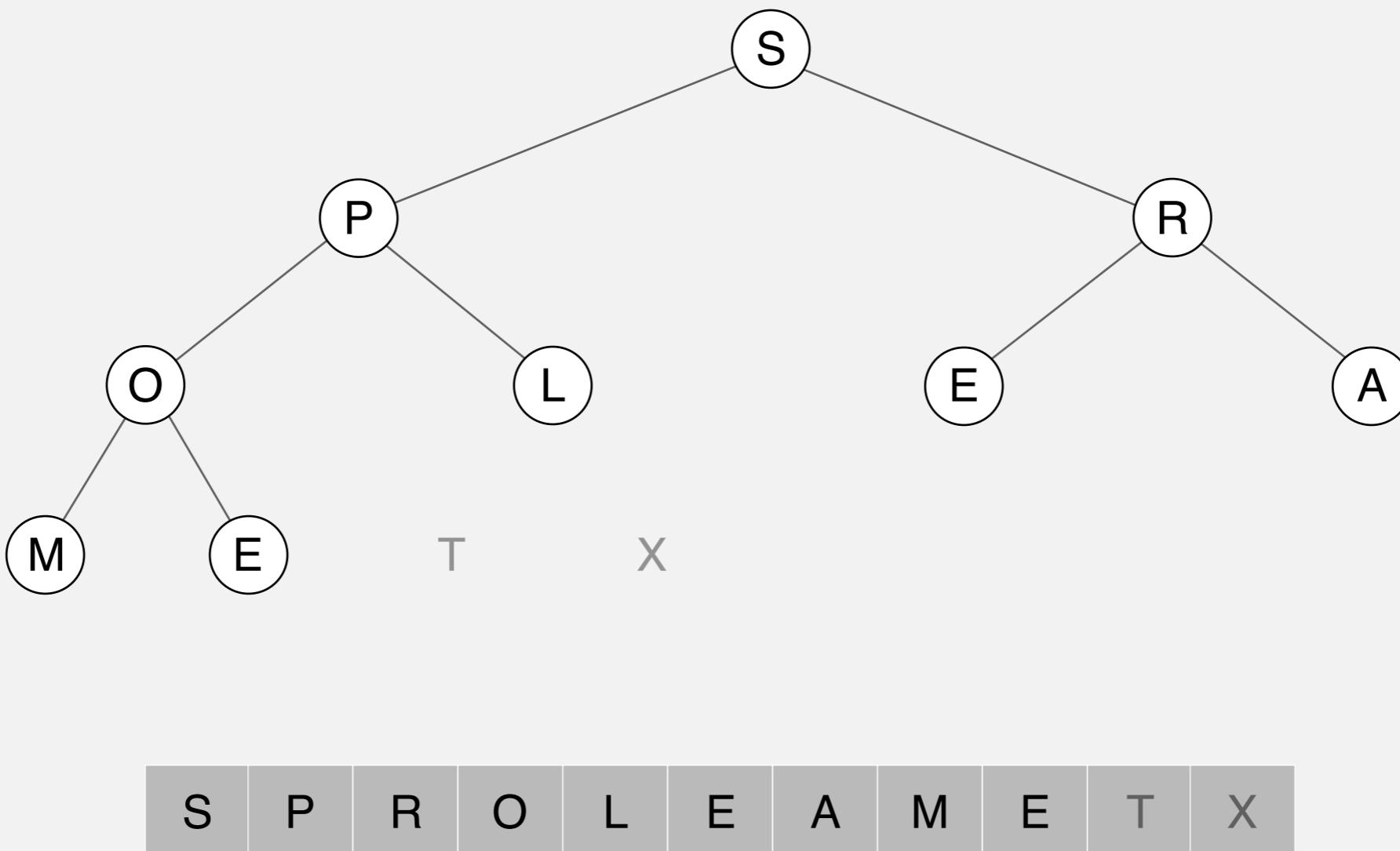
sink 1



S	P	R	O	L	E	A	M	E	T	X
1	3	3	6	T	X					

Heapsort

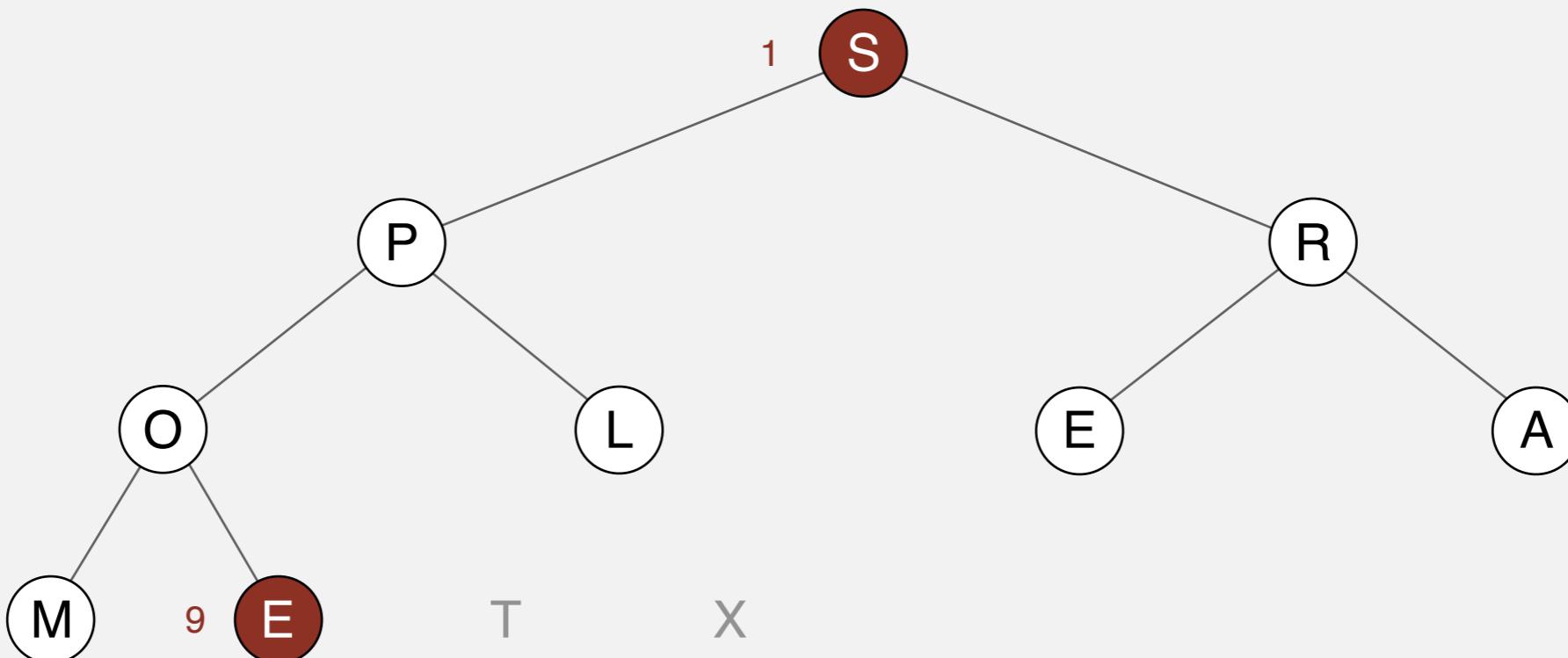
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

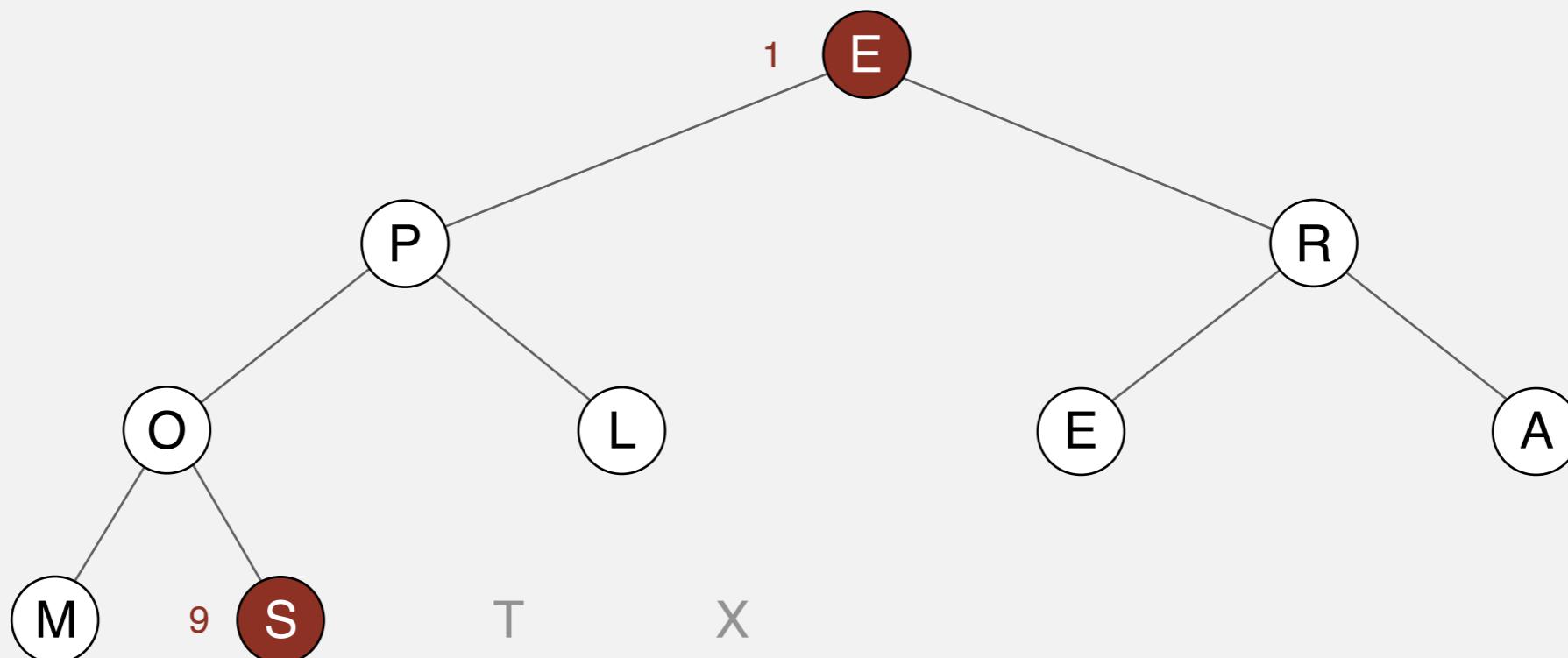


S	P	R	O	L	E	A	M	E	T	X
1								9		

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

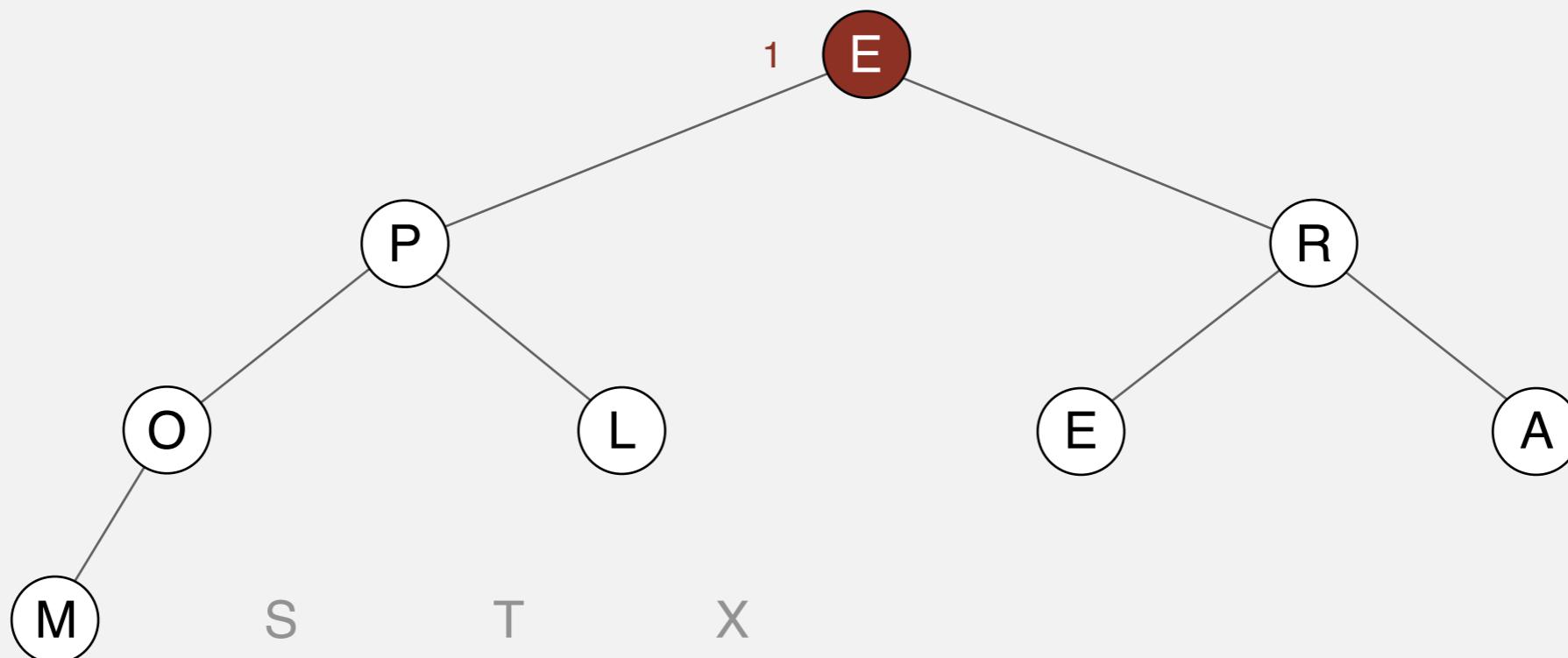
exchange 1 and 9



E	P	R	O	L	E	A	M	S	T	X
1								9		

Sortdown. Repeatedly delete the largest remaining item.

sink 1

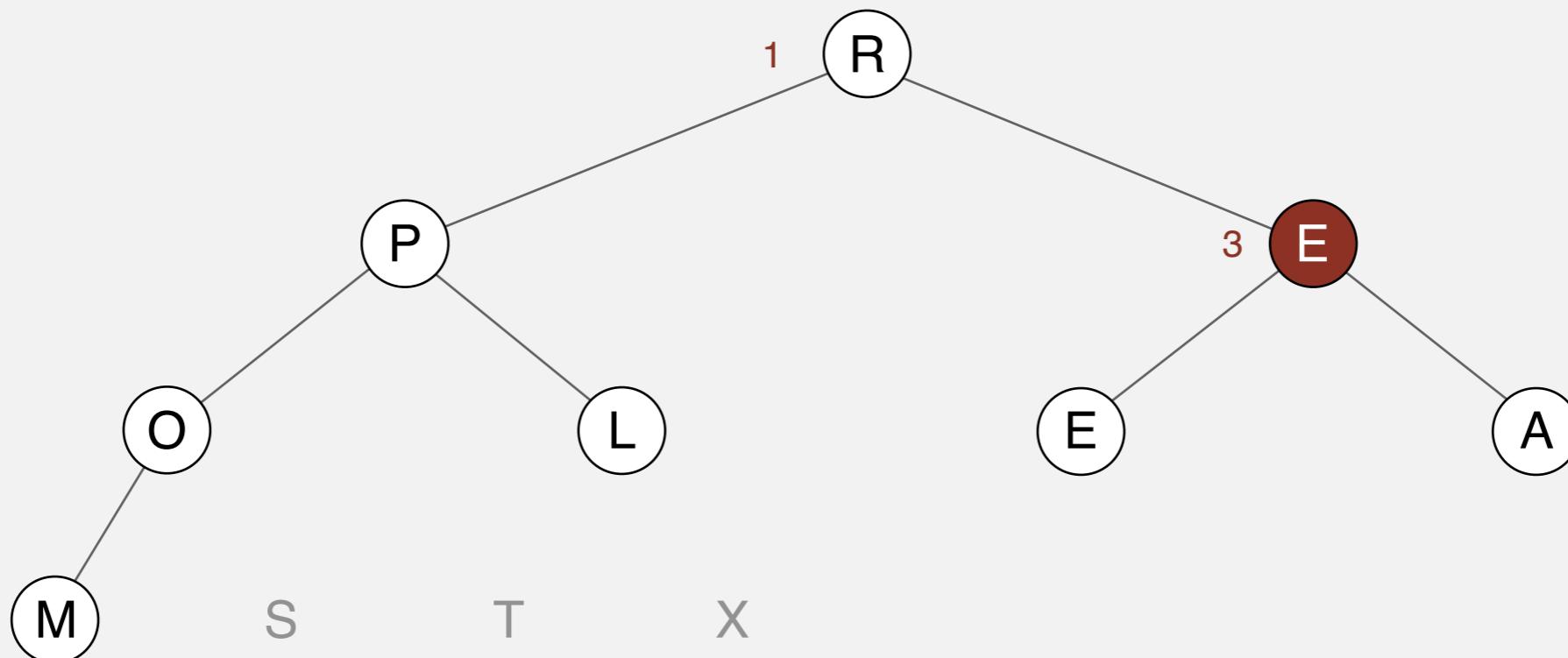


E	P	R	O	L	E	A	M	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

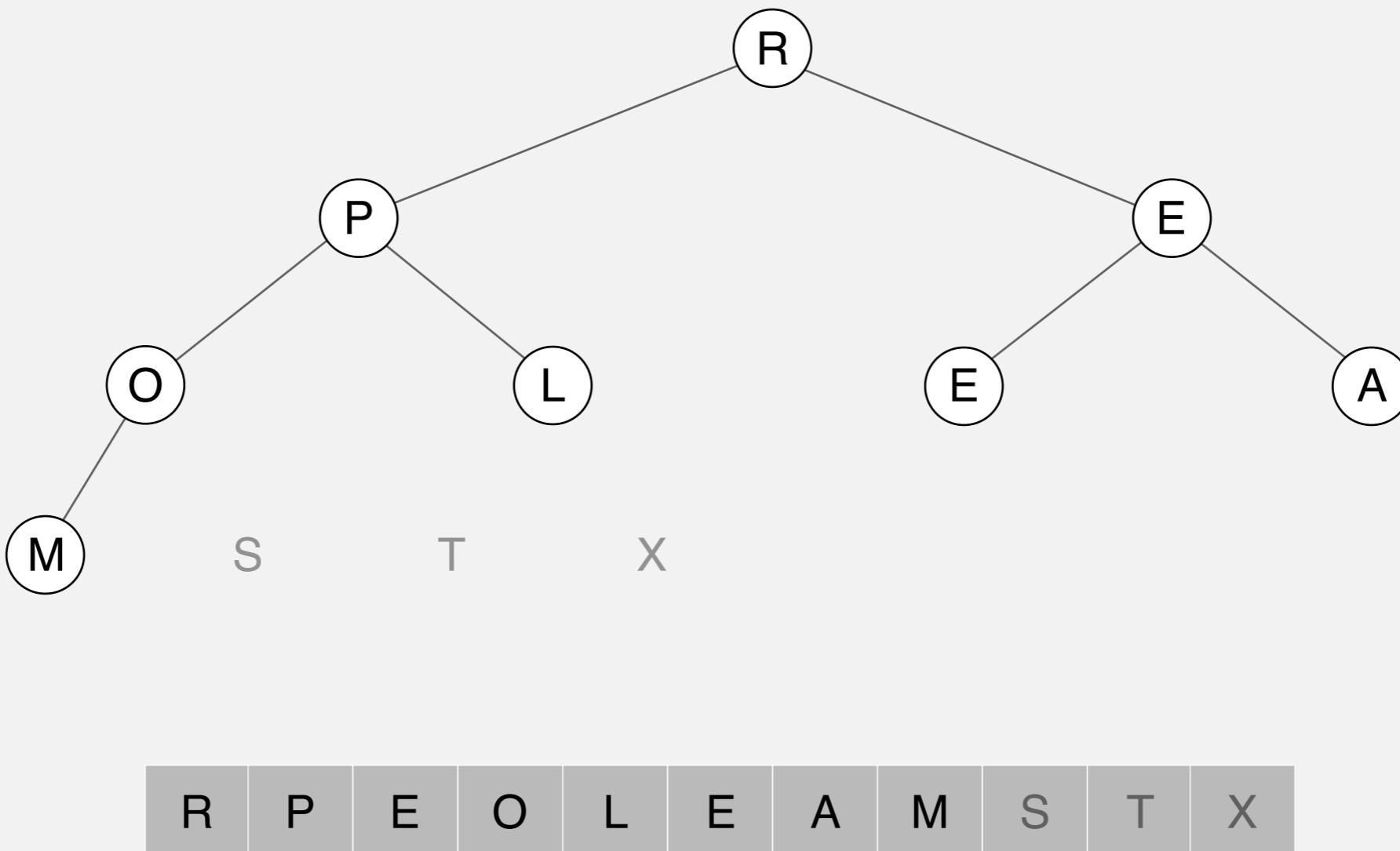
sink 1



R	P	E	O	L	E	A	M	S	T	X
1		3								

Heapsort

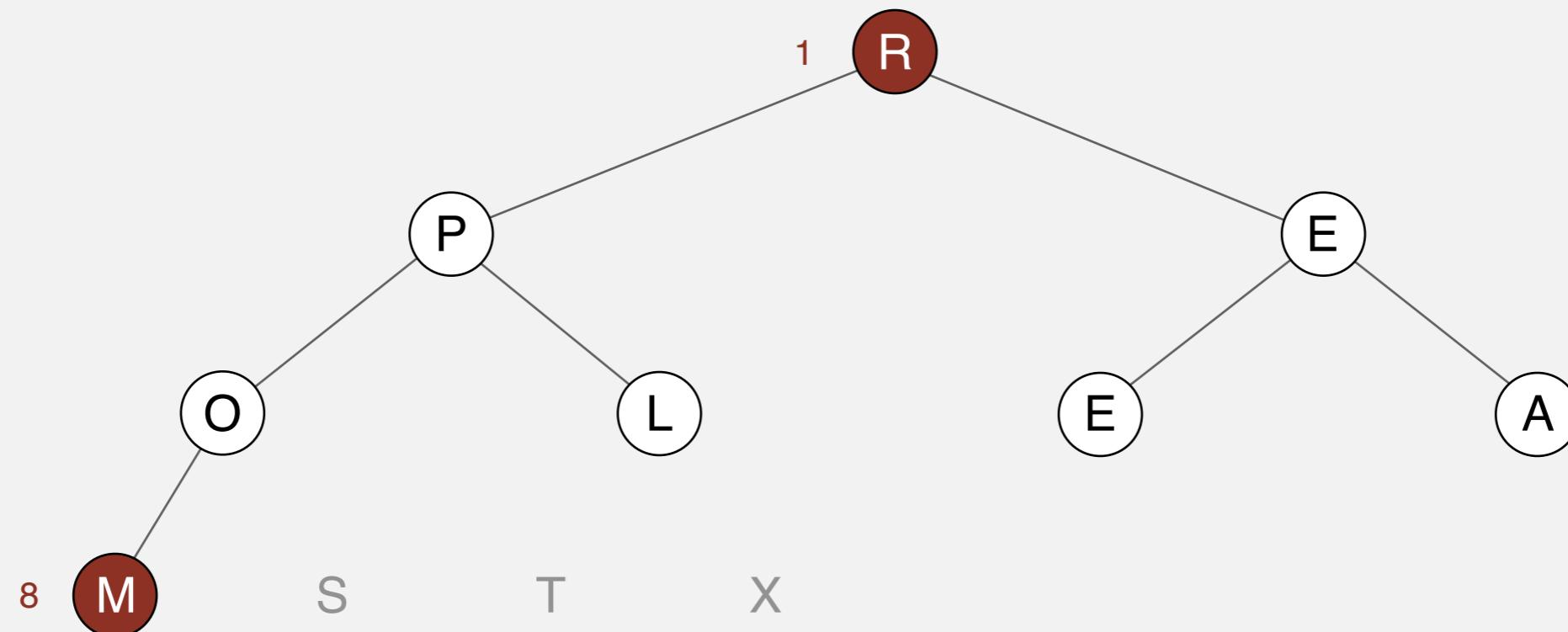
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

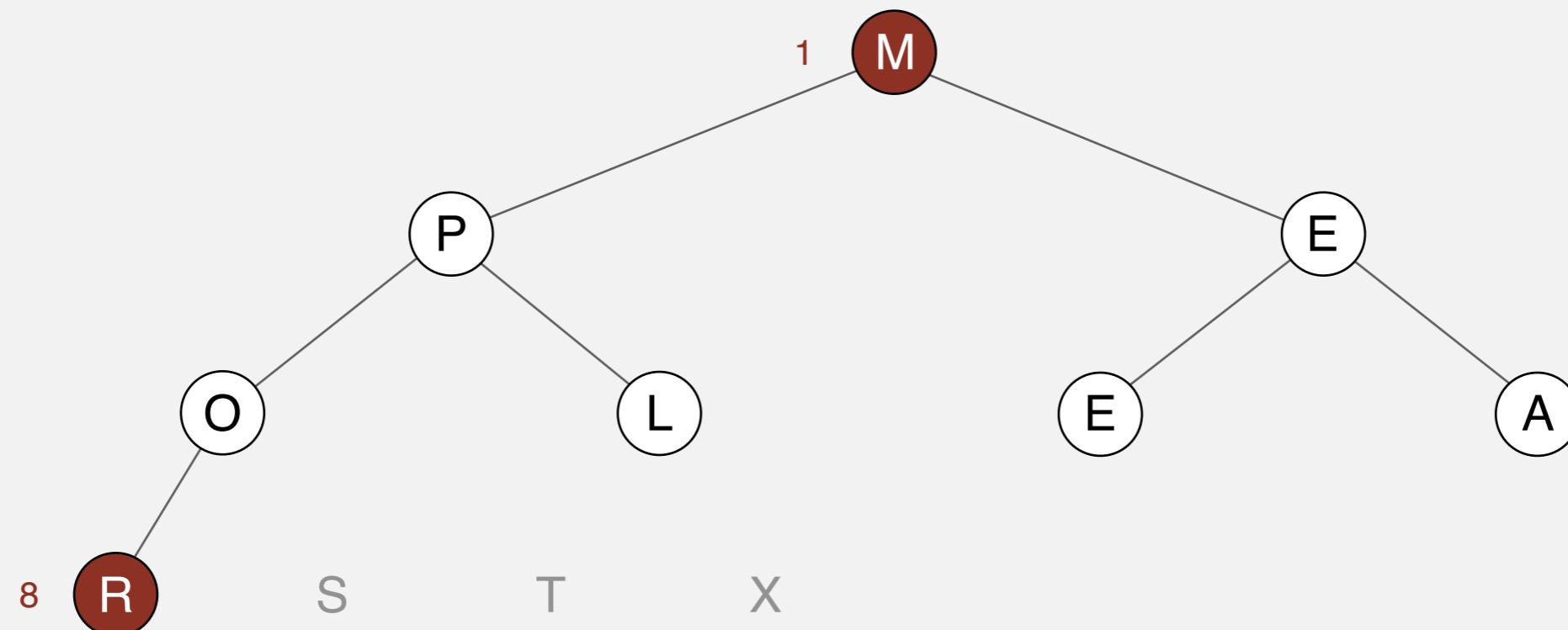
exchange 1 and 8



R	P	E	O	L	E	A	M	S	T	X
1							8			

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

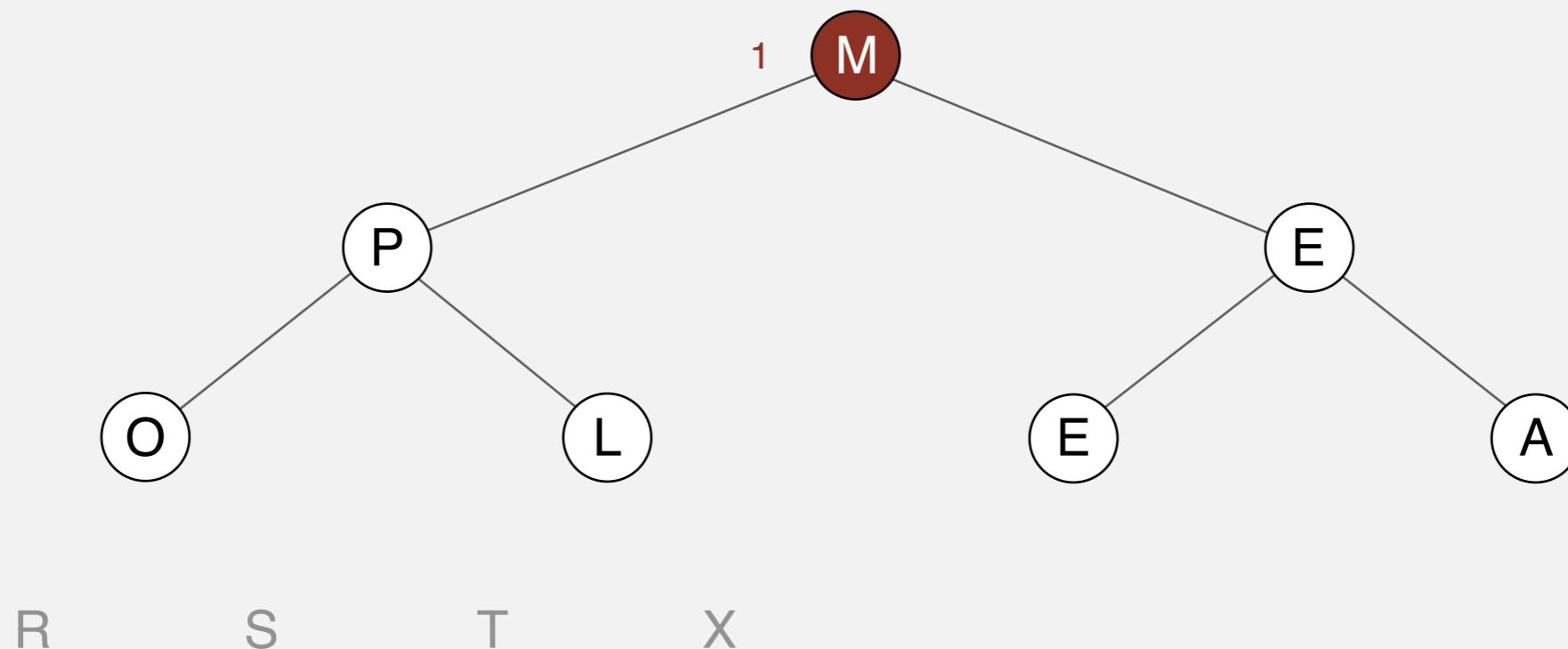


M	P	E	O	L	E	A	R	S	T	X
1							8			

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

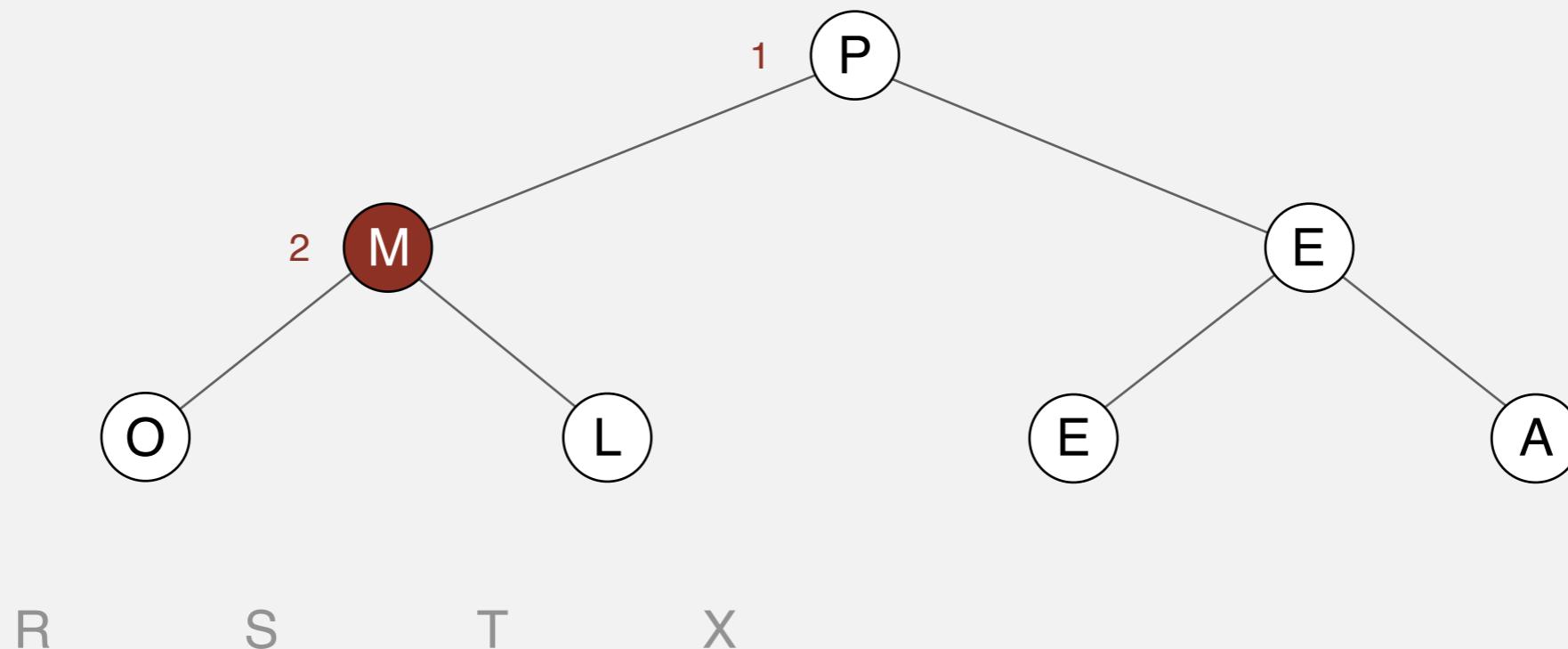


M	P	E	O	L	E	A	R	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

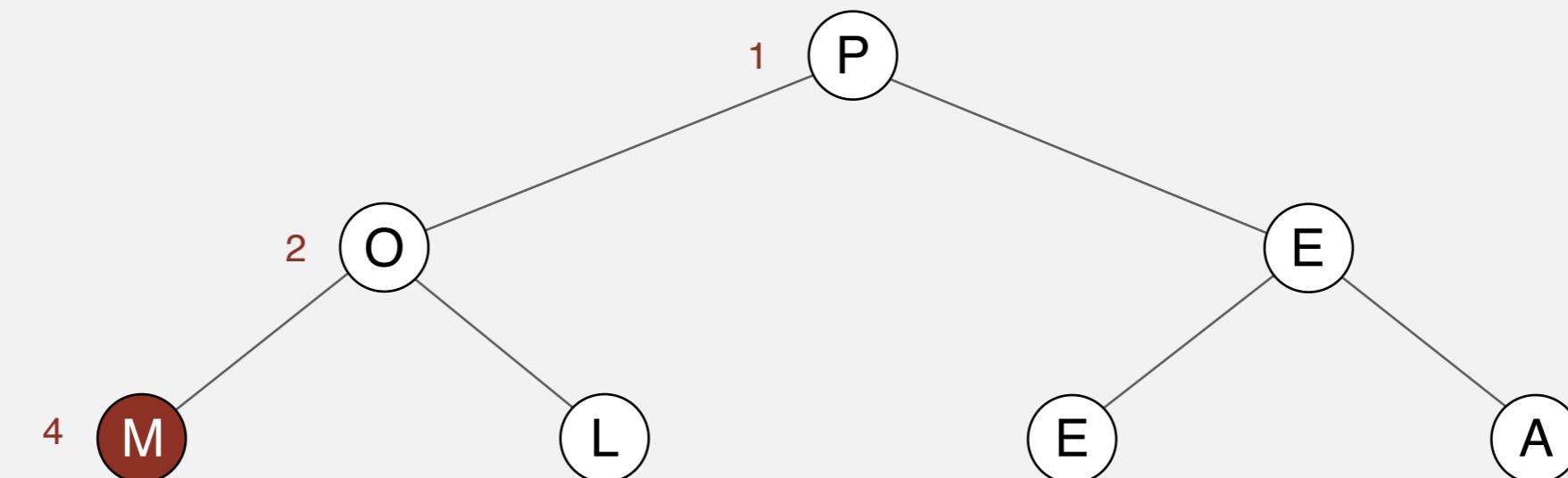


P	M	E	O	L	E	A	R	S	T	X
1	2									

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

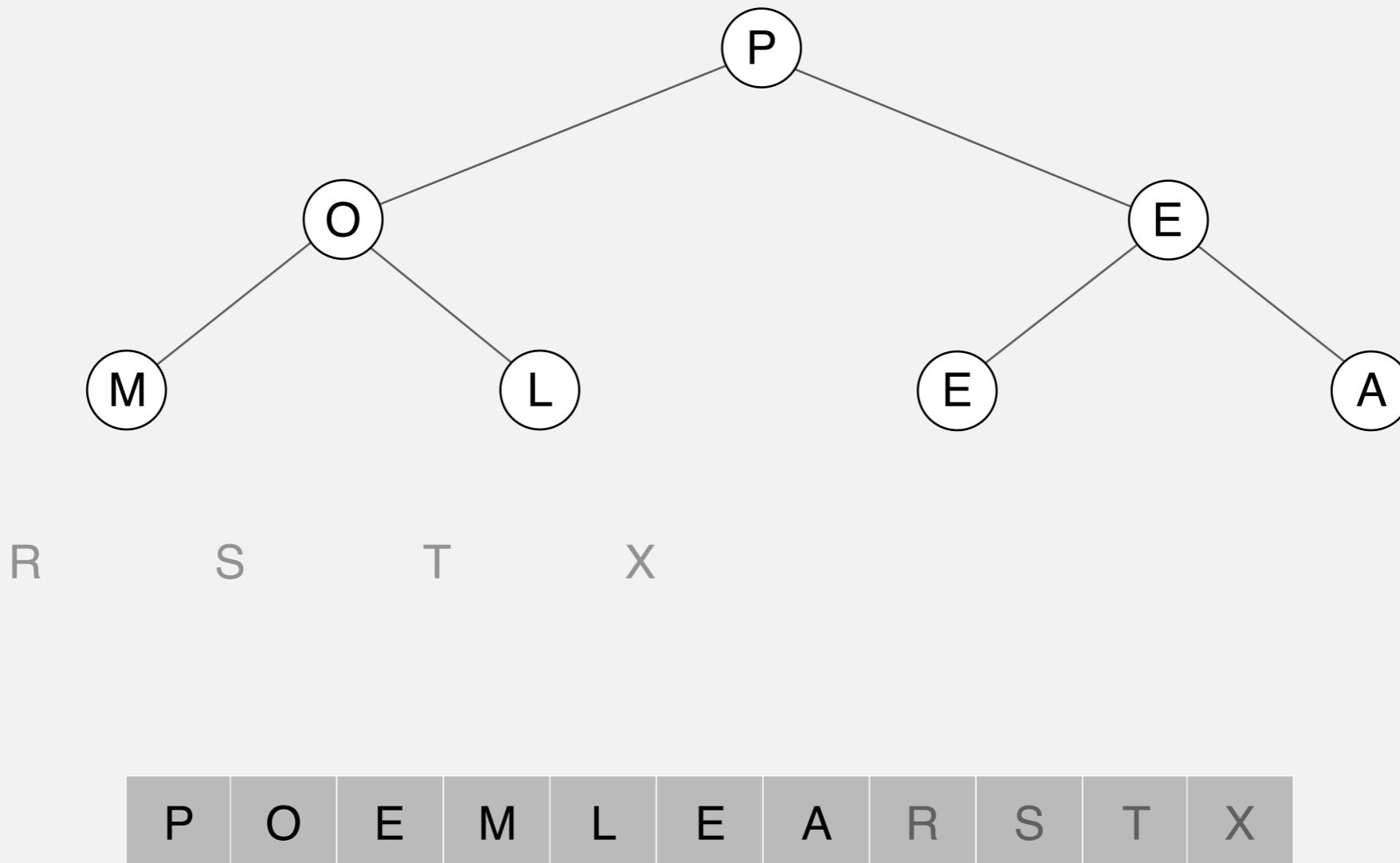


R S T X

P	O	E	M	L	E	A	R	S	T	X
1	2	4								

Heapsort

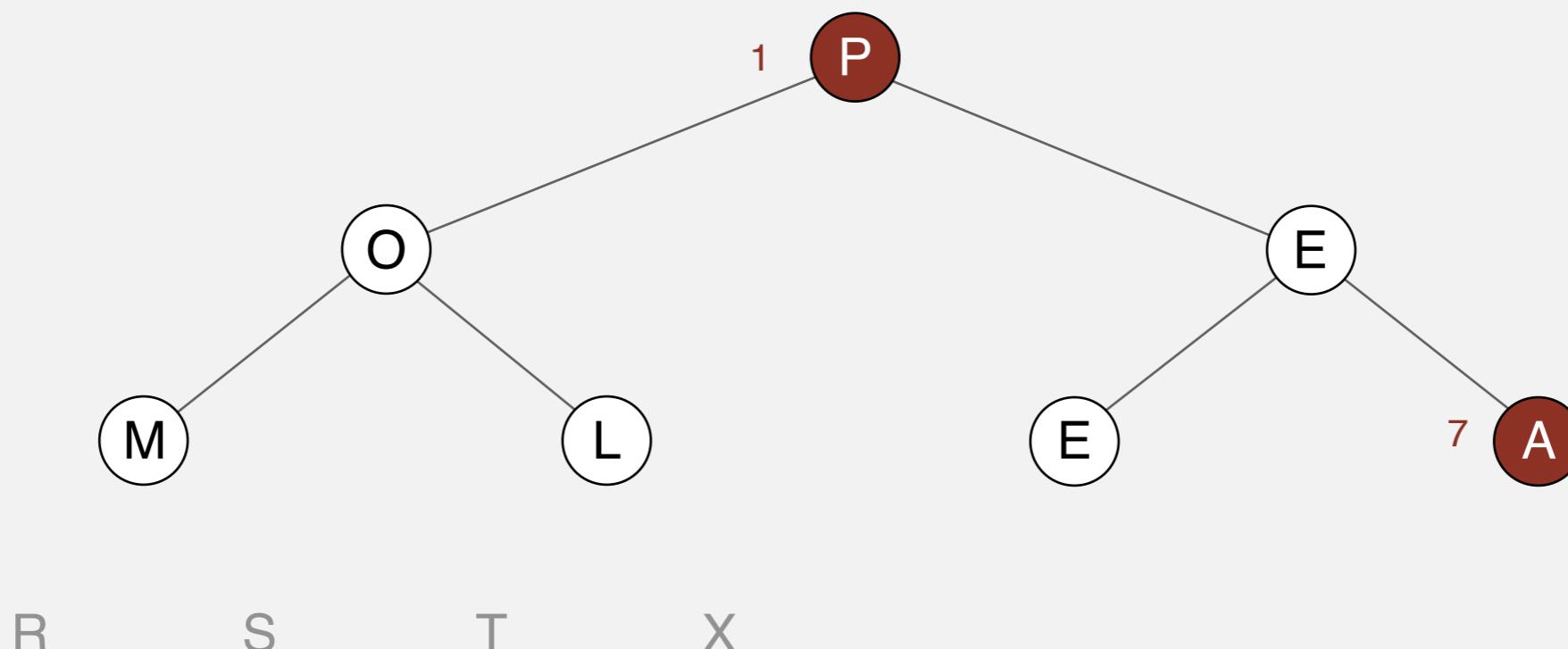
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7

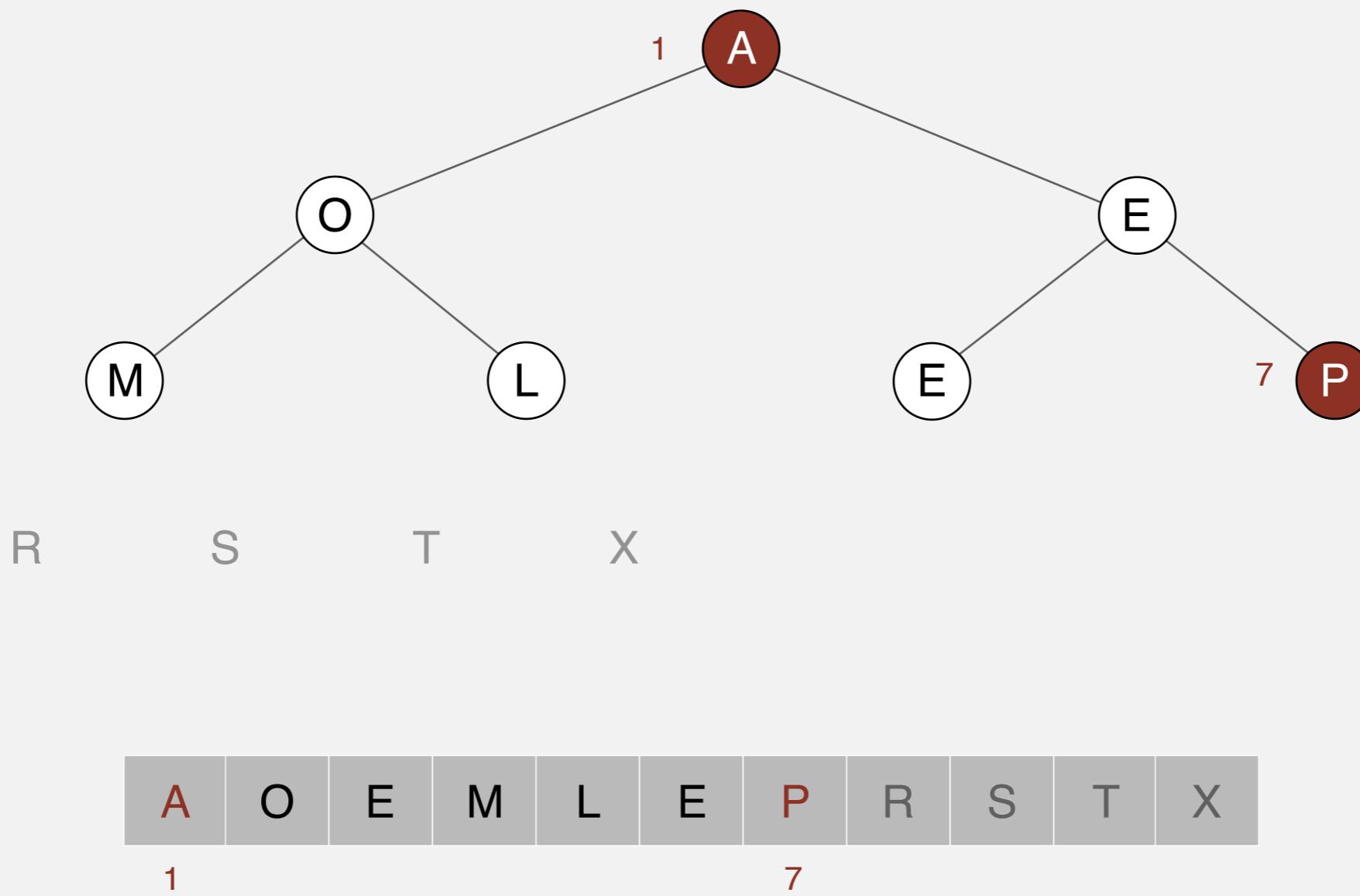


P	O	E	M	L	E	A	R	S	T	X
1						7				

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

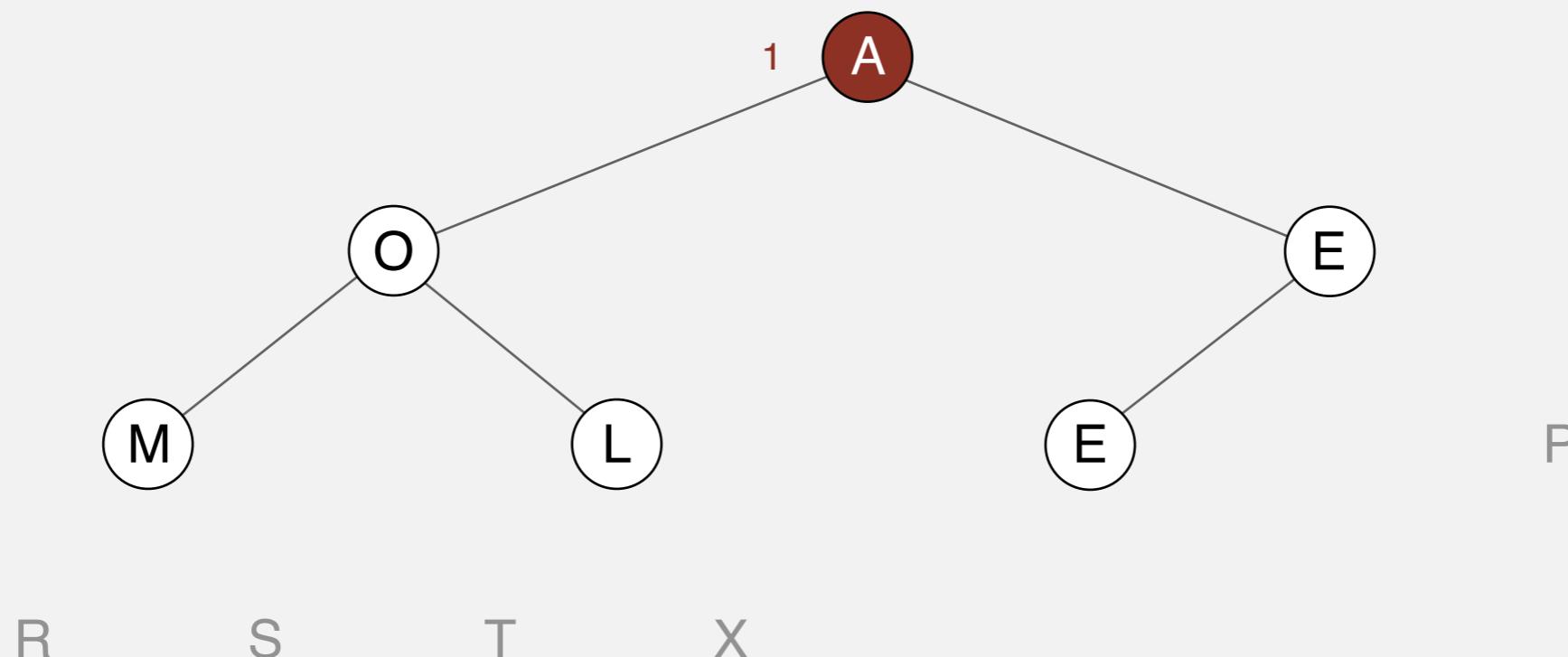
exchange 1 and 7



Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

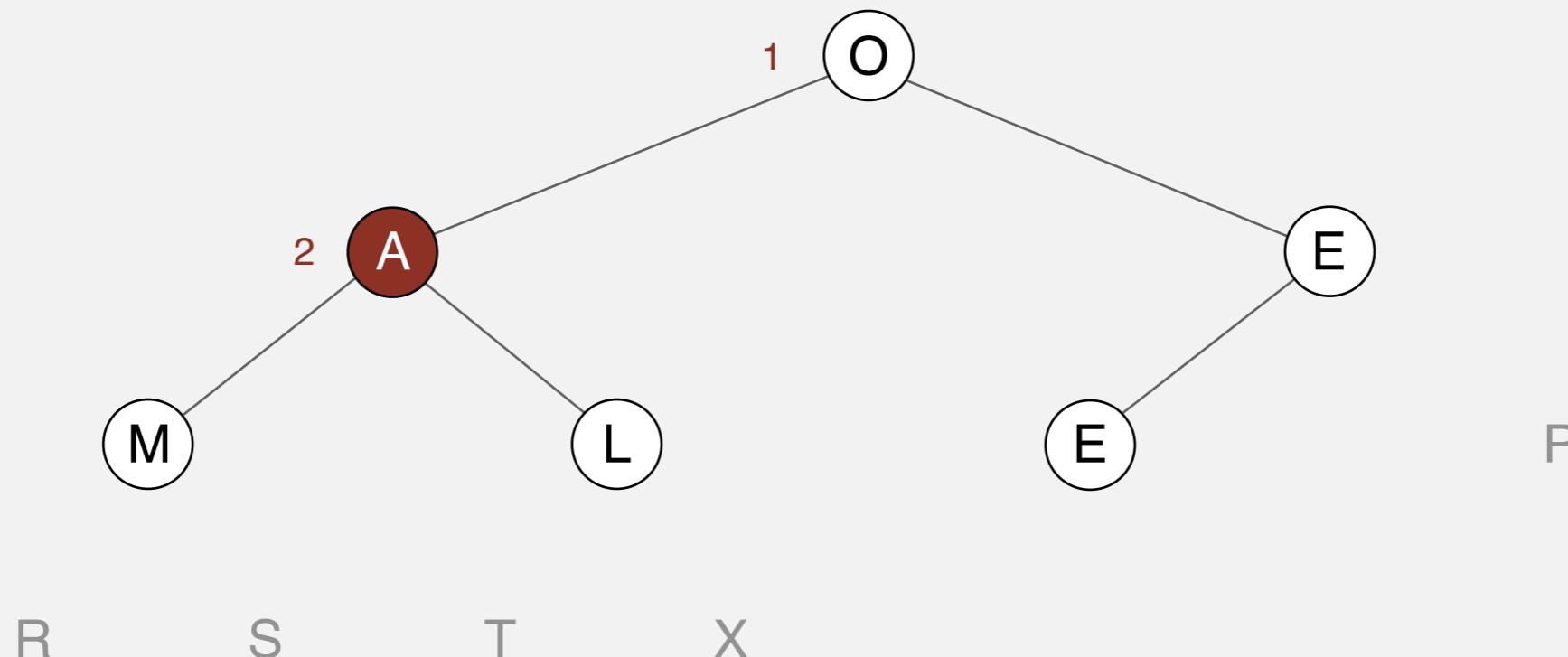


A	O	E	M	L	E	P	R	S	T	X
---	---	---	---	---	---	---	---	---	---	---

1

Sortdown. Repeatedly delete the largest remaining item.

sink 1

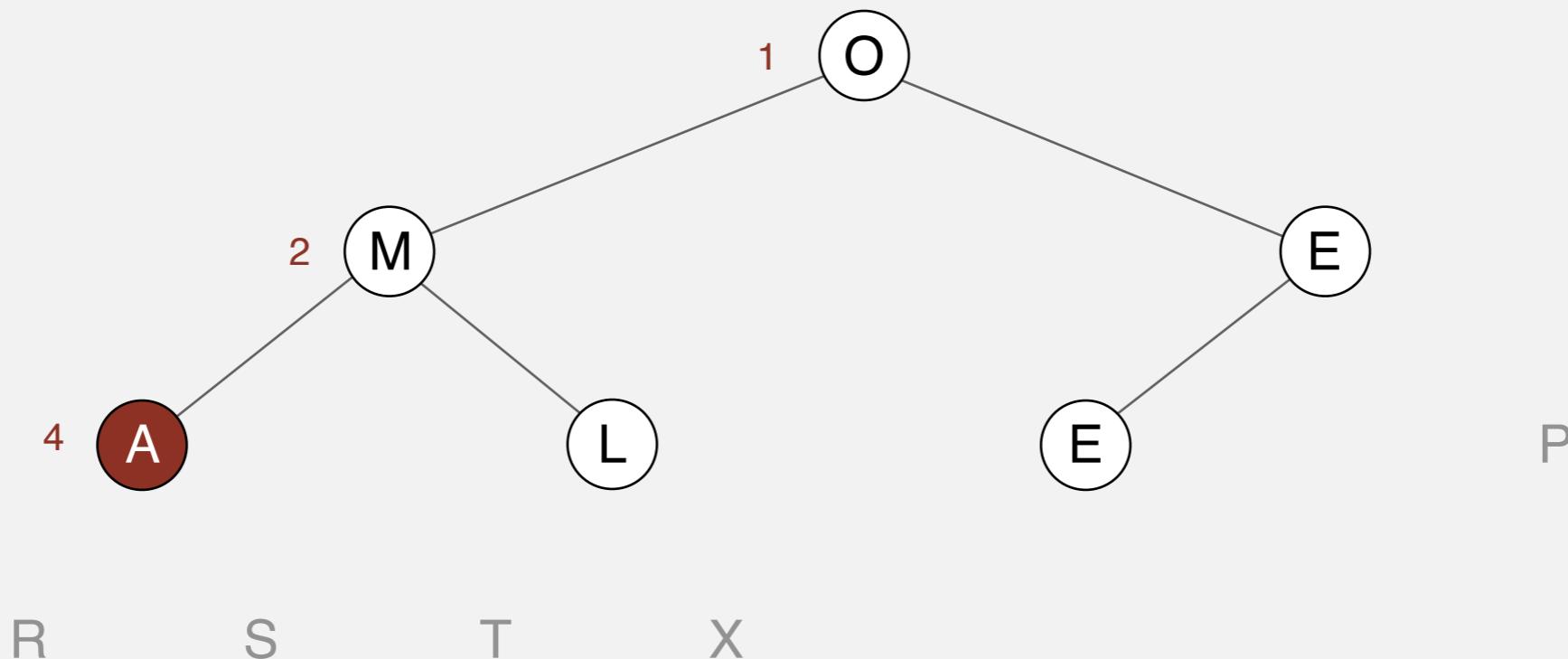


O	A	E	M	L	E	P	R	S	T	X
1	2									

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

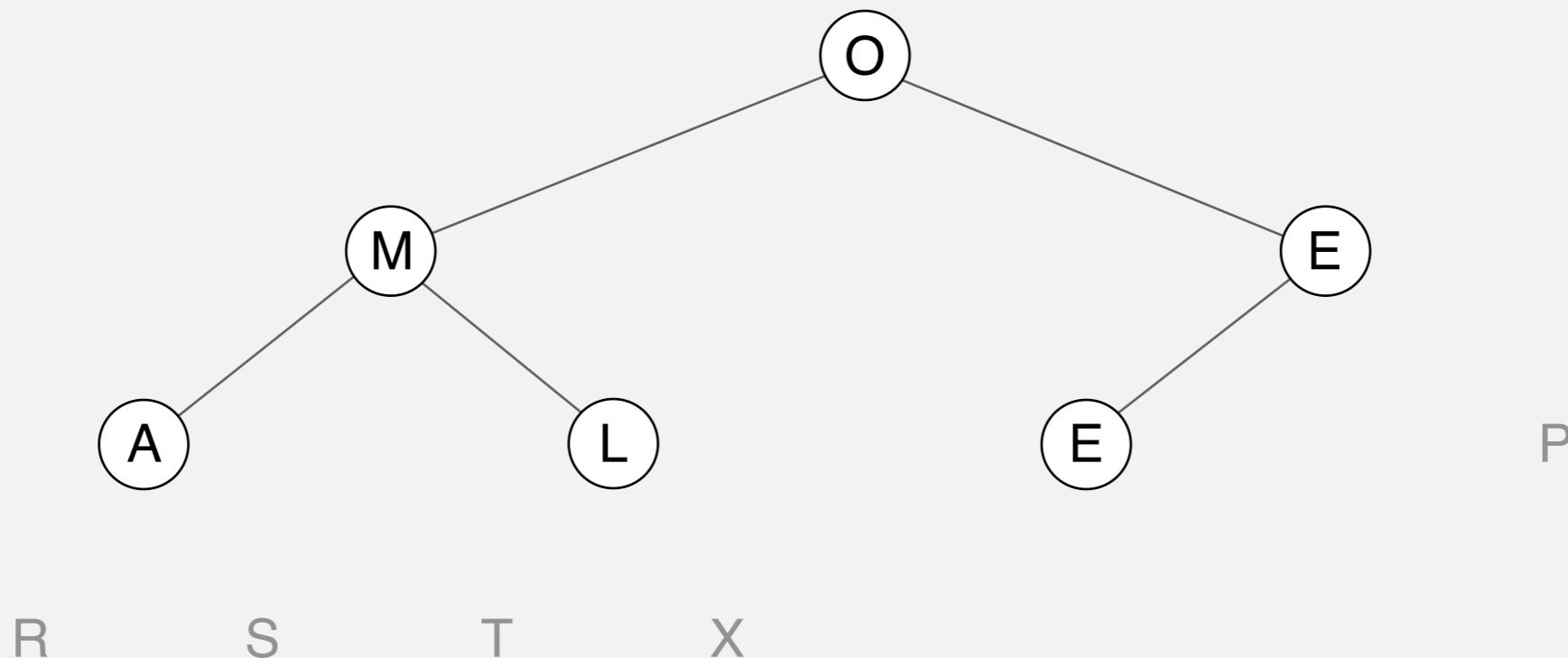


O	M	E	A	L	E	P	R	S	T	X
1	2		4							

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

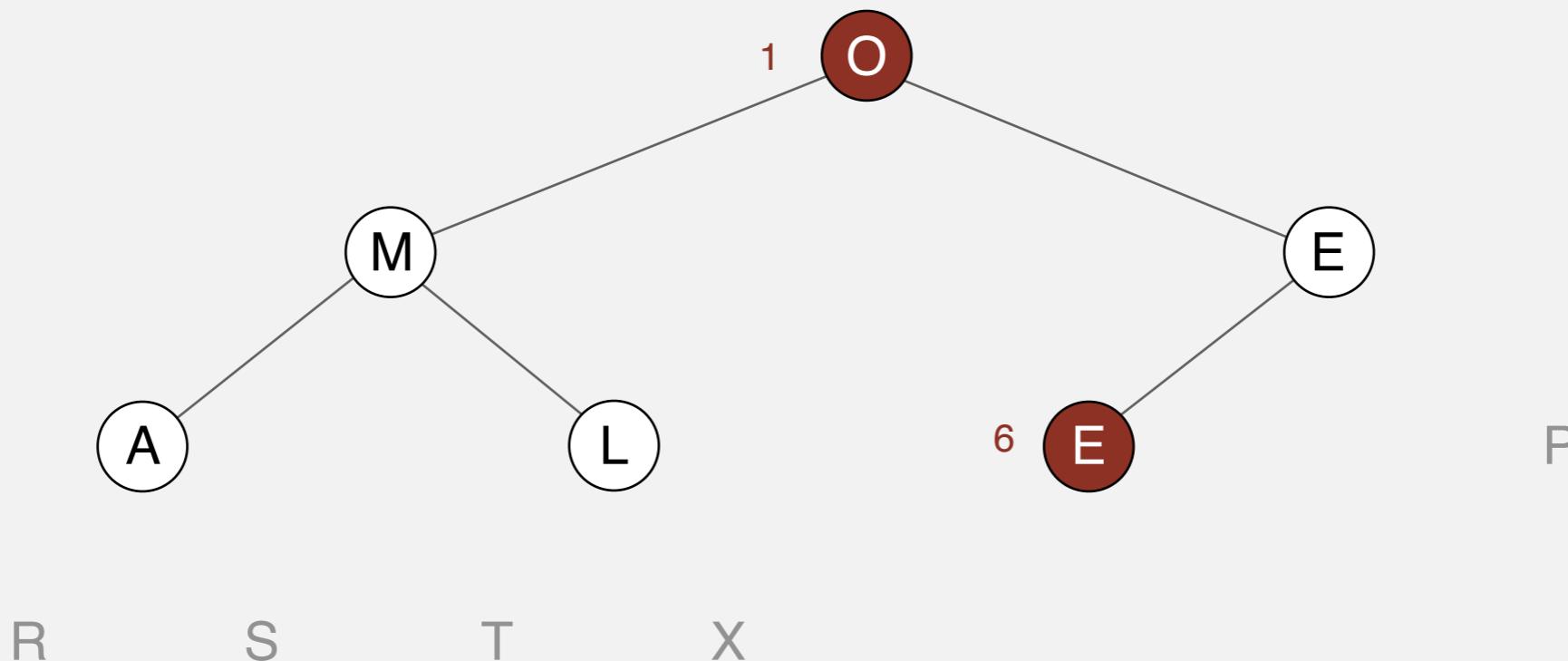


O	M	E	A	L	E	P	R	S	T	X
---	---	---	---	---	---	---	---	---	---	---

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

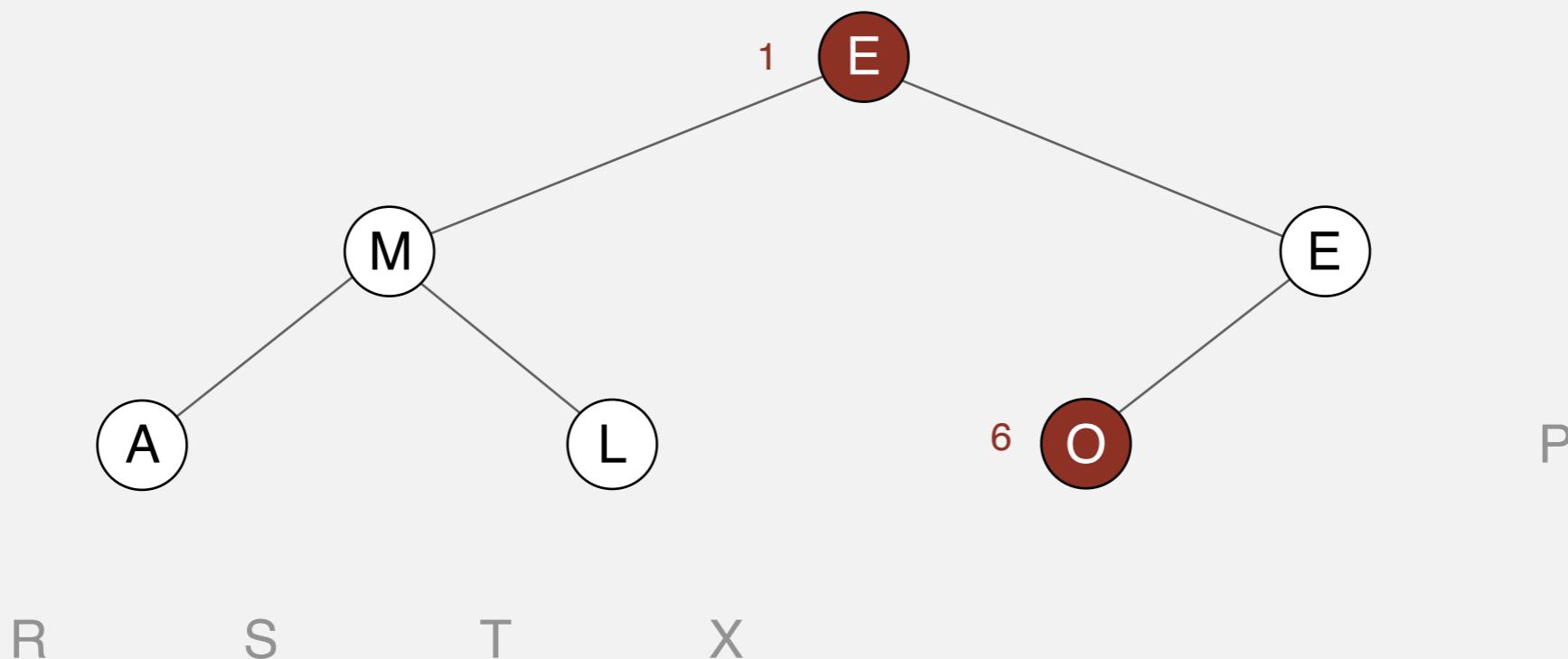


O	M	E	A	L	E	P	R	S	T	X
1					6					

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

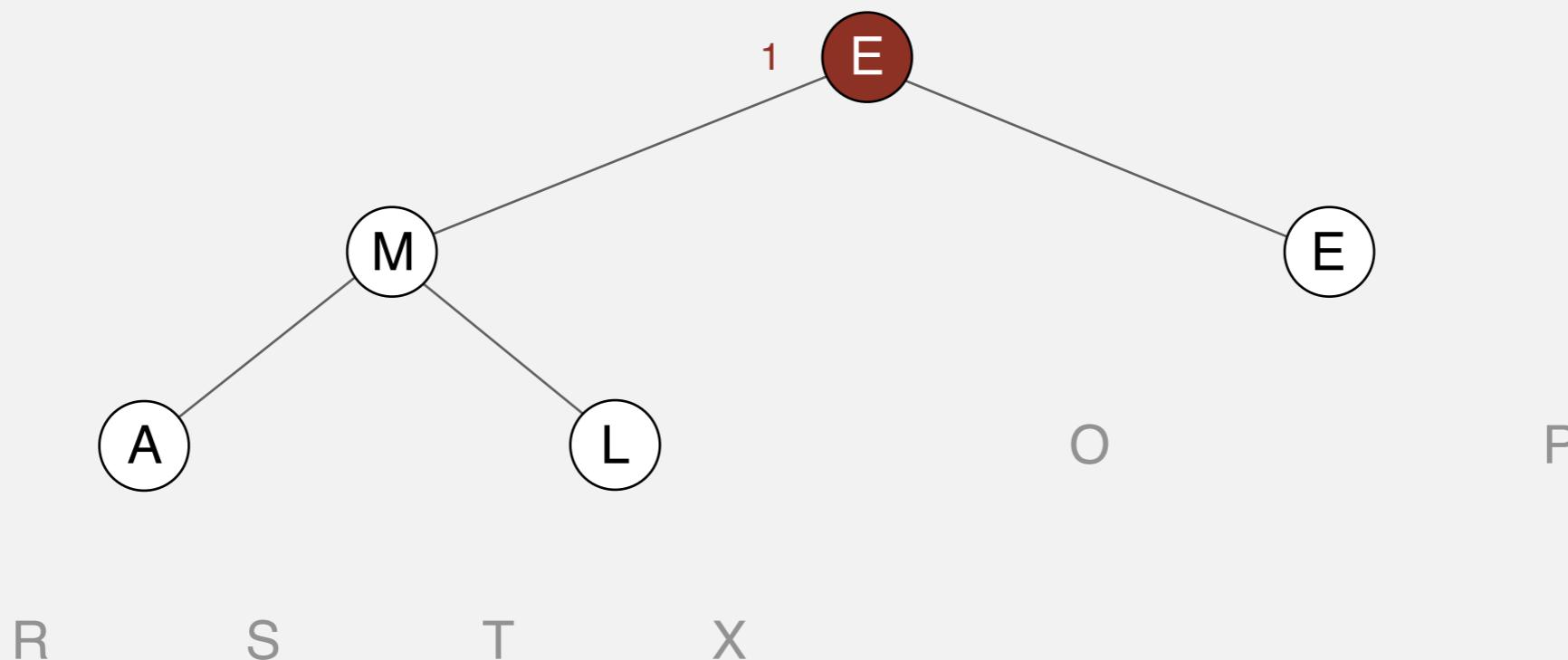
exchange 1 and 6



E	M	E	A	L	O	P	R	S	T	X
1					6					

Sortdown. Repeatedly delete the largest remaining item.

sink 1

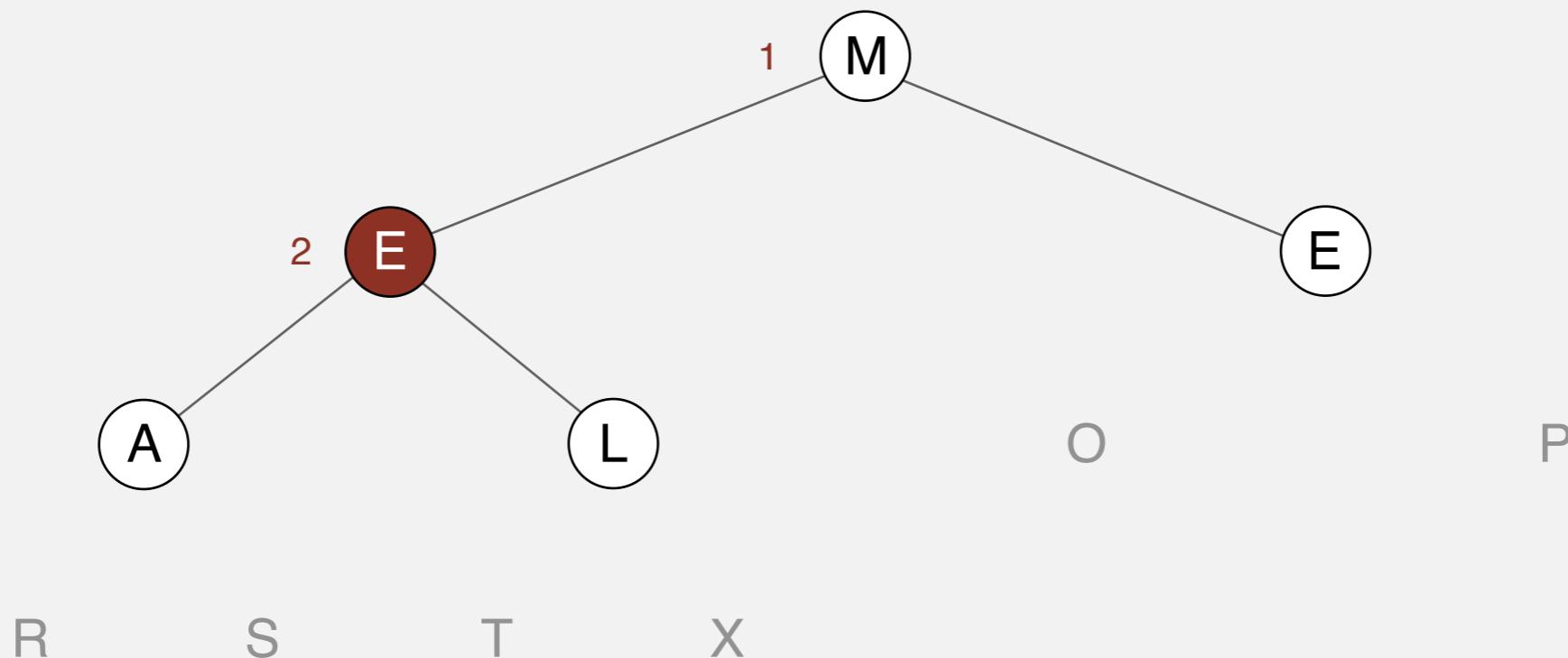


E	M	E	A	L	O	P	R	S	T	X
1										

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

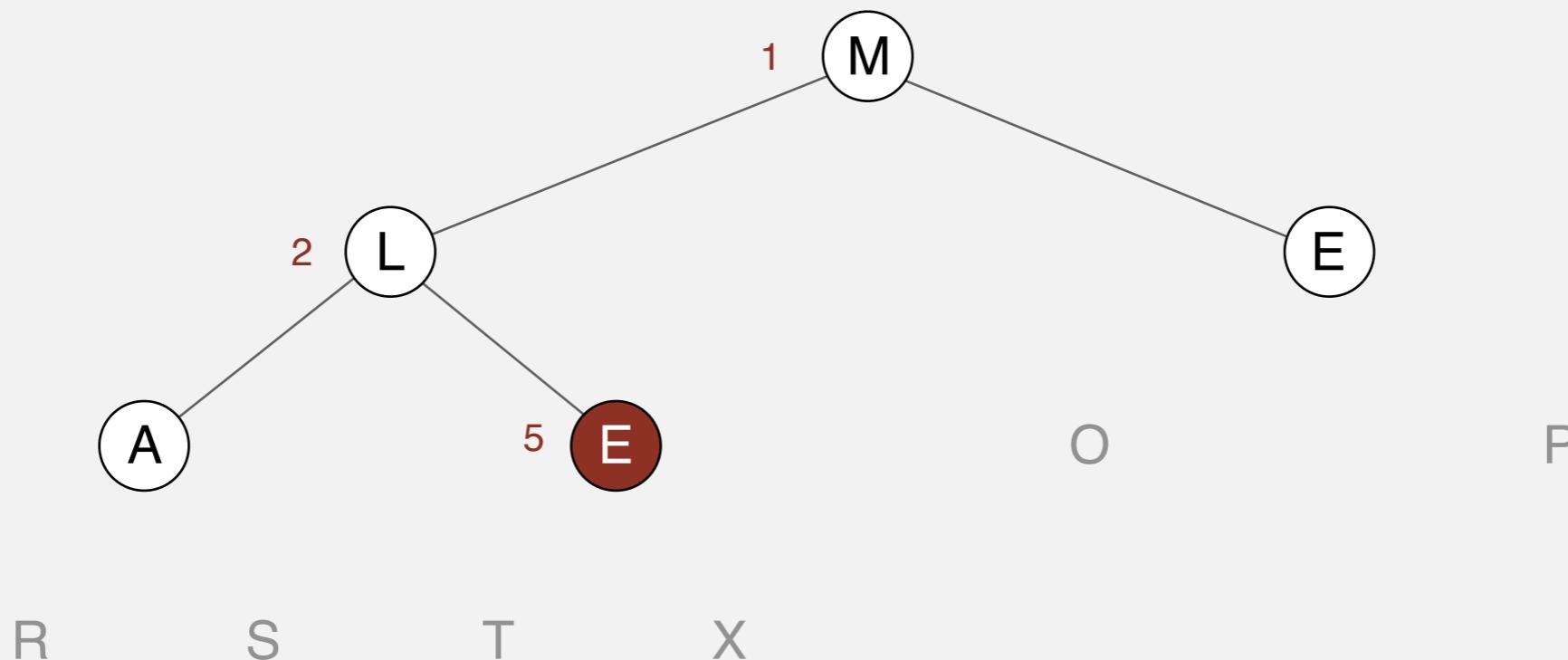
sink 1



M	E	E	A	L	O	P	R	S	T	X
1	2									

Sortdown. Repeatedly delete the largest remaining item.

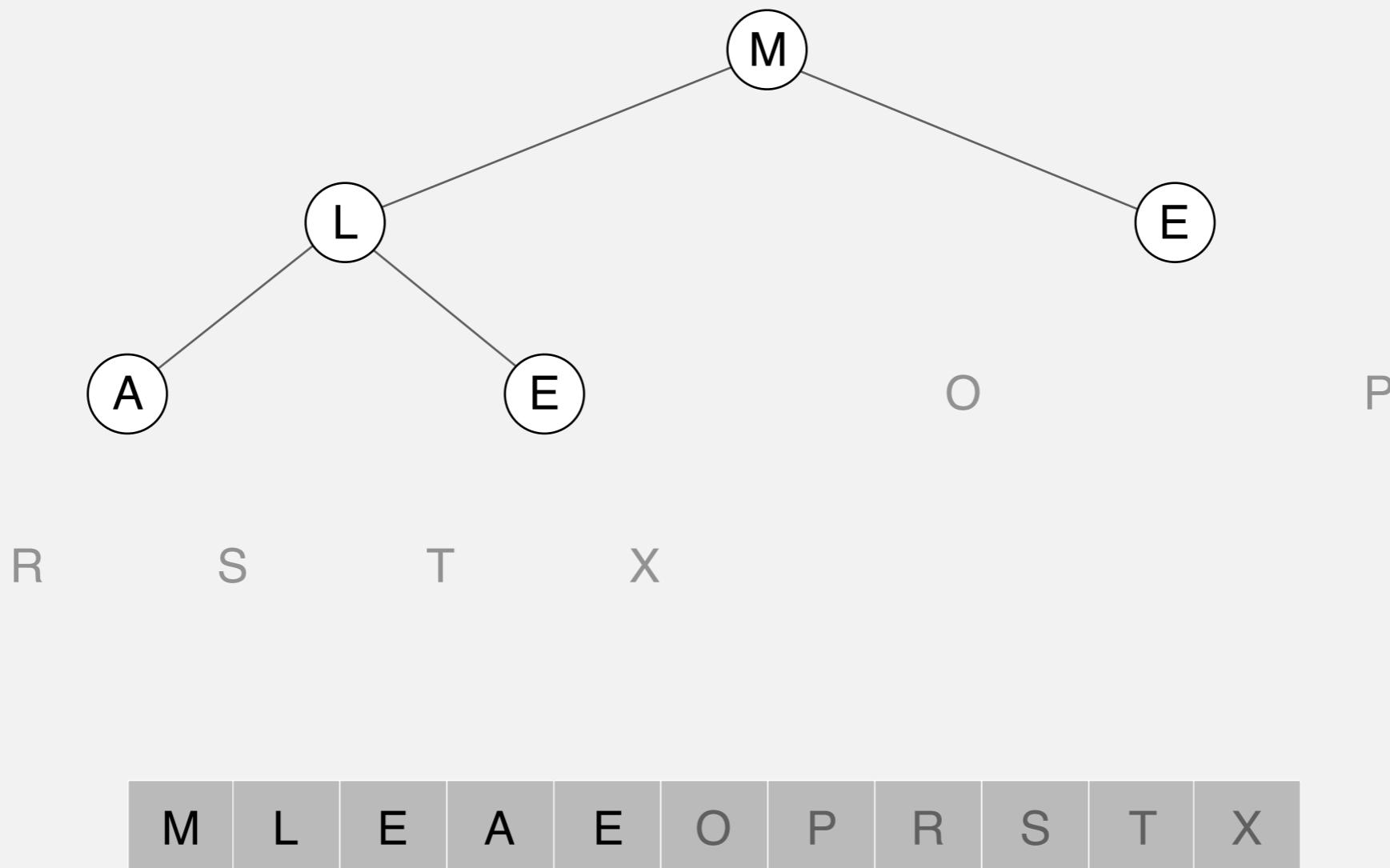
sink 1



M	L	E	A	E	O	P	R	S	T	X
1	2			5	0	0	0	0	0	0

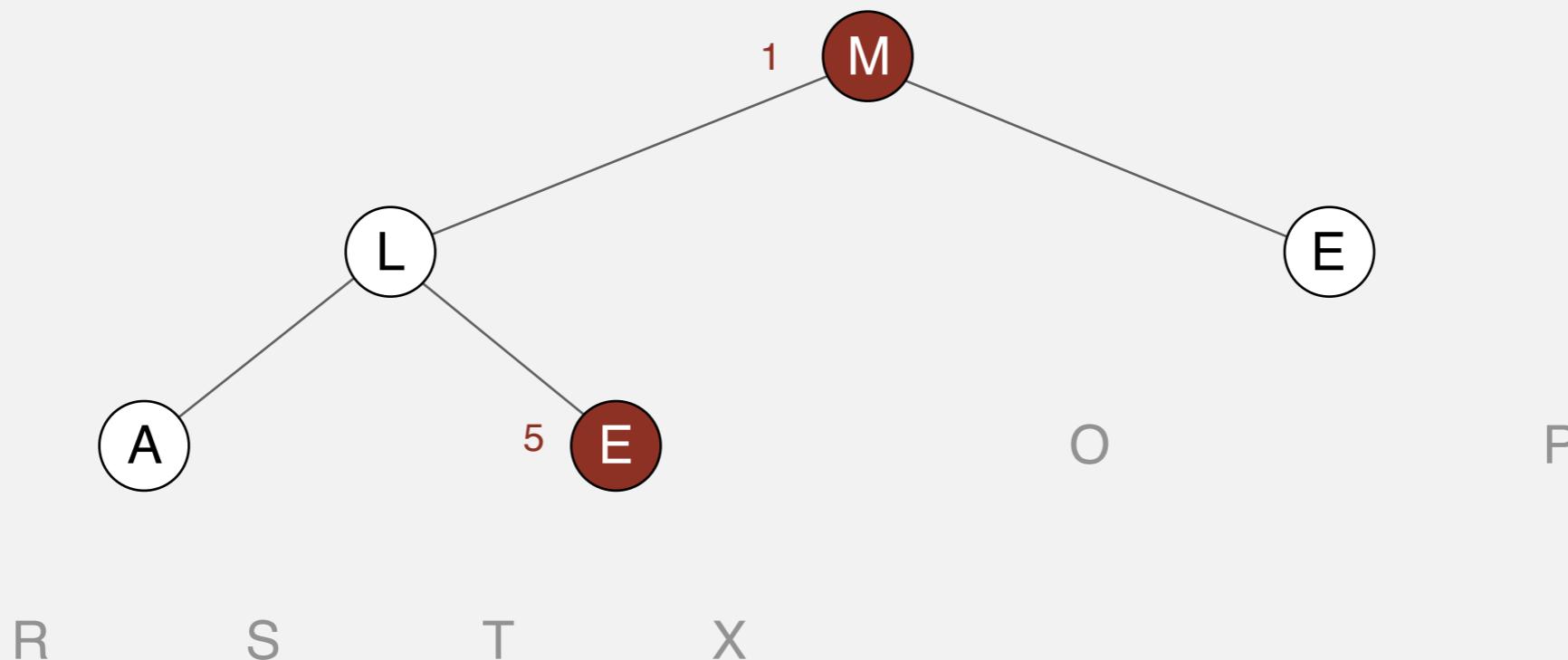
Heapsort

Sortdown. Repeatedly delete the largest remaining item.



Sortdown. Repeatedly delete the largest remaining item.

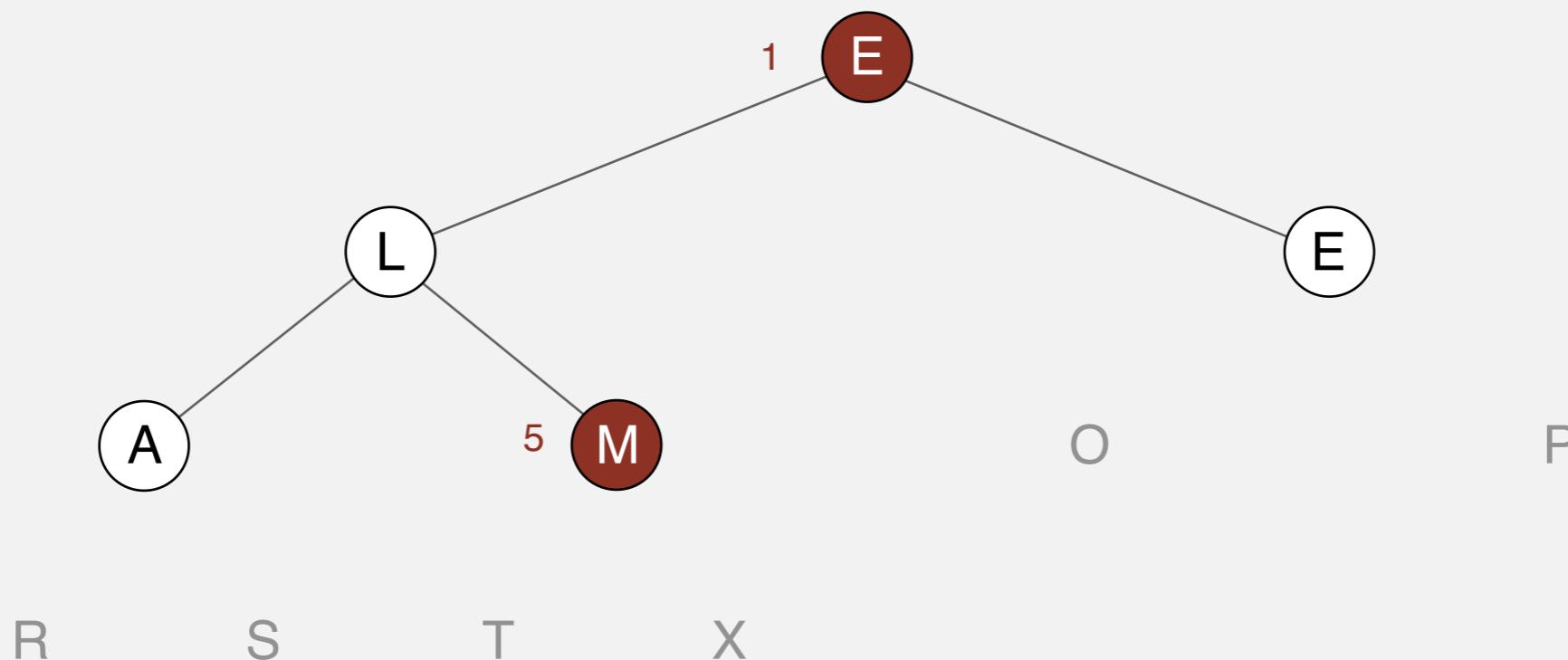
exchange 1 and 5



M	L	E	A	E	O	P	R	S	T	X
1				5						

Sortdown. Repeatedly delete the largest remaining item.

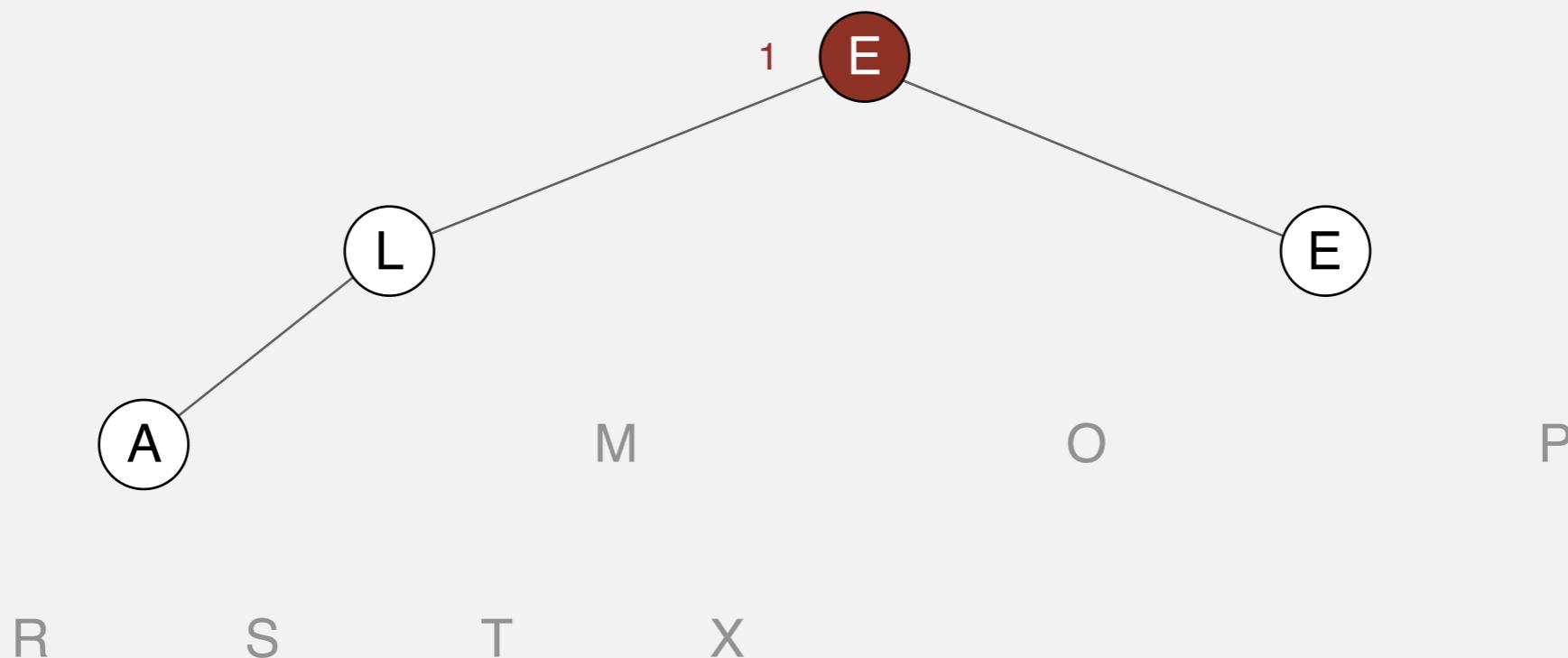
exchange 1 and 5



E	L	E	A	M	O	P	R	S	T	X
1				5						

Sortdown. Repeatedly delete the largest remaining item.

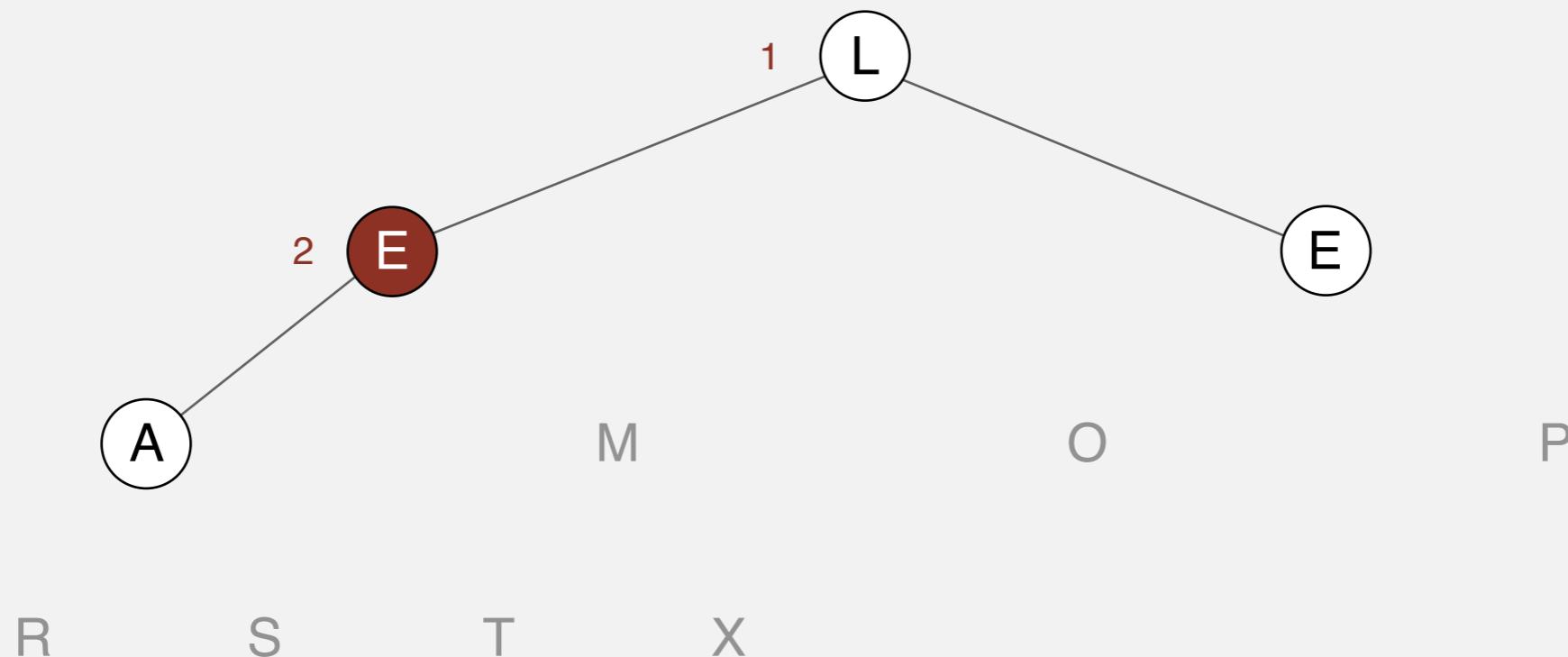
sink 1



E	L	E	A	M	O	P	R	S	T	X
---	---	---	---	---	---	---	---	---	---	---

Sortdown. Repeatedly delete the largest remaining item.

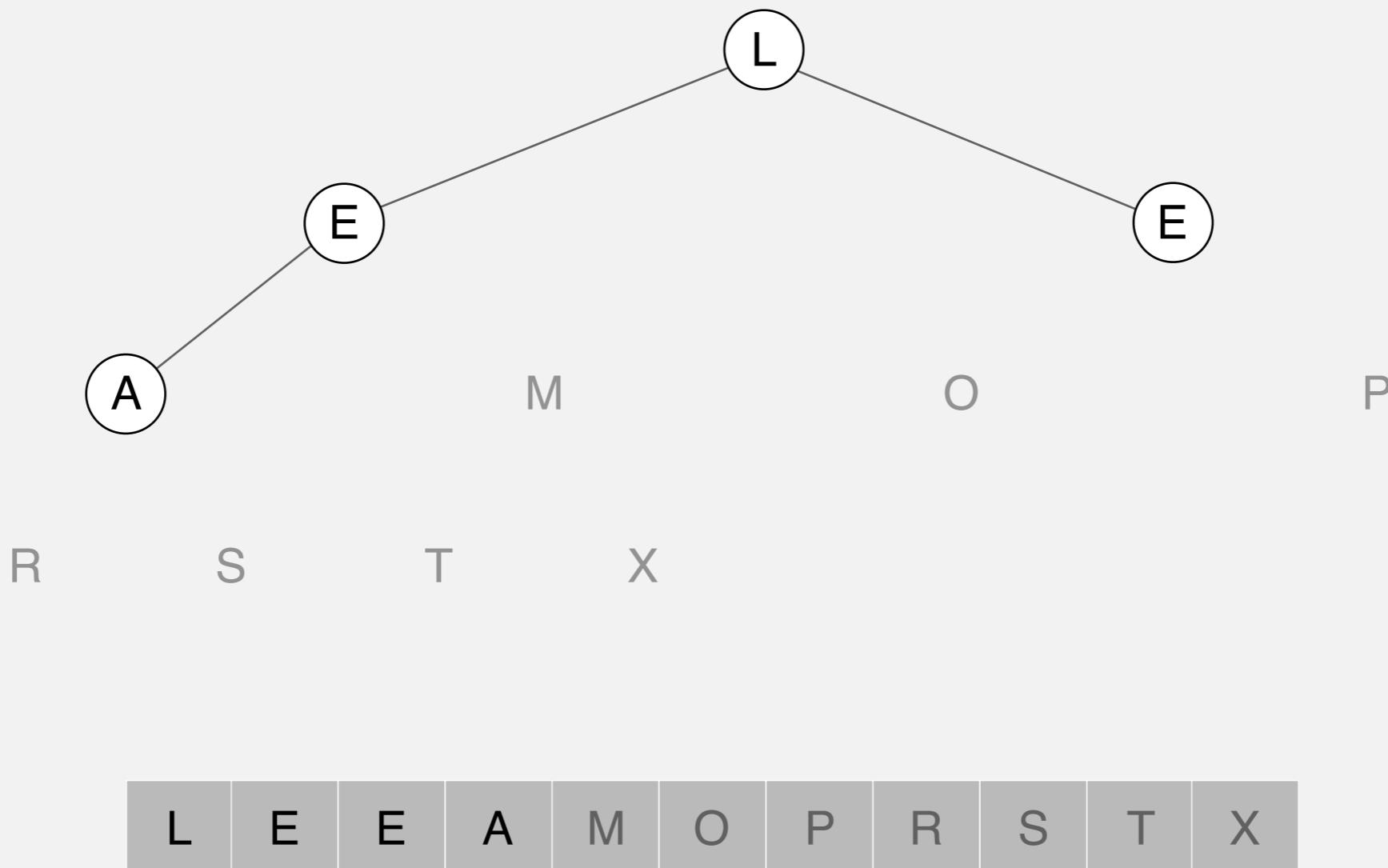
sink 1



L	E	E	A	M	O	P	R	S	T	X
1	2									

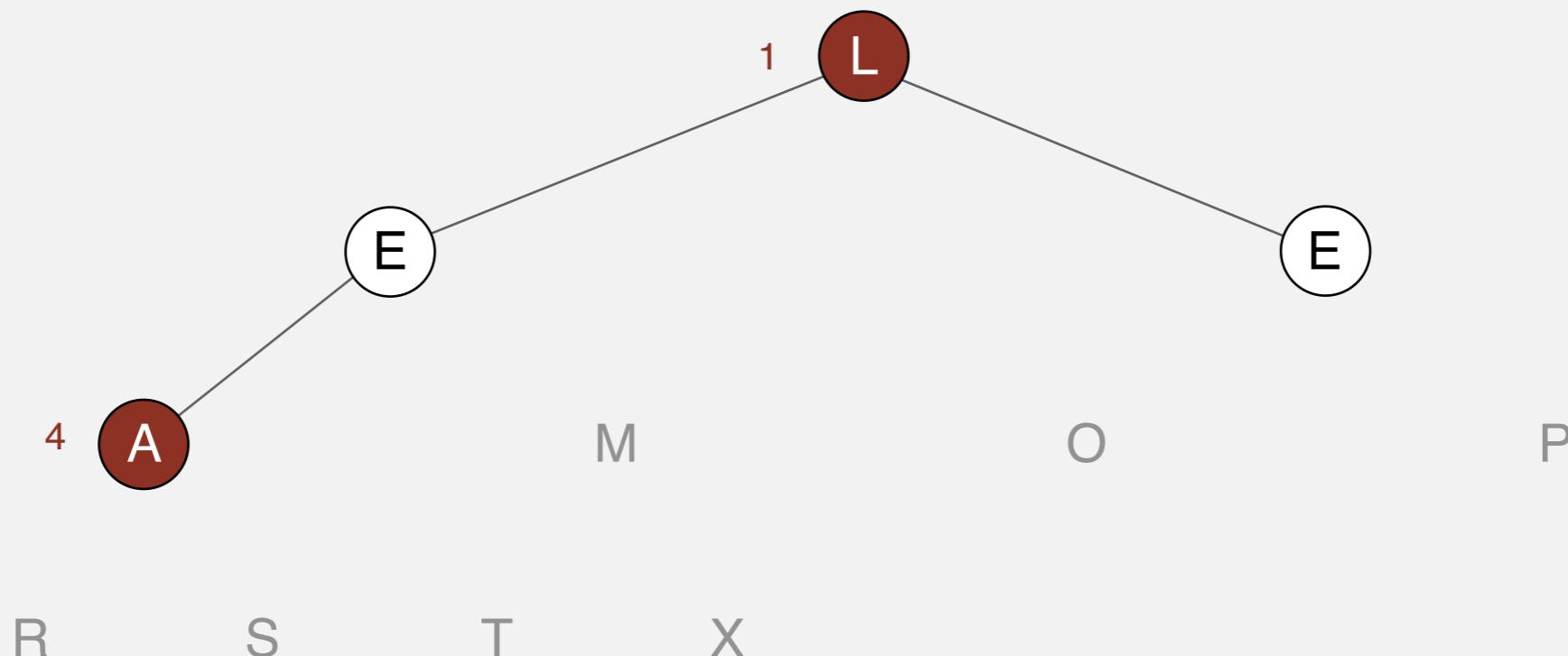
Heapsort

Sortdown. Repeatedly delete the largest remaining item.



Sortdown. Repeatedly delete the largest remaining item.

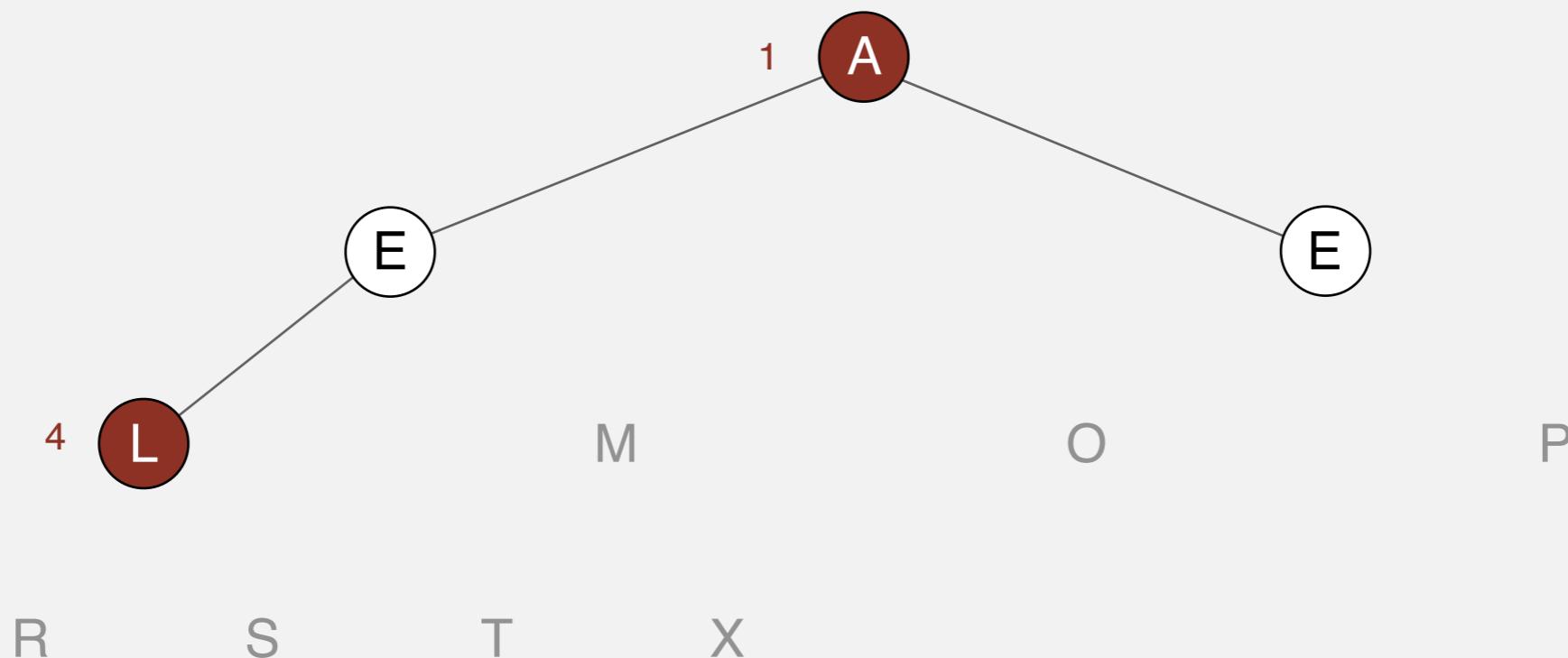
exchange 1 and 4



L	E	E	A	M	O	P	R	S	T	X
1			4							

Sortdown. Repeatedly delete the largest remaining item.

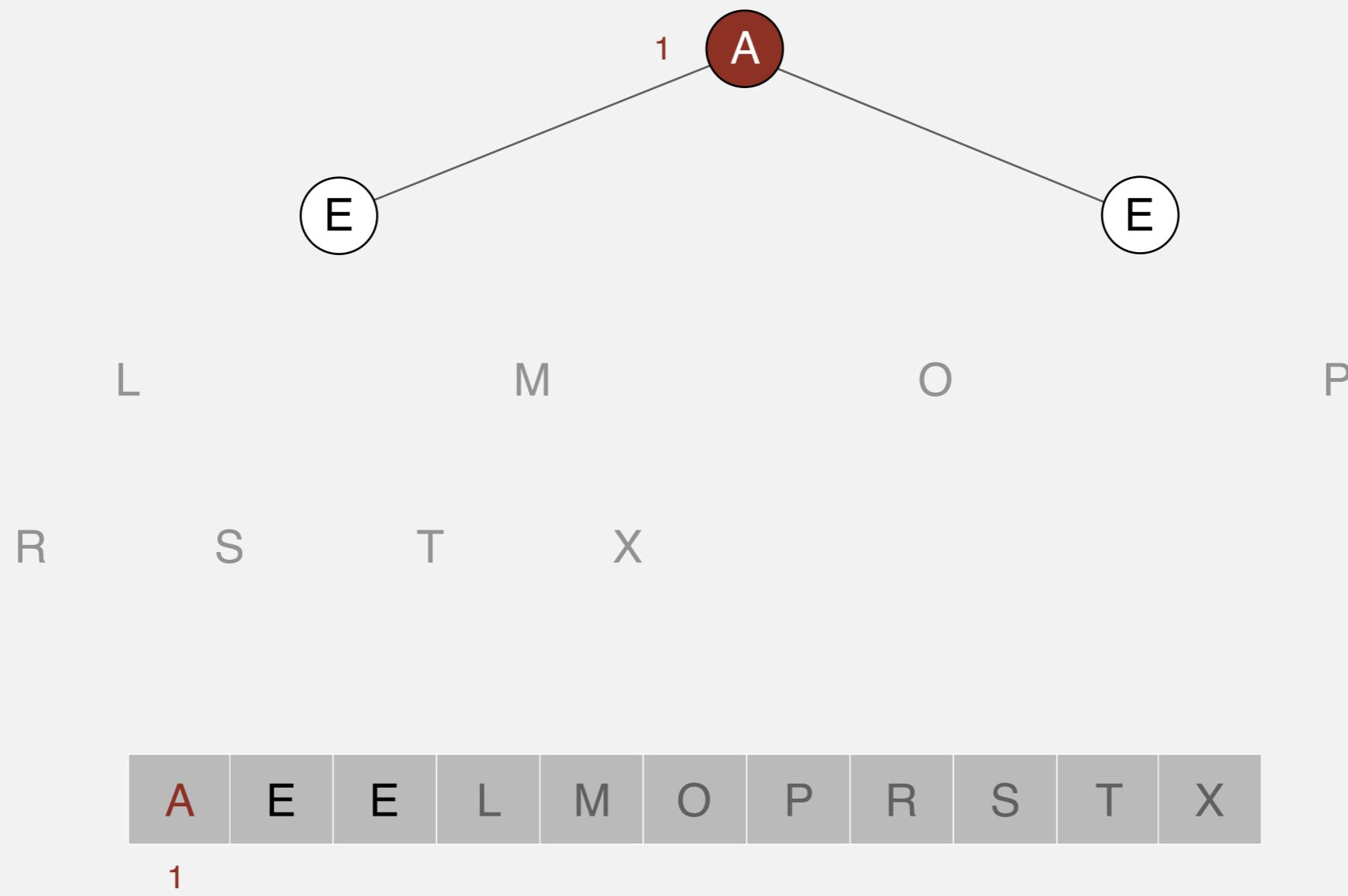
exchange 1 and 4



A	E	E	L	M	O	P	R	S	T	X
1			4							

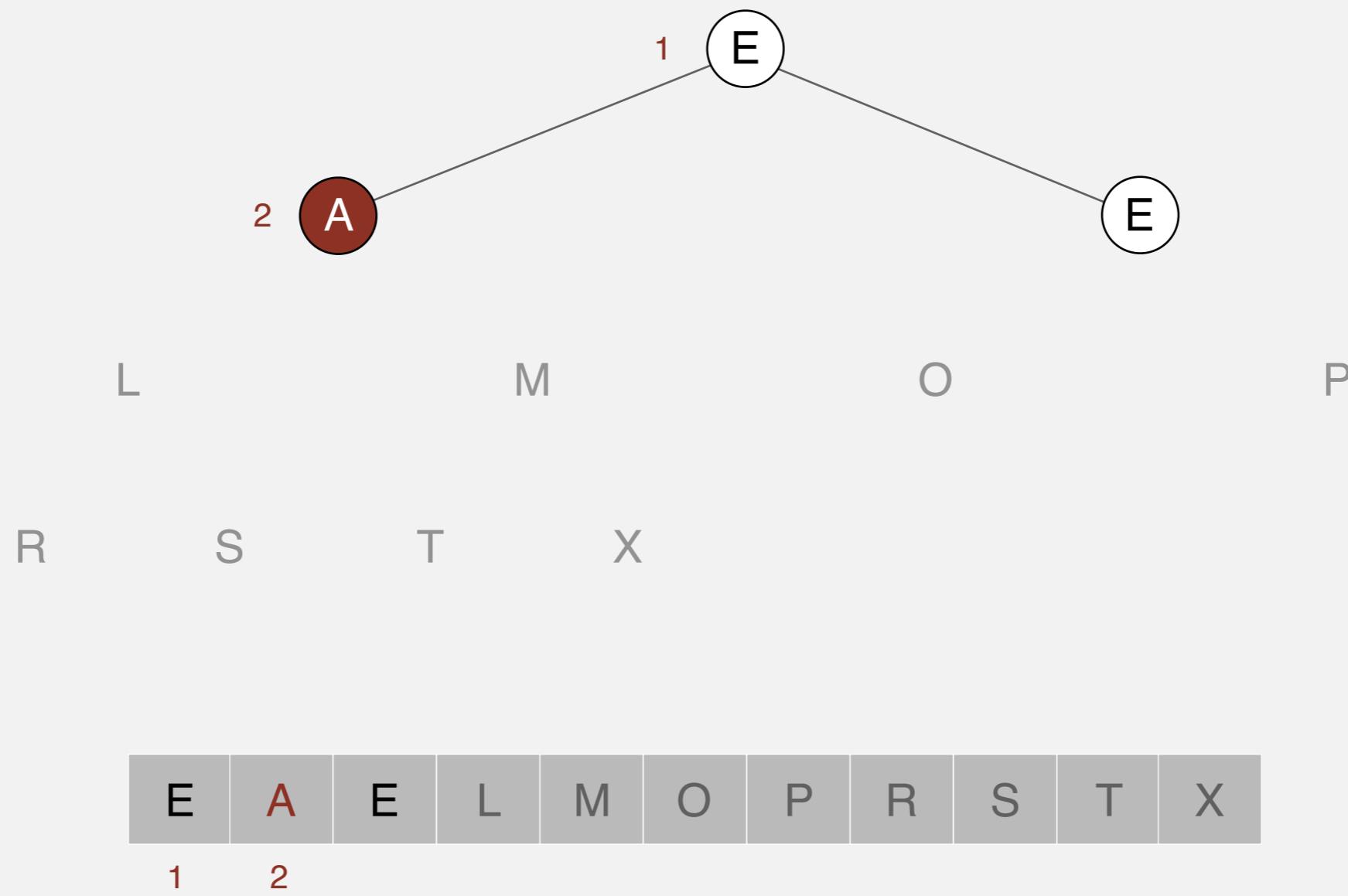
Sortdown. Repeatedly delete the largest remaining item.

sink 1



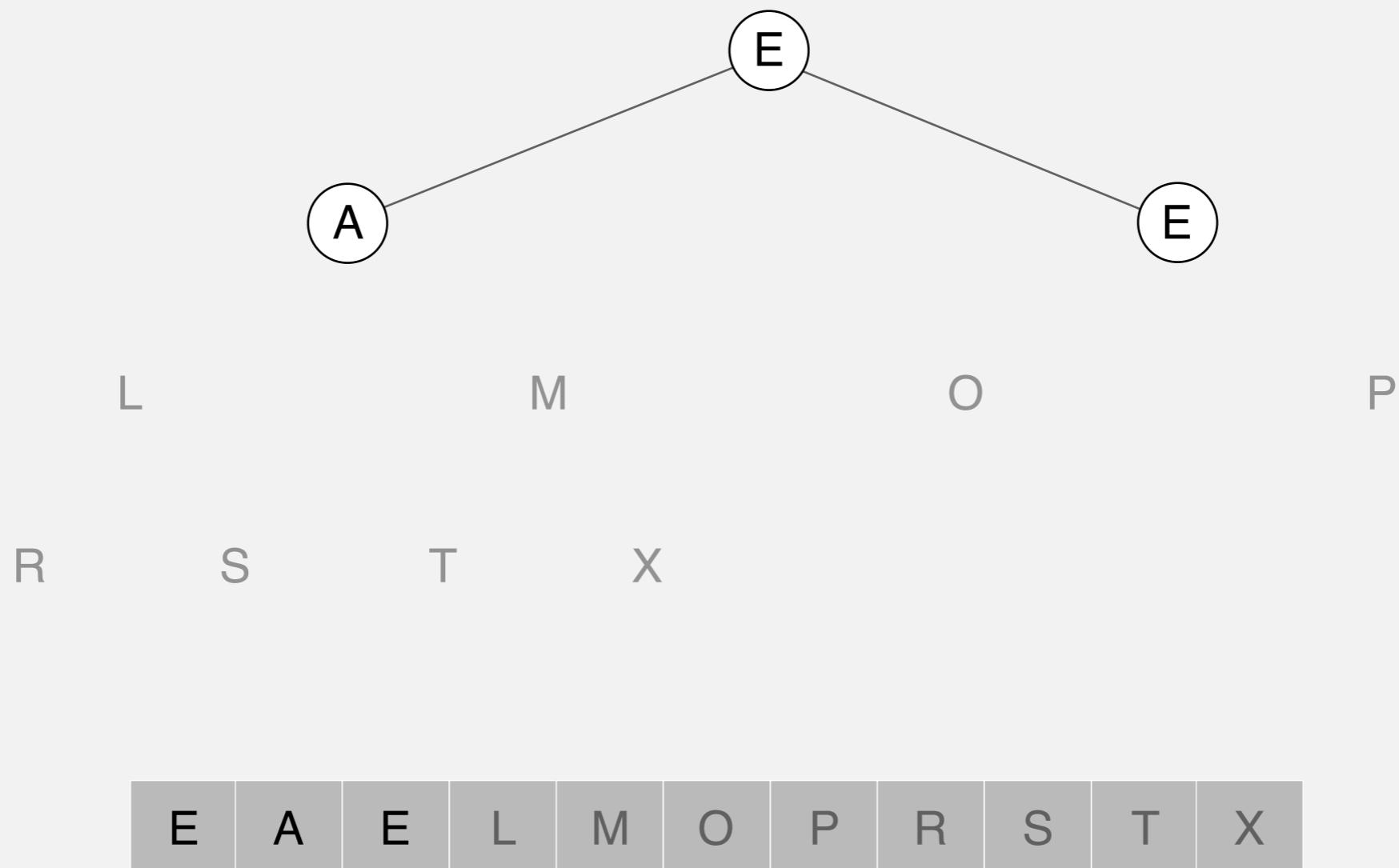
Sortdown. Repeatedly delete the largest remaining item.

sink 1



Heapsort

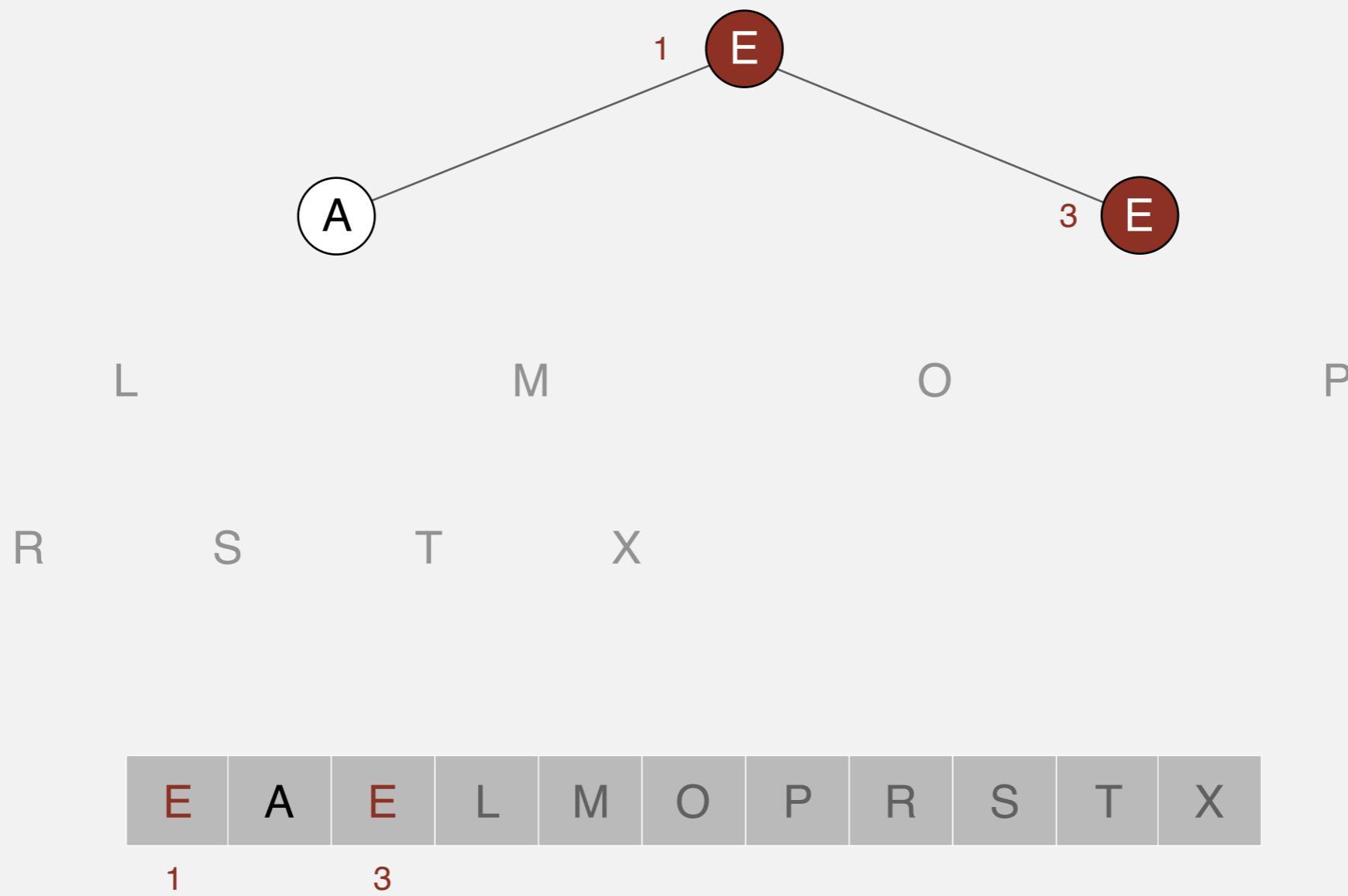
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

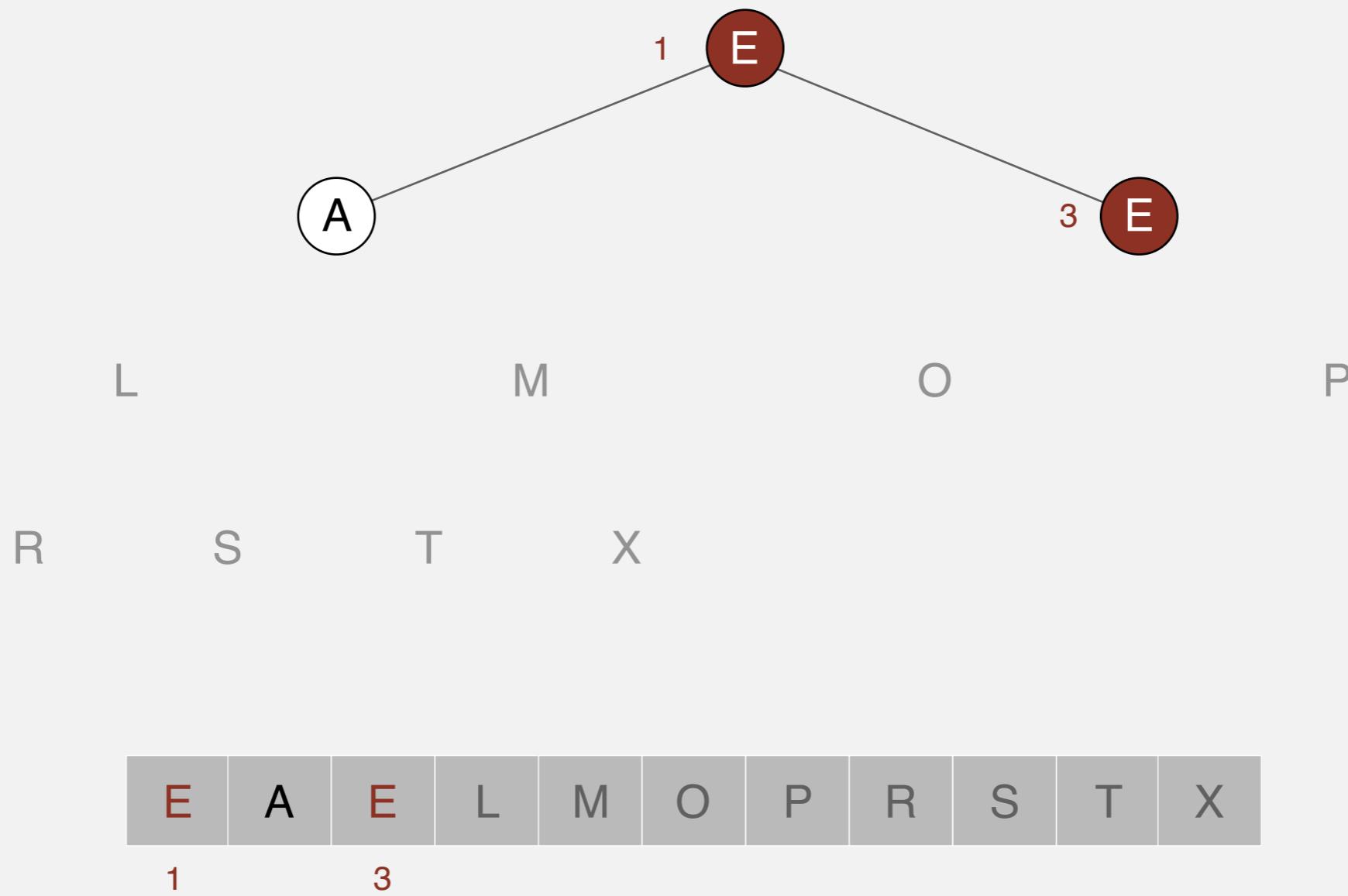
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3



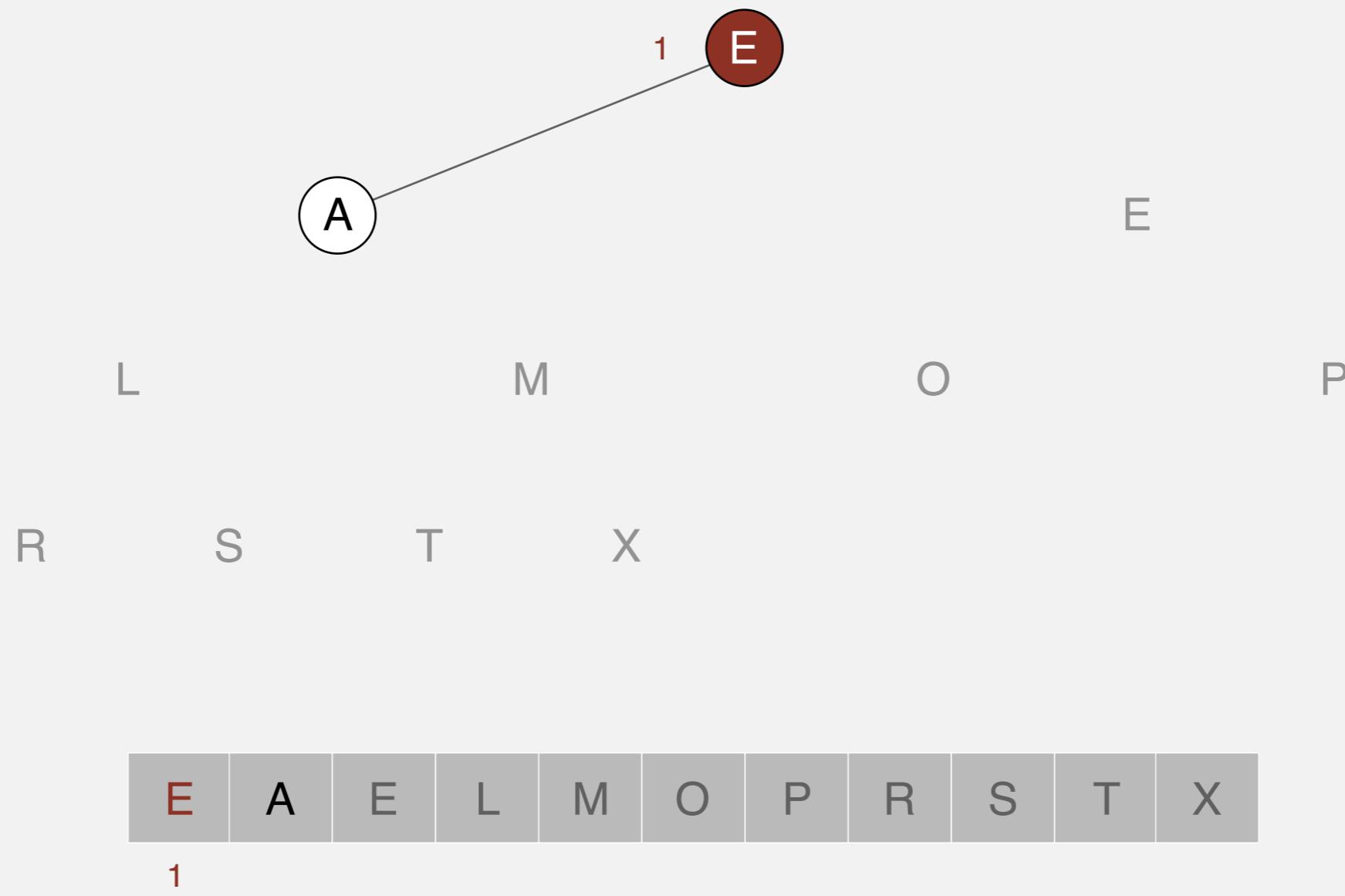
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3



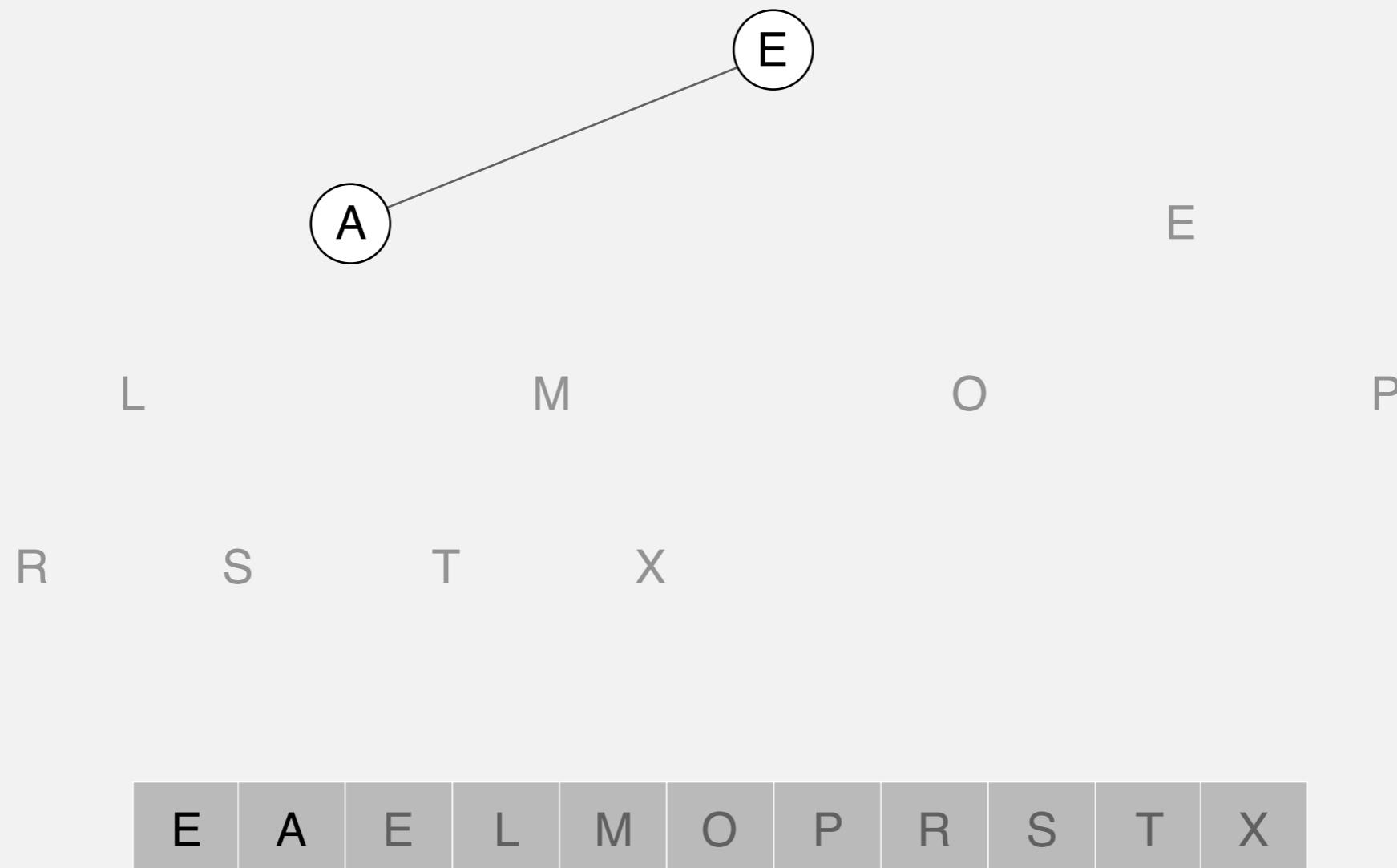
Sortdown. Repeatedly delete the largest remaining item.

sink 1



Heapsort

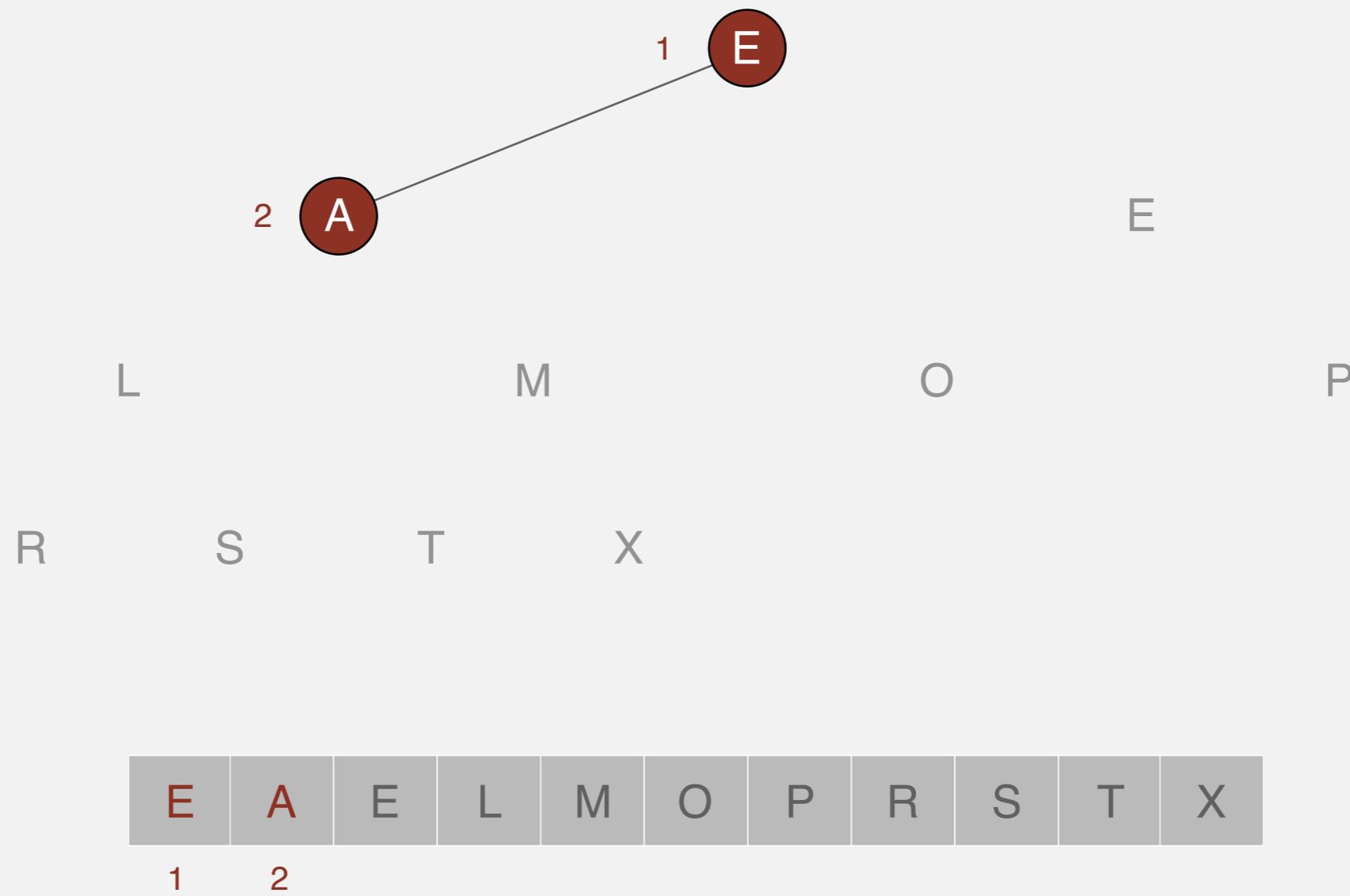
Sortdown. Repeatedly delete the largest remaining item.



Heapsort

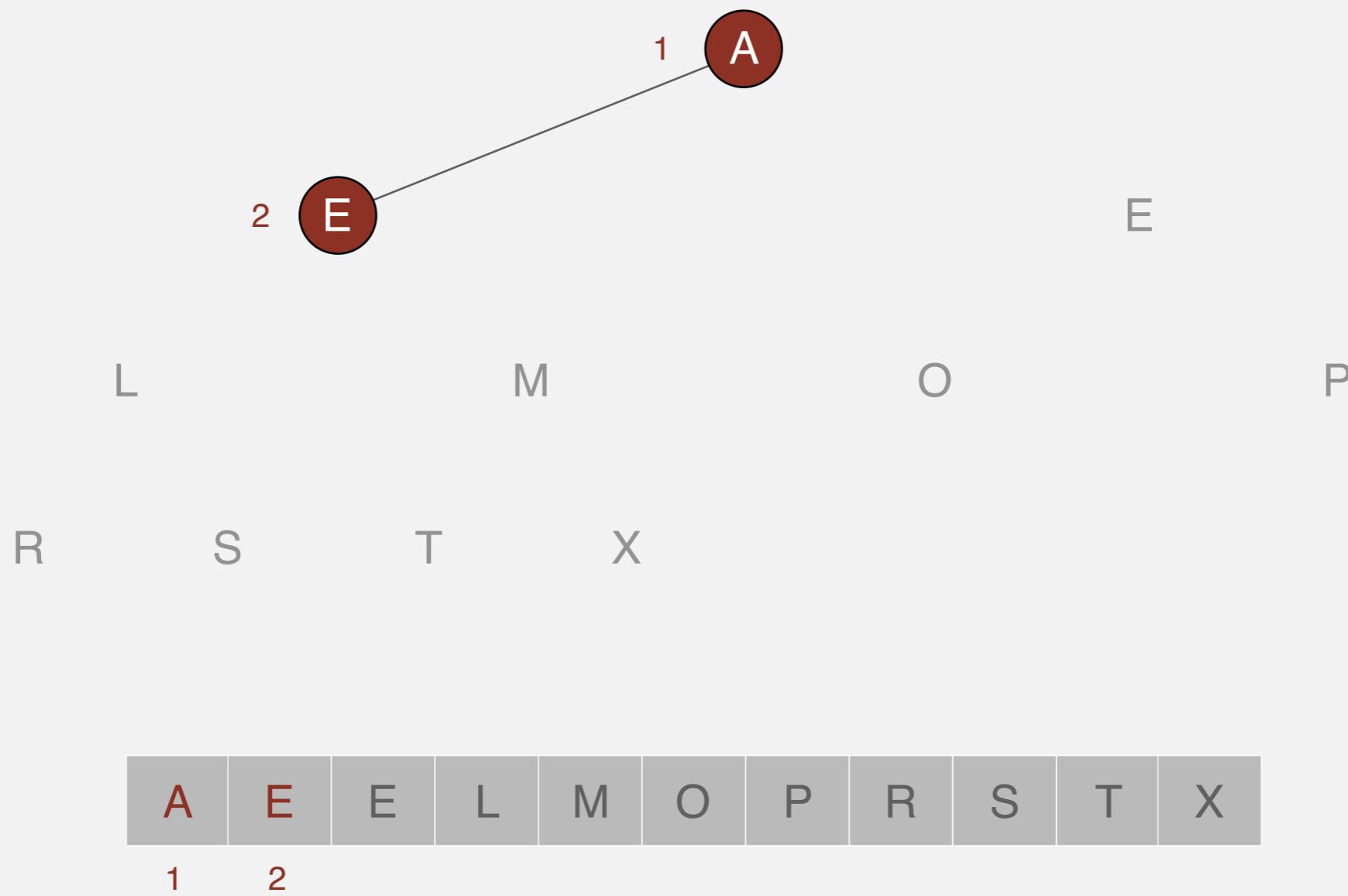
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2



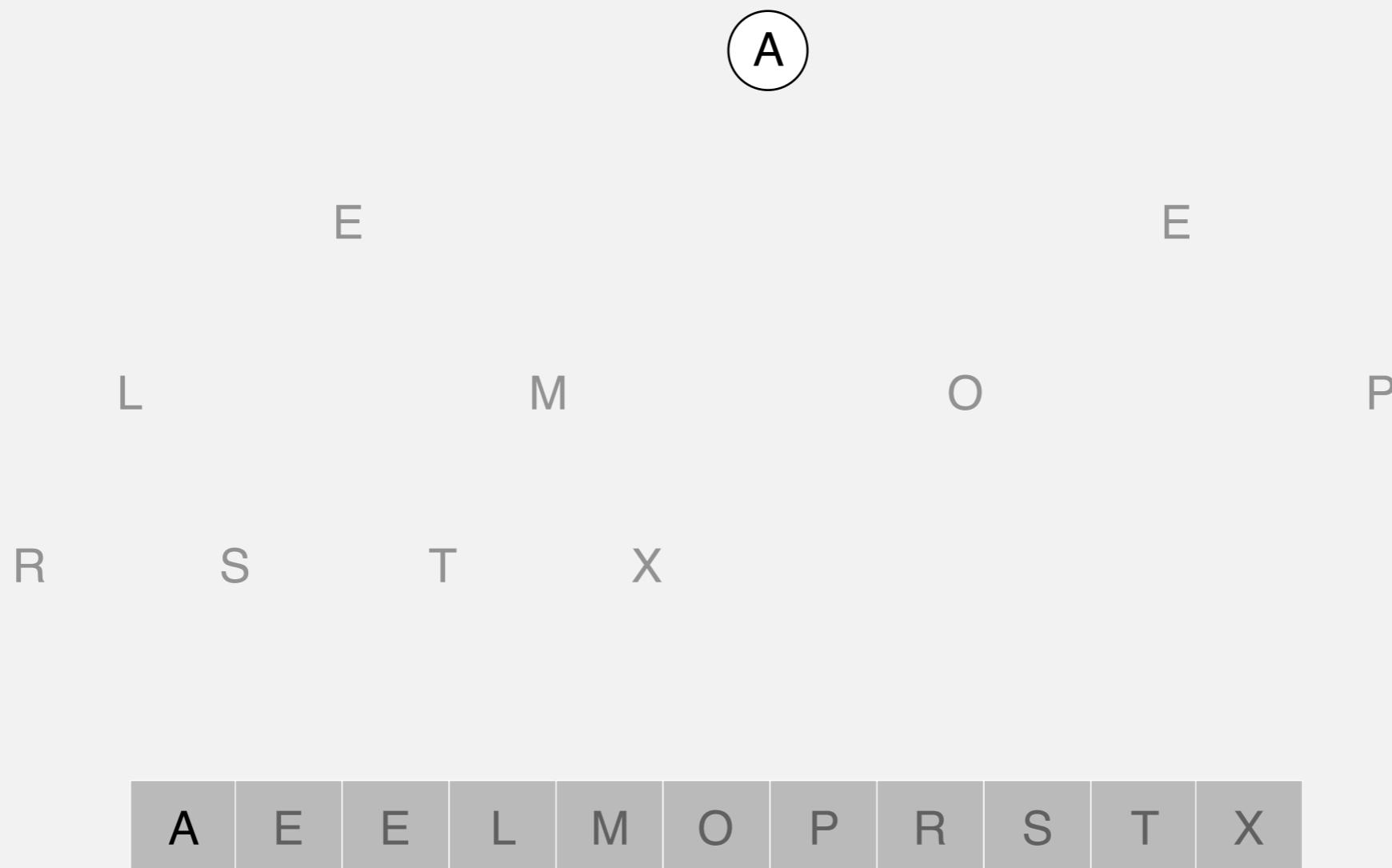
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2



Heapsort

Sortdown. Repeatedly delete the largest remaining item.



Sortdown. Repeatedly delete the largest remaining item.

end of sortdown phase



Heapsort

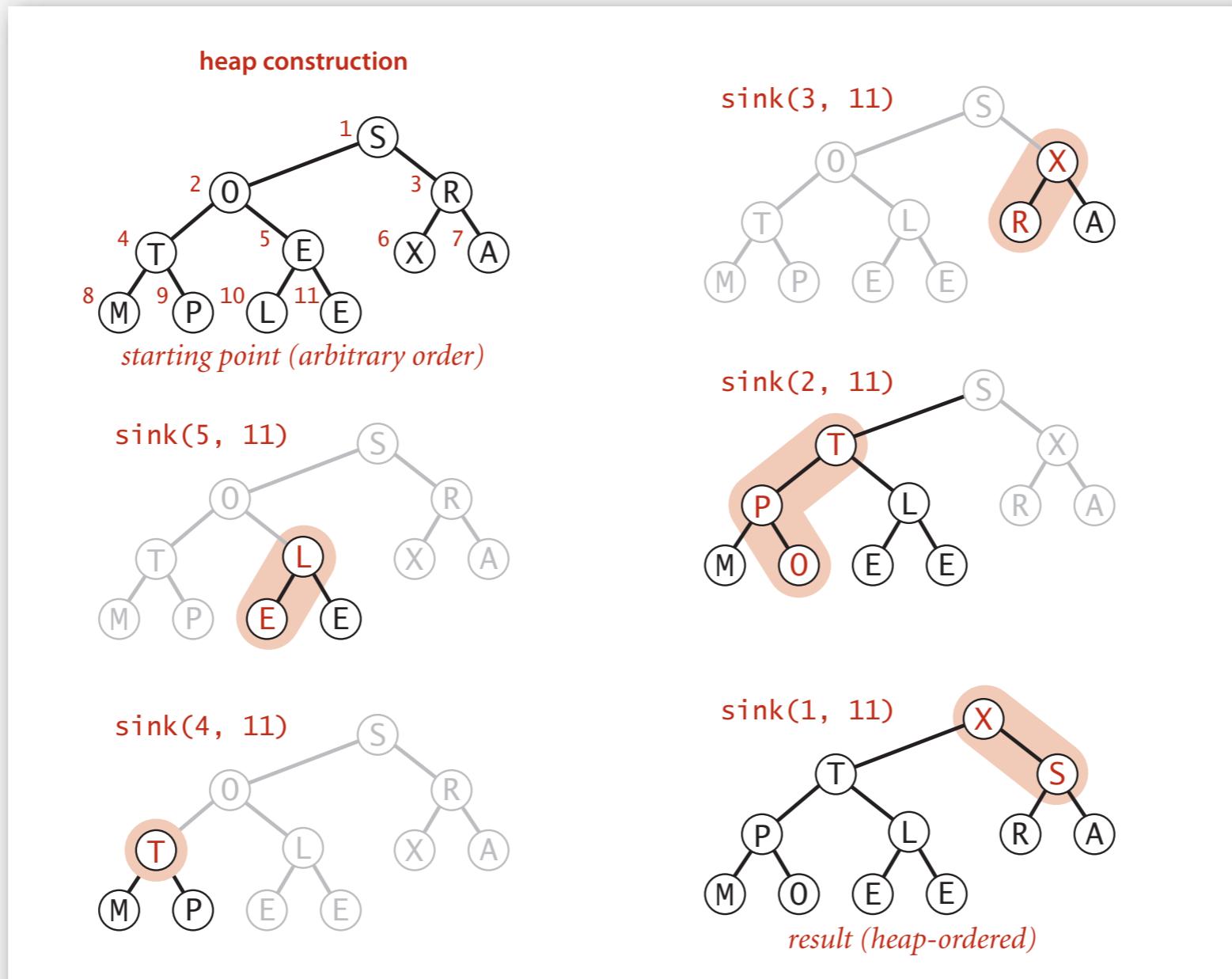
Ending point. Array in sorted order.



Heapsort: heap construction

First pass. Build heap using bottom-up method.

```
for (int k = N/2; k >= 1; k--)  
    sink(a, k, N);
```

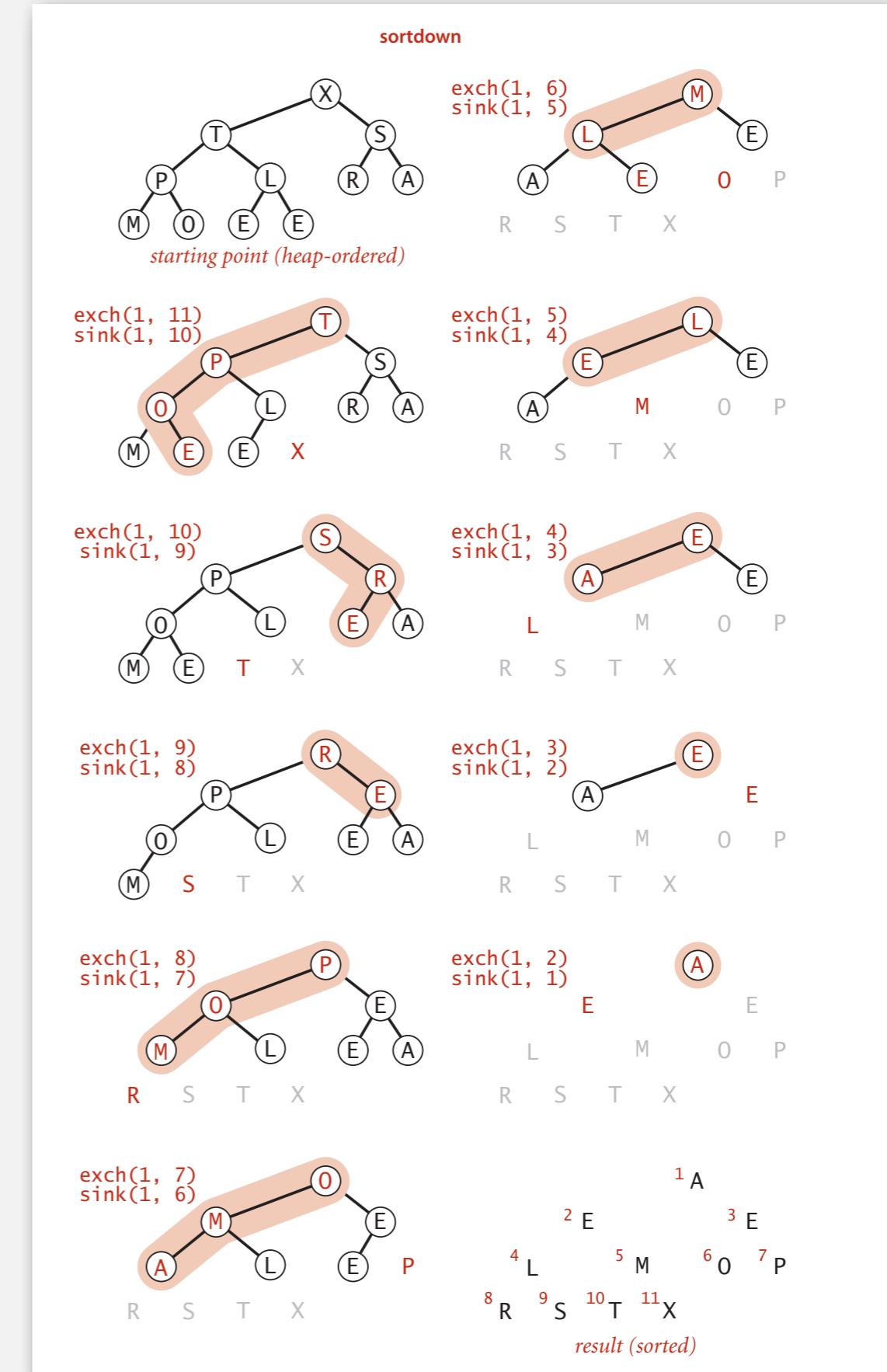


Heapsort: sortdown

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```



Heapsort: Java implementation

```
public class Heap
{
    public static void sort(Comparable[] pq)
    {
        int N = pq.length;
        for (int k = N/2; k >= 1; k--)
            sink(pq, k, N);
        while (N > 1)
        {
            exch(pq, 1, N);
            sink(pq, 1, --N);
        }
    }

    private static void sink(Comparable[] pq, int k, int N)
    { /* as before */ }

    private static boolean less(Comparable[] pq, int i, int j)
    { /* as before */ }

    private static void exch(Comparable[] pq, int i, int j)
    { /* as before */ }
}

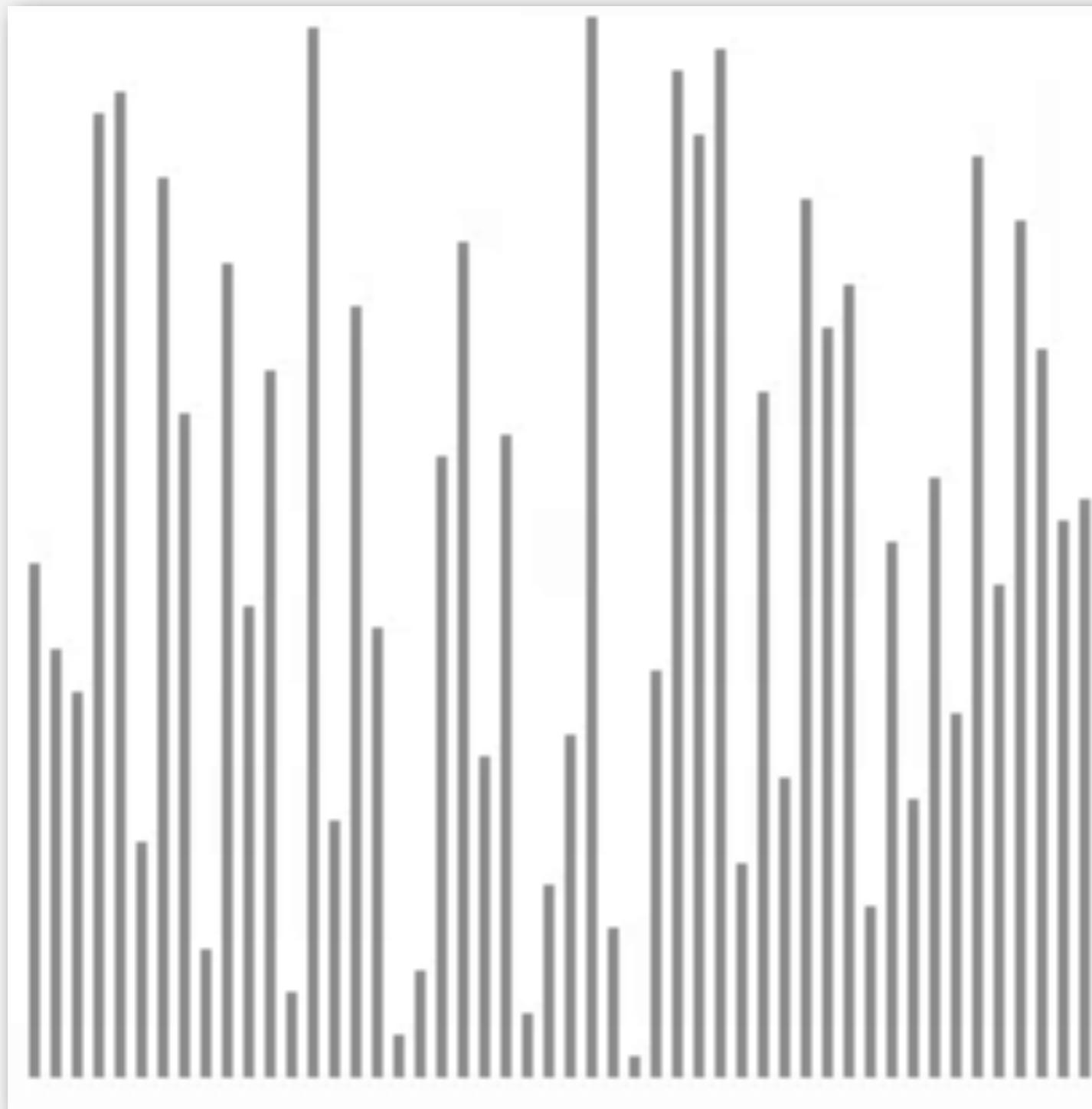
but convert from
1-based indexing to
0-base indexing
```

Heapsort: trace

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>		S	0	R	T	E	X	A	M	P	L	E	
11	5	S	0	R	T	L	X	A	M	P	E	E	
11	4	S	0	R	T	L	X	A	M	P	E	E	
11	3	S	0	X	T	L	R	A	M	P	E	E	
11	2	S	T	X	P	L	R	A	M	O	E	E	
11	1	X	T	S	P	L	R	A	M	O	E	E	
<i>heap-ordered</i>		X	T	S	P	L	R	A	M	O	E	E	
10	1	T	P	S	O	L	R	A	M	E	E	X	
9	1	S	P	R	O	L	E	A	M	E	T	X	
8	1	R	P	E	O	L	E	A	M	S	T	X	
7	1	P	O	E	M	L	E	A	R	S	T	X	
6	1	O	M	E	A	L	E	P	R	S	T	X	
5	1	M	L	E	A	E	O	P	R	S	T	X	
4	1	L	E	E	A	M	O	P	R	S	T	X	
3	1	E	A	E	L	M	O	P	R	S	T	X	
2	1	E	A	E	L	M	O	P	R	S	T	X	
1	1	A	E	E	L	M	O	P	R	S	T	X	
<i>sorted result</i>		A	E	E	L	M	O	P	R	S	T	X	
Heapsort trace (array contents just after each sink)													

Heapsort animation

50 random items



<http://www.sorting-algorithms.com/heap-sort>

Heapsort: mathematical analysis

Proposition. Heap construction uses fewer than $2N$ compares and exchanges.

Proposition. Heapsort uses at most $2N \lg N$ compares and exchanges.

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← $N \log N$ worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, **but:**

- Inner loop longer than quicksort's.
- Makes poor use of cache memory.
- Not stable.

Sorting algorithms: summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2/2$	$N^2/2$	$N^2/2$	N exchanges
insertion	x	x	$N^2/2$	$N^2/4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2/2$	$2N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2/2$	$2N \ln N$	N	improves quicksort in presence of duplicate keys
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
heap	x		$2N \lg N$	$2N \lg N$	$N \lg N$	$N \log N$ guarantee, in-place
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ELEMENTARY SEARCH ALGORITHMS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ **Symbol Tables**
- ▶ **API**
- ▶ **Elementary implementations**
- ▶ **Ordered operations**

SYMBOL TABLES

- ▶ API
- ▶ Elementary implementations
- ▶ Ordered operations

Symbol tables

Key-value pair abstraction.

- Insert a value with specified key.
- Given a key, search for the corresponding value.

Ex. DNS lookup.

- Insert URL with specified IP address.
- Given URL, find corresponding IP address.

URL	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

↑
key

↑
value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address given URL	URL	IP address
reverse DNS	find URL given IP address	IP address	URL
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```
public class ST<Key, Value>
```

ST()	<i>create a symbol table</i>
void put(Key key, Value val)	<i>put key-value pair into the table (remove key from table if value is null)</i>
Value get(Key key)	<i>value paired with key (null if key is absent)</i>
void delete(Key key)	<i>remove key (and its value) from table</i>
boolean contains(Key key)	<i>is there a value paired with key?</i>
boolean isEmpty()	<i>is the table empty?</i>
int size()	<i>number of key-value pairs in the table</i>
Iterable<Key> keys()	<i>all the keys in the table</i>

Conventions

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

Intended consequences.

- Easy to implement `contains()`.

```
public boolean contains(Key key)
{   return get(key) != null; }
```

- Can implement lazy version of `delete()`.

```
public void delete(Key key)
{   put(key, null); }
```

Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are Comparable, use compareTo().
- Assume keys are any generic type, use equals() to test equality.
- Assume keys are any generic type, use equals() to test equality; use hashCode() to scramble key.

specify Comparable in API.

built-in to Java
(stay tuned)

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: String, Integer, Double, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

Equality test

All Java classes inherit a method `equals()`.

Java requirements. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

equivalence
relation

Default implementation. (`x == y`)

do `x` and `y` refer to
the same object?

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, ...

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy.

```
public class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {

        if (this.day != that.day) return false;
        if (this.month != that.month) return false; ←
        if (this.year != that.year) return false;
        return true;
    }
}
```

check that all significant
fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

Safer to use `equals()` with inheritance
if fields in extending class contribute to
`equals()` the symmetry violated

```
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y)
    {
        if (y == this) return true;           ← optimize for true object equality
        if (y == null) return false;         ← check for null
        if (y.getClass() != this.getClass())
            return false;
        Date that = (Date) y;
        if (this.day != that.day) return false;
        if (this.month != that.month) return false; ← check that all significant
        if (this.year != that.year) return false;   fields are the same
        return true;
    }
}
```

must be `Object`.

cast is guaranteed to succeed

check that all significant
fields are the same

Equals design

"Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type and cast.
- Compare each significant field:
 - if field is a primitive type, use `==`
 - if field is an object, use `equals()` ← apply rule recursively
 - if field is an array, apply to each entry ← alternatively, use `Arrays.equals(a, b)` or
`Arrays.deepEquals(a, b)`,
but not `a.equals(b)`

Best practices.

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Only use necessary fields, e.g. a webpage is best defined by URL, not number of views.
- Make `compareTo()` consistent with `equals()`.



`x.equals(y)` if and only if `(x.compareTo(y) == 0)`

ST test client for traces

Build ST by associating value i with i^{th} string from standard input.

```
public static void main(String[] args)
{
    ST<String, Integer> st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}
```

output

The order of output
depends on the
underlying data
structure!

keys	S	E	A	R	C	H	E	X	A	M	P	L	E
values	0	1	2	3	4	5	6	7	8	9	10	11	12

A	8
C	4
E	12
H	5
L	11
M	9
P	10
R	3
S	0
X	7

ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.txt  
it was the best of times  
it was the worst of times  
it was the age of wisdom  
it was the age of foolishness  
it was the epoch of belief  
it was the epoch of incredulity  
it was the season of light  
it was the season of darkness  
it was the spring of hope  
it was the winter of despair
```

```
% java FrequencyCounter 1 < tinyTale.txt  
it 10
```

```
% java FrequencyCounter 8 < tale.txt  
business 122
```

```
% java FrequencyCounter 10 < leipzig1M.txt  
government 24763
```

tiny example
(60 words, 20 distinct)

real example
(135,635 words, 10,769 distinct)

real example
(21,191,455 words, 534,580 distinct)

Frequency counter implementation

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]);
        ST<String, Integer> st = new ST<String, Integer>();           ← create ST
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();                         ← read string and
            if (word.length() < minlen) continue;                      ← ignore short strings
            if (!st.contains(word)) st.put(word, 1);                  ← update frequency
            else st.put(word, st.get(word) + 1);
        }
        String max = "";
        st.put(max, 0);
        for (String word : st.keys())
            if (st.get(word) > st.get(max))
                max = word;
        StdOut.println(max + " " + st.get(max));
    }
}
```

← print a string with max freq

SYMBOL TABLES

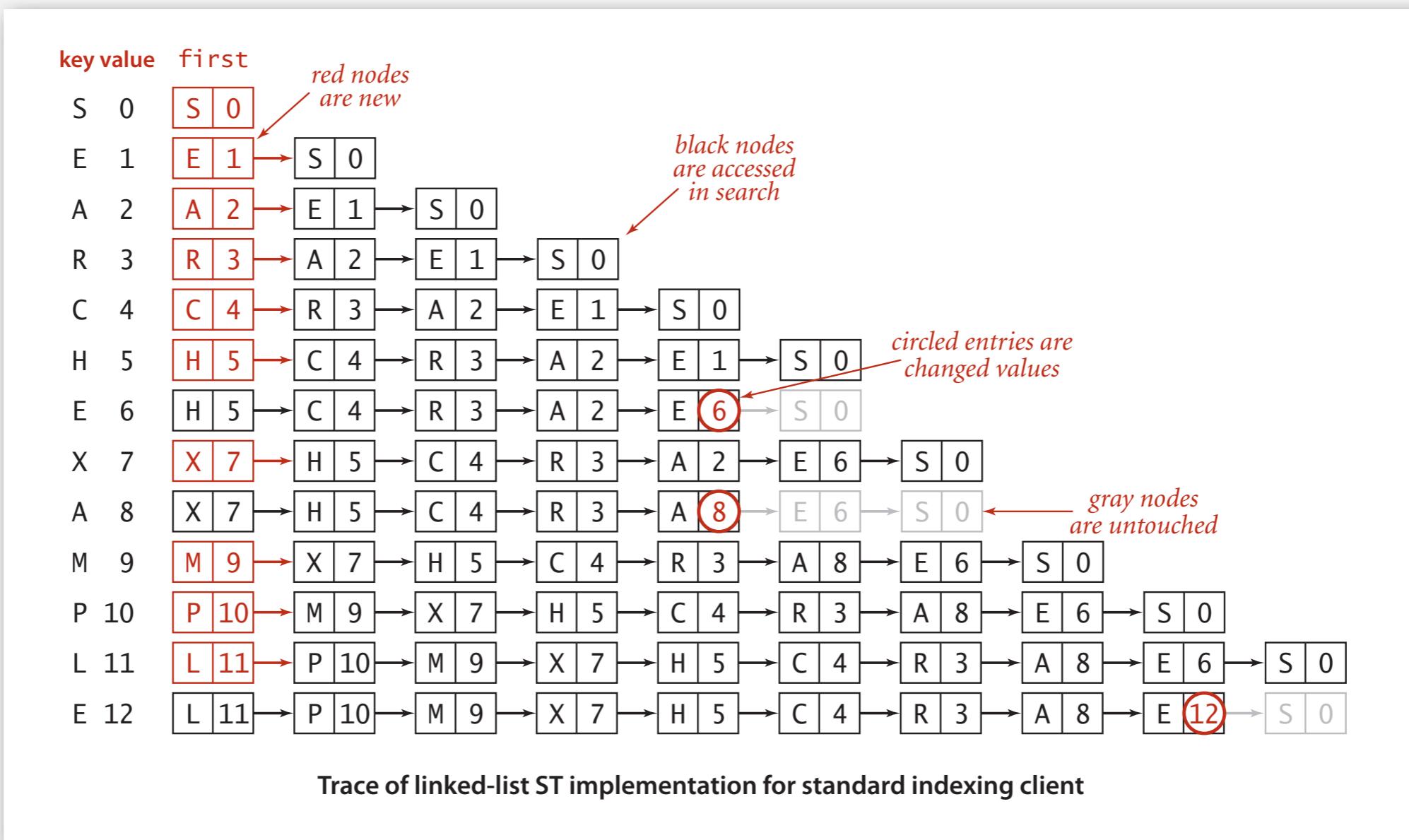
- ▶ API
- ▶ Elementary implementations
- ▶ Ordered operations

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

Insert. Scan through all keys until find a match; if no match add to front.



Elementary ST implementations: summary

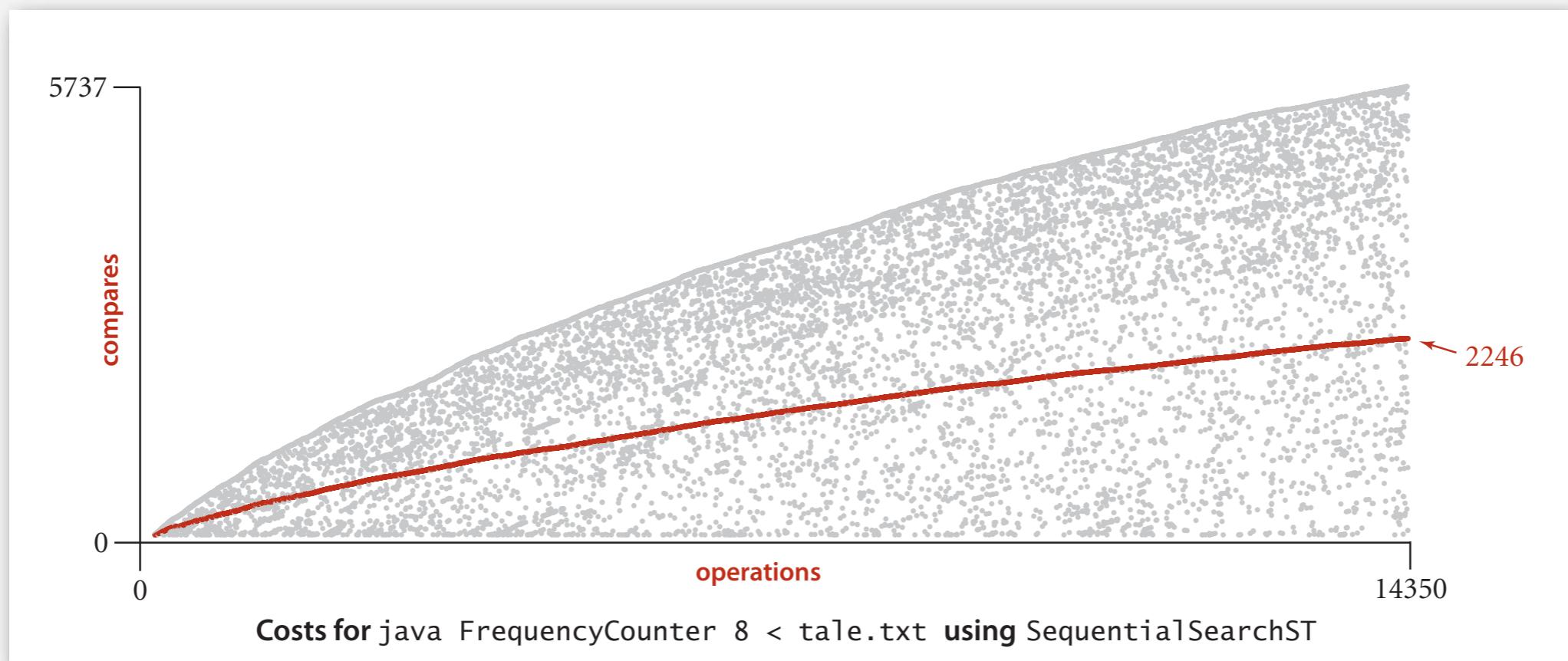
ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	<code>equals()</code>

Must search first
to avoid duplicates

Challenge. Efficient implementations of both search and insert.

Elementary ST implementations: summary

ST implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	<code>equals()</code>



Grey data points are observed costs for i^{th} operation, reds are their averages

Challenge. Efficient implementations of both search and insert.

Binary search

Data structure. Maintain an ordered array of key-value pairs.

Rank helper function. How many keys $< k$?

keys []										
successful search for P	0	1	2	3	4	5	6	7	8	9
lo hi m	0 9 4	A C E H L M P R S X								
	5 9 7	A C E H L M P R S X								
	5 6 5	A C E H L M P R S X								
	6 6 6	A C E H L M P R S X								
entries in black are $a[lo..hi]$										
entry in red is $a[m]$										
loop exits with $keys[m] = P$: return 6										
unsuccessful search for Q	lo hi m	0 9 4	A C E H L M P R S X							
	5 9 7	A C E H L M P R S X								
	5 6 5	A C E H L M P R S X								
	7 6 6	A C E H L M P R S X								
loop exits with $lo > hi$: return 7										
Trace of binary search for rank in an ordered array										

Binary search: Java implementation

```
public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}
```

```
private int rank(Key key)                                number of keys < key
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else if (cmp == 0) return mid;
    }
    return lo;
}
```

Binary search: mathematical analysis

Proposition. Binary search uses $\sim \lg N$ compares to search any array of size N .

Pf. $T(N) \equiv$ number of compares to binary search in a sorted array of size N .

\uparrow
left or right half

$$\leq T(\lfloor N/2 \rfloor) + 1$$

Recall lecture 2.

Binary search: trace of standard indexing client

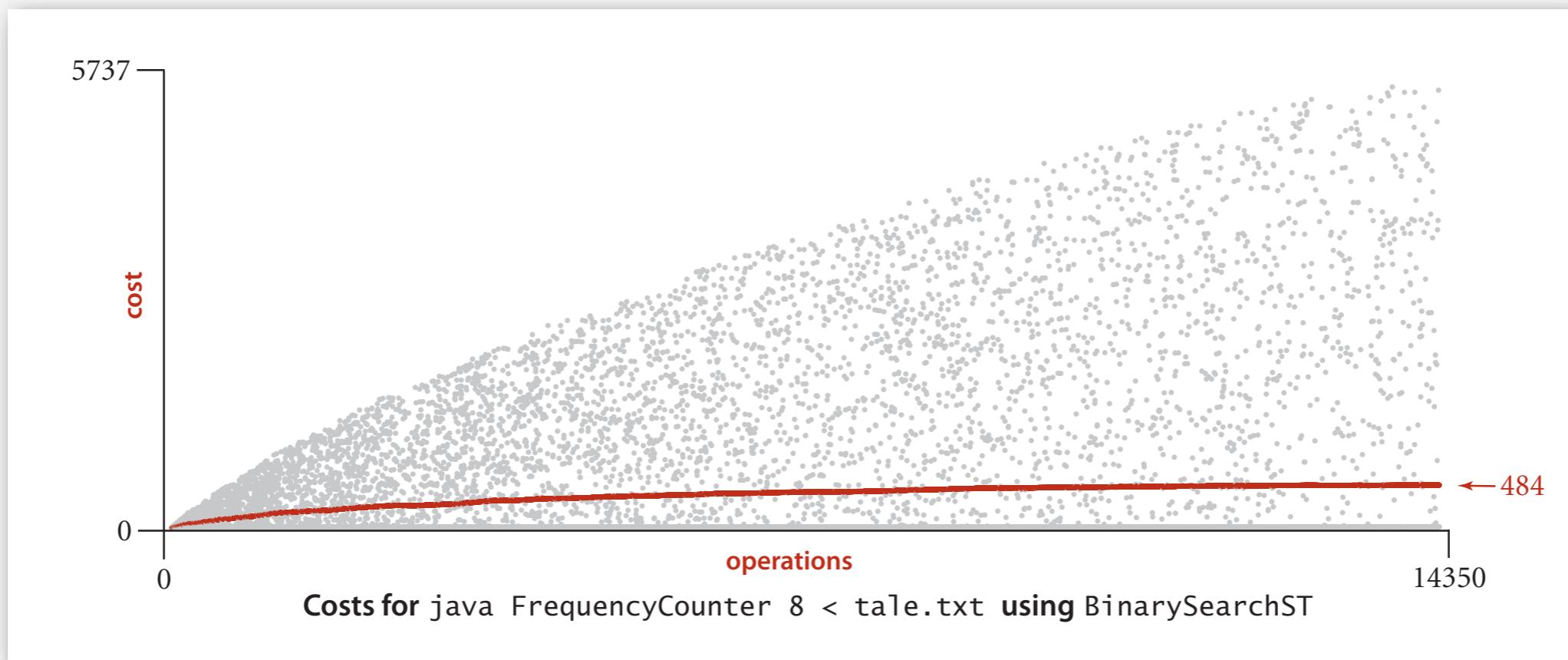
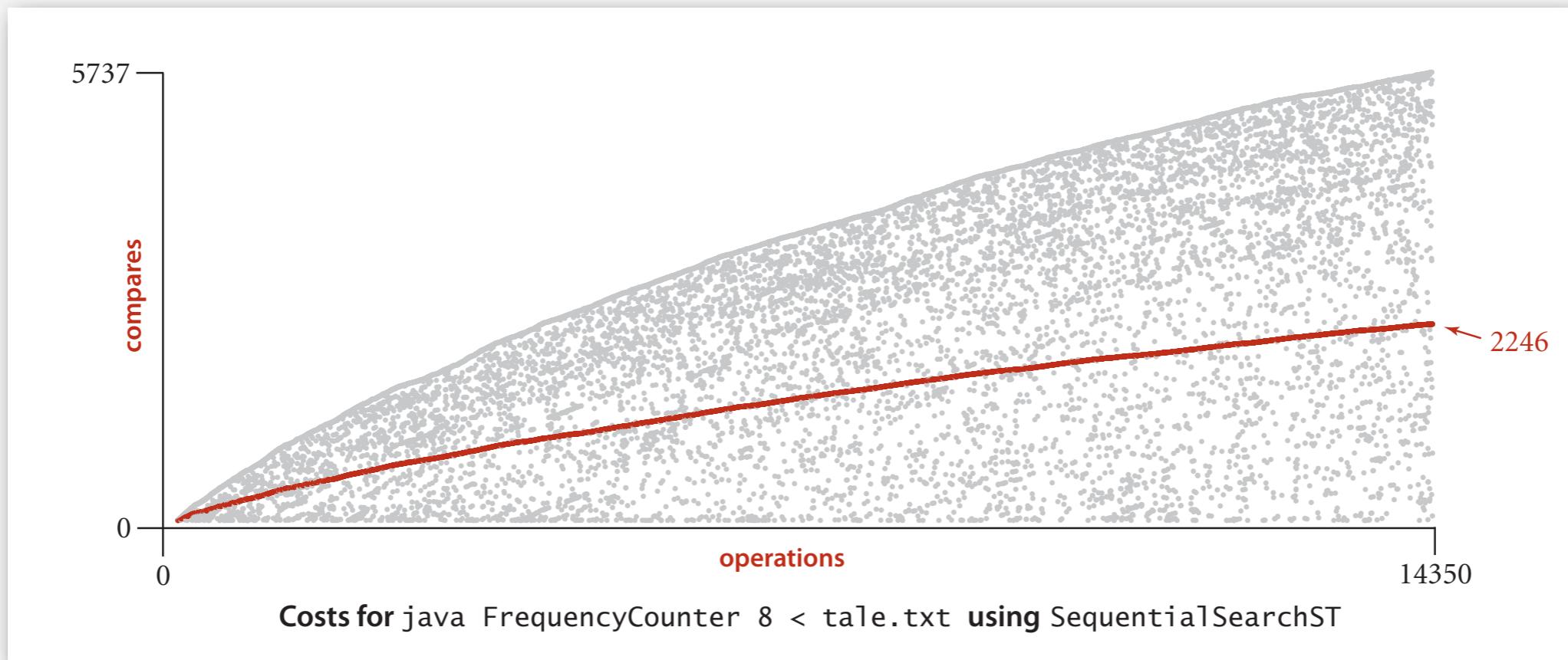
Problem. To insert, need to shift all greater keys over.

	keys[]										N	vals[]										
key	value	0	1	2	3	4	5	6	7	8	9		0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

Annotations:

- entries in red were inserted*: Points to the 'R' entry in row 3, column 4.
- entries in gray did not move*: Points to the 'H' entry in row 5, column 4.
- entries in black moved to the right*: Points to the '3' entry in row 4, column 5.
- circled entries are changed values*: Points to the circled '6' entry in row 6, column 4.

Elementary ST implementations: frequency counter



Elementary ST implementations: summary

ST implementation	worst-case cost (after N inserts)		average case (after N random inserts)		ordered iteration?	key interface
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N / 2	N	no	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	N / 2	yes	<code>compareTo()</code>

Challenge. Efficient implementations of both search and insert.

SYMBOL TABLES

- ▶ API
- ▶ Elementary implementations
- ▶ Ordered operations

Ordered symbol table API (Example Operations)

	<i>keys</i>	<i>values</i>
min()	→ 09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	→ Houston
get(09:00:13)	→ 09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00)	→ 09:03:13	Chicago
	09:10:11	Seattle
select(7)	→ 09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00)	→ 09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00)	→ 09:35:21	Chicago
	09:36:14	Seattle
max()	→ 09:37:44	Phoenix
size(09:15:00, 09:25:00)	is 5	
rank(09:10:25)	is 7	

Examples of ordered symbol-table operations

Ordered symbol table API

public class ST<Key extends Comparable<Key>, Value>	
ST()	<i>create an ordered symbol table</i>
void put(Key key, Value val)	<i>put key-value pair into the table (remove key from table if value is null)</i>
Value get(Key key)	<i>value paired with key (null if key is absent)</i>
void delete(Key key)	<i>remove key (and its value) from table</i>
boolean contains(Key key)	<i>is there a value paired with key?</i>
boolean isEmpty()	<i>is the table empty?</i>
int size()	<i>number of key-value pairs</i>
Key min()	<i>smallest key</i>
Key max()	<i>largest key</i>
Key floor(Key key)	<i>largest key less than or equal to key</i>
Key ceiling(Key key)	<i>smallest key greater than or equal to key</i>
int rank(Key key)	<i>number of keys less than key</i>
Key select(int k)	<i>key of rank k</i>
void deleteMin()	<i>delete smallest key</i>
void deleteMax()	<i>delete largest key</i>
int size(Key lo, Key hi)	<i>number of keys in [lo..hi]</i>
Iterable<Key> keys(Key lo, Key hi)	<i>keys in [lo..hi], in sorted order</i>
Iterable<Key> keys()	<i>all keys in the table, in sorted order</i>

Binary search: ordered symbol table operations summary

	sequential search	binary search
search	N	$\lg N$
insert	N	N
min / max	N	I
floor / ceiling	N	$\lg N$
rank	N	$\lg N$
select	N	I
ordered iteration	$N \log N$	N

The Problem:
Insert Operation

order of growth of the running time for ordered symbol table operations

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

BINARY SEARCH TREES

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ BSTs
- ▶ Ordered operations
- ▶ Deletion

Binary Search Tree (BST)

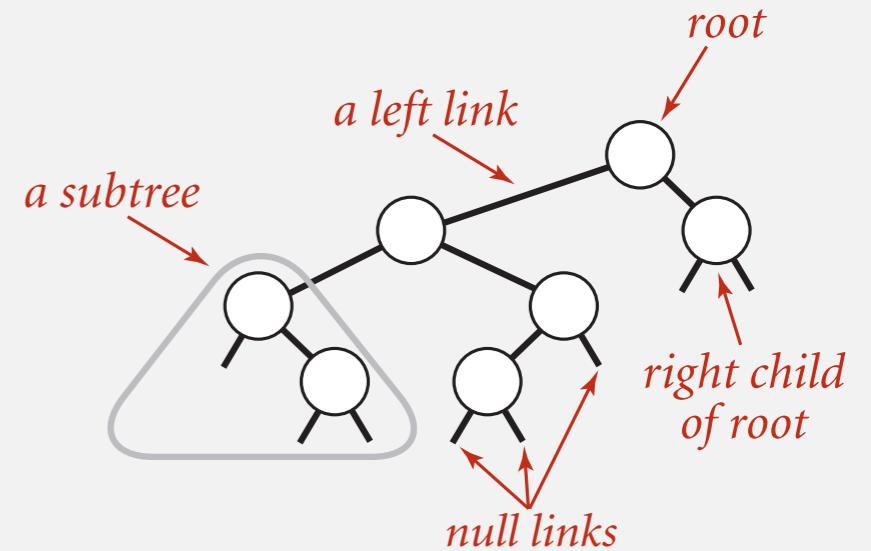
- Last lecture, we talked about binary search & linear search
 - One had high cost for reorganisation,
 - The other had high cost for searching
- In this lecture we will use Binary Trees, for searching
- Plan in a nutshell:
 - Assert a more strict property compared to the Heap-Property (in priority-queues), Remember what that was?
 - Know exactly which subtree to look for at each node

Binary search trees

Definition. A BST is a binary tree in **symmetric order**.

A binary tree is either:

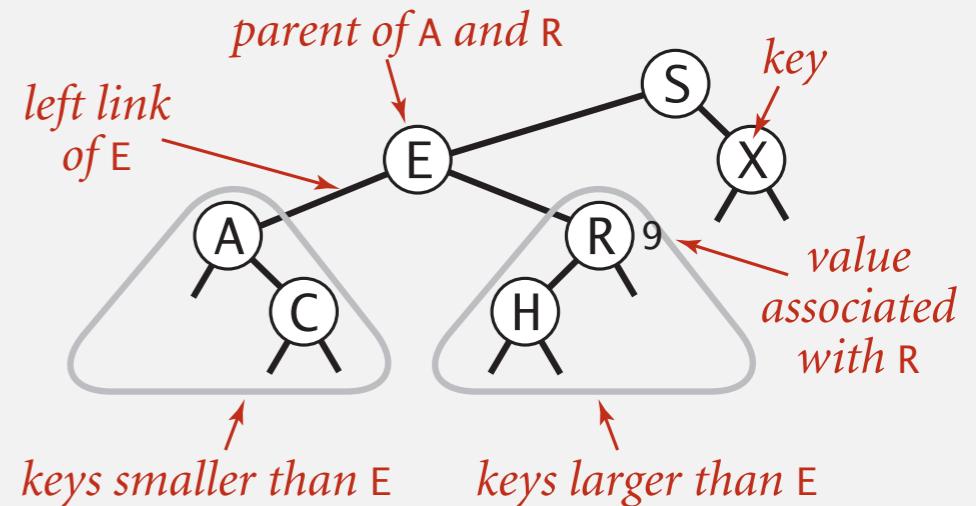
- Empty.
- Two disjoint binary trees (left and right).



Anatomy of a binary tree

Symmetric order. Each node has a **key**, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



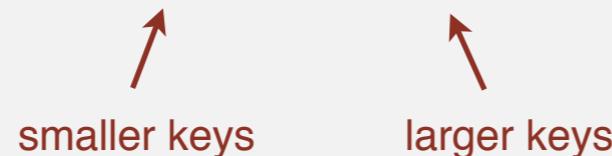
Anatomy of a binary search tree

BST representation in Java

Java definition. A BST is a reference to a root `Node`.

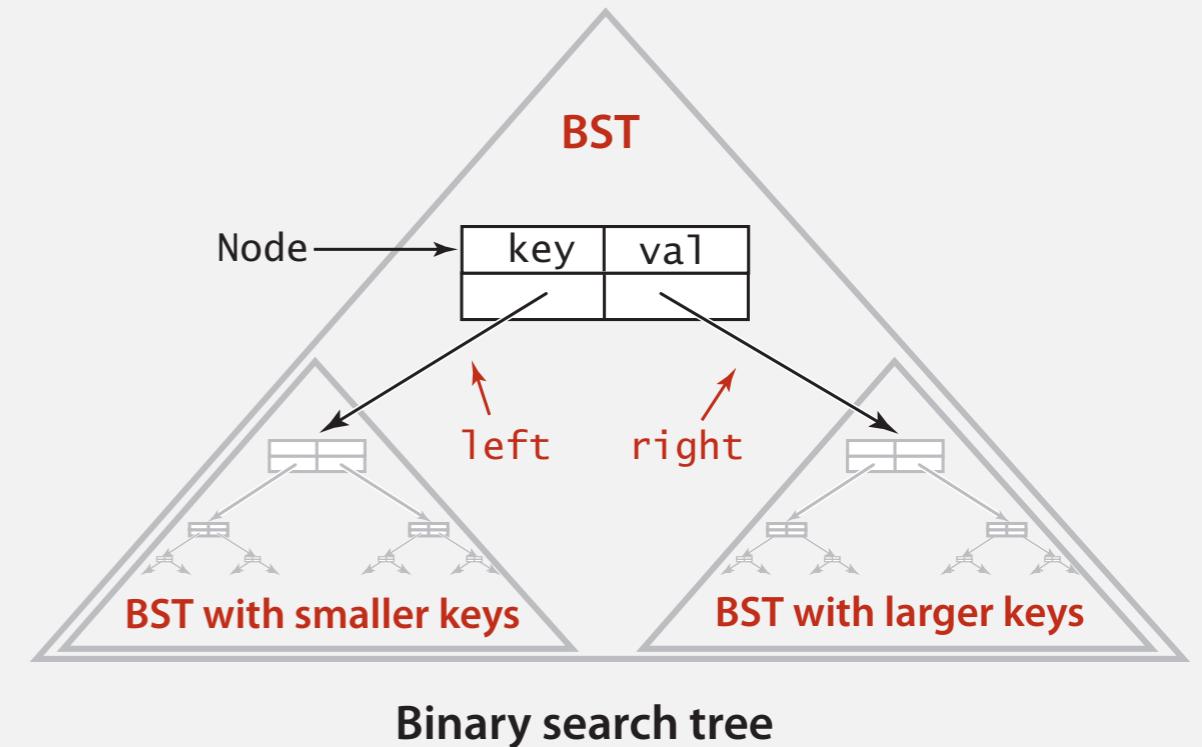
A `Node` is comprised of four fields:

- A `key` and a `value`.
- A reference to the left and right subtree.



```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

Key and value are generic types; Key is Comparable



Binary search tree

BST implementation (skeleton)

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

    private class Node
    { /* see previous slide */ }

    public void put(Key key, Value val)
    { /* see next slides */ }

    public Value get(Key key)
    { /* see next slides */ }

    public void delete(Key key)
    { /* see next slides */ }

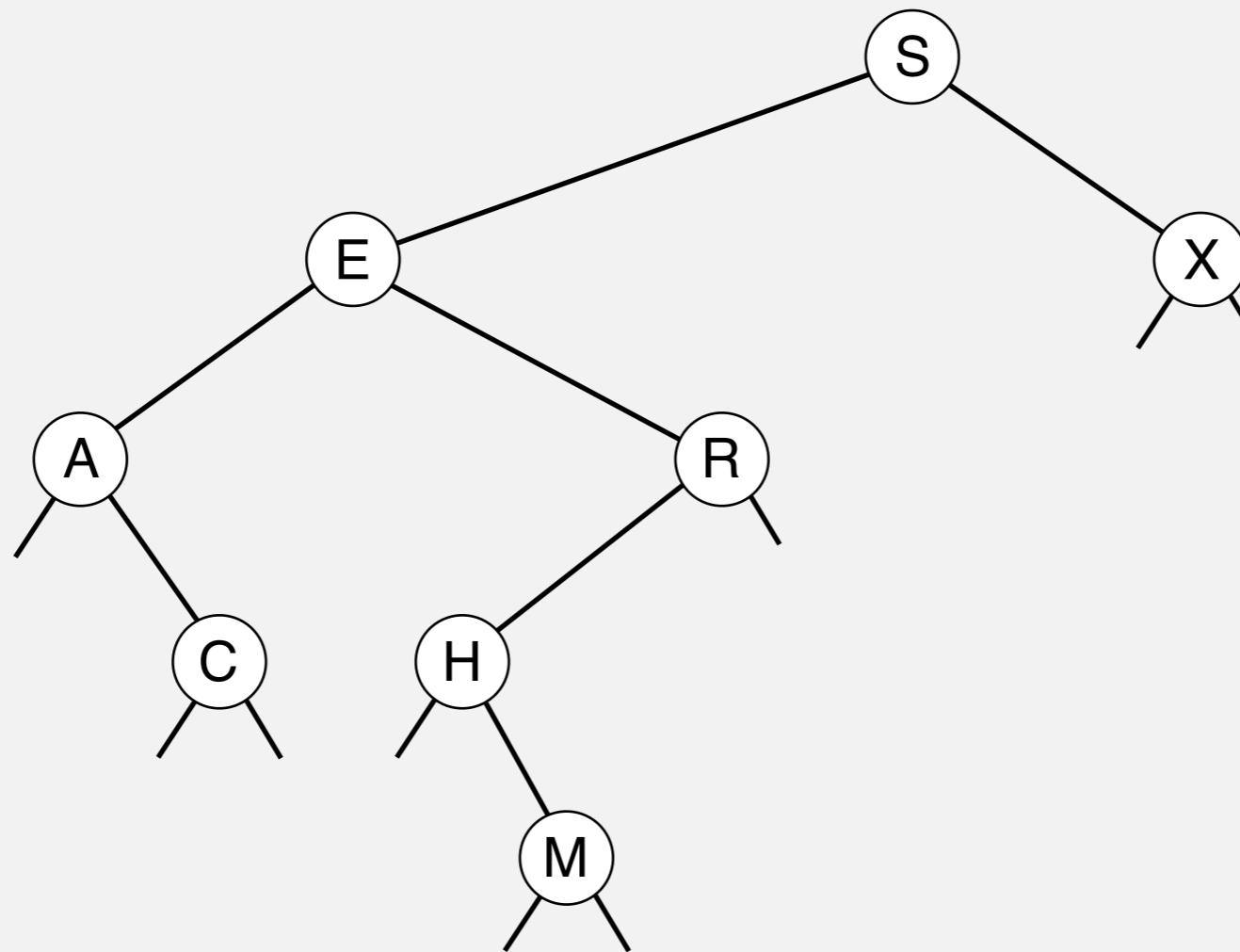
    public Iterable<Key> iterator()
    { /* see next slides */ }
}
```

← root of BST

Binary search tree operations

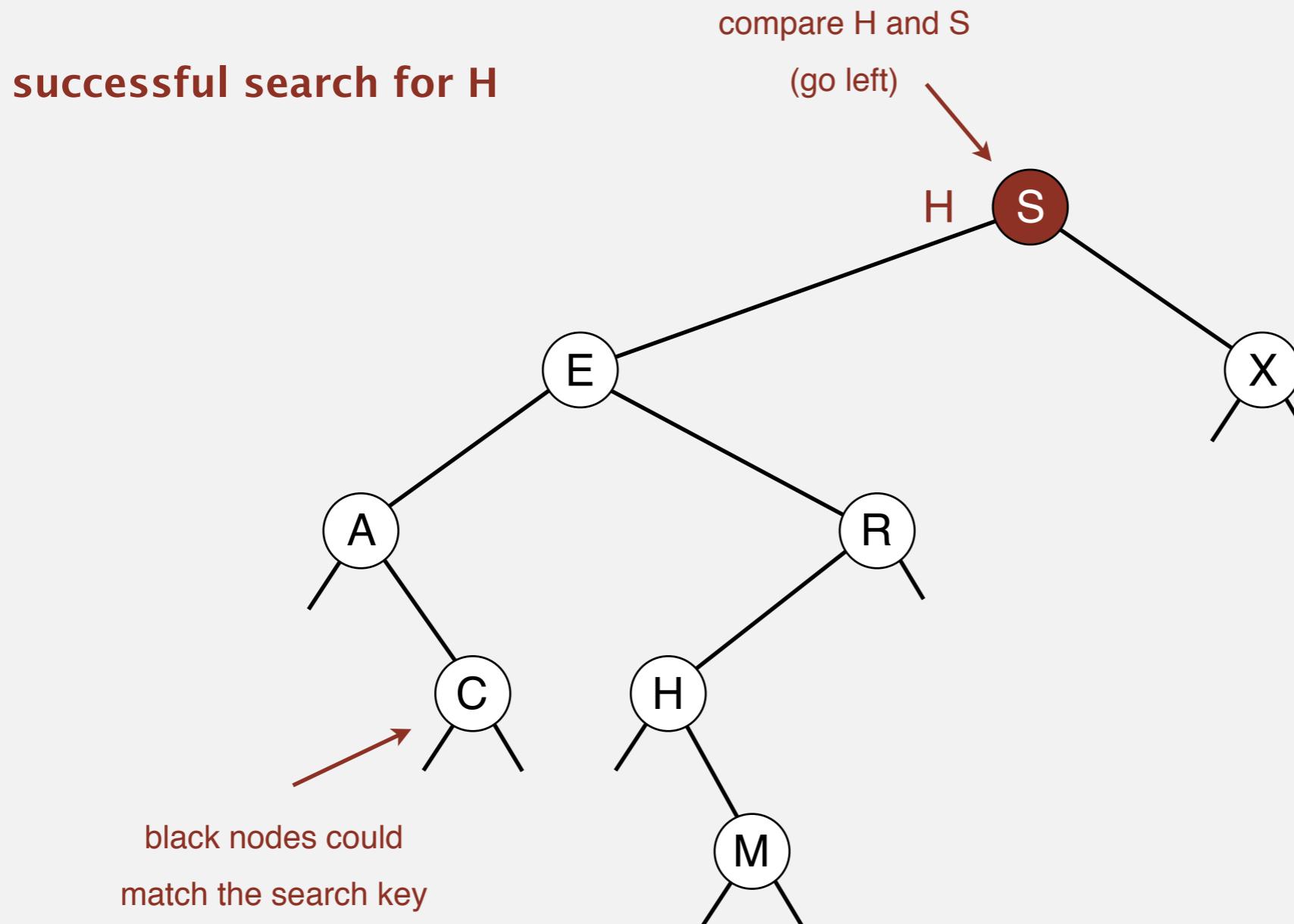
Search. If less, go left; if greater, go right; if equal, search hit.

successful search for H



Binary search tree operations

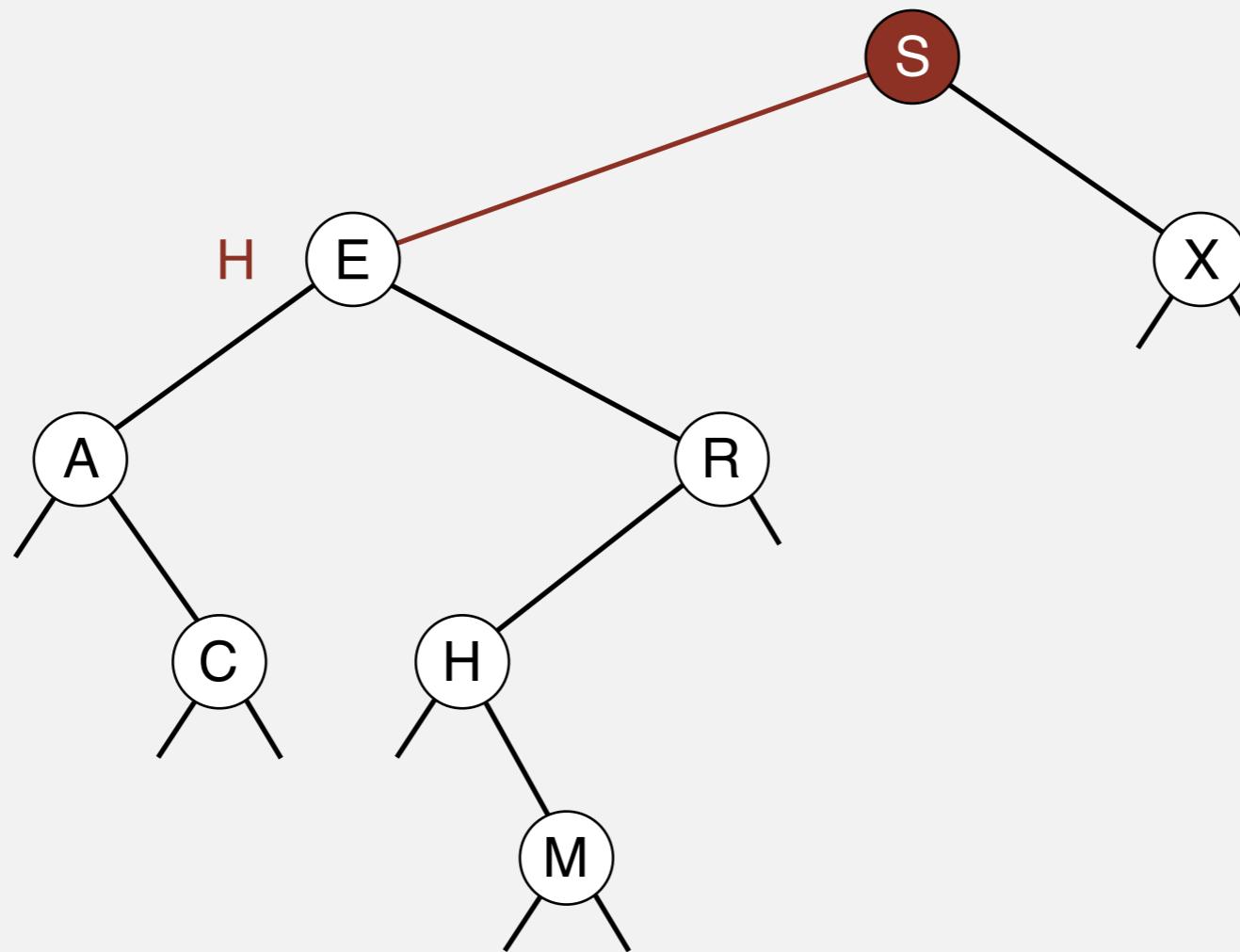
Search. If less, go left; if greater, go right; if equal, search hit.



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

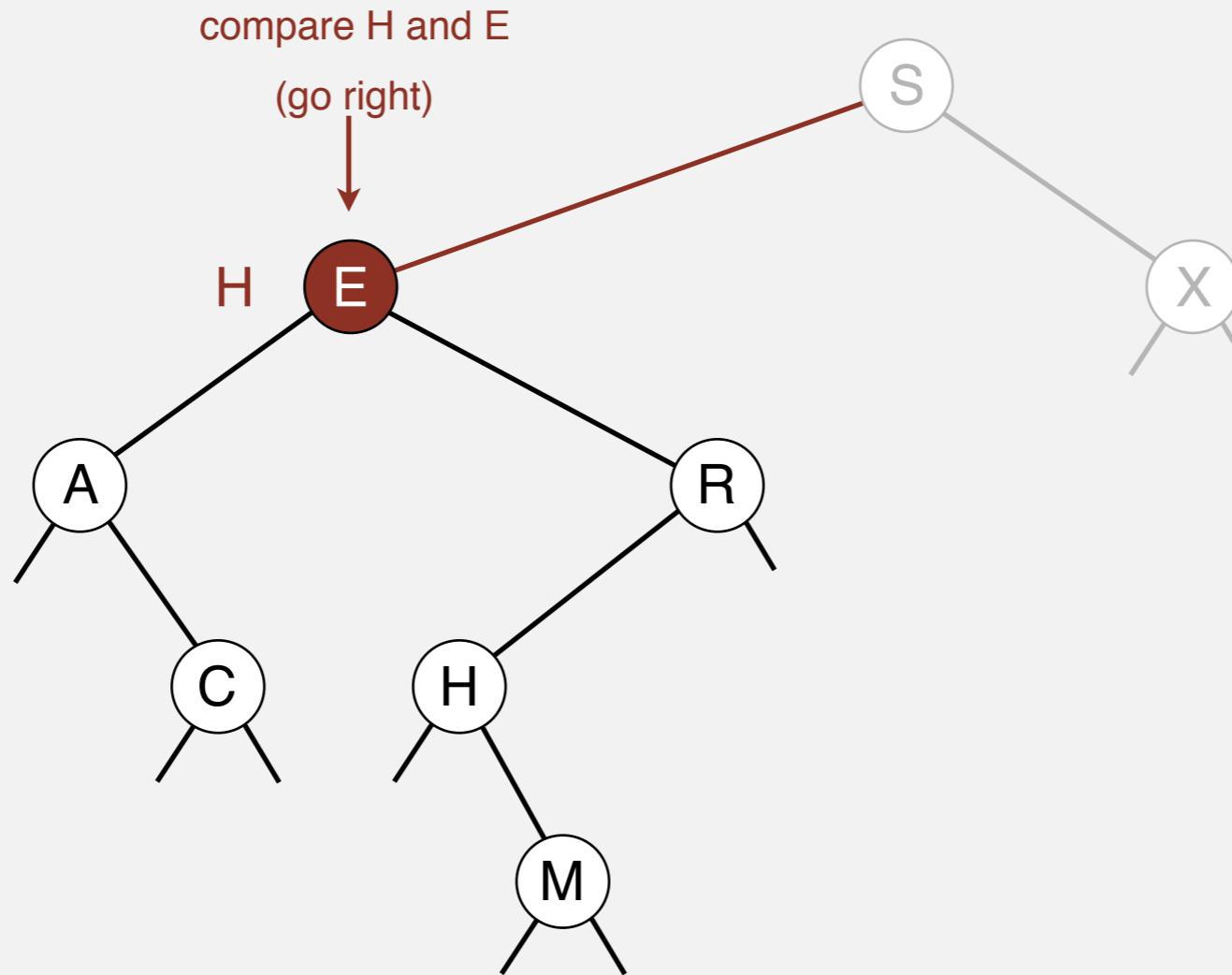
successful search for H



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

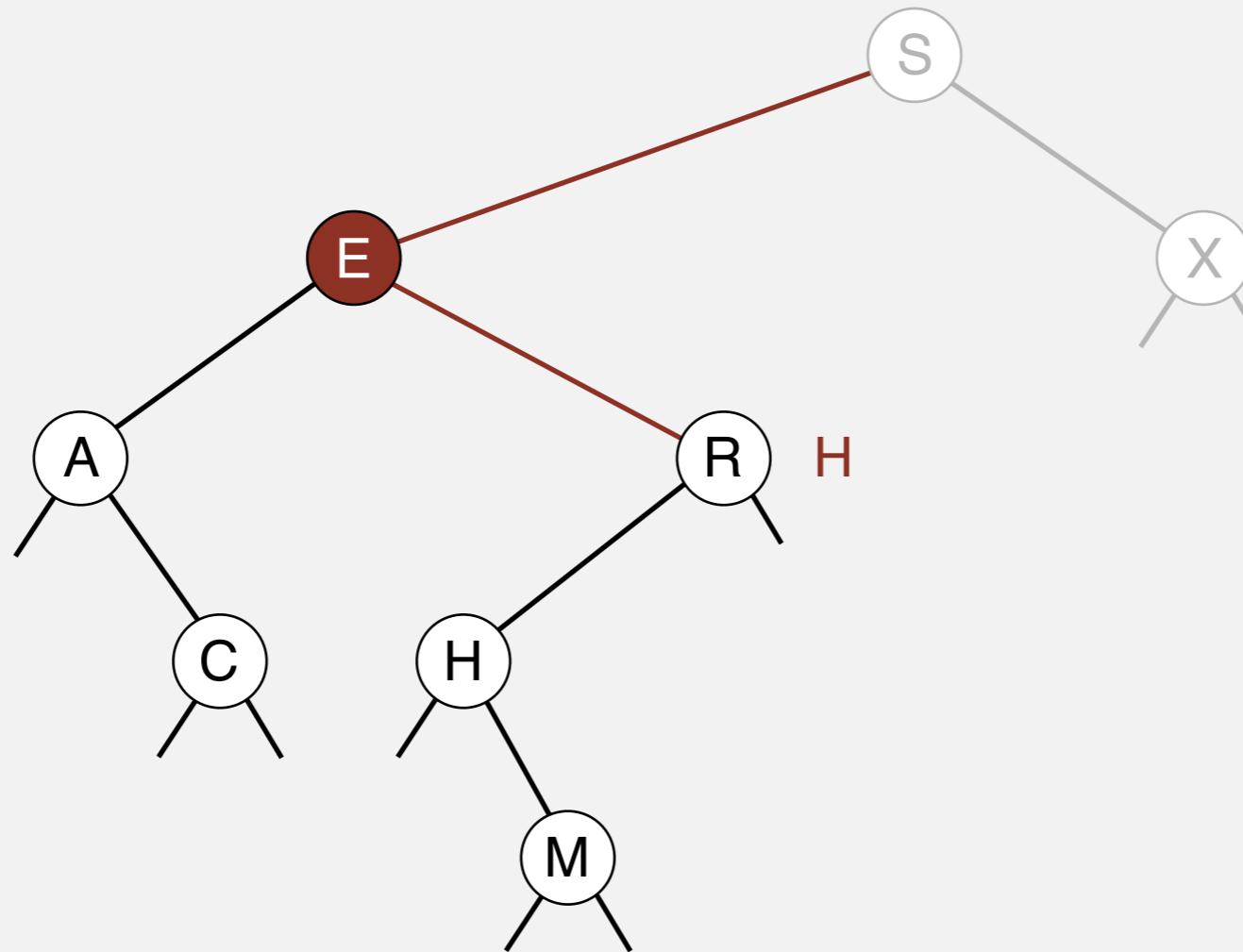
successful search for H



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

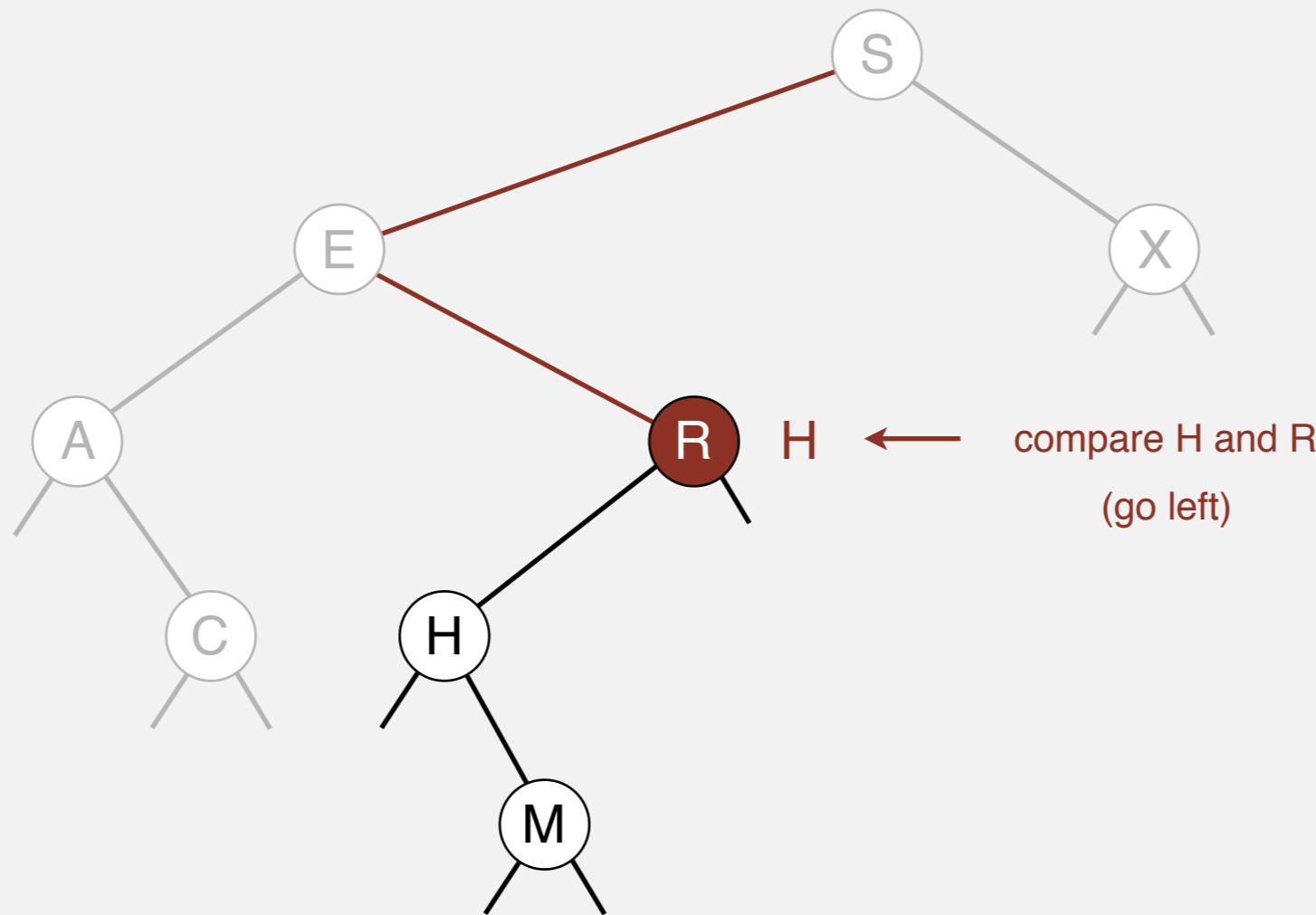
successful search for H



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

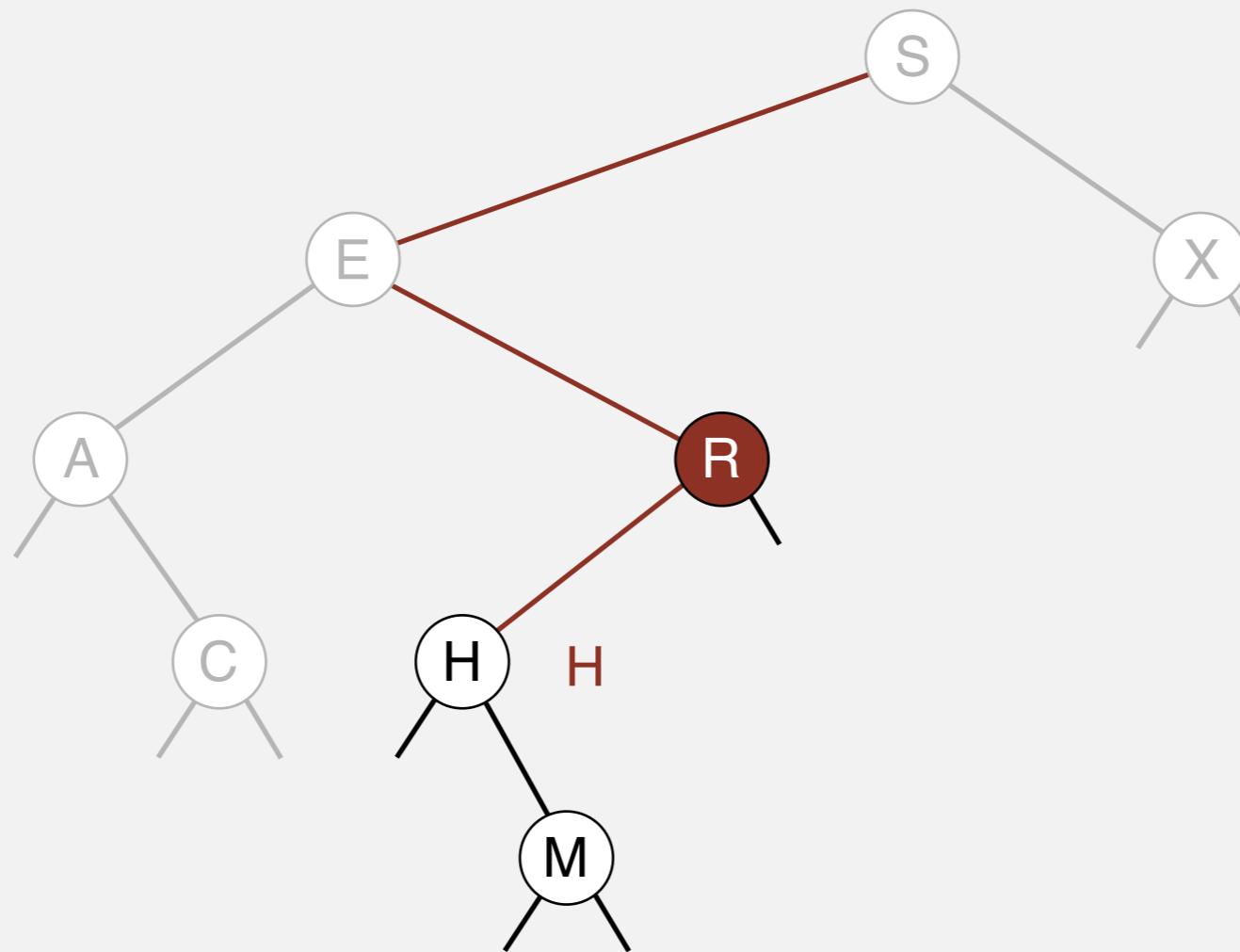
successful search for H



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

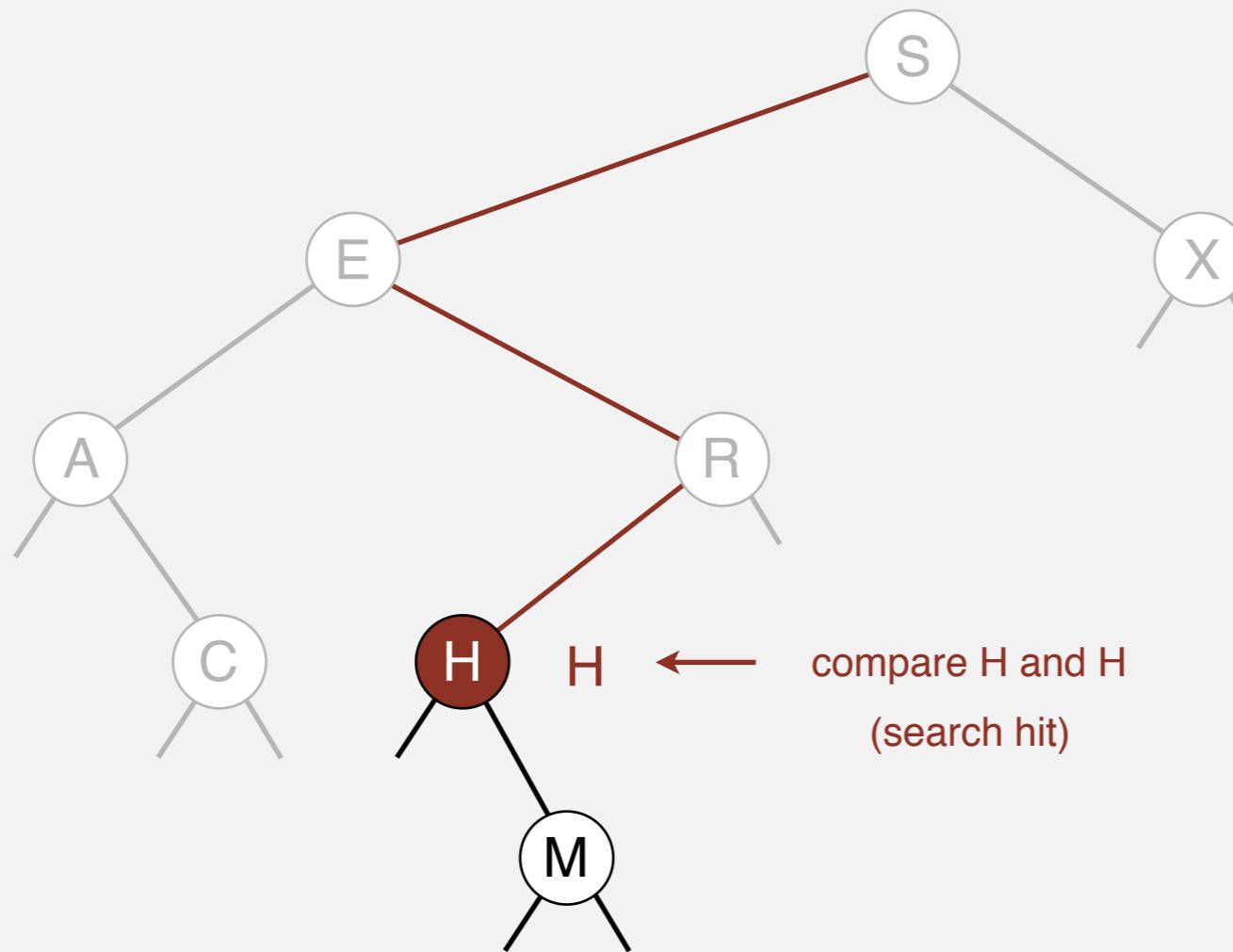
successful search for H



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

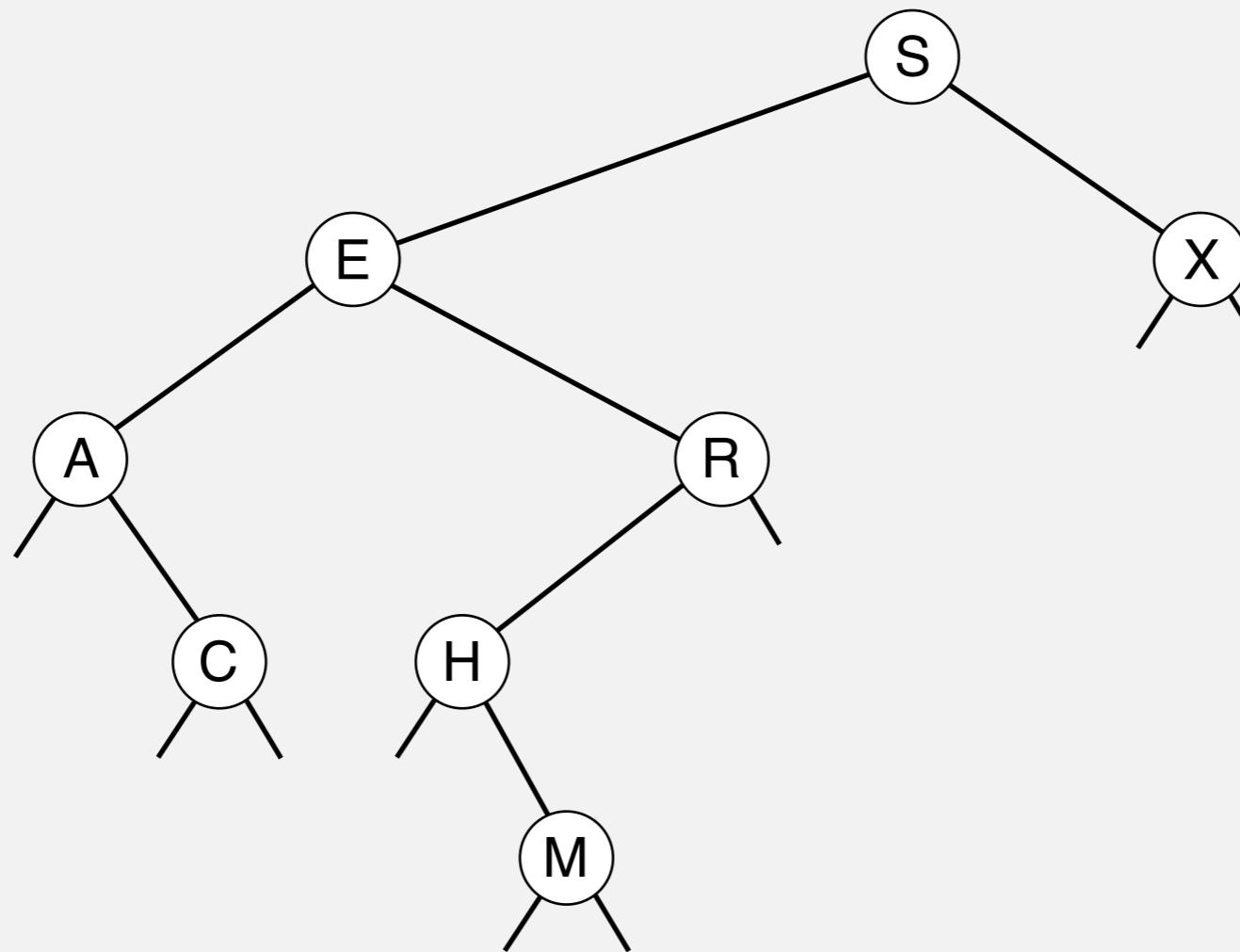
successful search for H



Binary search tree operations

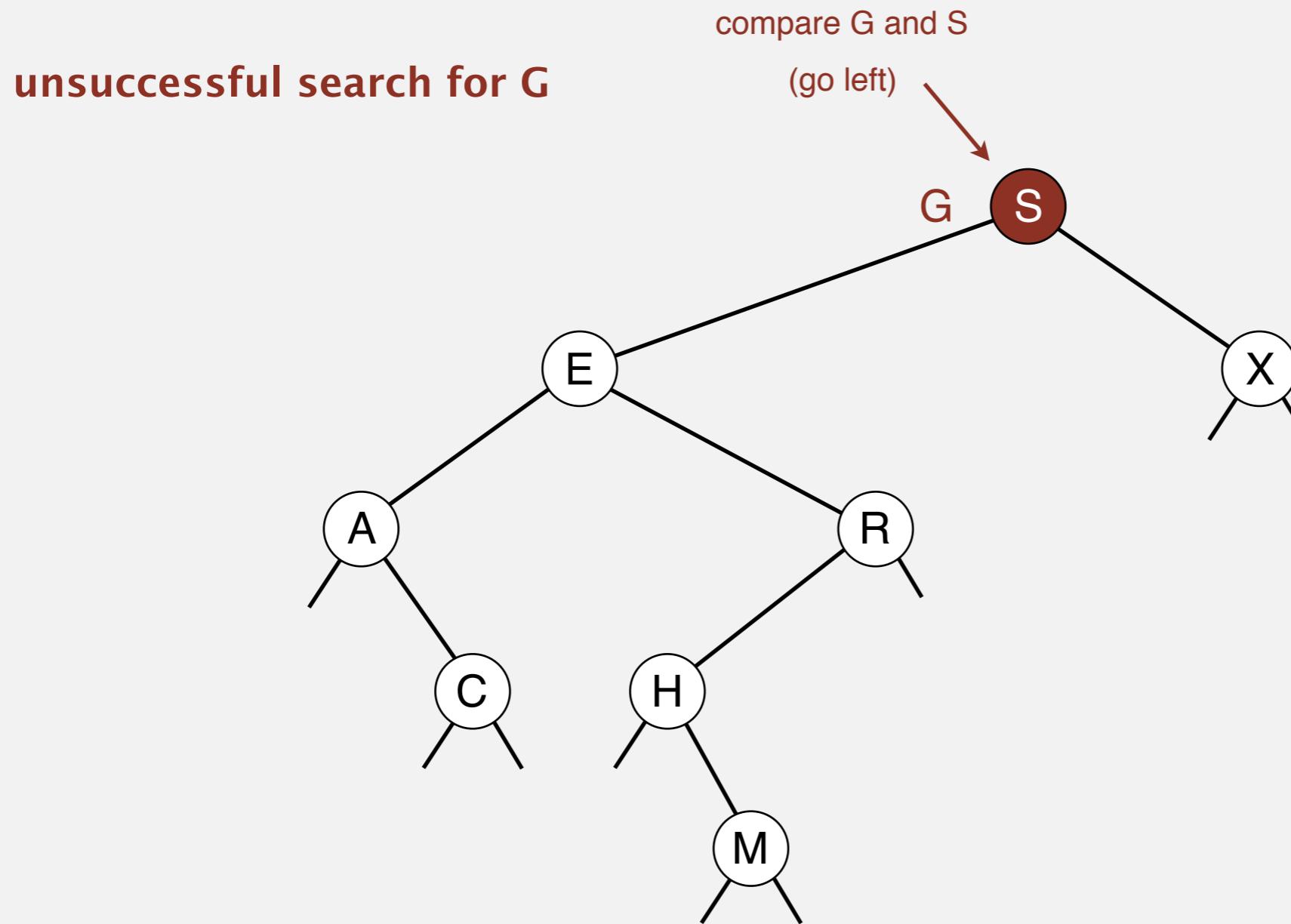
Search. If less, go left; if greater, go right; if equal, search hit.

unsuccessful search for G



Binary search tree operations

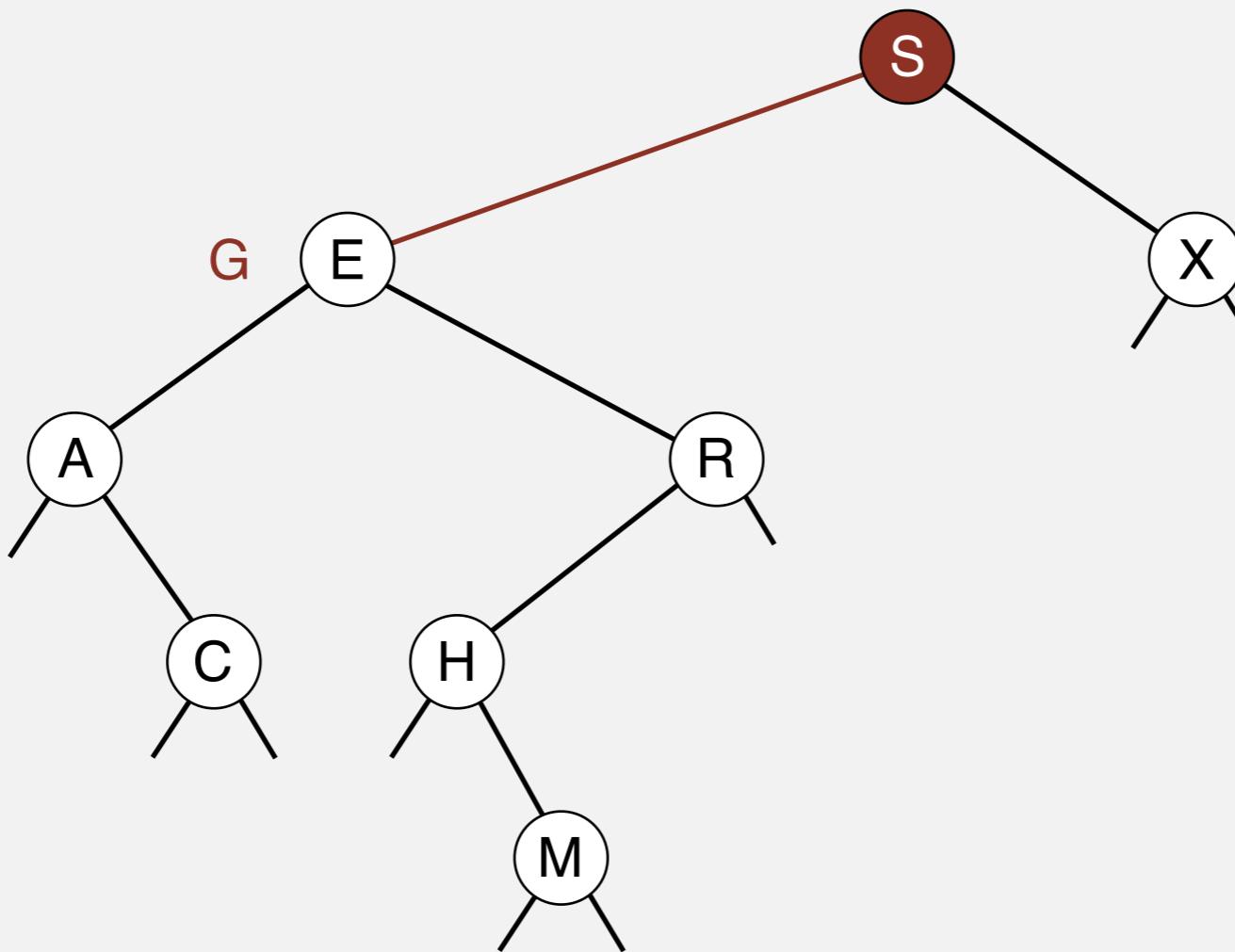
Search. If less, go left; if greater, go right; if equal, search hit.



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

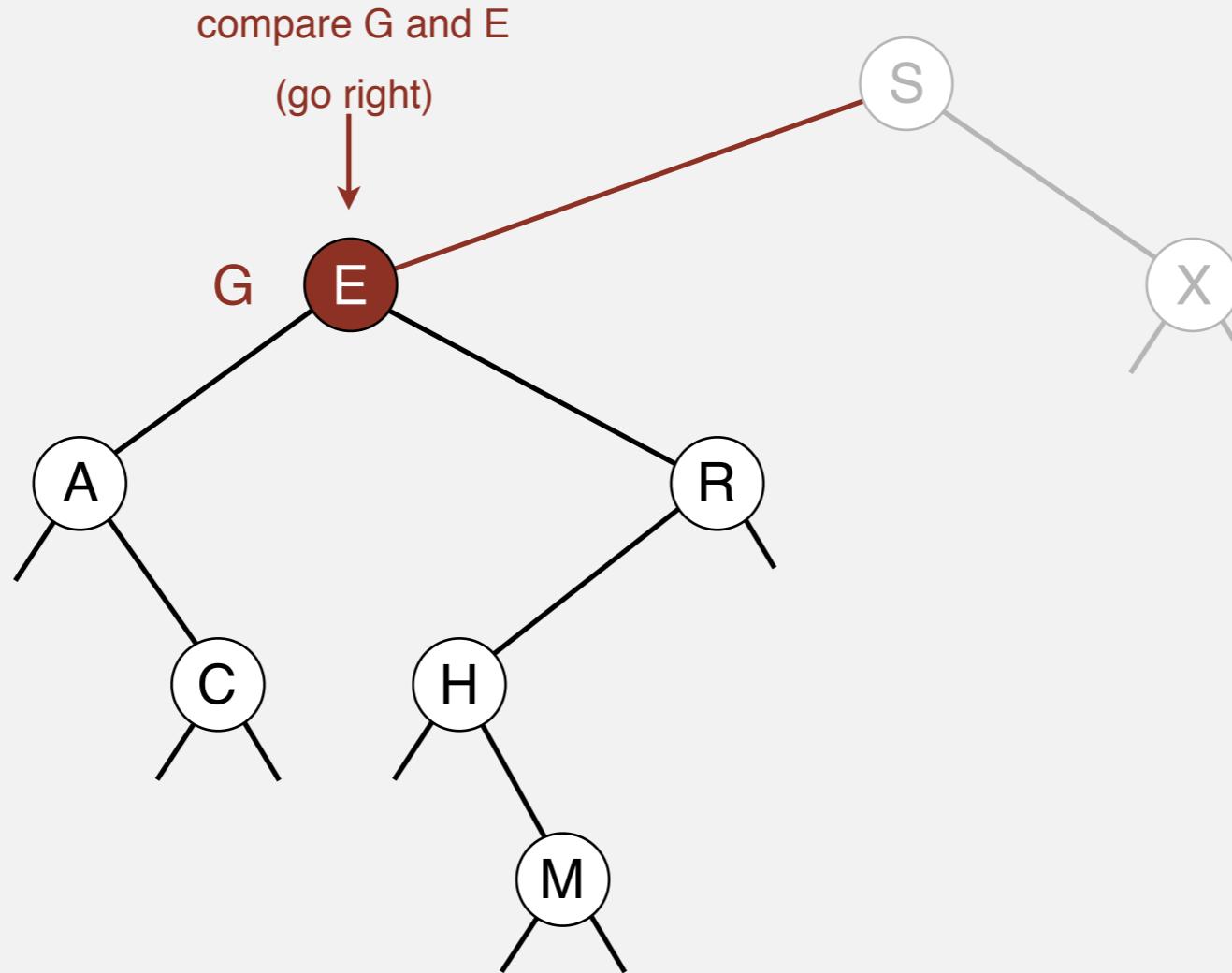
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

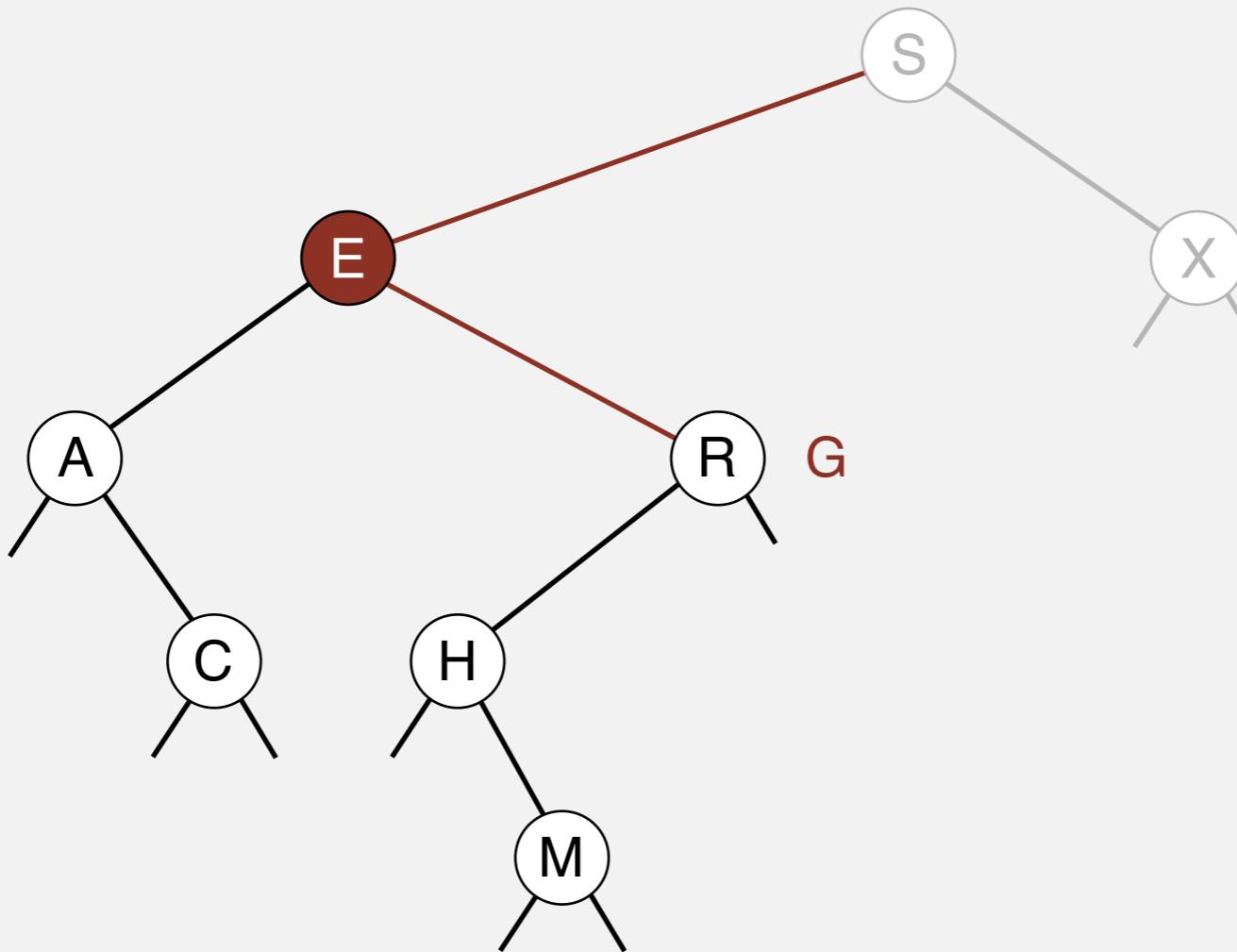
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

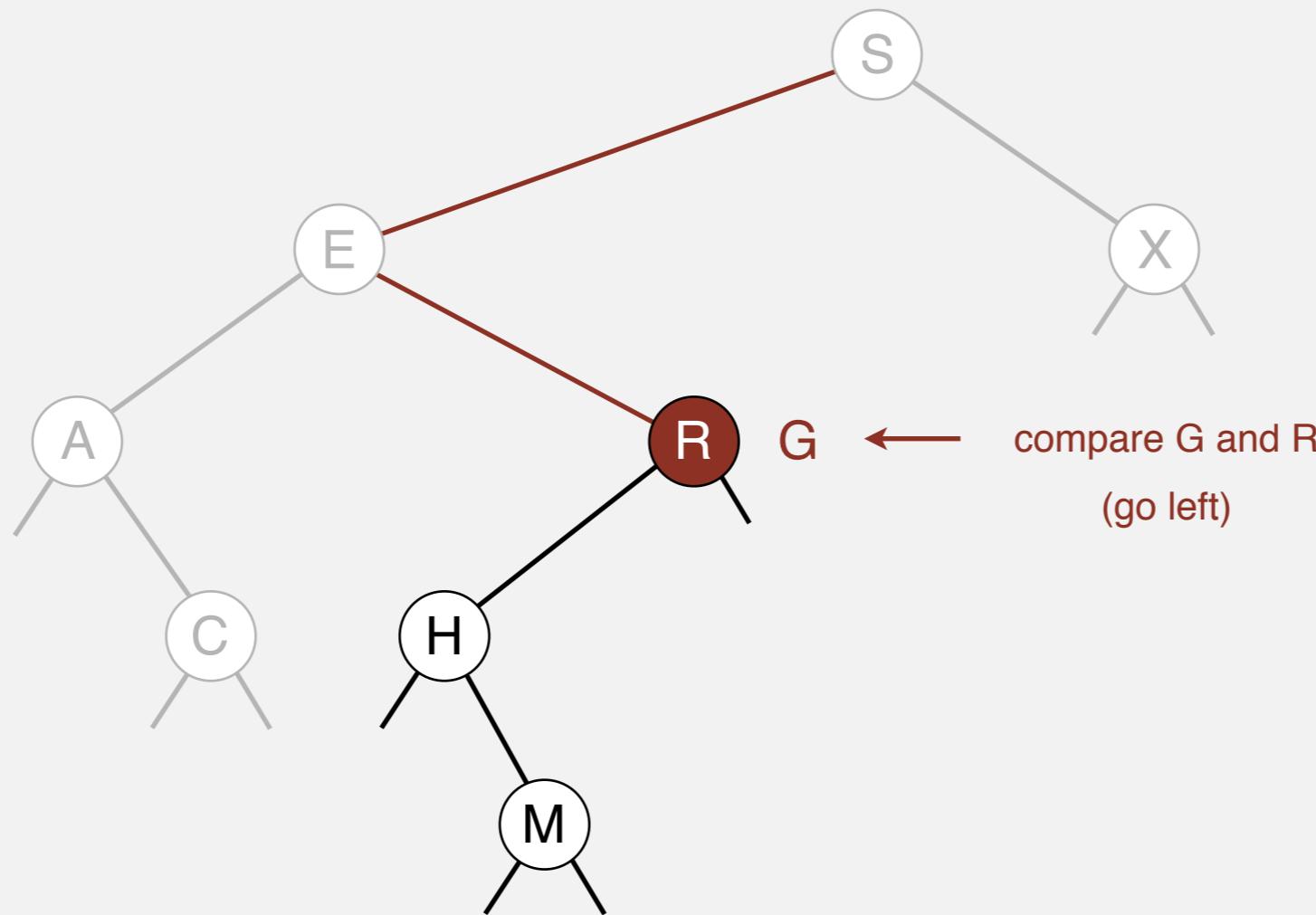
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

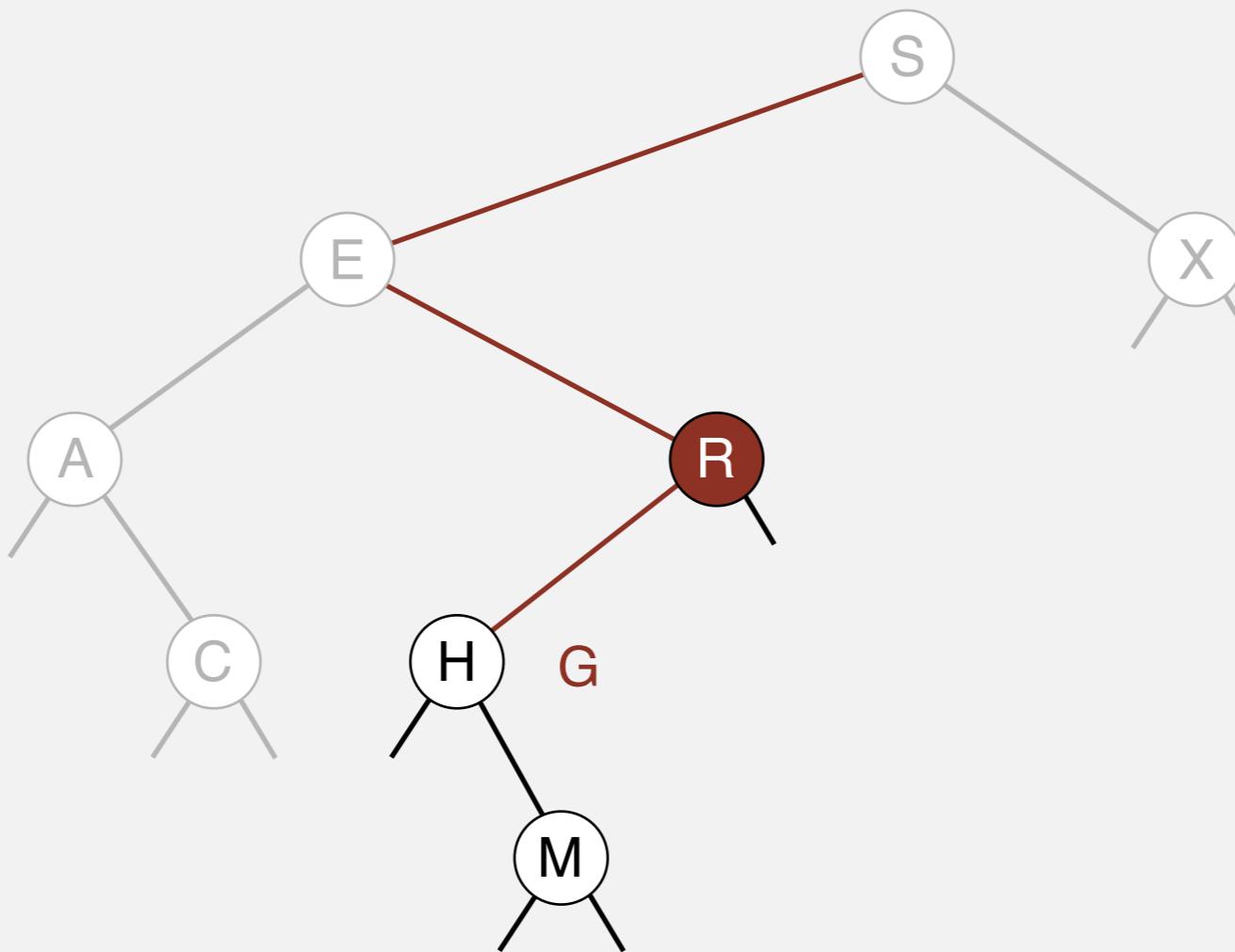
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

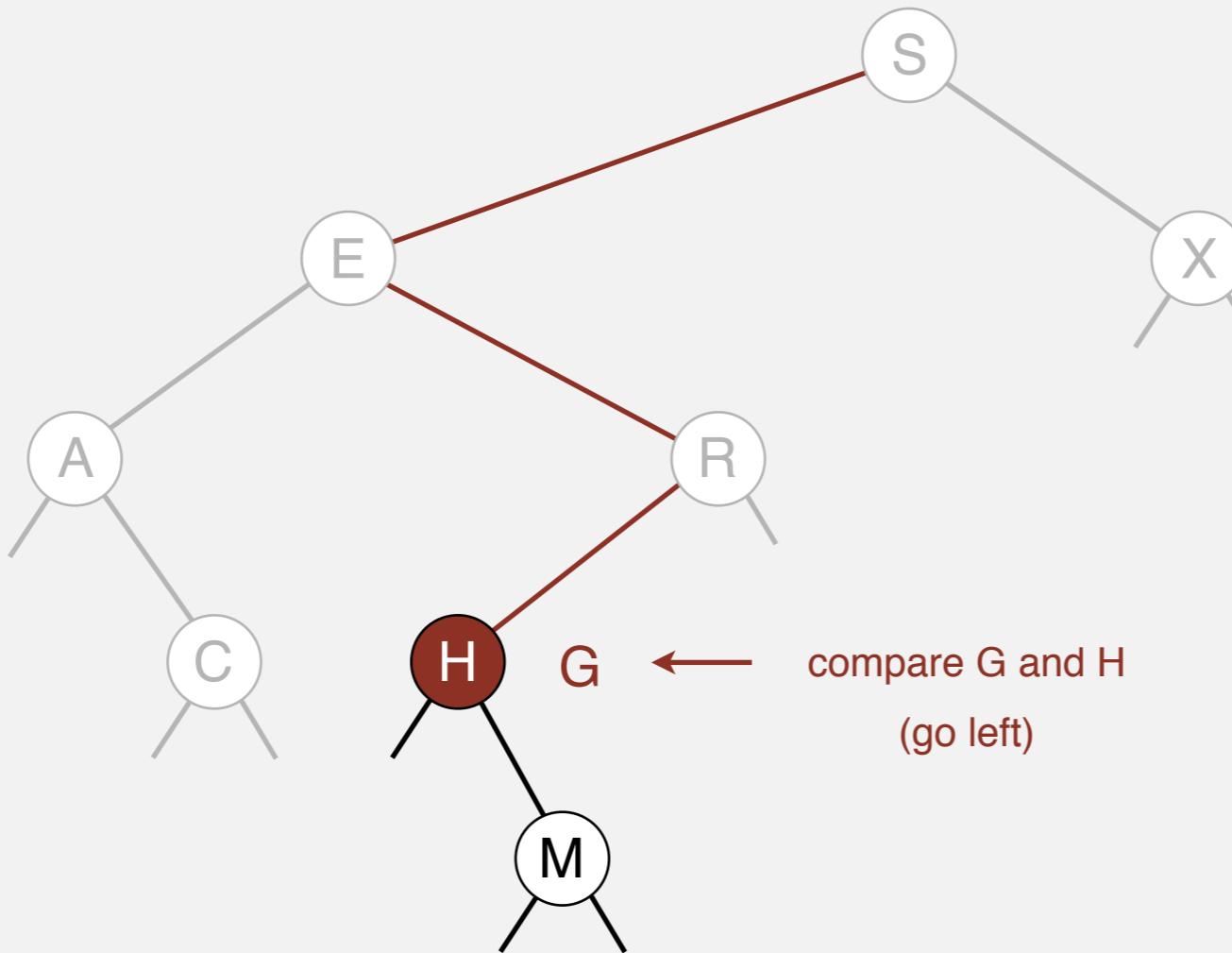
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

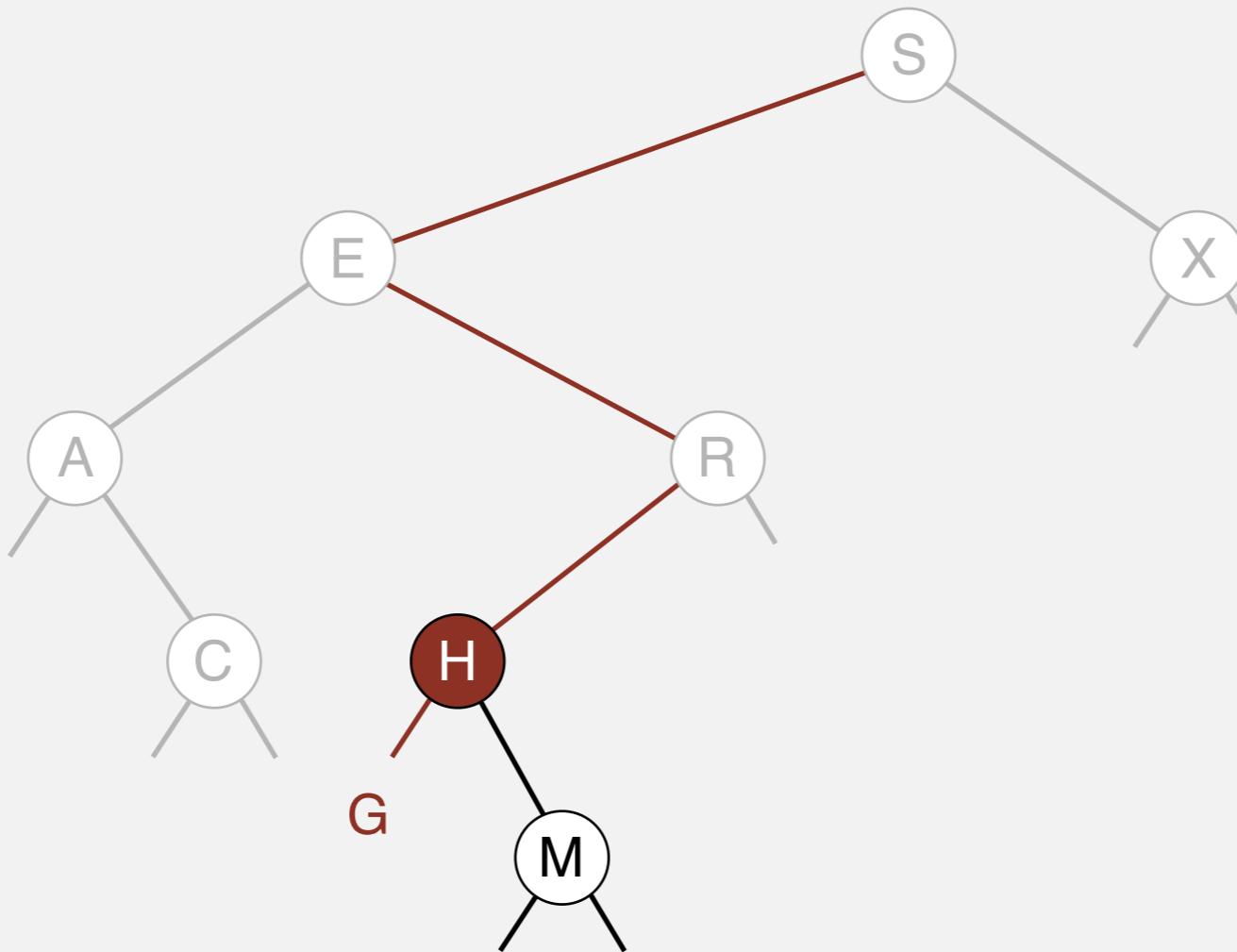
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

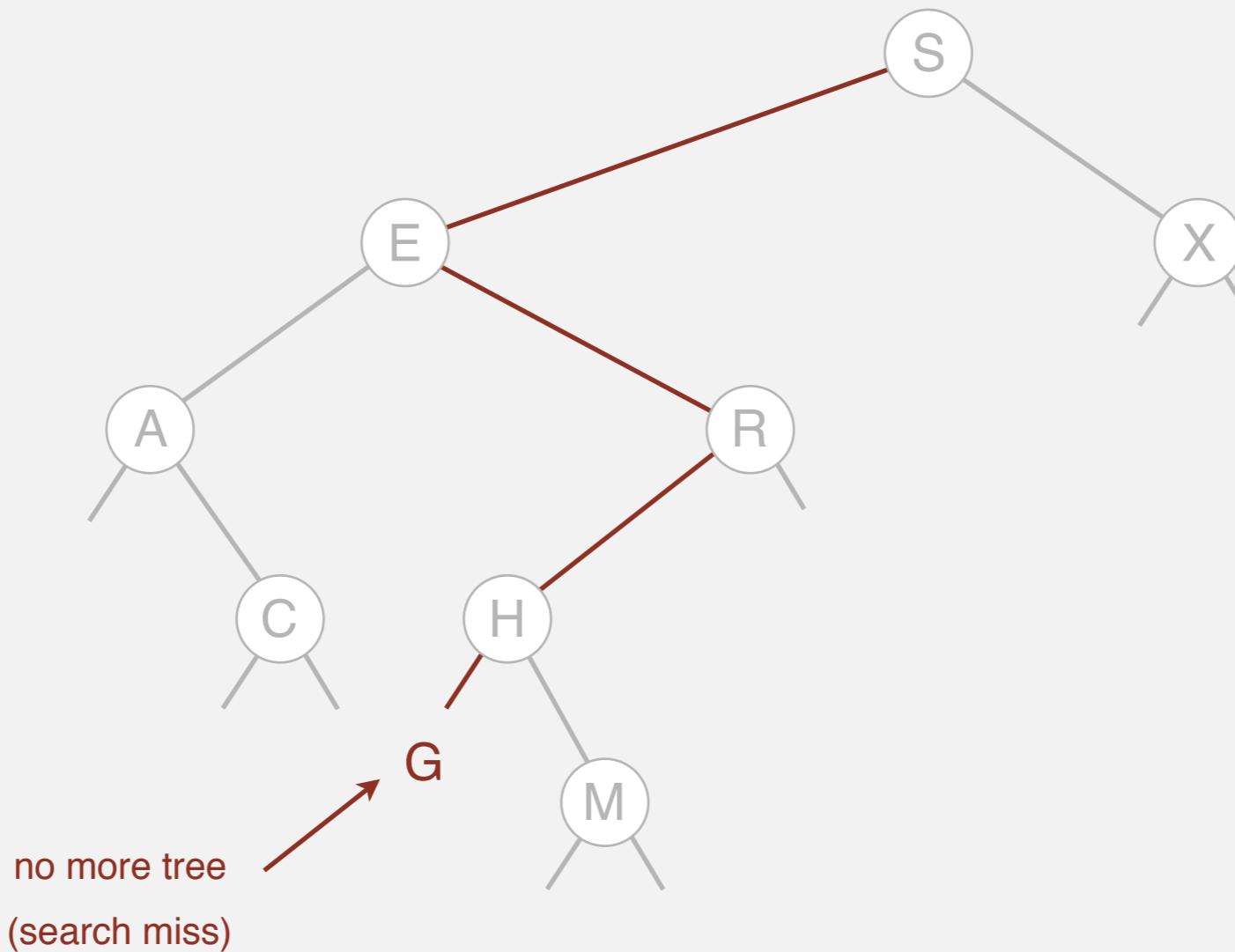
unsuccessful search for G



Binary search tree operations

Search. If less, go left; if greater, go right; if equal, search hit.

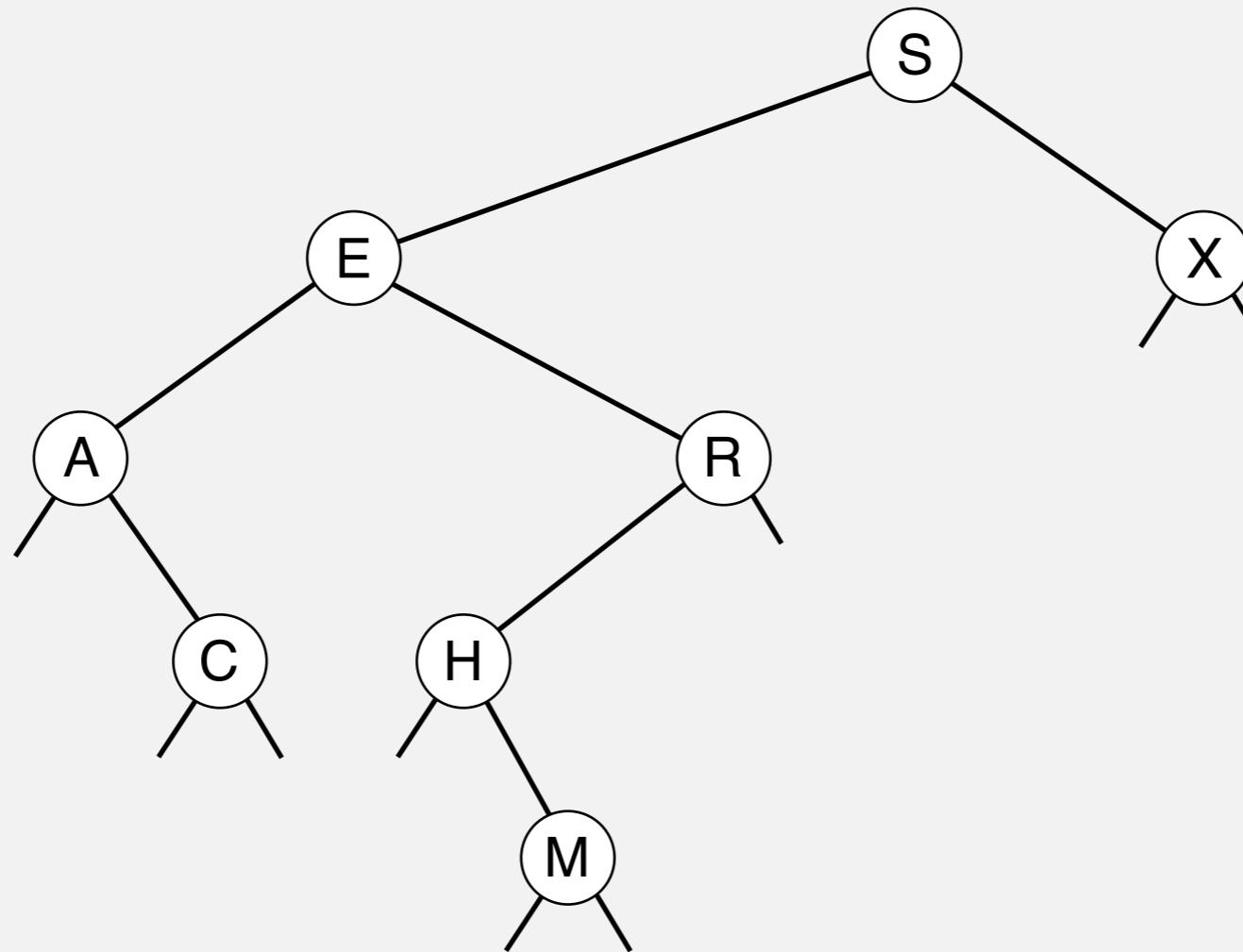
unsuccessful search for G



Binary search tree operations

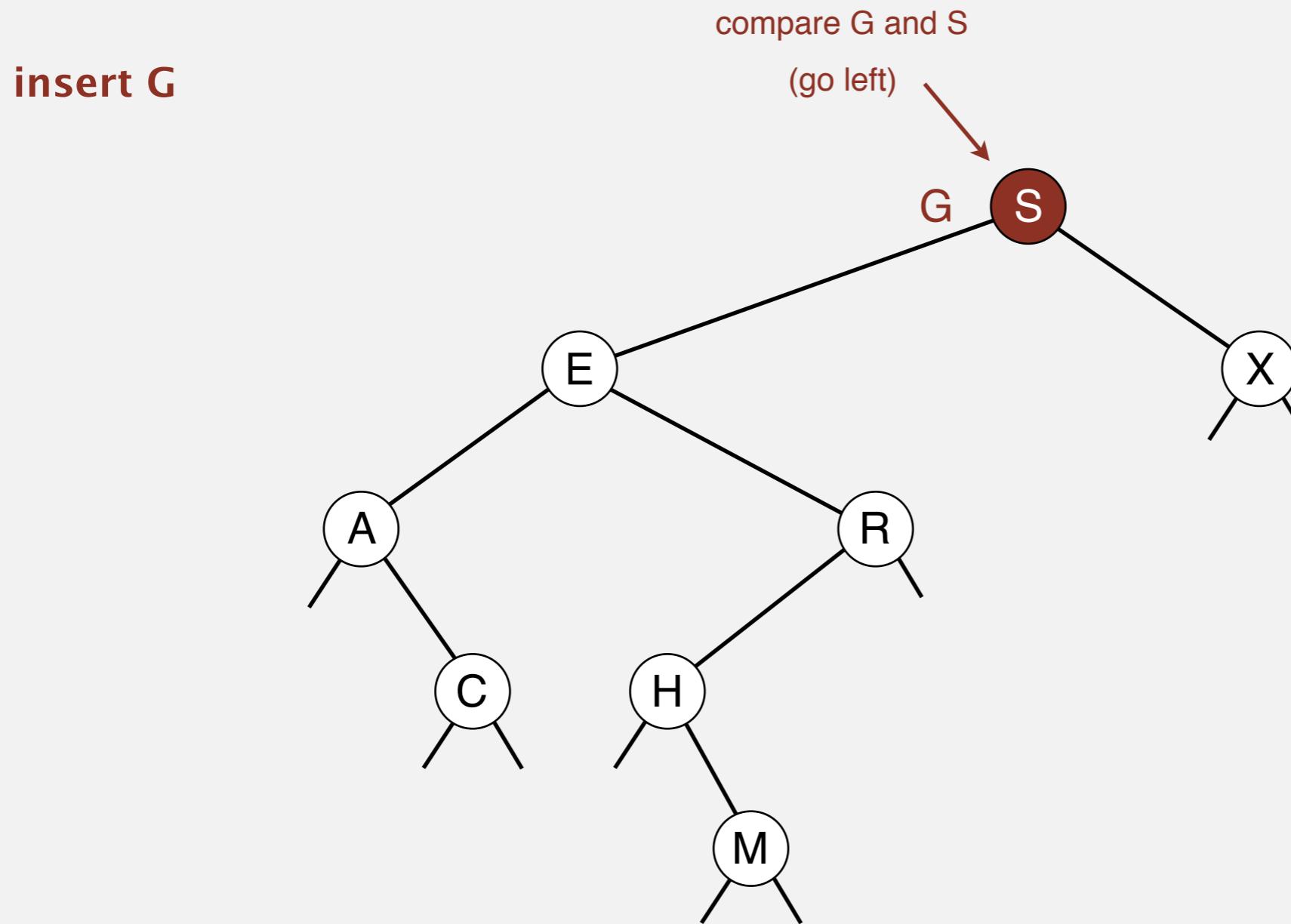
Insert. If less, go left; if greater, go right; if null, insert.

insert G



Binary search tree operations

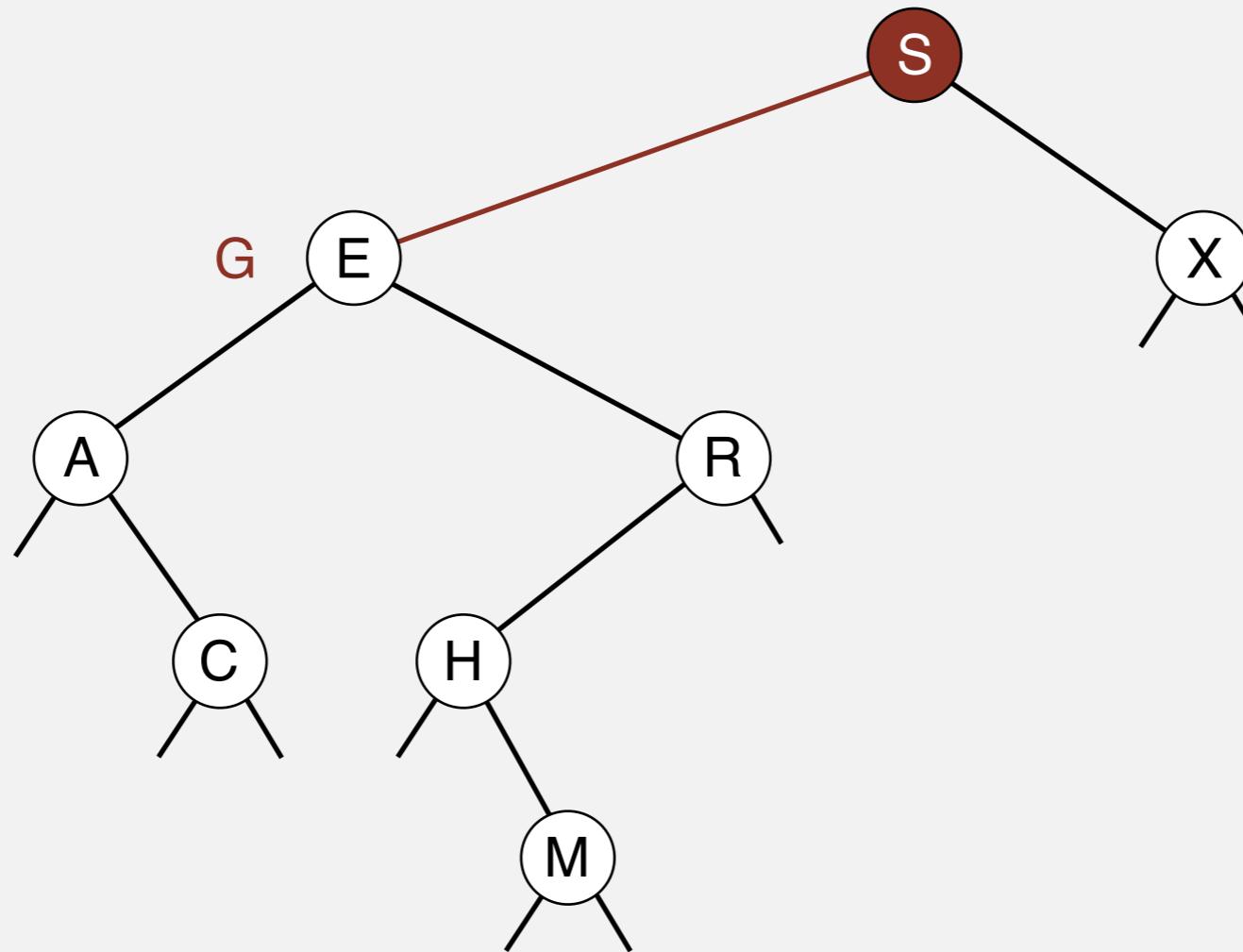
Insert. If less, go left; if greater, go right; if null, insert.



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

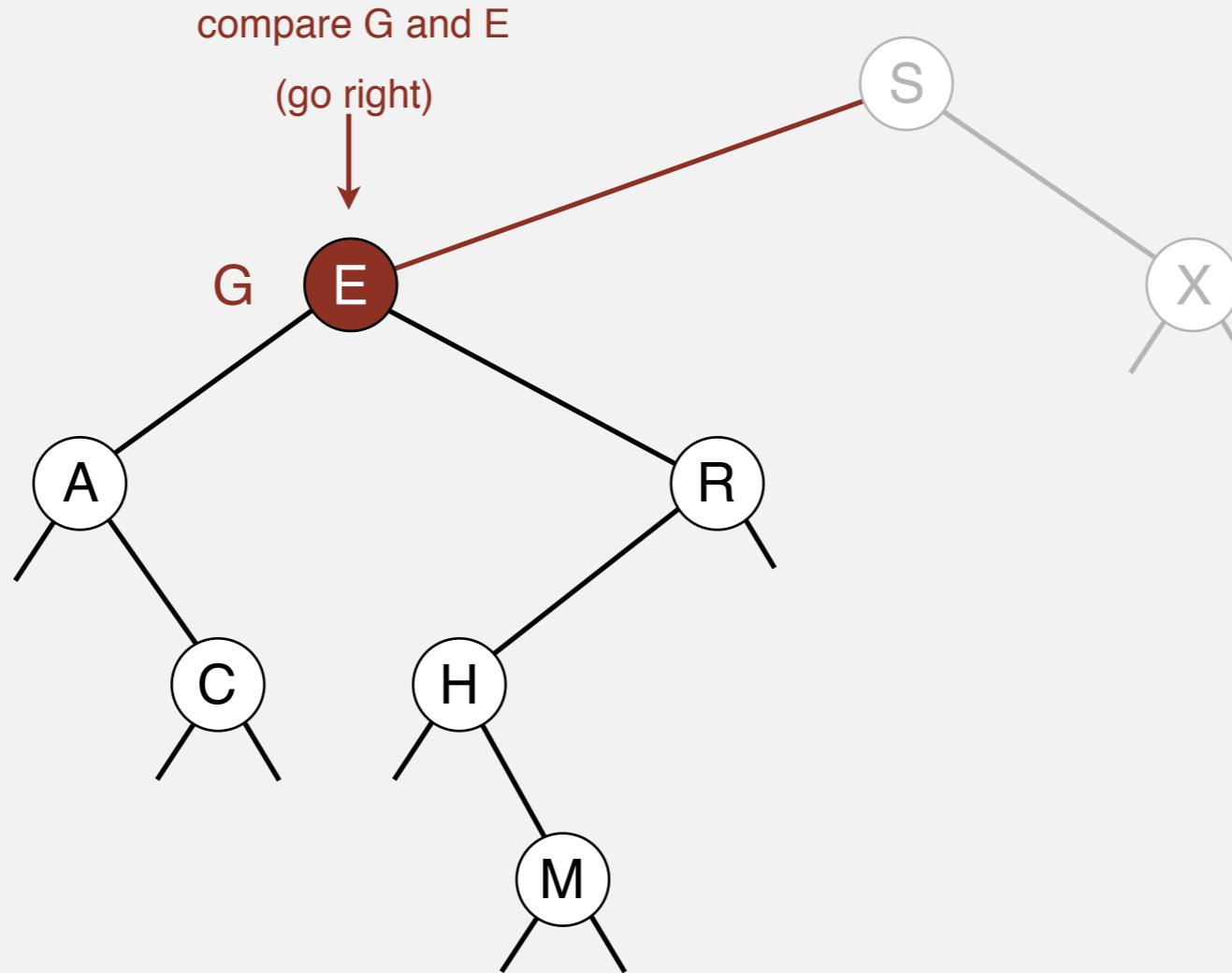
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

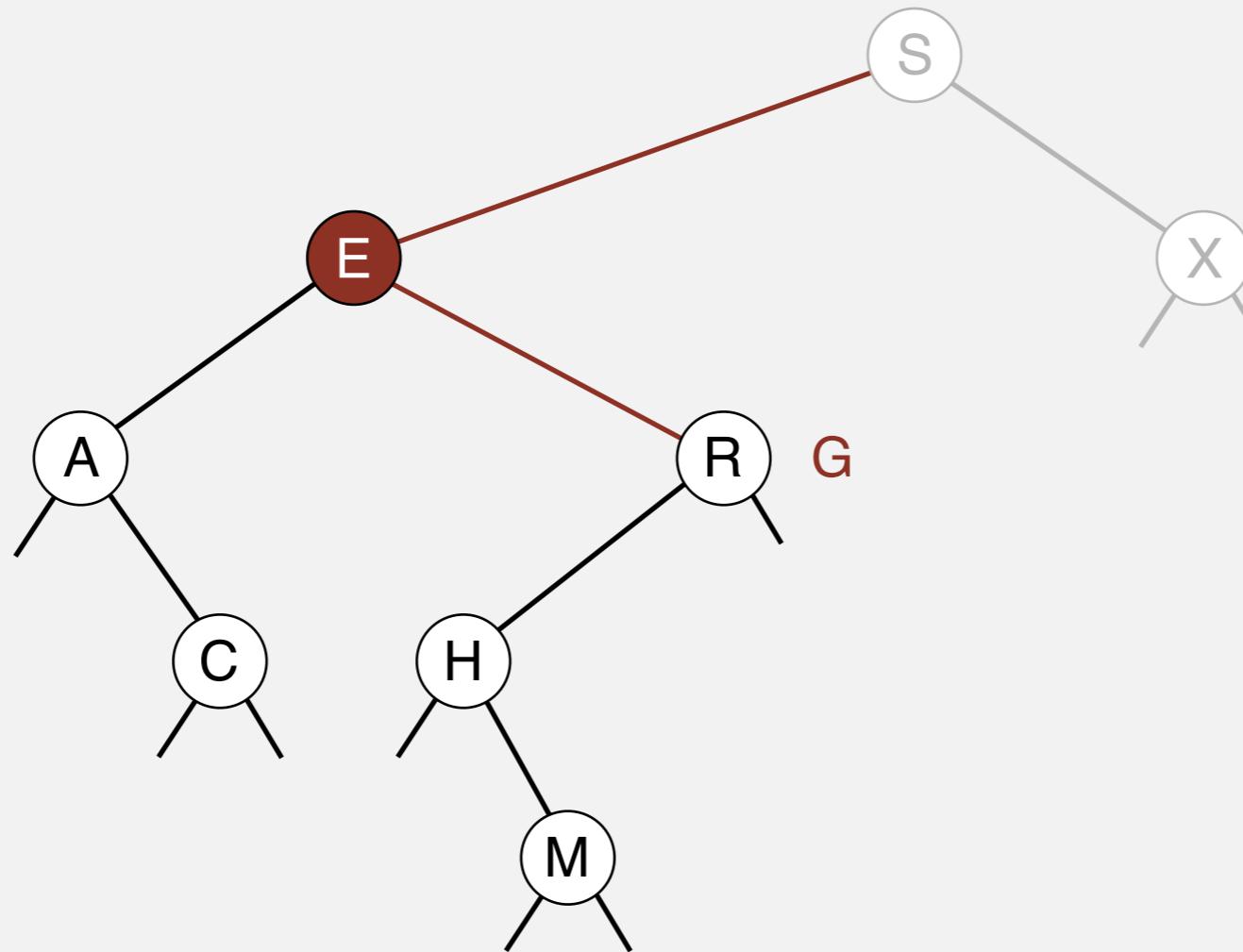
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

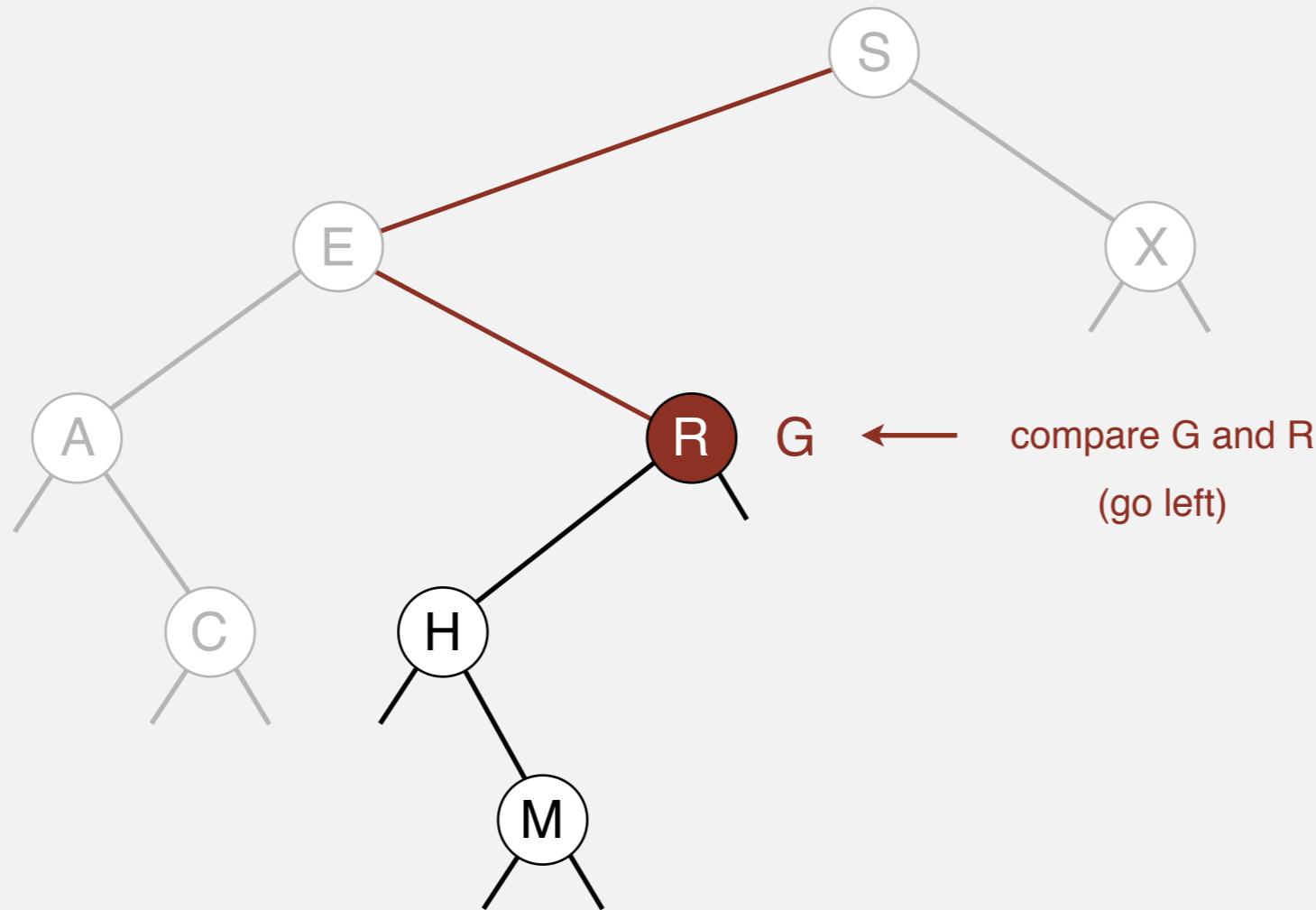
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

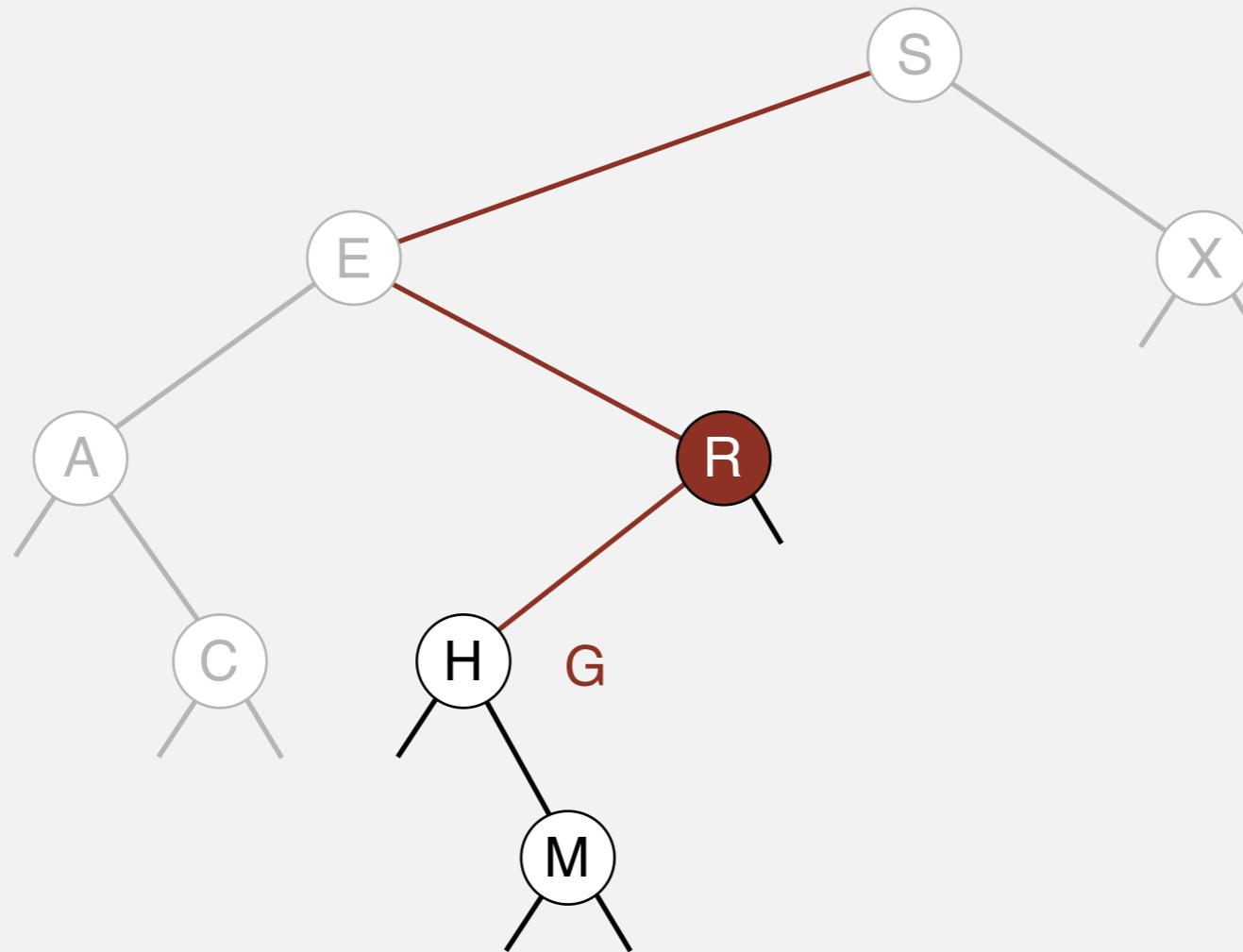
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

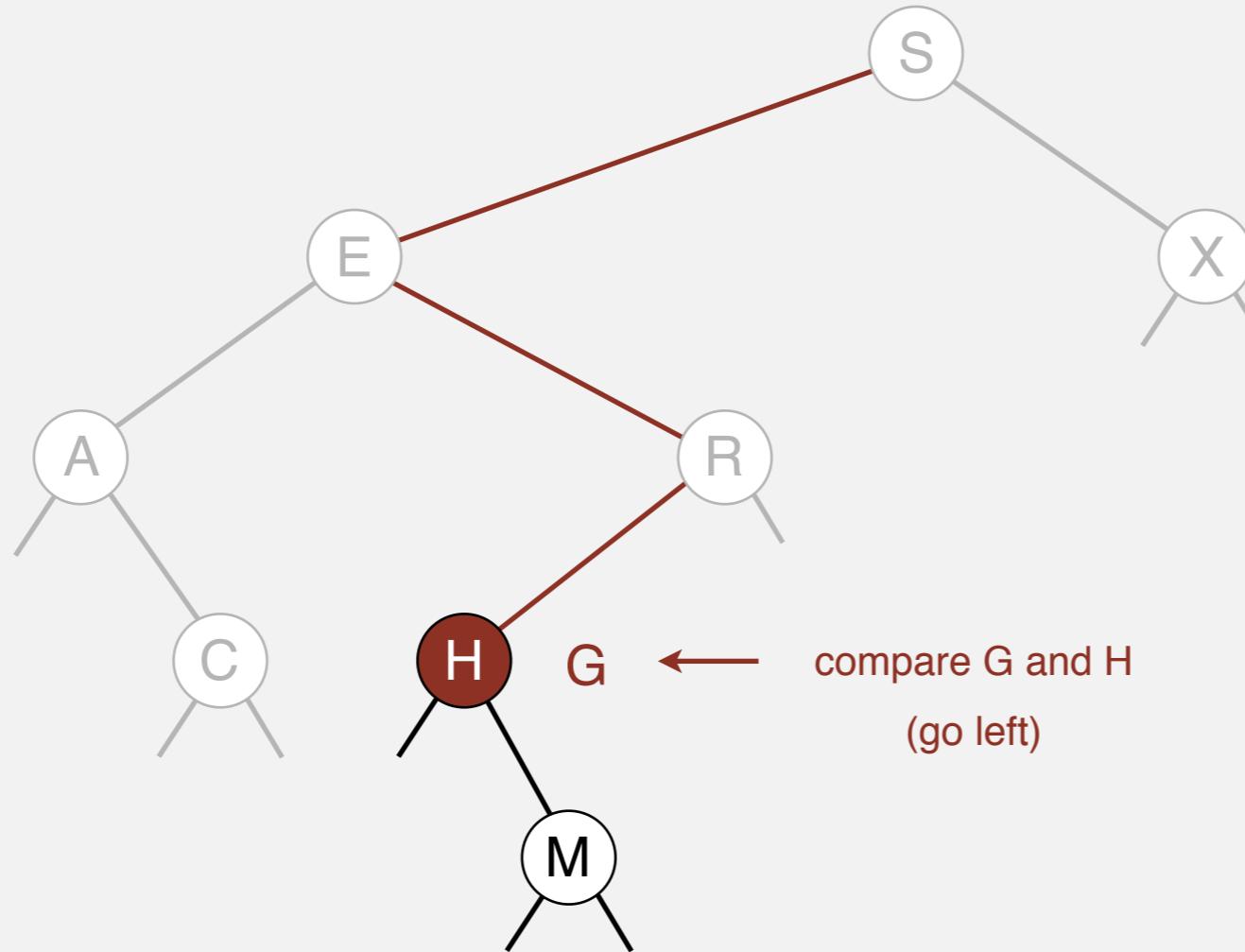
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

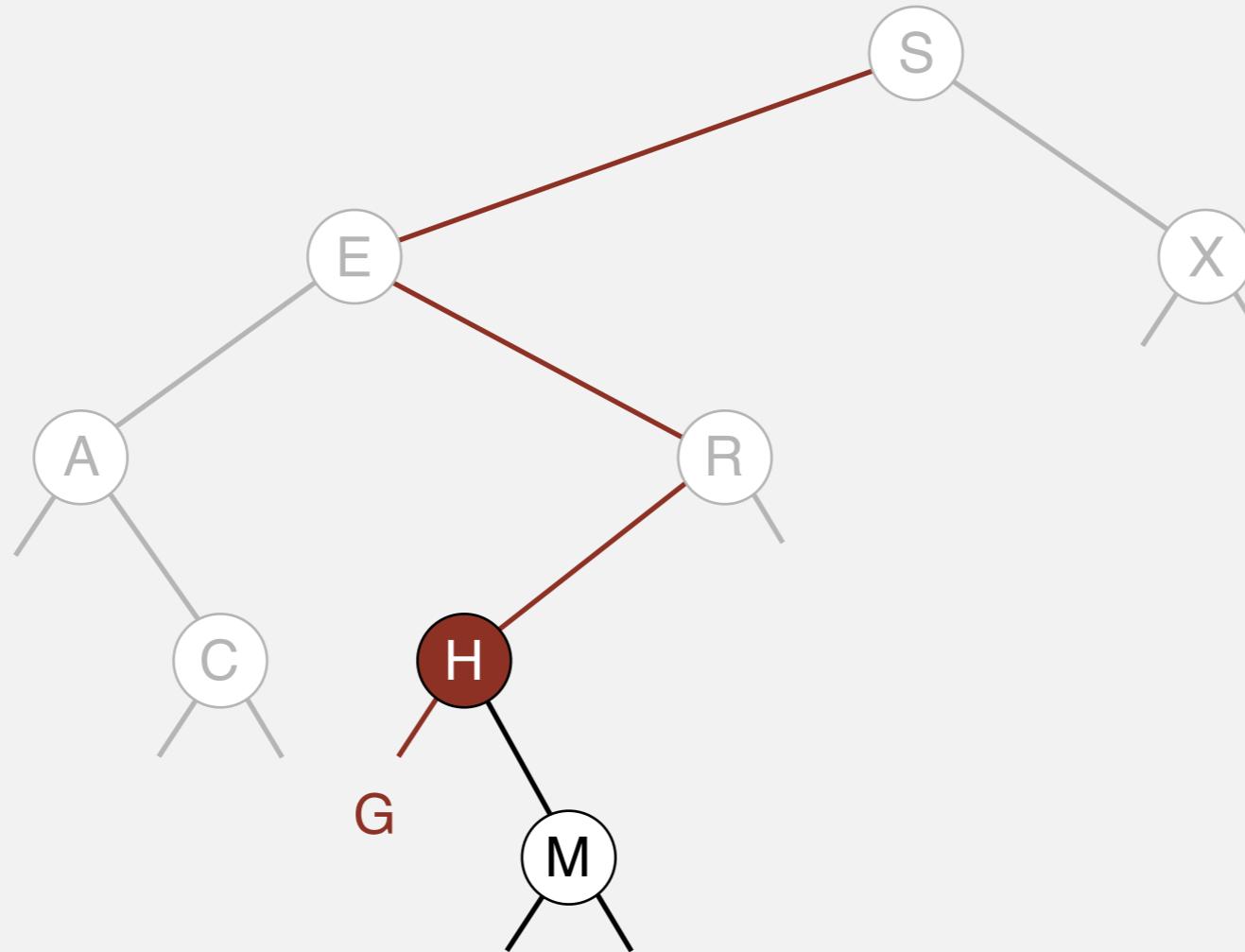
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

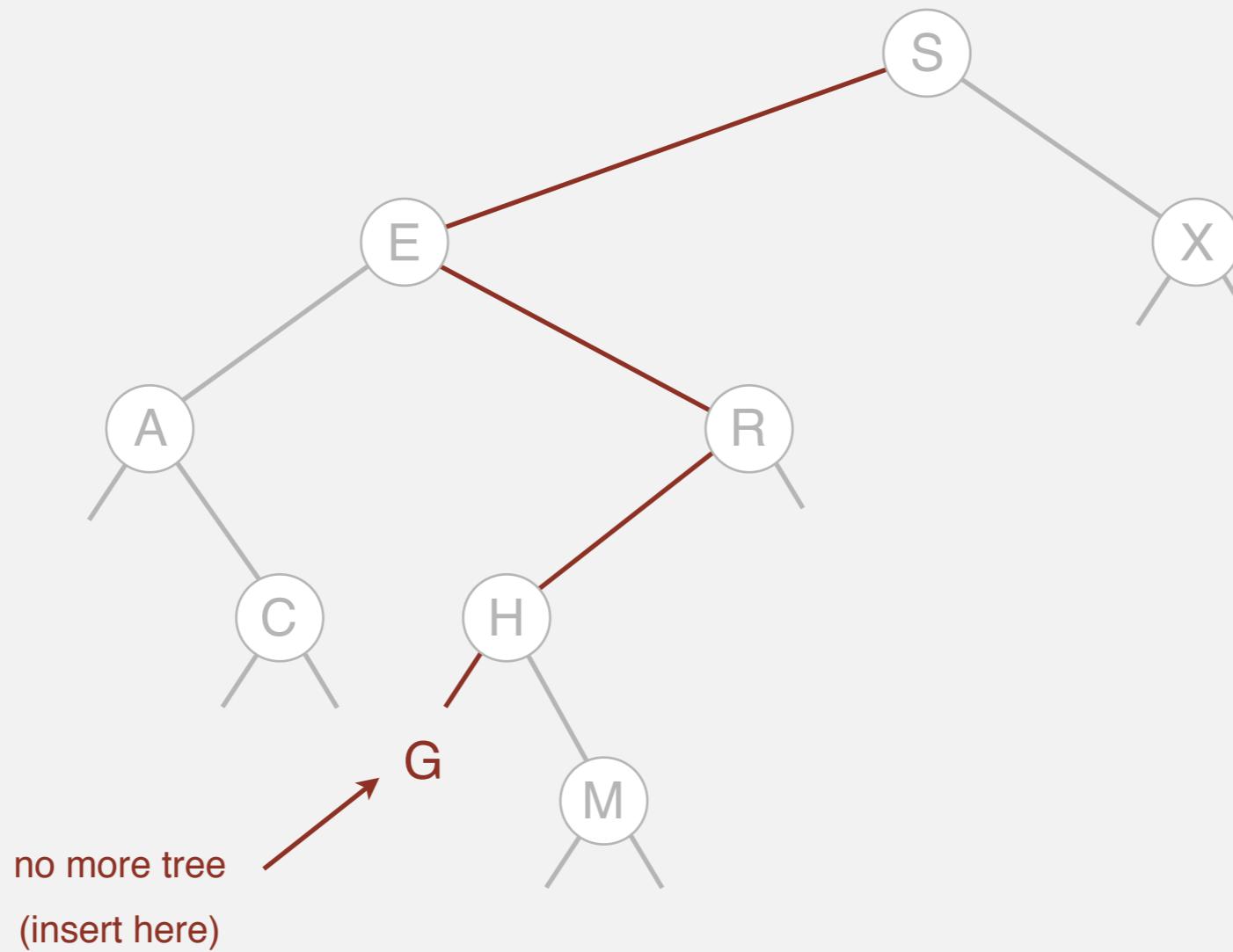
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

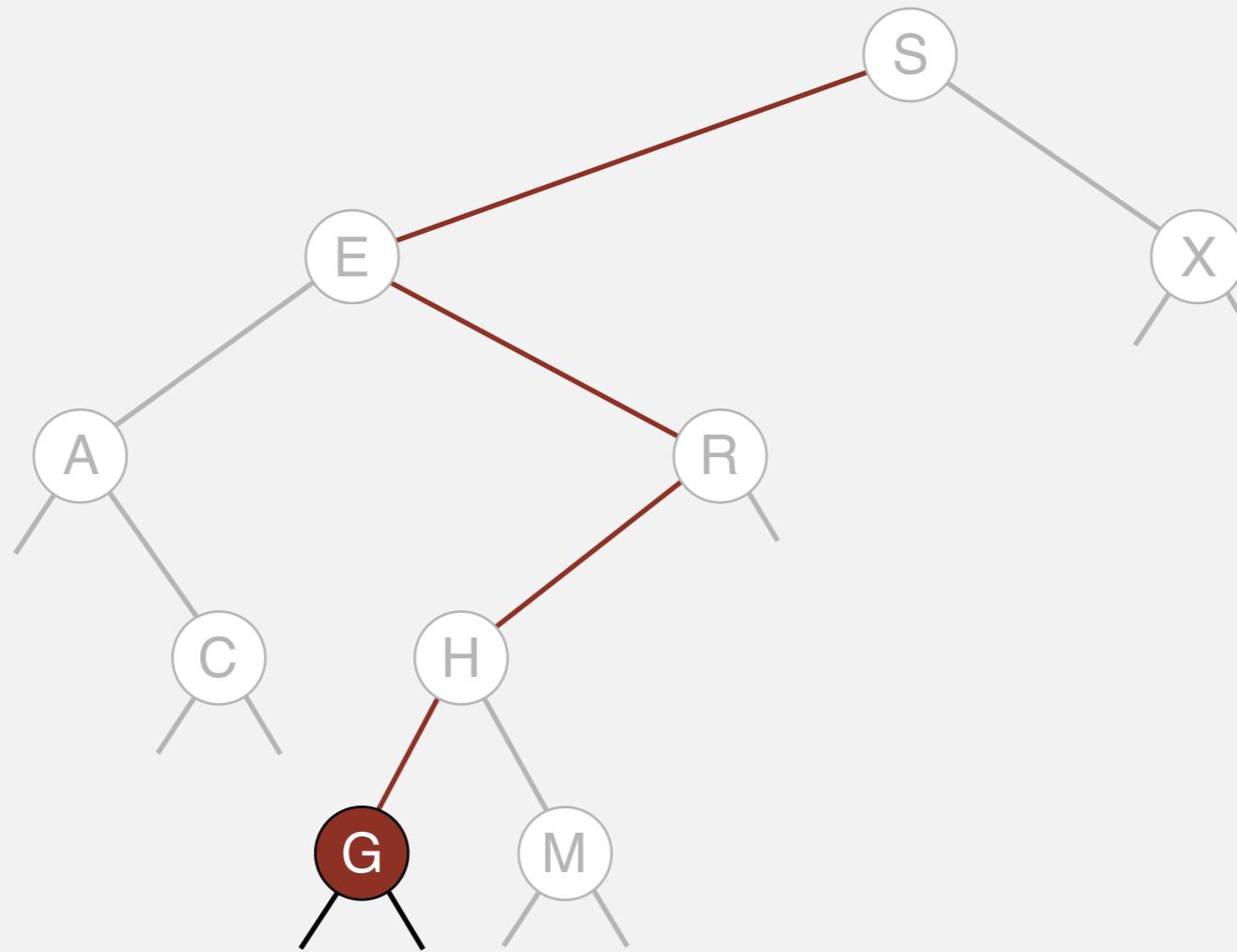
insert G



Binary search tree operations

Insert. If less, go left; if greater, go right; if null, insert.

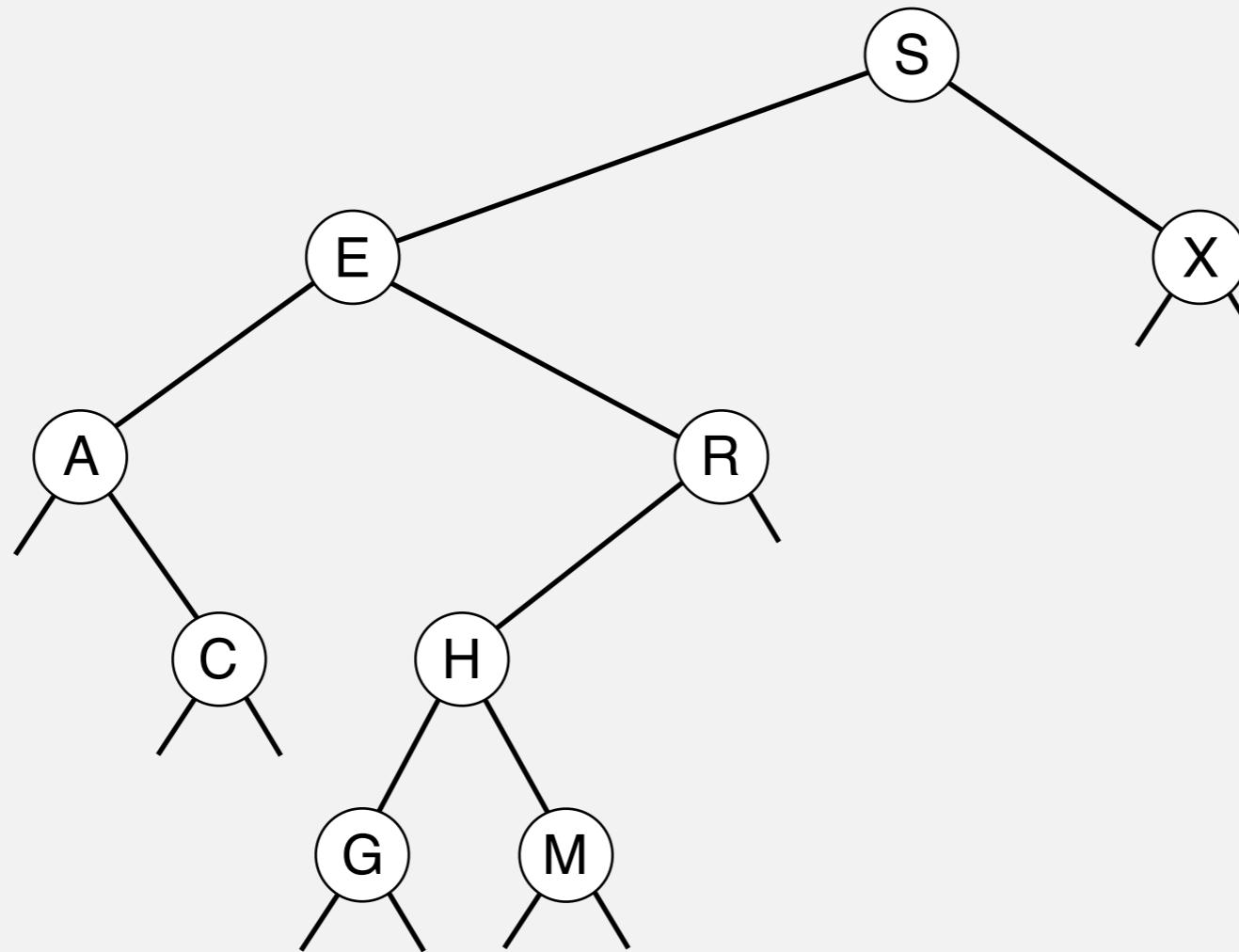
insert G



Binary search tree operations

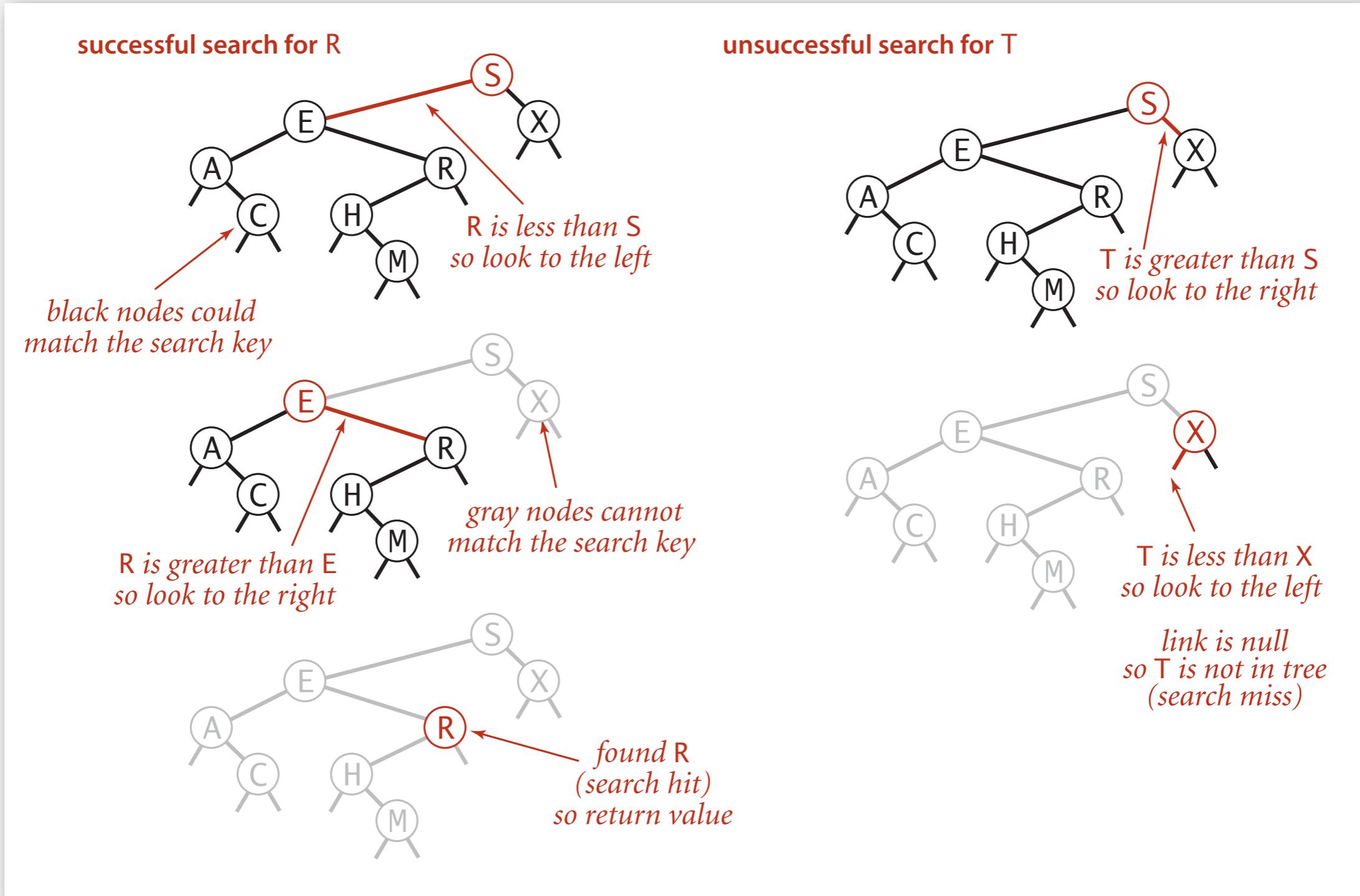
Insert. If less, go left; if greater, go right; if null, insert.

insert G



BST search

Get. Return value corresponding to given key, or `null` if no such key.



BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

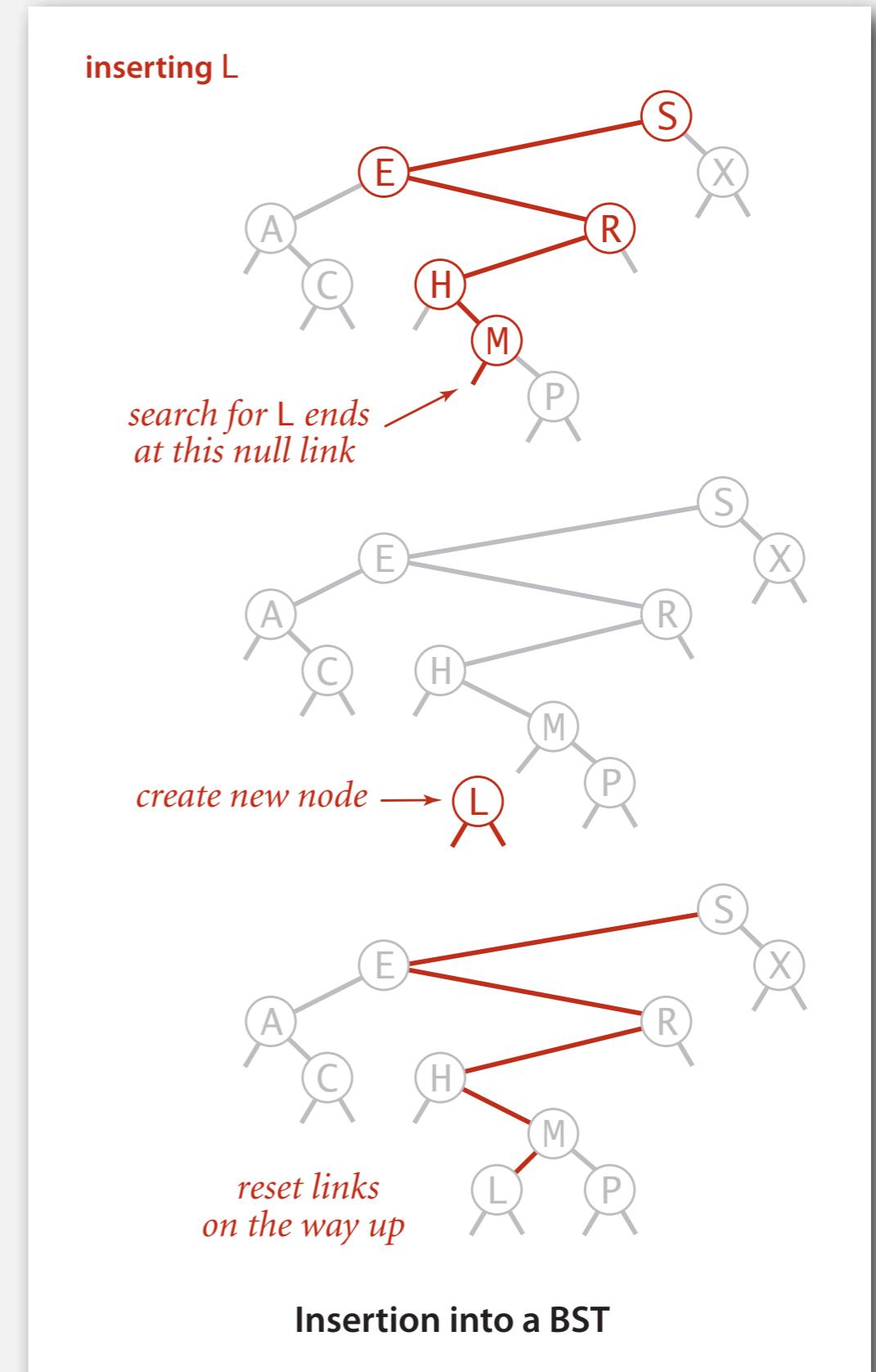
Cost. Number of compares is equal to $l + \text{depth of node}$.

BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree \Rightarrow reset value.
- Key not in tree \Rightarrow add new node.



BST insert: Java implementation

Put. Associate value with key.

```
public void put(Key key, Value val)
{   root = put(root, key, val);   }

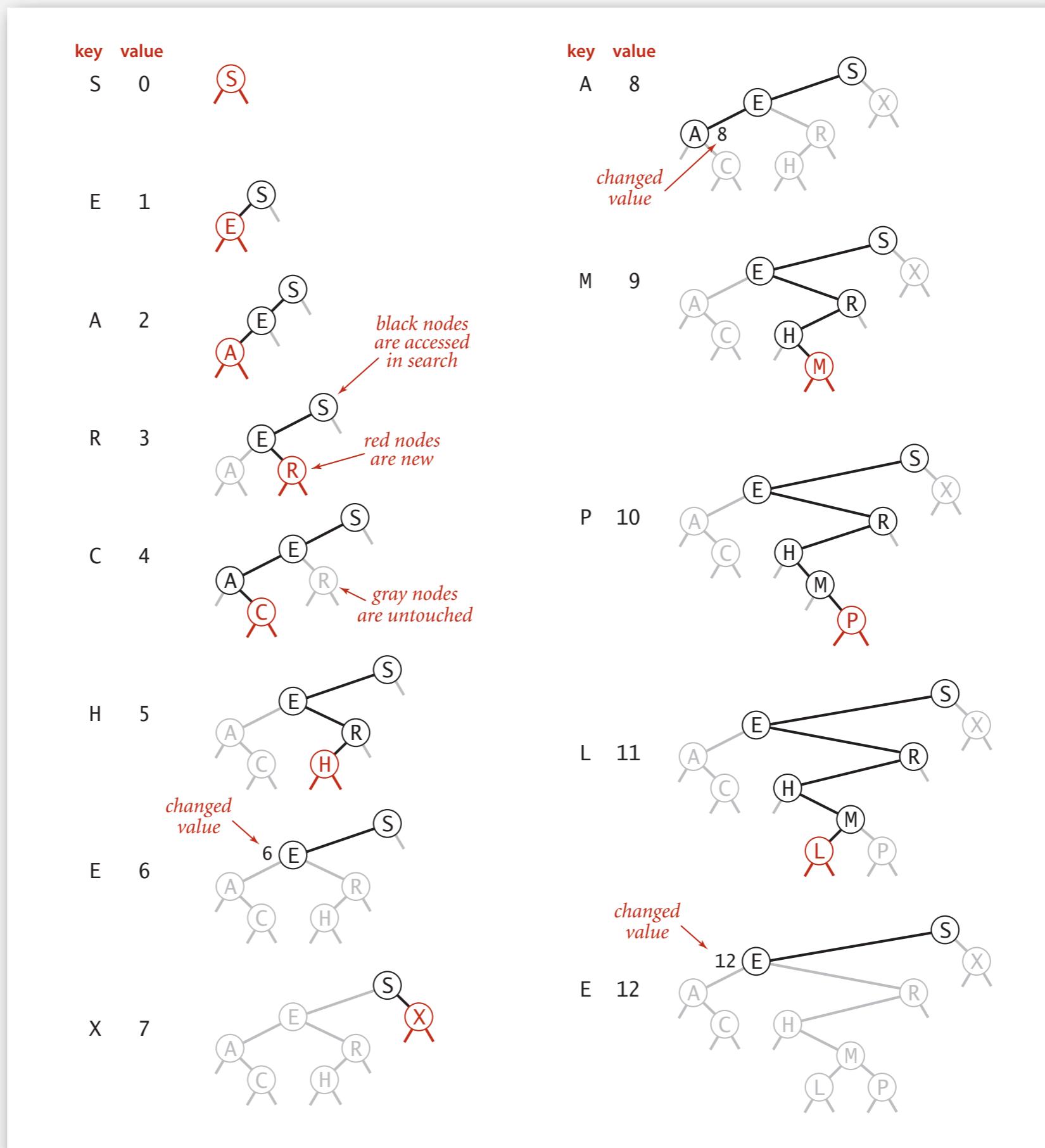
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0)
        x.left  = put(x.left,  key, val);
    else if (cmp > 0)
        x.right = put(x.right, key, val);
    else if (cmp == 0)
        x.val = val;
    return x;
}
```

concise, but tricky,
recursive code;
read carefully!

Always assign the subtree
returned from recursive
call to a child, but does it actually
change in each call ?

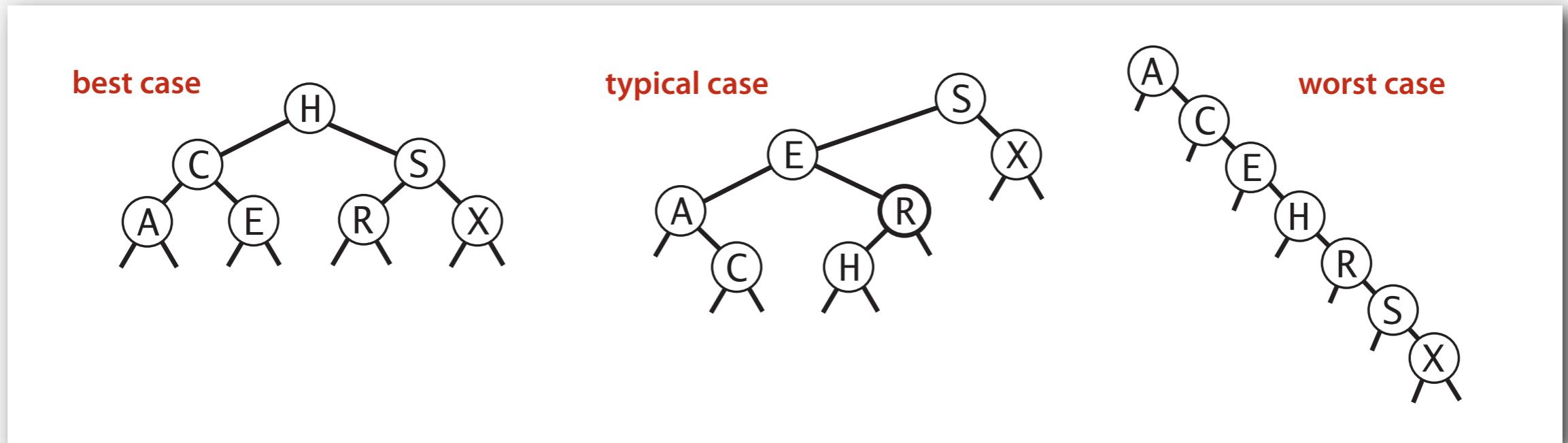
Cost. Number of compares is equal to $1 + \text{depth of node}$.

BST trace: standard indexing client



Tree shape

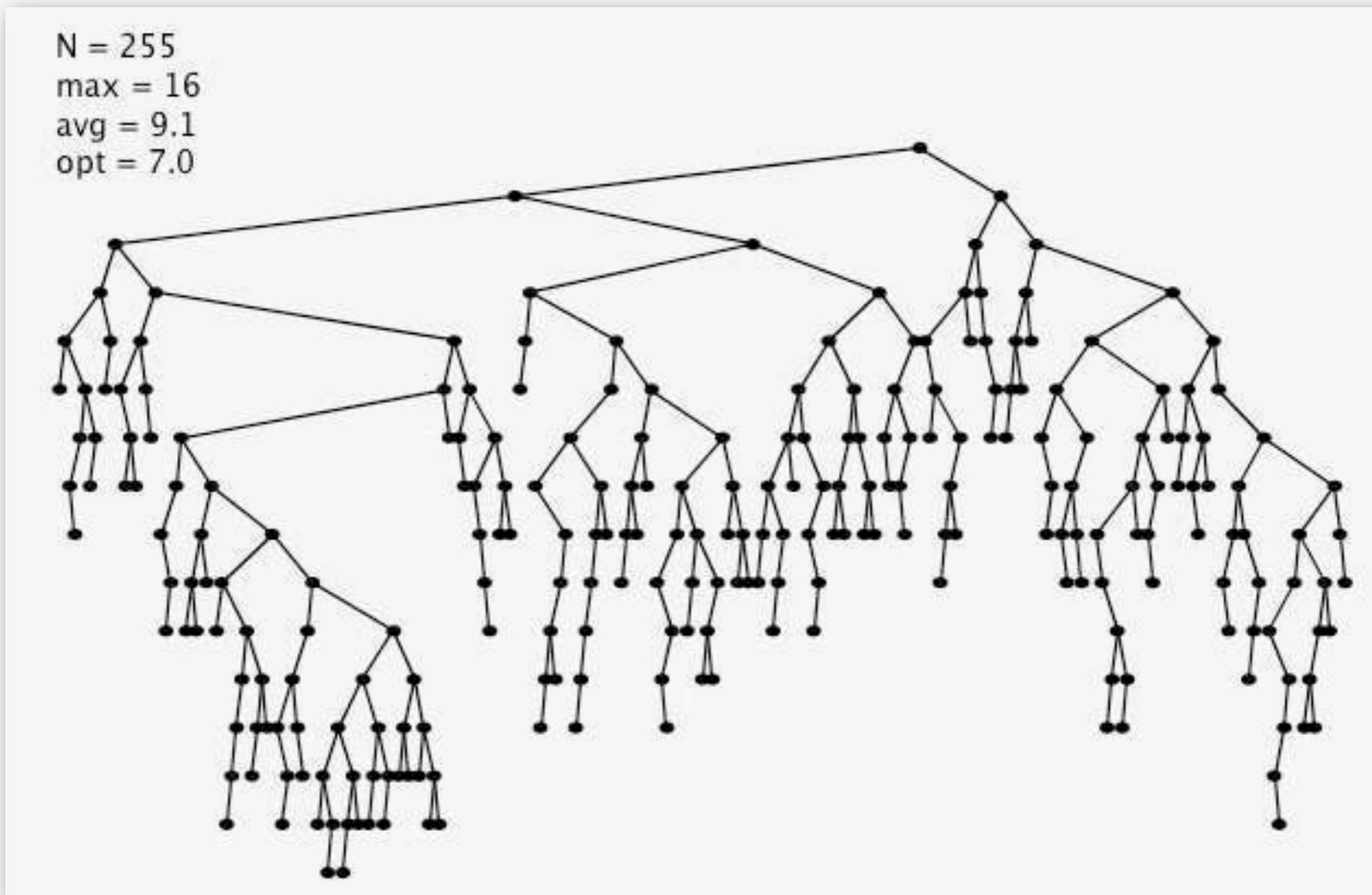
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to $l + \text{depth of node}$.



Remark. Tree shape depends on order of insertion.

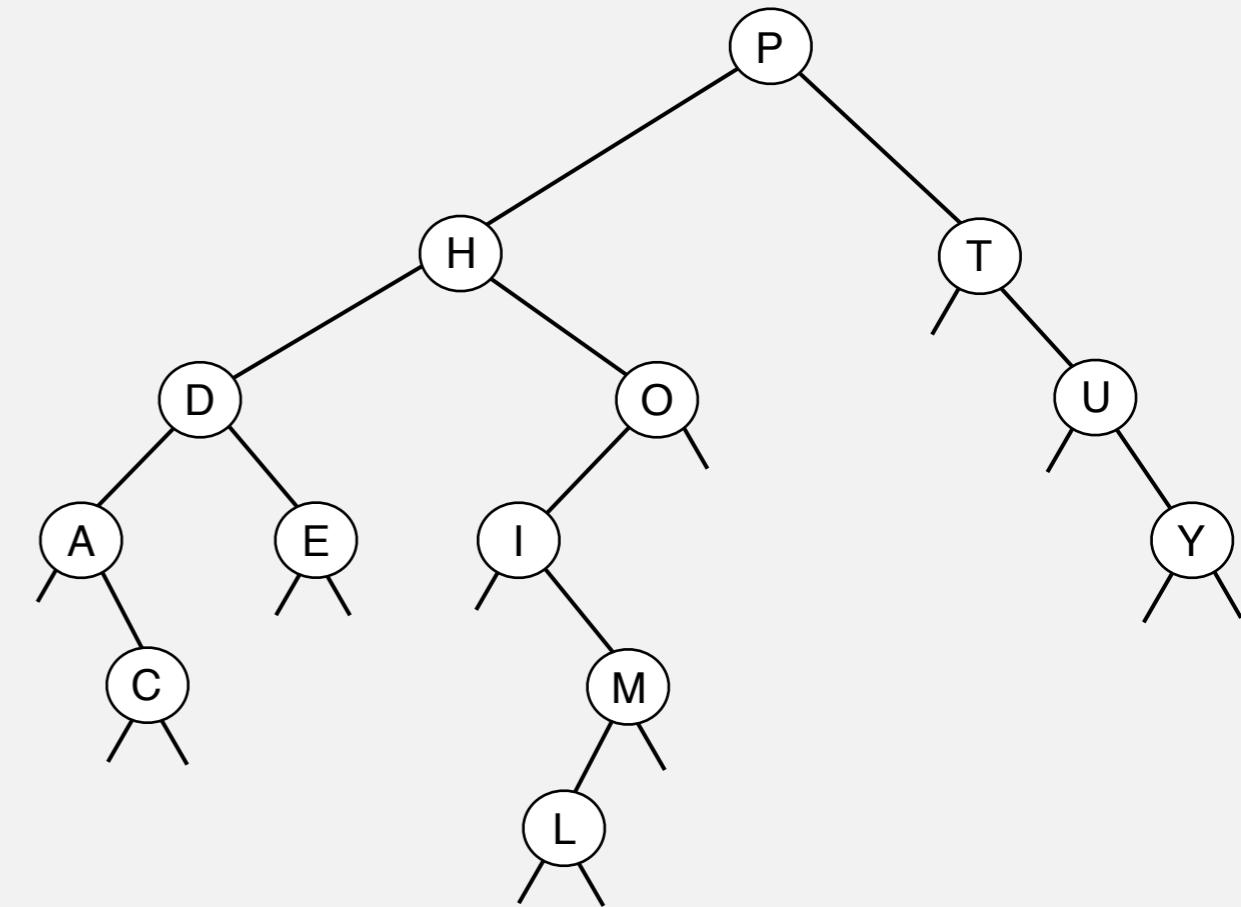
BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



Remark. Correspondence is 1-1 if array has no duplicate keys.

BSTs: mathematical analysis

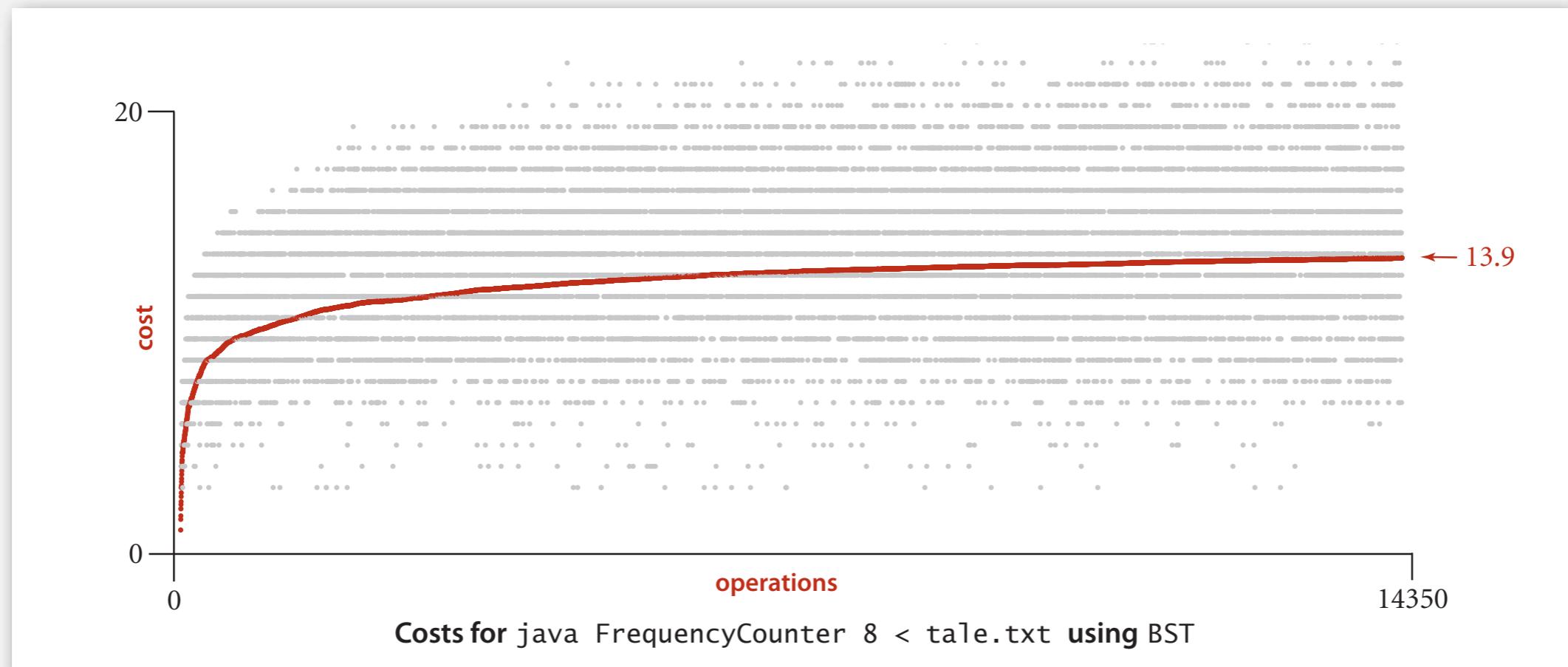
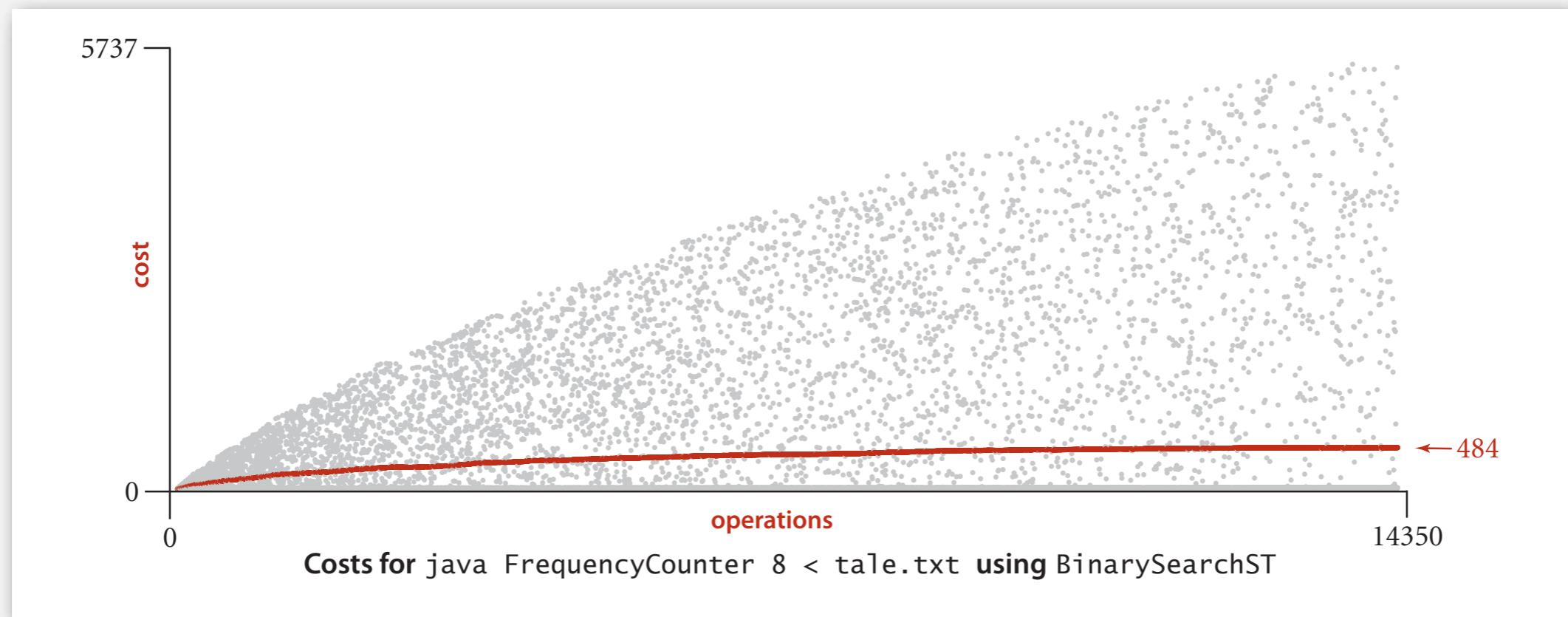
Proposition. If N distinct keys are inserted into a BST in **random** order, the expected number of compares for a search/insert is $O(\log N)$.

Pf. 1-1 correspondence with quicksort partitioning.

But... Worst-case height is N .

(exponentially small chance when keys are inserted in random order)

ST implementations: frequency counter



ST implementations: summary

implementation	guarantee		average case		ordered ops?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	N/2	N	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	$\lg N$	N/2	yes	<code>compareTo()</code>
BST	N	N	$\lg N$	$\lg N$	<i>stay tuned</i>	<code>compareTo()</code>

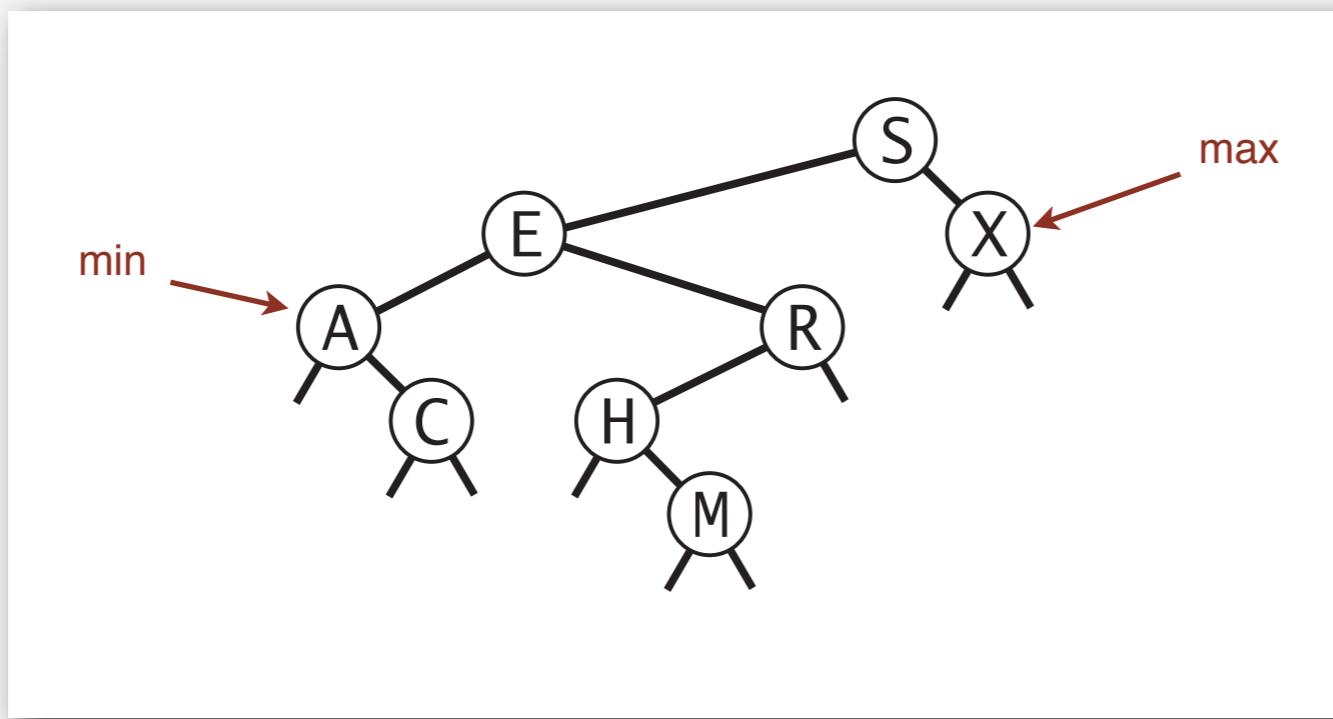
BINARY SEARCH TREES

- ▶ BSTs
- ▶ Ordered operations
- ▶ Deletion

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

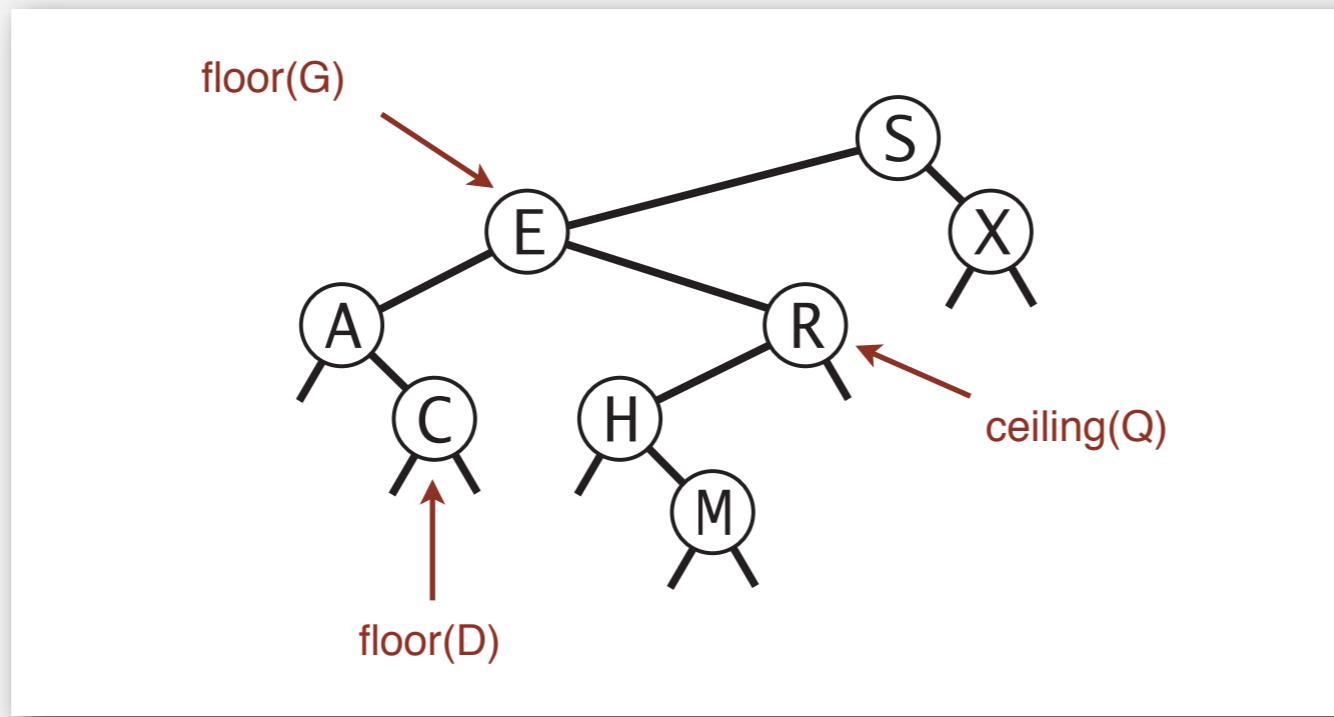


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key \leq to a given key.

Ceiling. Smallest key \geq to a given key.



Q. How to find the floor /ceiling?

Computing the floor

Case 1. [k equals the key at root]

The floor of k is k .

Case 2. [k is less than the key at root]

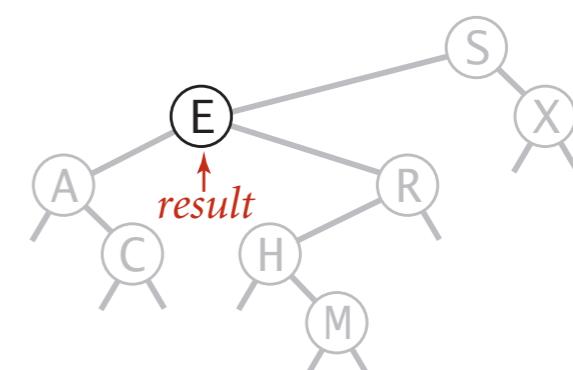
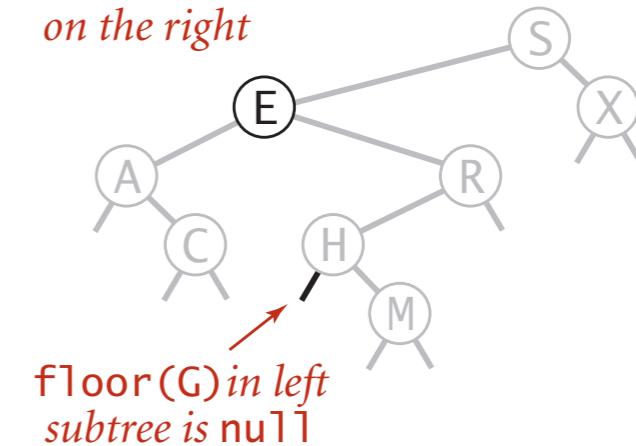
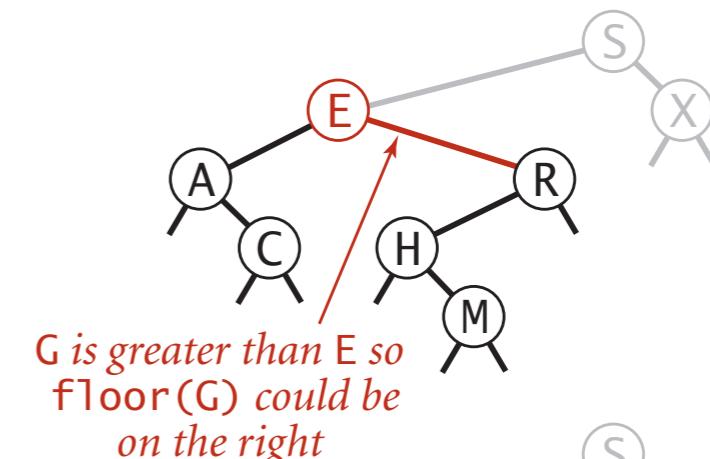
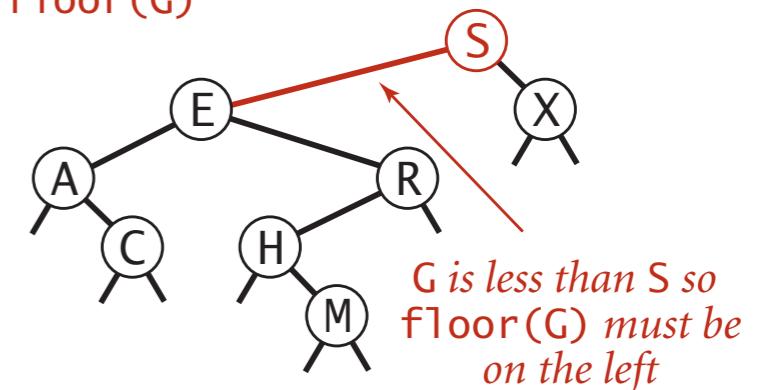
The floor of k is in the left subtree.

Case 3. [k is greater than the key at root]

The floor of k is in the right subtree

(if there is **any** key $\leq k$ in right subtree);
otherwise it is the key in the root.

finding $\text{floor}(G)$



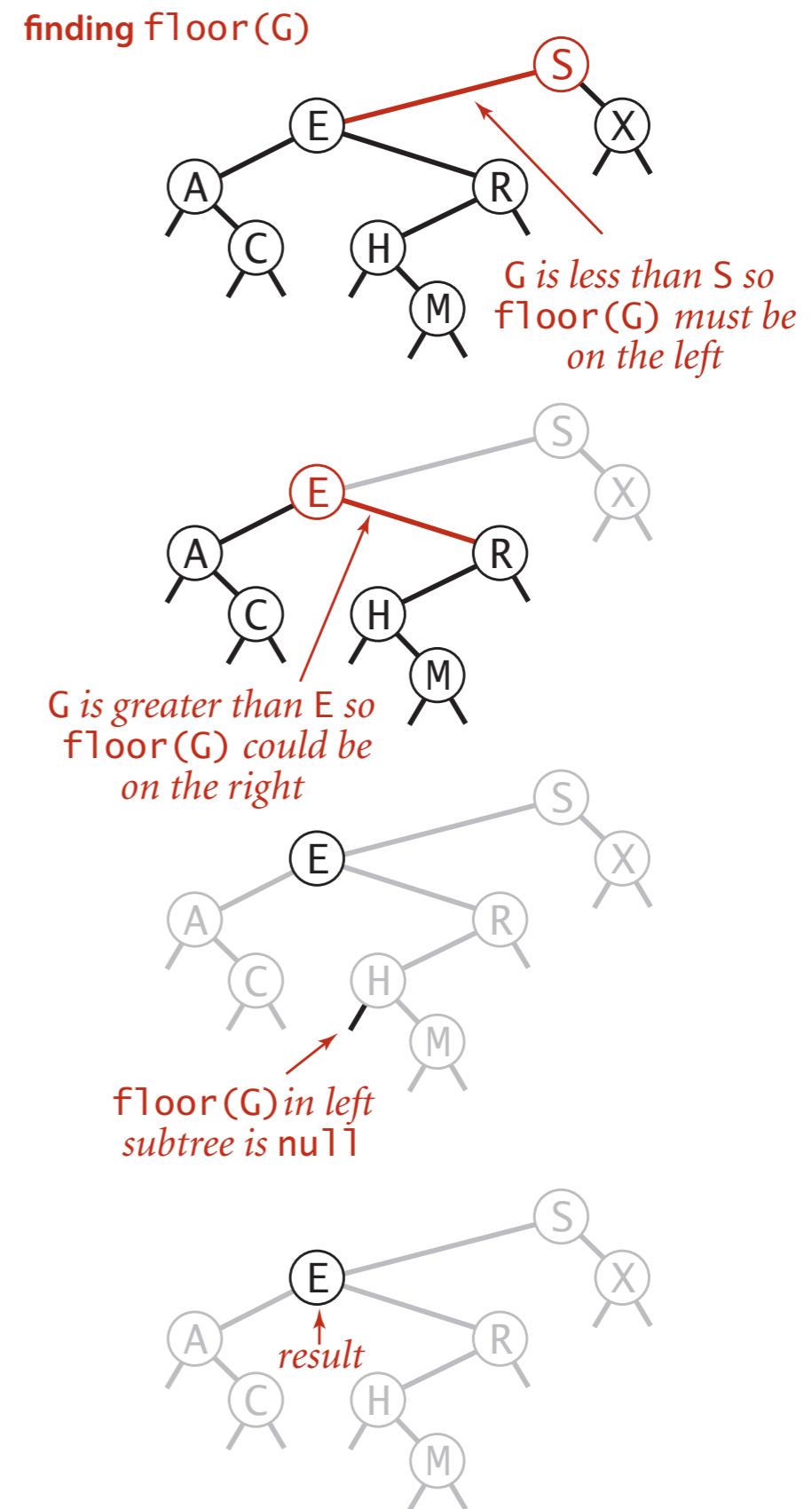
Computing the floor

```
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;

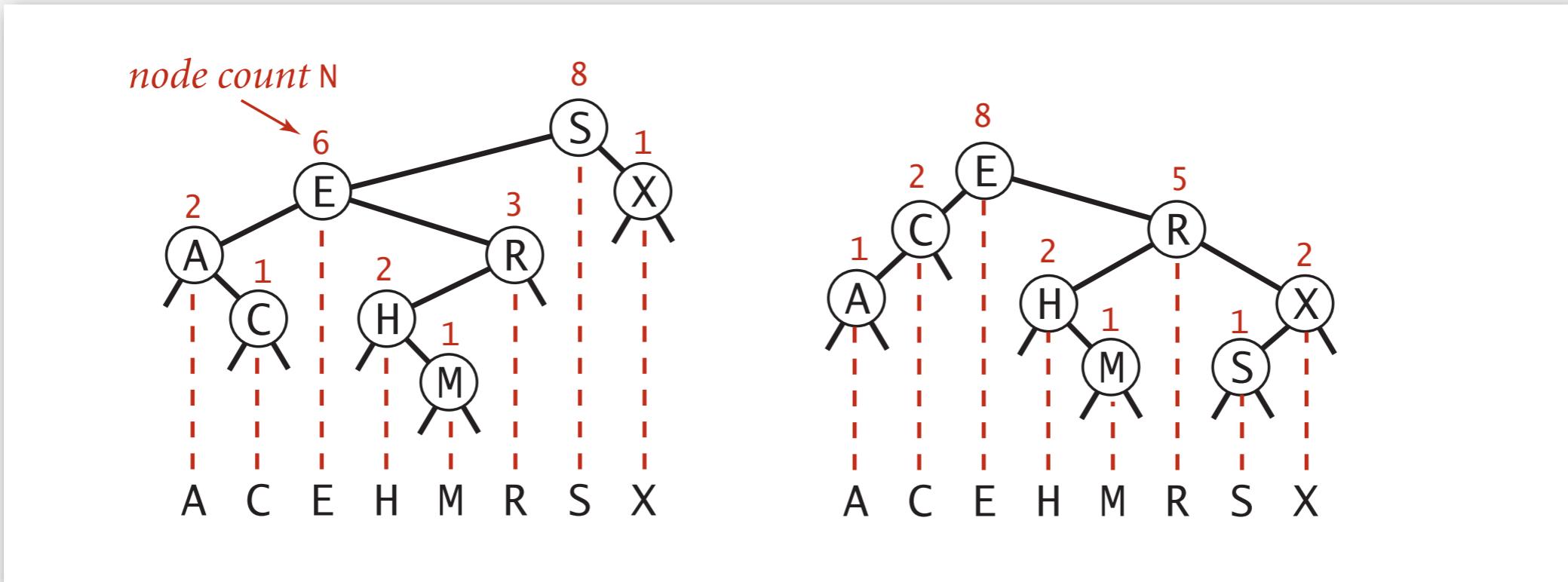
    if (cmp < 0) return floor(x.left, key);

    Node t = floor(x.right, key);
    if (t != null) return t;
    else return x;
}
```



Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node; to implement `size()`, return the count at the root.



Remark. This facilitates efficient implementation of `rank()` and `select()`.

BST implementation: subtree counts

```
private class Node  
{  
    private Key key;  
    private Value val;  
    private Node left;  
    private Node right;  
    private int N;  
}
```

number of nodes
in subtree

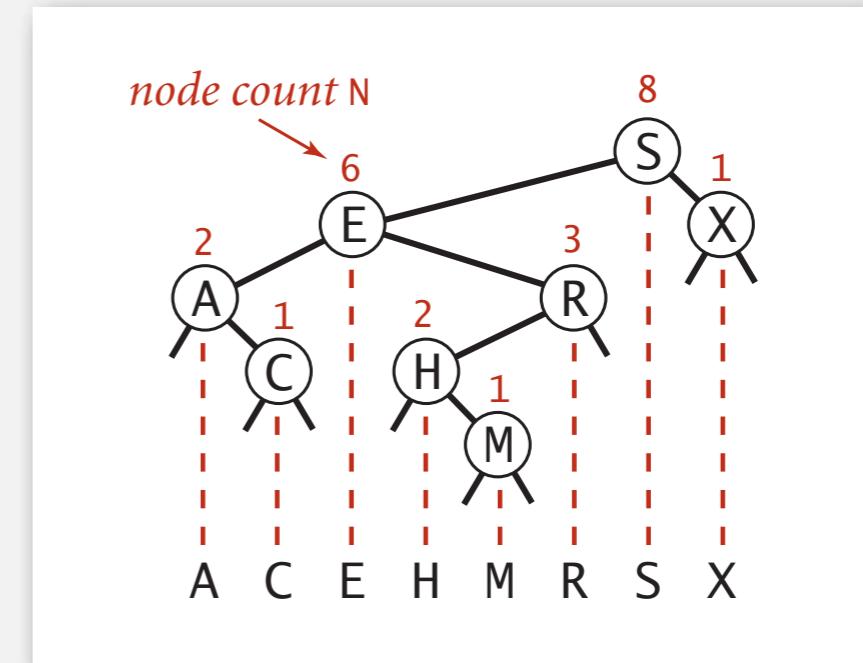
```
public int size()  
{    return size(root);    }  
  
private int size(Node x)  
{  
    if (x == null) return 0;  
    return x.N;    }  
    ↑  
    ok to call when  
    x is null
```

```
private Node put(Node x, Key key, Value val)  
{  
    if (x == null) return new Node(key, val);  
    int cmp = key.compareTo(x.key);  
    if (cmp < 0) x.left = put(x.left, key, val);  
    else if (cmp > 0) x.right = put(x.right, key, val);  
    else if (cmp == 0) x.val = val;  
    x.N = 1 + size(x.left) + size(x.right);  
    return x;  
}
```

Rank

Rank. How many keys $< k$?

Easy recursive algorithm (4 cases!)



```
public int rank(Key key)
{   return rank(key, root); }

private int rank(Key key, Node x)
{
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else if (cmp == 0) return size(x.left);
}
```

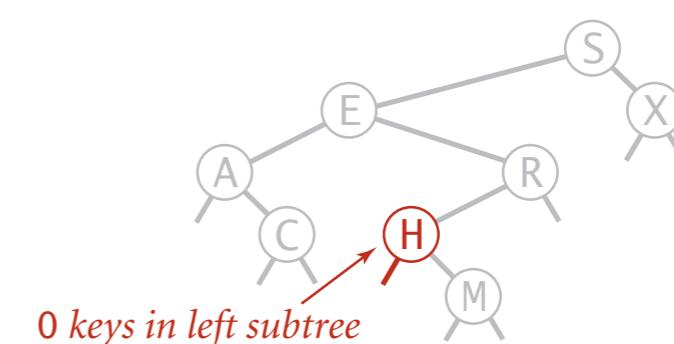
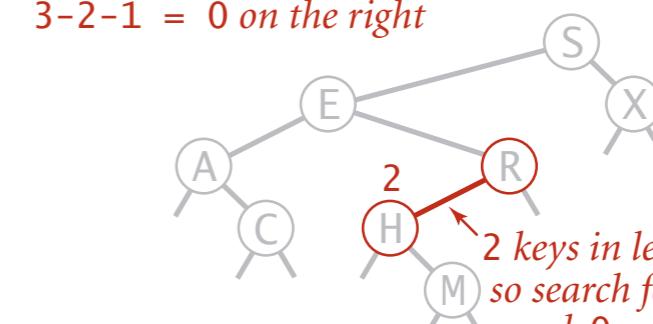
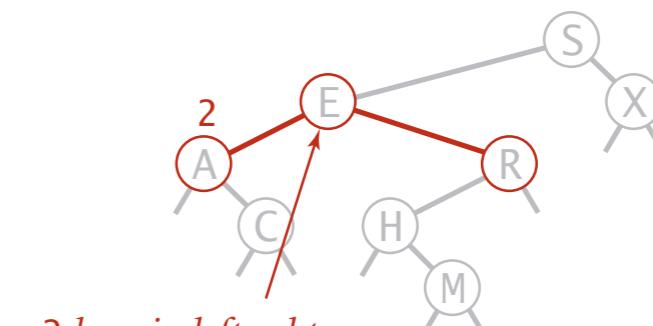
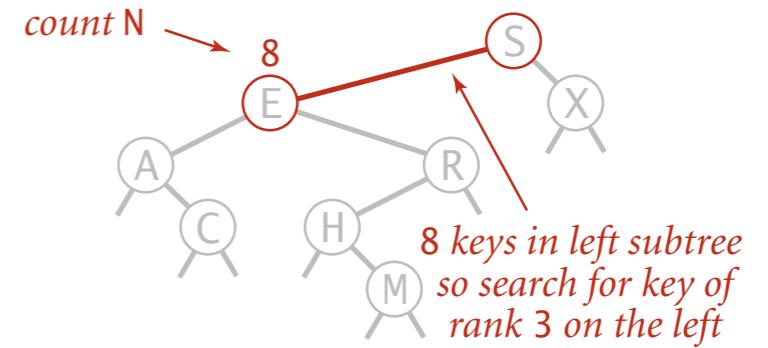
Selection

Select. Key of given rank.

```
public Key select(int k)
{
    if (k < 0) return null;
    if (k >= size()) return null;
    Node x = select(root, k);
    return x.key;
}

private Node select(Node x, int k)
{
    if (x == null) return null;
    int t = size(x.left);
    if (t > k)
        return select(x.left, k);
    else if (t < k)
        return select(x.right, k-t-1);
    else if (t == k)
        return x;
}
```

finding select(3)
the key of rank 3

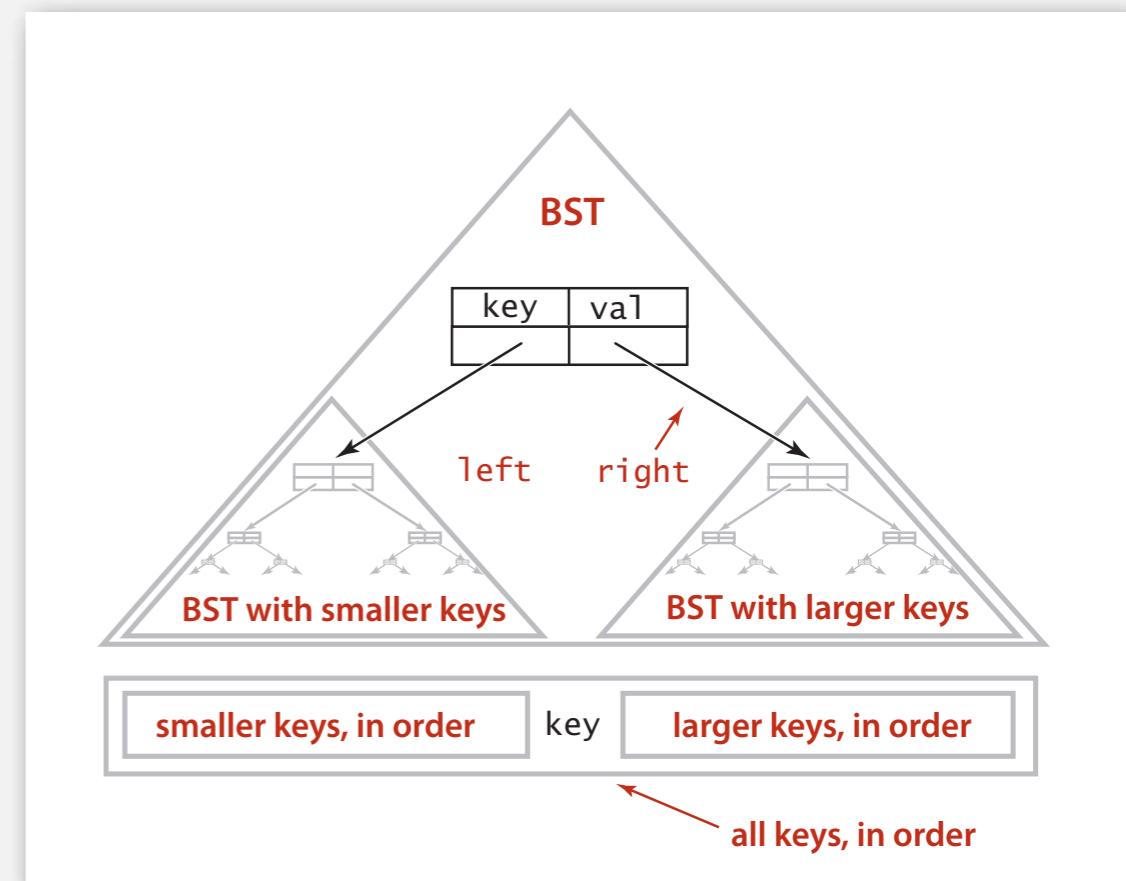


Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```



Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
inorder(S)
    inorder(E)
        inorder(A)
        enqueue A
    inorder(C)
        enqueue C
    enqueue E
inorder(R)
    inorder(H)
        enqueue H
    inorder(M)
        enqueue M
    enqueue R
enqueue S
inorder(X)
    enqueue X
```

A
C
E

H
M
R
S

X

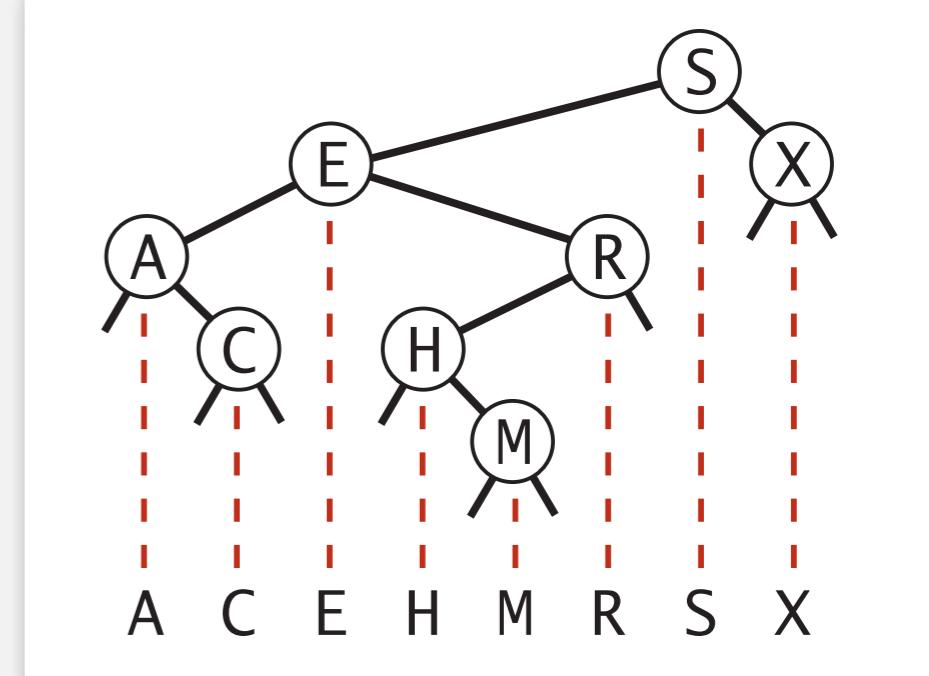
S
S E
S E A

S E A C

S E R
S E R H

S E R H M

S X



recursive calls

queue

function call stack

BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	N	$\lg N$	h
insert	I	N	h
min / max	N	I	h
floor / ceiling	N	$\lg N$	h
rank	N	$\lg N$	h
select	N	I	h
ordered iteration	$N \log N$	N	N

h = height of BST
(proportional to $\log N$)
if keys inserted in random order)

order of growth of running time of ordered symbol table operations

BINARY SEARCH TREES

- ▶ BSTs
- ▶ Ordered operations
- ▶ Deletion

ST implementations: summary

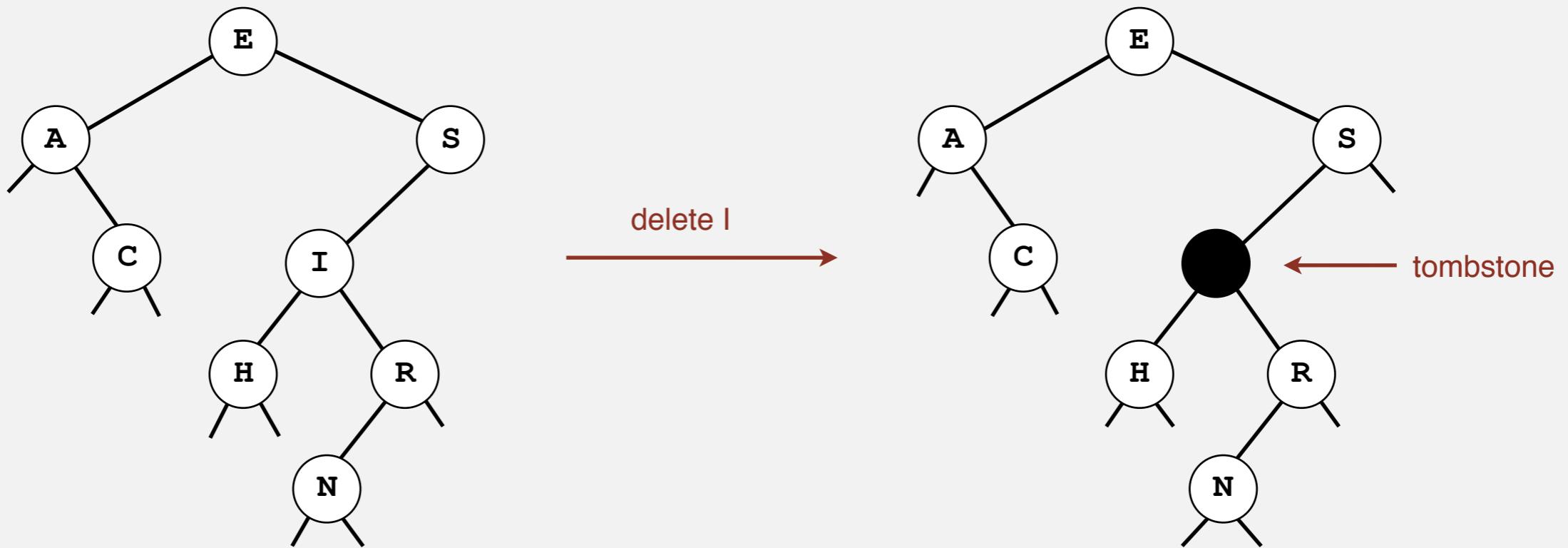
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$\lg N$	$\lg N$???	yes	<code>compareTo()</code>

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to `null`.
- Leave key in tree to guide searches (but don't consider it equal to search key).



Cost. $O(\log N')$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) overload.

