

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

# 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)	
public class MaxPQ<Key extends Comparable<Key>>	
MaxPQ()	<i>create an empty priority queue</i>
MaxPQ(Key[] a)	<i>create a priority queue with given keys</i>
void insert(Key v)	<i>insert a key into the priority queue</i>
Key delMax()	<i>return and remove the largest key</i>
boolean isEmpty()	<i>is the priority queue empty?</i>
Key max()	<i>return the largest key</i>
int size()	<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]
- 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.

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

$N$  huge,  $M$  large

**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.

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

$N$  huge,  $M$  large

**Constraint.** Not enough memory to store  $N$  items.

```
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 contains  
        pq.delMin();      largest M items  
    }  
                          ↑  
Transaction data  
type is Comparable  
(ordered by $$)
```

# Priority queue client example

---

**Challenge.** Find the largest  $M$  items in a stream of  $N$  items.

**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 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

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	$\log N$	$\log N$	$\log N$

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

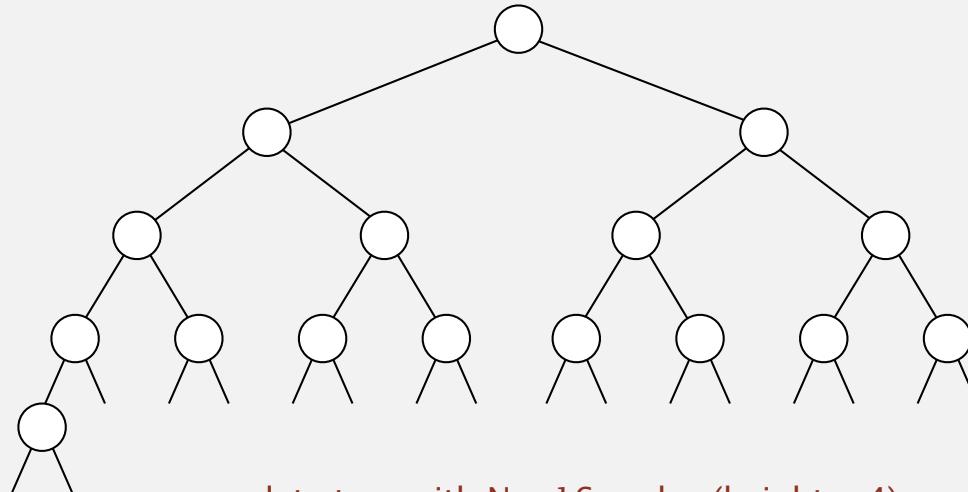
- ▶ *API and elementary implementations*
- ▶ ***binary heaps***
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Complete 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

# Binary heap representations

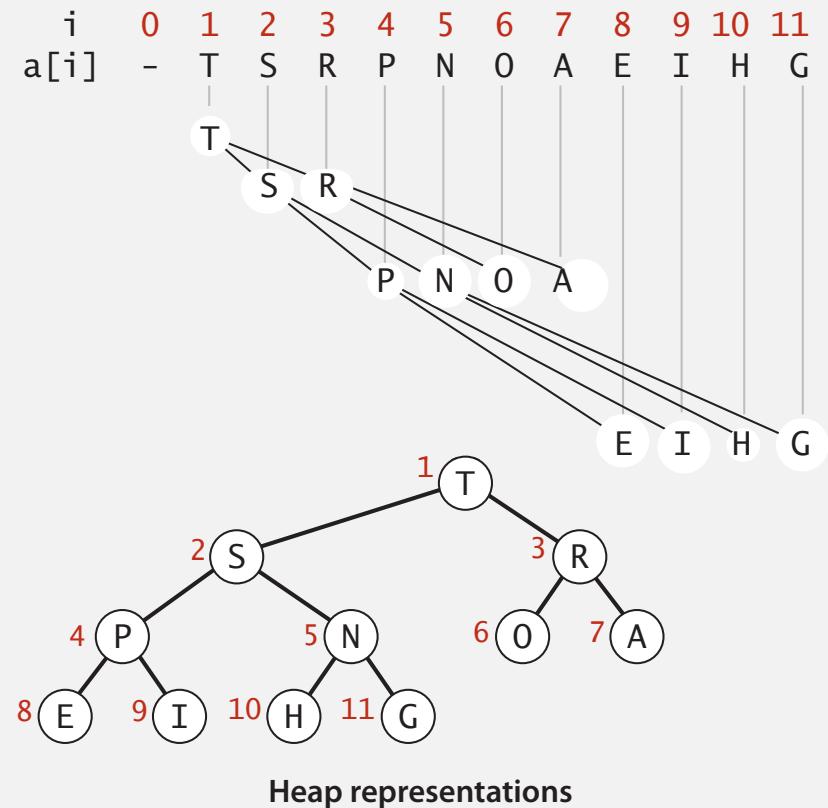
Binary heap. Array representation of a heap-ordered complete binary tree.

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!



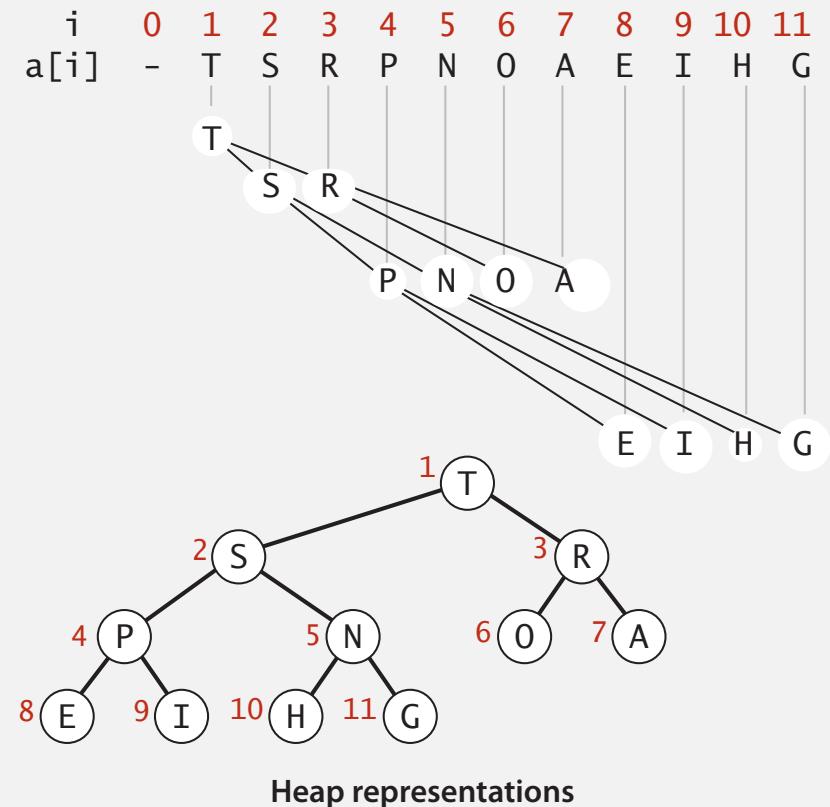
# Binary heap properties

---

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

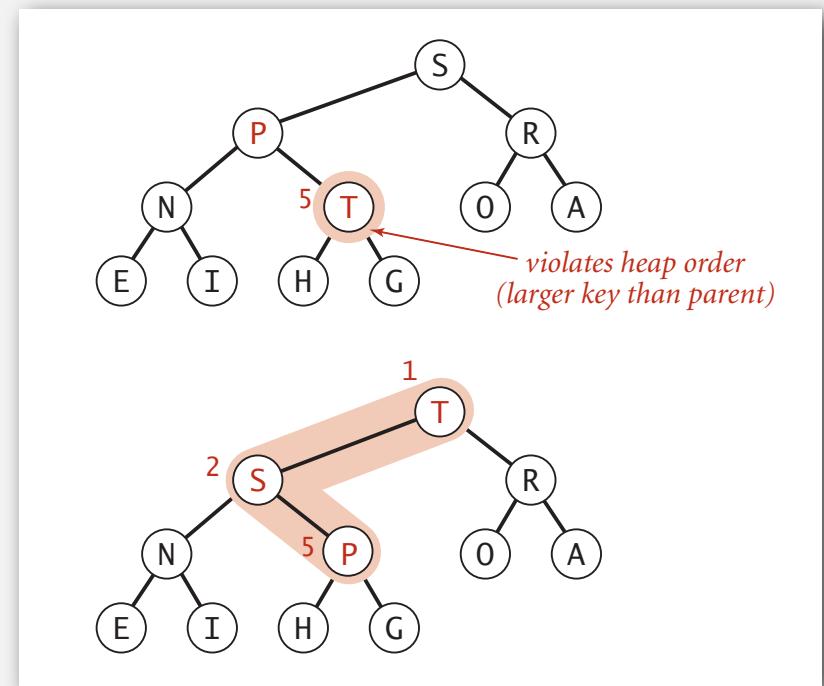
Scenario. Child's key becomes **larger** key than its parent's key.

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



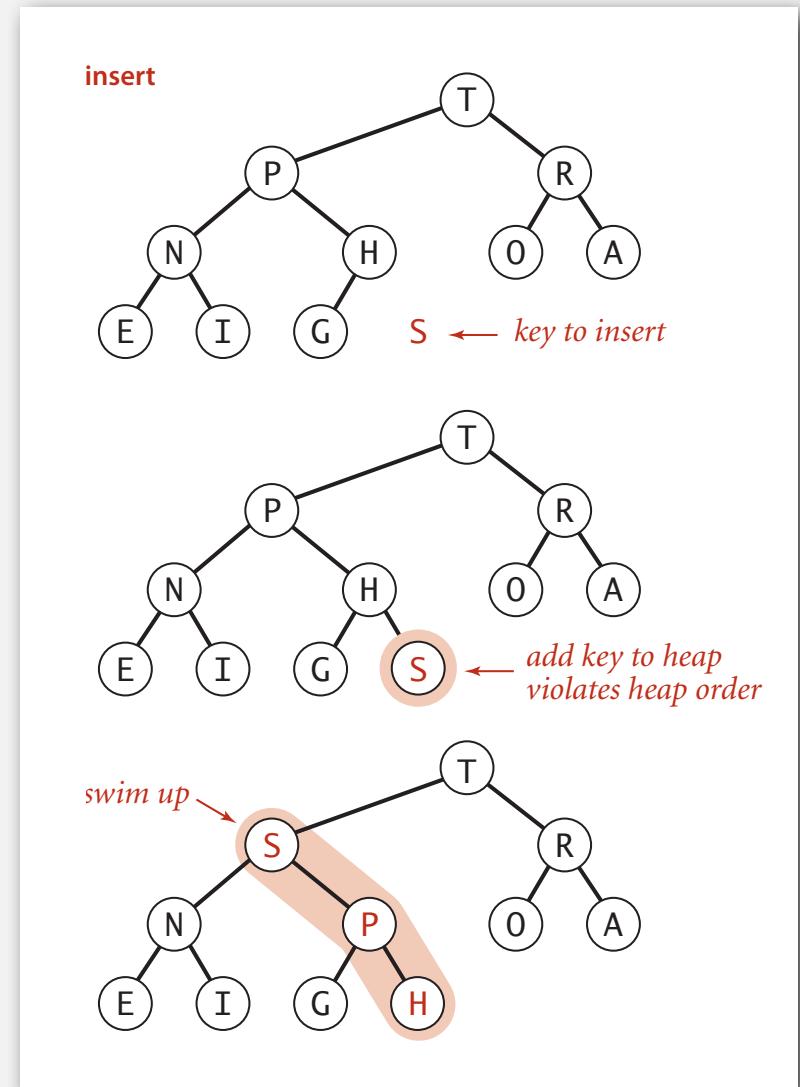
Peter principle. Node promoted to level of incompetence.

# 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

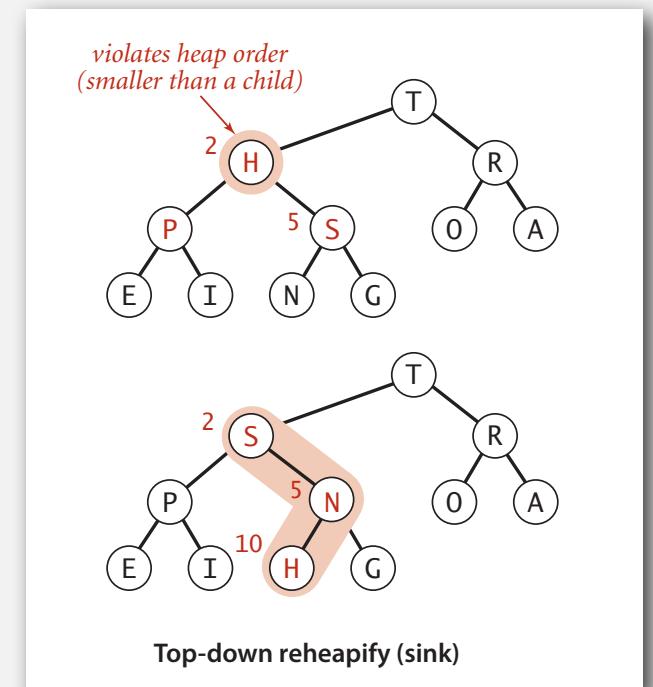
Scenario. Parent's key becomes **smaller** than one (or both) of its children's.

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)           children of node at k
    {                           are 2k and 2k+1
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```



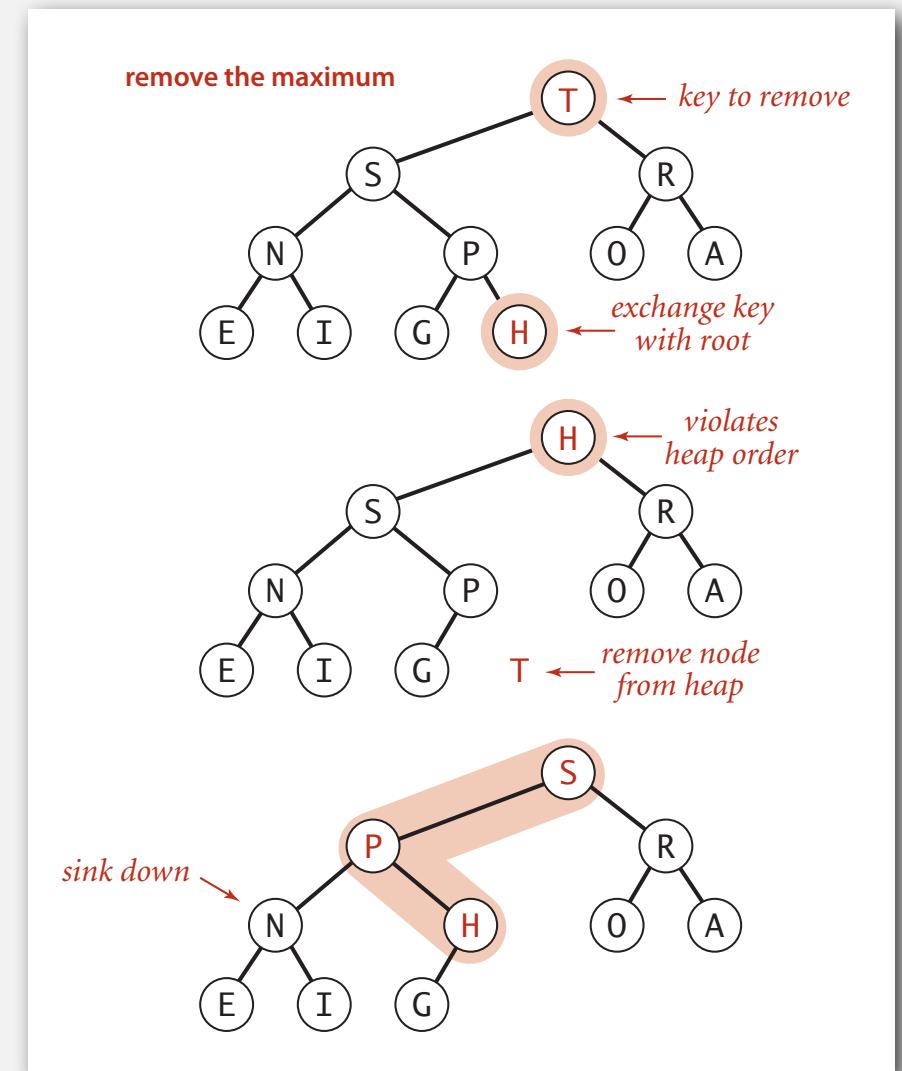
Power struggle. Better subordinate promoted.

