Computer Science 226

Algorithms and Data Structures

Spring 2009

Instructor:
Prof. Sedgewick

Course Overview

- outline
- why study algorithms?
- usual suspects
- coursework
- resources (web)
- resources (books)

What is COS 226?

- Intermediate-level survey course.
- Programming and problem solving with applications.
- Algorithm: method for solving a problem.
- Data structure: method to store information.

topic	data structures and algorithms
data types	stack, queue, union-find, priority queue
sorting	quicksort, mergesort, heapsort, radix sorts
searching	hash table, BST, red-black tree
graphs	BFS, DFS, Prim, Kruskal, Dijkstra
strings	KMP, Regular expressions, TST, Huffman, LZW
geometry	Graham scan, k-d tree, Voronoi diagram

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. CD player, DVD, MP3, JPG, DivX, HDTV, ...

Transportation. Airline crew scheduling, map routing, ...

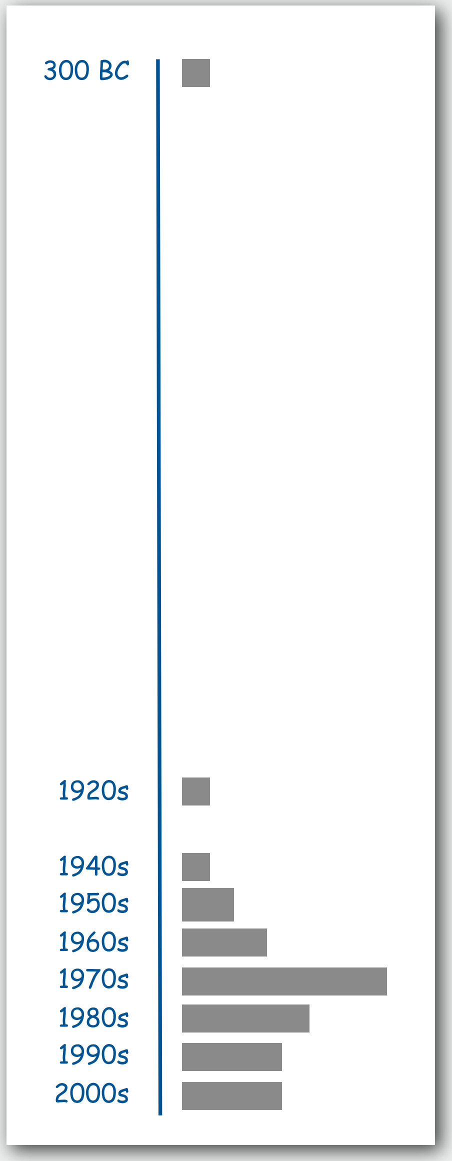
Physics. N-body simulation, particle collision simulation, ...

...

Why study algorithms?

Old roots, new opportunities.

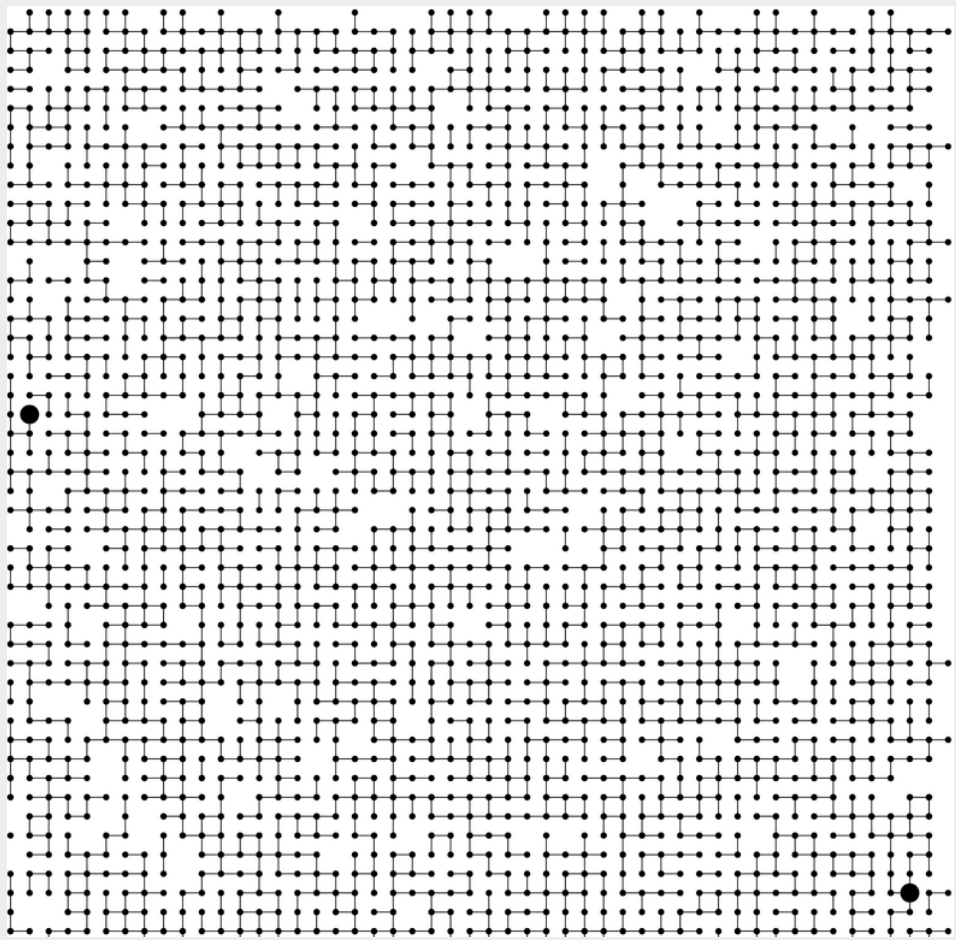
- Study of algorithms dates at least to Euclid
- Some important algorithms were discovered by undergraduates!



Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity. [stay tuned]



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

> “ *An algorithm must be seen to be believed.* ” — D. E. Knuth

Why study algorithms?

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

Computational models are replacing mathematical models in scientific enquiry

$$E = mc^2 \qquad F = \frac{G m_1 m_2}{r^2}$$

$$F = ma$$

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

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.

Nintendo

Adobe

Google™

LUCASFILM Ltd

D E Shaw & Co

Morgan Stanley

Microsoft

YAHOO!

P I X A R

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.
- They may unlock the secrets of life and of the universe.
- For fun and profit.

Why study anything else?

The usual suspects

Lectures. Introduce new material, answer questions.

Precepts. Answer questions, solve problems, discuss programming assignment.

first precept meets this week!

What	When	Where	Who	Office Hours
L01	MW 11-12:20	Bowen 222	Prof. Sedgewick	W 1-2 (Cafe Viv)
P01	Th 12:30	CS 102	Moritz Hardt	see web page
P01A	Th 12:30	Friend 112	Maia Ginsburg (lead preceptor)	see web page
P02	Th 1:30	CS 302	Martin Suchara	see web page
P03	Th 3:30	Friend 109	Aravindan Vijayaraghavan	see web page

FAQ.

- Not registered? Change precept? Use SCORE.
- See Donna O'Leary (CS 210) to resolve serious conflicts.
- See Maia Ginsburg (CS 205) for other course admin issues.

Coursework and grading

8 programming assignments. 45%

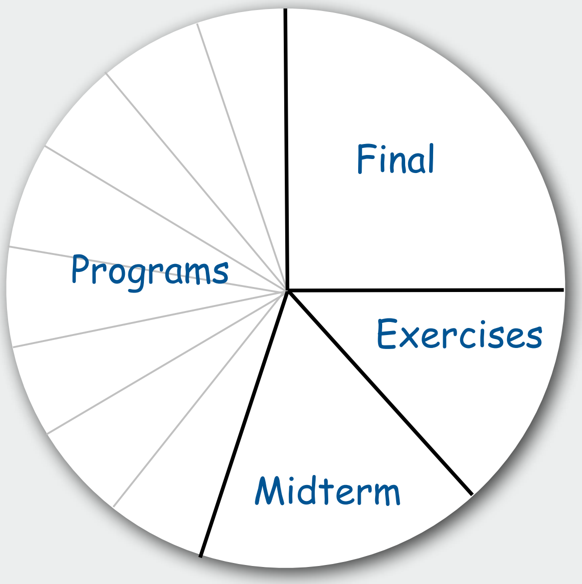
- Electronic submission.
- Due 11:55pm, starting Wednesday 9/17.

Exercises. 15%

- Due in lecture, starting Tuesday 9/16.

Exams.

- Closed-book with cheatsheet.
- Midterm. 15%
- Final. 25%



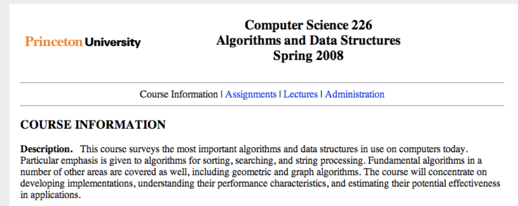
Staff discretion. To adjust borderline cases.

everyone needs to meet me (at least) once!

Resources (web)

Course content.

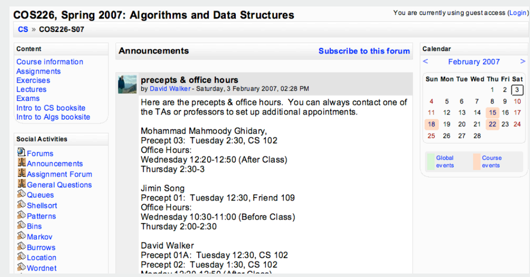
- Course info.
- Exercises.
- Lecture slides.
- Programming assignments.



`http://www.princeton.edu/~cos226`

Course administration.

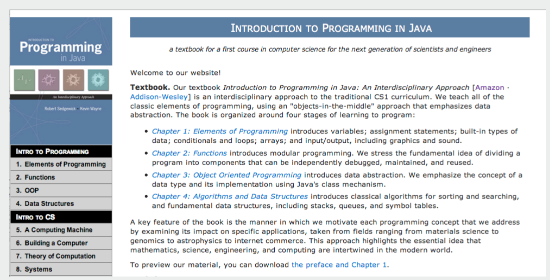
- Check grades.
- Submit assignments.



`https://moodle.cs.princeton.edu/course/view.php?id=40`

Booksites.

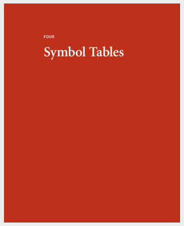
- Brief summary of content.
- Download code from lecture.



`http://www.cs.princeton.edu/IntroProgramming`
`http://www.cs.princeton.edu/algs4`

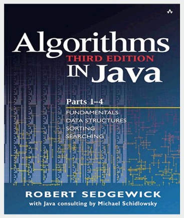
Resources (books)

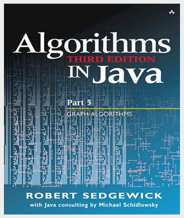
Algorithms 4th edition availability TBA



Algorithms in Java, 3rd edition

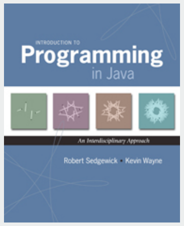
- Parts 1-4. [sorting, searching] recommended
- Part 5. [graph algorithms] required





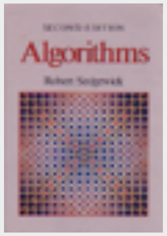
Introduction to Programming recommended

- Basic programming model.
- Elementary AofA and data structures.



Algorithms, 2nd edition availability TBA

- Strings.
- Geometric algorithms.



Union-Find Algorithms

- dynamic connectivity
- quick find
- quick union
- improvements
- applications

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

- **dynamic connectivity**
- quick find
- quick union
- improvements
- applications

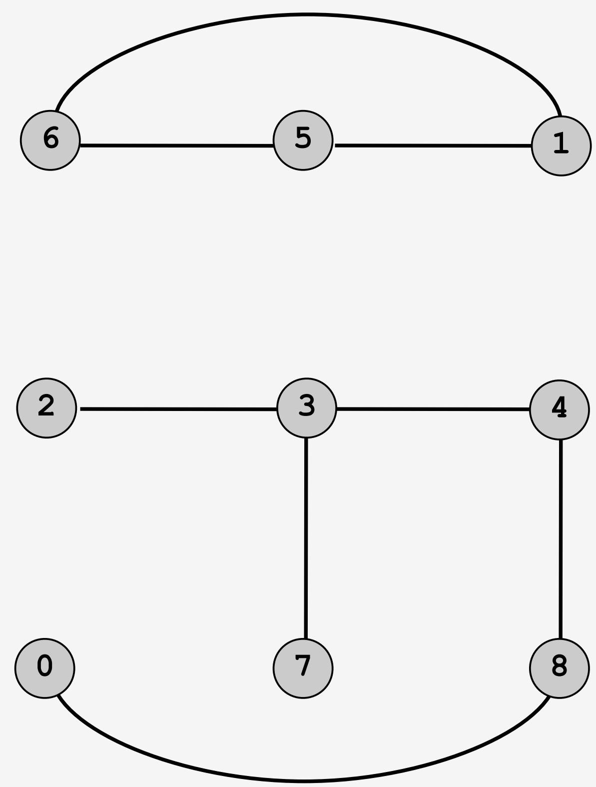
Dynamic connectivity

Given a set of objects

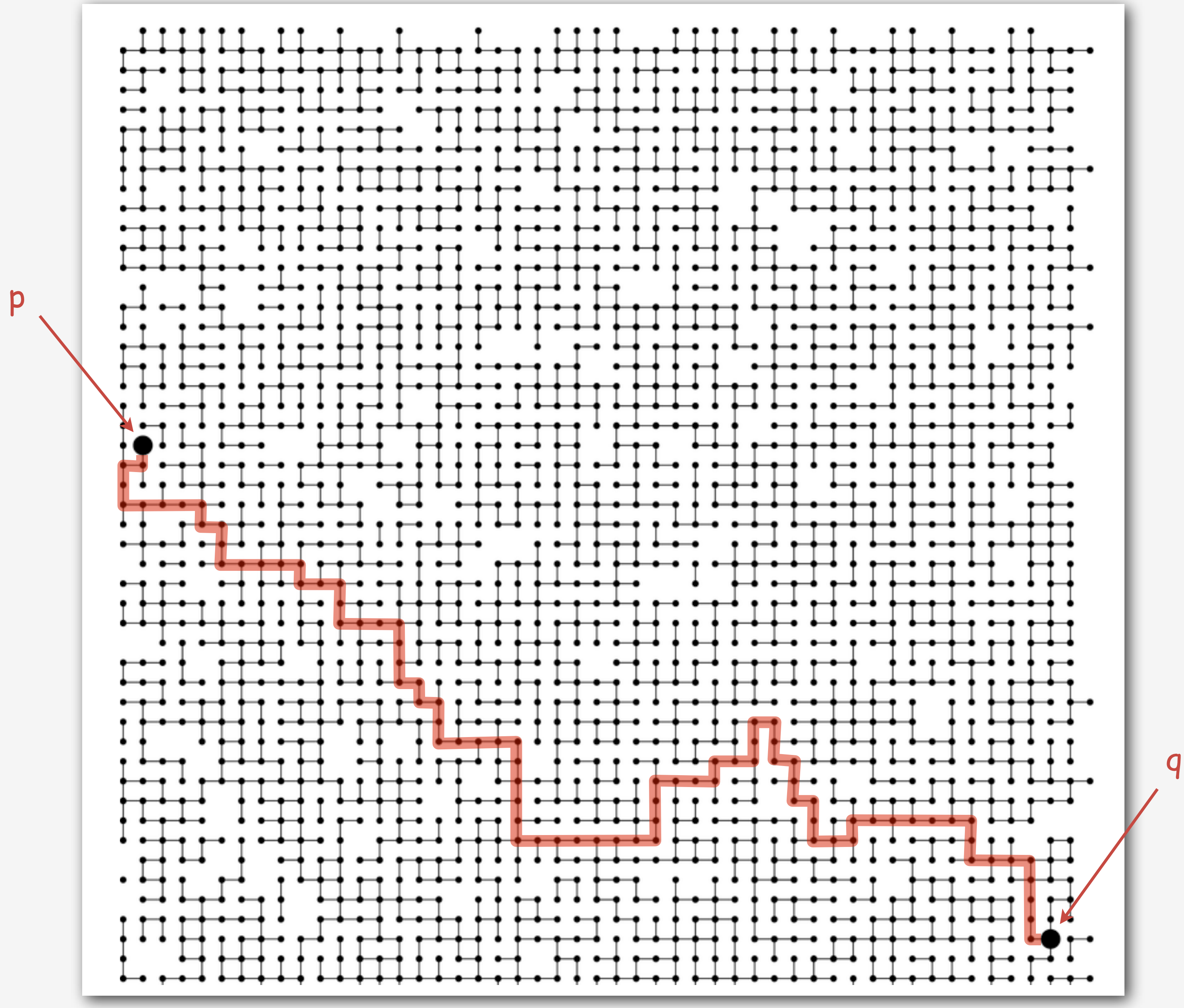
- Union: connect two objects.
- Find: is there a path connecting the two objects?

more difficult problem: find the path

```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
 find(0, 2)      no
 find(2, 4)      yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
 find(0, 2)      yes
 find(2, 4)      yes
```



Network connectivity: larger example



Modeling the objects

Dynamic connectivity applications involve manipulating objects of all types.

- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N-1.

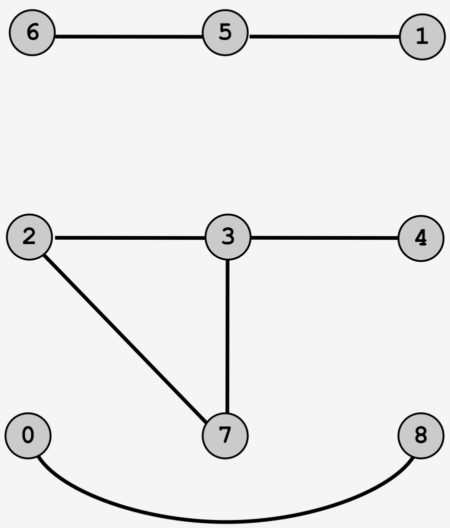
- Use integers as array index.
- Suppress details not relevant to union-find.

can use symbol table to translate from
object names to integers (stay tuned)

Modeling the connections

Transitivity.
If `p` is connected to `q` and `q` is connected to `r`, then `p` is connected to `r`.

Connected components. Maximal set of objects that are mutually connected.



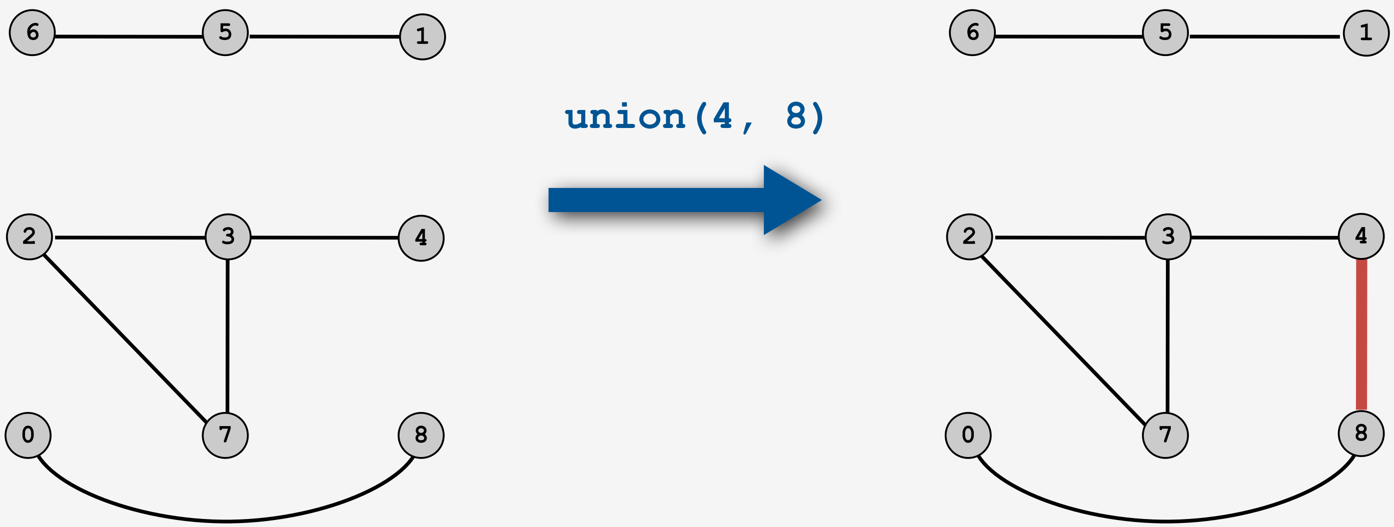
`{ 1 5 6 } { 2 3 4 7 } { 0 8 }`

connected components

Implementing the operations

Find query. Check if two objects are in the same set.

Union command. Replace sets containing two objects with their union.



`{ 1 5 6 } { 2 3 4 7 } { 0 8 }`

`{ 1 5 6 } { 0 2 3 4 7 8 }`

connected components

Union-find data type (API)

Goal. Design efficient data structure for union-find.

- Number of objects N can be huge.
- Number of operations M can be huge.
- Find queries and union commands may be intermixed.

public class UnionFind	
`UnionFind(int N)`	*create union-find data structure with N objects and no connections*
`boolean find(int p, int q)`	*are p and q in the same set?*
`void unite(int p, int q)`	*replace sets containing p and q with their union*

- dynamic connectivity
- **quick find**
- quick union
- improvements
- applications

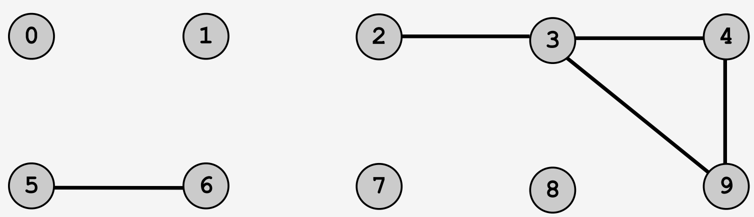
Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected



Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

id[3] = 9; id[6] = 6
3 and 6 not connected

Quick-find [eager approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `p` and `q` are connected if they have the same id.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	9	9	6	6	7	8	9

5 and 6 are connected
2, 3, 4, and 9 are connected

Find. Check if `p` and `q` have the same id.

id[3] = 9; id[6] = 6
3 and 6 not connected

Union. To merge sets containing `p` and `q`, change all entries with `id[p]` to `id[q]`.

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	6	6	6	6	6	7	8	6

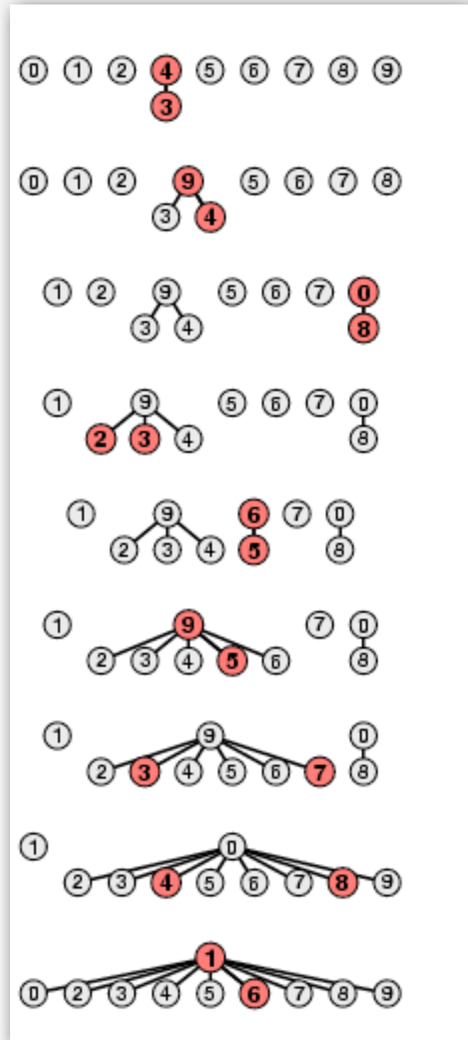
union of 3 and 6
2, 3, 4, 5, 6, and 9 are connected

problem: many values can change

Quick-find example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	9	9	5	6	7	8	9
8-0	0	1	2	9	9	5	6	7	0	9
2-3	0	1	9	9	9	5	6	7	0	9
5-6	0	1	9	9	9	6	6	7	0	9
5-9	0	1	9	9	9	9	9	7	0	9
7-3	0	1	9	9	9	9	9	9	0	9
4-8	0	1	0	0	0	0	0	0	0	0
6-1	1	1	1	1	1	1	1	1	1	1

problem: many values can change



Quick-find: Java implementation

```
public class QuickFind
{
   private int[] id;

   public QuickFind(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean find(int p, int q)
   {
      return id[p] == id[q];
   }

   public void unite(int p, int q)
   {
      int pid = id[p];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = id[q];
   }
}
```

set id of each object to itself
(N operations)

check if p and q have same id
(1 operation)

change all entries with id[p] to id[q]
(N operations)

Quick-find is too slow

Quick-find defect.

- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

algorithm	union	find
quick-find	N	1

Ex. May take N^2 operations to process N union commands on N objects.

Quadratic algorithms do not scale

Rough standard (for now).

- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

a truism (roughly) since 1950 !

Ex. Huge problem for quick-find.

- 10^9 union commands on 10^9 objects.
- Quick-find takes more than 10^{18} operations.
- 30+ years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.

- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!

- dynamic connectivity
- quick find
- **quick union**
- improvements
- applications

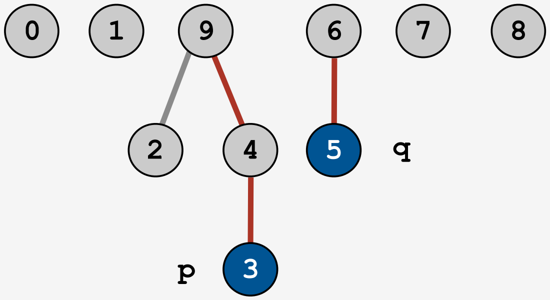
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



3's root is 9; 5's root is 6

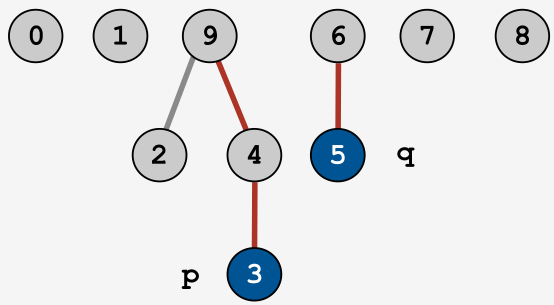
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if `p` and `q` have the same root.

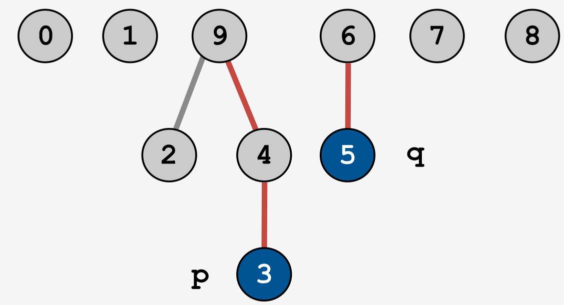
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of size `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change

i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	6	7	8	9



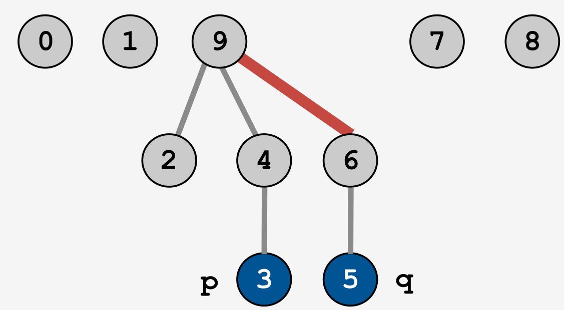
3's root is 9; 5's root is 6
3 and 5 are not connected

Find. Check if `p` and `q` have the same root.

Union. To merge subsets containing `p` and `q`, set the id of `q`'s root to the id of `p`'s root.

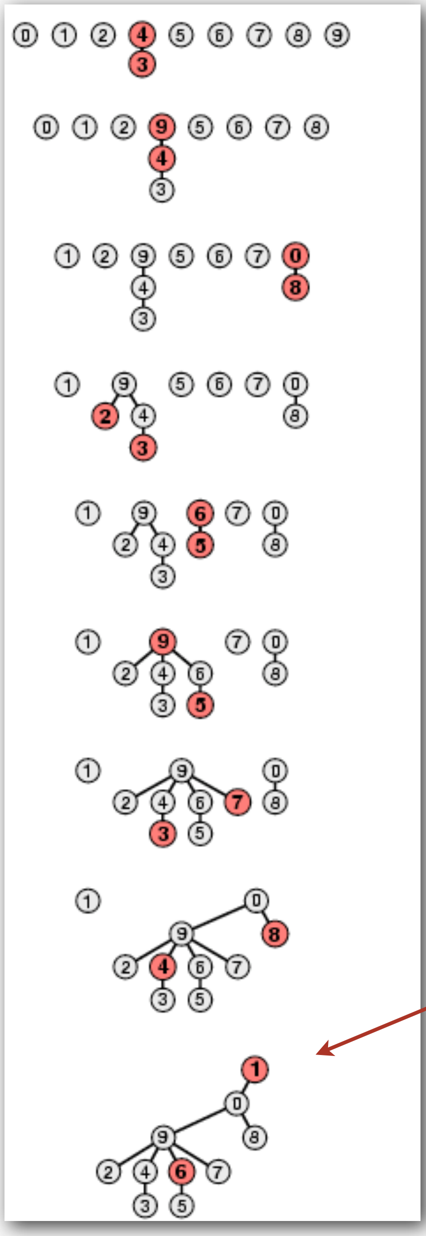
i	0	1	2	3	4	5	6	7	8	9
id[i]	0	1	9	4	9	6	9	7	8	9

only one value changes



Quick-union example

3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



problem:
trees can get tall

Quick-union: Java implementation

```
public class QuickUnion
{
   private int[] id;

   public QuickUnion(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   private int root(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean find(int p, int q)
   {
      return root(p) == root(q);
   }

   public void unite(int p, int q)
   {
      int i = root(p), j = root(q);
      id[i] = j;
   }
}
```

set id of each object to itself
(N operations)

chase parent parents until reach root
(depth of i operations)

check if p and q have same root
(depth of p and q operations)

change root of p to point to root of q
(depth of p and q operations)

Quick-union is also too slow

Quick-find defect.

- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N operations).

algorithm	union	find
quick-find	N	1
quick-union	N *	N

← worst case

* includes cost of finding root

- dynamic connectivity
- quick find
- quick union
- **improvements**
- applications

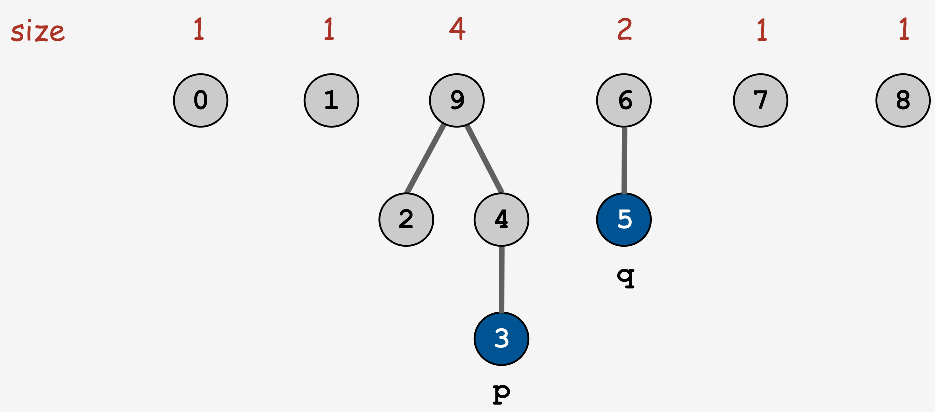
Improvement 1: weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each subset.
- Balance by linking small tree below large one.

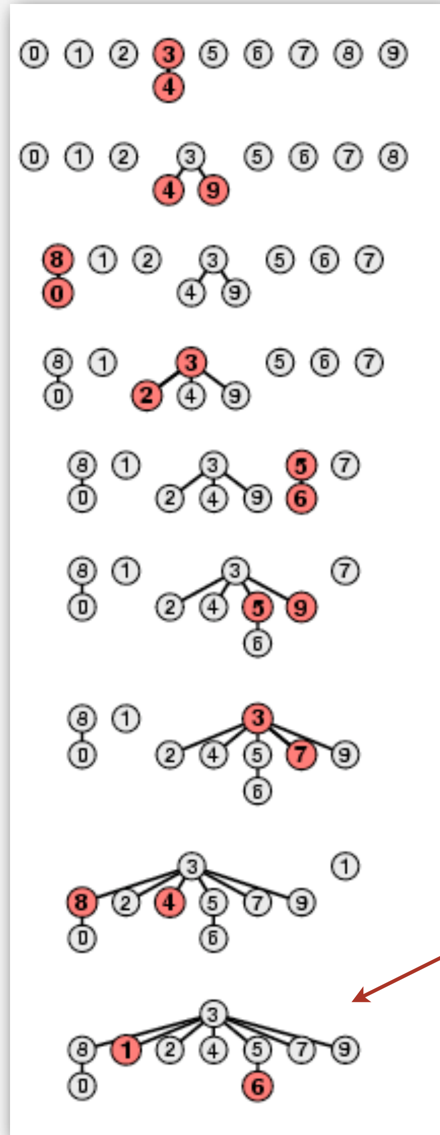
Ex. Union of `3` and `5`.

- Quick union: link `9` to `6`.
- Weighted quick union: link `6` to `9`.



Weighted quick-union example

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	5	3	3	3



no problem:
trees stay flat

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `sz[i]` to count number of objects in the tree rooted at `i`.

Find. Identical to quick-union.

```
return root(p) == root(q);
```

Union. Modify quick-union to:

- Merge smaller tree into larger tree.
- Update the `sz[]` array.

```
int i = root(p);
int j = root(q);
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

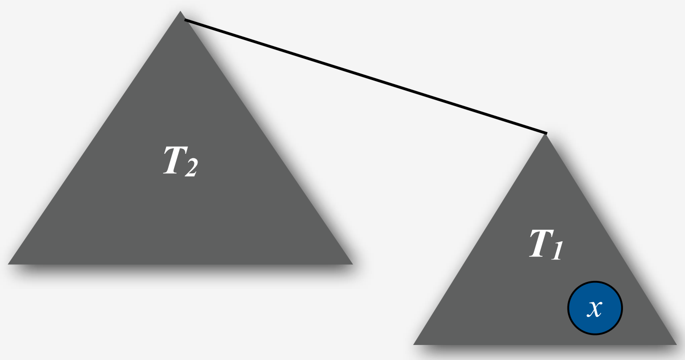
Analysis.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.
- Fact: depth is at most lg N. [needs proof]

Q. How does depth of x increase by 1?

A. Tree T_1 containing x is merged into another tree T_2.

- The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing x can double at most lg N times.



Weighted quick-union analysis

Analysis.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.
- Fact: depth is at most lg N. [needs proof]

algorithm	union	find
quick-find	N	1
quick-union	N *	N
weighted QU	lg N *	lg N

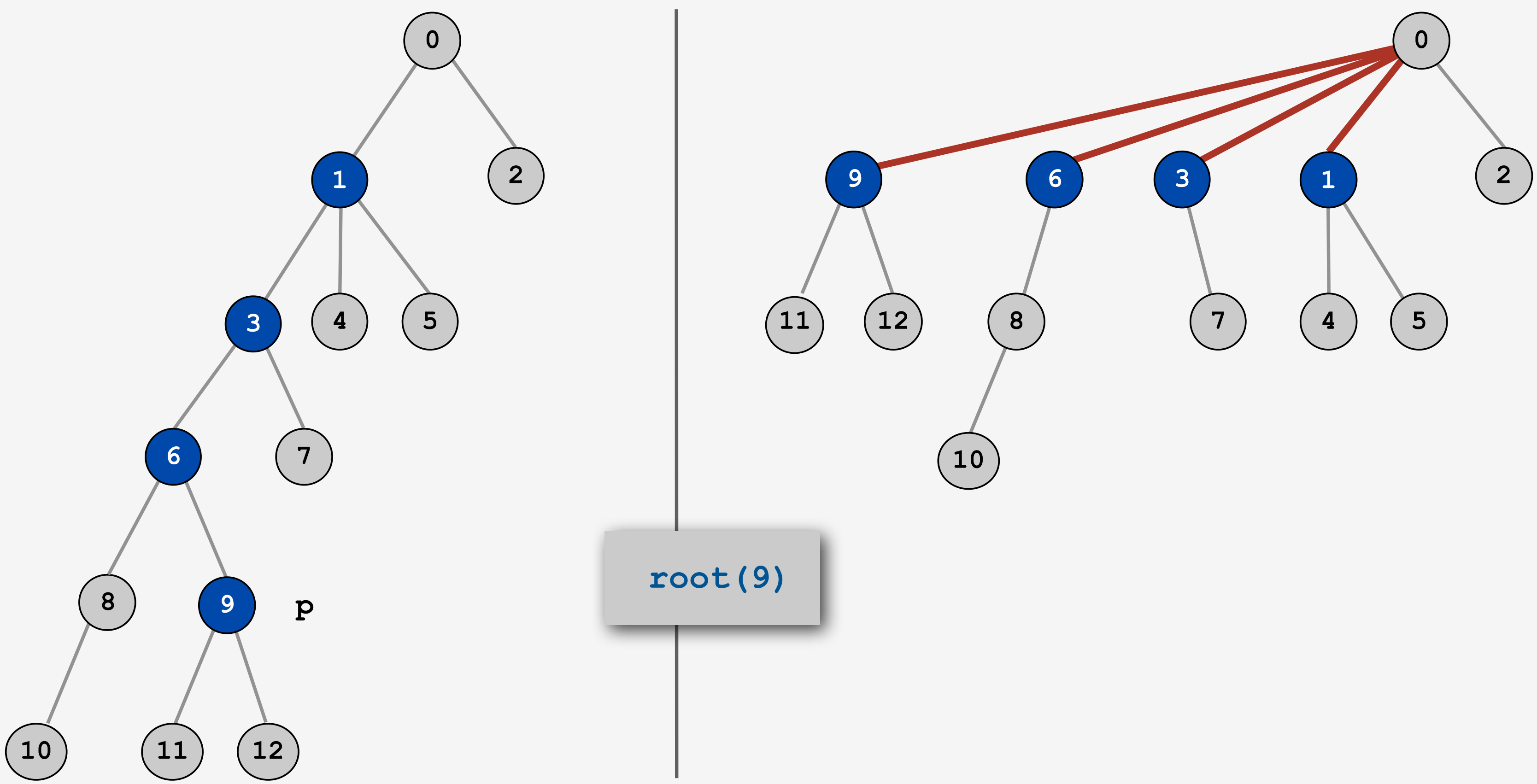
* includes cost of finding root

Q. Stop at guaranteed acceptable performance?

A. No, easy to improve further.

Improvement 2: path compression

Quick union with path compression. Just after computing the root of `p`, set the `id` of each examined node to `root(p)`.



Path compression: Java implementation

Standard implementation: add second loop to `root()` to set the id of each examined node to the root.

Simpler one-pass variant: halve the path length by making every other node in path point to its grandparent.

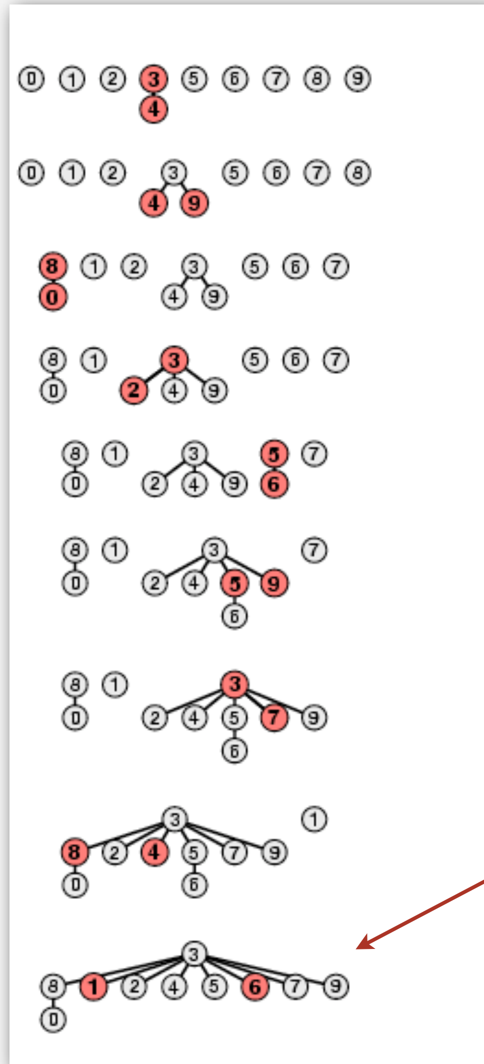
```
public int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression example

3-4	0	1	2	3	3	5	6	7	8	9
4-9	0	1	2	3	3	5	6	7	8	3
8-0	8	1	2	3	3	5	6	7	8	3
2-3	8	1	3	3	3	5	6	7	8	3
5-6	8	1	3	3	3	5	5	7	8	3
5-9	8	1	3	3	3	3	5	7	8	3
7-3	8	1	3	3	3	3	5	3	8	3
4-8	8	1	3	3	3	3	5	3	3	3
6-1	8	3	3	3	3	3	3	3	3	3



no problem:
trees stay VERY flat

WQUPC performance

Theorem. [Tarjan 1975] Starting from an empty data structure, any sequence of M union and find operations on N objects takes O(N + M lg* N) time.

- Proof is very difficult.
- But the algorithm is still simple!

actually O(N + M α(M, N))
see COS 423

Linear algorithm?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

because lg* N is a constant in this universe

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

lg* function
number of times needed to take the lg of a number until reaching 1

Amazing fact. No linear-time linking strategy exists.

Summary

Bottom line. WQUPC makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

M union-find operations on a set of N objects

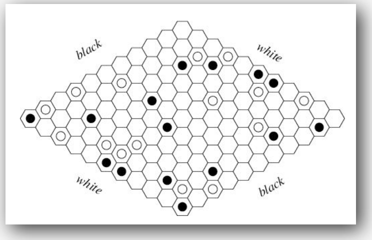
Ex. [10^9 unions and finds with 10^9 objects]

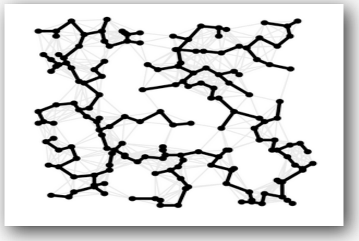
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

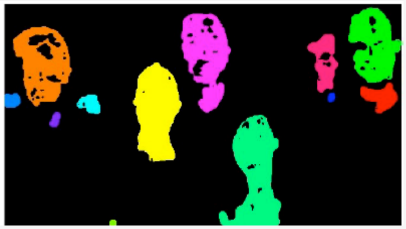
- dynamic connectivity
- quick find
- quick union
- improvements
- **applications**

Union-find applications

- Percolation.
- Games (Go, Hex).
- ✓ Network connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Morphological attribute openings and closings.
- Matlab's **`bwlabel()`** function in image processing.



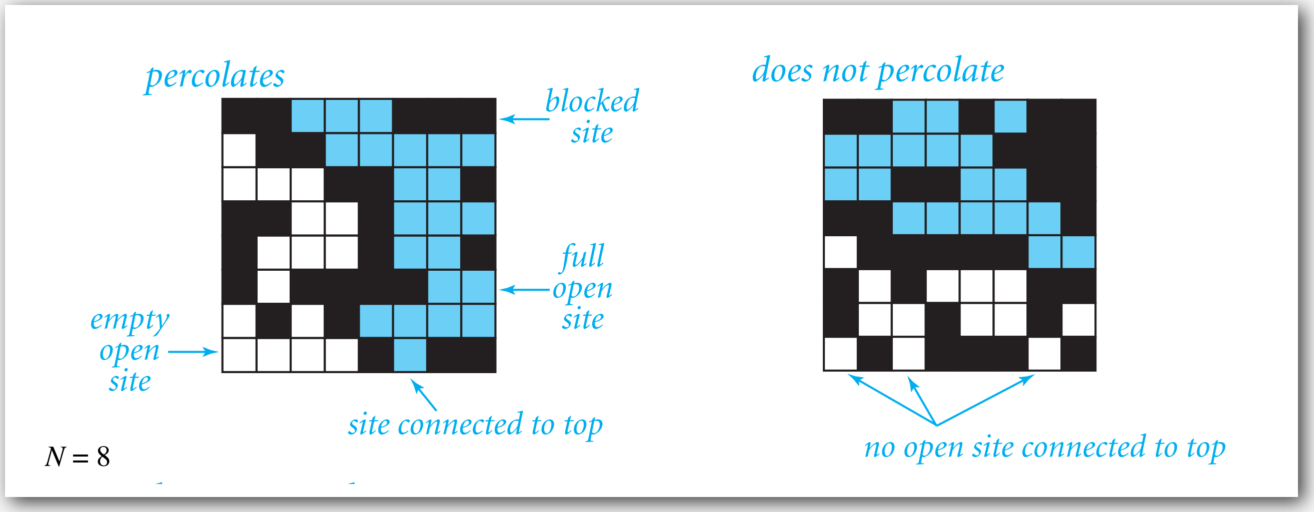




Percolation

A model for many physical systems:

- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1-p).
- System percolates if top and bottom are connected by open sites.



Percolation

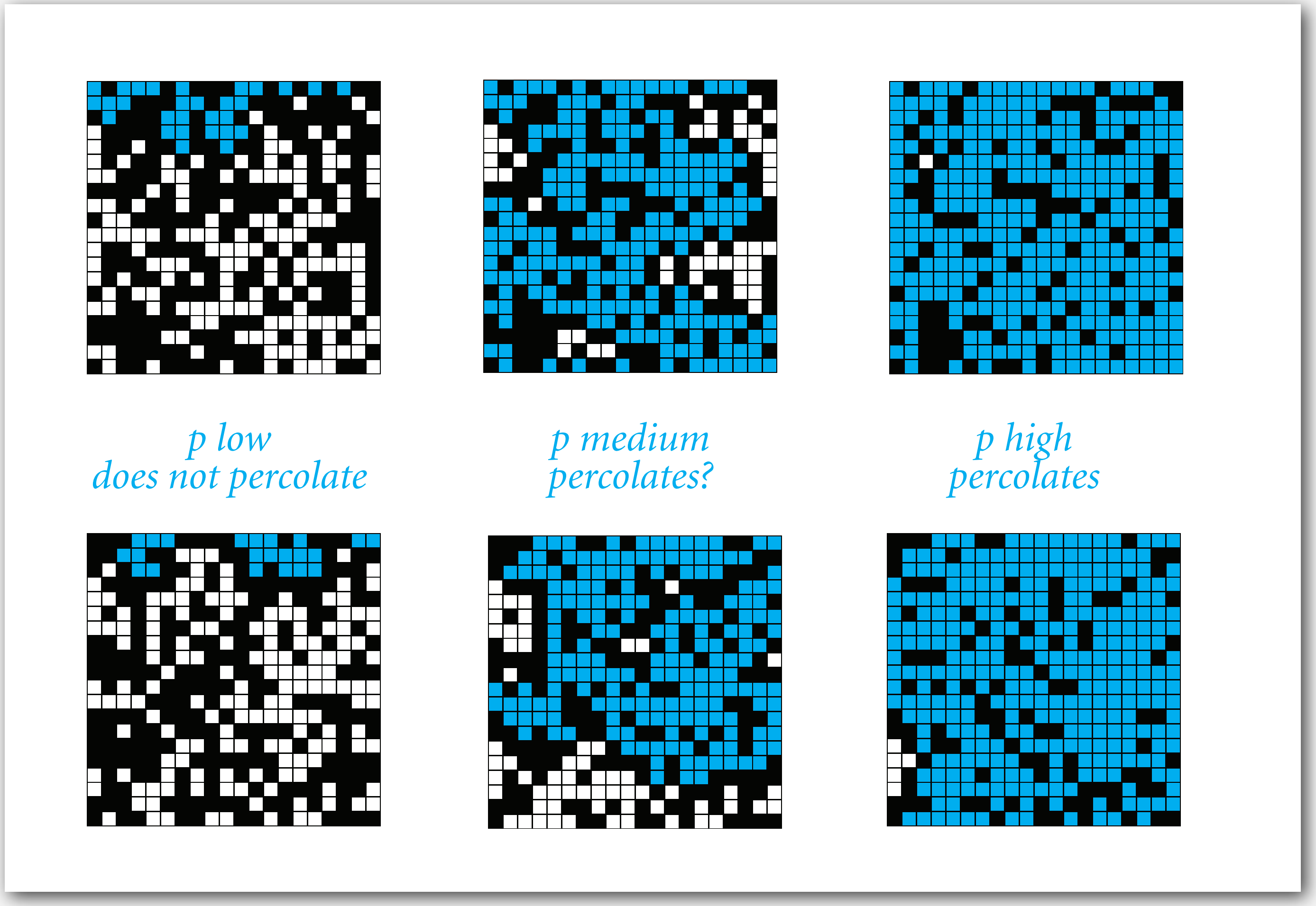
A model for many physical systems:

- N-by-N grid of sites.
- Each site is open with probability p (or blocked with probability 1-p).
- System percolates if top and bottom are connected by open sites.

model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Likelihood of percolation

Depends on site vacancy probability p.

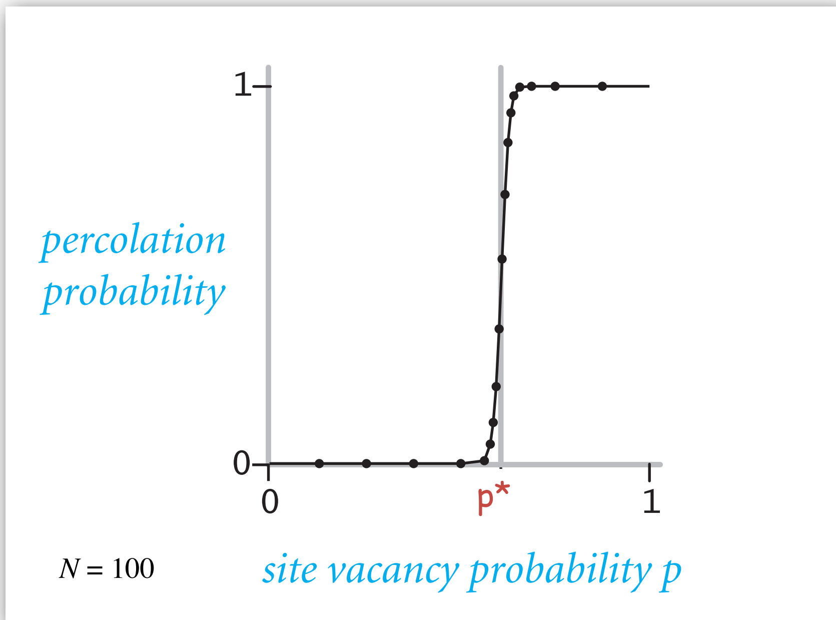


$N = 20$

Theory guarantees a sharp threshold p* (when N is large).

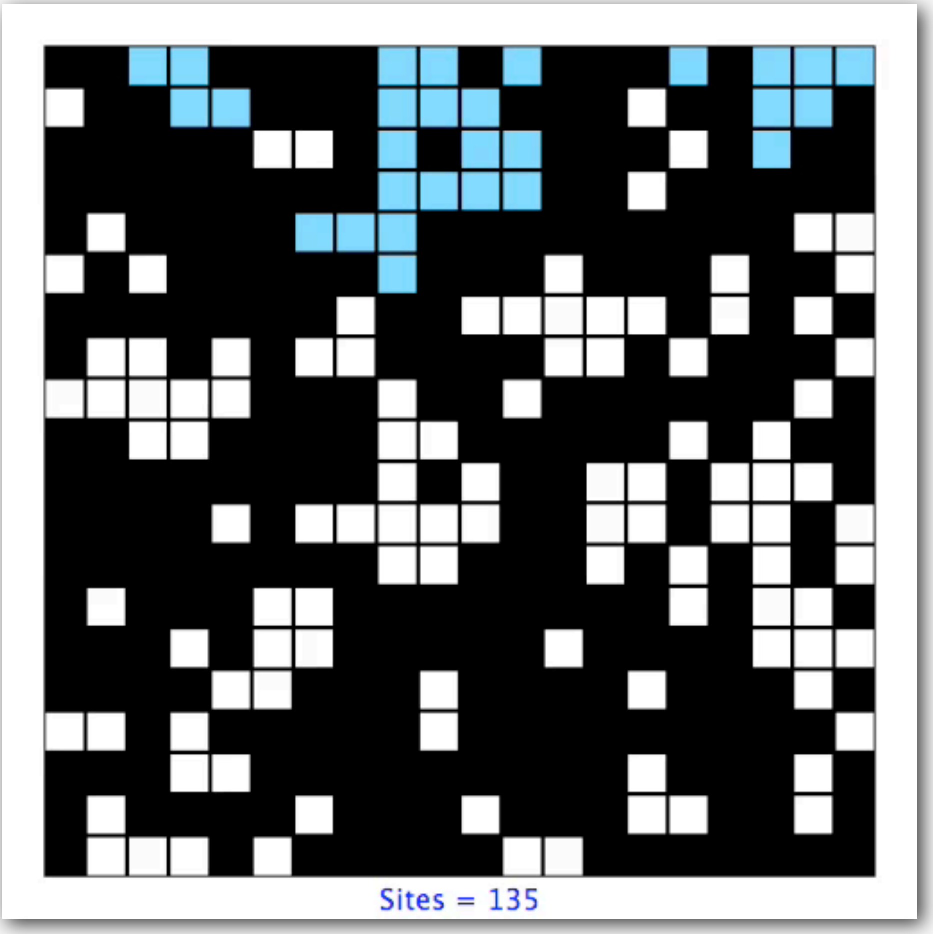
- p > p*: almost certainly percolates.
- p < p*: almost certainly does not percolate.

Q. What is the value of p* ?



Monte Carlo simulation

- Initialize N-by-N whole grid to be blocked.
- Make random sites open until top connected to bottom.
- Vacancy percentage estimates p*.

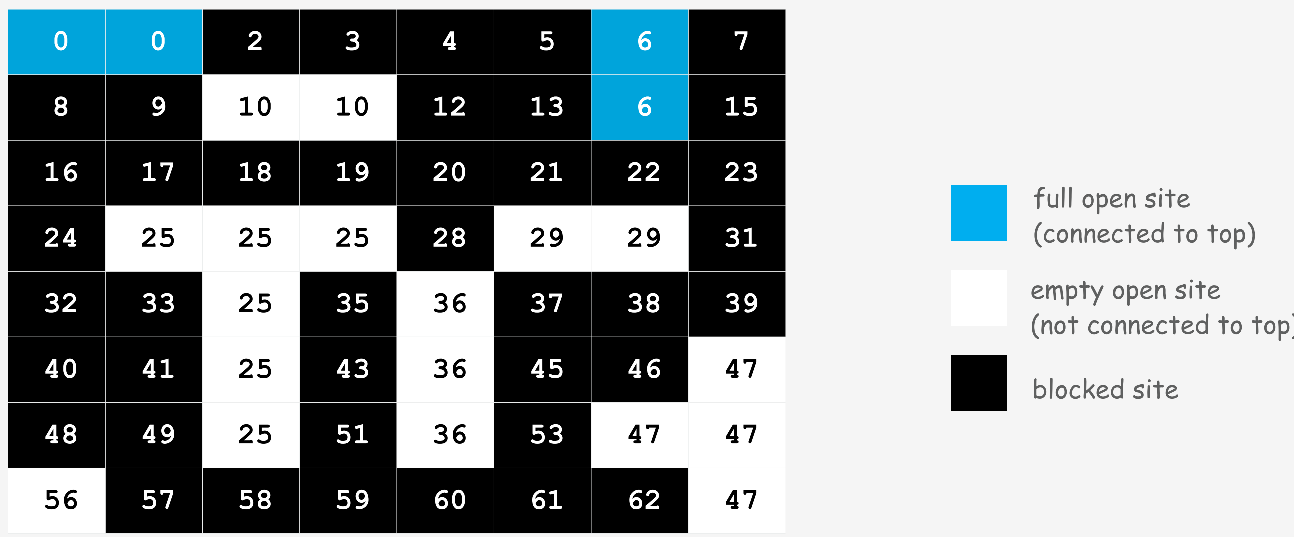


UF solution to find percolation threshold

How to check whether system percolates?

- Create object for each site.
- Sites are in same set if connected by open sites.
- Percolates if any site in top row is in same set as any site in bottom row.

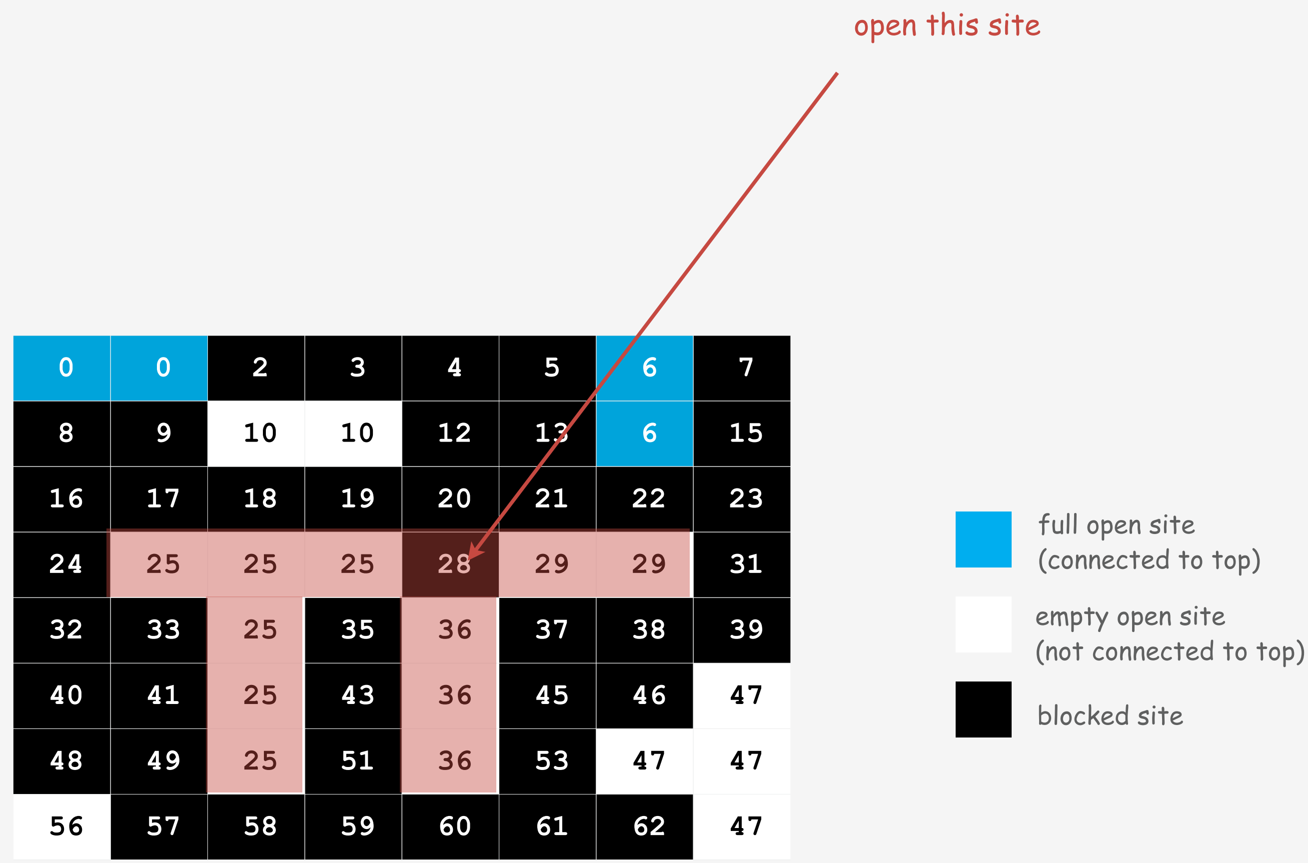
brute force alg would need to check N^2 pairs



$N = 8$

UF solution to find percolation threshold

Q. How to declare a new site open?

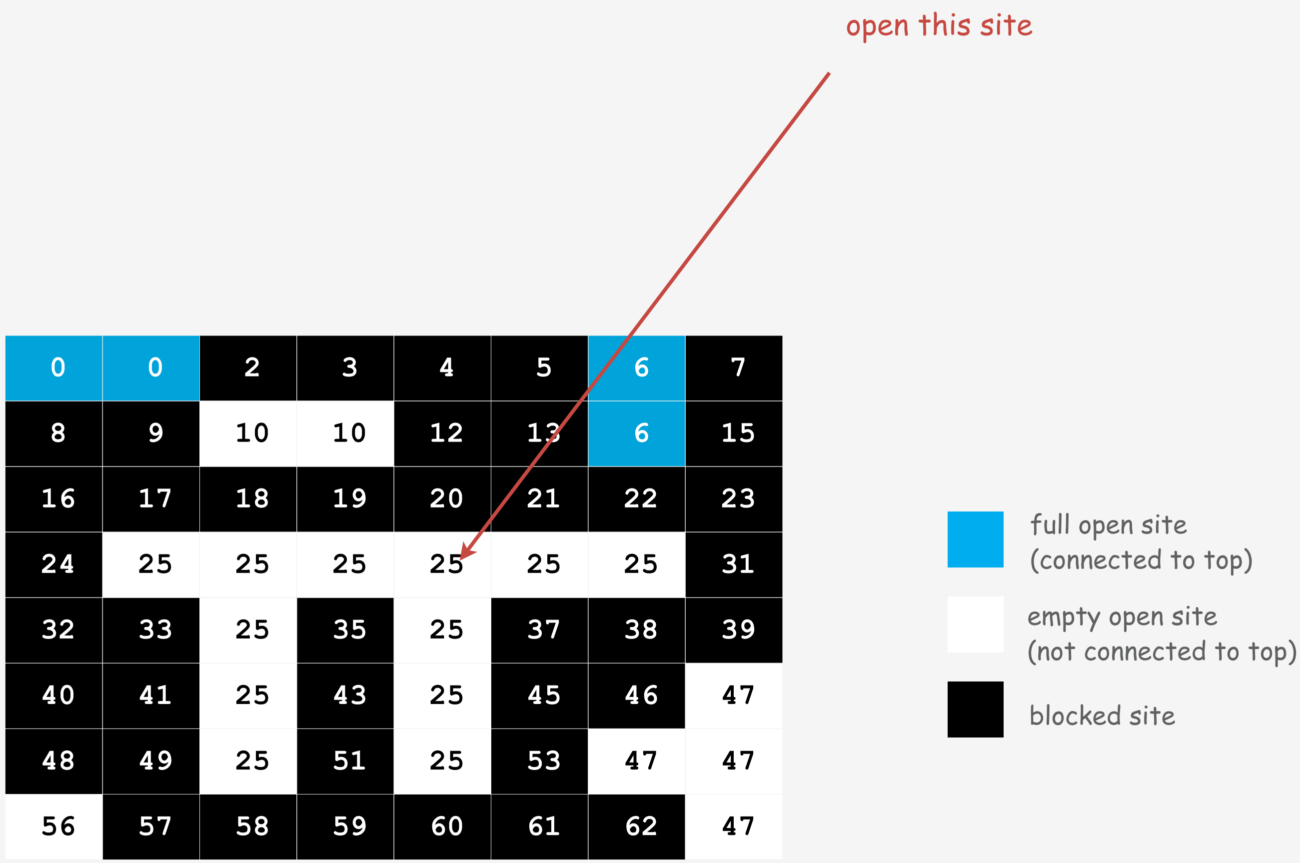


$N = 8$

UF solution to find percolation threshold

Q. How to declare a new site open?

A. Take union of new site and all adjacent open sites.



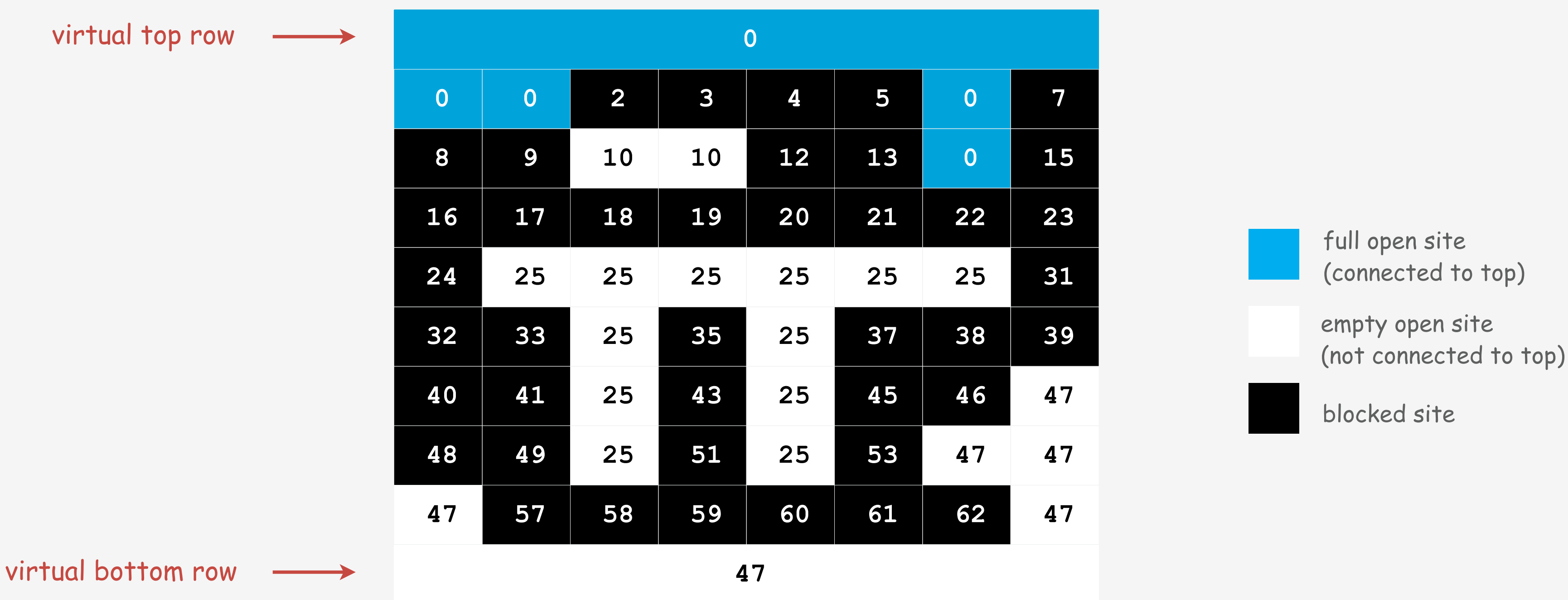
$N = 8$

UF solution: a critical optimization

Q. How to avoid checking all pairs of top and bottom sites?

A. Create a virtual top and bottom objects;

system percolates when virtual top and bottom objects are in same set.