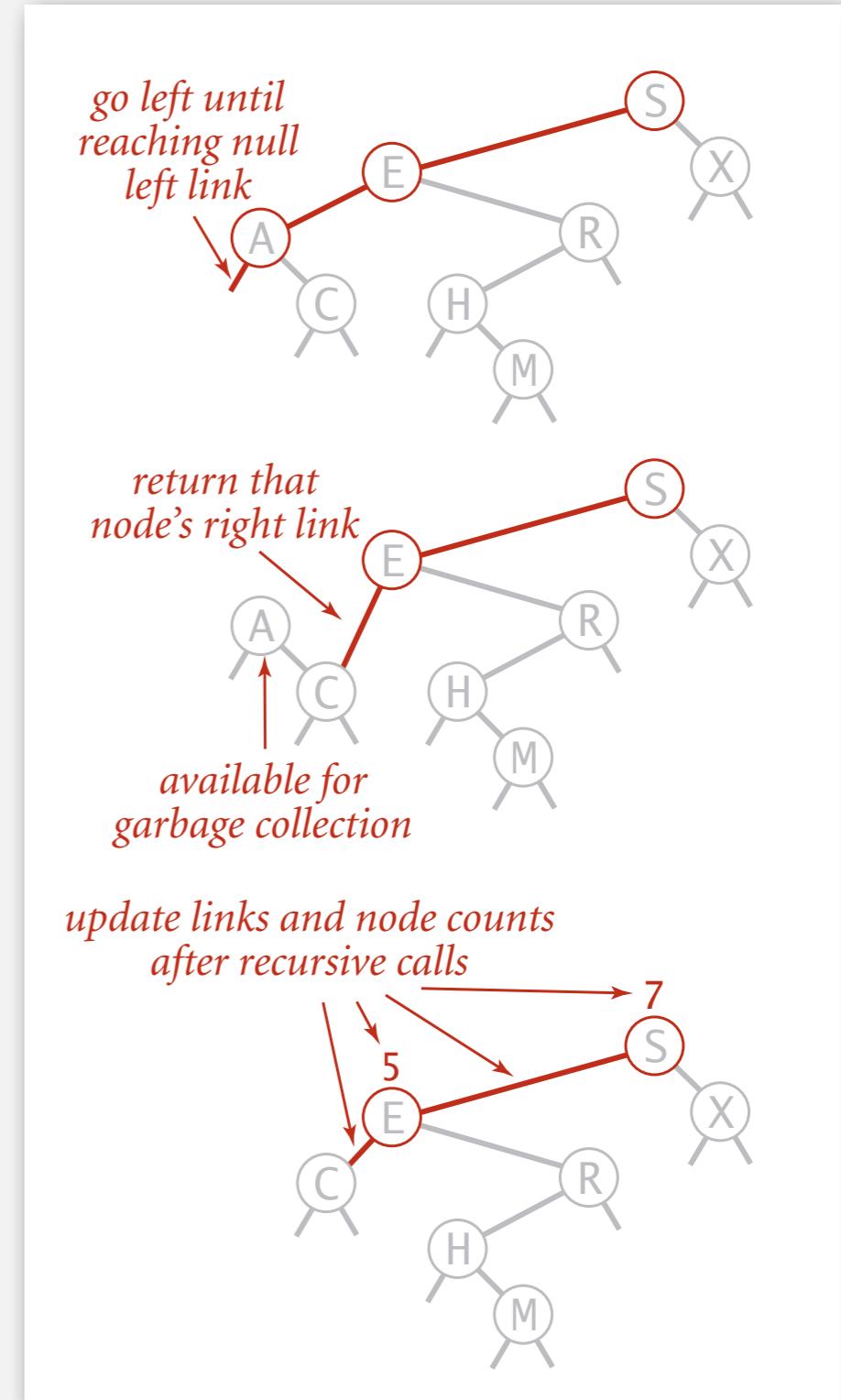
Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

```
public void deleteMin()
{   root = deleteMin(root);   }

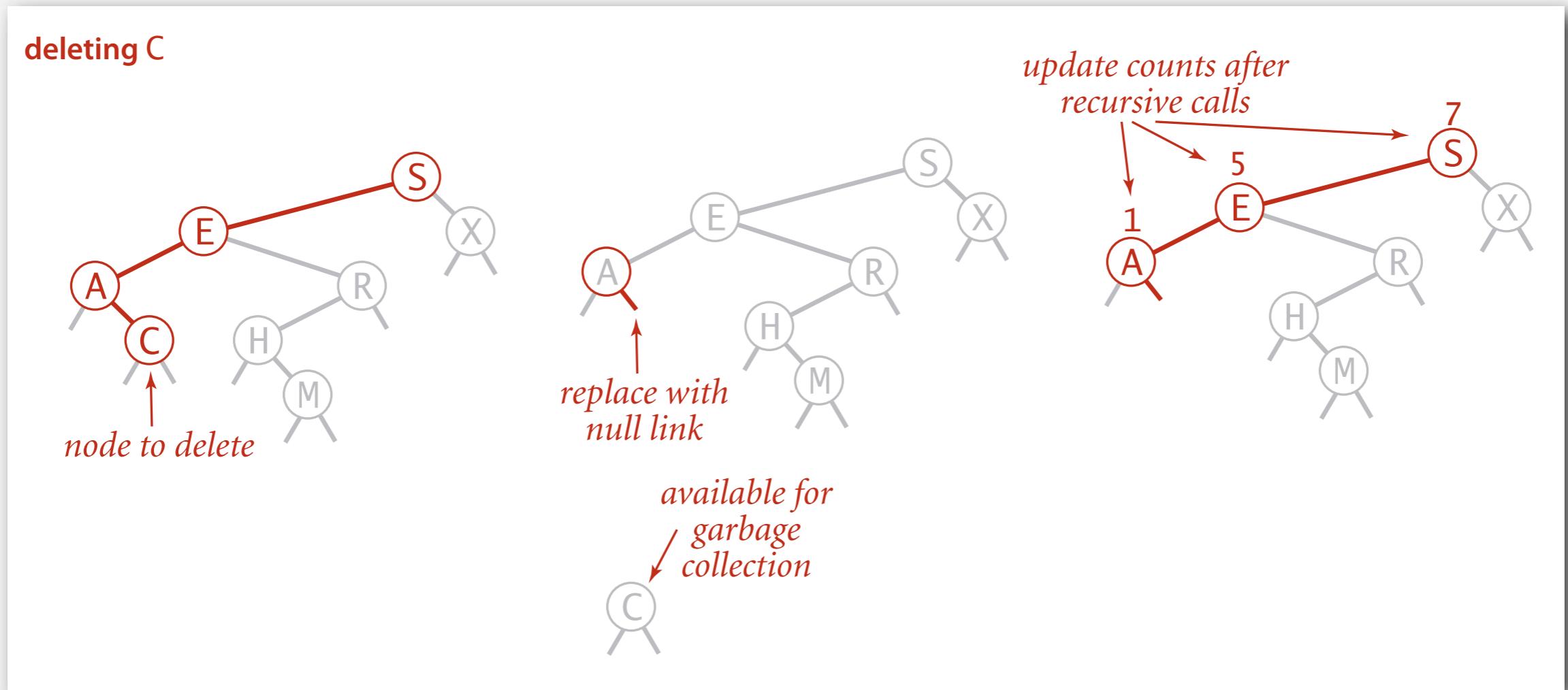
private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```



Hibbard deletion

To delete a node with key k : search for node t containing key k .

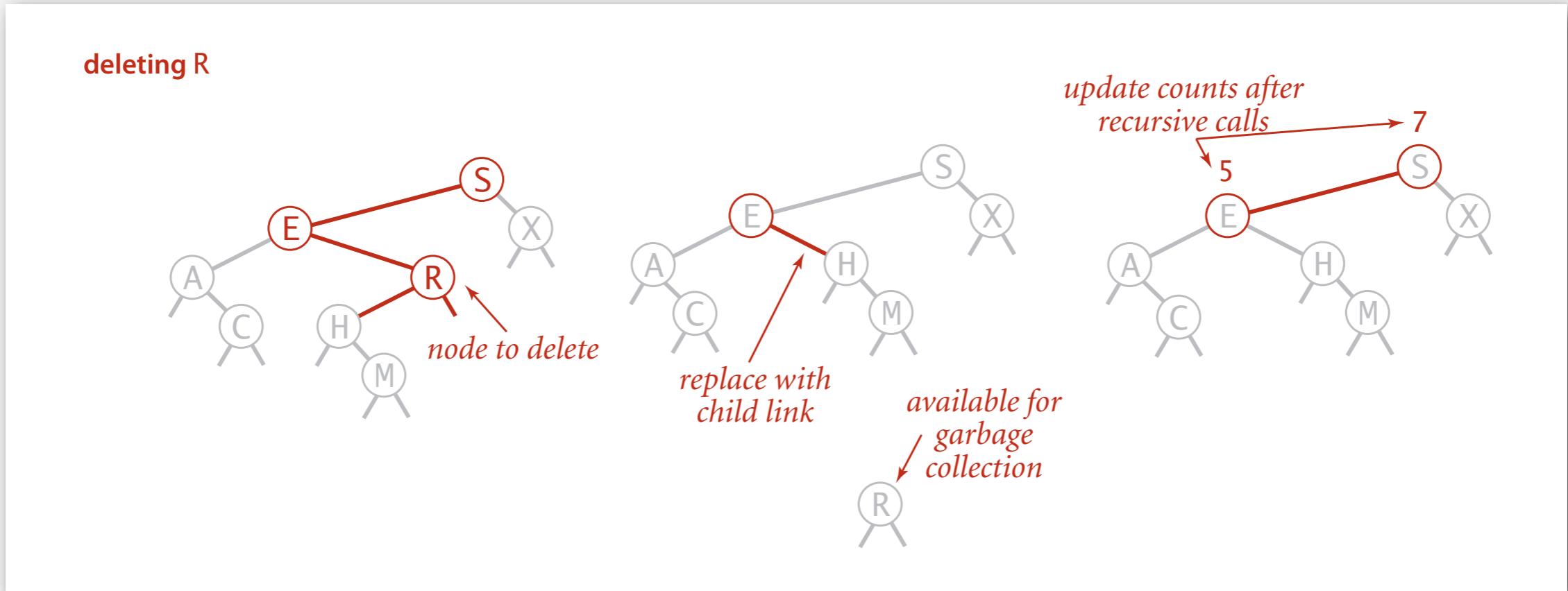
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

To delete a node with key k : search for node t containing key k .

Case I. [1 child] Delete t by replacing parent link.

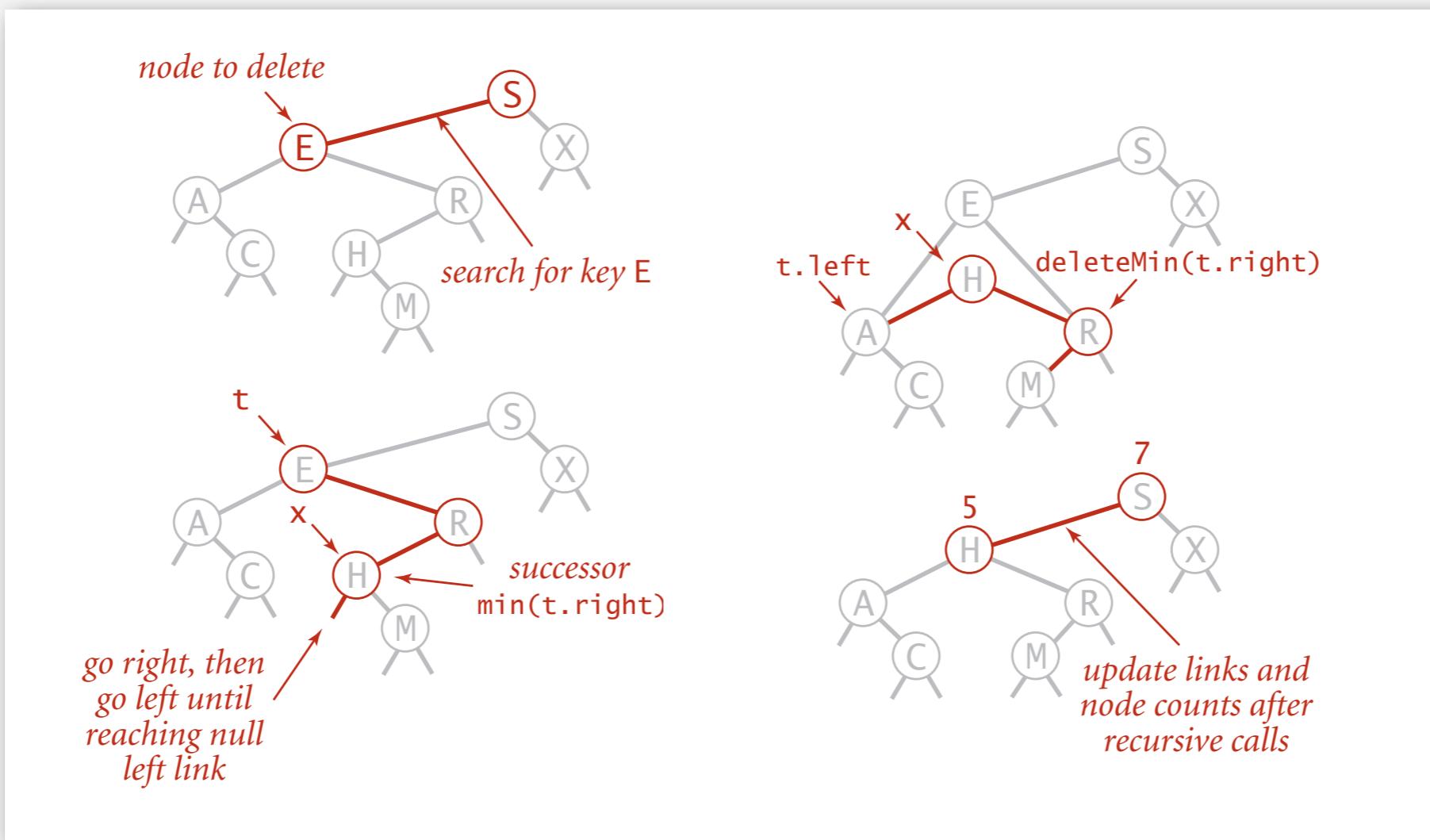


Hibbard deletion

To delete a node with key k : search for node t containing key k .

Case 2. [2 children]

- Find successor x of t .
 - Delete the minimum in t 's right subtree.
 - Put x in t 's spot.
- ← x has no left child
← but don't garbage collect x
← still a BST



Hibbard deletion: Java implementation

```
public void delete(Key key)
{   root = delete(root, key);  }

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key); ← search for key
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left; ← no right child
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right); ← replace with successor
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1; ← update subtree
    return x;                                counts
}
```

search for key

no right child

replace with
successor

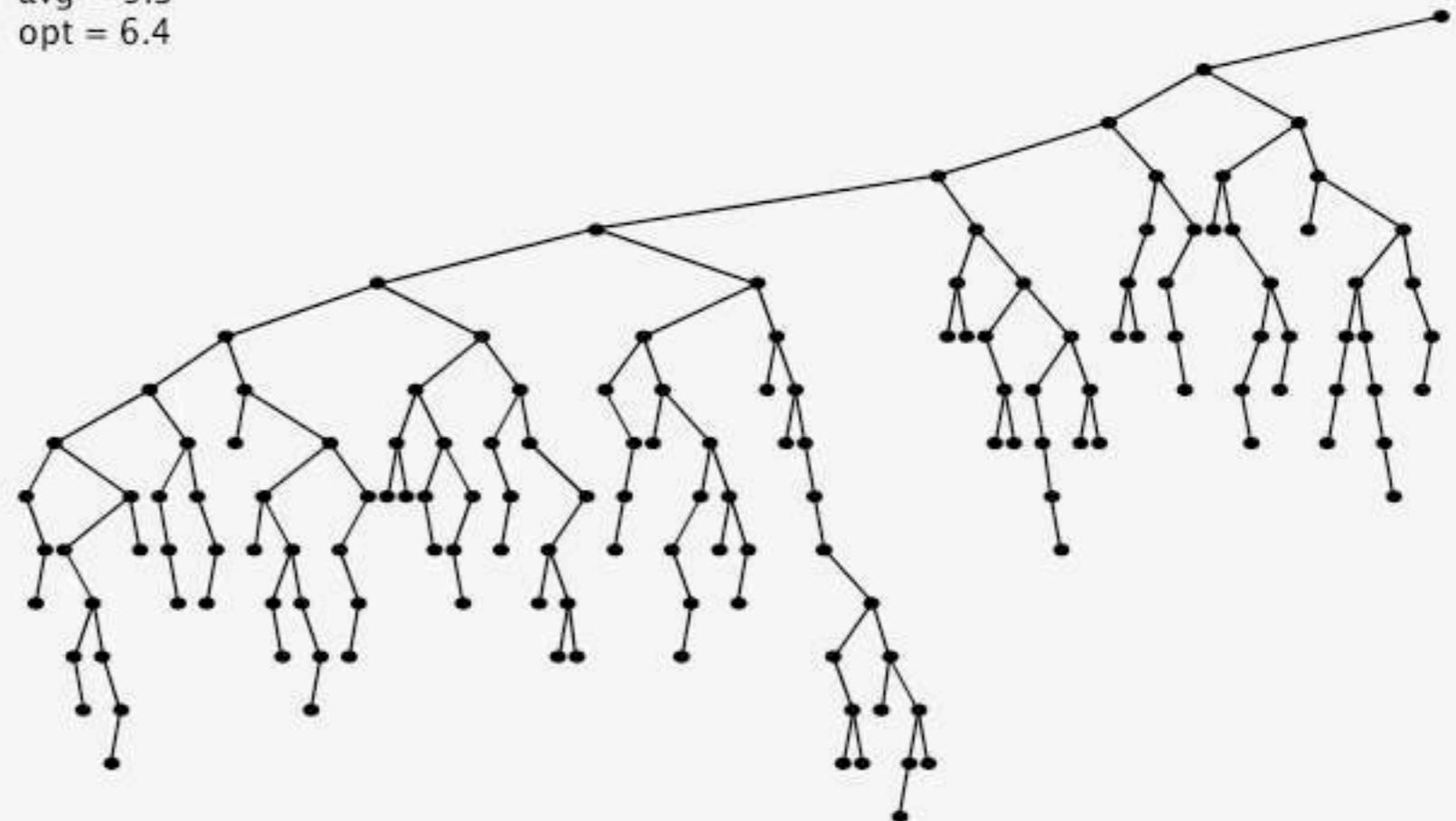
update subtree
counts

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.

If we always
delete from the
same side, the
shape of tree
will be not
random, the
right subtrees
are trimmed!

N = 150
max = 16
avg = 9.3
opt = 6.4

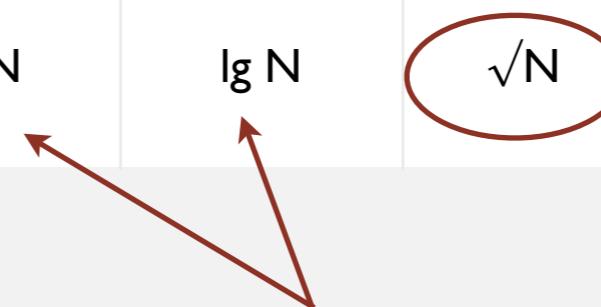


Surprising consequence. Trees not random (!) $\Rightarrow \sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$\lg N$	$\lg N$	\sqrt{N}	yes	<code>compareTo()</code>



other operations also become \sqrt{N}
if deletions allowed

Red-black BST. Guarantee logarithmic performance for all operations.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

BALANCED TREES

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

BALANCED SEARCH TREES

- ▶ **2-3 search trees**
- ▶ **Red-black BSTs**
- ▶ **B-trees**
- ▶ **Geometric applications of BSTs**

Text

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$?	yes	<code>compareTo()</code>
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	yes	<code>compareTo()</code>

► **Challenge.** Guarantee performance.

BALANCED SEARCH TREES

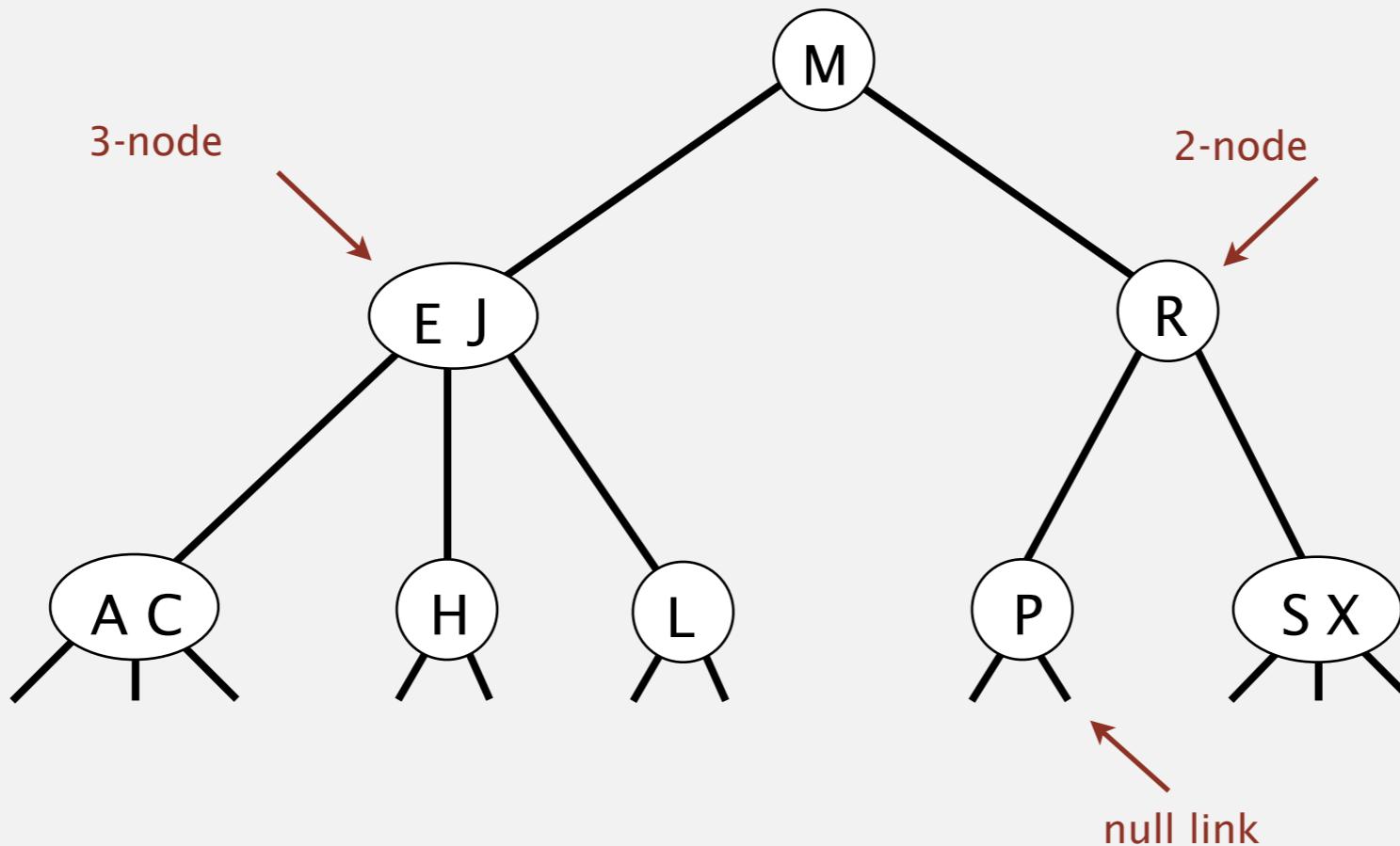
- ▶ 2-3 search trees
- ▶ Red-black BSTs
- ▶ B-trees
- ▶ Geometric applications of BSTs

2-3 tree

You can read it as 2 or 3 children tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

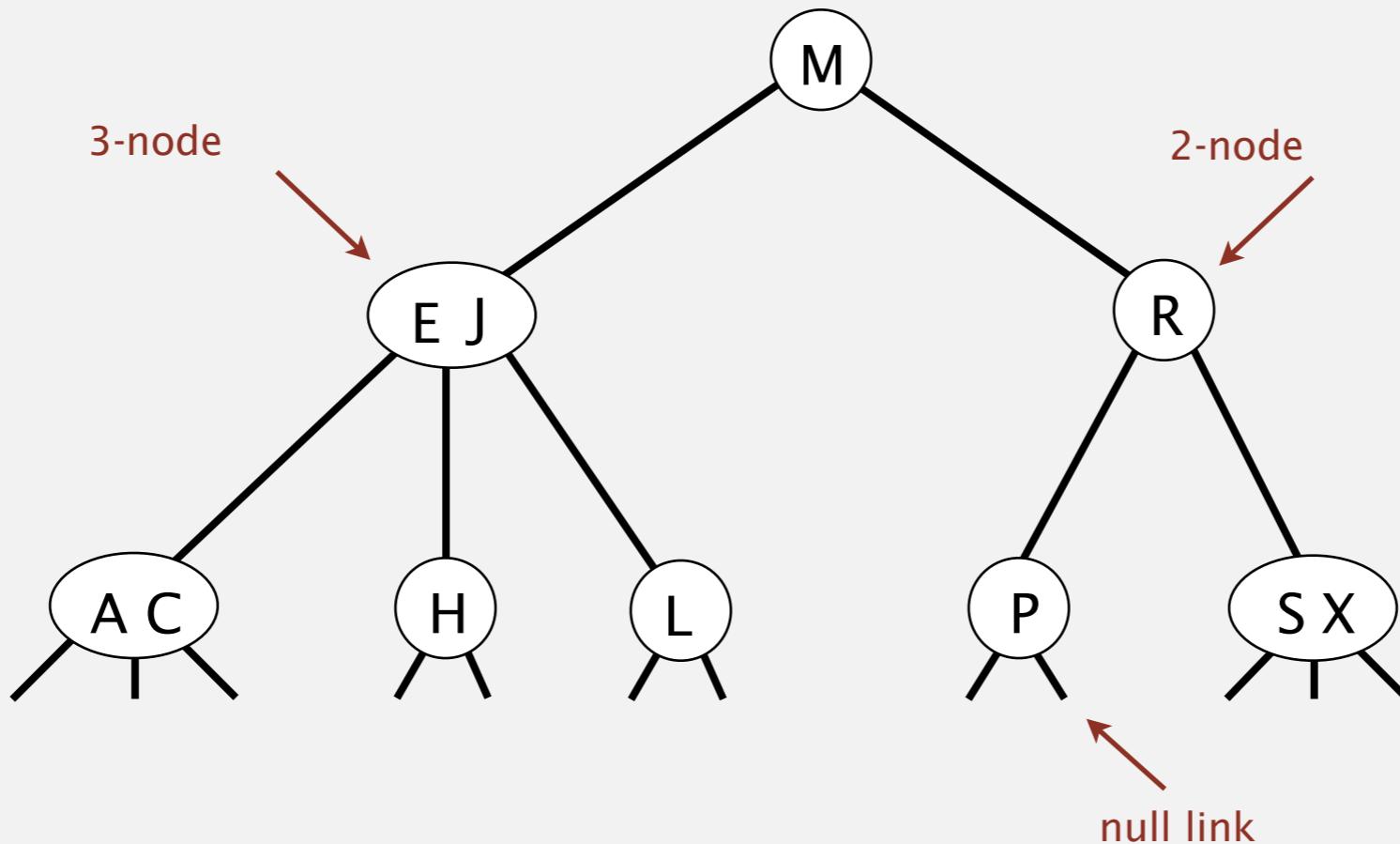


2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Our Aim is Perfect balance. Every path from root to null link has same length.



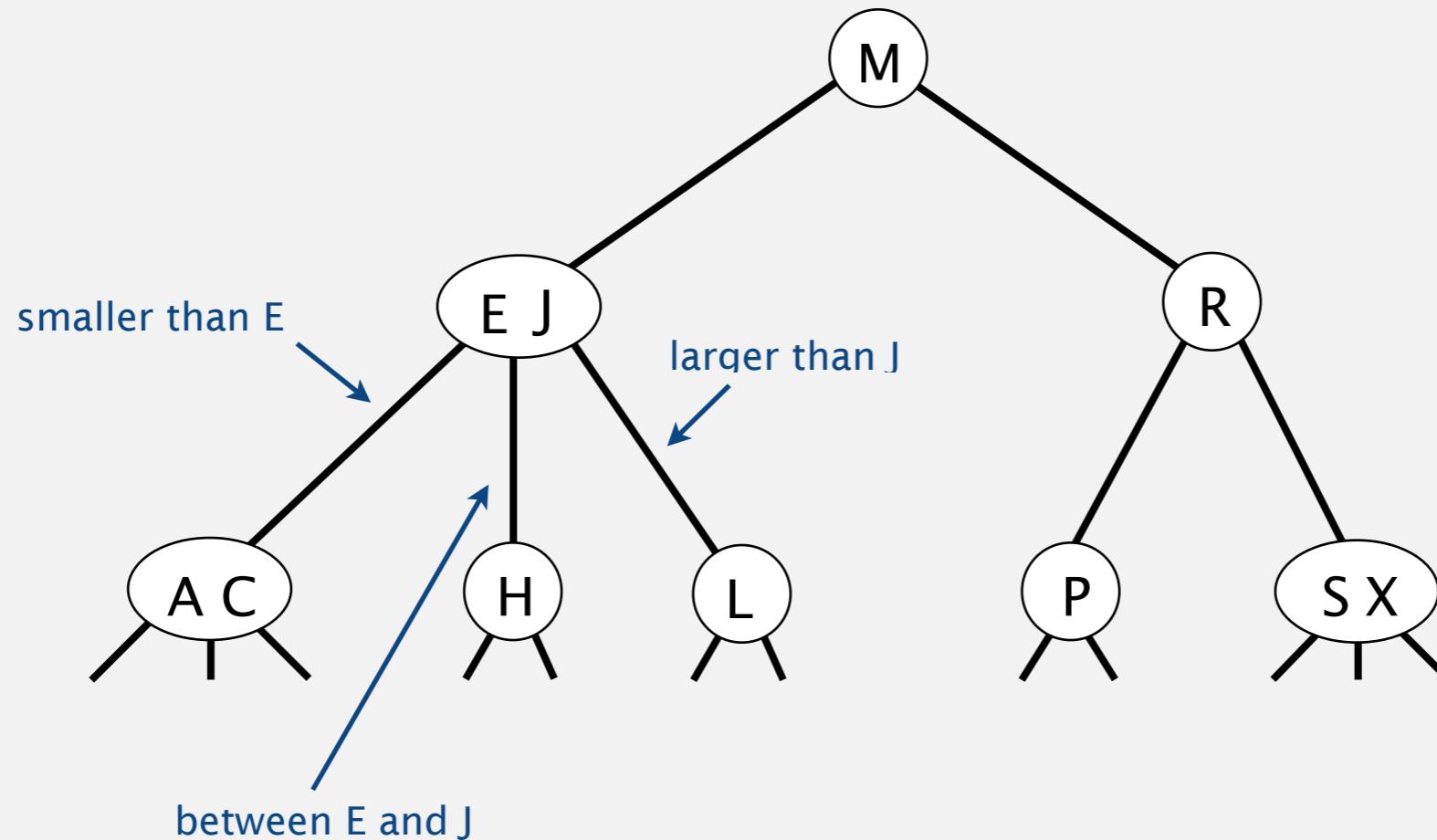
2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Perfect balance. Every path from root to null link has same length.

Symmetric order. Inorder traversal yields keys in ascending order.

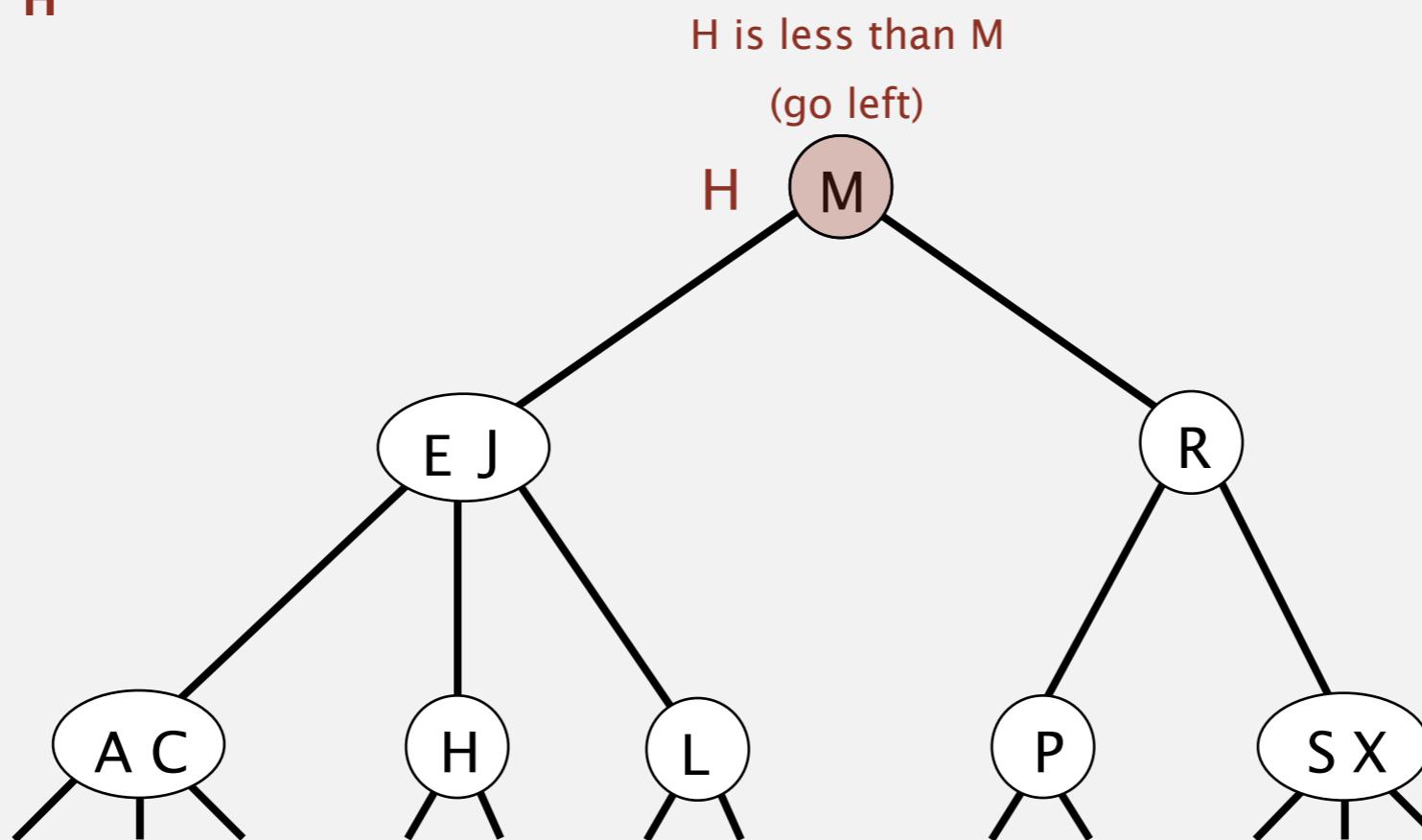


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H

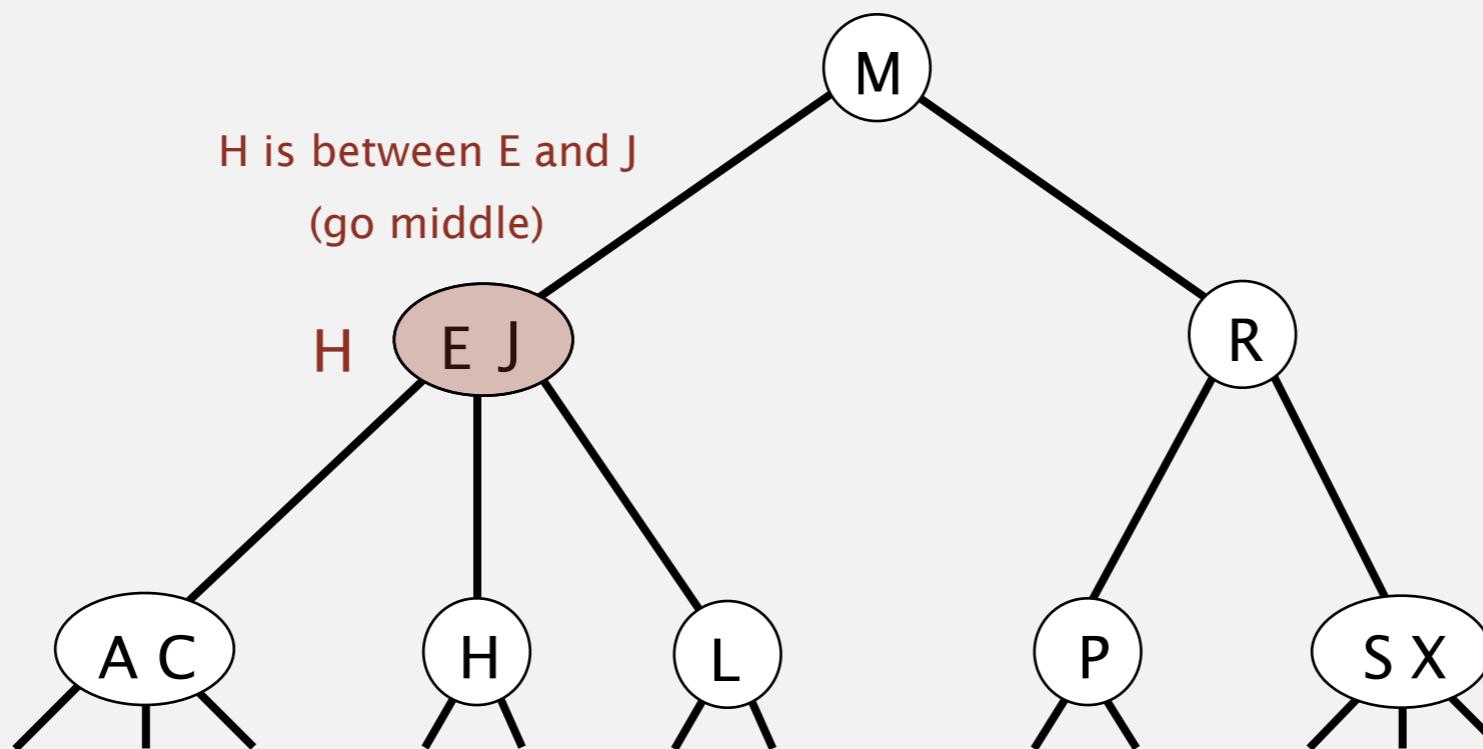


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H

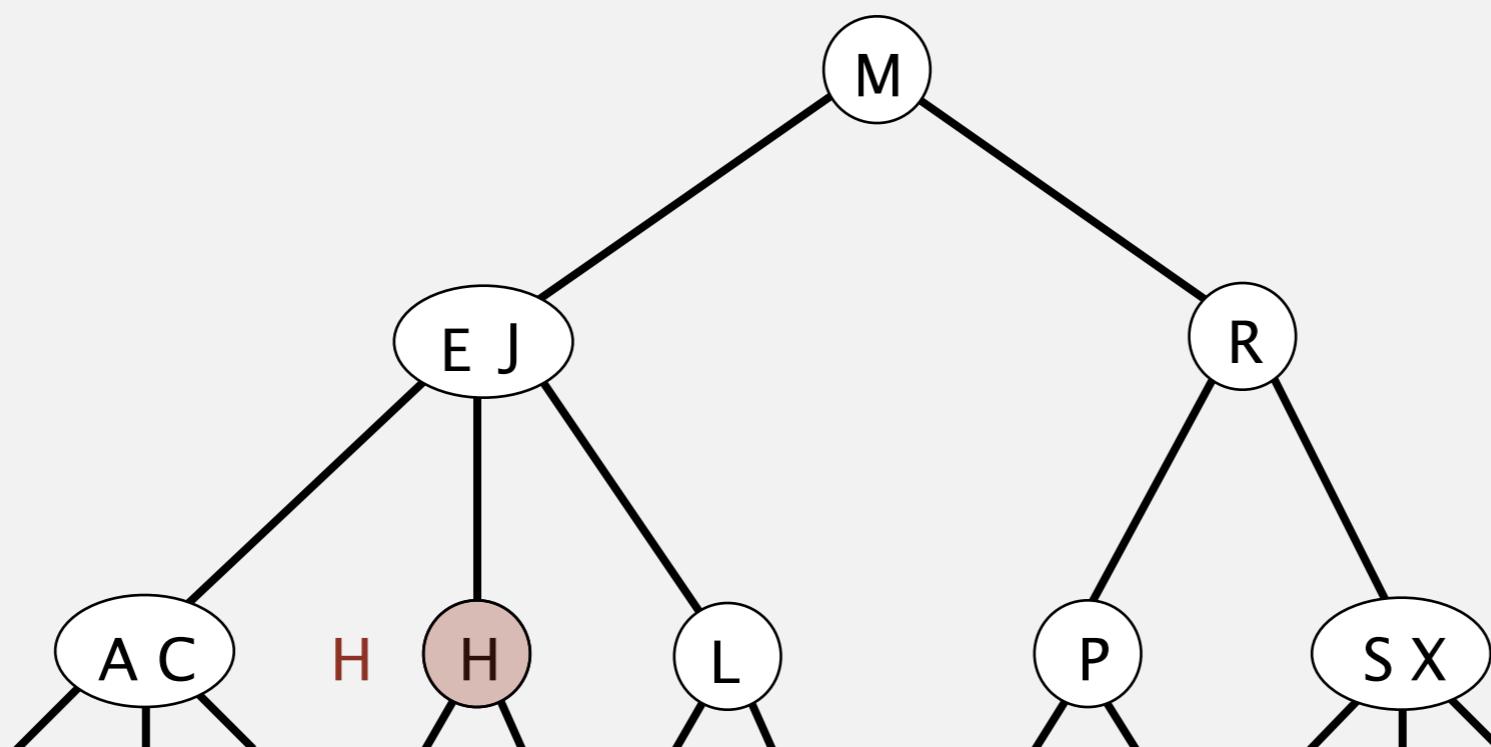


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for H



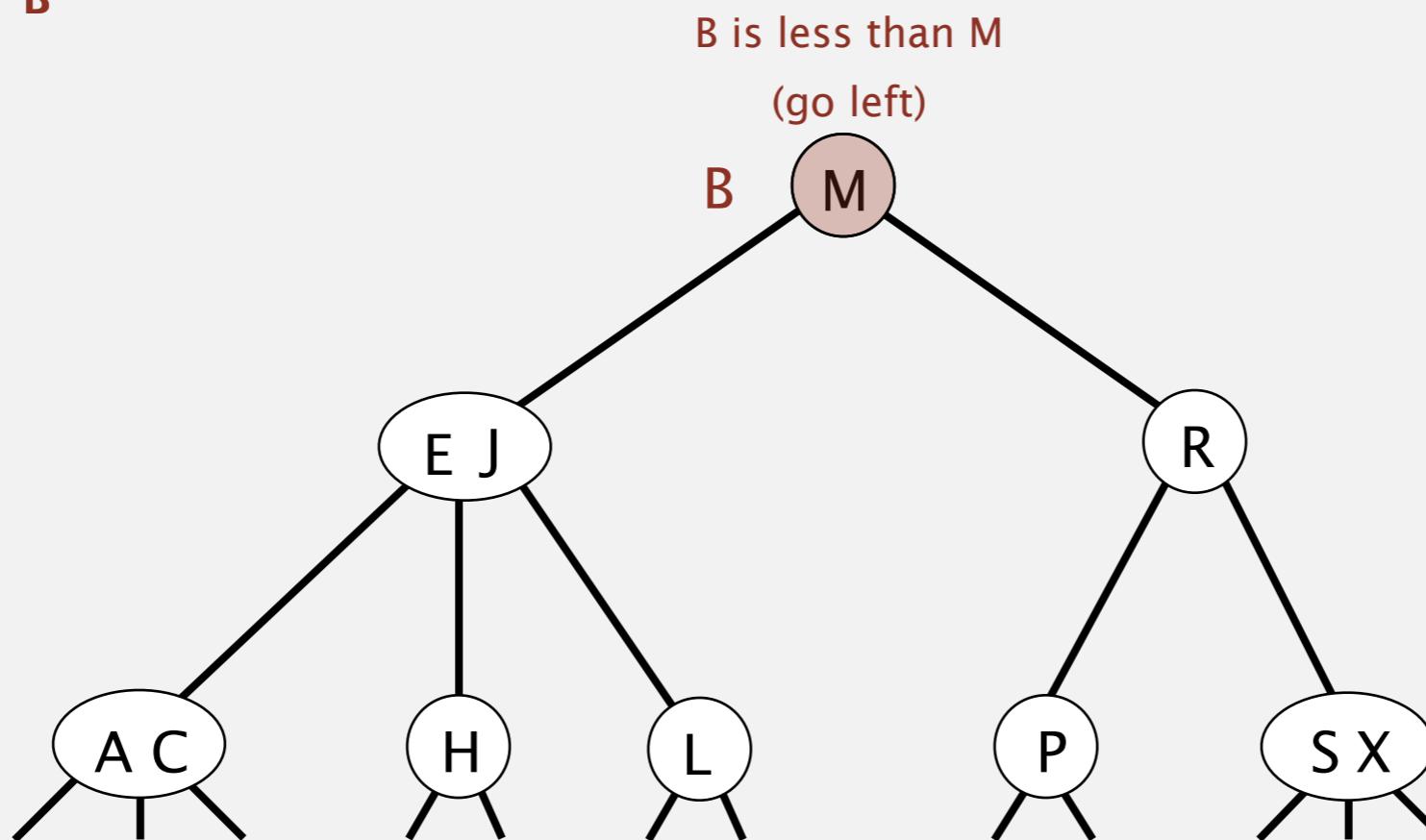
found H
(search hit)

2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for B

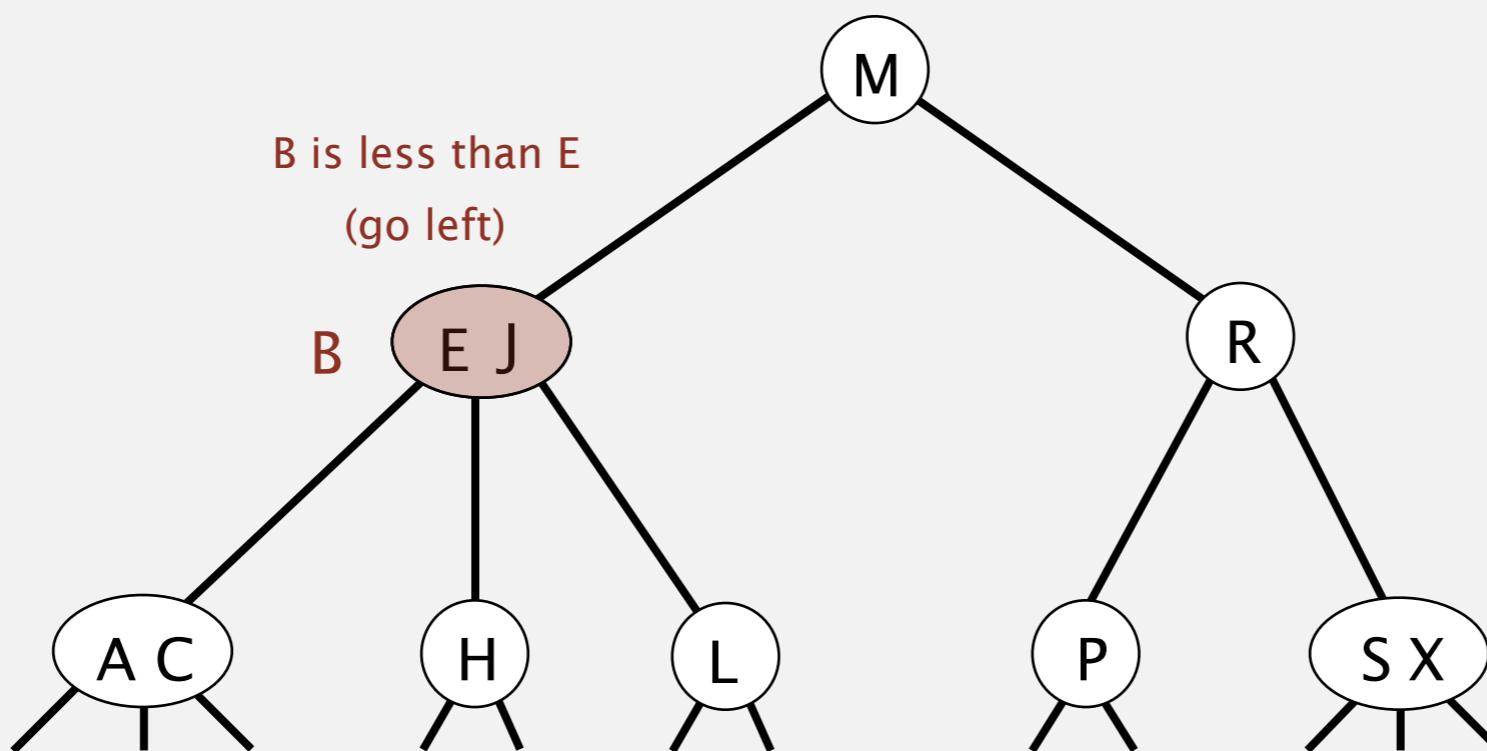


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for B

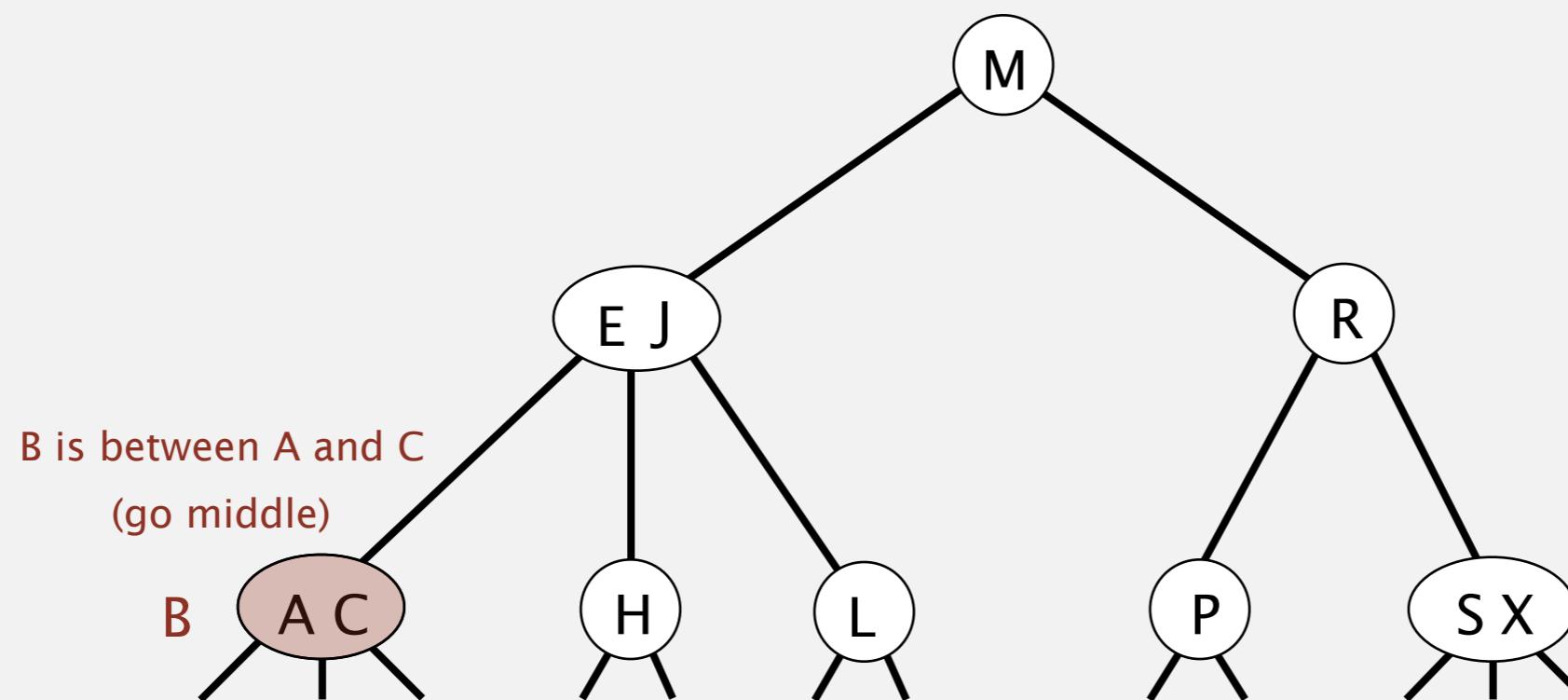


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

search for B

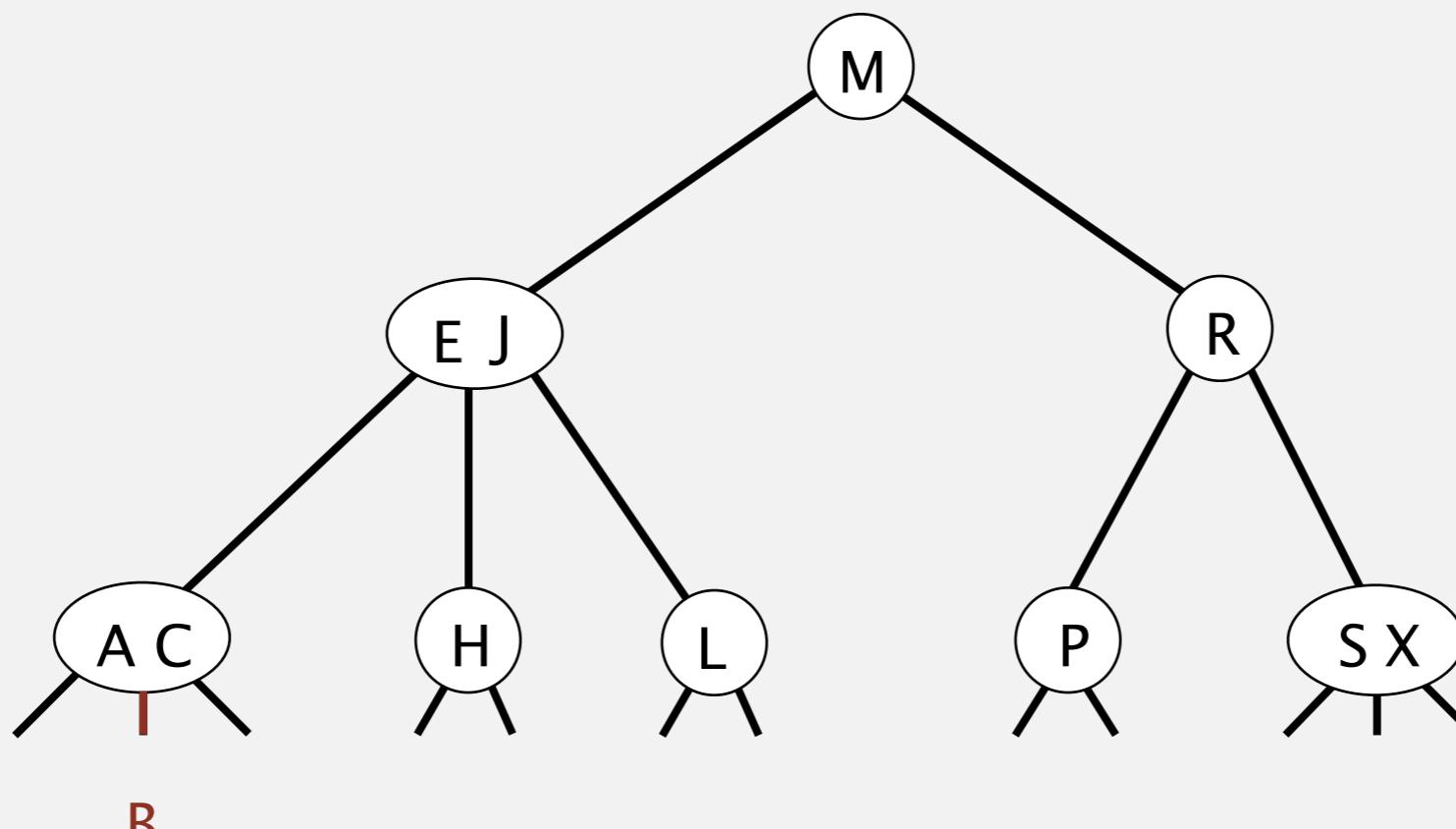


2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

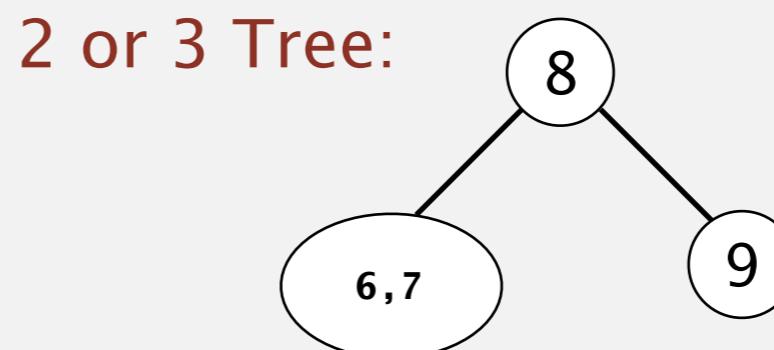
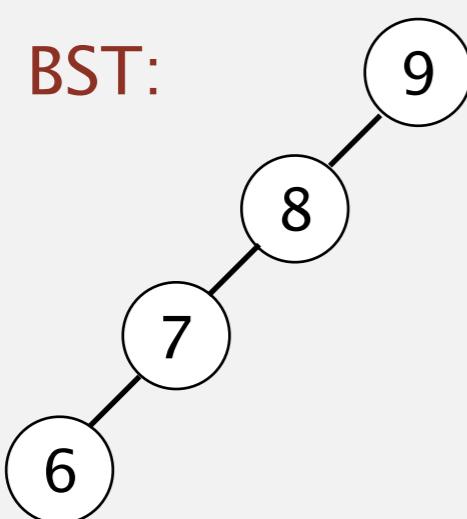
search for B



link is null
(search miss)

Insert Operation

- ▶ Problem with Binary Search Tree: when the tree grows from leaves, it is possible to always insert to same branch. (worst-case)
- ▶ Instead of growing the tree from bottom, try to grow upwards.
 - ▶ If there is space in a leaf, simply insert it
 - ▶ Otherwise push nodes from bottom to top, if done recursively the tree will be balanced as it grows (increasing the height by introducing a new root)
- ▶ If we keep on inserting to same branch;

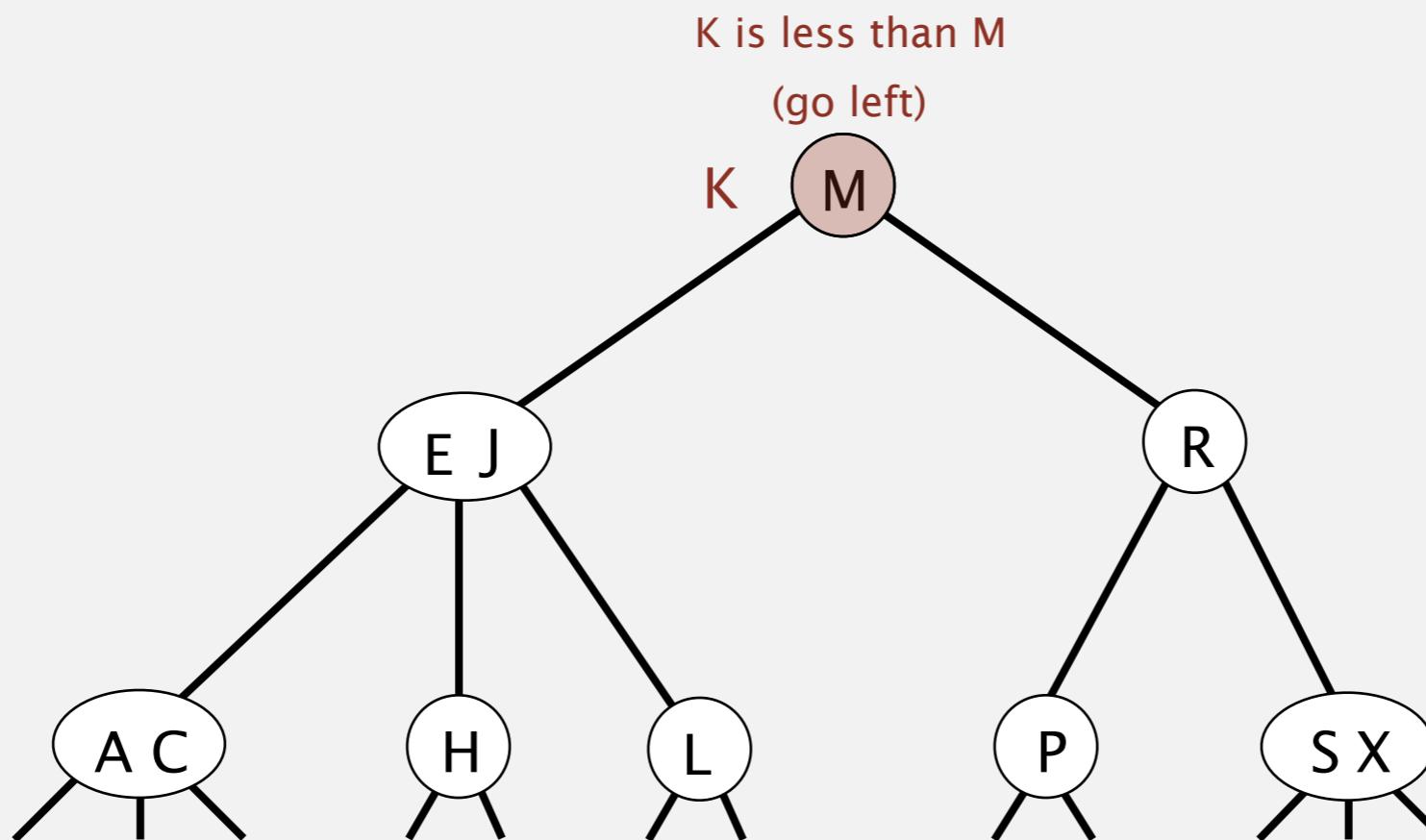


2-3 tree demo

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K

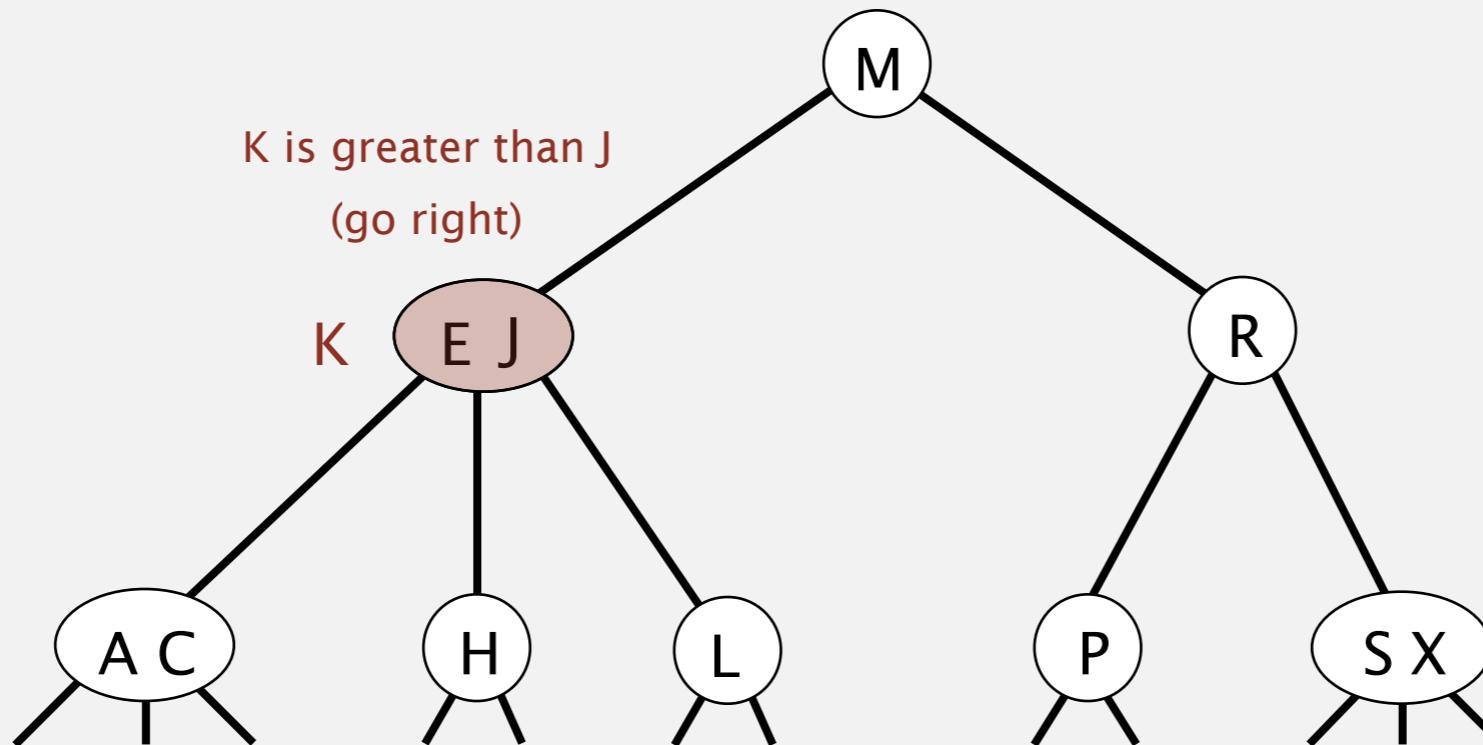


2-3 tree demo

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K

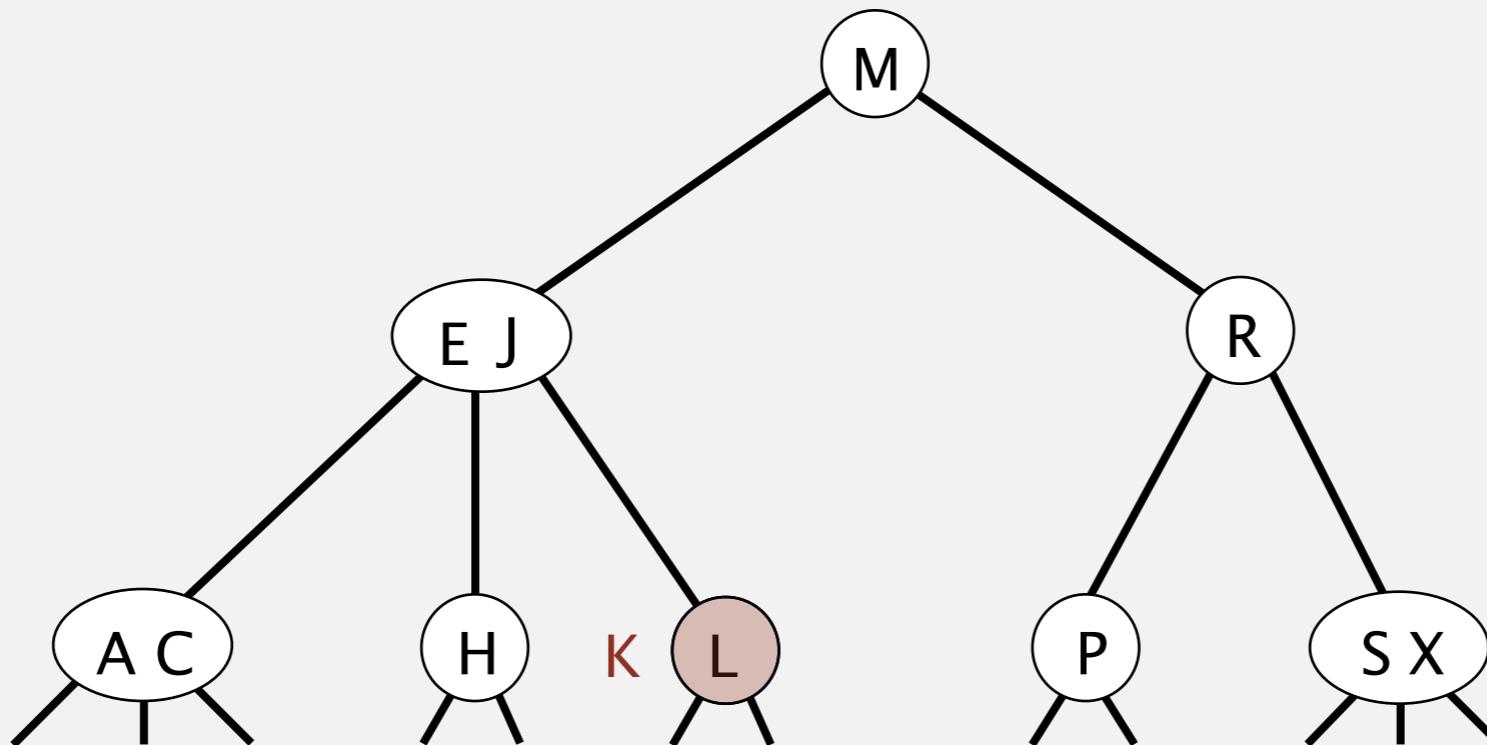


2-3 tree demo

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K



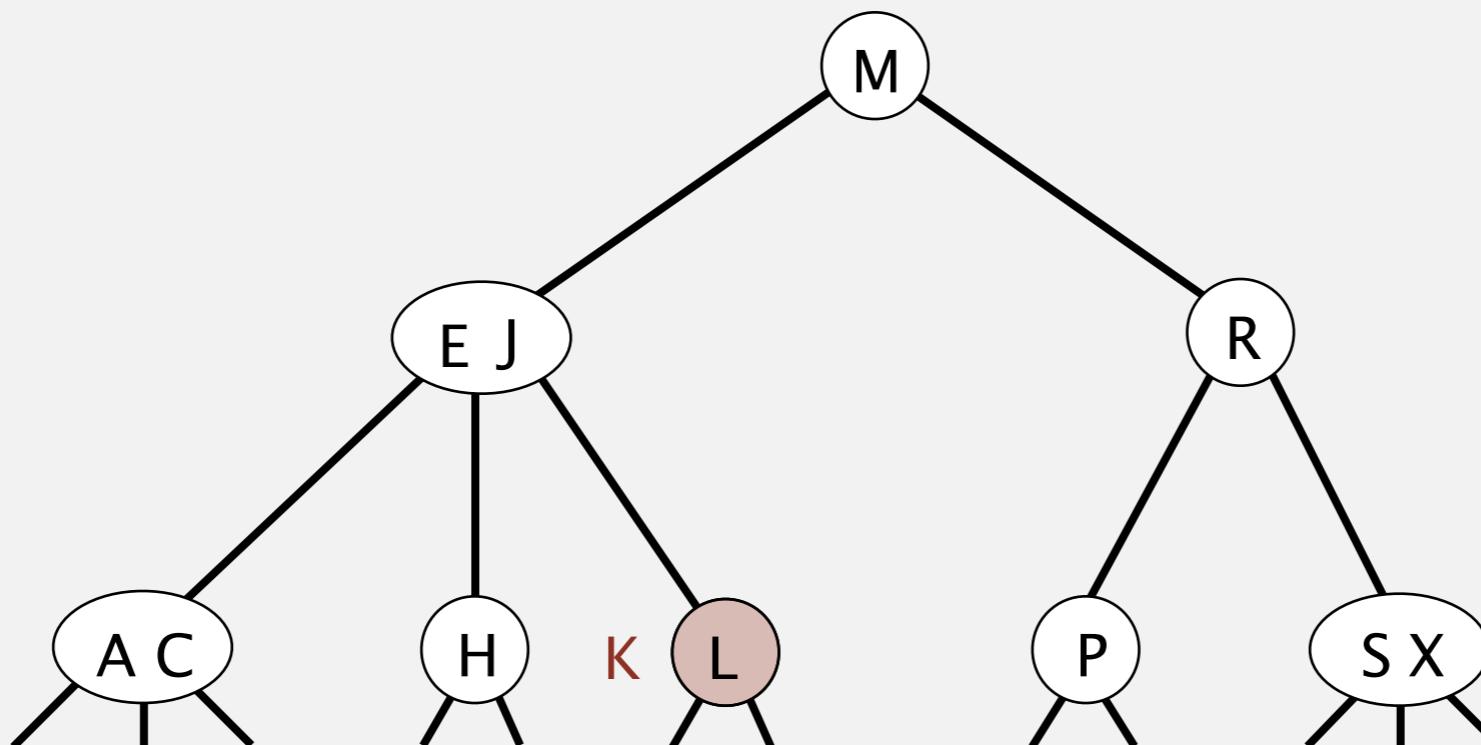
search ends here

2-3 tree demo

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K



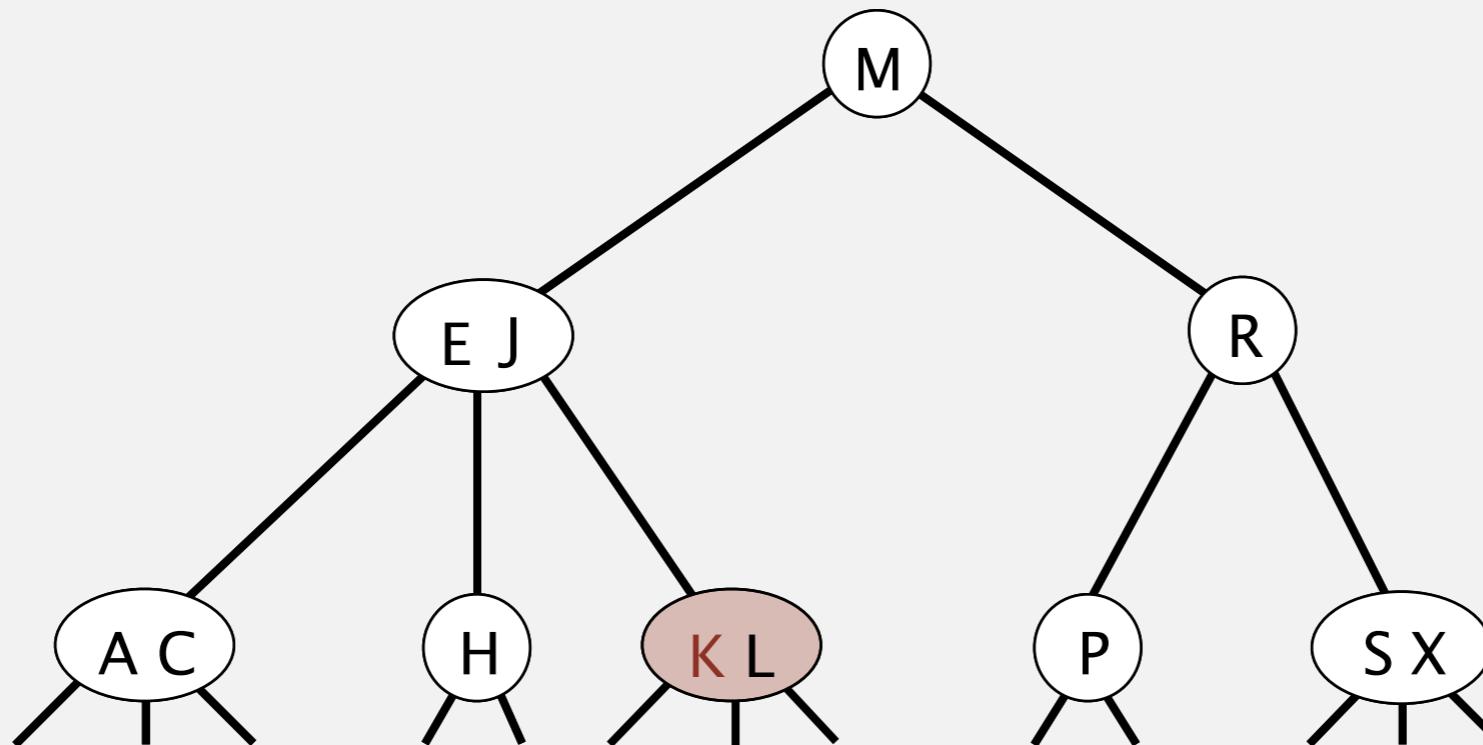
replace 2-node with
3-node containing K

2-3 tree demo

Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

insert K



2-3 tree demo

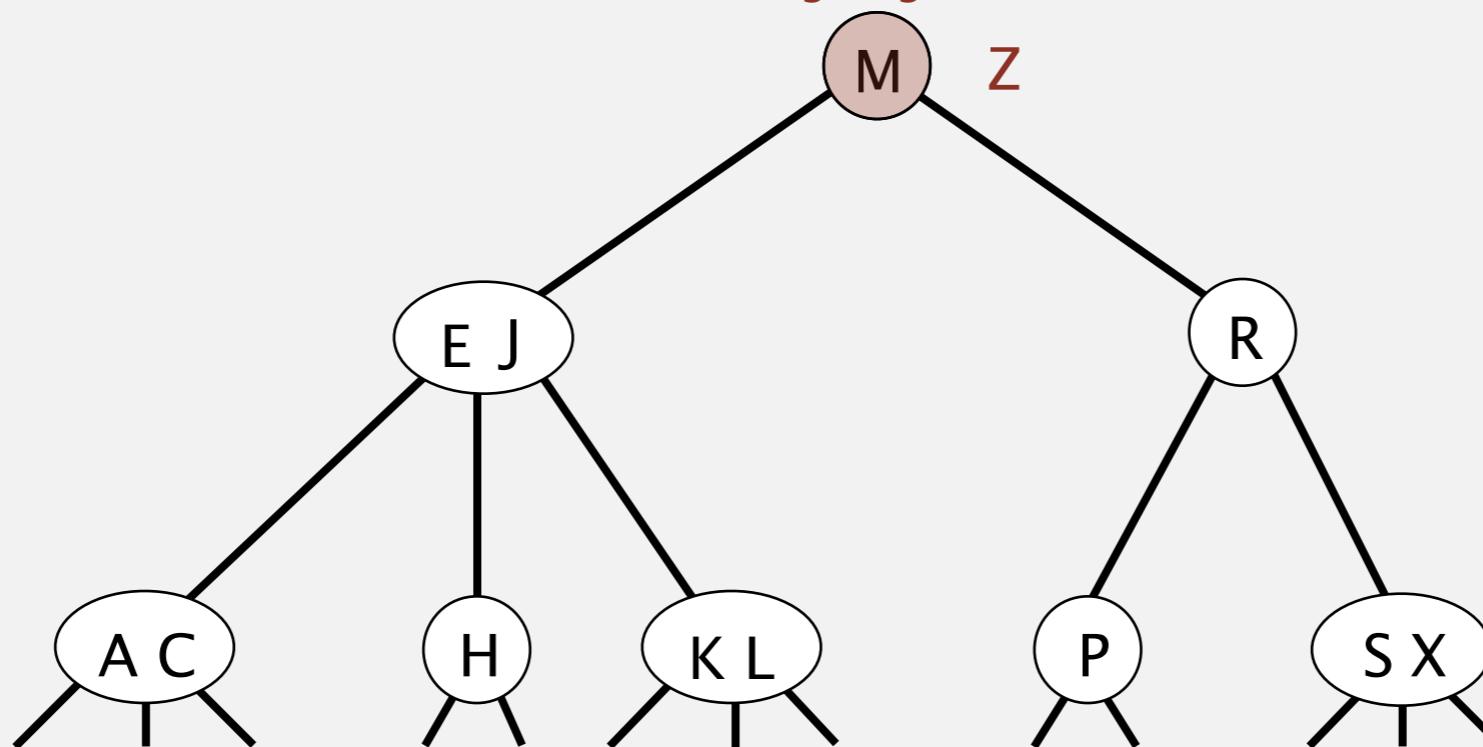
Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

Z is greater than M

(go right)

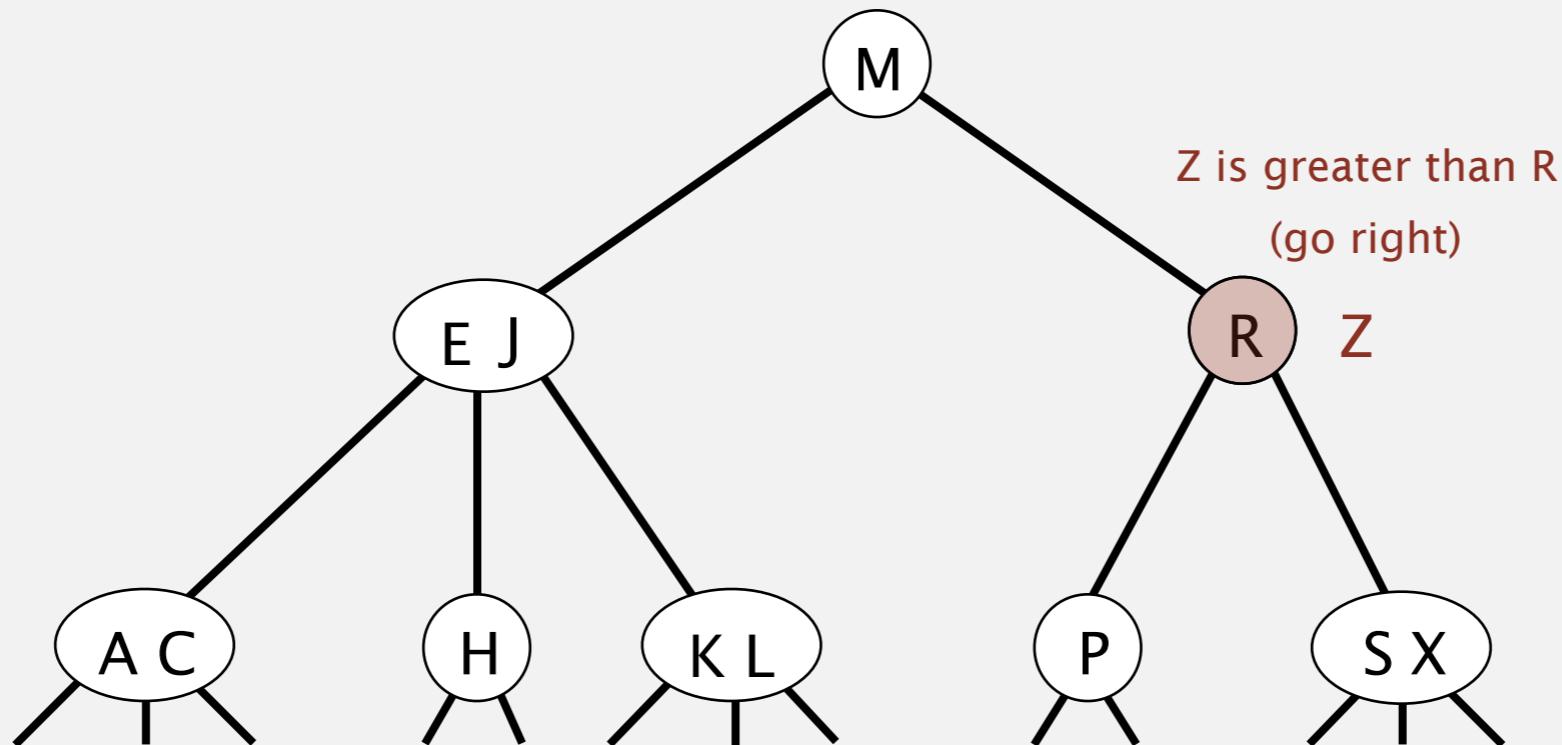


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

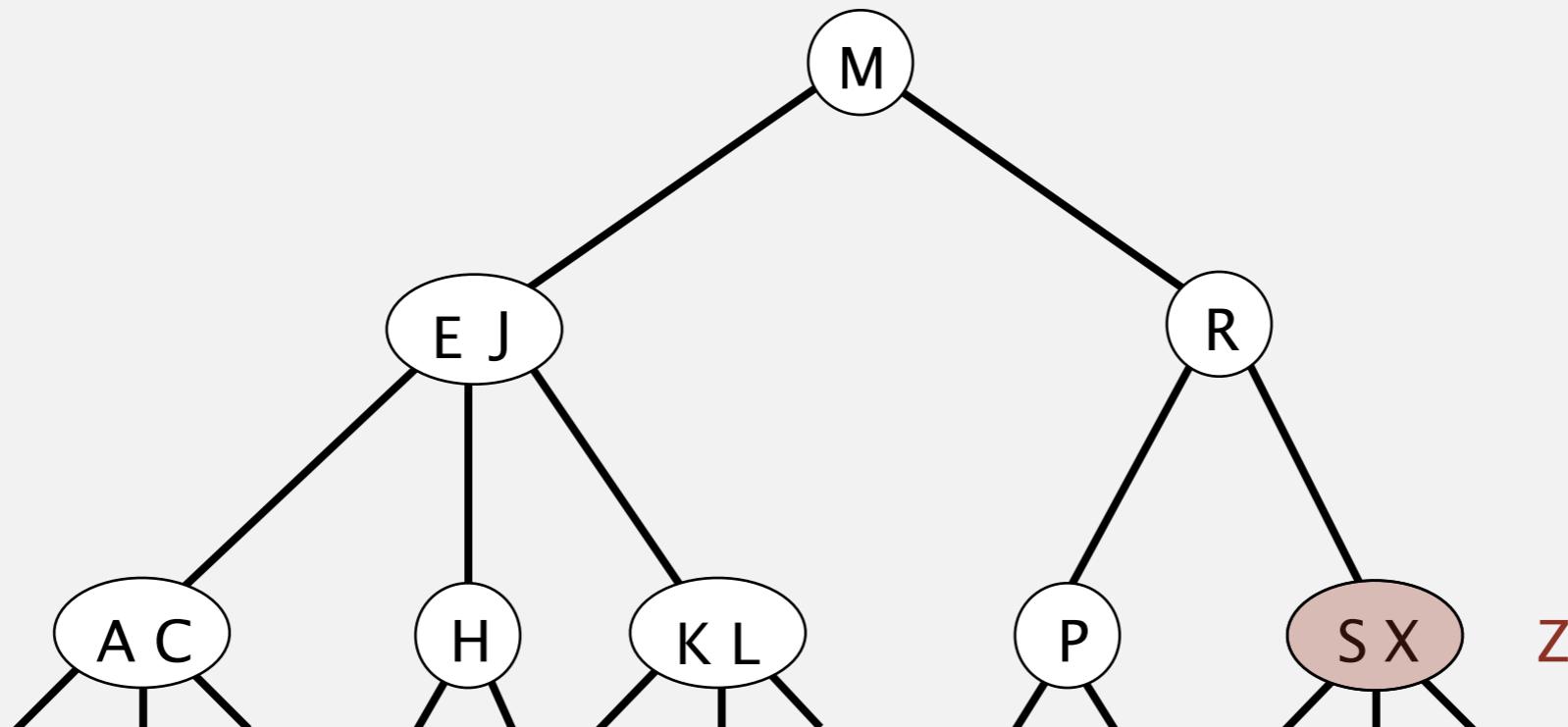


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z



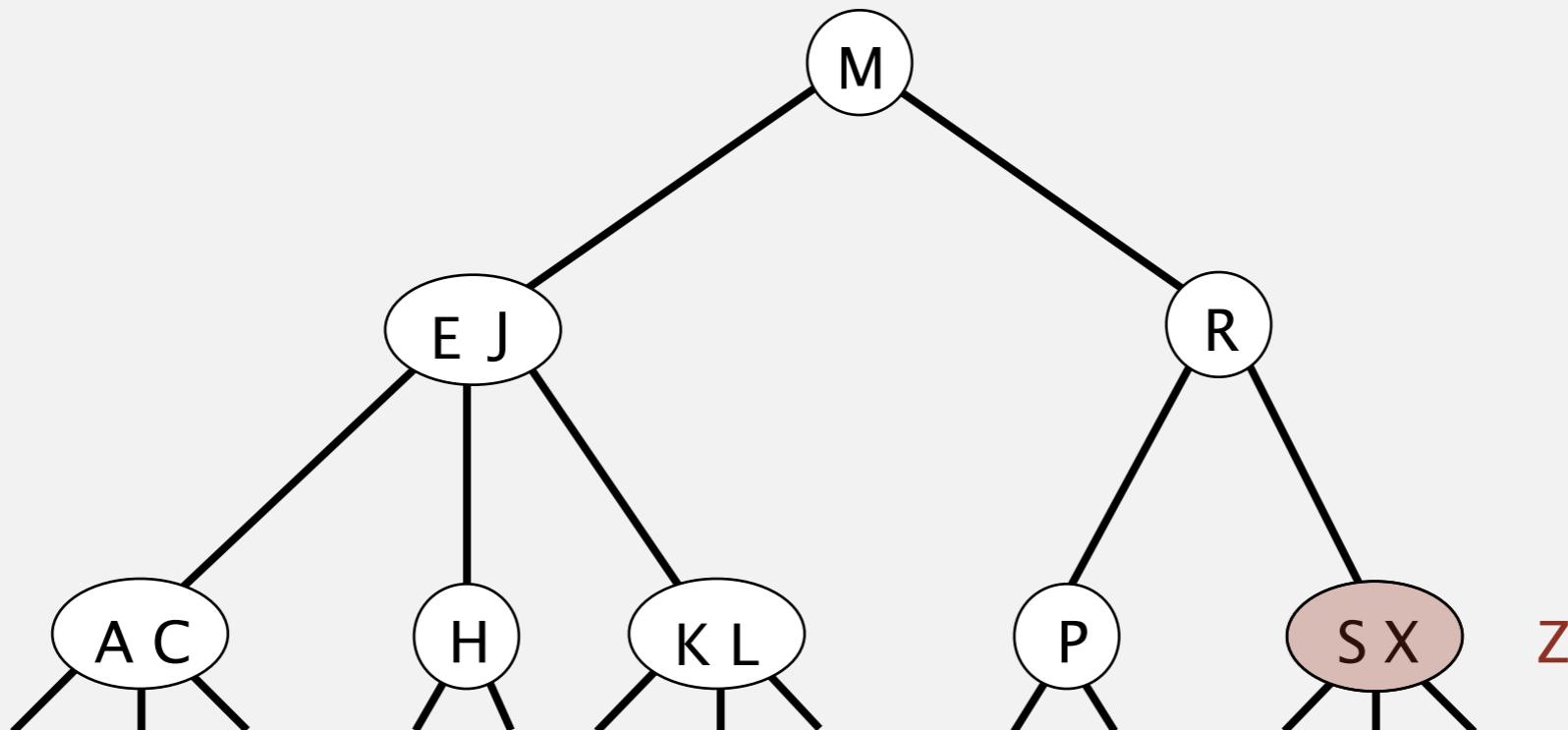
search ends here

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z



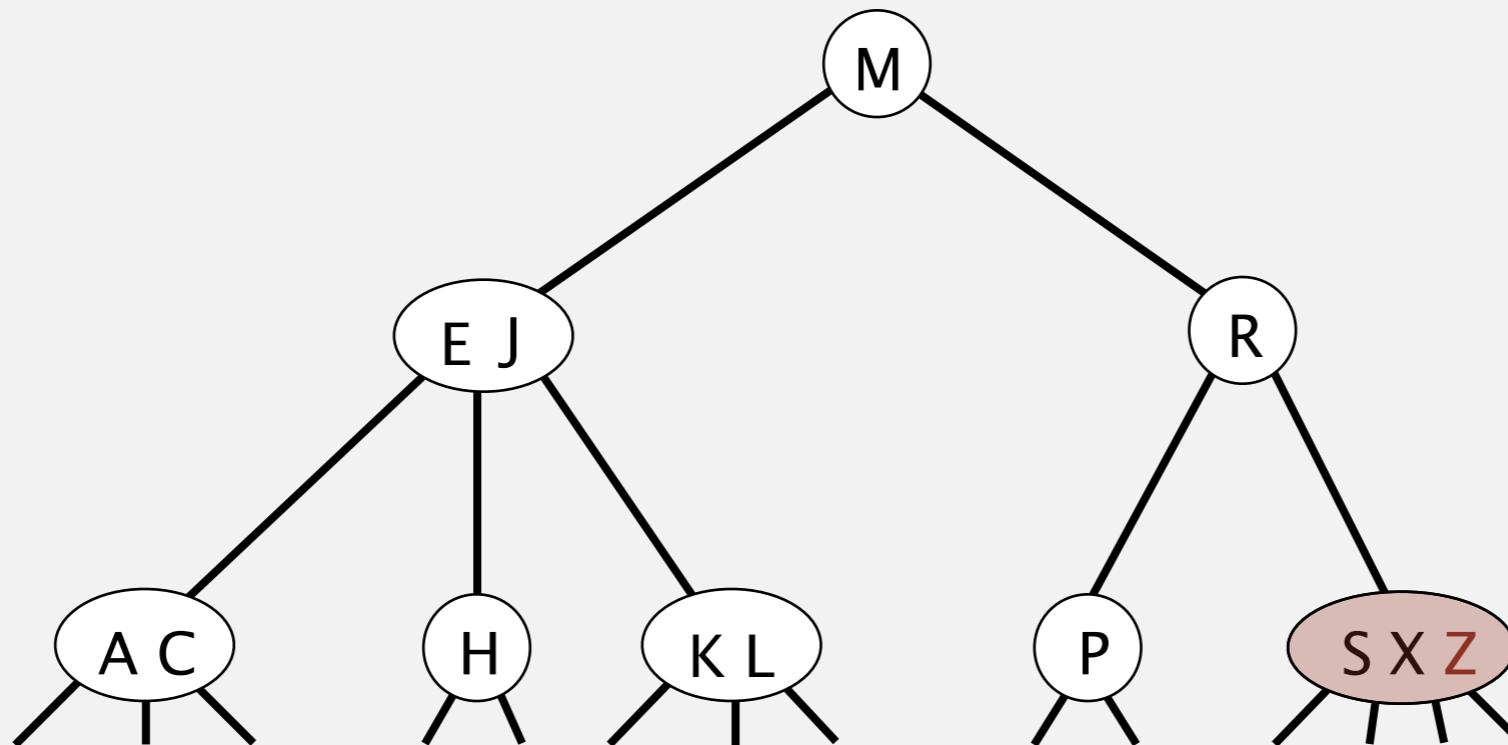
replace 3-node with
temporary 4-node containing Z

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

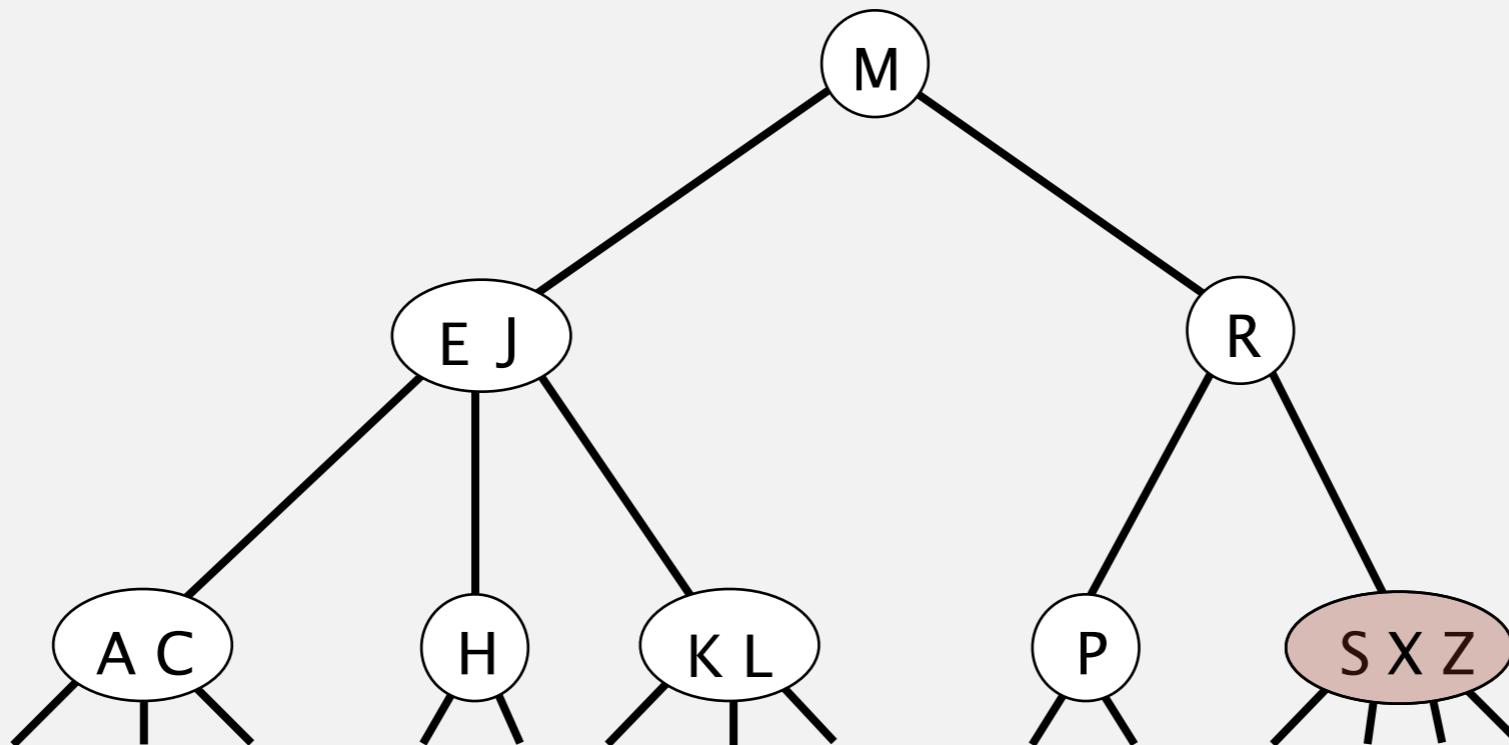


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z



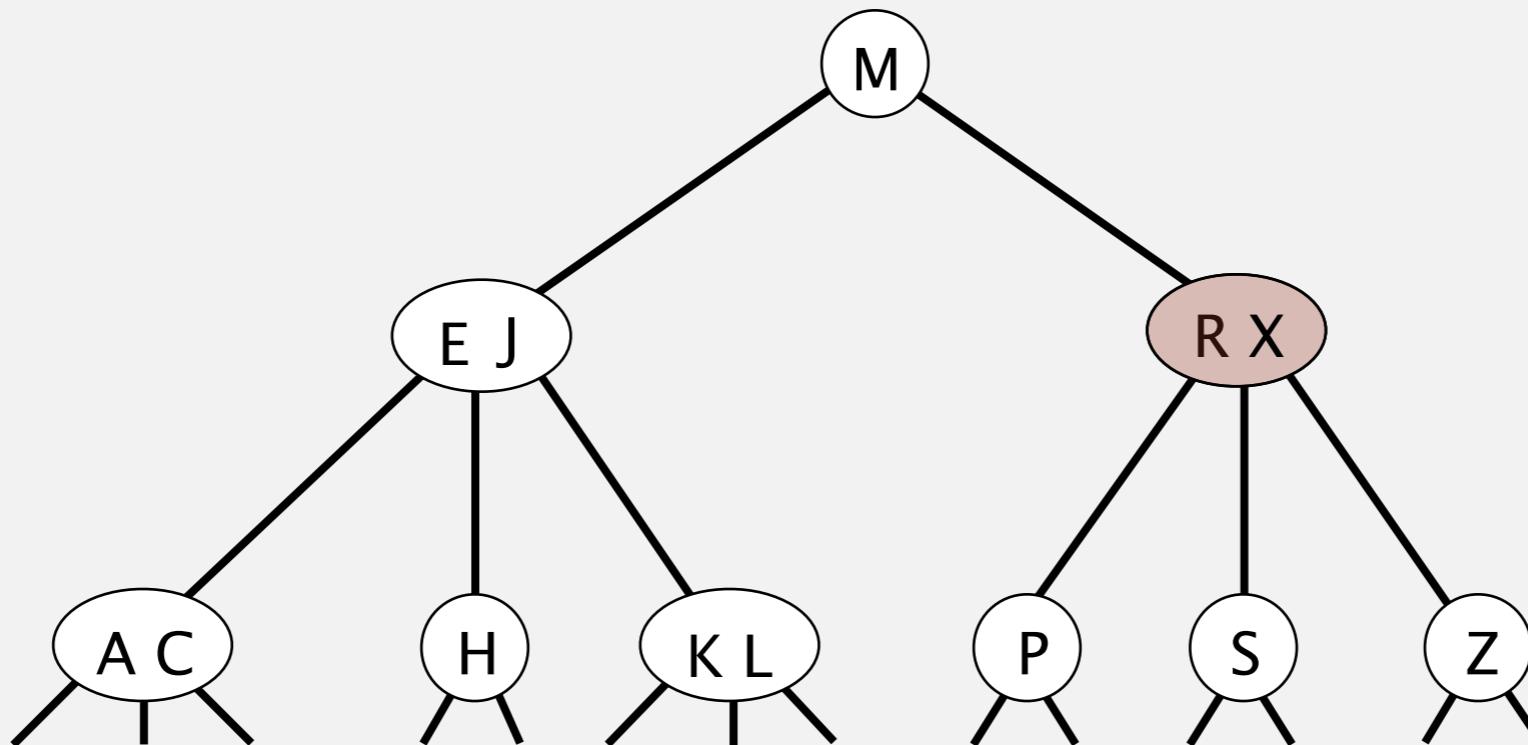
split 4-node into two 2-nodes
(pass middle key to parent)

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

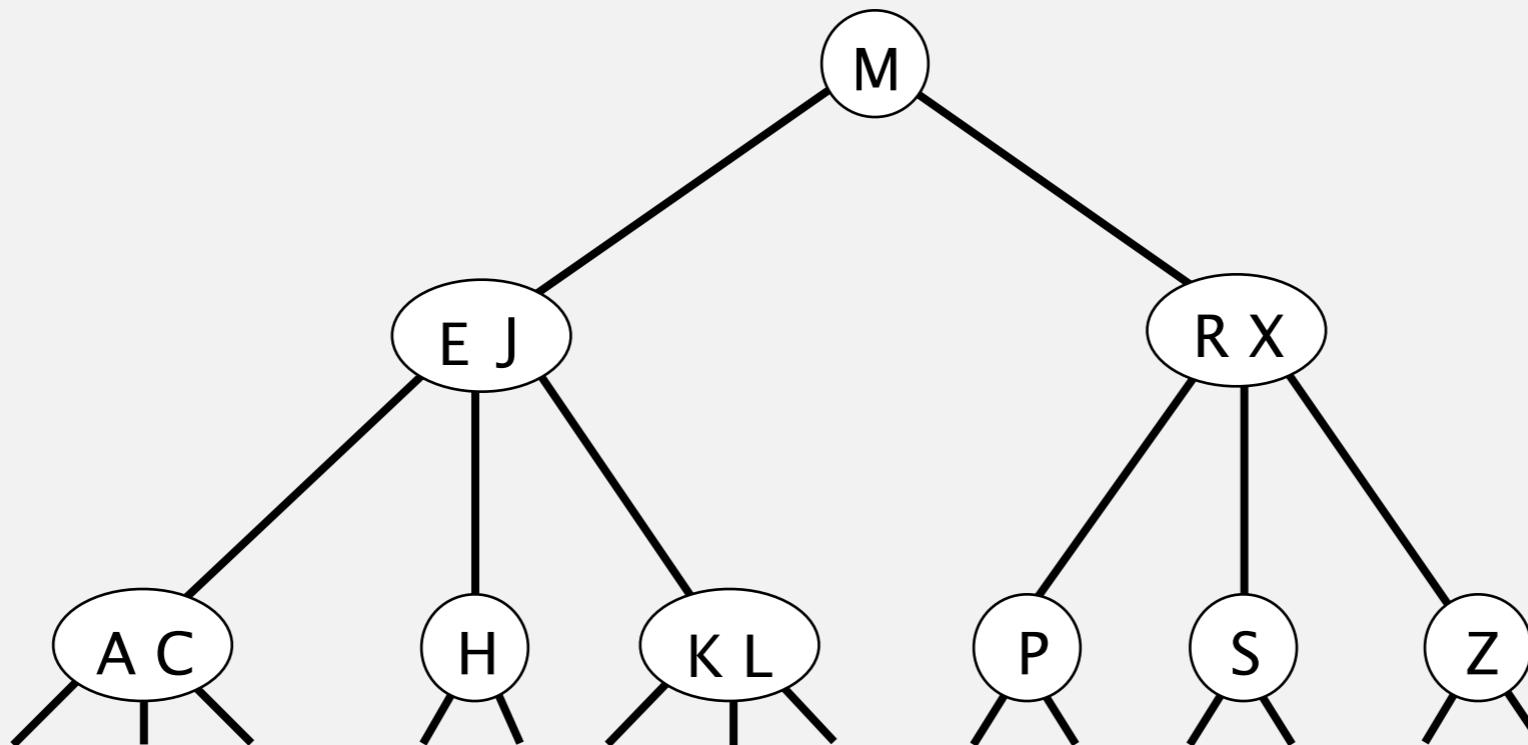


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.

insert Z

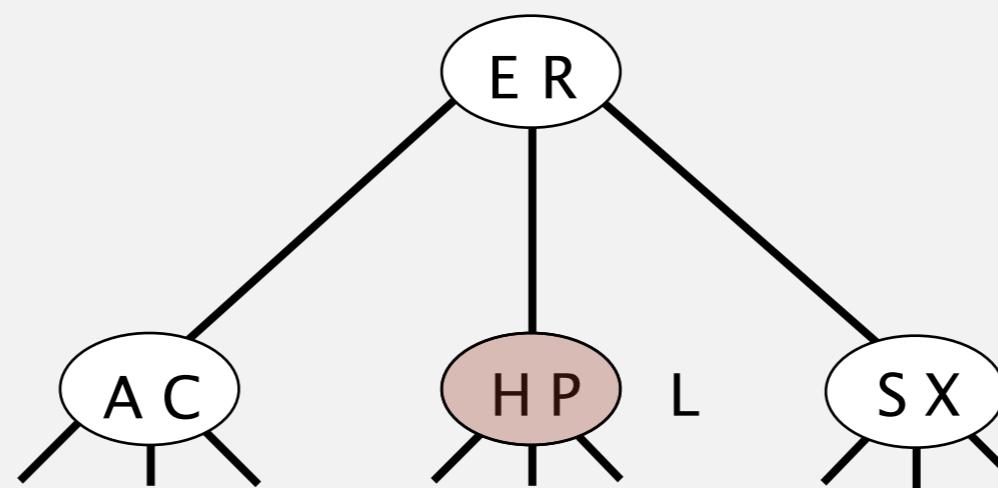


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert L



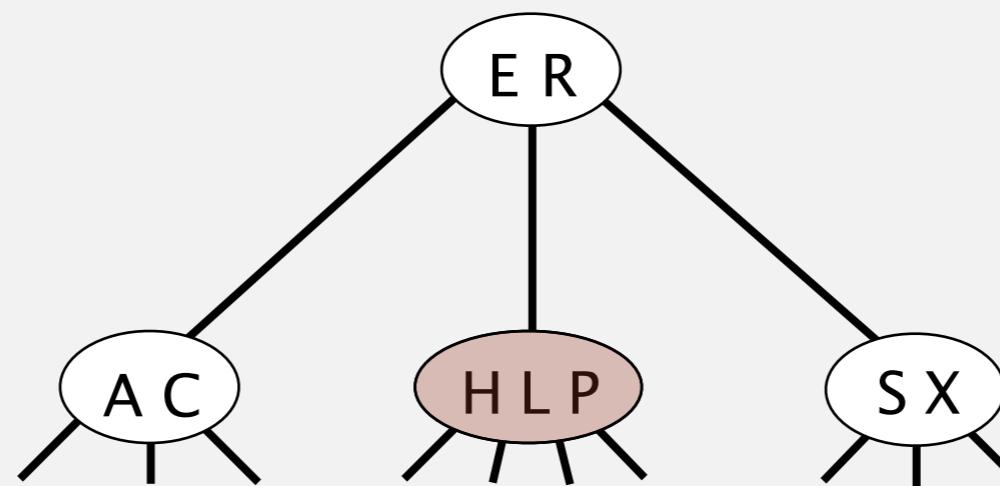
convert 3-node into 4-node

2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert L

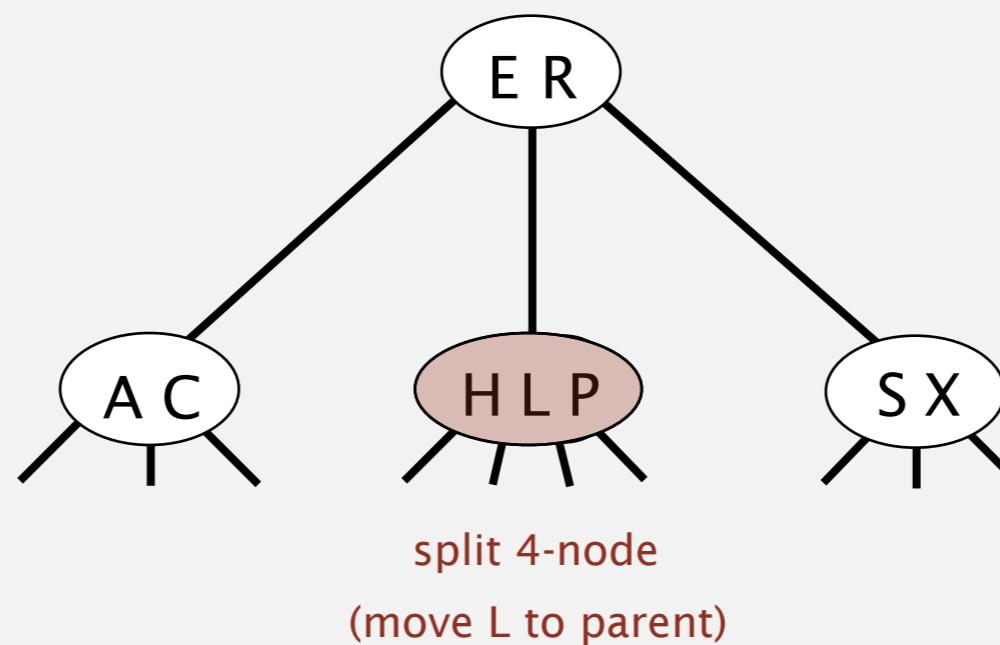


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert L

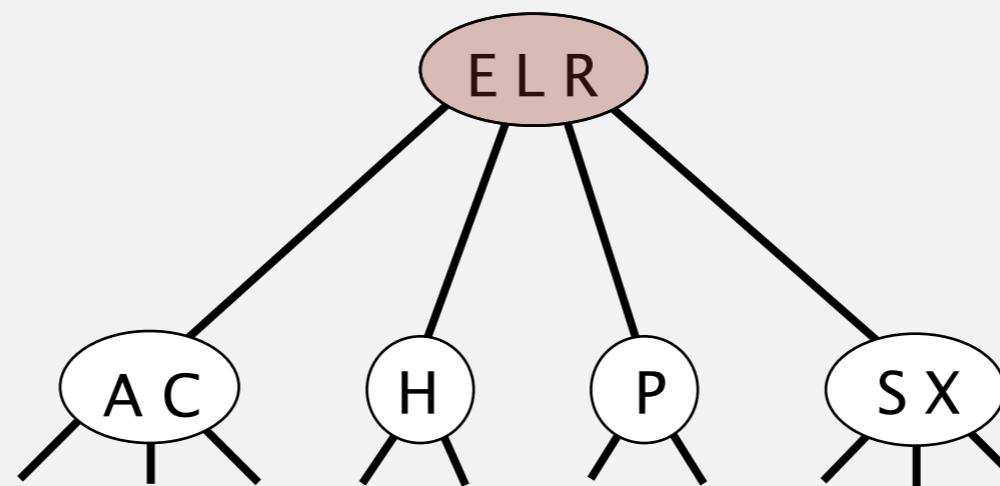


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert L

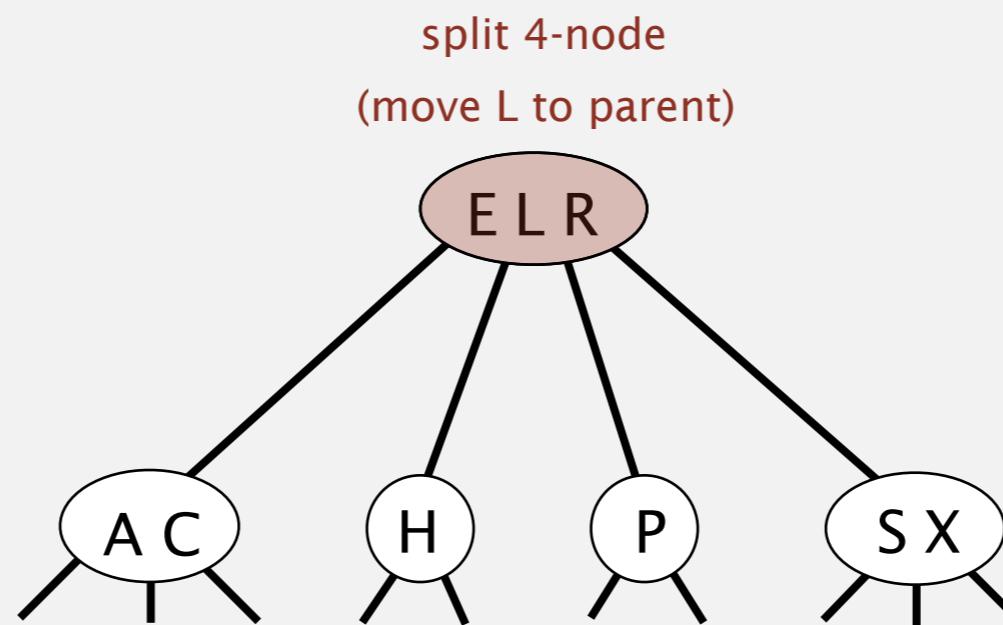


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert L



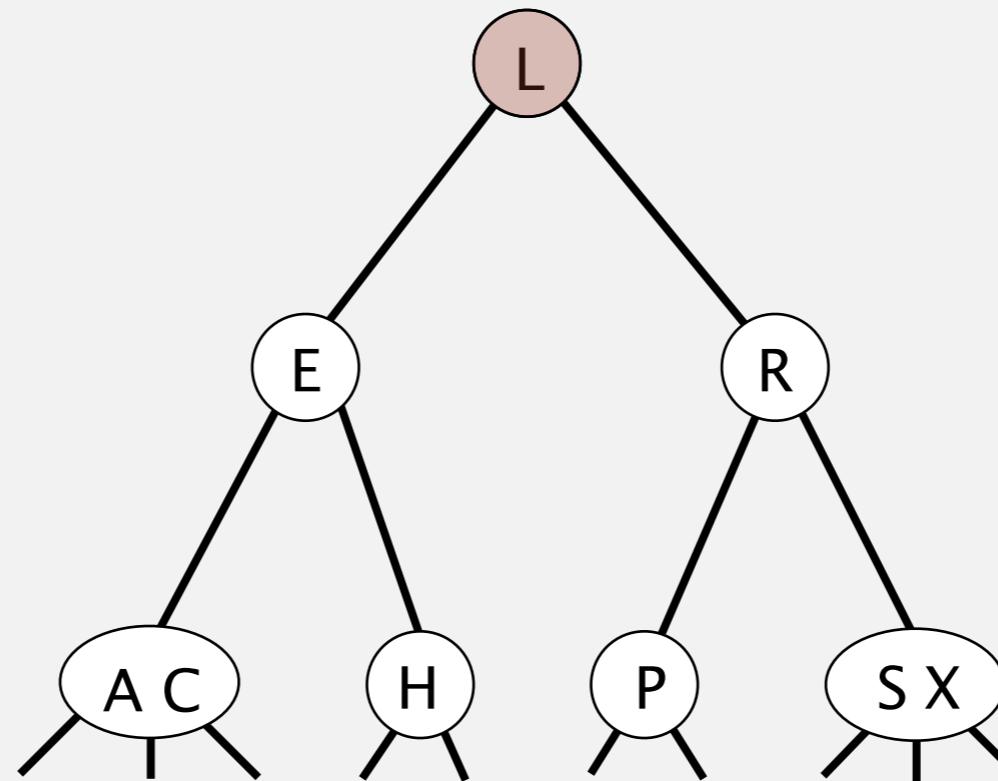
2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

height of tree increases by 1

insert L

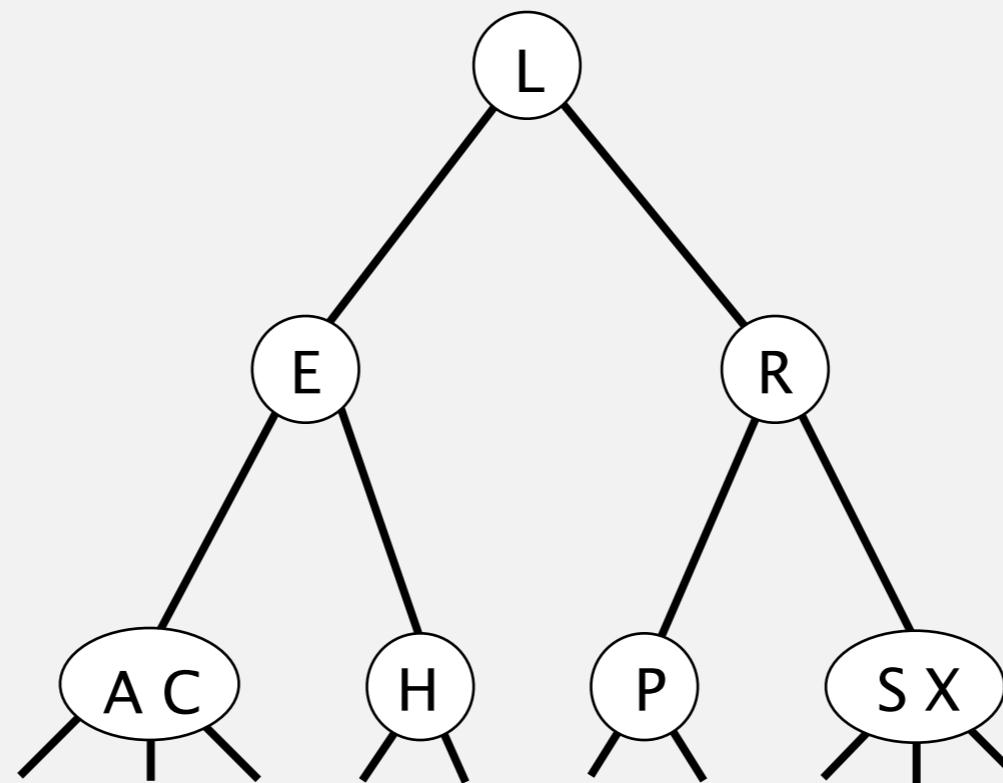


2-3 tree demo

Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

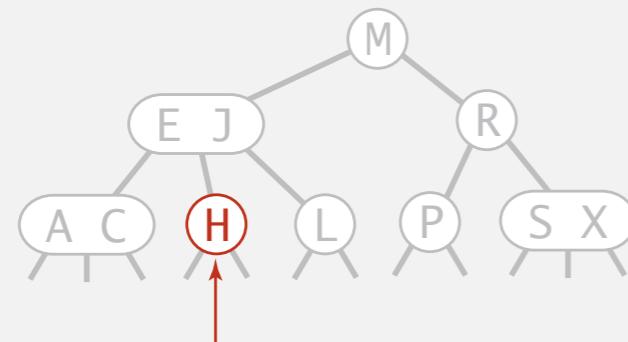
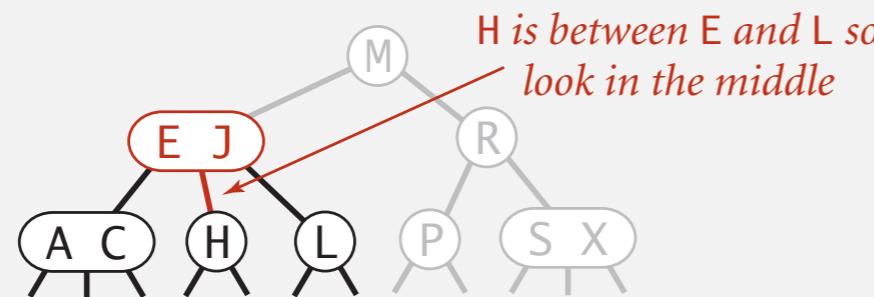
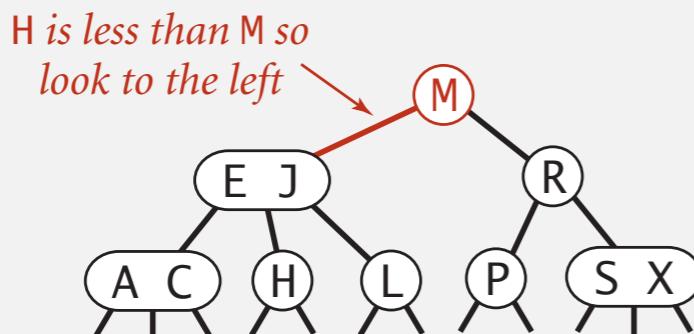
insert L



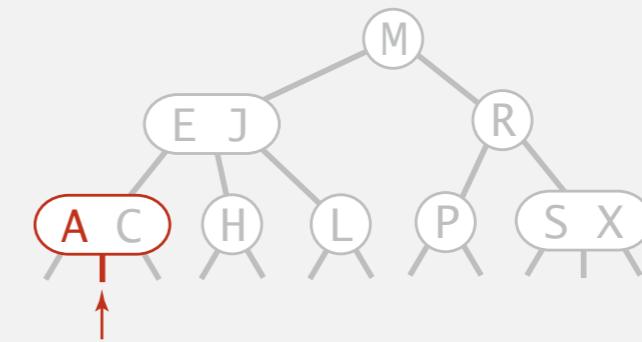
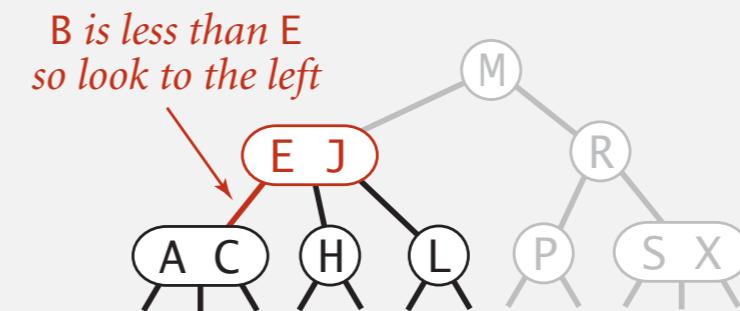
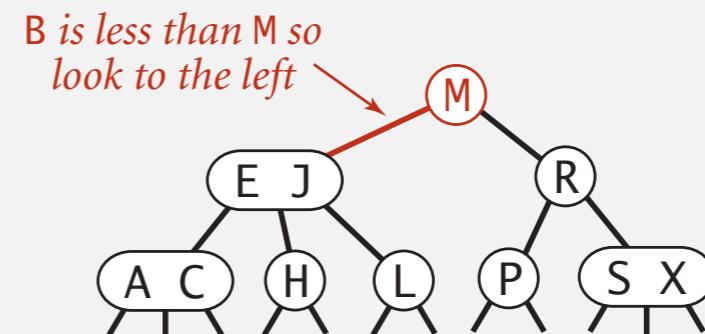
Search in a 2-3 tree

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

successful search for H



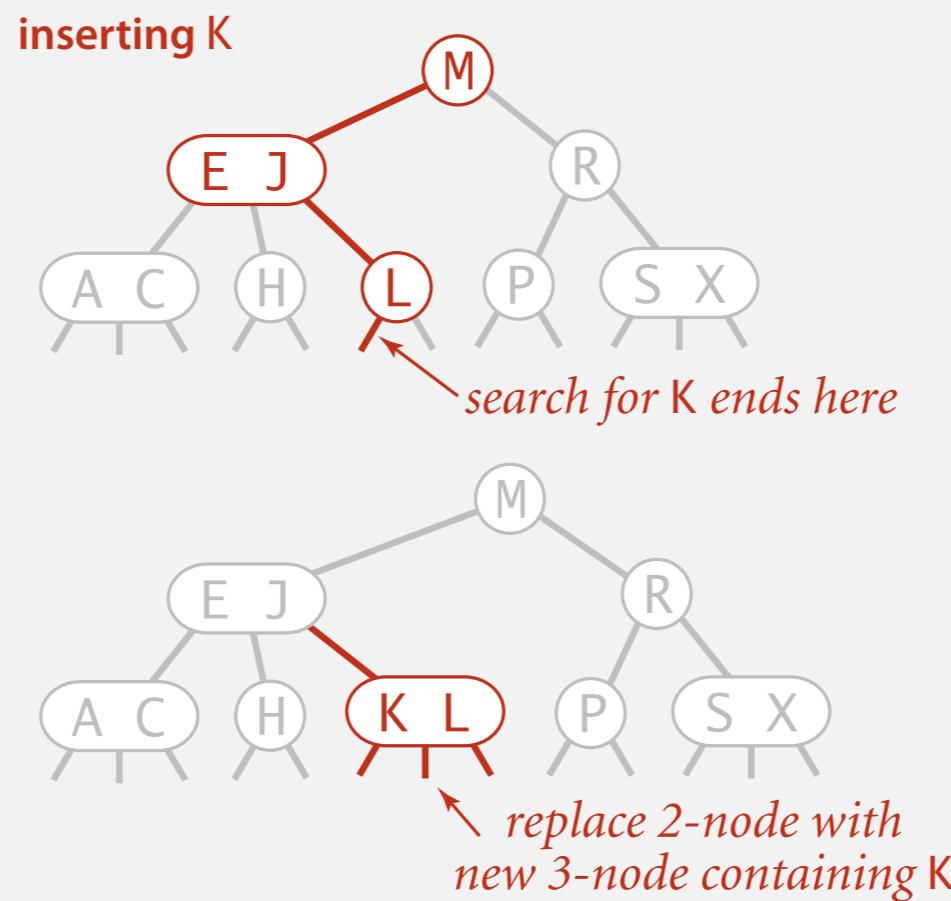
unsuccessful search for B



Insertion in a 2-3 tree

Case I. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

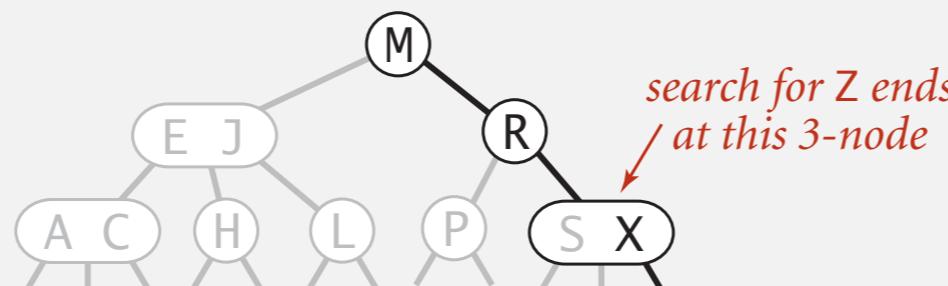


Insertion in a 2-3 tree

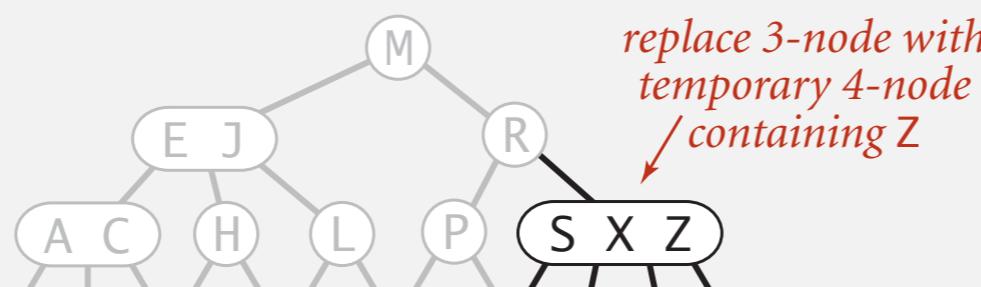
Case 2. Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.

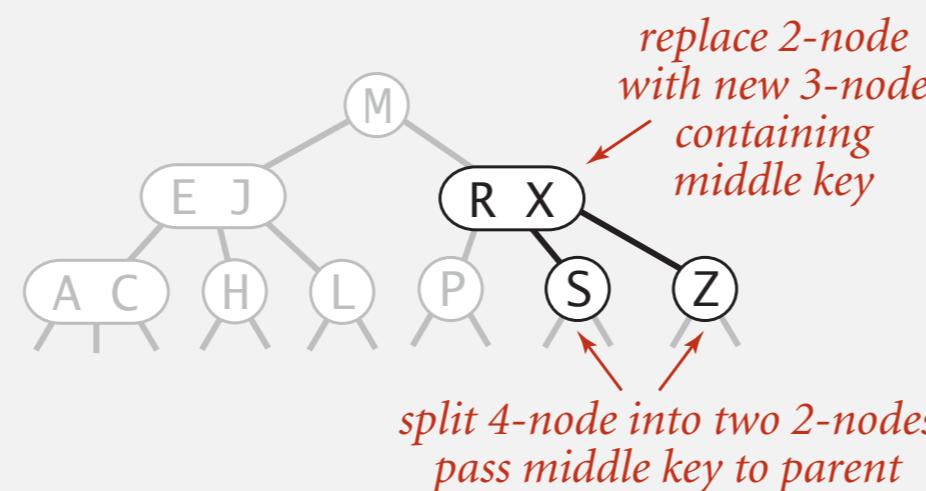
inserting Z



search for Z ends
at this 3-node



replace 3-node with
temporary 4-node
containing Z



replace 2-node
with new 3-node
containing
middle key

split 4-node into two 2-nodes
pass middle key to parent

Insertion in a 2-3 tree

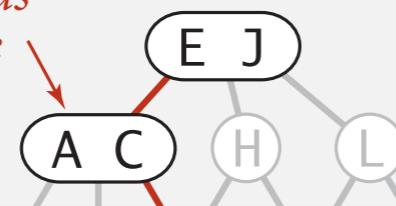
Case 2. Insert into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

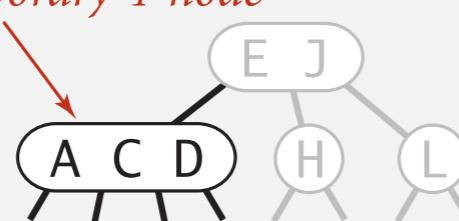
increases height by 1

inserting D

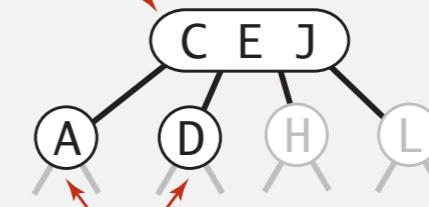
search for D ends
at this 3-node



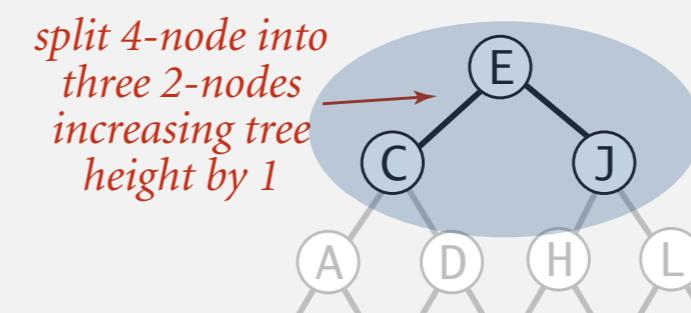
add new key D to 3-node
to make temporary 4-node



add middle key C to 3-node
to make temporary 4-node

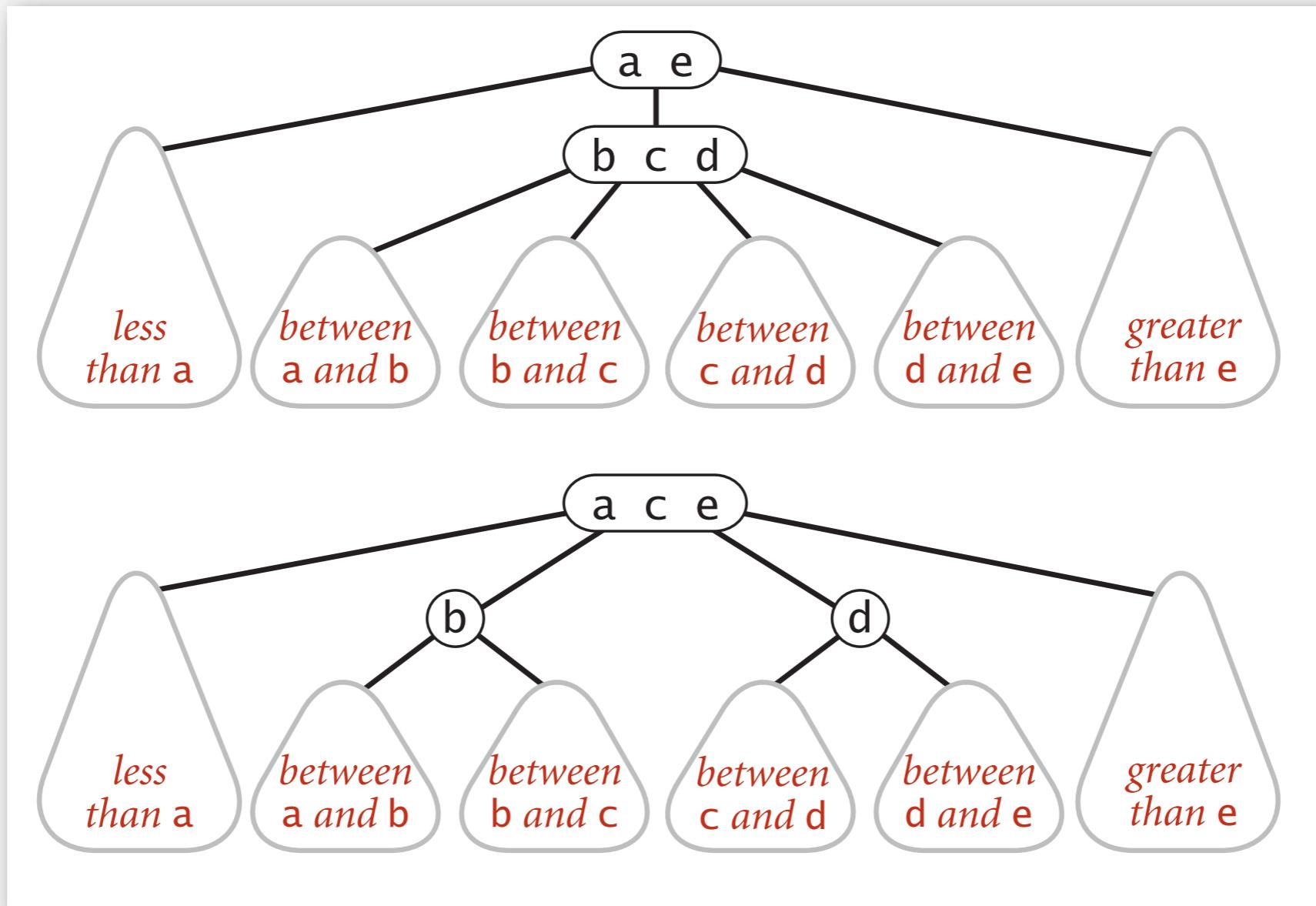


split 4-node into two 2-nodes
pass middle key to parent



Local transformations in a 2-3 tree

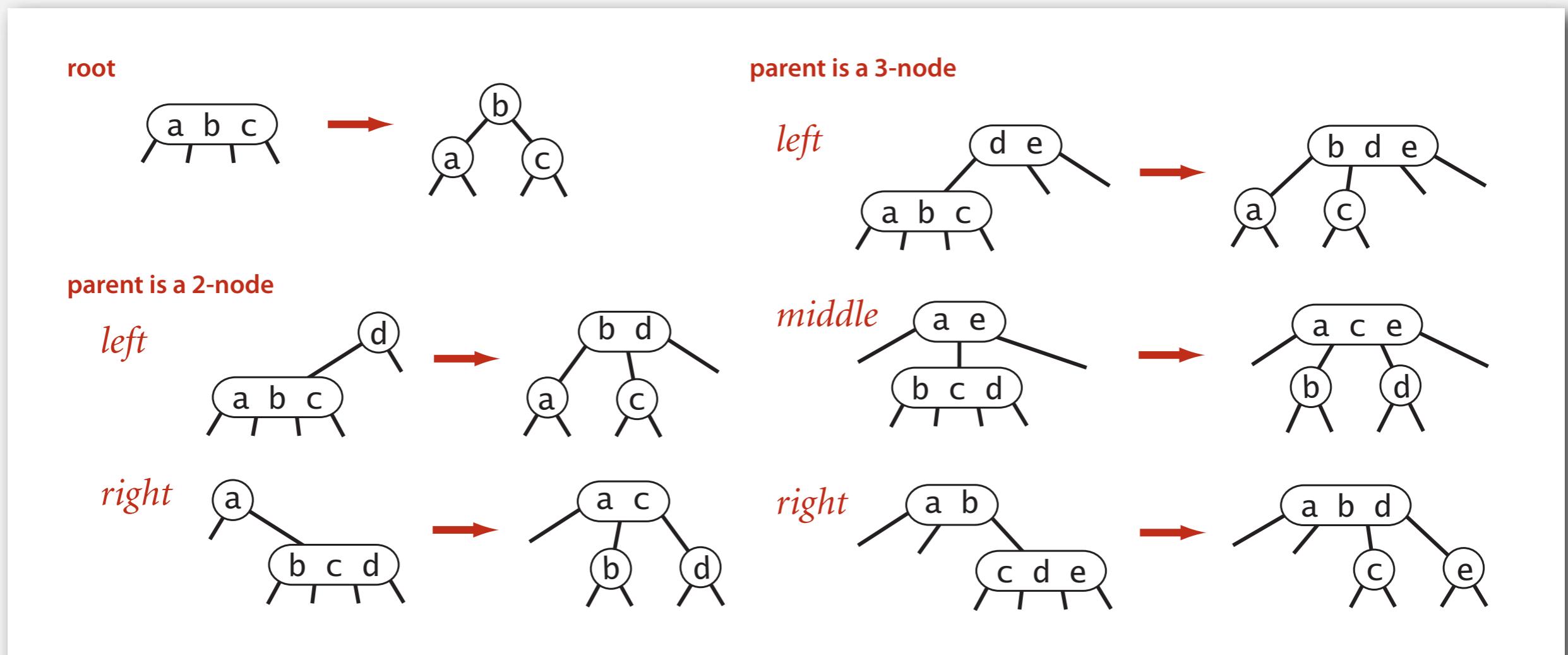
Splitting a 4-node is a **local** transformation: constant number of operations.



Global properties in a 2-3 tree

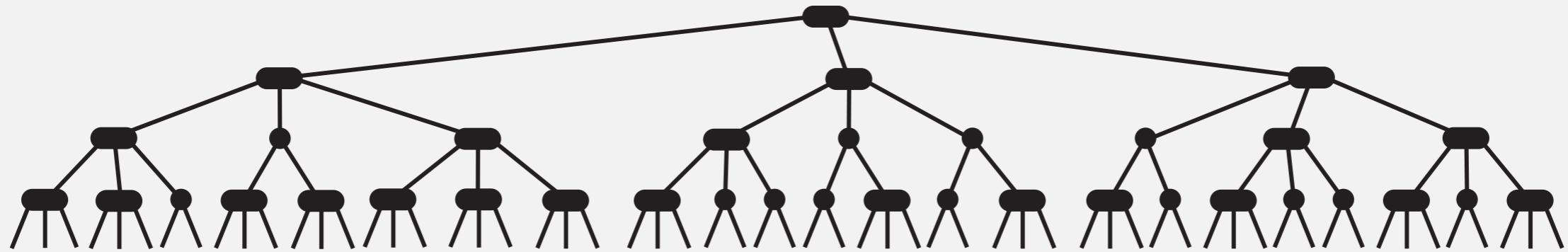
Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

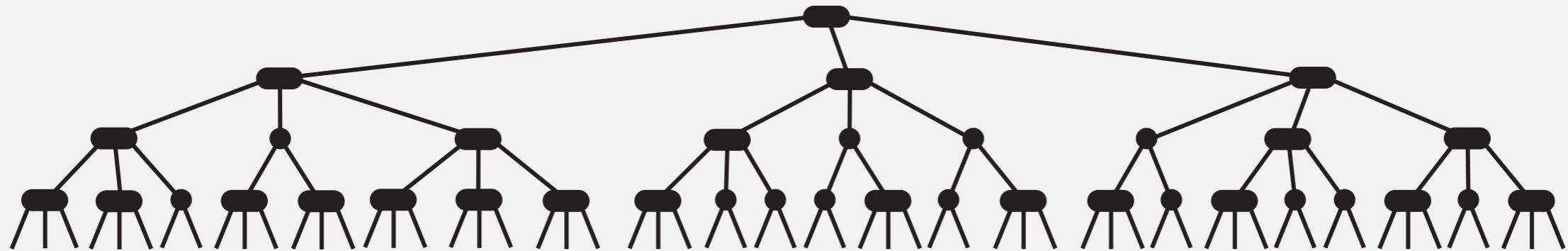


Tree height.

- Worst case:
- Best case:

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.



Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Guaranteed **logarithmic** performance for search and insert.

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$?	yes	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	yes	<code>compareTo()</code>



 constants depend upon implementation

2-3 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

Bottom line. Could do it, but there's a better way.

BALANCED SEARCH TREES

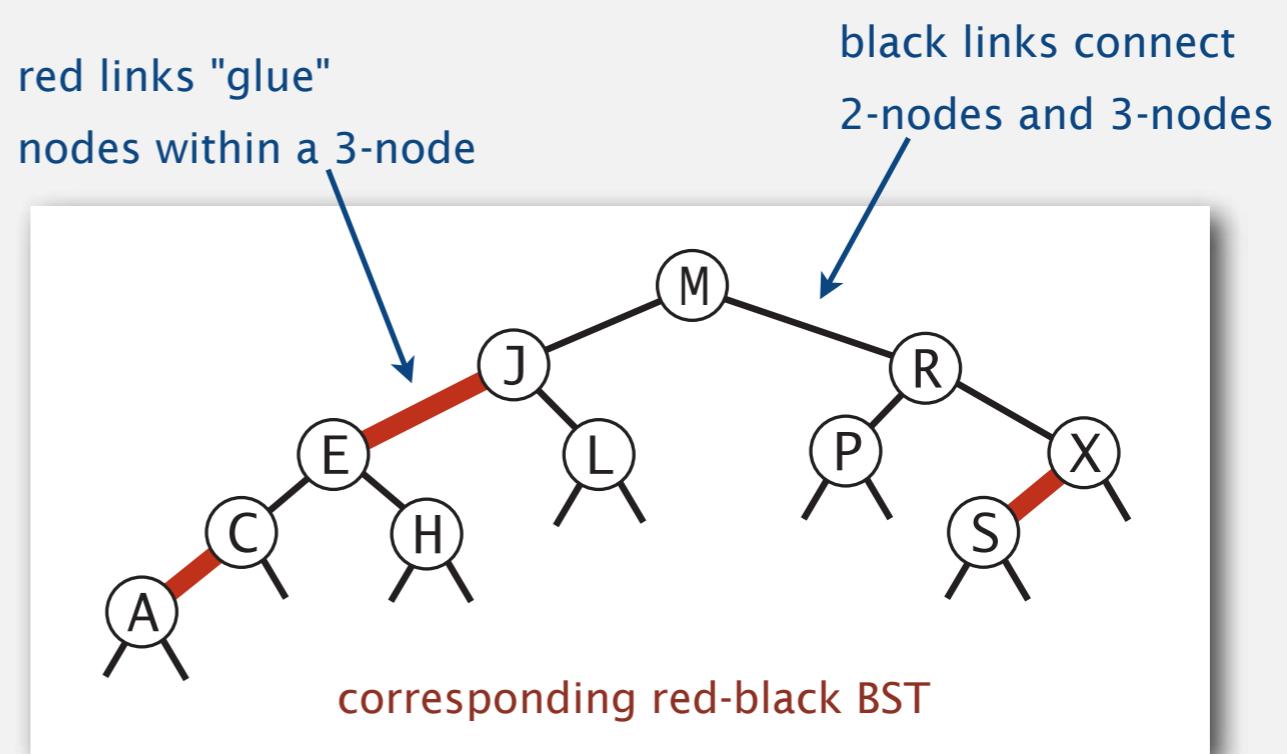
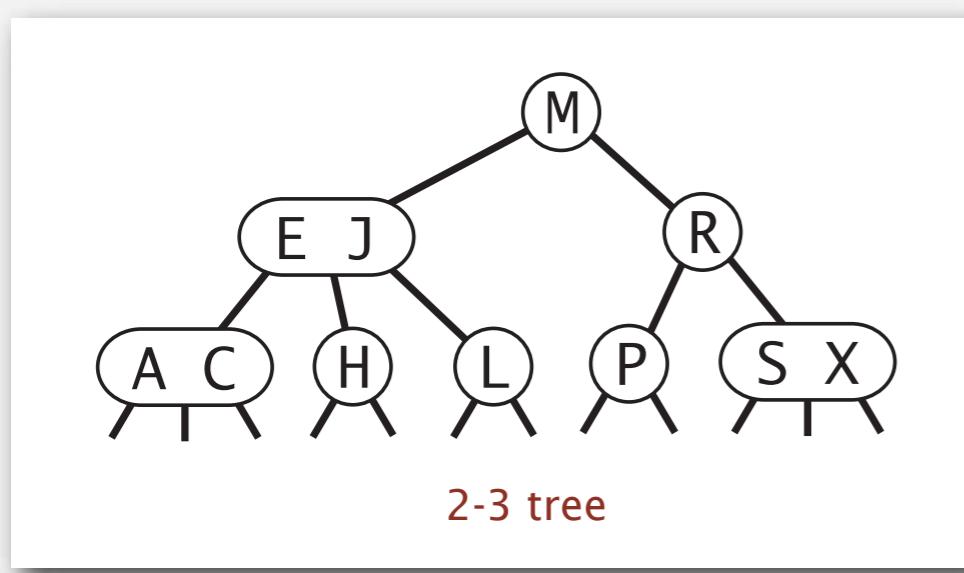
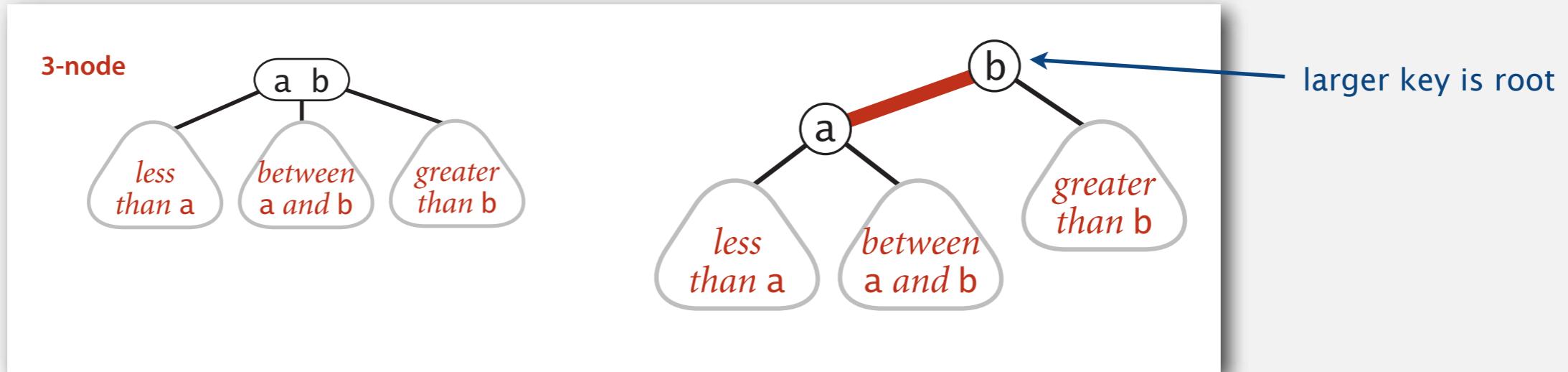
- ▶ 2-3 search trees
- ▶ Red-black BSTs
- ▶ B-trees
- ▶ Geometric applications of BSTs

Multiple Node Types

- ▶ In 2-3 Trees, the algorithm automatically balances the tree
- ▶ However, we have to keep track of two different node types, complicating the source code.
 - ▶ Nodes with one key
 - ▶ Nodes with two keys
- ▶ Instead of multiple nodes:
 - ▶ Multiple edge types; red and black
 - ▶ Rotations instead of Split

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.

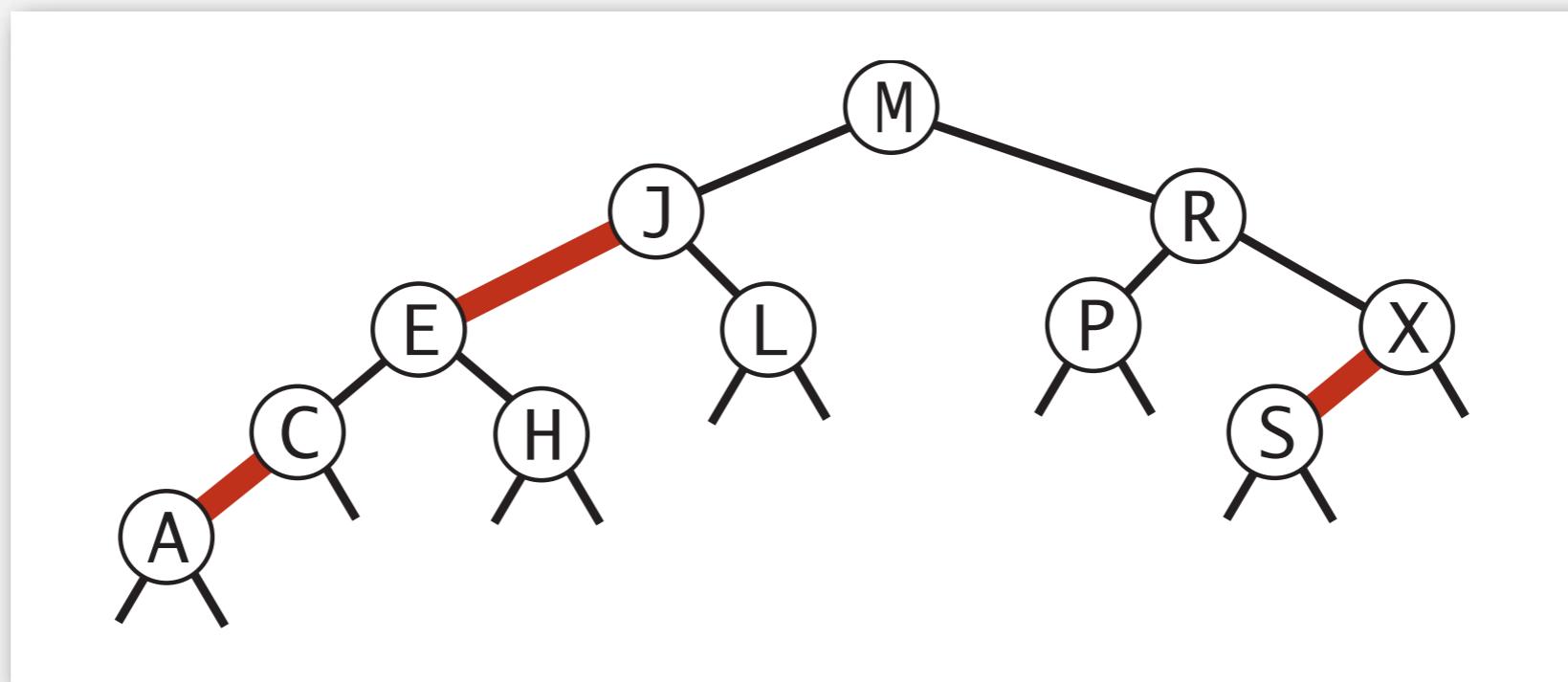


An equivalent definition

A BST such that:

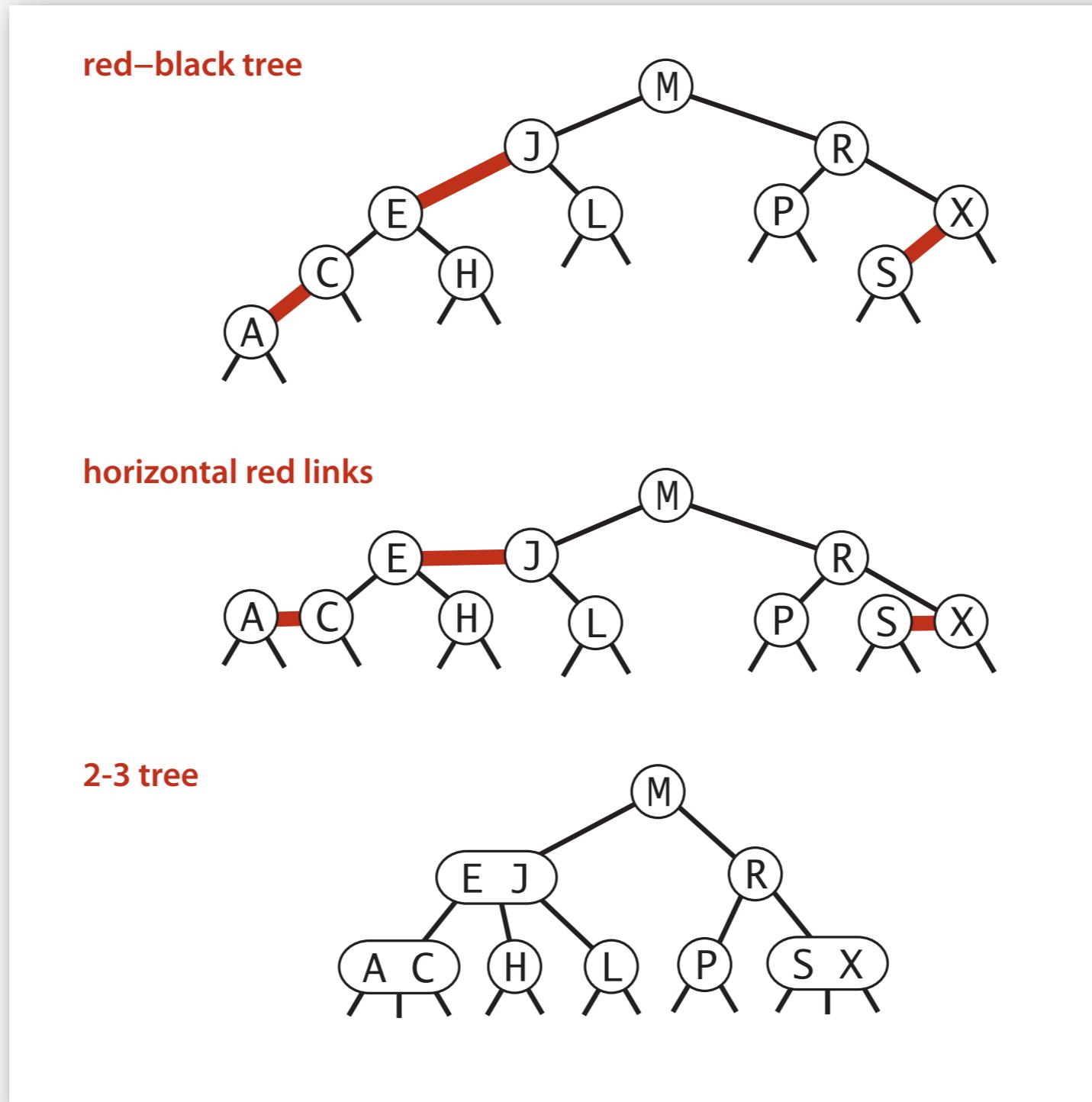
- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
 - We will only allow one red link to simulate 2 keys in node
 - A node with two red links would be the same as having 3 keys
- Red links lean left (correct ordering)

"perfect black balance"



Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1-1 correspondence between 2-3 and LLRB.



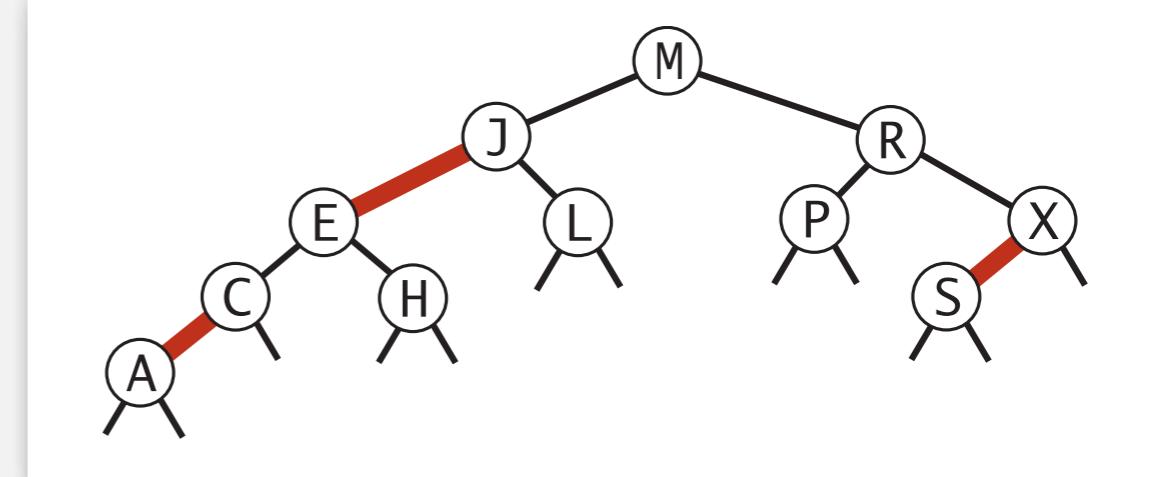
Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).



but runs faster because of better balance

```
public Val get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```



Remark. Most other ops (e.g., ceiling, selection, iteration) are also identical.

Red-black BST representation

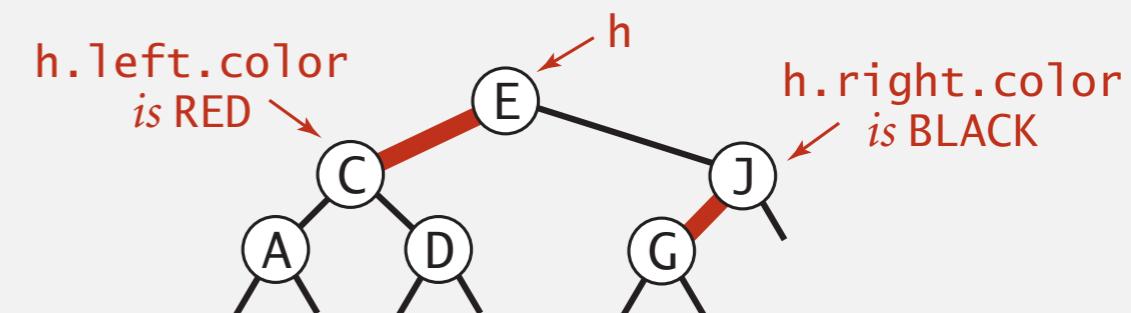
Each node is pointed to by precisely one link (from its parent) \Rightarrow
can encode color of links in nodes.

```
private static final boolean RED = true;
private static final boolean BLACK = false;
```

```
private class Node
{
    Key key;
    Value val;
    Node left, right;
    boolean color; // color of parent link
}
```

```
private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

null links are black

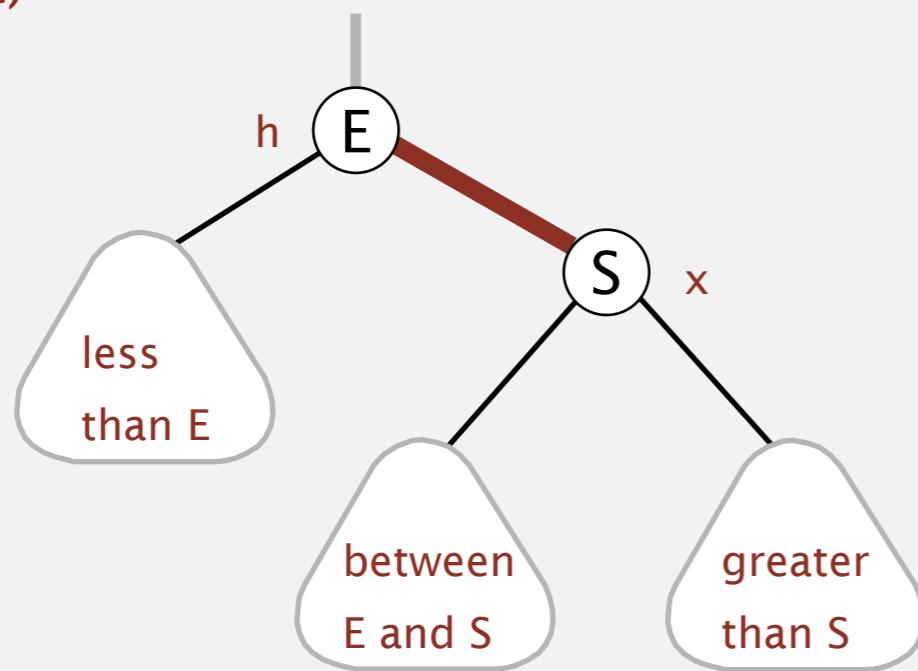


Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(before)



```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

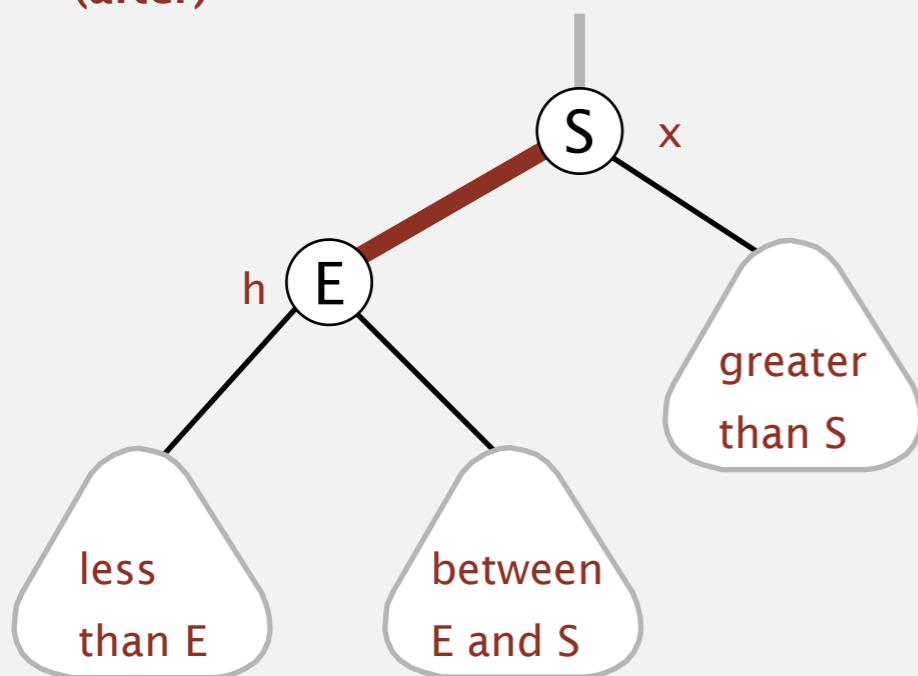
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(after)



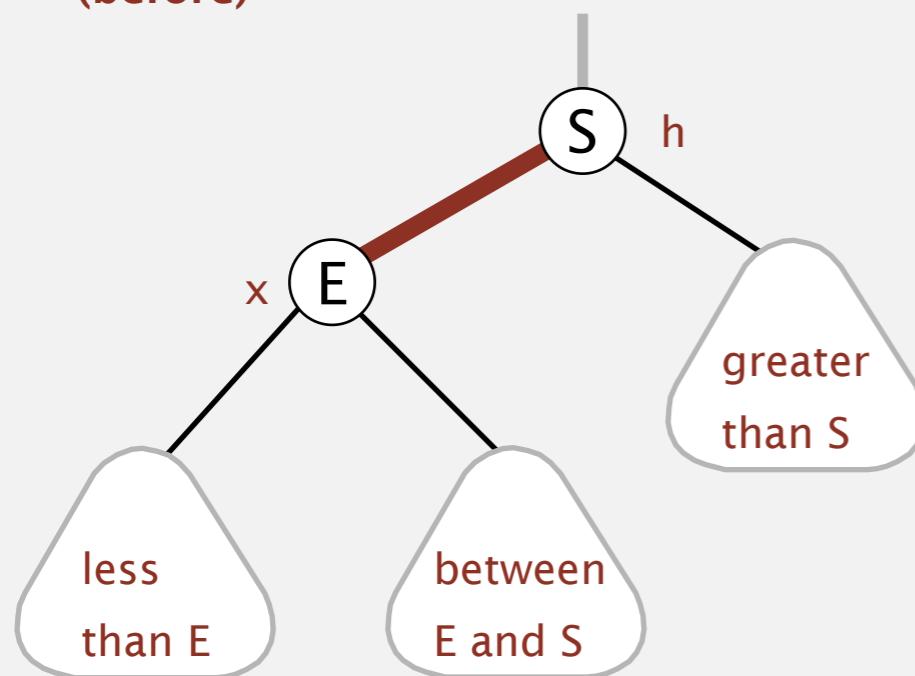
```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right
(before)



```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

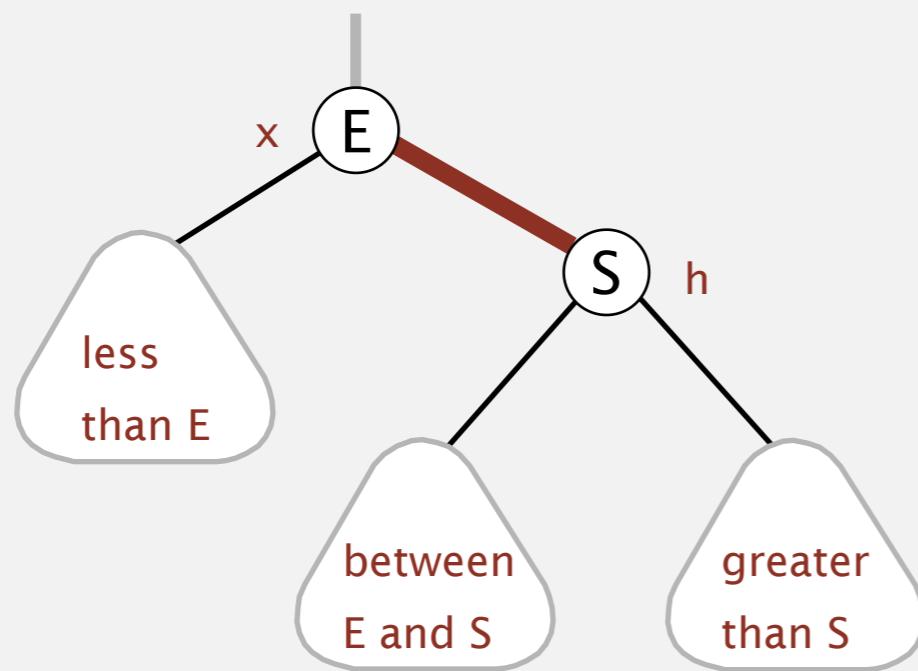
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(after)

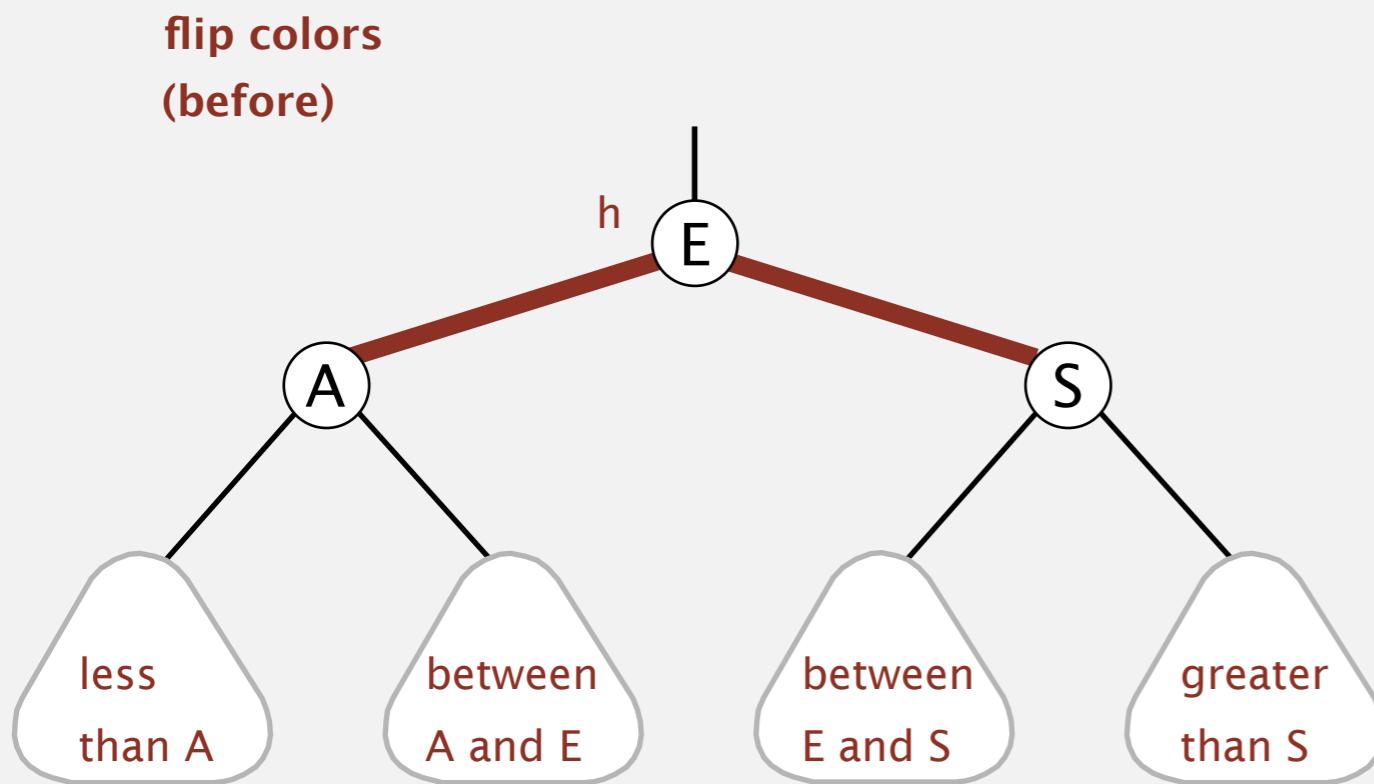


```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

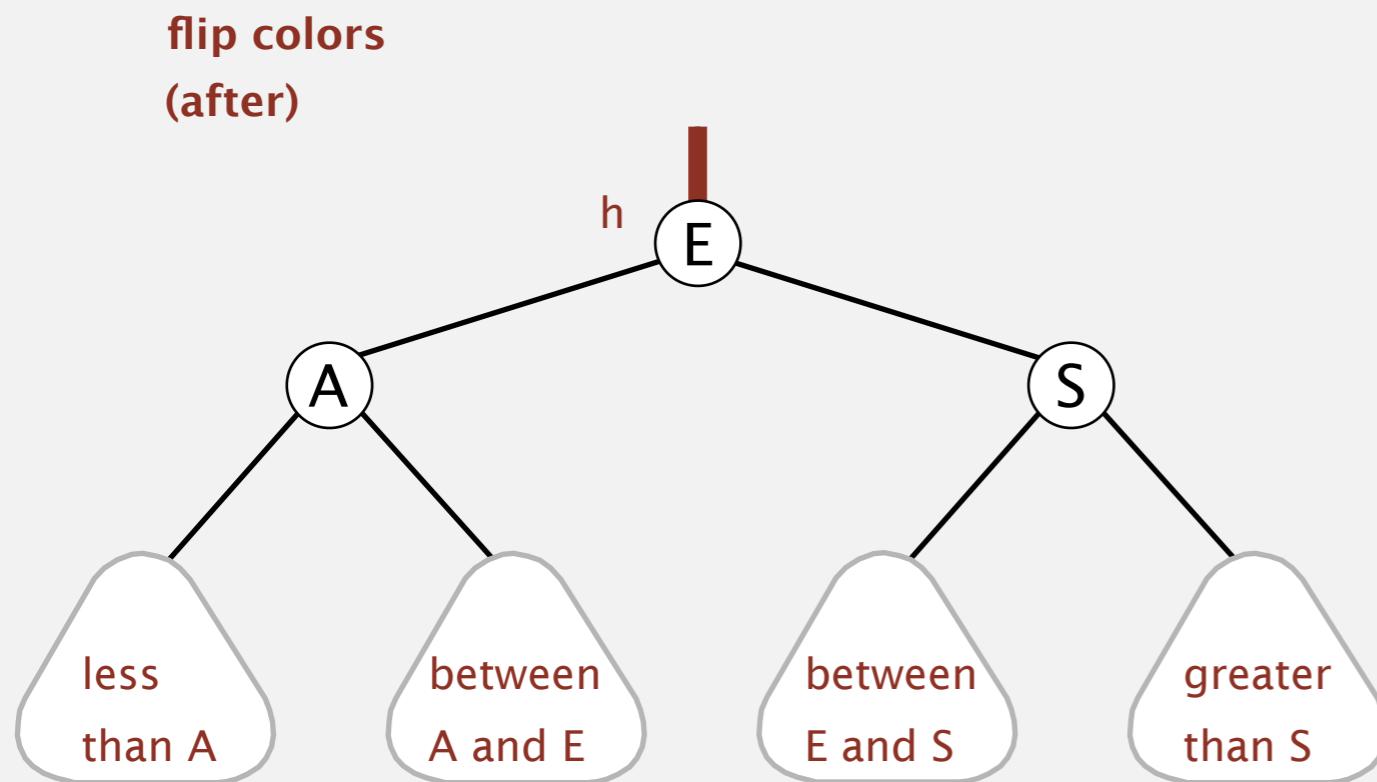


```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

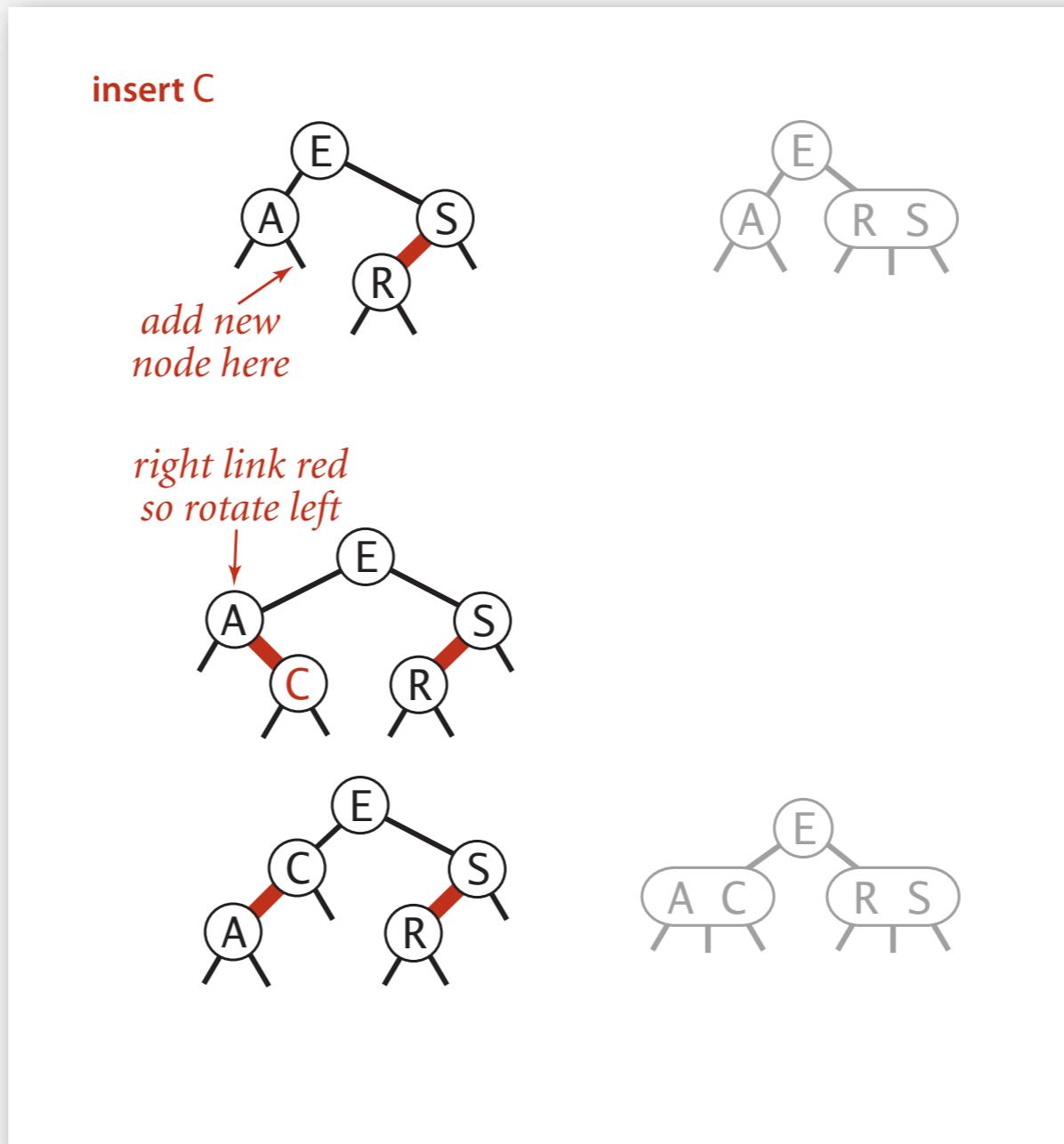


```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

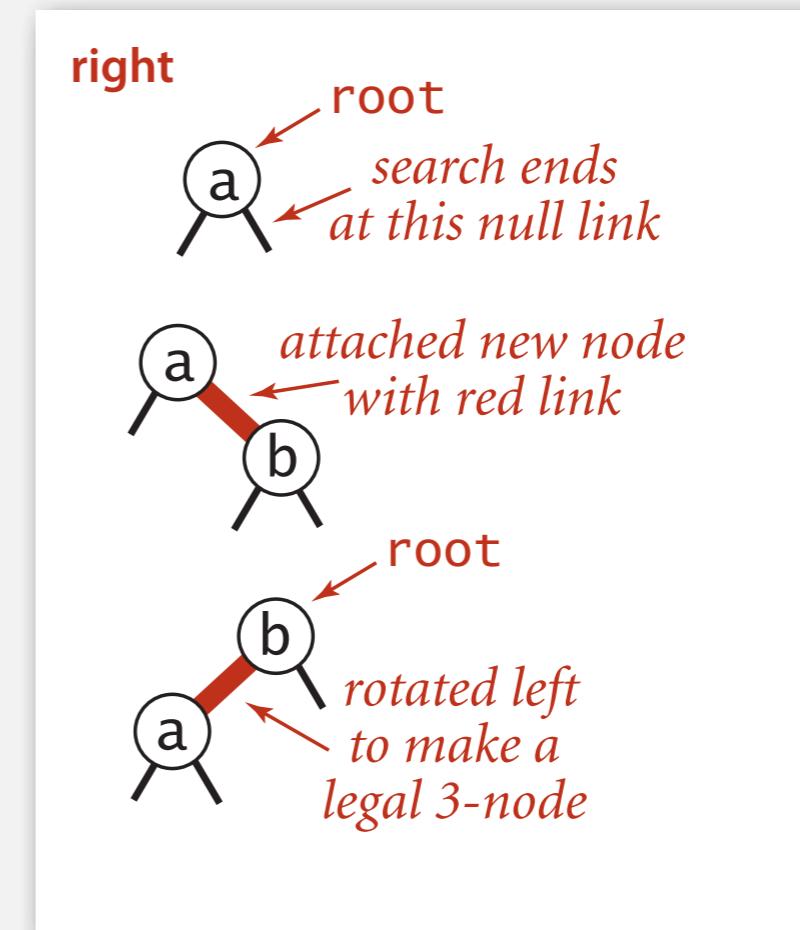
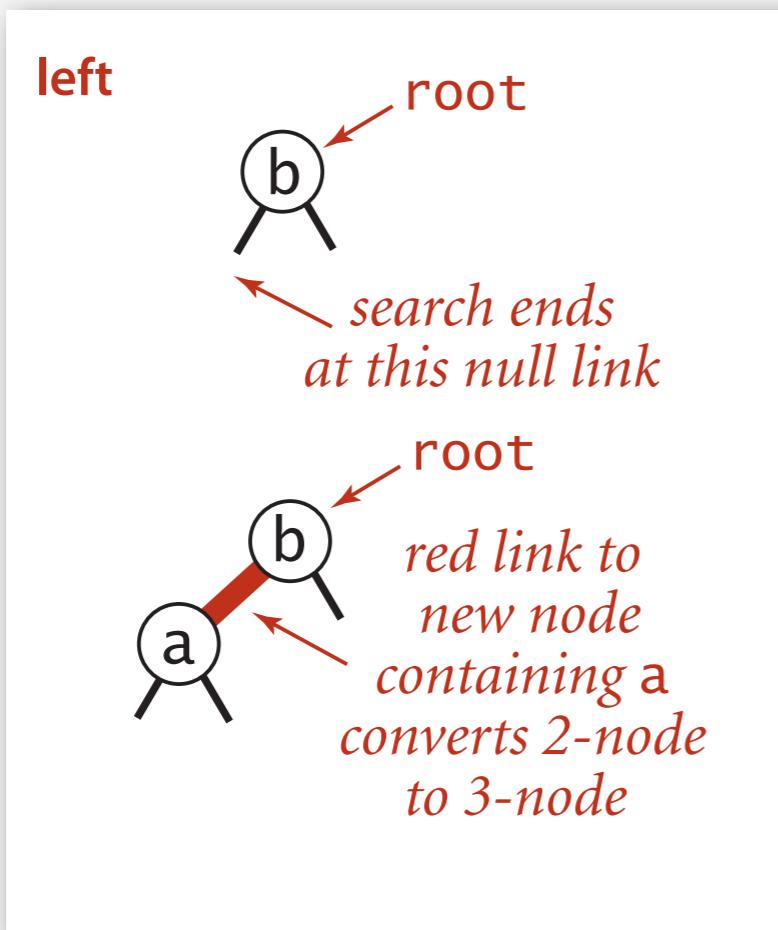
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black BST operations.



Insertion in a LLRB tree

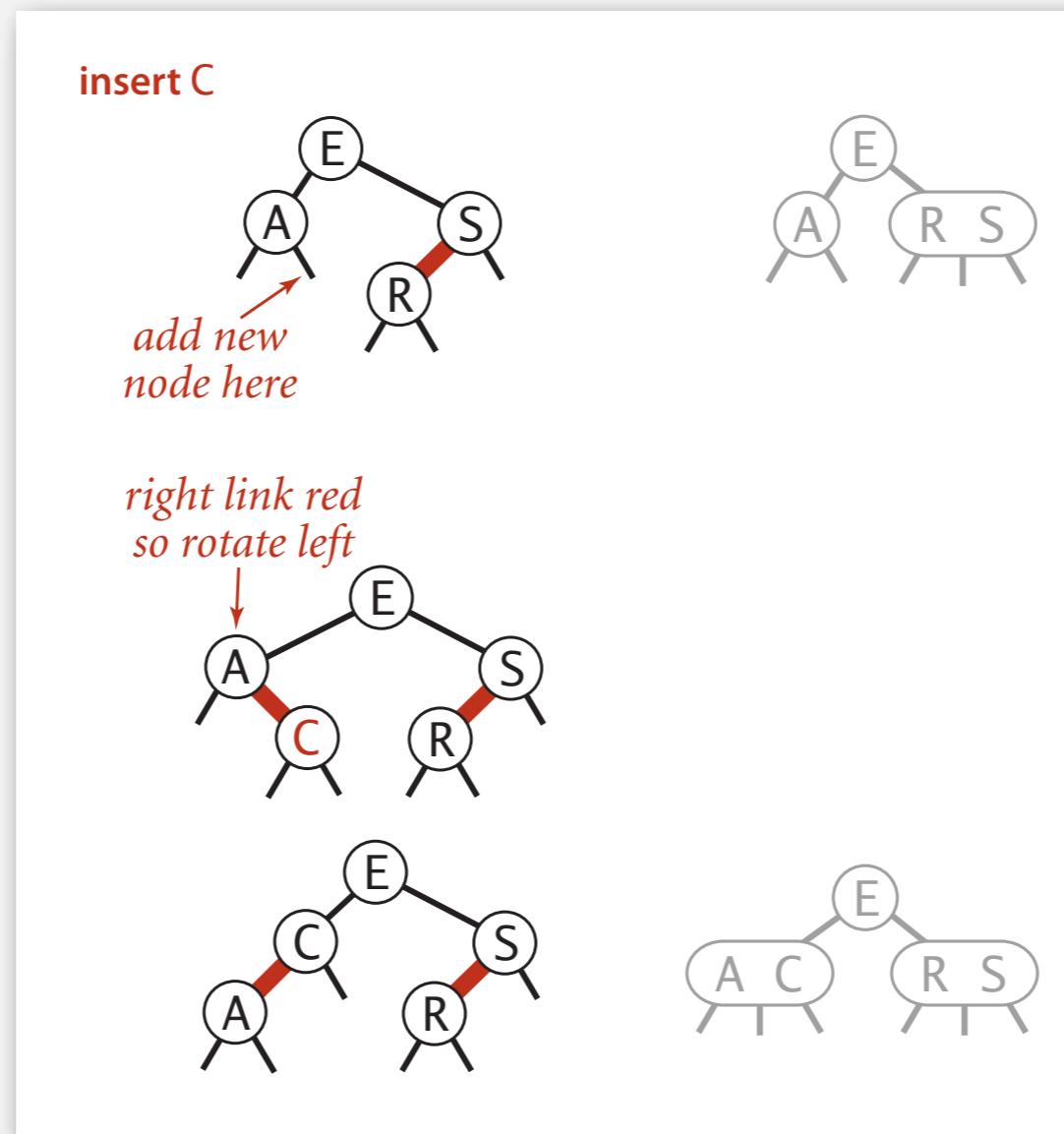
Warmup I. Insert into a tree with exactly 1 node.



Insertion in a LLRB tree

Case I. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red.
- If new red link is a right link, rotate left.

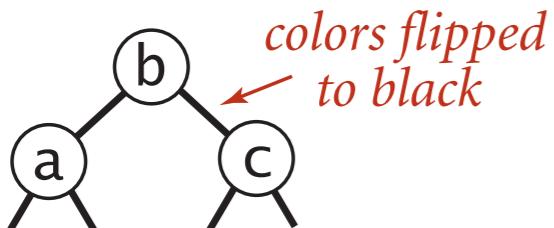
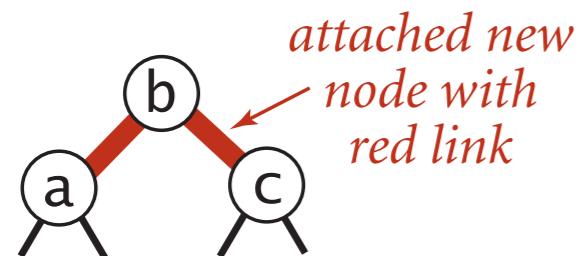


Insertion in a LLRB tree

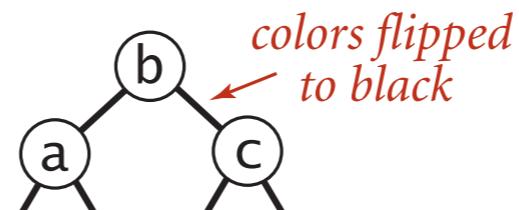
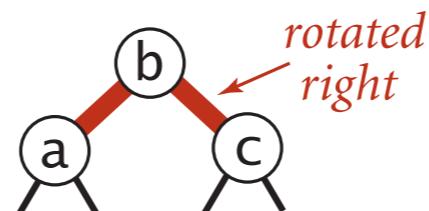
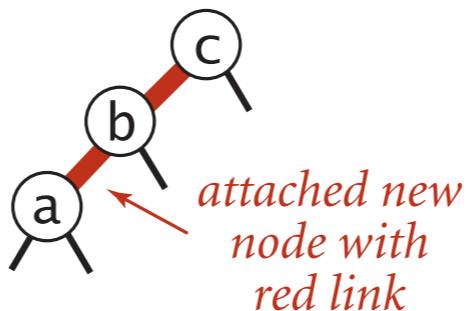
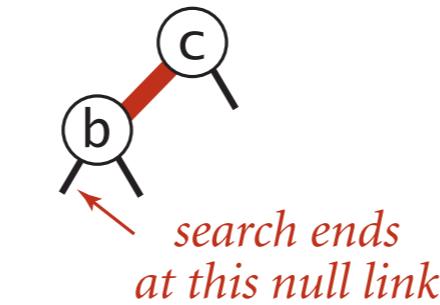
Warmup 2. Insert into a tree with exactly 2 nodes.

Think of this as a split in 2-3 tree

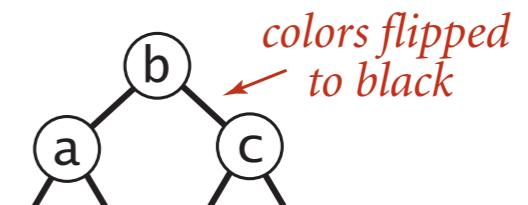
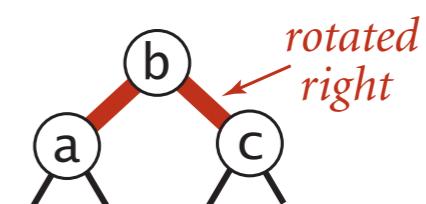
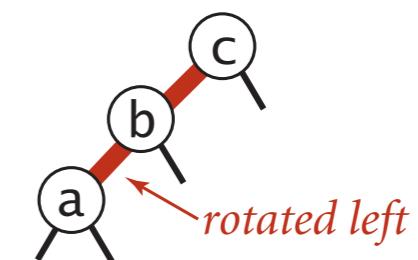
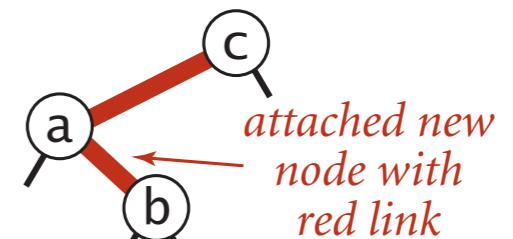
larger



smaller



between

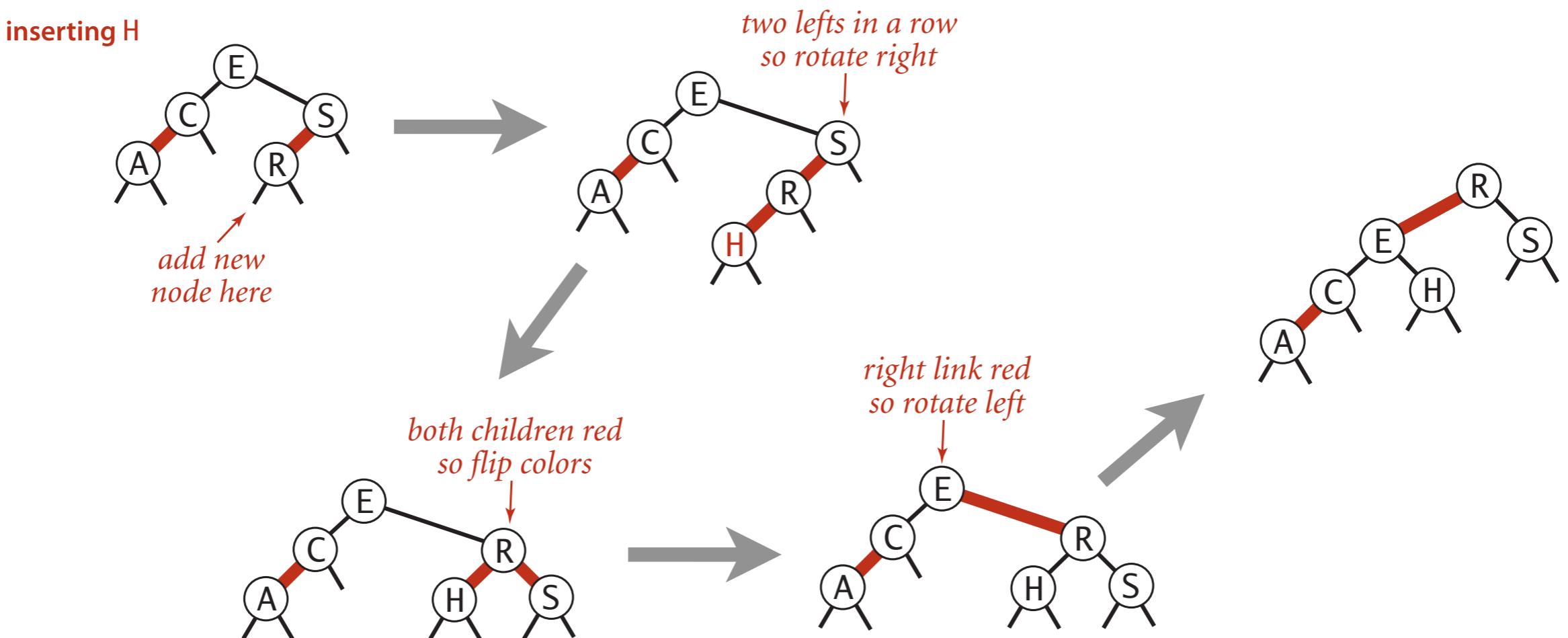


Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

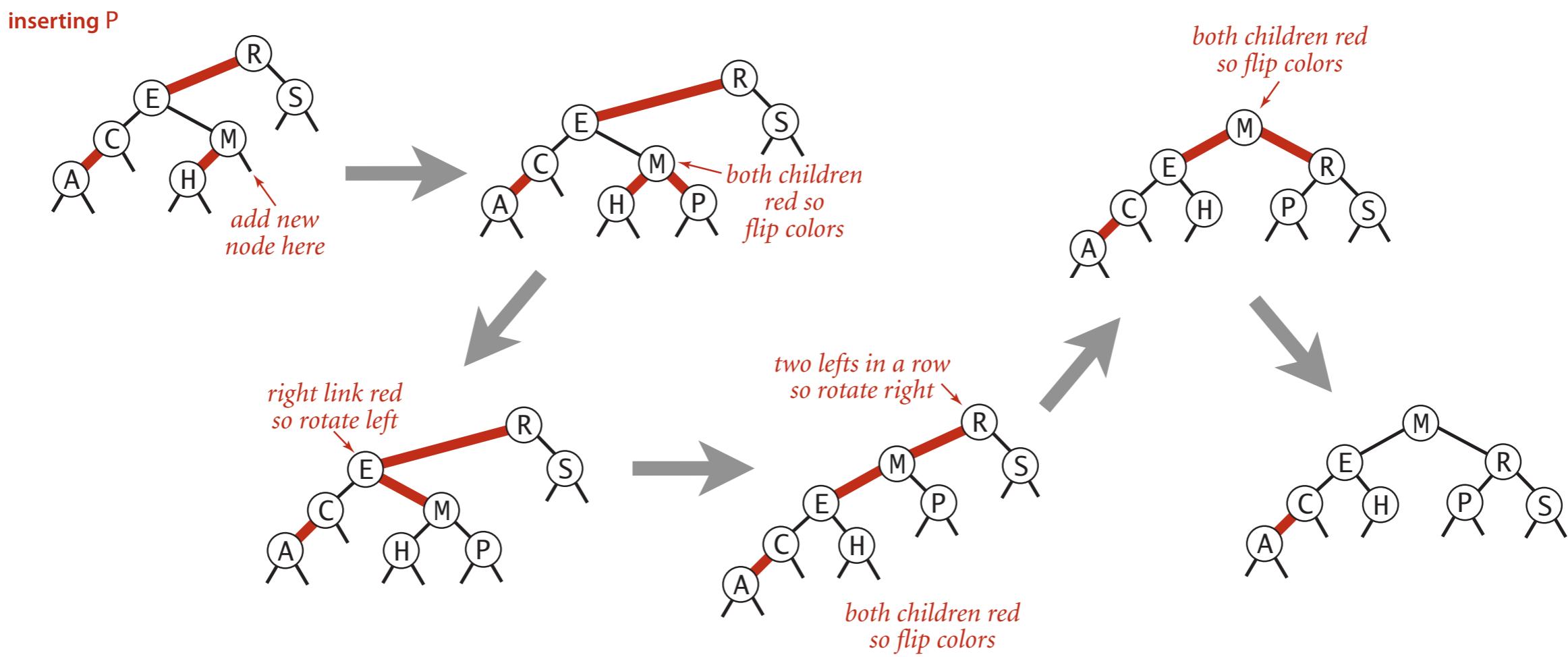
As with 2-3 Trees
we have to update parents,
bottom-to-top if we violate the
conditions



Insertion in a LLRB tree: passing red links up the tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).



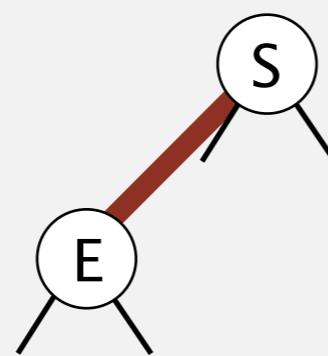
Red-black BST insertion

insert S



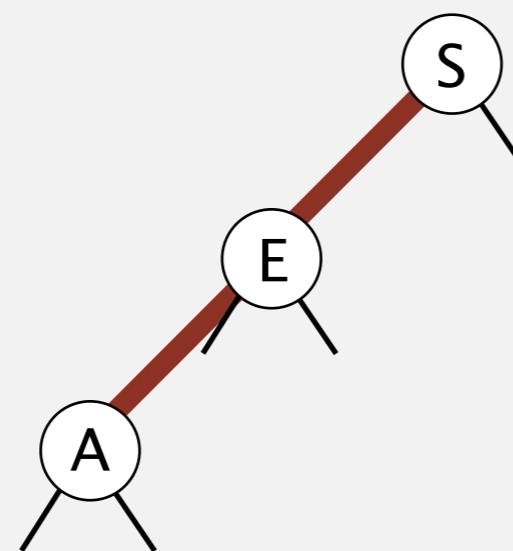
Red-black BST insertion

insert E



Red-black BST insertion

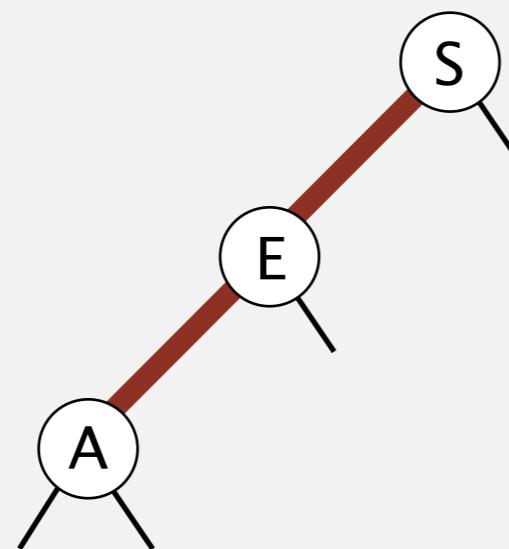
insert A



Red-black BST insertion

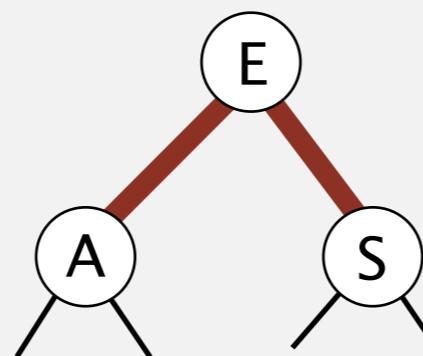
insert A

two left reds in a row
(rotate S right)



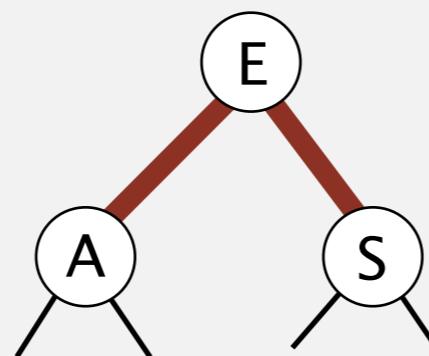
Red-black BST insertion

both children red
(flip colors)



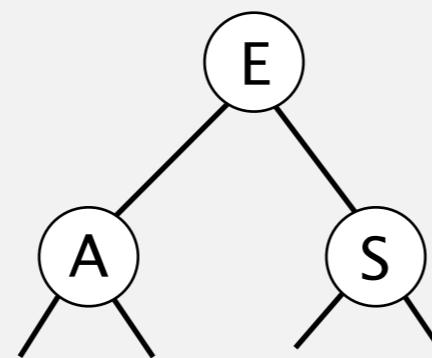
Red-black BST insertion

both children red
(flip colors)



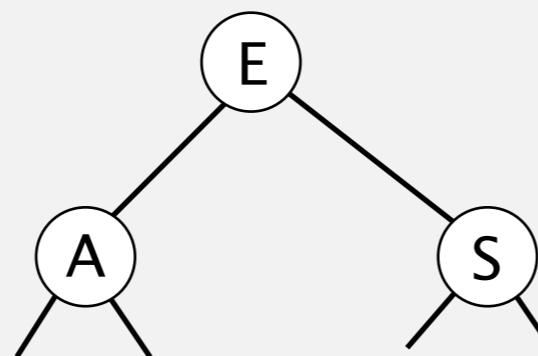
Red-black BST insertion

red-black BST



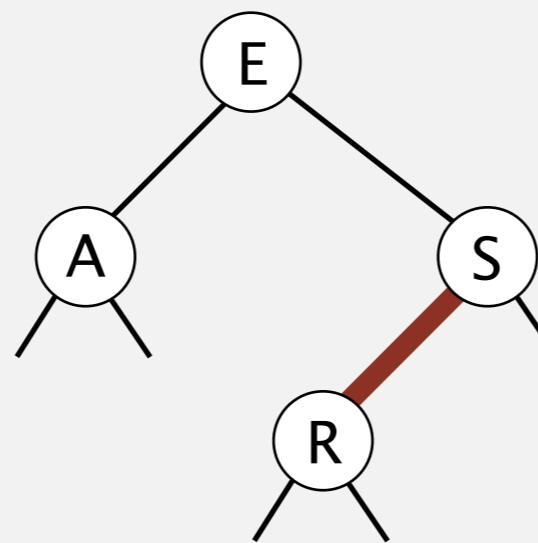
Red-black BST insertion

red-black BST



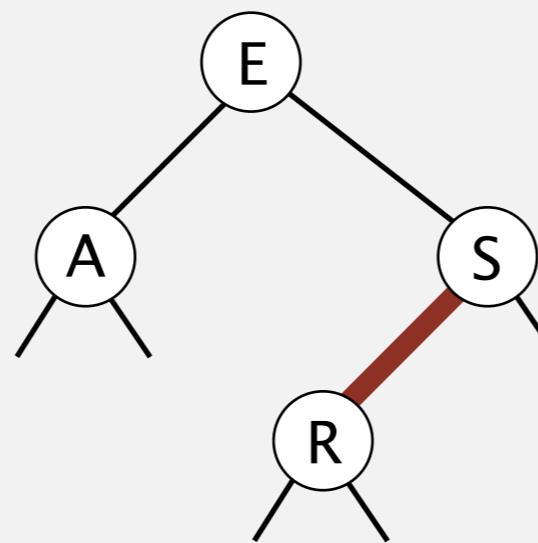
Red-black BST insertion

insert R



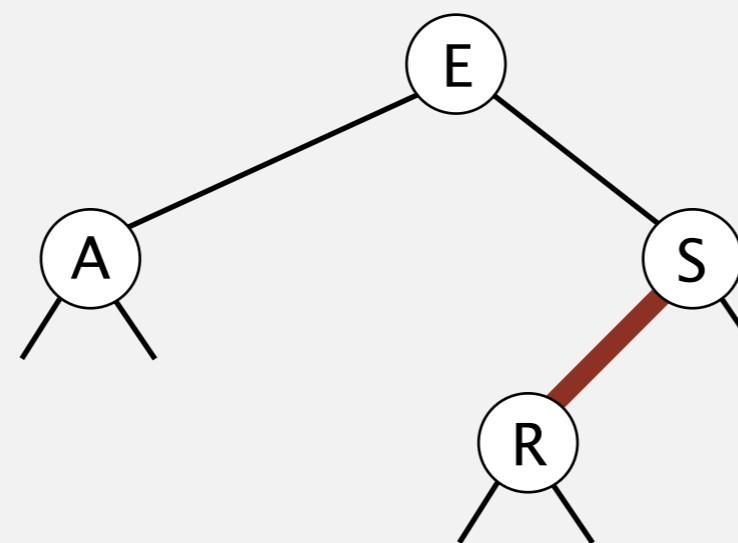
Red-black BST insertion

red-black BST



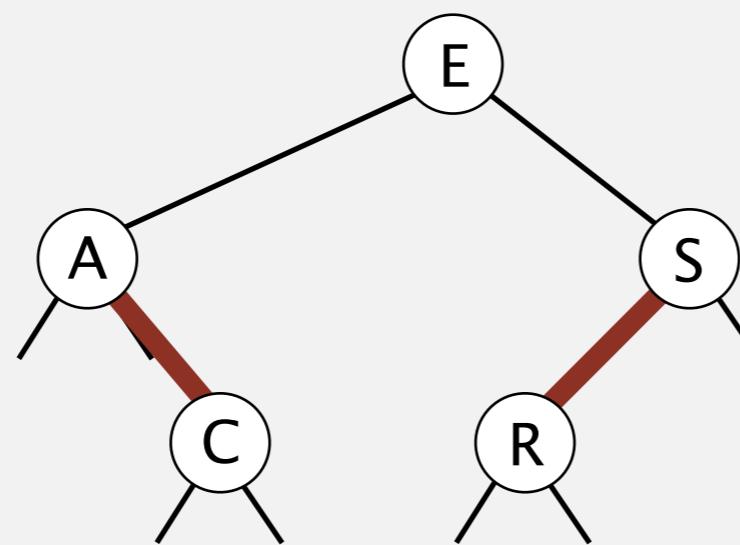
Red-black BST insertion

red-black BST

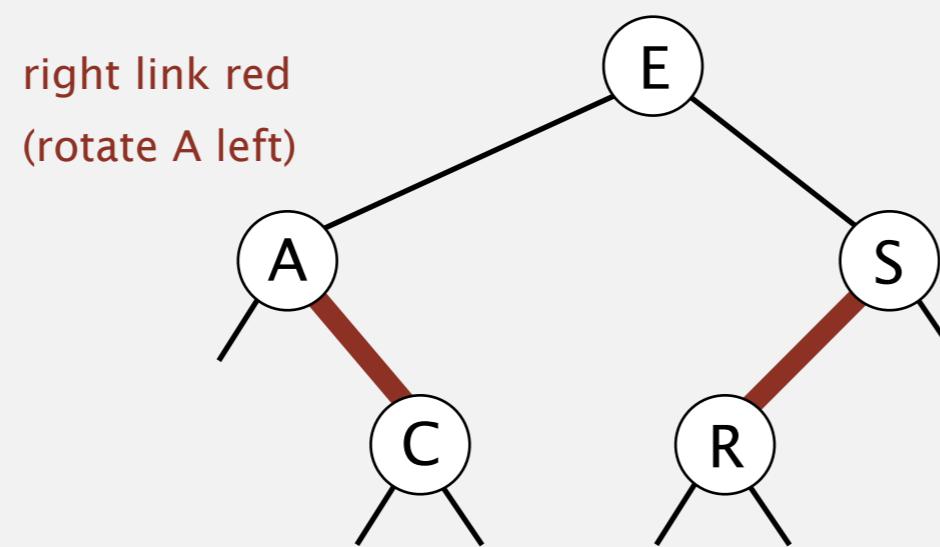


Red-black BST insertion

insert C

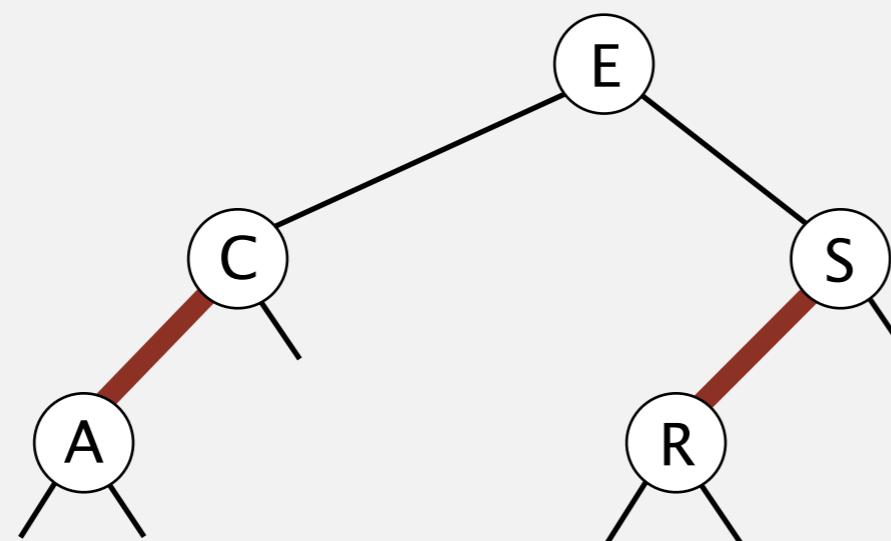


Red-black BST insertion



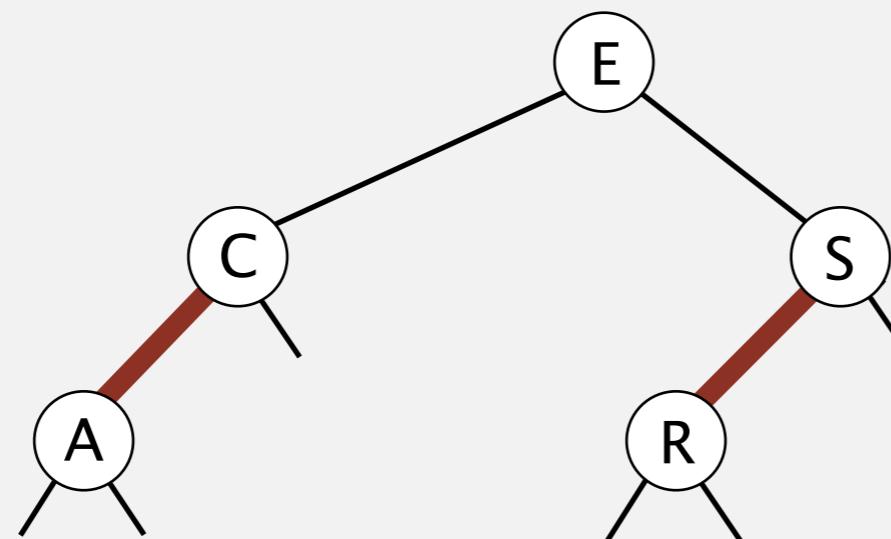
Red-black BST insertion

red-black BST



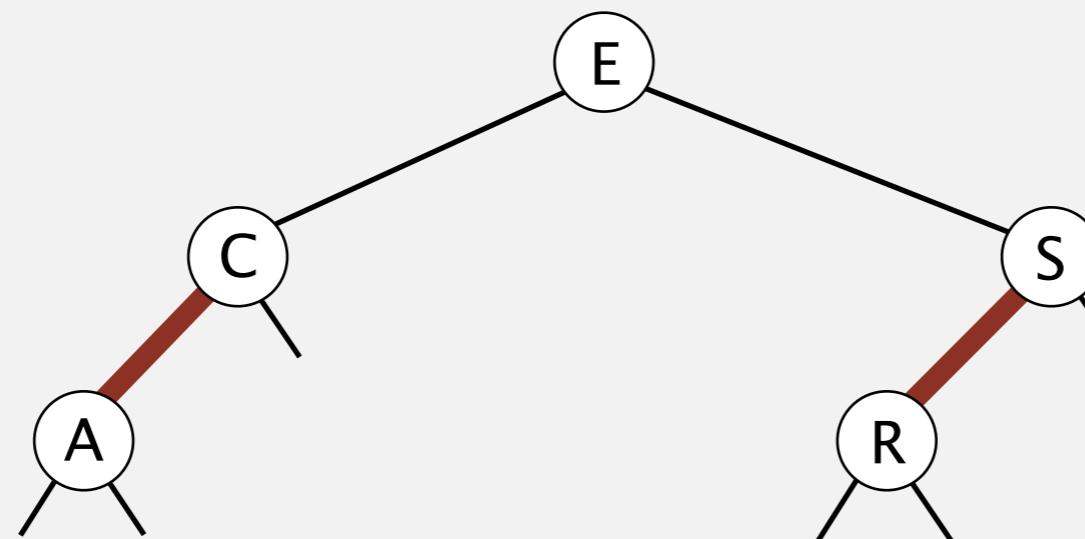
Red-black BST insertion

red-black BST



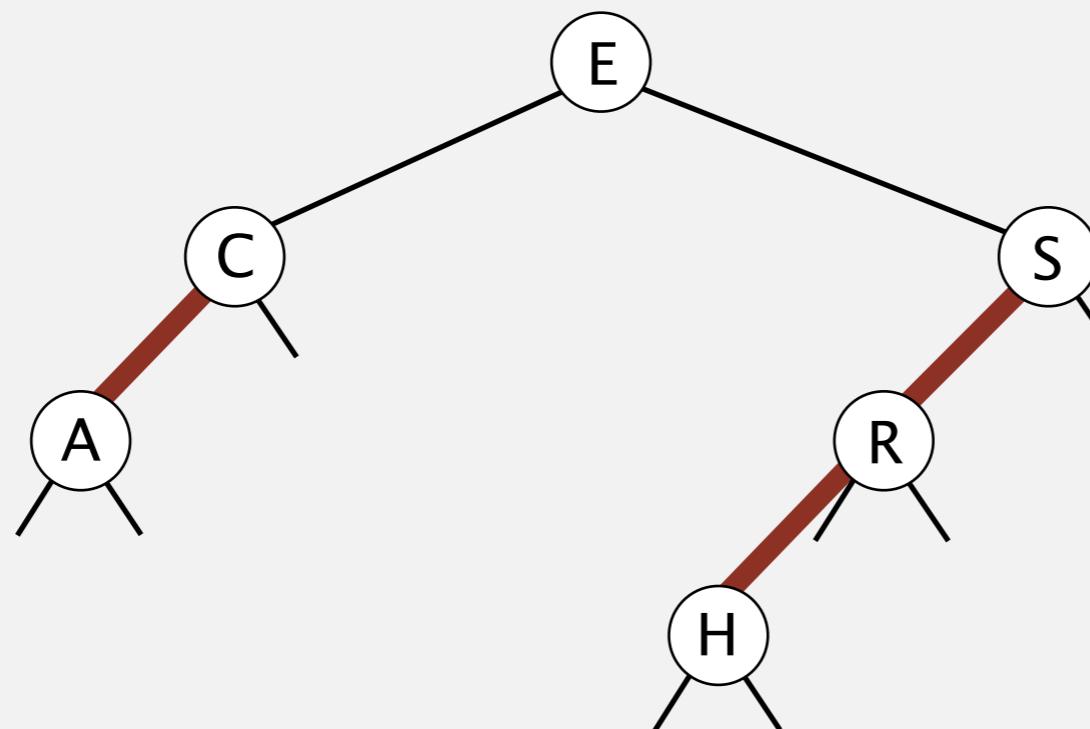
Red-black BST insertion

red-black BST

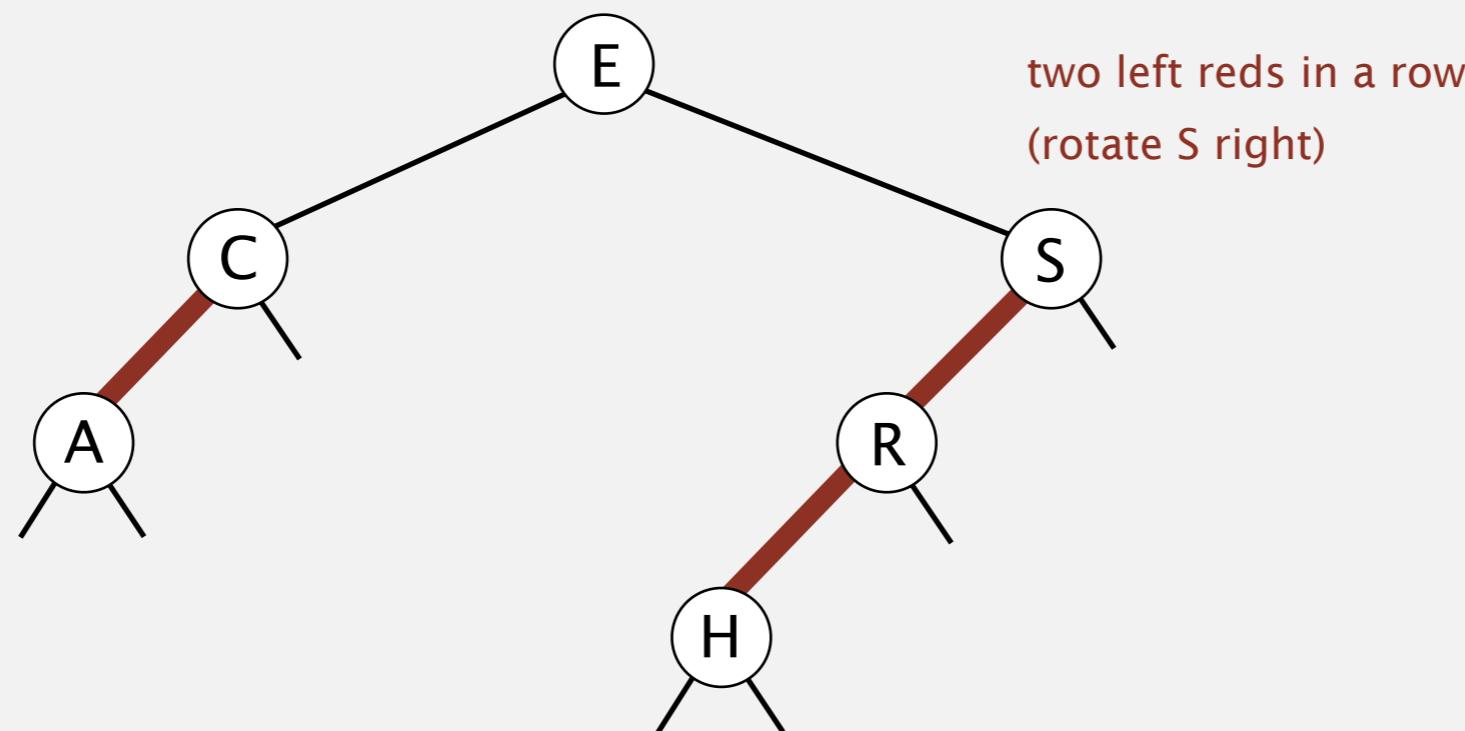


Red-black BST insertion

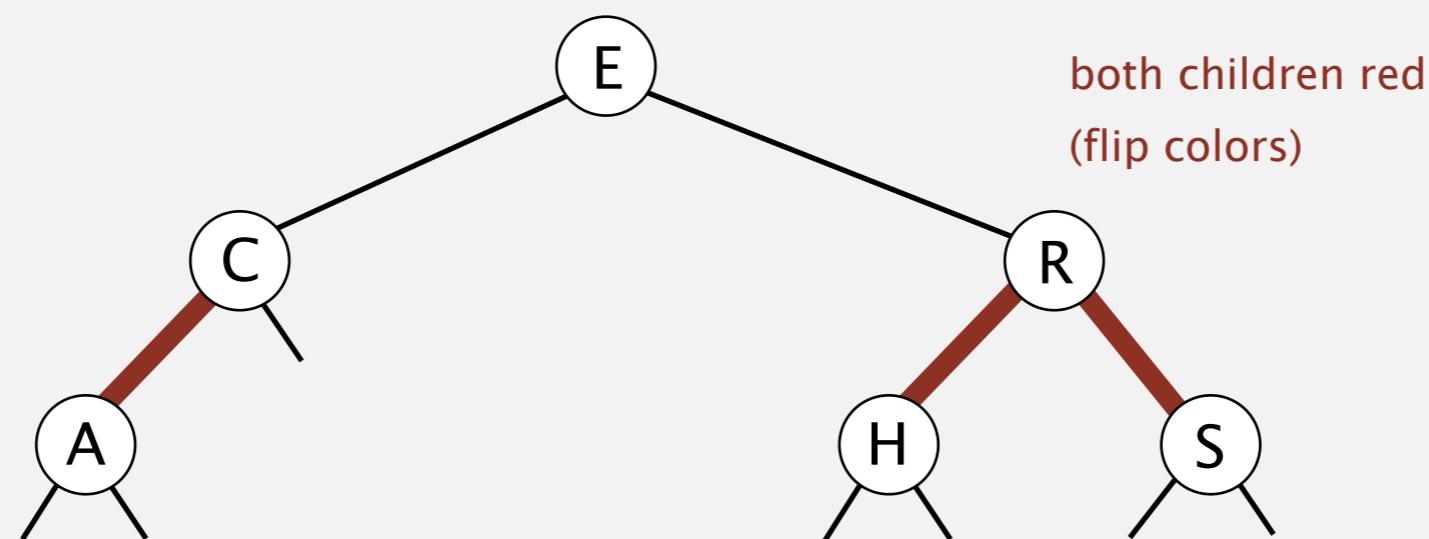
insert H



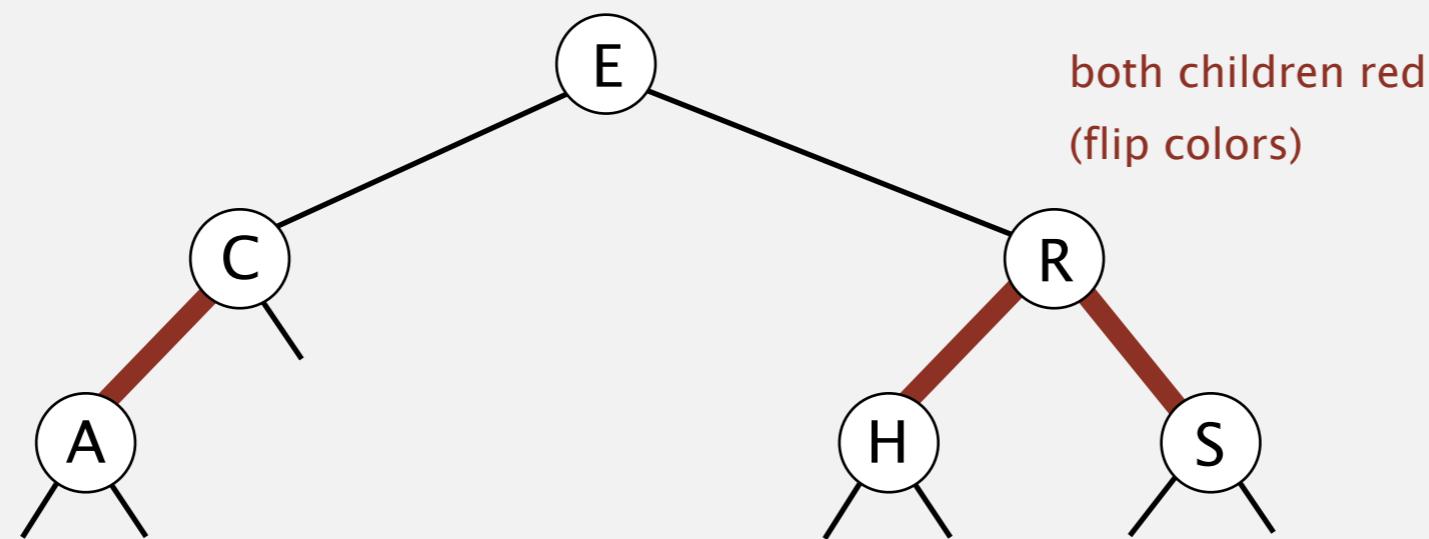
Red-black BST insertion



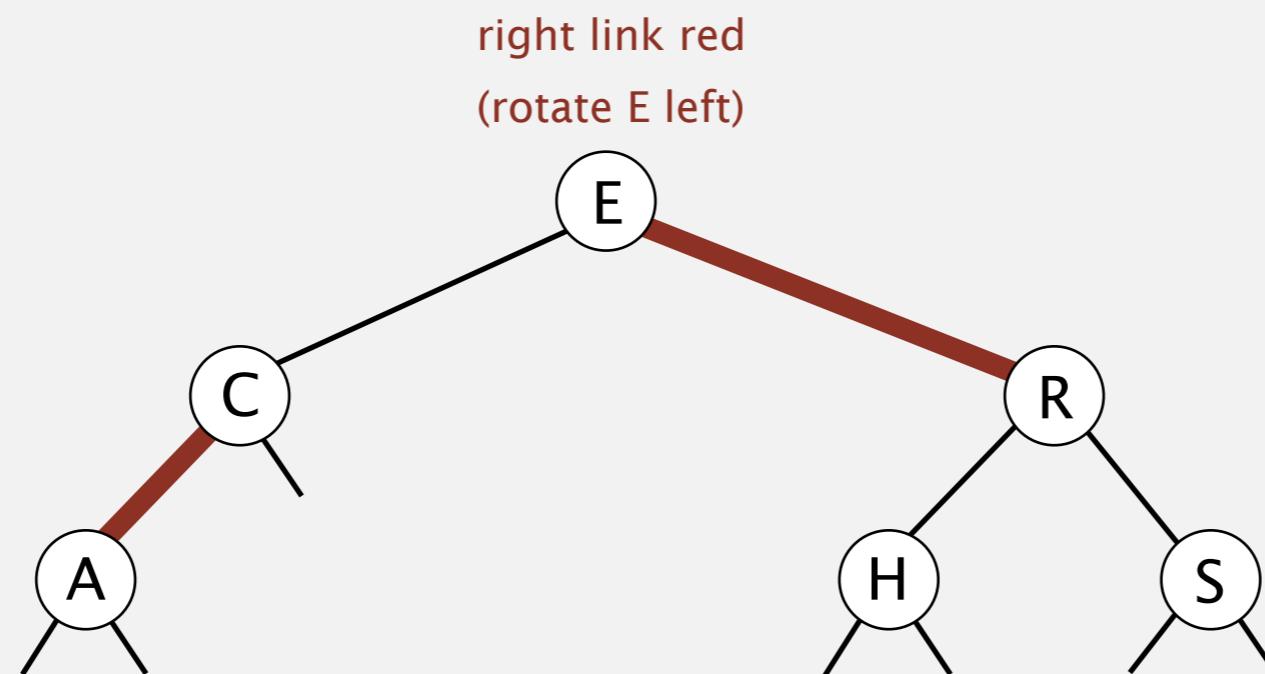
Red-black BST insertion



Red-black BST insertion

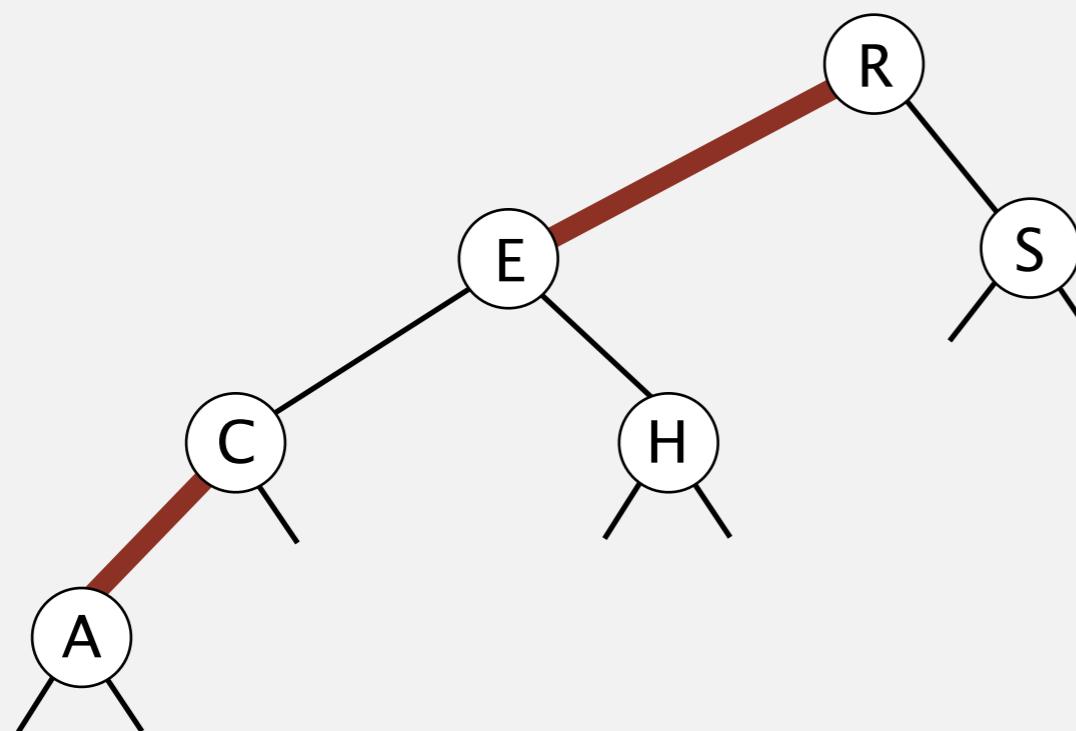


Red-black BST insertion



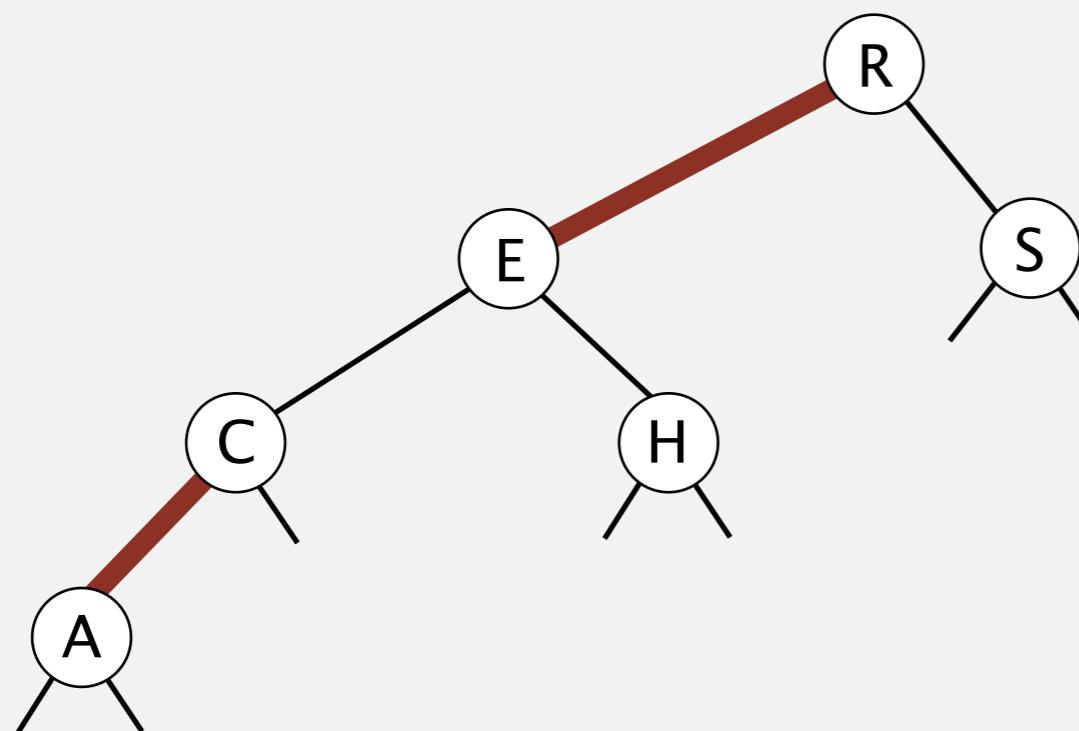
Red-black BST insertion

red-black BST



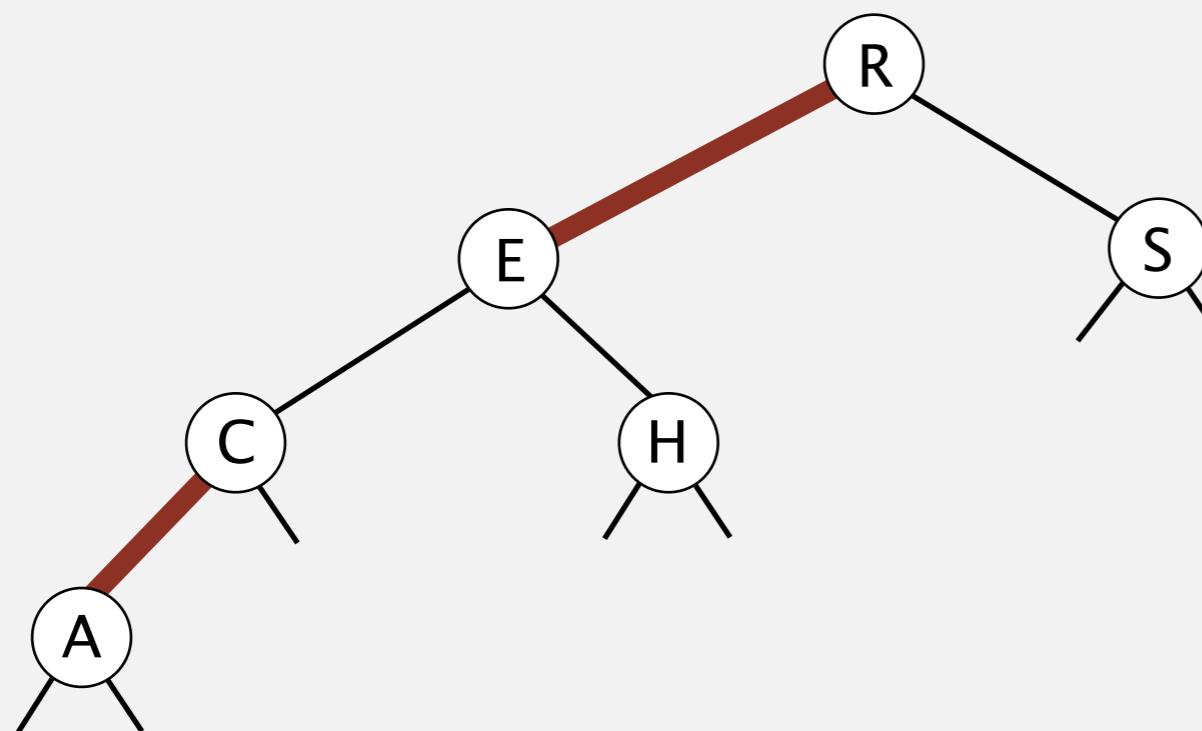
Red-black BST insertion

red-black BST



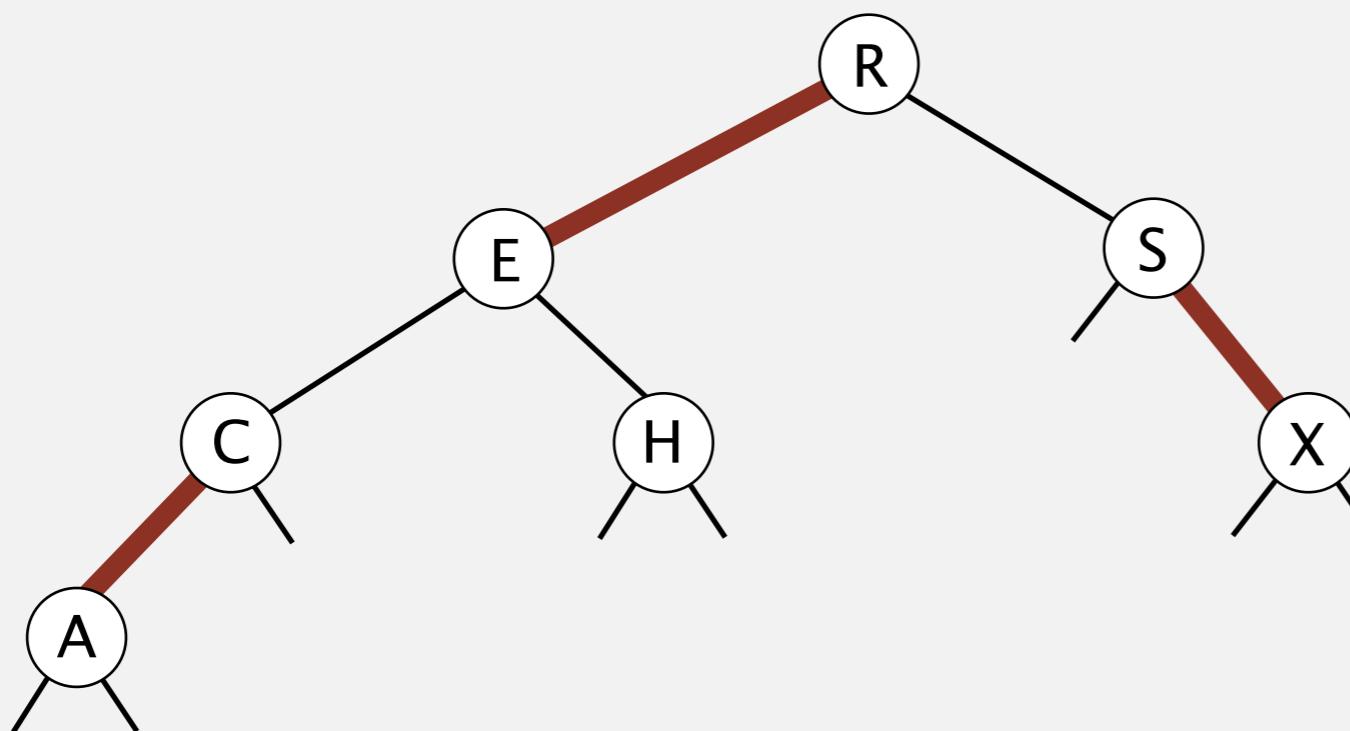
Red-black BST insertion

red-black BST



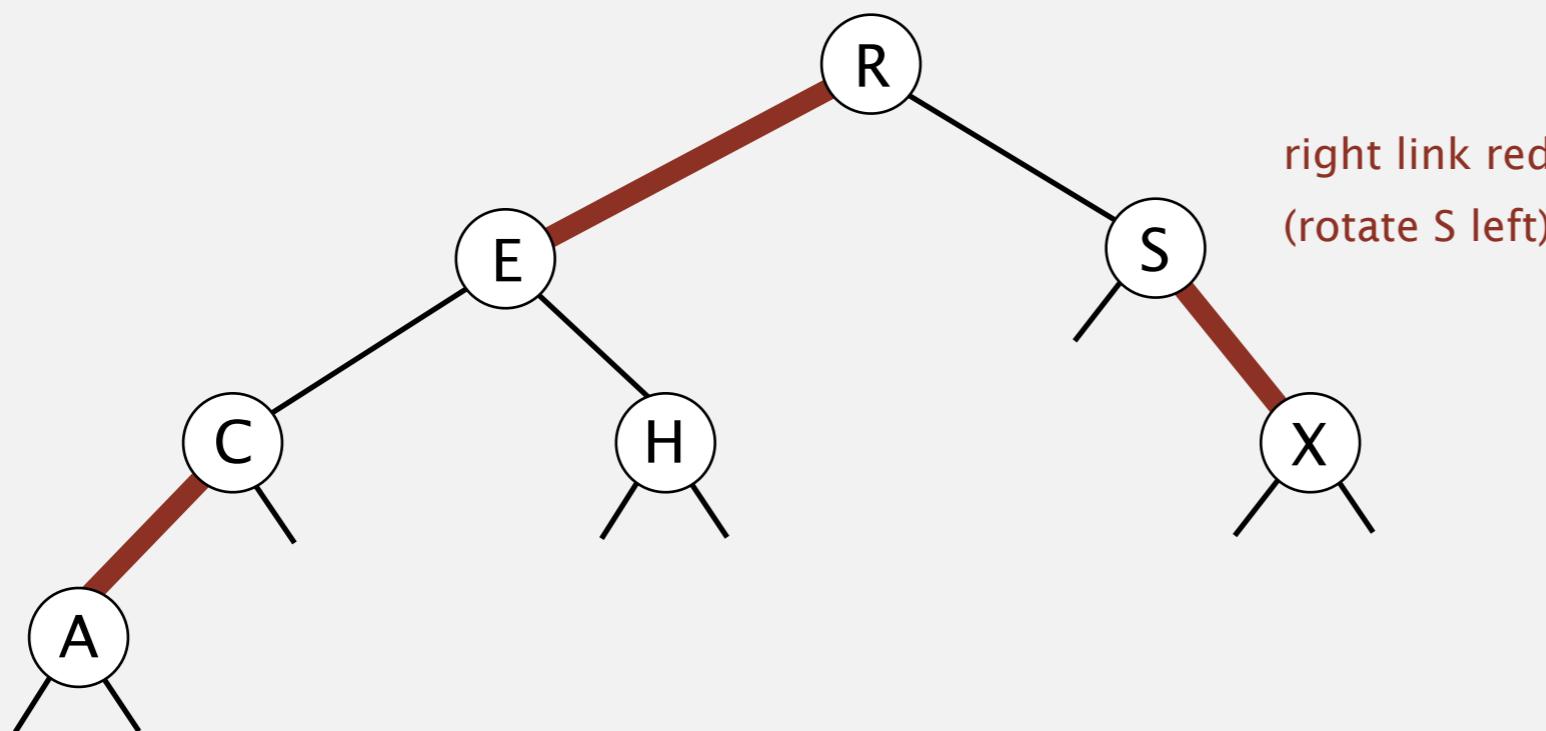
Red-black BST insertion

insert X



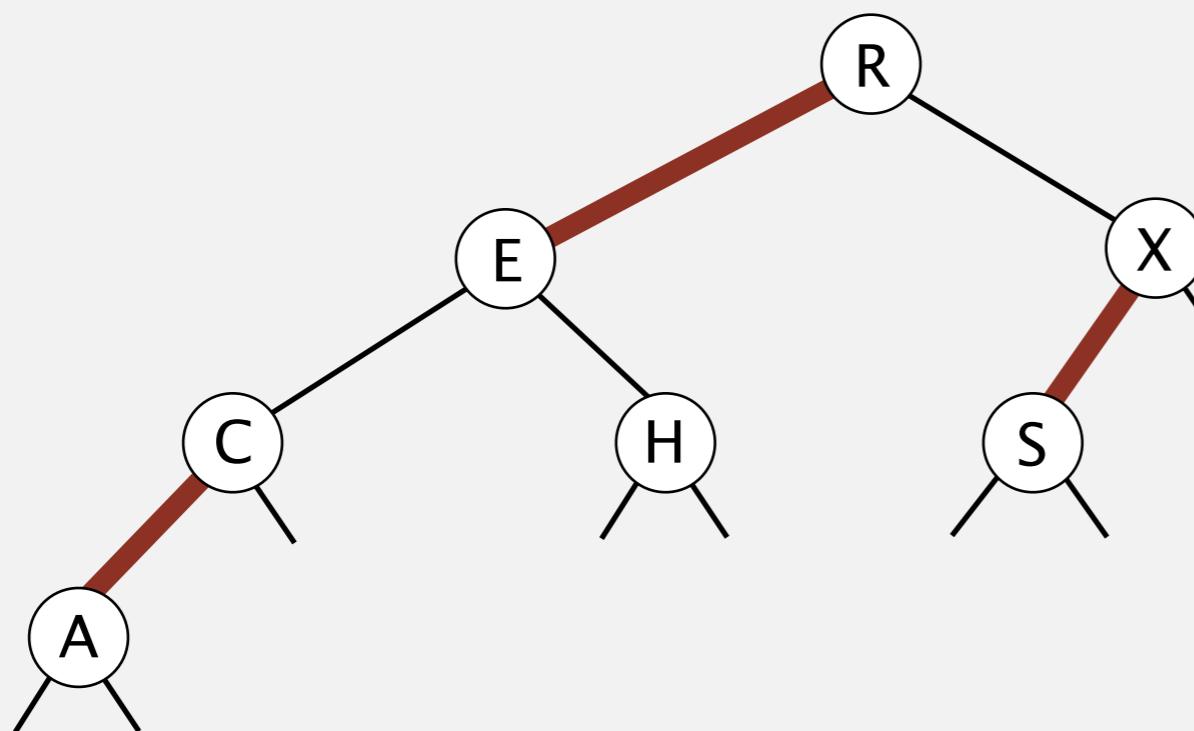
Red-black BST insertion

insert X



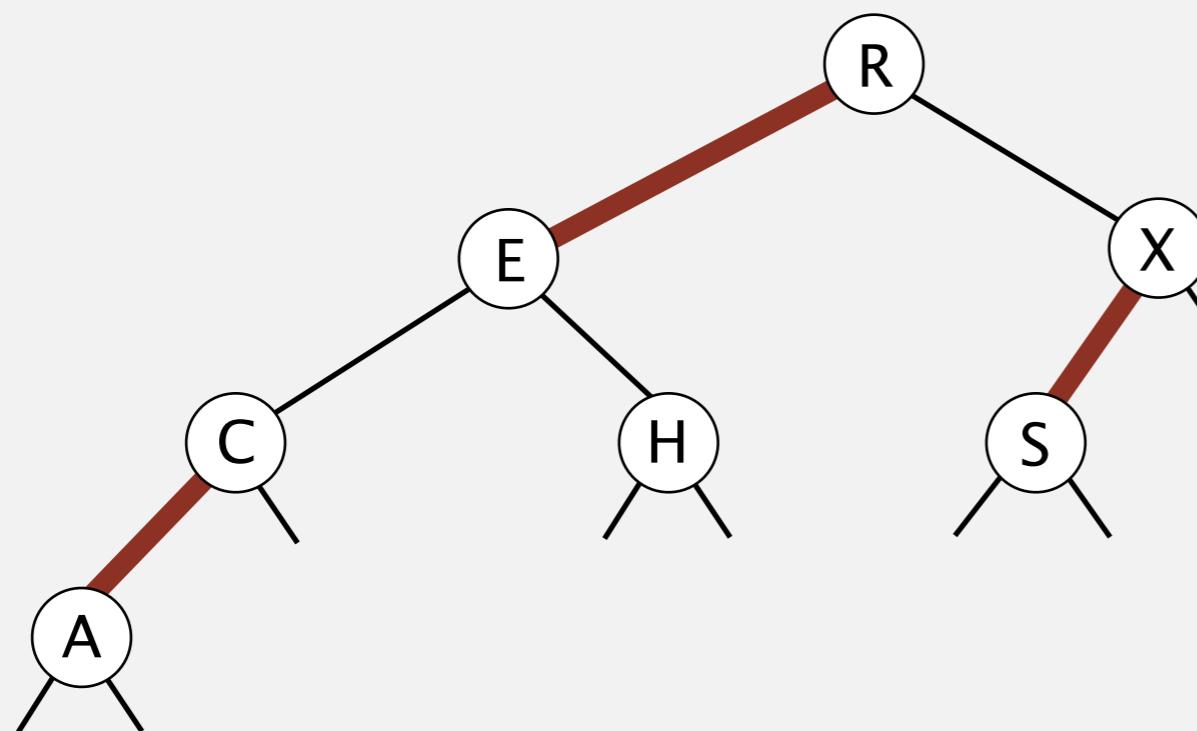
Red-black BST insertion

red-black BST



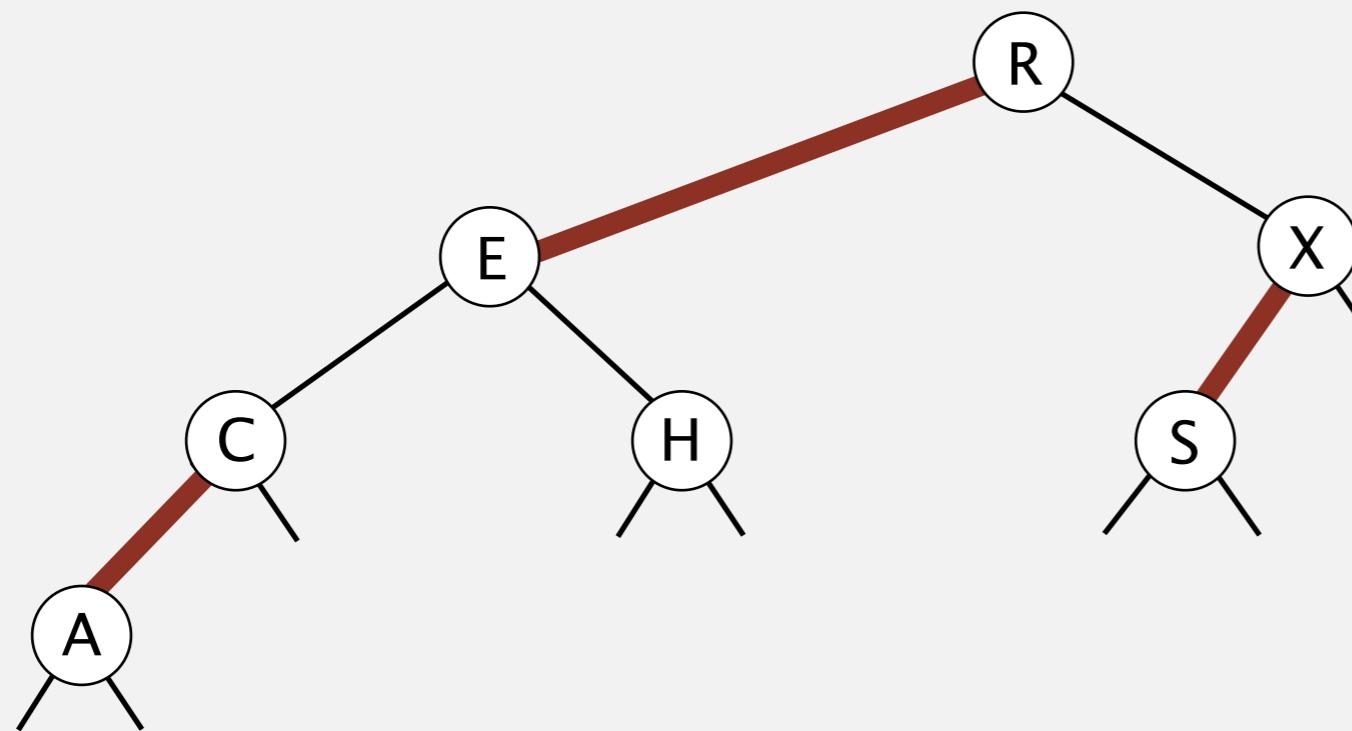
Red-black BST insertion

red-black BST



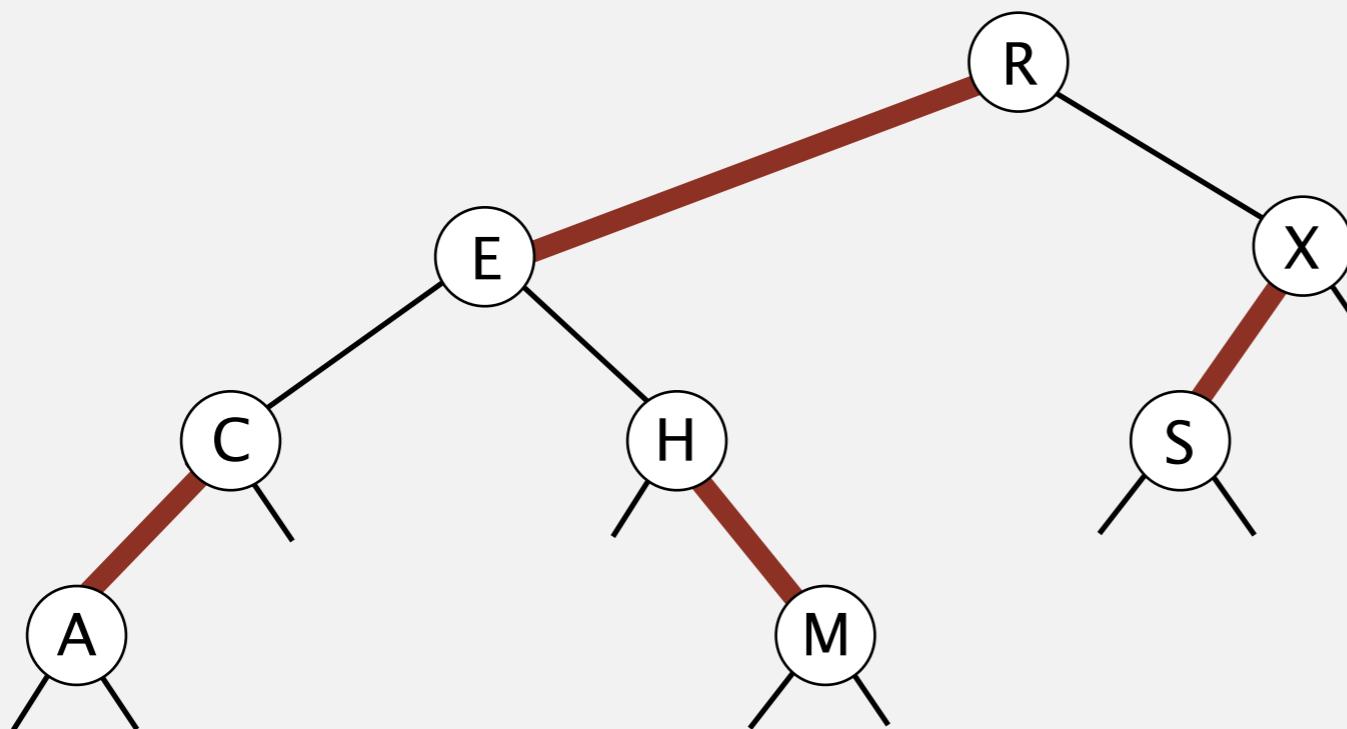
Red-black BST insertion

red-black BST



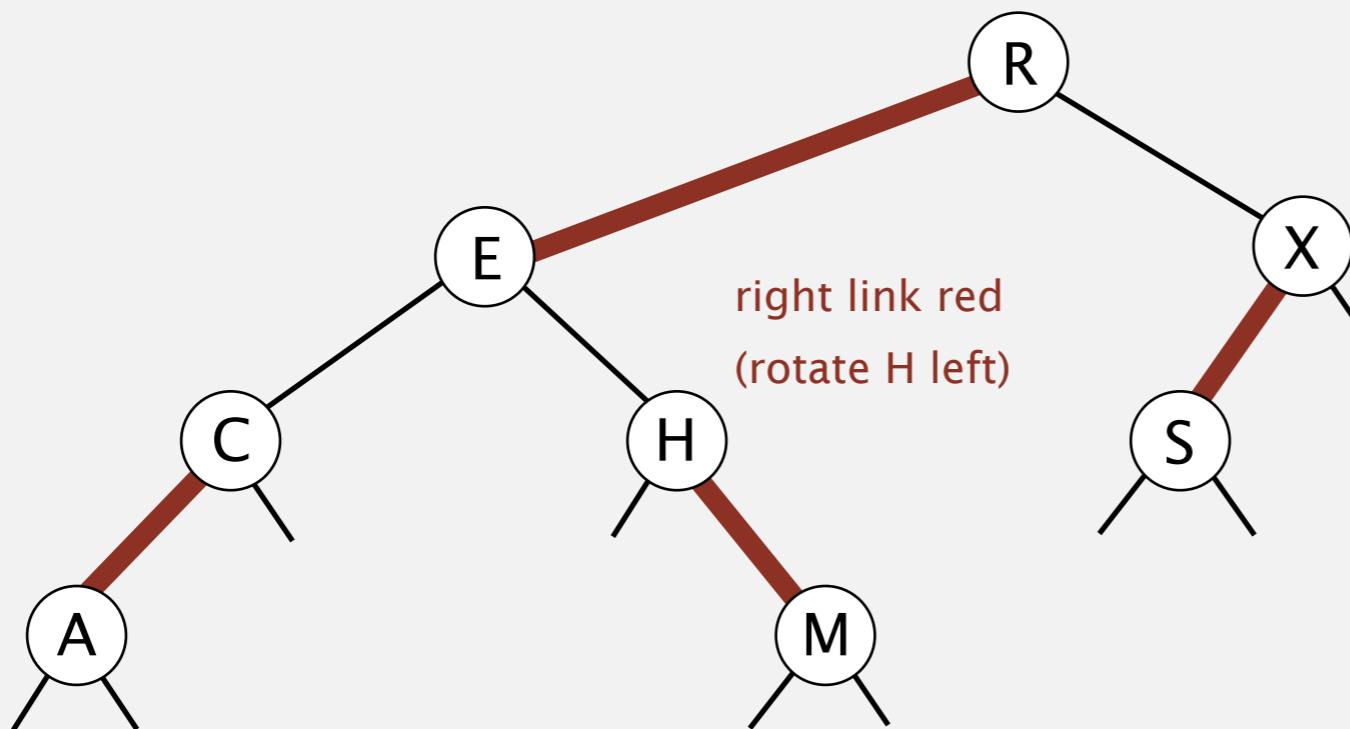
Red-black BST insertion

insert M



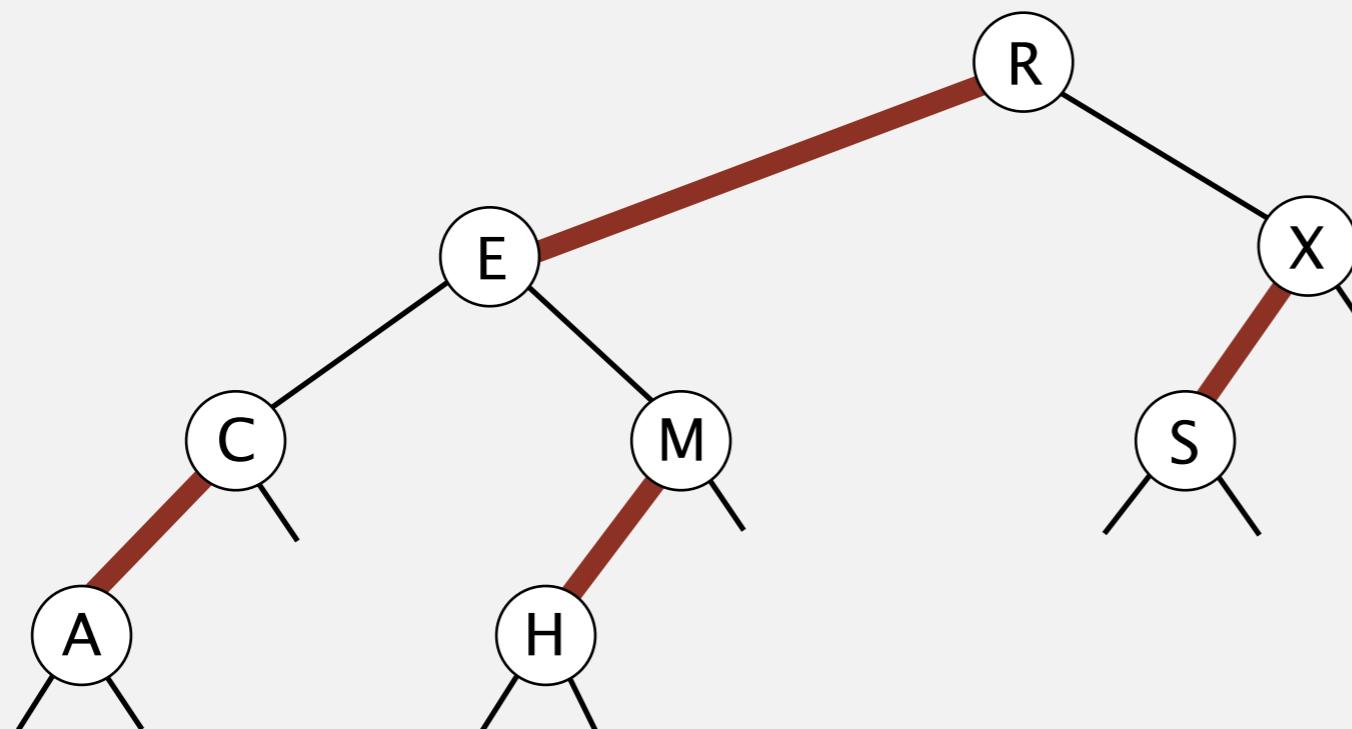
Red-black BST insertion

insert M



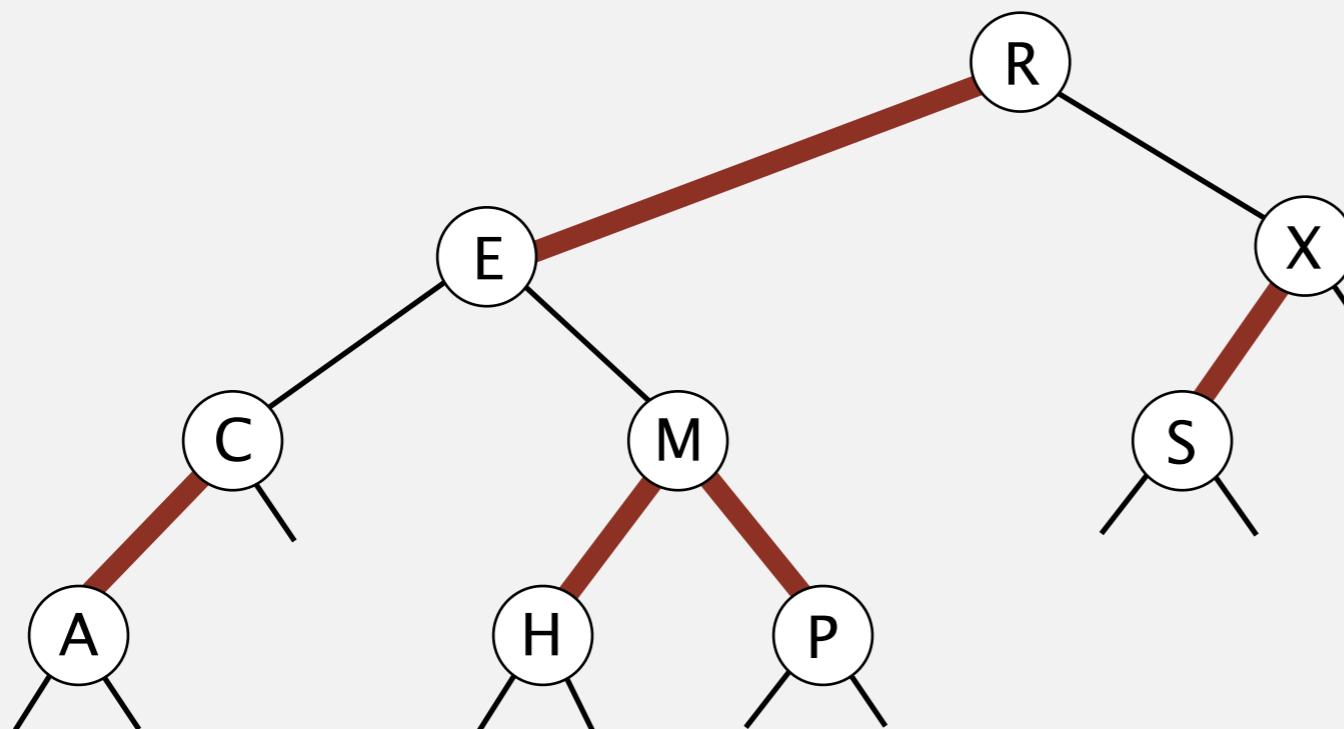
Red-black BST insertion

red-black BST



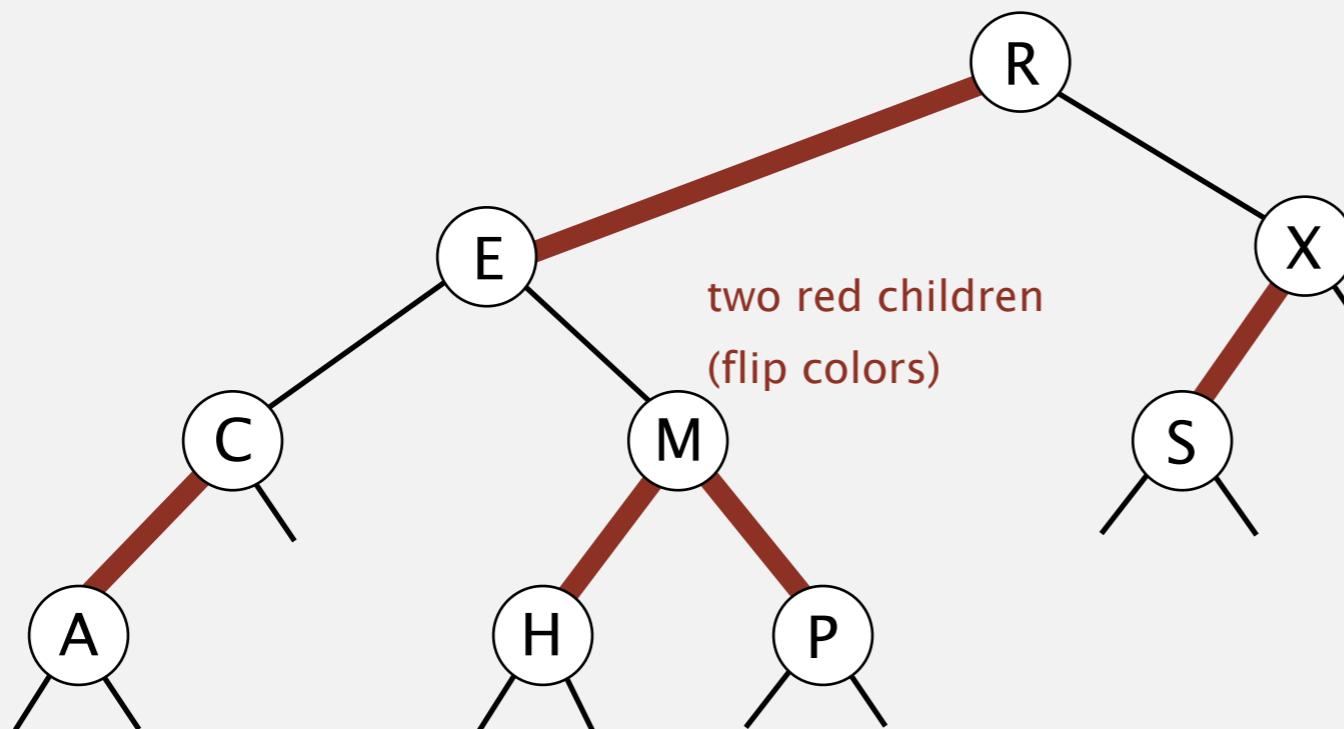
Red-black BST insertion

insert P



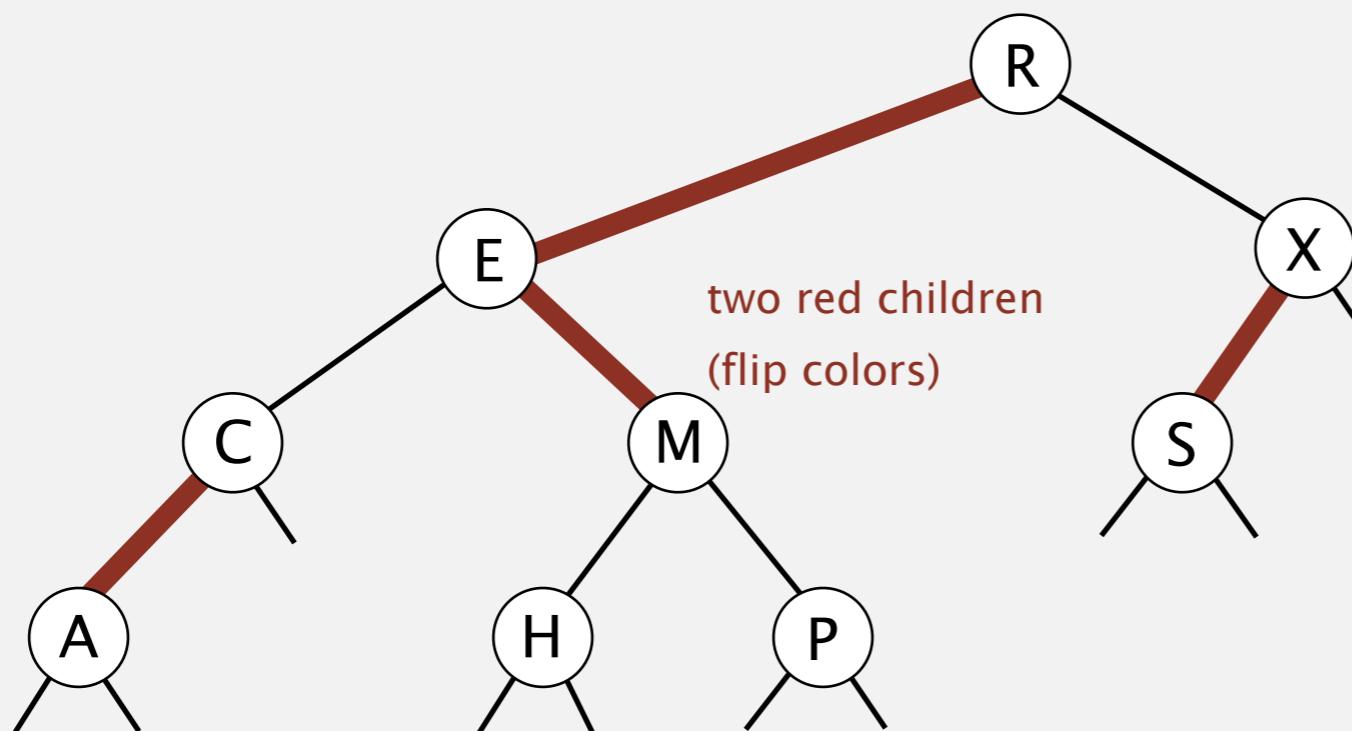
Red-black BST insertion

insert P

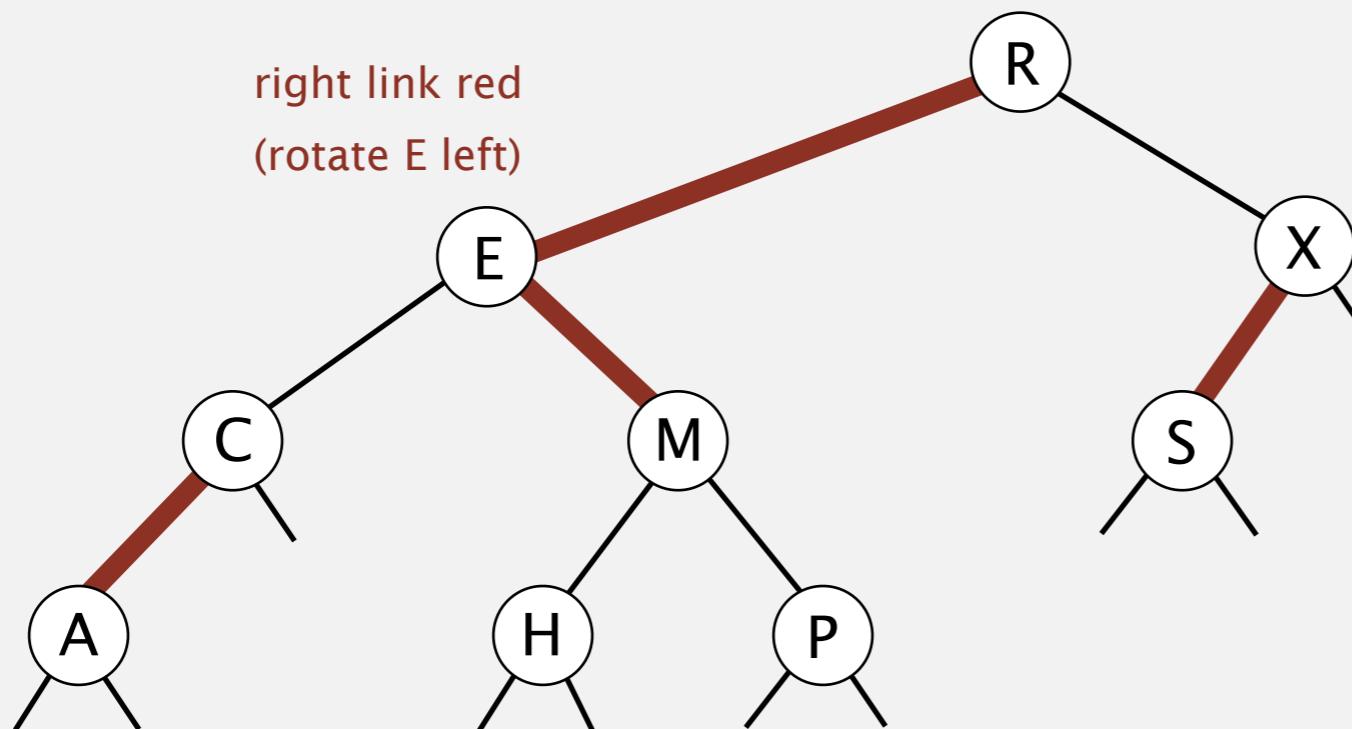


Red-black BST insertion

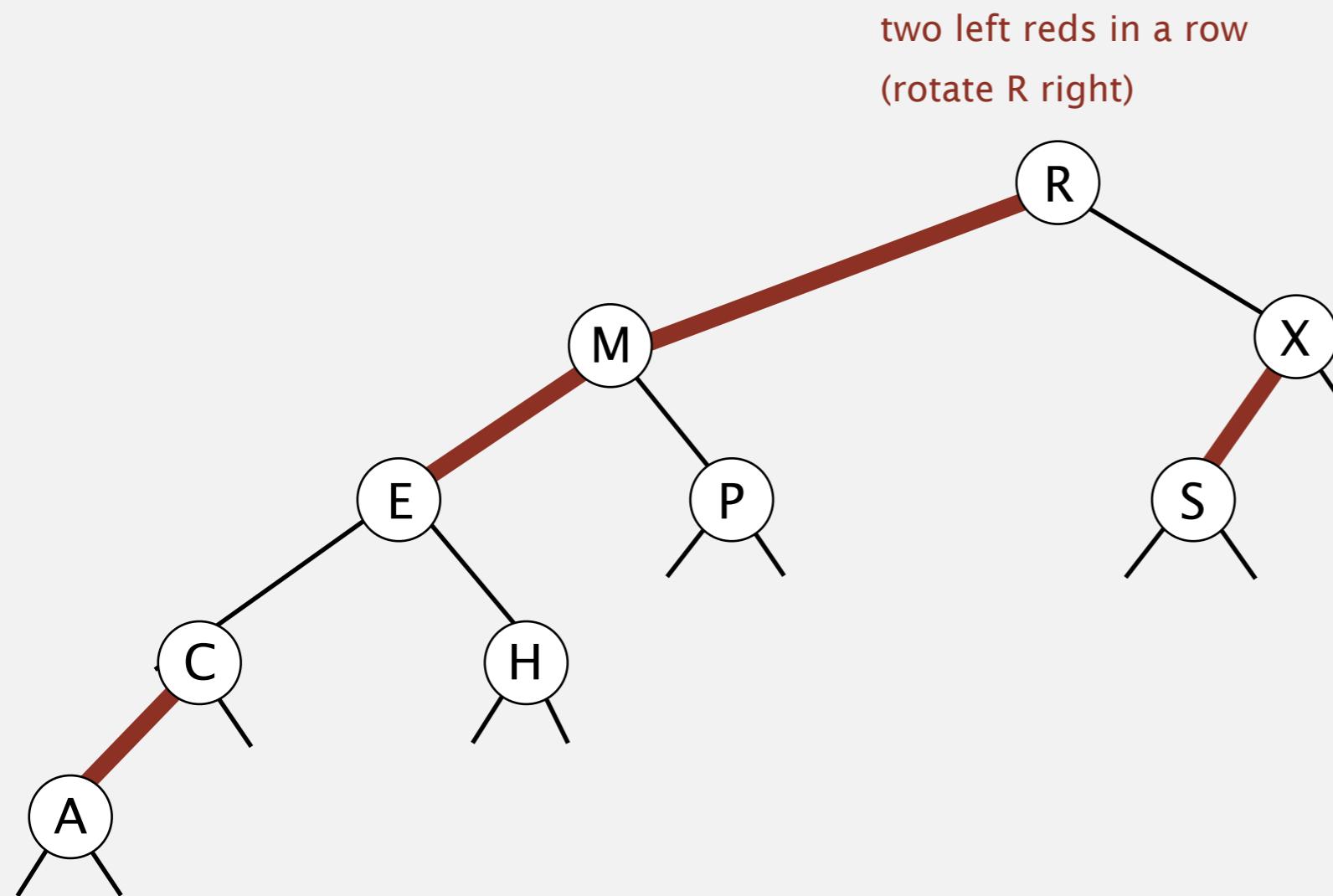
insert P



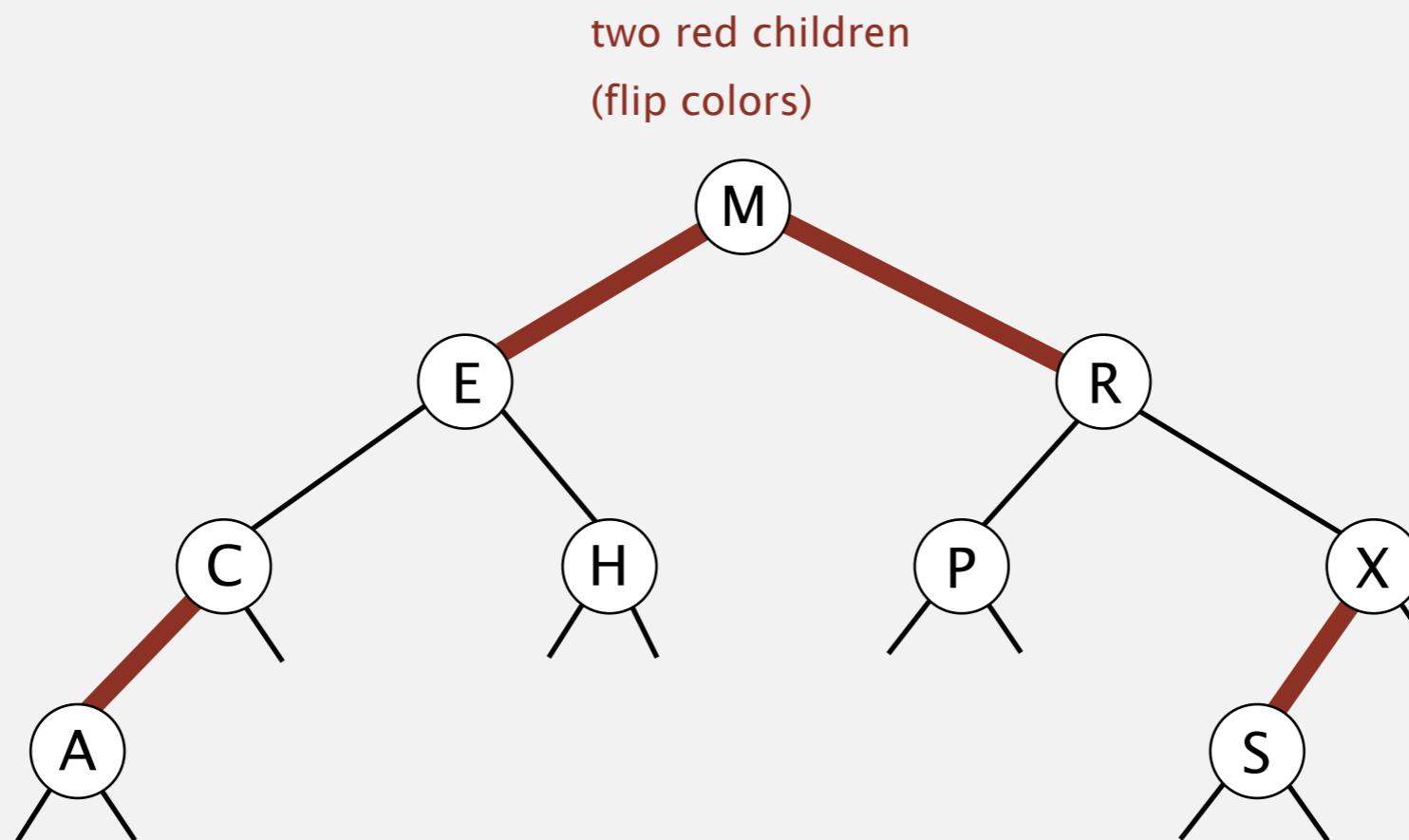
Red-black BST insertion



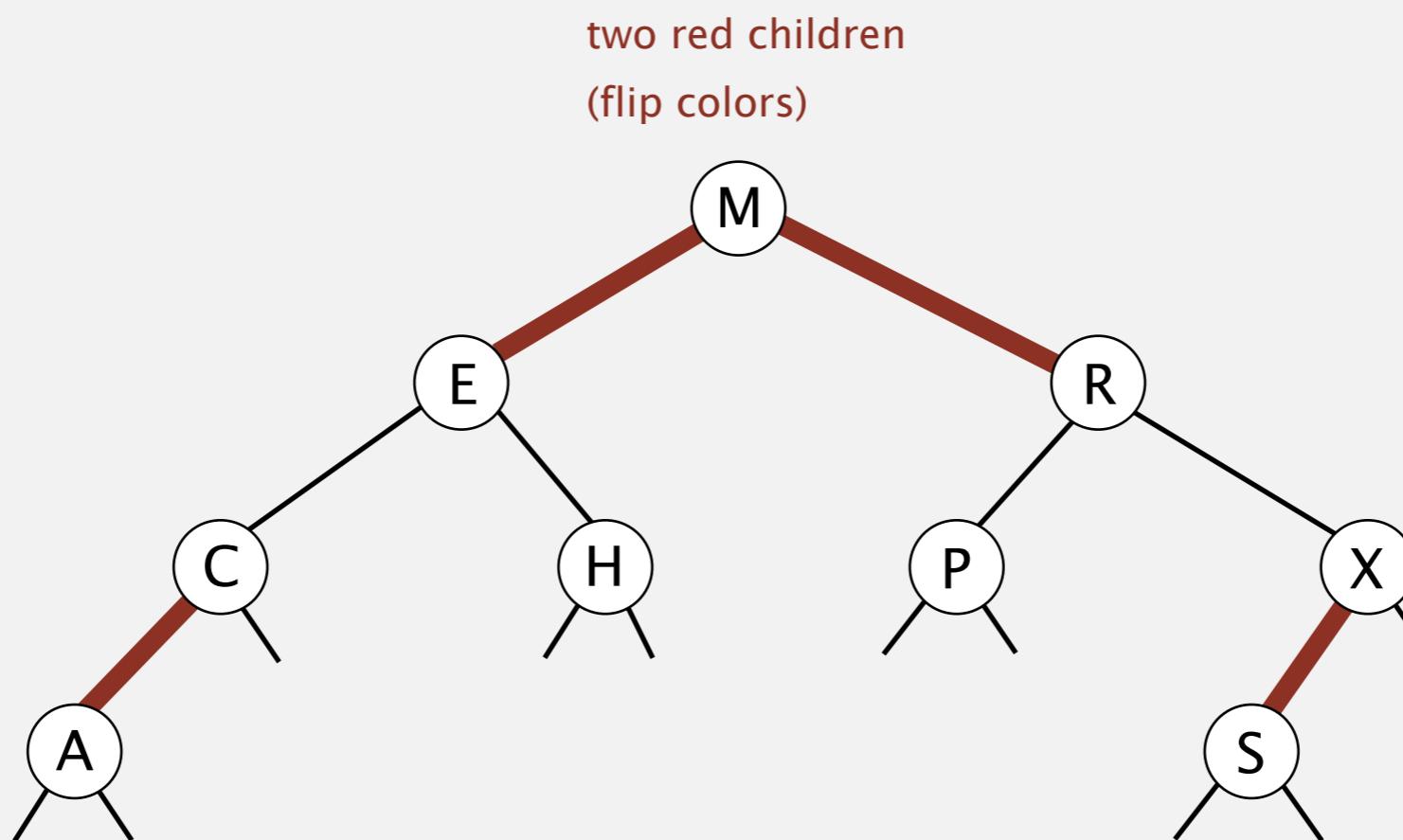
Red-black BST insertion



Red-black BST insertion

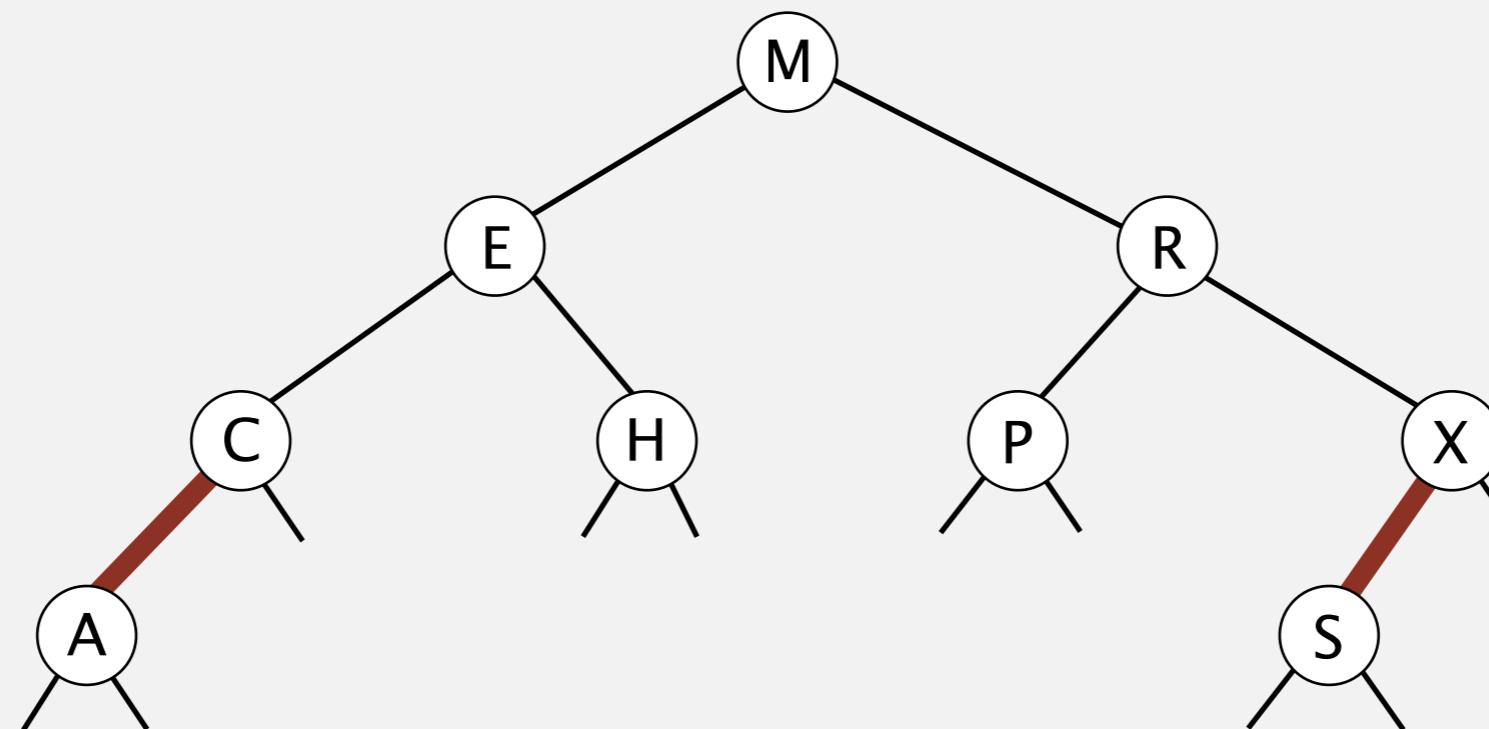


Red-black BST insertion



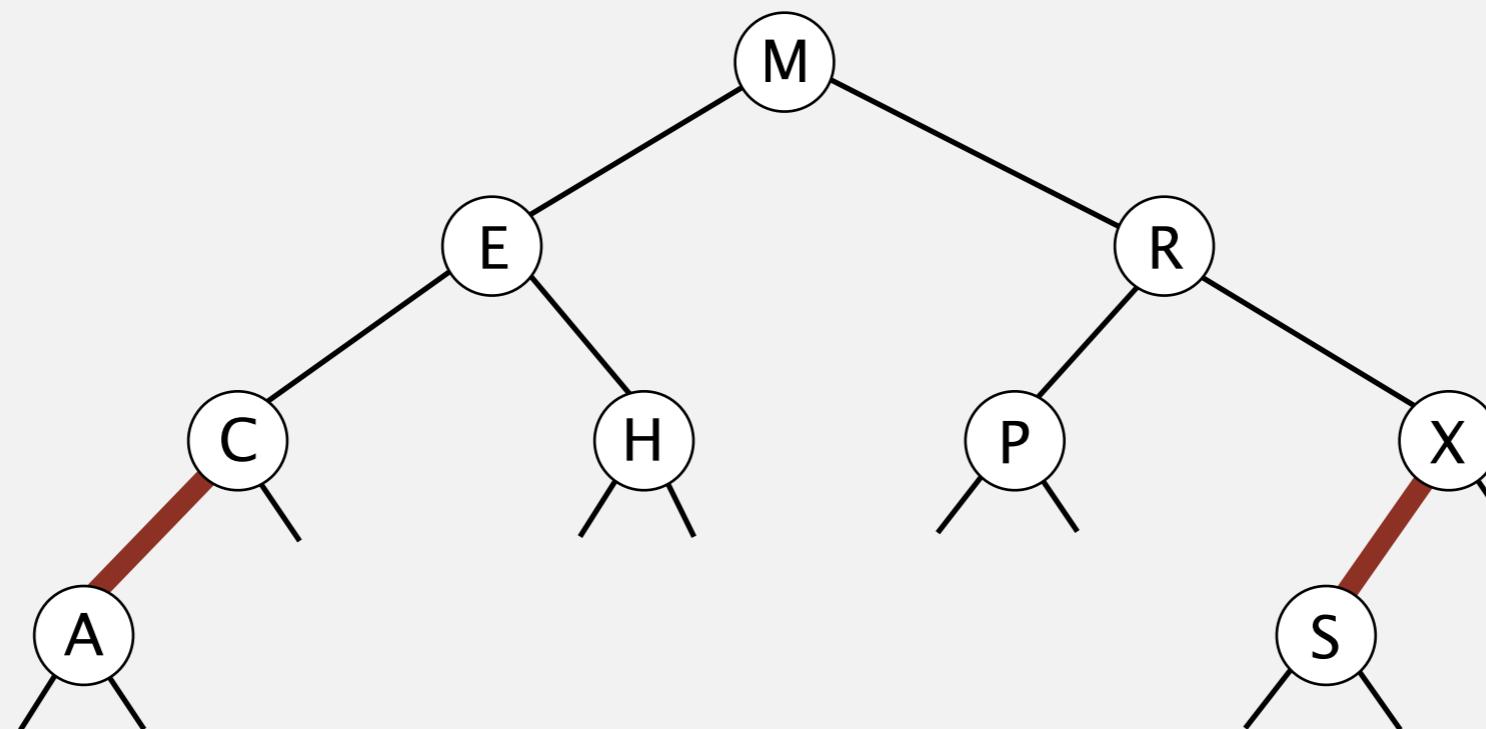
Red-black BST insertion

red-black BST



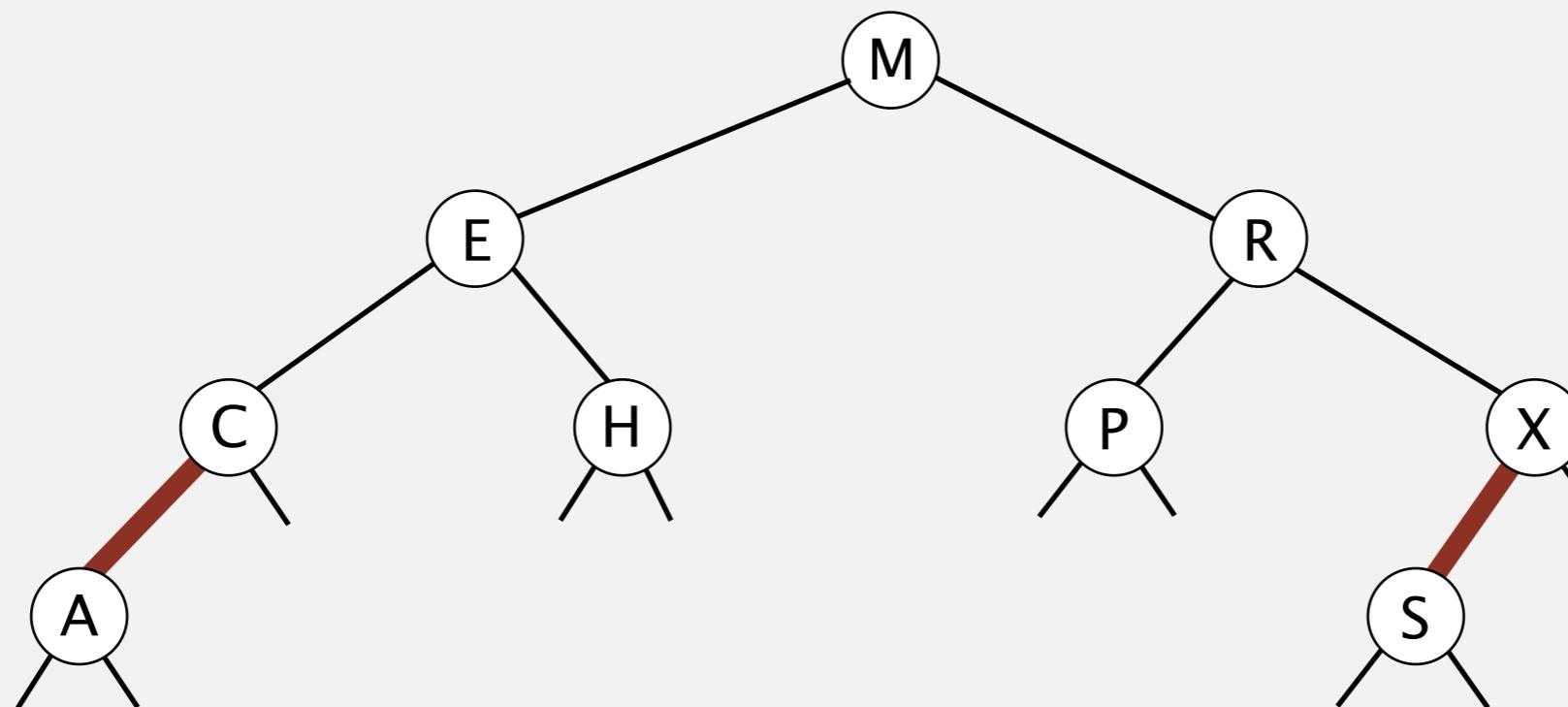
Red-black BST insertion

red-black BST



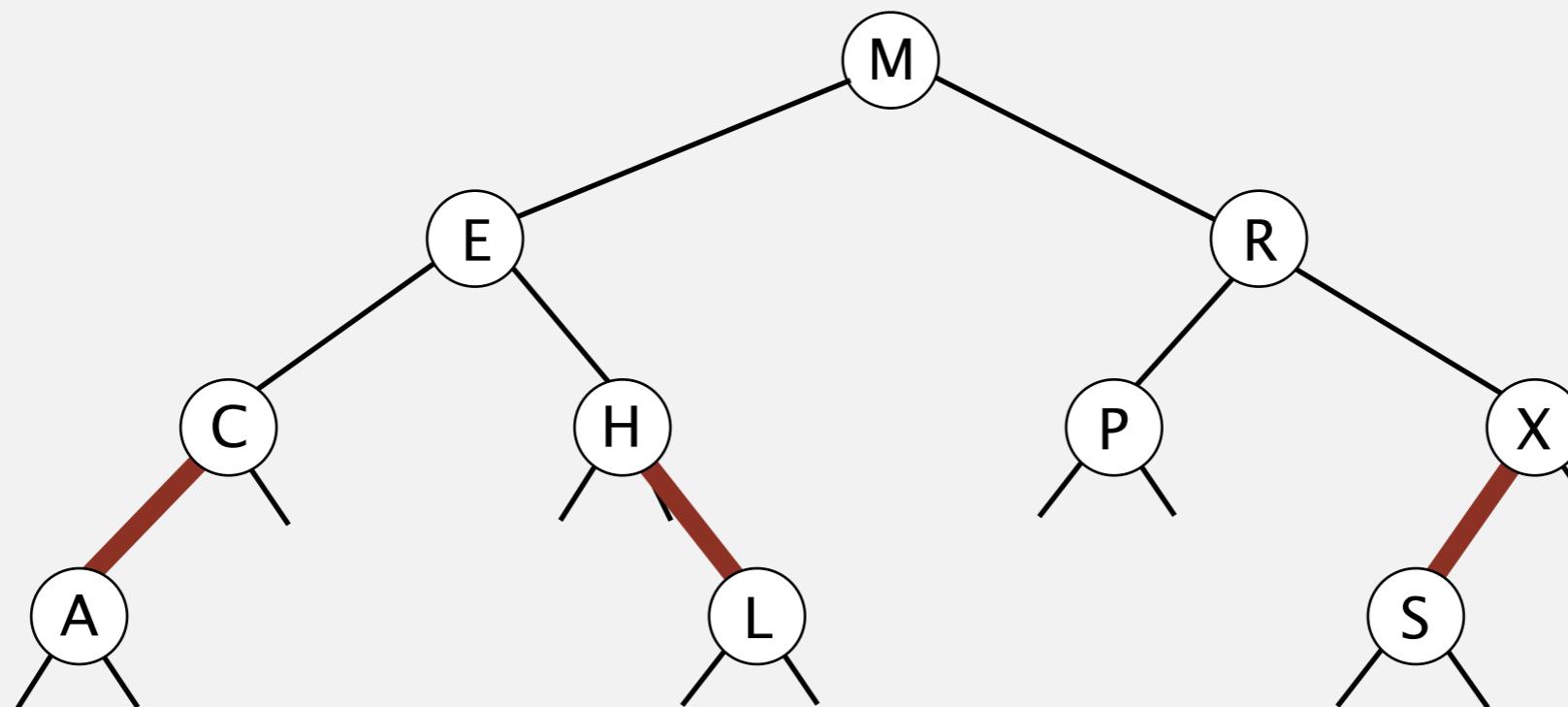
Red-black BST insertion

red-black BST



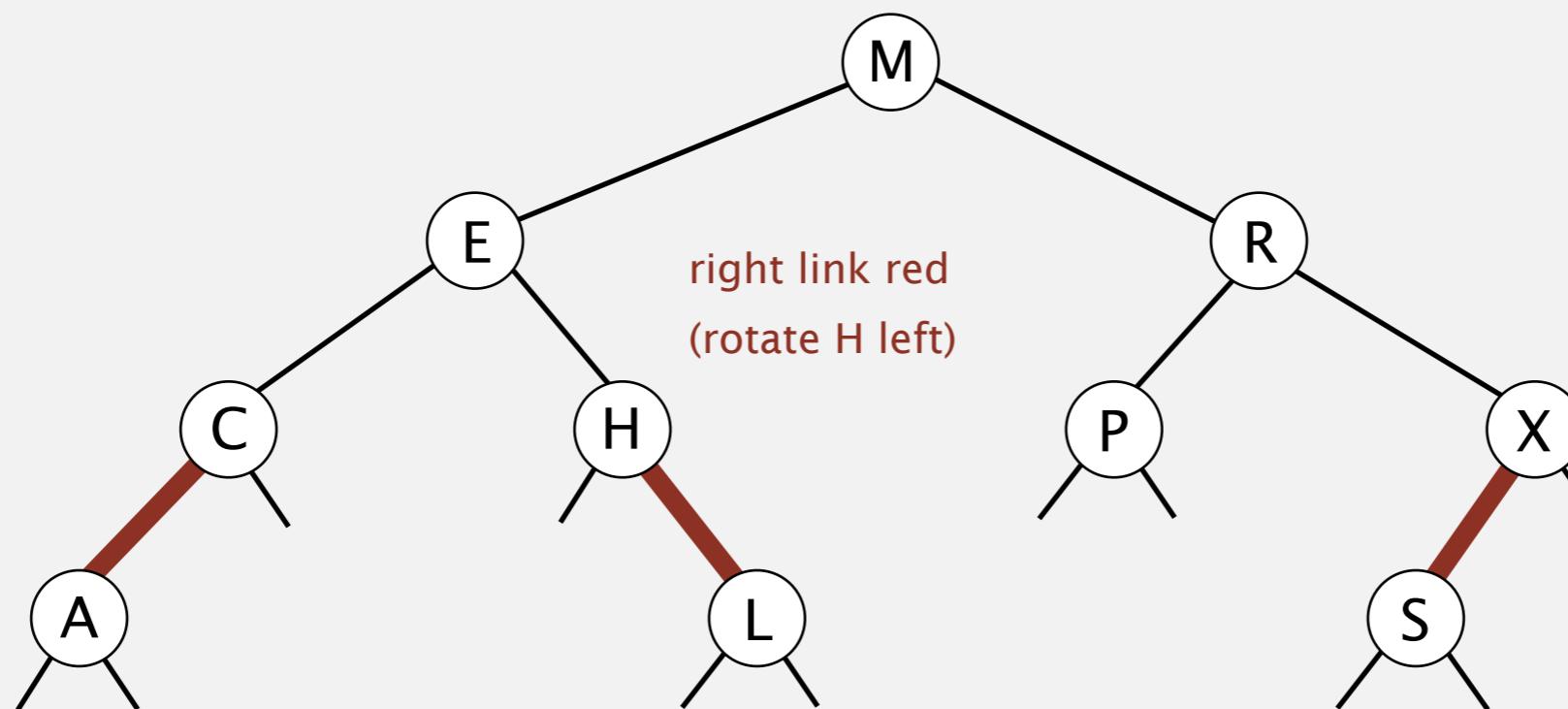
Red-black BST insertion

insert L



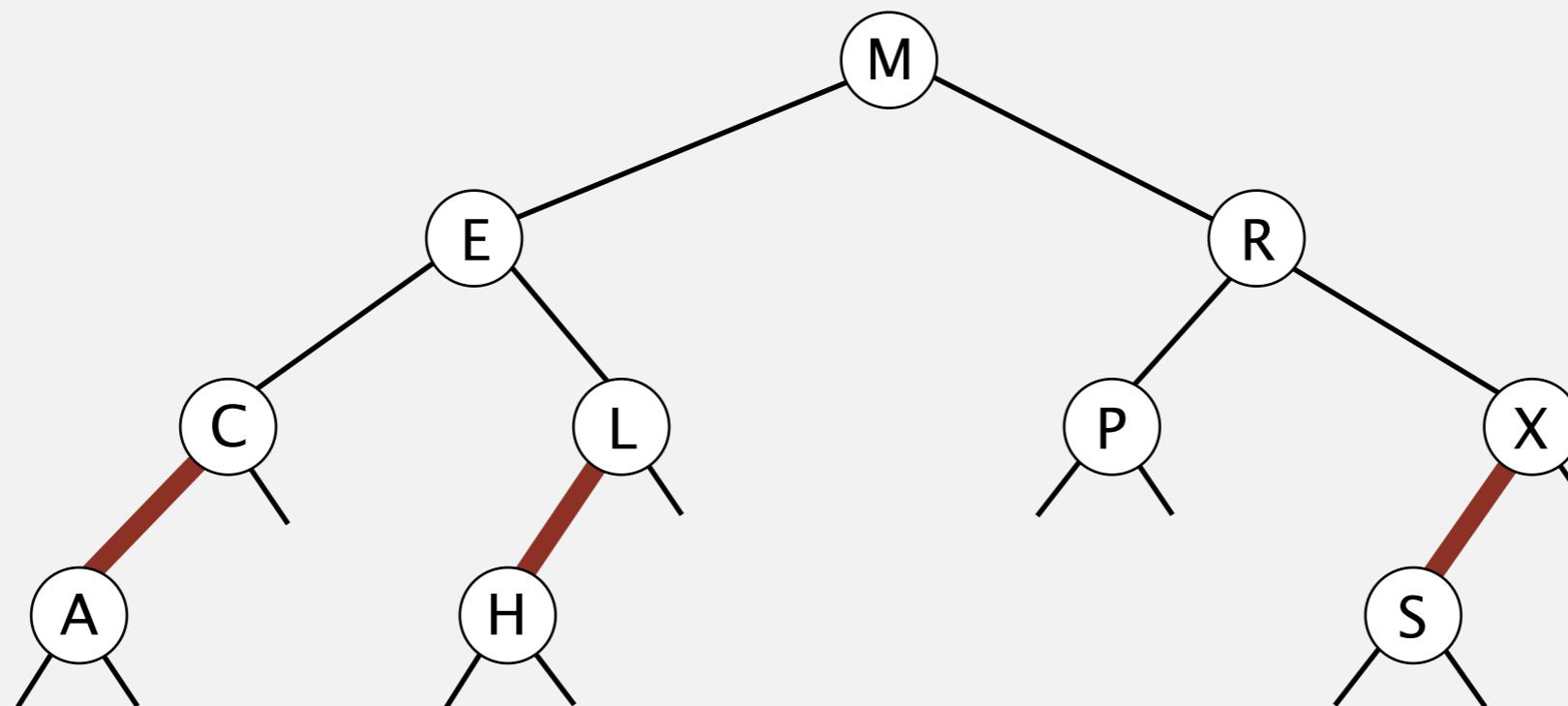
Red-black BST insertion

insert L



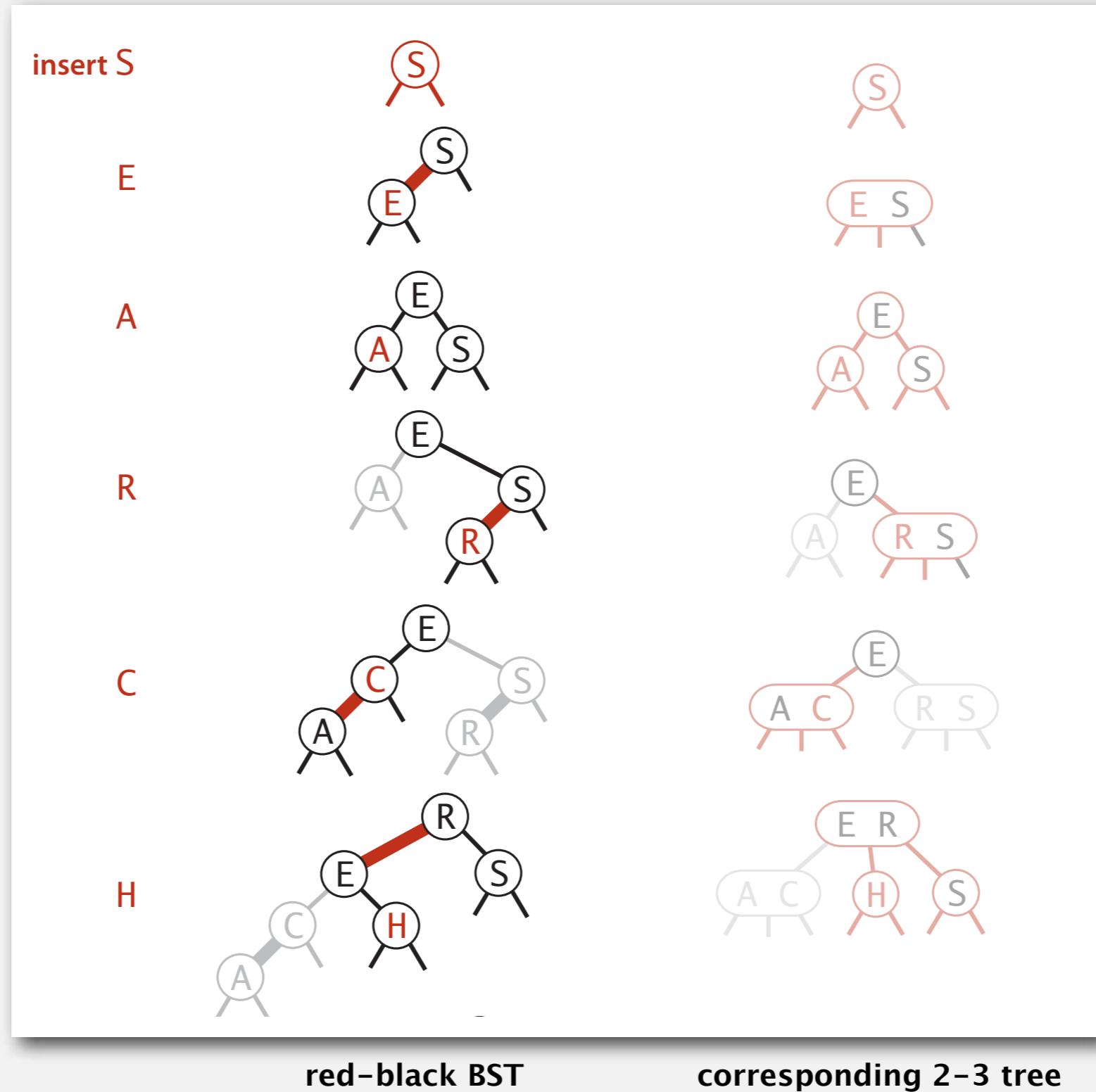
Red-black BST insertion

red-black BST



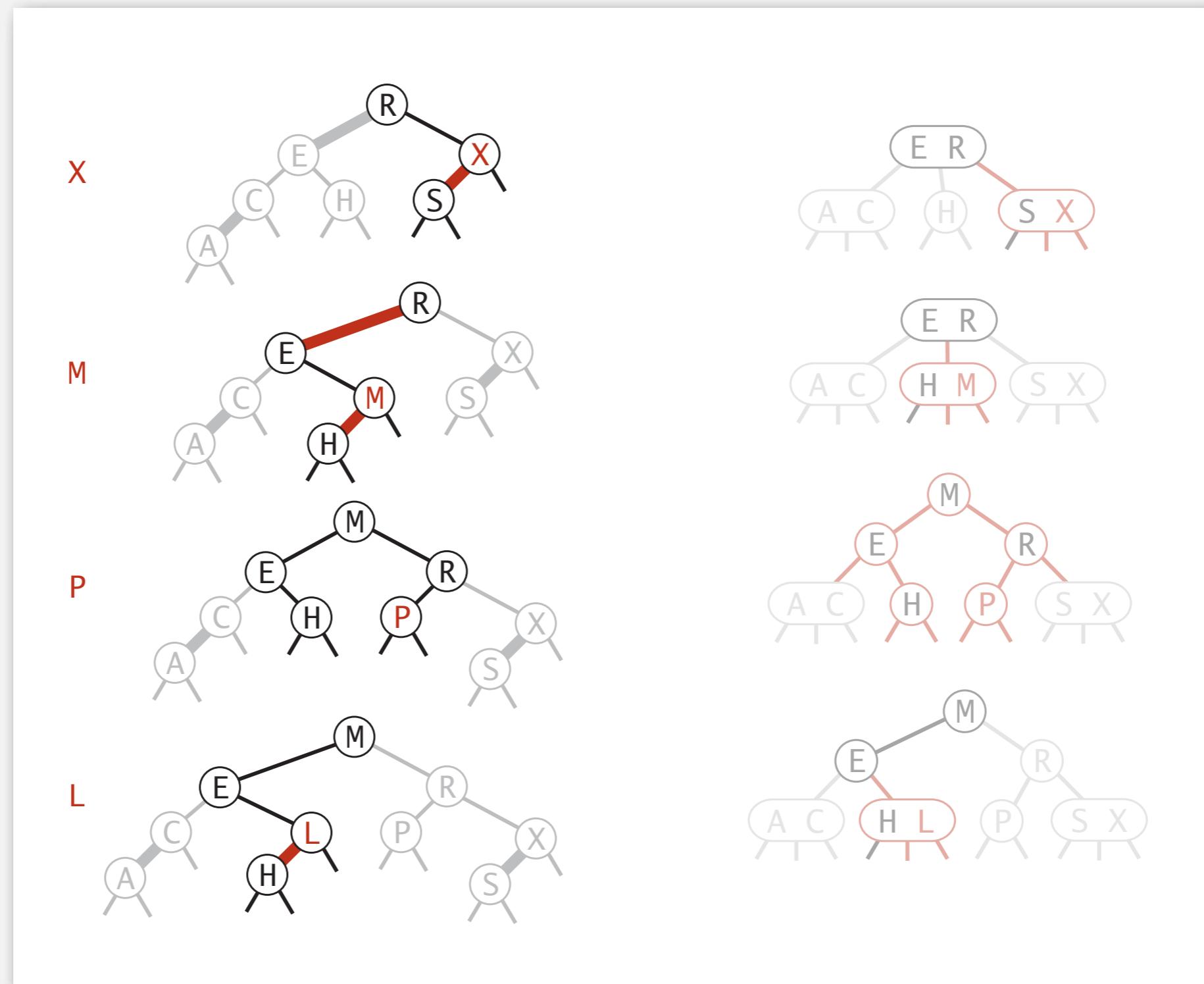
LLRB tree insertion trace

Standard indexing client.



LLRB tree insertion trace

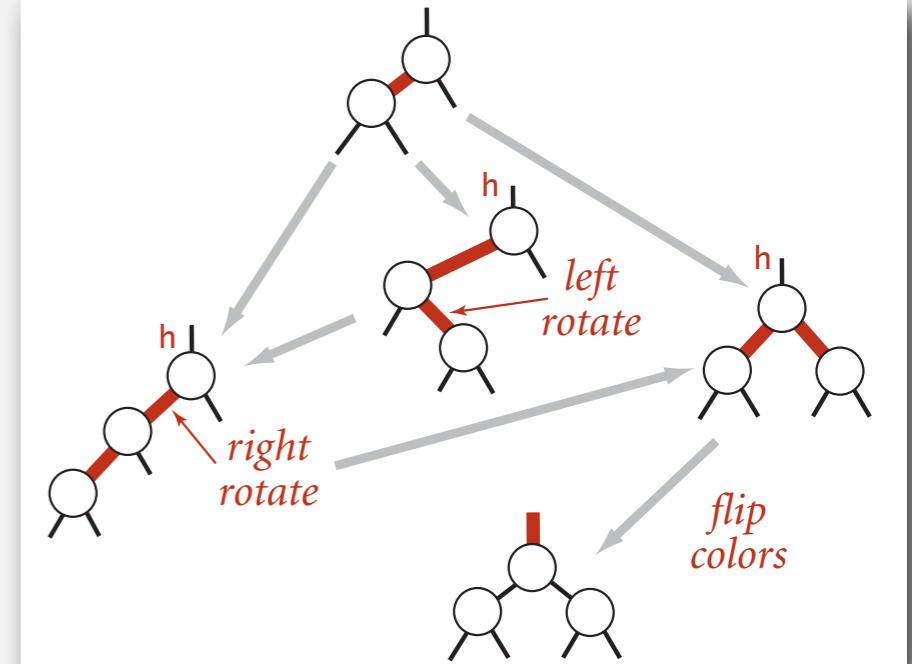
Standard indexing client (continued).



Insertion in a LLRB tree: Java implementation

Same code for both cases.

- Right child red, left child black: **rotate left**.
- Left child, left-left grandchild red: **rotate right**.
- Both children red: **flip colors**.



```
private Node put(Node h, Key key, Value val)
{
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if (cmp < 0) h.left = put(h.left, key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else if (cmp == 0) h.val = val;

    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left))   h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))       flipColors(h);

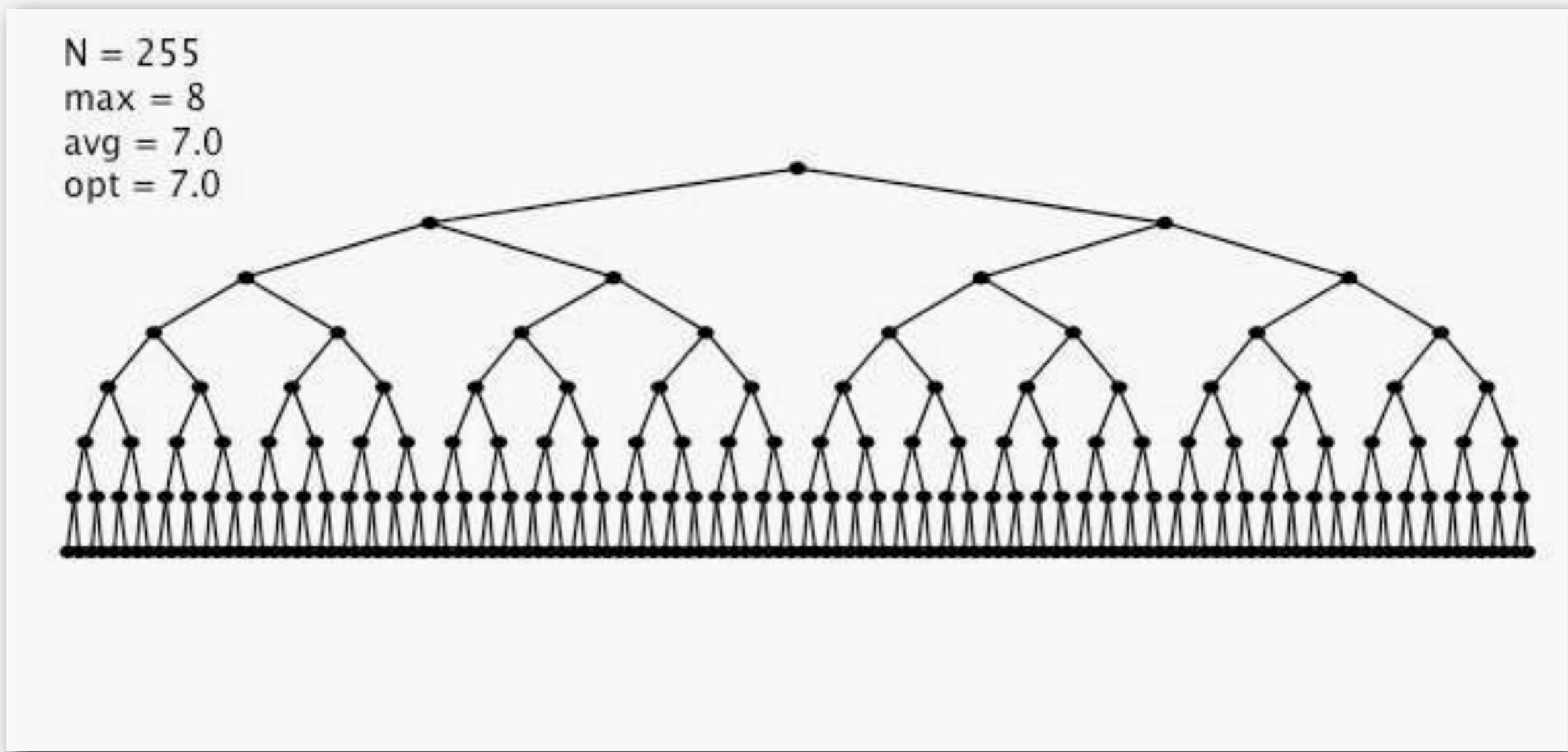
    return h;
}
```

only a few extra lines of code
provides near-perfect balance

insert at bottom
(and color red)

lean left
balance 4-node
split 4-node

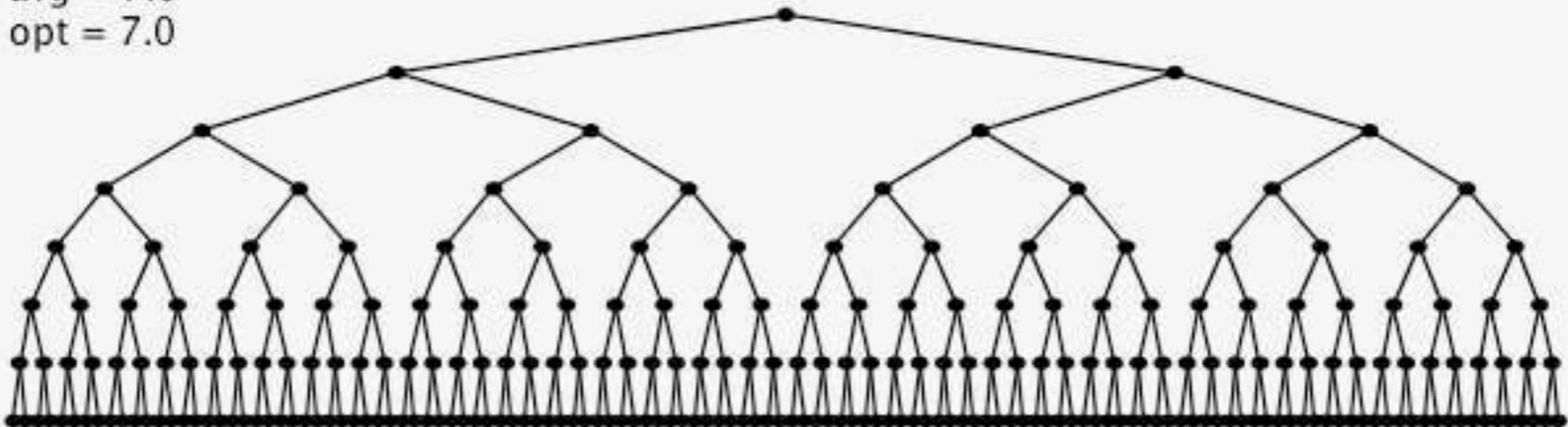
Insertion in a LLRB tree: visualization



Insertion in a LLRB tree: visualization

Remark. Only a few extra lines of code to standard BST insert.

N = 255
max = 8
avg = 7.0
opt = 7.0

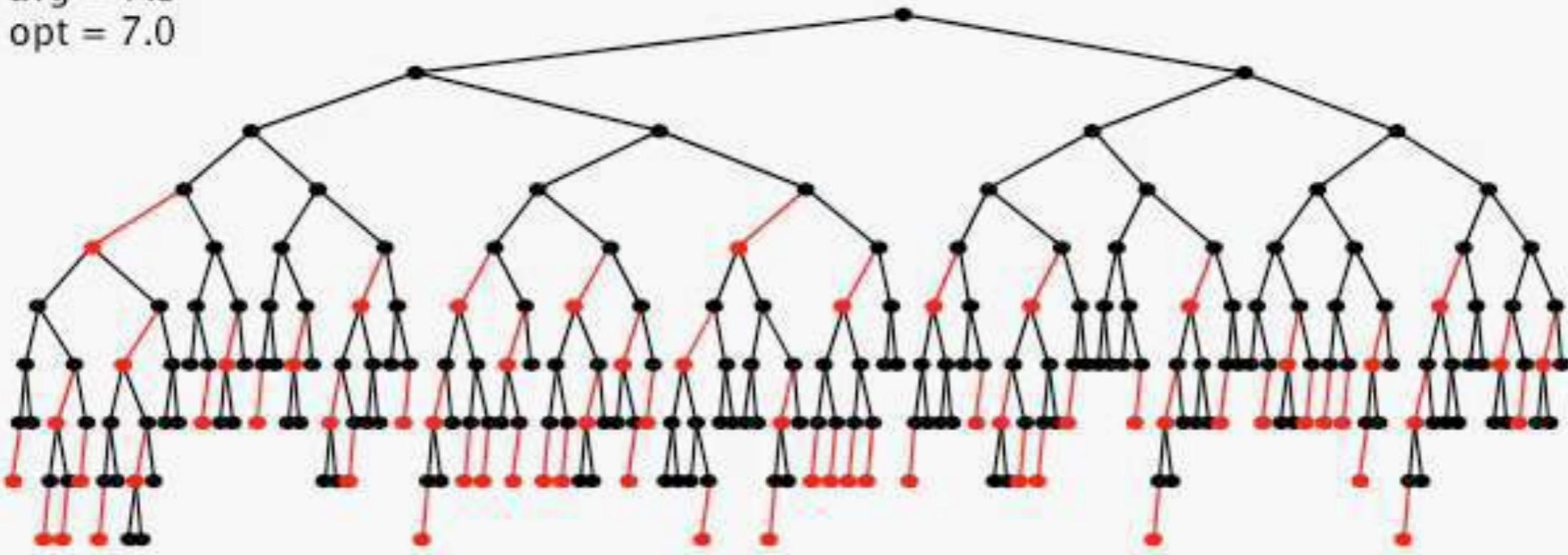


255 insertions in descending order

Insertion in a LLRB tree: visualization

Remark. Only a few extra lines of code to standard BST insert.

N = 255
max = 10
avg = 7.3
opt = 7.0



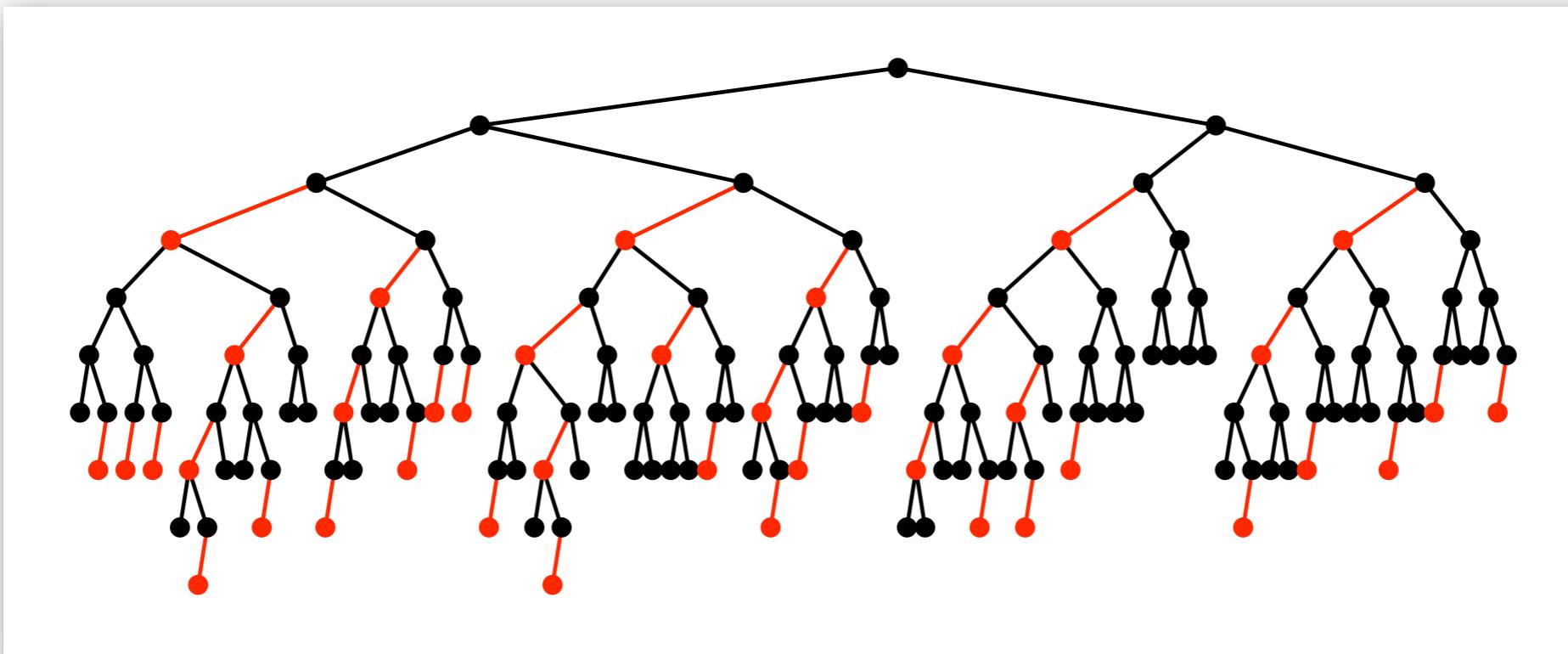
255 random insertions

Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

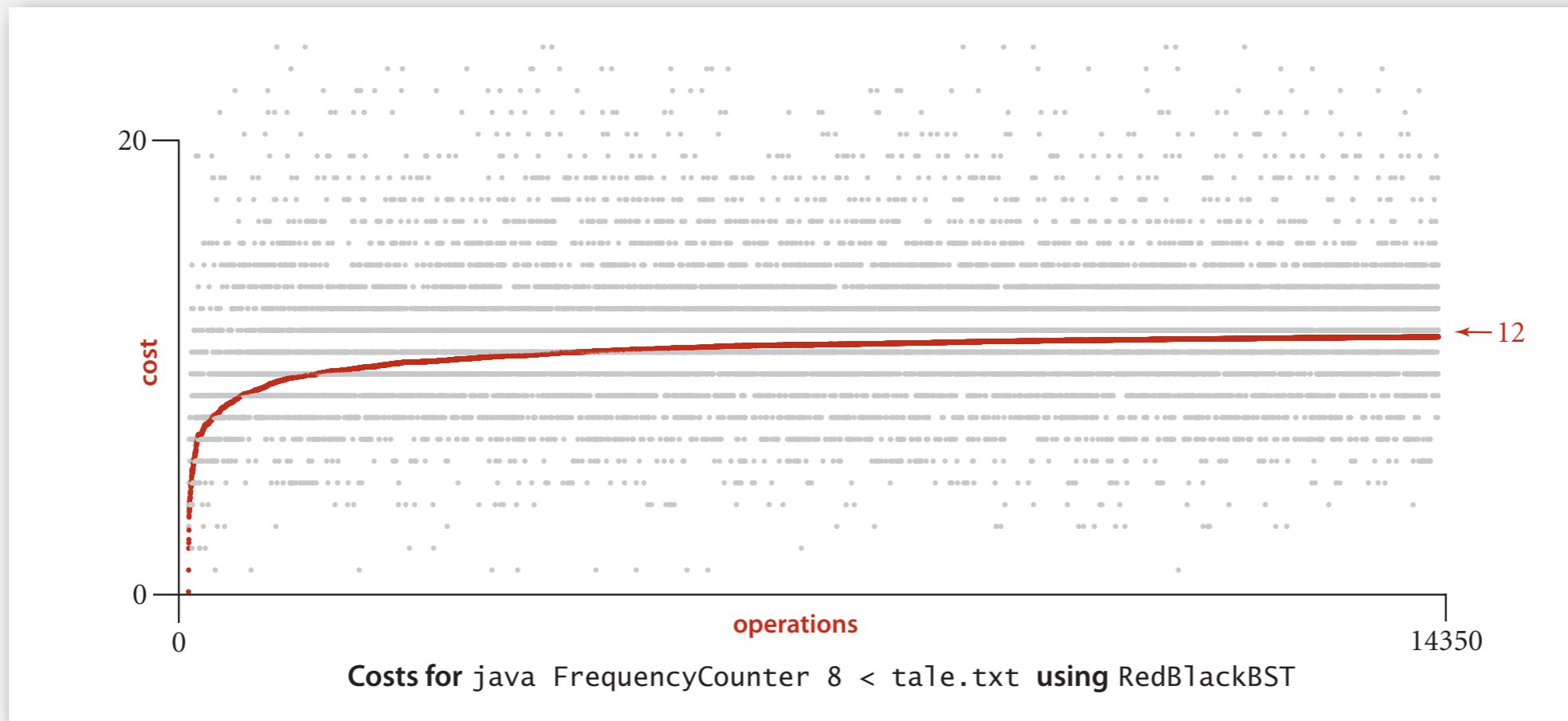
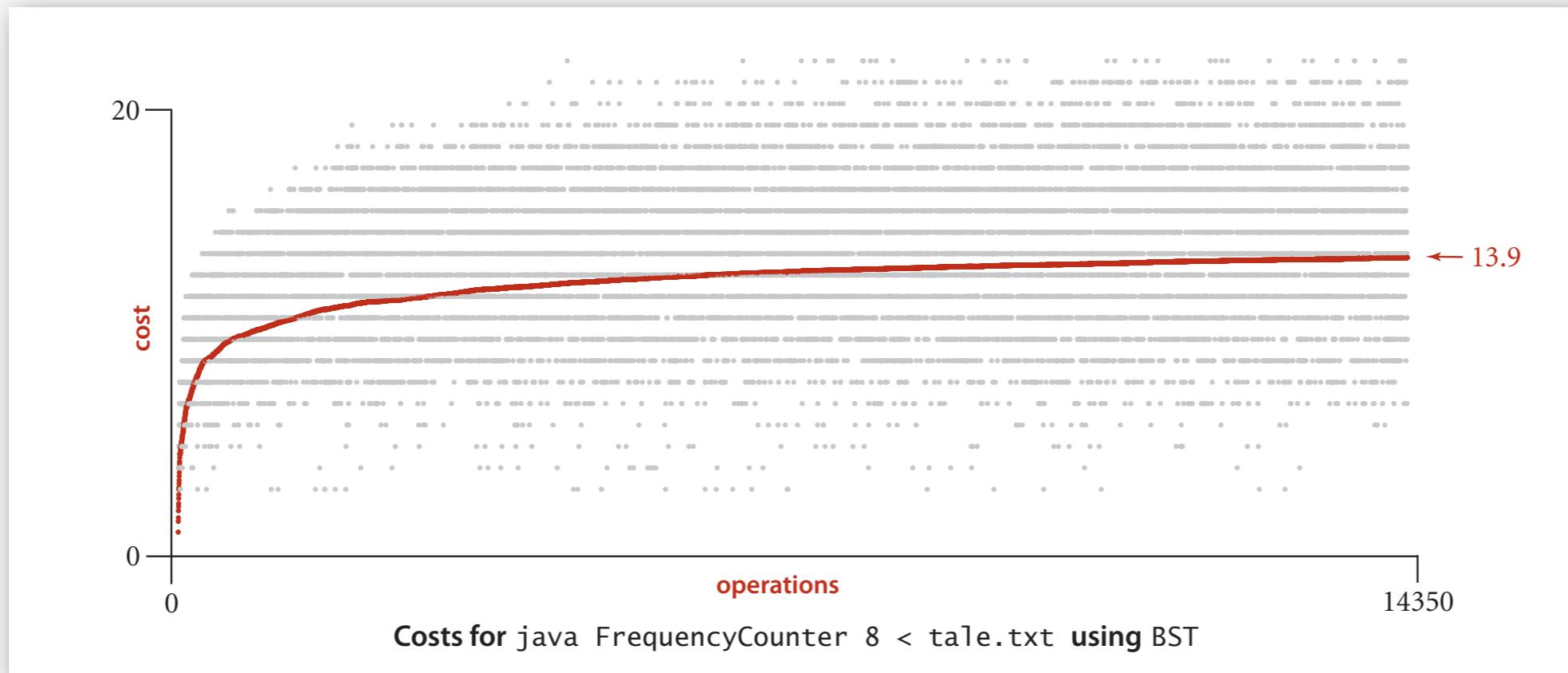
Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



Property. Height of tree is $\sim 1.00 \lg N$ in typical applications.

ST implementations: frequency counter



ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$?	yes	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	yes	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N^*$	$1.00 \lg N^*$	$1.00 \lg N^*$	yes	<code>compareTo()</code>

* exact value of coefficient unknown but extremely close to 1

BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ Red-black BSTs
- ▶ B-trees
- ▶ Geometric applications of BSTs

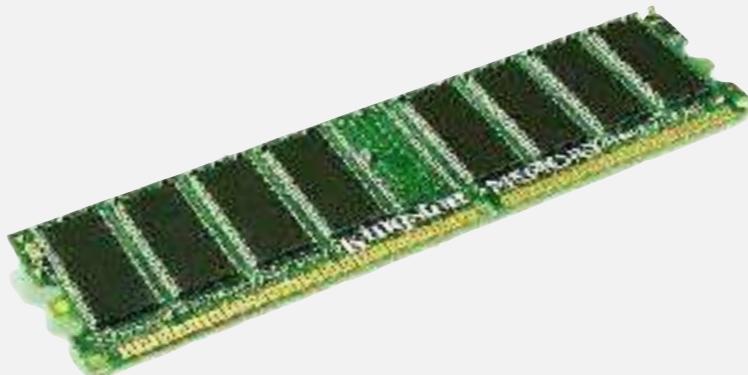
File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).



slow



fast

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

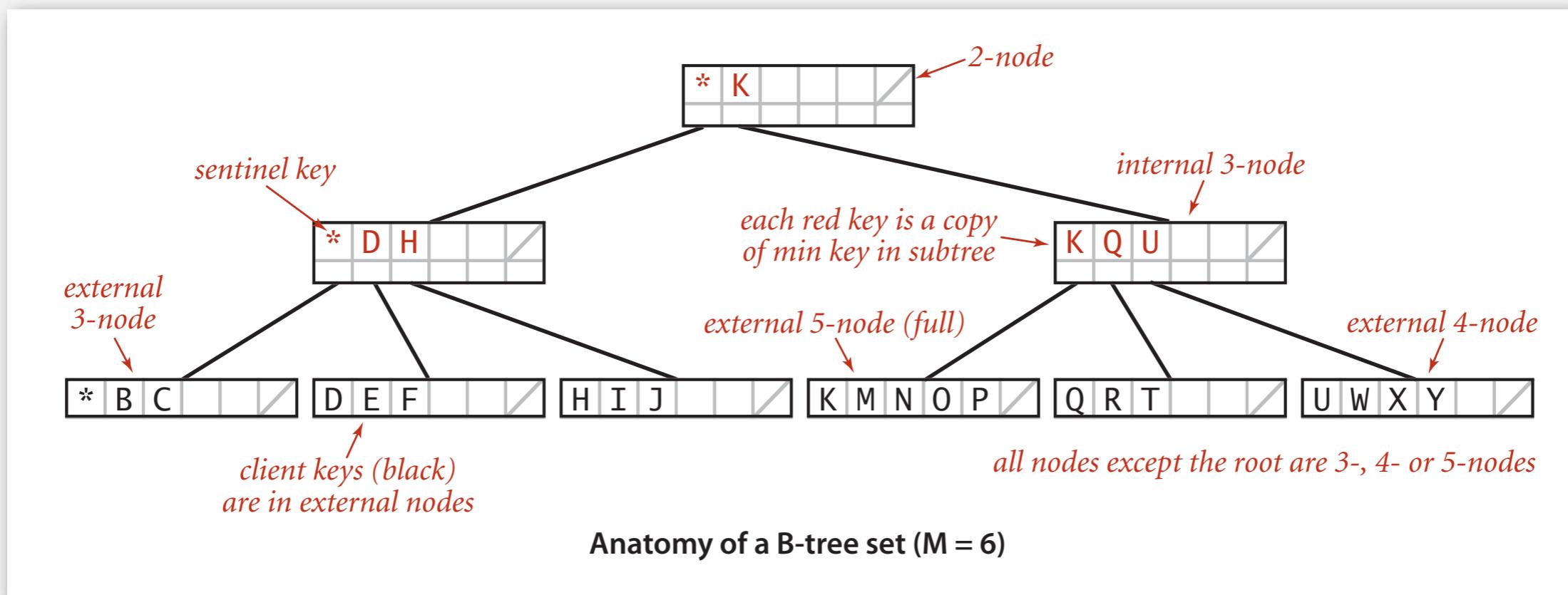
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

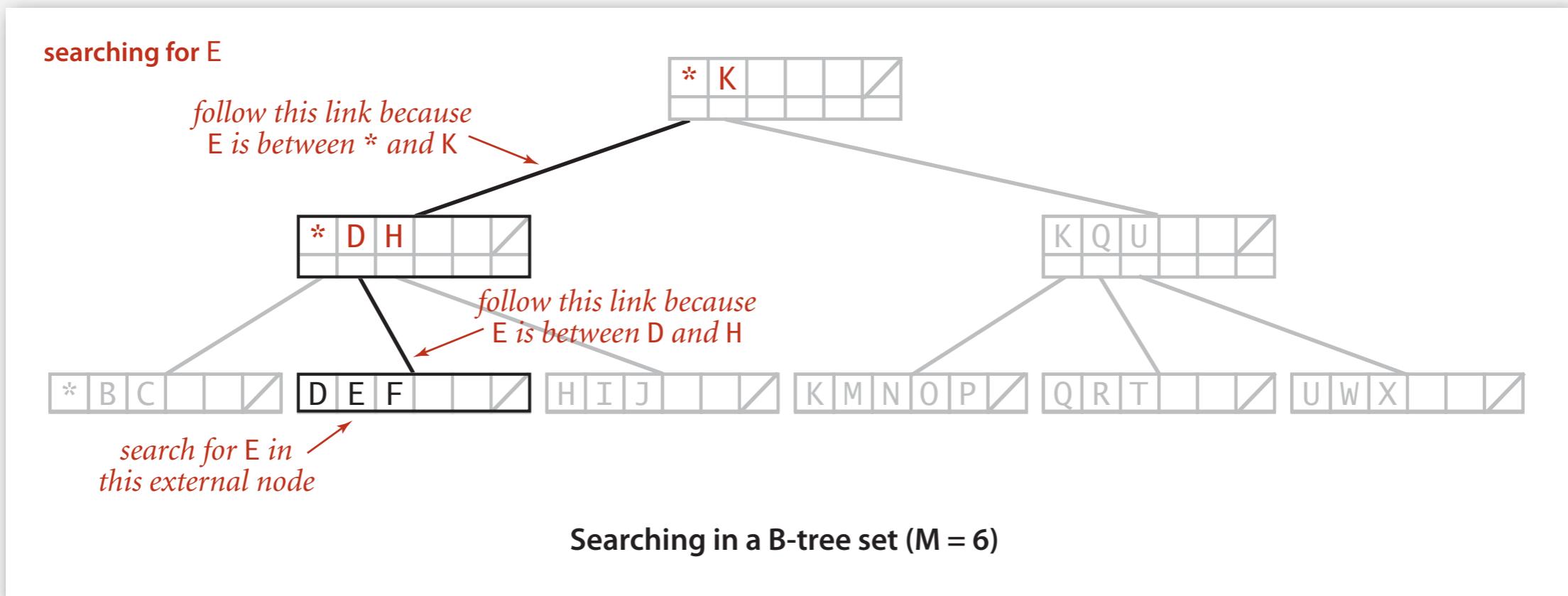
- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

choose M as large as possible so that M links fit in a page, e.g., $M = 1024$



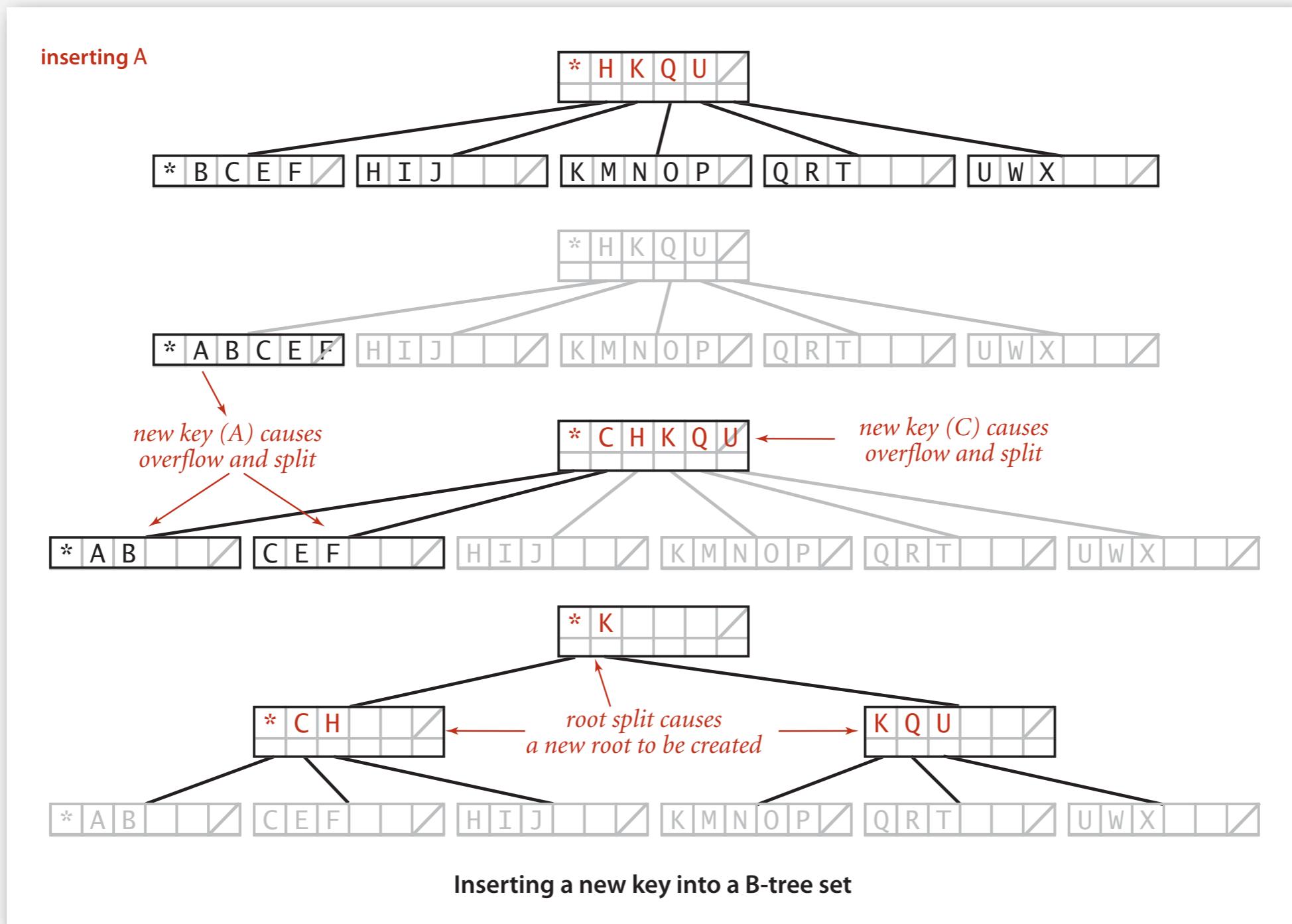
Searching in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Balance in B-tree

Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

Pf. All internal nodes (besides root) have between $M / 2$ and $M - 1$ links.

In practice. Number of probes is at most 4. \leftarrow
 $M = 1024$; $N = 62$ billion
 $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Building a large B tree



Balanced trees in the wild

Red-black trees are widely used as system symbol tables.

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: `map`, `multimap`, `multiset`.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.

B-tree variants. B+ tree, B*tree, B# tree, ...

B-trees (and variants) are widely used for file systems and databases.

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ Red-black BSTs
- ▶ B-trees
- ▶ Geometric applications of BSTs

GEOMETRIC APPLICATIONS OF BSTs

- ▶ kd trees

2-d orthogonal range search

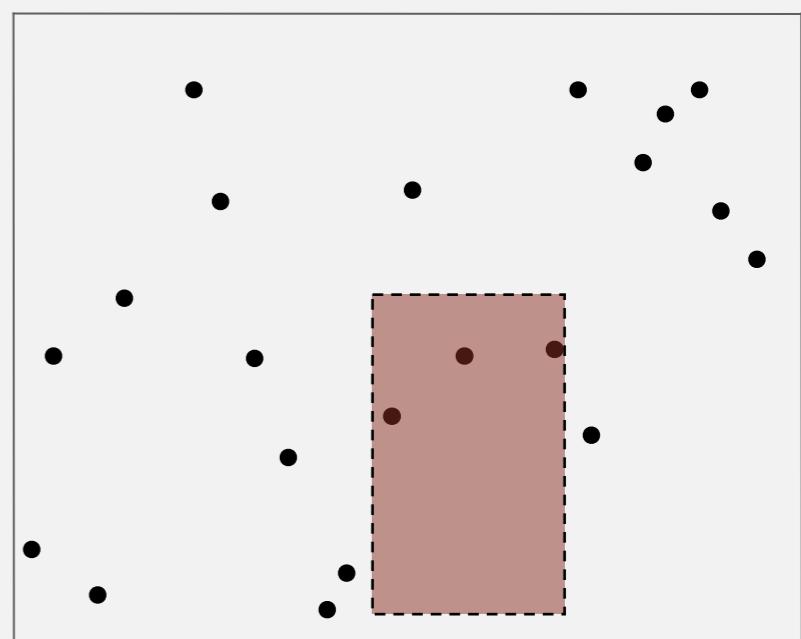
Extension of ordered symbol-table to 2d keys.

- Insert a 2d key.
- Delete a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range.
- Range count: number of keys that lie in a 2d range.

Geometric interpretation.

- Keys are point in the plane.
- Find/count points in a given *h-v rectangle*.

↑
rectangle is axis-aligned

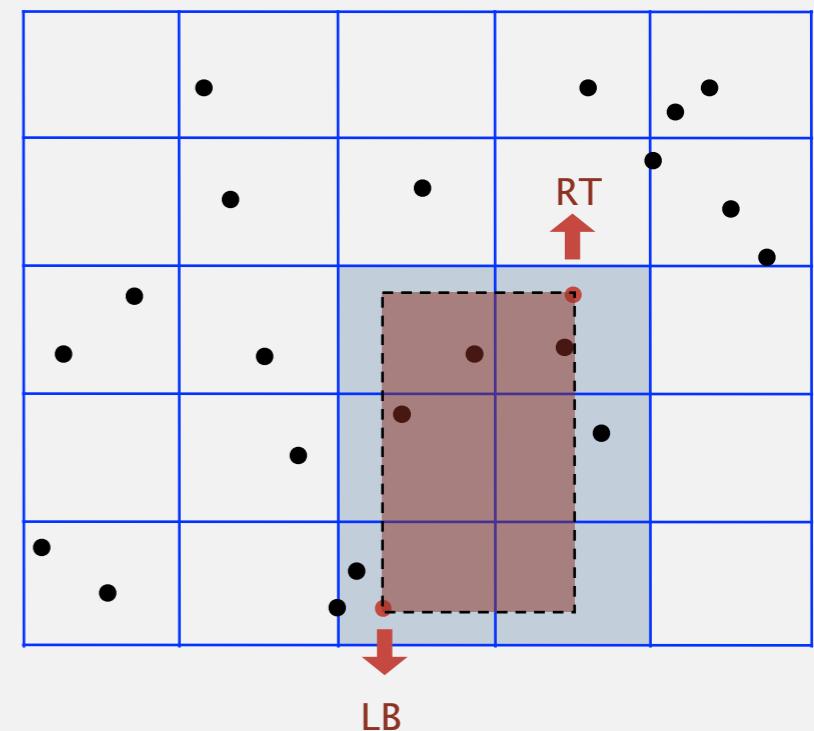


Applications. Networking, circuit design, databases,...

2d orthogonal range search: grid implementation

Grid implementation.

- Divide space into M -by- M grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add (x, y) to list for corresponding square.
- Range search: examine only those squares that intersect 2d range query.



2d orthogonal range search: grid implementation costs

Space-time tradeoff.

- Space: $M^2 + N$.
- Time: $1 + N/M^2$ per square examined, on average.

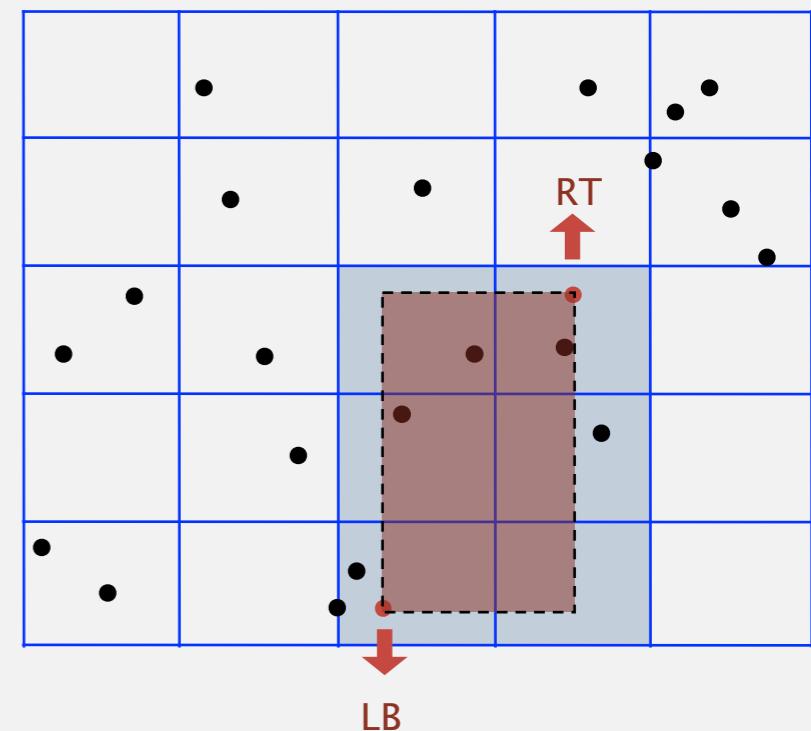
Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: \sqrt{N} -by- \sqrt{N} grid.

Running time. [if points are evenly distributed]

- Initialize data structure: N .
- Insert point: 1.
- Range search: 1 per point in range.

choose $M \sim \sqrt{N}$

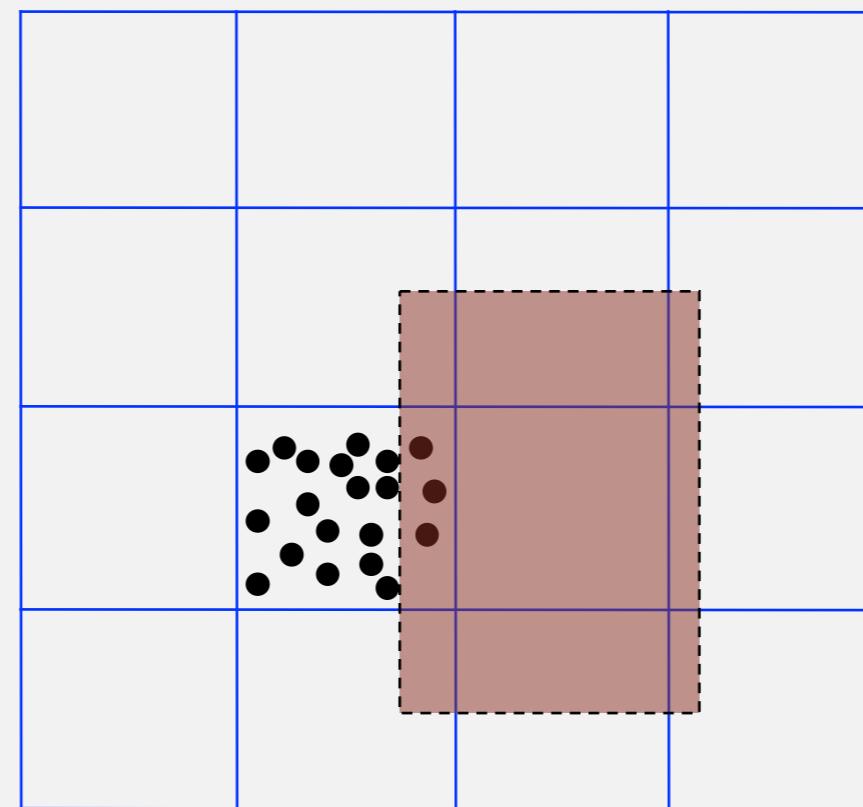


Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

- Lists are too long, even though average length is short.
- Need data structure that **gracefully** adapts to data.



Clustering

Grid implementation. Fast and simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.



13,000 points, 1000 grid squares



half the squares are empty



half the points are
in 10% of the squares

Space-partitioning trees

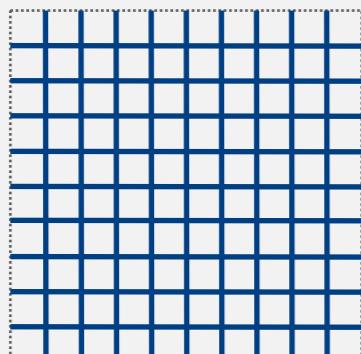
Use a **tree** to represent a recursive subdivision of 2d space.

Grid. Divide space uniformly into squares.

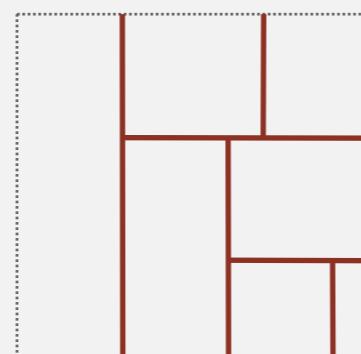
2d tree. Recursively divide space into two halfplanes.

Quadtree. Recursively divide space into four quadrants.

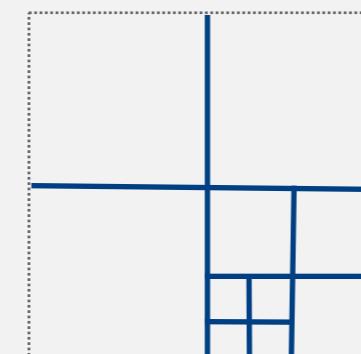
BSP tree. Recursively divide space into two regions.



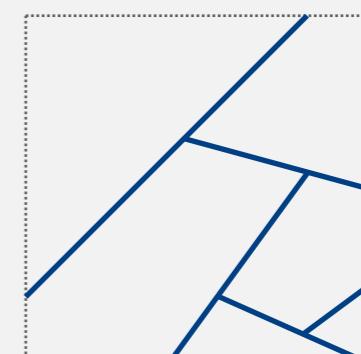
Grid



2d tree



Quadtrees

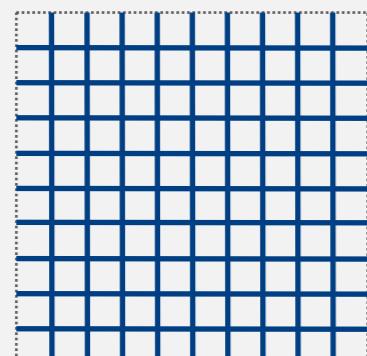


BSP tree

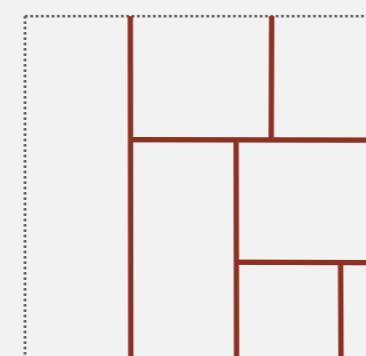
Space-partitioning trees: applications

Applications.

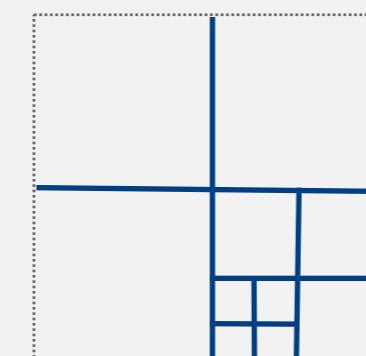
- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.



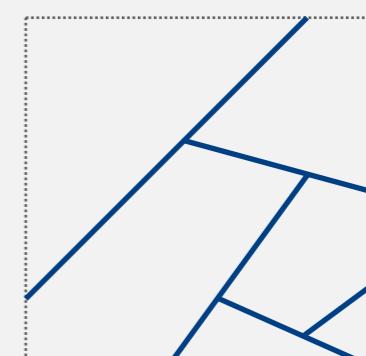
Grid



2d tree



Quadtree

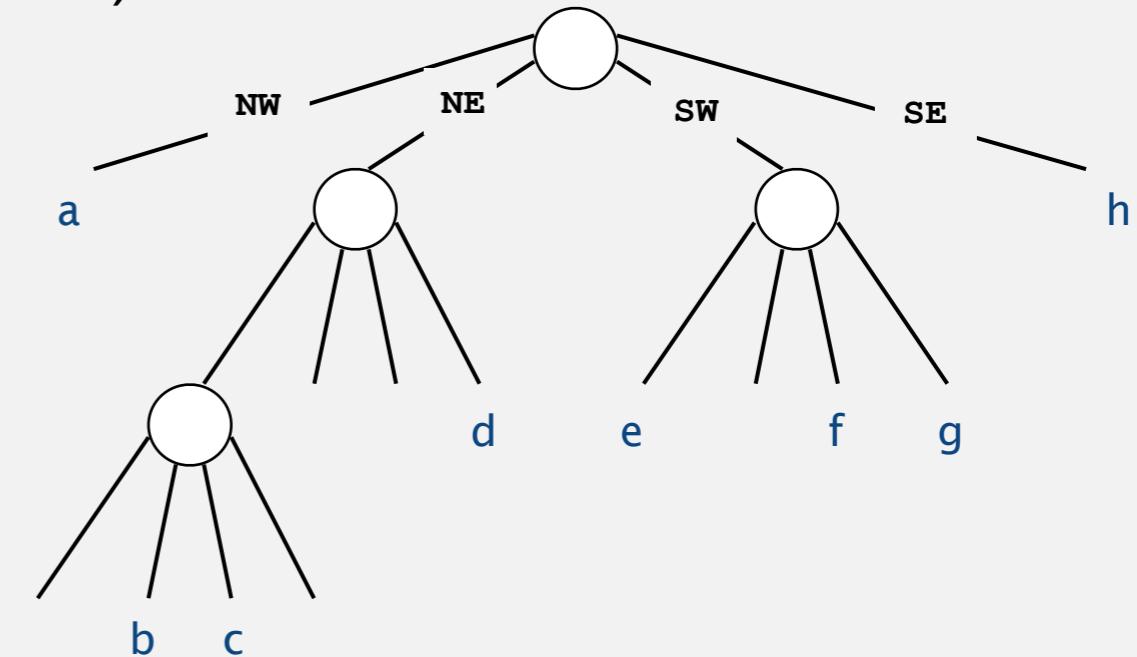
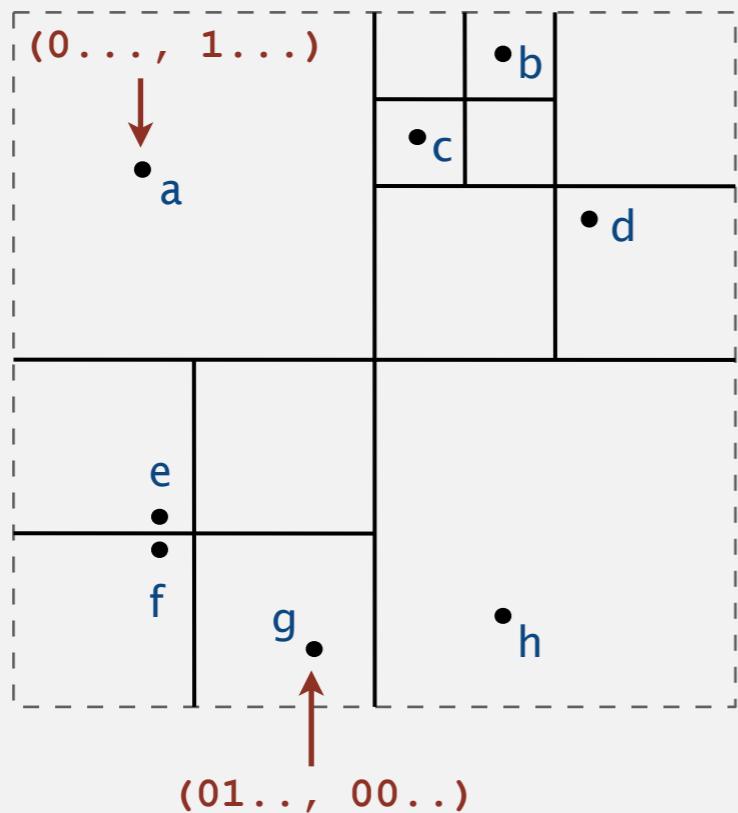


BSP tree

Quadtree

Idea. Recursively divide space into 4 quadrants.

Implementation. 4-way tree (actually a trie).

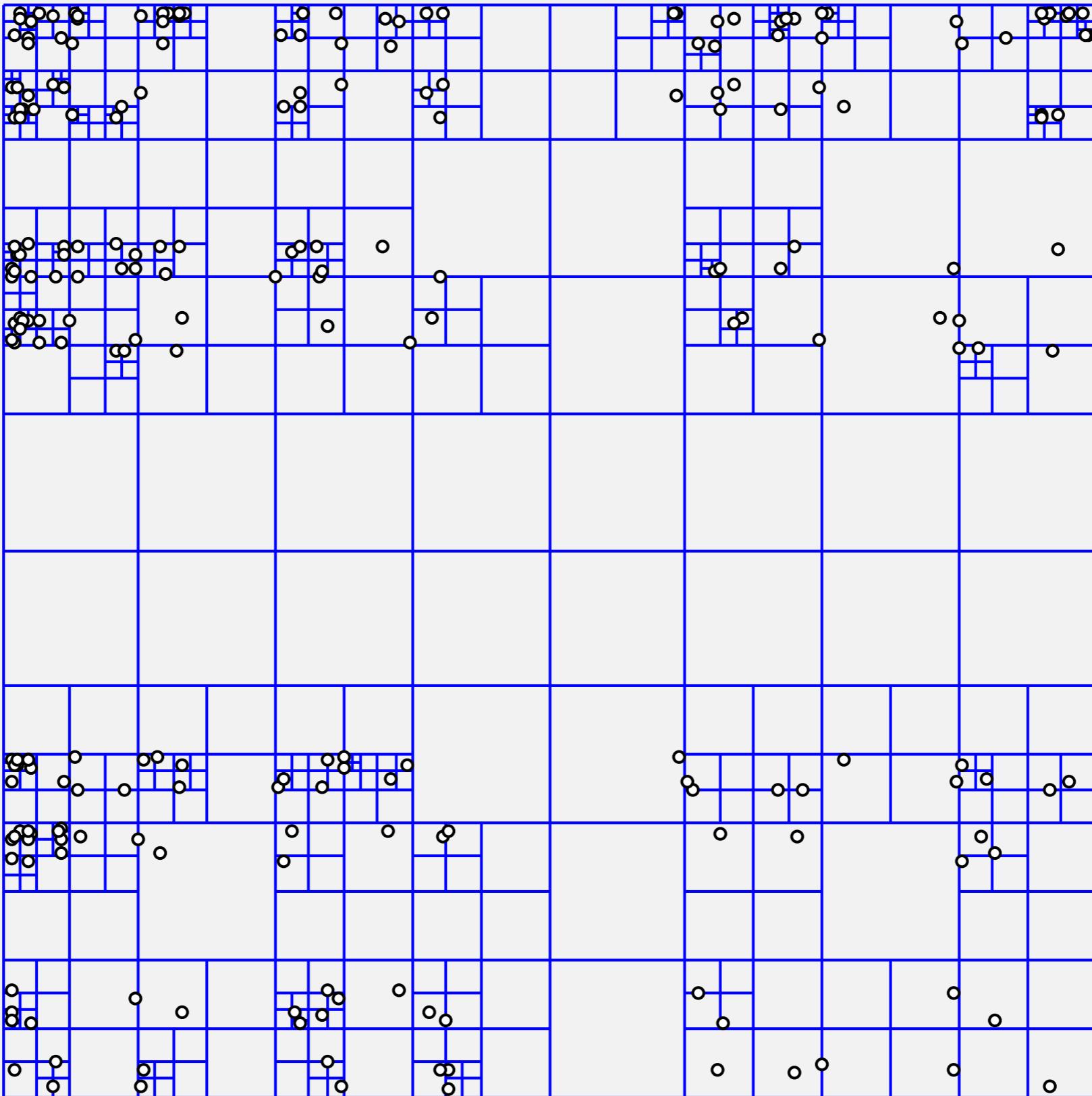


```
public class QuadTree
{
    private Quad quad;
    private Value val;
    private QuadTree NW, NE, SW, SE;
}
```

Benefit. Good performance in the presence of clustering.

Drawback. Arbitrary depth!

Quadtree: larger example



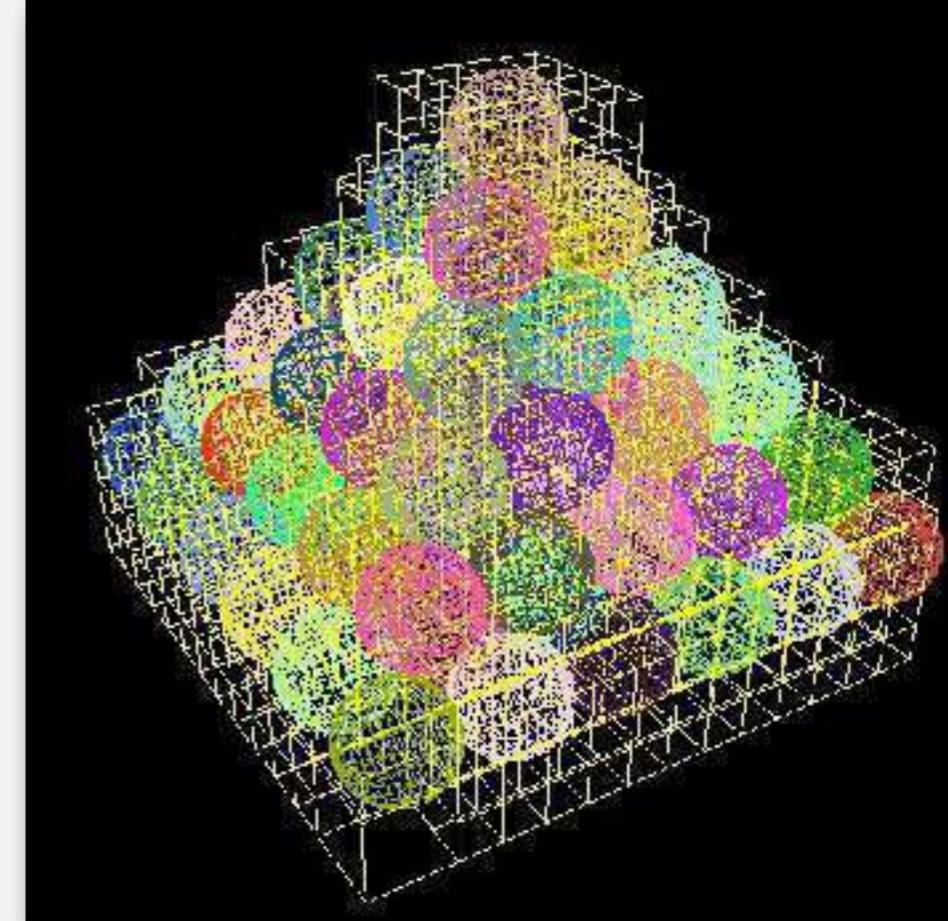
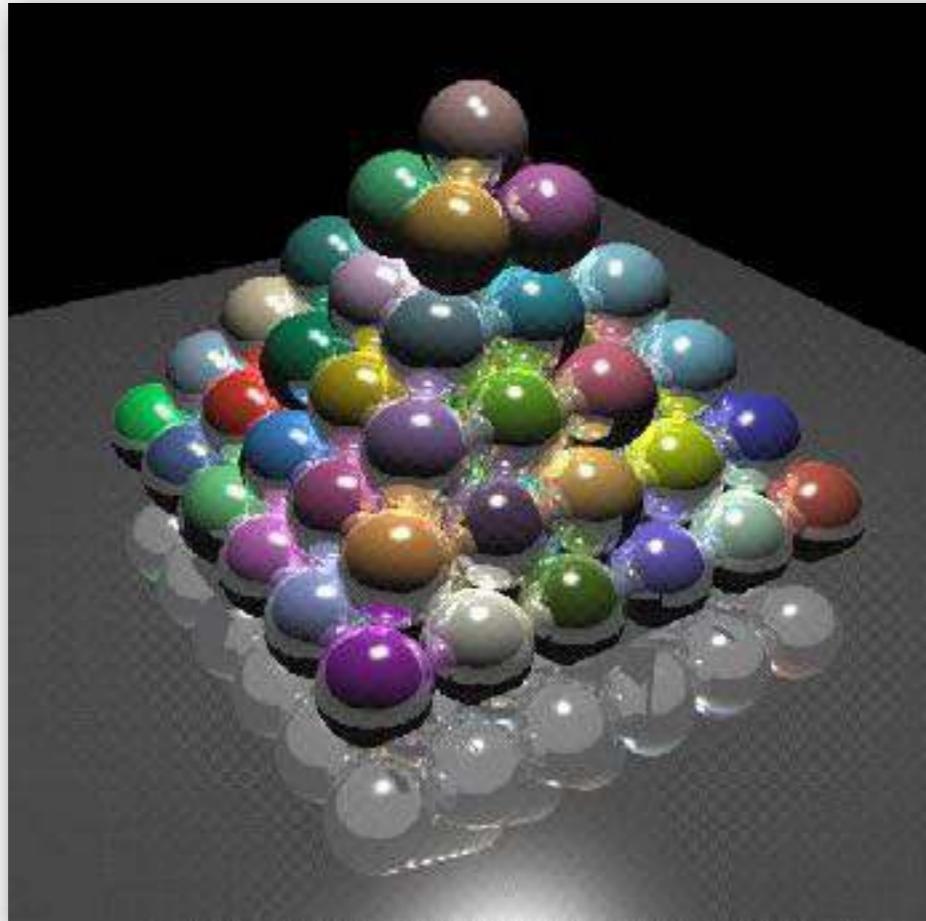
Curse of dimensionality

k-d range search. Orthogonal range search in k -dimensions.

Main application. Multi-dimensional databases.

3d space. Octrees: recursively subdivide 3d space into 8 octants.

100d space. Centrees: recursively subdivide 100d space into 2^{100} centrants???



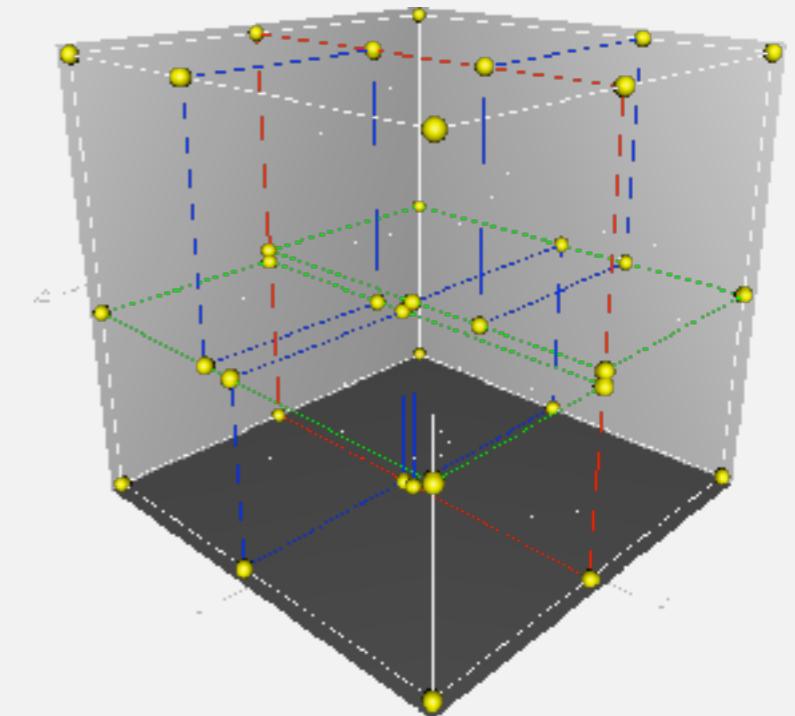
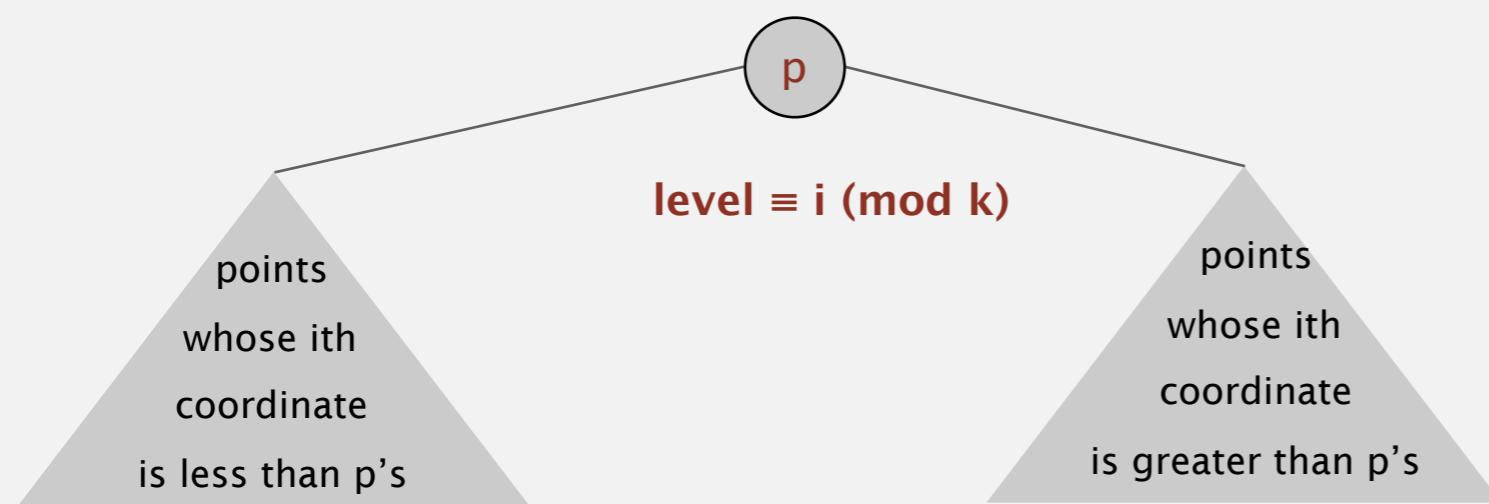
Raytracing with octrees

<http://graphics.cs.ucdavis.edu/~gregorsk/graphics/275.html>

Kd tree

Kd tree. Recursively partition k -dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.



Efficient, simple data structure for processing k -dimensional data.

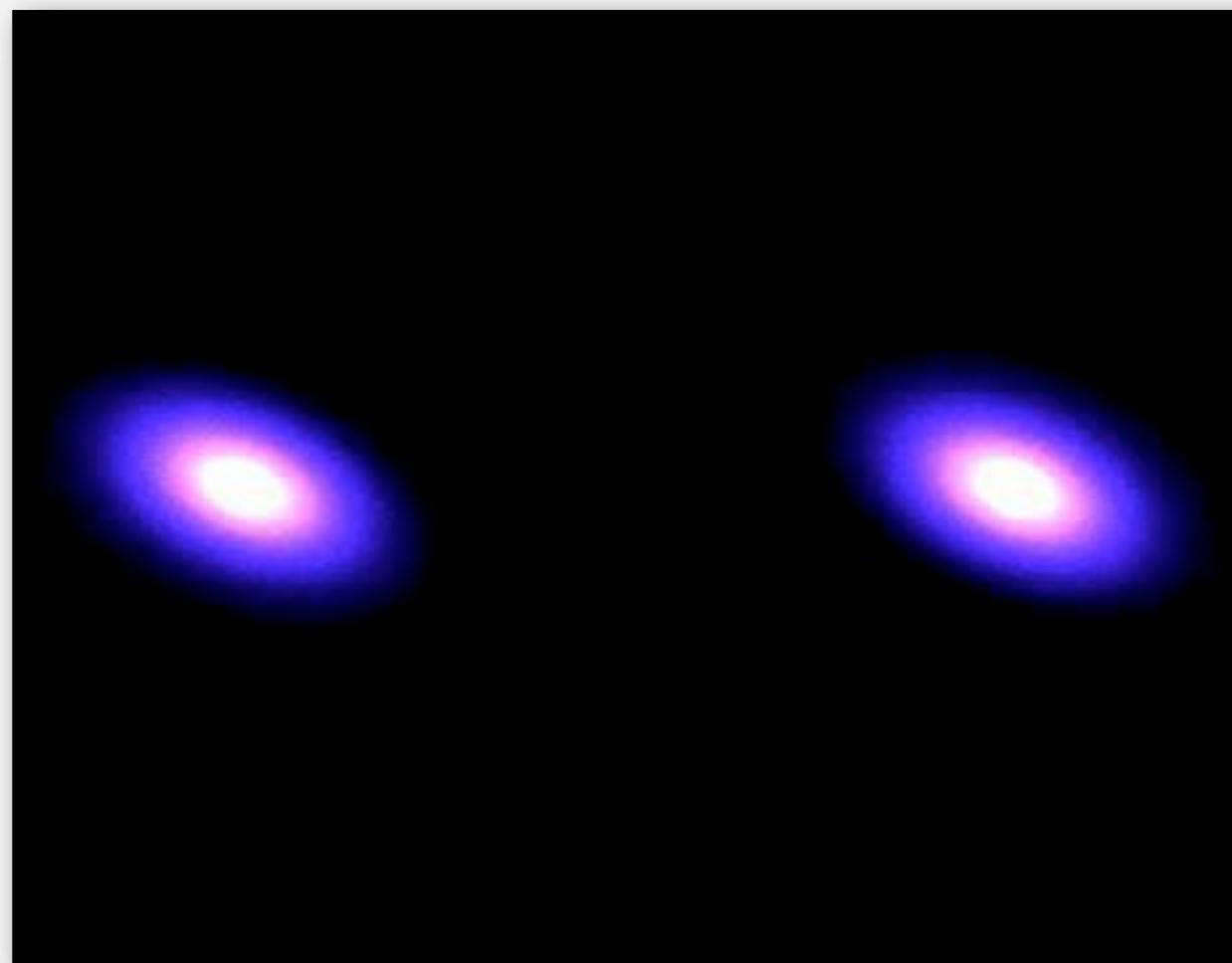
- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad in an algorithms class!



Jon Bentley

N-body simulation

Goal. Simulate the motion of N particles, mutually affected by gravity.



http://www.youtube.com/watch?v=ua7YIN4eL_w

Brute force. For each pair of particles, compute force.

$$F = \frac{G m_1 m_2}{r^2}$$

Appel algorithm for N-body simulation

Key idea. Suppose particle is far, far away from cluster of particles.

- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and **center of mass** of aggregate particle.



Appel algorithm for N-body simulation

- Build 3d-tree with N particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large.

SIAM J. SCI. STAT. COMPUT.
Vol. 6, No. 1, January 1985

© 1985 Society for Industrial and Applied Mathematics
008

AN EFFICIENT PROGRAM FOR MANY-BODY SIMULATION*

ANDREW W. APPEL†

Abstract. The simulation of N particles interacting in a gravitational force field is useful in astrophysics, but such simulations become costly for large N . Representing the universe as a tree structure with the particles at the leaves and internal nodes labeled with the centers of mass of their descendants allows several simultaneous attacks on the computation time required by the problem. These approaches range from algorithmic changes (replacing an $O(N^2)$ algorithm with an algorithm whose time-complexity is believed to be $O(N \log N)$) to data structure modifications, code-tuning, and hardware modifications. The changes reduced the running time of a large problem ($N = 10,000$) by a factor of four hundred. This paper describes both the particular program and the methodology underlying such speedups.

Impact. Running time per step is $N \log N$ instead of $N^2 \Rightarrow$ enables new research.

2.2 Mergesort



- ▶ mergesort
 - ▶ bottom-up mergesort
 - ▶ sorting complexity
 - ▶ comparators

Two classic sorting algorithms

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.

← today

- Java sort for objects.
- Perl, Python stable sort.

Quicksort.

← next lecture

- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

- ▶ **mergesort**
- ▶ **bottom-up mergesort**
- ▶ **sorting complexity**
- ▶ **comparators**

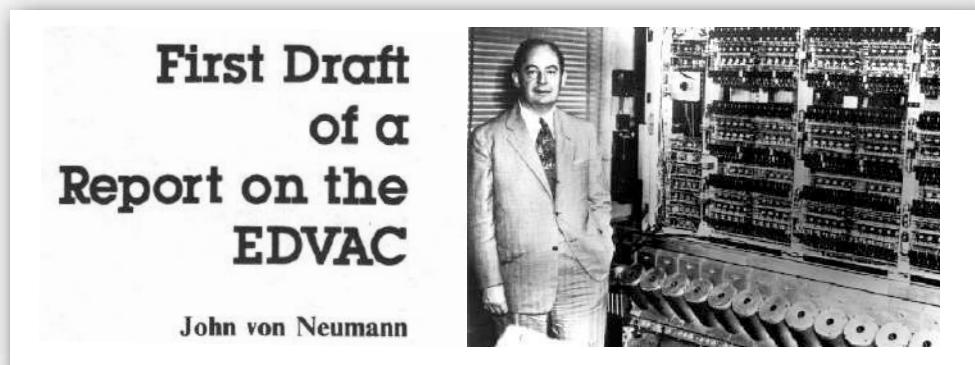
Mergesort

Basic plan.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E	
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E	
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X	
merge results	A	E	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview



Merging

Q. How to combine two sorted subarrays into a sorted whole.

A. Use an auxiliary array.

	a[]										aux[]											
k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9
input	E	E	G	M	R	A	C	E	R	T	-	-	-	-	-	-	-	-	-	-	-	
copy	E	E	G	M	R	A	C	E	R	T	E	E	G	M	R	A	C	E	R	T	-	
0	A										0	5										
1	A	C									0	6	E	E	G	M	R	A	C	E	R	T
2	A	C	E								0	7	E	E	G	M	R	C	E	R	T	
3	A	C	E	E							1	7	E	E	G	M	R		E	R	T	
4	A	C	E	E	E						2	7	E	G	M	R			E	R	T	
5	A	C	E	E	E	G					2	8		G	M	R			E	R	T	
6	A	C	E	E	E	G	M				3	8		G	M	R			R	T		
7	A	C	E	E	E	G	M	R			4	8			M	R			R	T		
8	A	C	E	E	E	G	M	R	R		5	8				R			R	T		
9	A	C	E	E	E	G	M	R	R	T	5	9							R	T		
merged result	A	C	E	E	E	G	M	R	R	T	6	10									T	
Abstract in-place merge trace																						

Merging: Java implementation

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
    assert isSorted(a, lo, mid);      // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi);    // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++)                                copy
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if         (i > mid)                      a[k] = aux[j++];          merge
        else if   (j > hi)                        a[k] = aux[i++];
        else if   (less(aux[j], aux[i]))  a[k] = aux[j++];
        else                           a[k] = aux[i++];
    }

    assert isSorted(a, lo, hi);      // postcondition: a[lo..hi] sorted
}
```



Assertions

Assertion. Statement to test assumptions about your program.