# 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;
}
```



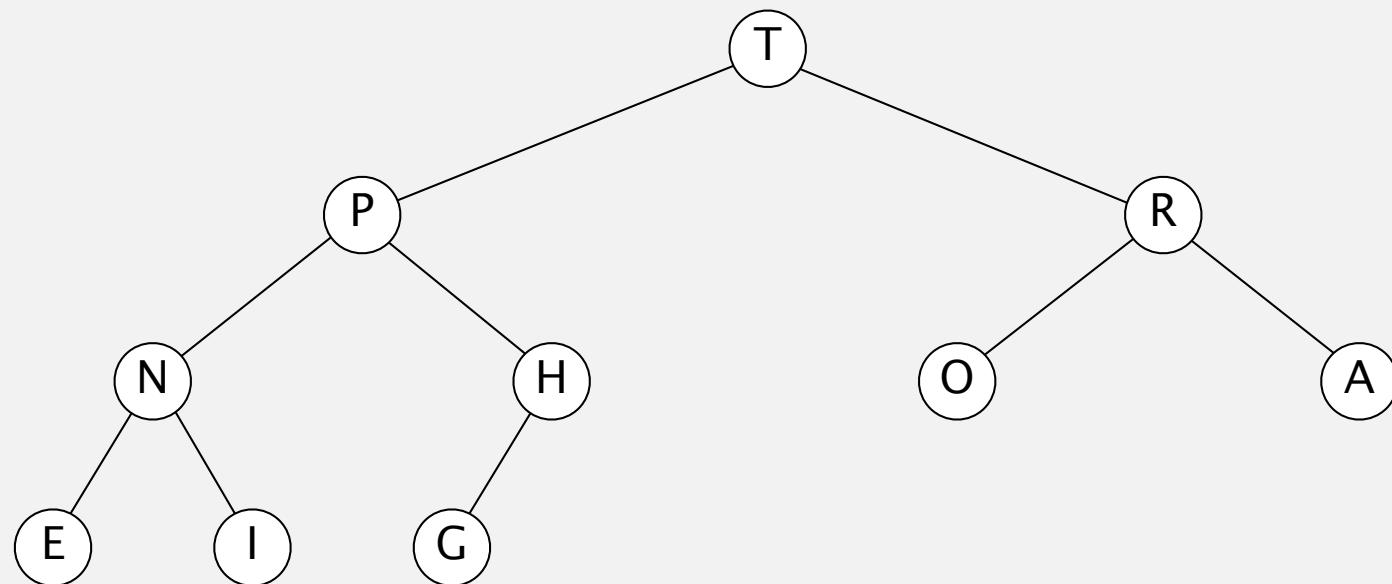
# Binary heap demo

---

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

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

heap ordered



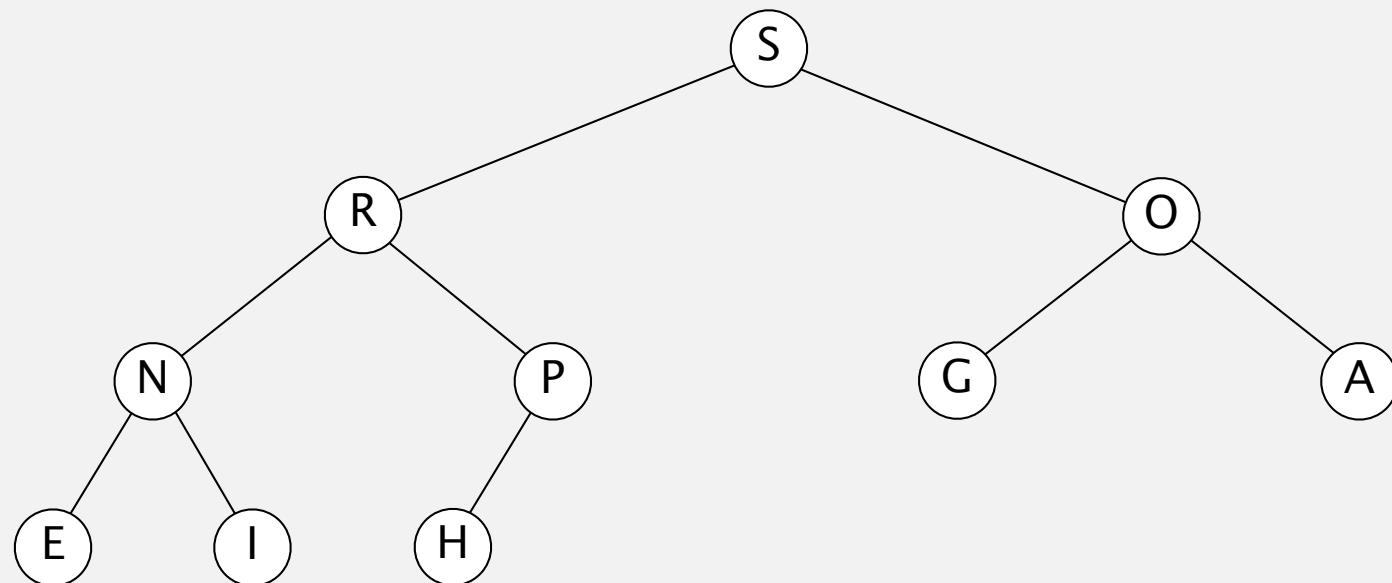
# Binary heap demo

---

**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)
    public Key delMax()
    {   /* see previous code */ }

    private void swim(int k)
    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;   }
}
```

fixed capacity  
(for simplicity)

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	$\log N$ †	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 {           ← can't override instance methods
    private final int N;
    private final double[] data;        ← all instance variables private and final

    public Vector(double[] data) {
        this.N = data.length;
        this.data = new double[N];
        for (int i = 0; i < N; i++)      ← defensive copy of mutable
            this.data[i] = data[i];       ← instance variables
    }

    ...
}
```

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.**

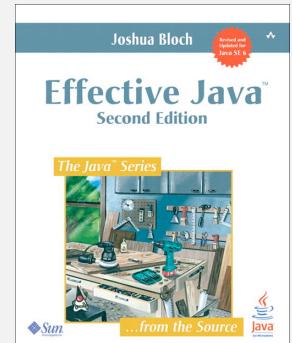
- Simplifies debugging.
- Safer in presence of hostile code.
- Simplifies concurrent programming.
- Safe to use as key in priority queue or symbol table.



**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)



# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ ***binary heaps***
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

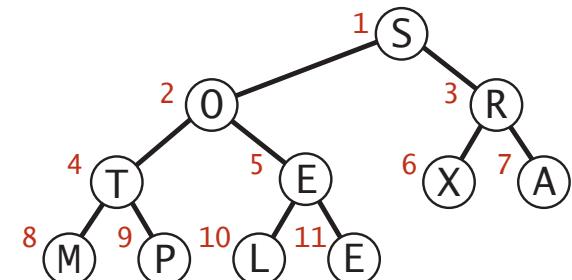
- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Heapsort

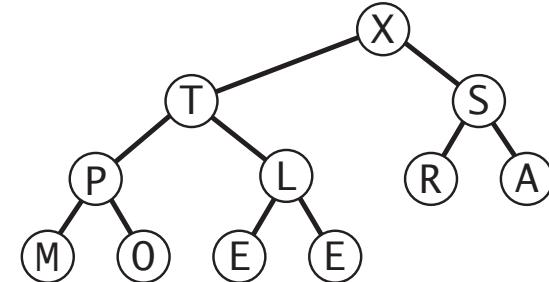
Basic plan for in-place sort.

- 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)

1 A  
2 E  
3 E  
4 L  
5 M  
6 O  
7 P  
8 R  
9 S  
10 T  
11 X

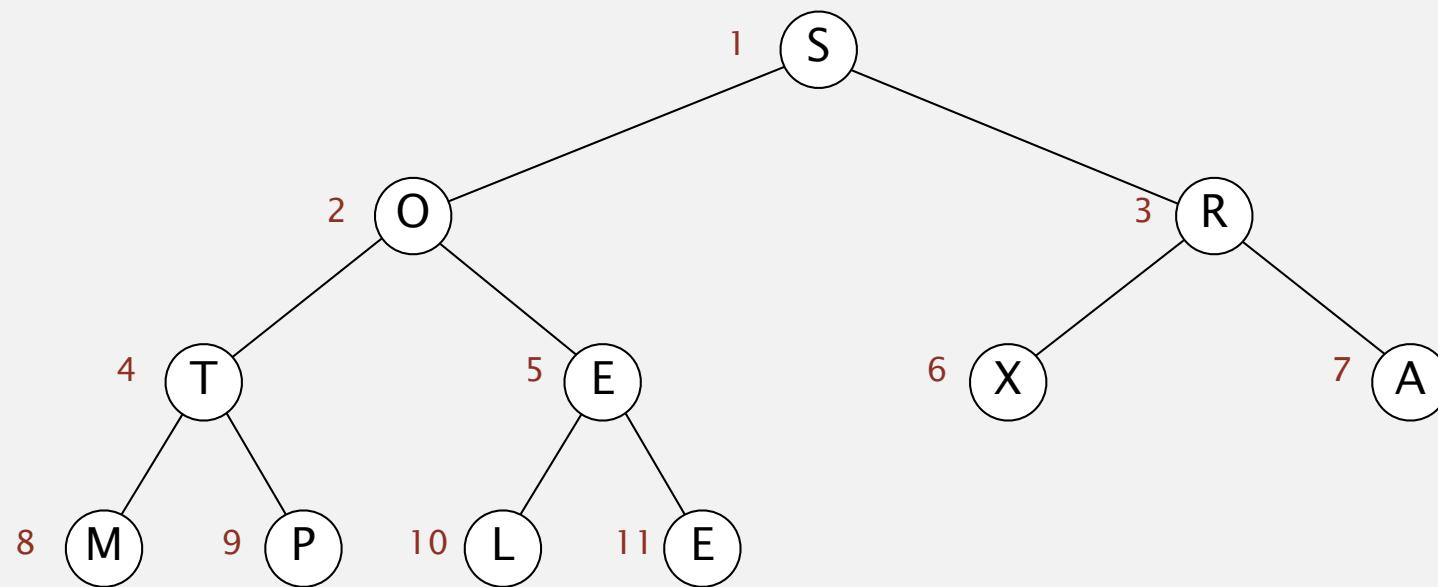
# Heapsort demo

Heap construction. Build max heap using bottom-up method.



we assume array entries are indexed 1 to N

array in arbitrary order



S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

# Heapsort demo

---

**Sortdown.** Repeatedly delete the largest remaining item.

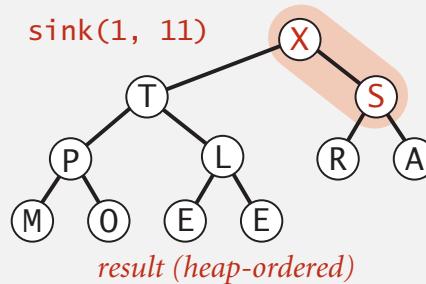
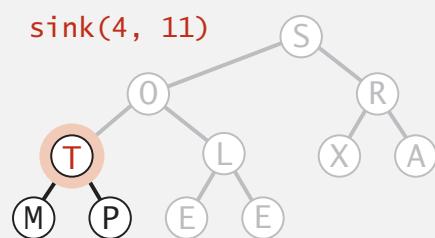
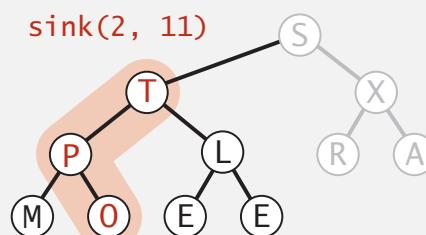
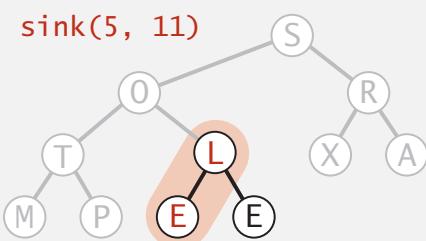
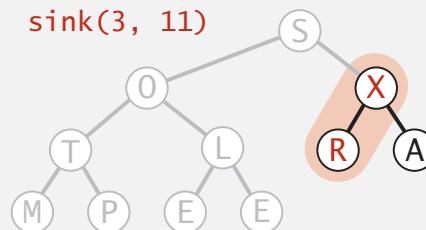
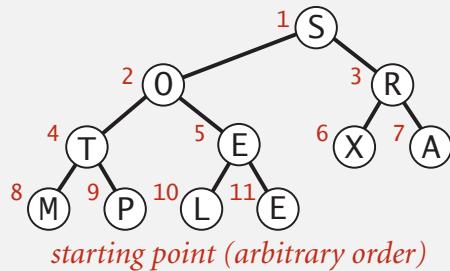
array in sorted order



# Heapsort: heap construction

First pass. Build heap using bottom-up method.