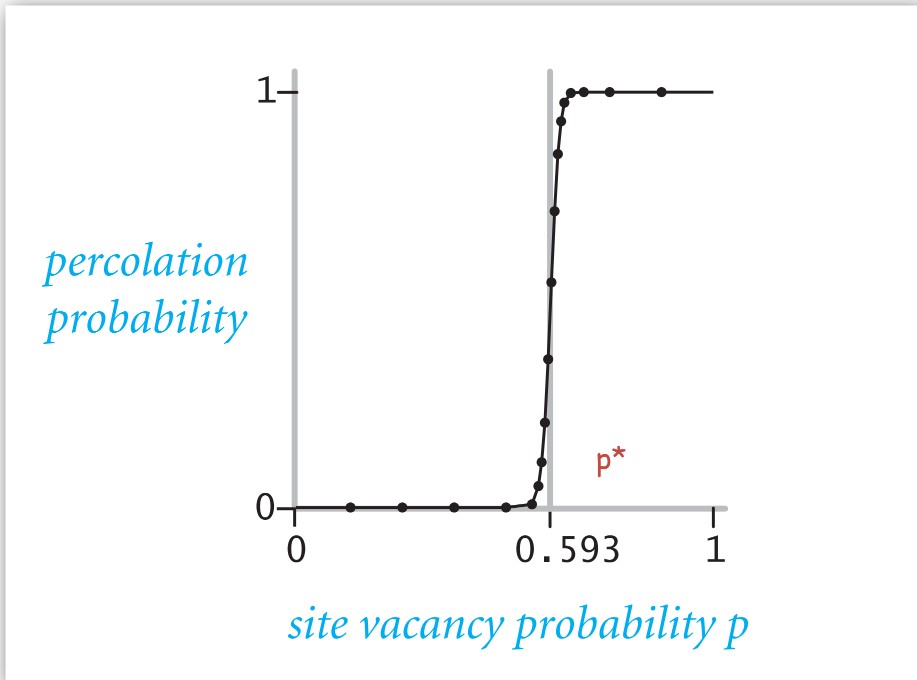
$N = 8$

Percolation threshold

Q. What is percolation threshold p* ?

A. About 0.592746 for large square lattices.

percolation constant known
only via simulation



Subtext of today's lecture (and this course)

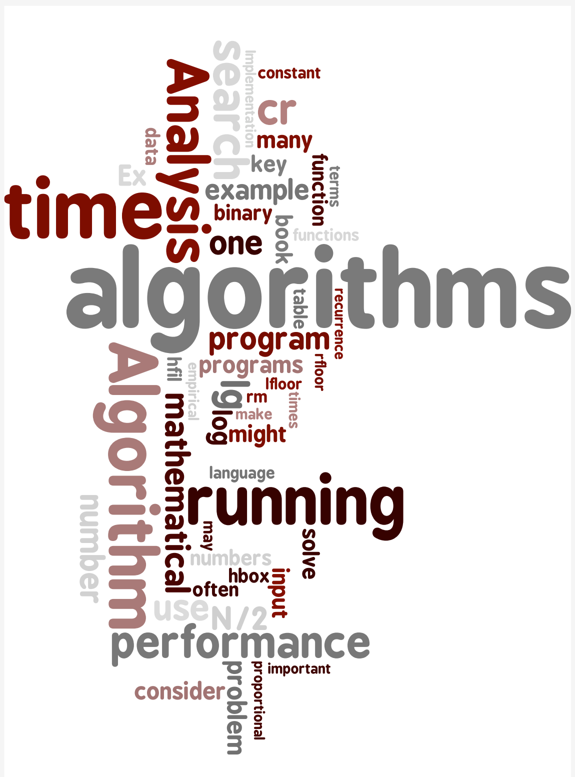
Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

Analysis of Algorithms

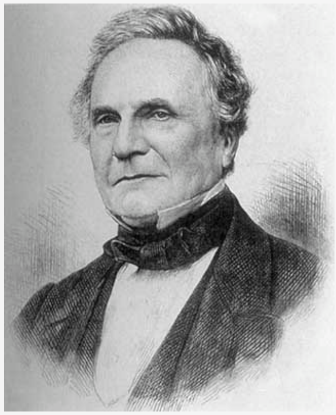


- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space

Reference:
Algorithms in Java, Chapter 2
Intro to Programming in Java, Section 4.1
`http://www.cs.princeton.edu/algs4`

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



Charles Babbage (1864)

Analytic Engine

how many times
do you have to
turn the crank?

Reasons to analyze algorithms

Predict performance.

Compare algorithms.

Provide guarantees.

Understand theoretical basis.

this course (COS 226)

theory of algorithms (COS 423)

Primary practical reason: avoid performance bugs.

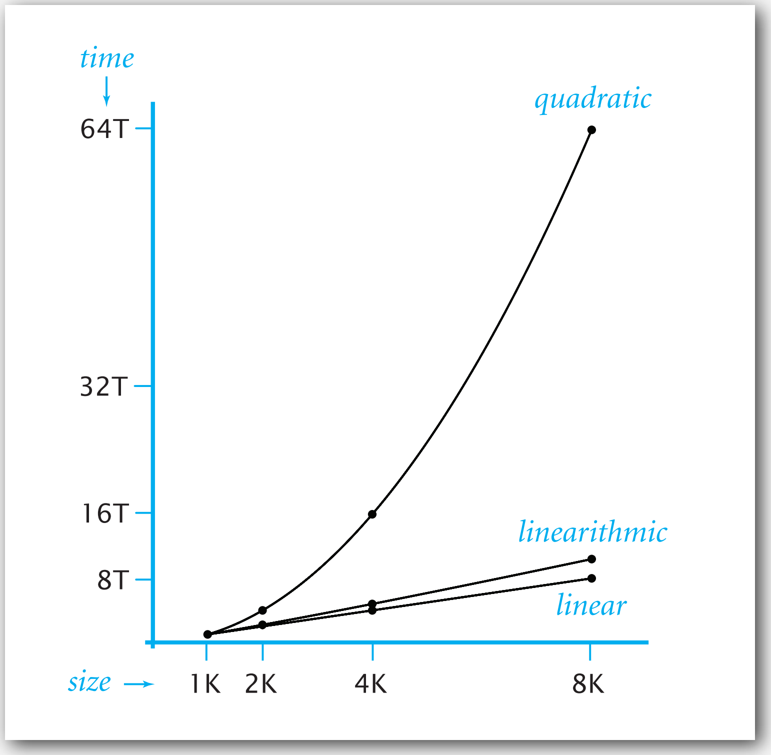
client gets poor performance because programmer did not understand performance characteristics

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.

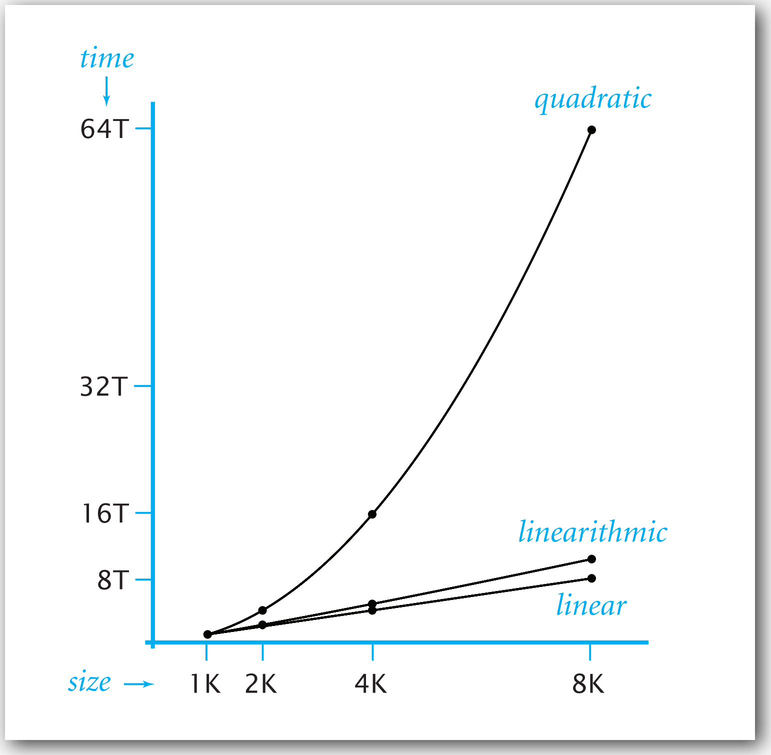
Freidrich Gauss
1805

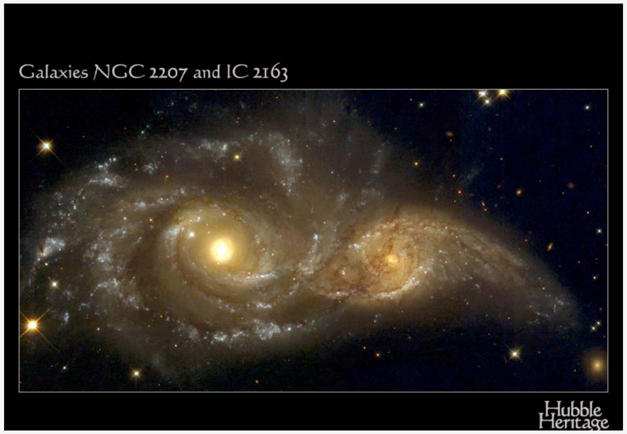


N-body Simulation.

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

Andrew Appel
PU '81





- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- measuring space

Scientific analysis of algorithms

A framework for predicting performance and comparing algorithms.

Scientific method.

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

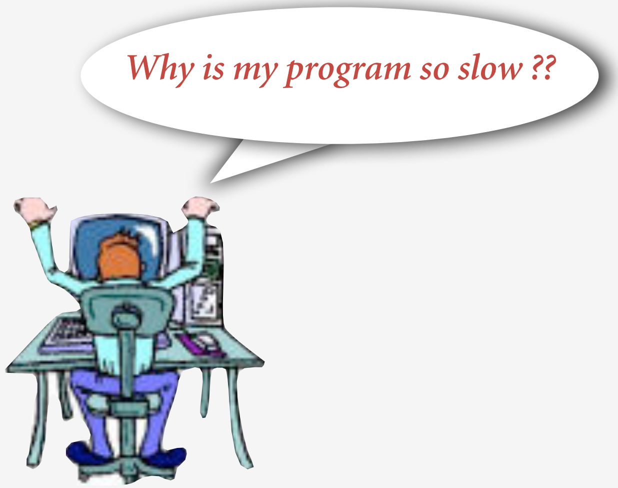
Principles.

- Experiments must be reproducible.
- Hypotheses must be falsifiable.

Universe = computer itself.

Experimental algorithmics

Every time you run a program you are doing an experiment!



First step. Debug your program!

Second step. Choose input model for experiments.

Third step. Run and time the program for problems of increasing size.

Example: 3-sum

3-sum. Given N integers, find all triples that sum to exactly zero.

```
% more input8.txt
8
 30 -30 -20 -10 40 0 10 5

% java ThreeSum < input8.txt
 4
 30 -30   0
 30 -20 -10
-30 -10  40
-10   0  10
```

Context. Deeply related to problems in computational geometry.

3-sum: brute-force algorithm

```
public class ThreeSum
{
   public static int count(long[] a)
   {
      int N = a.length;
      int cnt = 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)
                  cnt++;
      return cnt;
   }

   public static void main(String[] args)
   {
      long[] a = StdArrayIO.readLong1D();
      StdOut.println(count(a));
   }
}
```

check each triple

Empirical analysis

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

N	time (seconds) †
1024	0.26
2048	2.16
4096	17.18
8192	137.76

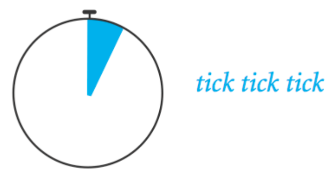
† Running Linux on Sun-Fire-X4100

Measuring the running time

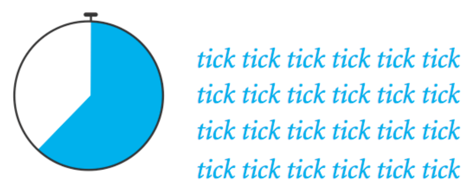
Q. How to time a program?

A. Manual.

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




```
0
% java ThreeSum < 2Kints.txt
```




```
2
391930676 -763182495 371251819
-326747290 802431422 -475684132
```

Measuring the running time

Q. How to time a program?

A. Automatic.

```
Stopwatch stopwatch = new Stopwatch();

ThreeSum.count(a);

double time = stopwatch.elapsedTime();
StdOut.println("Running time: " + time + " seconds");
```

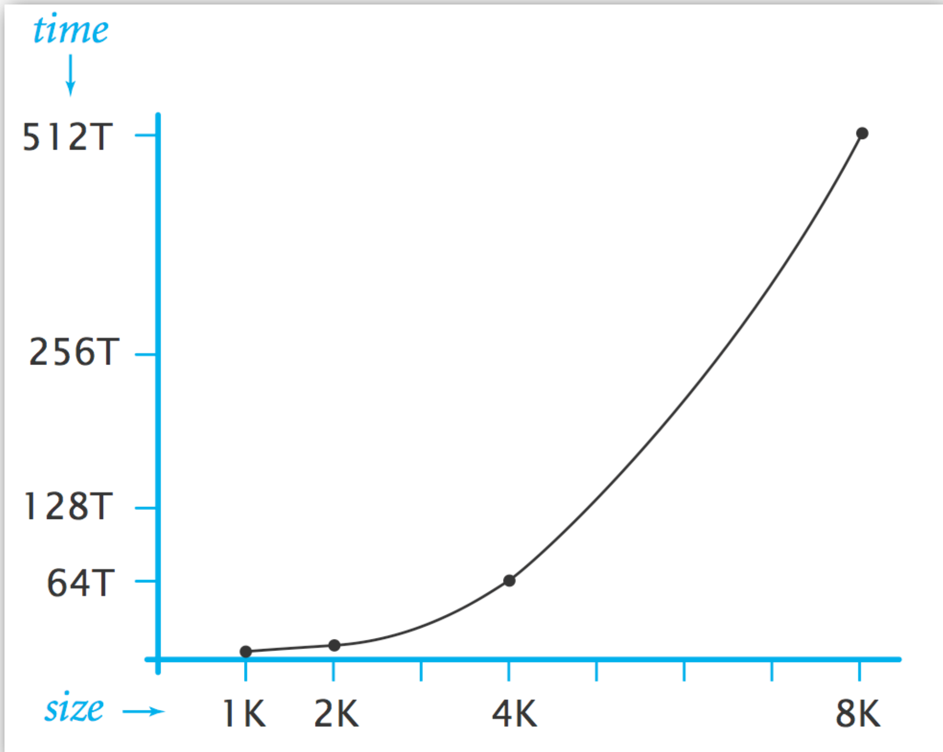
client code

```
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`, see `http://www.cs.princeton.edu/introcs/stdlib/`)

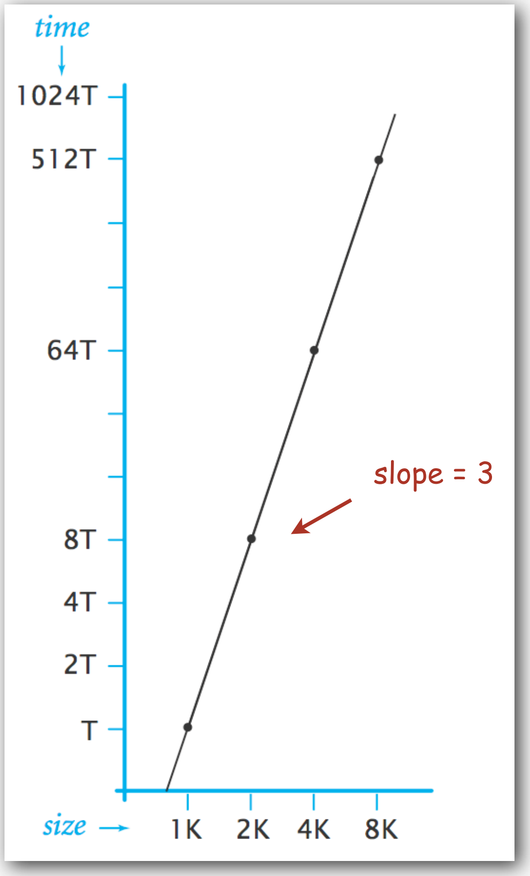
Data analysis

Plot plot running time as a function of input size N.



Data analysis

Log-log plot. Plot running time vs. input size N on log-log scale.



power law

Regression. Fit straight line through data points: $a\,N^{\,b}$.

slope

Hypothesis. Running time grows with the cube of the input size: $a\,N^{\,3}$.

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

Run program, doubling the size of the input.

N	time (seconds) †	ratio	lg ratio
512	0.03	-	
1024	0.26	7.88	2.98
2048	2.16	8.43	3.08
4096	17.18	7.96	2.99
8192	137.76	7.96	2.99

seems to converge to a constant b ≈ 3

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

Prediction and verification

Hypothesis. Running time is about $a N^3$ for input of size N.

Q. How to estimate a?

A. Run the program!

N	time (seconds)
4096	17.18
4096	17.15
4096	17.17

$$17.17 = a \times 4096^3$$
$$\Rightarrow \quad a = 2.5 \times 10^{-10}$$

Refined hypothesis. Running time is about $2.5 \times 10^{-10} \times N^3$ seconds.

Prediction. $1{,}100$ seconds for $N = 16{,}384$.

Observation.

N	time (seconds)
16384	1118.86

validates hypothesis!

Experimental algorithmics

Many obvious factors affect running time:

- Machine.
- Compiler.
- Algorithm.
- Input data.

More factors (not so obvious):

- Caching.
- Garbage collection.
- Just-in-time compilation.
- CPU use by other applications.

Bad news. It is often difficult to get precise measurements.

Good news. Easier than other sciences.

e.g., can run huge number of experiments

War story (from COS 126)

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

```
public class EditDistance
{
   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
1024	0.11
2048	0.35
4096	1.6
9182	6.5

Jenny

N	time
256	0.5
512	1.1
1024	1.9
2048	3.9

Kenny

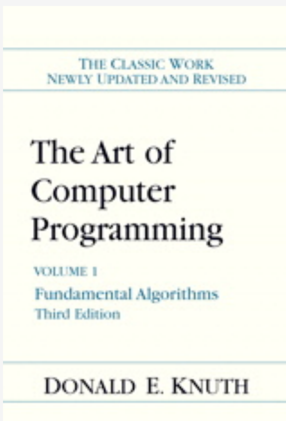
Jenny. $\sim c_1 N^2$ seconds.

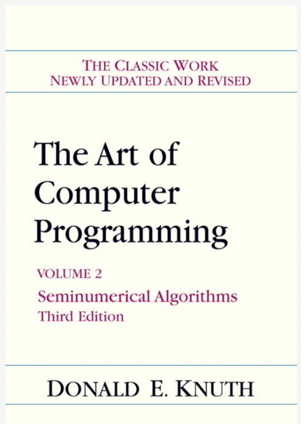
Kenny. $\sim c_2 N$ seconds.

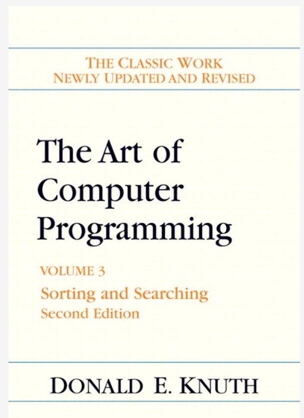
- estimating running time
- **mathematical analysis**
- order-of-growth hypotheses
- input models
- measuring space

Total running time: sum of cost × 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	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating point add	`a + b`	4.6
floating point multiply	`a * b`	4.2
floating point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

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

Cost of basic operations

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

Novice mistake. Abusive string concatenation.

Example: 1-sum

Q. How many instructions as a function of 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 comparison	$N + 1$
equal to comparison	N
array access	N
increment	$\leq 2N$

between N (no zeros)
and 2N (all zeros)

Example: 2-sum

Q. How many instructions as a function of 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 comparison	$1/2\ (N + 1)\ (N + 2)$
equal to comparison	$1/2\ N\ (N - 1)$
array access	$N\ (N - 1)$
increment	$\leq\ N^2$

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

tedious to count exactly

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. $6N^3 + 20N + 16 \sim 6N^3$

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

Ex 3. $6N^3 + \underbrace{17N^2 \lg N + 7N}_{\text{discard lower-order terms}} \sim 6N^3$

discard lower-order terms
(e.g., N = 1000 6 trillion vs. 169 million)

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

Example: 2-sum

Q. How long will it take as a function of 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"

operation	frequency	time per op	total time
variable declaration	$\sim N$	c_1	$\sim c_1 N$
assignment statement	$\sim N$	c_2	$\sim c_2 N$
less than comparison	$\sim 1/2\ N^2$	c_3	$\sim c_3 N^2$
equal to comparison	$\sim 1/2\ N^2$		
array access	$\sim N^2$	c_4	$\sim c_4 N^2$
increment	$\leq N^2$	c_5	$\leq c_5 N^2$
total			$\sim c N^2$

depends on input data

Example: 3-sum

Q. How many instructions as a function of 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)
            count++;
```

~ 1

$\sim N$

$\sim N^2 / 2$

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

"inner loop"

may be in inner loop, depends on input data

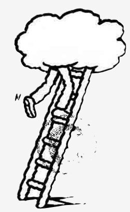
Remark. Focus on instructions in inner loop; ignore everything else!

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.



costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

A = variable declarations
B = assignment statements
C = compare
D = array access
E = increment

frequencies
(depend on algorithm, input)

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

- estimating running time
- mathematical analysis
- **order-of-growth hypotheses**
- input models
- measuring space

Common order-of-growth hypotheses

To determine order-of-growth:

- Assume a power law $T_N \sim a\, N^{\,b}$.
- Estimate exponent b with doubling hypothesis.
- Validate with mathematical analysis.

Ex. **`ThreeSumDeluxe.java`**

Food for precept. How is it implemented?

N	time (seconds) †
1,000	0.43
2,000	0.53
4,000	1.01
8,000	2.87
16,000	11.00
32,000	44.64
64,000	177.48

observations

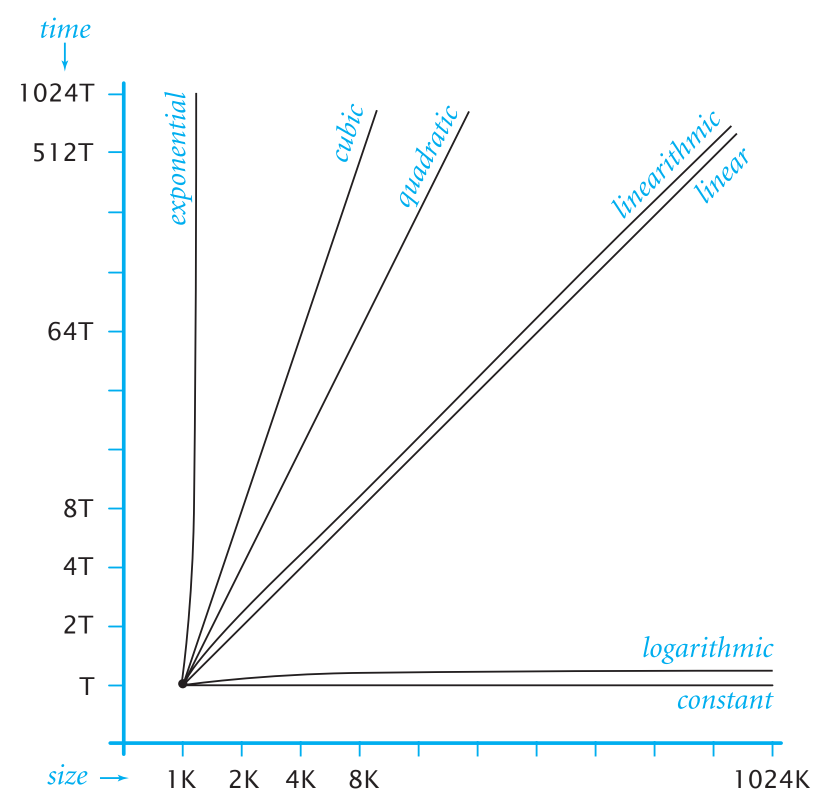
Caveat. Can't identify logarithmic factors with doubling hypothesis.

Common order-of-growth hypotheses

Good news. the small set of functions

$$1,\ \log N,\ N,\ N \log N,\ N^2,\ N^3, \text{ and } 2^N$$

suffices to describe order-of-growth of typical algorithms.



Orders of growth (log-log plot)

Common order-of-growth hypotheses

growth rate	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 lecture 5]	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 lecture 24]	exhaustive search	check all possibilities	T(N)

Practical implications of order-of-growth

Q. How many inputs can be processed in minutes?

Ex. Customers lost patience waiting "minutes" in 1970s; they still do.

Q. How long to process millions of inputs?

Ex. Population of NYC was "millions" in 1970s; still is.

For back-of-envelope calculations, assume:

decade	processor speed	instructions per second
1970s	1 MHz	10^6
1980s	10 MHz	10^7
1990s	100 MHz	10^8
2000s	1 GHz	10^9

seconds	equivalent
1	1 second
10	10 seconds
10^2	1.7 minutes
10^3	17 minutes
10^4	2.8 hours
10^5	1.1 days
10^6	1.6 weeks
10^7	3.8 months
10^8	3.1 years
10^9	3.1 decades
10^{10}	3.1 centuries
…	forever
10^{17}	age of universe

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

- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- **input models**
- measuring space

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 guarantee for all inputs

Average case. "Expected" cost

- need a model for "random" input
- provides a way to predict performance

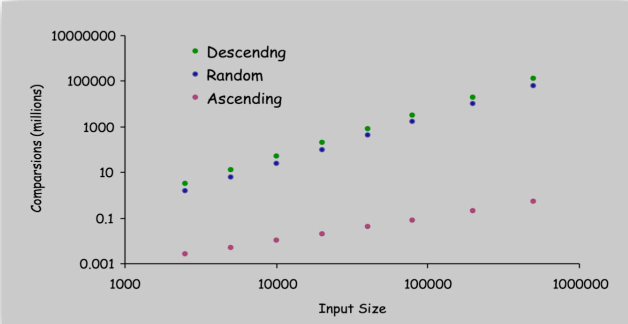
Ex 1. Array accesses for 3-sum

- Best: ~ $\frac{1}{2}N^2$.
- Average: ~ $\frac{1}{2}N^2$
- Worst: ~ $\frac{1}{2}N^2$

Ex 2. Compares for insertion sort

- Best: N-1.
- Average: ~ $\frac{1}{4}$ N^2
- Worst: $\frac{1}{2}N(N-1)$ ~ $\frac{1}{2}N^2$

(Details in Lecture 4)



Commonly-used notations

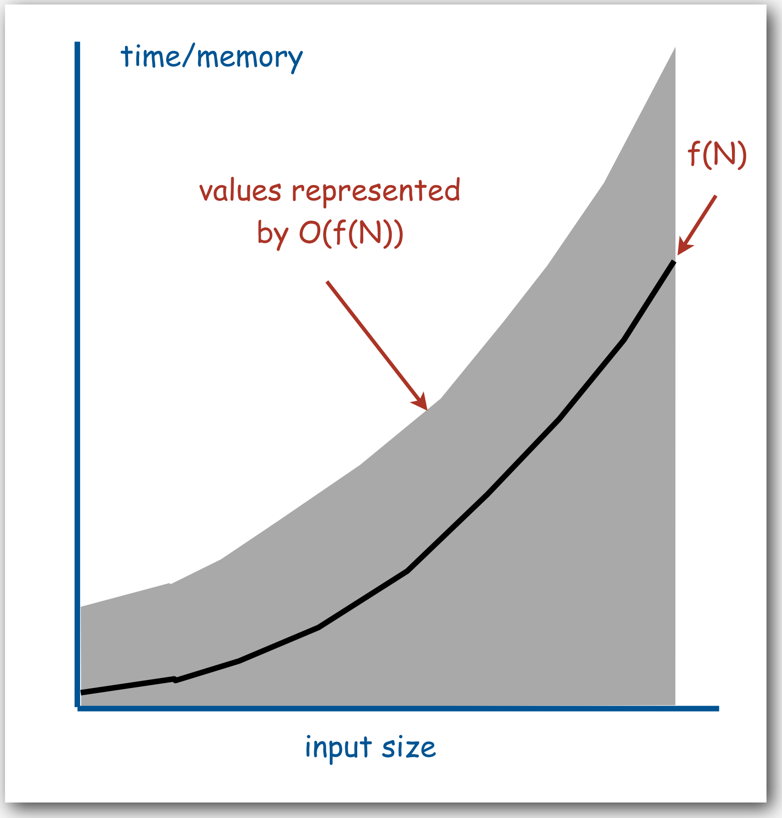
notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\ N^2$	$10\ N^2$ $10\ N^2 + 22\ N \log N$ $10\ N^2 + 2\ N + 37$	provide approximate model
Big Theta	asymptotic growth rate	$\Theta(N^2)$	N^2 $9000\ N^2$ $5\ N^2 + 22\ N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	N^2 $100\ N$ $22\ N \log N + 3\ N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$9000\ N^2$ N^5 $N^3 + 22\ N \log N + 3\ N$	develop lower bounds

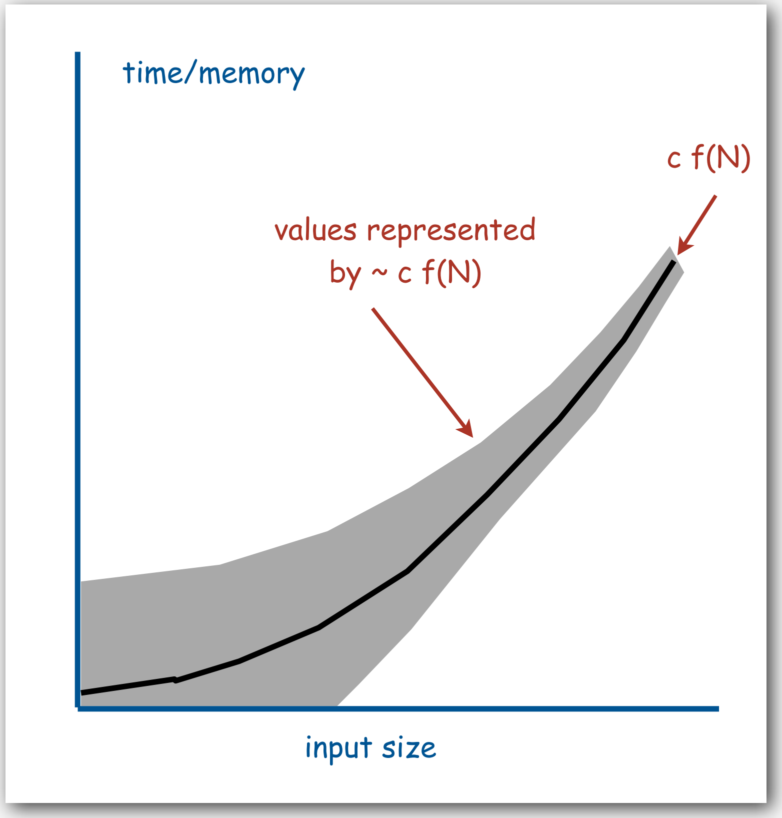
Common mistake. Interpreting big-Oh as an approximate model.

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).





- estimating running time
- mathematical analysis
- order-of-growth hypotheses
- input models
- **measuring space**

Typical memory requirements for primitive types in Java

Bit. 0 or 1.

Byte. 8 bits.

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

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

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

Typical memory requirements for arrays in Java

Array overhead. 16 bytes.

type	bytes
`char[]`	$2N + 16$
`int[]`	$4N + 16$
`double[]`	$8N + 16$

one-dimensional arrays

type	bytes
`char[][]`	$2N^2 + 20N + 16$
`int[][]`	$4N^2 + 20N + 16$
`double[][]`	$8N^2 + 20N + 16$

two-dimensional arrays

Q. What's the biggest `double[][]` array you can store on your computer?

A.

typical computer in 2008 has about 2GB memory

Typical memory requirements for objects in Java

Object overhead. 8 bytes.

Reference. 4 bytes.

Ex 1. A `Complex` object consumes 24 bytes of memory.

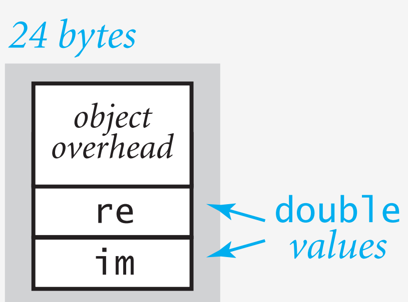
```
public class Complex
{
   private double re;
   private double im;
   ...
}
```

8 bytes overhead for object

8 bytes

8 bytes

24 bytes



Typical memory requirements for objects in Java

Object overhead. 8 bytes.

Reference. 4 bytes.

Ex 2. A virgin `String` of length N consumes 2N + 40 bytes.

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

8 bytes overhead for object

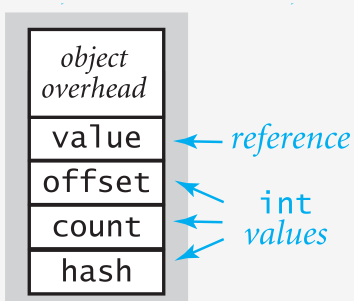
4 bytes

4 bytes

4 bytes

4 bytes for reference
(plus 2N + 16 bytes for array)

2N + 40 bytes



Example 1

Q. How much memory does this data type use as a function of N?

A.

```
public class QuickUWPC
{
   private int[] id;
   private int[] sz;

   public QuickUnion(int N)
   {
      id = new int[N];
      sz = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
      for (int i = 0; i < N; i++) sz[i] = 1;
}

   public boolean find(int p, int q) { ... }

   public void unite(int p, int q)   { ... }
}
```

Example 2

Q. How much memory does this code fragment use as a function of N?

A.

```
...
int N = Integer.parseInt(args[0]);
for (int i = 0; i < N; i++) {
    int[] a = new int[N];
    ...
}
```

Remark. Java automatically reclaims memory when it is no longer in use.

not always easy for Java to know

Turning the crank: summary

In principle, accurate mathematical models are available.

In practice, approximate mathematical models are easily achieved.

Timing may be flawed?

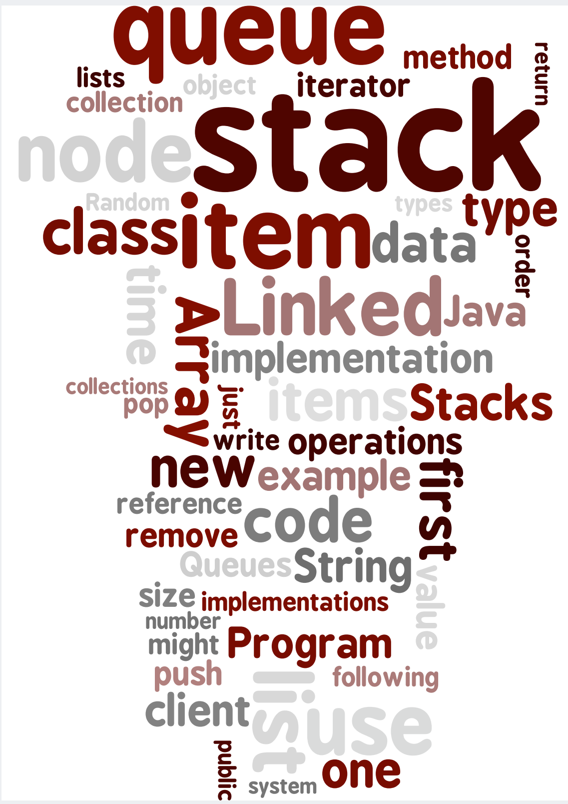
- Limits on experiments insignificant compared to other sciences.

- Mathematics might be difficult?
- Only a few functions seem to turn up.
- Doubling hypothesis cancels complicated constants.

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.

Stacks and Queues



- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Reference: Introduction to Programming in Java, Section 4.3

Stacks and queues

Fundamental data types.

- Values: sets of objects
- Operations: insert, remove, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

LIFO = "last in first out"

Stack. Remove the item most recently added.

Analogy. Cafeteria trays, Web surfing.

FIFO = "first in first out"

Queue. Remove the item least recently added.

Analogy. Registrar's line.

Client, implementation, interface

Separate interface and implementation so as to:

- Build layers of abstraction.
- Reuse software.
- Ex: stack, queue, symbol table, union-find, ...

Client: program using operations defined in interface.

Implementation: actual code implementing operations.

Interface: description of data type, basic operations.

Client, Implementation, Interface

Benefits.

- Client can't know details of implementation $\Rightarrow$ client has many implementation from which to choose.
- Implementation can't know details of client needs $\Rightarrow$ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

Client: program using operations defined in interface.
Implementation: actual code implementing operations.
Interface: description of data type, basic operations.

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Stacks

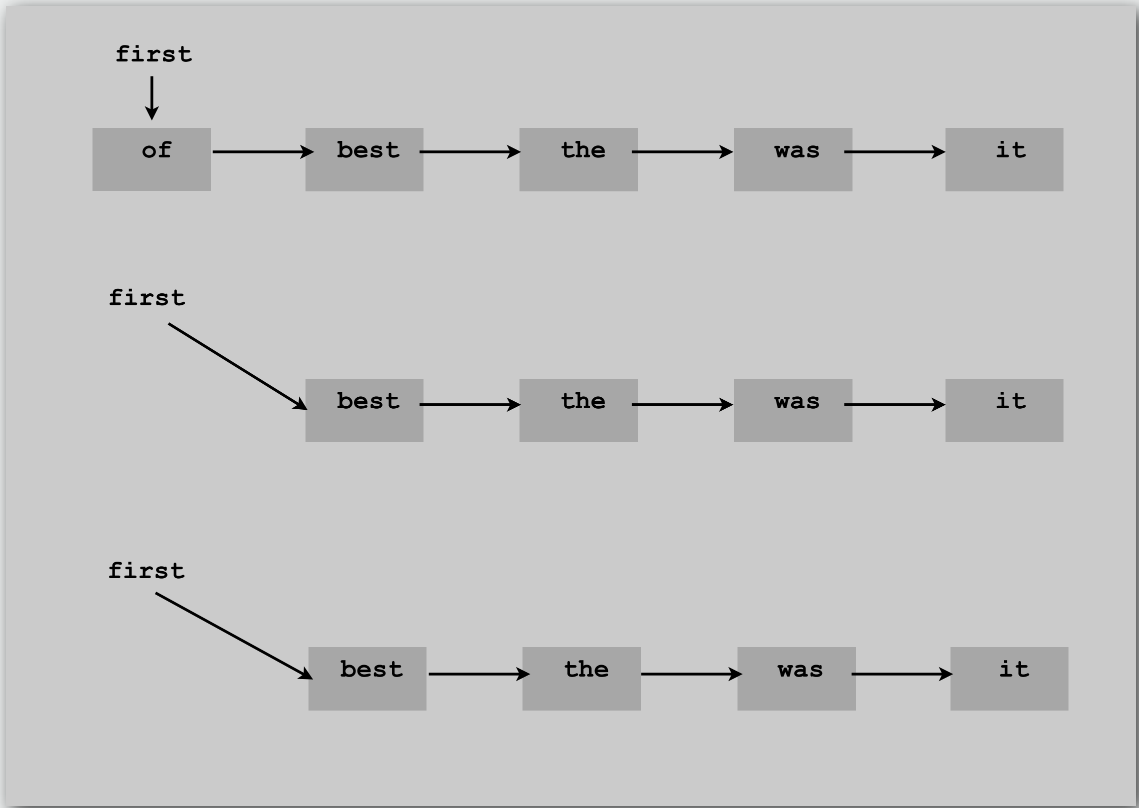
Stack operations.

- `push()` Insert a new item onto stack.
- `pop()` Remove and return the item most recently added.
- `isEmpty()` Is the stack empty?


```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(stack.pop());
      else                  stack.push(item);
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java StackOfStrings < tobe.txt
to be not that or be
```

Stack pop: linked-list implementation

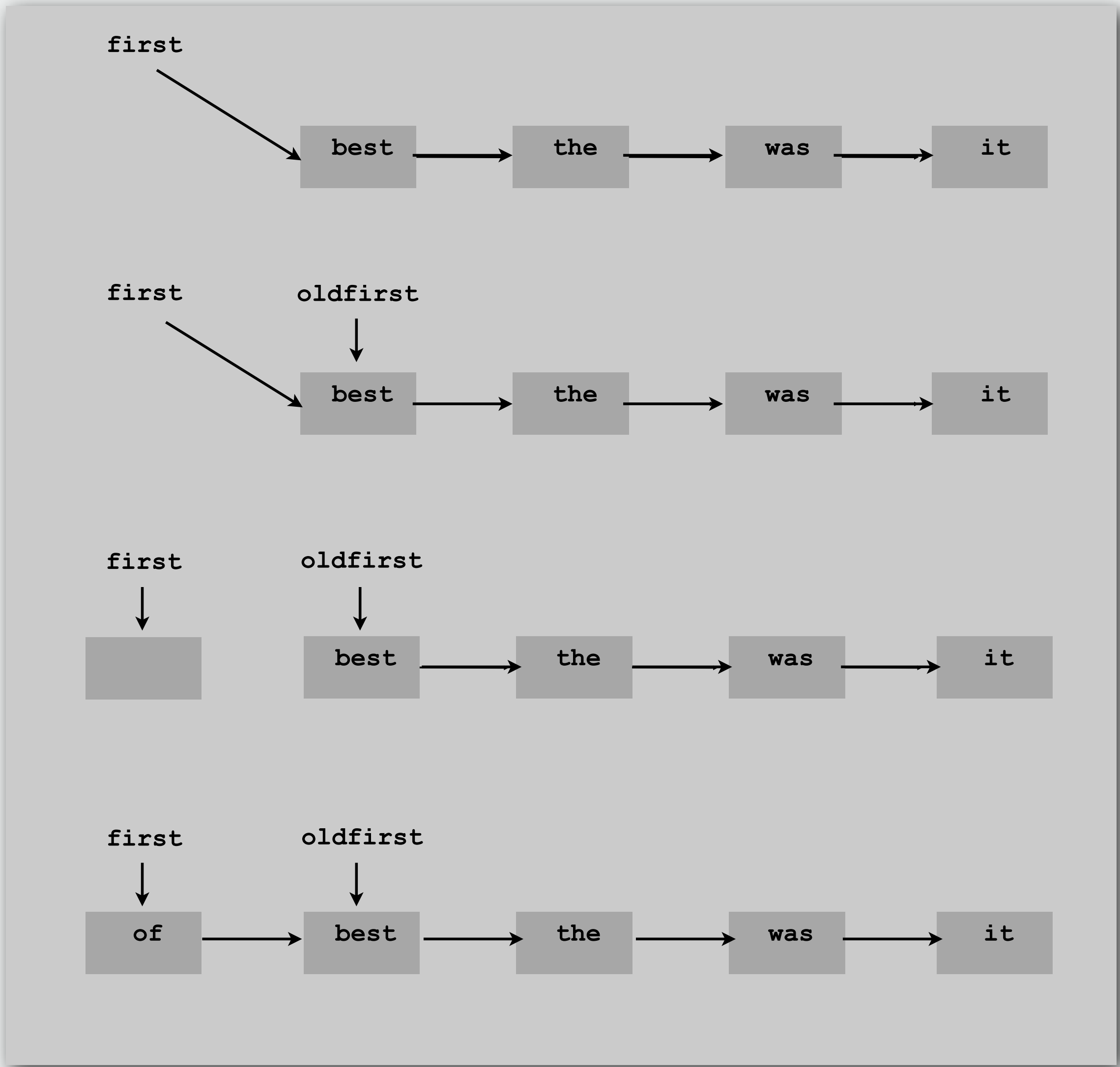



```
String item = first.item;
```

```
first = first.next;
```

```
return item;
```

Stack push: linked-list implementation




```
Node oldfirst = first;
```

```
Node first = new Node();
```

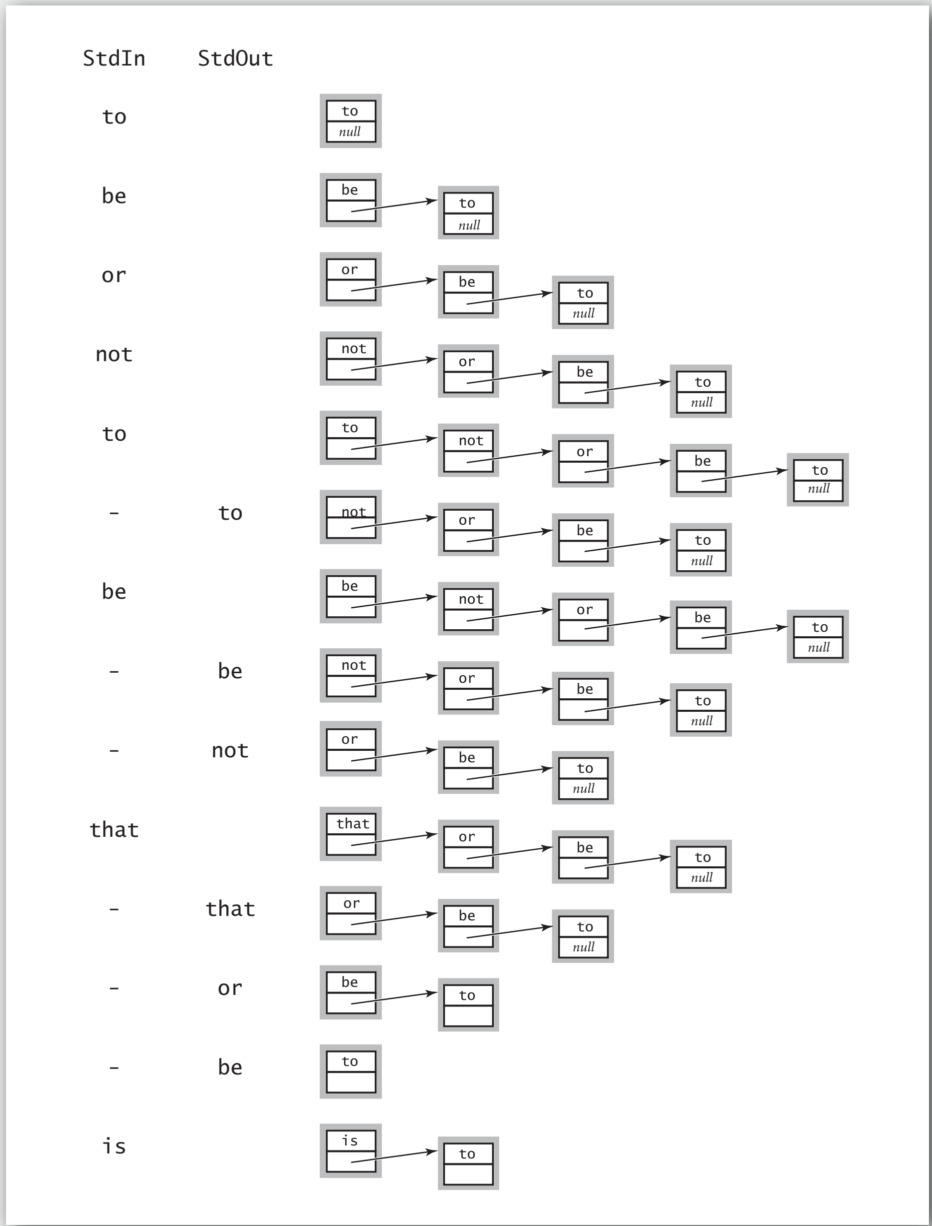
```
first.item = "of";
first.next = oldfirst;
```

Stack: linked-list implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      if (isEmpty()) throw new RuntimeException();
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

"inner class"

Stack: linked-list trace



Stack: array implementation

Array implementation of a stack.

- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.

`s[]`	`it`	`was`	`the`	`best`	`of`	`times`	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
							`N`			`capacity = 10`

Stack: array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int capacity)
   {  s = new String[capacity];  }

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

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

decrement N;
then use to index into array

```
public String pop()
{
   String item = s[--N];
   s[N] = null;
   return item;
}
```

this version avoids "loitering"

garbage collector only reclaims memory if no outstanding references

Stack: dynamic array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of `s[]` by 1.
- `pop()`: decrease size of `s[]` by 1.

Too expensive.

- Need to copy all item to a new array.
- Inserting N items takes time proportional to $1 + 2 + \ldots + N \sim N^2/2$.

infeasible for large N

Goal. Ensure that array resizing happens infrequently.

Stack: dynamic array implementation

Q. How to grow array?

"repeated doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
public StackOfStrings() {  s = new String[2];  }

public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

private void resize(int capacity)
{
   String[] dup = new String[capacity];
   for (int i = 0; i < N; i++)
      dup[i] = s[i];
   s = dup;
}
```

$1 + 2 + 4 + \ldots + N/2 + N \sim 2N$

Consequence. Inserting N items takes time proportional to N (not N^2).

Stack: dynamic array implementation

Q. How to shrink array?

First try.

- `push()`: double size of `s[]` when array is full.
- `pop()`: halve size of `s[]` when array is half full.

Too expensive

"thrashing"

- Consider push-pop-push-pop-... sequence when array is full.
- Time proportional to N per operation.

N = 5	it	was	the	best	of	*null*	*null*	*null*
N = 4	it	was	the	best				
N = 5	it	was	the	best	of	*null*	*null*	*null*
N = 4	it	was	the	best				

Stack: dynamic array implementation

Q. How to shrink array?

Efficient solution.

- `push()`: double size of `s[]` when array is full.
- `pop()`: halve size of `s[]` when array is one-quarter full.

```
public String pop()
{
   String item = s[N-1];
   s[N-1] = null;
   N--;
   if (N > 0 && N == s.length/4) resize(s.length / 2);
   s[N++] = item;
   return item;
}
```

Invariant. Array is always between 25% and 100% full.

Stack: dynamic array implementation trace

StdIn	StdOut	N	a.length	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
		0	1	*null*							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	*null*				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	*null*	*null*	*null*
-	to	4	8	to	be	or	not	*null*	*null*	*null*	*null*
be		5	8	to	be	or	not	be	*null*	*null*	*null*
-	be	4	8	to	be	or	not	*null*	*null*	*null*	*null*
-	not	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
that		4	8	to	be	or	that	*null*	*null*	*null*	*null*
-	that	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
-	or	2	4	to	be	*null*	*null*				
-	be	1	2	to	*null*						
is		2	2	to	is						

Amortized analysis

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from empty data structure, any sequence of M ops takes time proportional to M.

running time for doubling stack with N elements

	worst	best	amortized
construct	1	1	1
push	N	1	1
pop	N	1	1

doubling or shrinking

Remark. WQUPC used amortized bound: starting from empty data structure, any sequence of M union and find ops takes $O((M+N) \log^* N)$ time.

Stack implementations: memory usage

Linked list implementation. ~ 16N bytes.

```
private class Node
{
   String item;
   Node next;
}
```

8 bytes overhead for object

4 bytes

4 bytes

16 bytes

Doubling array. Between ~ 4N (100% full) and ~ 16N (25% full).

```
public class DoublingStackOfStrings
{
   private String[] s;
   private int N = 0;
```

4 bytes × array size

4 bytes

Remark. Our analysis doesn't include the memory for the items themselves.

Stack implementations: dynamic array vs. linked List

Tradeoffs. Can implement with either array or linked list; client can use interchangeably. Which is better?

Linked list.

- Every operation takes constant time in worst-case.
- Uses extra time and space to deal with the links.

Array.

- Every operation takes constant amortized time.
- Less wasted space.

Queues

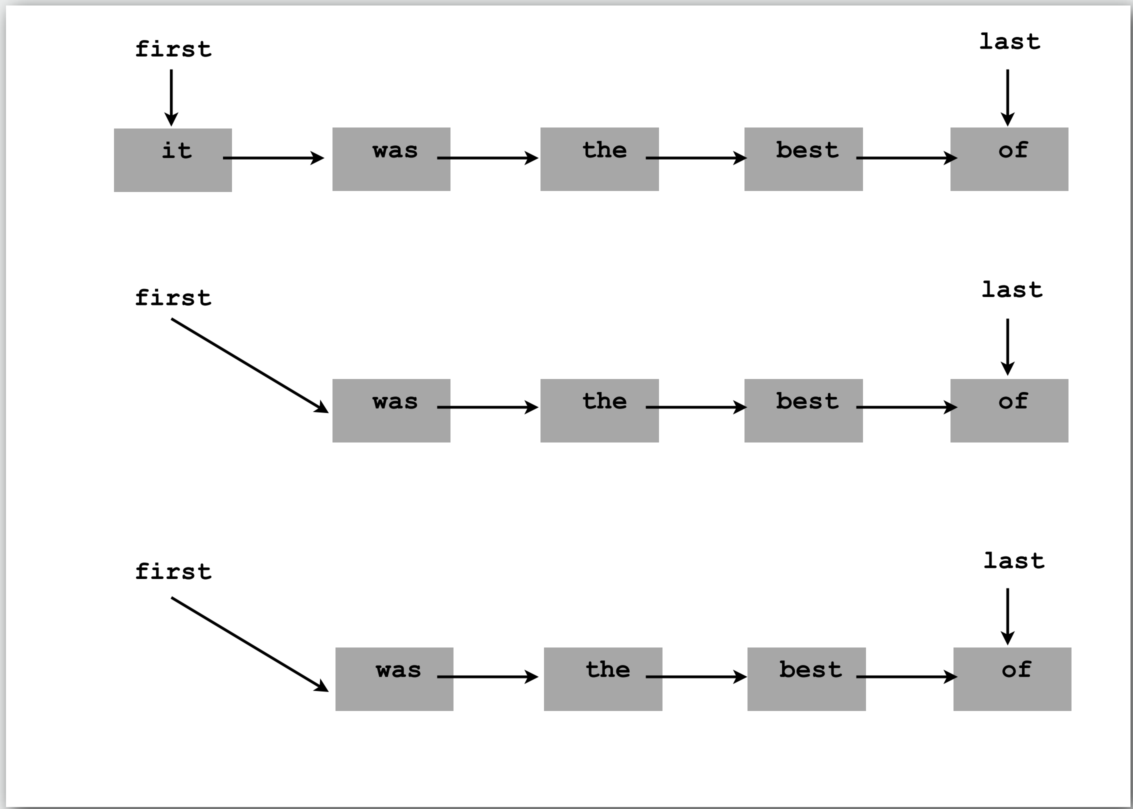
Queue operations.

- **`enqueue()`** Insert a new item onto queue.
- **`dequeue()`** Delete and return the item least recently added.
- **`isEmpty()`** Is the queue empty?

```
public static void main(String[] args)
{
   QueueOfStrings q = new QueueOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(q.dequeue());
      else                  q.enqueue(item);
      else
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java QueueOfStrings < tobe.txt
to be or not to be
```

Queue dequeue: linked list implementation

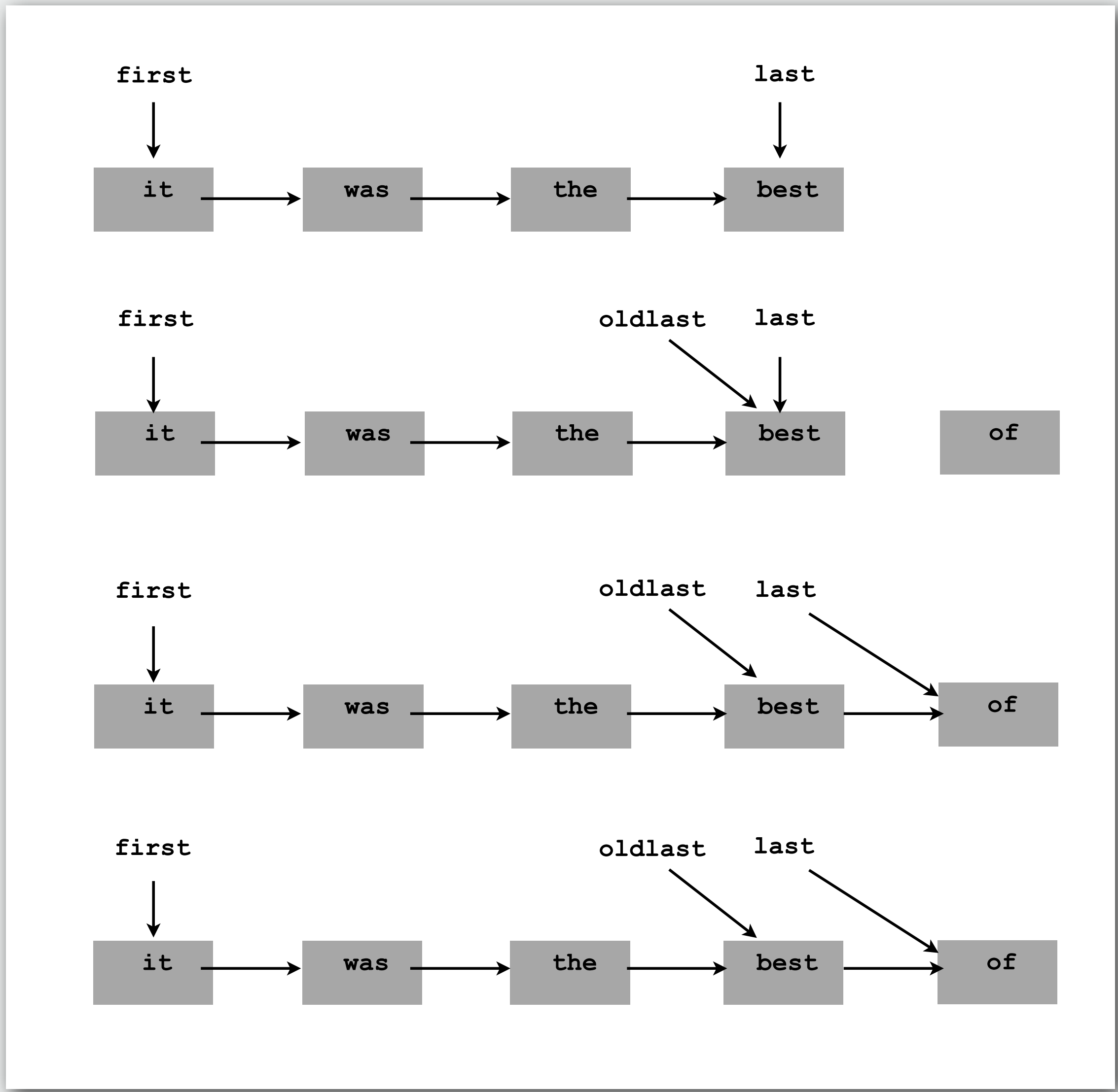



```
String item = first.item;
```

```
first = first.next;
```

```
return item;
```

Queue enqueue: linked list implementation




```
Node oldlast = last;
```

```
Node last  = new Node();
last.item = "of";
last.next = null;
```

```
oldlast.next = last;
```

Queue: linked list implementation

```
public class QueueOfStrings
{
   private Node first, last;

   private class Node
   { String item; Node next; }

   public boolean isEmpty()
   { return first == null; }

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

Queue: dynamic array implementation

Array implementation of a queue.

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.
- Add repeated doubling and shrinking.

`q[]`	*null*	*null*	`the`	`best`	`of`	`times`	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
			`head`				`tail`			

`capacity = 10`

- stacks
- dynamic resizing
- queues
- **generics**
- iterators
- applications

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5. [hence, used in AlgsJava]

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 2. Implement a stack with items of type `Object`.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 3. Java generics.

- Avoid casting in both client and implementation.
- Discover type mismatch errors at compile-time instead of run-time.

type parameter

```
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

compile-time error

Guiding principles. Welcome compile-time errors; avoid run-time errors.