- Helps detect logic bugs.
- Documents code.

Java assert statement. Throws an exception unless boolean condition is true.

```
assert isSorted(a, lo, hi);
```

Can enable or disable at runtime. ⇒ No cost in production code.

```
java -ea MyProgram    // enable assertions
java -da MyProgram    // disable assertions (default)
```

Best practices. Use to check internal invariants. Assume assertions will be disabled in production code (e.g., don't use for external argument-checking).

Mergesort: Java implementation

```
public class Merge
{
    private static Comparable[] aux;

    private static void merge(Comparable[] a, int lo, int mid, int hi)
    { /* as before */ }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid+1, hi);
        merge(a, lo, m, hi);
    }

    public static void sort(Comparable[] a)
    {
        aux = new Comparable[a.length];
        sort(a, 0, a.length - 1);
    }
}
```



Mergesort trace

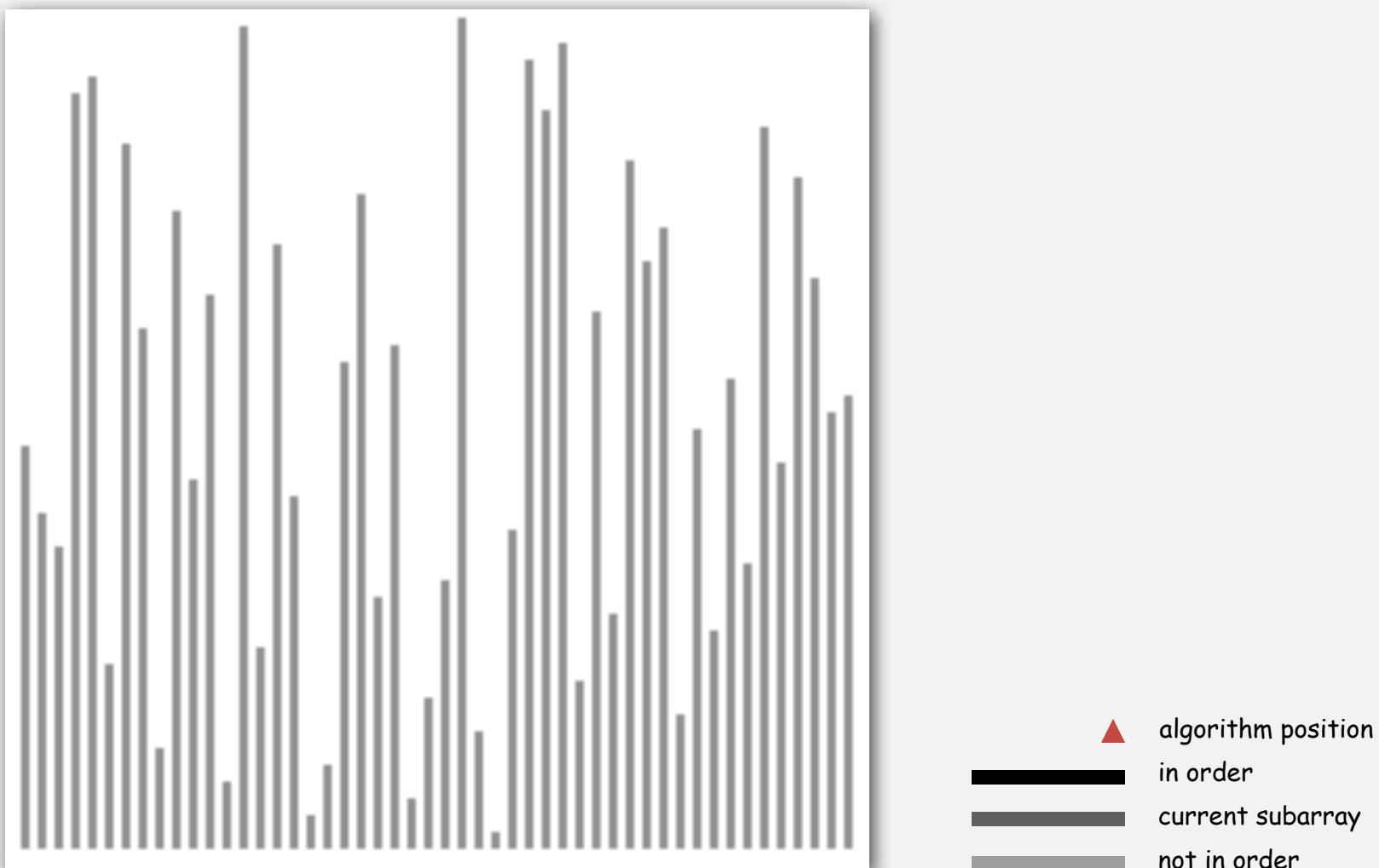
	a[]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
merge(a, 0, 0, 1)	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Trace of merge results for top-down mergesort

result after recursive call

Mergesort animation

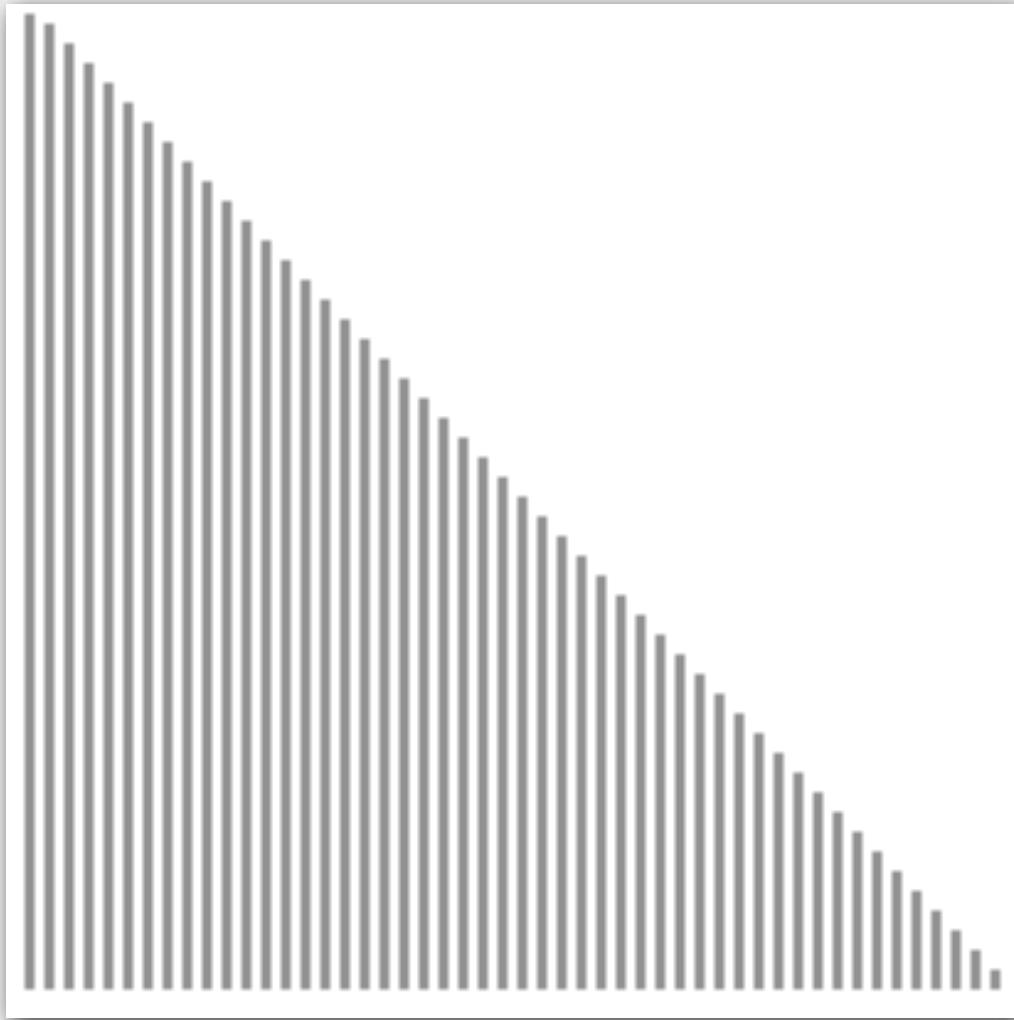
50 random elements



<http://www.sorting-algorithms.com/merge-sort>

Mergesort animation

50 reverse-sorted elements



- ▲ algorithm position
- █ in order
- █ current subarray
- █ not in order

<http://www.sorting-algorithms.com/merge-sort>

Mergesort: empirical analysis

Running time estimates:

- Home pc executes 10^8 comparisons/second.
- Supercomputer executes 10^{12} comparisons/second.

	insertion sort (N^2)			mergesort ($N \log N$)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Bottom line. Good algorithms are better than supercomputers.

Mergesort: mathematical analysis

Proposition. Mergesort uses $\sim 2 N \lg N$ data moves to sort any array of size N .

Def. $D(N) =$ number of data moves to mergesort an array of size N .

$$= D(N/2) + D(N/2) + 2N$$

left half right half merge

Mergesort recurrence. $D(N) = 2 D(N/2) + 2N$ for $N > 1$, with $T(1) = 0$.

- Not quite right for odd N .
- Similar recurrence holds for many divide-and-conquer algorithms.

Solution. $D(N) \sim 2 N \lg N$.

- For simplicity, we'll prove when N is a power of 2.
- True for all N . [see COS 340]

Mergesort recurrence: proof 2

Mergesort recurrence. $D(N) = 2 D(N/2) + 2N$ for $N > 1$, with $D(1) = 0$.

Proposition. If N is a power of 2, then $D(N) = 2 N \lg N$.

Pf.

$\begin{aligned} D(N) &= 2 D(N/2) + 2N \\ D(N)/N &= 2 D(N/2)/N + 2 \\ &= D(N/2)/(N/2) + 2 \\ &= D(N/4)/(N/4) + 2 + 2 \\ &= D(N/8)/(N/8) + 2 + 2 + 2 \\ &\quad \dots \\ &= D(N/N)/(N/N) + 2 + 2 + \dots + 2 \\ &= 2 \lg N \end{aligned}$	<p>given</p> <p>divide both sides by N</p> <p>algebra</p> <p>apply to first term</p> <p>apply to first term again</p> <p>stop applying, $T(1) = 0$</p>
---	--

Mergesort recurrence: proof 3

Mergesort recurrence. $D(N) = 2 D(N / 2) + 2 N$ for $N > 1$, with $D(1) = 0$.

Proposition. If N is a power of 2, then $D(N) = 2 N \lg N$.

Pf. [by induction on N]

- **Base case:** $N = 1$.
- **Inductive hypothesis:** $D(N) = 2N \lg N$.
- **Goal:** show that $D(2N) = 2(2N)\lg (2N)$.

$$\begin{aligned}D(2N) &= 2 D(N) + 4N \\&= 4 N \lg N + 4 N \\&= 4 N (\lg (2N) - 1) + 4N \\&= 4 N \lg (2N)\end{aligned}$$

given

inductive hypothesis

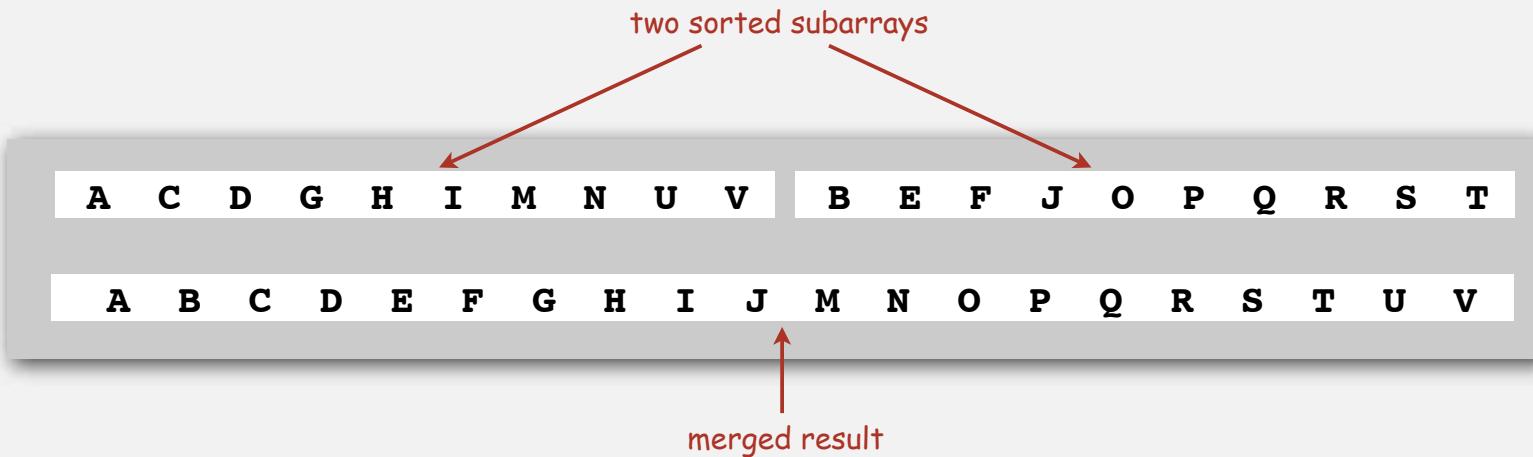
algebra

QED

Mergesort analysis: memory

Proposition G. Mergesort uses extra space proportional to N.

Pf. The array `aux[]` needs to be of size N for the last merge.



Def. A sorting algorithm is **in-place** if it uses $O(\log N)$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

Challenge for the bored. In-place merge. [Kronrud, 1969]

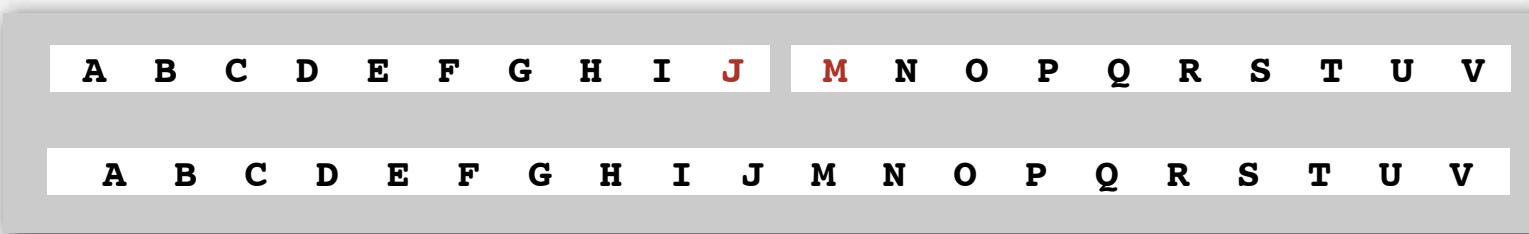
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 elements.

Stop if already sorted.

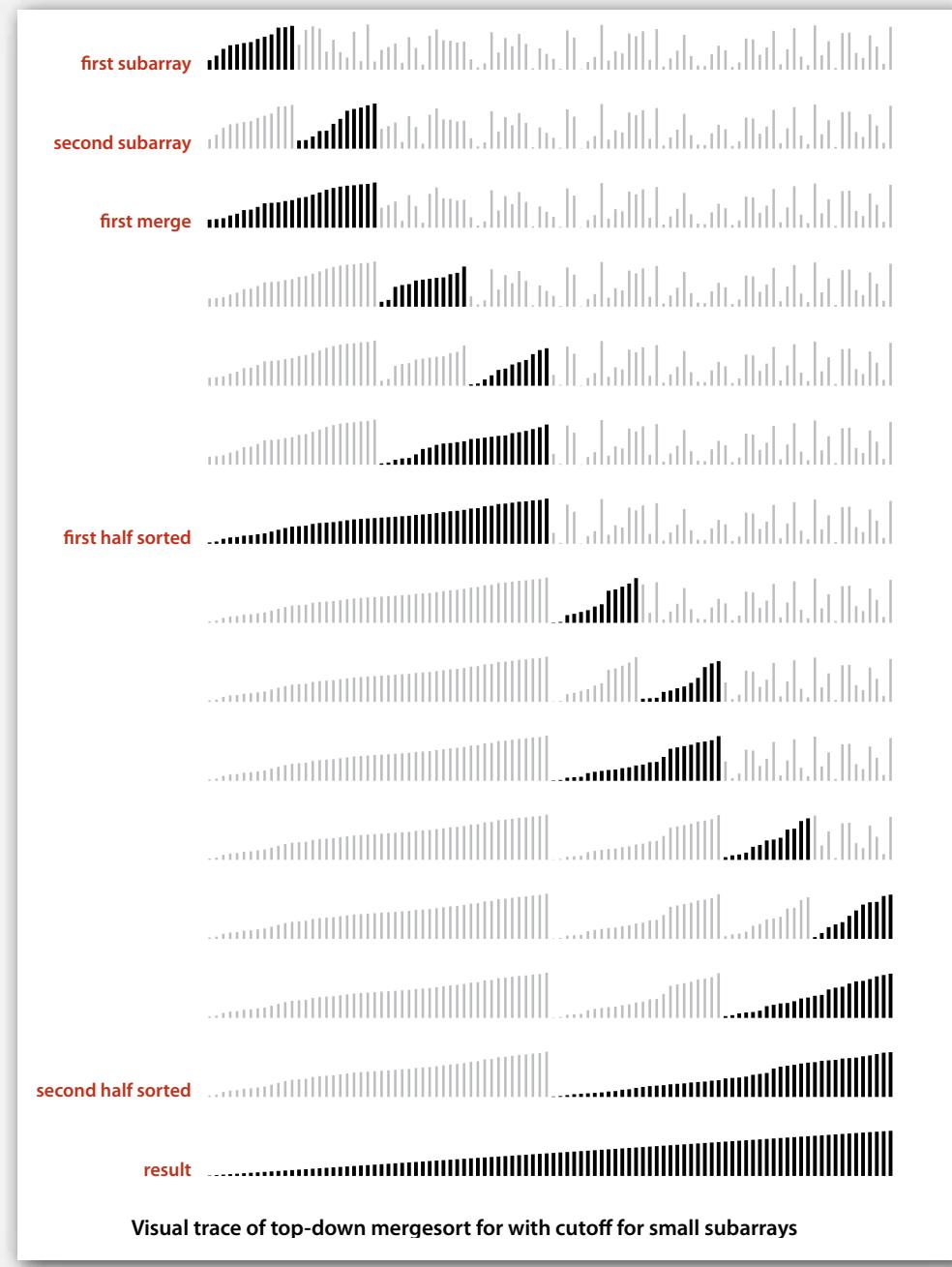
- Is biggest element in first half \leq smallest element in second half?
- Helps for partially-ordered arrays.



Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

Ex. See `MergeX.java` or `Arrays.sort()`.

Mergesort visualization



- ▶ mergesort
- ▶ bottom-up mergesort
- ▶ sorting complexity
- ▶ comparators

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16,

	a[i]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz = 2	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 4, 4, 5)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 6, 6, 7)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, 8, 8, 9)	E	M	G	R	E	S	O	R	E	T	X	A	M	P	L	E
merge(a, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	L	E
merge(a, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	E	L
sz = 4																
merge(a, 0, 1, 3)	E	G	M	R	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, 4, 5, 7)	E	G	M	R	E	O	R	S	E	T	A	X	M	P	E	L
merge(a, 8, 9, 11)	E	G	M	R	E	O	R	S	A	E	T	X	M	P	E	L
merge(a, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	E	L	M	P
sz = 8																
merge(a, 0, 3, 7)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
sz = 16																
merge(a, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X
Trace of merge results for bottom-up mergesort																

Bottom line. No recursion needed!

Bottom-up mergesort: Java implementation

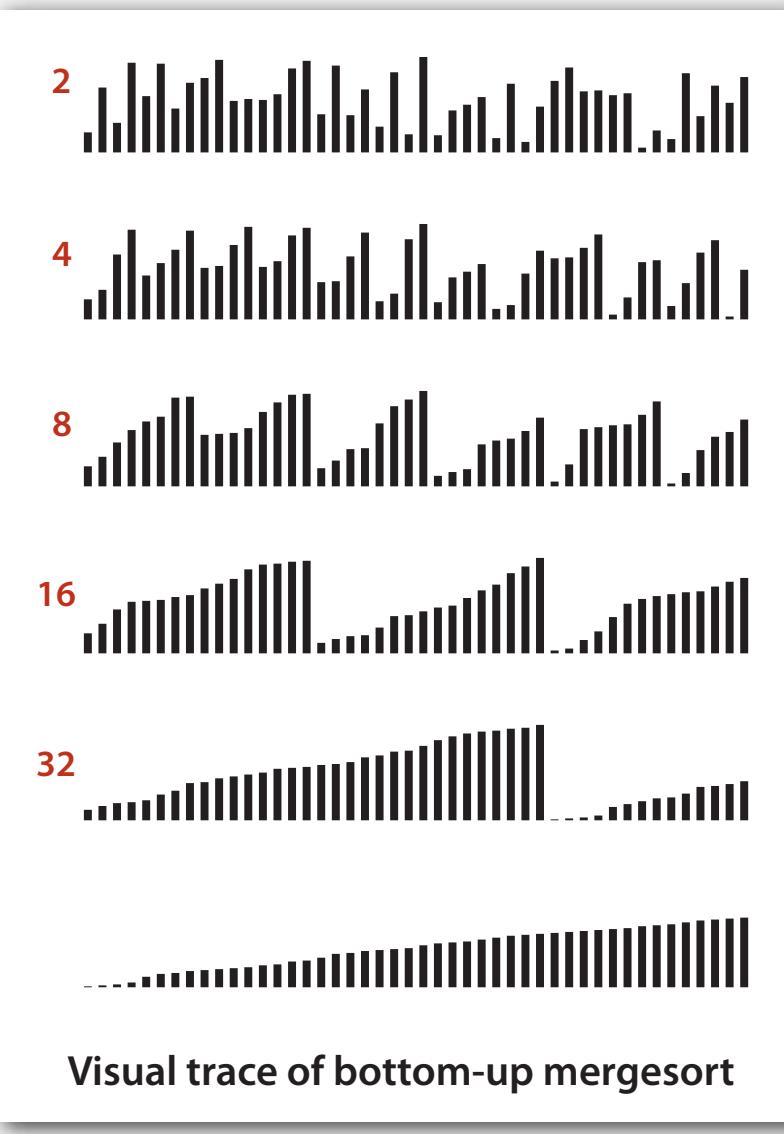
```
public class MergeBU
{
    private static Comparable[] aux;

    private static void merge(Comparable[] a, int lo, int mid, int hi)
    { /* as before */ }

    public static void sort(Comparable[] a)
    {
        int N = a.length;
        aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo += sz+sz)
                merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
```

Bottom line. Concise industrial-strength code, if you have the space.

Bottom-up mergesort: visual trace



BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

HASHING, SEARCH APPLICATIONS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

ST implementations: summary

implementation	worst-case cost (after N inserts)			average-case cost (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>

Q. Can we do better?

A. Yes, but with different access to the data (if we don't need ordered ops).

Hashing: basic plan

Save items in a **key-indexed table** (index is a function of the key).

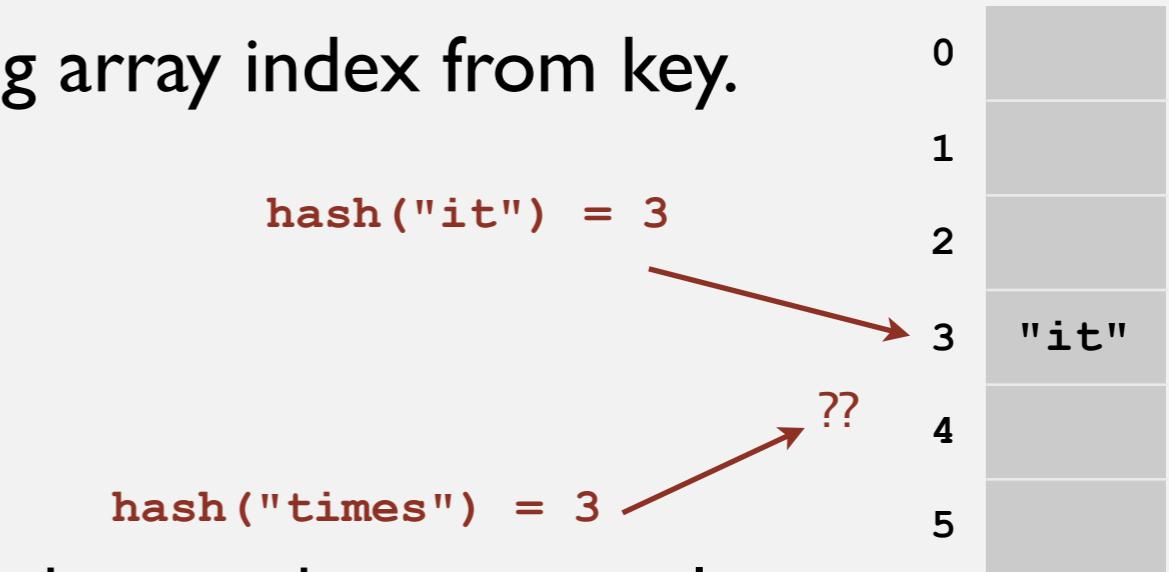
Hash function. Method for computing array index from key.

Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.
- Collision resolution: Algorithm and data structure to handle two keys that hash to the same array index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as index. Very large index table, few collisions
- No time limitation: trivial collision resolution with sequential search. Small table, lots of collisions, must search within the cell.
- Space and time limitations: hashing (the real world).



HASHING

- ▶ Hash functions
- ▶ Separate chaining
- ▶ Linear probing

Computing the hash function

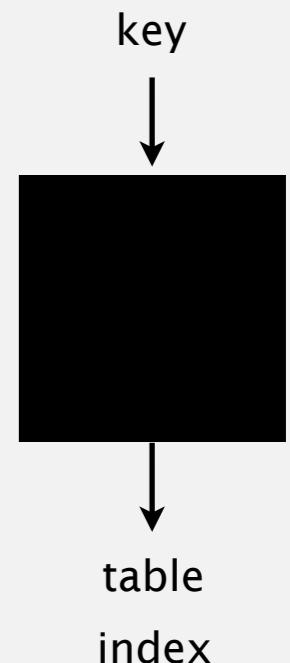
Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.

thoroughly researched problem,
still problematic in practical applications

Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.



Ex 2. Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

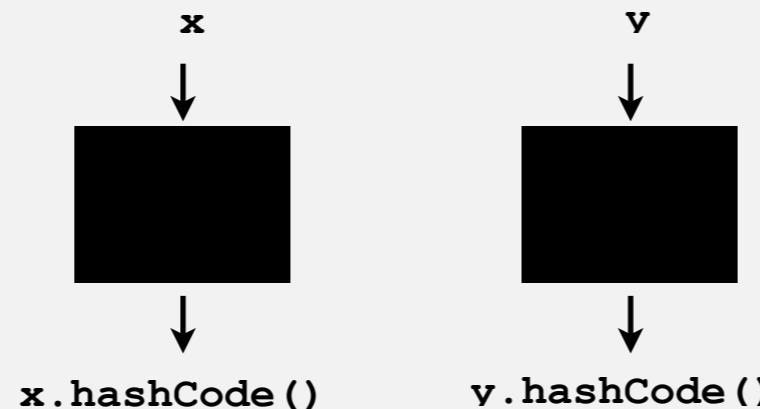
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return 17.

Customized implementations. `Integer`, `Double`, `String`, `File`, `URL`, `Date`, ...

User-defined types. Users are on their own.

Implementing hash code: integers, booleans, and doubles

Java library implementations

```
public final class Integer
{
    private final int value;
    ...

    public int hashCode()
    {   return value;   }
}
```

```
public final class Boolean
{
    private final boolean value;
    ...

    public int hashCode()
    {
        if (value) return 1231;
        else       return 1237;
    }
}
```

```
public final class Double
{
    private final double value;
    ...

    public int hashCode()
    {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Implementing hash code: strings

Java library implementation

```
public final class String
{
    private final char[] s;
    ...

    public int hashCode()
    {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

ith character of s

char	Unicod e
...	...
'a'	97
'b'	98
'c'	99
...	...

- Horner's method to hash string of length L : L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \dots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex. `String s = "call";`

$$\begin{aligned} \text{int code} = s.hashCode(); &\quad \leftarrow 3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \\ &\quad = 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \\ &\quad \quad \quad \text{(Horner's method)} \end{aligned}$$

Implementing hash code: strings

Performance optimization.

- Cache the hash value in an instance variable.
- Return cached value.

```
public final class String
{
    private int hash = 0;                                ← cache of hash code
    private final char[] s;
    ...

    public int hashCode()
    {
        int h = hash;
        if (h != 0) return h;                            ← return cached value
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        hash = h;                                       ← store cache of hash code
        return h;
    }
}
```

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
    private final String who;
    private final Date when;
    private final double amount;

    public Transaction(String who, Date when, double amount)
    { /* as before */ }

    ...

    public boolean equals(Object y)
    { /* as before */ }

    public int hashCode()
    {
        int hash = 17;           ← nonzero constant
        hash = 31*hash + who.hashCode(); ← for reference types,
                                         use hashCode()
        hash = 31*hash + when.hashCode(); ← for primitive types,
                                         use hashCode()
        hash = 31*hash + ((Double) amount).hashCode(); ← of wrapper type
        return hash;
    }
}
```

nonzero constant

typically a small prime

for reference types,
use hashCode()

for primitive types,
use hashCode()
of wrapper type

Hash code design

"Standard" recipe for user-defined types.

- Combine each significant field using the $31x + y$ rule.
- If field is a primitive type, use wrapper type `hashCode()`.
- If field is null, return 0.
- If field is a reference type, use `hashCode()`. ← **applies rule recursively**
- If field is an array, apply to each entry. ← **or use `Arrays.deepHashCode()`**

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Keys are bitstring; "universal" hash functions exist.

Basic rule. Need to use the whole key to compute hash code;
consult an expert for state-of-the-art hash codes.

Modular hashing

Hash code. An `int` between -2^{31} and $2^{31}-1$.

Hash function. An `int` between 0 and $M-1$ (for use as array index).

typically a prime or power of 2

```
private int hash(Key key)
{   return key.hashCode() % M; }
```

bug

```
private int hash(Key key)
{   return Math.abs(key.hashCode()) % M; }
```

1-in-a-billion bug

hashCode() of "polygenelubricants" is -2^{31}

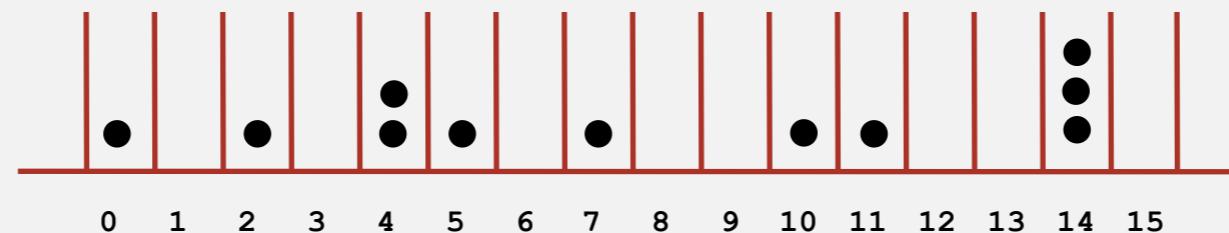
```
private int hash(Key key)
{   return (key.hashCode() & 0x7fffffff) % M; }
```

correct

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

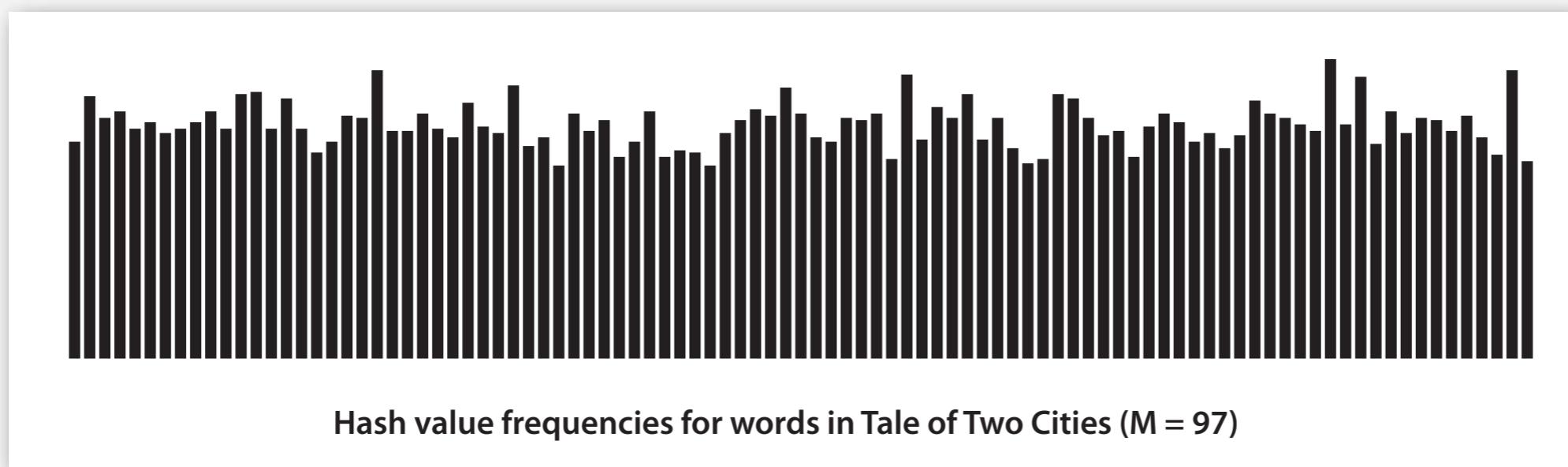
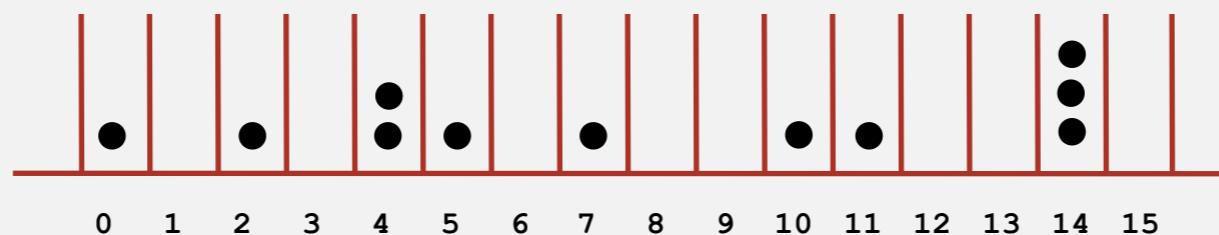
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Java's `String` data uniformly distribute the keys of Tale of Two Cities

HASHING

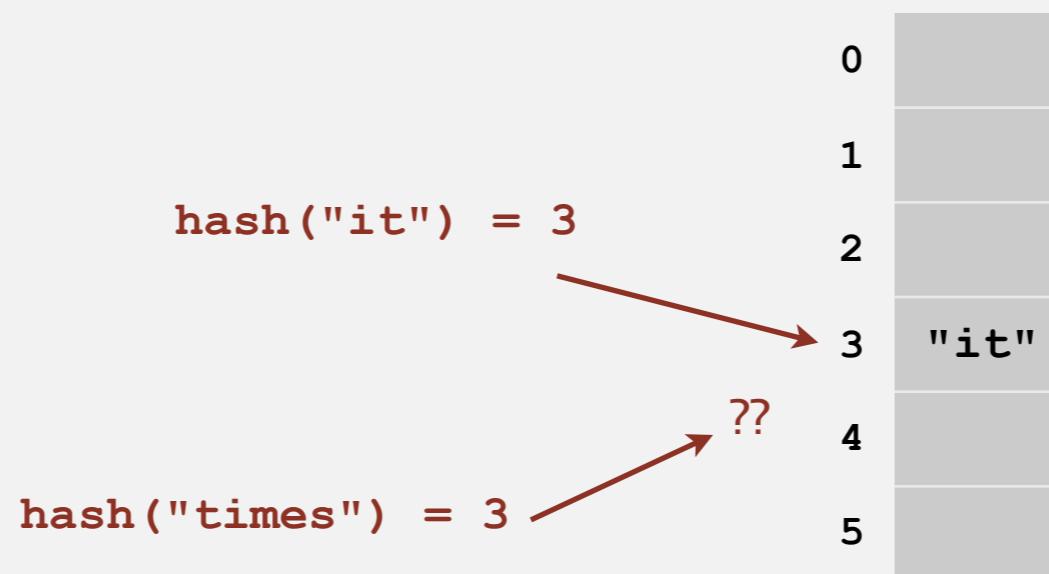
- ▶ Hash functions
- ▶ Separate chaining
- ▶ Linear probing

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem \Rightarrow can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing \Rightarrow collisions will be evenly distributed.

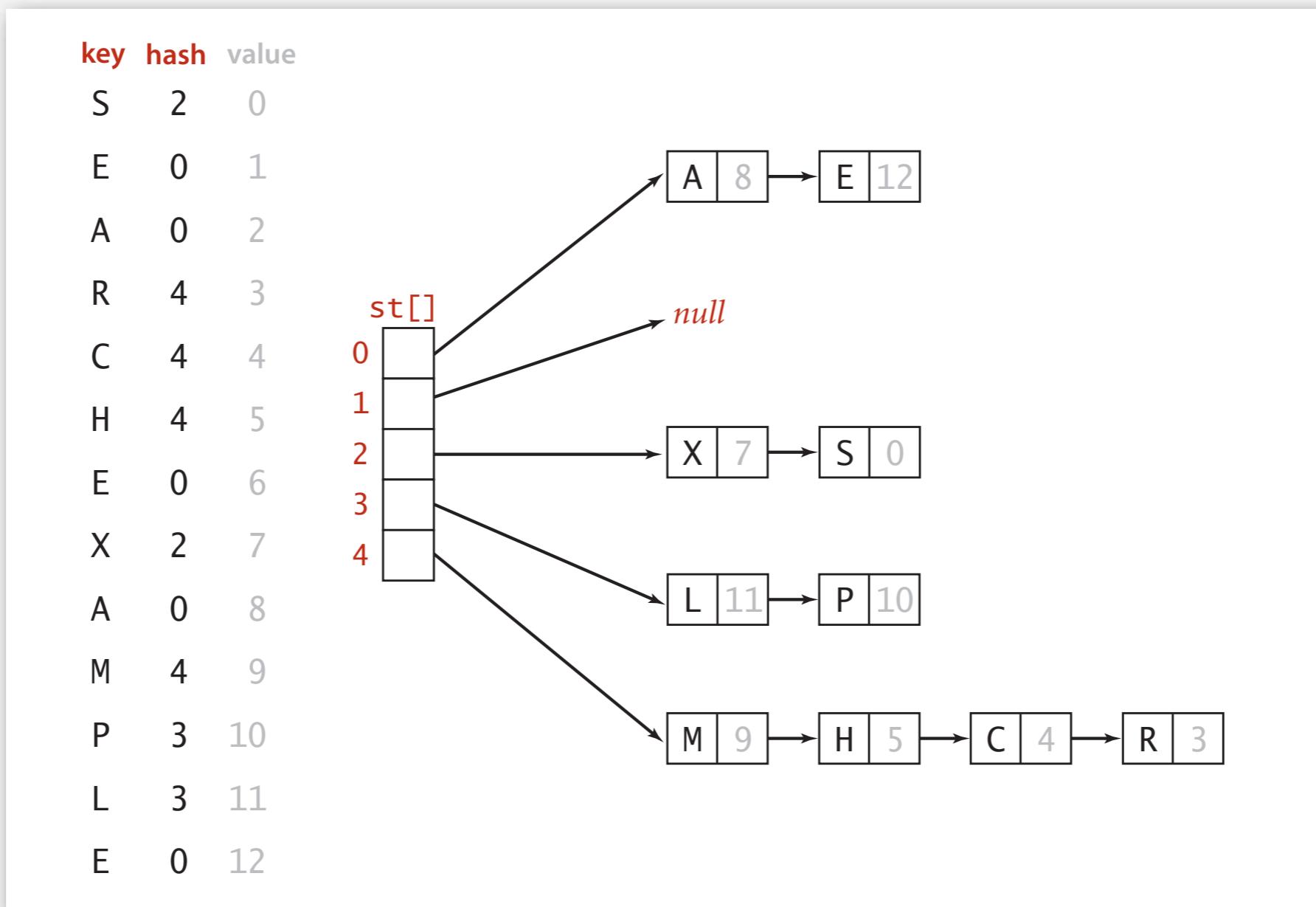
Challenge. Deal with collisions efficiently.



Separate chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: need to search only i^{th} chain.



Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;                      // number of chains
    private Node[] st = new Node[M];          // array of chains

    private static class Node
    {
        private Object key; ← no generic array creation
        private Object val; ← (declare key and value of type Object)
        private Node next;
        ...
    }

    private int hash(Key key)
    {   return (key.hashCode() & 0x7fffffff) % M;   }

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }
}
```

array doubling
and halving
code omitted

Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;                      // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

    private int hash(Key key)
    {   return (key.hashCode() & 0x7fffffff) % M;   }

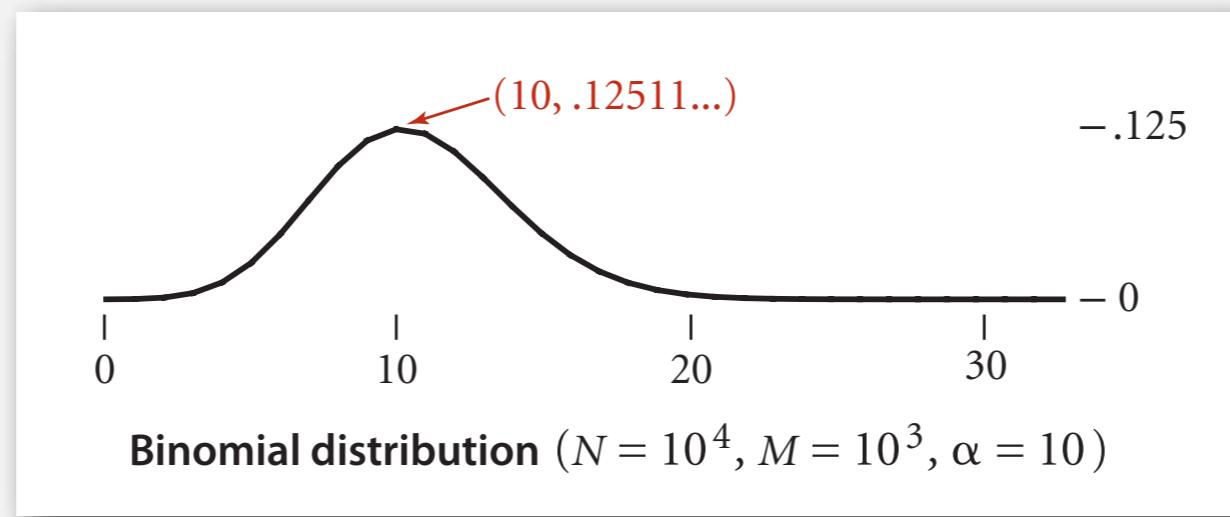
    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }

}
```

Analysis of separate chaining

Proposition. Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Consequence. Number of probes for search/insert is proportional to N / M .

- M too large \Rightarrow too many empty chains.
- M too small \Rightarrow chains too long.
- Typical choice: $M \sim N / 5 \Rightarrow$ constant-time ops.

equals () and hashCode ()

↑
M times faster than
sequential search

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
red-black tree	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>
separate chaining	N *	N *	N *	3-5 *	3-5 *	3-5 *	no	<code>equals()</code>

* under uniform hashing assumption

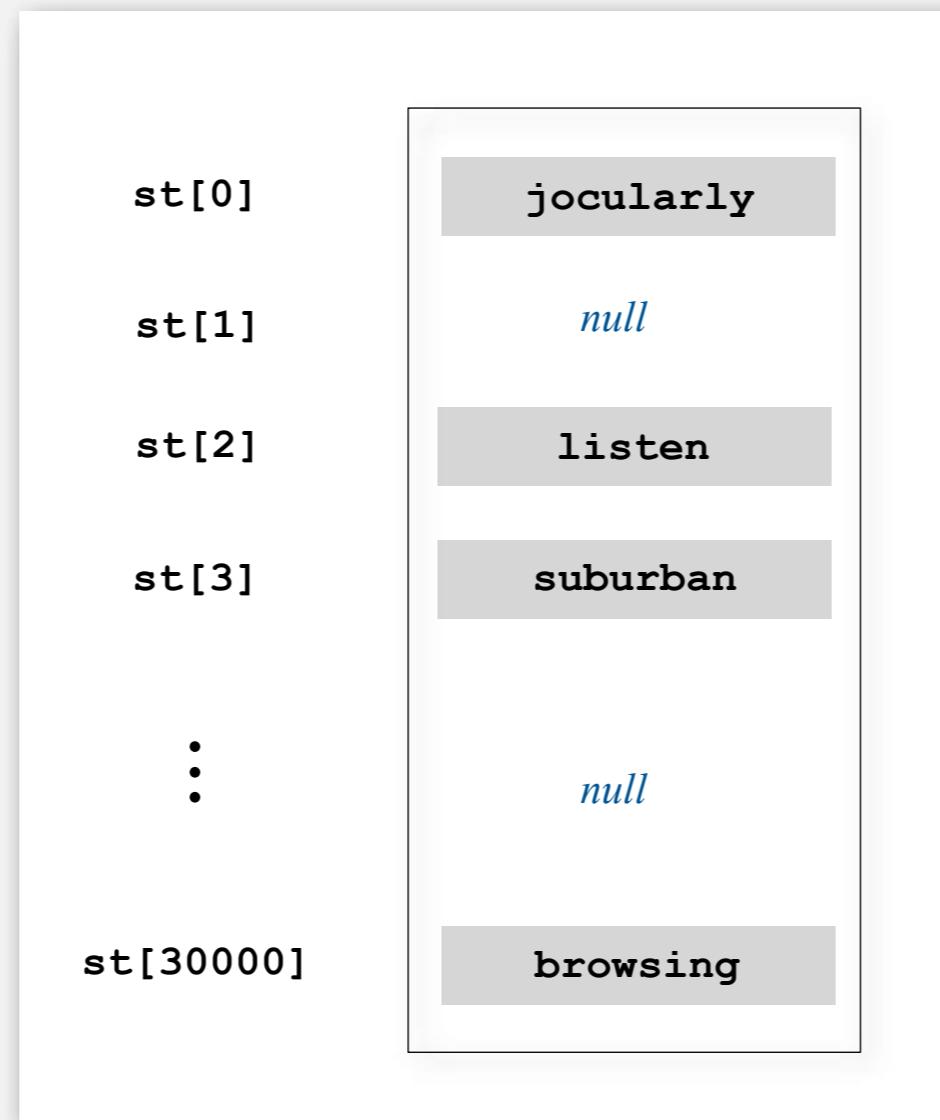
HASHING

- ▶ Hash functions
- ▶ Separate chaining
- ▶ Linear probing

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rochester-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

A horizontal array of 16 gray cells, indexed from 0 to 15. The label "st[]" is positioned to the left of the first cell. Below the array, the text "1 = 16" is displayed.

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert S

$\text{hash}(S) = 6$



$M = 16$

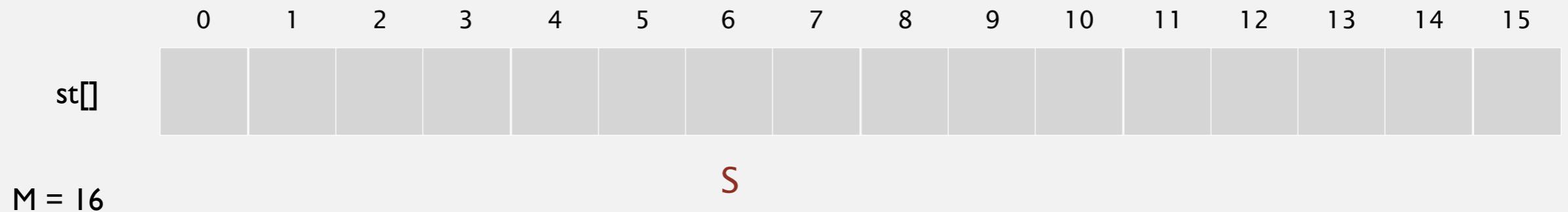
Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert S

hash(S) = 6



Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert S

hash(S) = 6



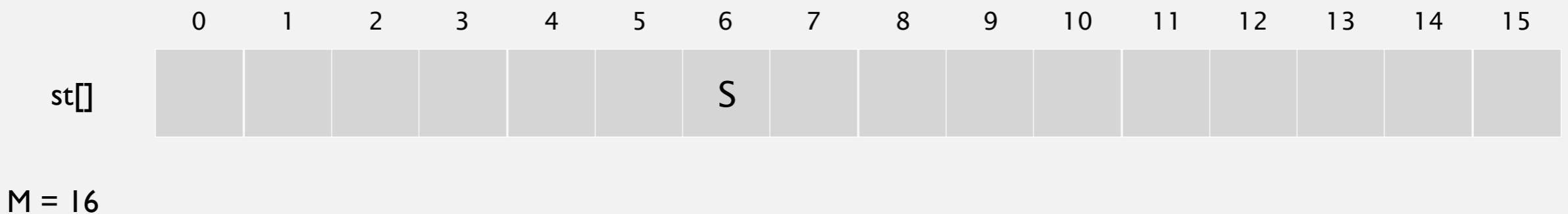
$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table



Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert E

hash(E) = 10



$M = 16$

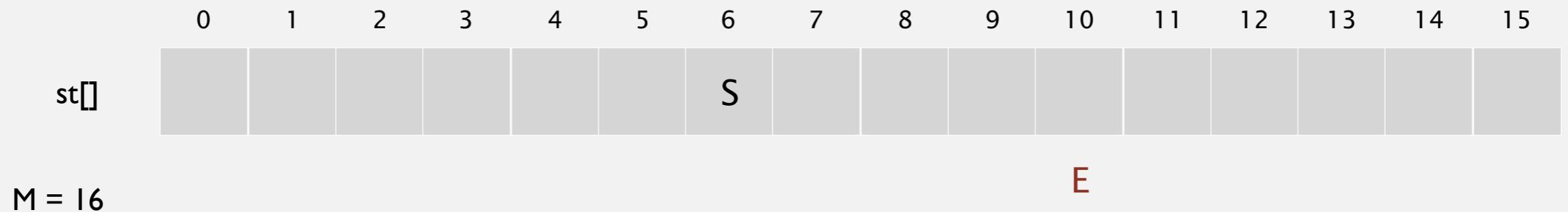
Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert E

hash(E) = 10



Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert E

hash(E) = 10

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]						S				E					

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

The diagram shows a horizontal array of 16 slots, labeled from 0 to 15 above. The slots are represented by gray rectangles. At slot 6, there is a large bold letter 'S'. At slot 10, there is a large bold letter 'E'. All other slots are empty.

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert A

hash(A) = 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]						S				E					

$M = 16$

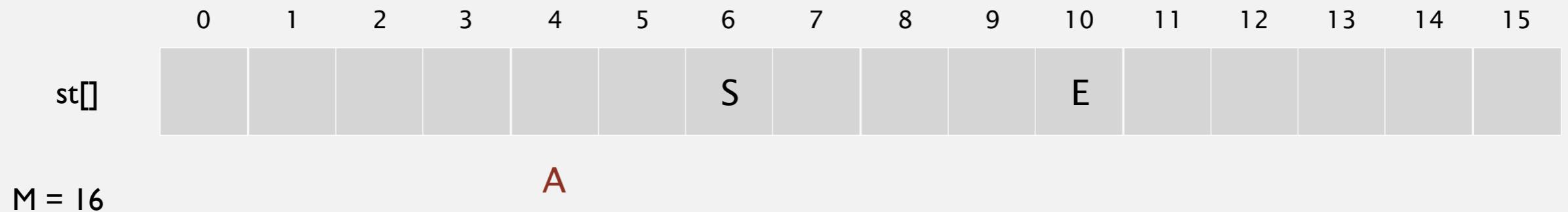
Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert A

$$\text{hash}(A) = 4$$



Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert A

hash(A) = 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A		S				E					

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A		S				E					

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert R

hash(R) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A		S				E					

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert R

hash(R) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	S				E						R

M = 16

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert R

hash(R) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	S				E				R		

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

The diagram shows a horizontal array of 16 slots, indexed from 0 to 15 above the array. The slots are represented by light gray rectangles. Labels are placed in specific slots:

- Slot 4 contains the letter **A**.
- Slot 5 contains the letter **S**.
- Slot 9 contains the letter **E**.
- Slot 12 contains the letter **R**.

M = 16

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert C

hash(C) = 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	S				E				R		

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert C

hash(C) = 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	S				E				R		

M = 16

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert C

hash(C) = 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S			E				R		

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert H

hash(H) = 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S			E				R		

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert H

$$\text{hash}(H) = 4$$

The diagram shows a horizontal array of 16 cells, indexed from 0 to 15. Cells 4, 5, 6, 9, and 14 contain the letters A, C, S, E, and R respectively. Above index 7, the letter H is written in red.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S			E					R	

H

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert H

hash(H) = 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S			E				R		

M = 16 H

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert H

$$\text{hash}(H) = 4$$

Diagram illustrating a 16-element array `st[]` indexed from 0 to 15. The elements are represented by gray boxes. Indices are labeled above the array, and specific elements are labeled inside their respective boxes:

- Indices: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
- Elements:
 - Index 4: A
 - Index 5: C
 - Index 6: S
 - Index 10: E
 - Index 14: R
 - Index 9: H (highlighted in red)

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert H

hash(H) = 4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S				E				R	
M = 16												H				

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert H

hash(H) = 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S	H			E				R	

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E				R	

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert X

hash(X) = 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S	H			E				R	

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert X

$$\text{hash}(X) = 15$$

The diagram shows a horizontal array of 16 cells, indexed from 0 to 15 above each cell. The cells are shaded gray. Cells 4, 5, 6, 7, 10, and 15 contain the letters A, C, S, H, E, and X respectively. Cells 0, 1, 2, and 3 are empty.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				A	C	S	H		E					R	X

M = 16

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert X

hash(X) = 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S	H			E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]					A	C	S	H			E			R	X	

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert M

hash(M) = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S	H			E				R	X

M = 16

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert M

hash(M) = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]				A	C	S	H			E				R	X

M = 16 M

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert M

hash(M) = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E			R	X

M = 16

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M			A	C	S	H			E			R	X	

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M		A	C	S	H			E			R	X	

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]		M		A	C	S	H			E			R	X	

M = 16 P

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert P

$$\text{hash}(P) = 14$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M			A	C	S	H			E				R	X

$M = 16$ P P

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert P

hash(P) = 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H		E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert L

$$\text{hash}(L) = 6$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H		E				R	X	

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert L

$$\text{hash}(L) = 6$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H		E				R	X	

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H			E				R	X
M = 16												L				

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

insert L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search E

hash(E) = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16															E	

search hit
(return corresponding value)

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16									L							

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search L

$$\text{hash}(L) = 6$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E			R	X	
M = 16									L							

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search L

hash(L) = 6

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16									L							

search hit
(return corresponding value)

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

linear probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$$M = 16$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search K

$$\text{hash}(K) = 5$$

K

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[0]	P	M			A	C	S	H	L		E			R	X

$M = 16$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search K

$$\text{hash}(K) = 5$$

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16									K							

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16												K				

Linear probing hash table

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

search K

hash(K) = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X
M = 16	K												search miss (return null)			

Linear probing - Summary

Hash. Map key to integer i between 0 and $M - 1$.

Insert. Put at table index i if free; if not try $i + 1, i + 2$, etc.

Search. Search table index i ; if occupied but no match, try $i + 1, i + 2$, etc.

Note. Array size M must be greater than number of key-value pairs N .

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$$M = 16$$

Linear probing ST implementation

```
public class LinearProbingHashST<Key, Value>
{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];

    private int hash(Key key) { /* as before */ }

    public void put(Key key, Value val)
    {
        int i;
        for (i = hash(key); keys[i] != null; i = (i+1) % M)
            if (keys[i].equals(key))
                break;
        keys[i] = key;
        vals[i] = val;
    }

    public Value get(Key key)
    {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M)
            if (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
```

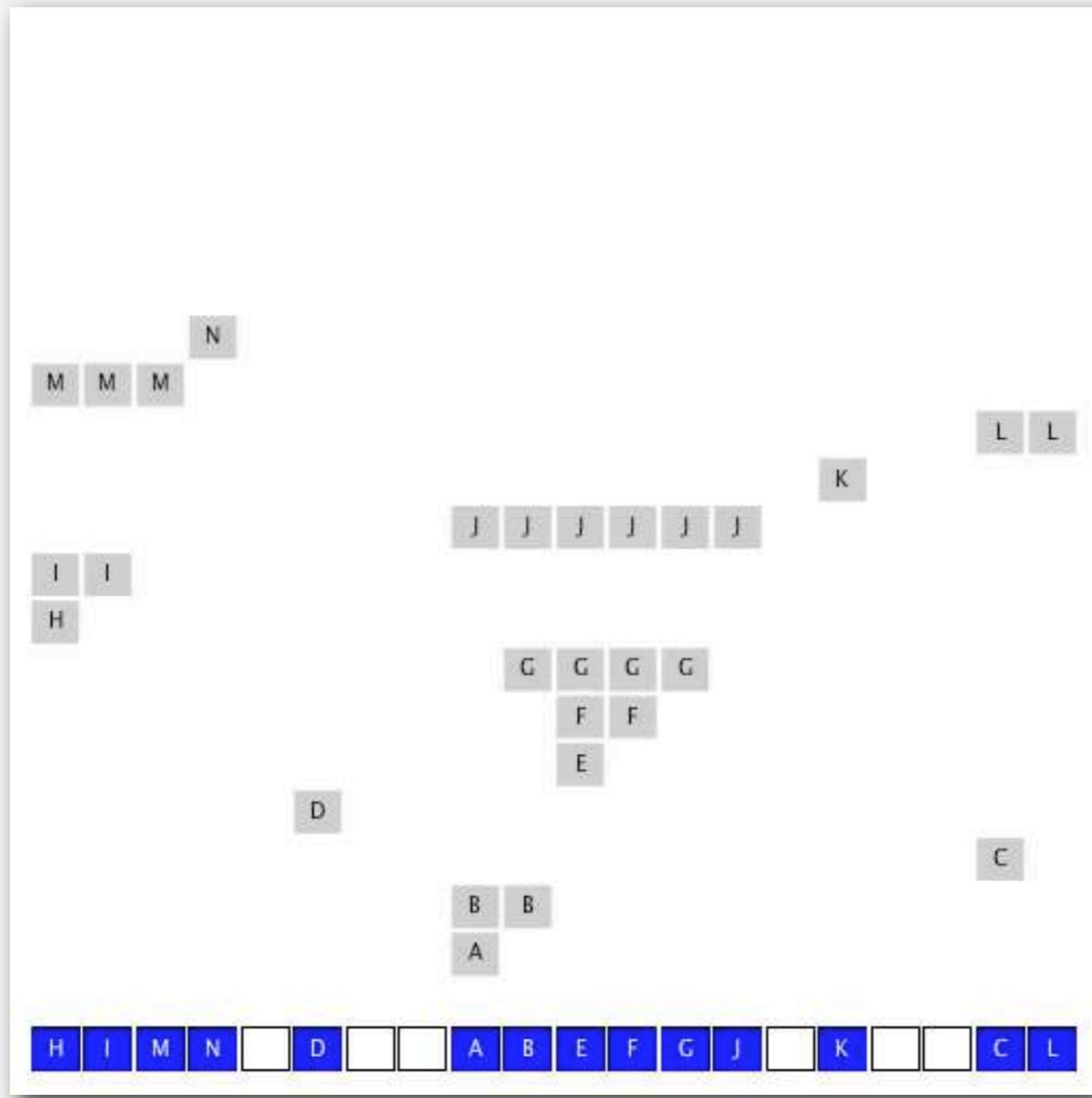
array doubling
and halving
code omitted



Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.

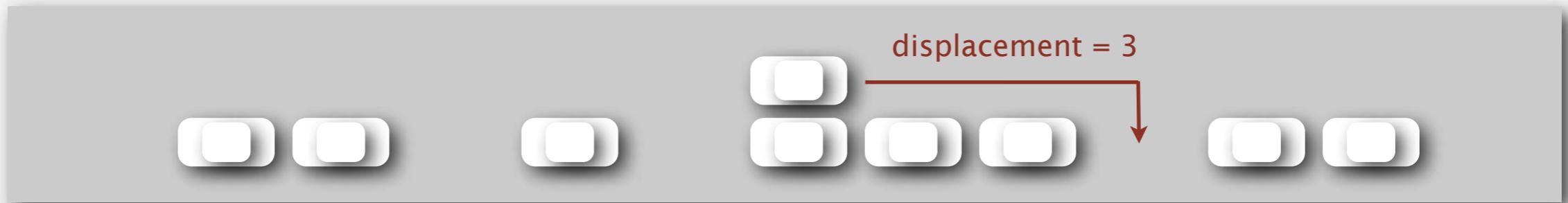


Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i + 1, i + 2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

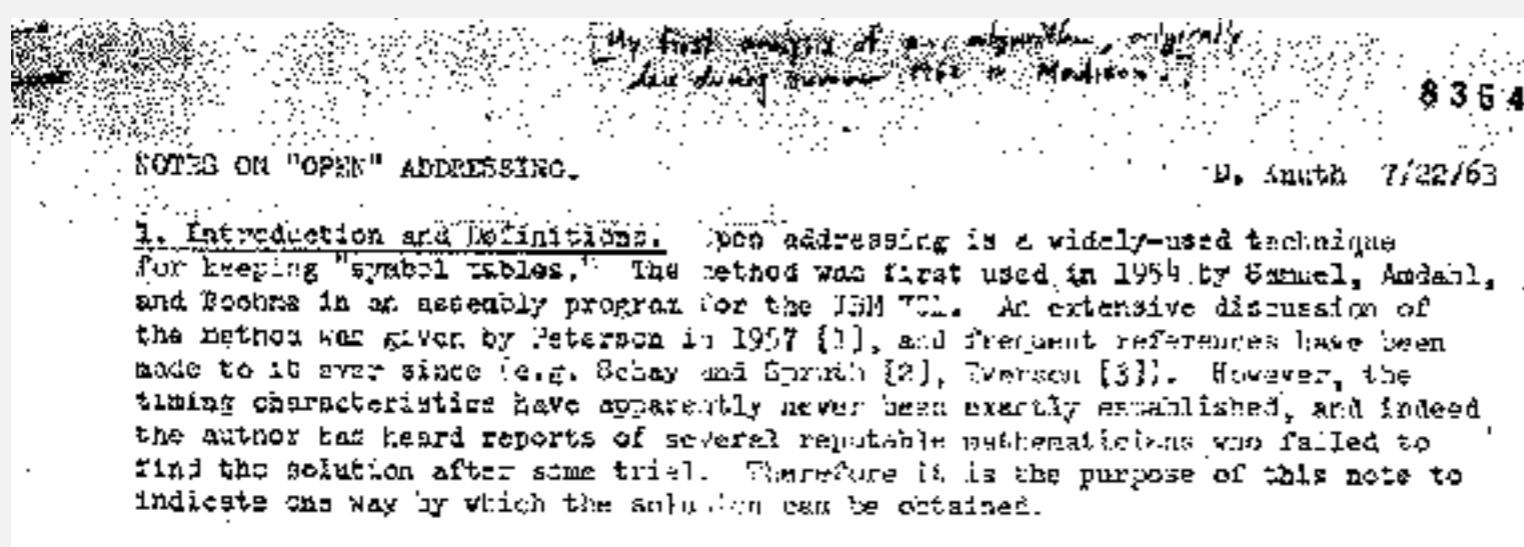
Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \quad \text{search hit}$$
$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right) \quad \text{search miss / insert}$$

Pf.



Parameters.

- M too large \Rightarrow too many empty array entries.
- M too small \Rightarrow search time blows up.
- Typical choice: $\alpha = N/M \sim 1/2$.
 # probes for search hit is about 3/2
probes for search miss is about 5/2

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$N/2$	N	$N/2$	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$N/2$	$N/2$	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
red-black tree	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes	<code>compareTo()</code>
separate chaining	N^*	N^*	N^*	$3-5^*$	$3-5^*$	$3-5^*$	no	<code>equals()</code>
linear probing	N^*	N^*	N^*	$3-5^*$	$3-5^*$	$3-5^*$	no	<code>equals()</code>

* under uniform hashing assumption

War story: String hashing in Java

String hashCode() in Java 1.1.

- For long strings: only examine 8-9 evenly spaced characters.
- Benefit: saves time in performing arithmetic.

```
public int hashCode()
{
    int hash = 0;
    int skip = Math.max(1, length() / 8);
    for (int i = 0; i < length(); i += skip)
        hash = s[i] + (37 * hash);
    return hash;
}
```

- Downside: great potential for bad collision patterns.

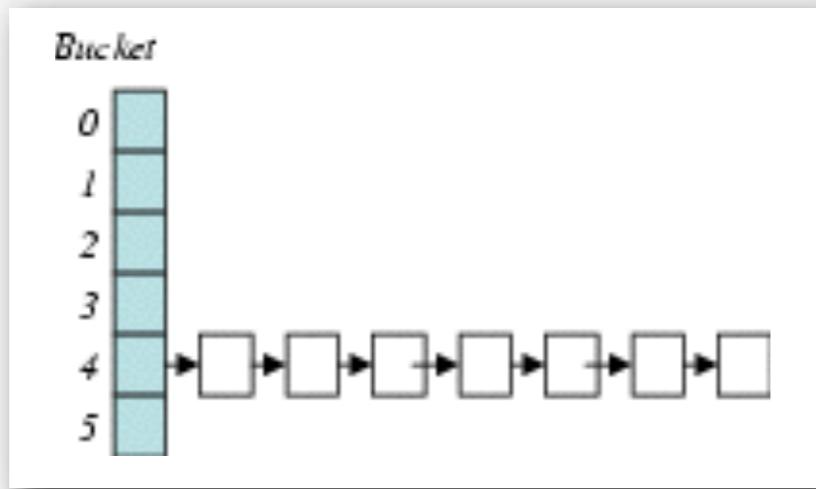
```
http://www.cs.princeton.edu/introcs/13loop>Hello.java
http://www.cs.princeton.edu/introcs/13loop>Hello.class
http://www.cs.princeton.edu/introcs/13loop>Hello.html
http://www.cs.princeton.edu/introcs/12type/index.html
↑      ↑      ↑      ↑      ↑      ↑      ↑      ↑
```

War story: algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?

A. Obvious situations: aircraft control, nuclear reactor, pacemaker.

A. Surprising situations: **denial-of-service** attacks.



malicious adversary learns your hash function
(e.g., by reading Java API) and causes a big pile-up
in single slot that grinds performance to a halt

Real-world exploits. [Crosby-Wallach 2003]

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem.
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hash code.

Solution. The base 31 hash code is part of Java's string API.

key	hashCode ()
"Aa"	2112
"BB"	2112

key	hashCode ()
"AaAaAaAa"	-540425984
"AaAaAaBB"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode ()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAA"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length 2N that hash to same value!

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160,

known to be insecure

```
String password = args[0];
MessageDigest sha1 =
MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Digital fingerprint, message digest, storing passwords.

Caveat. Too expensive for use in ST implementations.

Separate chaining vs. linear probing

Separate chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.

Q. How to delete?

Q. How to resize?

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate-chaining variant)

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Double hashing. (linear-probing variant)

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. (linear-probing variant)

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst case time for search.



Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

TODAY

- ▶ Hashing
- ▶ Search applications

SEARCH APPLICATIONS

- ▶ **Sets**
- ▶ **Dictionary clients**
- ▶ **Indexing clients**
- ▶ **Sparse vectors**

Set API

Mathematical set. A collection of distinct keys.

```
public class SET<Key extends Comparable<Key>>
```

SET()	<i>create an empty set</i>
void add(Key key)	<i>add the key to the set</i>
boolean contains(Key key)	<i>is the key in the set?</i>
void remove(Key key)	<i>remove the key from the set</i>
int size()	<i>return the number of keys in the set</i>
Iterator<Key> iterator()	<i>iterator through keys in the set</i>

Q. How to implement?

A. Remove “value” from any ST implementation

Exception filter

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
% more list.txt  
was it the of  
  
% java WhiteList list.txt < tinyTale.txt  
it was the of it was the of  
  
% java BlackList list.txt < tinyTale.txt  
best times worst times  
age wisdom age foolishness  
epoch belief epoch incredulity  
season light season darkness  
spring hope winter despair
```



list of exceptional words

Exception filter applications

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
public class WhiteList
{
    public static void main(String[] args)
    {
        SET<String> set = new SET<String>(); ←
        In in = new In(args[0]);
        while (!in.isEmpty())
            set.add(in.readString()); ←

        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (set.contains(word)) ←
                StdOut.println(word);
        }
    }
}
```

create empty set of strings

read in whitelist

print words not in list

Exception filter: Java implementation

- Read in a list of words from one file.
- Print out all words from standard input that are { in, not in } the list.

```
public class BlackList
{
    public static void main(String[] args)
    {
        SET<String> set = new SET<String>(); ←
        In in = new In(args[0]);
        while (!in.isEmpty())
            set.add(in.readString()); ←

        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (!set.contains(word))
                StdOut.println(word); ←
        }
    }
}
```

create empty set of strings

read in whitelist

print words not in list

SEARCH APPLICATIONS

- ▶ Sets
- ▶ Dictionary clients
- ▶ Indexing clients
- ▶ Sparse vectors

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex I. DNS lookup.

```
% java LookupCSV ip.csv 0 1
```

adobe.com
192.150.18.60

www.princeton.edu
128.112.128.15

ebay.edu
Not found


```
% java LookupCSV ip.csv 1 0
```

128.112.128.15
www.princeton.edu

999.999.999.99
Not found

URL is key **IP is value**

IP is key **URL is value**

```
% more ip.csv
```

www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.18.60
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...

Dictionary lookup

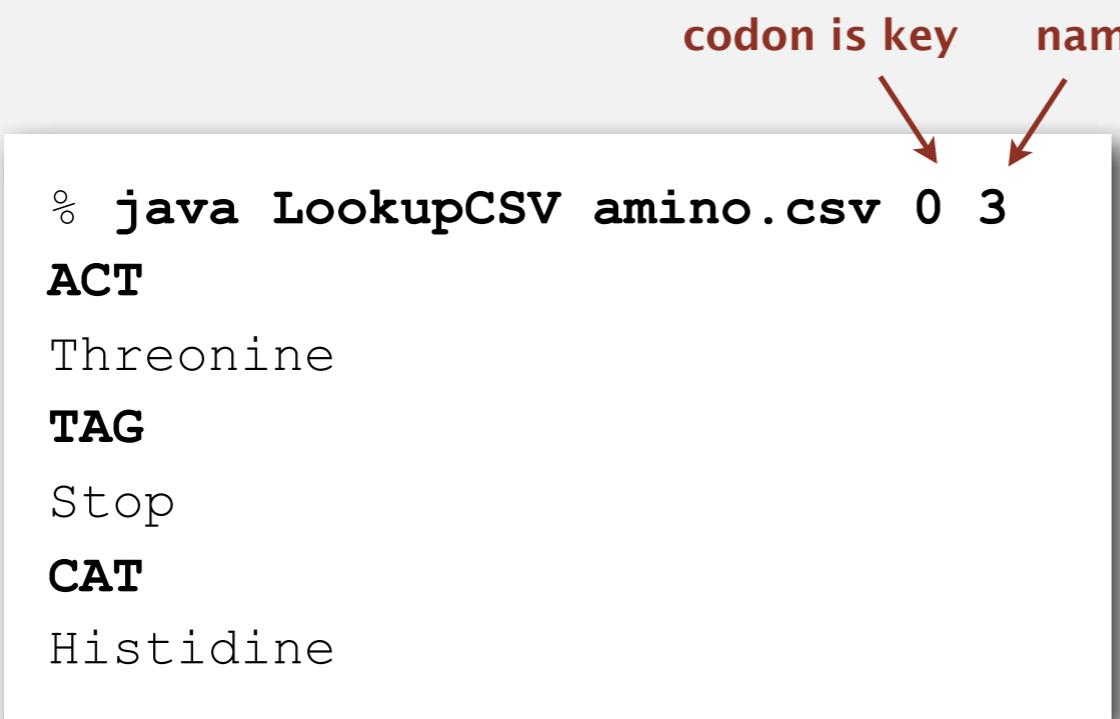
Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

```
% java LookupCSV amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine
```

codon is key **name is value**



```
% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
...
```

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 3. Class list.

```
% java LookupCSV classlist.csv 4 1  
eberl  
Ethan  
nwebb  
Natalie  
  
% java LookupCSV classlist.csv 4 3  
dpan  
P01
```

first name
login is key is value

precept
login is key is value

```
% more classlist.csv  
13,Berl,Ethan Michael,P01,eberl  
11,Bourque,Alexander Joseph,P01,abourque  
12,Cao,Phillips Minghua,P01,pcao  
11,Chehoud,Christel,P01,cchehoud  
10,Douglas,Malia Morioka,P01,malia  
12,Haddock,Sara Lynn,P01,shaddock  
12,Hantman,Nicole Samantha,P01,nhantman  
11,Hesterberg,Adam Classen,P01,ahesterb  
13,Hwang,Roland Lee,P01,rhwang  
13,Hyde,Gregory Thomas,P01,ghyde  
13,Kim,Hyunmoon,P01,hktwo  
11,Kleinfeld,Ivan Maximillian,P01,ikleinfel  
12,Korac,Damjan,P01,dkorac  
11,MacDonald,Graham David,P01,gmacdona  
10,Michal,Brian Thomas,P01,bmichal  
12,Nam,Seung Hyeon,P01,seungnam  
11,Nastasescu,Maria Monica,P01,mnastase  
11,Pan,Di,P01,dpan  
12,Partridge,Brenton Alan,P01,bpartrid  
13,Rilee,Alexander,P01,arilee  
13,Roopakalu,Ajay,P01,aroopaka  
11,Sheng,Ben C,P01,bsheng  
12,Webb,Natalie Sue,P01,nwebb  
...
```

Dictionary lookup: Java implementation

```
public class LookupCSV
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);
```

← process input file

```
        ST<String, String> st = new ST<String, String>();
        while (!in.isEmpty())
        {
            String line = in.readLine();
            String[] tokens = database[i].split(",");
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }
```

← build symbol table

```
        while (!StdIn.isEmpty())
        {
            String s = StdIn.readString();
            if (!st.contains(s)) StdOut.println("Not found");
            else StdOut.println(st.get(s));
        }
    }
}
```

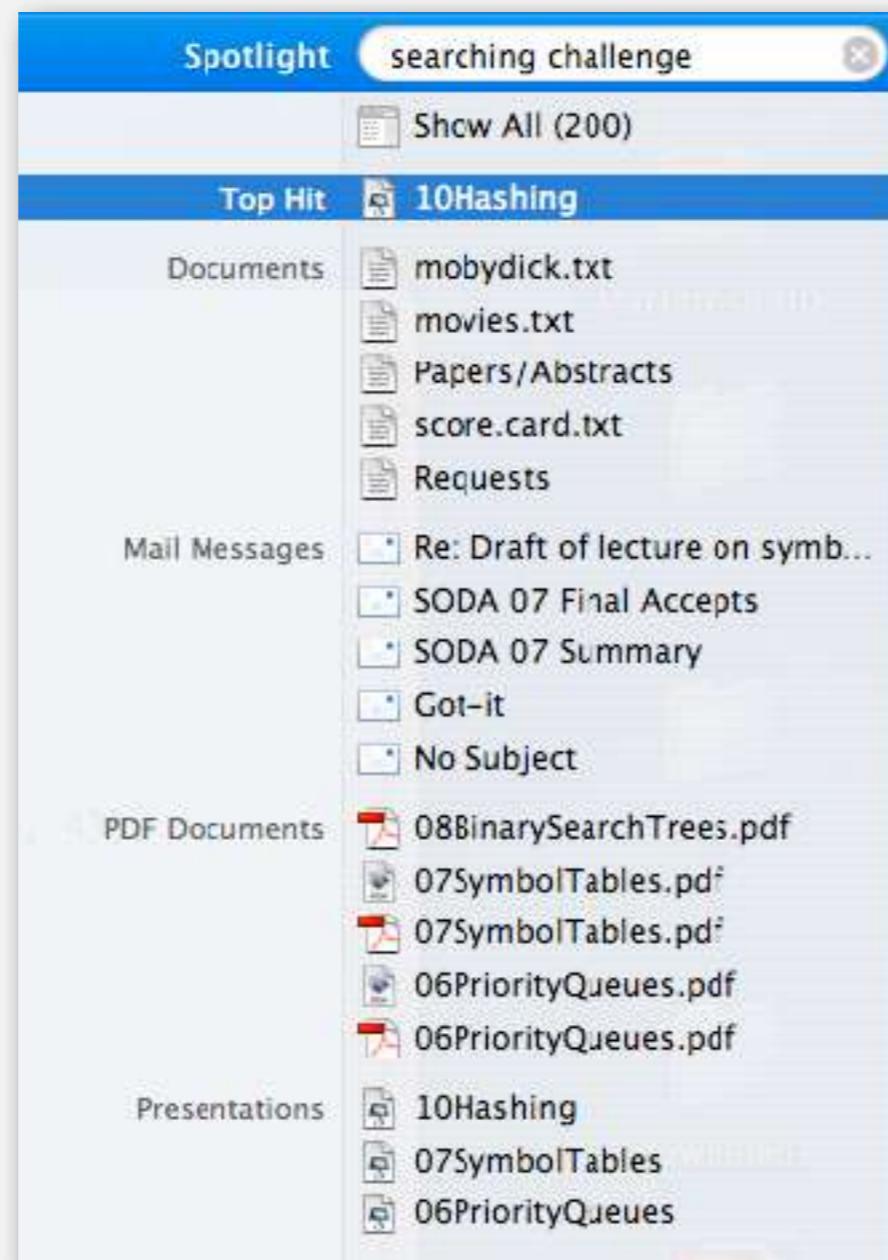
← process lookups
with standard I/O

SEARCH APPLICATIONS

- ▶ Sets
- ▶ Dictionary clients
- ▶ Indexing clients
- ▶ Sparse vectors

File indexing

Goal. Index a PC (or the web).



File indexing

Goal. Given a list of files specified, create an index so that you can efficiently find all files containing a given query string.

```
% ls *.txt  
aesop.txt magna.txt moby.txt  
sawyer.txt tale.txt  
  
% java FileIndex *.txt  
freedom  
magna.txt moby.txt tale.txt  
  
whale  
moby.txt  
  
lamb  
sawyer.txt aesop.txt
```

```
% ls *.java  
  
% java FileIndex *.java  
BlackList.java Concordance.java  
DeDup.java FileIndex.java ST.java  
SET.java WhiteList.java  
  
import  
FileIndex.java SET.java ST.java  
  
Comparator  
null
```

File indexing

Goal. Given a list of files specified, create an index so that you can efficiently find all files containing a given query string.

```
% ls *.txt  
aesop.txt magna.txt moby.txt  
sawyer.txt tale.txt  
  
% java FileIndex *.txt  
freedom  
magna.txt moby.txt tale.txt  
  
whale  
moby.txt  
  
lamb  
sawyer.txt aesop.txt
```

```
% ls *.java  
  
% java FileIndex *.java  
BlackList.java Concordance.java  
DeDup.java FileIndex.java ST.java  
SET.java WhiteList.java  
  
import  
FileIndex.java SET.java ST.java  
  
Comparator  
null
```

Solution. Key = query string; value = set of files containing that string.

File indexing

```
public class FileIndex
{
    public static void main(String[] args)
    {
        ST<String, SET<File>> st = new ST<String, SET<File>>(); ← symbol table

        for (String filename : args) { ← list of file names
            File file = new File(filename);
            In in = new In(file);
            while !(in.isEmpty())
            {
                String word = in.readString();
                if (!st.contains(word))
                    st.put(s, new SET<File>());
                SET<File> set = st.get(key);
                set.add(file); ← for each word in
                                file, add file to
                                corresponding set
            }
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            StdOut.println(st.get(query)); ← process queries
        }
    }
}
```

Goal. Index for an e-book.

Index

Abstract data type (ADT), 127-198
abstract classes, 163
classes, 129-136
collections of items, 127-133
creating, 157-164
defined, 128
duplicate items, 173-176
equivalence-relations, 159-162
FIFO queues, 163-171
first-class, 177-186
generic operations, 274
index items, 177
insert/remove operations, 138-139
modulo programming, 135
polynomial, 188-192
priority queues, 373-376
pushdown stack, 158-159
stubs, 155
symbol table, 497-506
ADT interfaces
array (`myArray`), 274
complex number (`Complex`), 181
existence table (ET), 663
full priority queue (PQfull), 397
indirect priority queue (PQI), 403
item (`myItem`), 273, 498
key (`myKey`), 498
polynomial (`Poly`), 189
print (`Point`), 124
priority queue (PQ), 375
queue of int (`intQueue`), 166

stack of int (`intStack`), 140
symbol table (ST), 503
text index (TI), 525
union-find (UF), 159
Abstract in-place merging, 351-353
Abstract operation, 10
Access control state, 131
Actual data, 31
Adapter class, 155-157
Adaptive sort, 268
Address, 84-85
Adjacency list, 120-123
depth-first search, 251-256
Adjacency matrix, 120-122
Ajtai, M., 464
Algorithm, 4-6, 27-64
abstract operations, 10, 31, 34-35
analyses of, 6
average/worst-case performance, 35, 60-62
big-O notation, 44-47
binary search, 51-59
computational complexity, 62-64
efficiency, 6, 30, 52
empirical analysis, 30-32, 58
exponential time, 219
implementation, 28-30
logarithmic function, 40-43
mathematical analysis, 33-36, 58
primary parameter, 36
probabilistic, 331
recurrences, 49-52, 57
recursive, 198
running time, 34-40
search, 53-56, 495
steps in, 22-23
See also Randomized algorithm
Amortization approach, 557, 627
Arithmetic operators, 177-179, 188, 191
Array, 12, 83
binary search, 57
dynamic allocation, 87
and linked lists, 92, 92-93
merging, 349-350
multidimensional, 117-118
references, 86-87, 89
sorting, 265-267, 273-276
and strings, 119
two-dimensional, 117-118, 122-124
vectors, 87
visualizations, 295
See also index, array
Array representation
binary tree, 38
bit O queue, 168-169
linked lists, 110
polynomial ADT, 191-192
priority queue, 377-378, 403, 406
pushdown stack, 158-159
random queue, 170
symbol table, 508, 511-512, 521
Asymptotic expression, 45-46
Average deviation, 80-81
Average-case performance, 35, 60-61
AVL tree, 583
B tree, 584, 692-704
external/internal pages, 695
4-5-6-7-8 tree, 693-704
Vaidyanathan, 701
various, 701-703
search/insert, 597-701
select/sort, 701
Balanced tree, 238, 555-598
B tree, 584
bottom up, 376, 384-385
height-balanced, 383
indexed sequential access, 690-692
performance, 575-576, 581-582, 595-598
randomized, 559-564
red-black, 577-585
skip lists, 587-591
splay, 566-571

Concordance

Goal. Preprocess a text corpus to support concordance queries: given a word, find all occurrences with their immediate contexts.

```
% java Concordance tale.txt  
cities  
tongues of the two *cities* that were blended in  
  
majesty  
their turnkeys and the *majesty* of the law fired  
me treason against the *majesty* of the people in  
of his most gracious *majesty* king george the third  
  
princeton  
no matches
```

Concordance

```
public class Concordance
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        String[] words = StdIn.readAll().split("\s+");
        ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
        for (int i = 0; i < words.length; i++)
        {
            String s = words[i];
            if (!st.contains(s))
                st.put(s, new SET<Integer>());
            SET<Integer> pages = st.get(s);
            set.put(i);
        }
    }

    while (!StdIn.isEmpty())
    {
        String query = StdIn.readString();
        SET<Integer> set = st.get(query);
        for (int k : set)
            // print words[k-5] to words[k+5]
        }
    }
}
```

← read text and build index

← process queries and print concordances

SEARCH APPLICATIONS

- ▶ **Sets**
- ▶ **Dictionary clients**
- ▶ **Indexing clients**
- ▶ **Sparse vectors**

Vectors and matrices

Vector. Ordered sequence of N real numbers.

Matrix. N-by-N table of real numbers.

vector operations

$$a = [0 \ 3 \ 15], \quad b = [-1 \ 2 \ 2]$$

$$a + b = [-1 \ 5 \ 17]$$

$$a \circ b = (0 \cdot -1) + (3 \cdot 2) + (15 \cdot 2) = 36$$

$$|a| = \sqrt{a \circ a} = \sqrt{0^2 + 3^2 + 15^2} = 3\sqrt{26}$$

matrix-vector multiplication

$$\begin{bmatrix} 0 & 1 & 1 \\ 2 & 4 & -2 \\ 0 & 3 & 15 \end{bmatrix} \times \begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 36 \end{bmatrix}$$

Sparse vectors and matrices

Sparse vector. An N -dimensional vector is **sparse** if it contains $O(1)$ nonzeros.

Sparse matrix. An N -by- N matrix is **sparse** if it contains $O(N)$ nonzeros.

Property. Large matrices that arise in practice are sparse.

$$[\begin{array}{ccccc} 0 & 0 & .36 & .36 & .18 \end{array}]$$

$$\left[\begin{array}{ccccc} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{array} \right]$$

Matrix-vector multiplication (standard implementation)

$$\begin{array}{c} \text{a}[][] \\ \left[\begin{array}{ccccc} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{array} \right] \end{array} \begin{array}{c} \text{x}[] \\ \left[\begin{array}{c} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{array} \right] \end{array} = \begin{array}{c} \text{b}[] \\ \left[\begin{array}{c} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{array} \right] \end{array}$$

```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// initialize a[][] and x[]
...
for (int i = 0; i < N; i++)
{
    sum = 0.0;
    for (int j = 0; j < N; j++)
        sum += a[i][j]*x[j];
    b[i] = sum;
}
```

nested loops
(N^2 running time)

Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.

Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10 .

A sparse matrix A is shown as a grid of dots. Most entries are black, representing zeros, while a few are blue, representing non-zero values. To the right of the matrix, the multiplication operator $*$ and the assignment operator $=$ are shown in red. To the right of the assignment operator is the result vector b , which consists of a vertical column of alternating blue and black dots.

$$\mathbf{A} \quad * \quad \mathbf{x} \quad = \quad \mathbf{b}$$

Vector representations

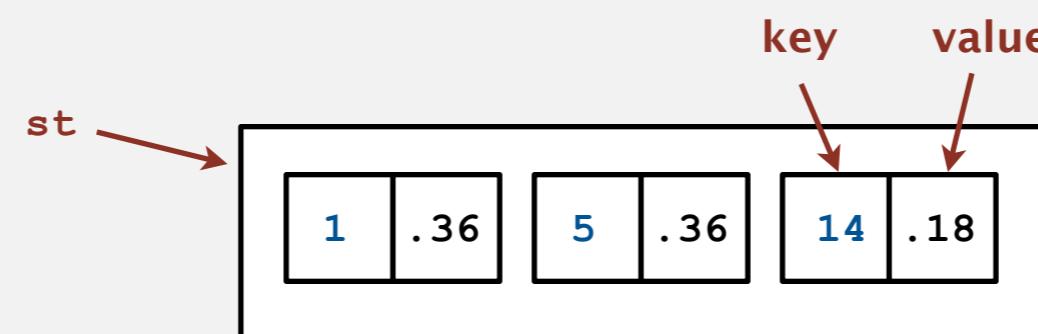
1D array (standard) representation.

- Constant time access to elements.
- Space proportional to N.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	.36	0	0	0	.36	0	0	0	0	0	0	0	0	.18	0	0	0	0	0

Symbol table representation.

- Key = index, value = entry.
- Efficient iterator.
- Space proportional to number of nonzeros.



Sparse vector data type

```
public class SparseVector
{
    private HashST<Integer, Double> v; ← HashST because order not important

    public SparseVector()
    {   v = new HashST<Integer, Double>();   } ← empty ST represents all 0s vector

    public void put(int i, double x)
    {   v.put(i, x);   } ← a[i] = value

    public double get(int i)
    {
        if (!v.contains(i)) return 0.0;
        else return v.get(i); ← return a[i]
    }

    public Iterable<Integer> indices()
    {   return v.keys();   }

    public double dot(double[] that)
    {
        double sum = 0.0;
        for (int i : indices())
            sum += that[i]*this.get(i);
        return sum;
    }
}
```

dot product is constant time for sparse vectors

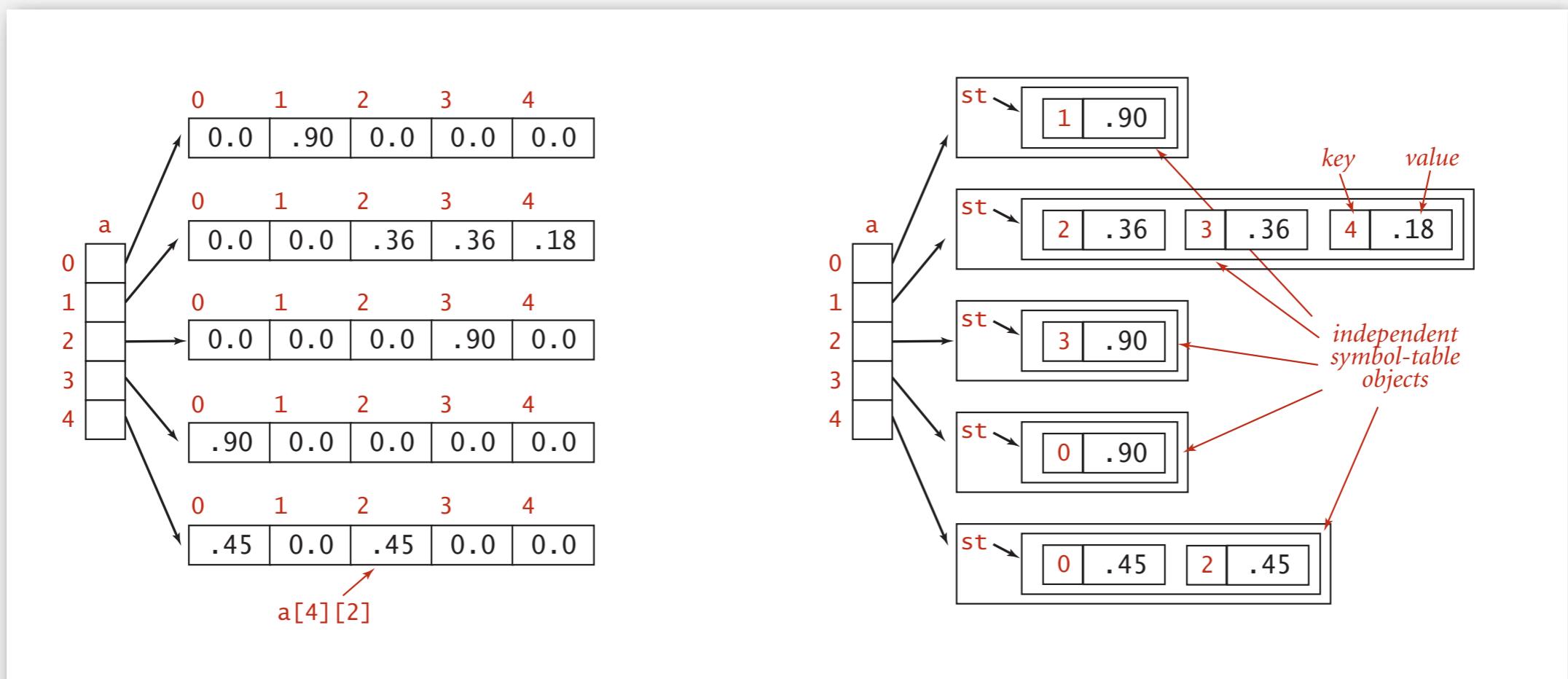
Matrix representations

2D array (standard) matrix representation: Each row of matrix is an **array**.

- Constant time access to elements.
- Space proportional to N^2 .

Sparse matrix representation: Each row of matrix is a **sparse vector**.

- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).



Sparse matrix-vector multiplication

$$\begin{array}{c} \text{a[][]} \\ \left[\begin{array}{ccccc} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{array} \right] \\ \text{x[]} \\ \left[\begin{array}{c} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{array} \right] \\ = \\ \text{b[]} \\ \left[\begin{array}{c} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{array} \right] \end{array}$$

```
..  
SparseVector[] a = new SparseVector[N];  
double[] x = new double[N];  
double[] b = new double[N];  
..  
// Initialize a[] and x[]  
..  
for (int i = 0; i < N; i++)  
    b[i] = a[i].dot(x);
```

linear running time
for sparse matrix

Sample searching challenge

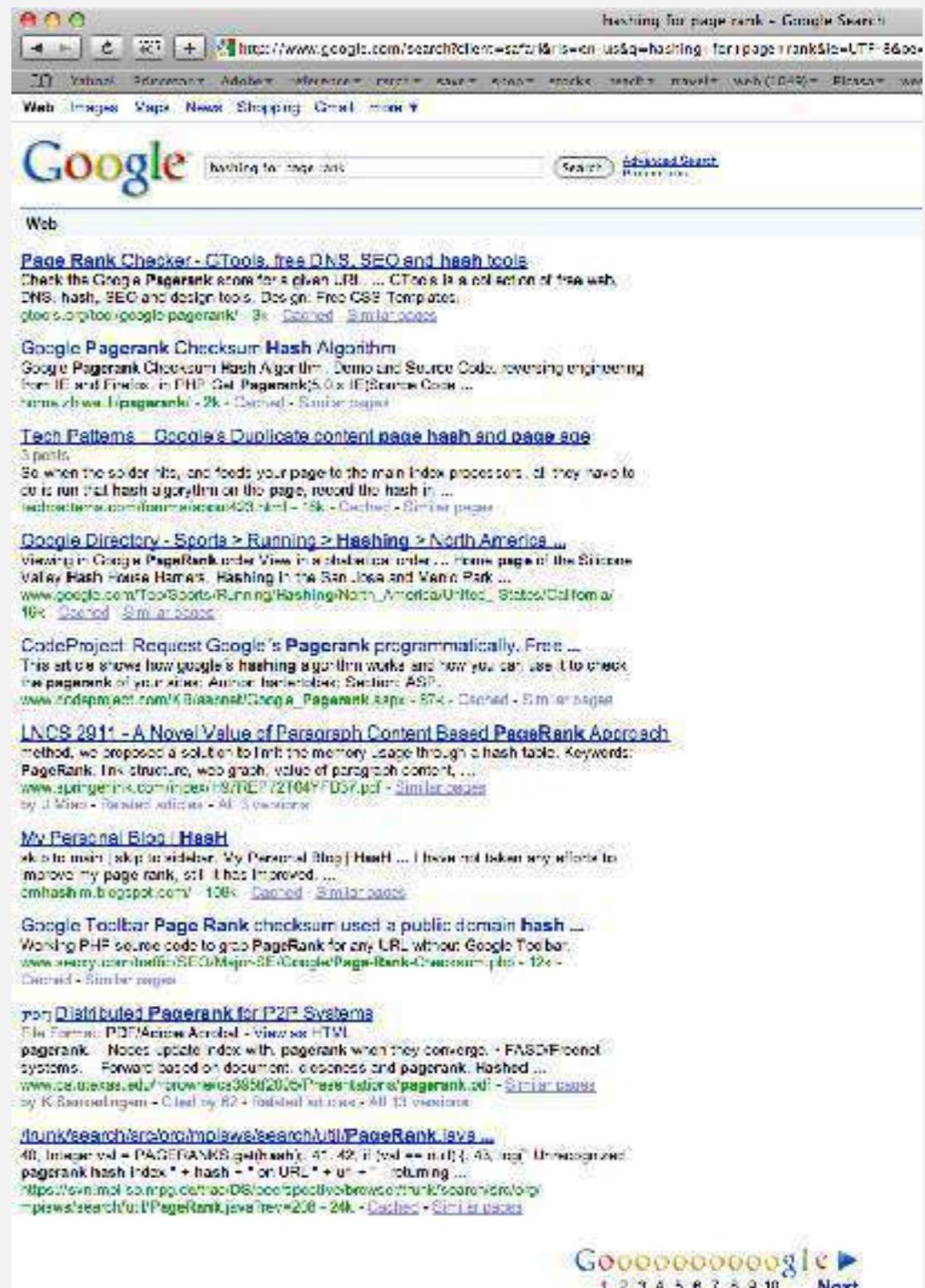
Problem. Rank pages on the web.

Assumptions.

- Matrix-vector multiply
- 10 billion+ rows
- sparse

Which “searching” method to use to access array values?

1. Standard 2D array representation
2. Symbol table
3. Doesn’t matter much.



Sample searching challenge

Problem. Rank pages on the web.

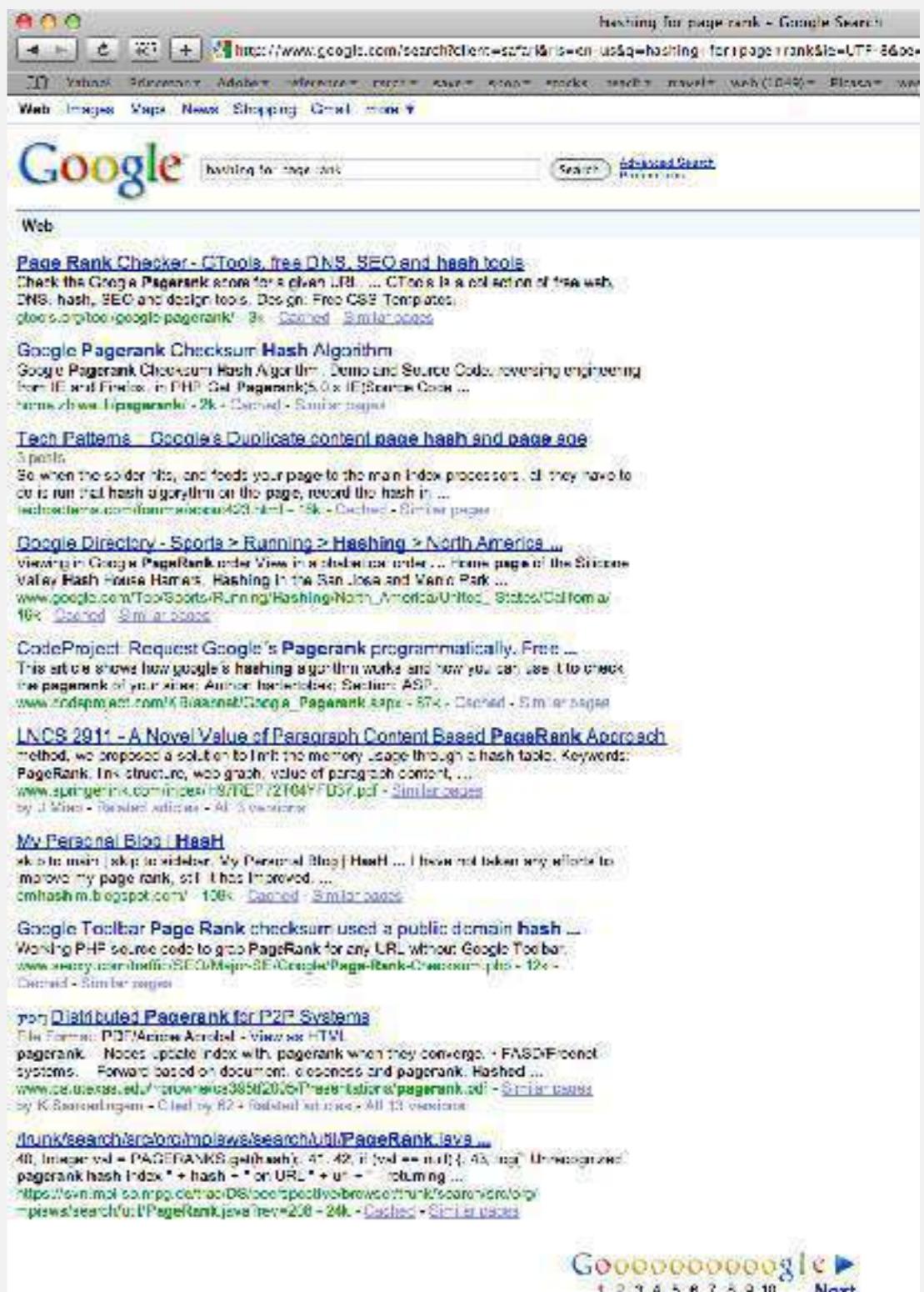
Assumptions.

- Matrix-vector multiply
- 10 billion+ rows
- sparse

Which “searching” method to use to access array values?

-
- ~~1. Standard 2D array representation~~
 - 2. Symbol table
 - 3. Doesn't matter much.

cannot be done
without fast
algorithm



Sparse vector data type

```
public class SparseVector
{
    private int N;                      // length
    private ST<Integer, Double> st;    // the elements

    public SparseVector(int N)
    {
        this.N = N;
        this.st = new ST<Integer, Double>();
    }

    public void put(int i, double value)
    {
        if (value == 0.0) st.remove(i);
        else             st.put(i, value);
    }

    public double get(int i)
    {
        if (st.contains(i)) return st.get(i);
        else                 return 0.0;
    }

    ...
}
```

← all 0s vector

← $a[i] = value$

← return $a[i]$

Sparse vector data type (cont)

```
public double dot(SparseVector that)
{
    double sum = 0.0;
    for (int i : this.st)
        if (that.st.contains(i))
            sum += this.get(i) * that.get(i);
    return sum;
}
```



dot product

```
public double norm()
{   return Math.sqrt(this.dot(this)); }
```



2-norm

```
public SparseVector plus(SparseVector that)
{
    SparseVector c = new SparseVector(N);
    for (int i : this.st)
        c.put(i, this.get(i));
    for (int i : that.st)
        c.put(i, that.get(i) + c.get(i));
    return c;
}
```



vector
sum

Sparse matrix data type

```
public class SparseMatrix
{
    private final int N;           // length
    private SparseVector[] rows;   // the elements

    public SparseMatrix(int N)
    {
        this.N = N;
        this.rows = new SparseVector[N];
        for (int i = 0; i < N; i++)
            this.rows[i] = new SparseVector(N);
    }

    public void put(int i, int j, double value)      ← a[i][j] = value
    {   rows[i].put(j, value);   }

    public double get(int i, int j)                   ← return a[i][j]
    {   return rows[i].get(j);   }

    public SparseVector times(SparseVector x)
    {
        SparseVector b = new SparseVector(N);
        for (int i = 0; i < N; i++)
            b.put(i, rows[i].dot(x));
        return b;
    }
}
```

all 0s matrix

$a[i][j] = \text{value}$

$\text{return } a[i][j]$

matrix-vector multiplication

Compressed row storage (CRS)

Compressed row storage.

- Store nonzeros in a 1D array `val[]`.
- Store column index of each nonzero in parallel 1D array `col[]`.
- Store first index of each row in array `row[]`.

$$A = \begin{bmatrix} 11 & 0 & 0 & 41 \\ 0 & 22 & 0 & 0 \\ 0 & 0 & 33 & 43 \\ 14 & 0 & 34 & 44 \\ 0 & 25 & 0 & 0 \\ 16 & 26 & 36 & 46 \end{bmatrix}$$

i	row[]
0	0
1	2
2	3
3	5
4	8
5	9
6	13

i	col[]	val[]
0	1	11
1	4	41
2	2	22
3	3	33
4	4	43
5	1	14
6	3	34
7	4	44
8	2	25
9	1	16
10	2	26
11	3	36
12	4	46

Compressed row storage (CRS)

Benefits.

- Cache-friendly.
- Space proportional to number of nonzeros.
- Very efficient matrix-vector multiply.

```
double[] y = new double[N];
for (int i = 0; i < n; i++)
    for (int j = row[i]; j < row[i+1]; j++)
        y[i] += val[j] * x[col[j]];
```

Downside. No easy way to add/remove nonzeros.

Applications. Sparse Matlab.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

UNDIRECTED GRAPHS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

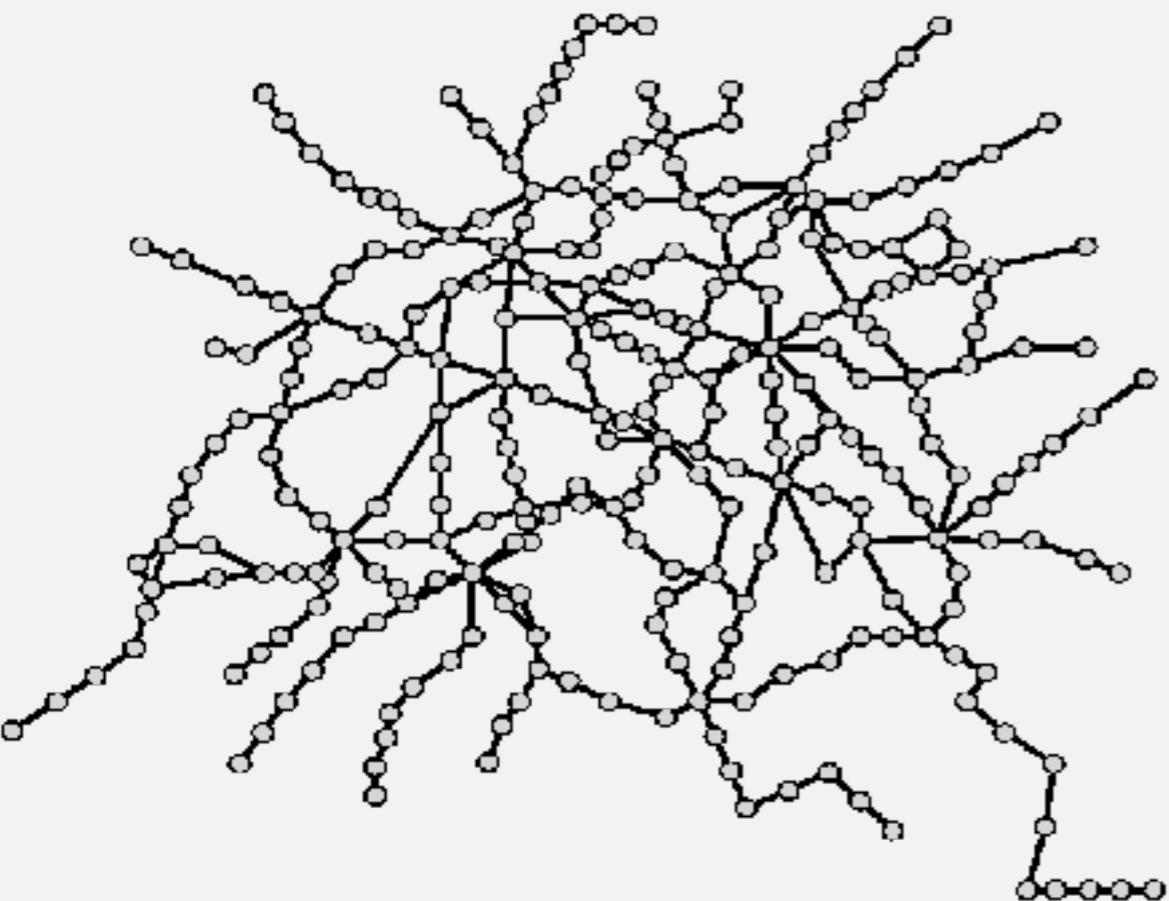
- ▶ **Undirected Graphs**
- ▶ **Graph API**
- ▶ **Depth-first search**
- ▶ **Breadth-first search**
- ▶ **Connected components**
- ▶ **Challenges**

Undirected graphs

Graph. Set of **vertices** connected pairwise by **edges**.

Why study graph algorithms?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of computer science and discrete math.



Graph applications

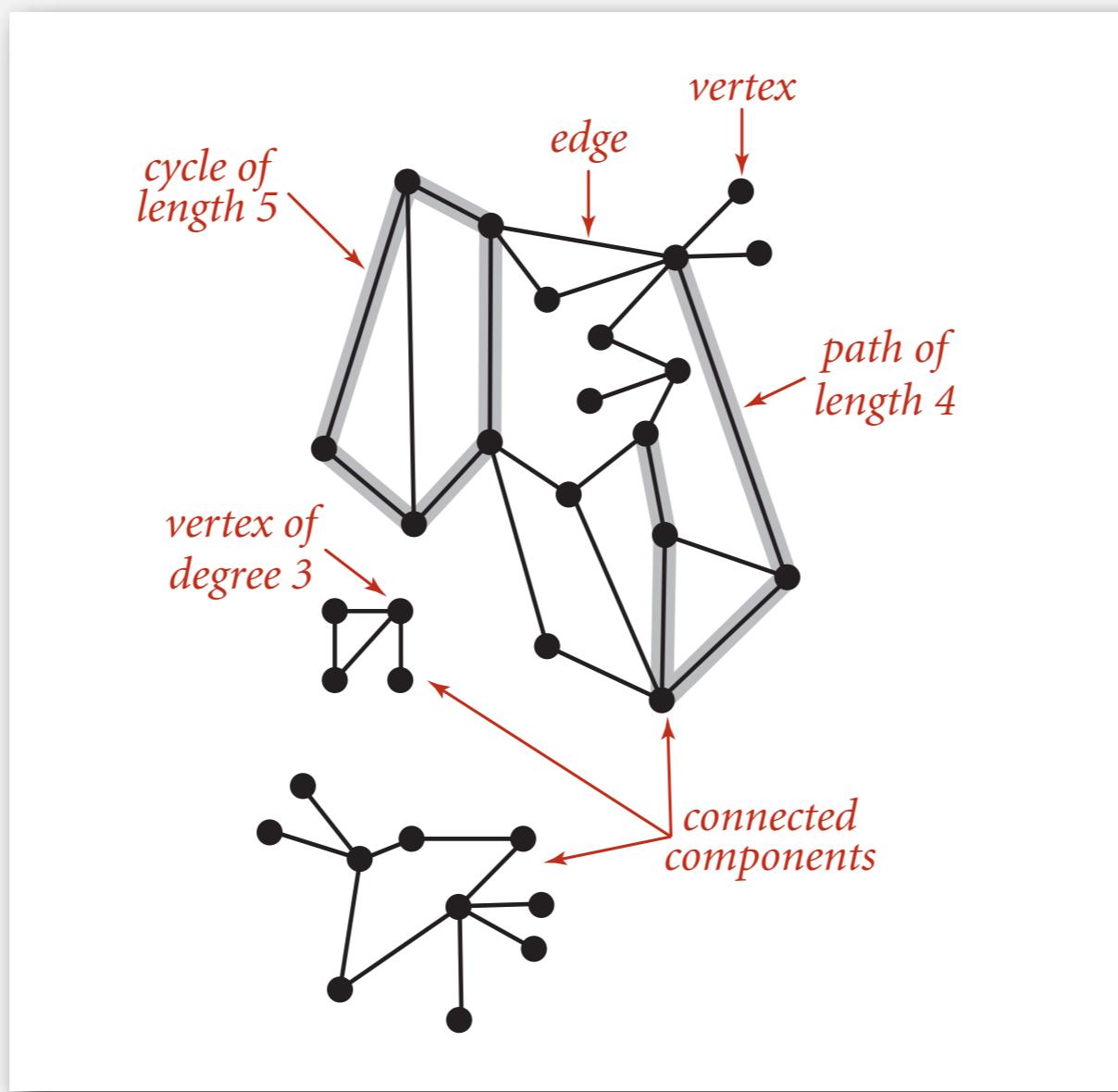
graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
chemical compound	molecule	bond

Graph terminology

Path. Sequence of vertices connected by edges.

Cycle. Path whose first and last vertices are the same.

Two vertices are **connected** if there is a path between them.



Some graph-processing problems

Path. Is there a path between s and t ?

Shortest path. What is the shortest path between s and t ?

Cycle. Is there a cycle in the graph?

Euler tour. Is there a cycle that uses each edge exactly once?

Hamilton tour. Is there a cycle that uses each vertex exactly once?

Connectivity. Is there a way to connect all of the vertices?

MST. What is the best way to connect all of the vertices?

Biconnectivity. Is there a vertex whose removal disconnects the graph?

Planarity. Can you draw the graph in the plane with no crossing edges?

Graph isomorphism. Do two adjacency lists represent the same graph?

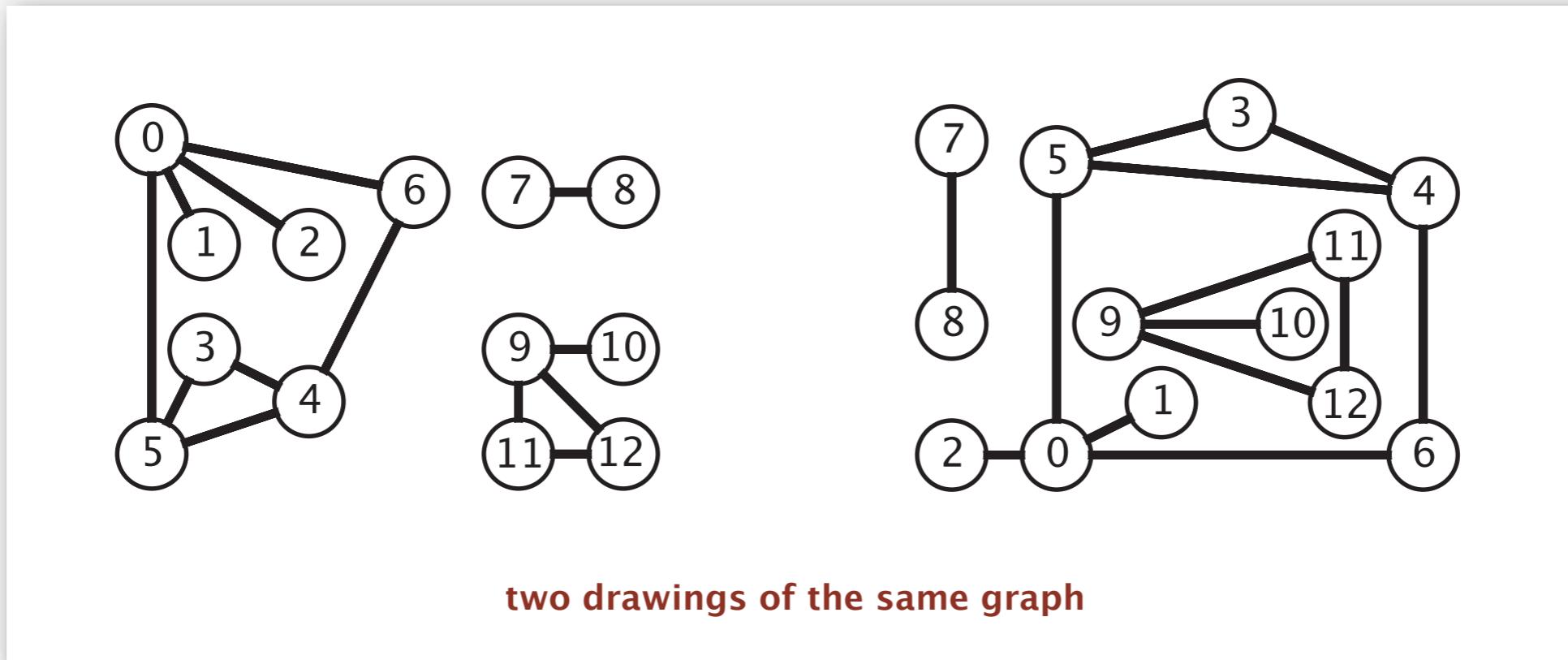
Challenge. Which of these problems are easy? difficult? intractable?

UNDIRECTED GRAPHS

- ▶ Graph API
- ▶ Depth-first search
- ▶ Breadth-first search
- ▶ Connected components
- ▶ Challenges

Graph representation

Graph drawing. Provides intuition about the structure of the graph.

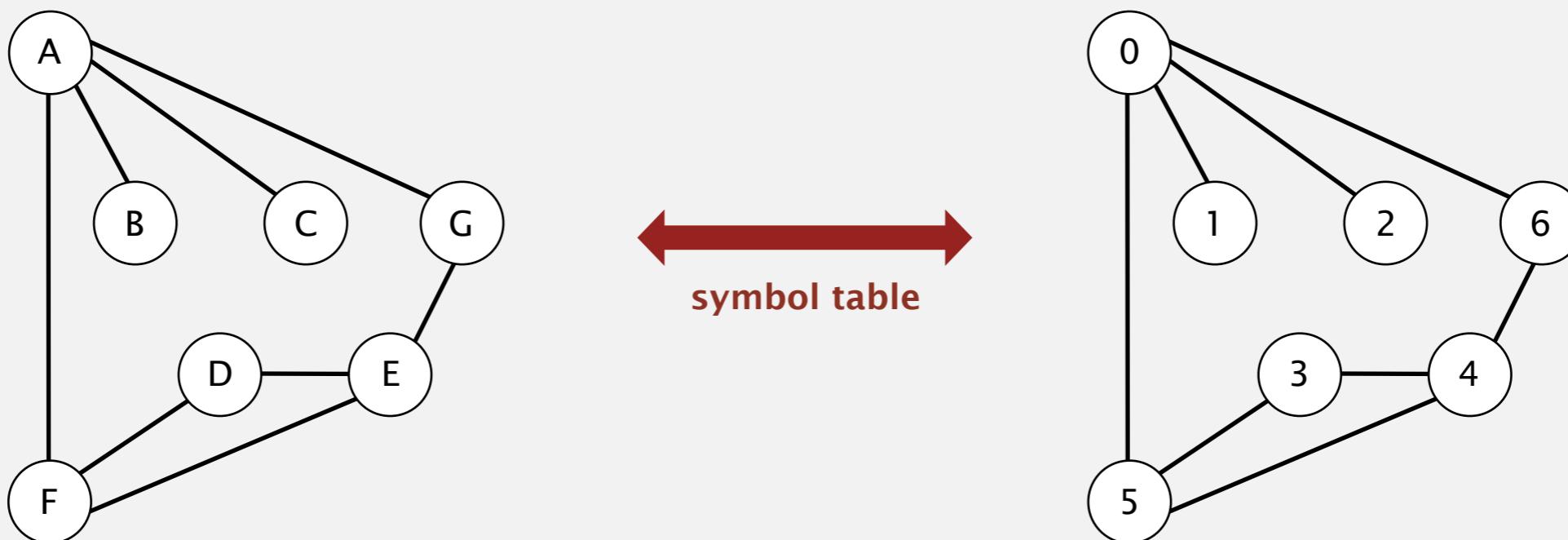


Caveat. Intuition can be misleading.

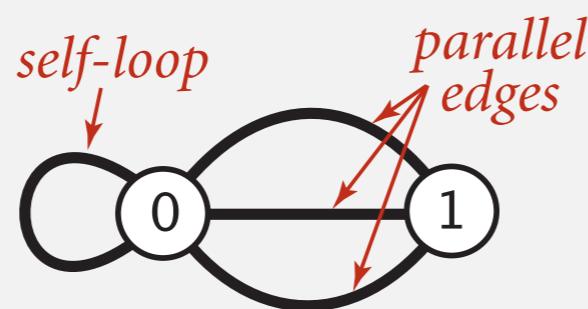
Graph representation

Vertex representation.

- This lecture: use integers between 0 and $V - 1$.
- Applications: convert between names and integers with symbol table.



Anomalies.



Graph API

```
public class Graph
```

```
    Graph(int V)
```

create an empty graph with V vertices

```
    Graph(In in)
```

create a graph from input stream

```
    void addEdge(int v, int w)
```

add an edge v-w

```
    Iterable<Integer> adj(int v)
```

vertices adjacent to v

```
    int V()
```

number of vertices

```
    int E()
```

number of edges

```
    String toString()
```

string representation

```
In in = new In(args[0]);
Graph G = new Graph(in);

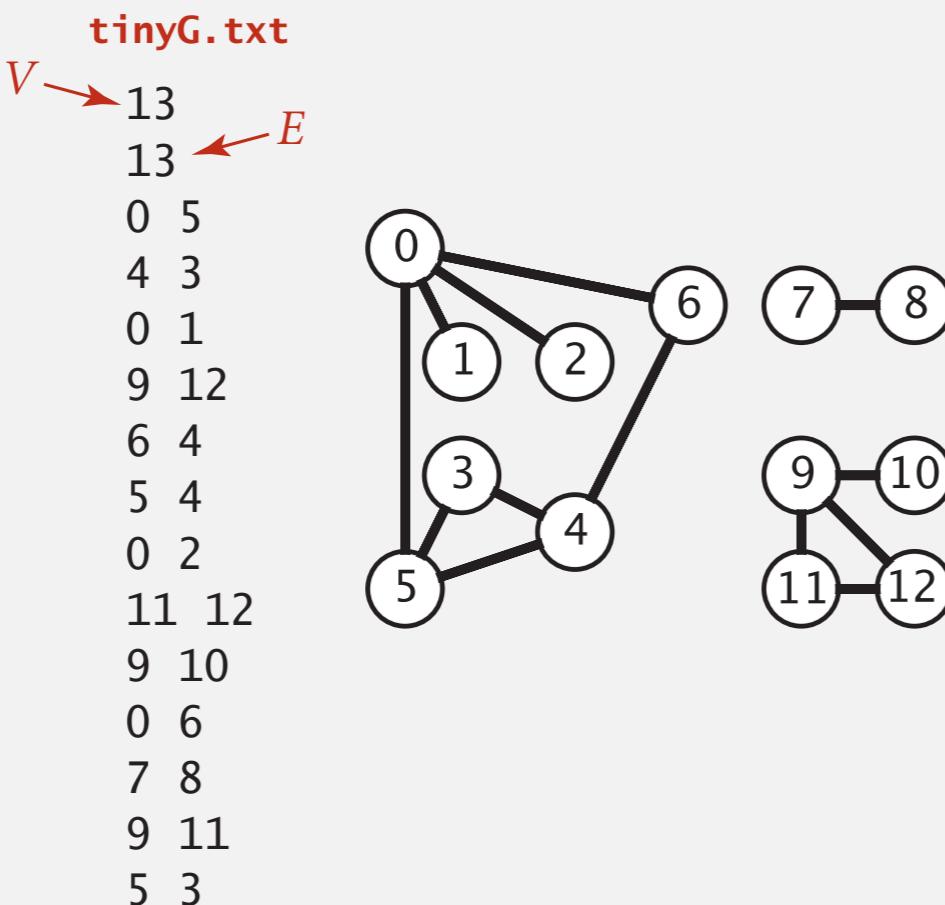
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "-" + w);
```

**read graph from
input stream**

**print out each
edge (twice)**

Graph API: sample client

Graph input format.



```
% java Test tinyG.txt  
0-6  
0-2  
0-1  
0-5  
1-0  
2-0  
3-5  
3-4  
...  
12-11  
12-9
```

```
In in = new In(args[0]);  
Graph G = new Graph(in);  
  
for (int v = 0; v < G.V(); v++)  
    for (int w : G.adj(v))  
        StdOut.println(v + "-" + w);
```

read graph from
input stream

print out each
edge (twice)

Typical graph-processing code

compute the degree of v

```
public static int degree(Graph G, int v)
{
    int degree = 0;
    for (int w : G.adj(v)) degree++;
    return degree;
}
```

compute maximum degree

```
public static int maxDegree(Graph G)
{
    int max = 0;
    for (int v = 0; v < G.V(); v++)
        if (degree(G, v) > max)
            max = degree(G, v);
    return max;
}
```

compute average degree

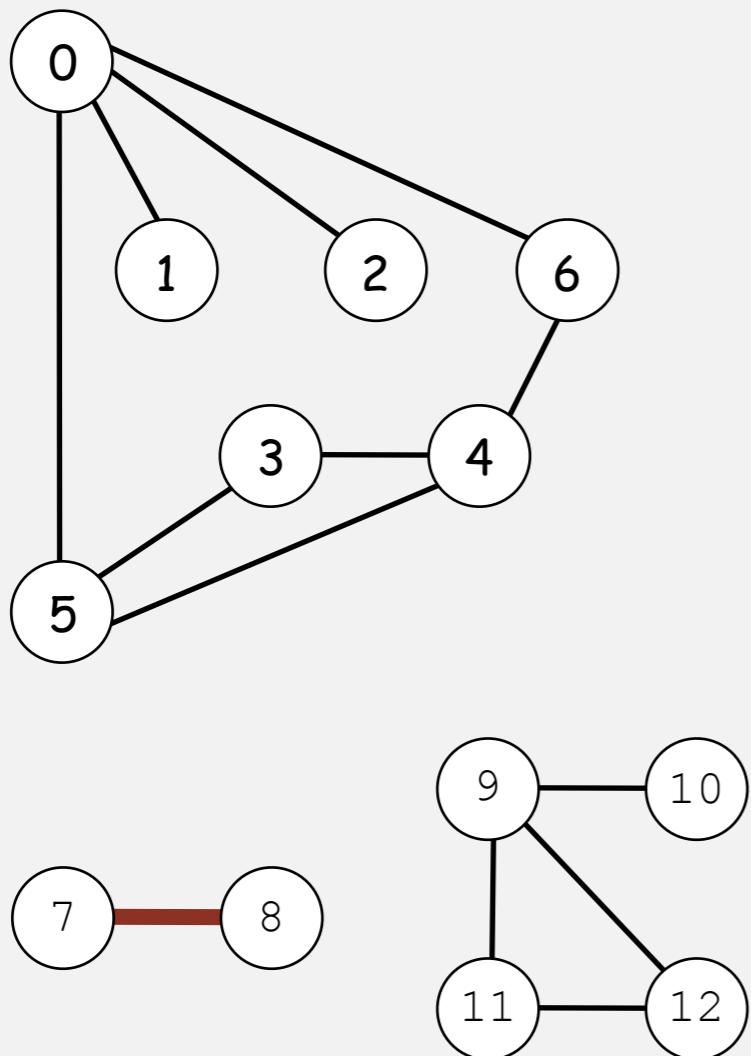
```
public static double averageDegree(Graph G)
{   return 2.0 * G.E() / G.V(); }
```

count self-loops

```
public static int numberOfSelfLoops(Graph G)
{
    int count = 0;
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            if (v == w) count++;
    return count/2; // each edge counted twice
}
```

Set-of-edges graph representation

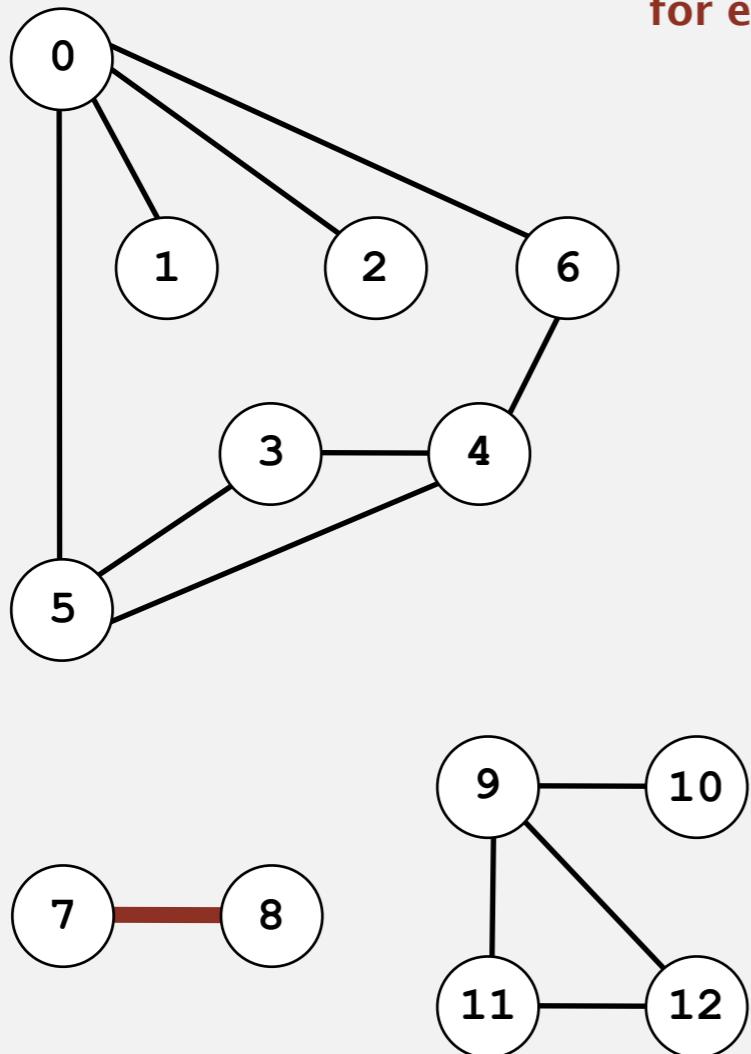
Maintain a list of the edges (linked list or array).



0	1
0	2
0	5
0	6
3	4
3	5
4	5
4	6
7	8
9	10
9	11
9	12
11	12

Adjacency-matrix graph representation

Maintain a two-dimensional V -by- V boolean array;
for each edge $v-w$ in graph: $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$.

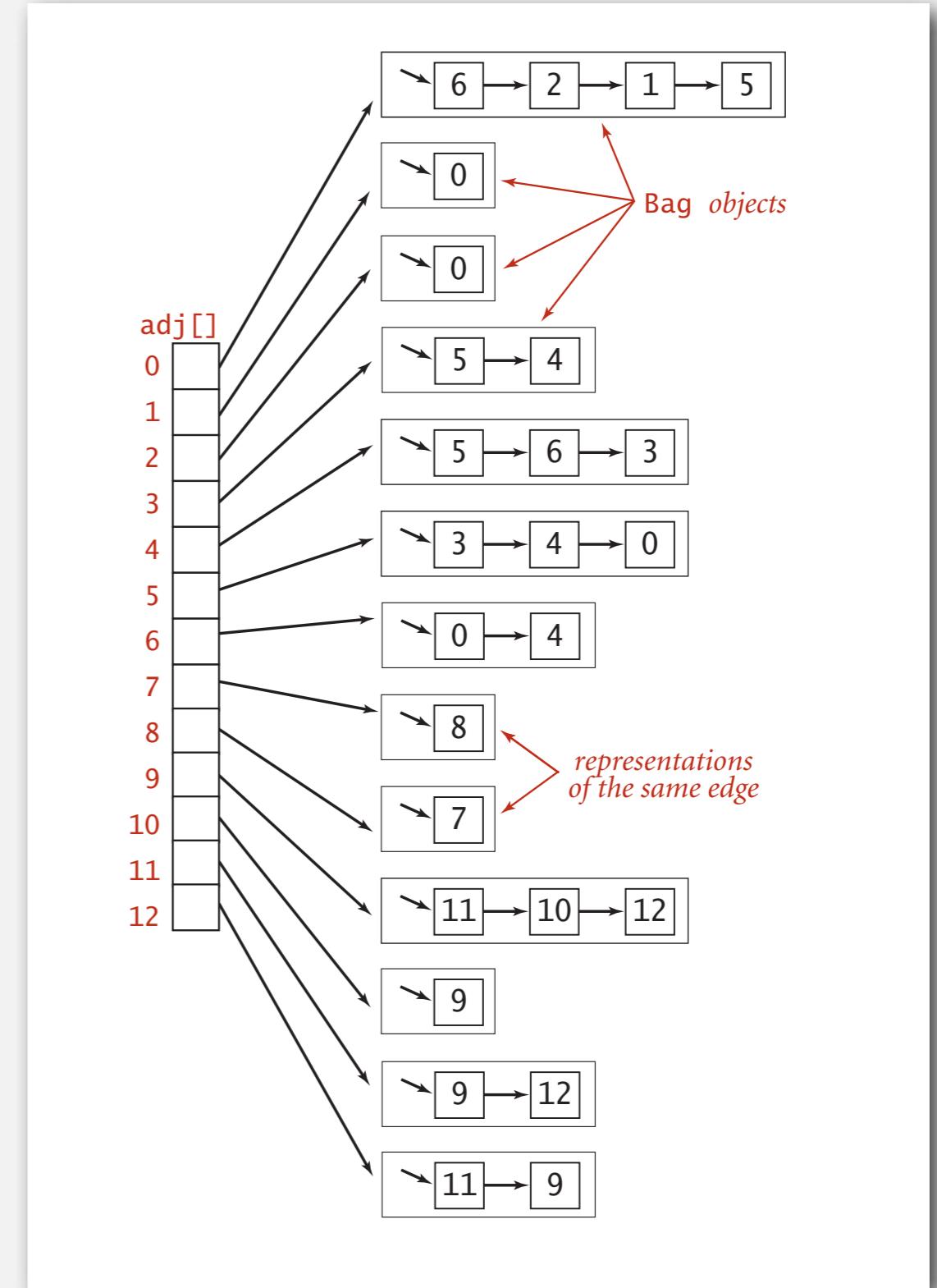
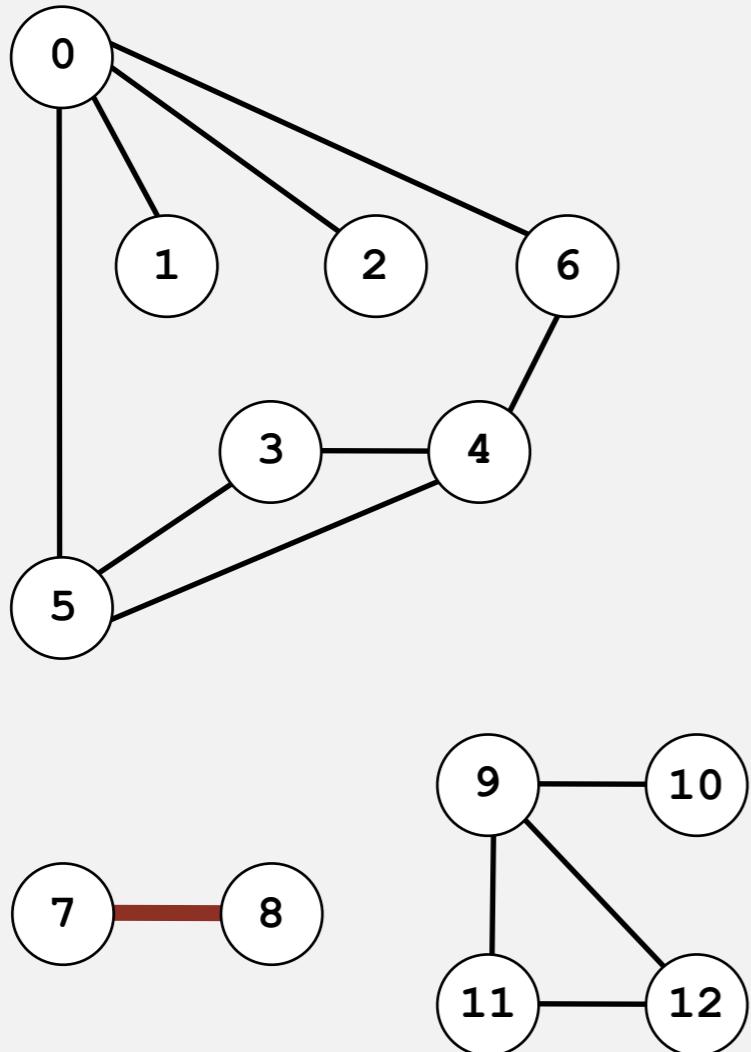


two entries
for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Adjacency-list graph representation

Maintain vertex-indexed array of lists.



Adjacency-list graph representation: Java implementation

```
public class Graph
{
    private final int V;
    private Bag<Integer>[] adj;
```

adjacency lists
(using Bag data type)

```
public Graph(int V)
{
    this.V = V;
    adj = (Bag<Integer>[]) new Bag[V];
    for (int v = 0; v < V; v++)
        adj[v] = new Bag<Integer>();
}
```

create empty graph
with v vertices

```
public void addEdge(int v, int w)
{
    adj[v].add(w);
    adj[w].add(v);
}
```

add edge $v-w$
(parallel edges allowed)

```
public Iterable<Integer> adj(int v)
{   return adj[v]; }
```

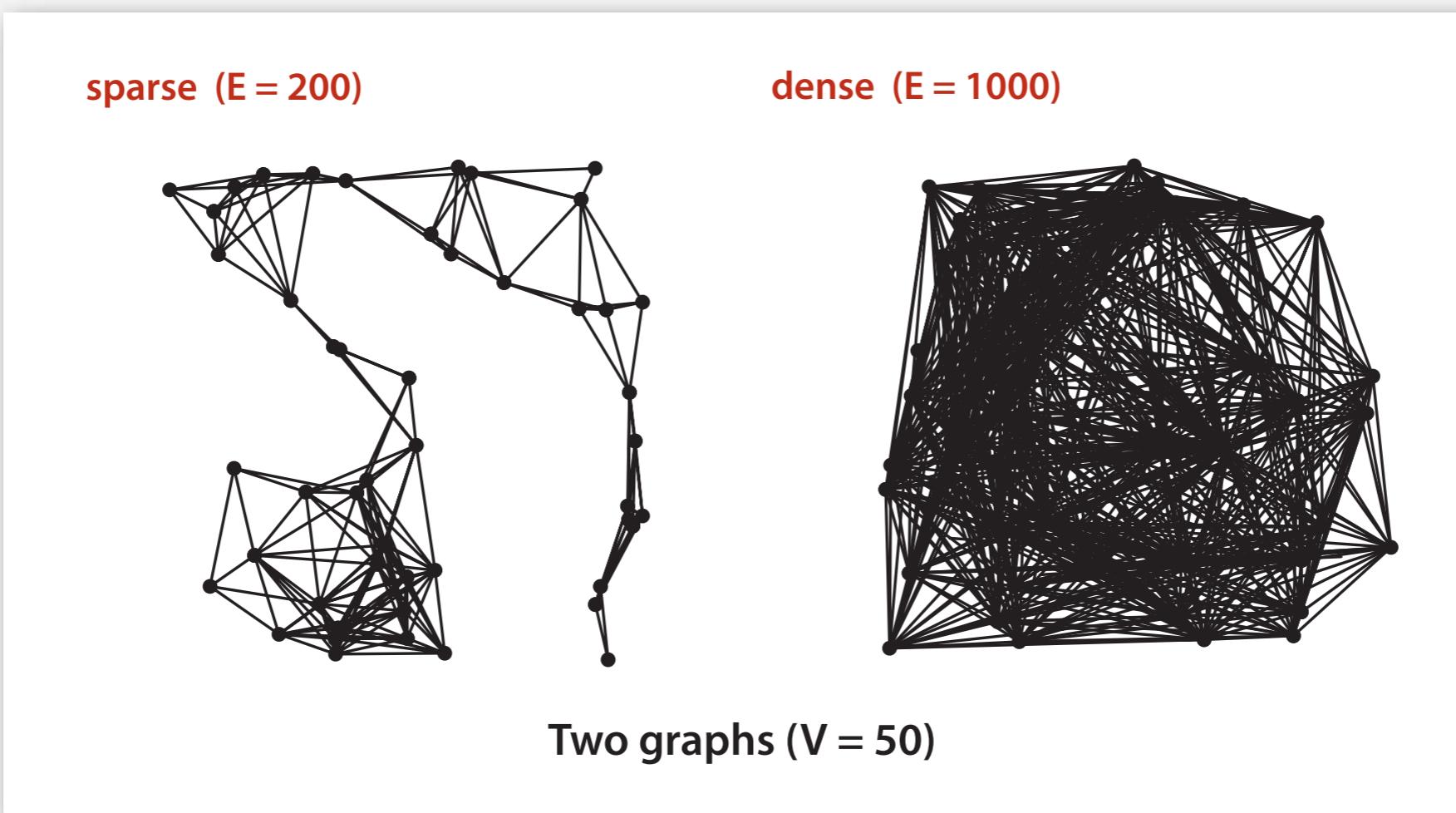
iterator for vertices adjacent to v

Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse**.

huge number of vertices,
small average vertex degree



Graph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices adjacent to v .
- Real-world graphs tend to be **sparse**.

huge number of vertices,
small average vertex degree

representation	space	add edge	edge between v and w ?	iterate over vertices adjacent to v ?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	$E + V$	1	$\text{degree}(v)$	$\text{degree}(v)$

* disallows parallel edges

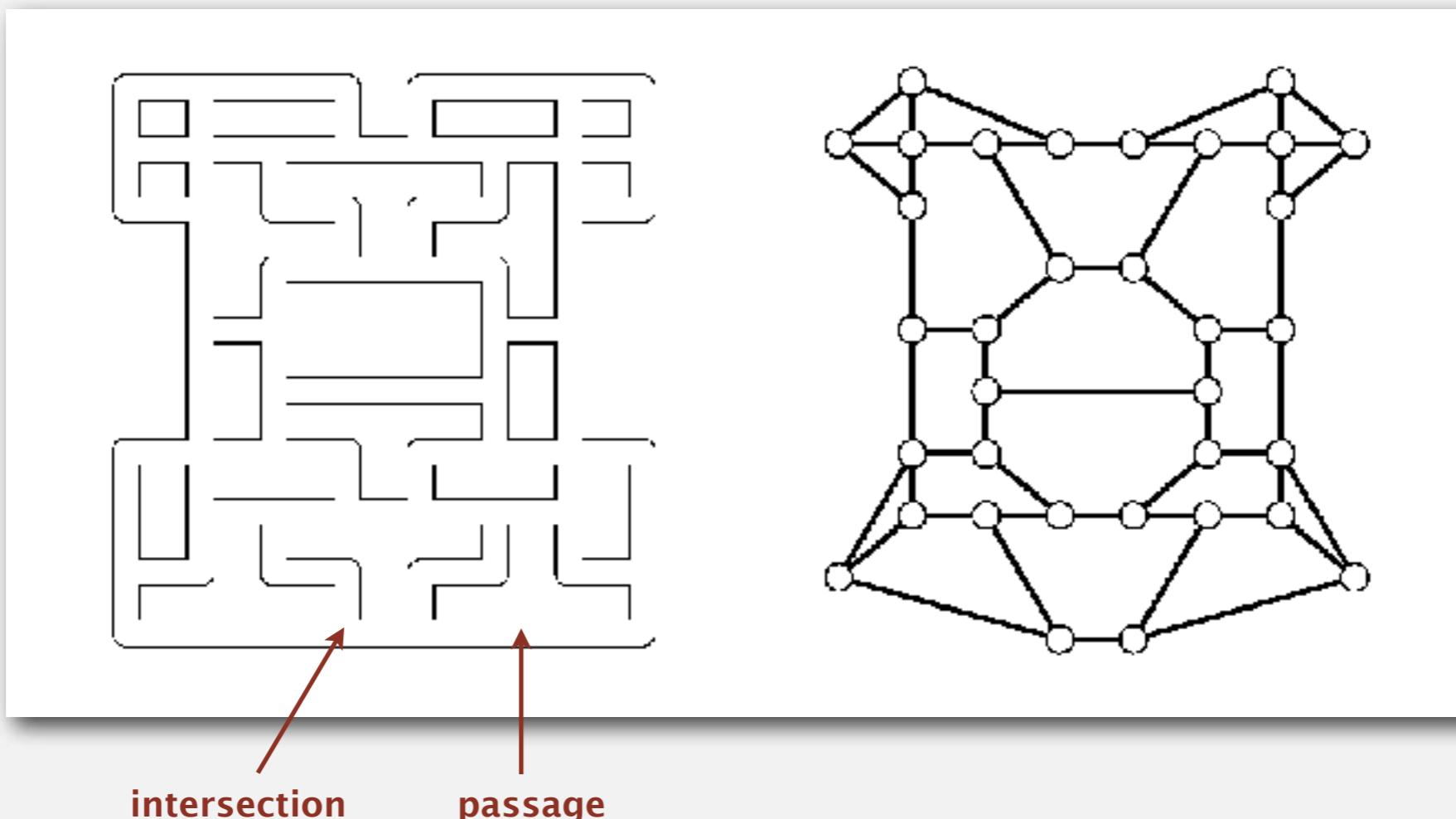
UNDIRECTED GRAPHS

- ▶ Graph API
- ▶ Depth-first search
- ▶ Breadth-first search
- ▶ Connected components
- ▶ Challenges

Maze exploration

Maze graphs.

- Vertex = intersection.
- Edge = passage.

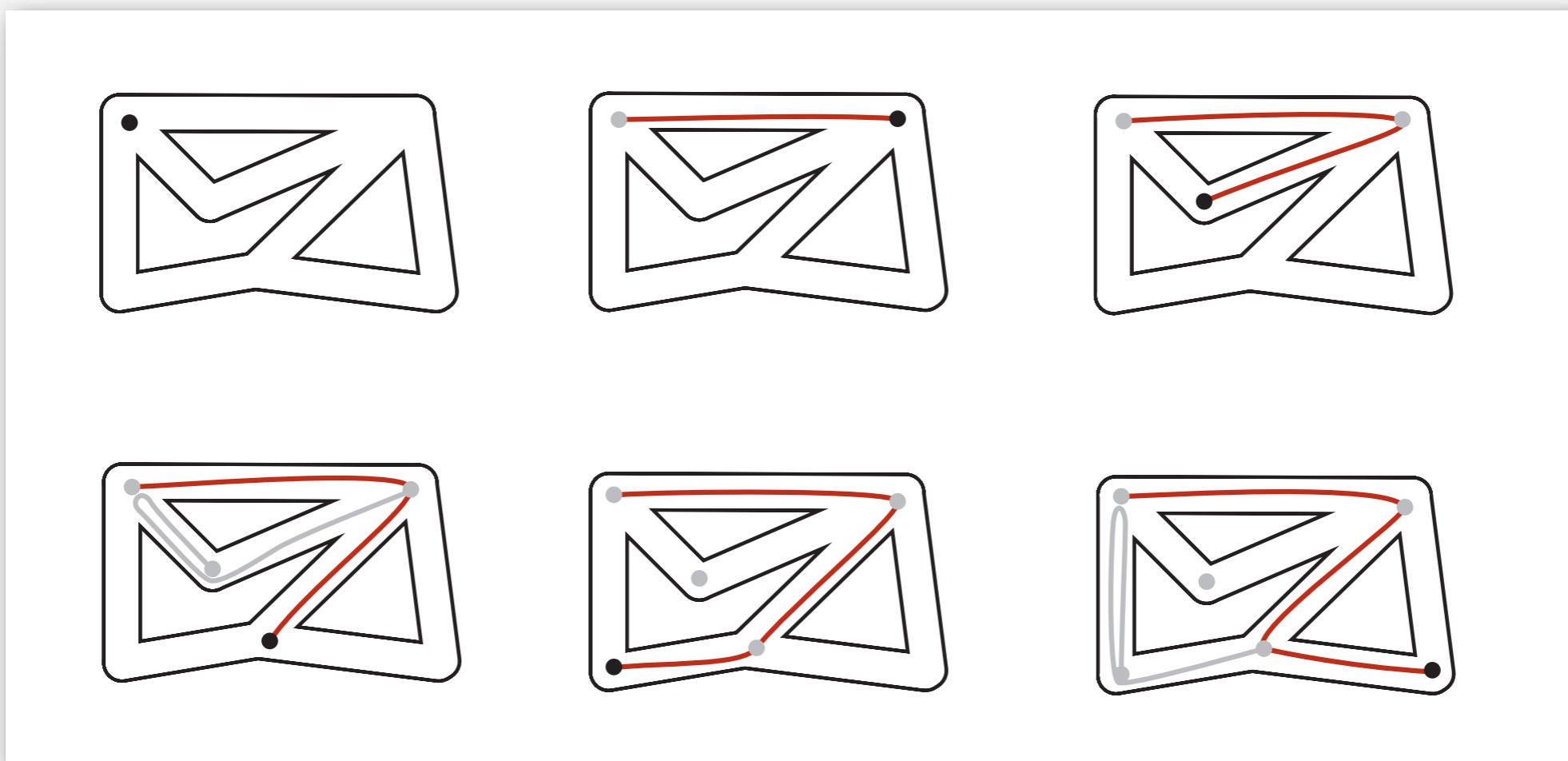


Goal. Explore every intersection in the maze.

Trémaux maze exploration

Algorithm.

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.



Depth-first search

Goal. Systematically search through a graph.

Idea. Mimic maze exploration.

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked
vertices w adjacent to v .

Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

Design challenge. How to implement?

Design pattern for graph processing

Design pattern. Decouple graph data type from graph processing.

- Create a `Graph` object.
- Pass the `Graph` to a graph-processing routine, e.g., `Paths`.
- Query the graph-processing routine for information.

```
public class Paths
```

```
    Paths(Graph G, int s)
```

find paths in G from source s

```
    boolean hasPathTo(int v)
```

is there a path from s to v?

```
    Iterable<Integer> pathTo(int v)
```

path from s to v; null if no such path

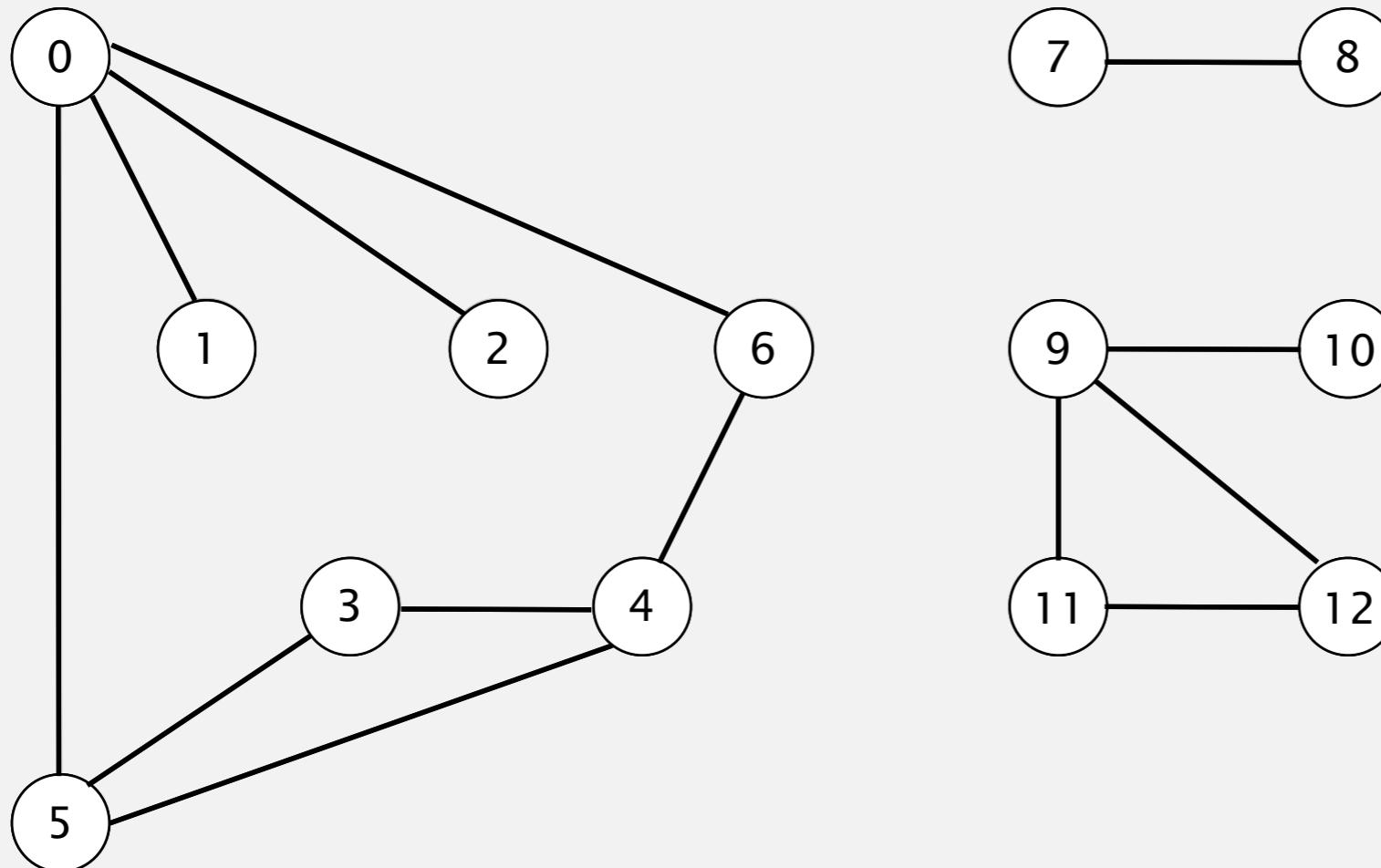
```
Paths paths = new Paths(G, s);
for (int v = 0; v < G.V(); v++)
    if (paths.hasPathTo(v))
        StdOut.println(v);
```

print all vertices
connected to s

Depth-first search

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



tinyG.txt

$V \rightarrow$ 13
 $E \rightarrow$ 13

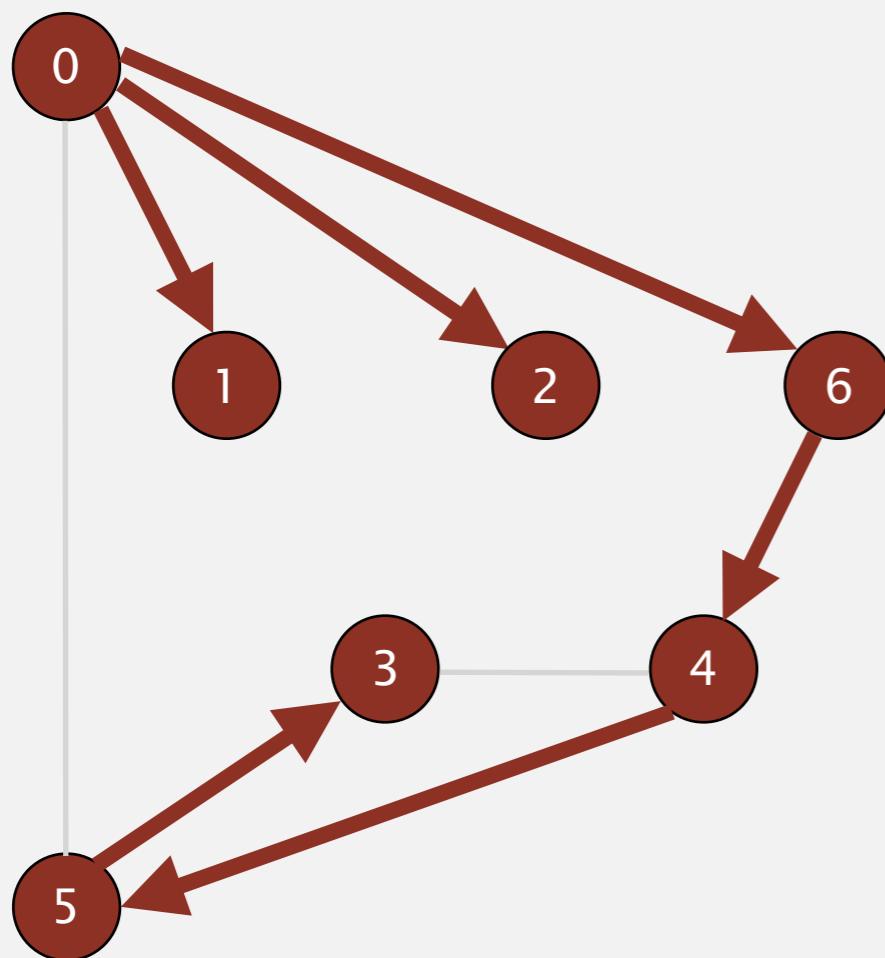
13	0 5
13	4 3
13	0 1
13	9 12
13	6 4
13	5 4
13	0 2
13	11 12
13	9 10
13	0 6
13	7 8
13	9 11
13	5 3

graph G

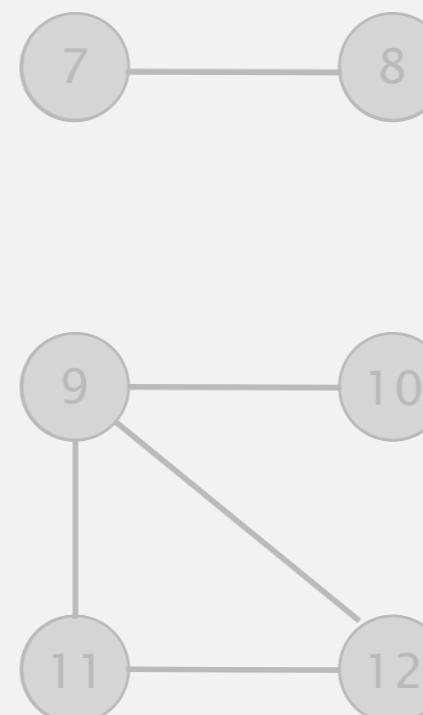
Depth-first search

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



vertices reachable from 0



v	marked[]	edgeTo[v]
0	T	-
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Depth-first search

Goal. Find all vertices connected to s (and a path).

Idea. Mimic maze exploration.

Algorithm.

- Use recursion (ball of string).
- Mark each visited vertex (and keep track of edge taken to visit it).
- Return (retrace steps) when no unvisited options.

Data structures.

- `boolean[] marked` to mark visited vertices.
- `int[] edgeTo` to keep tree of paths.
 $(\text{edgeTo}[w] == v)$ means that edge $v-w$ taken to visit w for first time

Depth-first search

```
public class DepthFirstPaths
{
    private boolean[] marked;
    private int[] edgeTo;
    private int s;
```

marked[v] = true
if v connected to s
edgeTo[v] = previous
vertex on path from s to v

```
public DepthFirstSearch(Graph G, int s)
{
    ...
    dfs(G, s);
}
```

initialize data structures
find vertices connected to s

```
private void dfs(Graph G, int v)
{
    marked[v] = true;
    for (int w : G.adj(v))
        if (!marked[w])
        {
            dfs(G, w);
            edgeTo[w] = v;
        }
}
```

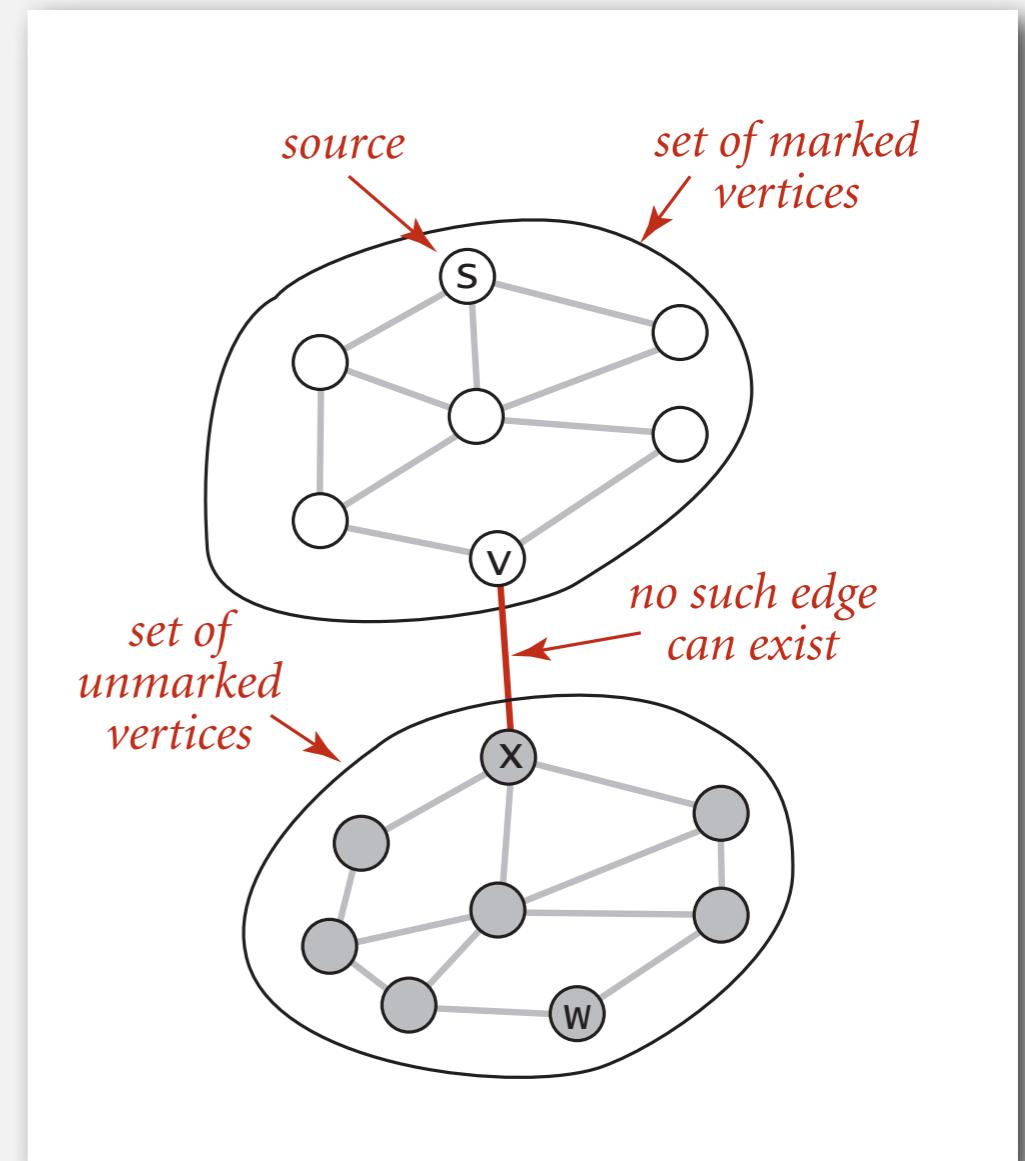
recursive DFS does the
work

Depth-first search properties

Proposition. DFS marks all vertices connected to s in time proportional to the sum of their degrees.

Pf.

- Correctness:
 - if w marked, then w connected to s (why?)
 - if w connected to s , then w marked
(if w unmarked, then consider last edge on a path from s to w that goes from a marked vertex to an unmarked one)
- Running time:
Each vertex connected to s is visited once.



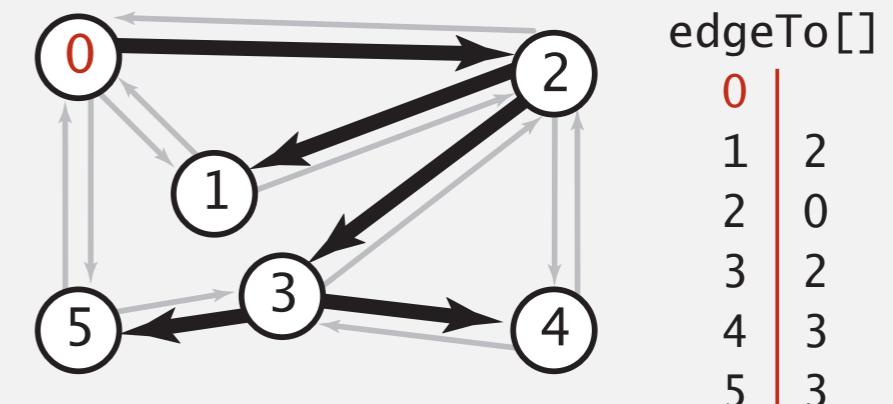
Depth-first search properties

Proposition. After DFS, can find vertices connected to s in constant time and can find a path to s (if one exists) in time proportional to its length.

Pf. `edgeTo[]` is a parent-link representation of a tree rooted at s .

```
public boolean hasPathTo(int v)
{   return marked[v];   }

public Iterable<Integer> pathTo(int v)
{
    if (!hasPathTo(v)) return null;
    Stack<Integer> path = new Stack<Integer>();
    for (int x = v; x != s; x = edgeTo[x])
        path.push(x);
    path.push(s);
    return path;
}
```



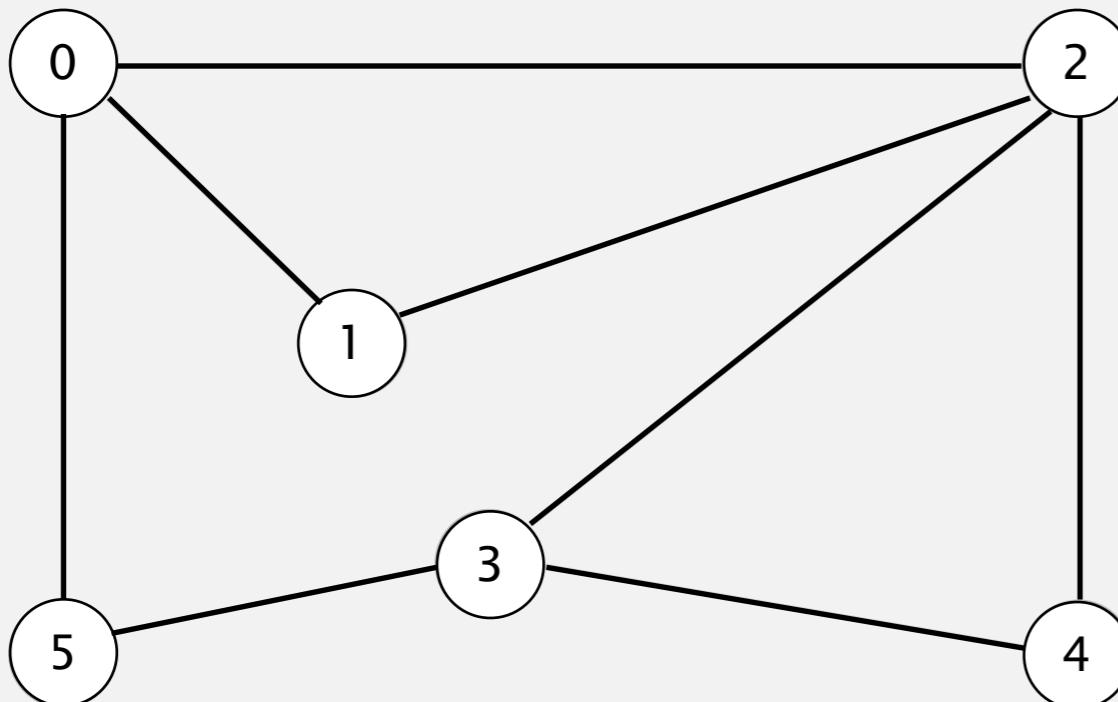
UNDIRECTED GRAPHS

- ▶ Graph API
- ▶ Depth-first search
- ▶ Breadth-first search
- ▶ Connected components
- ▶ Challenges

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



graph G

tinyCG.txt

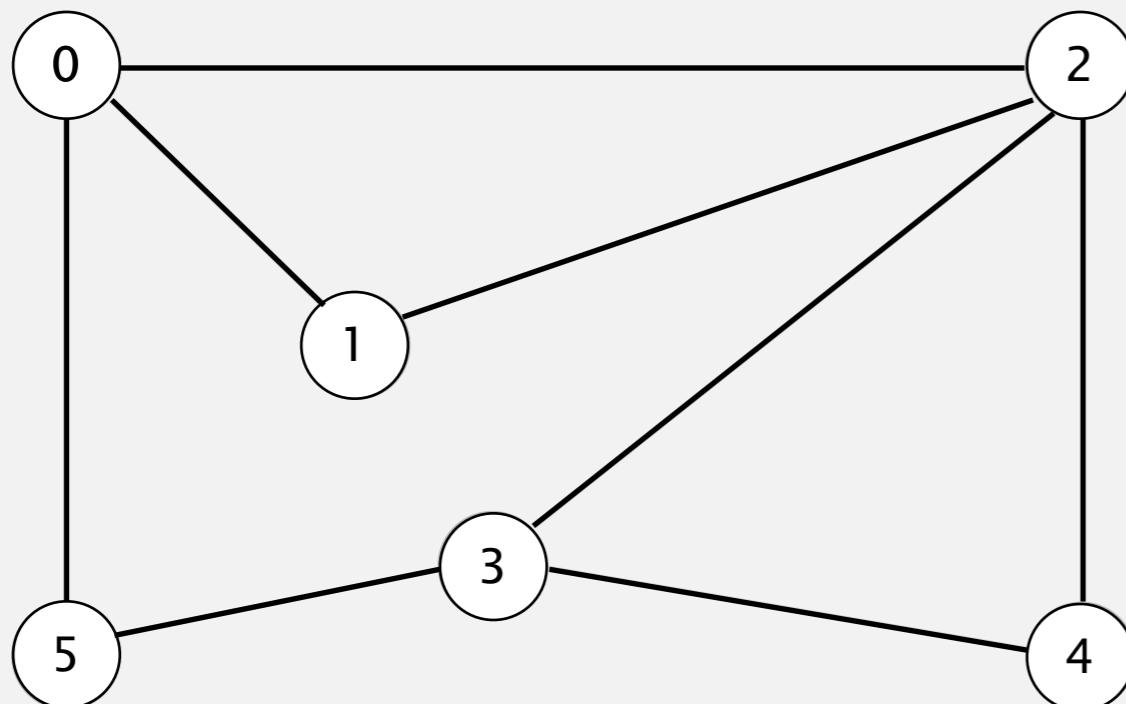
$V \rightarrow$ 6
8
0 5
2 4
2 3
1 2
0 1
3 4
3 5
0 2

$E \rightarrow$

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue



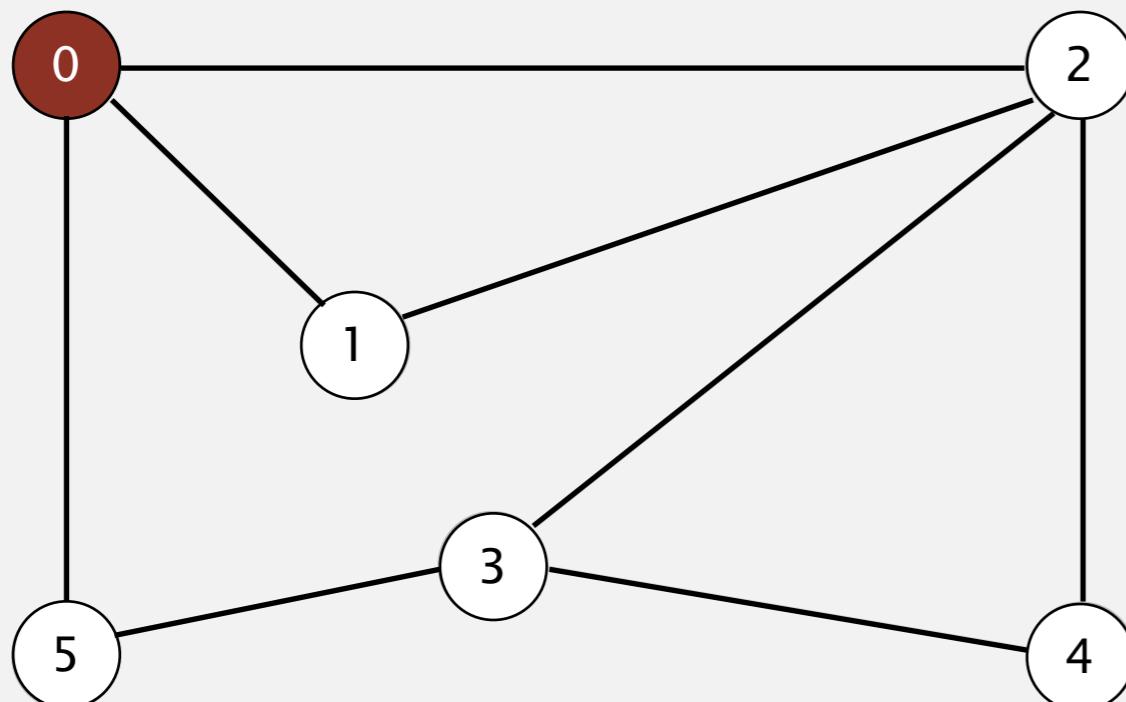
v	edgeTo[v]
0	-
1	-
2	-
3	-
4	-
5	-

add 0 to queue

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
0
1
2
3
4
5

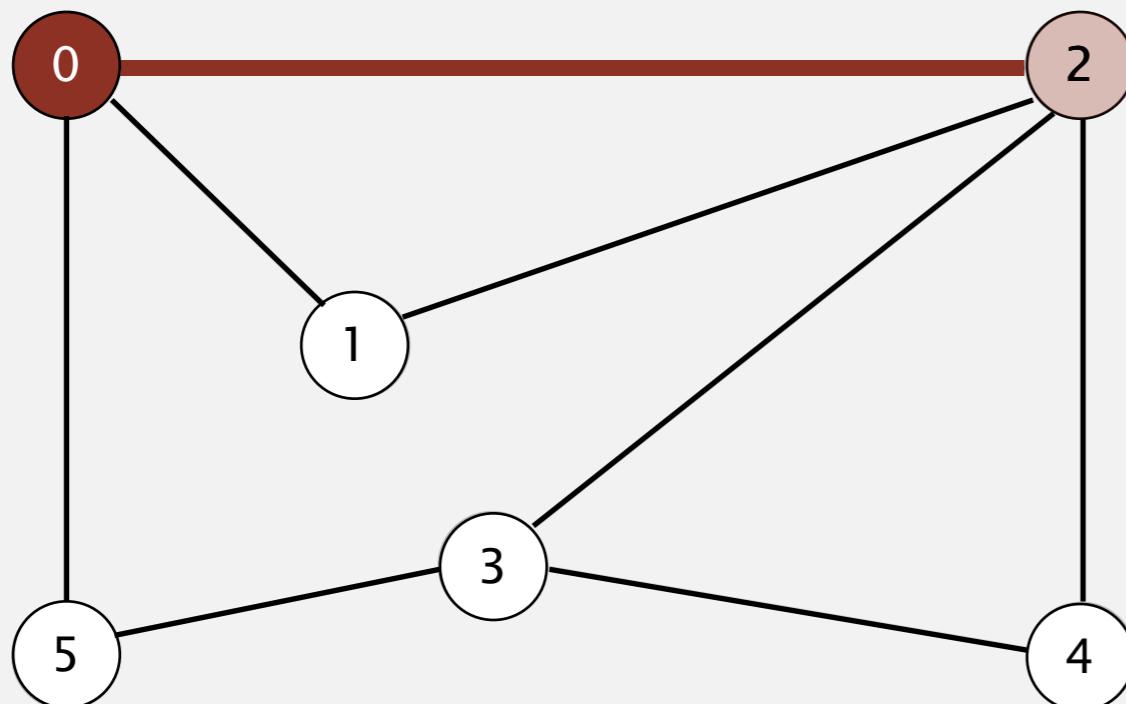
v	edgeTo[v]
0	-
1	-
2	-
3	-
4	-
5	-

dequeue 0

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue



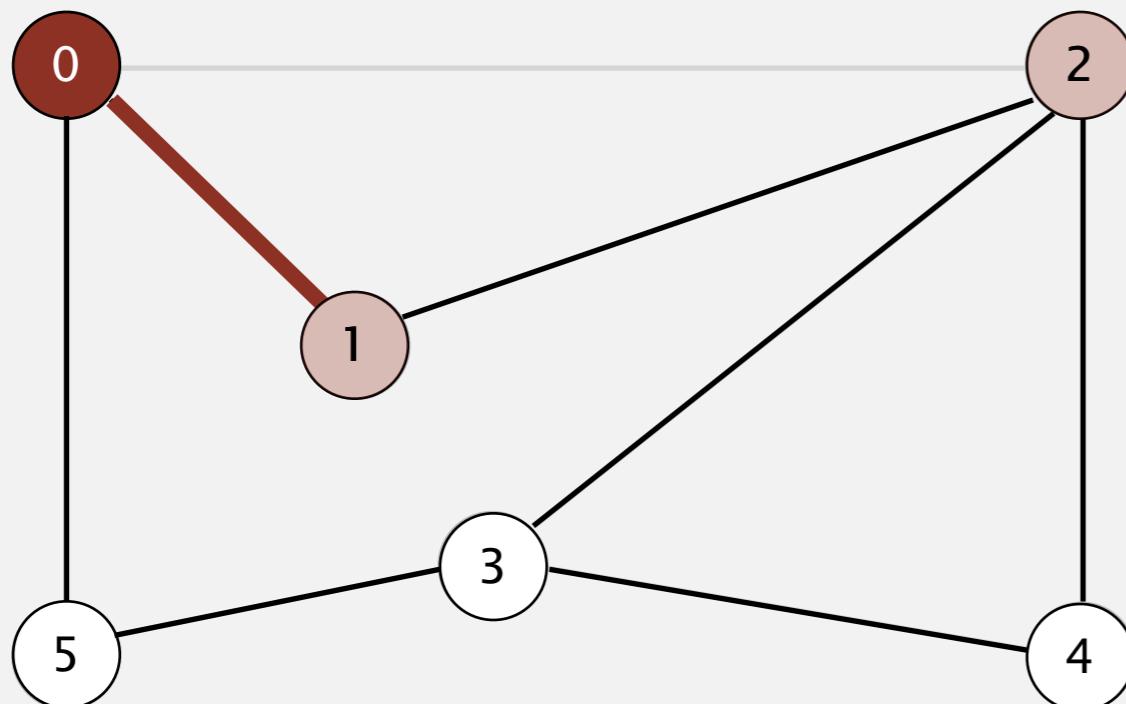
v	edgeTo[v]
0	-
1	-
2	0
3	-
4	-
5	-

dequeue 0

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
0
1
2
3
4
5

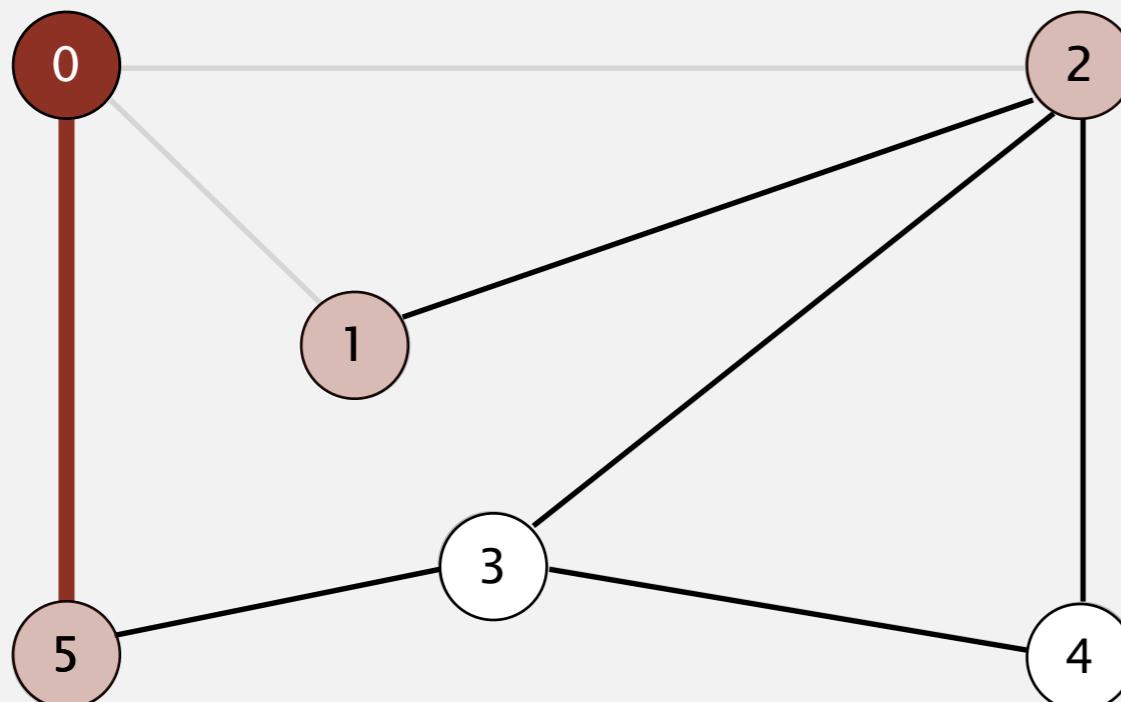
v	edgeTo[v]
0	-
1	0
2	0
3	-
4	-
5	-

dequeue 0

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
1
2

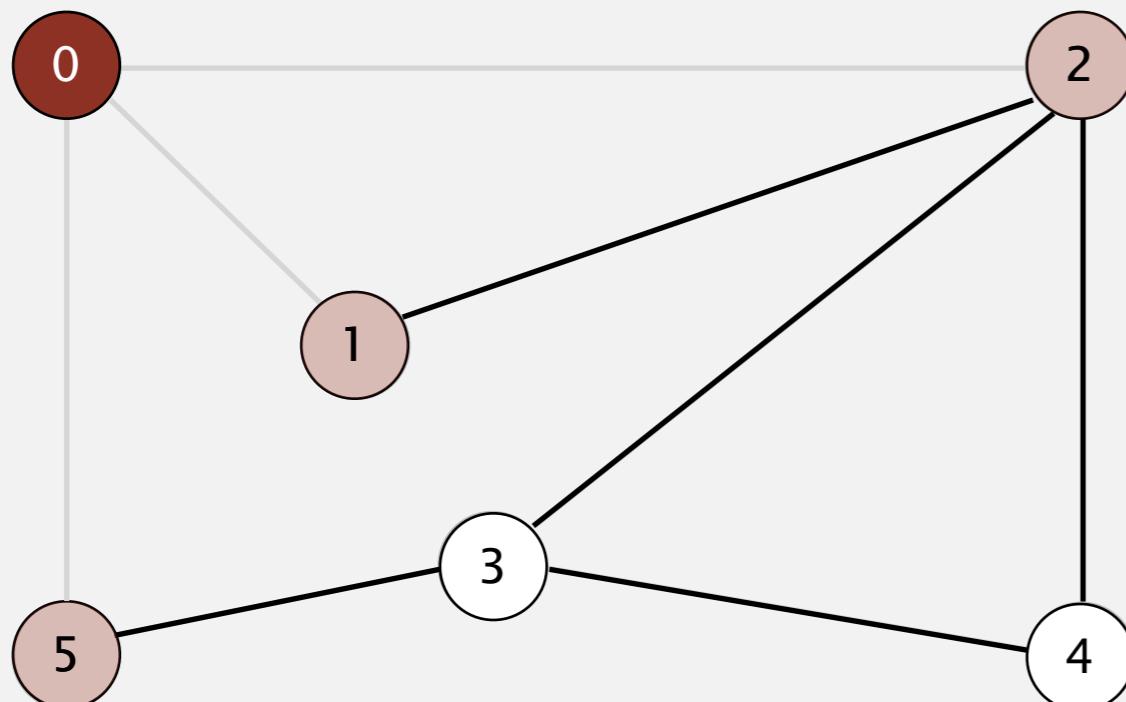
v	edgeTo[v]
0	-
1	0
2	0
3	-
4	-
5	0

dequeue 0

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
5
1
2
5
2

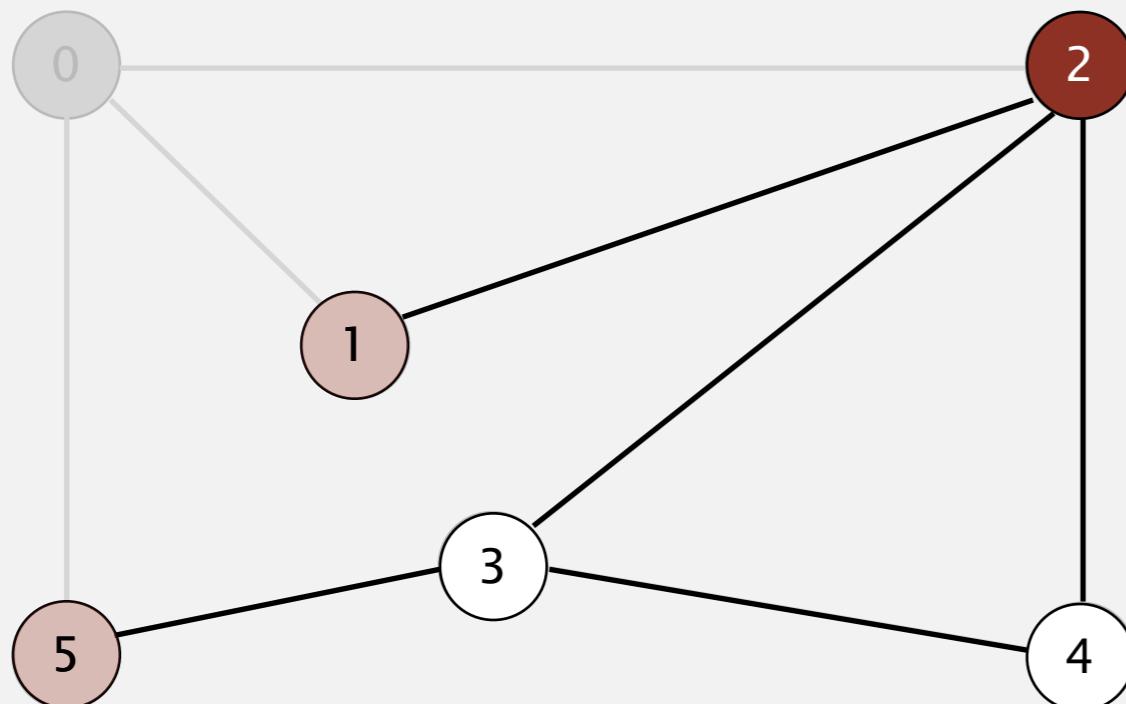
v	edgeTo[v]
0	-
1	0
2	0
3	-
4	-
5	0

0 done

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
5
1
2
5

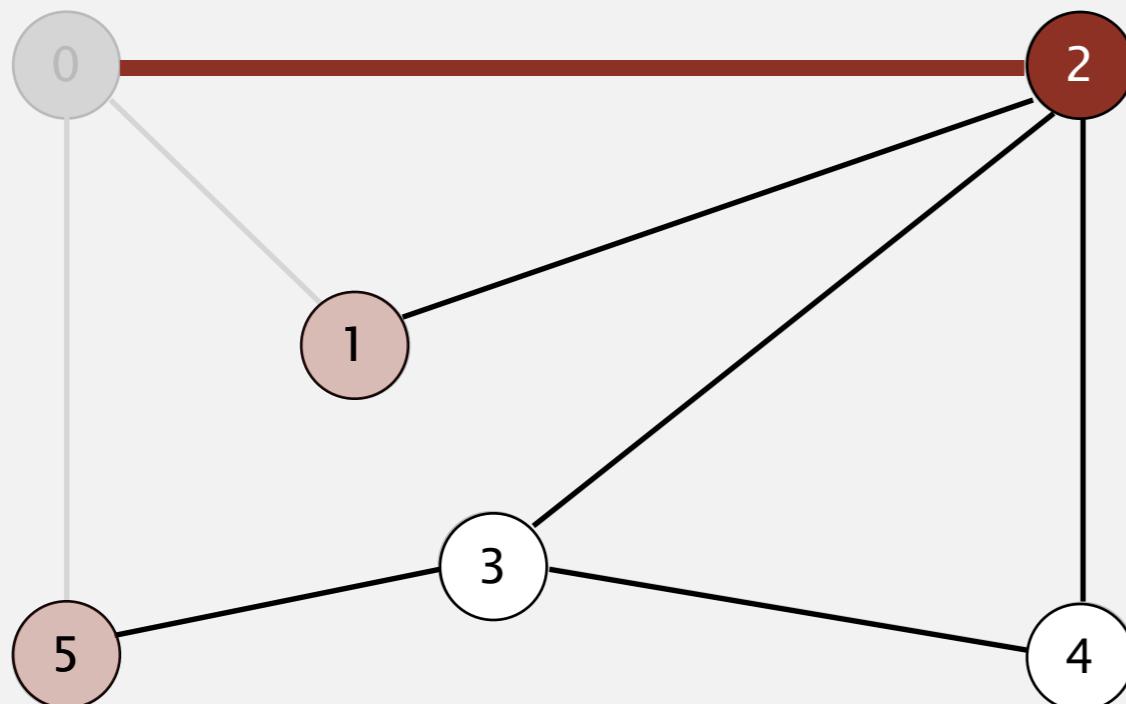
v	edgeTo[v]
0	-
1	0
2	0
3	-
4	-
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
5
1

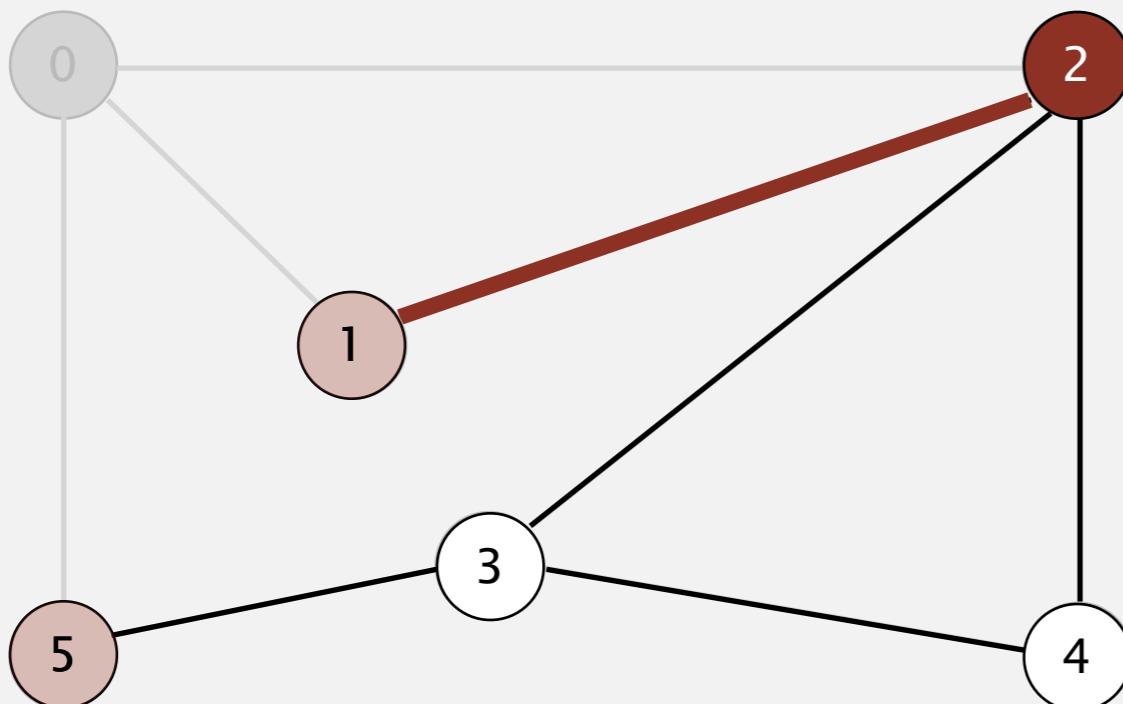
v	edgeTo[v]
0	-
1	0
2	0
3	-
4	-
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
5
1

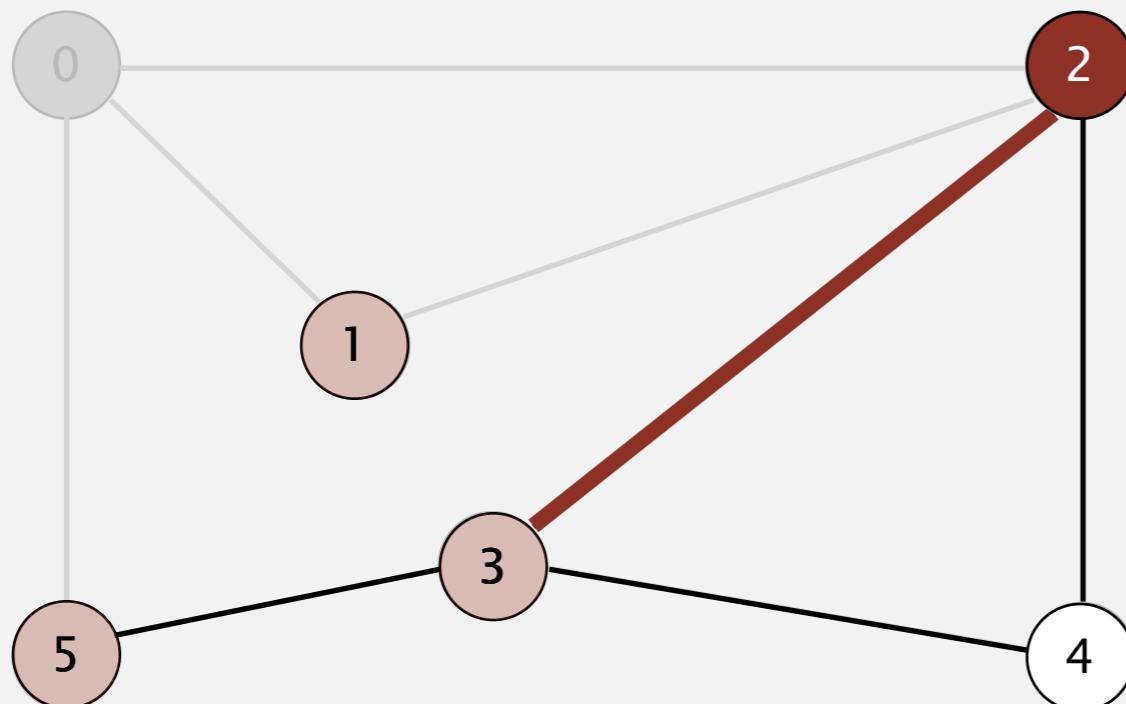
v	edgeTo[v]
0	-
1	0
2	0
3	-
4	-
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
5
1

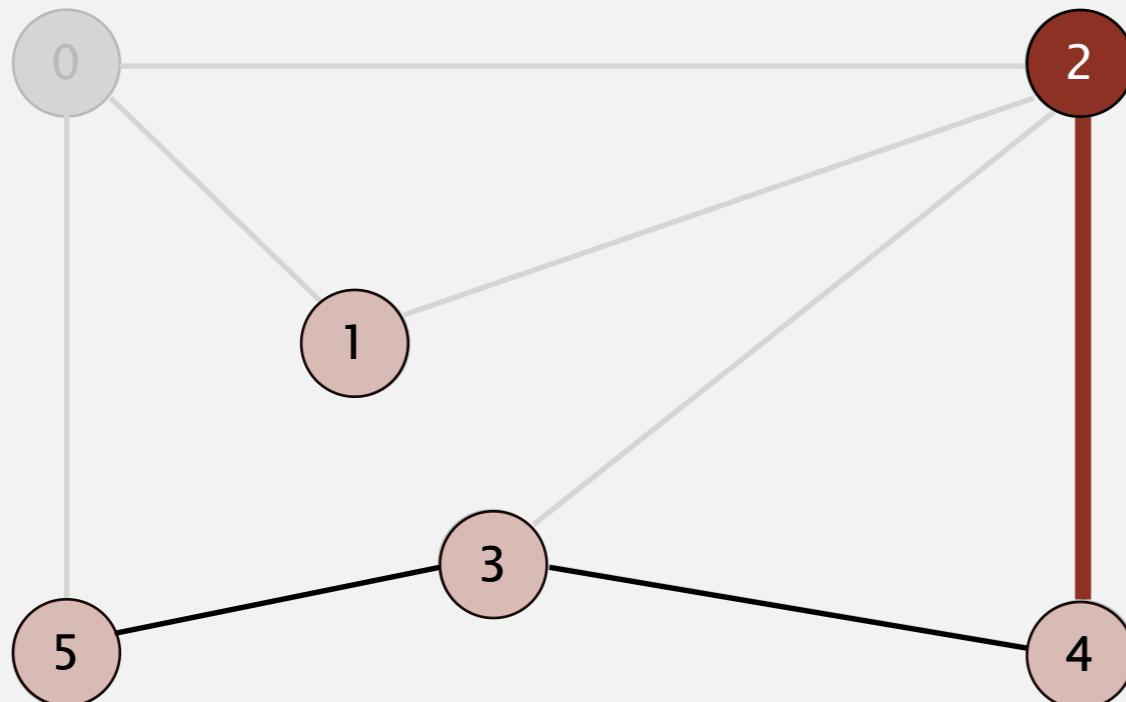
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	-
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
0
1
2
3
4
5
1

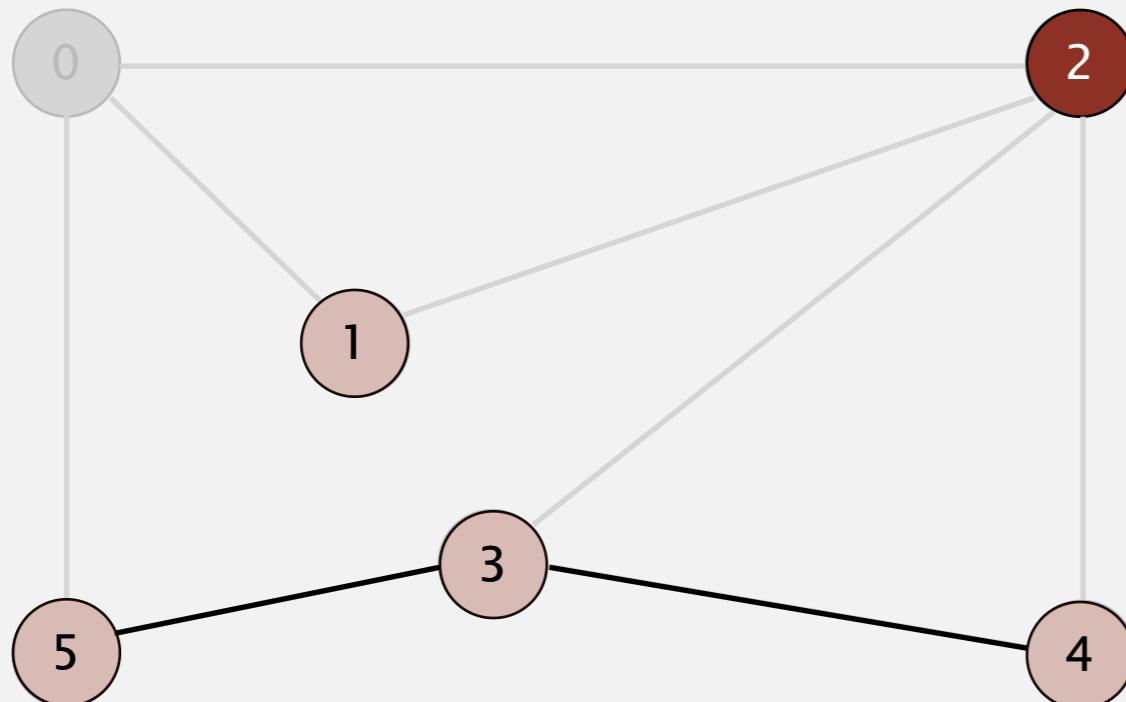
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 2

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
3
5
1

v edgeTo[v]

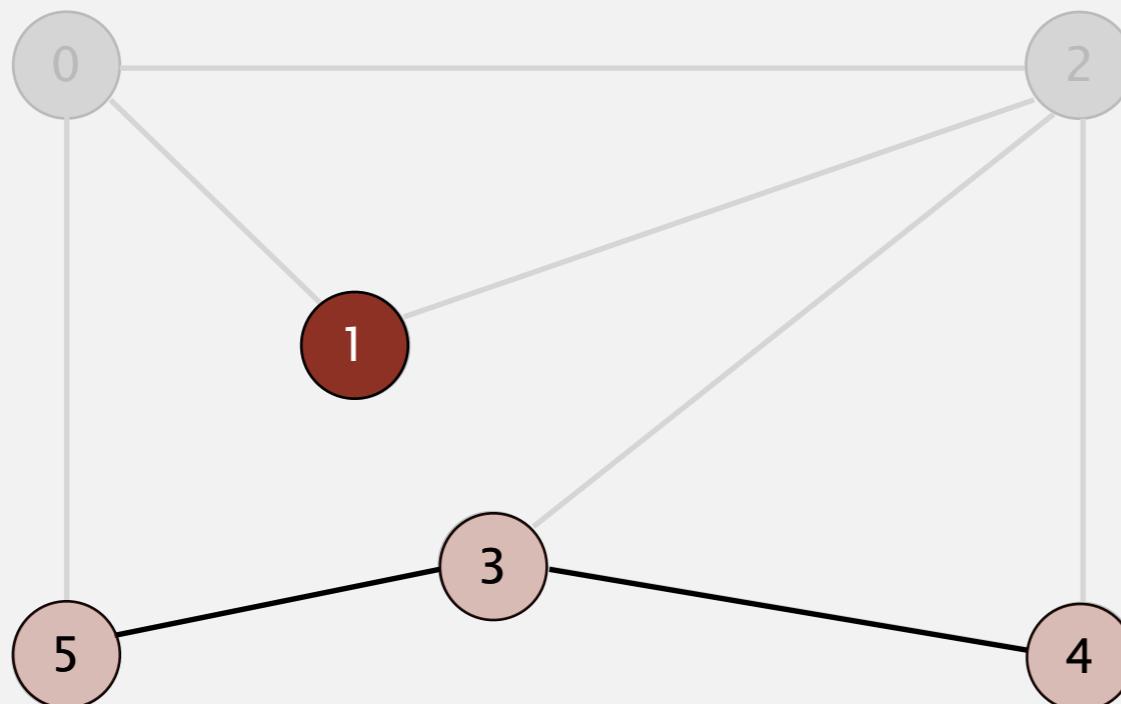
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

2 done

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
3
5
1

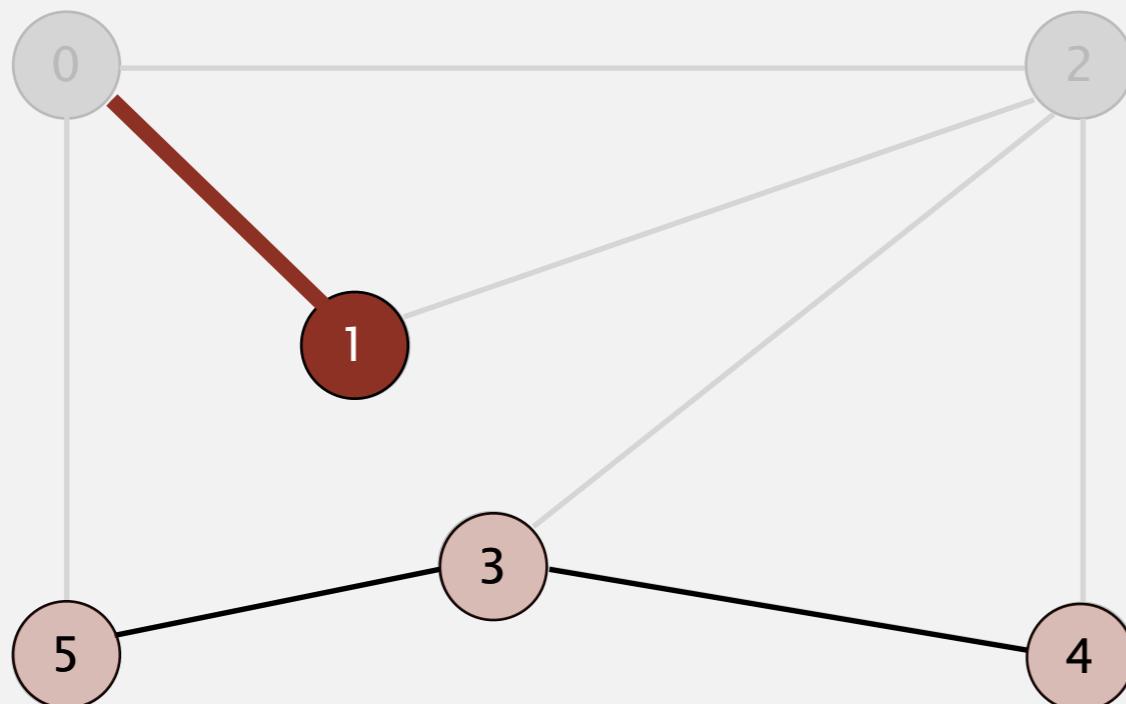
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 1

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
3
4
5
5

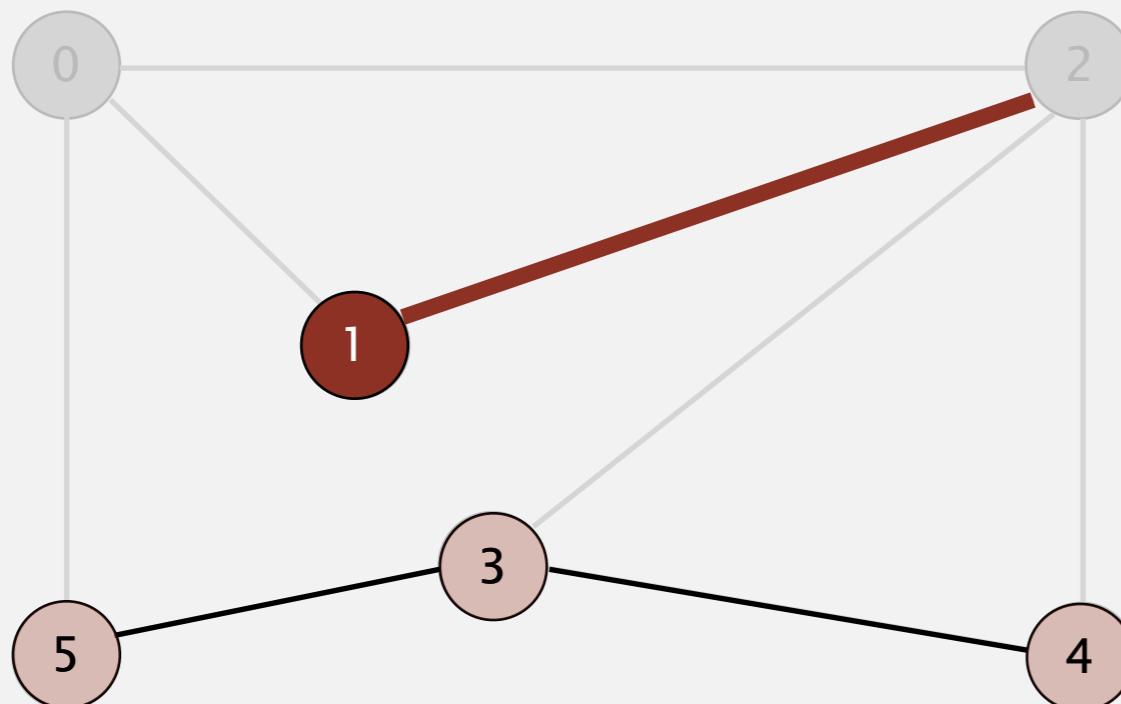
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 1

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
3
4
5
5

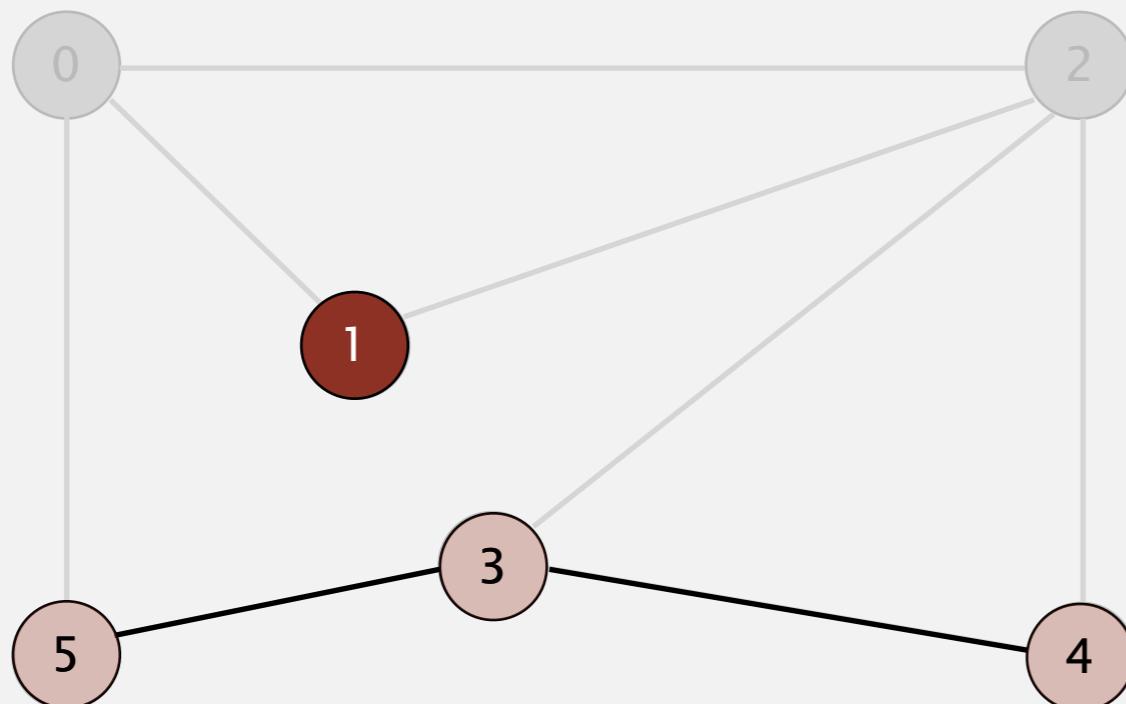
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 1

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
0
1
2
4
3
5
5

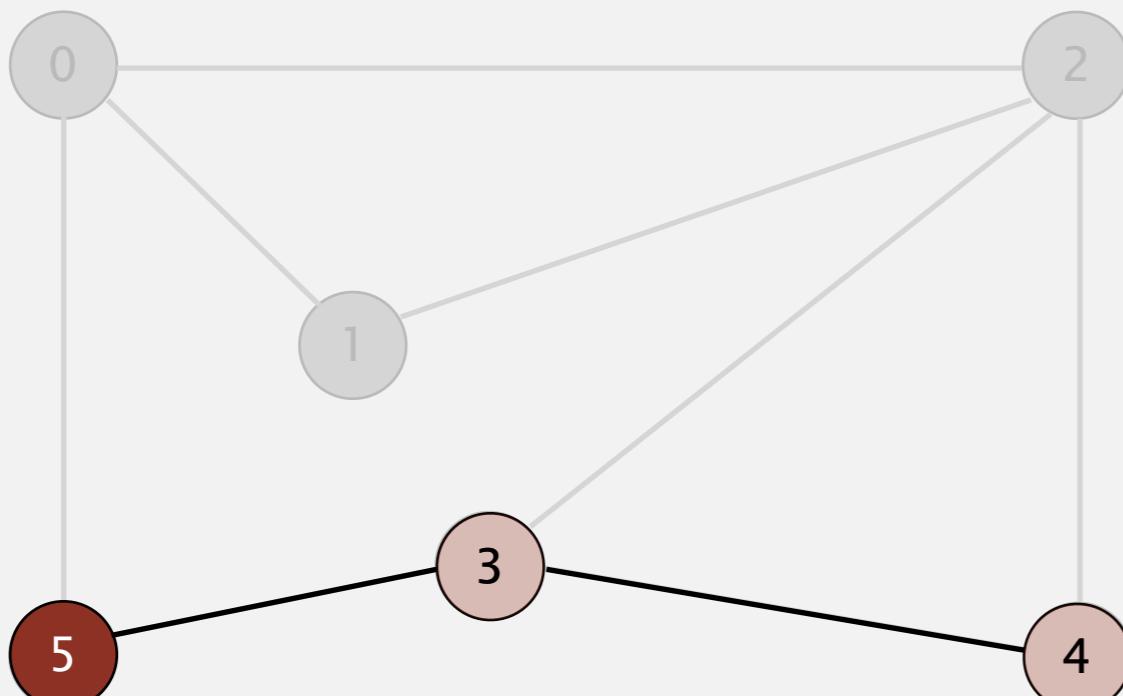
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

1 done

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
3
4
5

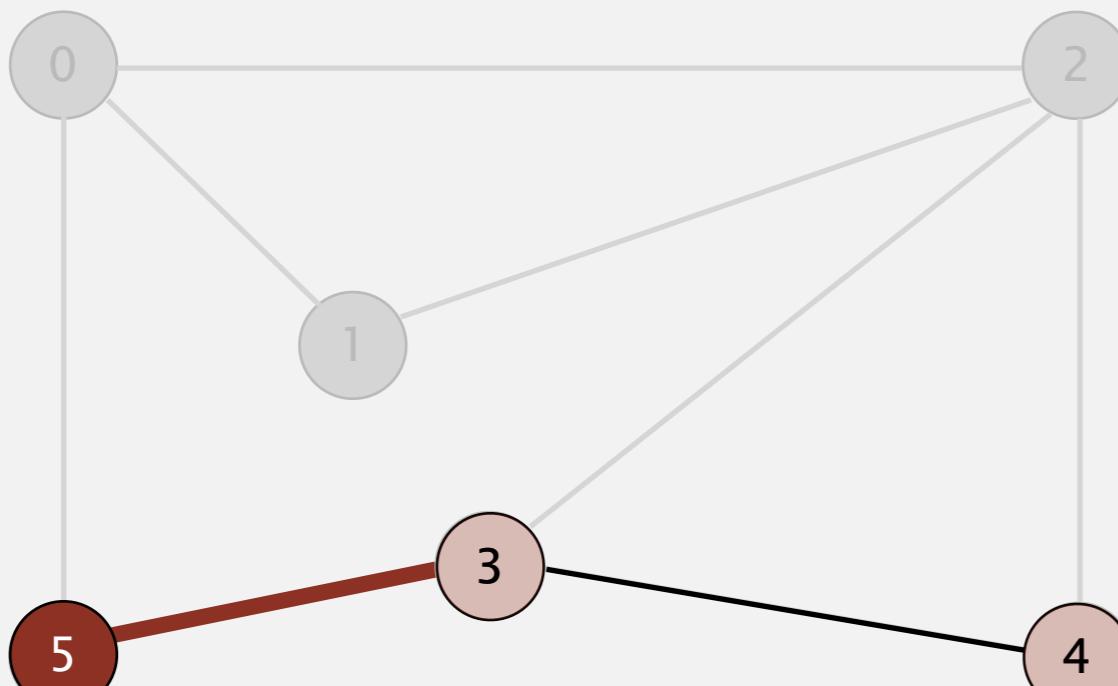
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 5

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
5
3

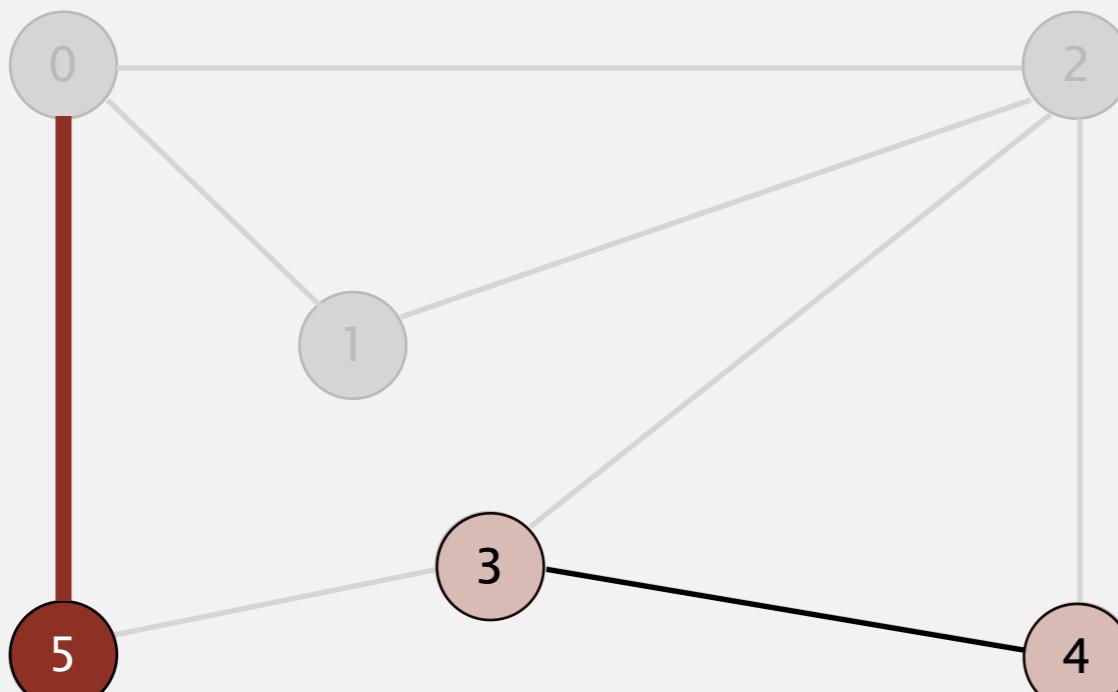
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 5

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
5
3

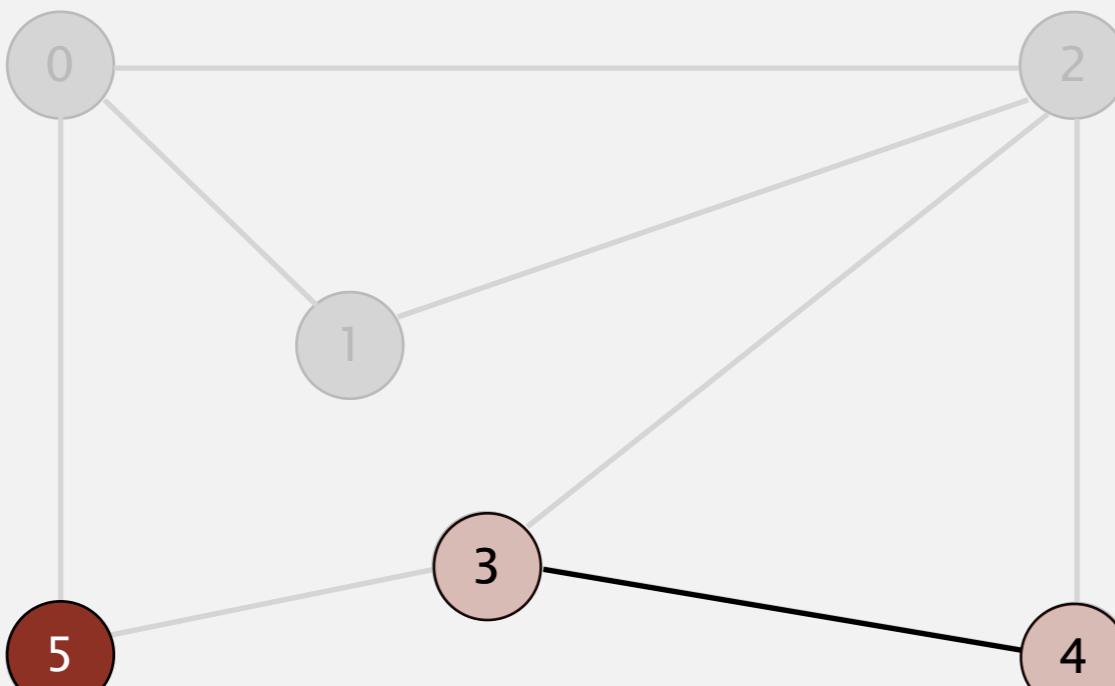
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 5

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
5
3

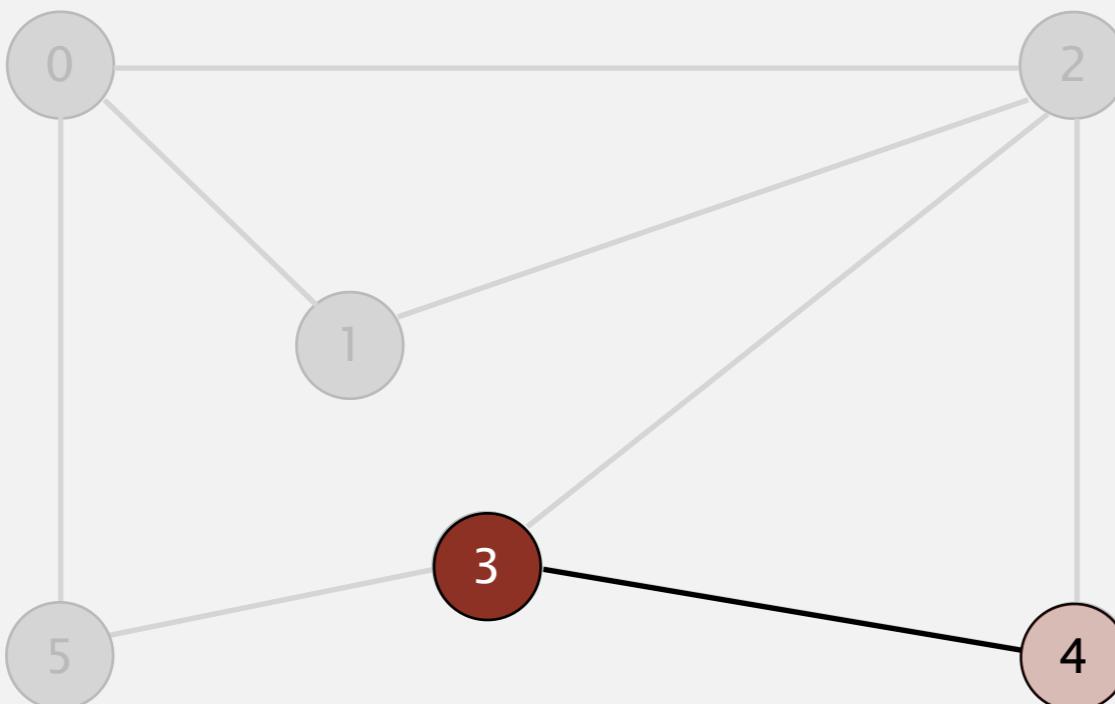
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

5 done

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

v
4
3

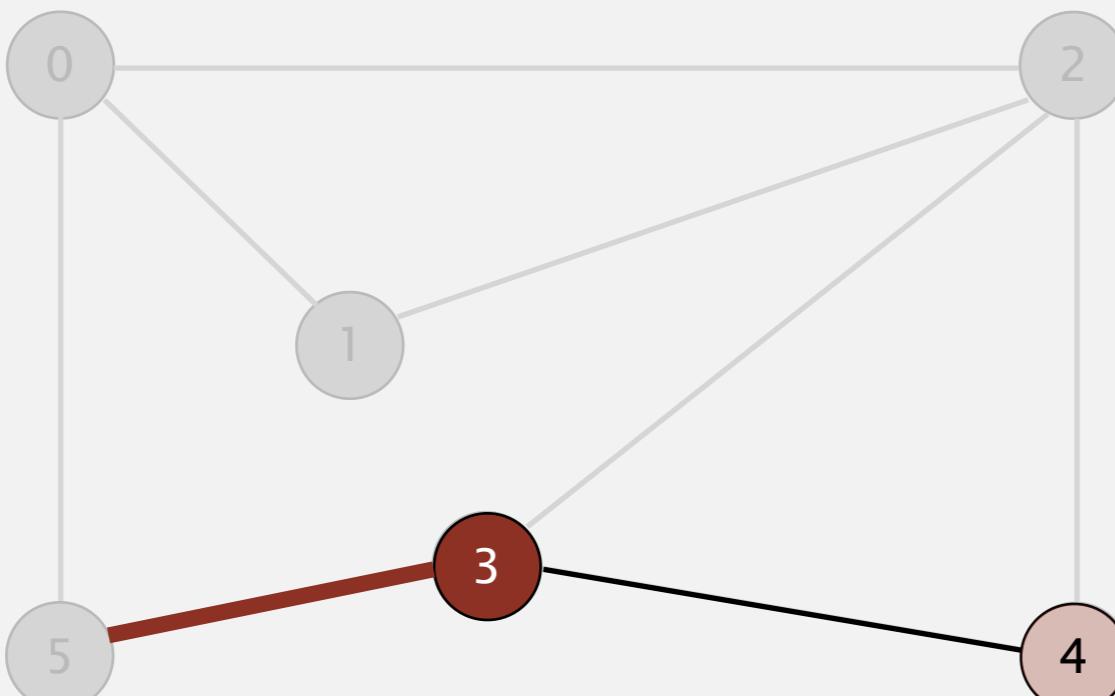
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

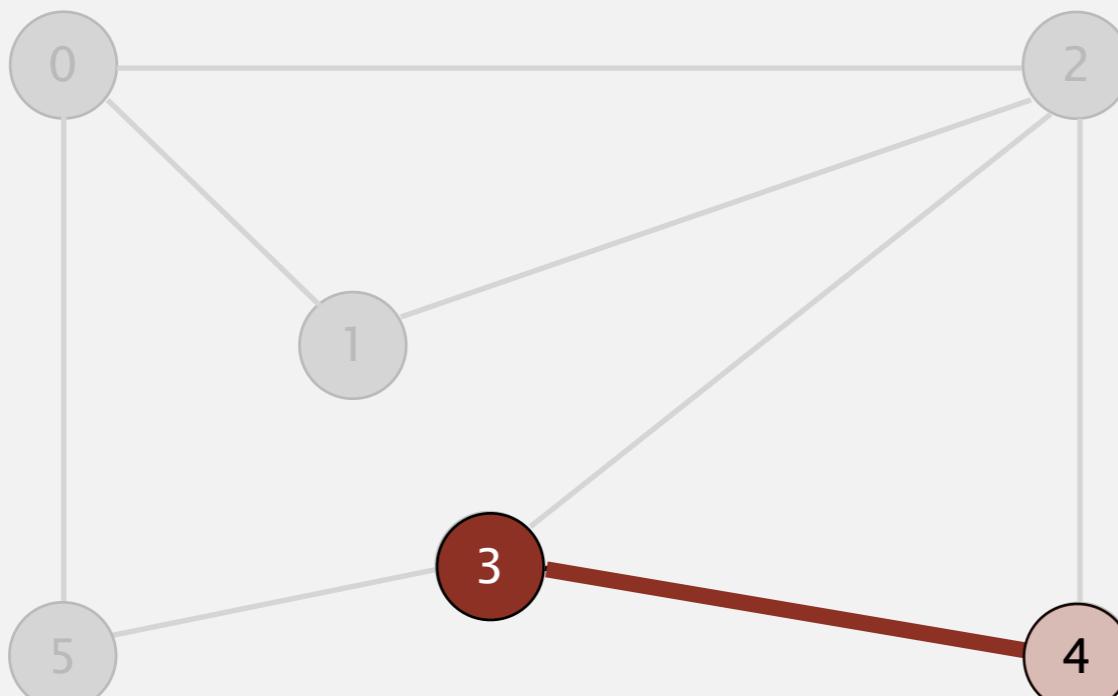
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



dequeue 3

queue

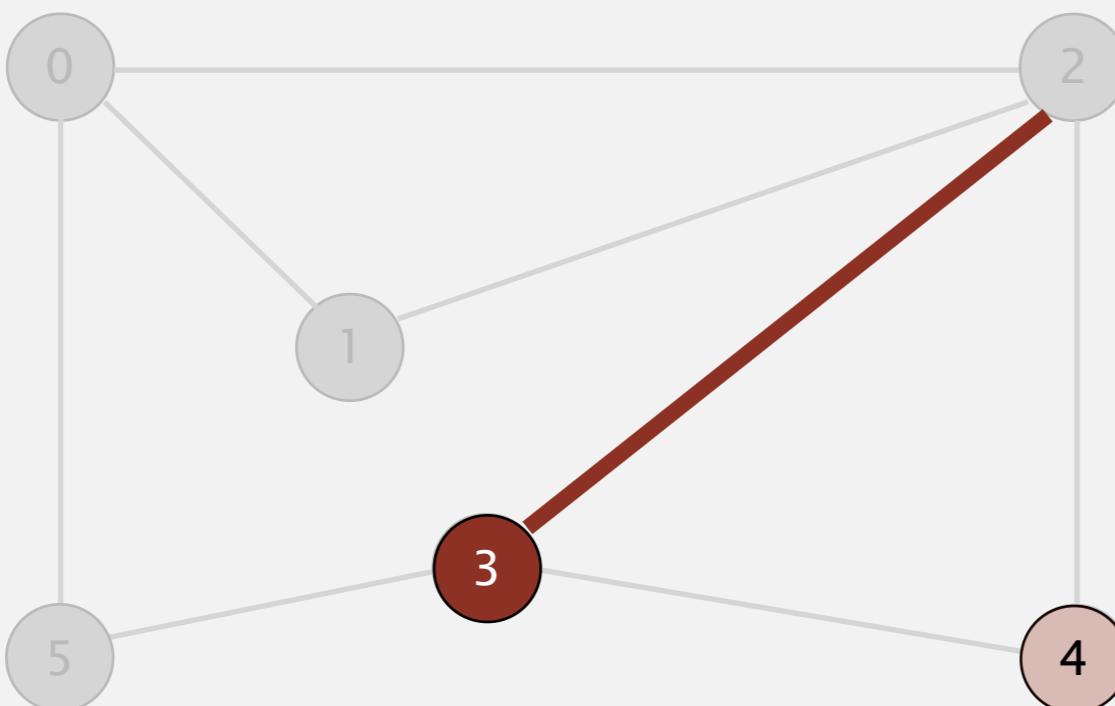
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

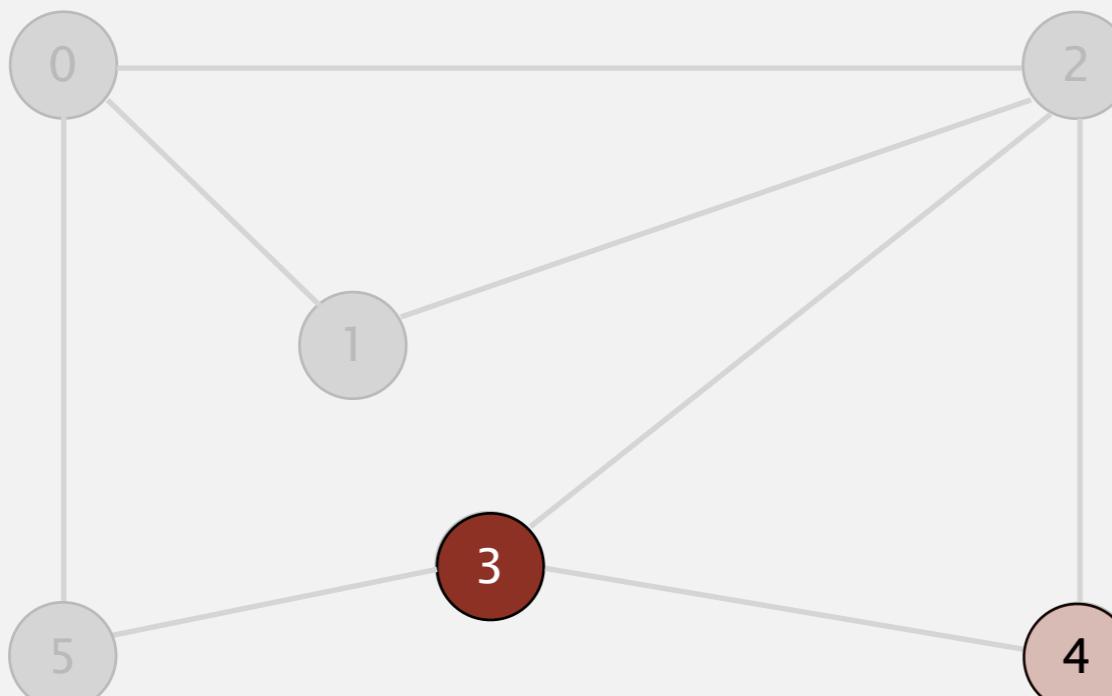
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 3

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

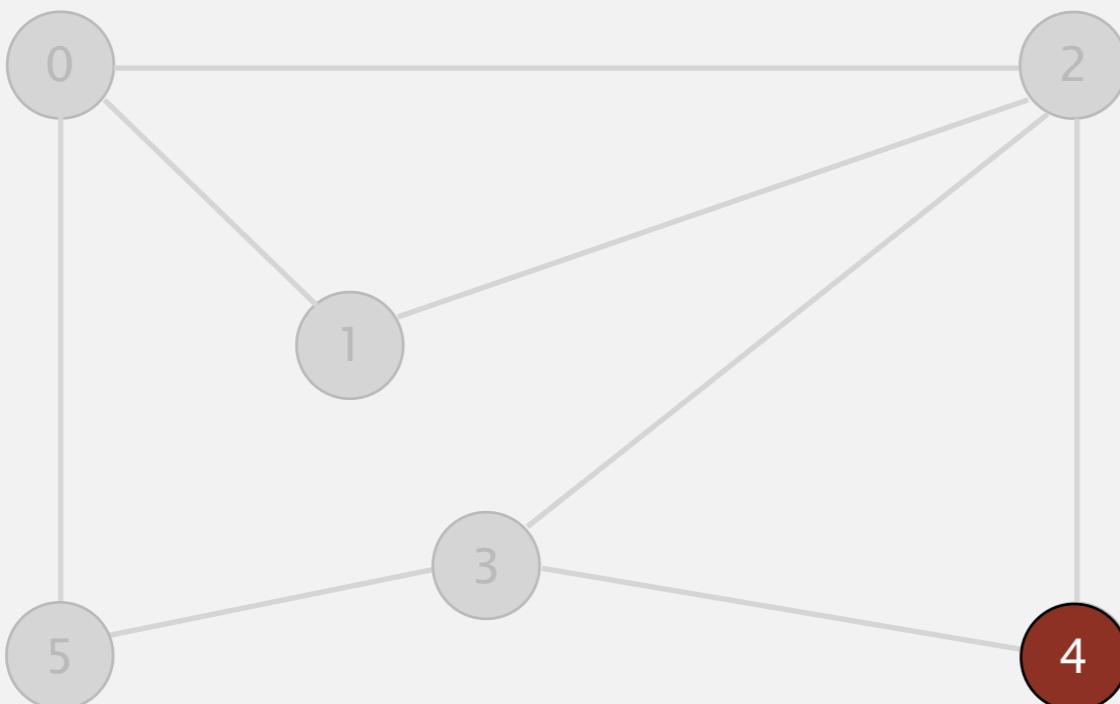
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

3 done

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

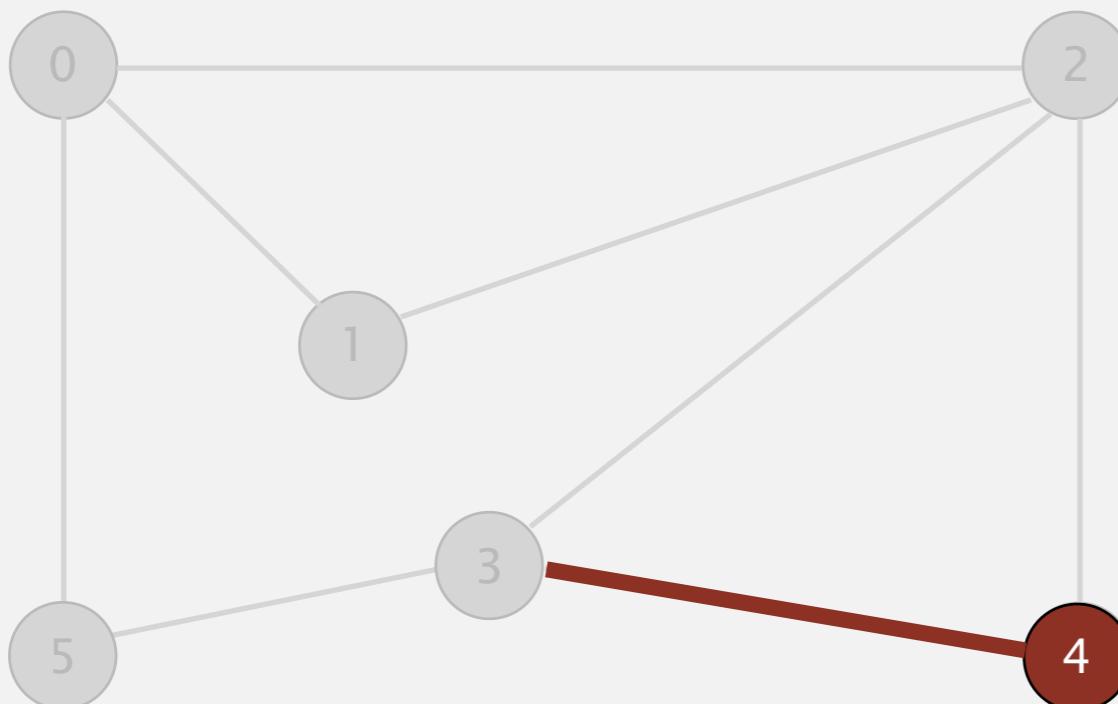
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 4

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

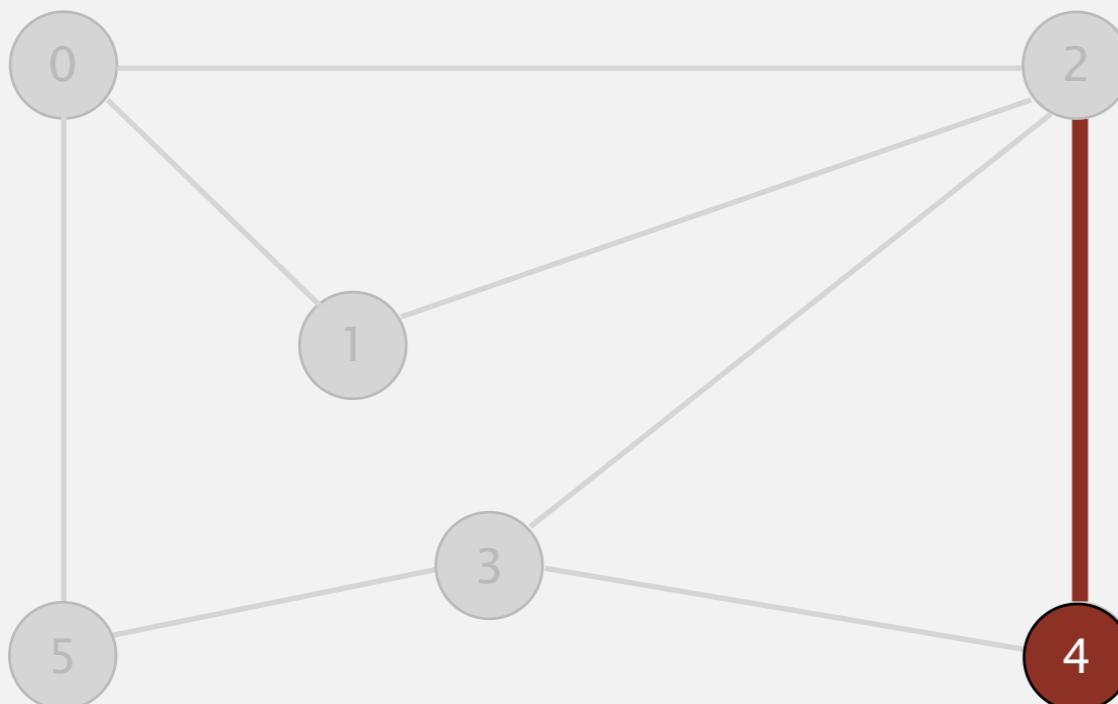
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 4

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



queue

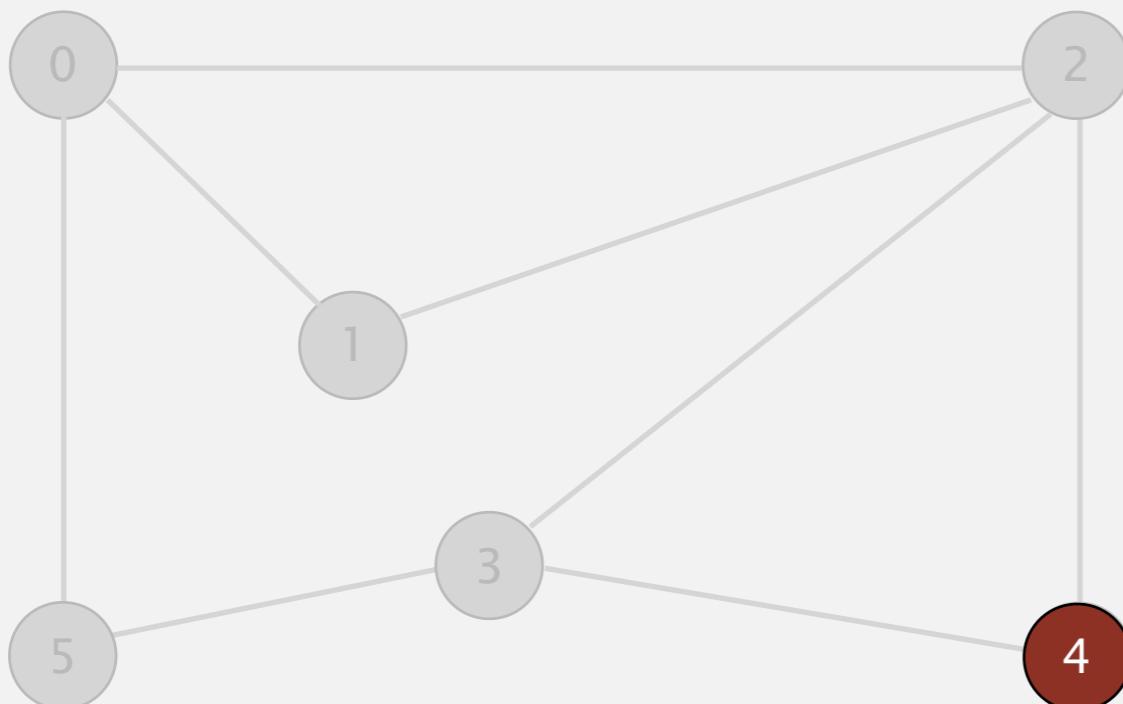
v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

dequeue 4

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



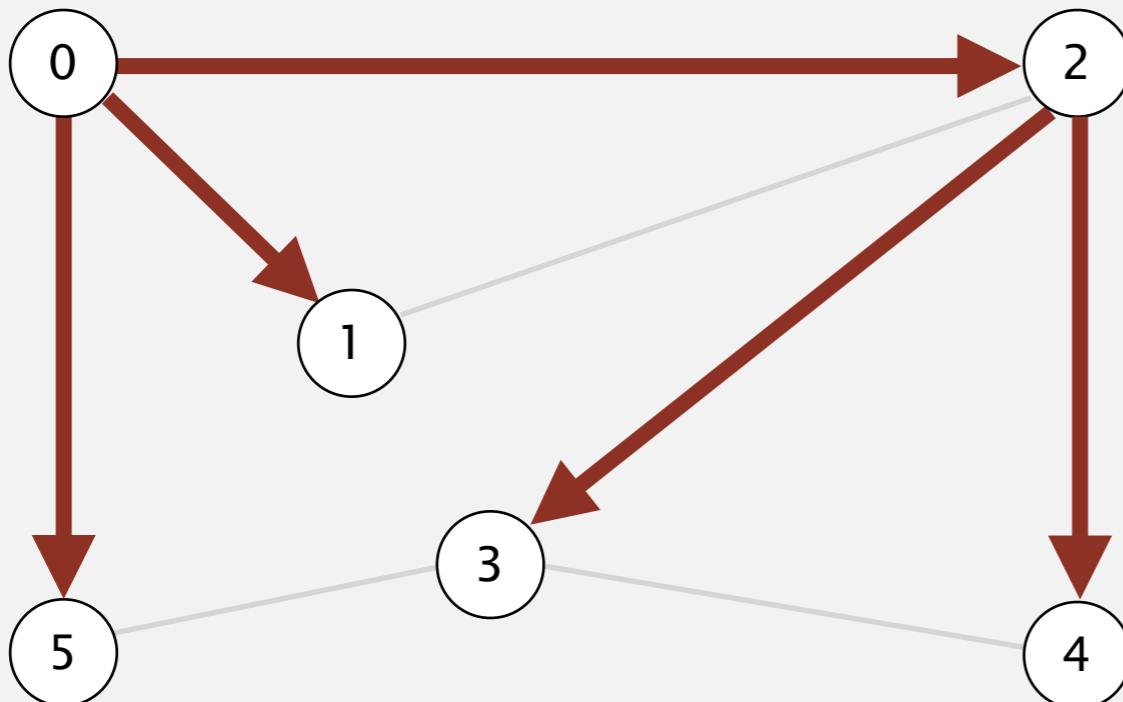
queue

v	edgeTo[v]
0	-
1	0
2	0
3	2
4	2
5	0

Breadth-first search

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



v	$\text{edgeTo}[v]$
0	-
1	0
2	0
3	2
4	2
5	0

done

Breadth-first search

Depth-first search. Put unvisited vertices on a **stack**.

Breadth-first search. Put unvisited vertices on a **queue**.

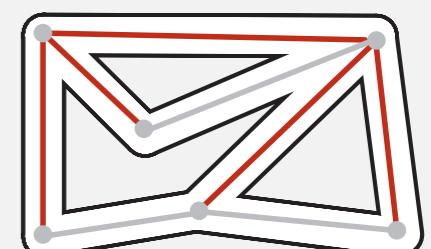
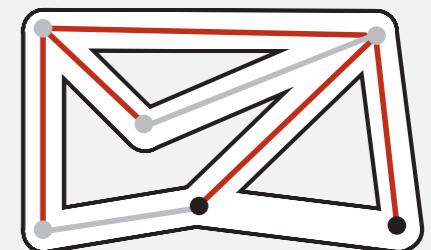
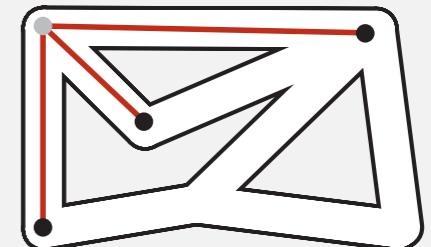
Shortest path. Find path from s to t that uses **fewest number of edges**.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v
- add each of v 's unvisited neighbors to the queue,
and mark them as visited.



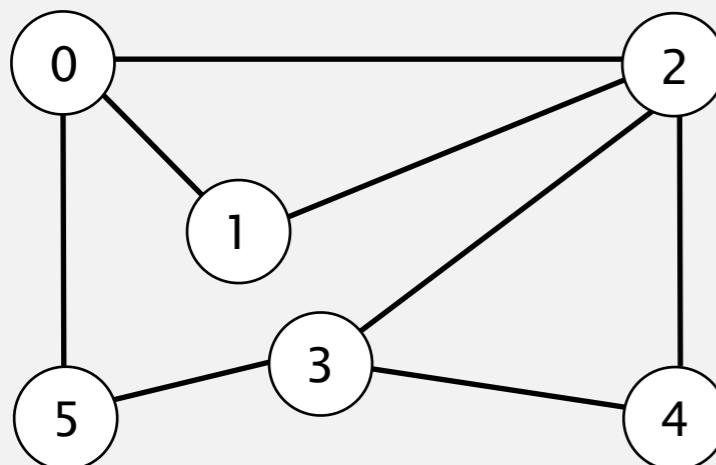
Intuition. BFS examines vertices in increasing distance from s .

Breadth-first search properties

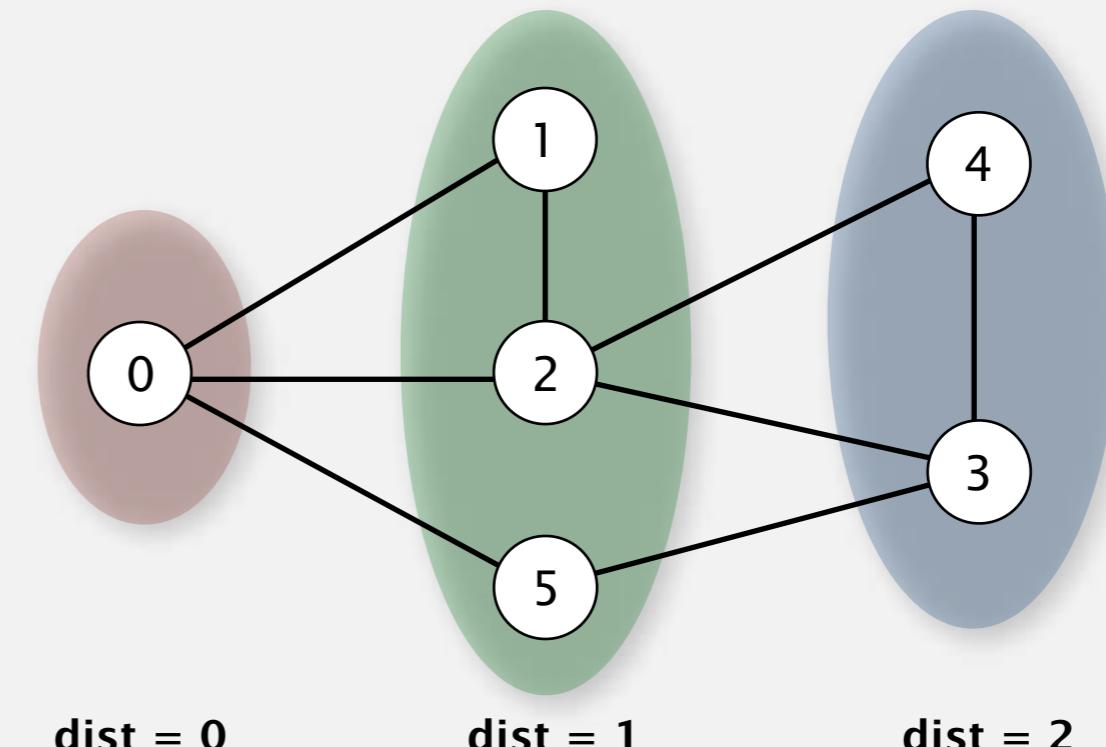
Proposition. BFS computes shortest path (number of edges) from s in a connected graph in time proportional to $E + V$.

Pf. [correctness] Queue always consists of zero or more vertices of distance k from s , followed by zero or more vertices of distance $k + 1$.

Pf. [running time] Each vertex connected to s is visited once.



standard drawing



dist = 0

dist = 1

dist = 2

Breadth-first search

```
public class BreadthFirstPaths
{
    private boolean[] marked;
    private boolean[] edgeTo[];
    private final int s;
    ...

    private void bfs(Graph G, int s)
    {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        while (!q.isEmpty())
        {
            int v = q.dequeue();
            for (int w : G.adj(v))
            {
                if (!marked[w])
                {
                    q.enqueue(w);
                    marked[w] = true;
                    edgeTo[w] = v;
                }
            }
        }
    }
}
```

UNDIRECTED GRAPHS

- ▶ Graph API
- ▶ Depth-first search
- ▶ Breadth-first search
- ▶ Connected components
- ▶ Challenges

Connectivity queries

Def. Vertices v and w are **connected** if there is a path between them.

Goal. Preprocess graph to answer queries: is v connected to w ?
in **constant time**.

```
public class CC
```

CC(Graph G)	<i>find connected components in G</i>
boolean connected(int v, int w)	<i>are v and w connected?</i>
int count()	<i>number of connected components</i>
int id(int v)	<i>component identifier for v</i>

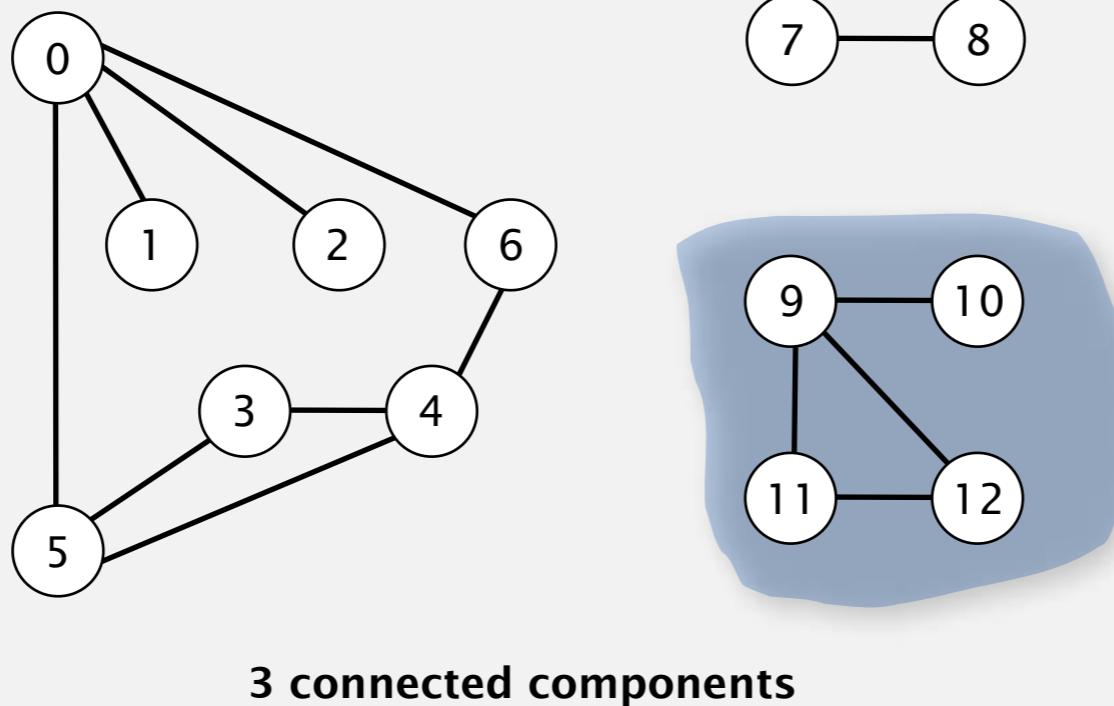
Depth-first search. [next few slides]

Connected components

The relation "is connected to" is an **equivalence relation**:

- Reflexive: v is connected to v .
- Symmetric: if v is connected to w , then w is connected to v .
- Transitive: if v connected to w and w connected to x , then v connected to x .

Def. A **connected component** is a maximal set of connected vertices.

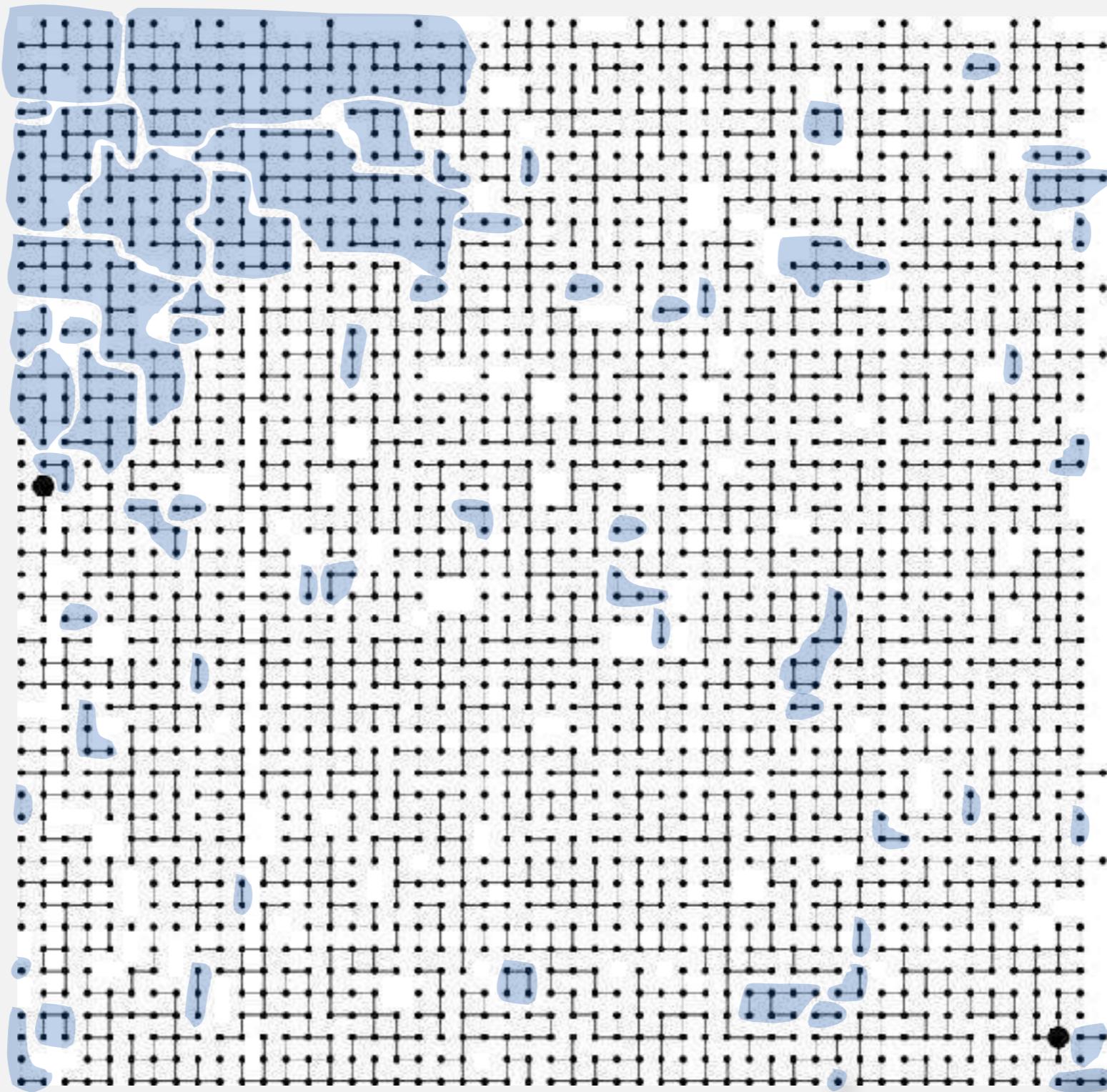


v	$\text{id}[v]$
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	1
9	2
10	2
11	2
12	2

Remark. Given connected components, can answer queries in constant time.

Connected components

Def. A **connected component** is a maximal set of connected vertices.



63 connected components

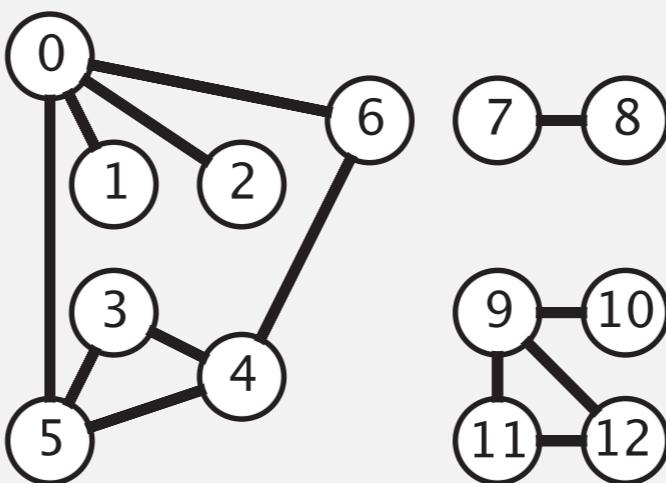
Connected components

Goal. Partition vertices into connected components.

Connected components

Initialize all vertices v as unmarked.

For each unmarked vertex v , run DFS to identify all vertices discovered as part of the same component.

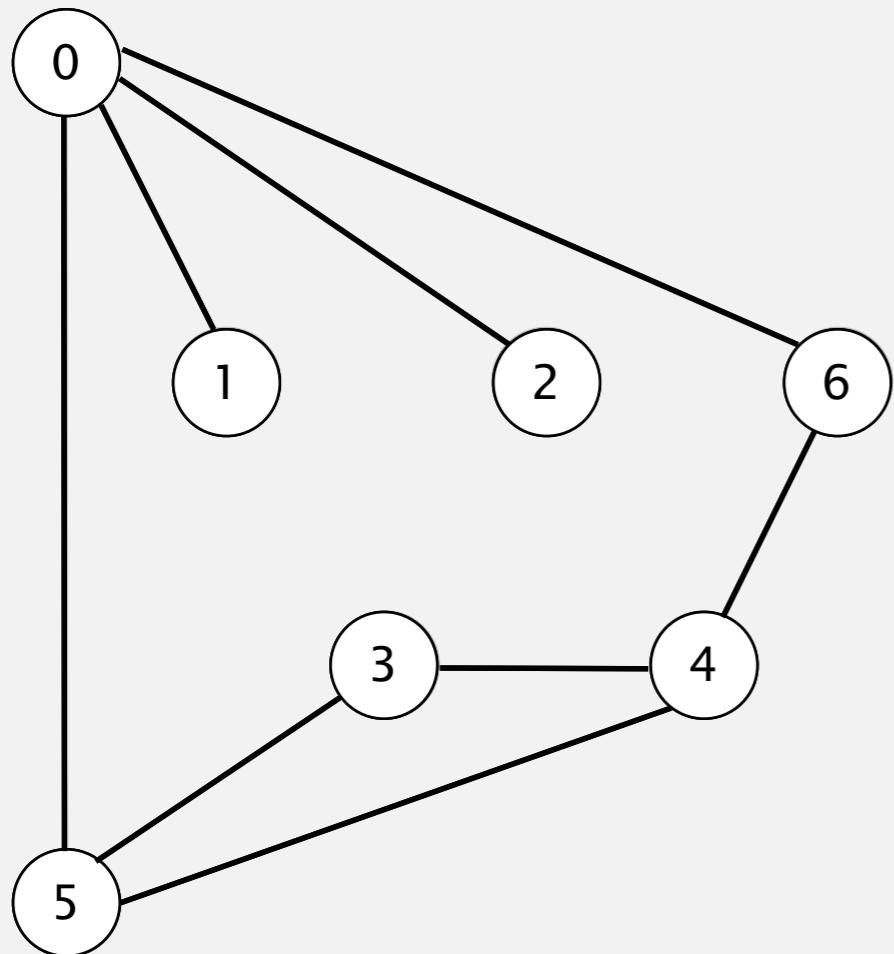


tinyG.txt
V → 13
13 ← E
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3

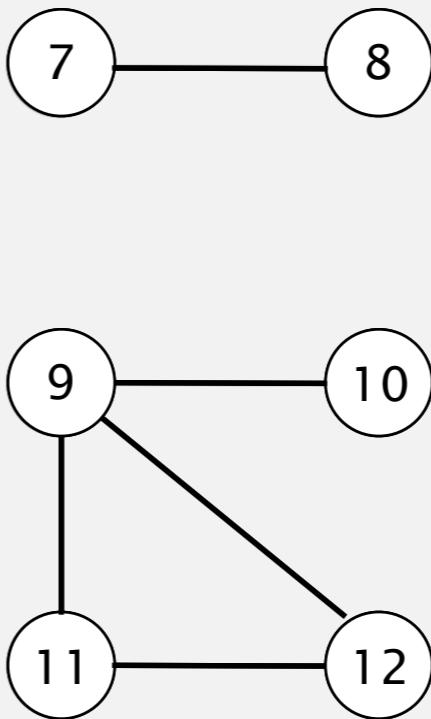
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



graph G

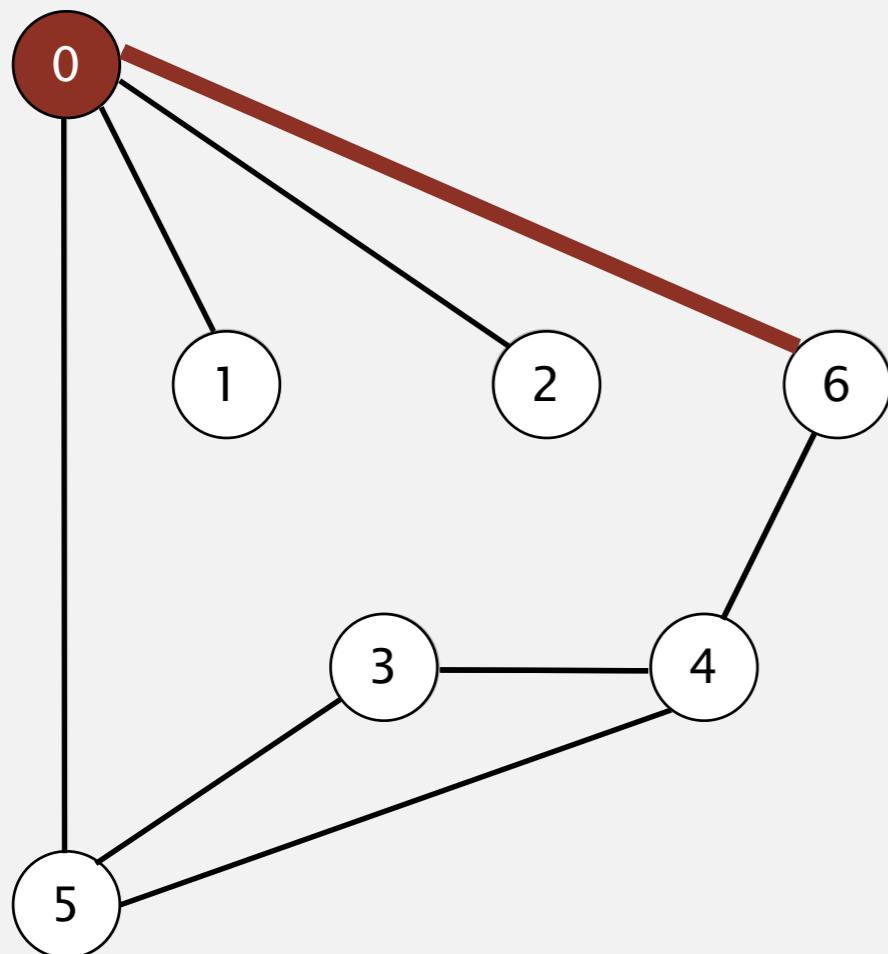


v	marked[]	cc[]
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

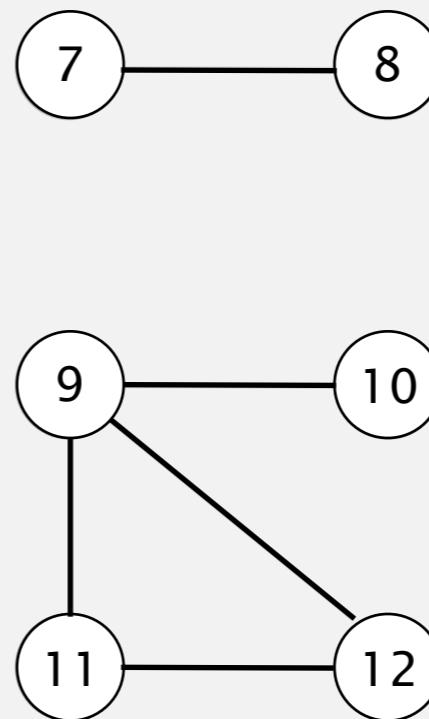
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 0

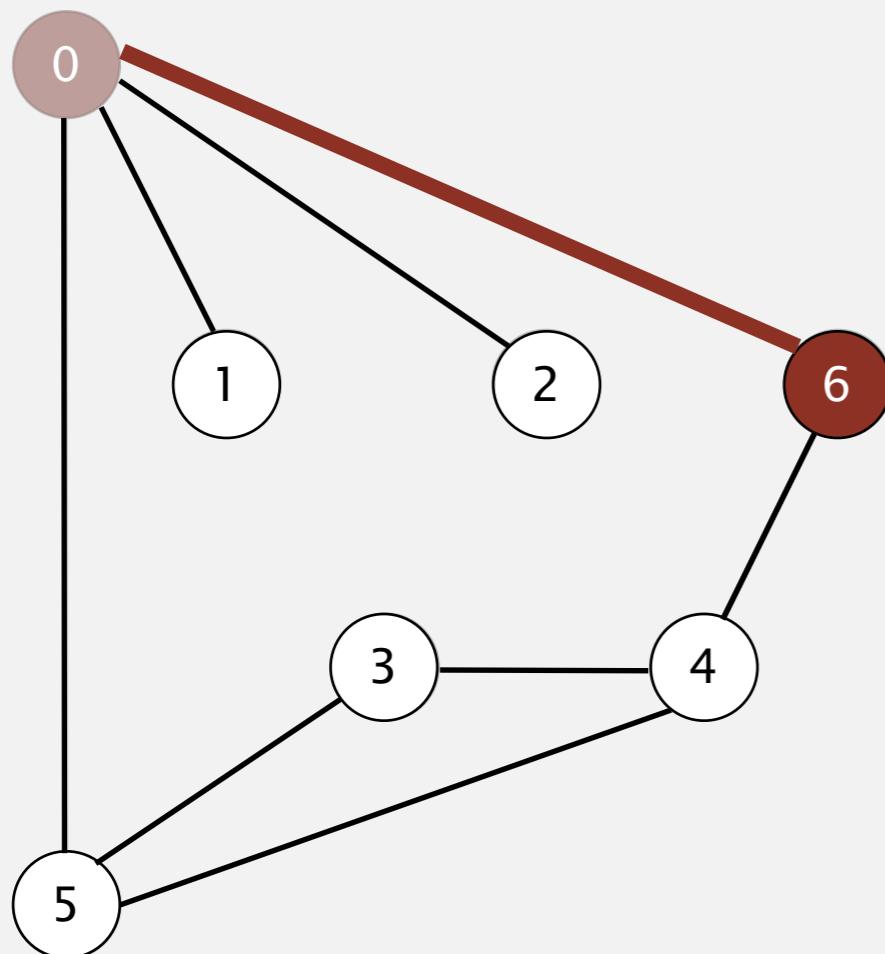


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

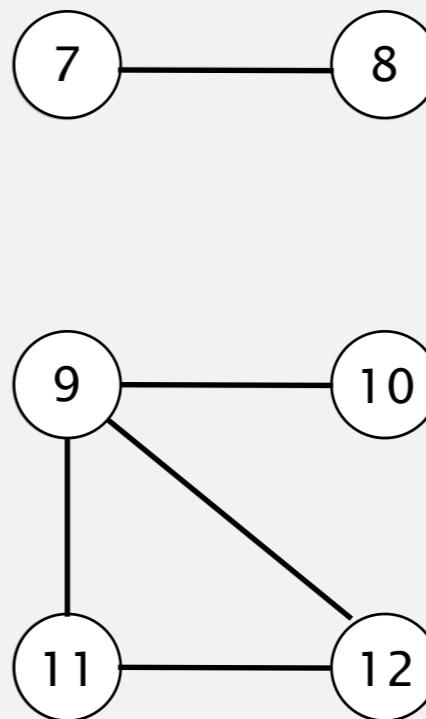
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 6

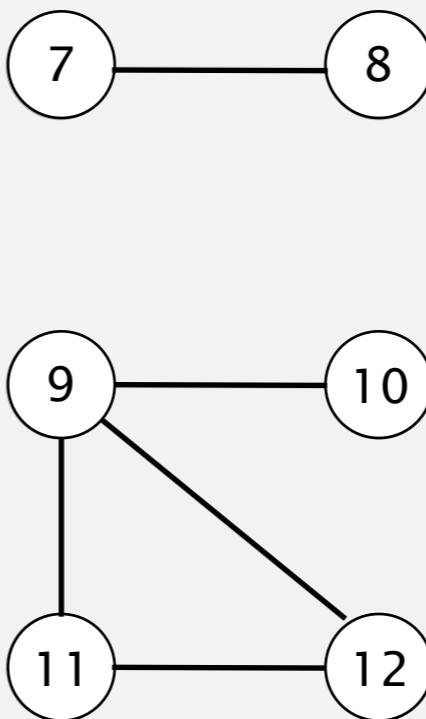
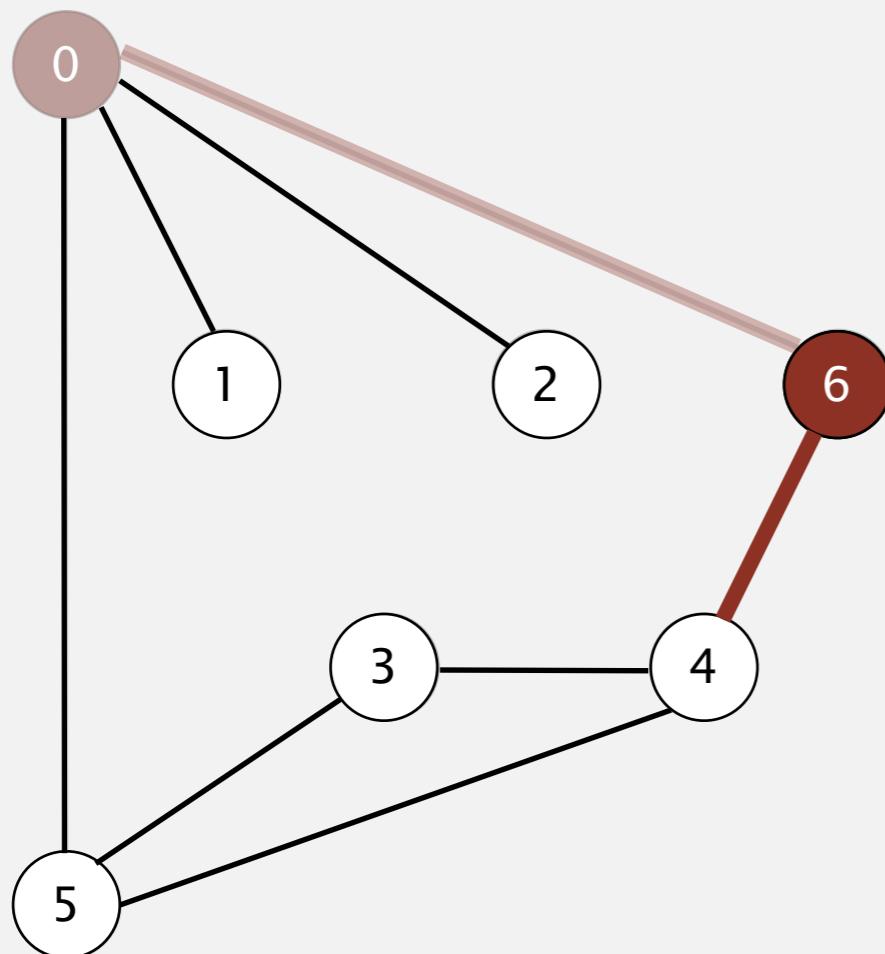


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

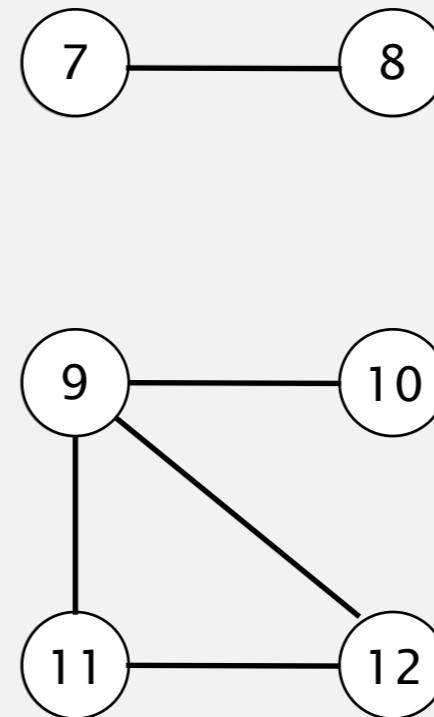
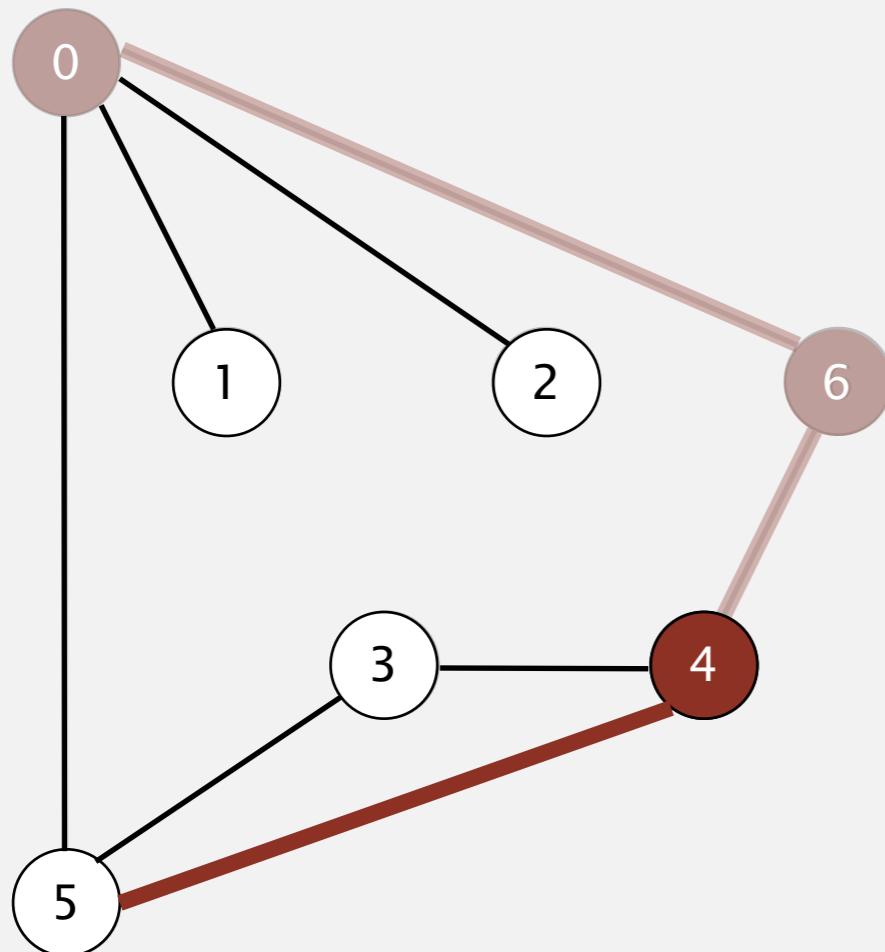


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

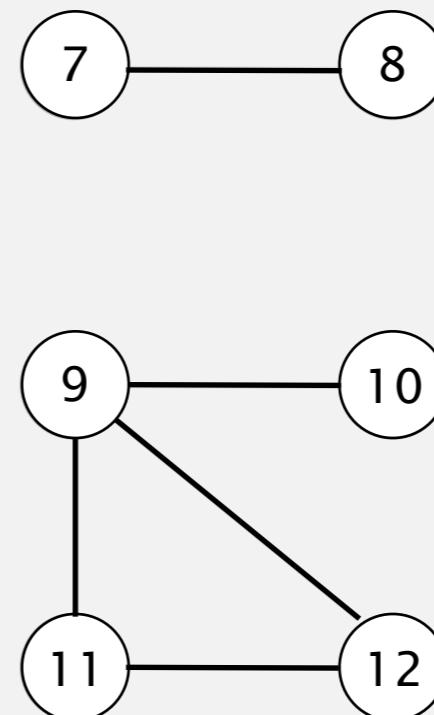
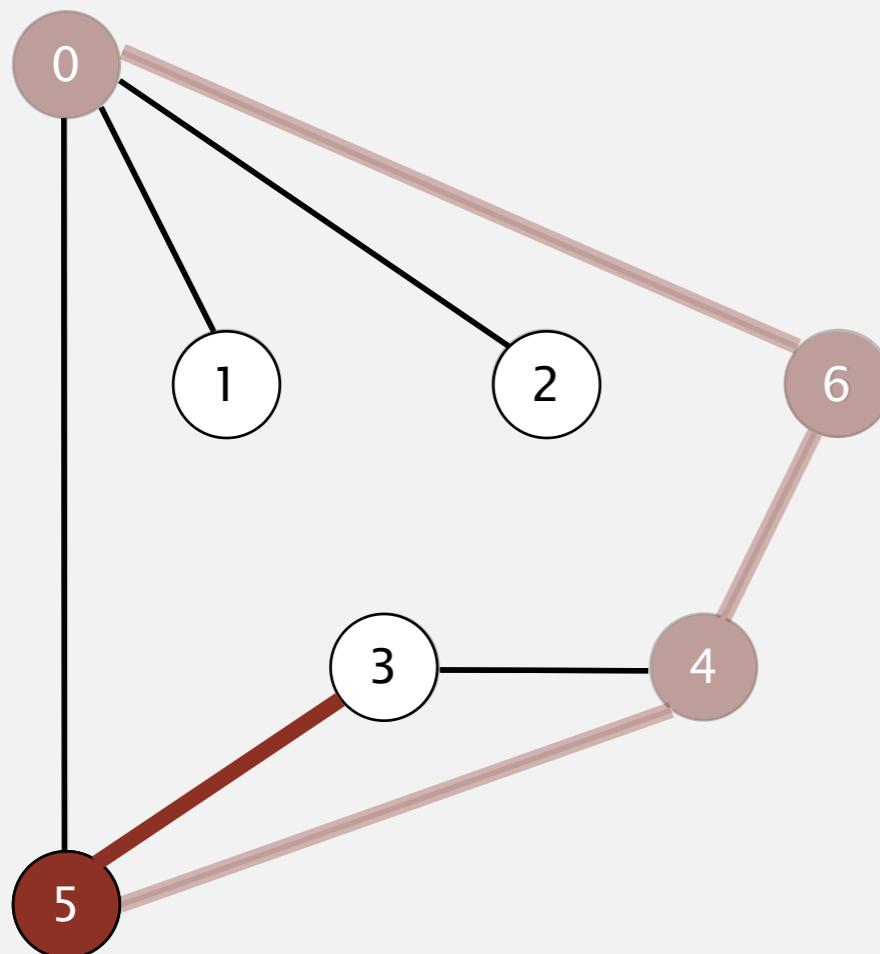


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	F	-
4	T	0
5	F	-
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

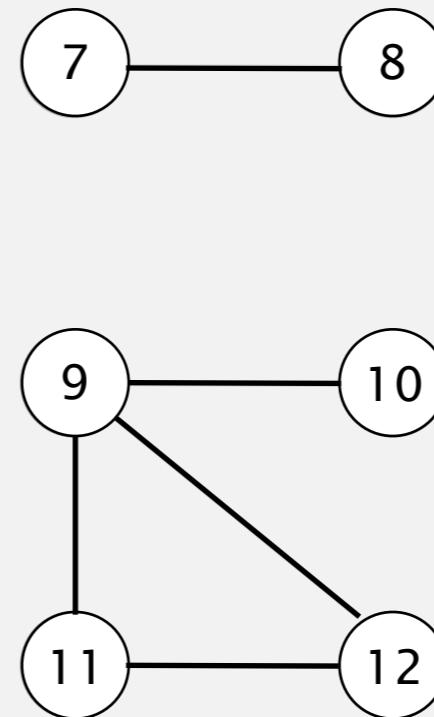
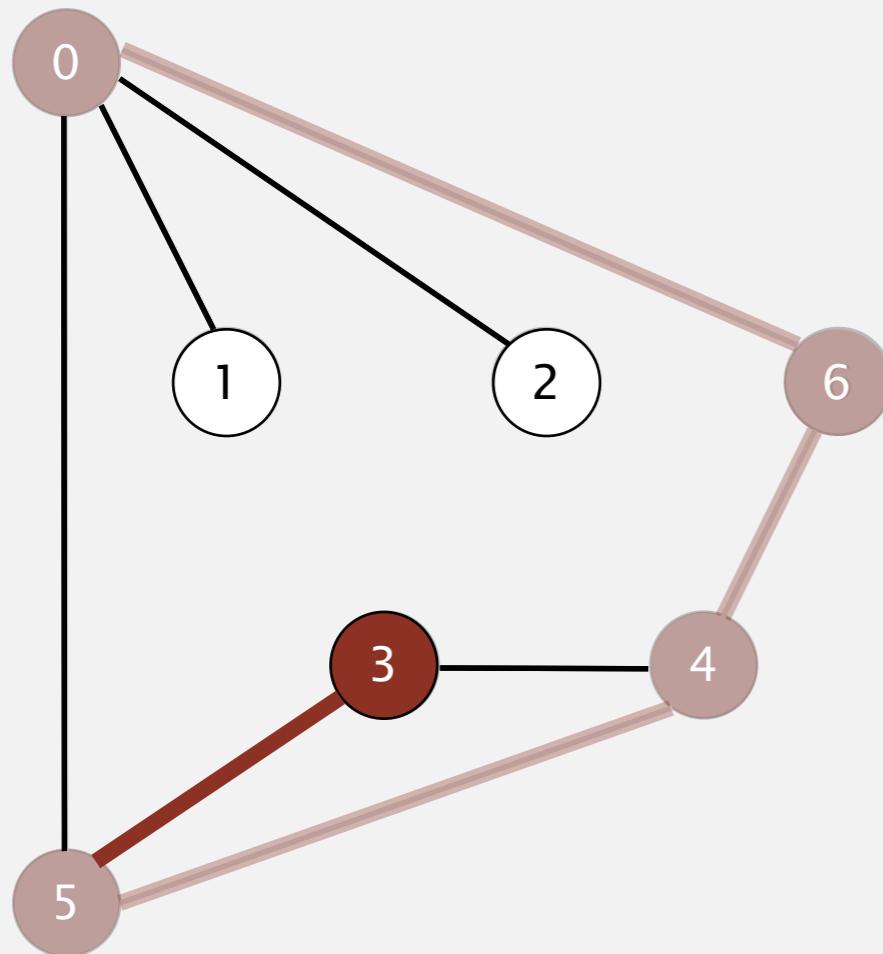


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	F	-
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

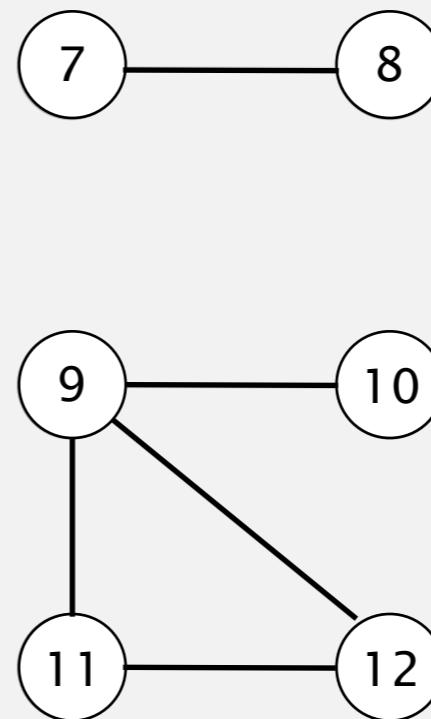
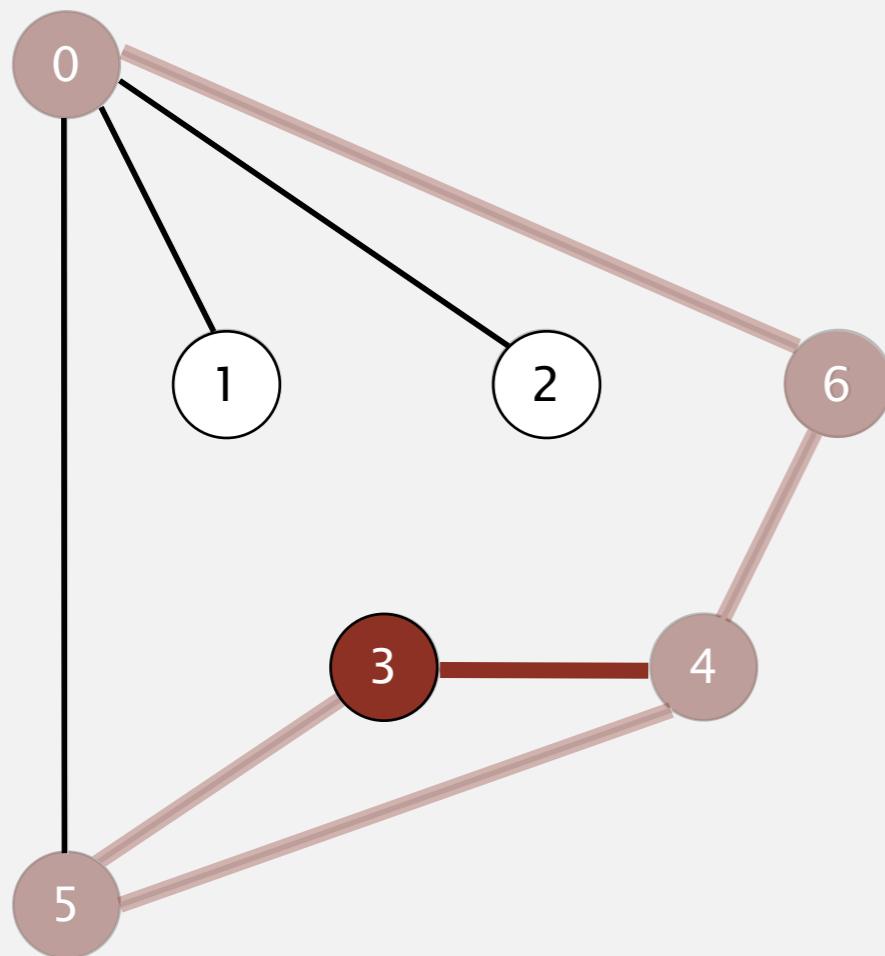


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

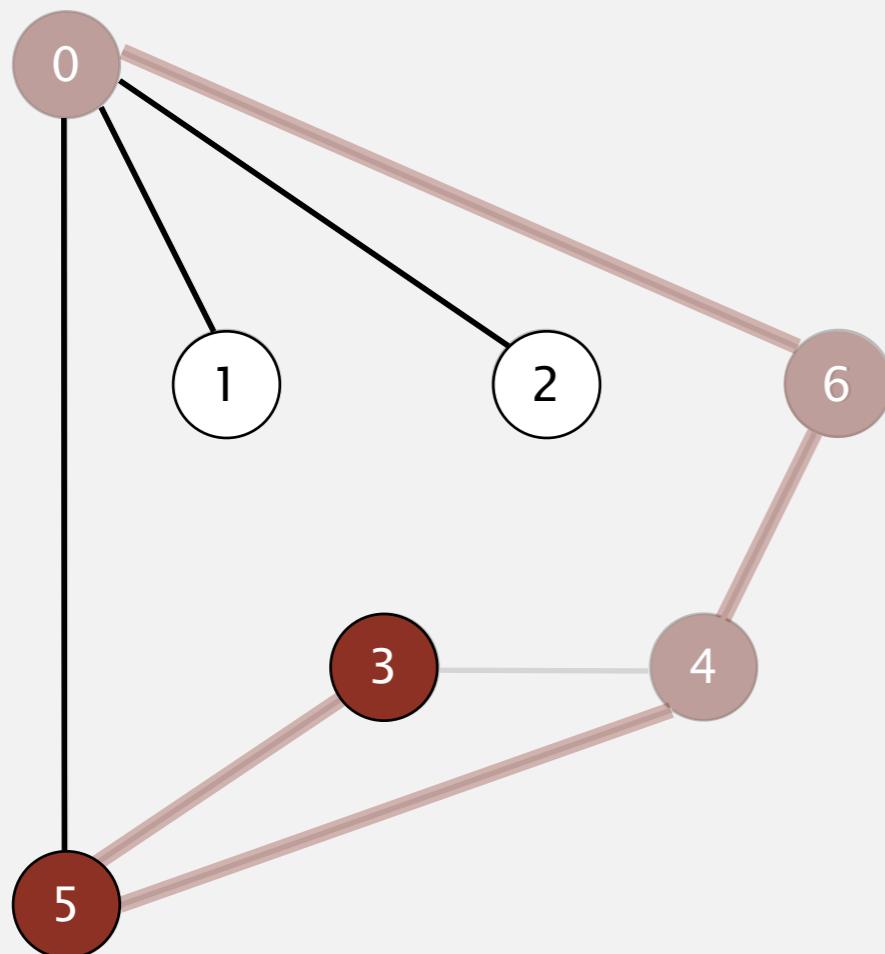


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

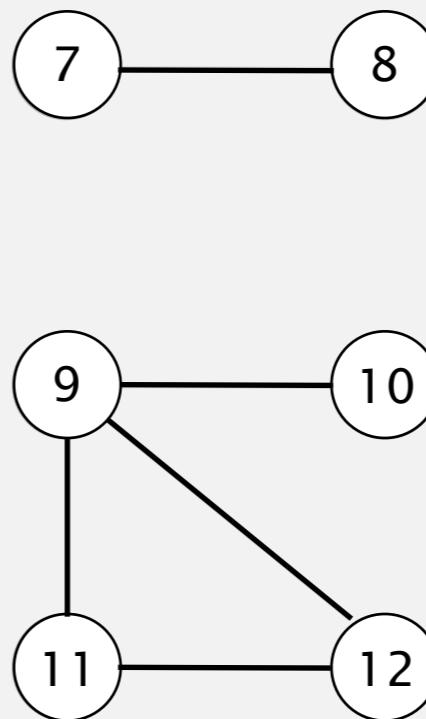
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



3 done

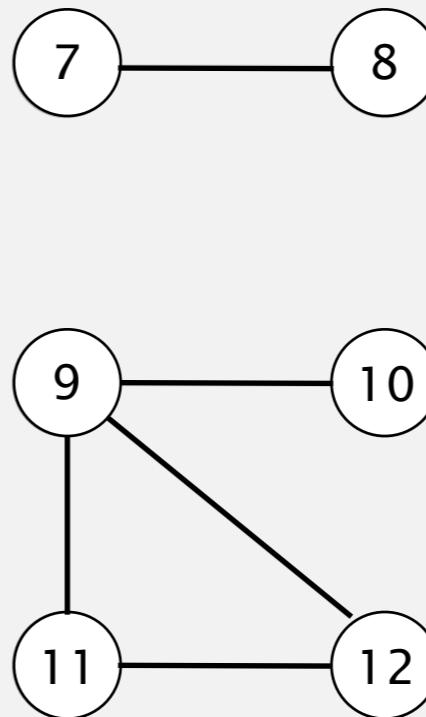
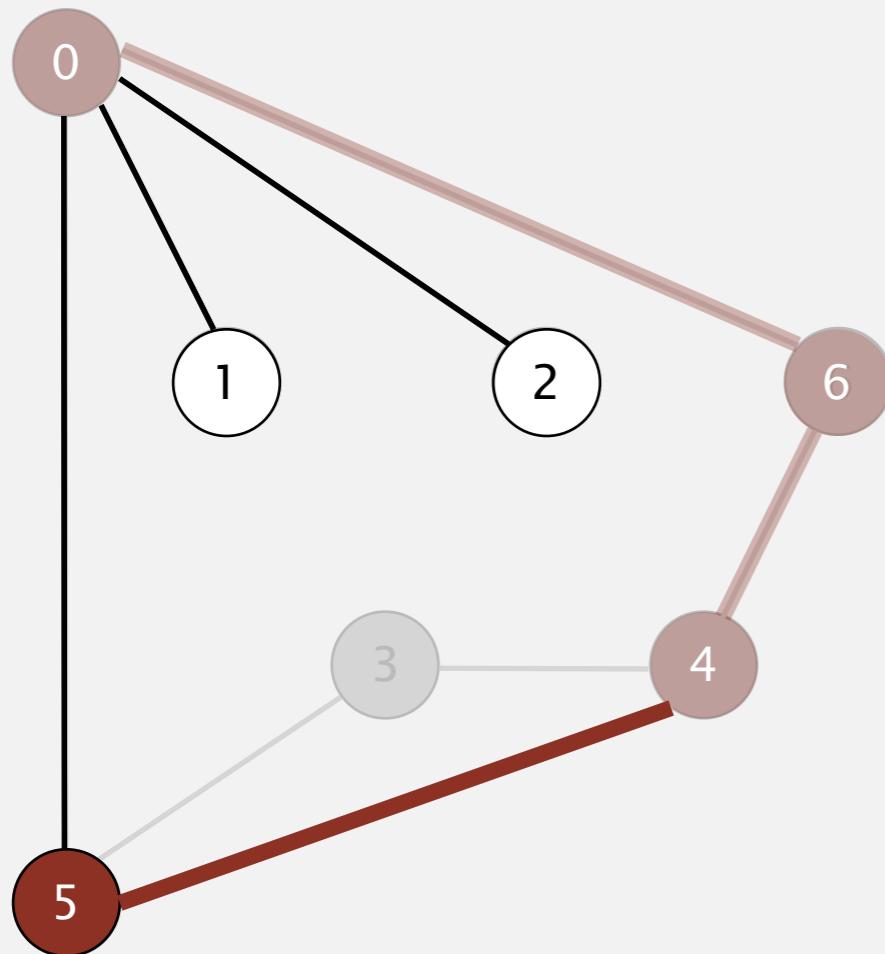


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

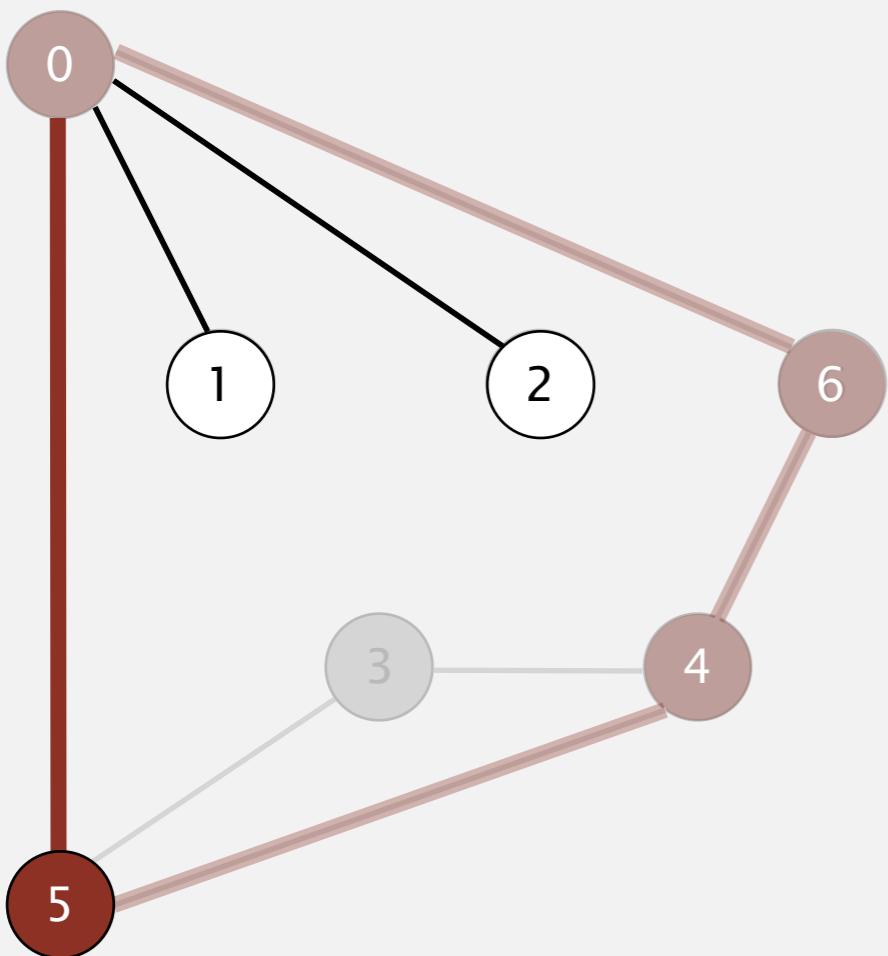


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

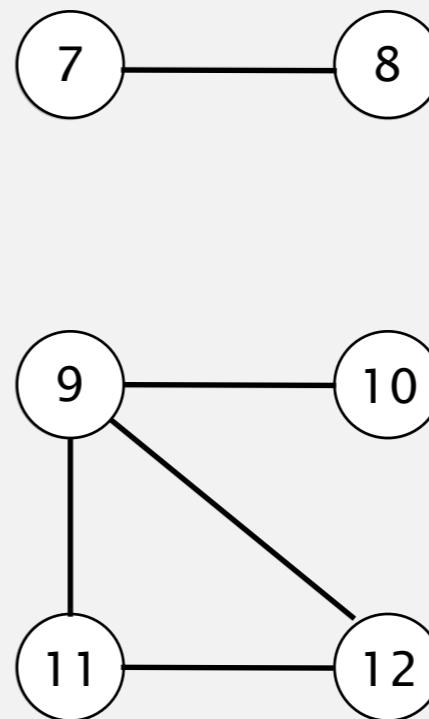
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 5

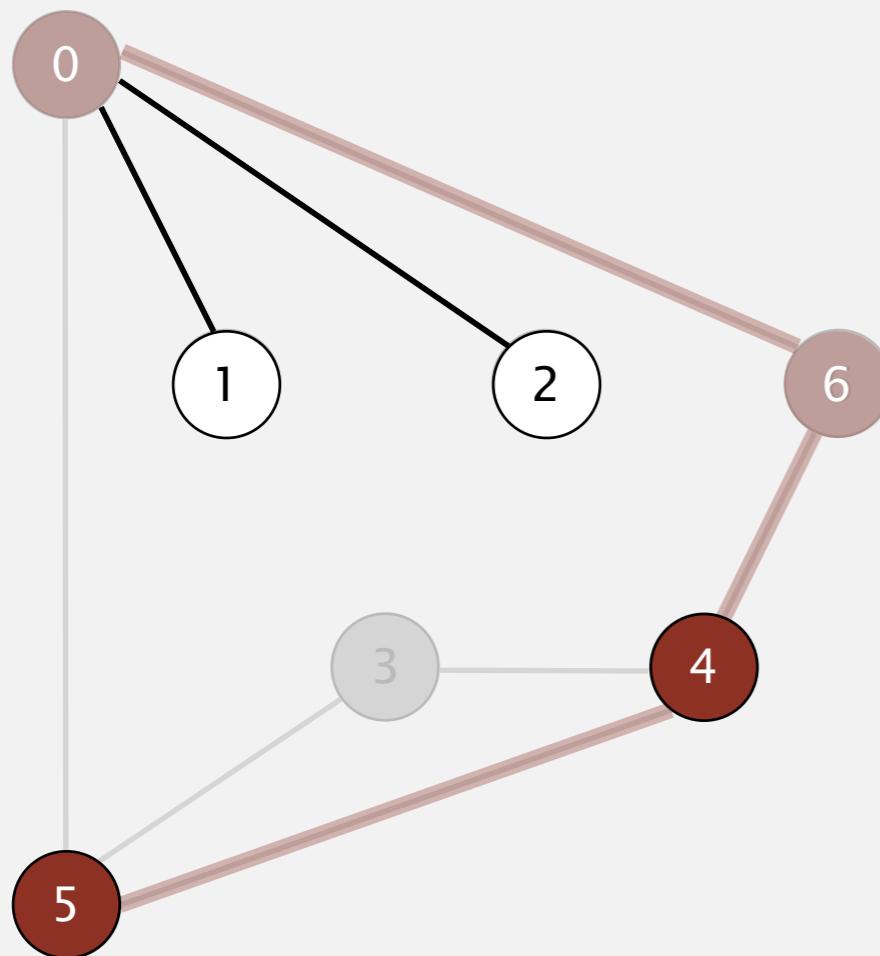


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

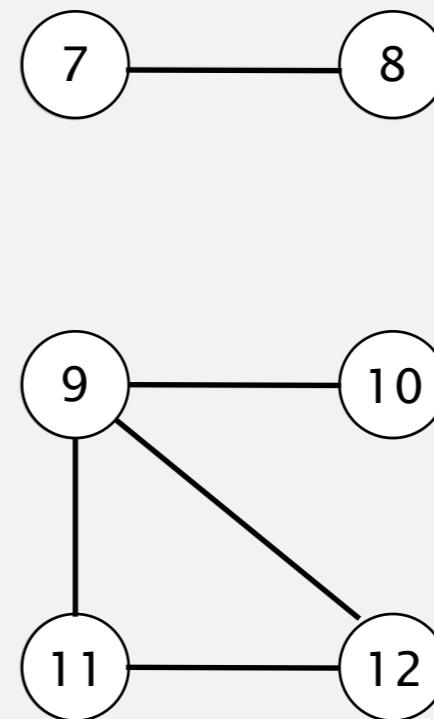
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



5 done

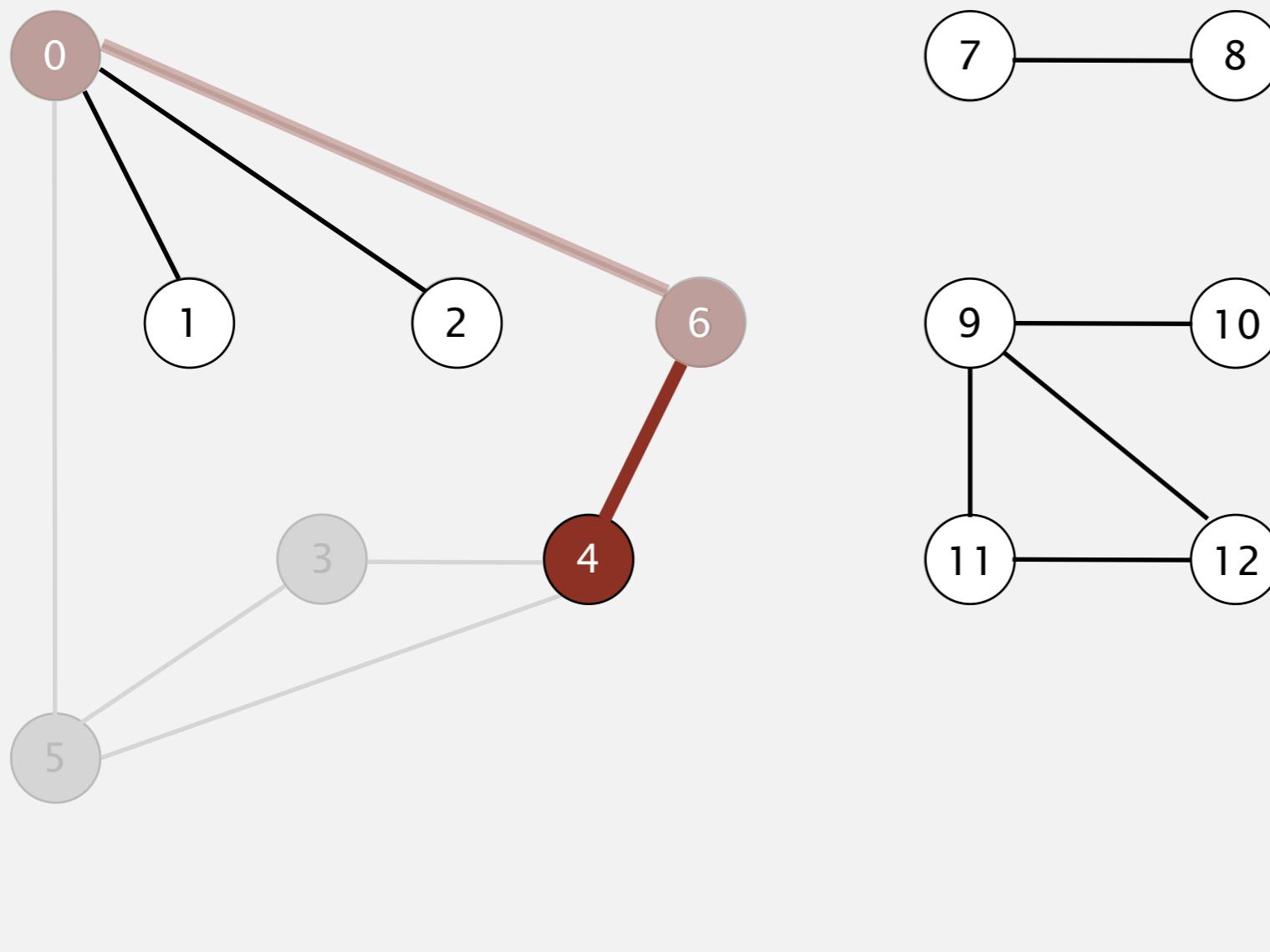


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

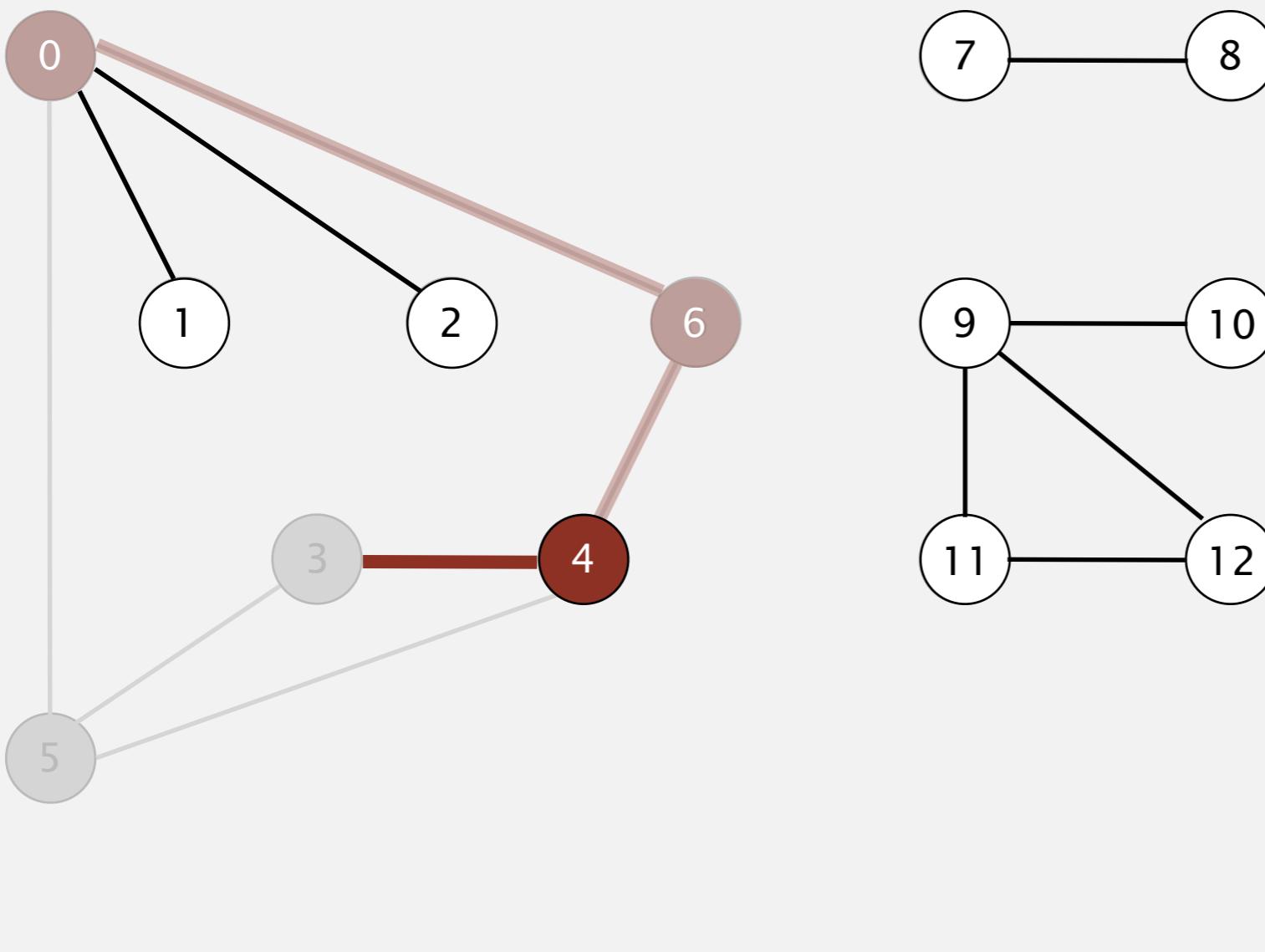


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

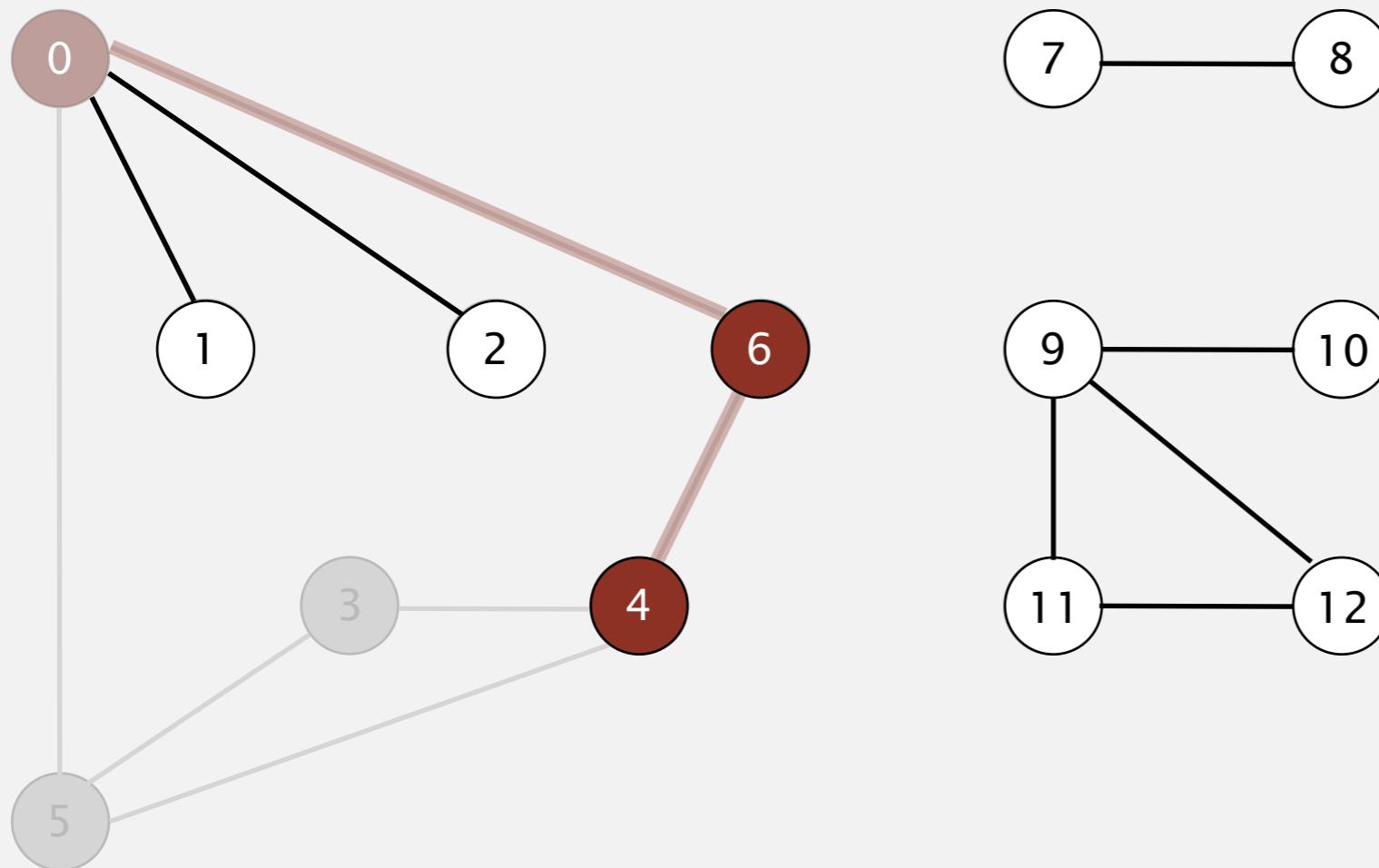


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



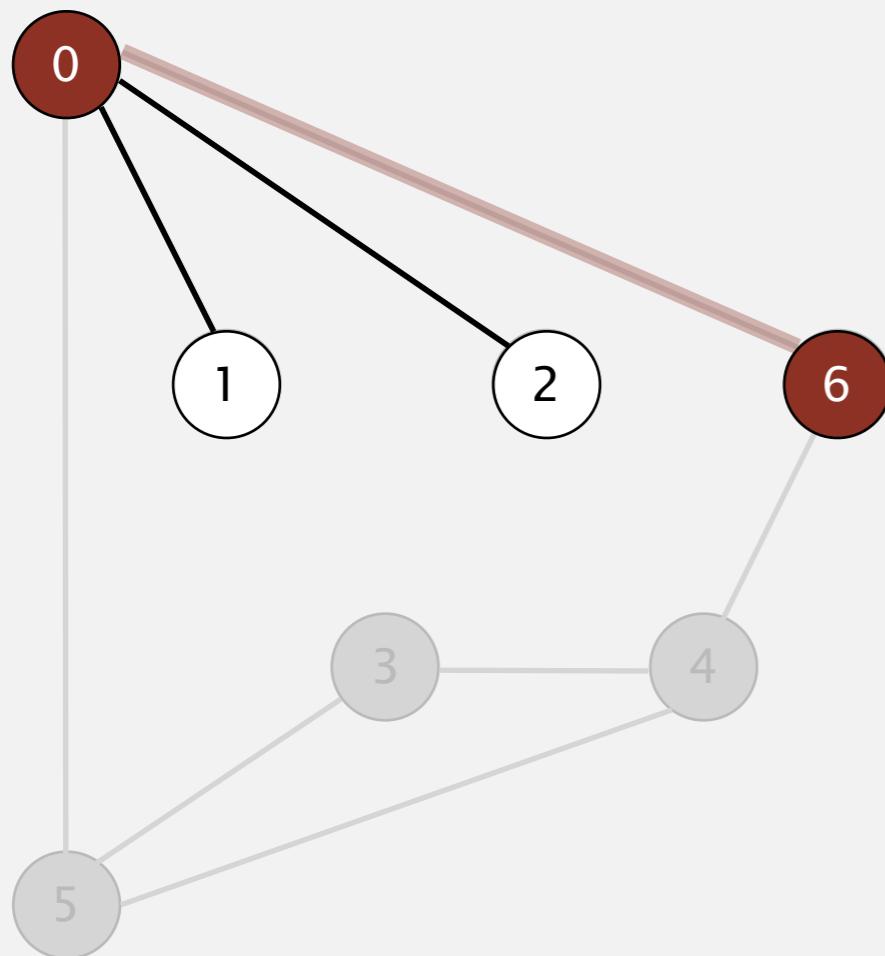
4 done

v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

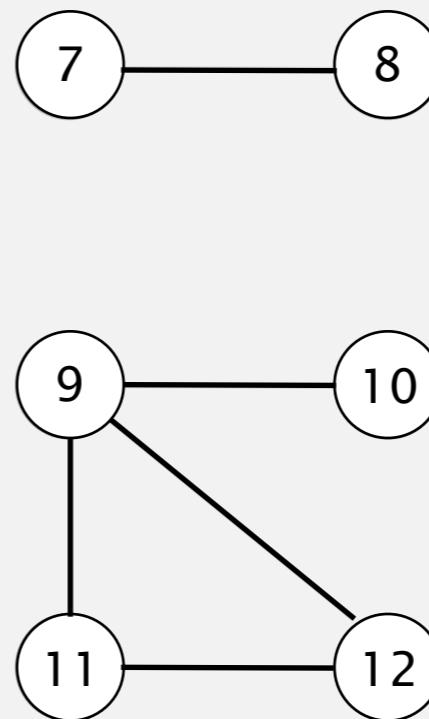
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



6 done

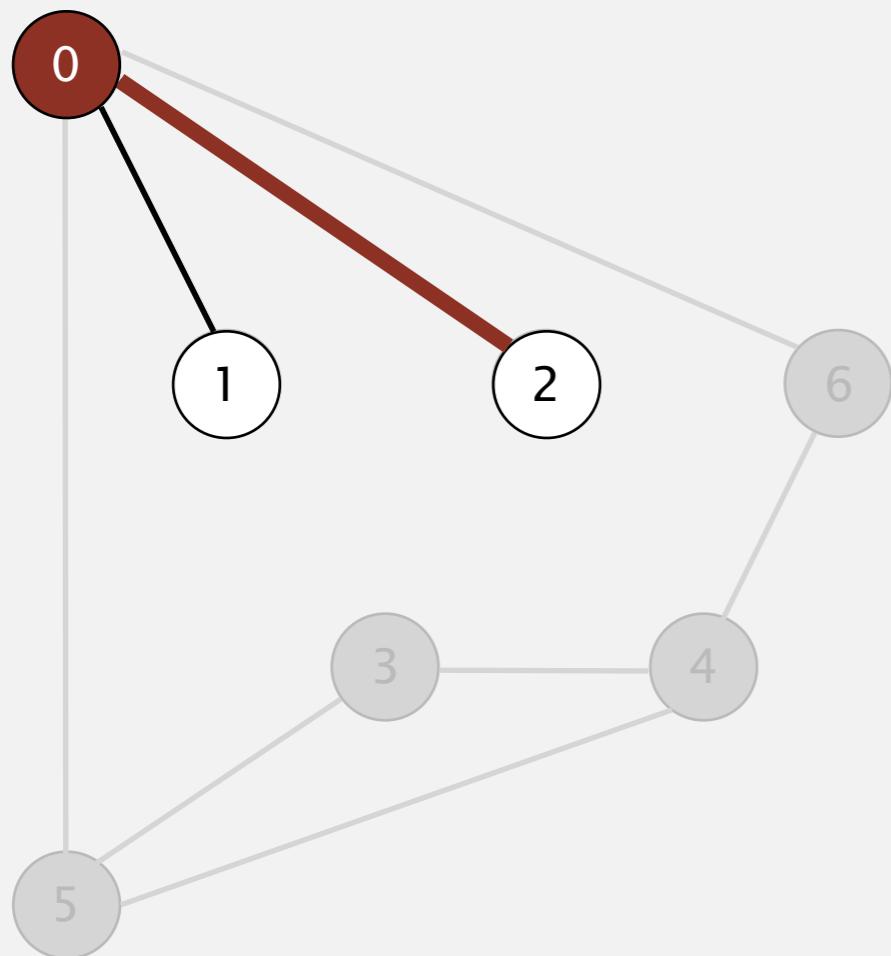


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

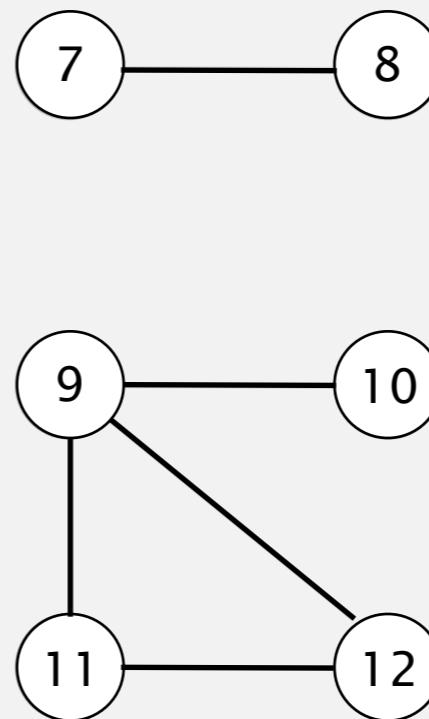
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 0

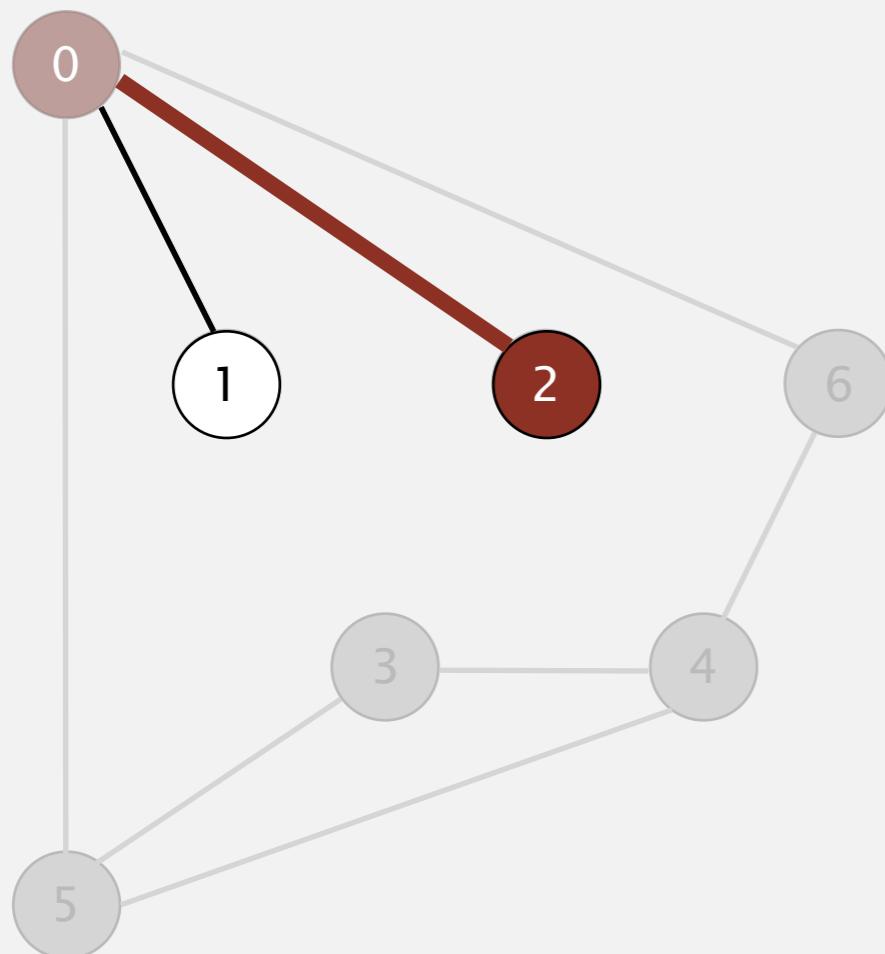


v	marked[]	cc[]
0	T	0
1	F	-
2	F	-
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

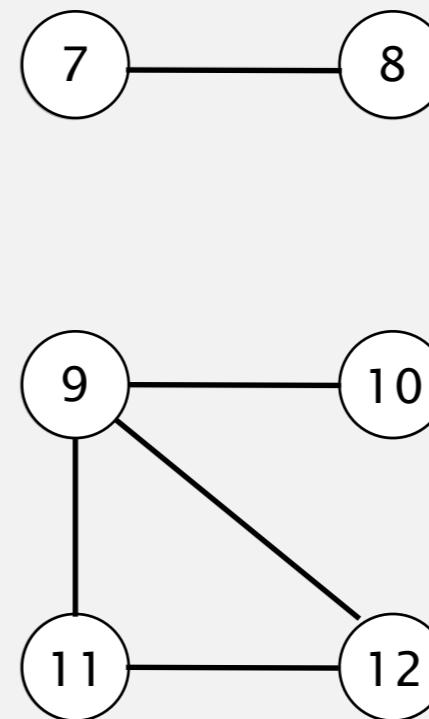
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 2

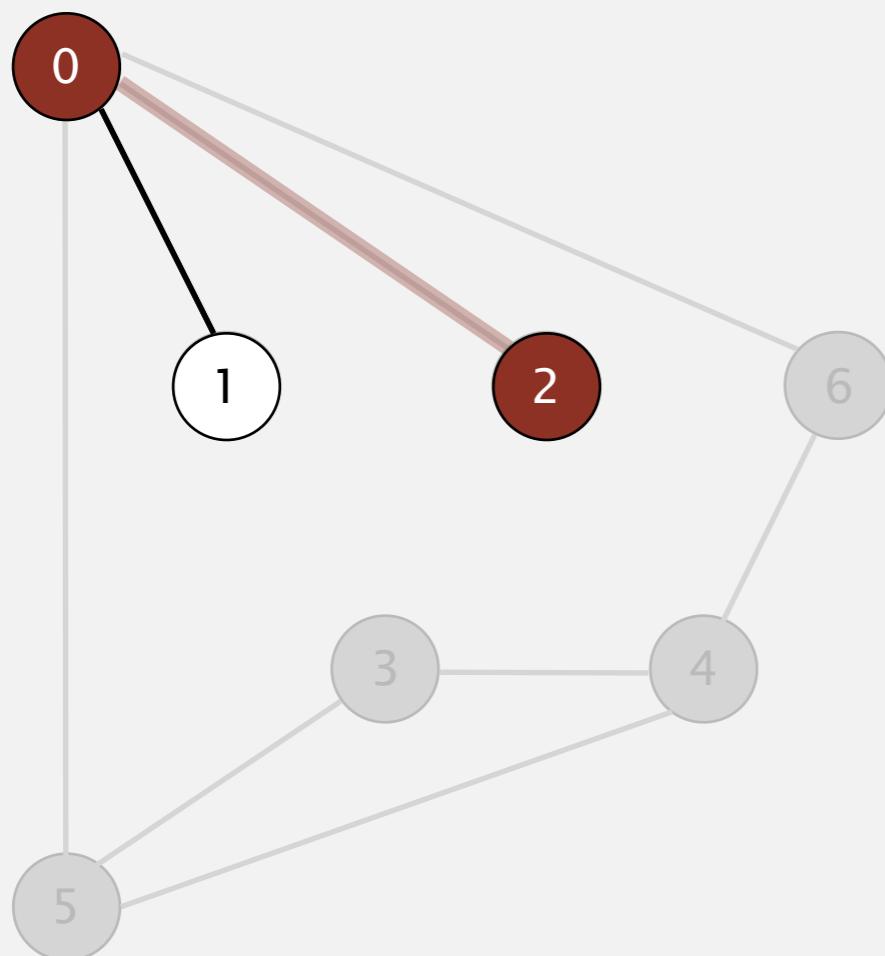


v	marked[]	cc[]
0	T	0
1	F	-
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

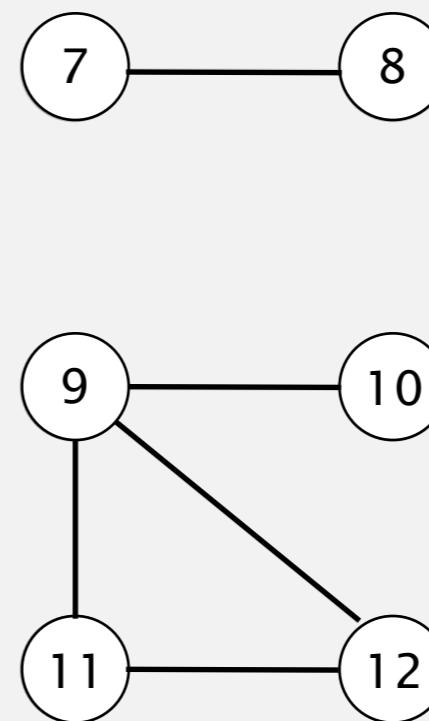
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



2 done

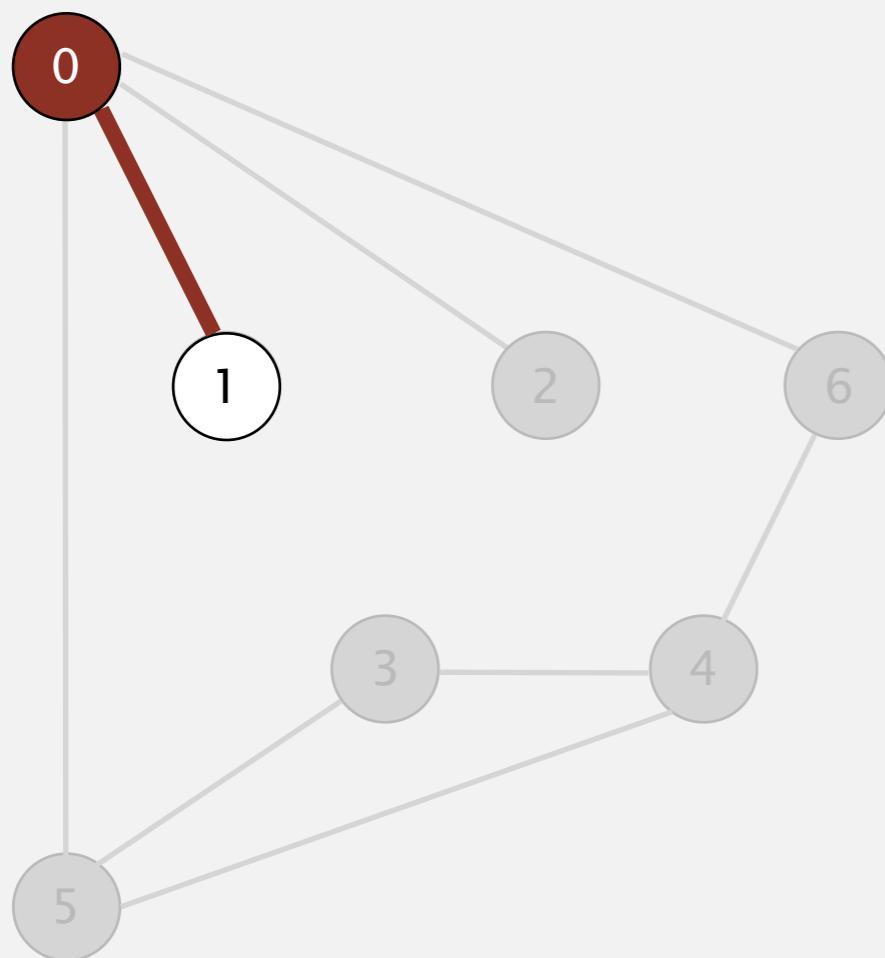


v	marked[]	cc[]
0	T	0
1	F	-
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

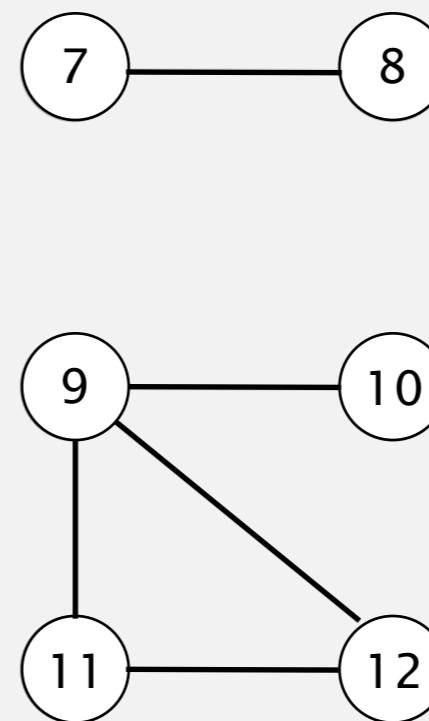
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 0

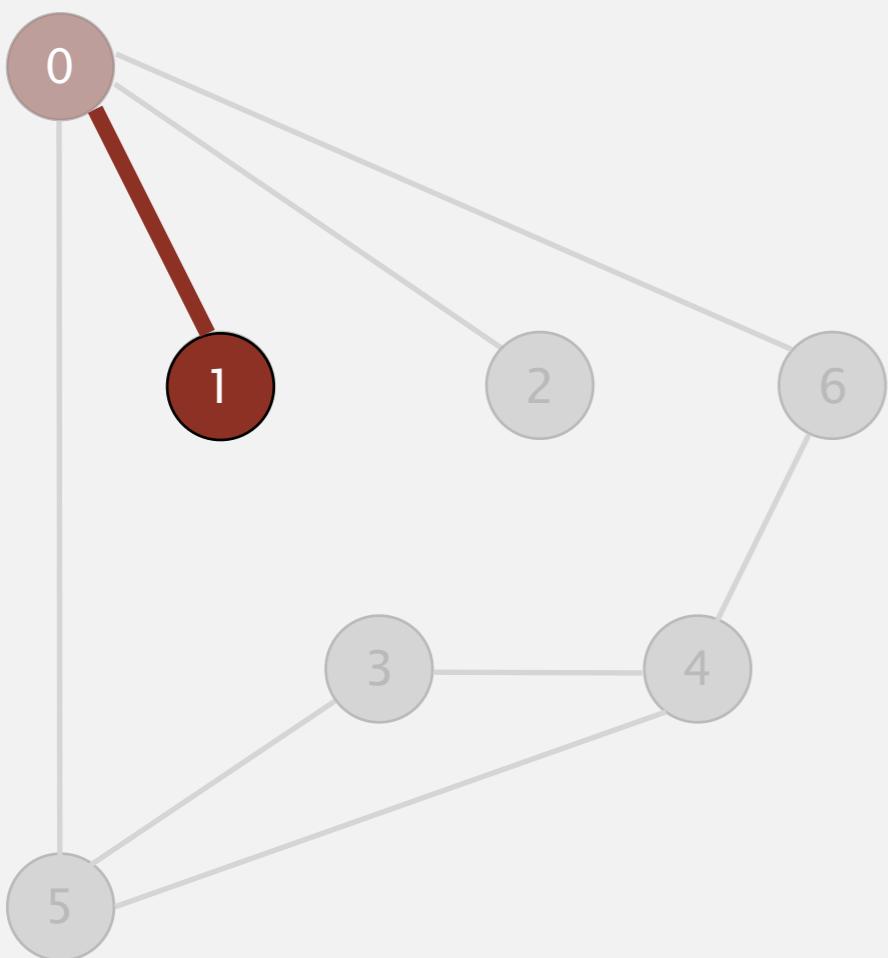


v	marked[]	cc[]
0	T	0
1	F	-
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

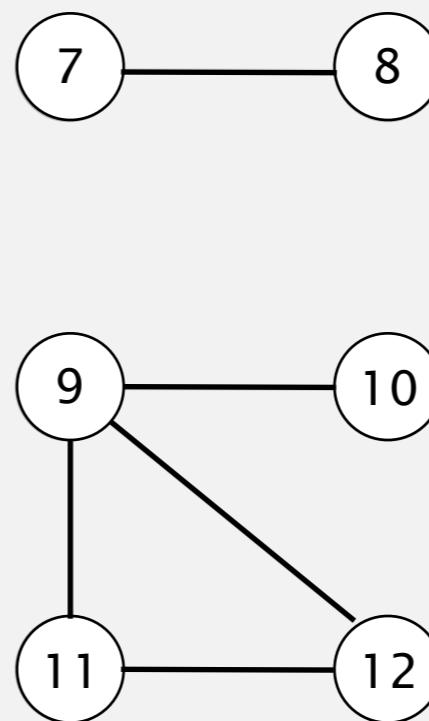
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 1

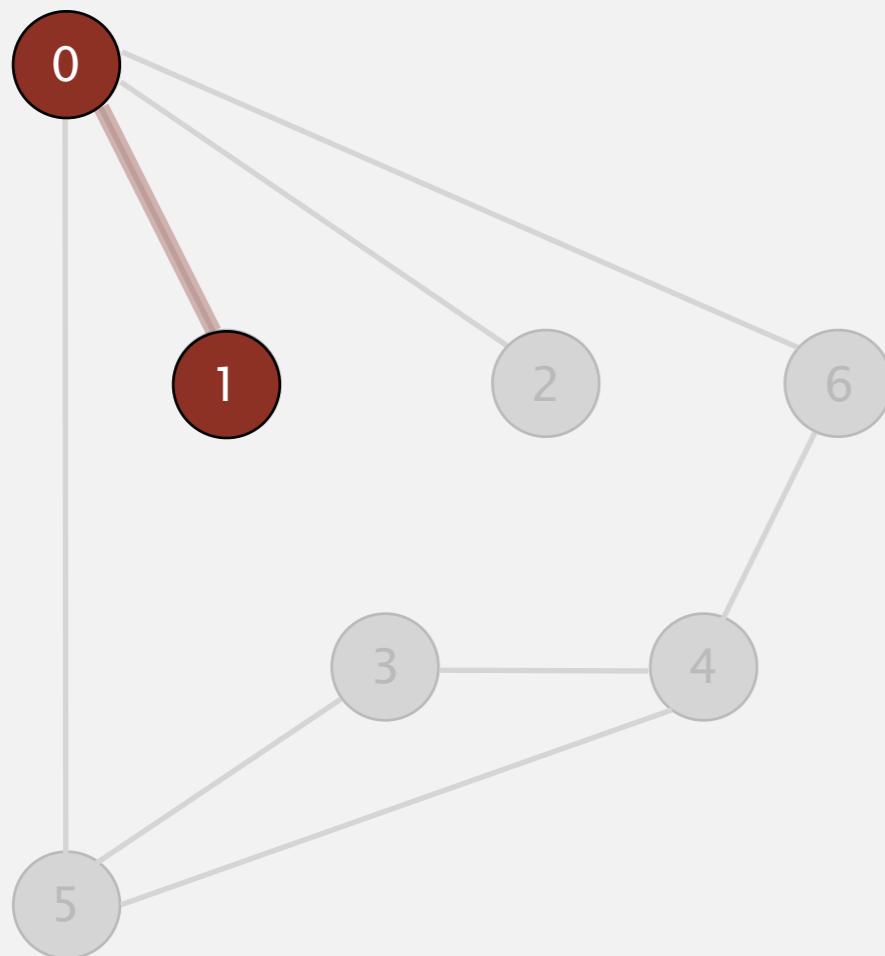


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

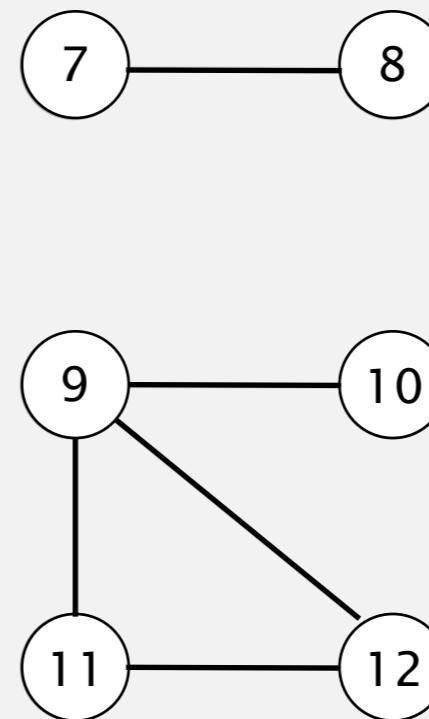
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



1 done

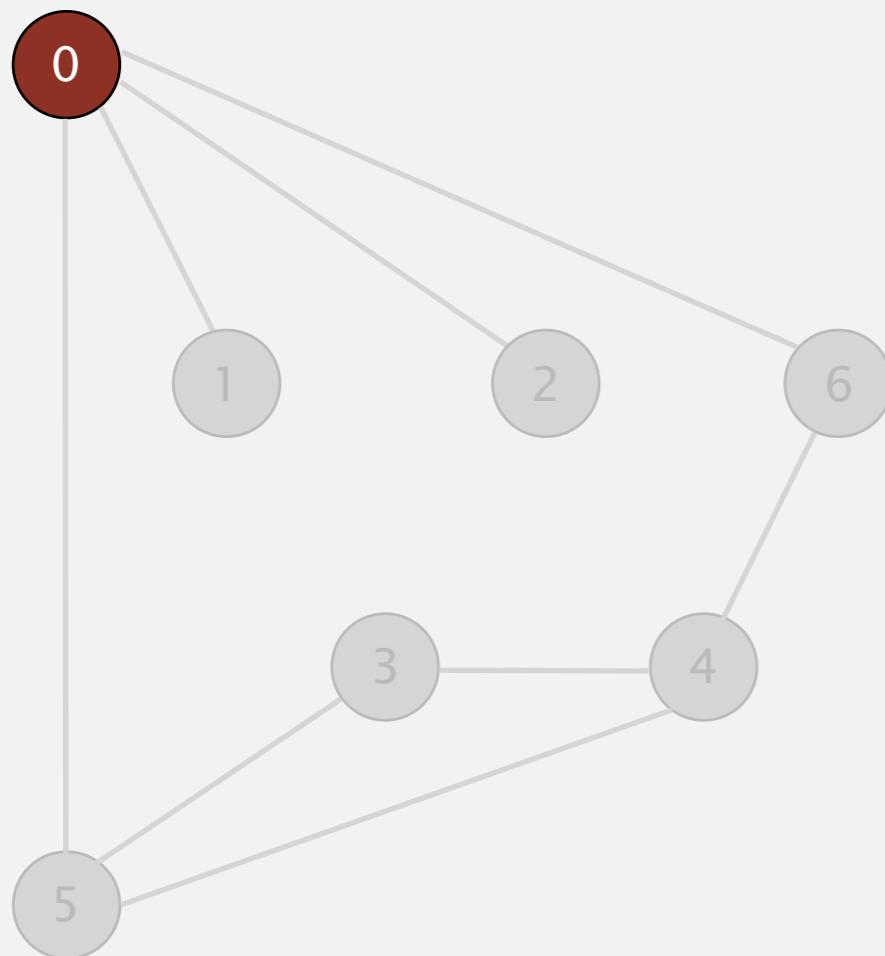


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

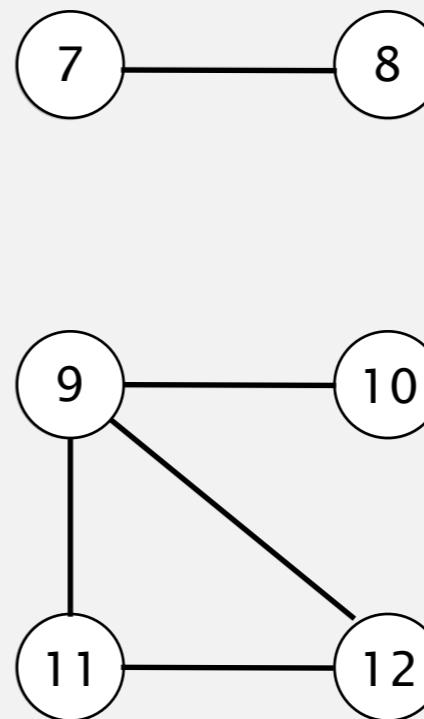
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



0 done

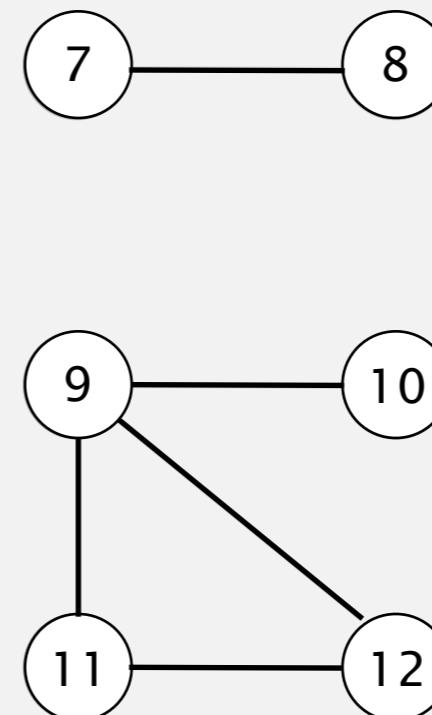
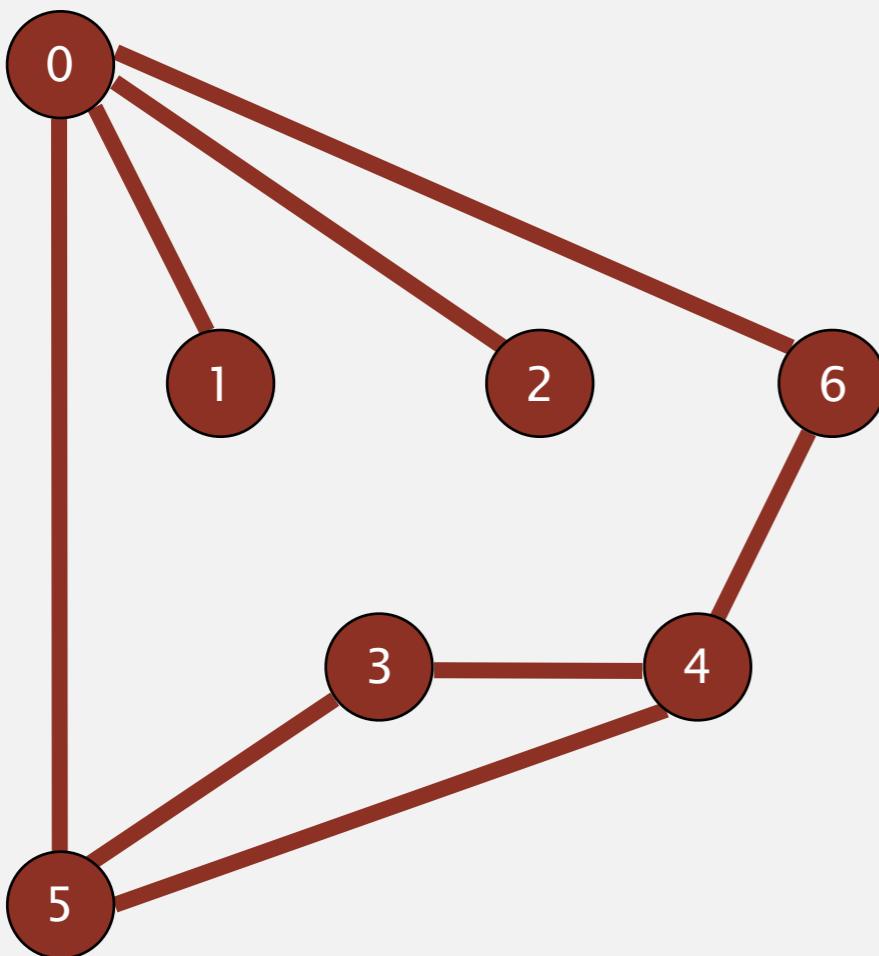


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



connected component

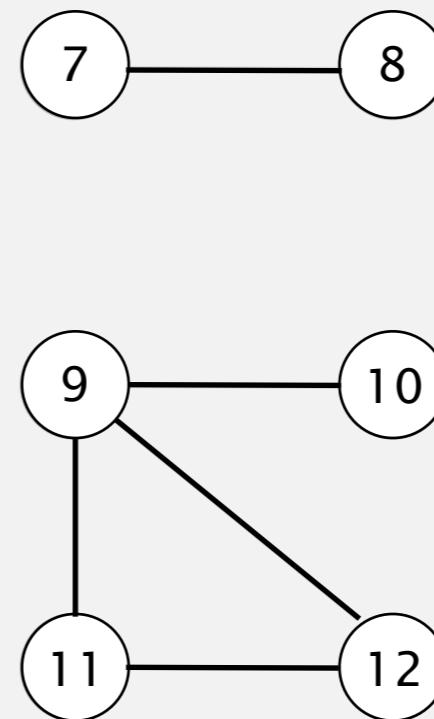
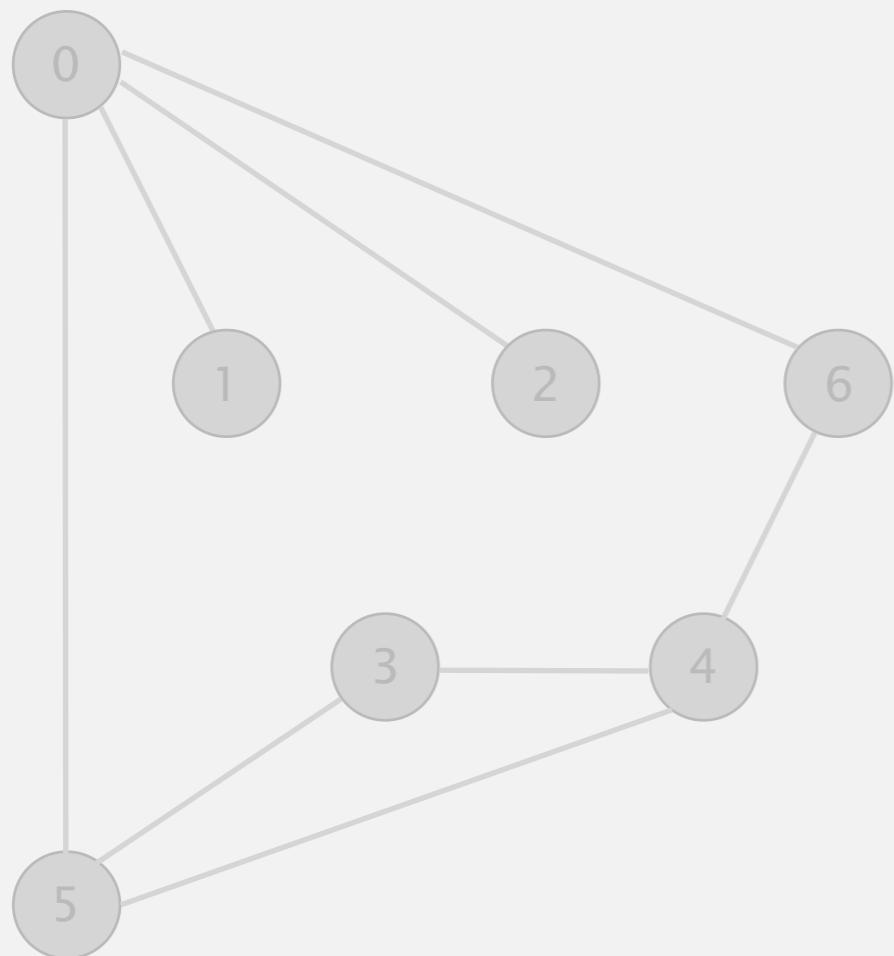
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

connected component: 0 1 2 3 4 5 6

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



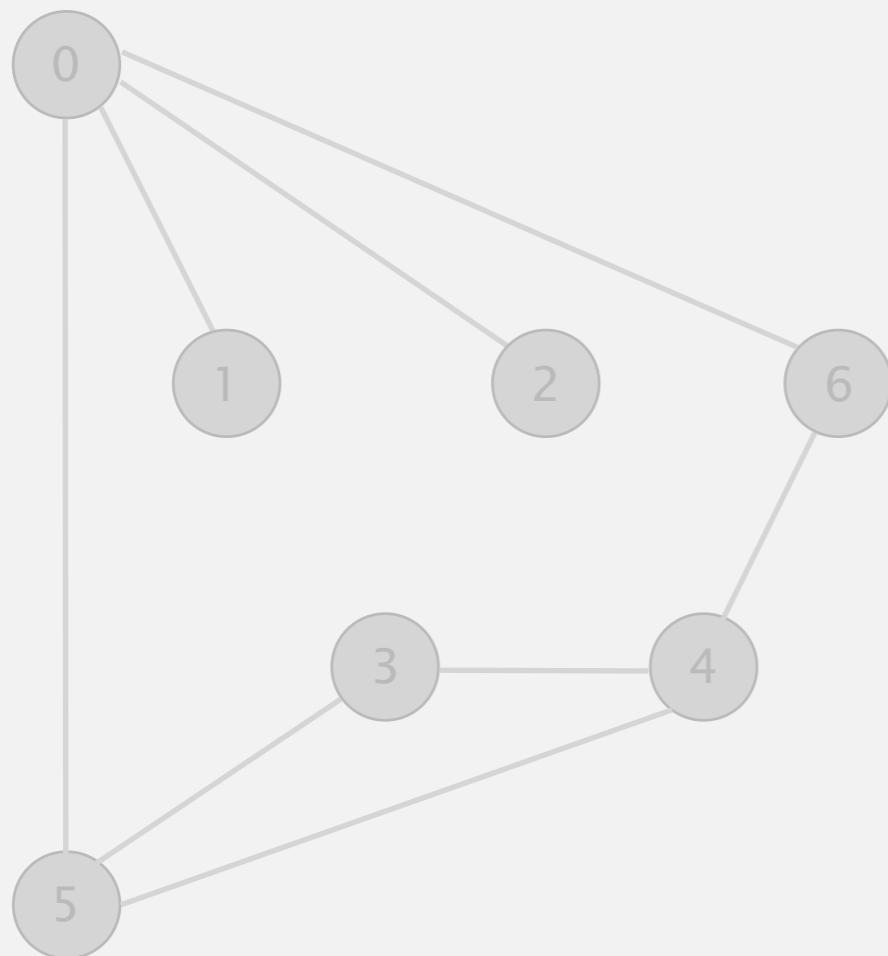
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

check 1 2 3 4 5 6

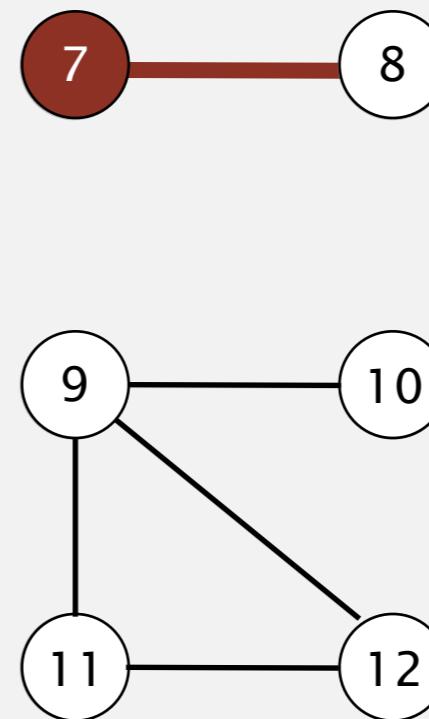
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 7

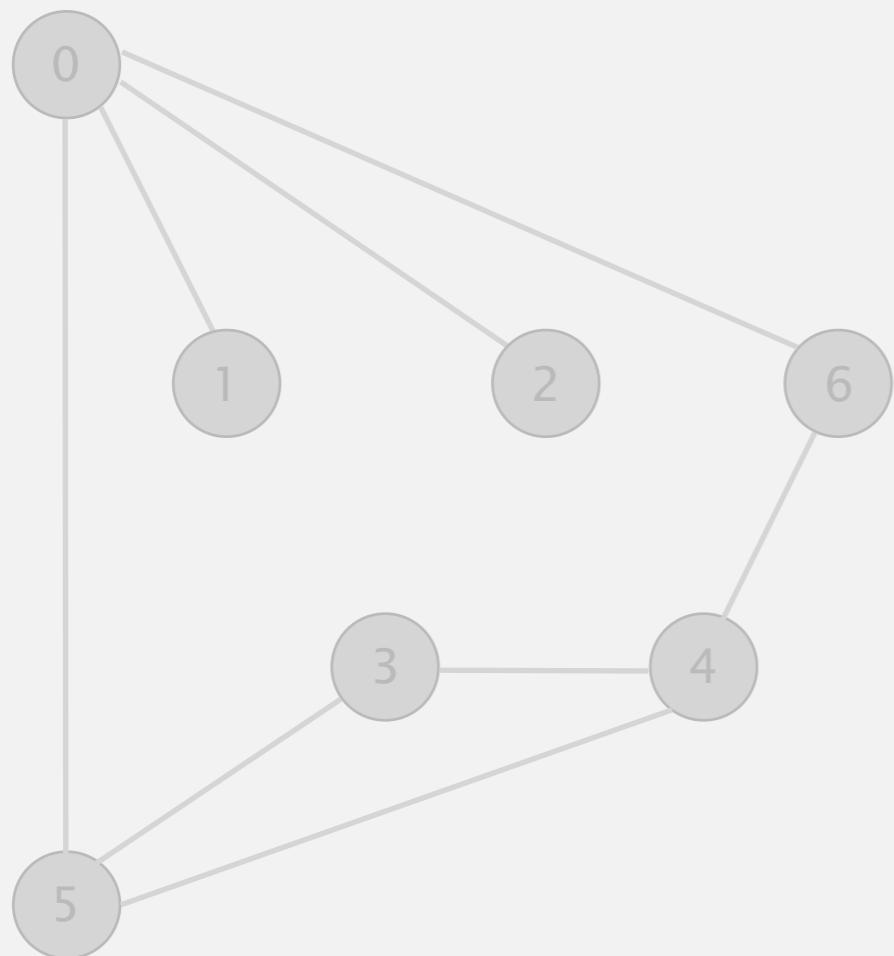


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

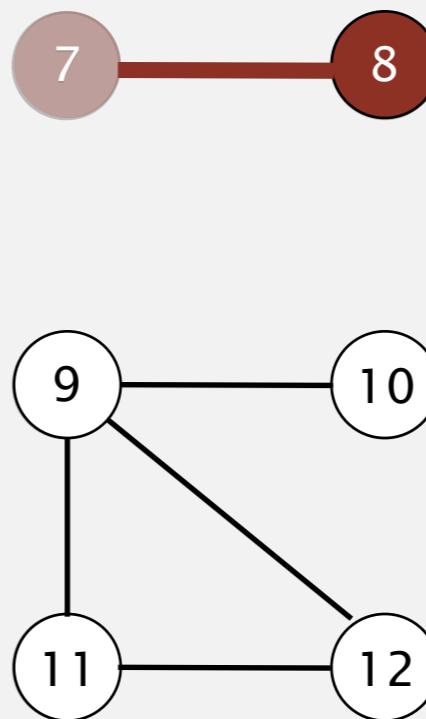
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



visit 8

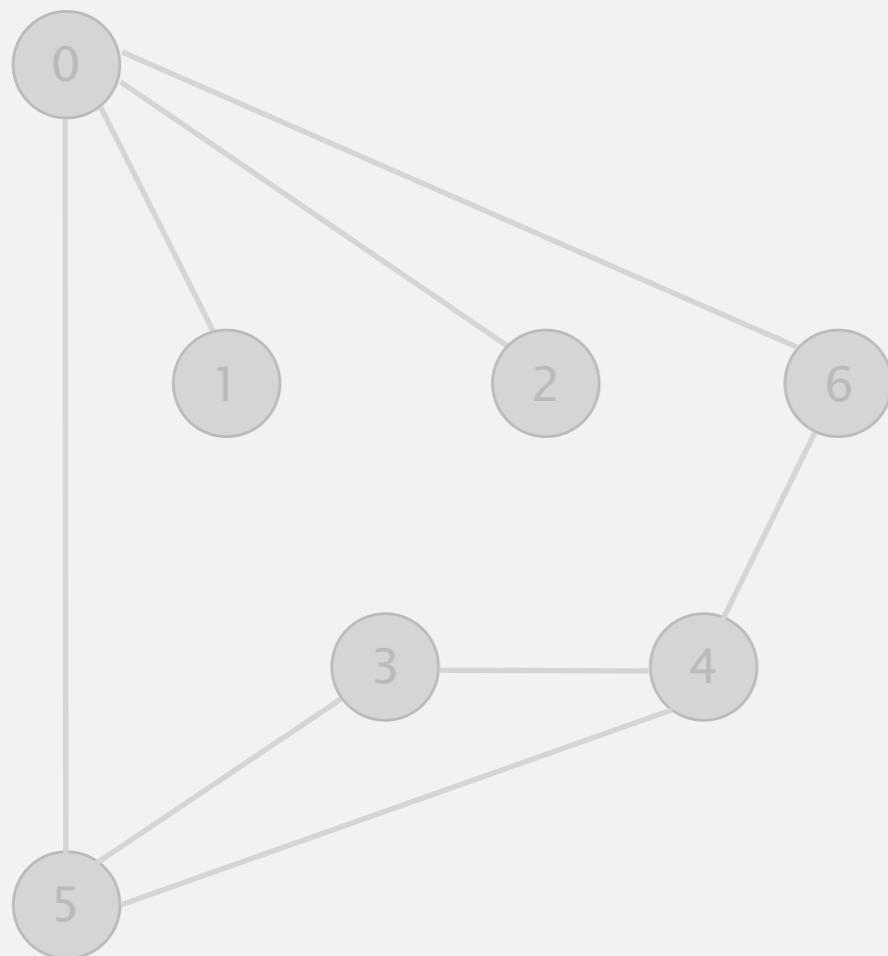


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	-
10	F	-
11	F	-
12	F	-

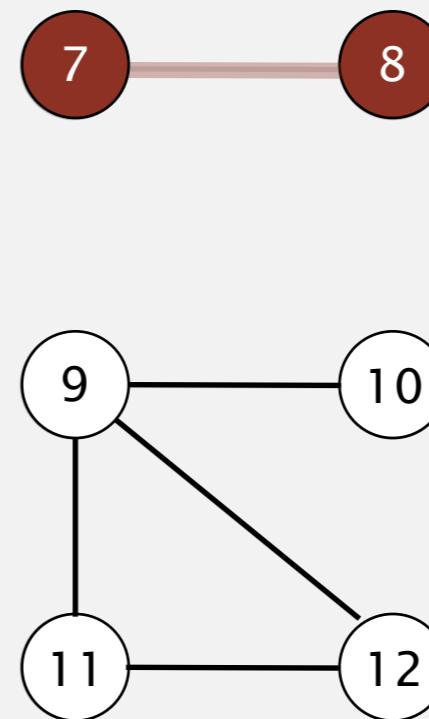
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



8 done

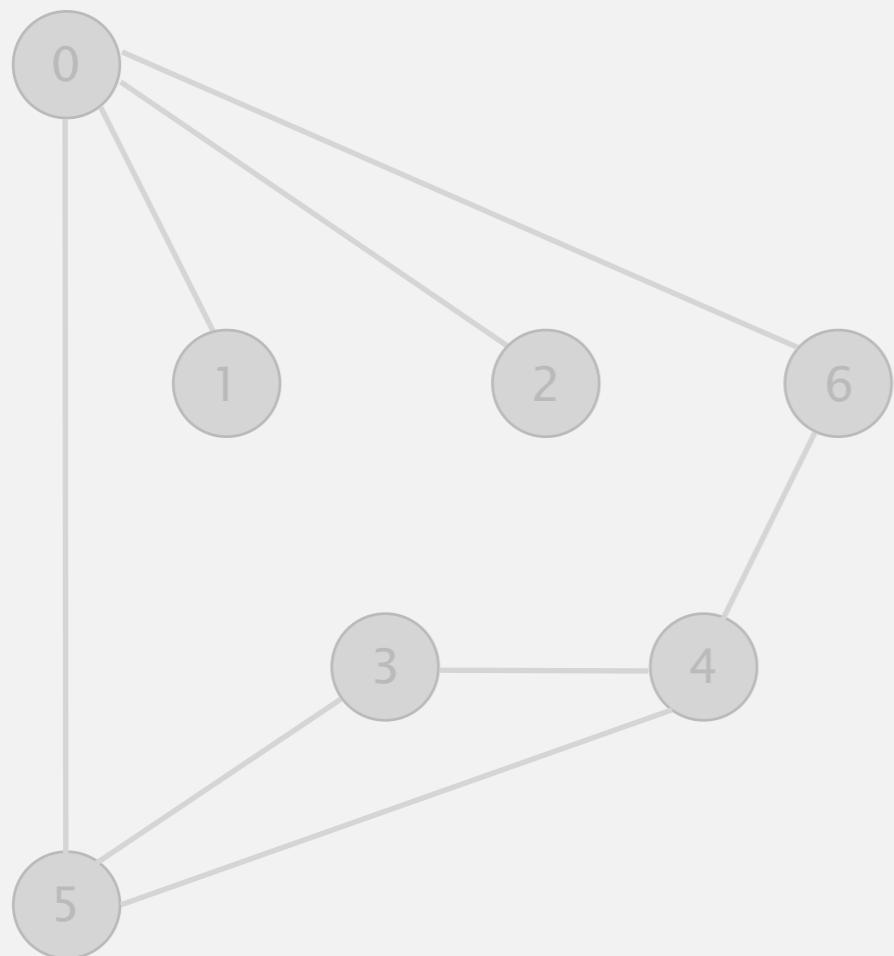


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	-
10	F	-
11	F	-
12	F	-

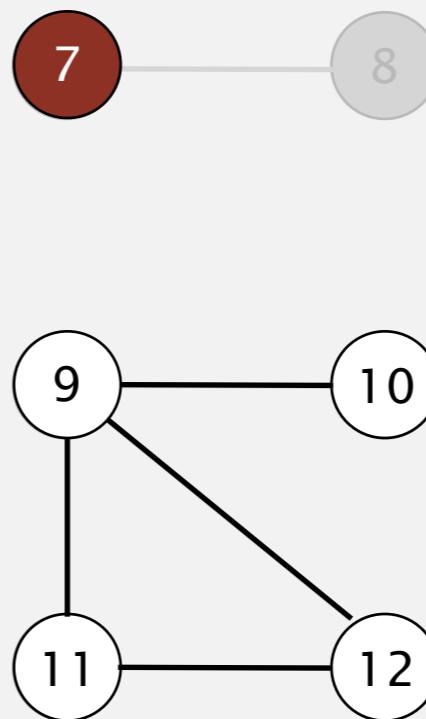
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



7 done

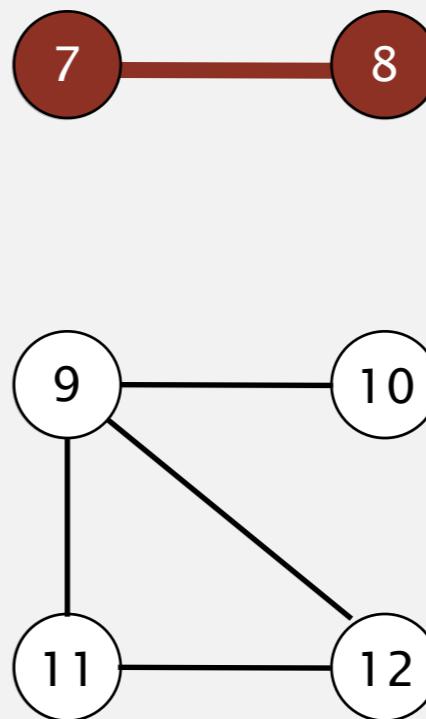
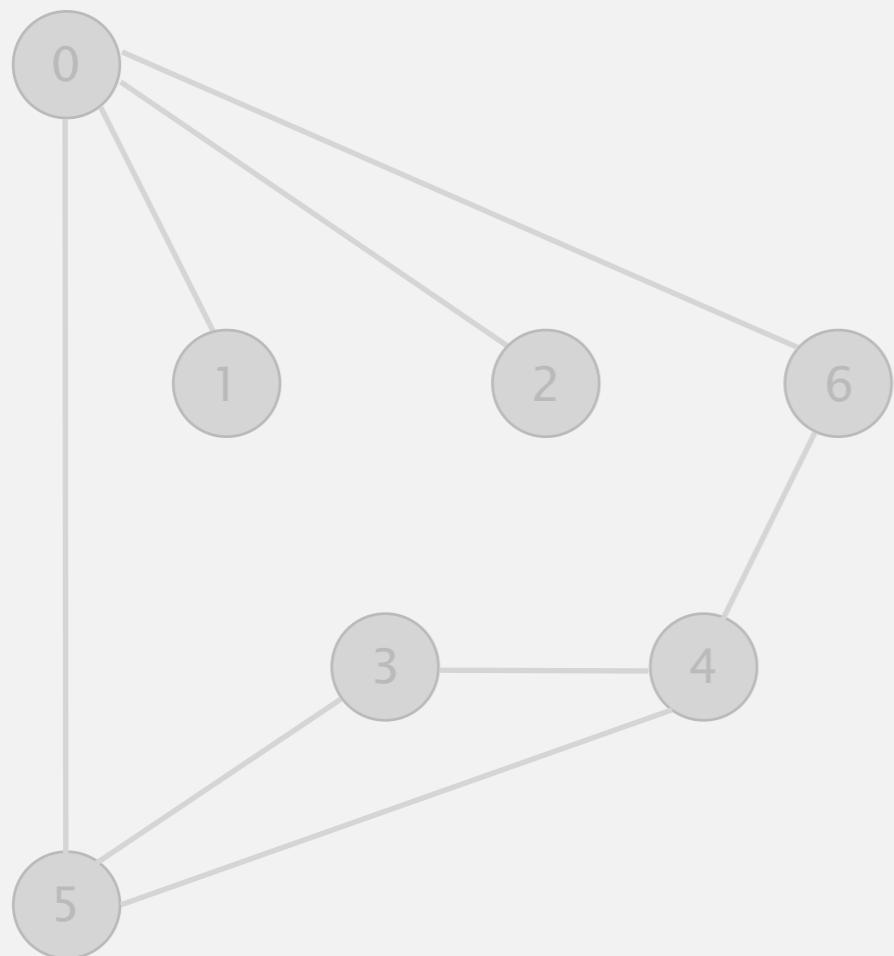


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	-
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



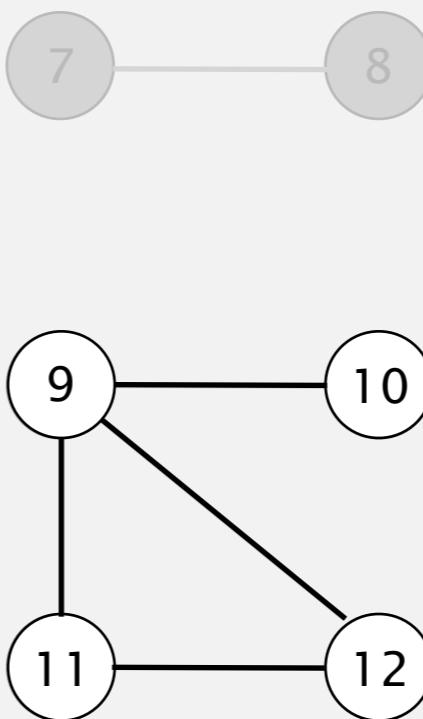
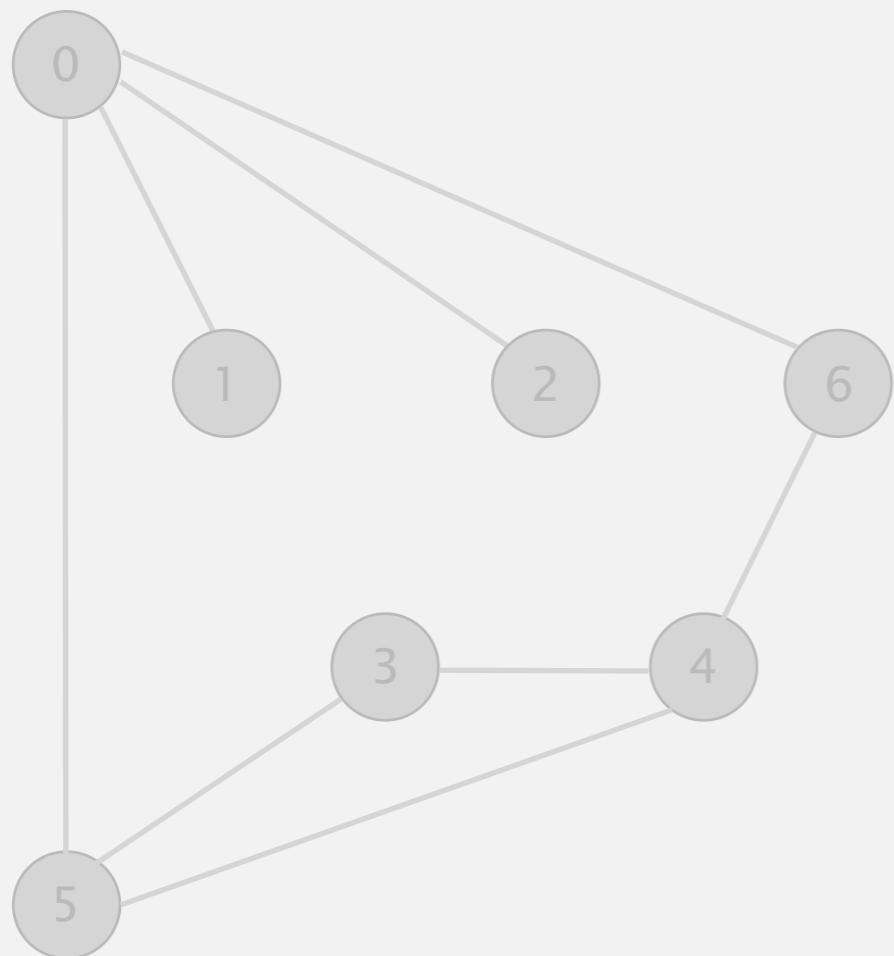
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	-
10	F	-
11	F	-
12	F	-

connected component: 7 8

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



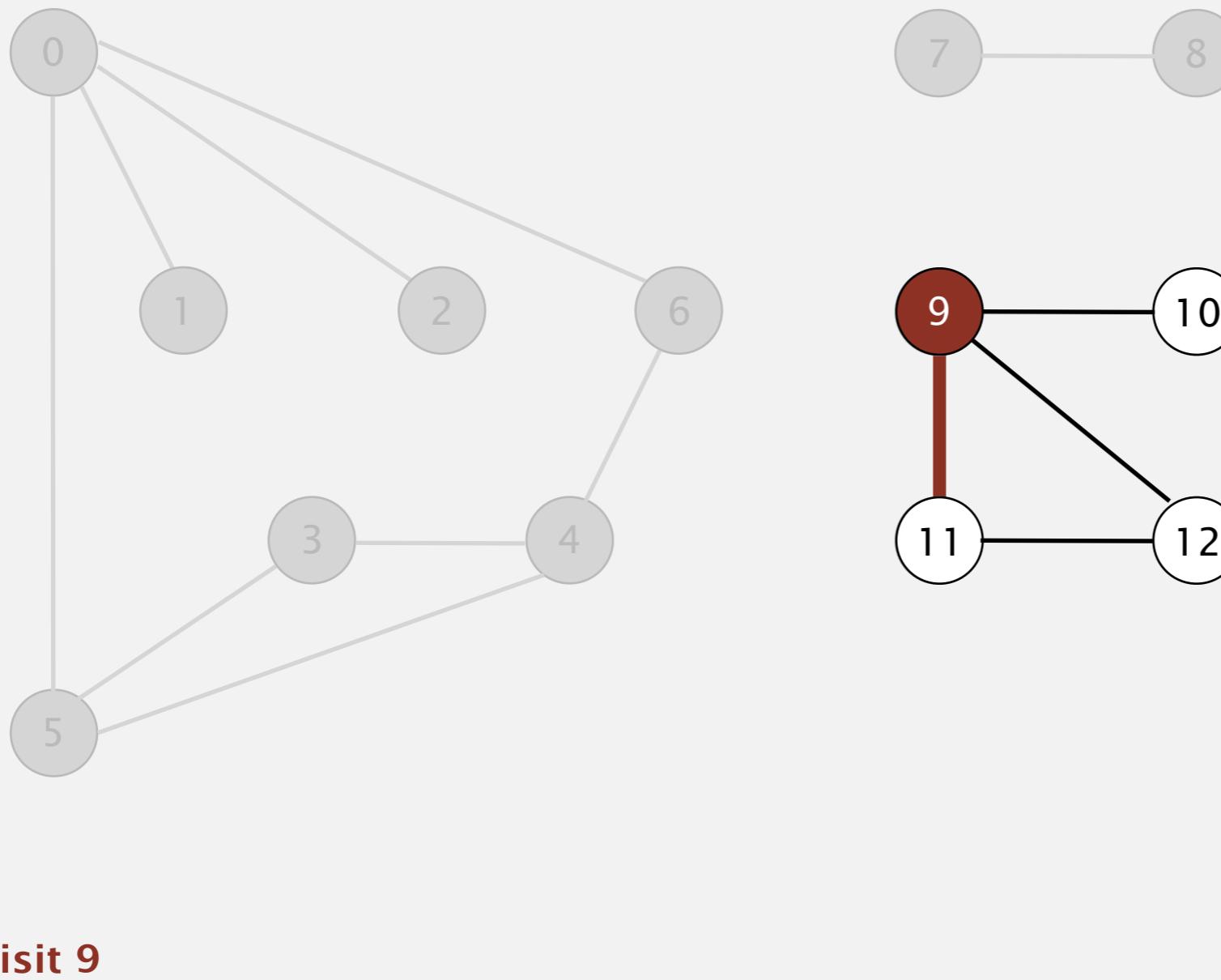
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	F	-
10	F	-
11	F	-
12	F	-

check 8

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

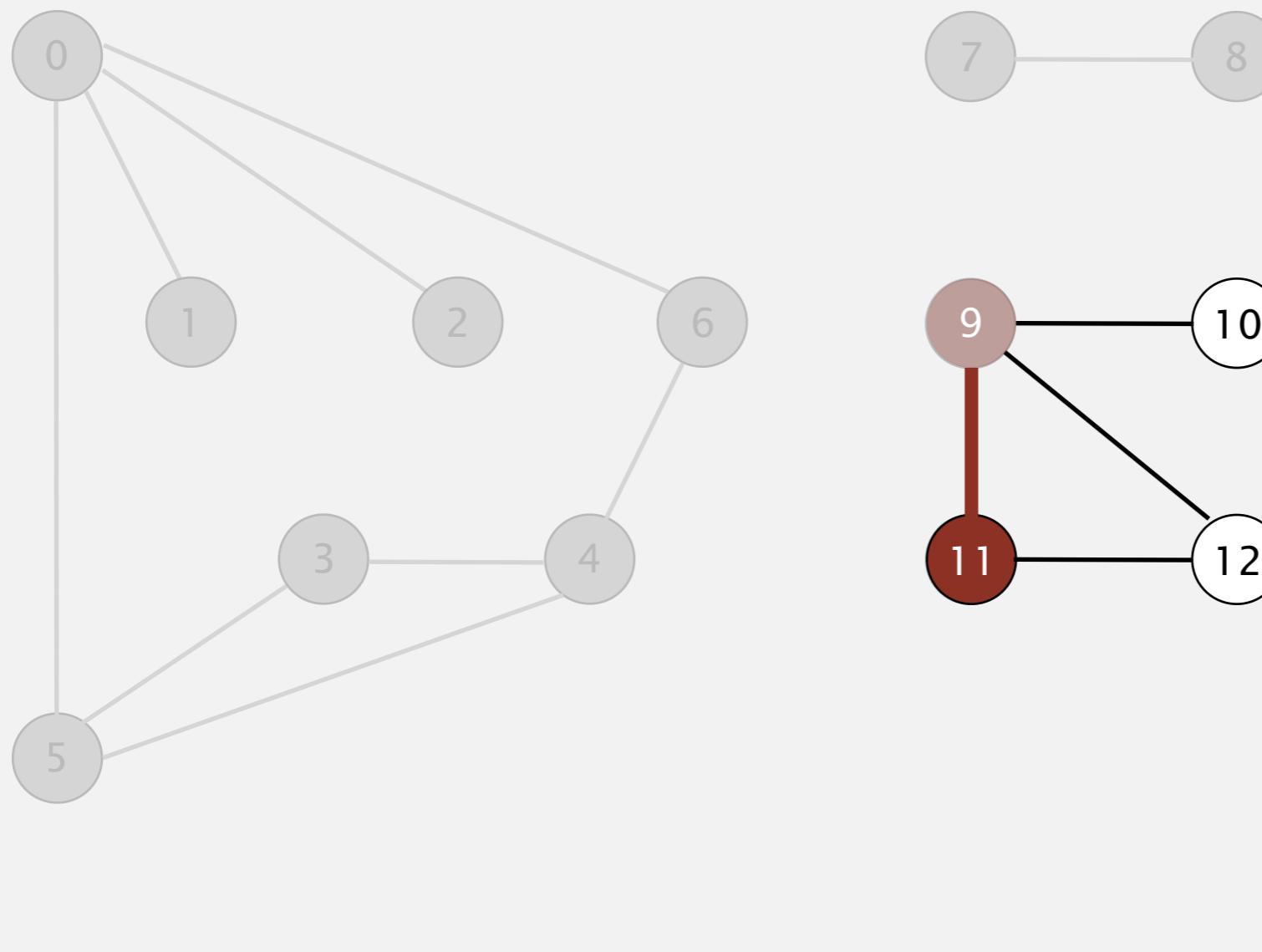


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	F	-
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

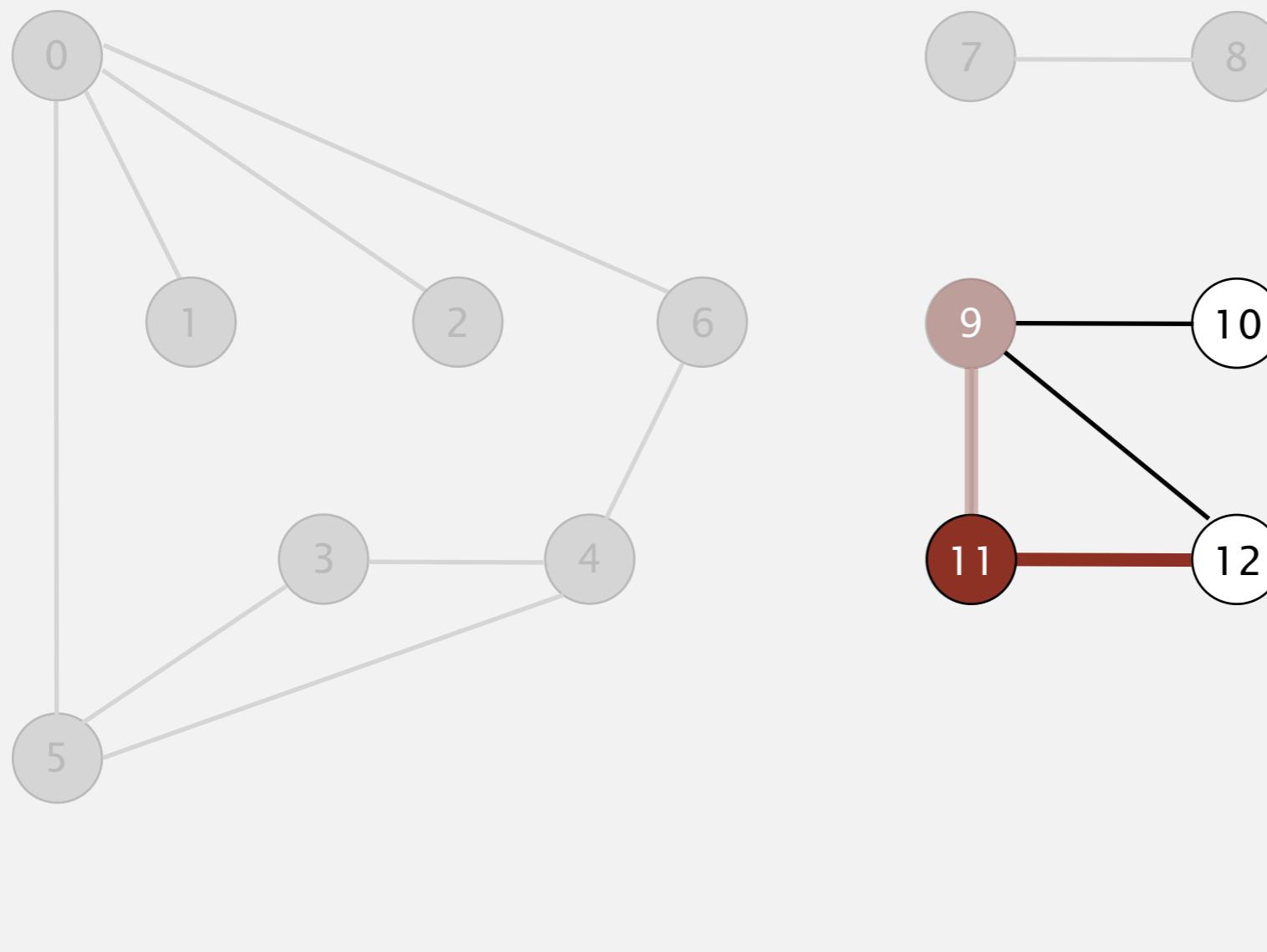


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

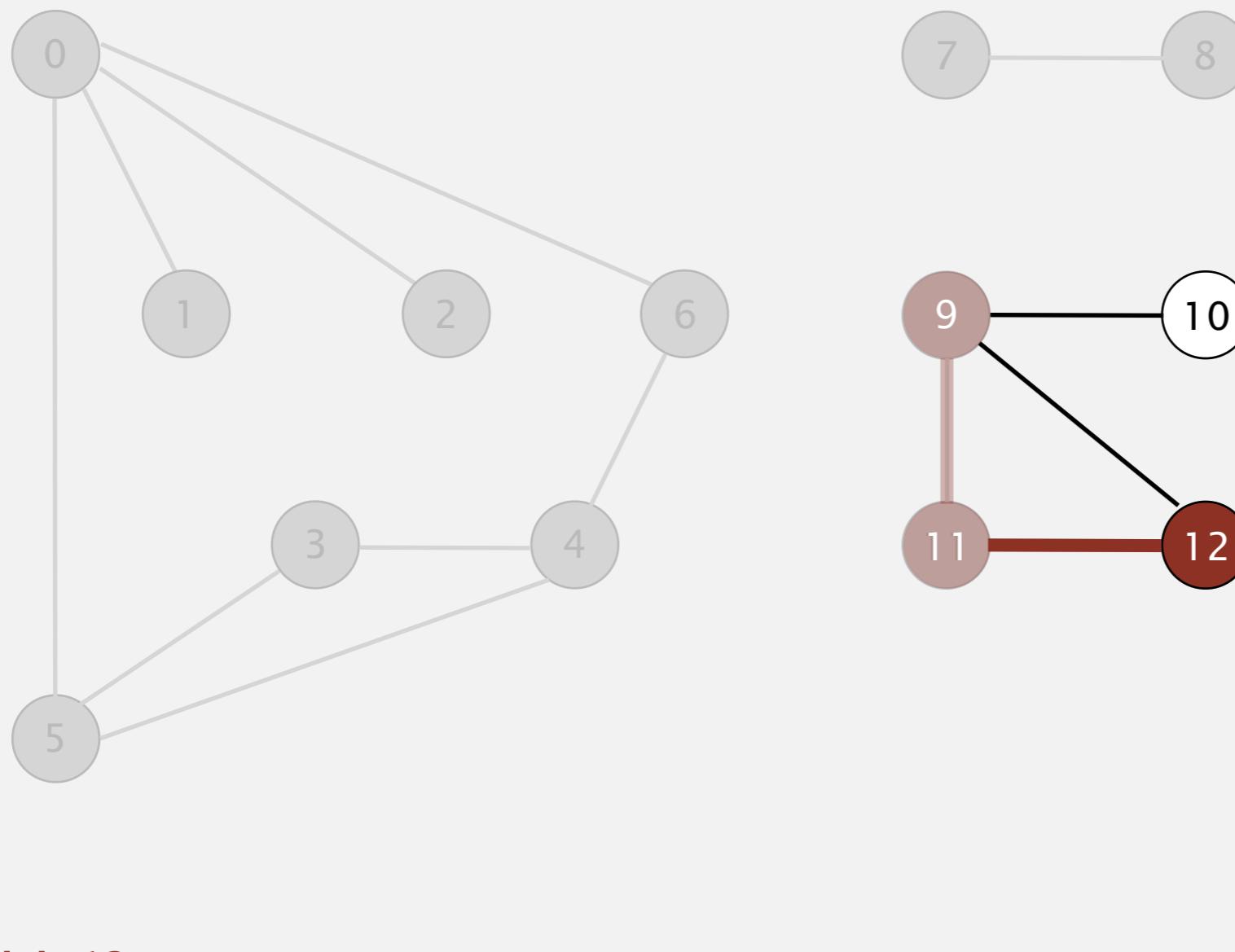


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	F	-

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

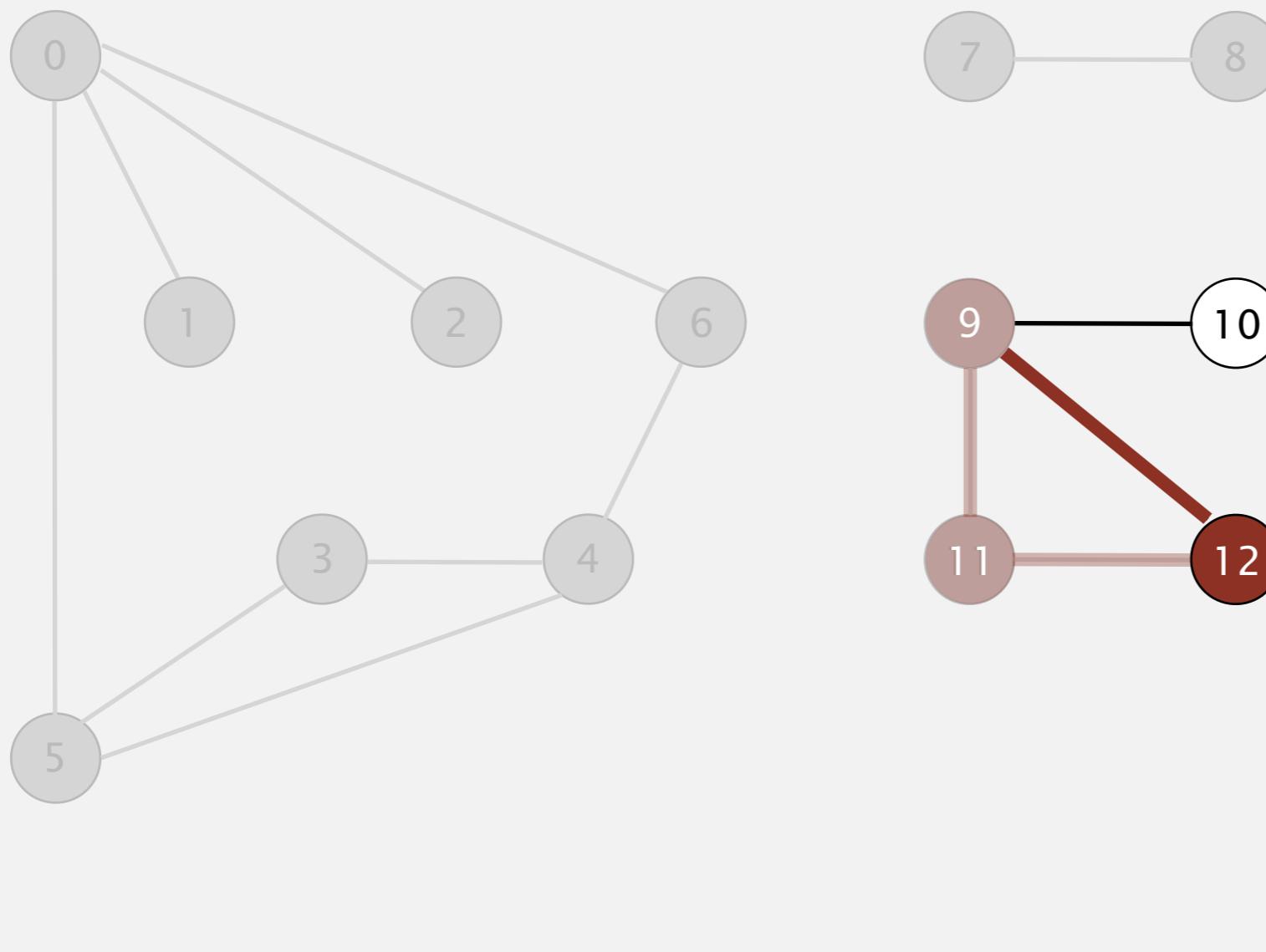


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

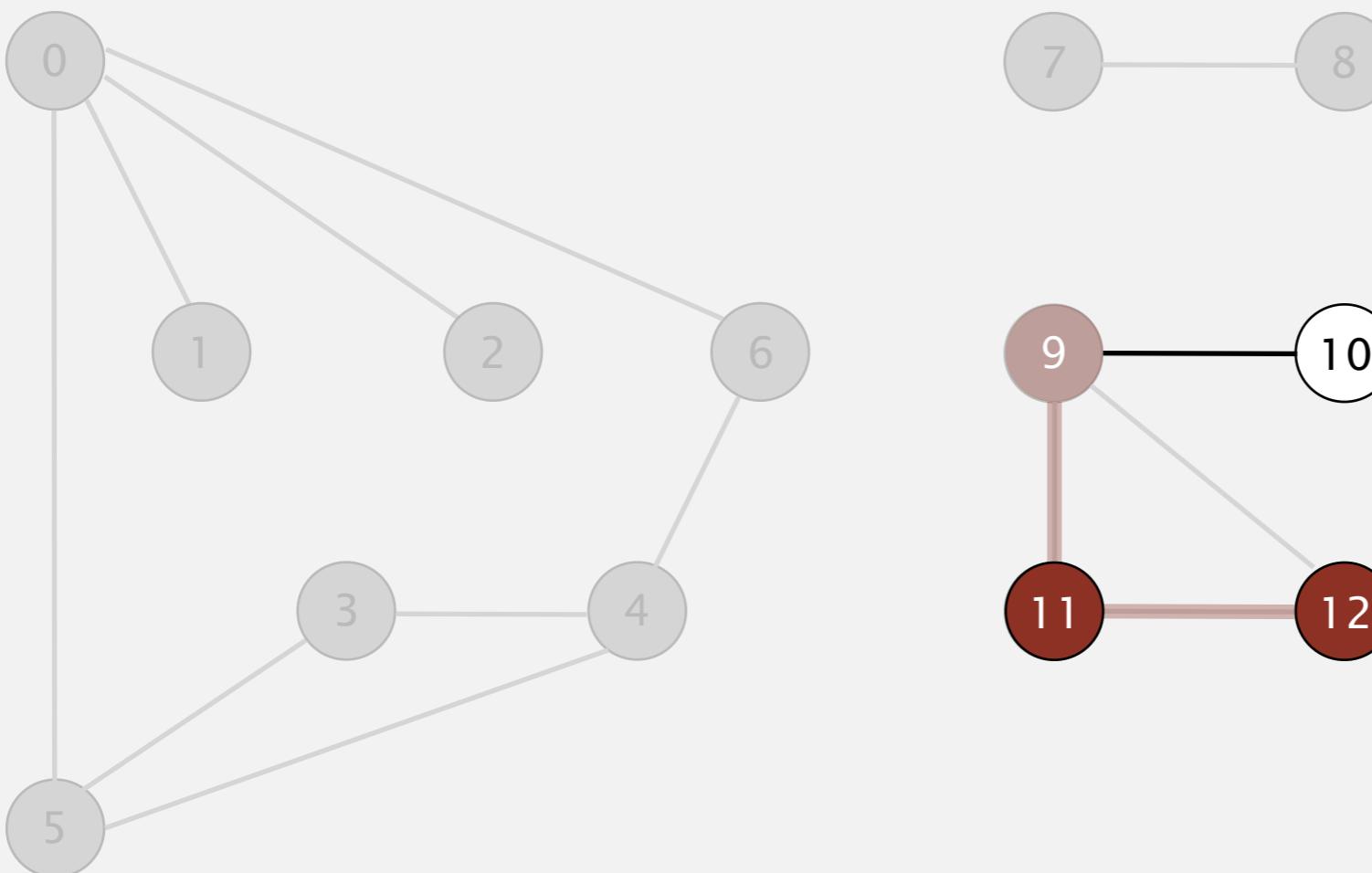


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



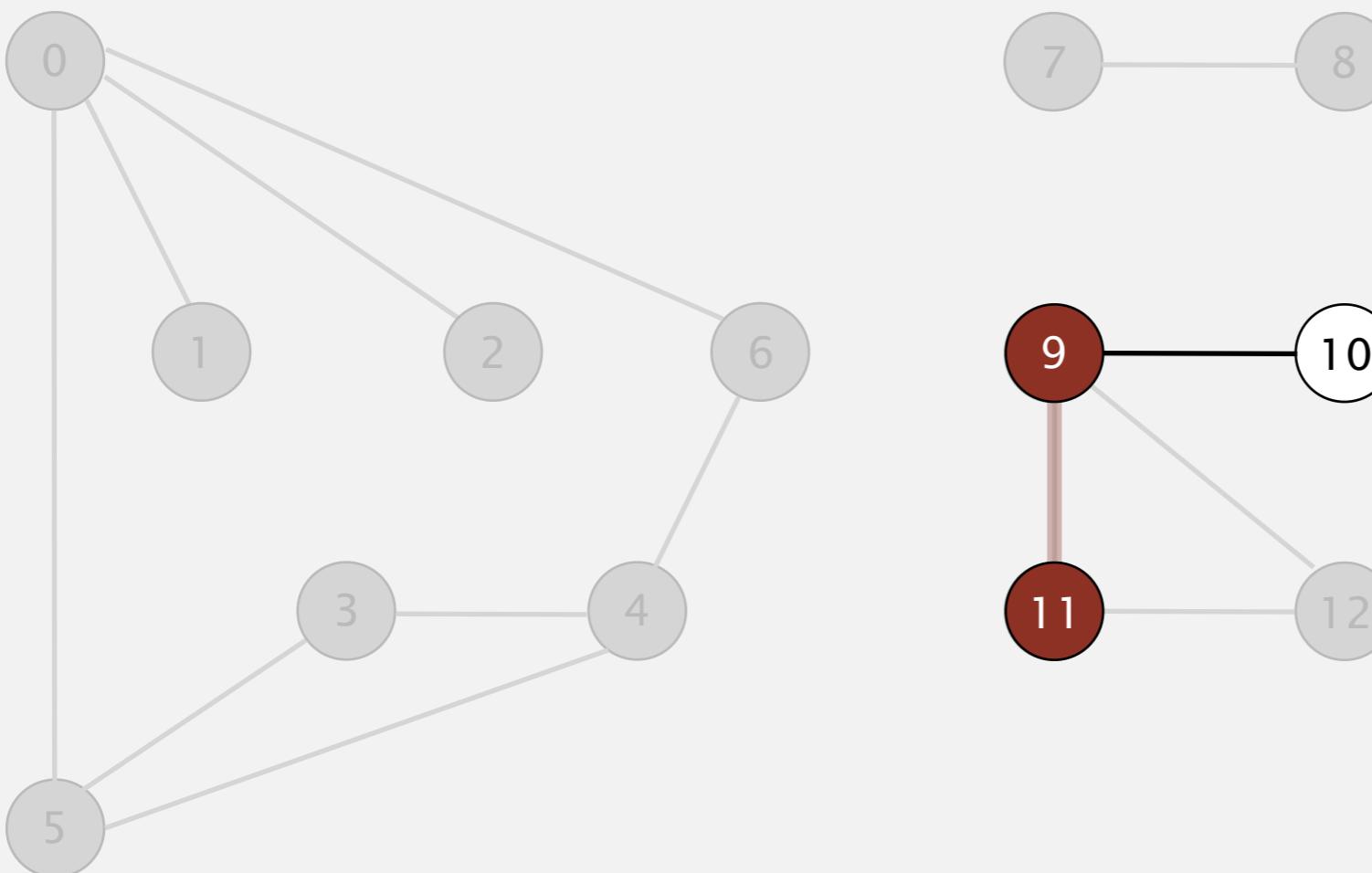
12 done

v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



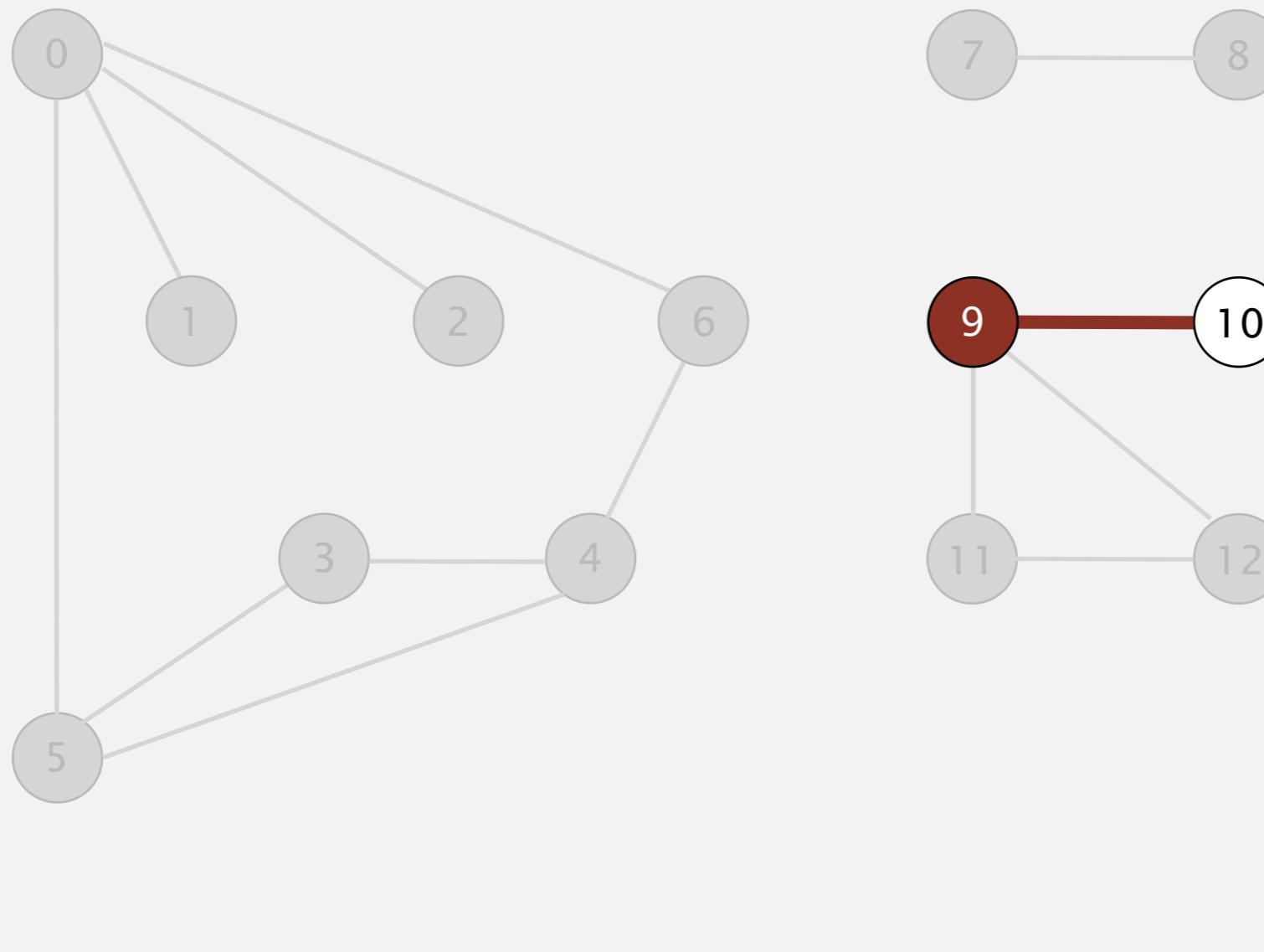
11 done

v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .

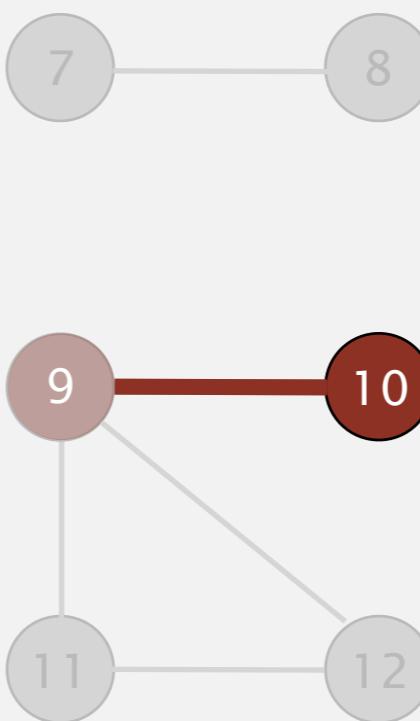
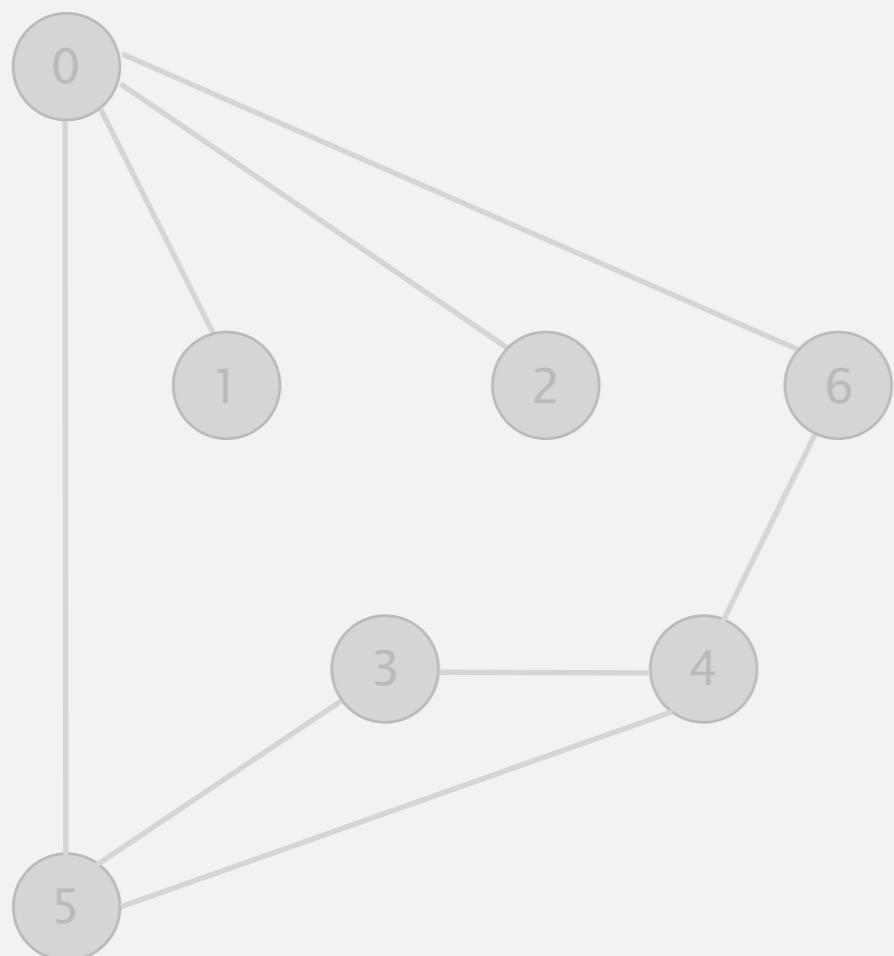


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	F	-
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



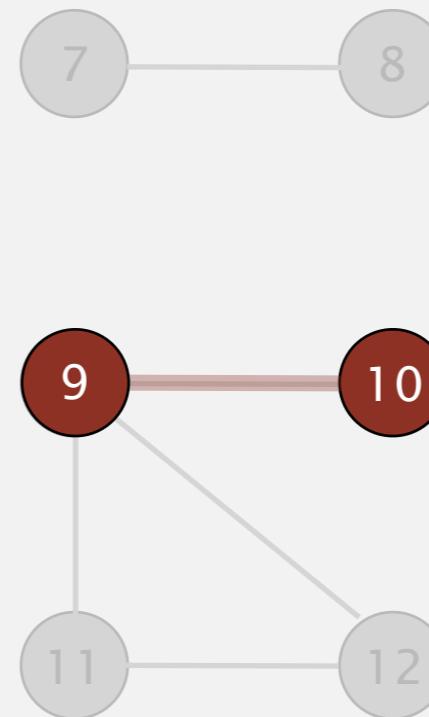
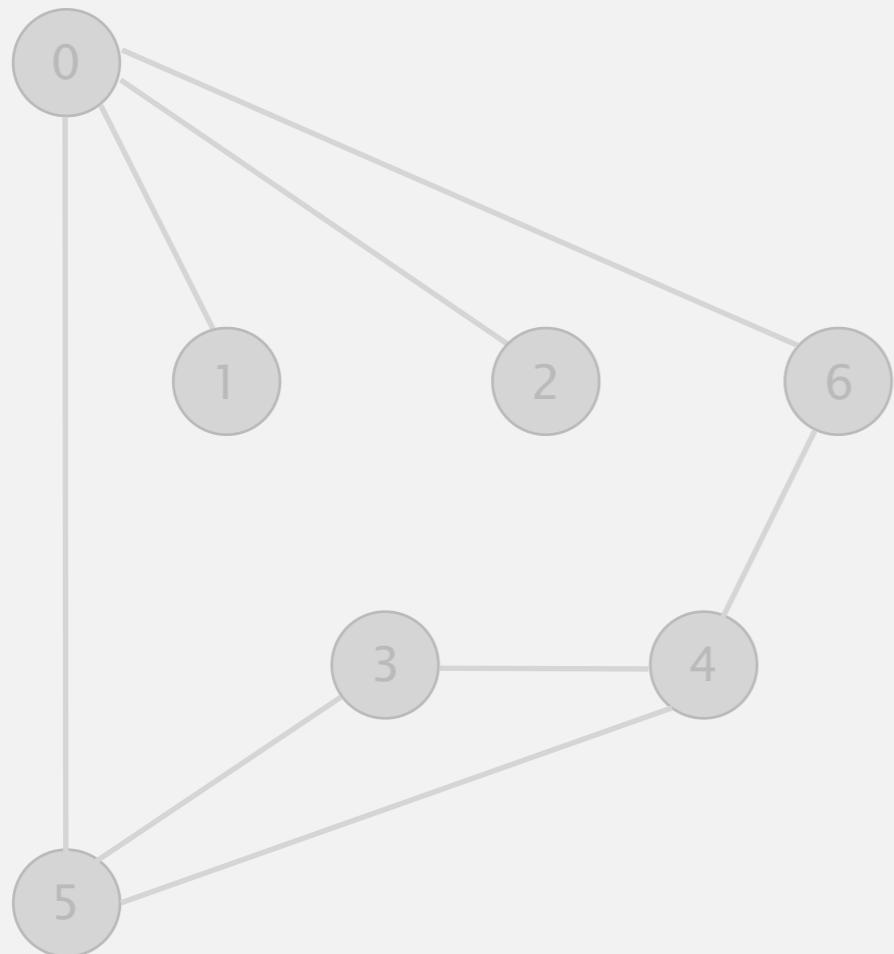
visit 10

v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



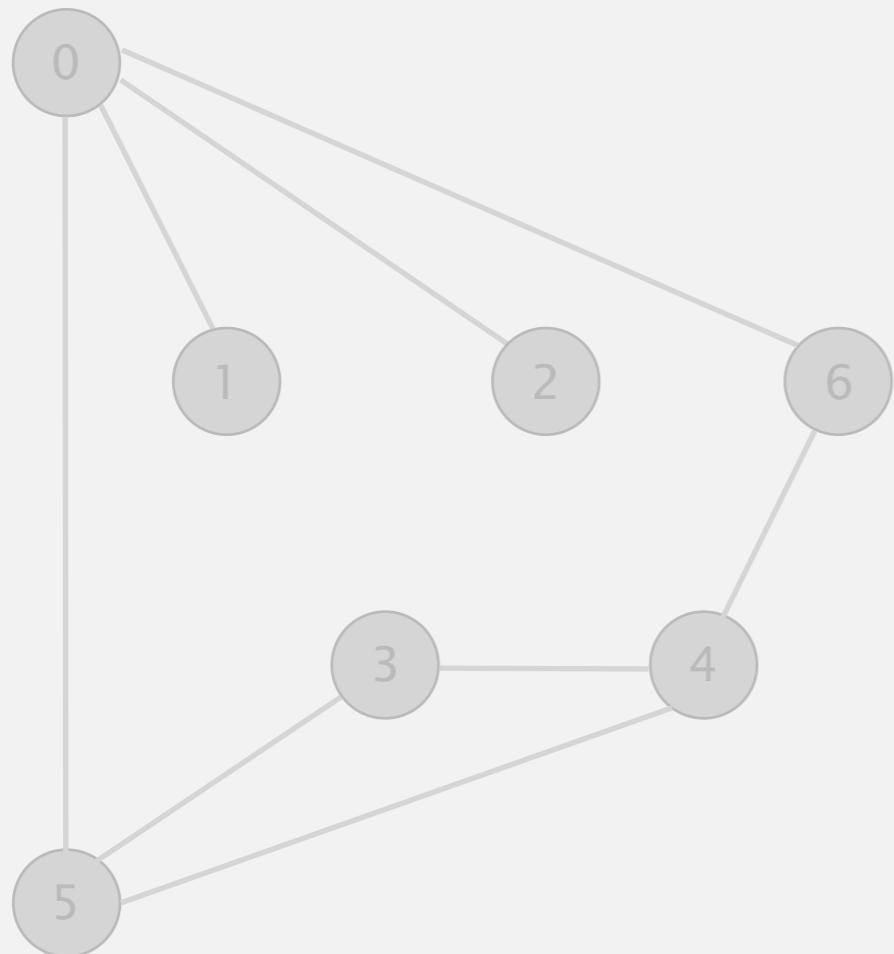
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

10 done

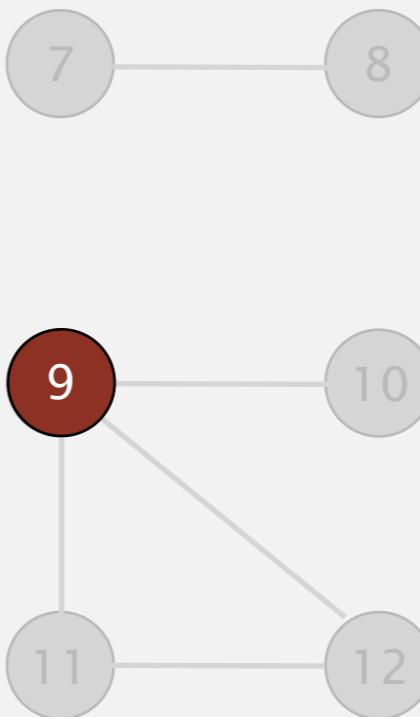
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



9 done

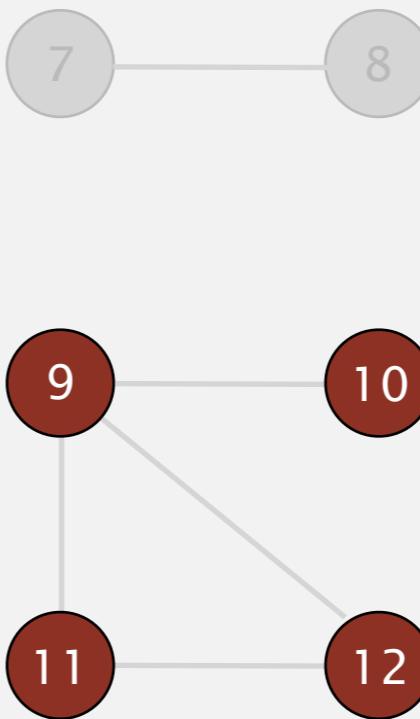
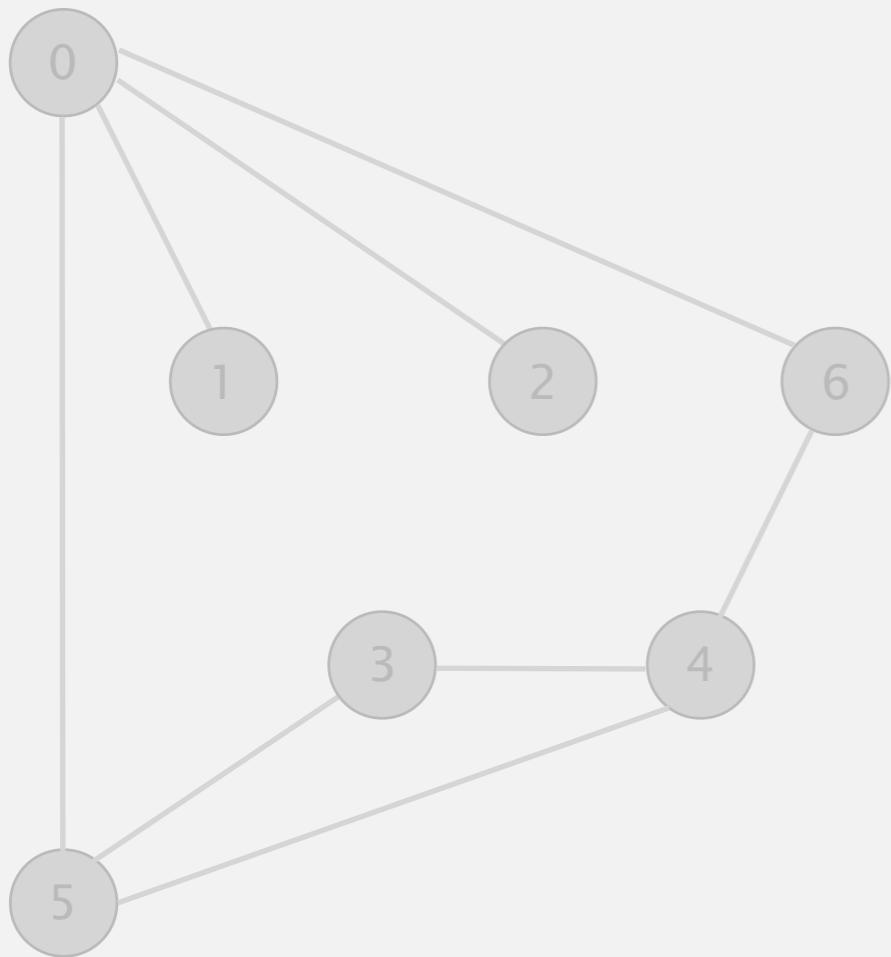


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



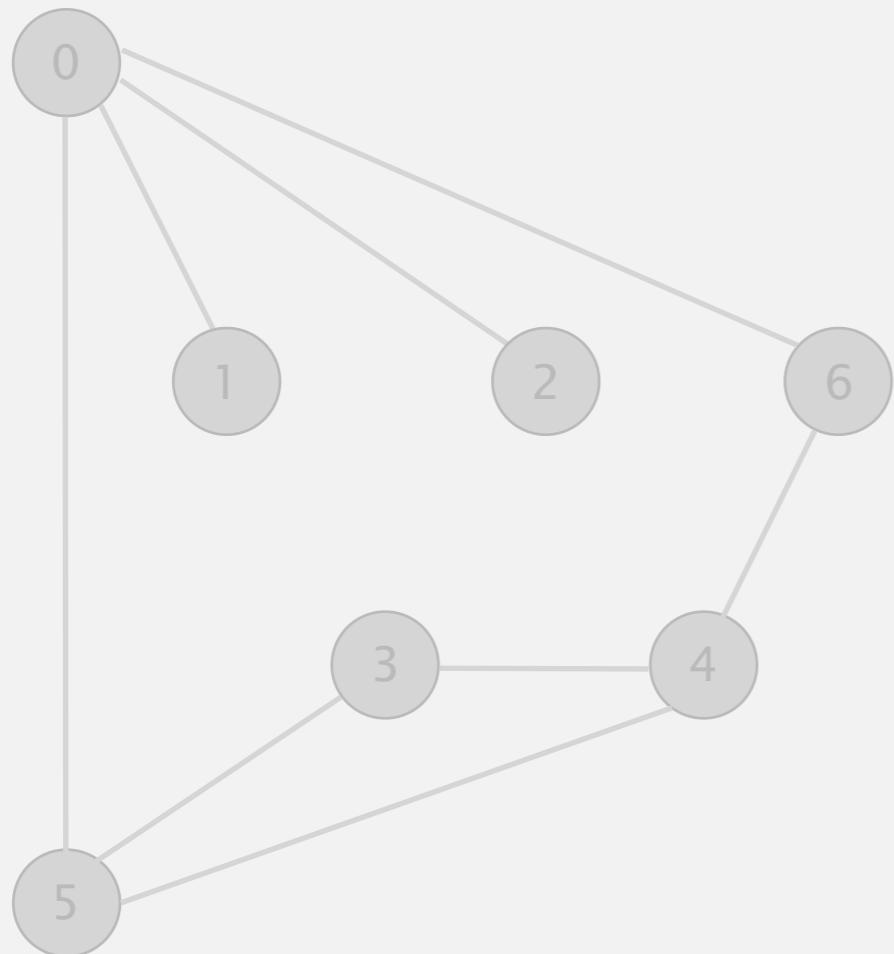
v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

connected component: 9 10 11 12

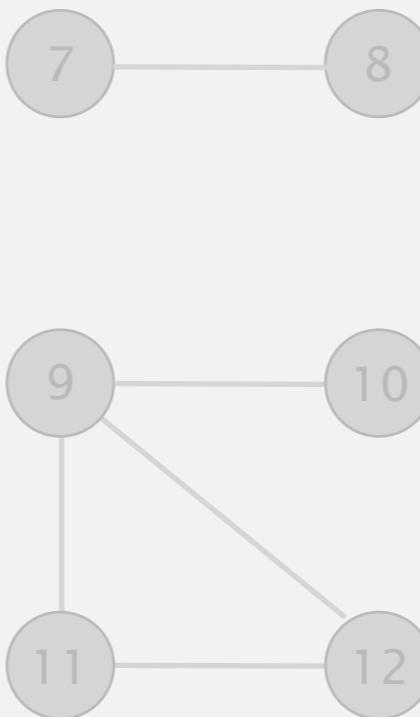
Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



check 10 11 12

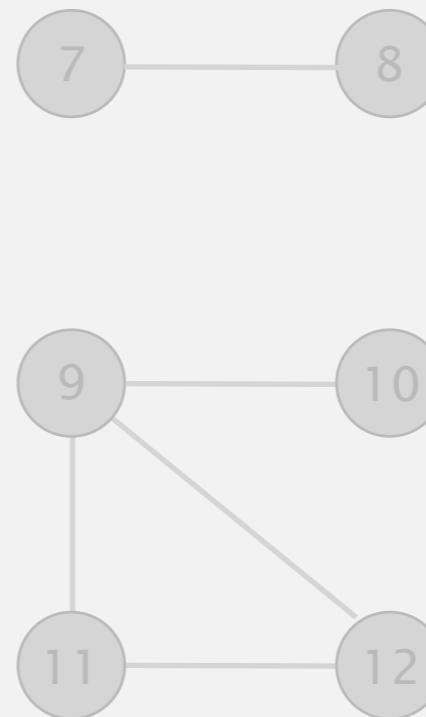
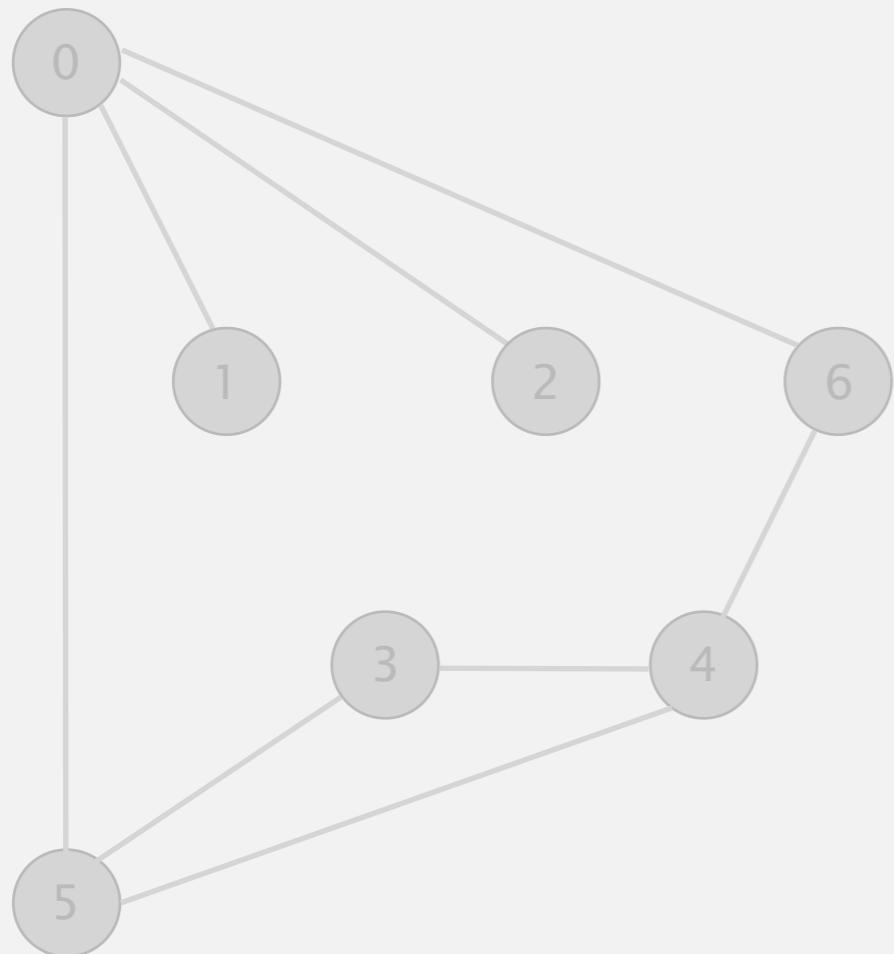


v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

Connected components

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



v	marked[]	cc[]
0	T	0
1	T	0
2	T	0
3	T	0
4	T	0
5	T	0
6	T	0
7	T	1
8	T	1
9	T	2
10	T	2
11	T	2
12	T	2

done

Finding connected components with DFS

```
public class CC
{
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    public int count()
    public int id(int v)
    private void dfs(Graph G, int v)

}
```

$\text{id}[v] = \text{id of component containing } v$
number of components

run DFS from one vertex in
each component

see next slide

Finding connected components with DFS (continued)

```
public int count()
{   return count; }

public int id(int v)
{   return id[v]; }

private void dfs(Graph G, int v)
{
    marked[v] = true;
    id[v] = count;
    for (int w : G.adj(v))
        if (!marked[w])
            dfs(G, w);
}
```

number of components

id of component containing v

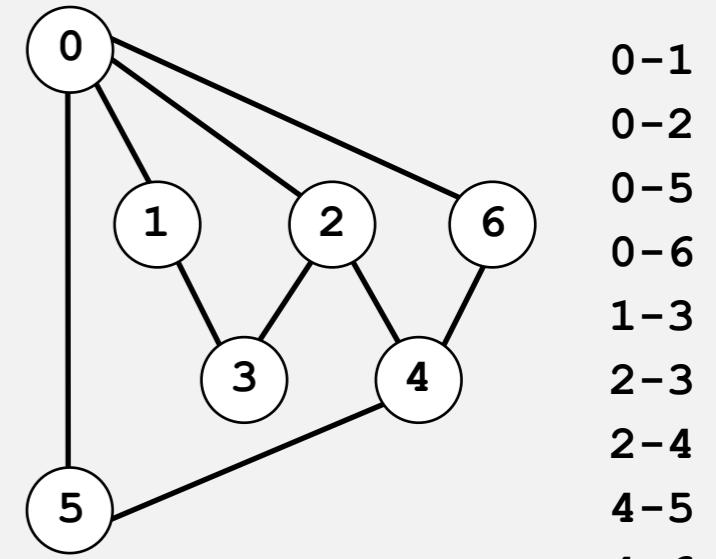
all vertices discovered in
same call of dfs have same id

UNDIRECTED GRAPHS

- ▶ Graph API
- ▶ Depth-first search
- ▶ Breadth-first search
- ▶ Connected components
- ▶ Challenges

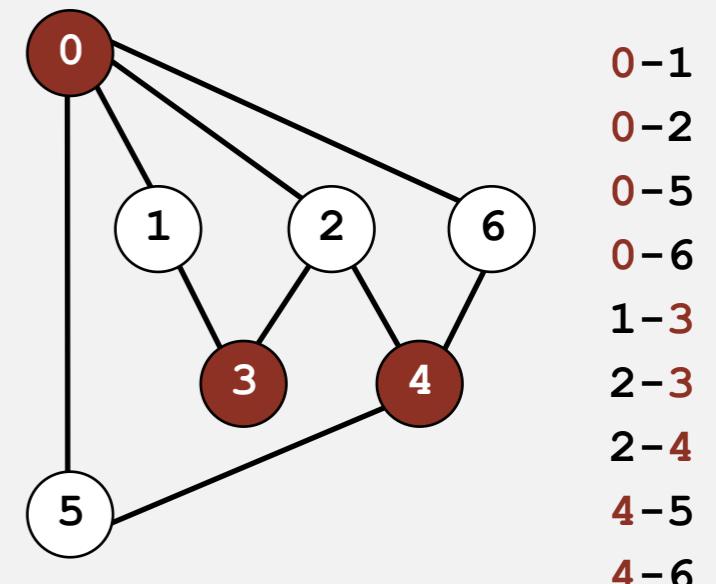
Graph-processing challenge I

Problem. Is a graph bipartite?



How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



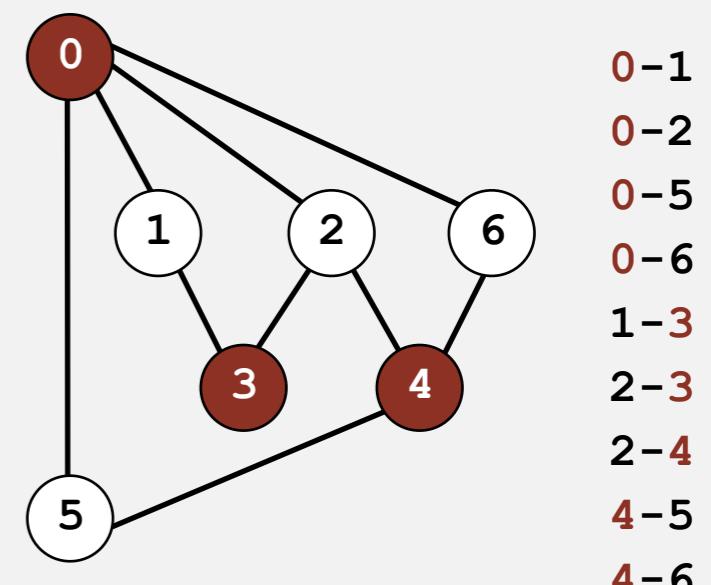
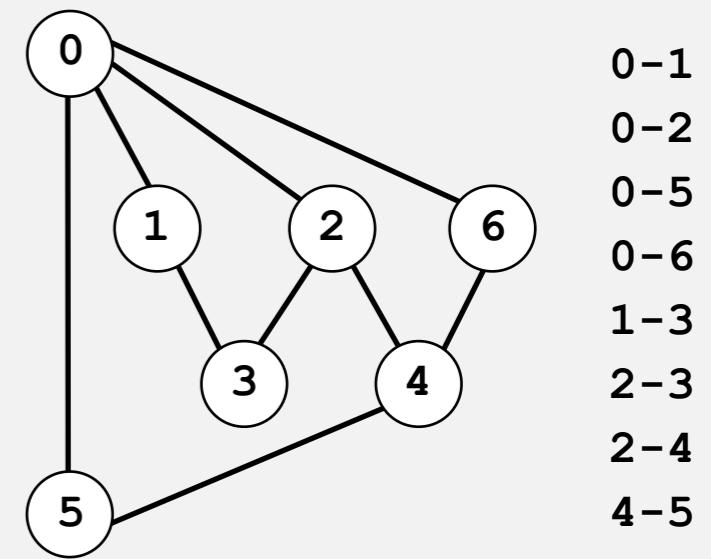
Graph-processing challenge I

Problem. Is a graph bipartite?

How difficult?

- Any programmer could do it.
- ✓ • Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

simple DFS-based solution
(see textbook)

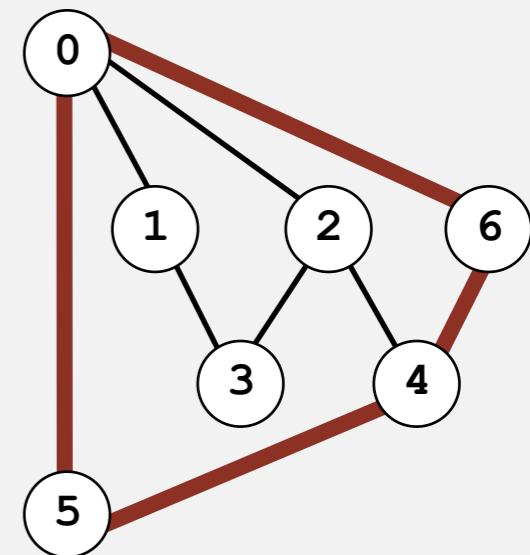
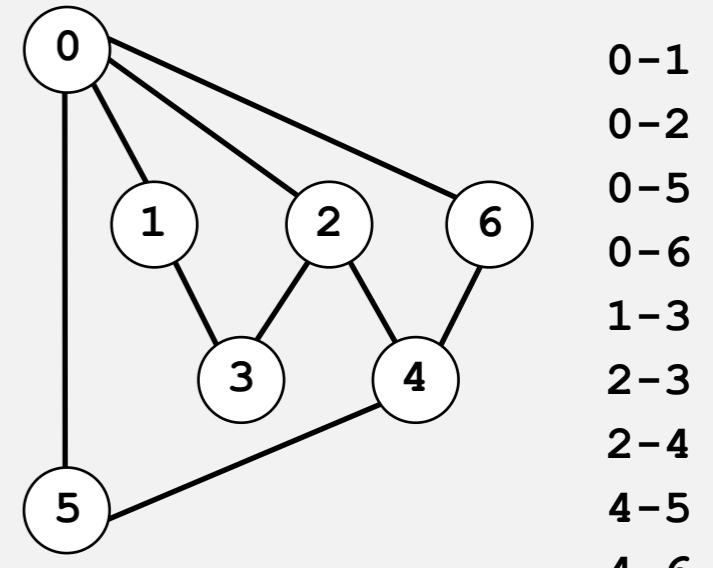


Graph-processing challenge 2

Problem. Find a cycle.

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



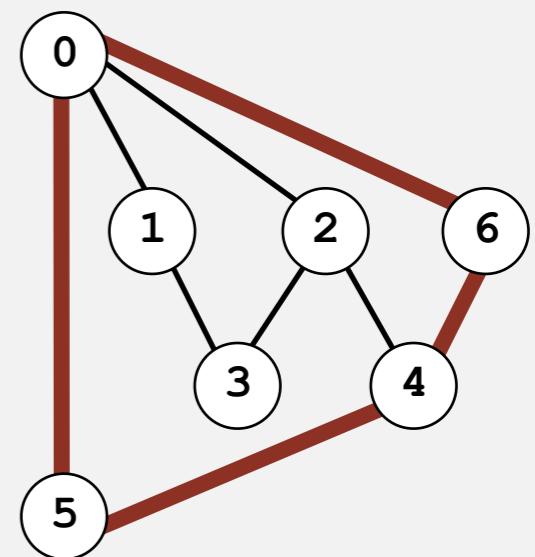
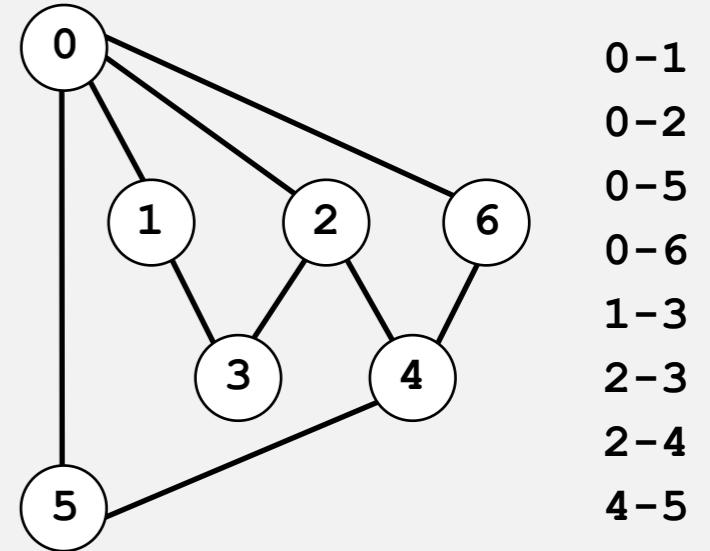
Graph-processing challenge 2

Problem. Find a cycle.

How difficult?

- Any programmer could do it.
- ✓ • Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

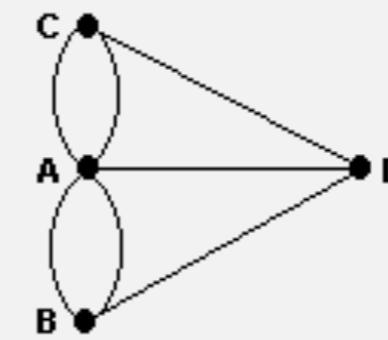
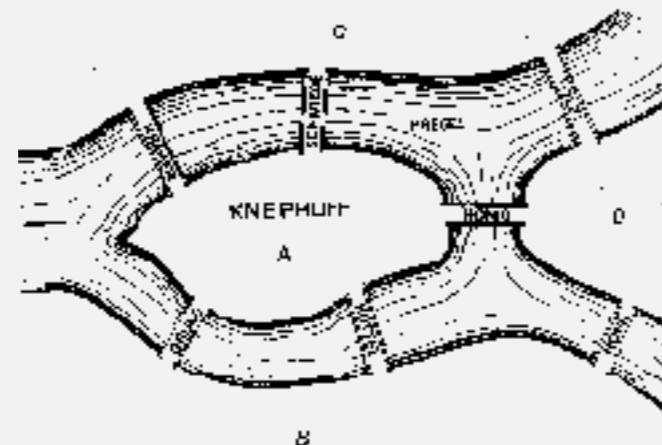
simple DFS-based solution
(see textbook)



Bridges of Königsberg

The Seven Bridges of Königsberg. [Leonhard Euler 1736]

“ ... in Königsberg in Prussia, there is an island A, called the Kneiphof; the river which surrounds it is divided into two branches ... and these branches are crossed by seven bridges. Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he could cross each bridge once and only once. ”



Euler tour. Is there a (general) cycle that uses each edge exactly once?

Answer. Yes iff connected and all vertices have **even** degree.

To find path. DFS-based algorithm (see textbook).

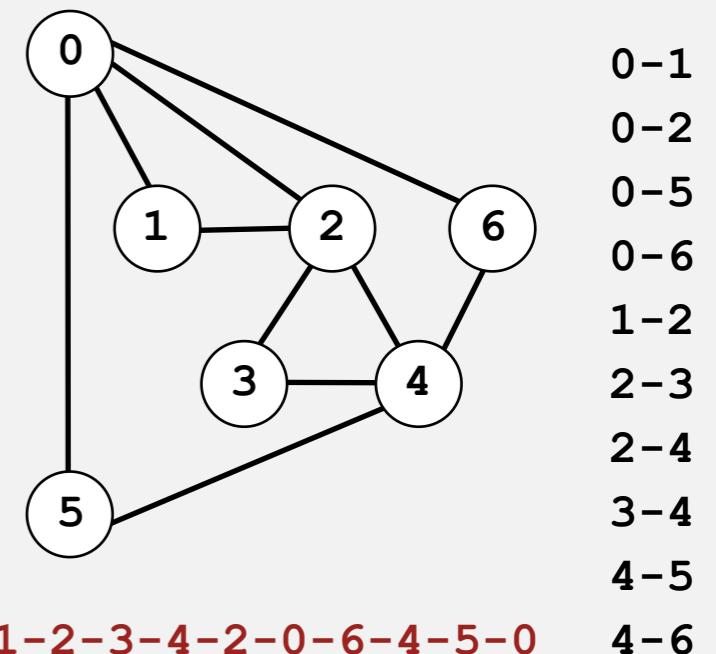
Graph-processing challenge 3

Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



Graph-processing challenge 3

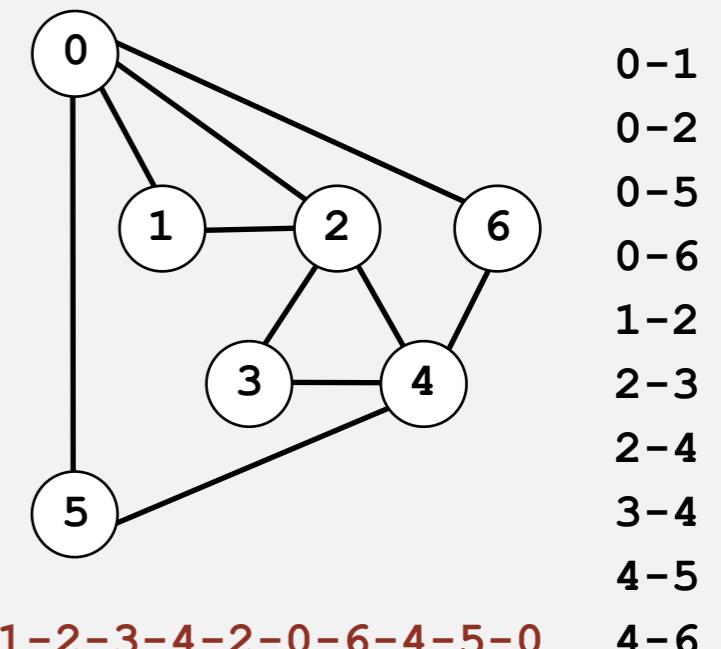
Problem. Find a cycle that uses every edge.

Assumption. Need to use each edge exactly once.

How difficult?

- Any programmer could do it.
- ✓ • Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.

Eulerian tour
(classic graph-processing problem)

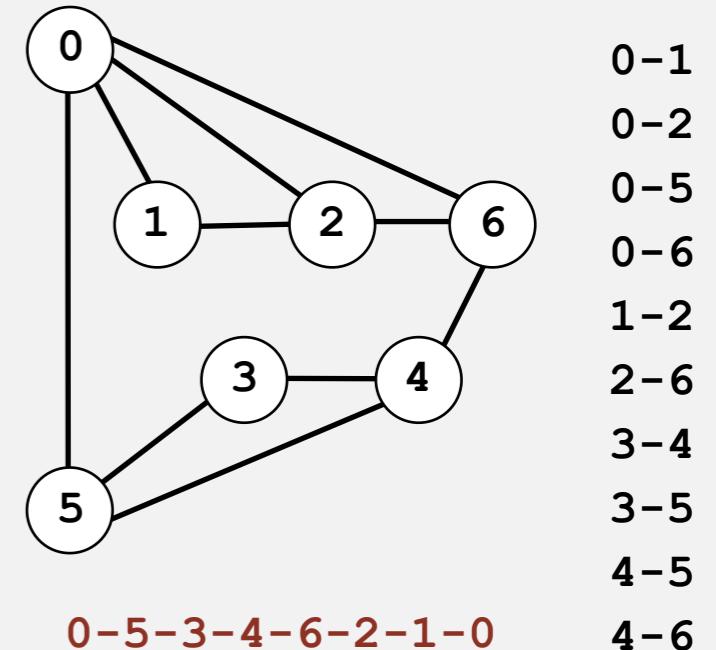


Graph-processing challenge 4

Problem. Find a cycle that visits every vertex exactly once.

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



Graph-processing challenge 4

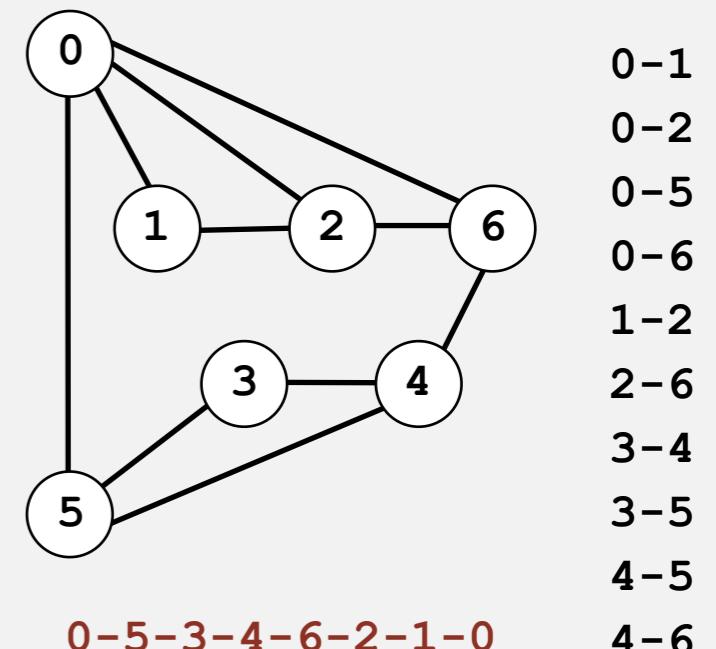
Problem. Find a cycle that visits every vertex.

Assumption. Need to visit each vertex exactly once.

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- ✓ • Intractable.
- No one knows.
- Impossible.

Hamiltonian cycle
(classical NP-complete problem)

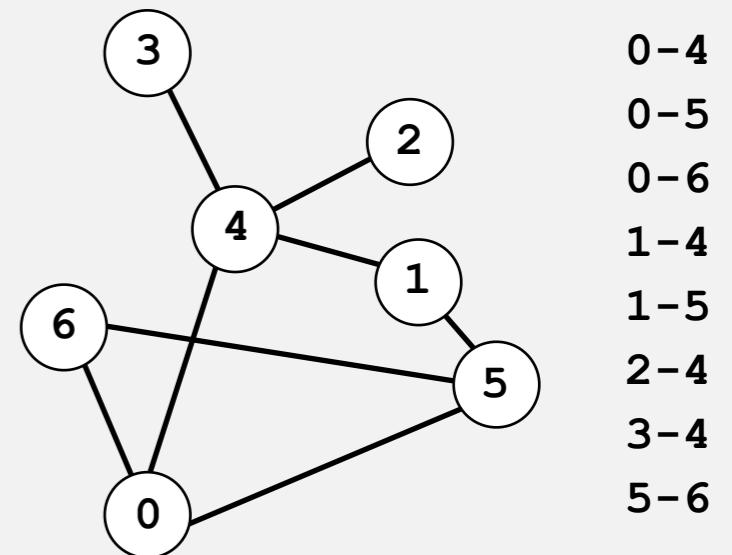
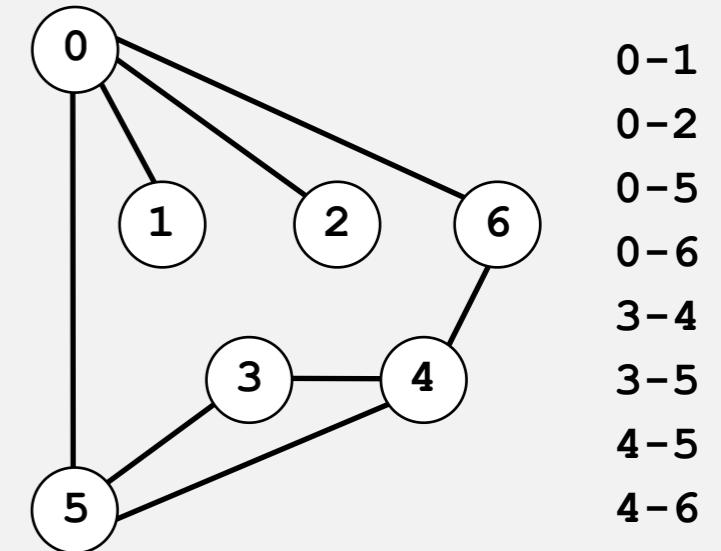


Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



$0 \leftrightarrow 4, 1 \leftrightarrow 3, 2 \leftrightarrow 2, 3 \leftrightarrow 6, 4 \leftrightarrow 5, 5 \leftrightarrow 0, 6 \leftrightarrow 1$

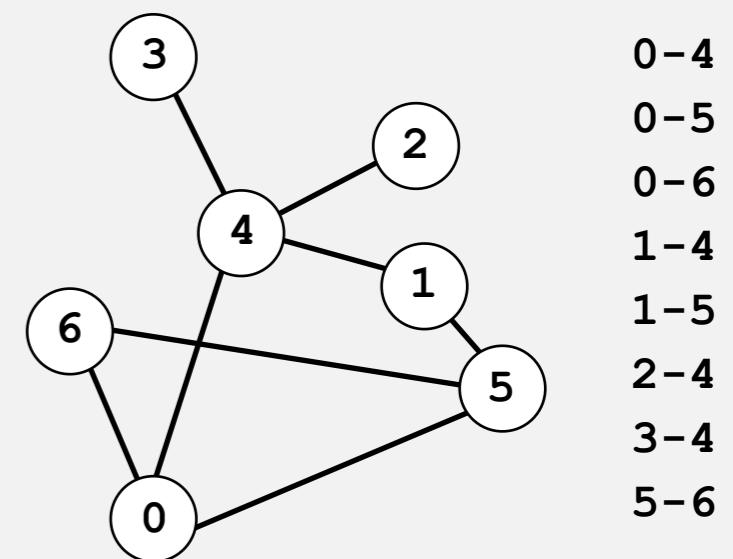
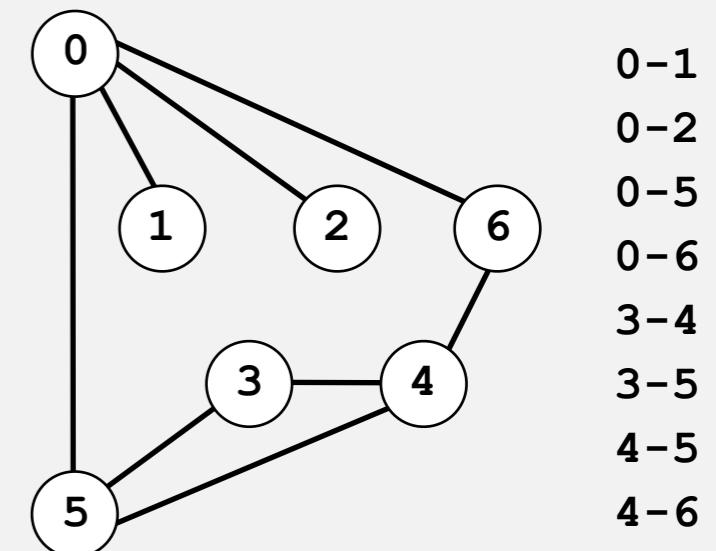
Graph-processing challenge 5

Problem. Are two graphs identical except for vertex names?

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- ✓ • No one knows.
- Impossible.

graph isomorphism is
longstanding open problem



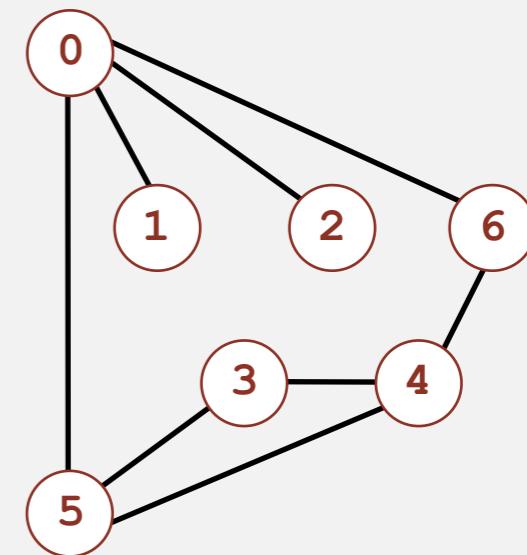
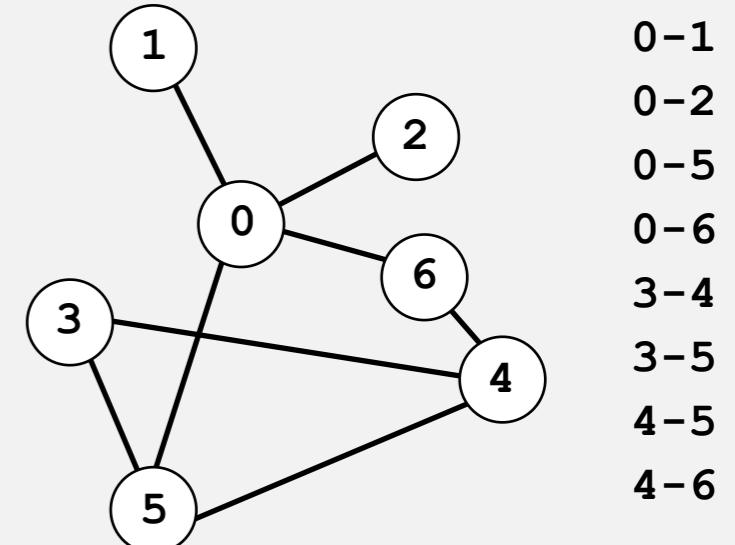
$0 \leftrightarrow 4, 1 \leftrightarrow 3, 2 \leftrightarrow 2, 3 \leftrightarrow 6, 4 \leftrightarrow 5, 5 \leftrightarrow 0, 6 \leftrightarrow 1$

Graph-processing challenge 6

Problem. Lay out a graph in the plane without crossing edges?

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- Hire an expert.
- Intractable.
- No one knows.
- Impossible.



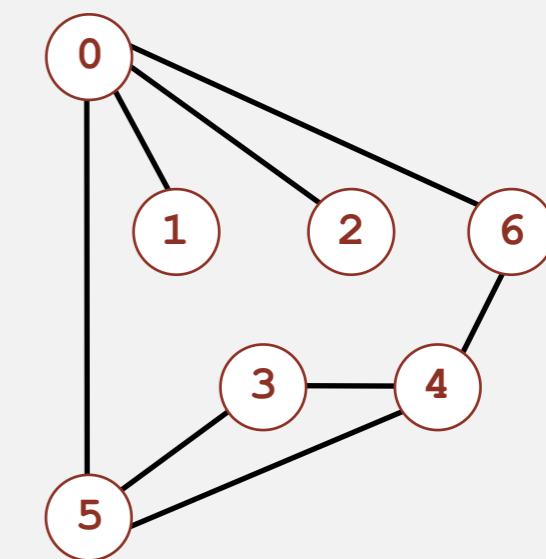
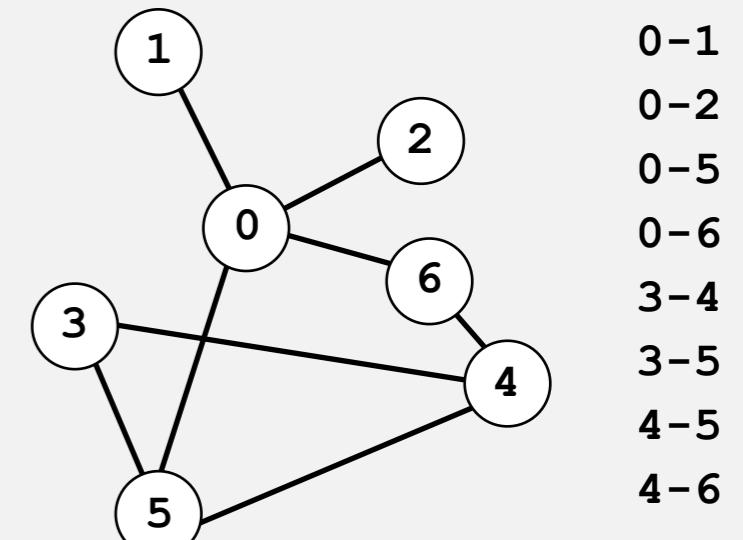
Graph-processing challenge 6

Problem. Lay out a graph in the plane without crossing edges?

How difficult?

- Any programmer could do it.
- Typical diligent algorithms student could do it.
- ✓ • Hire an expert.
- Intractable.
- No one knows.
- Impossible.

linear-time DFS-based planarity algorithm
discovered by Tarjan in 1970s
(too complicated for practitioners)



BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

DIRECTED GRAPHS

Mar. 31, 2016

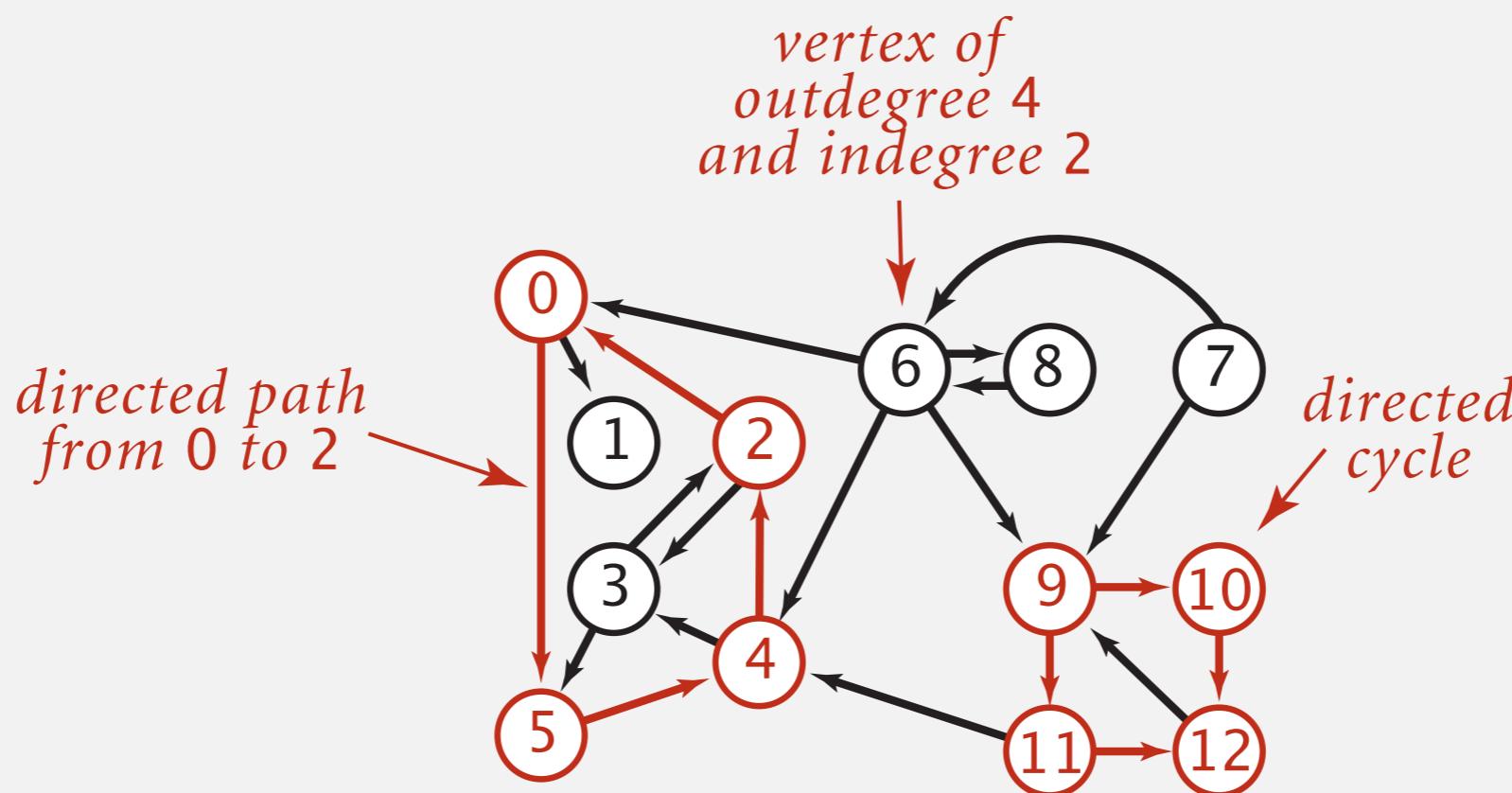
Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ **Directed Graphs**
- ▶ **Digraph API**
- ▶ **Digraph search**
- ▶ **Topological sort**
- ▶ **Strong components**

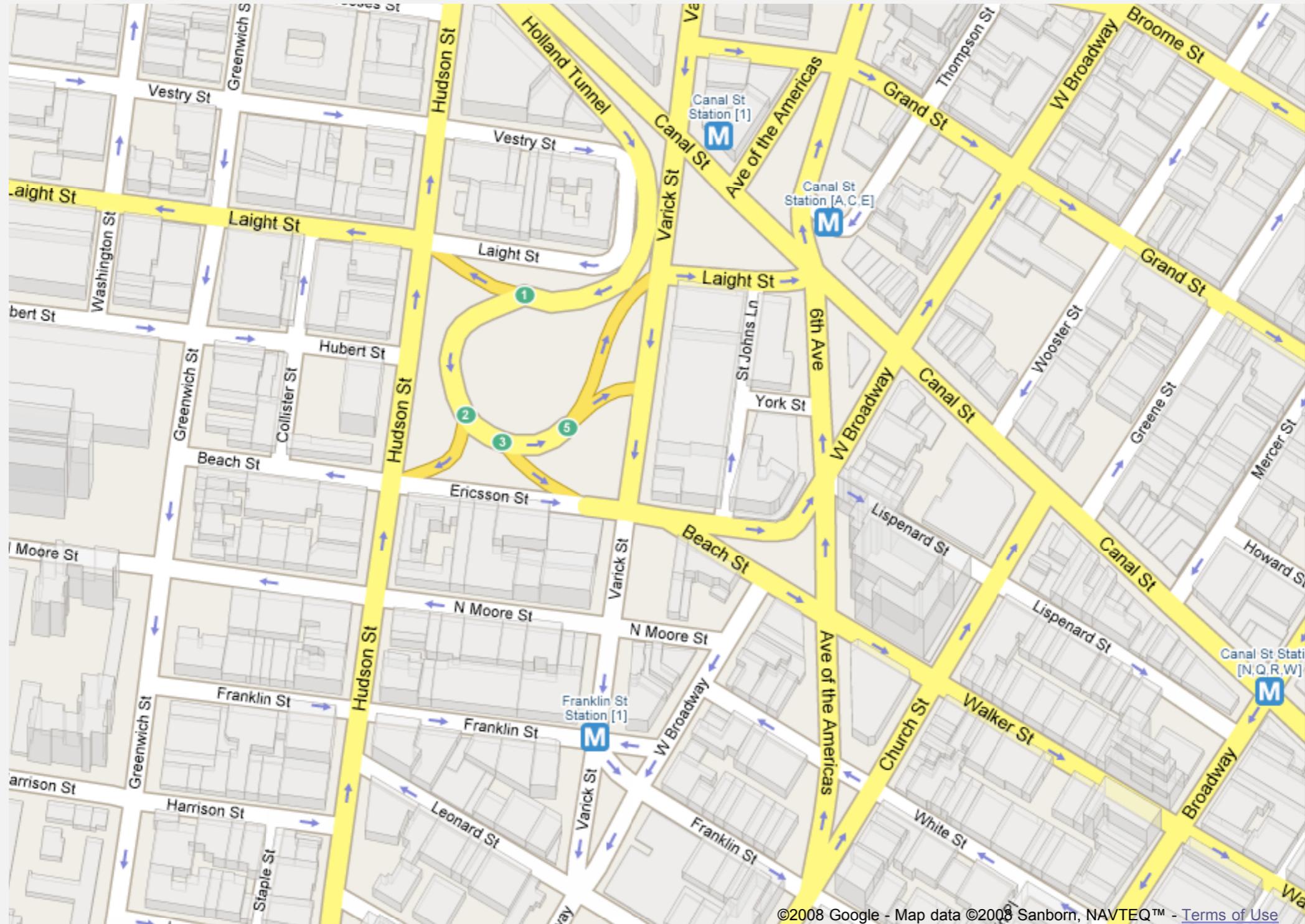
Directed graphs

Digraph. Set of vertices connected pairwise by **directed edges**.



Road network

Vertex = intersection; edge = one-way street.



Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hyponym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

Path. Is there a directed path from s to t ?

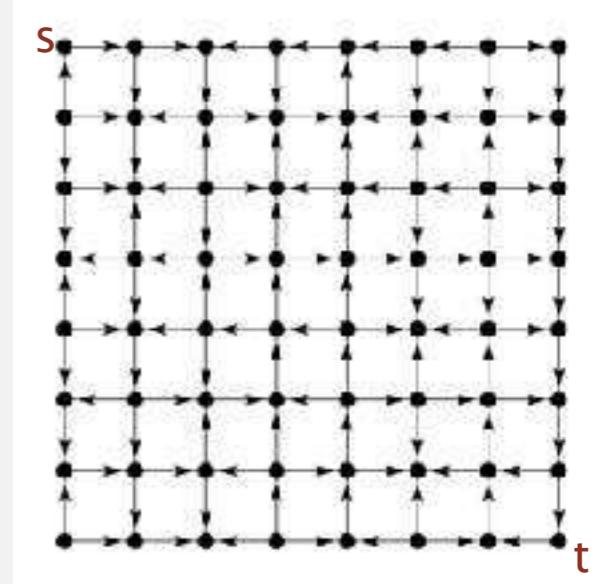
Shortest path. What is the shortest directed path from s to t ?

Topological sort. Can you draw the digraph so that all edges point upwards?

Strong connectivity. Is there a directed path between all pairs of vertices?

Transitive closure. For which vertices v and w is there a path from v to w ?

PageRank. What is the importance of a web page?



DIRECTED GRAPHS

- ▶ **Digraph API**
- ▶ **Digraph search**
- ▶ **Topological sort**
- ▶ **Strong components**

Digraph API

```
public class Digraph
```

```
    Digraph(int v)
```

create an empty digraph with V vertices

```
    Digraph(In in)
```

create a digraph from input stream

```
    void addEdge(int v, int w)
```

add a directed edge $v \rightarrow w$

```
    Iterable<Integer> adj(int v)
```

vertices pointing from v

```
    int V()
```

number of vertices

```
    int E()
```

number of edges

```
    Digraph reverse()
```

reverse of this digraph

```
    String toString()
```

string representation

```
In in = new In(args[0]);
Digraph G = new Digraph(in);

for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

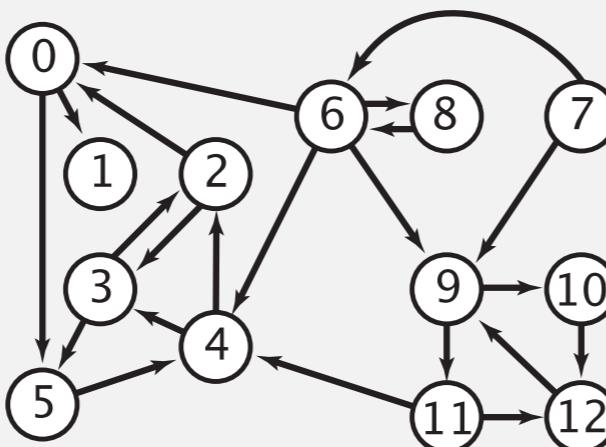
read digraph from
input stream

print out each
edge (once)

Digraph API

tinyDG.txt

V → 13
E ← 22
4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
⋮



```
% java Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
⋮
11->4
11->12
12->9
```

```
In in = new In(args[0]);
Digraph G = new Digraph(in);

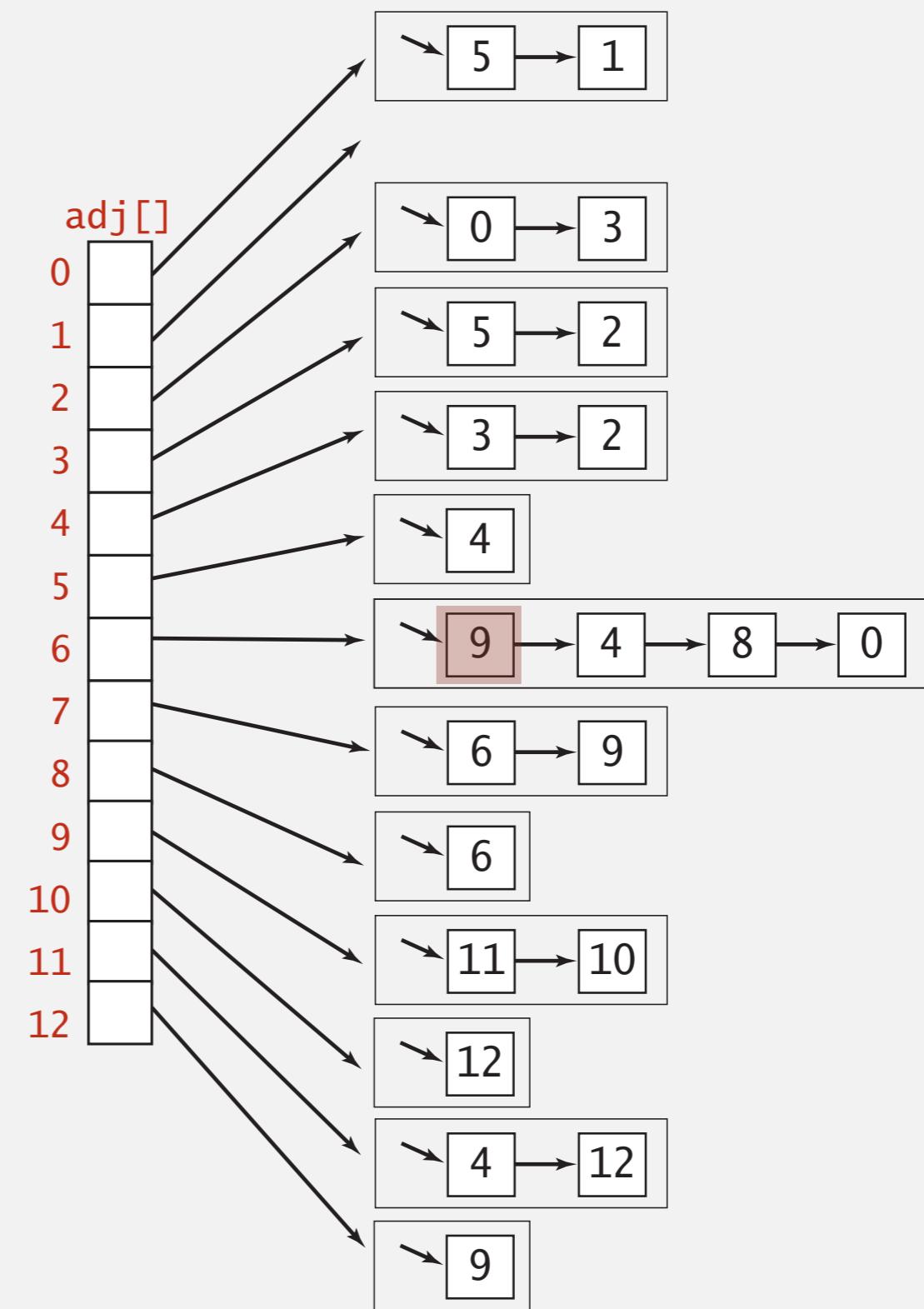
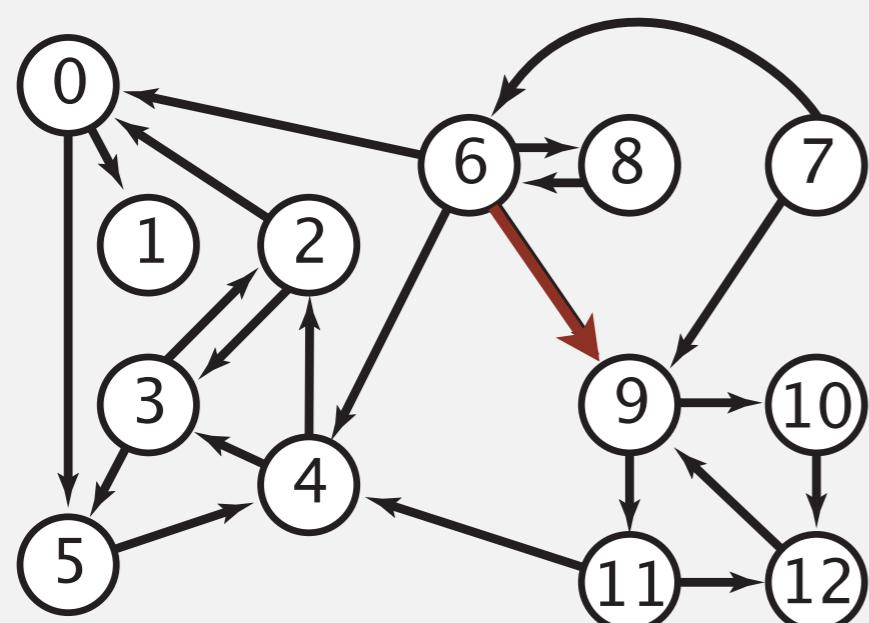
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

read digraph from
input stream

print out each
edge (once)

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.



Adjacency-lists graph representation: Java implementation

```
public class Graph
{
    private final int V;
    private final Bag<Integer>[] adj; ← adjacency lists

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) ← add edge v-w
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) ← iterator for vertices
    {   return adj[v];   } adjacent to v
}
```

Adjacency-lists digraph representation: Java implementation

```
public class Digraph
{
    private final int V;
    private final Bag<Integer>[] adj; ← adjacency lists

    public Digraph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) ← add edge v→w
    {
        adj[v].add(w);
    }

    public Iterable<Integer> adj(int v) ← iterator for vertices
    {   return adj[v];   }               pointing from v
}
```

Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v .
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	insert edge from v to w	edge from v to w ?	iterate over vertices pointing from v ?
list of edges	E	1	E	E
adjacency matrix	V^2	$1 \dagger$	1	V
adjacency lists	$E + V$	1	outdegree(v)	outdegree(v)

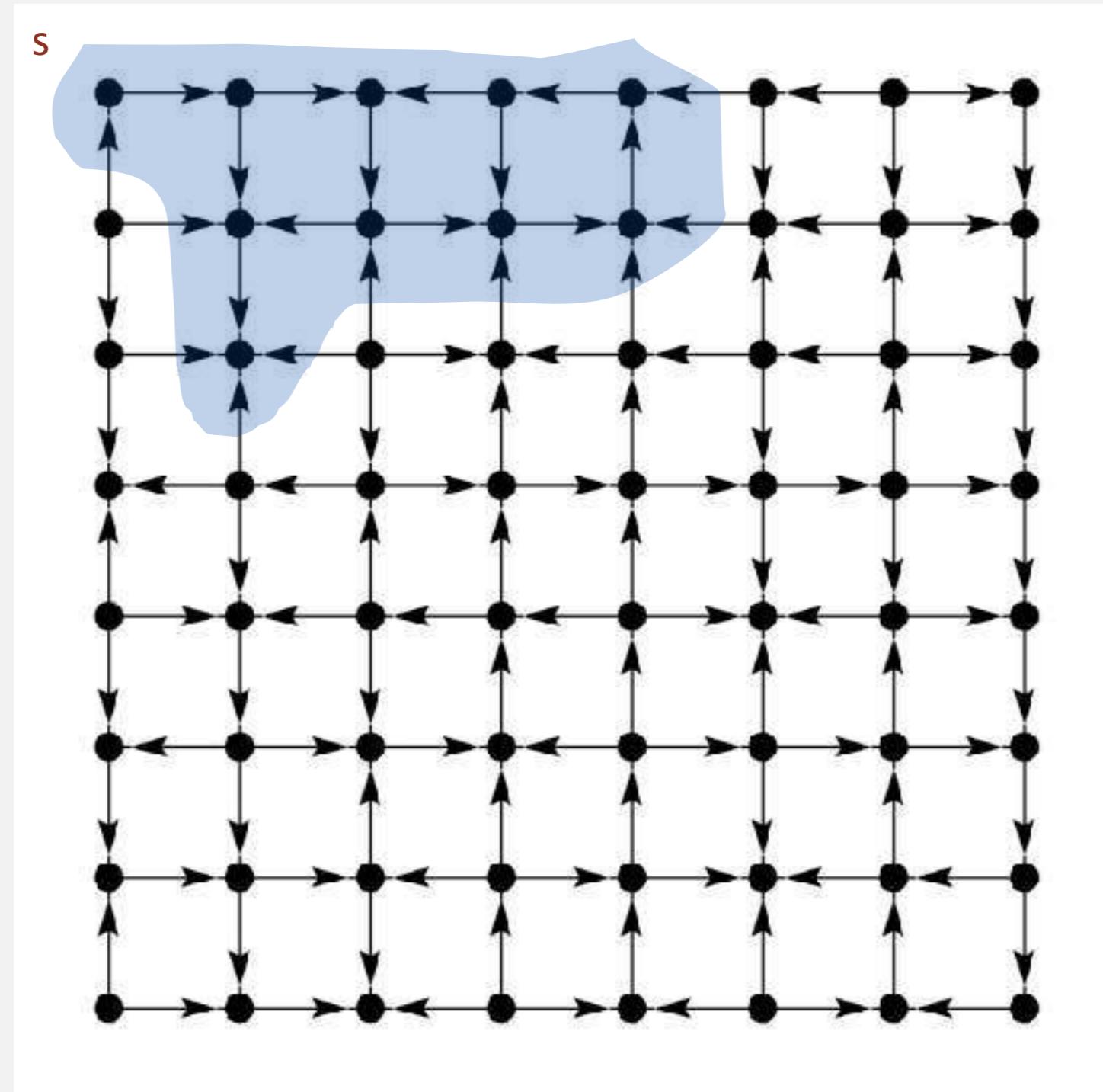
\dagger disallows parallel edges

DIRECTED GRAPHS

- ▶ **Digraph API**
- ▶ **Digraph search**
- ▶ **Topological sort**
- ▶ **Strong components**

Reachability

Problem. Find all vertices reachable from s along a directed path.



Depth-first search in digraphs

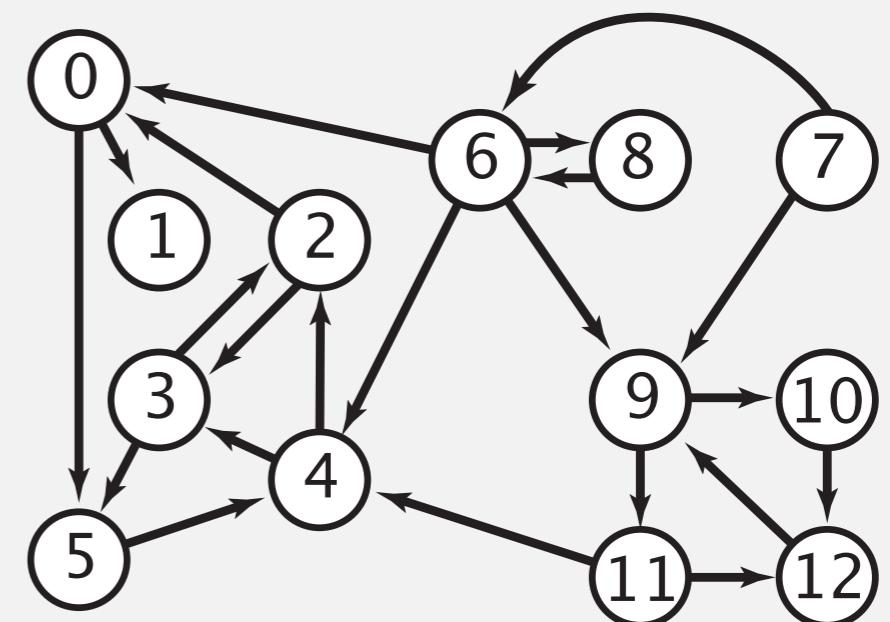
Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a **digraph** algorithm.

DFS (to visit a vertex v)

Mark v as visited.

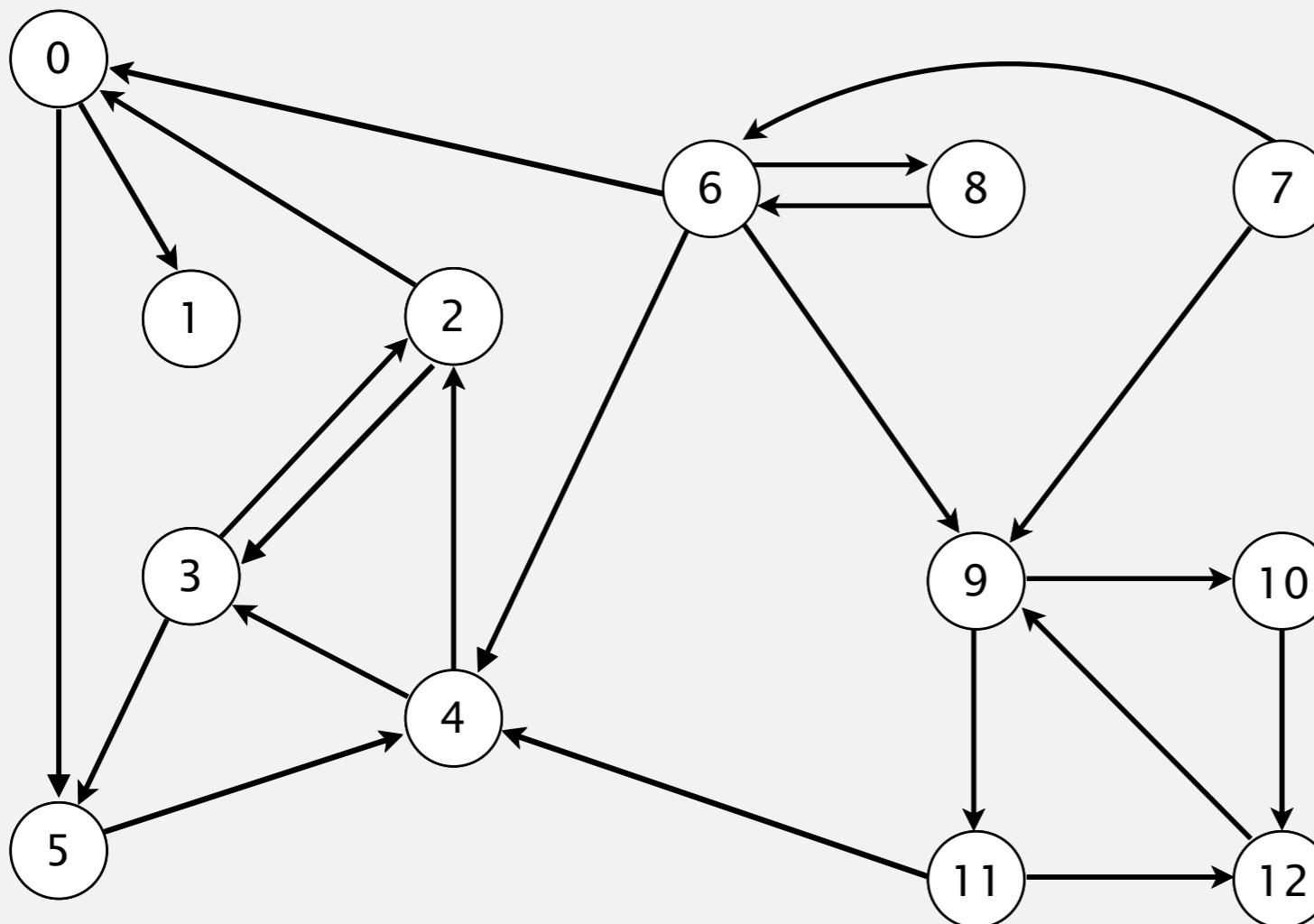
Recursively visit all unmarked
vertices w pointing from v.



Depth-first search

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v .



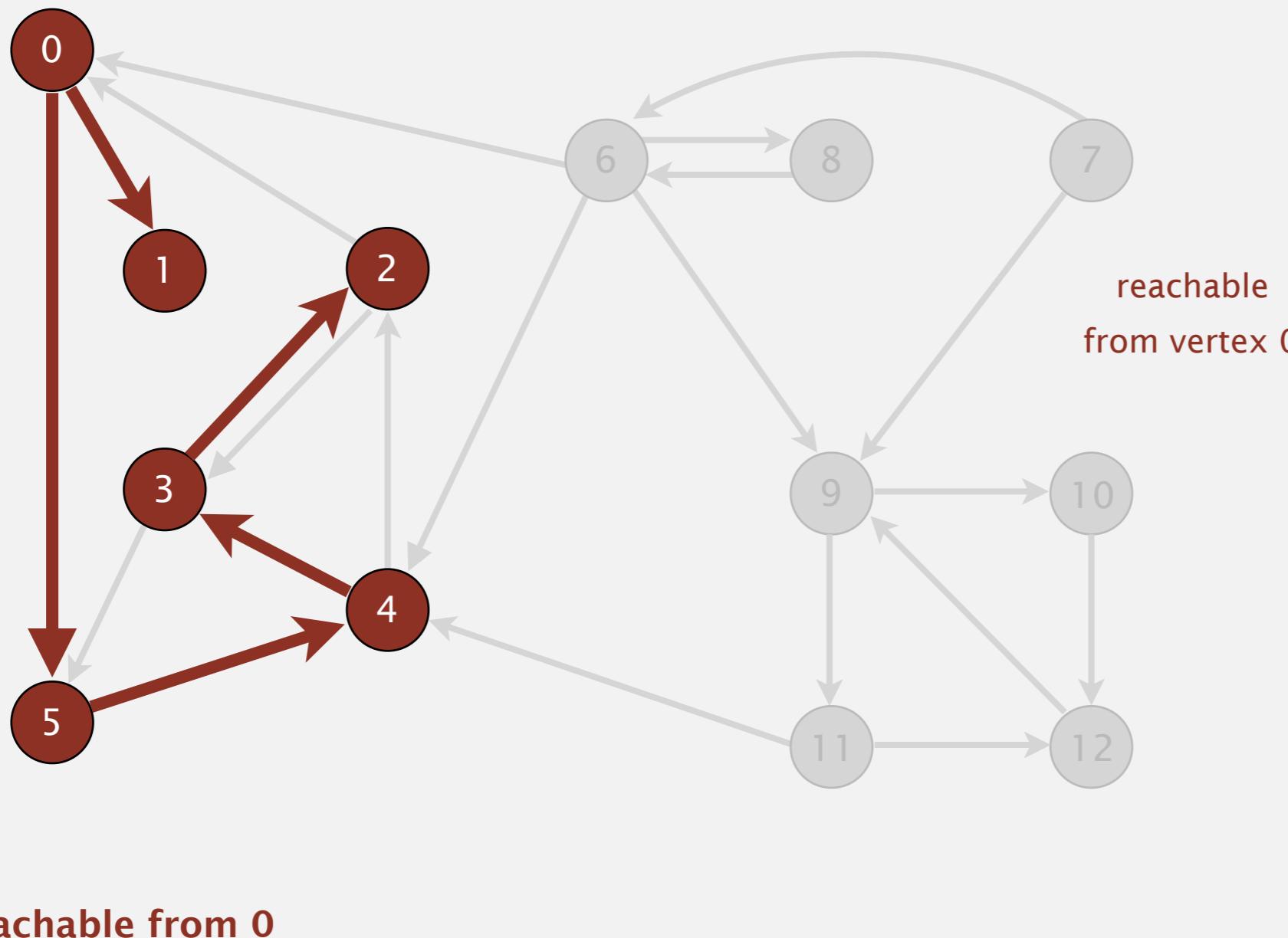
a directed graph

4→2
2→3
3→2
6→0
0→1
2→0
11→12
12→9
9→10
9→11
8→9
10→12
11→4
4→3
3→5
6→8
8→6
5→4
0→5
6→4
6→9
7→6

Depth-first search

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v .



v	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

Depth-first search (in undirected graphs)

Recall code for **undirected** graphs.

```
public class DepthFirstSearch
{
    private boolean[] marked; ← true if path to s

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()]; ← constructor marks
        dfs(G, s);               vertices connected to s
    }

    private void dfs(Graph G, int v) ← recursive DFS does the work
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) ← client can ask whether any
    {   return marked[v]; }           vertex is connected to s
}
```

Depth-first search (in directed graphs)

Code for **directed** graphs identical to undirected one.

[substitute **Digraph** for **Graph**]

```
public class DirectedDFS
{
    private boolean[] marked; ← true if path from s

    public DirectedDFS(Digraph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Digraph G, int v) ← recursive DFS does the work
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v) ← client can ask whether any
    {   return marked[v]; } vertex is reachable from s
}
```

Reachability application: program control-flow analysis

Every program is a digraph.

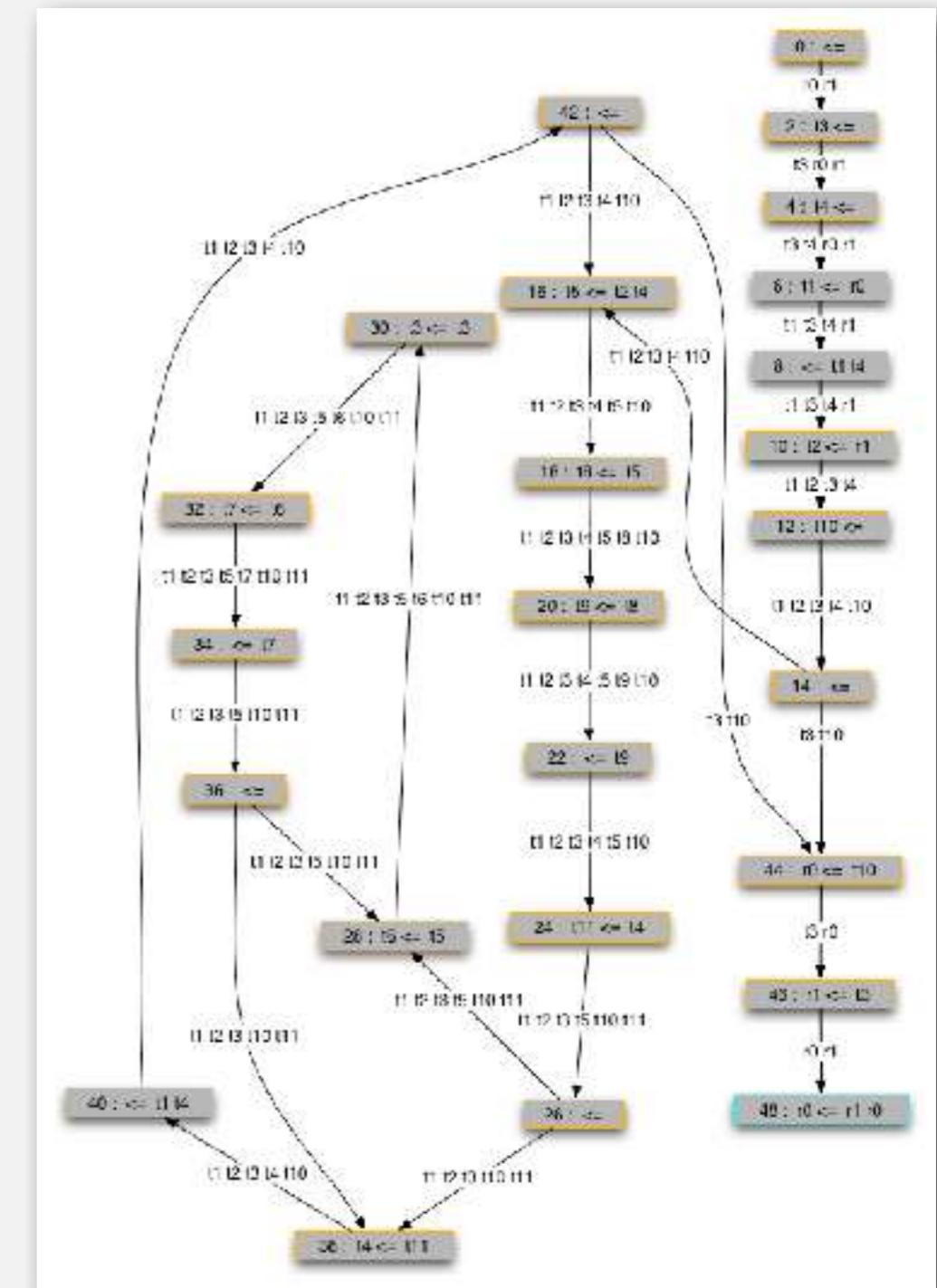
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



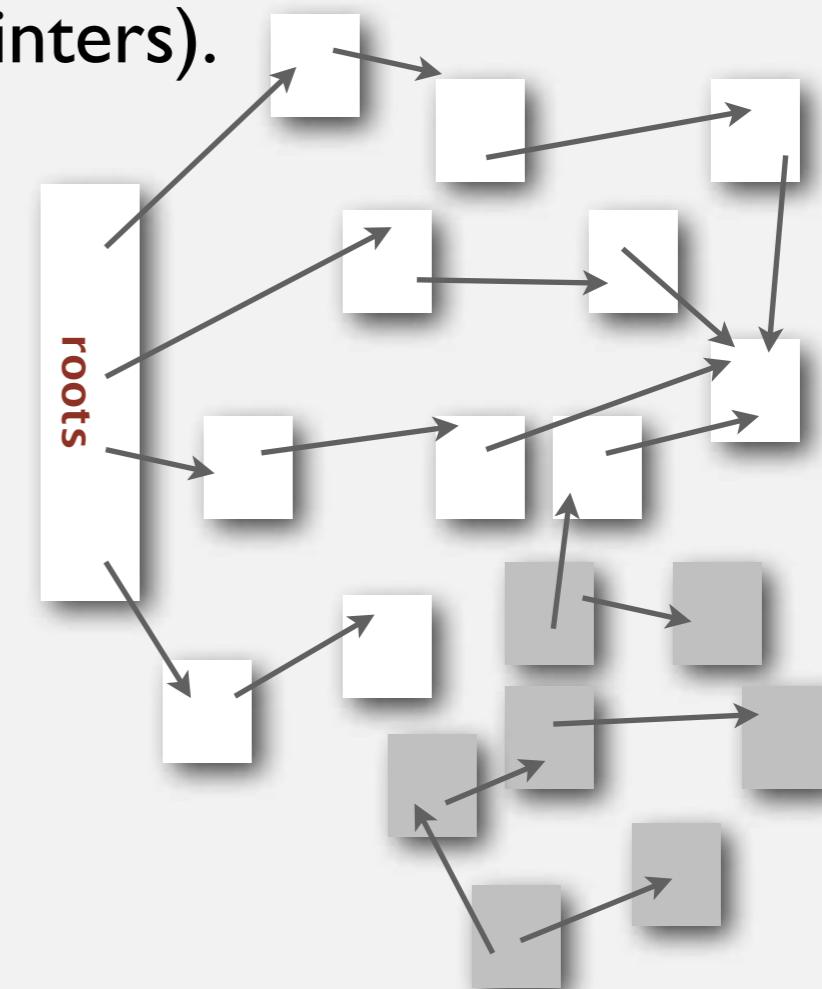
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program
(starting at a root and following a chain of pointers).

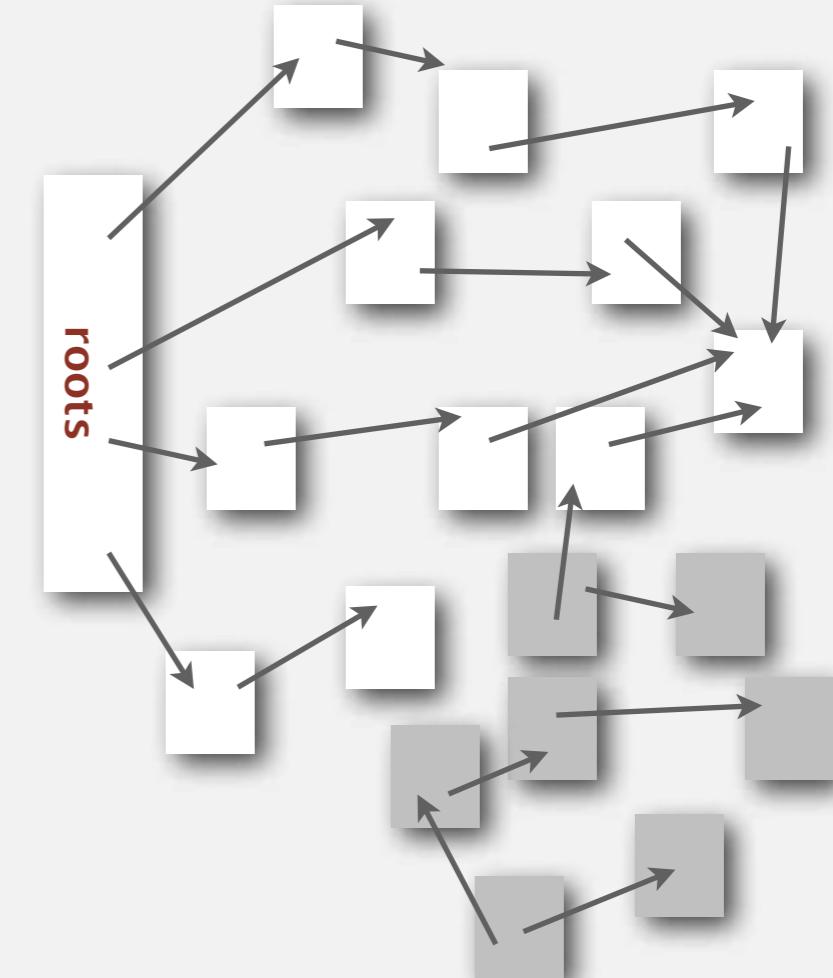


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



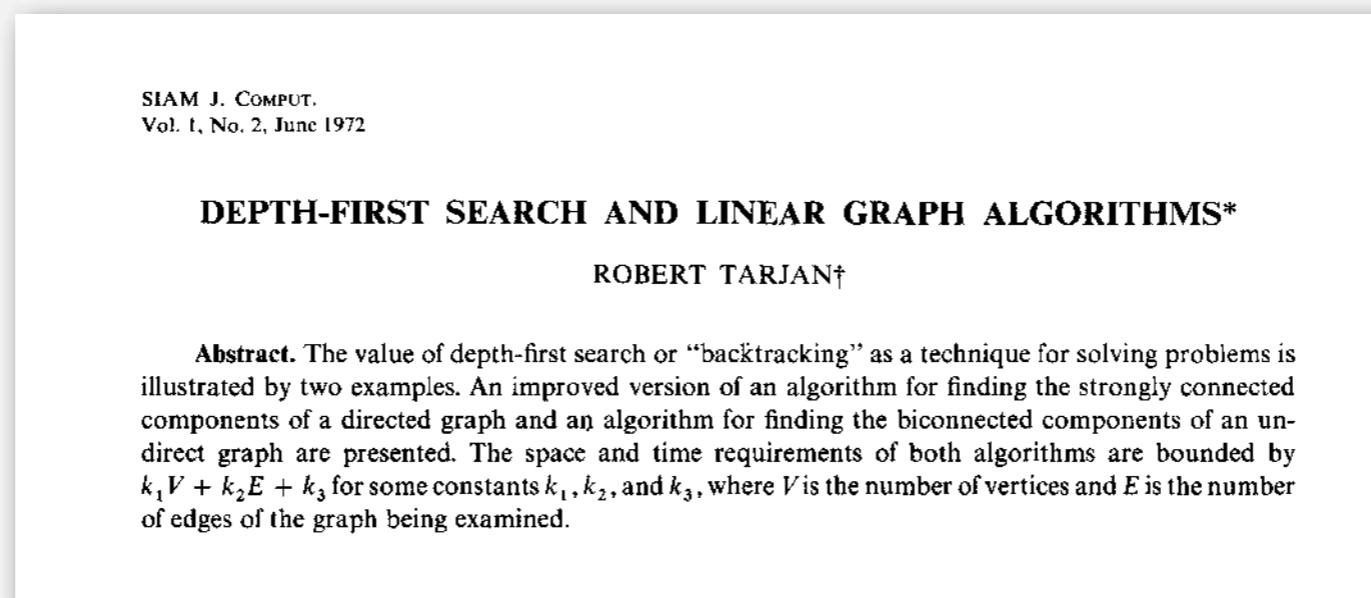
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

- ✓ • Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.



Breadth-first search in digraphs

Same method as for undirected graphs.

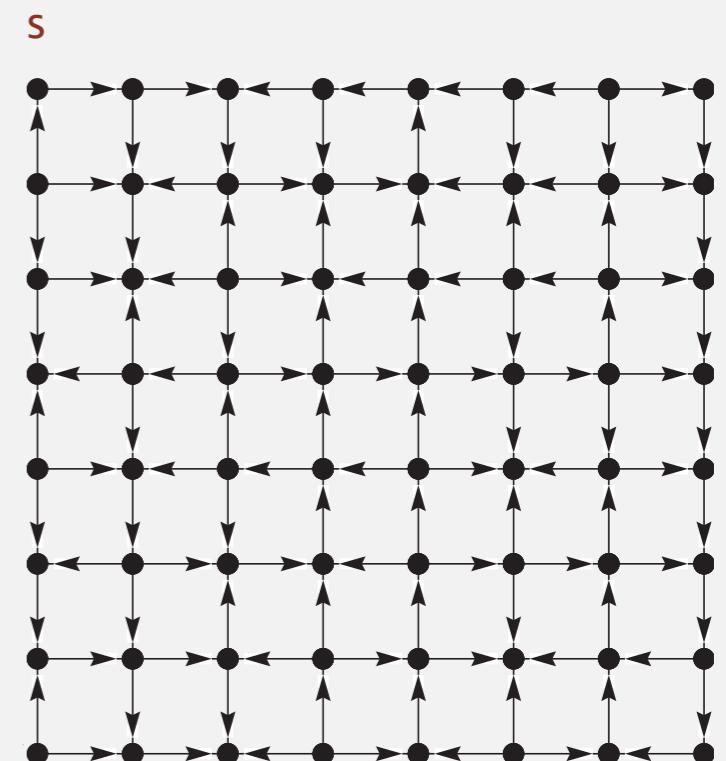
- Every undirected graph is a digraph (with edges in both directions).
- BFS is a **digraph** algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

- remove the least recently added vertex v
- for each unmarked vertex pointing from v :
 - add to queue and mark as visited.



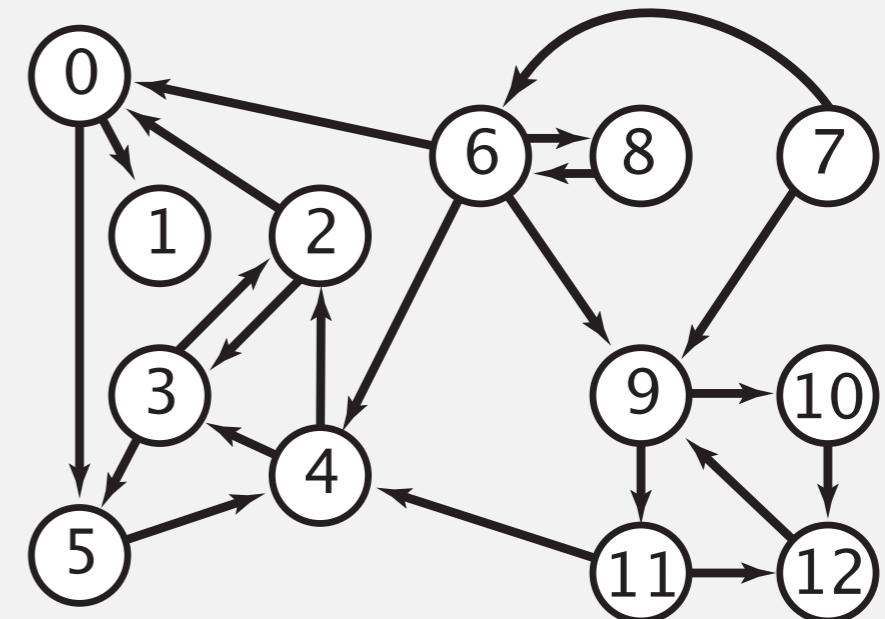
Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E+V$.

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a **set** of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. $S = \{1, 7, 10\}$.

- Shortest path to 4 is $7 \rightarrow 6 \rightarrow 4$.
- Shortest path to 5 is $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$.
- Shortest path to 12 is $10 \rightarrow 12$.



Q. How to implement multi-source constructor?

A. Use BFS, but initialize by enqueueing all source vertices.

Breadth-first search in digraphs application: web crawler

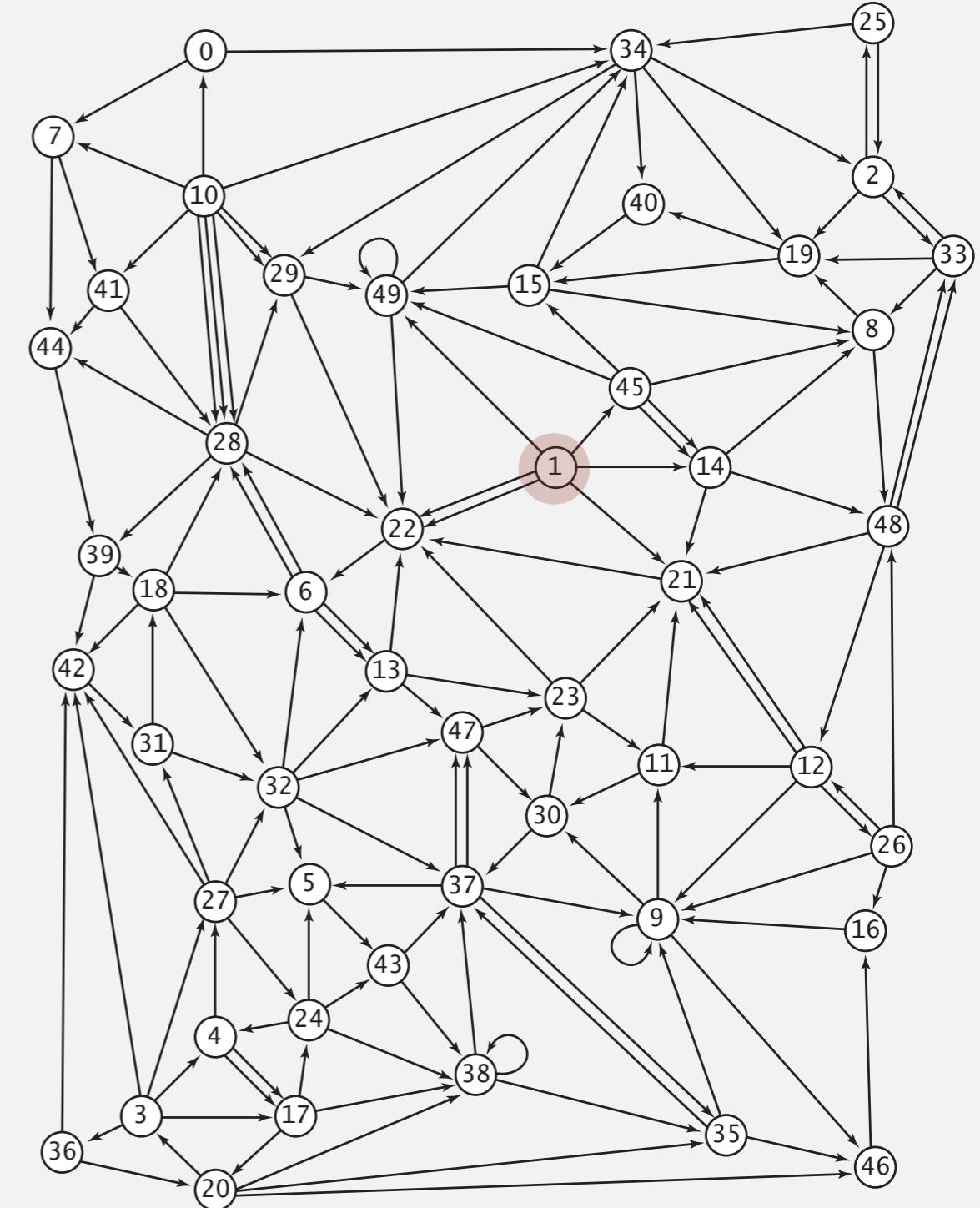
Goal. Crawl web, starting from some root web page, say www.princeton.edu.

Solution. BFS with implicit graph.

BFS.

- Choose root web page as source s .
- Maintain a **Queue** of websites to explore.
- Maintain a **SET** of discovered websites.
- Dequeue the next website and enqueue websites to which it links
(provided you haven't done so before).

Q. Why not use DFS?



Bare-bones web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();           ← queue of websites to crawl
SET<String> discovered = new SET<String>();          ← set of discovered websites

String root = "http://www.princeton.edu";
queue.enqueue(root);
discovered.add(root);

while (!queue.isEmpty())
{
    String v = queue.dequeue();
    StdOut.println(v);
    In in = new In(v);
    String input = in.readAll();

    String regexp = "http://(\w+\.\w+)*(\w+)";
    Pattern pattern = Pattern.compile(regexp);           ← read in raw html from next
    Matcher matcher = pattern.matcher(input);            ← website in queue
    while (matcher.find())
    {
        String w = matcher.group();
        if (!discovered.contains(w))
        {
            discovered.add(w);
            queue.enqueue(w);
        }
    }
}
```

use regular expression to find all URLs
in website of form `http://xxx.yyy.zzz`
[crude pattern misses relative URLs]

if undiscovered, mark it as discovered
and put on queue

DIRECTED GRAPHS

- ▶ **Digraph API**
- ▶ **Digraph search**
- ▶ **Topological sort**
- ▶ **Strong components**

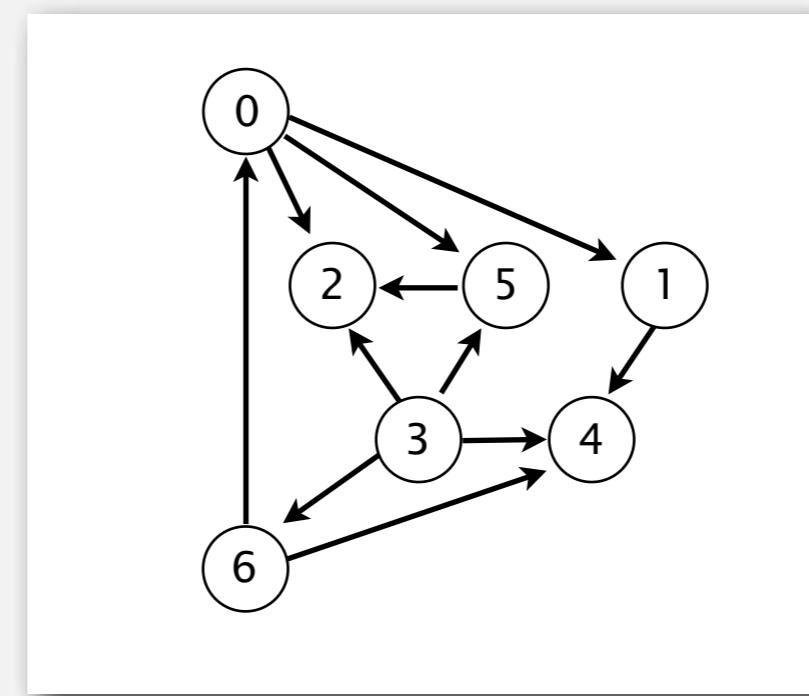
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

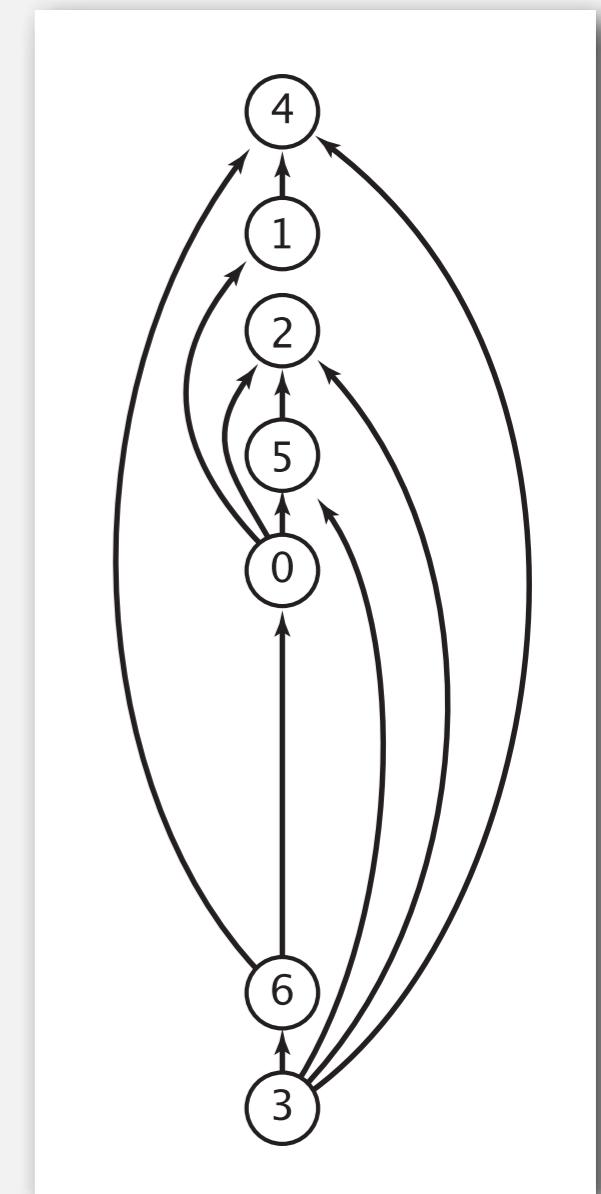
Digraph model. vertex = task; edge = precedence constraint.

- 0. Algorithms
- 1. Complexity Theory
- 2. Artificial Intelligence
- 3. Intro to CS
- 4. Cryptography
- 5. Scientific Computing
- 6. Advanced Programming

tasks



precedence constraint graph



feasible schedule

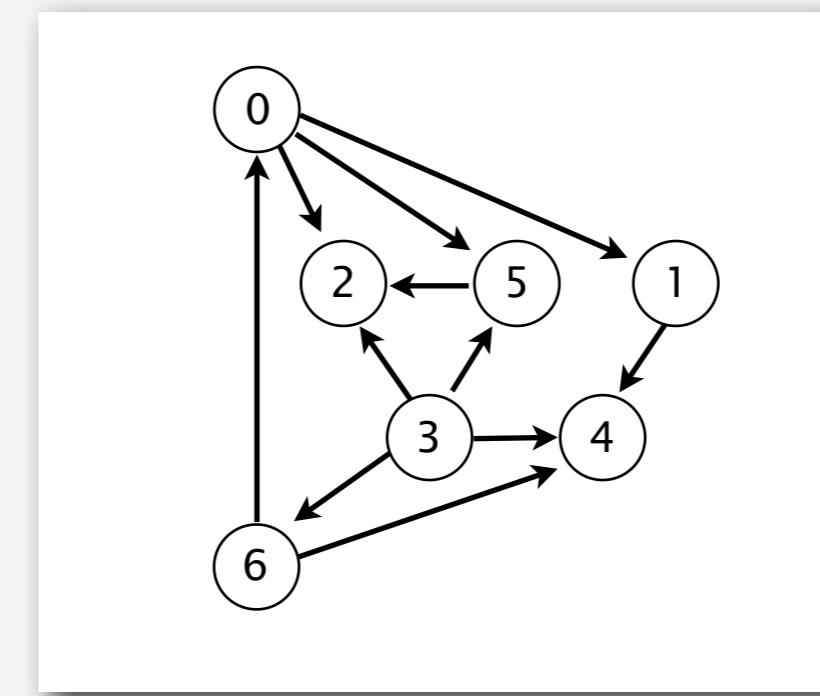
Topological sort

DAG. Directed **acyclic** graph.

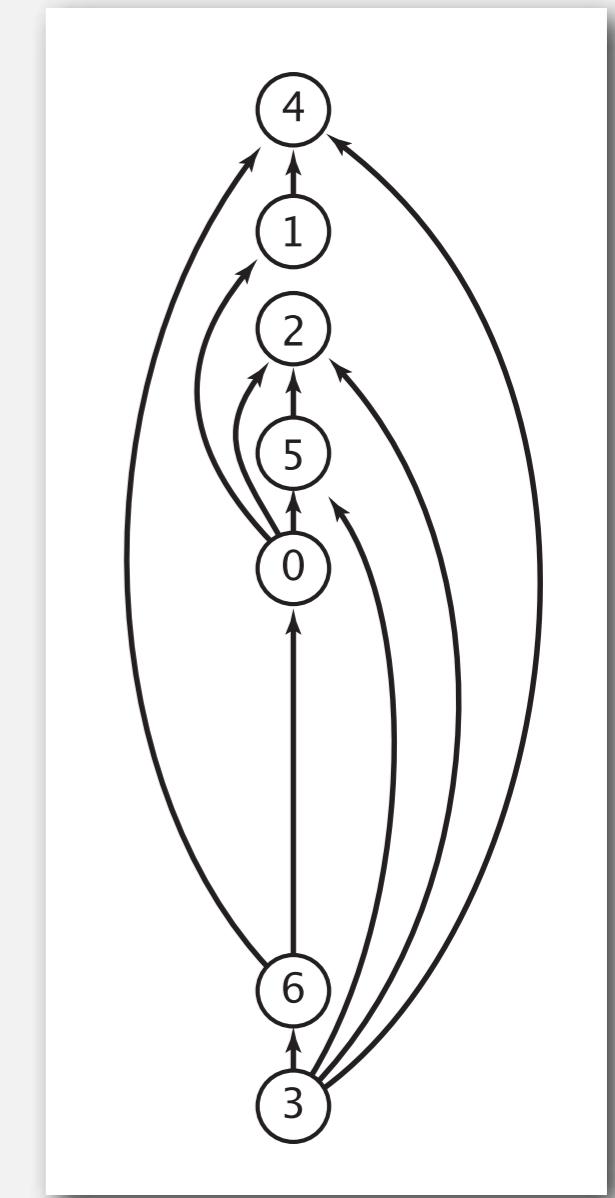
Topological sort. Redraw DAG so all edges point upwards.

$0 \rightarrow 5$	$0 \rightarrow 2$
$0 \rightarrow 1$	$3 \rightarrow 6$
$3 \rightarrow 5$	$3 \rightarrow 4$
$5 \rightarrow 2$	$6 \rightarrow 4$
$6 \rightarrow 0$	$3 \rightarrow 2$
$1 \rightarrow 4$	

directed edges



DAG

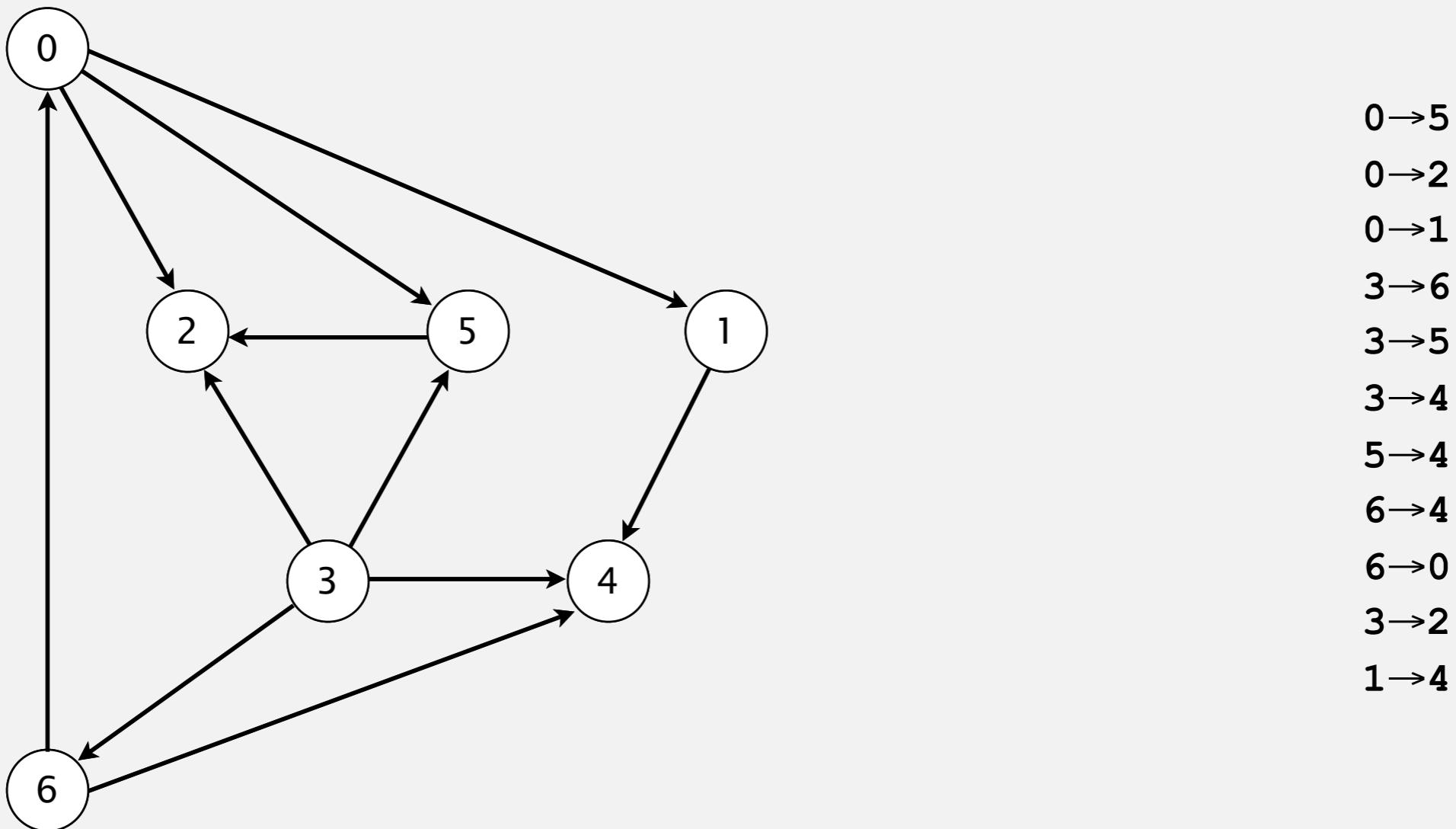


topological order

Solution. DFS. What else?

Topological sort algorithm

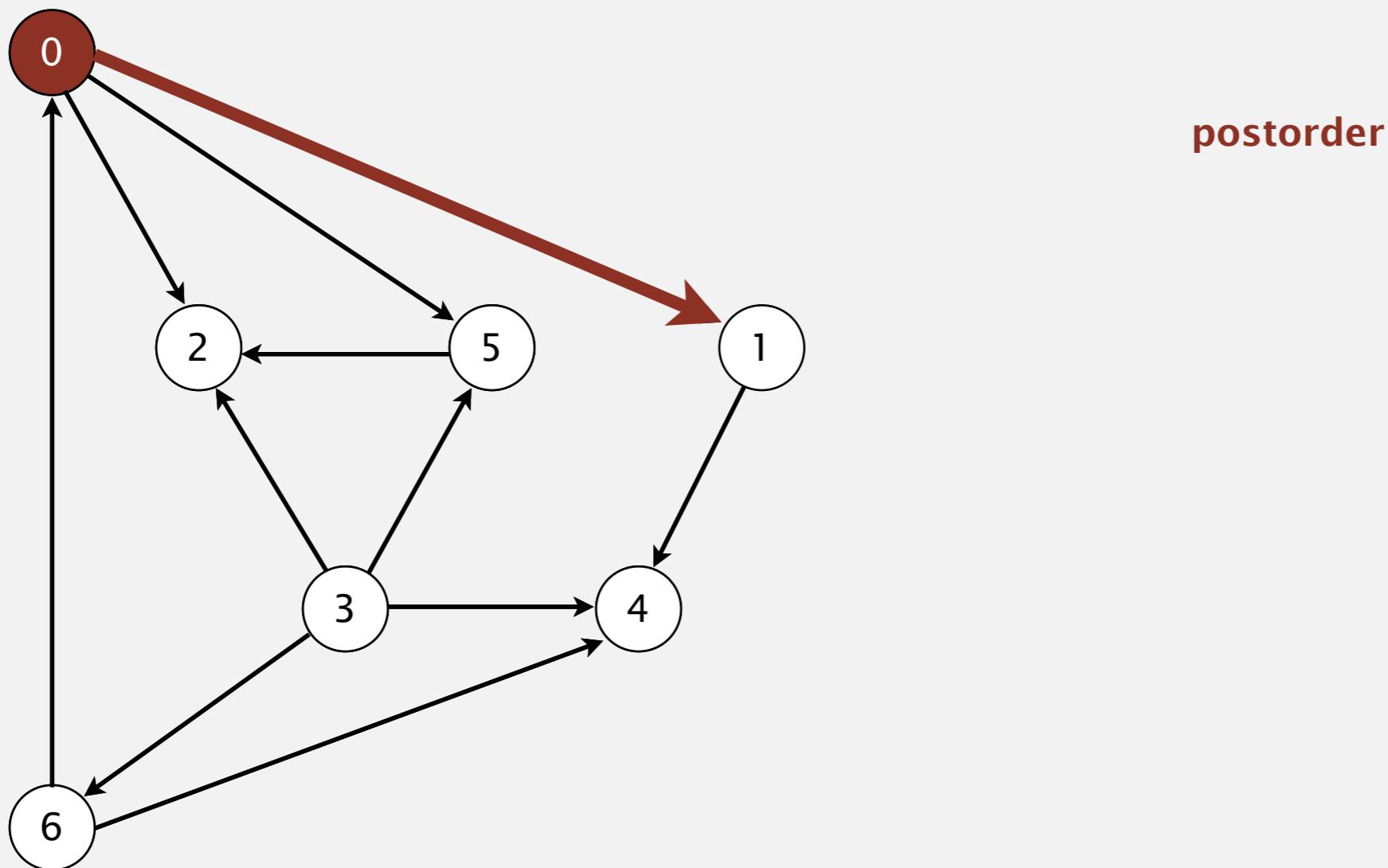
- Run depth-first search.
- Return vertices in reverse postorder.



a directed acyclic graph

Topological sort algorithm

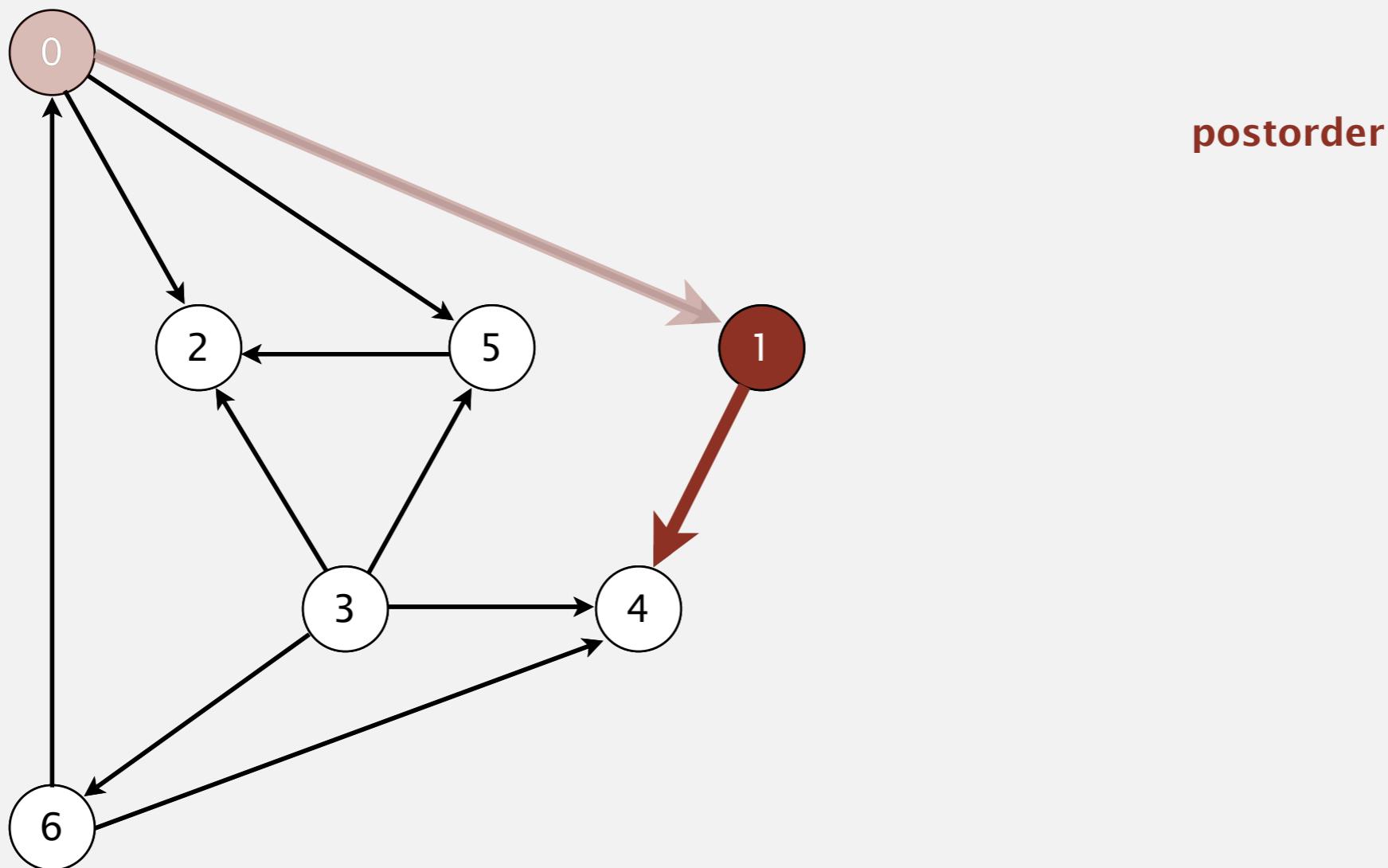
- Run depth-first search.
- Return vertices in reverse postorder.



visit 0: check 1, check 2, and check 5

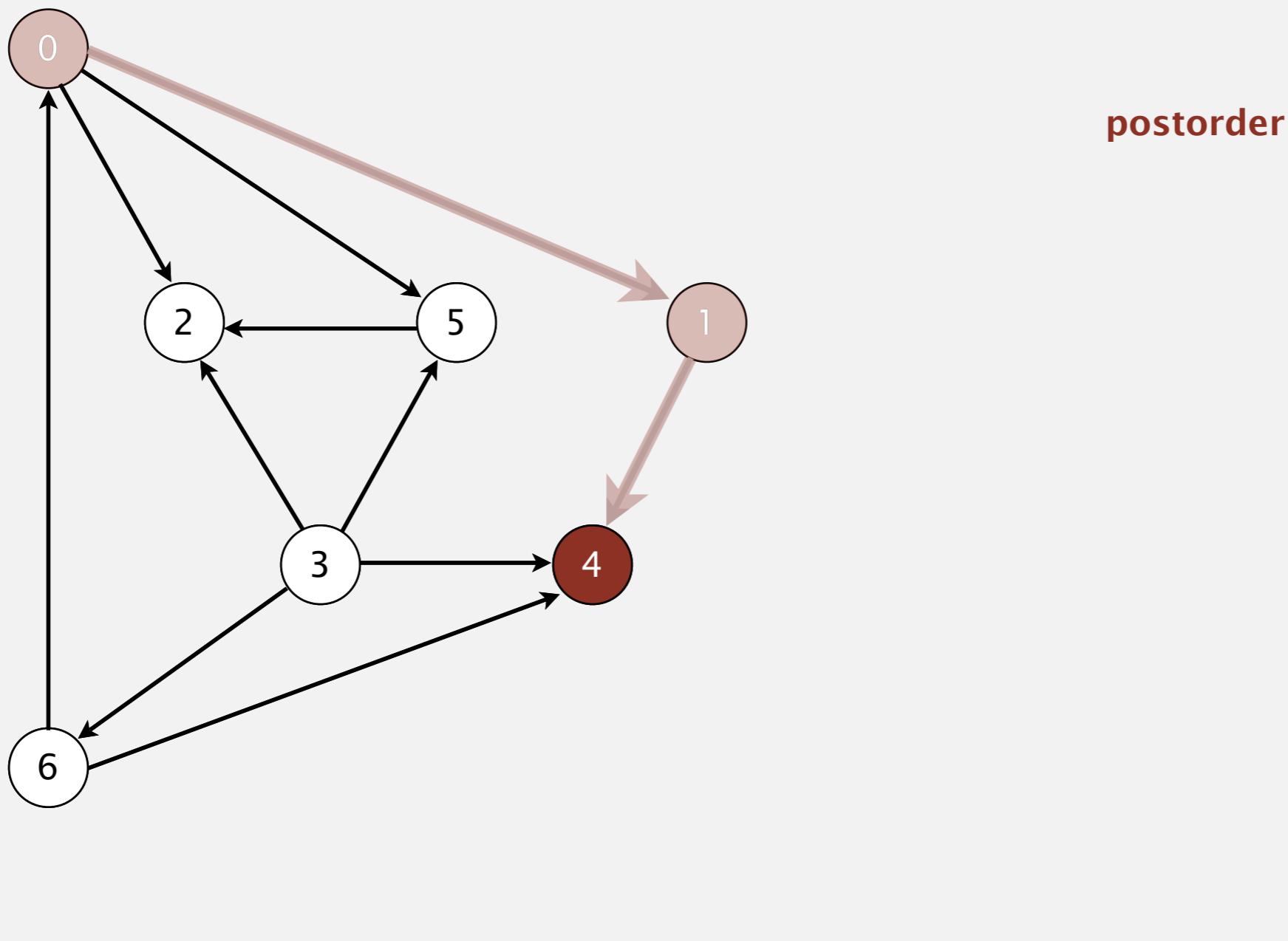
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



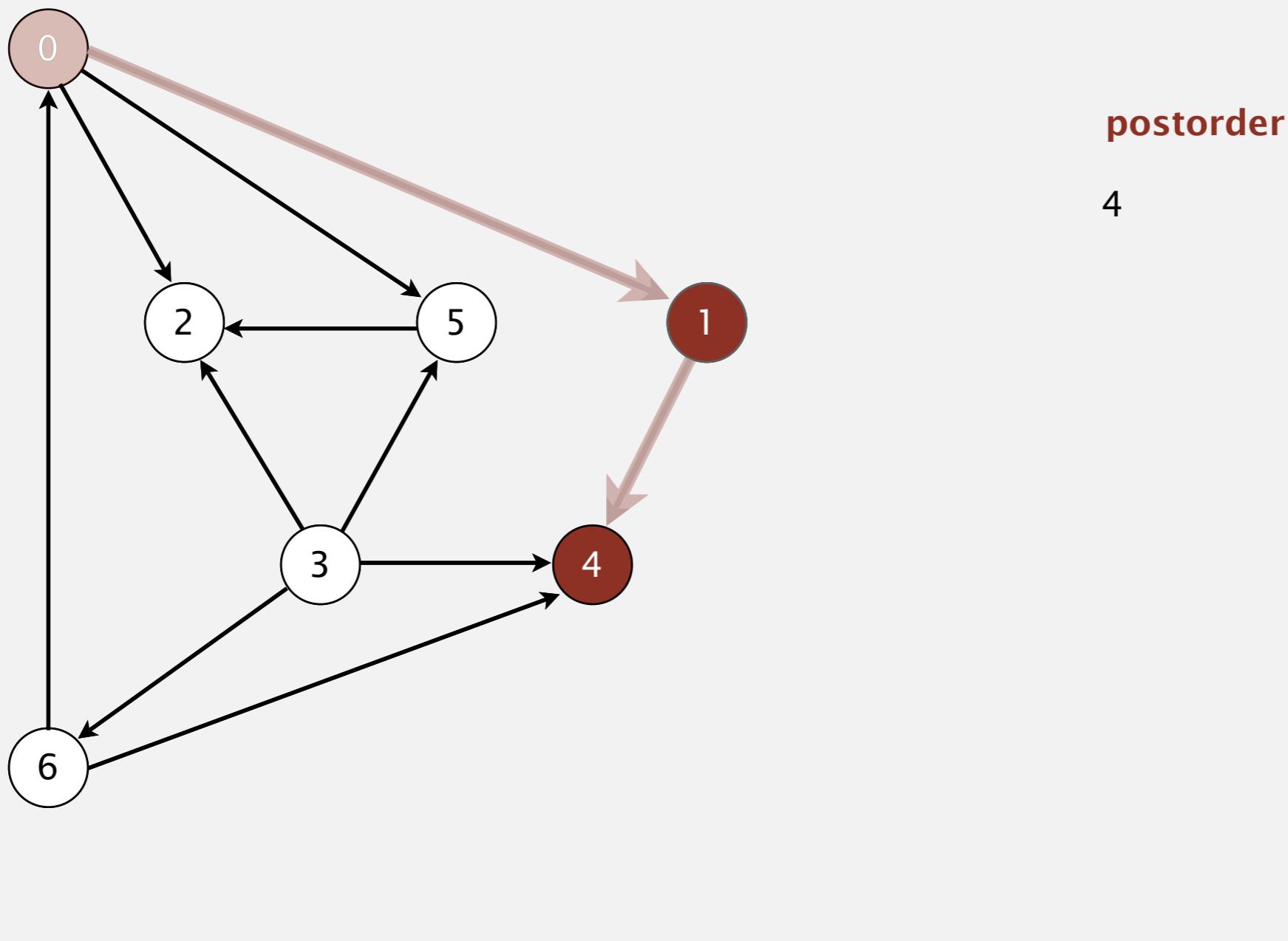
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



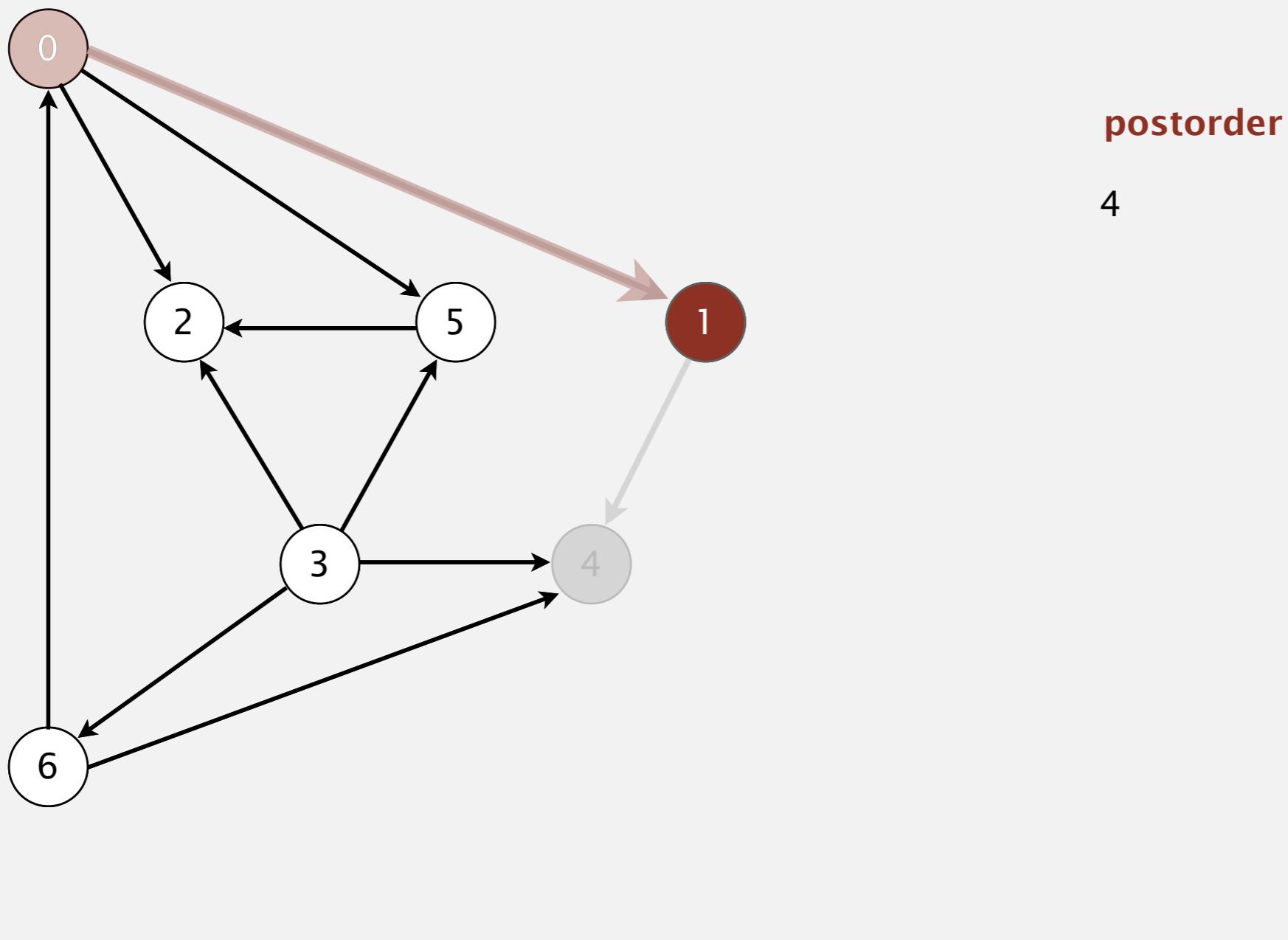
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



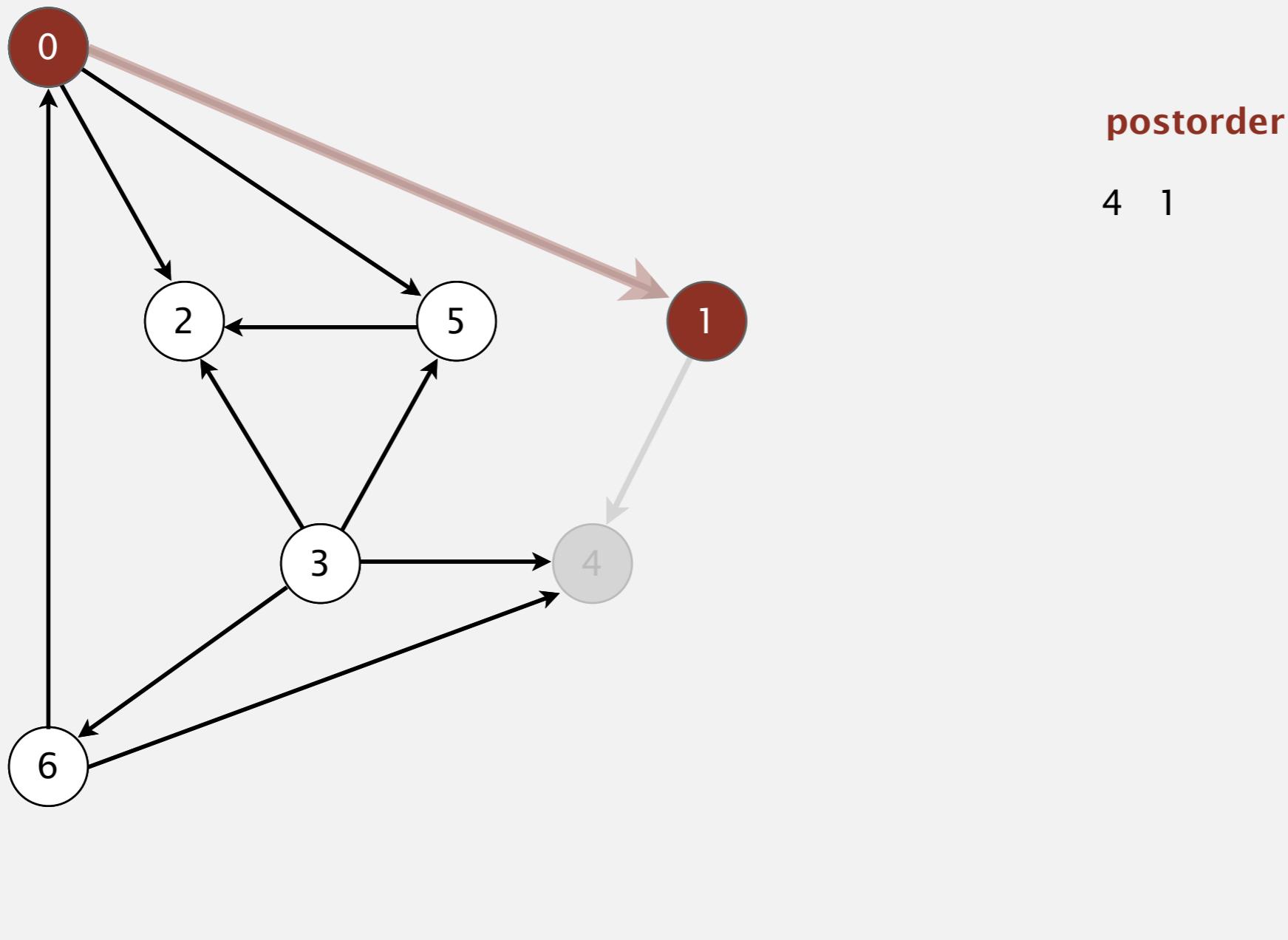
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



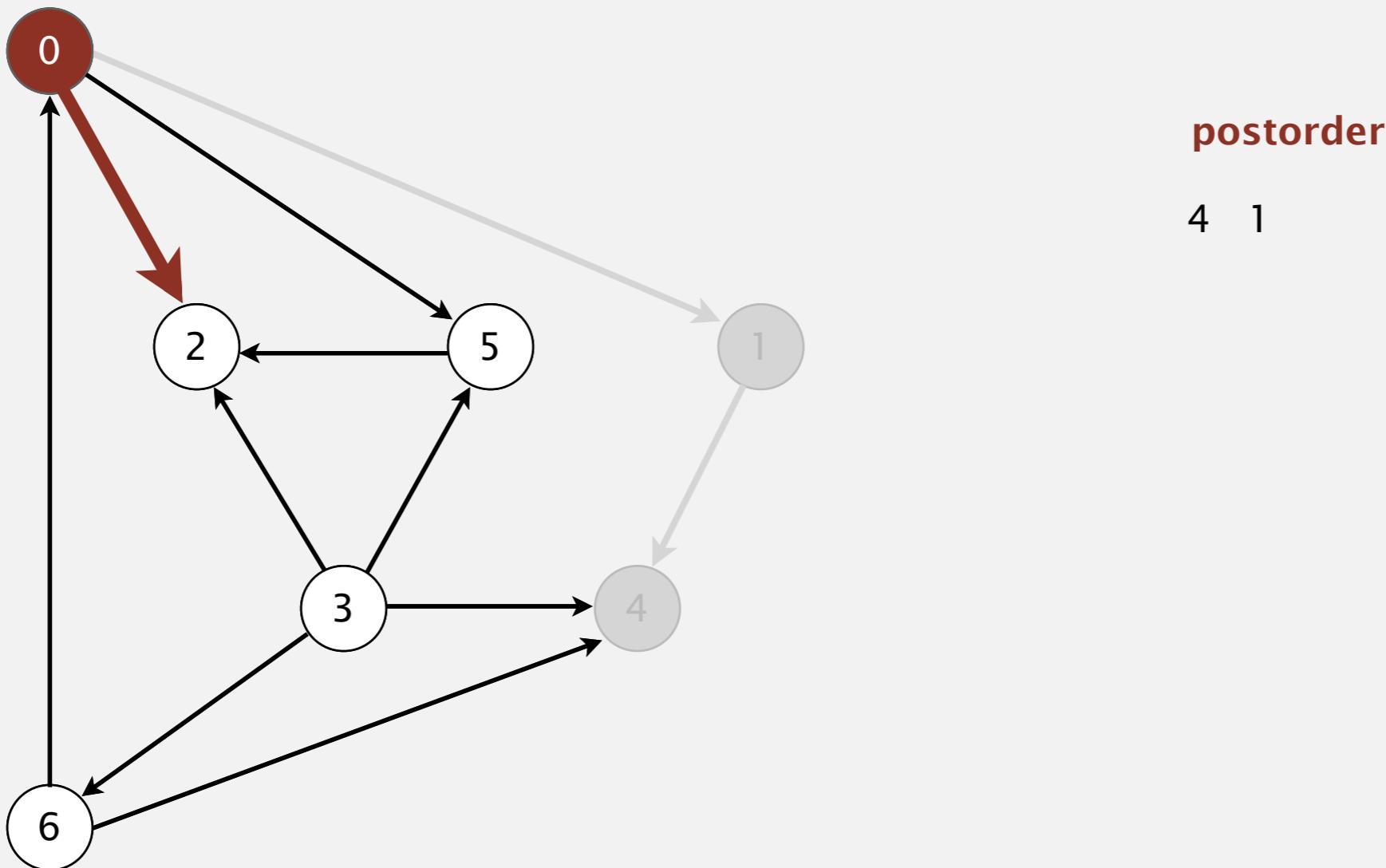
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



Topological sort algorithm

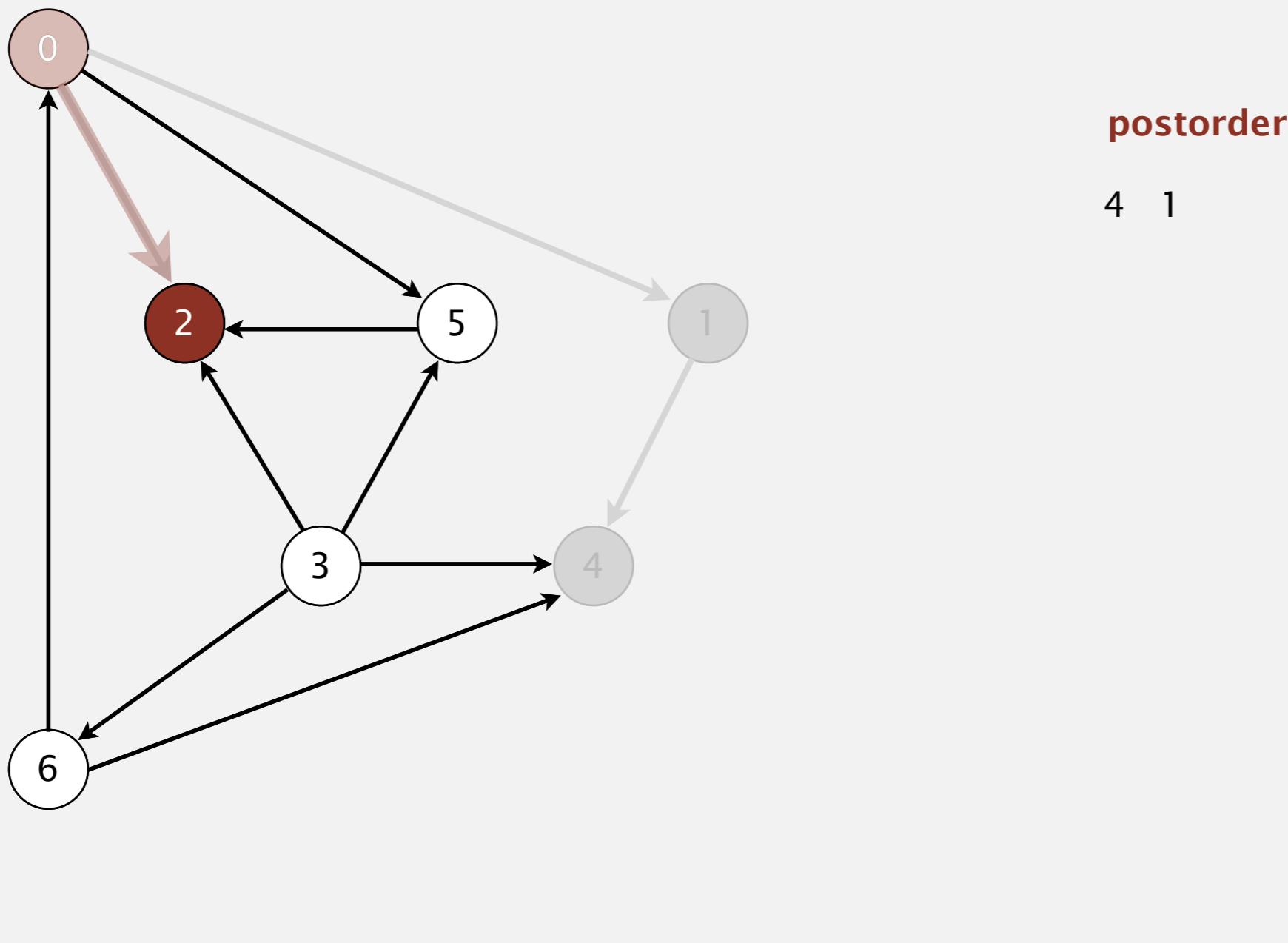
- Run depth-first search.
- Return vertices in reverse postorder.



visit 0: check 1, check 2, and check 5

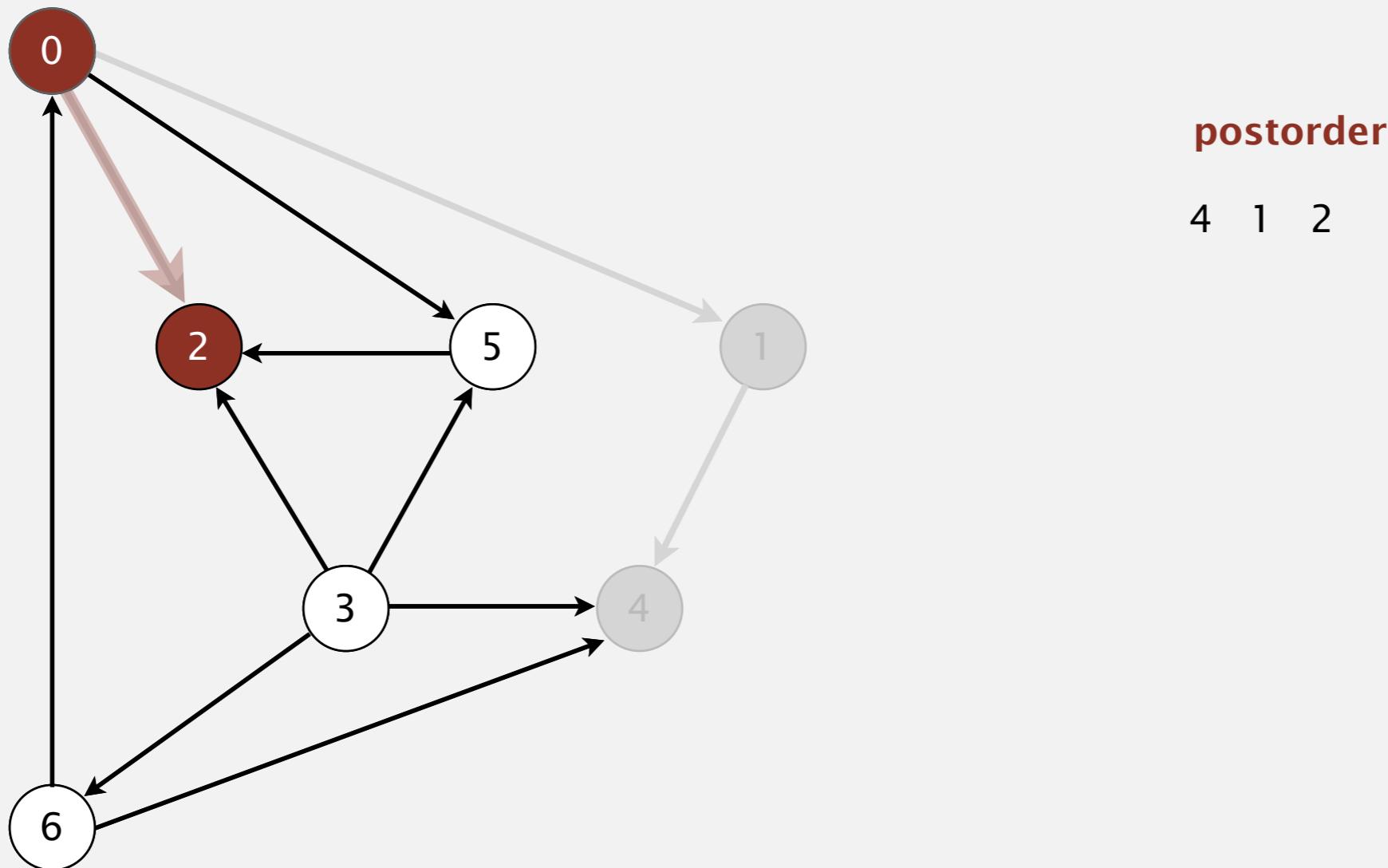
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



Topological sort algorithm

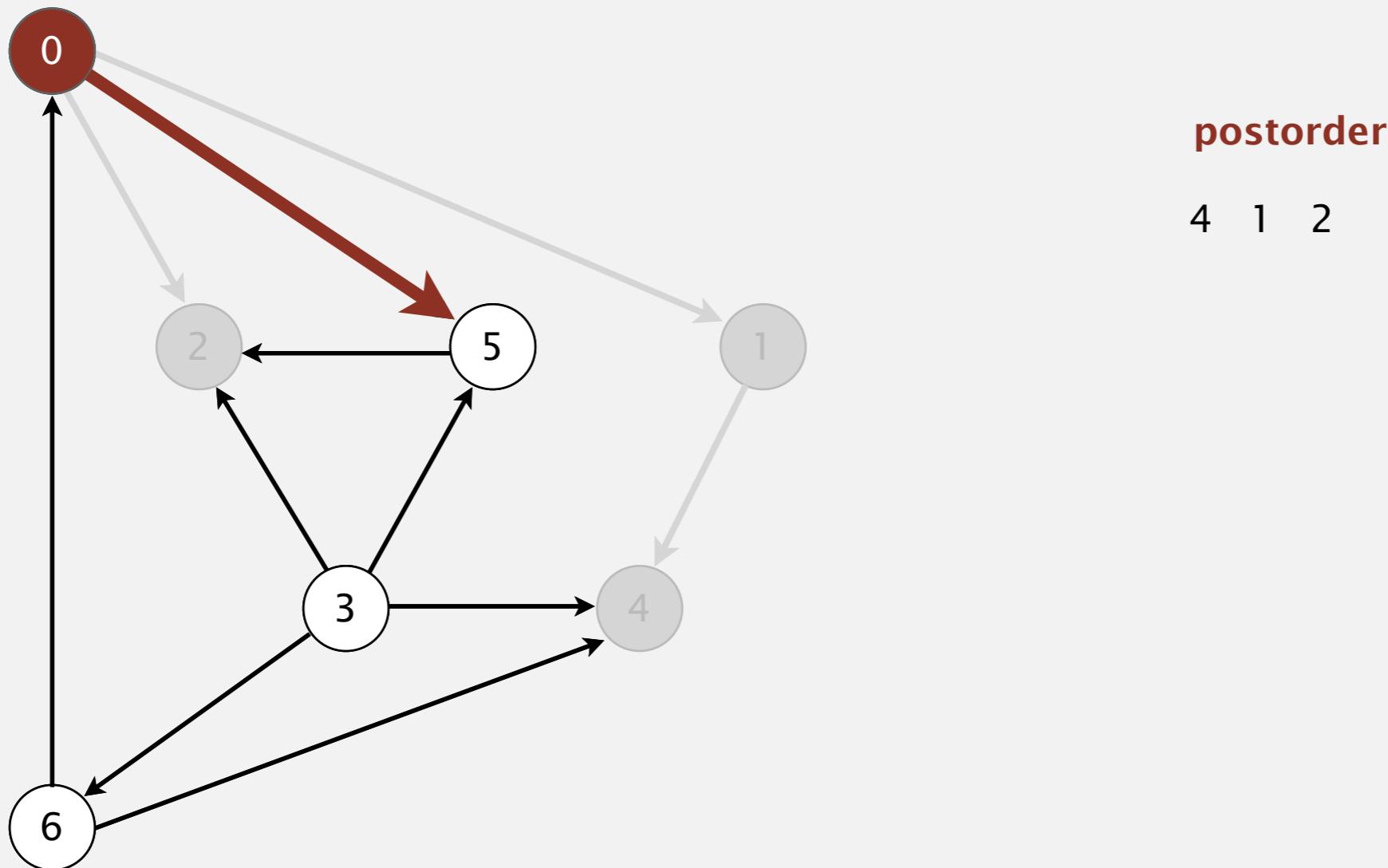
- Run depth-first search.
- Return vertices in reverse postorder.



2 done

Topological sort algorithm

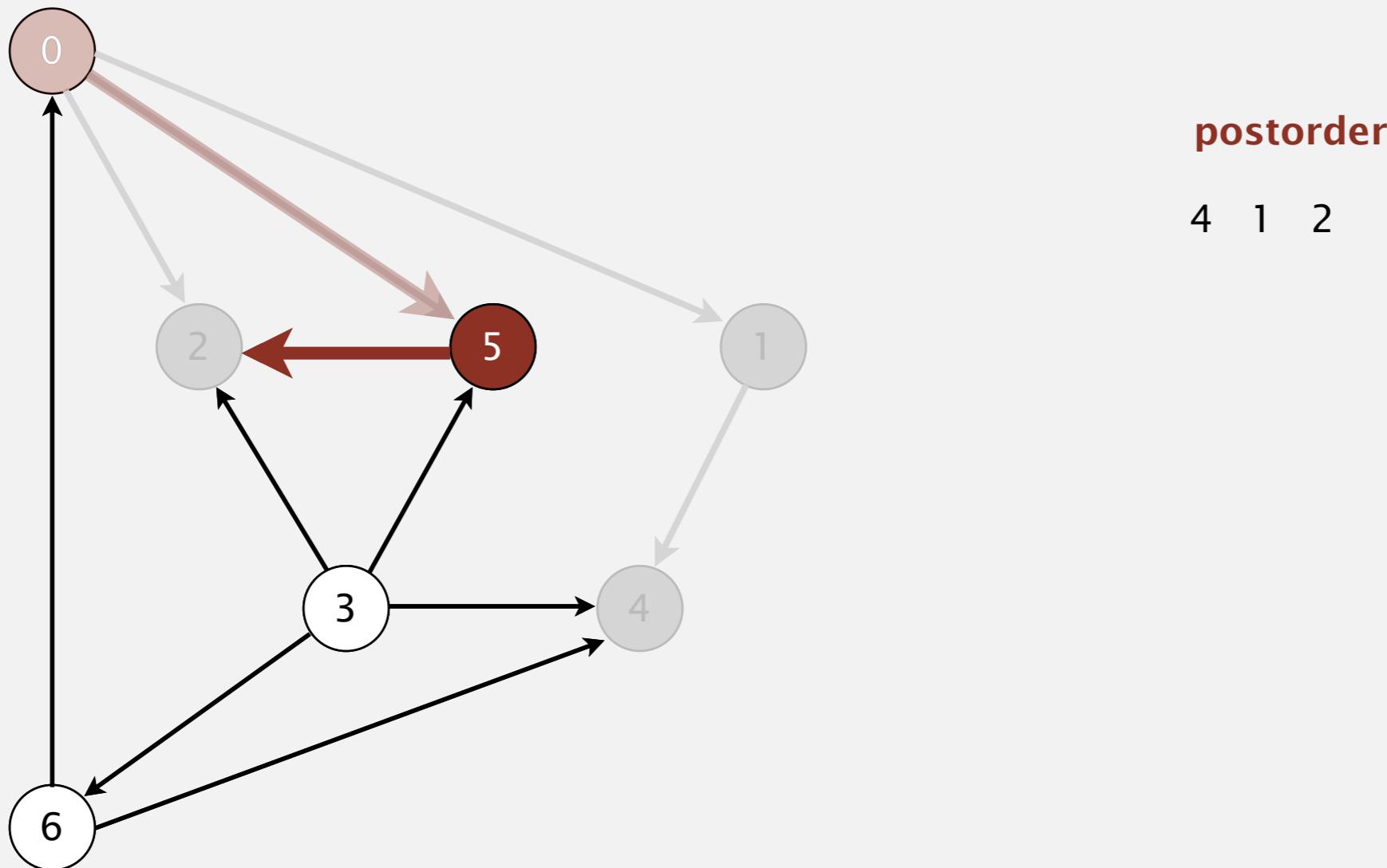
- Run depth-first search.
- Return vertices in reverse postorder.



visit 0: check 1, check 2, and check 5

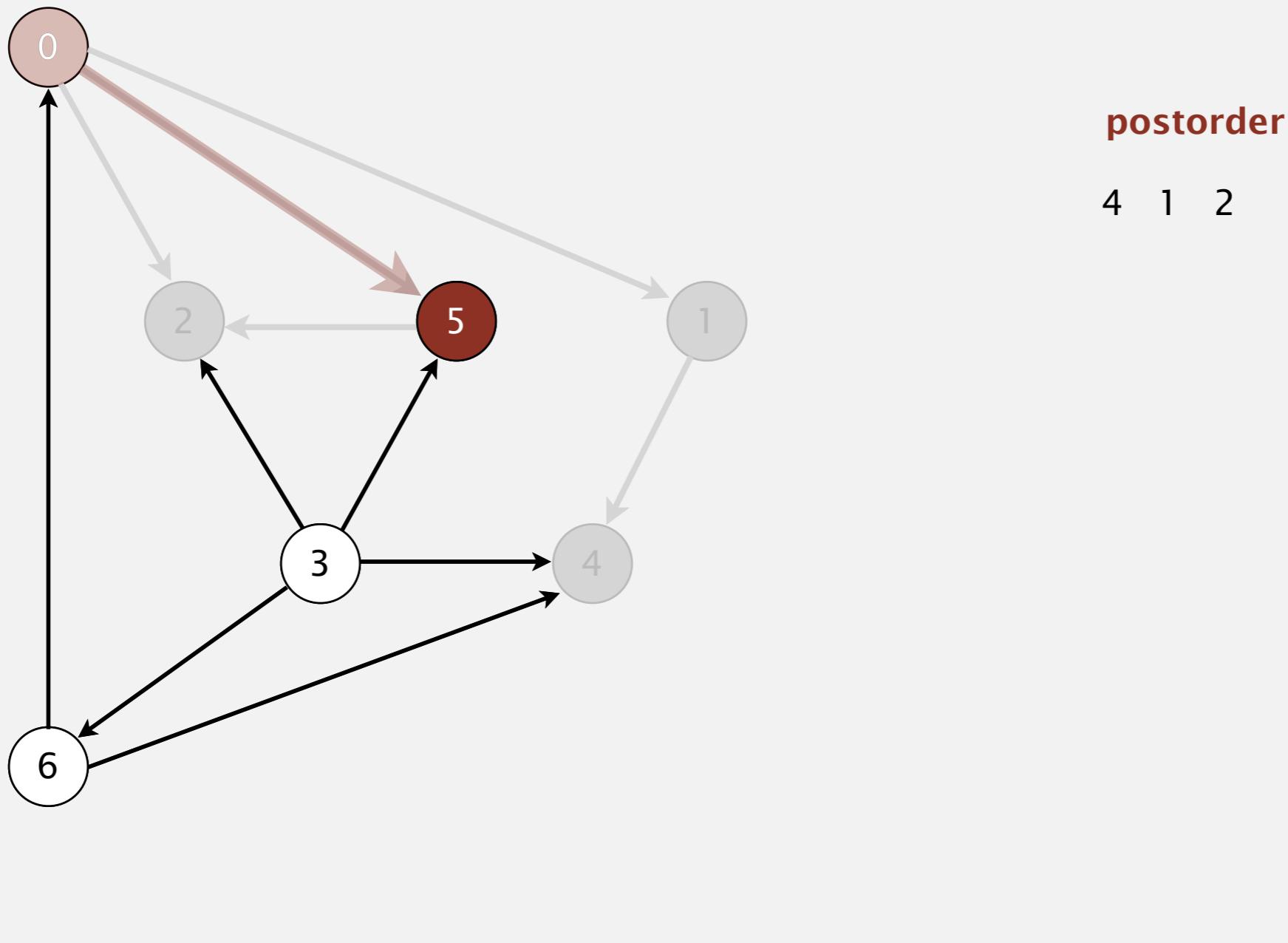
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



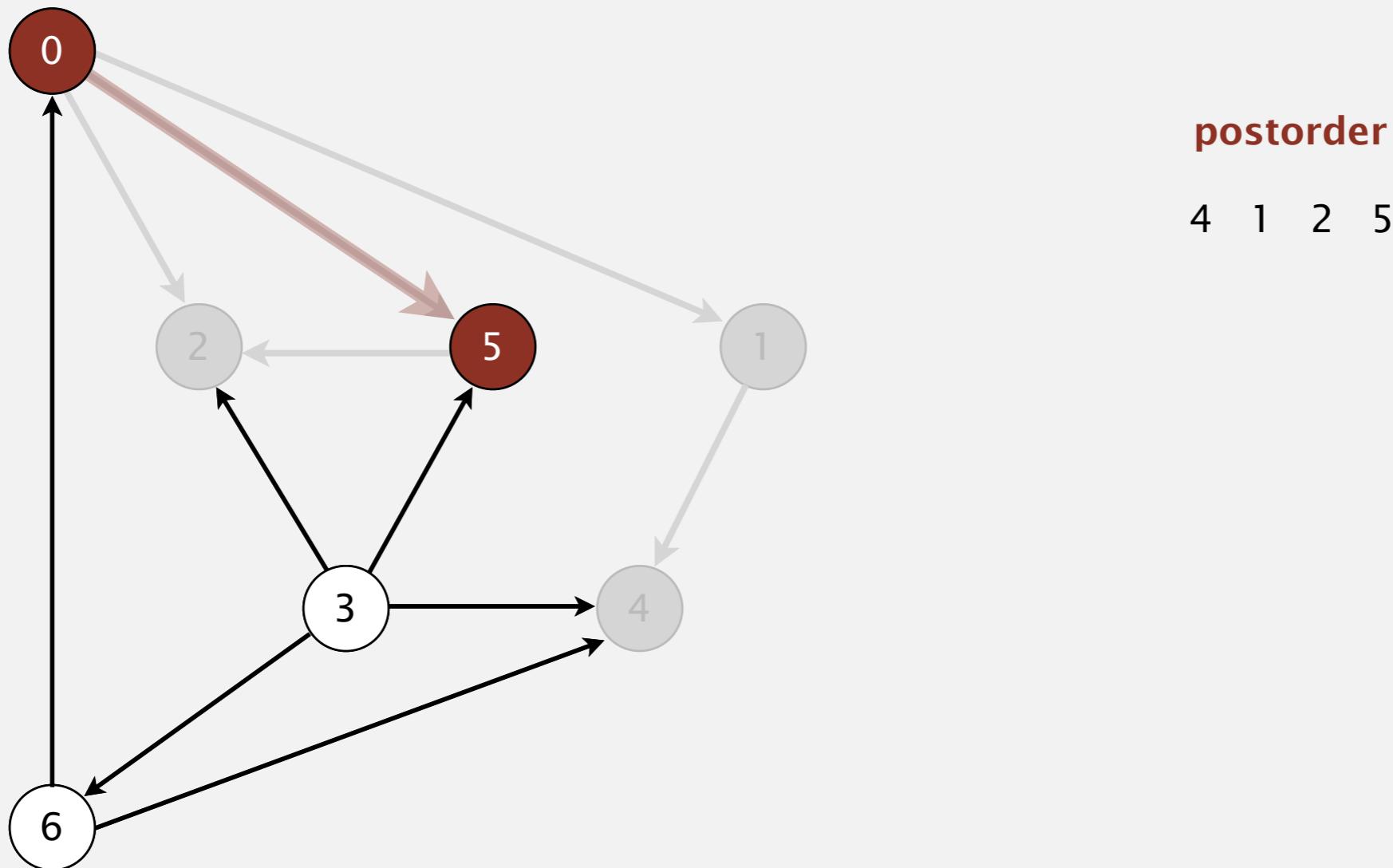
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



Topological sort algorithm

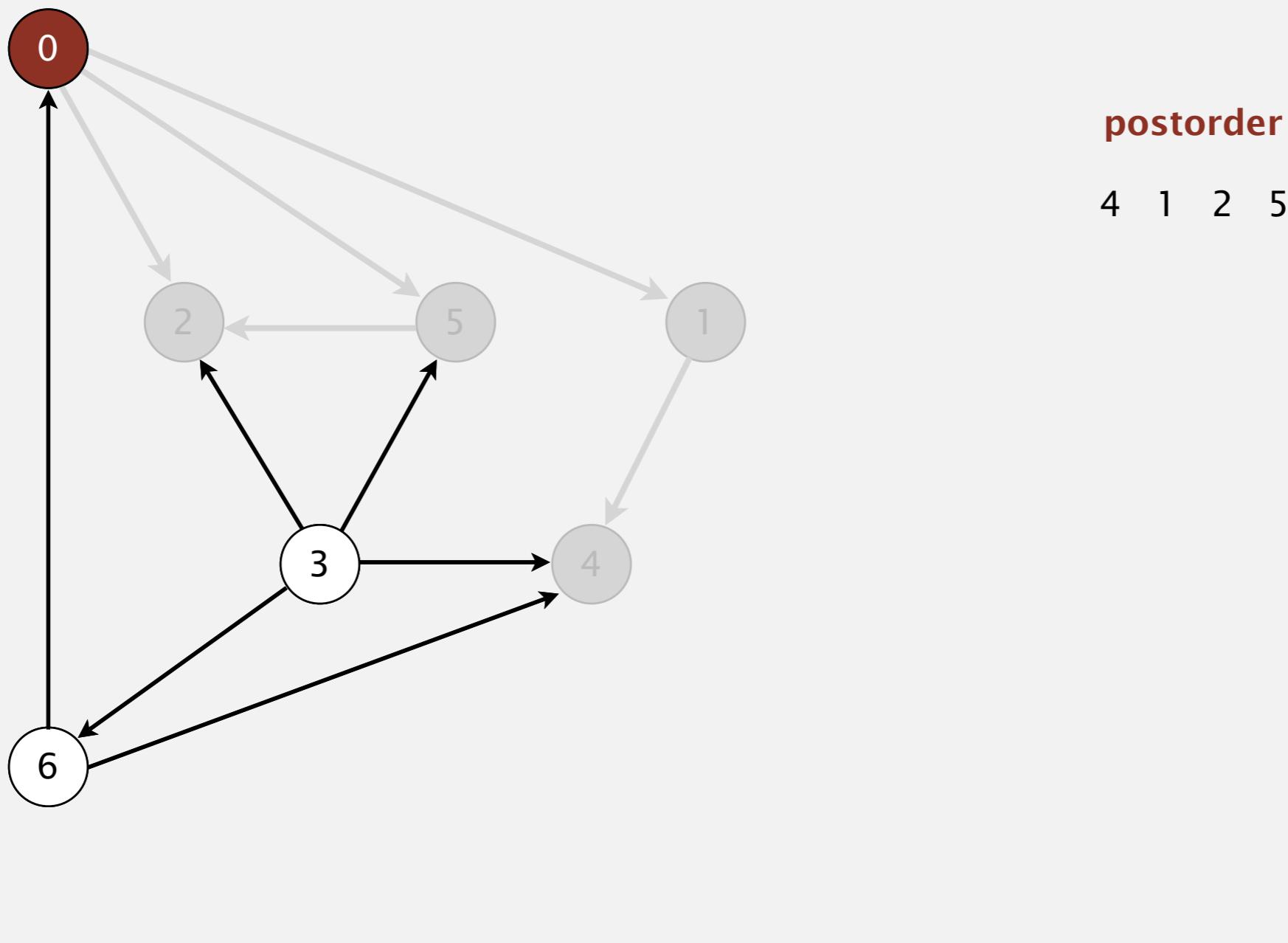
- Run depth-first search.
- Return vertices in reverse postorder.



5 done

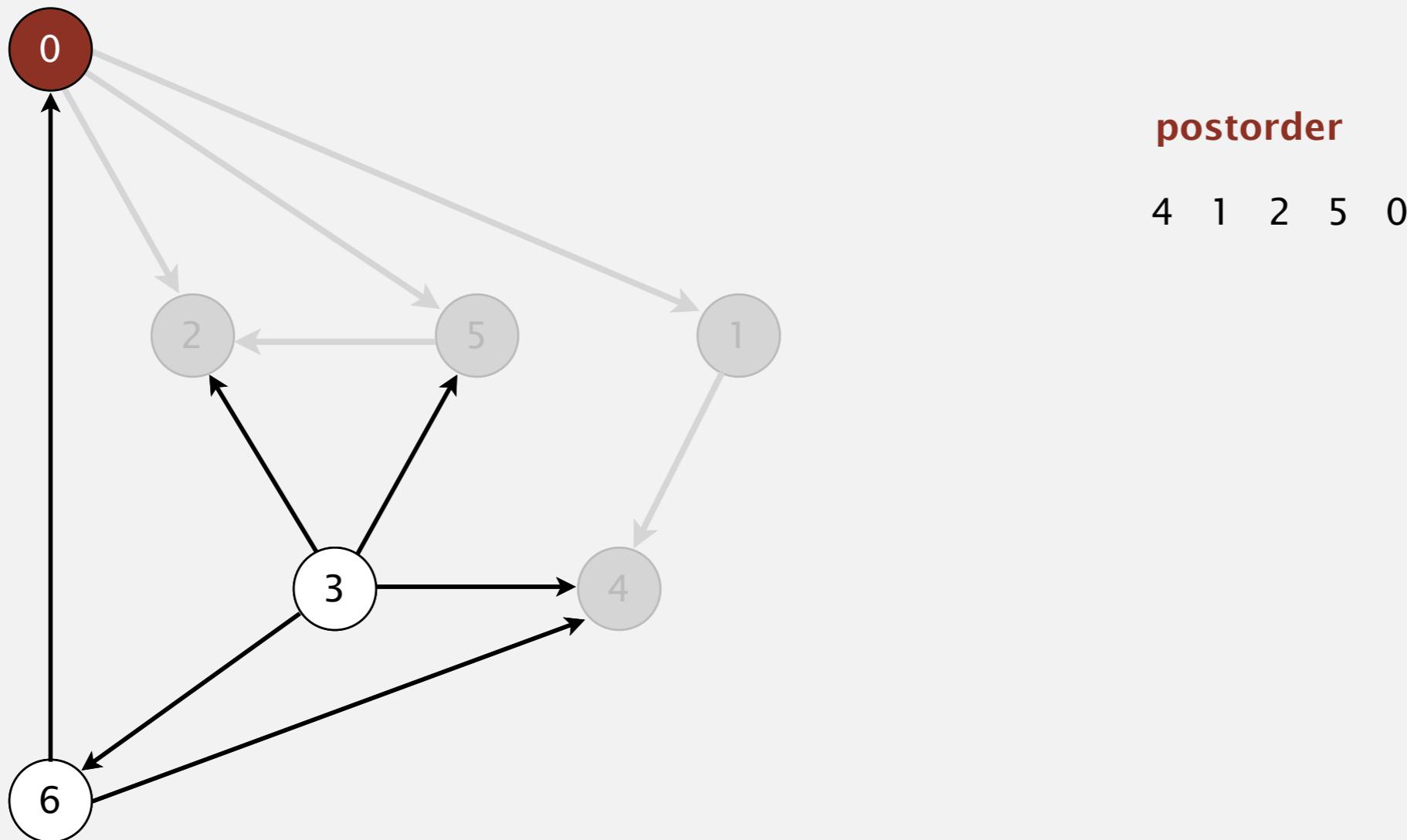
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



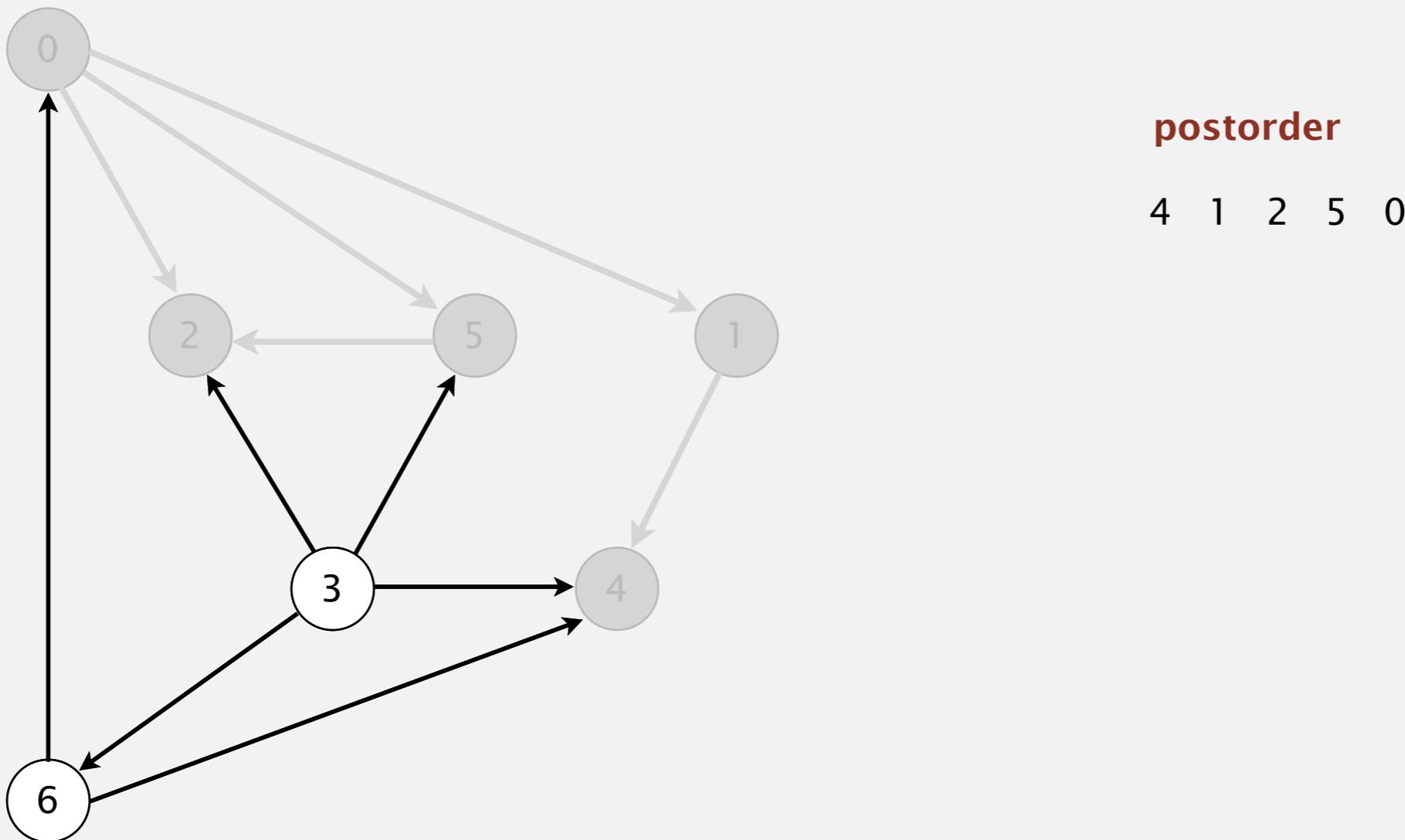
postorder

4 1 2 5 0

0 done

Topological sort algorithm

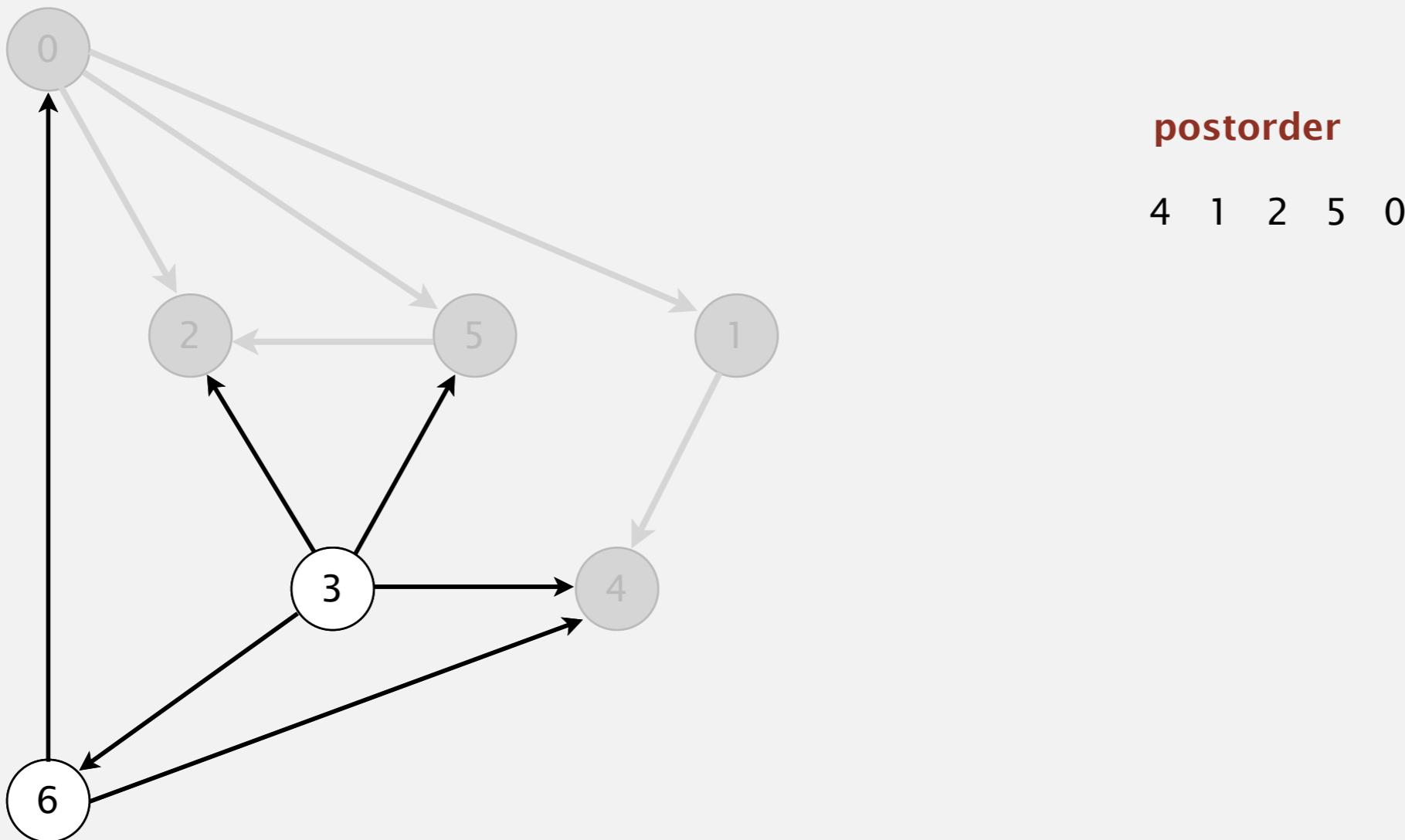
- Run depth-first search.
- Return vertices in reverse postorder.



check 1

Topological sort algorithm

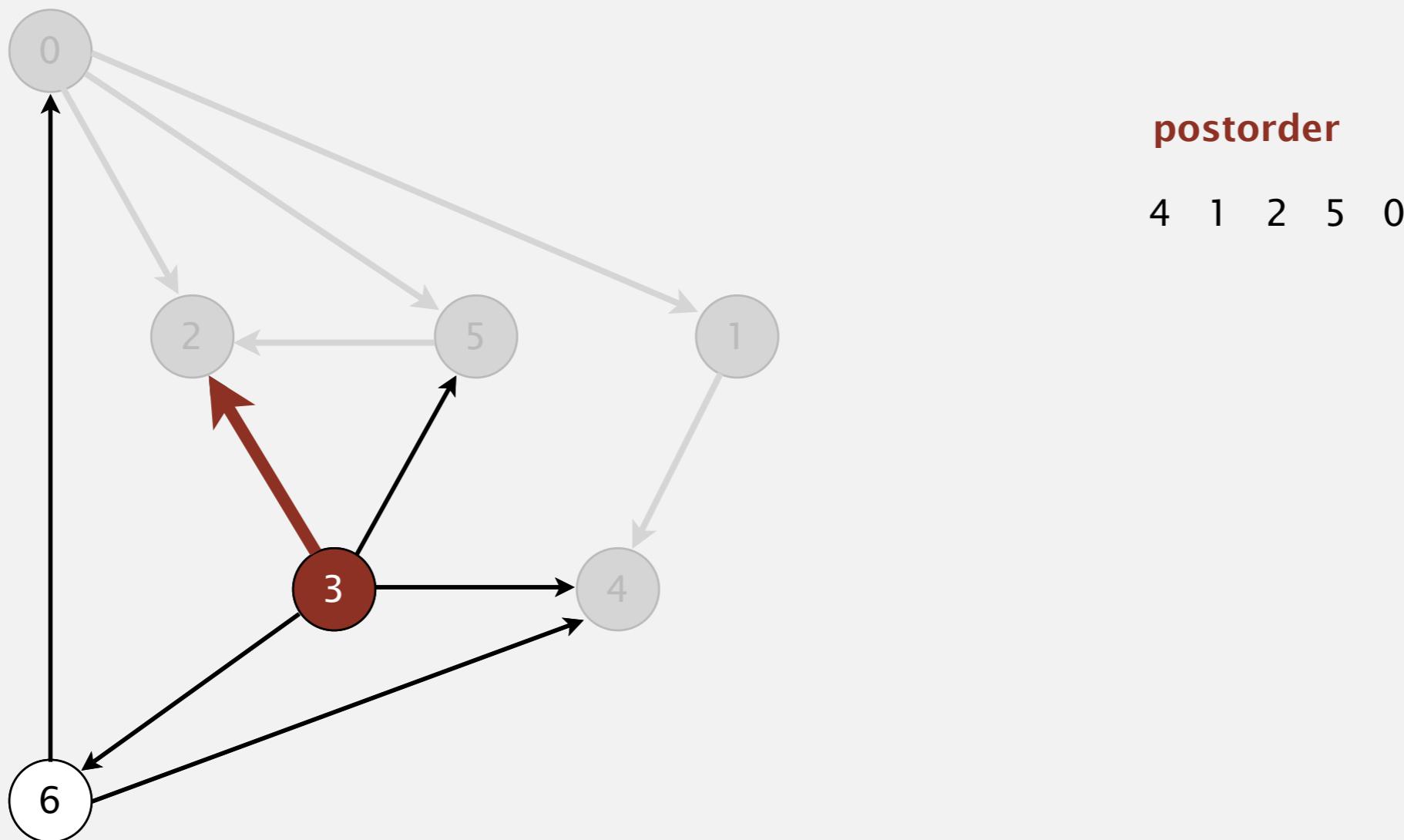
- Run depth-first search.
- Return vertices in reverse postorder.



check 2

Topological sort algorithm

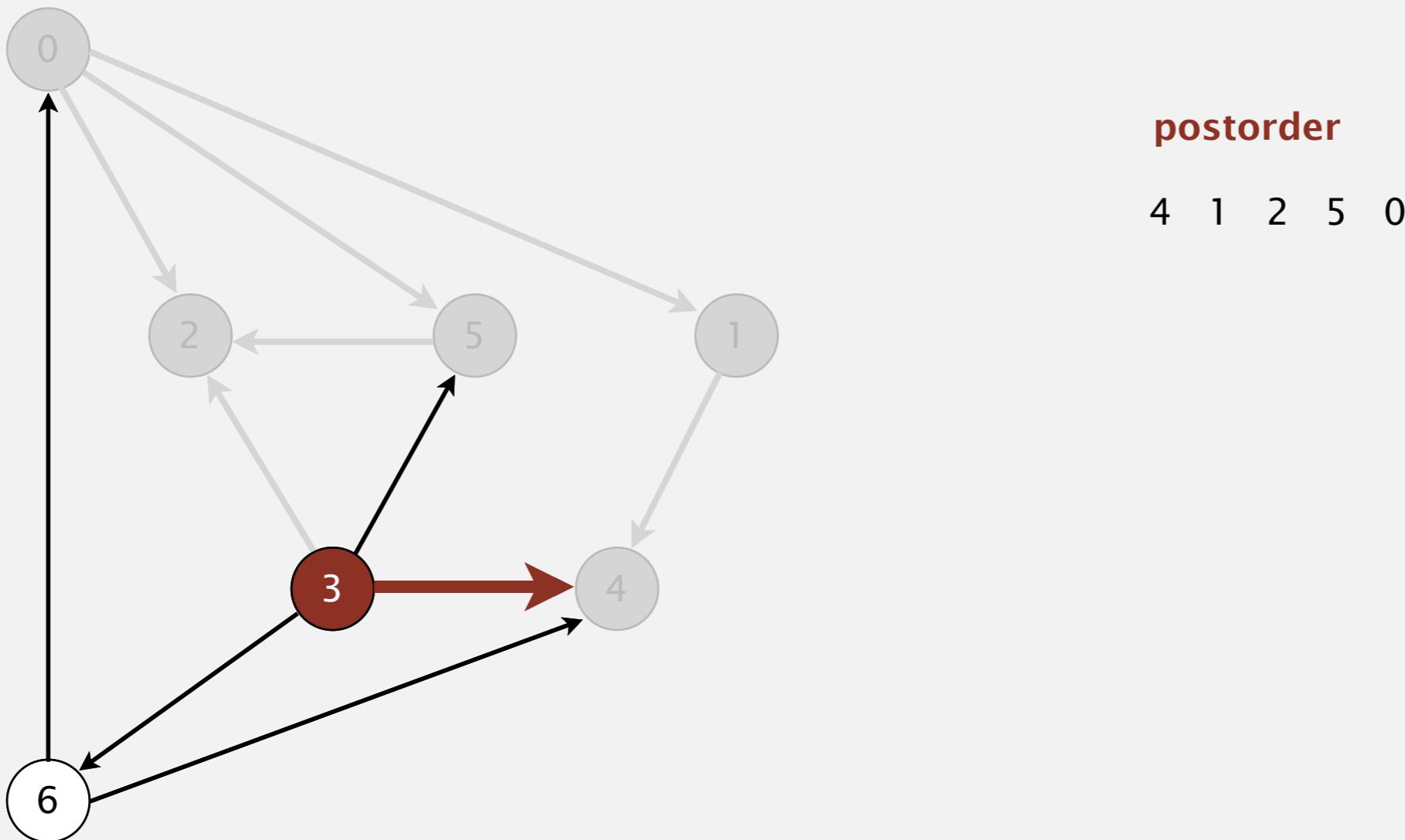
- Run depth-first search.
- Return vertices in reverse postorder.



visit 3: check 2, check 4, check 5, and check 6

Topological sort algorithm

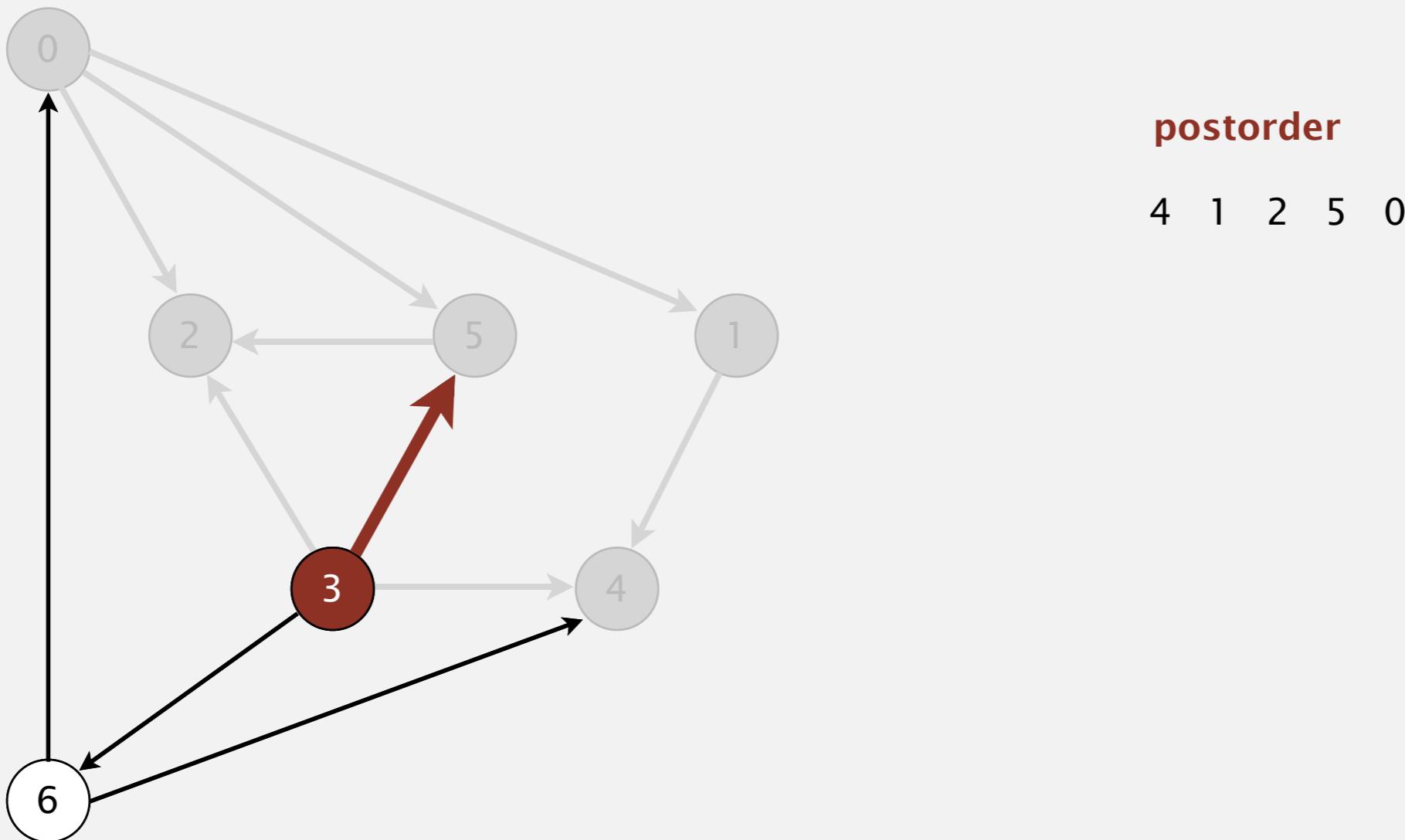
- Run depth-first search.
- Return vertices in reverse postorder.



visit 3: check 2, check 4, check 5, and check 6

Topological sort algorithm

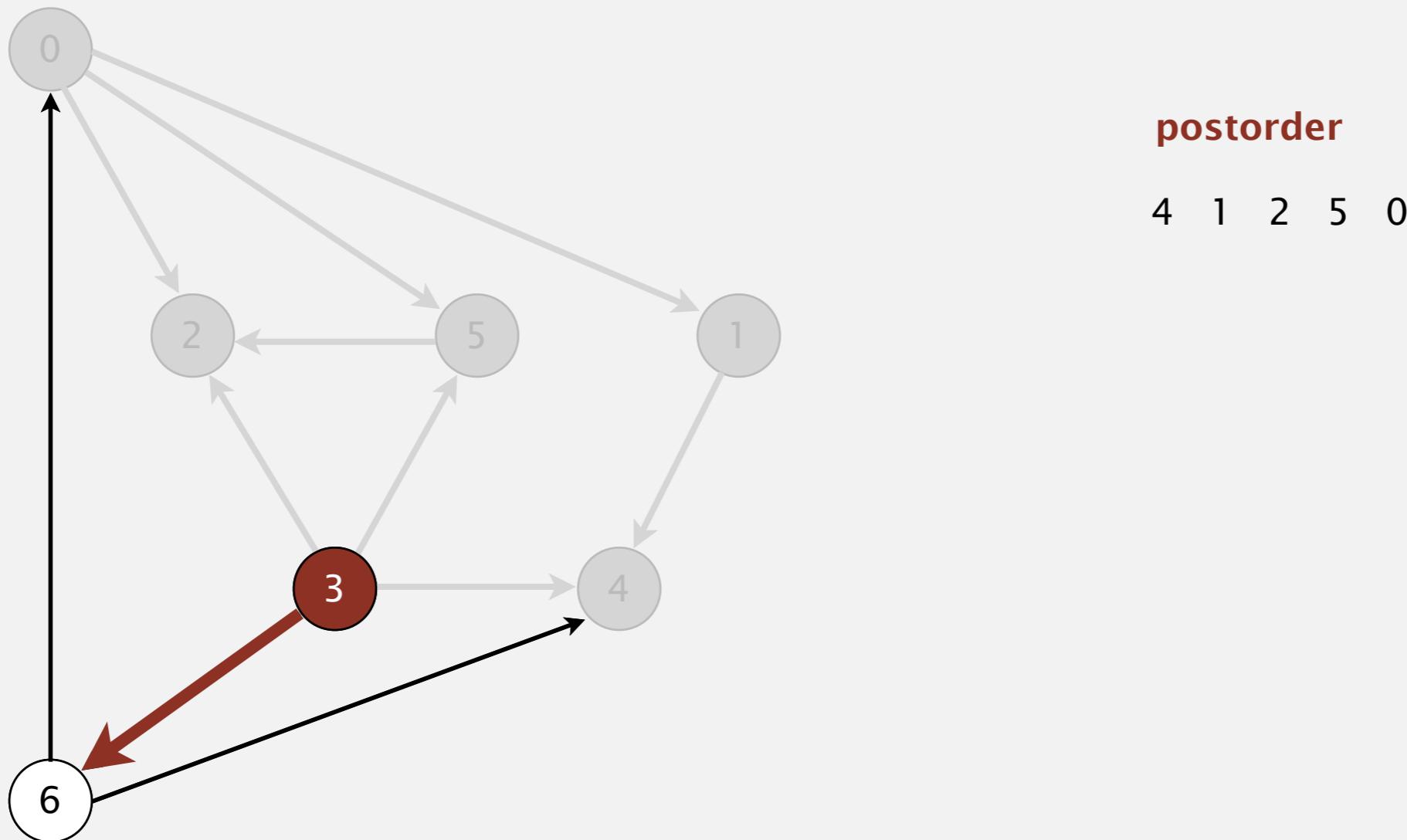
- Run depth-first search.
- Return vertices in reverse postorder.



visit 3: check 2, check 4, **check 5**, and check 6

Topological sort algorithm

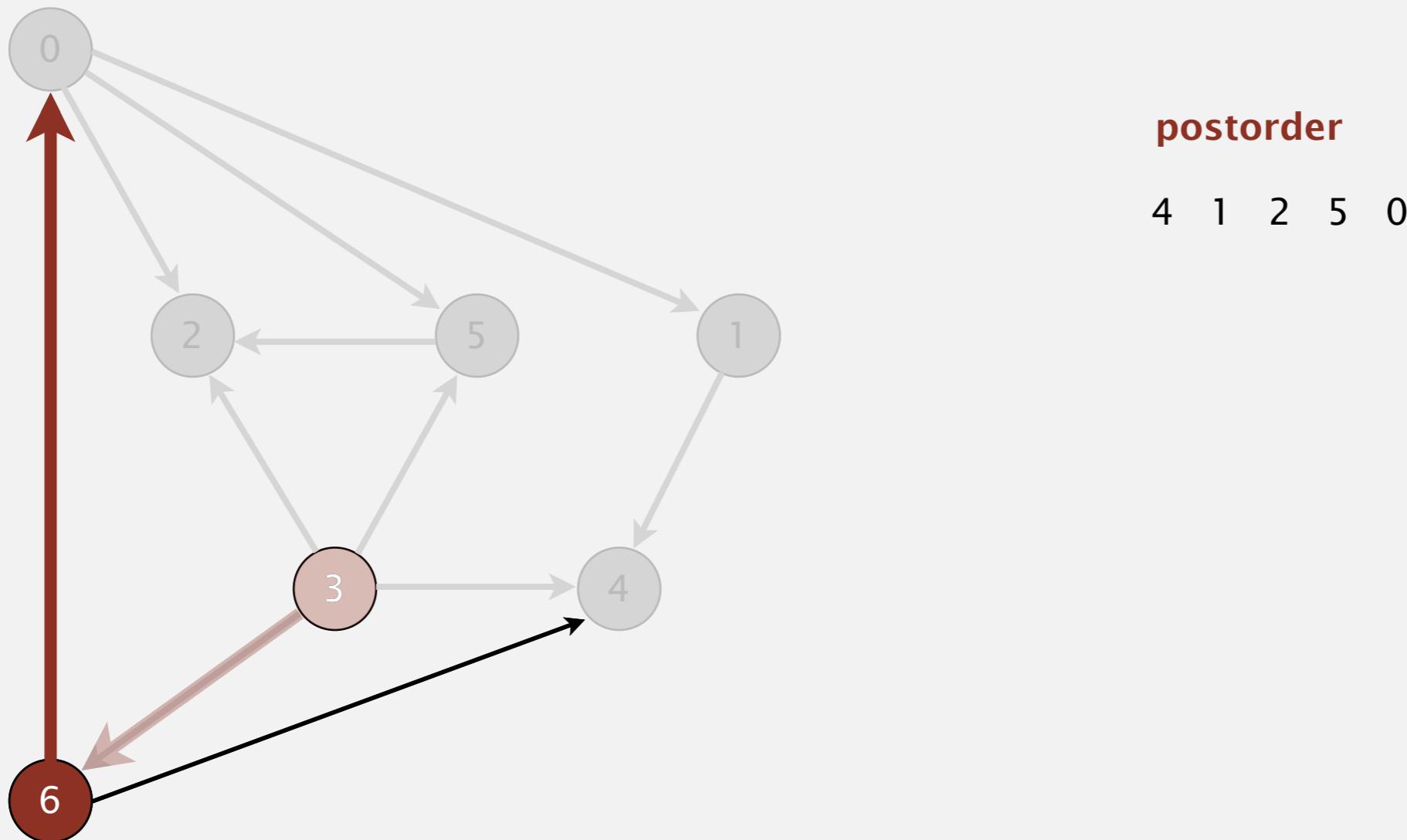
- Run depth-first search.
- Return vertices in reverse postorder.



visit 3: check 2, check 4, check 5, and check 6

Topological sort algorithm

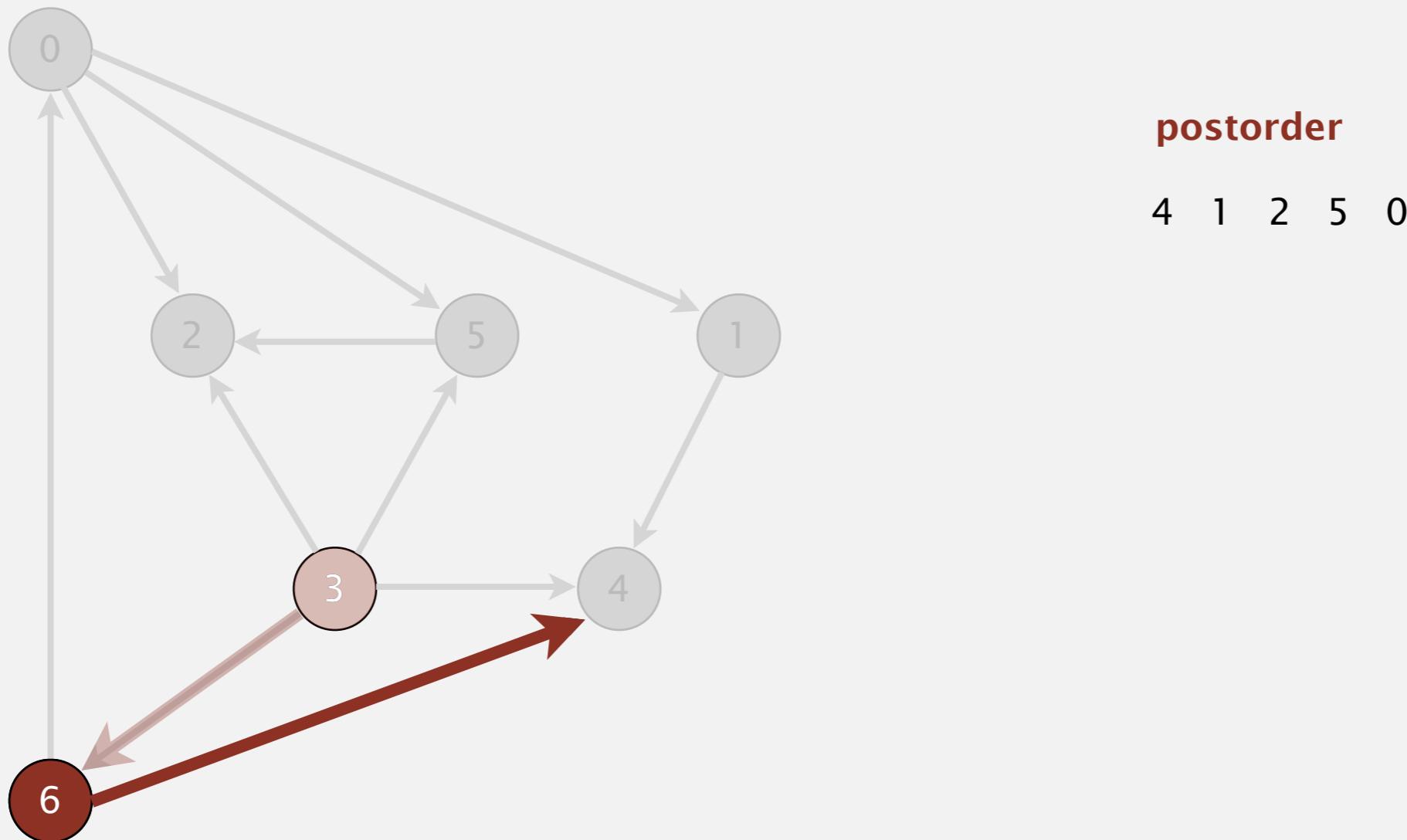
- Run depth-first search.
- Return vertices in reverse postorder.



visit 6: check 0 and check 4

Topological sort algorithm

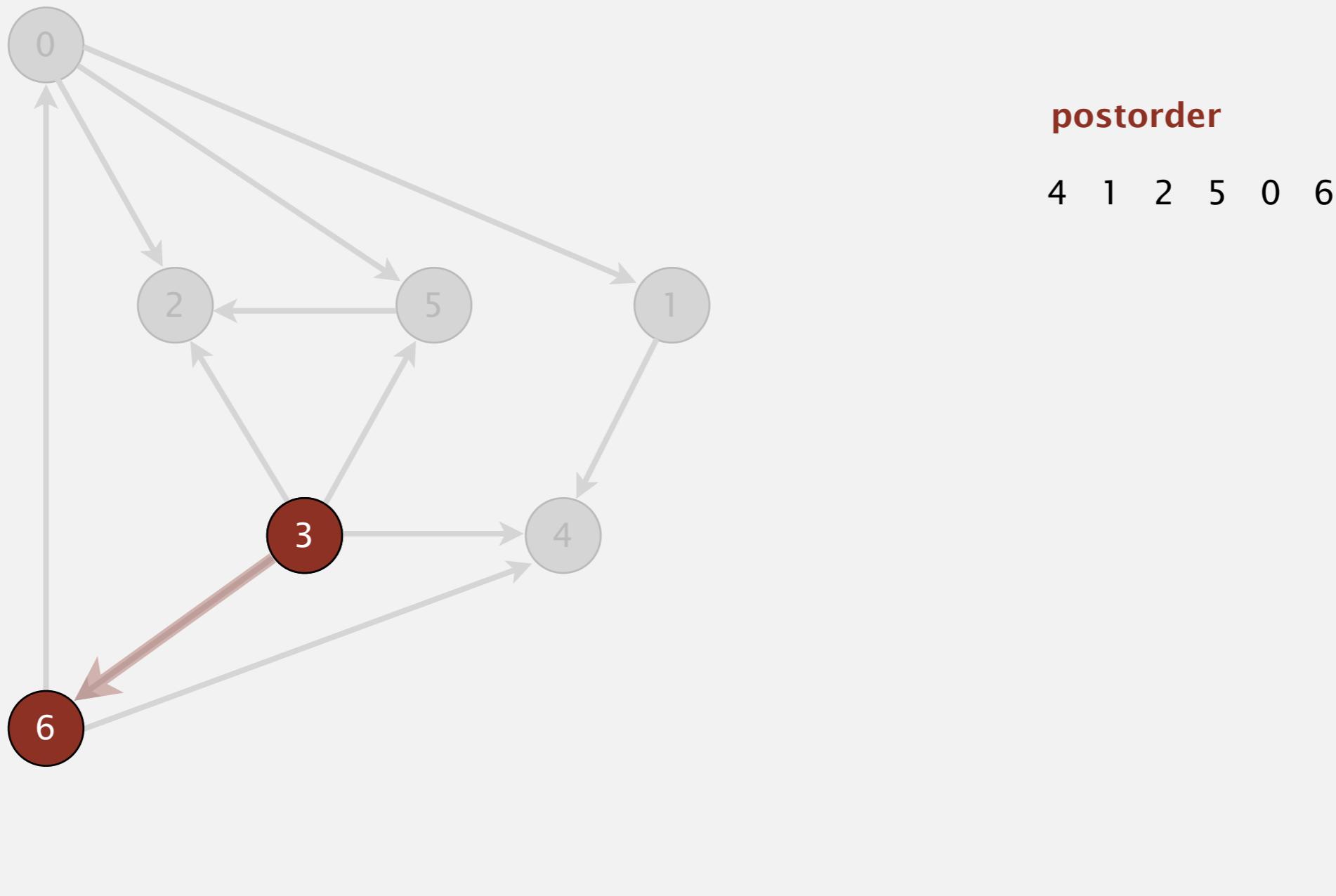
- Run depth-first search.
- Return vertices in reverse postorder.



visit 6: check 0 and check 4

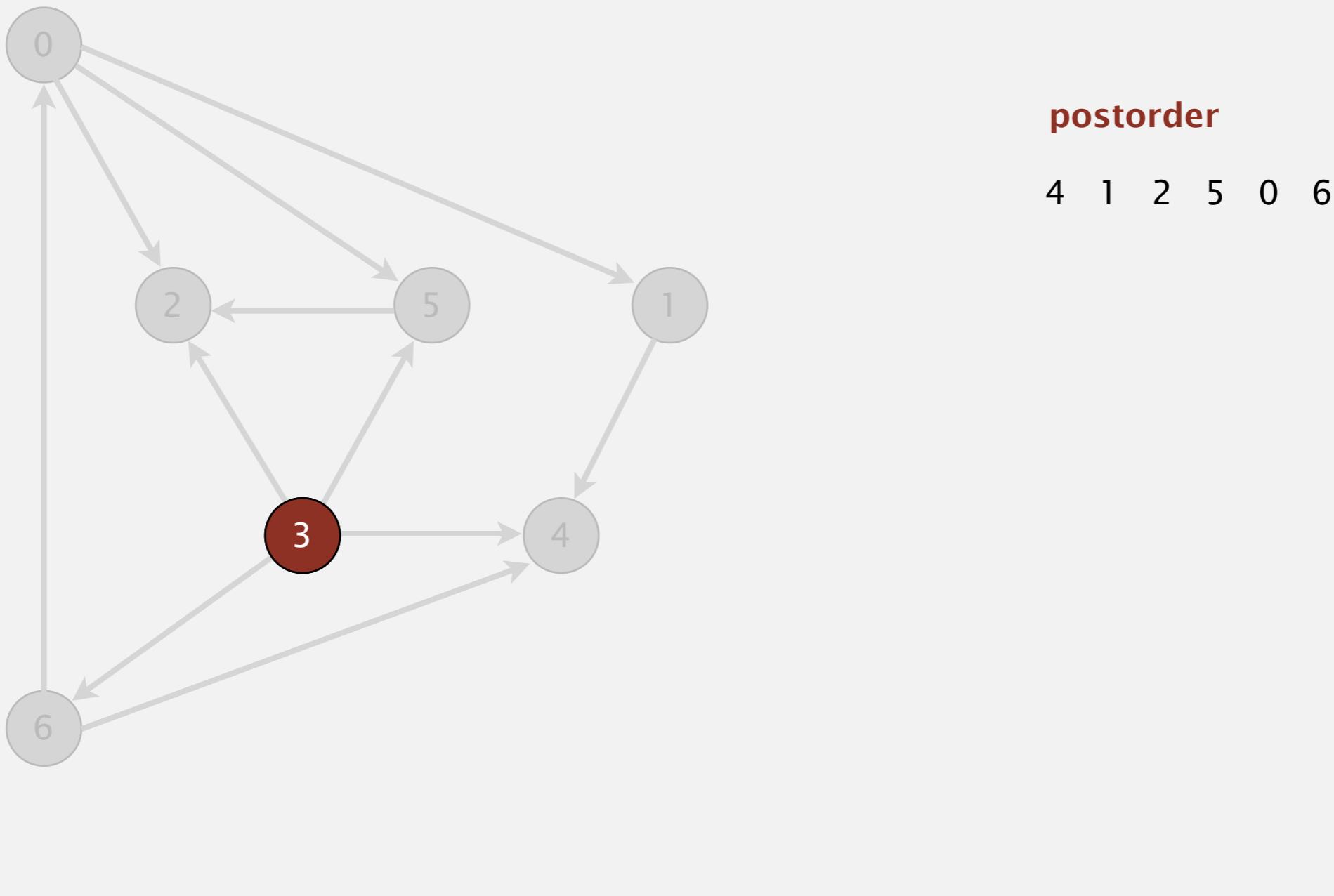
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



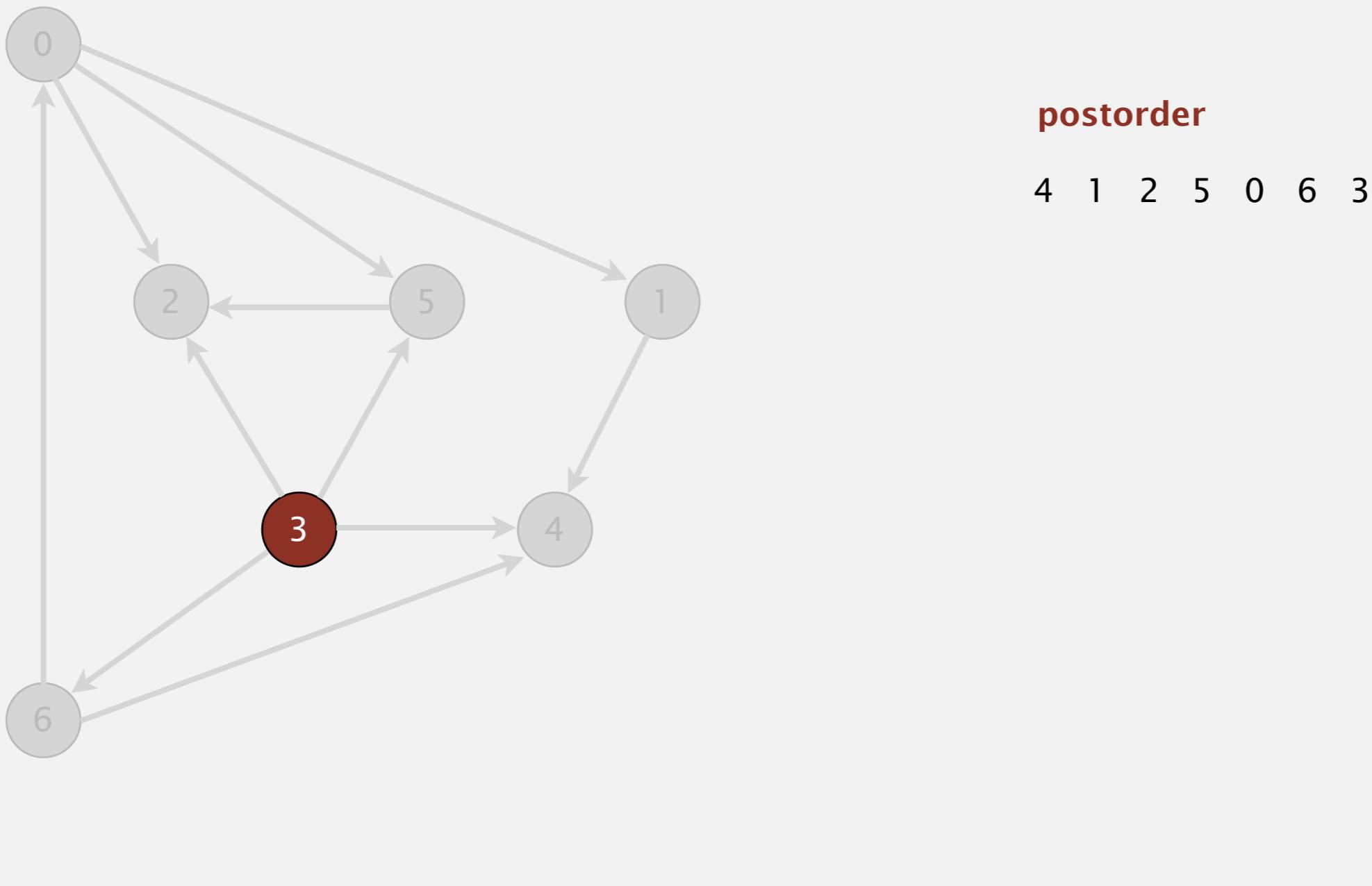
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



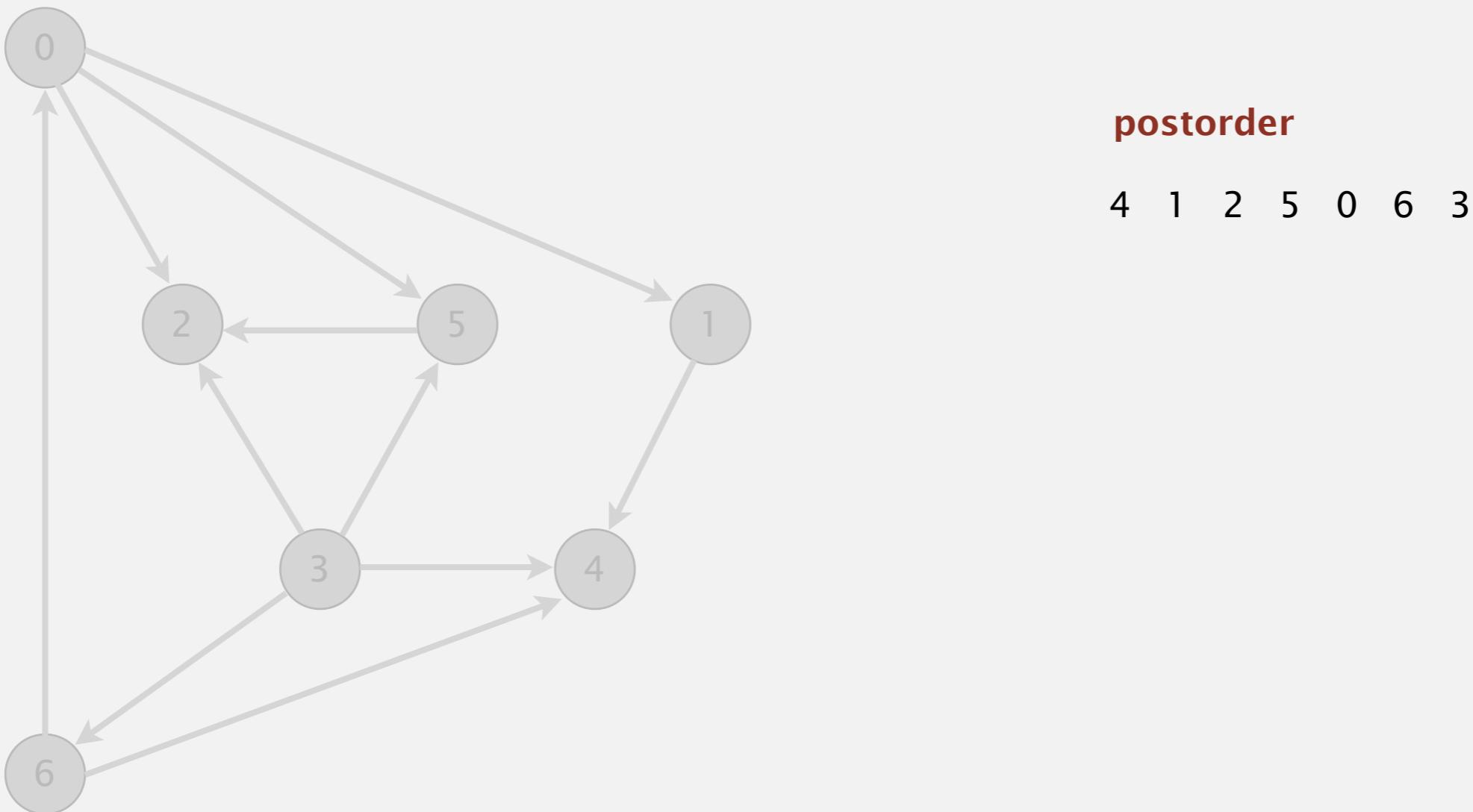
Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



Topological sort algorithm

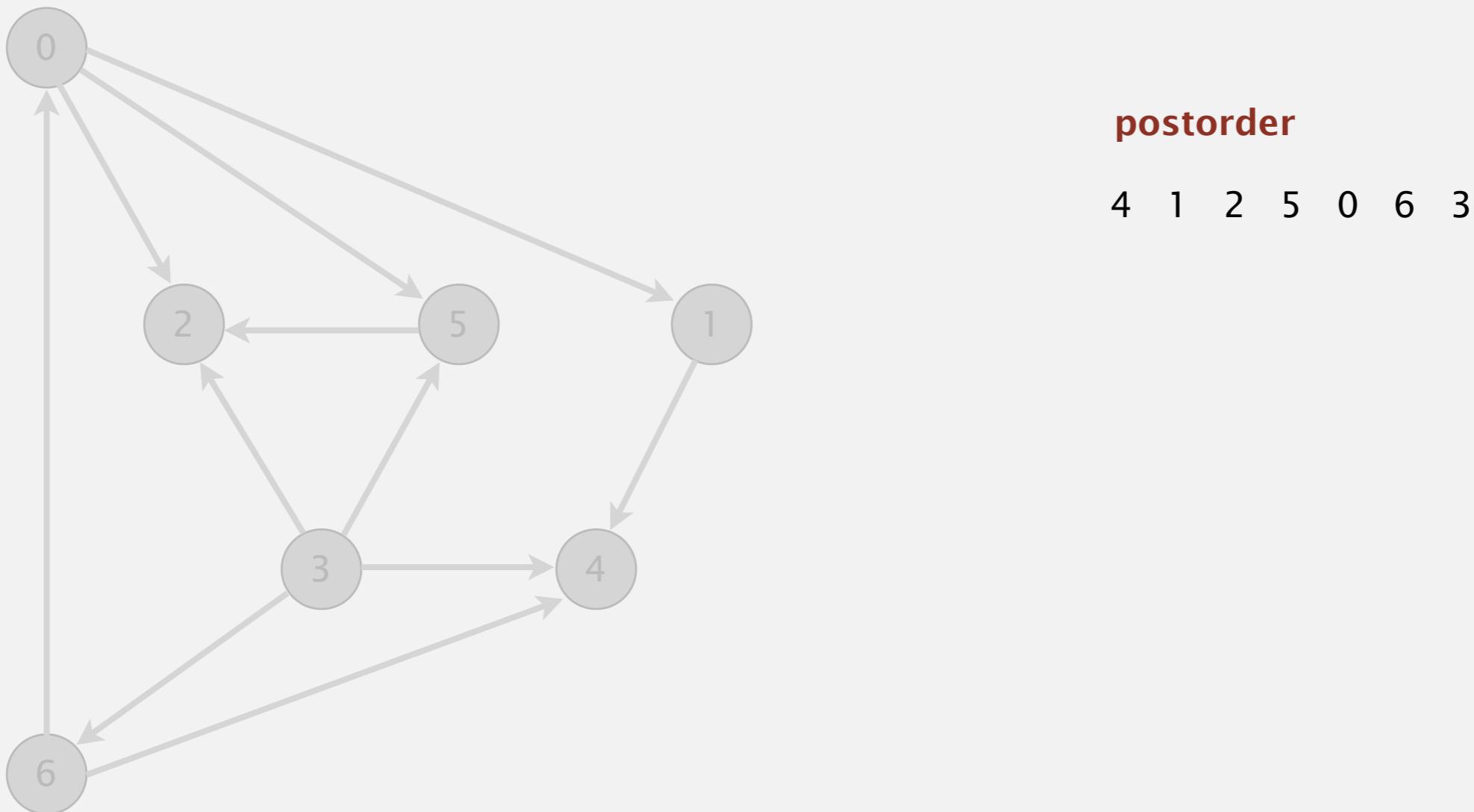
- Run depth-first search.
- Return vertices in reverse postorder.



check 4

Topological sort algorithm

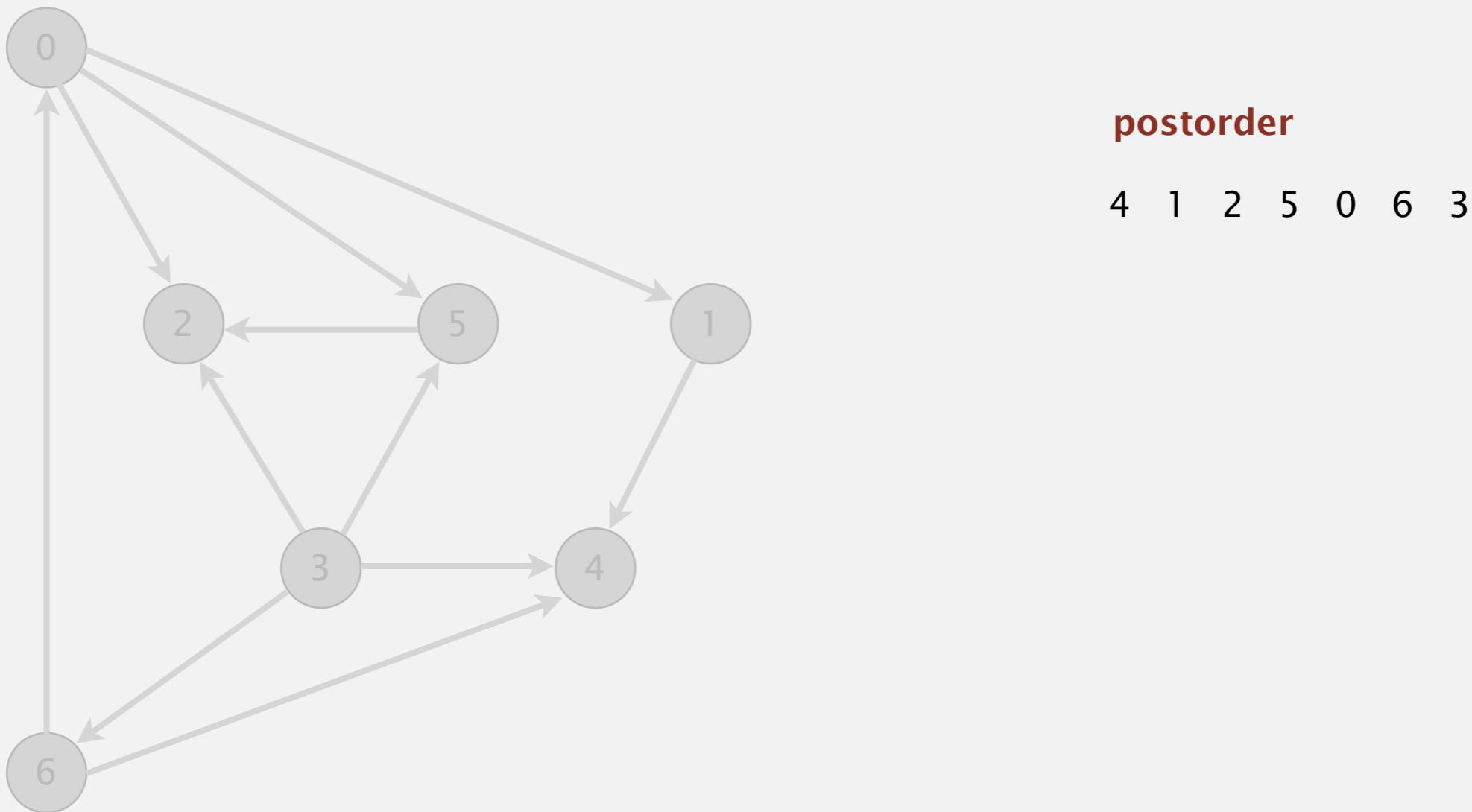
- Run depth-first search.
- Return vertices in reverse postorder.



check 5

Topological sort algorithm

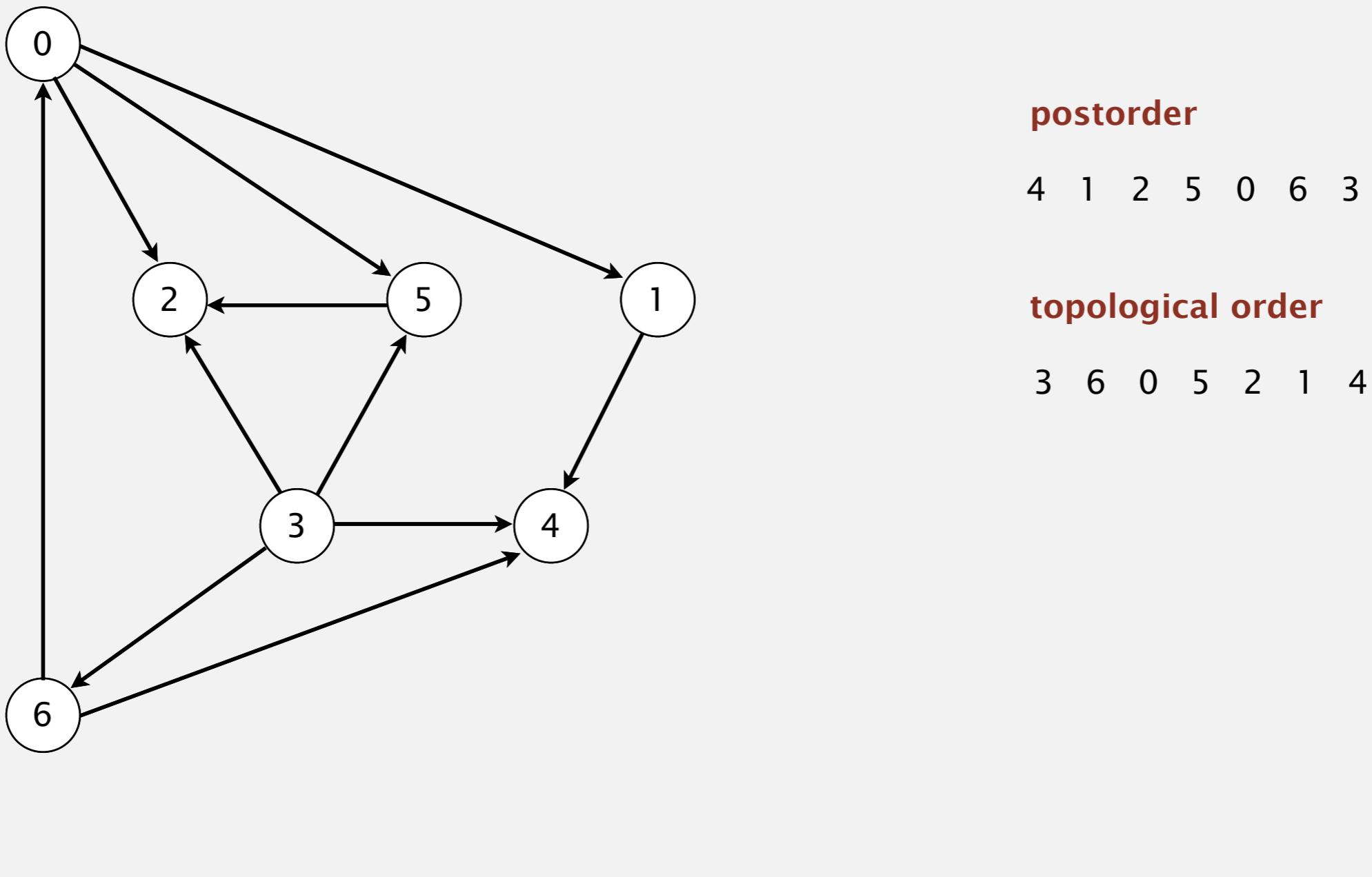
- Run depth-first search.
- Return vertices in reverse postorder.



check 6

Topological sort algorithm

- Run depth-first search.
- Return vertices in reverse postorder.



Depth-first search order

```
public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePost;

    public DepthFirstOrder(Digraph G)
    {
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        reversePost.push(v);
    }

    public Iterable<Integer> reversePost()
    { return reversePost; }
}
```

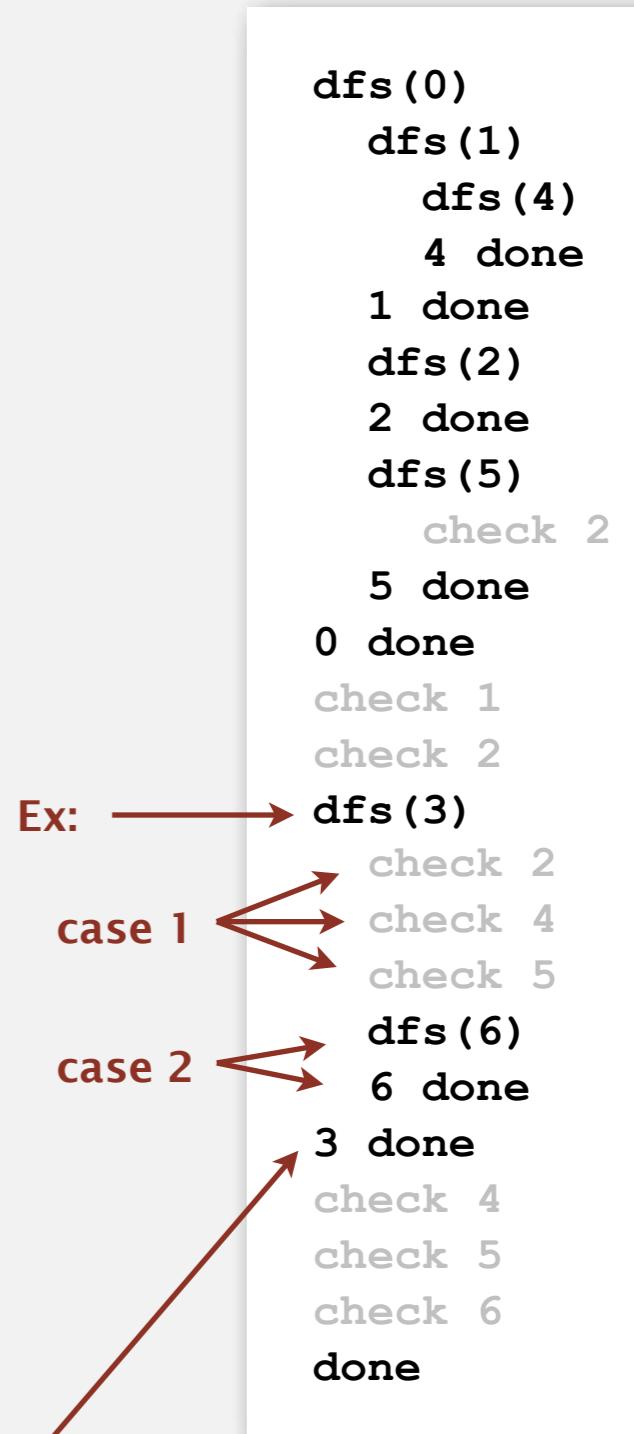
returns all vertices in
“reverse DFS postorder”

Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge $v \rightarrow w$. When $\text{dfs}(v)$ is called:

- Case 1: $\text{dfs}(w)$ has already been called and returned.
Thus, w was done before v .
- Case 2: $\text{dfs}(w)$ has not yet been called.
 $\text{dfs}(w)$ will get called directly or indirectly
by $\text{dfs}(v)$ and will finish before $\text{dfs}(v)$.
Thus, w will be done before v .
- Case 3: $\text{dfs}(w)$ has already been called,
but has not yet returned.
Can't happen in a DAG: function call stack contains
path from w to v , so $v \rightarrow w$ would complete a cycle.



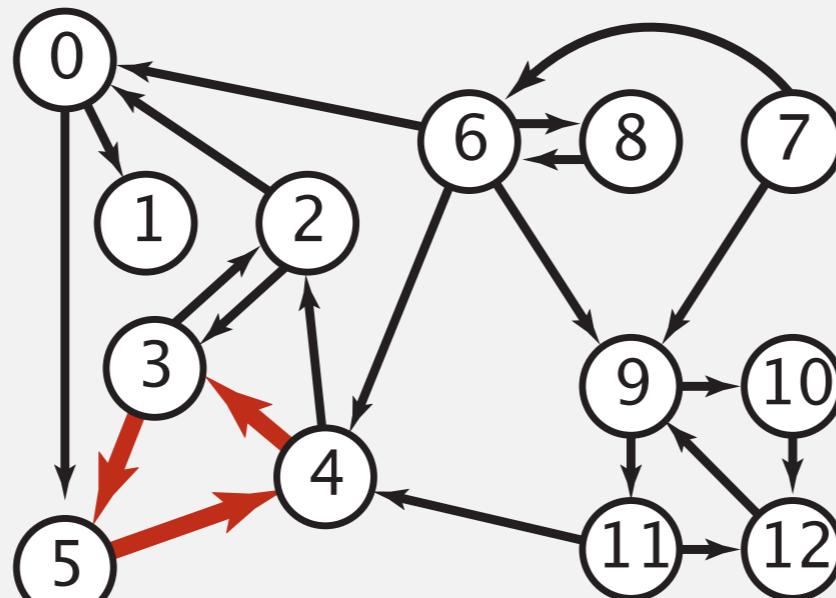
all vertices pointing from 3 are done before 3 is done,
so they appear after 3 in topological order

Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



a digraph with a directed cycle

Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else? See textbook.

Directed cycle detection application: precedence scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

<http://xkcd.com/754>

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```
public class A extends B
{
    ...
}
```

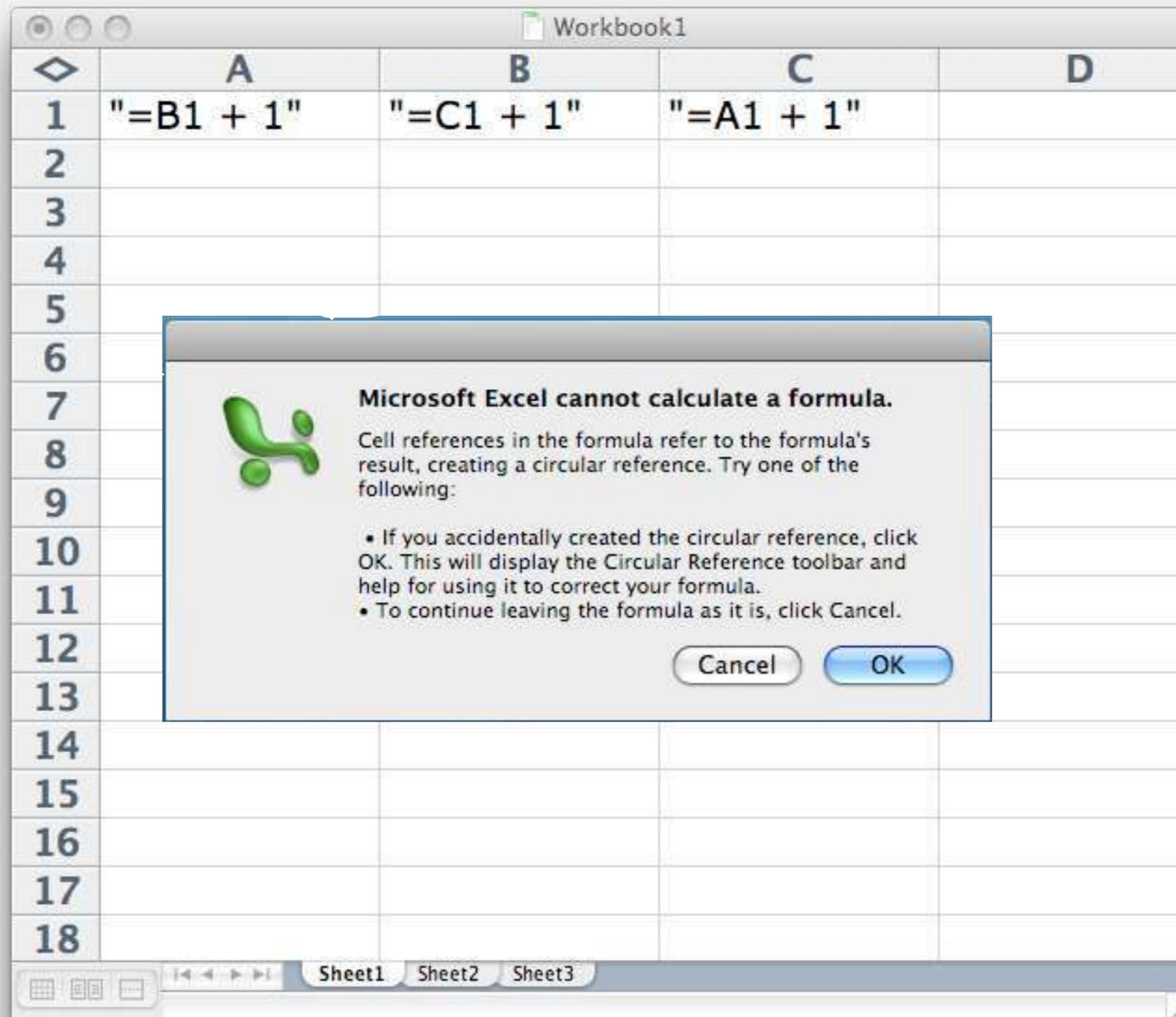
```
public class B extends C
{
    ...
}
```

```
public class C extends A
{
    ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
^
1 error
```

Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)



Directed cycle detection applications

- Causalities.
- Email loops.
- Compilation units.
- Class inheritance.
- Course prerequisites.
- Deadlocking detection.
- Precedence scheduling.
- Temporal dependencies.
- Pipeline of computing jobs.
- Check for symbolic link loop.
- Evaluate formula in spreadsheet.

DIRECTED GRAPHS

- ▶ Digraph API
- ▶ Digraph search
- ▶ Topological sort
- ▶ Strong components

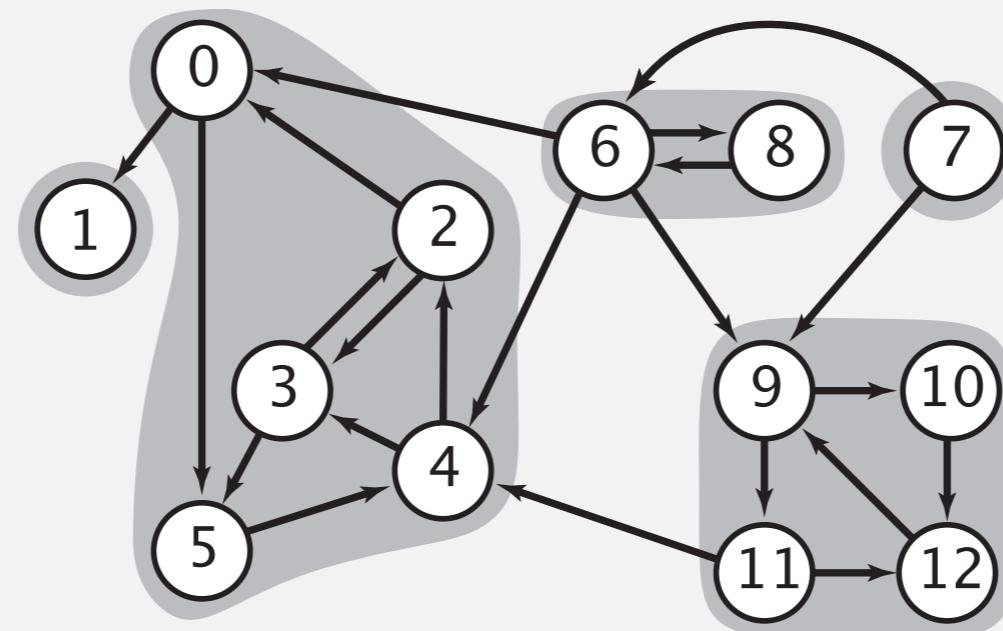
Strongly-connected components

Def. Vertices v and w are **strongly connected** if there is a directed path from v to w **and** a directed path from w to v .

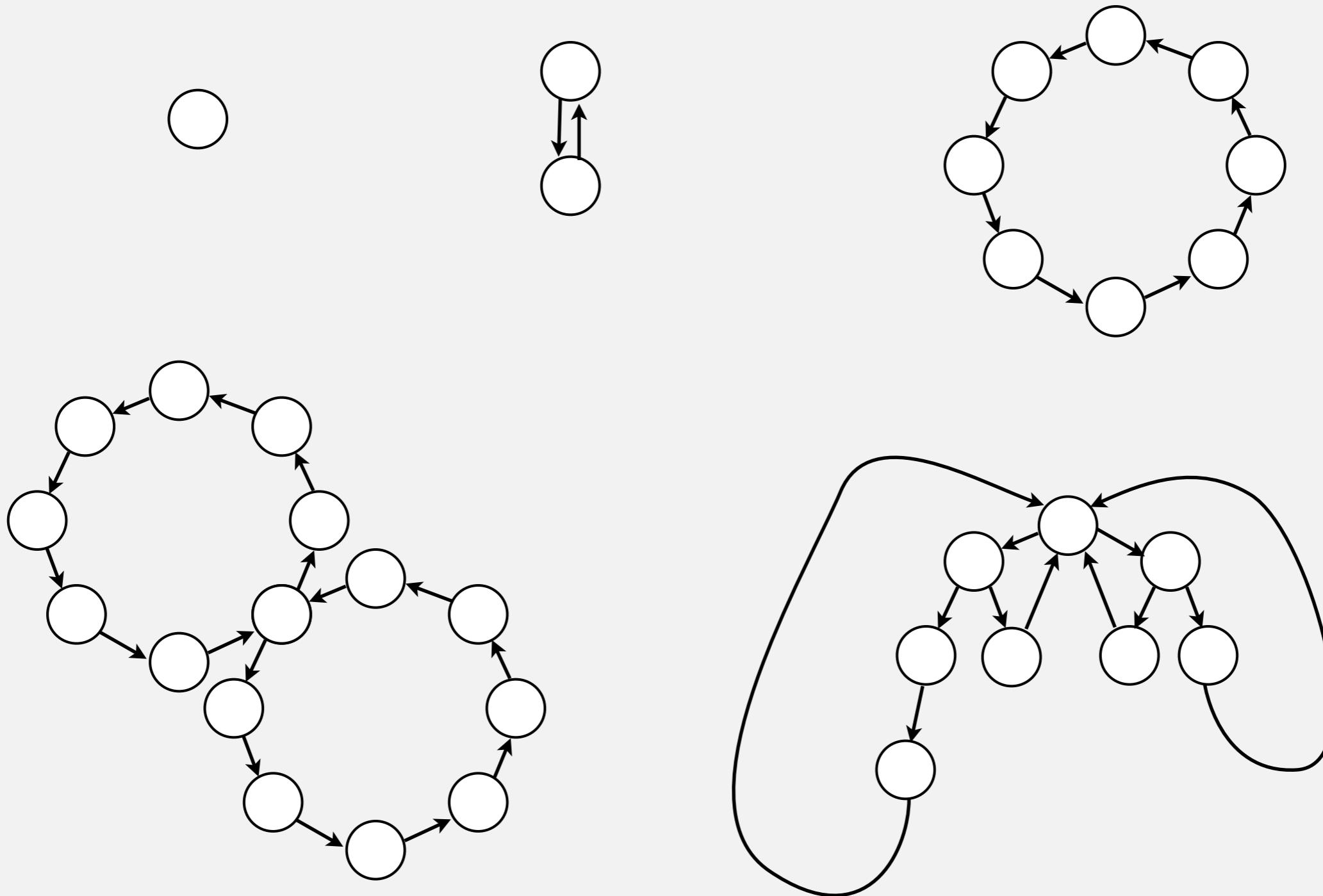
Key property. Strong connectivity is an **equivalence relation**:

- v is strongly connected to v .
- If v is strongly connected to w , then w is strongly connected to v .
- If v is strongly connected to w and w to x , then v is strongly connected to x .

Def. A **strong component** is a maximal subset of strongly-connected vertices.

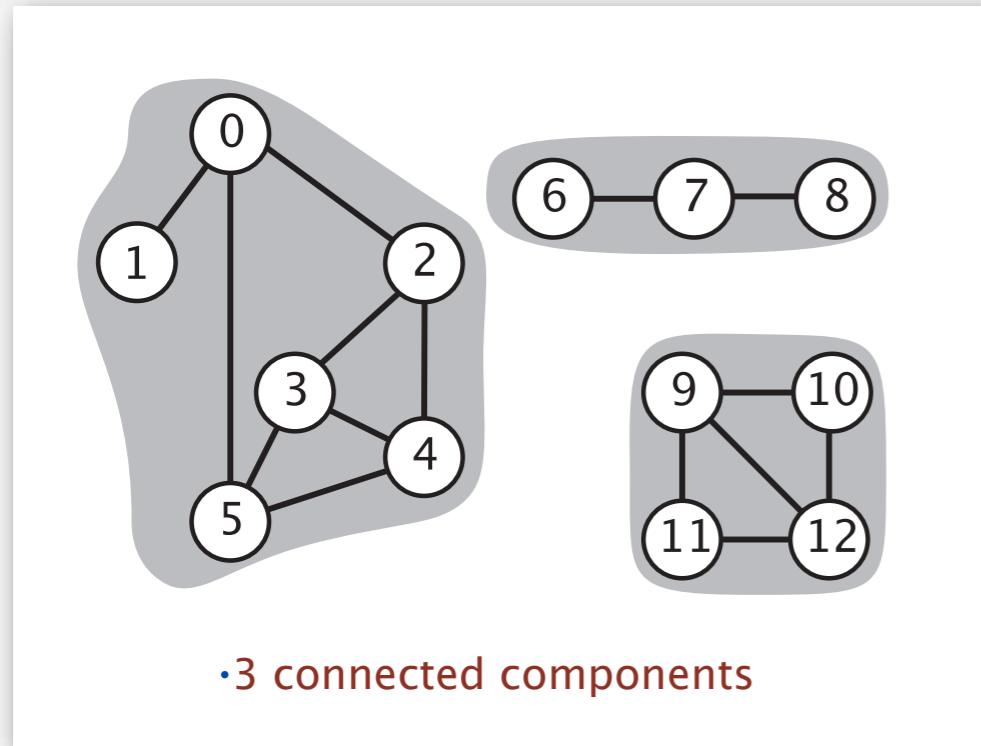


Examples of strongly-connected digraphs

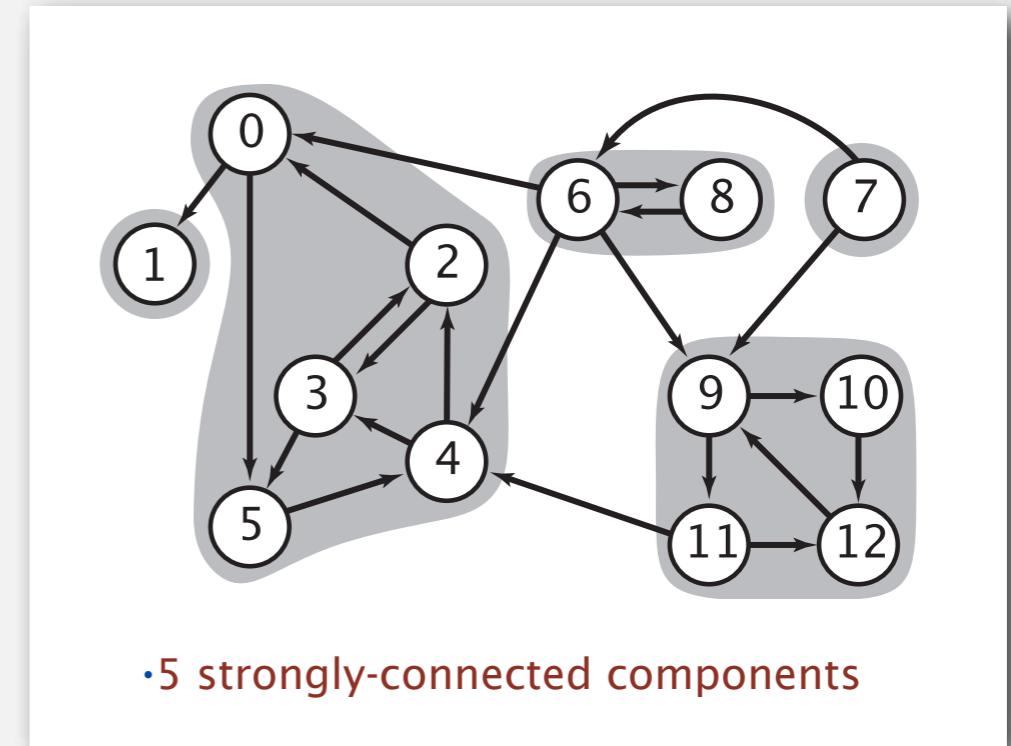


Connected components vs. strongly-connected components

- v and w are **connected** if there is a path between v and w



- v and w are **strongly connected** if there is a directed path from v to w and a directed path from w to v



connected component id (easy to compute with DFS)

	0	1	2	3	4	5	6	7	8	9	10	11	12
cc[]	0	0	0	0	0	0	1	1	2	2	2	2	2

```
public int connected(int v, int w)
{ return cc[v] == cc[w]; }
```



constant-time client connectivity query

strongly-connected component id (how to compute?)

	0	1	2	3	4	5	6	7	8	9	10	11	12
scc[]	1	0	1	1	1	1	3	4	3	2	2	2	2

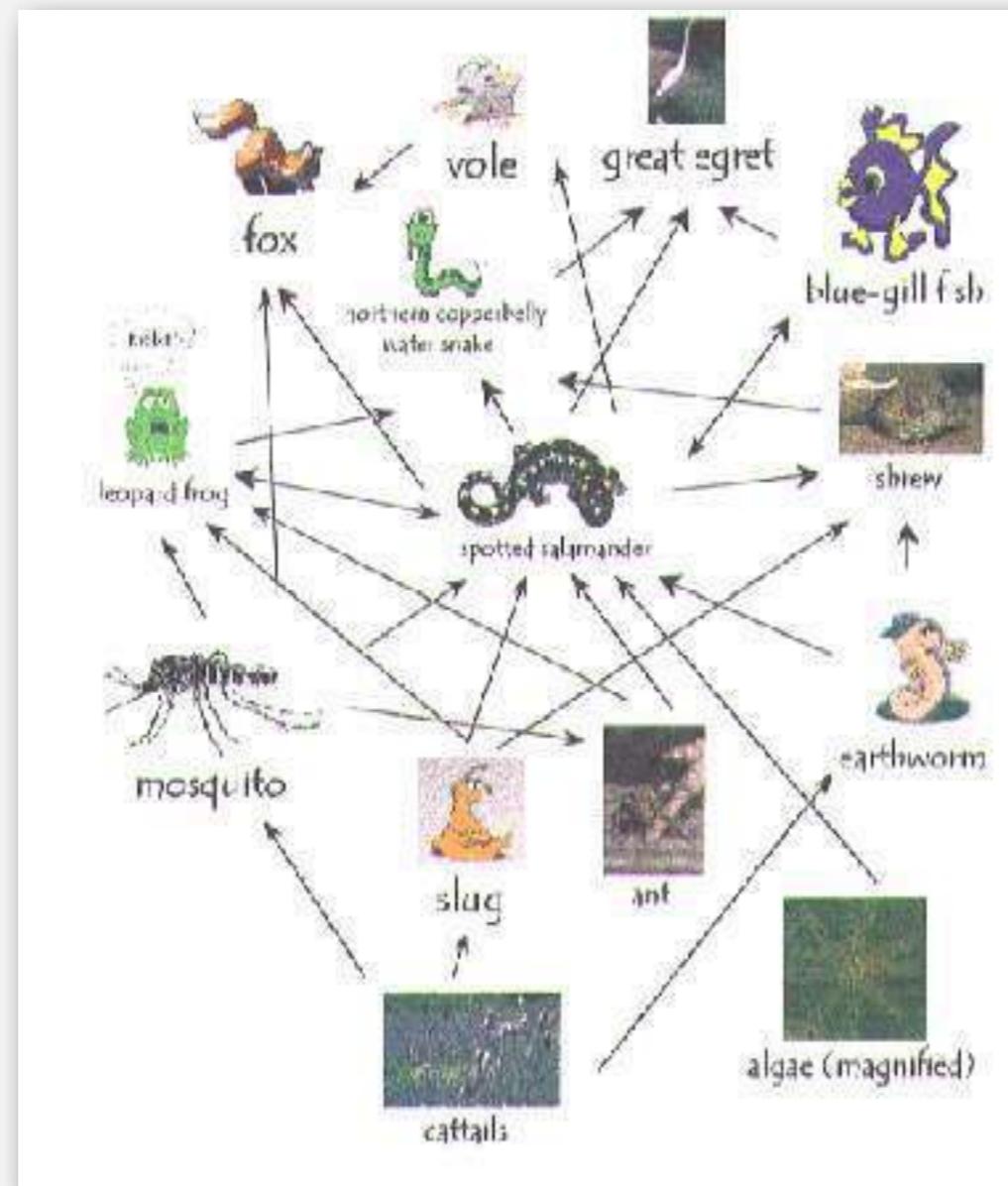
```
public int stronglyConnected(int v, int w)
{ return scc[v] == scc[w]; }
```



constant-time client strong-connectivity query

Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.



<http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif>

Strong component. Subset of species with common energy flow.

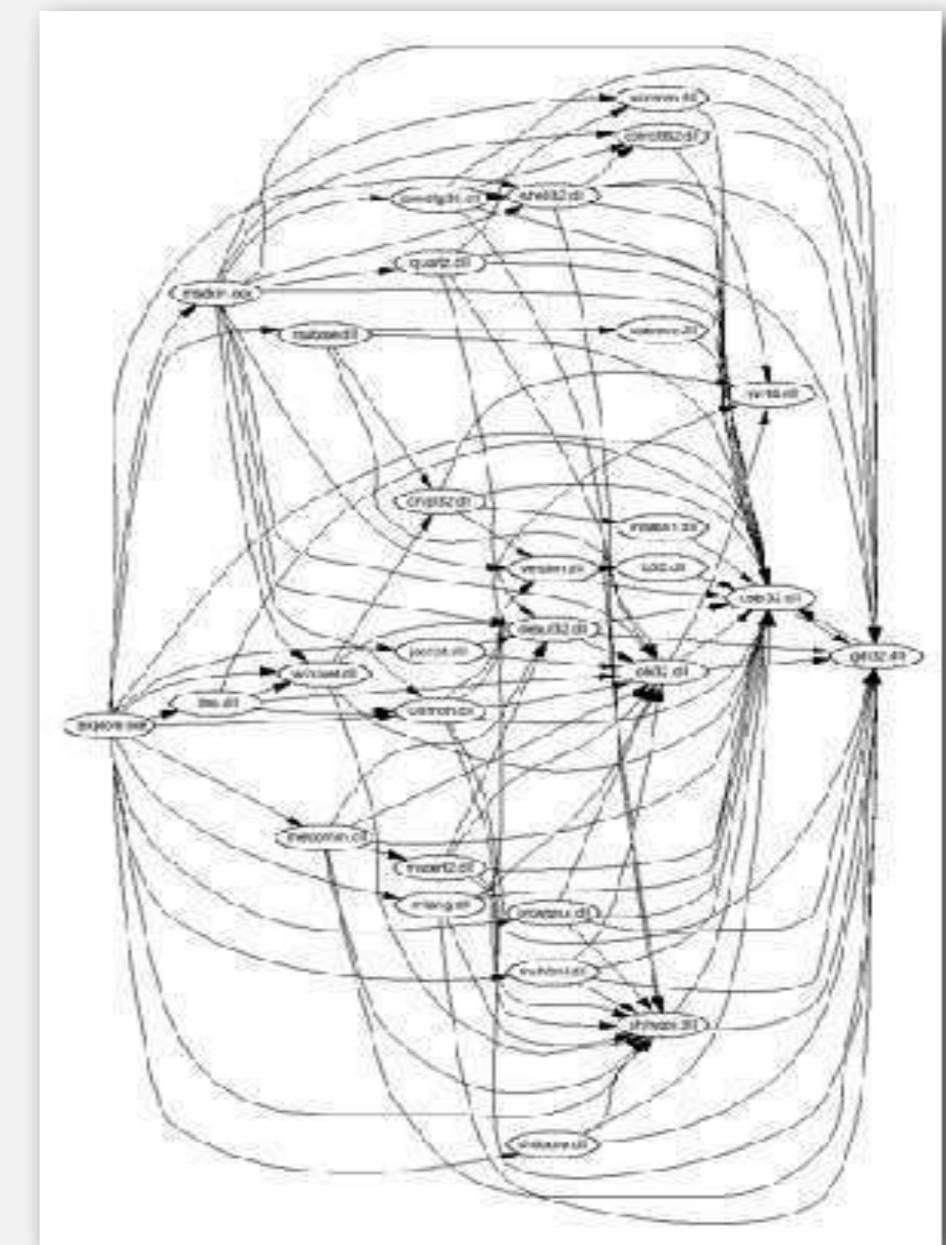
Strong component application: software modules

Software module dependency graph.

- Vertex = software module.
 - Edge: from module to dependency.



Firefox



Internet Explorer

Strong component. Subset of mutually interacting modules.

Approach I. Package strong components together.

Approach 2. Use to improve design!

Strong components algorithms: brief history

1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms.

- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.

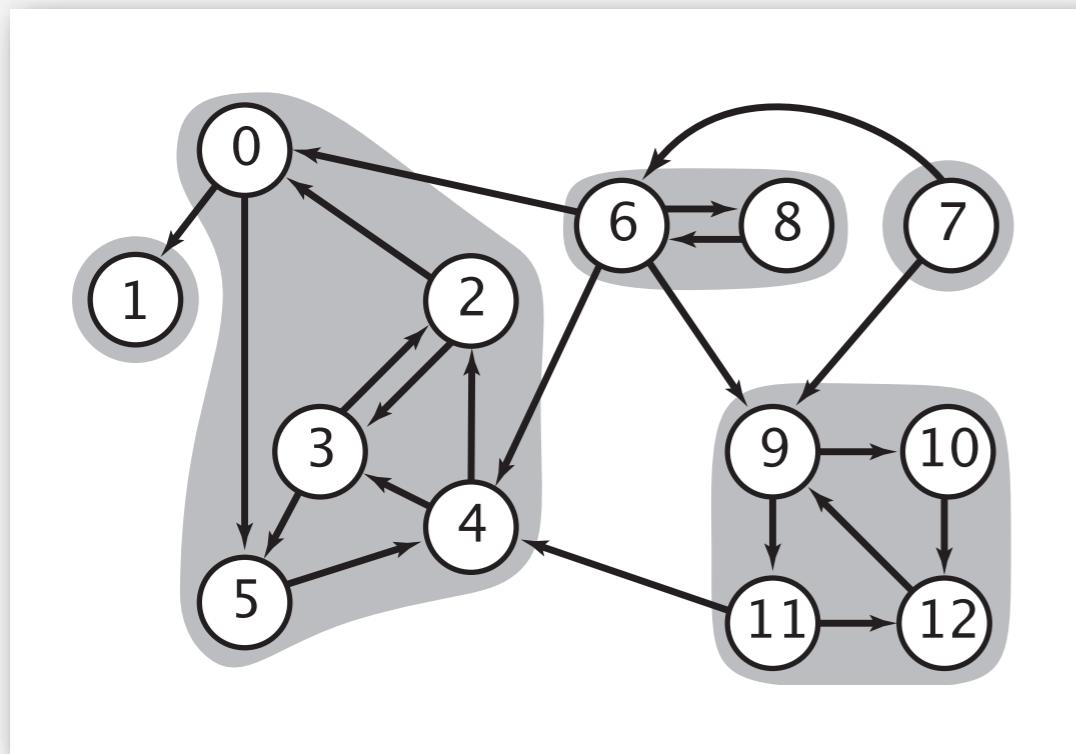
Kosaraju's algorithm: intuition

Reverse graph. Strong components in G are same as in G^R .

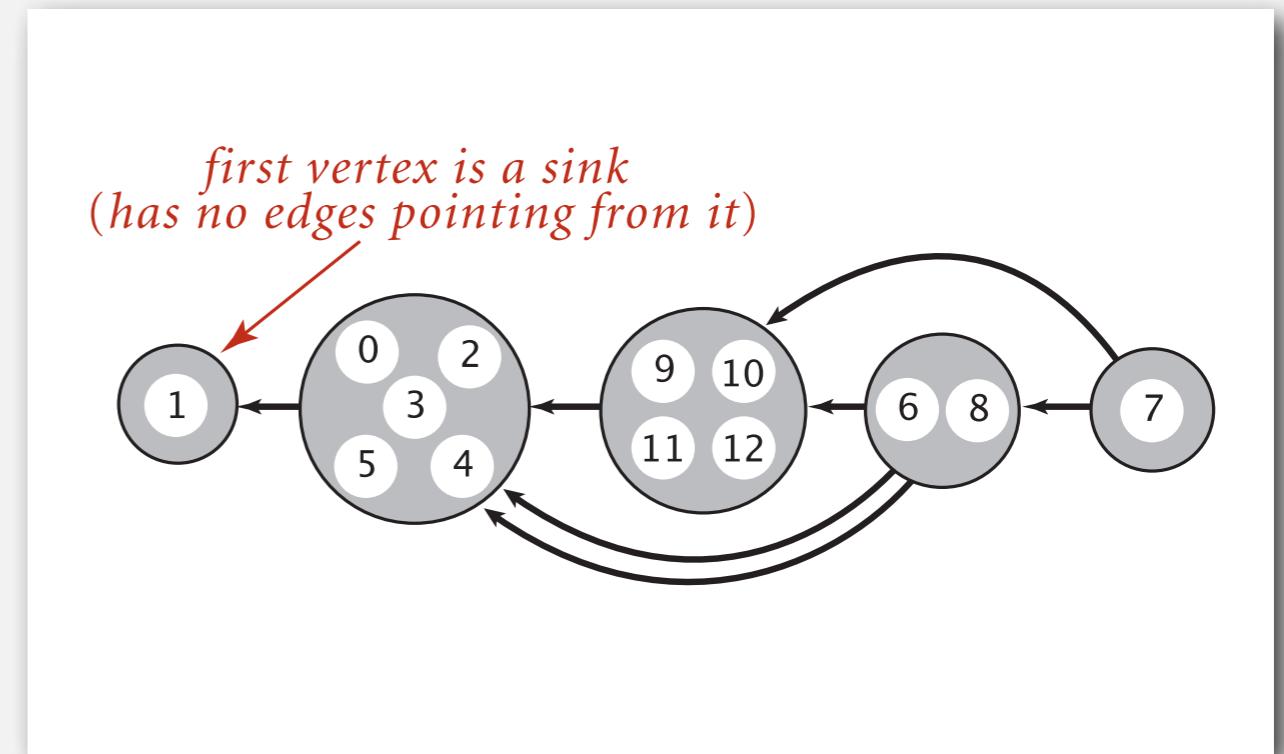
Kernel DAG. Contract each strong component into a single vertex.

Idea.

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.



digraph G and its strong components



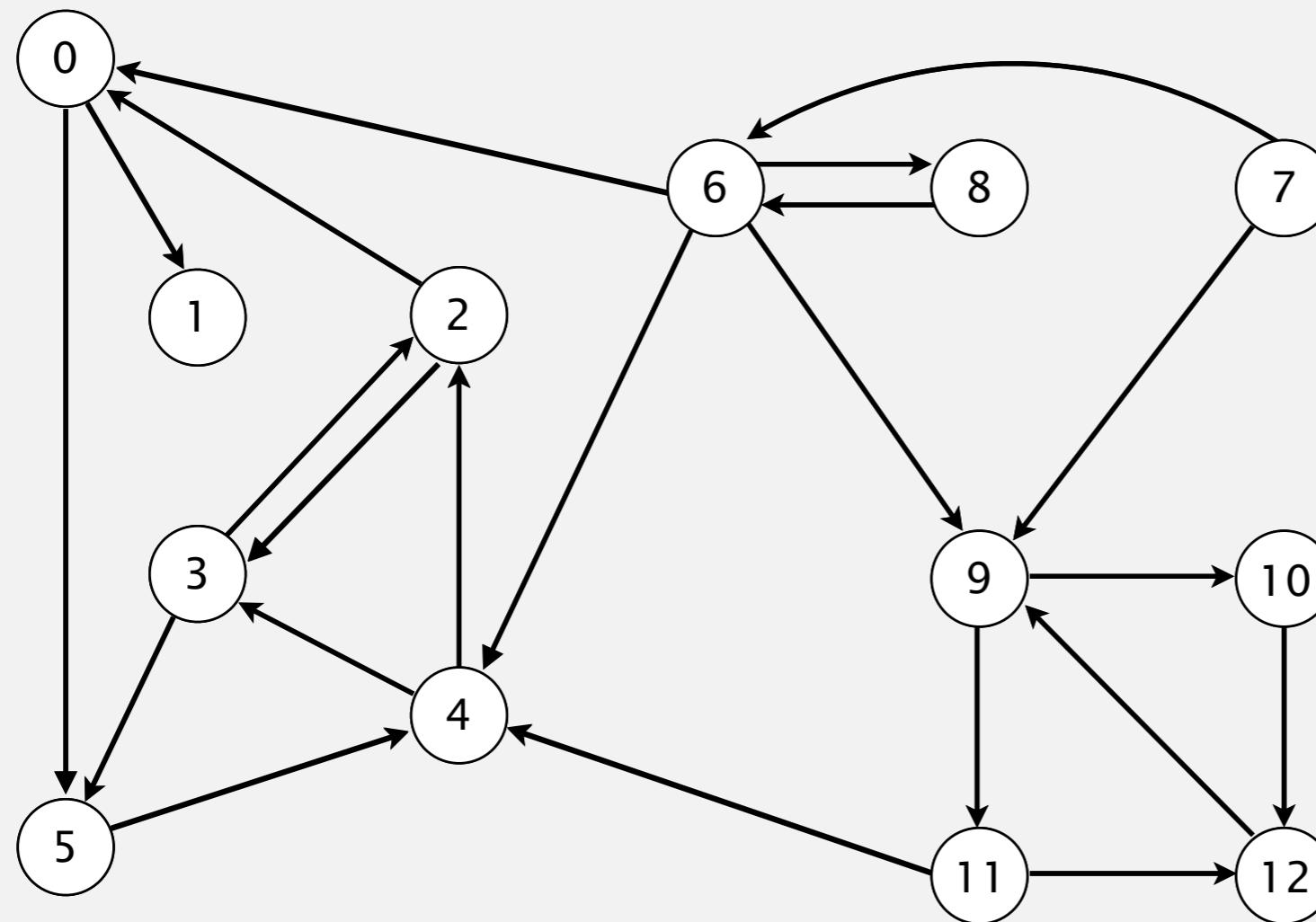
kernel DAG of G (in reverse topological order)

KOSARAJU'S ALGORITHM

- ▶ DFS in reverse graph
- ▶ DFS in original graph

Kosaraju-Sharir

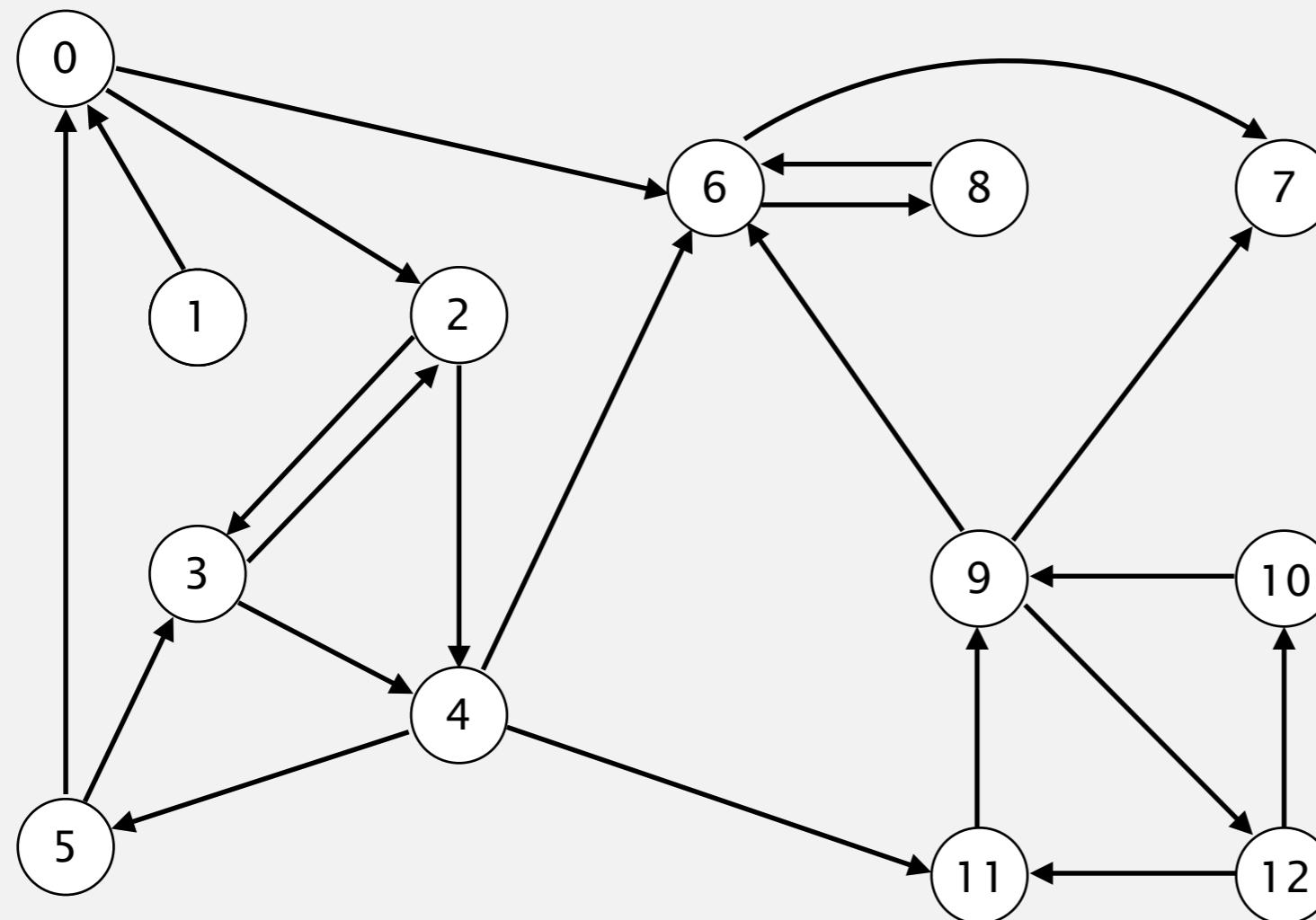
Phase I. Compute reverse postorder in G^R .



digraph G

Kosaraju-Sharir

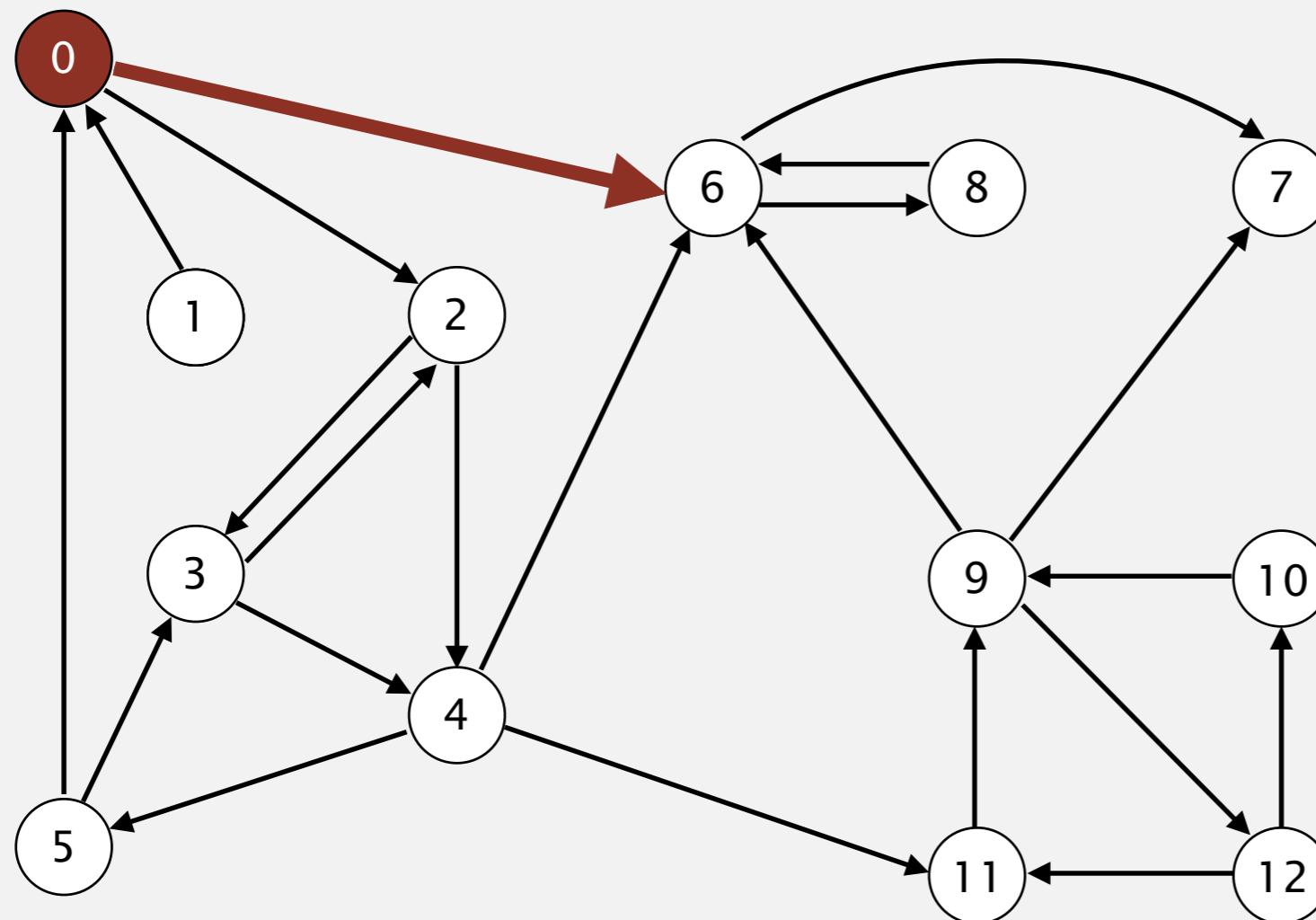
Phase I. Compute reverse postorder in G^R .



v	marked[v]
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

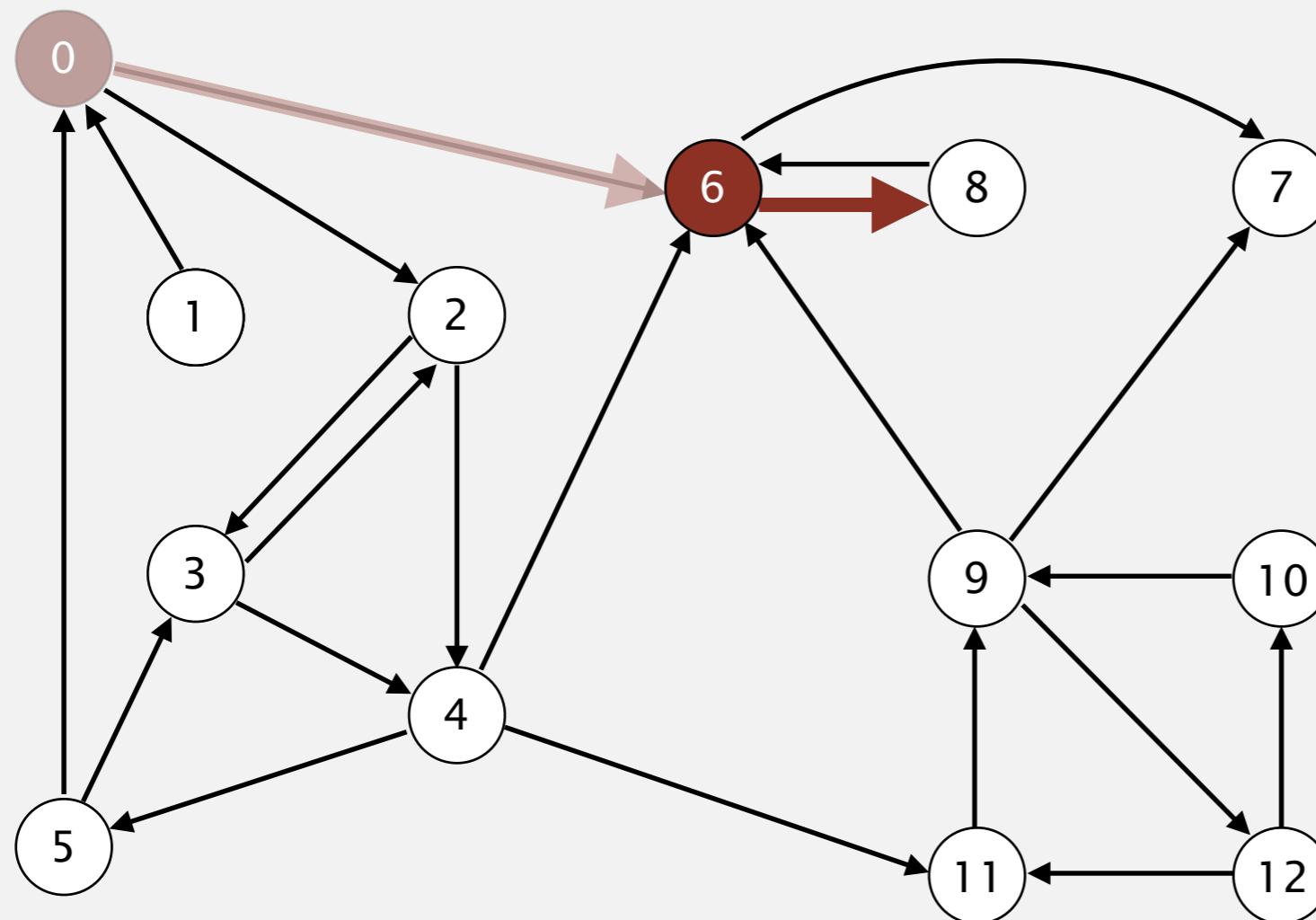


v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F
10	F
11	F
12	F

visit 0: check 6 and check 2

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

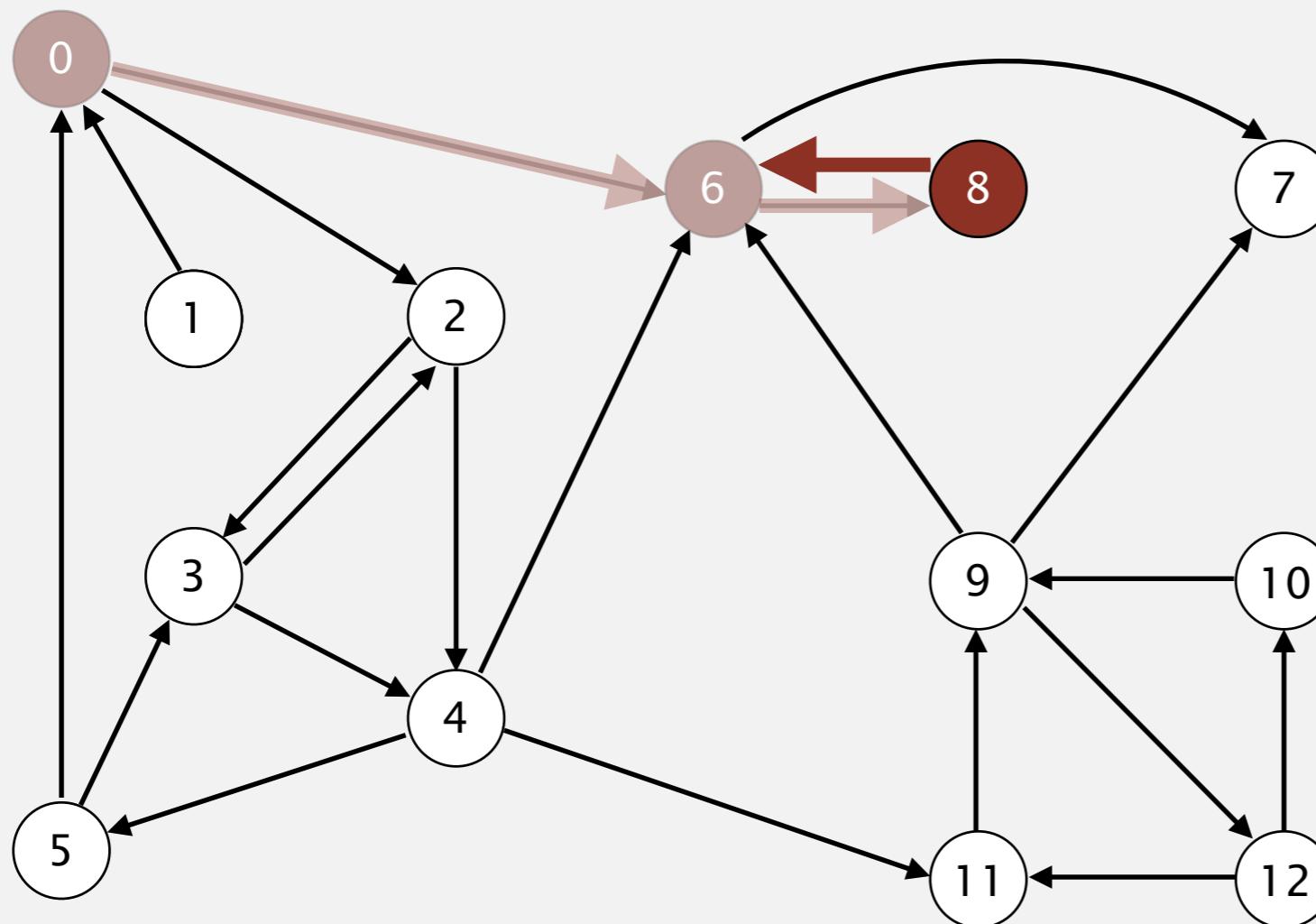


v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	F
9	F
10	F
11	F
12	F

visit 6: check 8 and check 7

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

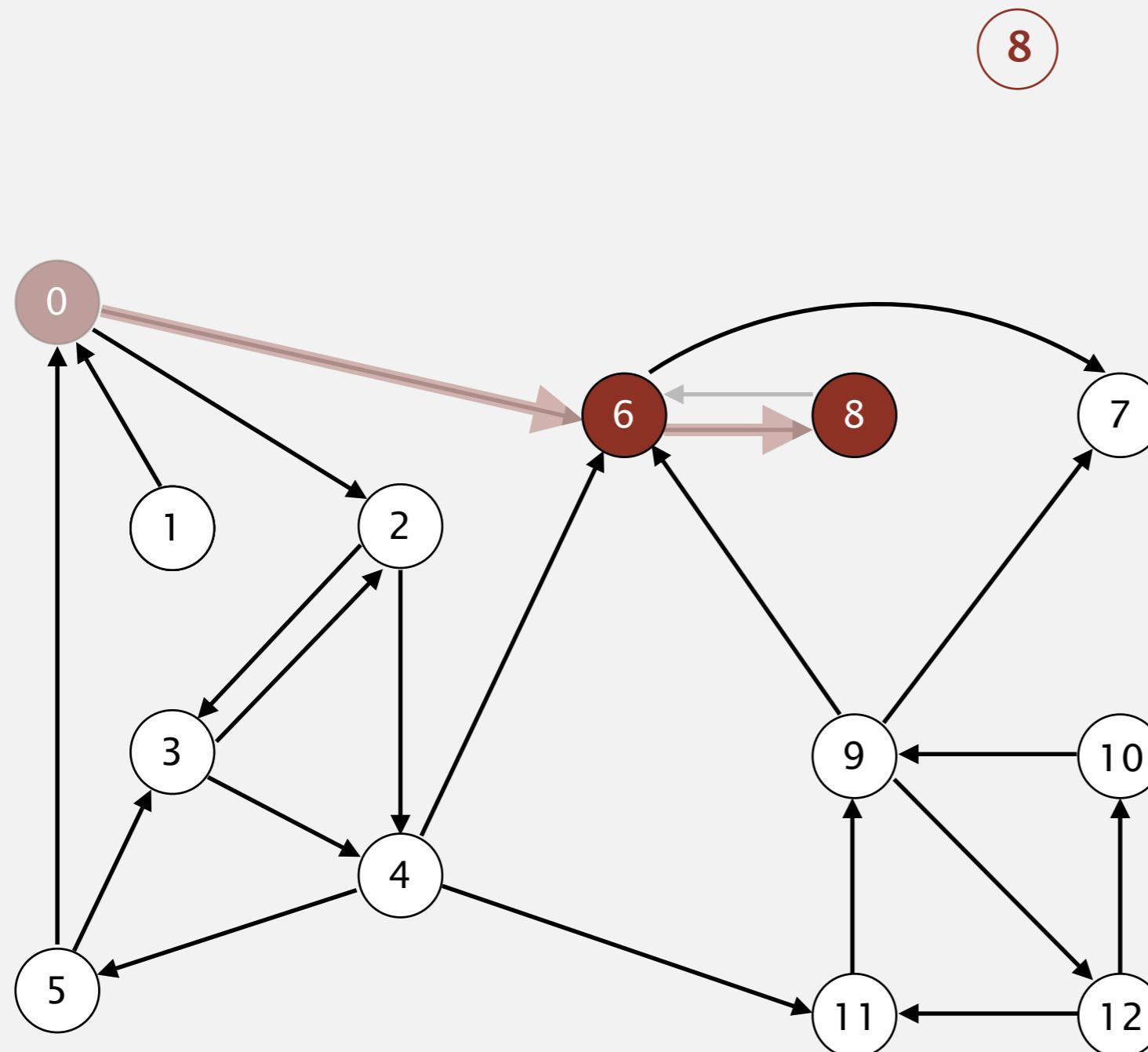


visit 8: check 6

v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .



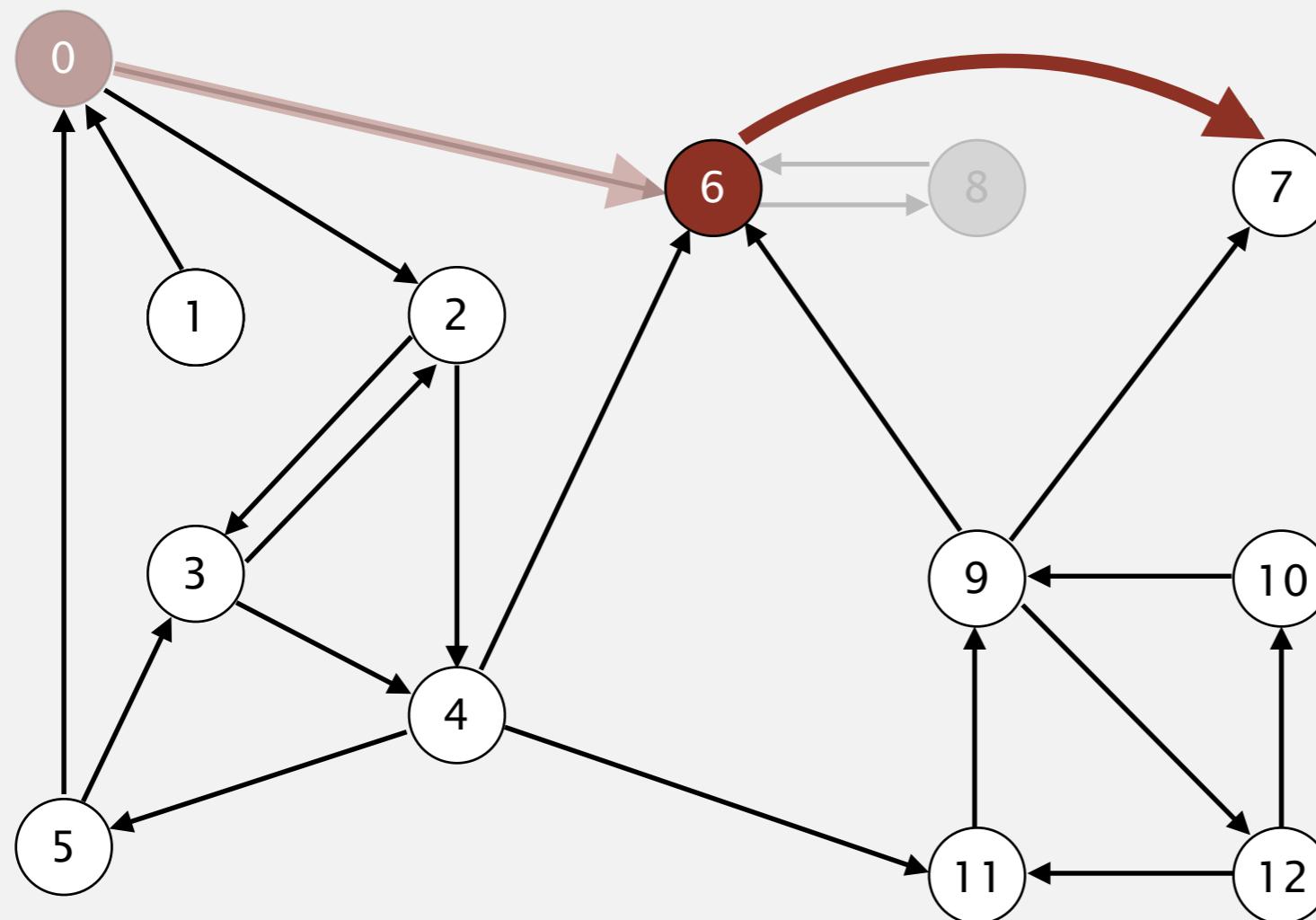
v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

8 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

8



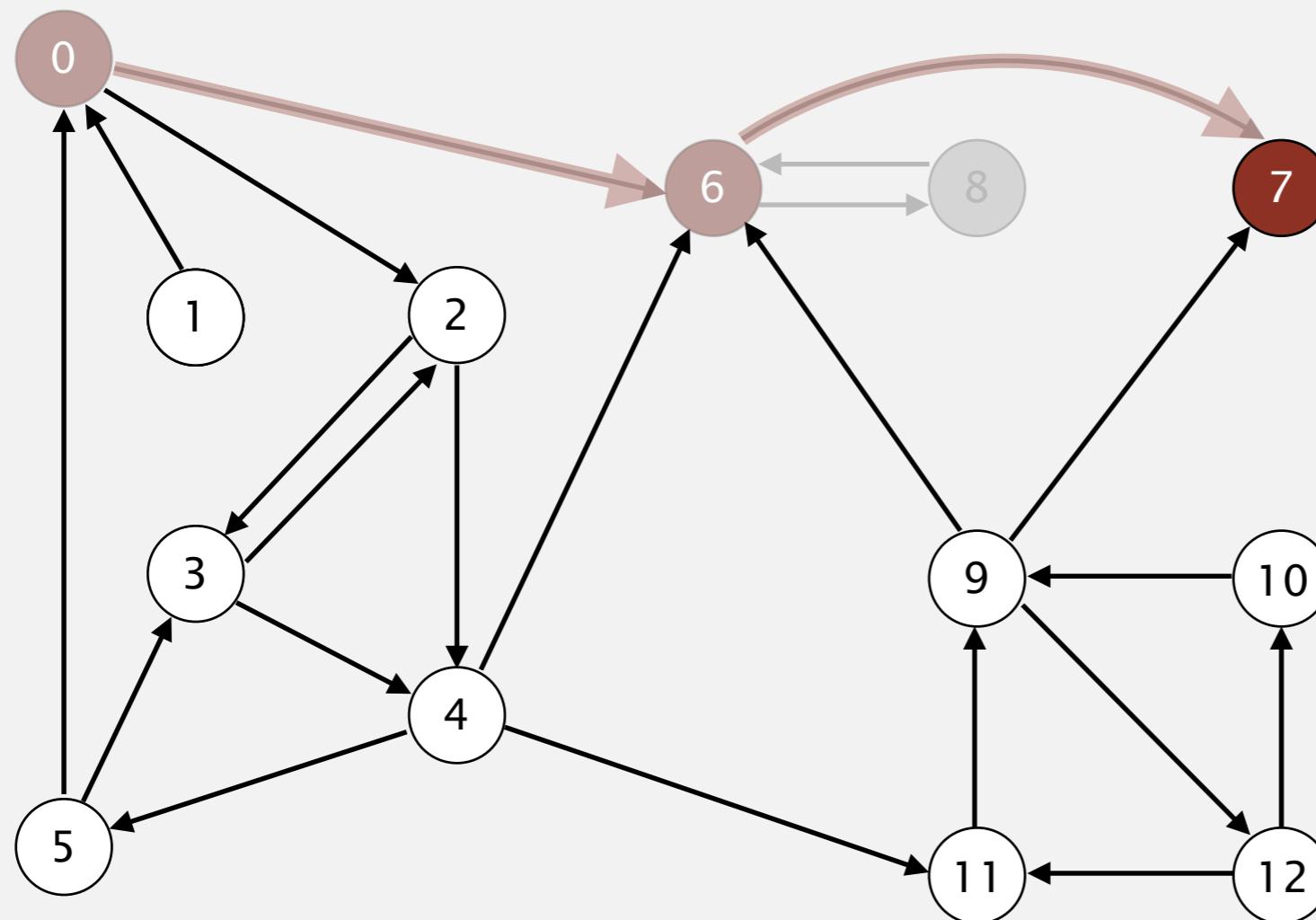
v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	F
8	T
9	F
10	F
11	F
12	F

visit 6: check 8 and check 7

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

8

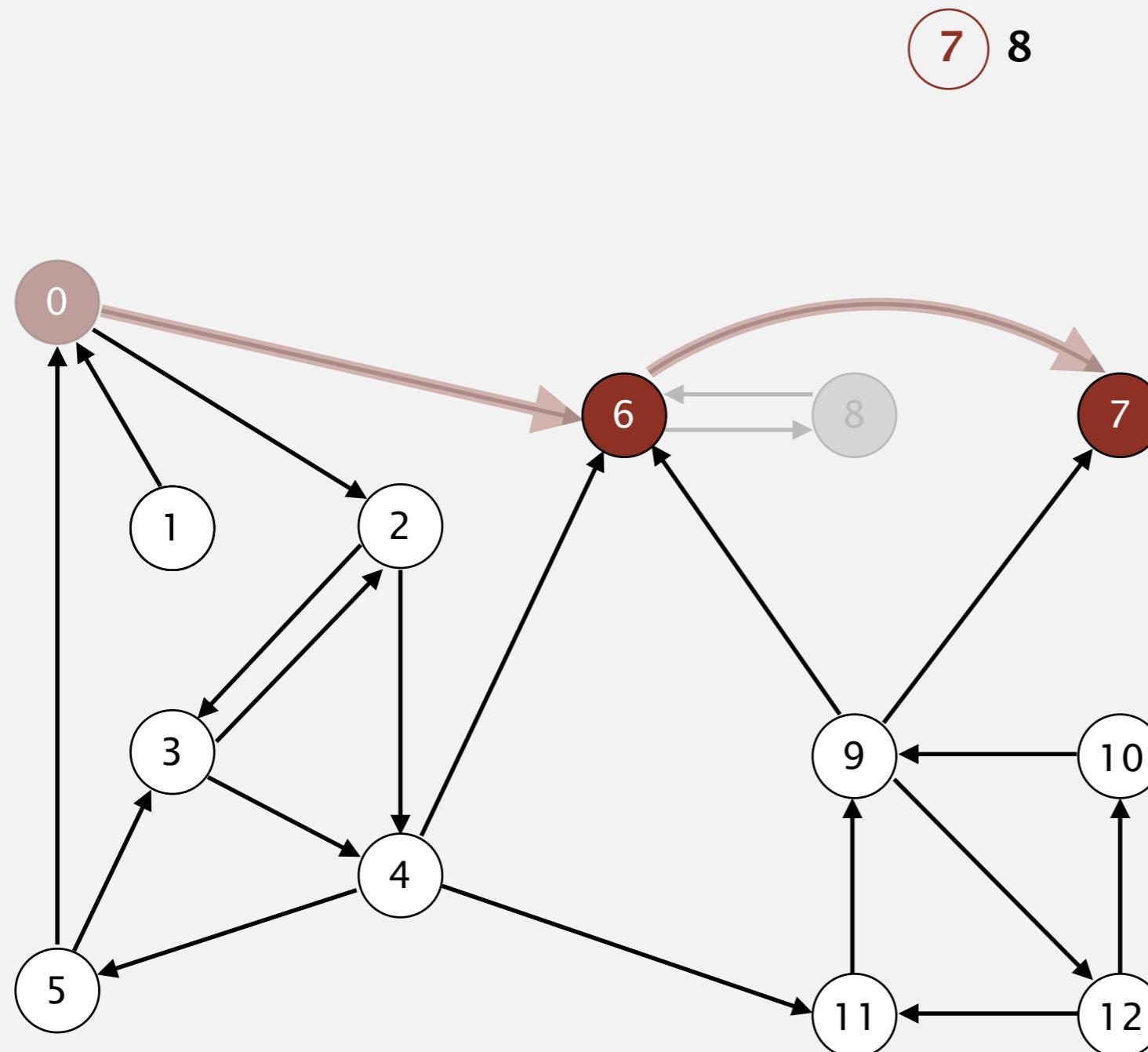


visit 7

v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

Kosaraju-Sharir

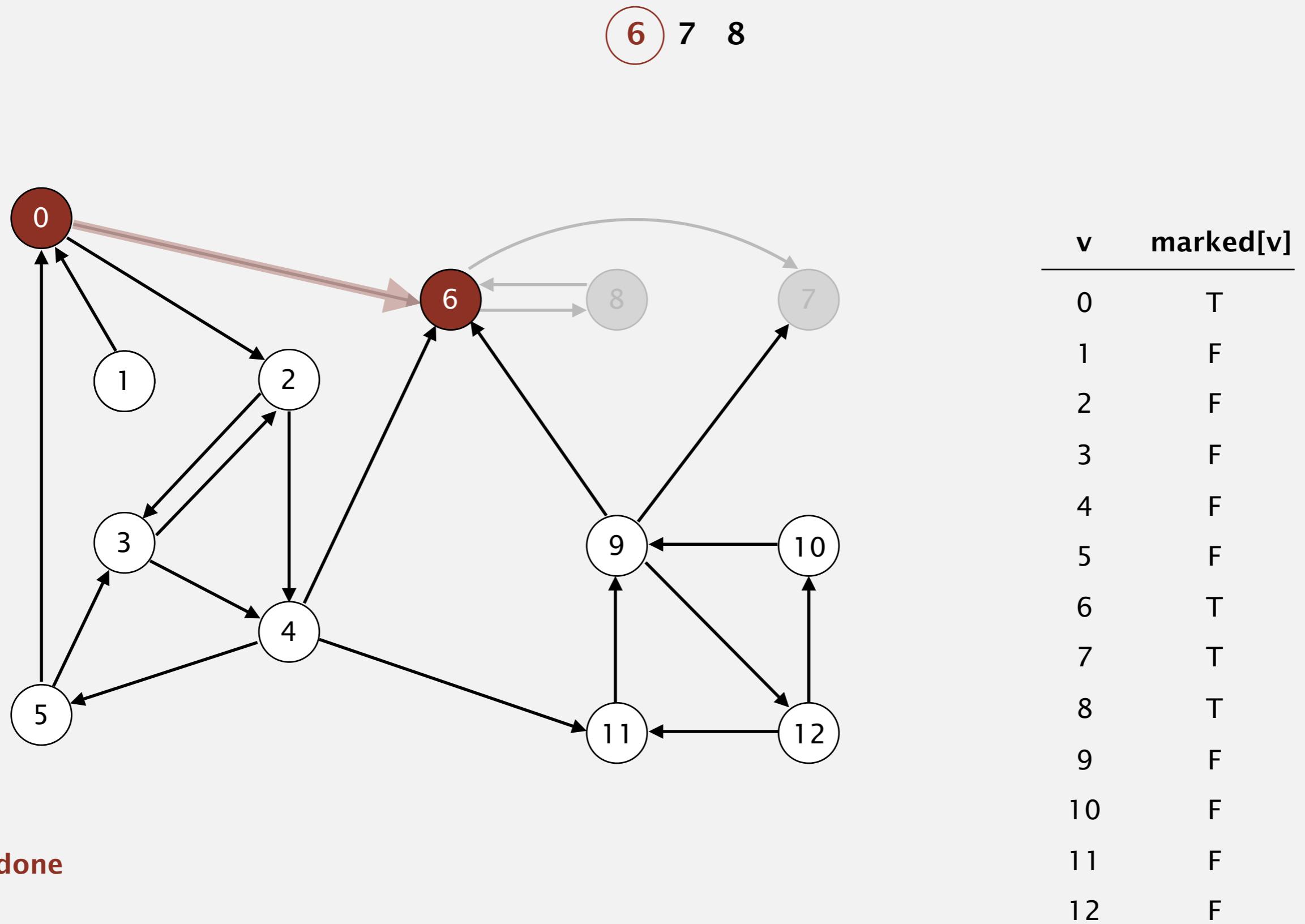
Phase I. Compute reverse postorder in G^R .



v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

Kosaraju-Sharir

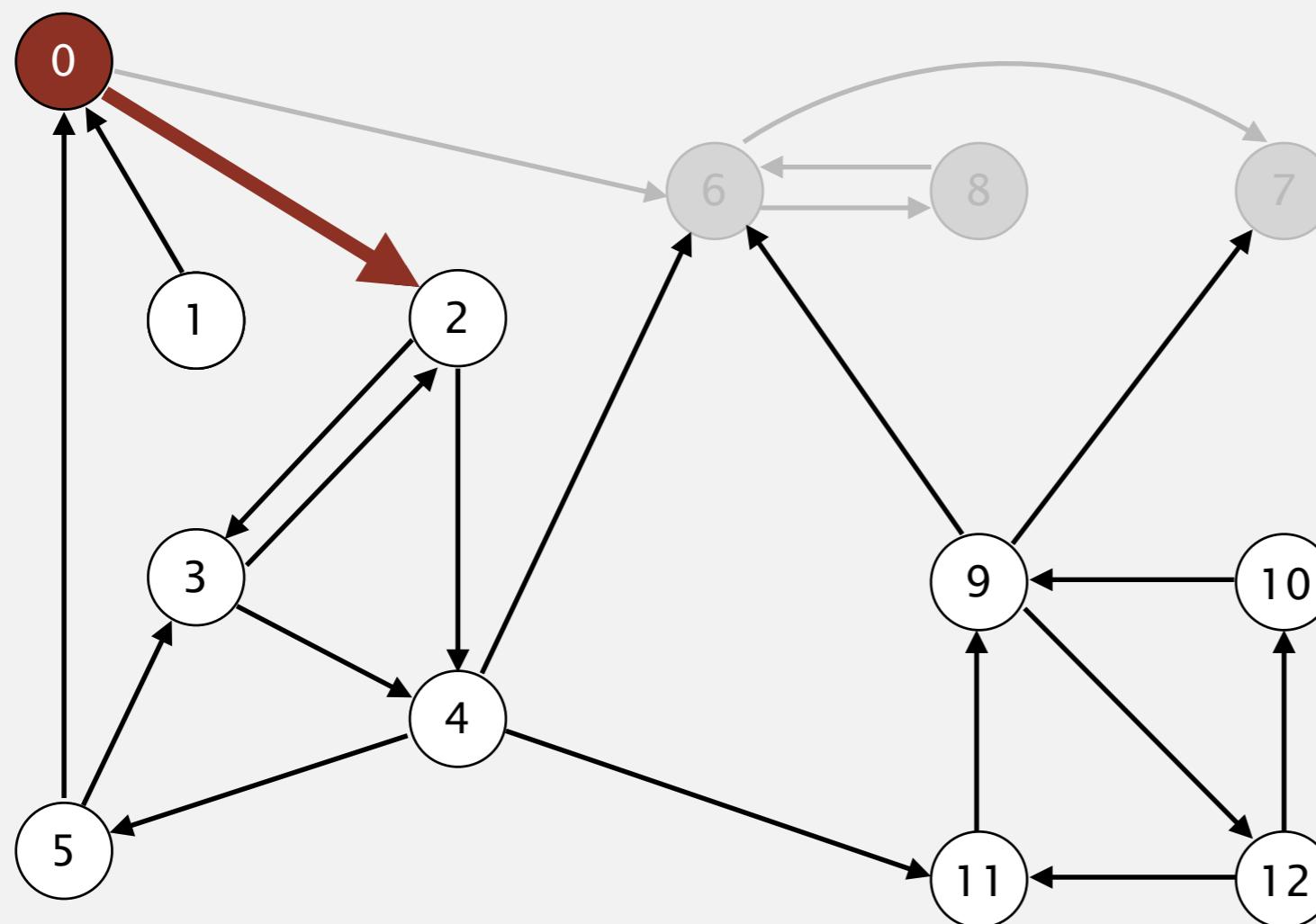
Phase I. Compute reverse postorder in G^R .



Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



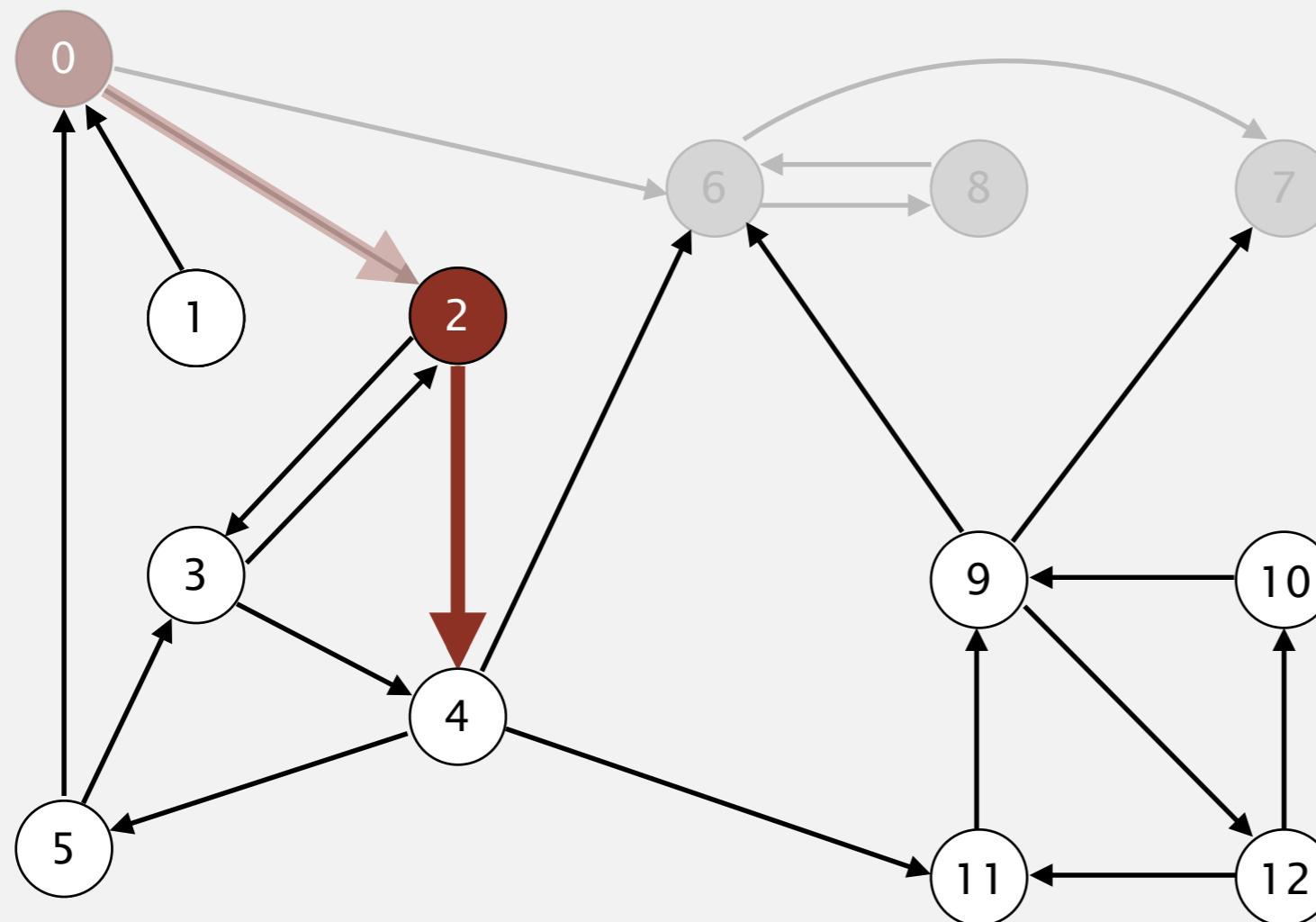
v	marked[v]
0	T
1	F
2	F
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 0: check 6 and check 2

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



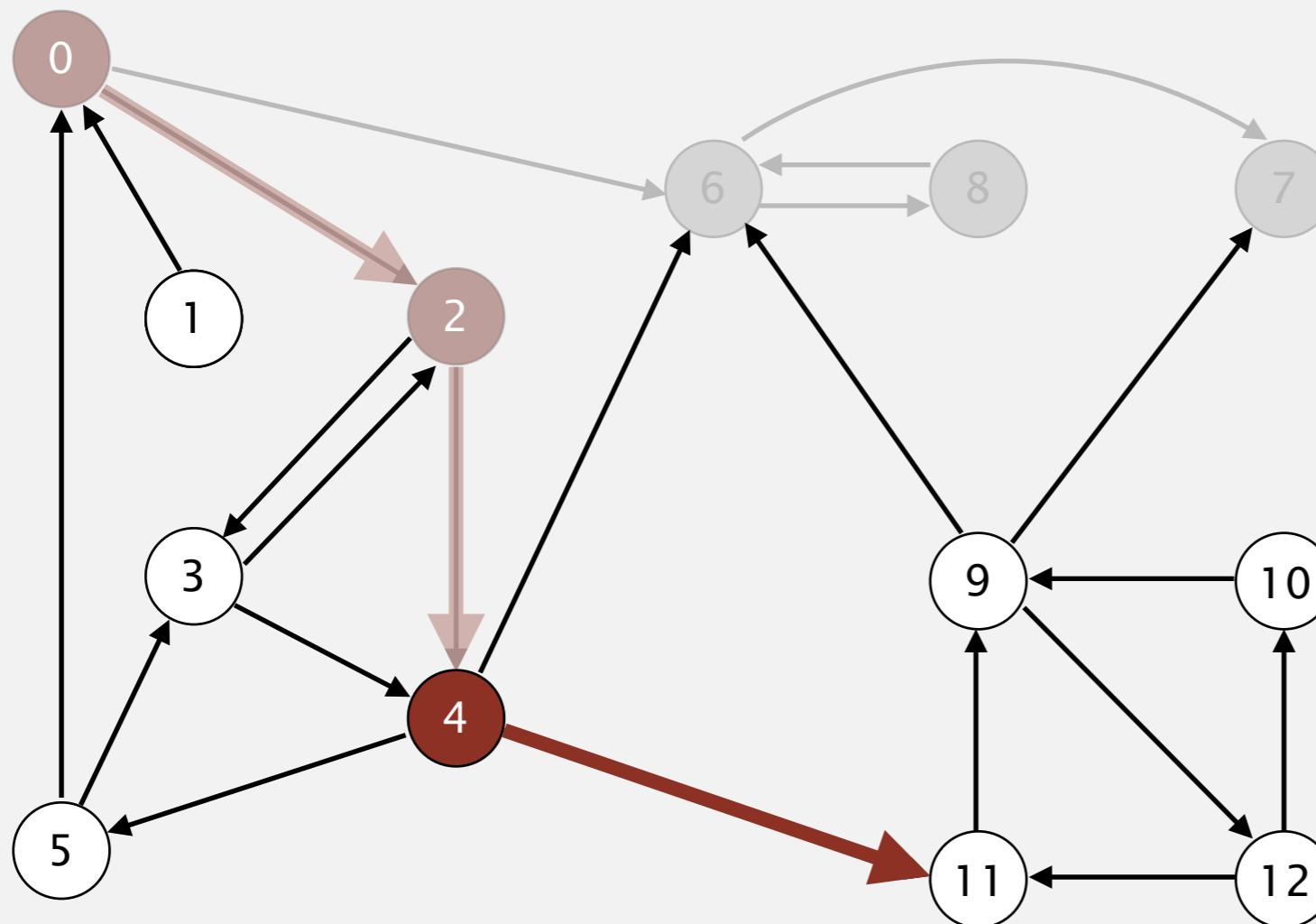
v	marked[v]
0	T
1	F
2	T
3	F
4	F
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 2: check 4 and check 3

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



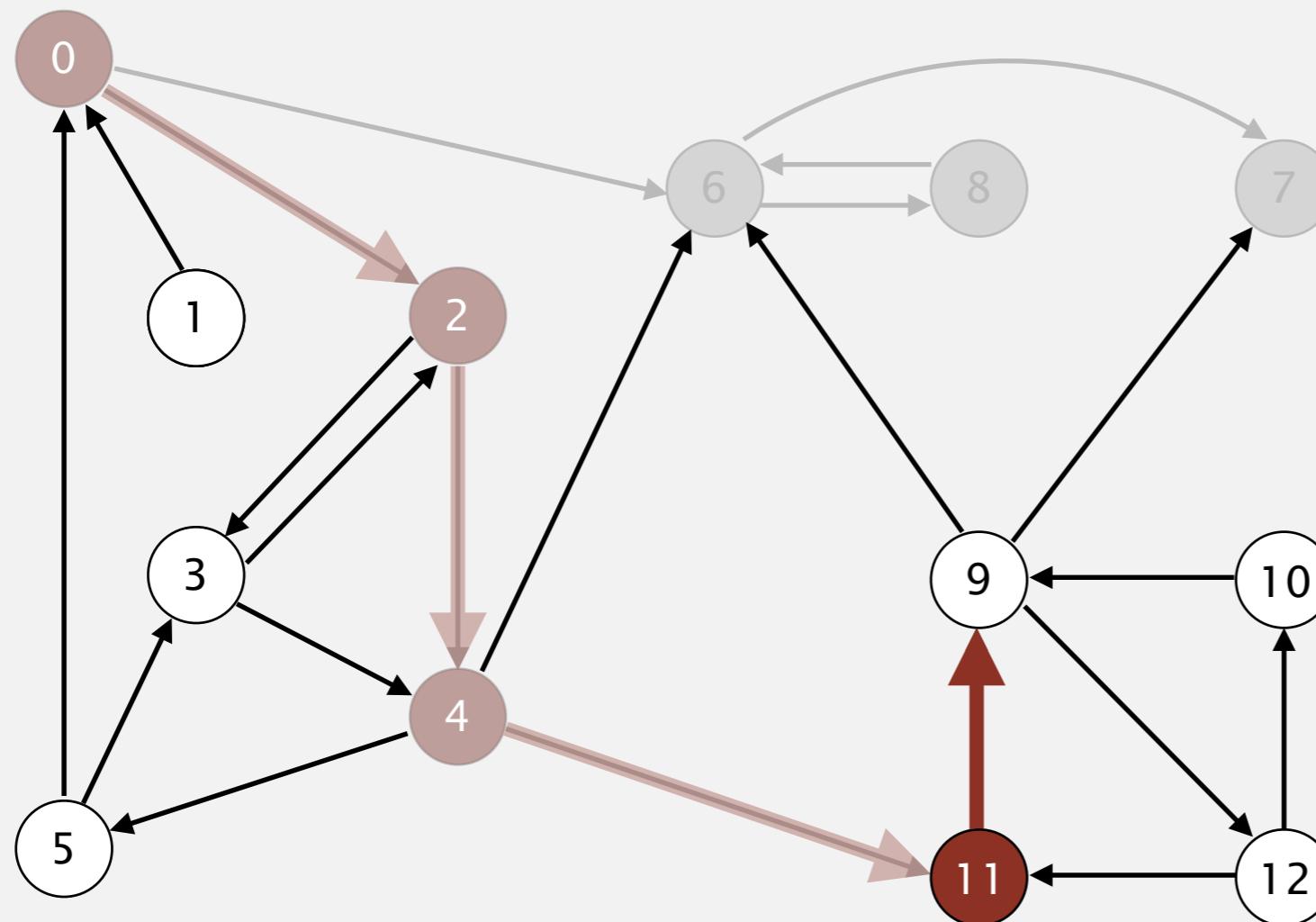
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	F
10	F
11	F
12	F

visit 4: check 11, check 6, and check 5

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



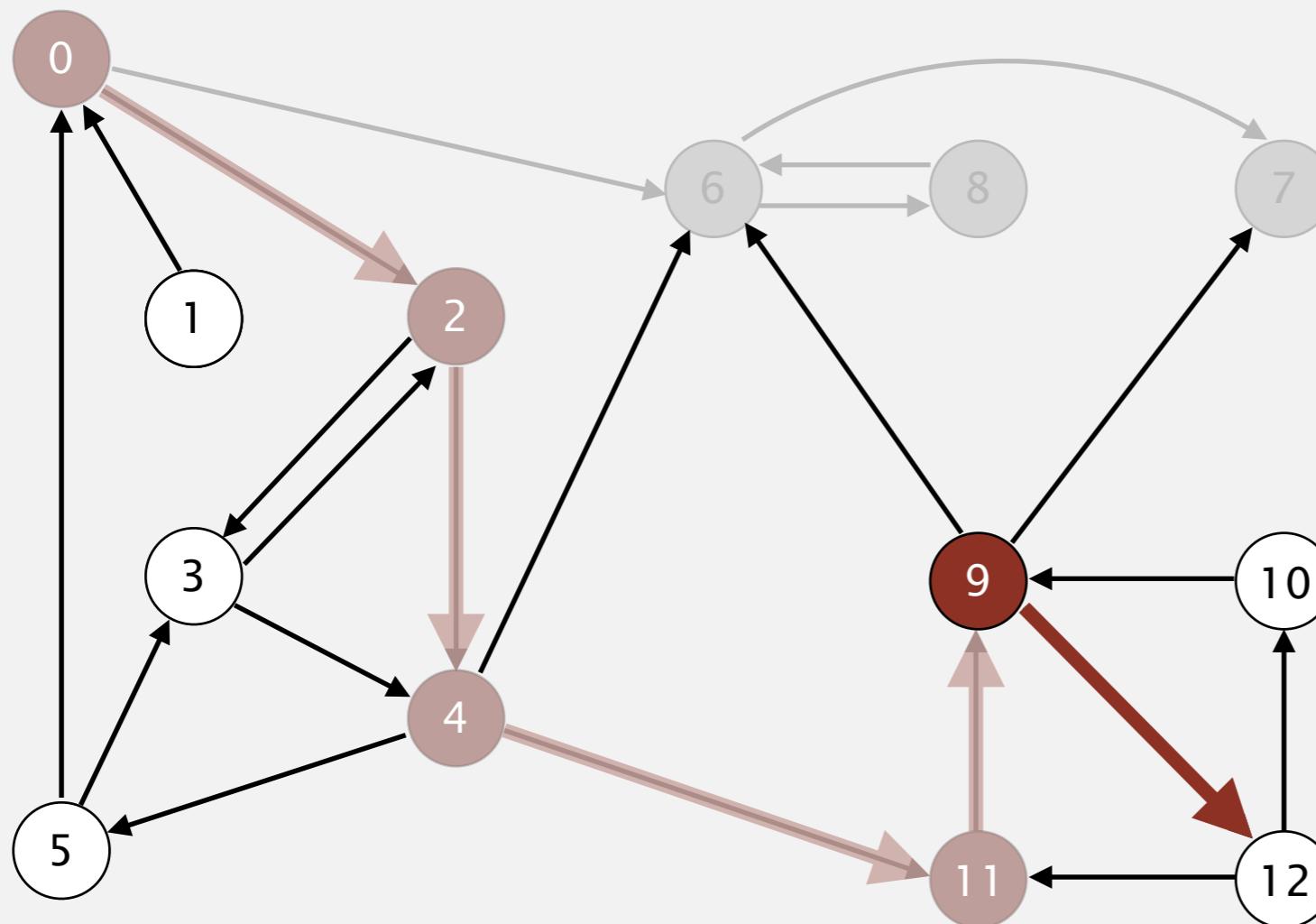
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	F
10	F
11	T
12	F

visit 11: check 9

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



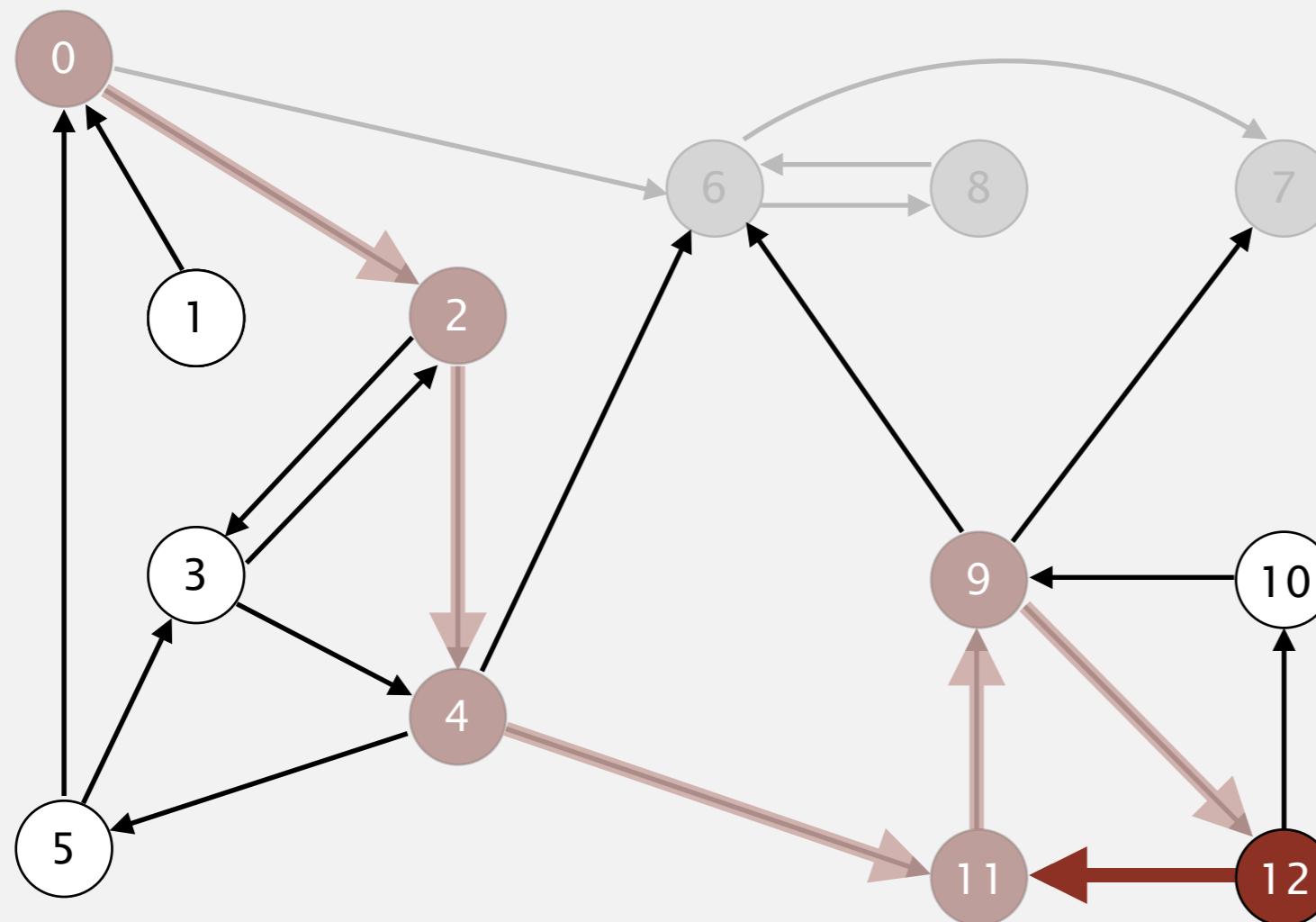
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	F

visit 9: check 12, check 7, and check 6

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



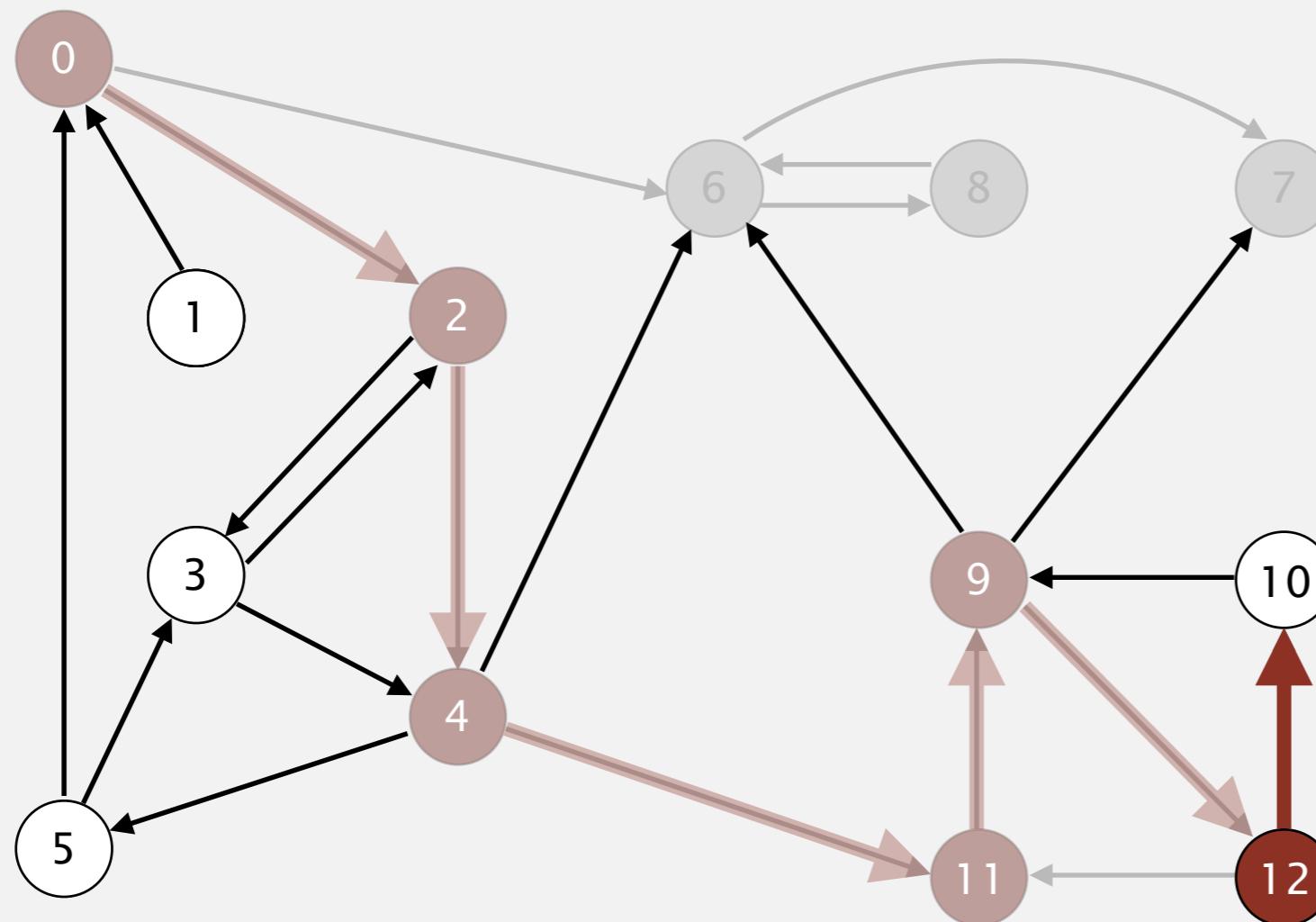
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	T

visit 12: check 11 and check 10

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8



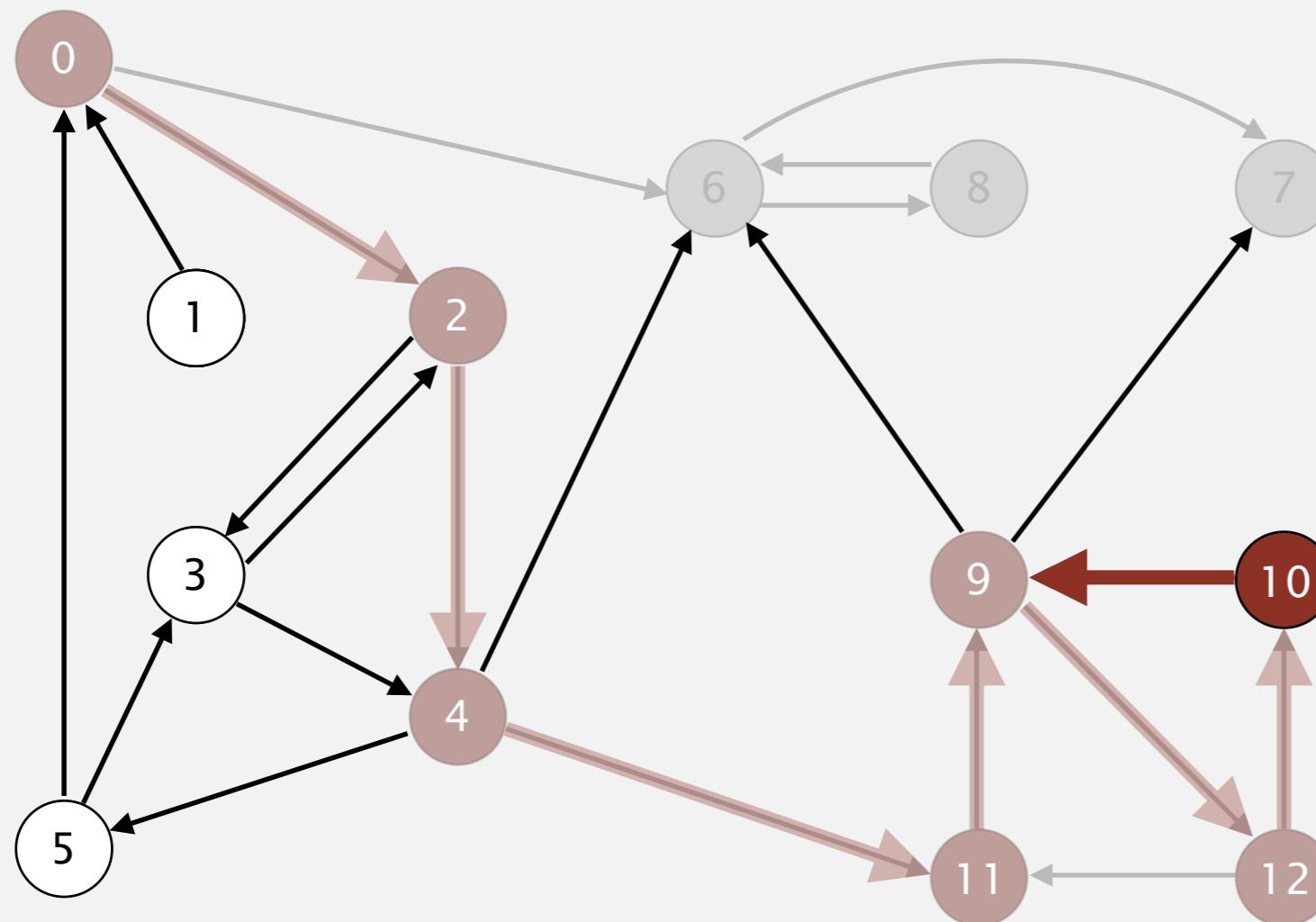
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	F
11	T
12	T

visit 12: check 11 and check 10

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

6 7 8

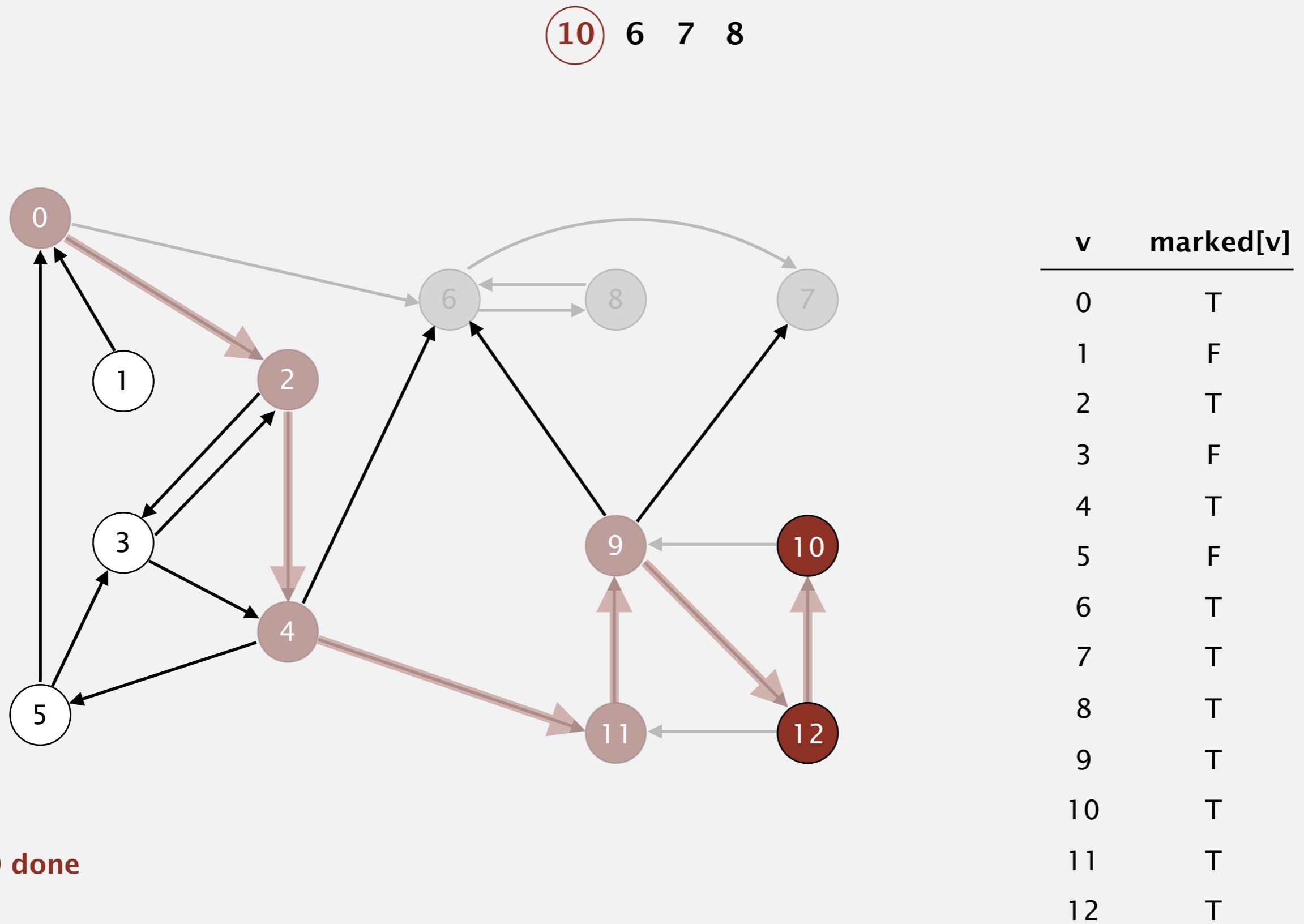


v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 10: check 9

Kosaraju-Sharir

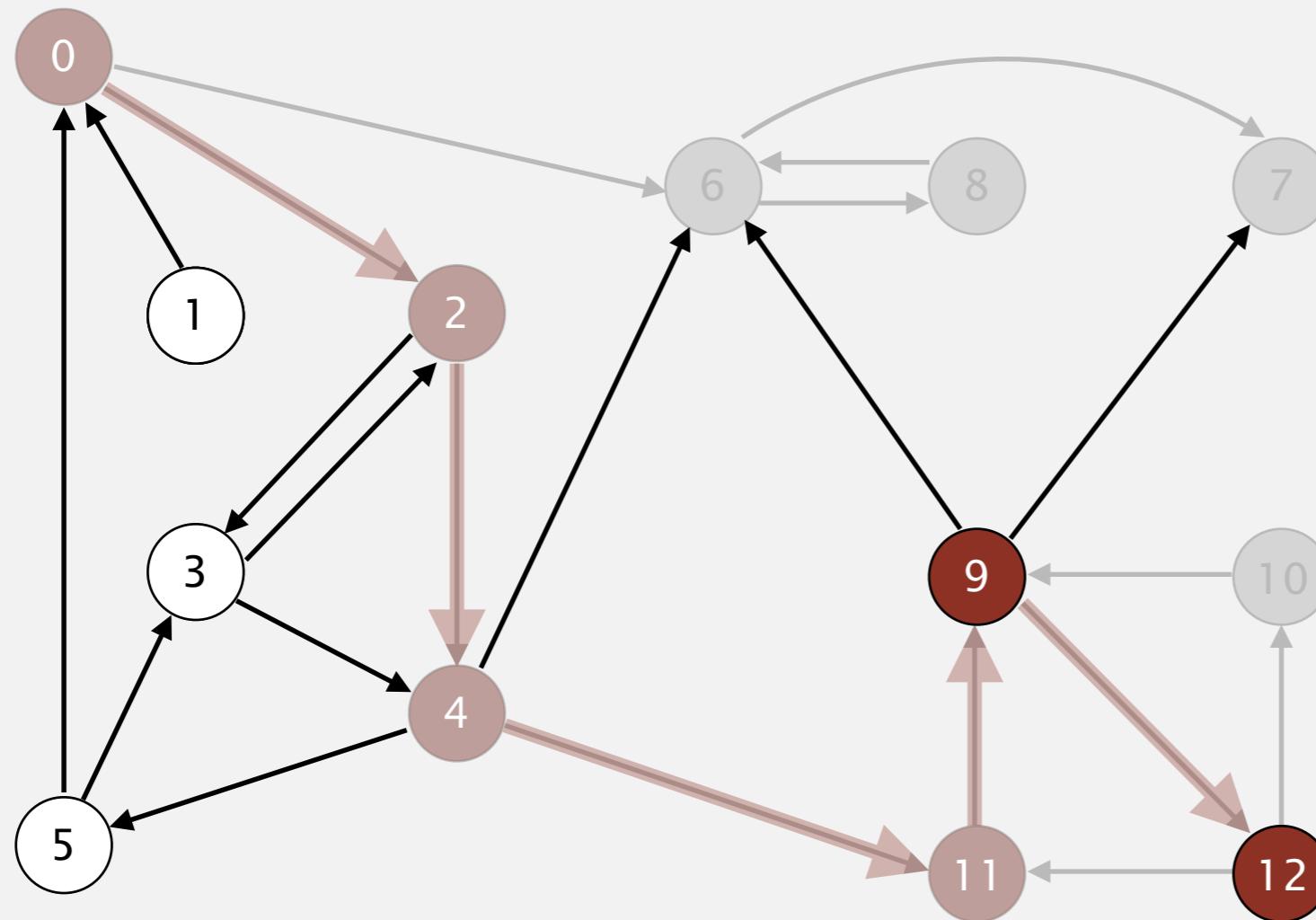
Phase I. Compute reverse postorder in G^R .



Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

12 10 6 7 8



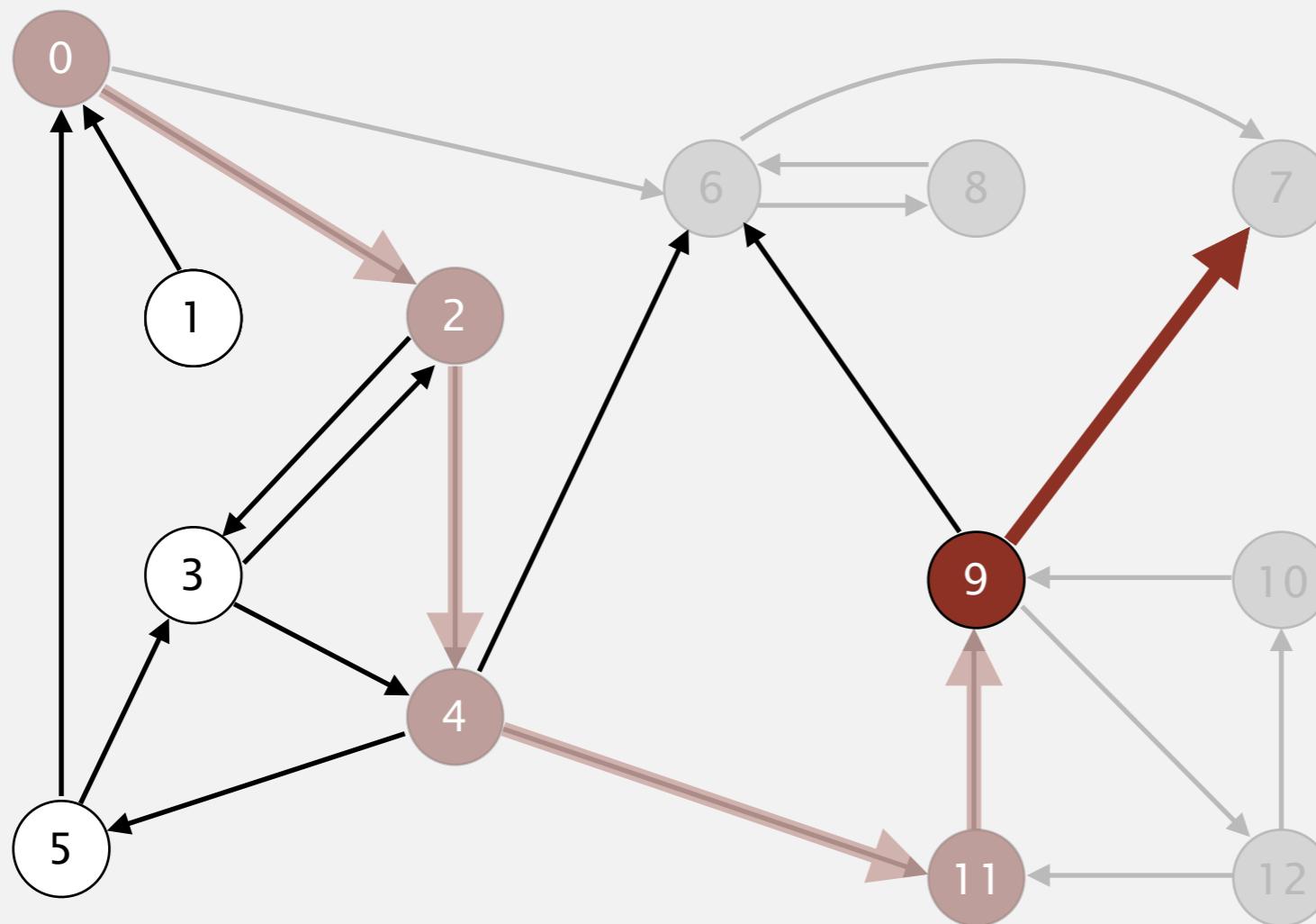
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

12 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

12 10 6 7 8



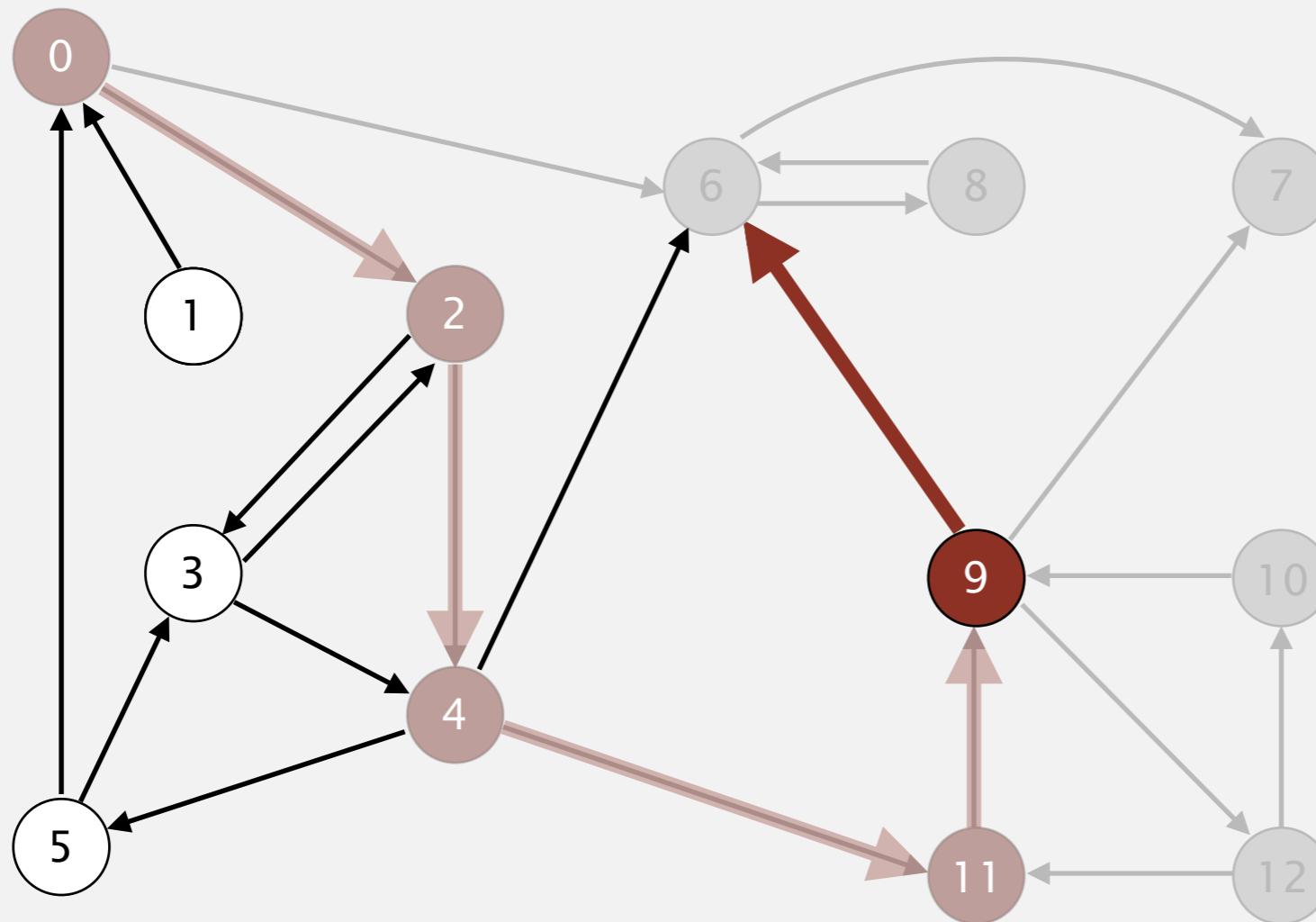
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 9: check 12, check 7, and check 6

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

12 10 6 7 8



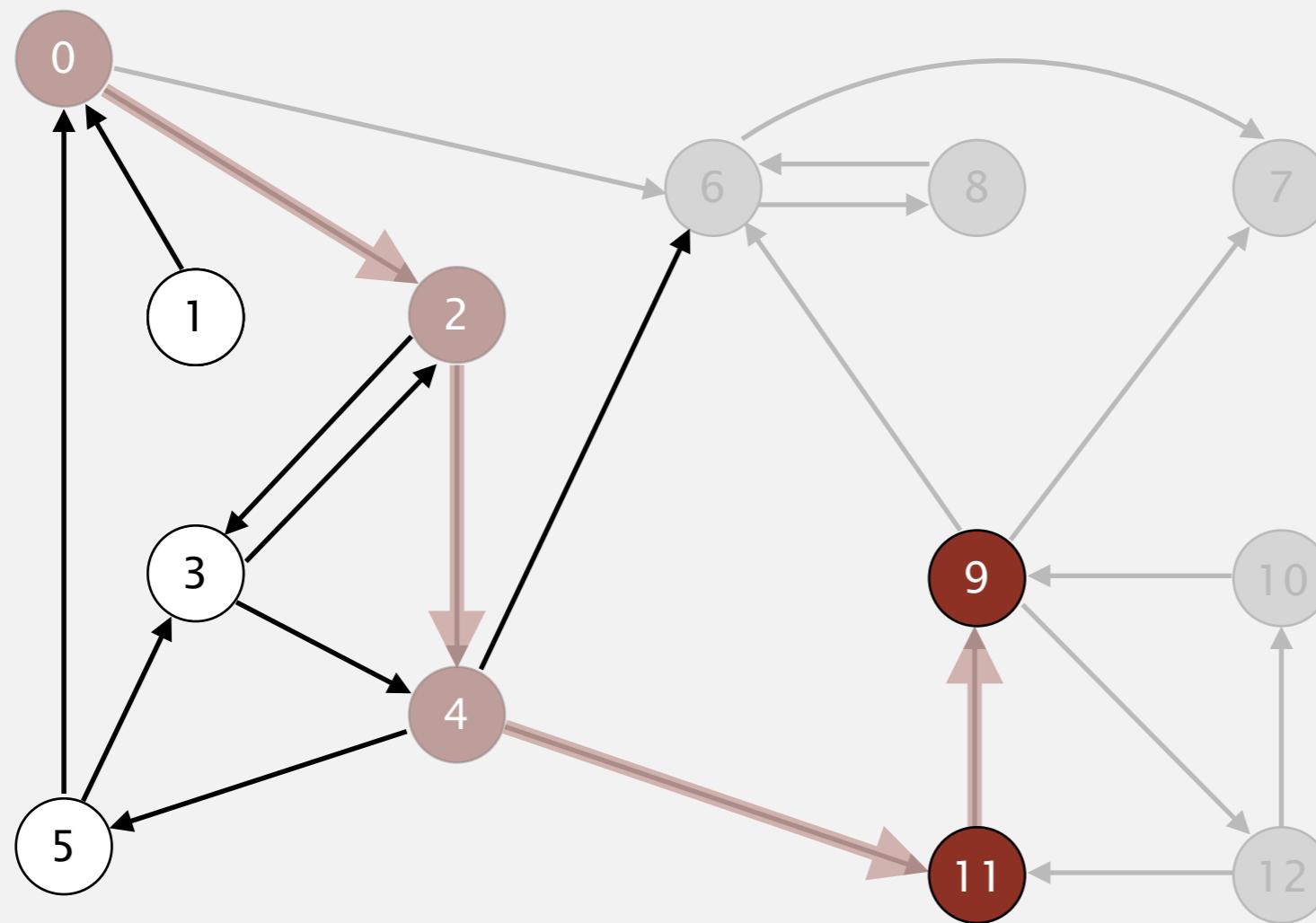
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 9: check 12, check 7, and check 6

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

9 12 10 6 7 8



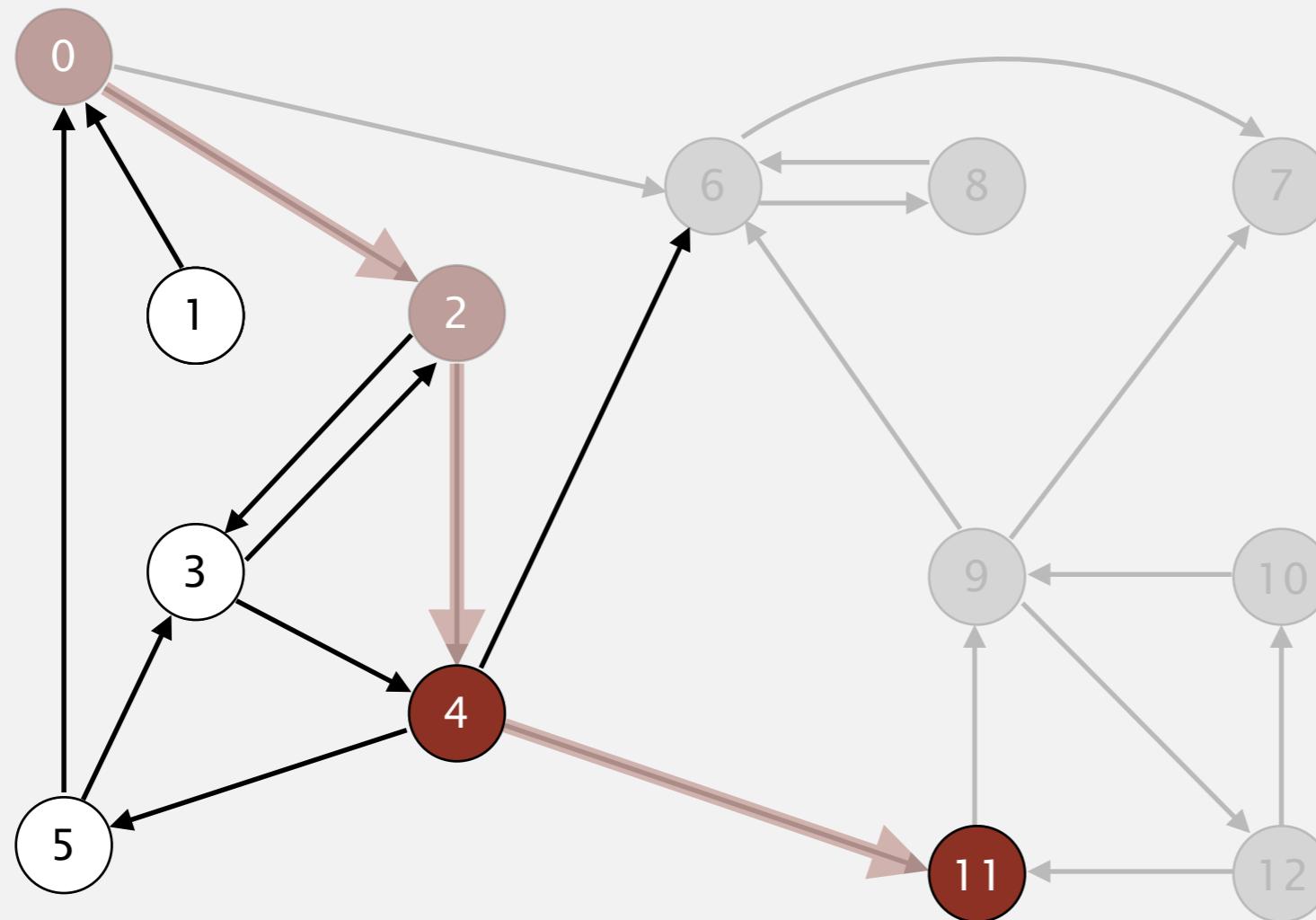
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

9 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

11 9 12 10 6 7 8



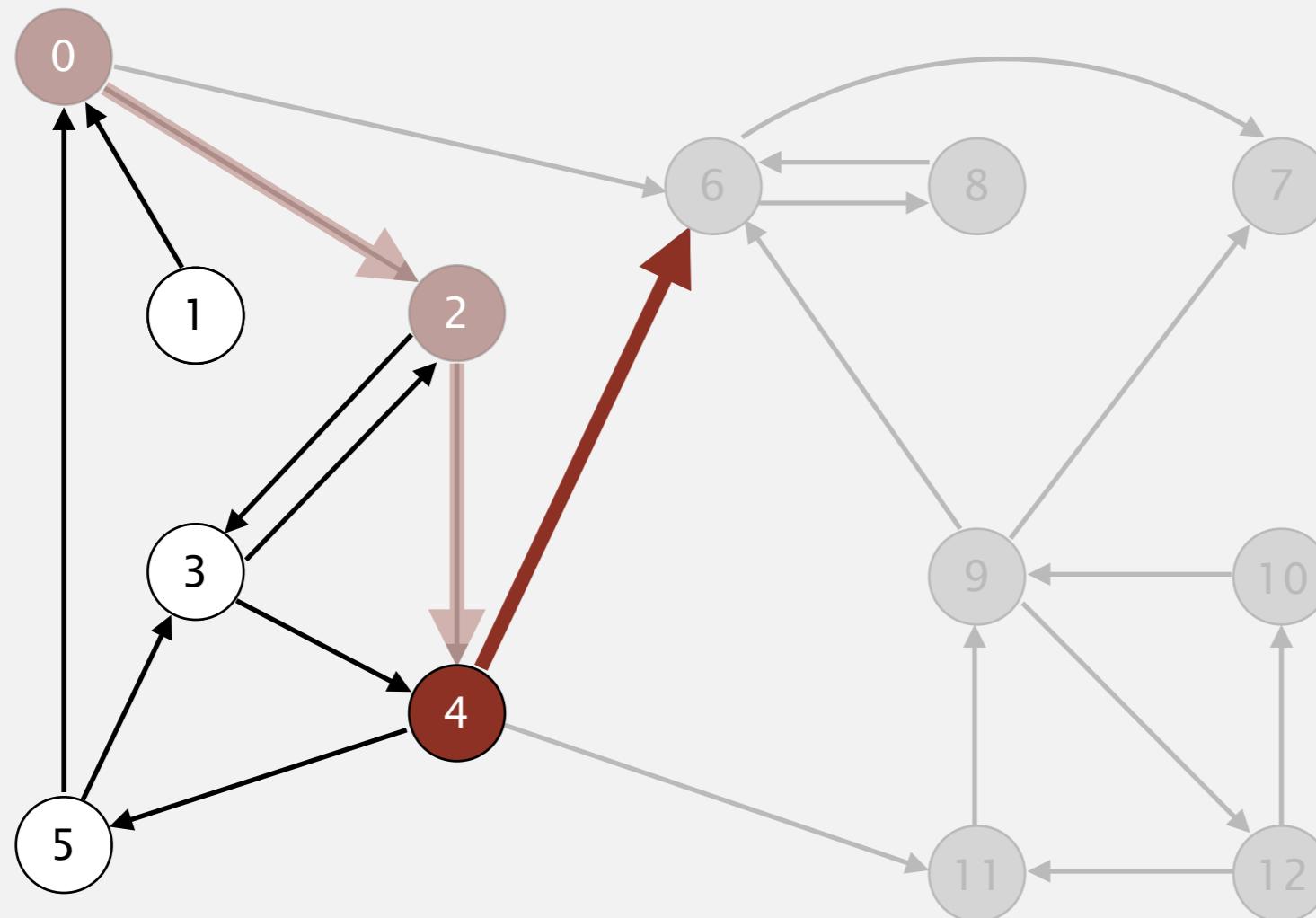
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

11 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

11 9 12 10 6 7 8



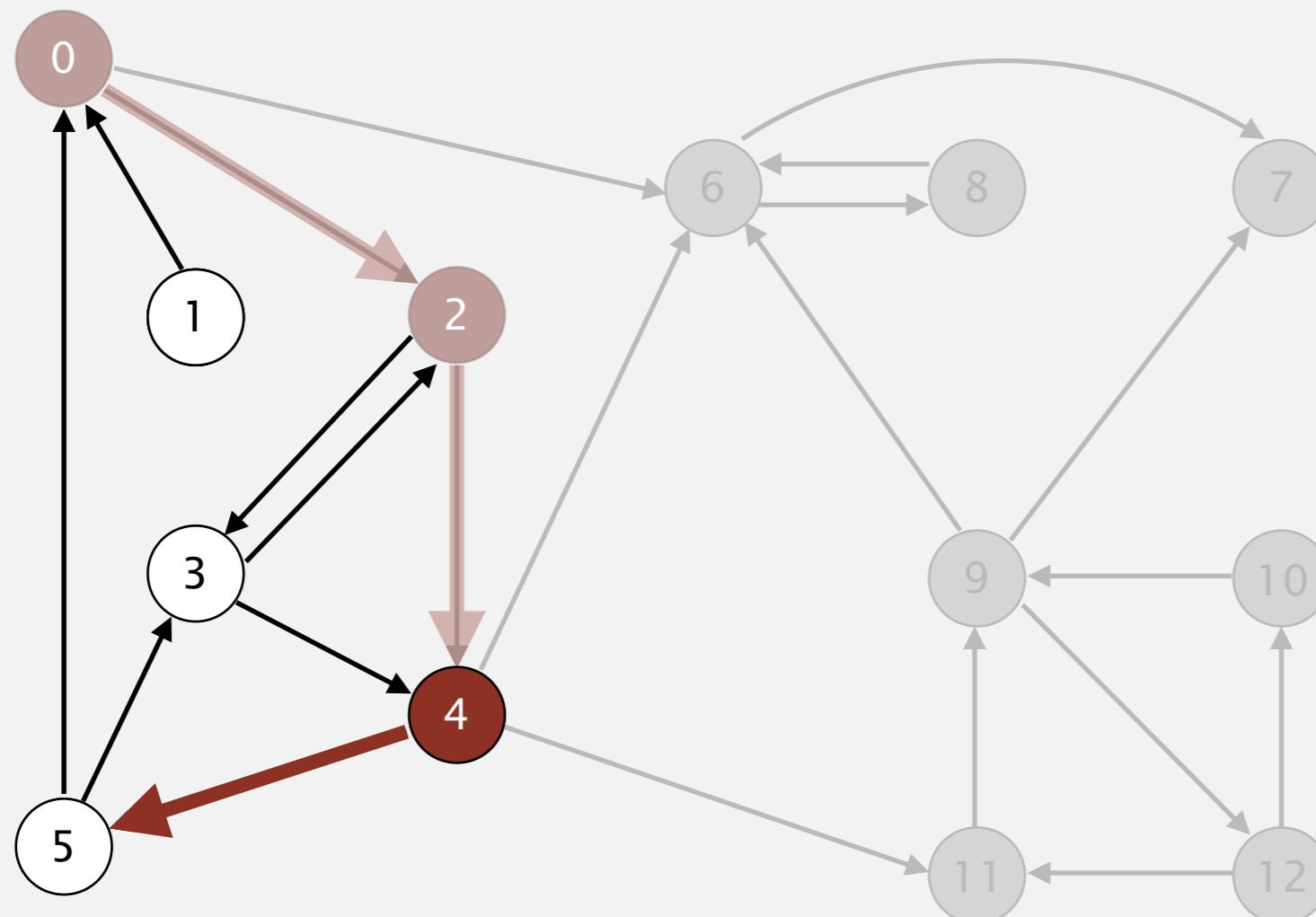
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 4: check 11, check 6, and check 5

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

11 9 12 10 6 7 8



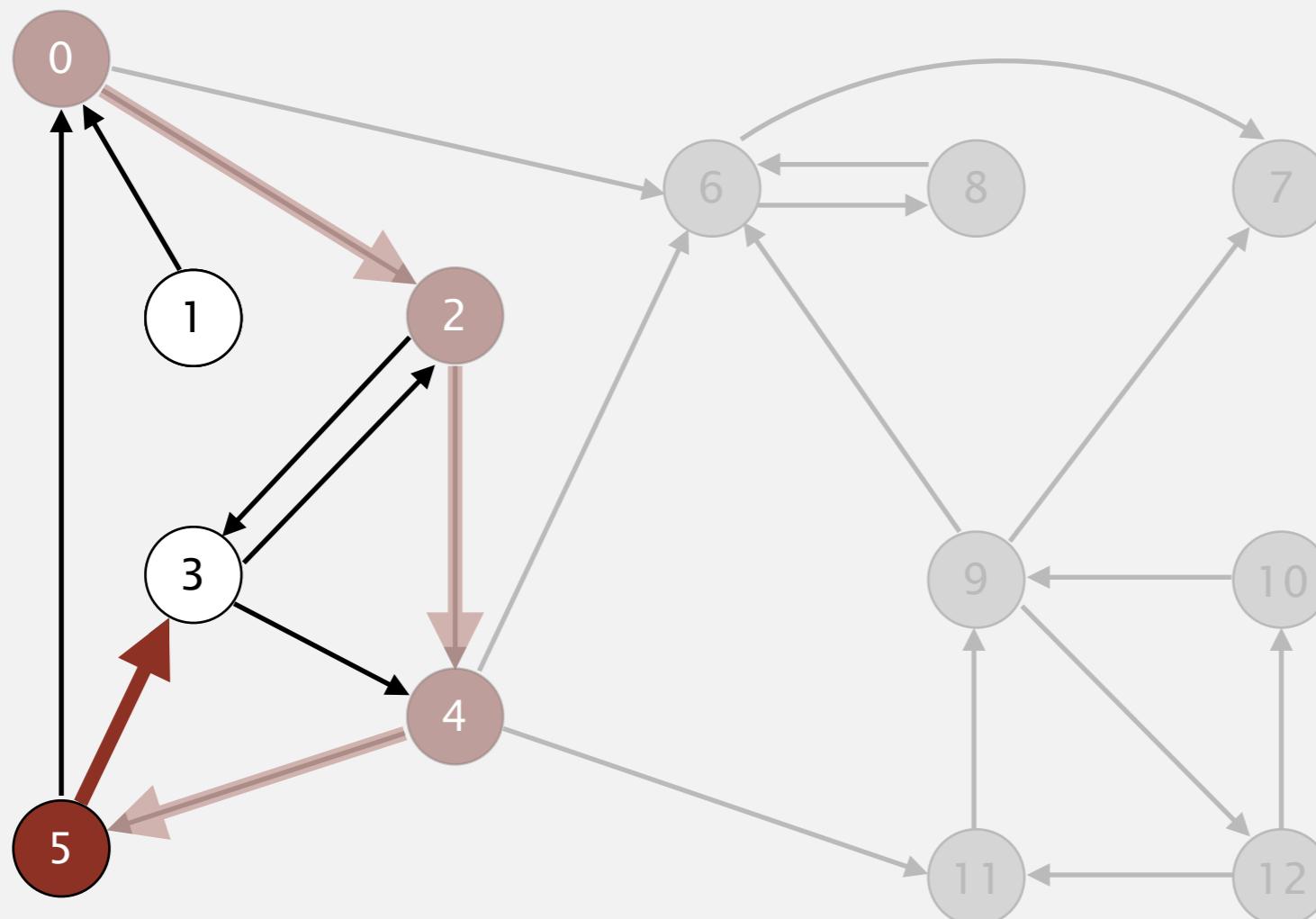
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	F
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 4: check 11, check 6, and check 5

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

11 9 12 10 6 7 8



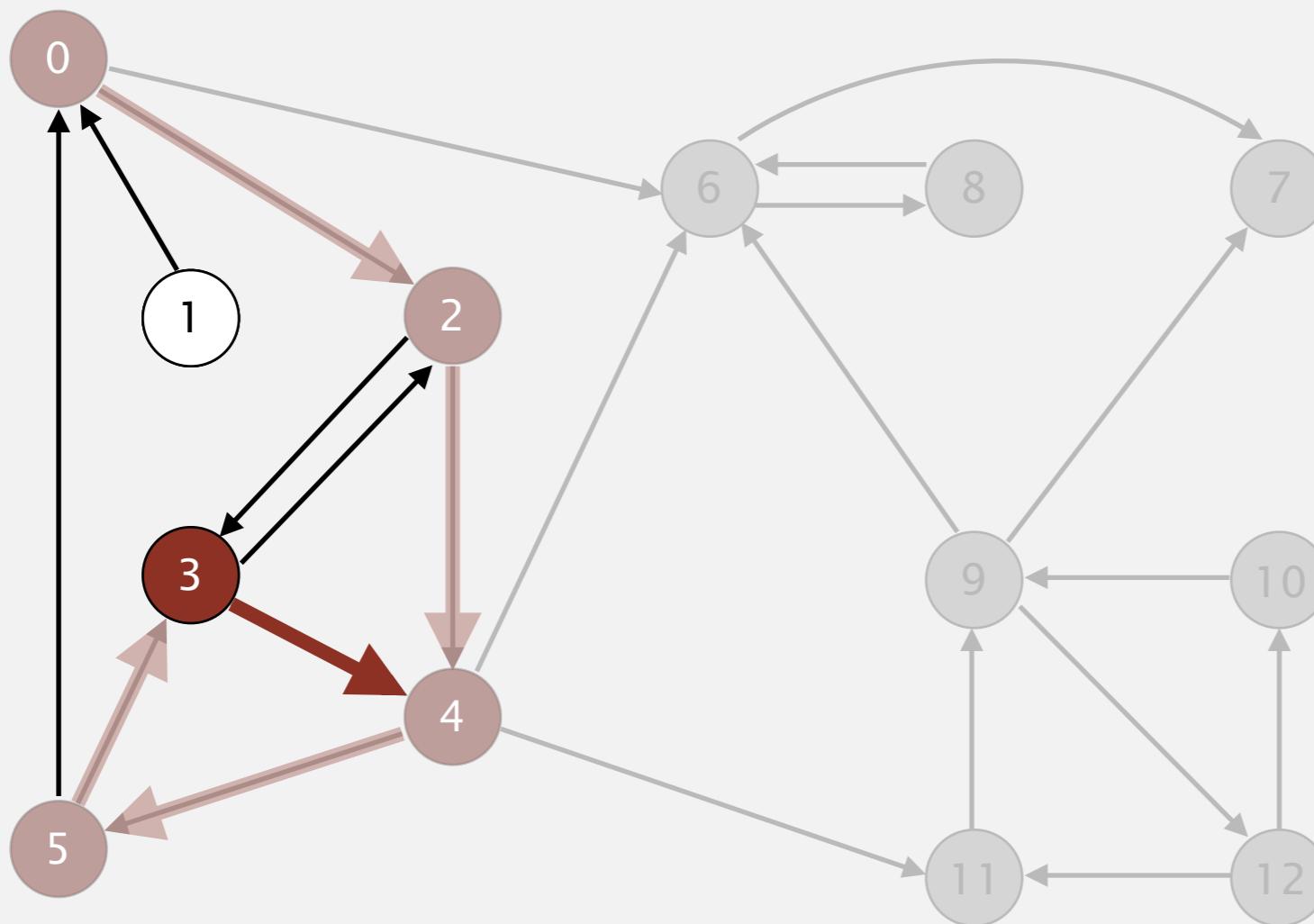
v	marked[v]
0	T
1	F
2	T
3	F
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 5: check 3 and check 0

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

11 9 12 10 6 7 8



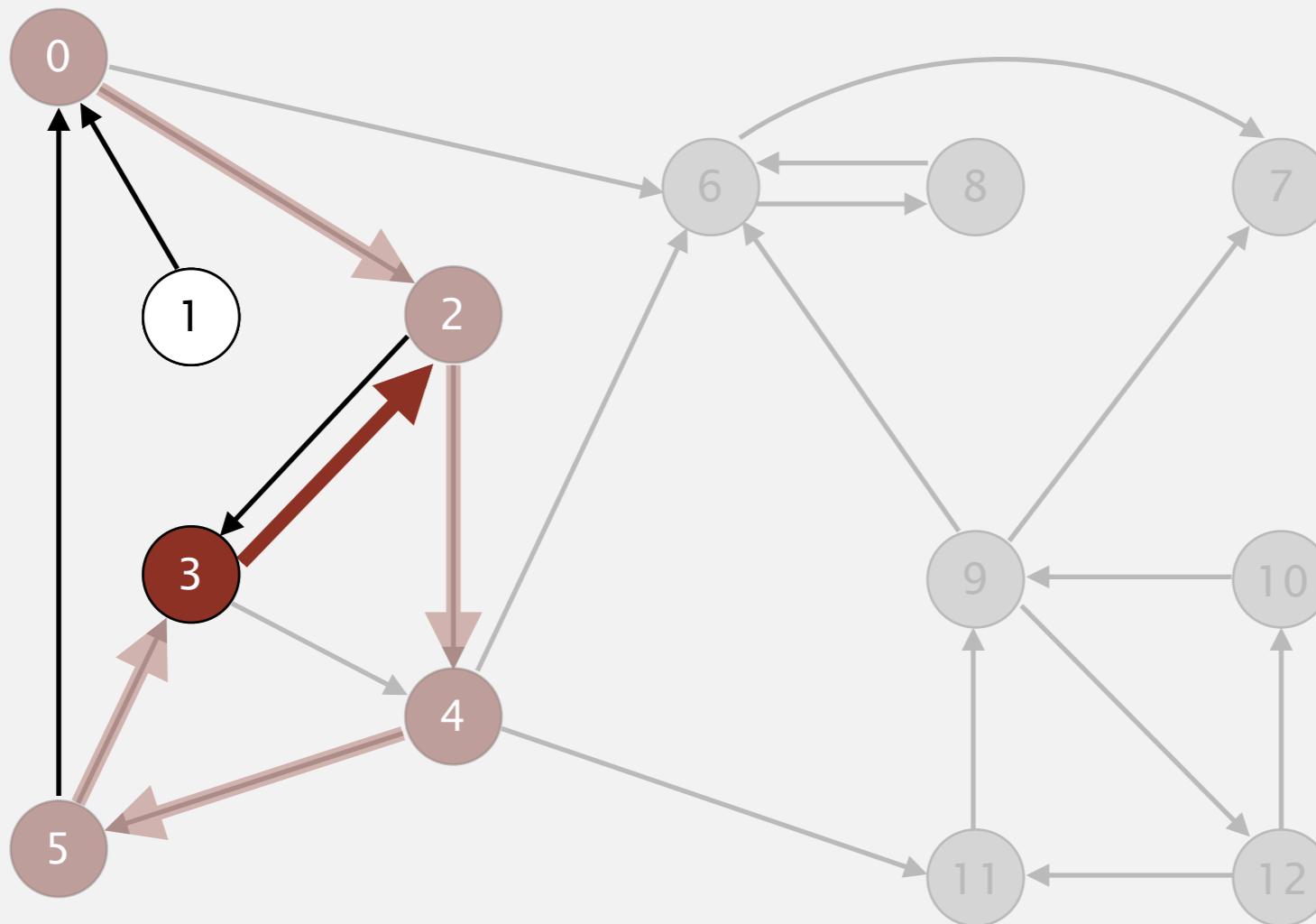
v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 3: check 4 and check 2

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

11 9 12 10 6 7 8

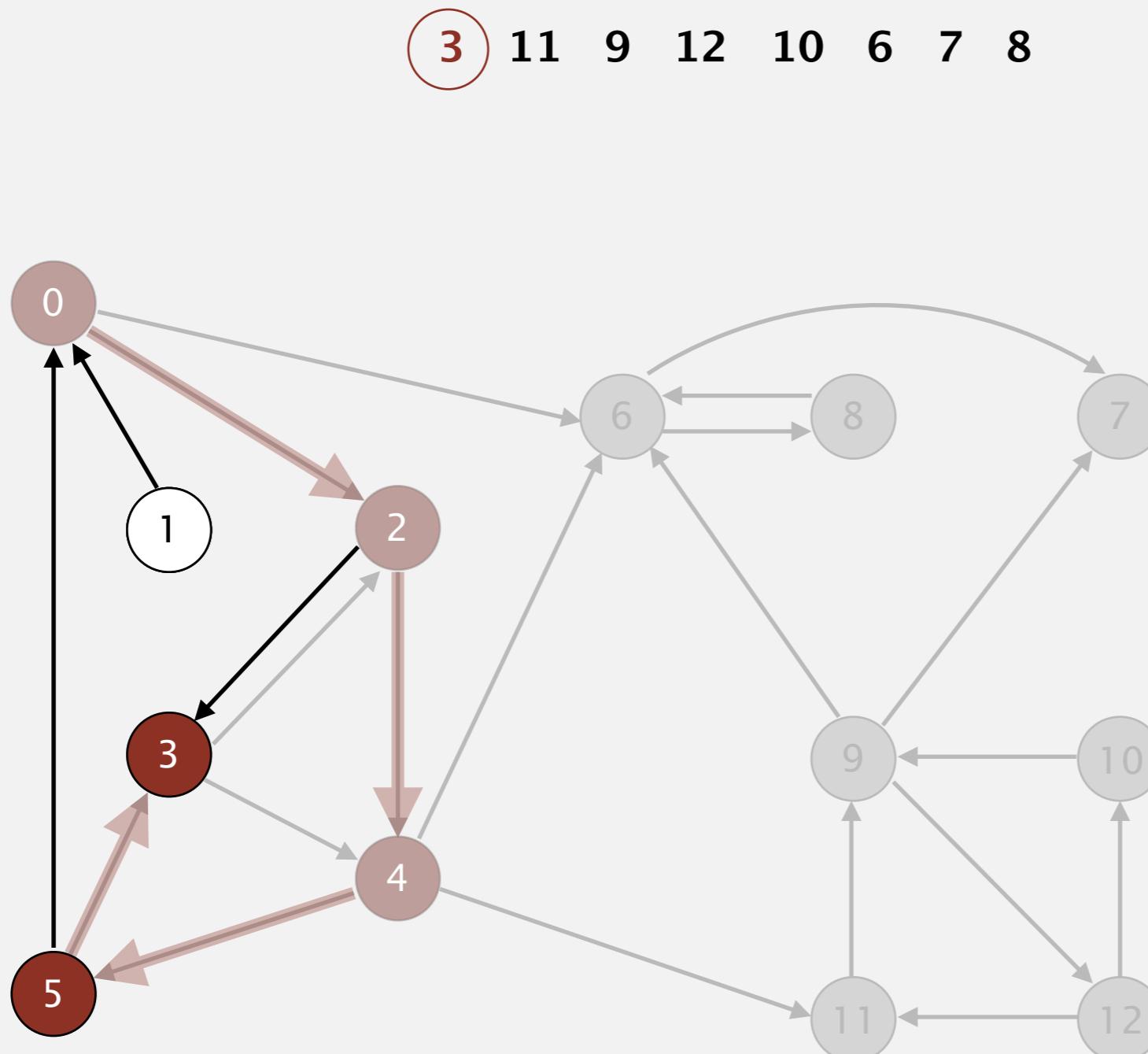


v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 3: check 4 and check 2

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .



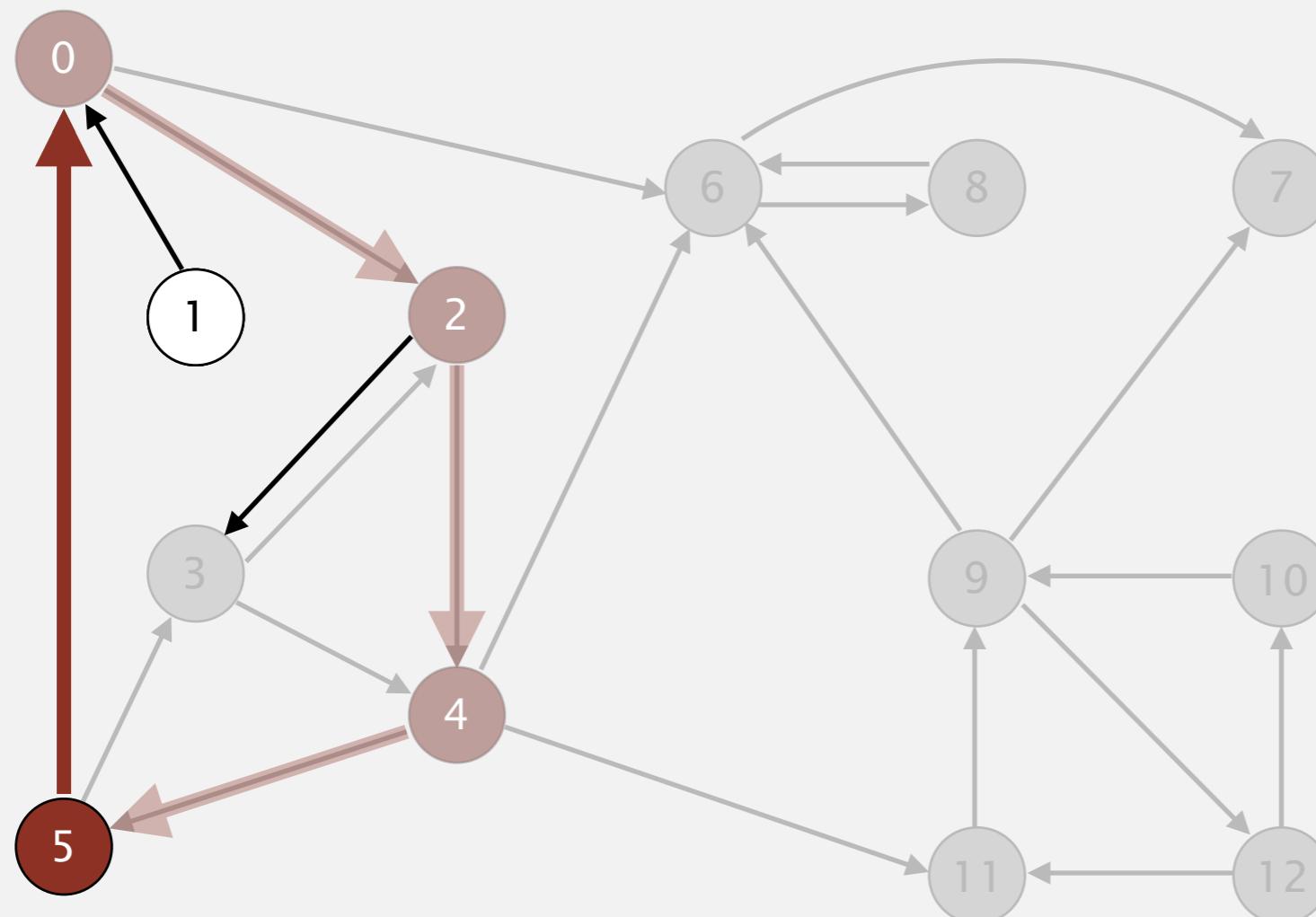
3 done

v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

3 11 9 12 10 6 7 8



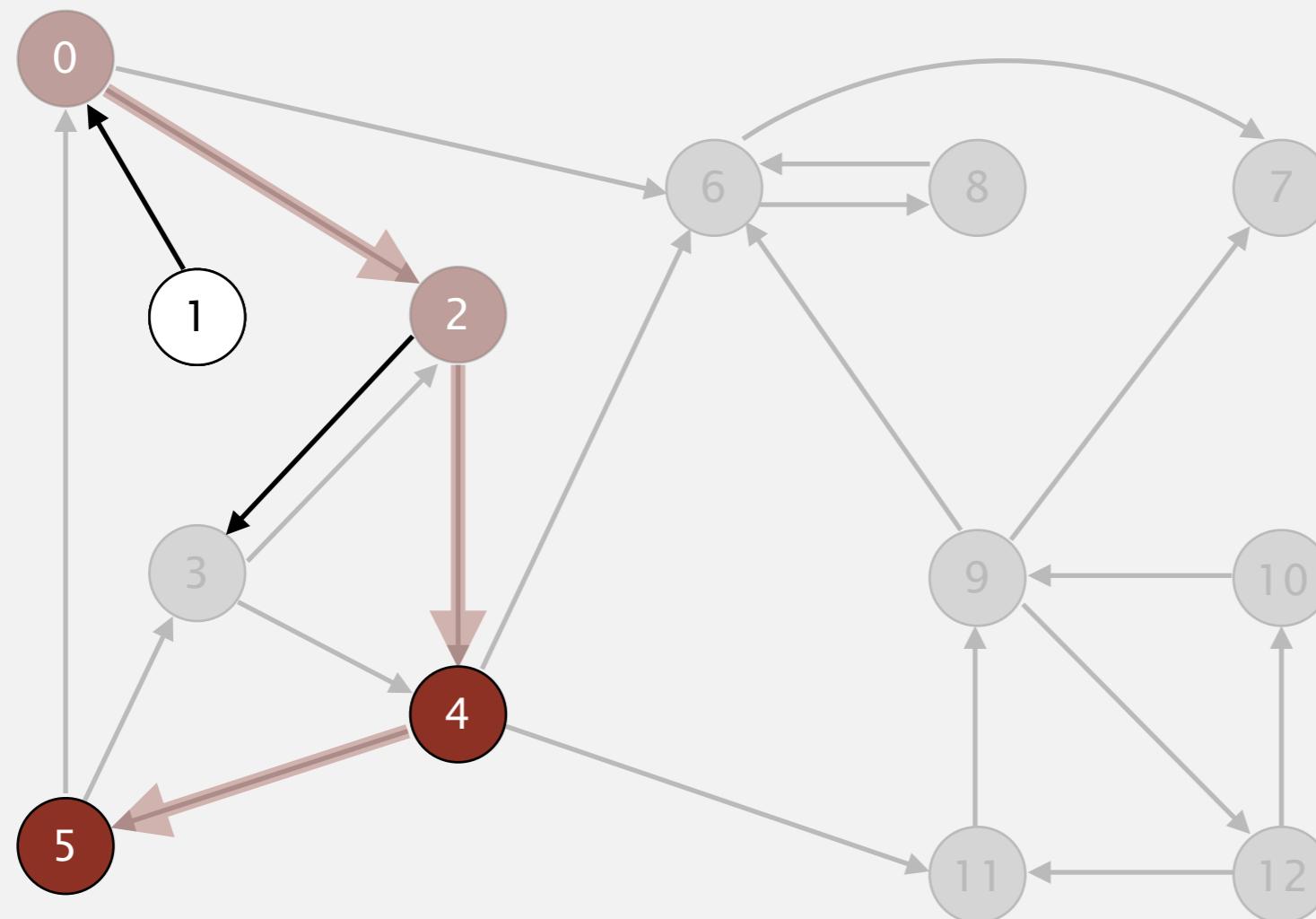
v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 5: check 3 and check 0

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

5 3 11 9 12 10 6 7 8



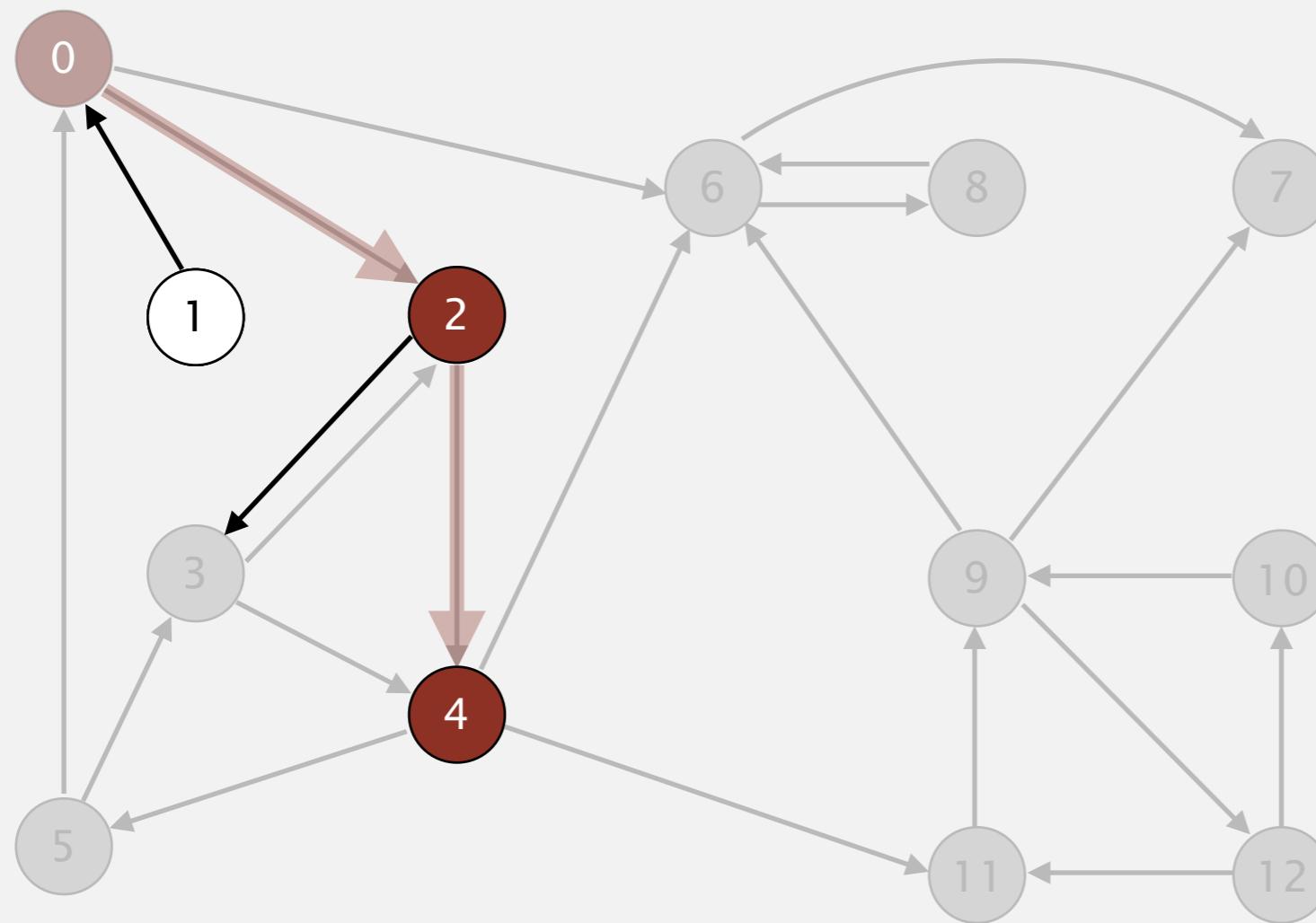
v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

5 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

4 5 3 11 9 12 10 6 7 8

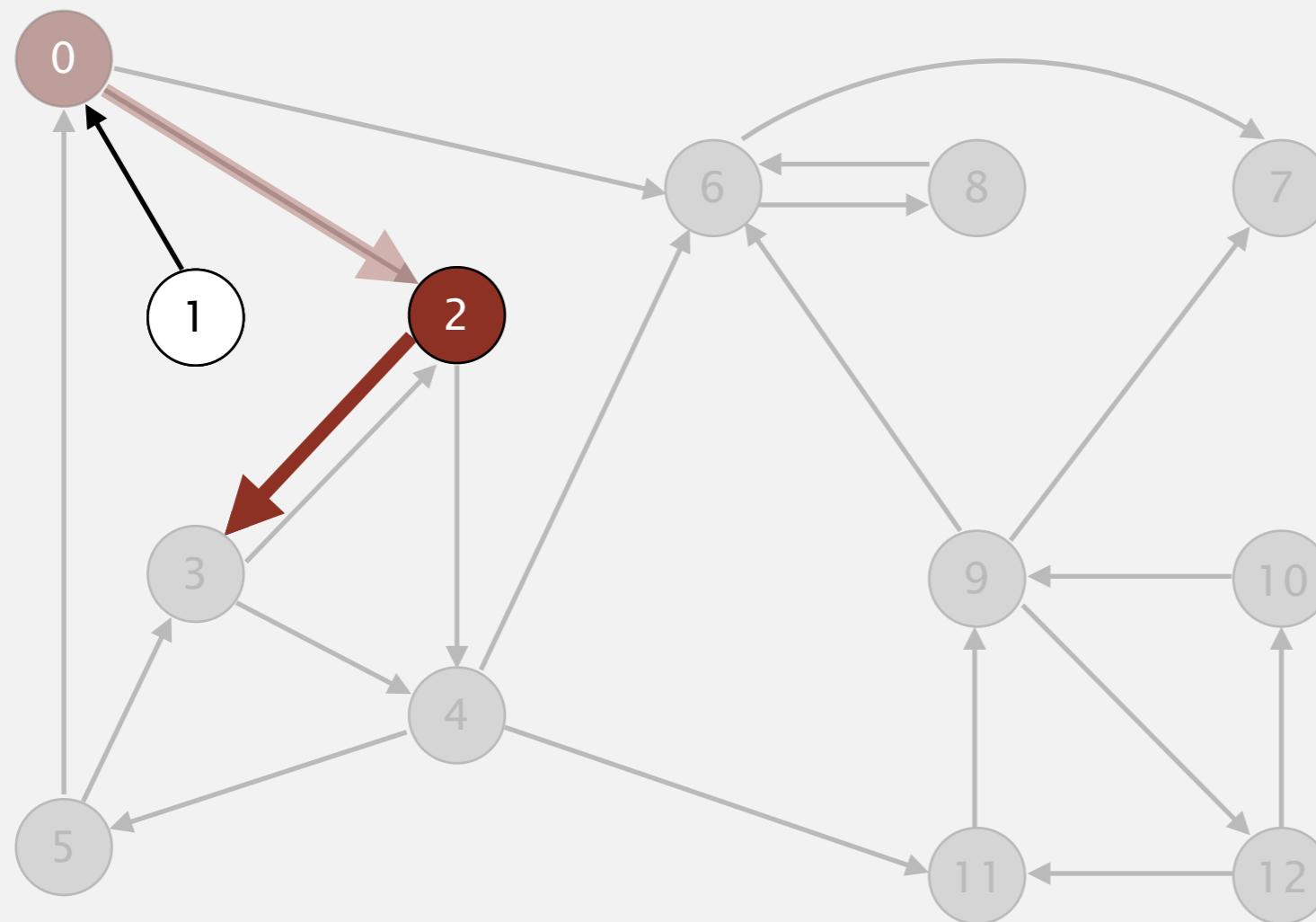


v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

4 5 3 11 9 12 10 6 7 8



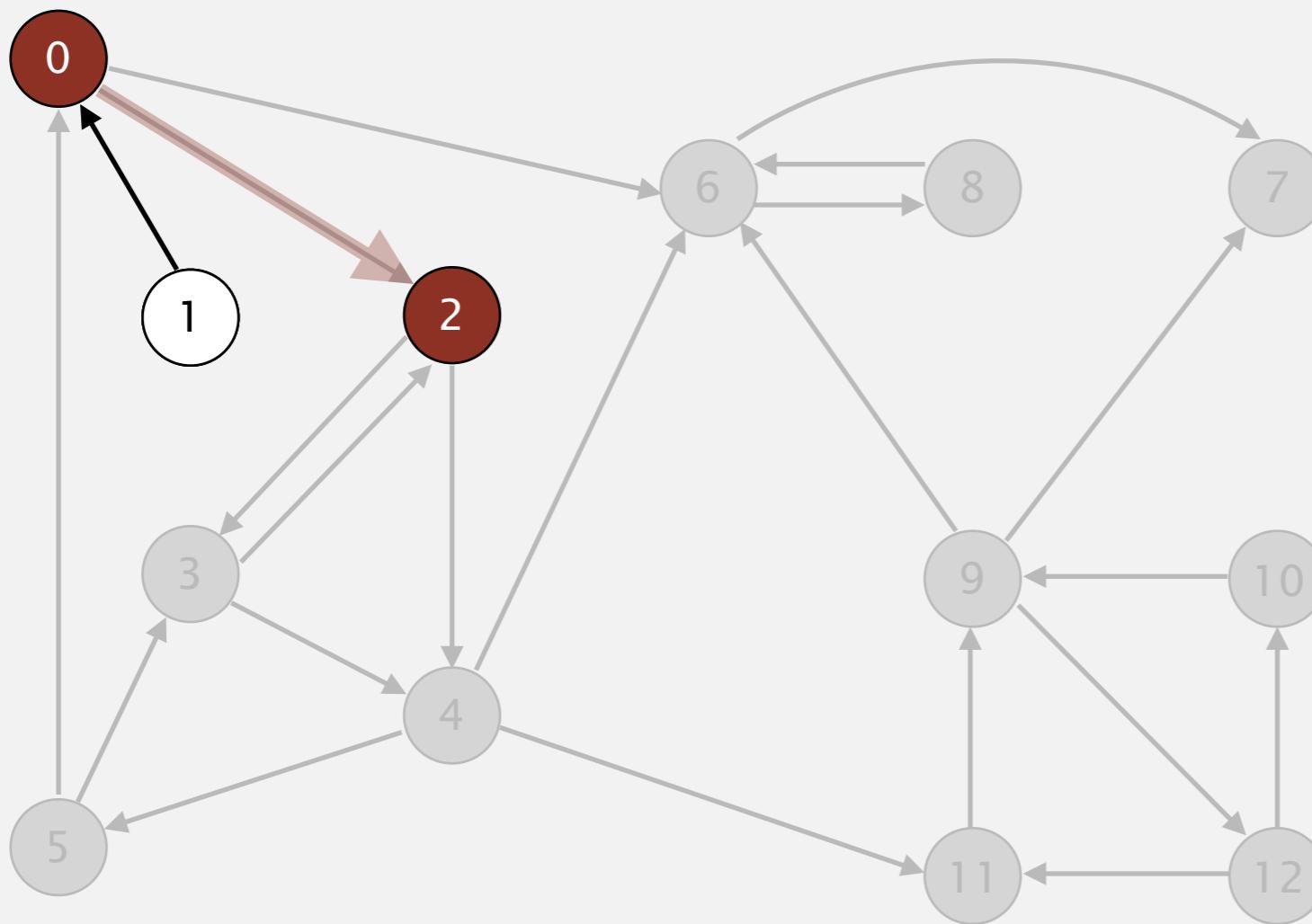
v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

visit 2: check 4 and check 3

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

2 4 5 3 11 9 12 10 6 7 8



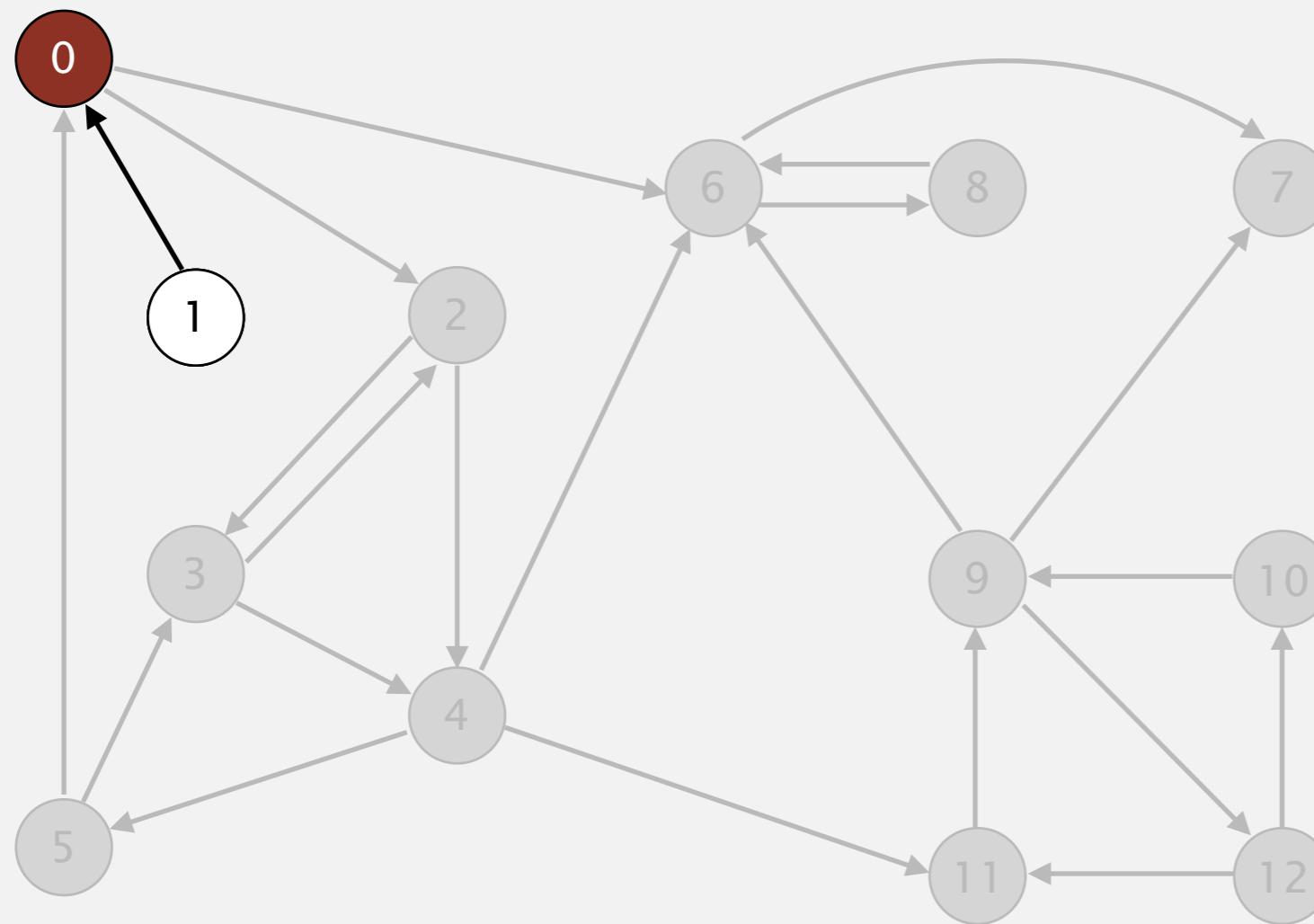
2 done

v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

0 2 4 5 3 11 9 12 10 6 7 8



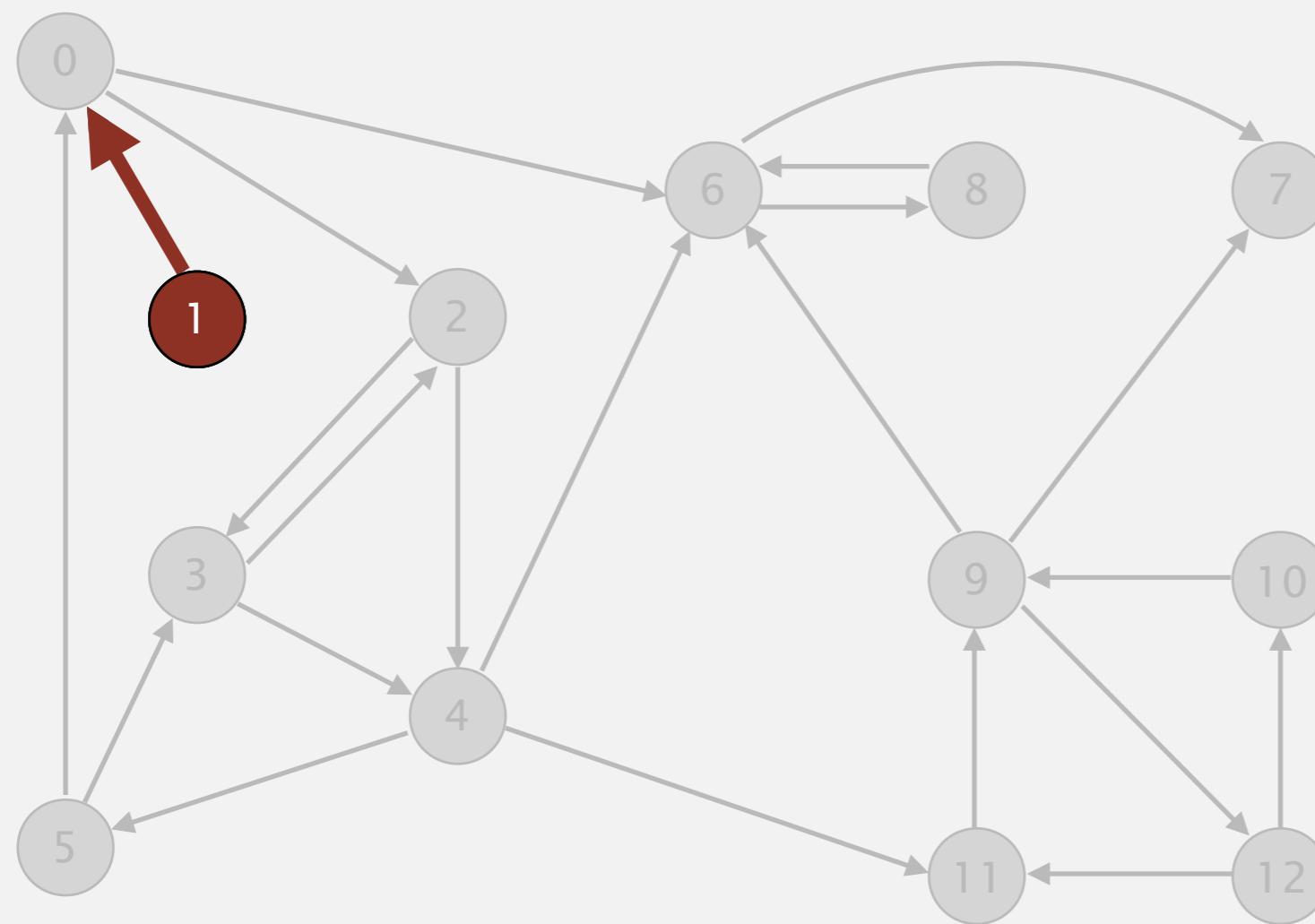
v	marked[v]
0	T
1	F
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

0 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

0 2 4 5 3 11 9 12 10 6 7 8

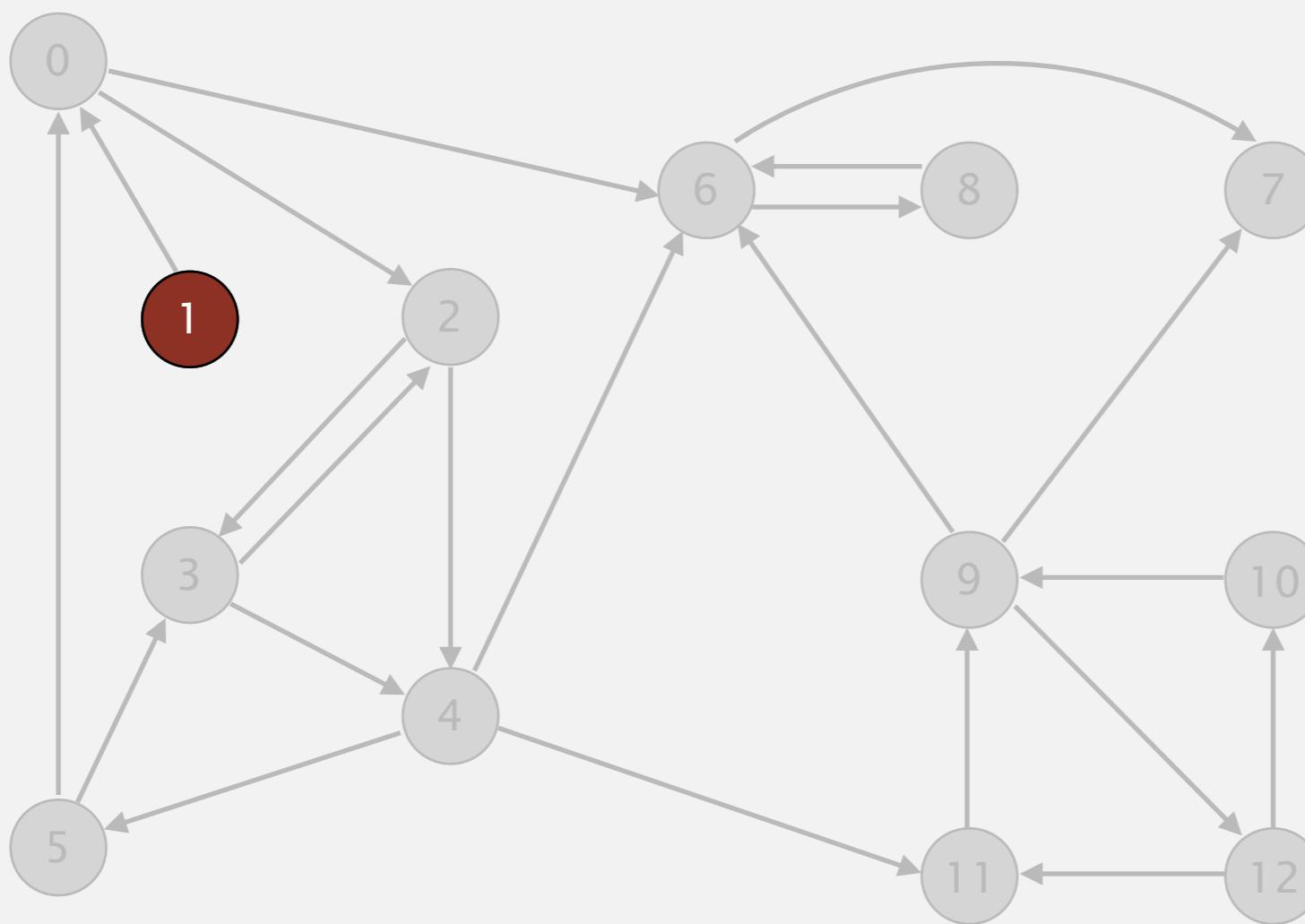


v	marked[v]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

① 0 2 4 5 3 11 9 12 10 6 7 8



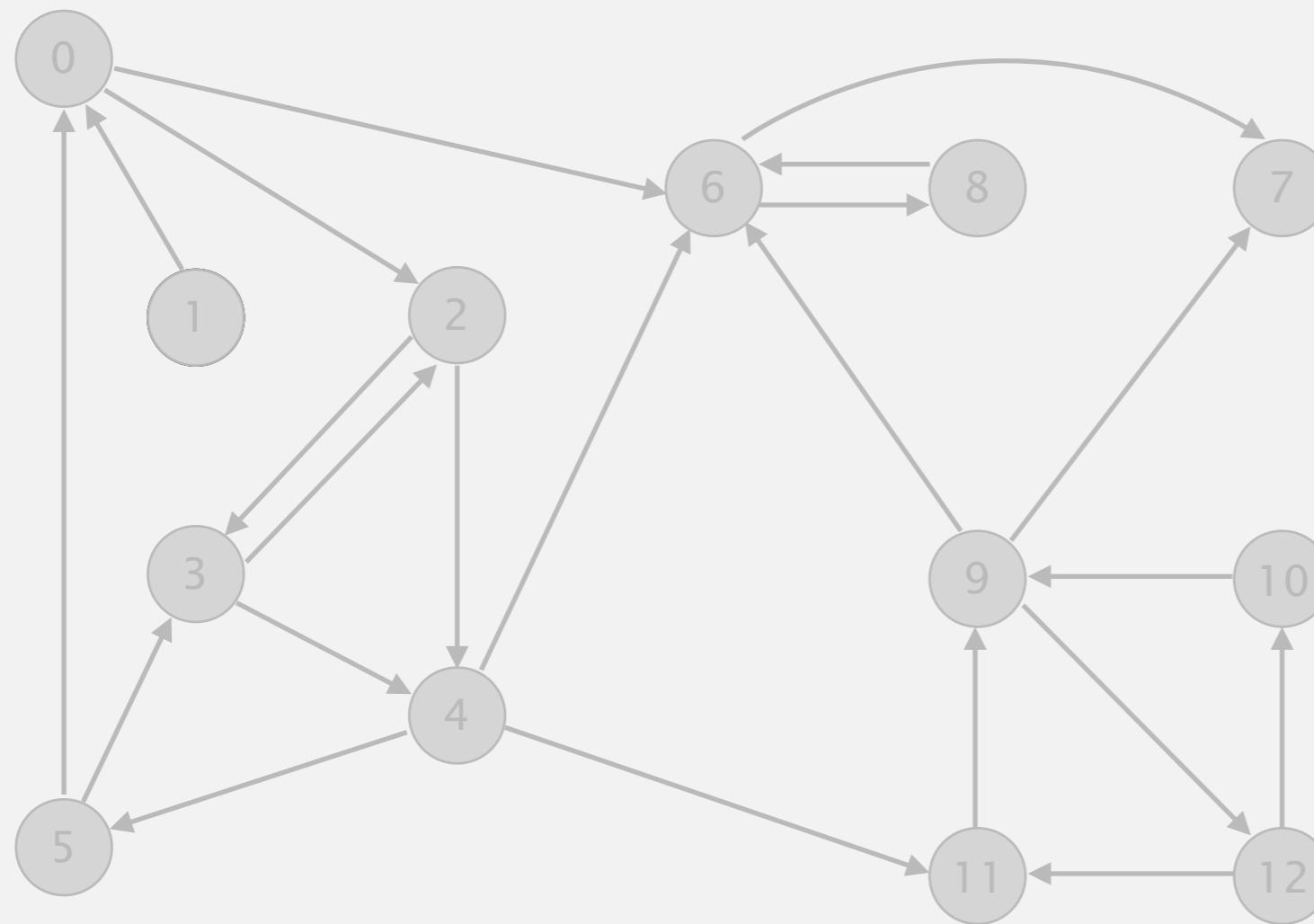
v	marked[v]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

1 done

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



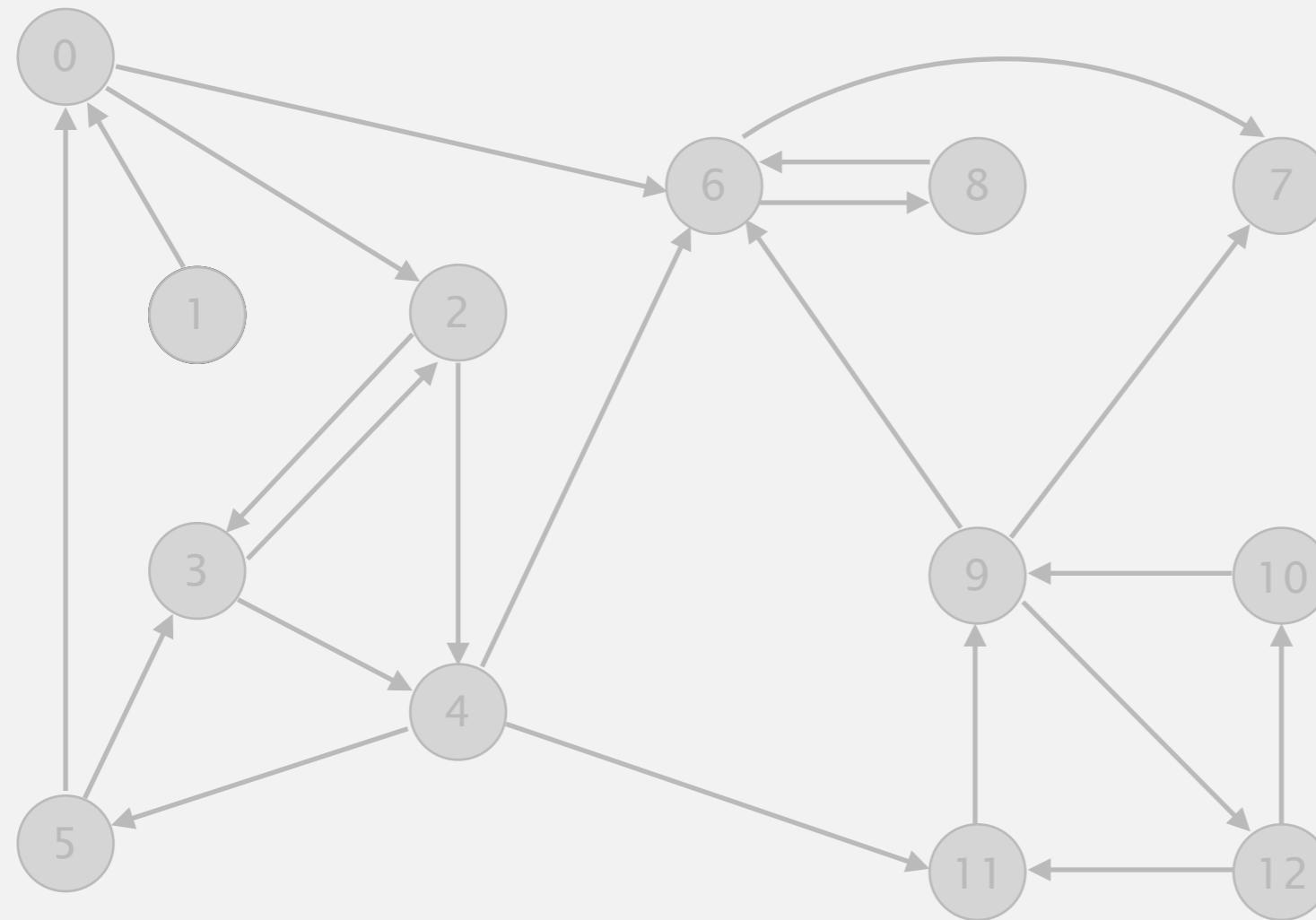
v	marked[v]
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T
10	T
11	T
12	T

check 2 3 4 5 6 7 8 9 10 11 12

Kosaraju-Sharir

Phase I. Compute reverse postorder in G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8

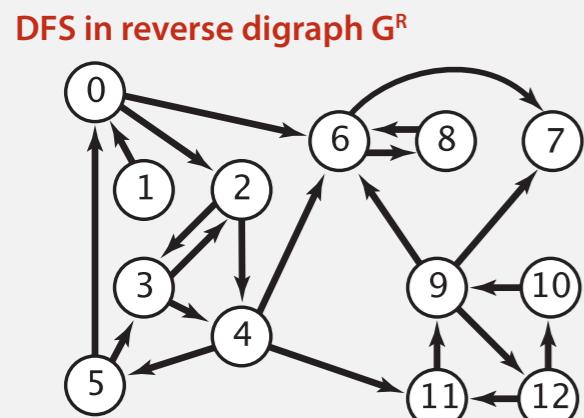


reverse digraph G^R

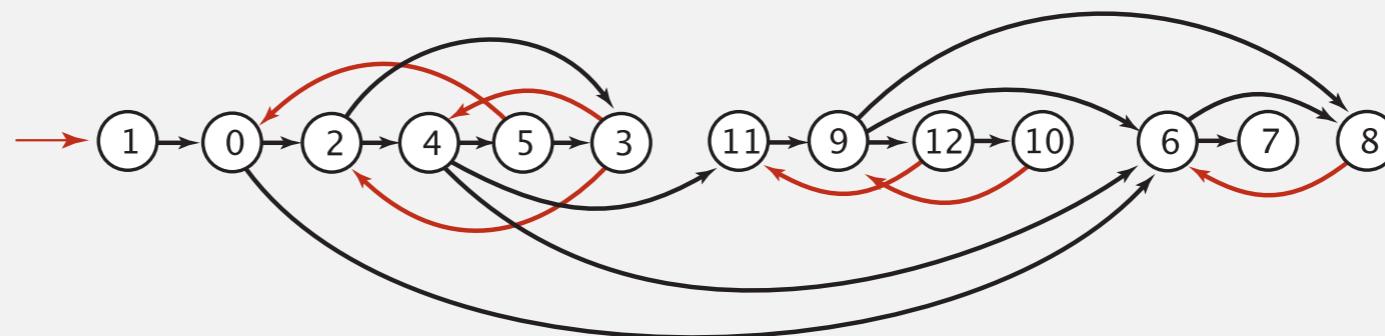
Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G , considering vertices in order given by first DFS.



check unmarked vertices in the order
0 1 2 3 4 5 6 7 8 9 10 11 12



reverse postorder for use in second dfs()
1 0 2 4 5 3 11 9 12 10 6 7 8

```
dfs(0)
  dfs(6)
    dfs(8)
      | check 6
      8 done
      dfs(7)
      7 done
    6 done
    dfs(2)
      dfs(4)
        dfs(11)
          dfs(9)
            dfs(12)
              check 11
              dfs(10)
                | check 9
                10 done
              12 done
              check 7
              check 6
...

```

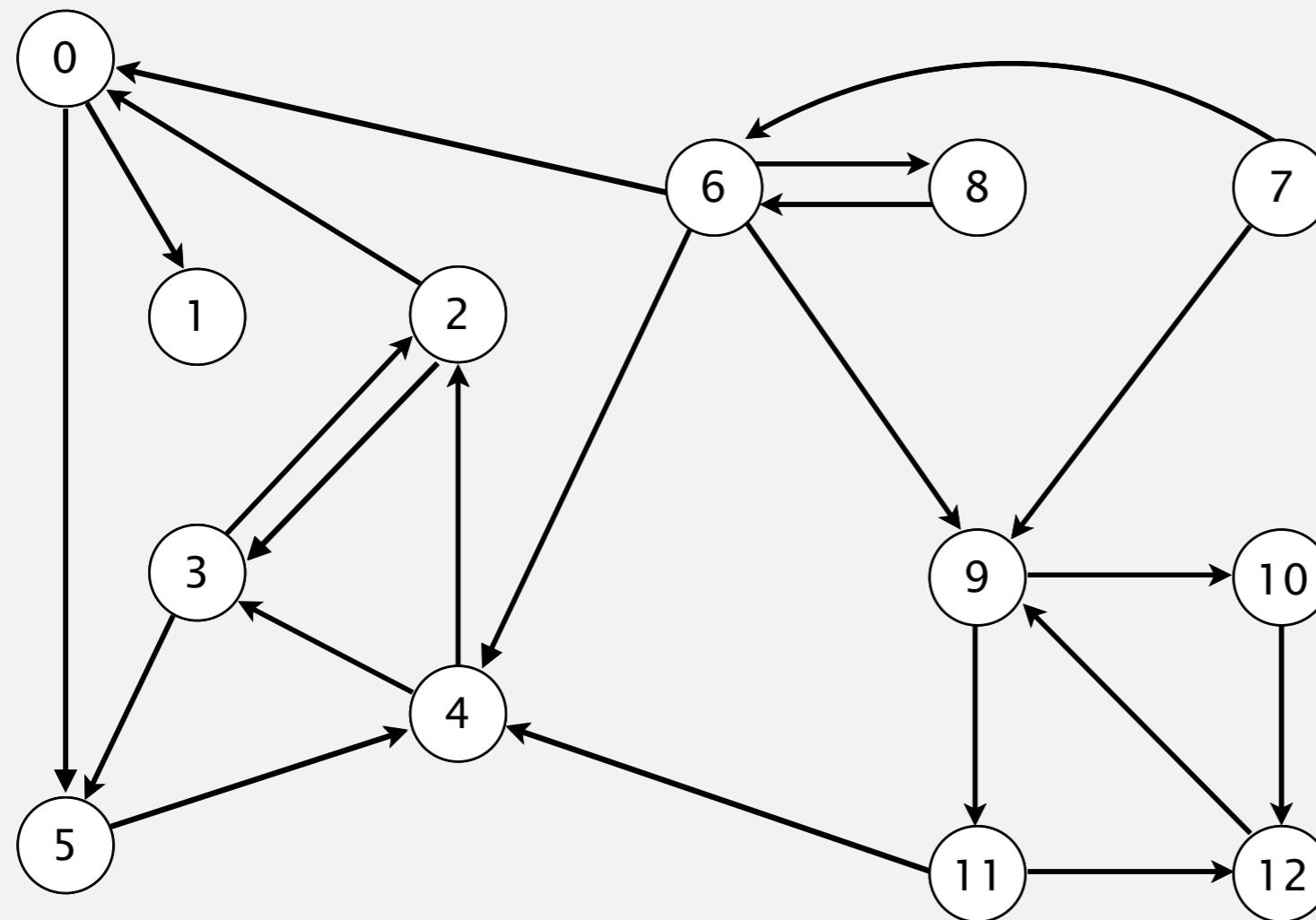
KOSARAJU'S ALGORITHM

- ▶ DFS in reverse graph
- ▶ DFS in original graph

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8

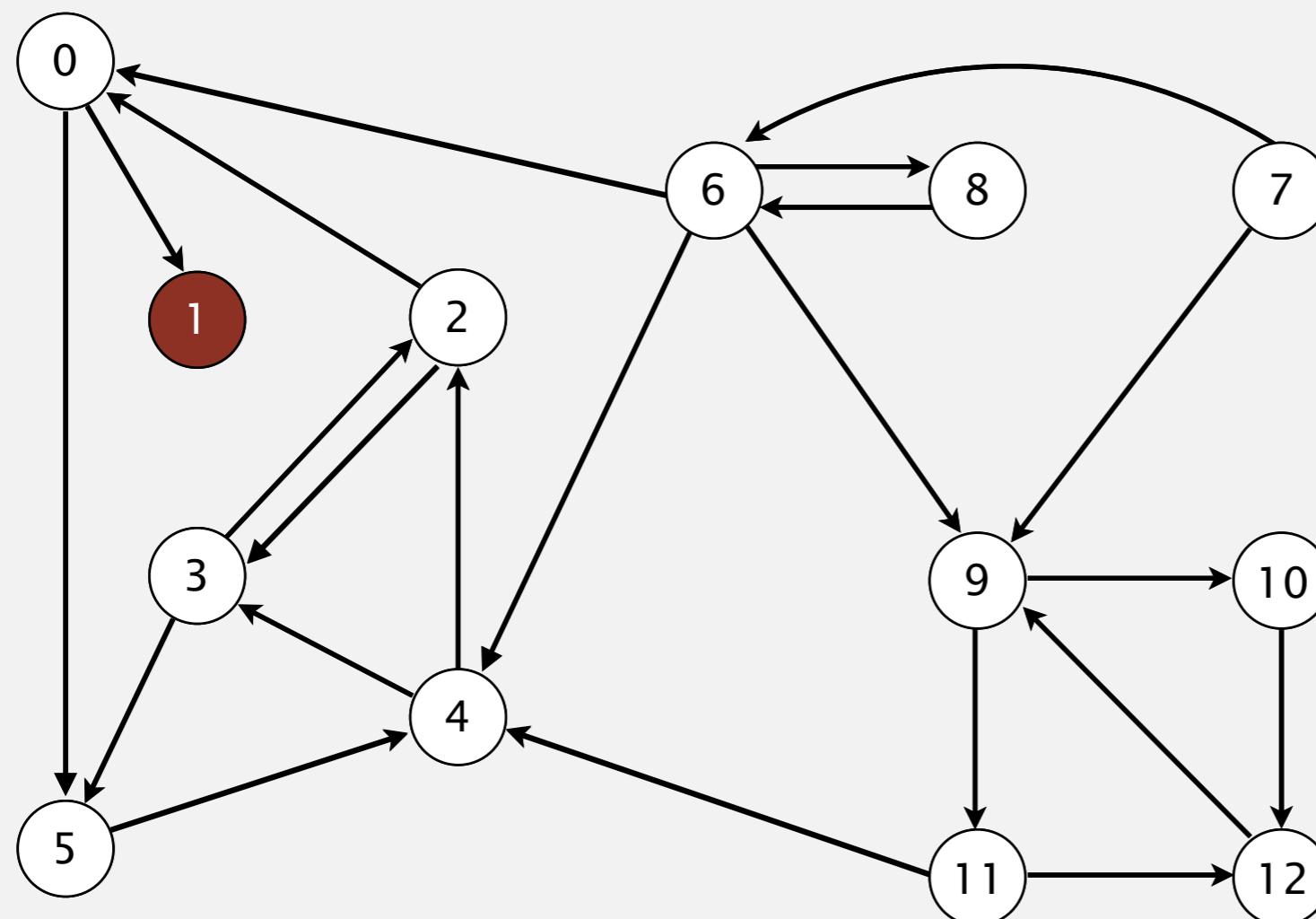


original digraph G

v	scc[v]
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . $\begin{array}{ccccccccccccc} 1 & 0 & 2 & 4 & 5 & 3 & 11 & 9 & 12 & 10 & 6 & 7 & 8 \end{array}$

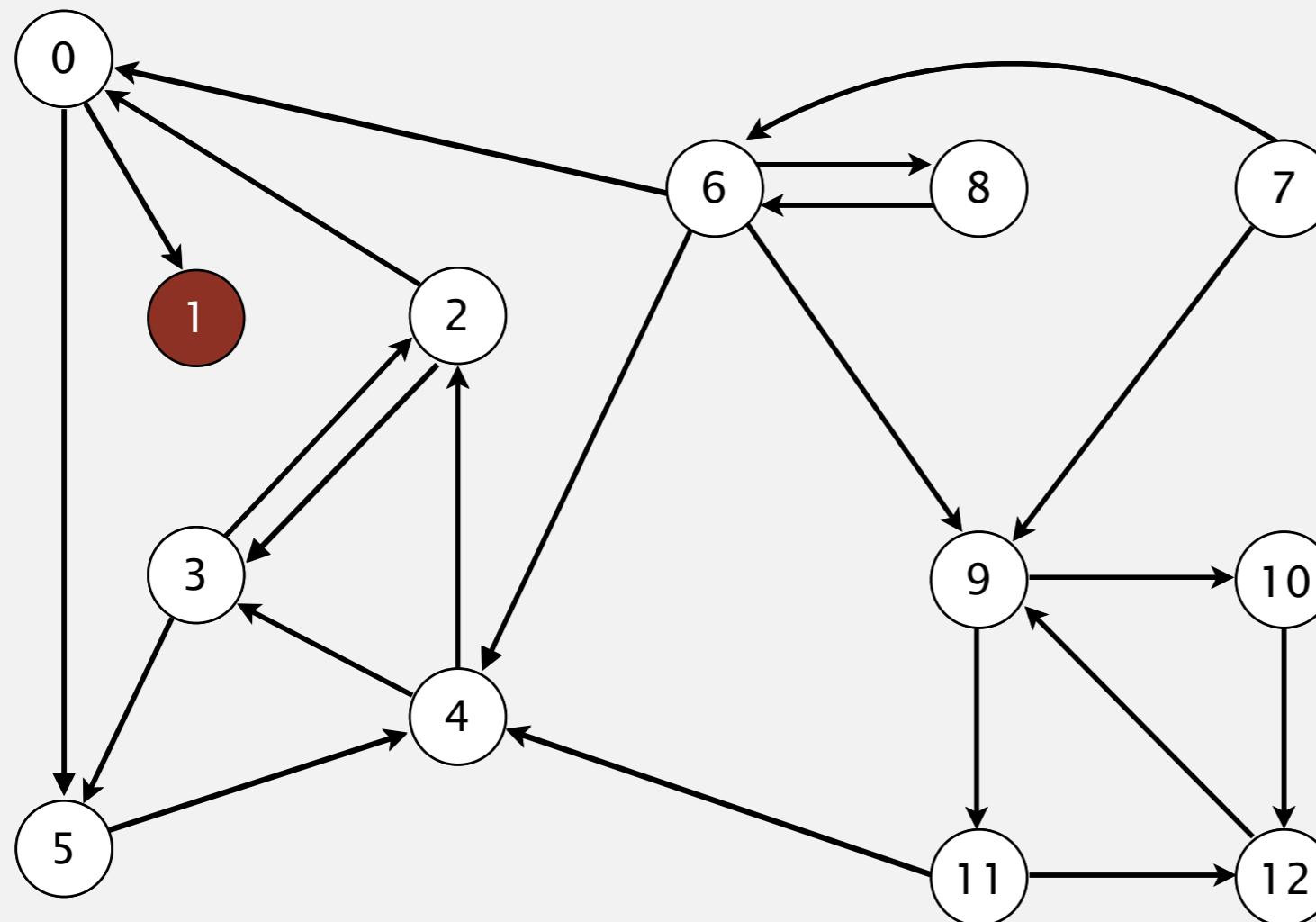


v	$scc[v]$
0	-
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 1

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

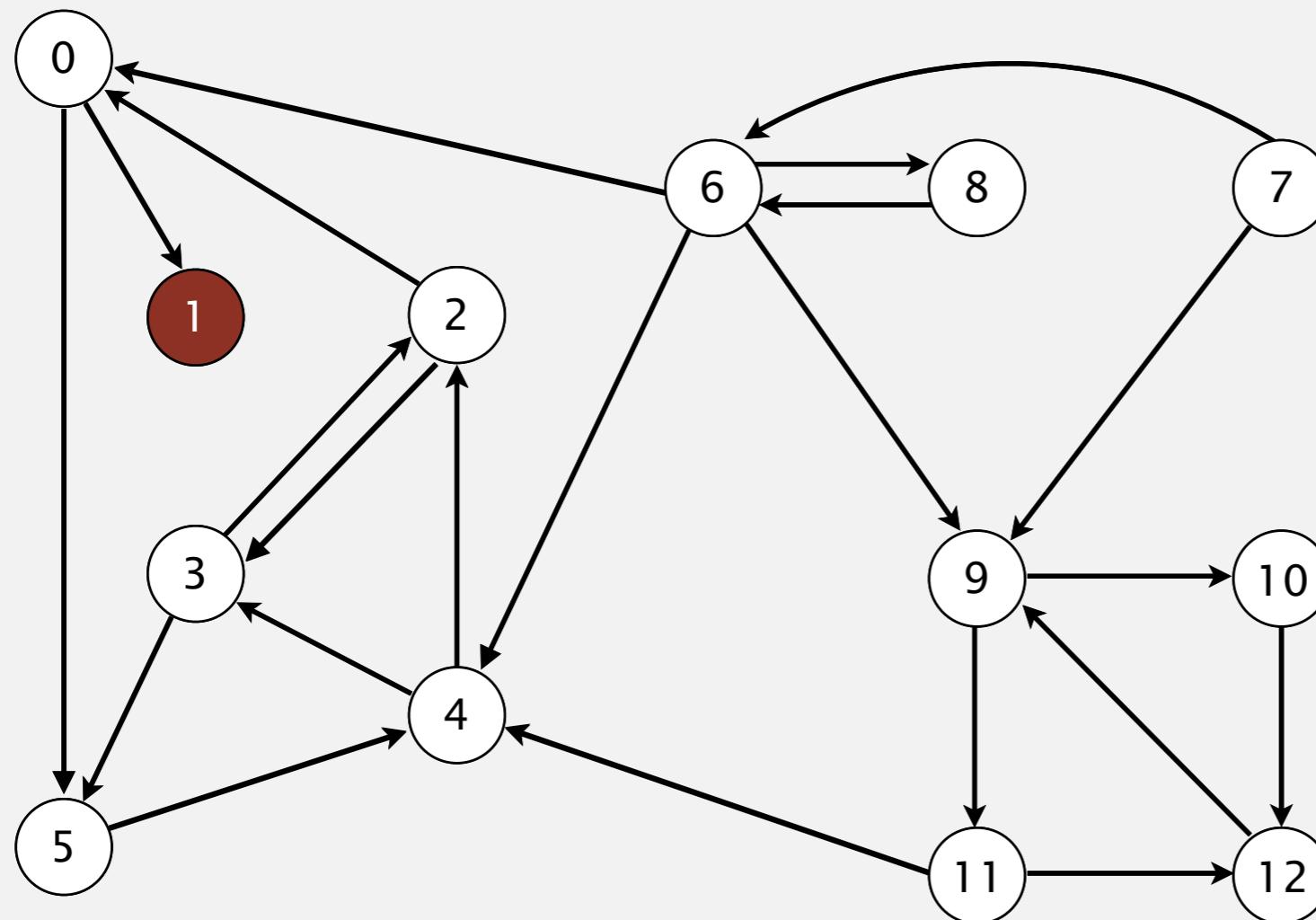


v	scc[v]
0	-
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

1 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

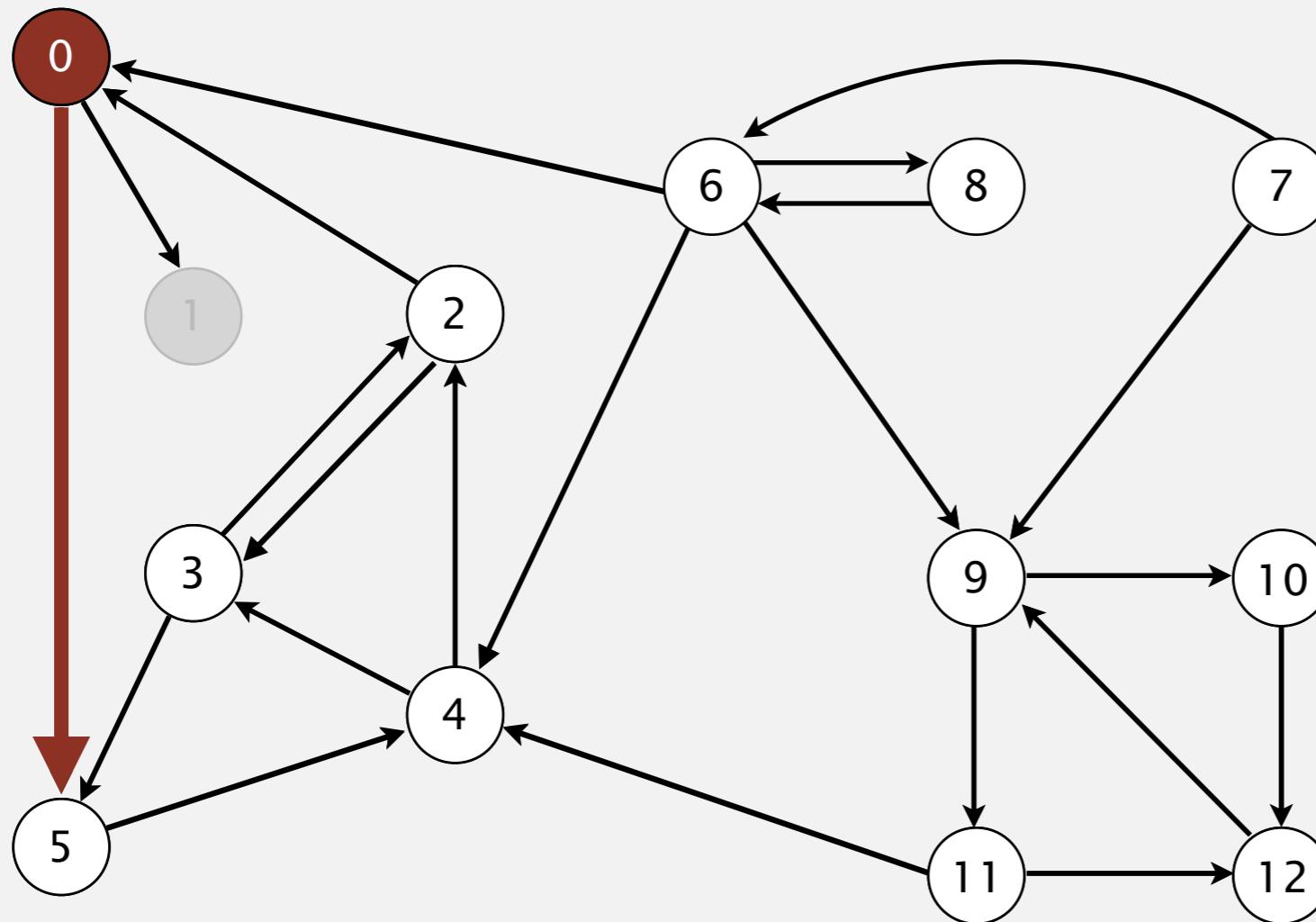


v	scc[v]
0	-
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

strong component: 1

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

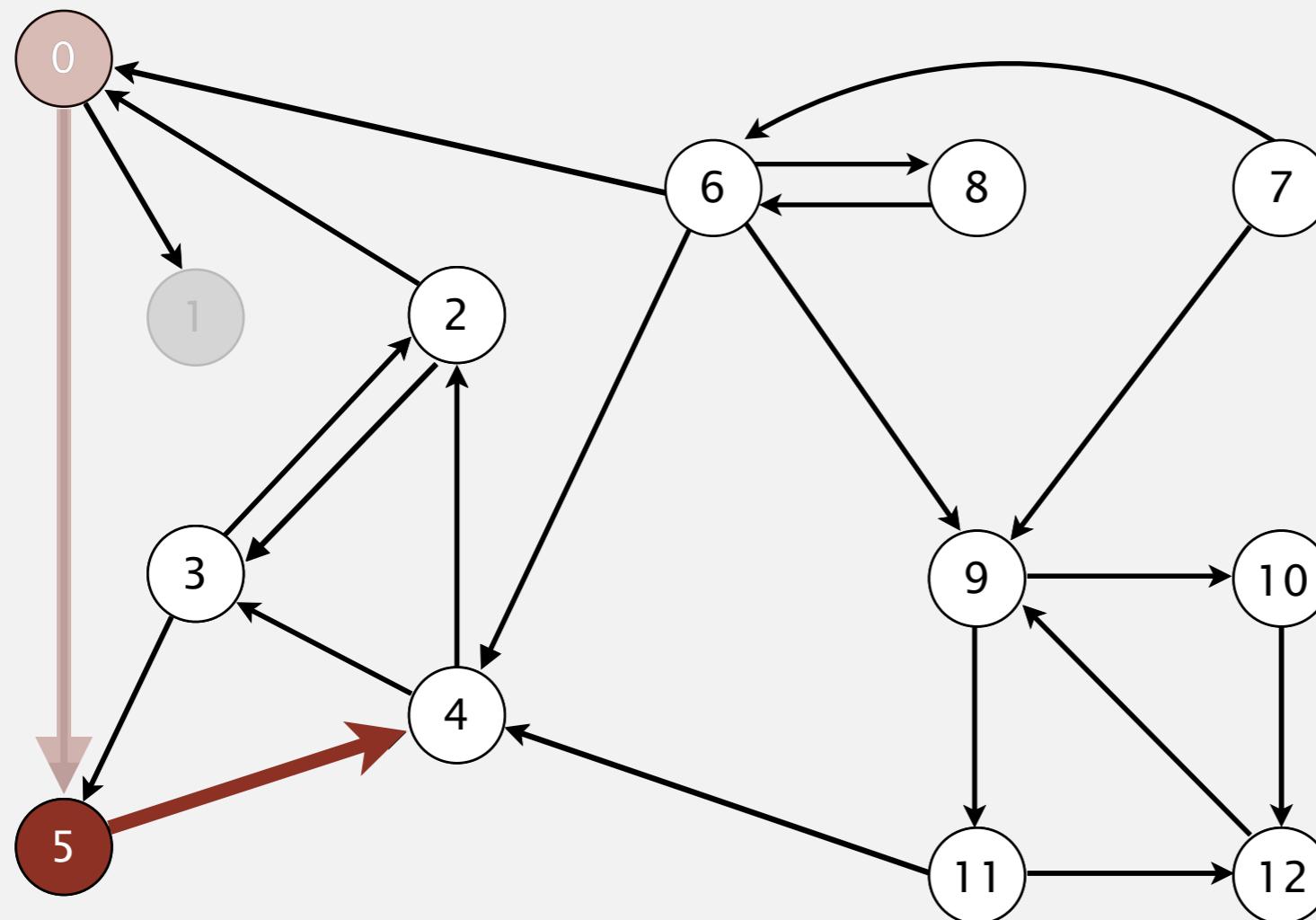


visit 0: check 5 and check 1

v	scc[v]
0	1
1	0
2	-
3	-
4	-
5	-
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

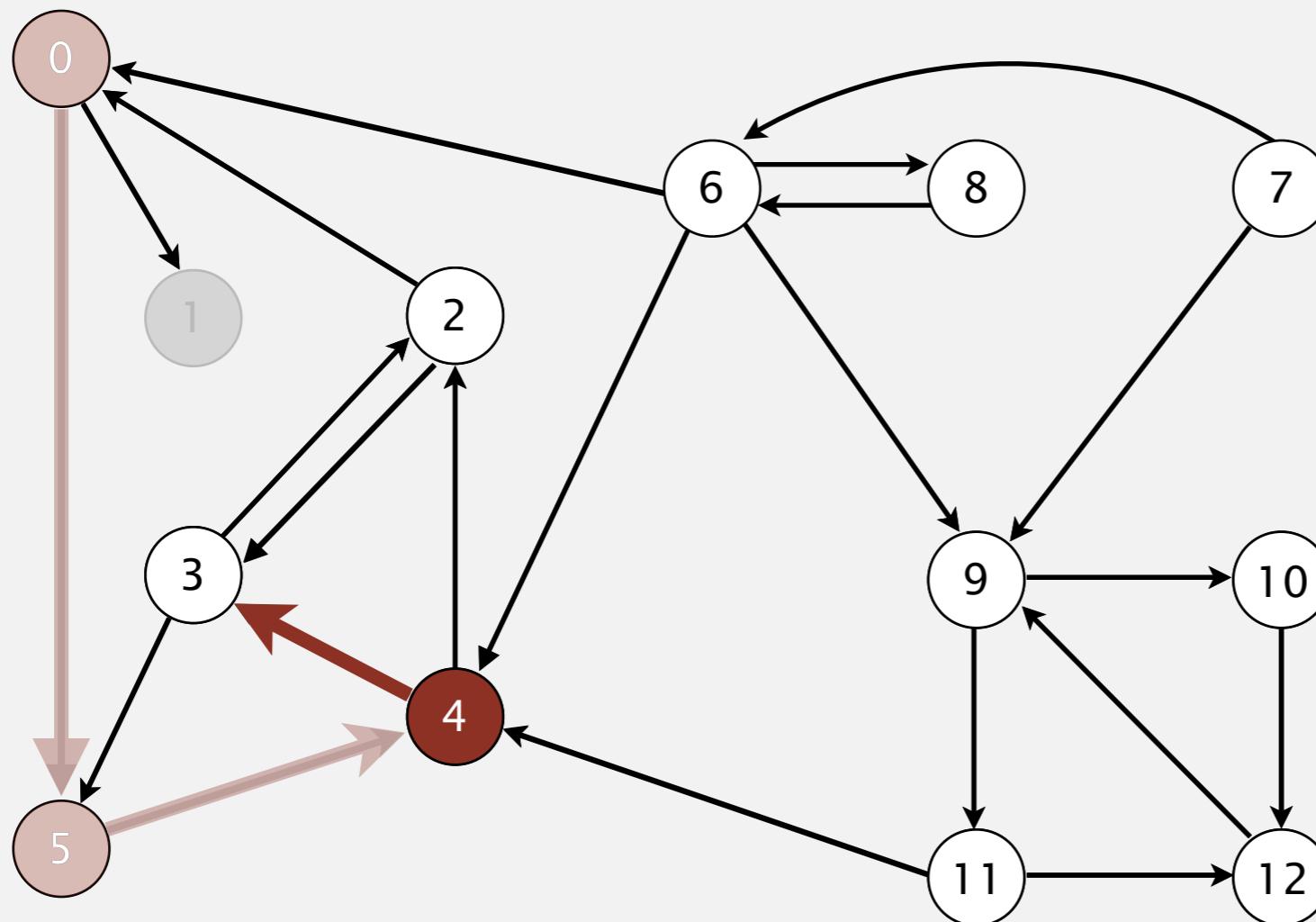


visit 5: check 4

v	scc[v]
0	1
1	0
2	-
3	-
4	-
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

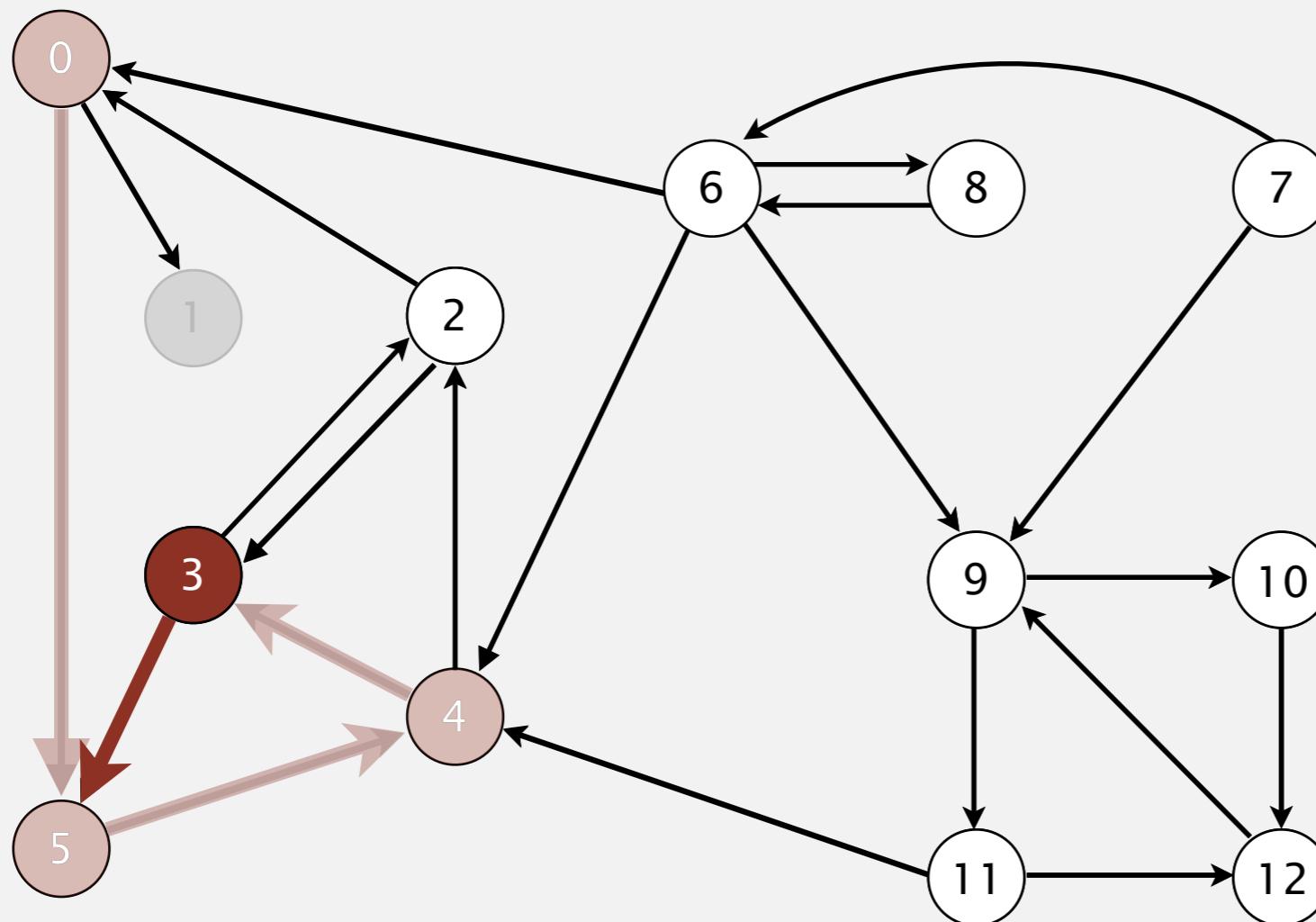


v	scc[v]
0	1
1	0
2	-
3	-
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 4: check 3 and check 2

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

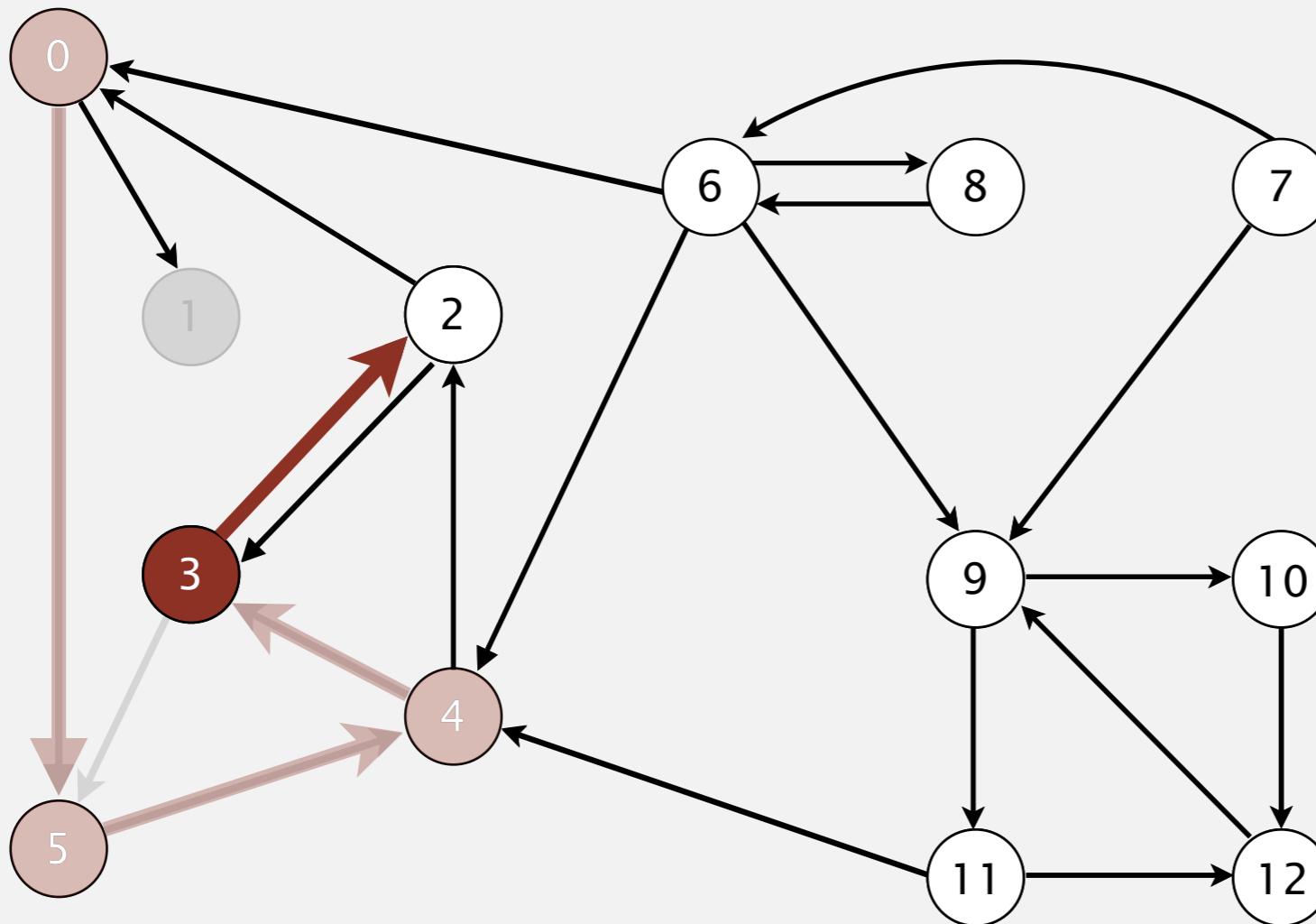


v	scc[v]
0	1
1	0
2	-
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 3: check 5 and check 2

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

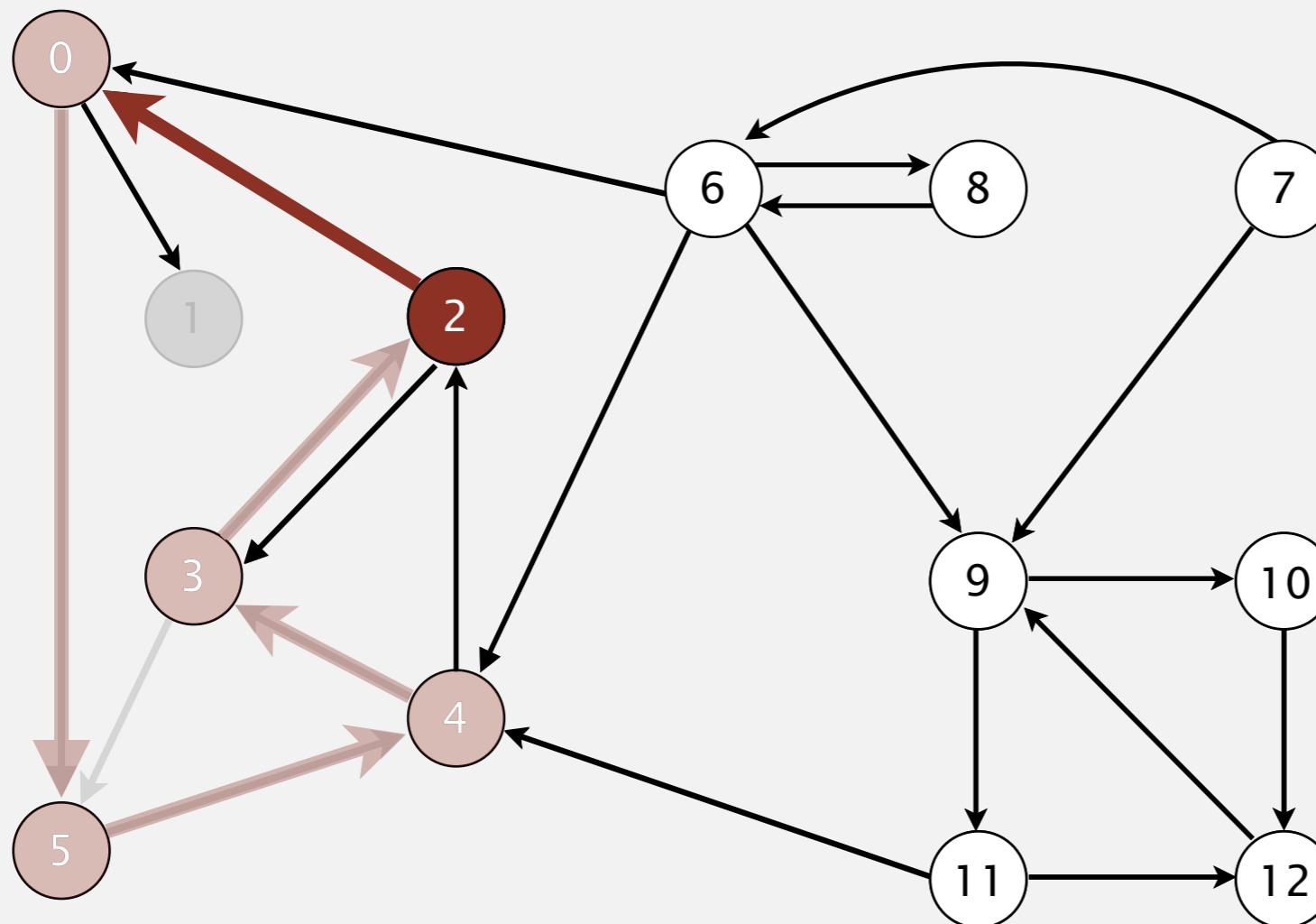


v	scc[v]
0	1
1	0
2	-
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 3: check 5 and check 2

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

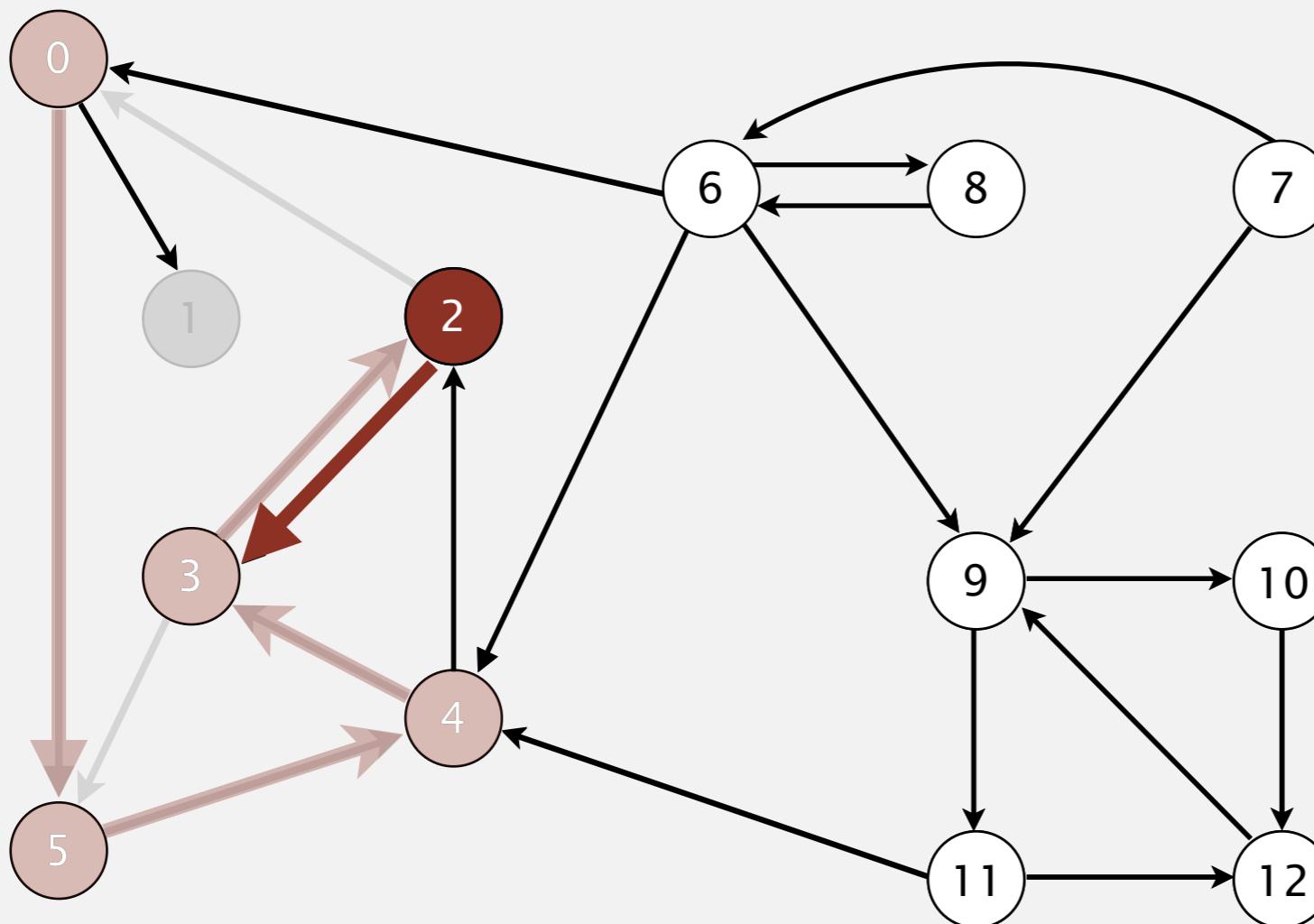


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 2: check 0 and check 3

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

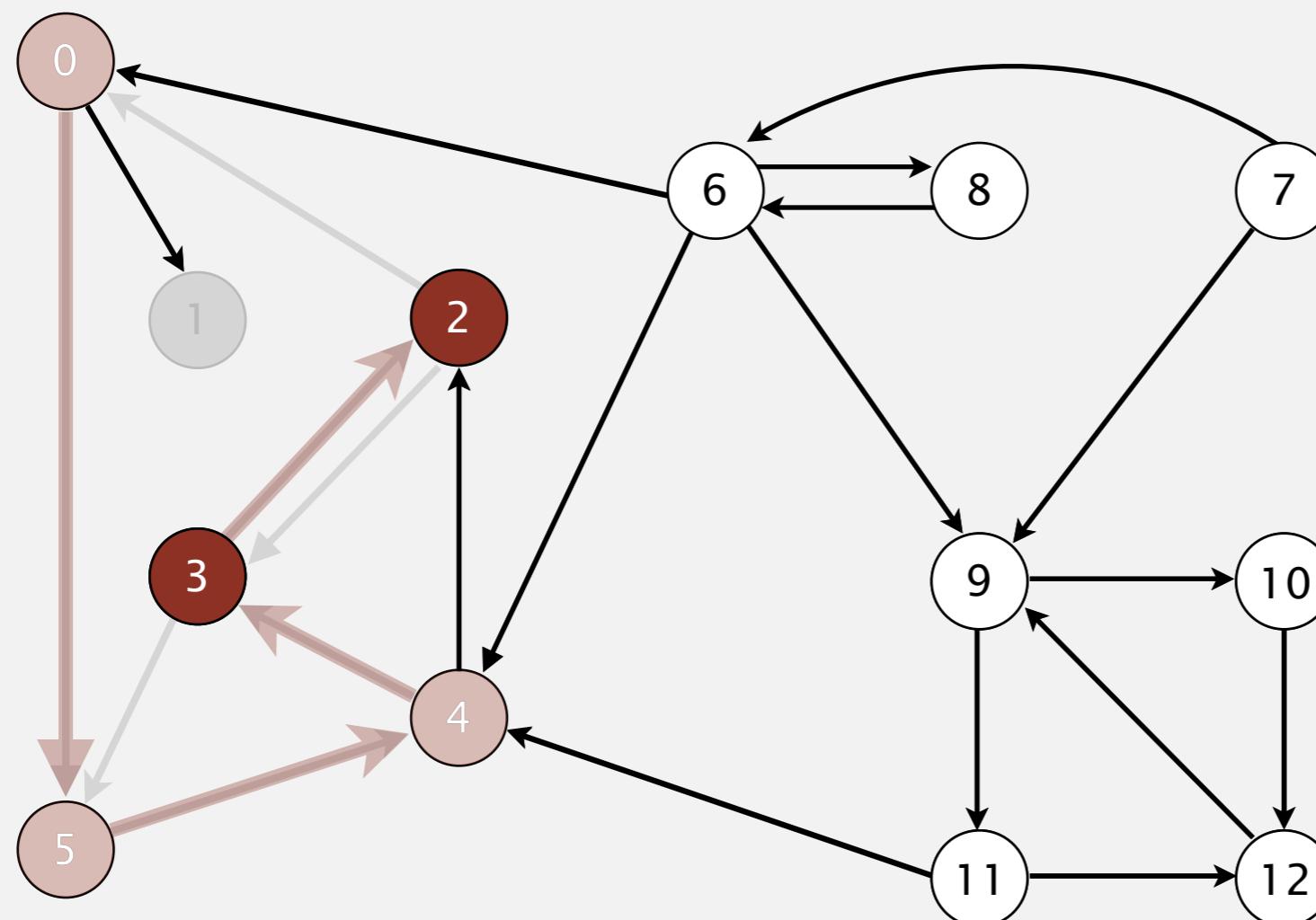


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 2: check 0 and check 3

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

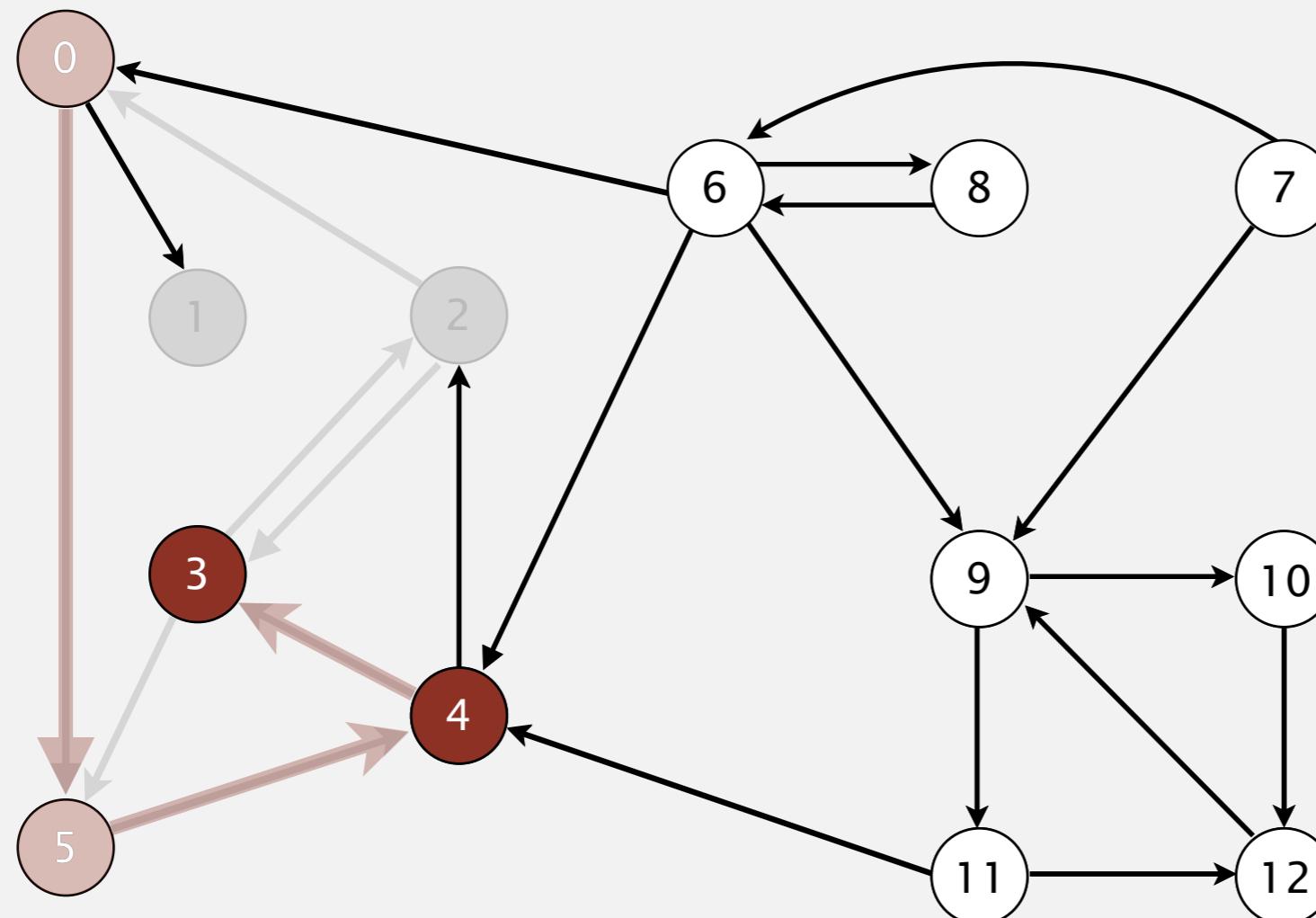


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

2 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

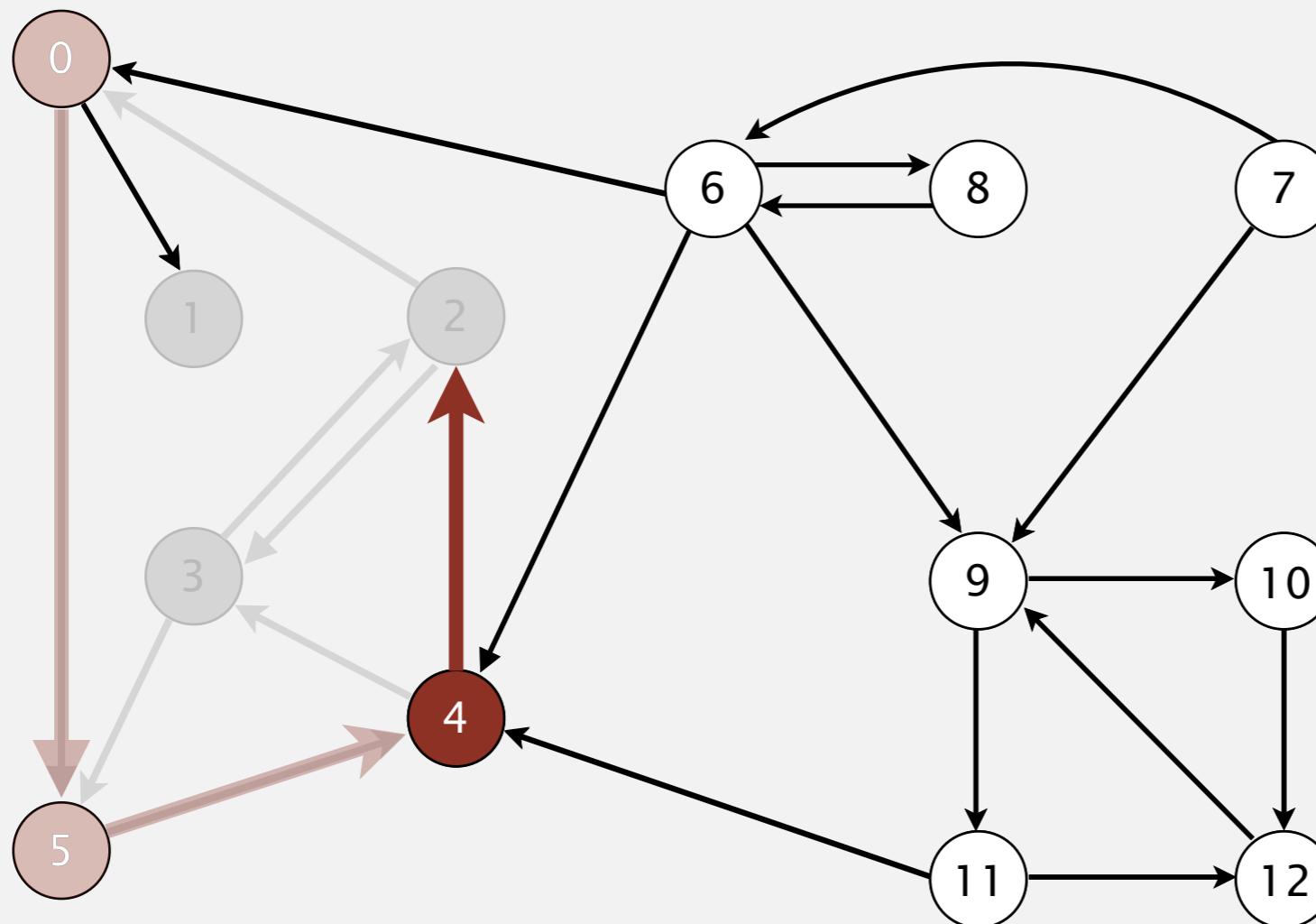


3 done

v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

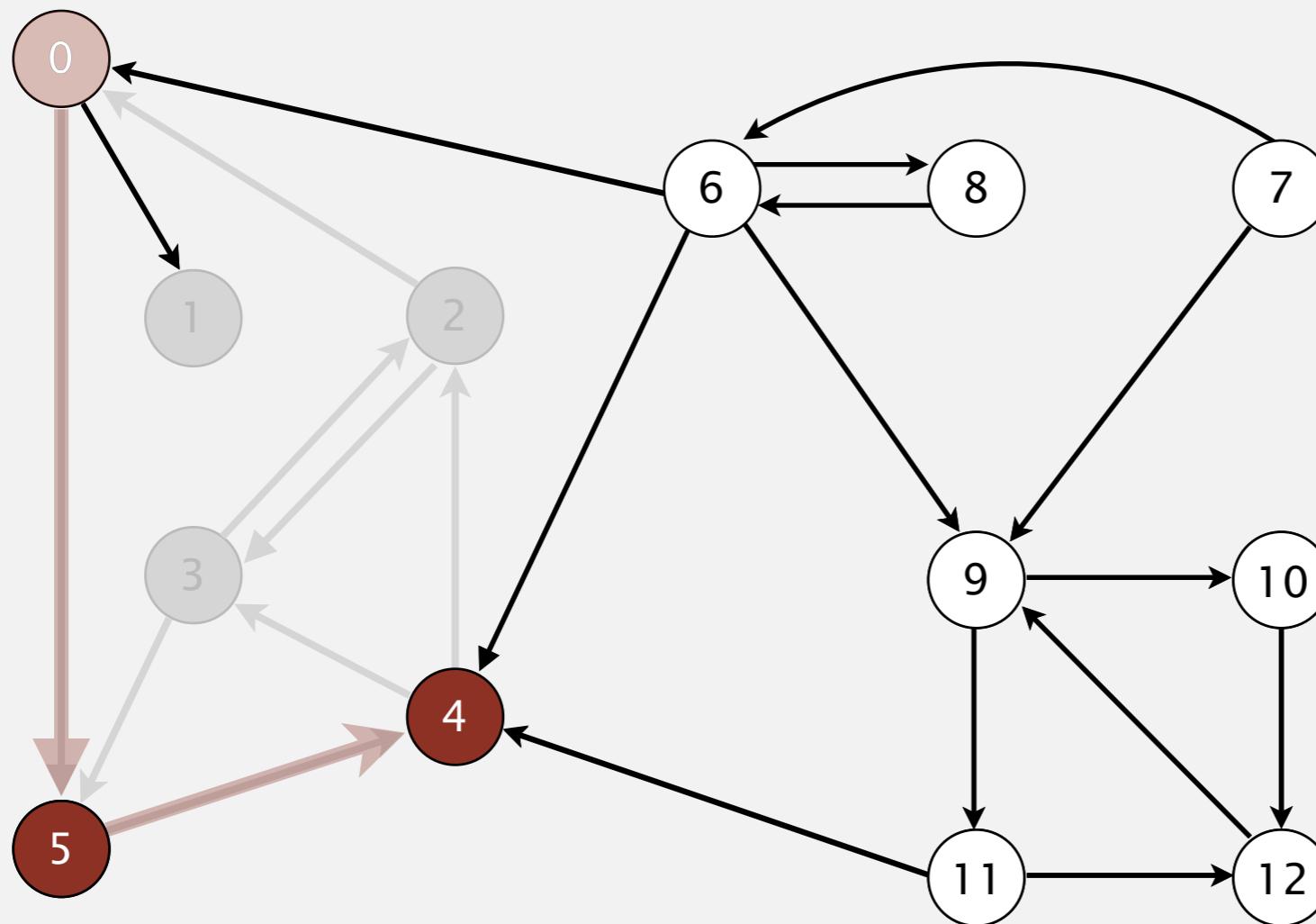


visit 4: check 3 and check 2

v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

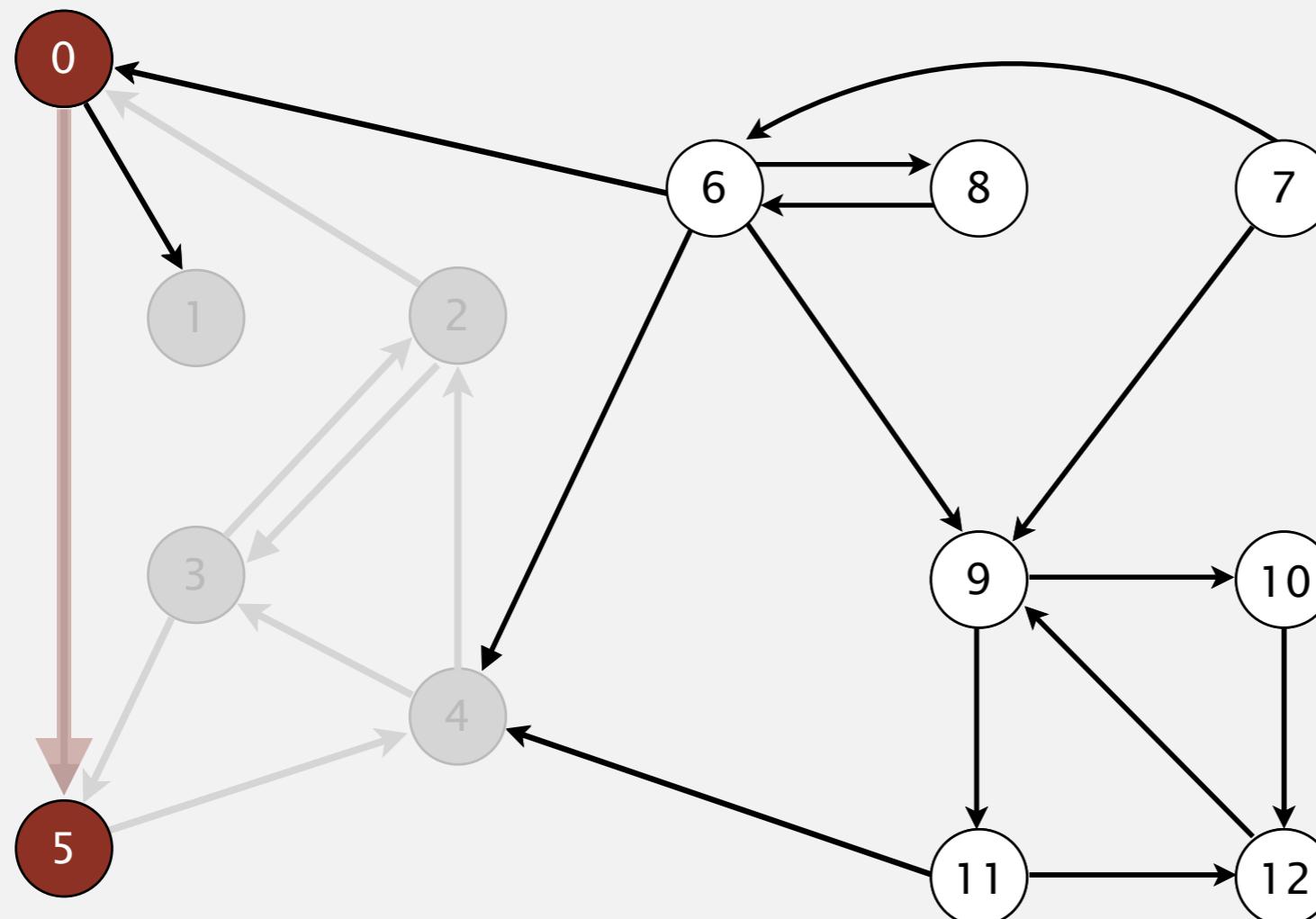


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

4 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

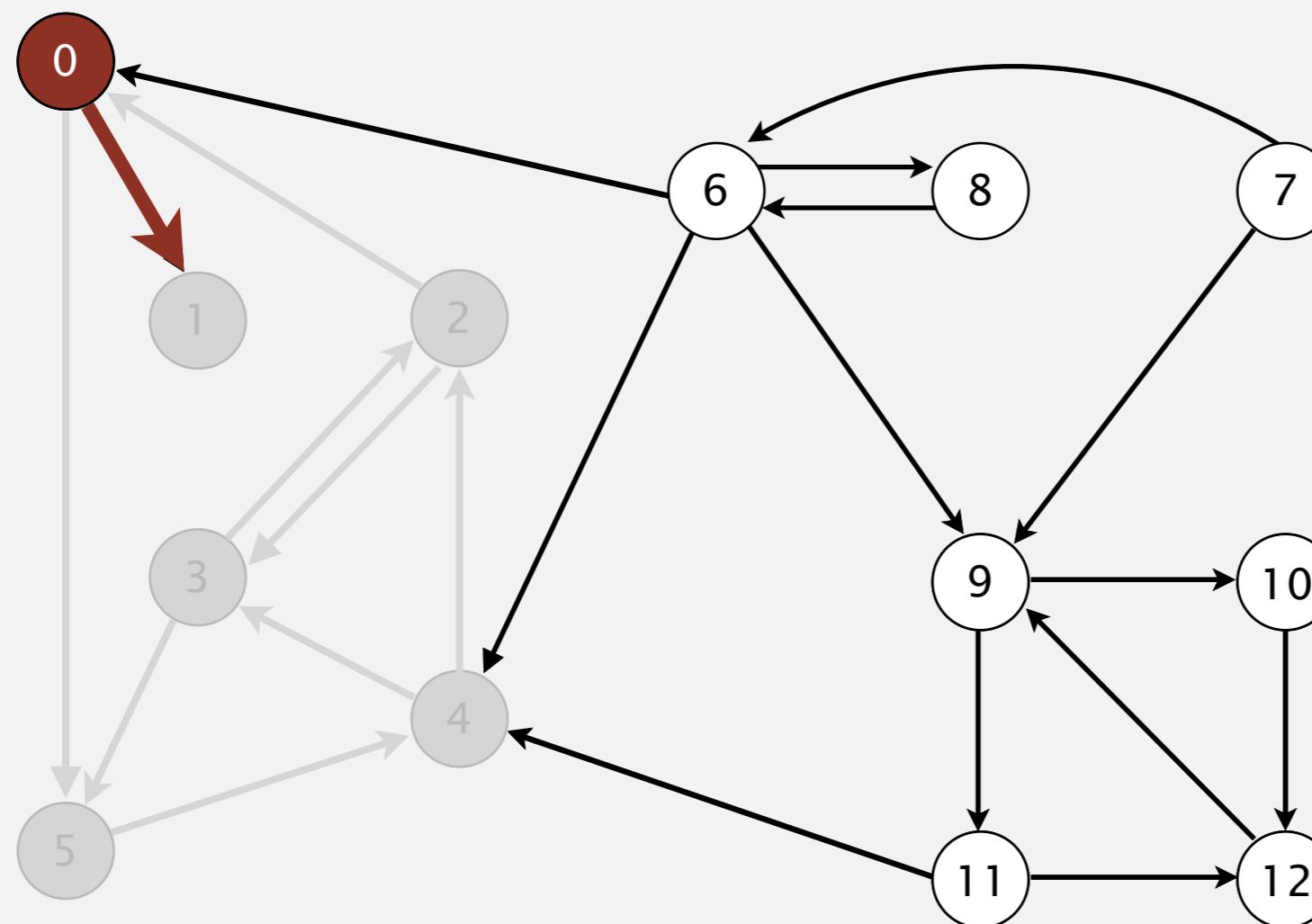


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

5 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

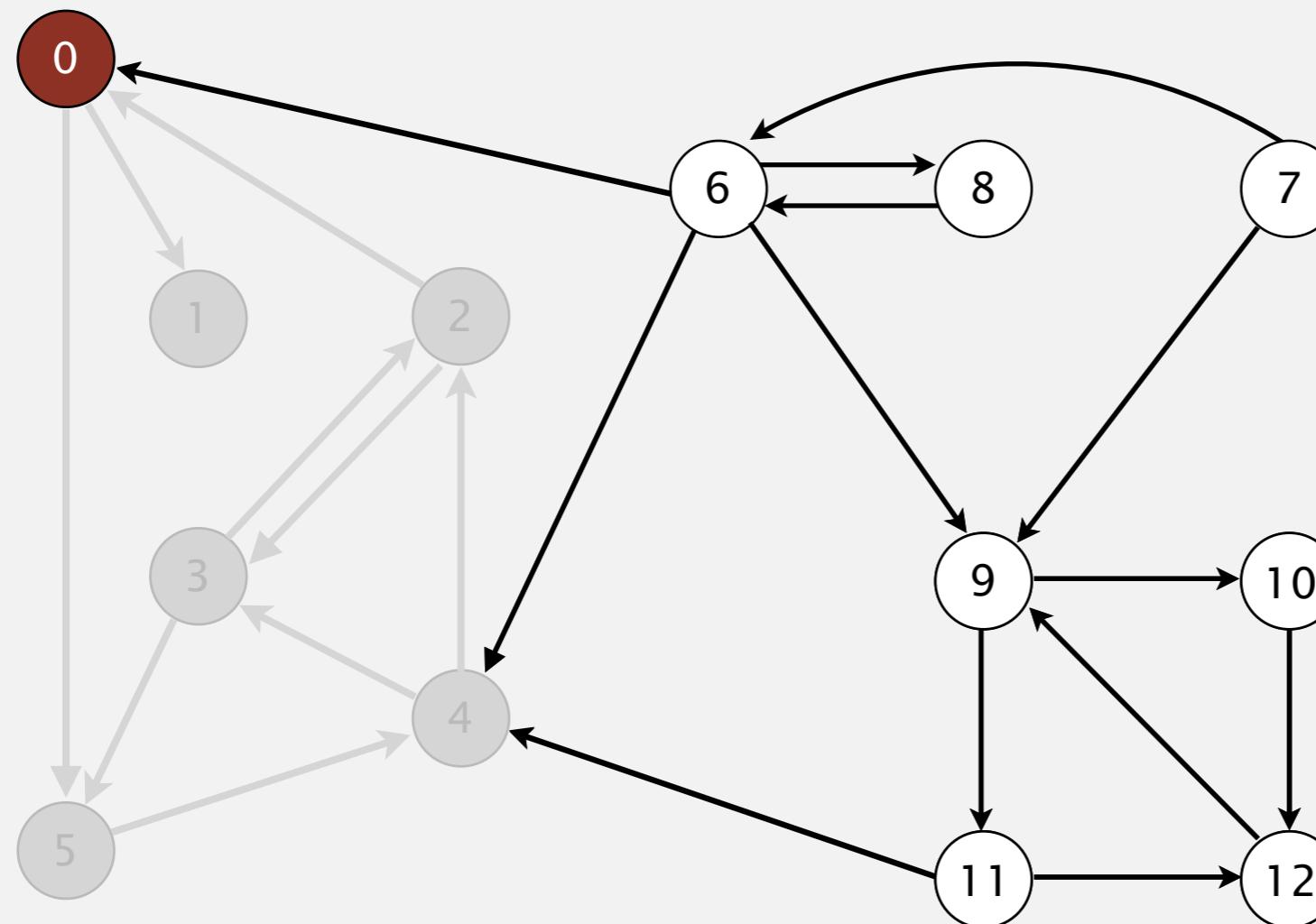


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

visit 0: check 5 and check 1

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R . 1 0 2 4 5 3 11 9 12 10 6 7 8

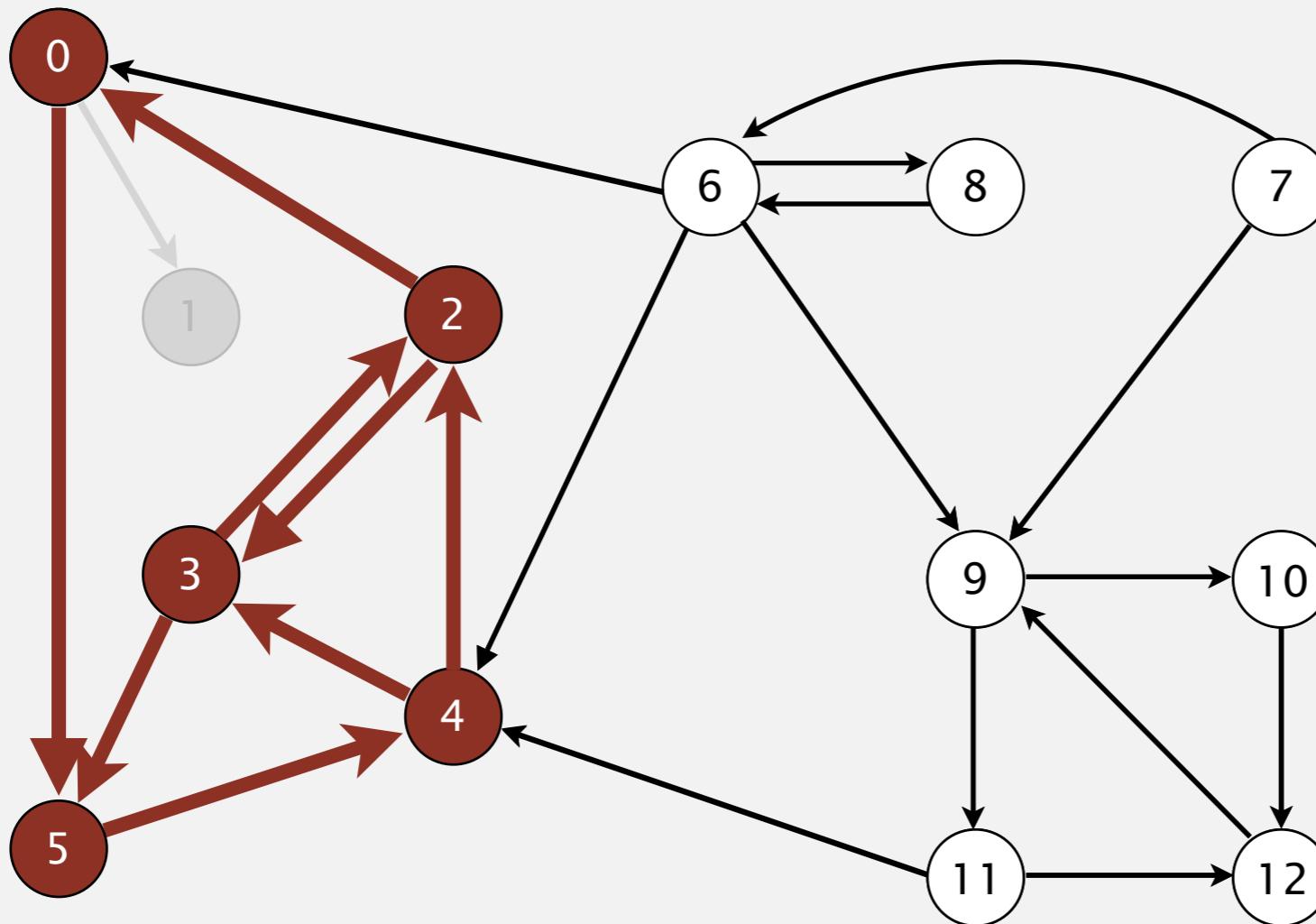


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

0 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .
1 0 2 4 5 3 11 9 12 10 6 7 8



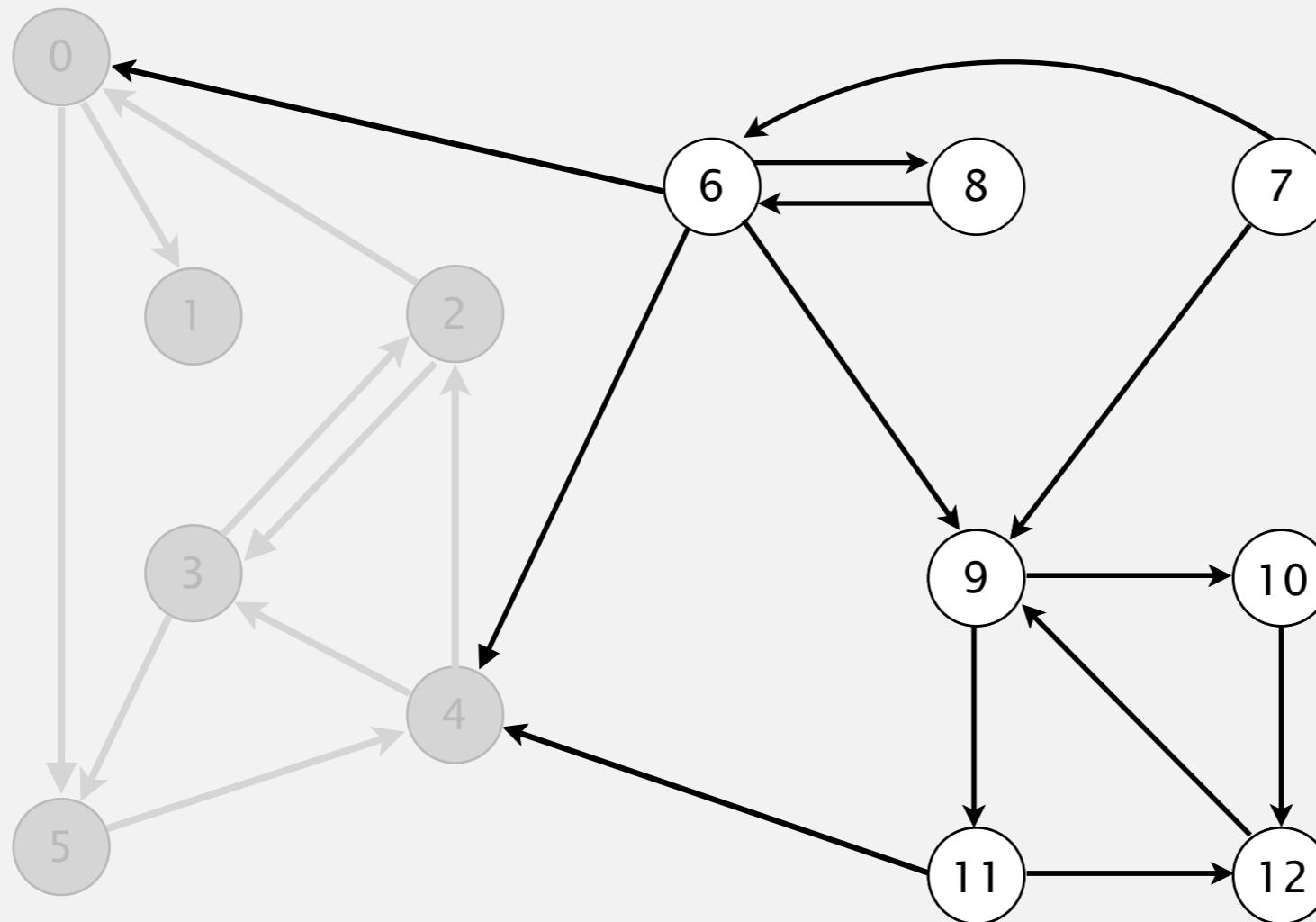
strong component: 0 2 3 4 5

v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 **2** 4 5 3 11 9 12 10 6 7 8



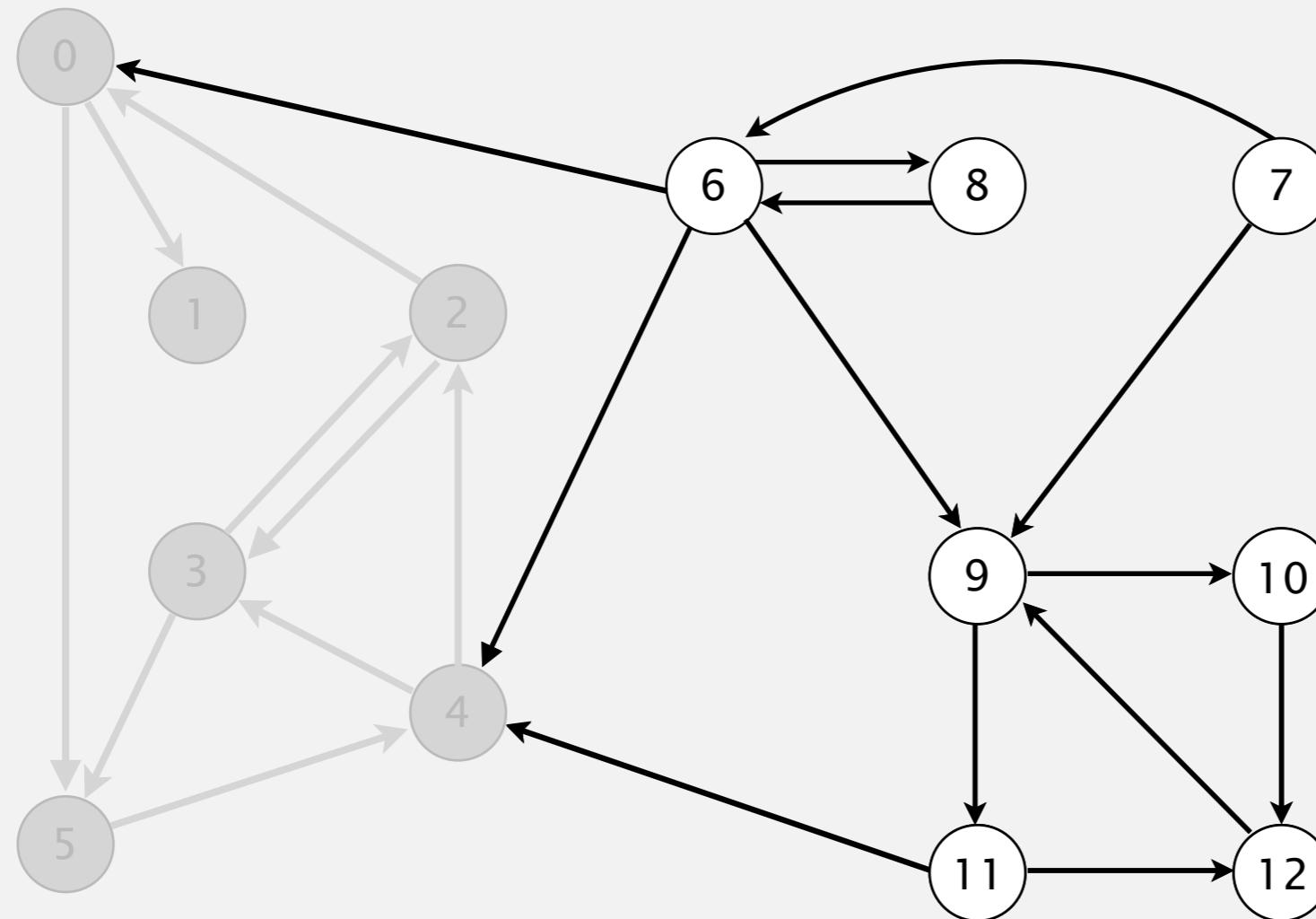
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 2

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



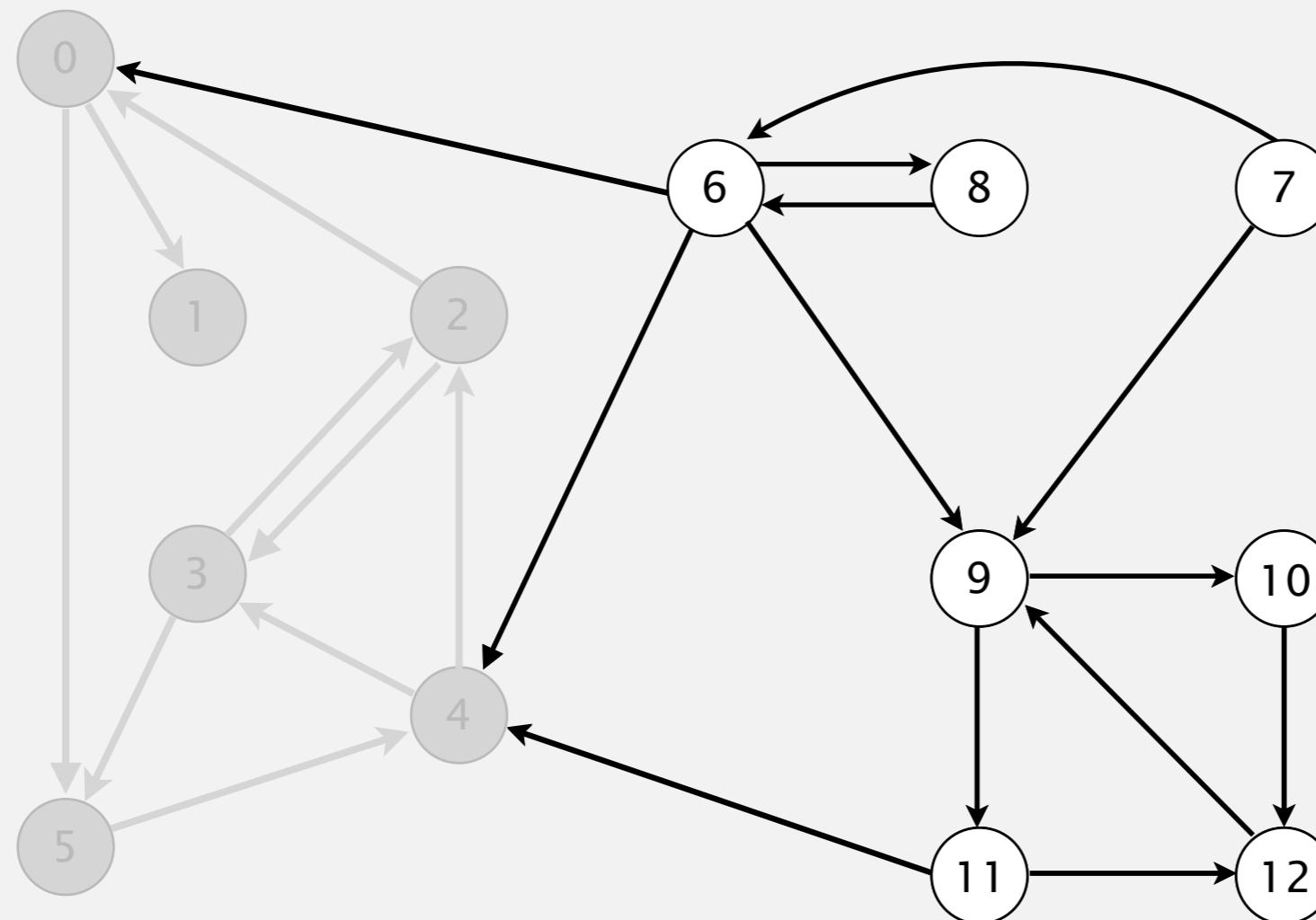
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 4

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



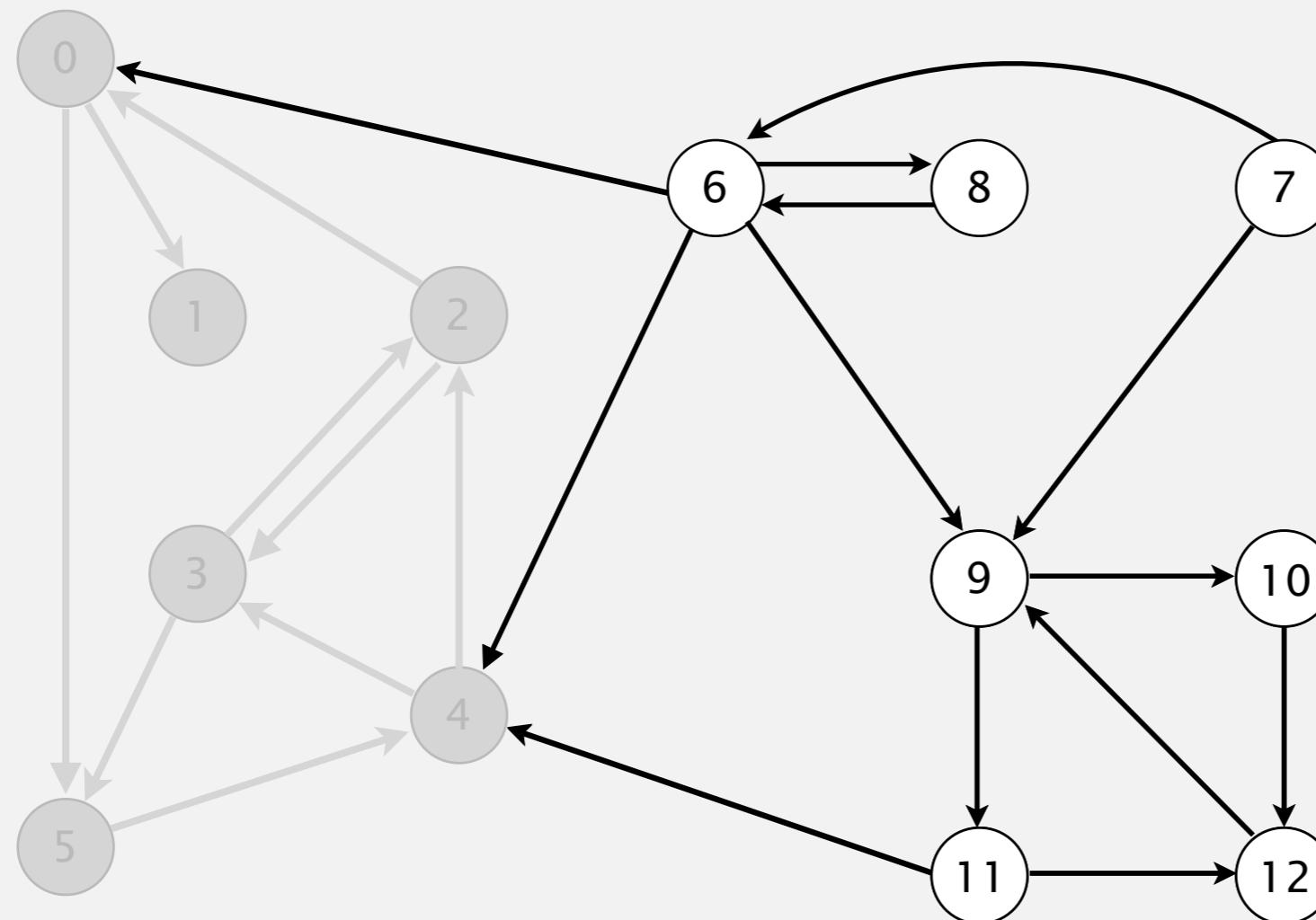
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 5

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 (3) 11 9 12 10 6 7 8



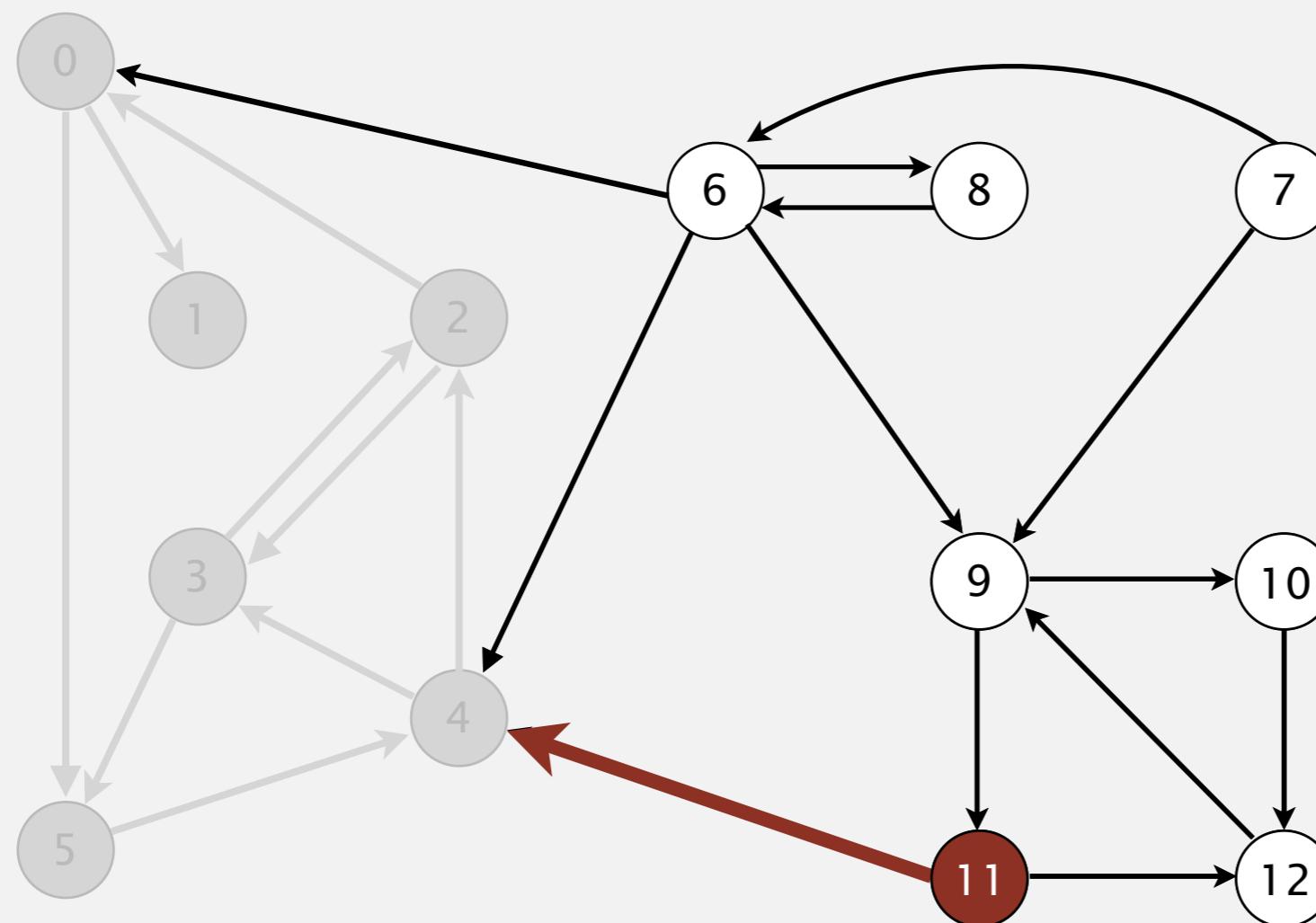
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	-
12	-

check 3

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



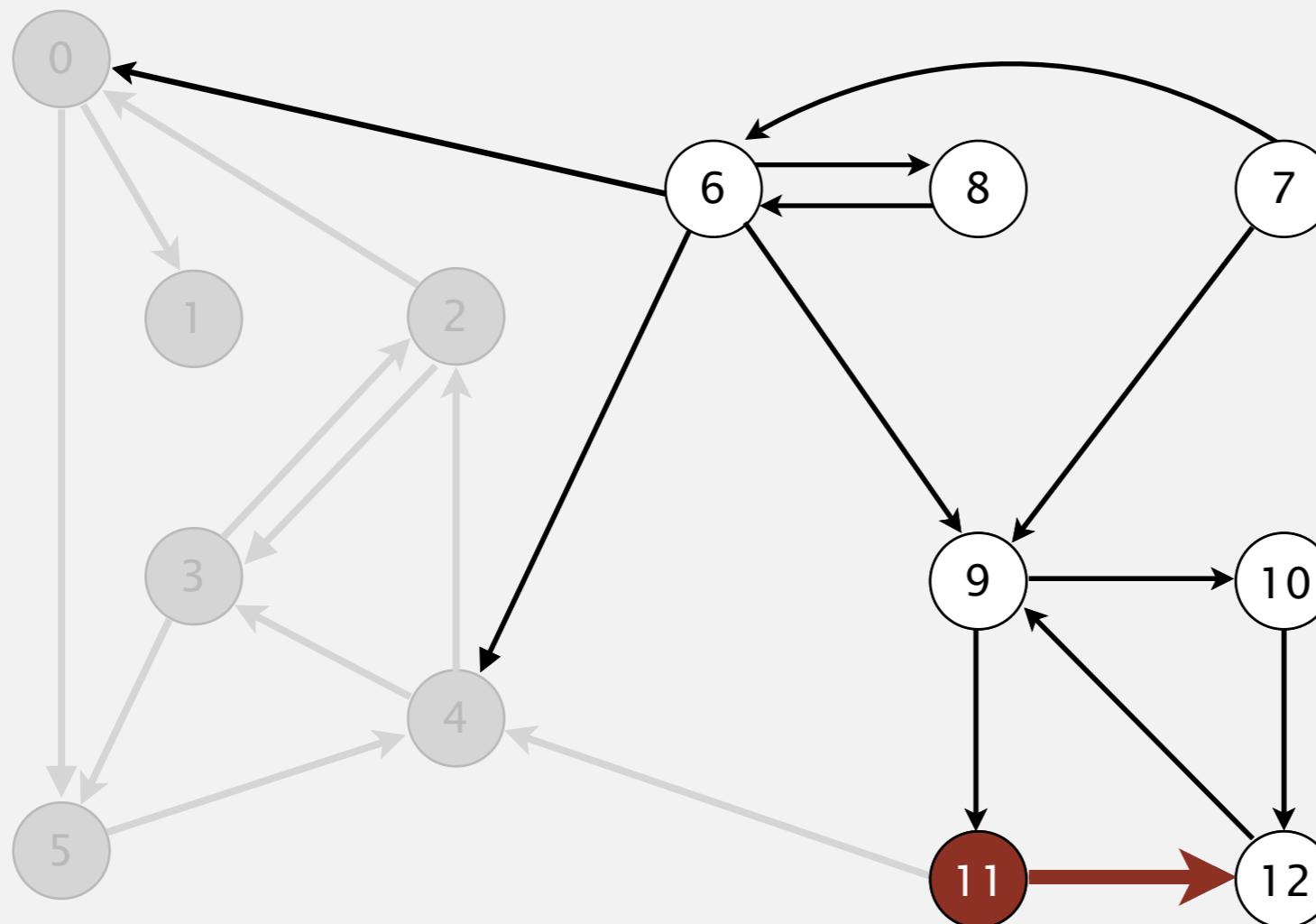
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	2
12	-

visit 11: check 4 and check 12

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



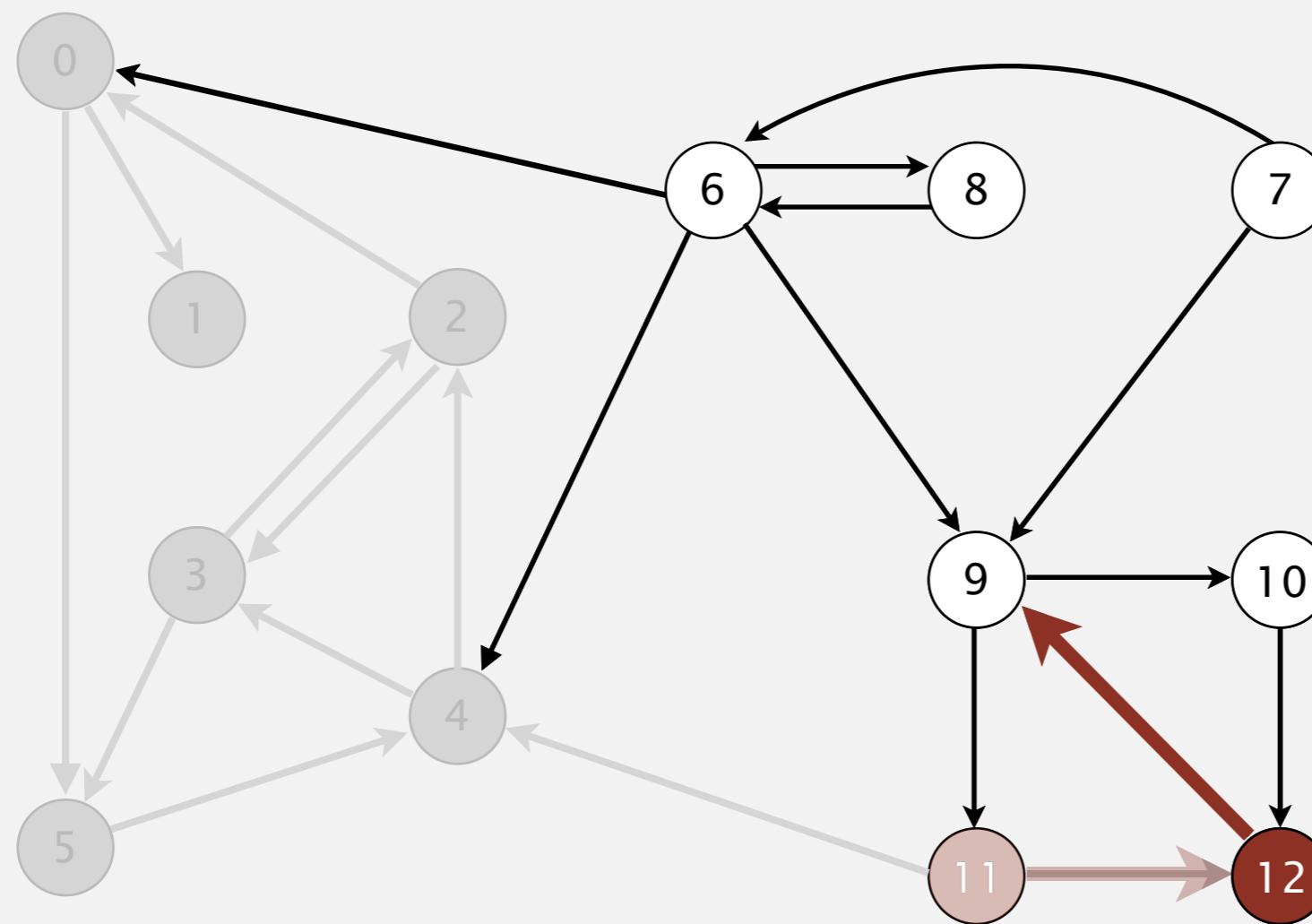
visit 11: check 4 and check 12

v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	2
12	-

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



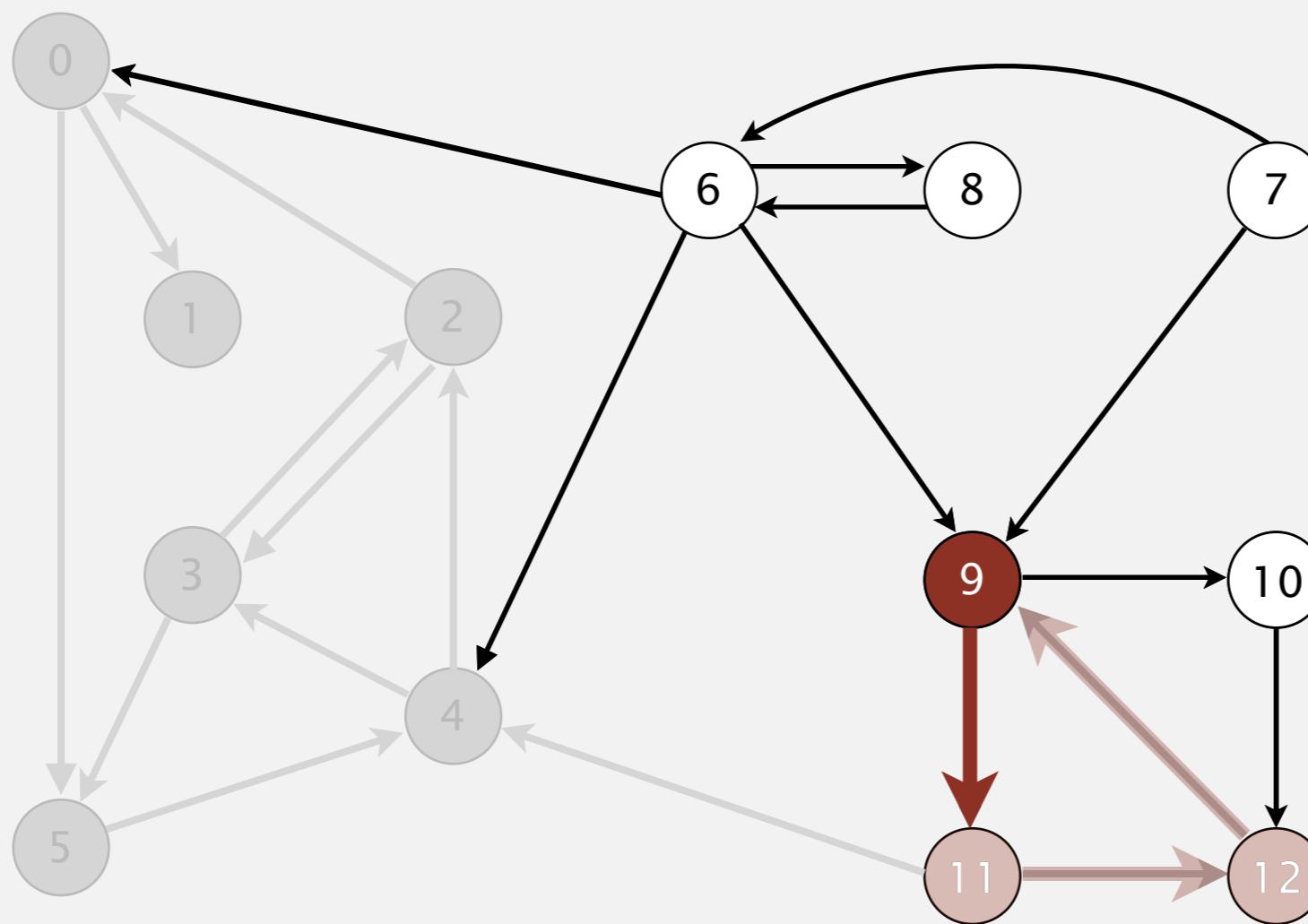
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	-
10	-
11	2
12	2

visit 12: check 9

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



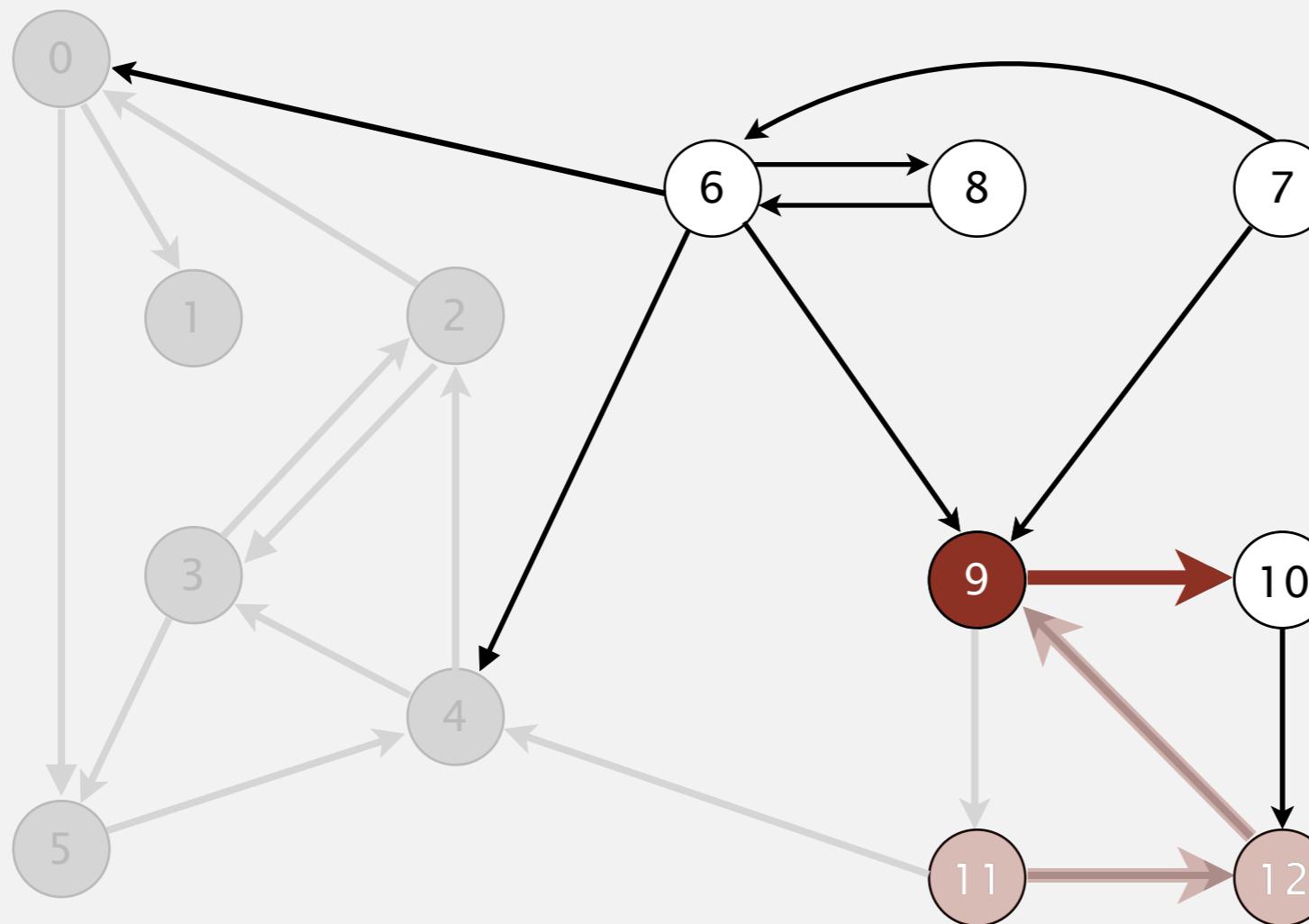
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	-
11	2
12	2

visit 9: check 11 and check 10

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



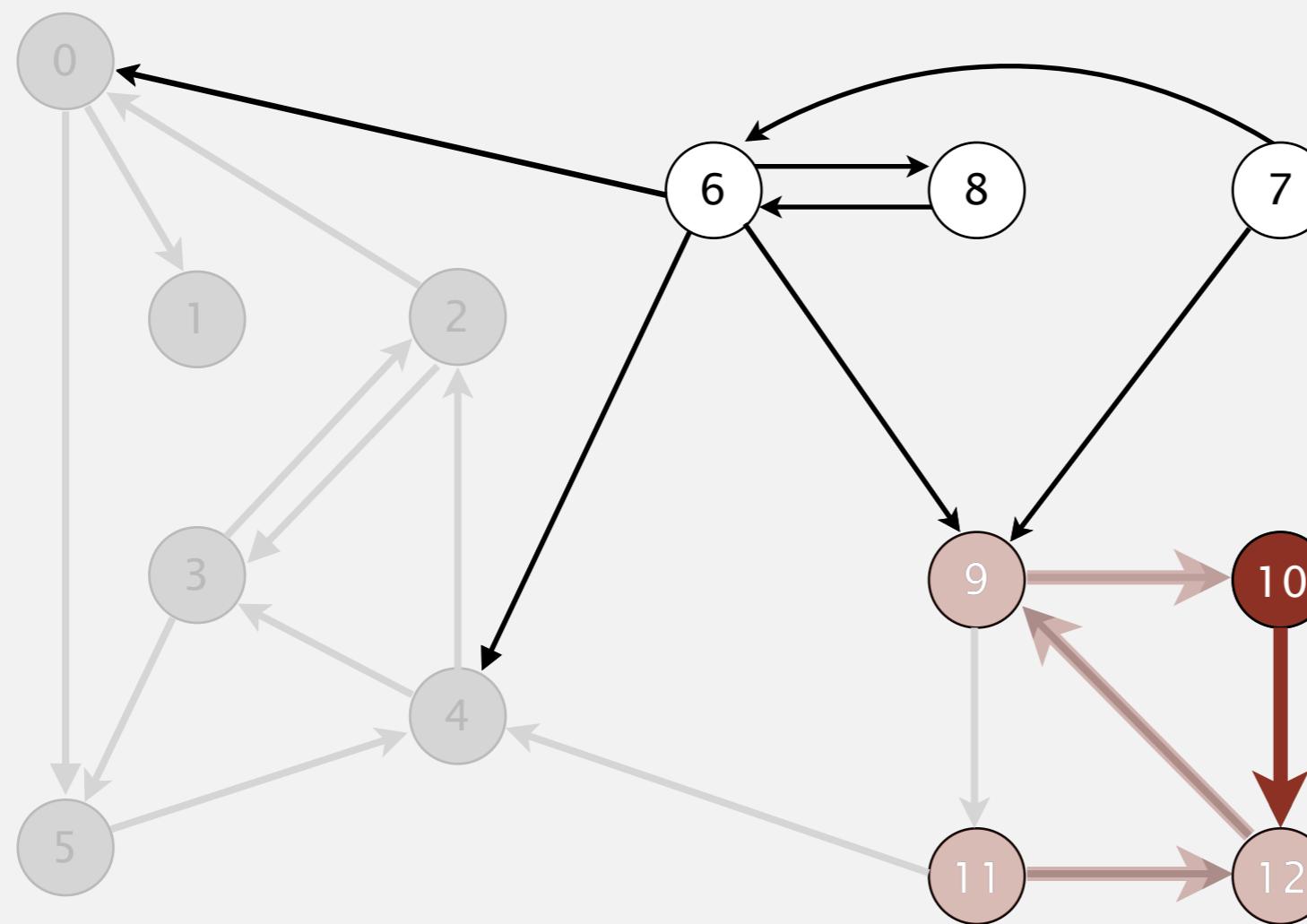
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	-
11	2
12	2

visit 9: check 11 and check 10

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



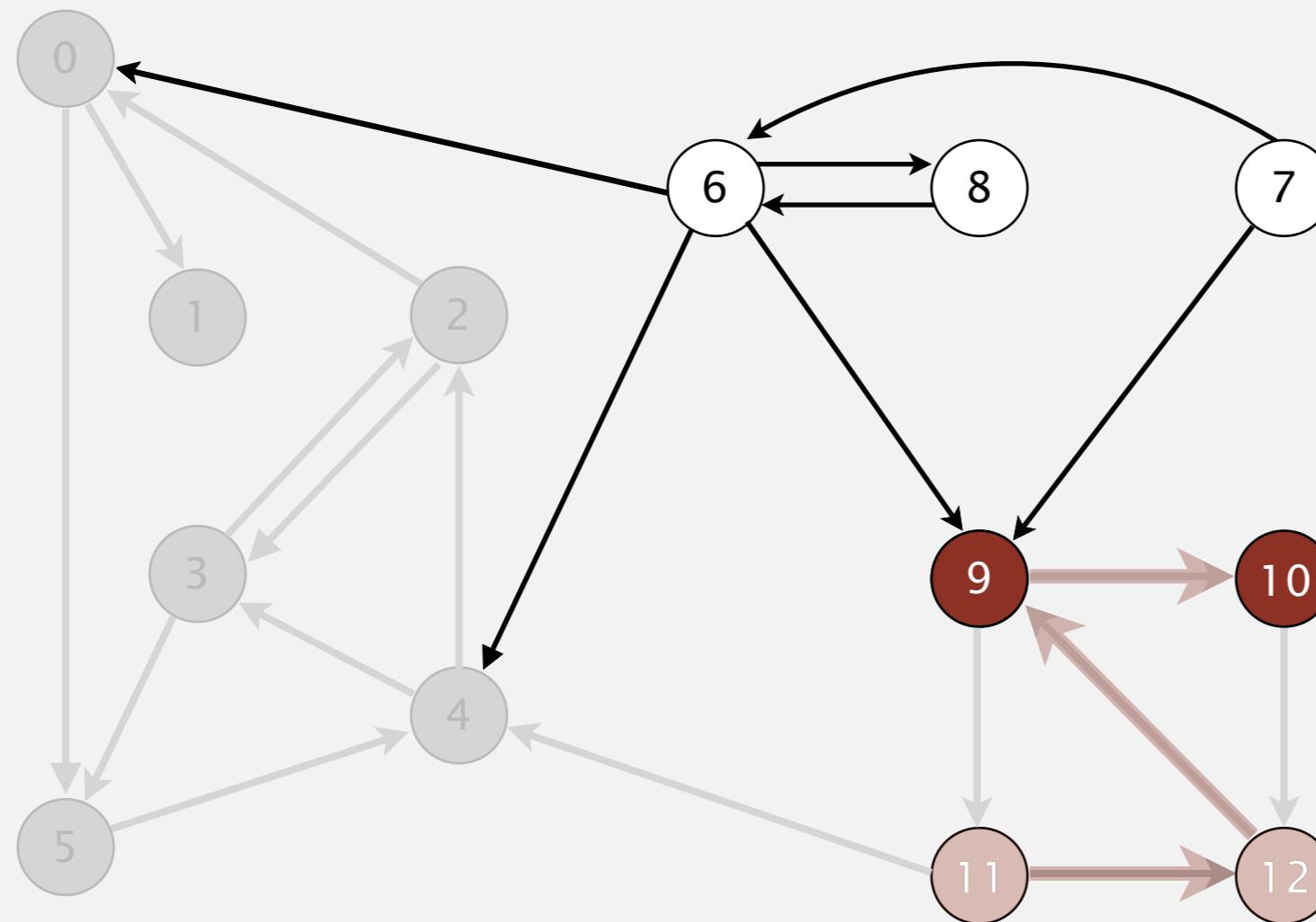
visit 10: check 12

v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



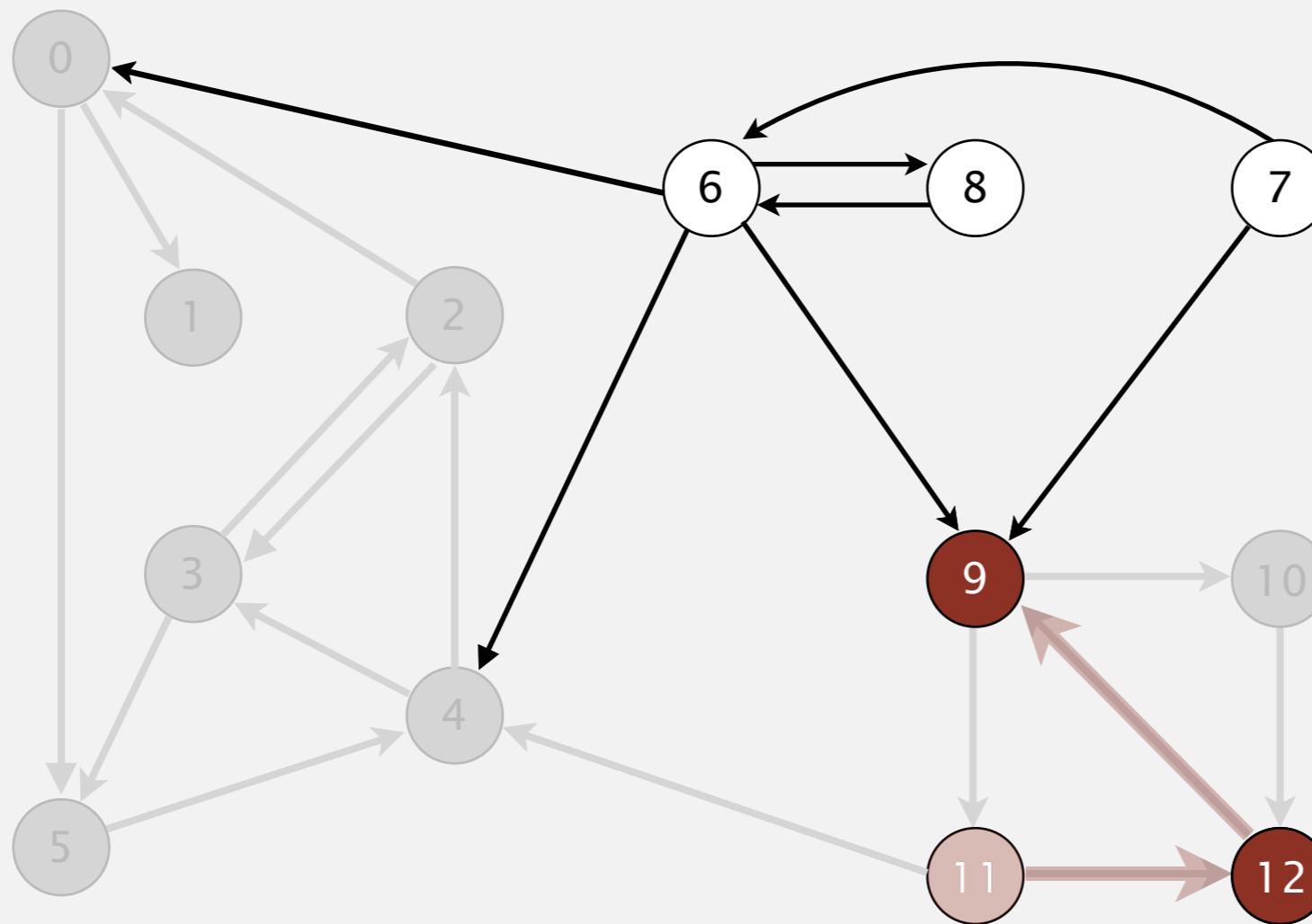
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

10 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



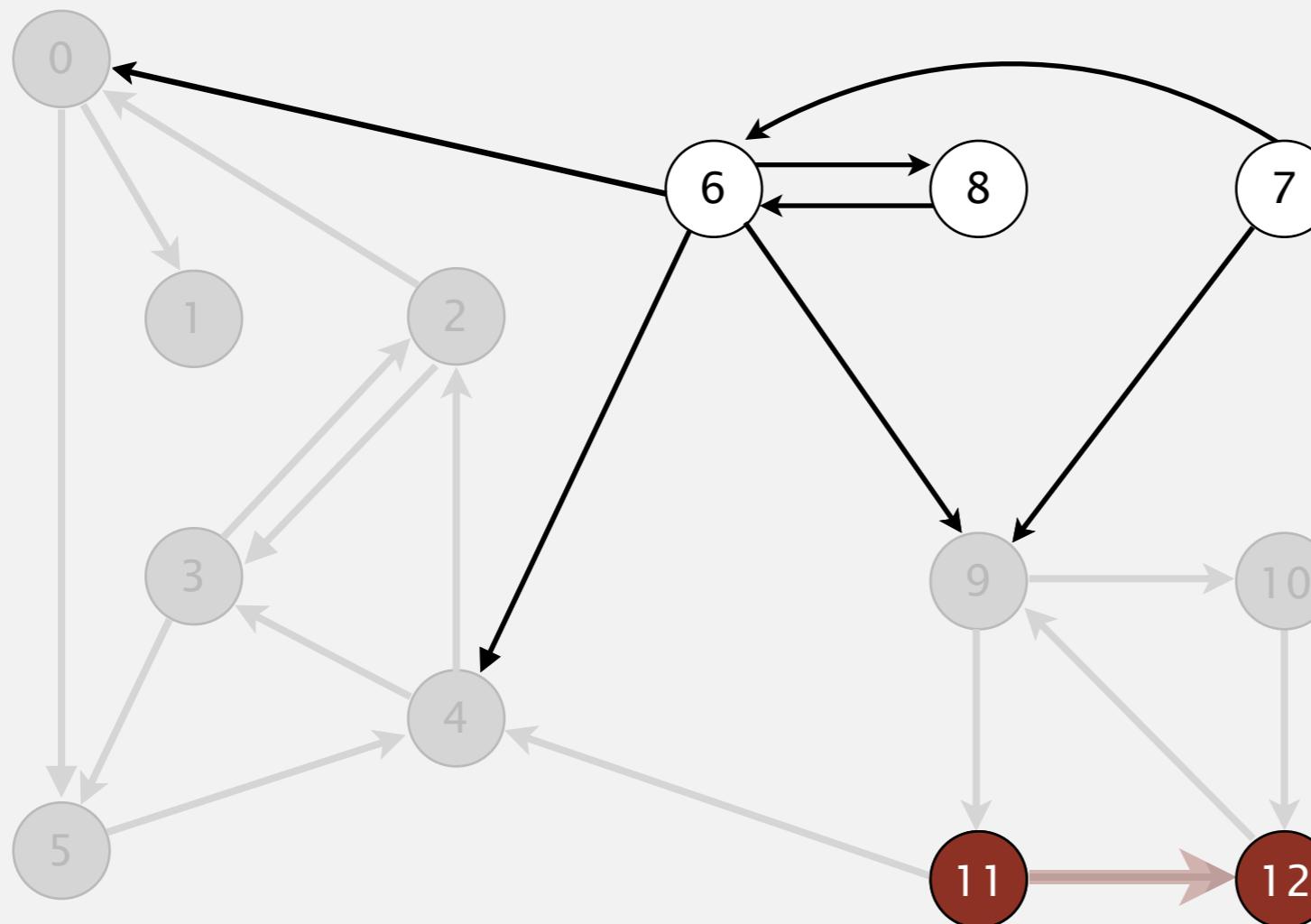
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

9 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



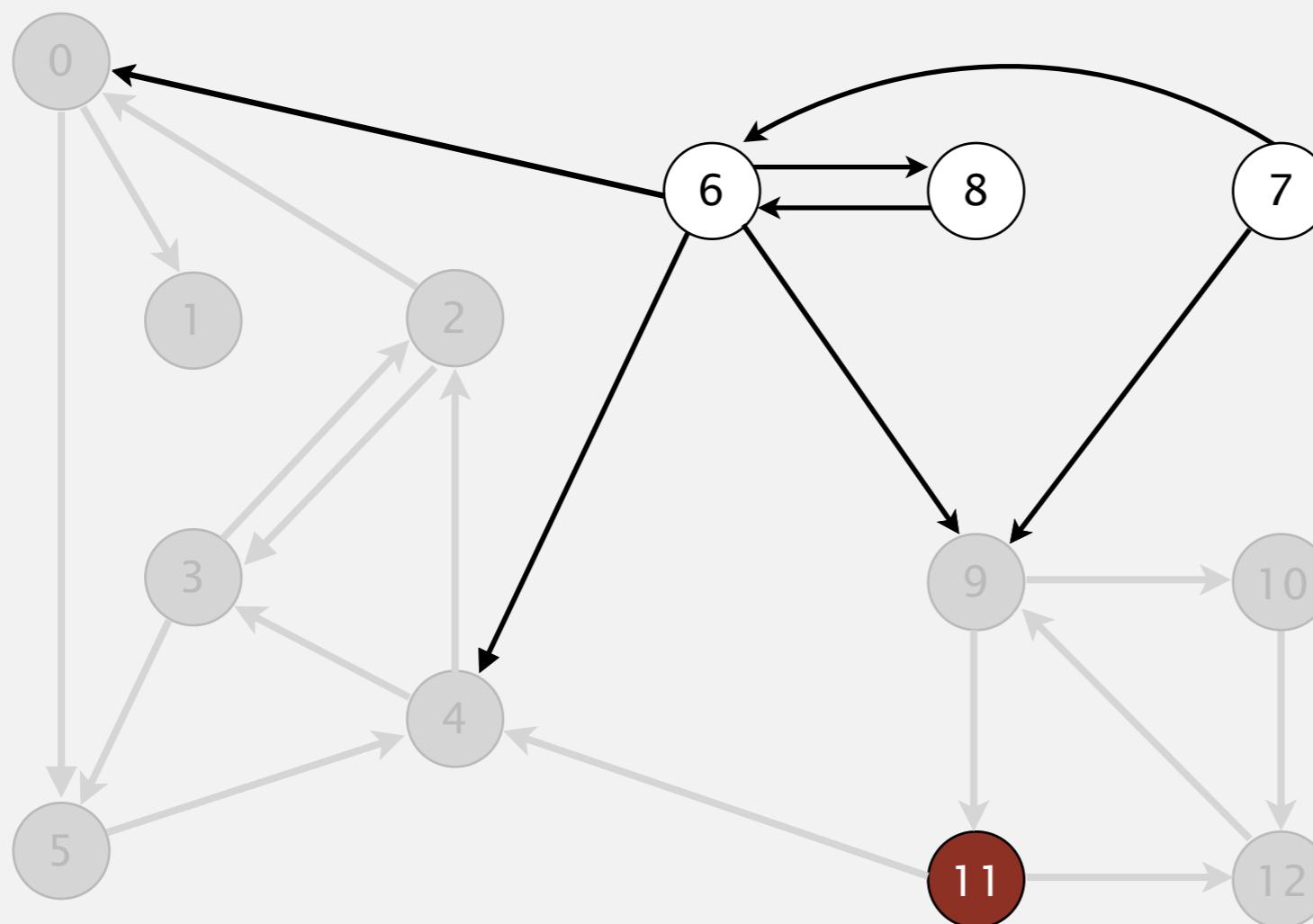
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

12 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



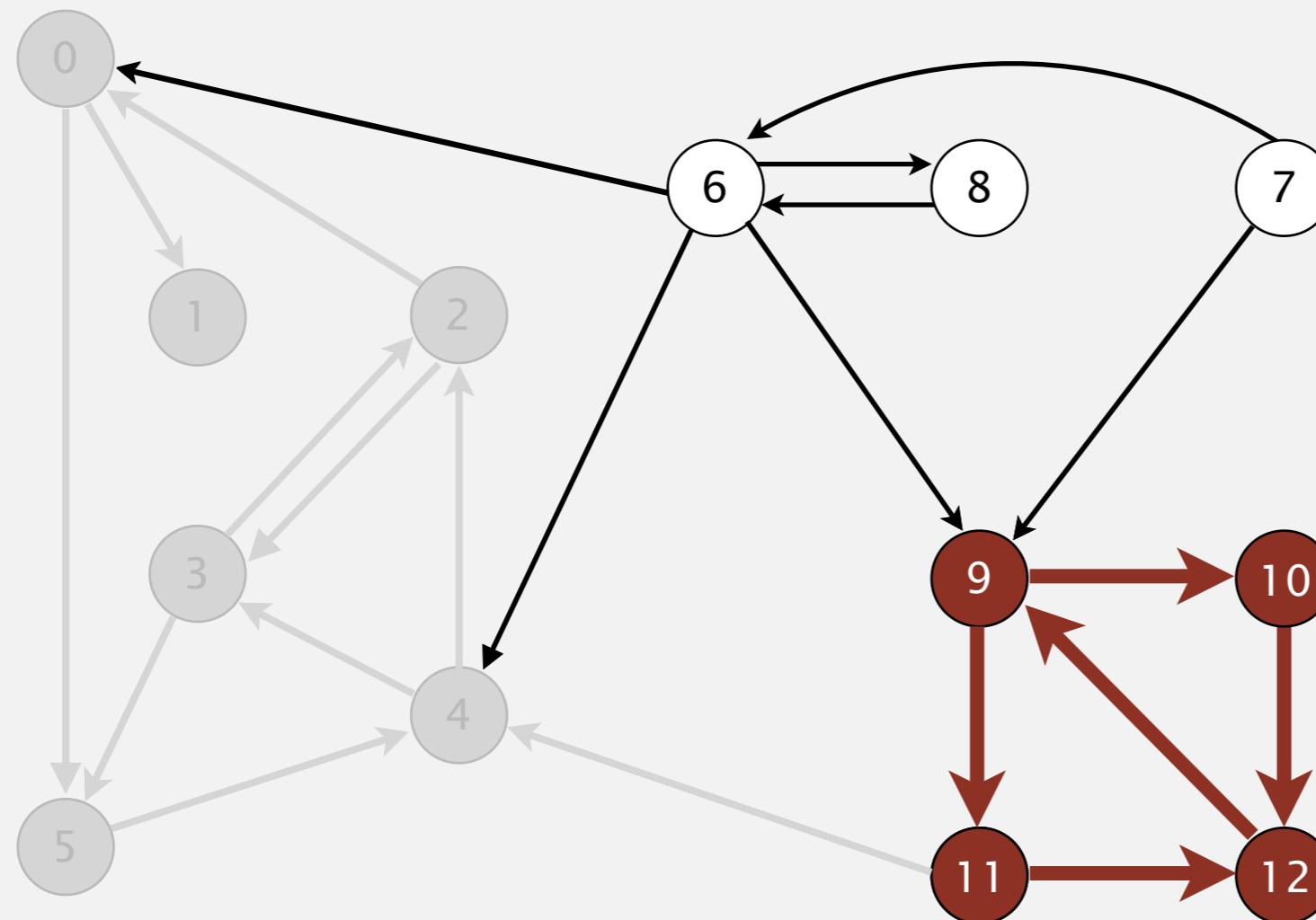
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

11 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 (11) 9 12 10 6 7 8



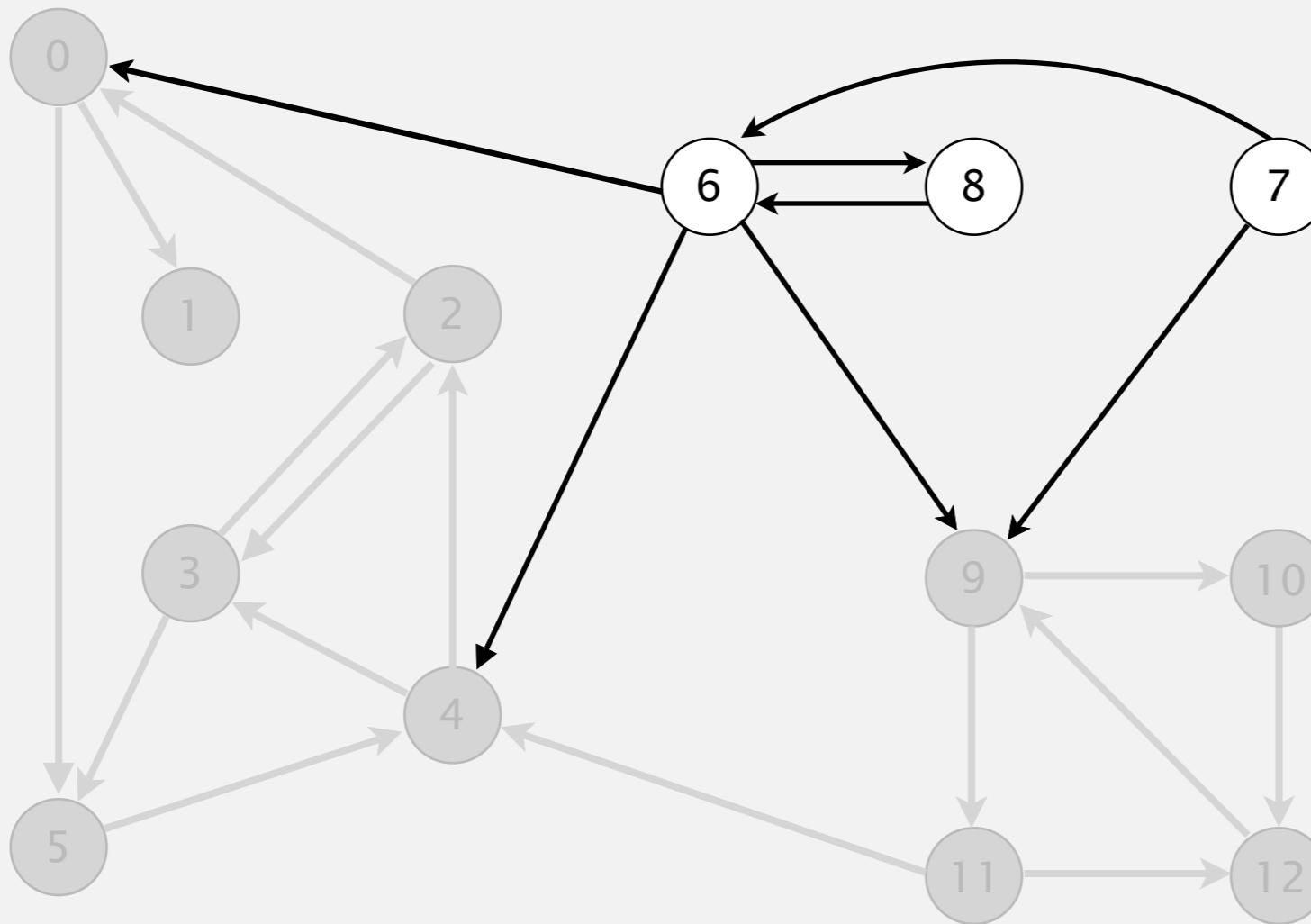
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

strong component: 9 10 11 12

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



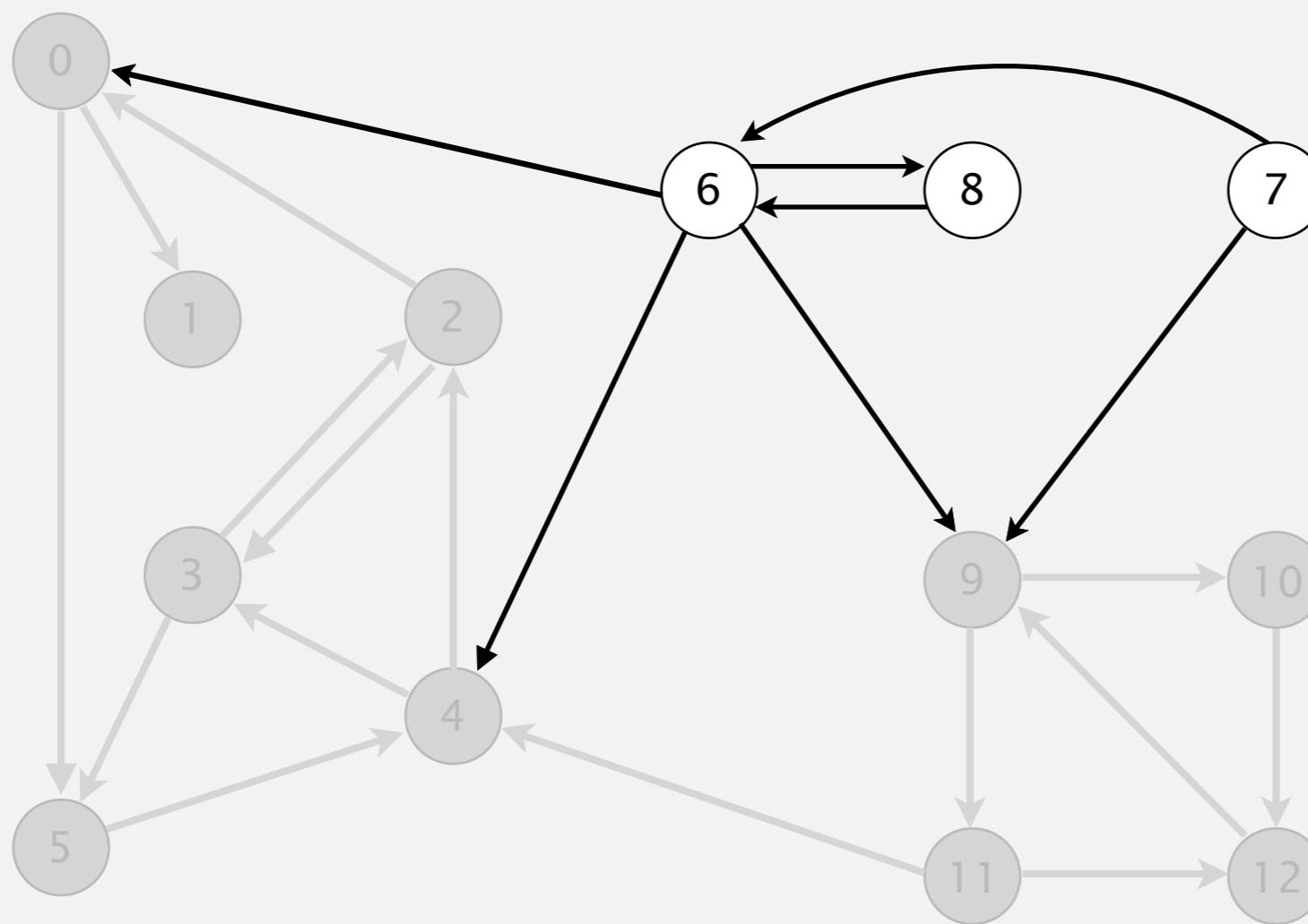
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

check 9

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



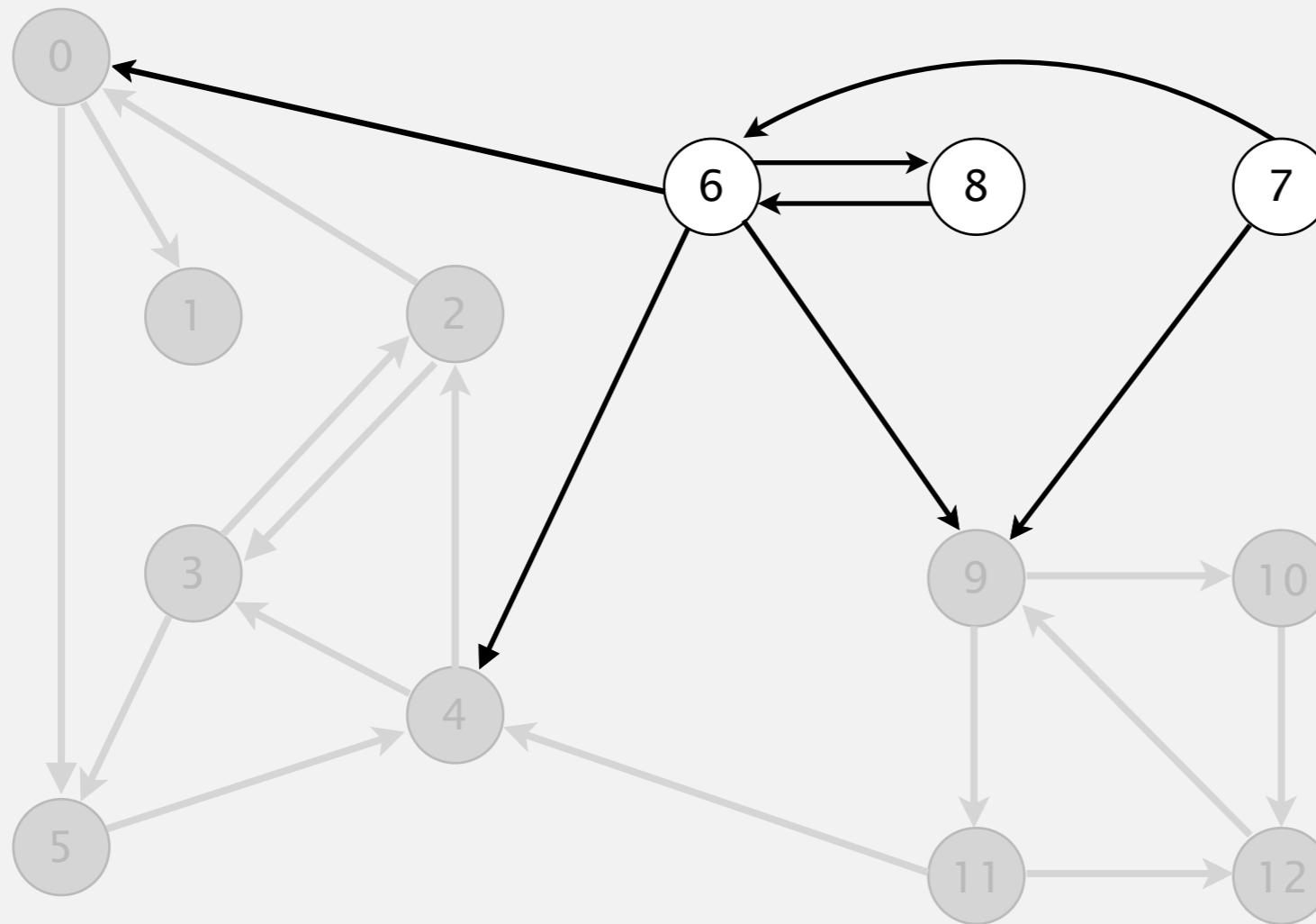
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

check 12

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 **10** 6 7 8



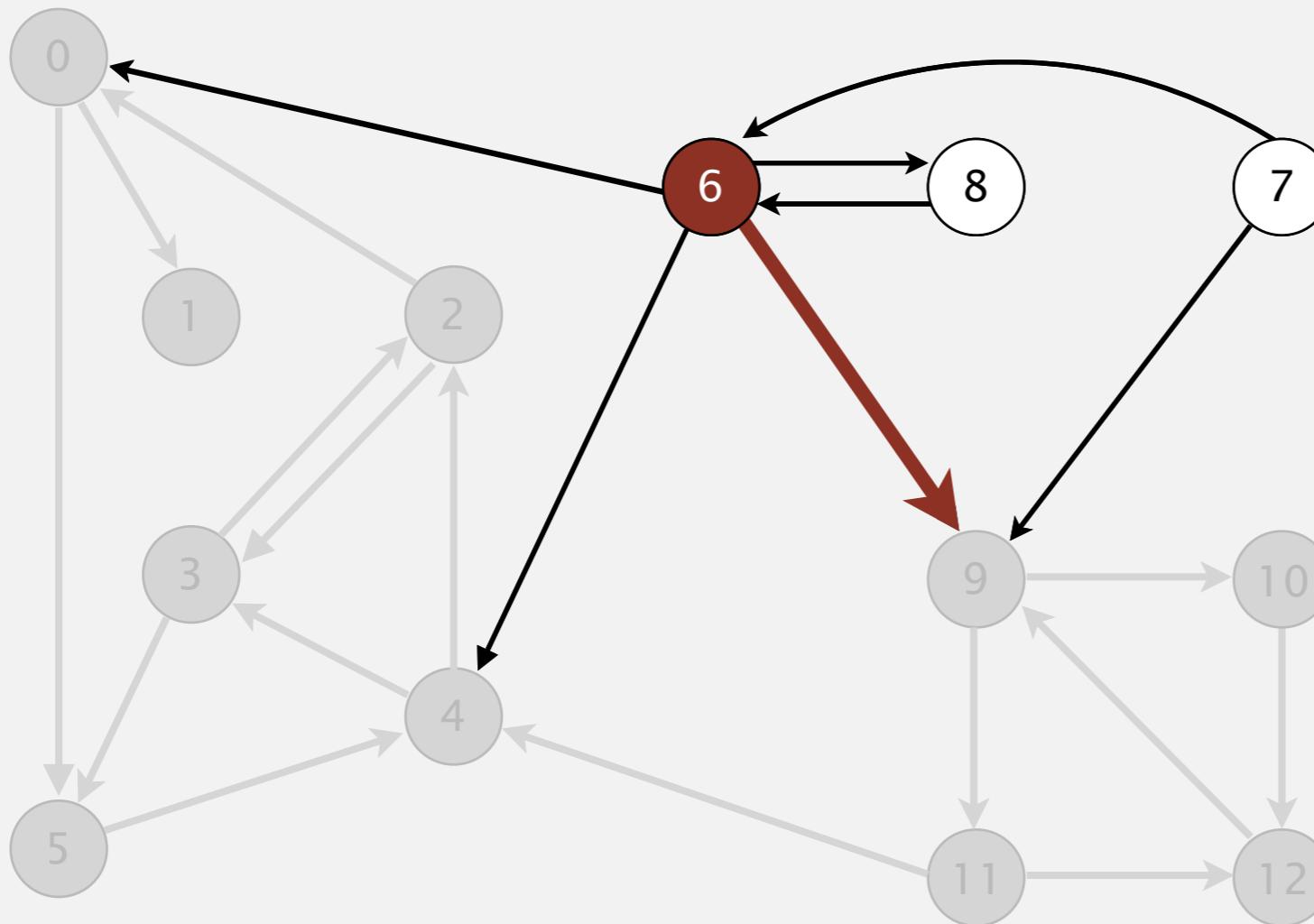
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	-
7	-
8	-
9	2
10	2
11	2
12	2

check 10

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



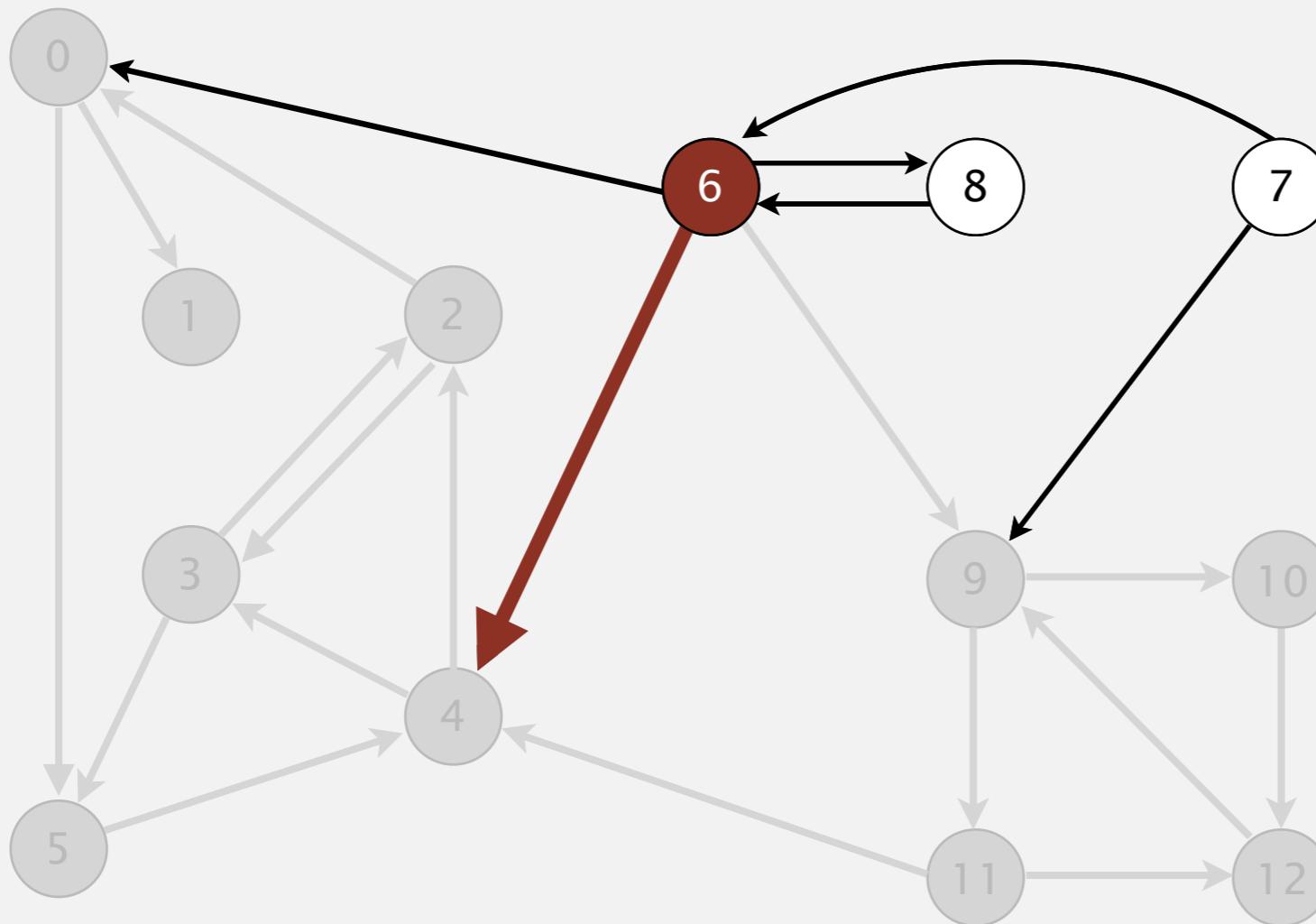
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	-
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



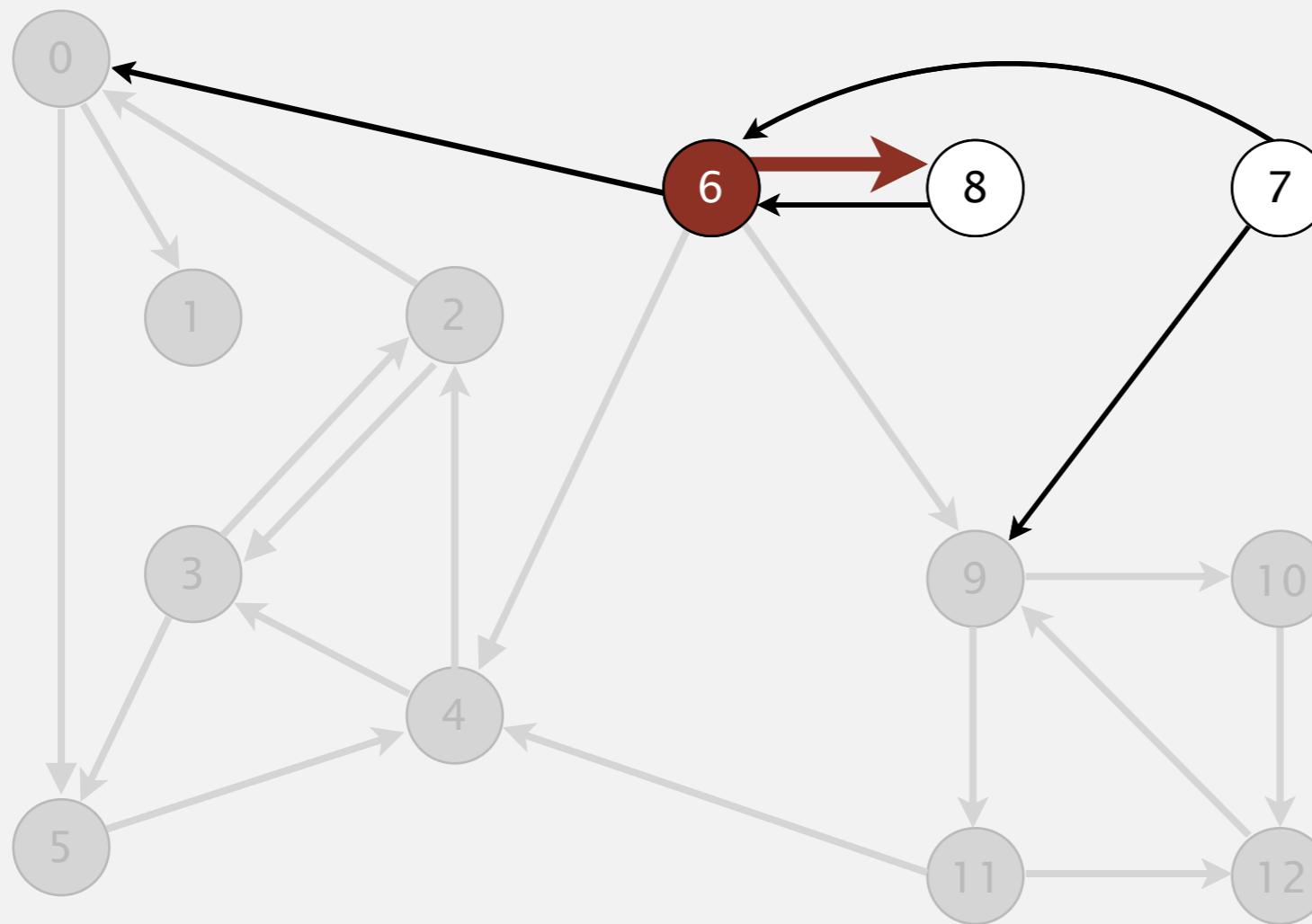
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	-
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



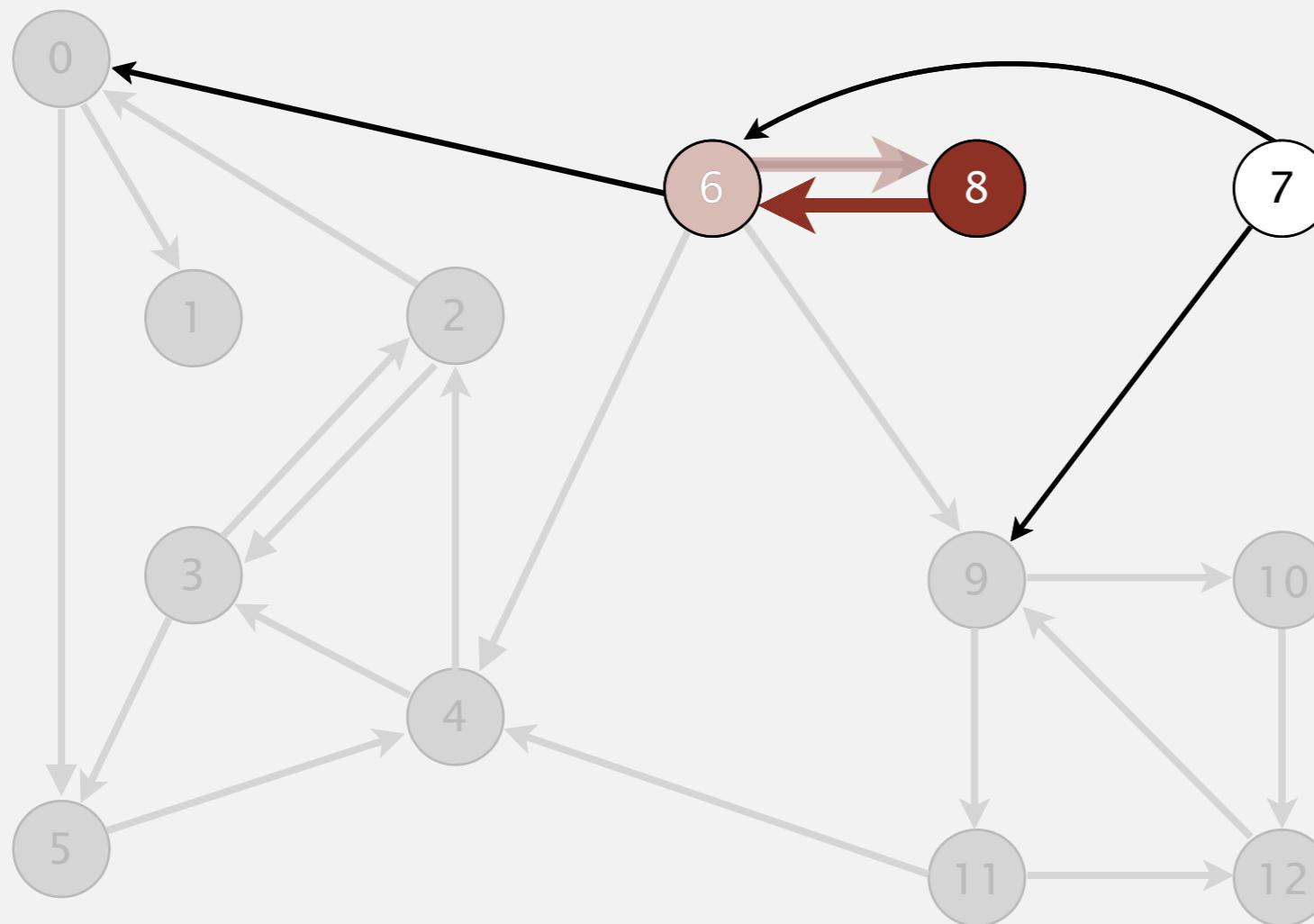
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	-
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



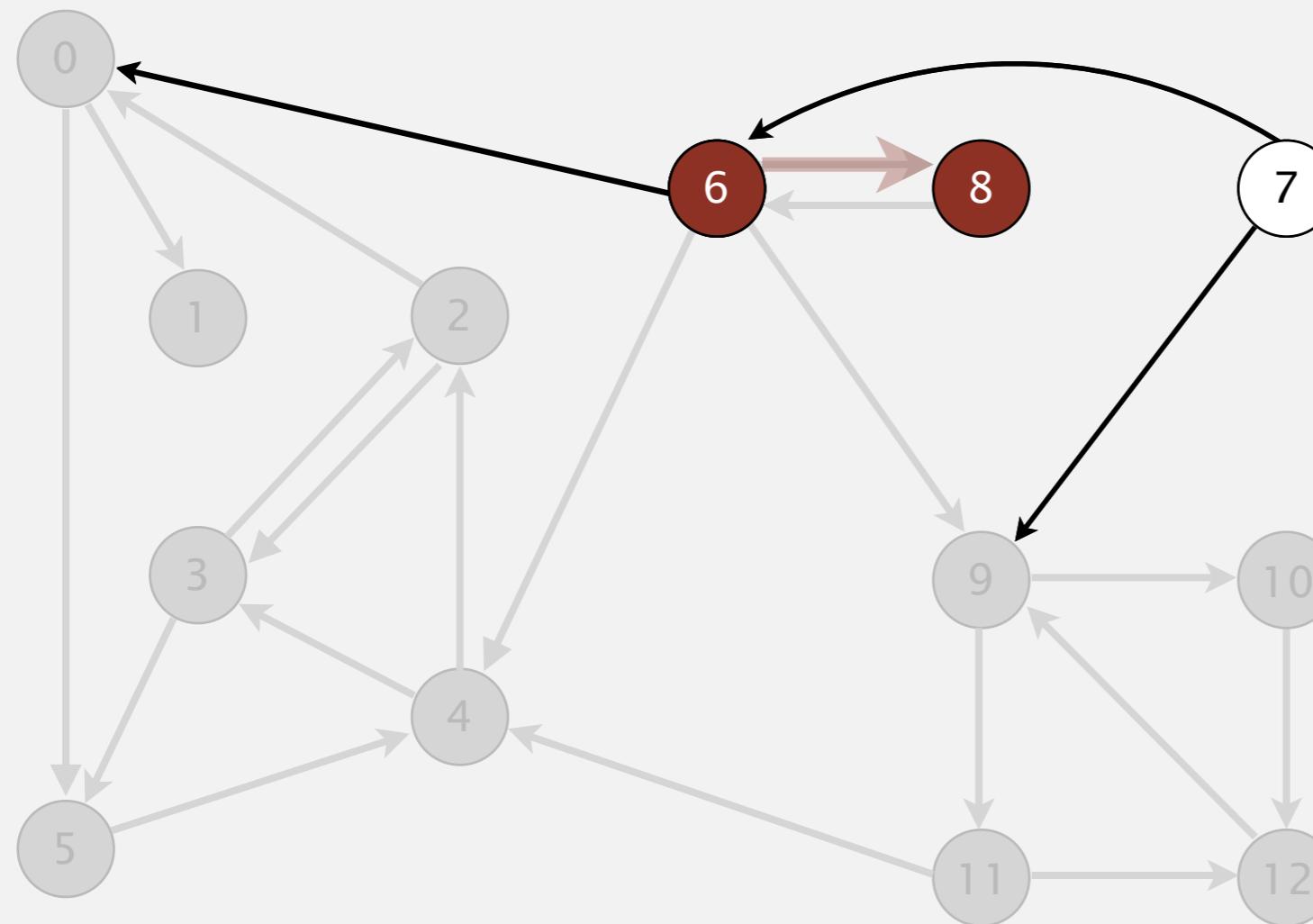
visit 8: check 6

v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



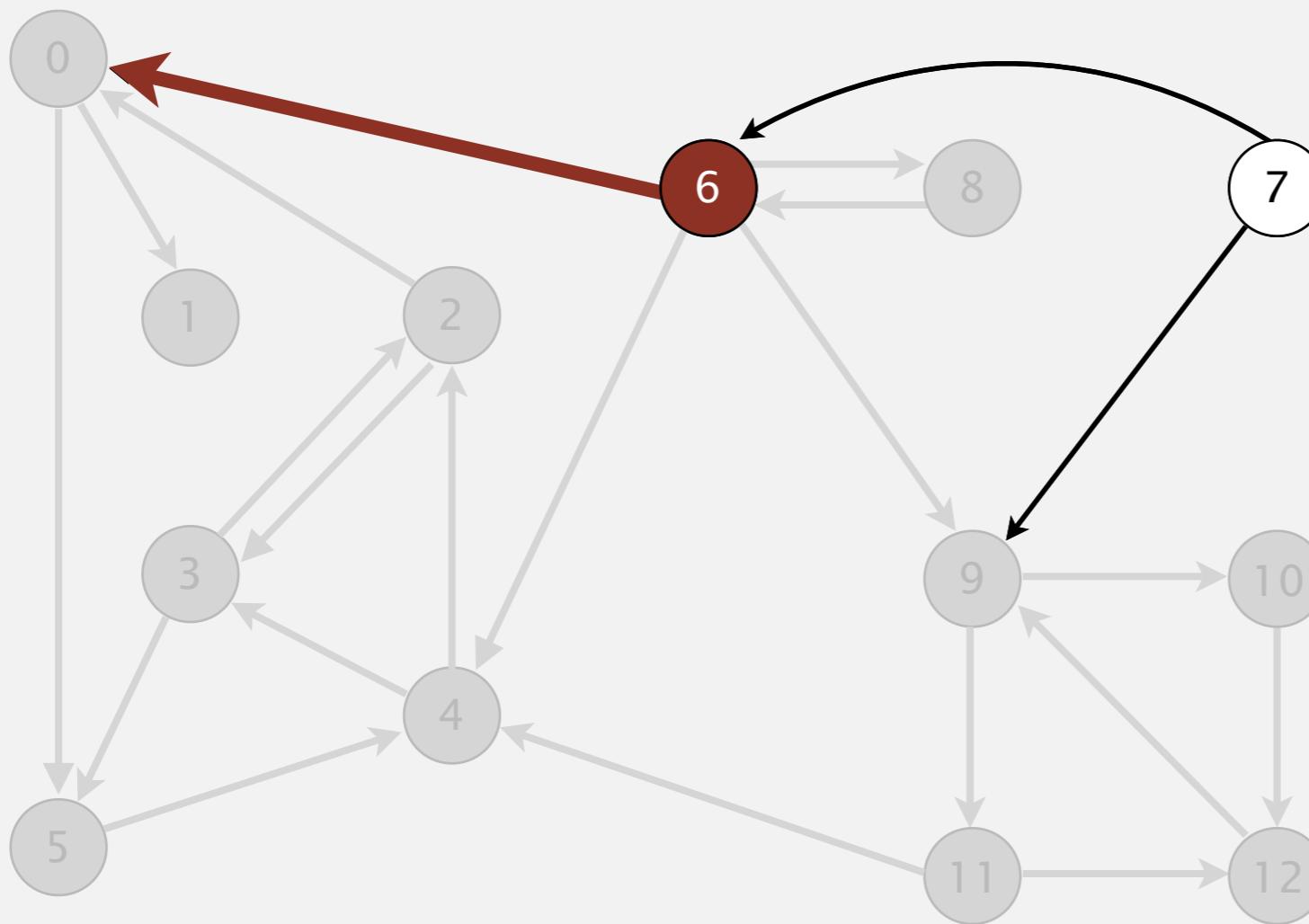
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

8 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



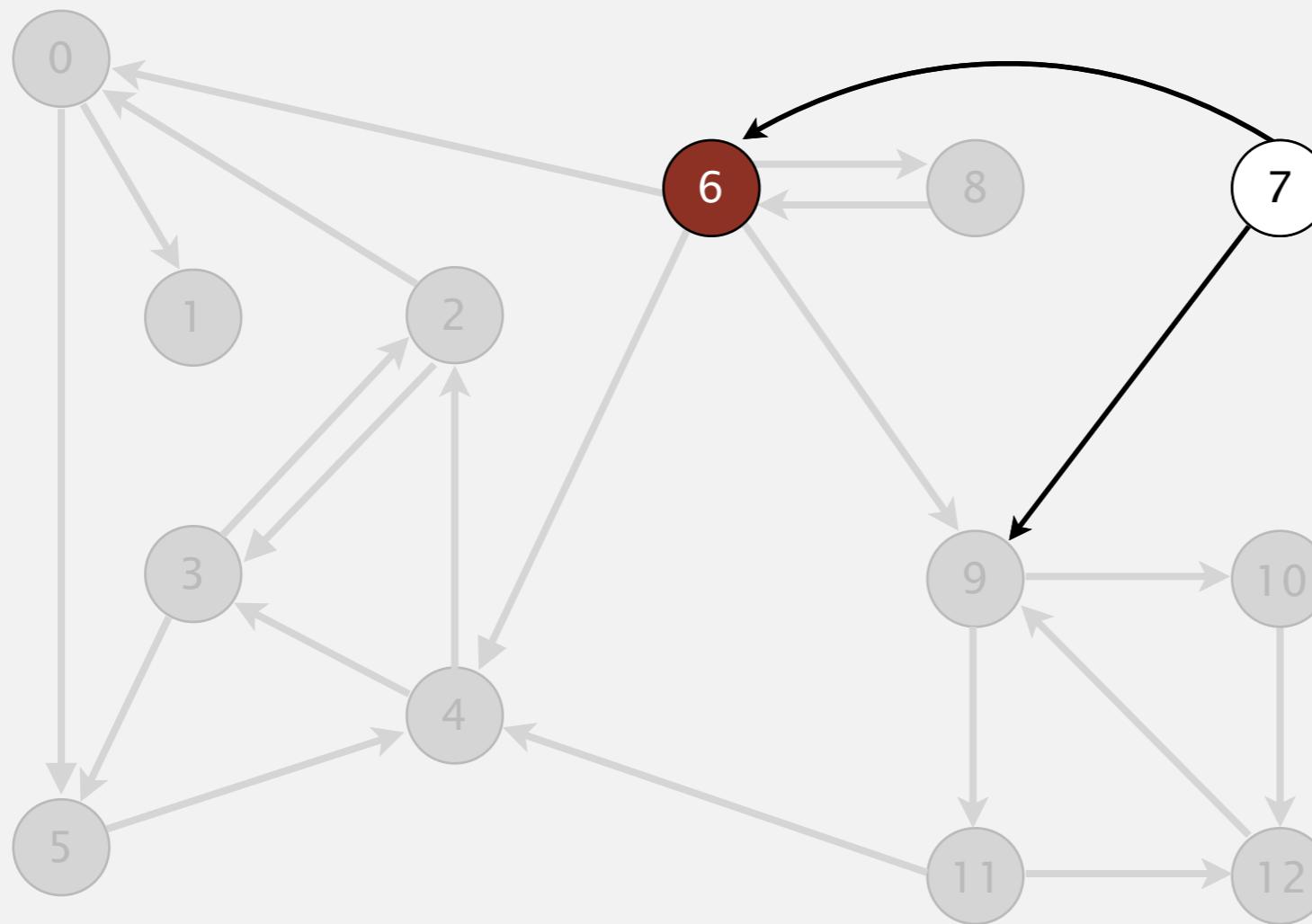
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

visit 6: check 9, check 4, check 8, and check 0

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



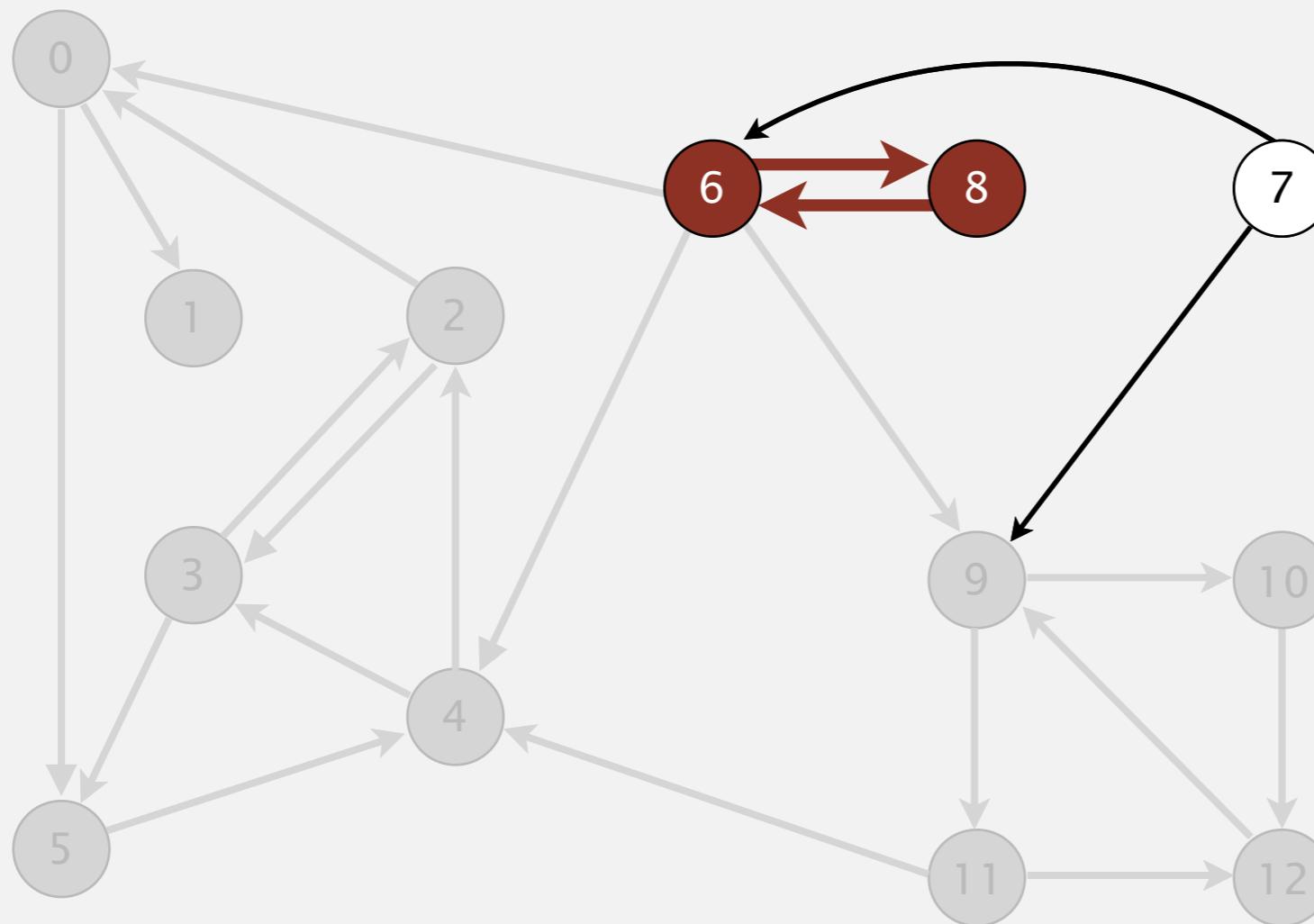
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

6 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



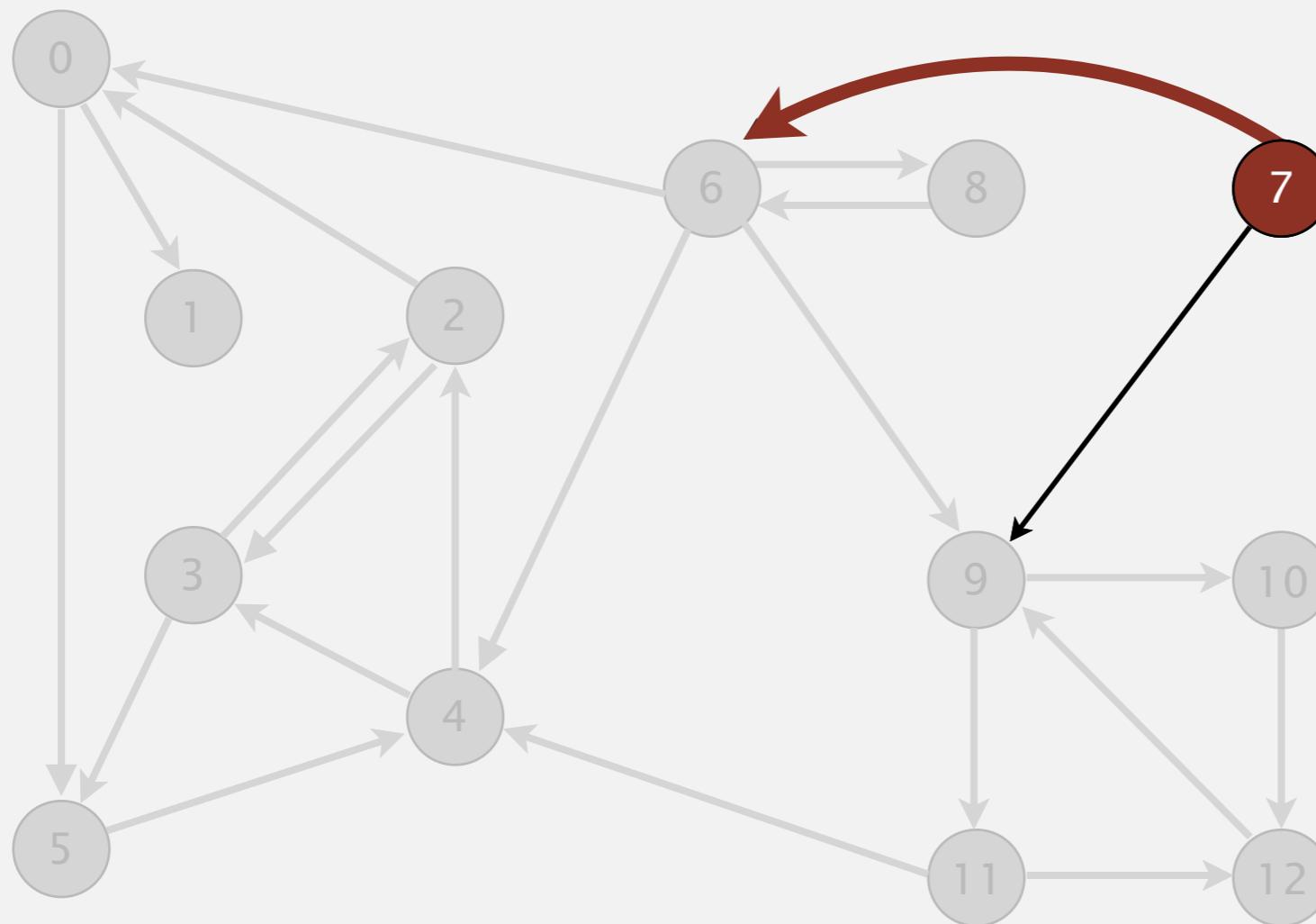
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	-
8	3
9	2
10	2
11	2
12	2

strong component: 6 8

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



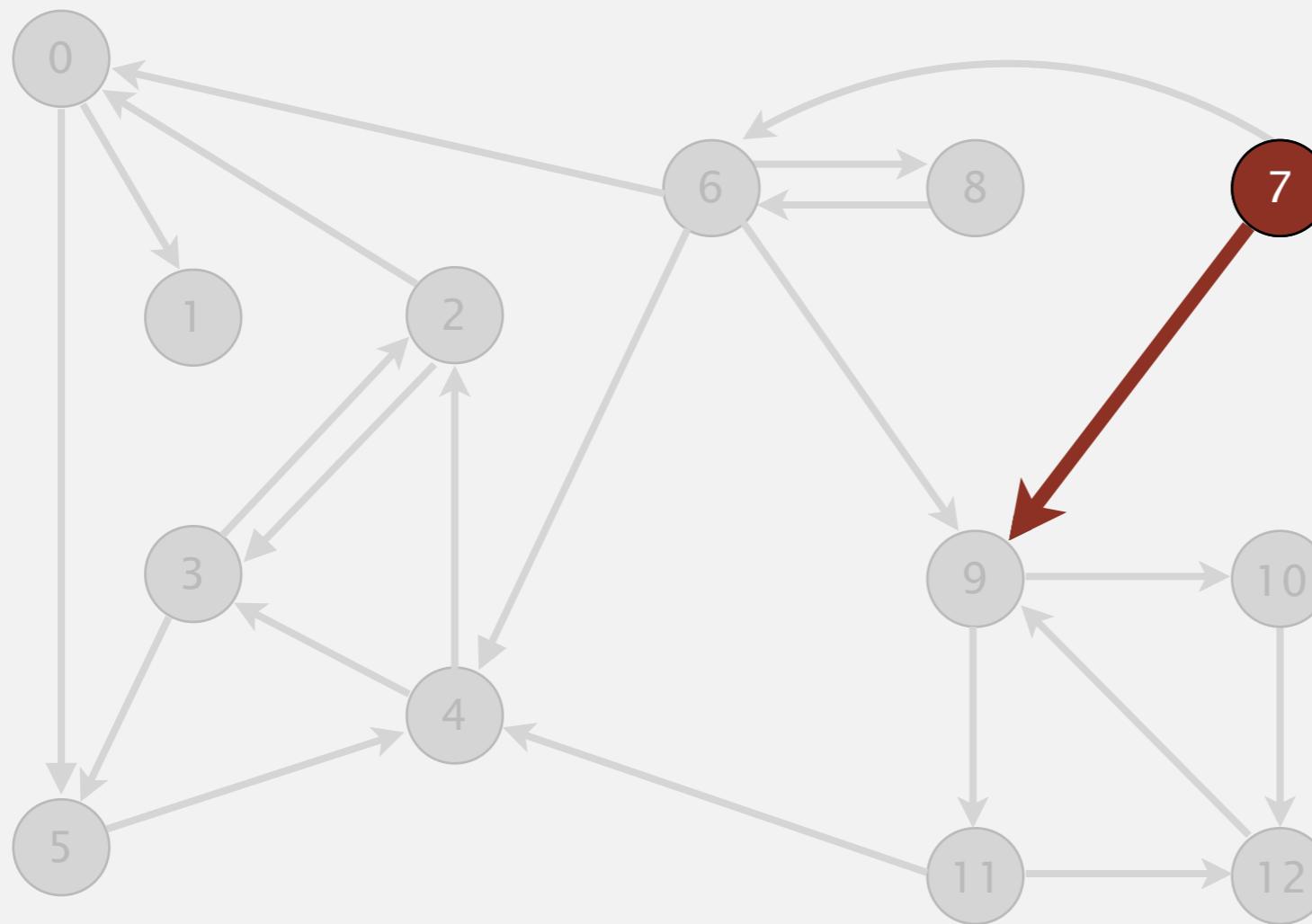
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

visit 7: check 6 and check 9

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



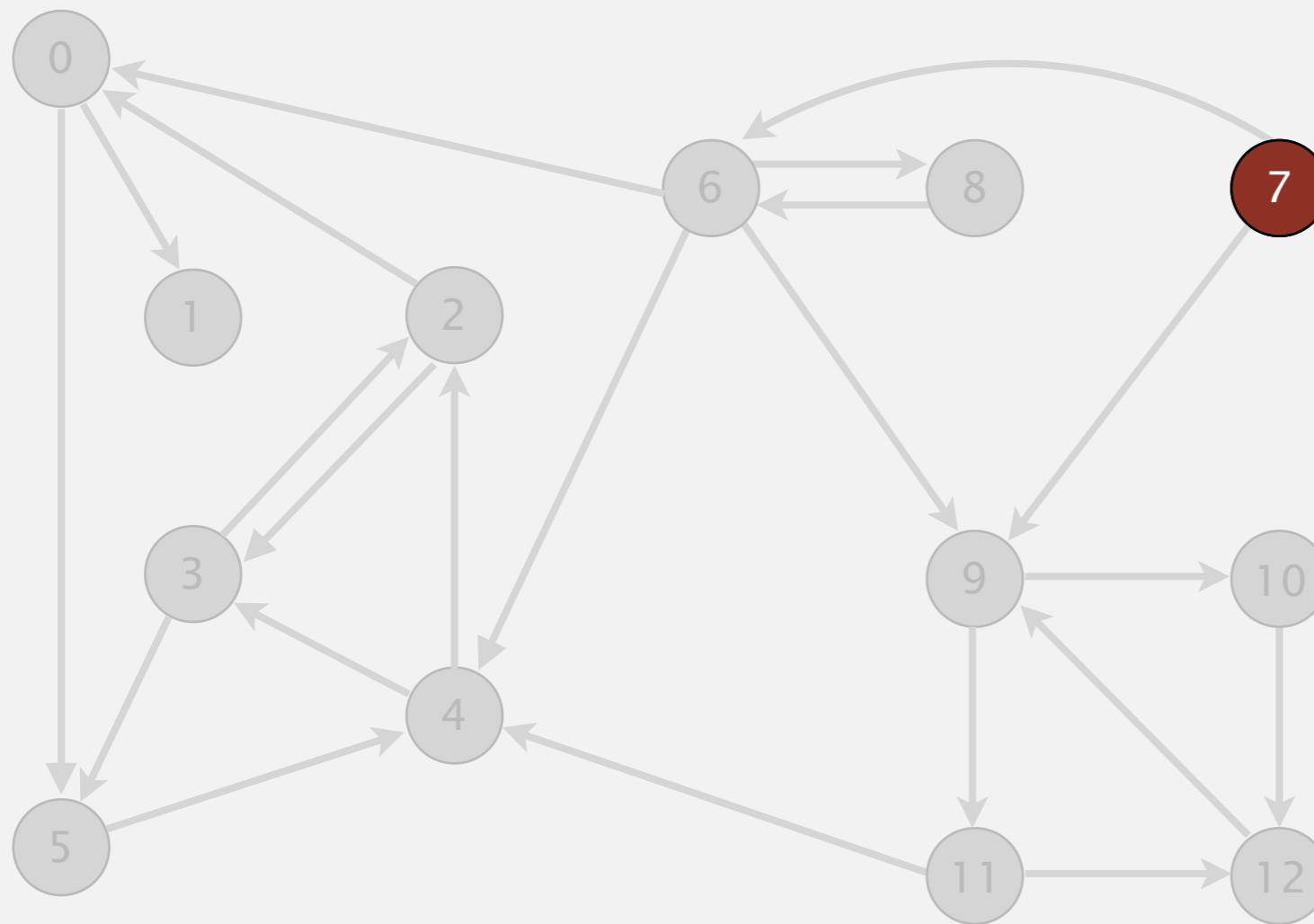
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

visit 7: check 6 and check 9

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



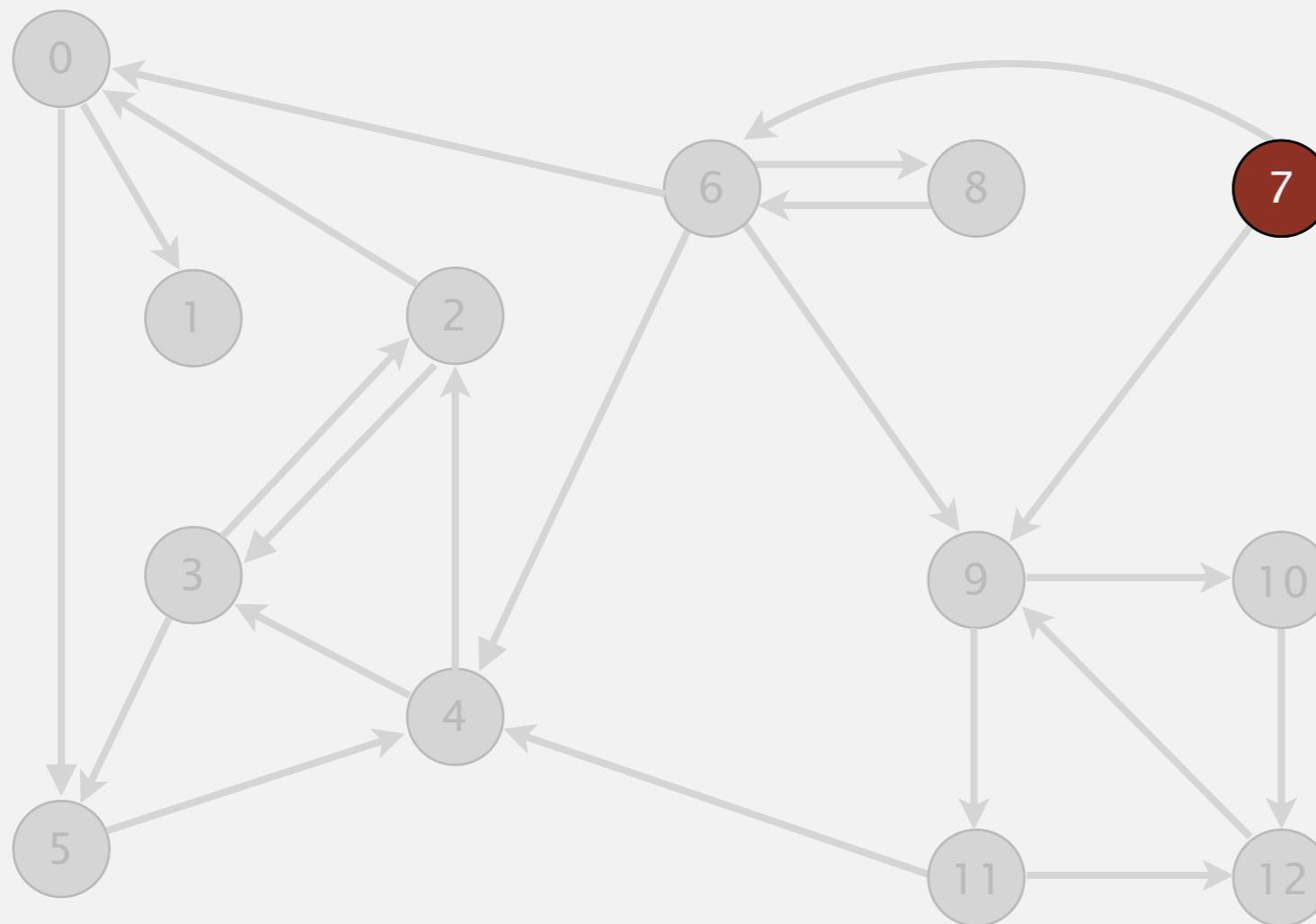
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

7 done

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8

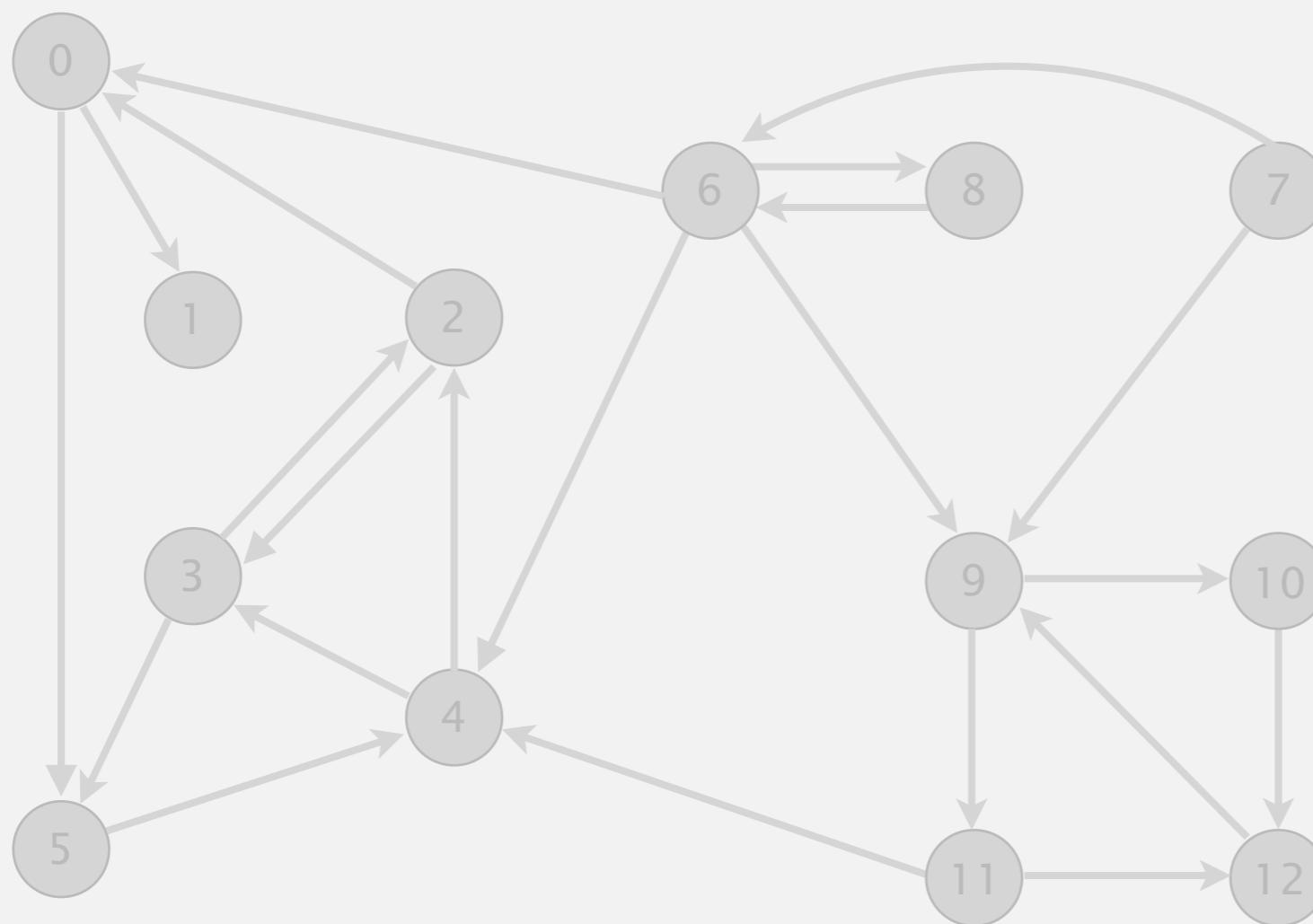


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

strong component: 7

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

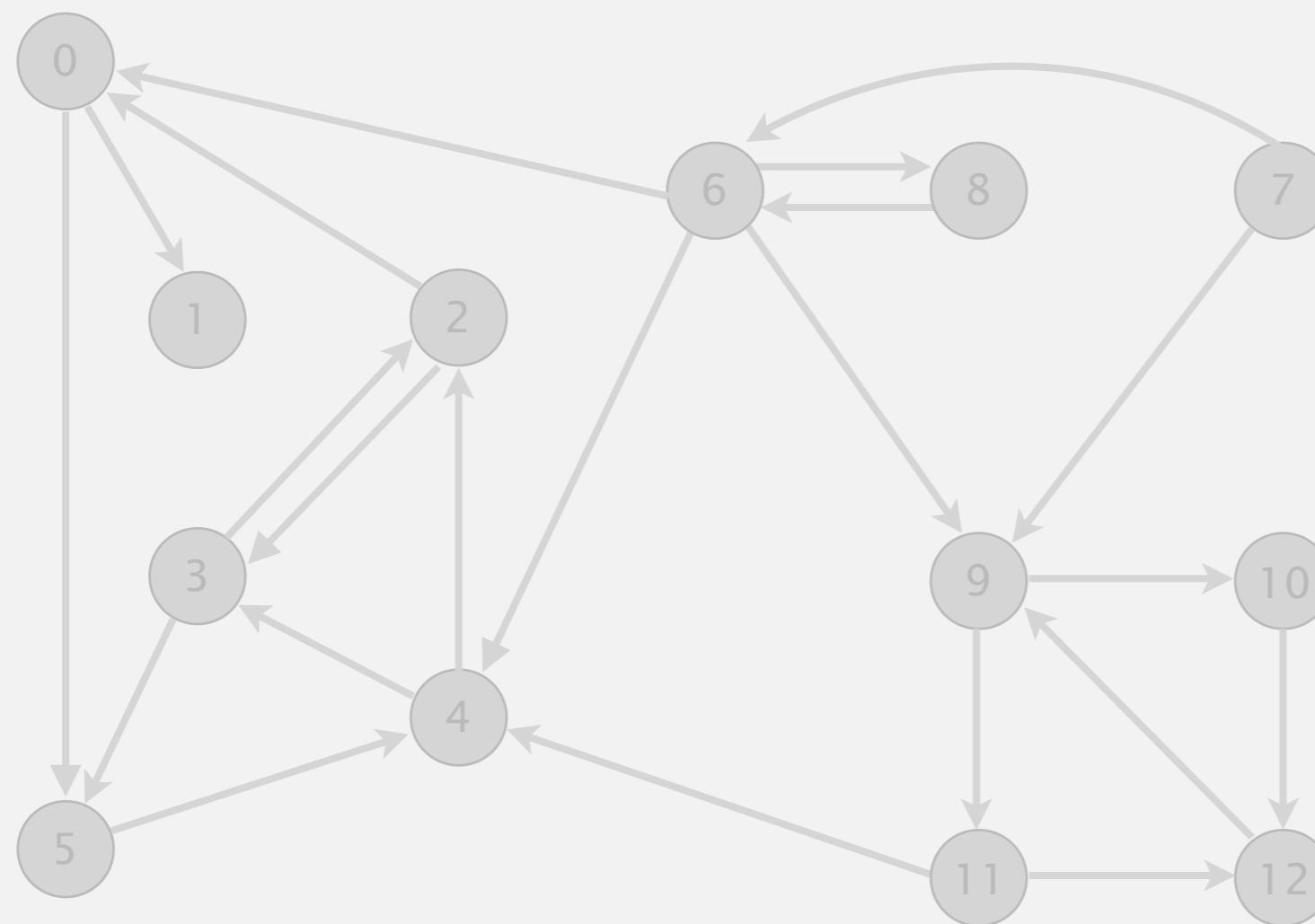


v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

check 8

Kosaraju-Sharir

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .



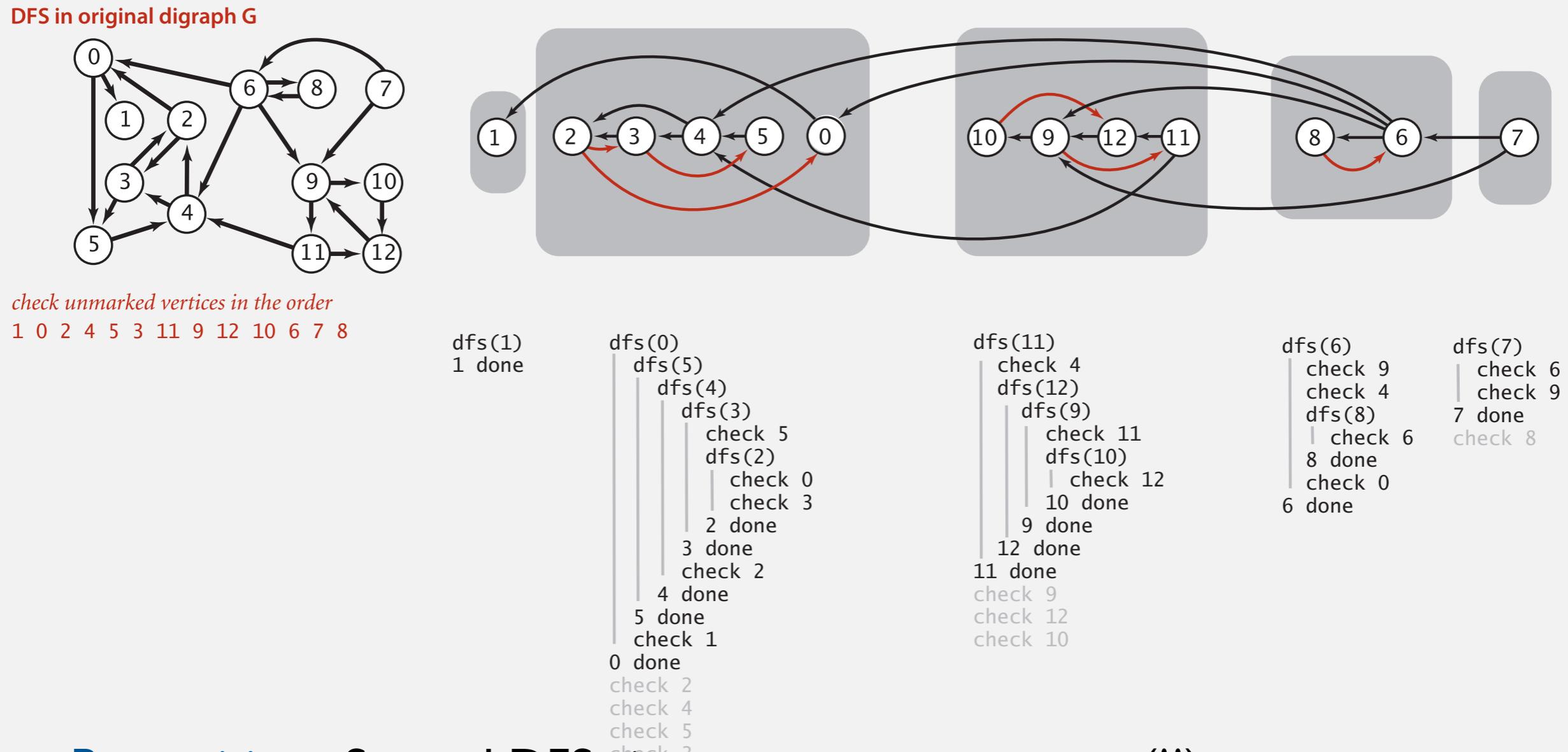
v	scc[v]
0	1
1	0
2	1
3	1
4	1
5	1
6	3
7	4
8	3
9	2
10	2
11	2
12	2

done

Kosaraju's algorithm

Simple (but mysterious) algorithm for computing strong components.

- Run DFS on G^R to compute reverse postorder.
- Run DFS on G , considering vertices in order given by first DFS.



Proposition. Second DFS gives strong components. (!!)

Connected components in an undirected graph (with DFS)

```
public class CC
{
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];

        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean connected(int v, int w)
    { return id[v] == id[w]; }
}
```

Strong components in a digraph (with two DFSs)

```
public class KosarajuSCC
{
    private boolean marked[];
    private int[] id;
    private int count;

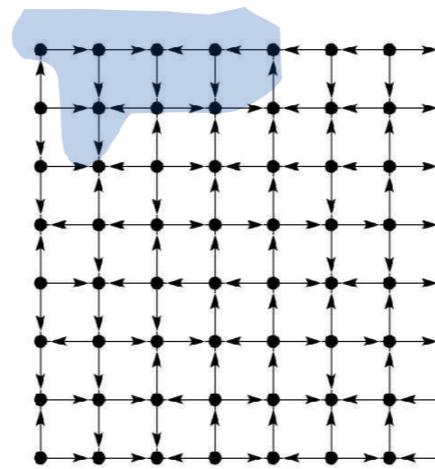
    public KosarajuSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for (int v : dfs.reversePost())
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean stronglyConnected(int v, int w)
    { return id[v] == id[w]; }
}
```

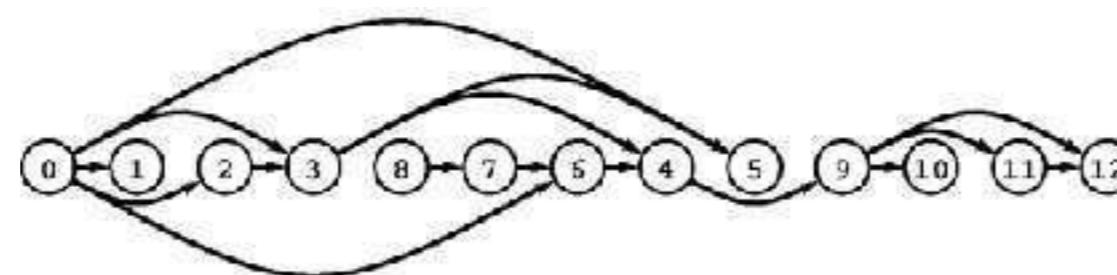
Digraph-processing summary: algorithms of the day

single-source
reachability



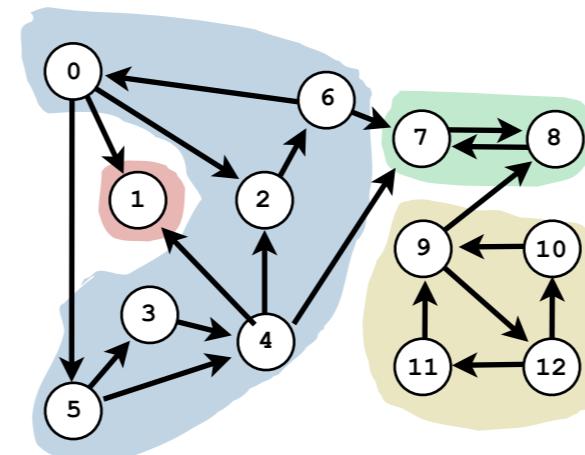
DFS

topological sort
(DAG)



DFS

strong
components



Kosaraju
DFS (twice)

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

MINIMUM SPANNING TREES

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

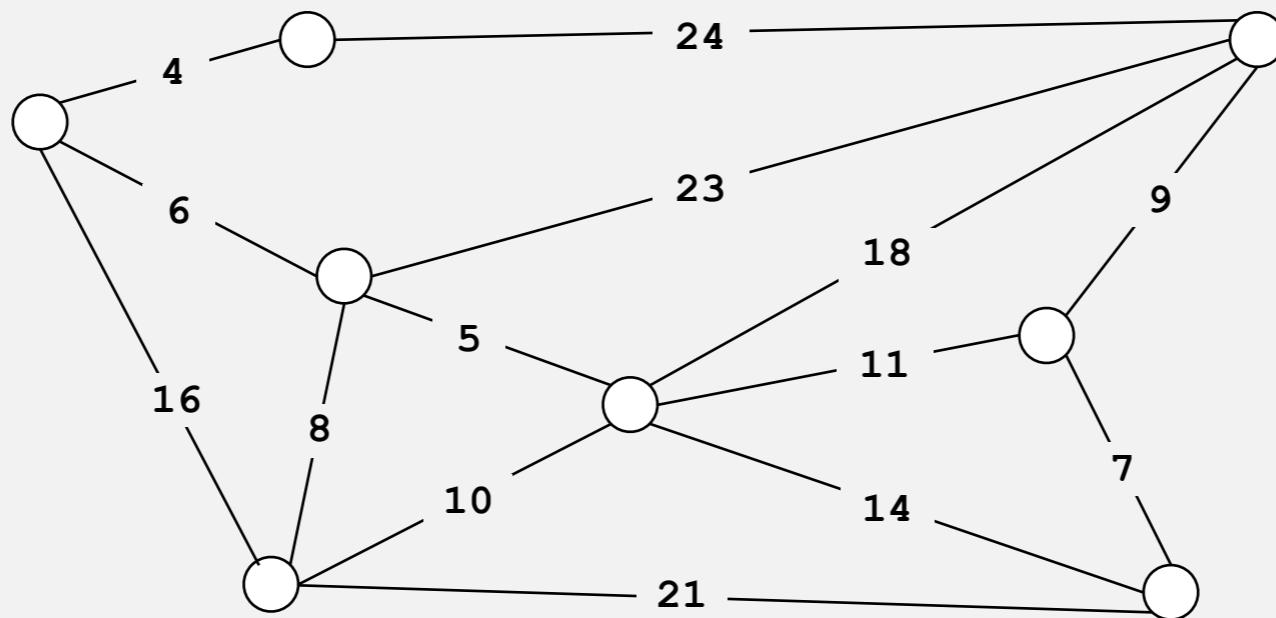
- ▶ **Minimum Spanning Trees**
- ▶ **Greedy algorithm**
- ▶ **Edge-weighted graph API**
- ▶ **Kruskal's algorithm**
- ▶ **Prim's algorithm**
- ▶ **Context**

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A **spanning tree** of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



graph G

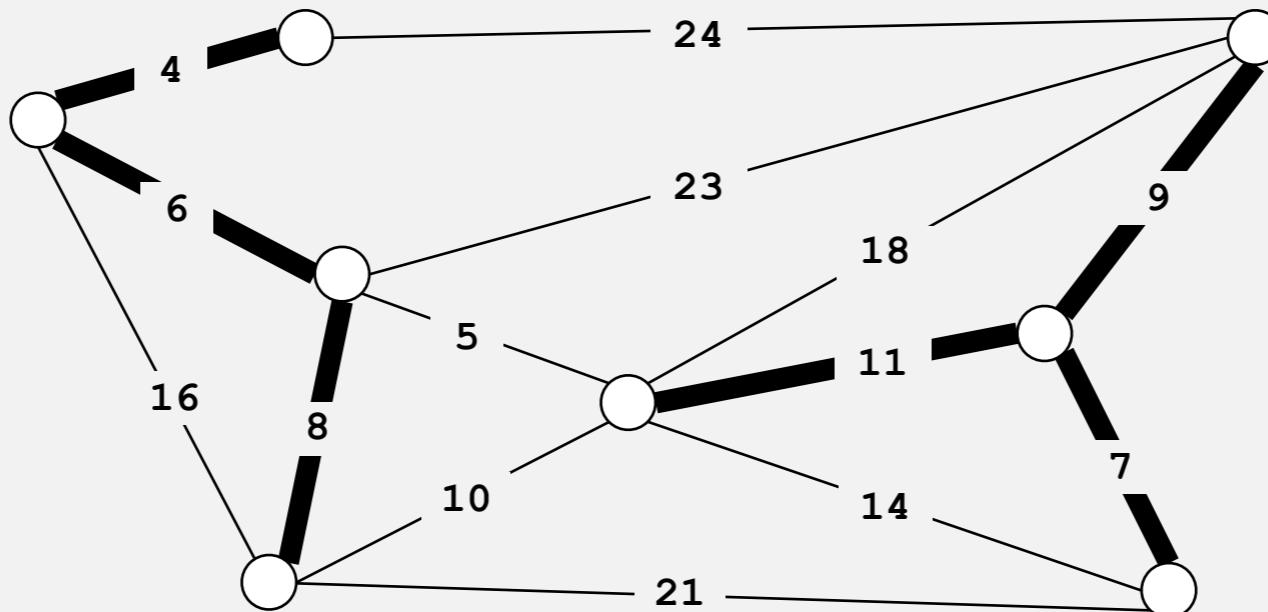
a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A **spanning tree** of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.

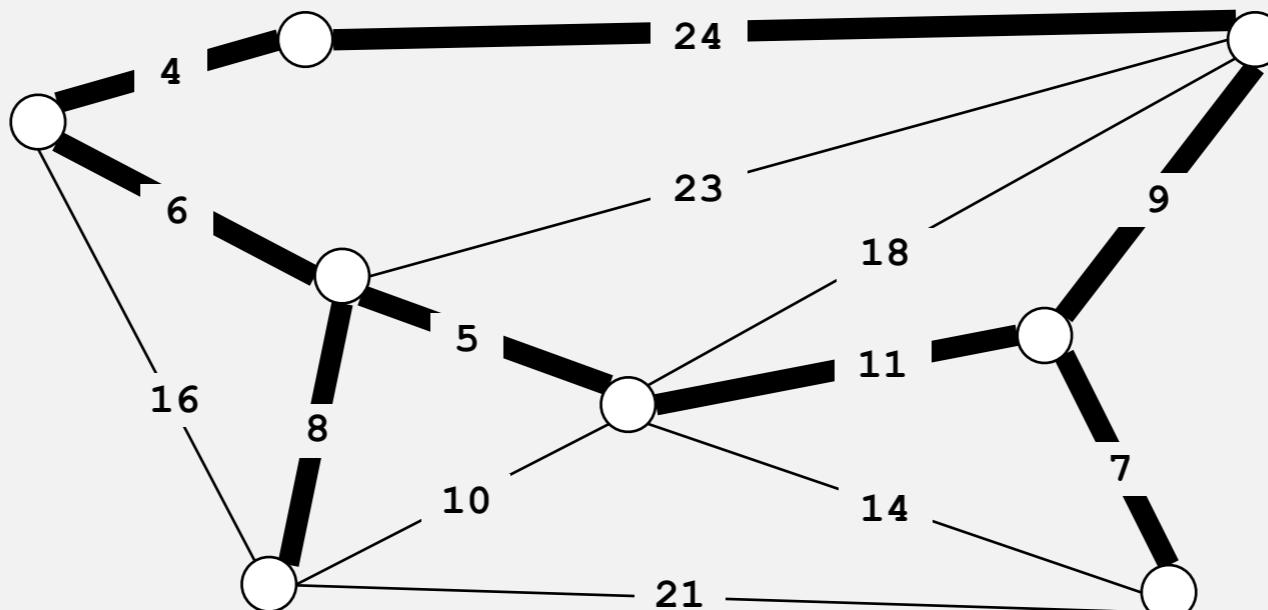


Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A **spanning tree** of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



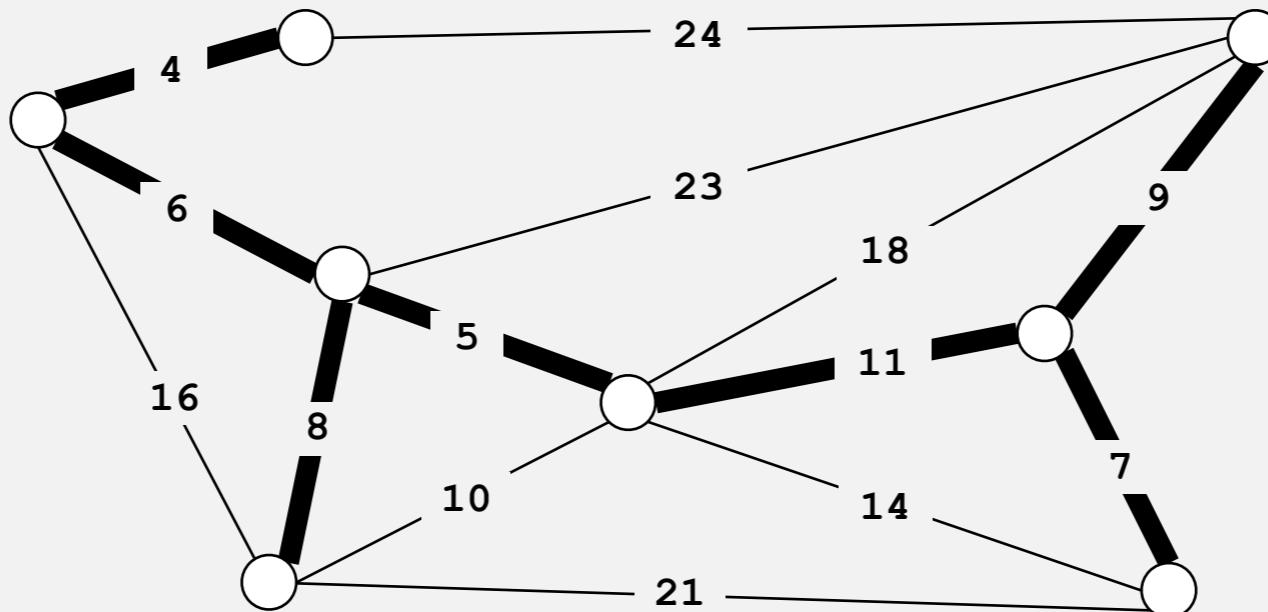
not acyclic

Minimum spanning tree

Given. Undirected graph G with positive edge weights (connected).

Def. A **spanning tree** of G is a subgraph T that is connected and acyclic.

Goal. Find a min weight spanning tree.



spanning tree T: cost = $50 = 4 + 6 + 8 + 5 + 11 + 9 + 7$

Brute force. Try all spanning trees?

Applications

MST is fundamental problem with diverse applications.

- Dithering.
- Cluster analysis.
- Max bottleneck paths.
- Real-time face verification.
- LDPC codes for error correction.
- Image registration with Renyi entropy.
- Find road networks in satellite and aerial imagery.
- Reducing data storage in sequencing amino acids in a protein.
- Model locality of particle interactions in turbulent fluid flows.
- Autoconfig protocol for Ethernet bridging to avoid cycles in a network.
- Approximation algorithms for NP-hard problems (e.g., TSP, Steiner tree).
- Network design (communication, electrical, hydraulic, cable, computer, road).

<http://www.ics.uci.edu/~eppstein/gina/mst.html>

MINIMUM SPANNING TREES

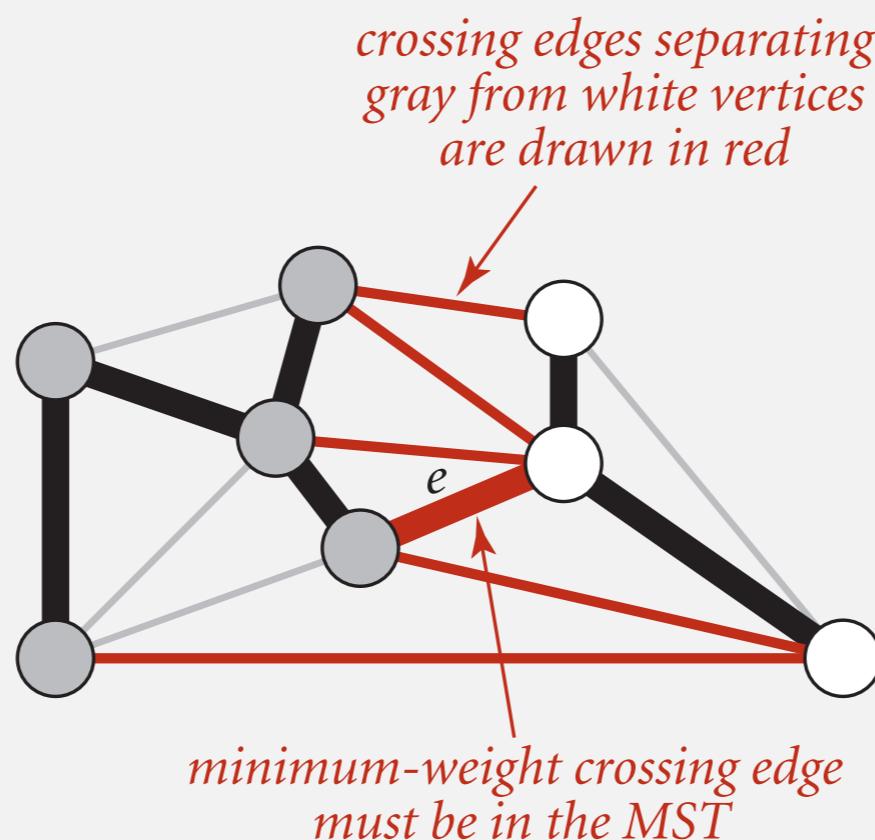
- ▶ Greedy algorithm
- ▶ Edge-weighted graph API
- ▶ Kruskal's algorithm
- ▶ Prim's algorithm
- ▶ Context

Cut property

Simplifying assumptions. Edge weights are distinct; graph is connected.

Def. A **cut** in a graph is a partition of its vertices into two (nonempty) sets. A **crossing edge** connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.



Cut property: correctness proof

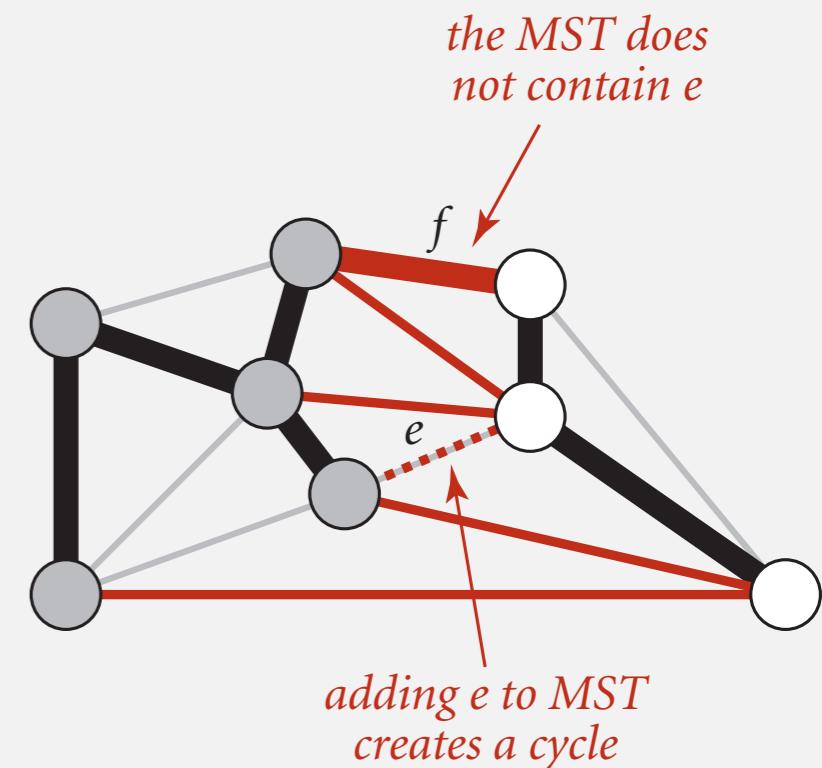
Simplifying assumptions. Edge weights are distinct; graph is connected.

Def. A **cut** in a graph is a partition of its vertices into two (nonempty) sets. A **crossing edge** connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

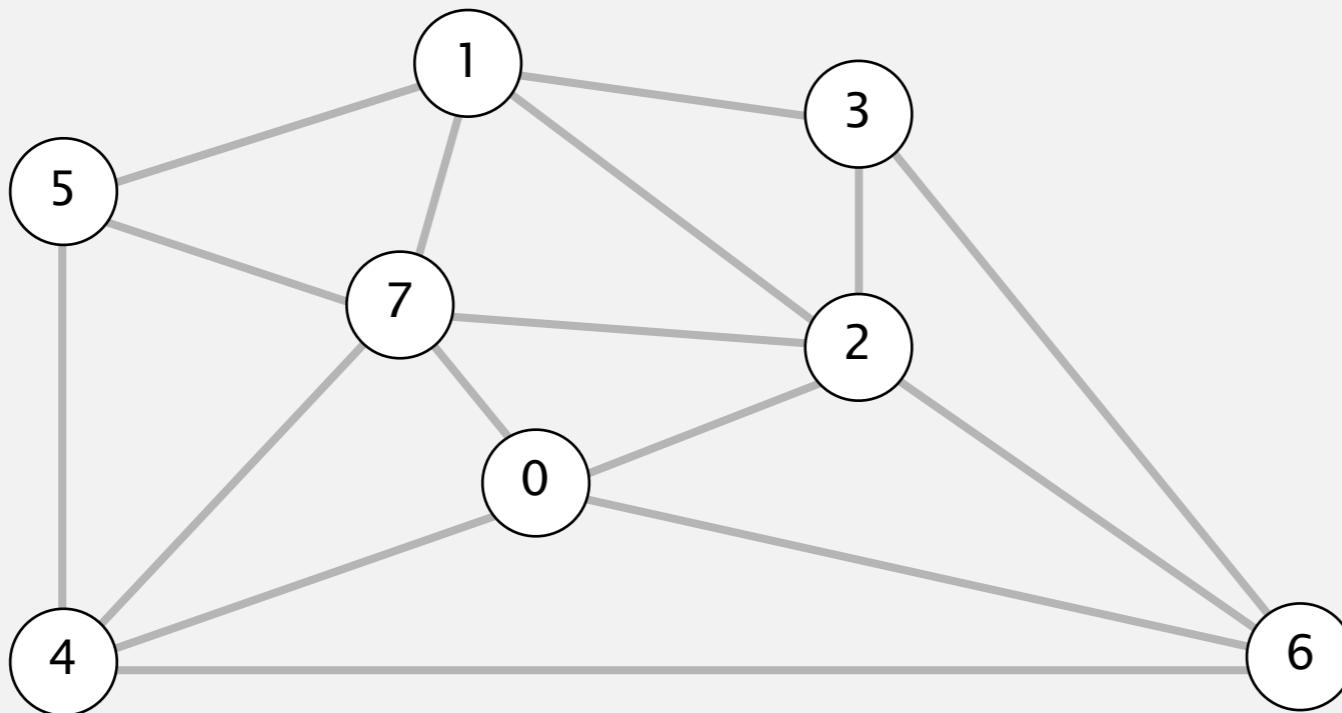
Pf. Let e be the min-weight crossing edge in cut.

- Suppose e is not in the MST.
- Adding e to the MST creates a cycle.
- Some other edge f in cycle must be a crossing edge.
- Removing f and adding e is also a spanning tree.
- Since weight of e is less than the weight of f , that spanning tree is lower weight.
- Contradiction. ■



Greedy MST algorithm

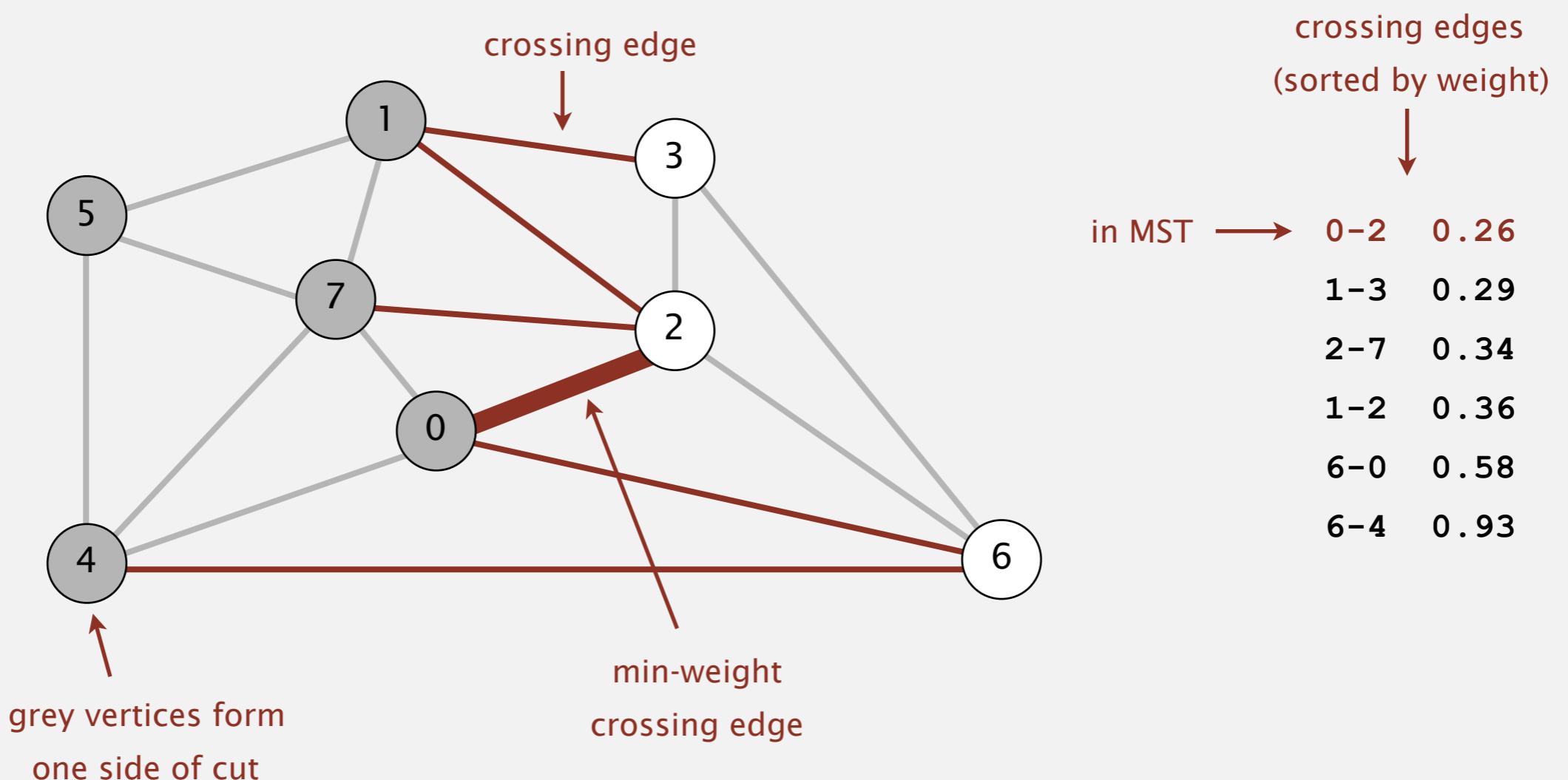
- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



an edge-weighted graph

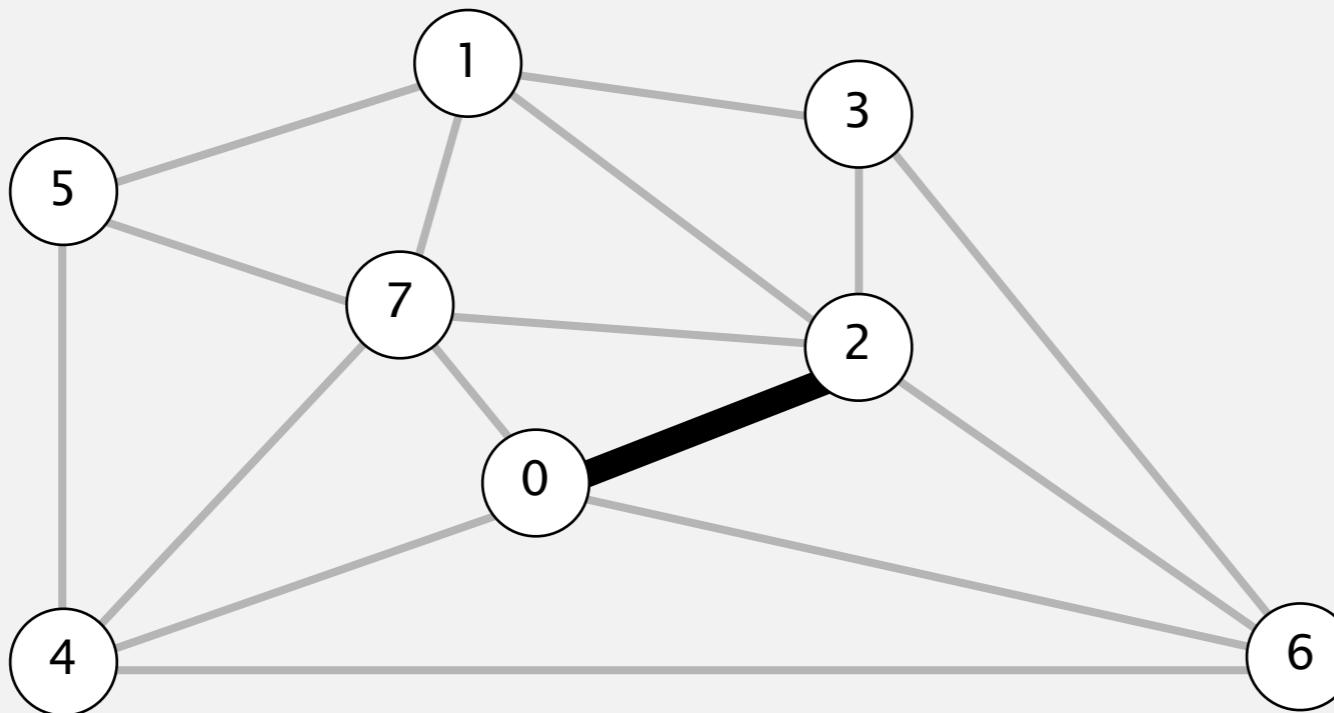
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

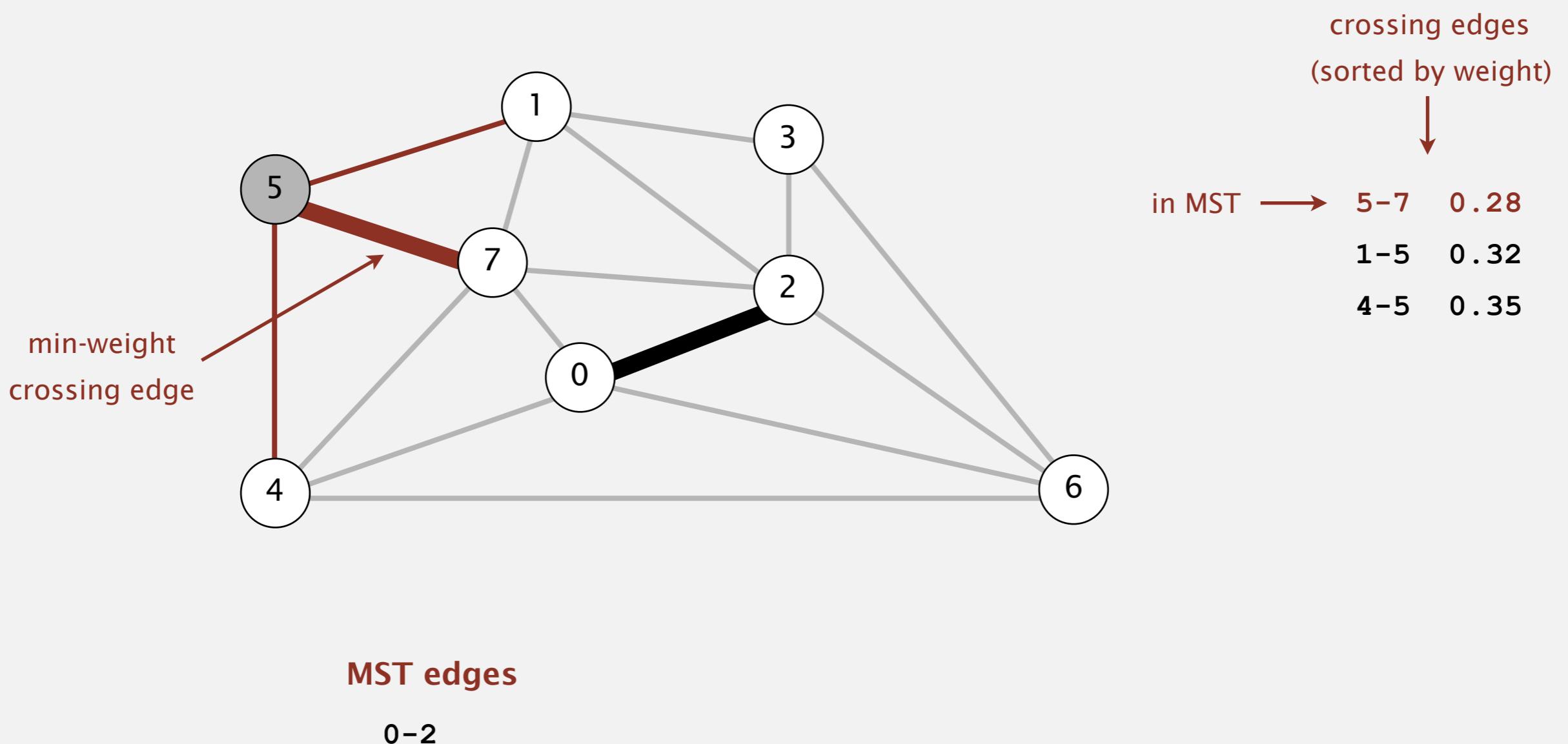


MST edges

0-2

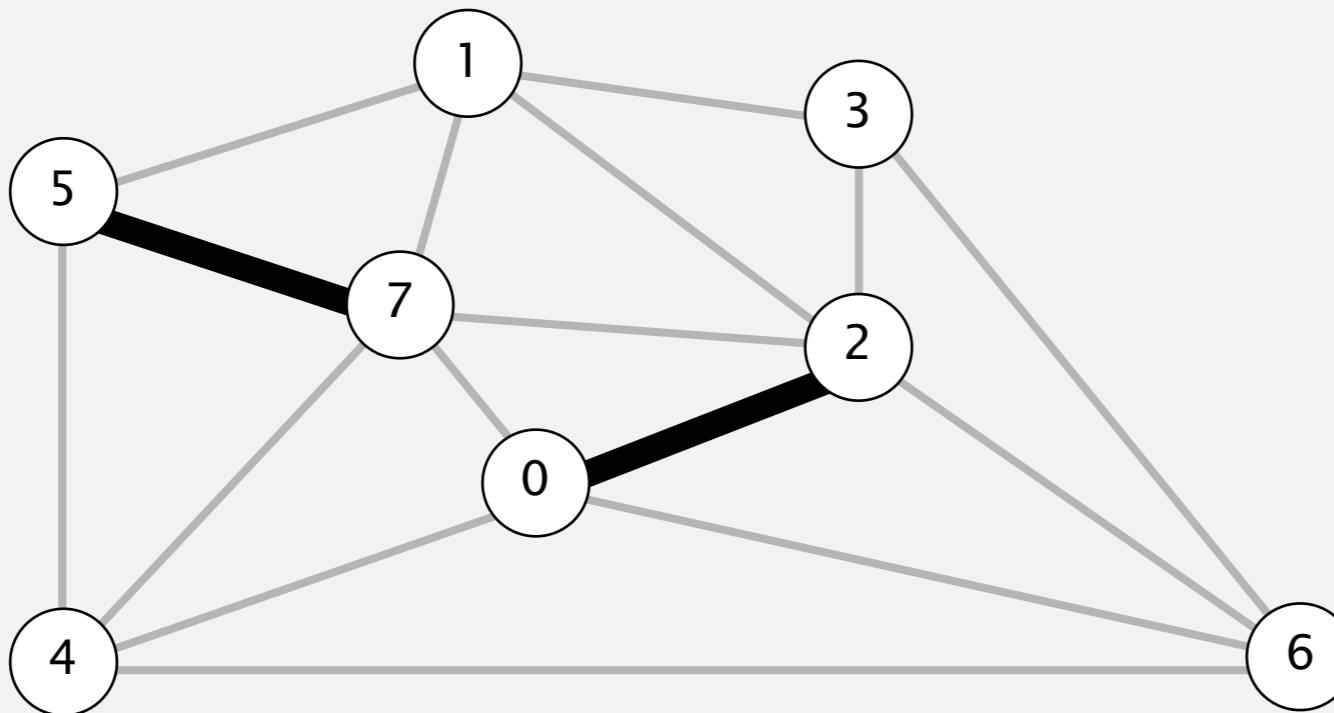
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



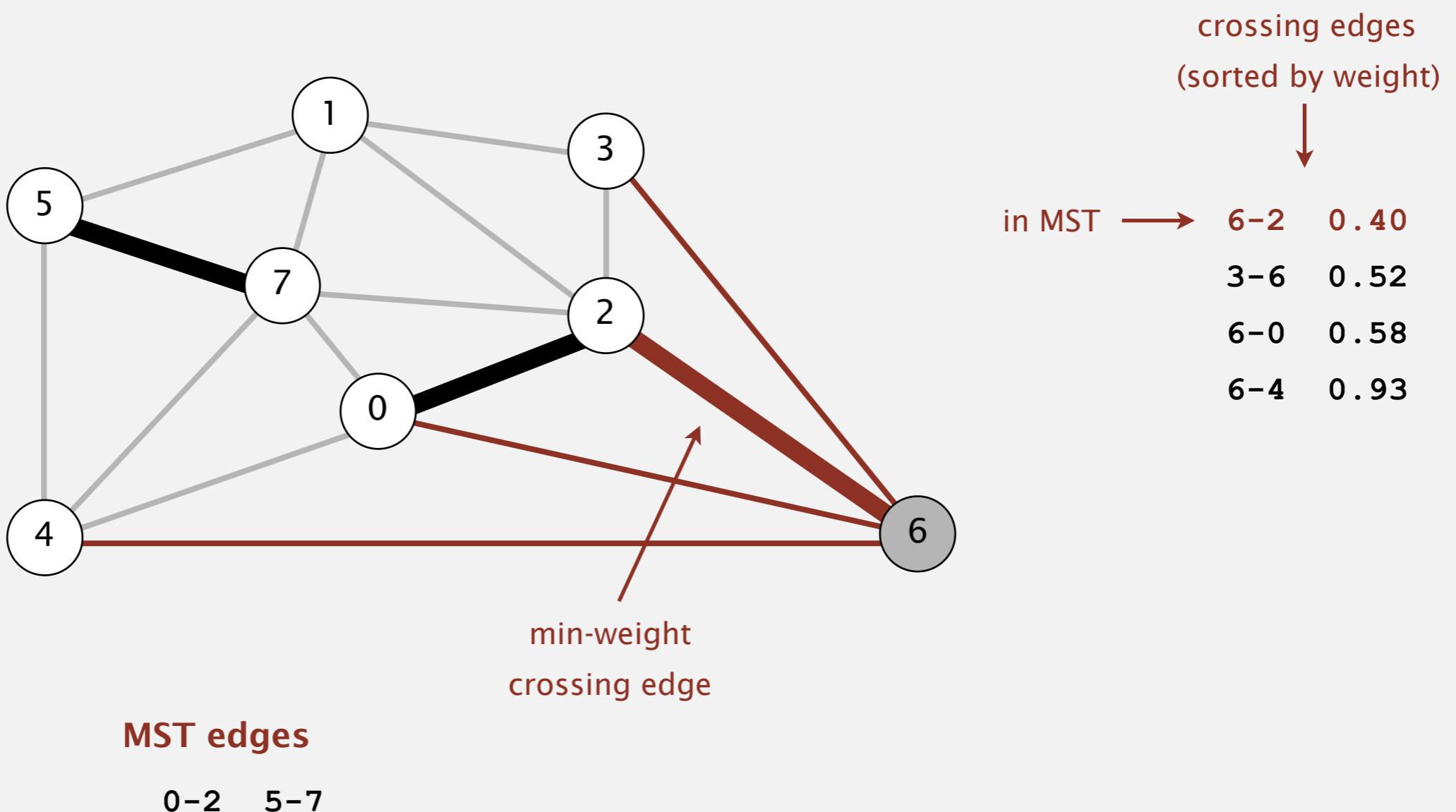
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



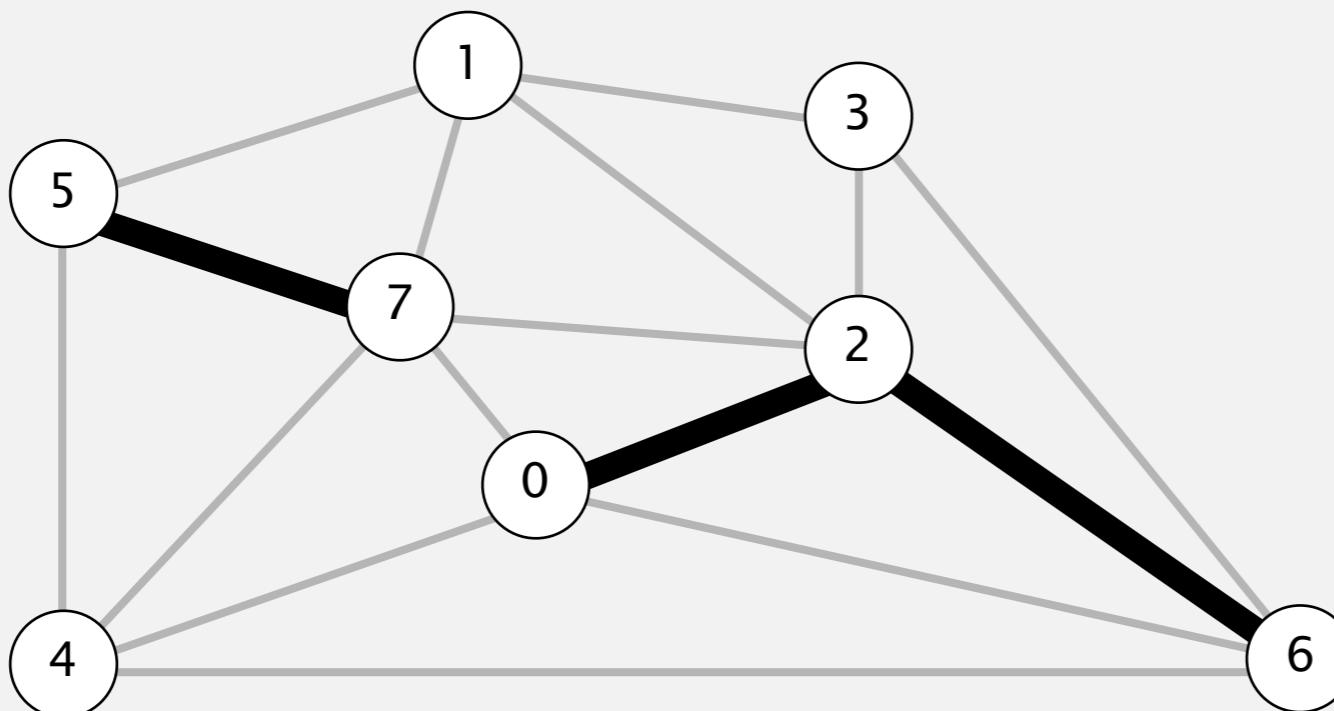
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm

- Start with all edges colored gray.
 - Find a cut with no black crossing edges, and color its min-weight edge black.
 - Repeat until $V - 1$ edges are colored black.

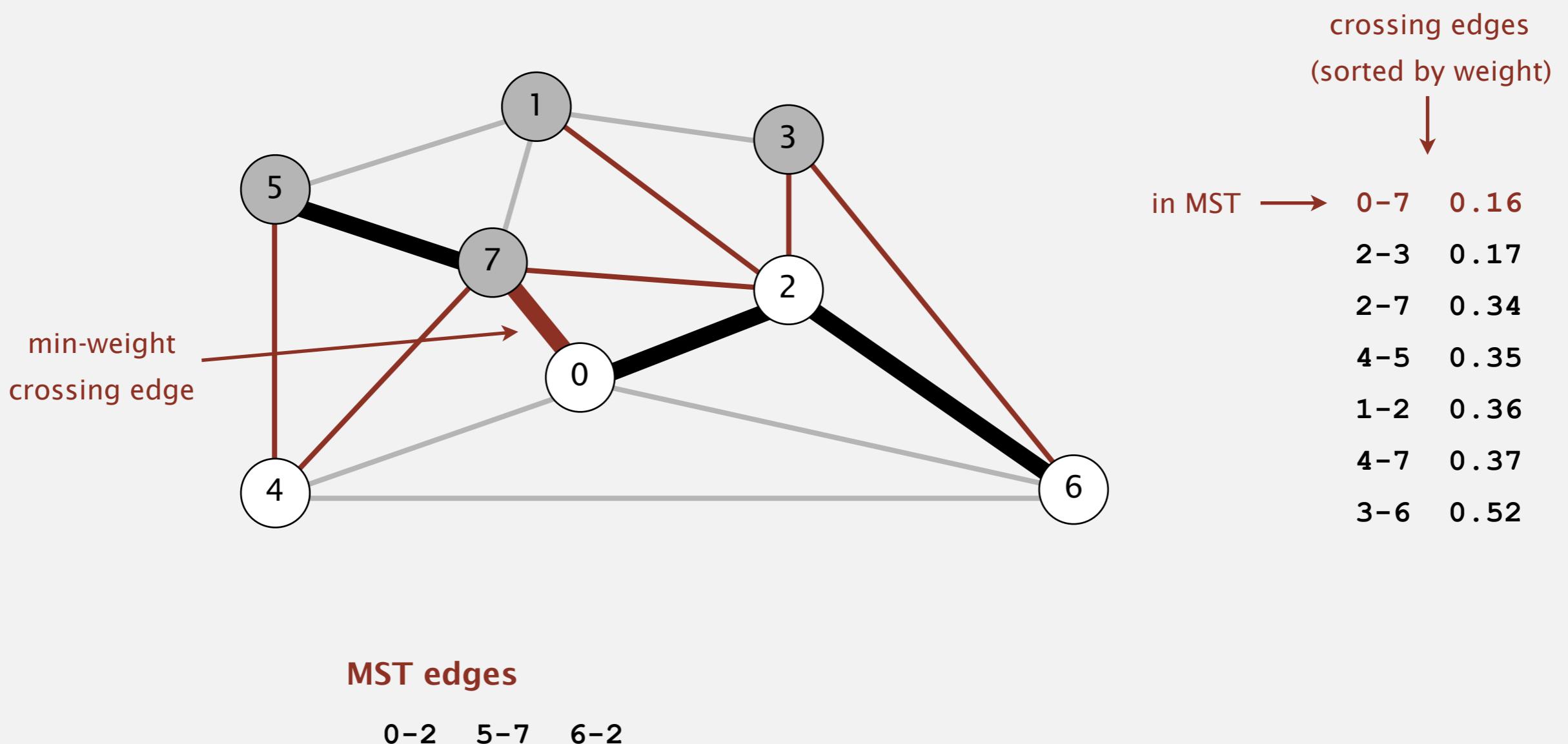


MST edges

0-2 5-7 6-2

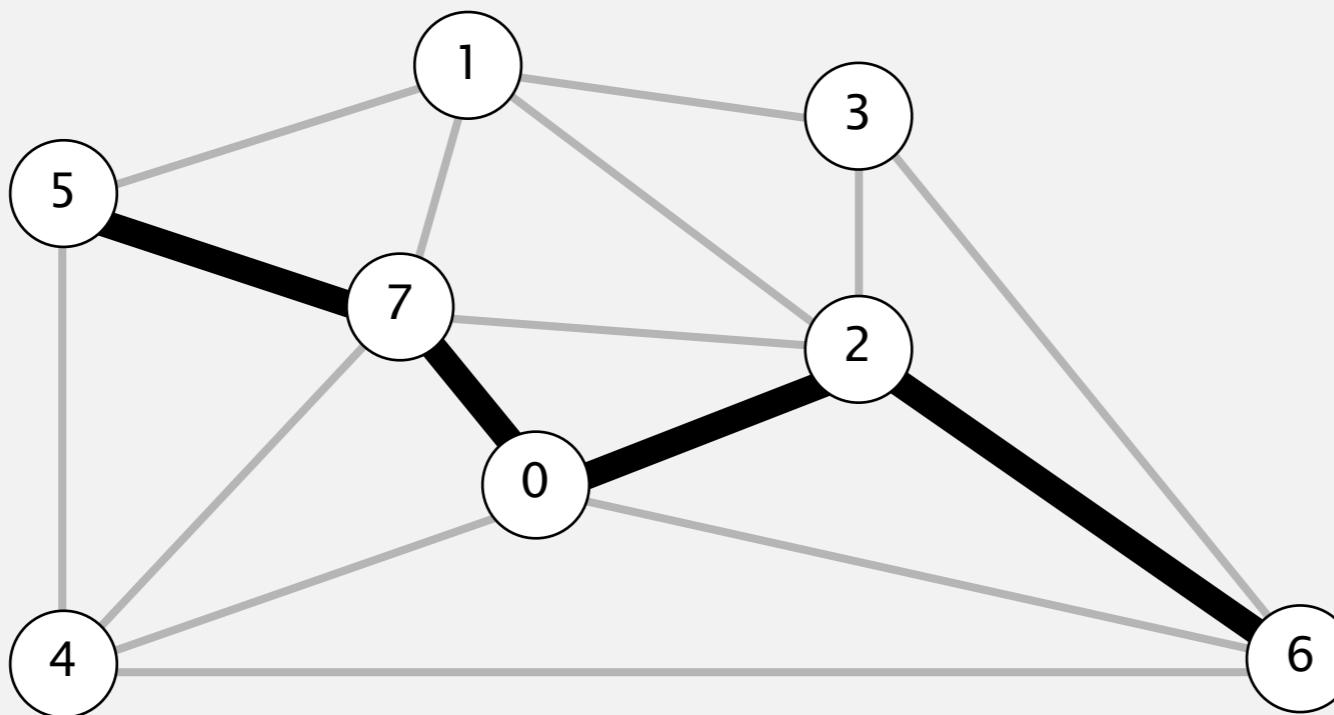
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

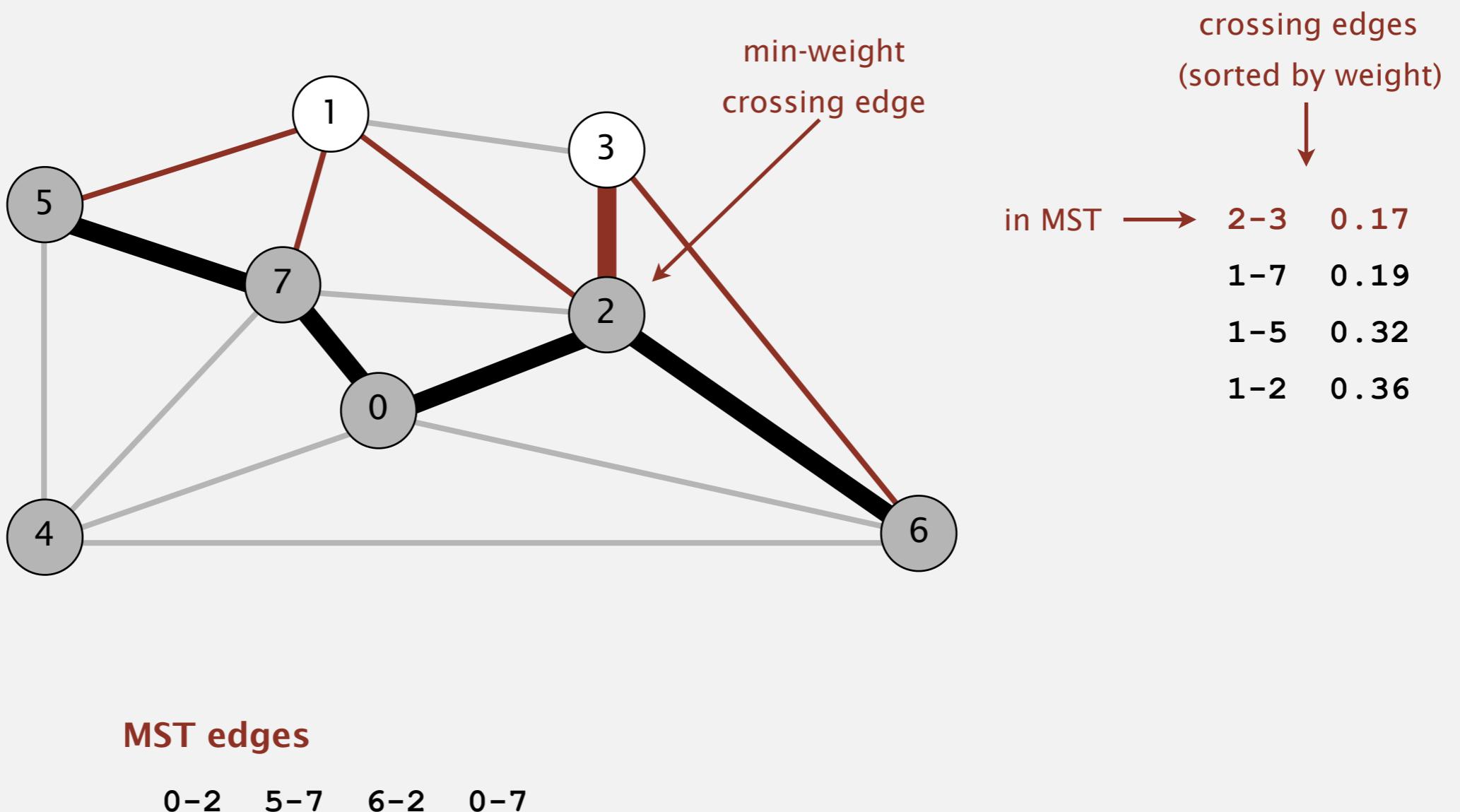


MST edges

0-2 5-7 6-2 0-7

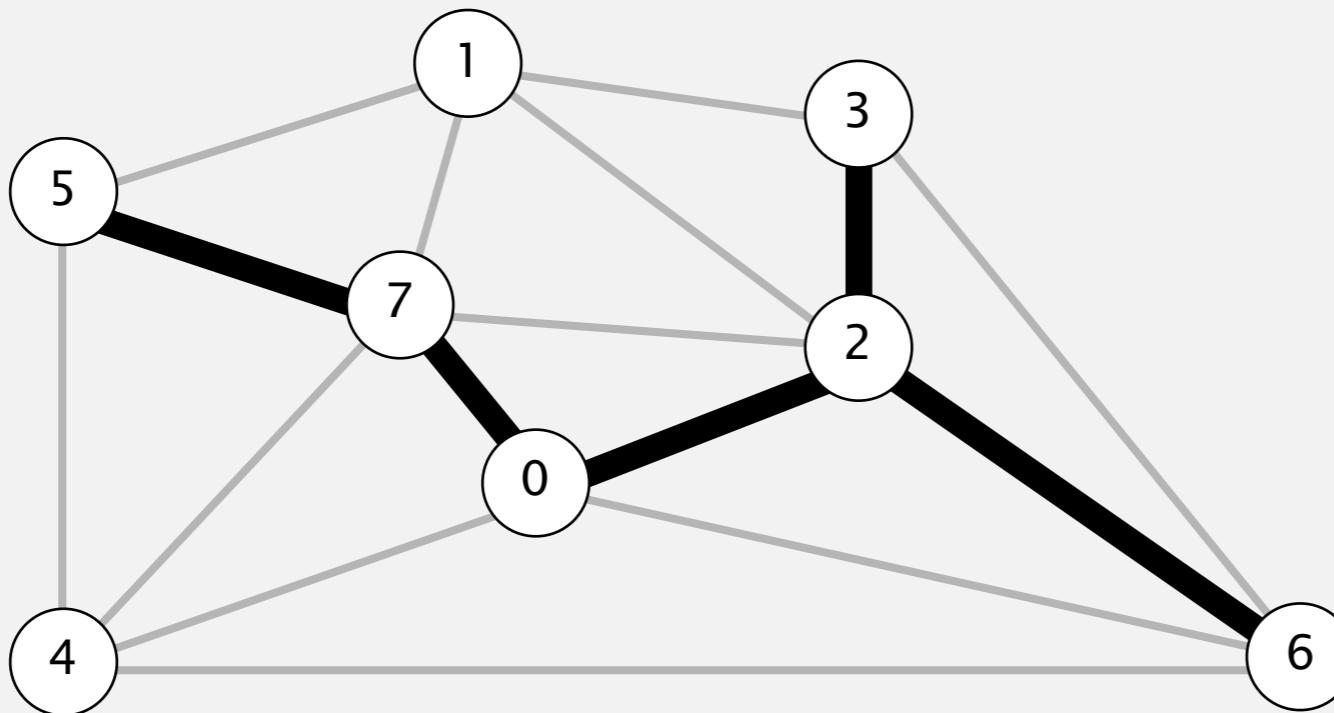
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

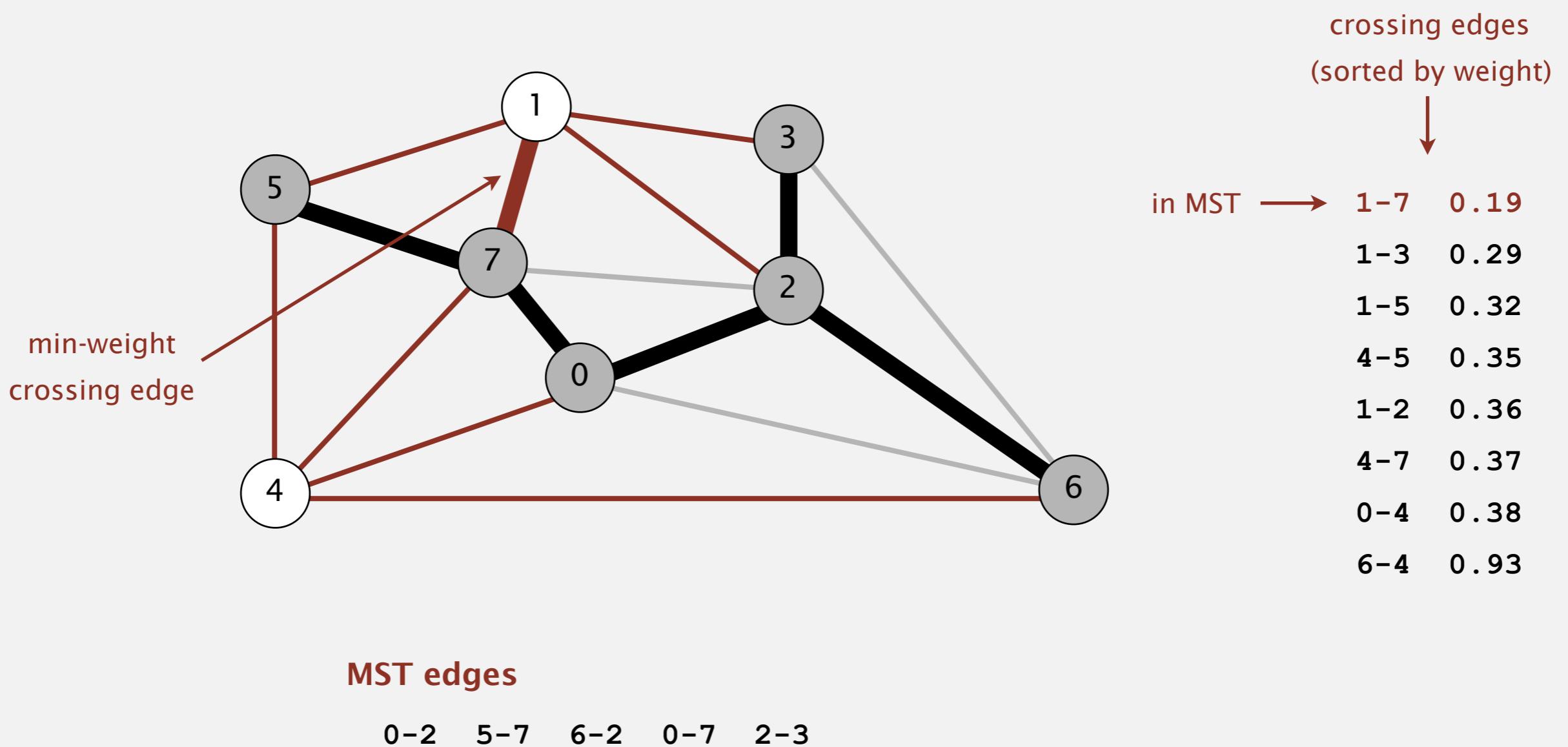


MST edges

0-2 5-7 6-2 0-7 2-3

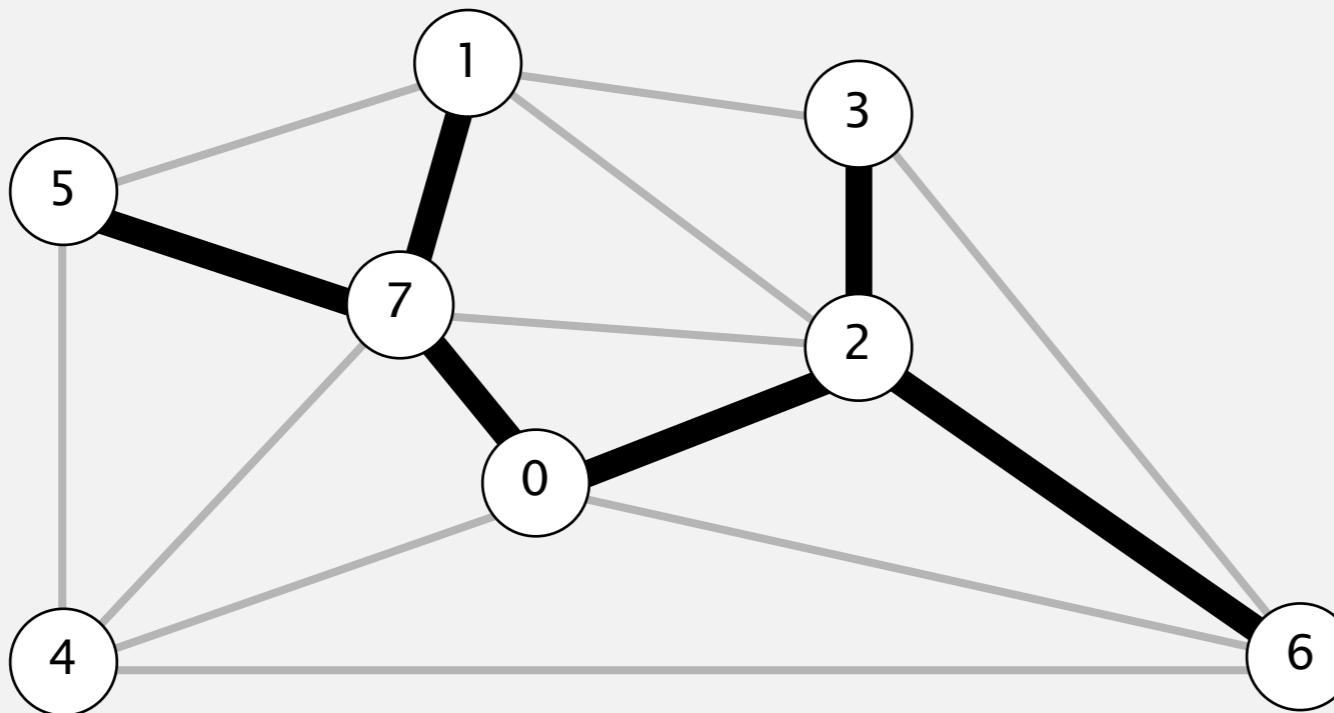
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

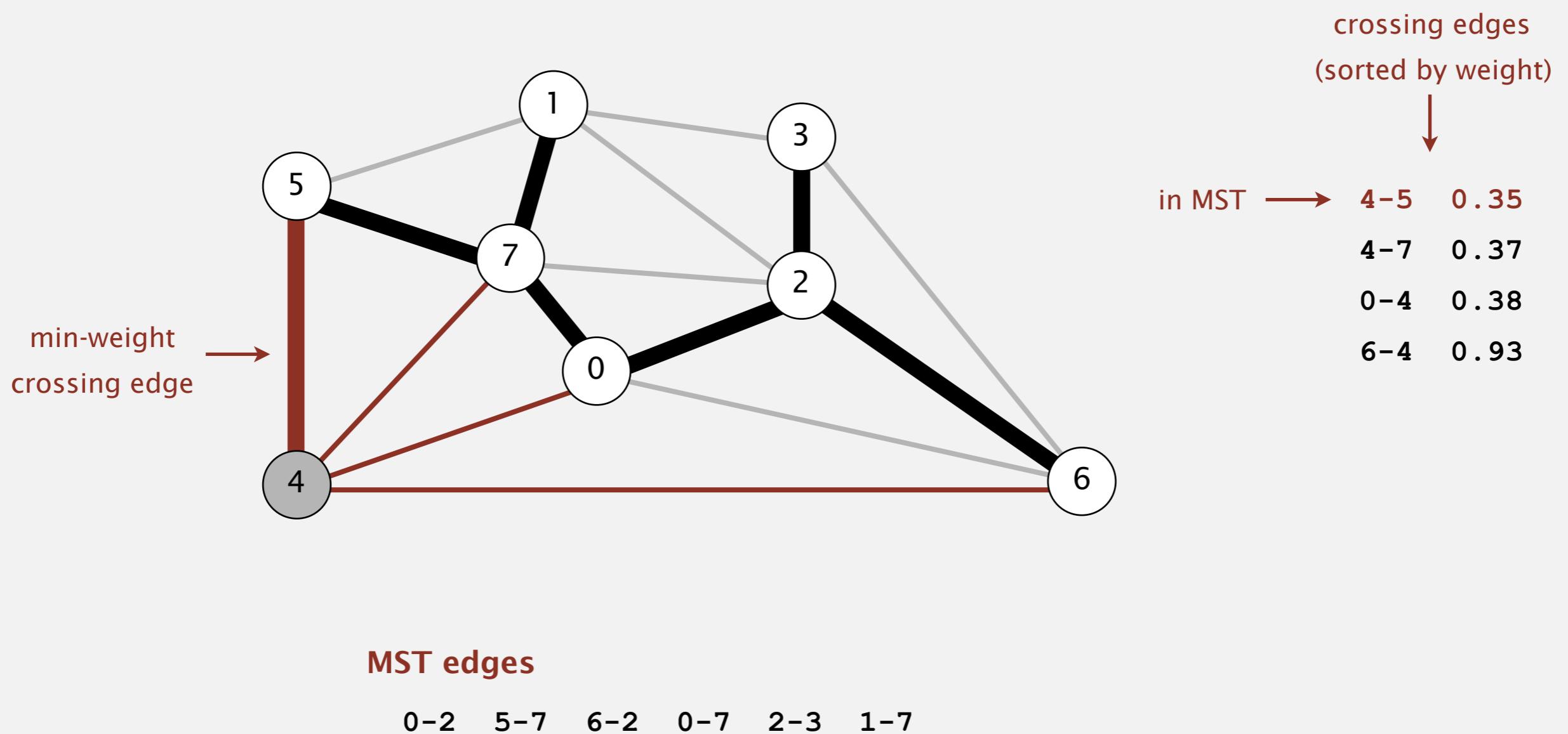


MST edges

0-2 5-7 6-2 0-7 2-3 1-7

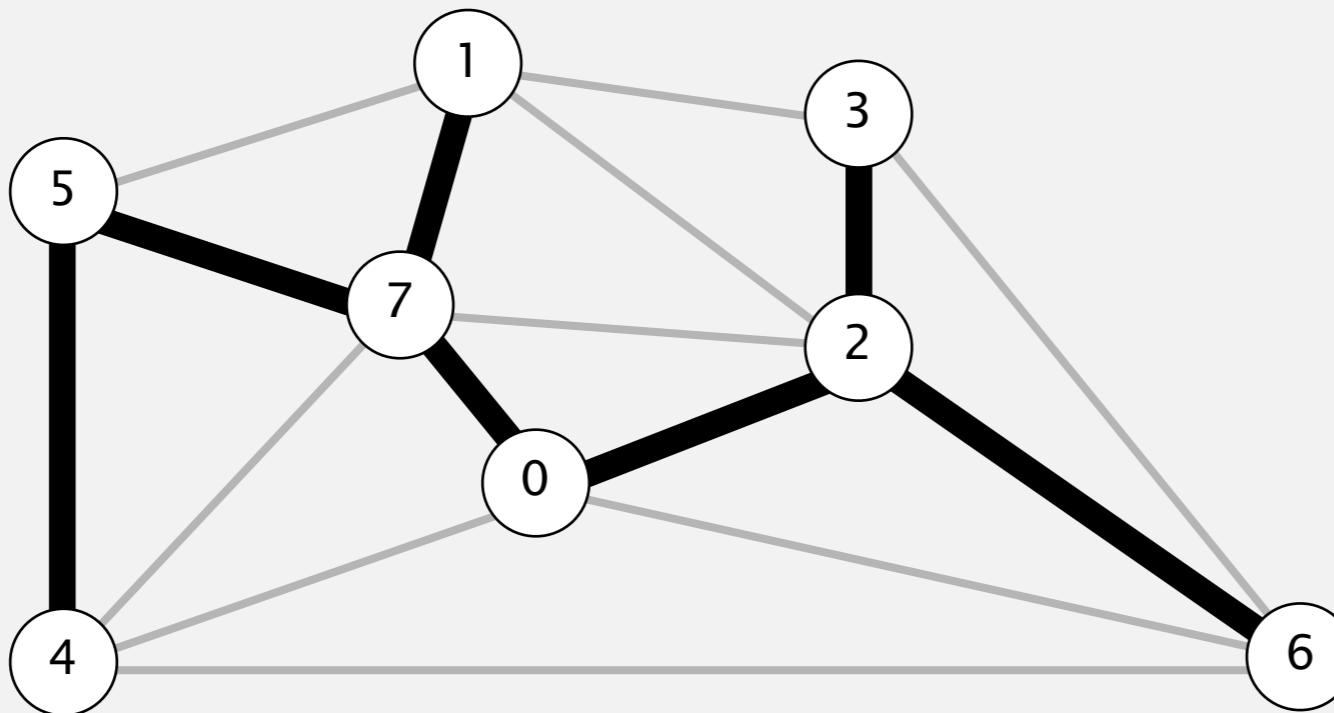
Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



Greedy MST algorithm

- Start with all edges colored gray.
- Find a cut with no black crossing edges, and color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.



MST edges

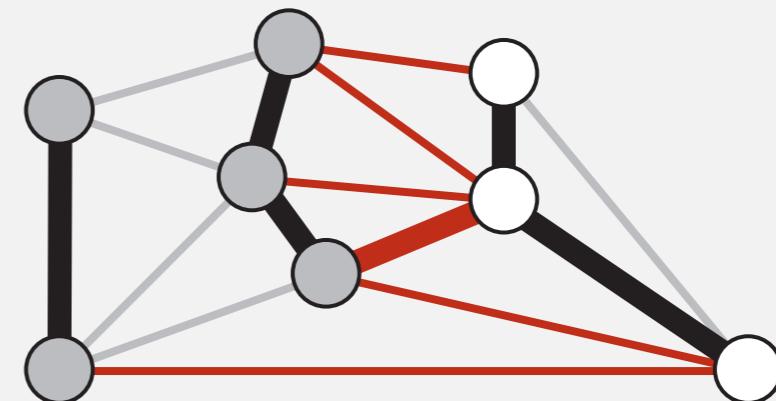
0-2 5-7 6-2 0-7 2-3 1-7 4-5

Greedy MST algorithm: correctness proof

Proposition. The greedy algorithm computes the MST.

Pf.

- Any edge colored black is in the MST (via cut property).
- If fewer than $V - 1$ black edges, there exists a cut with no black crossing edges.
(consider cut whose vertices are one connected component)



a cut with no black crossing edges

Greedy MST algorithm: efficient implementations

Proposition. The greedy algorithm computes the MST:

Efficient implementations. Choose cut? Find min-weight edge?

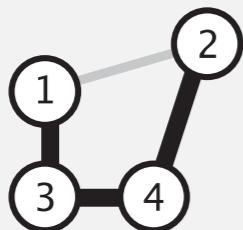
Ex 1. Kruskal's algorithm. [stay tuned]

Ex 2. Prim's algorithm. [stay tuned]

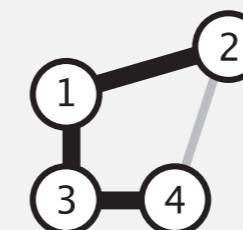
Ex 3. Borüvka's algorithm.

Removing two simplifying assumptions

- Q. What if edge weights are not all distinct?
- A. Greedy MST algorithm still correct if equal weights are present!
(our correctness proof fails, but that can be fixed)

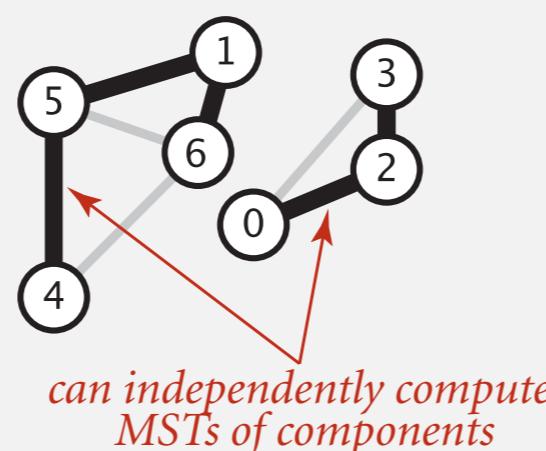


1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50



1	2	1.00
1	3	0.50
2	4	1.00
3	4	0.50

- Q. What if graph is not connected?
- A. Compute minimum spanning forest = MST of each component.



4	5	0.61
4	6	0.62
5	6	0.88
1	5	0.11
2	3	0.35
0	3	0.6
1	6	0.10
0	2	0.22

MINIMUM SPANNING TREES

- ▶ Greedy algorithm
- ▶ Edge-weighted graph API
- ▶ Kruskal's algorithm
- ▶ Prim's algorithm
- ▶ Context

Weighted edge API

Edge abstraction needed for weighted edges.