```
for (int k = N/2; k >= 1; k--)
    sink(a, k, N);
```

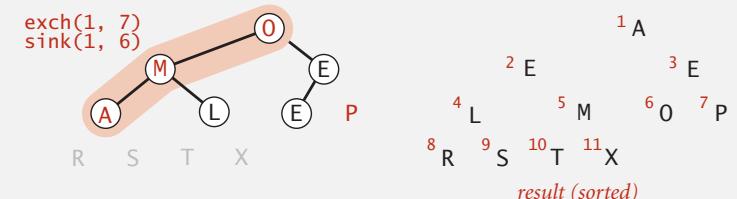
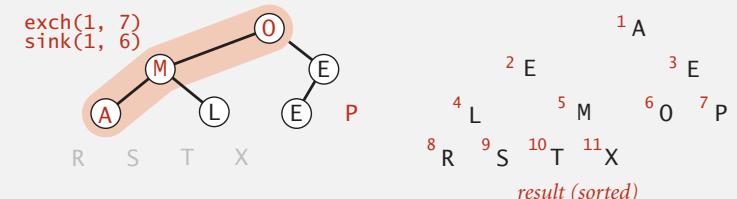
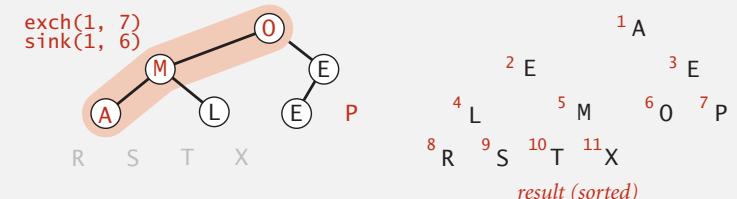
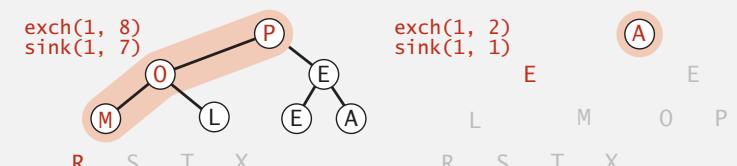
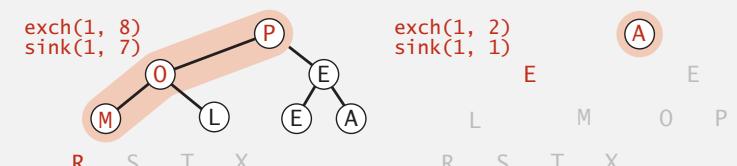
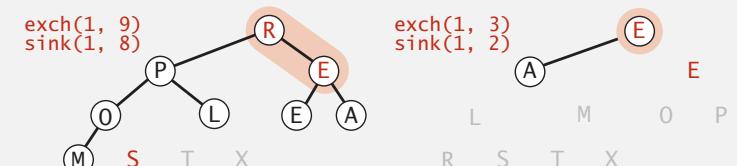
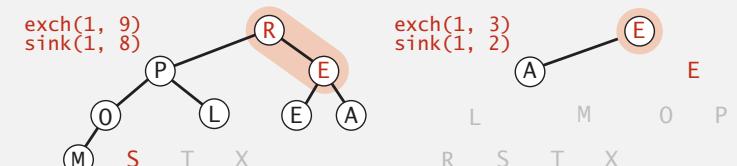
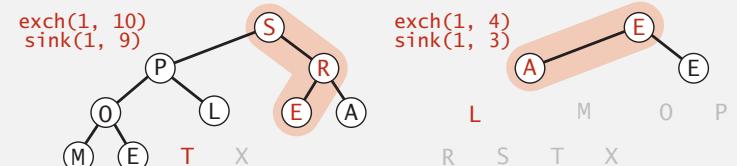
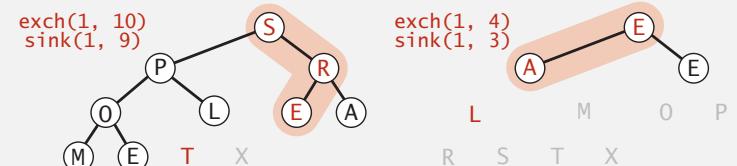
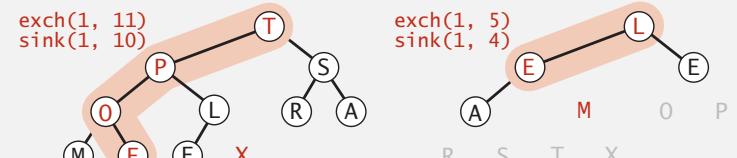
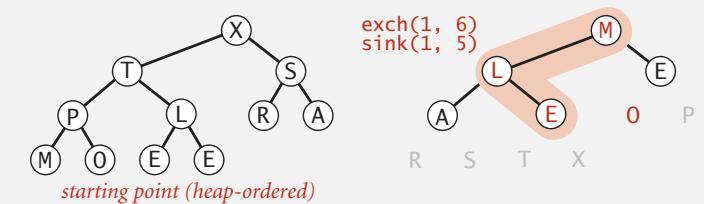


# Heapsort: sortdown

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[] a)
    {
        int N = a.length;
        for (int k = N/2; k >= 1; k--)
            sink(a, k, N);
        while (N > 1)
        {
            exch(a, 1, N);
            sink(a, 1, --N);
        }
    }

    private static void sink(Comparable[] a, int k, int N)
    { /* as before */ }

    private static boolean less(Comparable[] a, int i, int j)
    { /* as before */ }

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

but convert from  
1-based indexing to  
0-base indexing

# Heapsort: trace

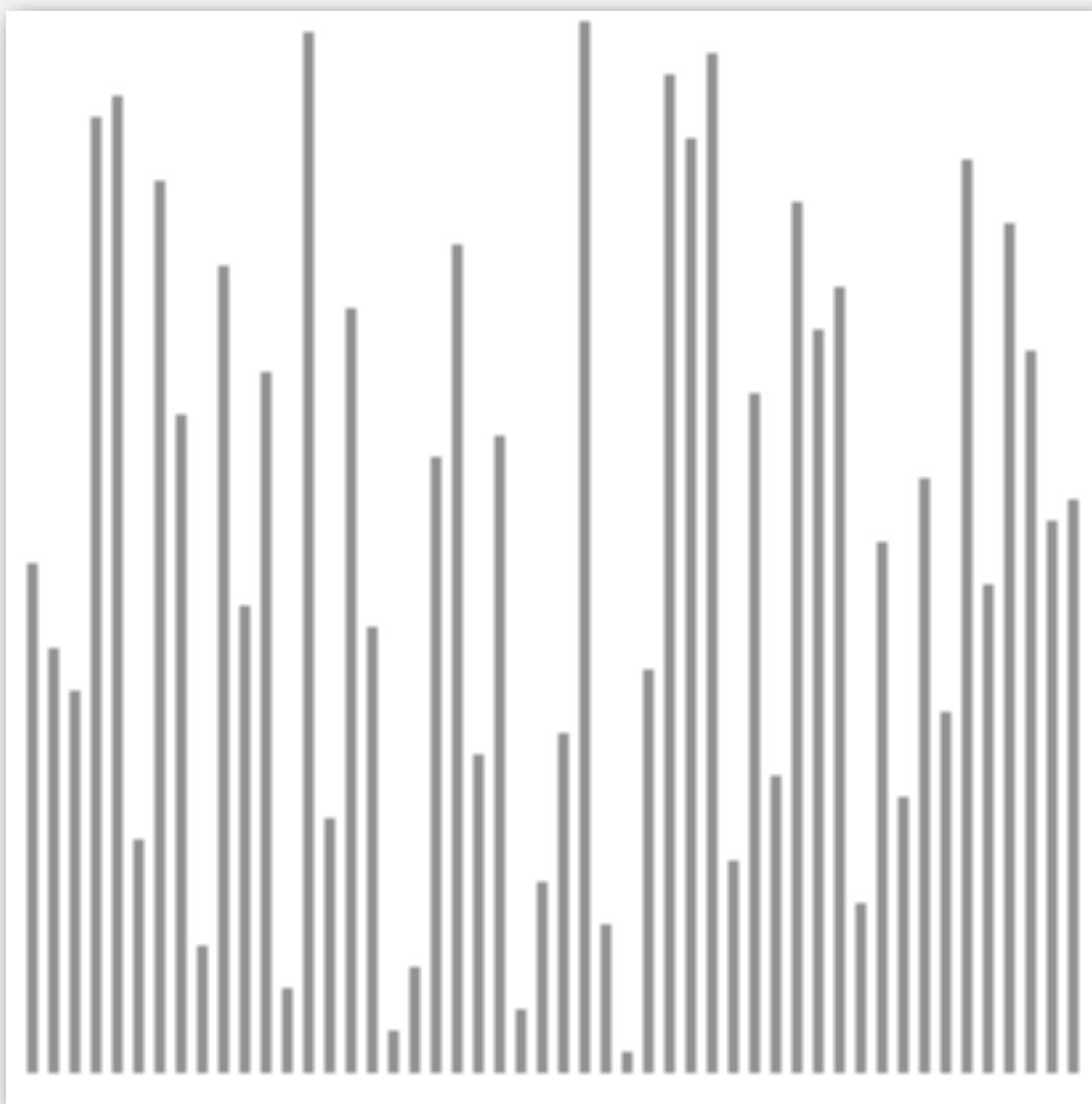
---

N	k	a[i]											
		0	1	2	3	4	5	6	7	8	9	10	11
<i>initial values</i>		S	O	R	T	E	X	A	M	P	L	E	
11	5	S	O	R	T	L	X	A	M	P	E	E	
11	4	S	O	R	T	L	X	A	M	P	E	E	
11	3	S	O	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  $\leq 2N$  compares and exchanges.

Proposition. Heapsort uses  $\leq 2N \lg N$  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

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

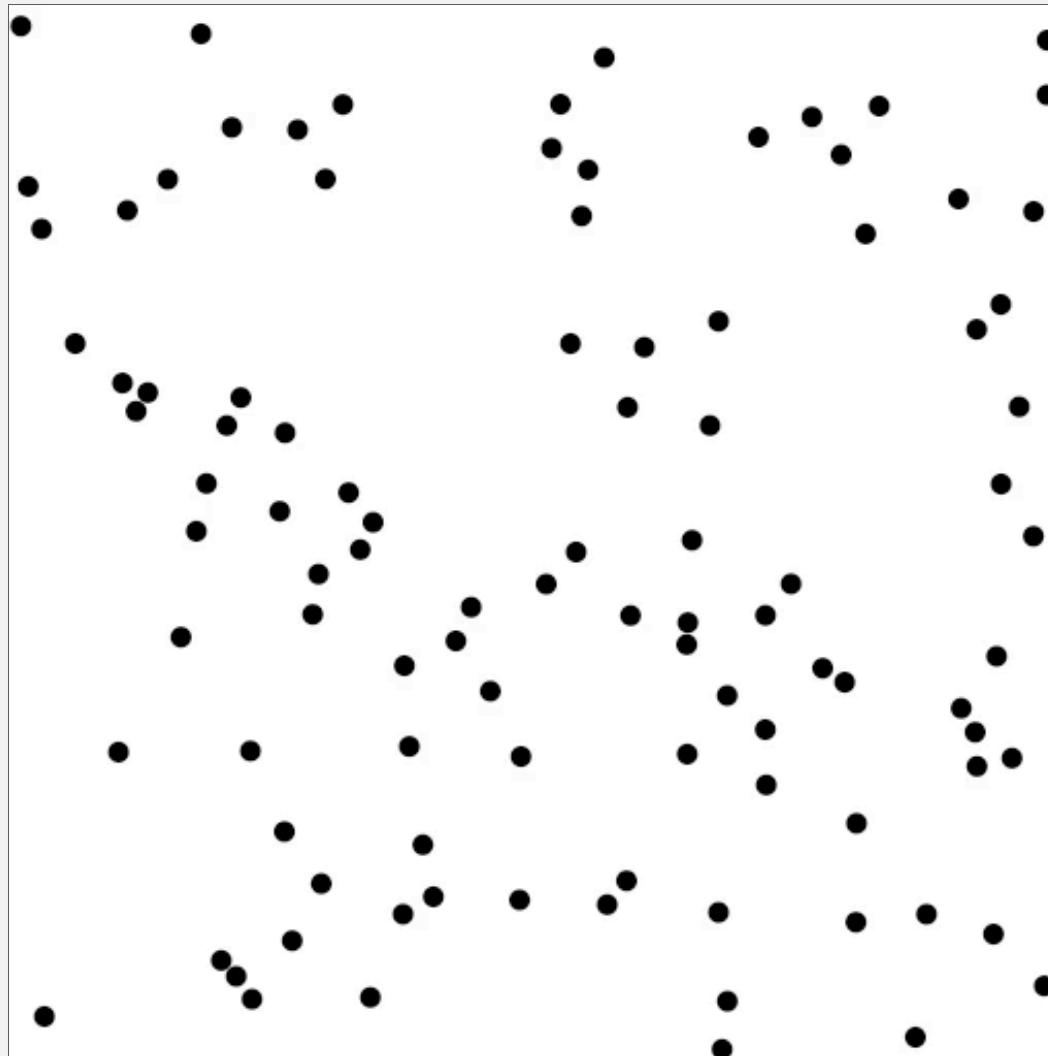
---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Molecular dynamics simulation of hard discs

---

**Goal.** Simulate the motion of  $N$  moving particles that behave according to the laws of elastic collision.



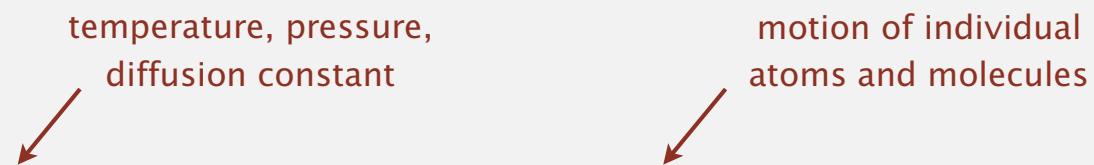
# Molecular dynamics simulation of hard discs

---

**Goal.** Simulate the motion of  $N$  moving particles that behave according to the laws of elastic collision.

## Hard disc model.

- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.



**Significance.** Relates macroscopic observables to microscopic dynamics.

- Maxwell-Boltzmann: distribution of speeds as a function of temperature.
- Einstein: explain Brownian motion of pollen grains.

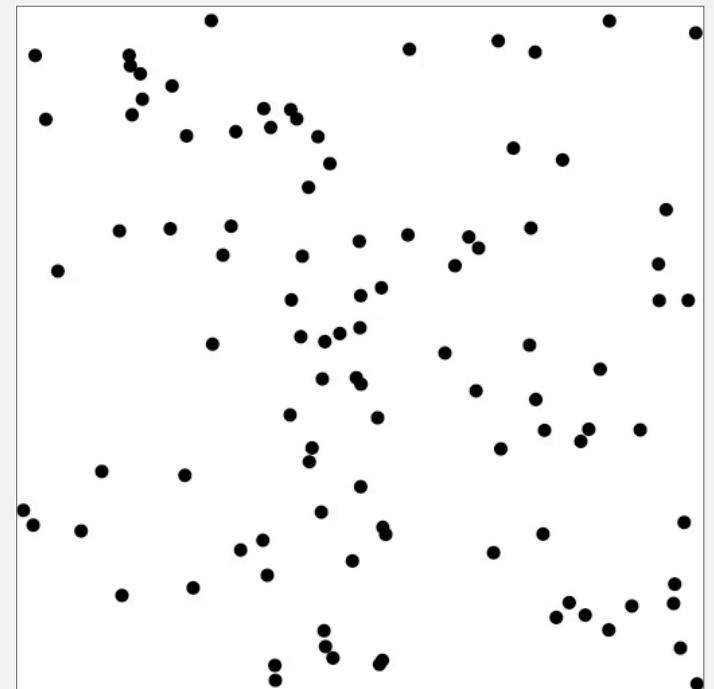
# Warmup: bouncing balls

Time-driven simulation.  $N$  bouncing balls in the unit square.

```
public class BouncingBalls
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Ball[] balls = new Ball[N];
        for (int i = 0; i < N; i++)
            balls[i] = new Ball();
        while(true)
        {
            StdDraw.clear();
            for (int i = 0; i < N; i++)
            {
                balls[i].move(0.5);
                balls[i].draw();
            }
            StdDraw.show(50);
        }
    }
}
```

main simulation loop

```
% java BouncingBalls 100
```



## Warmup: bouncing balls

```
public class Ball
{
    private double rx, ry;          // position
    private double vx, vy;          // velocity
    private final double radius;    // radius
    public Ball(...)
    { /* initialize position and velocity */ }

    public void move(double dt)
    {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }

    public void draw()
    { StdDraw.filledCircle(rx, ry, radius); }
}
```

check for collision with walls



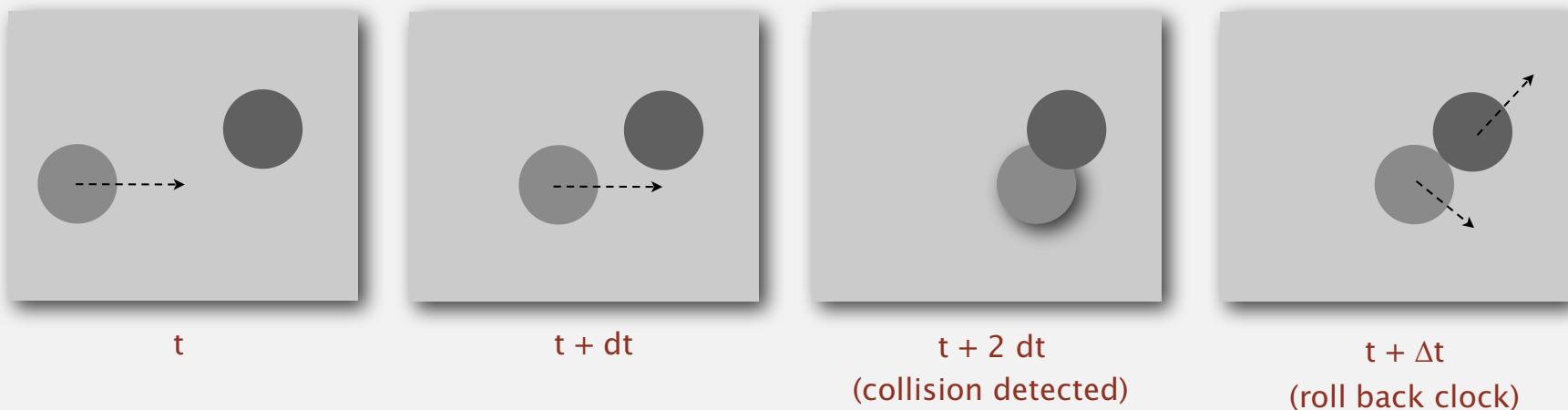
Missing. Check for balls colliding with each other.

- Physics problems: when? what effect?
- CS problems: which object does the check? too many checks?

# Time-driven simulation

---

- Discretize time in quanta of size  $dt$ .
- Update the position of each particle after every  $dt$  units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

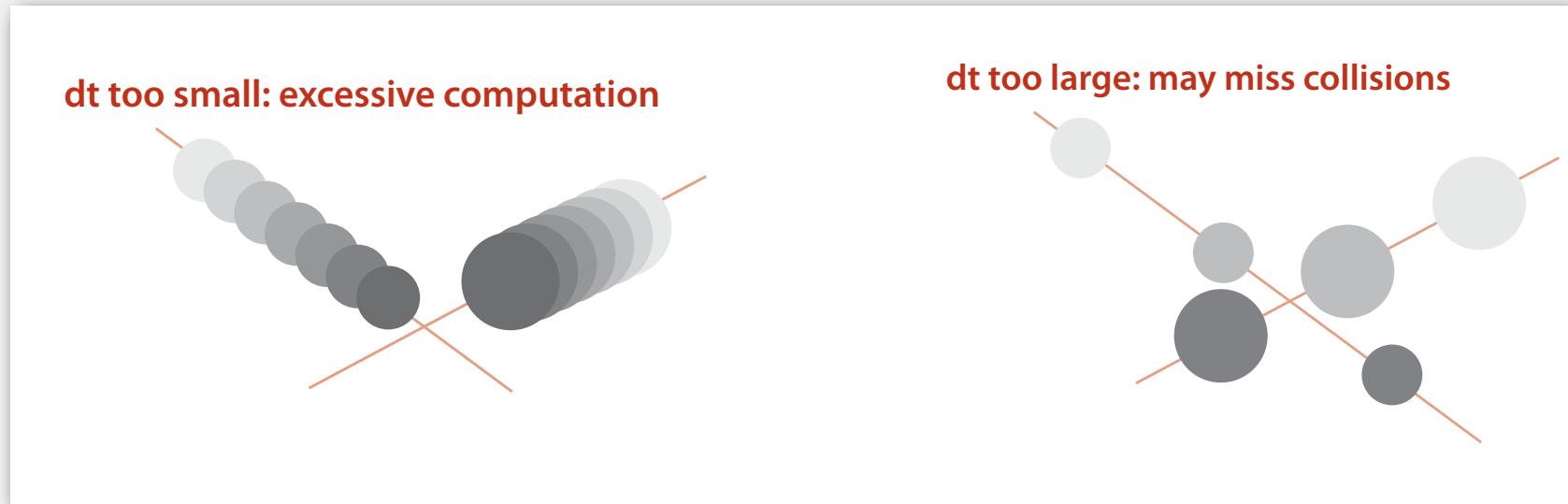


# Time-driven simulation

---

## Main drawbacks.

- $\sim N^2 / 2$  overlap checks per time quantum.
- Simulation is too slow if  $dt$  is very small.
- May miss collisions if  $dt$  is too large.  
(if colliding particles fail to overlap when we are looking)



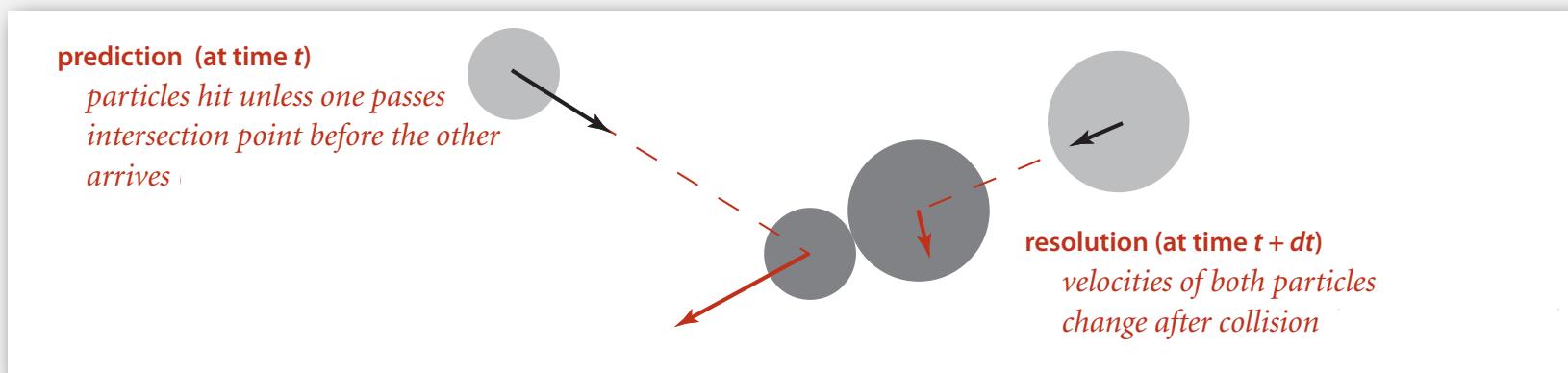
# Event-driven simulation

Change state only when something happens.

- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain PQ of collision events, prioritized by time.
- Remove the min = get next collision.

**Collision prediction.** Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

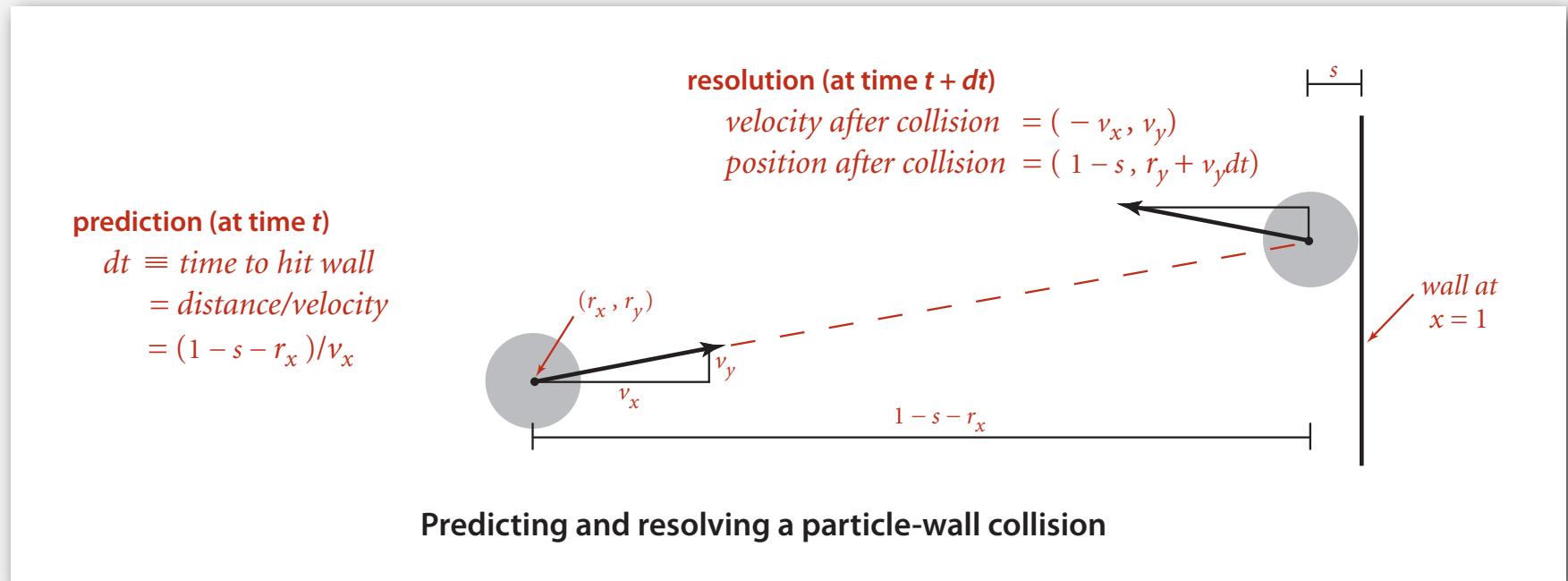
**Collision resolution.** If collision occurs, update colliding particle(s) according to laws of elastic collisions.



# Particle-wall collision

## Collision prediction and resolution.

- Particle of radius  $s$  at position  $(rx, ry)$ .
- Particle moving in unit box with velocity  $(vx, vy)$ .
- Will it collide with a vertical wall? If so, when?

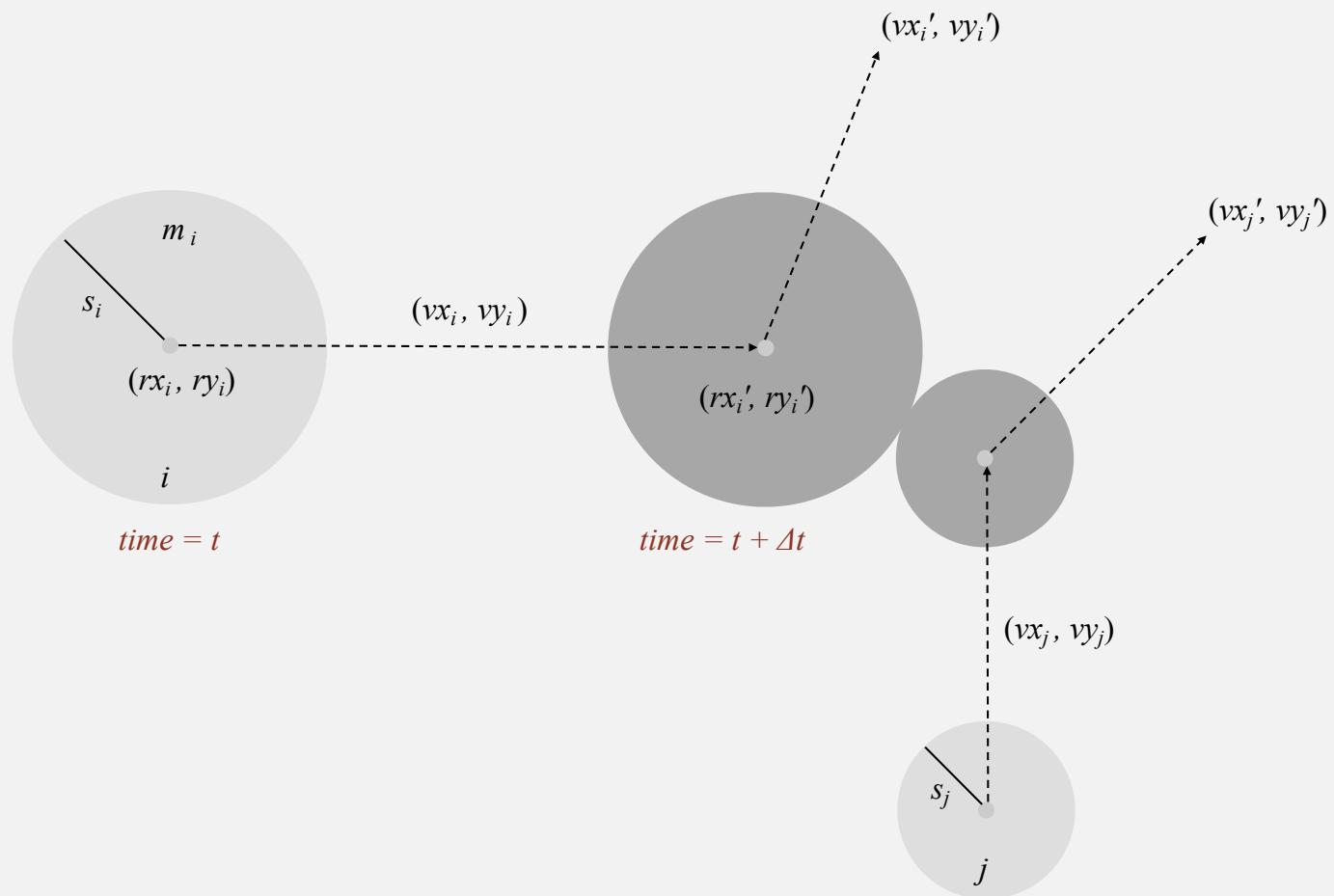


# Particle-particle collision prediction

---

## Collision prediction.

- Particle  $i$ : radius  $s_i$ , position  $(rx_i, ry_i)$ , velocity  $(vx_i, vy_i)$ .
- Particle  $j$ : radius  $s_j$ , position  $(rx_j, ry_j)$ , velocity  $(vx_j, vy_j)$ .
- Will particles  $i$  and  $j$  collide? If so, when?



# Particle-particle collision prediction

## Collision prediction.

- Particle  $i$ : radius  $s_i$ , position  $(rx_i, ry_i)$ , velocity  $(vx_i, vy_i)$ .
- Particle  $j$ : radius  $s_j$ , position  $(rx_j, ry_j)$ , velocity  $(vx_j, vy_j)$ .
- Will particles  $i$  and  $j$  collide? If so, when?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\ \infty & \text{if } d < 0 \\ -\frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise} \end{cases}$$

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v) (\Delta r \cdot \Delta r - \sigma^2) \quad \sigma = \sigma_i + \sigma_j$$

$$\Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j)$$

$$\Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j)$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$

$$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$$

$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$$

**Important note:** This is high-school physics, so we won't be testing you on it!

# Particle-particle collision resolution

**Collision resolution.** When two particles collide, how does velocity change?

$$\begin{aligned} vx_i' &= vx_i + Jx / m_i \\ vy_i' &= vy_i + Jy / m_i \\ vx_j' &= vx_j - Jx / m_j \\ vy_j' &= vy_j - Jy / m_j \end{aligned}$$

Newton's second law  
(momentum form)

$$Jx = \frac{J \Delta rx}{\sigma}, \quad Jy = \frac{J \Delta ry}{\sigma}, \quad J = \frac{2m_i m_j (\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

impulse due to normal force

(conservation of energy, conservation of momentum)

**Important note:** This is high-school physics, so we won't be testing you on it!

# Particle data type skeleton