Generic stack: linked list implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(String item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item;
      Node next;
   }

   public boolean isEmpty()
   {  return first == null;  }

   public void push(Item item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

generic type name

Generic stack: array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int capacity)
   {   s = new String[capacity];   }

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

   public void push(String item)
   {   s[N++] = item;   }

   public String pop()
   {   return s[--N];   }
}
```

```
public class Stack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int capacity)
   {   s = new Item[capacity];   }

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

   public void push(Item item)
   {   s[N++] = item;   }

   public Item pop()
   {   return s[--N];   }
}
```

the way it should be

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int capacity)
   {  s = new String[capacity];  }

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

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

```
public class Stack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int capacity)
   {  s = (Item[]) new Object[capacity]; }

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

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

the way it is

the ugly cast

Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

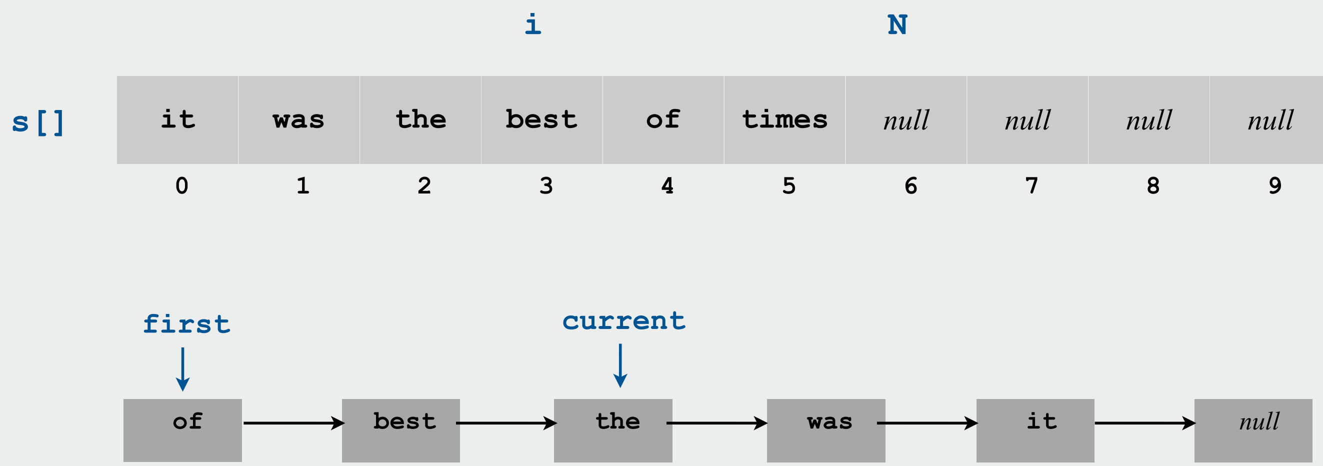
```
Stack<Integer> s = new Stack<Integer>();
s.push(17);          // s.push(new Integer(17));
int a = s.pop();     // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.



Java solution. Make stack `Iterable`.

Iterators

Q. What is an `Iterable` ?

A. Has a method that returns an `Iterator`.

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?

A. Has methods `hasNext()` and `next()`.

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();  ←  optional; use at your own risk
}
```

Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

"foreach" statement

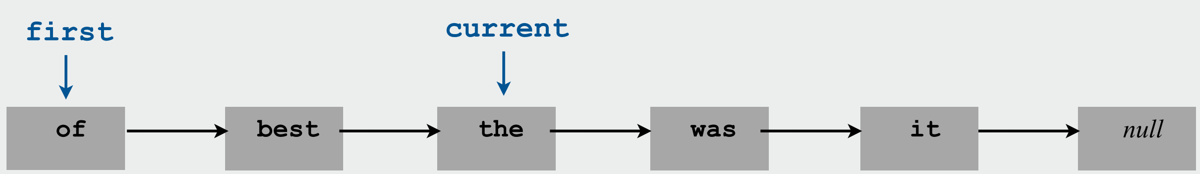
```
for (String s : stack)
   StdOut.println(s);
```

equivalent code

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```

Stack iterator: linked list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator();  }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;  }
        public void remove()     {  /* not supported */      }
        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
```



Stack iterator: array implementation

```java
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    …

    public Iterator<Item> iterator() { return new ArrayIterator(); }

    private class ArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;         }
        public void remove()     {  /* not supported */   }
        public Item next()       {  return s[--i];        }
    }
}
```

				i			N			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Stack applications

Real world applications.

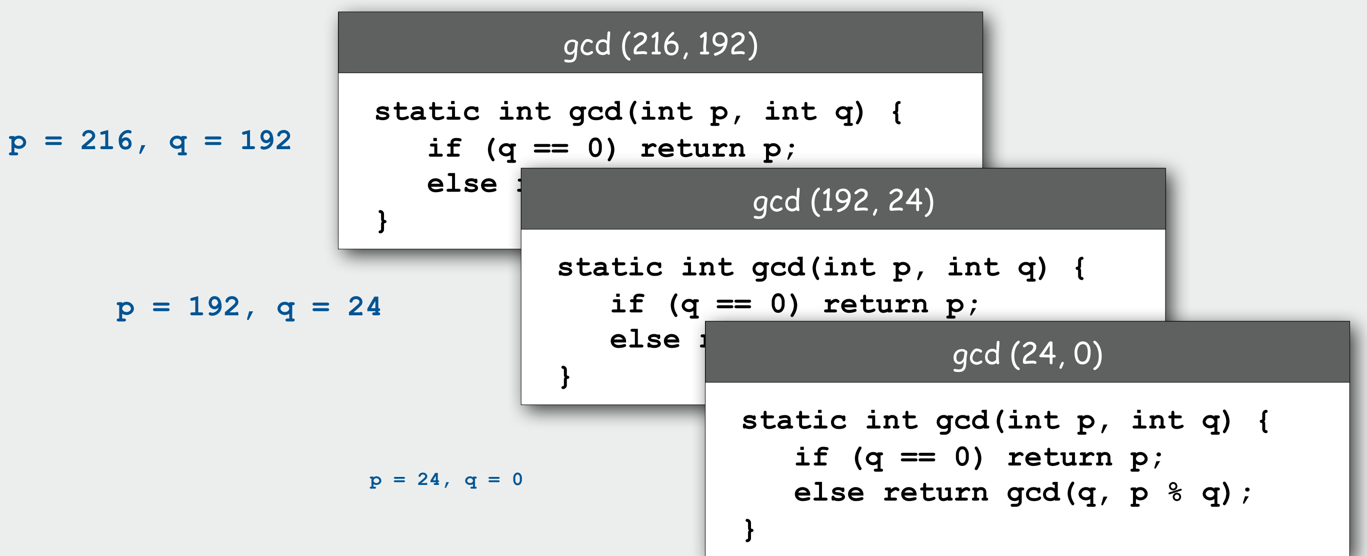
- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

How a compiler implements a function.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.

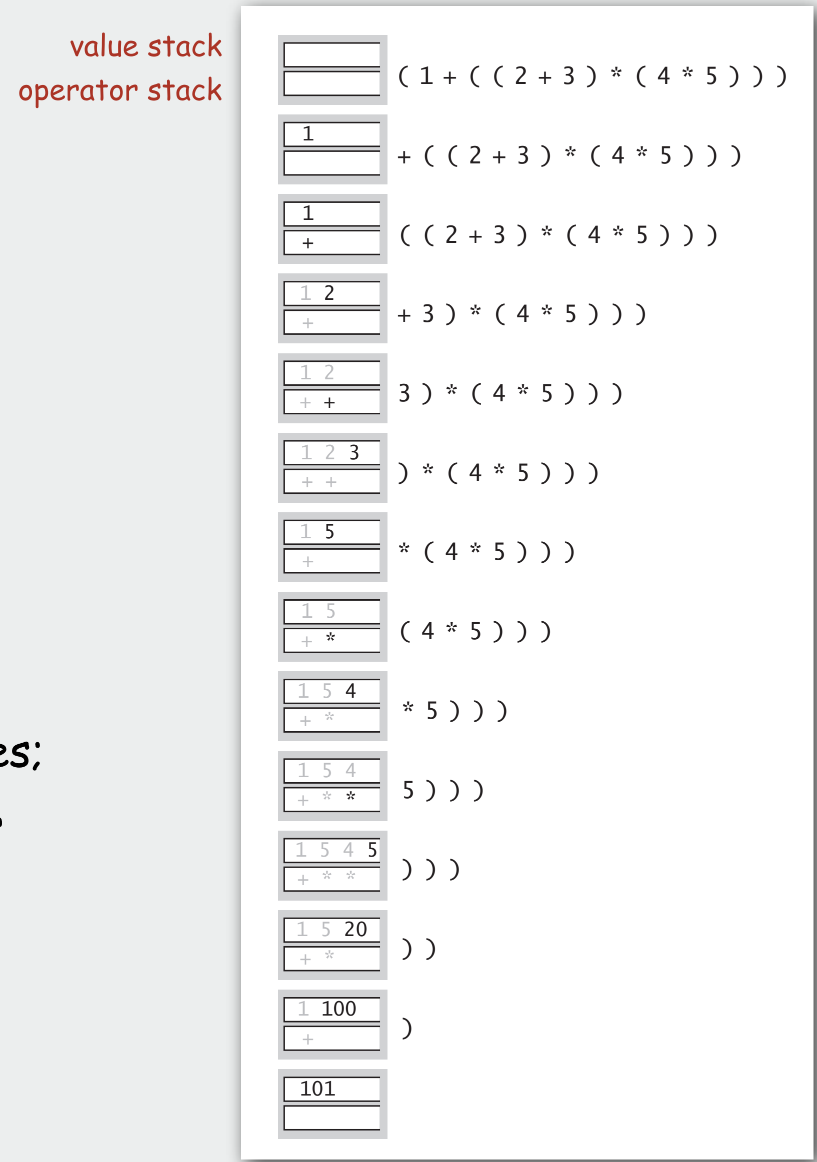


Arithmetic expression evaluation

Goal. Evaluate infix expressions.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

operand operator



Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

Arithmetic expression evaluation

```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Q. Why correct?

A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity.

Stack-based programming languages

Observation 1. The 2-stack algorithm computes the same value if the operator occurs after the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2. All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, …

PostScript

Page description language.

- Explicit stack.
- Full computational model
- Graphics engine.

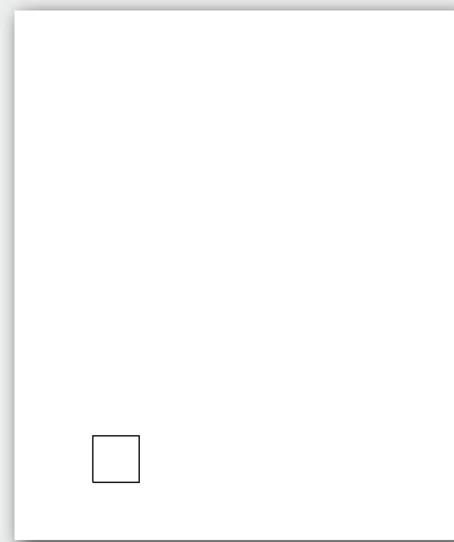
Basics.

- `%!`: "I am a PostScript program."
- Literal: "push me on the stack."
- Function calls take arguments from stack.
- Turtle graphics built in.

a PostScript program

```
%!
72 72 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
-72 0 rlineto
2 setlinewidth
stroke
```

its output



Data types.

- basic: integer, floating point, boolean, ...
- Graphics: font, path, curve, ...
- Full set of built-in operators.

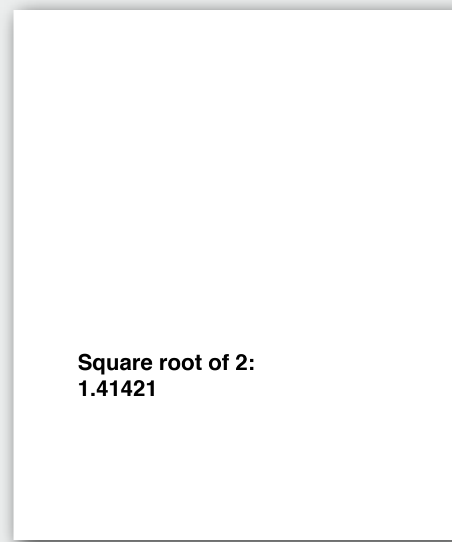
Text and strings.

System.out.print()

- Full font support.
- `show` (display a string, using current font).
- `cvs` (convert anything to a string).

toString()

```
%!
/Helvetica-Bold findfont 16 scalefont setfont
72 168 moveto
(Square root of 2:) show
72 144 moveto
2 sqrt 10 string cvs show
```



PostScript

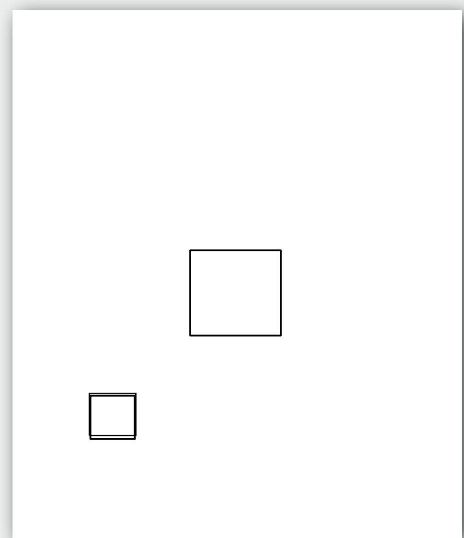
Variables (and functions).

- Identifiers start with /.
- def operator associates id with value.
- Braces.
- args on stack.

function definition

```
%!
/box
{
  /sz exch def
  0 sz rlineto
  sz 0 rlineto
  0 sz neg rlineto
  sz neg 0 rlineto
} def

72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke
```

function calls

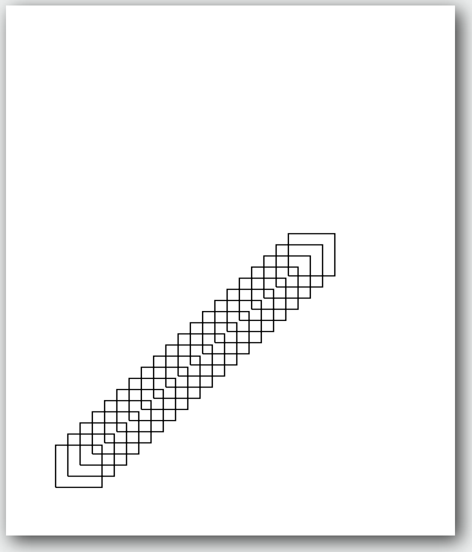


PostScript

For loop.

- "from, increment, to" on stack.
- Loop body in braces.
- `for` operator.

```
%!
\box
{
  ...
}

1 1 20
{ 19 mul dup 2 add moveto 72 box }
for
stroke
```



If-else conditional.

- Boolean on stack.
- Alternatives in braces.
- `if` operator.

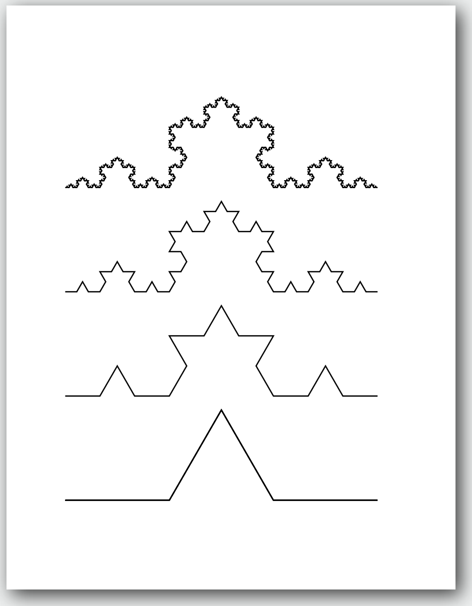
... (hundreds of operators)

PostScript

Application 1. All figures in Algorithms in Java

Application 2. Deluxe version of `StdDraw` also saves to PostScript for vector graphics.

```
%!
72 72 translate

/kochR
  {
    2 copy ge { dup 0 rlineto }
      {
        3 div
        2 copy kochR 60 rotate
        2 copy kochR -120 rotate
        2 copy kochR 60 rotate
        2 copy kochR
      } ifelse
    pop pop
  } def

0   0 moveto   81 243 kochR
0  81 moveto   27 243 kochR
0 162 moveto    9 243 kochR
0 243 moveto    1 243 kochR
stroke
```



See page 218

Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

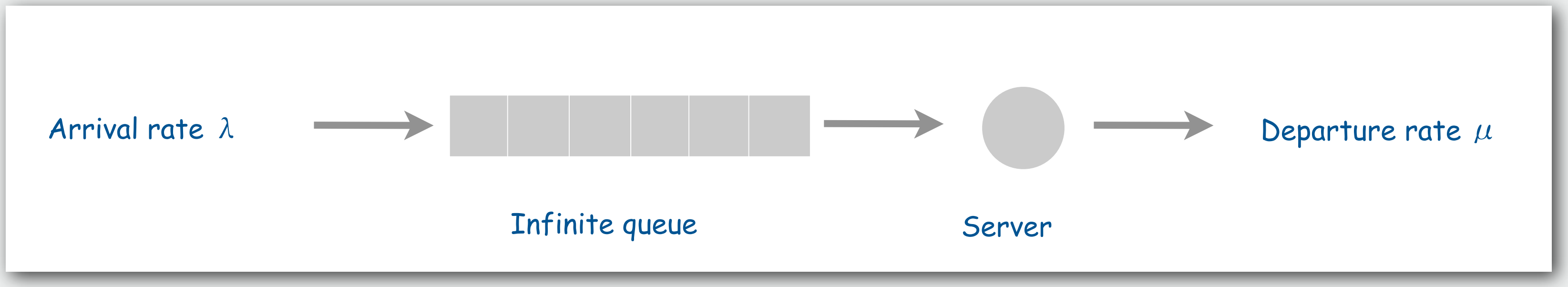
Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

M/M/1 queue.

- Customers arrive according to Poisson process at rate of λ per minute.
- Customers are serviced with rate of μ per minute.

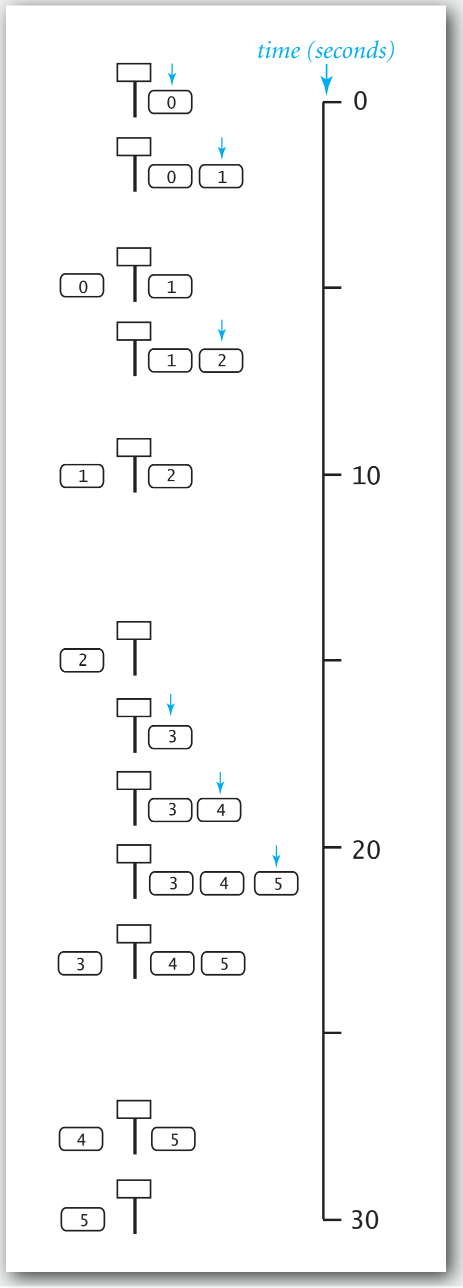
interarrival time has exponential distribution $\Pr[X \le x] = 1 - e^{-\lambda x}$
service time has exponential distribution $\Pr[X \le x] = 1 - e^{-\mu x}$



Q. What is average wait time W of a customer in system?

Q. What is average number of customers L in system?

M/M/1 queuing model: example simulation



	arrival	departure	wait
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

M/M/1 queuing model: event-based simulation

```
public class MM1Queue
{
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]);   // arrival rate
        double mu     = Double.parseDouble(args[1]);   // service rate
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);

        Queue<Double> queue = new Queue<Double>();
        Histogram hist = new Histogram("M/D/1 Queue", 60);

        while (true)
        {
            while (nextArrival < nextService)
            {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }

            double arrival = queue.dequeue();
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60,  (int) (Math.round(wait))));
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else                 nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```

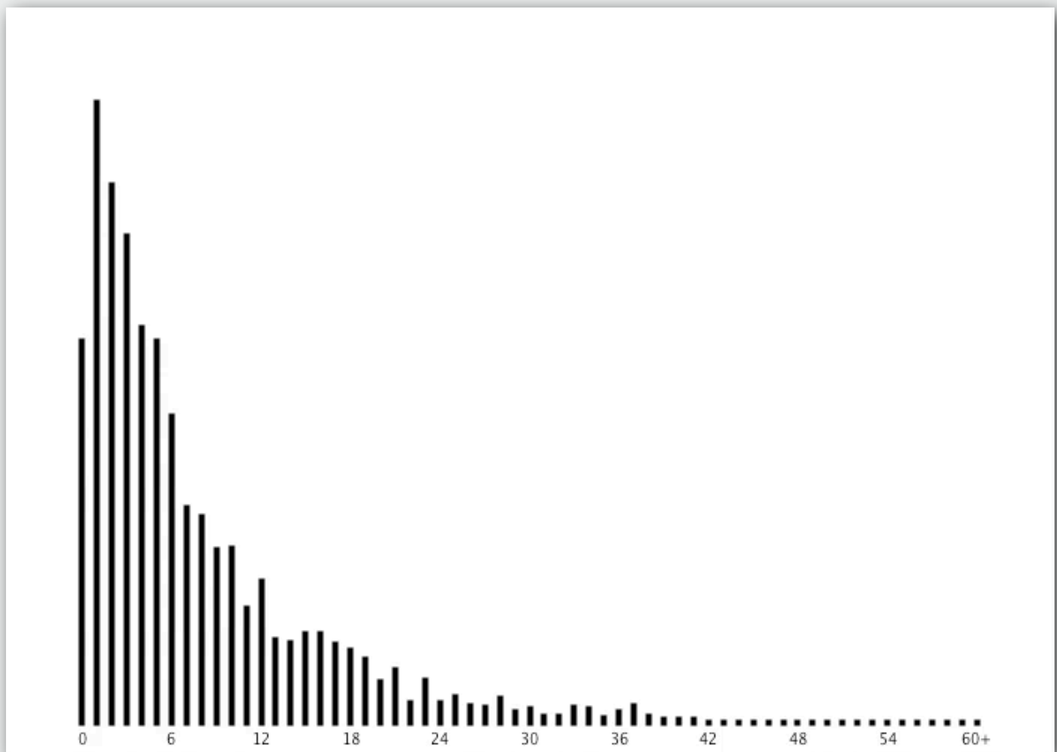
next event is an arrival

next event is a service completion

M/M/1 queuing model: experiments

Observation. If service rate μ is much larger than arrival rate λ, customers gets good service.

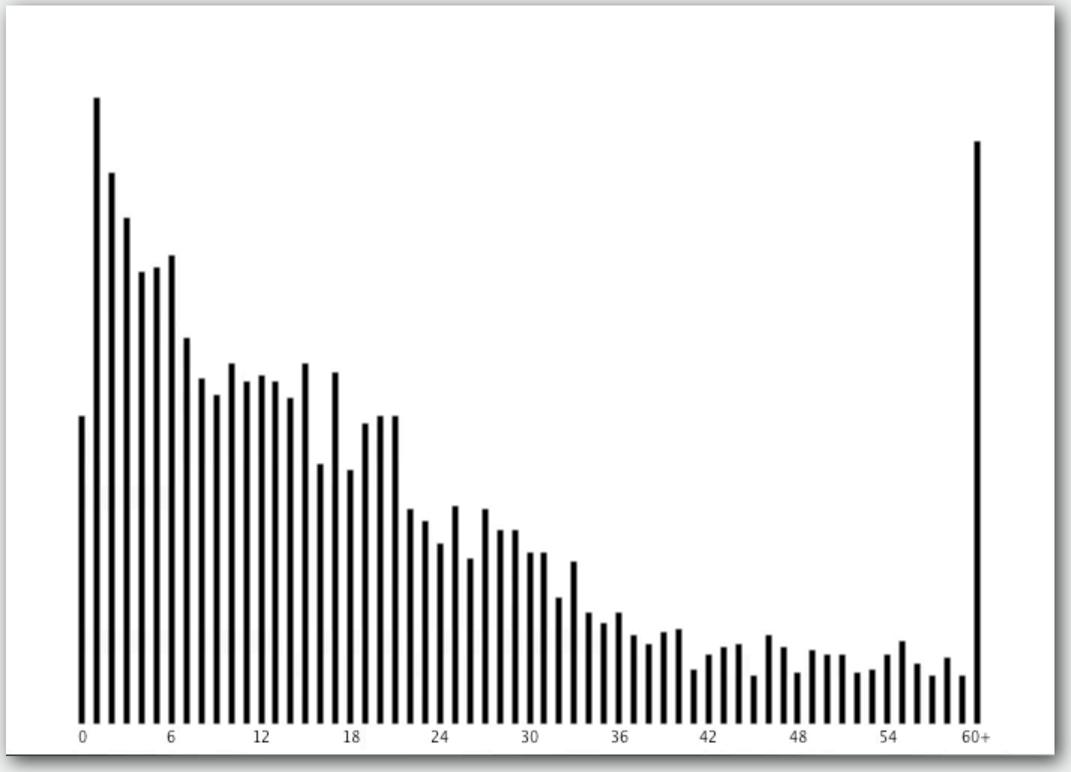
```
% java MM1Queue .2 .333
```



M/M/1 queuing model: experiments

Observation. As service rate μ approaches arrival rate λ, services goes to h***.

```
% java MM1Queue .2 .25
```



M/M/1 queuing model: experiments

Observation. As service rate μ approaches arrival rate λ, services goes to h***.

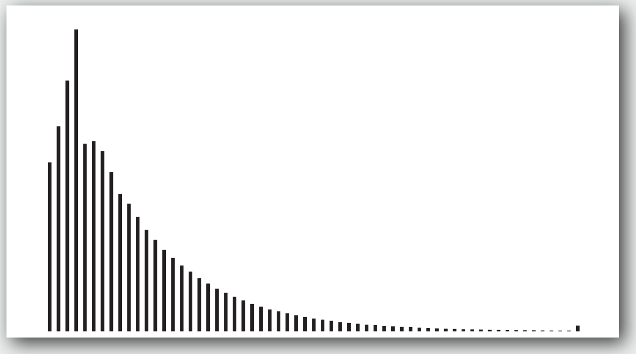
```
% java MM1Queue .2 .21
```

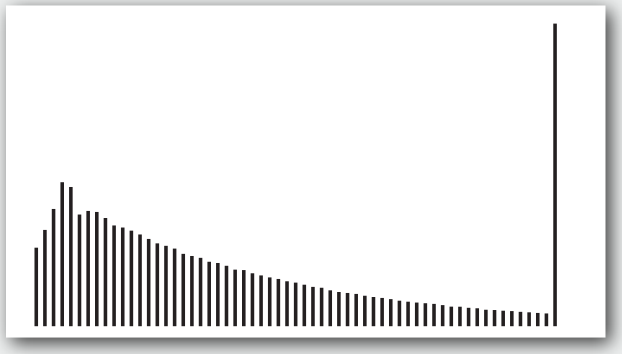
M/M/1 queue. Exact formulas known.

wait time W and queue length L approach infinity
as service rate approaches arrival rate

Little's Law

$$W = \frac{\lambda}{2\mu(\mu-\lambda)} + \frac{1}{\mu}, \quad L = \lambda W$$





More complicated queueing models. Event-based simulation essential!

Queueing theory. See ORFE 309.

Elementary Sorts

- rules of the game
- selection sort
- insertion sort
- sorting challenges
- shellsort

Reference: Algorithms in Java, 4th edition, Section 3.1

Sorting problem

Ex. Student record in a University.

file	Fox	1	A	243-456-9091	101 Brown
	Quilici	1	C	343-987-5642	32 McCosh
	Chen	2	A	884-232-5341	11 Dickinson
	Furia	3	A	766-093-9873	22 Brown
	Kanaga	3	B	898-122-9643	343 Forbes
record	Andrews	3	A	874-088-1212	121 Whitman
	Rohde	3	A	232-343-5555	115 Holder
	Battle	4	C	991-878-4944	308 Blair
key	Aaron	4	A	664-480-0023	097 Little
	Gazsi	4	B	665-303-0266	113 Walker

Sort. Rearrange array of N objects into ascending order.

Aaron	4	A	664-480-0023	097 Little
Andrews	3	A	874-088-1212	121 Whitman
Battle	4	C	991-878-4944	308 Blair
Chen	2	A	884-232-5341	11 Dickinson
Fox	1	A	243-456-9091	101 Brown
Furia	3	A	766-093-9873	22 Brown
Gazsi	4	B	665-303-0266	113 Walker
Kanaga	3	B	898-122-9643	343 Forbes
Rohde	3	A	232-343-5555	115 Holder
Quilici	1	C	343-987-5642	32 McCosh

Sample sort client

Goal. Sort any type of data.

Ex 1. Sort random numbers in ascending order.

```
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 standard input in alphabetical order.

```
public class StringSort
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAll().split("\\s+");
      Insertion.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

```
% more words3.txt
bed bug dad dot zoo ... all bad bin

% java StringSort < words.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 FileSort
{
   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]);
   }
}
```

```
% java FileSort .
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 to compare data of type `String`, `Double`, and `File` without any information about the type of an item?

Callbacks.

- Client passes array of objects to sorting routine.
- Sorting routine calls back object's compare function as needed.

Implementing callbacks.

- Java: interfaces.
- C: function pointers.
- C++: class-type functors.
- ML: first-class functions and functors.

Callbacks: roadmap

client

```
import java.io.File;
public class FileSort
{
   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]);
   }
}
```

object implementation

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

built in to Java

interface

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

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;
}
```

Key point: no reference to `File`

Comparable interface API

Comparable interface. Implement `compareTo()` so that `v.compareTo(w)`:

- Returns a negative integer if `v` is less than `w`.
- Returns a positive integer if `v` is greater than `w`.
- Returns zero if `v` is equal to `w`.

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

Total order. Implementation must ensure a total order.

- Reflexive: $(a = a)$.
- Antisymmetric: if $(a < b)$ then $(b < a)$; if $(a = b)$ then $(b = a)$.
- Transitive: if $(a \le b)$ and $(b \le c)$ then $(a \le c)$.

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

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

Implementing the Comparable interface: example 1

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

Implementing the Comparable interface: example 2

Domain names.

- Subdomain: **bolle.cs.princeton.edu**.
- Reverse subdomain: **edu.princeton.cs.bolle**.
- Sort by reverse subdomain to group by category.

```
public class Domain implements Comparable<Domain>
{
   private final String[] fields;
   private final int N;

   public Domain(String name)
   {
       fields = name.split("\\.");
       N = fields.length;
   }

   public int compareTo(Domain that)
   {
      for (int i = 0; i < Math.min(this.N, that.N); i++)
      {
         String s = fields[this.N - i - 1];
         String t = fields[that.N - i - 1];
         int cmp = s.compareTo(t);
         if      (cmp < 0) return -1;
         else if (cmp > 0) return +1;
      }
      return this.N - that.N;
   }
}
```

only use this trick when no danger of overflow

subdomains

```
ee.princeton.edu
cs.princeton.edu
princeton.edu
cnn.com
google.com
apple.com
www.cs.princeton.edu
bolle.cs.princeton.edu
```

reverse-sorted subdomains

```
com.apple
com.cnn
com.google
edu.princeton
edu.princeton.cs
edu.princeton.cs.bolle
edu.princeton.cs.www
edu.princeton.ee
```

Two useful sorting abstractions

Helper functions. Refer to data through compares and exchanges.

Less. Is object `v` less than `w` ?

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

Exchange. Swap object in array `a[]` at index `i` with the one at index `j`.

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

Testing

Q. How to test if an array is sorted?

```
private static boolean isSorted(Comparable[] a)
{
   for (int i = 1; i < a.length; i++)
      if (less(a[i], a[i-1])) return false;
   return true;
}
```

Q. If the sorting algorithm passes the test, did it correctly sort its input?

A. Yes, if data accessed only through **`exch()`** and **`less()`**.

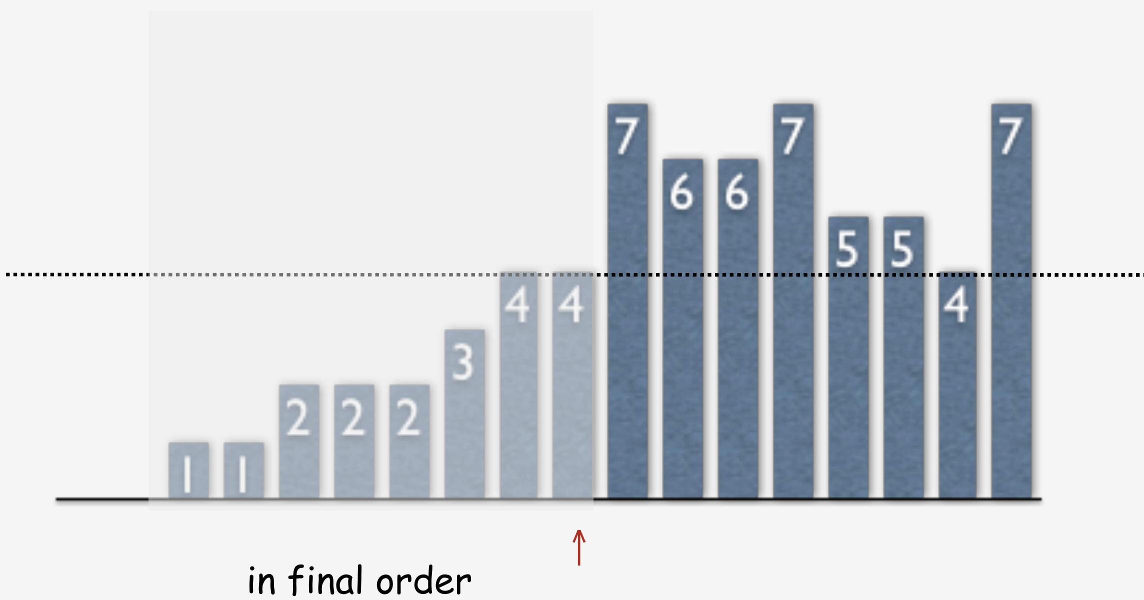
- rules of the game
- **selection sort**
- insertion sort
- sorting challenges
- shellsort

Selection sort

Algorithm. ↑ scans from left to right.

Invariants.

- Elements to the left of ↑ (including ↑) fixed and in ascending order.
- No element to right of ↑ is smaller than any element to its left.



Selection sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

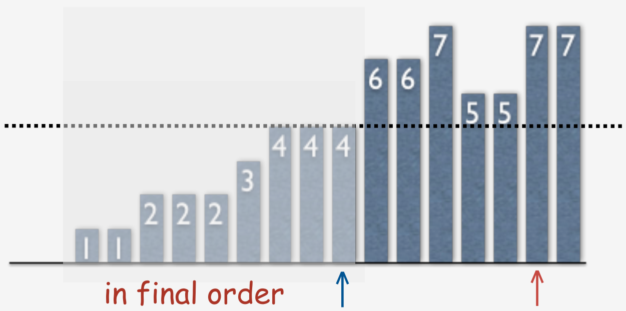
```
i++;
```

- Identify index of minimum item on right.

```
int min = i;
for (int j = i+1; j < N; j++)
   if (less(a[j], a[min]))
      min = j;
```

- Exchange into position.

```
exch(a, i, min);
```



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

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

Selection sort: mathematical analysis

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

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

entries in black are examined to find the minimum

entries in red are `a[min]`

entries in gray are in final position

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

Running time insensitive to input. Quadratic time, even if array is presorted.

Data movement is minimal. Linear number of exchanges.

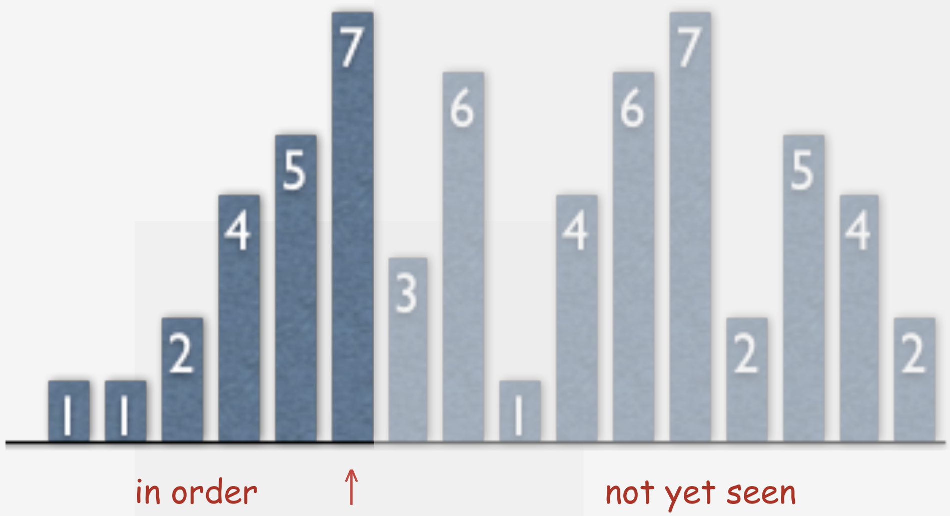
- rules of the game
- selection sort
- insertion sort
- sorting challenges
- shellsort

Insertion sort

Algorithm. ↑ scans from left to right.

Invariants.

- Elements to the left of ↑ (including ↑) are in ascending order.
- Elements to the right of ↑ have not yet been seen.



Insertion sort inner loop

To maintain algorithm invariants:

- Move the pointer to the right.

```
i++;
```

- Moving from right to left, exchange `a[i]` with each larger element to its left.

```
for (int j = i; j > 0; j--)
   if (less(a[j], a[j-1]))
         exch(a, j, j-1);
   else break;
```

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

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

Insertion sort: mathematical analysis

Proposition B. For randomly-ordered data with distinct keys, insertion sort uses ~ $N^2/4$ compares and $N^2/4$ exchanges on the average.

Pf. For randomly data, we expect each element to move halfway back.

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

entries in gray do not move

entry in red is a[j]

entries in black moved one position right for insertion

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

Insertion sort: best and worst case

Best case. If the input 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 input is in descending order (and no duplicates), insertion sort makes ~ $N^2/2$ compares and ~ $N^2/2$ exchanges.

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

Insertion sort: partially sorted inputs

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 $O(N)$.

- Ex 1. A small array appended to a large sorted array.
- Ex 2. An array with only a few elements out of place.

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

Pf. Number of exchanges equals the number of inversions.

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

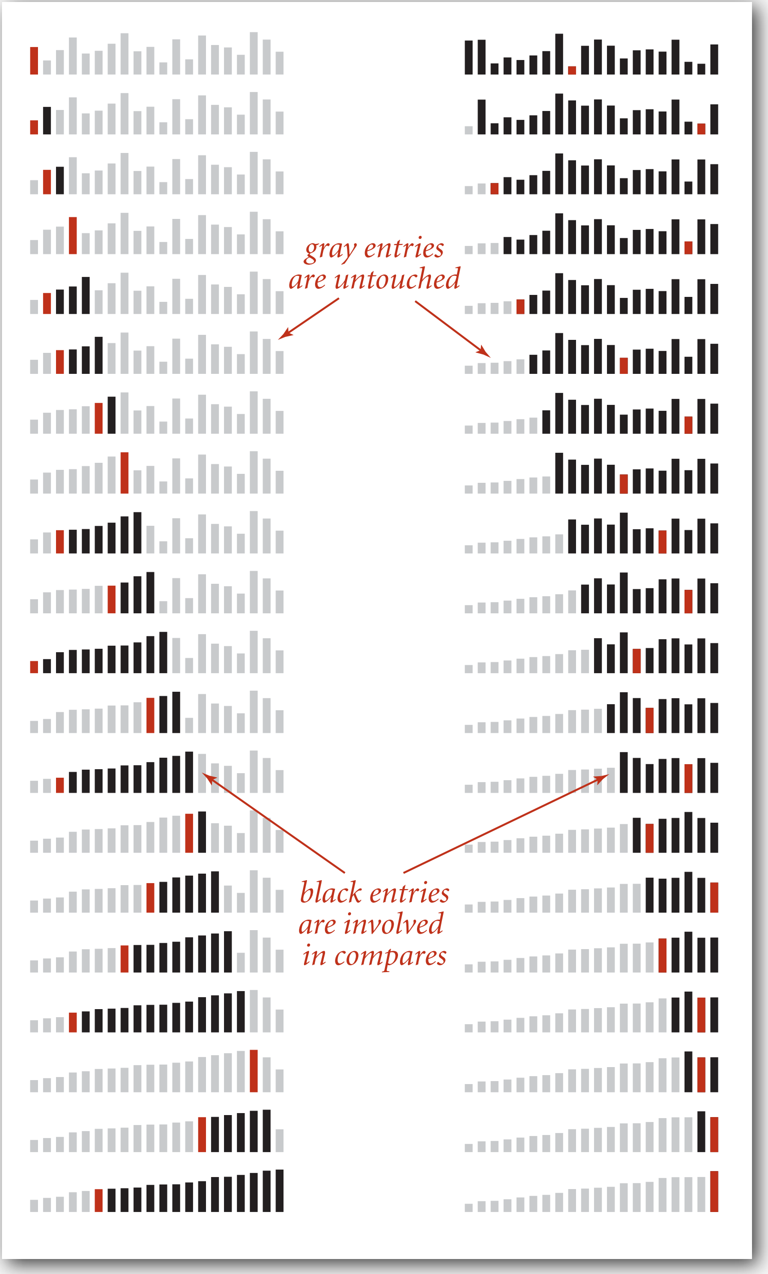
Sorting challenge 0

Input. Array of doubles.

Plot. Data proportional to length.

Name the sorting method.

- Insertion sort.
- Selection sort.



Sorting challenge 1

Problem. Sort a file of huge records with tiny keys.

Ex. Reorganize your MP3 files.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.

file → record → key →

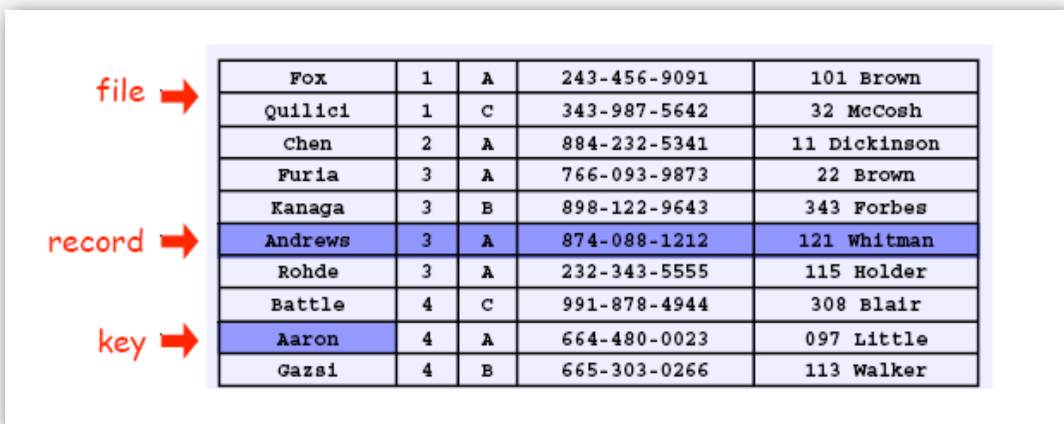
Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Problem. Sort a huge randomly-ordered file of small records.

Ex. Process transaction records for a phone company.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.



Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting challenge 3

Problem. Sort a huge number of tiny files (each file is independent)

Ex. Daily customer transaction records.

Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.

file

record

key

Fox	1	A	243-456-9091	101 Brown
Quilici	1	C	343-987-5642	32 McCosh
Chen	2	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	22 Brown
Kanaga	3	B	898-122-9643	343 Forbes
Andrews	3	A	874-088-1212	121 Whitman
Rohde	3	A	232-343-5555	115 Holder
Battle	4	C	991-878-4944	308 Blair
Aaron	4	A	664-480-0023	097 Little
Gazsi	4	B	665-303-0266	113 Walker

Sorting challenge 4

Problem. Sort a huge file that is already almost in order.

Ex. Resort a huge database after a few changes.

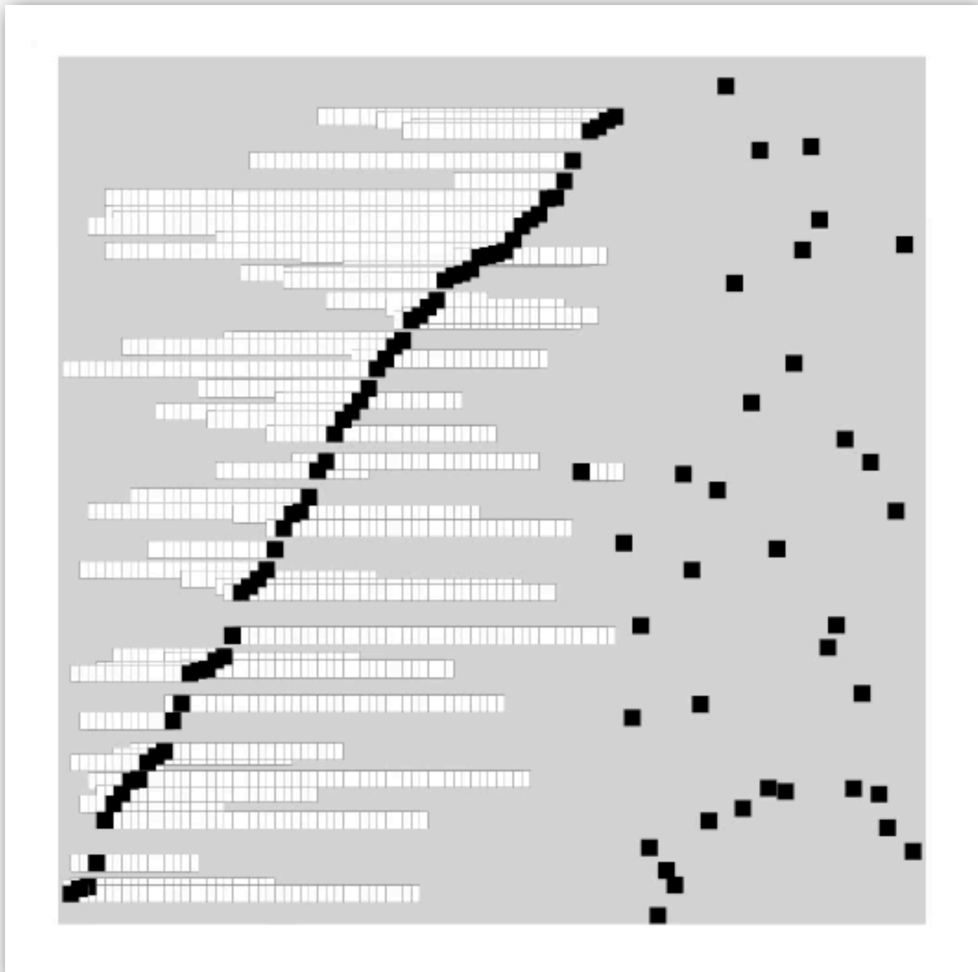
Which sorting method to use?

- System sort.
- Insertion sort.
- Selection sort.

file	Fox	1	A	243-456-9091	101 Brown
	Quilici	1	C	343-987-5642	32 McCosh
	Chen	2	A	884-232-5341	11 Dickinson
	Furia	3	A	766-093-9873	22 Brown
	Kanaga	3	B	898-122-9643	343 Forbes
record	Andrews	3	A	874-088-1212	121 Whitman
	Rohde	3	A	232-343-5555	115 Holder
	Battle	4	C	991-878-4944	308 Blair
key	Aaron	4	A	664-480-0023	097 Little
	Gazsi	4	B	665-303-0266	113 Walker

Insertion sort animation

Insertion sort animation



Reason it is slow: excessive data movement.

Shellsort overview

Idea. Move elements more than one position at a time by h-sorting the file.

an h-sorted file is h interleaved sorted files

h = 4

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

Shellsort. h-sort the file for a decreasing sequence 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 a file? Insertion sort, with stride length h.

3-sorting a file

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 subfiles.
- 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	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	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.

Pf. Harder than you'd think!

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

still 7-sorted

What increments to use?

1, 2, 4, 8, 16, 32 . . .

No.

1, 3, 7, 15, 31, 63, . . .

Maybe.

1, 4, 13, 40, 121, 363, . . .

OK, easy to compute.

1, 5, 19, 41, 109, 209, 505, . . .

Tough to beat in empirical studies.

Interested in learning more?

- See Algs 3 section 6.8 or Knuth volume 3 for details.

Shellsort: Java implementation

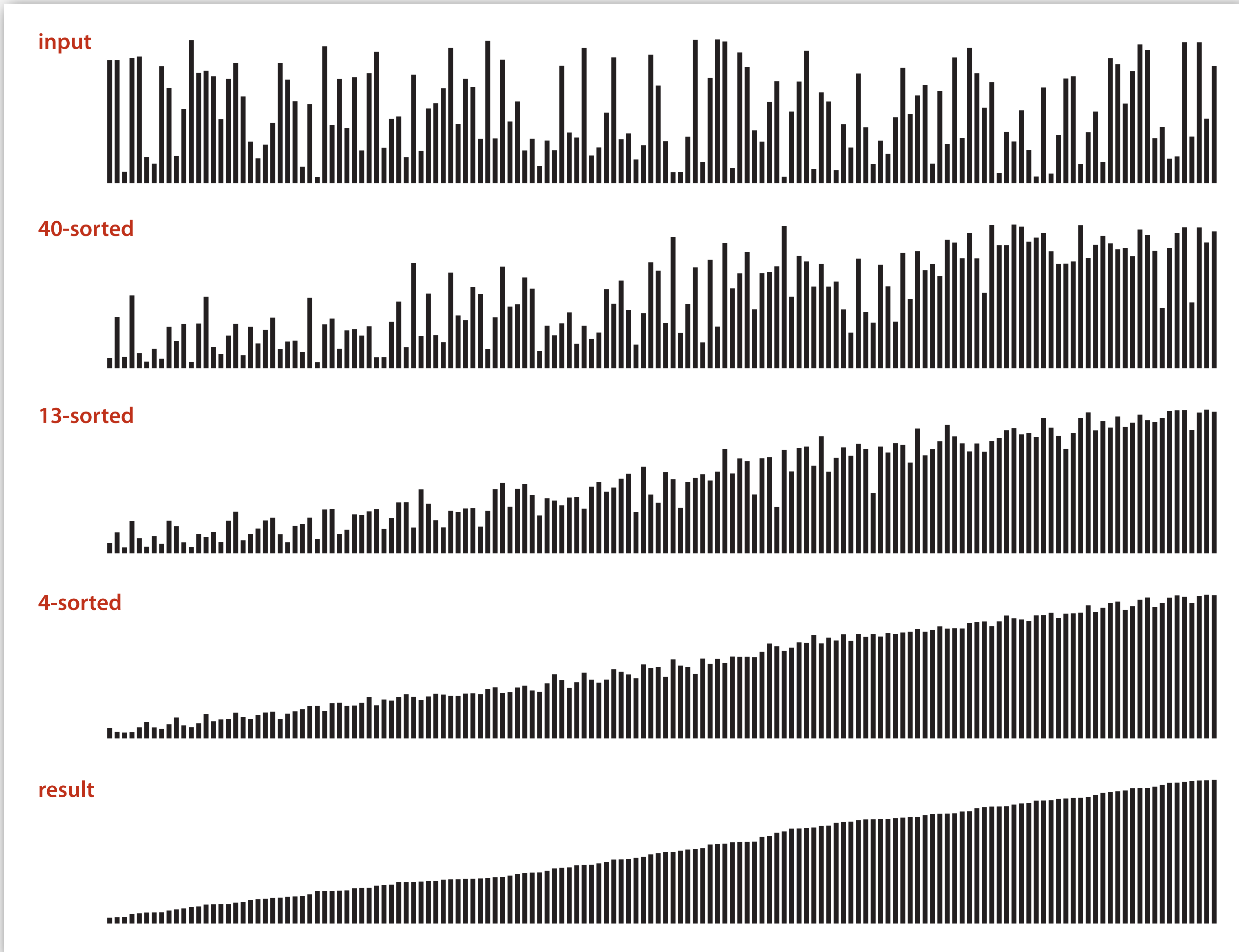
```
public class Shell
{  // Shellsort.
   public static void sort(Comparable[] a)
   {  // Sort a[] into increasing order.
      int N = a.length;

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

      while (h >= 1)
      {  // h-sort the file.
         for (int i = h; i < N; i++)
         {  // Insert a[i] among a[i-h], a[i-2*h], a[i-3*h]... .
            for (int j = i; j > 0 && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }

         h = h/3;
      }
   }

   private boolean less(Comparable v, Comparable w)
   // As before.
   private boolean exch(Comparable[] a, int i, int j)
   // As before.
}
```

magic increment sequence

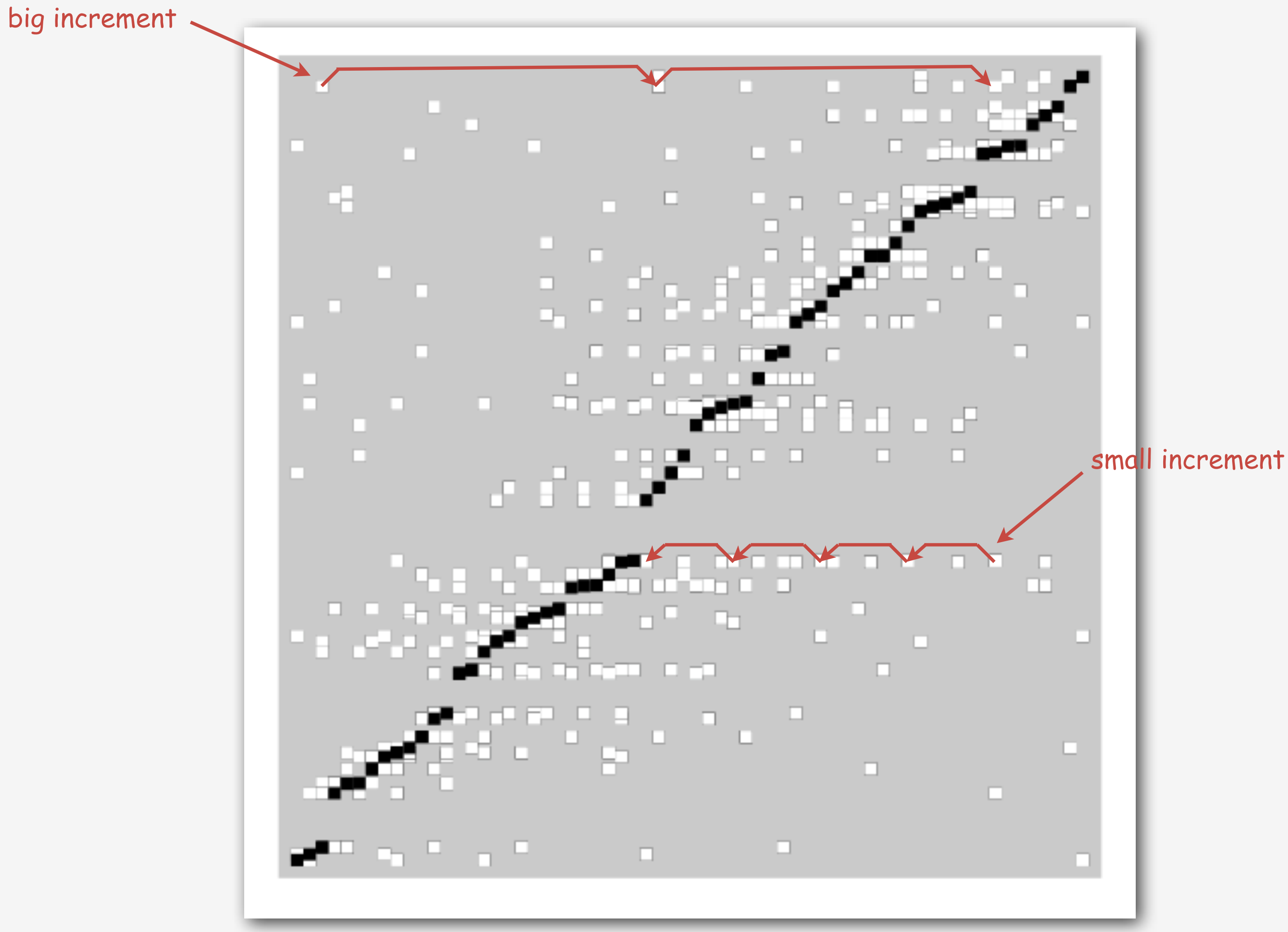
insertion sort

move to next increment

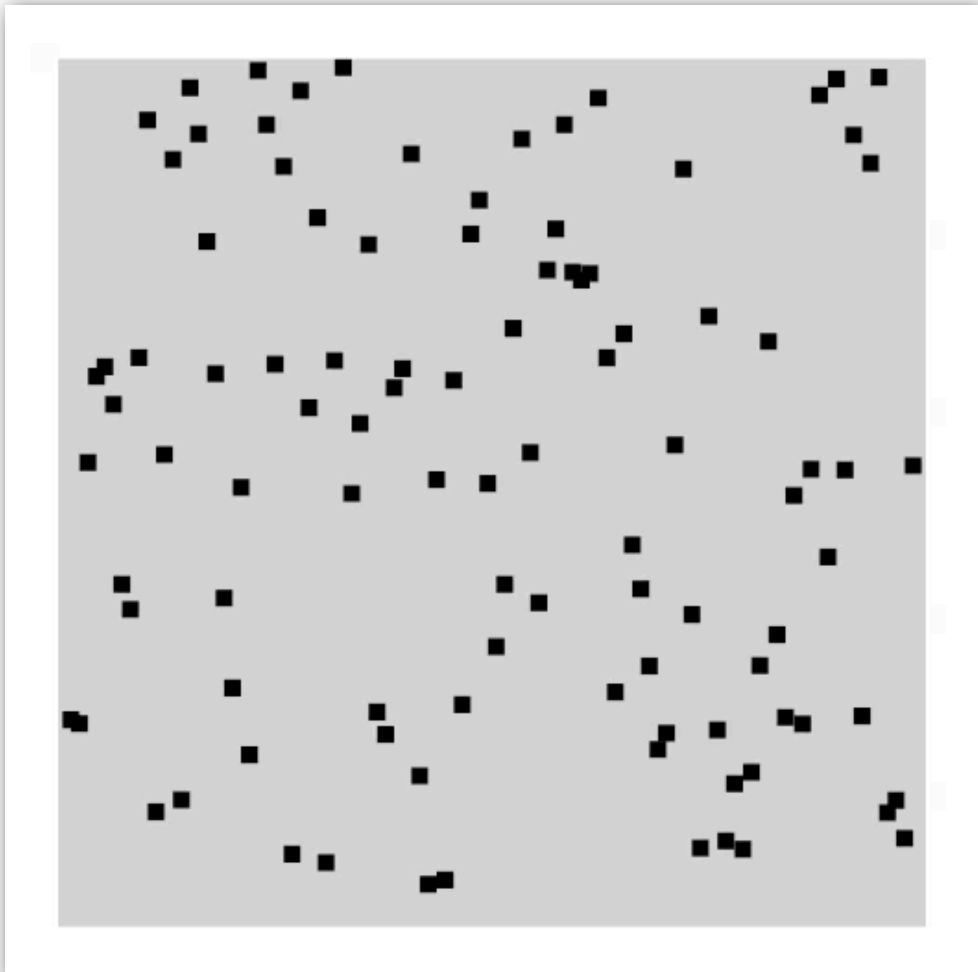
Visual trace of shellsort



Shellsort animation



Shellsort animation



Bottom line: substantially faster than insertion sort!

Shellsort: analysis

Proposition. The worst-case number of compares for shellsort using the increments 1, 4, 13, 40, ... is $O(N^{3/2})$.

Property. The number of compares used by shellsort with the 3x+1 increments is at most by a small multiple of N times the # of increments used.

N	compares	$N^{1.289}$	2.5 N lg N
5,000	93	58	106
10,000	209	143	230
20,000	467	349	495
40,000	1022	855	1059
80,000	2266	2089	2257

measured in thousands

Remark. Accurate model has not yet been discovered (!)

Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains.

Useful in practice.

- Fast unless file 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?
- Average case performance?

open problem: find a better increment sequence

Lesson. Some good algorithms are still waiting discovery.

Mergesort

- mergesort
- sorting complexity
- comparators

Reference:
Algorithms in Java. 4th Edition, Section 3.2
`http://www.cs.princeton.edu/algs4`

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

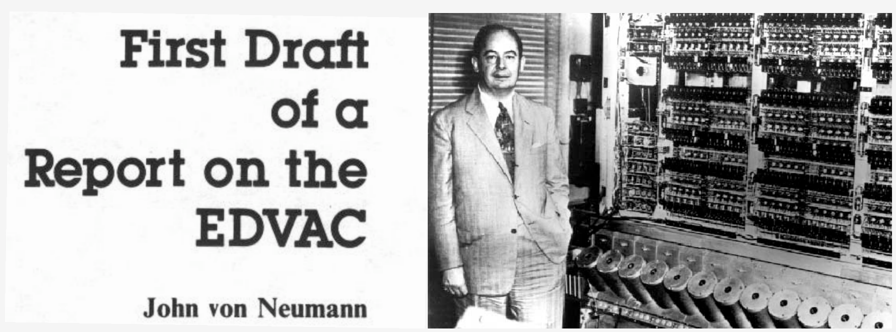
Mergesort

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	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview



Mergesort trace

	lo		hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				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,	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

Merging

Goal. Combine two sorted subarrays into a sorted whole.

Q. How to merge efficiently?

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	5										
	0	A										0	6	E	E	G	M	R	A	C	E	R	T
	1	A	C									0	7	E	E	G	M	R		C	E	R	T
	2	A	C	E								1	7	E	E	G	M	R			E	R	T
	3	A	C	E	E							2	7		E	G	M	R			E	R	T
	4	A	C	E	E	E						2	8			G	M	R			E	R	T
	5	A	C	E	E	E	G					3	8			G	M	R				R	T
	6	A	C	E	E	E	G	M				4	8				M	R				R	T
	7	A	C	E	E	E	G	M	R			5	8					R				R	T
	8	A	C	E	E	E	G	M	R	R		5	9									R	T
	9	A	C	E	E	E	G	M	R	R	T	6	10										T
merged result		A	C	E	E	E	G	M	R	R	T												

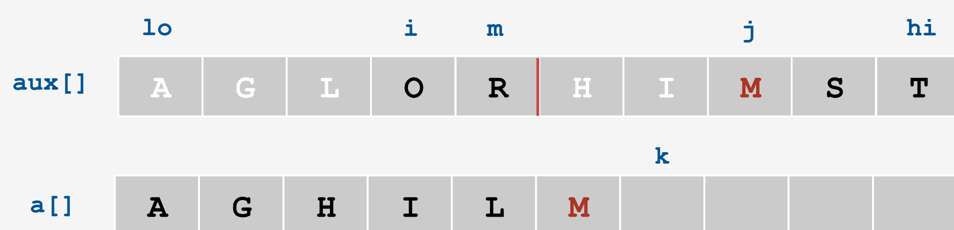
Abstract in-place merge trace

Merging: Java implementation

```
public static void merge(Comparable[] a, int lo, int m, int hi)
{  // Merge a[lo..m] with a[m+1..hi].