<code>public class Edge implements Comparable<Edge></code>	
<code> Edge(int v, int w, double weight)</code>	<i>create a weighted edge v-w</i>
<code> int either()</code>	<i>either endpoint</i>
<code> int other(int v)</code>	<i>the endpoint that's not v</i>
<code> int compareTo(Edge that)</code>	<i>compare this edge to that edge</i>
<code> double weight()</code>	<i>the weight</i>
<code> String toString()</code>	<i>string representation</i>



Idiom for processing an edge `e`: `int v = e.either(), w = e.other(v);`

Weighted edge: Java implementation

```
public class Edge implements Comparable<Edge>
{
    private final int v, w;
    private final double weight;

    public Edge(int v, int w, double weight) ← constructor
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int either() ← either endpoint
    {
        return v;
    }

    public int other(int vertex) ← other endpoint
    {
        if (vertex == v) return w;
        else return v;
    }

    public int compareTo(Edge that) ← compare edges by weight
    {
        if (this.weight < that.weight) return -1;
        else if (this.weight > that.weight) return +1;
        else
            return 0;
    }
}
```

Edge-weighted graph API

```
public class EdgeWeightedGraph
```

```
    EdgeWeightedGraph(int V)
```

create an empty graph with V vertices

```
    EdgeWeightedGraph(In in)
```

create a graph from input stream

```
    void addEdge(Edge e)
```

add weighted edge e to this graph

```
    Iterable<Edge> adj(int v)
```

edges incident to v

```
    Iterable<Edge> edges()
```

all edges in this graph

```
    int V()
```

number of vertices

```
    int E()
```

number of edges

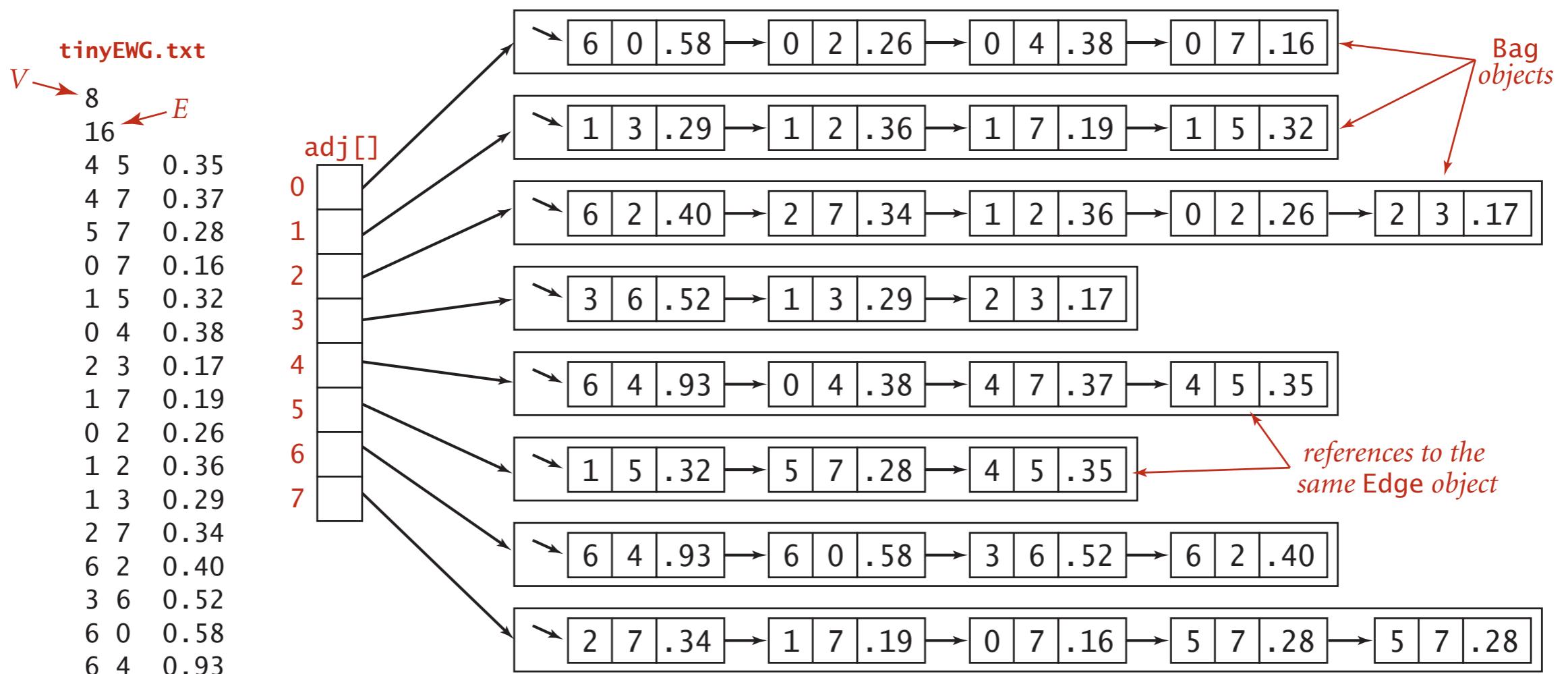
```
    String toString()
```

string representation

Conventions. Allow self-loops and parallel edges.

Edge-weighted graph: adjacency-lists representation

Maintain vertex-indexed array of Edge lists.



Edge-weighted graph: adjacency-lists implementation

```
public class EdgeWeightedGraph
{
    private final int V;
    private final Bag<Edge>[] adj;

    public EdgeWeightedGraph(int V)
    {
        this.V = V;
        adj = (Bag<Edge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Edge>();
    }

    public void addEdge(Edge e)
    {
        int v = e.either(), w = e.other(v);
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<Edge> adj(int v)
    {
        return adj[v];
    }
}
```

same as Graph, but adjacency lists of Edges instead of integers

constructor

add edge to both adjacency lists

Minimum spanning tree API

Q. How to represent the MST?

```
public class MST
```

```
MST(EdgeWeightedGraph G)
```

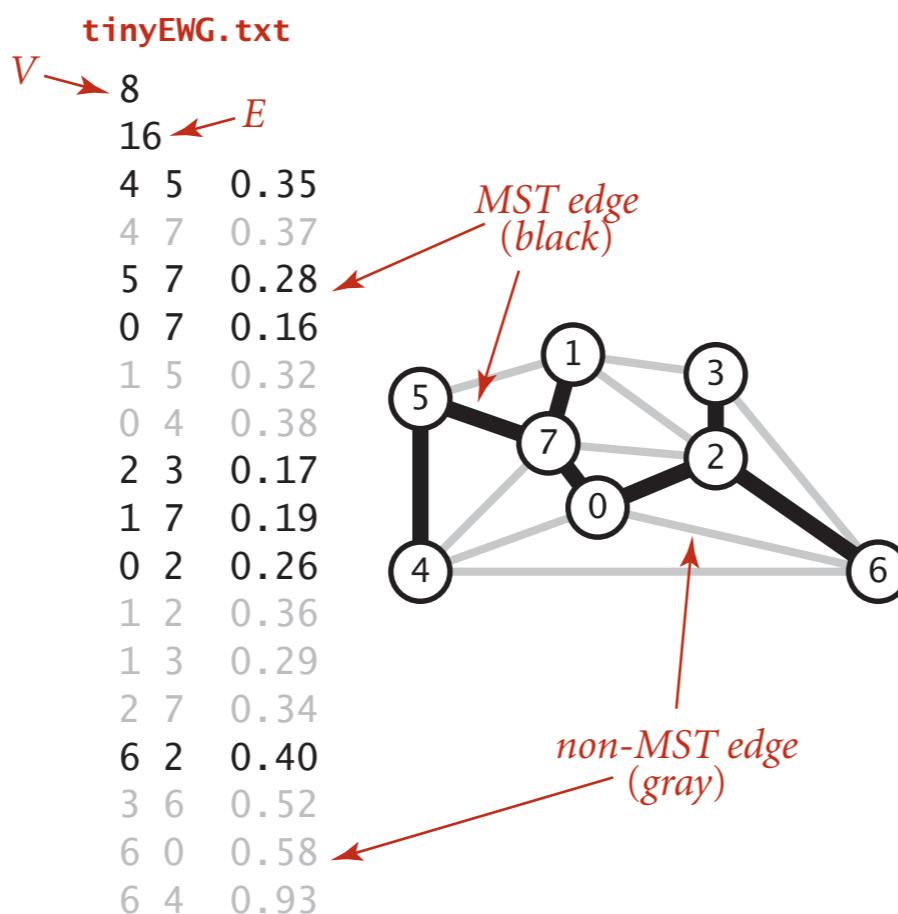
constructor

```
Iterable<Edge> edges()
```

edges in MST

```
double weight()
```

weight of MST



```
% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

Minimum spanning tree API

Q. How to represent the MST?

```
public class MST
```

```
MST(EdgeWeightedGraph G)
```

constructor

```
Iterable<Edge> edges()
```

edges in MST

```
double weight()
```

weight of MST

```
public static void main(String[] args)
{
    In in = new In(args[0]);
    EdgeWeightedGraph G = new EdgeWeightedGraph(in);
    MST mst = new MST(G);
    for (Edge e : mst.edges())
        StdOut.println(e);
    StdOut.printf("%.2f\n", mst.weight());
}
```

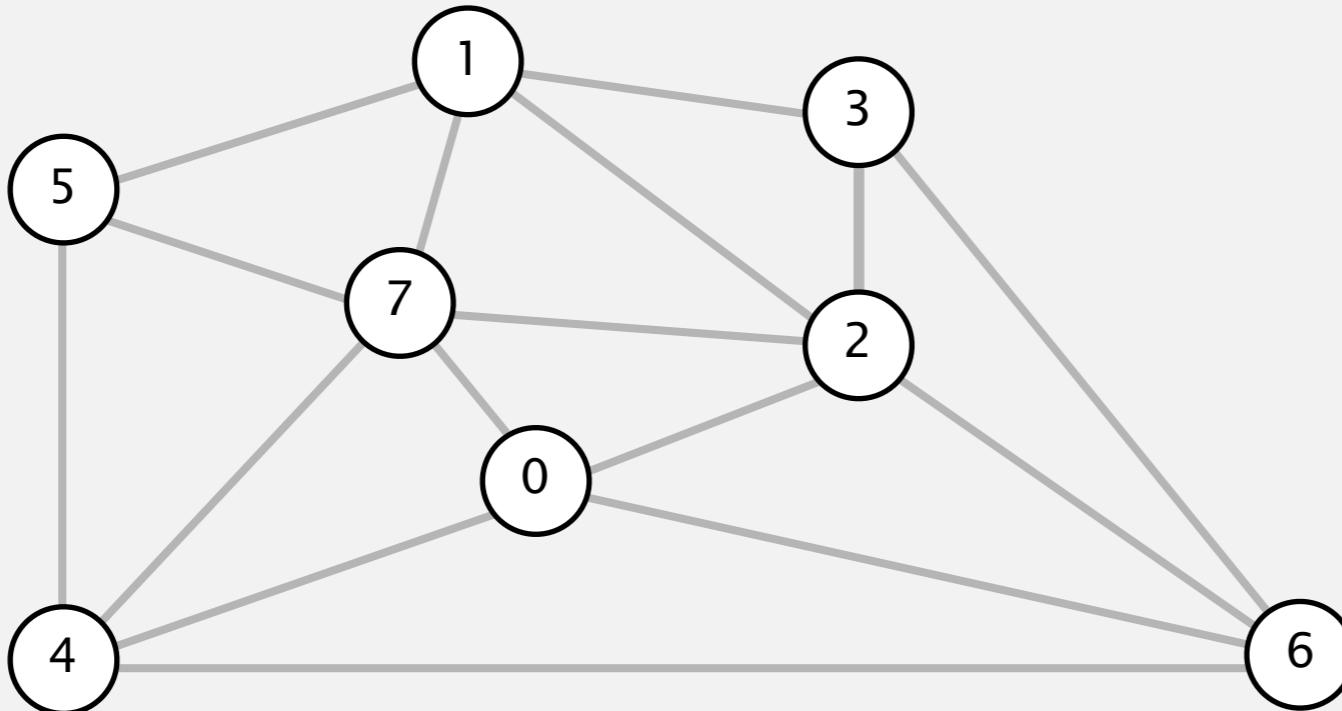
```
% java MST tinyEWG.txt
0-7 0.16
1-7 0.19
0-2 0.26
2-3 0.17
5-7 0.28
4-5 0.35
6-2 0.40
1.81
```

MINIMUM SPANNING TREES

- ▶ Greedy algorithm
- ▶ Edge-weighted graph API
- ▶ Kruskal's algorithm
- ▶ Prim's algorithm
- ▶ Context

Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.

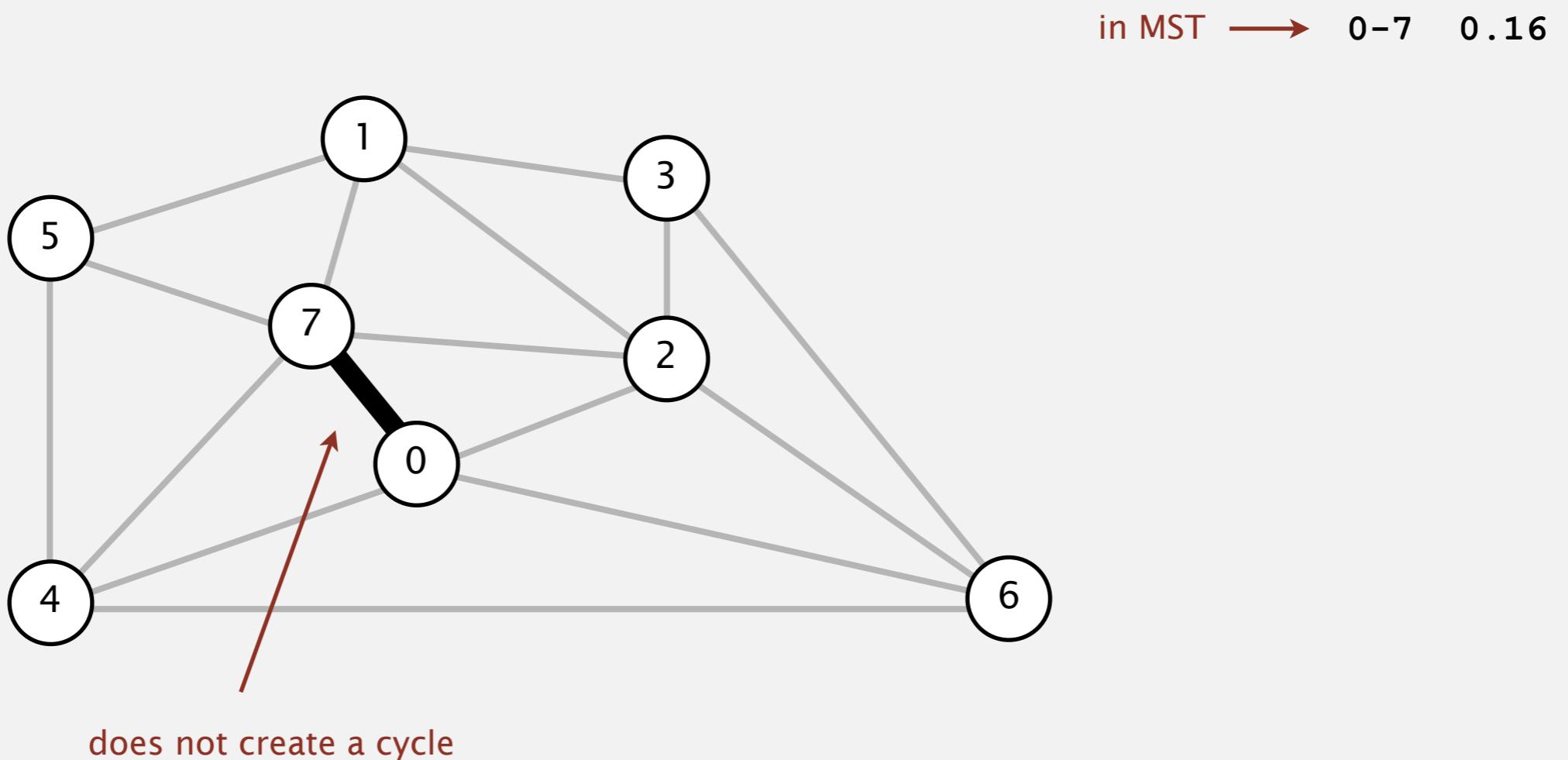


an edge-weighted graph

graph edges sorted by weight	
0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

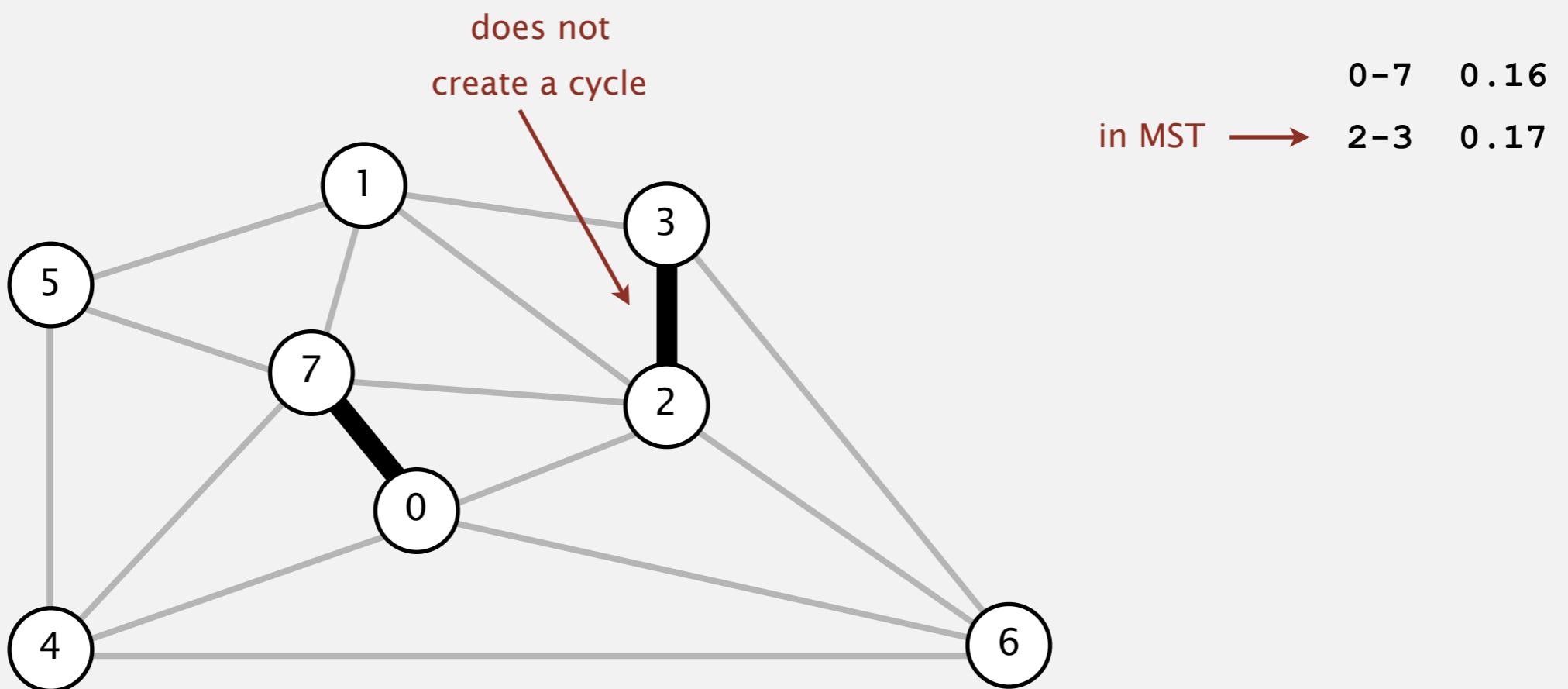
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



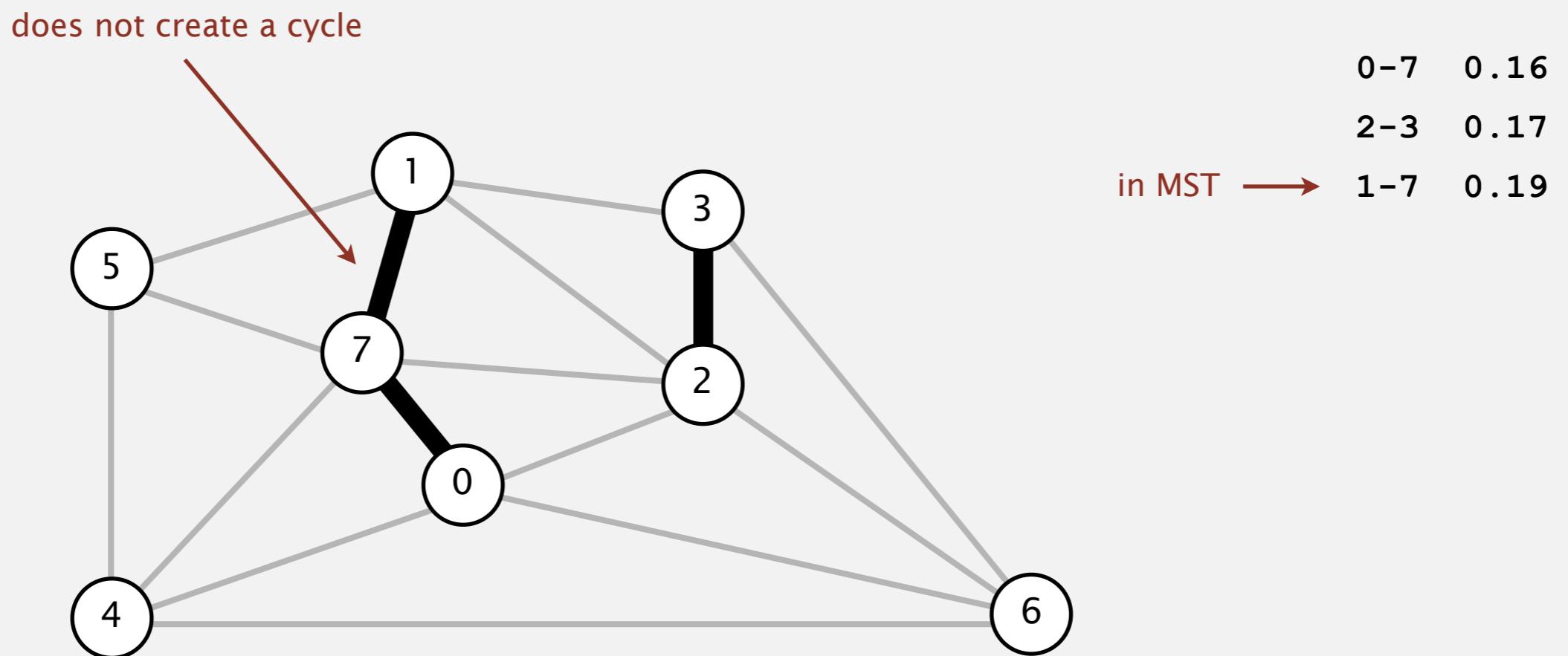
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



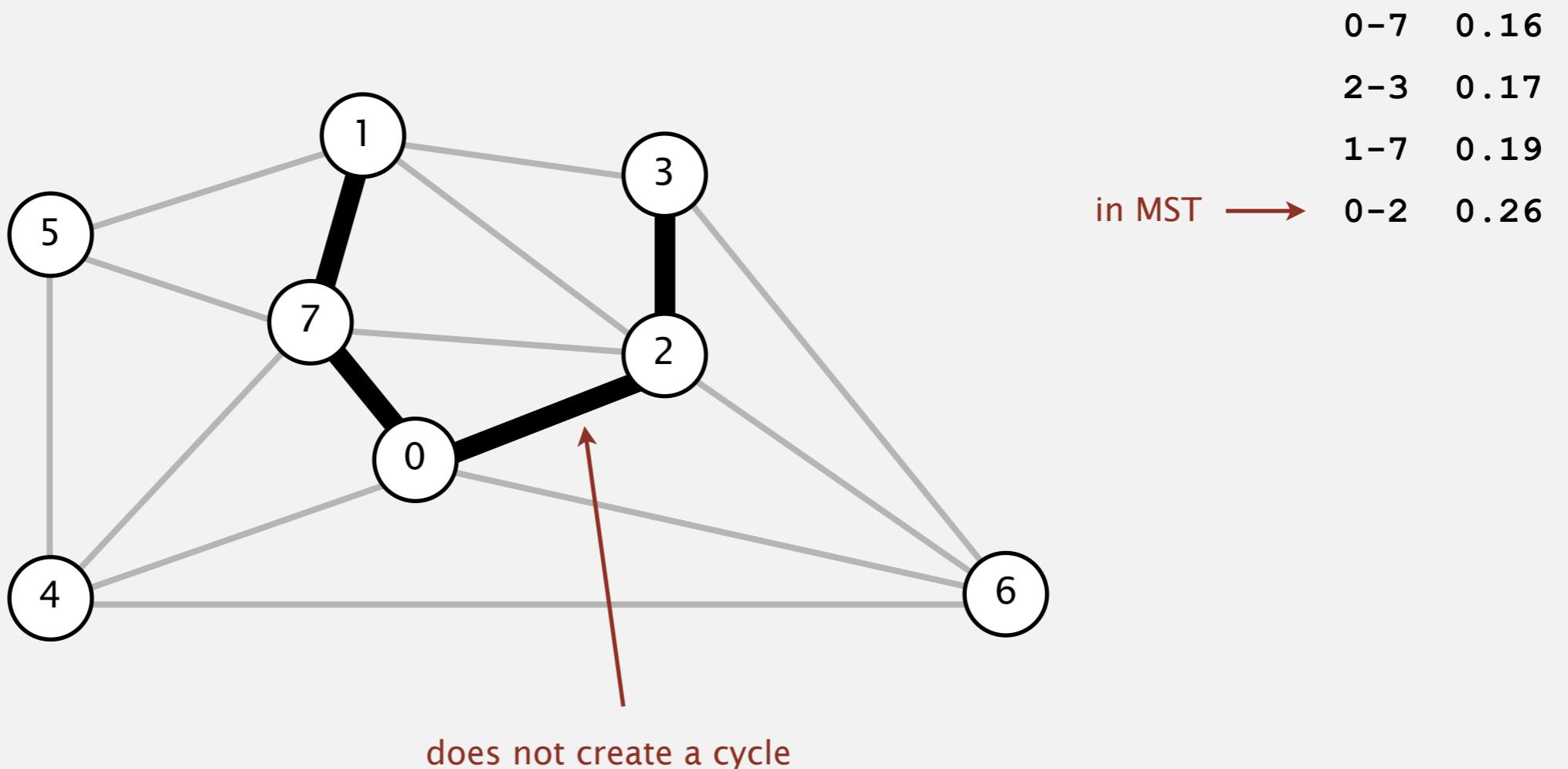
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



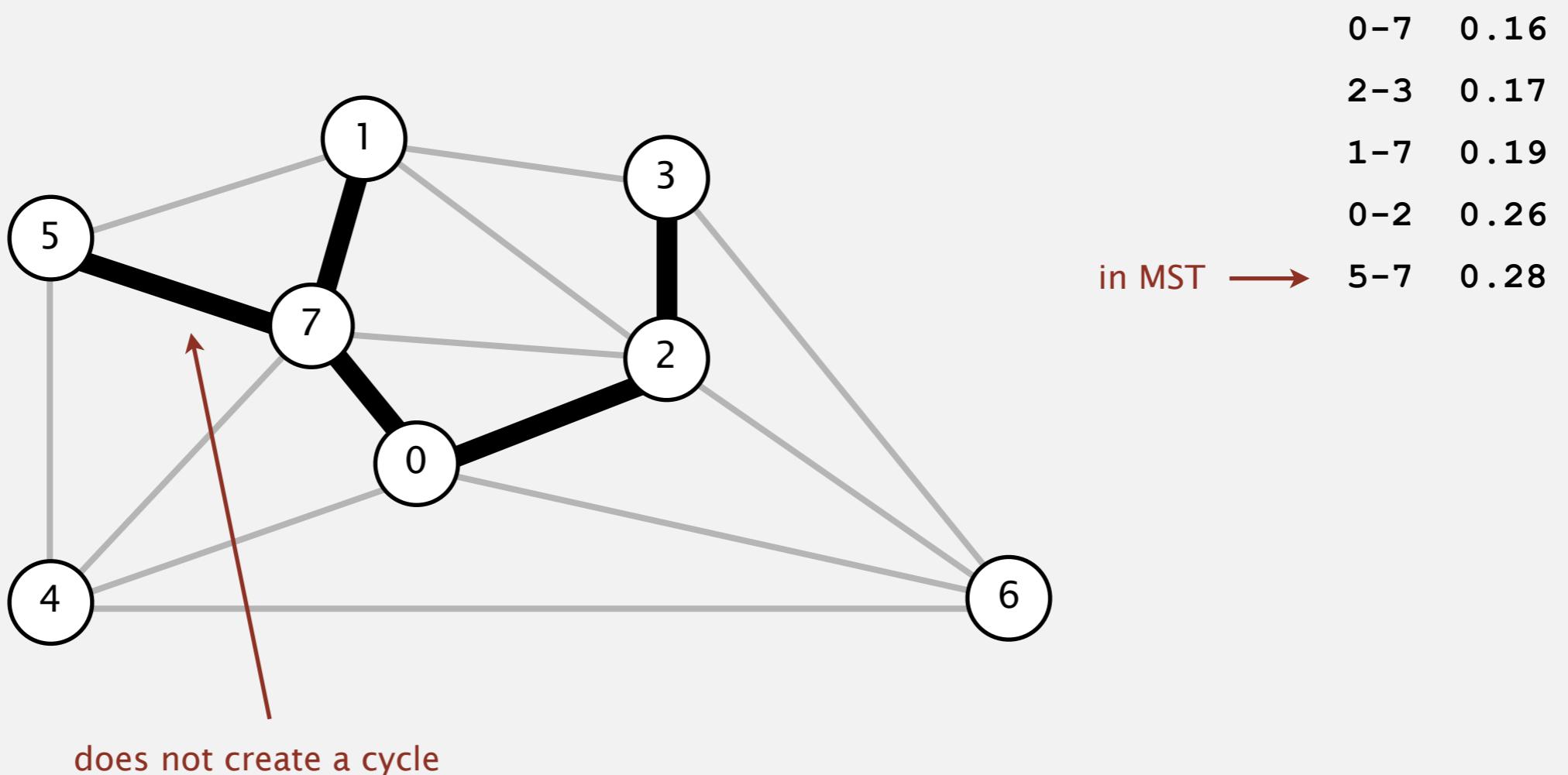
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



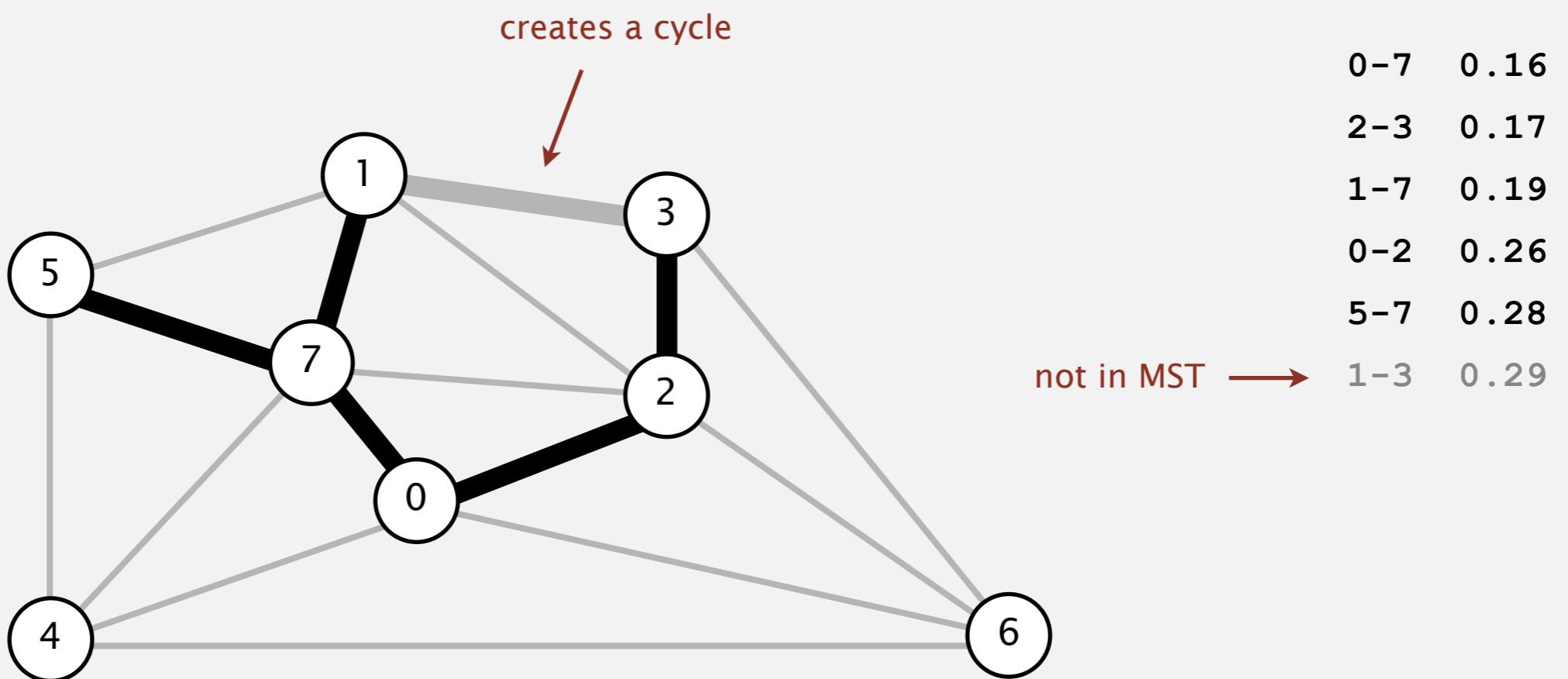
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



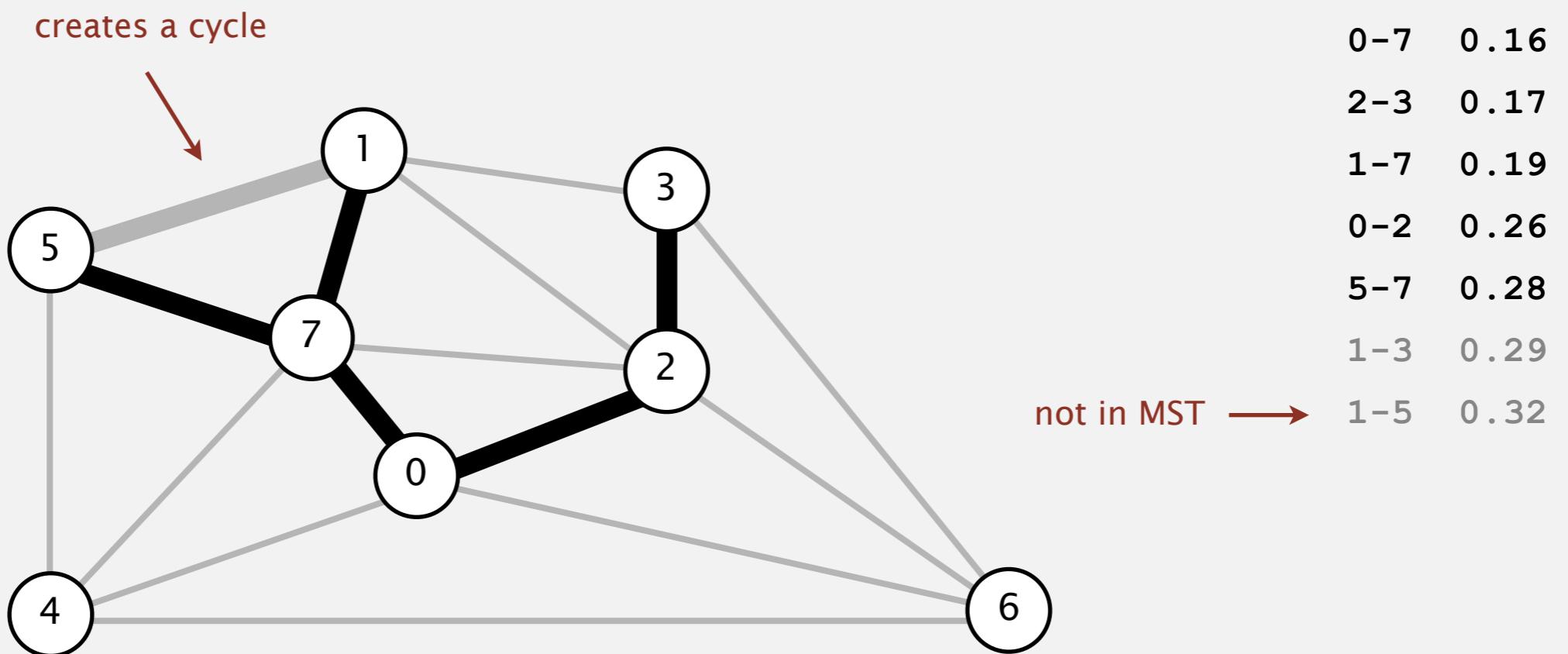
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



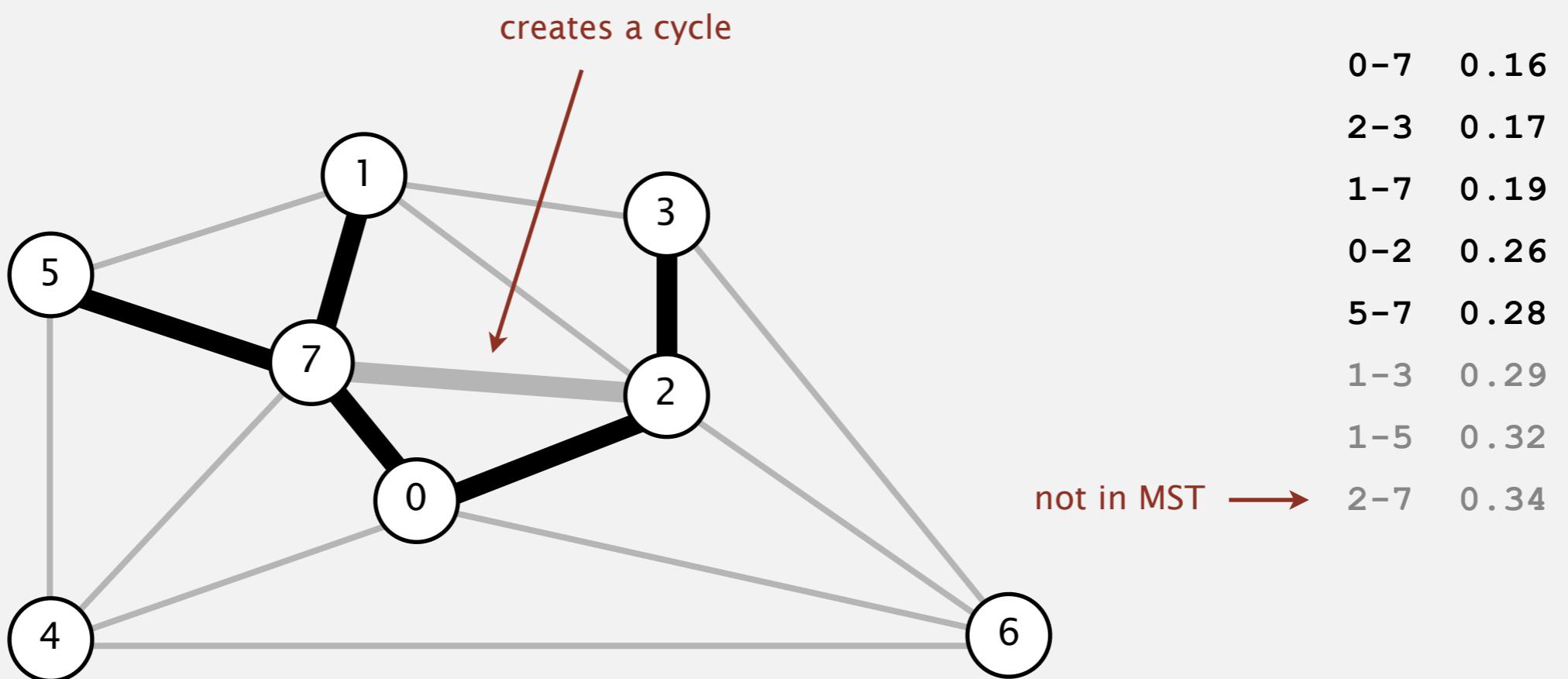
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



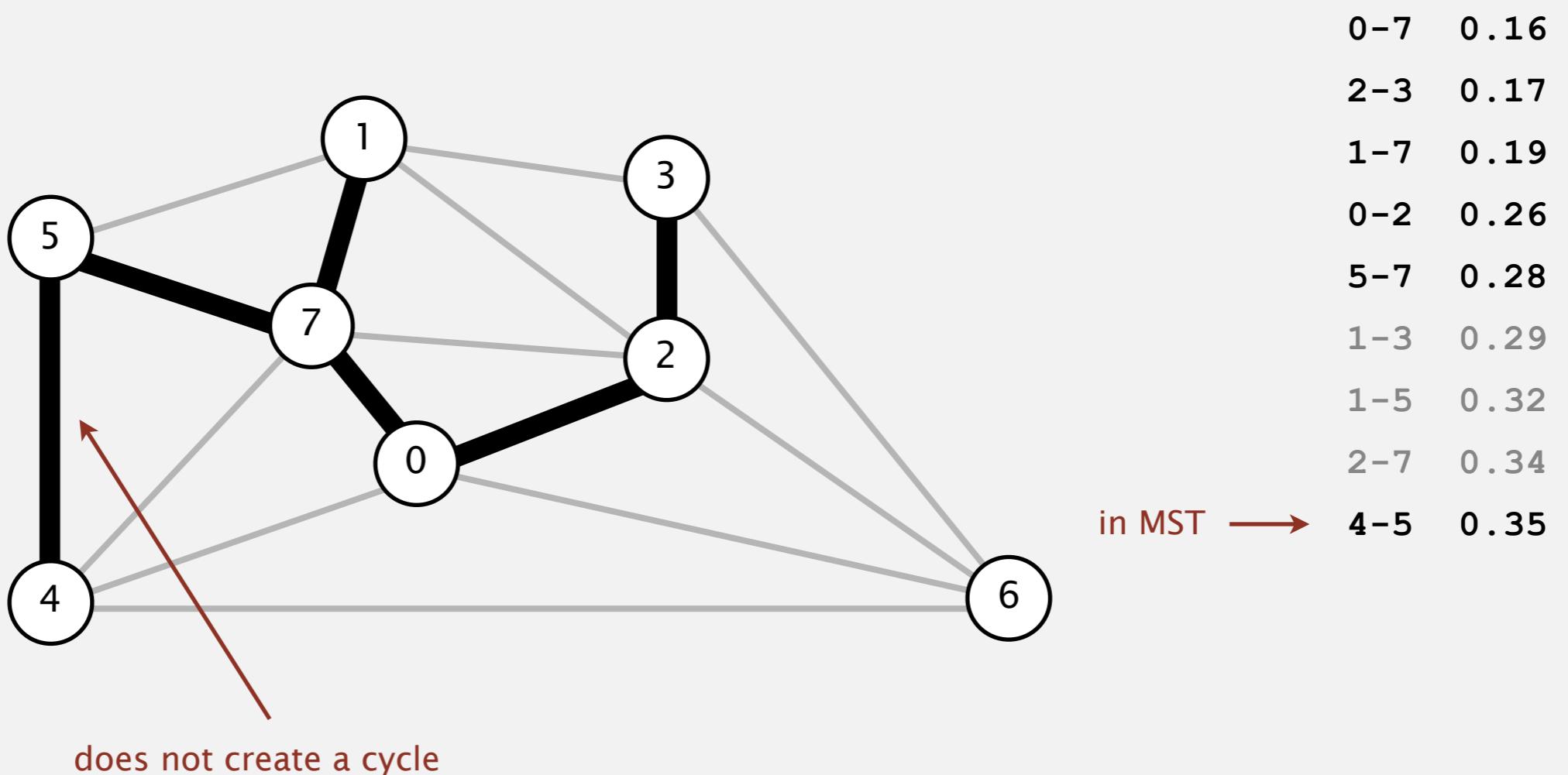
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



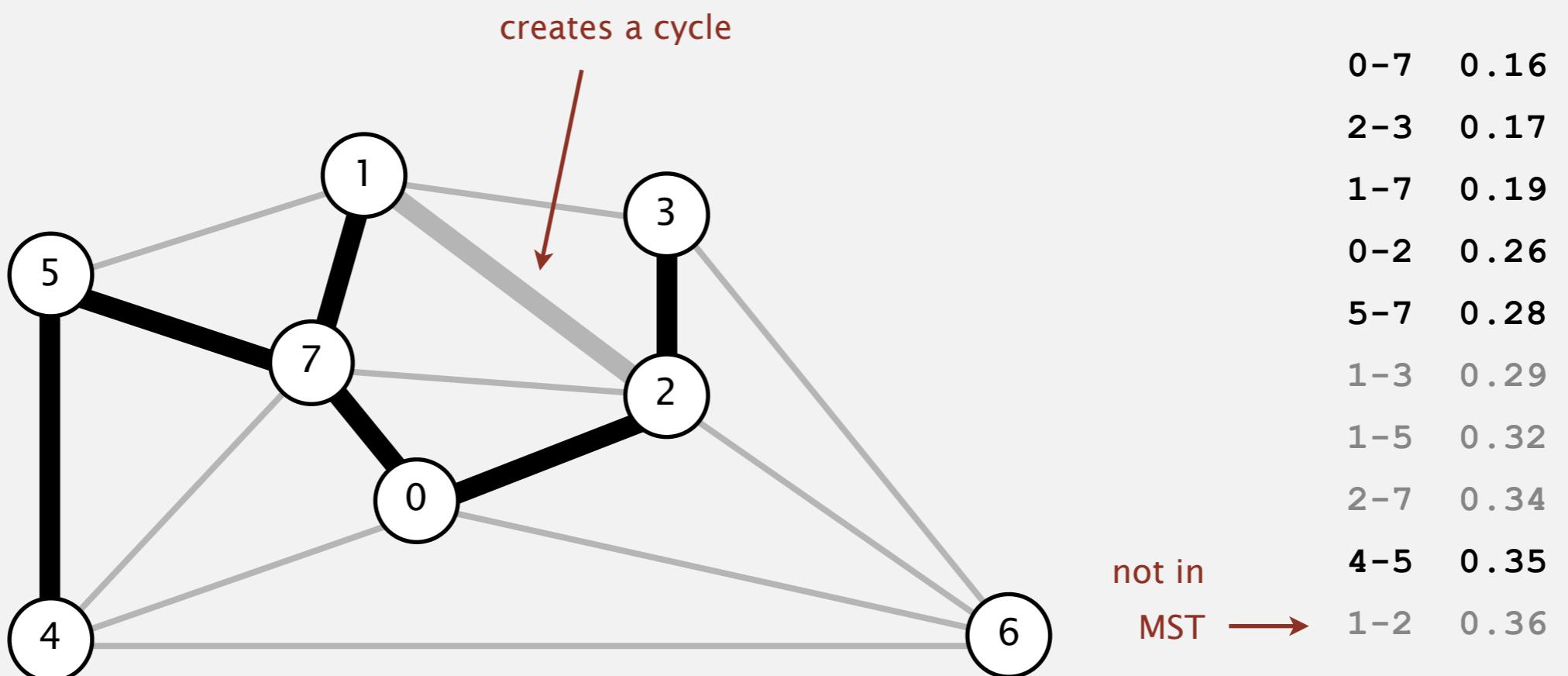
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



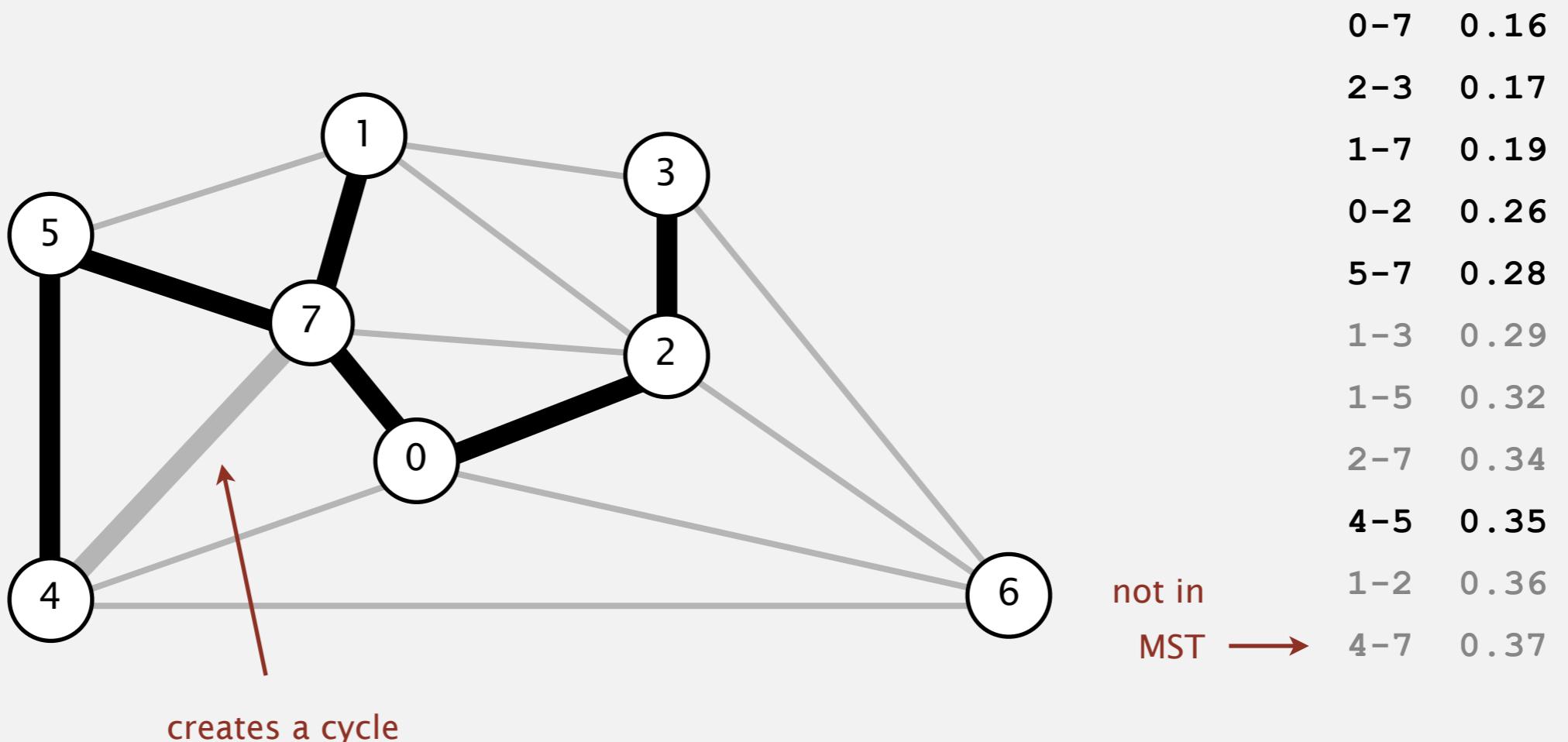
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



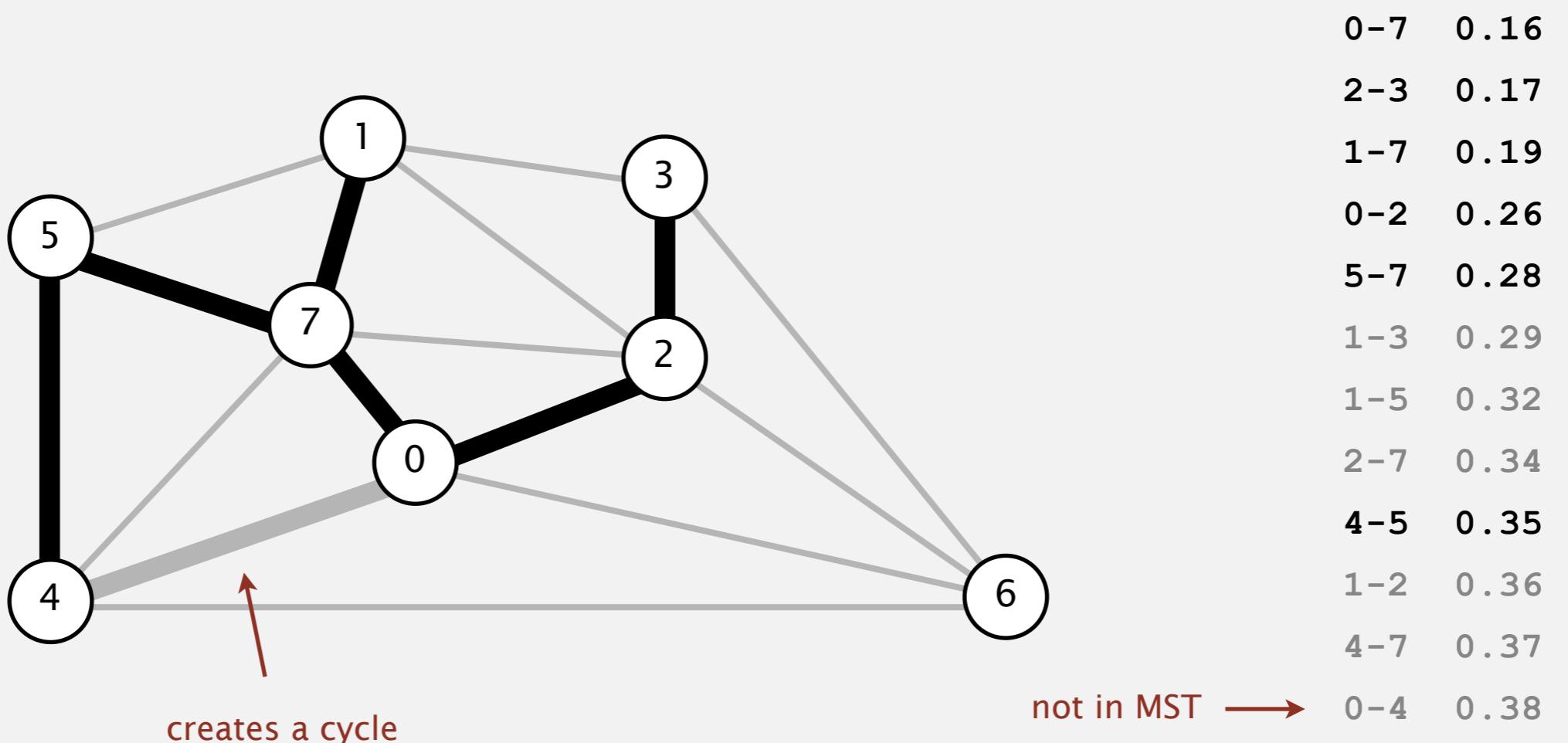
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



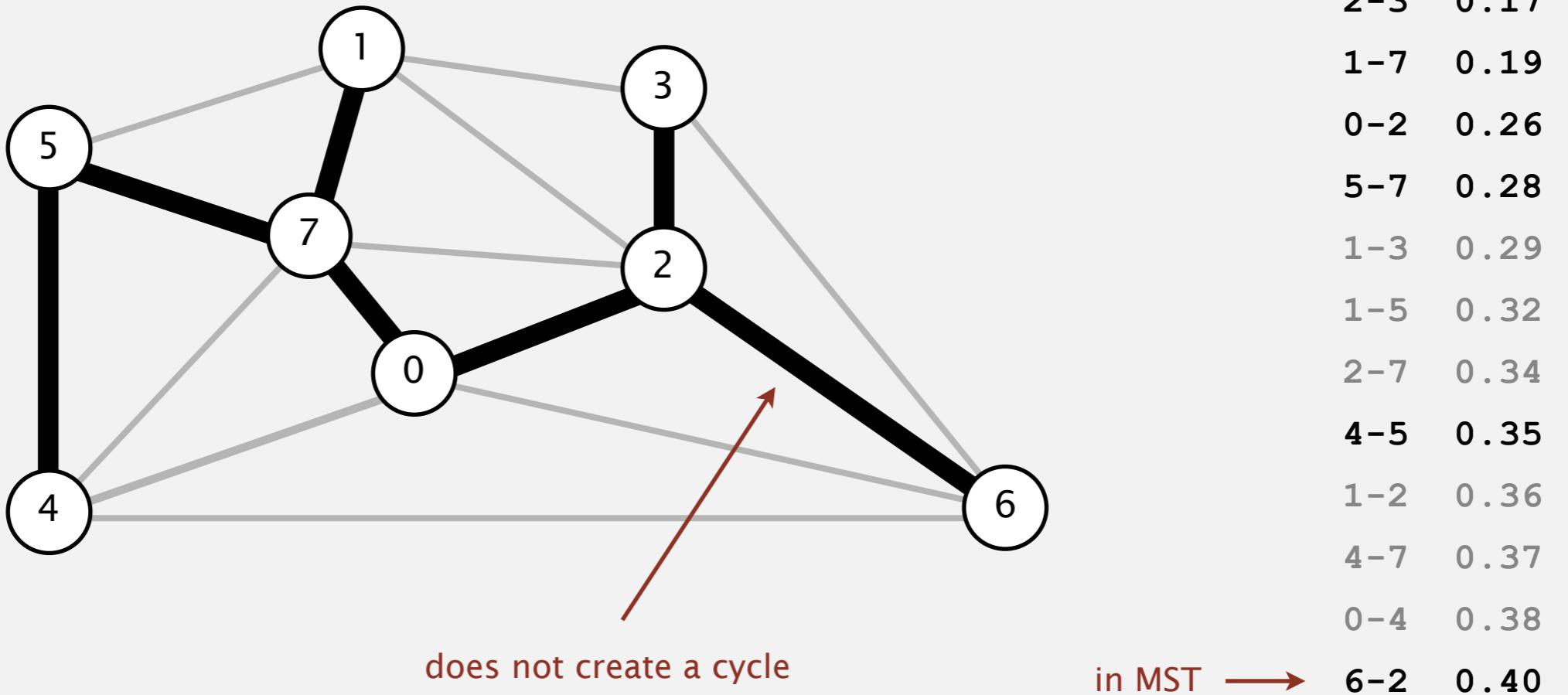
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



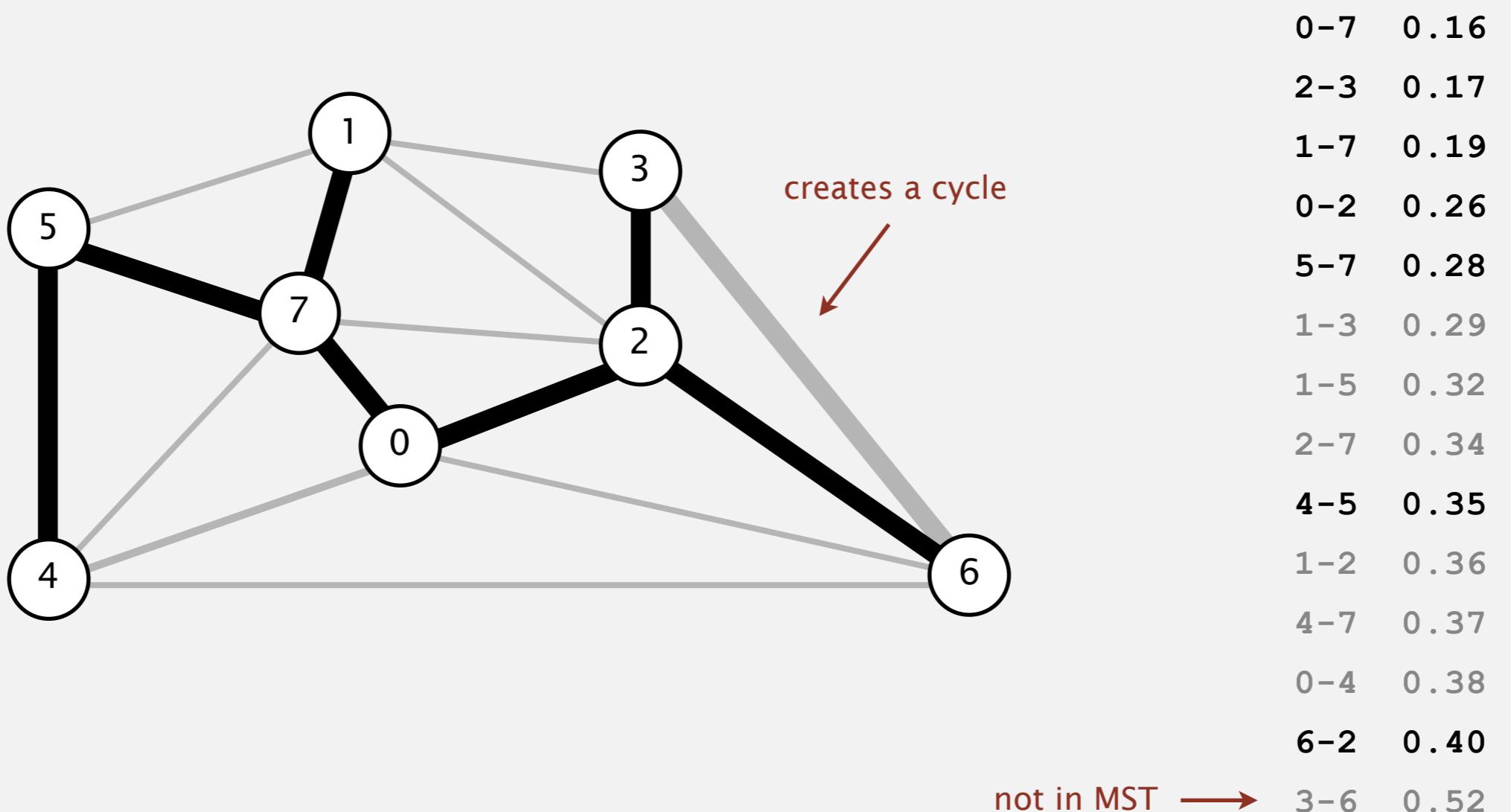
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



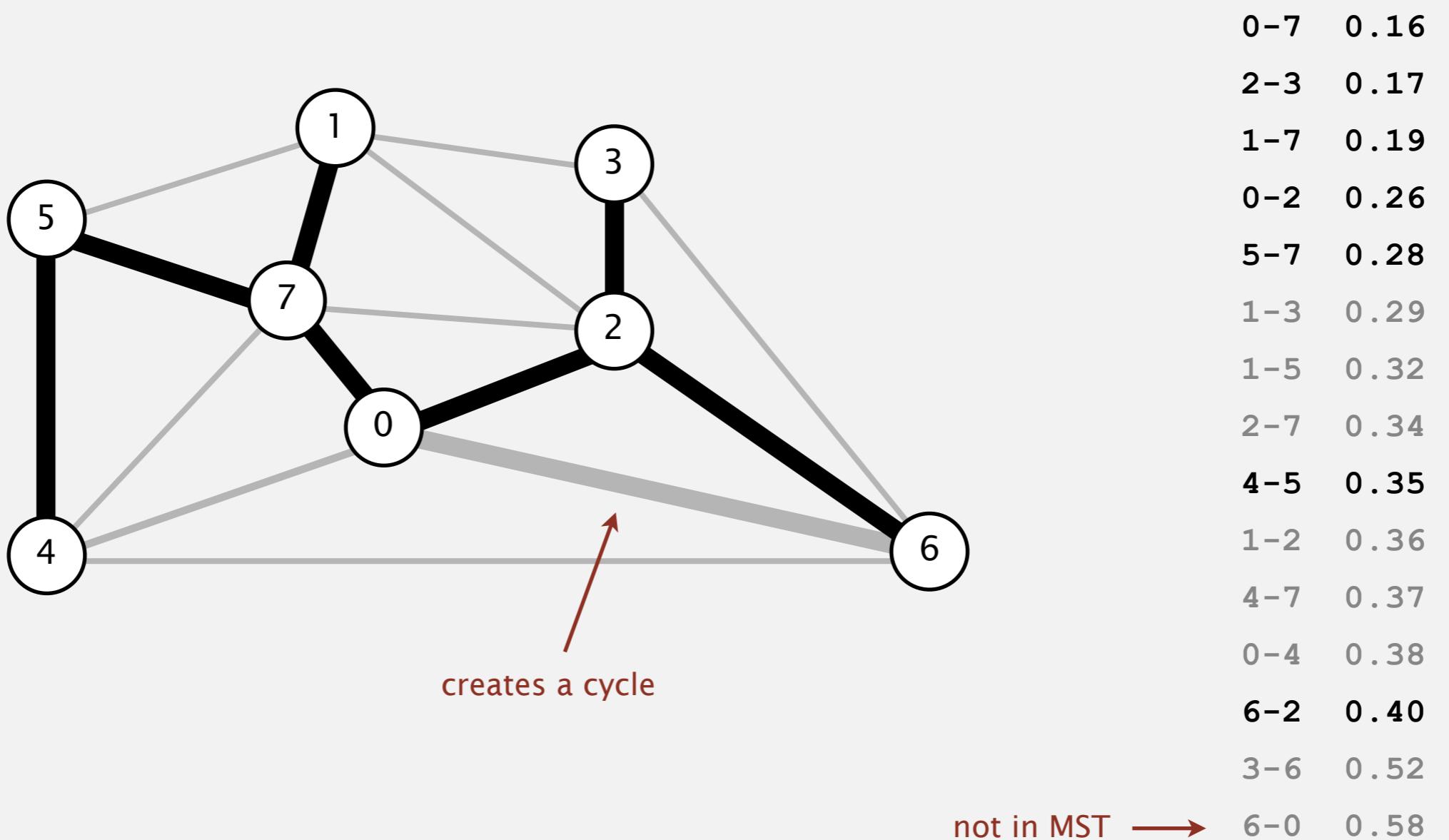
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



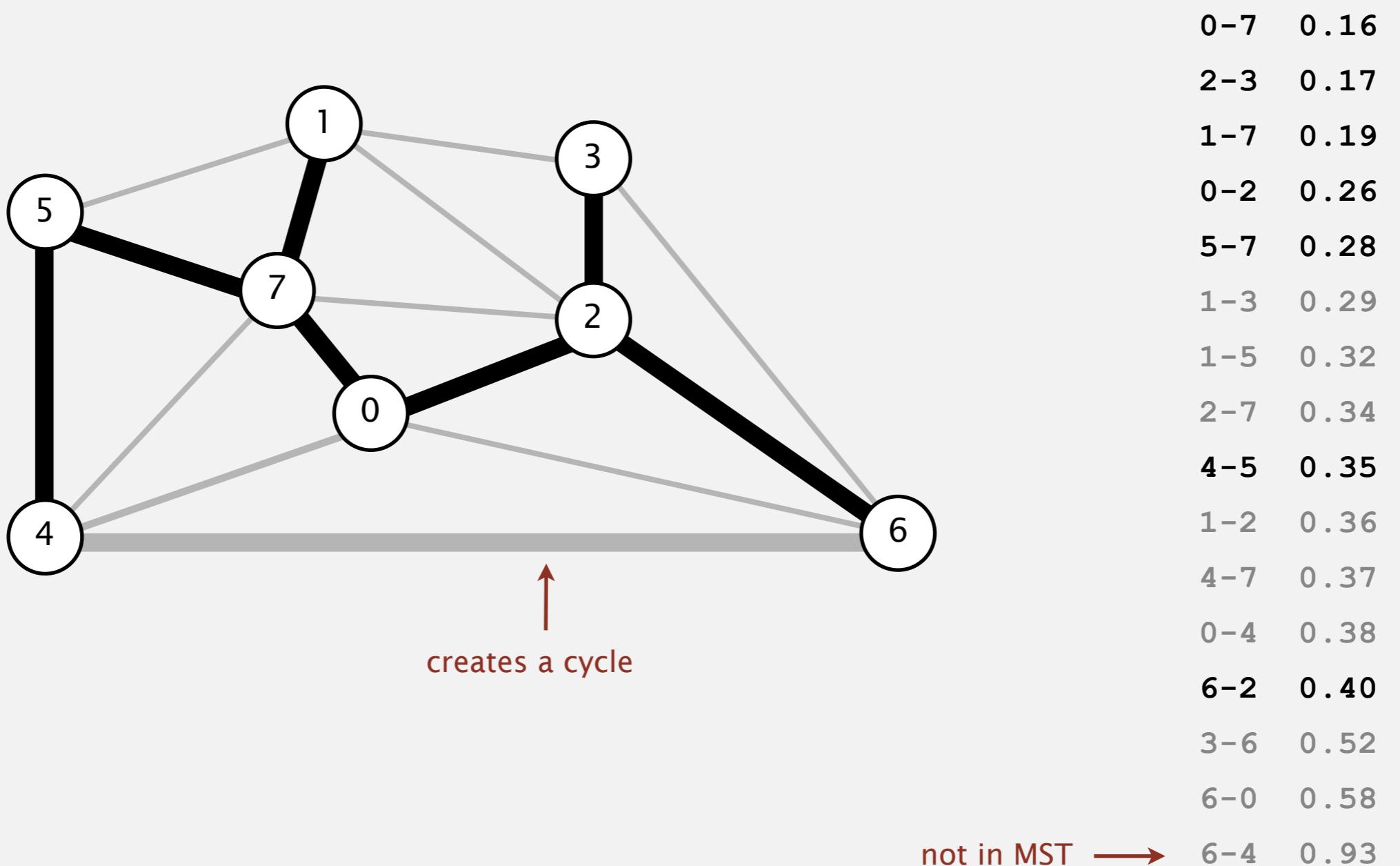
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



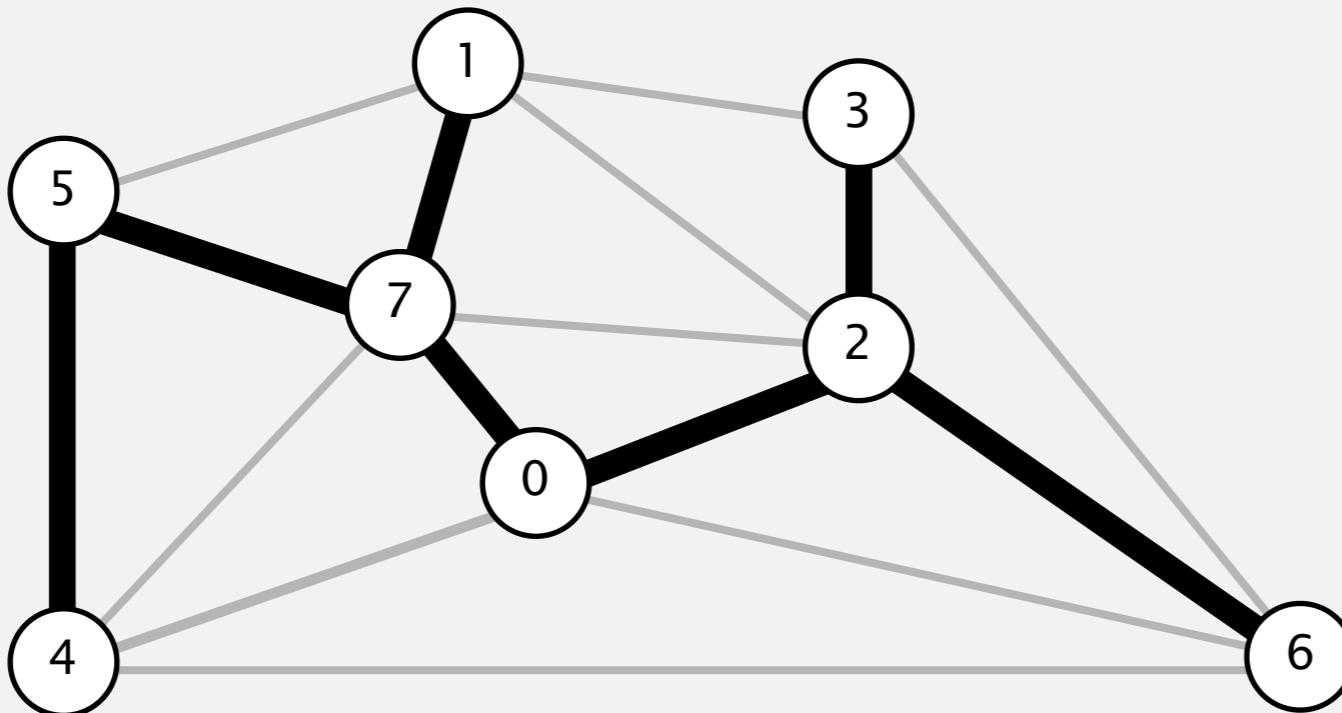
Kruskal's algorithm

- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



Kruskal's algorithm

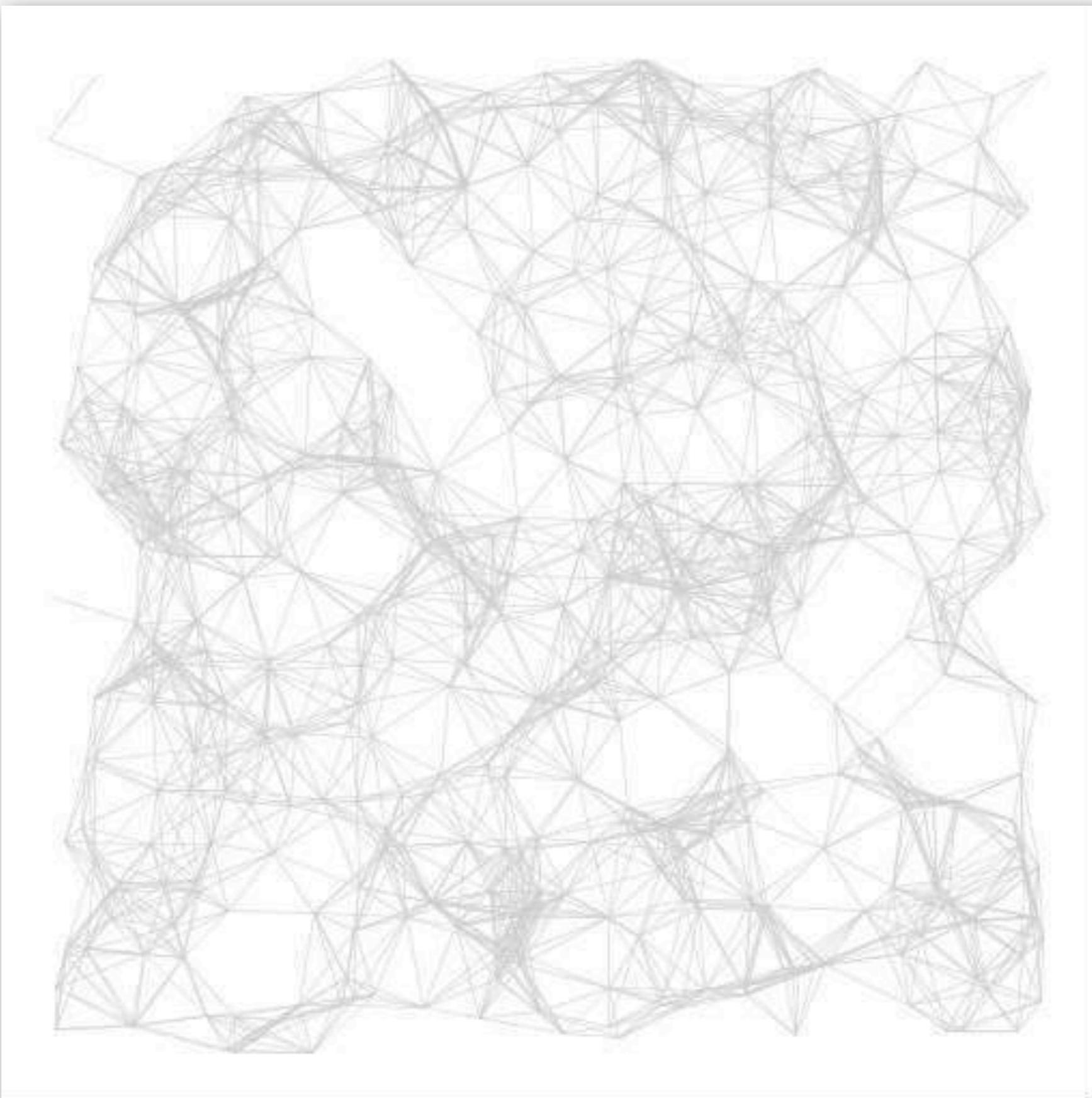
- Consider edges in ascending order of weight.
- Add next edge to tree T unless doing so would create a cycle.



a minimum spanning tree

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Kruskal's algorithm: visualization

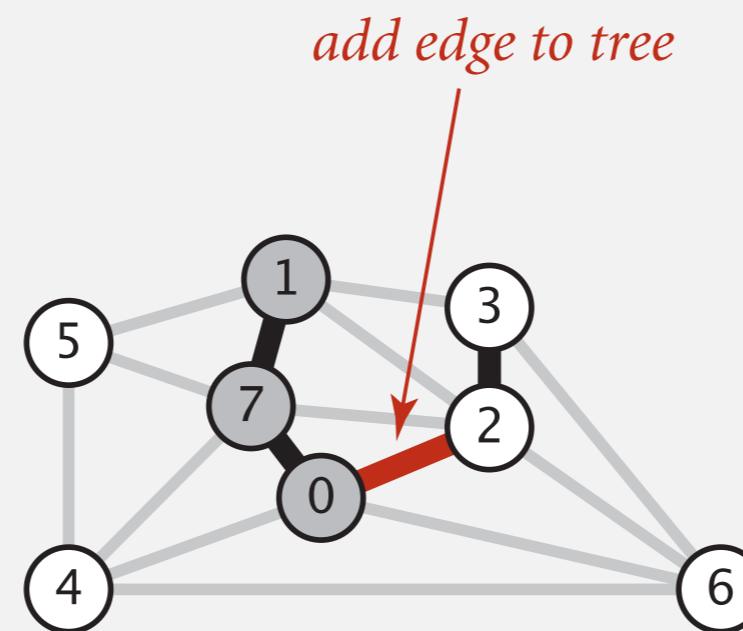


Kruskal's algorithm: correctness proof

Proposition. [Kruskal 1956] Kruskal's algorithm computes the MST.

Pf. Kruskal's algorithm is a special case of the greedy MST algorithm.

- Suppose Kruskal's algorithm colors the edge $e = v-w$ black.
- Cut = set of vertices connected to v in tree T .
- No crossing edge is black.
- No crossing edge has lower weight. Why?

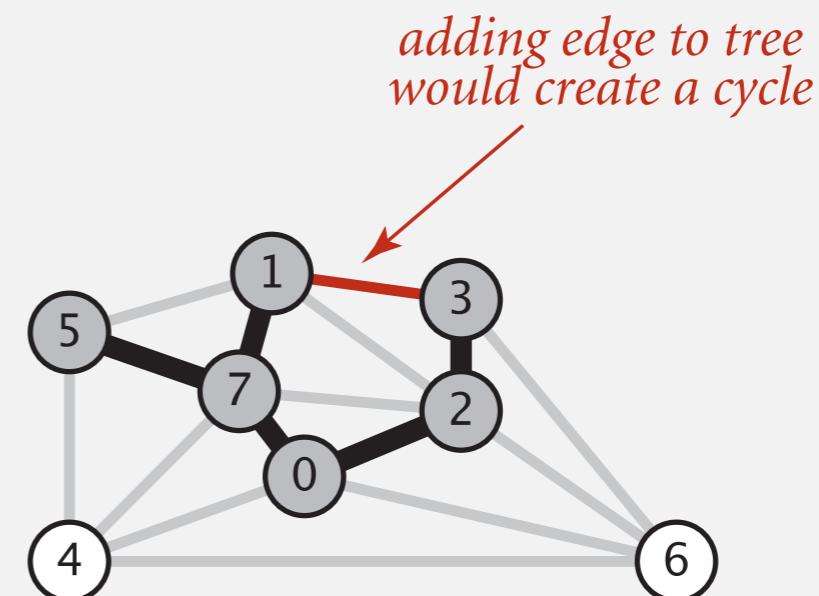
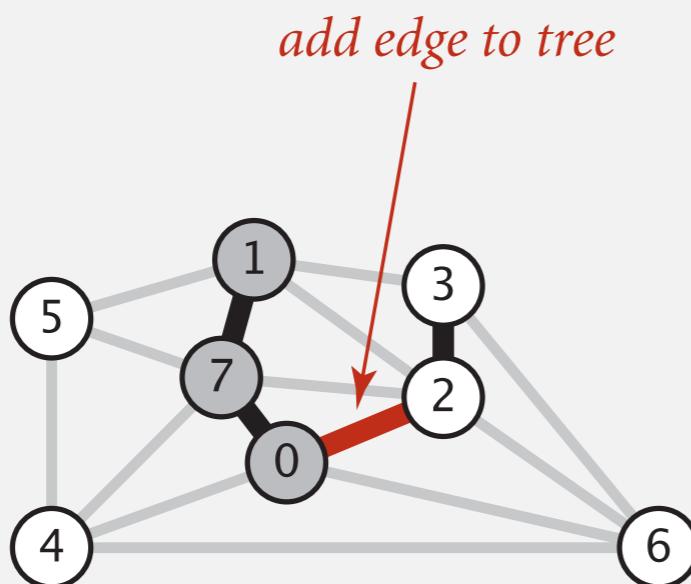


Kruskal's algorithm: implementation challenge

Challenge. Would adding edge $v-w$ to tree T create a cycle? If not, add it.

How difficult?

- $E + V$
- V ← run DFS from v , check if w is reachable
(T has at most $V - 1$ edges)
- $\log V$
- $\log^* V$ ← use the union-find data structure !
(\log^* function: number of times needed to take the \lg of a number until reaching 1)
- 1

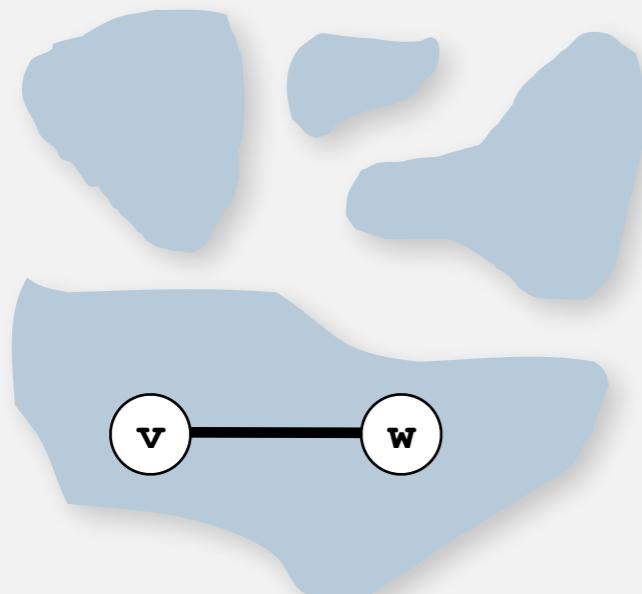


Kruskal's algorithm: implementation challenge

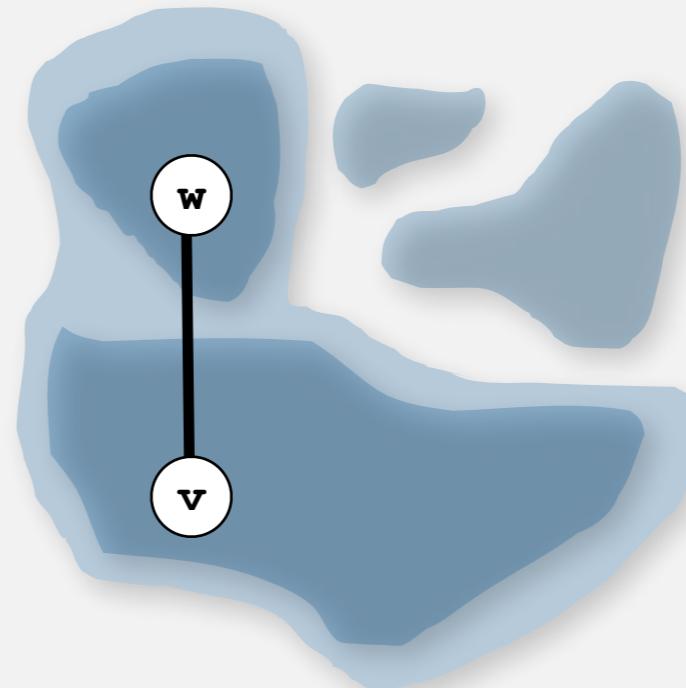
Challenge. Would adding edge $v-w$ to tree T create a cycle? If not, add it.

Efficient solution. Use the **union-find** data structure.

- Maintain a set for each connected component in T .
- If v and w are in same set, then adding $v-w$ would create a cycle.
- To add $v-w$ to T , merge sets containing v and w .



Case 1: adding $v-w$ creates a cycle



Case 2: add $v-w$ to T and merge sets containing v and w

Kruskal's algorithm: Java implementation

```
public class KruskalMST
{
    private Queue<Edge> mst = new Queue<Edge>();

    public KruskalMST(EdgeWeightedGraph G)
    {
        MinPQ<Edge> pq = new MinPQ<Edge>();
        for (Edge e : G.edges())
            pq.insert(e);

        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V()-1)
        {
            Edge e = pq.delMin();
            int v = e.either(), w = e.other(v);
            if (!uf.connected(v, w))
            {
                uf.union(v, w);
                mst.enqueue(e);
            }
        }

        public Iterable<Edge> edges()
        { return mst; }
    }
}
```

build priority queue

greedily add edges to MST

edge v-w does not create cycle

merge sets

add edge to MST

Kruskal's algorithm: running time

Proposition. Kruskal's algorithm computes MST in time proportional to $E \log E$ (in the worst case).

Pf.

operation	frequency	time per op
build pq	1	E
delete-min	E	$\log E$
union	V	$\log^* V \dagger$
connected	E	$\log^* V \dagger$

← log* function:
number of times needed to take
the lg of a number until reaching 1

† amortized bound using weighted quick union with path compression

recall: $\log^* V \leq 5$ in this universe

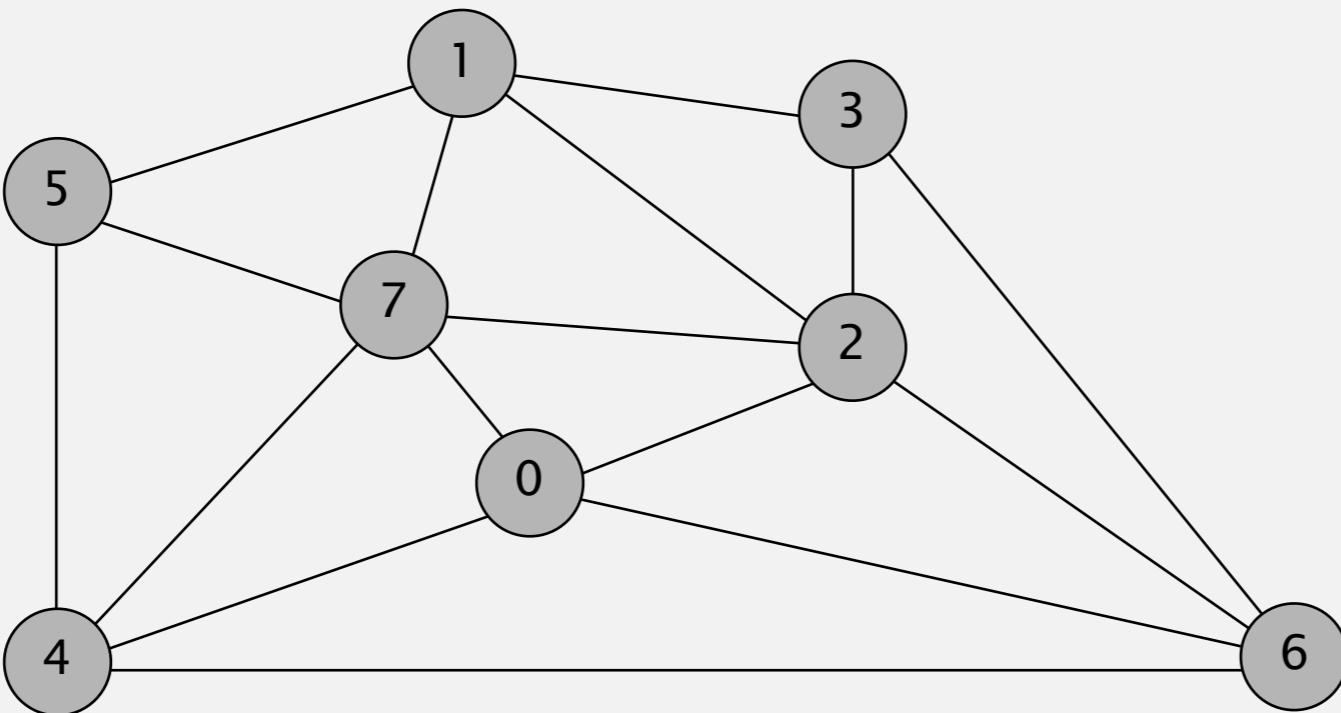
Remark. If edges are already sorted, order of growth is $E \log^* V$.

MINIMUM SPANNING TREES

- ▶ Greedy algorithm
- ▶ Edge-weighted graph API
- ▶ Kruskal's algorithm
- ▶ Prim's algorithm
- ▶ Context

Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

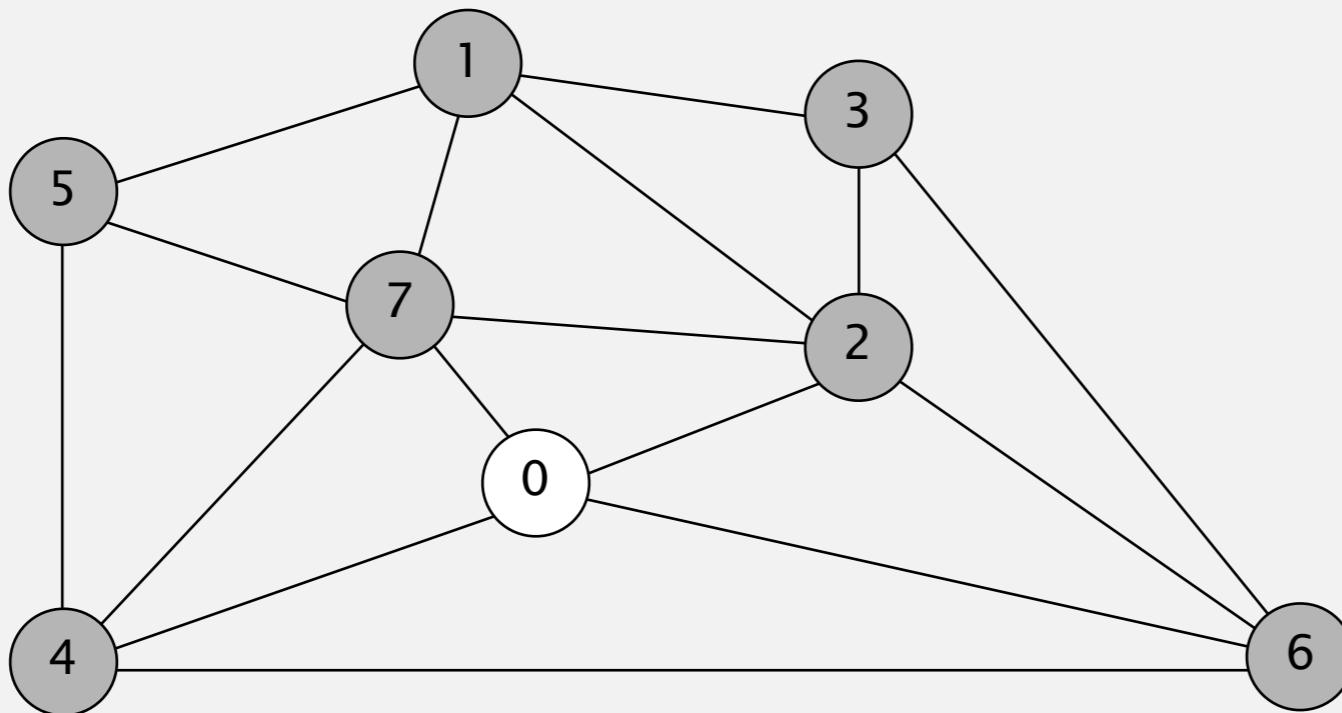


an edge-weighted graph

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

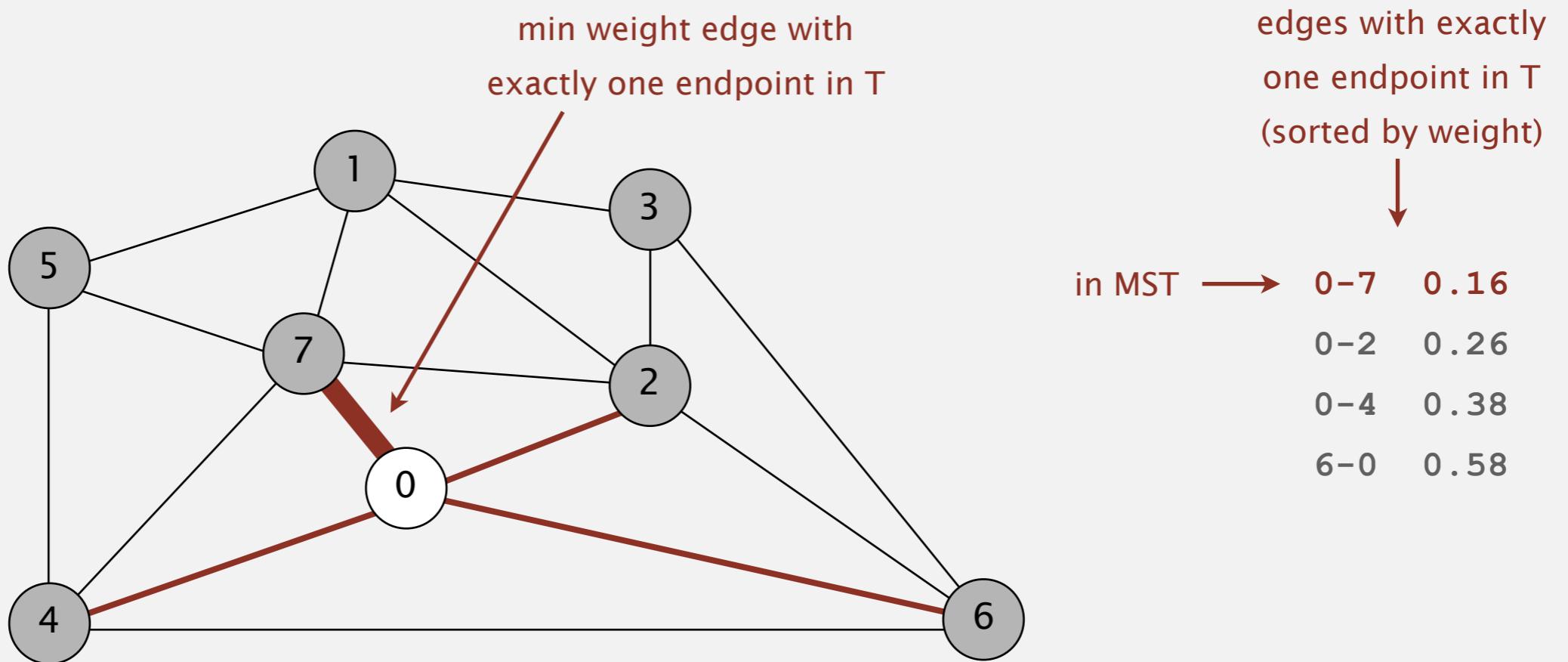
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



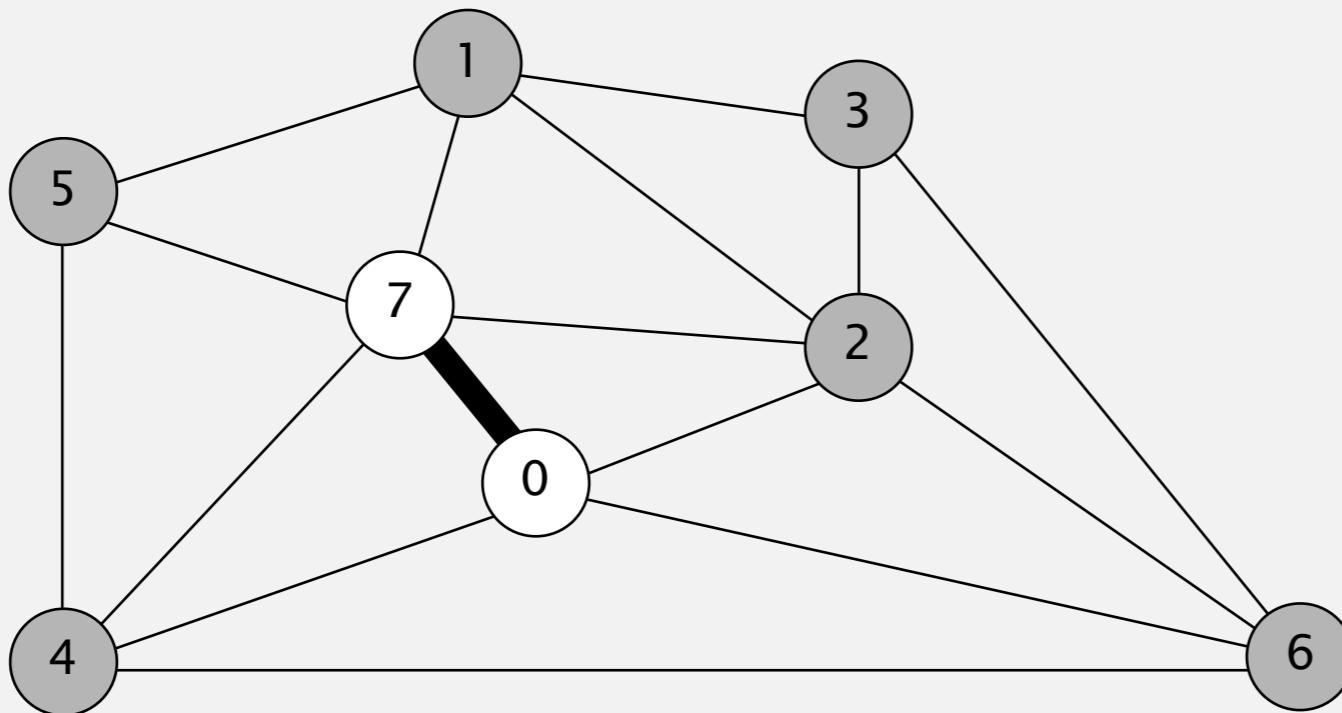
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

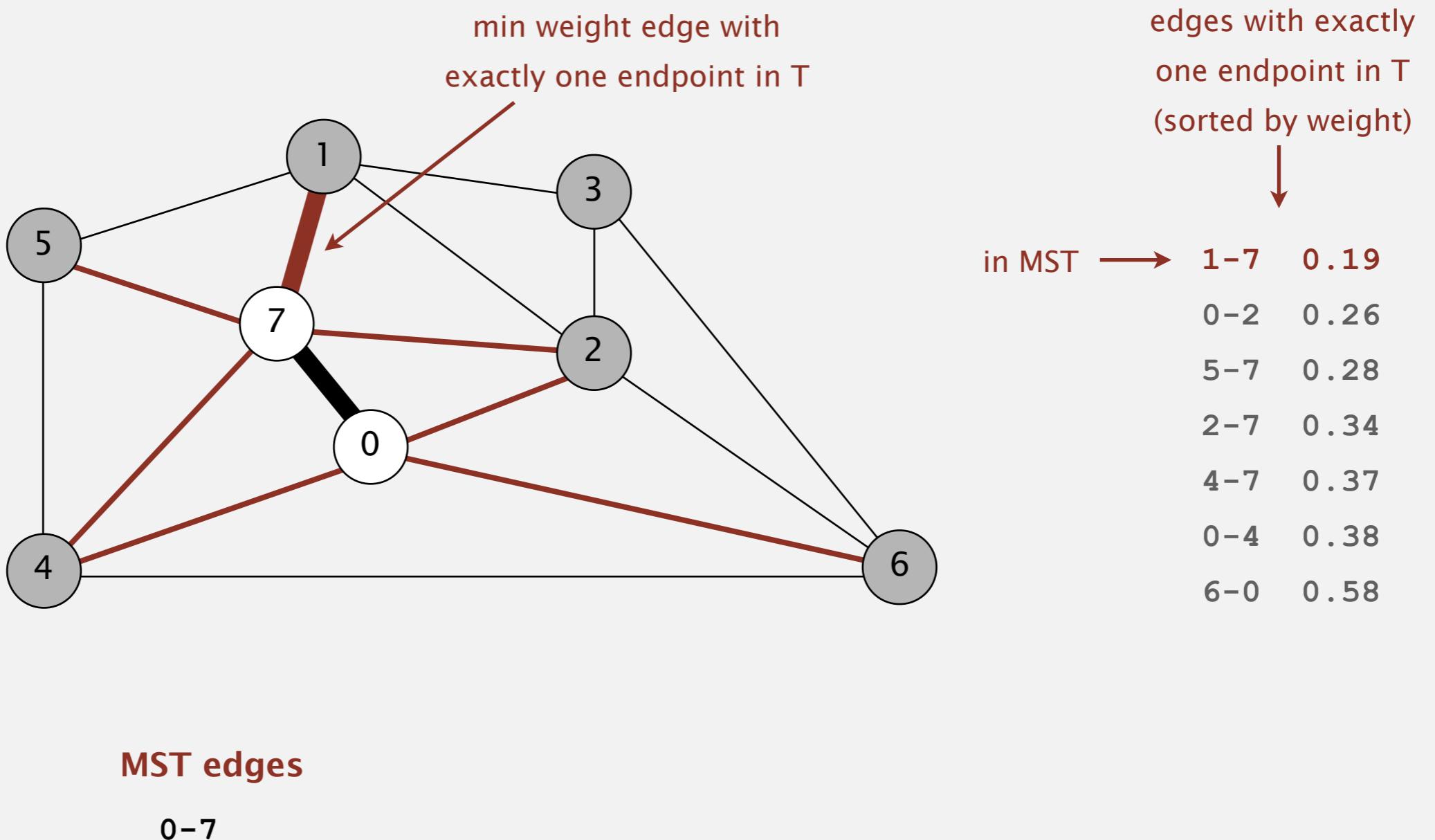


MST edges

0-7

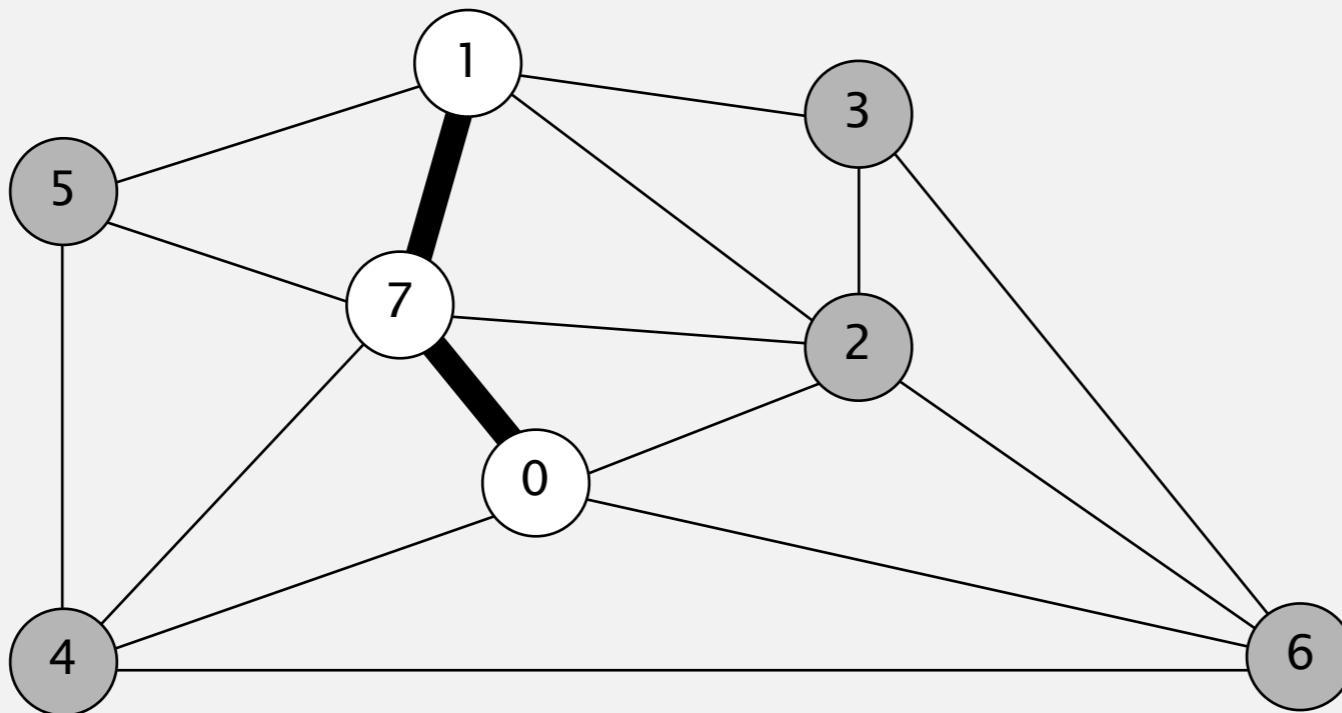
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

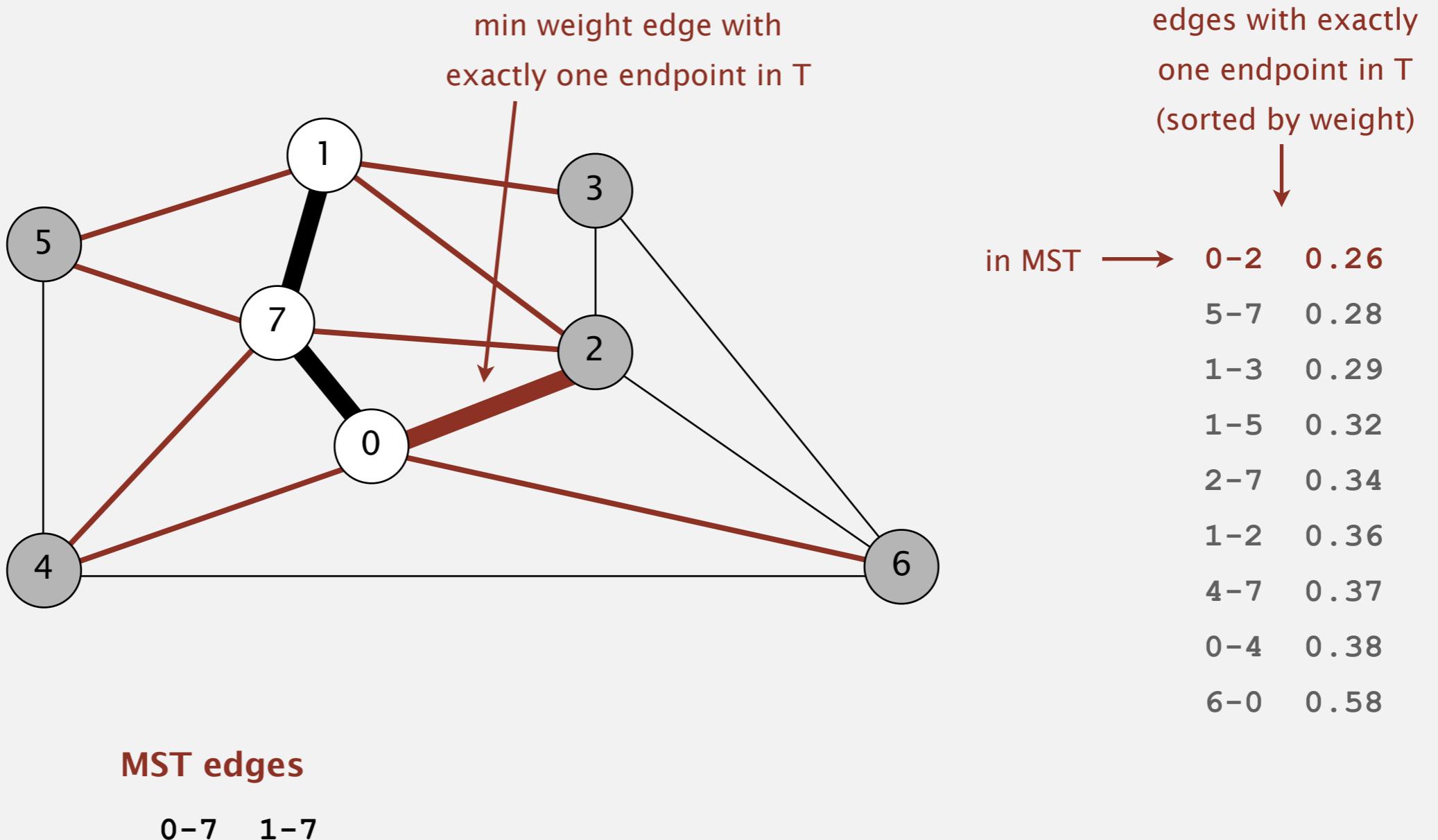


MST edges

0-7 1-7

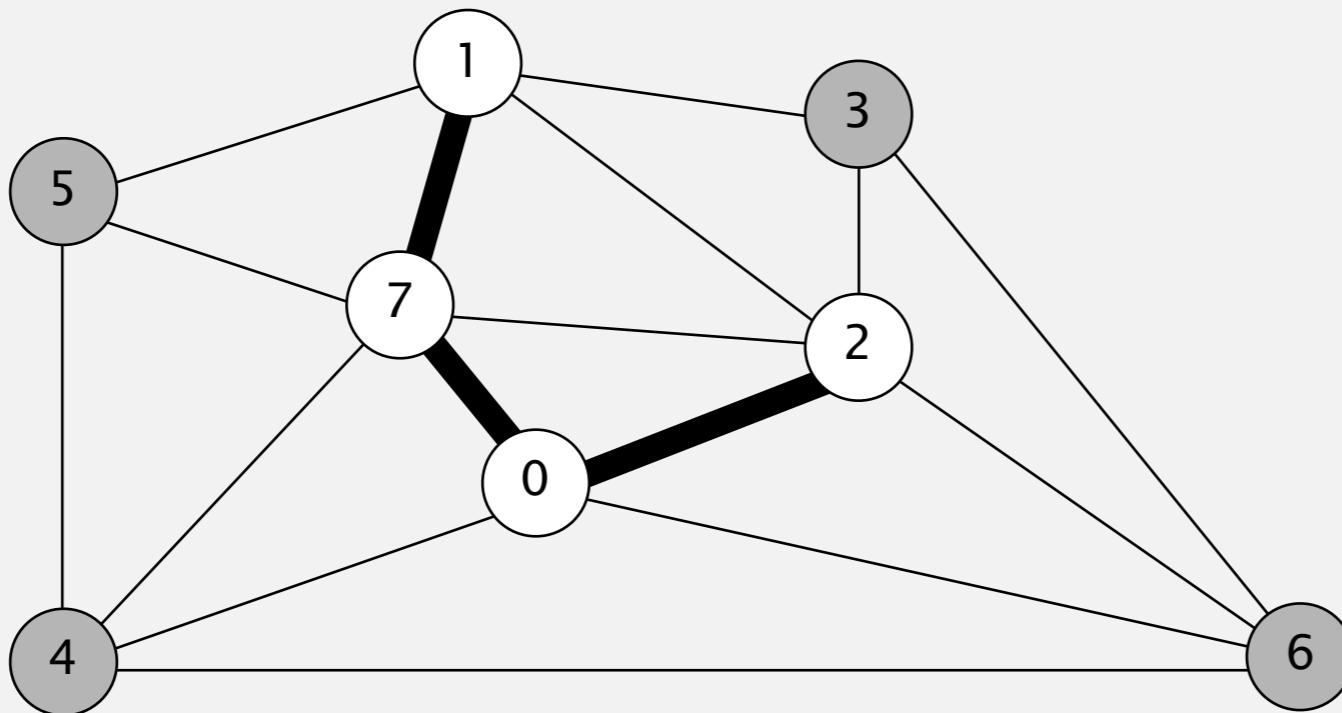
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

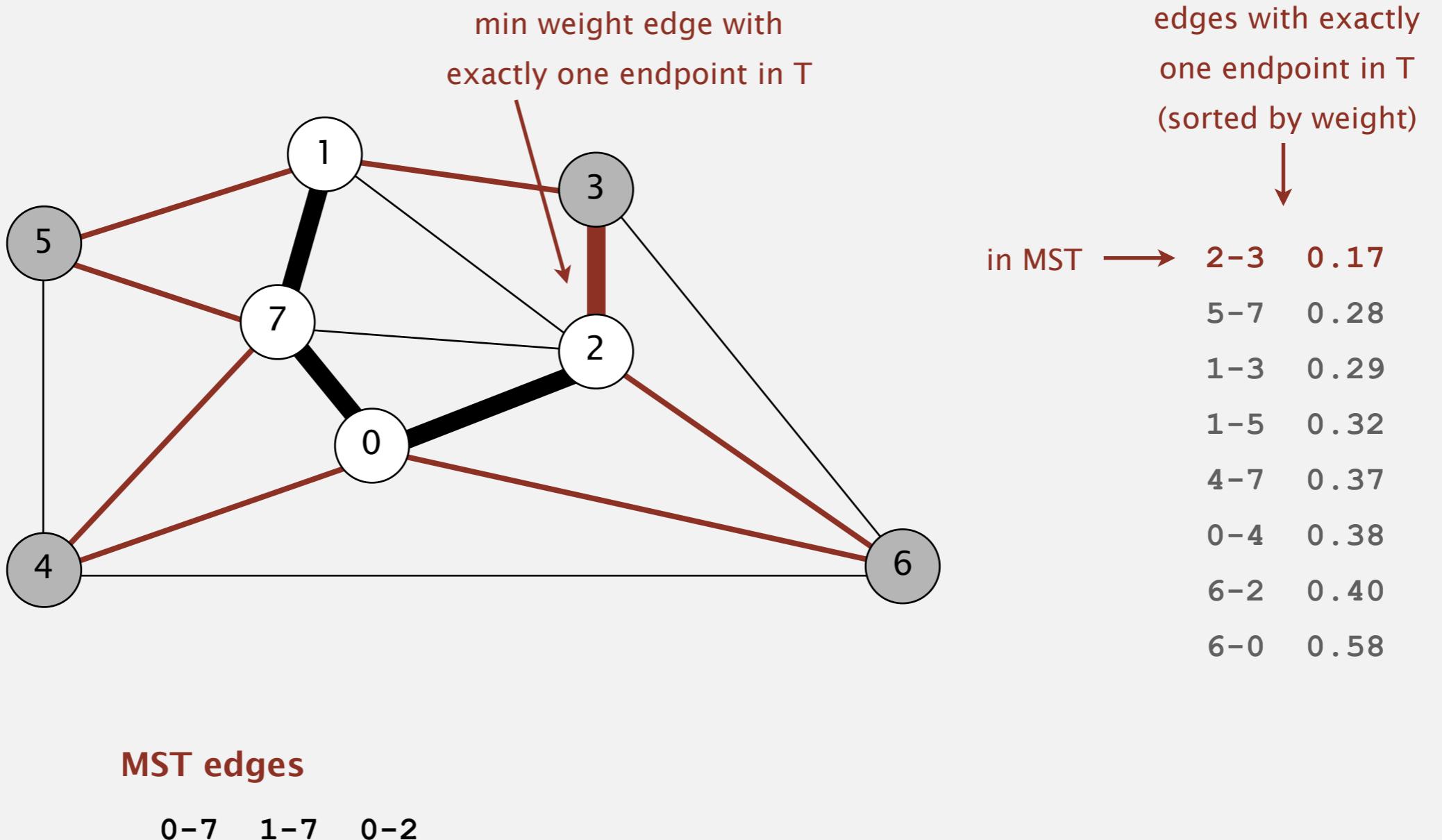


MST edges

0-7 1-7 0-2

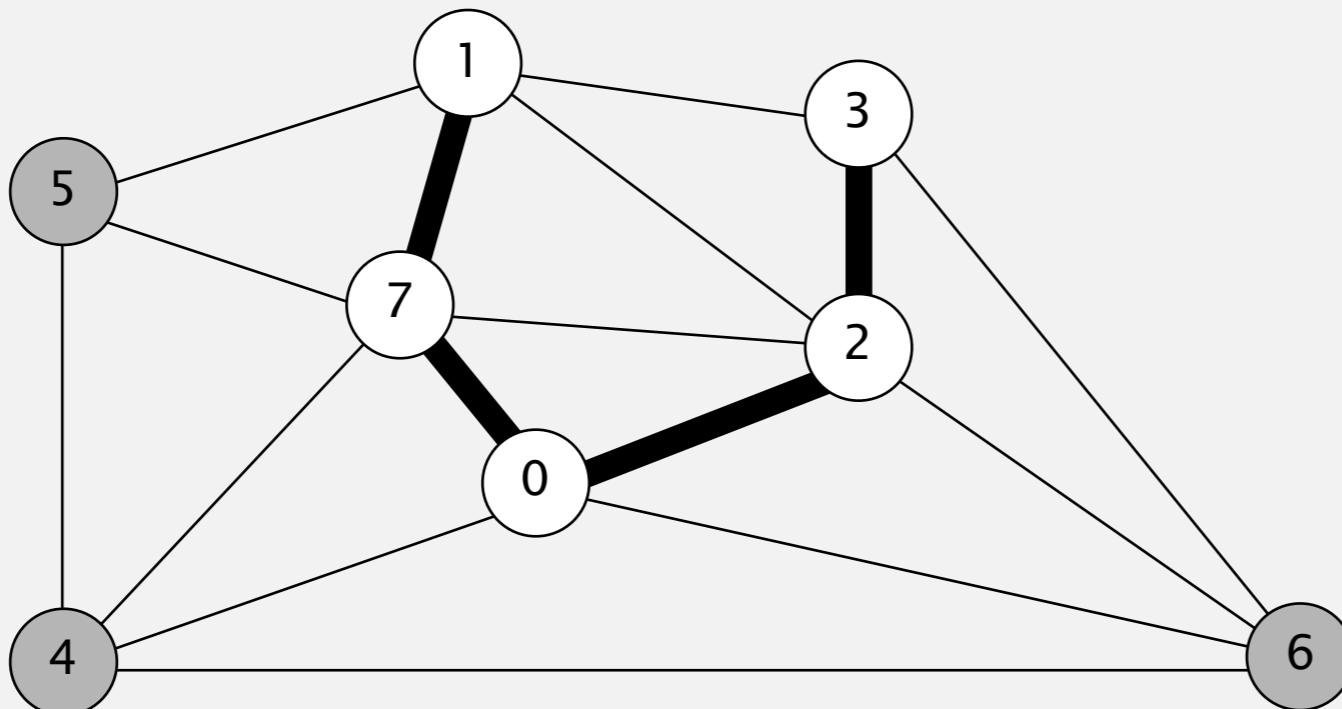
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

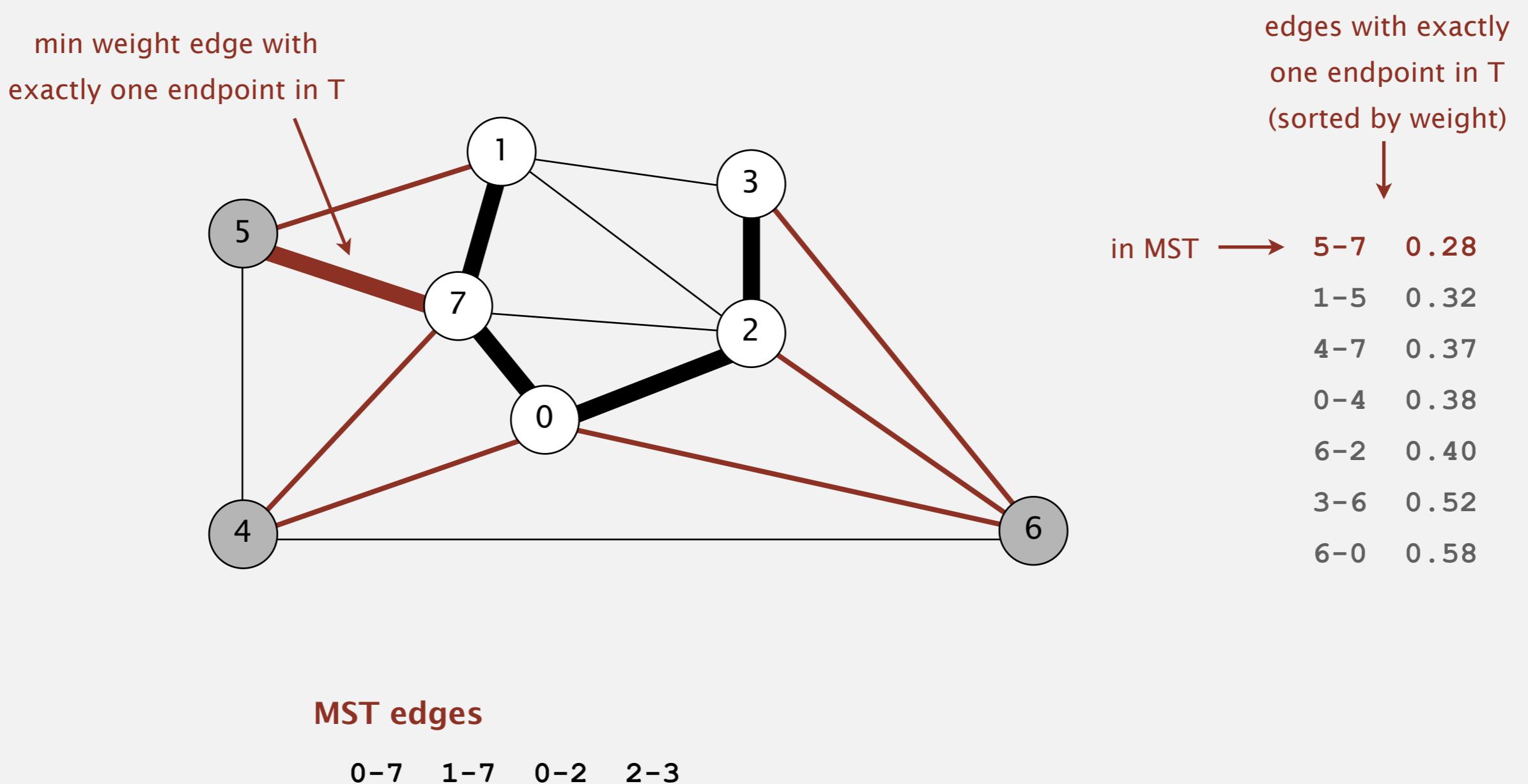


MST edges

0-7 1-7 0-2 2-3

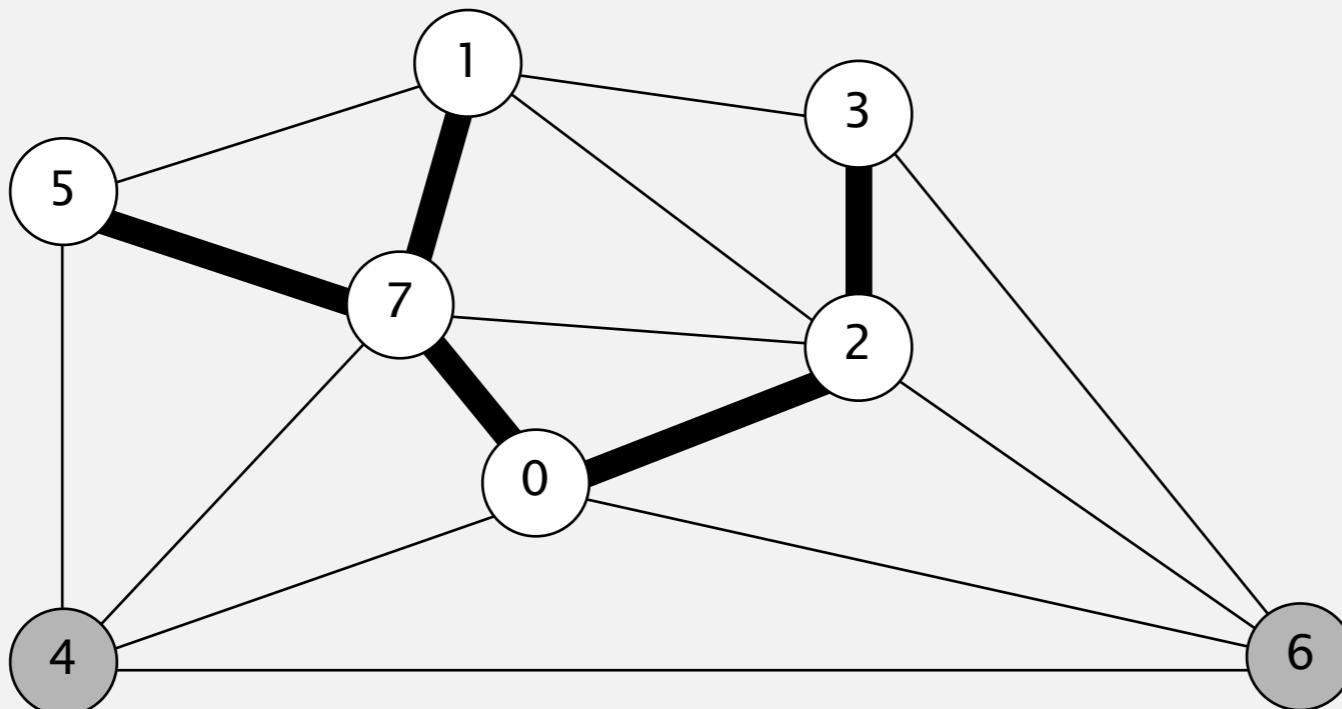
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

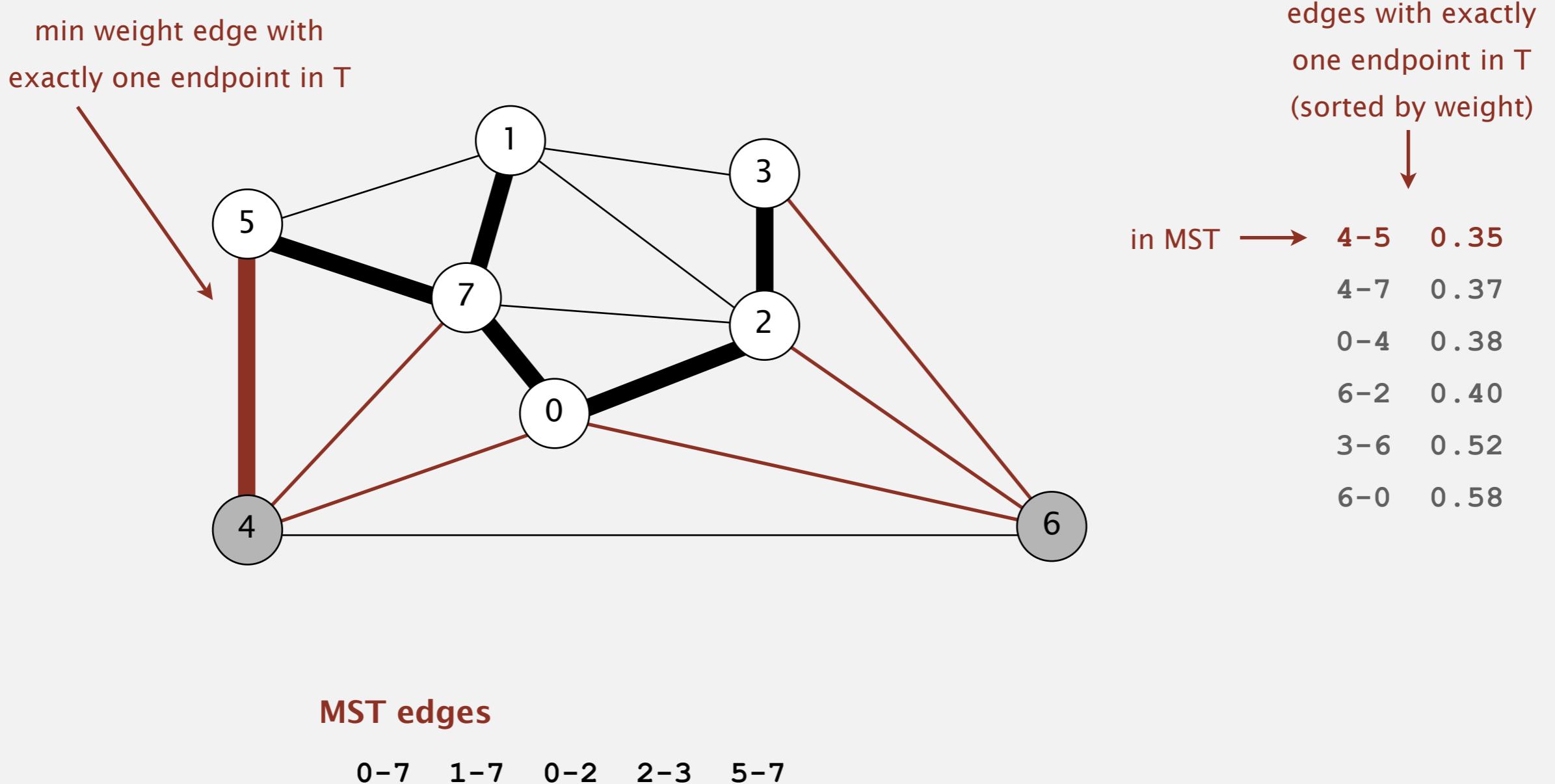


MST edges

0-7 1-7 0-2 2-3 5-7

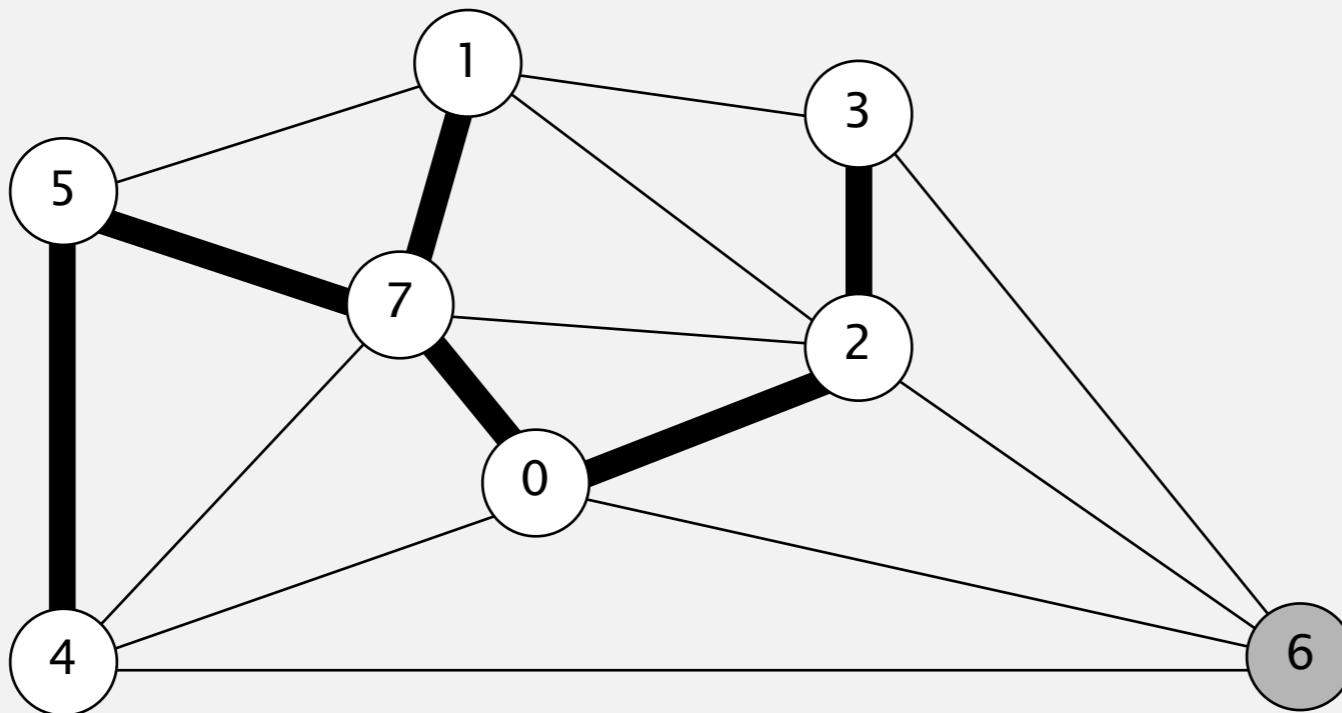
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

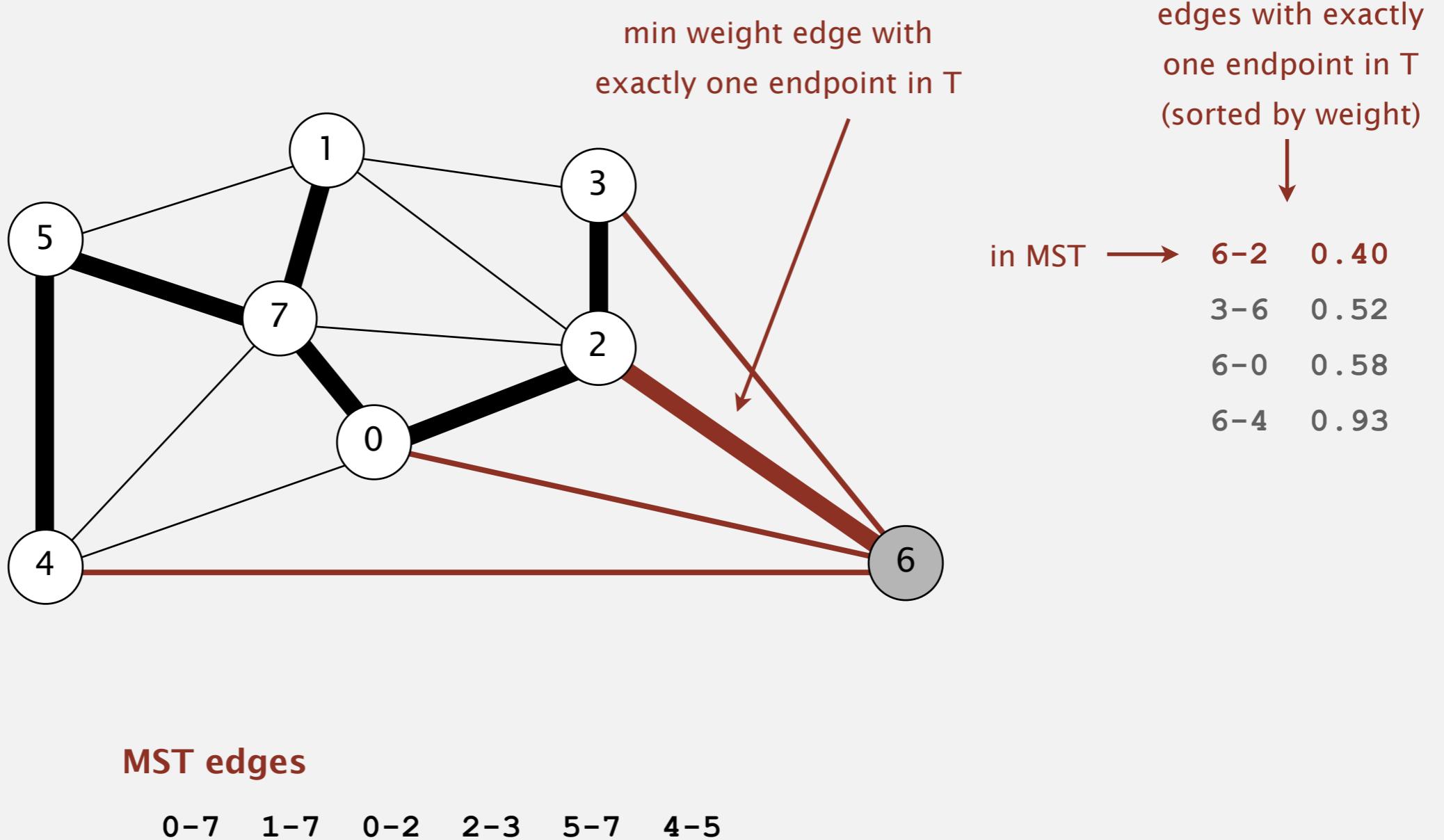


MST edges

0-7 1-7 0-2 2-3 5-7 4-5

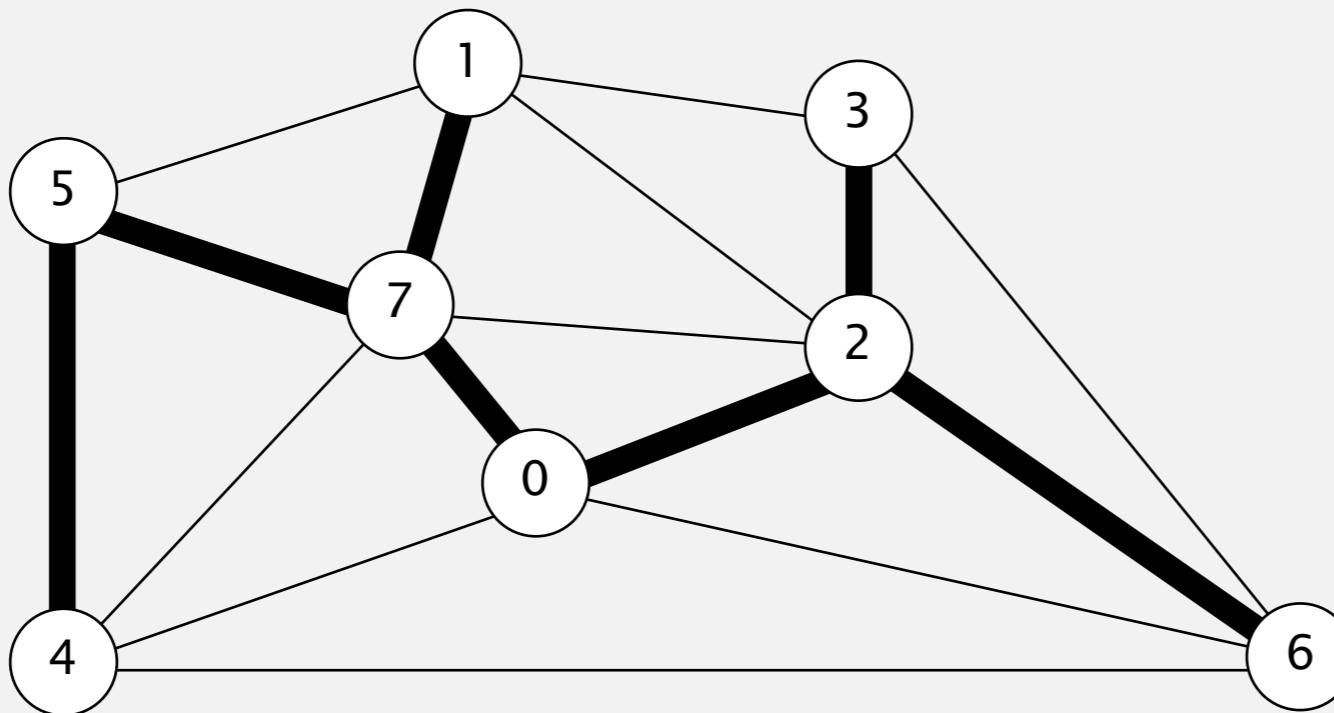
Prim's algorithm

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm

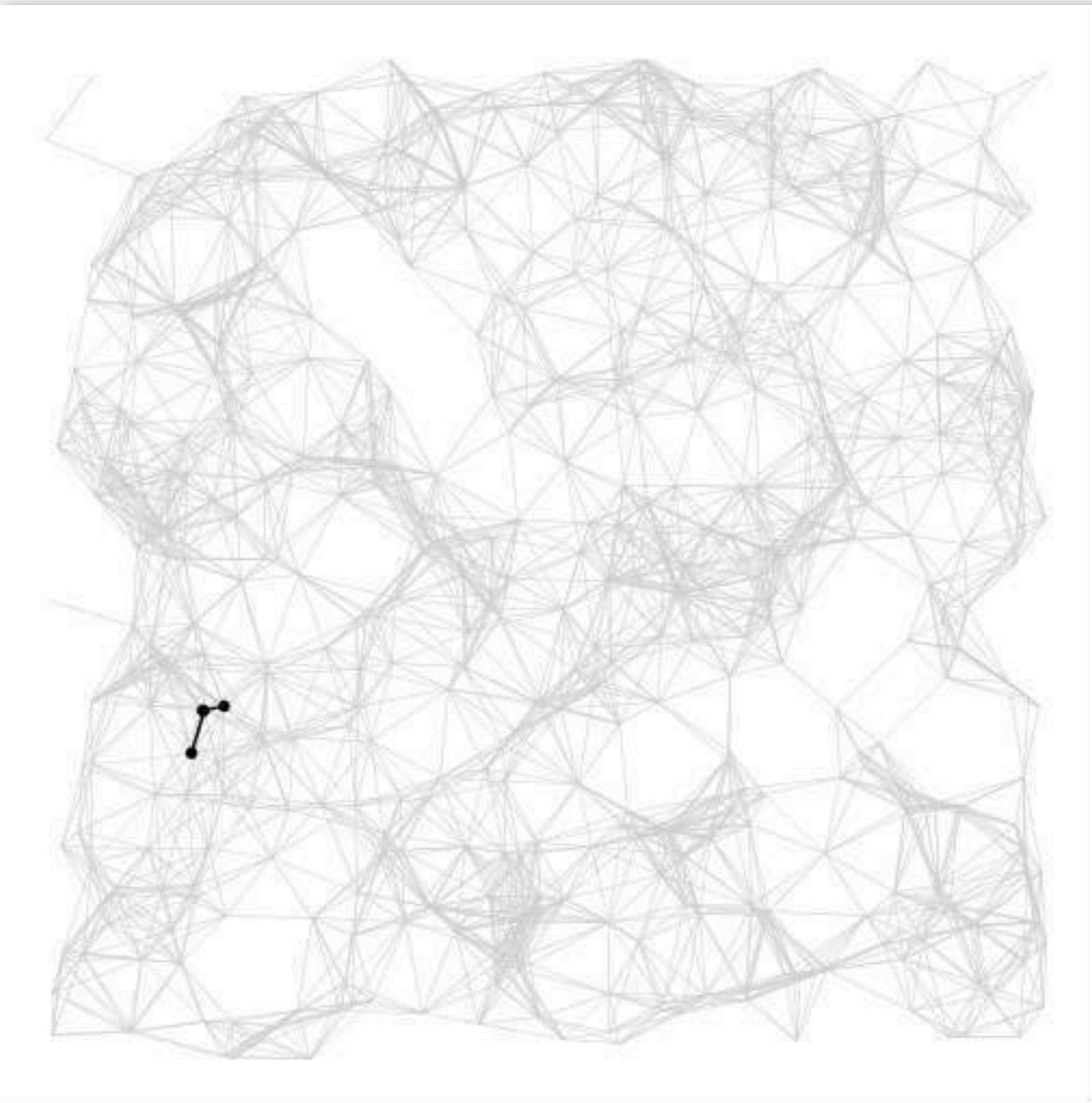
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



MST edges

0-7 1-7 0-2 2-3 5-7 4-5 6-2

Prim's algorithm: visualization



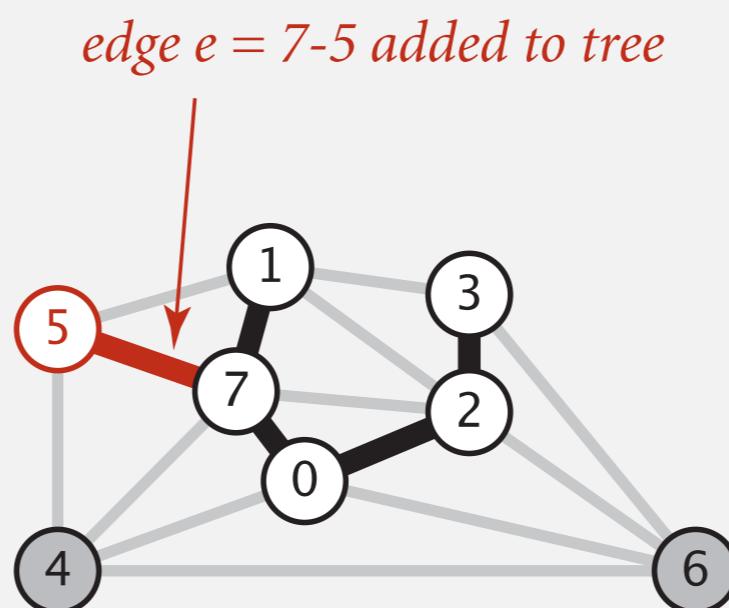
Prim's algorithm: proof of correctness

Proposition. [Jarník 1930, Dijkstra 1957, Prim 1959]

Prim's algorithm computes the MST.

Pf. Prim's algorithm is a special case of the greedy MST algorithm.

- Suppose edge $e = \min$ weight edge connecting a vertex on the tree to a vertex not on the tree.
- Cut = set of vertices connected on tree.
- No crossing edge is black.
- No crossing edge has lower weight.



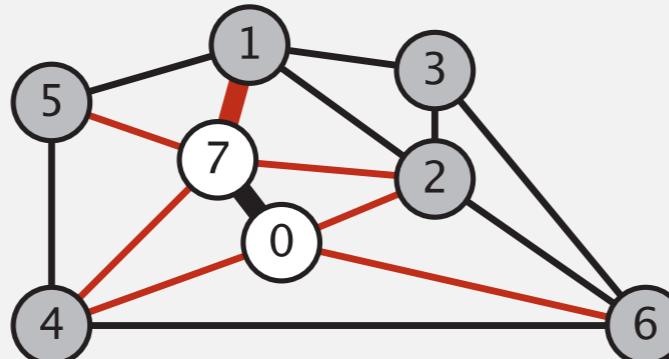
Prim's algorithm: implementation challenge

Challenge. Find the min weight edge with exactly one endpoint in T .

How difficult?

- E ← try all edges
- V
- $\log E$ ← use a priority queue !
- $\log^* E$
- 1

1-7 is min weight edge with
exactly one endpoint in T



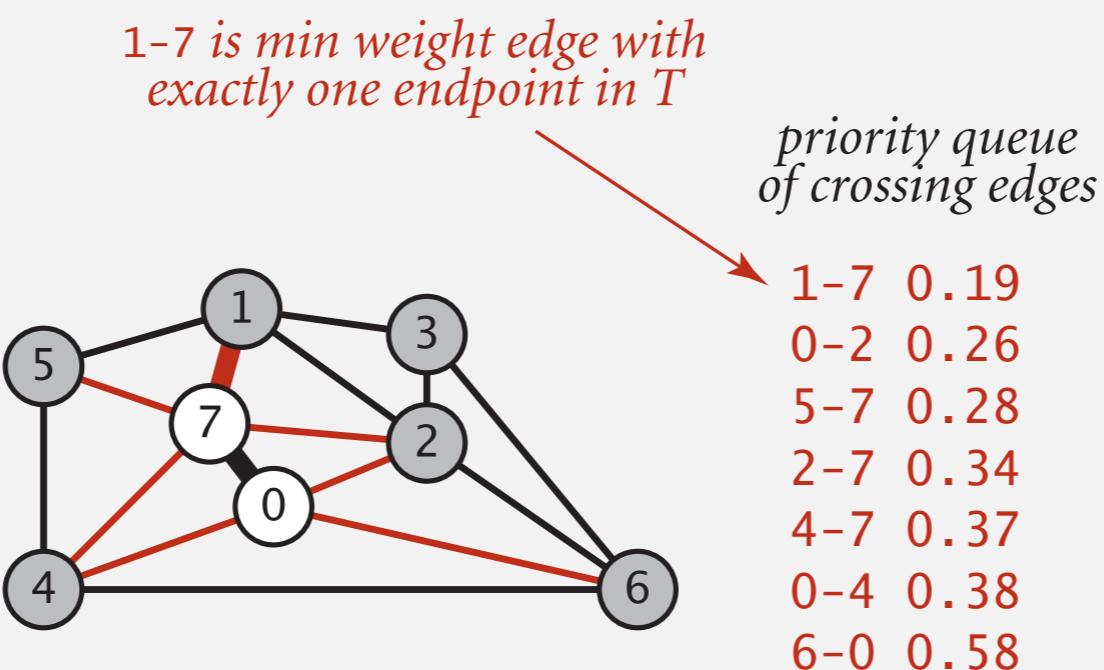
1-7 0.19
0-2 0.26
5-7 0.28
2-7 0.34
4-7 0.37
0-4 0.38
6-0 0.58

Prim's algorithm: lazy implementation

Challenge. Find the min weight edge with exactly one endpoint in T .

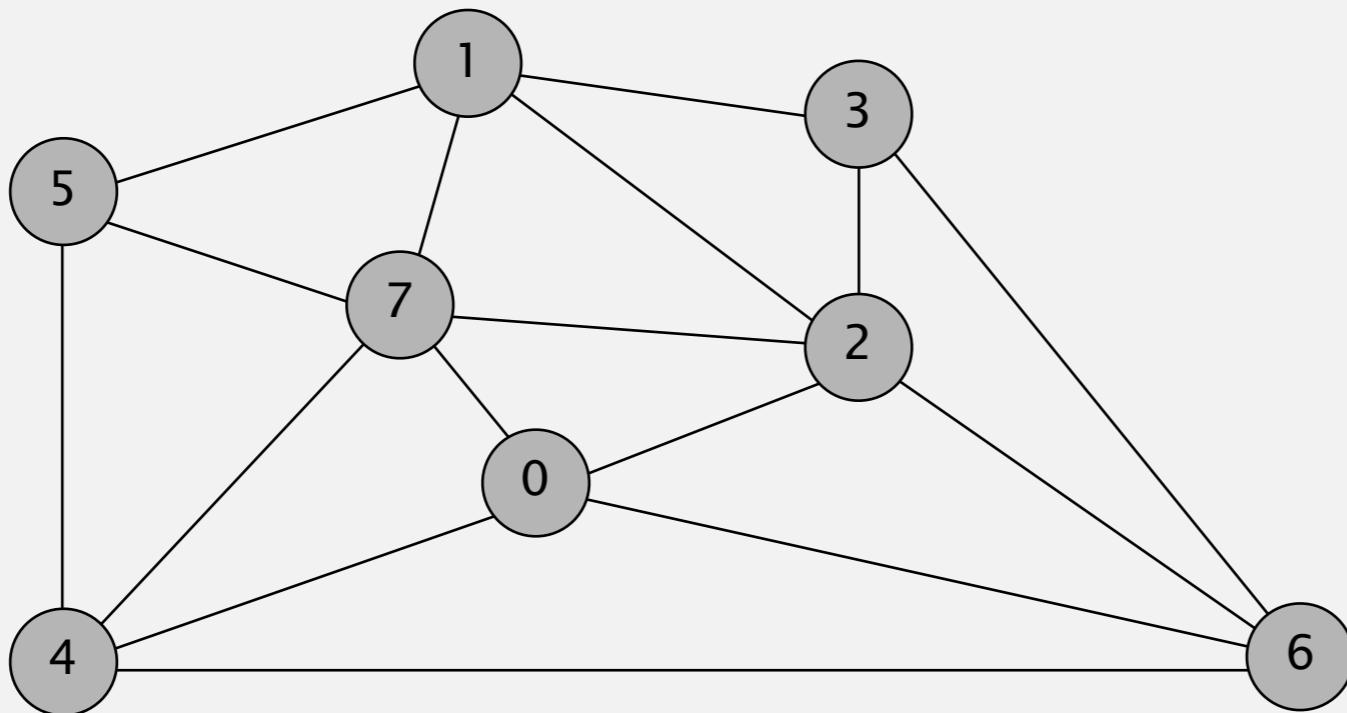
Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T .

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v-w$ to add to T .
- Disregard if both endpoints v and w are in T .
- Otherwise, let v be vertex not in T :
 - add to PQ any edge incident to v (assuming other endpoint not in T)
 - add v to T



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

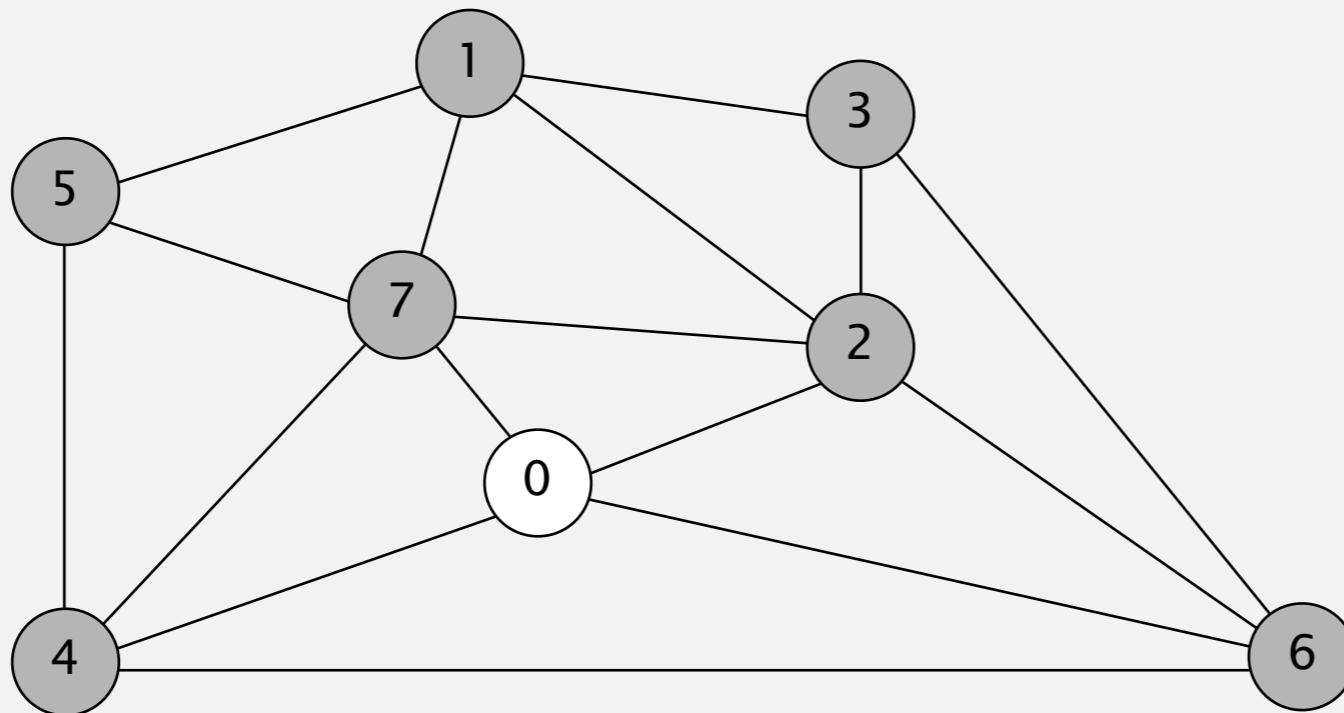


an edge-weighted graph

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

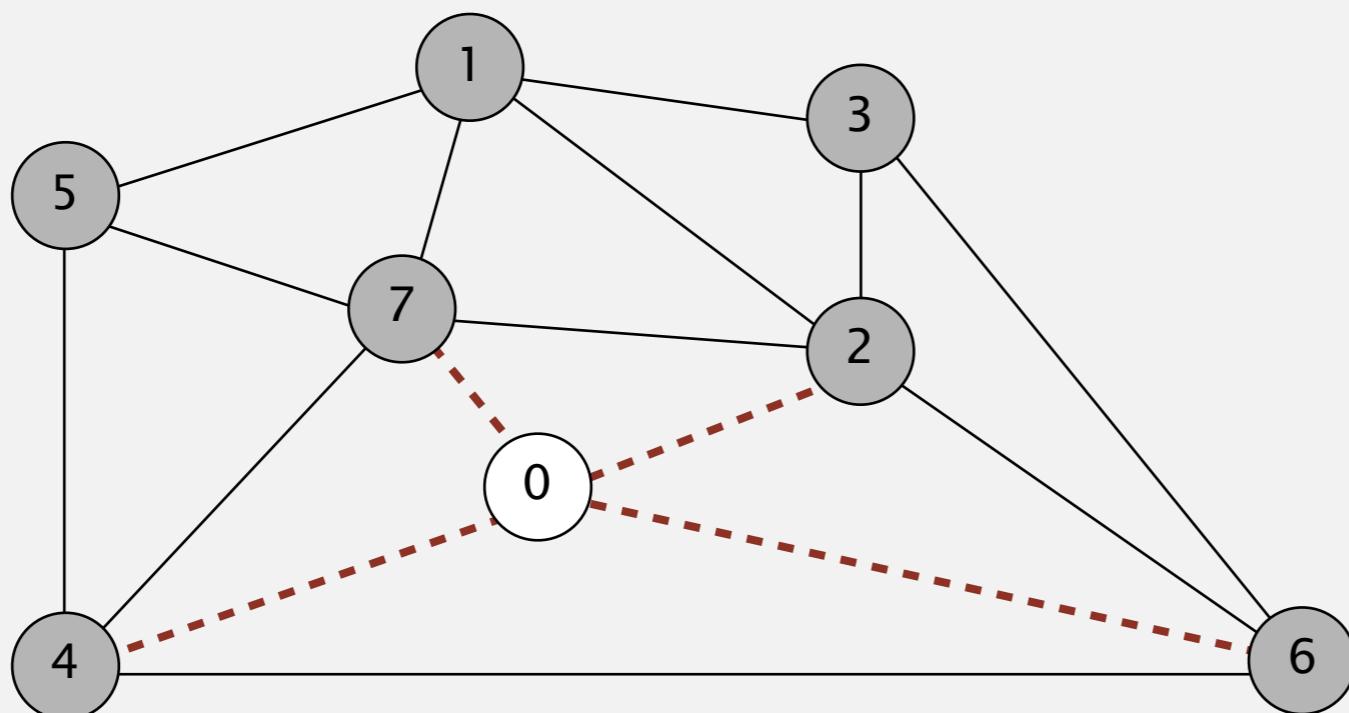
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

add to PQ all edges incident to 0



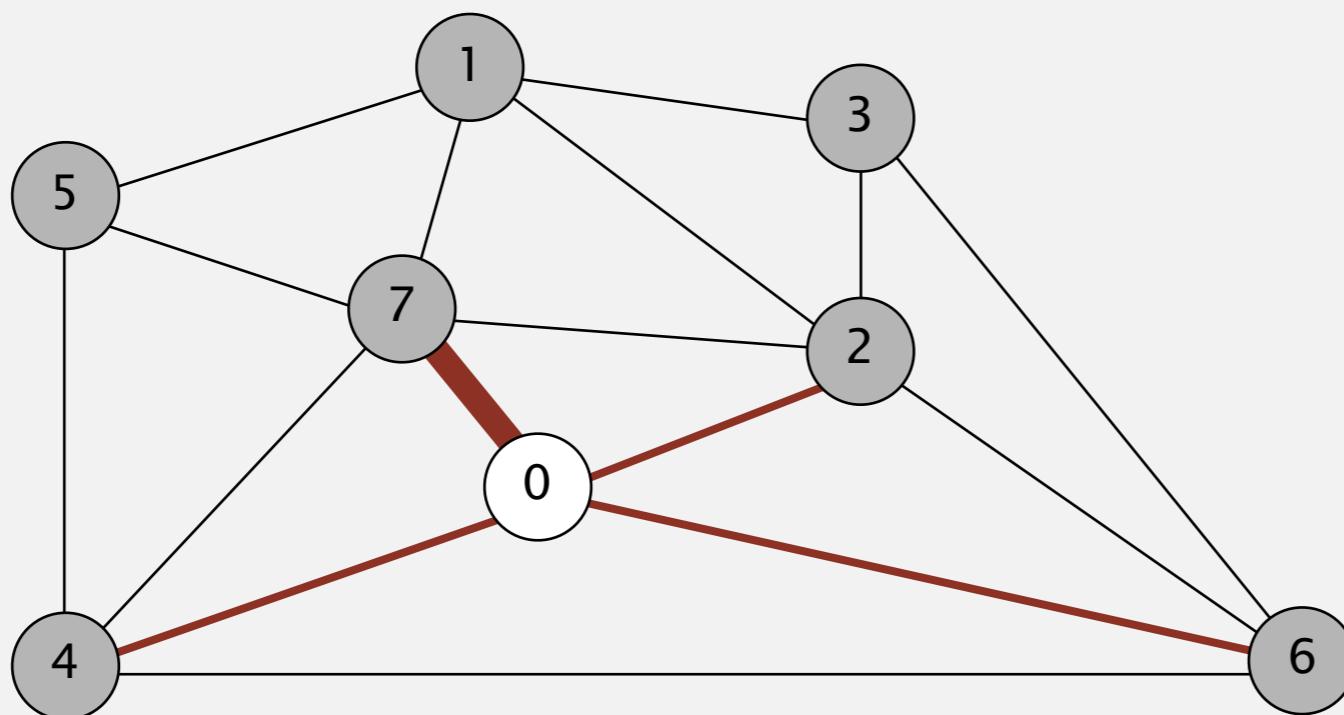
edges on PQ
(sorted by weight)

*	0-7	0.16
*	0-2	0.26
*	0-4	0.38
*	6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

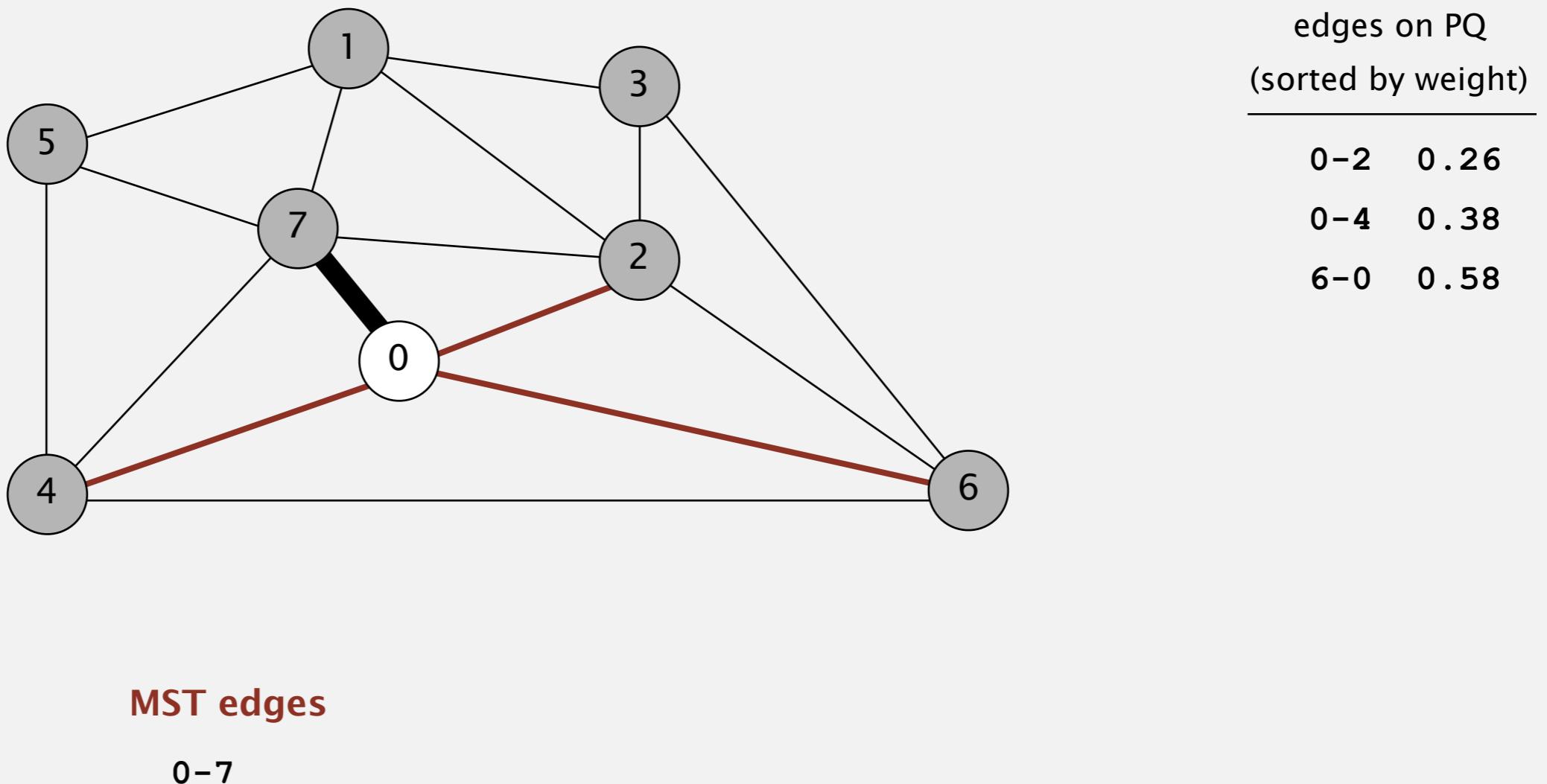
delete 0-7 and add to MST



edges on PQ (sorted by weight)	
0-7	0.16
0-2	0.26
0-4	0.38
6-0	0.58

Prim's algorithm - Lazy implementation

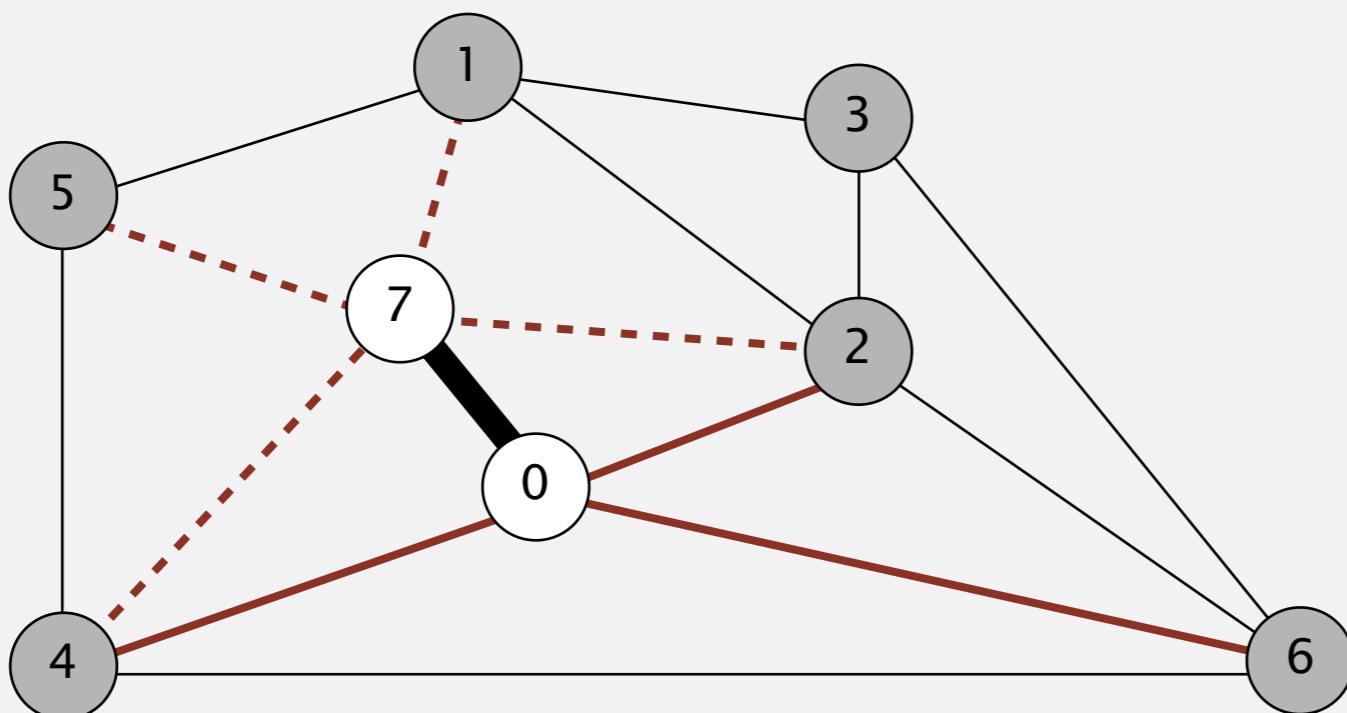
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

add to PQ all edges incident to 7

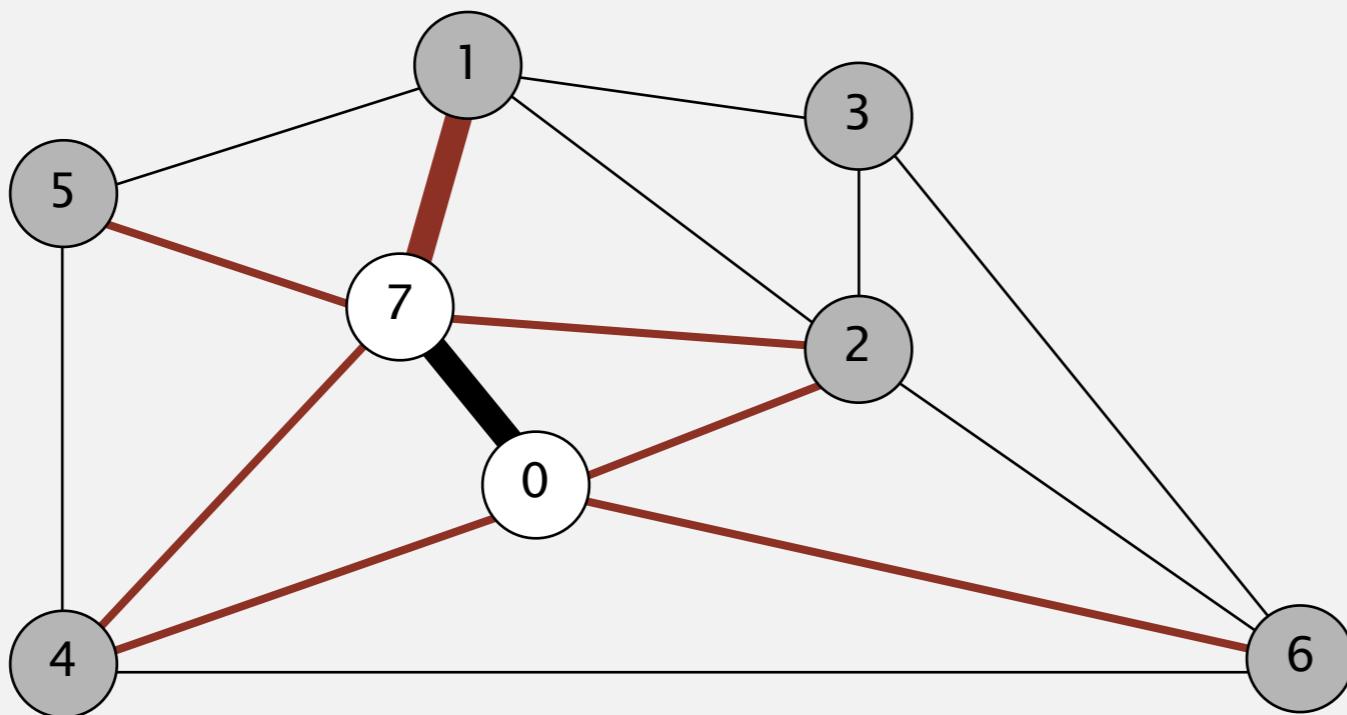


edges on PQ (sorted by weight)		
*	1-7	0.19
	0-2	0.26
*	5-7	0.28
*	2-7	0.34
*	4-7	0.37
	0-4	0.38
	6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

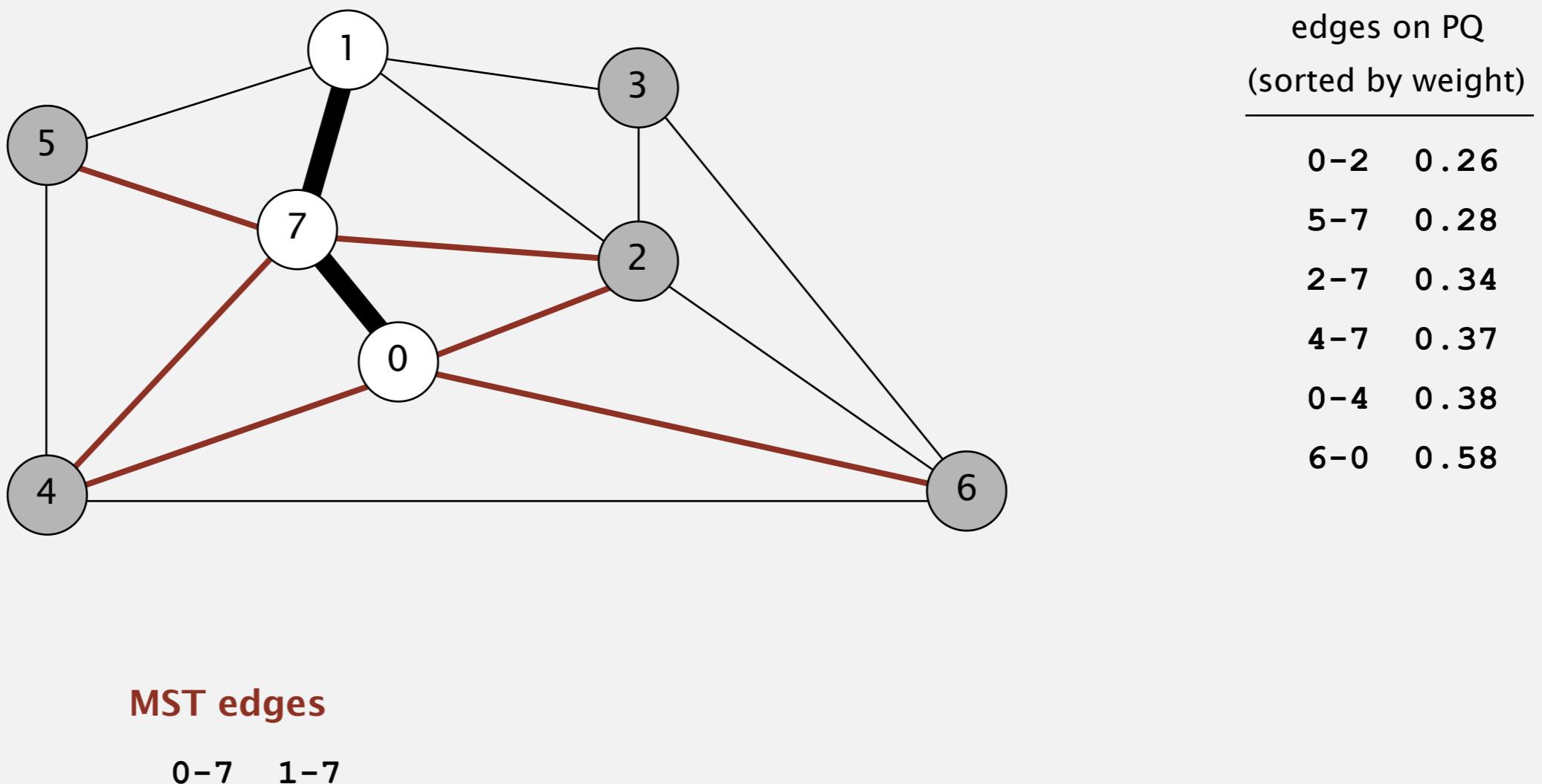
delete 1-7 and add to MST



edges on PQ (sorted by weight)	
1-7	0.19
0-2	0.26
5-7	0.28
2-7	0.34
4-7	0.37
0-4	0.38
6-0	0.58

Prim's algorithm - Lazy implementation

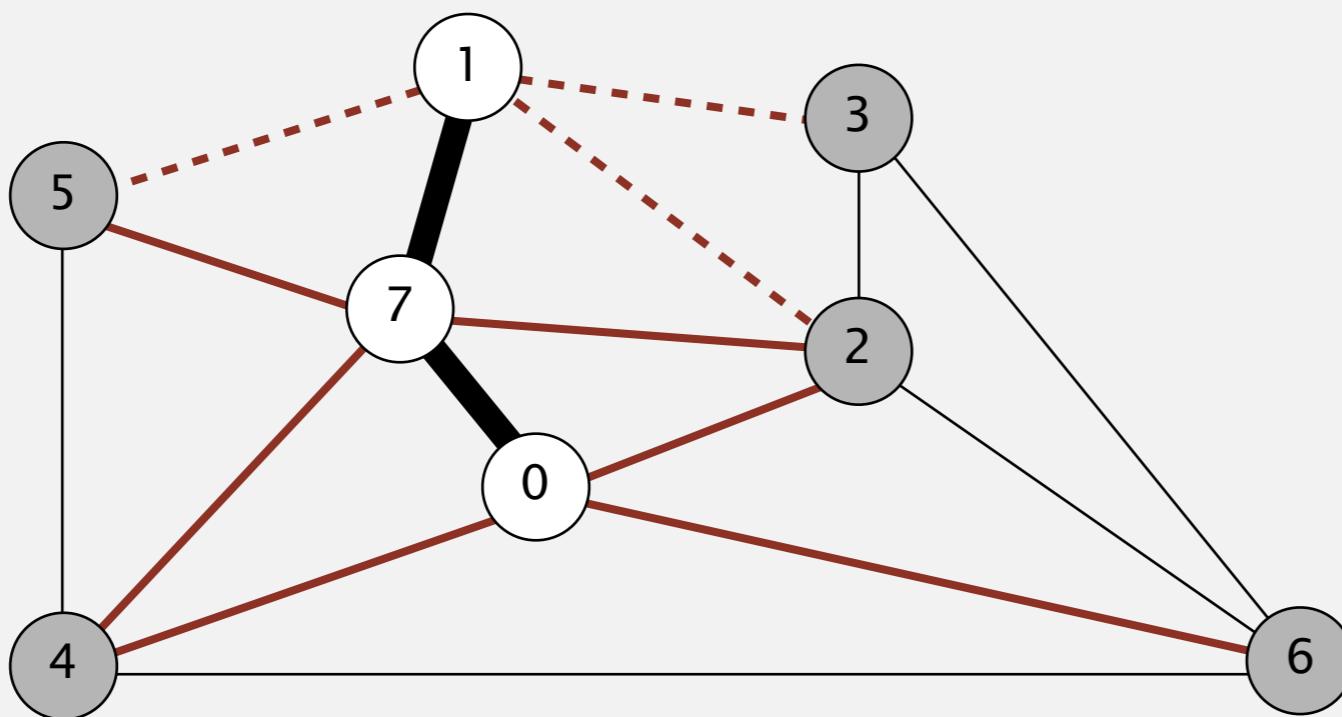
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

add to PQ all edges incident to 1



MST edges

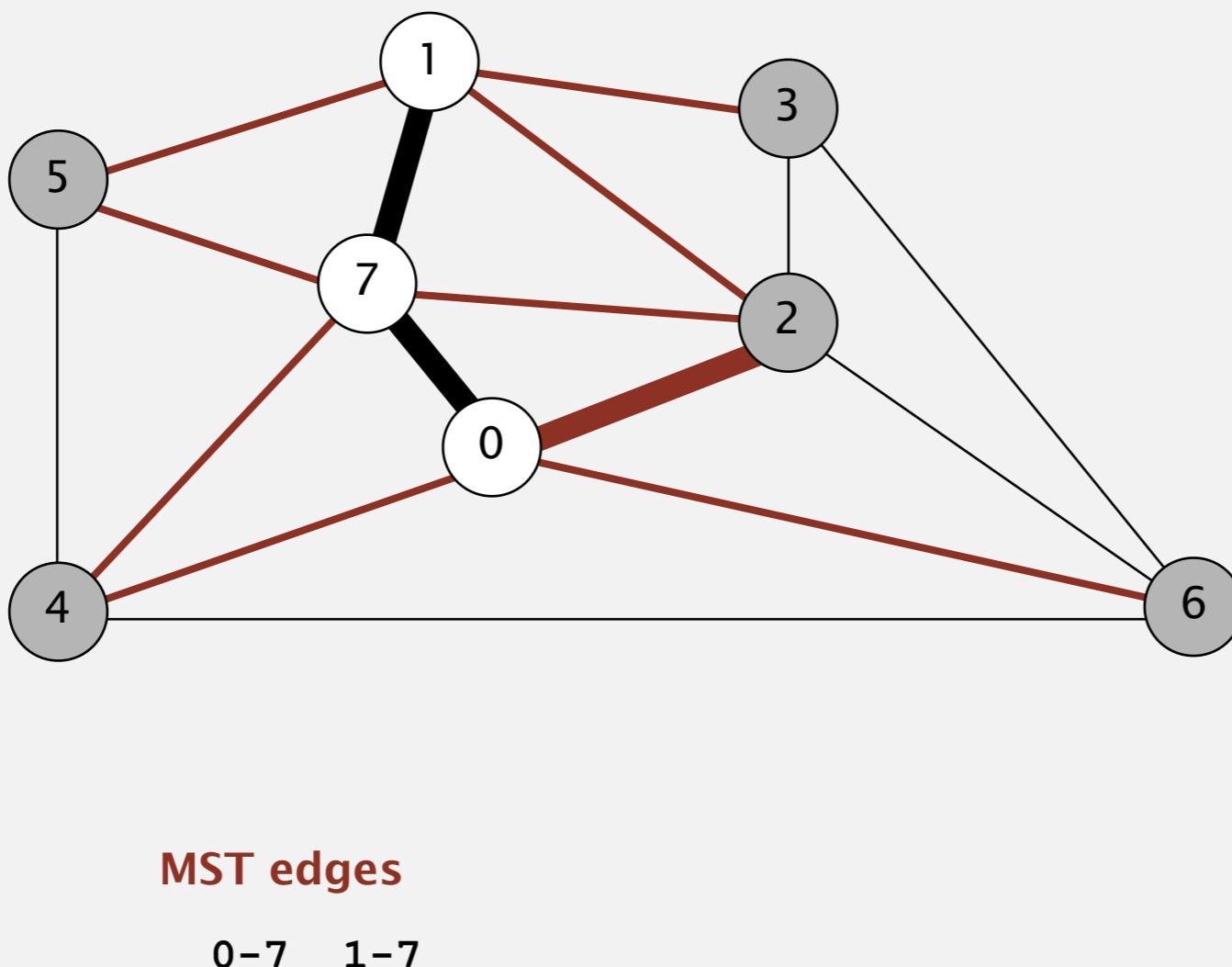
0-7 1-7

edges on PQ (sorted by weight)	
0-2	0.26
5-7	0.28
* 1-3	0.29
* 1-5	0.32
2-7	0.34
* 1-2	0.36
4-7	0.37
0-4	0.38
6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

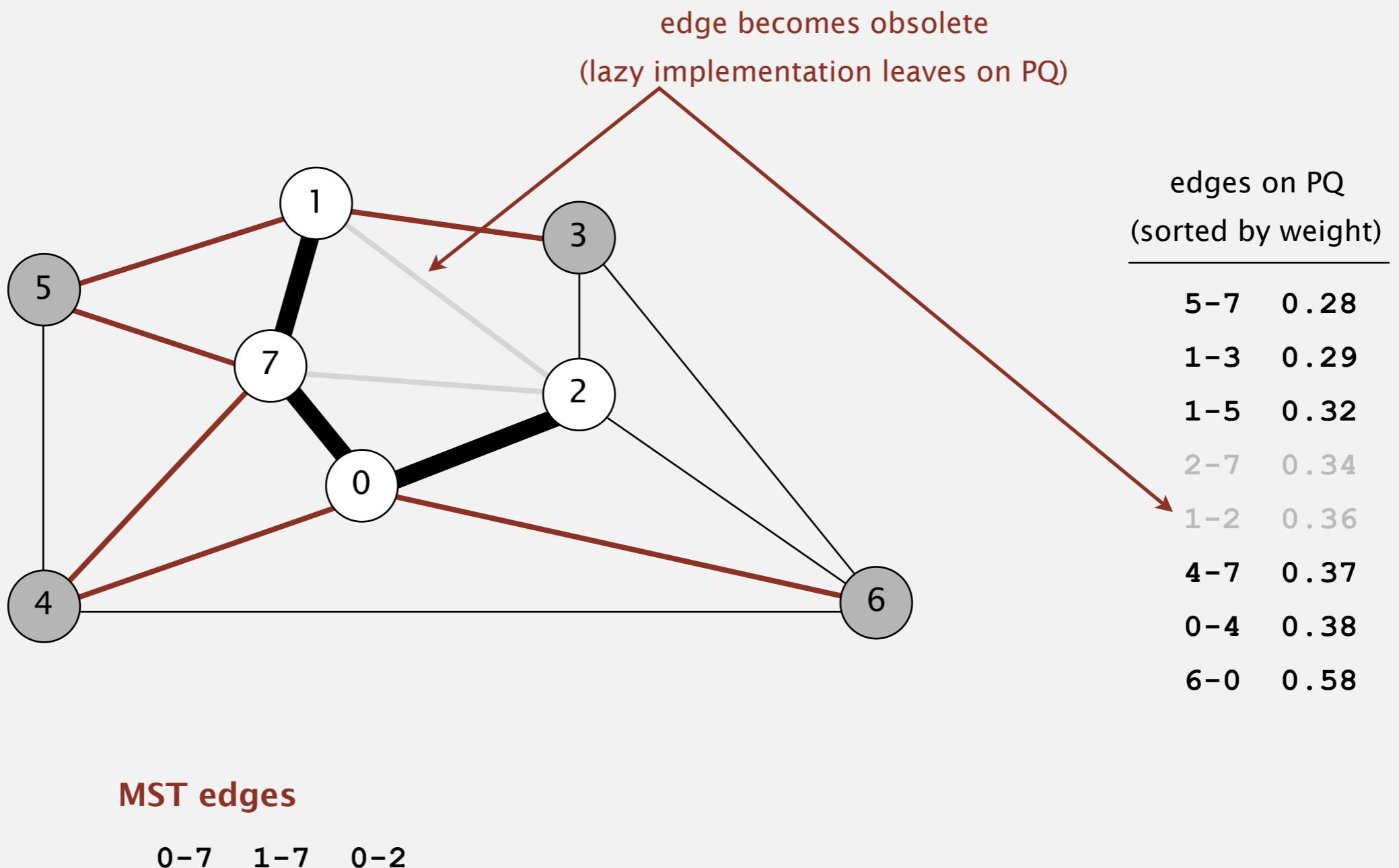
delete edge 0-2 and add to MST



edges on PQ (sorted by weight)	
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
1-2	0.36
4-7	0.37
0-4	0.38
6-0	0.58

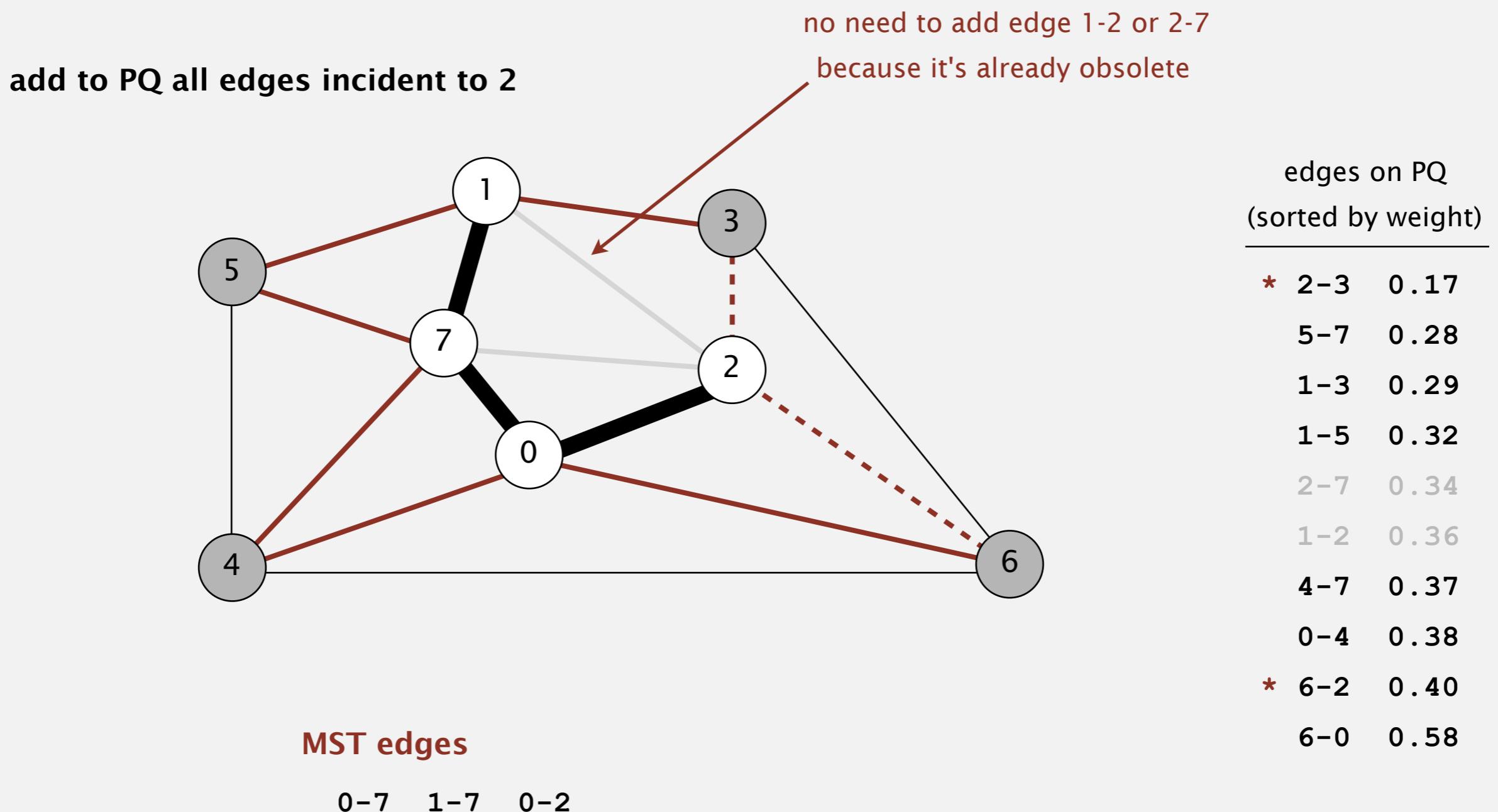
Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

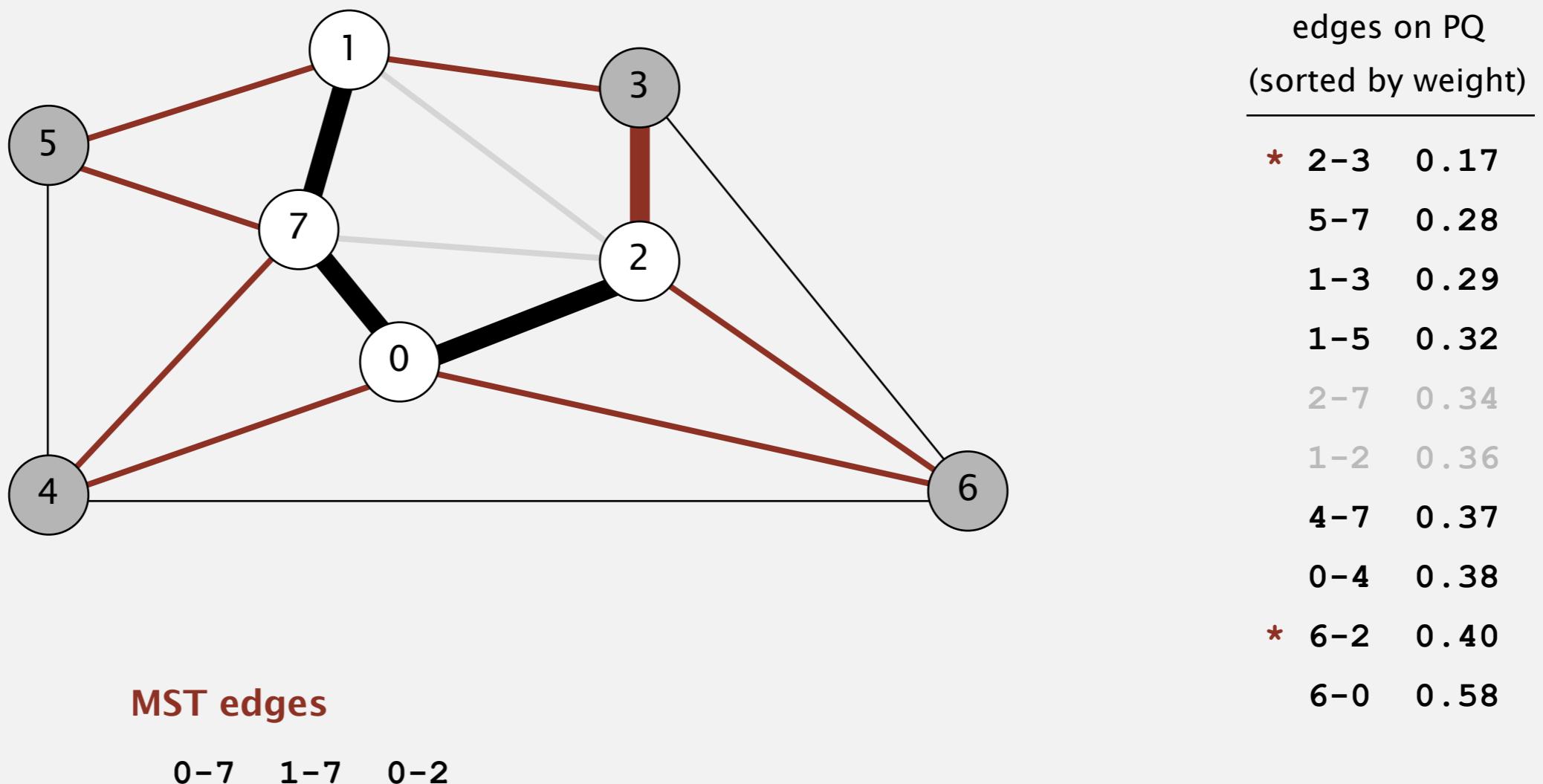
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

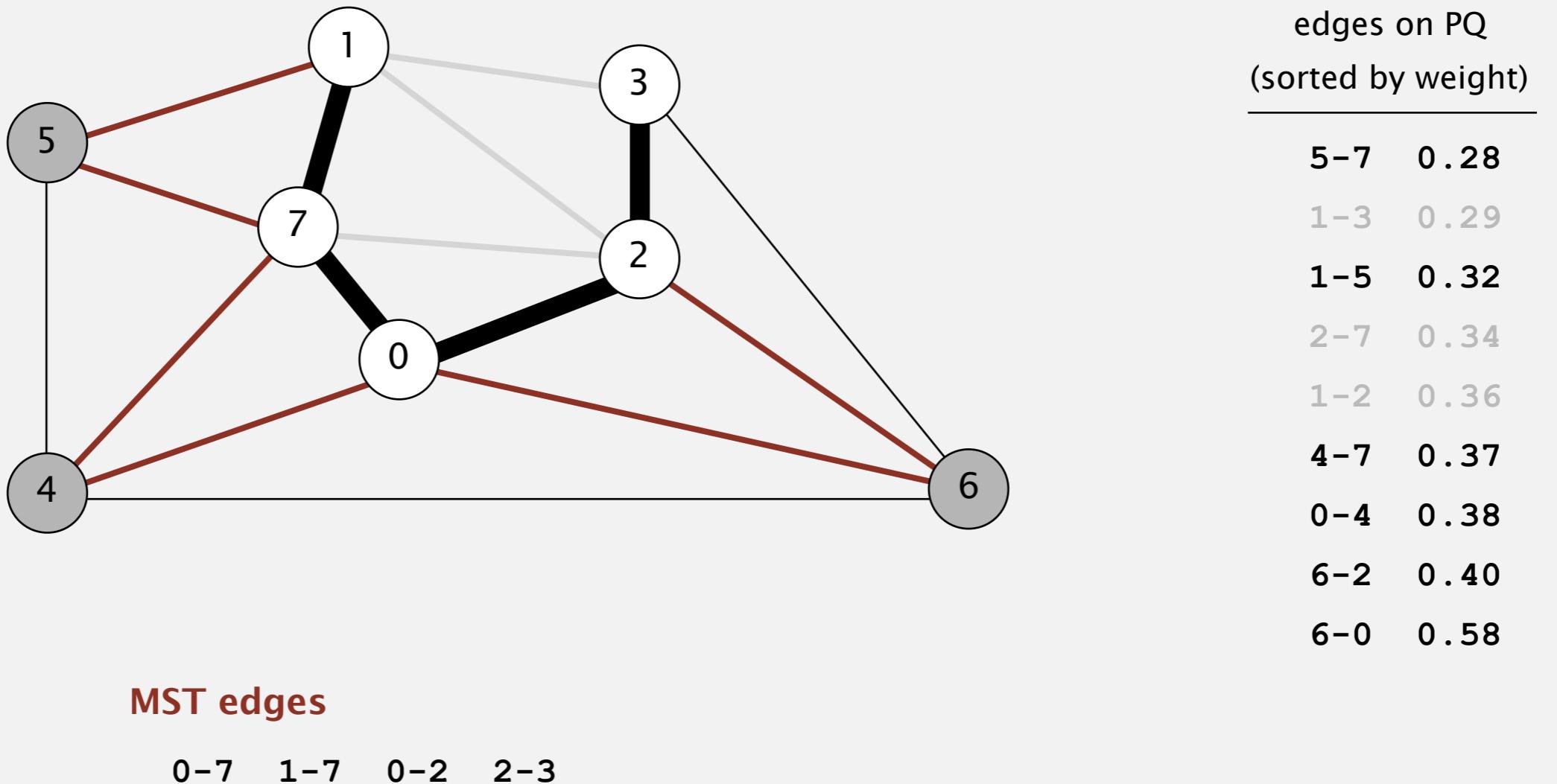
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 2-3 and add to MST



Prim's algorithm - Lazy implementation

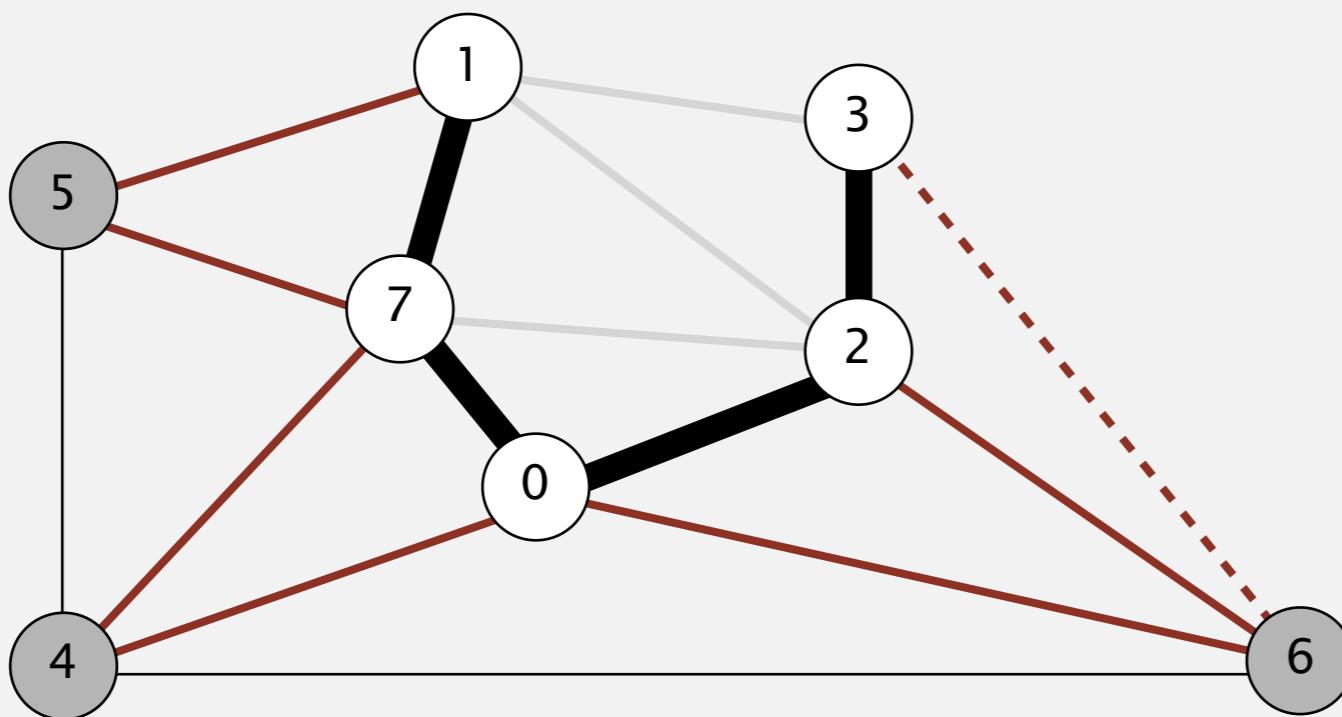
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

add to PQ all edges incident to 3



MST edges

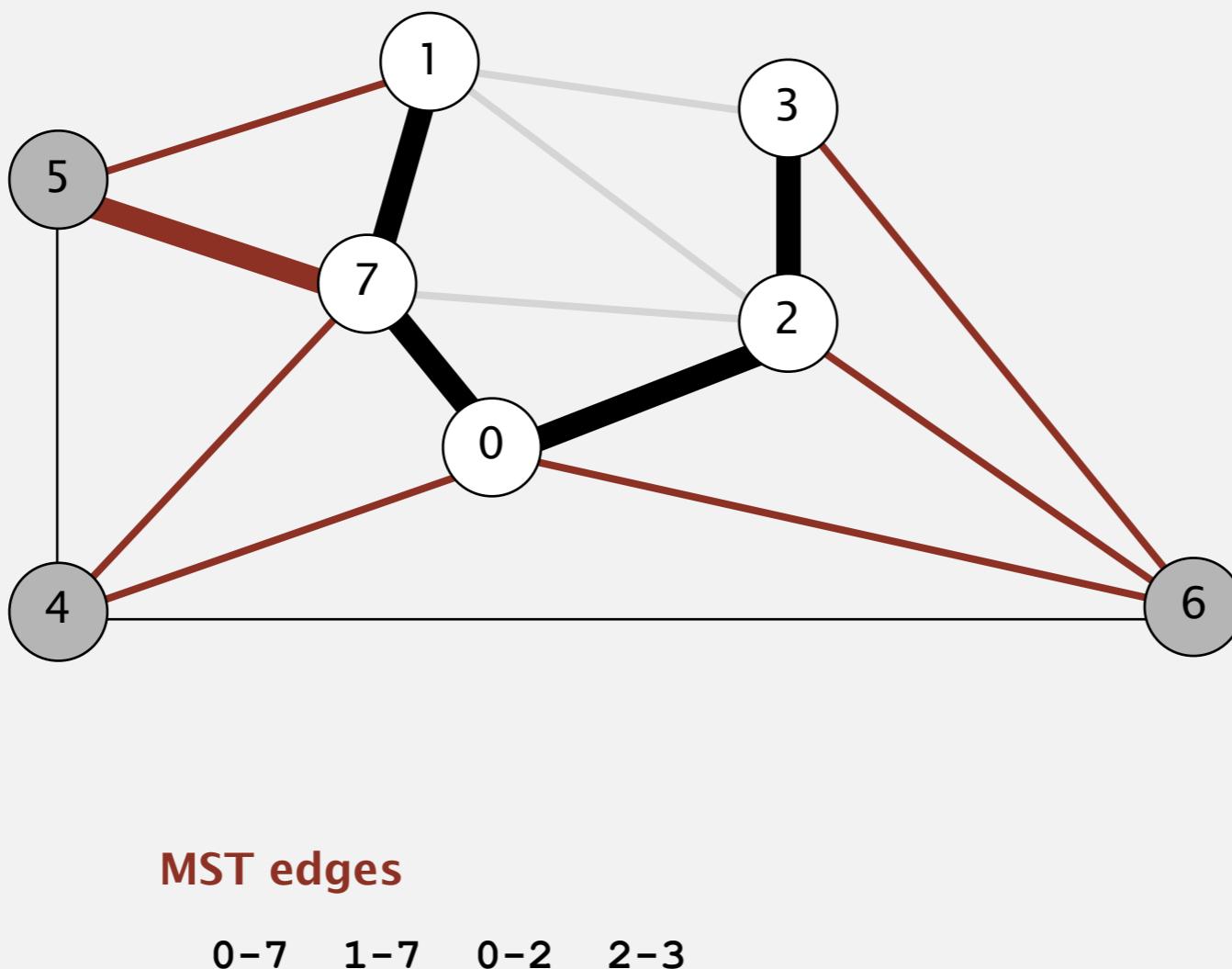
0-7 1-7 0-2 2-3

edges on PQ (sorted by weight)	
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
* 3-6	0.52
6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

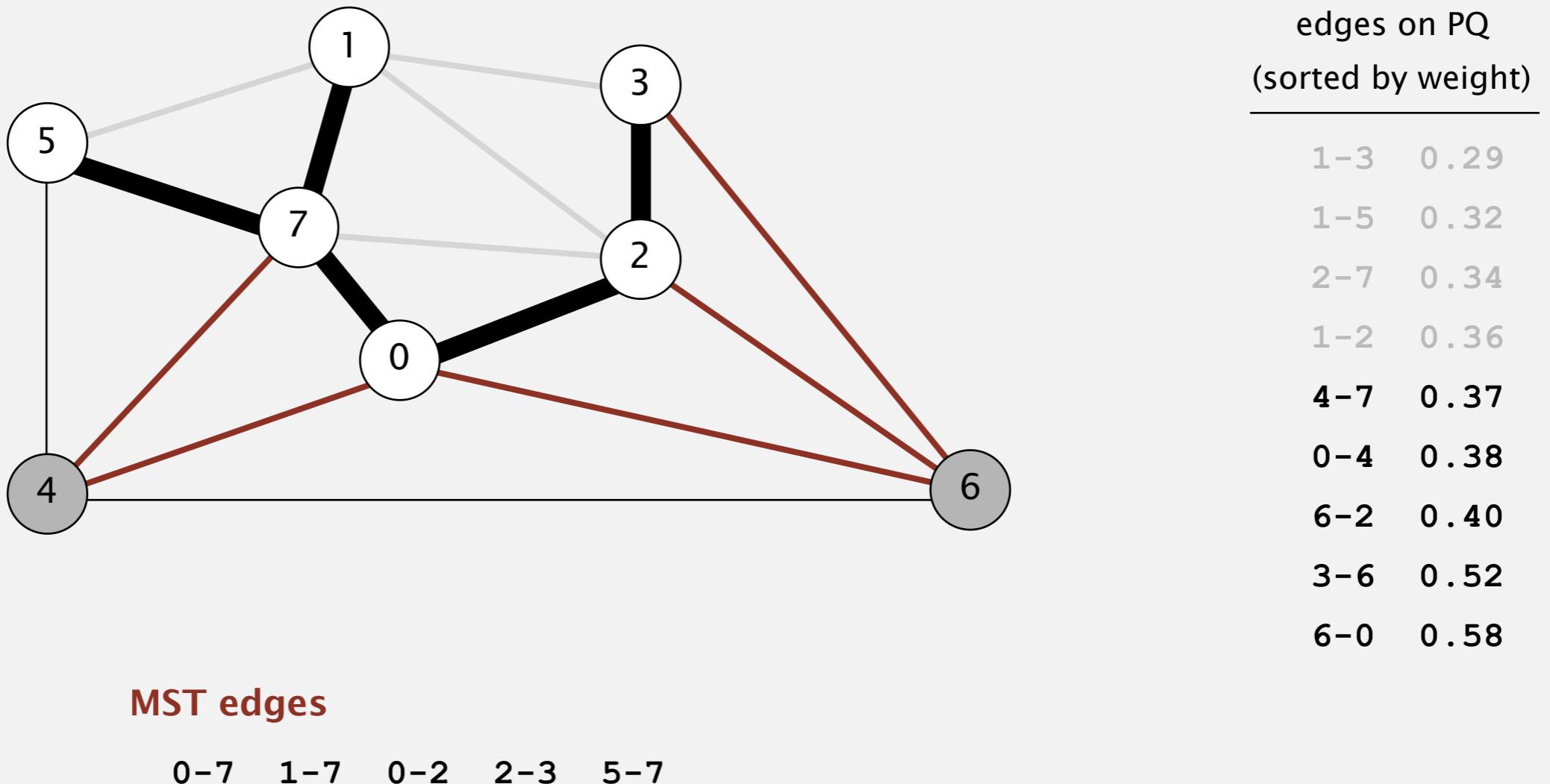
delete 5-7 and add to MST



edges on PQ (sorted by weight)	
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

Prim's algorithm - Lazy implementation

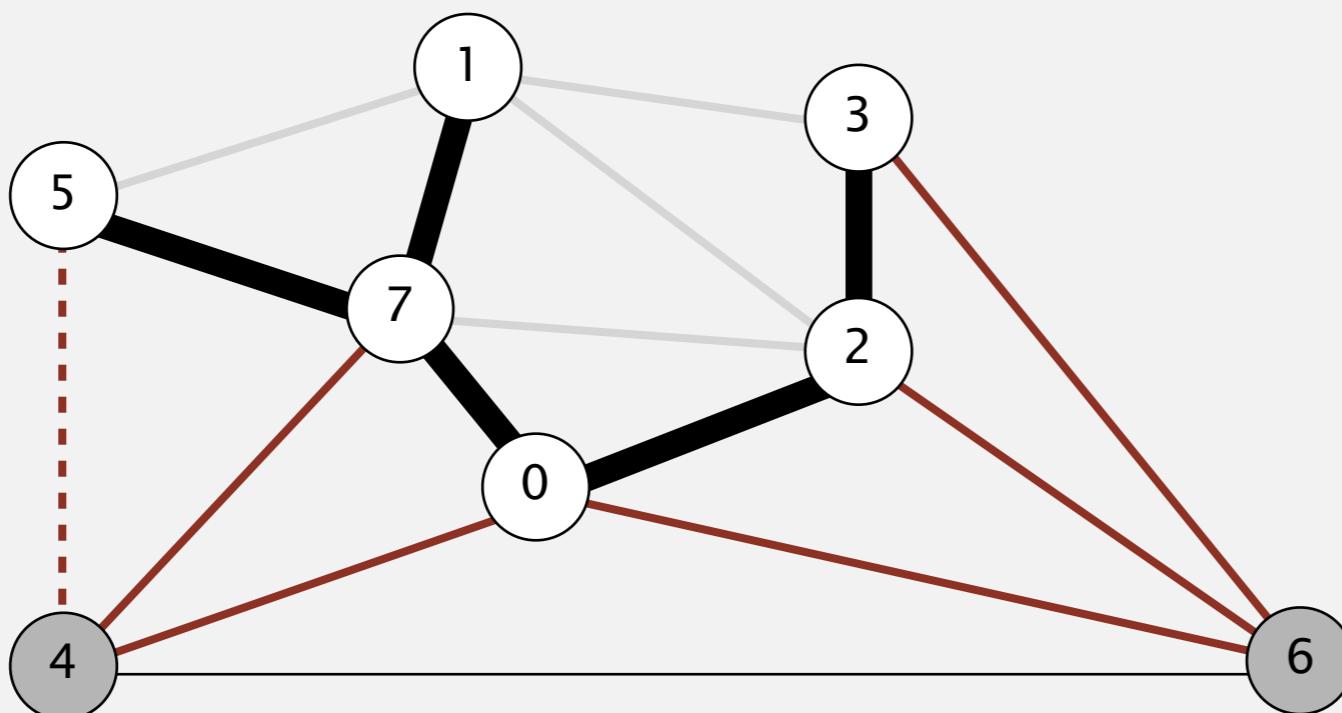
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

add to PQ all edges incident to 5



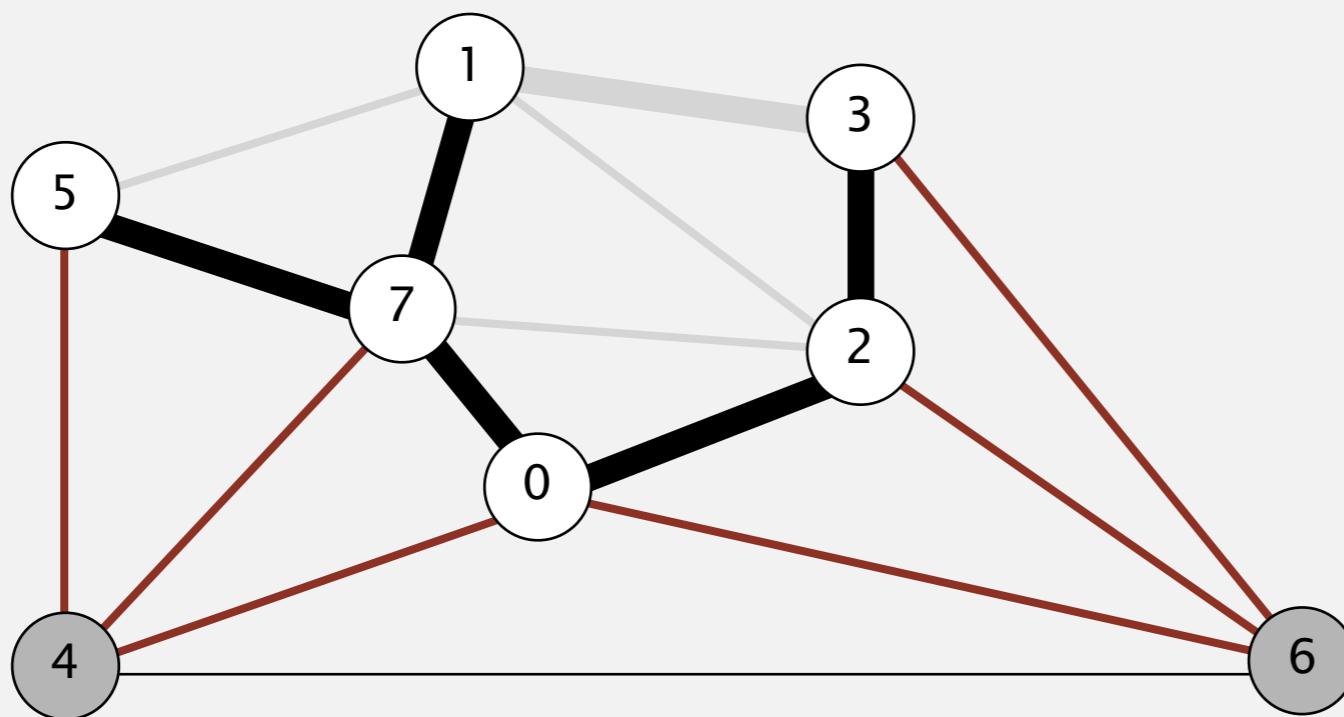
0-7 1-7 0-2 2-3 5-7

edges on PQ (sorted by weight)	
1-3	0.29
1-5	0.32
2-7	0.34
* 4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 1-3 and discard obsolete edge

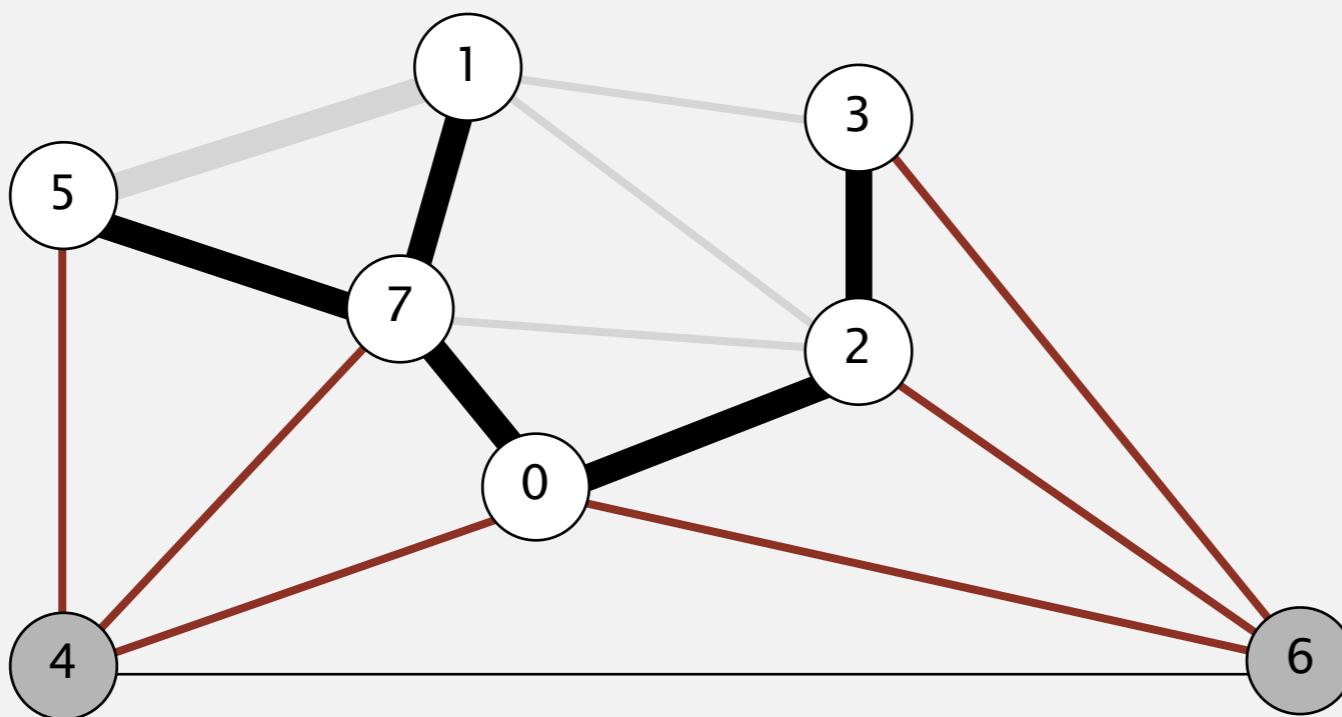


edges on PQ (sorted by weight)	
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 1-5 and discard obsolete edge



MST edges

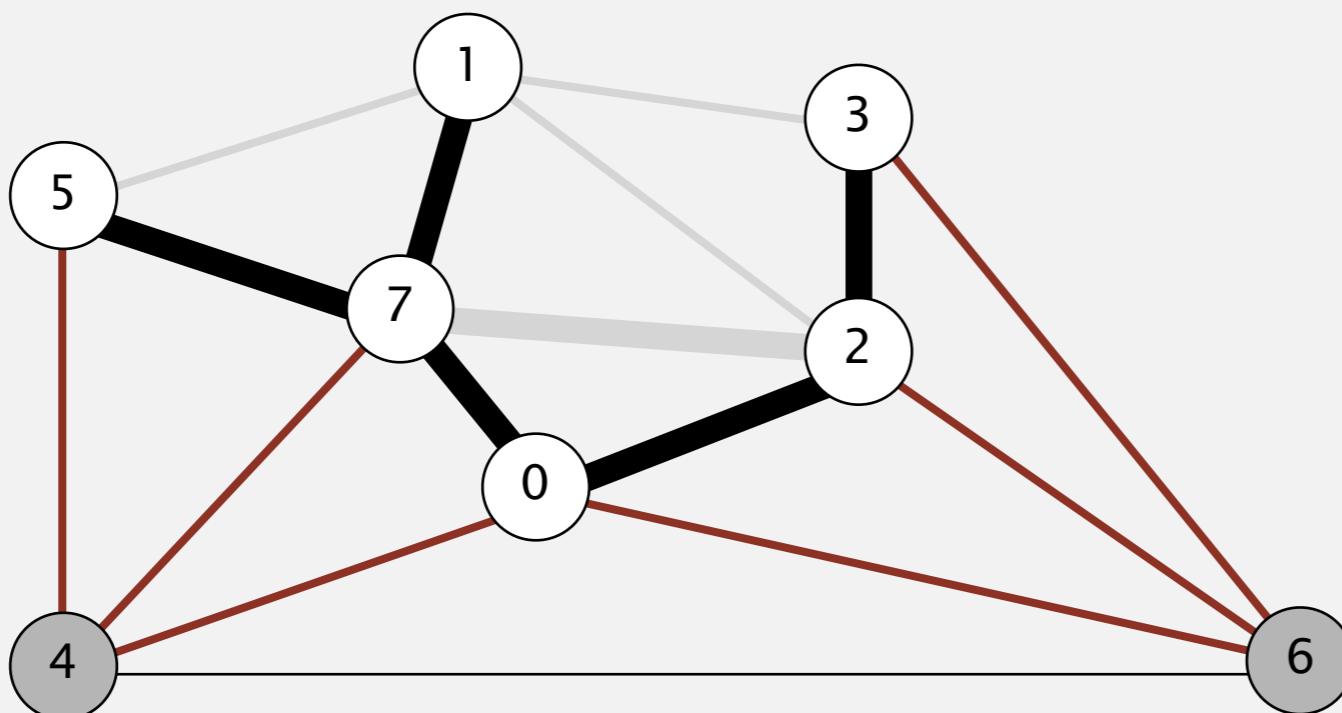
0-7 1-7 0-2 2-3 5-7

edges on PQ (sorted by weight)	
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 2-7 and discard obsolete edge



edges on PQ (sorted by weight)	
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

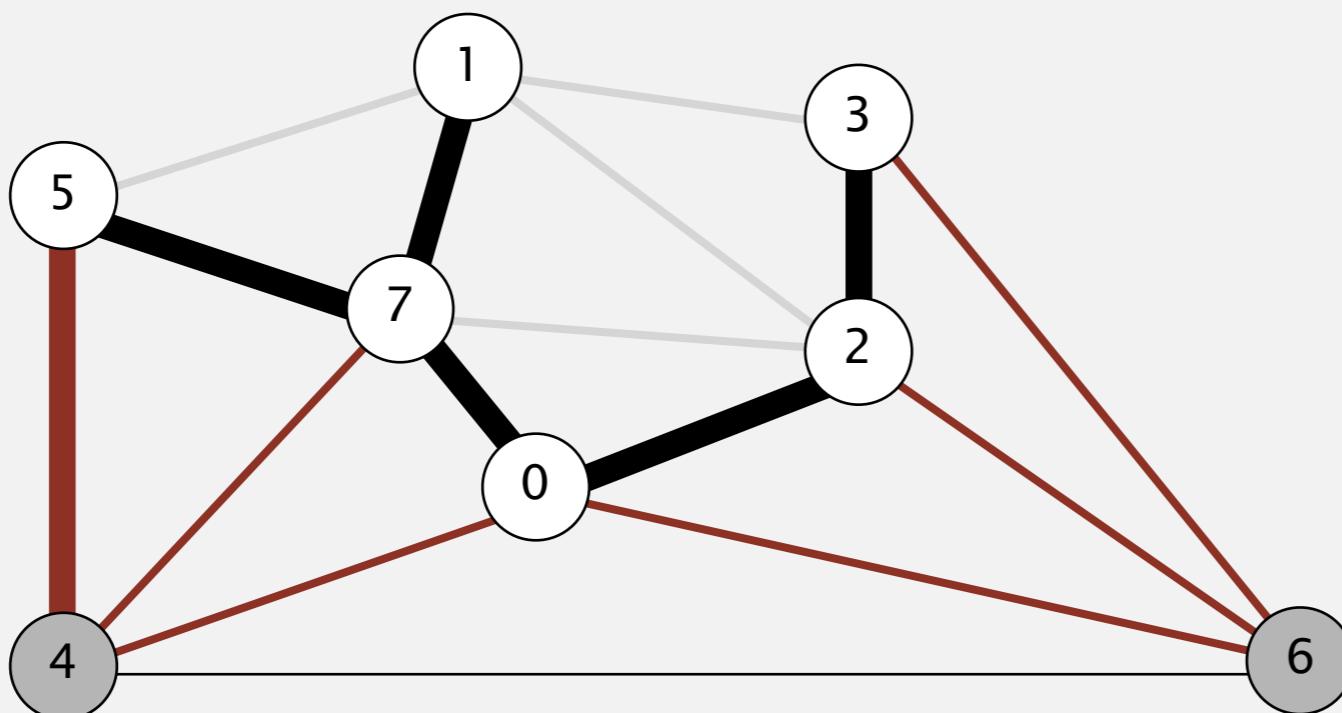
MST edges

0-7 1-7 0-2 2-3 5-7

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

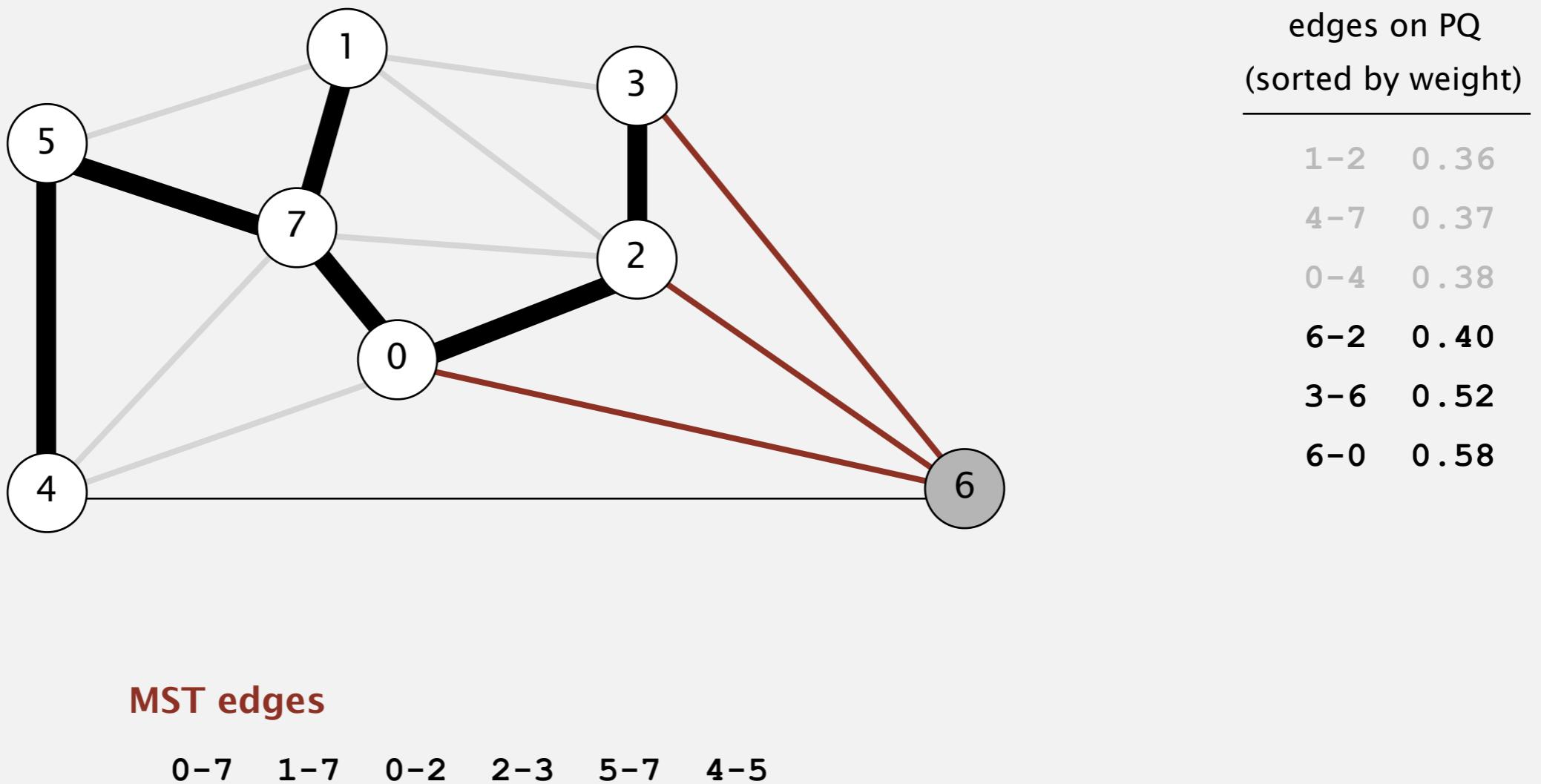
delete 4-5 and add to MST



edges on PQ (sorted by weight)	
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58

Prim's algorithm - Lazy implementation

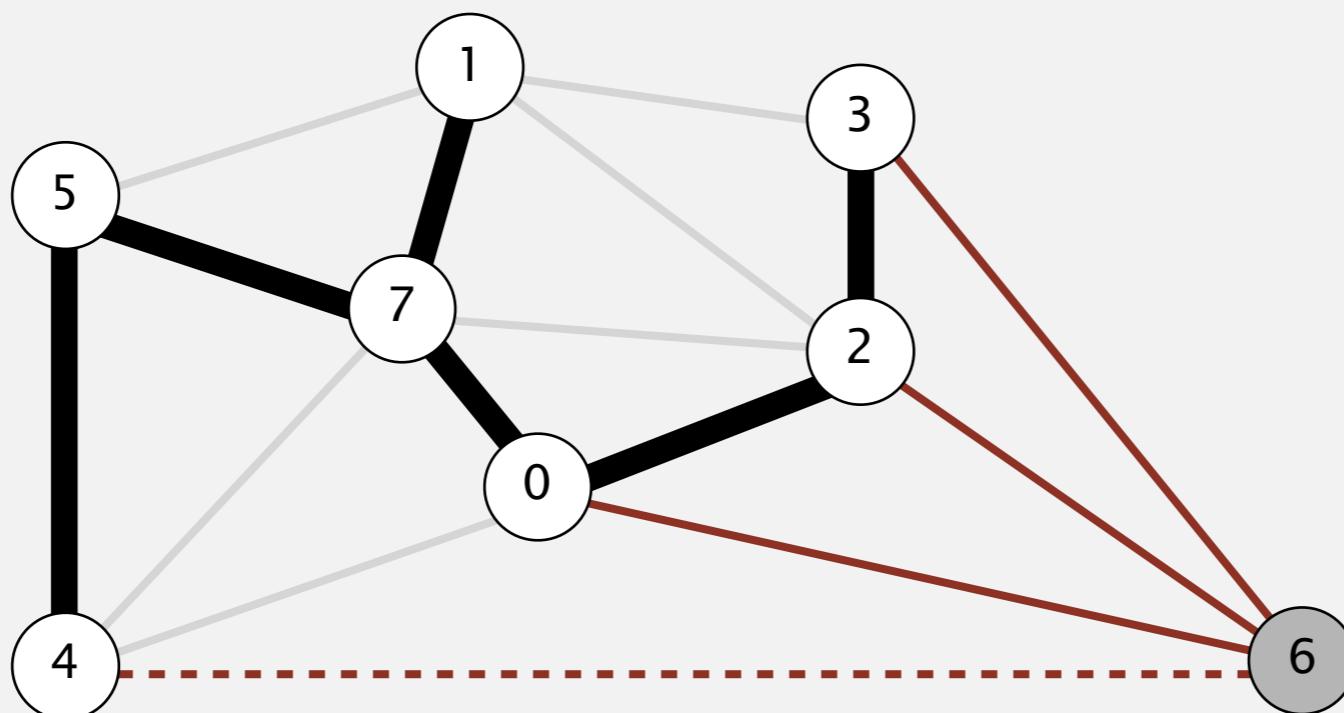
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

add to PQ all edges incident to 4

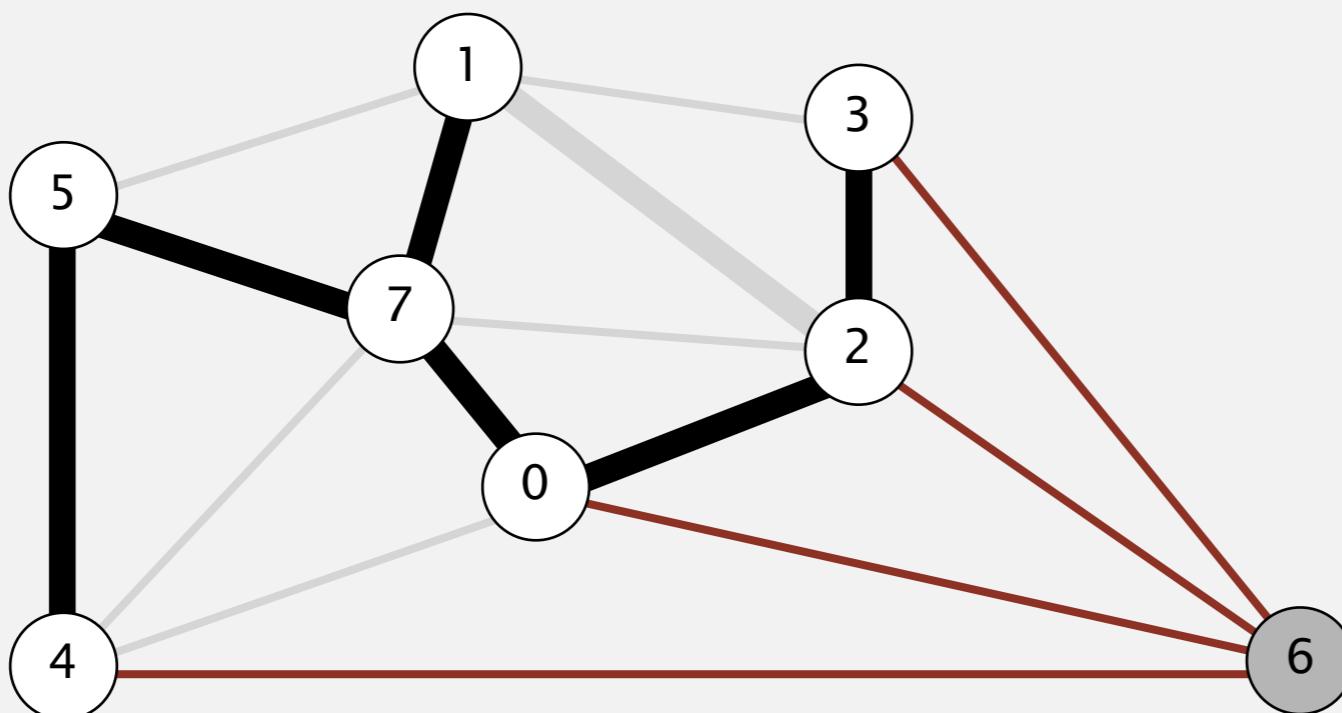


edges on PQ (sorted by weight)	
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
* 6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 1-2 and discard obsolete edge

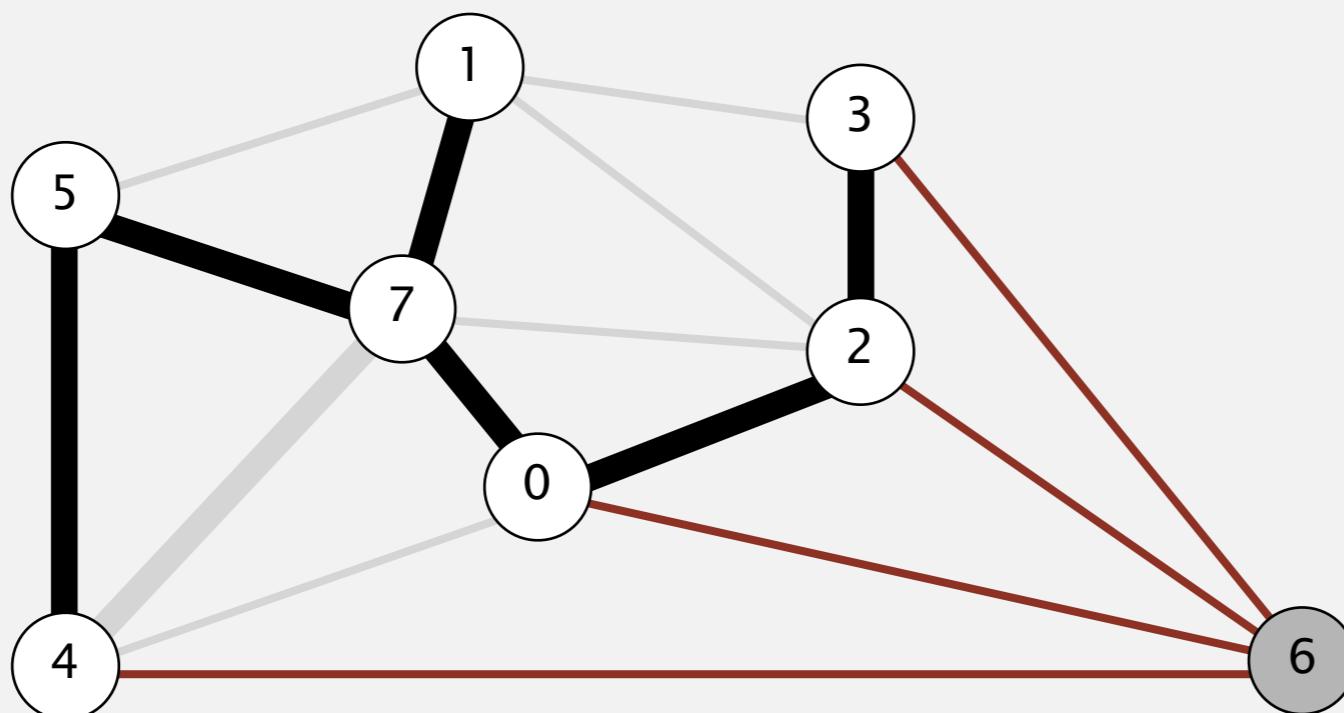


edges on PQ (sorted by weight)	
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 4-7 and discard obsolete edge



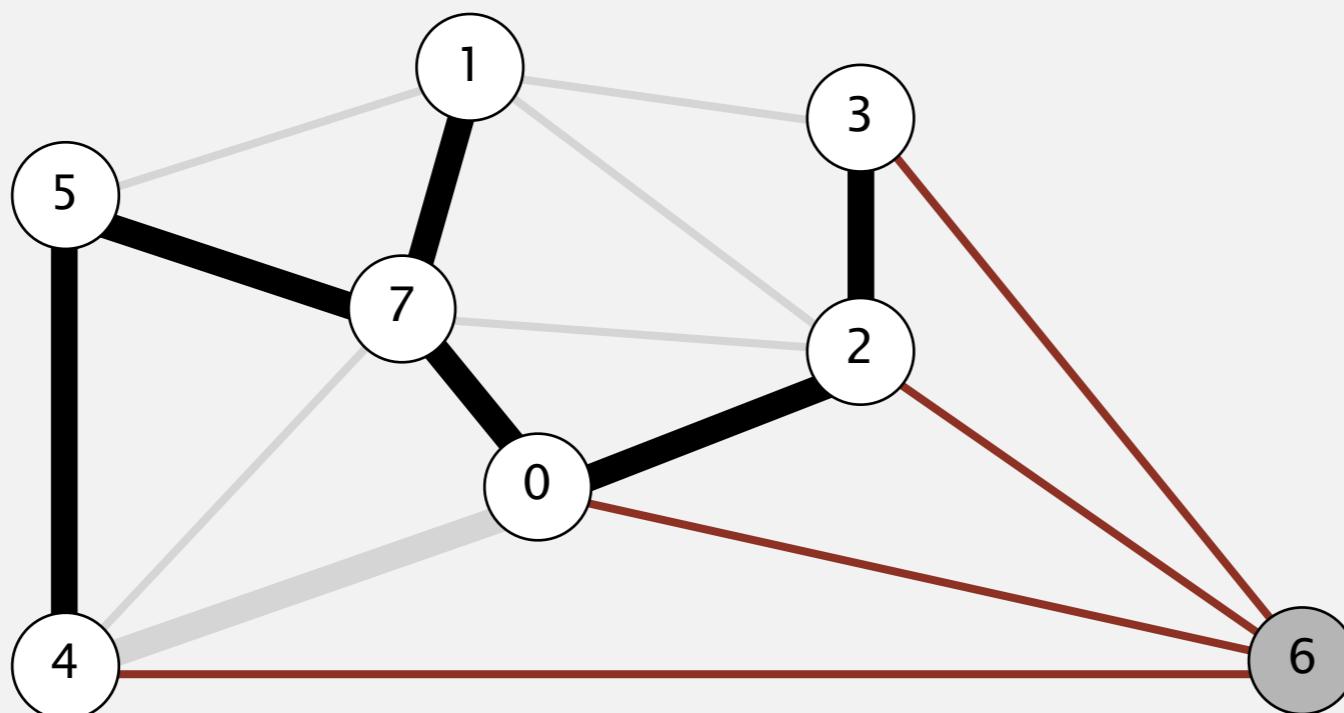
edges on PQ
(sorted by weight)

4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 0-4 and discard obsolete edge

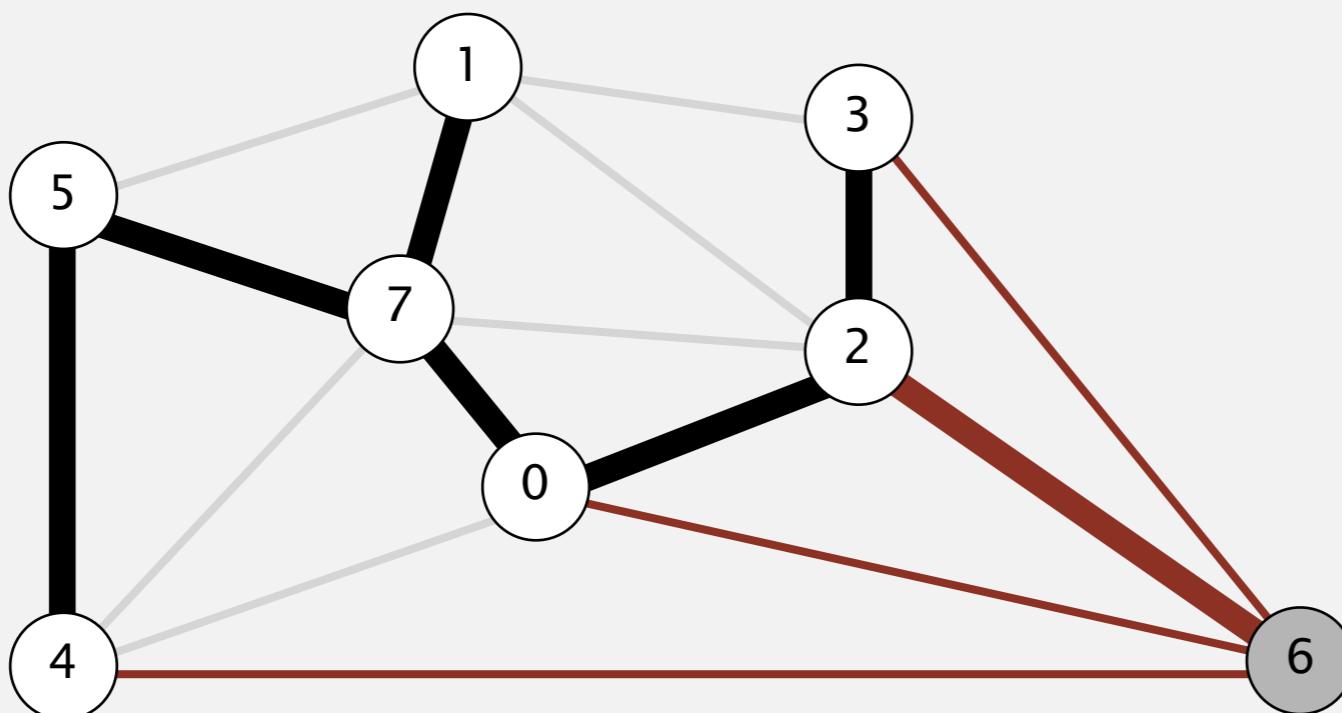


edges on PQ (sorted by weight)	
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 6-2 and add to MST

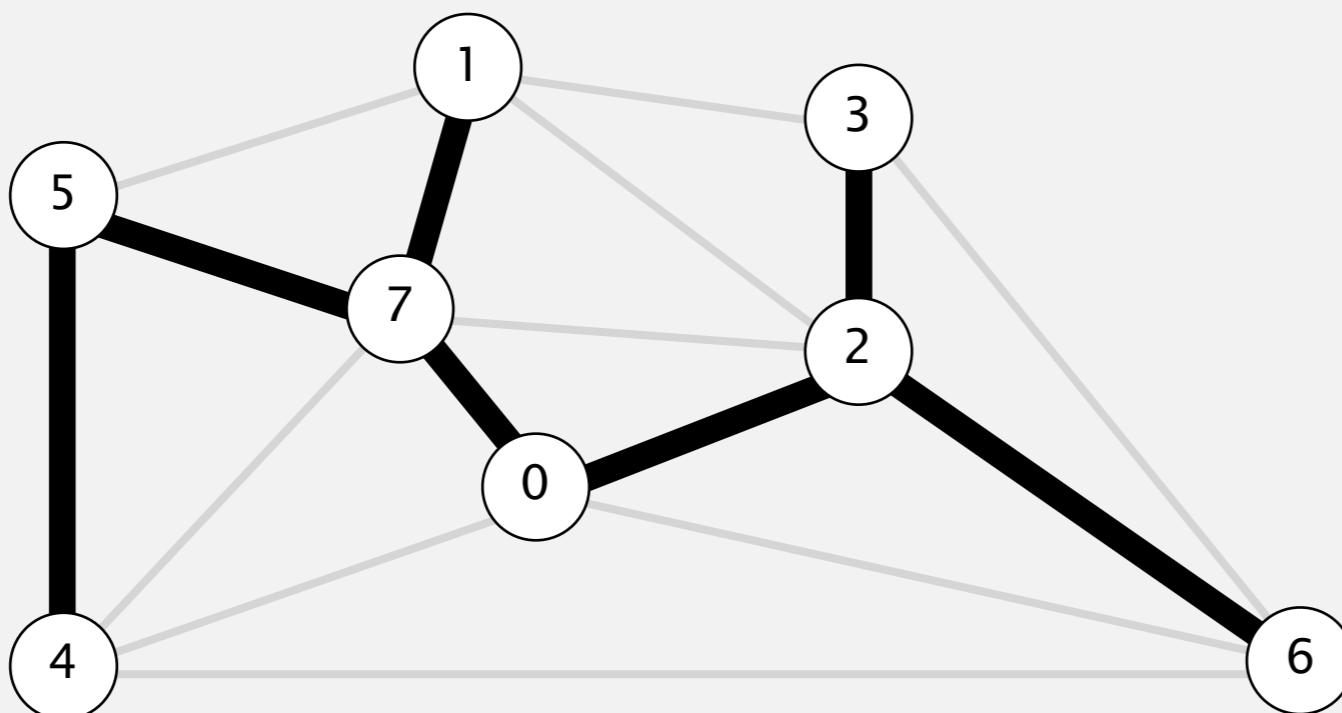


edges on PQ (sorted by weight)	
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

delete 6-2 and add to MST

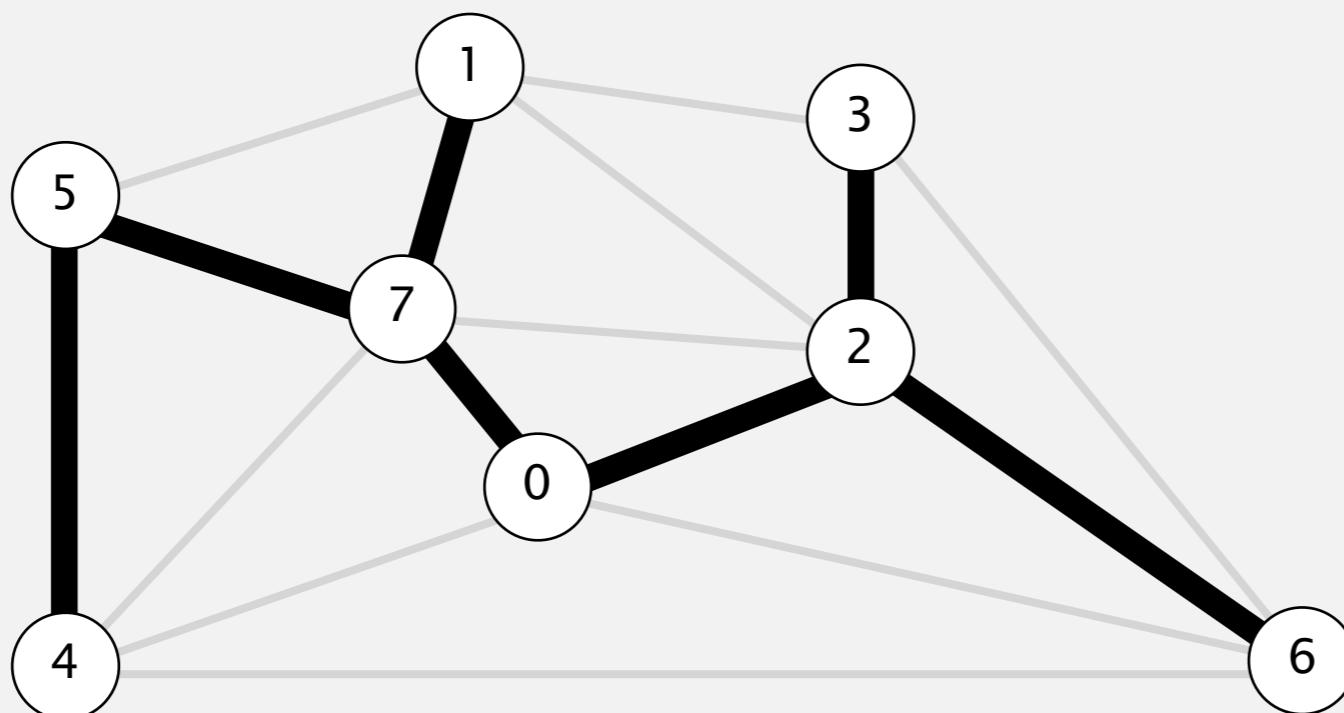


edges on PQ (sorted by weight)	
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

stop since $V-1$ edges

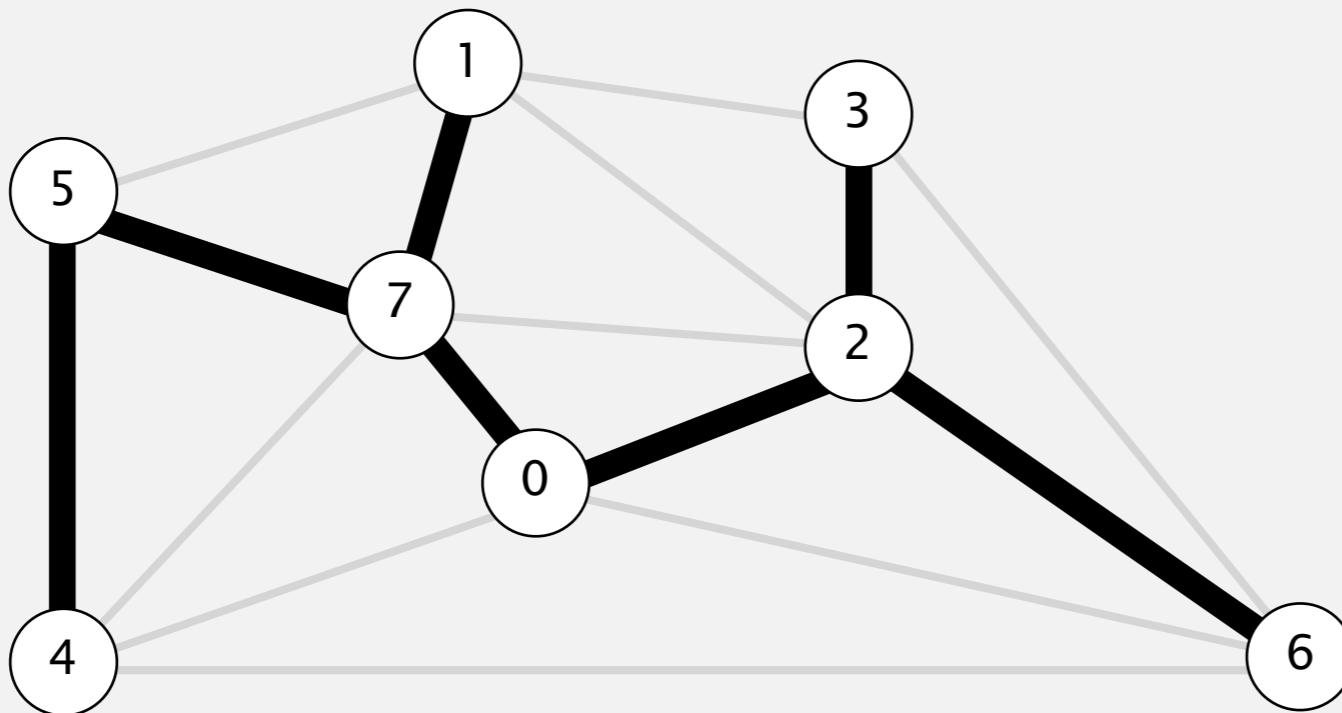


edges on PQ
(sorted by weight)

3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Lazy implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



MST edges

0-7 1-7 0-2 2-3 5-7 4-5 6-2

Prim's algorithm: lazy implementation

```
public class LazyPrimMST
{
    private boolean[] marked;      // MST vertices
    private Queue<Edge> mst;     // MST edges
    private MinPQ<Edge> pq;       // PQ of edges

    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        visit(G, 0);
    }

    while (!pq.isEmpty())
    {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (marked[v] && marked[w]) continue;
        mst.enqueue(e);
        if (!marked[v]) visit(G, v);
        if (!marked[w]) visit(G, w);
    }
}
```

assume G is connected

repeatedly delete the min weight edge $e = v-w$ from PQ

ignore if both endpoints in T

add edge e to tree

add v or w to tree

Prim's algorithm: lazy implementation

```
private void visit(WeightedGraph G, int v)
{
    marked[v] = true;
    for (Edge e : G.adj(v))
        if (!marked[e.other(v)])
            pq.insert(e);
}

public Iterable<Edge> mst()
{   return mst; }
```

- add v to T
- for each edge $e = v-w$, add to PQ if w not already in T

Lazy Prim's algorithm: running time

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Pf.

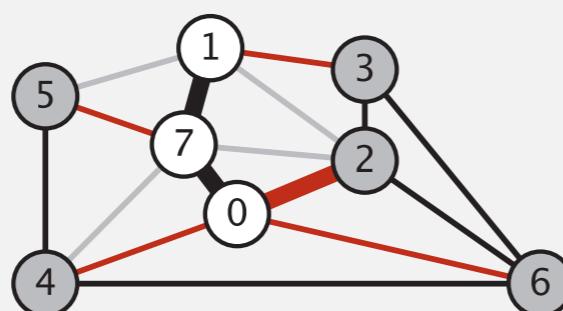
operation	frequency	binary heap
delete min	E	$\log E$
insert	E	$\log E$

Prim's algorithm: eager implementation

Challenge. Find min weight edge with exactly one endpoint in T .

Eager solution. Maintain a PQ of vertices connected by an edge to T , where priority of vertex v = weight of shortest edge connecting v to T .

- Delete min vertex v and add its associated edge $e = v-w$ to T .
- Update PQ by considering all edges $e = v-x$ incident to v
 - ignore if x is already in T
 - add x to PQ if not already on it
 - decrease priority of x if $v-x$ becomes shortest edge connecting x to T



pq has at most one entry per vertex

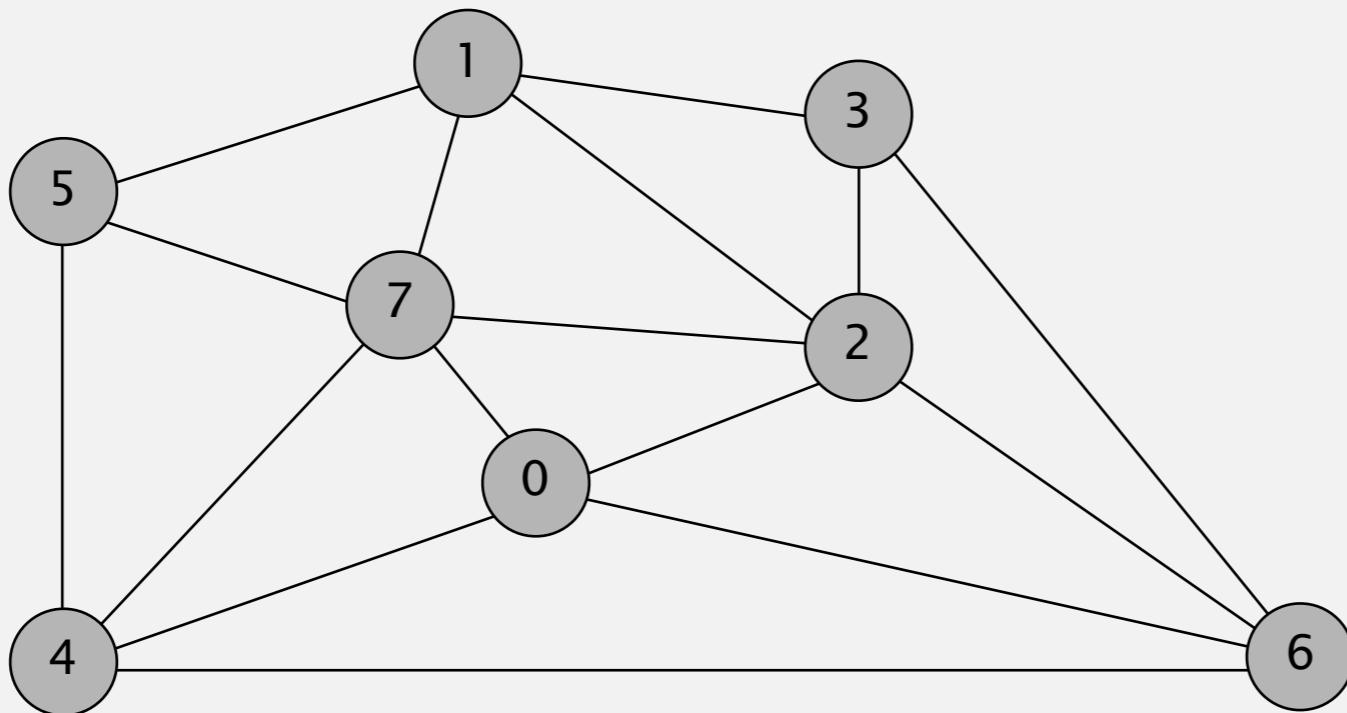
0		
1	1-7	0.19
2	0-2	0.26
3	1-3	0.29
4	0-4	0.38
5	5-7	0.28
6	6-0	0.58
7	0-7	0.16

red: on PQ

black: on MST

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

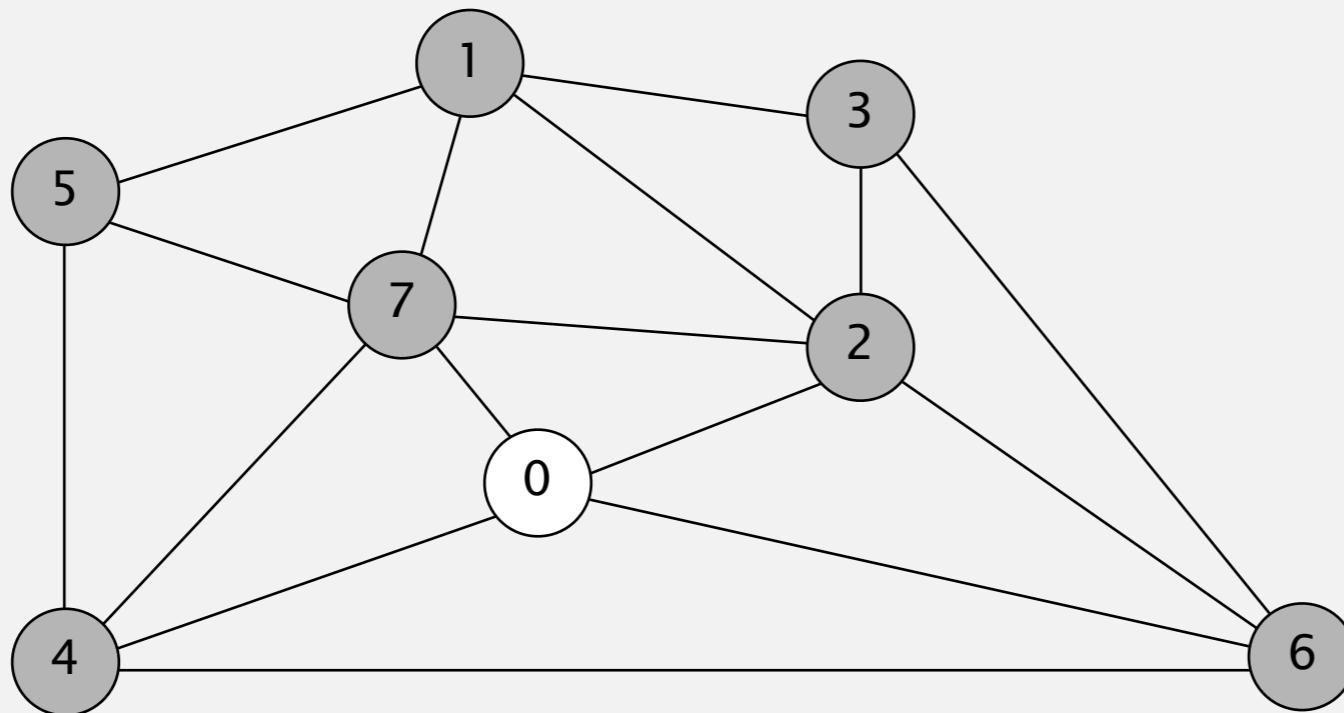


an edge-weighted graph

0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2-7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

Prim's algorithm - Eager implementation

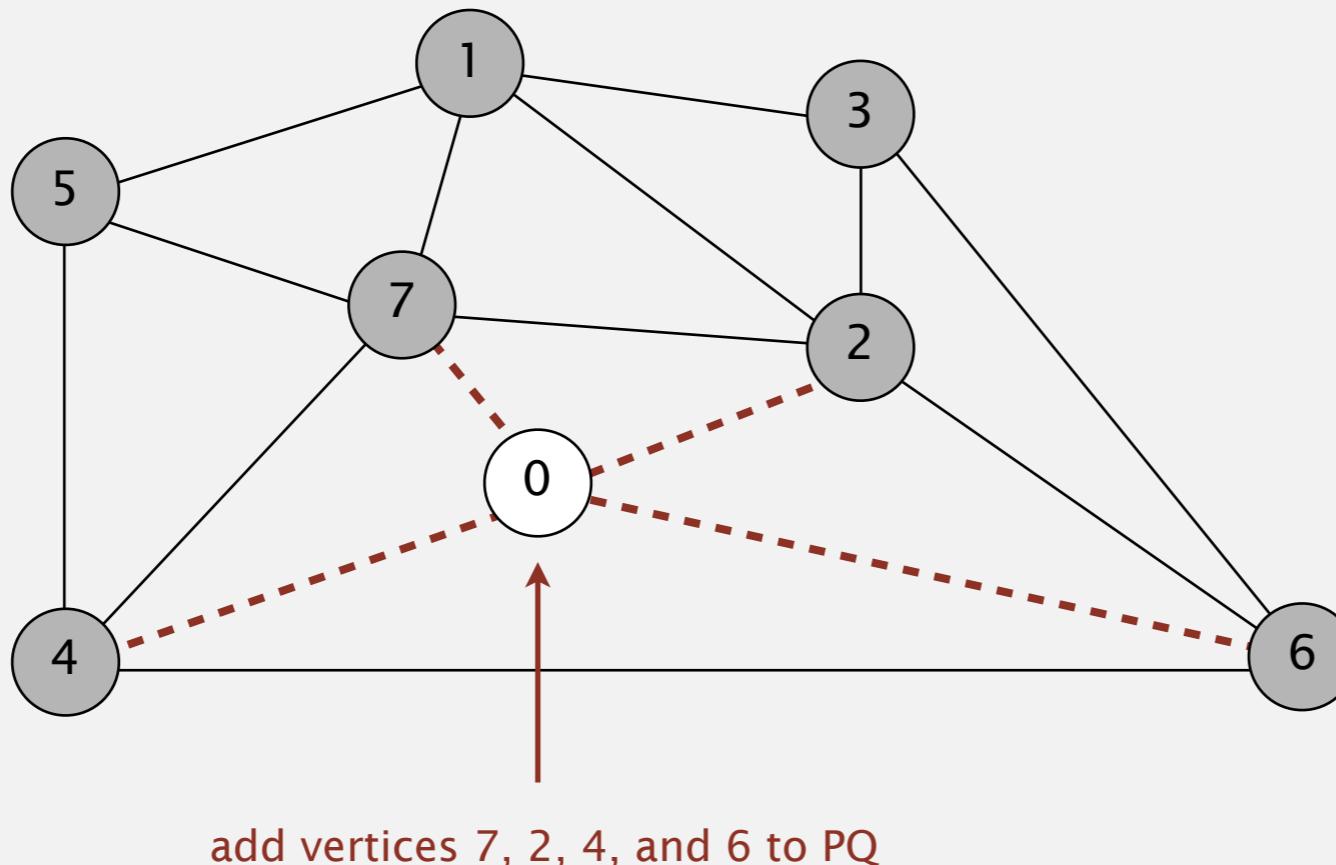
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo[]	distTo[]
0	-	-

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

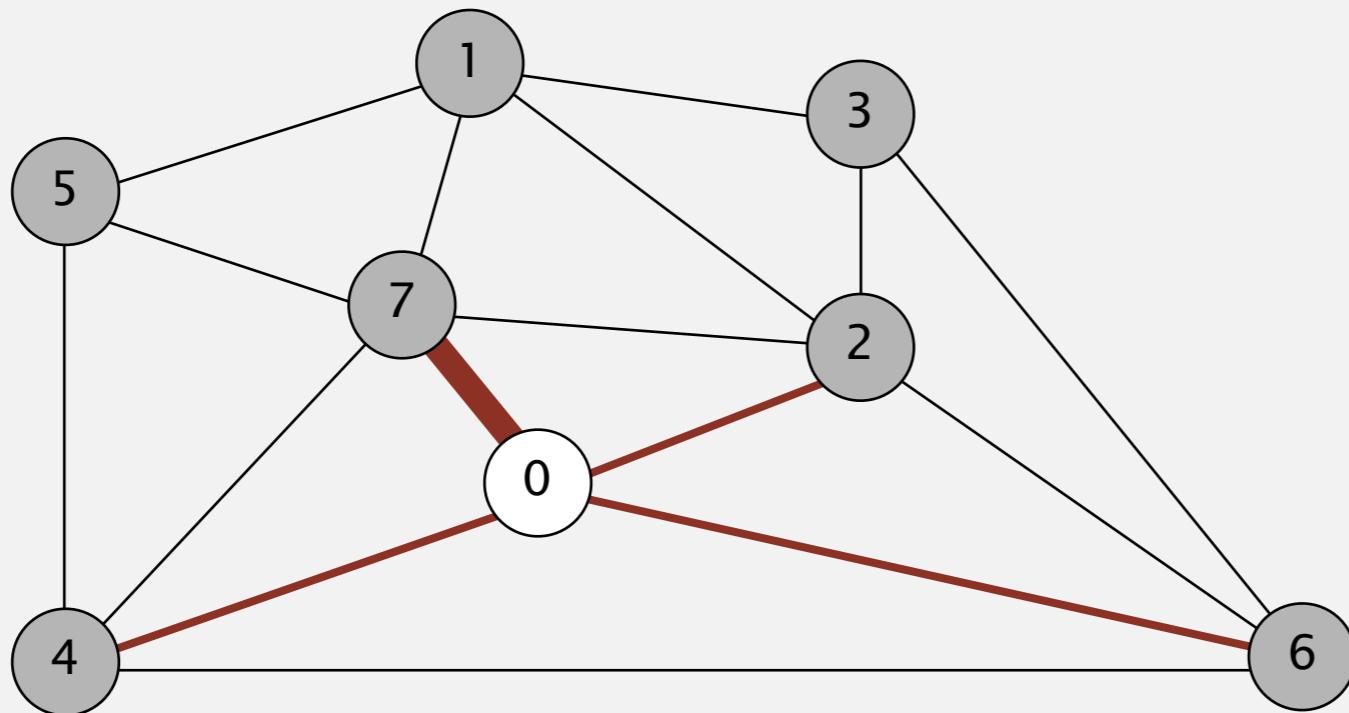


v	edgeTo []	distTo []
0	-	-
7	0-7	0.16
2	0-2	0.26
4	0-4	0.38
6	6-0	0.58

vertices on PQ
(sorted by weight)

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

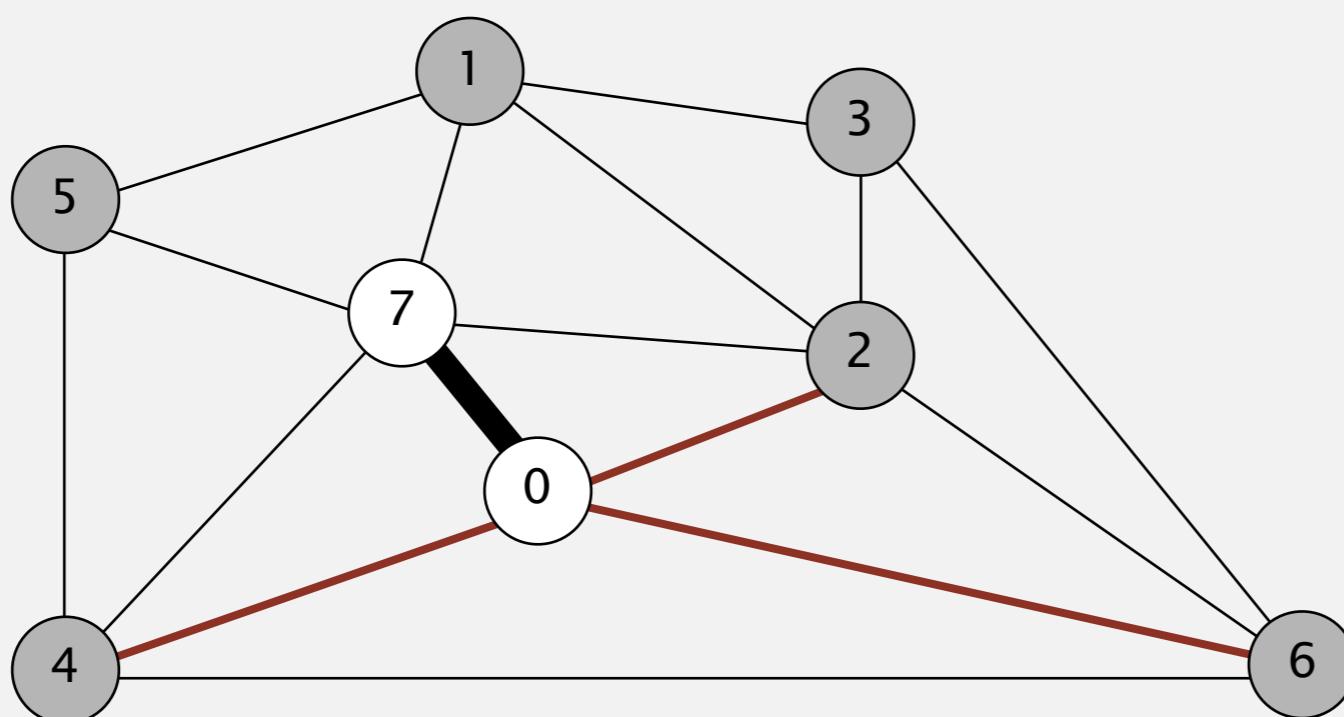


v	edgeTo []	distTo []
0	-	-
7	0-7	0.16
2	0-2	0.26
4	0-4	0.38
6	6-0	0.58

vertices on PQ
(sorted by weight)

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

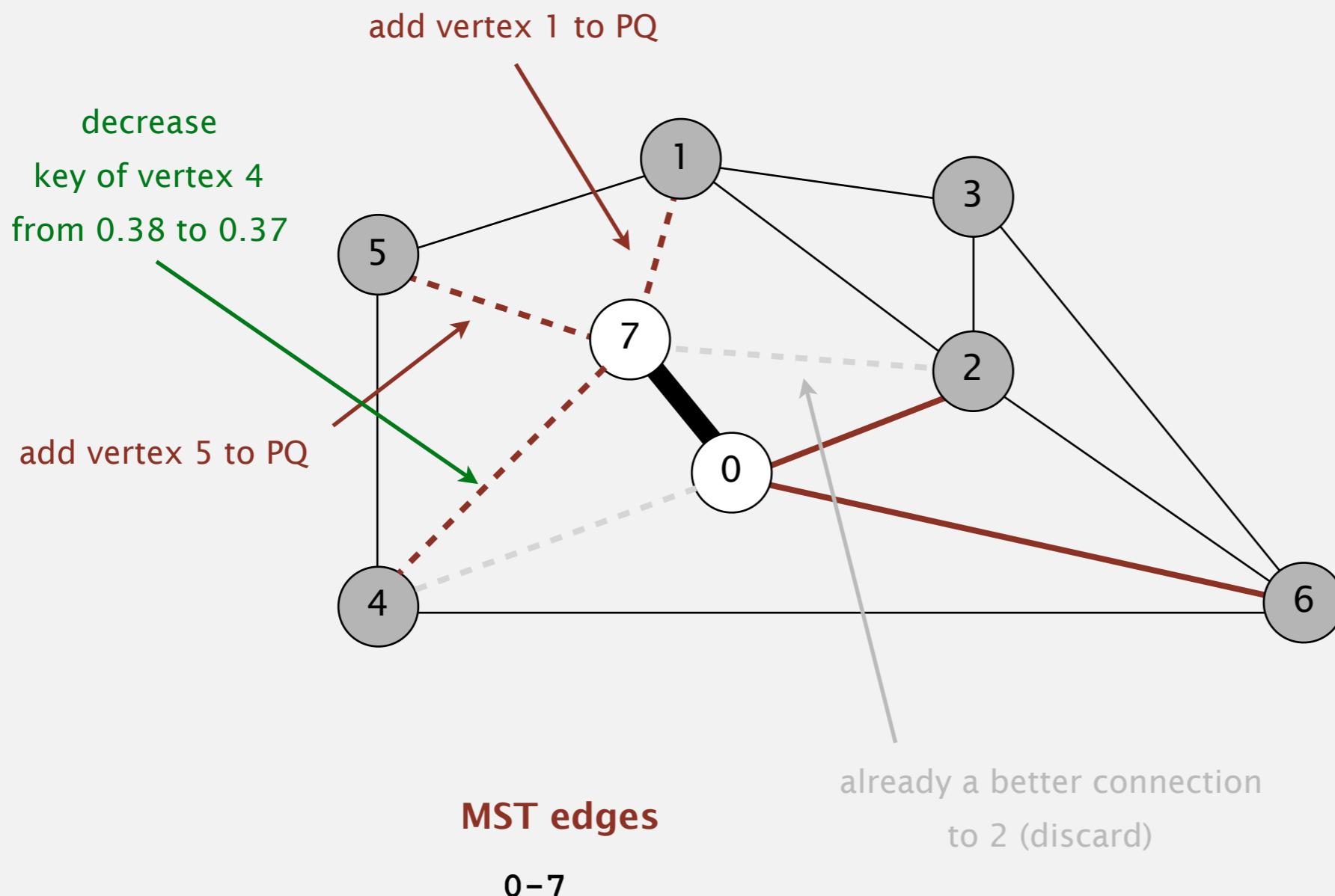


v	edgeTo []	distTo []
0	-	-
→ 7	0-7	0.16
2	0-2	0.26
4	0-4	0.38
6	6-0	0.58

vertices on PQ
(sorted by weight)

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

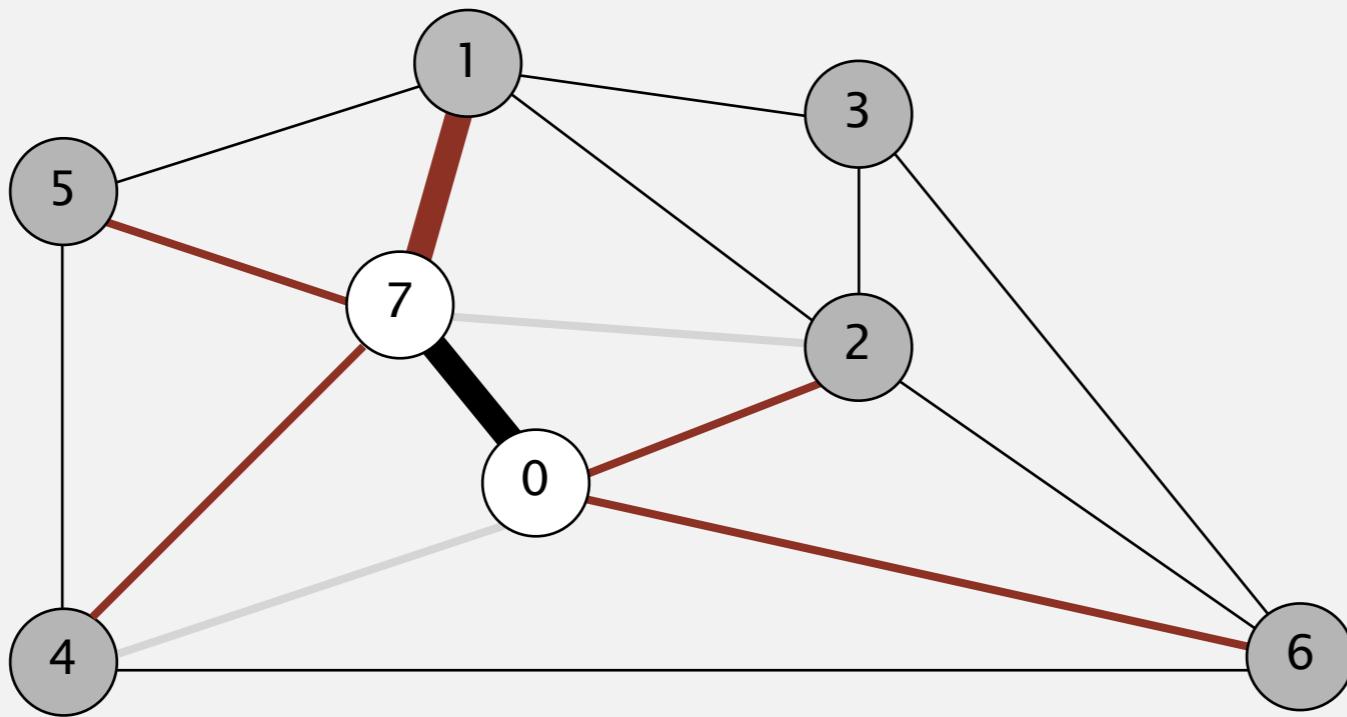


v	edgeTo[]	distTo[]
0	-	-
→ 7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
5	5-7	0.28
4	4-7	0.37
0-4	0.38	
6	6-0	0.58

**vertices on PQ
(sorted by weight)**

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

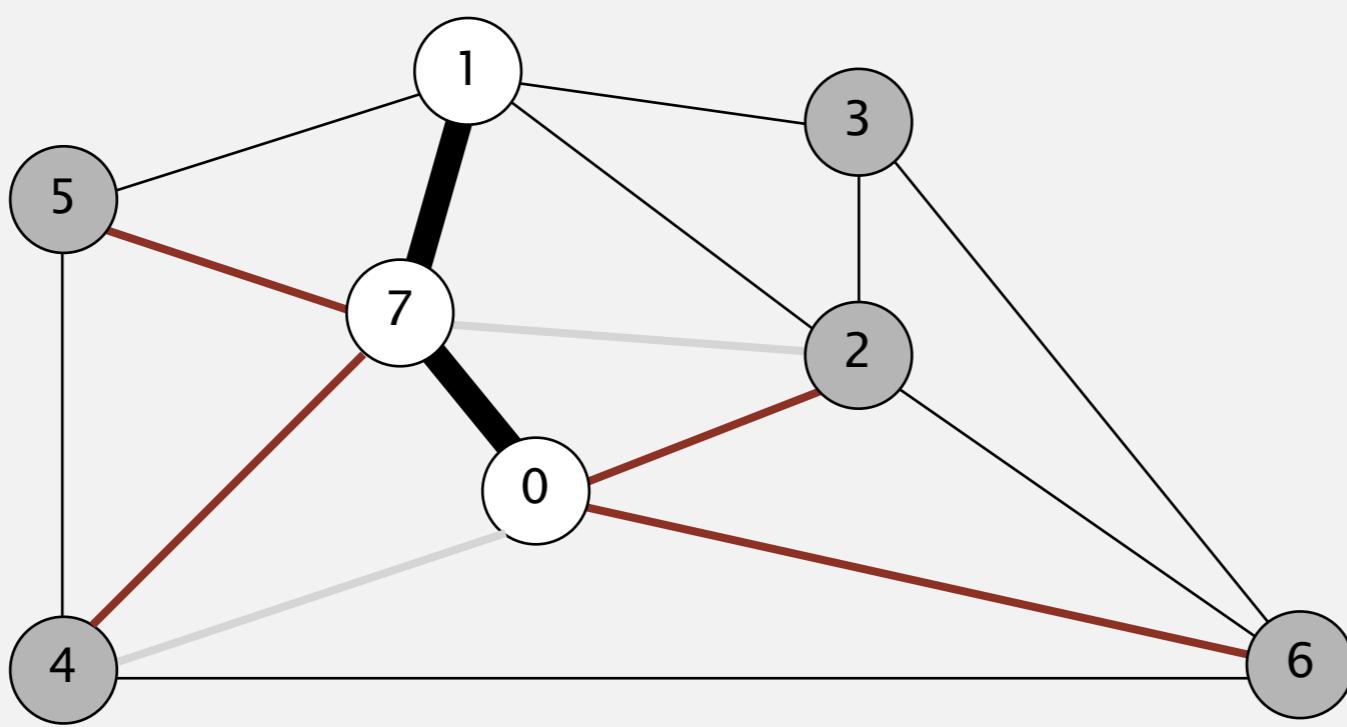


v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
5	5-7	0.28
4	4-7	0.37
6	6-0	0.58

vertices on PQ
(sorted by weight)

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



MST edges

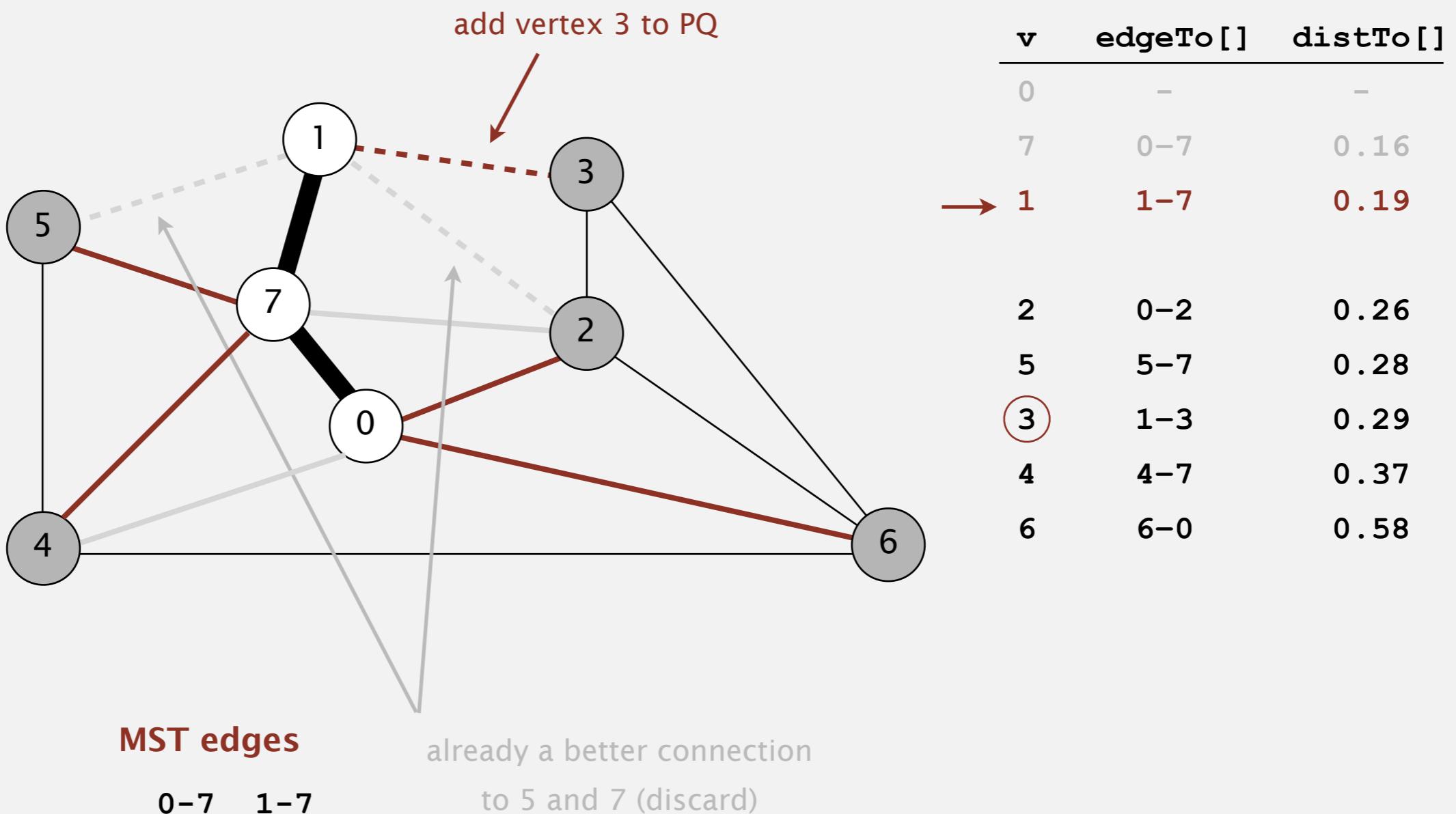
0-7 1-7

v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
→ 1	1-7	0.19
2	0-2	0.26
5	5-7	0.28
4	4-7	0.37
6	6-0	0.58

vertices on PQ
(sorted by weight)

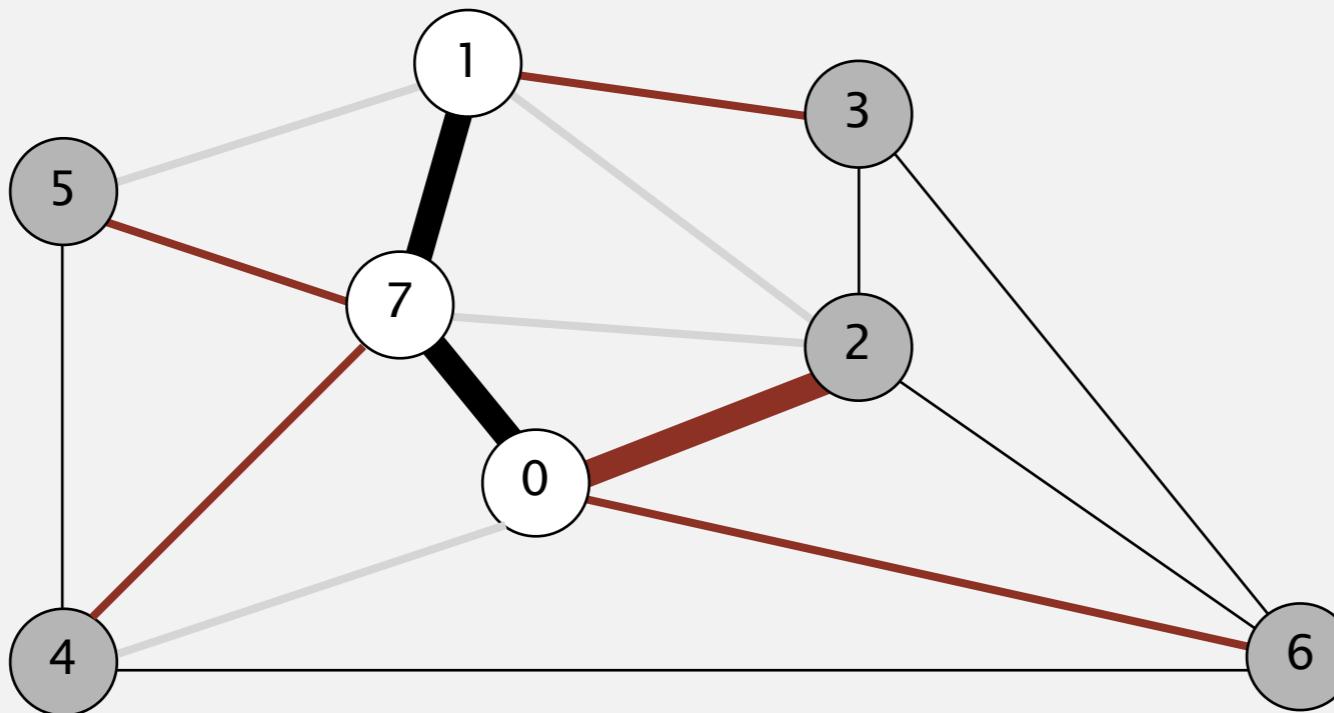
Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



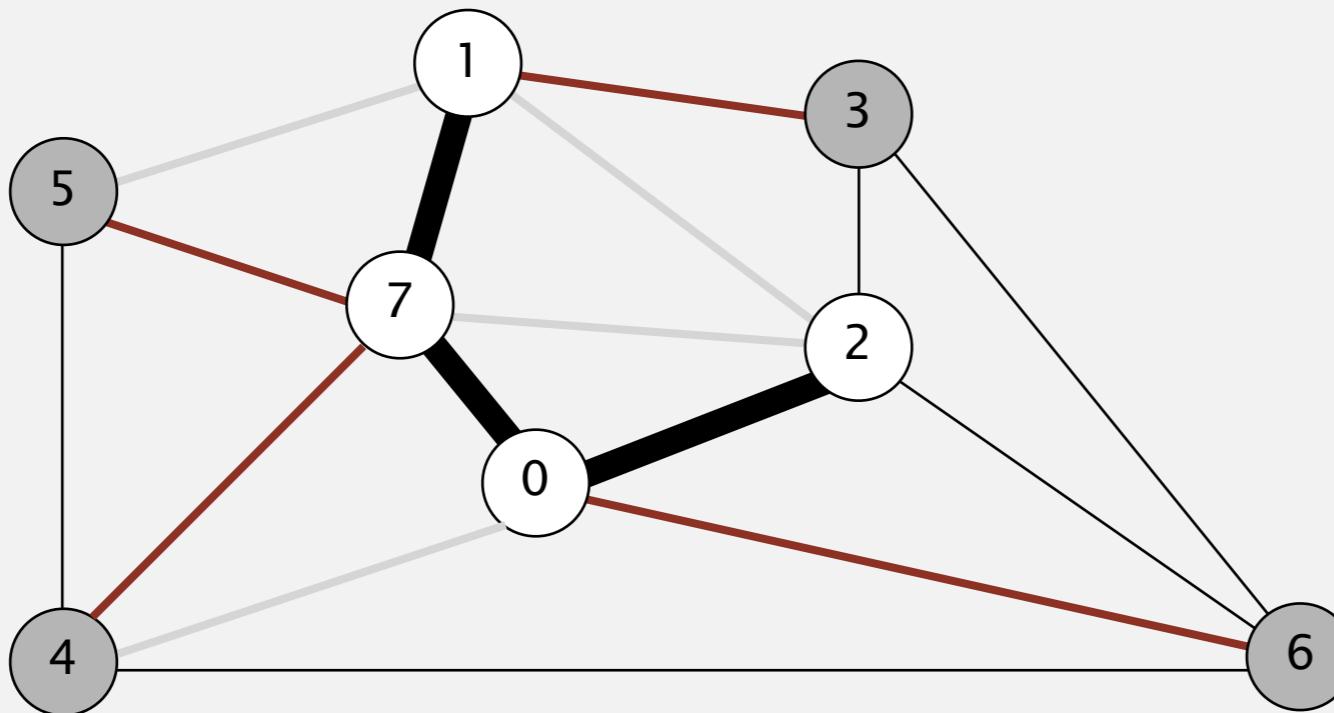
MST edges

0-7 1-7

v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
5	5-7	0.28
3	1-3	0.29
4	4-7	0.37
6	6-0	0.58

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



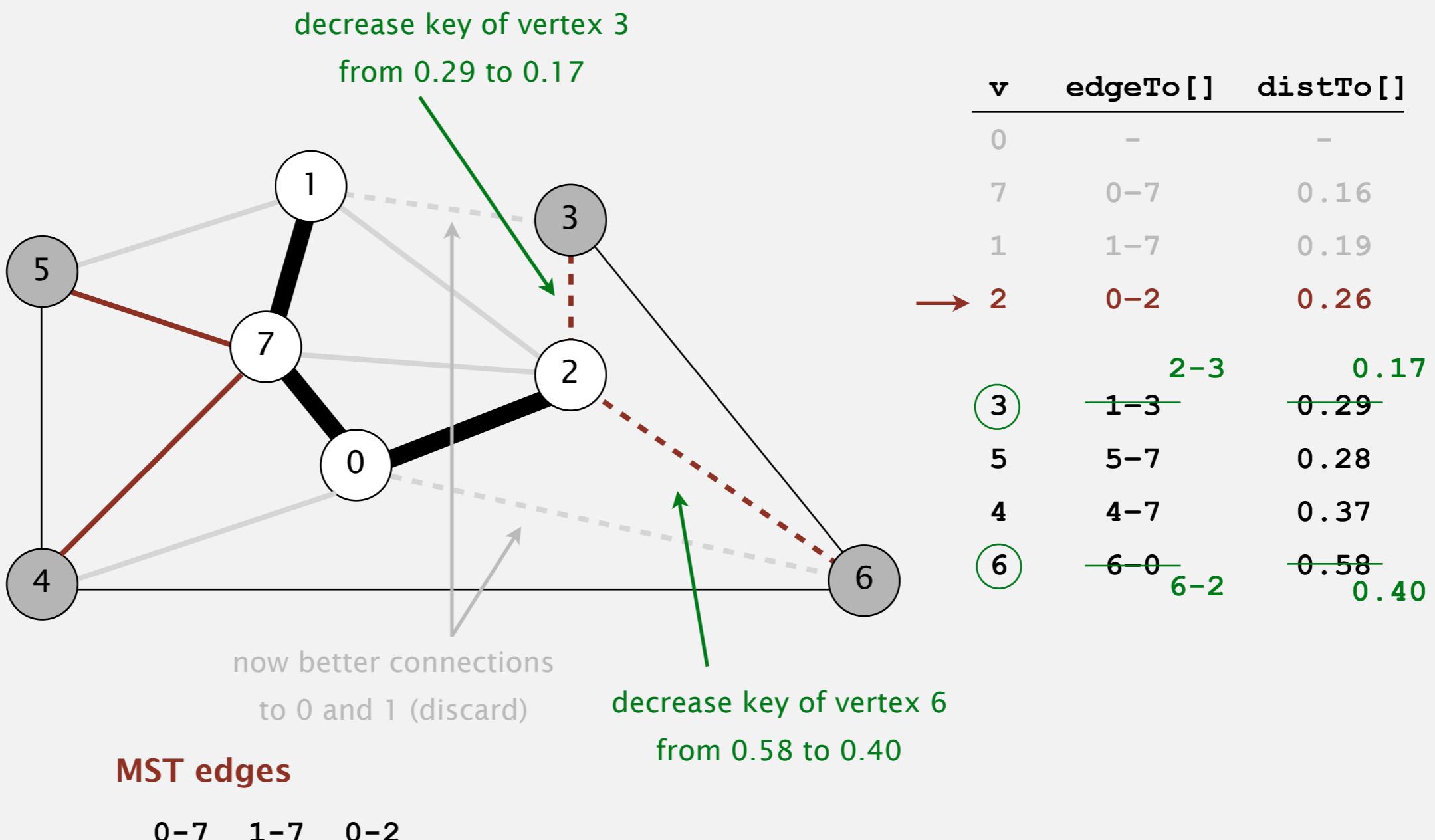
MST edges

0-7 1-7 0-2

v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
5	5-7	0.28
3	1-3	0.29
4	4-7	0.37
6	6-0	0.58

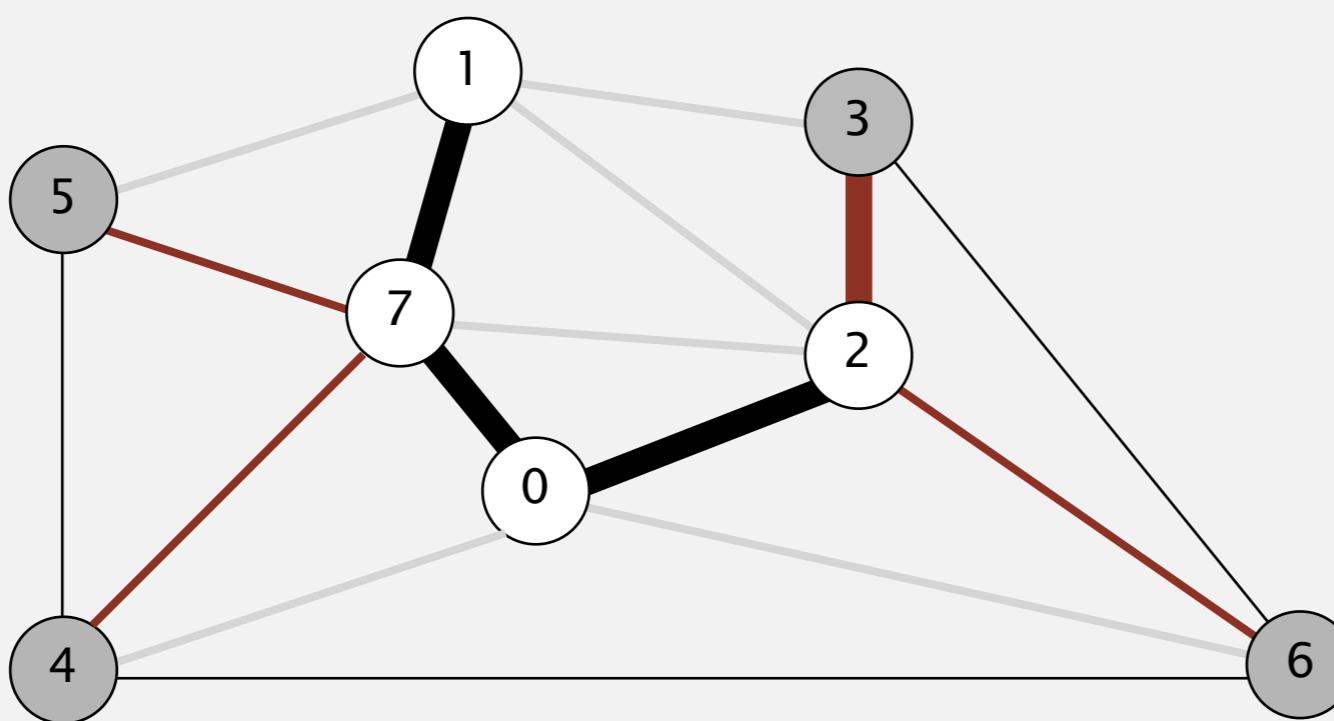
Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Eager implementation

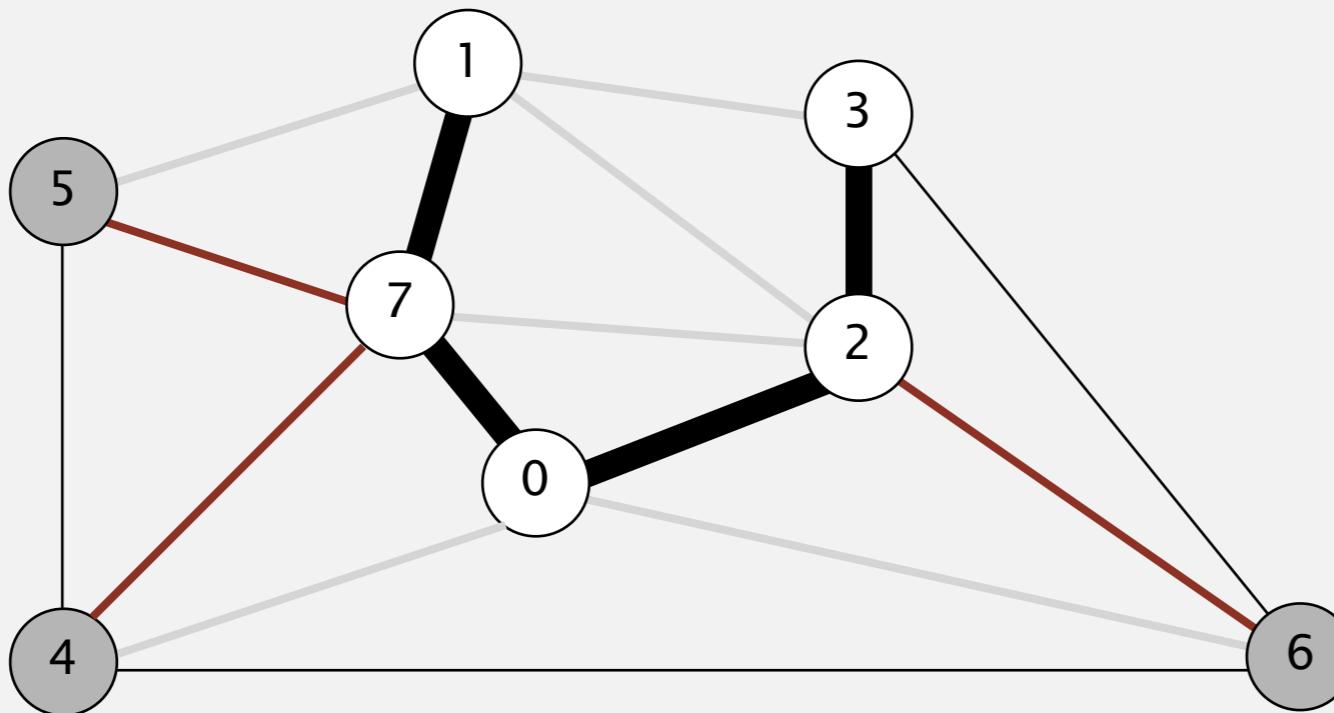
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo []	distTo []
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-7	0.37
6	6-2	0.40

Prim's algorithm - Eager implementation

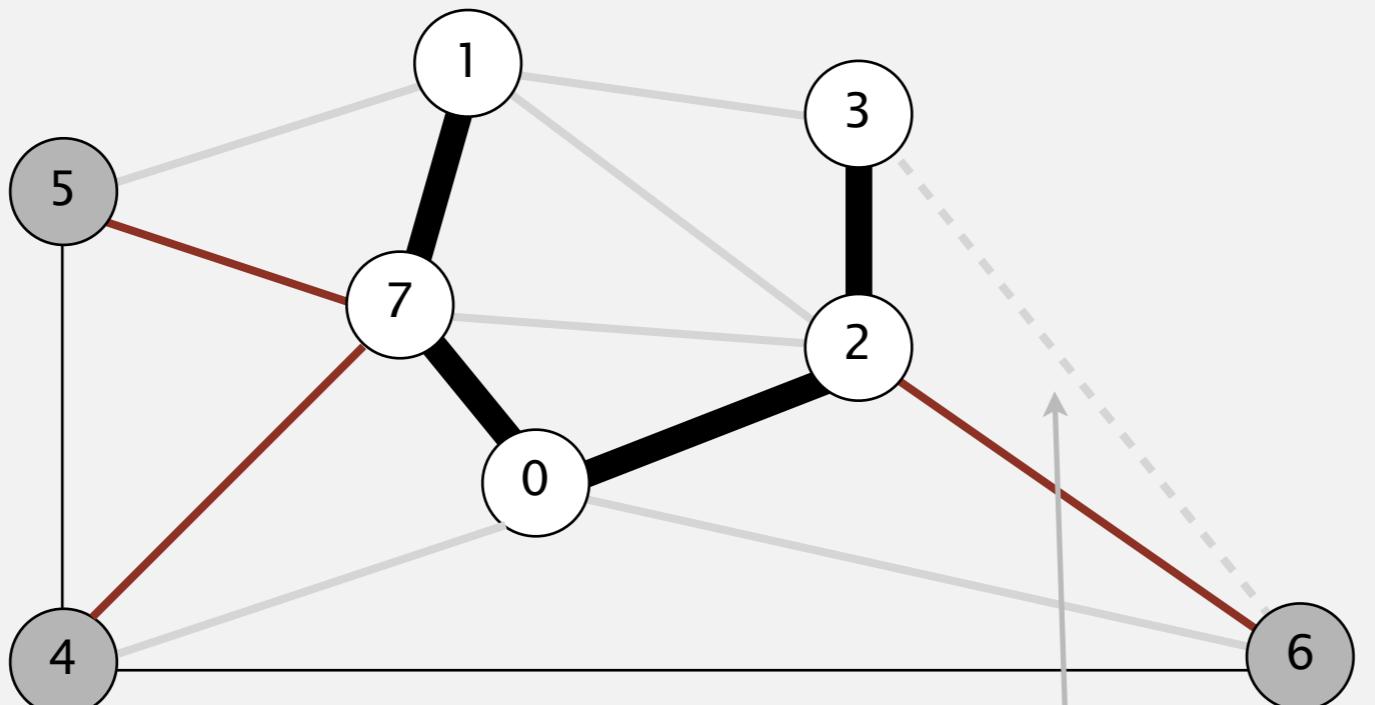
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-7	0.37
6	6-2	0.40

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.

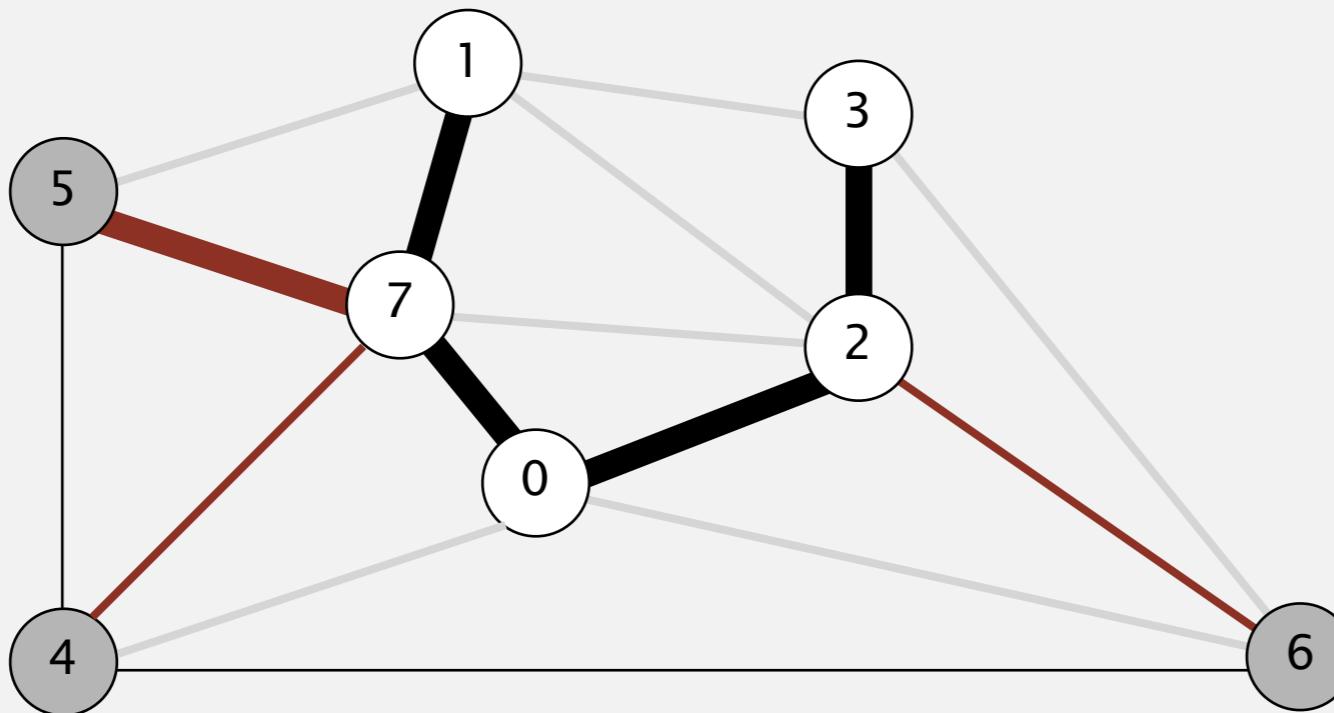


already a better connection
to 6 (discard)

v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-7	0.37
6	6-2	0.40

Prim's algorithm - Eager implementation

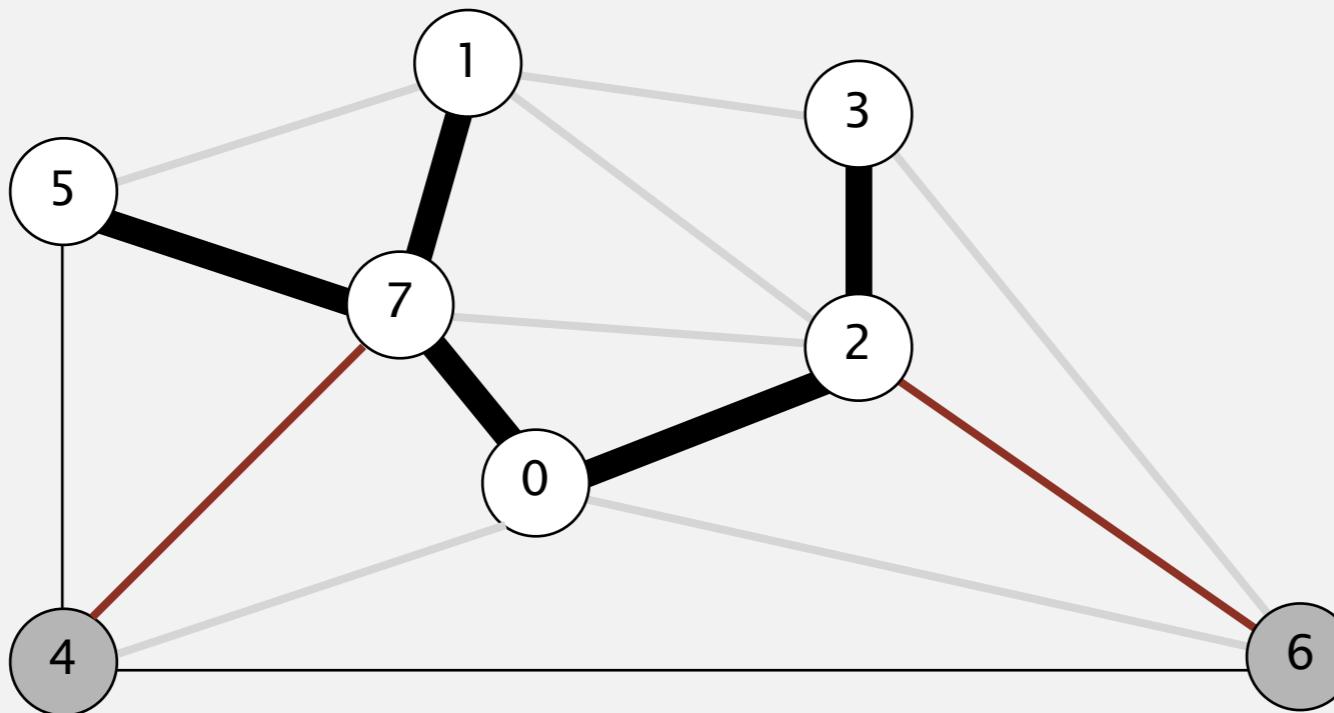
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-7	0.37
6	6-2	0.40

Prim's algorithm - Eager implementation

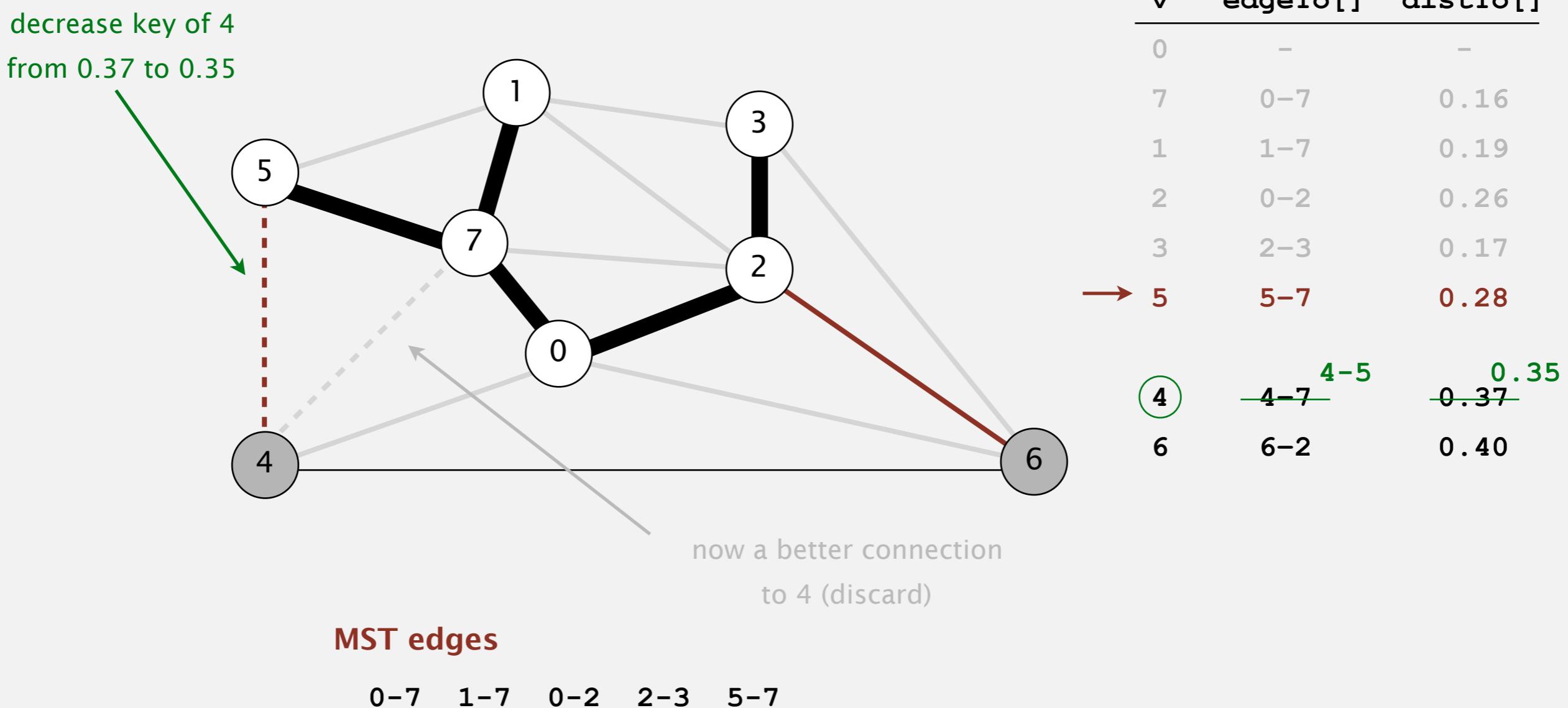
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo []	distTo []
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-7	0.37
6	6-2	0.40

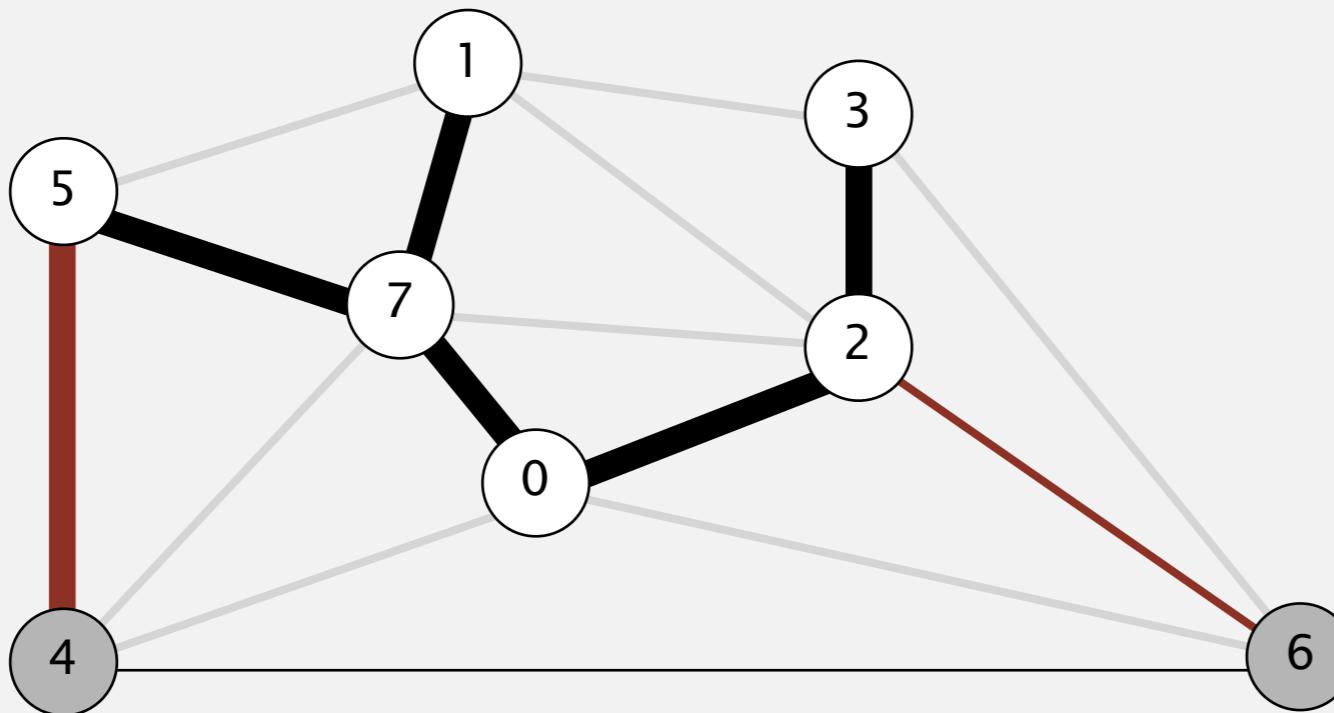
Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Eager implementation

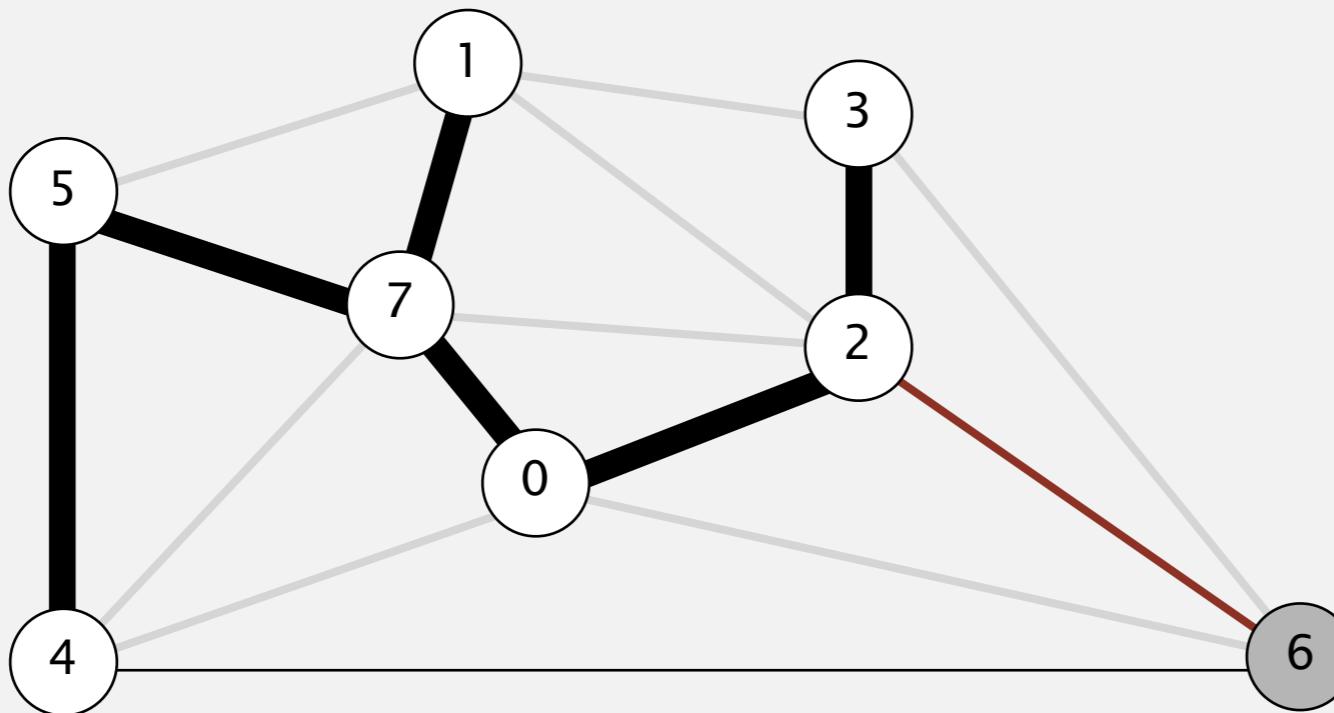
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo []	distTo []
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-5	0.35
6	6-2	0.40

Prim's algorithm - Eager implementation

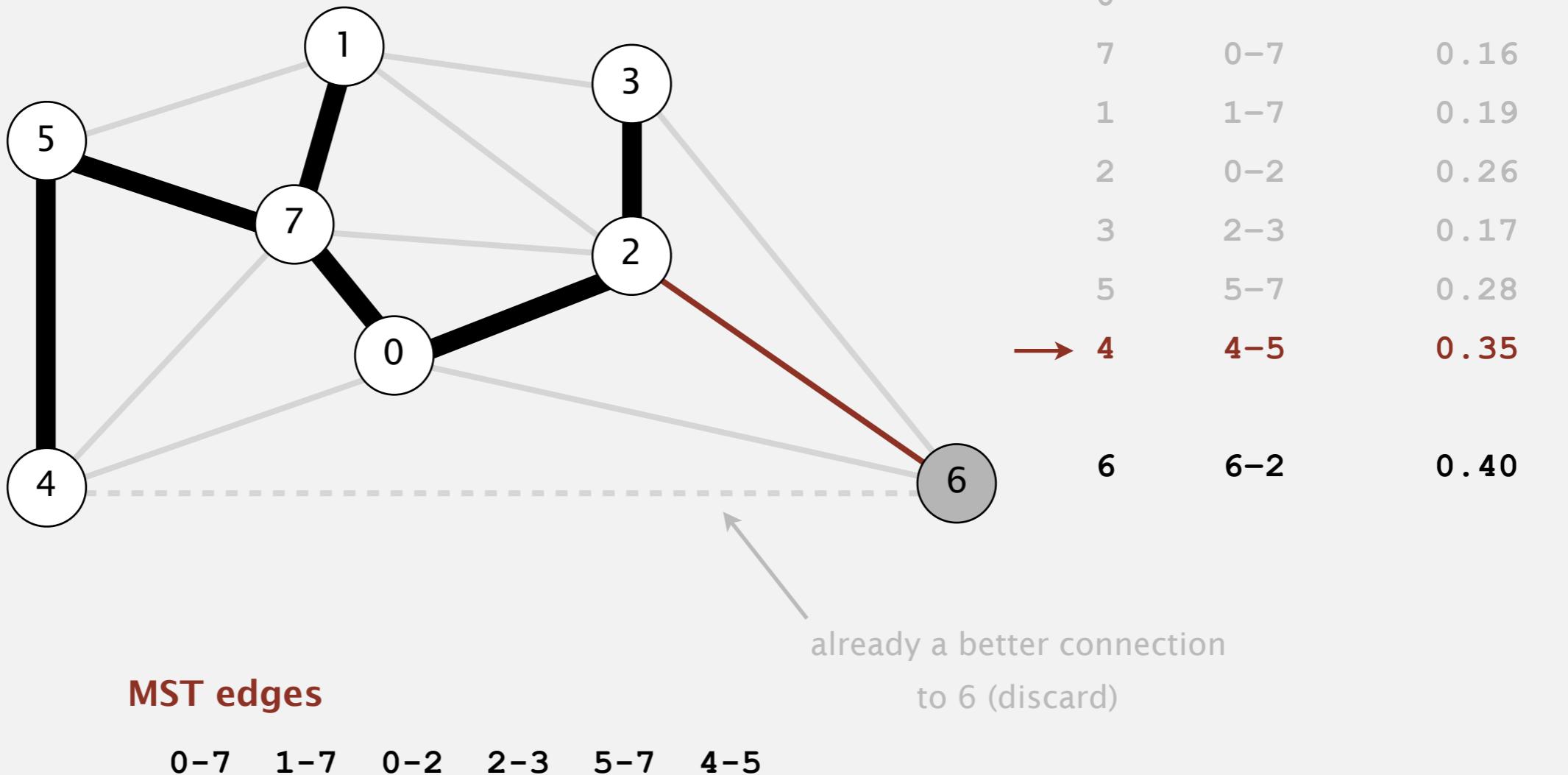
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo []	distTo []
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-5	0.35
6	6-2	0.40

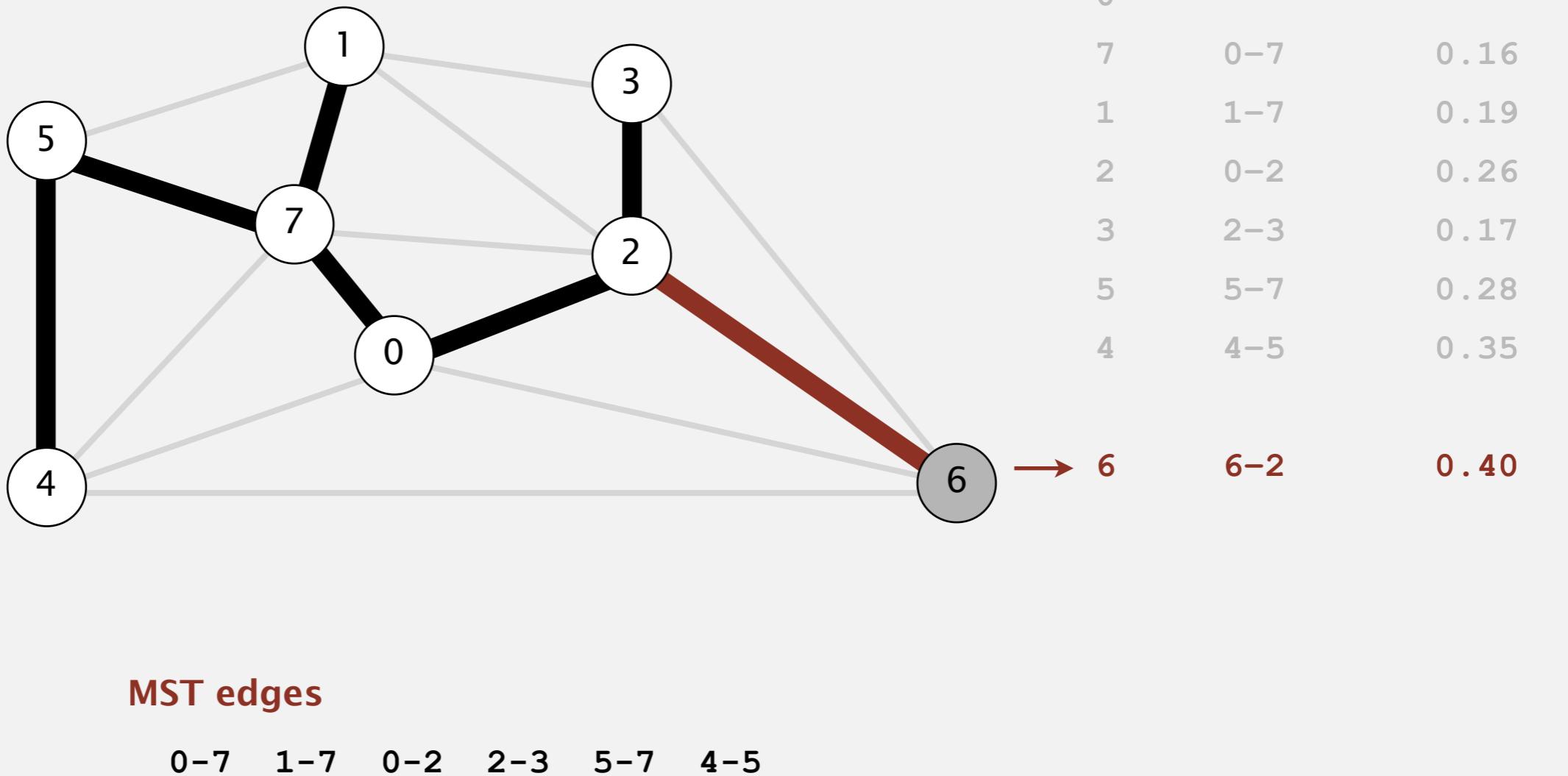
Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



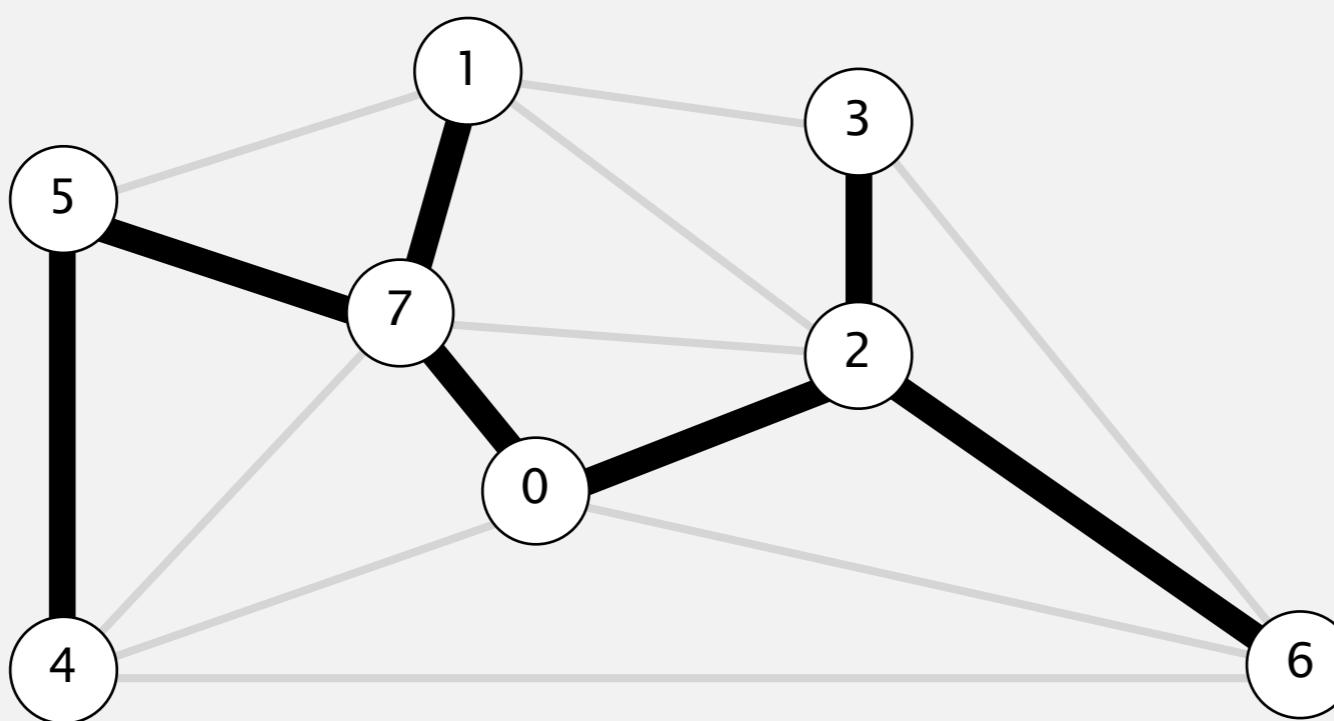
Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



Prim's algorithm - Eager implementation

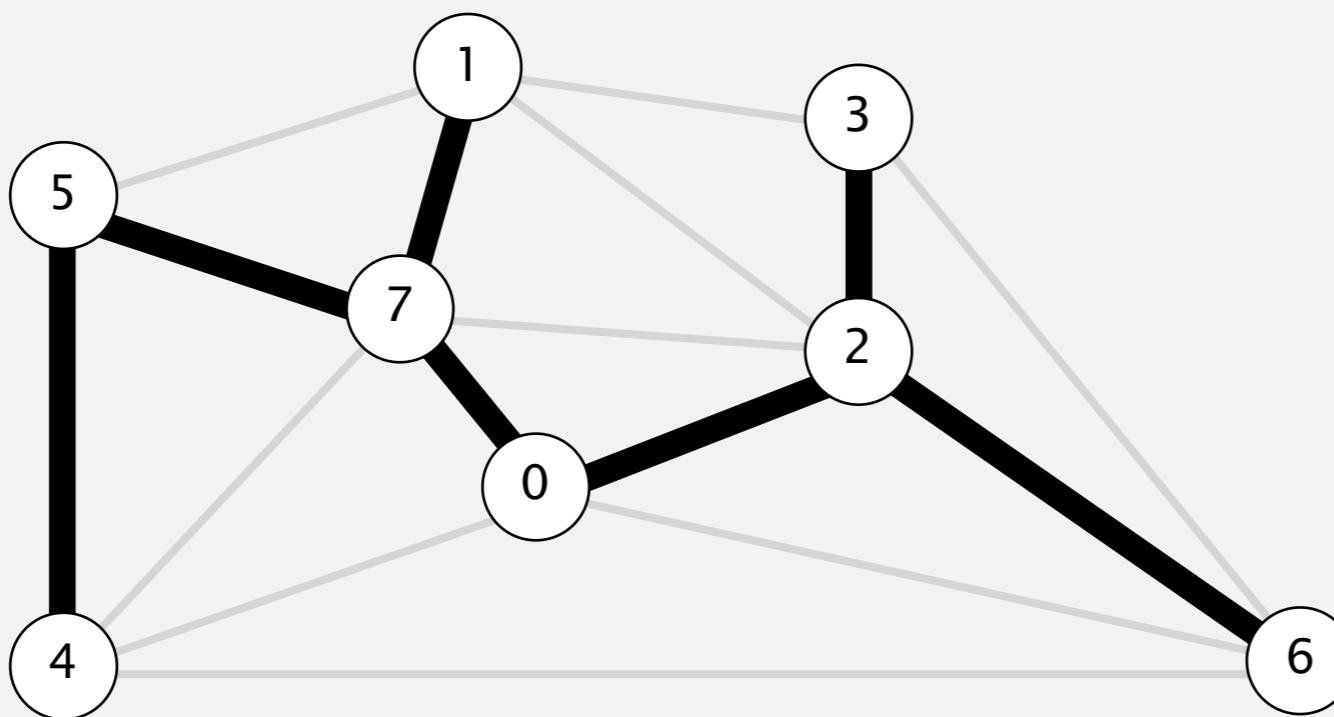
- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-5	0.35
6	6-2	0.40

Prim's algorithm - Eager implementation

- Start with vertex 0 and greedily grow tree T .
- Add to T the min weight edge with exactly one endpoint in T .
- Repeat until $V-1$ edges.



MST edges

0-7 1-7 0-2 2-3 5-7 4-5 6-2

v	edgeTo[]	distTo[]
0	-	-
7	0-7	0.16
1	1-7	0.19
2	0-2	0.26
3	2-3	0.17
5	5-7	0.28
4	4-5	0.35
6	6-2	0.40

Indexed priority queue

Associate an index between 0 and $N - 1$ with each key in a priority queue.

- Client can insert and delete-the-minimum.
- Client can change the key by specifying the index.

```
public class IndexMinPQ<Key extends Comparable<Key>>
```

```
IndexMinPQ(int N)
```

create indexed priority queue

with indices 0, 1, ..., N-1

associate key with index k

```
void insert(int k, Key key)
```

decrease the key associated with index k

```
void decreaseKey(int k, Key key)
```

is k an index on the priority queue?

```
boolean contains()
```

*remove a minimal key and return its
associated index*

```
int delMin()
```

is the priority queue empty?

```
boolean isEmpty()
```

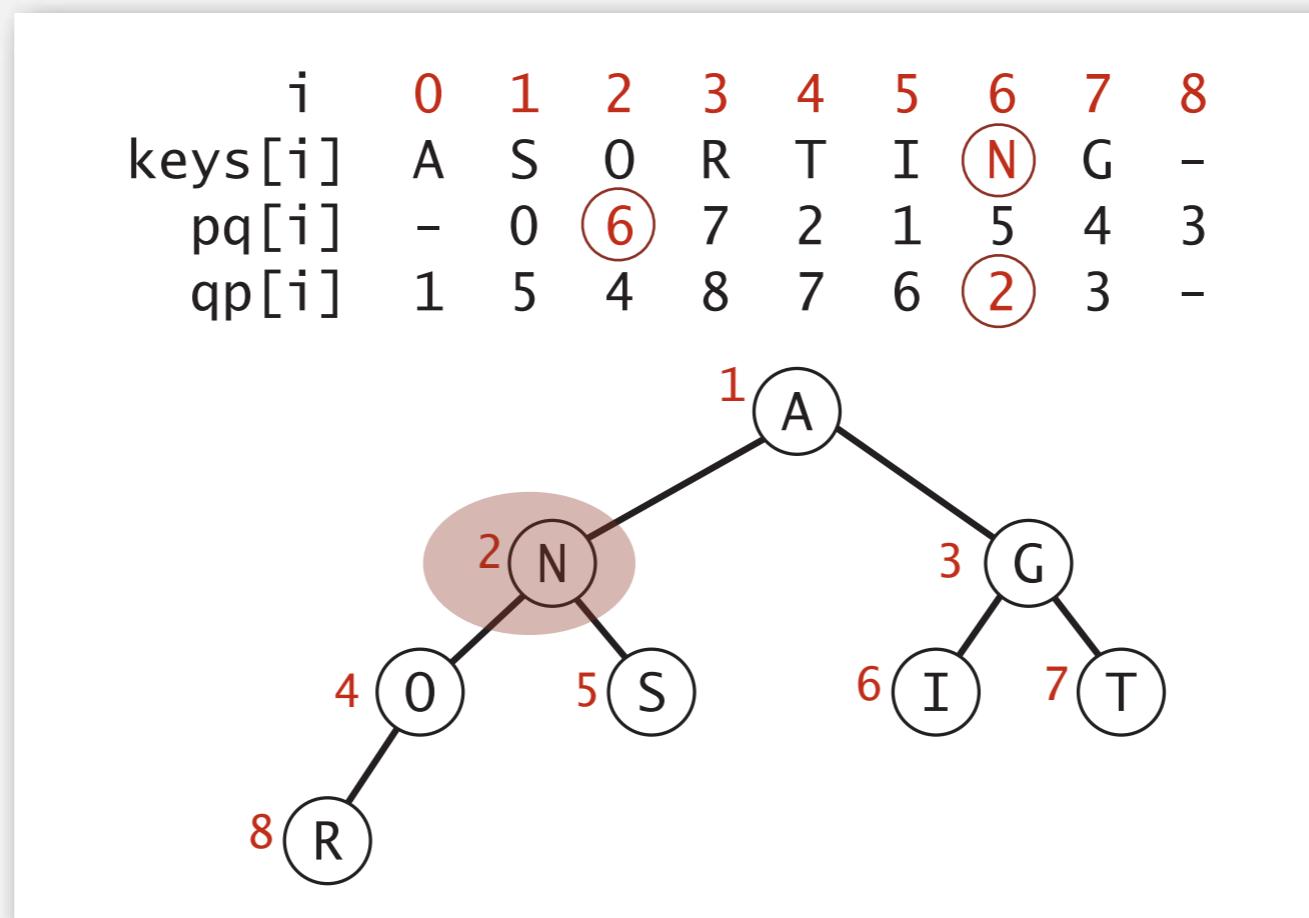
number of entries in the priority queue

```
int size()
```

Indexed priority queue implementation

Implementation.

- Start with same code as `MinPQ`.
- Maintain parallel arrays `keys[]`, `pq[]`, and `qp[]` so that:
 - `keys[i]` is the priority of `i`
 - `pq[i]` is the index of the key in heap position `i`
 - `qp[i]` is the heap position of the key with index `i`
- Use `swim(qp[k])` implement `decreaseKey(k, key)`.



Prim's algorithm: running time

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap (Johnson 1975)	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	$1 \dagger$	$\log V \dagger$	$1 \dagger$	$E + V \log V$

\dagger amortized

Bottom line.

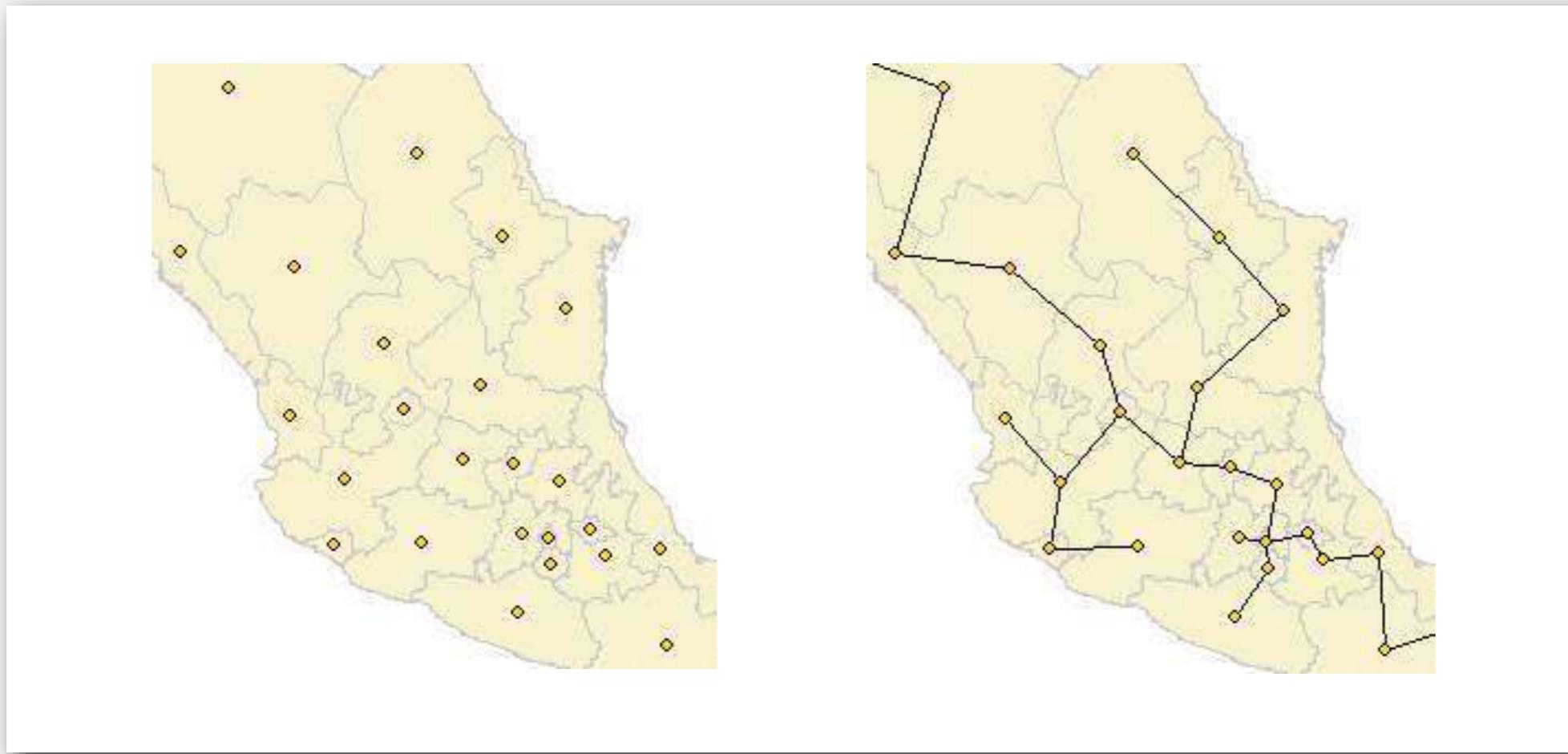
- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

MINIMUM SPANNING TREES

- ▶ Greedy algorithm
- ▶ Edge-weighted graph API
- ▶ Kruskal's algorithm
- ▶ Prim's algorithm
- ▶ Context

Euclidean MST

Given N points in the plane, find MST connecting them, where the distances between point pairs are their **Euclidean** distances.



Brute force. Compute $\sim N^2 / 2$ distances and run Prim's algorithm.

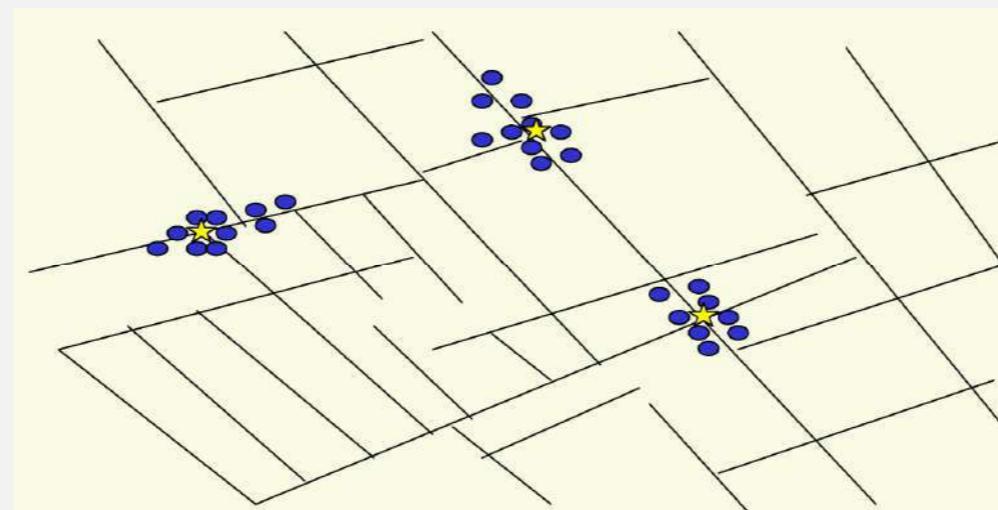
Ingenuity. Exploit geometry and do it in $\sim c N \log N$.

Scientific application: clustering

k-clustering. Divide a set of objects classify into k coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Goal. Divide into clusters so that objects in different clusters are far apart.



outbreak of cholera deaths in London in 1850s (Nina Mishra)

Applications.

- Routing in mobile ad hoc networks.
- Document categorization for web search.
- Similarity searching in medical image databases.
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

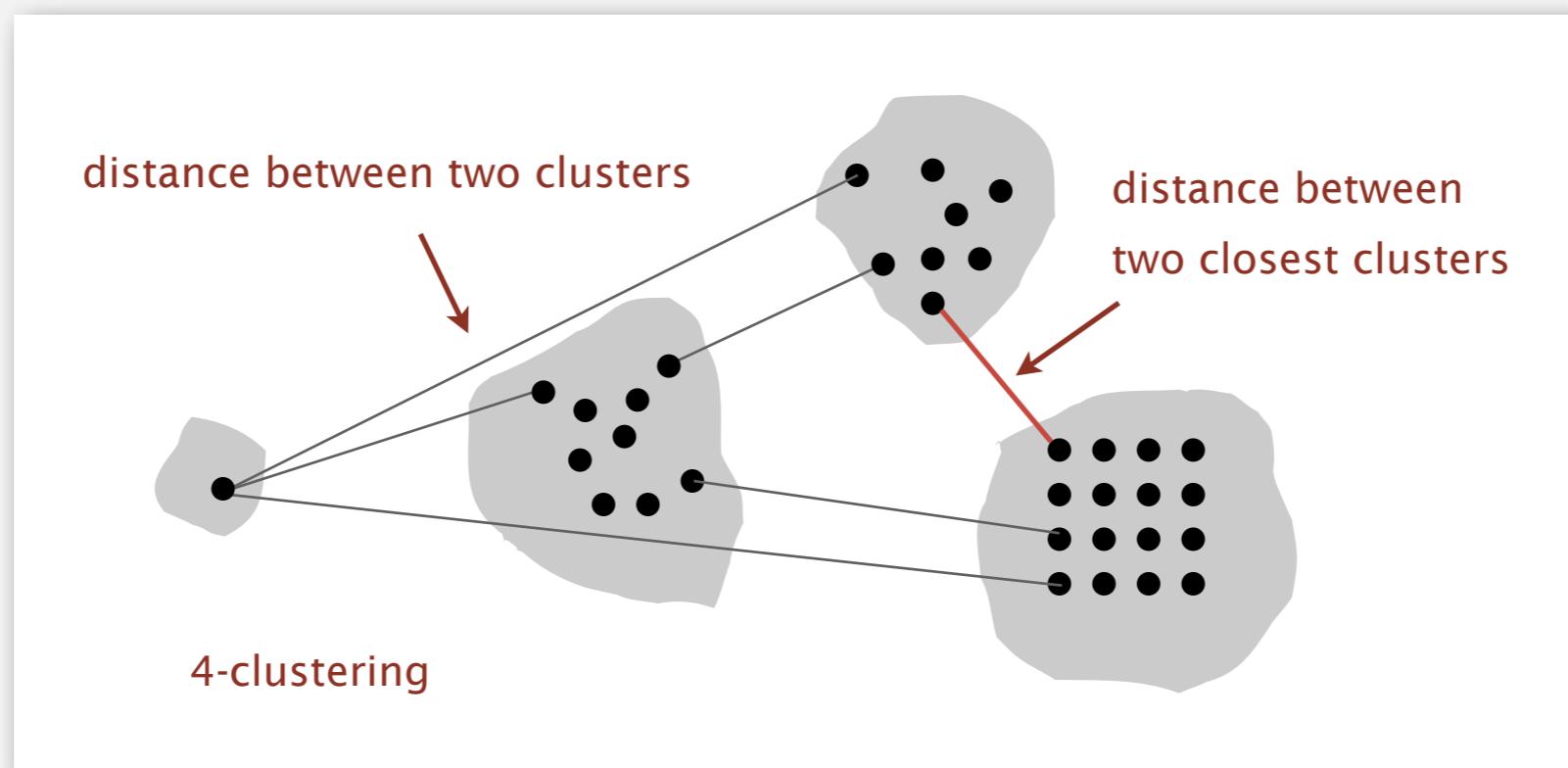
Single-link clustering

k-clustering. Divide a set of objects classify into k coherent groups.

Distance function. Numeric value specifying "closeness" of two objects.

Single link. Distance between two clusters equals the distance between the two closest objects (one in each cluster).

Single-link clustering. Given an integer k, find a k-clustering that maximizes the distance between two closest clusters.

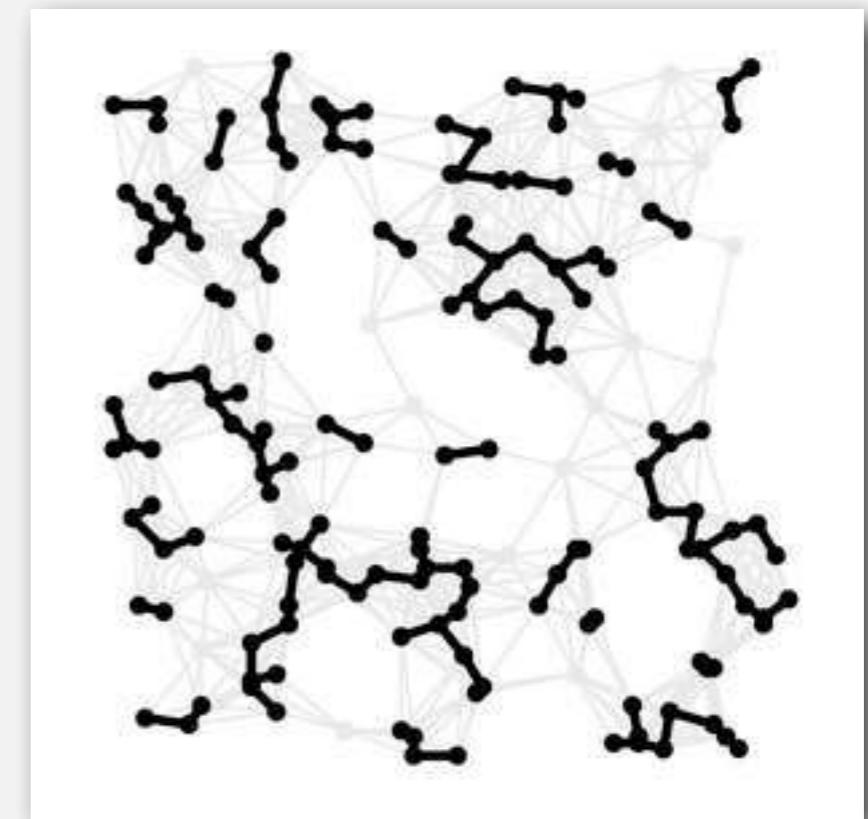


Single-link clustering algorithm

“Well-known” algorithm for single-link clustering:

- Form V clusters of one object each.
- Find the closest pair of objects such that each object is in a different cluster, and merge the two clusters.
- Repeat until there are exactly k clusters.

Observation. This is Kruskal's algorithm
(stop when k connected components).



Alternate solution. Run Prim's algorithm and delete $k-1$ max weight edges.

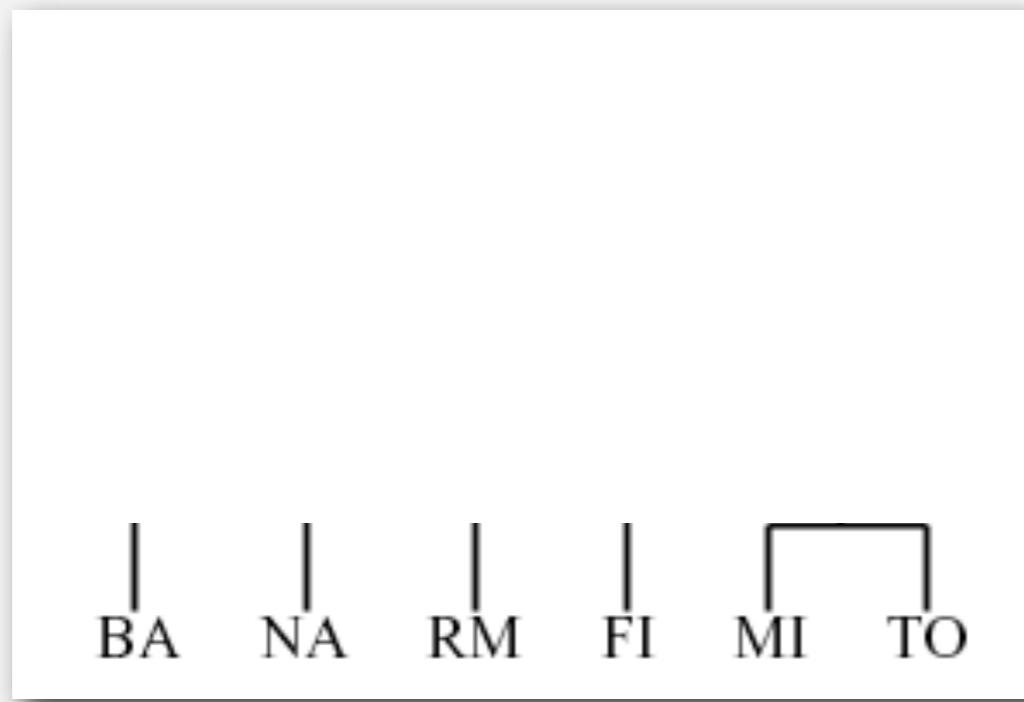
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



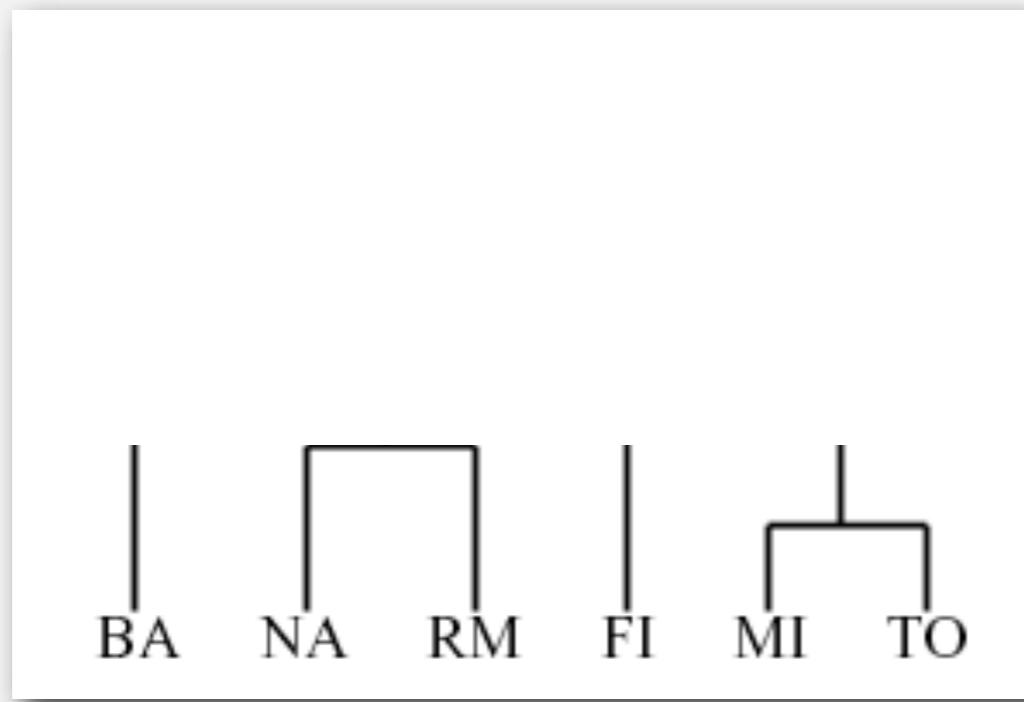
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



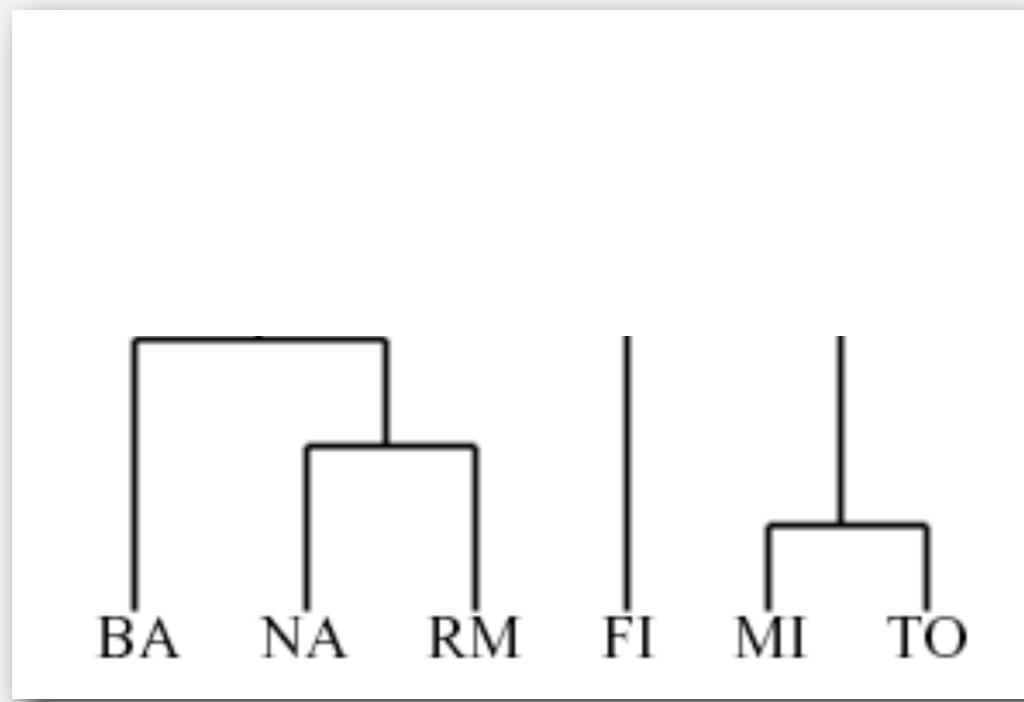
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



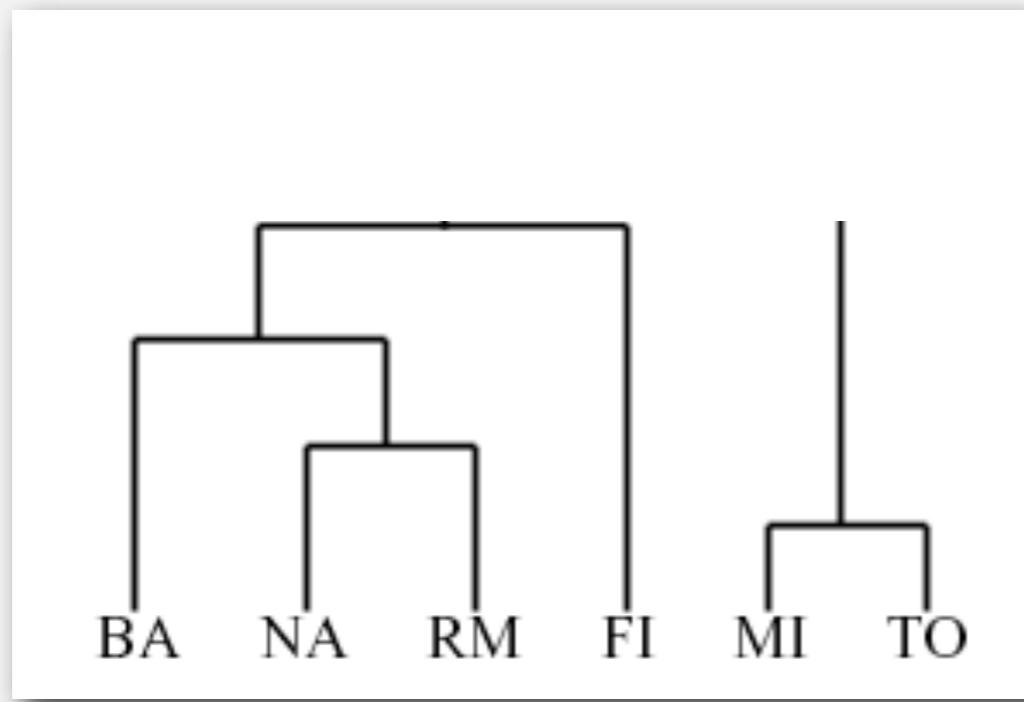
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



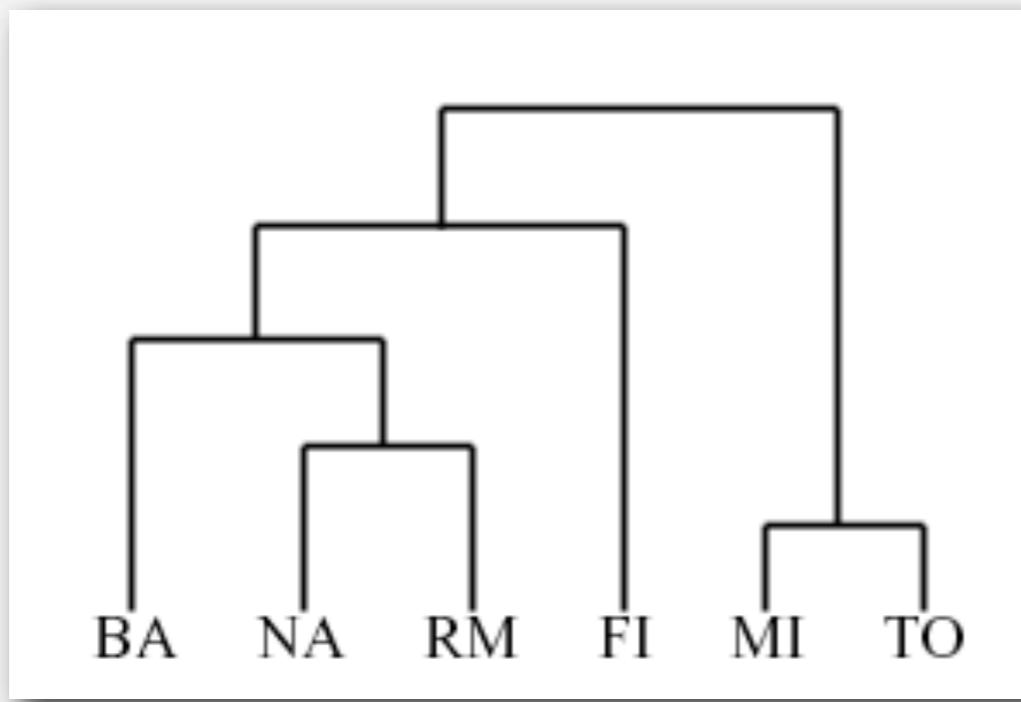
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



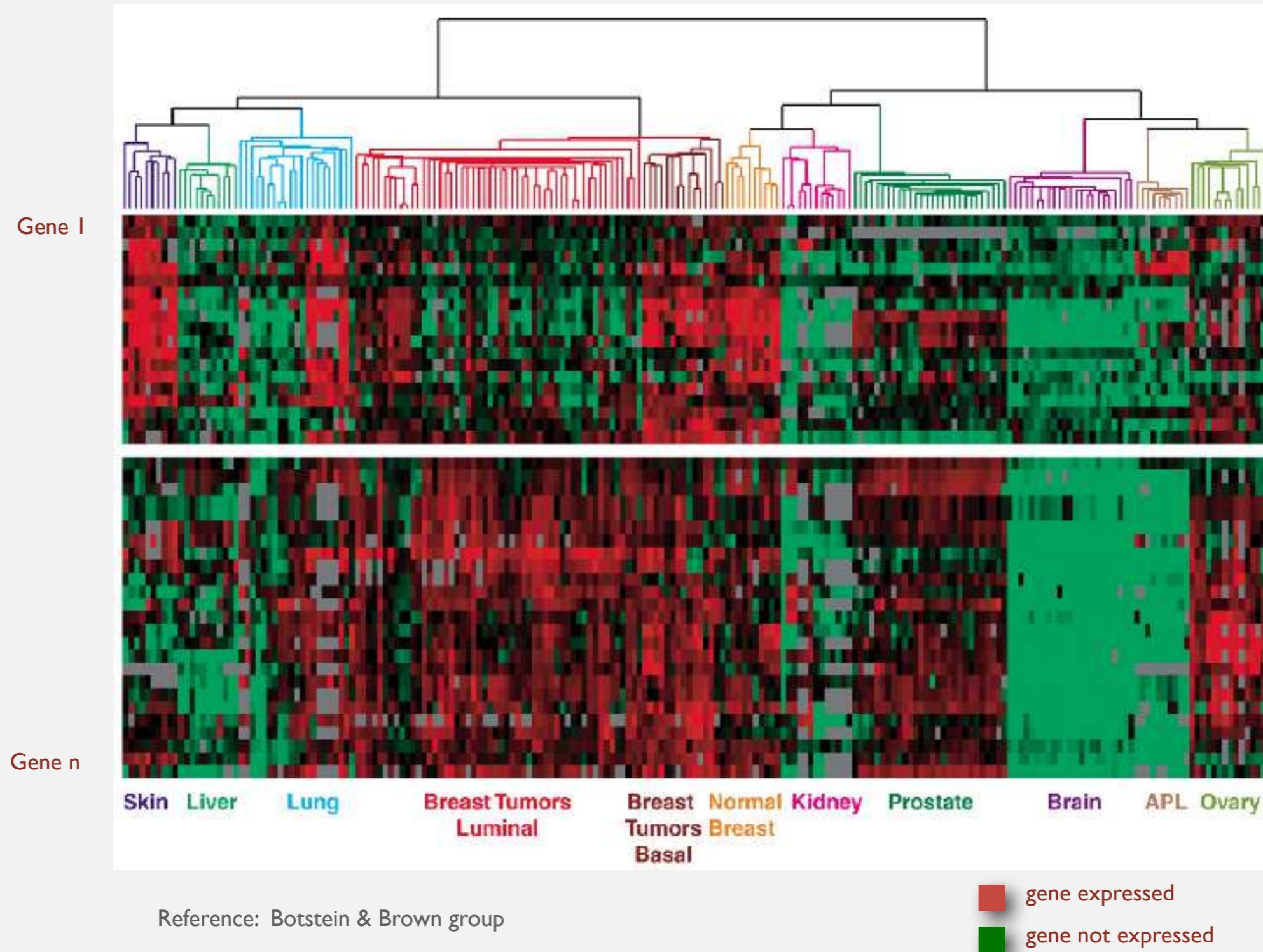
Dendrogram

Dendrogram. Tree diagram that illustrates arrangement of clusters.



Dendrogram of cancers in human

Tumors in similar tissues cluster together.



BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

SHORTEST PATH

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ Shortest Paths
- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ Dijkstra's algorithm
- ▶ Edge-weighted DAGs
- ▶ Negative weights

SHORTEST PATHS

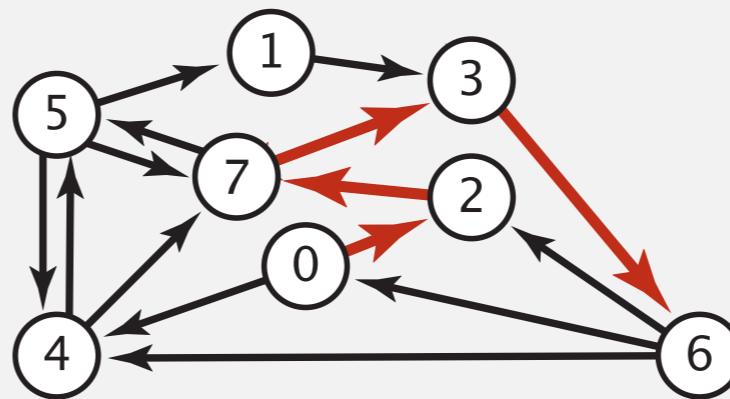
- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ Dijkstra's algorithm
- ▶ Edge-weighted DAGs
- ▶ Negative weights

Shortest paths in a weighted digraph

Given an edge-weighted digraph, find the shortest (directed) path from s to t .

edge-weighted digraph

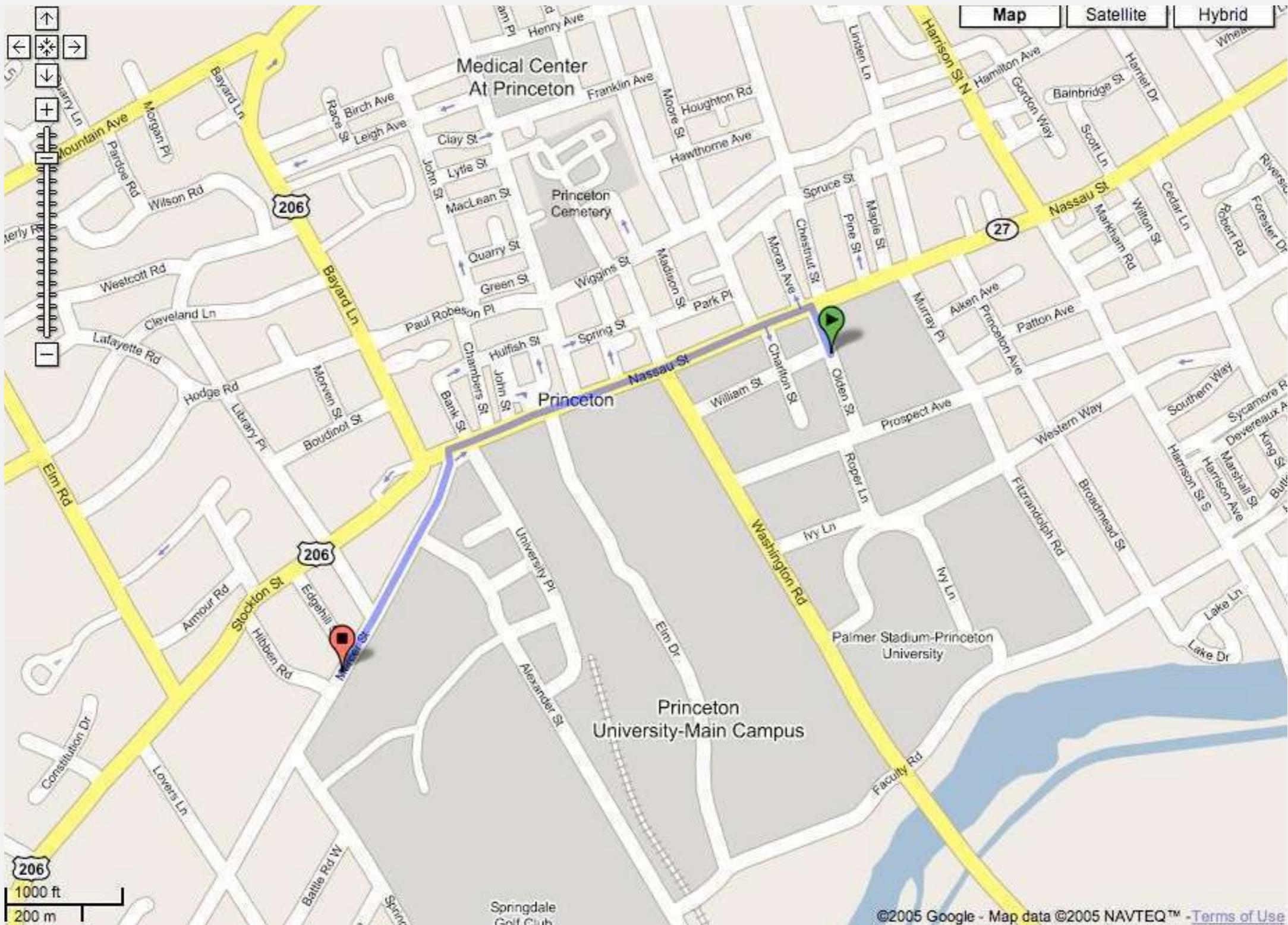
4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Google maps



©2005 Google - Map data ©2005 NAVTEQ™ - [Terms of Use](#)

Car navigation



Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.



http://en.wikipedia.org/wiki/Seam_carving



Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest path variants

Which vertices?

- Source-sink: from one vertex to another.
- Single source: from one vertex to every other.
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.
- Arbitrary weights.
- Euclidean weights.

Cycles?

- No directed cycles.
- No "negative cycles."

Simplifying assumption. Shortest paths from s to each vertex v exist.

SHORTEST PATHS

- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ Dijkstra's algorithm
- ▶ Edge-weighted DAGs
- ▶ Negative weights

Weighted directed edge API

```
public class DirectedEdge
```

```
    DirectedEdge(int v, int w, double weight)      weighted edge v→w
```

```
    int from()                                     vertex v
```

```
    int to()                                       vertex w
```

```
    double weight()                                weight of this edge
```

```
    String toString()                             string representation
```



Idiom for processing an edge e : `int v = e.from(), w = e.to();`

Weighted directed edge: implementation in Java

Similar to `Edge` for undirected graphs, but a bit simpler.