```
public class Particle
{
    private double rx, ry;          // position
    private double vx, vy;          // velocity
    private final double radius;    // radius
    private final double mass;      // mass
    private int count;              // number of collisions

    public Particle(...) { }

    public void move(double dt) { }
    public void draw() { }

    public double timeToHit(Particle that) { }
    public double timeToHitVerticalWall() { }
    public double timeToHitHorizontalWall() { }

    public void bounceOff(Particle that) { }
    public void bounceOffVerticalWall() { }
    public void bounceOffHorizontalWall() { }

}
```

predict collision  
with particle or wall

resolve collision  
with particle or wall

# Particle-particle collision and resolution implementation

```
public double timeToHit(Particle that)
{
    if (this == that) return INFINITY;
    double dx = that.rx - this.rx, dy = that.ry - this.ry;
    double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    if( dvdr > 0) return INFINITY; ← no collision
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = this.radius + that.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

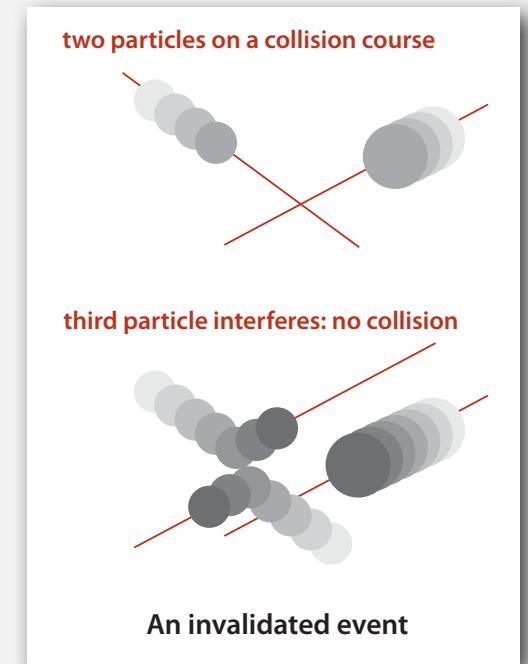
```
public void bounceOff(Particle that)
{
    double dx = that.rx - this.rx, dy = that.ry - this.ry;
    double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    double dist = this.radius + that.radius;
    double J = 2 * this.mass * that.mass * dvdr / ((this.mass + that.mass) * dist);
    double Jx = J * dx / dist;
    double Jy = J * dy / dist;
    this.vx += Jx / this.mass;
    this.vy += Jy / this.mass;
    that.vx -= Jx / that.mass;
    that.vy -= Jy / that.mass;
    this.count++;
    that.count++;     Important note: This is high-school physics, so we won't be testing you on it!
}
```

# Collision system: event-driven simulation main loop

## Initialization.

- Fill PQ with all potential particle-wall collisions.
- Fill PQ with all potential particle-particle collisions.

“potential” since collision may not happen if some other collision intervenes



## Main loop.

- Delete the impending event from PQ (min priority =  $t$ ).
- If the event has been invalidated, ignore it.
- Advance all particles to time  $t$ , on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

# Event data type

## Conventions.

- Neither particle null  $\Rightarrow$  particle-particle collision.
- One particle null  $\Rightarrow$  particle-wall collision.
- Both particles null  $\Rightarrow$  redraw event.

```
private class Event implements Comparable<Event>
{
    private double time;          // time of event
    private Particle a, b;        // particles involved in event
    private int countA, countB;   // collision counts for a and b

    public Event(double t, Particle a, Particle b) { }           ← create event

    public int compareTo(Event that)
    {   return this.time - that.time;   }                         ← ordered by time

    public boolean isValid()
    {   }
}
```

invalid if intervening collision

## Collision system implementation: skeleton

```
public class CollisionSystem
{
    private MinPQ<Event> pq;           // the priority queue
    private double t = 0.0;              // simulation clock time
    private Particle[] particles;       // the array of particles

    public CollisionSystem(Particle[] particles) { }

    private void predict(Particle a)      add to PQ all particle-wall and particle-
    {                                     -particle collisions involving this particle
        if (a == null) return;
        for (int i = 0; i < N; i++)
        {
            double dt = a.timeToHit(particles[i]);
            pq.insert(new Event(t + dt, a, particles[i]));
        }
        pq.insert(new Event(t + a.timeToHitVerticalWall() , a, null));
        pq.insert(new Event(t + a.timeToHitHorizontalWall(), null, a));
    }

    private void redraw() { }

    public void simulate() { /* see next slide */ }
}
```

# Collision system implementation: main event-driven simulation loop

```
public void simulate()
{
    pq = new MinPQ<Event>();
    for(int i = 0; i < N; i++) predict(particles[i]);
    pq.insert(new Event(0, null, null));
```

initialize PQ with collision events and redraw event

```
while(!pq.isEmpty())
{
    Event event = pq.delMin();
    if(!event.isValid()) continue;
    Particle a = event.a;
    Particle b = event.b;
```

get next event

```
    for(int i = 0; i < N; i++)
        particles[i].move(event.time - t);
    t = event.time;
```

update positions and time

```
    if      (a != null && b != null) a.bounceOff(b);
    else if (a != null && b == null) a.bounceOffVerticalWall()
    else if (a == null && b != null) b.bounceOffHorizontalWall();
    else if (a == null && b == null) redraw();
```

process event

```
    predict(a);
    predict(b);
```

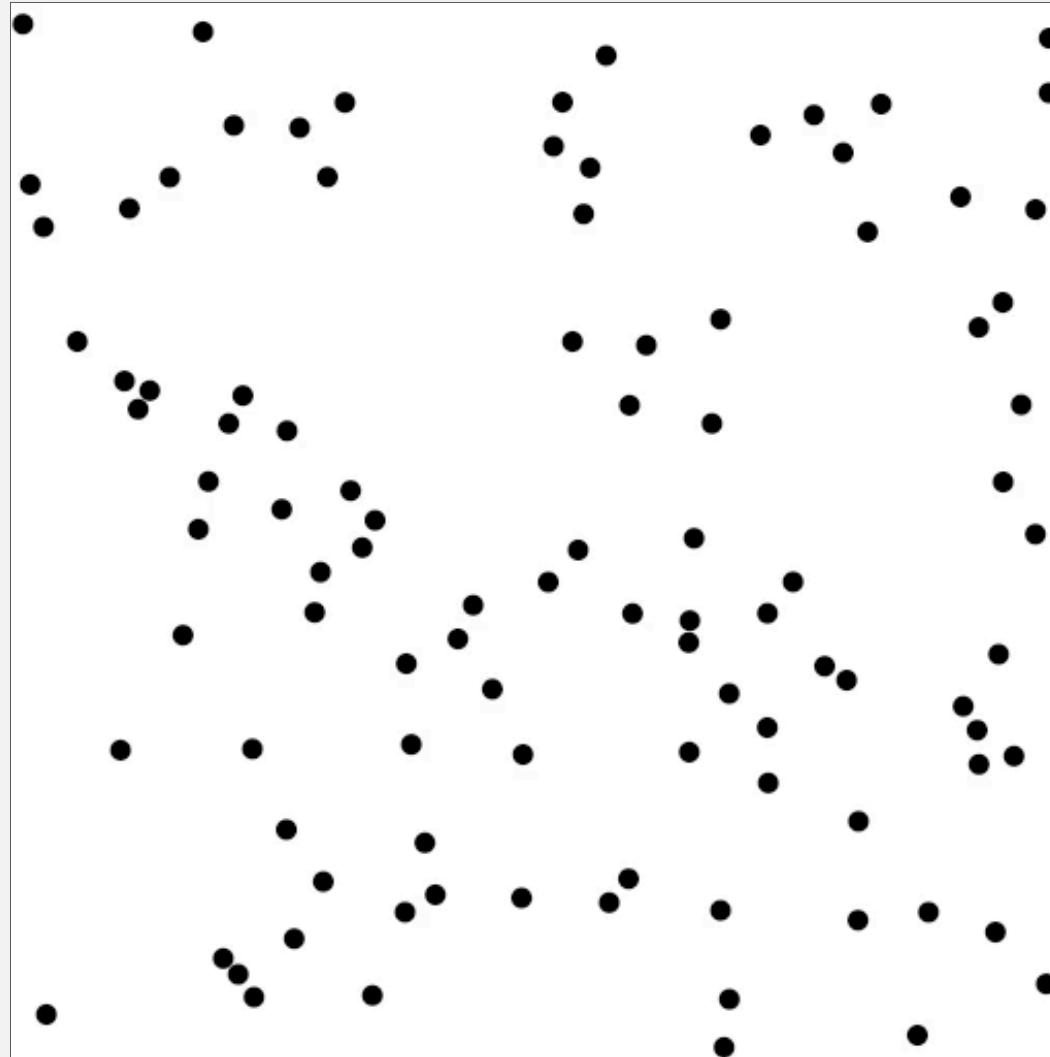
predict new events based on changes

```
}
```