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

   int i = lo, j = mid;
   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++];

}
```

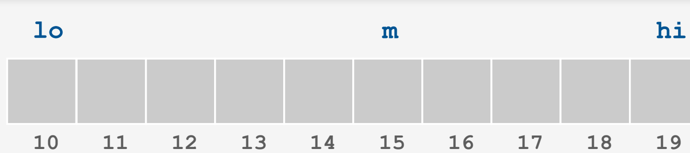
copy

merge

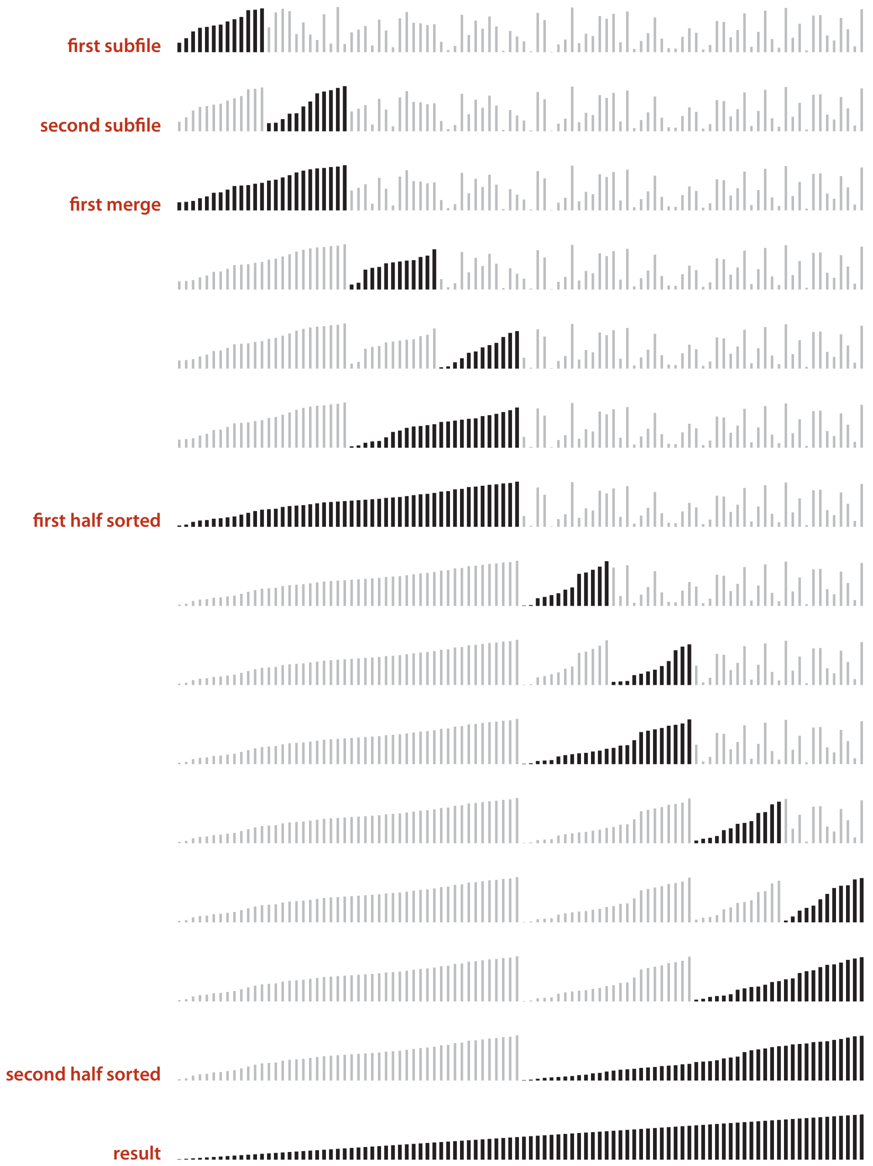


Mergesort: Java implementation

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

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

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

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

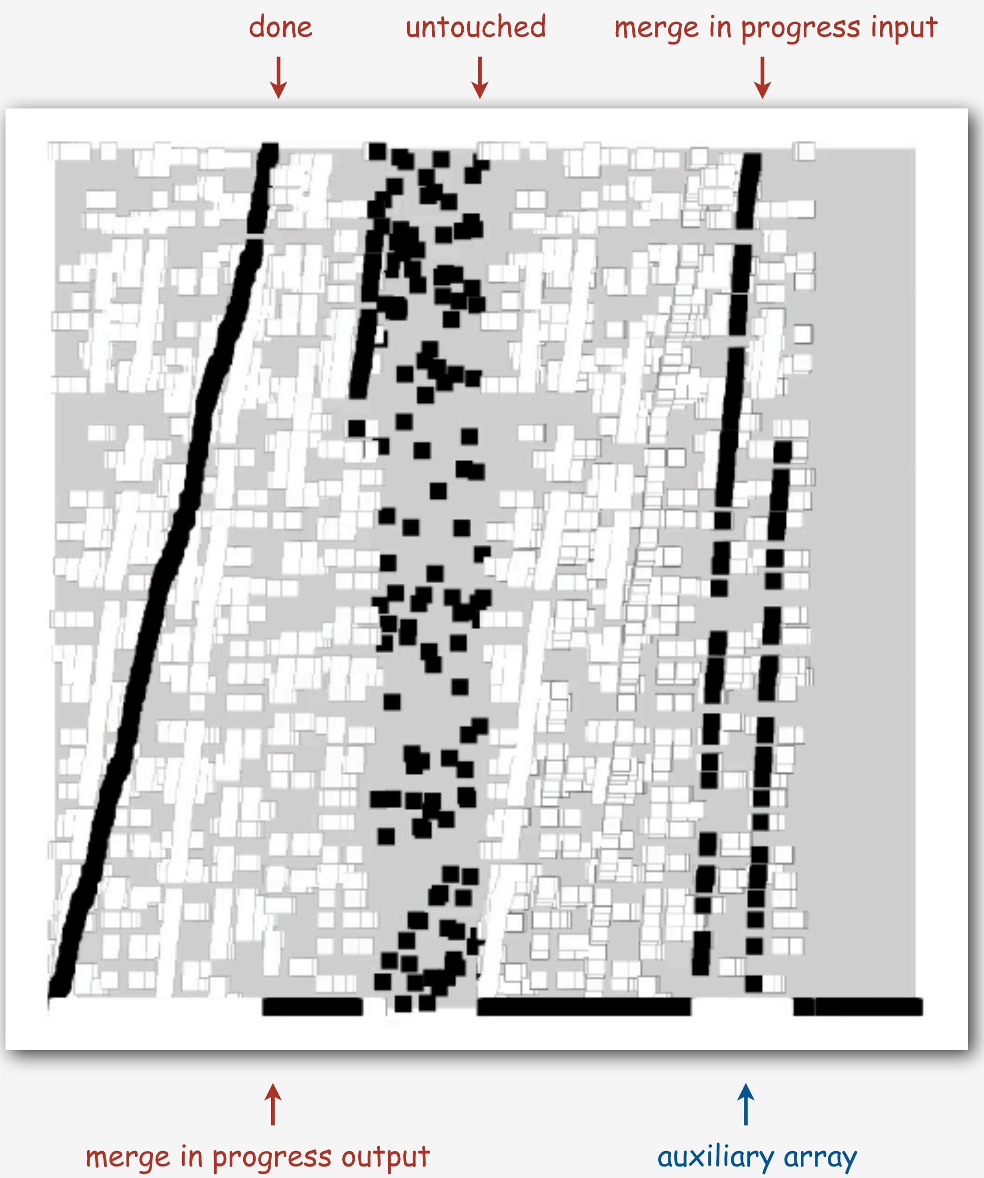


Mergesort visualization



Visual trace of top-down mergesort with cutoff for small subfiles

Mergesort animation



Mergesort animation

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 $\sim N \lg N$ compares to sort any array of size N.

Def. $\mathrm{T}(N)$ = number of compares to mergesort an array of size N.

$$= \underbrace{\mathrm{T}(N/2)}_{\text{left half}} + \underbrace{\mathrm{T}(N/2)}_{\text{right half}} + \underbrace{N}_{\text{merge}}$$

Mergesort recurrence. $\mathrm{T}(N) = 2\,\mathrm{T}(N/2) + N$ for $N > 1$, with $\mathrm{T}(1) = 0$.

- Not quite right for odd N.
- Same recurrence holds for many divide-and-conquer algorithms.

Solution. $\mathrm{T}(N) \sim N \lg N$.

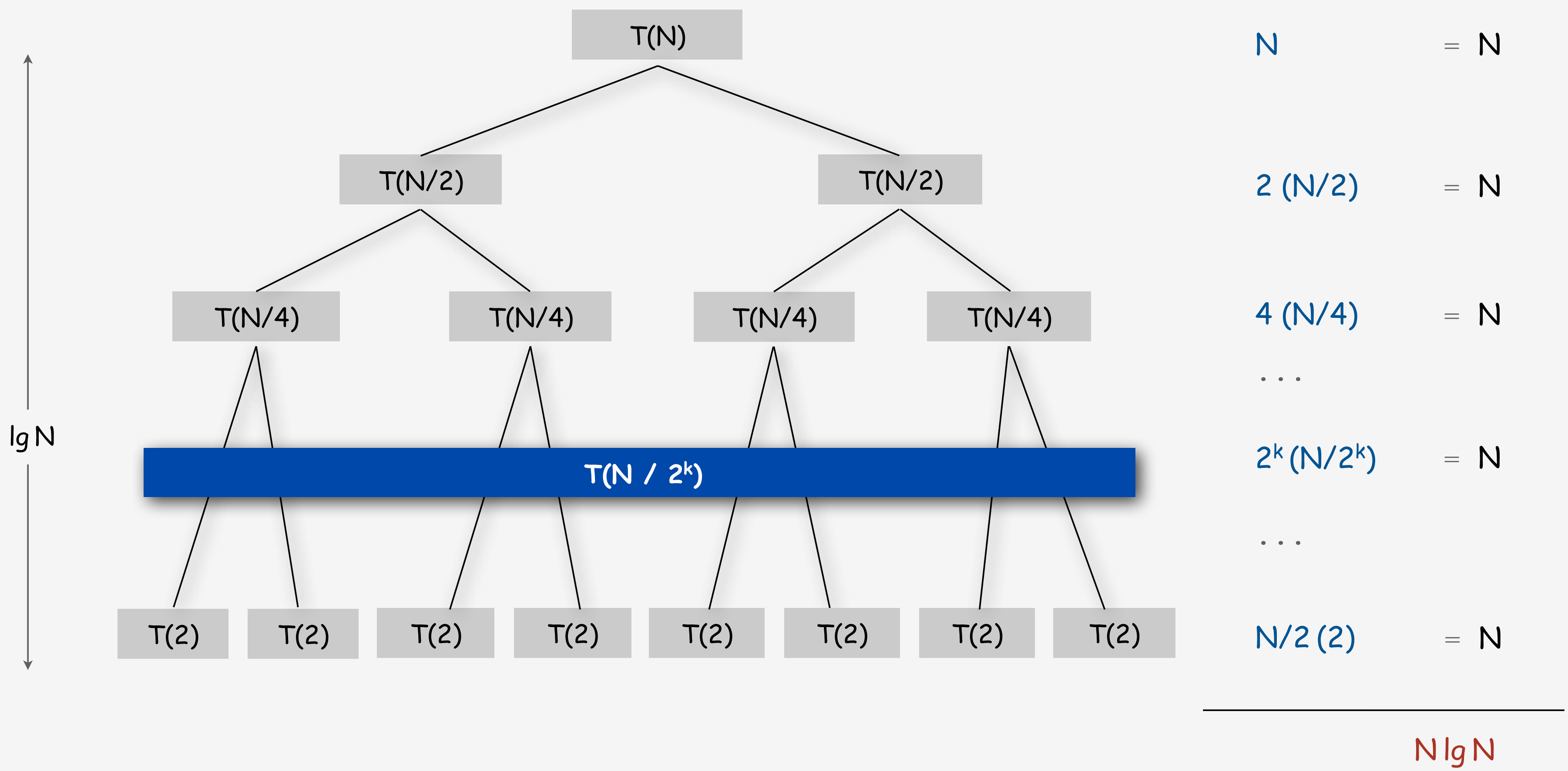
- For simplicity, we'll prove when N is a power of 2.
- True for all N. [see COS 340]

Mergesort recurrence: proof 1

Mergesort recurrence. $T(N) = 2\,T(N/2) + N$ for $N > 1$, with $T(1) = 0$.

Proposition. If N is a power of 2, then $T(N) = N \lg N$.

Pf.



Mergesort recurrence: proof 2

Mergesort recurrence. $\mathrm{T}(N) = 2\,\mathrm{T}(N/2) + N$ for $N > 1$, with $\mathrm{T}(1) = 0$.

Proposition. If N is a power of 2, then $\mathrm{T}(N) = N \lg N$.

Pf.

$T(N) = 2\,T(N/2) + N$	given
$T(N)/N = 2\,T(N/2)/N + 1$	divide both sides by N
$= T(N/2)/(N/2) + 1$	algebra
$= T(N/4)/(N/4) + 1 + 1$	apply to first term
$= T(N/8)/(N/8) + 1 + 1 + 1$	apply to first term again
$\ldots$	
$= T(N/N)/(N/N) + 1 + 1 + \ldots + 1$	stop applying, T(1) = 0
$= \lg N$	

Mergesort recurrence: proof 3

Mergesort recurrence. $\mathrm{T}(N) = 2\,\mathrm{T}(N/2) + N$ for $N > 1$, with $\mathrm{T}(1) = 0$.

Proposition. If N is a power of 2, then $\mathrm{T}(N) = N \lg N$.

Pf. [by induction on N]

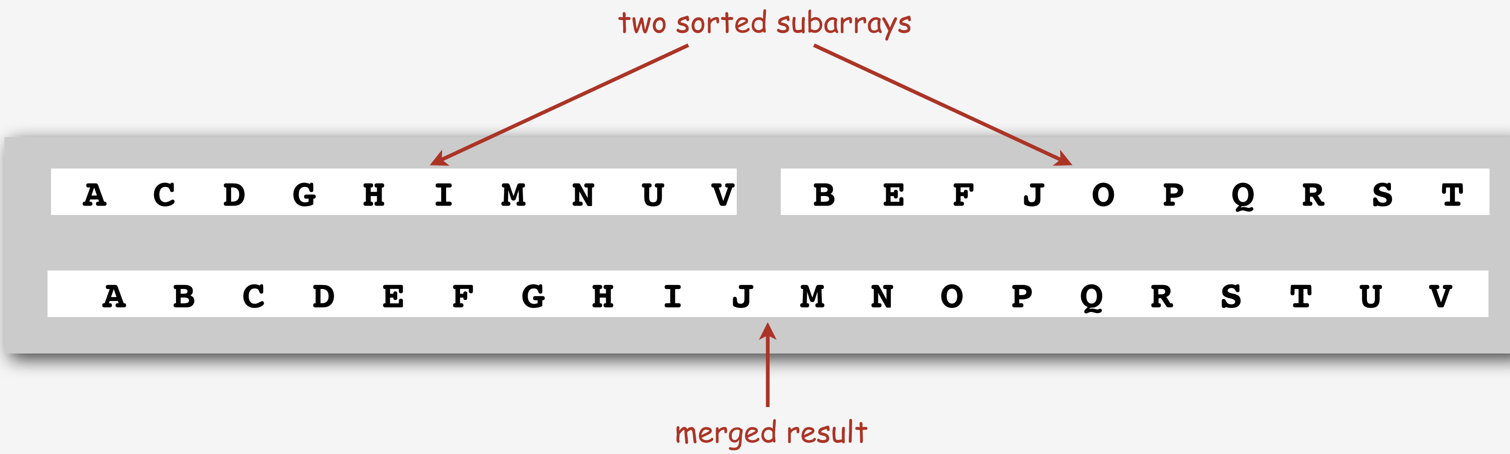
- Base case: $N = 1$.
- Inductive hypothesis: $\mathrm{T}(N) = N \lg N$.
- Goal: show that $\mathrm{T}(2N) = 2N \lg (2N)$.

T(2N) = 2 T(N) + 2N	given
= 2 N lg N + 2 N	inductive hypothesis
= 2 N (lg (2N) - 1) + 2N	algebra
= 2 N lg (2N)	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 ≤ smallest element in second half?
- Helps for nearly ordered lists.

A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V
A	B	C	D	E	F	G	H	I	J	M	N	O	P	Q	R	S	T	U	V

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 `Arrays.sort()`.

- 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, ...

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

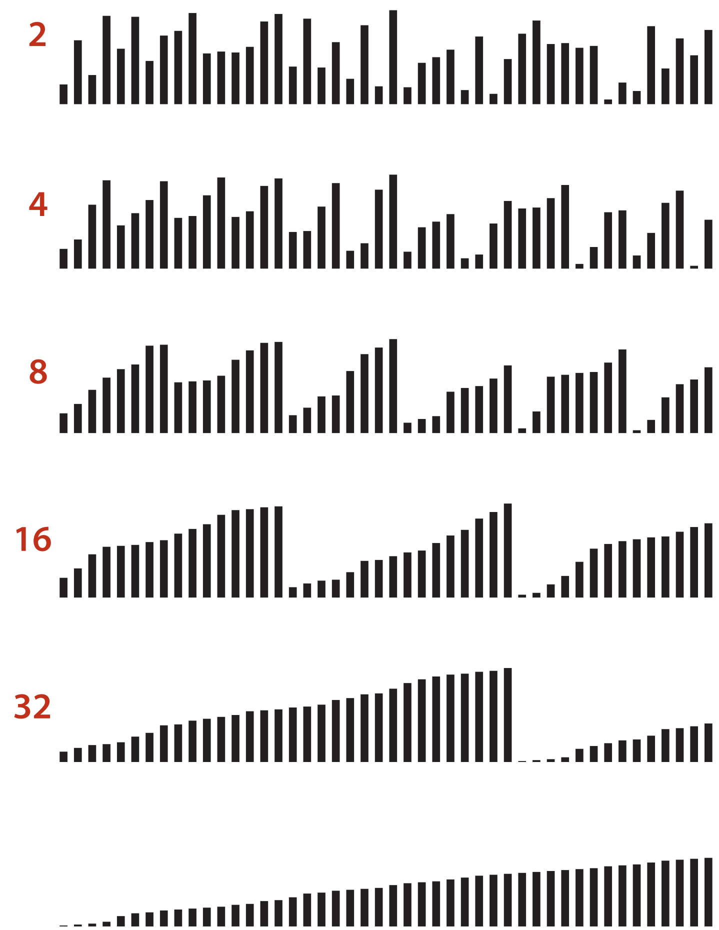
Bottom line. No recursion needed!

Bottom-up mergesort: Java implementation

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

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

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

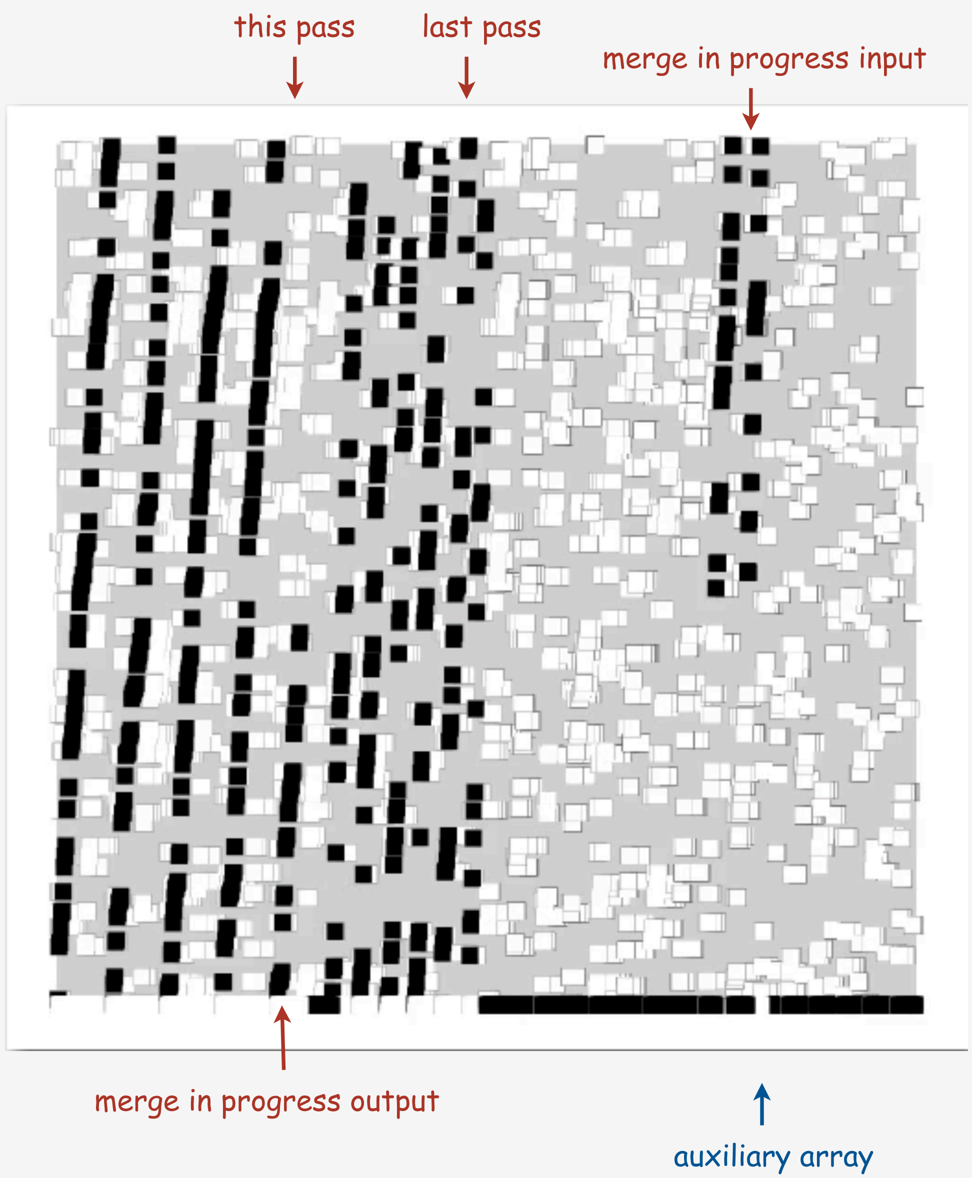
Bottom line. Concise industrial-strength code, if you have the space.

Bottom-up mergesort: visual trace

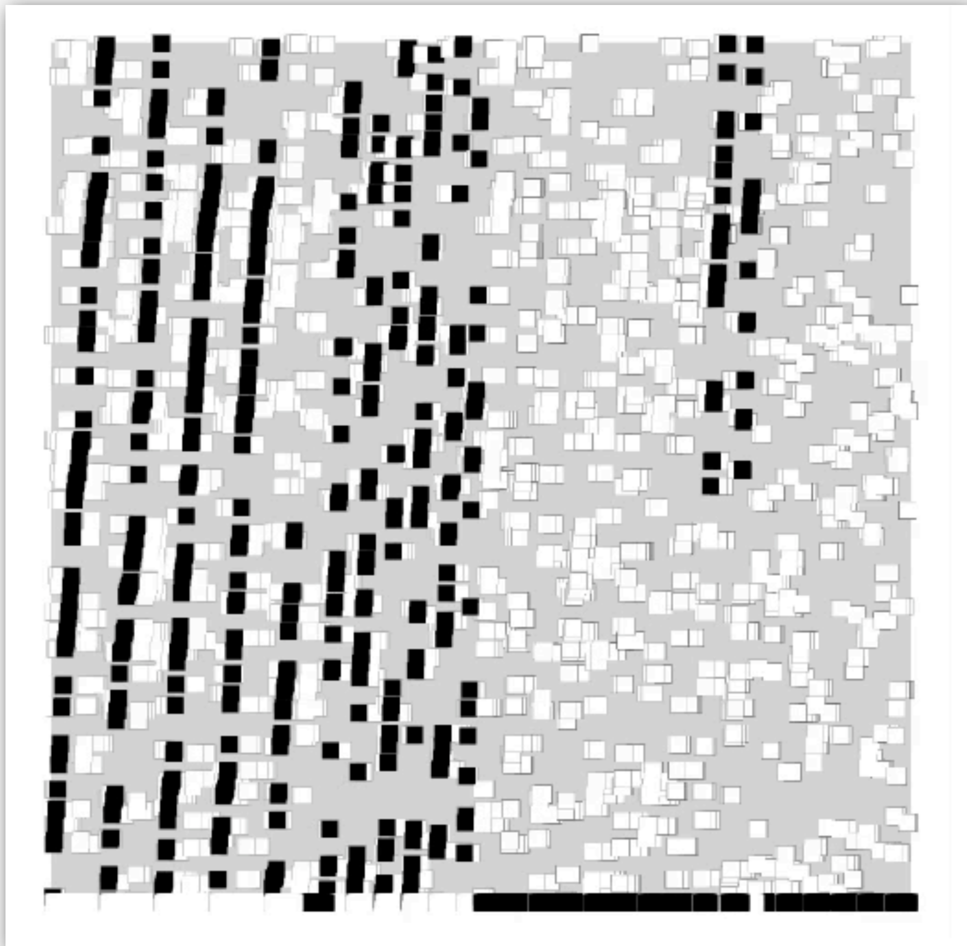


Visual trace of bottom-up mergesort

Botom-up mergesort animation



Botom-up mergesort animation



- mergesort
- bottom-up mergesort
- **sorting complexity**
- comparators

Complexity of sorting

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

Machine model. Focus on fundamental operations.

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 cost guarantee for X.

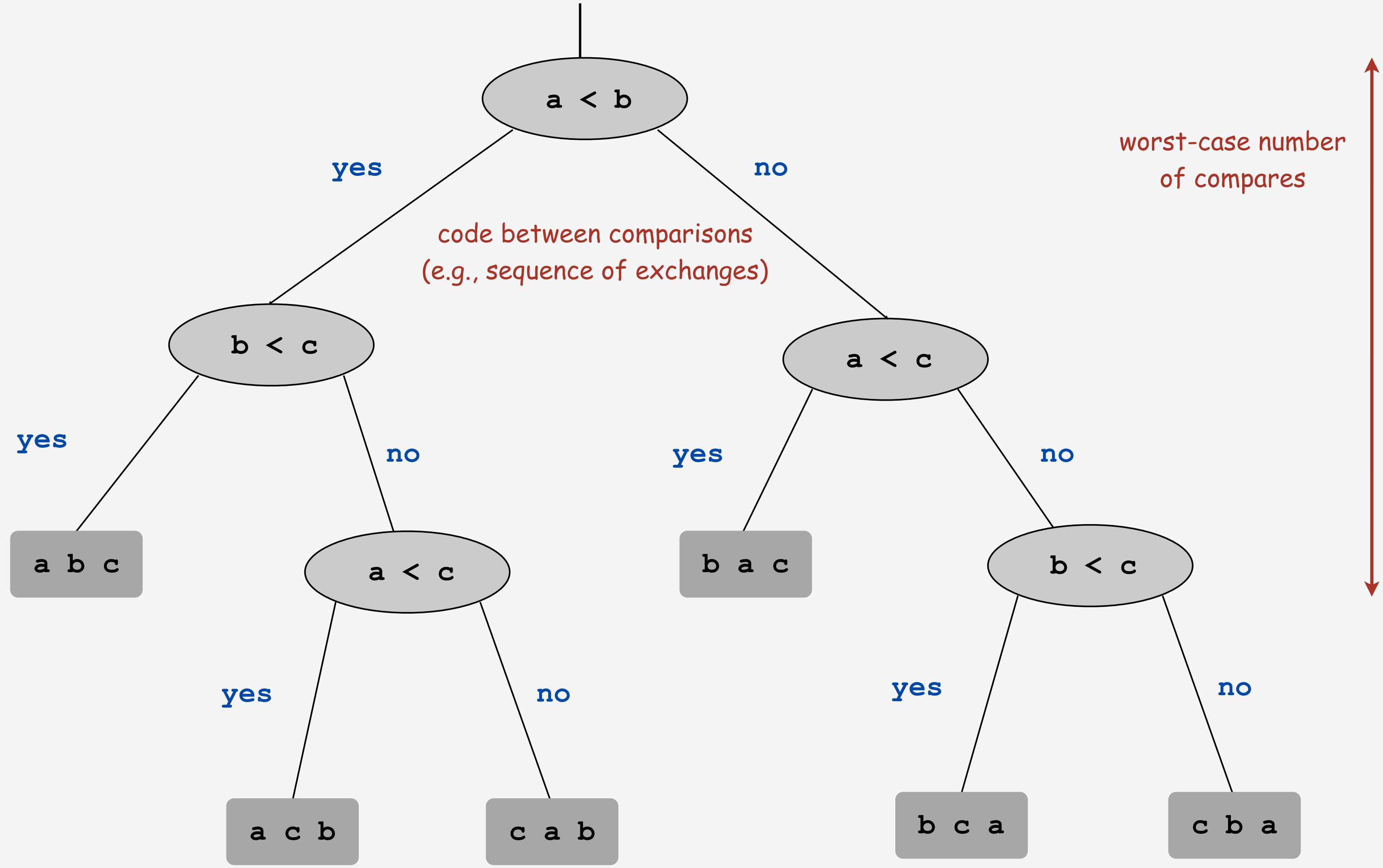
lower bound ~ upper bound

access information only through compares

Example: sorting.

- Machine model = # compares.
- Upper bound = ~ N lg N from mergesort.
- Lower bound = ~ N lg N ?
- Optimal algorithm = mergesort ?

Decision tree (for 3 distinct elements)

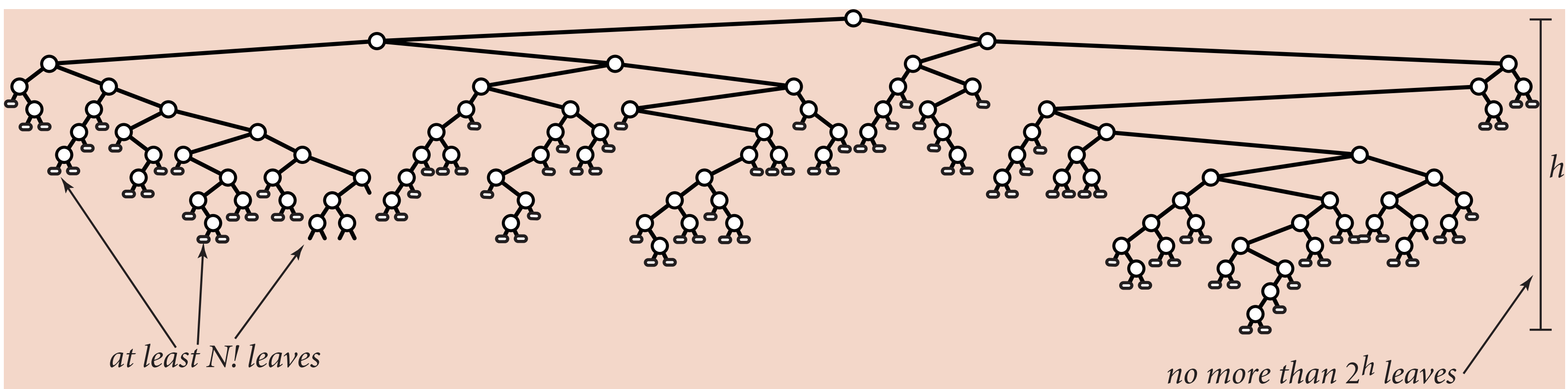


Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must use more than $N \lg N - 1.44\, N$ comparisons in the worst-case.

Pf.

- Assume input 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 more than $N \lg N - 1.44\, N$ comparisons in the worst-case.

Pf.

- Assume input 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.

$$
\begin{aligned}
2^h &\geq N\,! \\
h &\geq \lg N\,! \\
&\geq \lg (N / e)^N \quad \longleftarrow \text{Stirling's formula} \\
&= N \lg N - N \lg e \\
&\geq N \lg N - 1.44\, N
\end{aligned}
$$

Complexity of sorting

Machine model. Focus on fundamental operations.

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 cost guarantee for X.

Example: sorting.

- Machine model = # compares.
- Upper bound = ~ N lg N from mergesort.
- Lower bound = ~ N lg N.
- Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.

Complexity results in context

Other operations? Mergesort optimality is only about number of compares.

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.

Lessons. Use theory as a guide.

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

Complexity results in context (continued)

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

- The key values.
- Their initial arrangement.

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

insertion sort requires O(N) compares on an already sorted array

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

- mergesort
- bottom-up mergesort
- sorting complexity
- **comparators**

Natural order

Comparable interface: sort uses 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

Generalized compare

Comparable interface: sort uses type's natural order.

Problem 1. May want to use a non-natural order.

Problem 2. Desired data type may not come with a "natural" order.

Ex. Sort strings by:

- Natural order. `Now is the time`
- Case insensitive. `is Now the time`
- Spanish. `café cafetero cuarto churro nube ñoño`
- British phone book. `McKinley Mackintosh`

pre-1994 order for digraphs ch and ll and rr

```
String[] a;
...
Arrays.sort(a);
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(a, Collator.getInstance(Locale.SPANISH));
```

`import java.text.Collator;`

Comparators

Solution. Use Java's `Comparator` interface.

```
public interface Comparator<Key>
{
   public int compare(Key v, Key w);
}
```

Remark. The `compare()` method implements a total order like `compareTo()`.

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

- Can add any number of new orders to a data type.
- Can add an order to a library data type with no natural order.

Comparator example

Reverse order. Sort an array of strings in reverse order.

```
public class ReverseOrder implements Comparator<String>
{
   public int compare(String a, String b)
   {
      return b.compareTo(a);
   }
}
```

comparator implementation

```
   ...
   Arrays.sort(a, new ReverseOrder());
   ...
```

client

Sort implementation with comparators

To support comparators in our sort implementations:

- Pass `Comparator` to `sort()` and `less()`.
- Use it in `less()`.

Ex. Insertion sort.

pedantic Java code (see book for simpler version)

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

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

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

Generalized compare

Comparators enable multiple sorts of a single file (by different keys).

Ex. Sort students by name or by section.

```
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```

sort by name

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

then sort by section

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

Generalized compare

Ex. Enable sorting students by name or by section.

```
public class Student
{
   public static final Comparator<Student> BY_NAME = new ByName();
   public static final Comparator<Student> BY_SECT = new BySect();

   private final String name;
   private final int section;
   ...
   private static class ByName implements Comparator<Student>
   {
      public int compare(Student a, Student b)
      {  return a.name.compareTo(b.name);  }
   }

   private static class BySect implements Comparator<Student>
   {
      public int compare(Student a, Student b)
      {  return a.section - b.section;  }
   }
}
```

only use this trick if no danger of overflow

Generalized compare problem

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

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

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	2	A	991-878-4944	308 Blair
Fox	1	A	884-232-5341	11 Dickinson
Furia	3	A	766-093-9873	101 Brown
Gazsi	4	B	665-303-0266	22 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	3	A	232-343-5555	343 Forbes

`Arrays.sort(students, Student.BY_SECT);`

Fox	1	A	884-232-5341	11 Dickinson
Chen	2	A	991-878-4944	308 Blair
Kanaga	3	B	898-122-9643	22 Brown
Andrews	3	A	664-480-0023	097 Little
Furia	3	A	766-093-9873	101 Brown
Rohde	3	A	232-343-5555	343 Forbes
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	665-303-0266	22 Brown

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

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

Stability

Q. Which sorts are stable?

- Selection sort?
- Insertion sort?
- Shellsort?
- Mergesort?

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

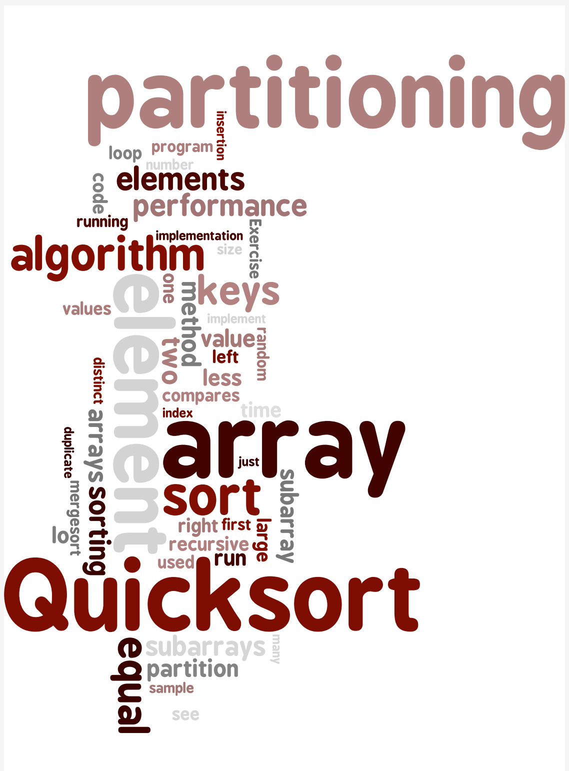
no longer sorted by time

still sorted by time

Stability when sorting on a second key

Open problem. Stable, inplace, N log N, practical sort??

Quicksort



- quicksort
- selection
- duplicate keys
- system sorts

Reference:
Algorithms in Java. 4th Edition, Section 3.2
`http://www.cs.princeton.edu/algs4`

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 20^{th} century in science and engineering.

Mergesort. ← last lecture

- Java sort for objects.
- Perl, Python stable sort.

Quicksort. ← this lecture

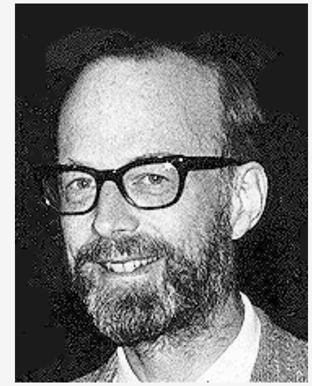
- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.

- quicksort
- selection
- duplicate keys
- system sorts

Quicksort

Basic plan.

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



Sir Charles Antony Richard Hoare
1980 Turing Award

input	Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
shuffle	K	R	A	T	E	E	L	P	U	I	M	Q	C	X	O	S
partition	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
sort left	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
sort right	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

partitioning element

not greater *not less*

Quicksort overview

Quicksort partitioning

Basic plan.

- Scan from left for an item that belongs on the right.
- Scan from right for item item that belongs on the left.
- Exchange.
- Continue until pointers cross.

			v							a[i]								
	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	-1	15	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	7	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	7	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	7	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	7	6	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result			E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

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]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

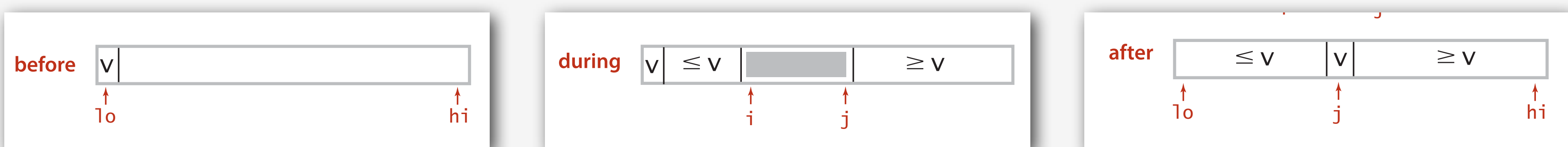
find item on right to swap

check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place



Quicksort: Java implementation

```
public class Quick
{
   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);
  }
}
```

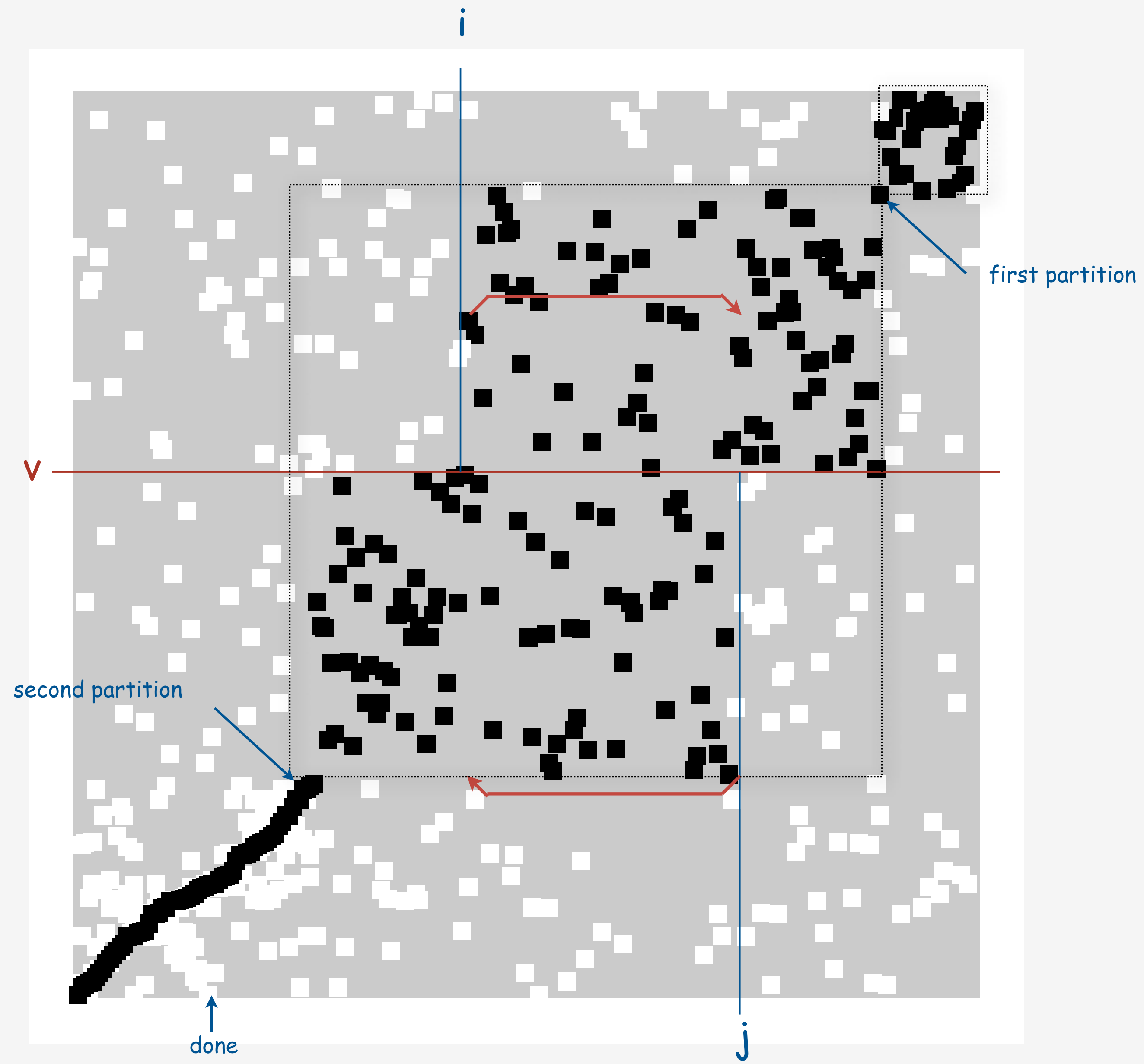
Quicksort trace

	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	E	L	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	2	4	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
	3	4	4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	3		3	A	C	E	E	I	K	S	P	U	T	M	Q	L	X	O	R
	6	12	15	A	C	E	E	I	K	L	P	O	R	M	Q	S	X	U	T
	6	10	11	A	C	E	E	I	K	L	P	O	M	Q	R	S	X	U	T
	6	7	9	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	6		6	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	8	9	9	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	8		8	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	11		11	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	13	13	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	15	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14		14	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

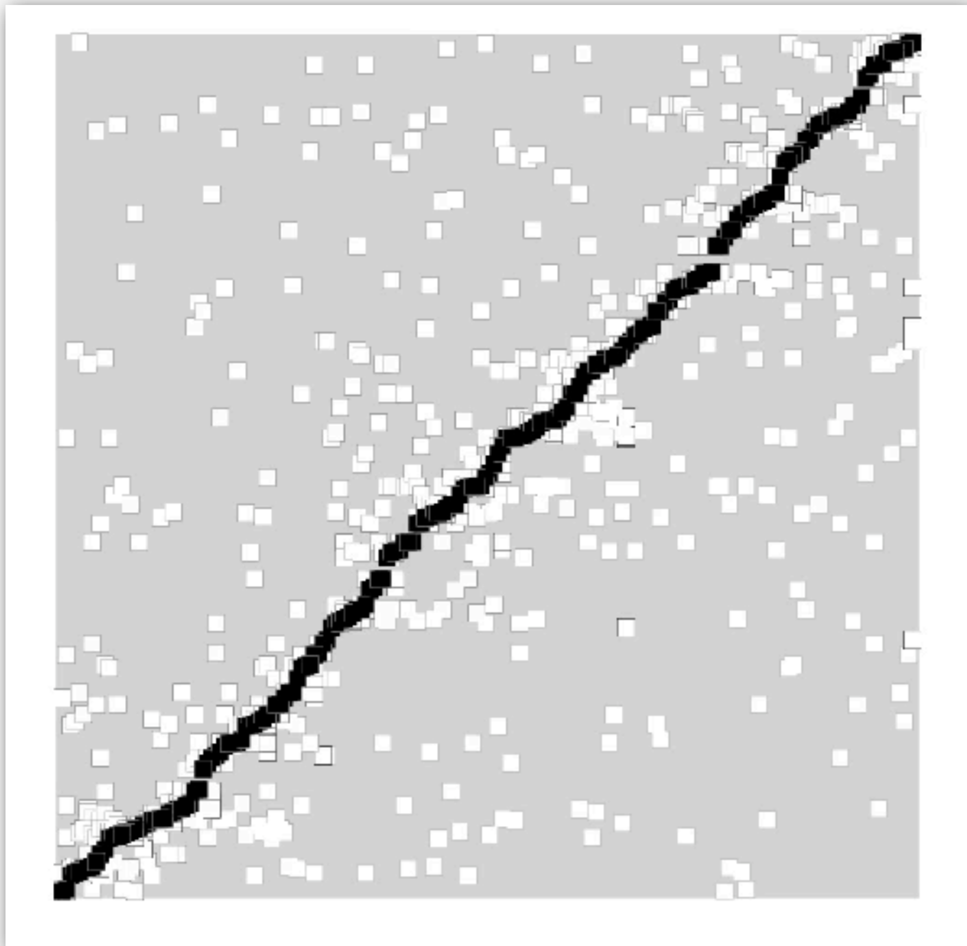
no partition for subarrays of size 1

Quicksort trace (array contents after each partition)

Quicksort animation



Quicksort animation



Quicksort: implementation details

Partitioning in-place. Using a spare 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 `(i == hi)` test is redundant, but the `(j == lo)` test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) best to stop on elements equal to the partitioning element.

Quicksort: 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)			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	0.3 sec	6 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: average-case analysis

Proposition I. The average number of compares C_N to quicksort an array of N elements is ~ 2N ln N (and the number of exchanges is ~ ⅓ N ln N).

Pf. C_N satisfies the recurrence $C_0 = C_1 = 0$ and for N ≥ 2:

$$C_N = \underbrace{(N+1)}_{\text{partitioning}} + \underbrace{\frac{C_0 + C_1 + \ldots + C_{N-1}}{N}}_{\text{left}} + \underbrace{\frac{C_{N-1} + C_{N-2} + \ldots + C_0}{N}}_{\text{right}}$$

partitioning probability

- Multiply both sides by N and collect terms:

$$NC_N = N(N+1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

- Subtract this from the same equation for N-1:

$$NC_N - (N-1)C_N = 2N + 2C_{N-1}$$

- Rearrange terms and divide by N(N+1):

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Quicksort: average-case analysis

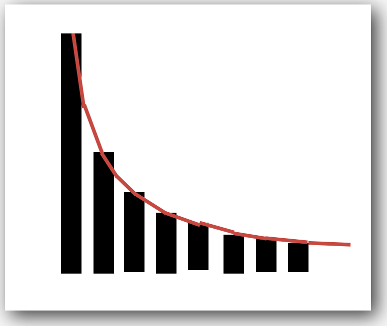
- Repeatedly apply above equation:

$$
\begin{aligned}
\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
&= \frac{2}{1} + \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{N+1}
\end{aligned}
$$

previous equation

- Approximate by an integral:

$$
\begin{aligned}
C_N &\sim 2(N+1)\left(1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{N}\right) \\
&\sim 2(N+1)\int_1^N \frac{1}{x}\,dx
\end{aligned}
$$



- Finally, the desired result:

$$C_N \sim 2(N+1)\ln N \approx 1.39 N \lg N$$

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N-1) + (N-2) + \ldots + 1 \sim N^2 / 2$.
- More likely that your computer is struck by lightning.

Average case. Number of compares is $\sim 1.39 N \lg N$.

- 39% 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 input:

- Is sorted or reverse sorted
- Has many duplicates (even if randomized!) [stay tuned]

Quicksort: practical improvements

Median of sample.

- Best choice of pivot element = median.
- Estimate true median by taking median of sample.

Insertion sort small files.

- Even quicksort has too much overhead for tiny files.
- Can delay insertion sort until end.

Optimize parameters.

~ 12/7 N ln N comparisons

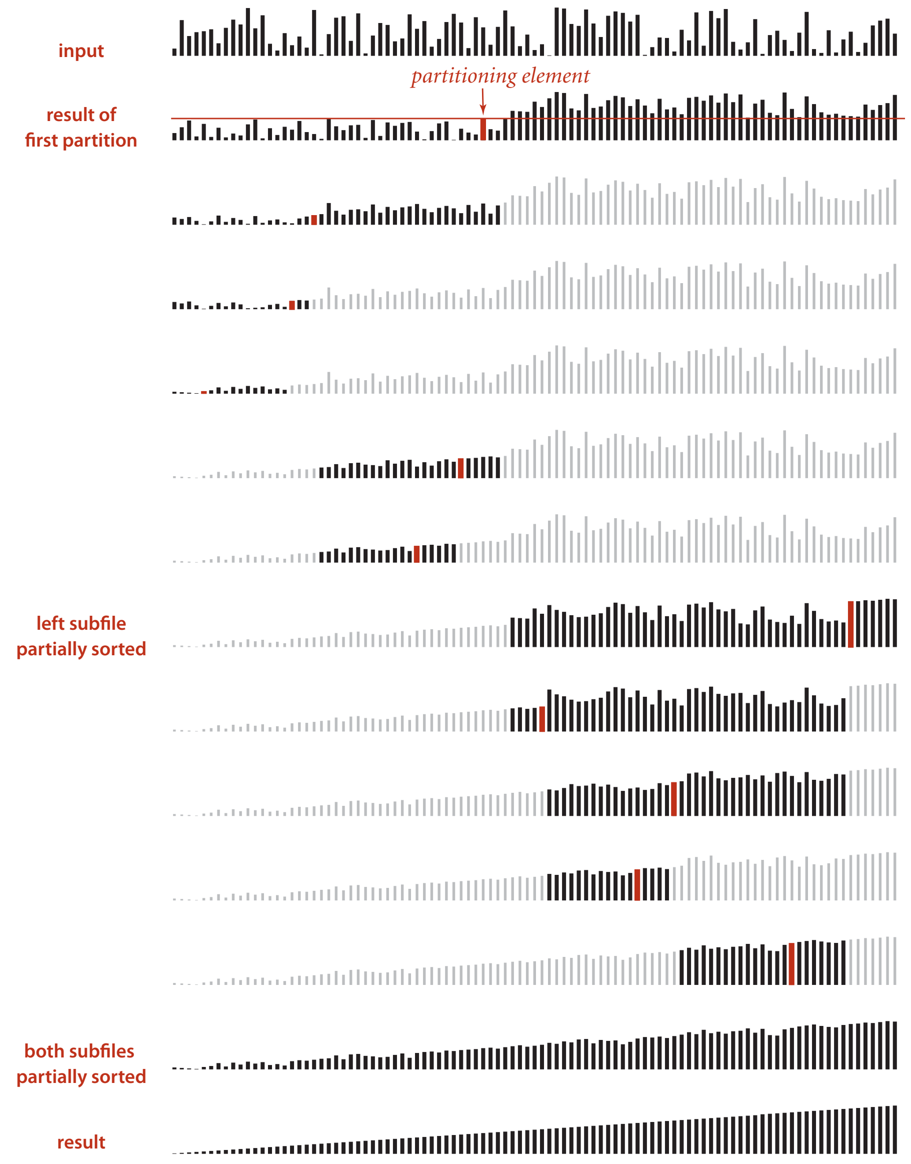
- Median-of-3 random elements.
- Cutoff to insertion sort for ≈ 10 elements.

Non-recursive version.

guarantees O(log N) stack size

- Use explicit stack.
- Always sort smaller half first.

Quicksort with cutoff to insertion sort: visualization



Quicksort with median-of-3 partitioning and cutoff for small subfiles

- quicksort
- **selection**
- duplicate keys
- system sorts

Selection

Goal. Find the k^{th} largest element.

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 O(N log N) upper bound.
- Easy O(N) upper bound for k = 1, 2, 3.
- Easy Ω(N) lower bound.

Which is true?
- Ω(N log N) lower bound? ← is selection as hard as sorting?
- O(N) upper bound? ← is there a linear-time algorithm for all k?

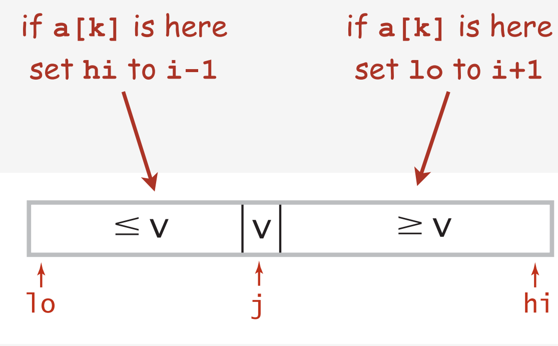
Quick-select

Partition array so that:

- Element `a[i]` is in place.
- No larger element to the left of `i`.
- No smaller element to the right of `i`.

Repeat in one subarray, depending on `i`; finished when `i` equals `k`.

```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
       int i = partition(a, lo, hi);
       if      (i < k) lo = i + 1;
       else if (i > k) hi = i - 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 roughly splits array in half: $N + N/2 + N/4 + \ldots + 1 \sim 2N$ compares.
- Formal analysis similar to quicksort analysis yields:

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

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

Remark. Quick-select might use $\sim N^2/2$ compares, but as with quicksort, the random shuffle provides a probabilistic guarantee.

Theoretical context for selection

Challenge. Design algorithm whose worst-case running time is linear.

Proposition. [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.

Remark. But, algorithm is too complicated to be useful in practice.

Use theory as a guide.

- Still worthwhile to seek practical linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select if you don't need a full sort.

Generic methods

In our `select()` implementation, client needs a cast.

```
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
   a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

hazardous cast required

The compiler also complains.

```
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Q. How to fix?

Generic methods

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

```
public class Quick
{
    public  static <Key extends Comparable<Key>> Key select(Key[] a, int k)
    {  /* as before */  }

    public  static <Key extends Comparable<Key>> void sort(Key[] a)
    {  /* as before */  }

    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi)
    {  /* as before */  }

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

    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j)
    {  Key swap = a[i]; a[i] = a[j]; a[j] = swap;  }

}
```

generic type variable (value inferred from argument a[])

return type matches array type

can declare variables of generic type

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

- quicksort
- selection
- **duplicate keys**
- system sorts

Duplicate keys

Often, purpose of sort is to bring records with duplicate keys together.

- Sort population by age.
- Find collinear points. ← see Assignment 3
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge file.
- 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 ~ 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 implementations
also have this defect

S T O P O N E Q U A L K E Y S

swap swap

Duplicate keys: the problem

Mistake. Put all keys equal to the partitioning element on one side.

Consequence. $\sim N^2 / 2$ 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 keys equal to the partitioning element.

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
```

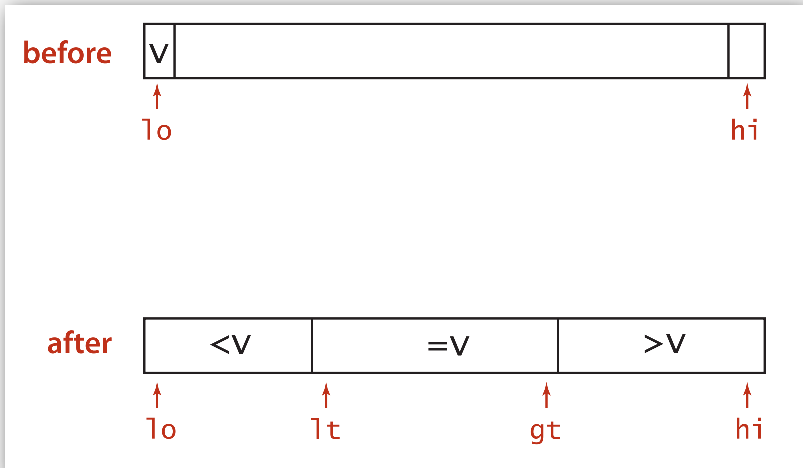
Desirable. Put all keys equal to the partitioning element in place.

```
A A A B 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:

- Elements between `lt` and `gt` equal to partition element `v`.
- No larger elements to left of `lt`.
- No smaller elements to right of `gt`.



Dutch national flag problem. [Edsger Dijkstra]

- Convention 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.

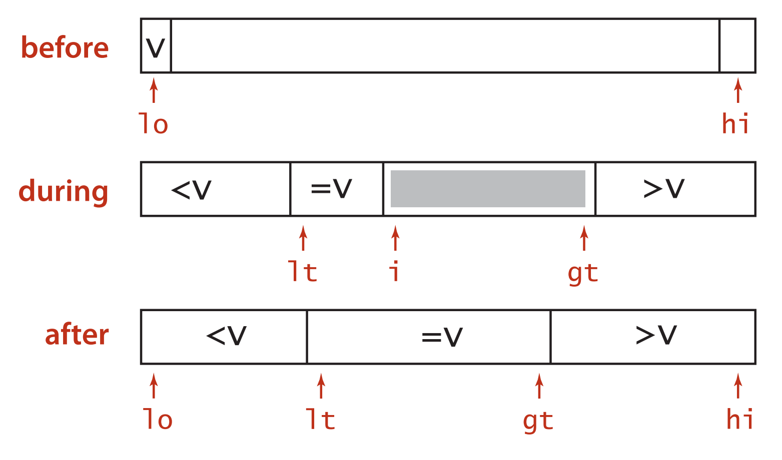
3-way partitioning: Dijkstra's solution

3-way partitioning.

- Let `v` be partitioning element `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`

All the right properties.

- In-place.
- Not much code.
- Small overhead if no equal keys.



3-way partitioning

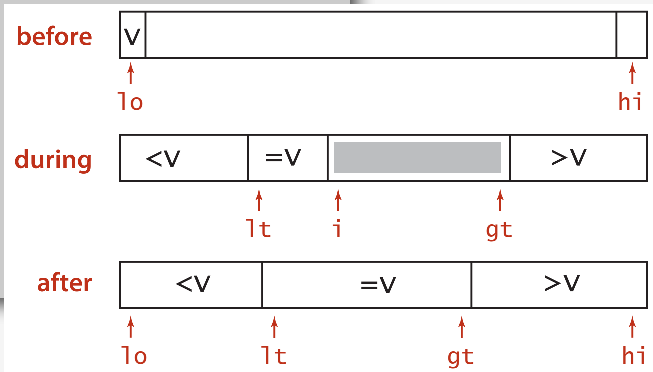
3-way partitioning: trace

lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R	R	W	W	W
2	5	7	B	B	R	R	R	R	B	R	W	W	W	W
2	6	7	B	B	R	R	R	R	B	R	W	W	W	W
3	7	7	B	B	B	R	R	R	R	R	W	W	W	W
3	8	7	B	B	B	R	R	R	R	R	W	W	W	W

3-way partitioning trace (array contents after each loop iteration)

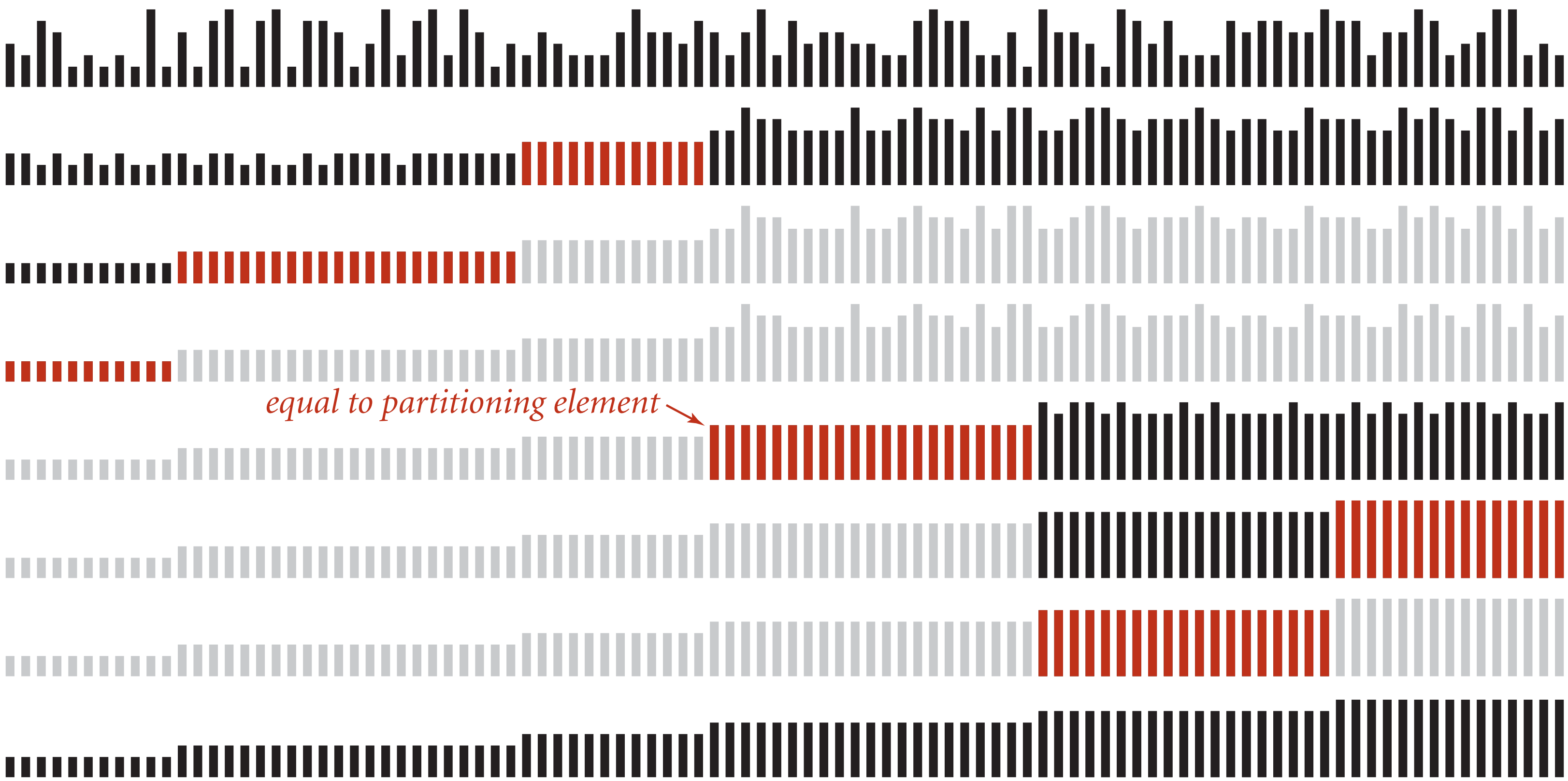
3-way quicksort: Java implementation

```
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 partitioning

3-way quicksort: visual trace



Visual trace of quicksort with 3-way partitioning

Duplicate keys: lower bound

Sorting lower bound. If there are n distinct keys and the i^{th} smallest one occurs x_i times, any compare-based sorting algorithm must use at least

$$- \sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

N lg N when all distinct;
linear when only a constant number of distinct keys

compares in the worst case.