```
public class DirectedEdge
{
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from()
    { return v; }

    public int to()
    { return w; }

    public int weight()
    { return weight; }
}
```

`from()` and `to()` replace
`either()` and `other()`

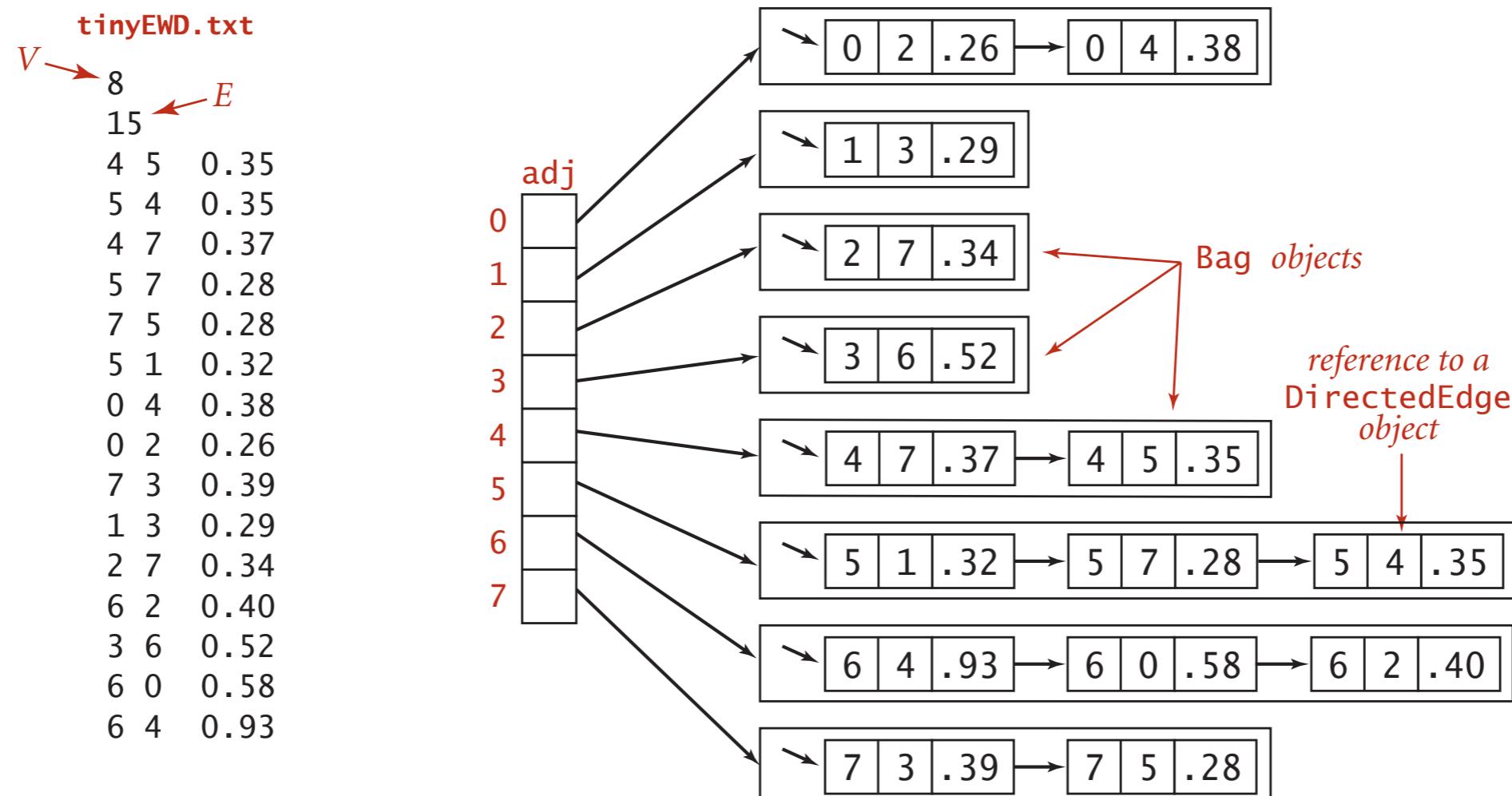
Edge-weighted digraph API

```
public class EdgeWeightedDigraph
```

<code>EdgeWeightedDigraph(int v)</code>	<i>edge-weighted digraph with V vertices</i>
<code>EdgeWeightedDigraph(In in)</code>	<i>edge-weighted digraph from input stream</i>
<code>void addEdge(DirectedEdge e)</code>	<i>add weighted directed edge e</i>
<code>Iterable<DirectedEdge> adj(int v)</code>	<i>edges pointing from v</i>
<code>int V()</code>	<i>number of vertices</i>
<code>int E()</code>	<i>number of edges</i>
<code>Iterable<DirectedEdge> edges()</code>	<i>all edges</i>
<code>String toString()</code>	<i>string representation</i>

Conventions. Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists representation



Edge-weighted digraph: adjacency-lists implementation in Java

Same as EdgeWeightedGraph except replace Graph with Digraph.

```
public class EdgeWeightedDigraph
{
    private final int V;
    private final Bag<Edge>[] adj;

    public EdgeWeightedDigraph(int V)
    {
        this.V = V;
        adj = (Bag<DirectedEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v)
    {   return adj[v];   }
}
```

←
add edge $e = v \rightarrow w$ only to
 v 's adjacency list

Single-source shortest paths API

Goal. Find the shortest path from s to every other vertex.

```
public class SP
```

```
    SP(EdgeWeightedDigraph G, int s) shortest paths from s in graph G
```

```
    double distTo(int v) length of shortest path from s to v
```

```
    Iterable <DirectedEdge> pathTo(int v) shortest path from s to v
```

```
    boolean hasPathTo(int v) is there a path from s to v?
```

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
    StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
    for (DirectedEdge e : sp.pathTo(v))
        StdOut.print(e + " ");
    StdOut.println();
}
```

Single-source shortest paths API

Goal. Find the shortest path from s to every other vertex.

```
public class SP
```

```
    SP(EdgeWeightedDigraph G, int s) shortest paths from s in graph G
```

```
    double distTo(int v) length of shortest path from s to v
```

```
    Iterable <DirectedEdge> pathTo(int v) shortest path from s to v
```

```
    boolean hasPathTo(int v) is there a path from s to v?
```

```
% java SP tinyEWD.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38  4->5 0.35
0 to 6 (1.51): 0->2 0.26  2->7 0.34  7->3 0.39  3->6 0.52
0 to 7 (0.60): 0->2 0.26  2->7 0.34
```

SHORTEST PATHS

- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ Dijkstra's algorithm
- ▶ Edge-weighted DAGs
- ▶ Negative weights

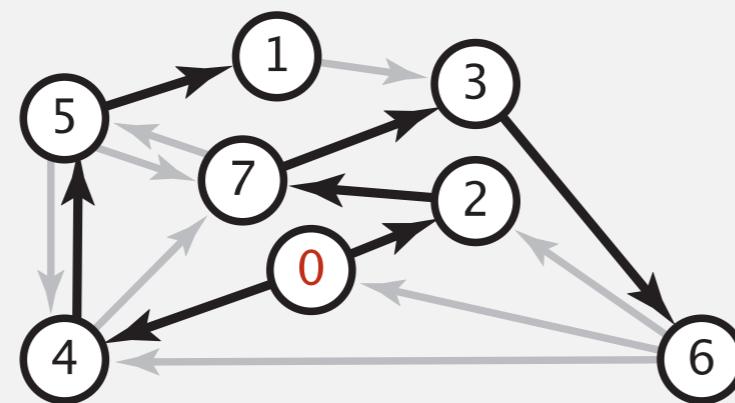
Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A **shortest-paths tree (SPT)** solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

- `distTo[v]` is length of shortest path from s to v .
- `edgeTo[v]` is last edge on shortest path from s to v .



shortest-paths tree from 0

Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A **shortest-paths tree (SPT)** solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

- `distTo[v]` is length of shortest path from s to v .
- `edgeTo[v]` is last edge on shortest path from s to v .

```
public double distTo(int v)
{   return distTo[v]; }
```

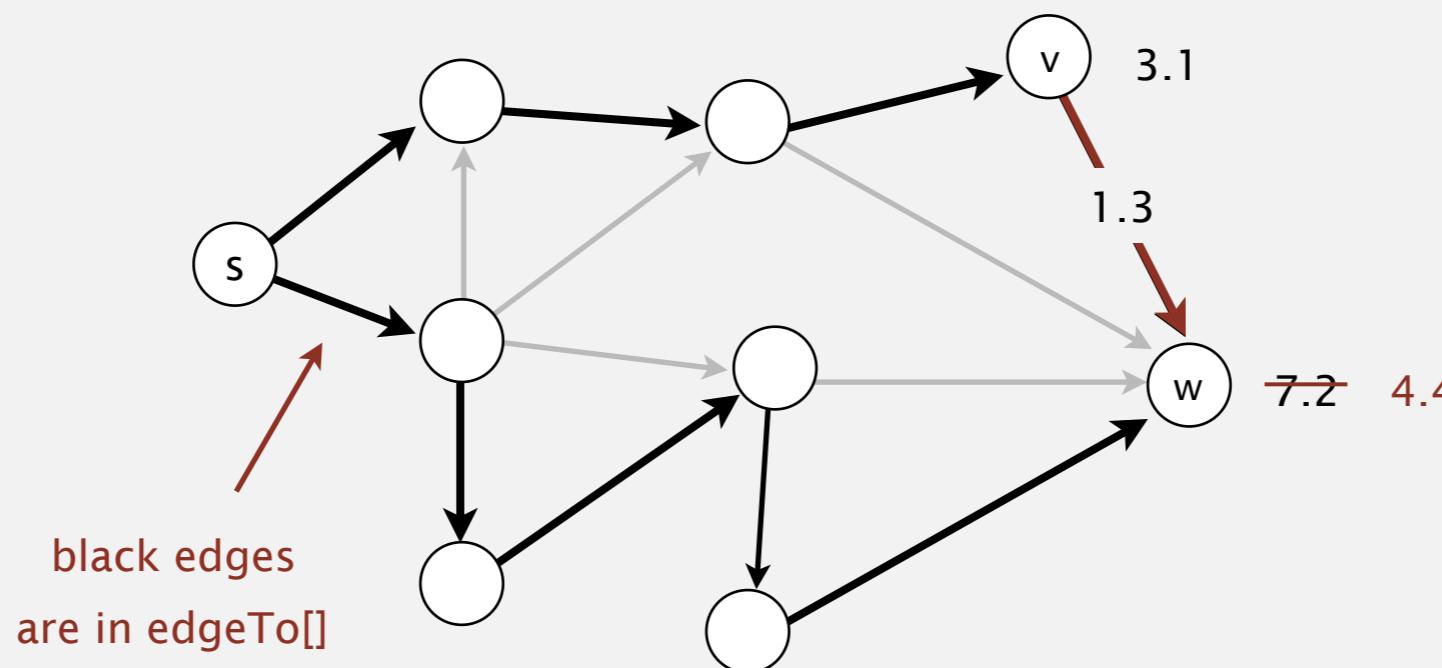
```
public Iterable<DirectedEdge> pathTo(int v)
{
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```

Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest **known** path from s to v .
- `distTo[w]` is length of shortest **known** path from s to w .
- `edgeTo[w]` is last edge on shortest **known** path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v ,
update `distTo[w]` and `edgeTo[w]`.

$v \rightarrow w$ successfully relaxes



Edge relaxation

Relax edge $e = v \rightarrow w$.

- `distTo[v]` is length of shortest **known** path from s to v .
- `distTo[w]` is length of shortest **known** path from s to w .
- `edgeTo[w]` is last edge on shortest **known** path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v ,
update `distTo[w]` and `edgeTo[w]`.

```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Shortest-paths optimality conditions

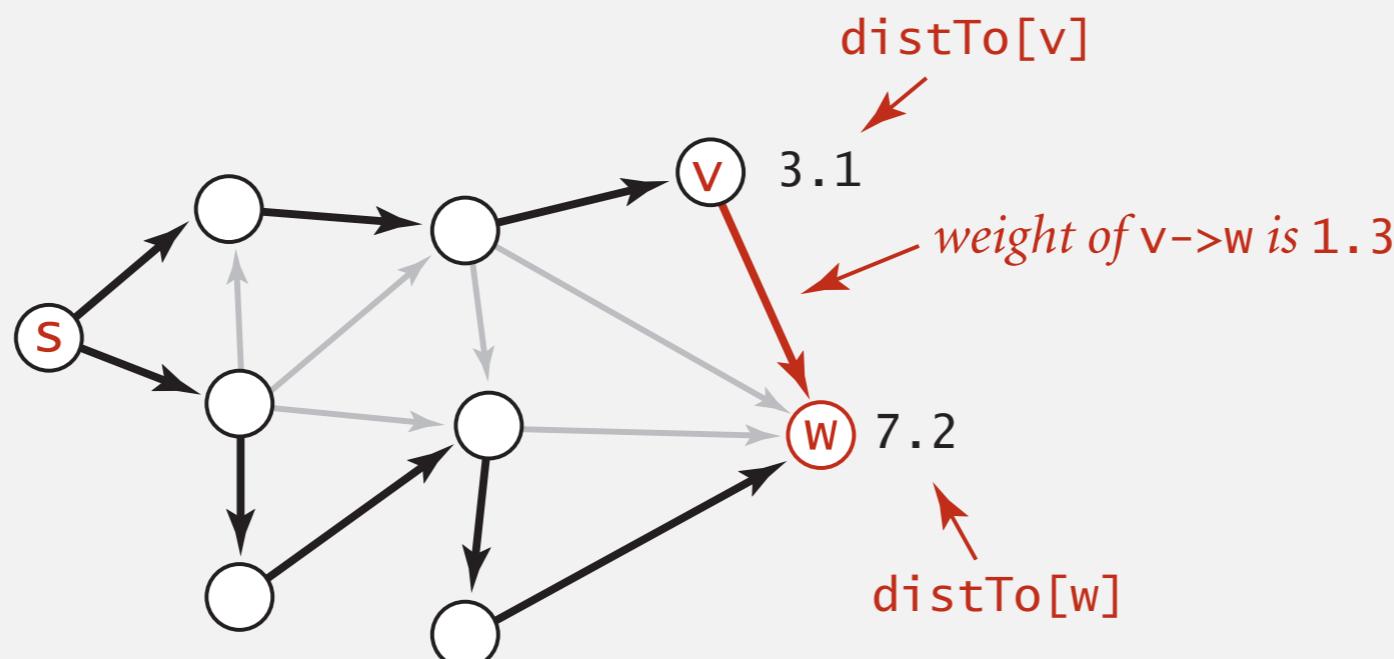
Proposition. Let G be an edge-weighted digraph.

Then `distTo[]` are the shortest path distances from s iff:

- For each vertex v , `distTo[v]` is the length of some path from s to v .
- For each edge $e = v \rightarrow w$, `distTo[w] ≤ distTo[v] + e.weight()`.

Pf. \Leftarrow [necessary]

- Suppose that `distTo[w] > distTo[v] + e.weight()` for some edge $e = v \rightarrow w$.
- Then, e gives a path from s to w (through v) of length less than `distTo[w]`.



Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then $\text{distTo}[]$ are the shortest path distances from s iff:

- For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Pf. \Rightarrow [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = w$ is a shortest path from s to w .
- Then,
$$\begin{aligned}\text{distTo}[v_k] &\leq \text{distTo}[v_{k-1}] + e_k.\text{weight}() \\ \text{distTo}[v_{k-1}] &\leq \text{distTo}[v_{k-2}] + e_{k-1}.\text{weight}() \\ &\quad \vdots \\ \text{distTo}[v_1] &\leq \text{distTo}[v_0] + e_1.\text{weight}()\end{aligned}$$

$e_i = i^{\text{th}}$ edge on shortest path from s to w
- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:
$$\text{distTo}[w] = \text{distTo}[v_k] \leq e_k.\text{weight}() + e_{k-1}.\text{weight}() + \dots + e_1.\text{weight}()$$

weight of shortest path from s to w
- Thus, $\text{distTo}[w]$ is the weight of shortest path to w . ■

$\text{distTo}[w]$
↑
weight of some path from s to w

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.

Proposition. Generic algorithm computes SPT (if it exists) from s .

Pf sketch.

- Throughout algorithm, $\text{distTo}[v]$ is the length of a simple path from s to v (and $\text{edgeTo}[v]$ is last edge on path).
- Each successful relaxation decreases $\text{distTo}[v]$ for some v .
- The entry $\text{distTo}[v]$ can decrease at most a finite number of times. ■

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.
-

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).

SHORTEST PATHS

- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ **Dijkstra's algorithm**
- ▶ Edge-weighted DAGs
- ▶ Negative weights

Edsger W. Dijkstra: select quotes

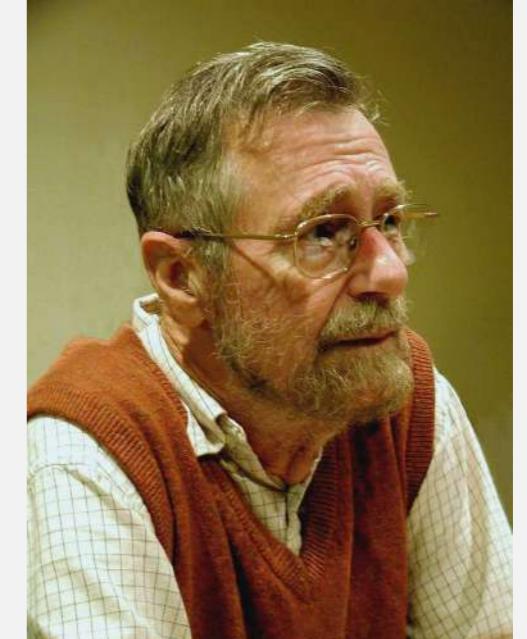
“Do only what only you can do.”

“In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”

“The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.”

“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”

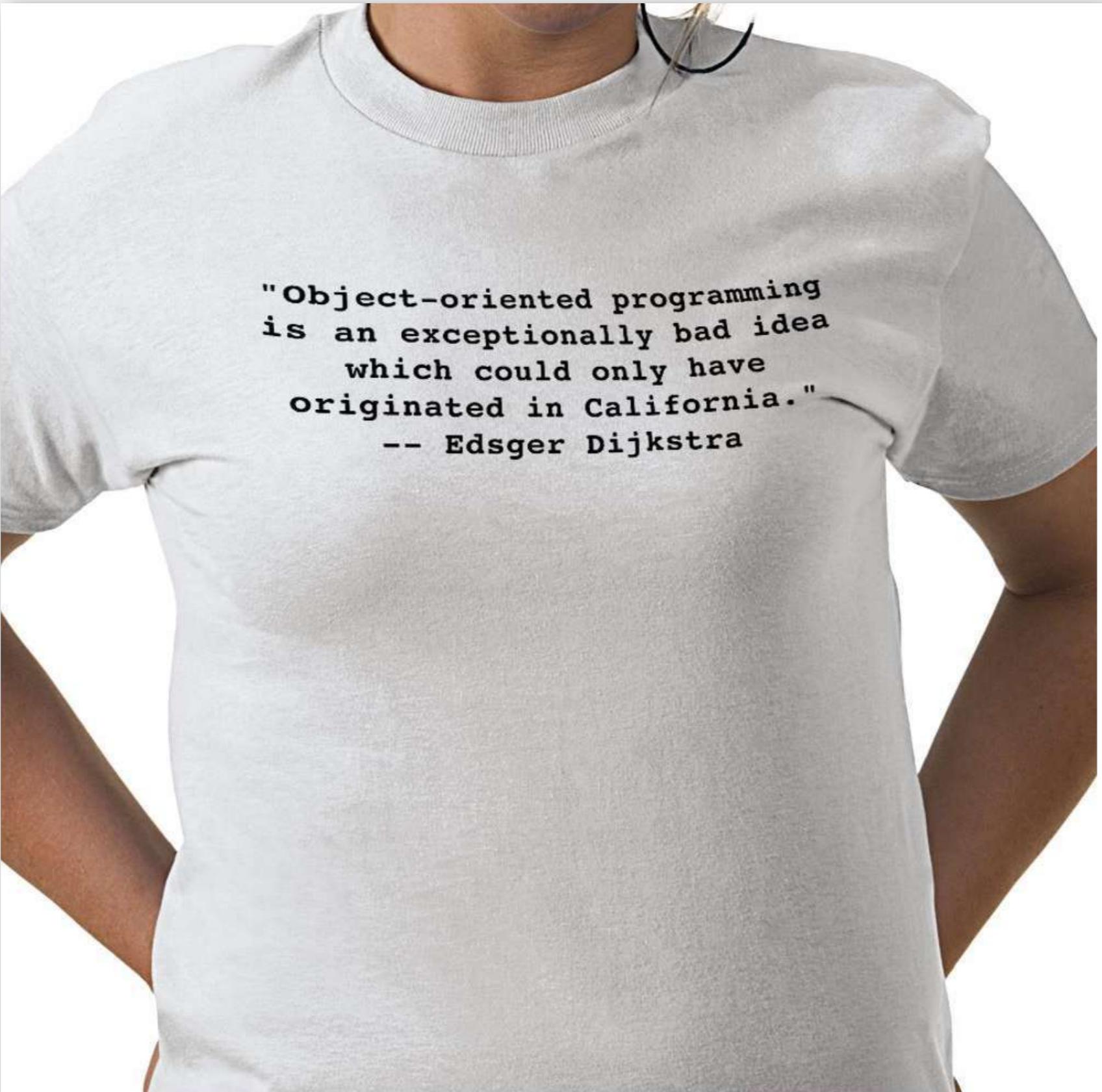
“APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.”



Edsger W. Dijkstra
Turing award 1972

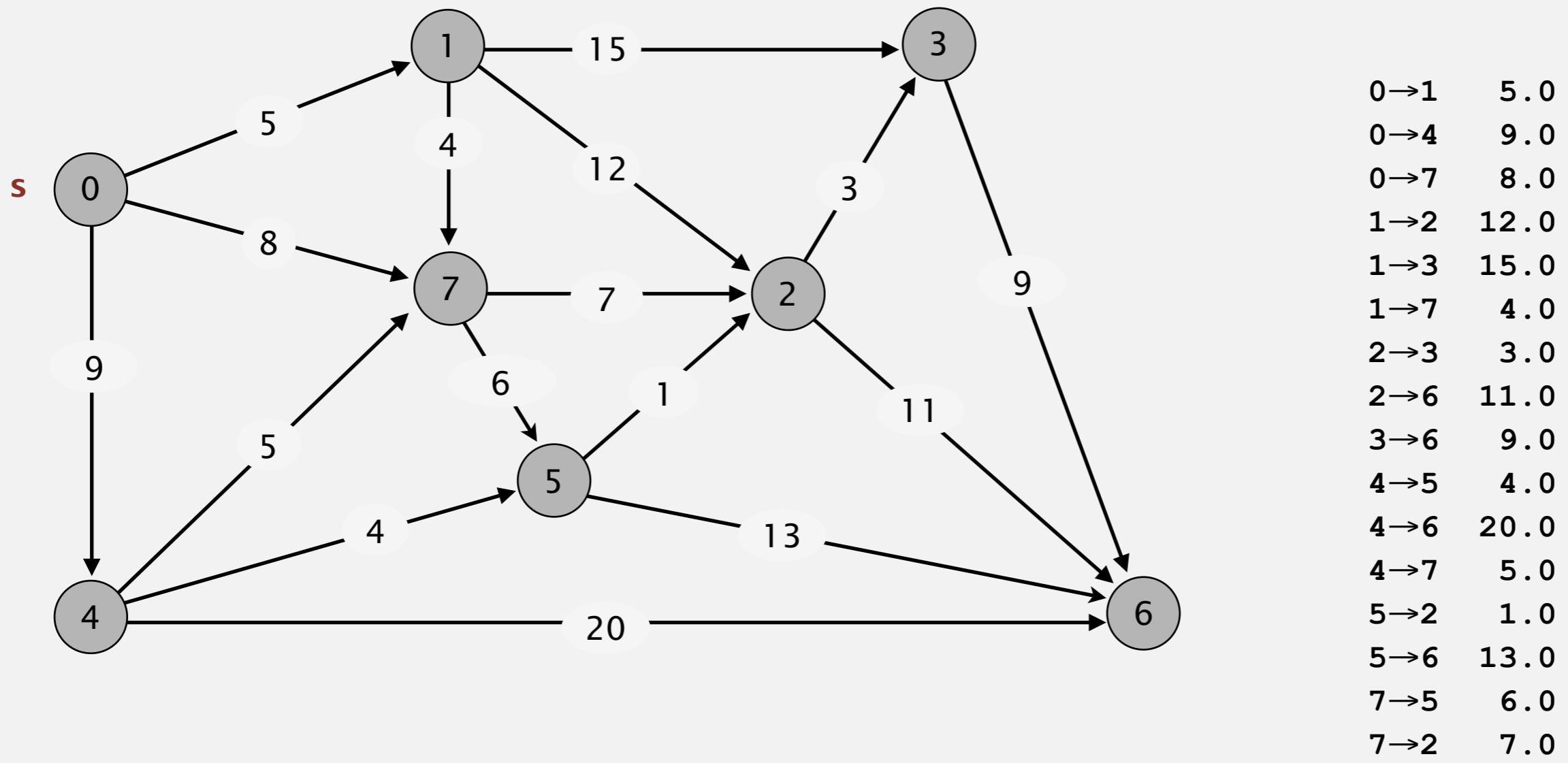
www.cs.utexas.edu/users/EWD

Edsger W. Dijkstra: select quotes



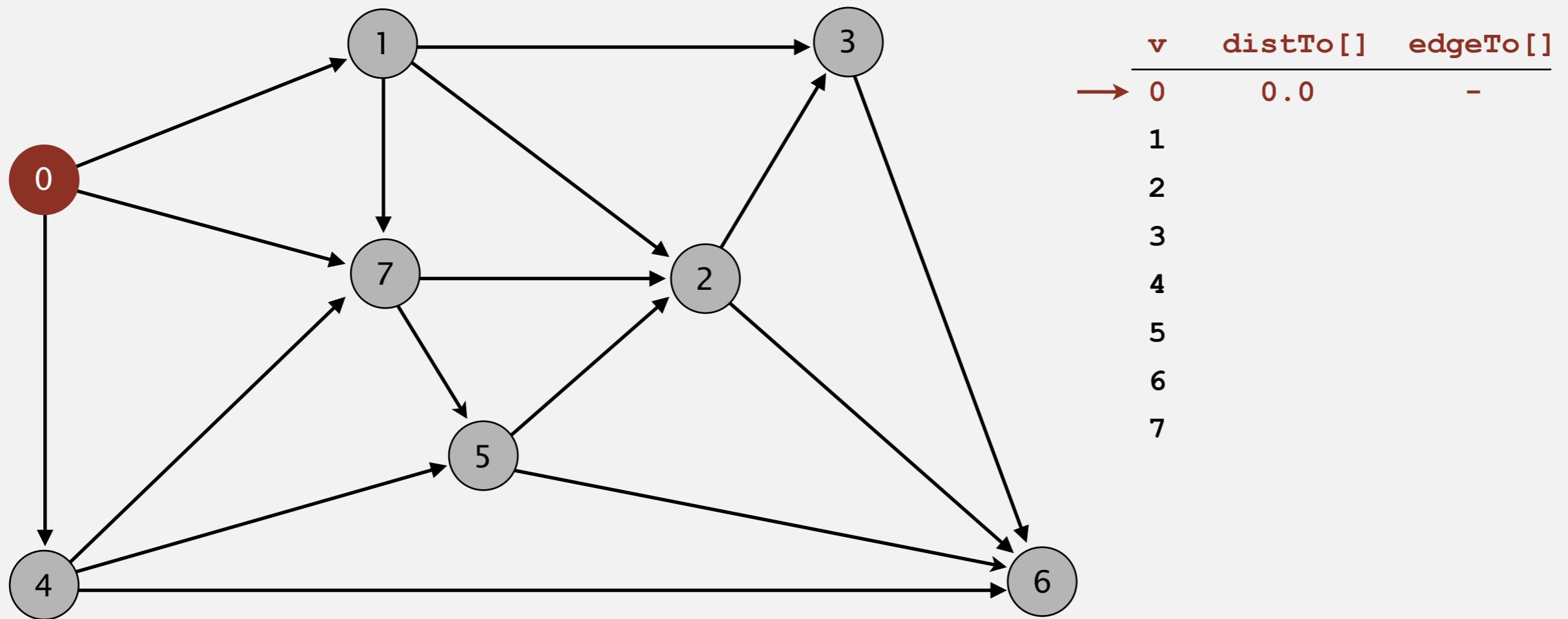
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



Dijkstra's algorithm

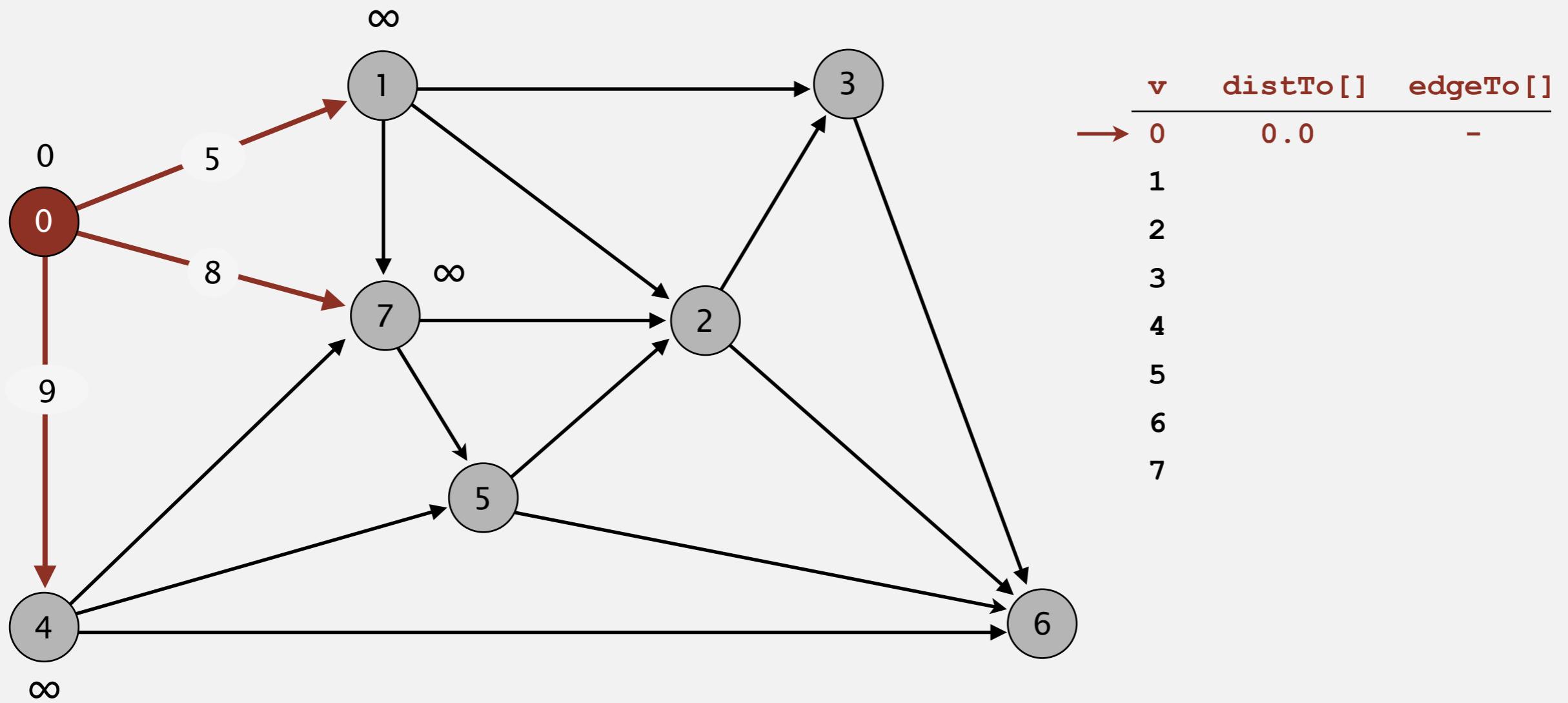
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



choose source vertex 0

Dijkstra's algorithm

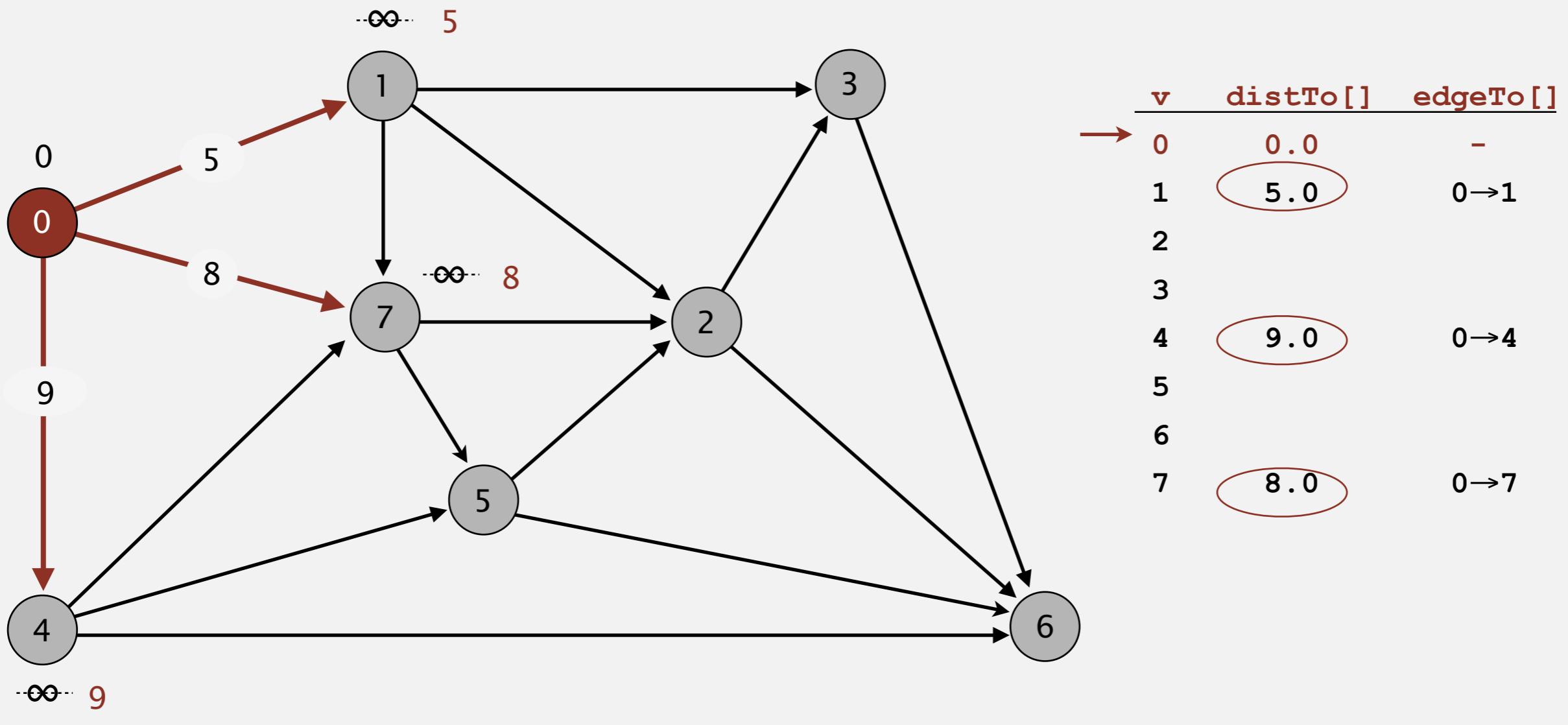
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 0

Dijkstra's algorithm

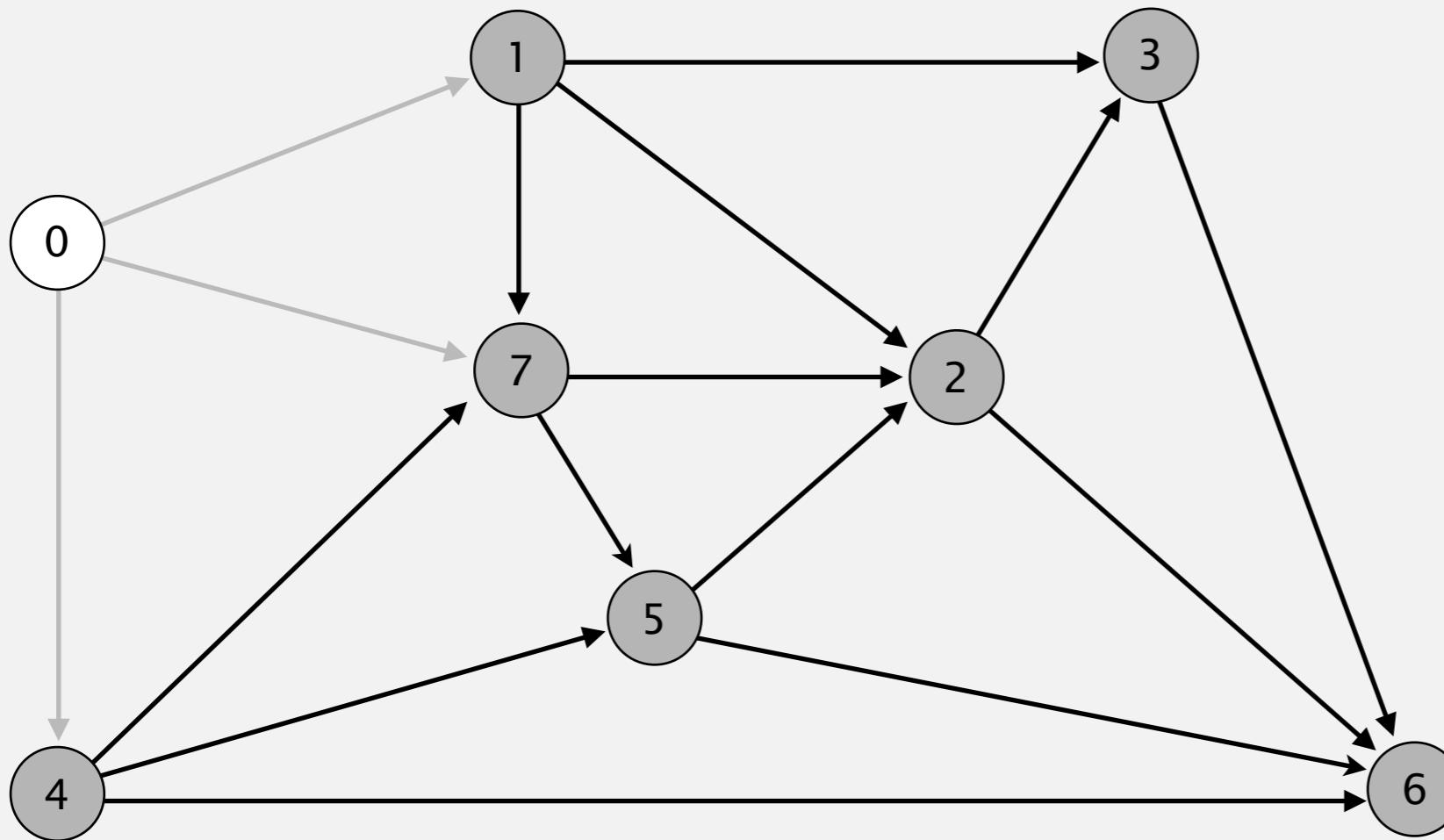
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 0

Dijkstra's algorithm

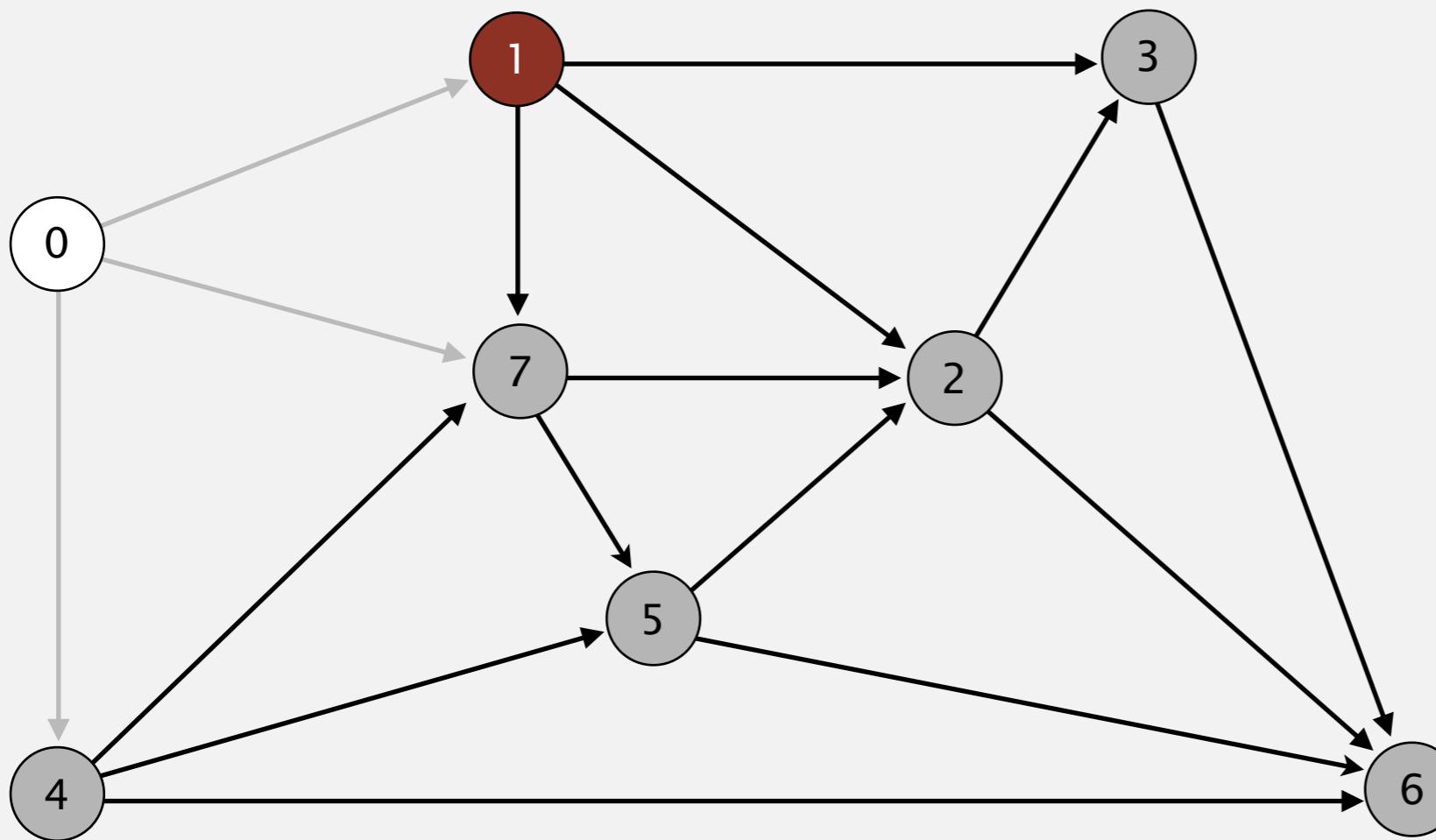
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.

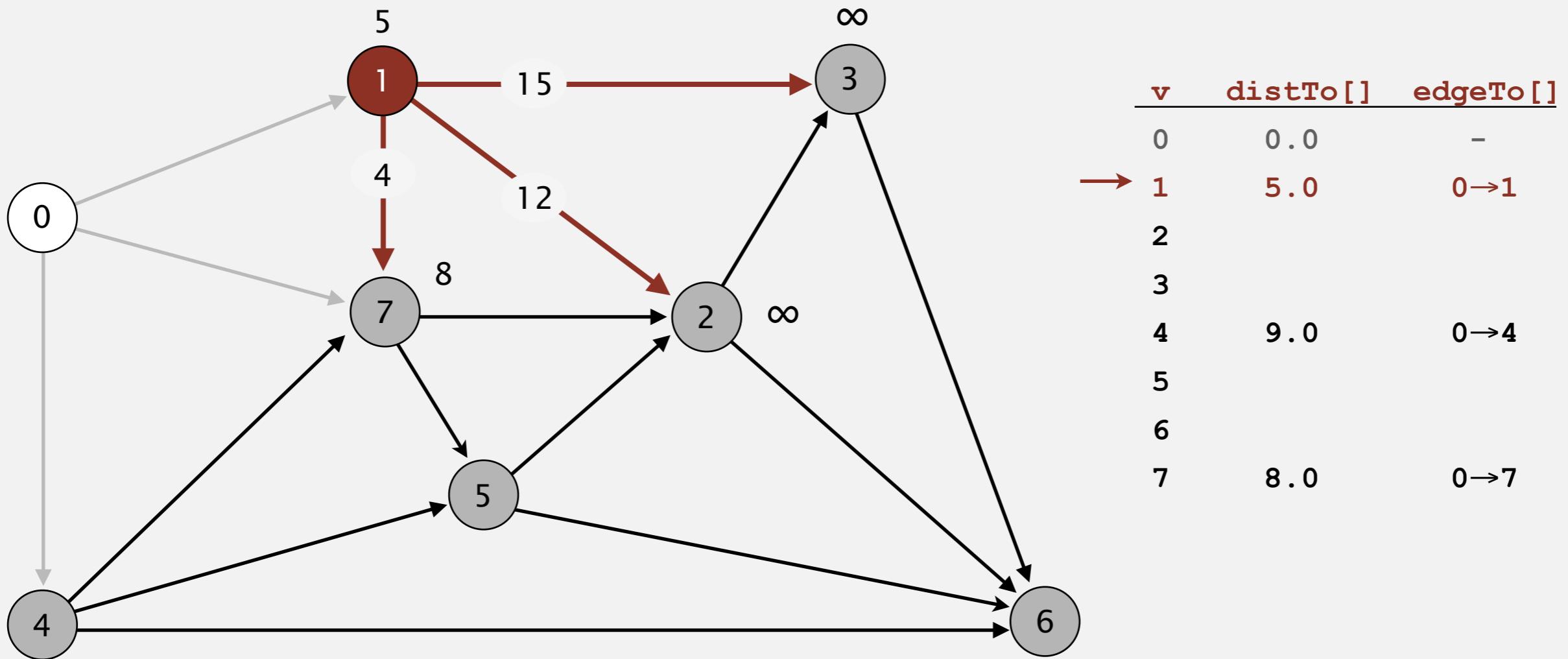


choose vertex 1

v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Dijkstra's algorithm

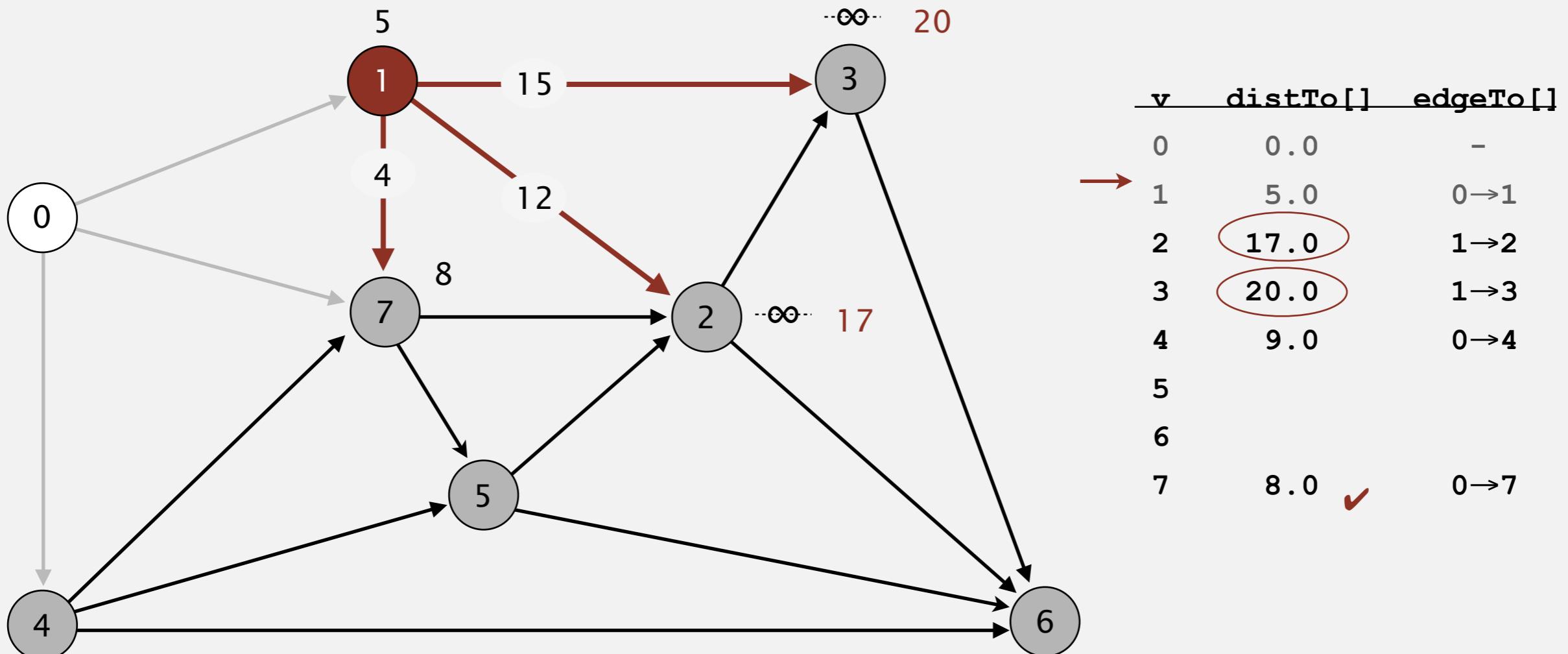
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 1

Dijkstra's algorithm

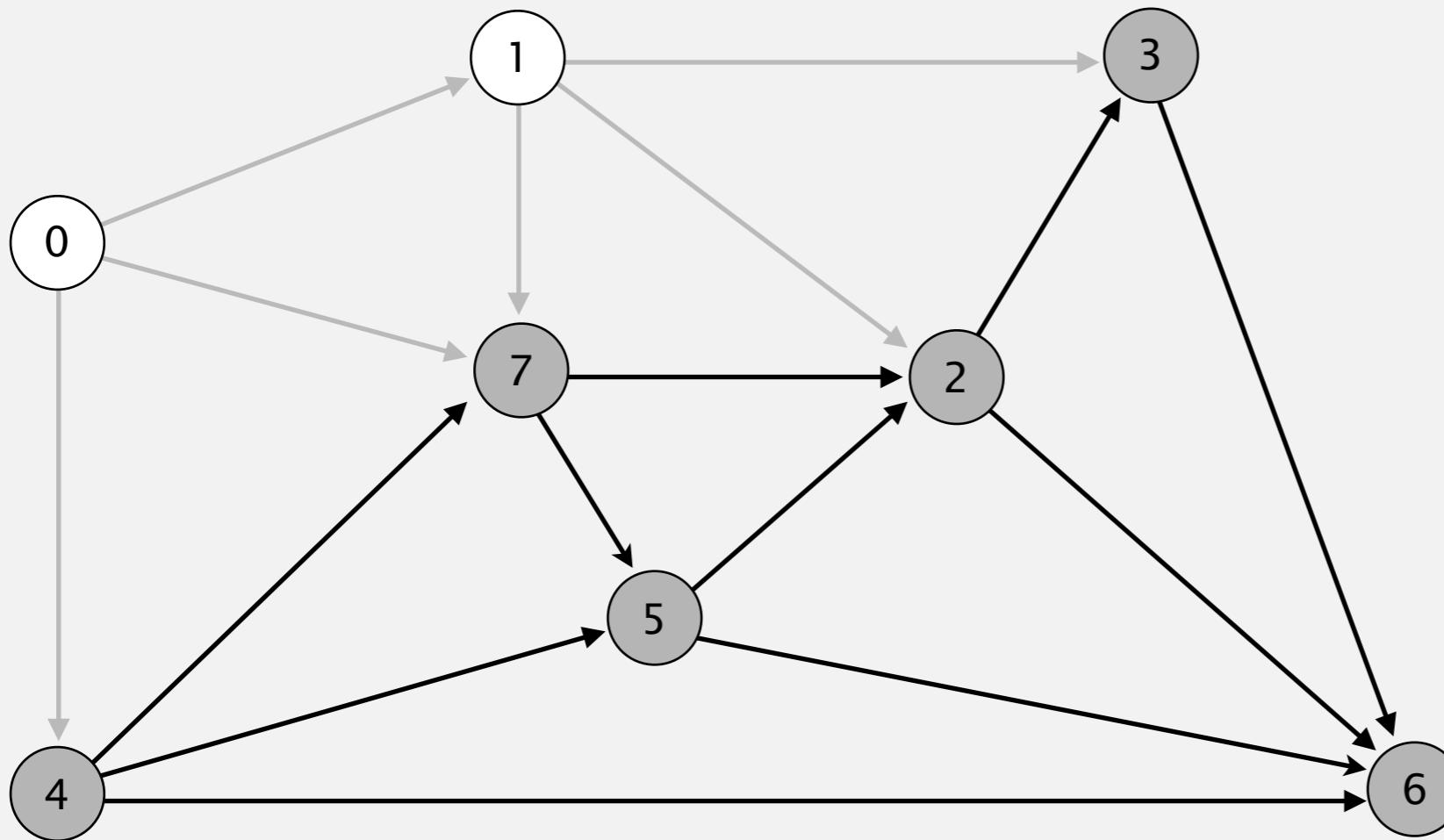
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 1

Dijkstra's algorithm

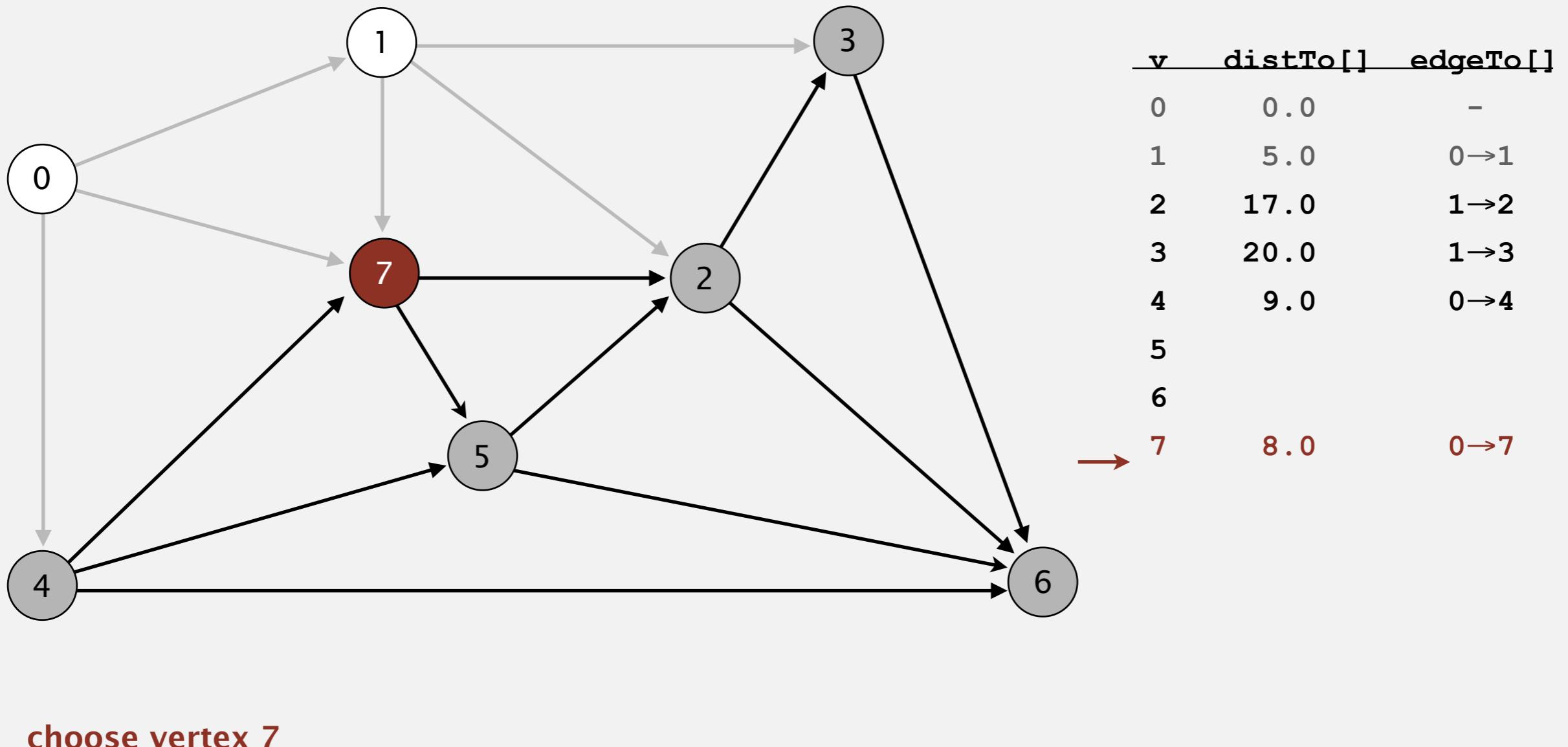
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

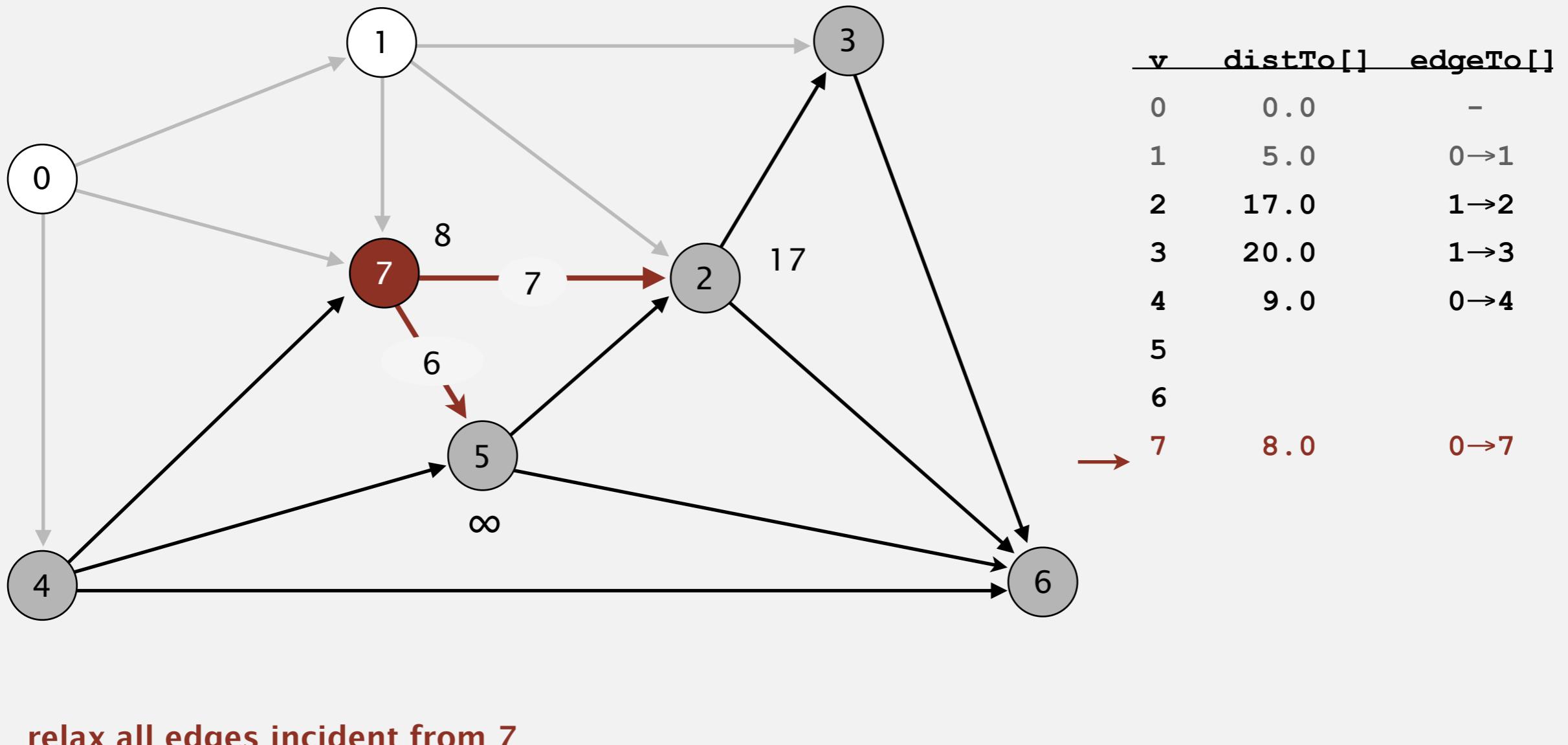
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



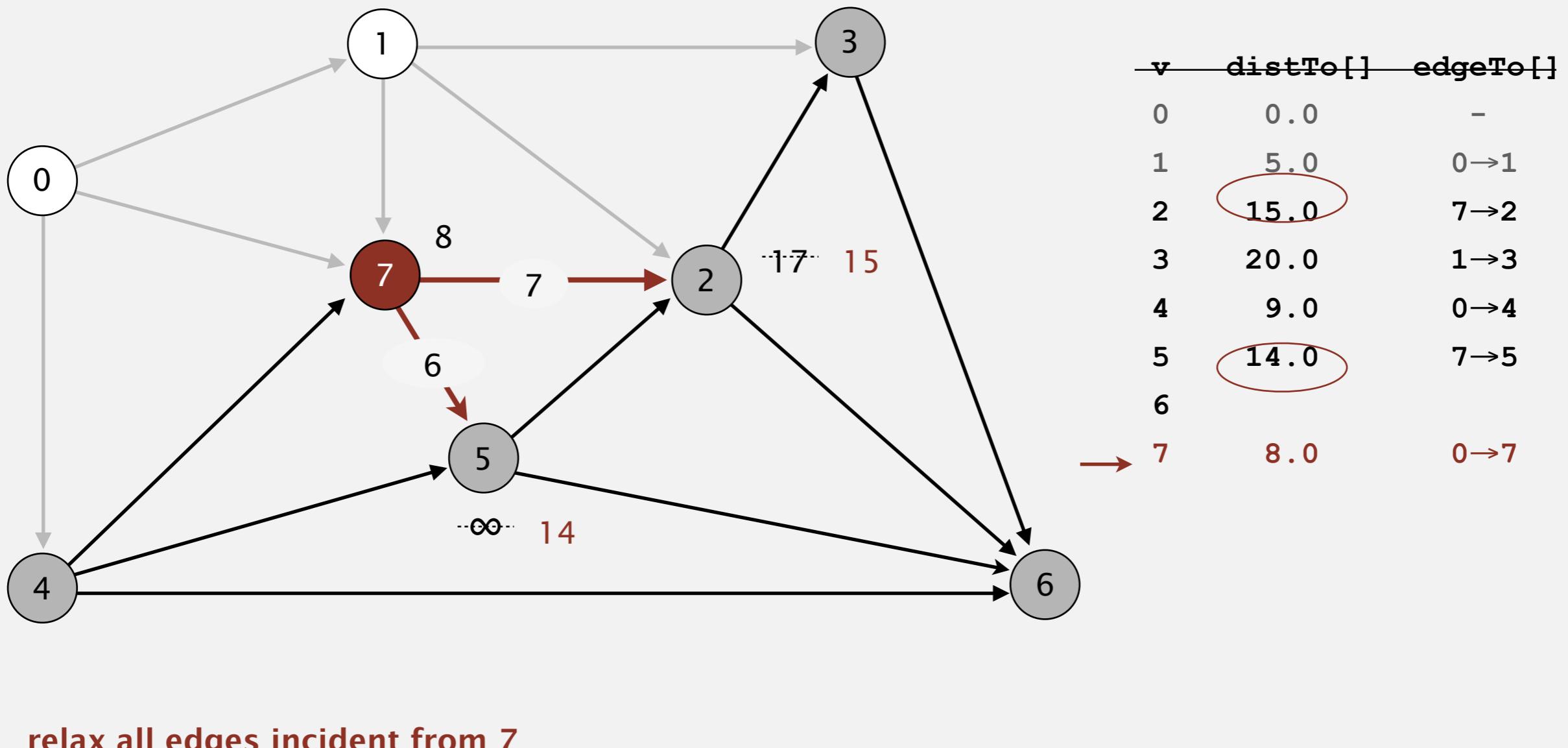
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



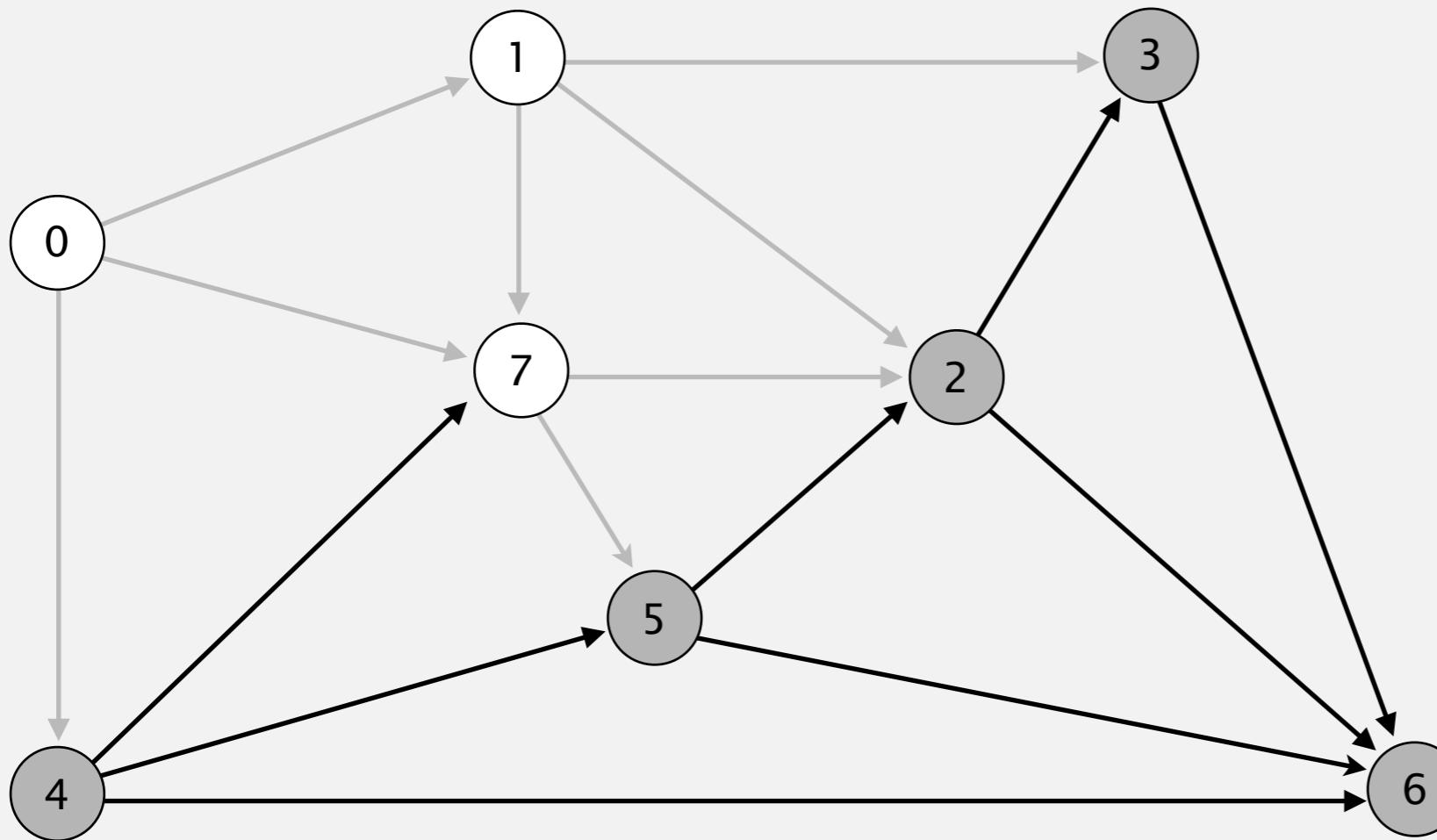
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



Dijkstra's algorithm

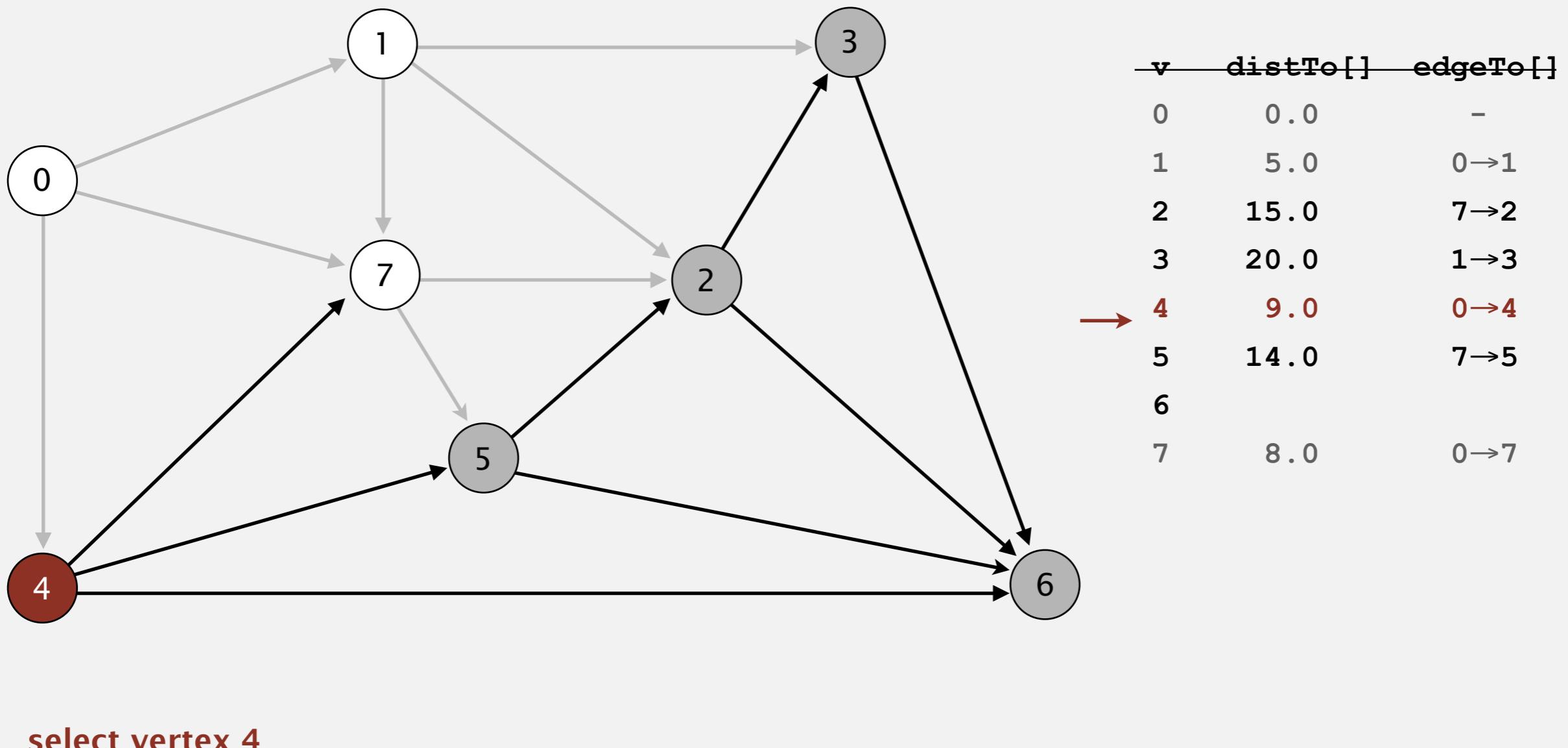
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	14.0	7→5
6		
7	8.0	0→7

Dijkstra's algorithm

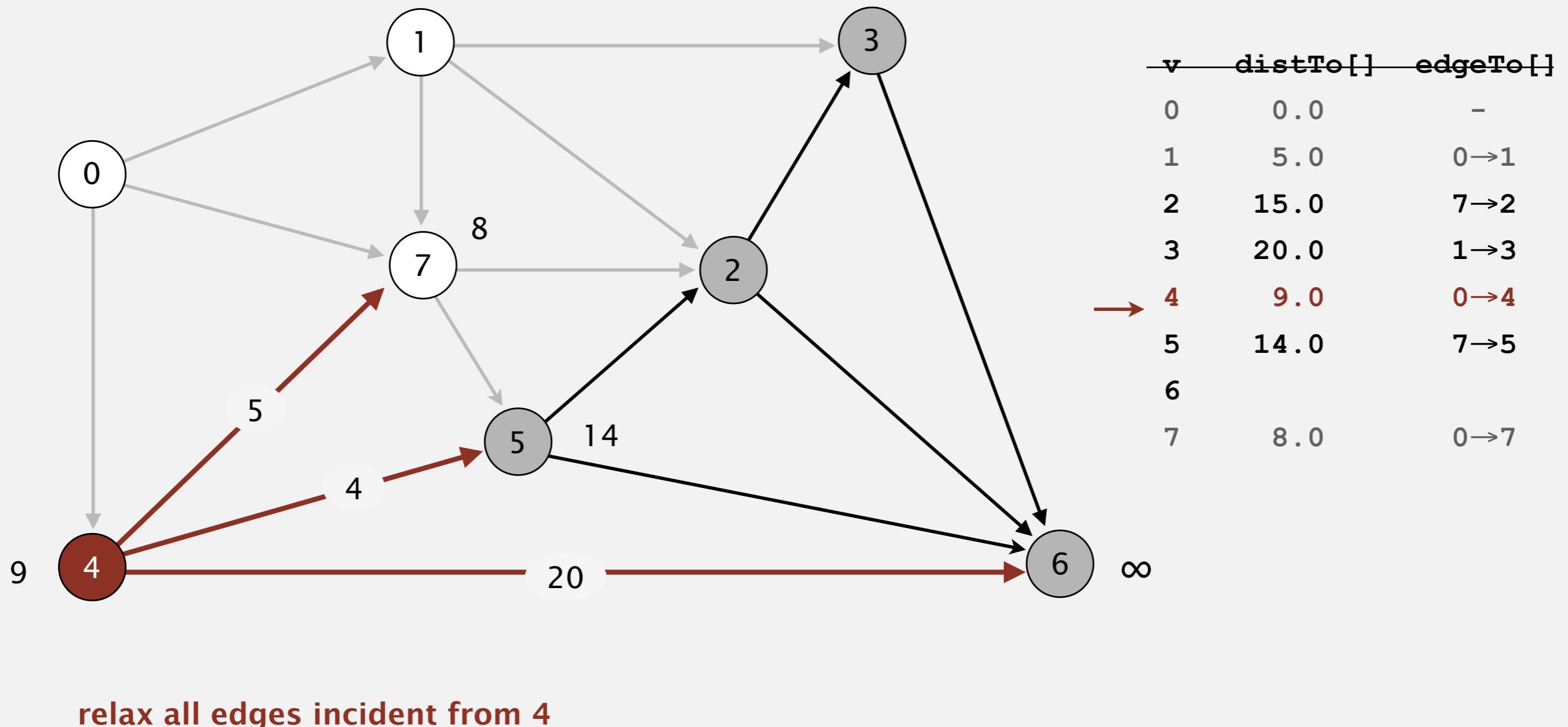
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



select vertex 4

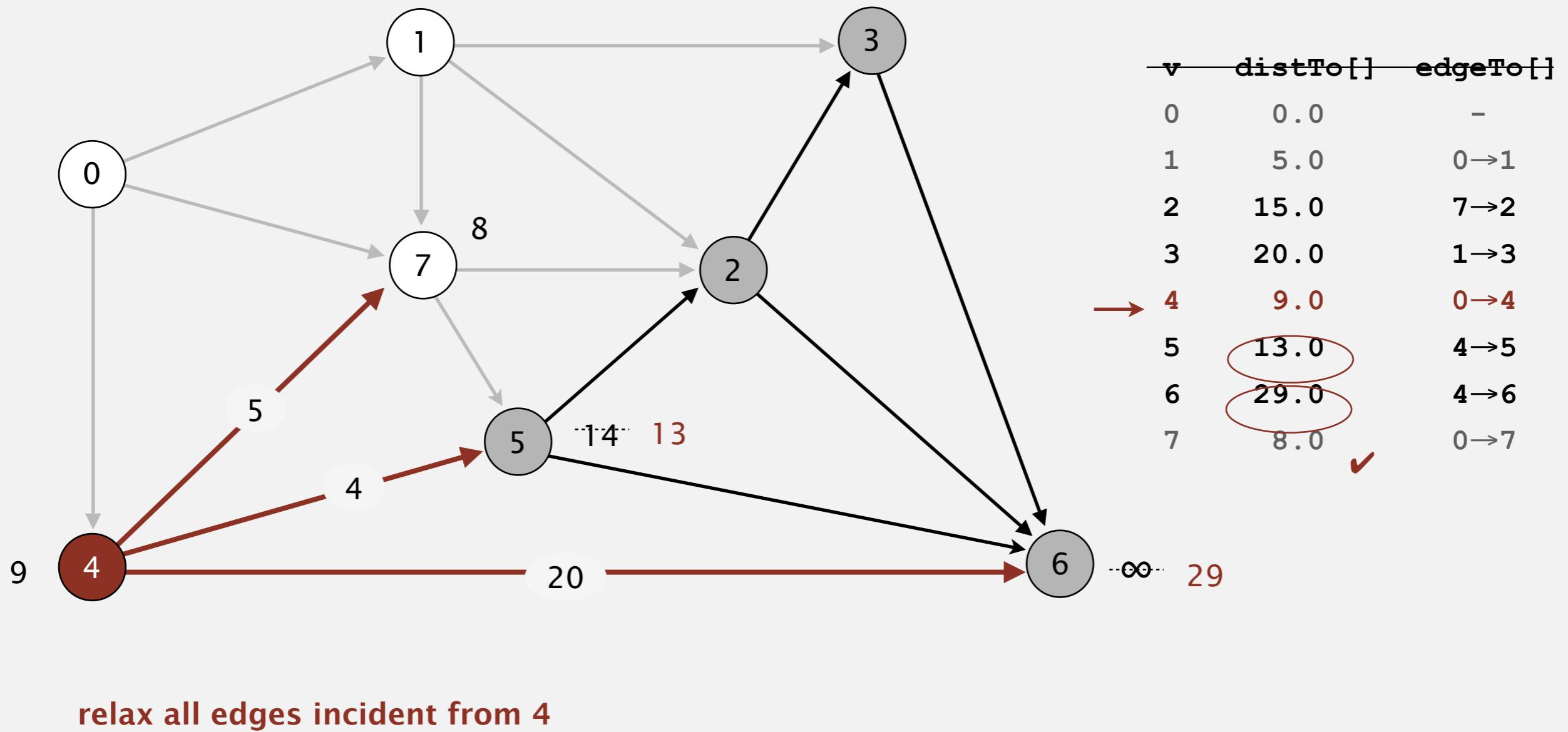
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



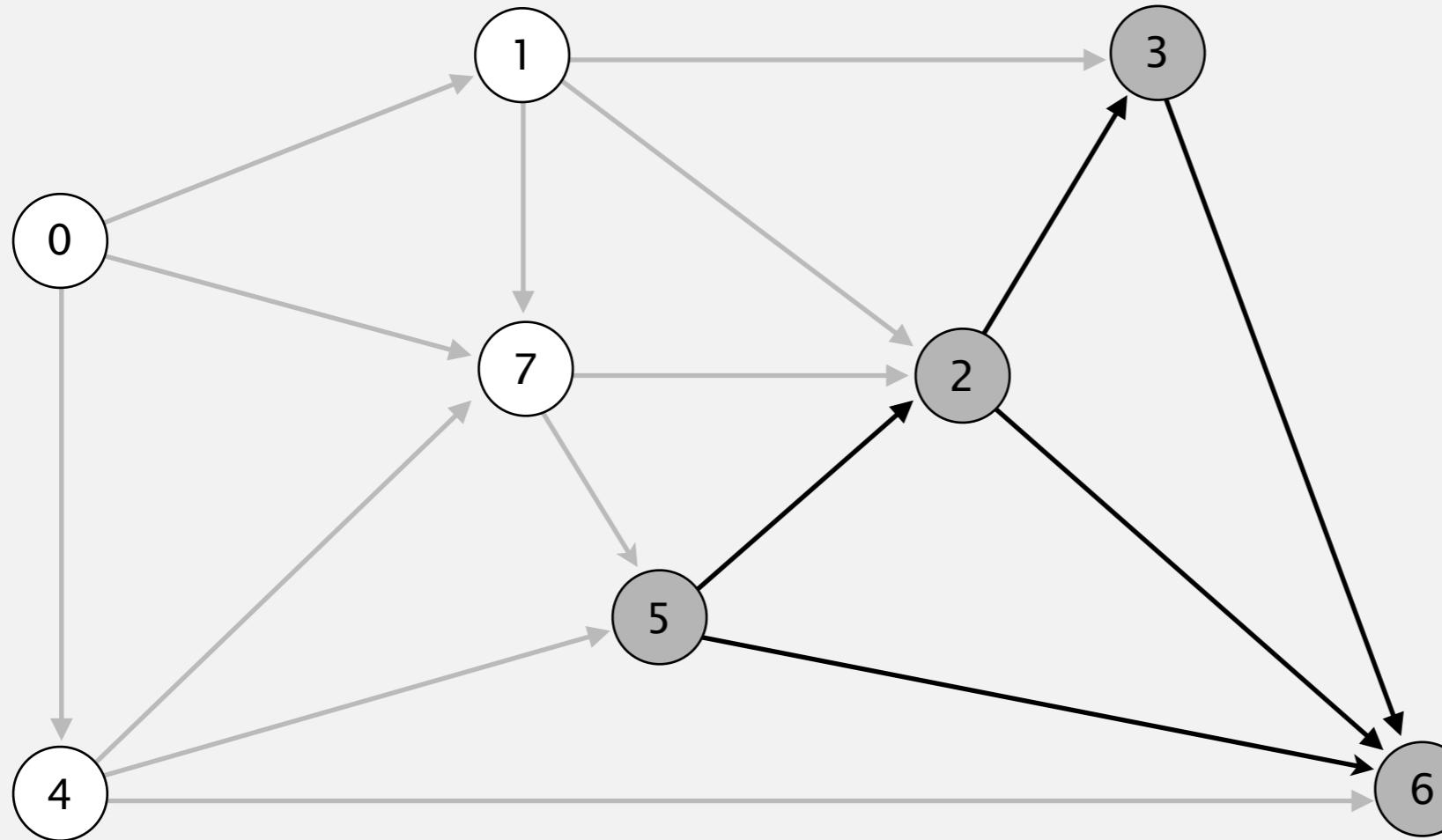
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



Dijkstra's algorithm

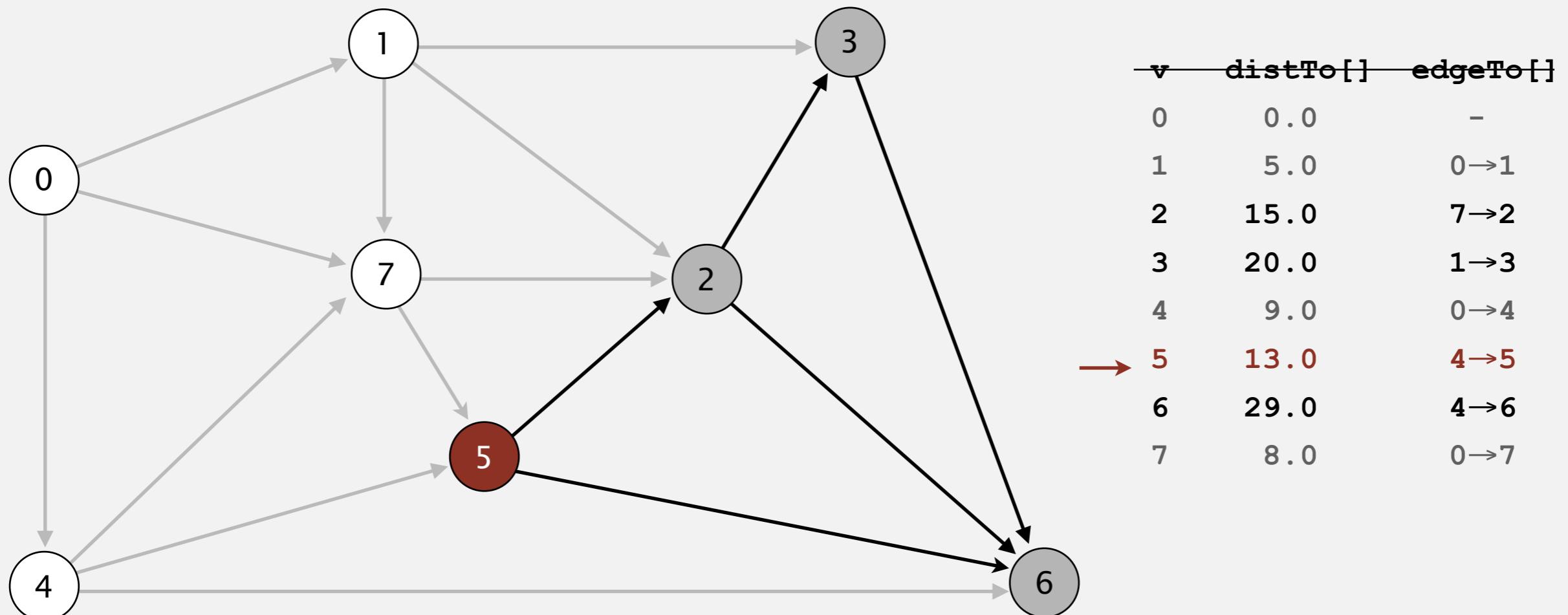
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Dijkstra's algorithm

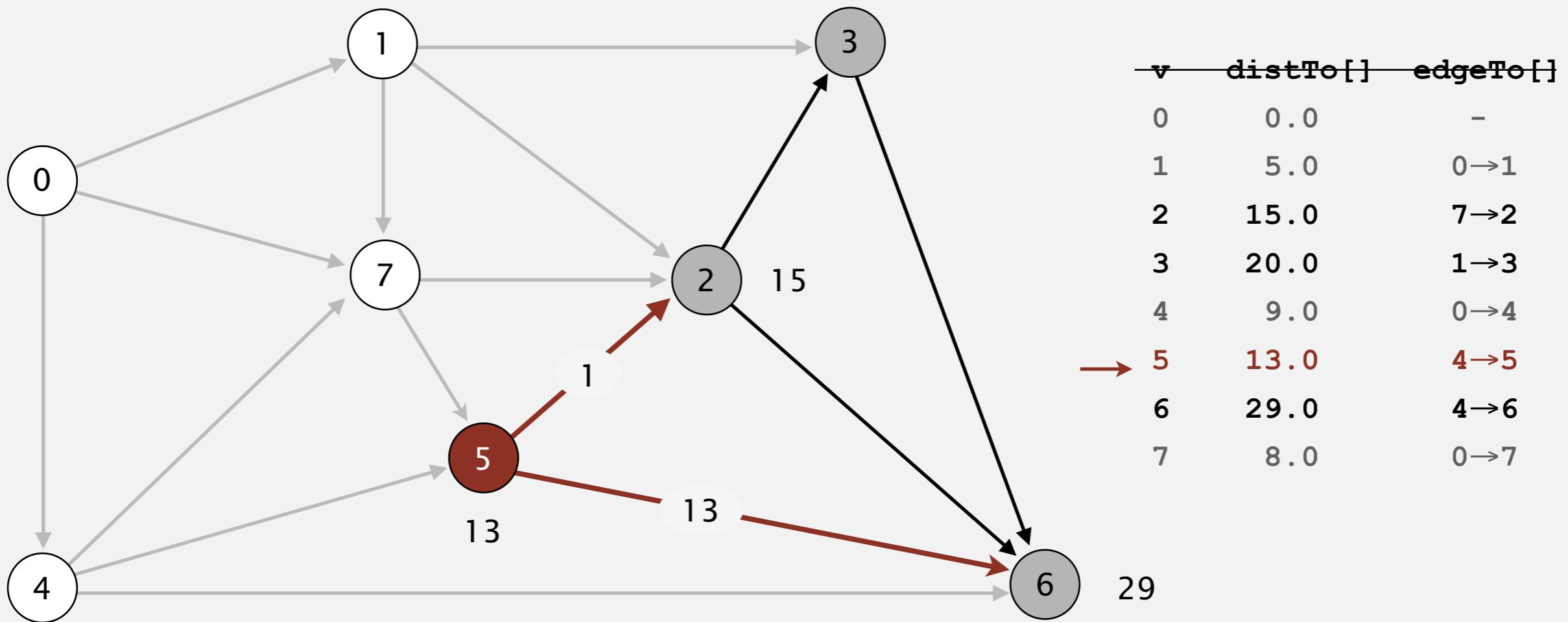
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



select vertex 5

Dijkstra's algorithm

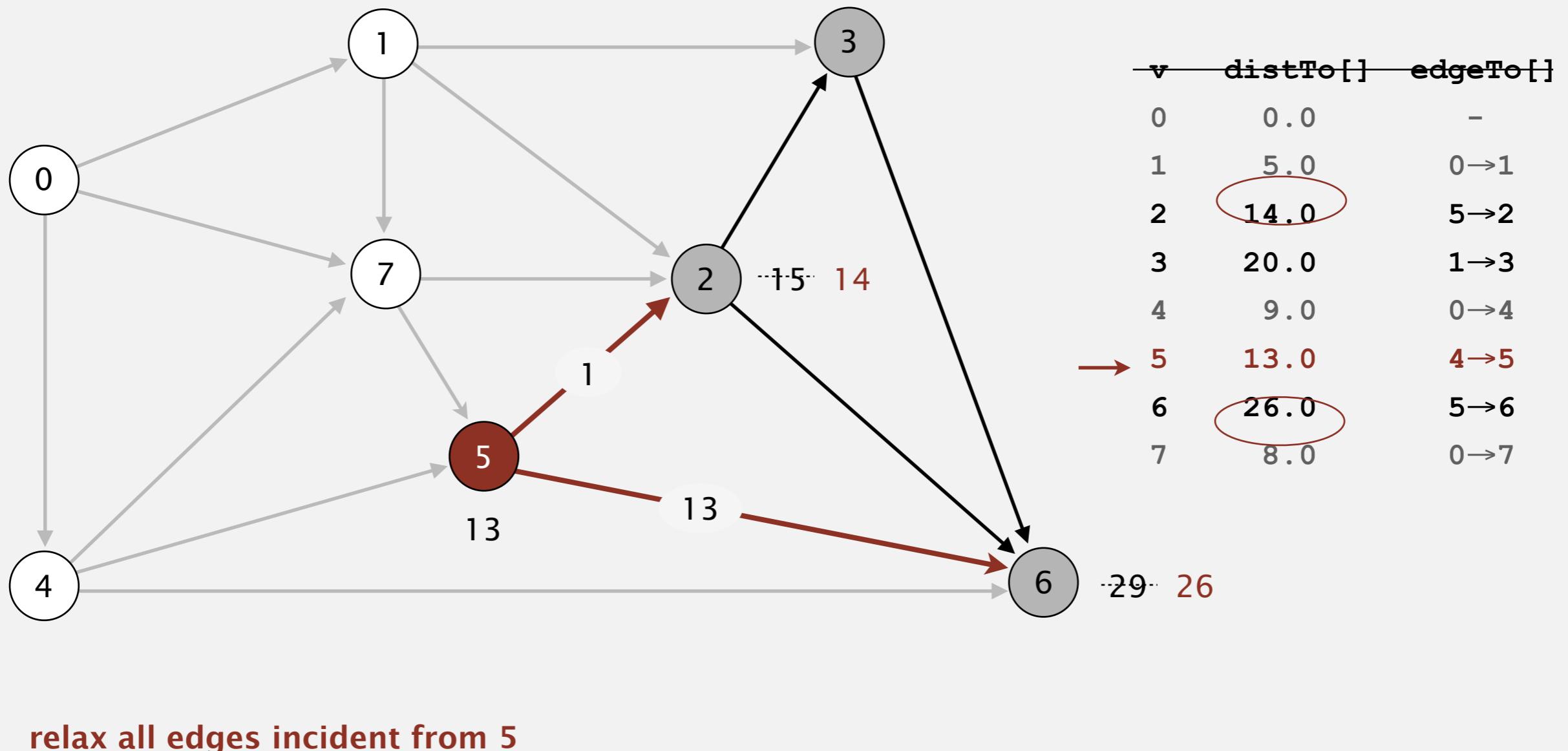
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 5

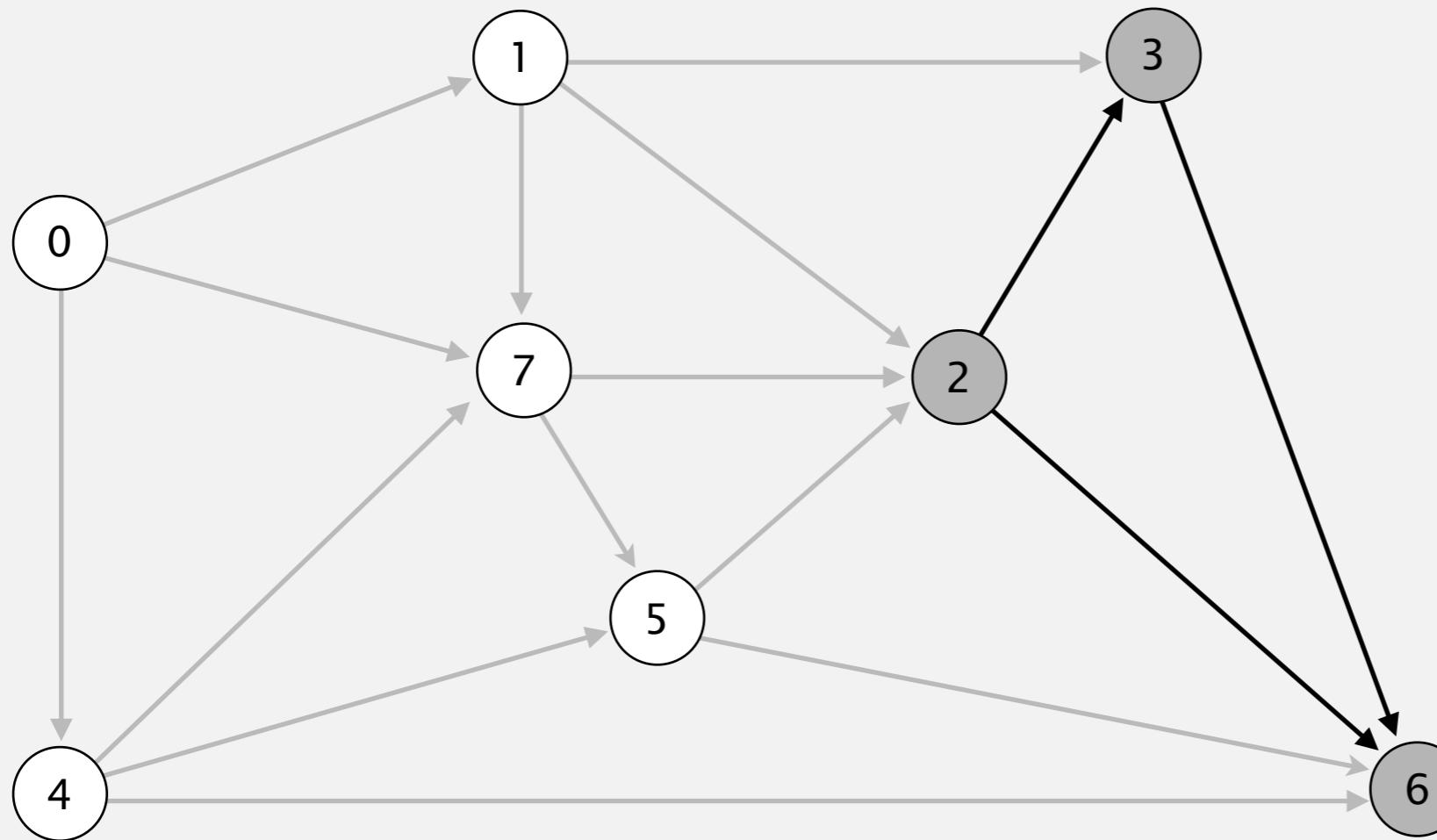
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



Dijkstra's algorithm

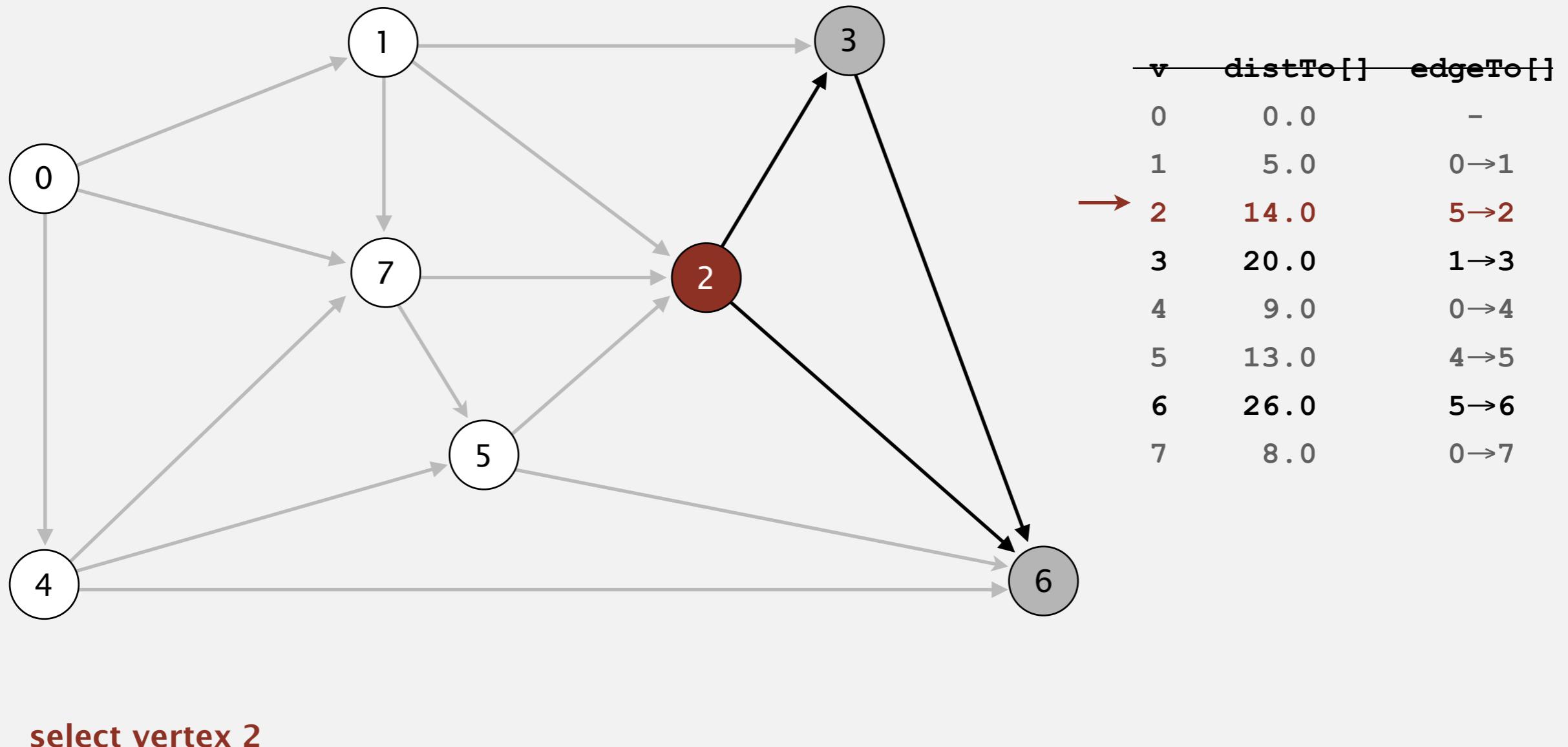
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

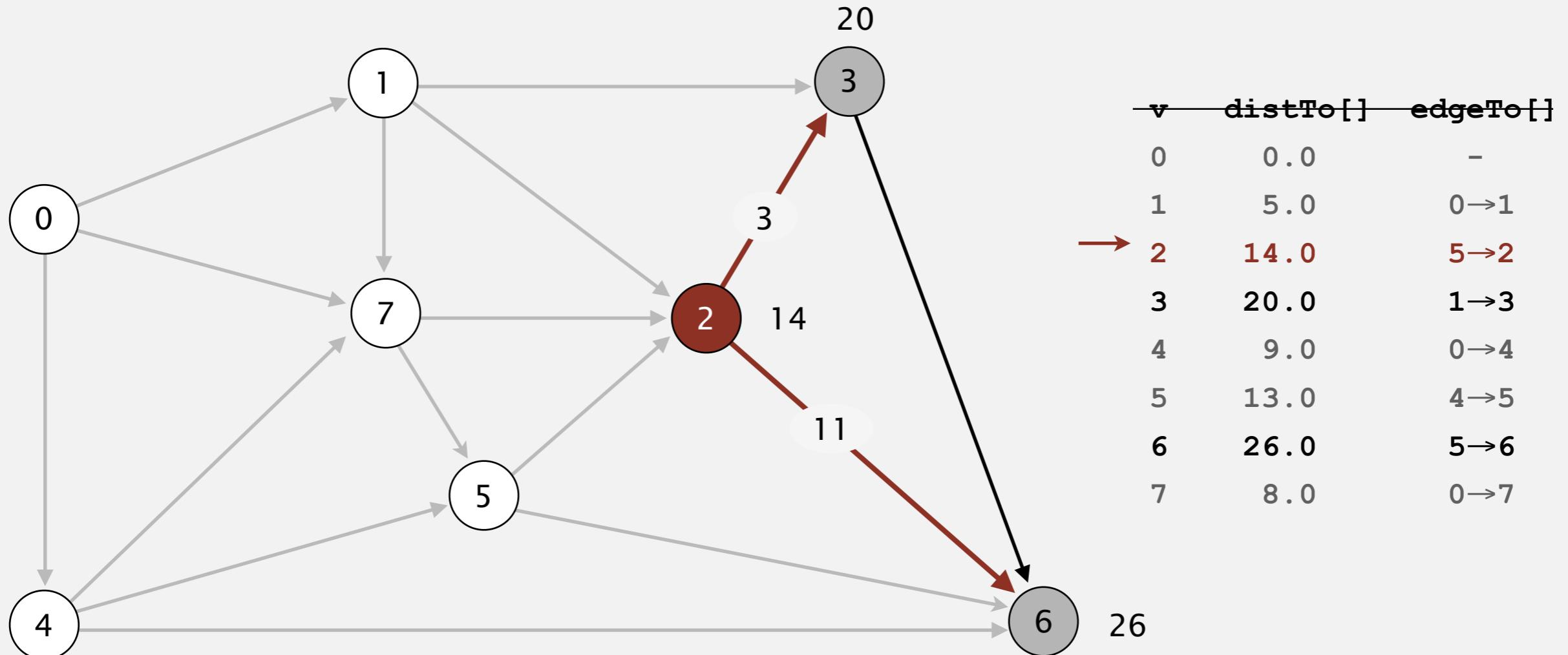
Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



Dijkstra's algorithm

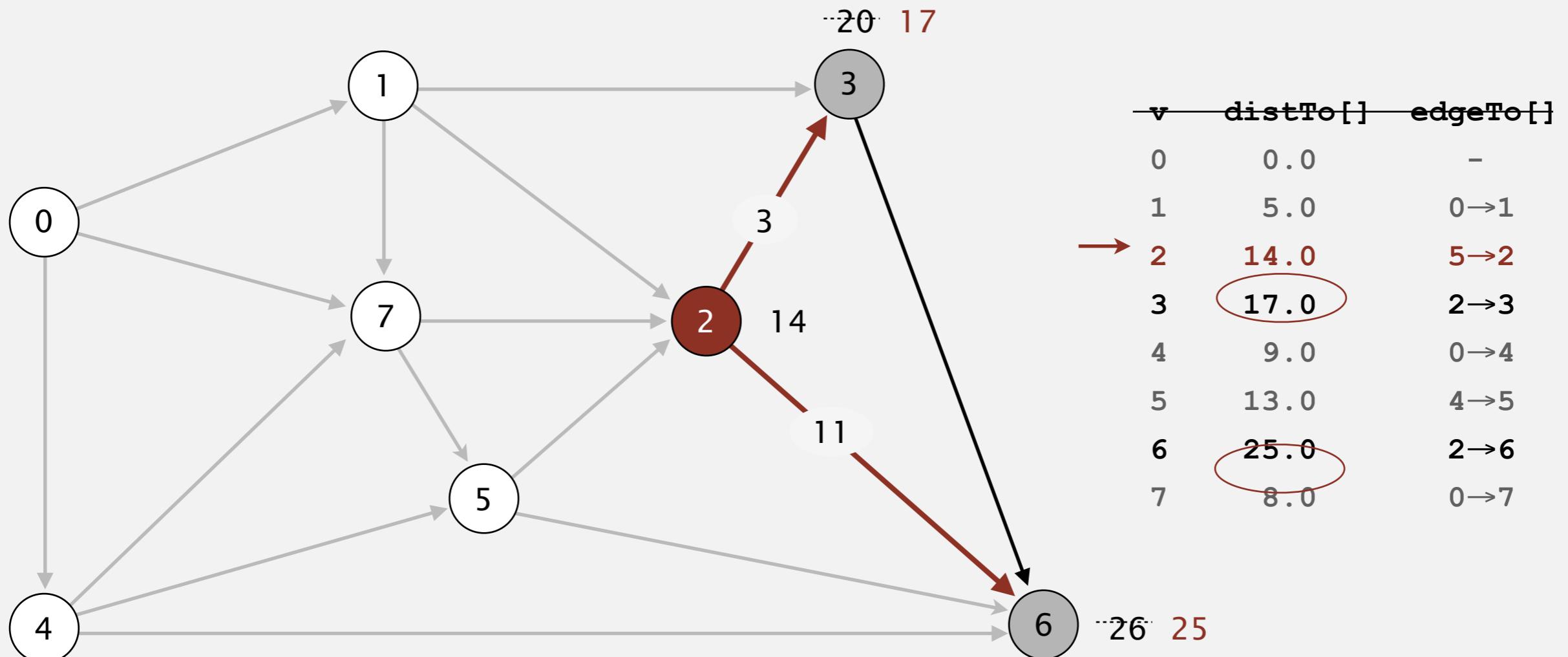
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 2

Dijkstra's algorithm

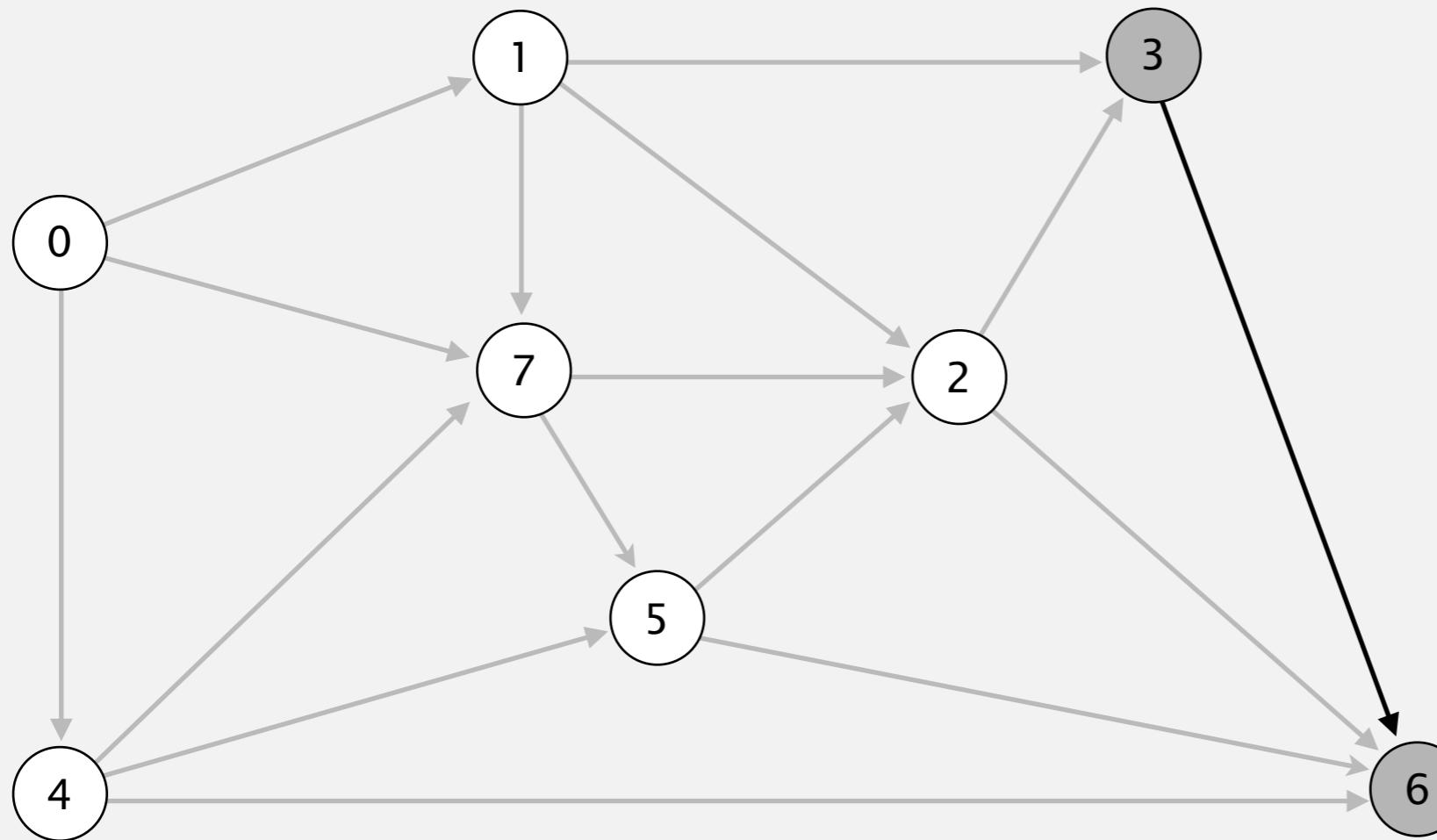
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 2

Dijkstra's algorithm

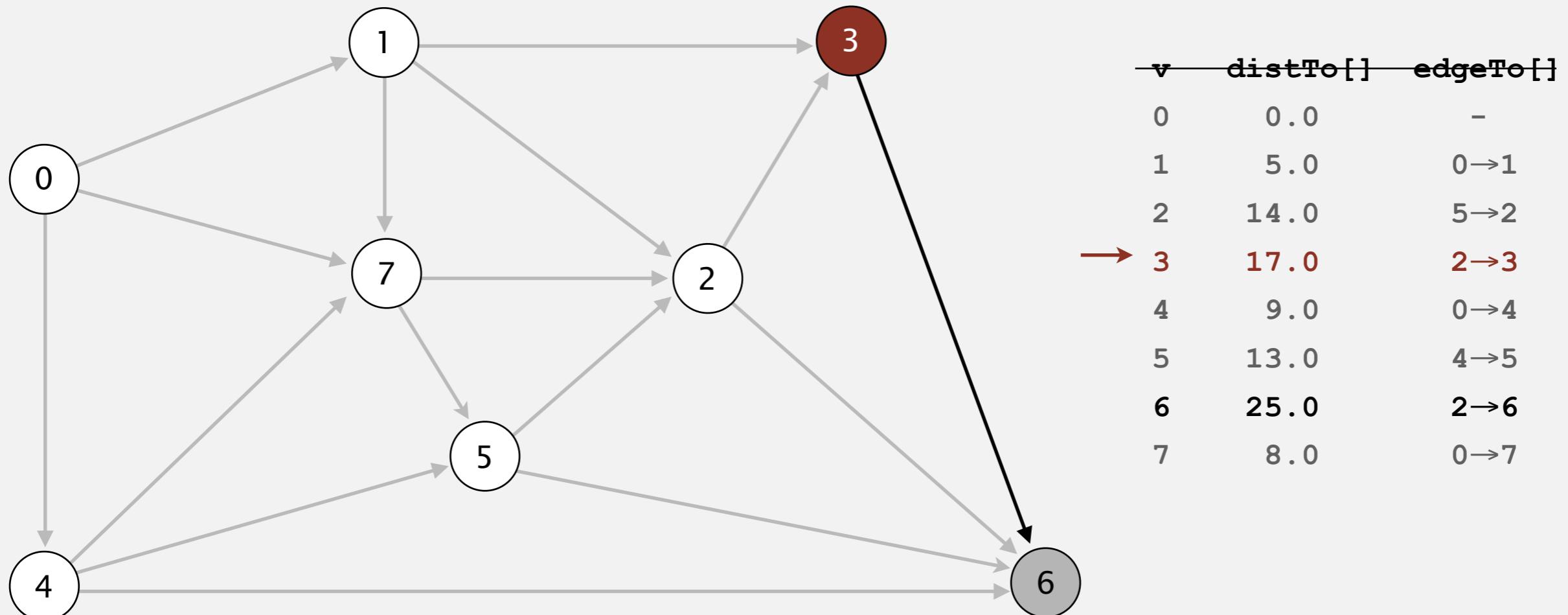
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm

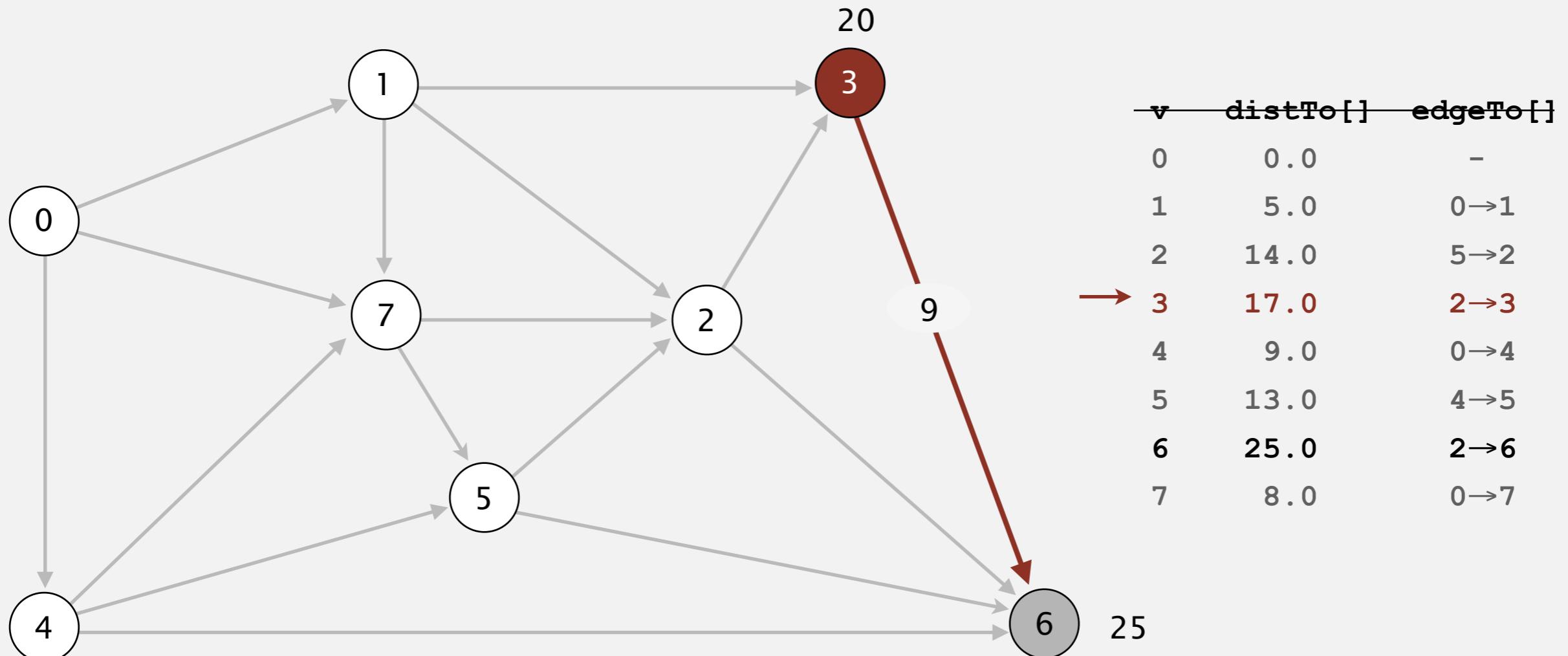
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



select vertex 3

Dijkstra's algorithm

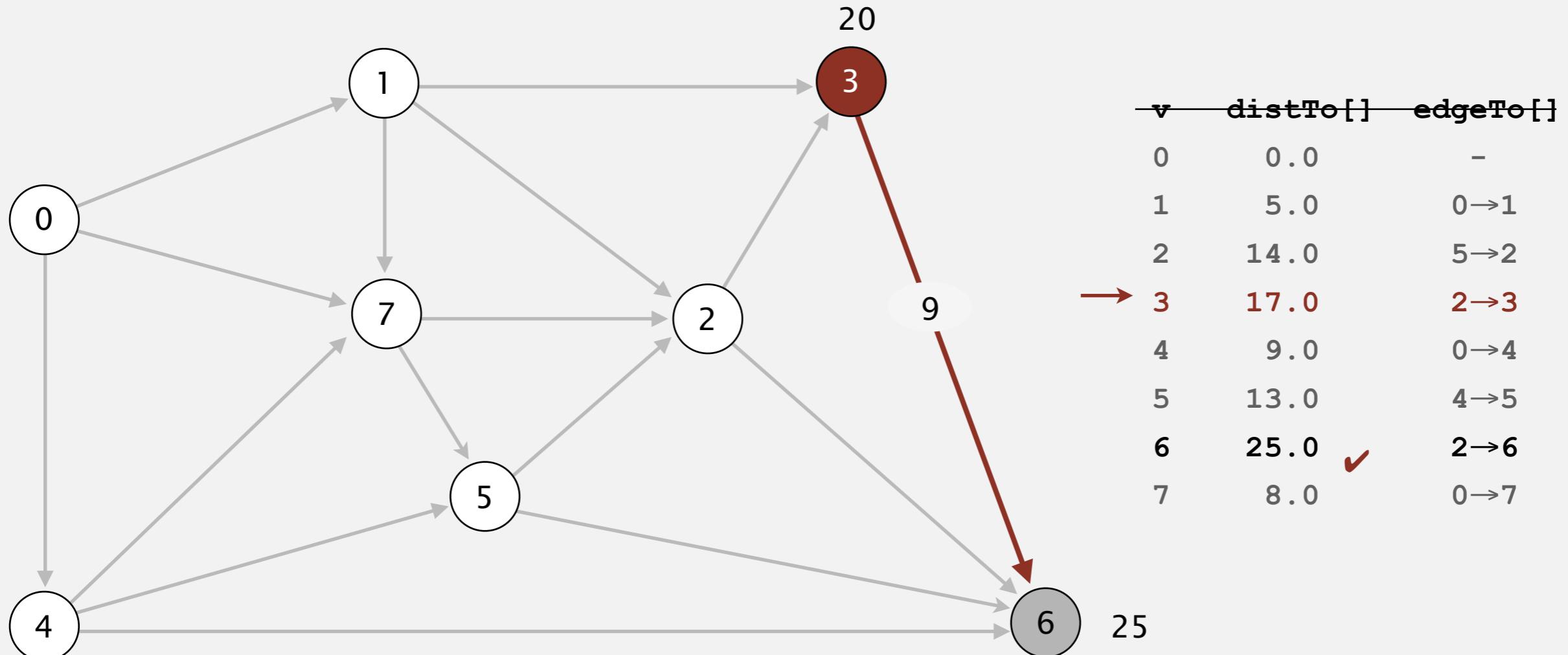
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 3

Dijkstra's algorithm

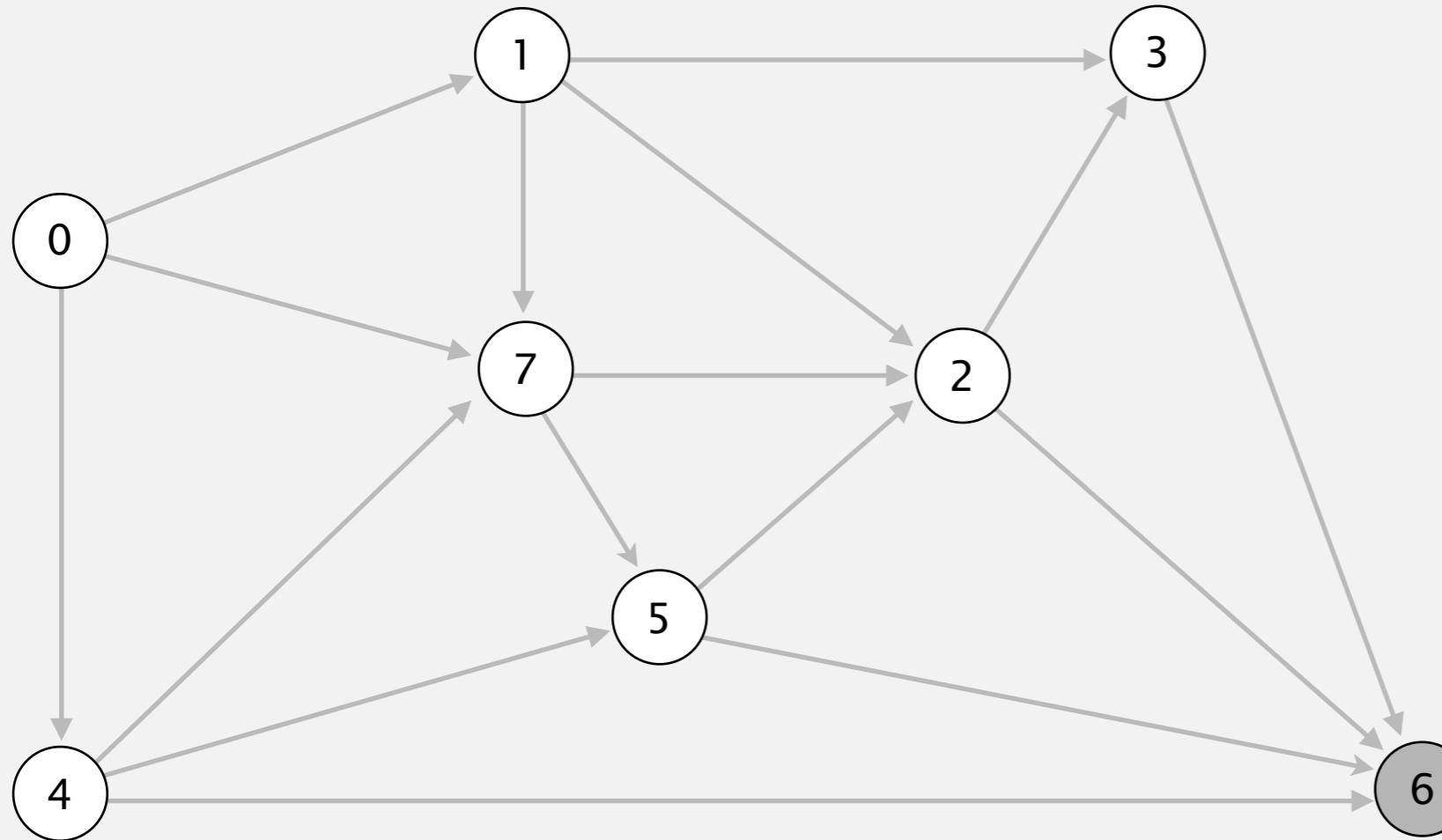
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 3

Dijkstra's algorithm

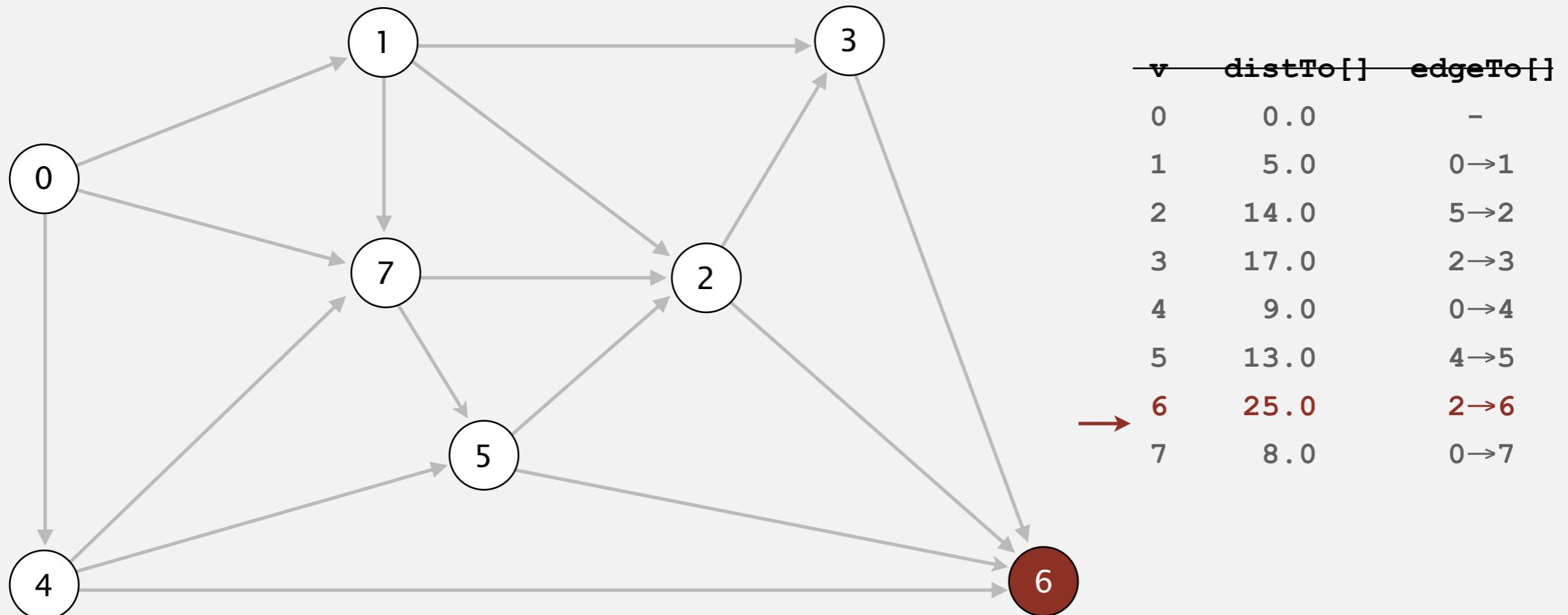
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm

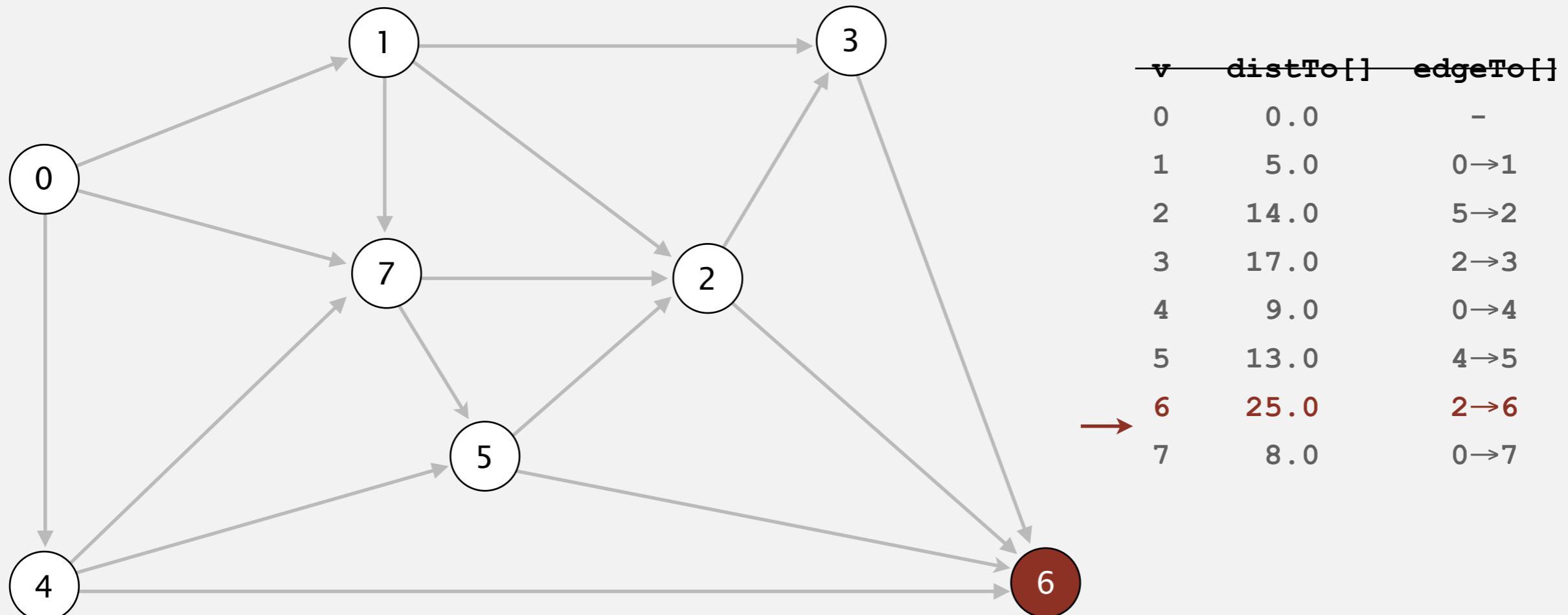
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



select vertex 6

Dijkstra's algorithm

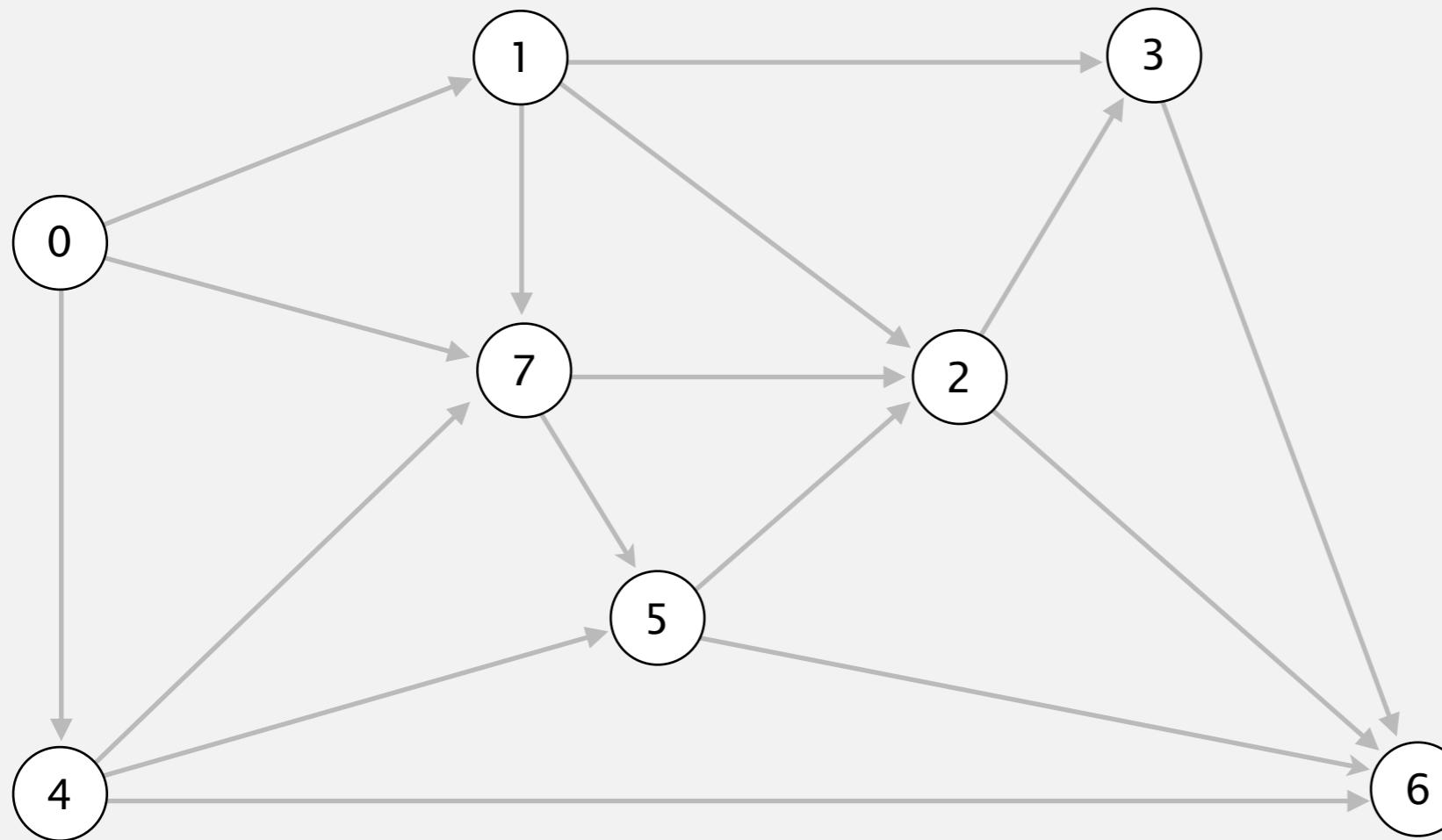
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



relax all edges incident from 6

Dijkstra's algorithm

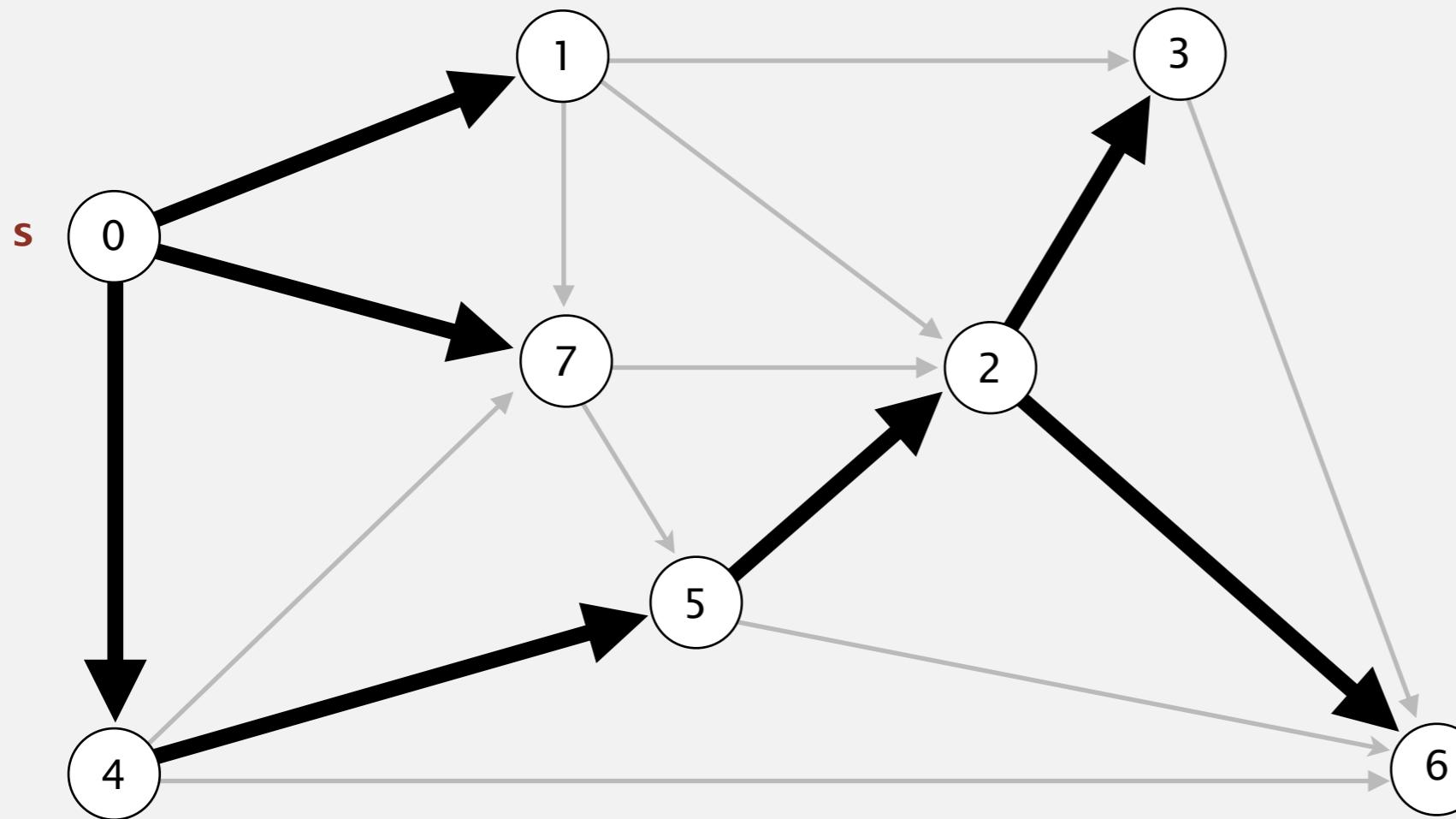
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm

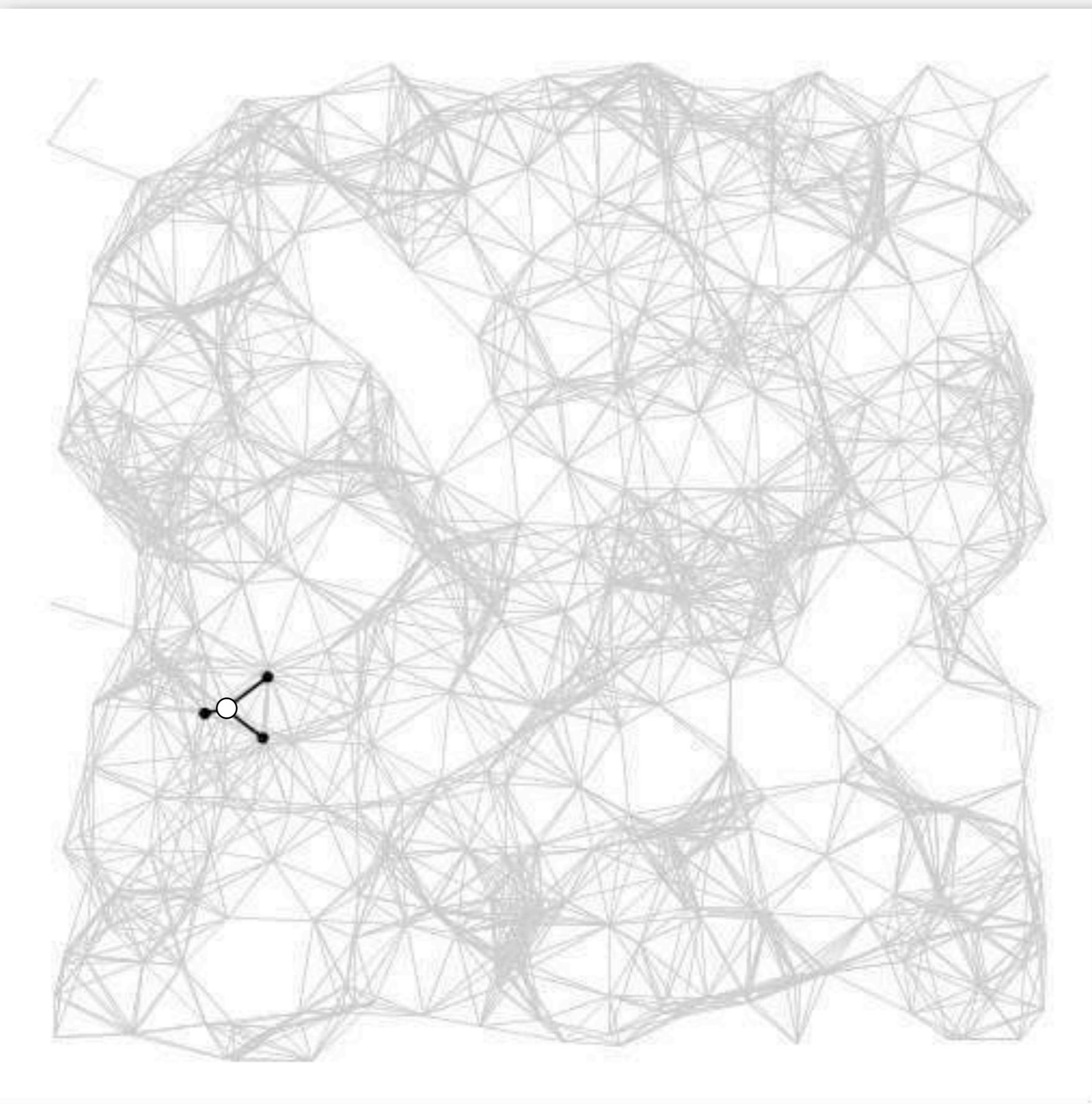
- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest `distTo[]` value).
- Add vertex to tree and relax all edges incident from that vertex.



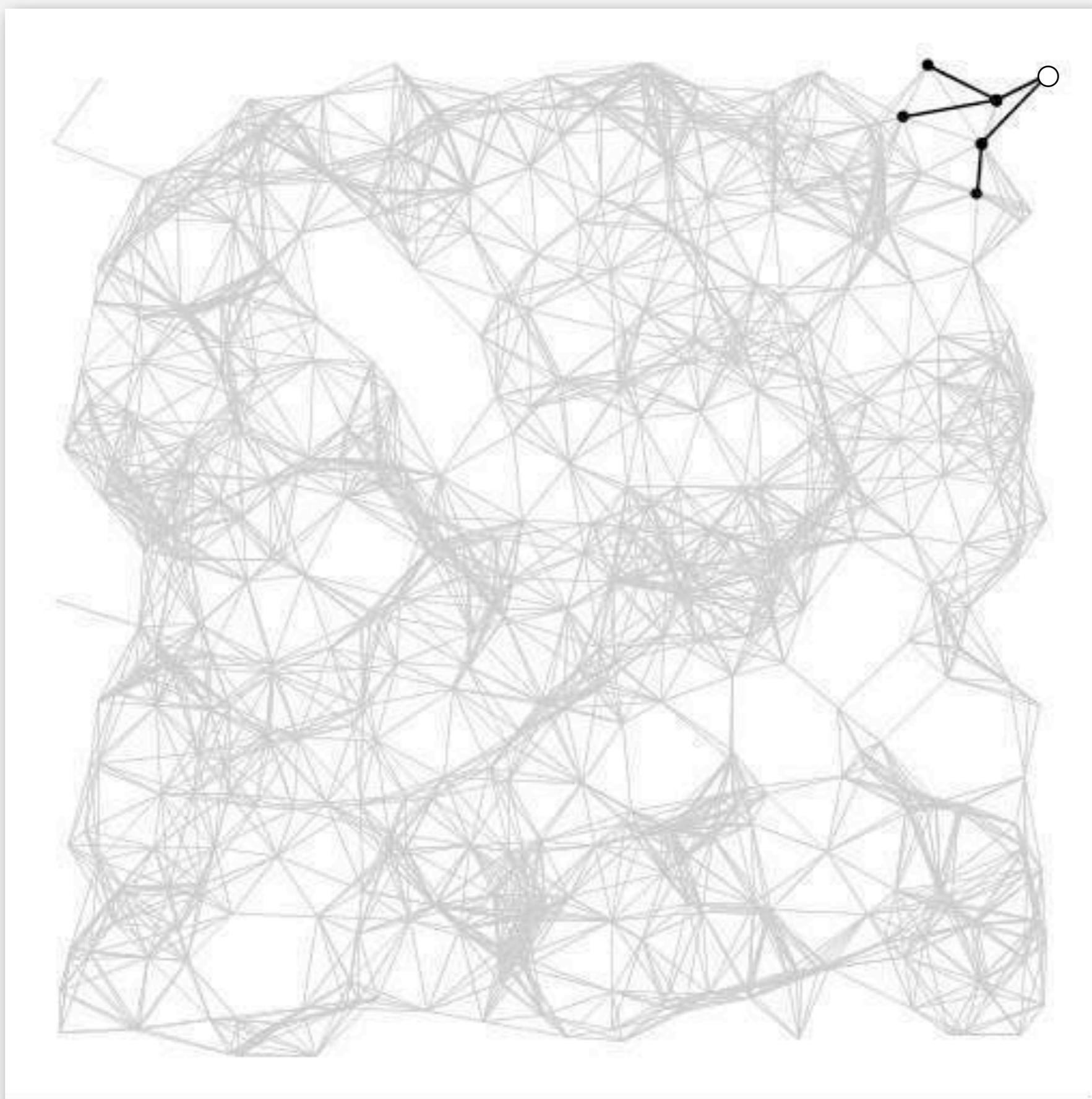
shortest-paths tree from vertex s

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Dijkstra's algorithm visualization



Dijkstra's algorithm visualization



Dijkstra's algorithm: correctness proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when v is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.
- Inequality holds until algorithm terminates because:
 - $\text{distTo}[w]$ cannot increase \leftarrow $\text{distTo}[]$ values are monotone decreasing
 - $\text{distTo}[v]$ will not change \leftarrow edge weights are nonnegative and we choose lowest $\text{distTo}[]$ value at each step
- Thus, upon termination, shortest-paths optimality conditions hold. ■

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());
        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        pq.insert(s, 0.0);
        while (!pq.isEmpty())
        {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }

    private void relax(DirectedEdge e)
    {
        if (distTo[e.to] > distTo[e.from] + e.weight)
        {
            distTo[e.to] = distTo[e.from] + e.weight;
            edgeTo[e.to] = e;
            pq.update(e.to, distTo[e.to]);
        }
    }
}
```

←
relax vertices in order
of distance from s

Dijkstra's algorithm: Java implementation

```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else                  pq.insert      (w, distTo[w]);
    }
}
```



update PQ

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap (Johnson 1975)	$d \log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	1^\dagger	$\log V^\dagger$	1^\dagger	$E + V \log V$

\dagger amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- d-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Priority-first search

Insight. Four of our graph-search methods are the same algorithm!

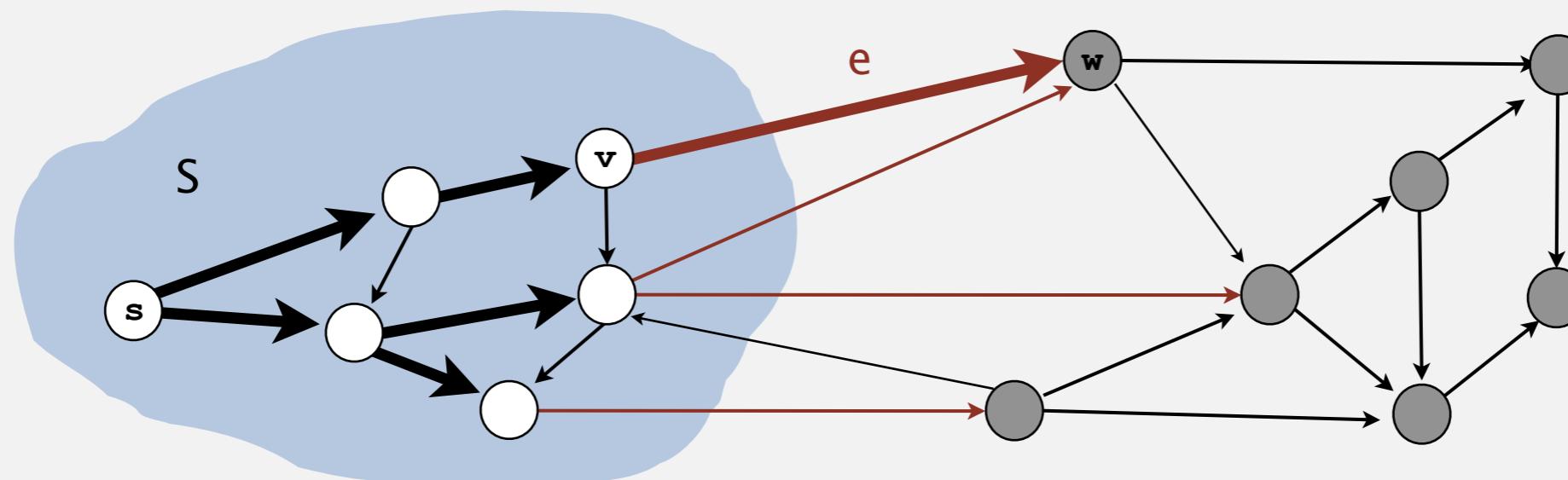
- Maintain a set of explored vertices S .
- Grow S by exploring edges with exactly one endpoint leaving S .

DFS. Take edge from vertex which was discovered most recently.

BFS. Take edge from vertex which was discovered least recently.

Prim. Take edge of minimum weight.

Dijkstra. Take edge to vertex that is closest to S .



Challenge. Express this insight in reusable Java code.

SHORTEST PATHS

- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ Dijkstra's algorithm
- ▶ Edge-weighted DAGs
- ▶ Negative weights

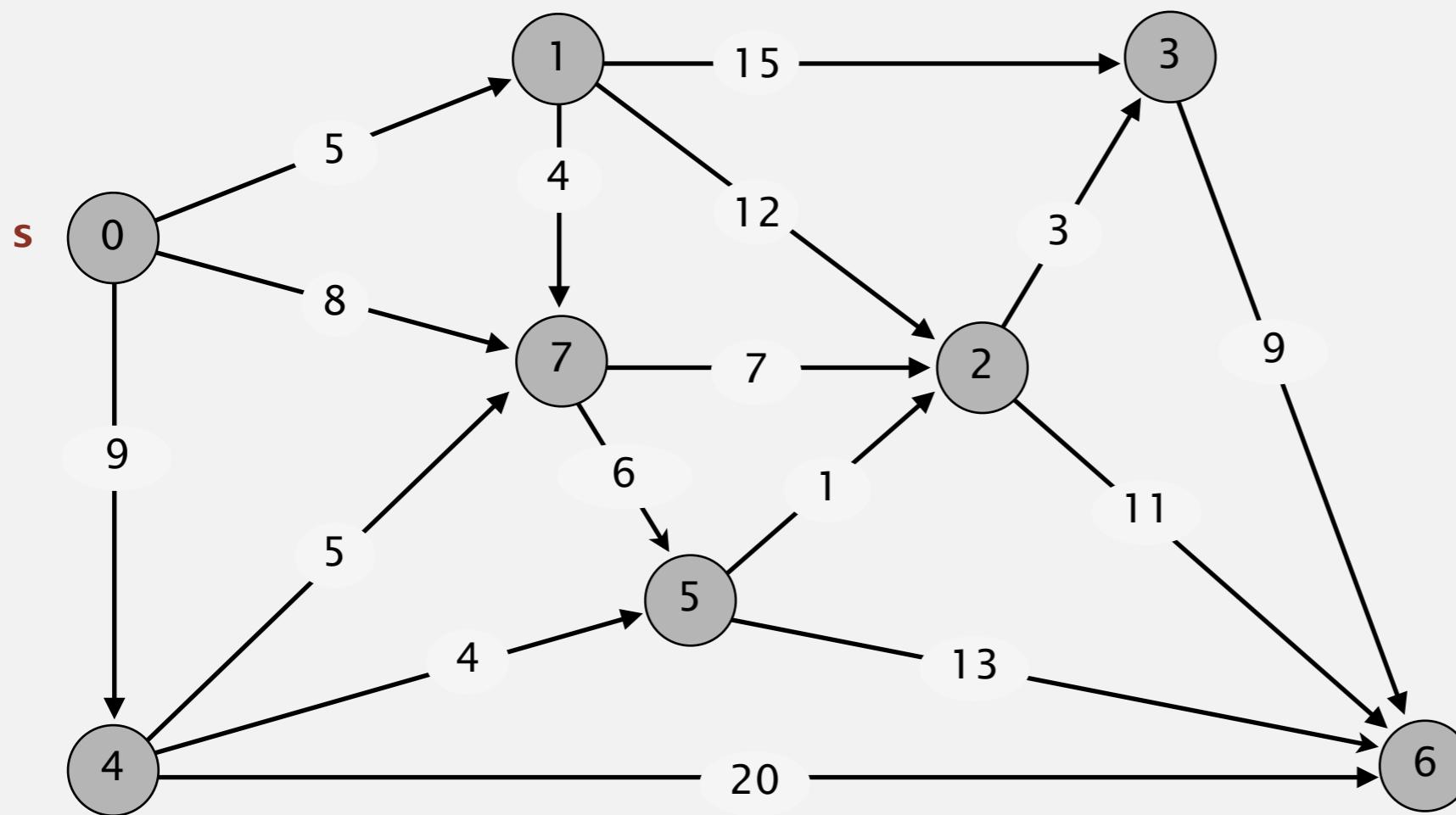
Acyclic edge-weighted digraphs

Q. Suppose that an edge-weighted digraph has no directed cycles.
Is it easier to find shortest paths than in a general digraph?

A. Yes!

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

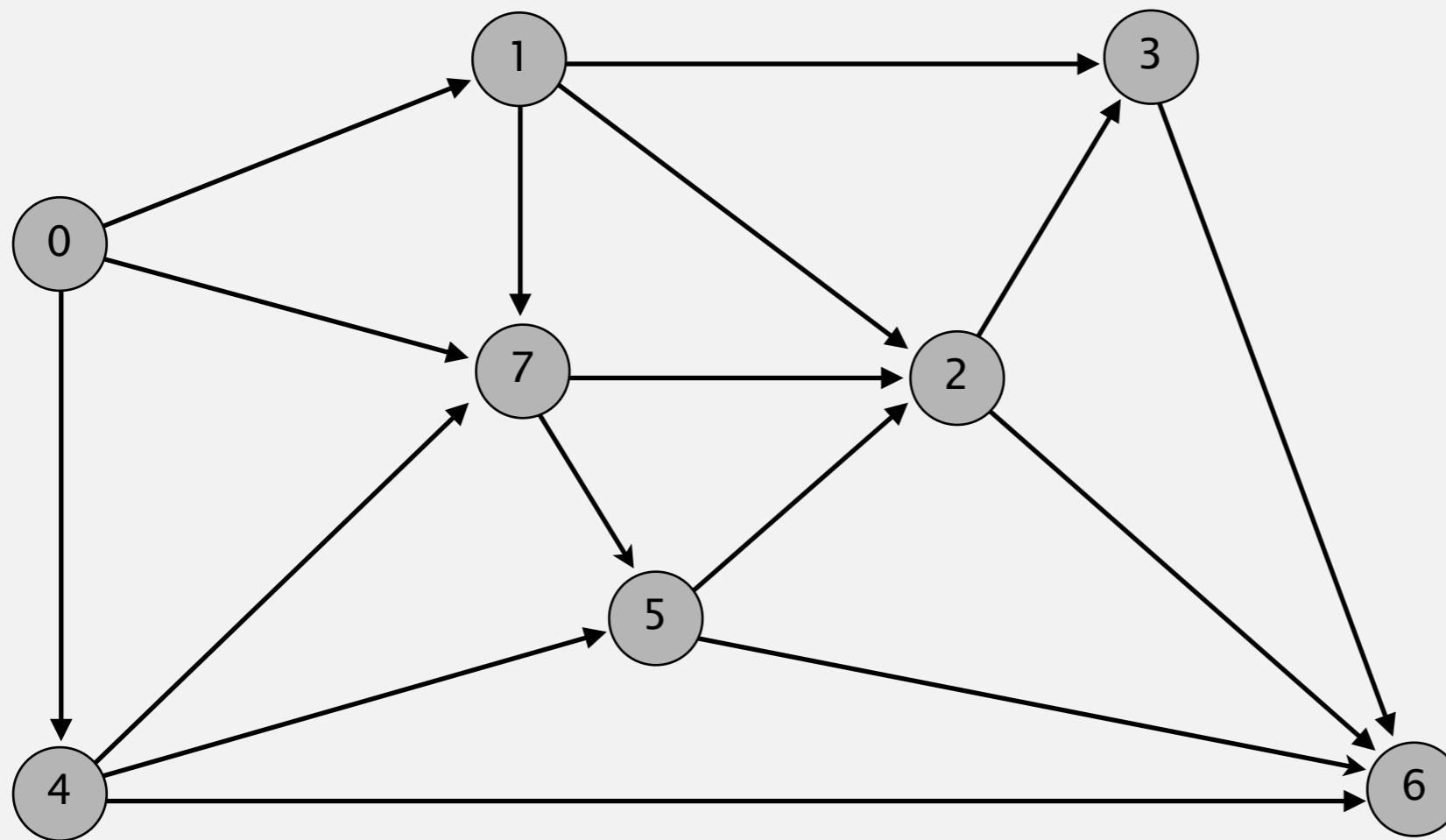


an edge-weighted DAG

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Topological sort algorithm

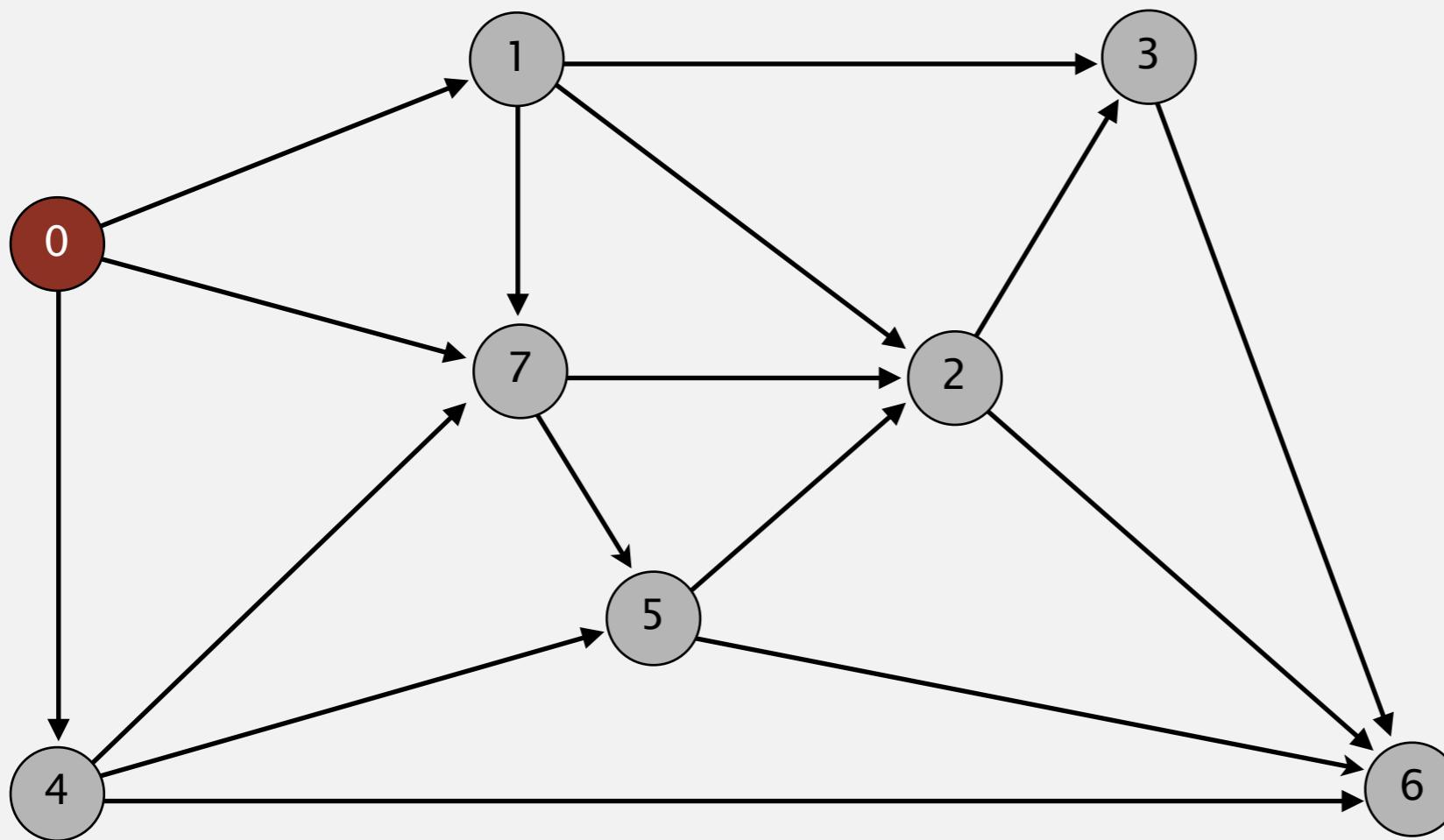
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



topological order: 0 1 4 7 5 2 3 6

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

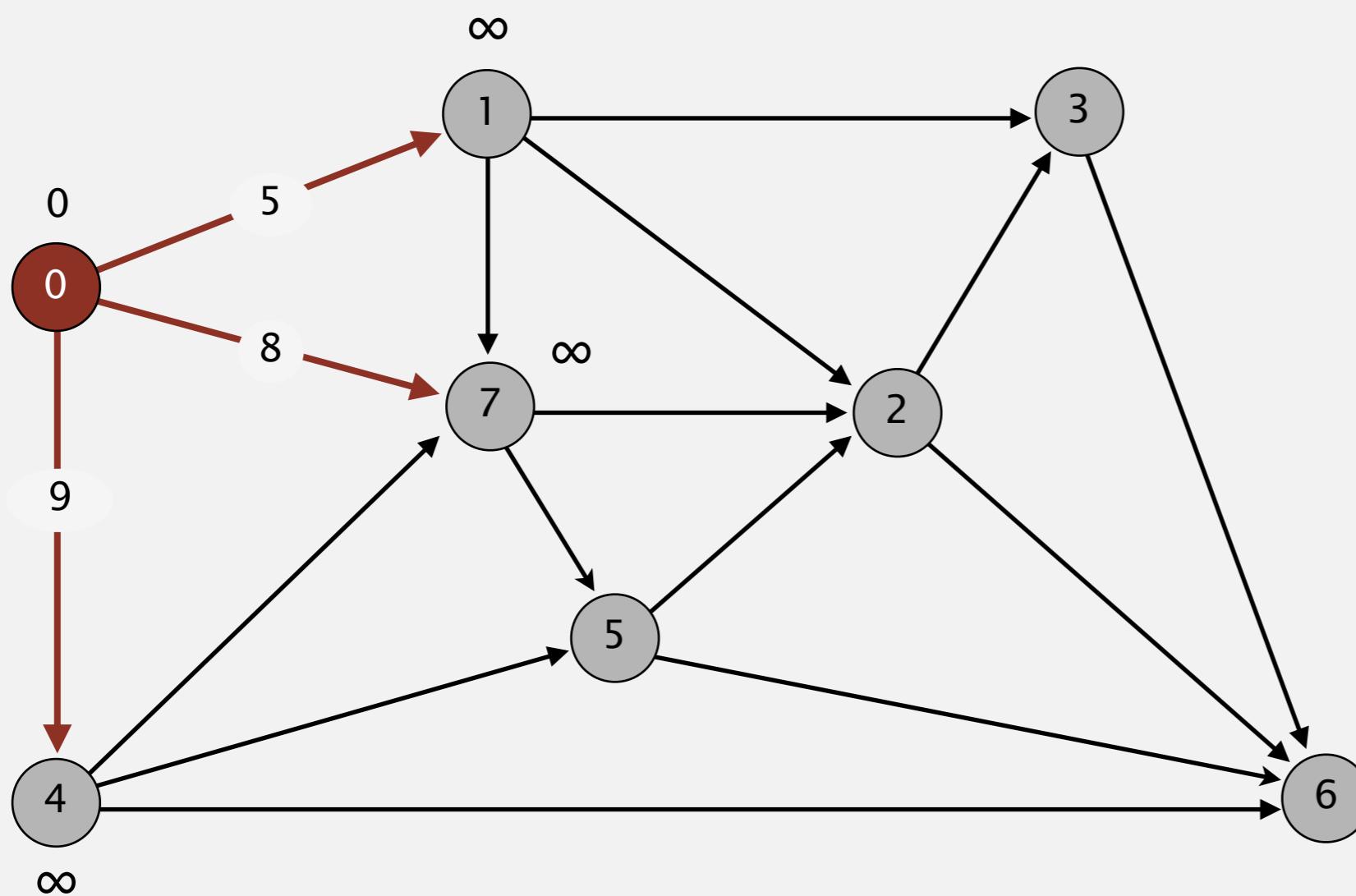


choose vertex 0

v	distTo []	edgeTo []
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

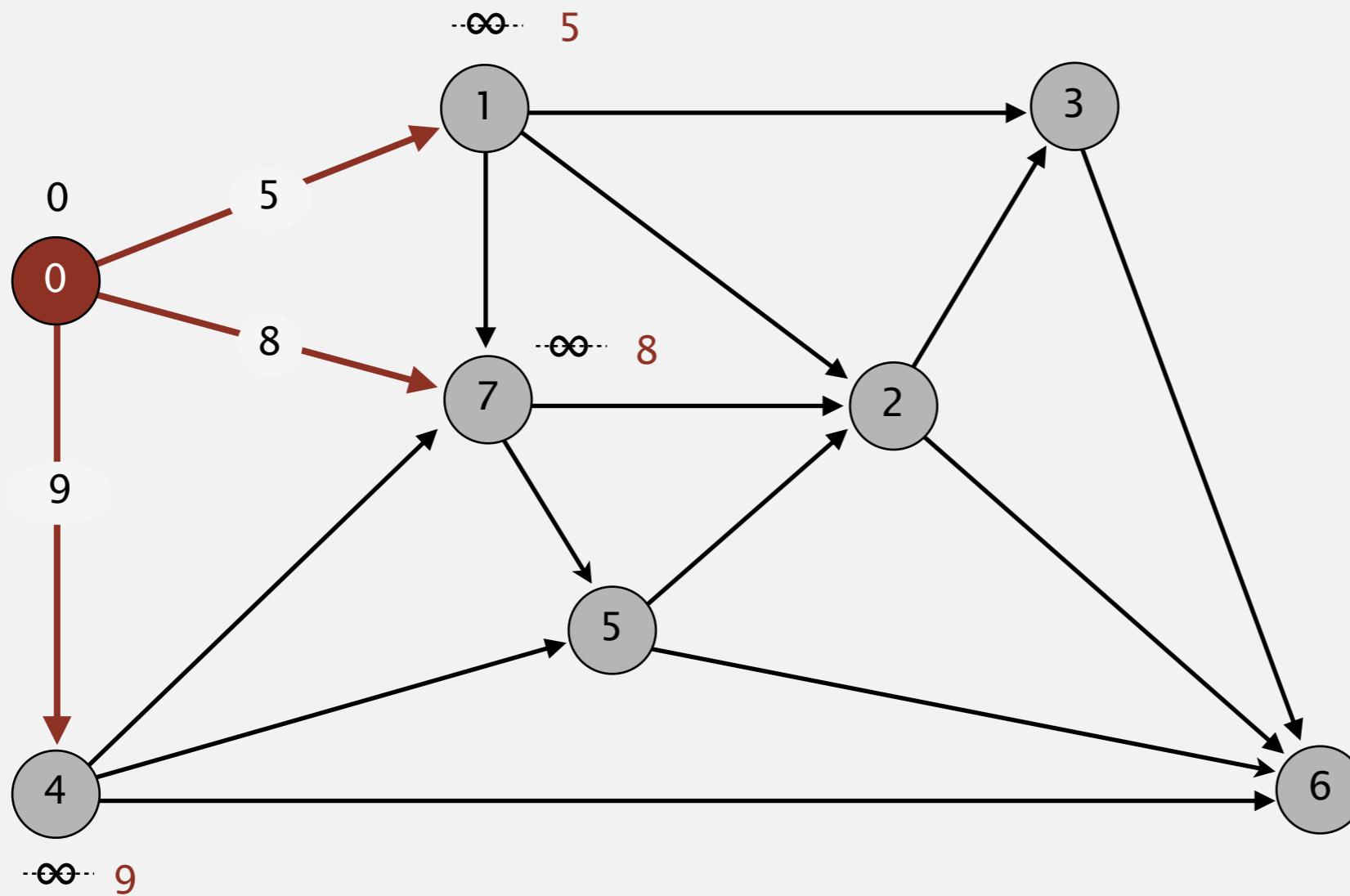


v	distTo []	edgeTo []
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

relax all edges incident from 0

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

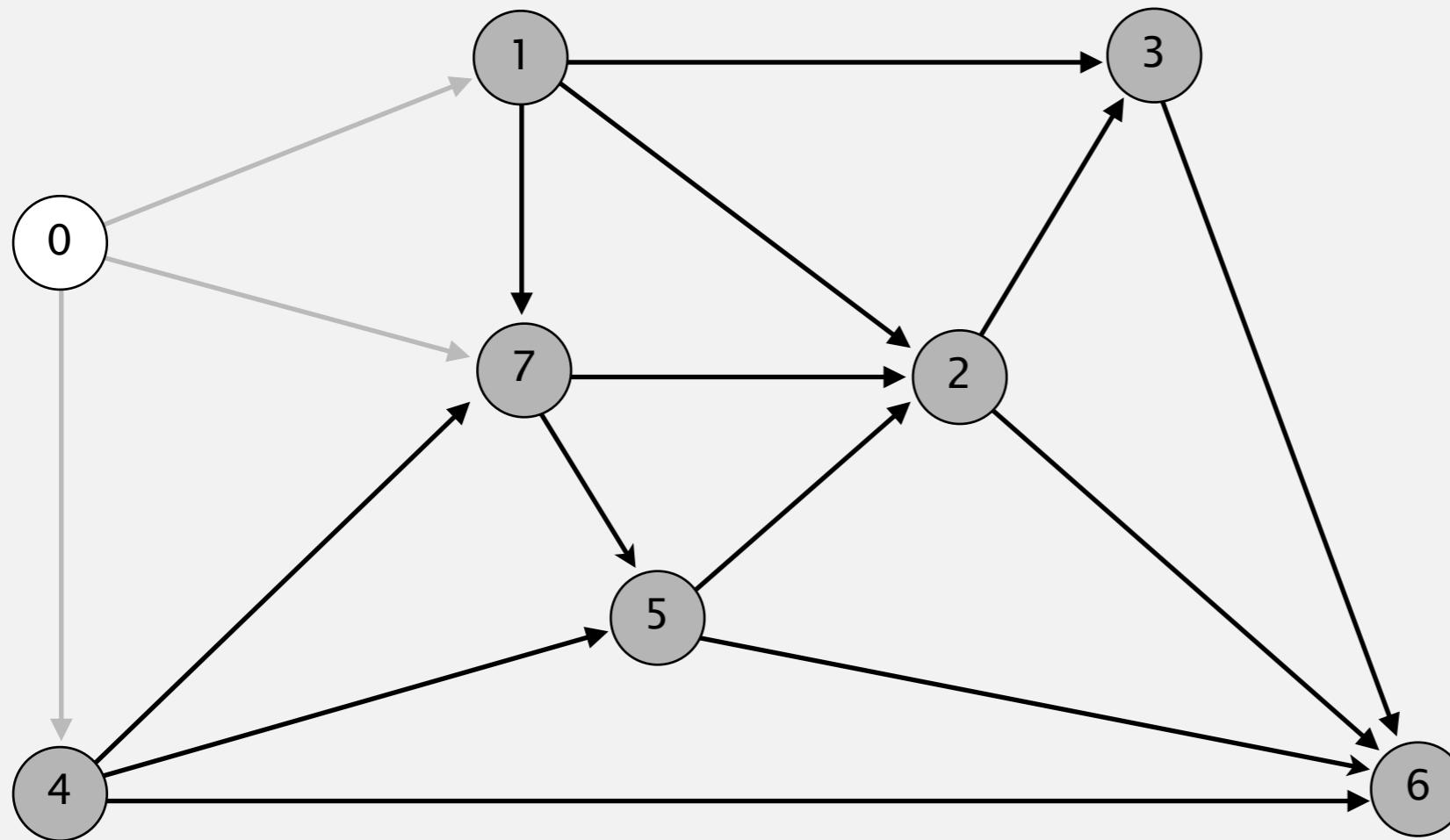


relax all edges incident from 0

v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

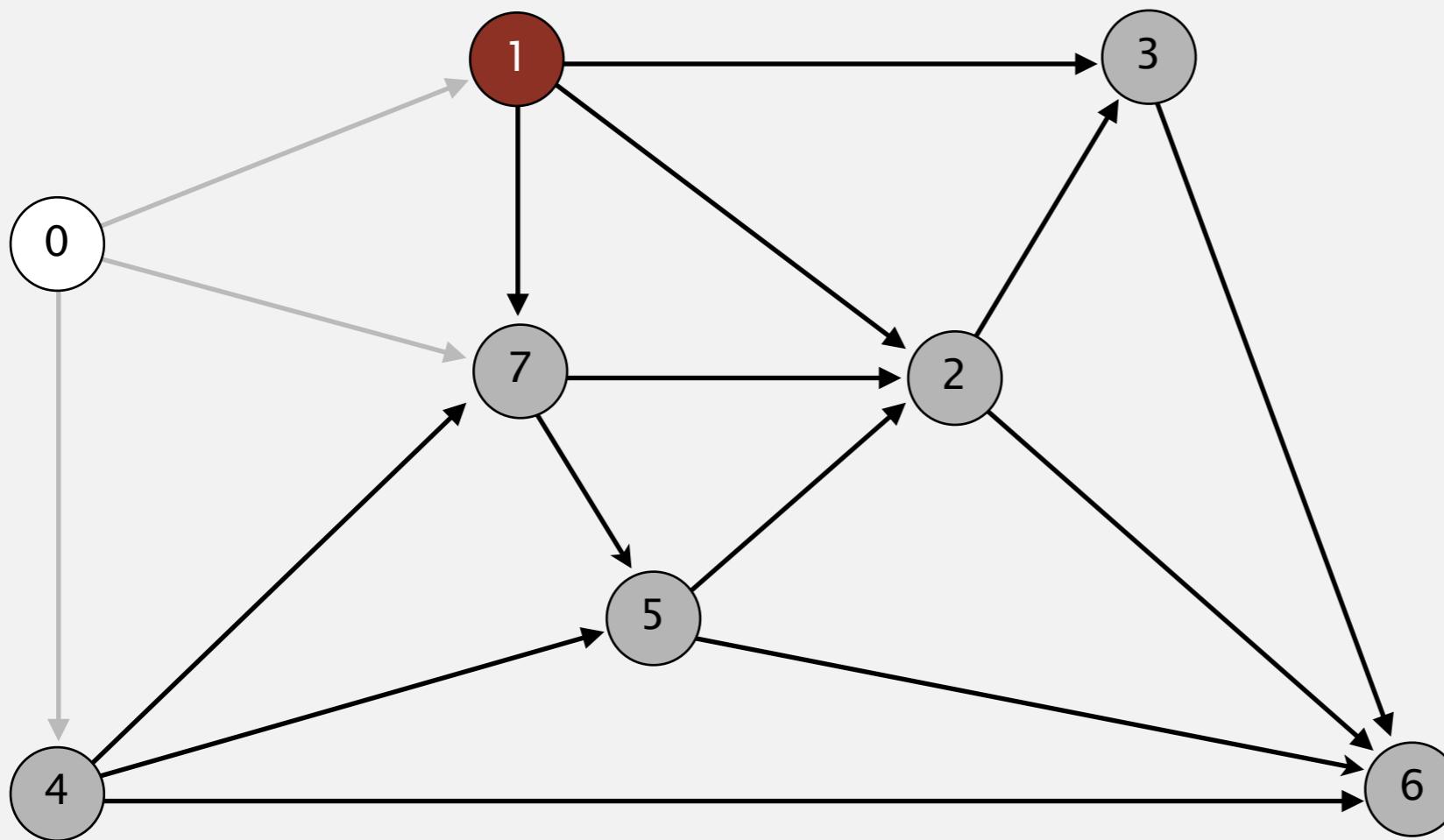


↓
0 1 4 7 5 2 3 6

v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

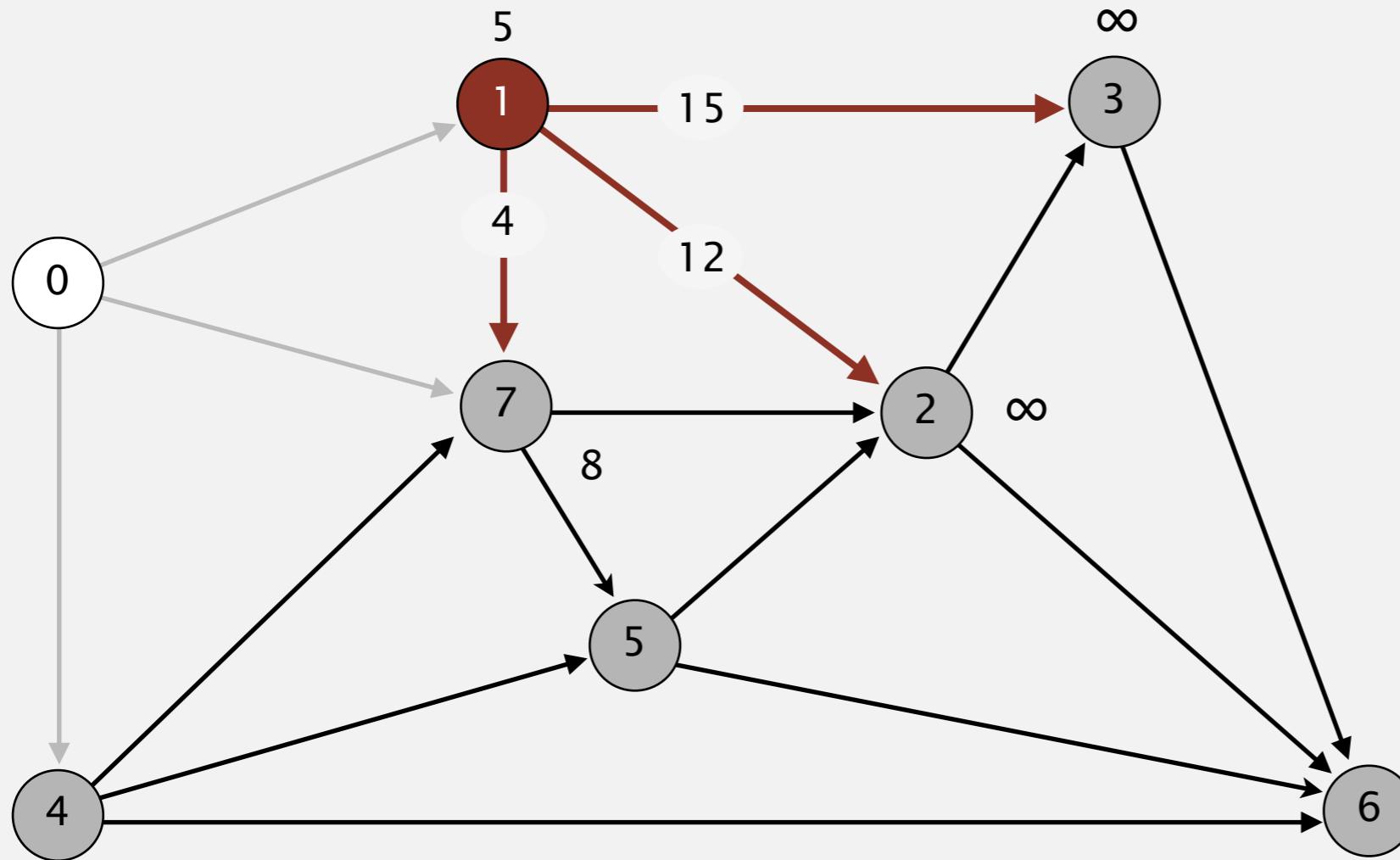


choose vertex 1

v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

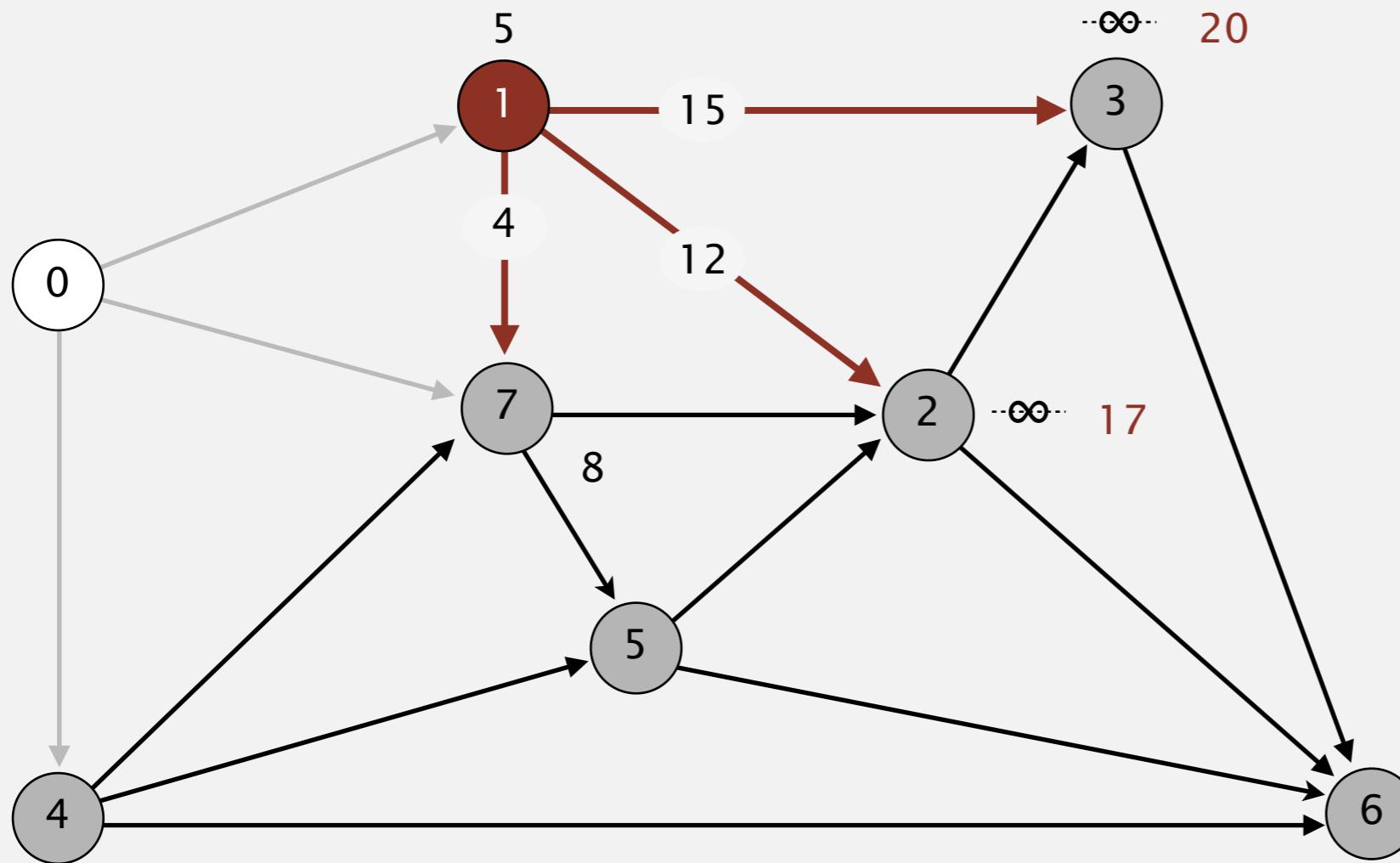


v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

relax all edges incident from 1

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

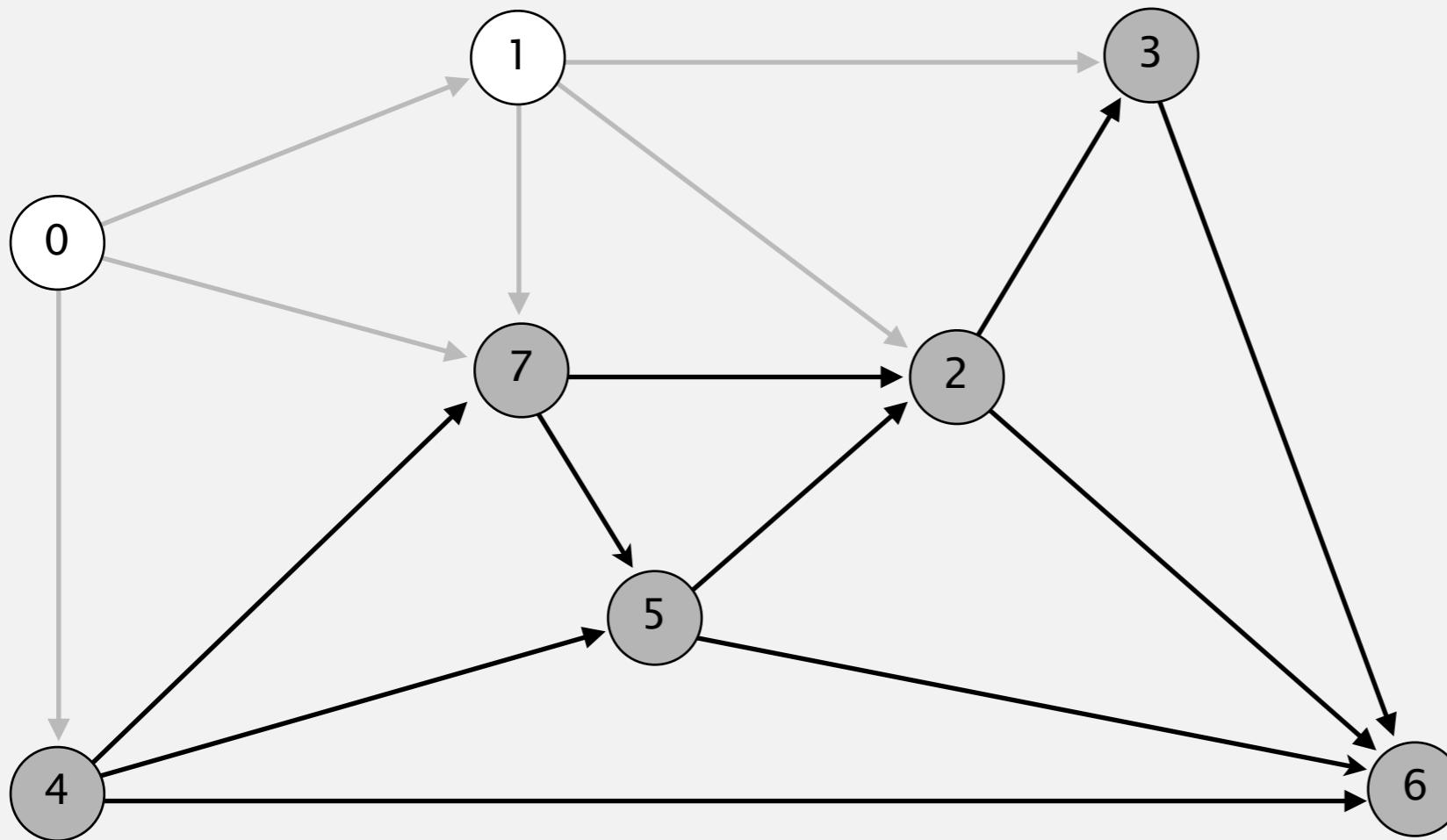


relax all edges incident from 1

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

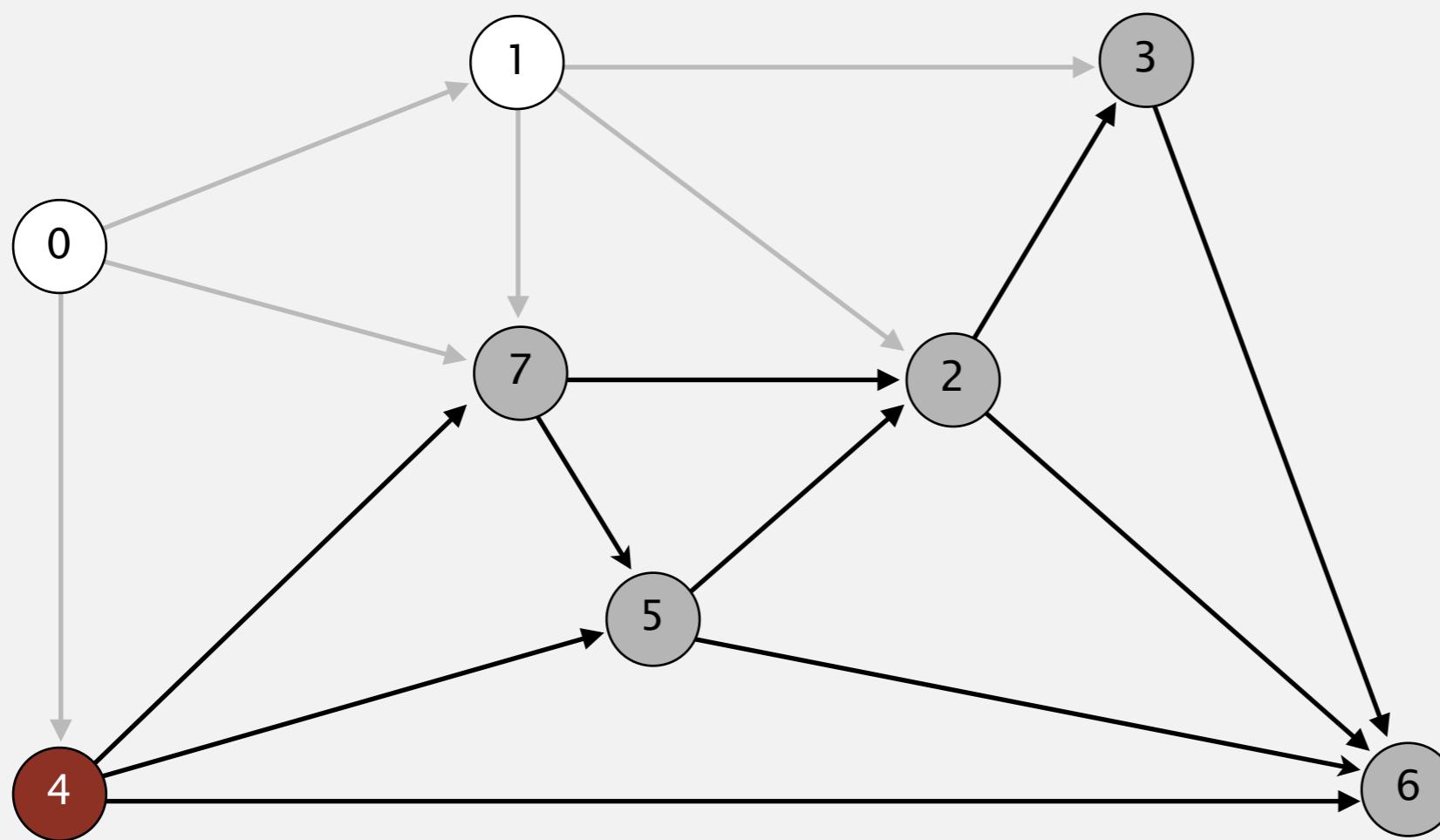


0 1 4 7 5 2 3 6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.



↓
0 1 4 7 5 2 3 6

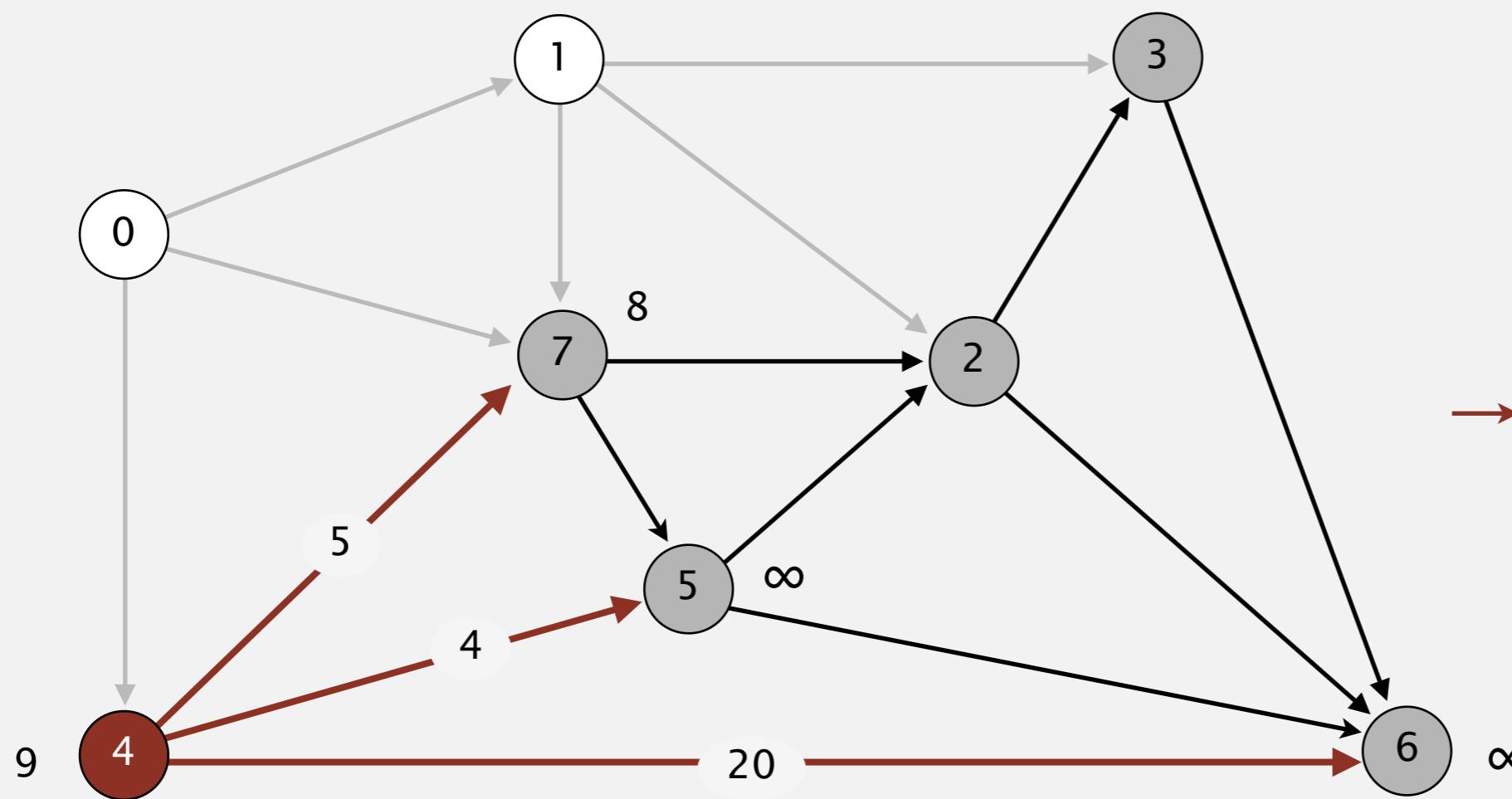
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

select vertex 4

(Dijkstra would have selected vertex 7)

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.



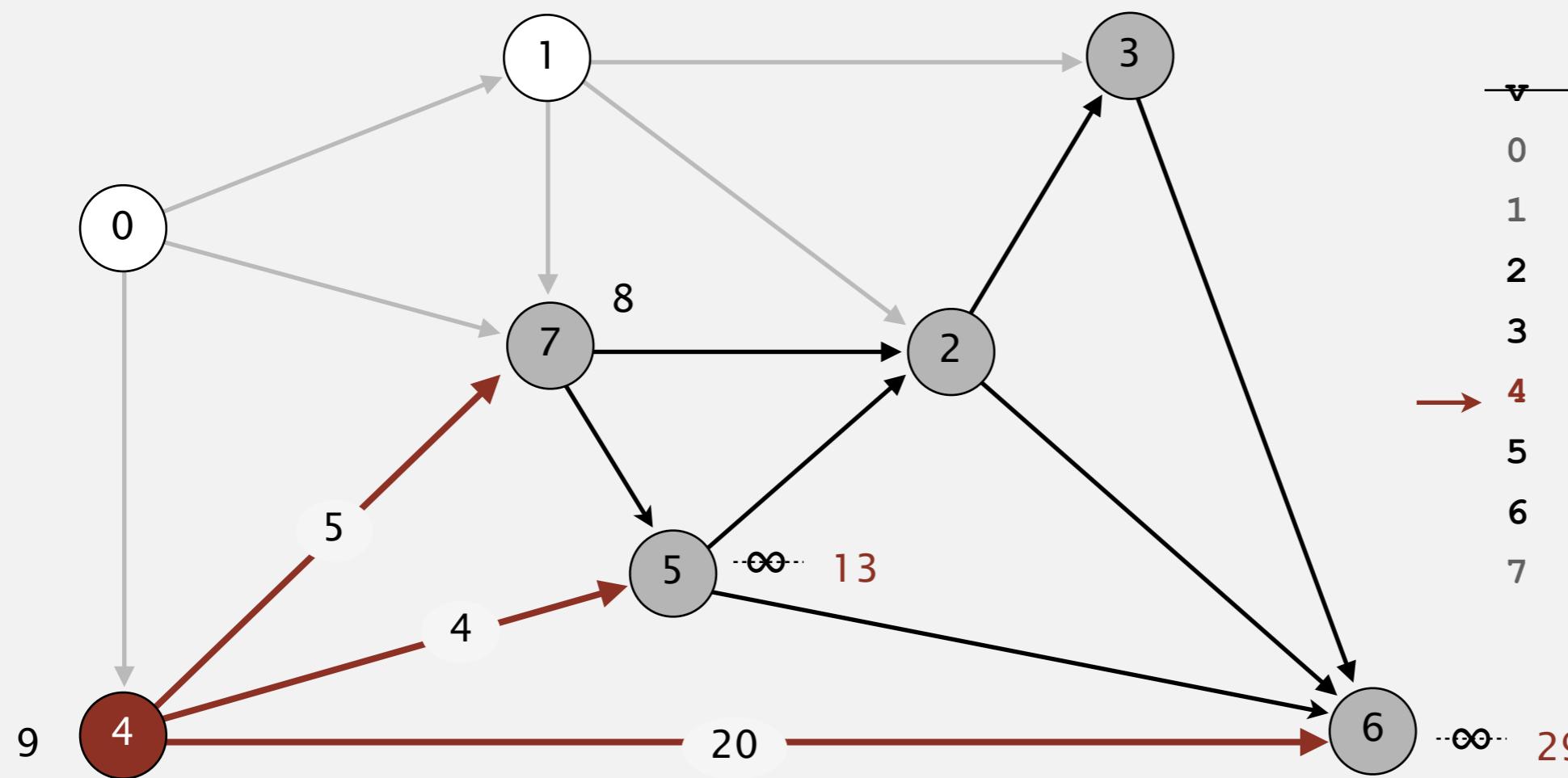
0 1 4 7 5 2 3 6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

relax all edges incident from 4

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

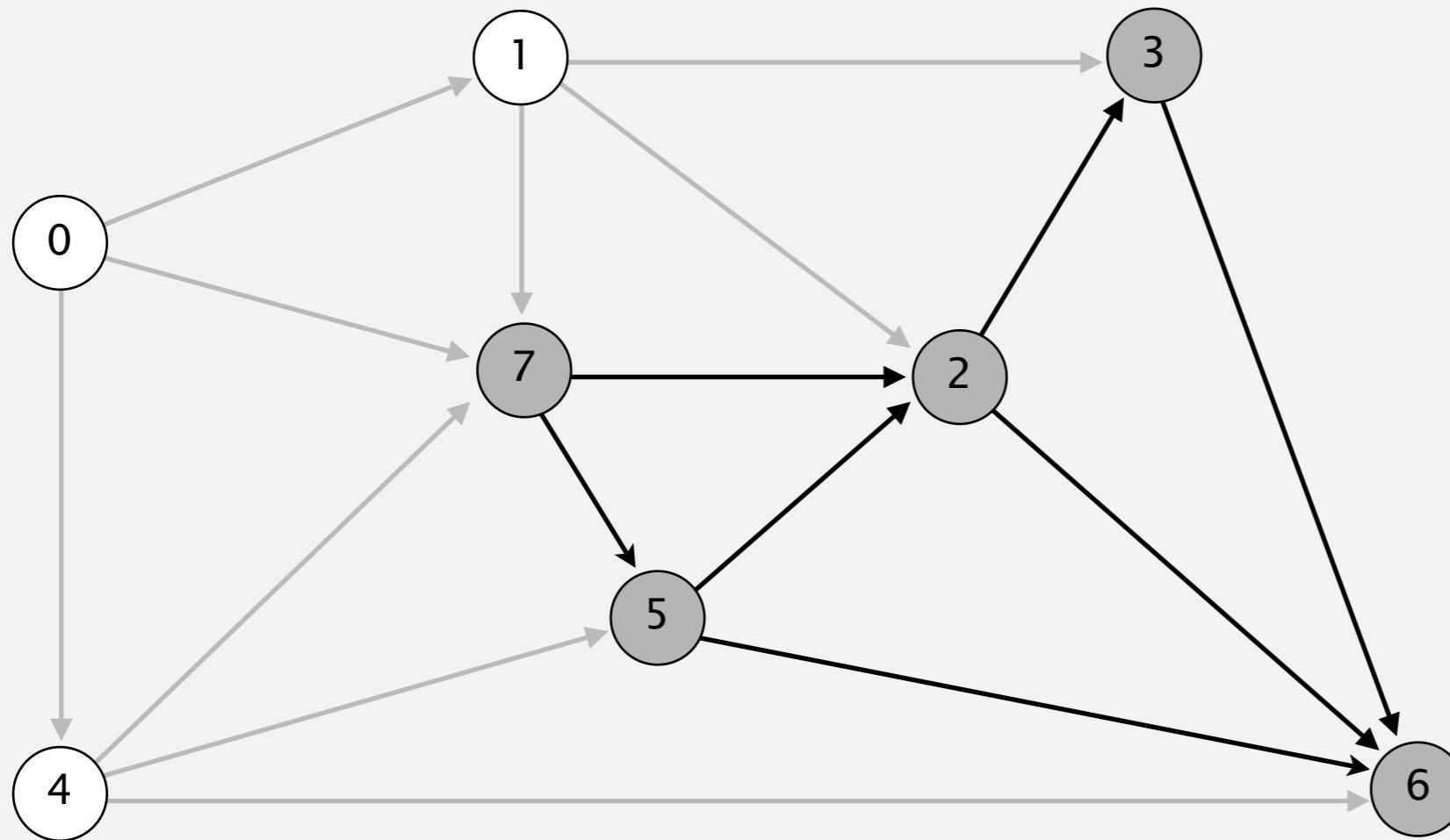


relax all edges incident from 4

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Topological sort algorithm

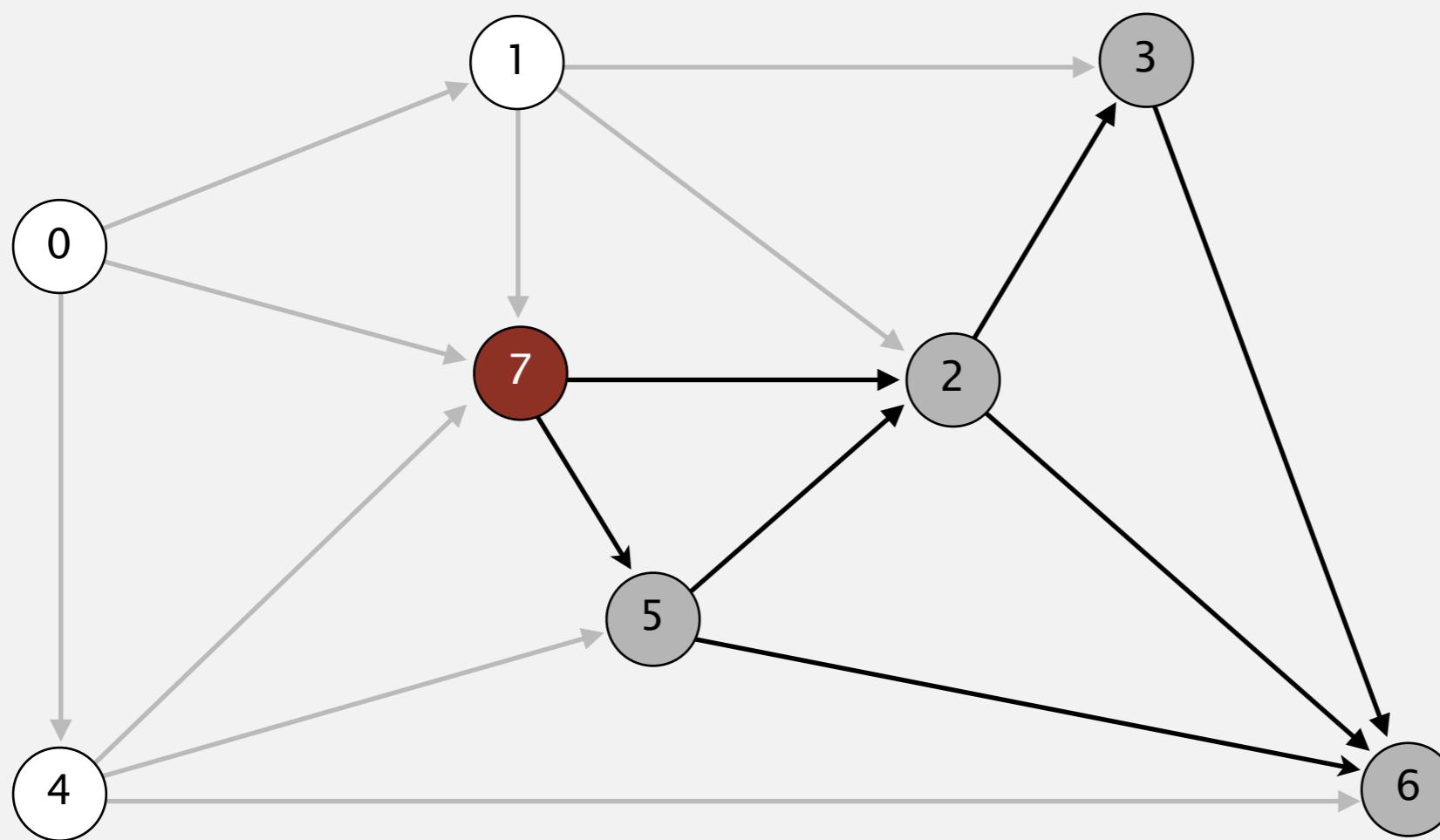
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

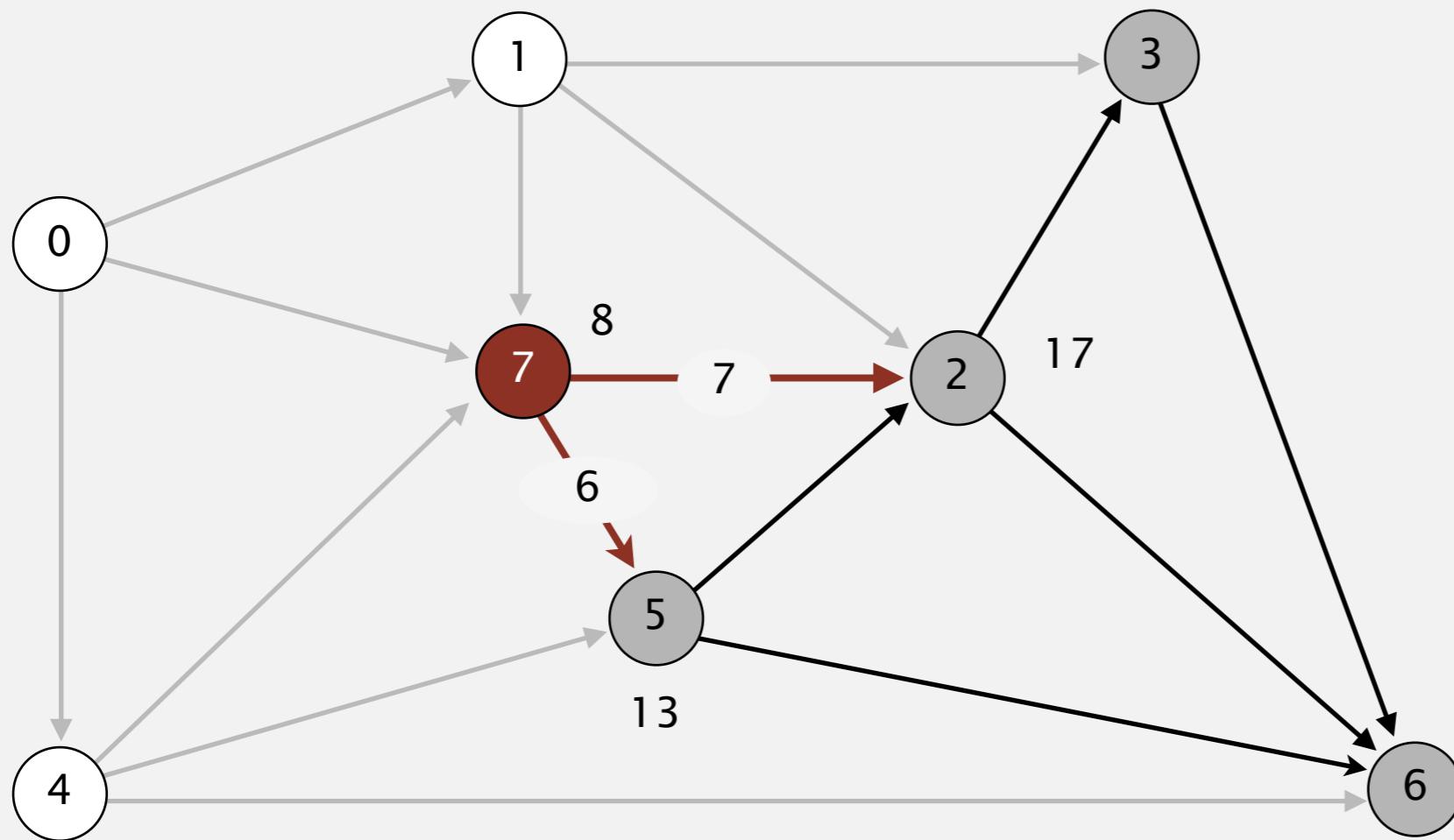


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

choose vertex 7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

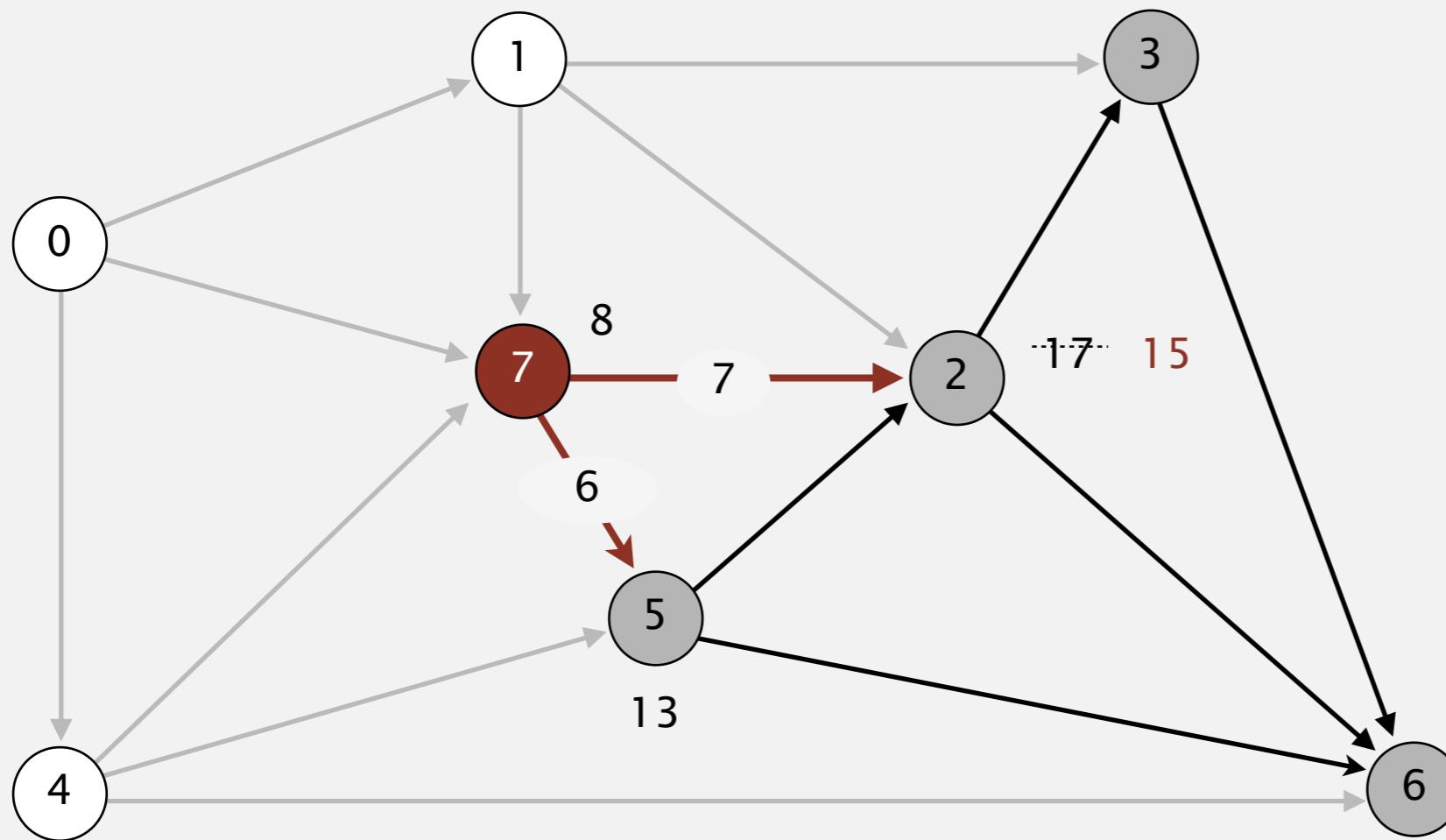


relax all edges incident from 7

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

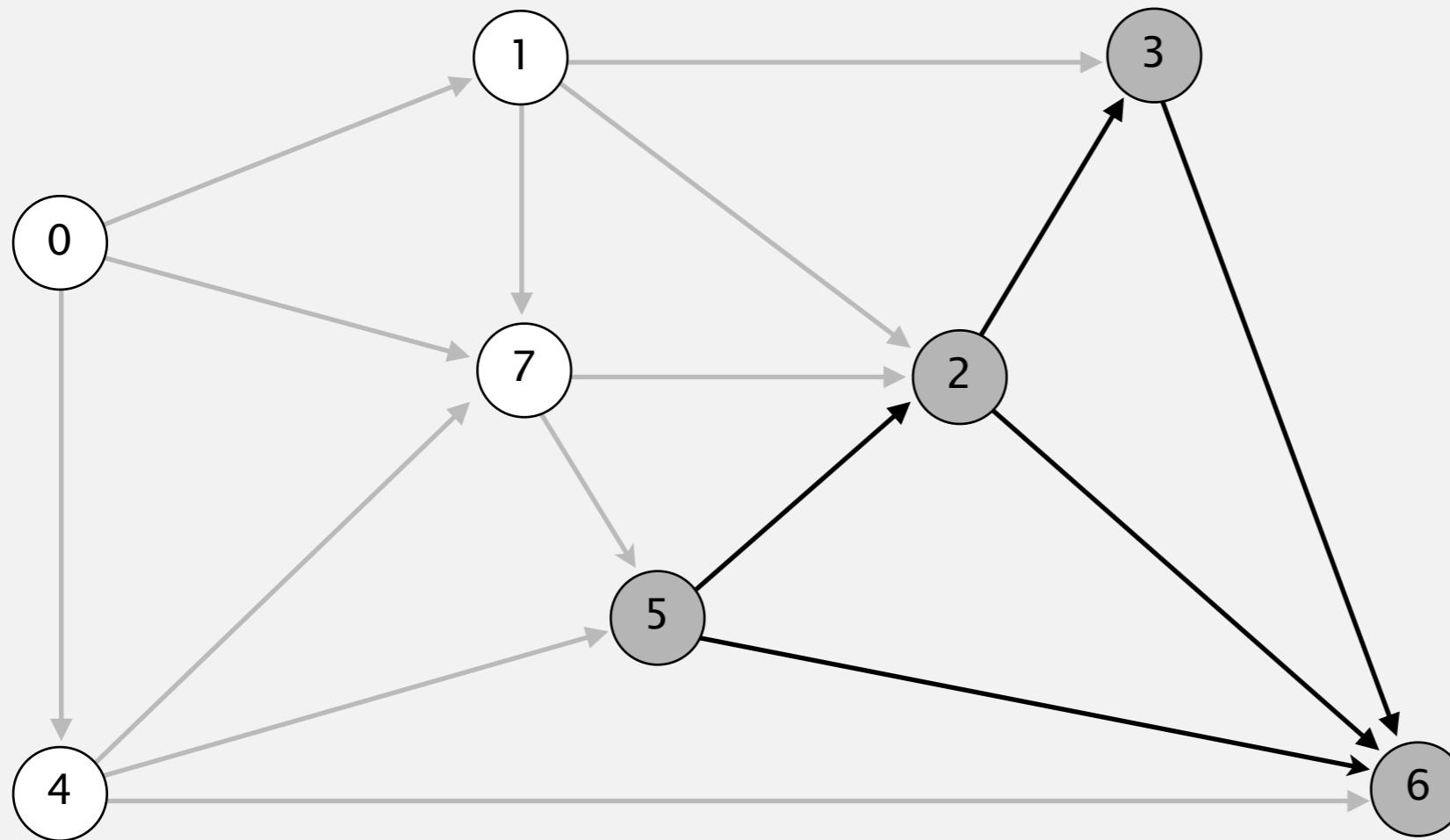


relax all edges incident from 7

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Topological sort algorithm

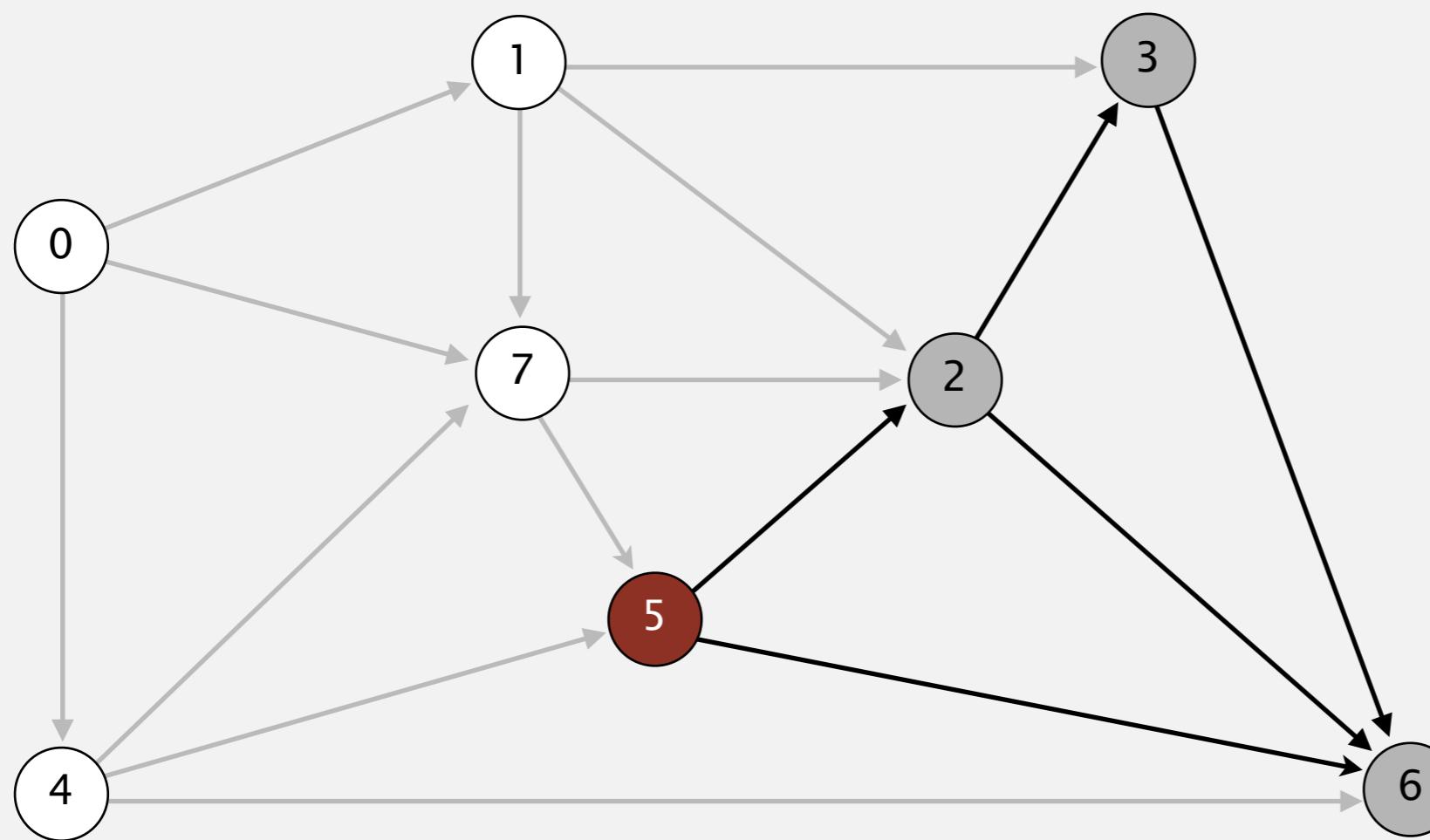
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

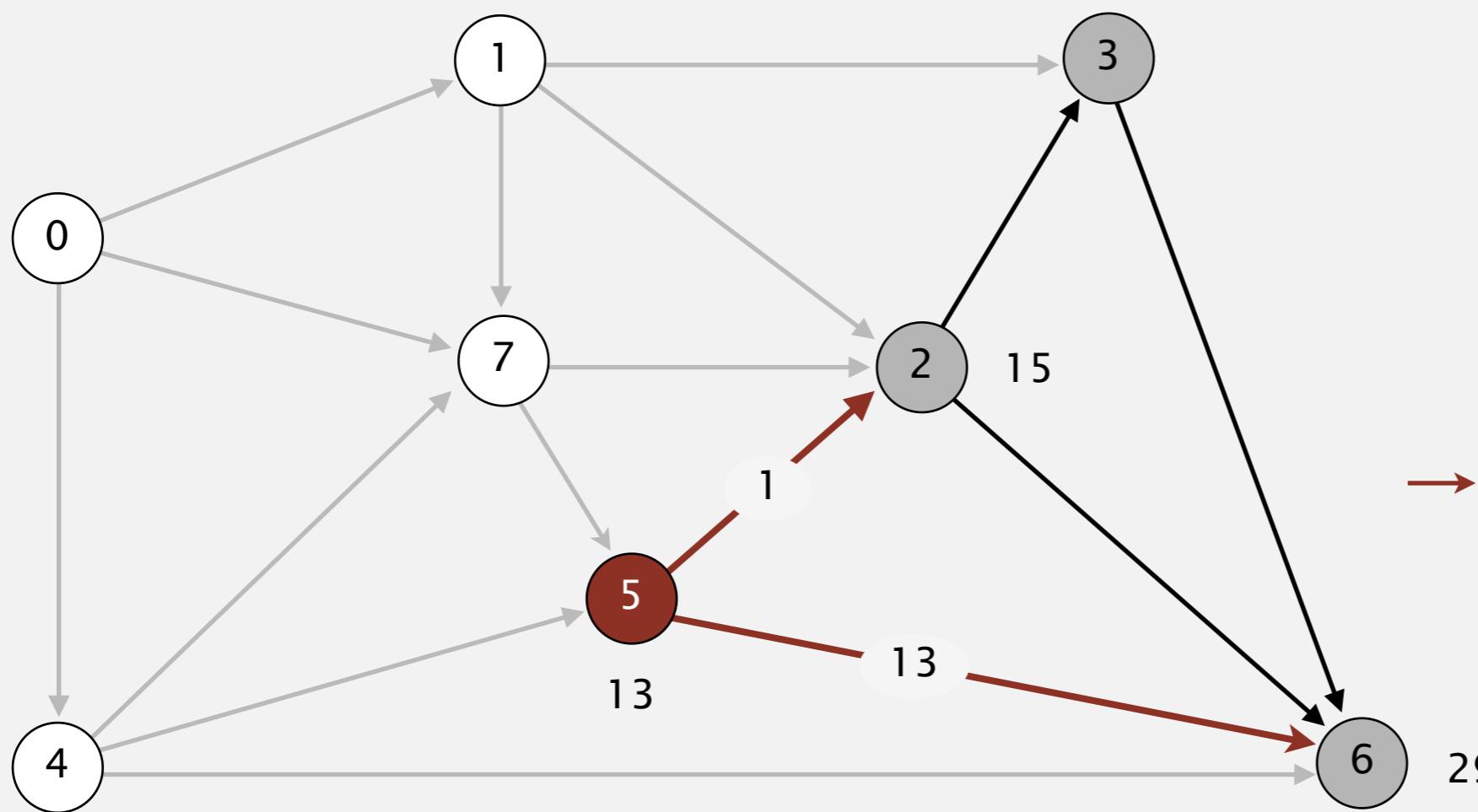


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

select vertex 5

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

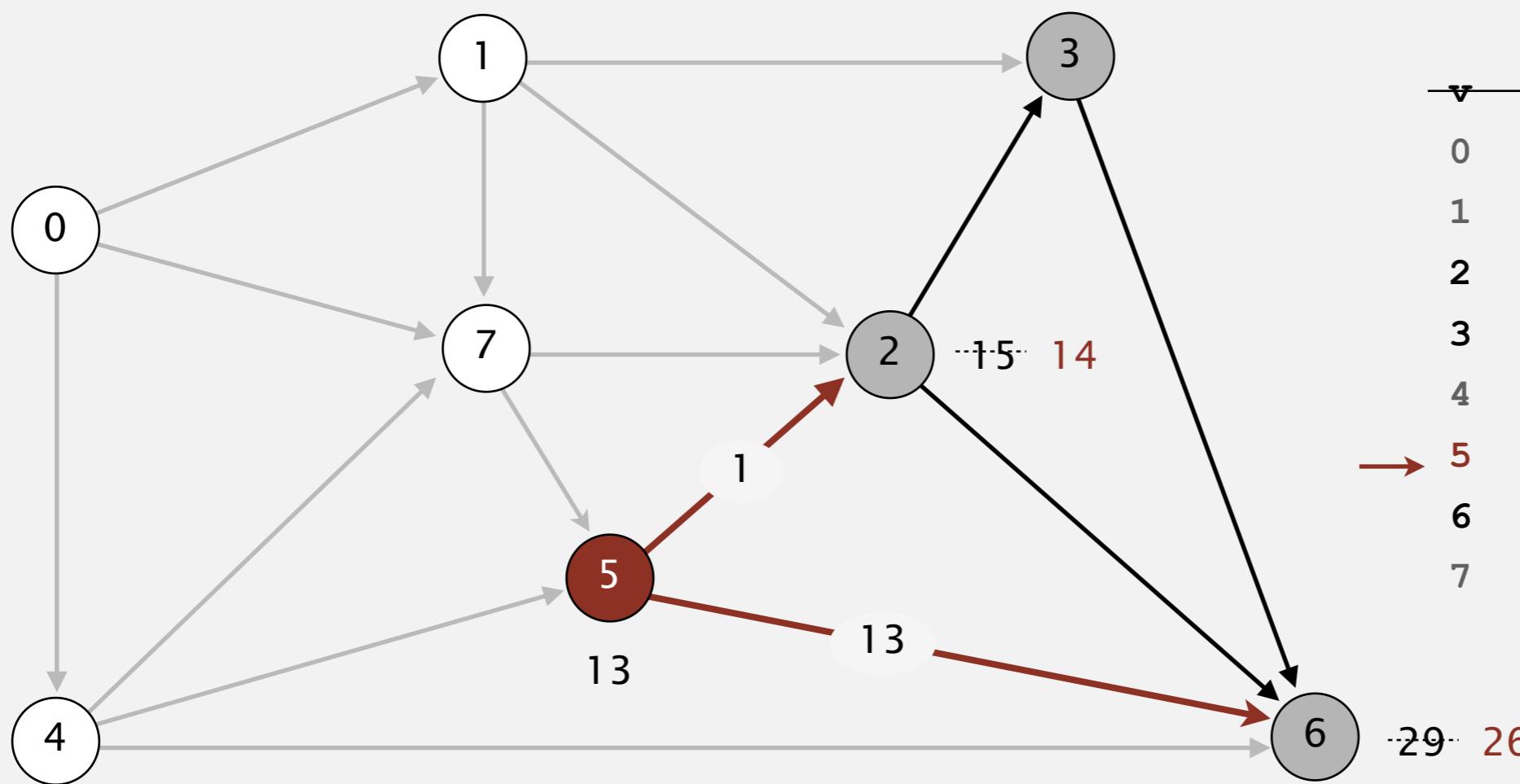


relax all edges incident from 5

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	15.0	7→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	29.0	4→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

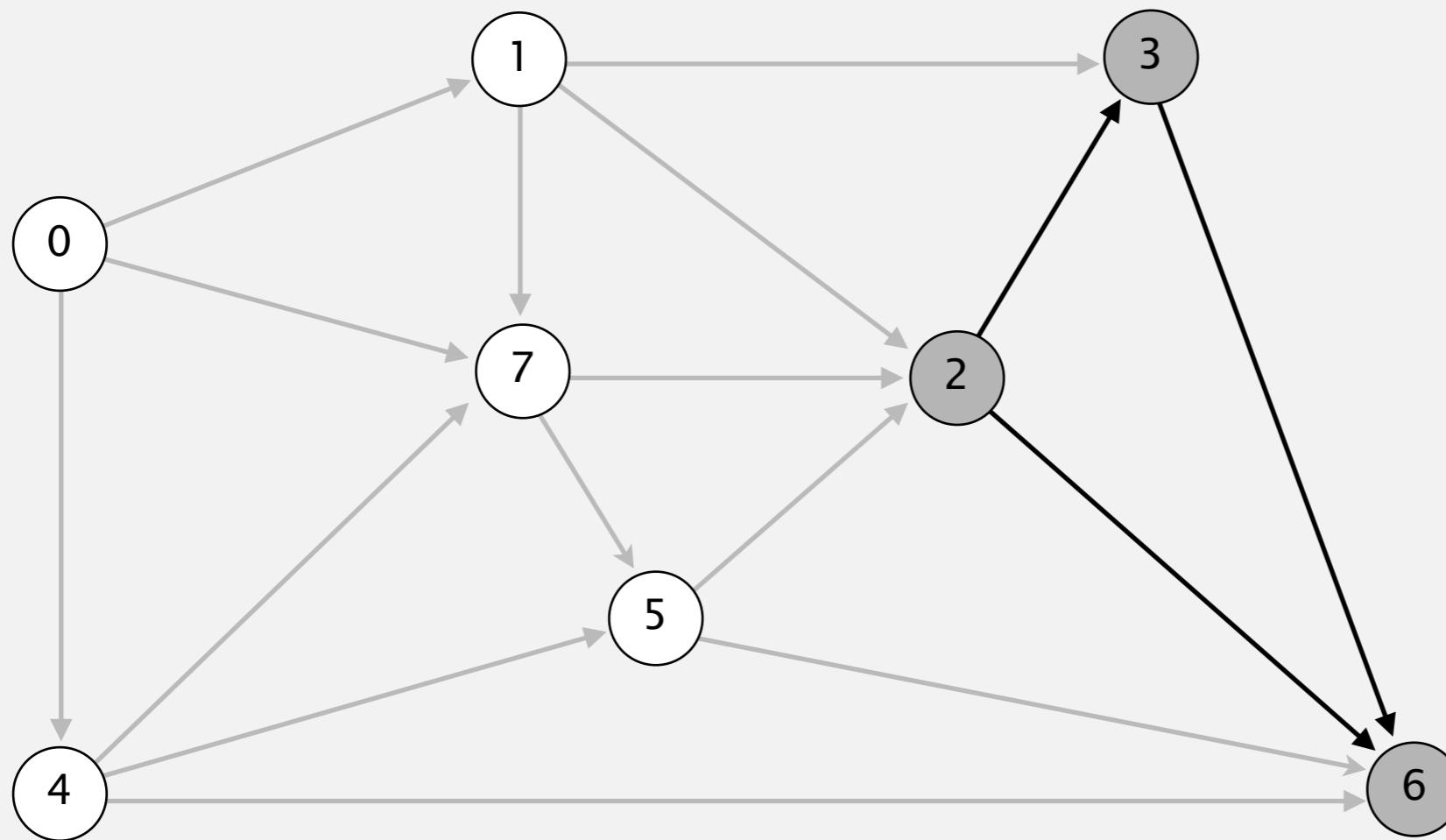


relax all edges incident from 5

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Topological sort algorithm

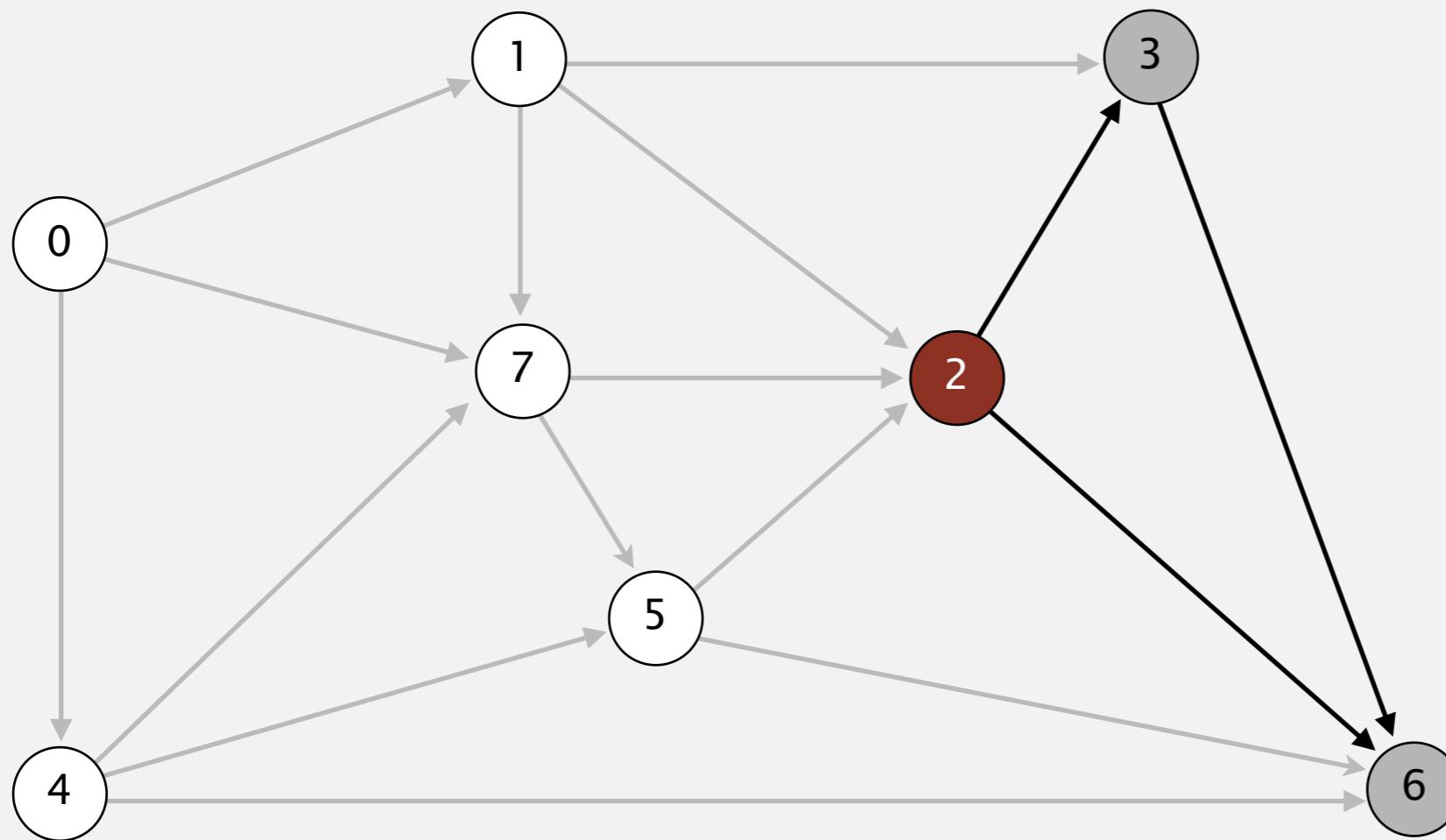
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

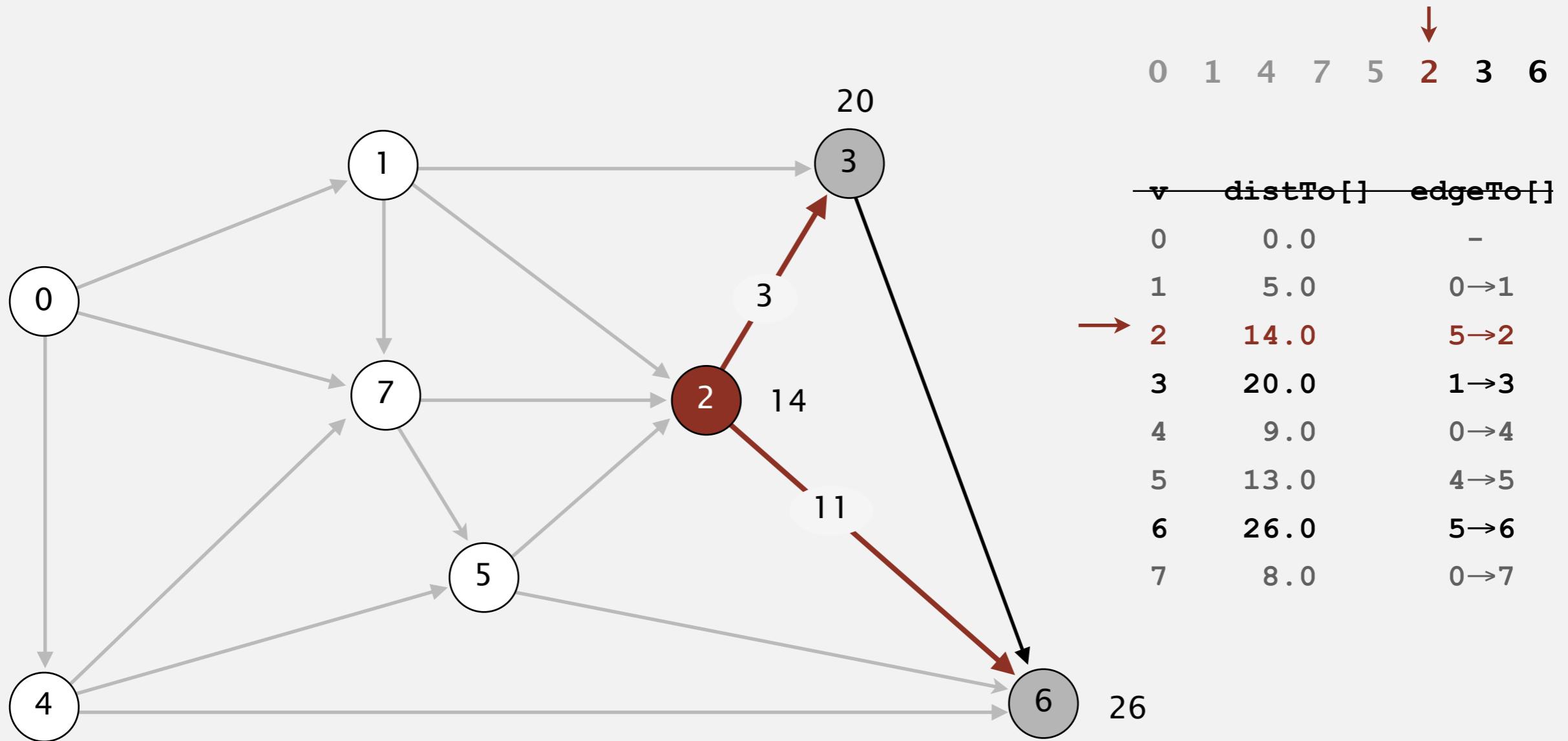


select vertex 2

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Topological sort algorithm

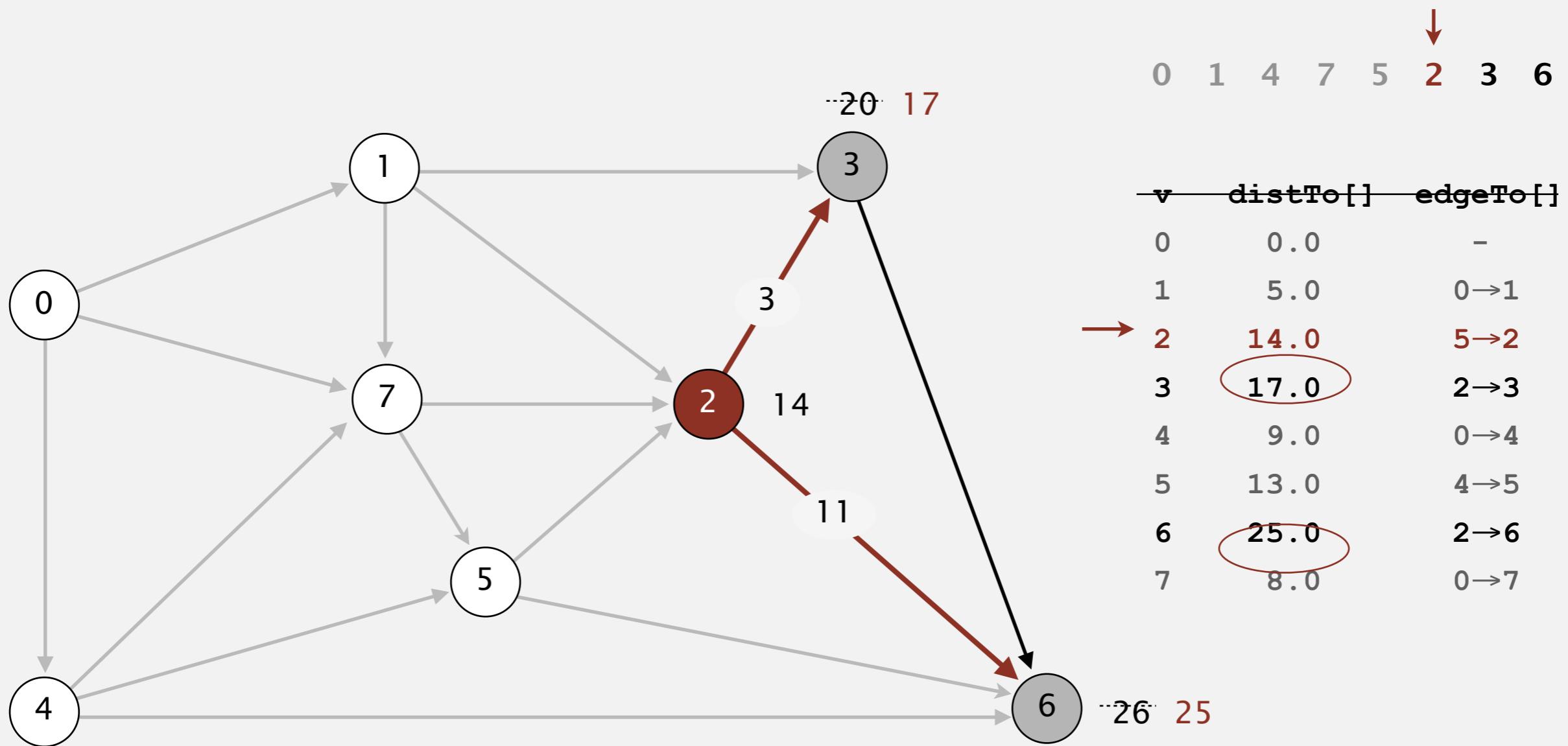
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



relax all edges incident from 2

Topological sort algorithm

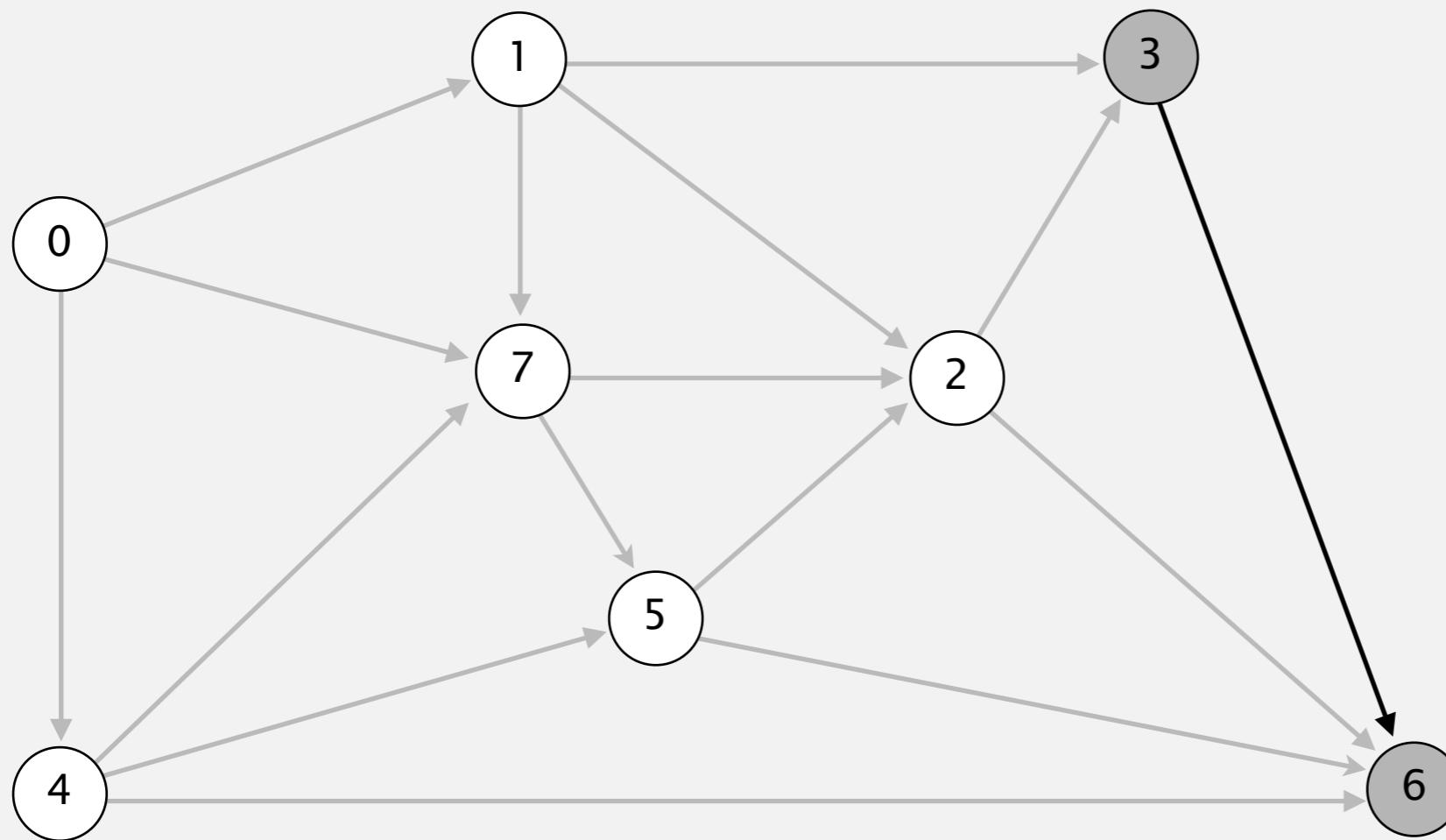
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



relax all edges incident from 2

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

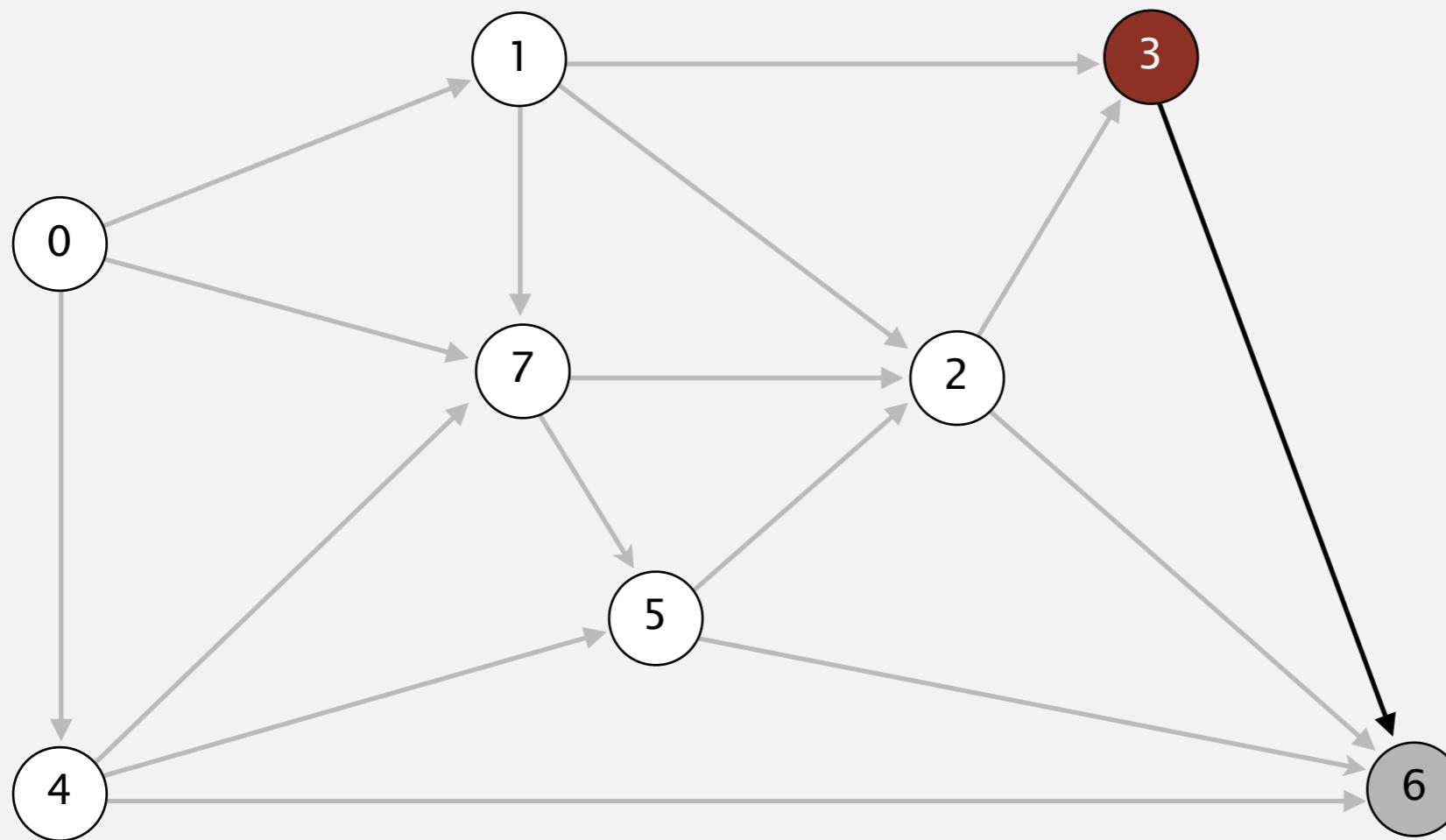


0 1 4 7 5 2 3 6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

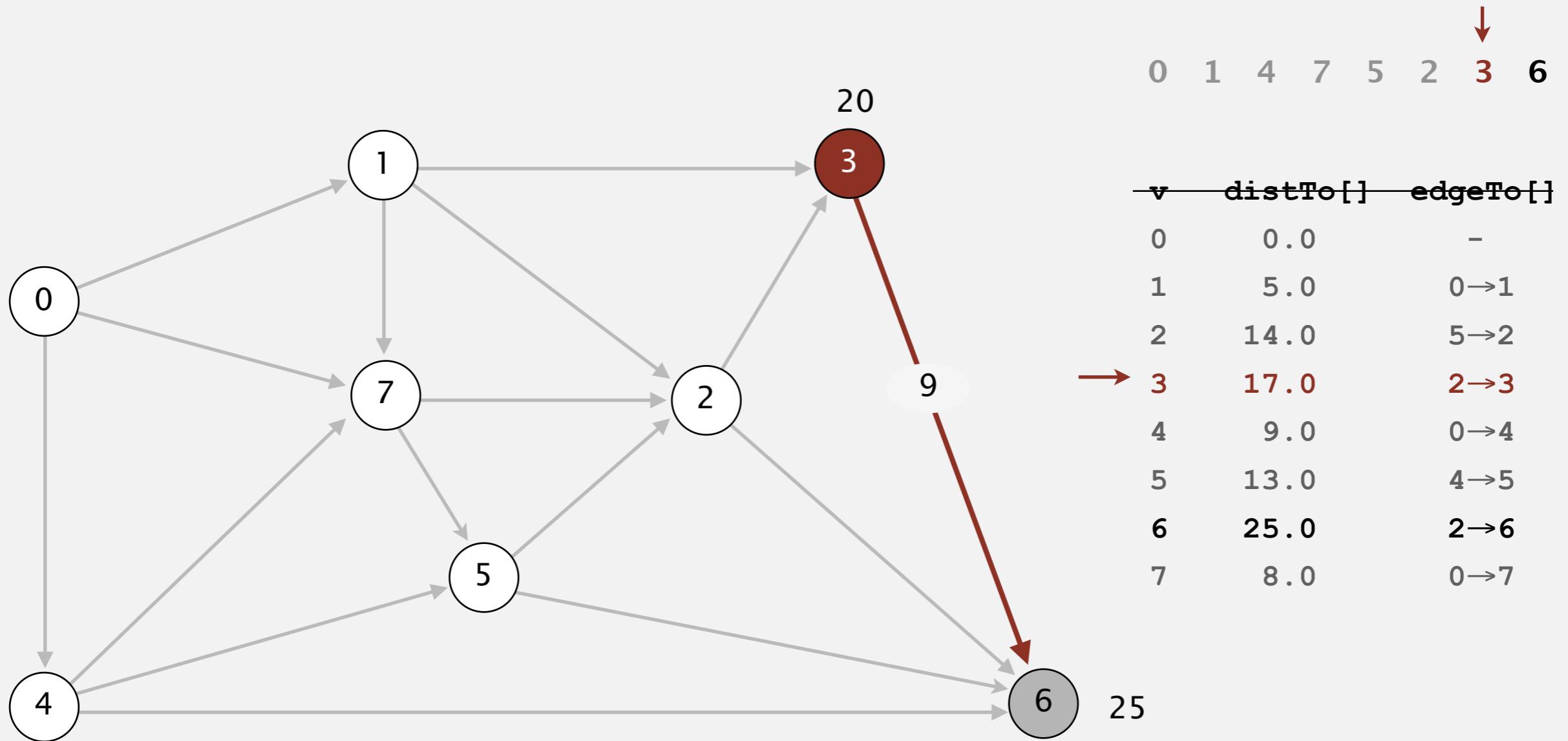


select vertex 3

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Topological sort algorithm

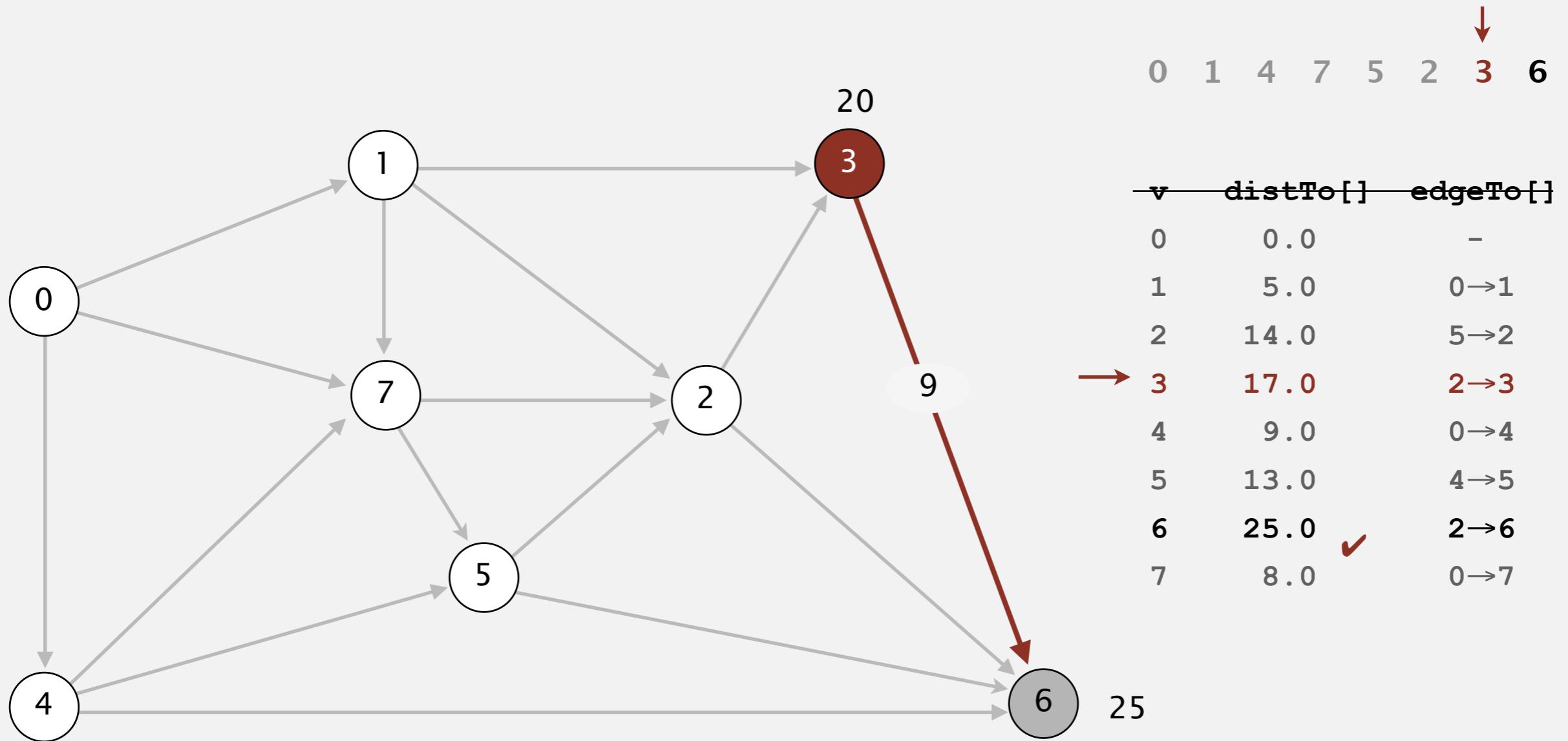
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



relax all edges incident from 3

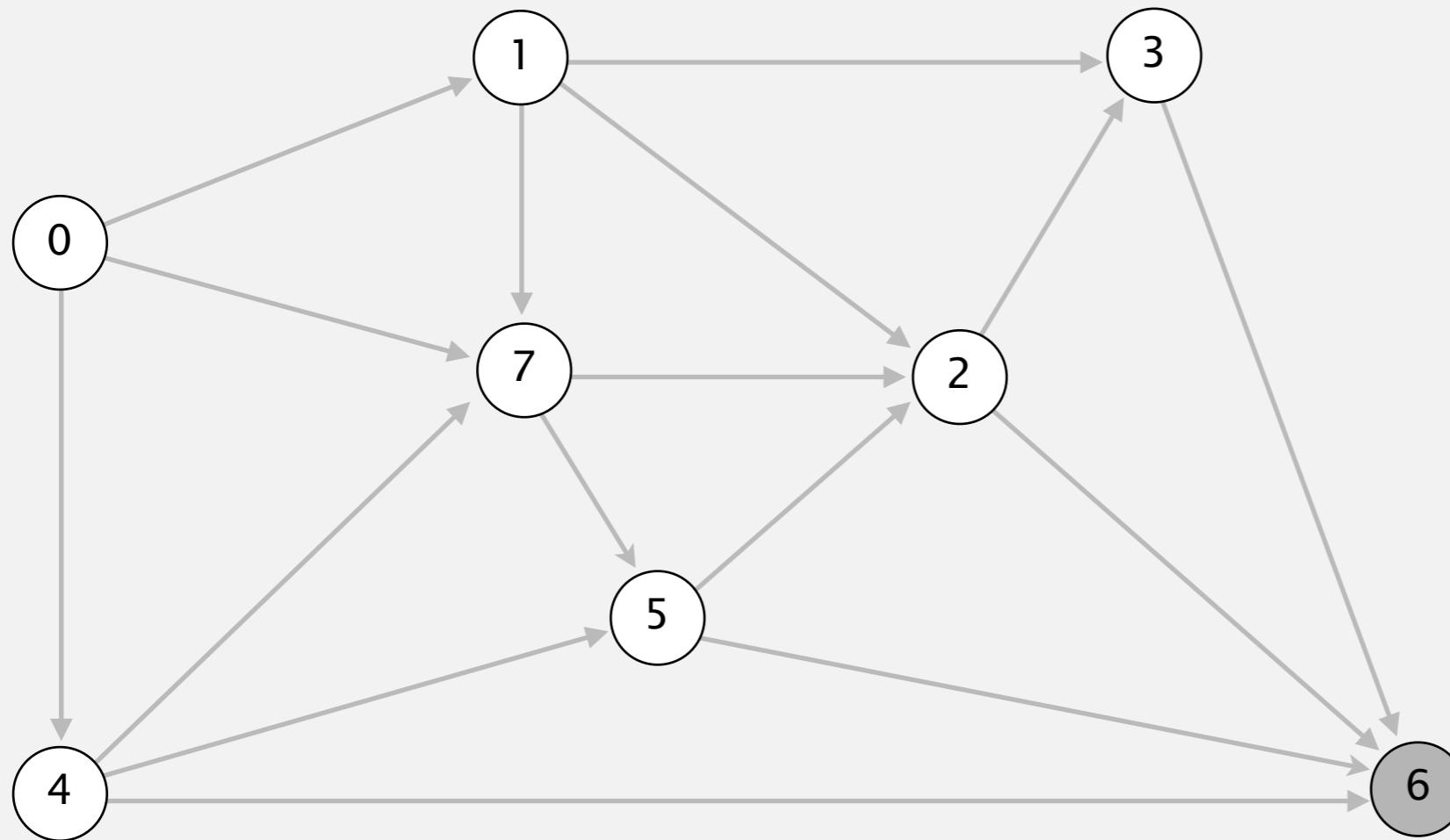
Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.



Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.

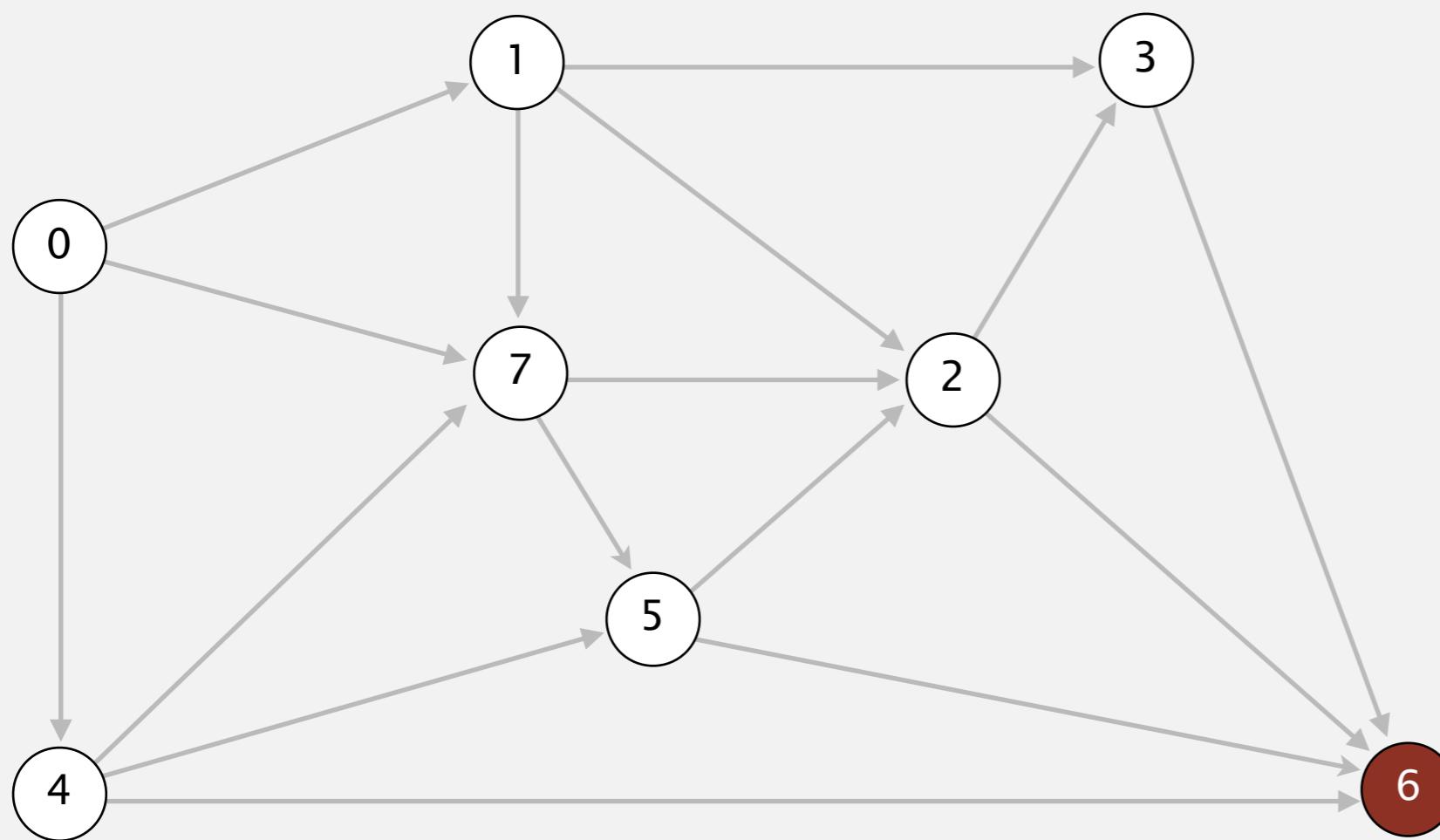


0 1 4 7 5 2 3 6 ↓

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.



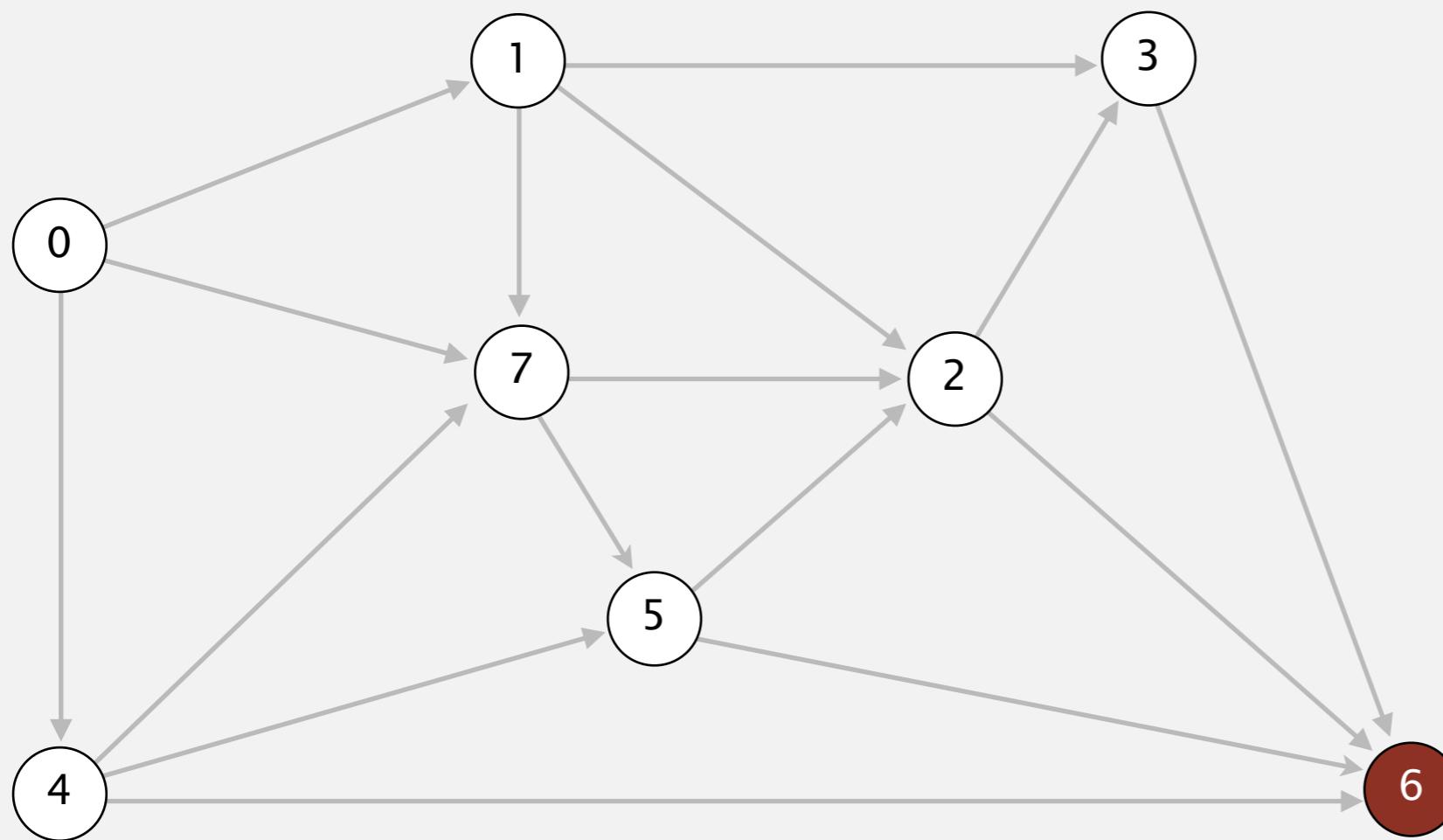
0 1 4 7 5 2 3 ↓
6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

select vertex 6

Topological sort algorithm

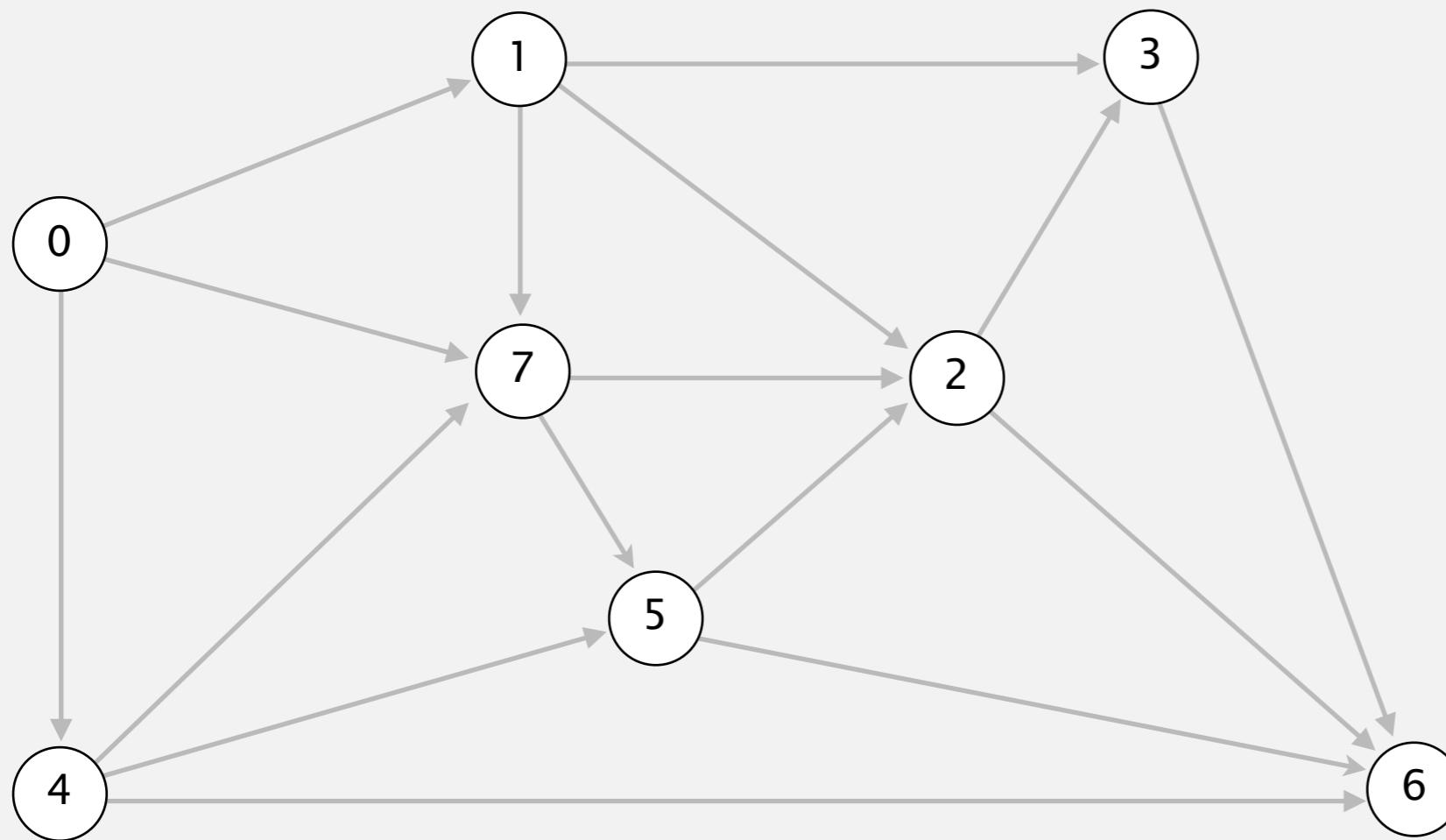
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Topological sort algorithm

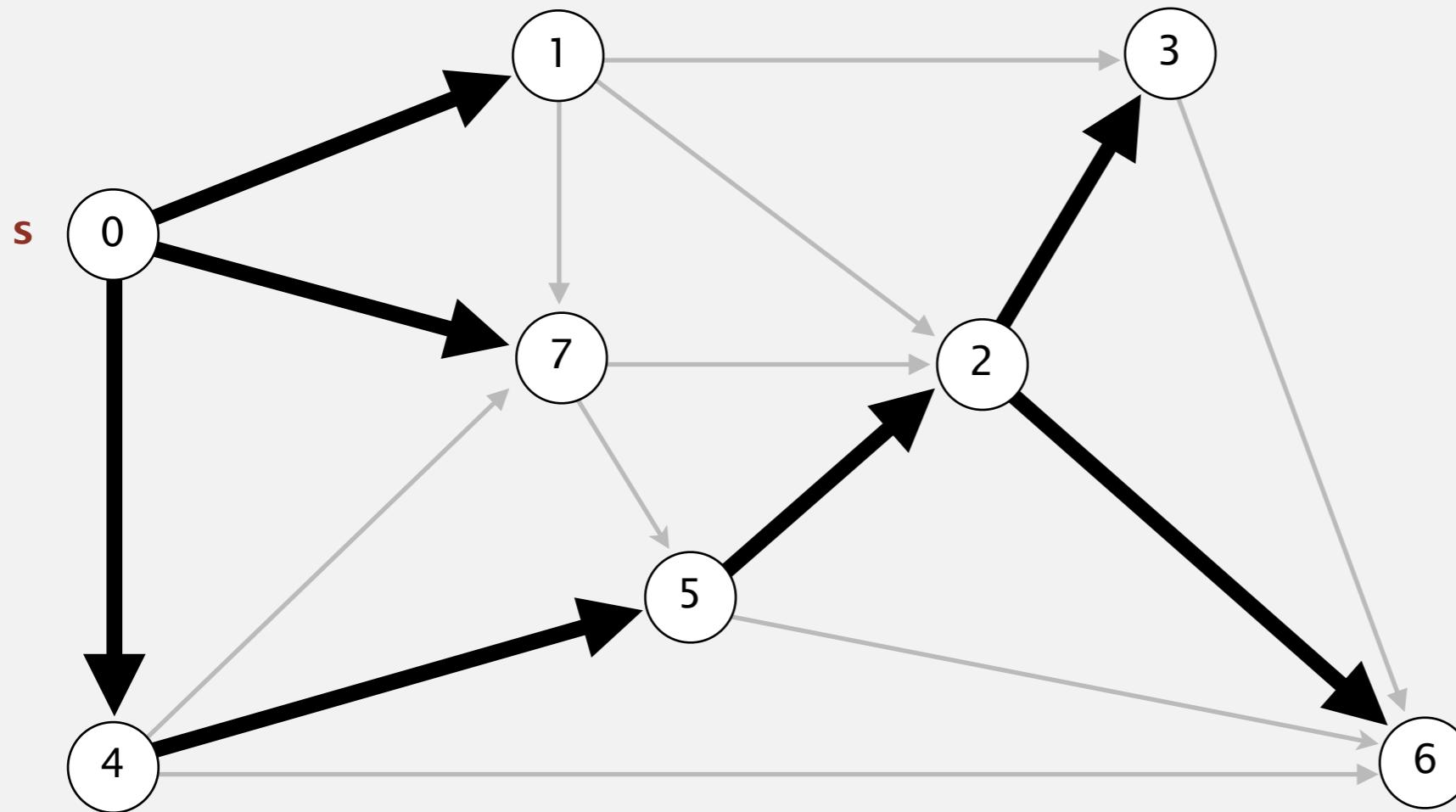
- Consider vertices in topological order.
- Relax all edges incident from that vertex.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Topological sort algorithm

- Consider vertices in topological order.
- Relax all edges incident from that vertex.



shortest-paths tree from vertex s

0	1	4	7	5	2	3	6
v	distTo[]	edgeTo[]					
0	0.0	-					
1	5.0	0→1					
2	14.0	5→2					
3	17.0	2→3					
4	9.0	0→4					
5	13.0	4→5					
6	25.0	2→6					
7	8.0	0→7					

Shortest paths in edge-weighted DAGs

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

edge weights
can be negative!

Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when v is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.
- Inequality holds until algorithm terminates because:
 - $\text{distTo}[w]$ cannot increase ← *distTo[] values are monotone decreasing*
 - $\text{distTo}[v]$ will not change ← *because of topological order, no edge pointing to v will be relaxed after v is relaxed*
- Thus, upon termination, shortest-paths optimality conditions hold. ■

Shortest paths in edge-weighted DAGs

```
public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        Topological topological = new Topological(G); ← topological order
        for (int v : topological.order())
            for (DirectedEdge e : G.adj(v))
                relax(e);
    }
}
```

Content-aware resizing

Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.



Shai Avidan
Mitsubishi Electric Research Lab
Ariel Shamir
The interdisciplinary Center & MERL

Content-aware resizing

Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.



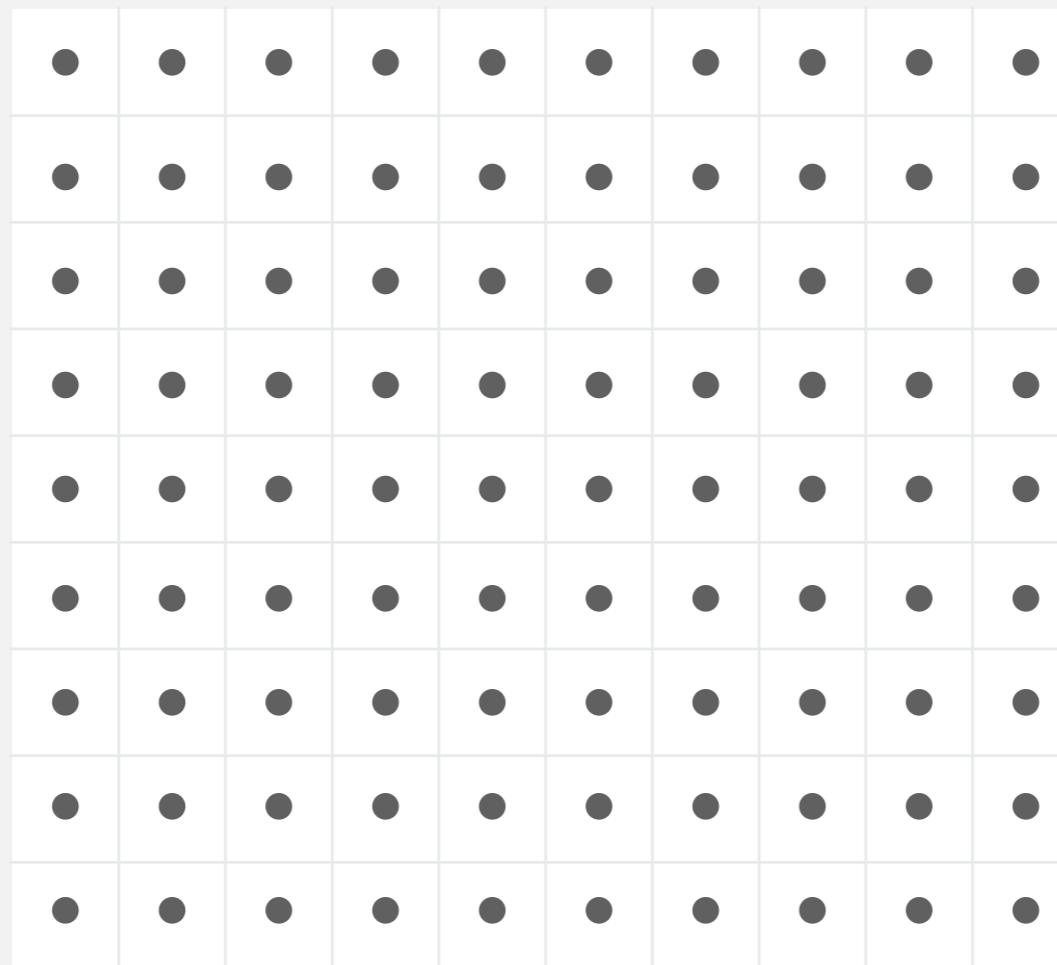
In the wild. Photoshop CS 5, Imagemagick, GIMP, ...



Content-aware resizing

To find vertical seam:

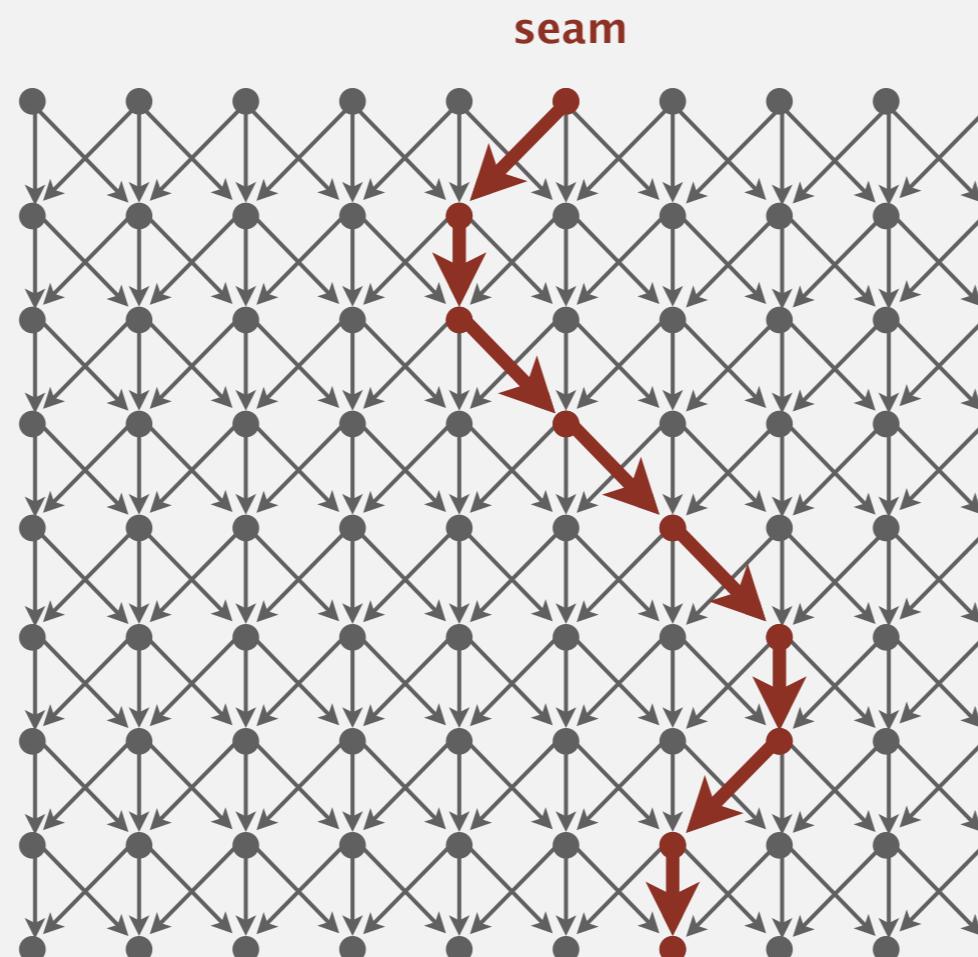
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path from top to bottom.



Content-aware resizing

To find vertical seam:

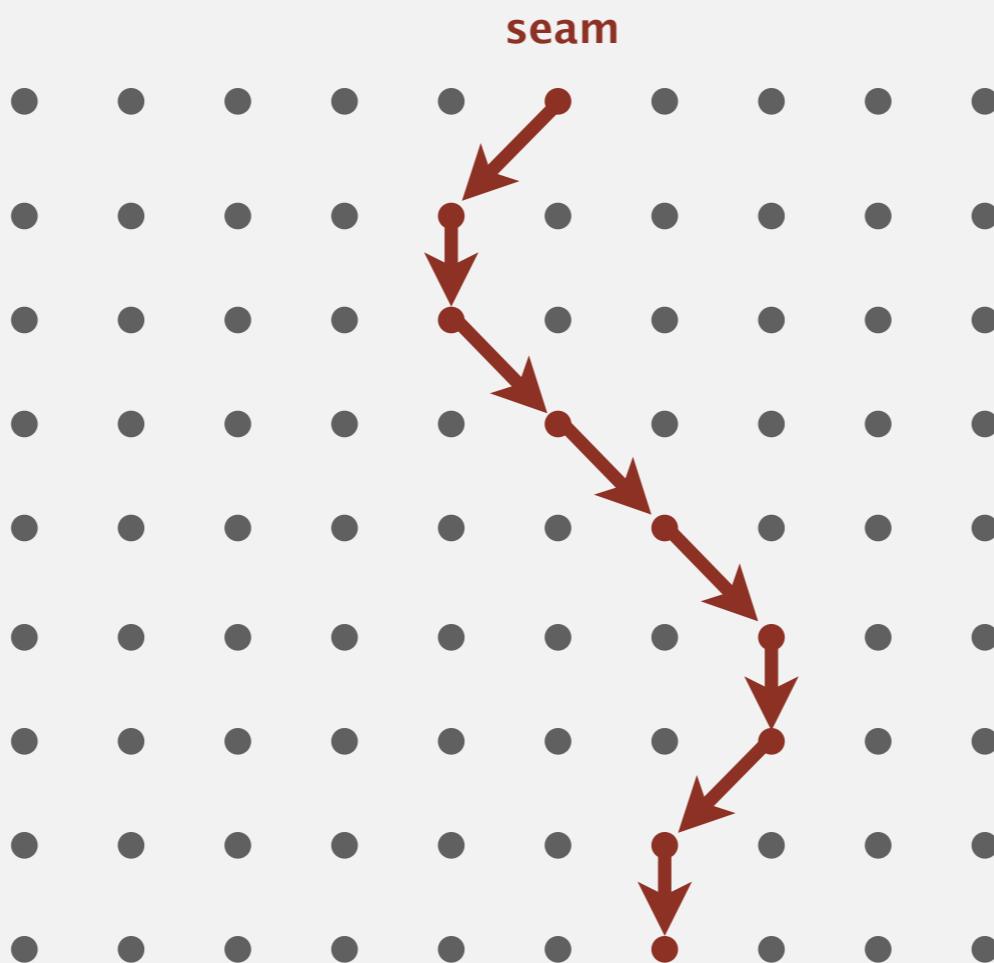
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path from top to bottom.



Content-aware resizing

To remove vertical seam:

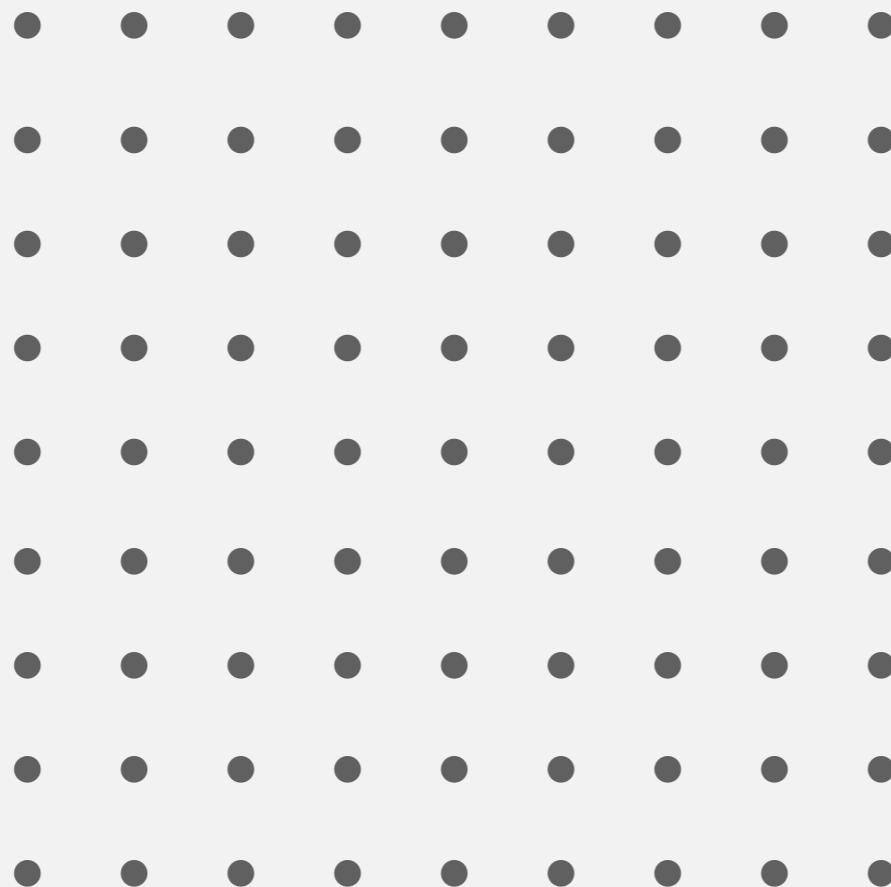
- Delete pixels on seam (one in each row).



Content-aware resizing

To remove vertical seam:

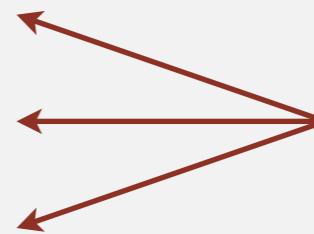
- Delete pixels on seam (one in each row).



Longest paths in edge-weighted DAGs

Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
- Find shortest paths.
- Negate weights in result.



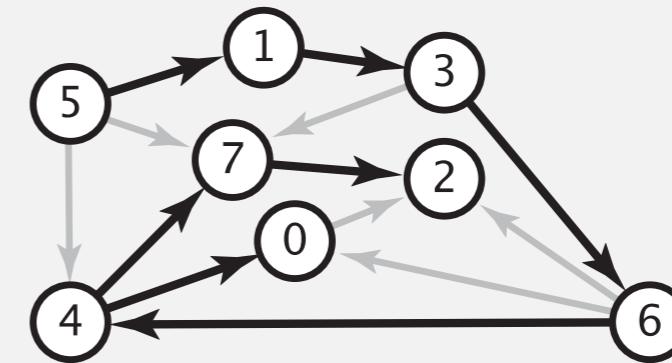
equivalent: reverse sense of equality in `relax()`

longest paths input

5->4	0.35
4->7	0.37
5->7	0.28
5->1	0.32
4->0	0.38
0->2	0.26
3->7	0.39
1->3	0.29
7->2	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

shortest paths input

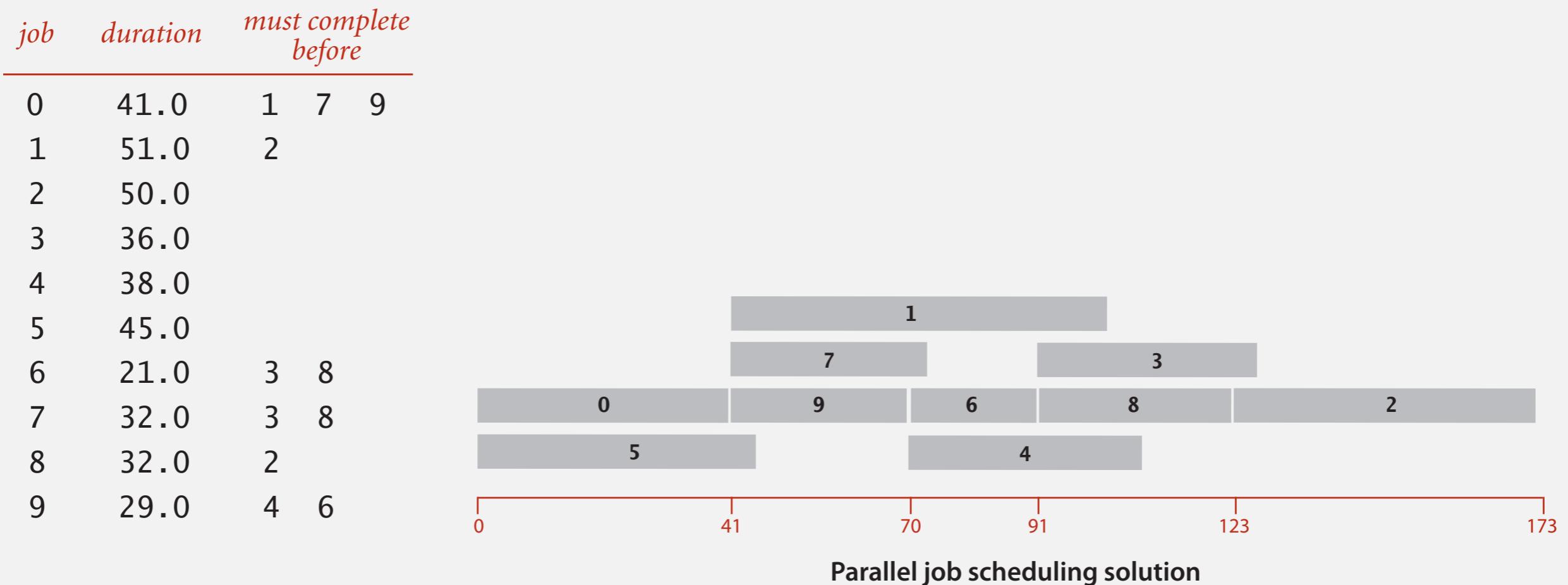
5->4	-0.35
4->7	-0.37
5->7	-0.28
5->1	-0.32
4->0	-0.38
0->2	-0.26
3->7	-0.39
1->3	-0.29
7->2	-0.34
6->2	-0.40
3->6	-0.52
6->0	-0.58
6->4	-0.93



Key point. Topological sort algorithm works even with negative edge weights.

Longest paths in edge-weighted DAGs: application

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

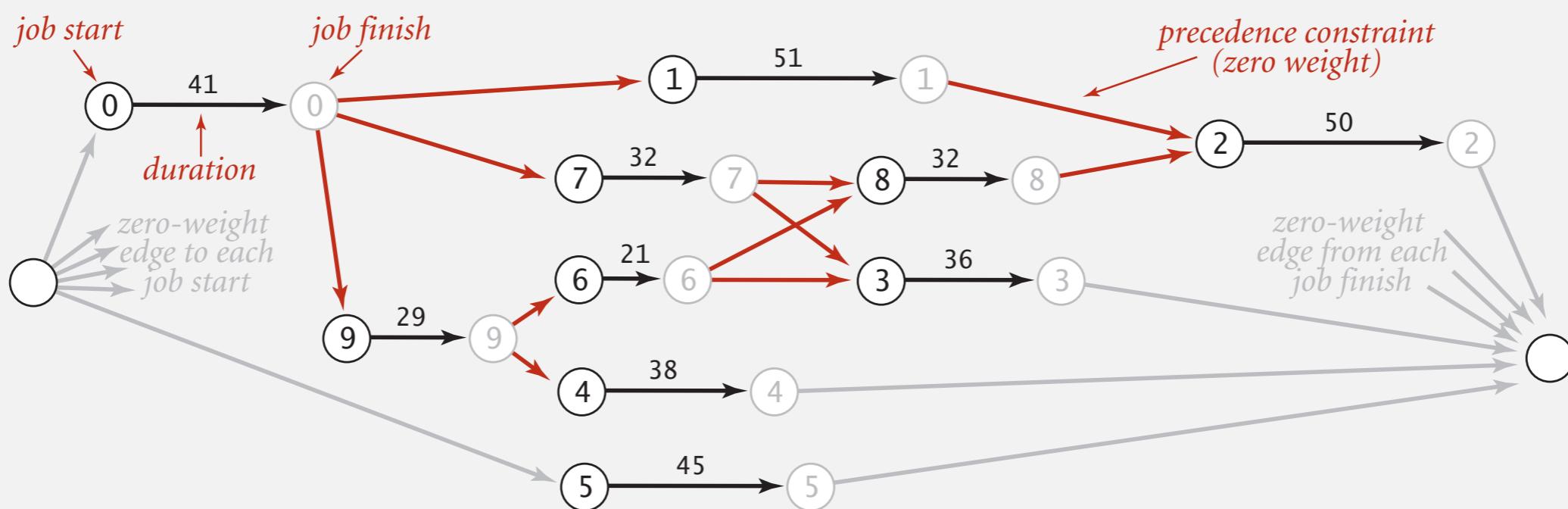


Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

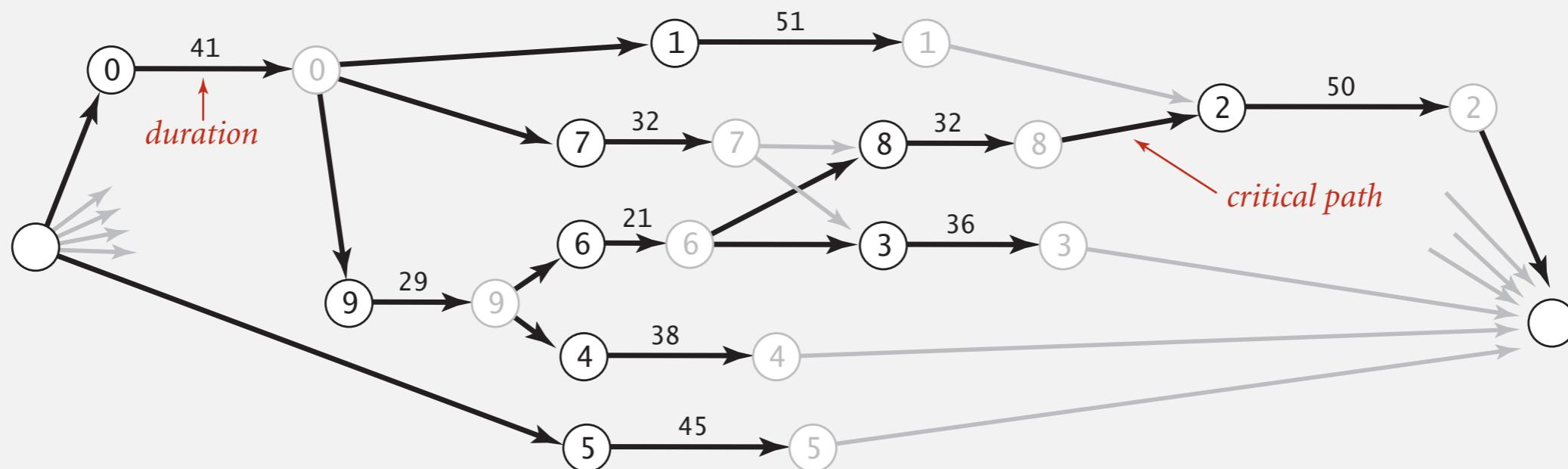
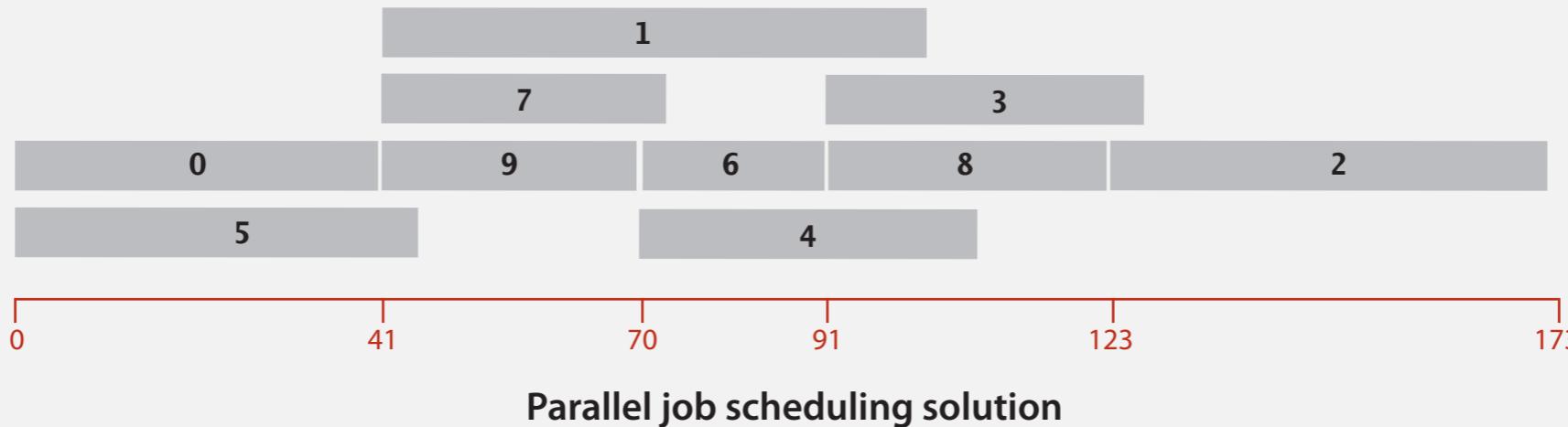
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
 - begin to end (weighted by duration)
 - source to begin (0 weight)
 - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

job	duration	must complete before		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	



Critical path method

CPM. Use longest path from the source to schedule each job.

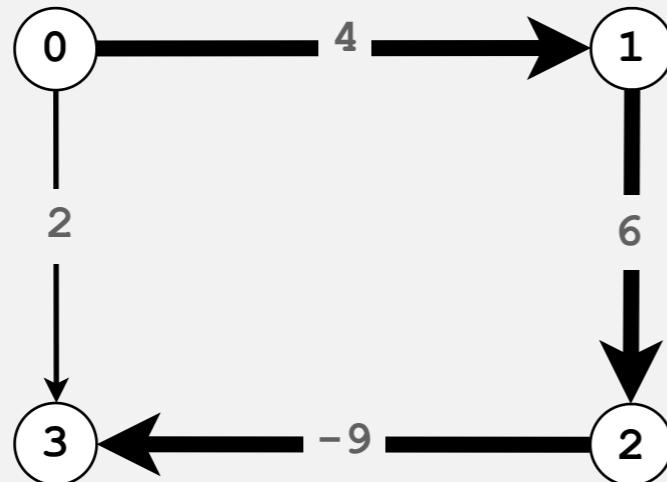


SHORTEST PATHS

- ▶ Edge-weighted digraph API
- ▶ Shortest-paths properties
- ▶ Dijkstra's algorithm
- ▶ Edge-weighted DAGs
- ▶ Negative weights

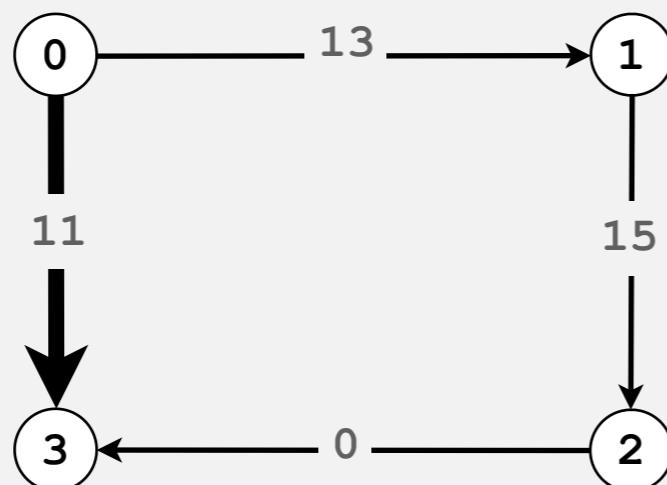
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 9 to each edge weight changes the
shortest path from $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ to $0 \rightarrow 3$.

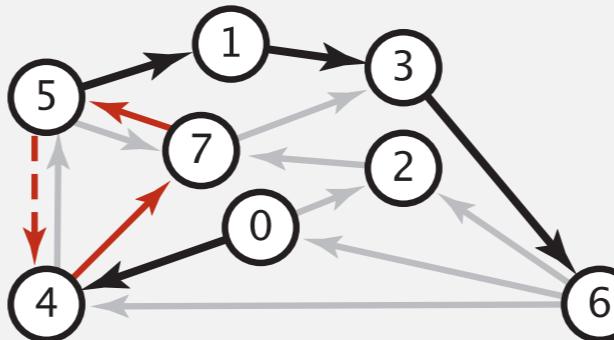
Bad news. Need a different algorithm.

Negative cycles

Def. A **negative cycle** is a directed cycle whose sum of edge weights is negative.

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

shortest path from 0 to 6

0->4->7->5->4->7->5...>1->3->6

Proposition. A SPT exists iff no negative cycles.



assuming all vertices reachable from s

Bellman-Ford algorithm

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

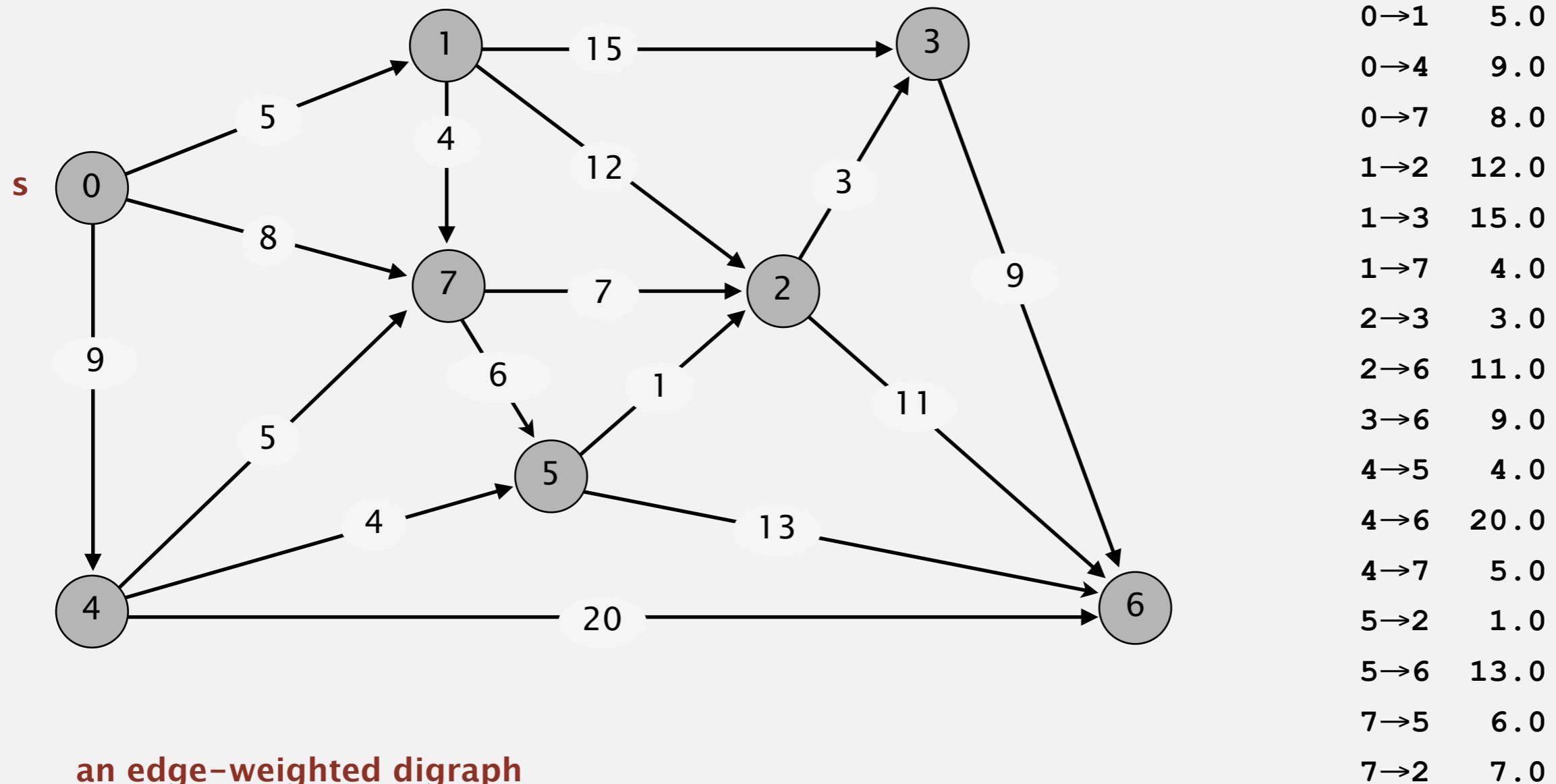
- Relax each edge.

```
for (int i = 0; i < G.V(); i++)
    for (int v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```

← pass i (relax each edge)

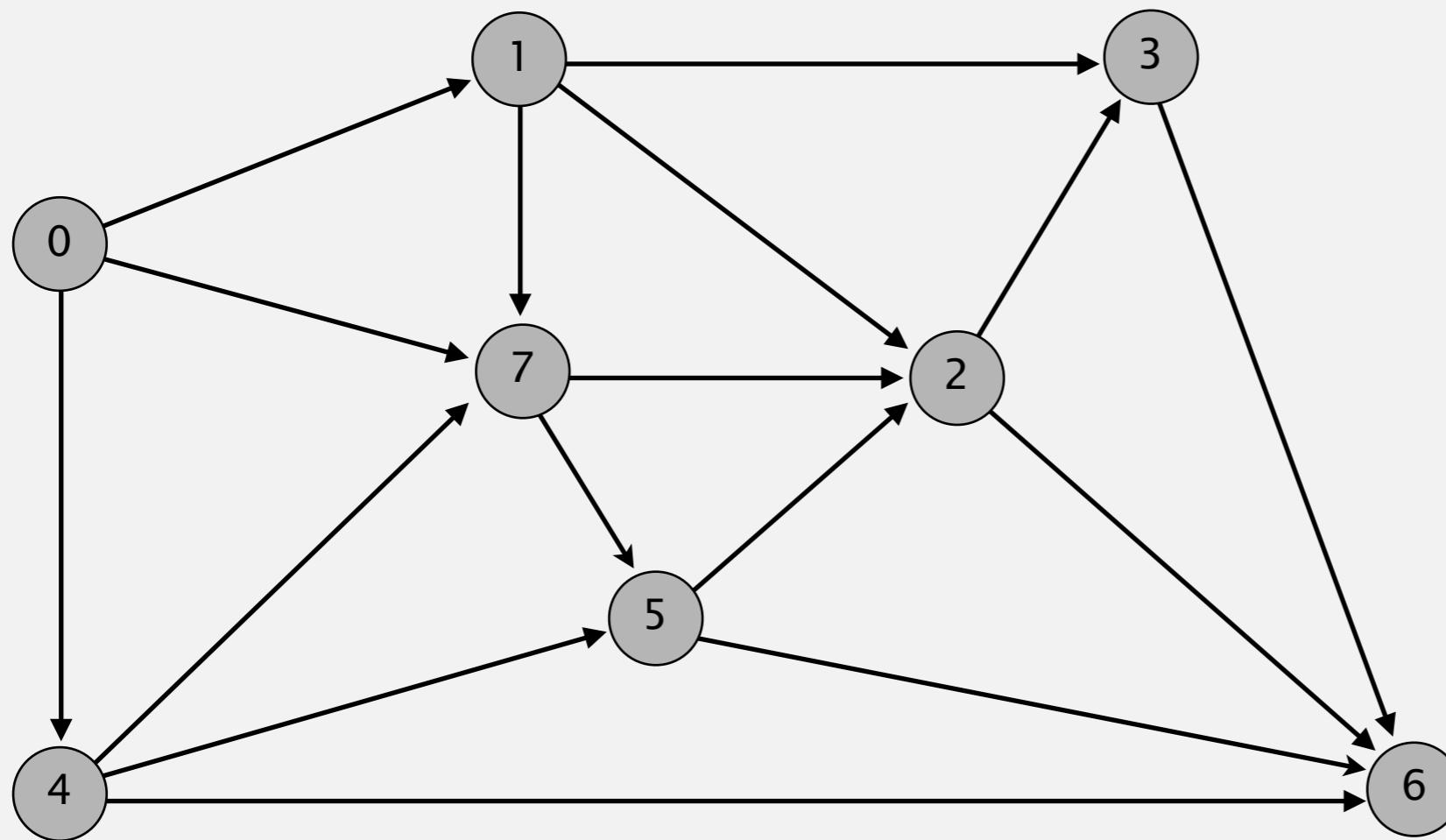
Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

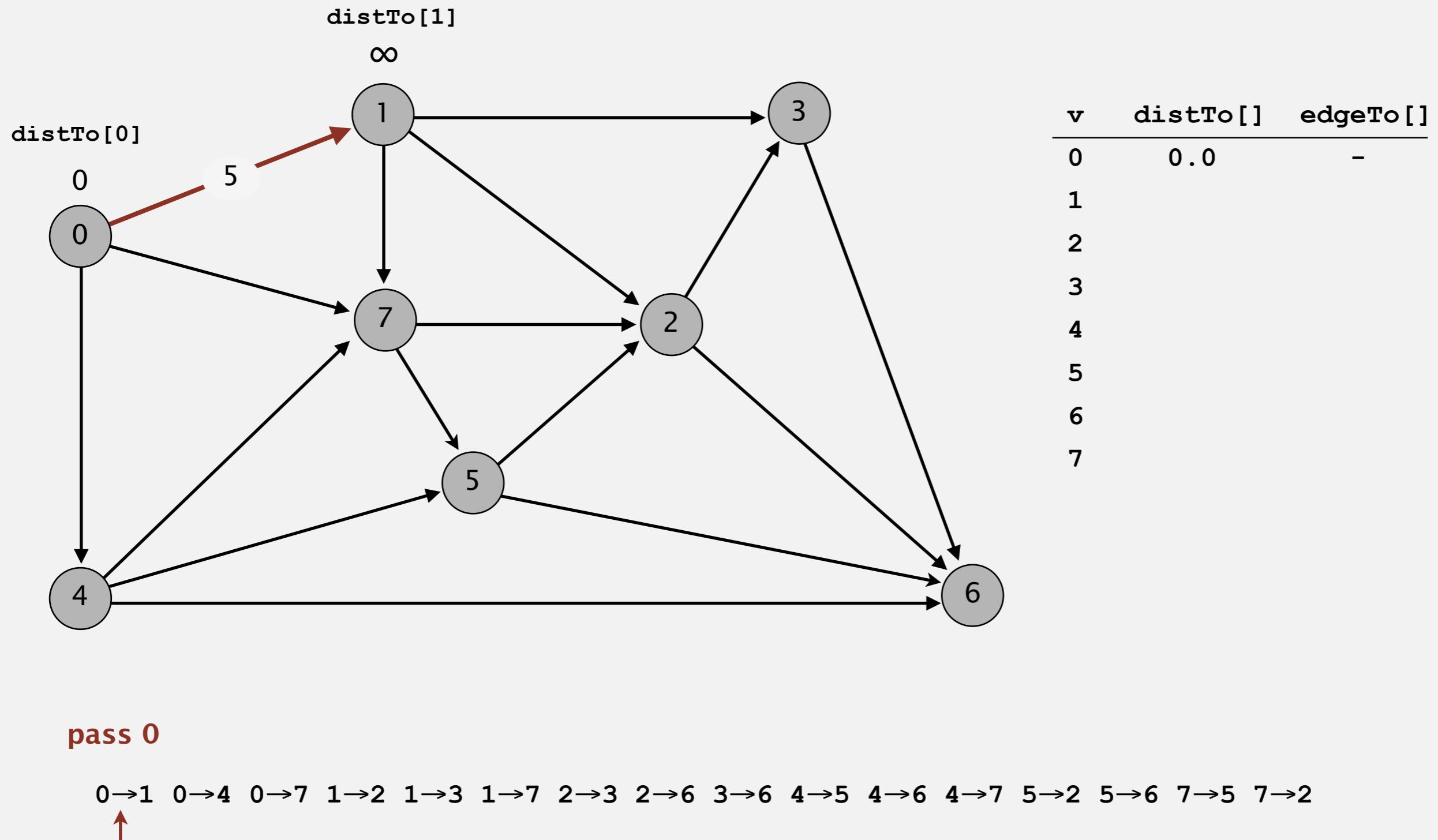


initialize

v	distTo []	edgeTo []
0	0.0	-
1		
2		
3		
4		
5		
6		
7		

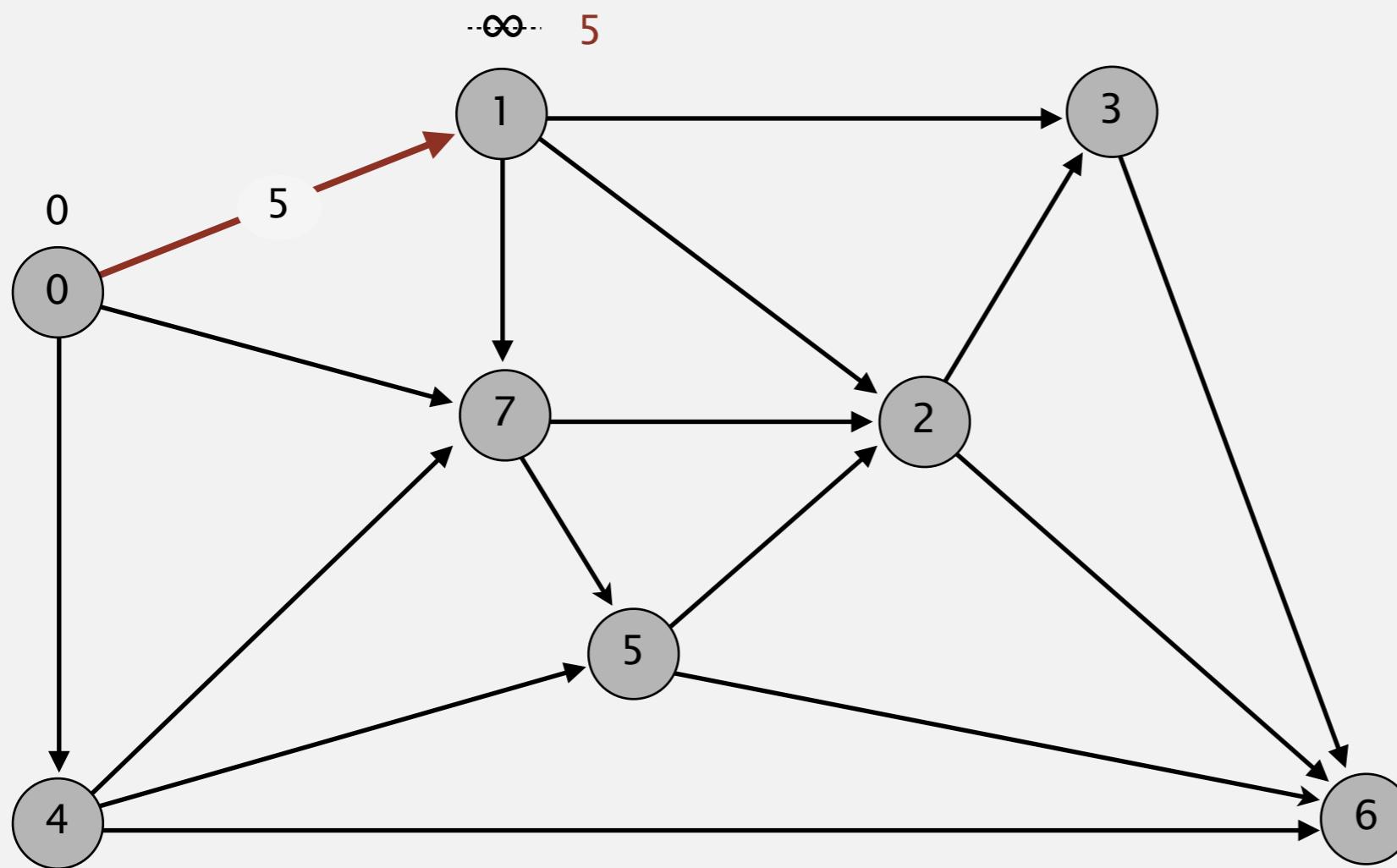
Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



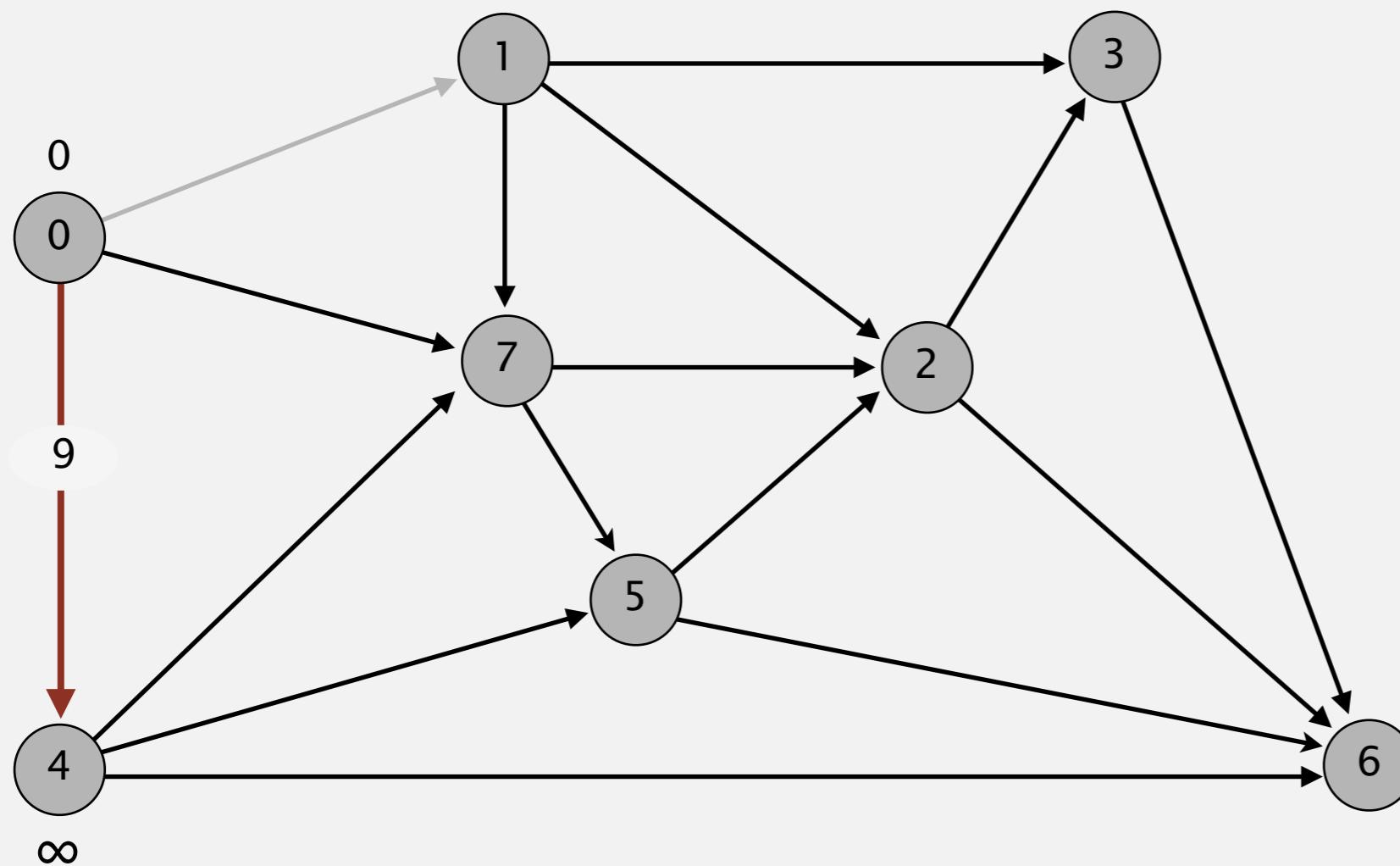
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
↑

v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

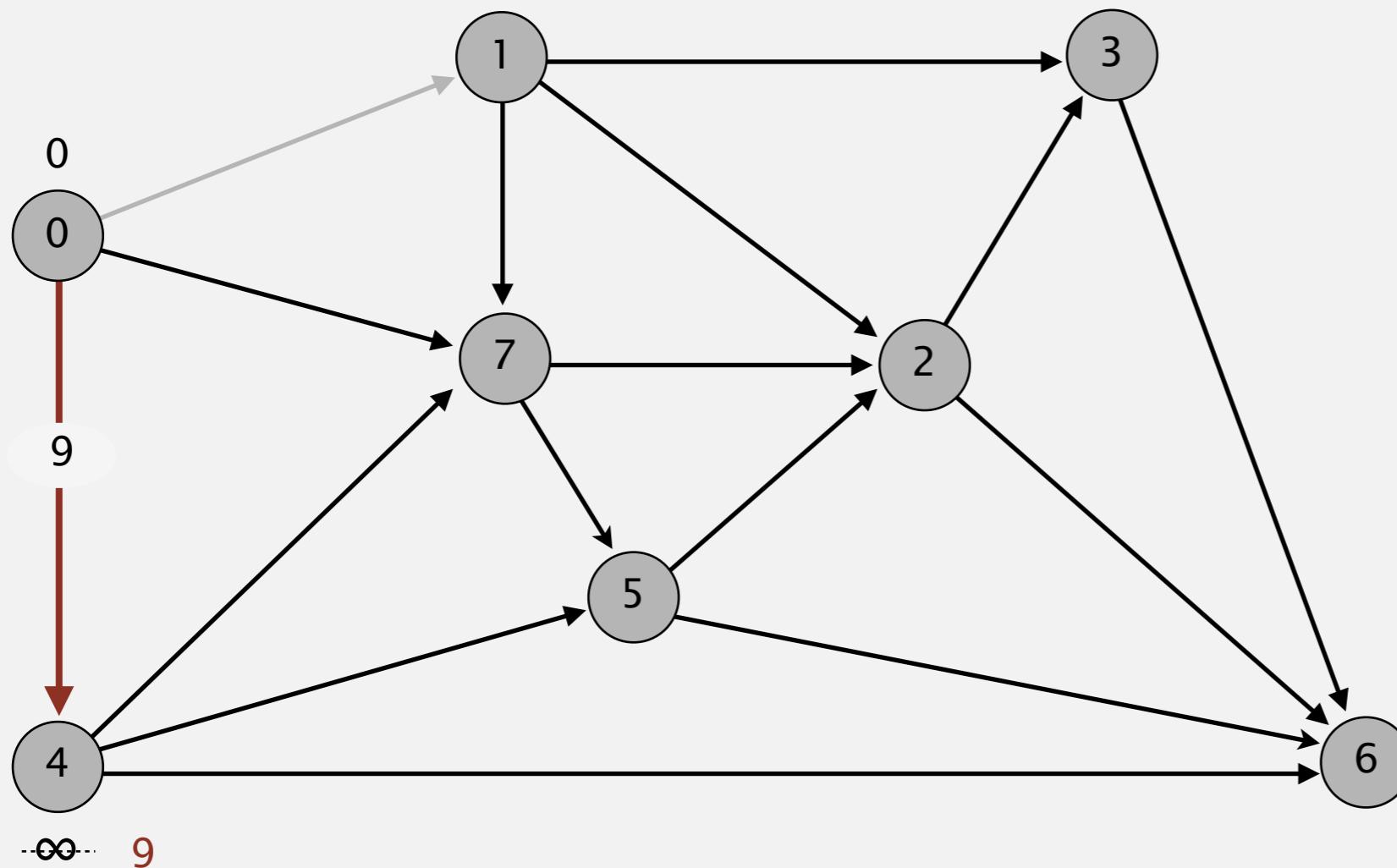
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4		
5		
6		
7		

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

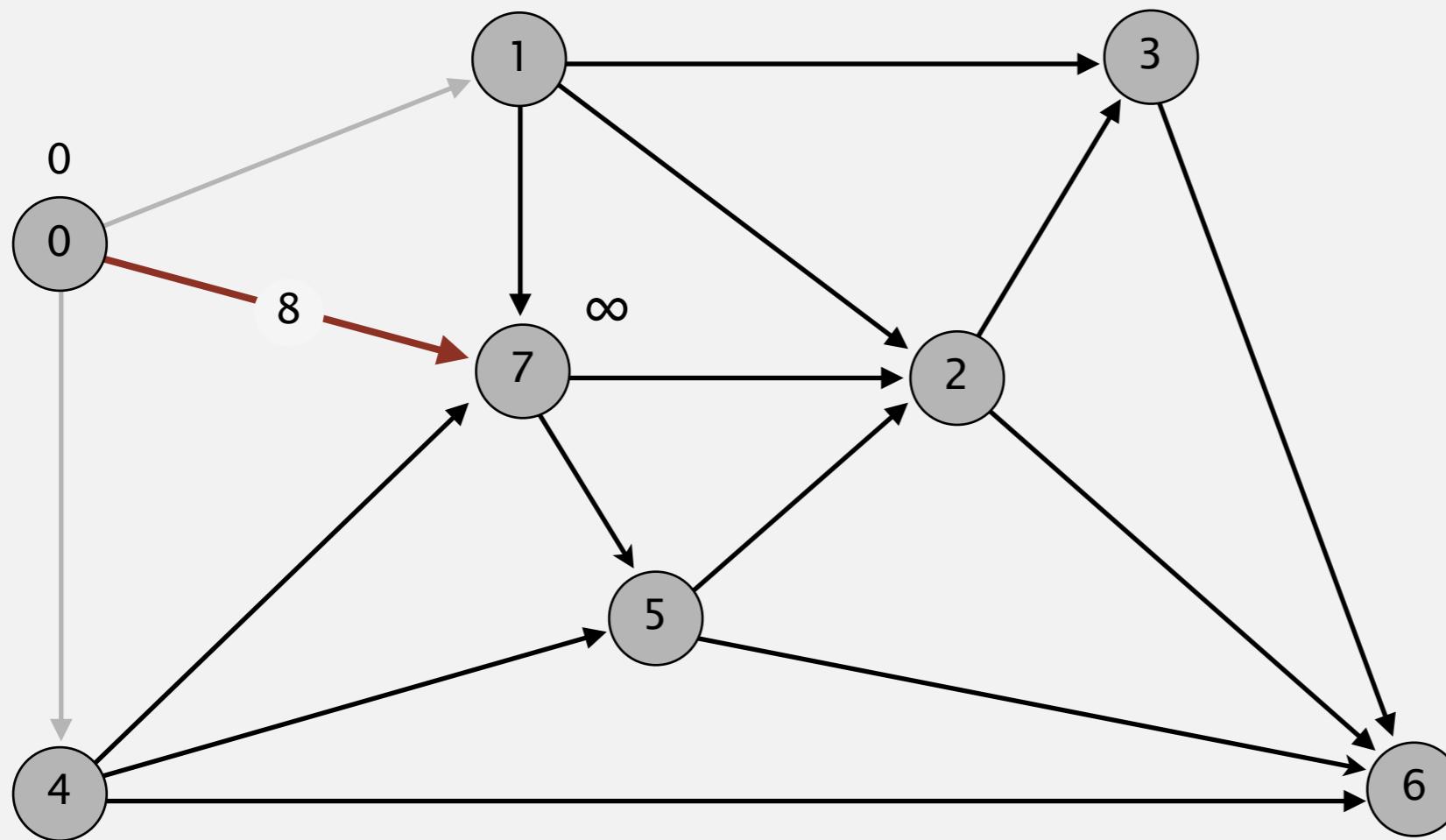
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

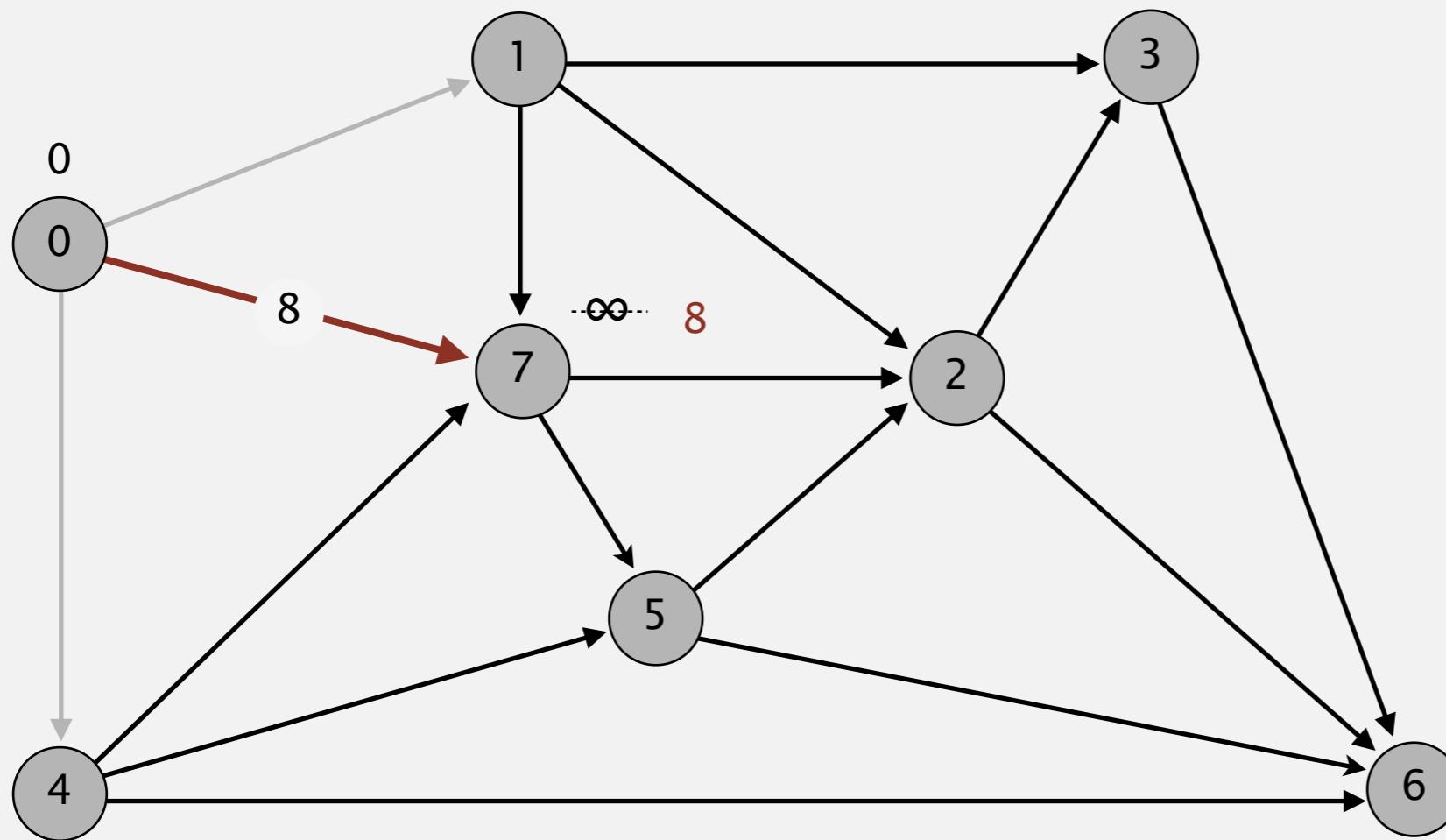
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7		

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

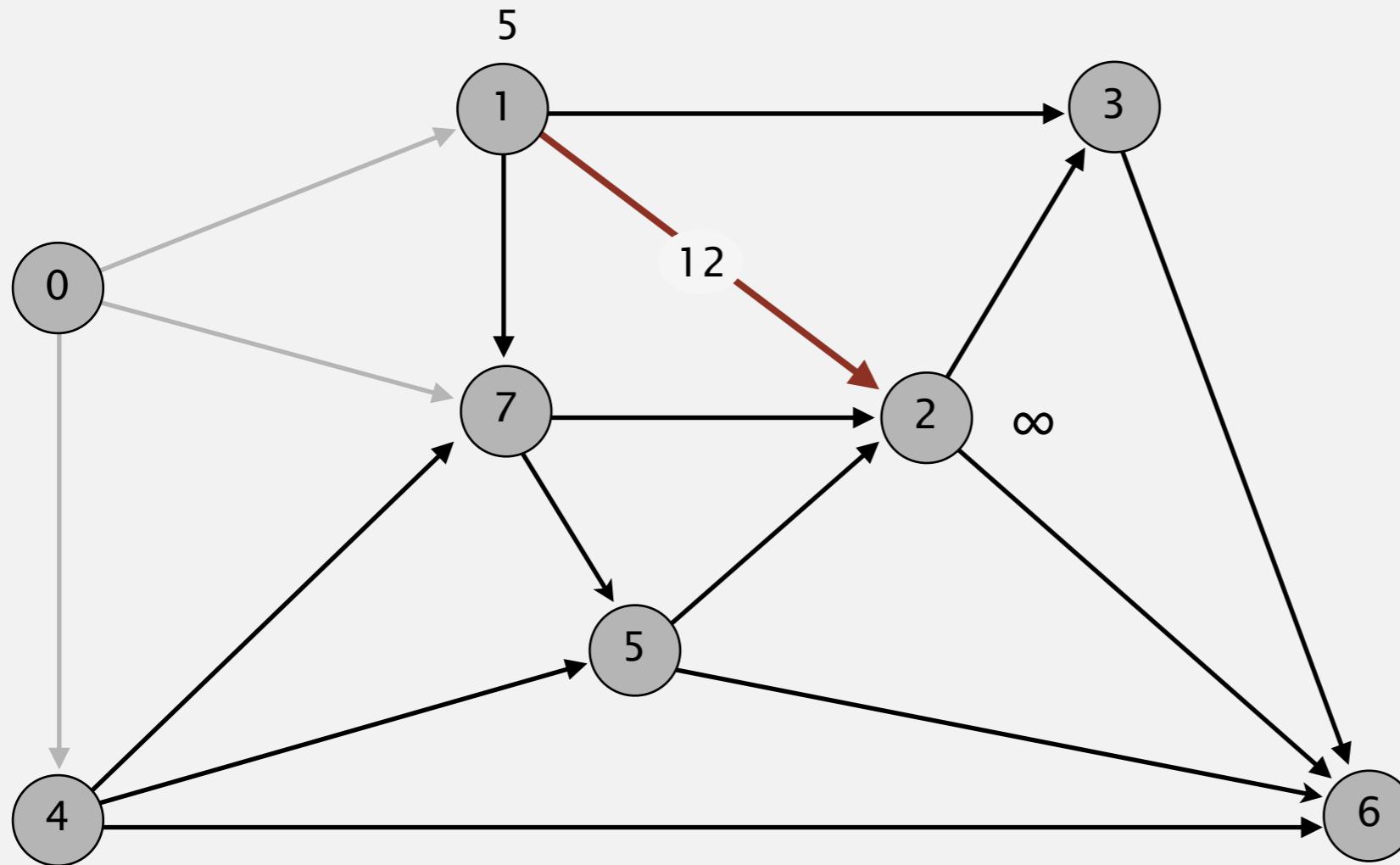
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

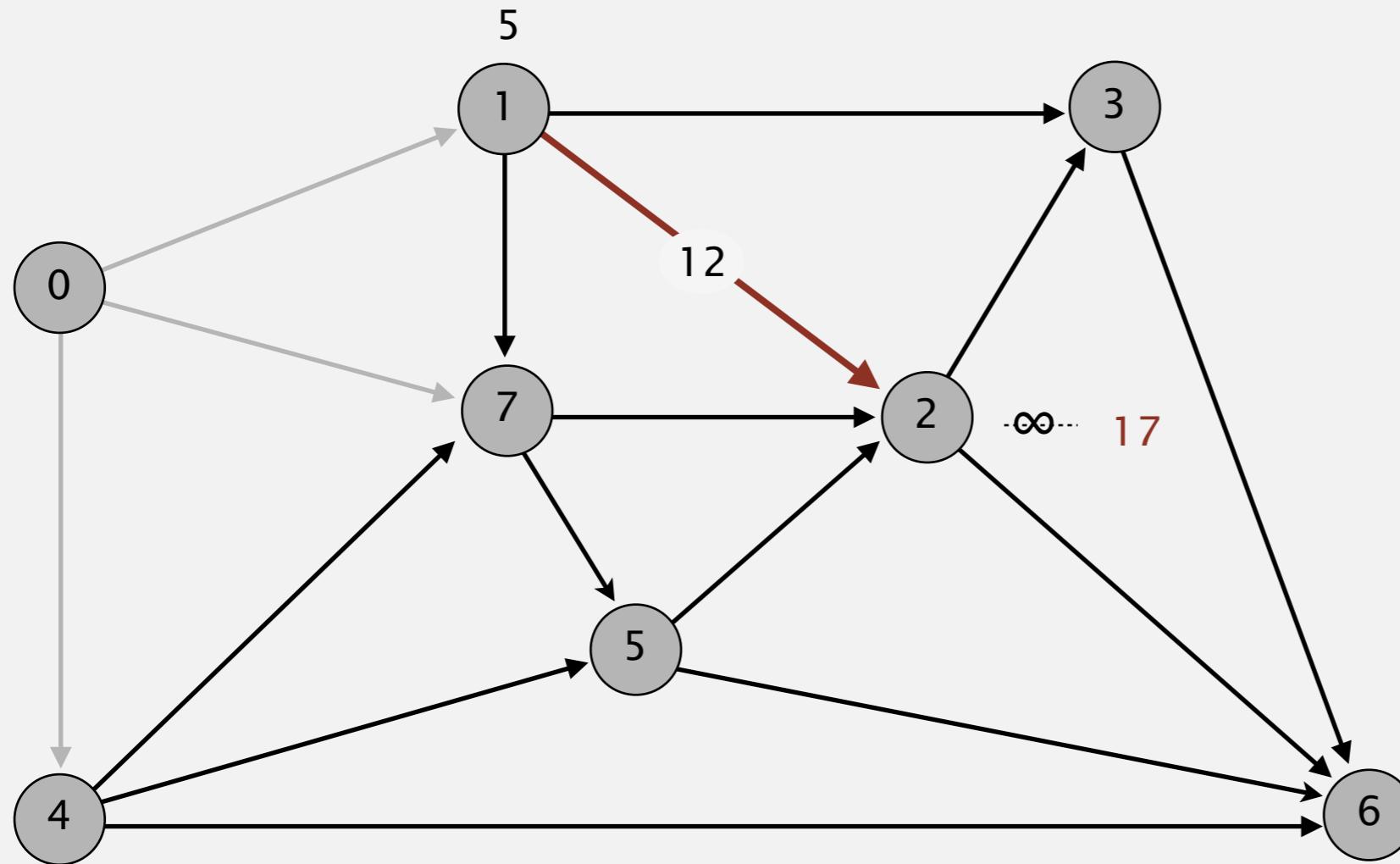
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo []	edgeTo []
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

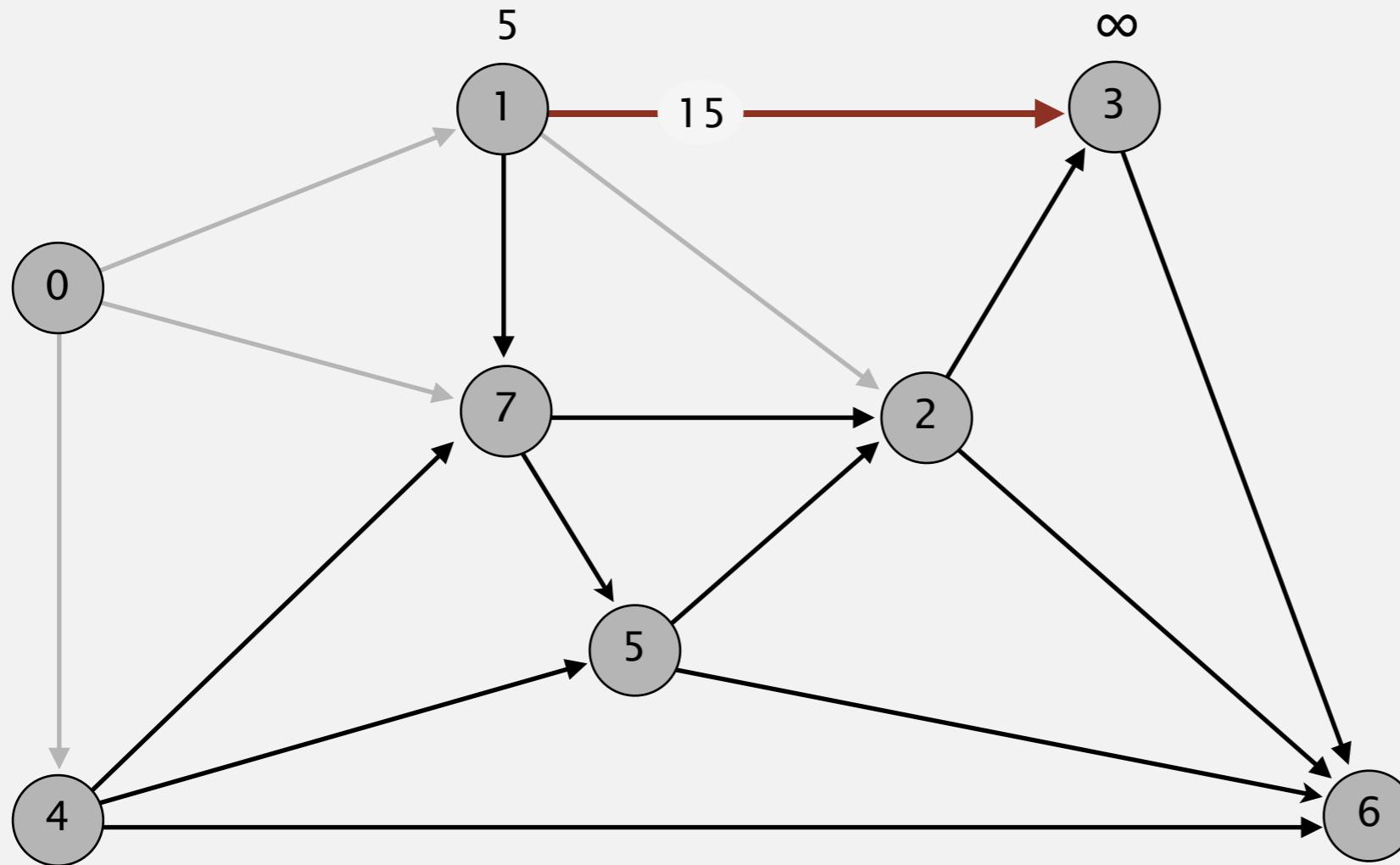
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

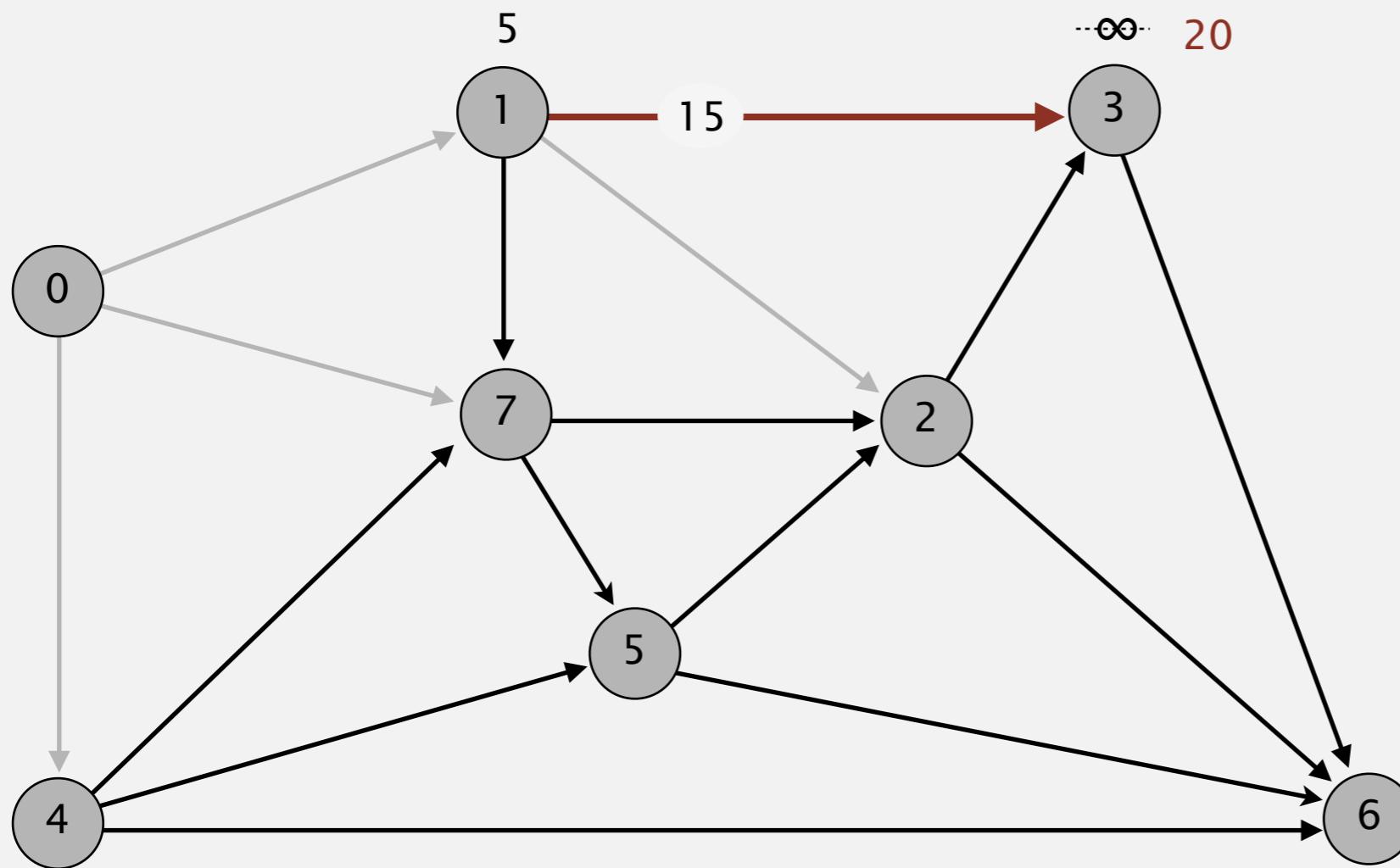
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

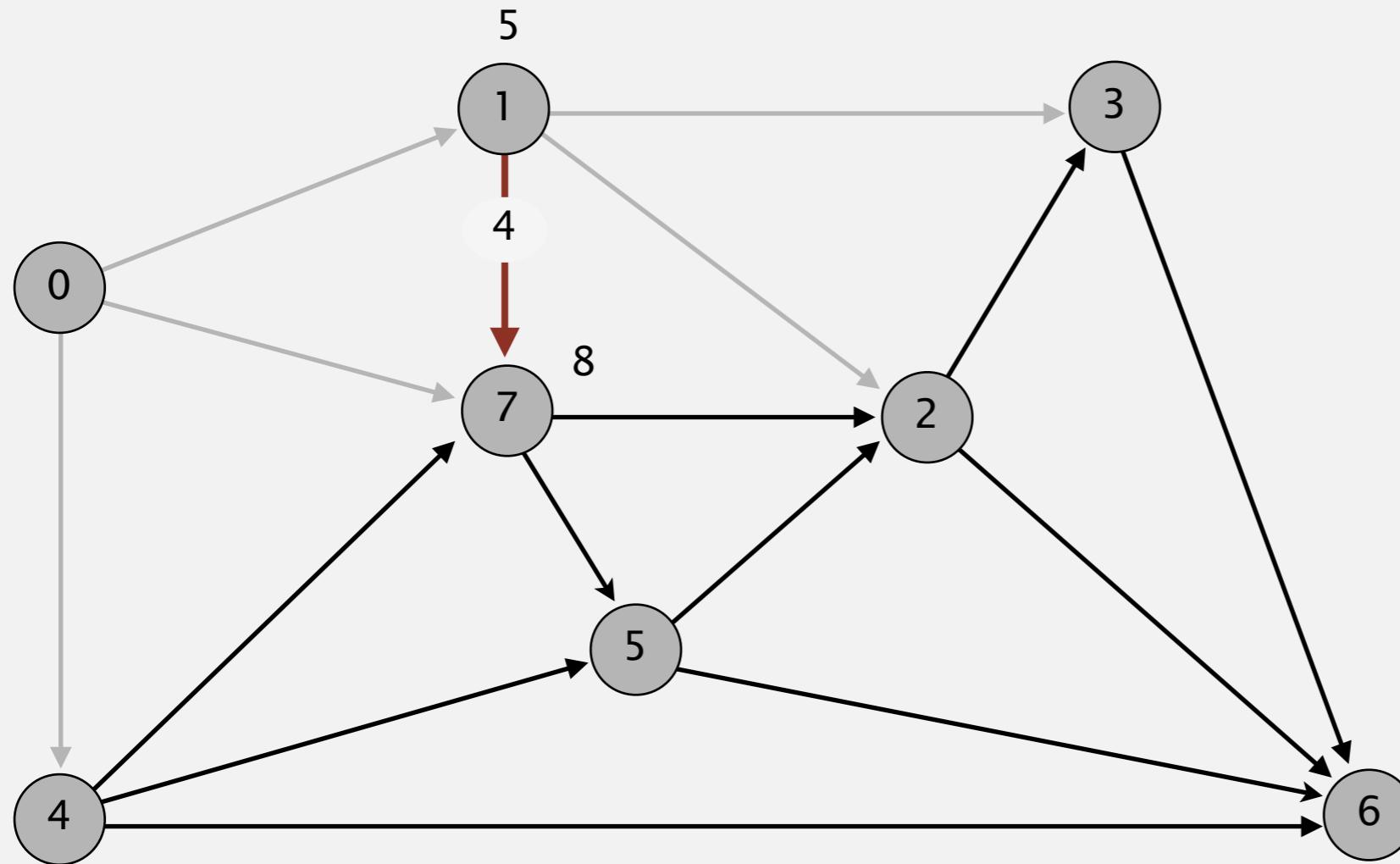
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

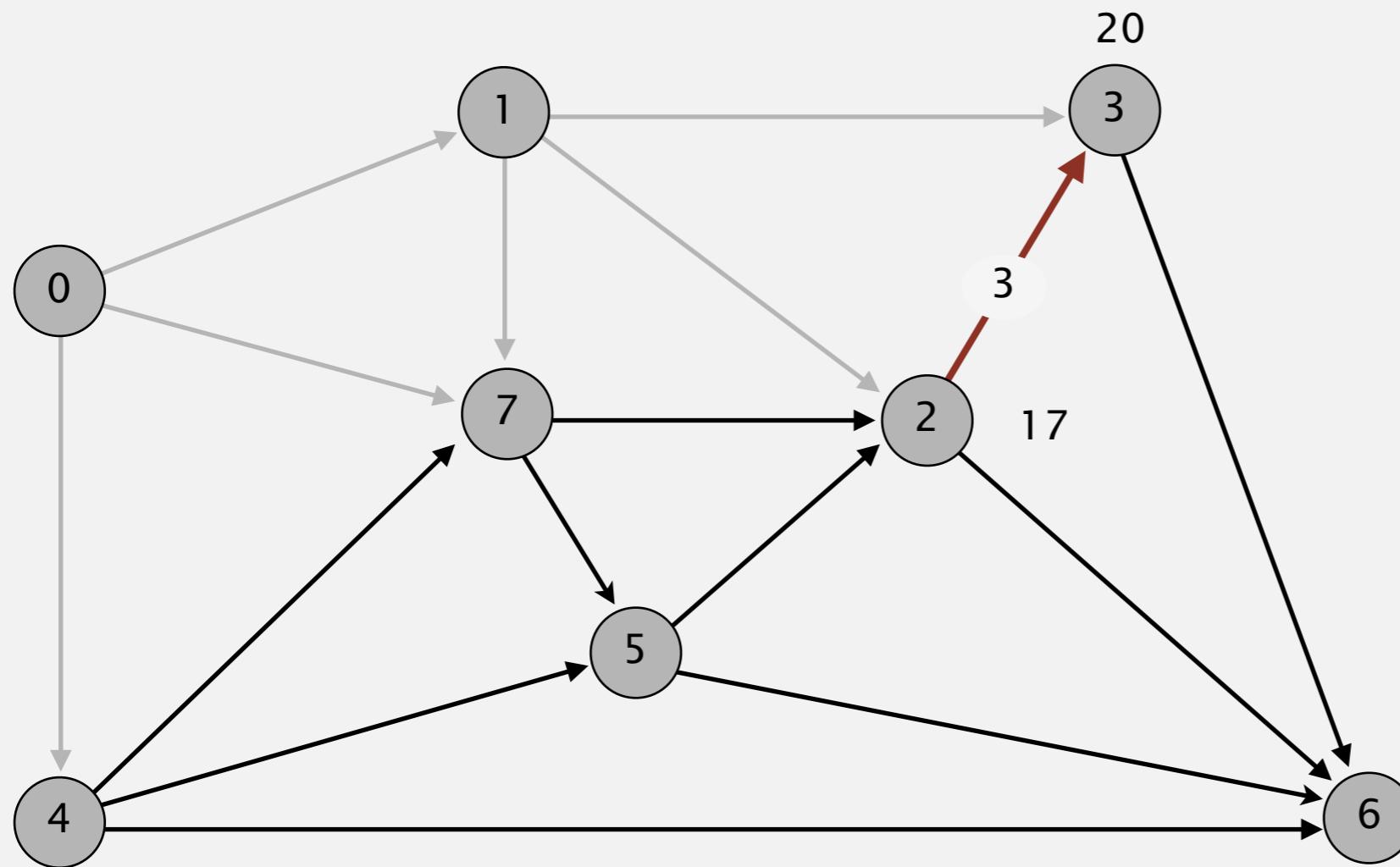
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

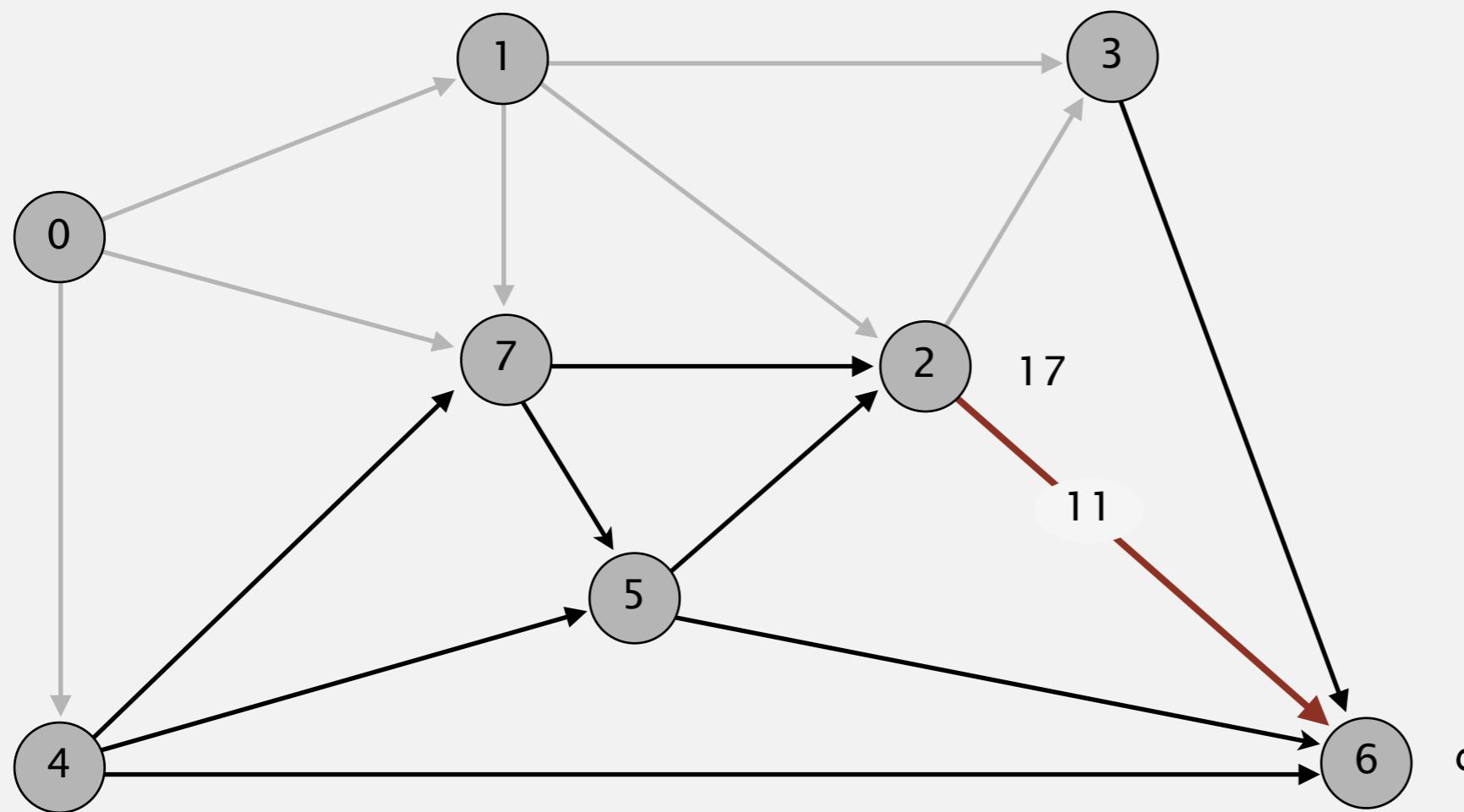
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6		
7	8.0	0→7

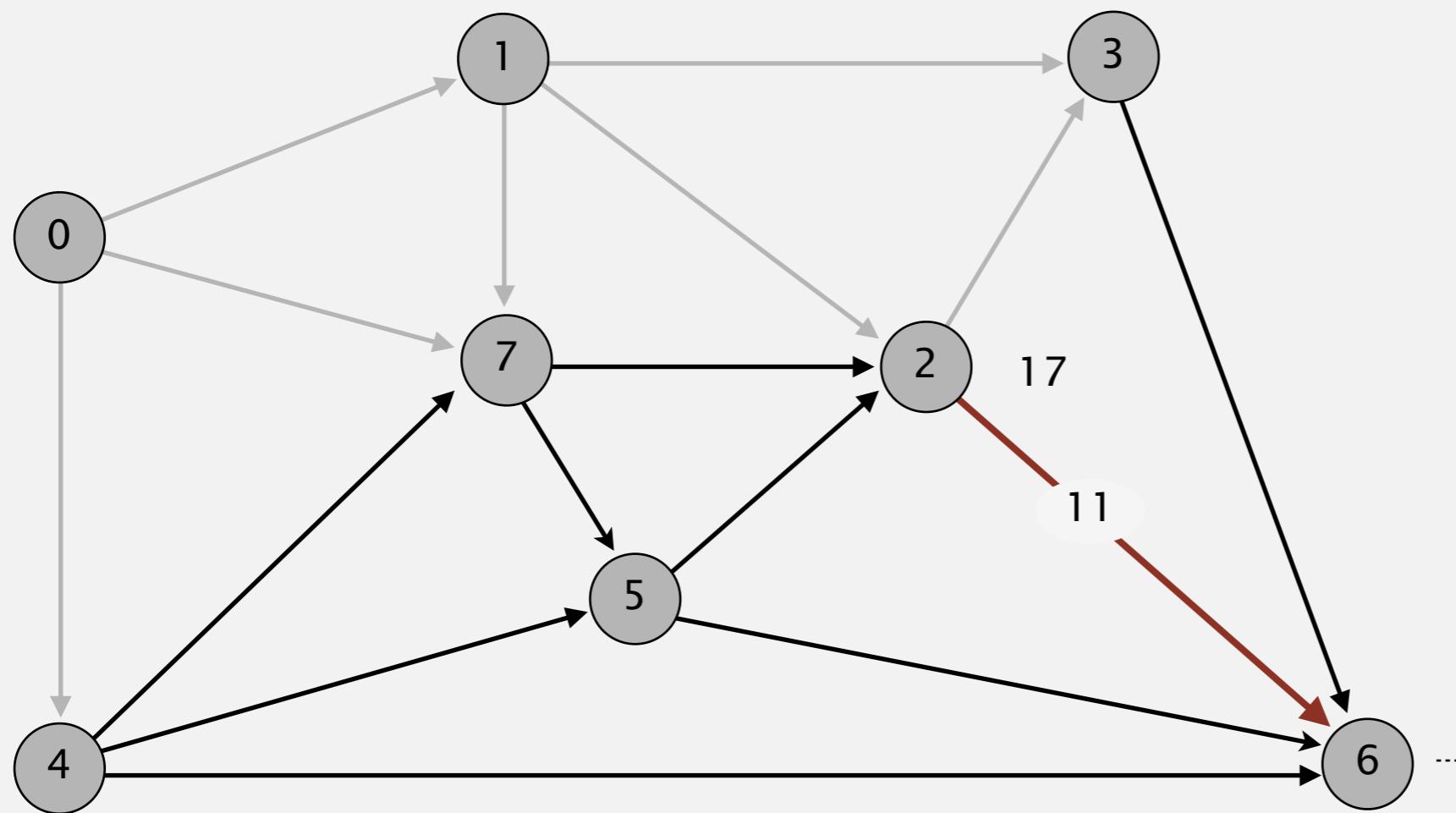
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