# Particle collision simulation example 1

---

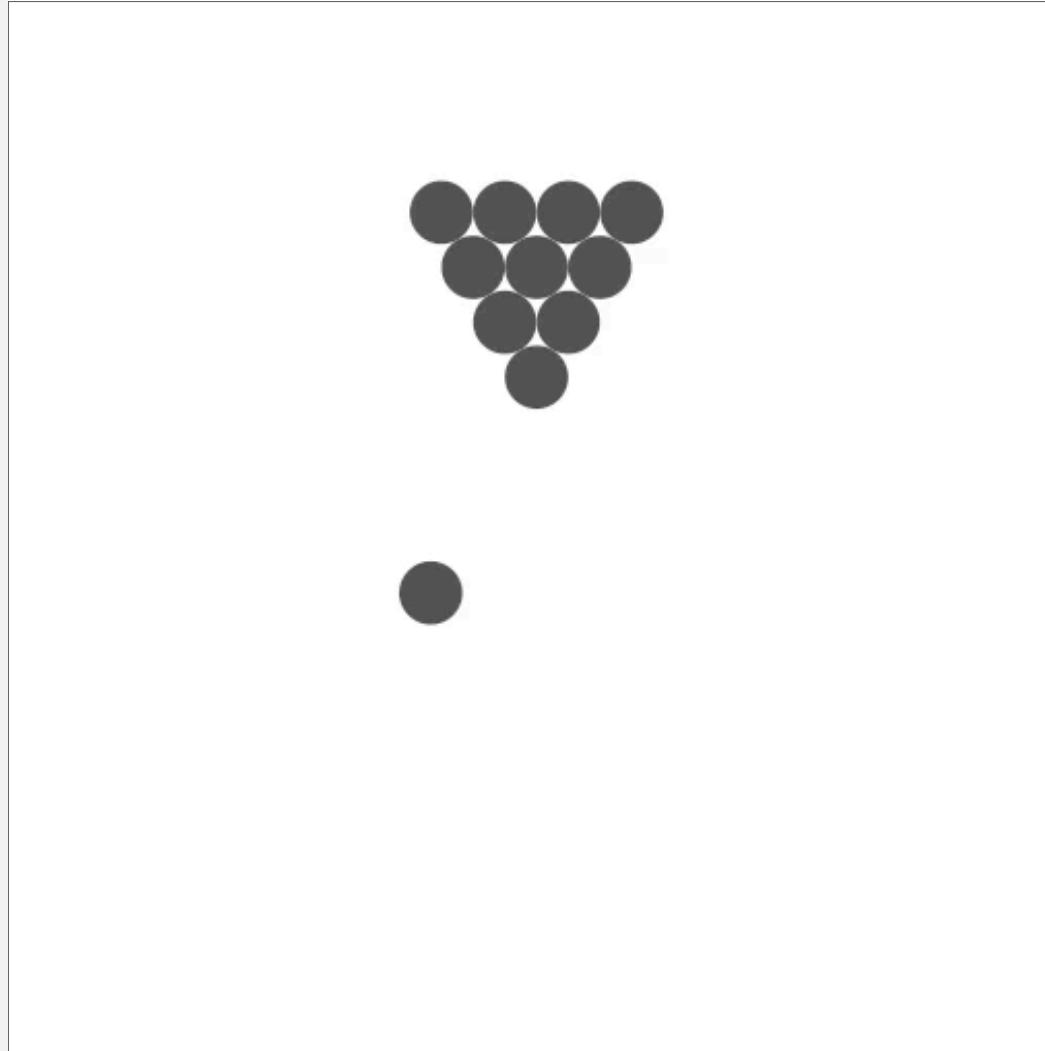
```
% java CollisionSystem 100
```



## Particle collision simulation example 2

---

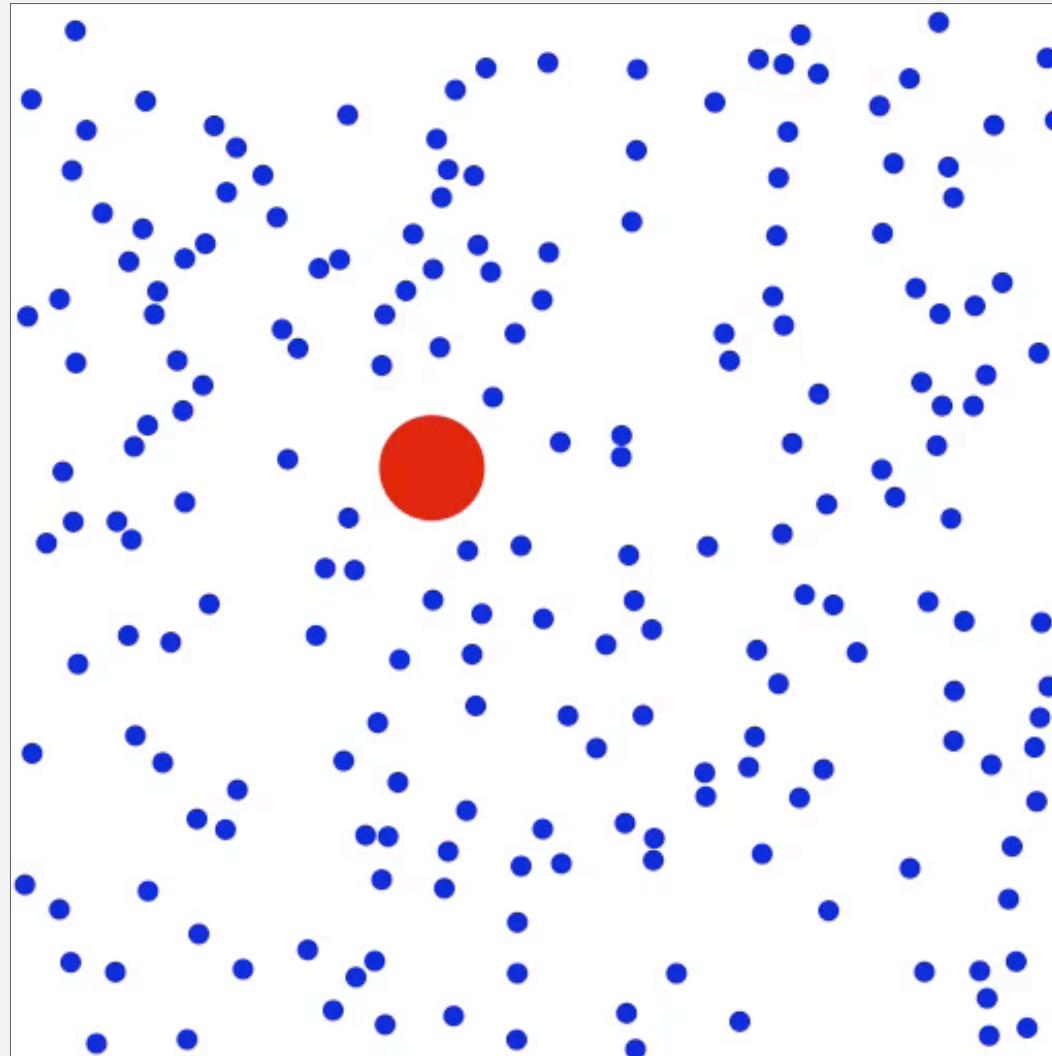
```
% java CollisionSystem < billiards.txt
```



## Particle collision simulation example 3

---

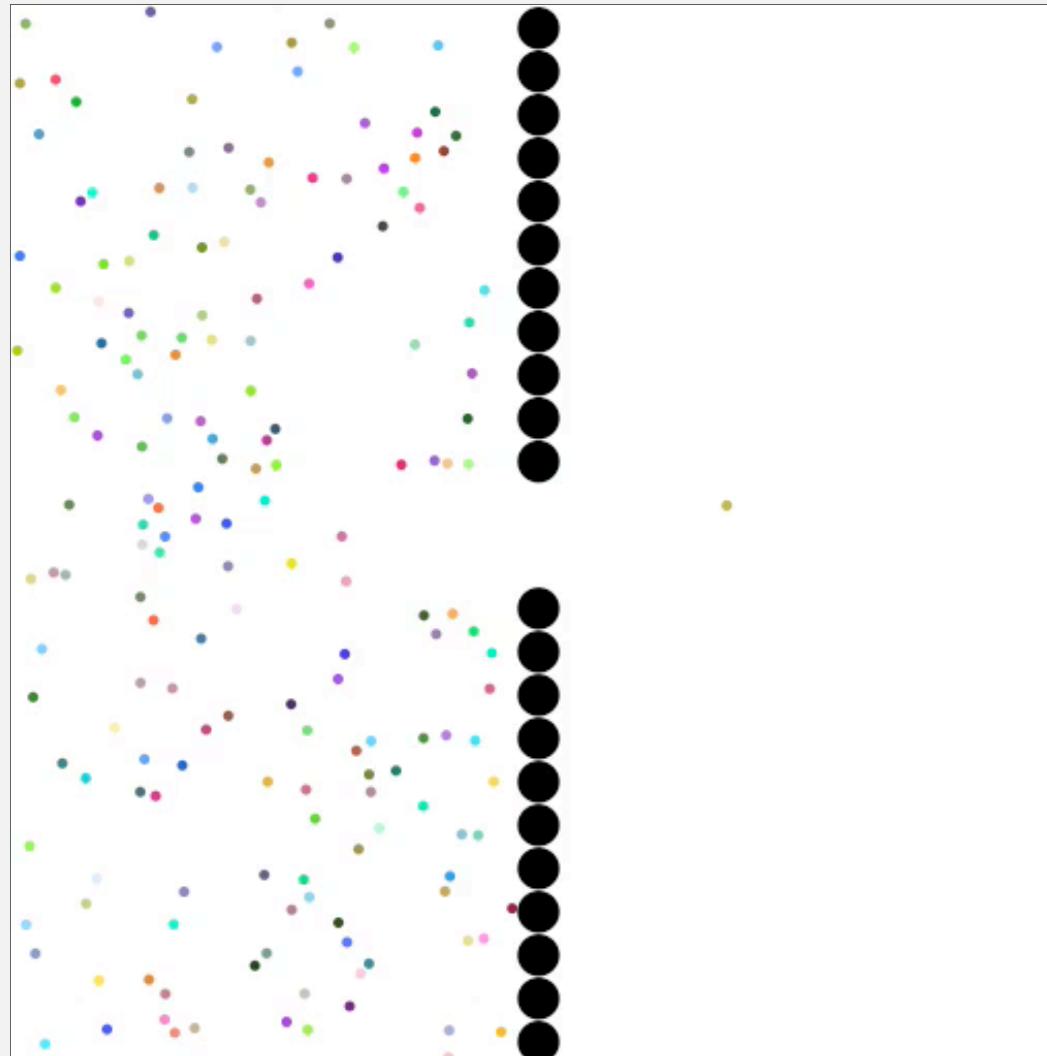
```
% java CollisionSystem < brownian.txt
```



# Particle collision simulation example 4

---

```
% java CollisionSystem < diffusion.txt
```



# Algorithms

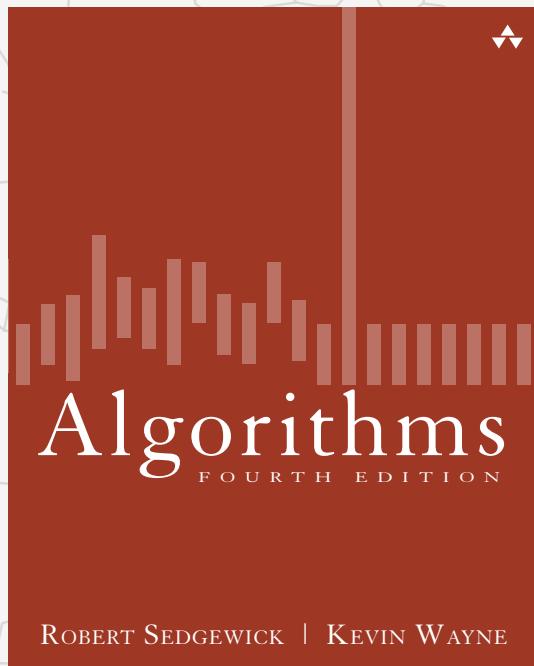
ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*



<http://algs4.cs.princeton.edu>

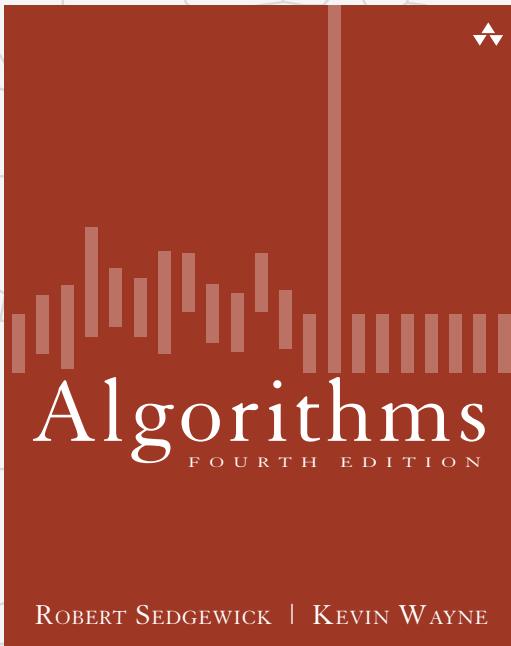
## 2.4 PRIORITY QUEUES

---

- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

## 3.1 SYMBOL TABLES

---

- ▶ API
- ▶ *elementary implementations*
- ▶ *ordered operations*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.1 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 domain name with specified IP address.
- Given domain name, find corresponding IP address.

domain name	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	domain name	IP address
reverse DNS	find domain name	IP address	domain name
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: Integer, Double, String, 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`, `java.io.File`, ...

**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.

typically unsafe to use equals() with inheritance  
(would violate symmetry)

```
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()) ← objects must be in the same class
            return false;                  (religion: getClass() vs. instanceof)
        Date that = (Date) y;
        if (this.day != that.day) return false; ← cast is guaranteed to succeed
        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.  
Why? Experts still debate.

# 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.
- 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**

A	8
C	4
E	12
H	5
L	11
M	9
P	10
R	3
S	0
X	7

<b>keys</b>	S	E	A	R	C	H	E	X	A	M	P	L	E
<b>values</b>	0	1	2	3	4	5	6	7	8	9	10	11	12

## 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 update frequency
            if (word.length() < minlen) continue; ← ignore short strings
            if (!st.contains(word)) st.put(word, 1);
            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; ← print a string with max freq
        StdOut.println(max + " " + st.get(max));
    }
}
```

create ST

read string and update frequency

print a string with max freq

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.1 SYMBOL TABLES

---

▶ API

▶ *elementary implementations*

▶ *ordered operations*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.1 SYMBOL TABLES

---

▶ API

▶ *elementary implementations*

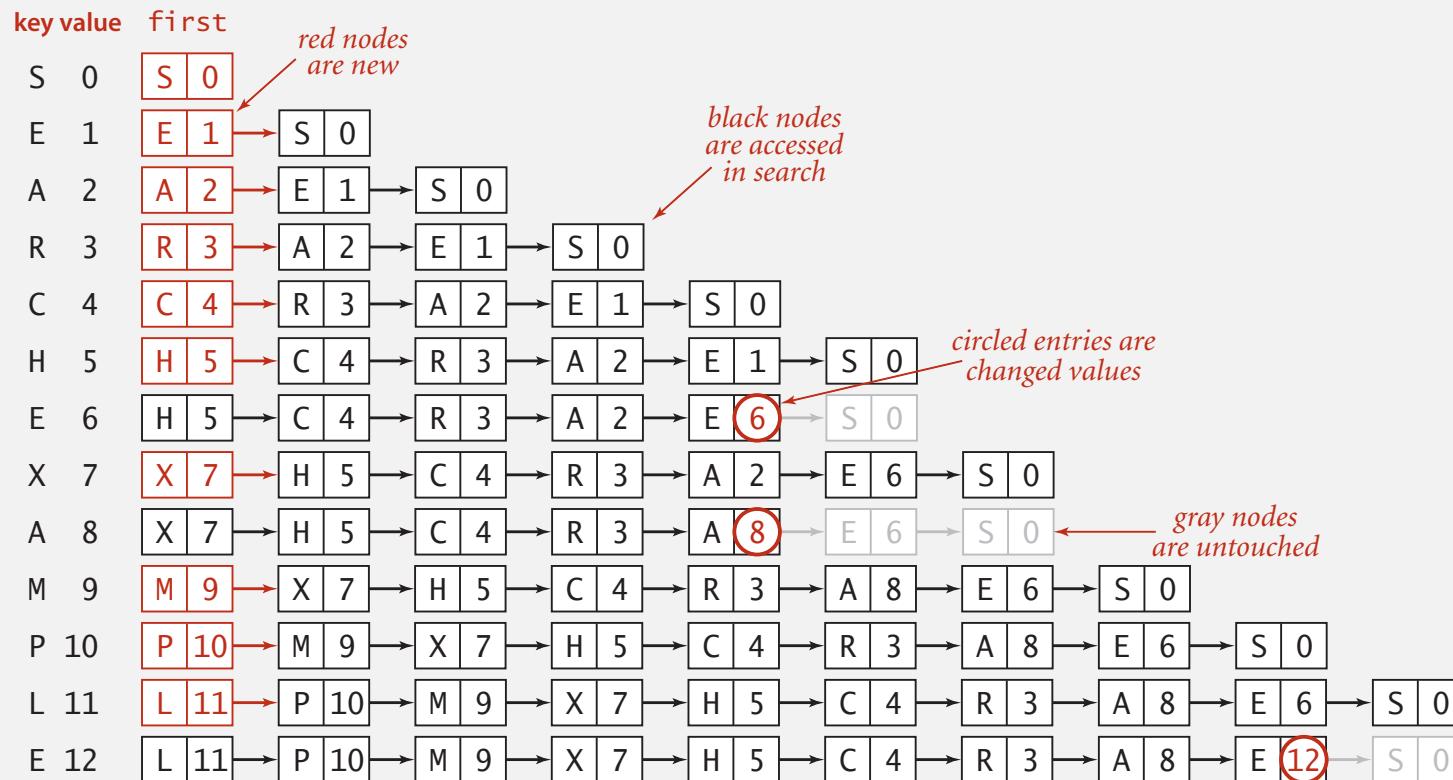
▶ *ordered operations*

# 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.



Trace of linked-list ST implementation for standard indexing client

# 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	equals()

**Challenge.** Efficient implementations of both search and insert.

# Binary search in an ordered array

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	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: trace of standard indexing client

Problem. To insert, need to shift all greater keys over.

		keys[]										vals[]										
key	value	0	1	2	3	4	5	6	7	8	9	N	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

# 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	equals()
binary search (ordered array)	log N	N	log N	N / 2	yes	compareTo()

**Challenge.** Efficient implementations of both search and insert.

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.1 SYMBOL TABLES

---

▶ API

▶ *elementary implementations*

▶ *ordered operations*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.1 SYMBOL TABLES

---

- ▶ API
- ▶ *elementary implementations*
- ▶ *ordered operations*

## Examples of ordered symbol table API

---

	keys	values
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		

# 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 / delete	N	N
min / max	N	1
floor / ceiling	N	$\lg N$
rank	N	$\lg N$
select	N	1
ordered iteration	$N \lg N$	N

order of growth of the running time for ordered symbol table operations

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

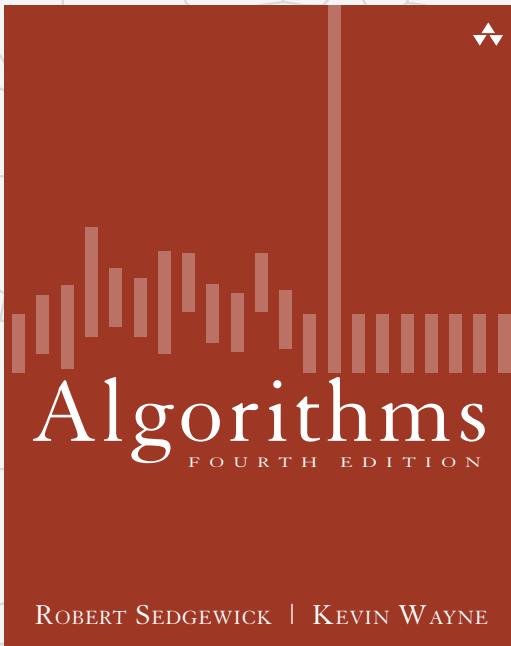
## 3.1 SYMBOL TABLES

---

- ▶ API
- ▶ *elementary implementations*
- ▶ *ordered operations*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