Proposition. [Sedgewick-Bentley, 1997]
Quicksort with 3-way partitioning is entropy-optimal.
Pf. [beyond scope of course]

Bottom line. Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

- selection
- duplicate keys
- comparators
- **system sorts**

Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Load balancing on a parallel computer.

. . .

non-obvious applications

Every system needs (and has) a system sort!

Java uses both mergesort and quicksort.

- `Arrays.sort()` sorts array of `Comparable` or any primitive type.
- Uses quicksort for primitive types; mergesort for objects.

```
import java.util.Arrays;

public class StringSort
{
   public static void main(String[] args)
   {
      String[] a = StdIn.readAll().split("\\s+");
      Arrays.sort(a);
      for (int i = 0; i < N; i++)
         StdOut.println(a[i]);
   }
}
```

Q. Why use different algorithms, depending on type?

Java system sort for primitive types

Engineering a sort function. [Bentley-McIlroy, 1993]

- Original motivation: improve `qsort()`.
- Basic algorithm = 3-way quicksort with cutoff to insertion sort.
- Partition on Tukey's ninther: median of the medians of 3 samples, each of 3 elements.

approximate median-of-9

nine evenly spaced elements	R	L	A	P	M	C	G	A	X	Z	K	R	B	R	J	J	E
groups of 3	R	A	M		G	X	K		B	J	E						
medians	M	K	E														
ninther	K																

Why use Tukey's ninther?

- Better partitioning than sampling.
- Less costly than random.

Achilles heel in Bentley-McIlroy implementation (Java system sort)

Based on all this research, Java's system sort is solid, right?

A killer input.

more disastrous consequences in C

- Blows function call stack in Java and crashes program.
- Would take quadratic time if it didn't crash first.

```
% more 250000.txt
0
218750
222662
11
166672
247070
83339
...
```

```
% java IntegerSort < 250000.txt
Exception in thread "main"
java.lang.StackOverflowError
   at java.util.Arrays.sort1(Arrays.java:562)
   at java.util.Arrays.sort1(Arrays.java:606)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   at java.util.Arrays.sort1(Arrays.java:608)
   ...
```

250,000 integers
between 0 and 250,000

Java's sorting library crashes, even if
you give it as much stack space as Windows allows

Achilles heel in Bentley-McIlroy implementation (Java system sort)

McIlroy's devious idea. [A Killer Adversary for Quicksort]

- Construct malicious input while running system quicksort, in response to elements compared.
- If `v` is partitioning element, commit to `(v < a[i])` and `(v < a[j])`, but don't commit to `(a[i] < a[j])` or `(a[j] > a[i])` until `a[i]` and `a[j]` are compared.

Consequences.

- Confirms theoretical possibility.
- Algorithmic complexity attack: you enter linear amount of data; server performs quadratic amount of work.

Remark. Attack is not effective if array is shuffled before sort.

Q. Why do you think system sort is deterministic?

System sort: Which algorithm to use?

Many sorting algorithms to choose from:

Internal sorts.

- Insertion sort, selection sort, bubblesort, shaker sort.
- Quicksort, mergesort, heapsort, samplesort, shellsort.
- Solitaire sort, red-black sort, splaysort, Dobosiewicz sort, psort, ...

External sorts. Poly-phase mergesort, cascade-merge, oscillating sort.

Radix sorts. Distribution, MSD, LSD, 3-way radix quicksort.

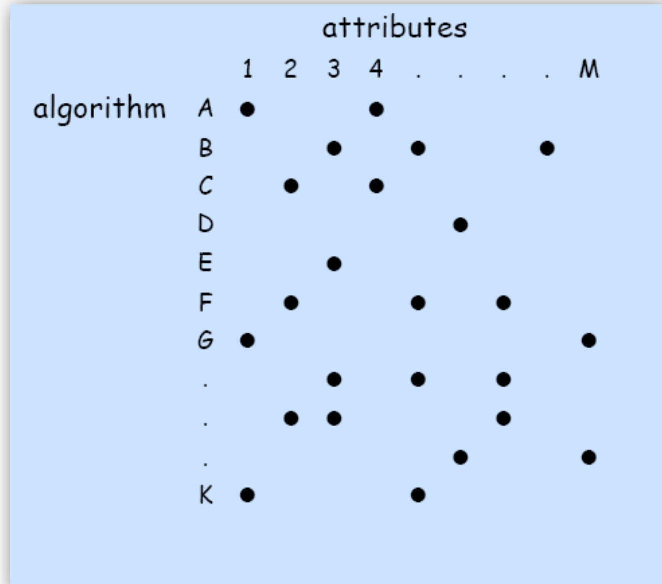
Parallel sorts.

- Bitonic sort, Batcher even-odd sort.
- Smooth sort, cube sort, column sort.
- GPUsort.

System sort: Which algorithm to use?

Applications have diverse attributes.

- Stable?
- Multiple keys?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small records?
- Is your file randomly ordered?
- Need guaranteed performance?



many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.

Cannot cover all combinations of attributes.

Q. Is the system sort good enough?

A. Usually.

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
quick	×		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	×		$N^2 / 2$	$2 N \ln N$	N	improves quicksort in presence of duplicate keys
merge		×	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
???	×	×	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

Which sorting algorithm?

data	data	data	data	data	data	data	data
type	fifo	find	find	exch	hash	exch	exch
hash	hash	hash	hash	fifo	heap	fifo	fifo
heap	heap	heap	heap	find	type	heap	find
sort	exch	leaf	leaf	hash	link	find	hash
link	less	link	link	heap	list	link	heap
list	left	list	list	leaf	push	hash	leaf
push	leaf	push	push	left	sort	left	left
find	find	root	root	less	find	less	less
root	lifo	sort	sort	lifo	leaf	path	lifo
leaf	push	tree	tree	link	root	leaf	link
tree	tree	type	type	list	tree	lifo	list
null	null	exch	null	null	left	next	next
path	path	fifo	path	path	node	root	node
node	node	left	node	node	null	list	null
left	list	less	left	push	path	push	path
less	link	lifo	less	tree	exch	null	push
exch	sort	next	exch	type	less	swap	root
sink	sink	node	sink	sink	sink	node	sink
swim	swim	null	swim	swim	swim	swim	sort
next	next	path	next	next	fifo	sort	swap
swap	swap	sink	swap	swap	lifo	type	swim
fifo	type	swap	fifo	sort	next	sink	tree
lifo	root	swim	lifo	root	swap	tree	type
original	?	?	?	?	?	?	sorted

Priority Queues

- API
- elementary implementations
- binary heaps
- heapsort
- event-based simulation

Remove by (largest) value.

```
public class MaxPQ<Key extends Comparable<Key>>
```

	`MaxPQ()`	*create a priority queue*
	`MaxPQ(maxN)`	*create a priority queue of initial capacity maxN*
`void`	`insert(Key v)`	*insert a key into the priority queue*
`Key`	`max()`	*return the largest key*
`Key`	`delMax()`	*return and remove the largest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`int`	`size()`	*number of entries in the priority queue*

API for a generic priority queue

stack	last in, first out
queue	first in, first out
priority queue	largest out

operation	*argument*	*return value*
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

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

Problem. Find the largest M in a stream of N elements.

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

Constraint. Not enough memory to store N elements.

Solution. Use a min-oriented priority queue.

```
MinPQ<String> pq = new MinPQ<String>();

while(!StdIn.isEmpty())
{
   String s = StdIn.readString();
   pq.insert(s);
   if (pq.size() > M)
      pq.delMin();
}

while (!pq.isEmpty())
   System.out.println(pq.delMin());
```

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

cost of finding the largest M in a stream of N items

- API
- **elementary implementations**
- binary heaps
- heapsort
- event-based simulation

Priority queue: unordered and ordered array implementation

operation	argument	return value	size	contents (unordered)							contents (ordered)						
insert	P		1	P							P						
insert	Q		2	P	Q						P	Q					
insert	E		3	P	Q	E					E	P	Q				
remove max		Q	2	P	E						E	P					
insert	X		3	P	E	X					E	P	X				
insert	A		4	P	E	X	A				A	E	P	X			
insert	M		5	P	E	X	A	M			A	E	M	P	X		
remove max		X	4	P	E	M	A				A	E	M	P			
insert	P		5	P	E	M	A	P			A	E	M	P	P		
insert	L		6	P	E	M	A	P	L		A	E	L	M	P	P	
insert	E		7	P	E	M	A	P	L	E	A	E	E	L	M	P	P
remove max		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 UnorderedPQ(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()` as for sorting

Priority queue elementary implementations

Challenge. Implement all operations efficiently.

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

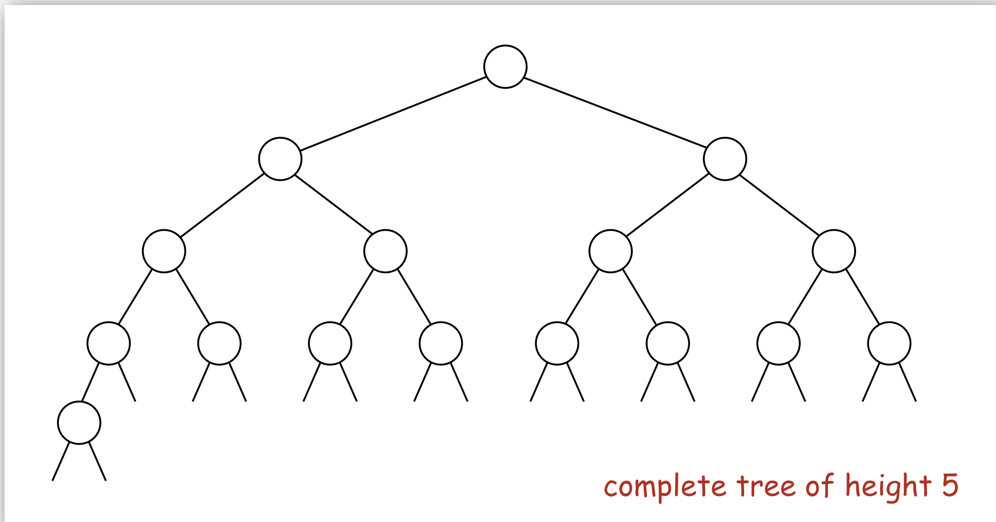
order-of-growth running time for PQ with N items

- API
- elementary implementations
- **binary heaps**
- heapsort
- event-based simulation

Binary tree

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

Complete tree. Perfectly balanced, except for bottom level.



$N = 16$
$\lfloor \lg N \rfloor = 4$
height = 5

Property. Height of complete tree with N nodes is $1 + \lfloor \lg N \rfloor$.

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

Binary heap

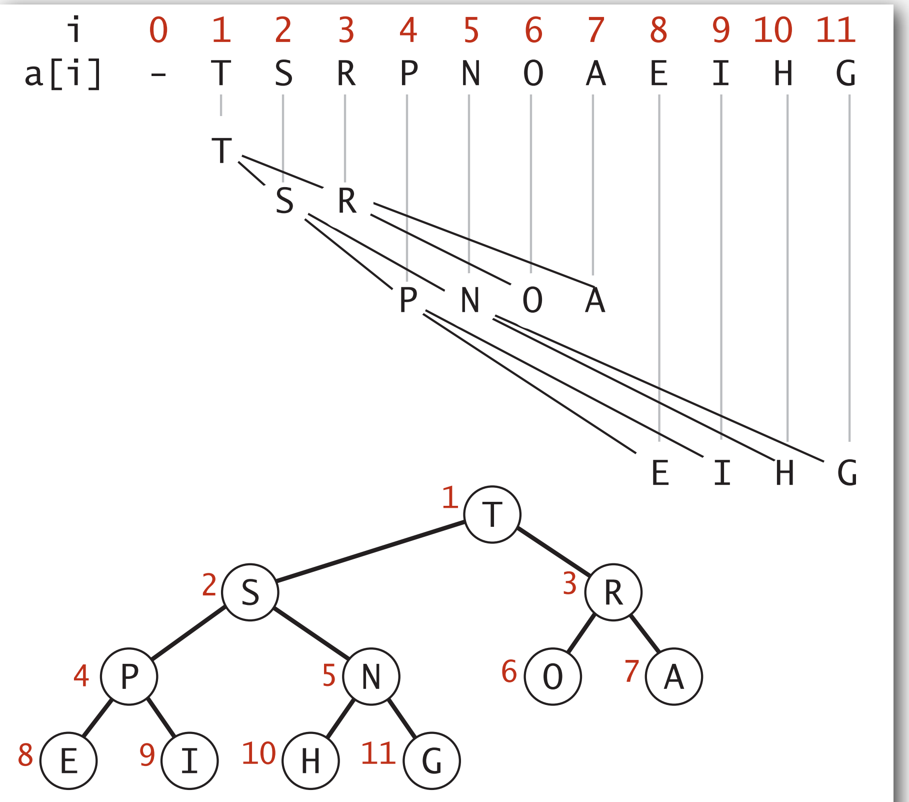
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.

Array representation.

- Take nodes in level order.
- No explicit links needed!



Heap representations

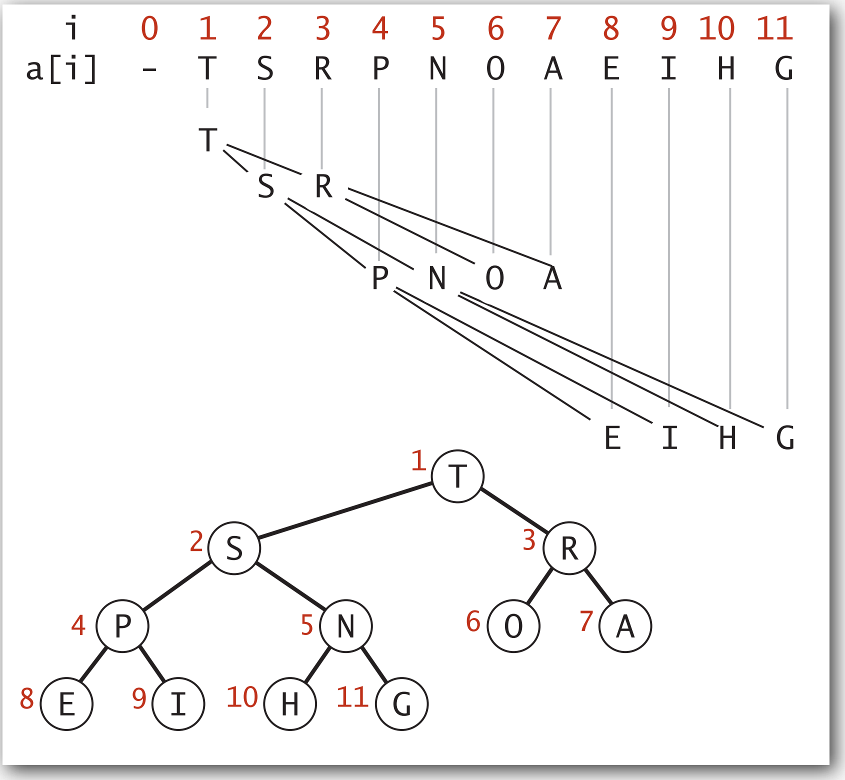
Binary heap properties

Property A. Largest key is at root.

indices start at 1

Property B. 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

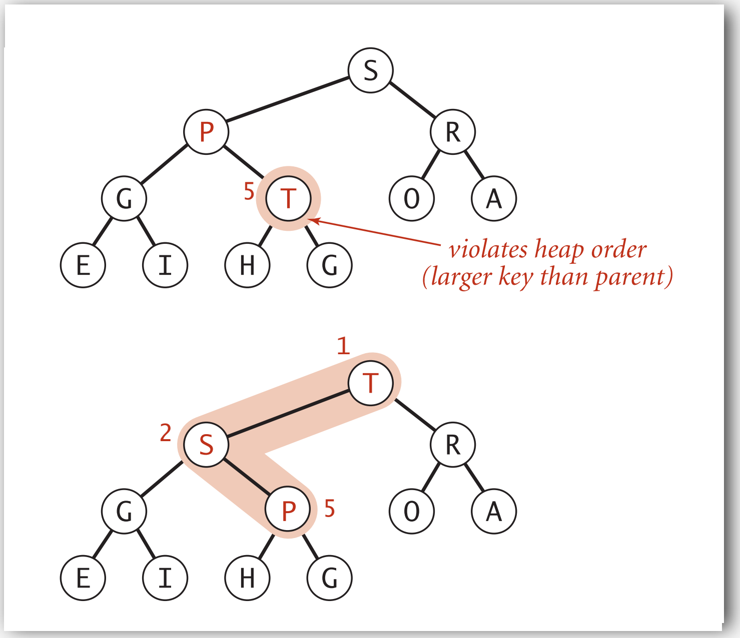
Scenario. Exactly one node has a larger key than its parent.

To eliminate the violation:

- Exchange with its 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

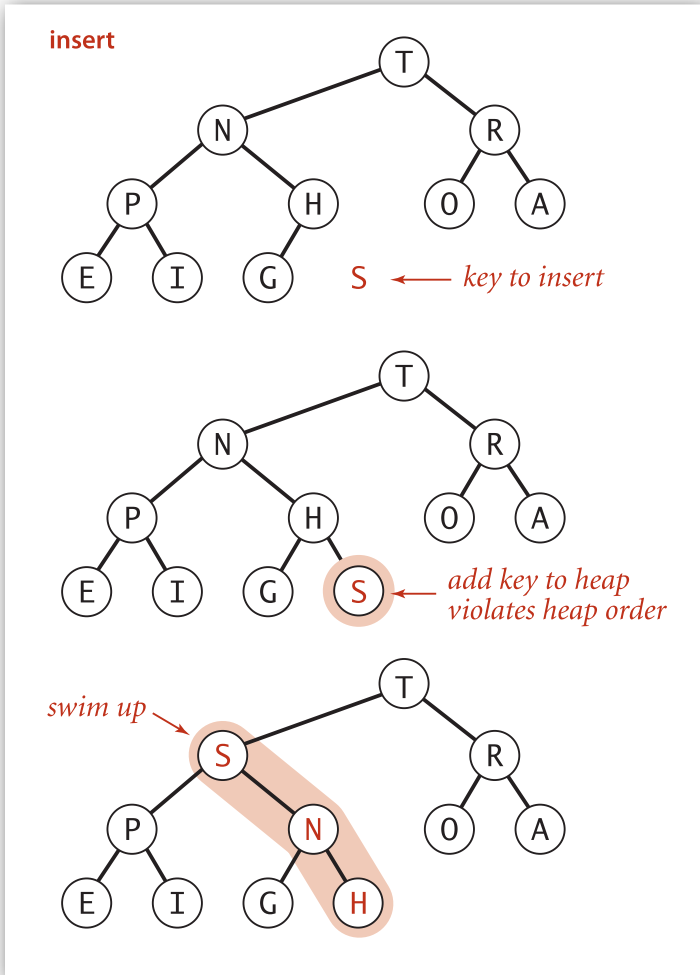


Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then promote.

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



Demotion in a heap

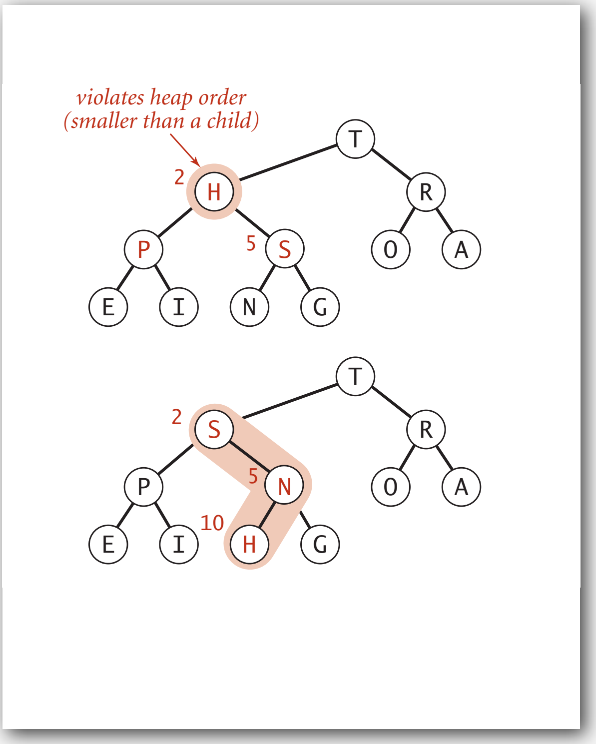
Scenario. Exactly one node has a smaller key than does a child.

To eliminate the violation:

- Exchange with larger child.
- Repeat until heap order restored.

```
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



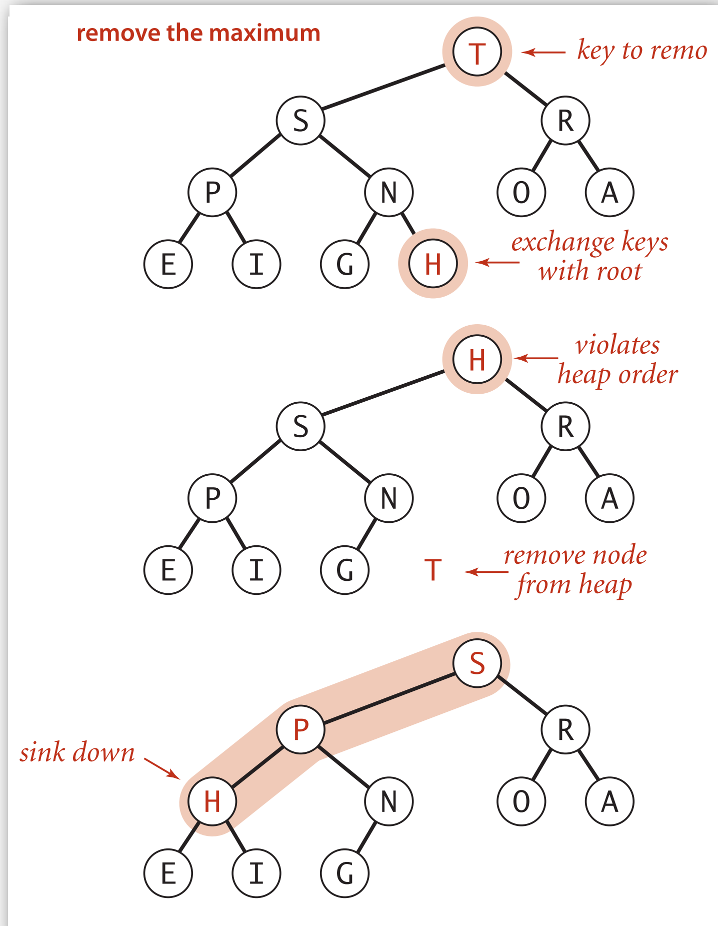
Power struggle. Better subordinate promoted.

Delete the maximum in a heap

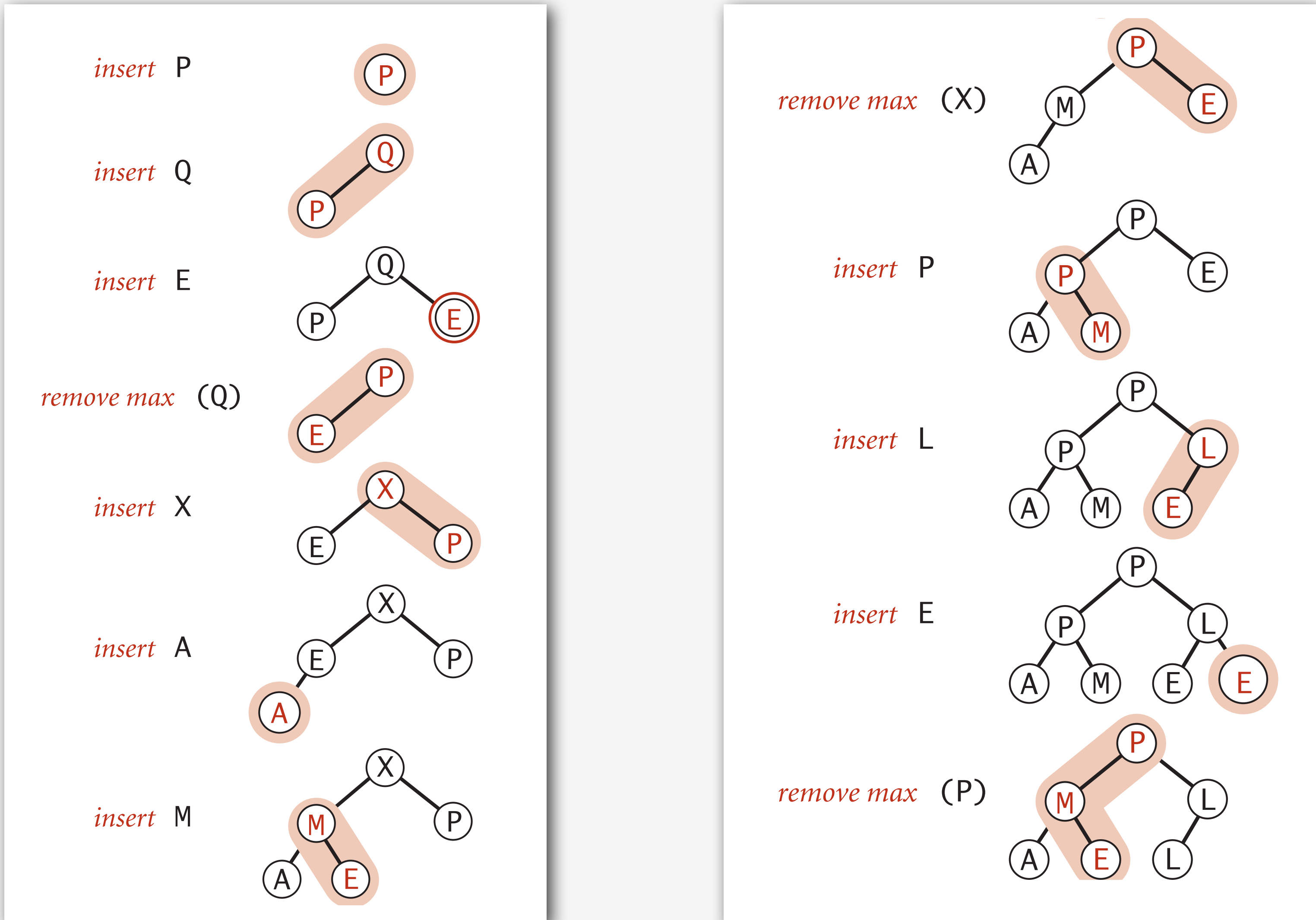
Delete max. Exchange root with node at end, then demote.

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

prevent loitering



Heap operations



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

Binary heap considerations

Minimum-oriented priority queue.

- Replace `less()` with `greater()`.
- Implement `greater()`.

Dynamic array resizing.

- Add no-arg constructor.
- Apply repeated doubling and shrinking. ← leads to O(log N) amortized time per op

Immutability of keys.

- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Other operations.

- Remove an arbitrary item.
- Change the priority of an item.

easy to implement with `sink()` and `swim()` [stay tuned]

Priority queues implementation cost summary

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	log N	log N	1

order-of-growth running time for PQ with N items

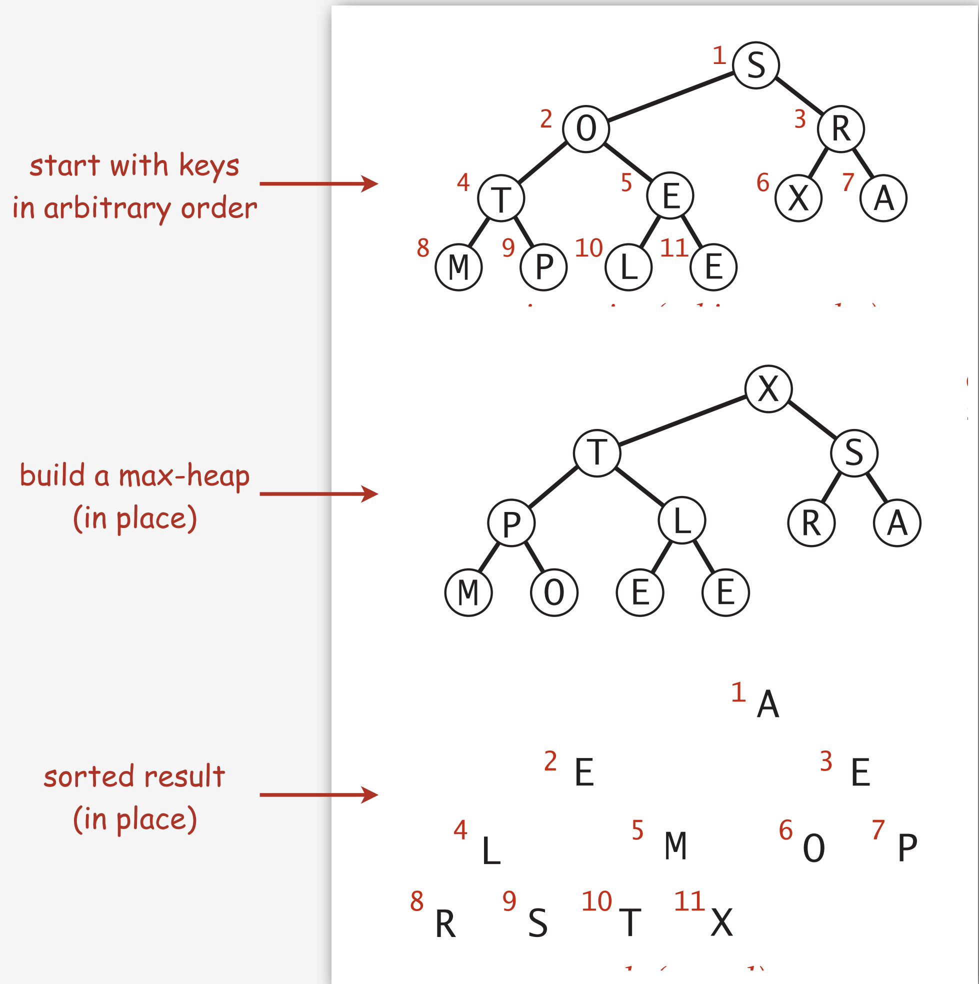
Hopeless challenge. Make all operations constant time.

Q. Why hopeless?

- API
- elementary implementations
- binary heaps
- **heapsort**
- event-based simulation

Basic plan for in-place sort.

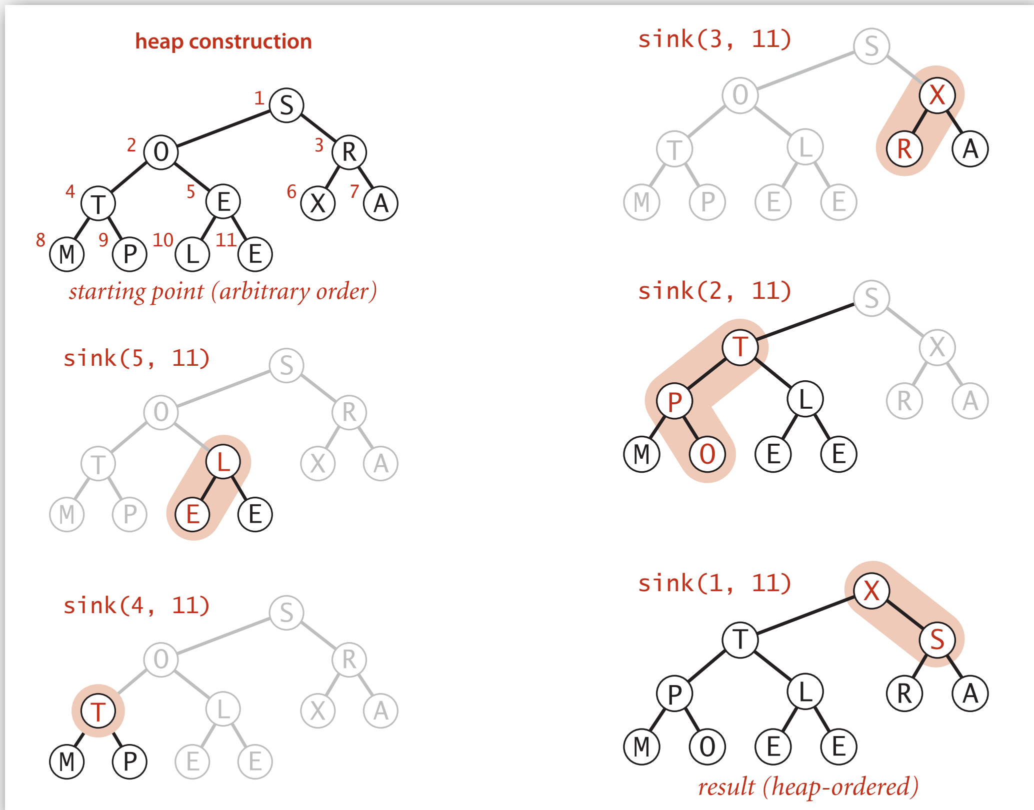
- Create max-heap with all N keys.
- Repeatedly remove the maximum key.



Heapsort

First pass. Build heap using bottom-up method.

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

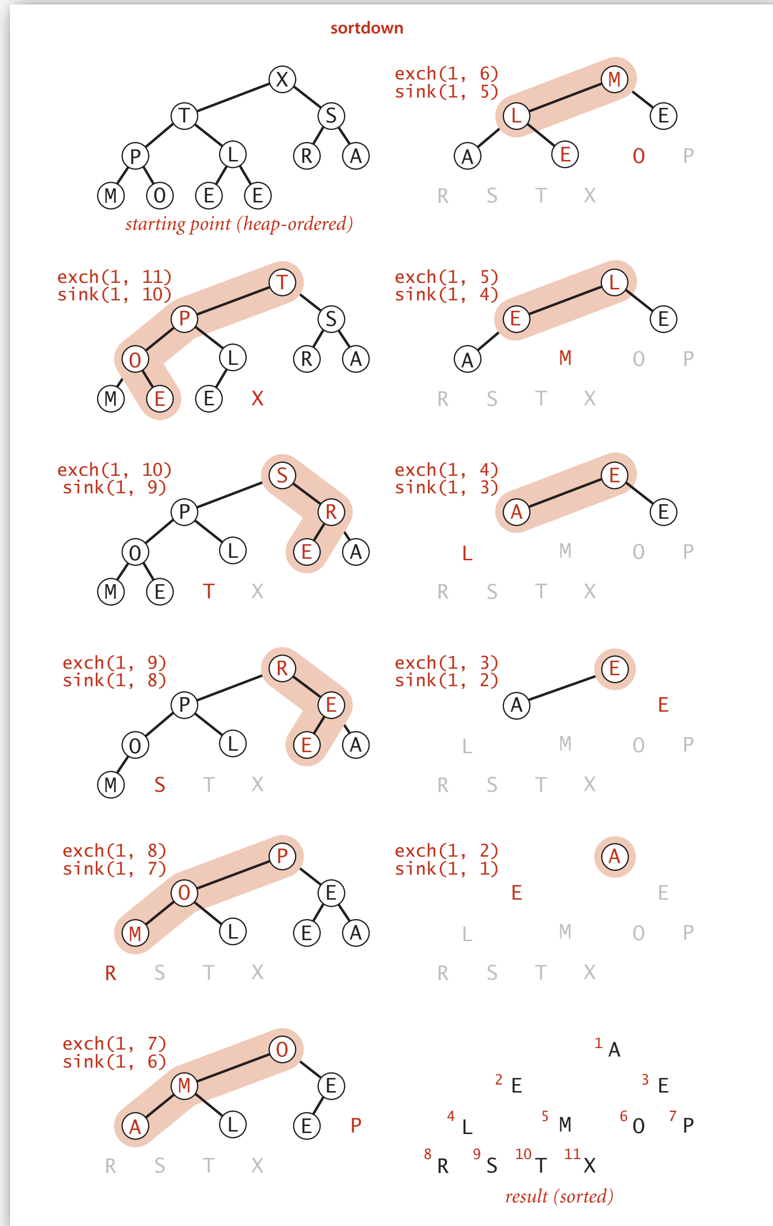


Heapsort

Second pass. Sort.

- 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 use 1-based indexing

Heapsort: trace

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			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
heap-ordered			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
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

Property D. At most 2 N lg N compares.

Significance. Sort in N log N worst-case without using extra memory.

- 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

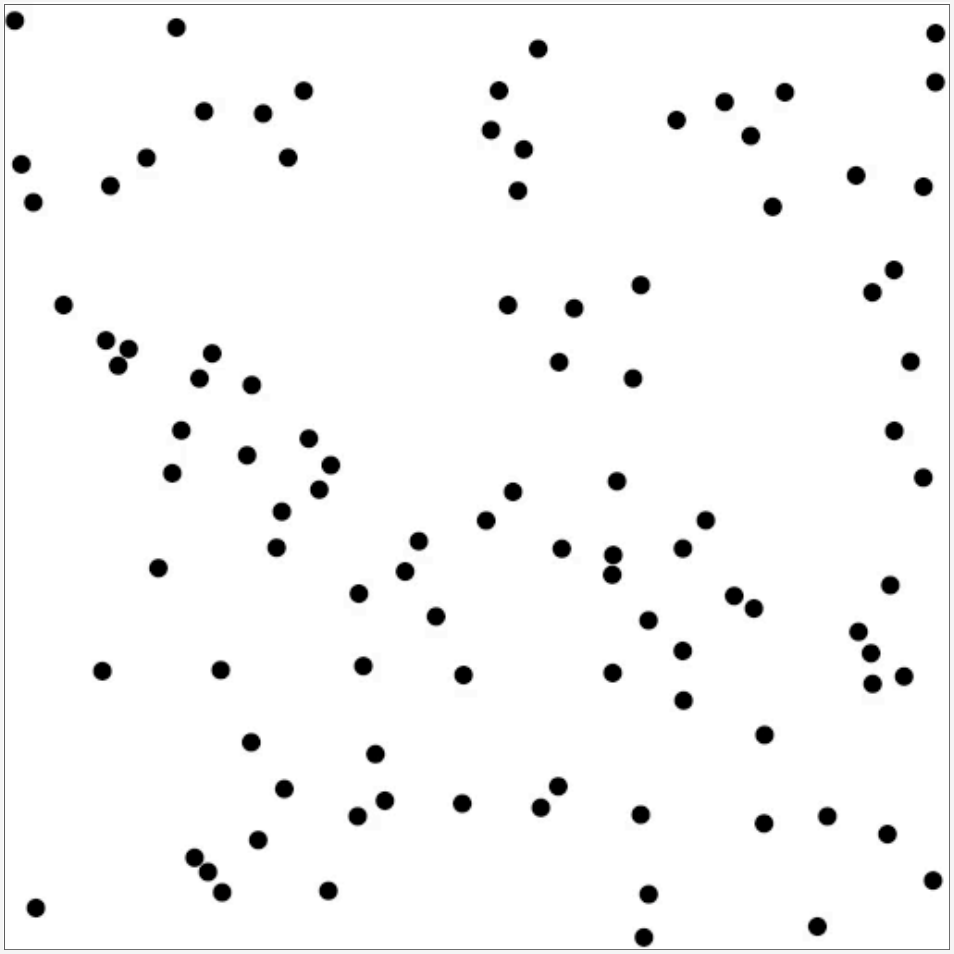
Sorting algorithms: summary

	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		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	$2 N \ln N$	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		$2 N \lg N$	$2 N \lg N$	$N \lg N$	$N \log N$ guarantee, in-place
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

- API
- elementary implementations
- binary heaps
- heapsort
- **event-based 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 are exerted.

temperature, pressure, diffusion constant

motion of individual atoms and molecules

Significance. Relates macroscopic observables to microscopic dynamics.

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

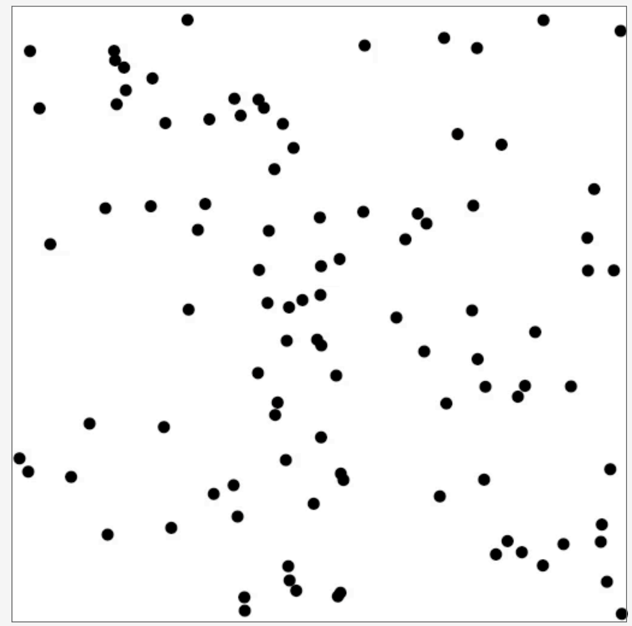
Warmup: bouncing balls

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);   }
}
```

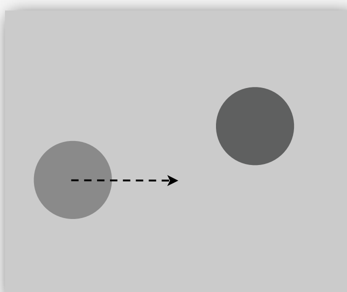
check for collision with walls

Missing. Check for balls colliding with each other.

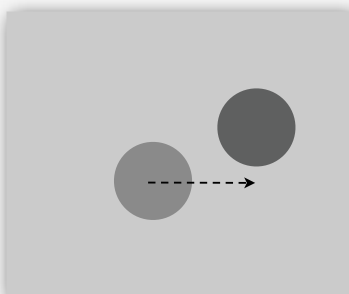
- Physics problems: when? what effect?
- CS problems: what object does the checks? too many checks?

Time-driven simulation

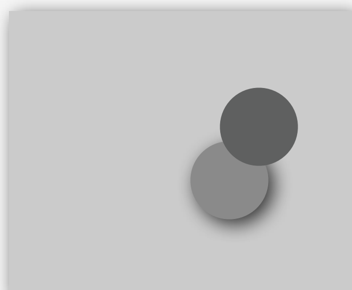
- Discretize time in quanta of size dt.
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.



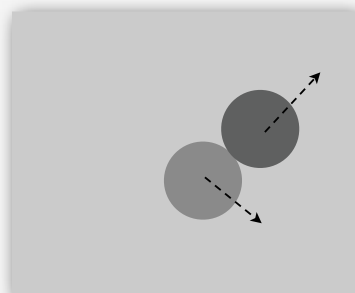
t



t + dt



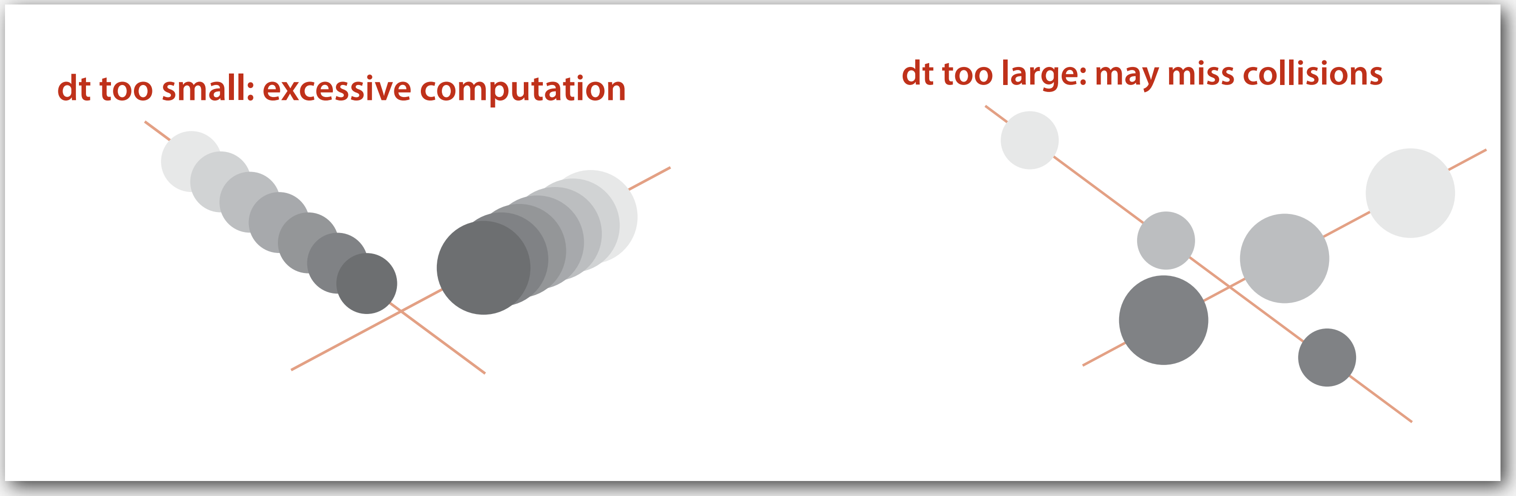
t + 2 dt
(collision detected)



t + Δt
(roll back clock)

Main drawbacks.

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



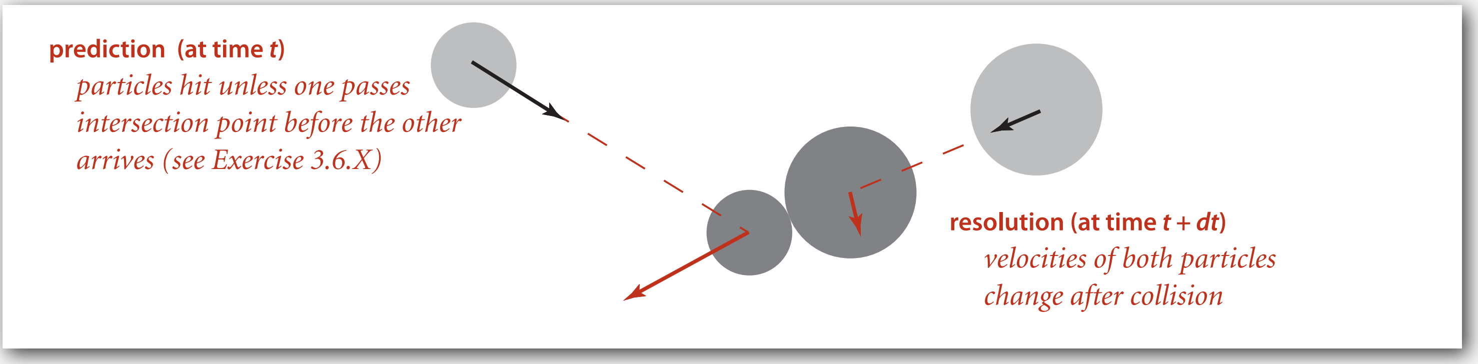
Event-driven simulation

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?

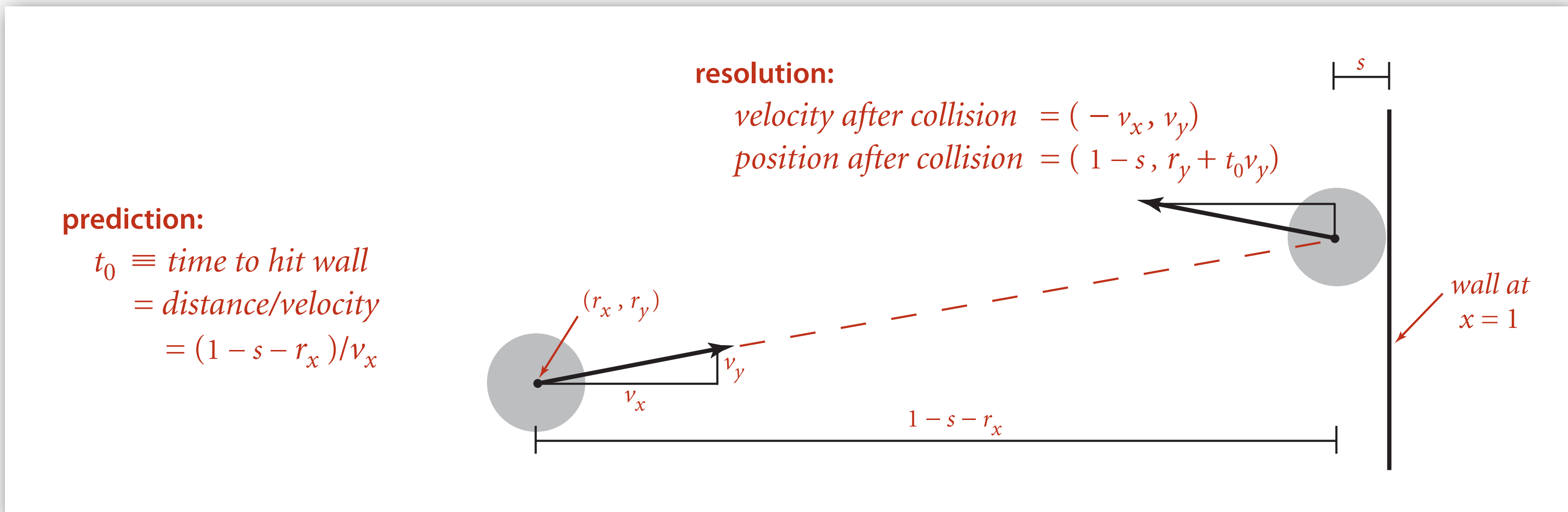
Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.



Particle-wall collision

Collision prediction and resolution.

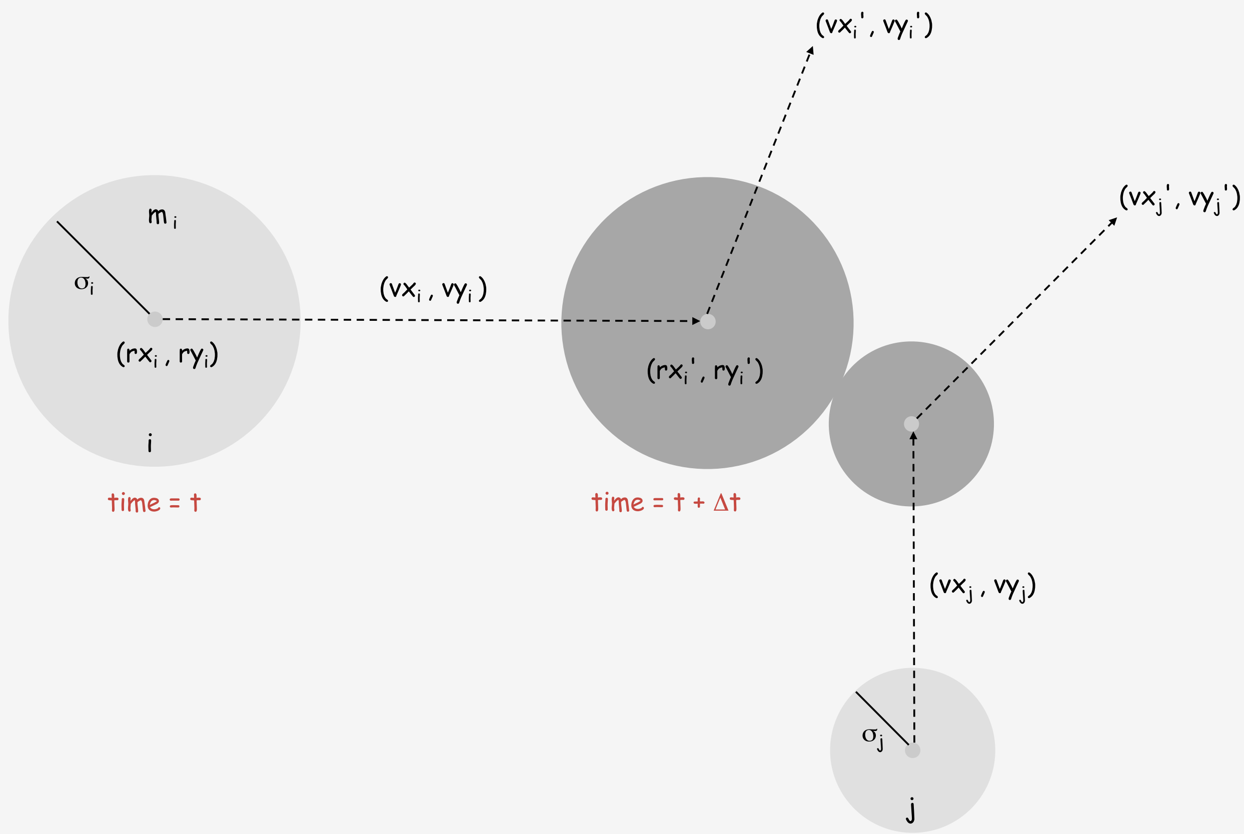
- Particle of radius σ at position (rx, ry).
- Particle moving in unit box with velocity (vx, vy).
- Will it collide with a vertical wall? If so, when?



Particle-particle collision prediction

Collision prediction.

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

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

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

$$\Delta v = (\Delta vx,\ \Delta vy) = (vx_i - vx_j,\ vy_i - vy_j)$$
$$\Delta r = (\Delta rx,\ \Delta ry) = (rx_i - rx_j,\ ry_i - ry_j)$$

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

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?

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

Newton's second law
(momentum form)

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

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 dt(Particle that) { }
    public double dtX() { }
    public double dtY() { }

    public void bounce(Particle that) { }
    public void bounceX() { }
    public void bounceY() { }

}
```

predict collision with particle or wall

resolve collision with particle or wall

Particle-particle collision and resolution implementation

```
public double dt(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;
   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;
}
```

no collision

```
public void bounce(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++;
}
```

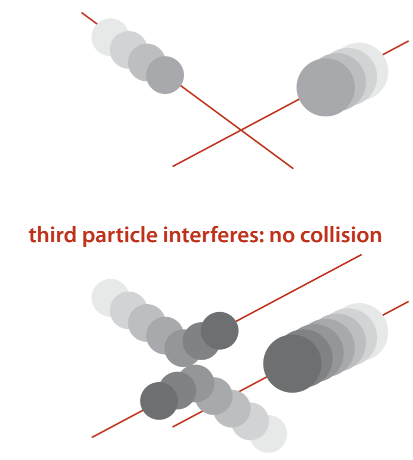
Important note: This is high-school physics, so we won't be testing you on it!

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



An invalidated event

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` ⇒ particle-particle collision.
- One particle `null` ⇒ particle-wall collision.
- Both particles `null` ⇒ redraw event.

```
public 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) { }

    public double time()    { return time; }
    public Particle a()     { return a;    }
    public Particle b()     { return b;    }

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

    public boolean isValid()
    {   }
}
```

create event

accessor methods

ordered by time

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)
    {
       if (a == null) return;
       for (int i = 0; i < N; i++)
       {
          double dt = a.dt(particles[i]);
          pq.insert(new Event(t + dt, a, particles[i]));
       }
       pq.insert(new Event(t + a.dtX(), a, null));
       pq.insert(new Event(t + a.dtY(), null, a));
 }

    private void redraw()  { }

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

add all particle-wall and particle-particle collisions involving this particle to the PQ

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));

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

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

      if      (a != null && b != null) a.bounce(b);
      else if (a != null && b == null) a.bounceX()
      else if (a == null && b != null) b.bounceY();
      else if (a == null && b == null) redraw();

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

initialize PQ with collision events and redraw event

get next event

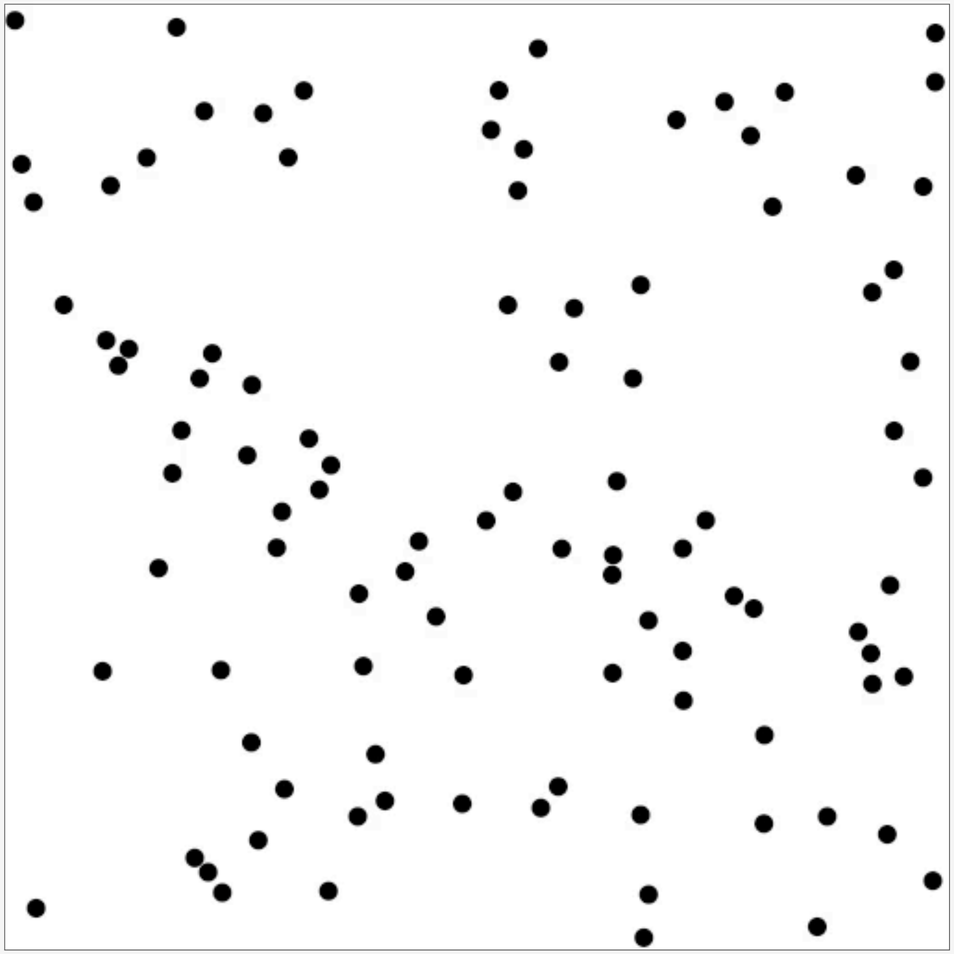
update positions and time

process event

predict new events based on changes

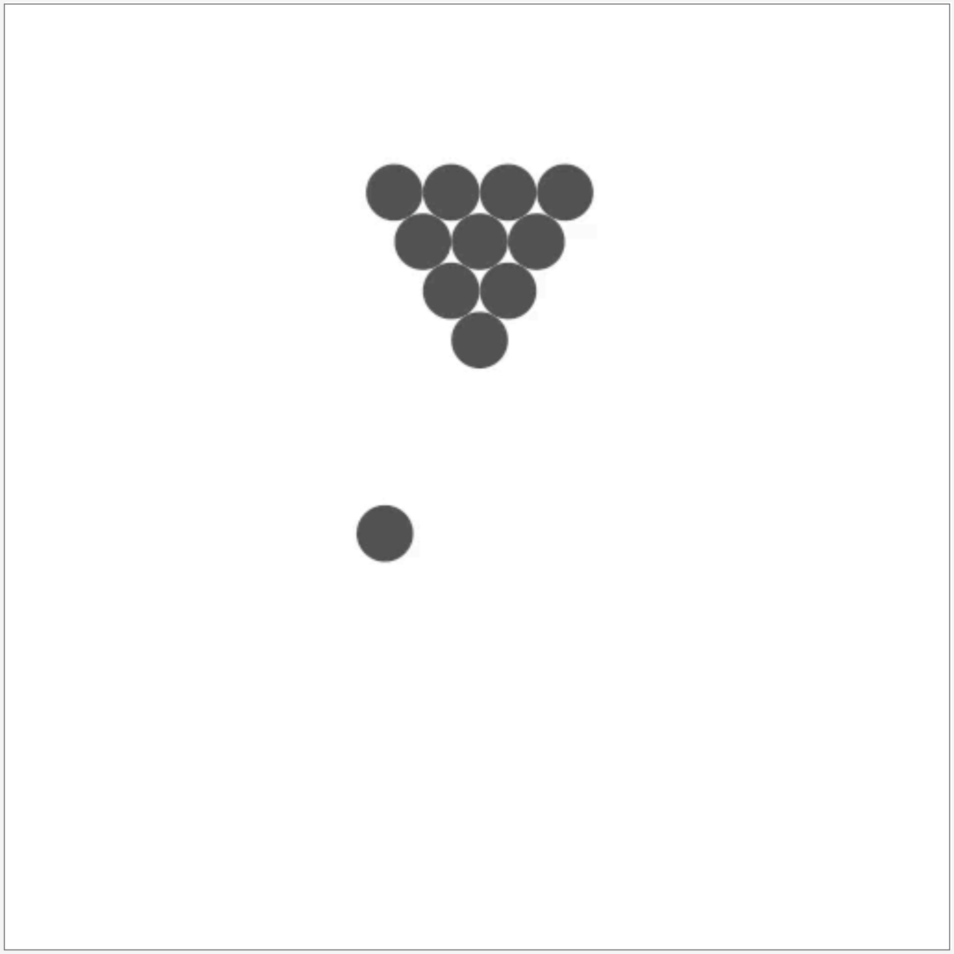
Simulation example 1

`% java CollisionSystem 100`



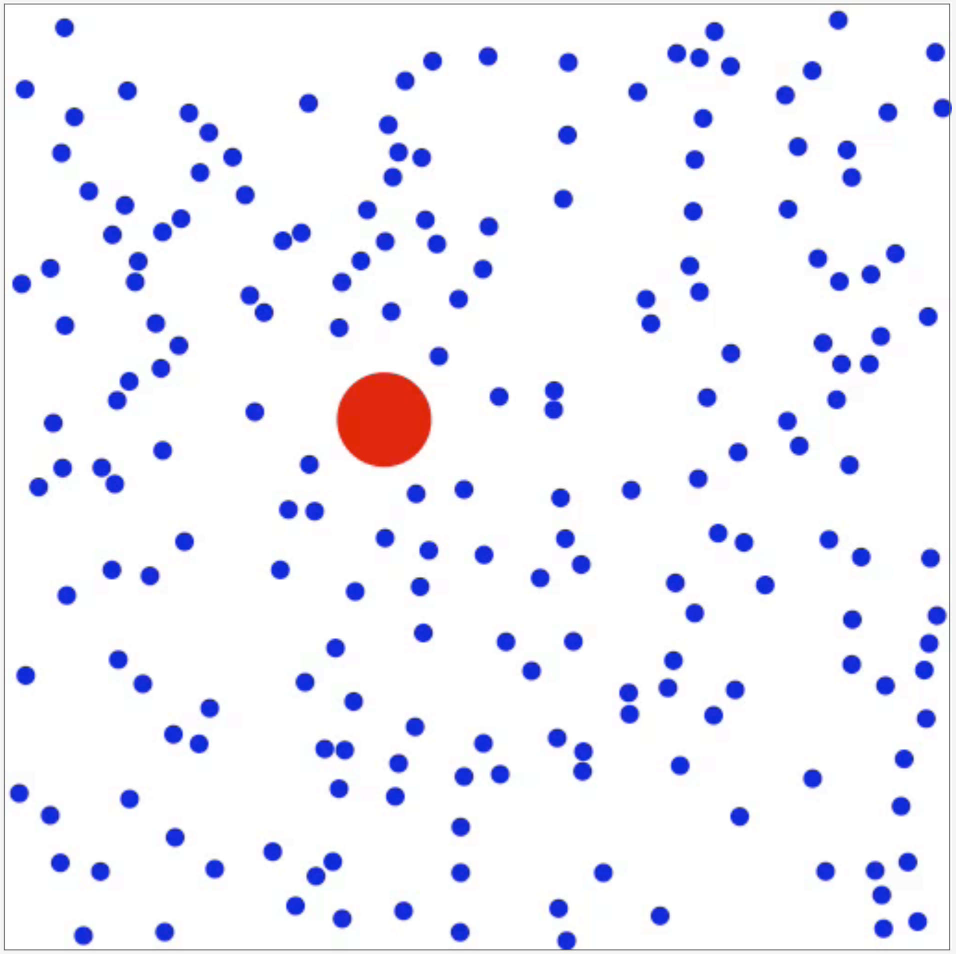
Simulation example 2

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



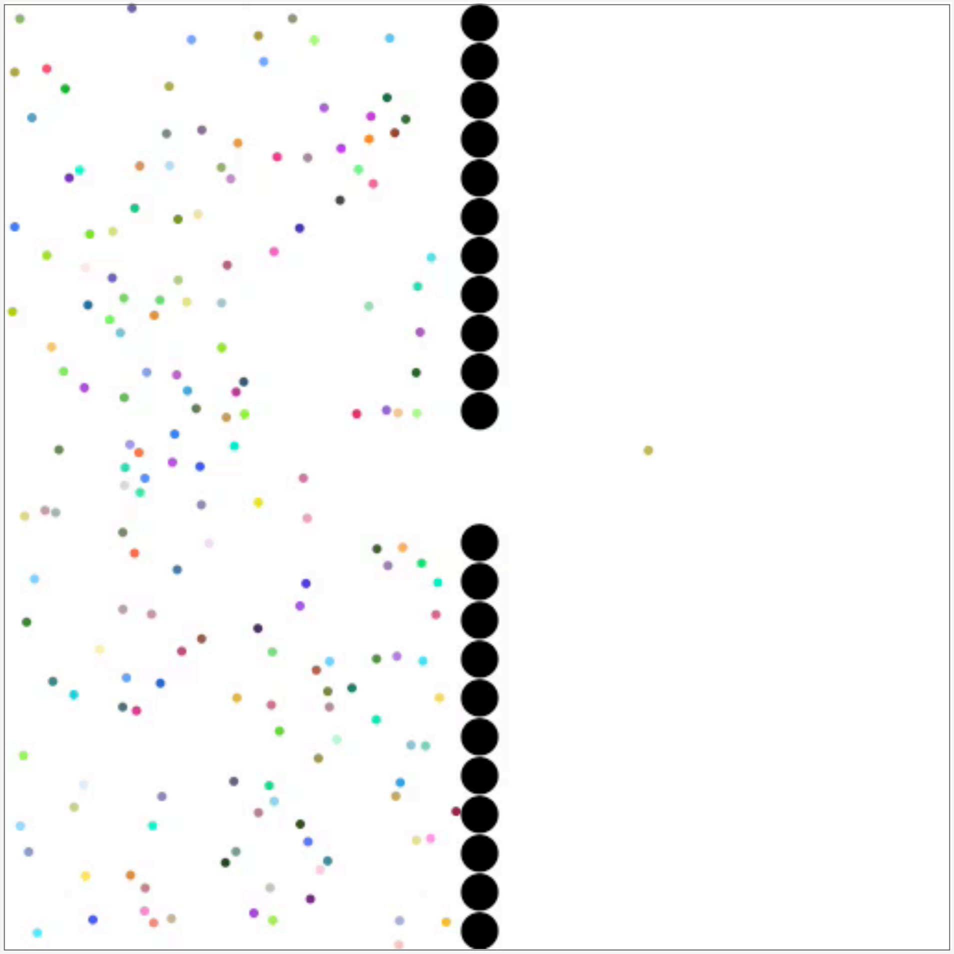
Simulation example 3

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



Simulation example 4

% java CollisionSystem < diffusion.txt



Symbol Tables

- API
- sequential search
- binary search
- BSTs
- ordered operations
- deletion in BSTs

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 ↑ (URL) value ↑ (IP address)

Symbol table applications

application	purpose of search	key	value
dictionary	look up word	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	value and type
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

Symbol table API

Associative array abstraction. Associate one value with each key.

`public class ST<Key, Value>`

	`ST()`	*create a symbol table*
`void`	`put(Key key, Value val)`	*put key-value pair into the symbol table (remove* `key` *from table if value is null)* ← `a[key] = val;`
`Value`	`get(Key key)`	*value paired with* `key` *(*`null` *if* `key` *is absent)* ← `a[key]`
`void`	`delete(Key key)`	*remove* `key` *(and its value) from table*
`boolean`	`contains(Key key)`	*is there a value paired with* `key`*?*
`boolean`	`isEmpty()`	*is the table empty?*
`int`	`size()`	*number of key-value pairs in the table*
`Iterable<Key>`	`keys()`	*all the keys in the symbol table*

API for a generic basic symbol table

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 boolean 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 and `hashCode()` to scramble key.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `BigInteger`, …
- Mutable in Java: `Date`, `GregorianCalendar`, `StringBuilder`, …

ST test client for traces

Build ST by associating value i with ith command-line argument.