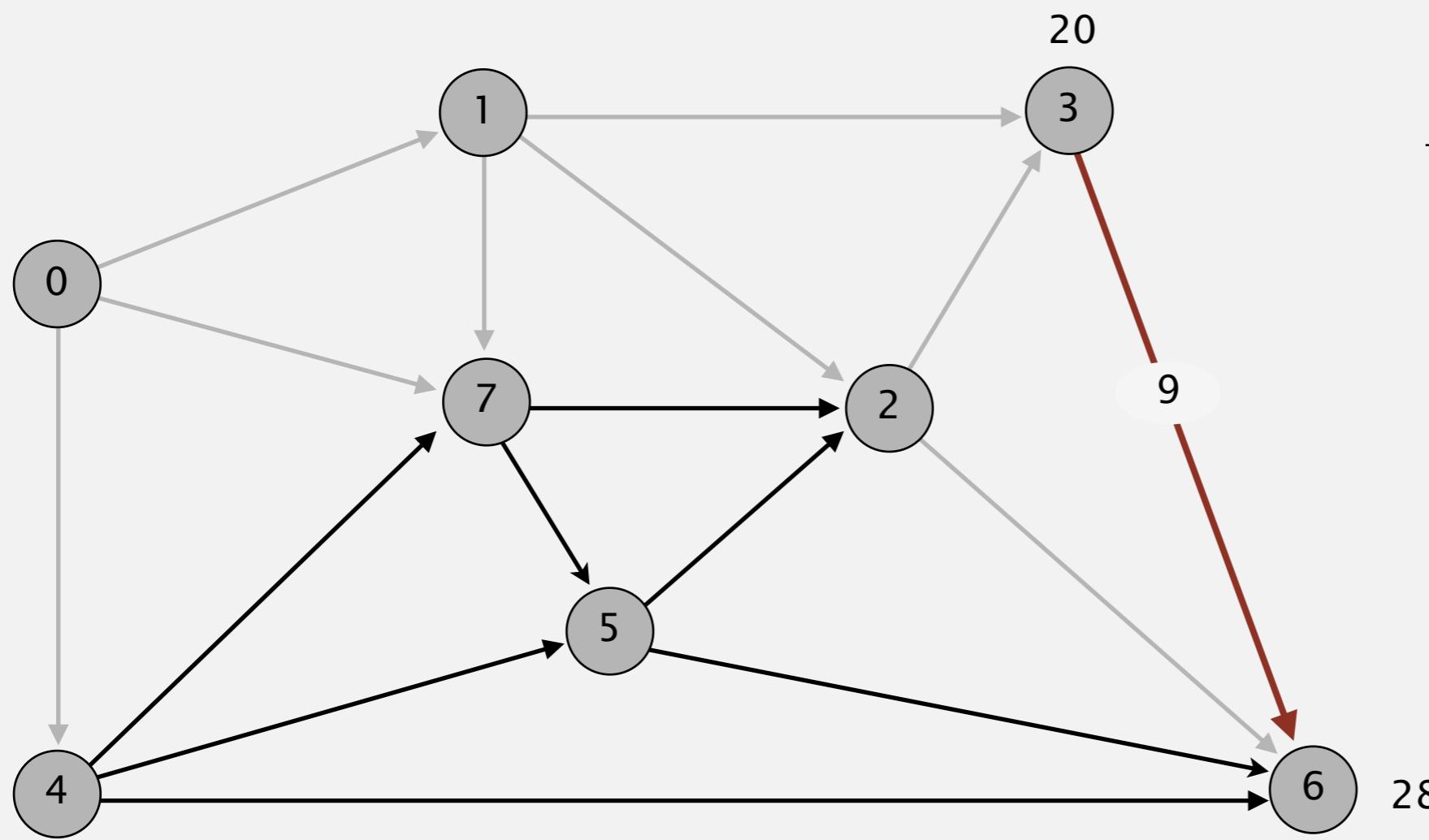
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



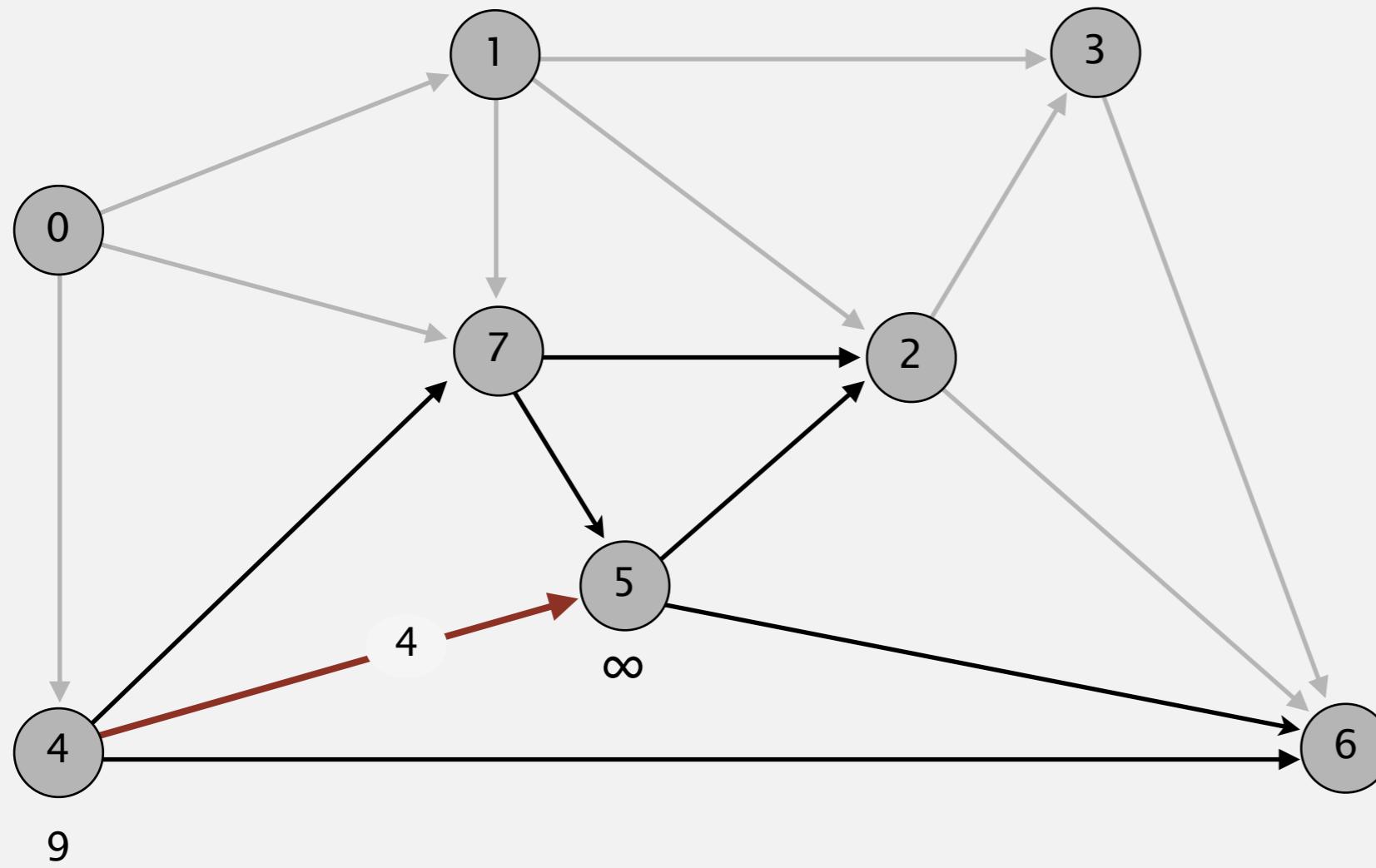
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

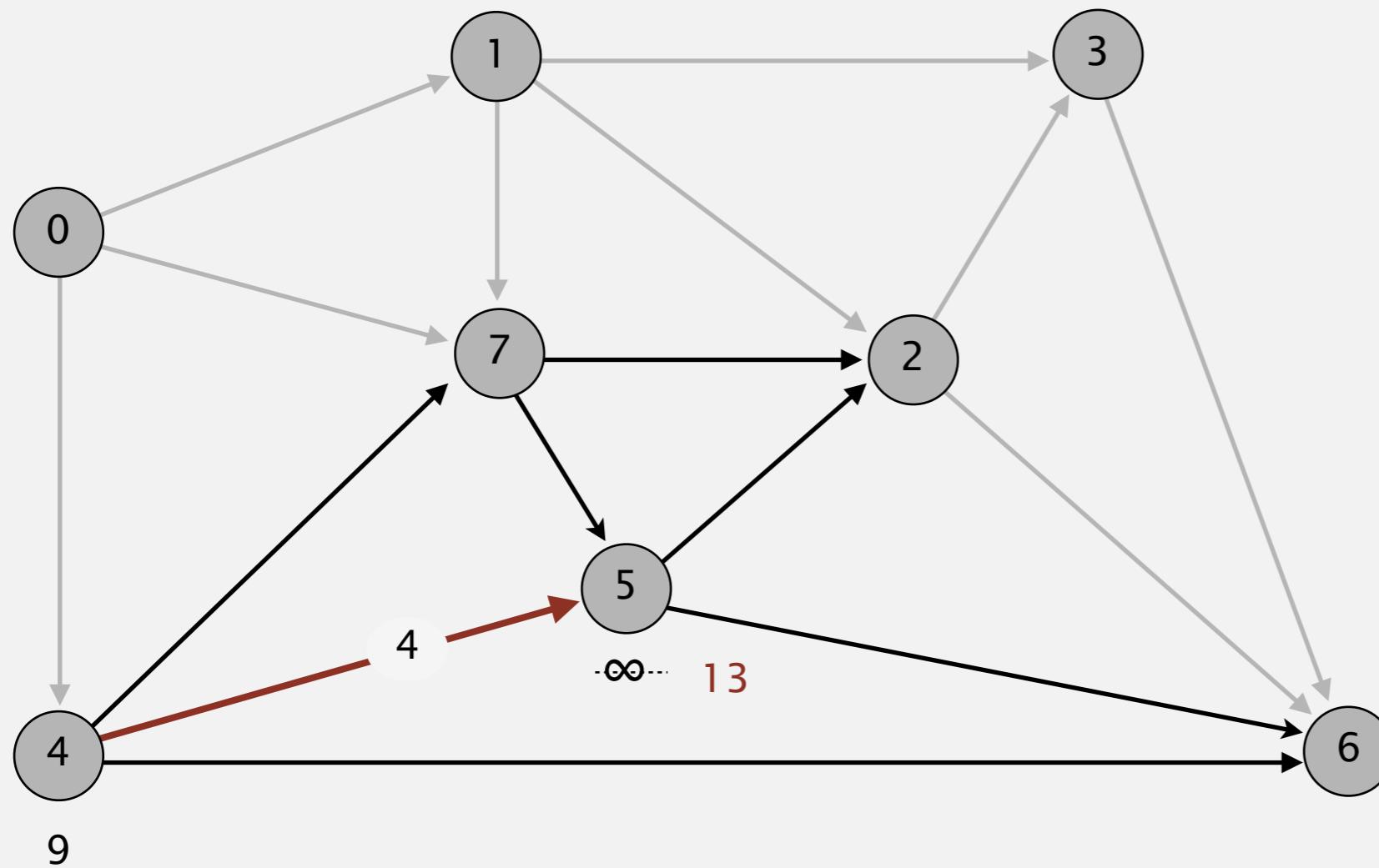
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5		
6	28.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



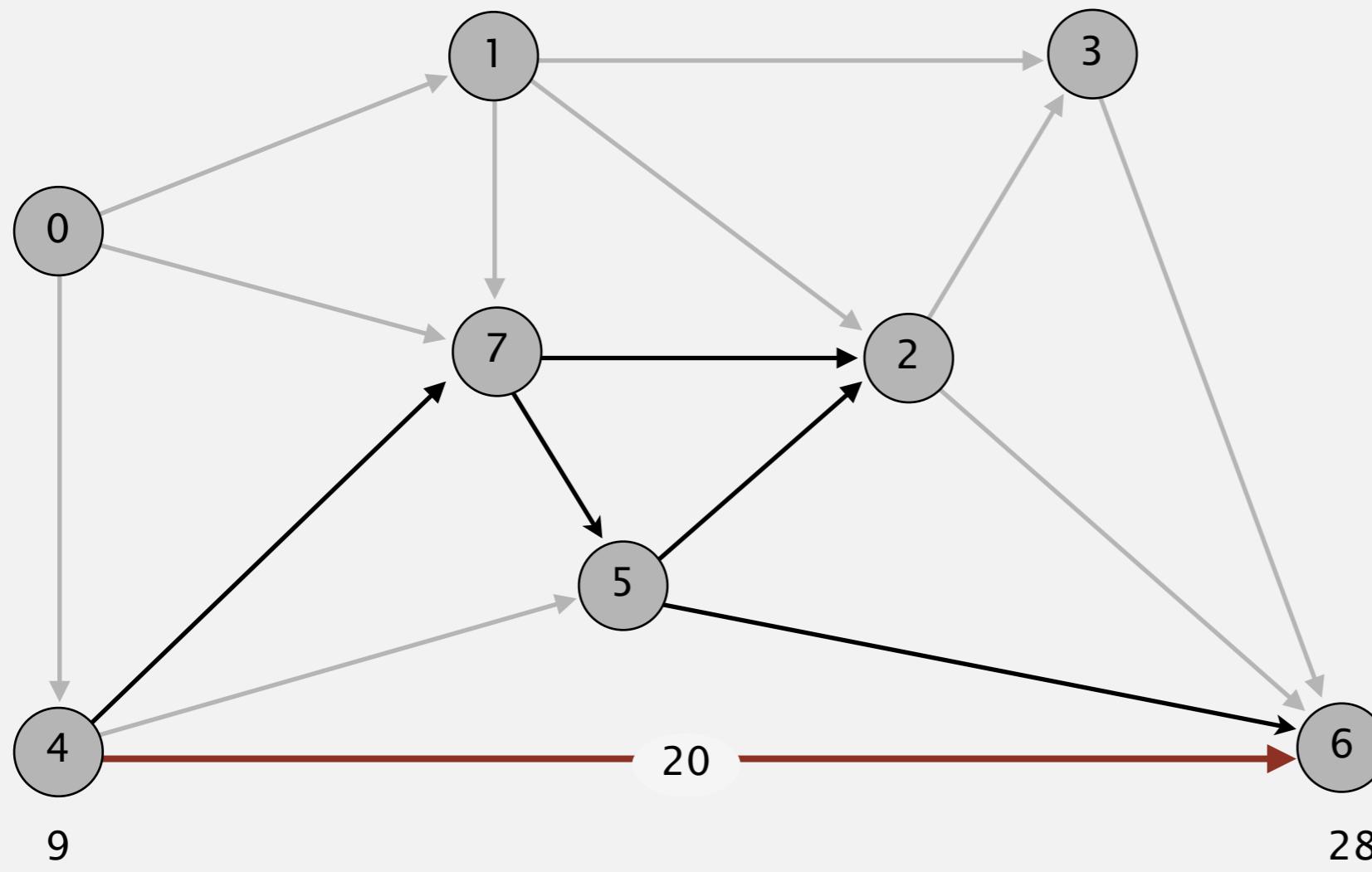
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

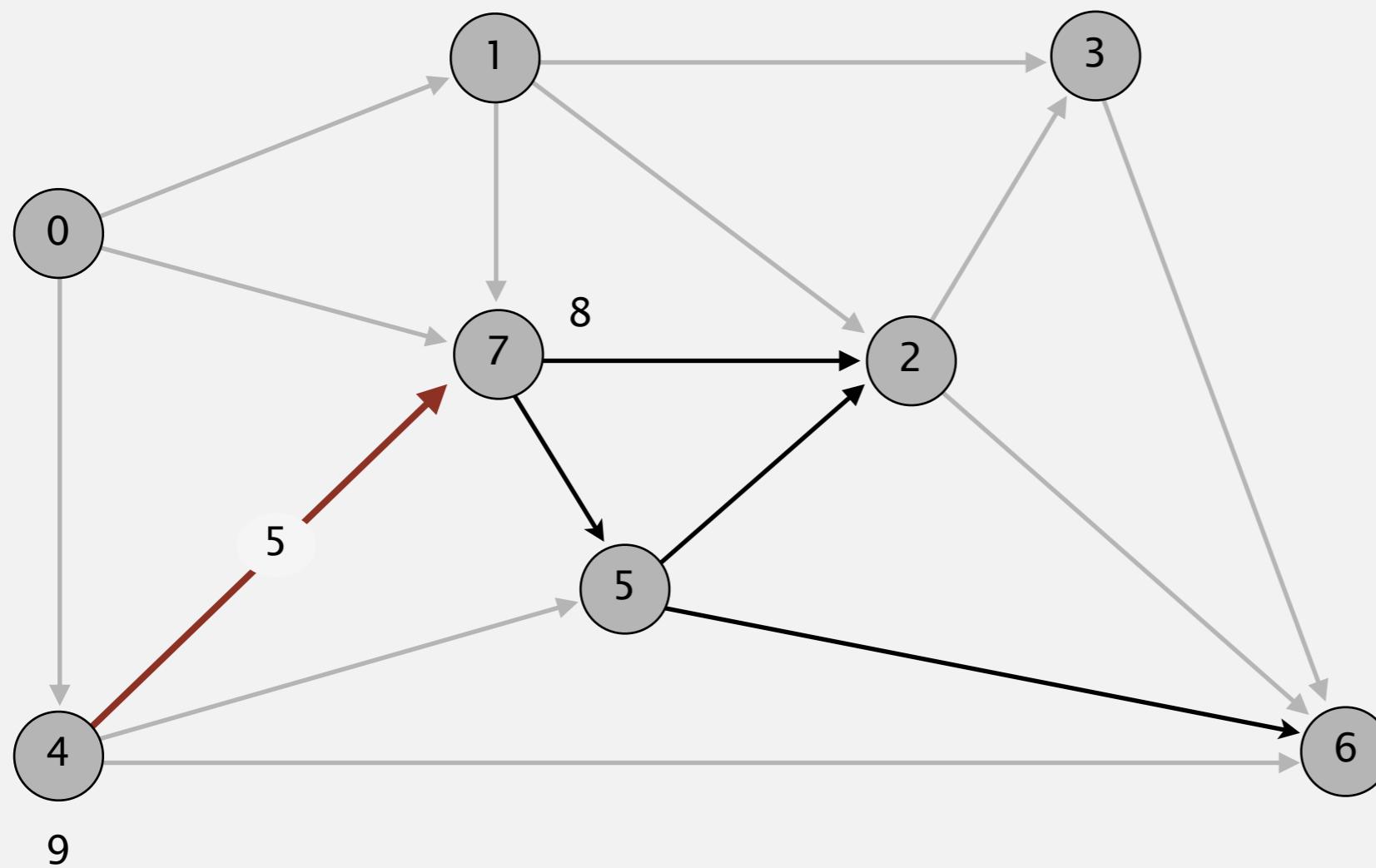
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



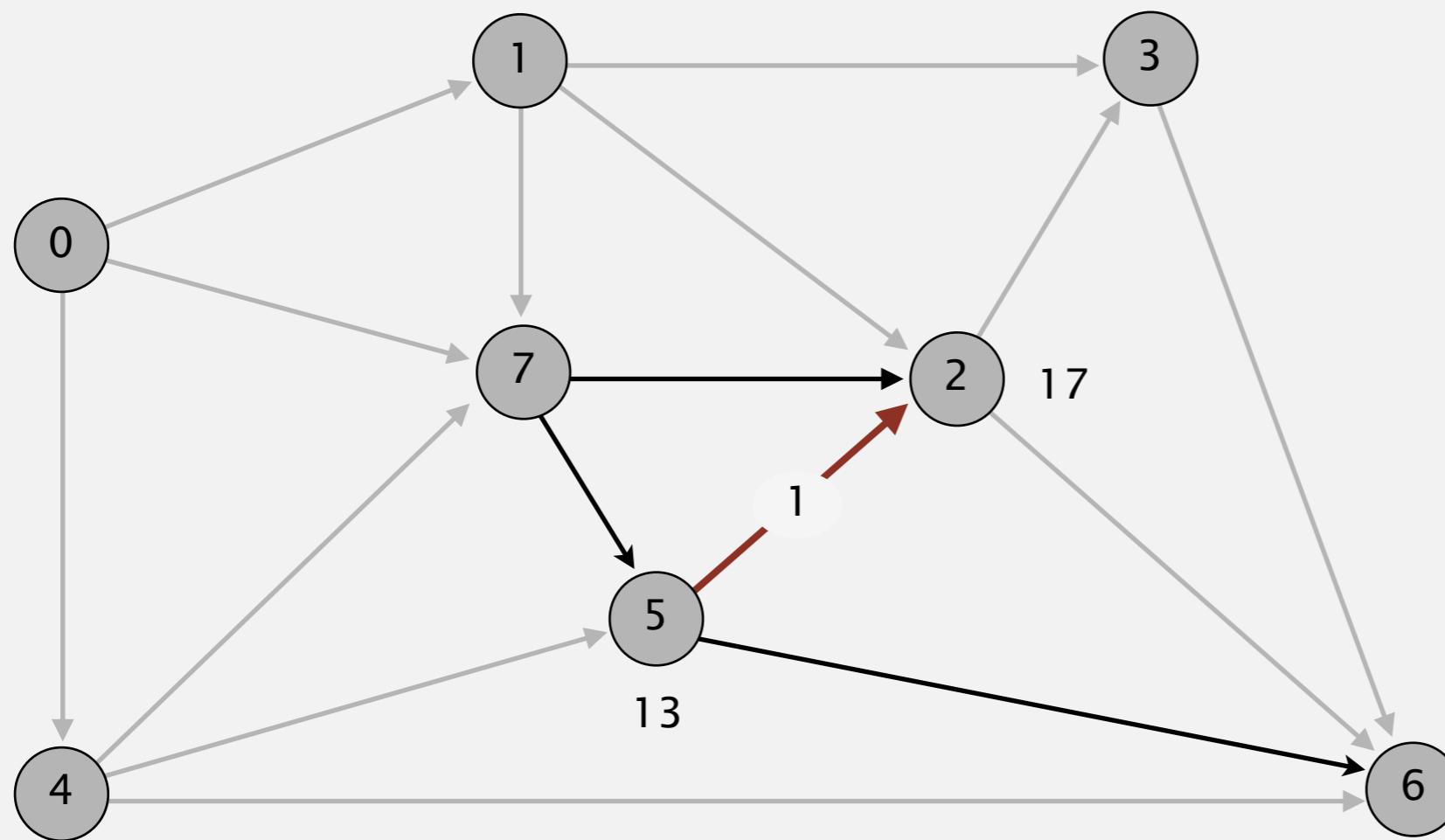
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



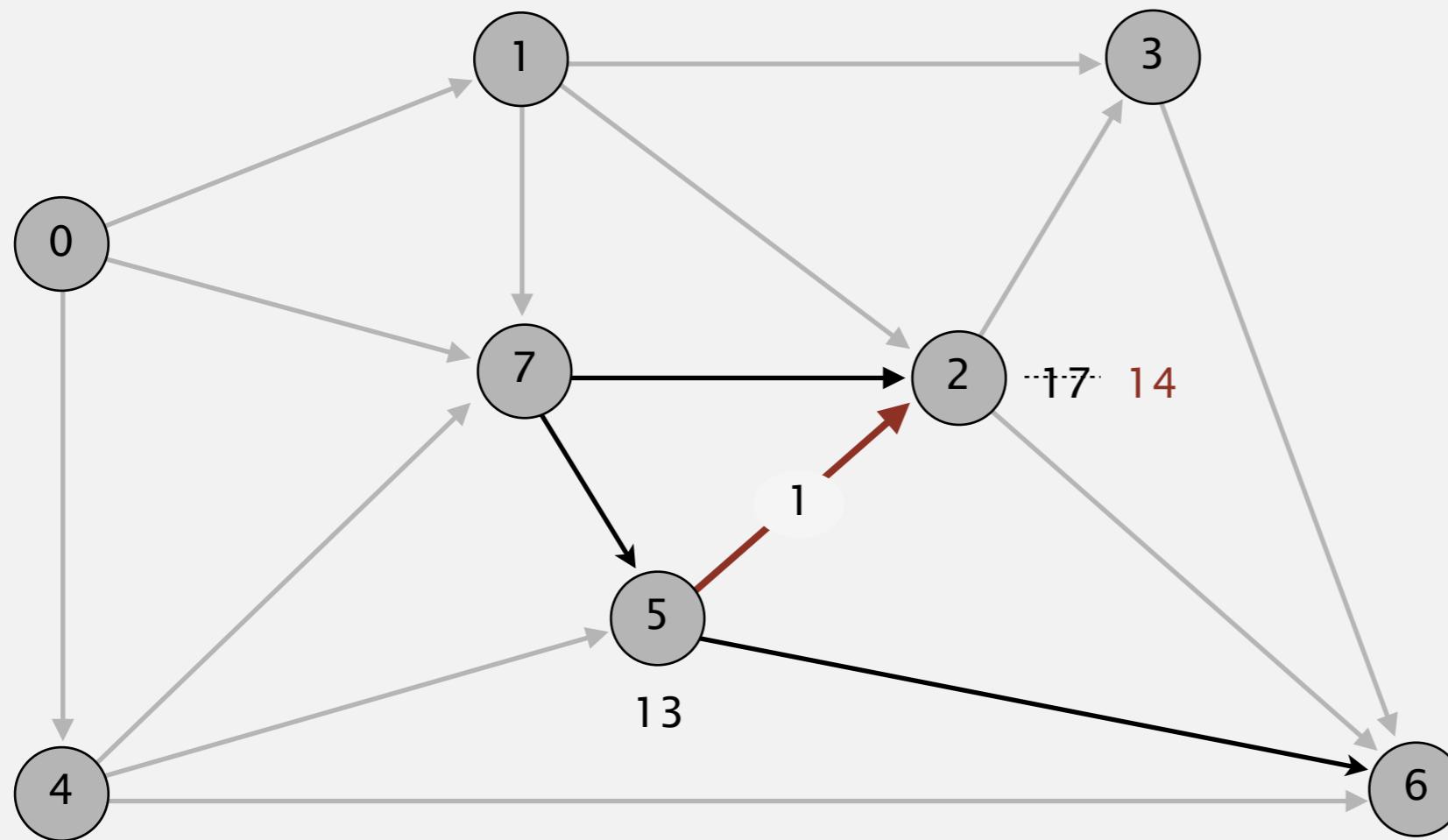
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	17.0	1→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



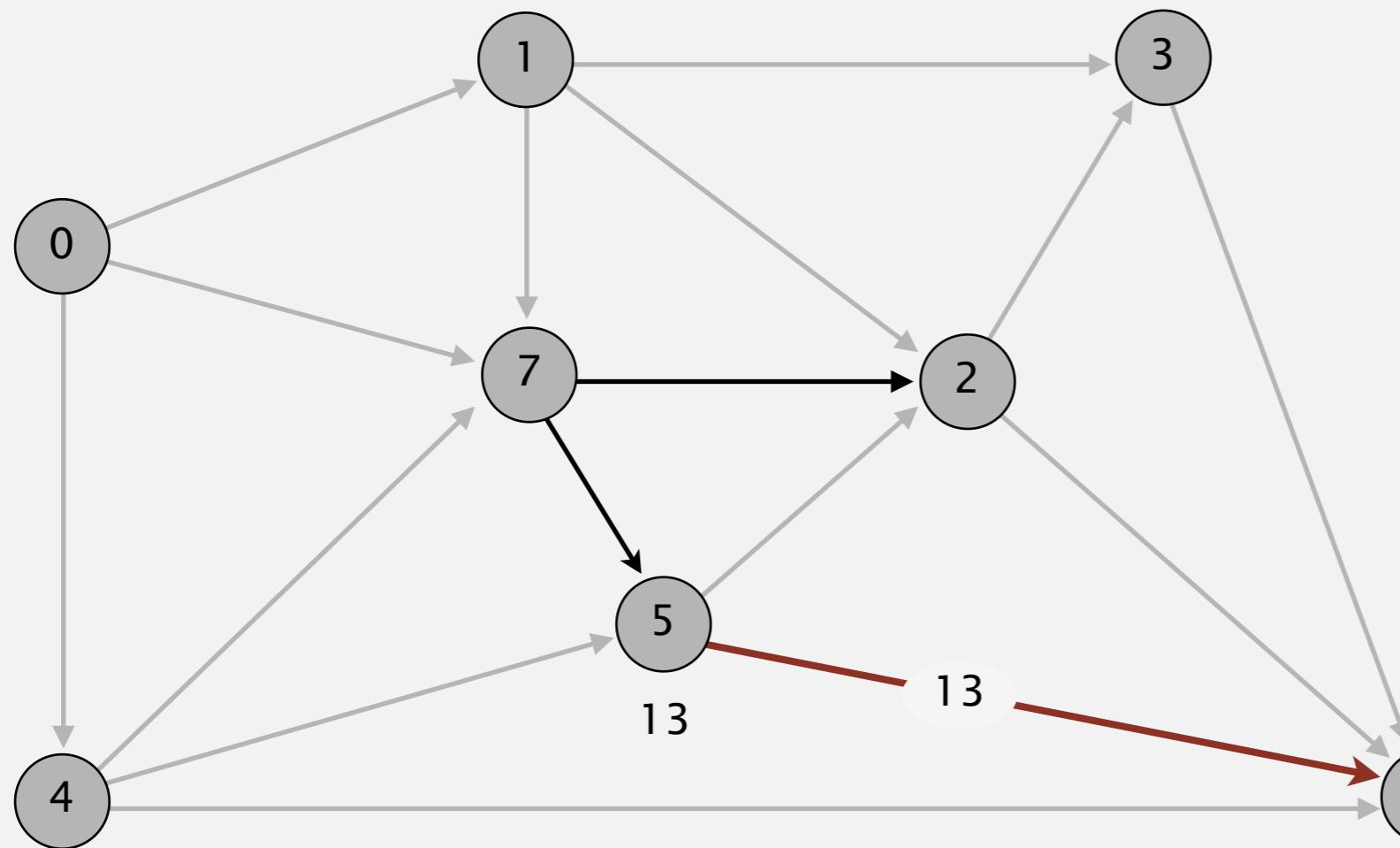
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 0

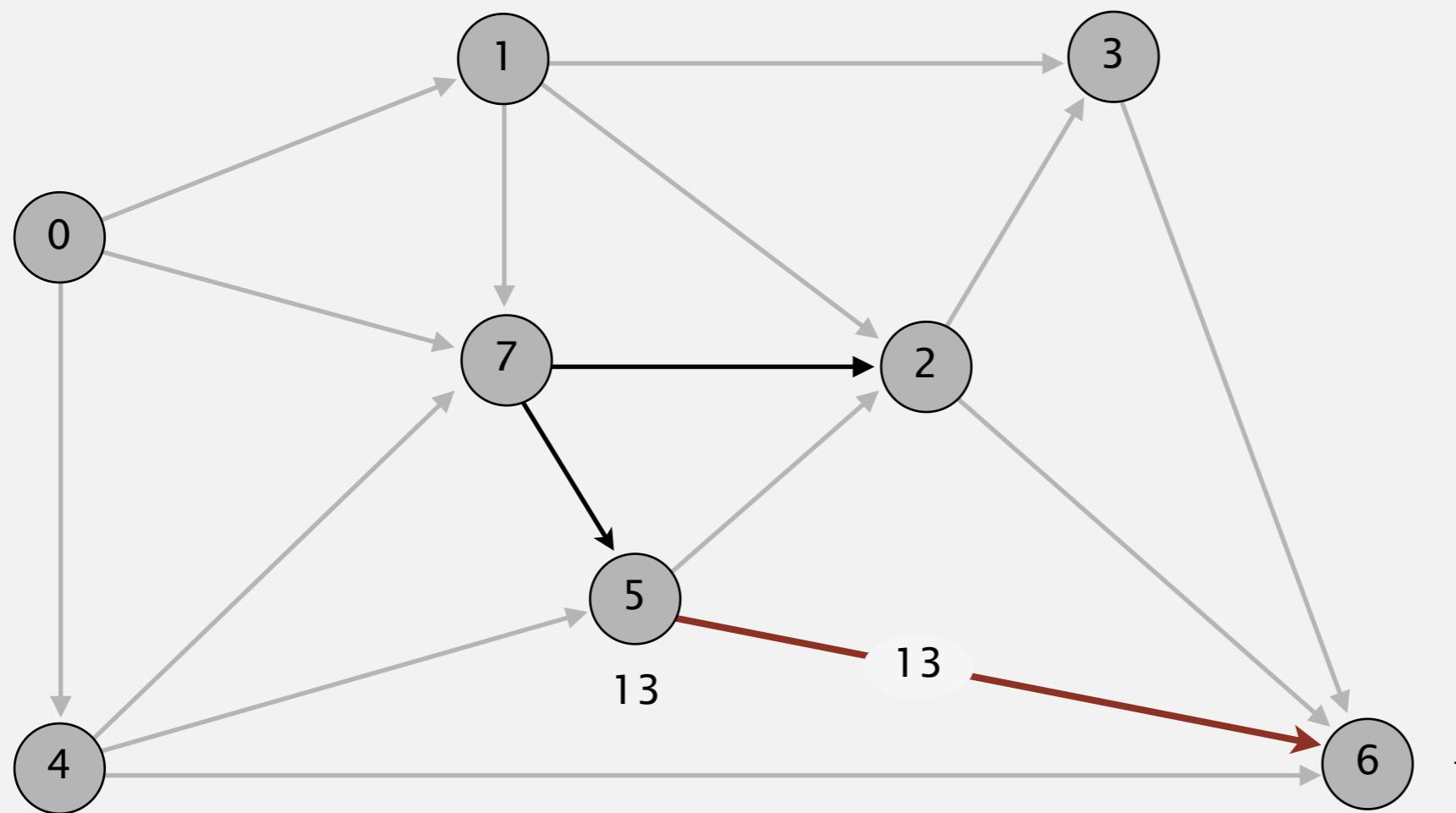
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	28.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

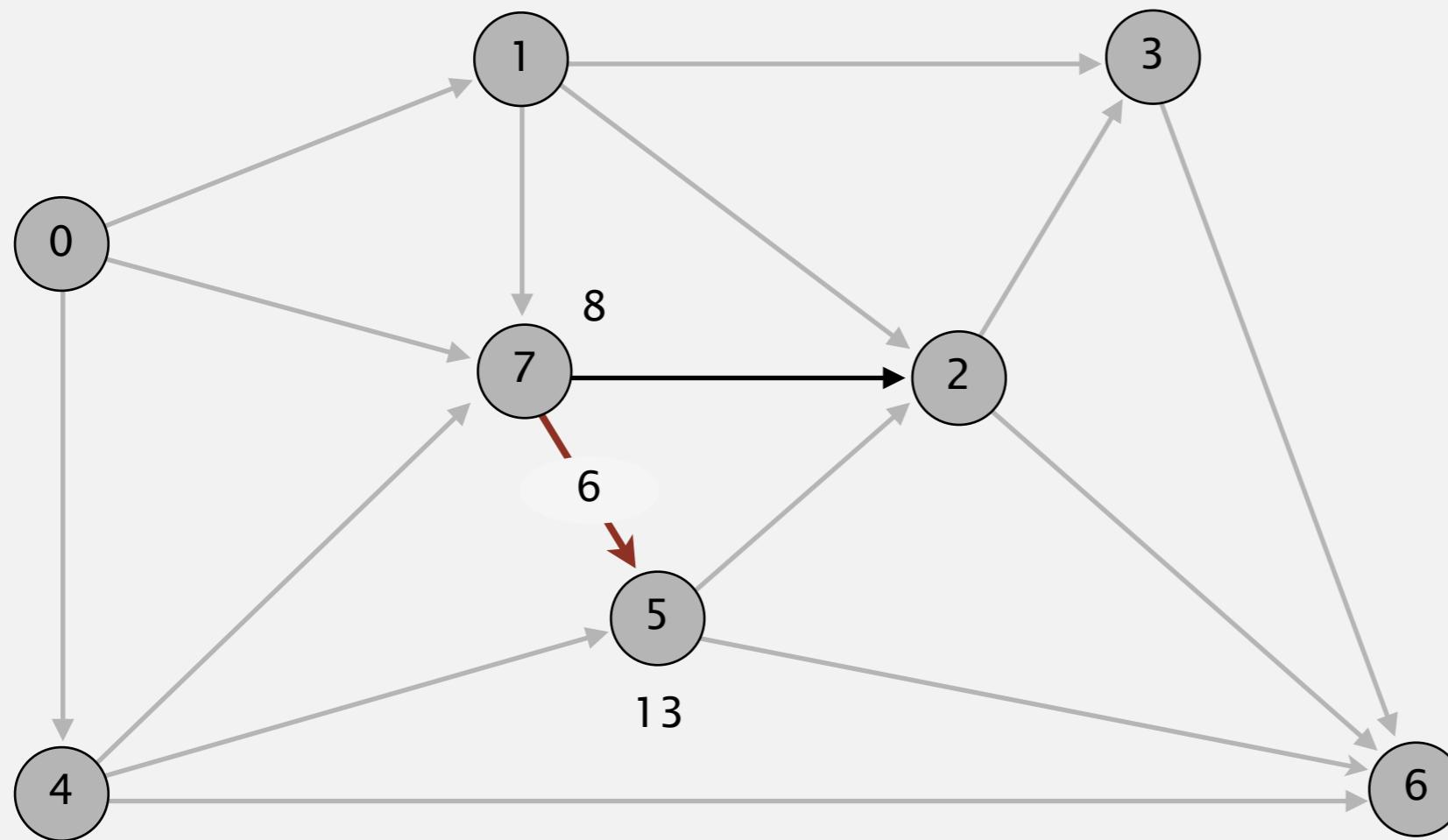
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

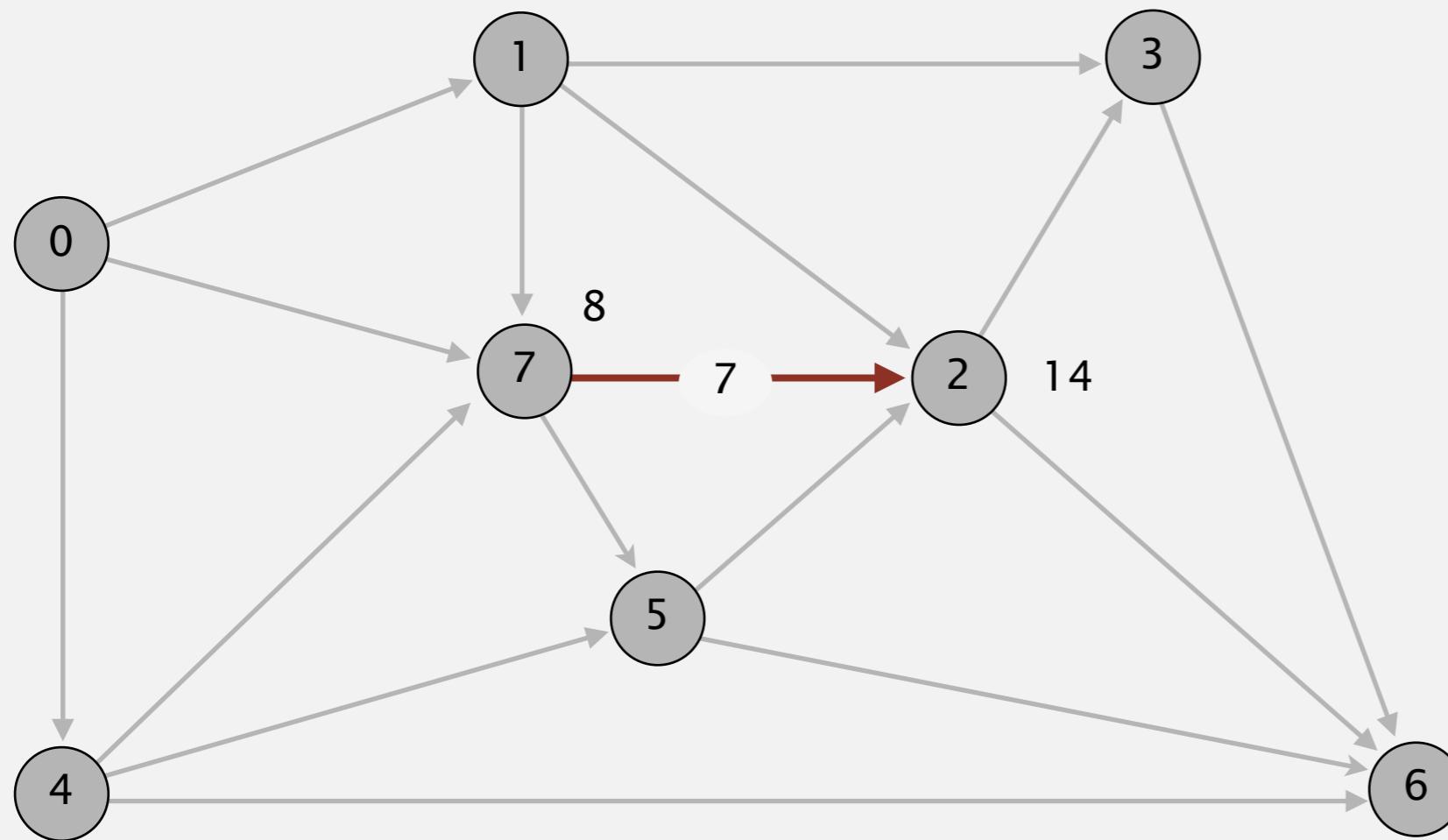
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

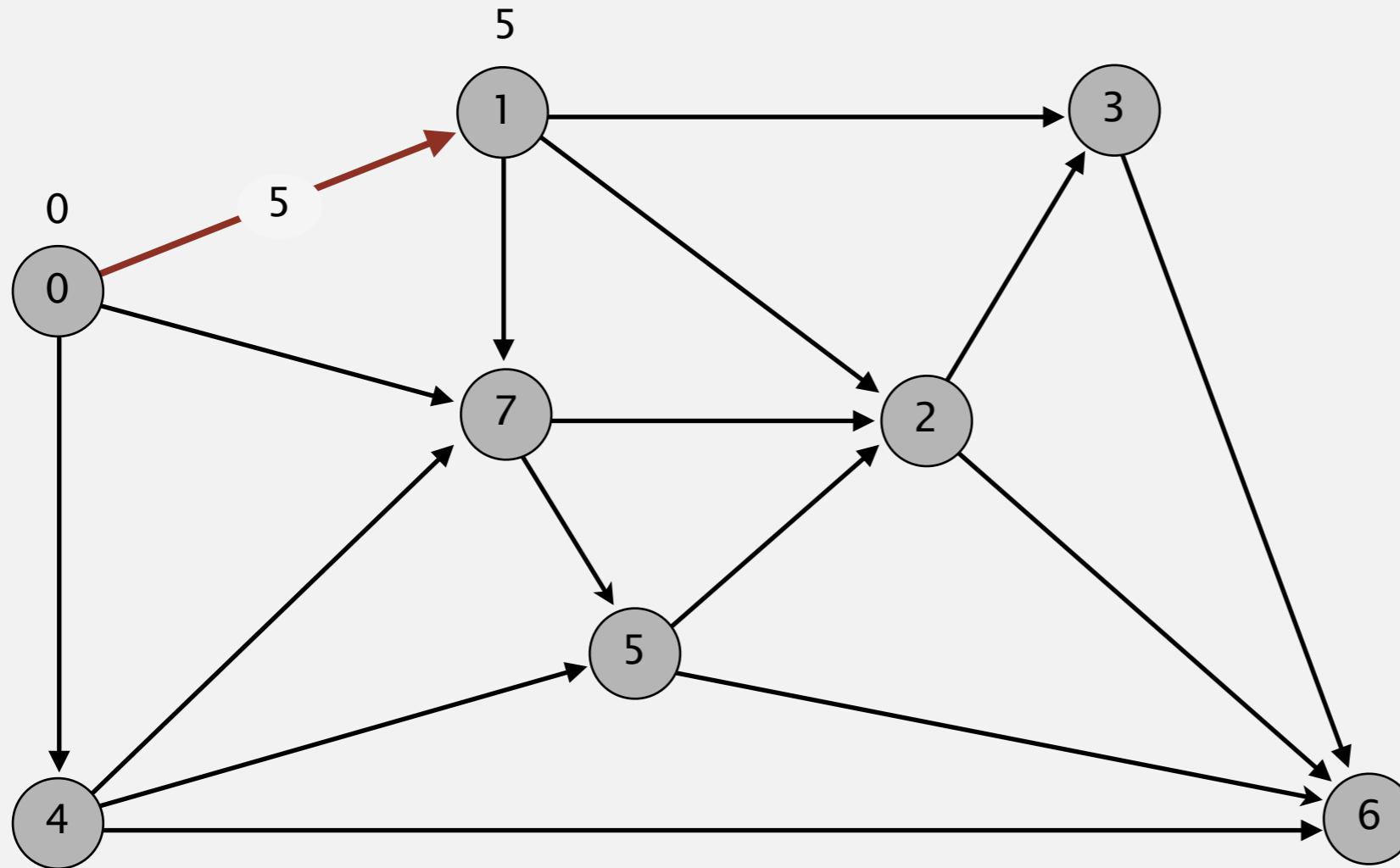
pass 0

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



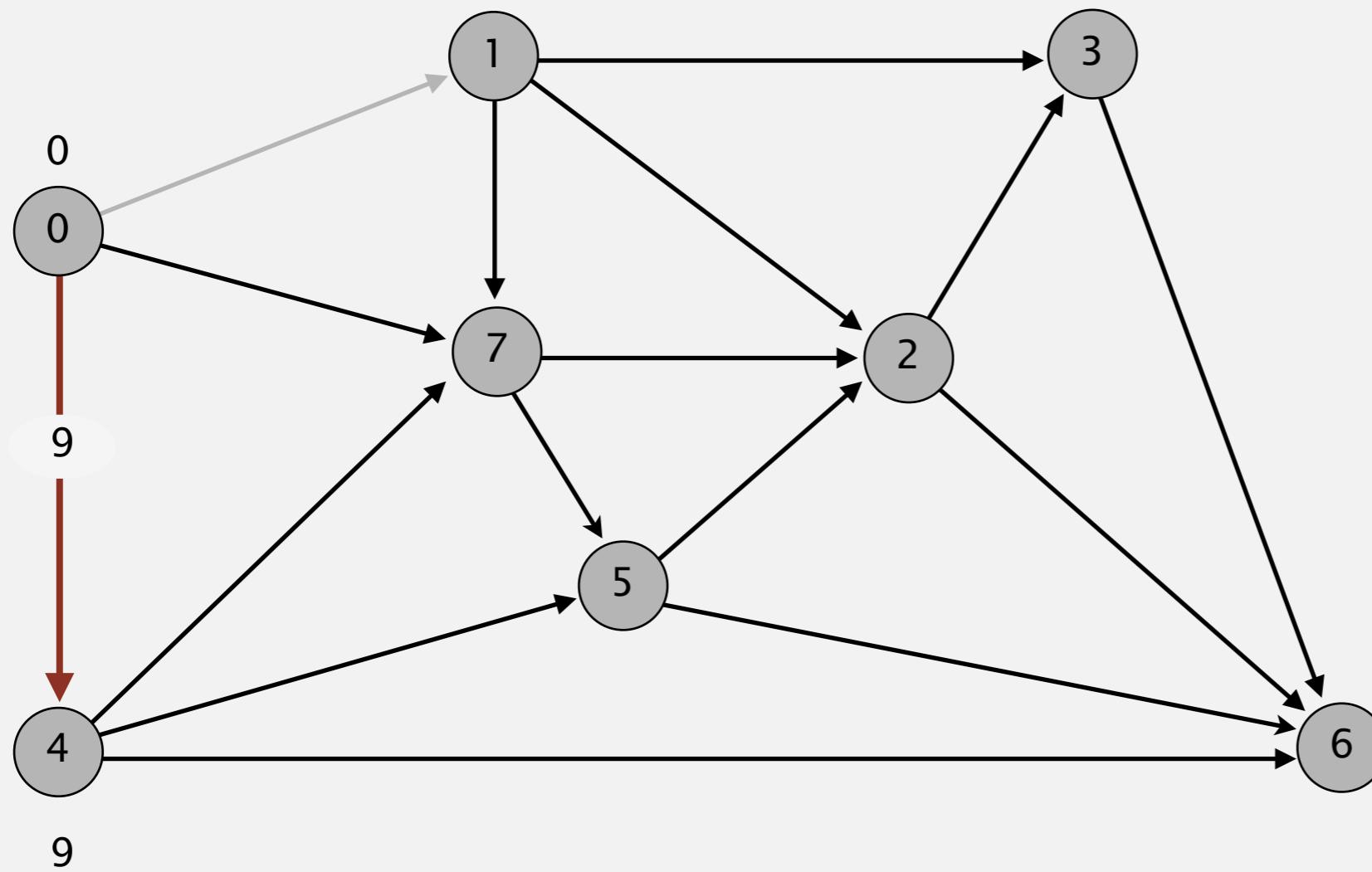
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
↑

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 1

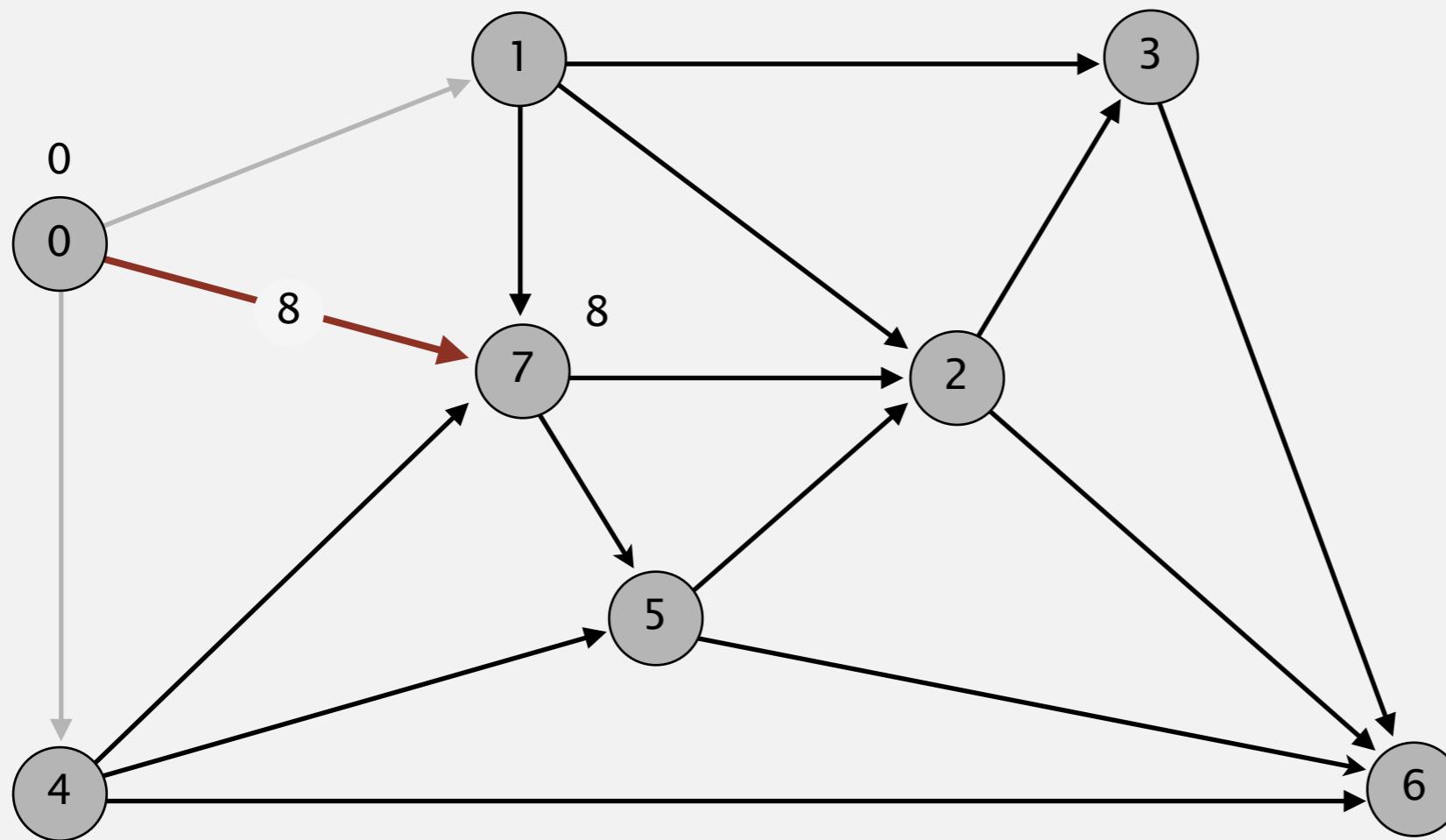
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



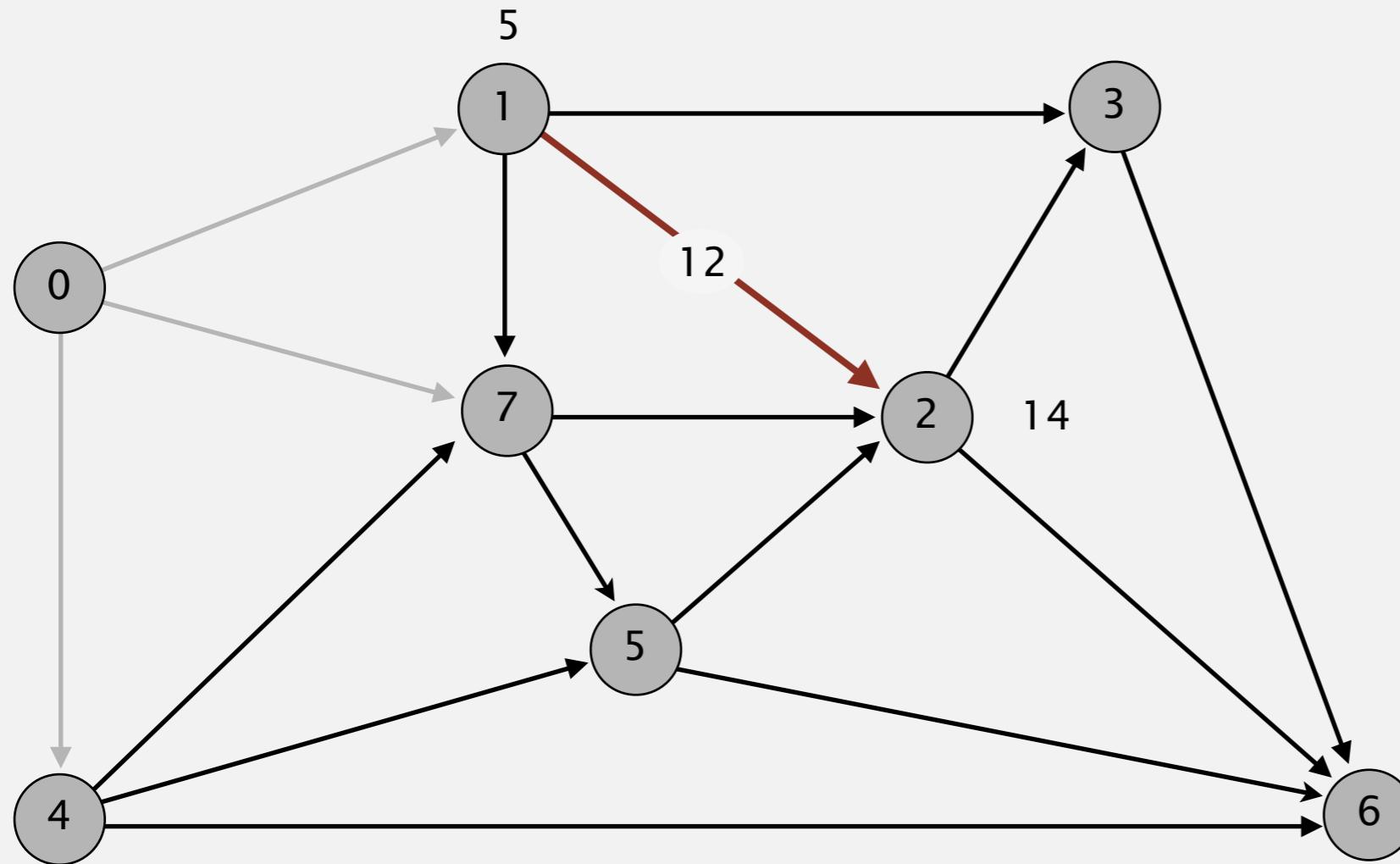
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 1

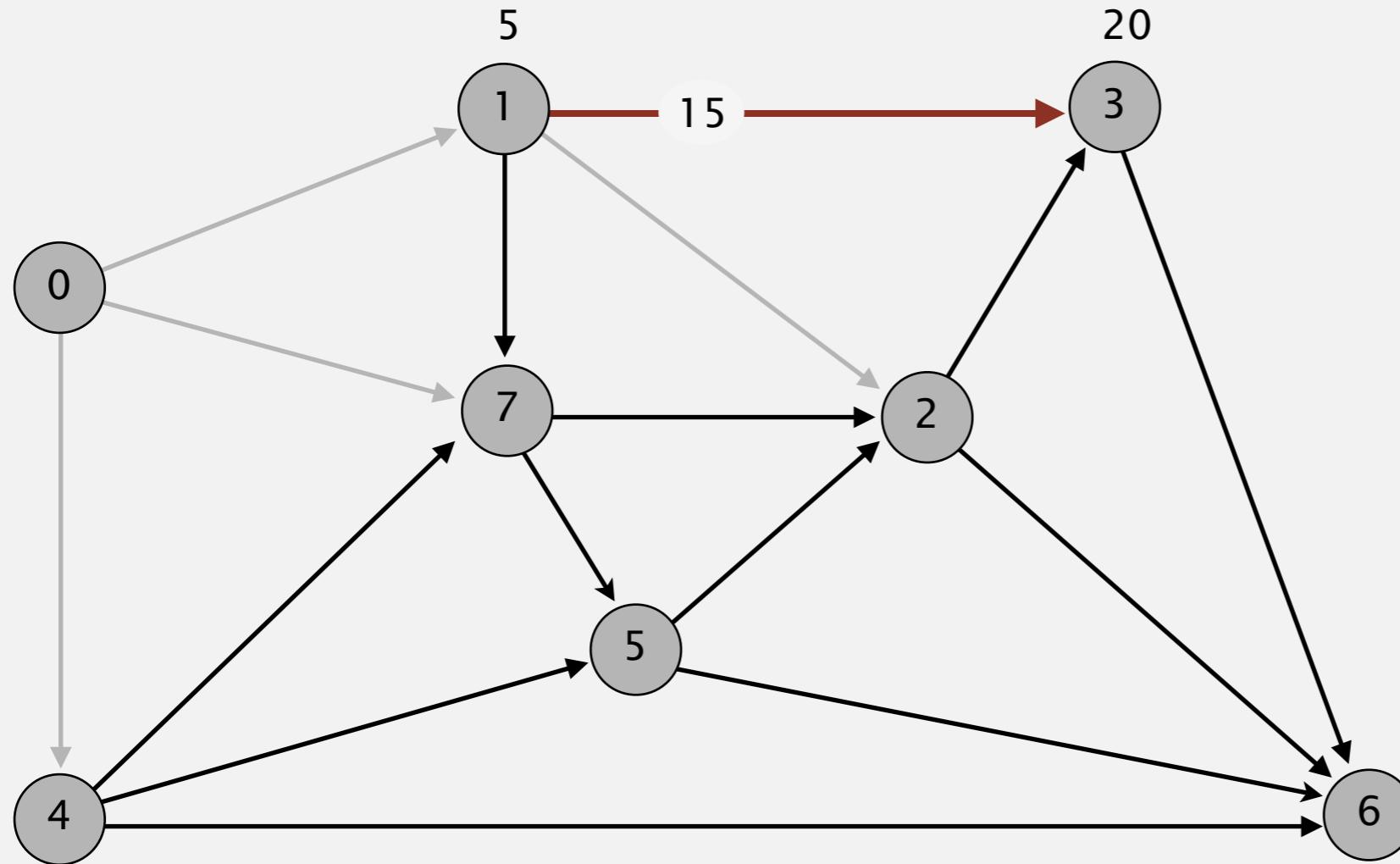
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

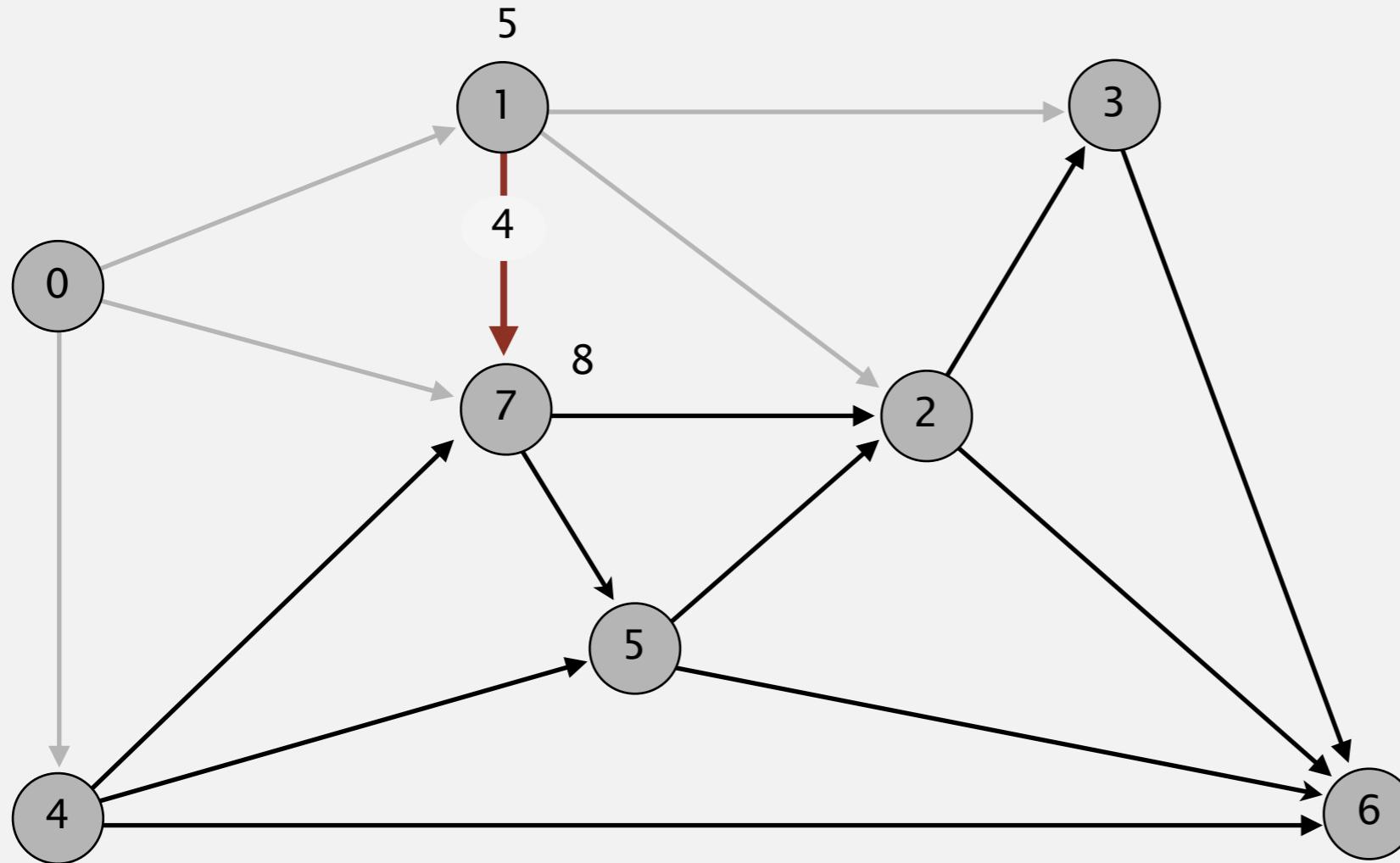
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 1

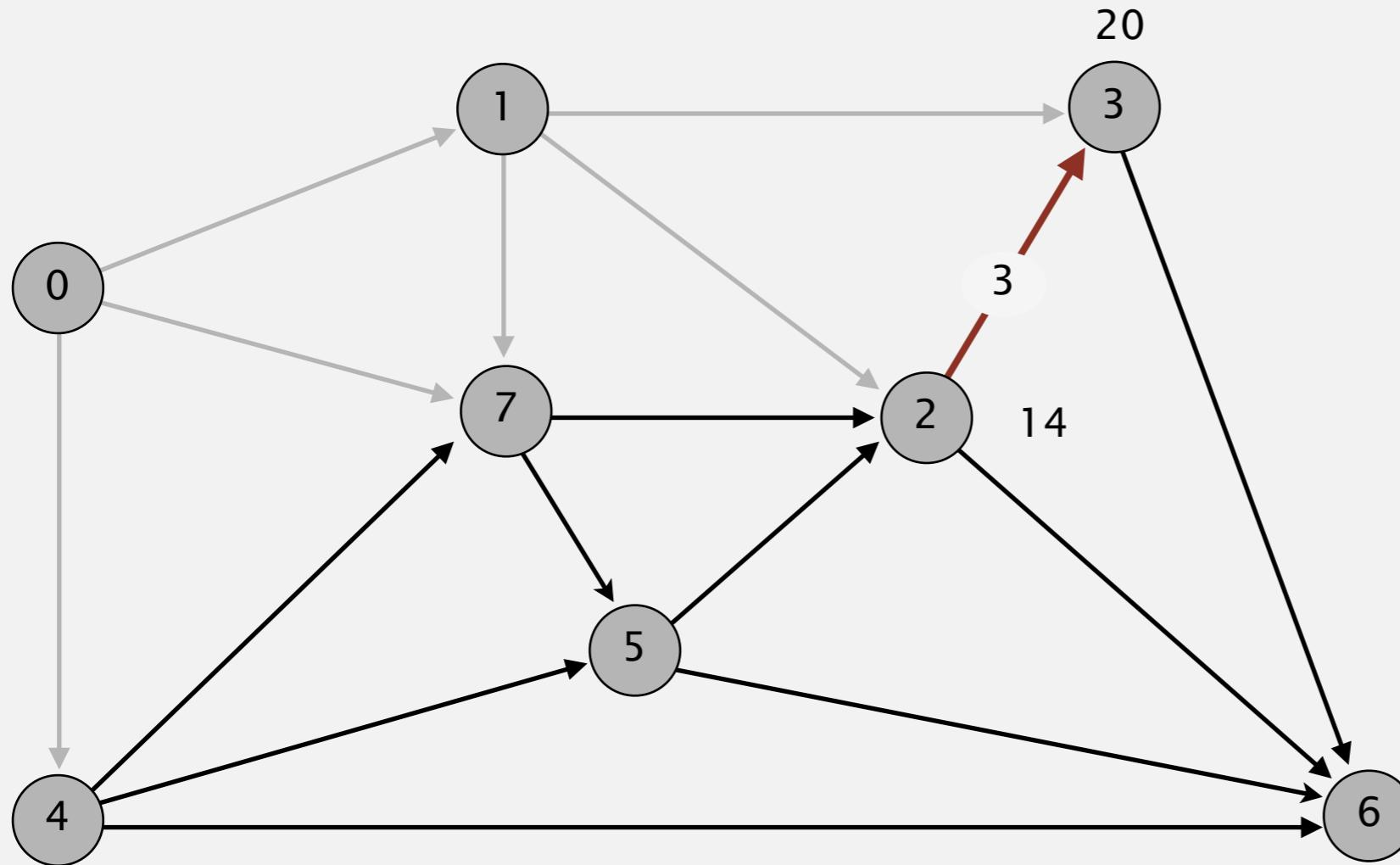
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	20.0	1→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

pass 1

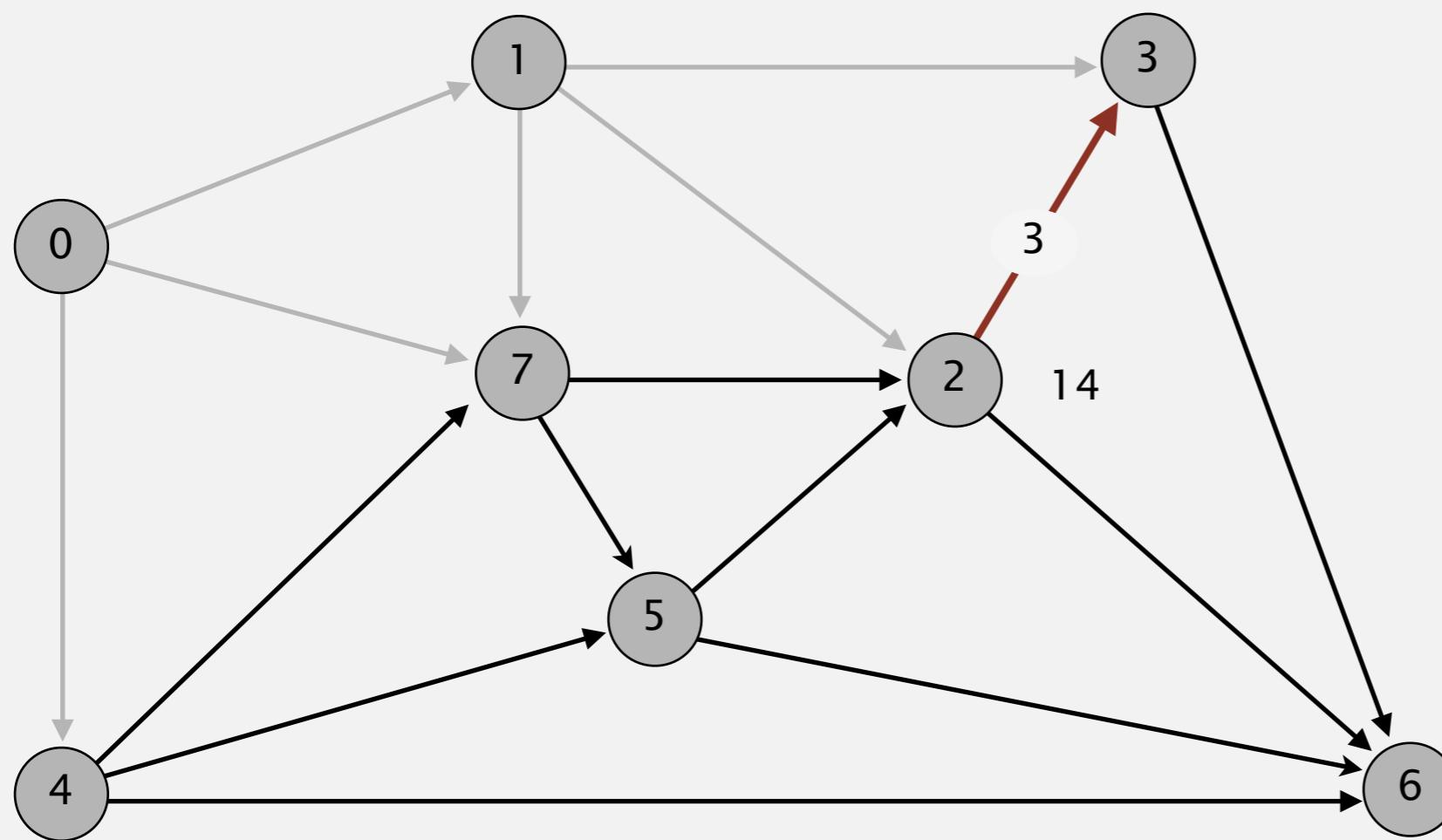
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

2-3 successfully relaxed
in pass 1, but not pass 0



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

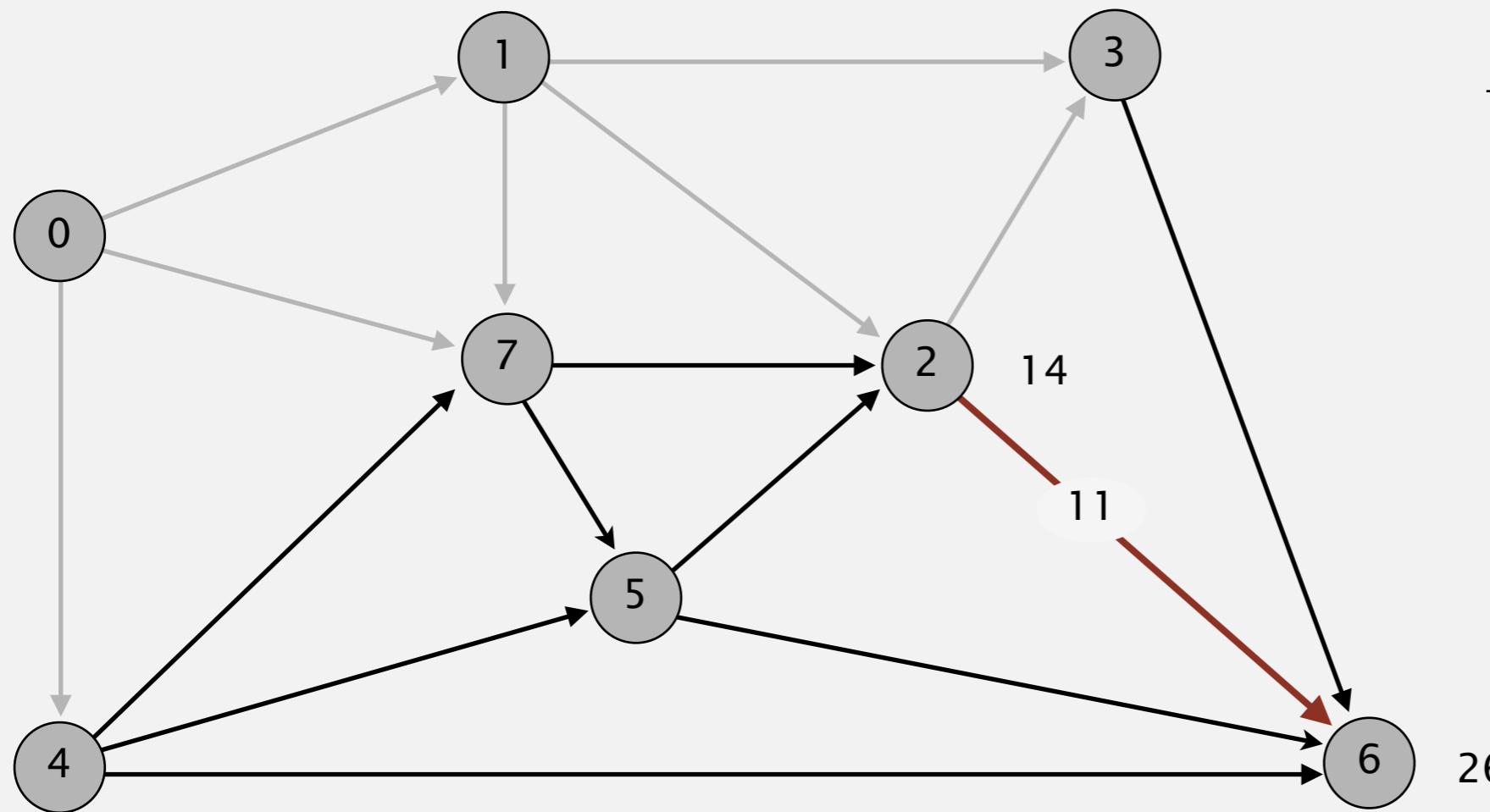
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	26.0	5→6
7	8.0	0→7

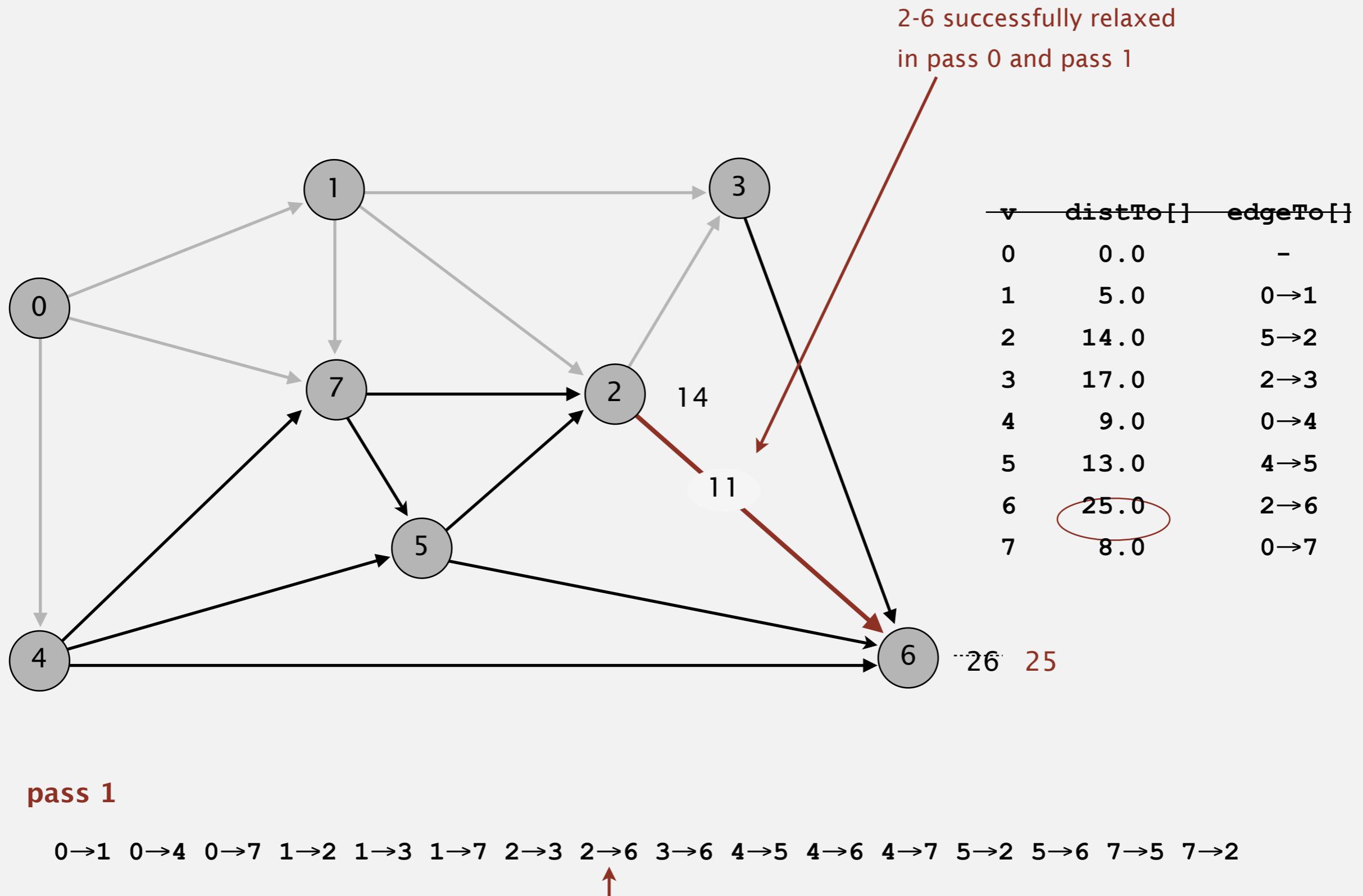
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



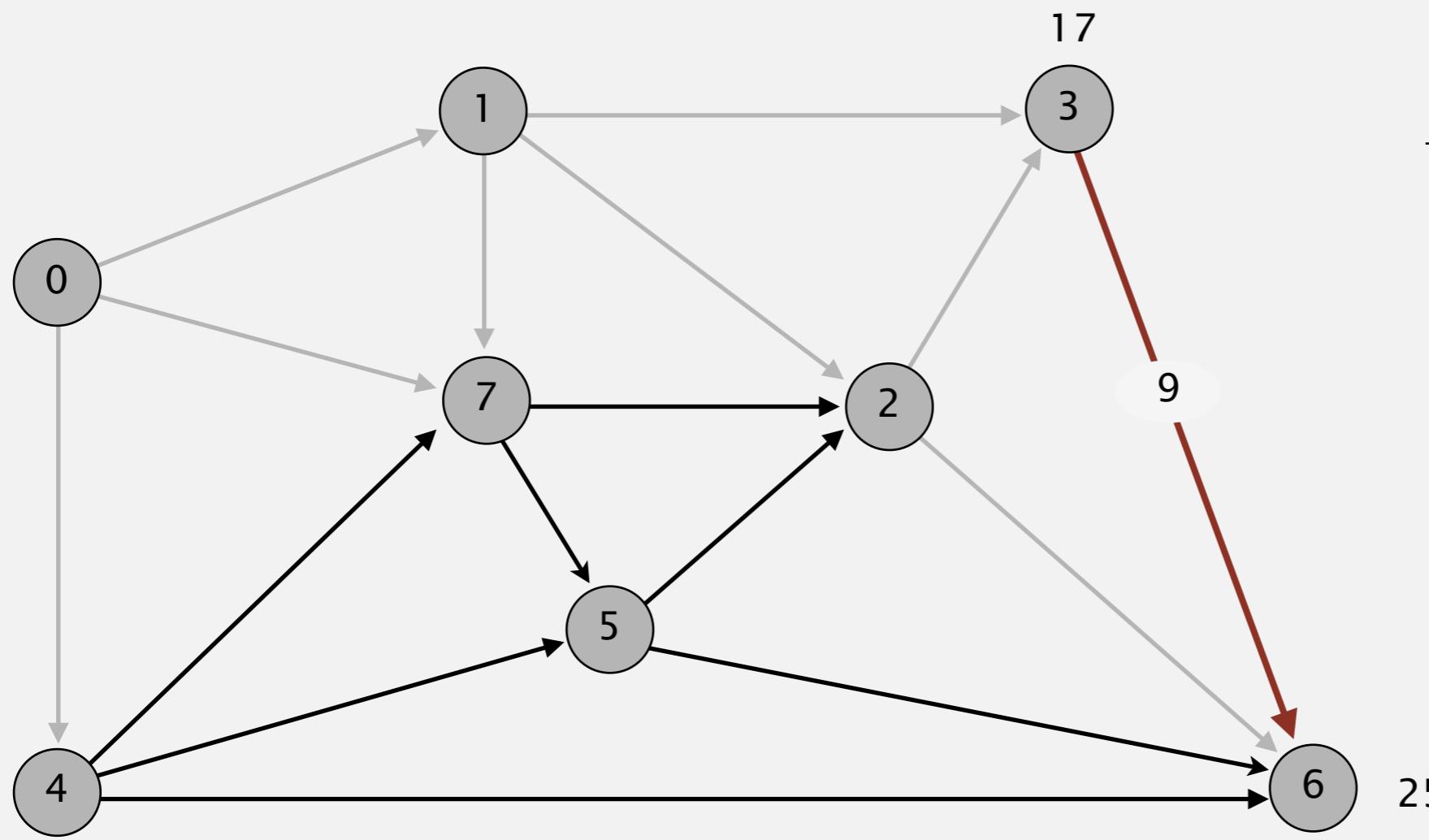
Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



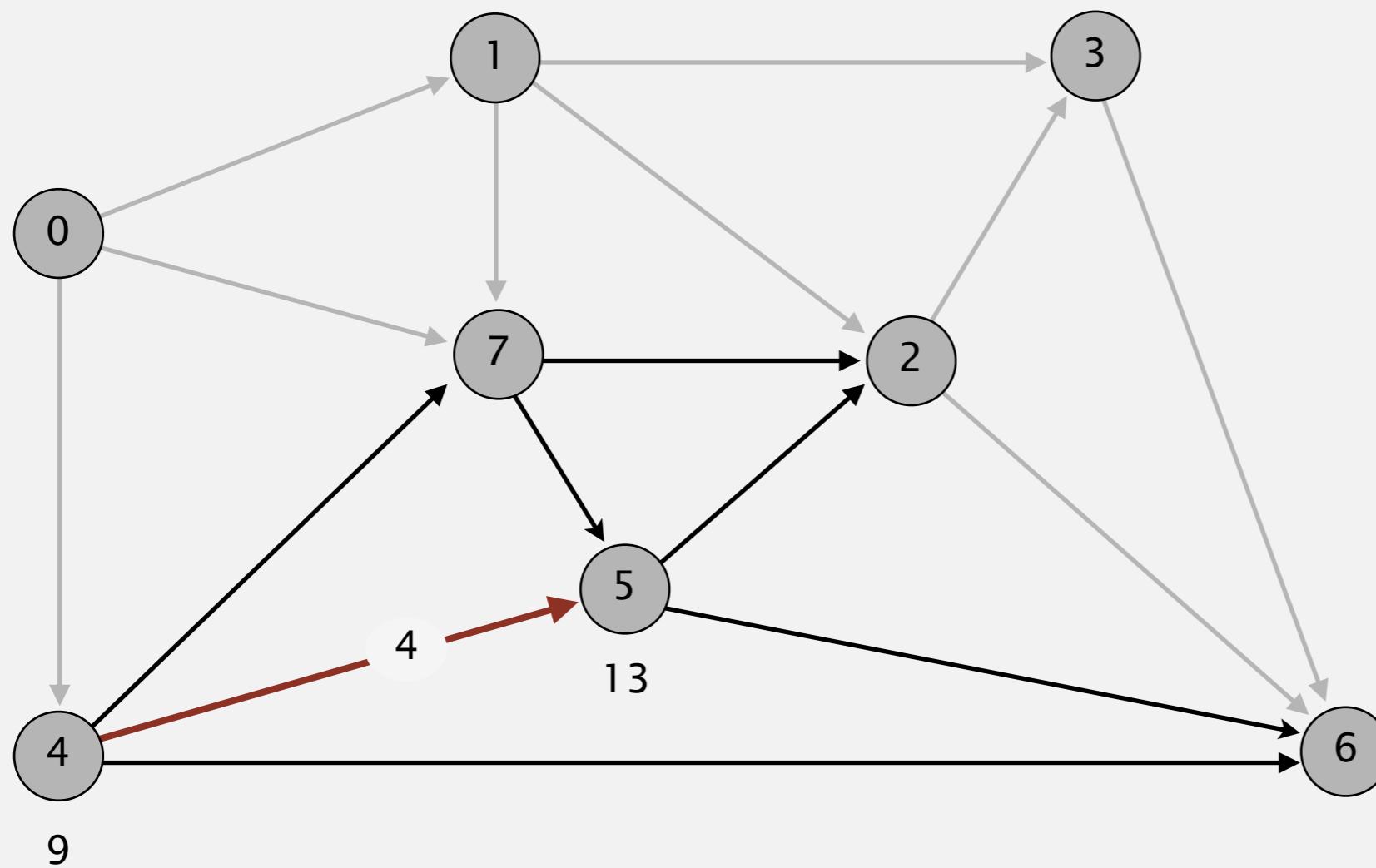
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 1

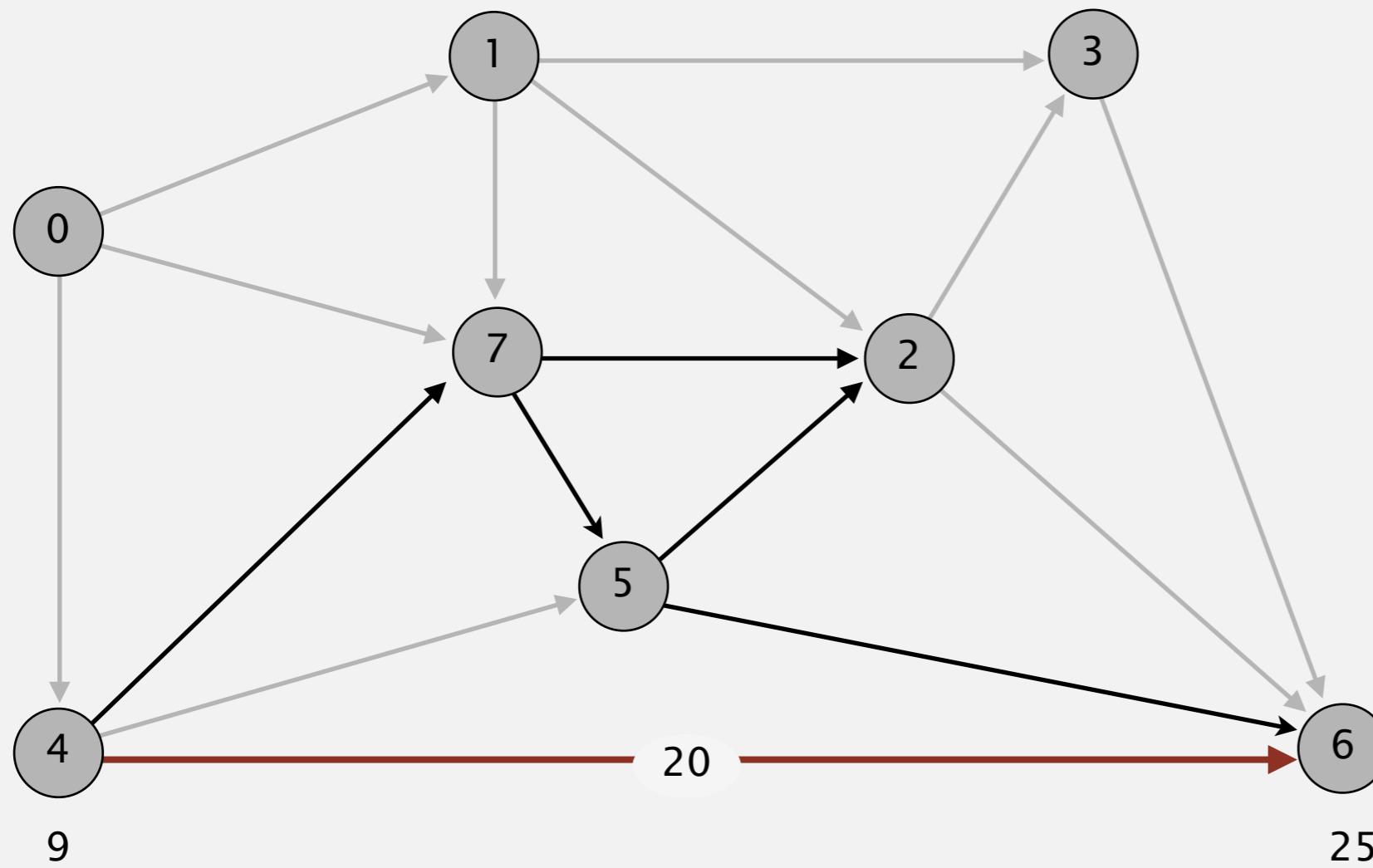
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



pass 1

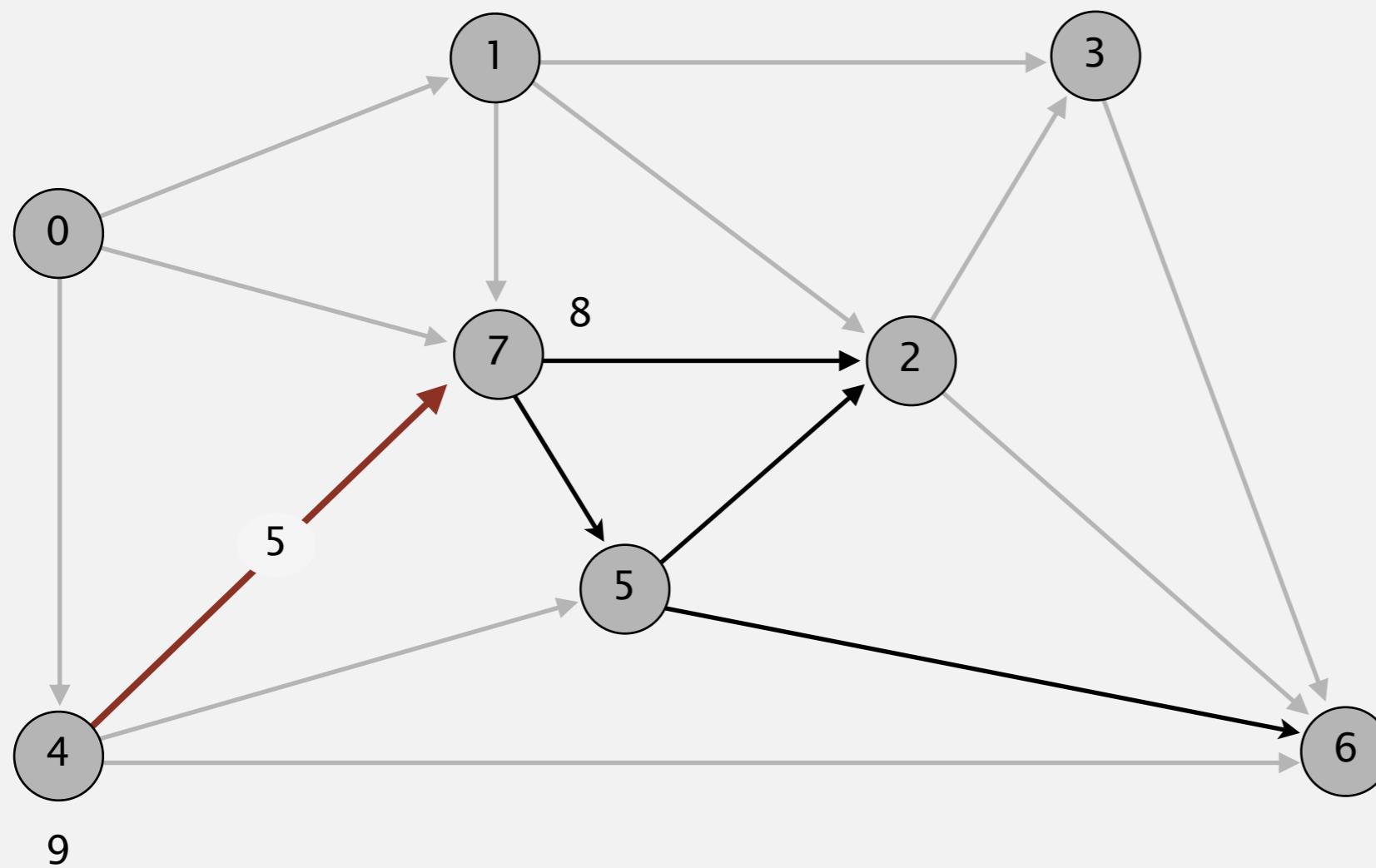
0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



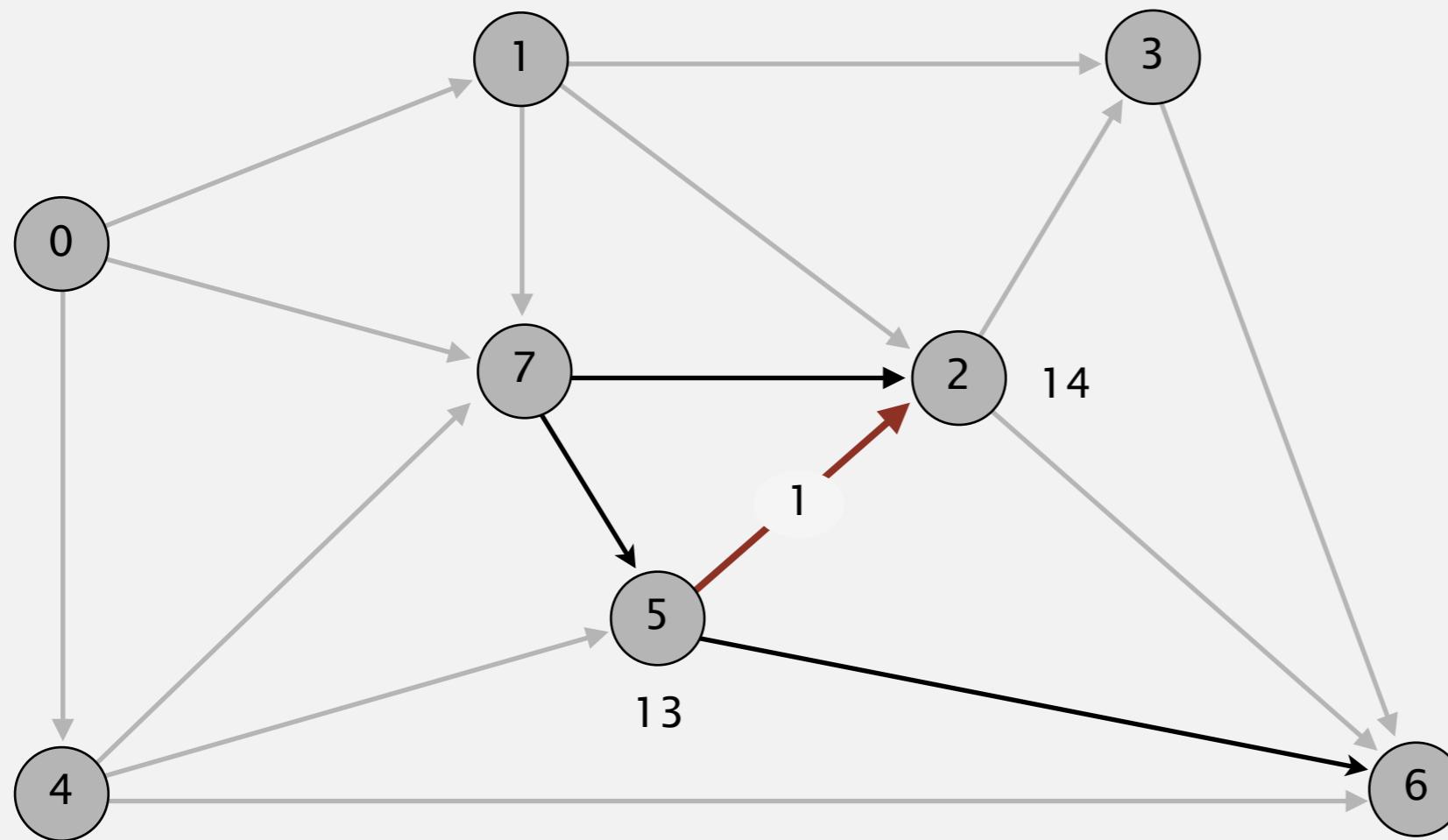
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



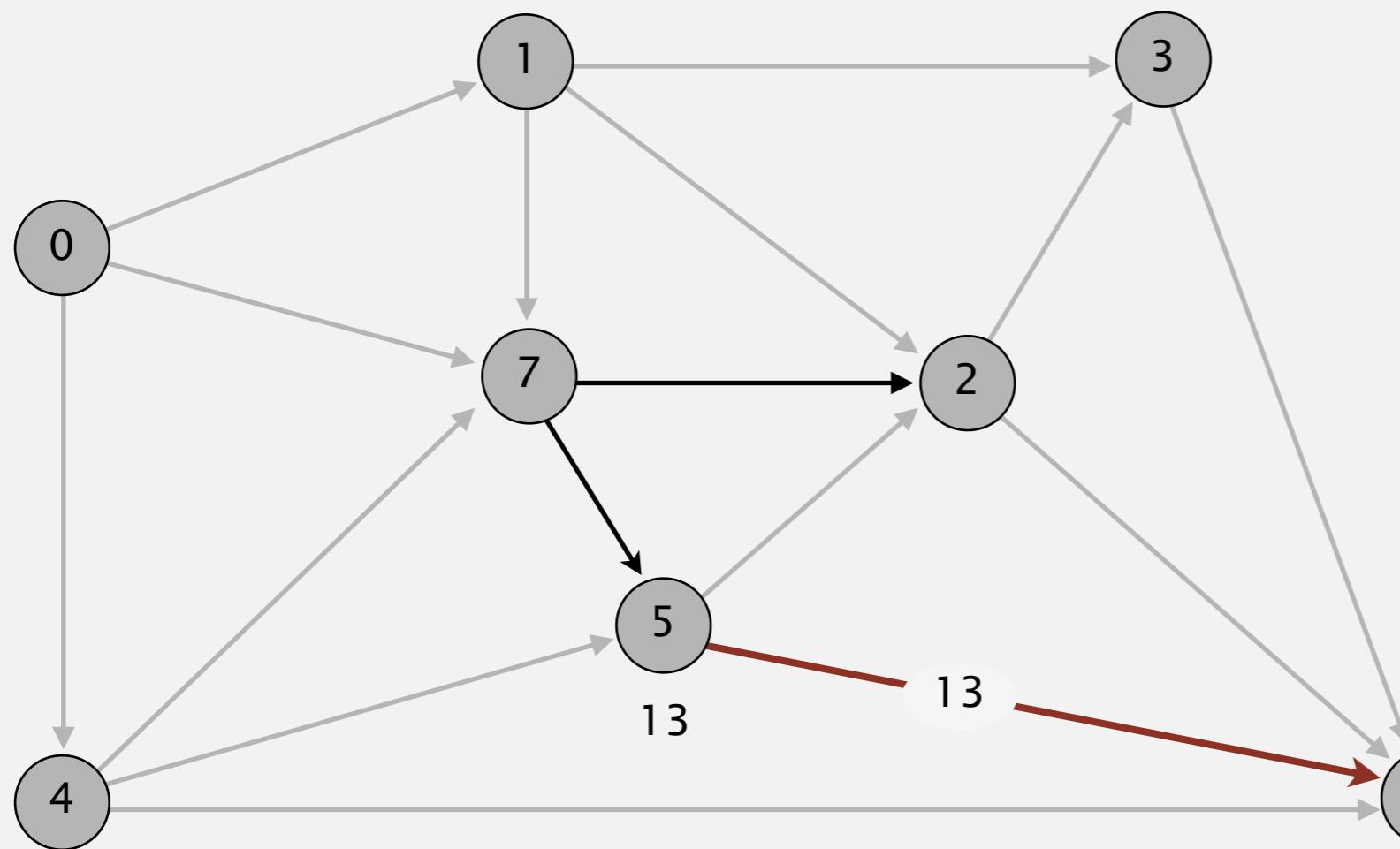
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

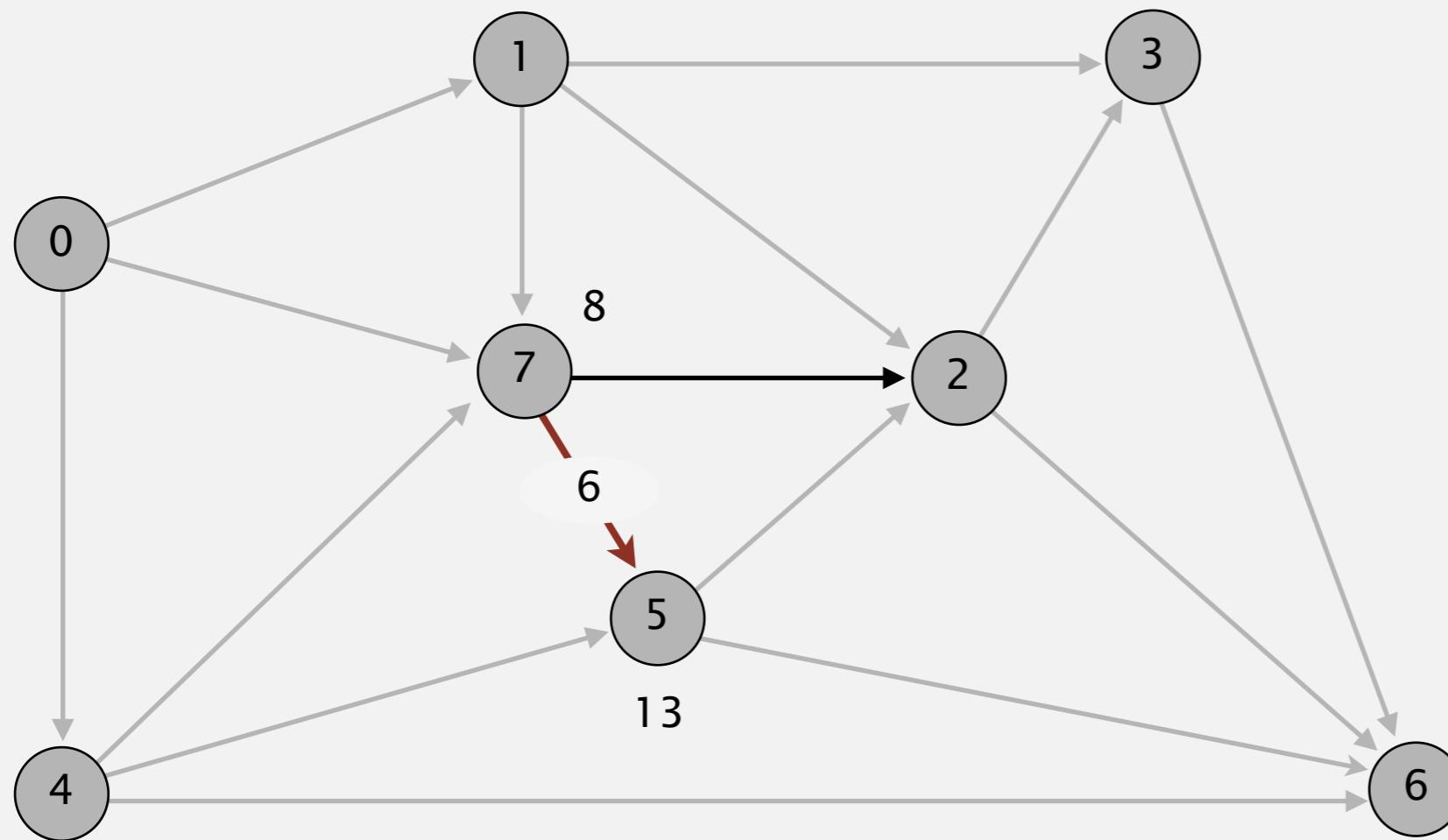
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

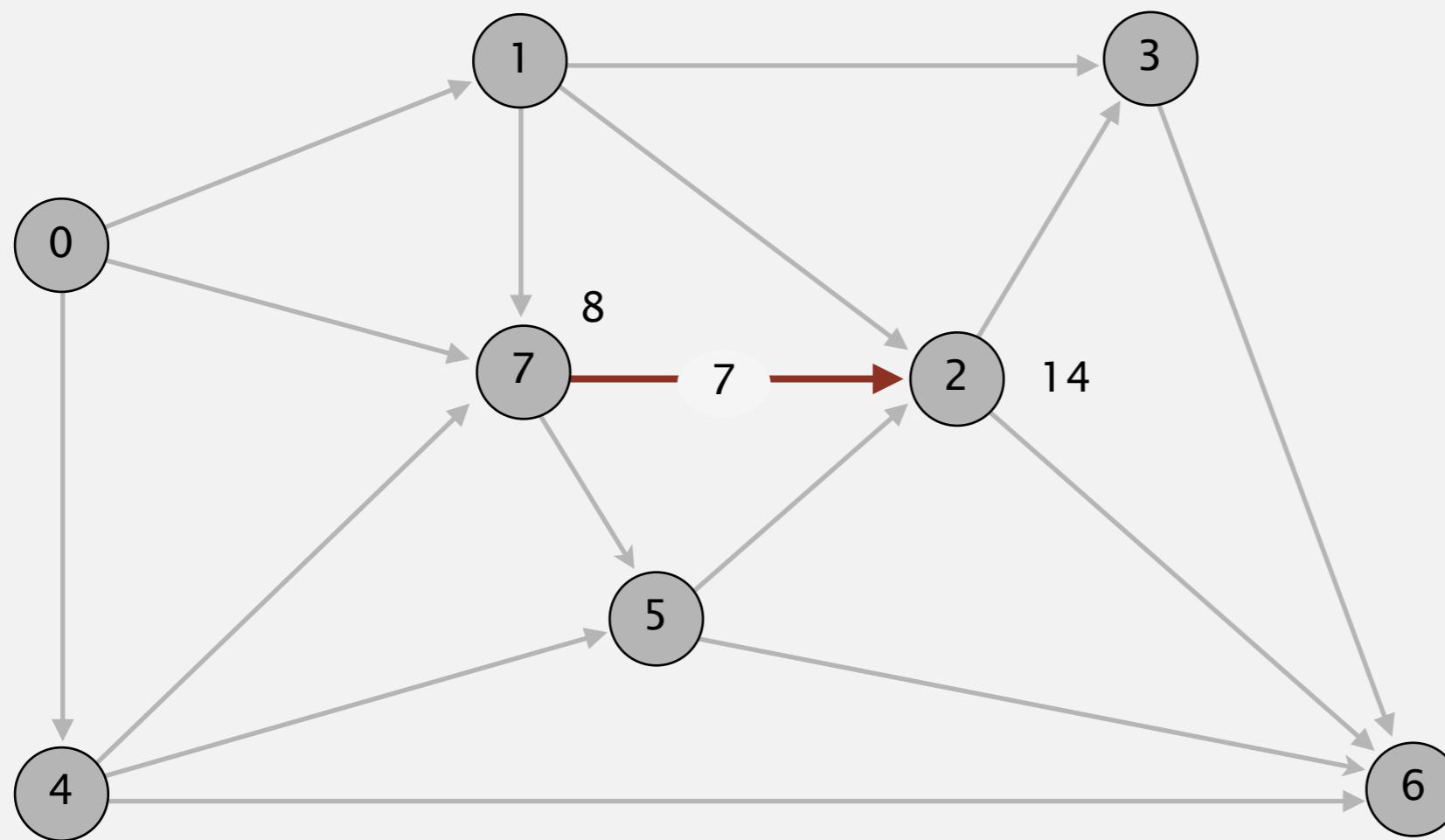
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

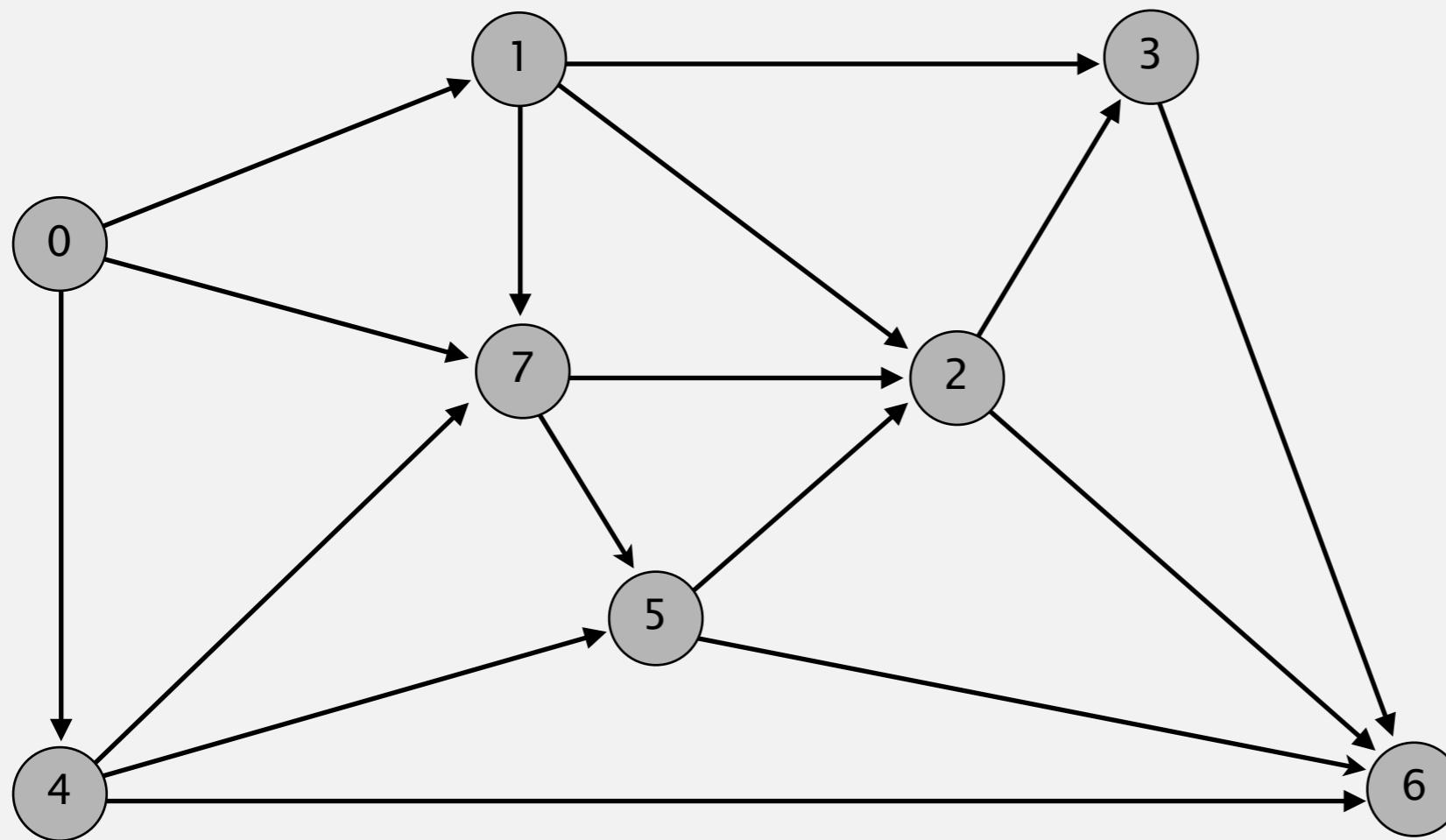
pass 1

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2



Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



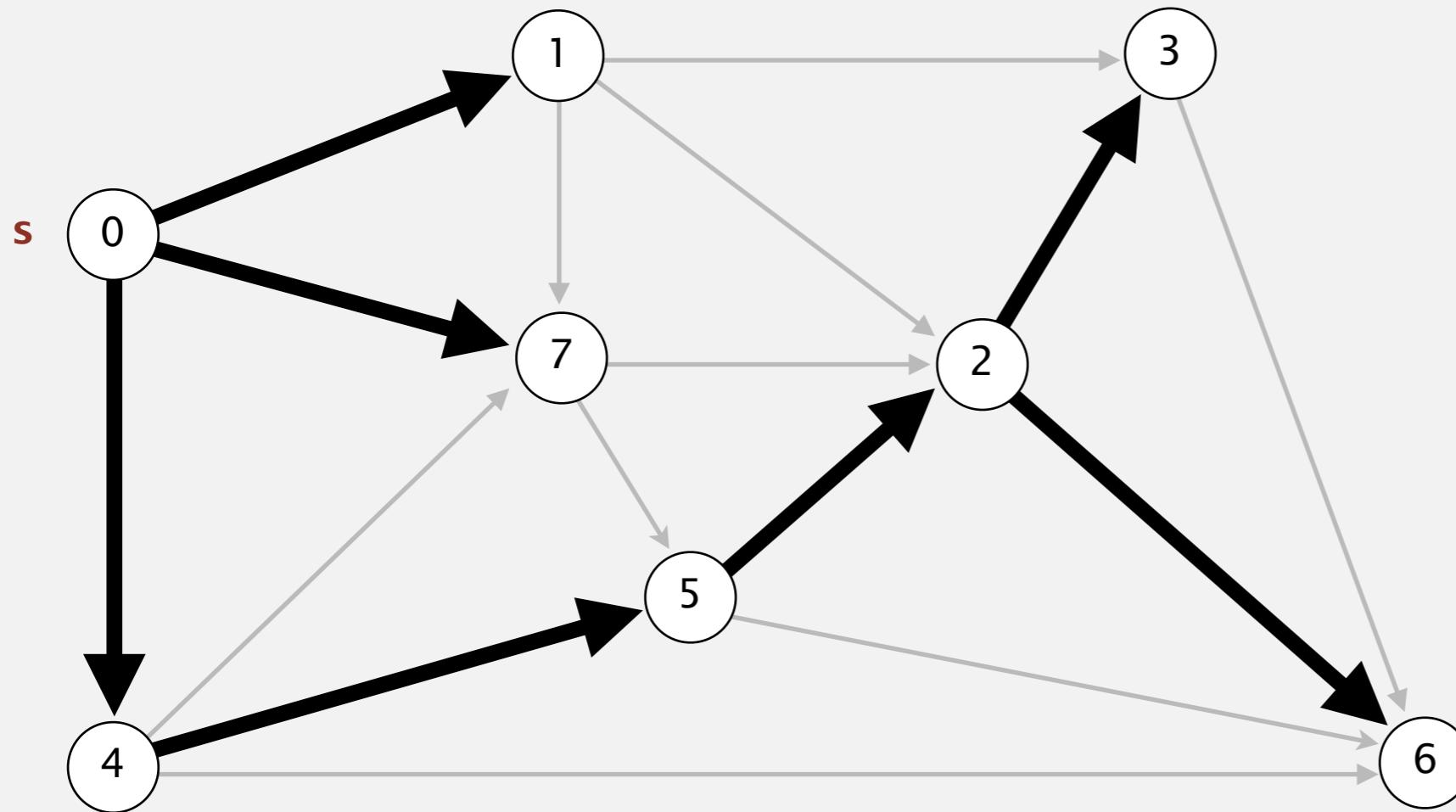
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

pass 2, 3, 4, ... (no further changes)

0→1 0→4 0→7 1→2 1→3 1→7 2→3 2→6 3→6 4→5 4→6 4→7 5→2 5→6 7→5 7→2
↑

Bellman-Ford algorithm demo

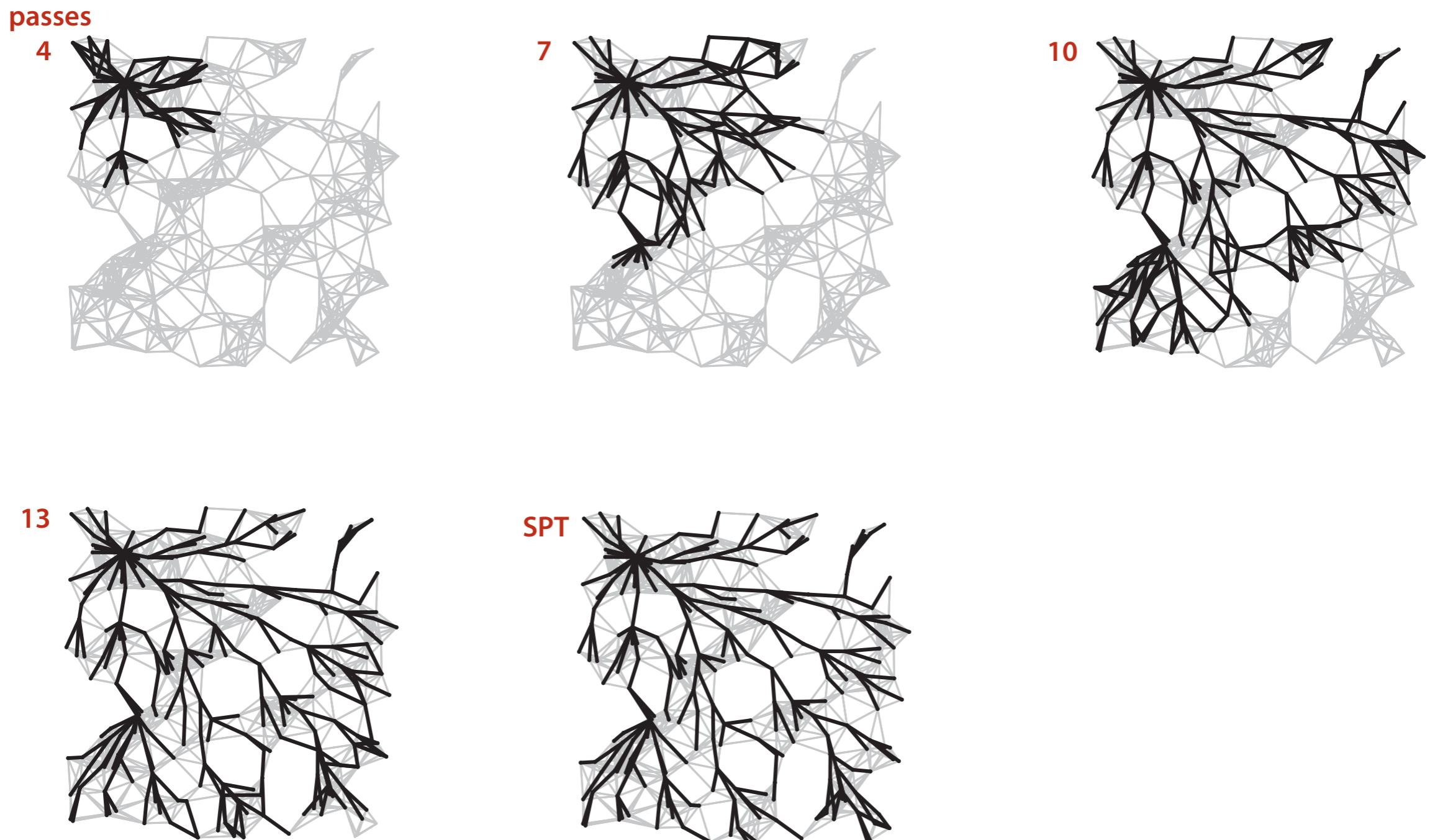
Repeat V times: relax all E edges.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

Bellman-Ford algorithm visualization



Bellman-Ford algorithm: analysis

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

- Relax each edge.
-

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After pass i , found shortest path containing at most i edges.

Bellman-Ford algorithm: practical improvement

Observation. If `distTo[v]` does not change during pass i , no need to relax any edge pointing from v in pass $i + 1$.

FIFO implementation. Maintain queue of vertices whose `distTo[]` changed.



be careful to keep at most one copy
of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Bellman-Ford algorithm: Java implementation

```
public class BellmanFordSP
{
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private boolean[] onQ;
    private Queue<Integer> queue;

    public BellmanFordSPT(EdgeWeightedDigraph G, int s)
    {
        distTo = new double[G.V()];
        edgeTo = new DirectedEdge[G.V()];
        onQ = new boolean[G.V()];
        queue = new Queue<Integer>();

        for (int v = 0; v < V; v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        queue.enqueue(s);
        while (!queue.isEmpty())
        {
            int v = queue.dequeue();
            onQ[v] = false;
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (!onQ[w])
        {
            queue.enqueue(w);
            onQ[w] = true;
        }
    }
}
```

queue of vertices whose
distTo[] value changes

Single source shortest-paths implementation: cost summary

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	$E + V$	$E + V$	V
Dijkstra (binary heap)	no negative weights	$E \log V$	$E \log V$	V
Bellman-Ford	no negative cycles	$E V$	$E V$	V
Bellman-Ford (queue-based)		$E + V$	$E V$	V

Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

Negative cycle. Add two method to the API for SP.

`boolean hasNegativeCycle()`

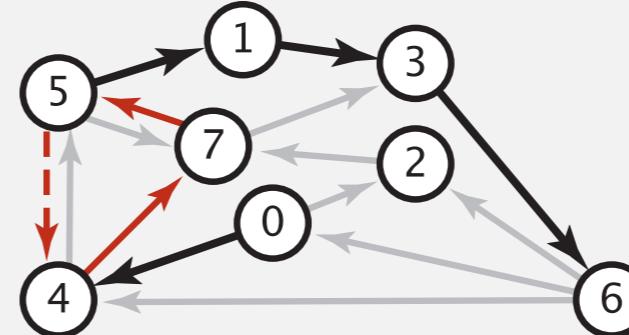
is there a negative cycle?

`Iterable <DirectedEdge> negativeCycle()`

negative cycle reachable from s

digraph

```
4->5  0.35
5->4 -0.66
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```

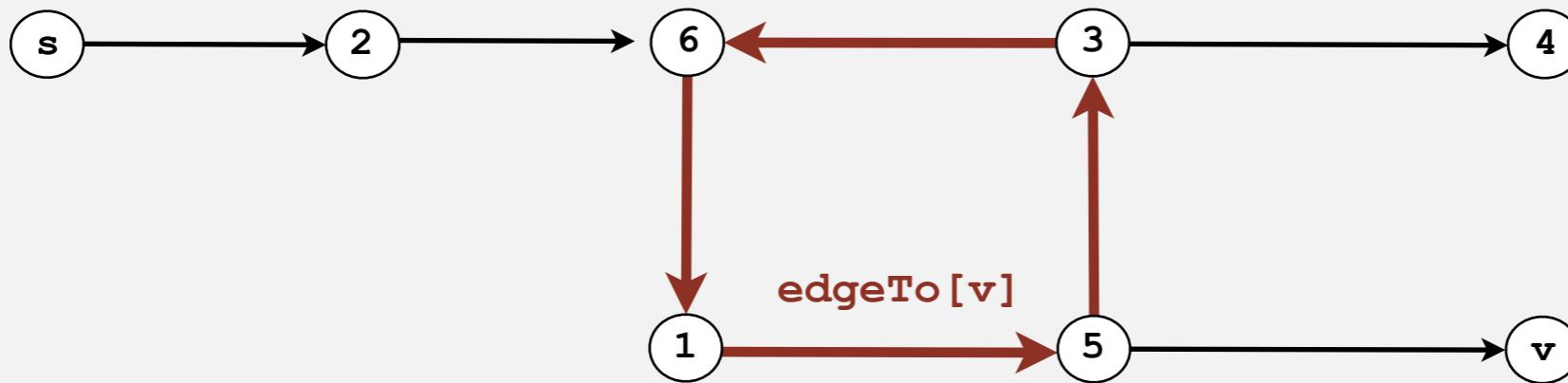


negative cycle (-0.66 + 0.37 + 0.28)

$5 \rightarrow 4 \rightarrow 7 \rightarrow 5$

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.



Proposition. If any vertex v is updated in phase V , there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.35	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.62	1	0.953
CAD	0.995	0.732	0.65	1.049	1

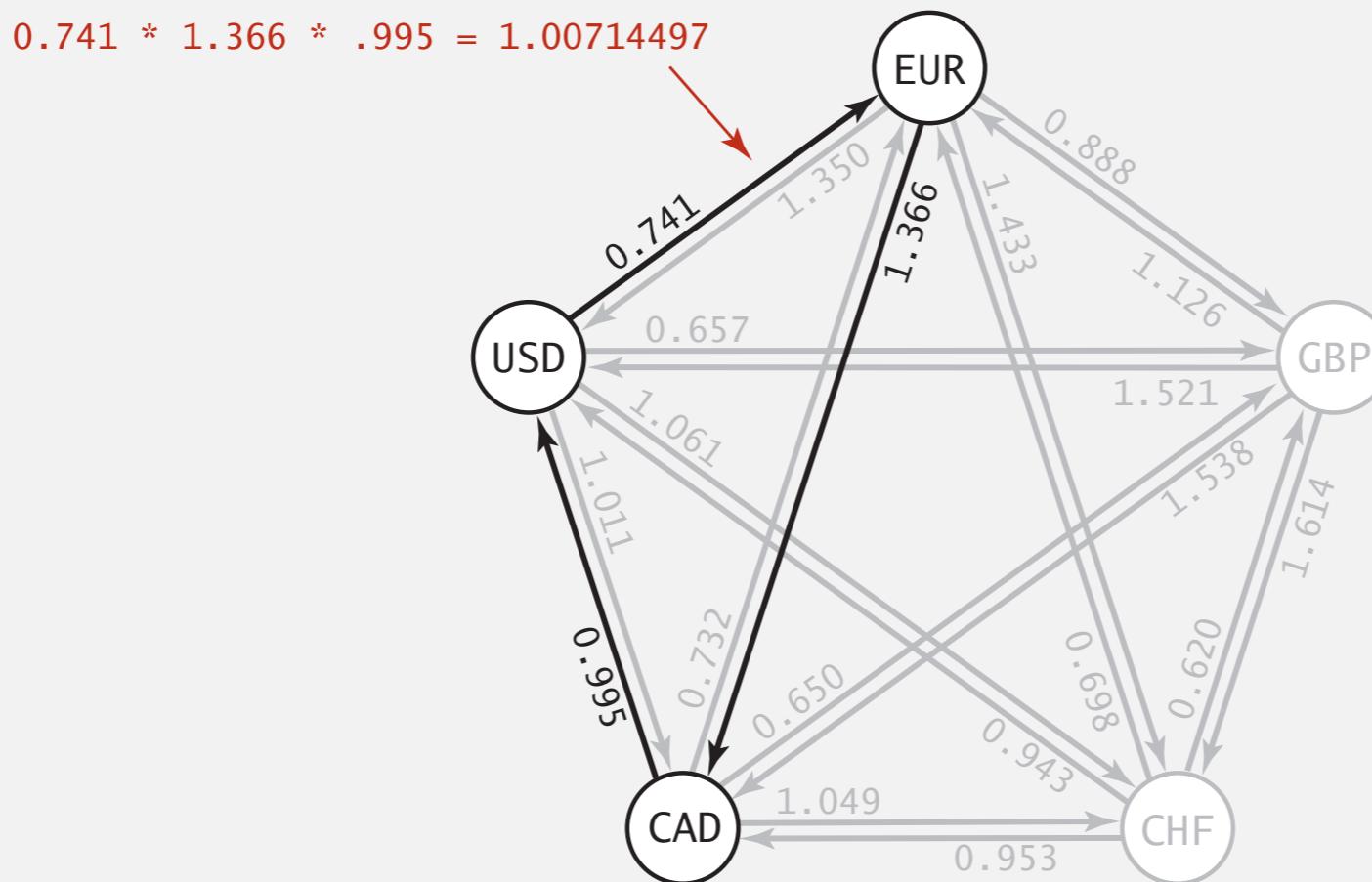
Ex. \$1,000 \Rightarrow 741 Euros \Rightarrow 1,012.206 Canadian dollars \Rightarrow \$1,007.14497.

$$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$$

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1 .

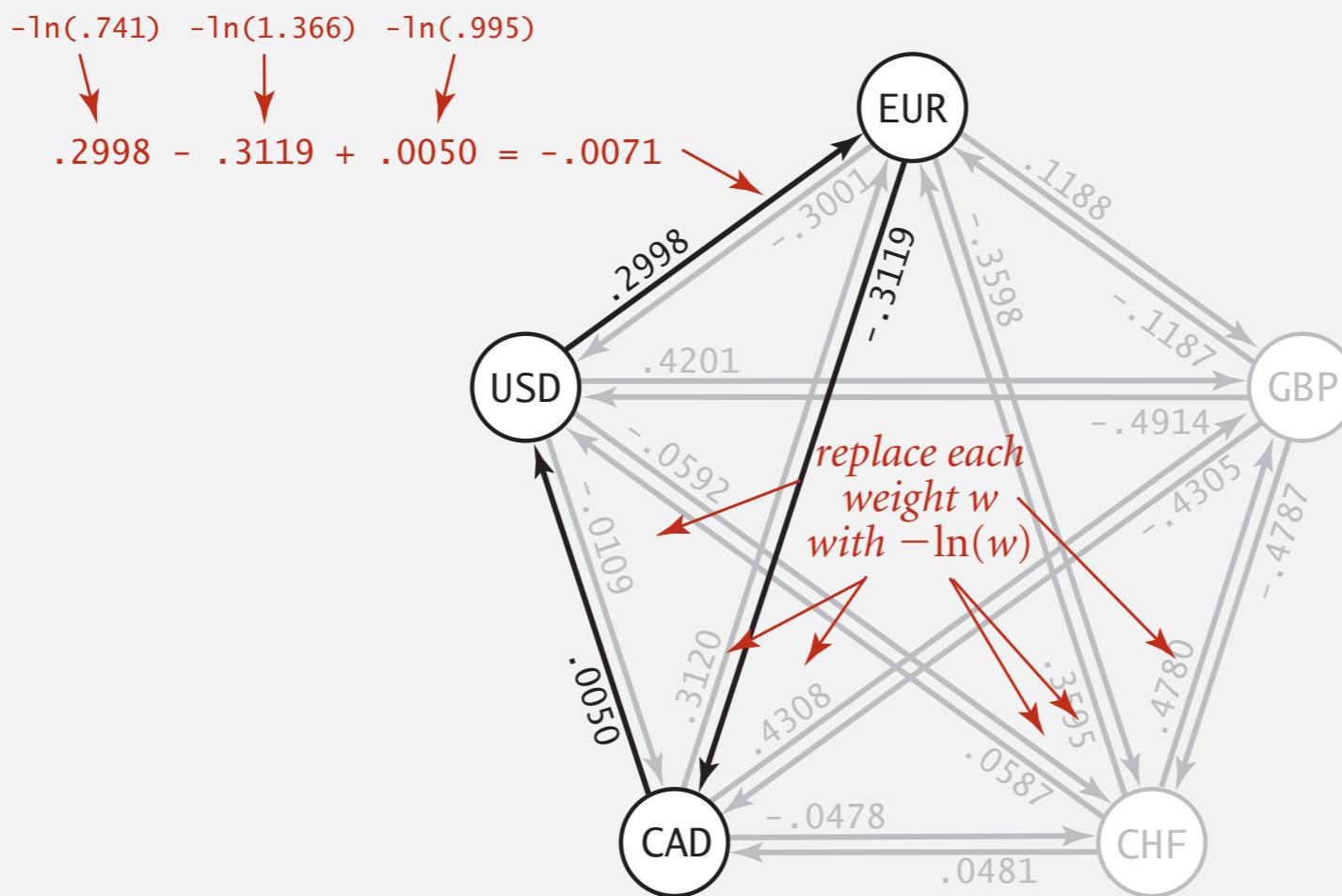


Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $-\ln$ (exchange rate from currency v to w).
 - Multiplication turns to addition; > 1 turns to < 0 .
 - Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



Remark. Fastest algorithm is extraordinarily valuable!

Shortest paths summary

Dijkstra's algorithm.

- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.

- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.

- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

Shortest-paths is a broadly useful problem-solving model.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

STRING SORTS

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

- ▶ **String sorts**
- ▶ **Key-indexed counting**
- ▶ **LSD radix sort**
- ▶ **MSD radix sort**
- ▶ **3-way radix quicksort**
- ▶ **Suffix arrays**

String processing

String. Sequence of characters.

Important fundamental abstraction.

- Information processing.
- Genomic sequences.
- Communication systems (e.g., email).
- Programming systems (e.g., Java programs).
- ...

“The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology.” — M. V. Olson

The char data type

C char data type. Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Need more bits to represent certain characters.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

Java char data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

A á ð ö
U+0041 U+00E1 U+2202 U+1D50A
Unicode characters

I (heart) Unicode



The String data type

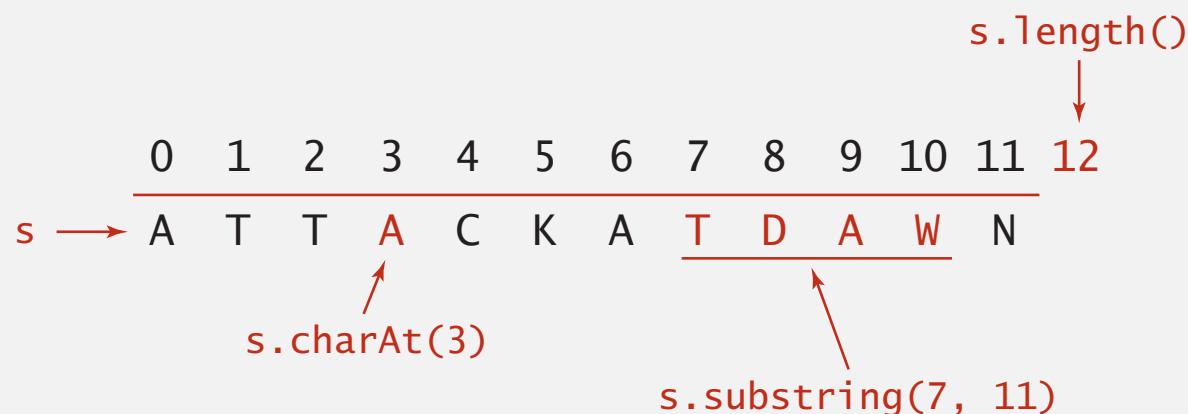
String data type. Sequence of characters (immutable).

Length. Number of characters.

Indexing. Get the i^{th} character.

Substring extraction. Get a contiguous sequence of characters.

String concatenation. Append one character to end of another string.



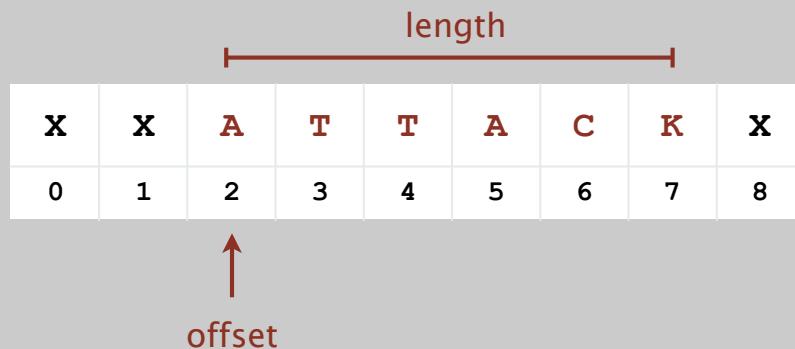
The String data type: Java implementation

```
public final class String implements Comparable<String>
{
    private char[] val;      // characters
    private int offset;     // index of first char in array
    private int length;     // length of string
    private int hash;        // cache of hashCode()
```

```
public int length()
{   return length; }
```

val[]

```
public char charAt(int i)
{   return value[i + offset]; }
```



```
private String(int offset, int length, char[] val)
{
```

```
    this.offset = offset;
    this.length = length;
    this.val    = val;
}
```

copy of reference to
original char array

```
public String substring(int from, int to)
{   return new String(offset + from, to - from, val); }
```

...

The String data type: performance

String data type. Sequence of characters (immutable).

Design Choice. Immutable, cache or share the backing array

Underlying implementation. Immutable `char[]` array, offset, and length.

String		
operation	guarantee	extra space
<code>length()</code>	1	1
<code>charAt()</code>	1	1
<code>substring()</code>	1	1
<code>concat()</code>	N	N

Memory. $40 + 2N$ bytes for a virgin `String` of length N .

can use `byte[]` or `char[]` instead of `String` to save space
(but lose convenience of `String` data type)

The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).

Design Choice. Easier to update, can't cache or share array.

Underlying implementation. Resizing `char[]` array and `length`.

operation	String		StringBuilder	
	guarantee	extra space	guarantee	extra space
<code>length()</code>				
<code>charAt()</code>				
<code>substring()</code>			N	N →
<code>concat()</code>	N	N	*	*

* amortized

Actually as of Java
1.7 this is O(n) for
String as well. Before
1.7 the initial String
and substring shared
the backing array (no
need to copy!)

Remark. `StringBuffer` data type is similar, but thread safe (and slower).

String vs. StringBuilder

Q. How to efficiently reverse a string?

A.

```
public static String reverse(String s)
{
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

quadratic time
String concatenation creates a new String and all chars in backing array are copied to new one.

B.

```
public static String reverse(String s)
{
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}
```

linear time
The backing array is updated. Sometimes may need to expand the array but amortised cost is $O(1)$

String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.

```
public static String[] suffixes(String s)
{
    int N = s.length();
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = s.substring(i, N);
    return suffixes;
}
```

linear time and
linear space

Since Strings are
immutable, the backing
array of larger String can
be shared with substring.
In Java 1.7 they changed
it, now cost is the same as
below!

B.

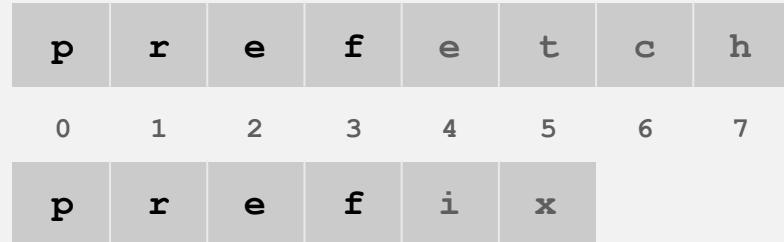
```
public static String[] suffixes(String s)
{
    int N = s.length();
    StringBuilder sb = new StringBuilder(s);
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = sb.substring(i, N);
    return suffixes;
}
```

quadratic time and
quadratic space

The array of
StringBuilder can
change, so can't share
with substring.

Longest common prefix

Q. How long to compute length of longest common prefix?



```
public static int lcp(String s, String t)
{
    int N = Math.min(s.length(), t.length());
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i))
            return i;
    return N;
}
```

linear time (worst case)
sublinear time (typical case)

Running time. Proportional to length D of longest common prefix.

Remark. Also can compute `compareTo()` in sublinear time.

Alphabets

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

Complexity of some algorithms will depend on this

name	$R()$	$\lg R()$	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIJKLMNOPQRSTUVWXYZ
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
ASCII	128	7	<i>ASCII characters</i>
EXTENDED_ASCII	256	8	<i>extended ASCII characters</i>
UNICODE16	65536	16	<i>Unicode characters</i>

STRING SORTS

- ▶ Key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ Suffix arrays

Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	yes	<code>compareTo()</code>
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	<code>compareTo()</code>

* probabilistic

Lower bound. $\sim N \lg N$ compares required by any compare-based algorithm.

Q. Can we do better (despite the lower bound)?

A. Yes, if we don't depend on key compares.

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

Remark. Keys may have associated data \Rightarrow
can't just count up number of keys of each value.

input		sorted result
<i>name</i>	<i>section</i>	(by section)
Anderson	2	Harris 1
Brown	3	Martin 1
Davis	3	Moore 1
Garcia	4	Anderson 2
Harris	1	Martinez 2
Jackson	3	Miller 2
Johnson	4	Robinson 2
Jones	3	White 2
Martin	1	Brown 3
Martinez	2	Davis 3
Miller	2	Jackson 3
Moore	1	Jones 3
Robinson	2	Taylor 3
Smith	4	Williams 3
Taylor	3	Garcia 4
Thomas	4	Johnson 4
Thompson	4	Smith 4
White	2	Thomas 4
Williams	3	Thompson 4
Wilson	4	Wilson 4

↑
keys are
small integers

Key-indexed counting demo (Count Sort)

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

R=6

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	use
0	d	a for 0
1	a	b for 1
2	c	c for 2
3	f	d for 3
4	f	e for 4
5	b	f for 5
6	d	
7	b	
8	f	
9	b	
10	e	
11	a	

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

count
frequencies

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	offset by 1 [stay tuned]
0	d	
1	a	
2	c	
3	f	
4	f	
5	b	
6	d	
7	b	
8	f	
9	b	
10	e	
11	a	

a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

compute
cumulates → count[r+1] += count[r];
or prefix-sum

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	r	count[r]
0	d	a	0
1	a	b	2
2	c	c	5
3	f	d	6
4	f	e	8
5	b	f	9
6	d	-	12
7	b	-	-
8	f	-	-
9	b	-	-
10	e	-	-
11	a	-	-

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	
1	a	1	
2	c	r count[r]	2
3	f	a	0
4	f	b	2
5	b	c	5
6	d	d	6
7	b	e	8
8	f	f	9
9	b	-	12
10	e		9
11	a		10
			11

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	
1	a	1	
2	c	r count[r]	2
3	f	a	0
4	f	b	2
5	b	c	5
6	d	d	7
7	b	e	8
8	f	f	9
9	b	-	12
10	e		9
11	a		10
			11

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	
6	d	6	
7	b	7	
8	f	8	
9	b	9	
10	e	10	
11	a	11	
		r count[r]	
		a	1
		b	2
		c	5
		d	7
		e	8
		f	9
		-	12

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	1
6	d	6	2
7	b	7	
8	f	8	
9	b	9	
10	e	10	
11	a	11	
		r count[r]	
		a	
		b	
		c	6
		d	7
		e	8
		f	9
		-	12

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	1
6	d	6	2
7	b	7	6
8	f	8	c
9	b	9	d
10	e	-	7
11	a	10	8
		9	10
		10	f
		11	

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	
3	f	3	
4	f	4	
5	b	5	1
6	d	6	2
7	b	7	6
8	f	8	c
9	b	9	d
10	e	-	7
11	a	10	8
		11	11

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	b
3	f	3	
4	f	4	
5	b	5	1
6	d	6	3
7	b	7	
8	f	8	6
9	b	9	7
10	e	10	8
11	a	11	11
		-	12

move
items

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	b
3	f	3	
4	f	4	
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	
9	b	9	f
10	e	10	f
11	a	11	

r count[*r*]

a	1	3	
b	3	4	
c	6	5	c
d	8	6	d
e	8	7	d
f	11	8	
-	12	9	f
		10	f
		11	

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

move
items →

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]
0	d
1	a
2	c
3	f
4	f
5	b
6	d
7	b
8	f
9	b
10	e
11	a

i	aux[i]
0	a
1	
2	b
3	b
4	
5	c
6	d
7	d
8	d
9	f
10	f
11	

r	count[r]
a	1
b	4
c	6
d	8
e	8
f	11
-	12

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	b
3	f	3	b
4	f	4	
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	
9	b	9	f
10	e	10	f
11	a	11	f
r	count[r]	-	12

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	
9	b	9	f
10	e	10	f
11	a	11	f

r count[*r*]

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

move
items →

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	e
9	b	9	f
10	e	10	f
11	a	11	f

r count[*r*]

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	a
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	e
9	b	9	f
-	-	10	f
10	e	11	f
11	a		

move
items

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	a
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	e
9	b	9	f
-	-	10	f
10	e	11	f
11	a		

move
items

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

copy
back

i	a[i]	i	aux[i]
0	a	0	a
1	a	1	a
2	b	2	b
3	b	3	b
4	b	4	b
5	c	5	c
6	d	6	d
7	d	7	d
8	e	8	e
9	f	9	f
10	f	10	f
11	f	11	f

r count[r]

a	2
b	5
c	6
d	8
e	9
f	12
-	12

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

count
frequencies

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	offset by 1 [stay tuned]
0	d	
1	a	
2	c	
3	f	
4	f	
5	b	
6	d	
7	b	
8	f	
9	b	
10	e	
11	a	

a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

compute
cumulates

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	r	count[r]
0	d	a	0
1	a	b	2
2	c	c	5
3	f	d	6
4	f	e	8
5	b	f	9
6	d	-	12
7	b	-	-
8	f	-	-
9	b	-	-
10	e	-	-
11	a	-	-

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

For the index of duplicates → for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	a
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	e
9	b	9	f
-	-	10	f
10	e	11	f
11	a		

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

copy
back

i	a[i]	i	aux[i]
0	a	0	a
1	a	1	a
2	b	2	b
3	b	3	b
4	b	4	b
5	c	5	c
6	d	6	d
7	d	7	d
8	e	8	e
9	f	9	f
10	f	10	f
11	f	11	f

r count[r]

Key-indexed counting: analysis

Proposition. Key-indexed counting uses $\sim 11N + 4R$ array accesses to sort N items whose keys are integers between 0 and $R - 1$.

Proposition. Key-indexed counting uses extra space proportional to $N + R$.

Stable?



a[0]	Anderson	2	Harris	1	aux[0]
a[1]	Brown	3	Martin	1	aux[1]
a[2]	Davis	3	Moore	1	aux[2]
a[3]	Garcia	4	Anderson	2	aux[3]
a[4]	Harris	1	Martinez	2	aux[4]
a[5]	Jackson	3	Miller	2	aux[5]
a[6]	Johnson	4	Robinson	2	aux[6]
a[7]	Jones	3	White	2	aux[7]
a[8]	Martin	1	Brown	3	aux[8]
a[9]	Martinez	2	Davis	3	aux[9]
a[10]	Miller	2	Jackson	3	aux[10]
a[11]	Moore	1	Jones	3	aux[11]
a[12]	Robinson	2	Taylor	3	aux[12]
a[13]	Smith	4	Williams	3	aux[13]
a[14]	Taylor	3	Garcia	4	aux[14]
a[15]	Thomas	4	Johnson	4	aux[15]
a[16]	Thompson	4	Smith	4	aux[16]
a[17]	White	2	Thomas	4	aux[17]
a[18]	Williams	3	Thompson	4	aux[18]
a[19]	Wilson	4	Wilson	4	aux[19]

Depends on the
Alphabet size / Max
integer value

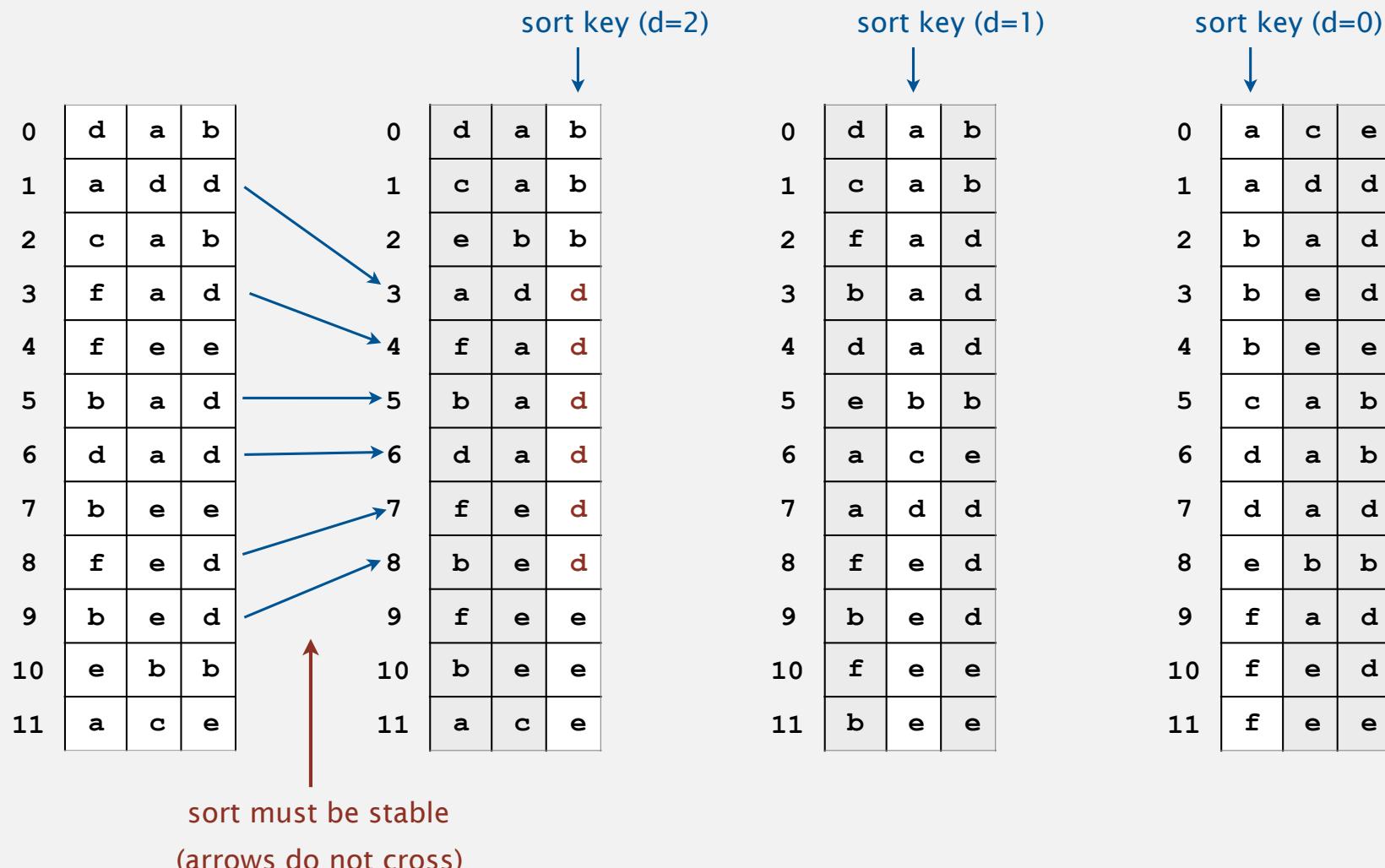
STRING SORTS

- ▶ Key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ Suffix arrays

Least-significant-digit-first string sort

LSD string (radix) sort.

- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



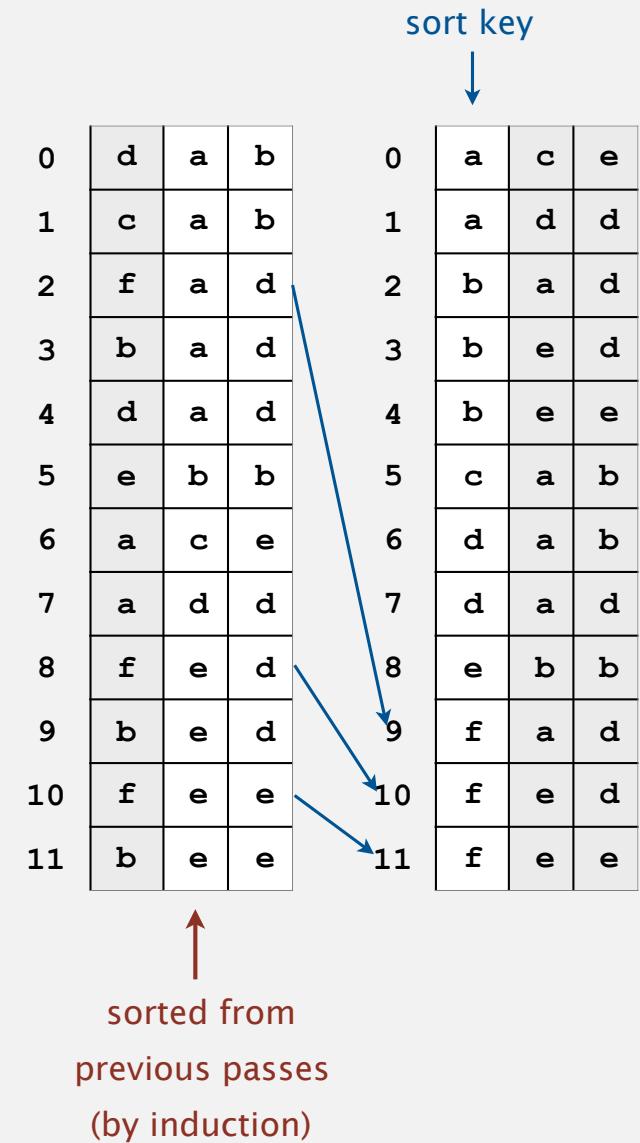
LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on i]

After pass i , strings are sorted by last i characters.

- If two strings differ on sort key,
key-indexed sort puts them in proper relative order.
- If two strings agree on sort key,
stability keeps them in proper relative order.
- [Thinking about the future]
 - If the characters not yet examined differ, it doesn't matter what we do now
 - If the characters not yet examined agree, stability ensures later pass won't affect order.



LSD string sort: Java implementation

```
public class LSD
{
    public static void sort(String[] a, int W)           ← fixed-length W strings
    {
        int R = 256;                                     ← radix R
        int N = a.length;
        String[] aux = new String[N];

        for (int d = W-1; d >= 0; d--)                  ← do key-indexed counting
        {                                                 for each digit from right to left
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++)
                a[i] = aux[i];
        }
    }
}
```

key-indexed
counting
(count sort)

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	yes	<code>compareTo()</code>
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	<code>compareTo()</code>
LSD †	$2 W N$	$2 W N$	$N + R$	yes	<code>charAt()</code>

* probabilistic

† fixed-length W keys

Q. What if strings do not have same length?

String sorting challenge I

Problem. Sort a huge commercial database on a fixed-length key.

Ex. Account number, date, Social Security number, ...

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ • LSD string sort.



256 (or 65,536) counters;
Fixed-length strings sort in W passes.

B14-99-8765	
756-12-AD46	
CX6-92-0112	
332-WX-9877	
375-99-QWAX	
CV2-59-0221	
387-SS-0321	
KJ-0 .. 12388	
715-YT-013C	
MJ0-PP-983F	
908-KK-33TY	
BBN-63-23RE	
48G-BM-912D	
982-ER-9P1B	
WBL-37-PB81	
810-F4-J87Q	
LE9-N8-XX76	
908-KK-33TY	
B14-99-8765	
CX6-92-0112	
CV2-59-0221	
332-WX-23SQ	
332-6A-9877	

String sorting challenge 2a

Problem. Sort one million 32-bit integers.

Ex. Google (or presidential) interview. Obama answered “Bubble Sort is not the way to go”

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.



Google CEO Eric Schmidt interviews Barack Obama

String sorting challenge 2a

Problem. Sort one million 32-bit integers.

Can view 32-bit integers as:

- Strings of length $W=1$ over alphabet of size $R=2^{32}$
- Strings of length $W=2$ over alphabet of size $R=2^{16}$
- Strings of length $W=3$ over alphabet of size $R=2^8$

...

- Each LSD sort out of W takes $N+R$
- If $R=2^{16}$ then we can ignore R , and reduce to $O(N)$

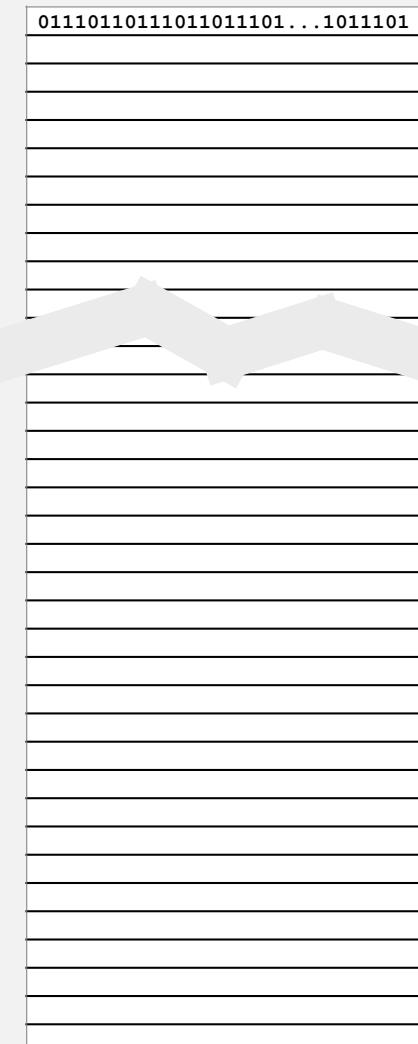
String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.



String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

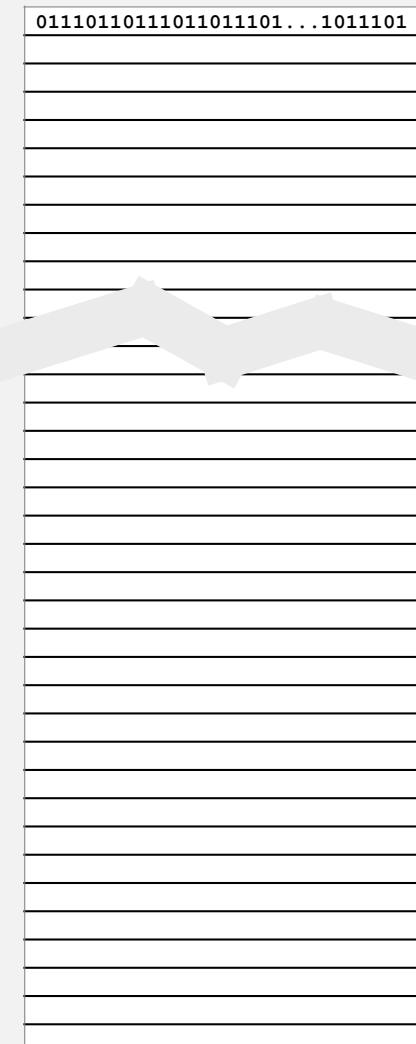
- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ • LSD string sort.



Divide each word into eight 16-bit “chars”

$2^{16} = 65,536$ counters.

Sort in 8 passes.



String sorting challenge 2b

Problem. Sort huge array of random 128-bit numbers.

Ex. Supercomputer sort, internet router.

Which sorting method to use?

- ✓ • Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- ✓ • LSD string sort.

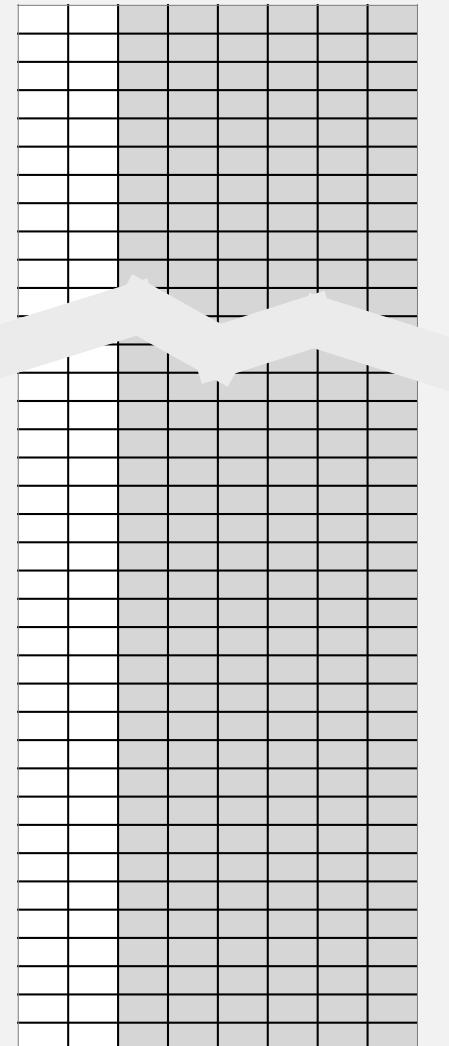
Divide each word into eight 16-bit “chars”

$2^{16} = 65,536$ counters

LSD sort on leading 32 bits in 2 passes

Finish with insertion sort

Examines only ~25% of the data



How to take a census in 1900s?

1880 Census. Took 1,500 people 7 years to manually process data.

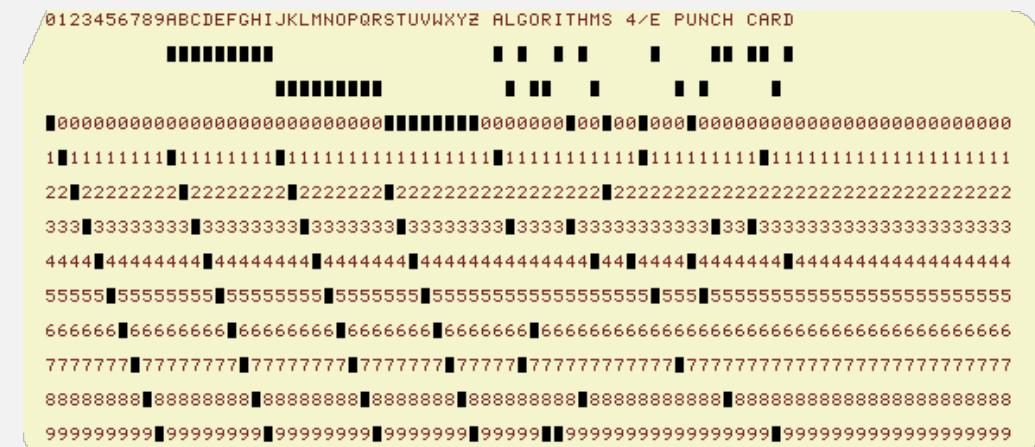


Herman Hollerith. Developed counting and sorting machine to automate.

- Use punch cards to record data (e.g., gender, age).
- Machine sorts one column at a time (into one of 12 bins).
- Typical question: how many women of age 20 to 30?



Hollerith tabulating machine and sorter



punch card (12 holes per column)

1890 Census. Finished months early and under budget!

How to get rich sorting in 1900s?

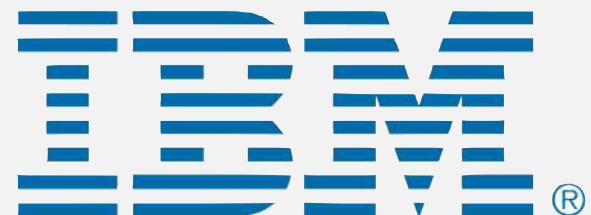
Punch cards. [1900s to 1950s]

- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

Hollerith's company later merged with 3 others to form Computing Tabulating Recording Corporation (CTR); the company was renamed in 1924.



IBM 80 Series Card Sorter (650 cards per minute)



LSD string sort: a moment in history (1960s)



card punch



punched cards



card reader



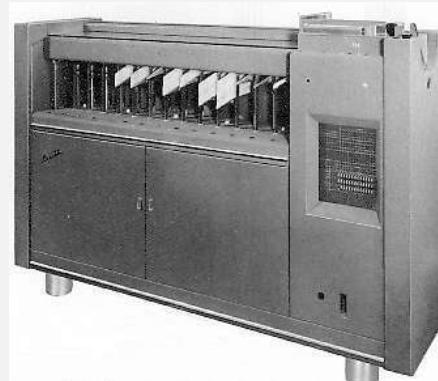
mainframe



line printer

To sort a card deck

- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted



card sorter

STRING SORTS

- ▶ Key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ Suffix arrays

Most-significant-digit-first string sort

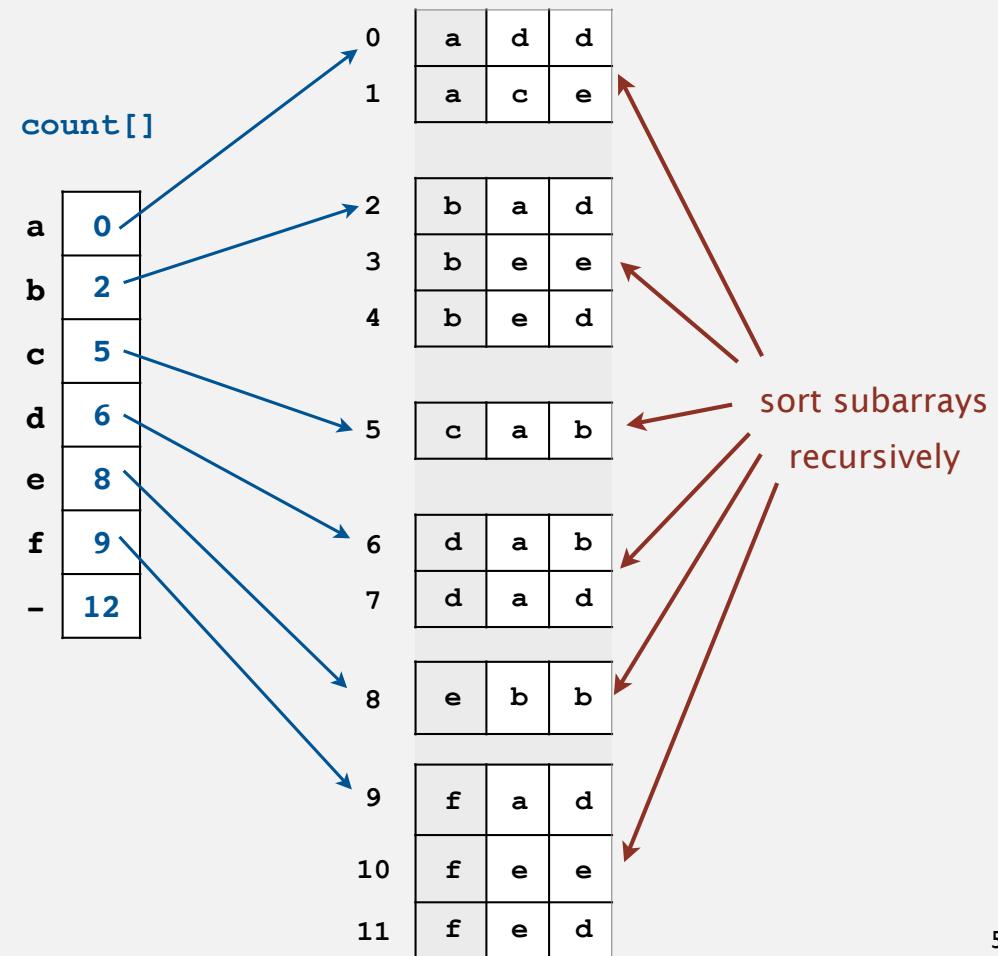
MSD string (radix) sort.

- Partition array into R pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).

0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort key



MSD string sort: example

input			d							
she	are	are		are						
sells	by	to	o	by						
seashells	she	sells		seashells	sea	sea	sea	sea	seas	sea
by	sells	seashells		sea	seashells	seashells	seashells	seashells	seashells	seashells
the	seashells	sea		seashells						
sea	sea	sells		seals	sells	sells	sells	sells	sells	sells
shore	shore	seashells		seals	sells	sells	sells	sells	sells	sells
the	shells	she		she						
shells	she	shore		shore	shore	shore	shore	shore	shells	shells
she	sells	shells		shells	shells	shells	shells	shells	shore	shore
sells	surely	she		she						
are	seashells	surely		surely						
surely	the	hi	hi	the						
seashells	the	the		the						

need to examine every character in equal keys	end-of-string goes before any char value	output
are	are	are
by	by	by
sea	sea	sea
seashells	seashells	seashells
seashells	seashells	seashells
sells	sells	sells
sells	sells	sells
she	she	she
shells	shells	shells
she	she	she
shore	shore	shore
surely	surely	surely
the	the	the
the	the	the

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Variable-length strings

Treat strings as if they had an extra char at end (smaller than any char).

why smaller?

0	s	e	a	-1							
1	s	e	a	s	h	e	l	l	s	-1	
2	s	e	l	l	s	-1					
3	s	h	e	-1							
4	s	h	e	-1							
5	s	h	e	l	l	s	-1				
6	s	h	o	r	e	-1					
7	s	u	r	e	l	y	-1				

she before shells

```
private static int charAt(String s, int d)
{
    if (d < s.length()) return s.charAt(d);
    else return -1;
}
```

C strings. Have extra char '\0' at end \Rightarrow no extra work needed.

MSD string sort: Java implementation

```
public static void sort(String[] a)
{
    aux = new String[a.length];
    sort(a, aux, 0, a.length, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int[] count = new int[R+2];
    for (int i = lo; i <= hi; i++)
        count[charAt(a[i], d) + 2]++;
    for (int r = 0; r < R+1; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d) + 1]++] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];

    for (int r = 0; r < R; r++)                      sort R subarrays recursively
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

can recycle `aux[]` array
but not `count[]` array

key-indexed counting

MSD string sort: potential for disastrous performance

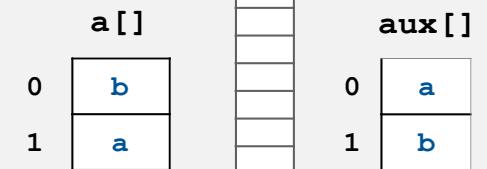
Observation 1. Much too slow for small subarrays.

- Each function call needs its own `count[]` array.
- ASCII (256 counts): 100x slower than copy pass for $N = 2$.
- Unicode (65,536 counts): 32,000x slower for $N = 2$.

`count[]`



Observation 2. Huge number of small subarrays because of recursion.



`a []`

0	b
1	a

`aux []`

0	a
1	b

Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.
- Implement `less()` so that it compares starting at d^{th} character.

```
public static void sort(String[] a, int lo, int hi, int d)
{
    for (int i = lo; i <= hi; i++)
        for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
            exch(a, j, j-1);
}
```

```
private static boolean less(String v, String w, int d)
{   return v.substring(d).compareTo(w.substring(d)) < 0; }
```



in Java, forming and comparing
substrings is faster than directly
comparing chars with `charAt()`

MSD string sort: performance

Number of characters examined.

- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!



compareTo() based sorts can also be sublinear!	Random (sublinear)	Non-random with duplicates (nearly linear)	Worst case (linear)
	1EI0402	are	1DNB377
	1HYL490	by	1DNB377
	1R0Z572	sea	1DNB377
	2HXE734	seashells	1DNB377
	2IYE230	seashells	1DNB377
	2XOR846	sells	1DNB377
	3CDB573	sells	1DNB377
	3CVP720	she	1DNB377
	3IGJ319	she	1DNB377
	3KNA382	shells	1DNB377
	3TAV879	shore	1DNB377
	4CQP781	surely	1DNB377
	4QGI284	the	1DNB377
	4YHV229	the	1DNB377

Characters examined by MSD string sort

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	yes	<code>compareTo()</code>
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	<code>compareTo()</code>
LSD †	$2 NW$	$2 NW$	$N + R$	yes	<code>charAt()</code>
MSD ‡	$2 NW$	$N \log_R N$	$N + DR$	yes	<code>charAt()</code>

D = function-call stack depth
(length of longest prefix match)



* probabilistic

† fixed-length W keys

‡ average-length W keys

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

- Accesses memory "randomly" (cache inefficient).
- Inner loop has a lot of instructions.
- Extra space for `count[]`.
- Extra space for `aux[]`.

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

Goal. Combine advantages of MSD and quicksort.

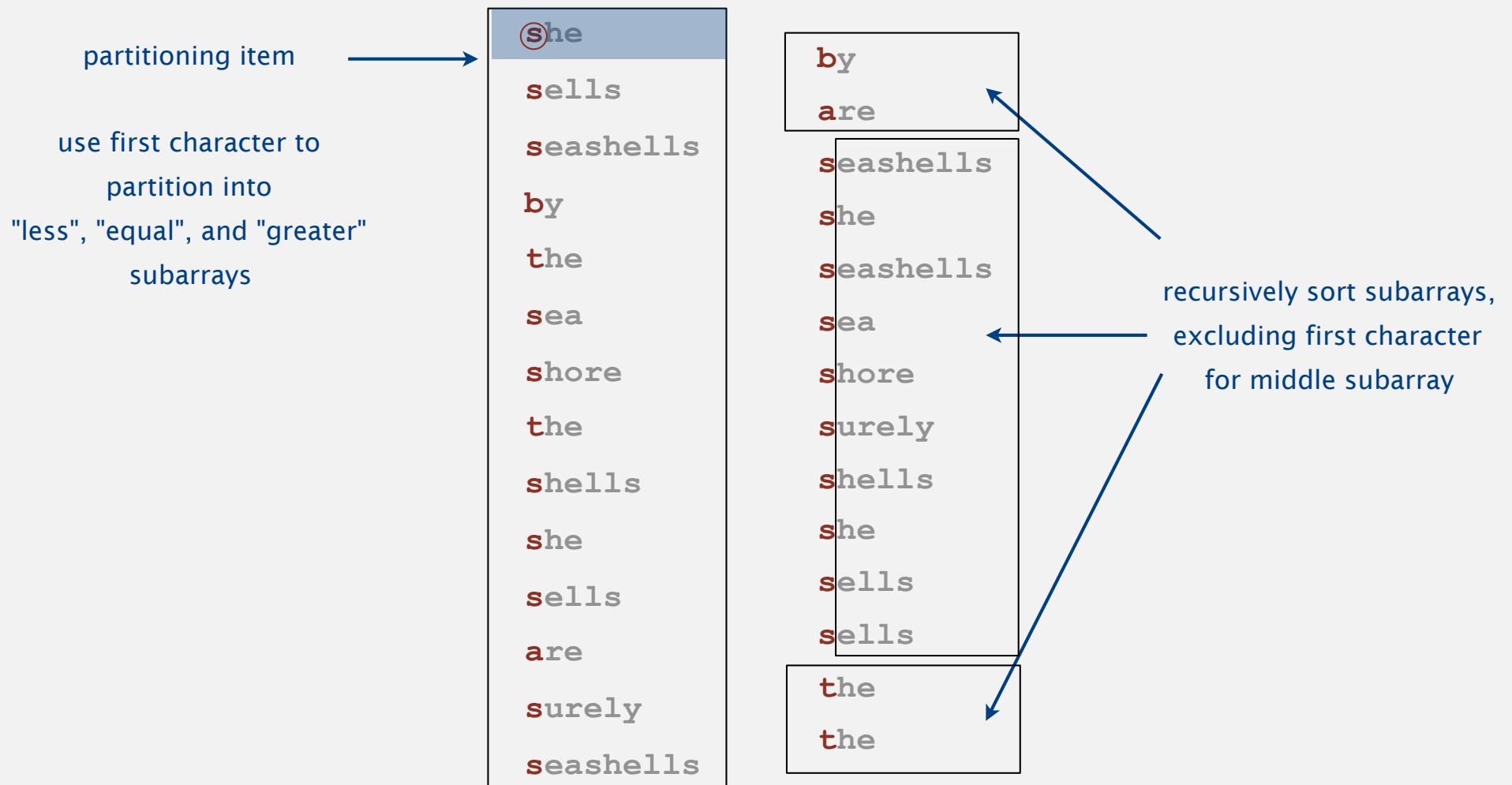
STRING SORTS

- ▶ Key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ Suffix arrays

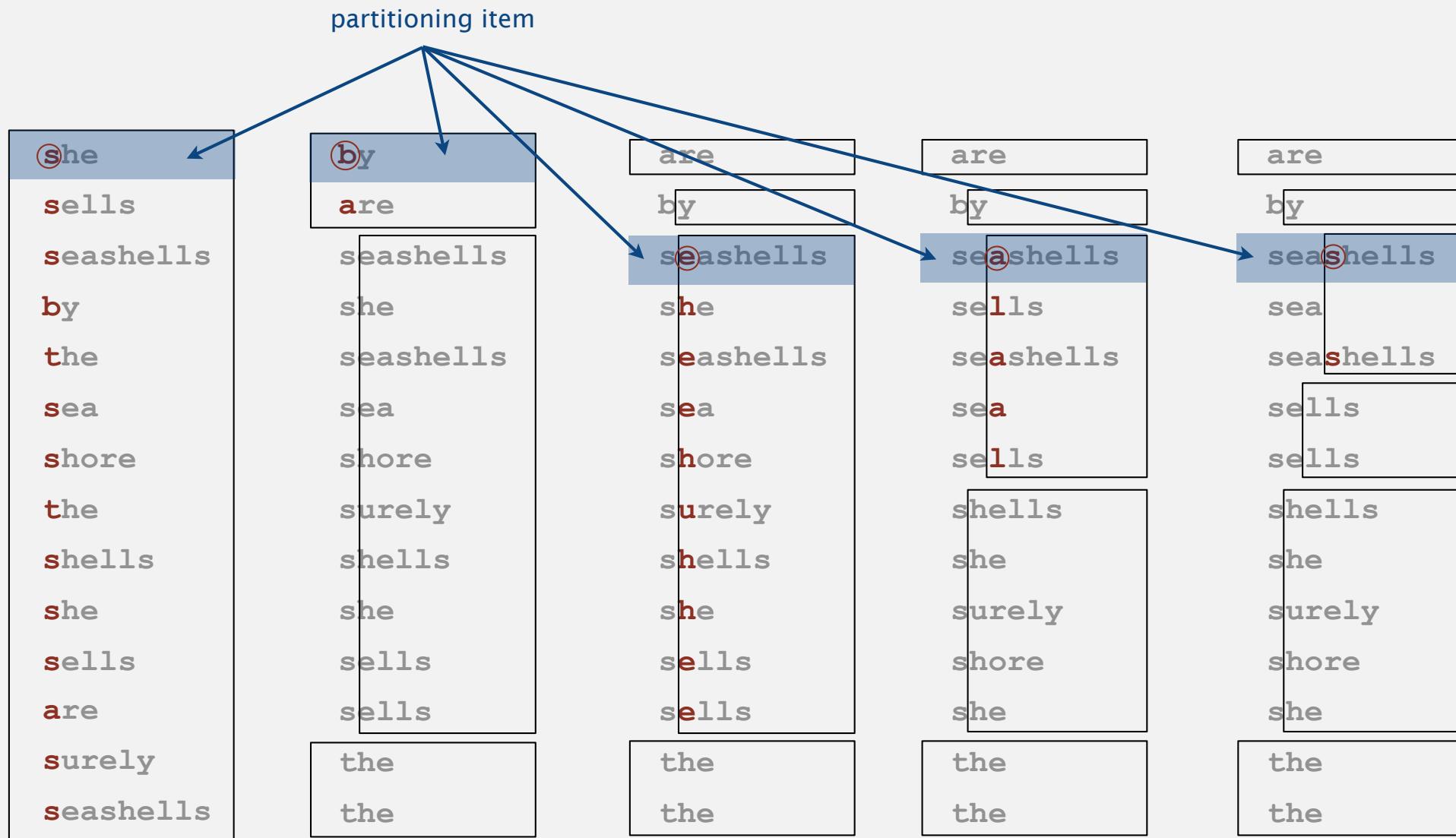
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the d^{th} character.

- Less overhead than R -way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char
(but does re-examine characters not equal to the partitioning char).



3-way string quicksort: trace of recursive calls



Trace of first few recursive calls for 3-way string quicksort (subarrays of size 1 not shown)

3-way string quicksort: Java implementation

```
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

private static void sort(String[] a, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    int v = charAt(a[lo], d);
    int i = lo + 1;
    while (i <= gt)
    {
        int t = charAt(a[i], d);
        if         (t < v) exch(a, lt++, i++);
        else if   (t > v) exch(a, i, gt--);
        else             i++;
    }
    sort(a, lo, lt-1, d);
    if (v >= 0) sort(a, lt, gt, d+1); ← sort 3 subarrays recursively
    sort(a, gt+1, hi, d);
}
```

3-way partitioning
(using d^{th} character)

to handle variable-length strings

3-way string quicksort vs. standard quicksort

Standard quicksort.

- Uses $\sim 2N \ln N$ **string compares** on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.

- Uses $\sim 2N \ln N$ **character compares** on average for random strings.
- Avoids re-comparing long common prefixes.

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm blends tries and binary

that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

3-way string quicksort vs. MSD string sort

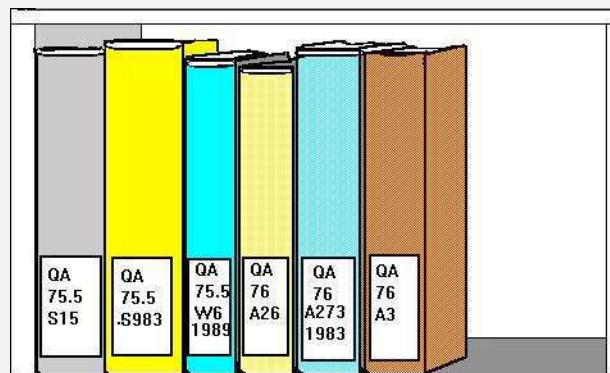
MSD string sort.

- Is cache-inefficient.
- Too much memory storing `count[]`.
- Too much overhead reinitializing `count[]` and `aux[]`.

3-way string quicksort.

- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

library of Congress call numbers



Bottom line. 3-way string quicksort is the method of choice for sorting strings.

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$N^2 / 2$	$N^2 / 4$	1	yes	<code>compareTo()</code>
mergesort	$N \lg N$	$N \lg N$	N	yes	<code>compareTo()</code>
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	<code>compareTo()</code>
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	<code>compareTo()</code>
LSD †	$2 NW$	$2 NW$	$N + R$	yes	<code>charAt()</code>
MSD ‡	$2 NW$	$N \log_R N$	$N + DR$	yes	<code>charAt()</code>
3-way string quicksort	$1.39 W N \lg N$ *	$1.39 N \lg N$	$\log N + W$	no	<code>charAt()</code>

* probabilistic

† fixed-length W keys

‡ average-length W keys

STRING SORTS

- ▶ Key-indexed counting
- ▶ LSD radix sort
- ▶ MSD radix sort
- ▶ 3-way radix quicksort
- ▶ Suffix arrays

Keyword-in-context search

Given a text of N characters, preprocess it to enable fast substring search
(find all occurrences of query string context).

```
% java KWIC tale.txt 15 ← characters of
      search                      surrounding context
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
dispersing in search of other carri
n that bed and search the straw hold

better thing
t is a far far better thing that i do than
some sense of better things else forgotte
was capable of better things mr carton ent
```

Applications. Linguistics, databases, web search, word processing,

Suffix sort

input string

a a c a a g t t t a c a a g c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

form suffixes

0	a a c a a g t t t a c a a g c
1	a c a a g t t t a c a a g c
2	c a a g t t t a c a a g c
3	a a g t t t a c a a g c
4	a g t t t a c a a g c
5	g t t t a c a a g c
6	t t t a c a a g c
7	t t a c a a g c
8	t a c a a g c
9	a c a a g c
10	c a a g c
11	a a g c
12	a g c
13	g c
14	c

sort suffixes to bring repeated substrings together

0	a a c a a g t t t a c a a g c
11	a a g c
3	a a g t t t a c a a g c
9	a c a a g c
1	a c a a g t t t a c a a g c
12	a g c
4	a g t t t a c a a g c
14	c
10	c a a g c
2	c a a g t t t a c a a g c
13	g c
5	g t t t a c a a g c
8	t a c a a g c
7	t t a c a a g c
6	t t t a c a a g c



Keyword-in-context search: suffix-sorting solution

- Preprocess: **suffix sort** the text.
- Query: **binary search** for query; scan until mismatch.

KWIC search for "search" in Tale of Two Cities

		:
632698	s e a l e d _ m y _ l e t t e r _ a n d _ ...	
713727	s e a m s t r e s s _ i s _ l i f t e d _ ...	
660598	s e a m s t r e s s _ o f _ t w e n t y _ ...	
67610	s e a m s t r e s s _ w h o _ w a s _ w i ...	
4430	s e a r c h - f o r - c o n t r a b a n d ...	
42705	s e a r c h - f o r - y o u r - f a t h e ...	
499797	s e a r c h - o f - h e r - h u s b a n d ...	
182045	s e a r c h - o f - i m p o v e r i s h e ...	
143399	s e a r c h - o f - o t h e r - c a r r i ...	
411801	s e a r c h - t h e - s t r a w - h o l d ...	
158410	s e a r e d - m a r k i n g - a b o u t - ...	
691536	s e a s - a n d - m a d a m e - d e f a r ...	
536569	s e a s e - a - t e r r i b l e - p a s s ...	
484763	s e a s e - t h a t - h a d - b r o u g h ...	
	:	

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

```
a a c a a g t t t a c a a g c a t g a t g c t g t a c t a  
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a  
c c t a a t c c t t g t g t g t a c a c a c a c a c t a c t a  
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a  
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t  
a g a t a g a t a g a c c c c t a g a t a c a c a c a t a c a  
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a  
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c  
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a  
g a c a g a a a a a a a a a c t c t a t a c t a t a a a a a
```

Applications. Bioinformatics, cryptanalysis, data compression, ...

Longest repeated substring: a musical application

Visualize repetitions in music. <http://www.bewitched.com>

Mary Had a Little Lamb



Bach's Goldberg Variations



Longest repeated substring

Given a string of N characters, find the longest repeated substring.

Brute-force algorithm.

- Try all indices i and j for start of possible match.
- Compute longest common prefix (LCP) for each pair.



Analysis. Running time $\leq D N^2$, where D is length of longest match.

Longest repeated substring: a sorting solution

input string

a a c a a g t t t a c a a g c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

form suffixes

0	a a c a a g t t t a c a a g c
1	a c a a g t t t a c a a g c
2	c a a g t t t a c a a g c
3	a a g t t t a c a a g c
4	a g t t t a c a a g c
5	g t t t a c a a g c
6	t t t a c a a g c
7	t t a c a a g c
8	t a c a a g c
9	a c a a g c
10	c a a g c
11	a a g c
12	a g c
13	g c
14	c

sort suffixes to bring repeated substrings together

0	a a c a a g t t t a c a a g c
11	a a g c
3	a a g t t t a c a a g c
9	a c a a g c
1	a c a a g t t t a c a a g c
12	a g c
4	a g t t t a c a a g c
14	c
10	c a a g c
2	c a a g t t t a c a a g c
13	g c
5	g t t t a c a a g c
8	t a c a a g c
7	t t a c a a g c
6	t t t a c a a g c

compute longest prefix between adjacent suffixes

a a c a a g t t t a c a a g c

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Longest repeated substring: Java implementation

```
public String lrs(String s)
{
    int N = s.length();

    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = s.substring(i, N);

    Arrays.sort(suffixes);

    String lrs = "";
    for (int i = 0; i < N-1; i++)
    {
        int len = lcp(suffixes[i], suffixes[i+1]);
        if (len > lrs.length())
            lrs = suffixes[i].substring(0, len);
    }
    return lrs;
}
```

create suffixes
(linear time and space)

sort suffixes

find LCP between
adjacent suffixes in
sorted order

```
% java LRS < moby dick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```

Sorting challenge

Problem. Five scientists A, B, C, D , and E are looking for long repeated substring in a genome with over 1 billion nucleotides.

- A has a grad student do it by hand.
- B uses brute force (check all pairs).
- C uses suffix sorting solution with insertion sort.
- D uses suffix sorting solution with LSD string sort.
- ✓ • E uses suffix sorting solution with 3-way string quicksort.

 but only if LRS is not long (!)

Q. Which one is more likely to lead to a cure cancer?

Longest repeated substring: empirical analysis

input file	characters	brute	suffix sort	length of LRS
<code>LRS.java</code>	2.162	0.6 sec	0.14 sec	73
<code>amendments.txt</code>	18.369	37 sec	0.25 sec	216
<code>aesop.txt</code>	191.945	1.2 hours	1.0 sec	58
<code>mobydick.txt</code>	1.2 million	43 hours †	7.6 sec	79
<code>chromosome11.txt</code>	7.1 million	2 months †	61 sec	12.567
<code>pi.txt</code>	10 million	4 months †	84 sec	14
<code>pipi.txt</code>	20 million	forever †	???	10 million

† estimated

Suffix sorting: worst-case input

Bad input: longest repeated substring very long.

- Ex: same letter repeated N times.
- Ex: two copies of the same Java codebase.

	form suffixes	sorted suffixes
0	t w i n s t w i n s	i n s
1	w i n s t w i n s	i n s t w i n s
2	i n s t w i n s	n s
3	n s t w i n s	n s t w i n s
4	s t w i n s	s
5	t w i n s	s t w i n s
6	w i n s	t w i n s
7	i n s	t w i n s t w i n s
8	n s	w i n s
9	s	w i n s t w i n s

LRS needs at least $1 + 2 + 3 + \dots + D$ character compares,
where $D = \text{length of longest match}$

Running time. Quadratic (or worse) in the length of the longest match.

Suffix sorting challenge

Problem. Suffix sort an arbitrary string of length N .

Q. What is worst-case running time of best algorithm for problem?

- Quadratic.
- ✓ • Linearithmic. ← Manber's algorithm
- ✓ • Linear. ← suffix trees (beyond our scope)
- Nobody knows.

Suffix sorting in linearithmic time

Manber's MSD algorithm overview.

- Phase 0: sort on first character using key-indexed counting sort.
- Phase i : given array of suffixes sorted on first 2^{i-1} characters, create array of suffixes sorted on first 2^i characters.

Worst-case running time. $N \lg N$.

- Finishes after $\lg N$ phases.
- Can perform a phase in linear time. (!) [ahead]

Linearithmic suffix sort example: phase 0

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a a b c b a b a a a a a 0
2	b a a a a a b c b a b a a a a a 0
3	a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a a 0
12	a a a a a a 0
13	a a a a a 0
14	a a a a 0
15	a a a 0
16	a a 0
17	0

key-indexed counting sort (first character)

17	0
1	a b a a a a a b c b a b a a a a a 0
16	a 0
3	a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
15	a a 0
14	a a a 0
13	a a a a 0
12	a a a a a 0
10	a b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
9	b a b a a a a a 0
11	b a a a a a 0
7	b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
8	c b a b a a a a a 0

↑
sorted

Linearithmic suffix sort example: phase I

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a a b c b a b a a a a a 0
2	b a a a a a b c b a b a a a a a 0
3	a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a a 0
12	a a a a a a 0
13	a a a a a 0
14	a a a a 0
15	a a a 0
16	a a 0
17	0

index sort (first two characters)

17	0
16	a 0
12	a a a a a 0
3	a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
13	a a a a a 0
15	a a a 0
14	a a a a 0
6	a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
10	a b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
9	b a b a a a a a 0
11	b a a a a a a 0
2	b a a a a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0

↑
sorted

Linearithmic suffix sort example: phase 2

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a a b c b a b a a a a a 0
2	b a a a a a b c b a b a a a a a 0
3	a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a a 0
12	a a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first four characters)

17	0
16	a 0
15	a a 0
14	a a a 0
3	a a a a b c b a b a a a a a 0
12	a a a a a 0
13	a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
10	a b a a a a a 0
6	a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0 0 0
11	b a a a a a 0
0	b a b a a a b c b a b a a a a a 0
9	b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0

↑
sorted

Linearithmic suffix sort example: phase 3

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a a b c b a b a a a a a 0
2	b a a a a a b c b a b a a a a a 0
3	a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a 0
12	a a a a a 0
13	a a a a 0
14	a a a 0
15	a a 0
16	a 0
17	0

index sort (first eight characters)

17	0
16	a 0
15	a a 0
14	a a a 0
13	a a a a 0
12	a a a a a 0
3	a a a a b c b a b a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
10	a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
11	b a a a a a a 0
2	b a a a a b c b a b a a a a a 0 0 0
9	b a b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0

finished (no equal keys)

Constant-time string compare by indexing into inverse

	original suffixes	index sort (first four characters)	inverse frequencies
0	b a b a a a a b c b a b a a a a a 0	17 0	0 14
1	a b a a a a a b c b a b a a a a a 0	16 a 0	1 9
2	b a a a a a b c b a b a a a a a 0	15 a a 0	2 12
3	a a a a a b c b a b a a a a a 0	14 a a a 0	3 4
4	a a a b c b a b a a a a a 0	3 a a a a b c b a b a a a a a 0	4 7
5	a a b c b a b a a a a a 0	12 a a a a a 0	5 8
6	a b c b a b a a a a a 0	13 a a a a 0	6 11
7	b c b a b a a a a a 0	4 a a a b c b a b a a a a a 0	7 16
8	c b a b a a a a a 0	5 a a b c b a b a a a a a 0	8 17
9	b a b a a a a a 0	1 a b a a a a a b c b a b a a a a a 0	9 15
10	a b a a a a a 0	10 a b a a a a a a 0	10 10
11	b a a a a a a 0	6 a b c b a b a a a a a 0	11 13
12	a a a a a a 0	2 b a a a a b c b a b a a a a a 0 0 0	12 5
13	a a a a a 0	11 b a a a a a a 0	13 6
14	a a a a 0	0 b a b a a a a b c b a b a a a a 0	14 3
15	a a 0	9 b a b a a a a a a 0	15 2
16	a 0	7 b c b a b a a a a a 0	16 1
17	0	8 c b a b a a a a a 0	17 0

To do this, inverse-index should be computed for the previous phase. May use for only the last phase

`suffixes4[13] ≤ suffixes4[4]` (because `inverse[13] < inverse[4]`)

`SO suffixes8[9] ≤ suffixes8[0]`

Suffix sort: experimental results

time to suffix sort (seconds)

algorithm	moby dick.txt	aesop aesop.txt
brute-force	36.000 †	4000 †
quicksort	9,5	167
LSD	not fixed length	not fixed length
MSD	395	out of memory
MSD with cutoff	6,8	162
3-way string quicksort	2,8	400
Manber MSD	17	8,5

† estimated

String sorting summary

We can develop linear-time sorts.

- Key compares not necessary for string keys.
- Use characters as index in an array.

We can develop sublinear-time sorts.

- Should measure amount of data in keys, not number of keys.
- Not all of the data has to be examined.

3-way string quicksort is asymptotically optimal.

- $1.39 N \lg N$ chars for random data.

Long strings are rarely random in practice.

- Goal is often to learn the structure!
- May need specialized algorithms.

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

SUBSTRING SEARCH

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

TODAY

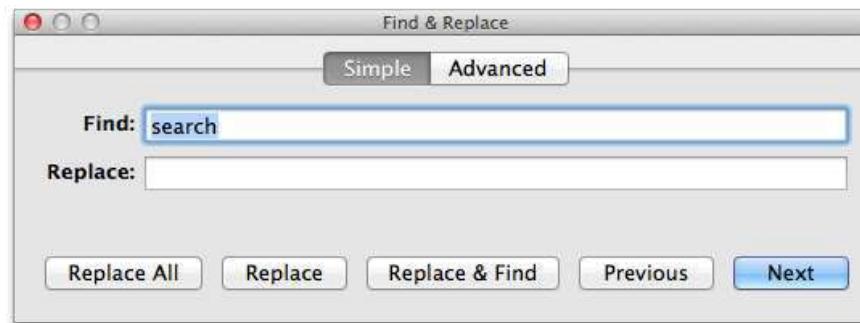
- ▶ **Substring search**
- ▶ **Brute force**
- ▶ **Knuth-Morris-Pratt**
- ▶ **Boyer-Moore**
- ▶ **Rabin-Karp**

Substring search

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E



Substring search applications

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.



<http://citp.princeton.edu/memory>

Substring search applications

Goal. Find pattern of length M in a text of length N .

typically $N \gg M$

pattern → N E E D L E

Identify patterns indicative of spam.

- The logo for SpamAssassin, which includes a detailed illustration of a hawk's head and body in flight, perched atop a blue rectangular box containing the brand name.
- PROFITS
 - LOSE WEIGHT
 - There is no catch.
 - This is a one-time mailing.
 - This message is sent in compliance with spam regulations.



Substring search applications

Electronic surveillance.



Need to monitor all
internet traffic.
(security)

No way!
(privacy)



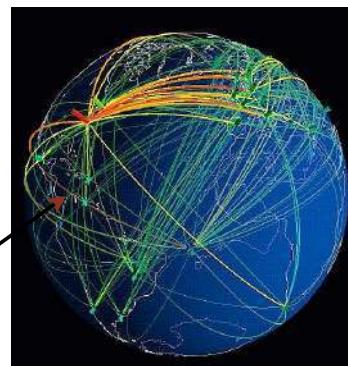
Well, we're mainly
interested in
“ATTACK AT DAWN”



OK. Build a
machine that just
looks for that.



“ATTACK AT DAWN”
substring search
machine
found



Substring search applications

Screen scraping. Extract relevant data from web page.

Ex. Find string delimited by **** and **** after first occurrence of pattern **Last Trade:**.



```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>452.92</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
...
```

Screen scraping: Java implementation

Java library. The `indexOf()` method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

```
public class StockQuote
{
    public static void main(String[] args)
    {
        String name = "http://finance.yahoo.com/q?s=";
        In in = new In(name + args[0]);
        String text = in.readAll();
        int start    = text.indexOf("Last Trade:", 0);
        int from     = text.indexOf("<b>", start);
        int to       = text.indexOf("</b>", from);
        String price = text.substring(from + 3, to);
        StdOut.println(price);
    }
}

% java StockQuote goog
582.93

% java StockQuote msft
24.84
```

SUBSTRING SEARCH

- ▶ **Brute force**
- ▶ **Knuth-Morris-Pratt**
- ▶ **Boyer-Moore**
- ▶ **Rabin-Karp**

Brute-force substring search

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A	← pat						
1	0	1		A	B	R	A						
2	1	3			B	R	A						
3	0	3				A	B	R	A				
4	1	5					B	R	A				
5	0	5						R	A				
6	4	10							A	B	R	A	

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

return i when j is M

match

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

i	j	i + j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5						A	D	A	C	R	

```
public static int search(String pat, String txt)
{
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == M) return i; ← index in text where
    }                                pattern starts
    return N; ← not found
}
```

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	$i+j$	0	1	2	3	4	5	6	7	8	9
			txt →	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B ← pat					
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

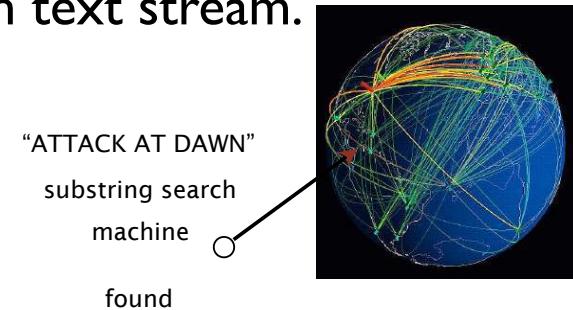
$\overbrace{\quad\quad\quad\quad}^{\text{match}}$

Worst case. $\sim M N$ char compares.

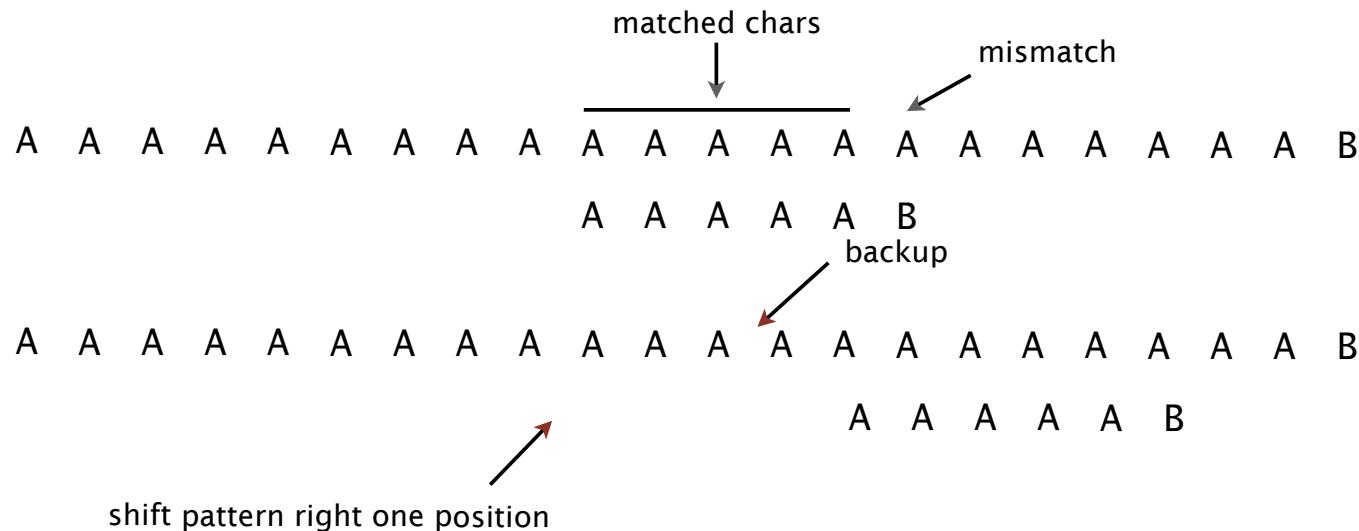
Backup

In many applications, we want to avoid backup in text stream.

- Treat input as stream of data.
- Abstract model: standard input.



Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- i points to end of sequence of already-matched chars in text.
- j stores number of already-matched chars (end of sequence in pattern).

i	j	0	1	2	3	4	5	6	7	8	9	10
		A	B	A	C	A	D	A	B	R	A	C
7	3					A	D	A	C	R		
5	0					A	D	A	C	R		

```
public static int search(String pat, String txt)
{
    int i, N = txt.length();
    int j, M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++)
    {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0; }                                ← backup
    }
    if (j == M) return i - M;
    else          return N;
}
```

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their attack at dawn party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

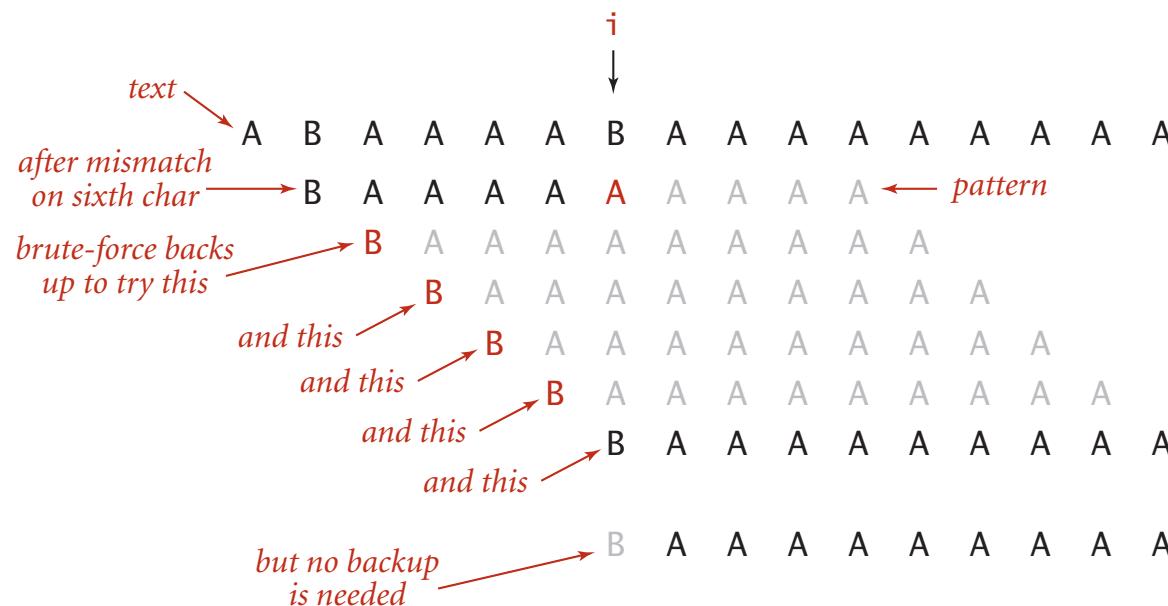
SUBSTRING SEARCH

- ▶ **Brute force**
- ▶ **Knuth-Morris-Pratt**
- ▶ **Boyer-Moore**
- ▶ **Rabin-Karp**

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern **BAAAAAAAAAA**.

assuming { A, B } alphabet



Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

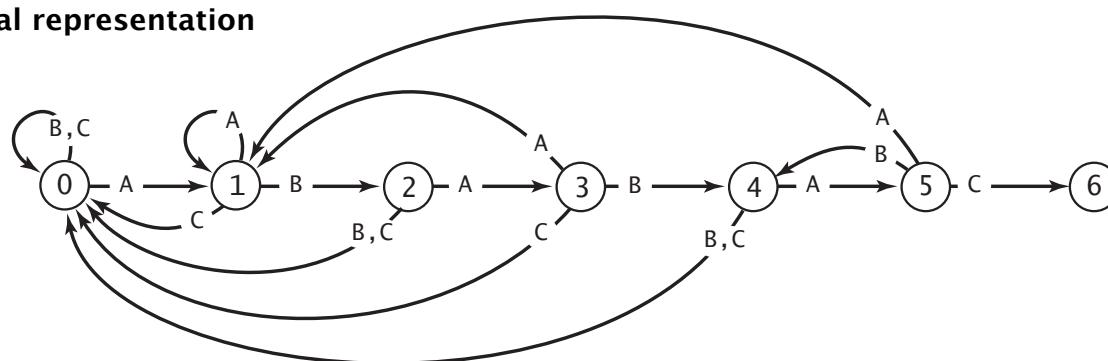
- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

internal representation

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[] [j]	1	1	3	1	5	1
A	0	2	0	4	0	4
B	0	0	0	0	0	6
C	0	0	0	0	0	6

If in state **j** reading char **c**:
if **j** is **6** halt and accept
• else move to state **dfa[c][j]**

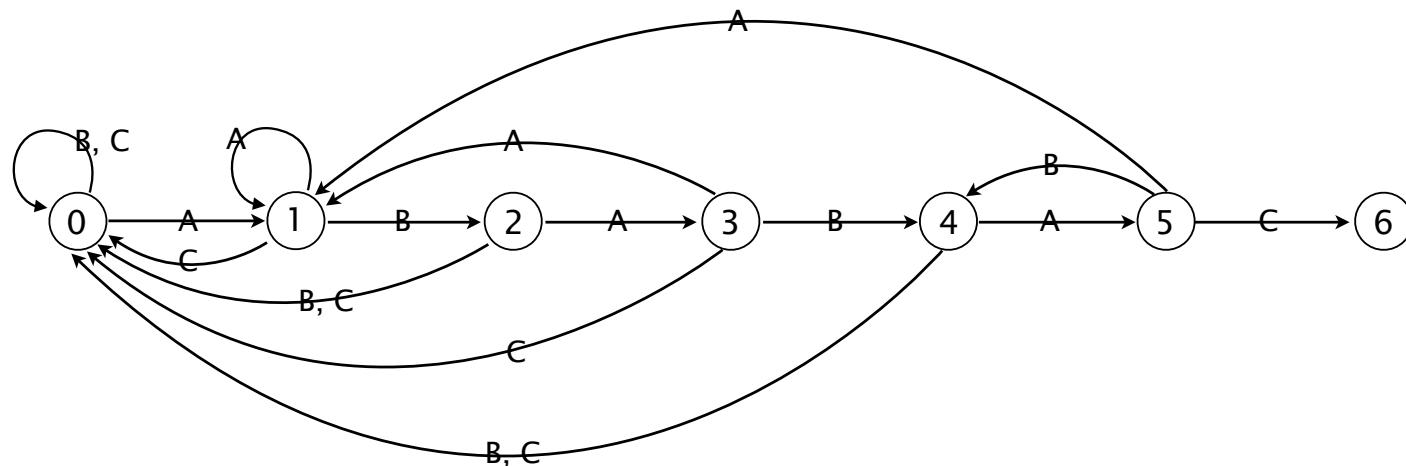
graphical representation



DFA simulation

A A B A C A A B A B A C A A

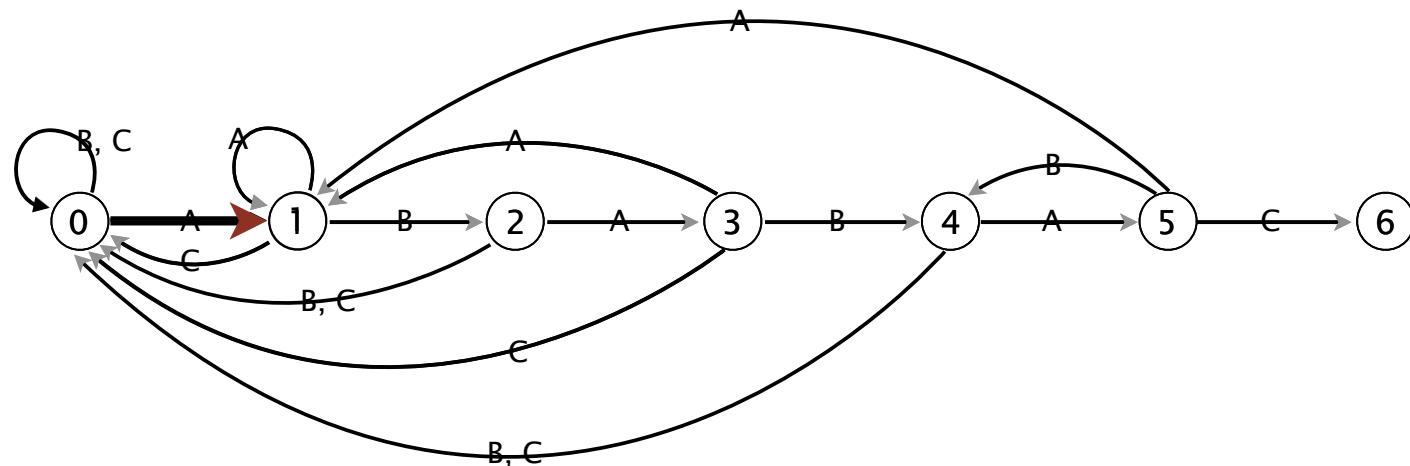
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	



DFA simulation

A A B A C A A B A B A C A A

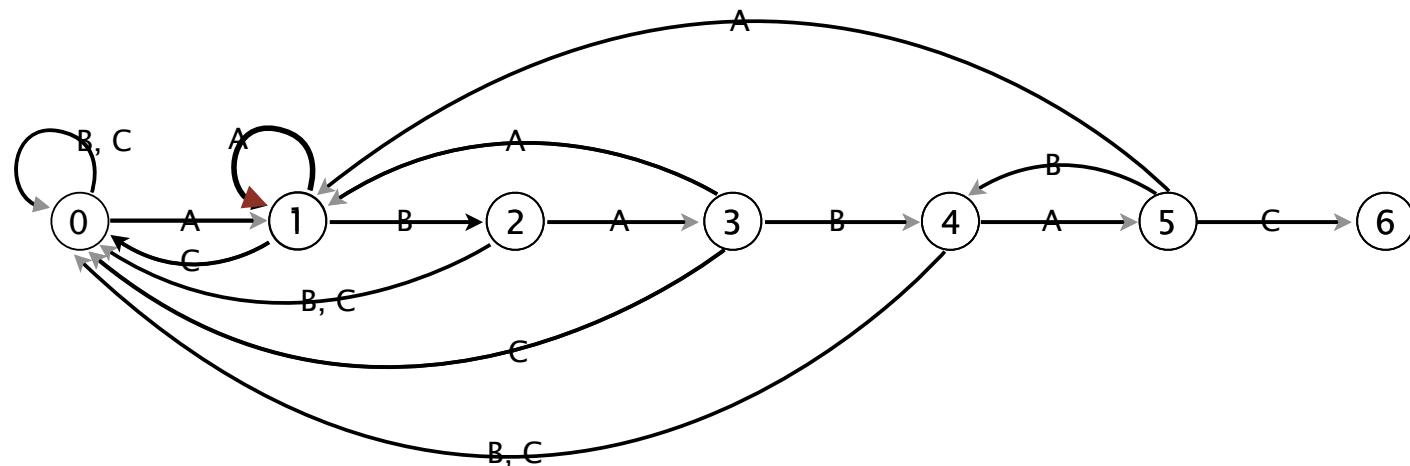
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	B	0	2	0	4	0	4
		0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A

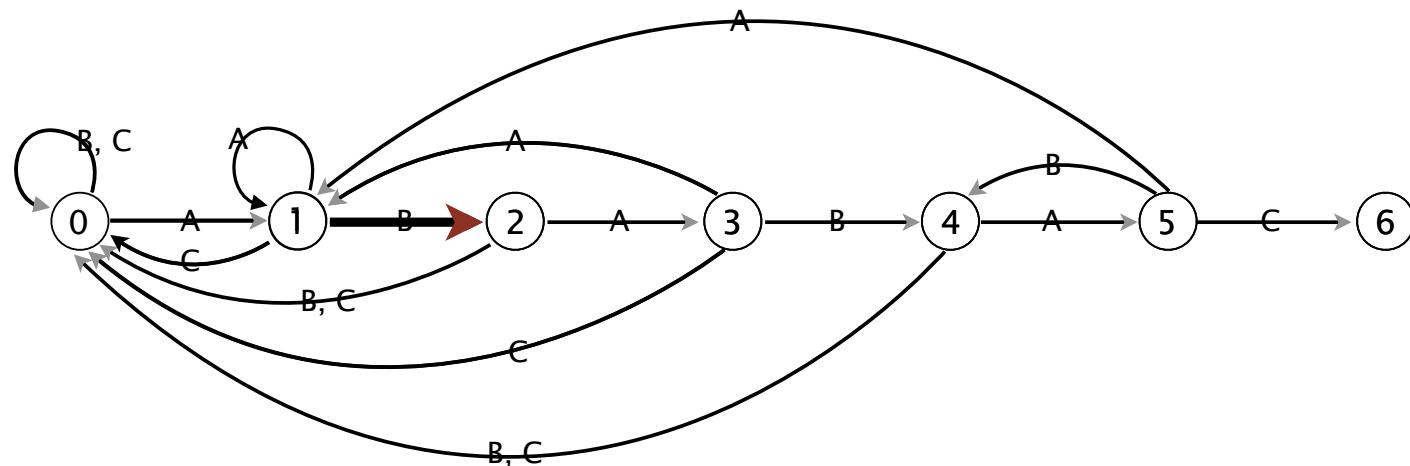
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	B	0	2	0	4	0	4
		0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A
↑

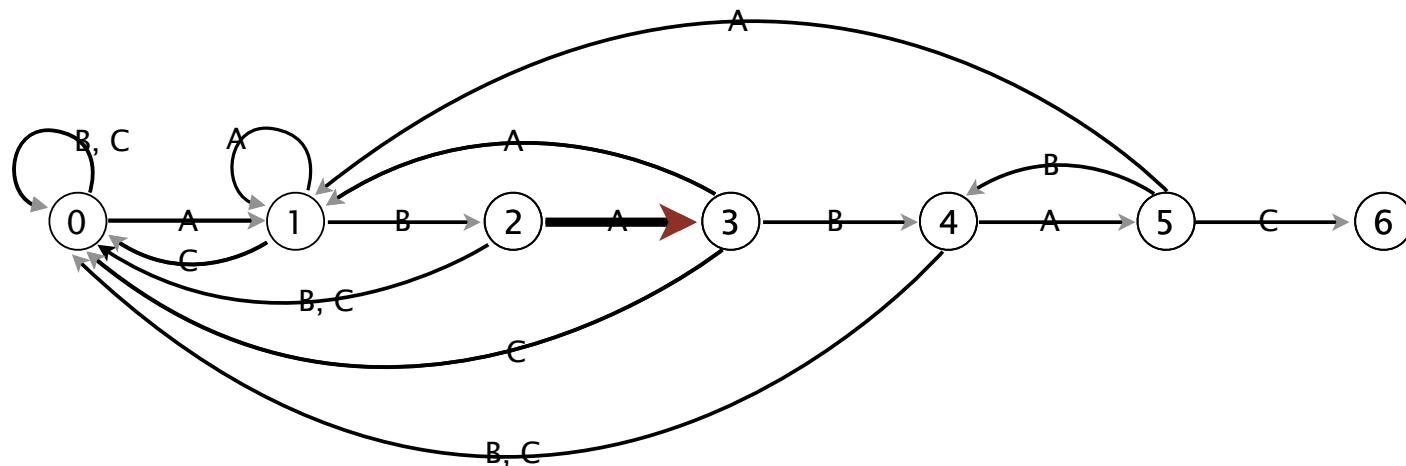
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	B	0	2	0	4	0	4
C		0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A
↑

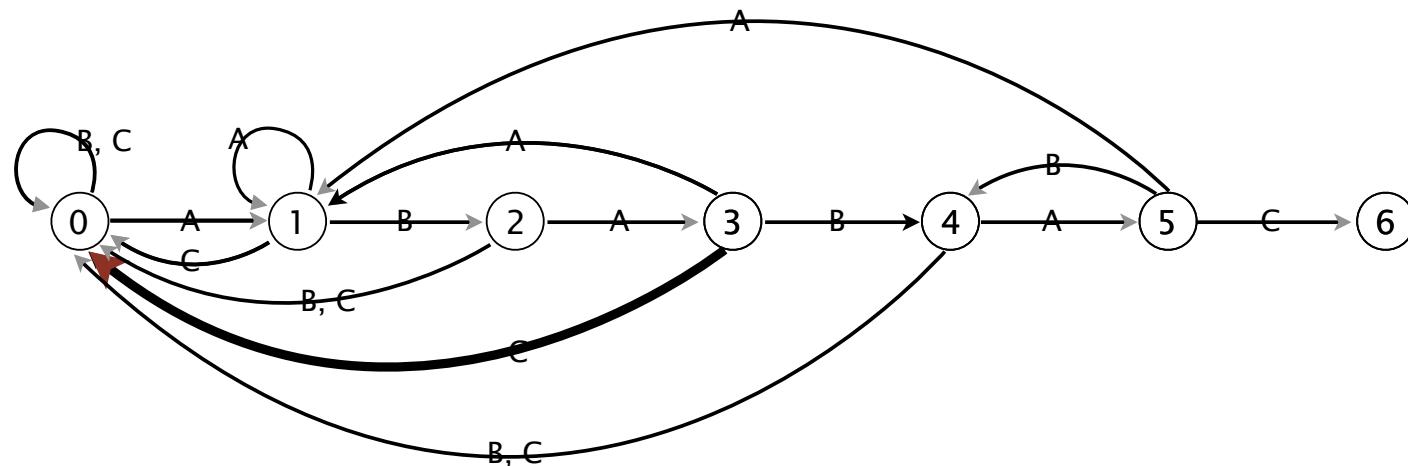
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	



DFA simulation

A A B A C A A B A B A C A A
↑

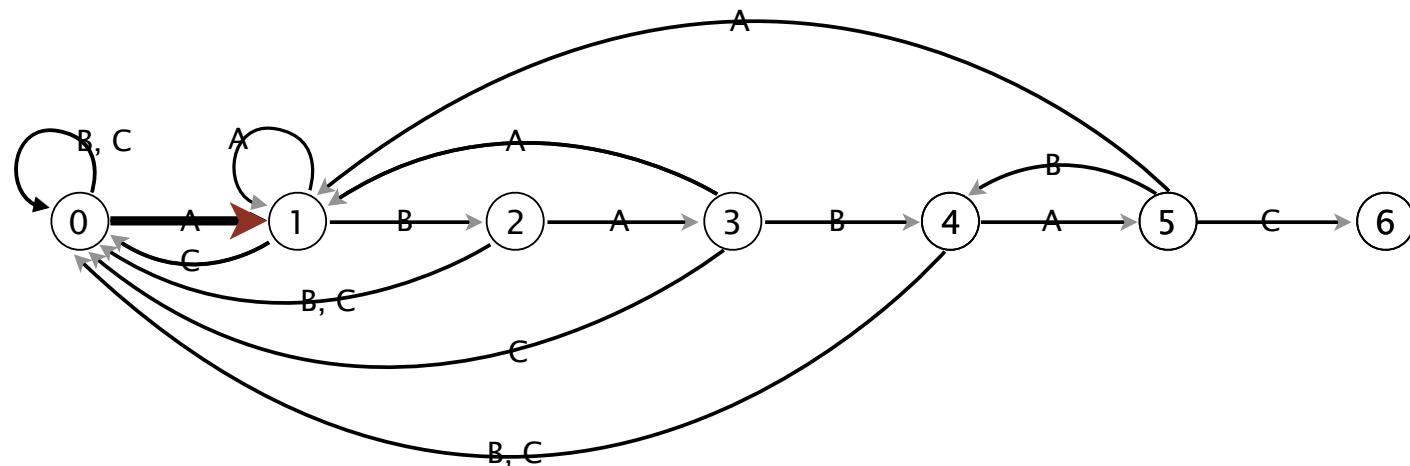
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	B	0	2	0	4	0	4
		0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A
↑

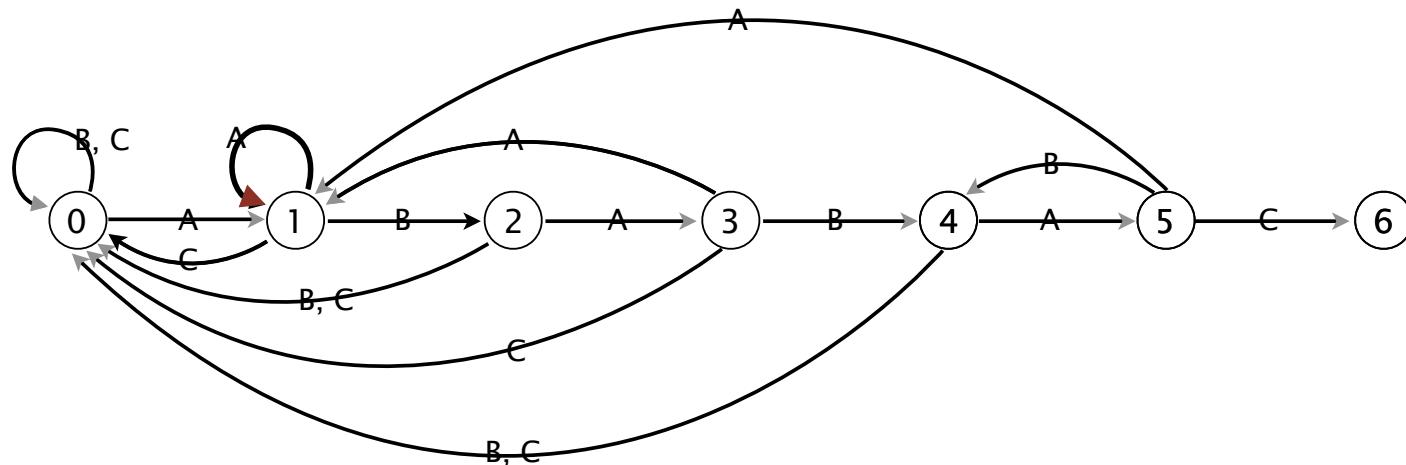
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	



DFA simulation

A A B A C A A B A B A C A A
↑

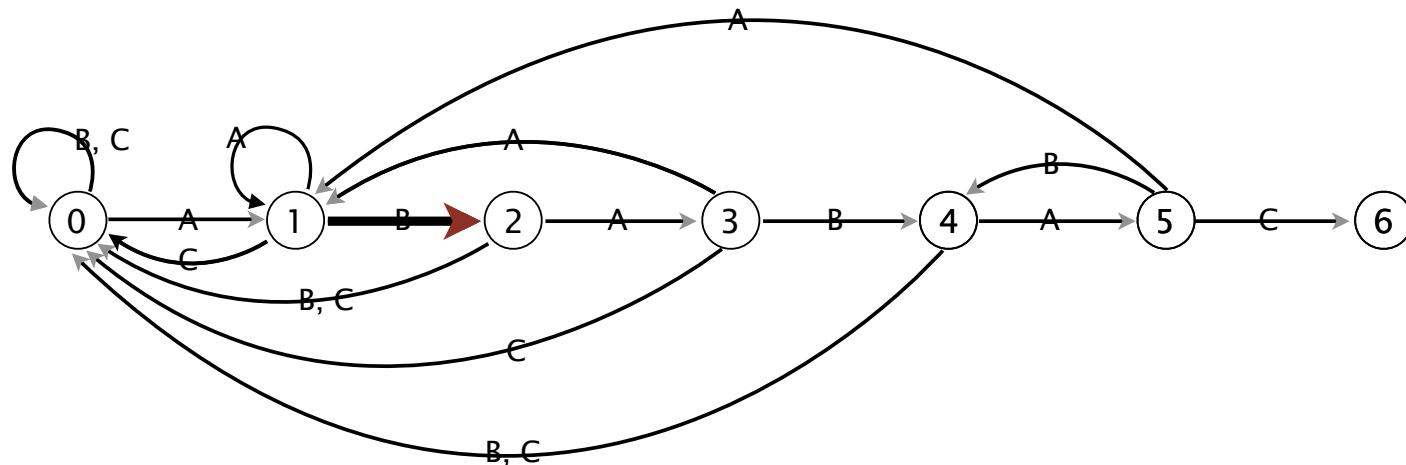
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	



DFA simulation

A A B A C A A B A B A C A A
↑

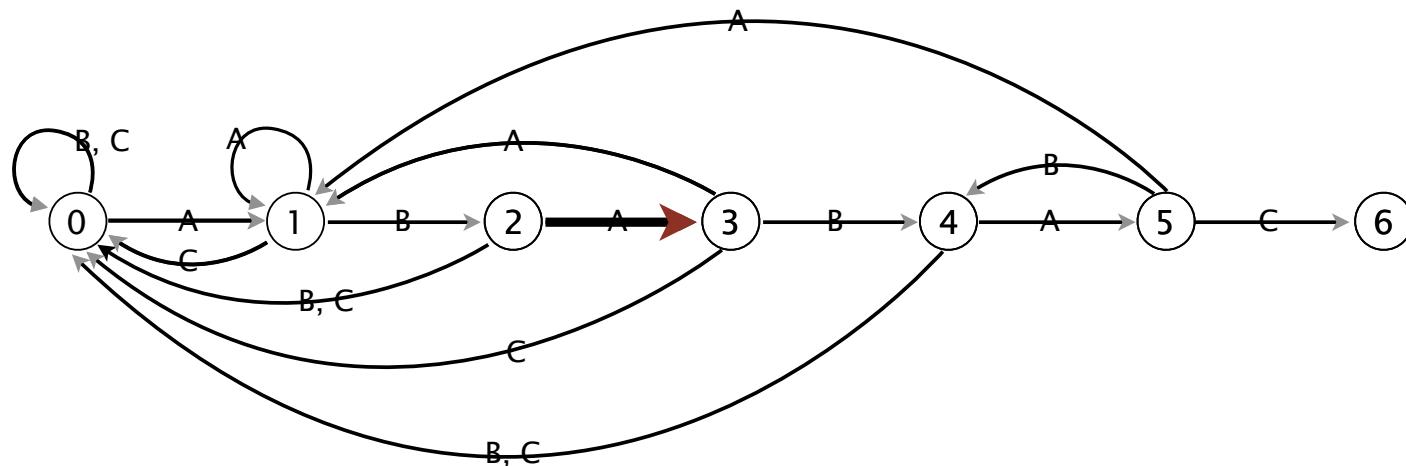
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	



DFA simulation

A A B A C A A B A B A C A A
↑

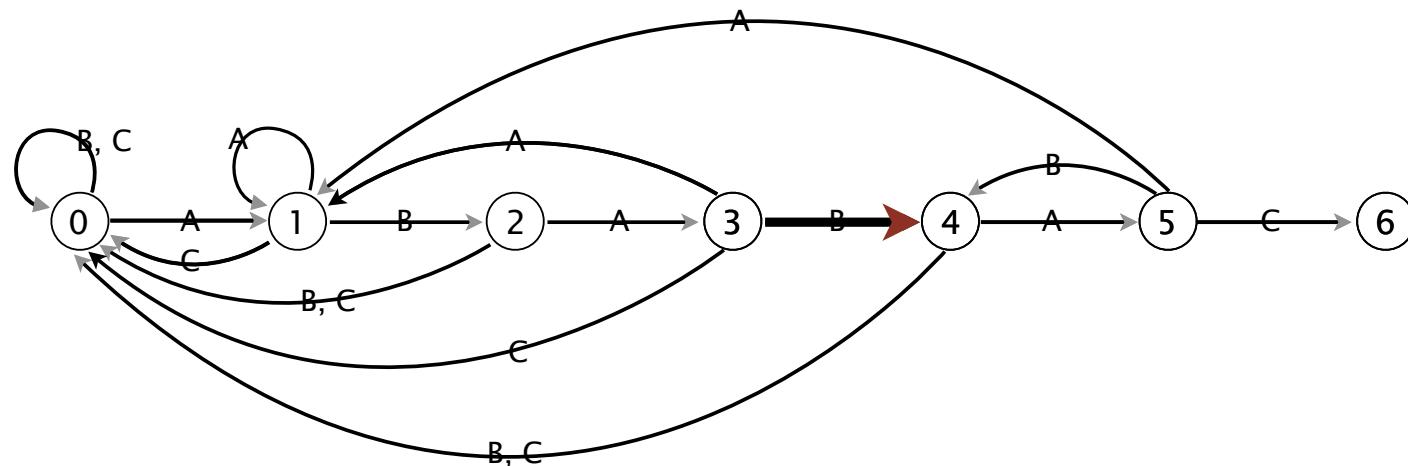
		0	1	2	3	4	5
pat.charAt(j)	dfa[][][j]	A	B	A	B	A	C
		1	1	3	1	5	1
A	0	2	0	4	0	4	
B	0	0	0	0	0	0	6
C	0	0	0	0	0	0	



DFA simulation

A A B A C A A B A B A C A A
↑

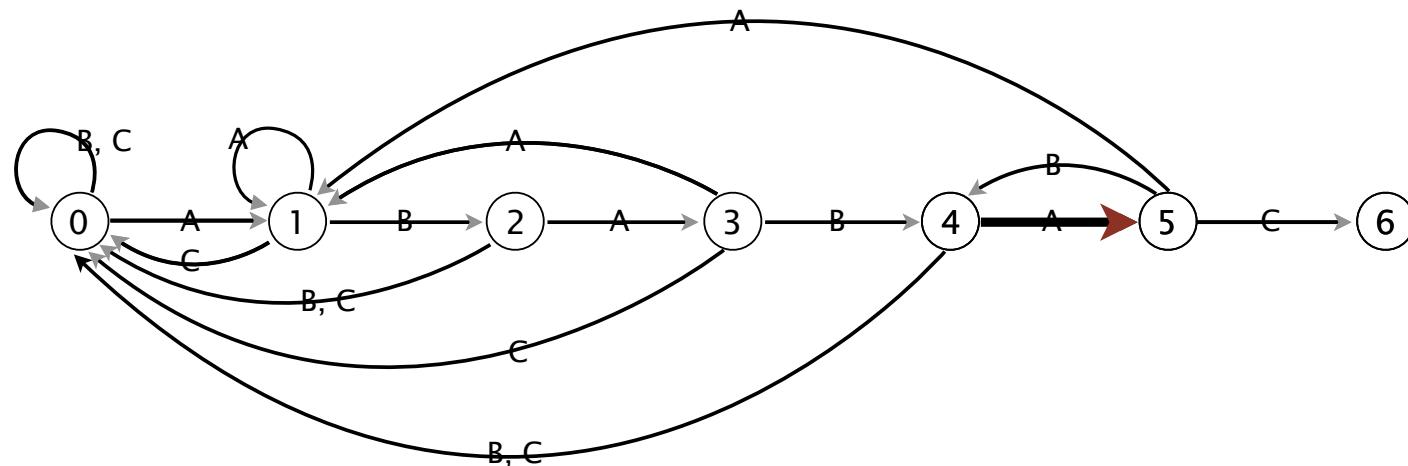
pat.charAt(j)	0	1	2	3	4	5
dfa[][][j]	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A
↑

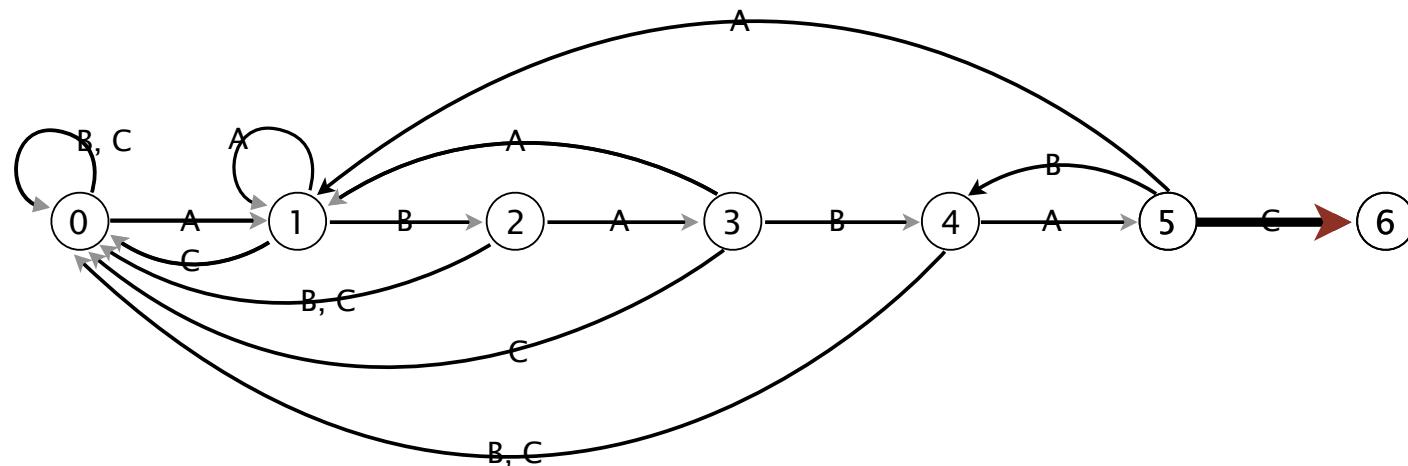
pat.charAt(j)	0	1	2	3	4	5
dfa[][][j]	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A
↑

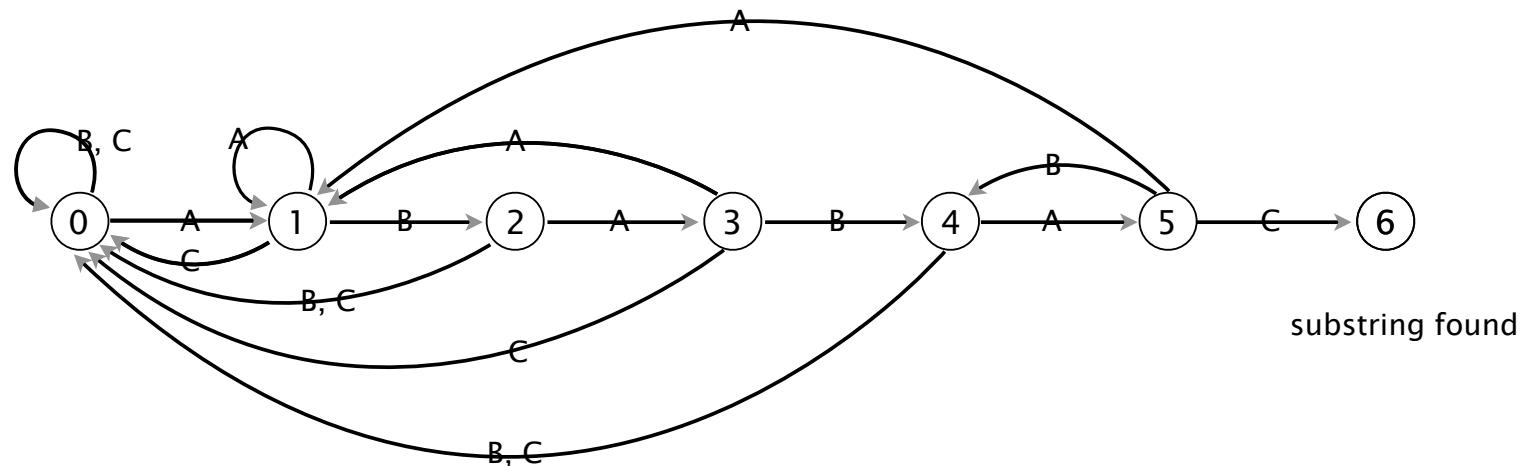
pat.charAt(j)	0	1	2	3	4	5
dfa[][][j]	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



DFA simulation

A A B A C A A B A B A C A A
↑

pat.charAt(j)	0	1	2	3	4	5
dfa[][][j]	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



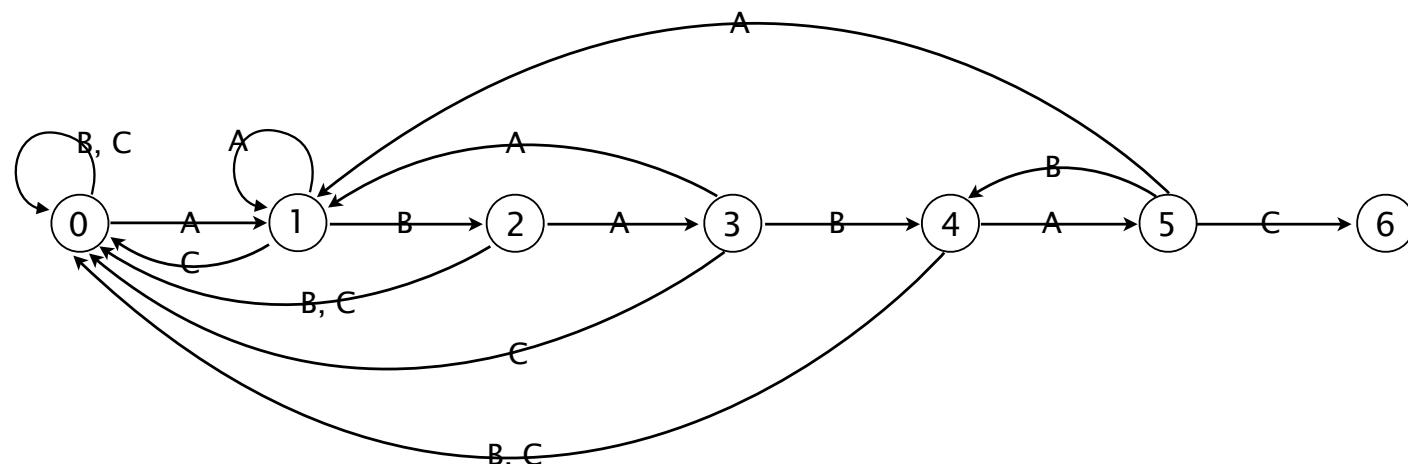
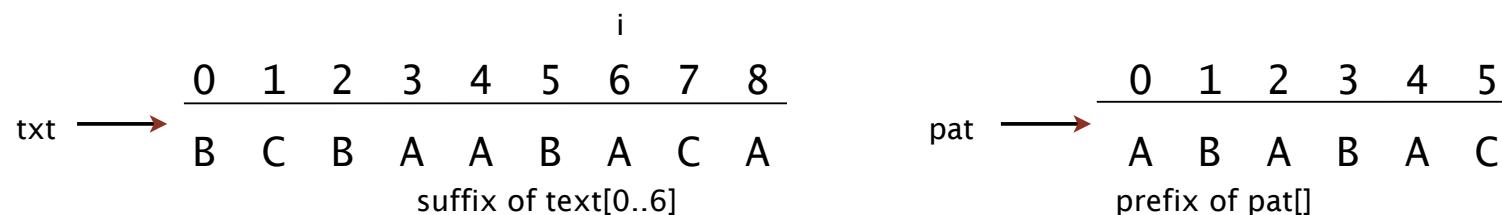
Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in `txt[i]`?

A. State = number of characters in pattern that have been matched.

length of longest prefix of pat[]
that is a suffix of txt[0..i]

Ex. DFA is in state 3 after reading in `txt[0..6]`.



Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];           ← no backup
    if (j == M) return i - M;
    else         return N;
}
```

Running time.

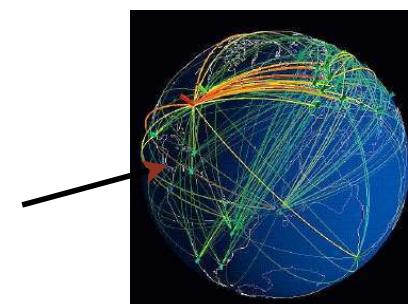
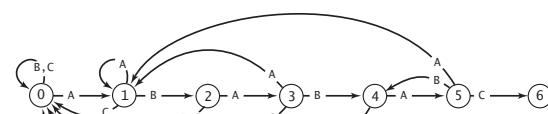
- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.
- Could use input stream.

```
public int search(In in)
{
    int i, j;
    for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
        j = dfa[in.readChar()][j];           ← no backup
    if (j == M) return i - M;
    else         return NOT_FOUND;
}
```



Knuth-Morris-Pratt construction

Include one state for each character in pattern (plus accept state).

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][]j	A	B				
	C					

Constructing the DFA for KMP substring search for A B A B A C

0

1

2

3

4

5

6

Knuth-Morris-Pratt construction

Match transition. If in state j and next char $c == \text{pat.charAt}(j)$, go to $j+1$.

↑
first j characters of pattern
have already been matched ↑
next char matches ↑
now first $j+1$ characters of
pattern have been matched

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1		3		5	
	B		2		4		
	C					6	

Constructing the DFA for KMP substring search for A B A B A C

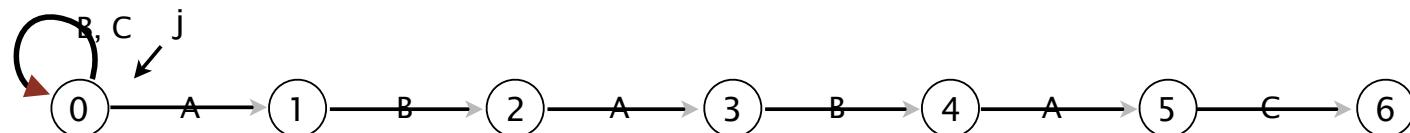


Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
<code>pat.charAt(j)</code>	A	A	B	A	B	A	C
	B	1		3		5	
<code>dfa[][][j]</code>	C	0	2		4		6

Constructing the DFA for KMP substring search for A B A B A C

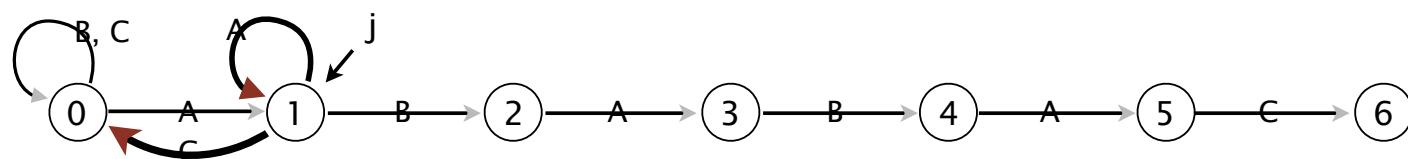


Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

$\text{pat.charAt}(j)$	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	1	1	3		5	
B	0	2		4		
C	0	0				6

Constructing the DFA for KMP substring search for A B A B A C

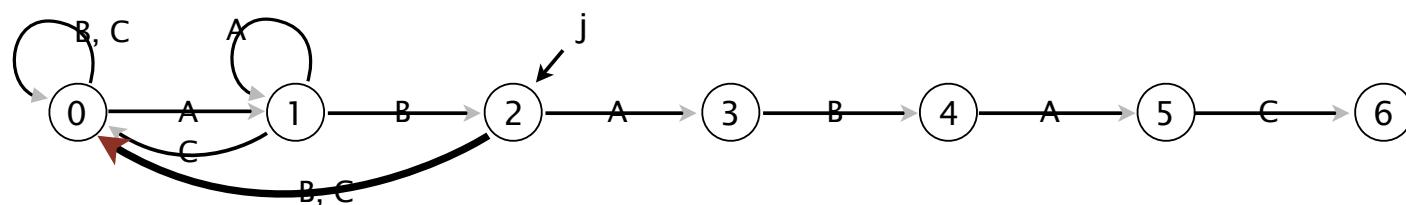


Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

$\text{pat.charAt}(j)$	0	1	2	3	4	5
A	A	B	A	B	A	C
A	1	1	3		5	
dfa[][][j]	B	0	2	0	4	
C	0	0	0			6

Constructing the DFA for KMP substring search for A B A B A C

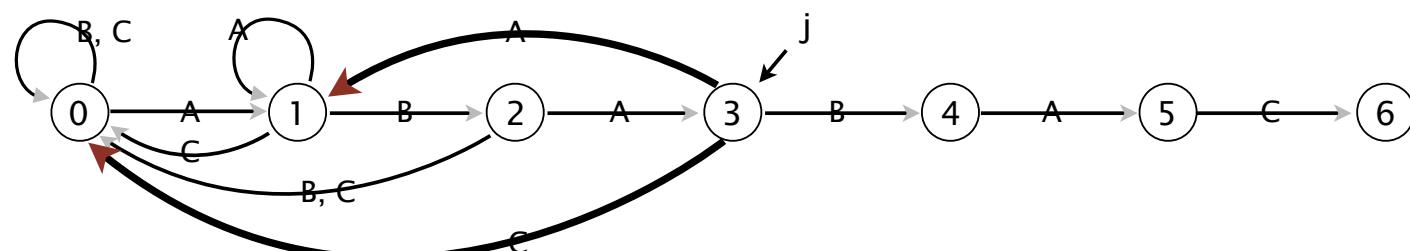


Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

$\text{pat.charAt}(j)$	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	1	1	3	1	5	
B	0	2	0	4		
C	0	0	0	0		6

Constructing the DFA for KMP substring search for A B A B A C

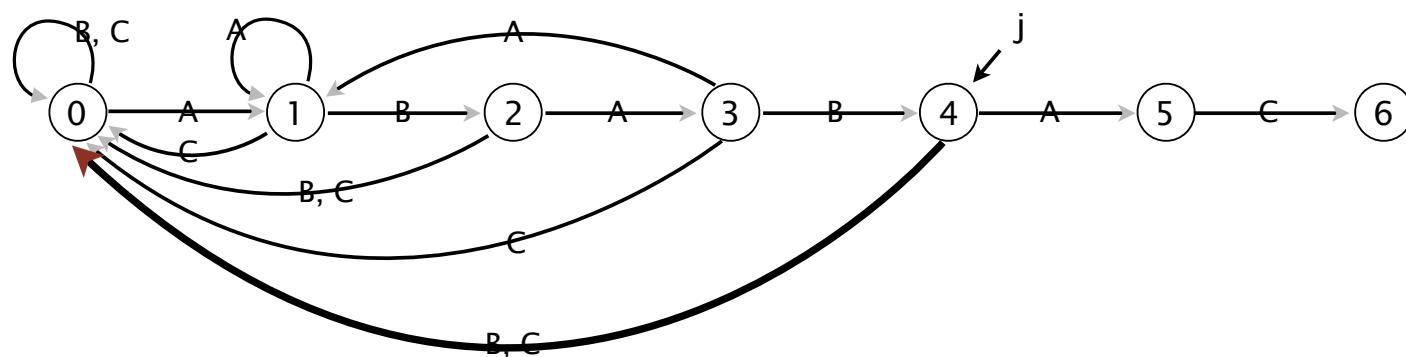


Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

		0	1	2	3	4	5
<code>pat.charAt(j)</code>	A	A	B	A	B	A	C
	B	1	1	3	1	5	
<code>dfa[][][j]</code>	C	0	2	0	4	0	
		0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C

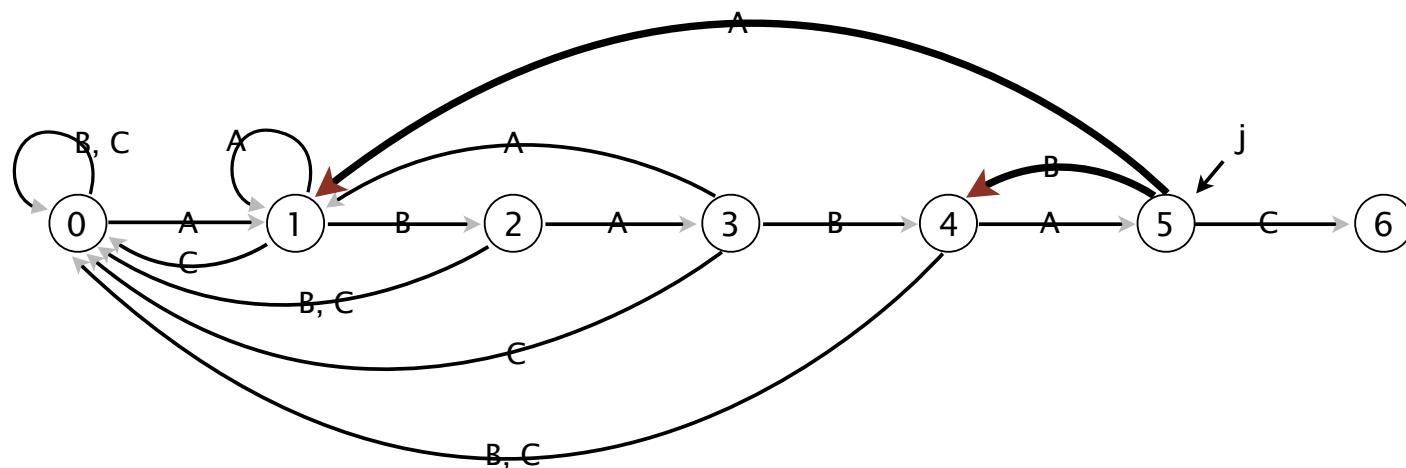


Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

$\text{pat.charAt}(j)$	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6

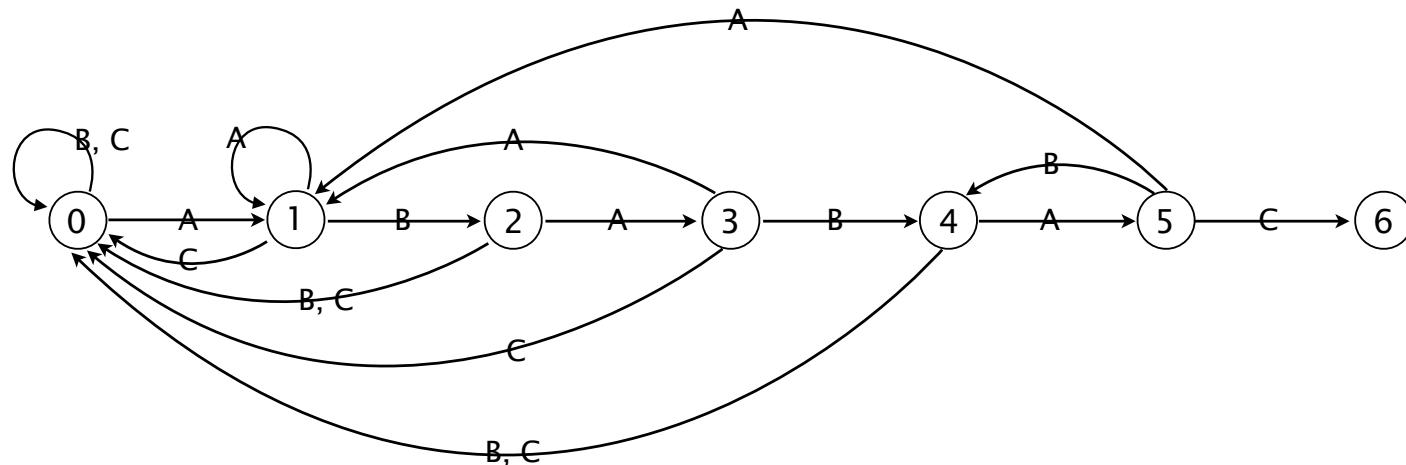
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt construction

pat.charAt(j)	0	1	2	3	4	5
dfa[][][j]	A	B	A	B	A	C
A	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).

<code>pat.charAt(j)</code>	0	1	2	3	4	5
<code>dfa[] [j]</code>	A	B	A	B	A	C
A						
B						
C						

0

1

2

3

4

5

6

How to build DFA from pattern?

Match transition. If in state j and next char $c == \text{pat.charAt}(j)$, go to $j+1$.

↑
first j characters of pattern
have already been matched ↑
next char matches ↑
now first $j+1$ characters of
pattern have been matched

		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	A	1		3		5	
	B		2		4		
							6



How to build DFA from pattern?

Mismatch transition. If in state j and next char $c \neq \text{pat.charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by c .

To compute $\text{dfa}[c][j]$: Simulate $\text{pat}[1..j-1]$ on DFA and take transition c .
Running time. Seems to require j steps.

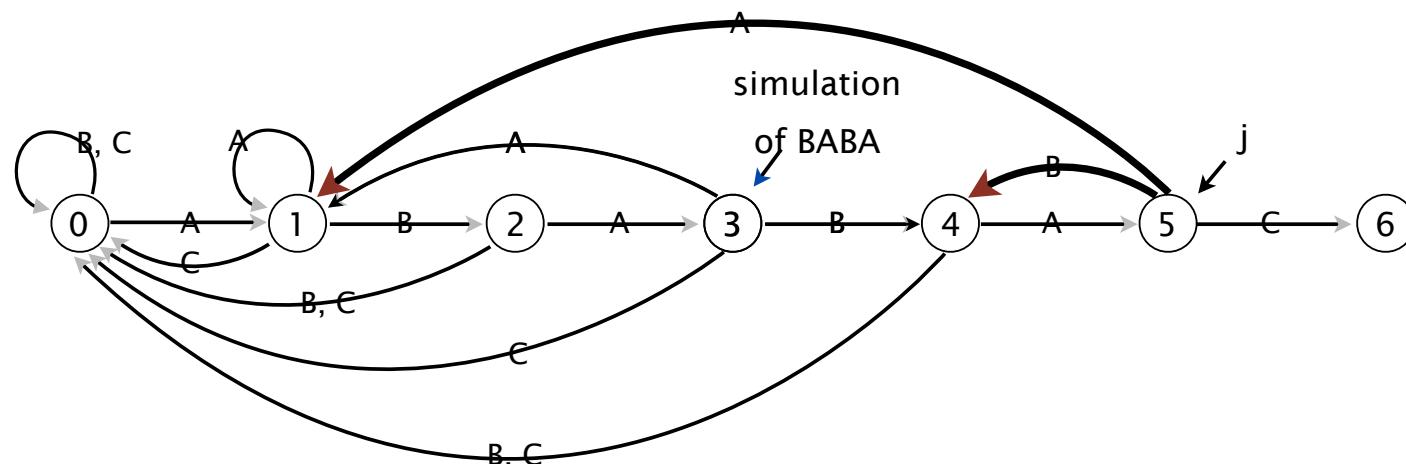
still under construction (!)

Ex. $\text{dfa}['A'][5] = 1; \quad \text{dfa}['B'][5] = 4$

simulate BABA;
take transition 'A'
 $= \text{dfa}['A'][3]$

simulate BABA;
take transition 'B'
 $= \text{dfa}['B'][3]$

j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C



How to build DFA from pattern?

Mismatch transition. If in state j and next char $c \neq \text{pat.charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by c .

To compute $\text{dfa}[c][j]$: Simulate $\text{pat}[1..j-1]$ on DFA and take transition c .
Running time. Takes only constant time if we maintain state X .

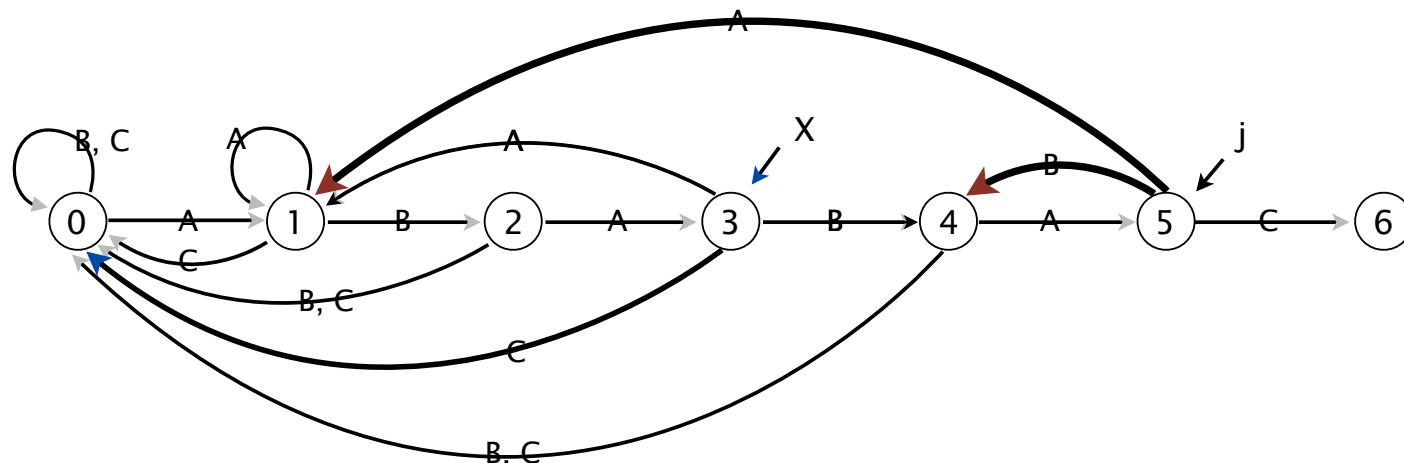
Ex. $\text{dfa}['A'][5] = 1$; $\text{dfa}['B'][5] = 4$; $\text{dfa}['C'][5] = 5$

from state X ,
take transition 'A'
 $= \text{dfa}['A'][X]$

from state X ,
take transition 'B'
 $= \text{dfa}['B'][X]$

$\cdot X' = 0$
from state X ,
take transition 'C'
 $= \text{dfa}['C'][X]$

0	1	2	3	4	5
A	B	A	B	A	C



Knuth-Morris-Pratt construction (in linear time)

Include one state for each character in pattern (plus accept state).

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][]j	A	B				
	C					

Constructing the DFA for KMP substring search for A B A B A C

0

1

2

3

4

5

6

Knuth-Morris-Pratt construction (in linear time)

Match transition. For each state j , $\text{dfa}[\text{pat.charAt}(j)][j] = j+1$.

↑
first j characters of pattern
have already been matched ↑
now first $j+1$ characters of
pattern have been matched

$\text{pat.charAt}(j)$	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	1		3		5	
C		2		4		6

Constructing the DFA for KMP substring search for A B A B A C



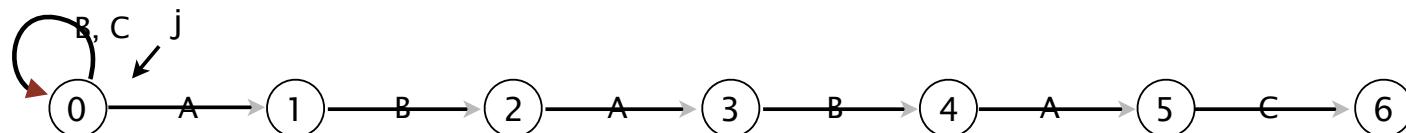
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For state o and char $c \neq \text{pat.charAt}(j)$,

set $\text{dfa}[c][0] = o$.

		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	A	1		3		5	
	B	0	2		4		
C	0						6

Constructing the DFA for KMP substring search for A B A B A C



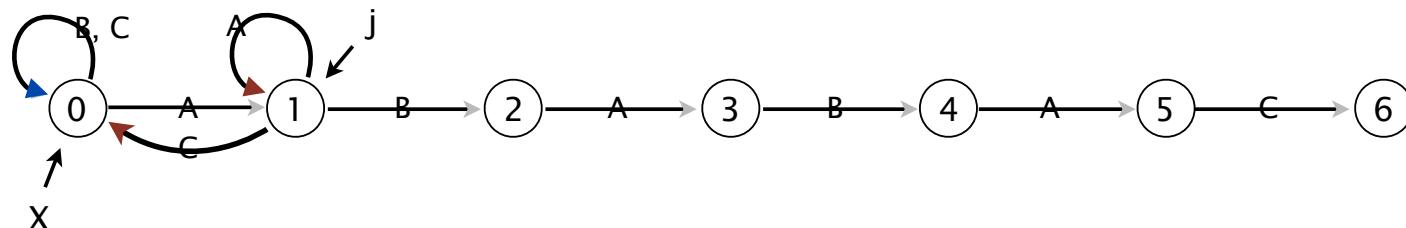
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

$X = \text{simulation of empty string}$

		0	1	2	3	4	5
		A	B	A	B	A	C
pat.charAt(j)	A	1		3		5	
	B	0	2		4		
	C	0				6	

Constructing the DFA for KMP substring search for A B A B A C



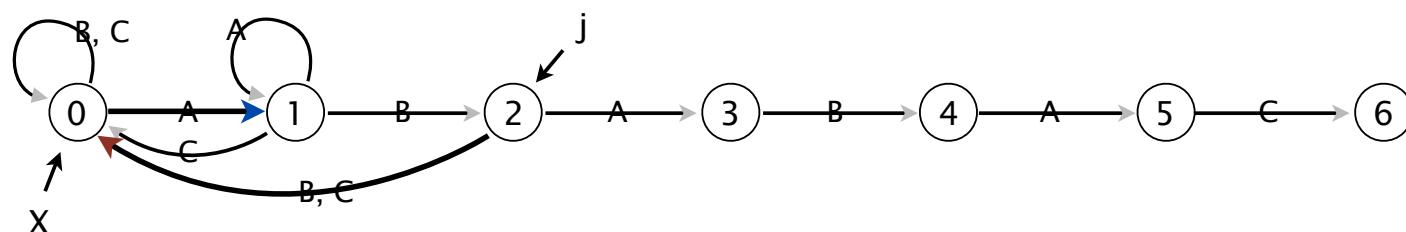
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

$X = \text{simulation of } B$

		0	1	2	3	4	5
		A	B	A	B	A	C
$\text{pat.charAt}(j)$	A	1	1	3		5	
	B	0	2		4		
	C	0	0			6	

Constructing the DFA for KMP substring search for A B A B A C



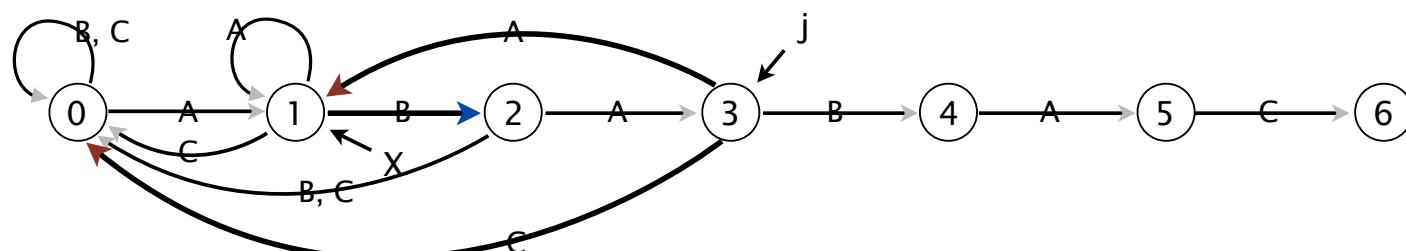
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

$X = \text{simulation of B A}$

		0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	A	B	A	B	A	C
	B	1	1	3		5	
$\text{dfa}[][j]$	C	0	2	0	4		6

Constructing the DFA for KMP substring search for A B A B A C



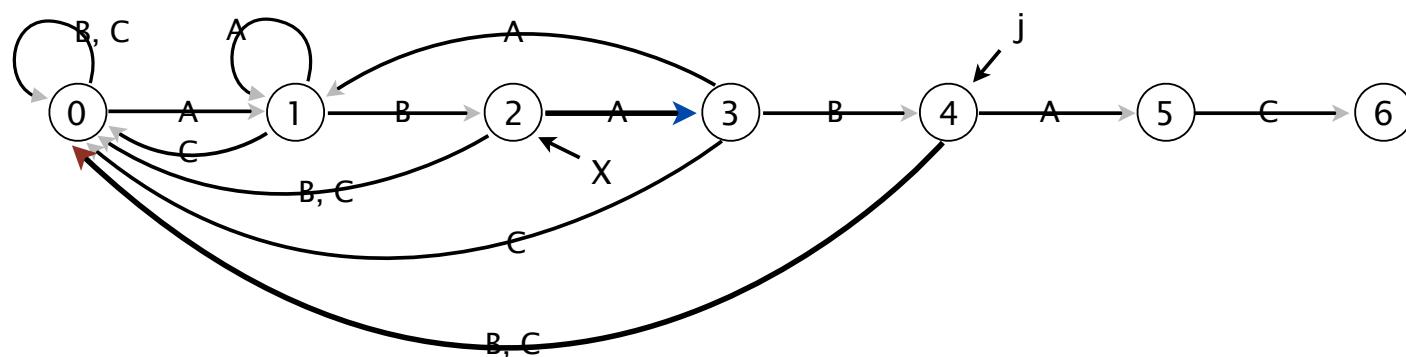
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

$X = \text{simulation of B A B}$

		0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	A	B	A	B	A	C
	B	1	1	3	1	5	
$\text{dfa}[][j]$	C	0	2	0	4		6

Constructing the DFA for KMP substring search for A B A B A C



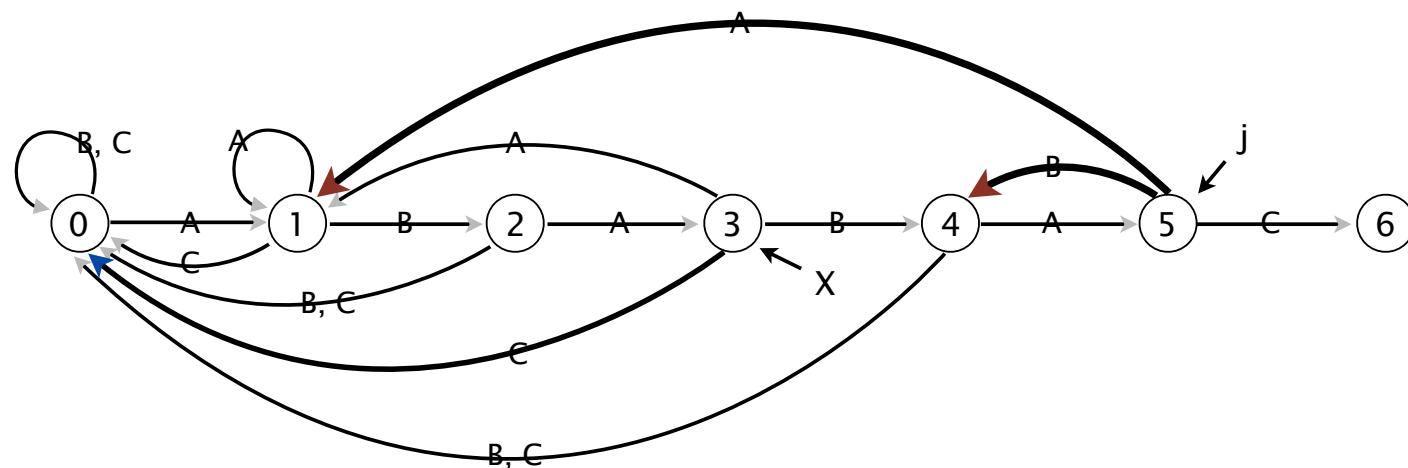
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

$X = \text{simulation of B A B A}$

		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	
dfa[][], j	C	0	2	0	4	0	

Constructing the DFA for KMP substring search for A B A B A C



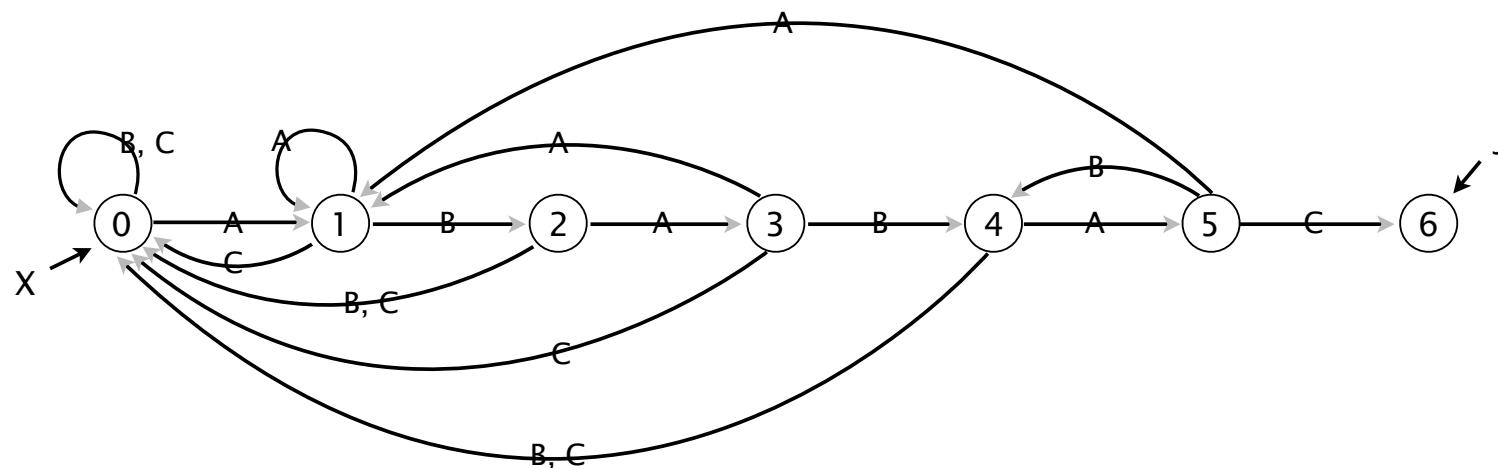
Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

$X = \text{simulation of B A B A C}$

	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C
A	1	1	3	1	5	1
$\text{dfa}[][j]$	B	0	2	0	4	0
C	0	0	0	0	0	6

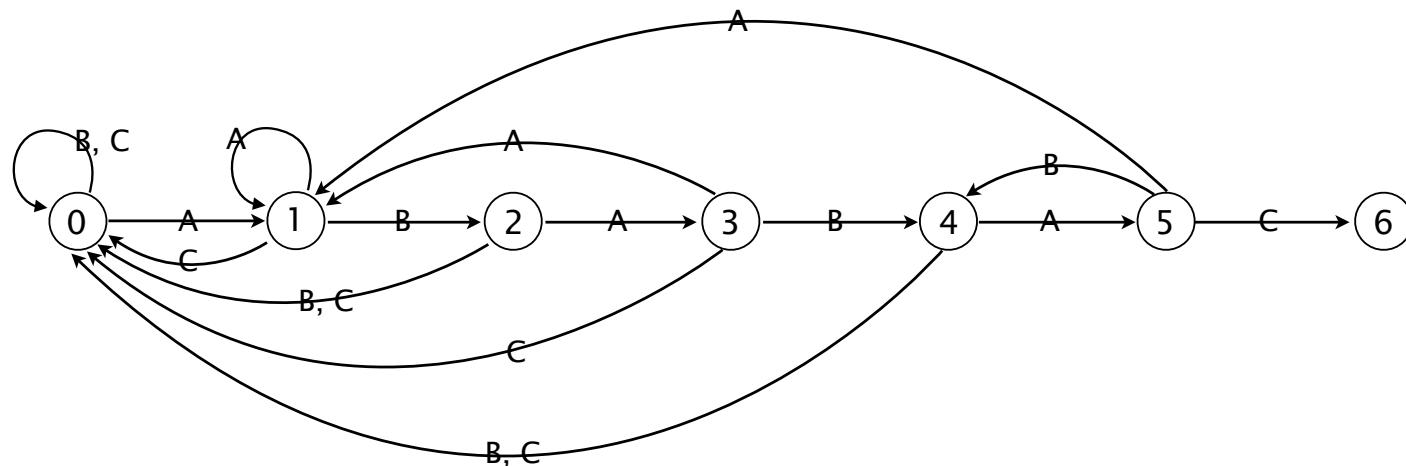
Constructing the DFA for KMP substring search for A B A B A C



Knuth-Morris-Pratt construction (in linear time)

pat.charAt(j)	0	1	2	3	4	5
dfa[] [j]	A	B	A	B	A	C
	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



Constructing the DFA for KMP substring search: Java implementation

For each state j :

- Copy $\text{dfa}[][\text{x}]$ to $\text{dfa}[][\text{j}]$ for mismatch case.
- Set $\text{dfa}[\text{pat.charAt(j)}][\text{j}]$ to $j+1$ for match case.
- Update x .

```
public KMP(String pat)
{
    this.pat = pat;
    M = pat.length();
    dfa = new int[R][M];
    dfa[pat.charAt(0)][0] = 1;
    for (int X = 0, j = 1; j < M; j++)
    {
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];      ← copy mismatch cases
        dfa[pat.charAt(j)][j] = j+1;   ← set match case
        X = dfa[pat.charAt(j)][X];    ← update restart state
    }
}
```

Running time. M character accesses (but space proportional to $R M$).

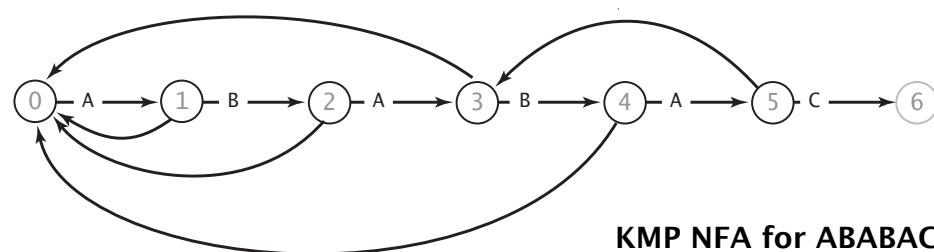
KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N .

Pf. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

Proposition. KMP constructs `dfa[][]` in time and space proportional to $R M$.

Larger alphabets. Improved version of KMP constructs `nfa[]` in time and space proportional to M .



Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
 - Knuth: inspired by esoteric theorem, discovered linear-time algorithm
 - Pratt: made running time independent of alphabet size
 - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

FAST PATTERN MATCHING IN STRINGS*

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.



Don Knuth



Jim Morris



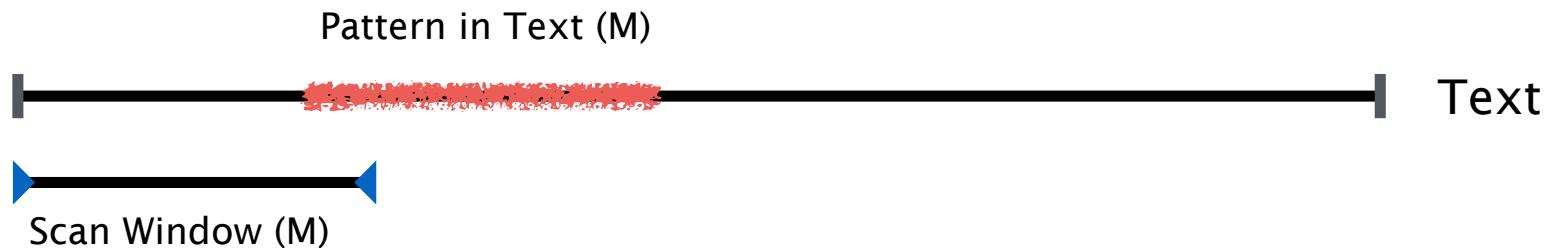
Vaughan Pratt

SUBSTRING SEARCH

- ▶ **Brute force**
- ▶ **Knuth-Morris-Pratt**
- ▶ **Boyer-Moore**
- ▶ **Rabin-Karp**

Boyer Moore Intuition

- Scan the text with a window of M chars (length of pattern)



- Case 1: Scan Window is exactly on top of the searched pattern



- Starting from one end check if all characters are equal. (We must check!)

- Case 2: Scan Window starts after the pattern starts.



Boyer Moore Intuition (2)

- Case 3: Scan Window starts before the pattern starts



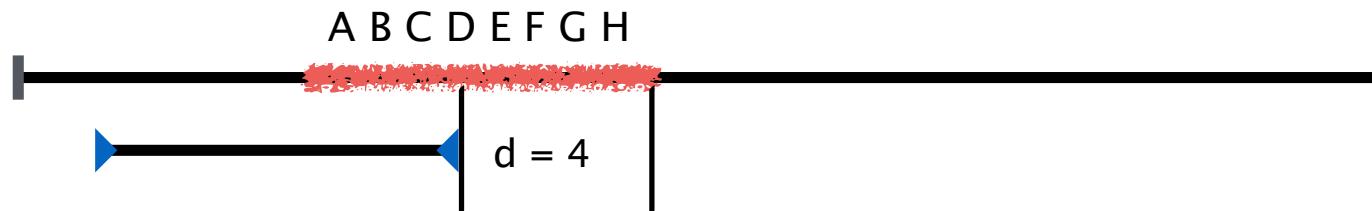
- Case 4: Independent



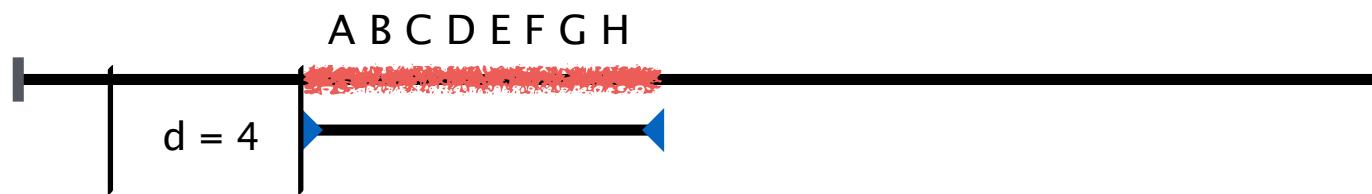
- In case 4, simply shift window M characters
- Avoid Case 2
- Convert Case 3 to Case 1, by shifting appropriately

Boyer Moore Intuition (3)

- If we can recognise the character in the scan window end-point, we can find how many characters to shift.



- So, for example D is the 4th character, we must shift window 4 characters so that they overlap.



Boyer Moore Intuition (4)

- A potential problem, the character in the text can repeat.
- For example, pattern = XXAXX and the text is

A X A X A X A X X A X X A X A X A X A X

- Solution: be conservative, choose the instance with the least Shift (so we cannot miss the others).

Boyer Moore Intuition (5)

- A X A X A X A X X A X X A X A X A X
Search: XXAXX
- So, for the example when it is A at the endpoint we must shift for 2 characters.
- text: AAAAX we have a mismatch in last A, now we must shift only once, so that we can check the configuration where the A we found moves to middle.
- text: AAYXX we have a mismatch in Y, now we must shift 3 times as we know that the last 2 characters are in pattern and they can be repeating in the first 3 characters.

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as M text chars when finding one not in the pattern.
 - First we check the character in index `pattern.length()-1`
 - It is N which is not E, so we know that first 5 characters is not a match. Shift text 5 characters
 - S != E so shift 5, E == E so we can check for the `pattern.length()-2`, L!=N, skip 4.

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
		F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A	
		<i>text</i> →																								
0	5	N	E	E	D	L	E	← <i>pattern</i>																		
5	5						N	E	E	D	L	E														
11	4											N	E	E	D	L	E									
15	0												N	E	E	D	L	E								

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case I. Mismatch character not in pattern.

	i									
before										
txt T L E									
pat	N E E D L E									
after T L E									
txt										
pat	N E E D L E									

mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.

before	i	↓															
txt	N	L	E	
pat							N	E	E	D	L	E					

after	i	↓															
txt	N	L	E	
pat							N	E	E	D	L	E					

mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

before ↓
i

txt E L E
pat N E E D L E

aligned with rightmost E? ↓
i

txt E L E
pat N E E D L E

mismatch character 'E' in pattern: align text 'E' with rightmost pattern 'E'?

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

before	i	↓															
txt	E	L	E	
pat		N	E	E	D	L	E										
after	i	↓															
txt	E	L	E	
pat		N	E	E	D	L	E										

mismatch character 'E' in pattern: increment i by 1

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Precompute index of rightmost occurrence of character c in pattern
(-1 if character not in pattern).

```
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < M; j++)
    right[pat.charAt(j)] = j;
```

<u>c</u>	N	E	E	D	L	E	<u>right[c]</u>
0	-1	-1	-1	-1	-1	-1	-1
A	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3
E	-1	-1	1	2	2	5	5
...							-1
L	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0
...							-1

Boyer-Moore skip table computation

Boyer-Moore: Java implementation

```
public int search(String txt)
{
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N-M; i += skip)
    {
        skip = 0;
        for (int j = M-1; j >= 0; j--)
        {
            if (pat.charAt(j) != txt.charAt(i+j))           ← compute skip value
            {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
            in case other term is nonpositive
        }
        if (skip == 0) return i;                         ← match
    }
    return N;
}
```

Another Example

SEARCH FOR: XXXX

A X A X A X A X X X A X A X X X X A A A
A X A X A X A X X X A X A X X X X A A A

If the window scan points to an unrecognised character, we can skip past that character. For this example, for the initial step we first match X at the end, when check for previous character (A) which is not in the string we skip 3 steps. The X at the end, we matched can still be the first character of the pattern, so we do not skip that.

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim N/M$ character compares to search for a pattern of length M in a text of length N . \nearrow sublinear!

Worst-case. Can be as bad as $\sim MN$.

i	skip	0	1	2	3	4	5	6	7	8	9
	txt →	B	B	B	B	B	B	B	B	B	B
0	0	A	B	B	B	B	B	B	B	B	B
1	1		A	B	B	B	B				
2	1			A	B	B	B	B			
3	1				A	B	B	B	B		
4	1					A	B	B	B	B	
5	1						A	B	B	B	B

Boyer-Moore variant. Can improve worst case to $\sim 3N$ by adding a KMP-like rule to guard against repetitive patterns.

SUBSTRING SEARCH

- ▶ **Brute force**
- ▶ **Knuth-Morris-Pratt**
- ▶ **Boyer-Moore**
- ▶ **Rabin-Karp**

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of pattern characters 0 to $M - 1$.
- For each i , compute a hash of text characters i to $M + i - 1$.
- If pattern hash = text substring hash, check for a match.

pat.charAt(i)															
i	0	1	2	3	4										
	2	6	5	3	5										
	2	6	5	3	5	%	997	=	613						
txt.charAt(i)															
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	
0	3	1	4	1	5	%	997	=	508						
1		1	4	1	5	9	%	997	=	201					
2			4	1	5	9	2	%	997	=	715				
3				1	5	9	2	6	%	997	=	971			
4					5	9	2	6	5	%	997	=	442		
5						9	2	6	5	3	%	997	=	929	
6 ← return i = 6							2	6	5	3	5	%	997	=	613
															match ↘

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M -digit, base- R integer, modulo Q .

Horner's method. Linear-time method to evaluate degree- M polynomial.

<code>pat.charAt()</code>					
i	0	1	2	3	4
	2	6	5	3	5
0	2	$\% 997 = 2$			
1	2	6	$\% 997 = (2*10 + 6) \% 997 = 26$		
2	2	6	5	$\% 997 = (26*10 + 5) \% 997 = 265$	
3	2	6	5	3	$\% 997 = (265*10 + 3) \% 997 = 659$
4	2	6	5	3	5

```
// Compute hash for M-digit key
private long hash(String key, int M)
{
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + key.charAt(j)) % Q;
    return h;
}
```

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i .

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

Key property. Can update hash function in constant time!

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

↑ ↑ ↑ ↑
current subtract multiply add new
value leading digit by radix trailing digit (can precompute R^{M-2})

i	...	2	3	4	5	6	7	...
<i>current value</i>	1	4	1	5	9	2	6	5
<i>new value</i>		4	1	5	9	2	6	5

$$\begin{array}{r} 4 & 1 & 5 & 9 & 2 & \text{current value} \\ - & 4 & 0 & 0 & 0 & 0 \\ & 1 & 5 & 9 & 2 & \text{subtract leading digit} \\ & * & 1 & 0 & & \text{multiply by radix} \\ 1 & 5 & 9 & 2 & 0 & \\ & & & + & 6 & \text{add new trailing digit} \\ 1 & 5 & 9 & 2 & 6 & \text{new value} \end{array}$$

Rabin-Karp substring search example

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
0	3	1	%	997	=	3											
1	3	1	%	997	=	(3*10 + 1) % 997 = 31	Q										
2	3	1	4	%	997	=	(31*10 + 4) % 997 = 314										
3	3	1	4	1	%	997	=	(314*10 + 1) % 997 = 150									
4	3	1	4	1	5	%	997	=	(150*10 + 5) % 997 = 508	RM	R						
5	1	4	1	5	9	%	997	=	((508 + 3*(997 - 30))*10 + 9) % 997 = 201								
6		4	1	5	9	2	%	997	=	((201 + 1*(997 - 30))*10 + 2) % 997 = 715							
7			1	5	9	2	6	%	997	=	((715 + 4*(997 - 30))*10 + 6) % 997 = 971						
8				5	9	2	6	5	%	997	=	((971 + 1*(997 - 30))*10 + 5) % 997 = 442	match				
9					9	2	6	5	3	%	997	=	((442 + 5*(997 - 30))*10 + 3) % 997 = 929				
10	←	return i-M+1 = 6		2	6	5	3	5	%	997	=	((929 + 9*(997 - 30))*10 + 5) % 997 = 613					

Rabin-Karp: Java implementation

```
public class RabinKarp
{
    private long patHash;          // pattern hash value
    private int M;                 // pattern length
    private long Q;                // modulus
    private int R;                 // radix
    private long RM;               //  $R^{M-1} \bmod Q$ 

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = longRandomPrime();           ← a large prime
                                         (but avoid overflow)

        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat, M);
    }

    private long hash(String key, int M)
    { /* as before */ }

    public int search(String txt)
    { /* see next slide */ }
}
```

a large prime
(but avoid overflow)

← precompute $R^{M-1} \bmod Q$

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();                                check for hash collision
    int txtHash = hash(txt, M);                         using rolling hash function
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

Las Vegas version. Check for substring match if hash match;
continue search if false collision.

Rabin-Karp analysis

Theory. If Q is a sufficiently large random prime (about $M N^2$), then the probability of a false collision is about $1 / N$.

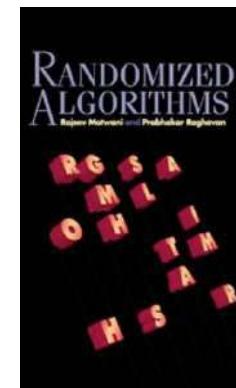
Practice. Choose Q to be a large prime (but not so large as to cause overflow). Under reasonable assumptions, probability of a collision is about $1 / Q$.

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $M N$).



Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Substring search cost summary

Cost of searching for an M -character pattern in an N -character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1 N$	yes	yes	1
Knuth-Morris-Pratt	<i>full DFA</i> (Algorithm 5.6)	$2N$	$1.1 N$	no	yes	MR
	<i>mismatch transitions only</i>	$3N$	$1.1 N$	no	yes	M
Boyer-Moore	<i>full algorithm</i>	$3N$	N / M	yes	yes	R
	<i>mismatched char heuristic only</i> (Algorithm 5.7)	MN	N / M	yes	yes	R
Rabin-Karp [†]	<i>Monte Carlo</i> (Algorithm 5.8)	$7N$	$7N$	no	yes^{\dagger}	1
	<i>Las Vegas</i>	$7N^{\dagger}$	$7N$	yes	yes	1

[†] probabilistic guarantee, with uniform hash function

BBM 202 - ALGORITHMS



HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

DATA COMPRESSION

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

DATA COMPRESSION

- ▶ Run-length coding
- ▶ Huffman compression

Data compression

Compression reduces the size of a file:

- To save **space** when storing it.
- To save **time** when transmitting it.
- Most files have lots of redundancy.

Who needs compression?

- Moore's law: # transistors on a chip doubles every 18-24 months.
- Parkinson's law: data expands to fill space available.
- Text, images, sound, video, ...

“Everyday, we create 2.5 quintillion bytes of data—so much that 90% of the data in the world today has been created in the last two years alone. ” — IBM report on big data (2011)

Basic concepts ancient (1950s), best technology recently developed.

Applications

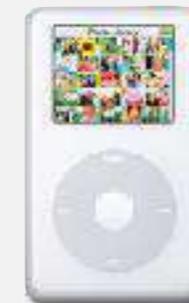
Generic file compression.

- Files: GZIP, BZIP, 7z.
- Archivers: PKZIP.
- File systems: NTFS, HFS+, ZFS.



Multimedia.

- Images: GIF, JPEG.
- Sound: MP3.
- Video: MPEG, DivX™, HDTV.



Communication.

- ITU-T T4 Group 3 Fax.
- V.42bis modem.
- Skype.



Databases. Google, Facebook,



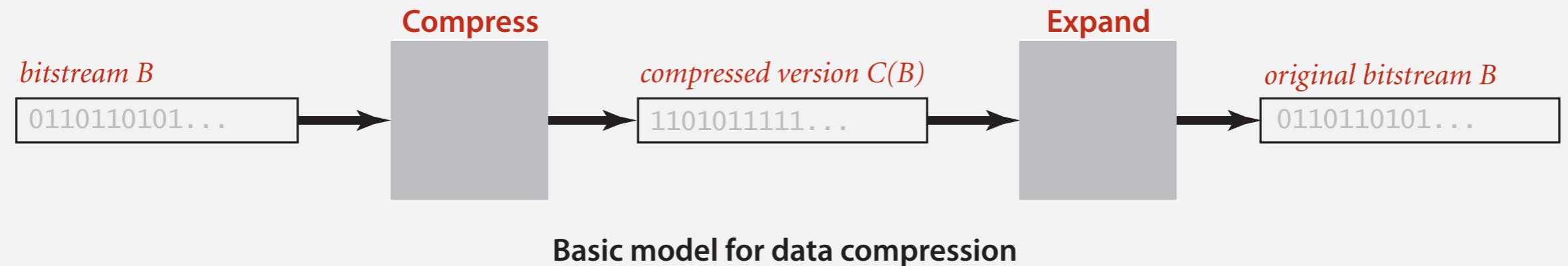
Lossless compression and expansion

Message. Binary data B we want to compress.

uses fewer bits (you hope)

Compress. Generates a "compressed" representation $C(B)$.

Expand. Reconstructs original bitstream B .



Compression ratio. Bits in $C(B)$ / bits in B .

Ex. 50-75% or better compression ratio for natural language.

Food for thought

Data compression has been omnipresent since antiquity:

- Number systems.
- Natural languages.
- Mathematical notation.

|||| |

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$$

has played a central role in communications technology,

- Grade 2 Braille.
- Morse code.
- Telephone system.

b	r	a	i			e
•○ ●○ ○○	●○ ●○ ○○	●○ ○○ ○○	○● ○○ ○○	●○ ●○ ○○	●○ ●○ ○○	●○ ○○ ○○
but	rather	a		like	like	every

and is part of modern life.

- MP3.
- MPEG.



Q. What role will it play in the future?

Data representation: genomic code

Genome. String over the alphabet { A, C, T, G }.

Goal. Encode an N -character genome: ATAGATGCATAG...

Standard ASCII encoding.

- 8 bits per char.
- $8N$ bits.

char	hex	binary
A	41	01000001
C	43	01000011
T	54	01010100
G	47	01000111

Two-bit encoding.

- 2 bits per char.
- $2N$ bits.

char	binary
A	00
C	01
T	10
G	11

Fixed-length code. k -bit code supports alphabet of size 2^k .

Amazing but true. Initial genomic databases in 1990s used ASCII.

Reading and writing binary data

Binary standard input and standard output. Libraries to read and write bits from standard input and to standard output.

```
public class BinaryStdIn
```

boolean readBoolean() *read 1 bit of data and return as a boolean value*

char readChar() *read 8 bits of data and return as a char value*

char readChar(int r) *read r bits of data and return as a char value*

[similar methods for byte (8 bits); short (16 bits); int (32 bits); long and double (64 bits)]

boolean isEmpty() *is the bitstream empty?*

void close() *close the bitstream*

```
public class BinaryStdOut
```

void write(boolean b) *write the specified bit*

void write(char c) *write the specified 8-bit char*

void write(char c, int r) *write the r least significant bits of the specified char*

[similar methods for byte (8 bits); short (16 bits); int (32 bits); long and double (64 bits)]

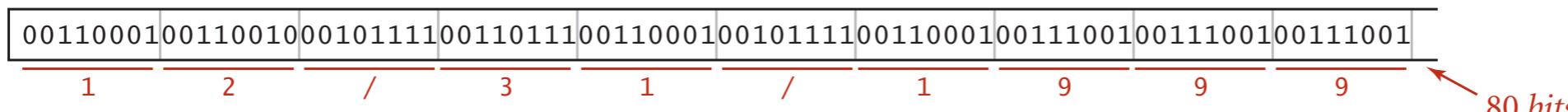
void close() *close the bitstream*

Writing binary data

Date representation. Three different ways to represent 12/31/1999.

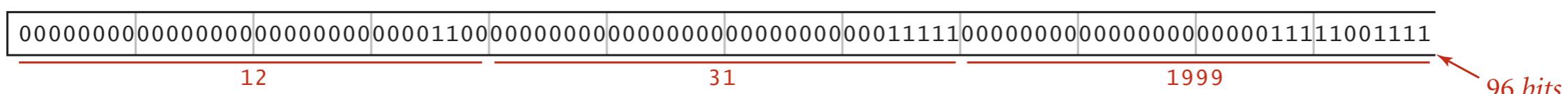
A character stream (StdOut)

```
StdOut.print(month + "/" + day + "/" + year)
```



Three ints (BinaryStdOut)

```
BinaryStdOut.write(month);  
BinaryStdOut.write(day);  
BinaryStdOut.write(year);
```



A 4-bit field, a 5-bit field, and a 12-bit field (BinaryStdOut)

```
BinaryStdOut.write(month, 4);
BinaryStdOut.write(day, 5);
BinaryStdOut.write(year, 12);
```



Binary dumps

Q. How to examine the contents of a bitstream?

Standard character stream

```
% more abra.txt  
ABRACADABRA!
```

Bitstream represented as 0 and 1 characters

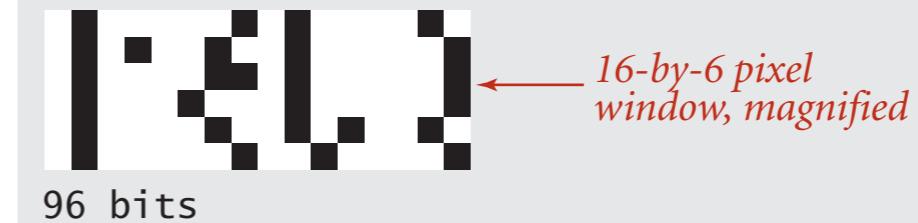
```
% java BinaryDump 16 < abra.txt  
0100000101000010  
0101001001000001  
0100001101000001  
0100010001000001  
0100001001010010  
0100000100100001  
96 bits
```

Bitstream represented with hex digits

```
% java HexDump 4 < abra.txt  
41 42 52 41  
43 41 44 41  
42 52 41 21  
12 bytes
```

Bitstream represented as pixels in a Picture

```
% java PictureDump 16 6 < abra.txt
```



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

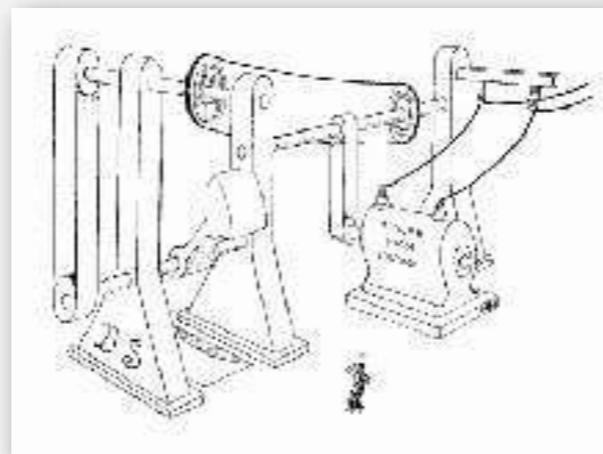
Universal data compression

US Patent 5,533,051 on "Methods for Data Compression", which is capable of compression **all** files.

[Slashdot](#) reports of the Zero Space Tuner™ and BinaryAccelerator™.

*"ZeoSync has announced a breakthrough in data compression that allows for 100:1 lossless compression of **random** data. If this is true, our bandwidth problems just got a lot smaller.... "*

Physical analog. Perpetual motion machines.



Gravity engine by Bob Schadewald

Universal data compression

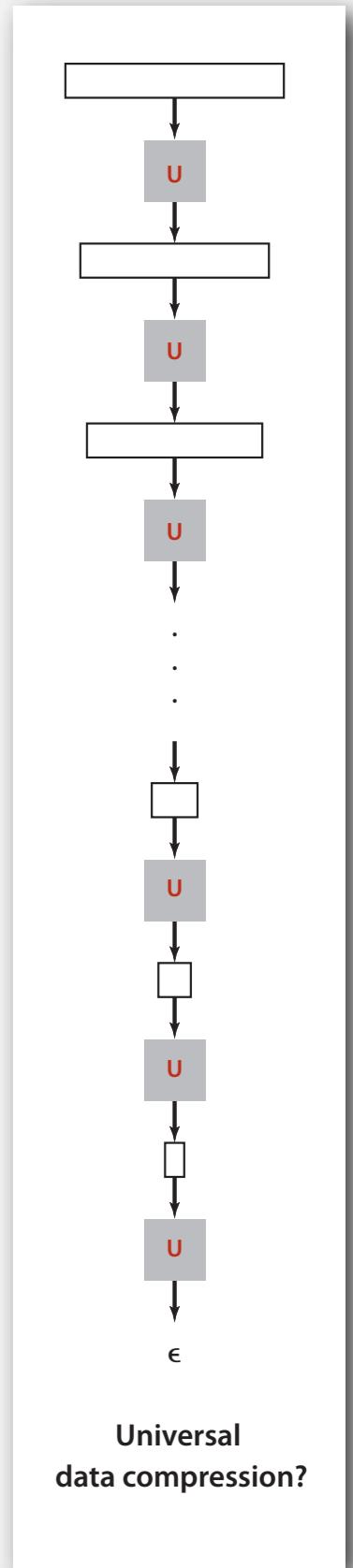
Proposition. No algorithm can compress every bitstring.

Pf 1. [by contradiction]

- Suppose you have a universal data compression algorithm U that can compress every bitstream.
- Given bitstring B_0 , compress it to get smaller bitstring B_1 .
- Compress B_1 to get a smaller bitstring B_2 .
- Continue until reaching bitstring of size 0.
- Implication: all bitstrings can be compressed to 0 bits!

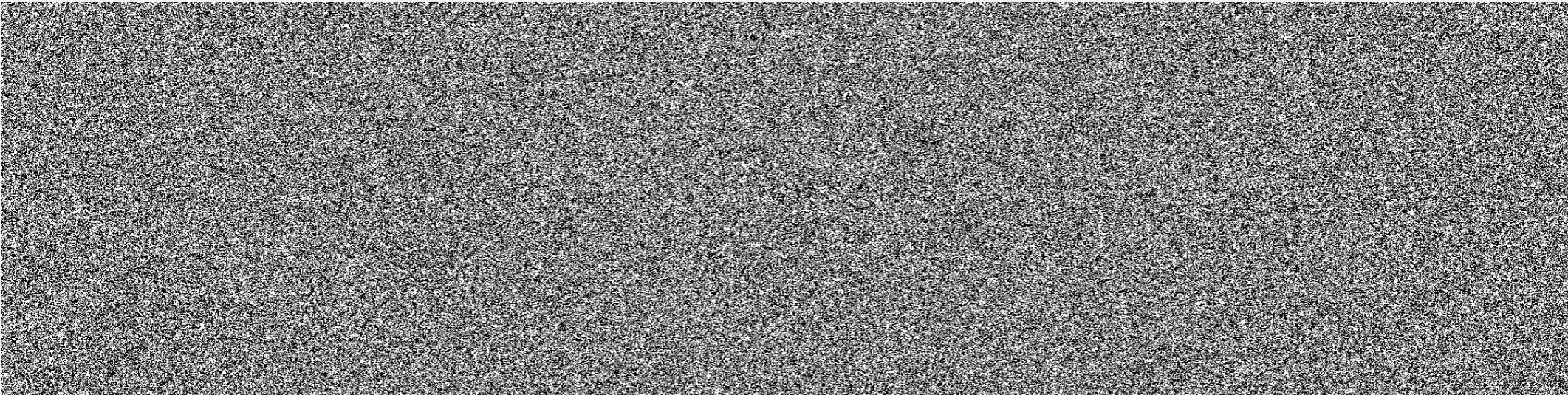
Pf 2. [by counting]

- Suppose your algorithm that can compress all 1,000-bit strings.
- 2^{1000} possible bitstrings with 1,000 bits.
- Only $1 + 2 + 4 + \dots + 2^{998} + 2^{999}$ can be encoded with ≤ 999 bits.
- Similarly, only 1 in 2^{499} bitstrings can be encoded with ≤ 500 bits!



Undecidability

```
% java RandomBits | java PictureDump 2000 500
```



1000000 bits

A difficult file to compress: one million (pseudo-) random bits

```
public class RandomBits
{
    public static void main(String[] args)
    {
        int x = 11111;
        for (int i = 0; i < 1000000; i++)
        {
            x = x * 314159 + 218281;
            BinaryStdOut.write(x > 0);
        }
        BinaryStdOut.close();
    }
}
```

Rdenudcany in Enlgsih Inagugae

Q. How much redundancy is in the English language?

“ ... randomising letters in the middle of words [has] little or no effect on the ability of skilled readers to understand the text. This is easy to denmtrasote. In a pubiltacion of New Scnieitst you could ramdinose all the letetrs, keipeng the first two and last two the same, and reibadailty would hadrly be aftcfeed. My ansaylis did not come to much beucase the thoery at the time was for shape and senqeuce retigcionon. Saberi's work sugsests we may have some pofrweul palrlael prsooscers at work. The resaon for this is suerly that idnetiyfing coentnt by paarllel prseocsing speeds up regnicoiton. We only need the first and last two letetrs to spot chganes in meniang.” — *Graham Rawlinson*

A. Quite a bit

Redundancy in Turkish Language

Q. How much redundancy is in the Turkish language?

“ Bir İngiliz Üniverisitelerinde yılalon arşaitramya grœ,
kleimleirn hrfalreiinn hnagi sridaa yzalıdkılraı ömneli
dğeliimş. Öenlmi oaln brincii ve snonucnu hrfain
yrenide omlsaımyş. Ardakai hfraliren srısaı kriash
oslada ouknyuorumş. Çnükü kleimrei hraf hrafdğeil bri
btün oalark oykuorumuşz” — *Anonymous*

A. Quite a bit

DATA COMPRESSION

- ▶ Run-length coding
- ▶ Huffman compression

Run-length encoding

Simple type of redundancy in a bitstream. Long runs of repeated bits.

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
40 bits

Representation. Use 4-bit counts to represent alternating runs of 0s and 1s:
15 0s, then 7 1s, then 7 0s, then 11 1s.

1 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 ← 16 bits (instead of 40)
15 7 7 11

Q. How many bits to store the counts?

A. We'll use 8 (but 4 in the example above).

Q. What to do when run length exceeds max count?

A. If longer than 255, intersperse runs of length 0.

Applications. JPEG, ITU-T T4 Group 3 Fax, ...

Run-length encoding: Java implementation

```
public class RunLength
{
    private final static int R    = 256;           ← maximum run-length count
    private final static int lgR = 8;              ← number of bits per count

    public static void compress()
    {   /* see textbook */   }

    public static void expand()
    {
        boolean bit = false;
        while (!BinaryStdIn.isEmpty())
        {
            int run = BinaryStdIn.readInt(lgR);   ← read 8-bit count from standard input
            for (int i = 0; i < run; i++)
                BinaryStdOut.write(bit);          ← write 1 bit to standard output
            bit = !bit;
        }
        BinaryStdOut.close();                  ← pad 0s for byte alignment
    }
}
```

An application: compress a bitmap

Typical black-and-white-scanned image.

- 300 pixels/inch.
 - 8.5-by-11 inches.
 - $300 \times 8.5 \times 300 \times 11 = 8.415$ million bits.

Observation. Bits are mostly white.

Typical amount of text on a page.

40 lines × 75 chars per line = 3,000 chars.

A typical bitmap, with run lengths for each row

Black and white bitmap compression: another approach

Fax machine (~1980).

- Slow scanner produces lines in sequential order.
- Compress to save time (reduce number of bits to send).

Electronic documents (~2000).

- High-resolution scanners produce huge files.
- Compress to save space (reduce number of bits to save).

Idea.

- use OCR to get back to ASCII (!)
- use Huffman on ASCII string (!)

Bottom line. Any extra information about file can yield dramatic gains.

DATA COMPRESSION

- ▶ Run-length coding
- ▶ Huffman compression

Variable-length codes

Use different number of bits to encode different chars.

Ex. Morse code: • • • - - - • • •

Issue. Ambiguity.

SOS ?

V7 ?

IAMIE ?

EEWNI ?

In practice. Use a medium gap to separate codewords.

codeword for S is a prefix
of codeword for V

Letters	Numbers
A	•—
B	—•••
C	—•—•
D	—••
E	•
F	••—•
G	——•
H	••••
I	••
J	•——
K	—•—
L	•—••
M	——
N	—•
O	———
P	•——•
Q	——•—
R	•—•
S	•••
T	—
U	••—
V	•••—
W	•——
X	—••—
Y	—•—
Z	——••

Variable-length codes

Q. How do we avoid ambiguity?

A. Ensure that no codeword is a **prefix** of another.

Ex 1. Fixed-length code.

Ex 2. Append special stop char to each codeword.

Ex 3. General prefix-free code.

Codeword table

<i>key</i>	<i>value</i>
!	101
A	0
B	1111
C	110
D	100
R	1110

Compressed bitstring

01111111001100100011111100101 ← 30 bits
A B RA CA DA B RA !

Codeword table

<i>key</i>	<i>value</i>
!	101
A	11
B	00
C	010
D	100
R	011

Compressed bitstring

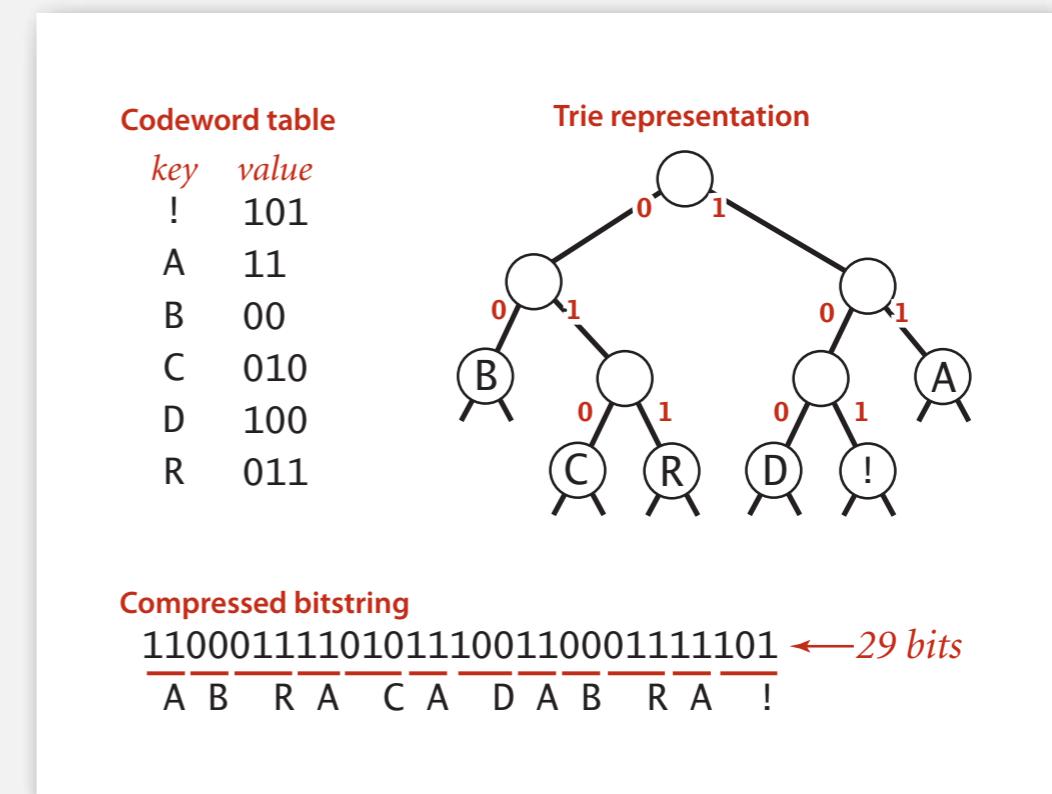
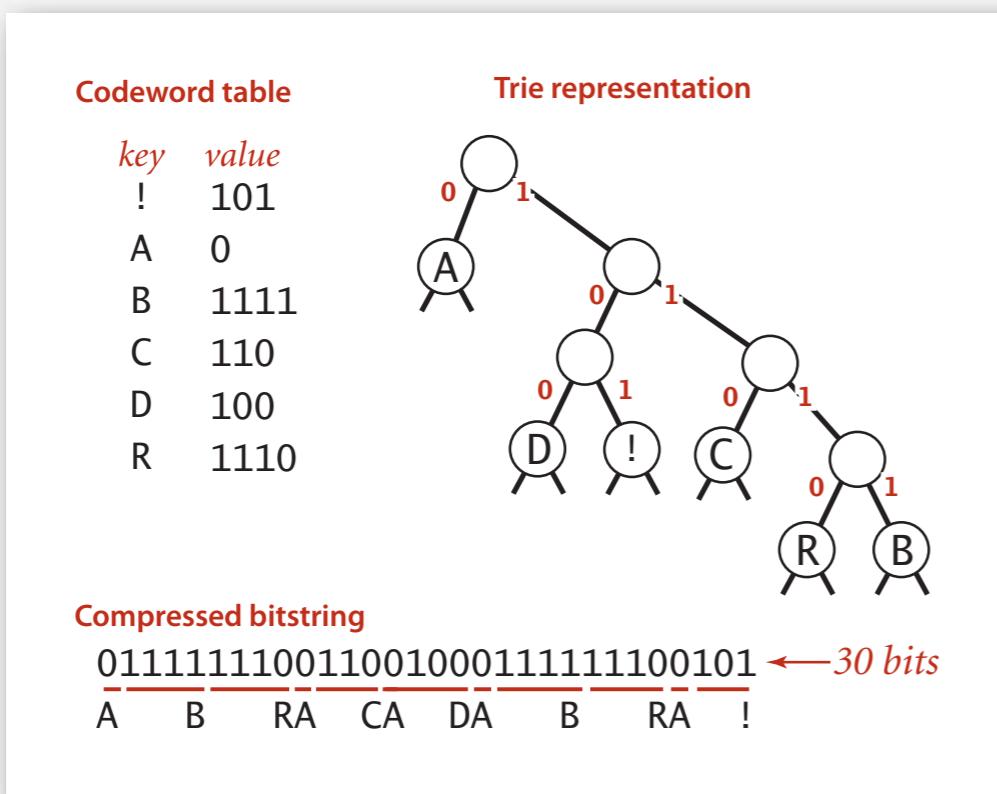
11000111101011100110001111101 ← 29 bits
A B RA CA DAB RA !

Prefix-free codes: trie representation

Q. How to represent the prefix-free code?

A. A binary trie!

- Chars in leaves.
- Codeword is path from root to leaf.



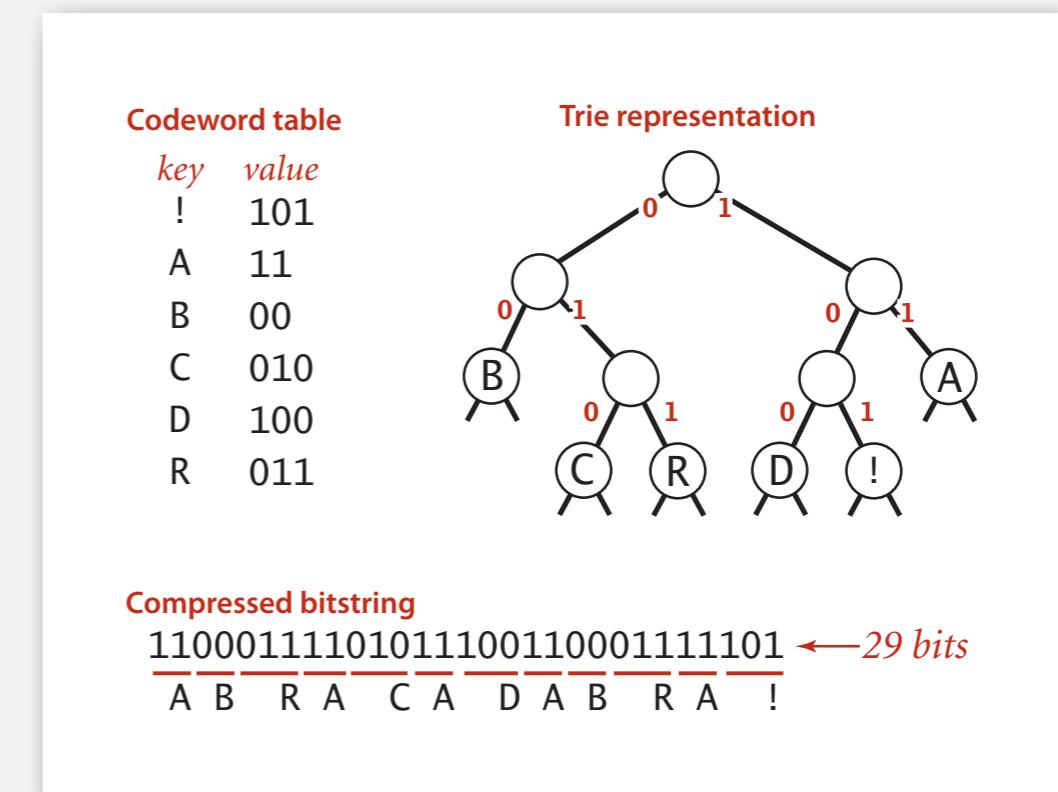
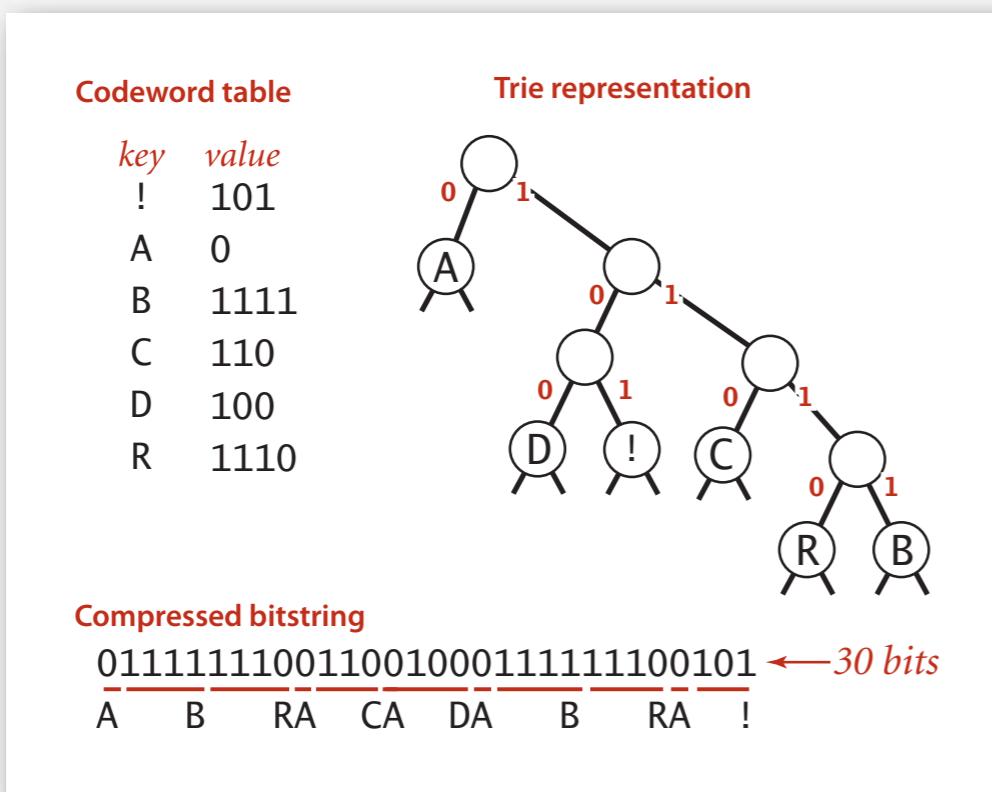
Prefix-free codes: compression and expansion

Compression.

- Method 1: start at leaf; follow path up to the root; print bits in reverse.
- Method 2: create ST of key-value pairs.

Expansion.

- Start at root.
- Go left if bit is 0; go right if 1.
- If leaf node, print char and return to root.



Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
    private char ch;      // Unused for internal nodes.
    private int freq;    // Unused for expand.
    private final Node left, right;

    public Node(char ch, int freq, Node left, Node right)
    {
        this.ch      = ch;
        this.freq   = freq;
        this.left   = left;
        this.right  = right;
    }

    public boolean isLeaf()
    { return left == null && right == null; }

    public int compareTo(Node that)
    { return this.freq - that.freq; }
}
```

← initializing constructor

← is Node a leaf?

← compare Nodes by frequency
(stay tuned)

Prefix-free codes: expansion

```
public void expand()
{
    Node root = readTrie();
    int N = BinaryStdIn.readInt();           ← read in encoding trie
                                              ← read in number of chars

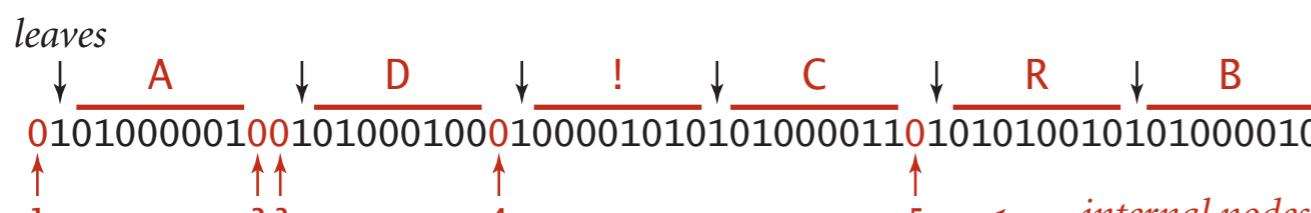
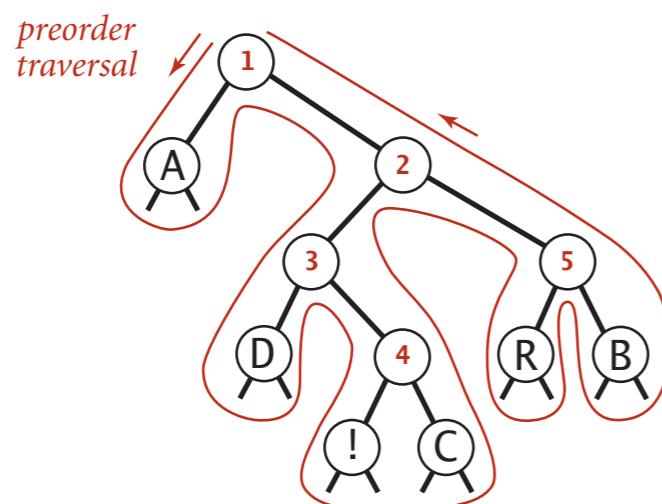
    for (int i = 0; i < N; i++)
    {
        Node x = root;
        while (!x.isLeaf())
            ← expand codeword for ith char
        {
            if (!BinaryStdIn.readBoolean())
                x = x.left;
            else
                x = x.right;
        }
        BinaryStdOut.write(x.ch, 8);
    }
    BinaryStdOut.close();
}
```

Running time. Linear in input size N .

Prefix-free codes: how to transmit

Q. How to write the trie?

A. Write preorder traversal of trie; mark leaf and internal nodes with a bit.



Using preorder traversal to encode a trie as a bitstream

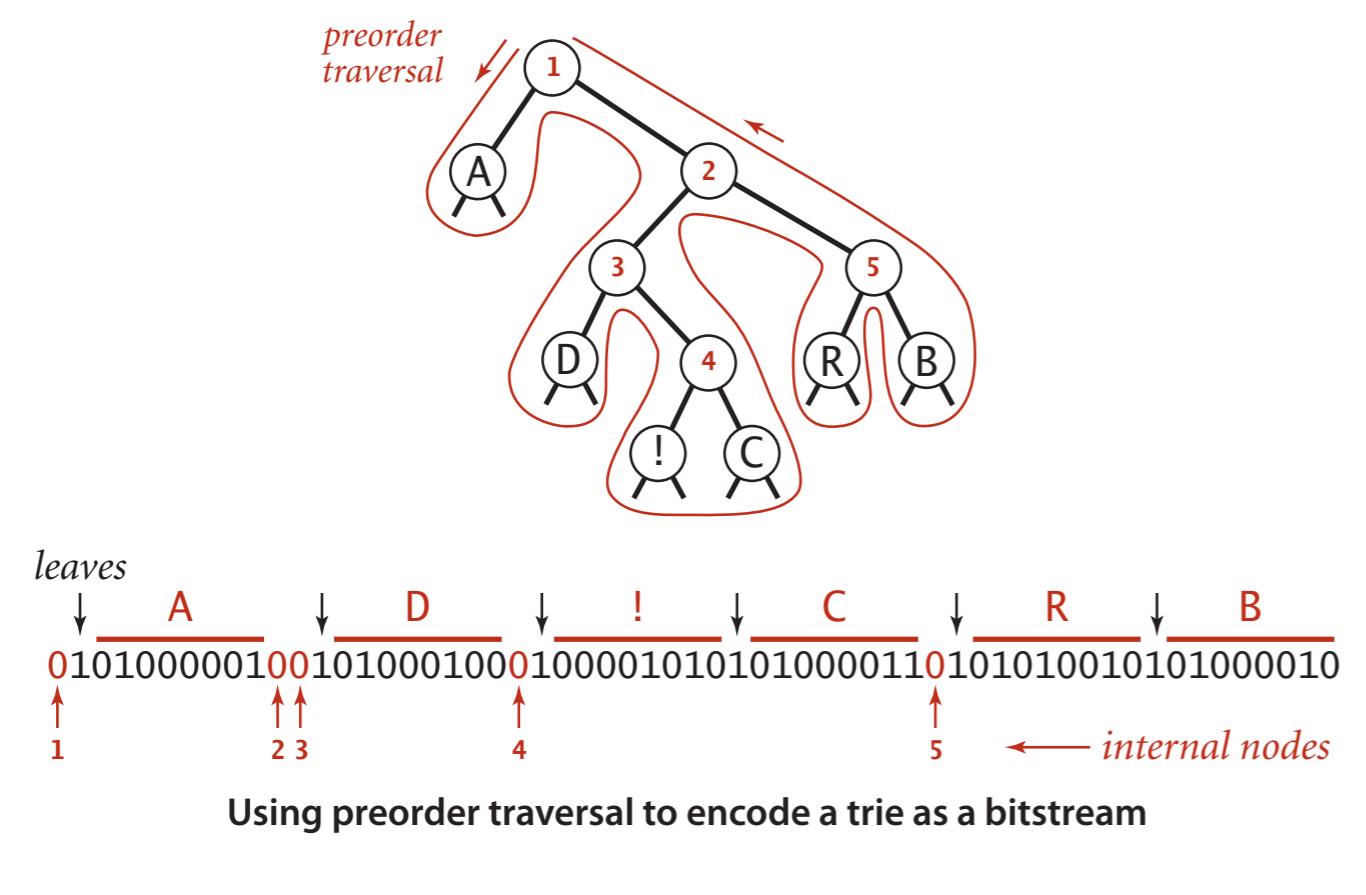
```
private static void writeTrie(Node x)
{
    if (x.isLeaf())
    {
        BinaryStdOut.write(true);
        BinaryStdOut.write(x.ch, 8);
        return;
    }
    BinaryStdOut.write(false);
    writeTrie(x.left);
    writeTrie(x.right);
}
```

Note. If message is long, overhead of transmitting trie is small.

Prefix-free codes: how to transmit

Q. How to read in the trie?

A. Reconstruct from preorder traversal of trie.



```
private static Node readTrie()
{
    if (BinaryStdIn.readBoolean())
    {
        char c = BinaryStdIn.readChar(8);
        return new Node(c, 0, null, null);
    }
    Node x = readTrie();
    Node y = readTrie();
    return new Node('\0', 0, x, y);
}
```

not used for
internal nodes

Shannon-Fano codes

Q. How to find best prefix-free code?

Shannon-Fano algorithm:

- Partition symbols S into two subsets S_0 and S_1 of (roughly) equal frequency.
- Codewords for symbols in S_0 start with 0; for symbols in S_1 start with 1.
- Recur in S_0 and S_1 .

char	freq	encoding
A	5	0...
C	1	0...

$S_0 = \text{codewords starting with 0}$

char	freq	encoding
B	2	1...
D	1	1...
R	2	1...
!	1	1...

$S_1 = \text{codewords starting with 1}$

Problem 1. How to divide up symbols?

Problem 2. Not optimal!

Huffman algorithm

- Count frequency for each character in input.

char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	

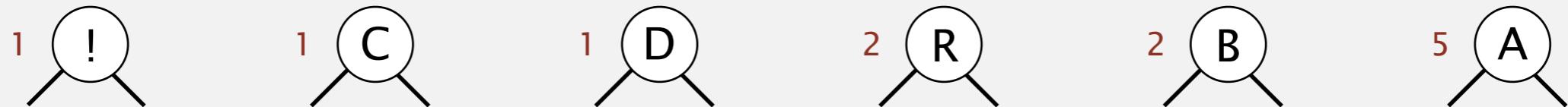
input

A B R A C A D A B R A !

Huffman algorithm

- Start with one node corresponding to each character with weight equal to frequency.

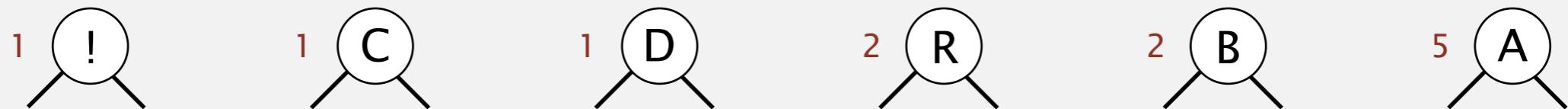
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

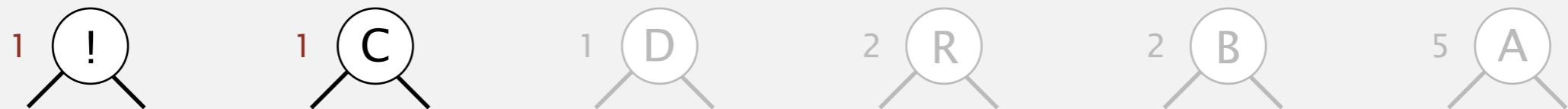
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

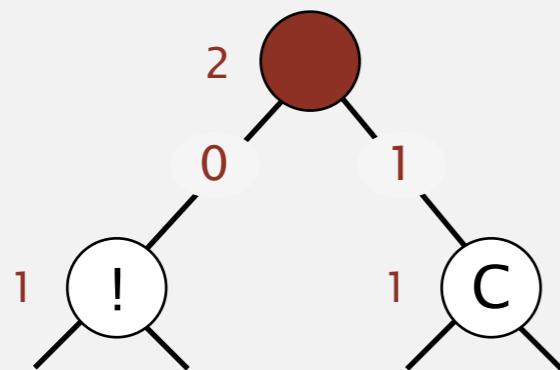
char	freq	encoding
A	5	
B	2	
C	1	
D	1	
R	2	
!	1	



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

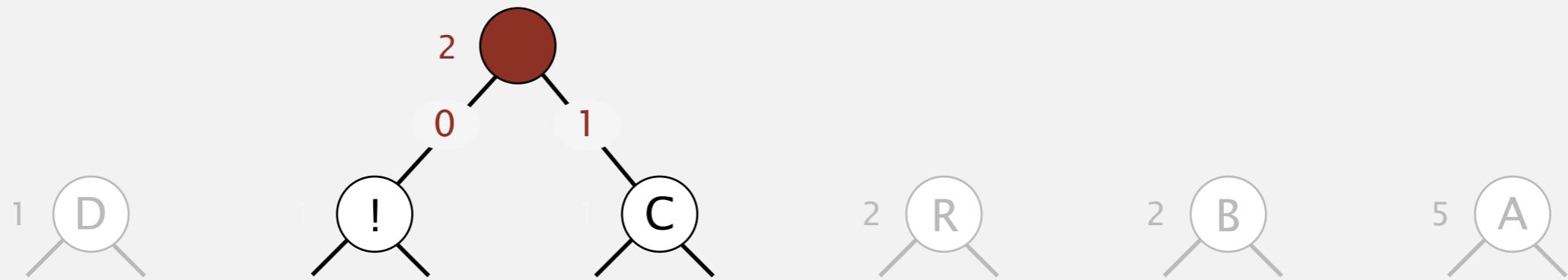
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

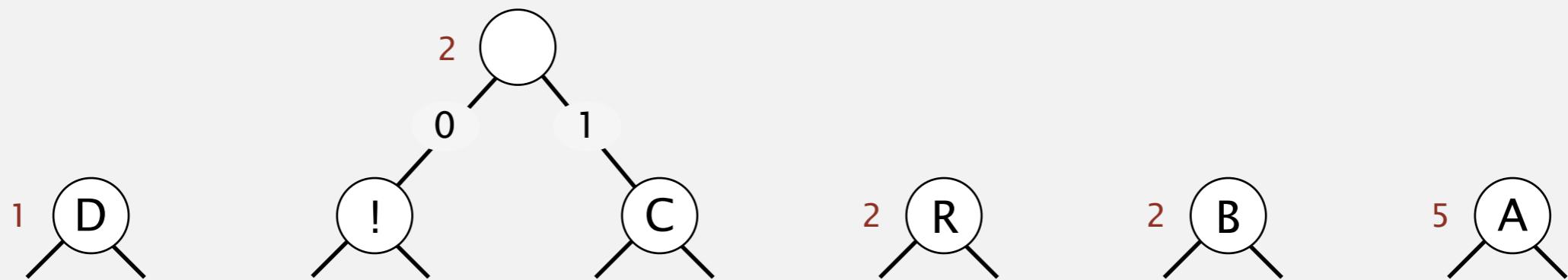
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

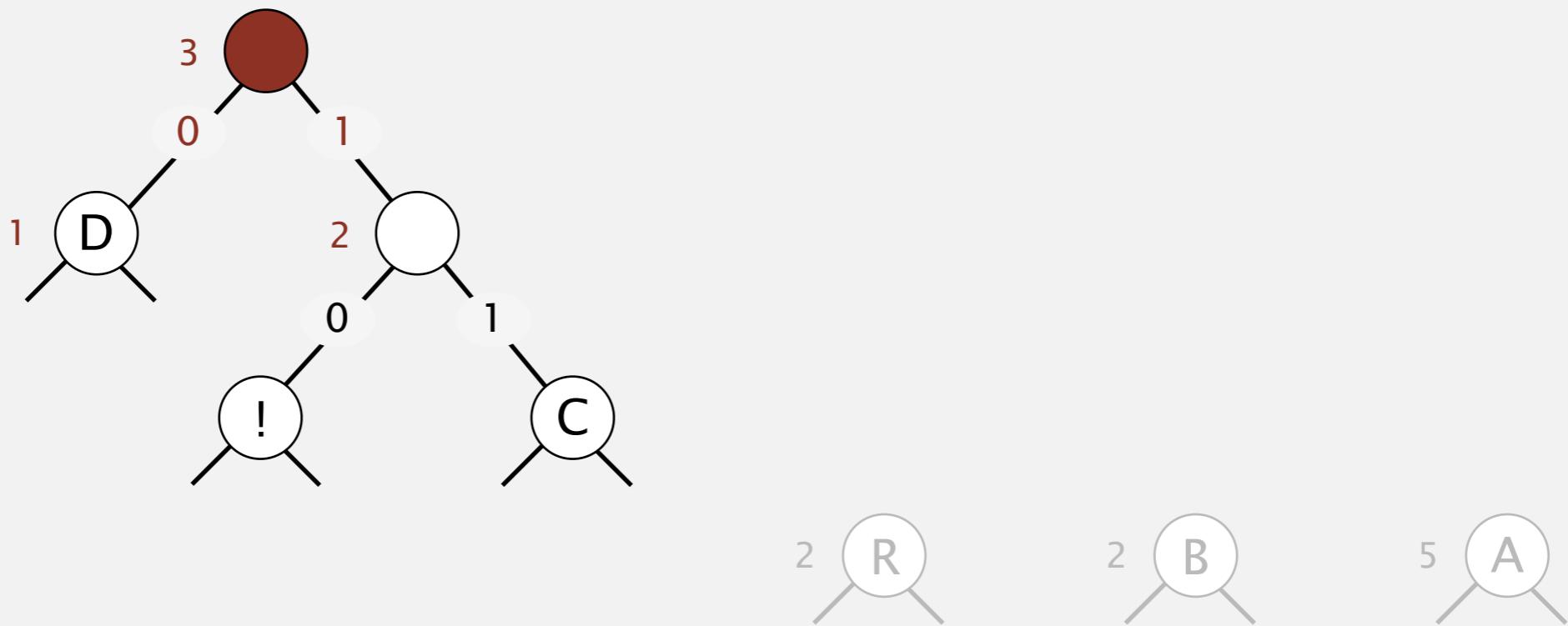
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	
R	2	
!	1	0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

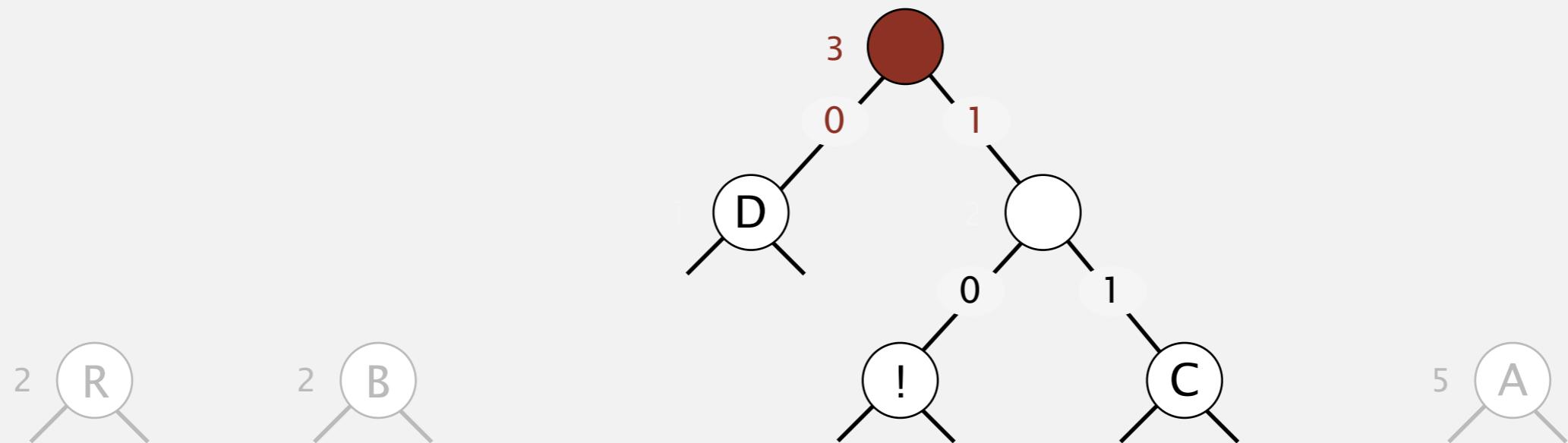
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	0
R	2	
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

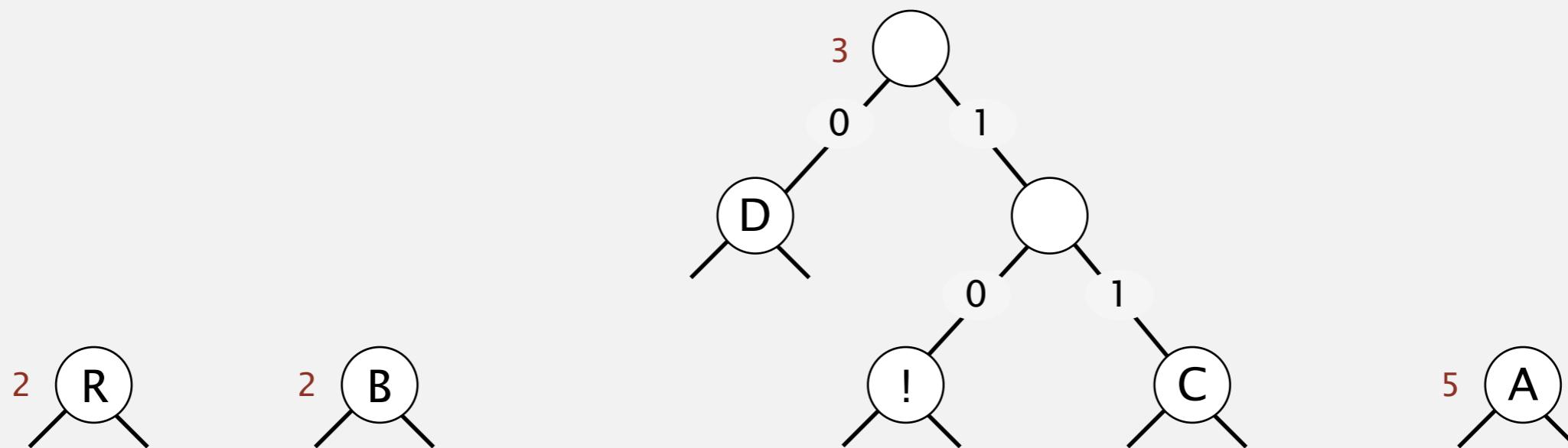
char	freq	encoding
A	5	
B	2	
C	1	1
D	1	0
R	2	
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

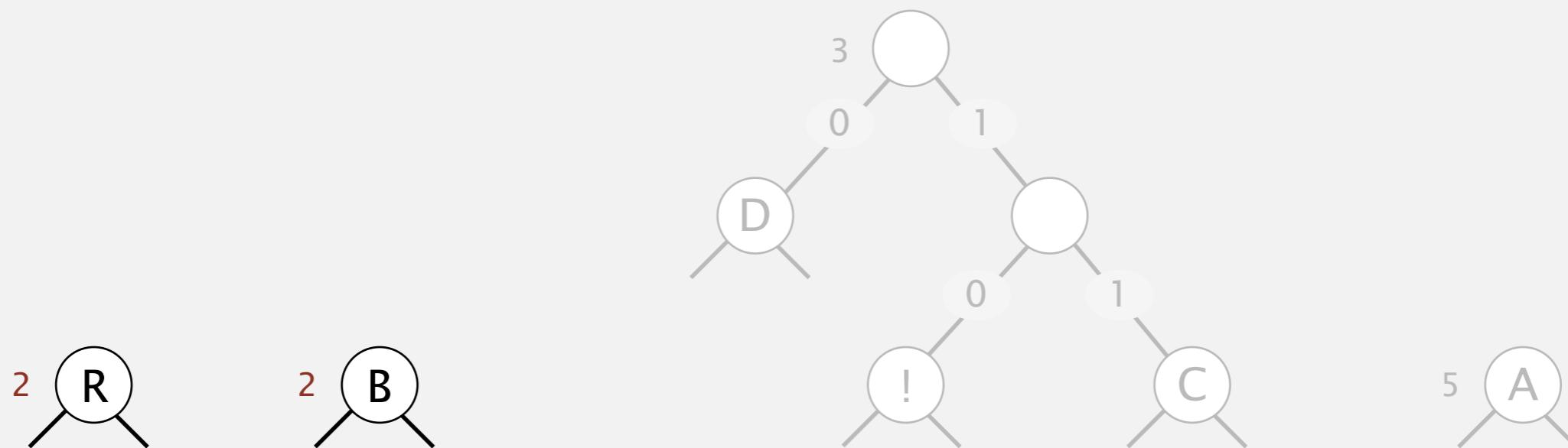
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

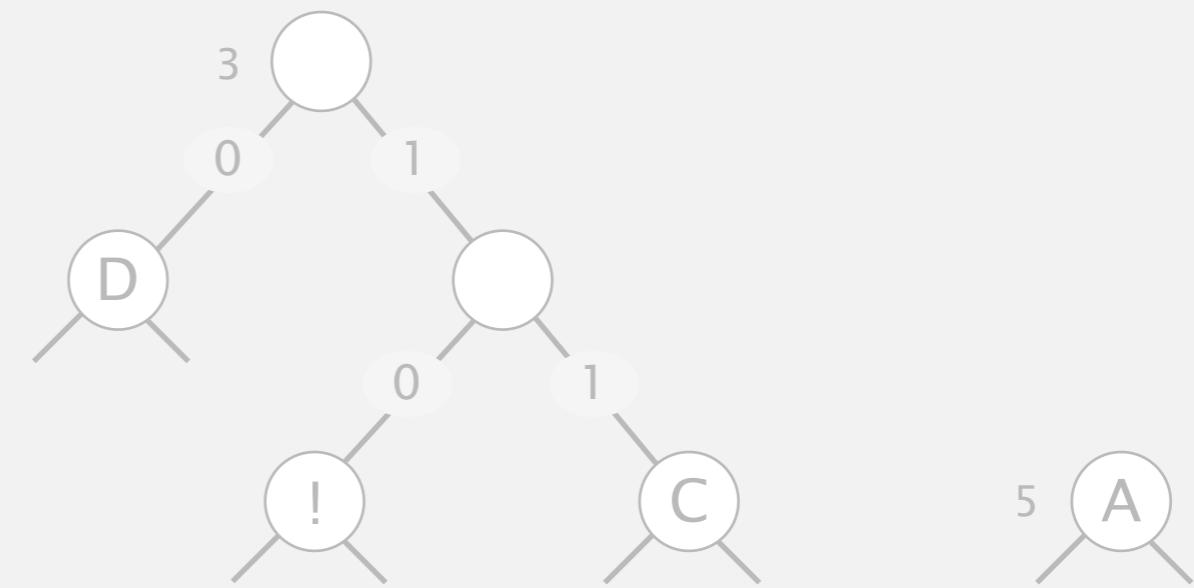
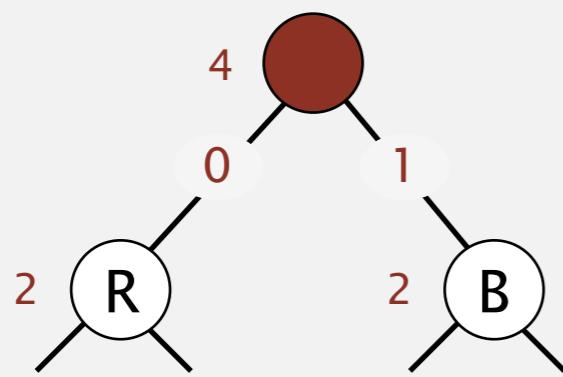
char	freq	encoding
A	5	
B	2	
C	1	1 1
D	1	0
R	2	
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

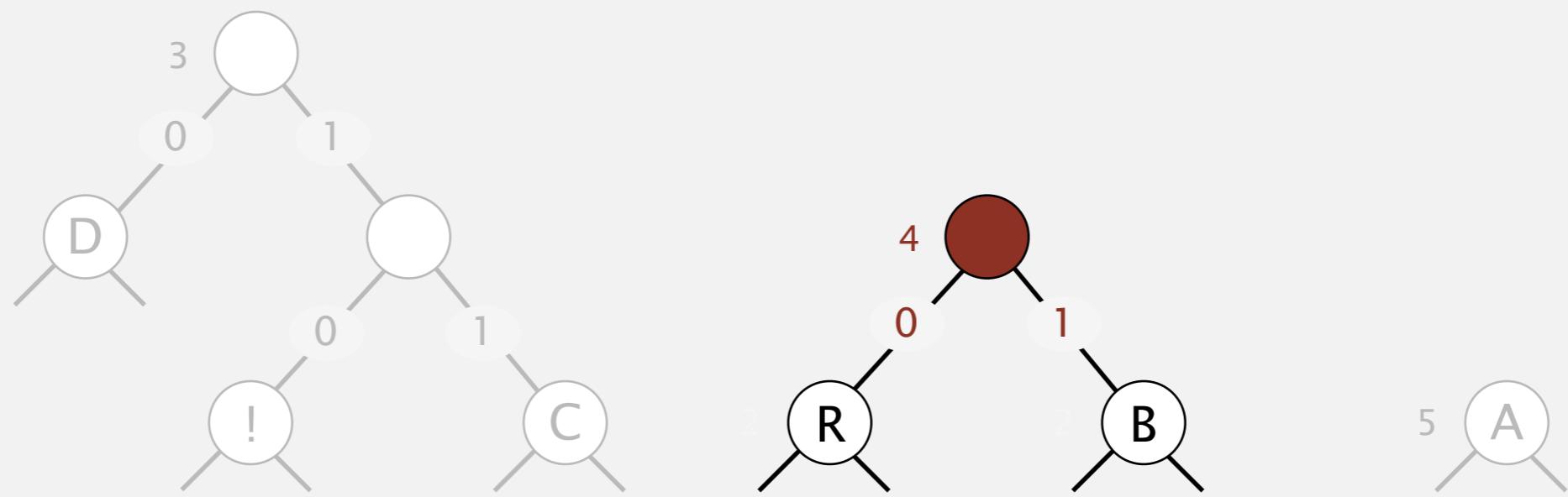
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

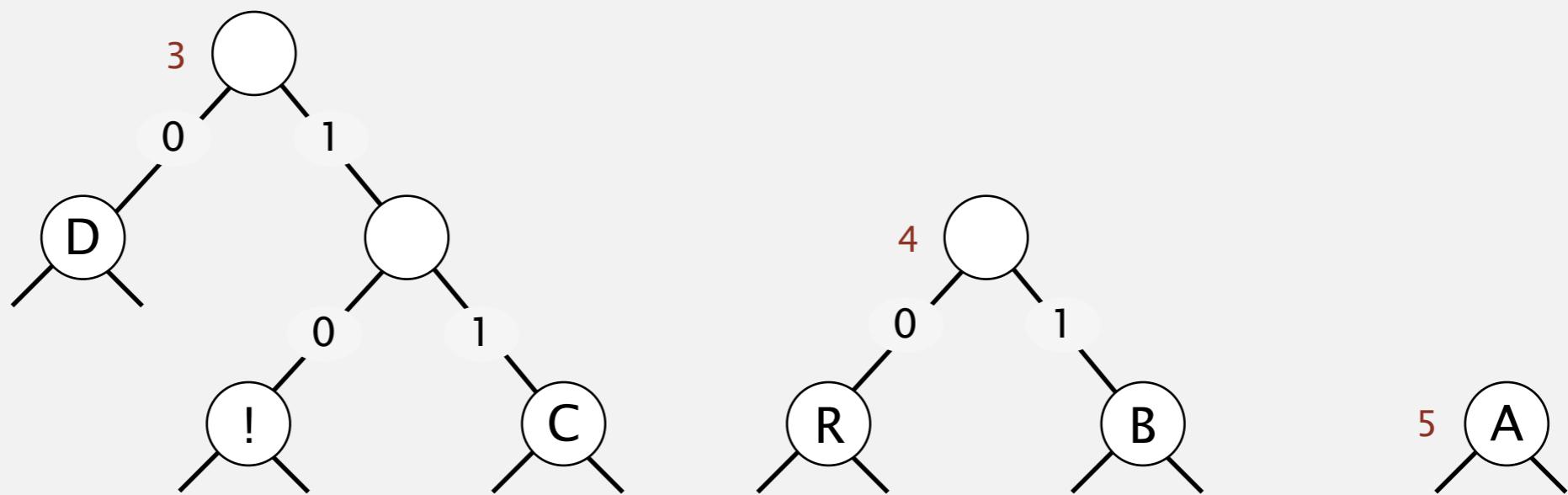
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

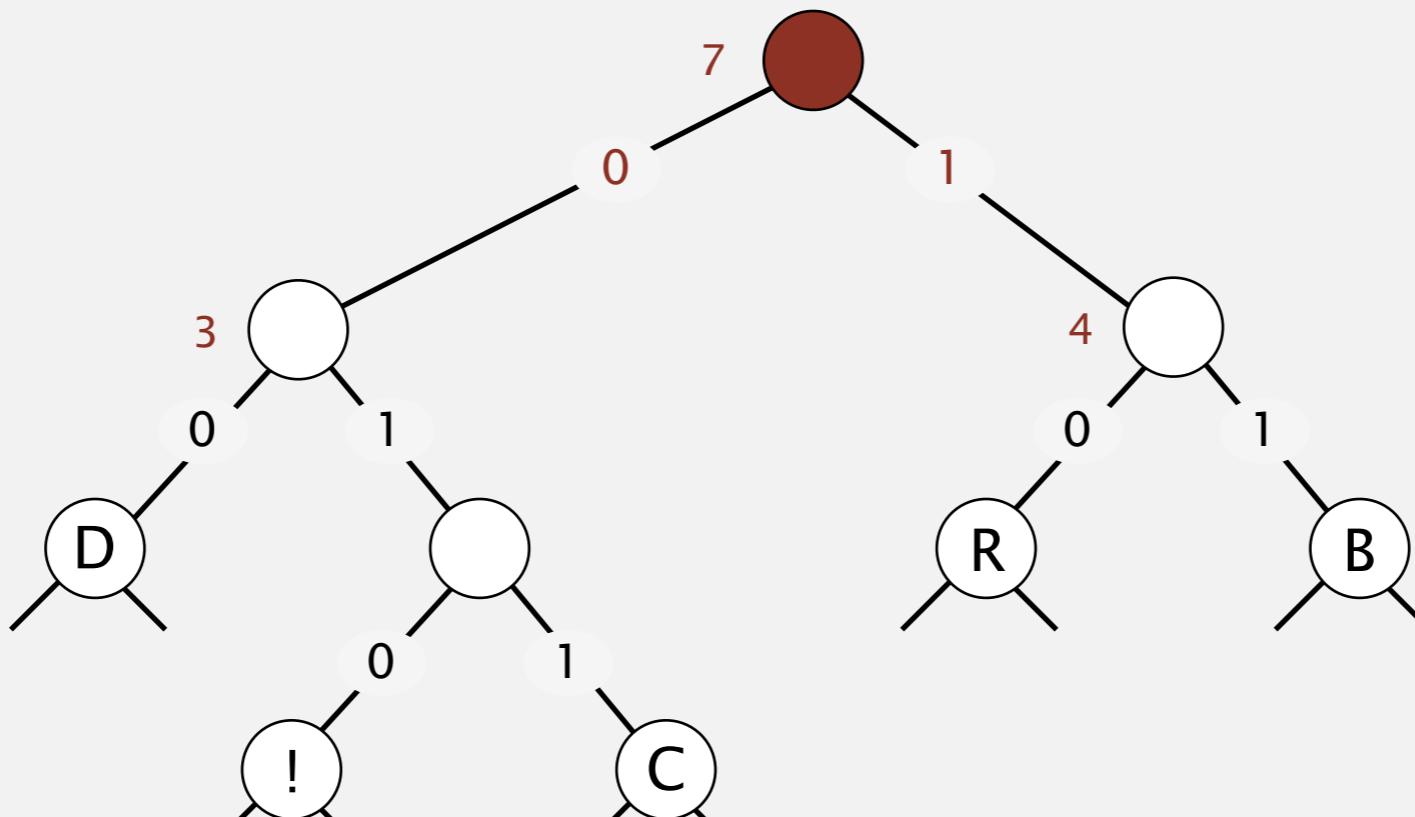
char	freq	encoding
A	5	
B	2	1
C	1	1 1
D	1	0
R	2	0
!	1	1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

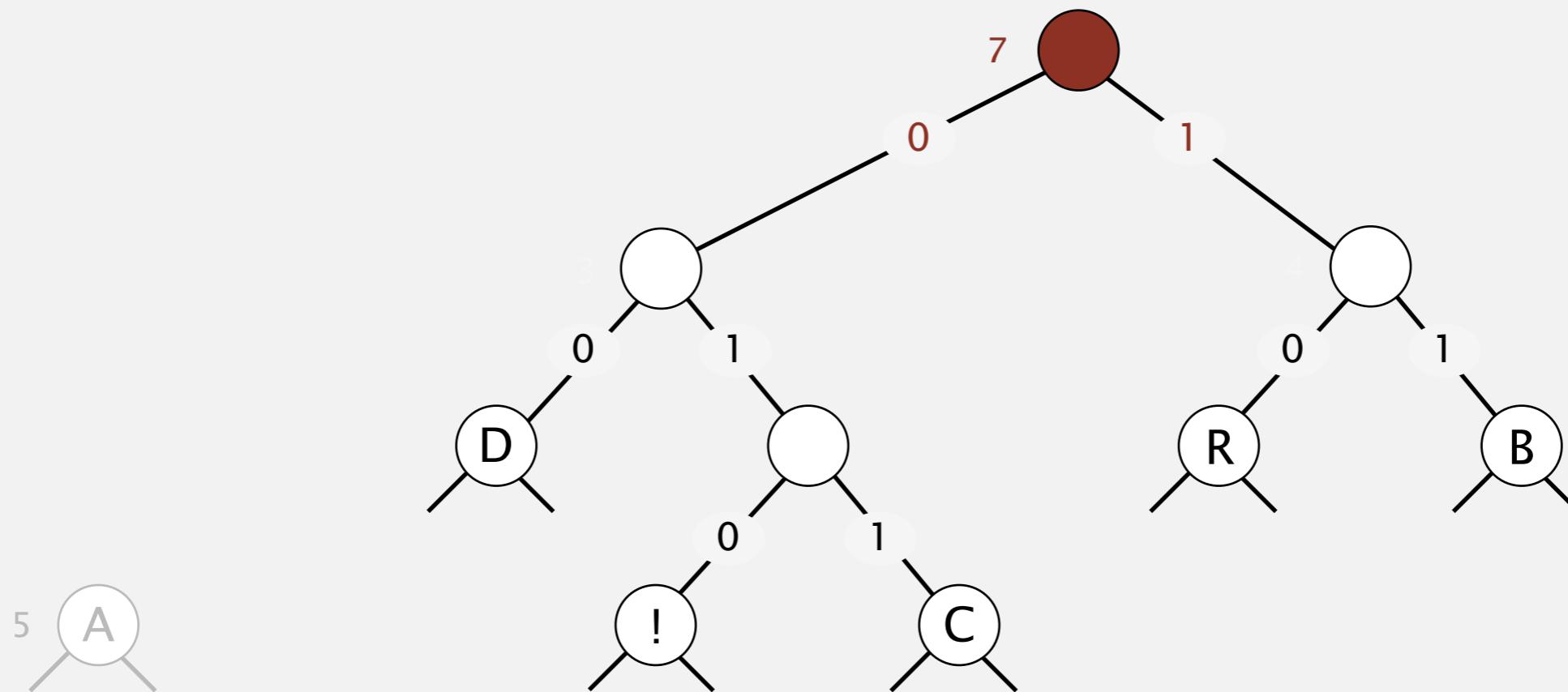
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.

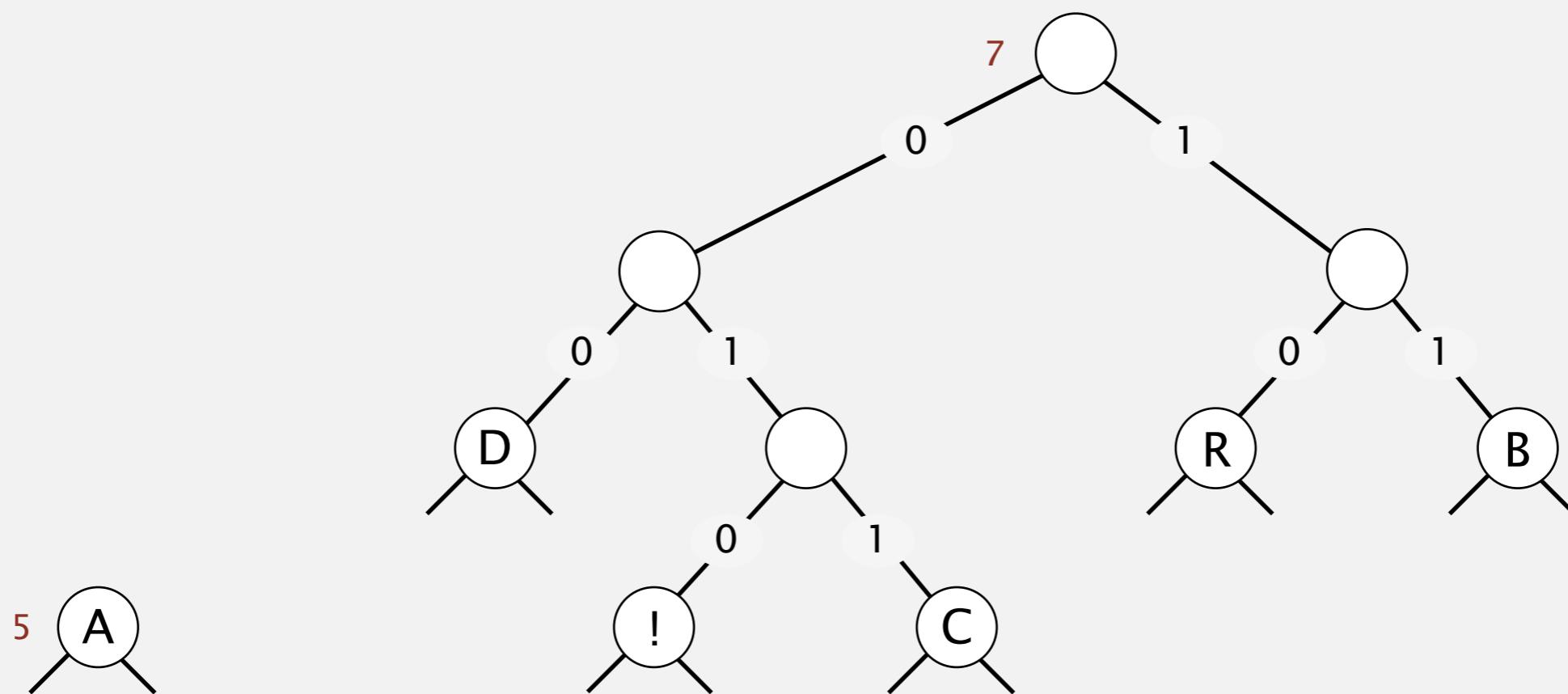
char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



Huffman algorithm

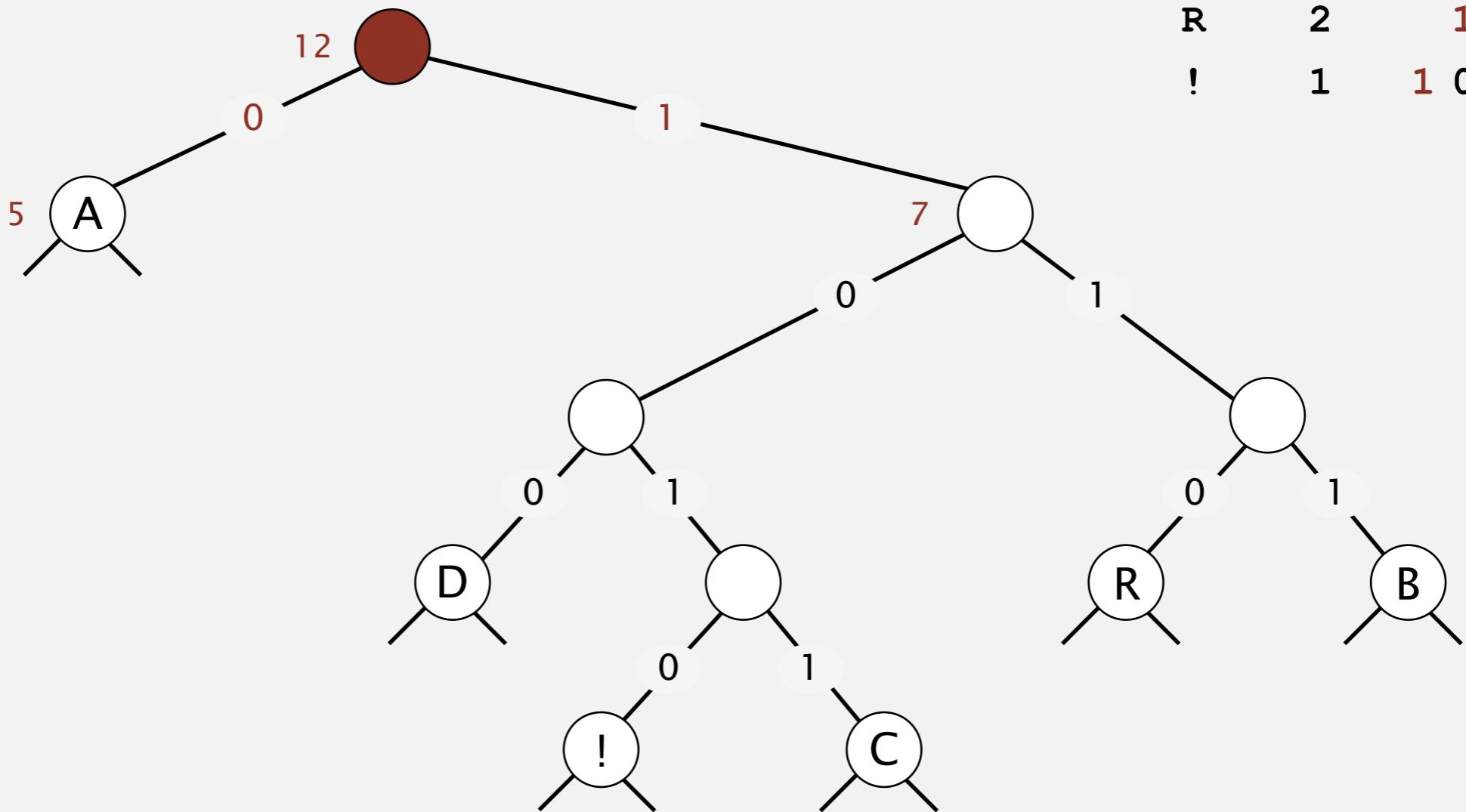
- Select two tries with min weight.
 - Merge into single trie with cumulative weight.

char	freq	encoding
A	5	
B	2	1 1
C	1	0 1 1
D	1	0 0
R	2	1 0
!	1	0 1 0



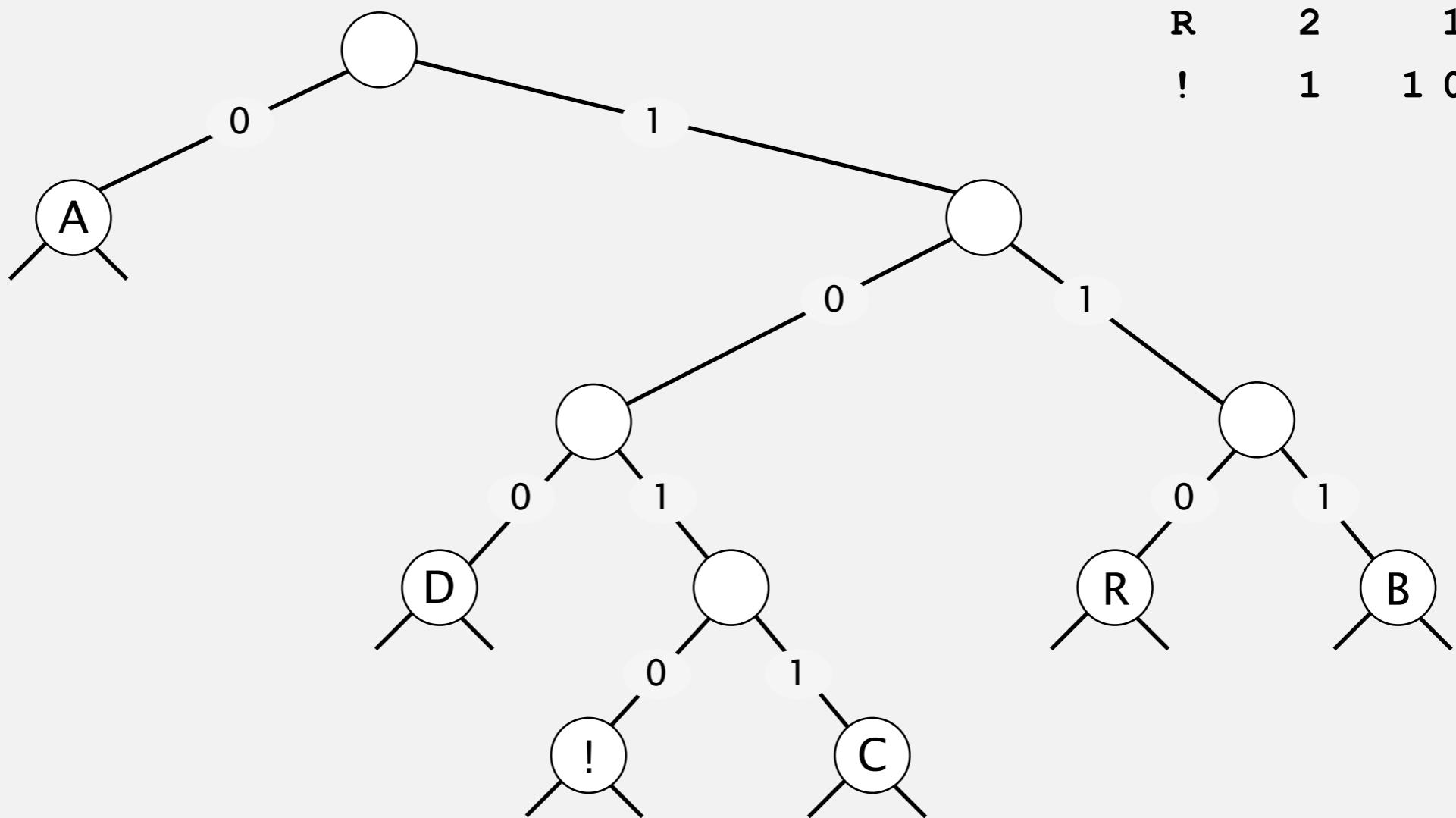
Huffman algorithm

- Select two tries with min weight.
- Merge into single trie with cumulative weight.



Huffman algorithm

char	freq	encoding
A	5	0
B	2	1 1 1
C	1	1 0 1 1
D	1	1 0 0
R	2	1 1 0
!	1	1 0 1 0



Huffman codes

Q. How to find best prefix-free code?

Huffman algorithm:

- Count frequency $\text{freq}[i]$ for each char i in input.
- Start with one node corresponding to each char i (with weight $\text{freq}[i]$).
- Repeat until single trie formed:
 - select two tries with min weight $\text{freq}[i]$ and $\text{freq}[j]$
 - merge into single trie with weight $\text{freq}[i] + \text{freq}[j]$

Applications:



Constructing a Huffman encoding trie: Java implementation

```
private static Node buildTrie(int[] freq)
{
    MinPQ<Node> pq = new MinPQ<Node>();
    for (char i = 0; i < R; i++)
        if (freq[i] > 0)
            pq.insert(new Node(i, freq[i], null, null));

    while (pq.size() > 1)
    {
        Node x = pq.delMin();
        Node y = pq.delMin();
        Node parent = new Node('\0', x.freq + y.freq, x, y);
        pq.insert(parent);
    }

    return pq.delMin();
}
```

initialize PQ with singleton tries

merge two smallest tries

not used for internal nodes total frequency two subtrees

Huffman encoding summary

Proposition. [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code.

Pf. See textbook.

↑
no prefix-free code uses fewer bits

Implementation.

- Pass 1: tabulate char frequencies and build trie.
- Pass 2: encode file by traversing trie or lookup table.

Running time. Using a binary heap $\Rightarrow N + R \log R$.

↑ ↑
input size alphabet size

Q. Can we do better?