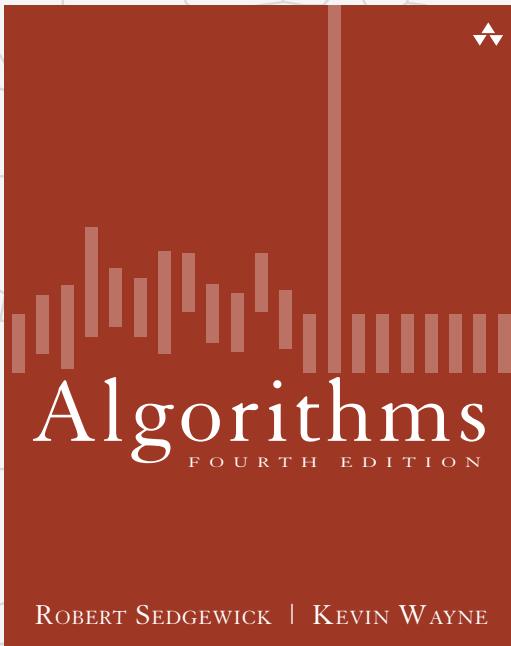


<http://algs4.cs.princeton.edu>

## 3.1 SYMBOL TABLES

---

- ▶ API
- ▶ *elementary implementations*
- ▶ *ordered operations*



<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

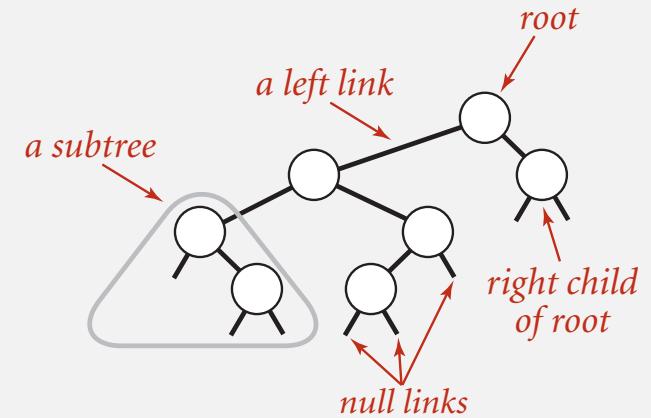
# Binary search trees

---

**Definition.** A BST is a binary tree in symmetric order.

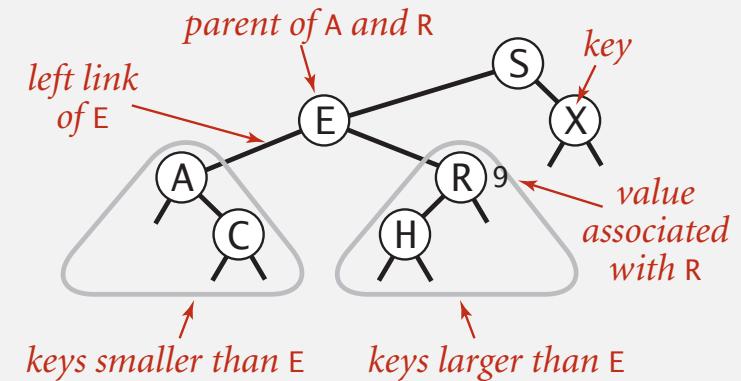
A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).



**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.



# BST representation in Java

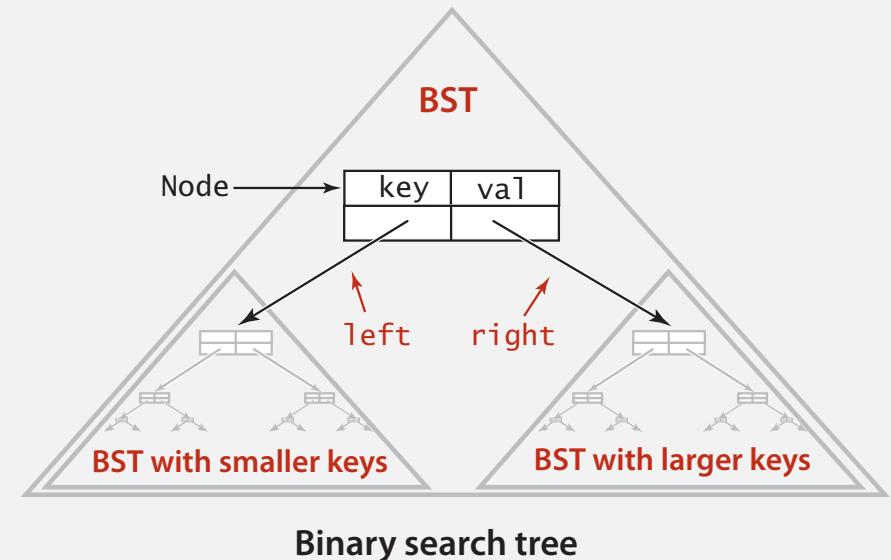
Java definition. A BST is a reference to a root Node.

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

# BST implementation (skeleton)

---

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;                                ← root of BST

    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 */ }

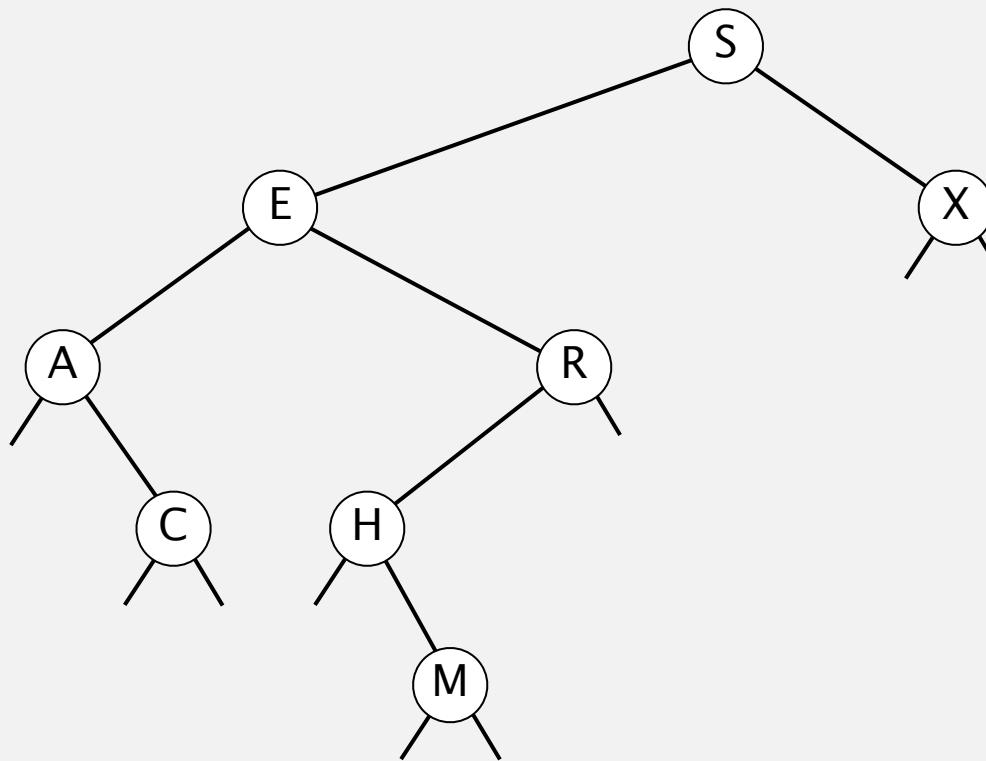
}
```

# Binary search tree demo

---

Search. If less, go left; if greater, go right; if equal, search hit.

successful search for H

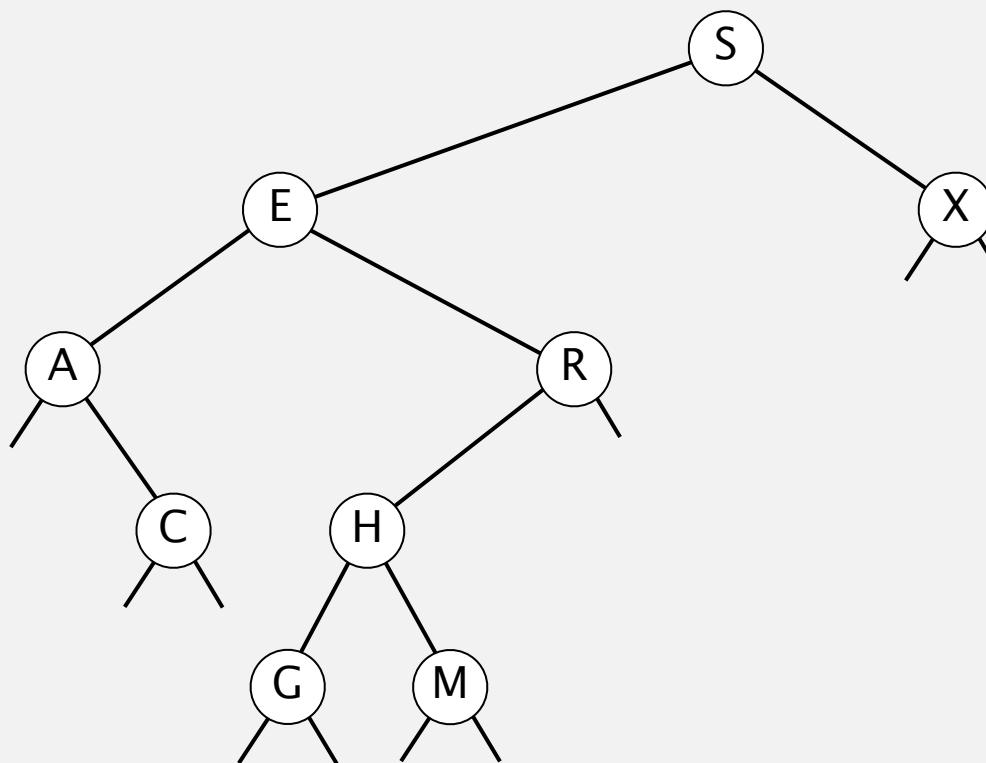


# Binary search tree demo

---

Insert. If less, go left; if greater, go right; if null, insert.

insert G



## 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 1 + depth of node.

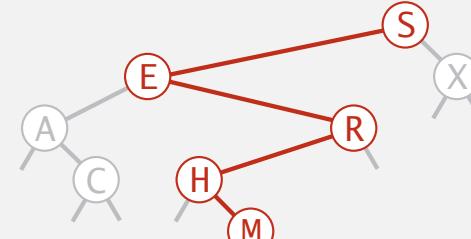
# BST insert

**Put.** Associate value with key.

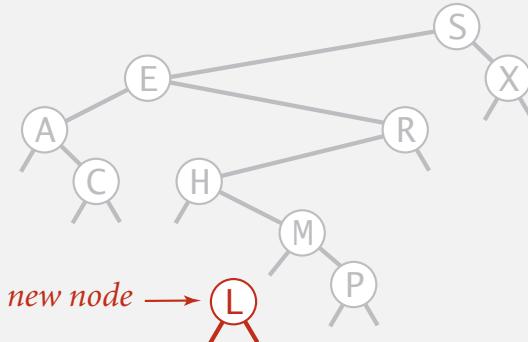
Search for key, then two cases:

- Key in tree  $\Rightarrow$  reset value.
- Key not in tree  $\Rightarrow$  add new node.

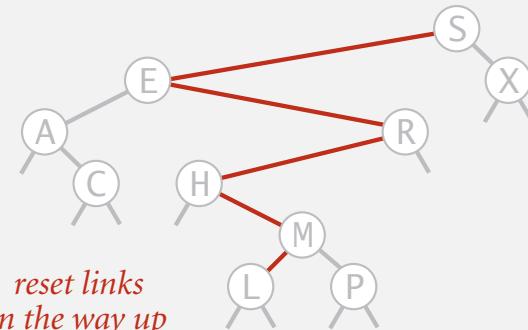
inserting L



search for L ends  
at this null link



create new node → (L)



reset links  
on the way up

Insertion into a BST

# 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;
}
```

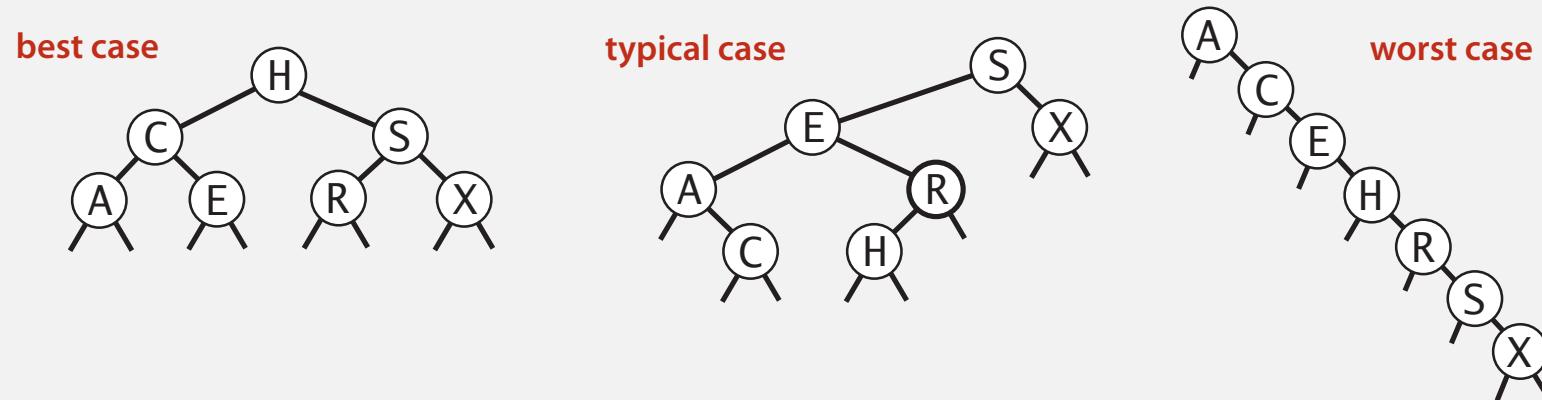
concise, but tricky,  
recursive code;  
read carefully!

**Cost.** Number of compares is equal to 1 + depth of node.

## Tree shape

---

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to  $1 + \text{depth of node}$ .

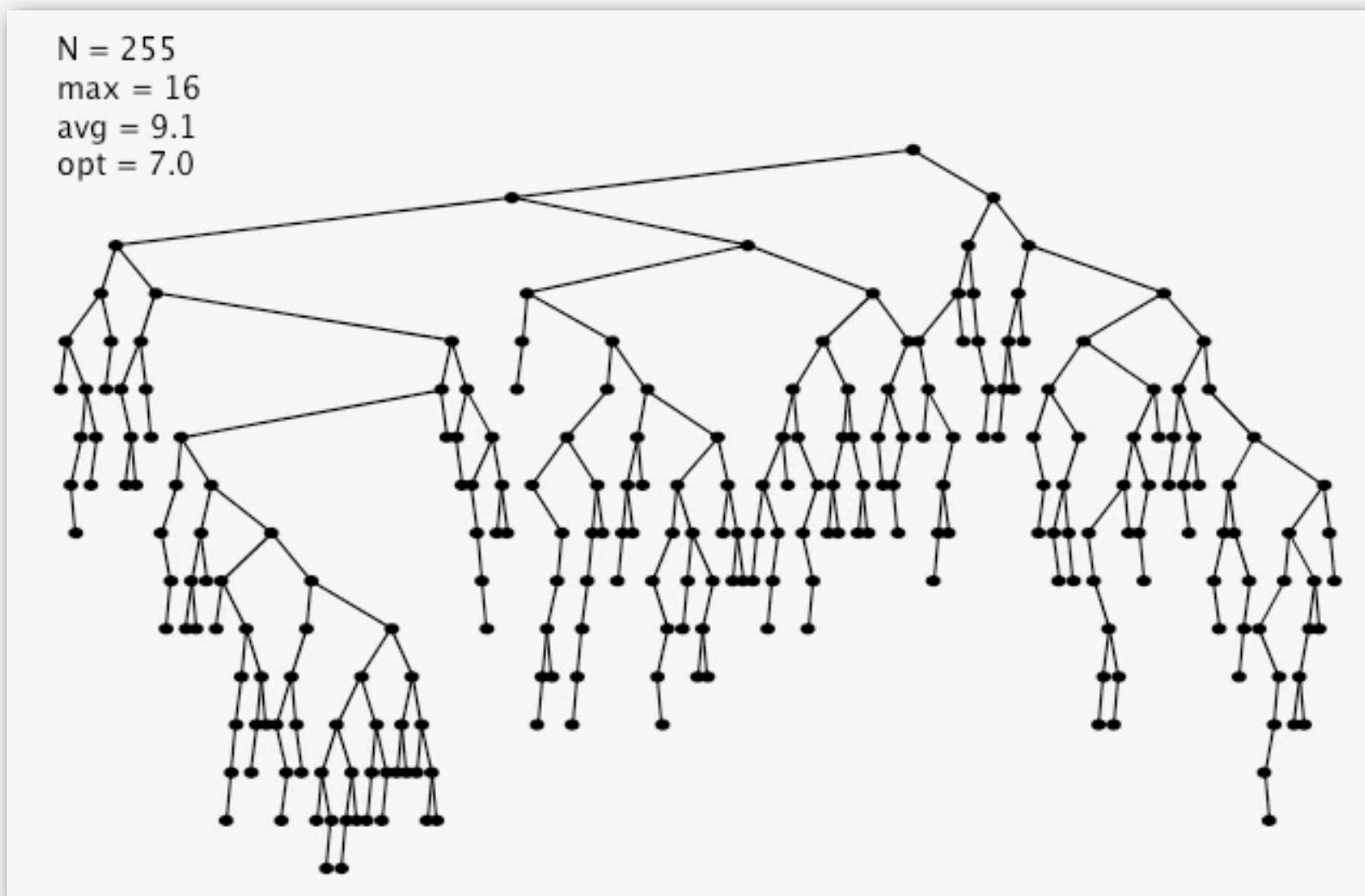


**Remark.** Tree shape depends on order of insertion.

## BST insertion: random order visualization

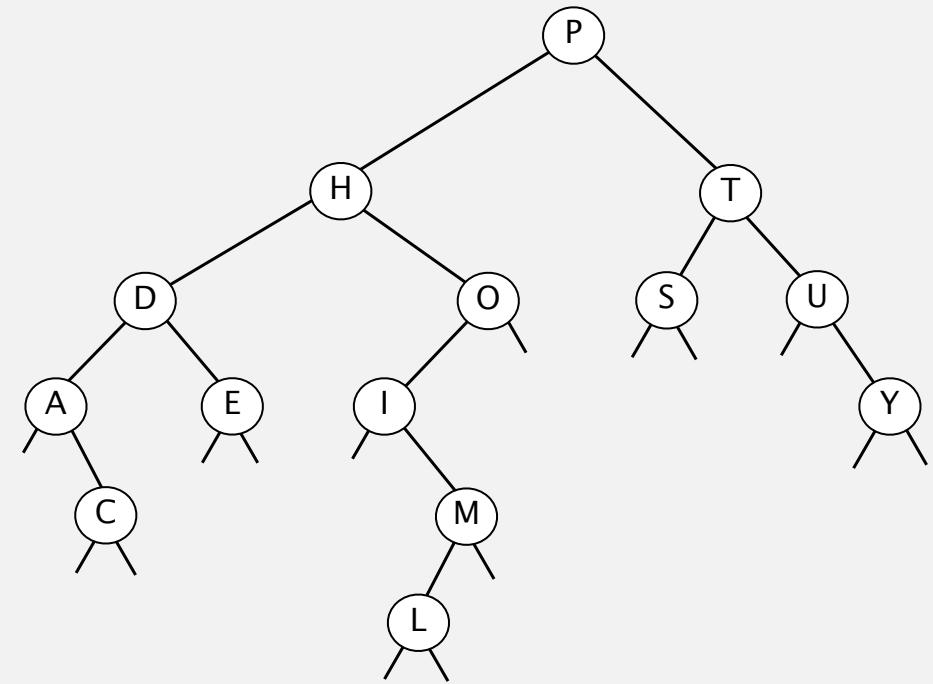
---

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	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  $\sim 2 \ln N$ .