```
public static void main(String[] args)
{
   ST<String, Integer> st = new ST<String, Integer>();
   for (int i = 0; i < args.length; i++)
      st.put(args[i], i);
   for (String s : st)
      StdOut.println(s + " " + st.get(s));
}
```

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

output

```
A  8
C  4
E  12
H  5
L  9
M  11
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 the number of times each string appears.

```
% more tiny.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


% java FrequencyCounter 0 < tiny.txt
2 age
1 best
1 foolishness
4 it
4 of
4 the
2 times
4 was
1 wisdom
1 worst
```

tiny example
24 words
10 distinct

```
% more tale.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
...

% java FrequencyCounter 0 < tale.txt
2941 a
1 aback
1 abandon
10 abandoned
1 abandoning
1 abandonment
1 abashed
1 abate
1 abated
...
```

real example
137177 words
9888 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>();
      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (word.length() < minlen) continue;
         if (!st.contains(word)) st.put(word, 1);
         else                    st.put(word, st.get(word) + 1);
      }
      String max = "";
      for (String word : st.keys())
         if (st.get(word) > st.get(max))
            max = word;
      StdOut.println(max + " " + st.get(max));
   }
}
```

create ST

ignore short strings

read string and update frequency

print all strings

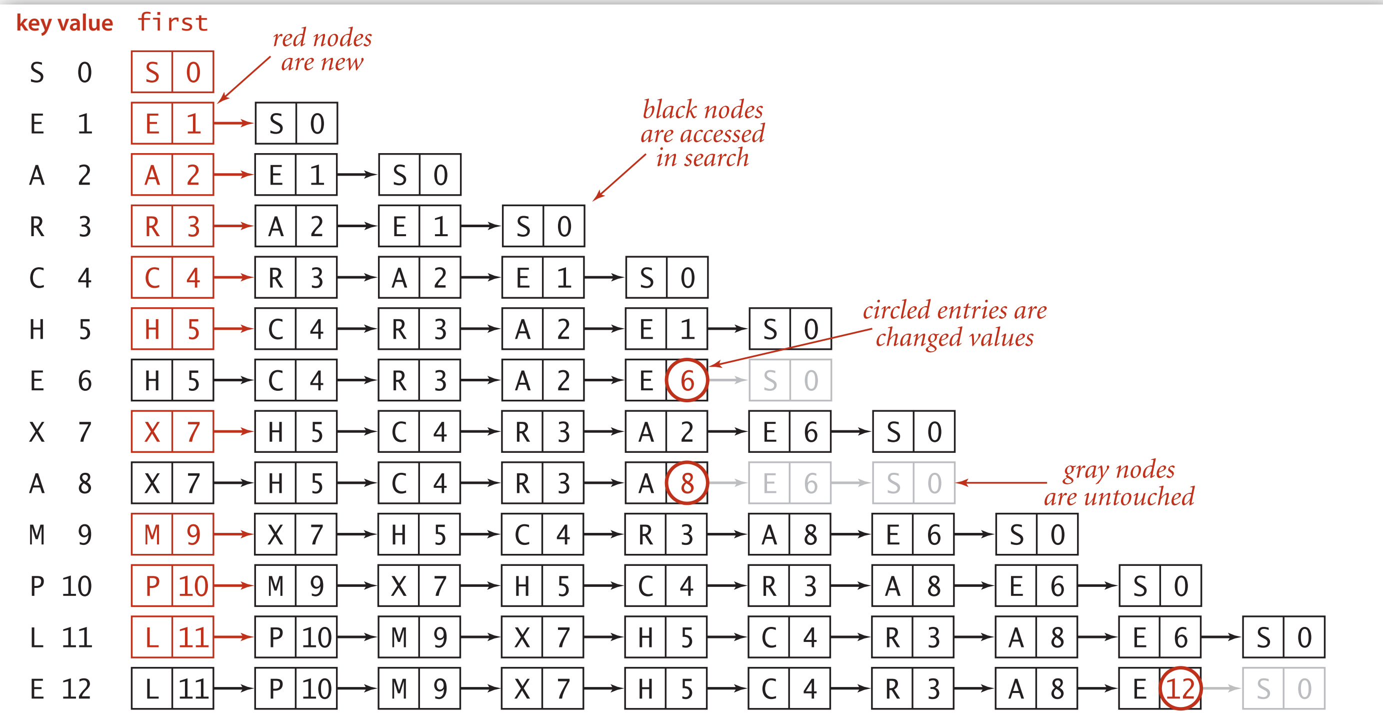
- API
- **sequential search**
- binary search
- BSTs
- applications

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.

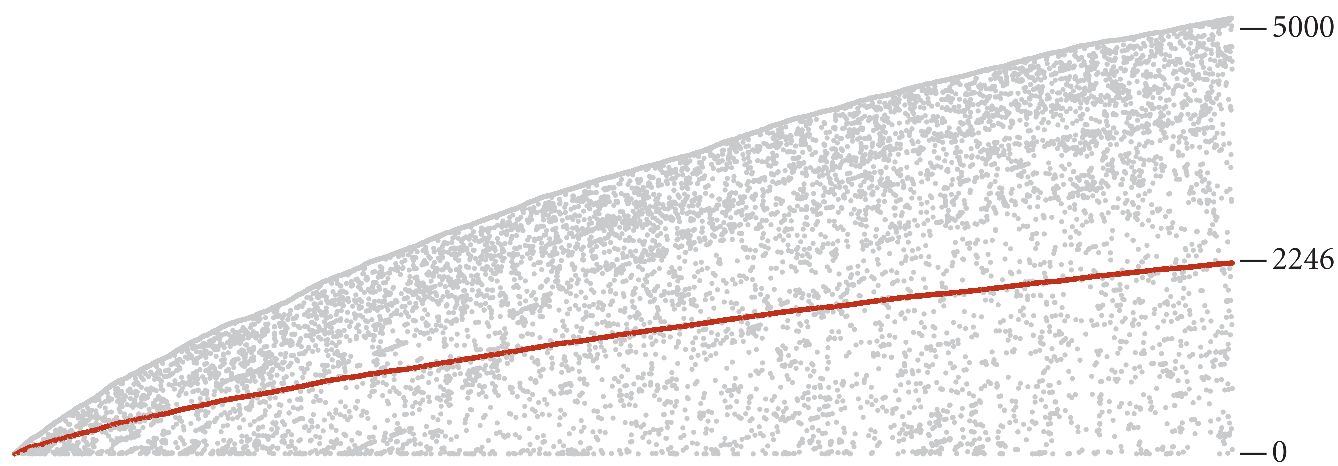
Insert. Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

Elementary ST implementations: summary

ST implementation	worst case		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
sequential search (unordered list)	N	N	$N / 2$	N	no	`equals()`



Costs for `java FrequencyCounter 8 < tale.txt` **using** `LinkedListST`

Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- **binary search**
- BSTs
- applications

Binary search

Data structure. Maintain an ordered array of key-value pairs.

Search. Binary search.

Insert. Binary search for key; if no match insert and shift larger keys.

successful search for P

lo	hi	m	keys[] 0	1	2	3	4	5	6	7	8	9
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	1	2	3	4	5	6	7	8	9
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)
{
   int i = bsearch(key);
   if (i == -1) return null;
   return vals[i];
}
```

symbol table method

```
private int bsearch(Key key)
{
   int lo = 0, hi = N-1;
   while (lo <= hi)
   {
       int m = lo + (hi - lo) / 2;
       int cmp = key.compareTo(keys[m]);
       if      (cmp  < 0) hi = m - 1;
       else if (cmp  > 0) lo = m + 1;
       else if (cmp == 0) return m;
   }
   return -1;
}
```

helper binary search method

not found

Binary search: mathematical analysis

Proposition. Binary search uses $\sim \lg N$ compares to search any array of size N.

Def. $\mathrm{T}(N) \equiv$ number of compares to binary search in a sorted array of size N.

$$\leq \underbrace{\mathrm{T}(N/2)}_{\text{left or right half}} + 1$$

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

- Not quite right for odd N.
- Same recurrence holds for many algorithms.

Solution. $\mathrm{T}(N) \sim \lg N$.

- For simplicity, we'll prove when N is a power of 2.
- True for all N. [see COS 340]

Binary search recurrence

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

Proposition. If N is a power of 2, then $T(N) \le \lg N + 1$.

Pf.

$$
\begin{aligned}
T(N) &\le T(N/2) + 1 && \text{given} \\
&\le T(N/4) + 1 + 1 && \text{apply recurrence to first term} \\
&\le T(N/8) + 1 + 1 + 1 && \text{apply recurrence to first term} \\
&\;\;\cdots \\
&\le T(N/N) + 1 + 1 + \ldots + 1 && \text{stop applying, } T(1) = 1 \\
&= \lg N + 1
\end{aligned}
$$

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

entries in red were inserted

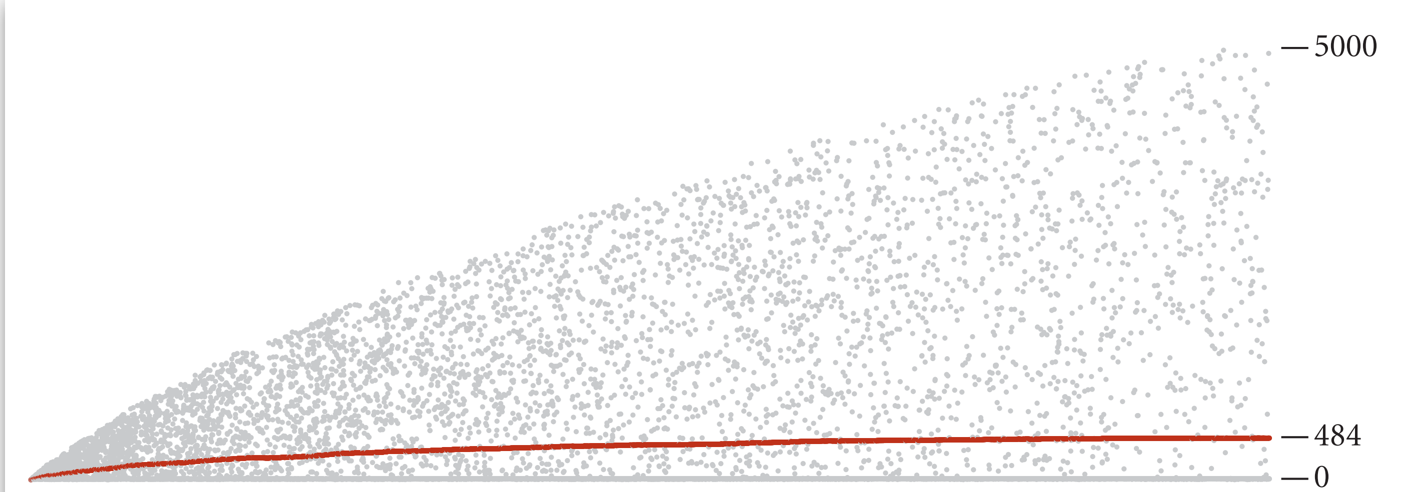
entries in gray did not move

entries in black moved to the right

circled entries are changed values

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	`equals()`
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	yes	`compareTo()`



Costs for `java FrequencyCounter 8 < tale.txt` **using** `OrderedArrayST`

Challenge. Efficient implementations of both search and insert.

- API
- sequential search
- binary search
- **challenges**

Searching challenge 1A

Problem. Maintain symbol table of song names for an iPod.

Assumption A. Hundreds of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 1B

Problem. Maintain symbol table of song names for an iPod.

Assumption B. Thousands of songs.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2A:

Problem. IP lookups in a web monitoring device.

Assumption A. Billions of lookups, millions of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 2B

Problem. IP lookups in a web monitoring device.

Assumption B. Billions of lookups, thousands of distinct addresses.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 3

Problem. Frequency counts in "Tale of Two Cities."

Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

Searching challenge 4

Problem. Spell checking for a book.

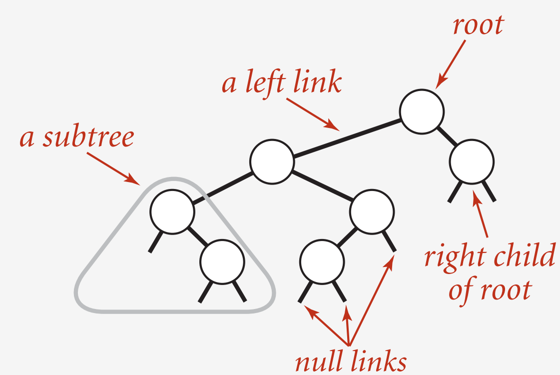
Assumptions. Dictionary has 25,000 words; book has 100,000+ words.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.

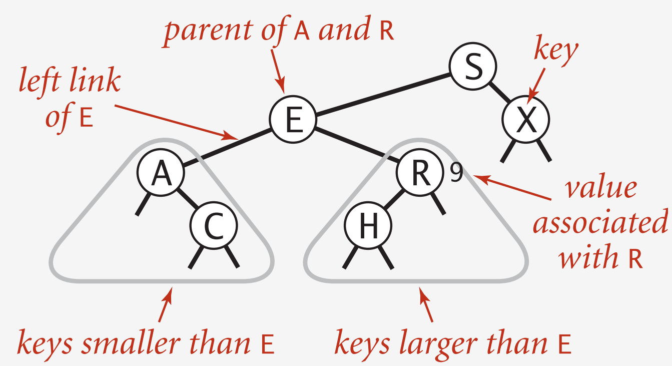
- API
- sequential search
- binary search
- challenges
- **BSTs**

Def. A BST is a binary tree in symmetric order.



Anatomy of a binary tree

A binary tree is either:
- Empty.
- Two disjoint binary trees (left and right).



Anatomy of a binary search 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.

BST representation in Java

A BST is a reference to a root node.

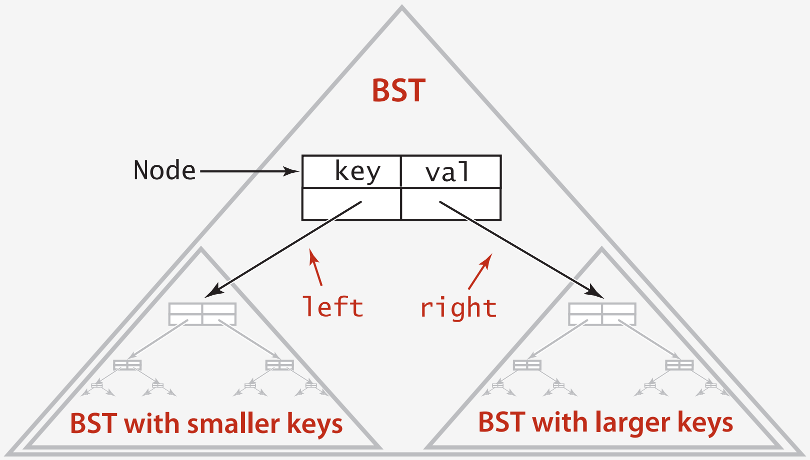
A `Node` is comprised of four fields:

- A `Key` and a `Value`.
- A reference to the left and right subtree.

smaller keys　　larger keys

```
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

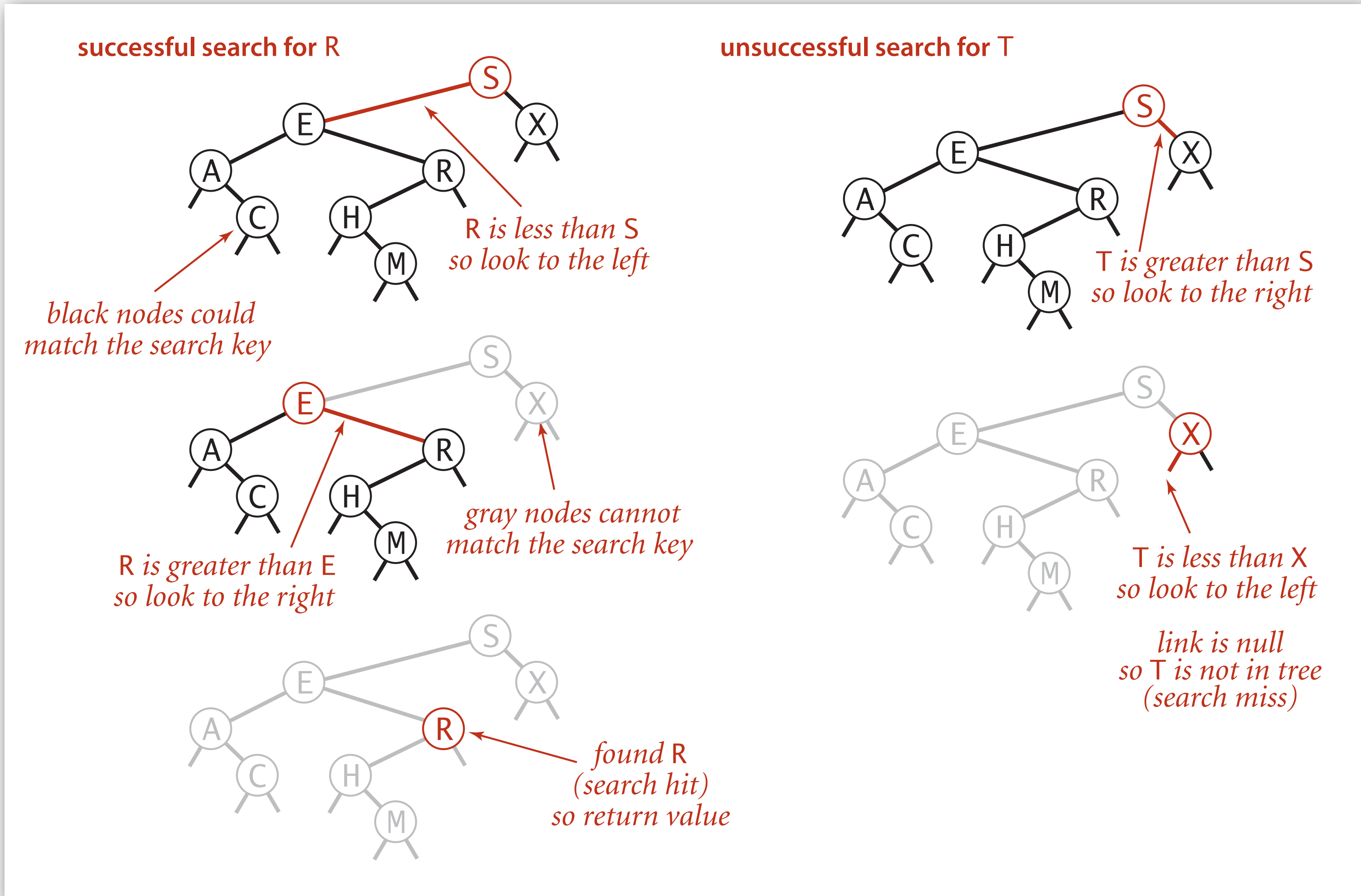
BST implementation (skeleton)

```
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

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;
}
```

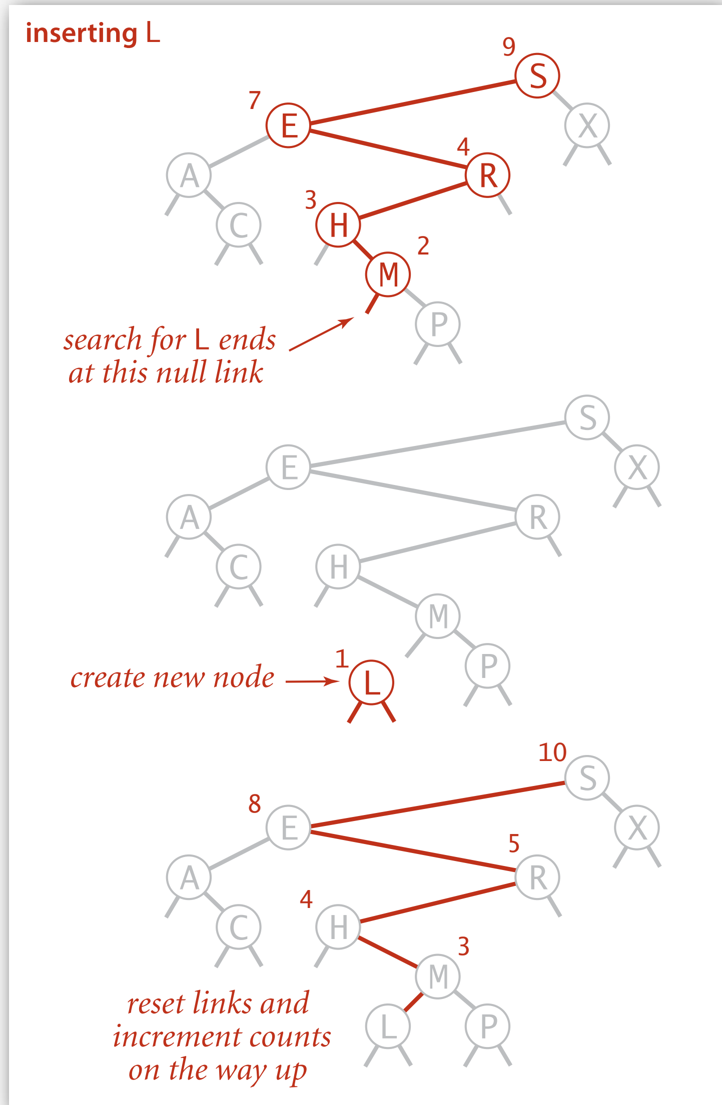
Running time. Proportional to depth of node.

BST insert

Put. Associate value with key.

Search for key, then two cases:

- key in tree: reset value
- key not in tree: add new node

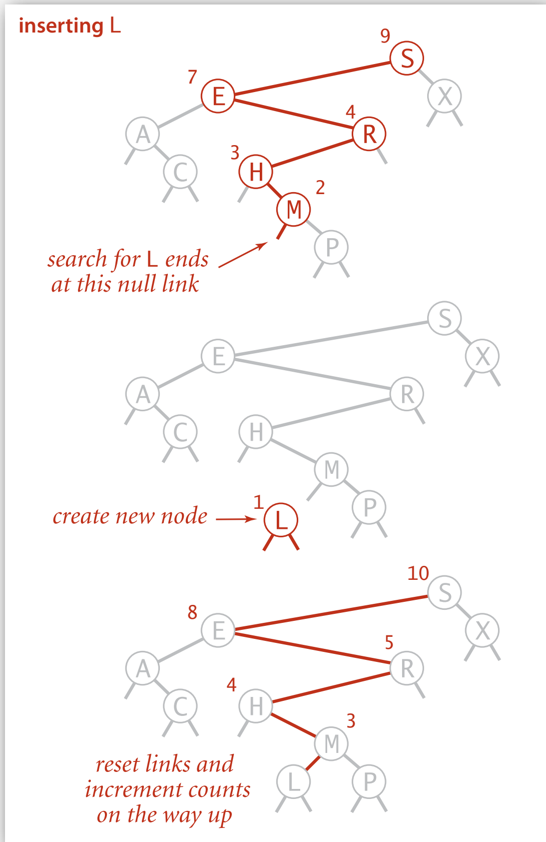


BST insert: Java implementation

Put. Associate value with key.

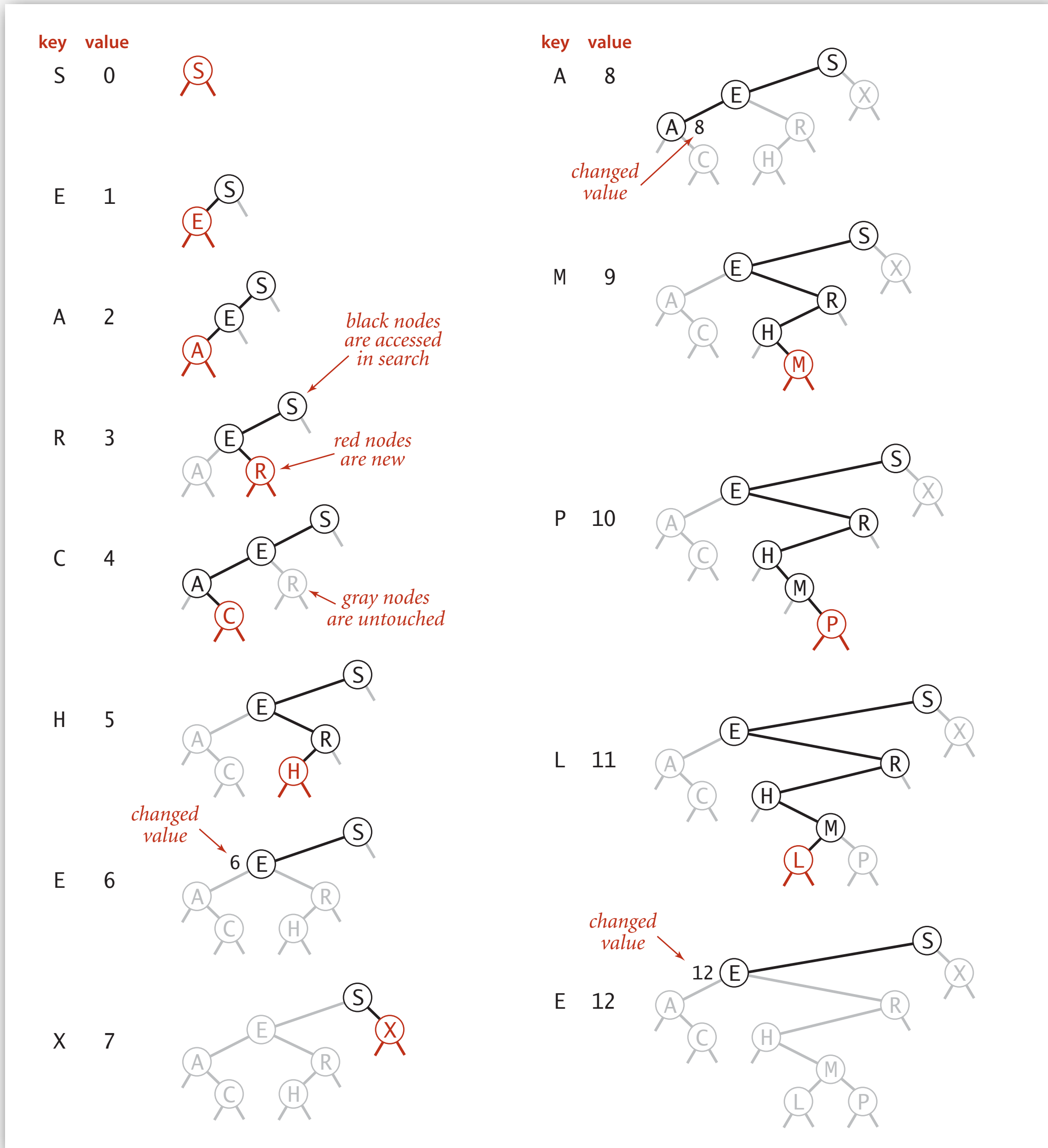
concise, but tricky, recursive code; read carefully!

```
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;
}
```



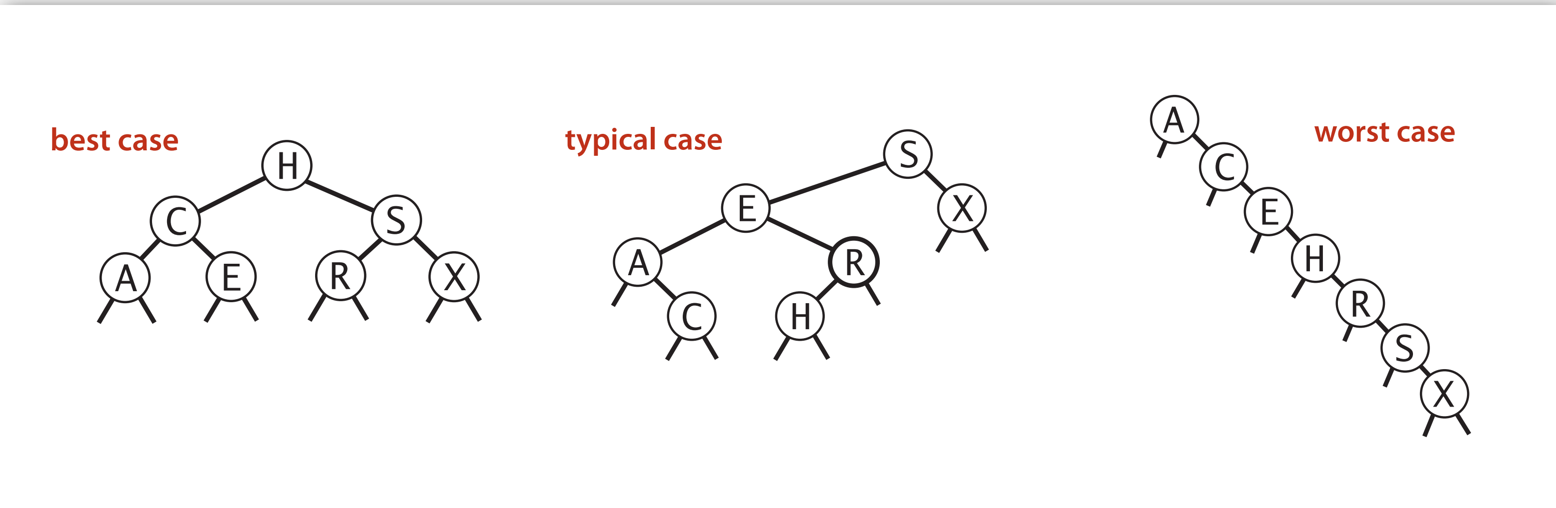
Running time. Proportional to depth of node.

BST trace: standard indexing client



Tree shape

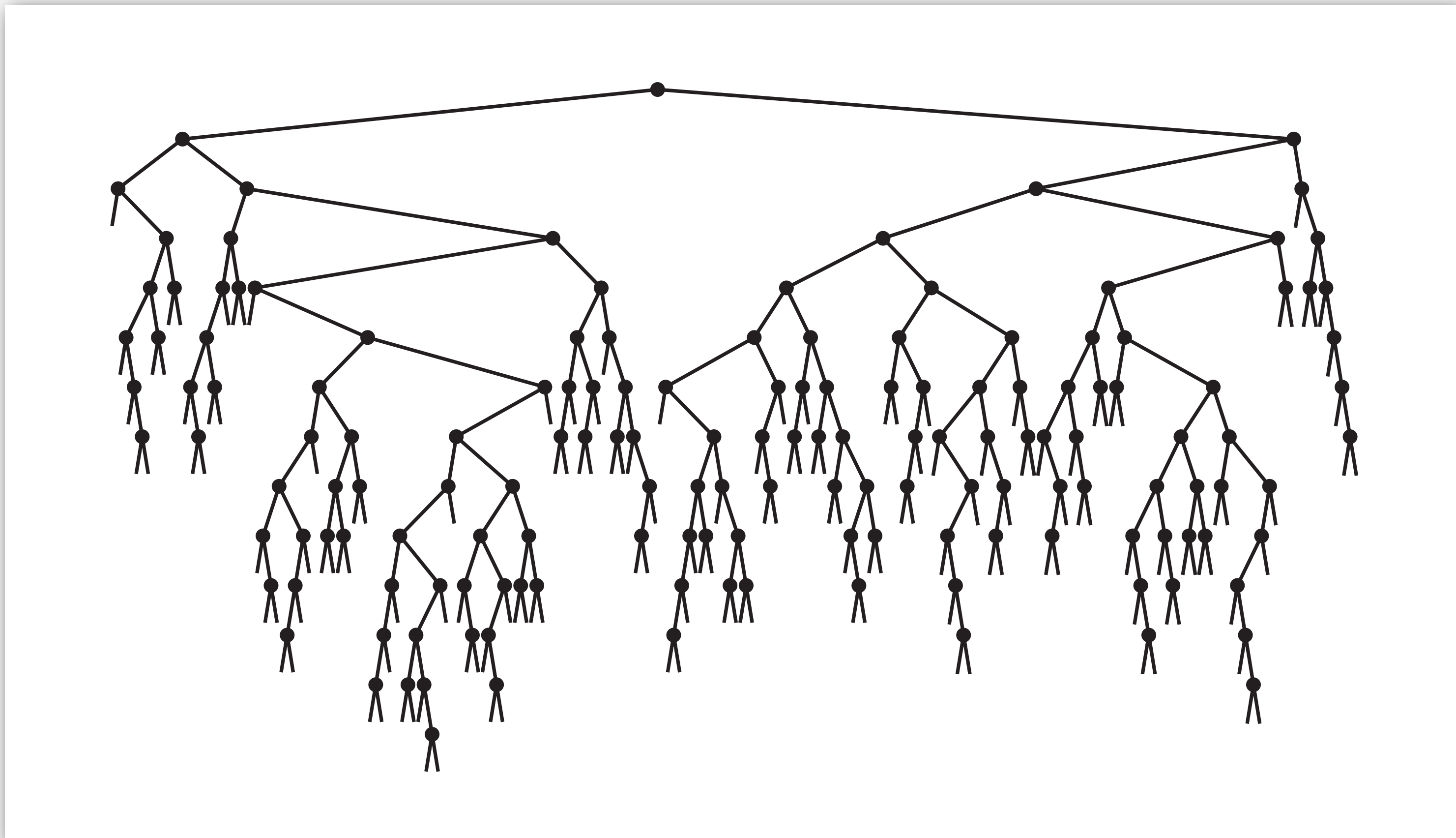
- Many BSTs correspond to same set of keys.
- Cost of search/insert is proportional to depth of node.



Remark. Tree shape depends on order of insertion.

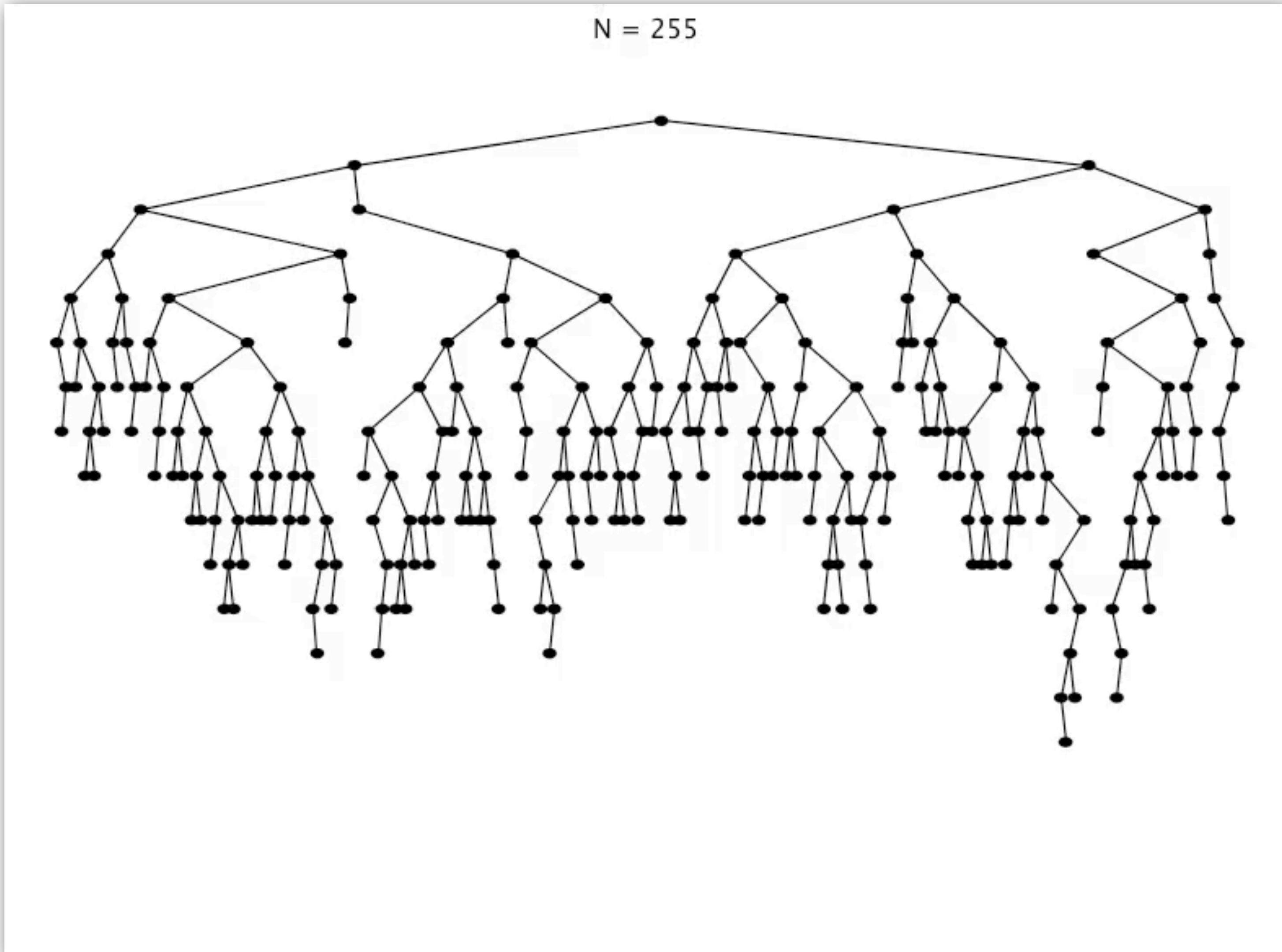
BST insertion: random order

Observation. If keys inserted in random order, tree stays relatively flat.

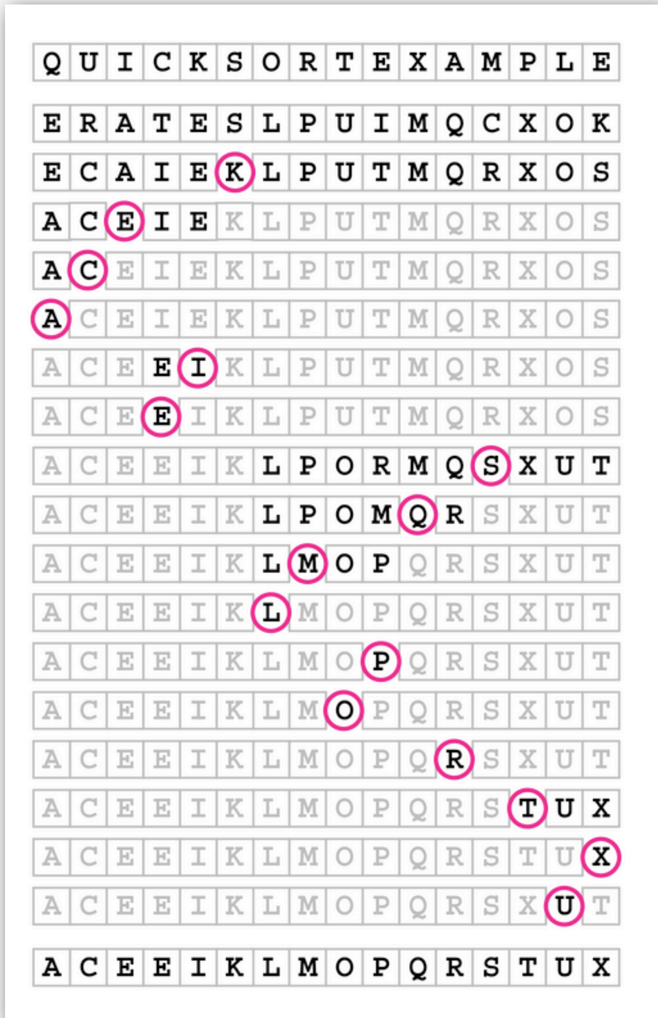


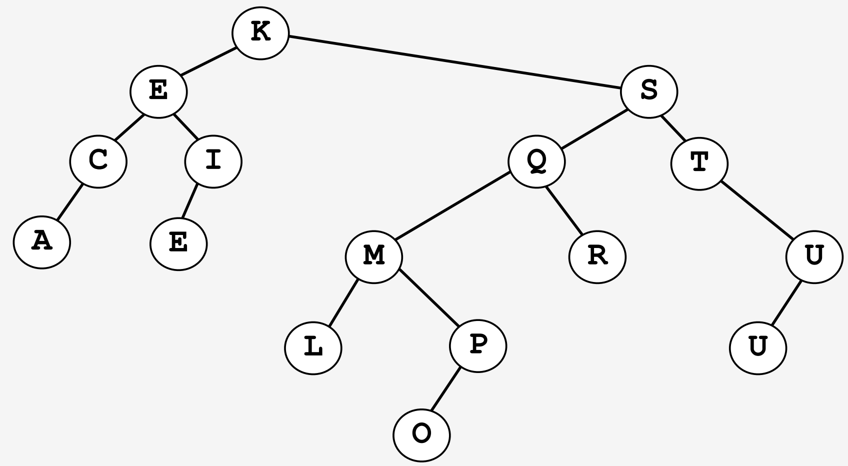
BST insertion: random order visualization

Ex. Insert keys in random order.



Correspondence between BSTs and quicksort partitioning





Remark. Correspondence is 1-1 if no duplicate keys.

BSTs: mathematical analysis

Proposition. If keys are inserted in random order, the expected number of compares for a search/insert is ~ 2 ln N.

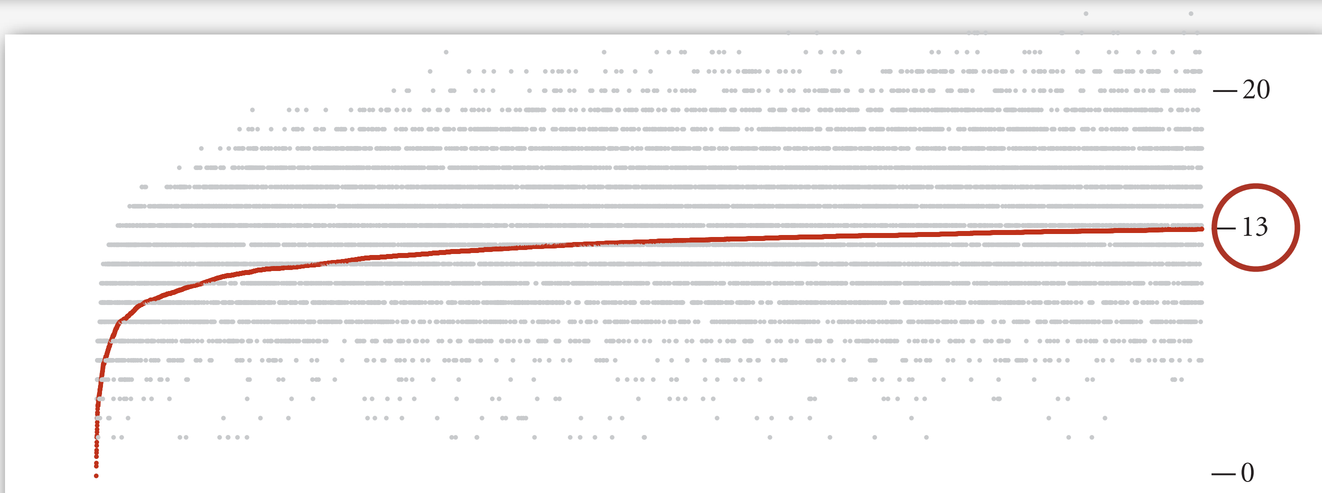
Pf. 1-1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If keys are inserted in random order, expected height of tree is ~ 4.311 ln N.

But… Worst-case for search/insert/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	`equals()`
binary search (ordered array)	lg N	N	lg N	N/2	yes	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	?	`compareTo()`



Costs for `java FrequencyCounter 8 < tale.txt` using BST

Next challenge. Ordered symbol tables ops in BSTs.

- basic implementations
- randomized BSTs
- **ordered symbol table ops**

Ordered symbol table operations

Minimum. Smallest key in table.

Maximum. Largest key in table.

Floor. Largest key ≤ to a given key.

Ceiling. Smallest key ≥ to a given key.

Rank. Number of keys < than given key.

Select. Key of given rank.

Size. Number of keys in a given range.

Iterator. All keys in order.

	keys	*values*
min() →	09:00:00	Chicago
	09:00:03	Phoenix
get(09:00:13) →	09:00:13	Houston
	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

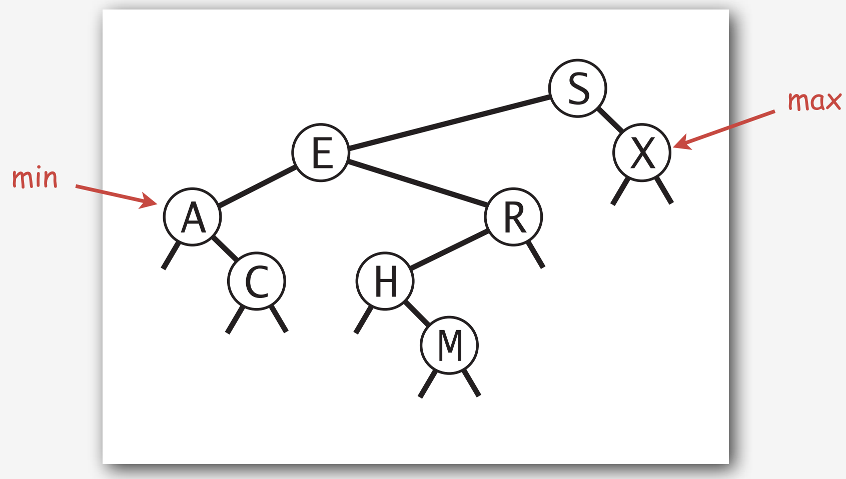
size(09:15:00, 09:25:00) *is* 5

rank(09:10:25) *is* 7

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.



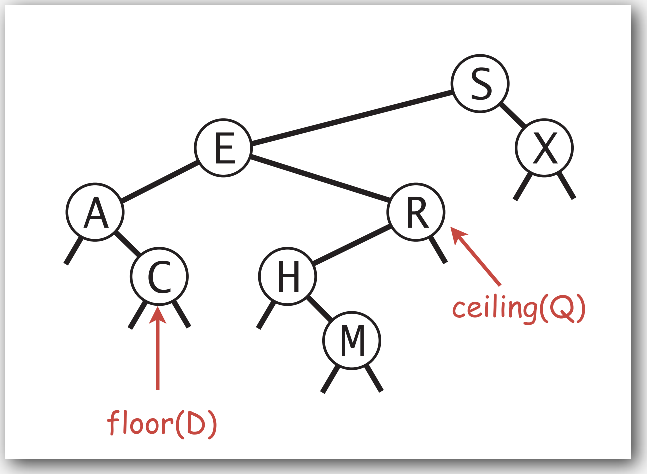
Q. How to find the min / max.

A.

Floor and ceiling

Floor. Largest key ≤ to a given key.

Ceiling. Smallest key ≥ to a given key.

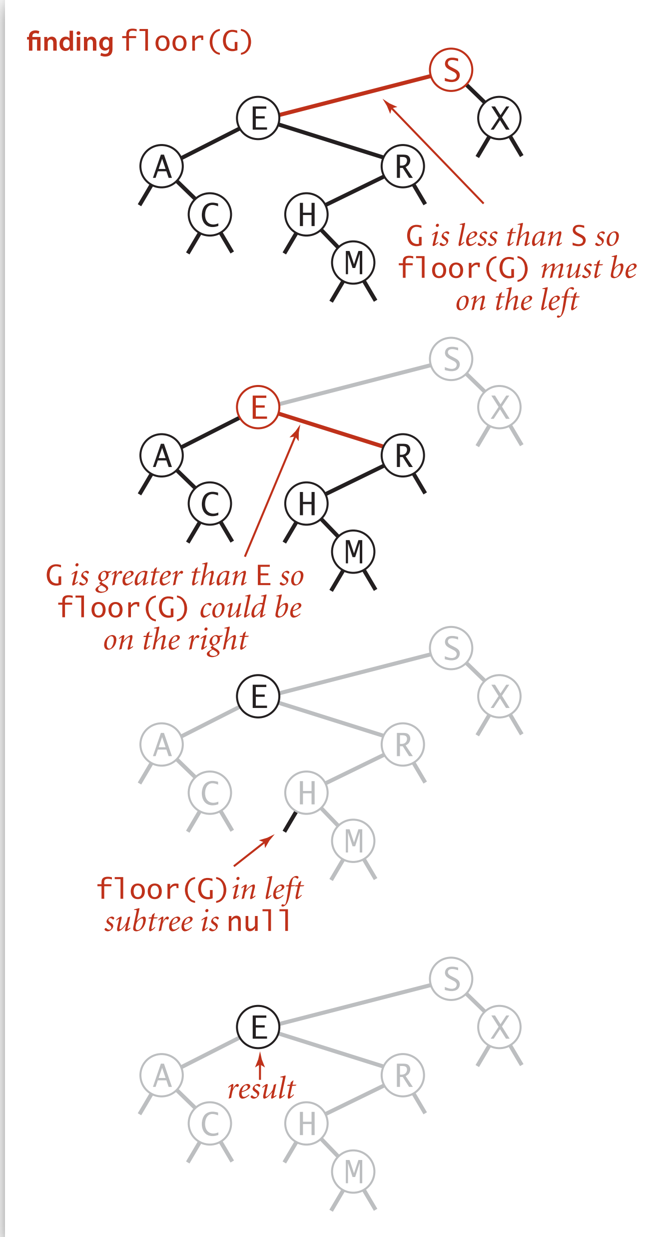


Q. How to find the floor /ceiling.

A.

Computing the floor

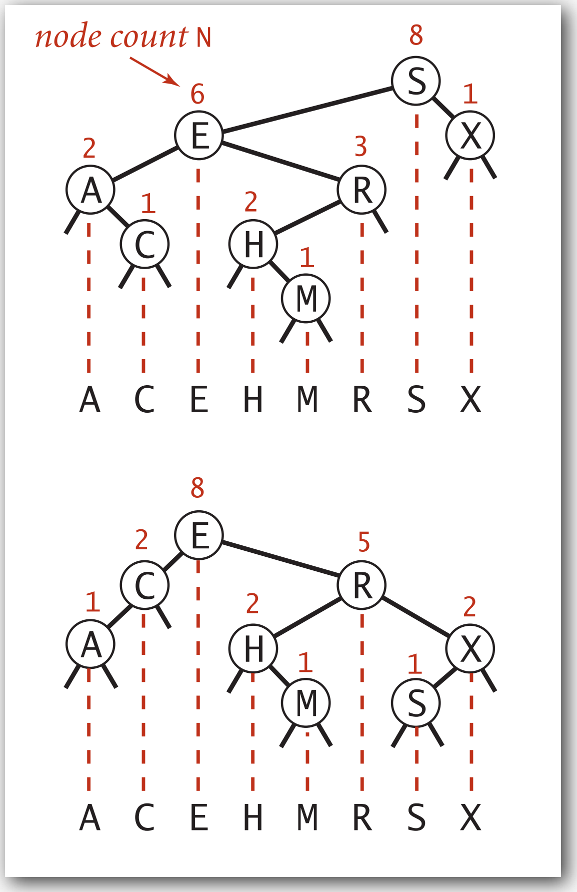
```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   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 and `size()`

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 and `size()`

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int N;
}
```

nodes in subtree

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

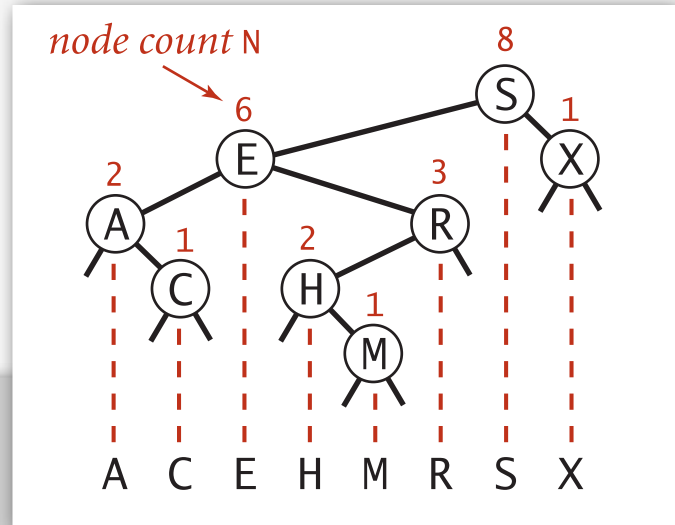
```
private int size(Node x)
{
   if (x == null) return 0;
   return x.N;
}
```

```
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

How many keys < k ?

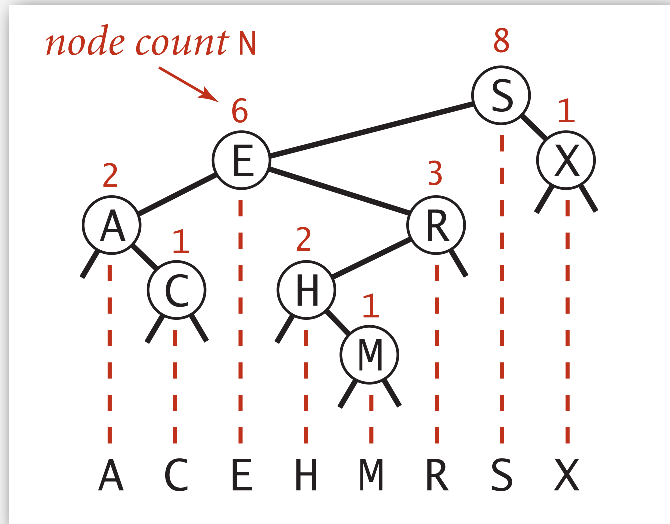
Easy recursive algorithm (4 cases!)




```
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              return size(x.left);
}
```

Range count

How many keys between `lo` and `hi`?



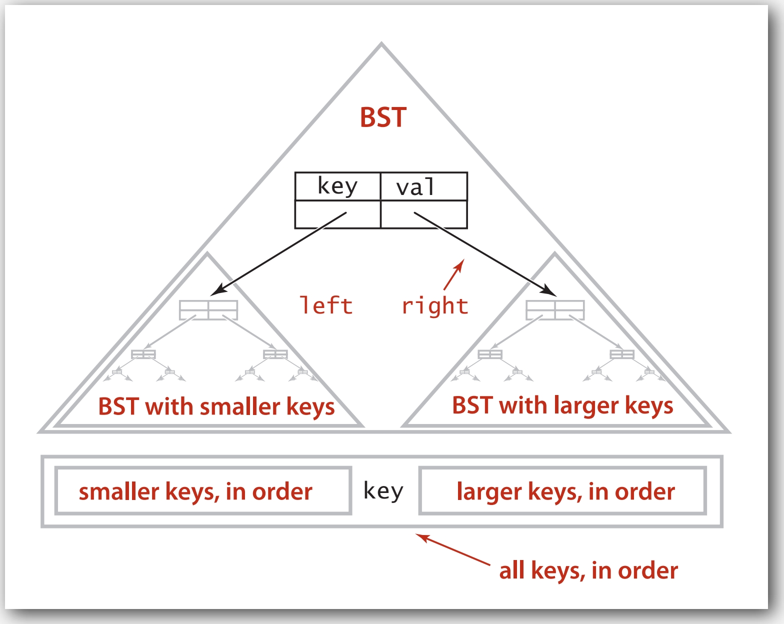

```
public int size(Key lo, Key hi)
{
   if (contains(hi)) return rank(hi) - rank(lo) - 1;
   else              return rank(hi) - rank(lo);
}
```

number of keys < hi

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> allKeys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, queue);
    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);
}
```

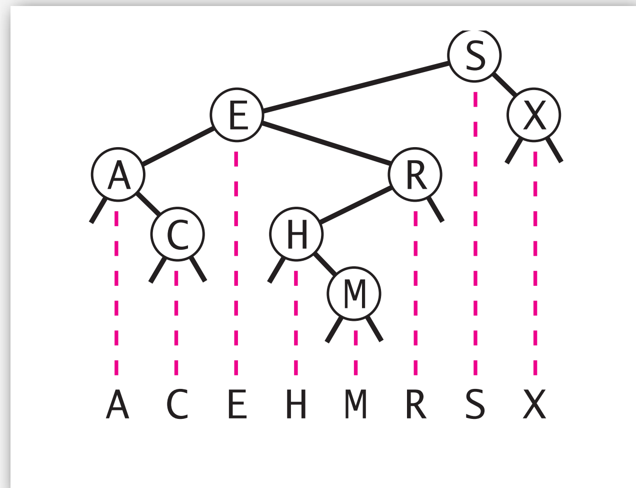


Property. Inorder traversal of a BST yields keys in ascending order.

Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

recursive calls	queue	function call stack
visit(S)		S
visit(E)		S E
visit(A)		S E A
enqueue A	A	
visit(C)		S E A C
enqueue C	C	
enqueue E	E	
visit(R)		S E R
visit(H)		S E R H
enqueue H	H	
visit(M)		S E R H M
enqueue M	M	
print R	R	
enqueue S	S	
visit(X)		S X
enqueue X	X	



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	?	yes	`compareTo()`

Next.

- Deletion in BSTs
- Can we guarantee logarithmic performance?

Searching challenge 3 (revisited):

Problem. Frequency counts in "Tale of Two Cities"

Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?

1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn't matter much, all fast enough.
5) BSTs. ✓

insertion cost < 10000 * 1.38 * lg 10000 < .2 million
lookup cost < 135000 * 1.38 * lg 10000 < 2.5 million

‣ basic implementations

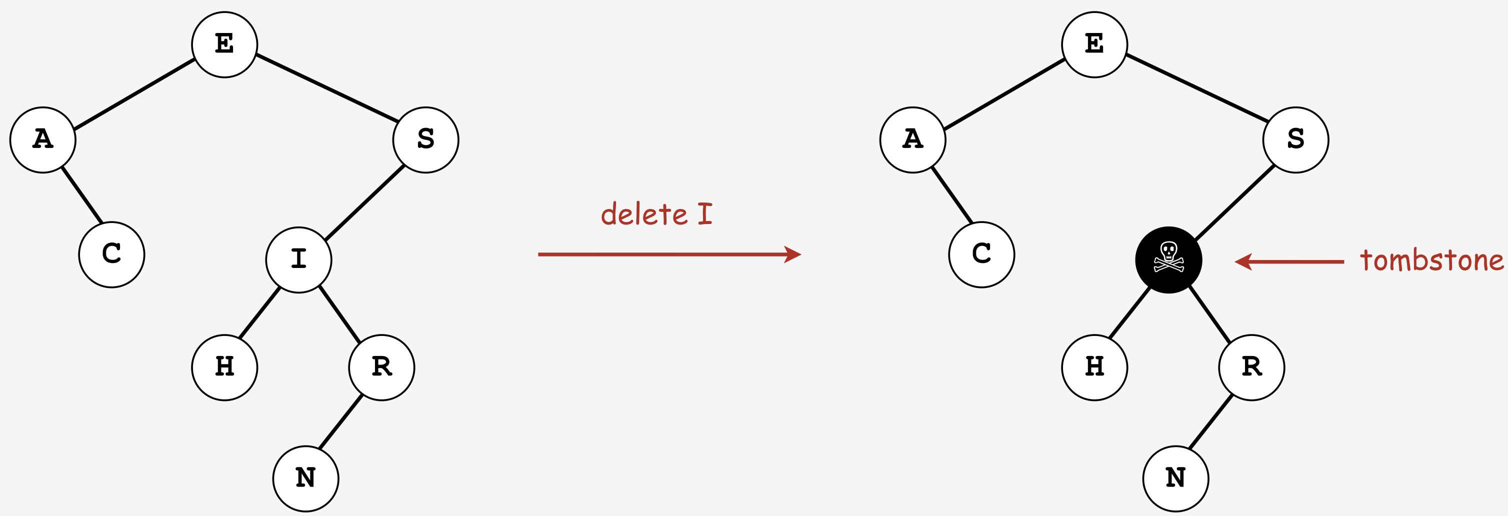
‣ randomized BSTs

‣ **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 elements ever inserted in the BST.

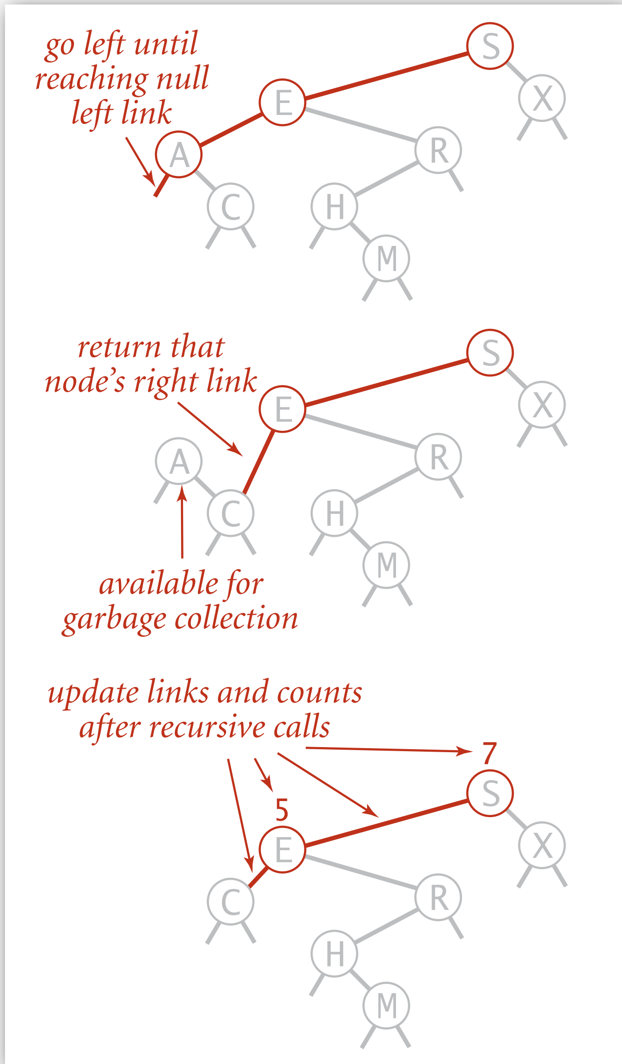
Unsatisfactory solution. Tombstone overload.

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.

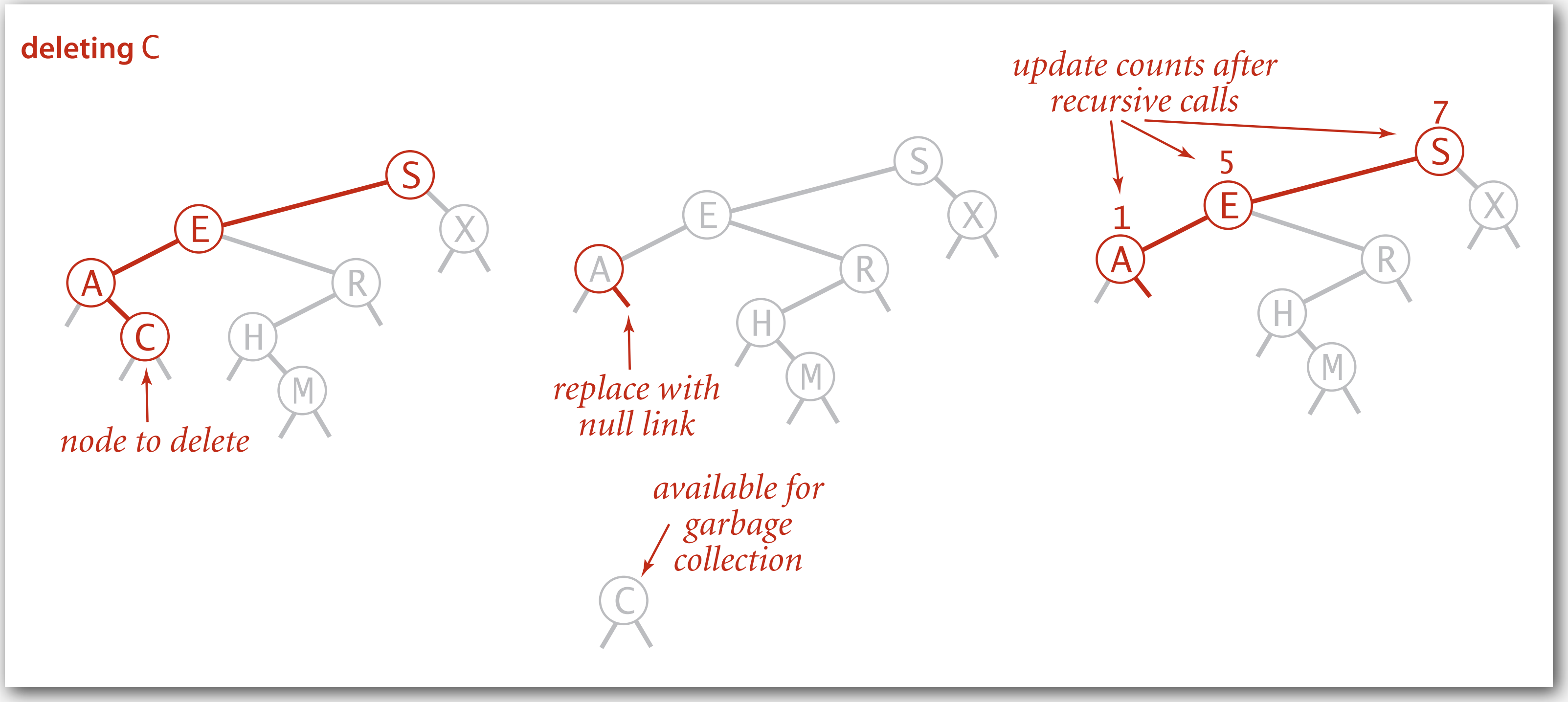
```
public void deleteMin()
{  root = deleteMin(root);  }

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.

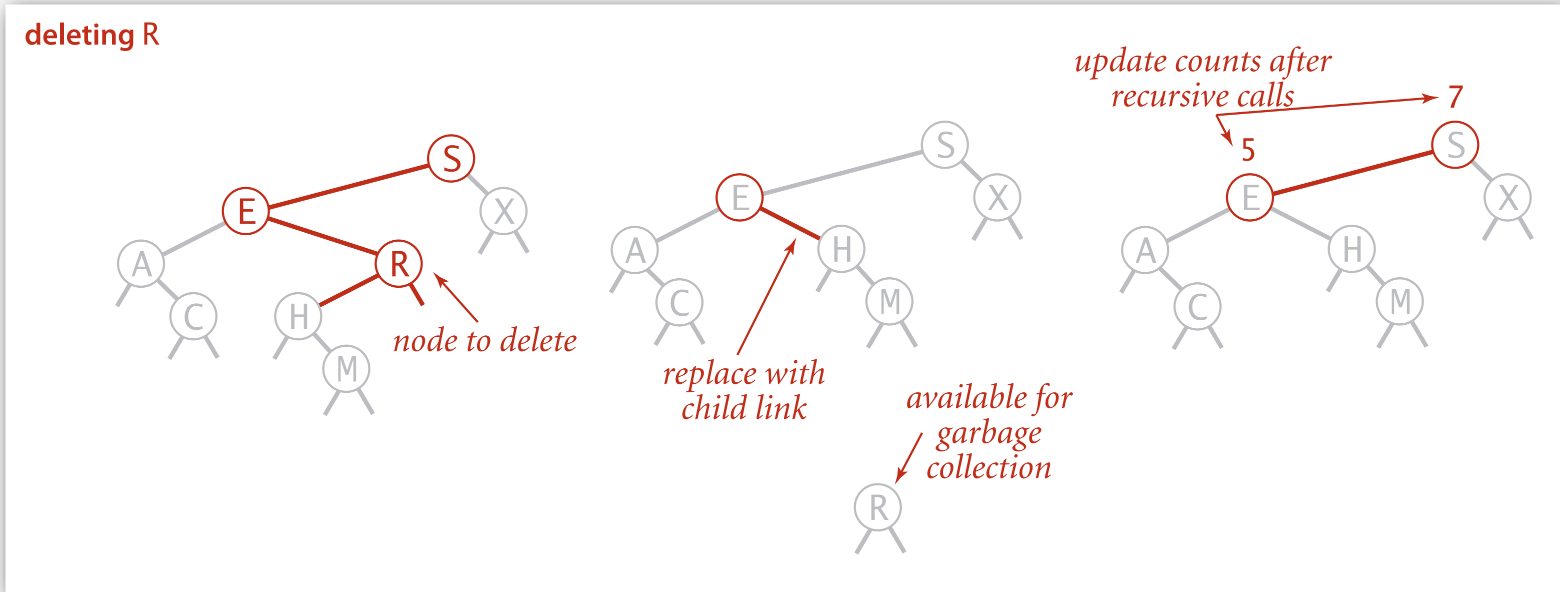
Case 0. [0 children] Delete t by setting parent link to null.



Hibbard deletion

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

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

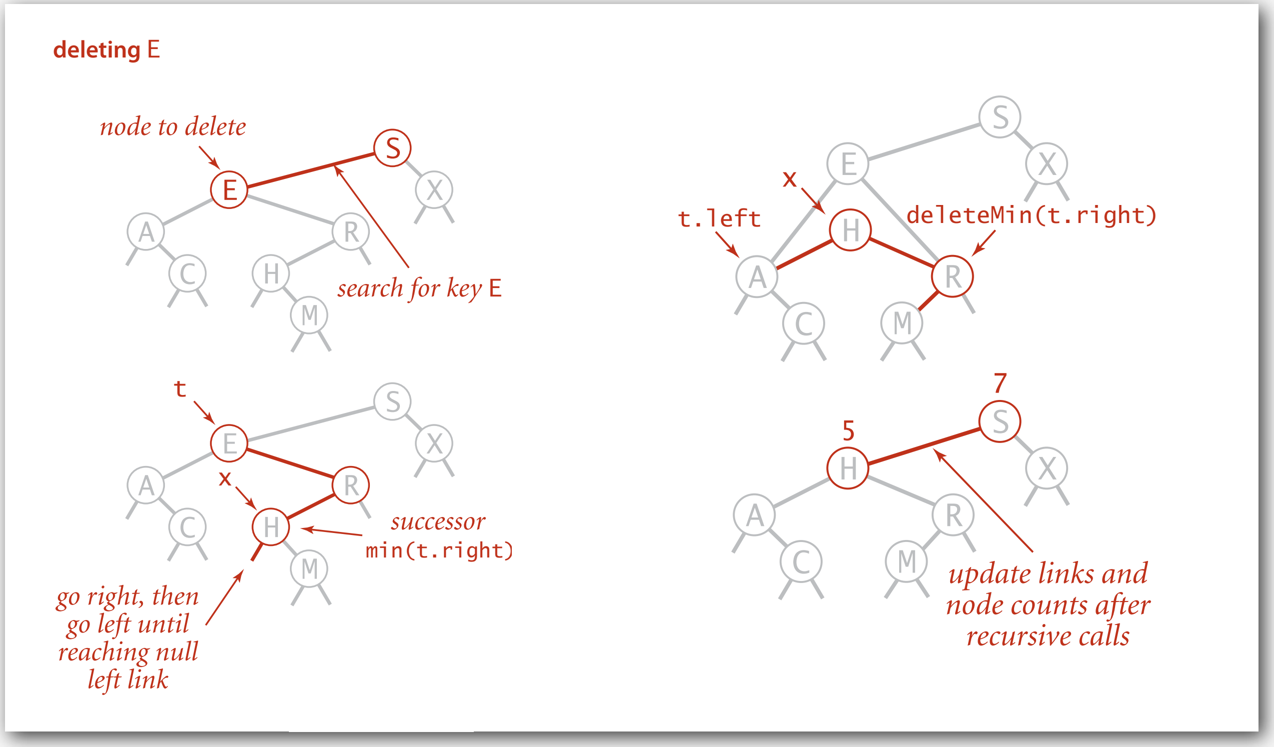


Hibbard deletion

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);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

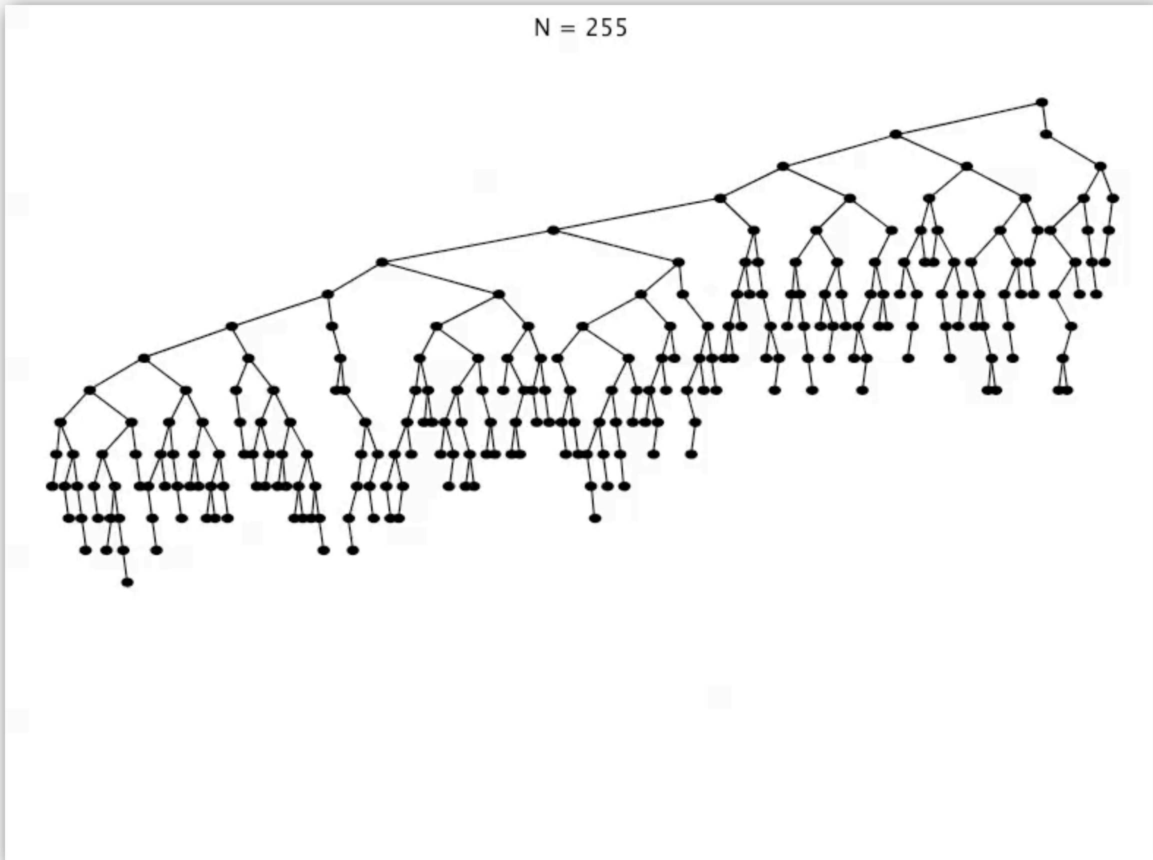
no right child

replace with successor

update subtree counts

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.

Balanced Trees

- 2-3 trees
- red-black trees
- B-trees

Reference:
Algorithms in Java. 4th Edition, Section 3.2
`http://www.cs.princeton.edu/algs4`

Symbol table review

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	?	yes	`compareTo()`
Goal	log N	log N	log N	log N	log N	log N	yes	`compareTo()`

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black trees, B-trees.

introduced to the world in
COS 226, Fall 2007
(see handout)

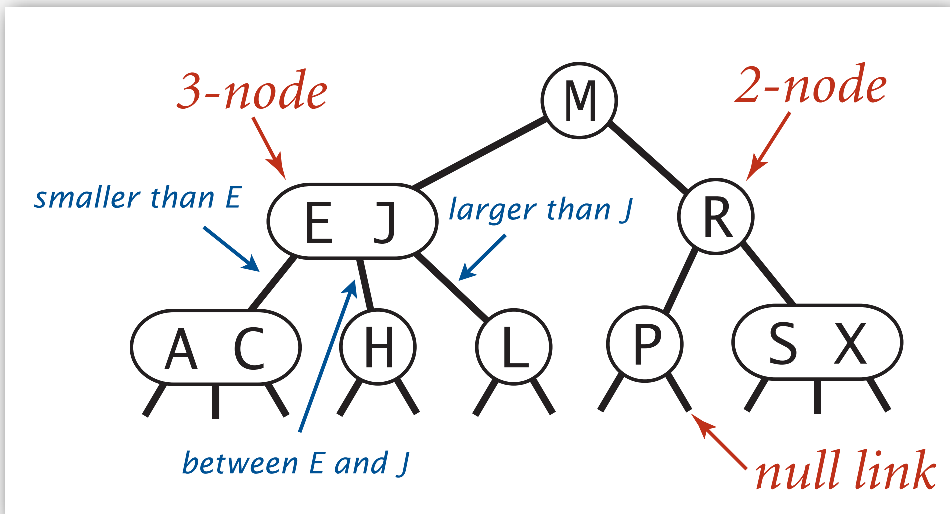
- **2-3 trees**
- **red-black trees**
- **B-trees**

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

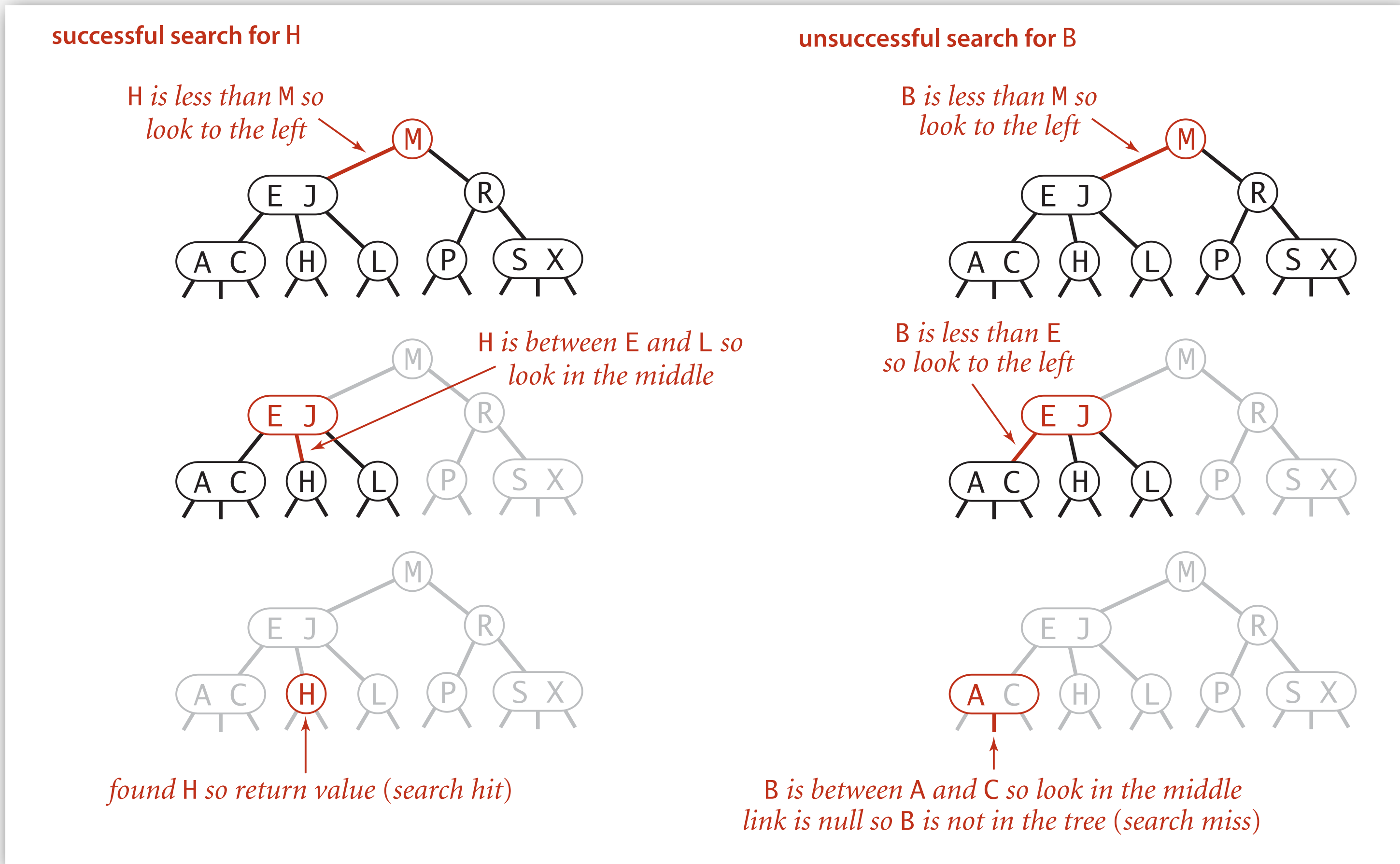
Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length.



Search in a 2-3 tree

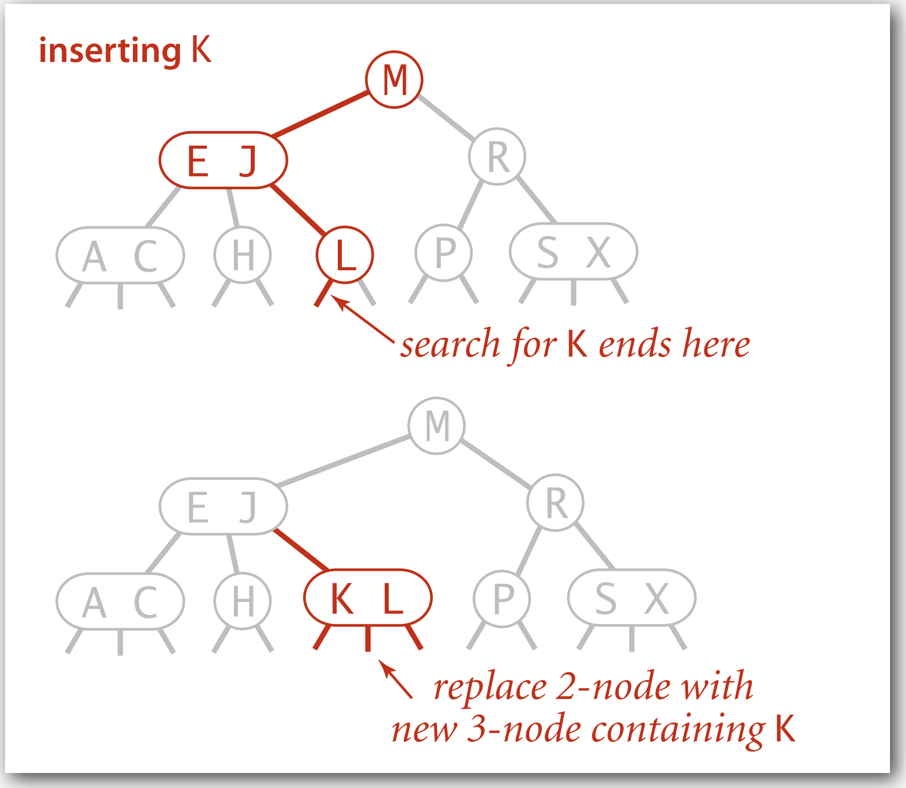
- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).



Insertion in a 2-3 tree

Case 1. Insert into a 2-node at bottom.

- Search for key, as usual.
- Replace 2-node with 3-node.

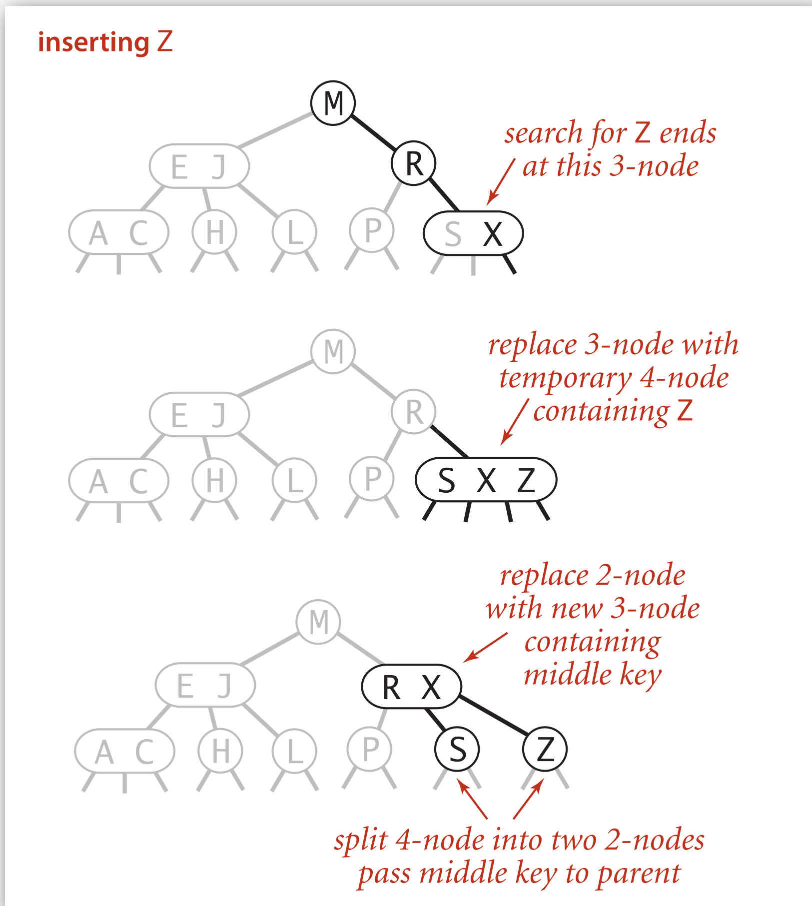


Insertion in a 2-3 tree

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.

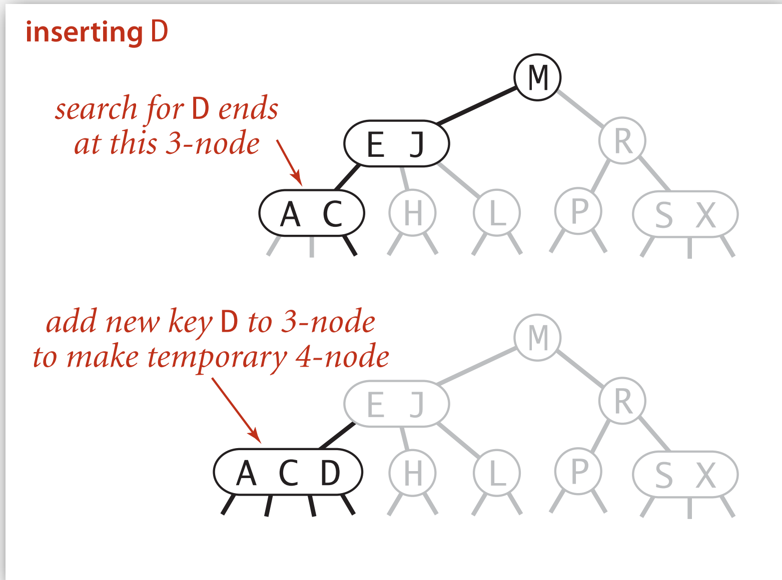
why middle key?

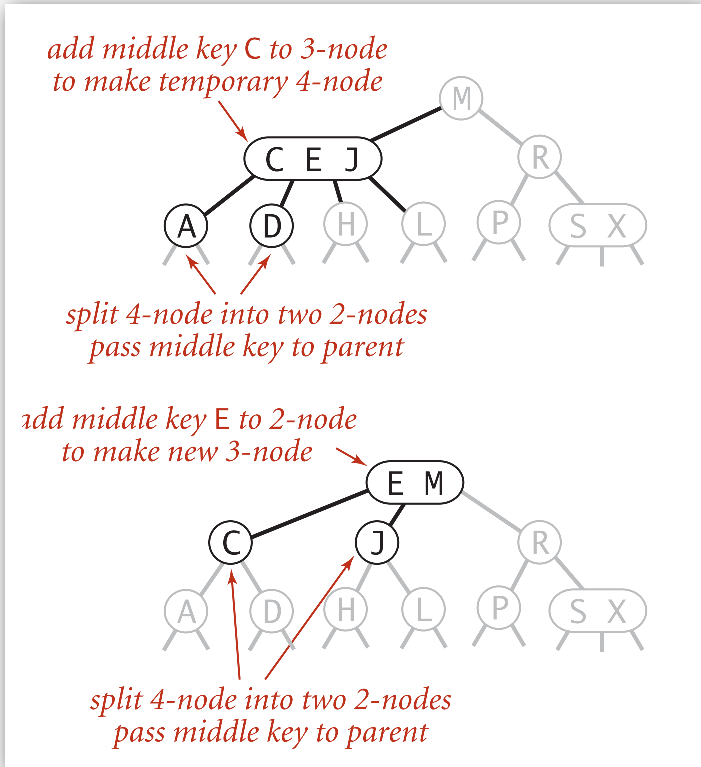


Insertion in a 2-3 tree

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.

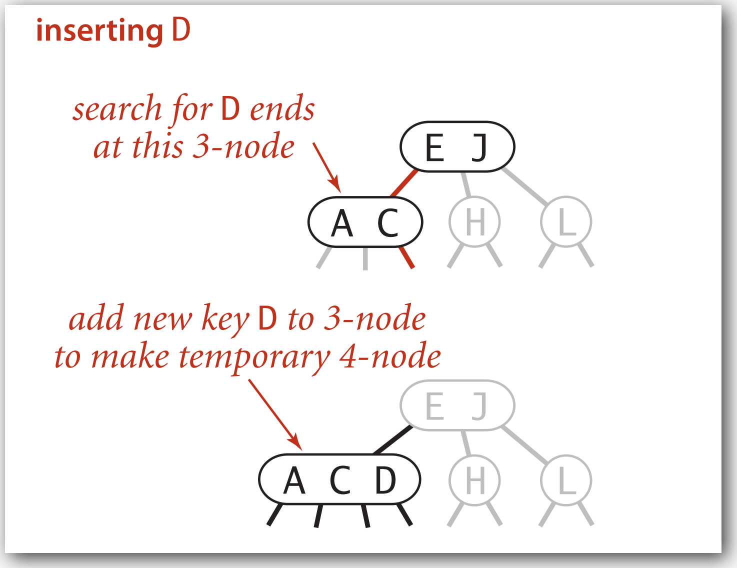


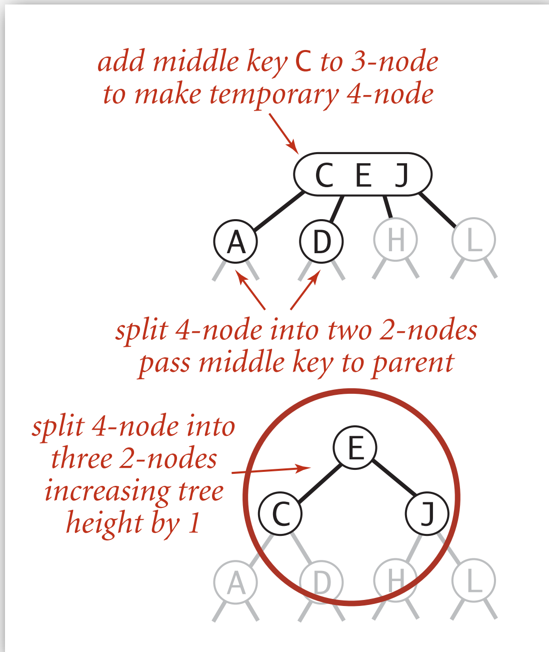


Insertion in a 2-3 tree

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.

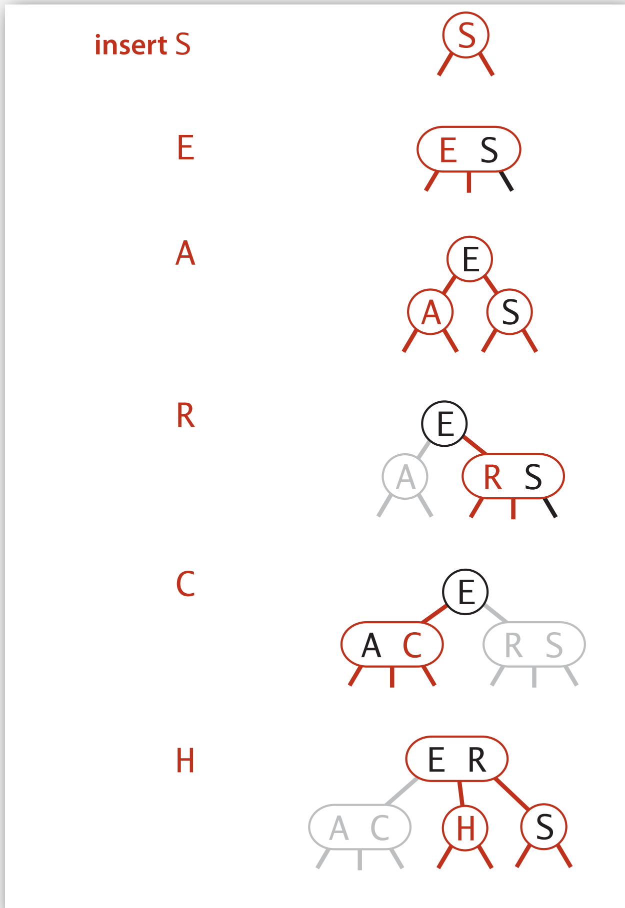


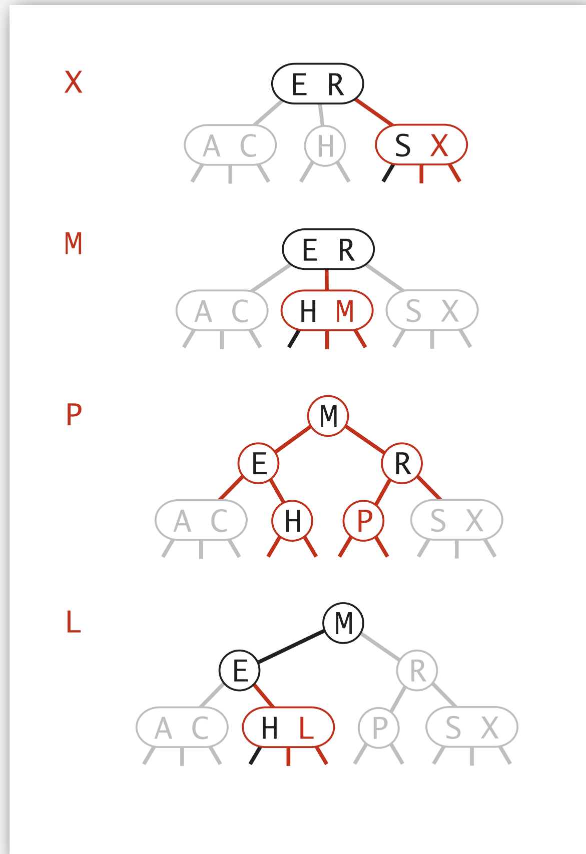


Remark. Splitting the root increases height by 1.

2-3 tree construction trace

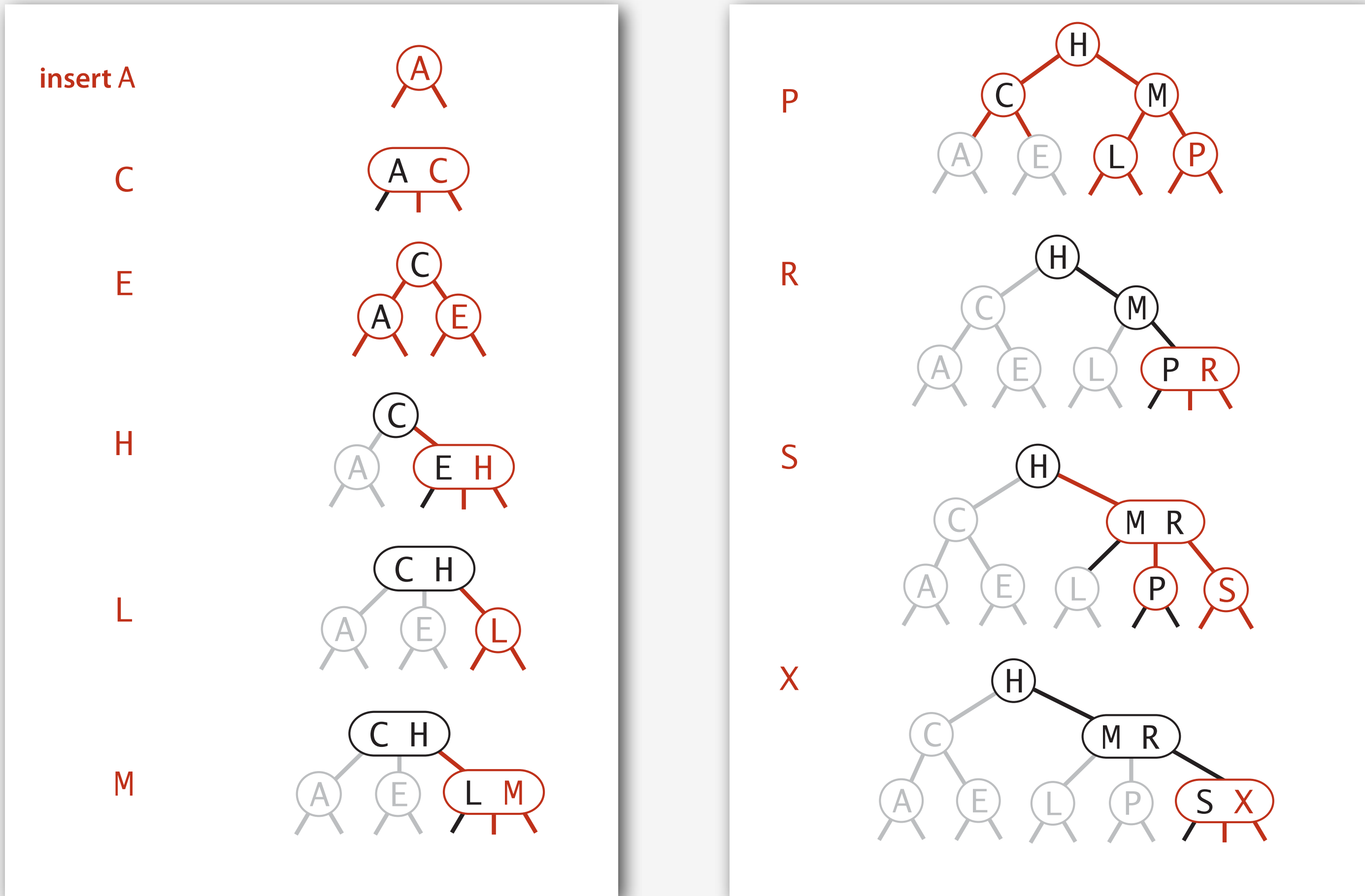
Standard indexing client.





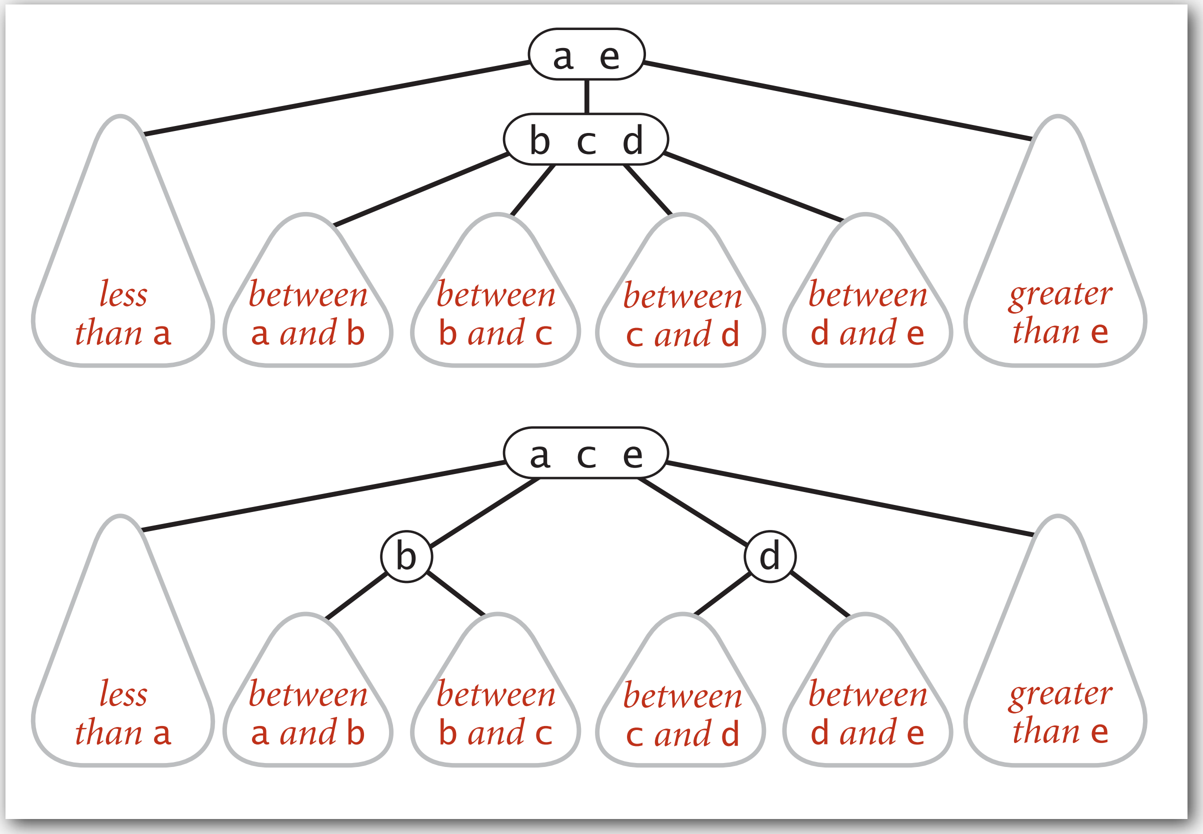
2-3 tree construction trace

The same keys inserted in ascending order.



Local transformations in a 2-3 tree

Splitting a 4-node is a local transformation: constant number of steps.

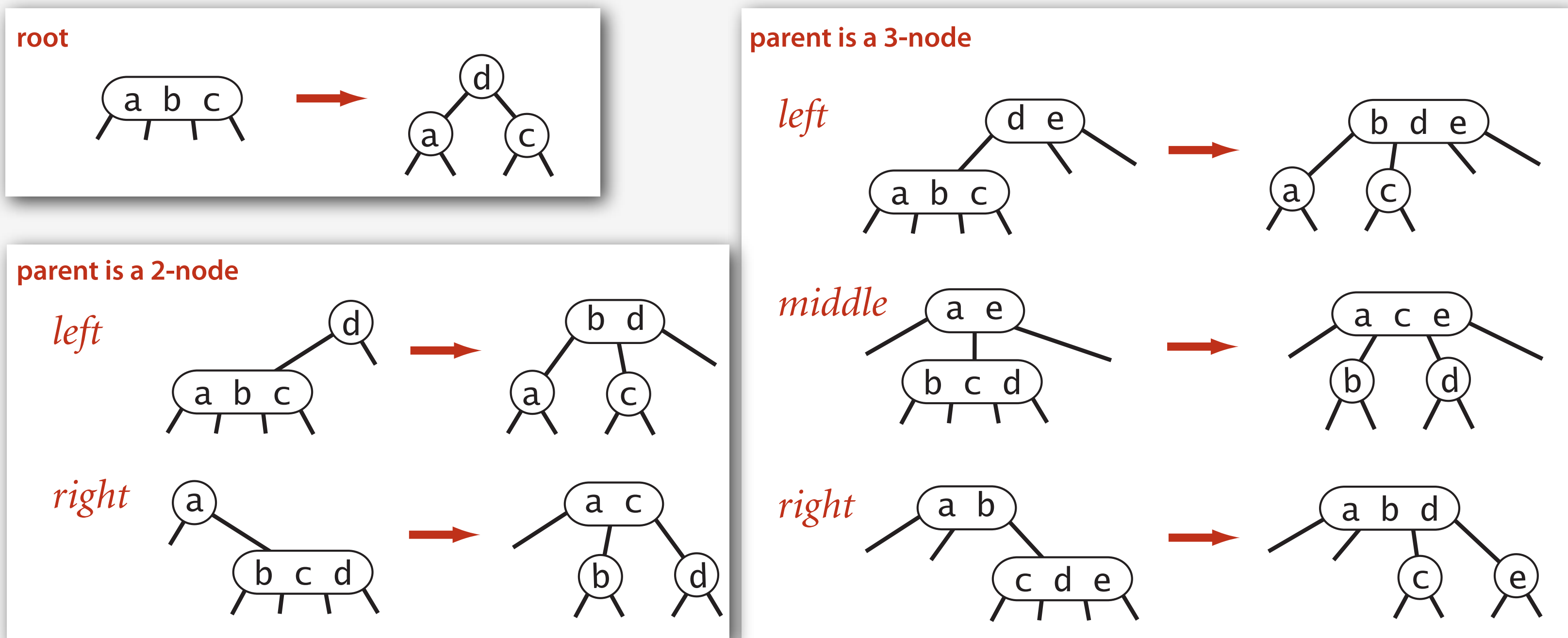


Global properties in a 2-3 tree

Invariant. Symmetric order.

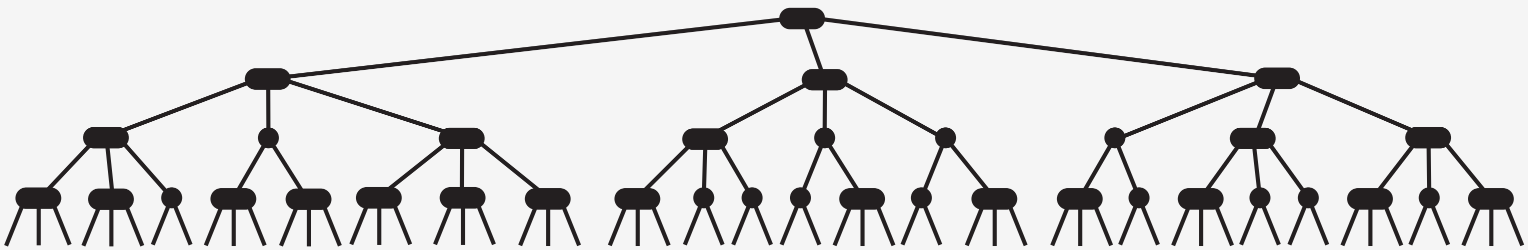
Invariant. Perfect balance.

Pf. Each transformation maintains order and balance.



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

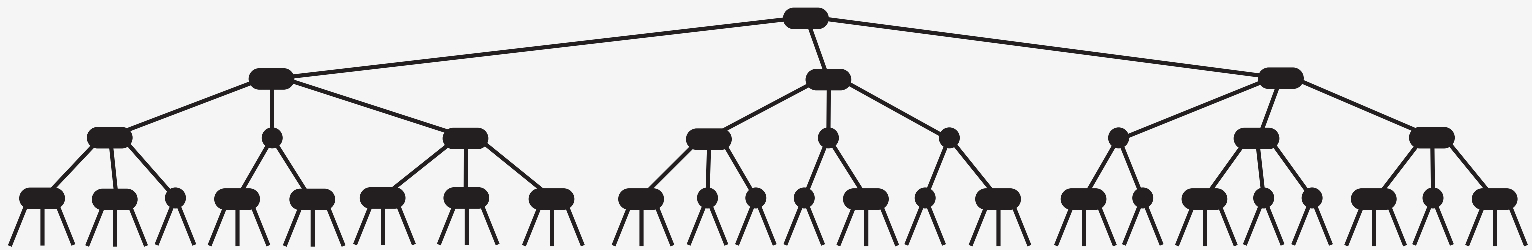


Tree height.

- 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	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	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`

constants depend upon implementation

2-3 tree: implementation?

Direct implementation is complicated, because:

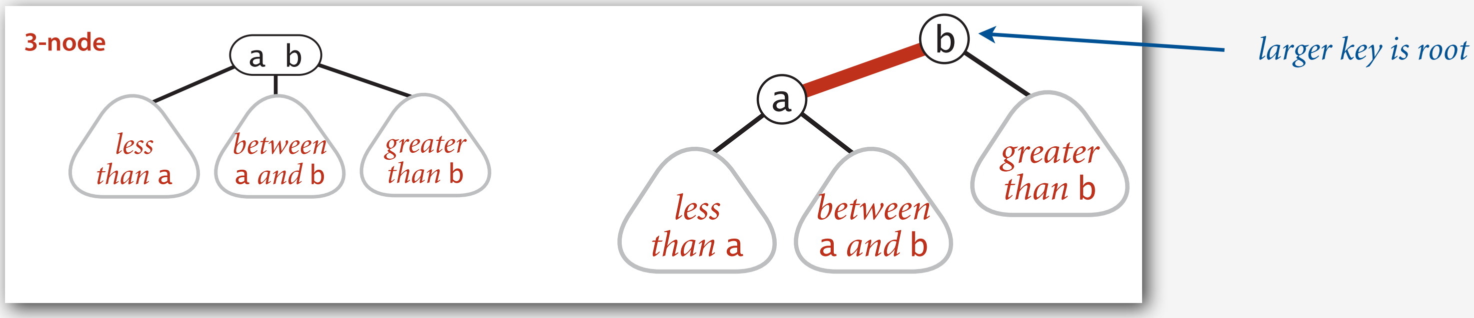
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

Bottom line. Could do it, but there's a better way.

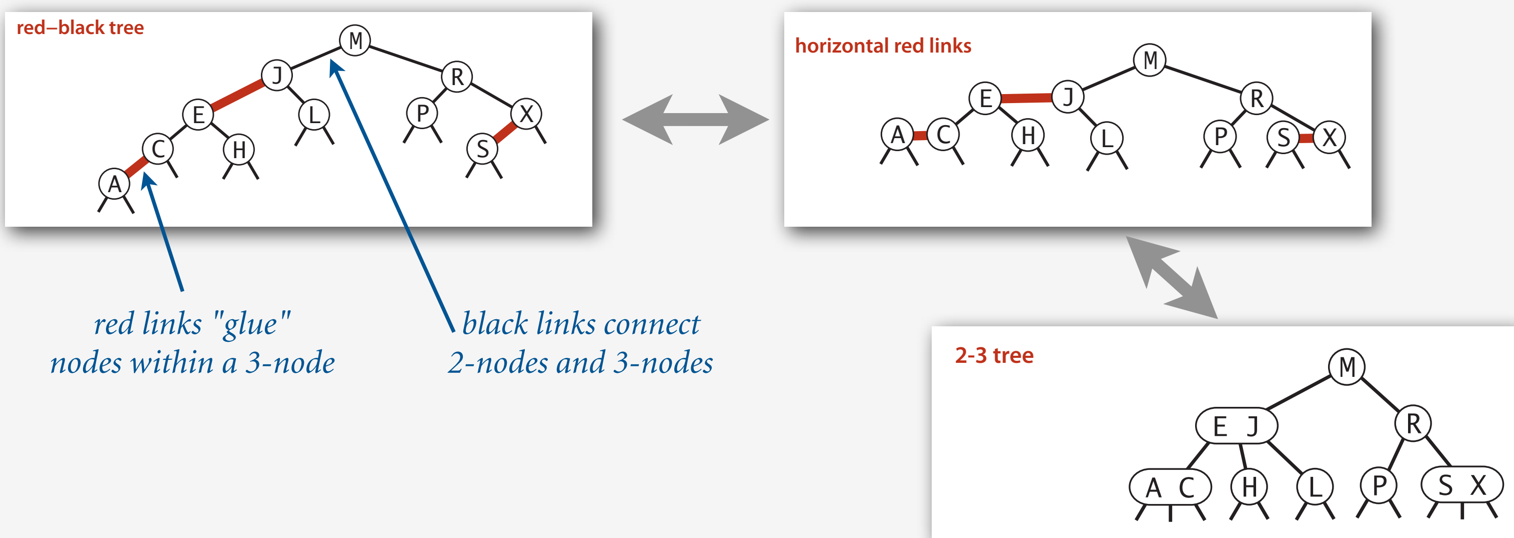
- 2-3-4 trees
- red-black trees
- B-trees

Left-leaning red-black trees (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.



Key property. 1–1 correspondence between 2–3 and LLRB.

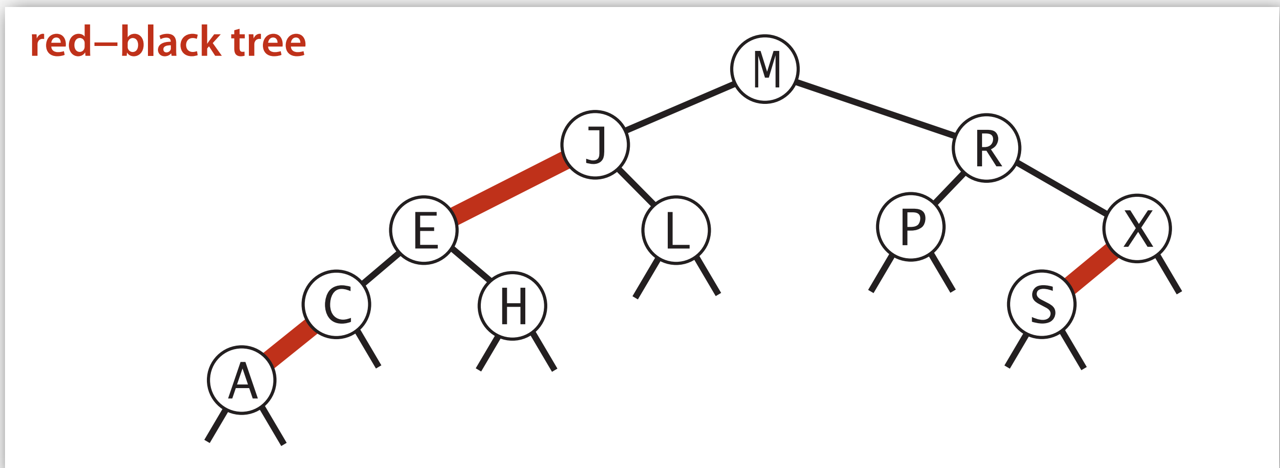


An equivalent definition

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.
- Red links lean left.

"perfect black balance"

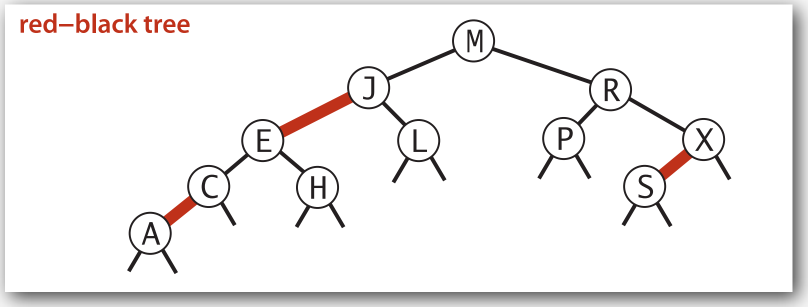


Search implementation for red-black trees

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;
}
```



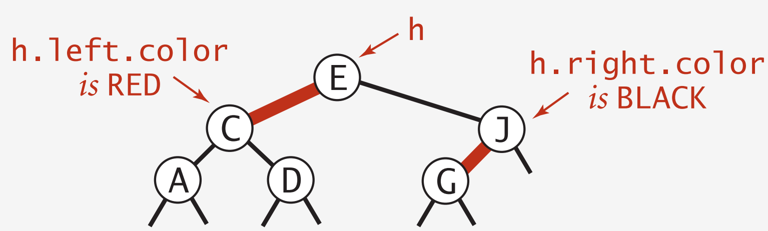
Remark. Many other ops (e.g., ceiling, selection, iteration) are also identical.

Red-black tree representation

Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

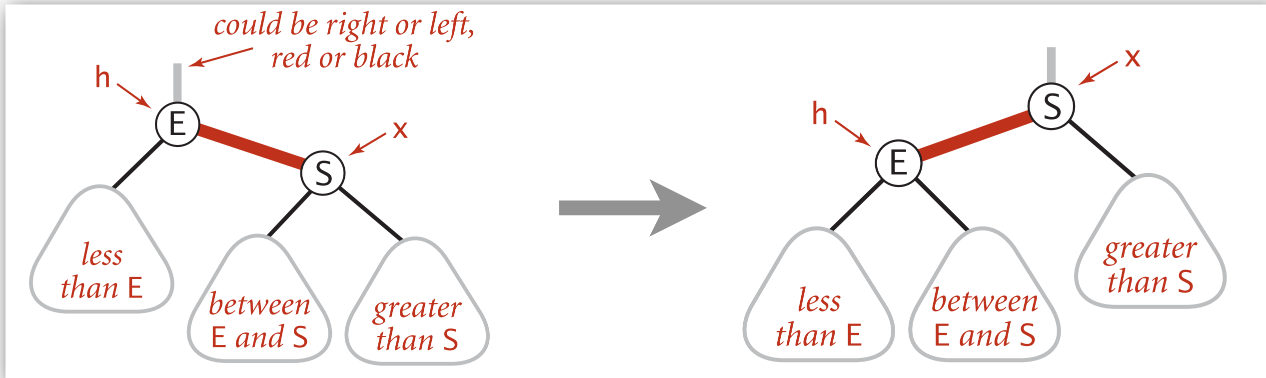
```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

null links are black



Elementary red-black tree operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

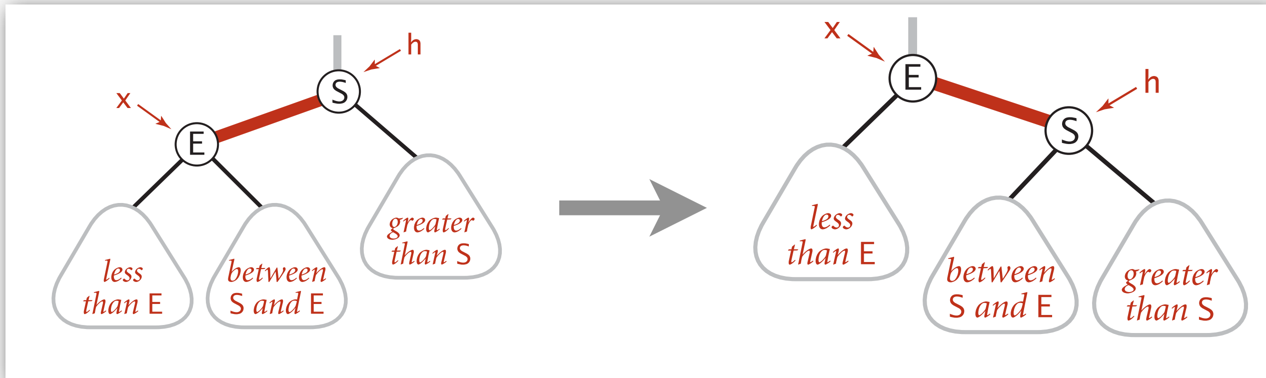



```
private Node rotateLeft(Node h)
{
   x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black tree operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

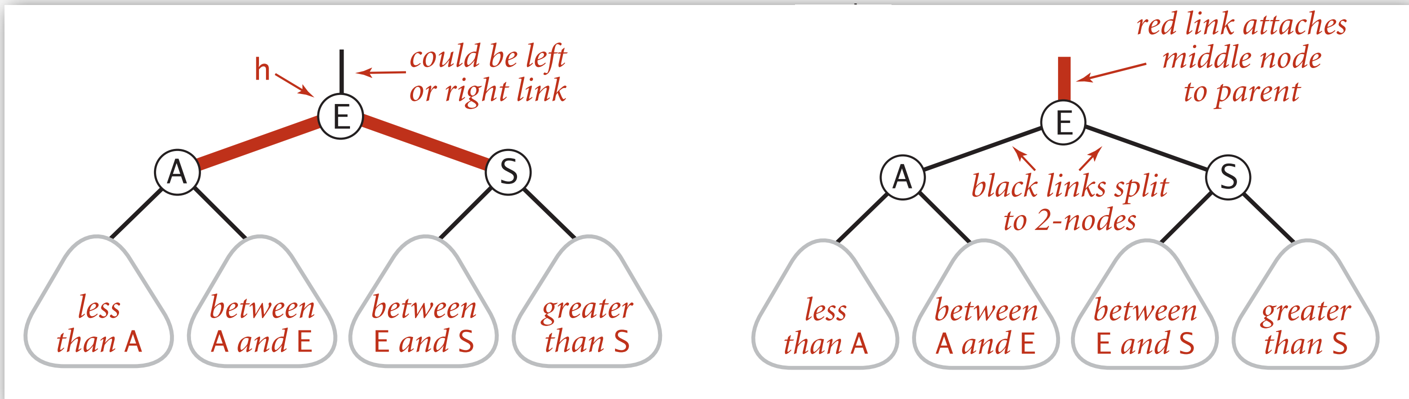



```
private Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black tree operations

Color flip. Recolor to split a (temporary) 4-node.