**Pf.** 1-1 correspondence with quicksort partitioning.

**Proposition.** [Reed, 2003] If  $N$  distinct keys are inserted in random order, expected height of tree is  $\sim 4.311 \ln N$ .

**How Tall is a Tree?**

Bruce Reed  
CNRS, Paris, France  
[reed@moka.ccr.jussieu.fr](mailto:reed@moka.ccr.jussieu.fr)

**ABSTRACT**

Let  $H_n$  be the height of a random binary search tree on  $n$  nodes. We show that there exists constants  $\alpha = 4.31107\dots$  and  $\beta = 1.95\dots$  such that  $\mathbf{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$ . We also show that  $\text{Var}(H_n) = O(1)$ .

**But...** Worst-case height is  $N$ .  
(exponentially small chance when keys are inserted in random order)

# 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	$1.39 \lg N$	$1.39 \lg N$	next	<code>compareTo()</code>

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

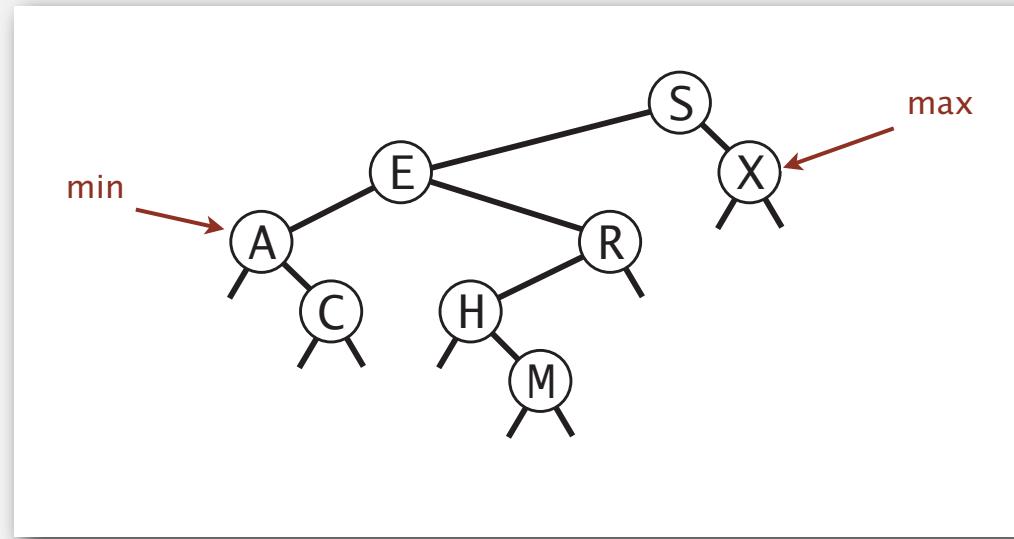
- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# Minimum and maximum

---

Minimum. Smallest key in table.

Maximum. Largest key in table.



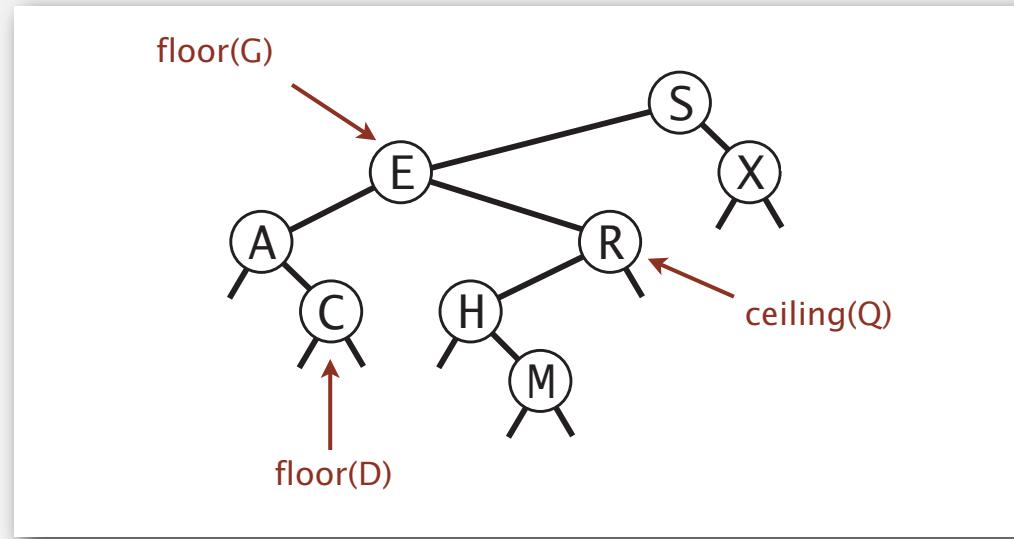
Q. How to find the min / max?

# Floor and ceiling

---

Floor. Largest key  $\leq$  a given key.

Ceiling. Smallest key  $\geq$  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]

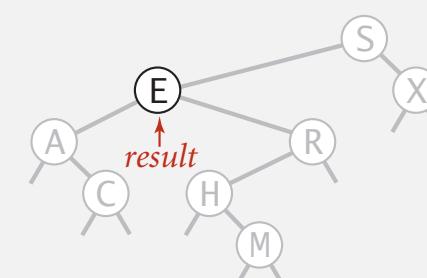
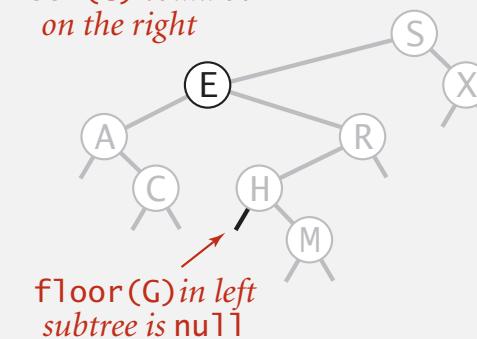
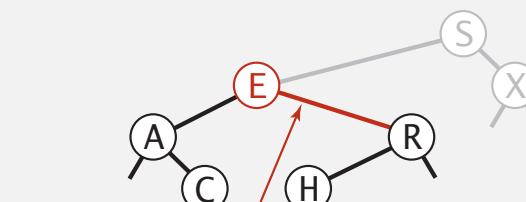
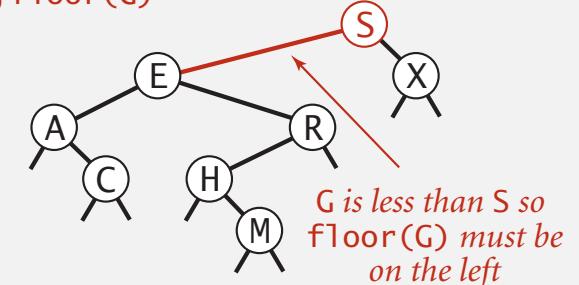
The floor of  $k$  is in the left subtree.

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  $\text{floor}(G)$



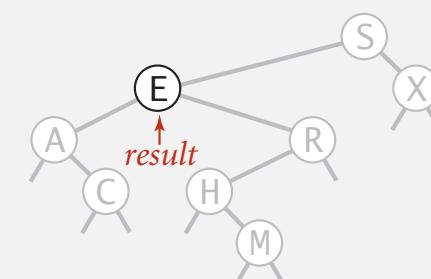
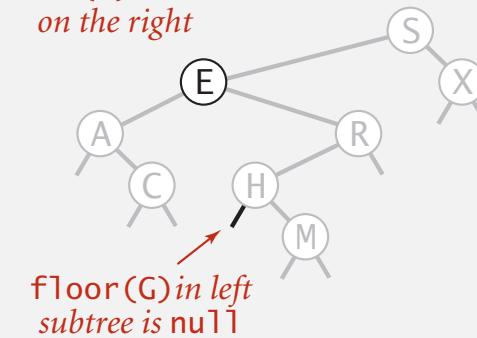
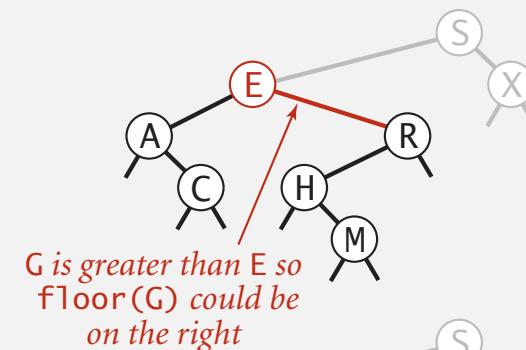
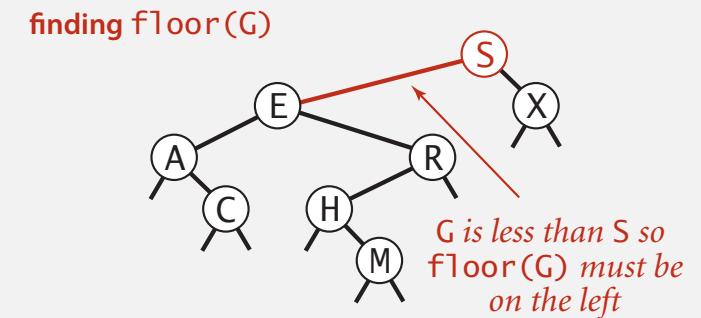
# Computing the floor

```
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);

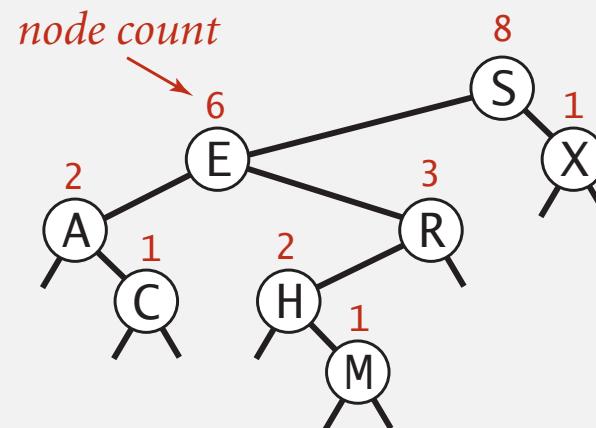
    Node t = floor(x.right, key);
    if (t != null) return t;
    else           return x;
}
```



## 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 count;
}
```

number of nodes in subtree

```
public int size()
{   return size(root); }
```

```
private int size(Node x)
{
    if (x == null) return 0;
    return x.count; }
```

ok to call  
when x is null

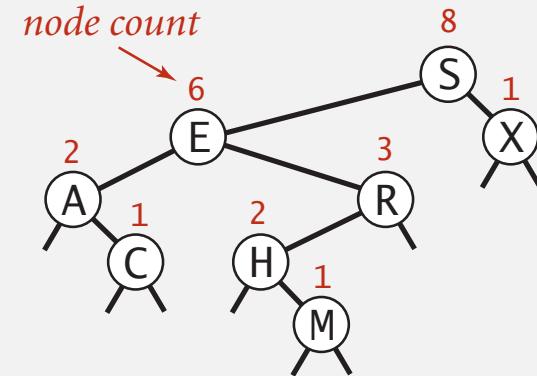
```
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else if (cmp == 0) x.val  = val;
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

# Rank

---

Rank. How many keys  $< k$ ?

Easy recursive algorithm (3 cases!)



```
public int rank(Key key)
{   return rank(key, root);  }

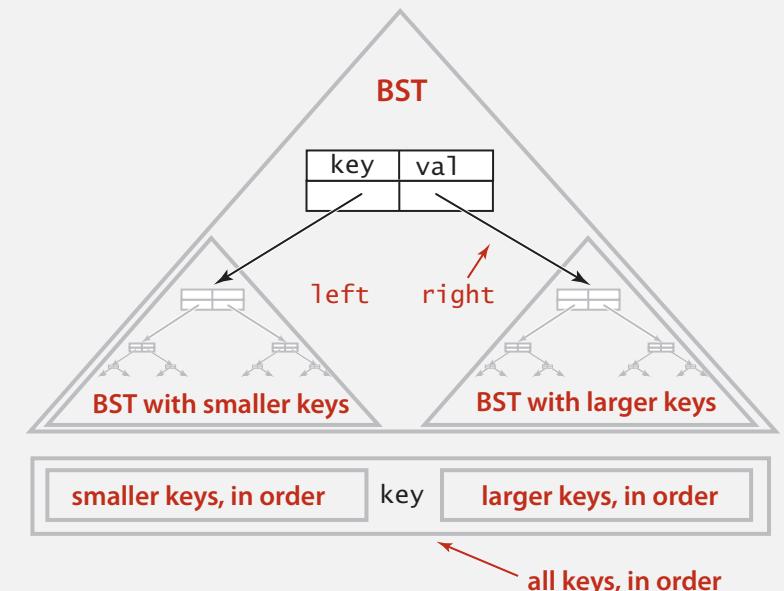
private int rank(Key key, Node x)
{
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else if (cmp == 0) return size(x.left);
}
```

# 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.

# BST: ordered symbol table operations summary

---

	sequential search	binary search	BST
search	N	$\lg N$	$h$
insert	N	N	$h$
min / max	N	1	$h$
floor / ceiling	N	$\lg N$	$h$
rank	N	$\lg N$	$h$
select	N	1	$h$
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**

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

# ST implementations: summary

---

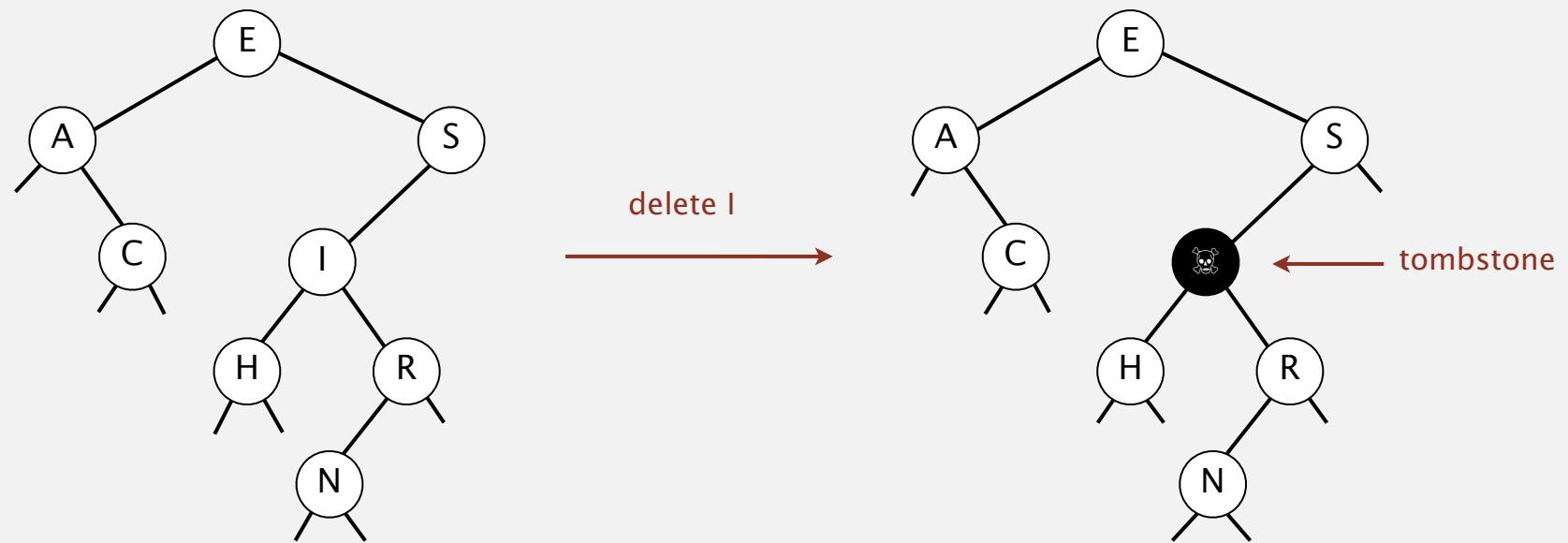
implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	???	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 search (but don't consider it equal in search).



**Cost.**  $\sim 2 \ln N'$  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.

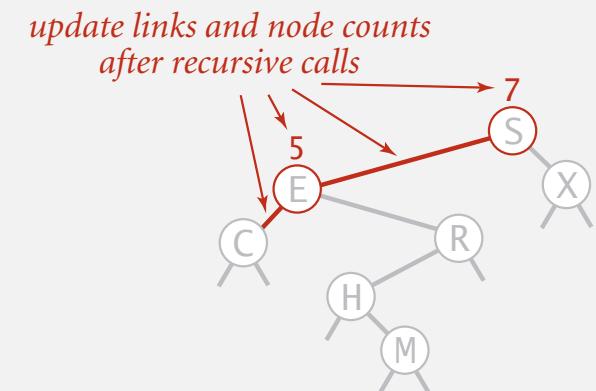
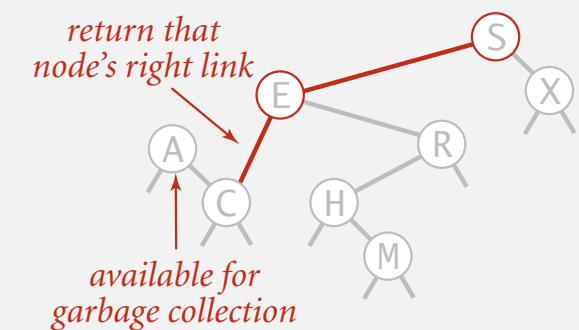
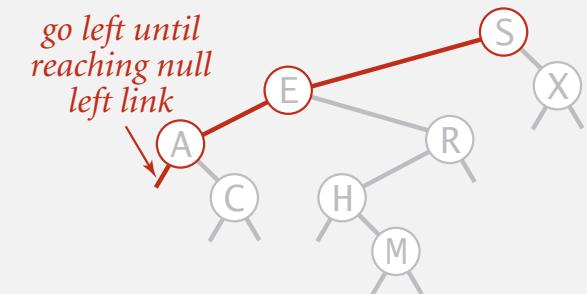
# Deleting the minimum

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);  }

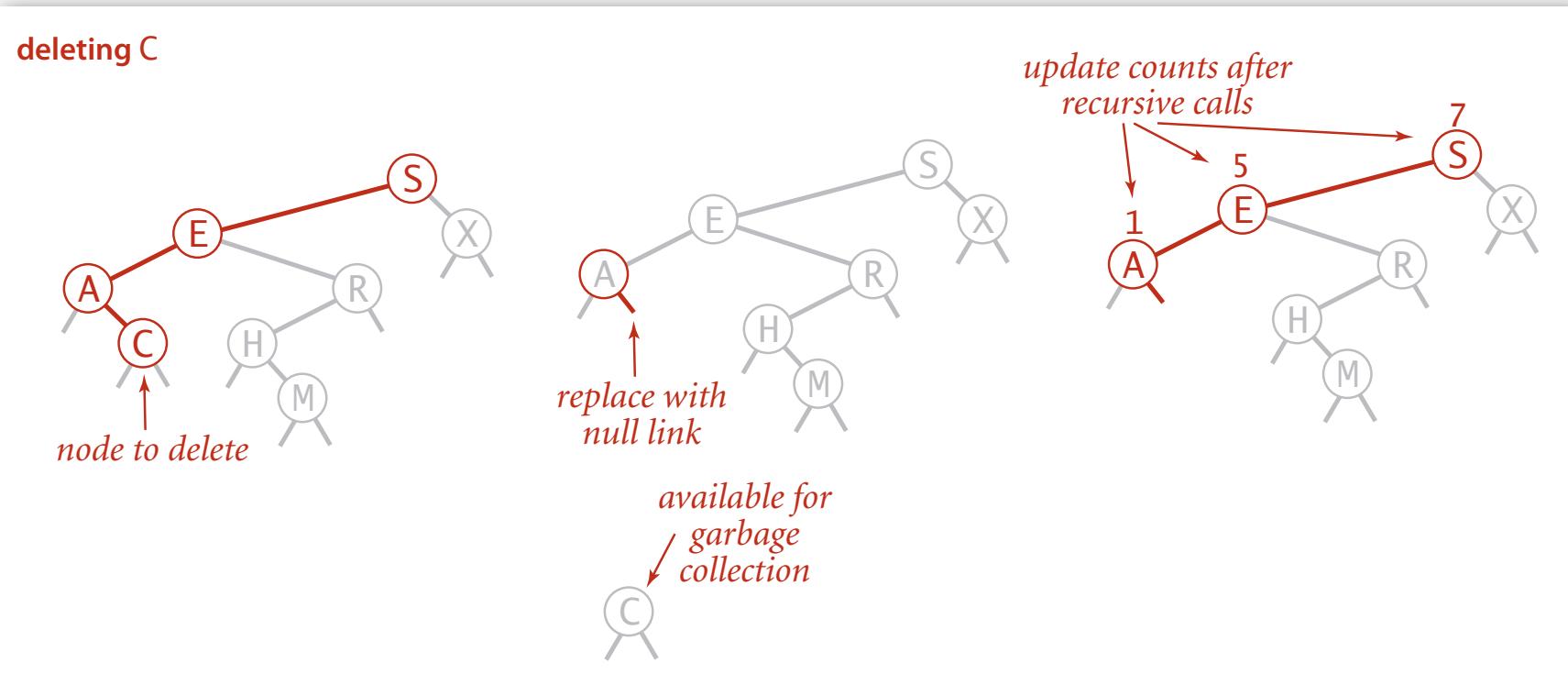
private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}
```



# Hibbard deletion

To delete a node with key k: search for node t containing key k.

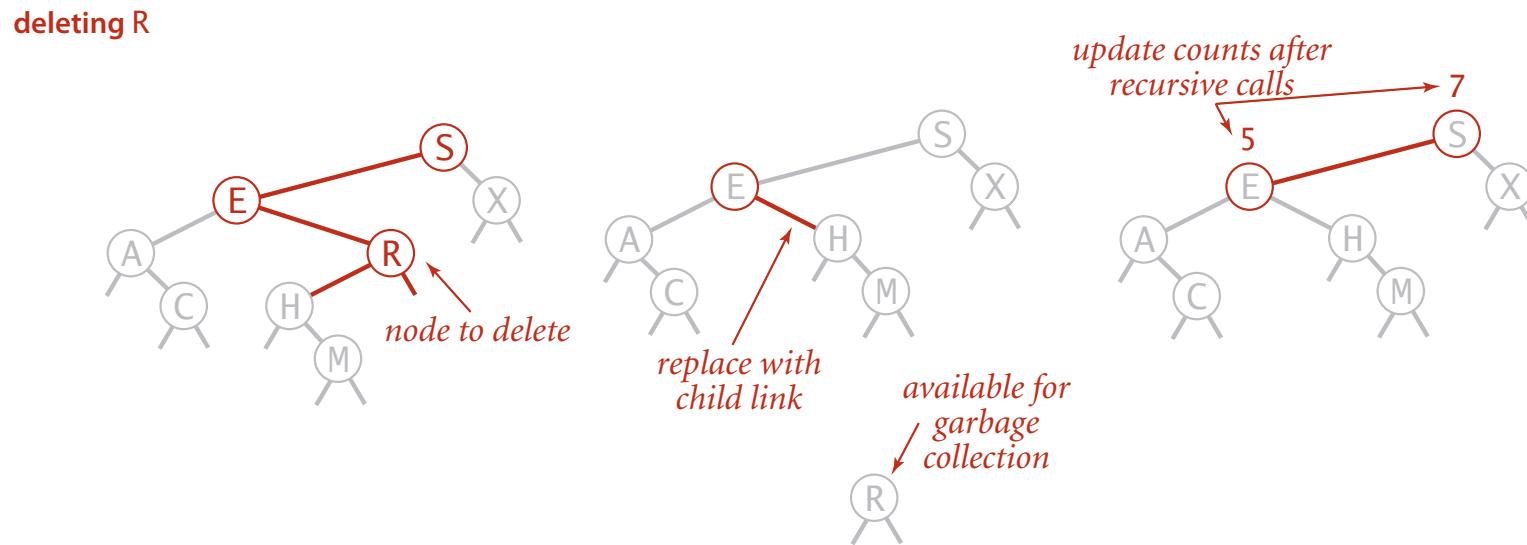
Case 0. [0 children] Delete t by setting parent link to null.



# Hibbard deletion

To delete a node with key k: search for node t containing key k.

Case 1. [1 child] Delete t by replacing parent link.

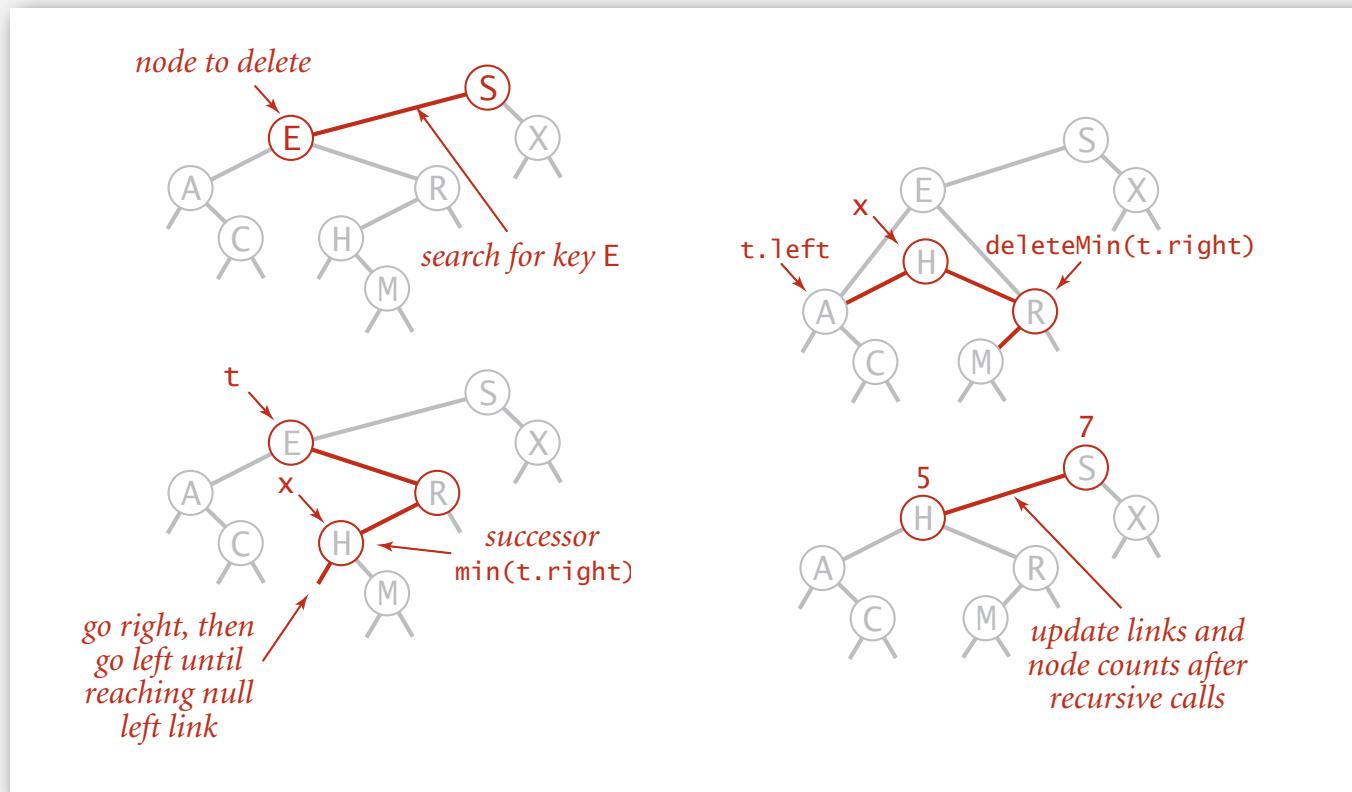


# Hibbard deletion

To delete a node with key  $k$ : search for node  $t$  containing key  $k$ .

## Case 2. [2 children]

- Find successor  $x$  of  $t$ . ←  $x$  has no left child
- Delete the minimum in  $t$ 's right subtree. ← but don't garbage collect  $x$
- Put  $x$  in  $t$ 's spot. ← still a BST



# Hibbard deletion: Java implementation

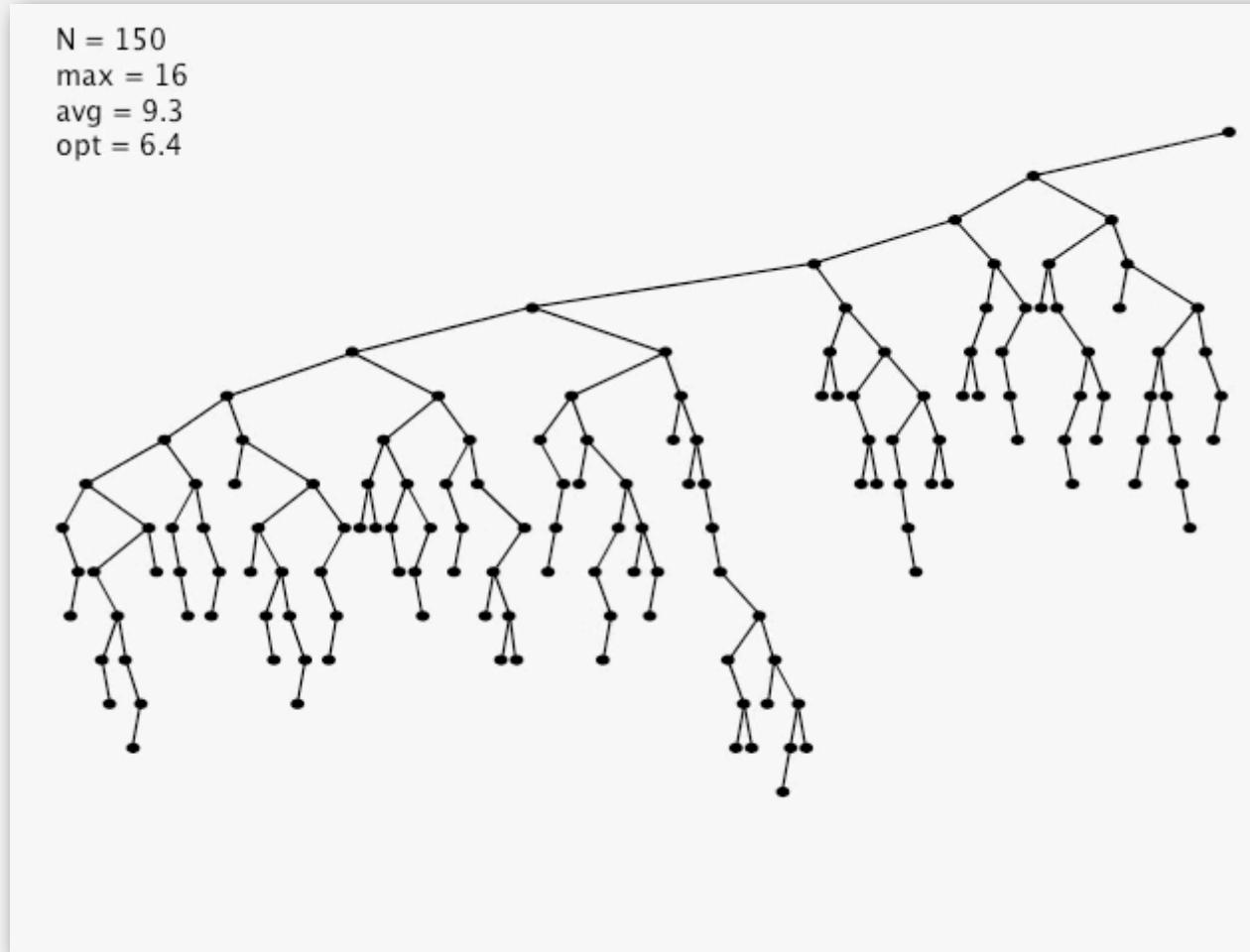
```
public void delete(Key key)
{   root = delete(root, key);  }

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
        if (x.left  == null) return x.right; ← no left child
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right); ← replace with successor
        x.left = t.left;
    }
    x.count = size(x.left) + size(x.right) + 1; ← update subtree counts
    return x;
}
```

## Hibbard deletion: analysis

---

Unsatisfactory solution. Not symmetric.



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	equals()
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$N/2$	$N/2$	yes	compareTo()
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	yes	compareTo()

other operations also become  $\sqrt{N}$   
if deletions allowed

Next lecture. **Guarantee** logarithmic performance for all operations.

# Algorithms

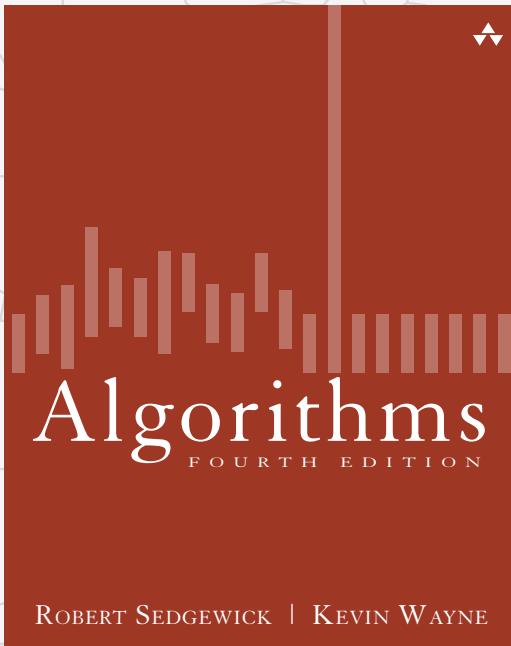
ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*



<http://algs4.cs.princeton.edu>

## 3.2 BINARY SEARCH TREES

---

- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*