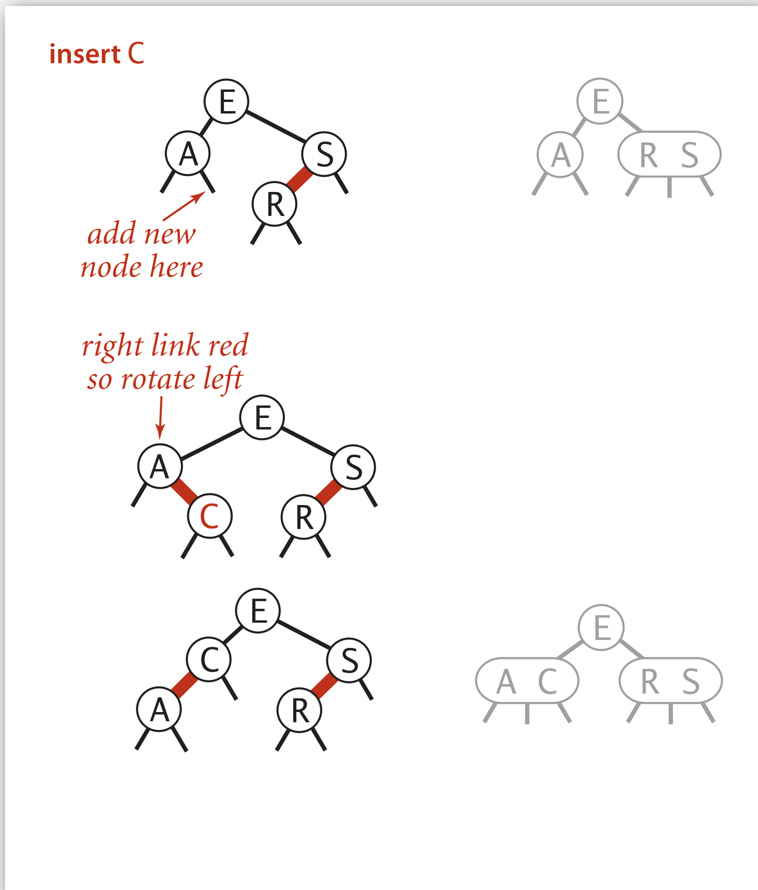



```
private void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

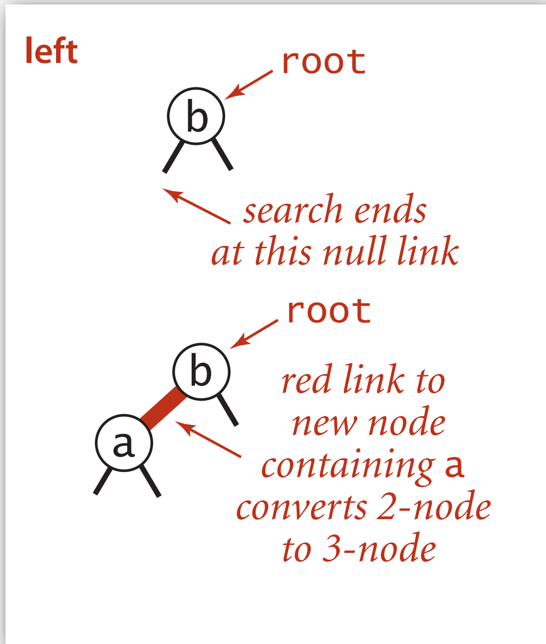
Insertion in a LLRB tree: overview

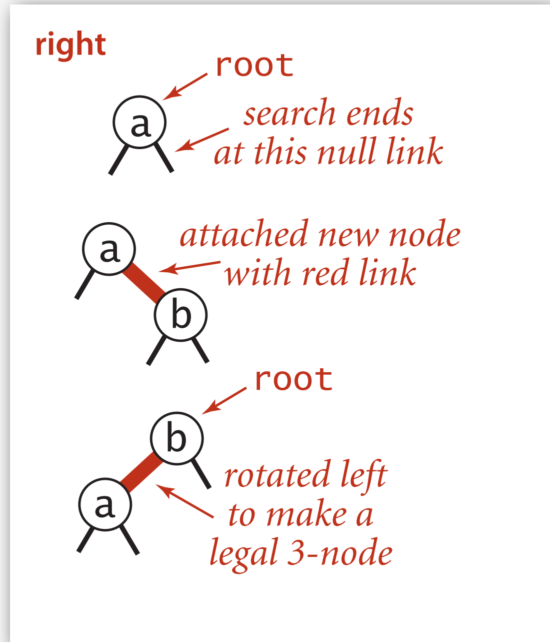
Basic strategy. Maintain 1-1 correspondence with 2-3 trees by applying elementary red-black tree operations



Insertion in a LLRB tree

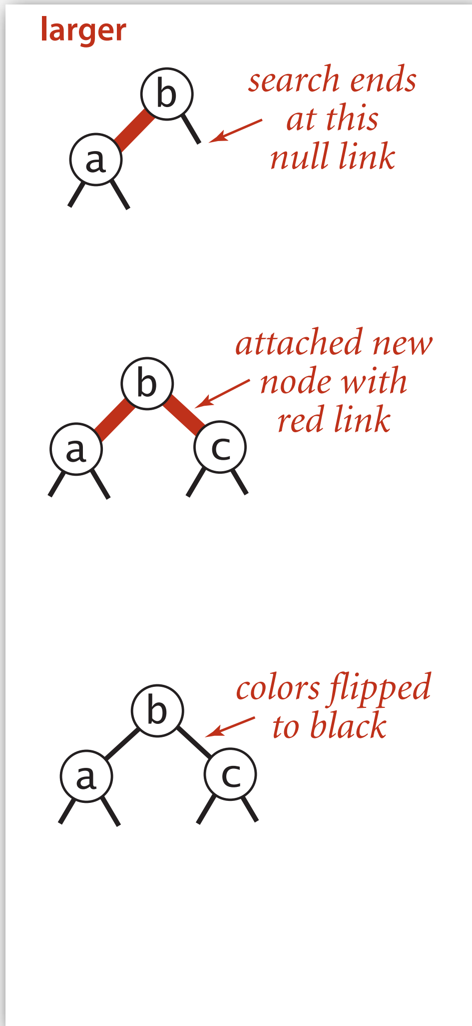
Warmup 1. Insert into a tree with exactly 1 node.

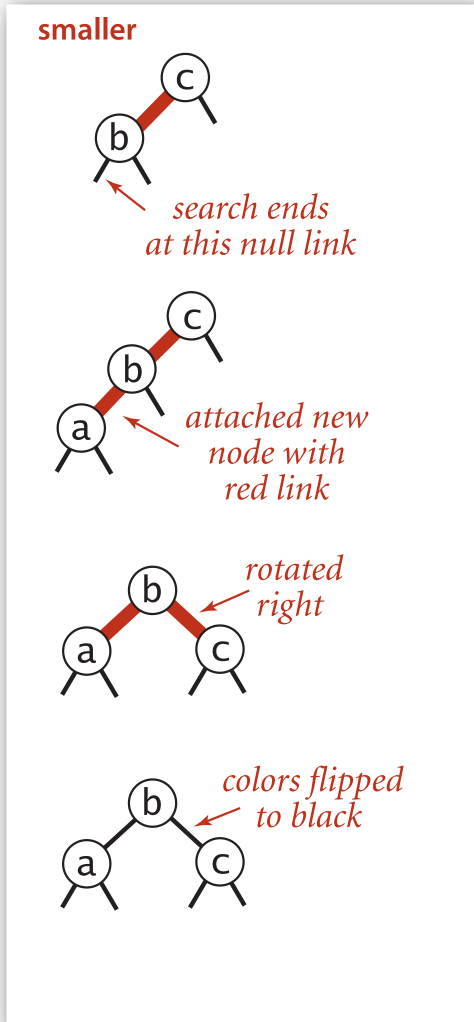


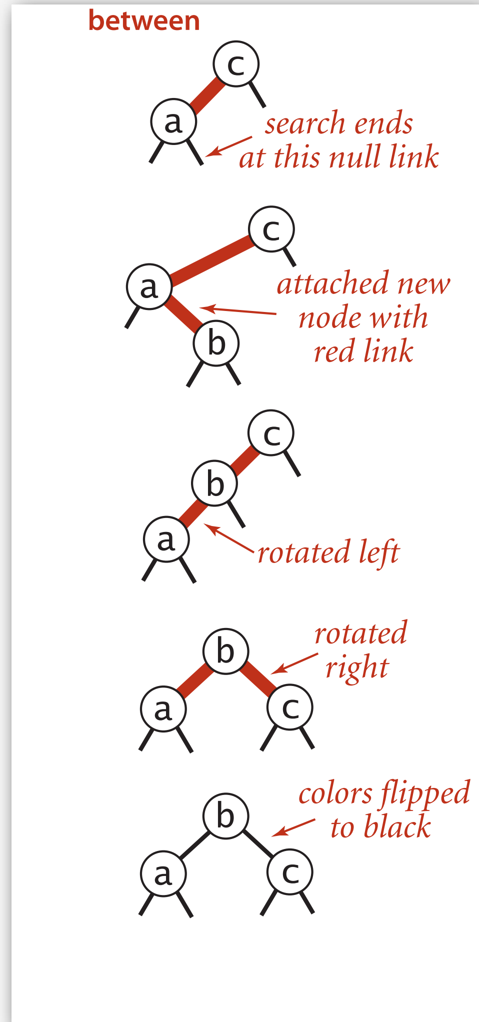


Insertion in a LLRB tree

Warmup 2. Insert into a tree with exactly 2 nodes.



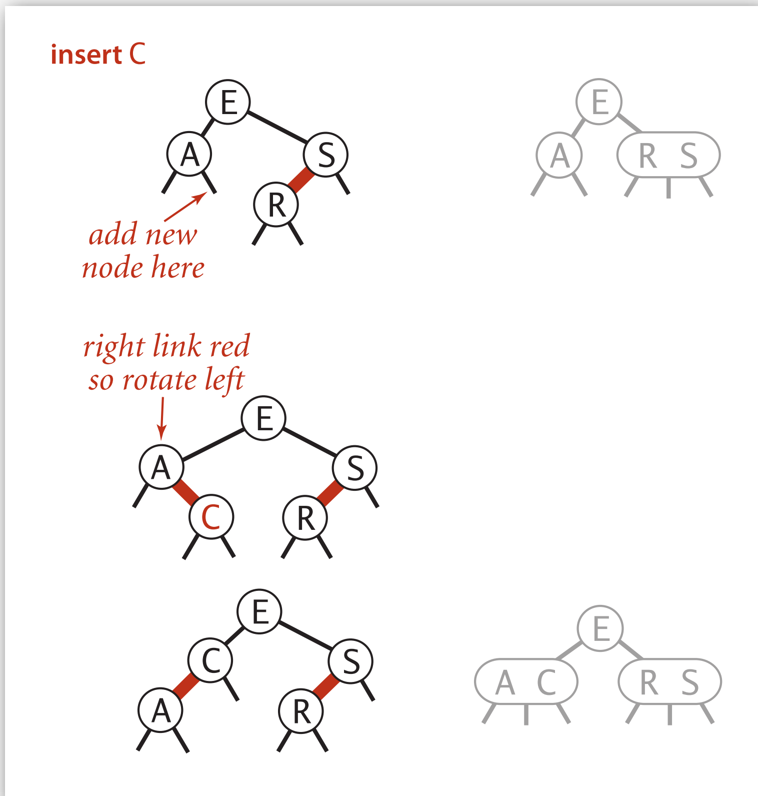




Insertion in a LLRB tree

Case 1. Insert into a 2-node at the bottom.

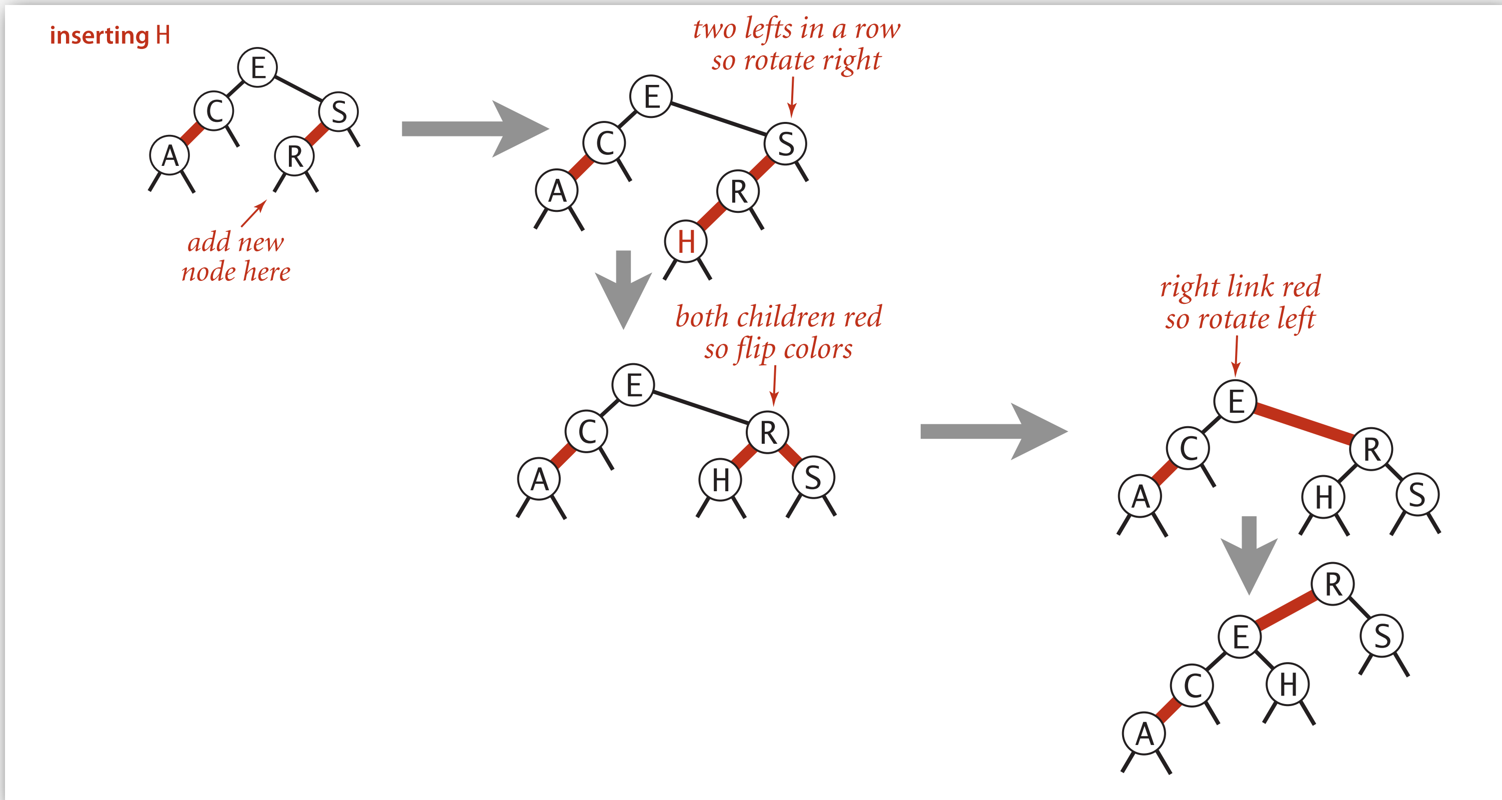
- Do standard BST insert; color new link red.
- If new red link is a right link, rotate left.



Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

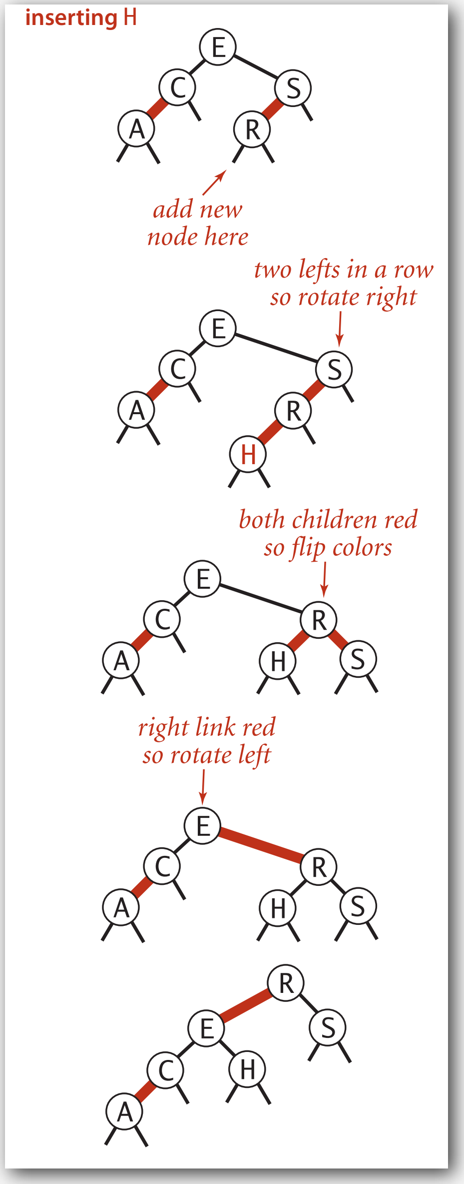
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).



Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

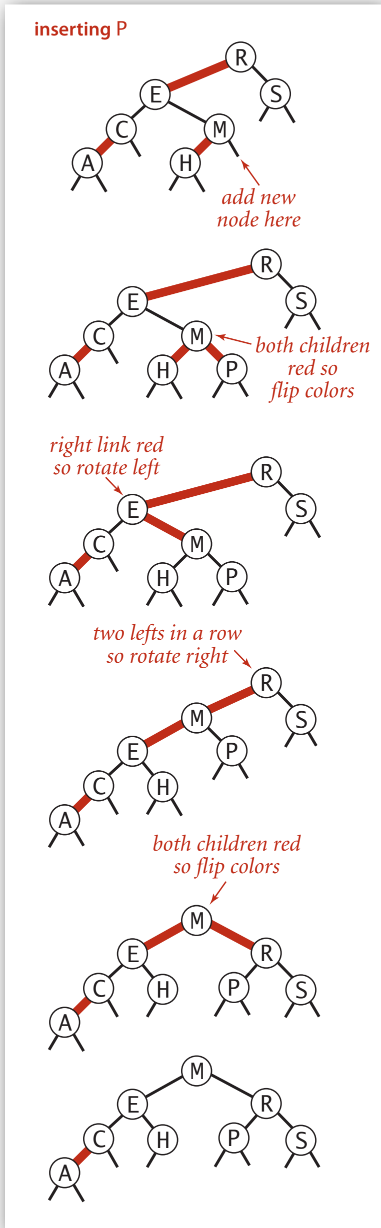
- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).



Insertion in a LLRB tree: passing red links up the tree

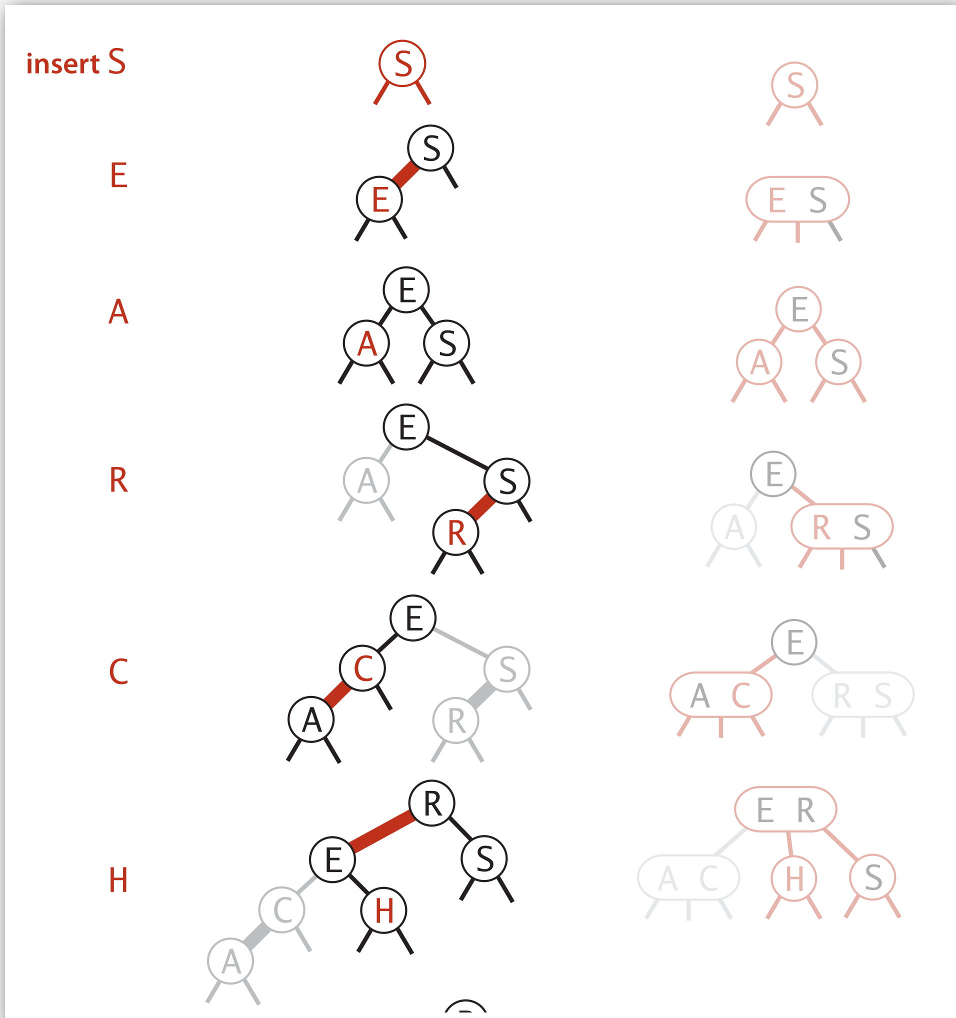
Case 2. Insert into a 3-node at the bottom.

- 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).



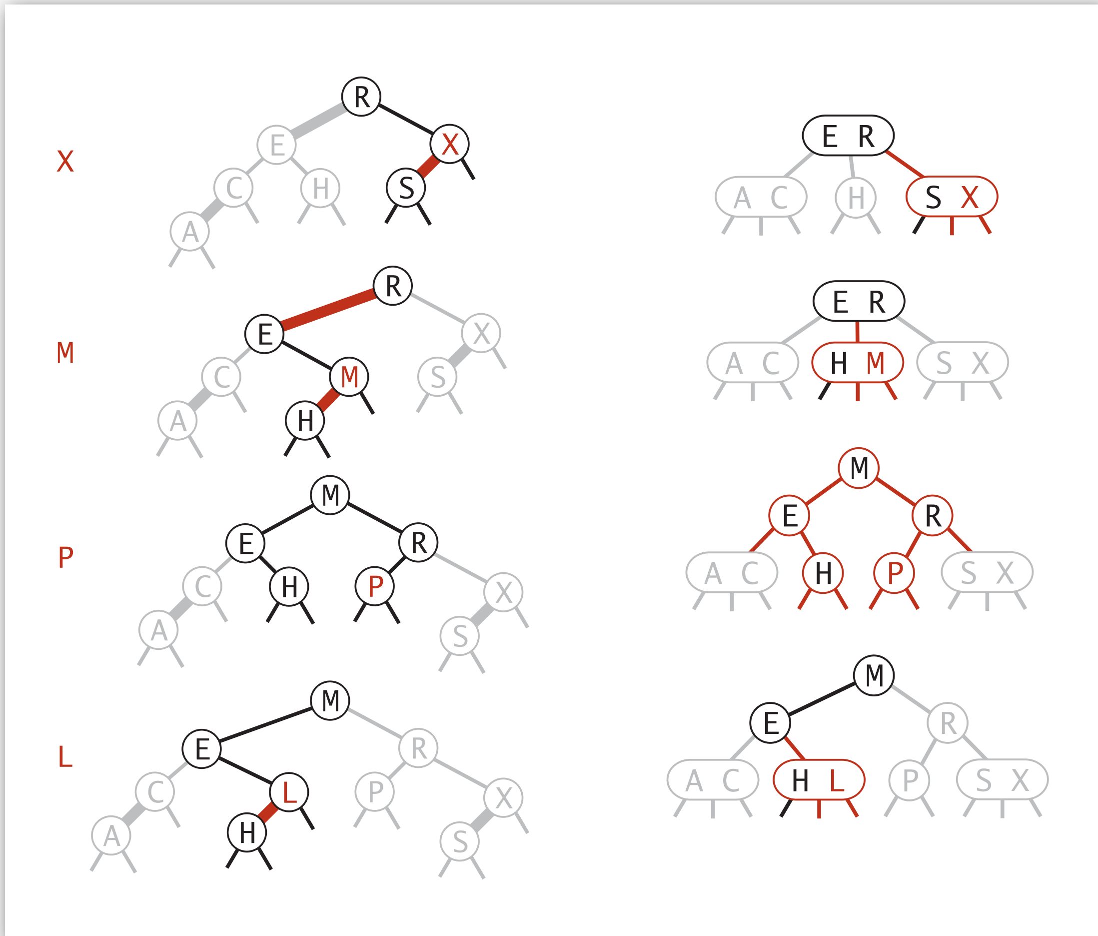
LLRB tree construction trace

Standard indexing client.



LLRB tree construction trace

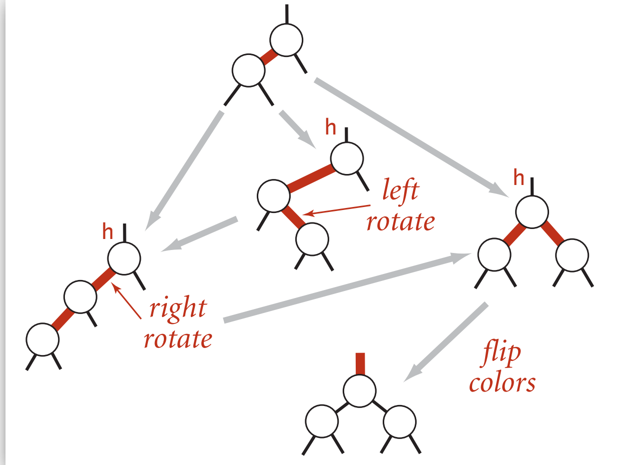
Standard indexing client (continued).



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 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))     h = flipColors(h);

   return h;
}
```

insert at bottom

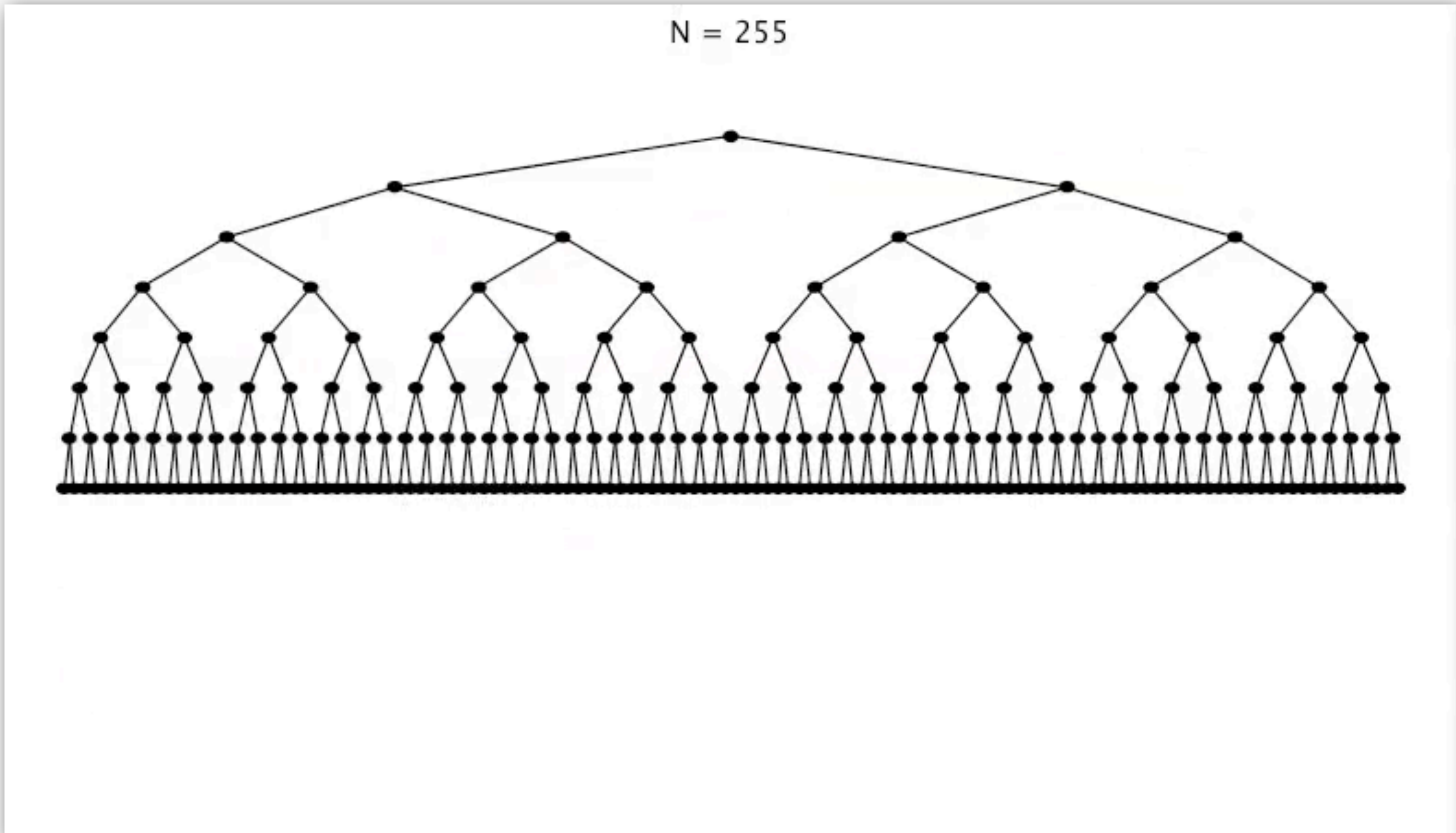
lean left

balance 4-node

split 4-node

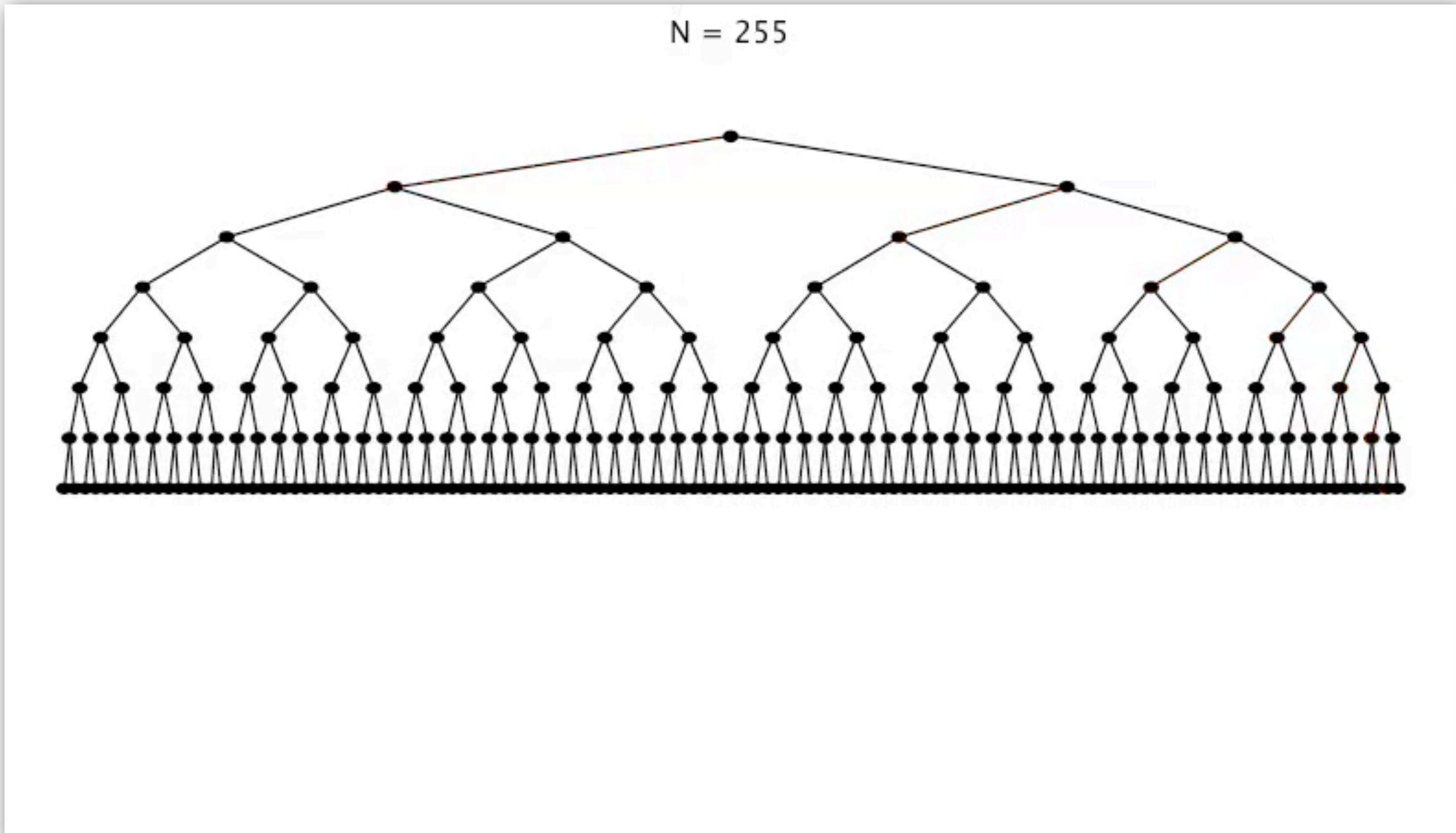
only a few extra lines of code
to provide near-perfect balance

Insertion in a LLRB tree: visualization



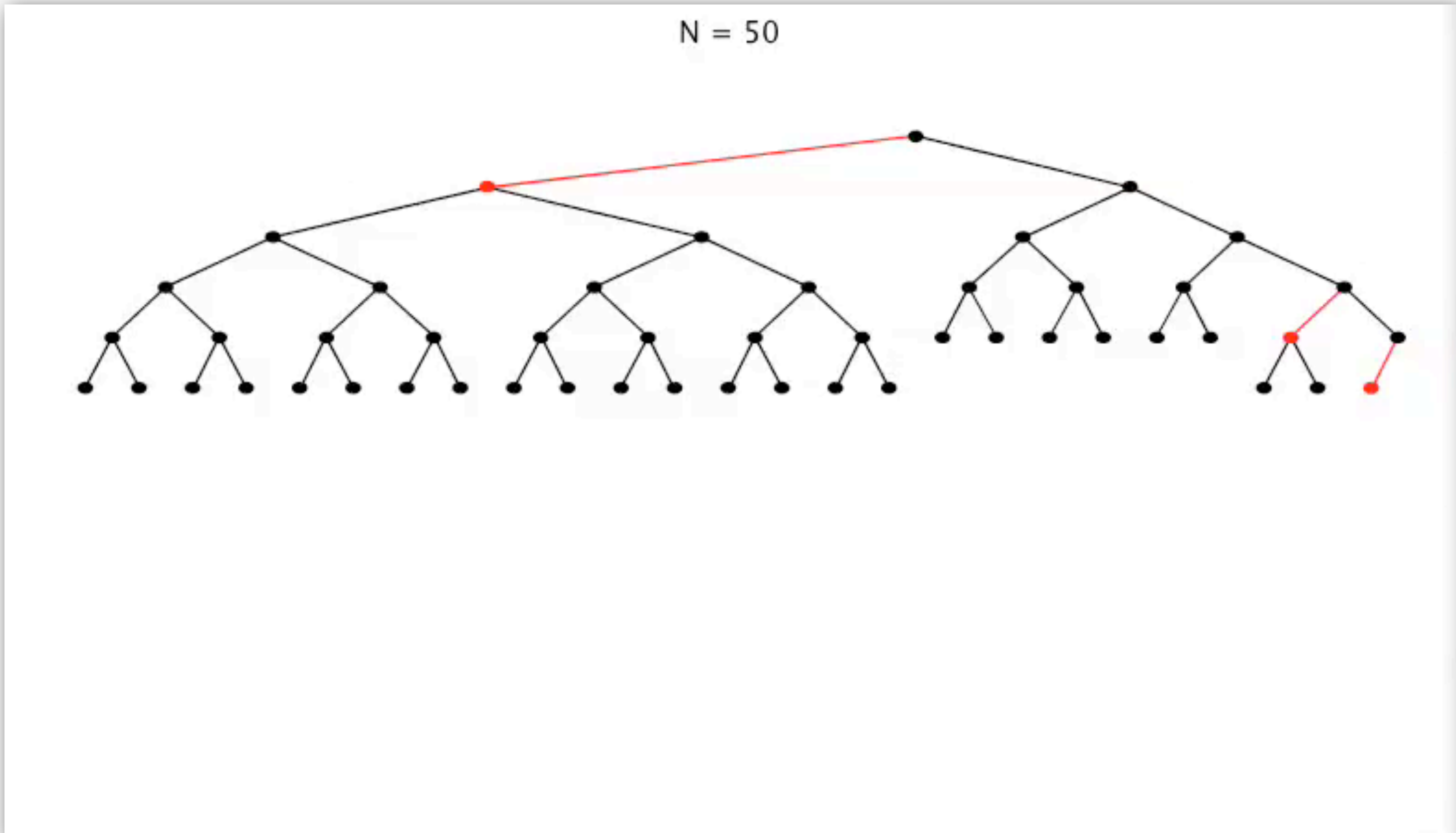
255 insertions in ascending order

Insertion in a LLRB tree: visualization



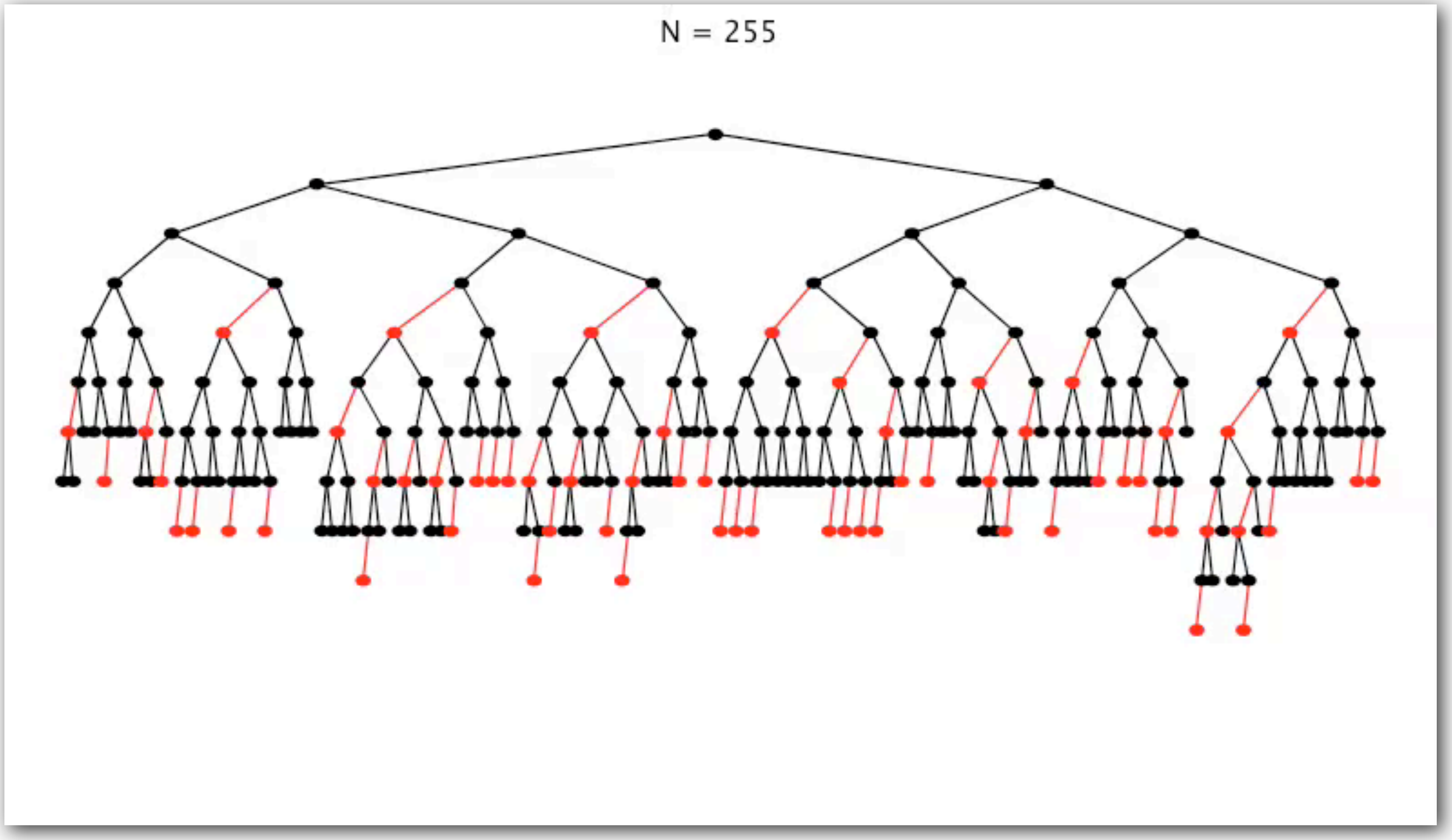
255 insertions in descending order

Insertion in a LLRB tree: visualization



50 random insertions

Insertion in a LLRB tree: visualization



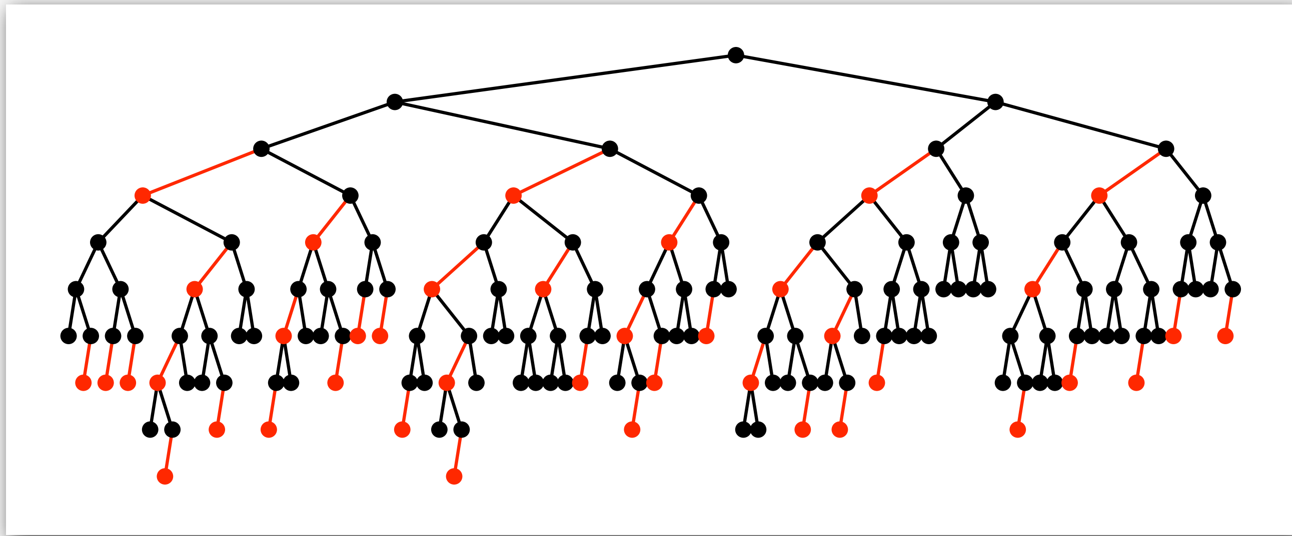
255 random insertions

Balance in LLRB trees

Proposition. Height of tree is ≤ 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 ~ 1.00 lg N in typical applications.

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	?	yes	`compareTo()`
2-3 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`
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	`compareTo()`

* exact value of coefficient unknown but extremely close to 1

Costs for `java FrequencyCounter 8 < tale.txt` **using** `RedBlackBST`

Why left-leaning trees?

old code (that students had to learn in the past)

```
private Node put(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp < 0)
   {
      x.left = put(x.left, key, val, false);
      if (isRed(x) && isRed(x.left) && sw)
         x = rotateRight(x);
      if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotateRight(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else if (cmp > 0)
   {
      x.right = put(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotateLeft(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotateLeft(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   else x.val = val;
   return x;
}
```

extremely tricky

new code (that you have to learn)

```
public Node put(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);
   int cmp = kery.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 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))
      h = flipColors(h);

  return h;
}
```

straightforward
(if you've paid attention)

Why left-leaning trees?

Simplified code.

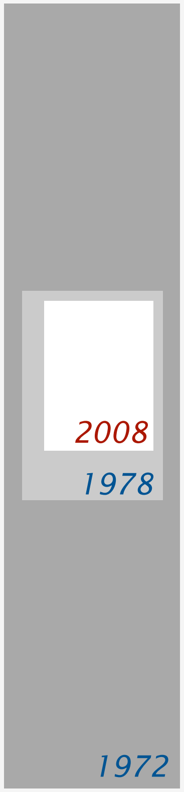
- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.

- Delete min/max.
- Arbitrary delete.

Improves widely-used algorithms.

- AVL trees, 2-3 trees, 2-3-4 trees.
- Red-black trees.



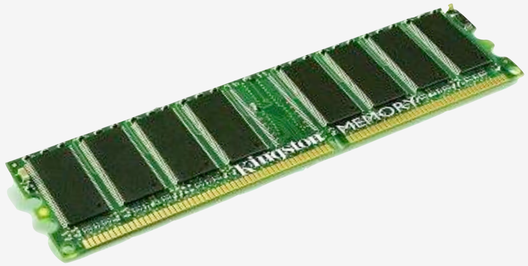
Bottom line. Left-leaning red-black trees are the simplest balanced BST to implement and the fastest in practice.

- 2-3-4 trees
- red-black trees
- **B-trees**

File system model

Page. Contiguous block of data (e.g., a file or 4096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).



Model. Time required for a probe is much larger than time to accessdata within a page.

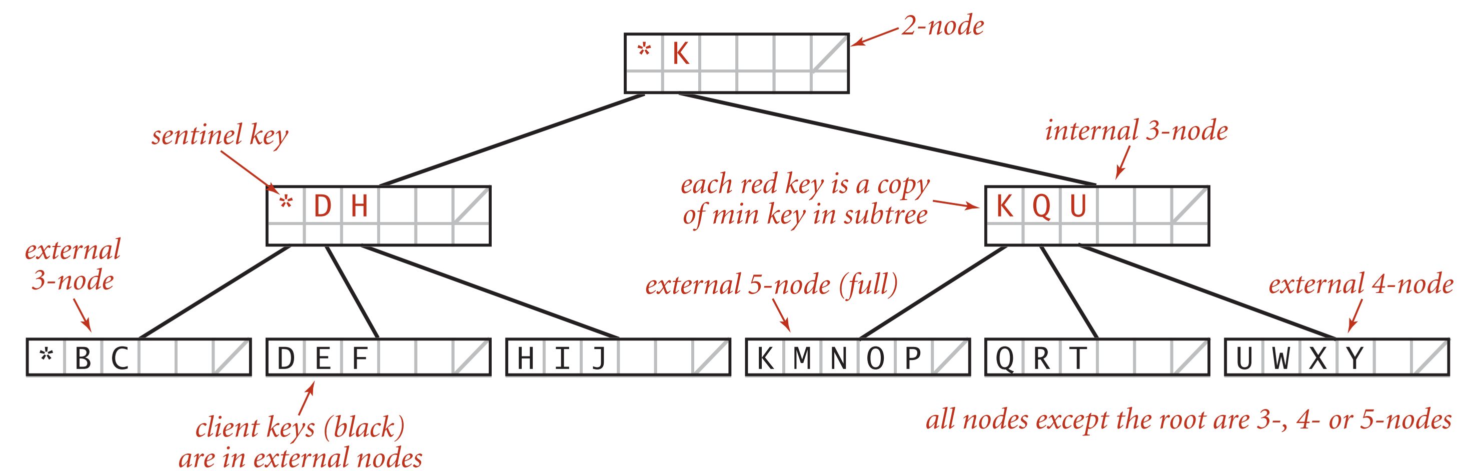
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to M links per node.

- At least 1 entry at root.
- At least M/2 links in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

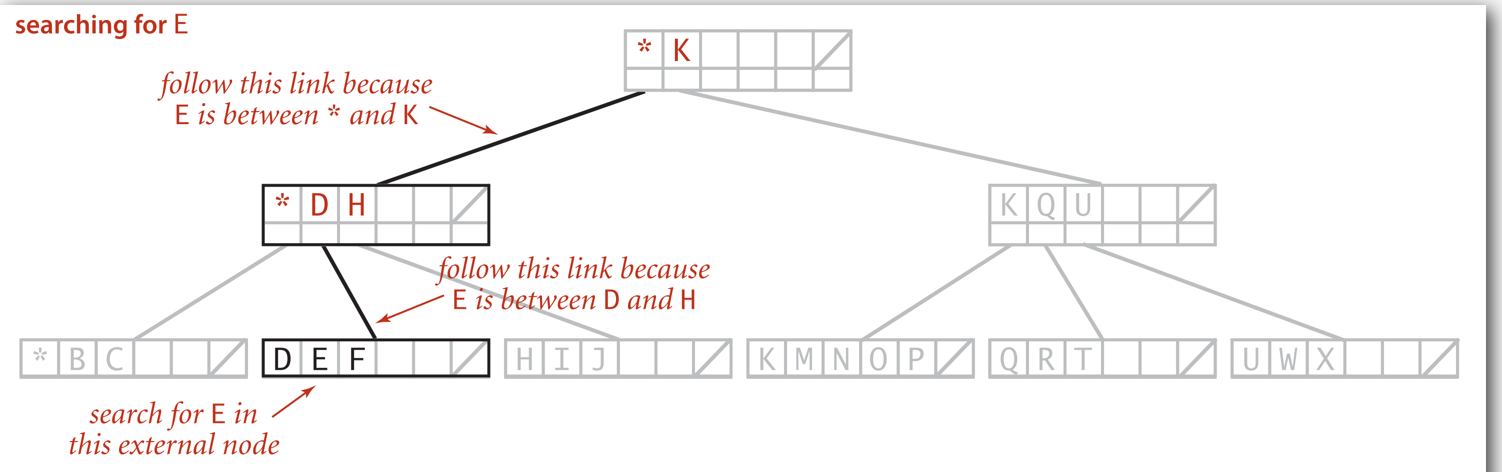
choose M as large as possible so that M links fit in a page, e.g., M = 1000



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

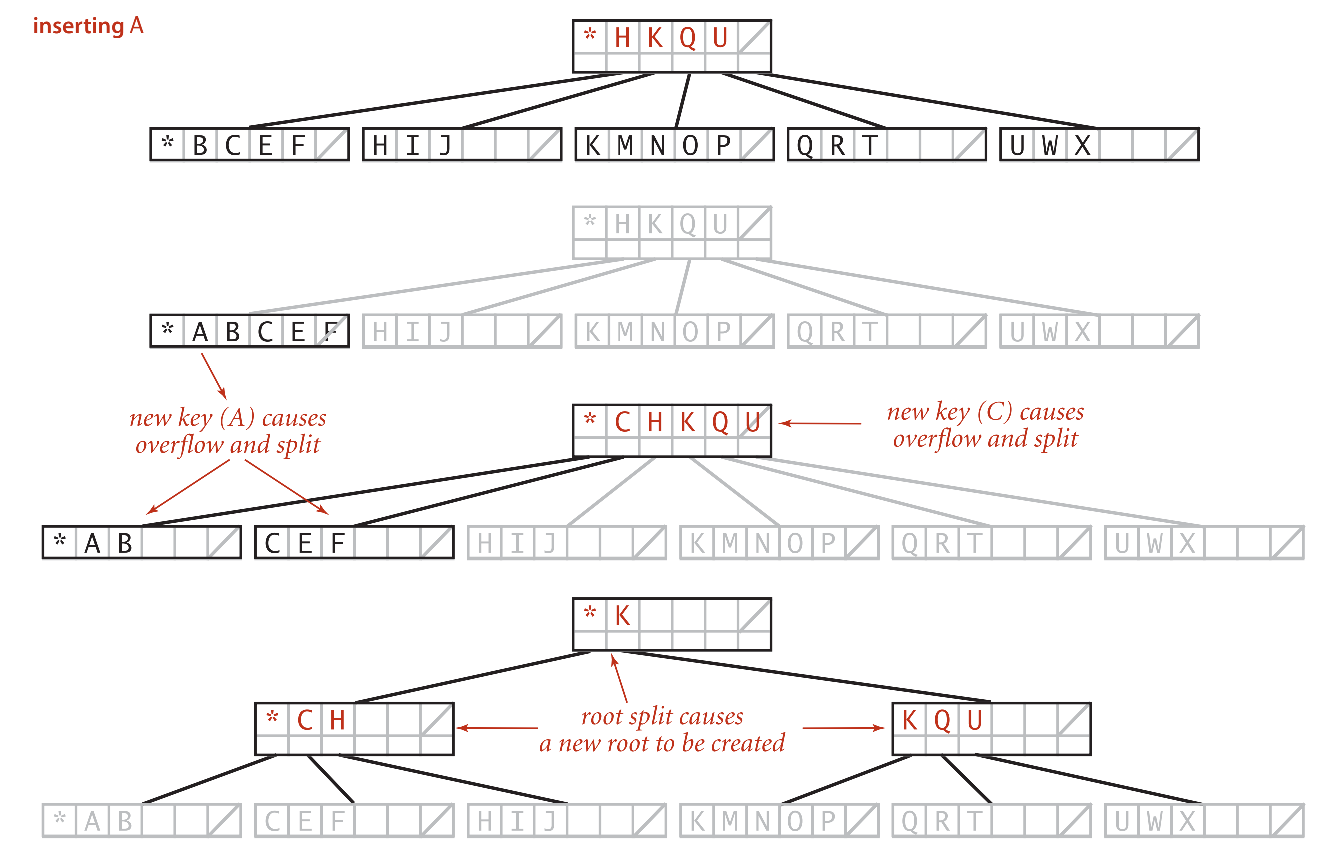
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split (M+1)-nodes on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree

Probes. A search or insert in a B-tree of order M with N items requires between $\log_M N$ and $\log_{M/2} N$ probes.

Pf. All internal nodes (besides root) have between M/2 and M links.

In practice. Number of probes is at most 4! ← M = 1000; N = 62 billion $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Balanced trees in the wild

Red-black trees are widely used as system symbol tables.

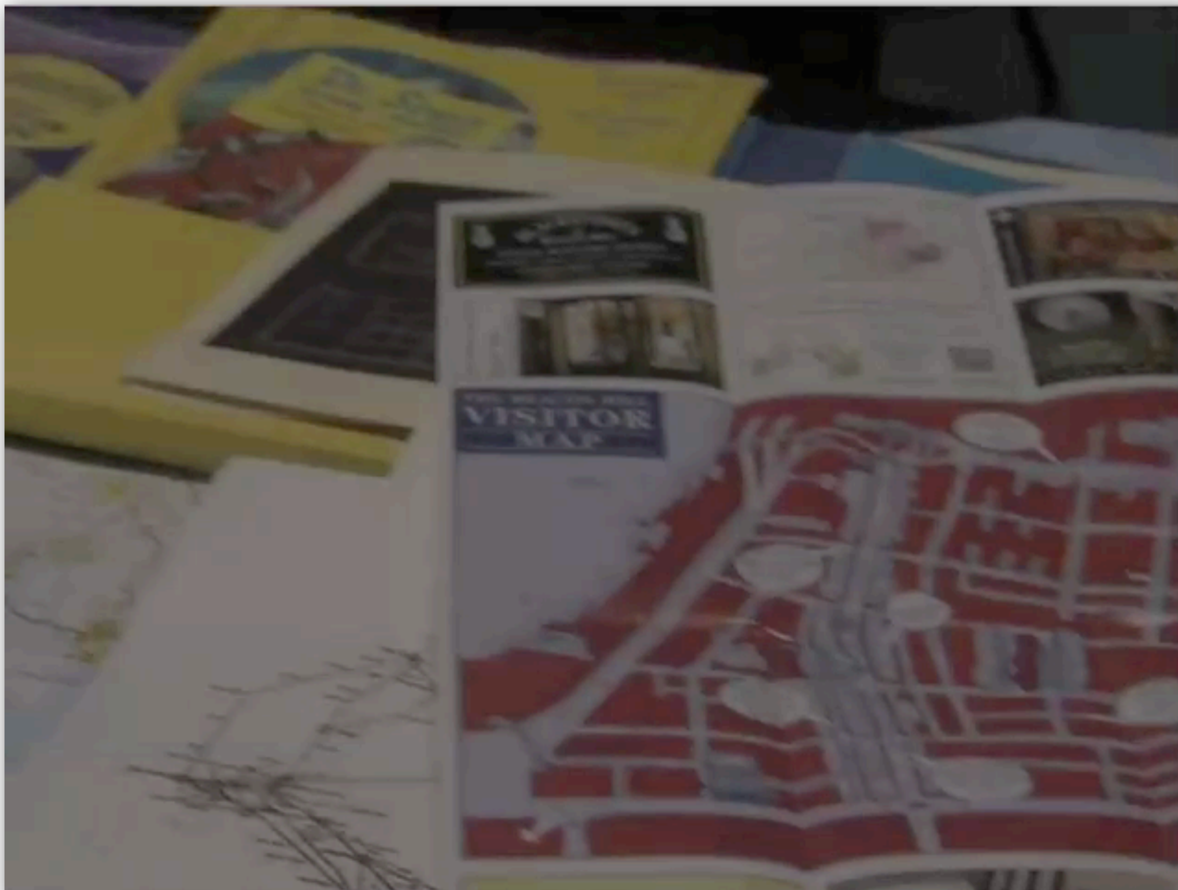
- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.

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.

Red-black trees in the wild



*Common sense. Sixth sense.
Together they're the
FBI's newest team.*

Red-black trees in the wild

ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

Nicole is tapping away at a computer keyboard. She finds something.

Hashing

References:

Algorithms in Java, Chapter 14

`http://www.cs.princeton.edu/algs4/44hash`

- hash functions
- separate chaining
- linear probing
- applications

Optimize judiciously

" More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity. " — William A. Wulf

" We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. " — Donald E. Knuth

" We follow two rules in the matter of optimization:
Rule 1: Don't do it.
Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution. " — M. A. Jackson

Reference: Effective Java by Joshua Bloch

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	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
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	`compareTo()`

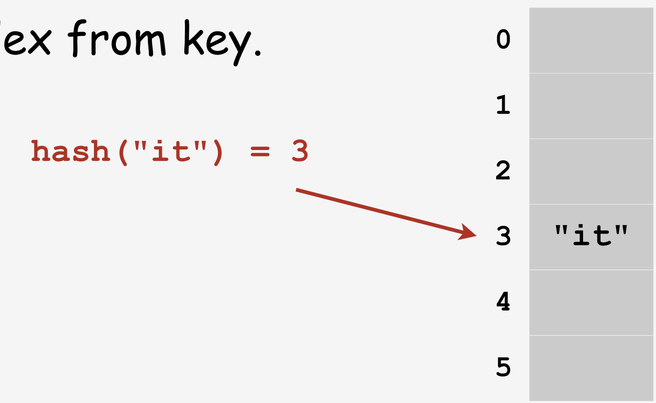
Q. Can we do better?

A. Yes, but with different access to the data.

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.



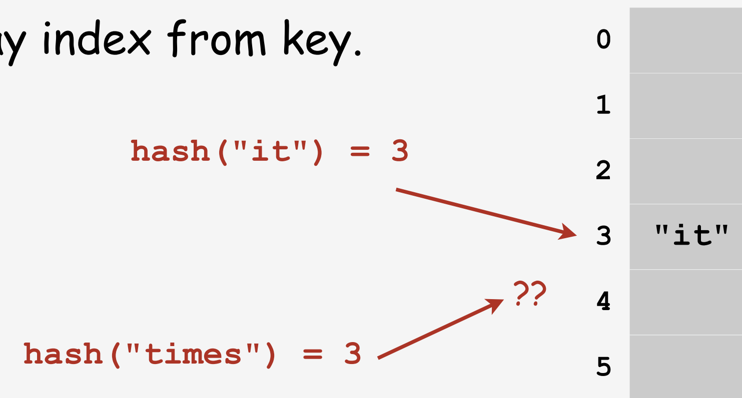
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

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.
- No time limitation: trivial collision resolution with sequential search.
- Limitations on both time and space: hashing (the real world).

- hash functions
- separate chaining
- linear probing
- applications

Equality test

Needed because hash methods do not use `CompareTo()`.

All Java classes have a method `equals()`, inherited from `Object`.

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`.

do `x` and `y` refer to the same object?

Default implementation (inherited from `Object`). `(x == y)`

Customized implementations. `Integer`, `Double`, `String`, `URI`, `Date`, …

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy

```
public       class Record
{
   private final String name;
   private final int id;
   private final double value;
   ...

   public boolean equals(Record y)
   {






      Record that =          y;
      return (this.id == that.id) &&
             (this.value == that.value) &&
             (this.equals(that.name));
   }
}
```

check that all significant fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

no safe way to use **`equals()`** with inheritance

```
public final class Record
{
   private final String name;
   private final int id;
   private final double value;
   ...

   public boolean equals(Object y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      Record that = (Record) y;
      return (this.id == that.id) &&
             (this.value == that.value) &&
             (this.equals(that.name));
   }
}
```

must be **`Object`**.
Why? Experts still debate.

optimize for true object equality

check for **`null`**

objects must be in the same class

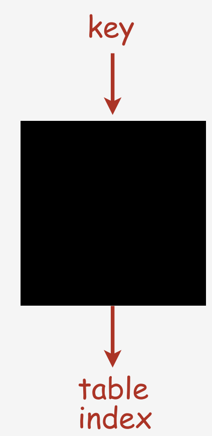
check that all significant
fields are the same

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.

573 = California, 574 = Alaska
(assigned in chronological order within geographic region)

- Bad: first three digits.
- Better: last three digits.

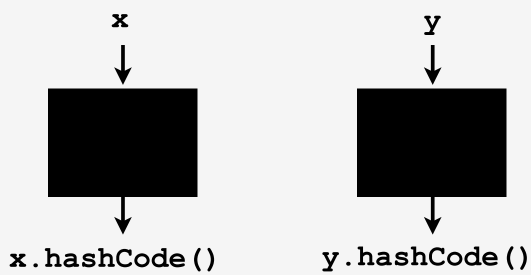
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes have a method `hashCode()`, which returns an `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation (inherited from `Object`). Memory address of `x`.

Customized implementations. `Integer`, `Double`, `String`, `URI`, `Date`, …

User-defined types. Users are on their own.

Implementing hash code: integers and doubles

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
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

```
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	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = 31^{L-1} \cdot s^0 + \ldots + 31^2 \cdot s^{L-3} + 31^1 \cdot s^{L-2} + 31^0 \cdot s^{L-1}$.

Ex.

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

$= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$

A poor hash code

Ex. Strings (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/13loop/index.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

Implementing hash code: user-defined types

```
public final class Record
{
   private String name;
   private int id;
   private double value;

   public Record(String name, int id, double value)
   {  /* as before */  }

   ...

   public boolean equals(Object y)
   {  /* as before */  }

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + name.hashCode();
      hash = 31*hash + id;
      hash = 31*hash + Double.valueOf(value).hashCode();
      return hash;
   }
}
```

nonzero constant

typically a small prime

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 built-in hash code.
- If field is an array, apply to each element.
- If field is an object, apply rule recursively.

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Need a theorem for each type to ensure reliability.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Hash functions

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

Bug.

```
private int hash(Key key)
{  return key.hashCode() % M;  }
```

1-in-a billion bug.

```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

Correct.

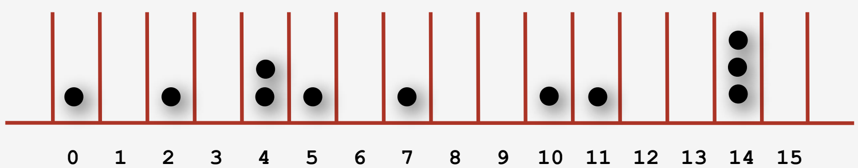
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

- hash functions
- **separate chaining**
- linear probing
- applications

Helpful results from probability theory

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.

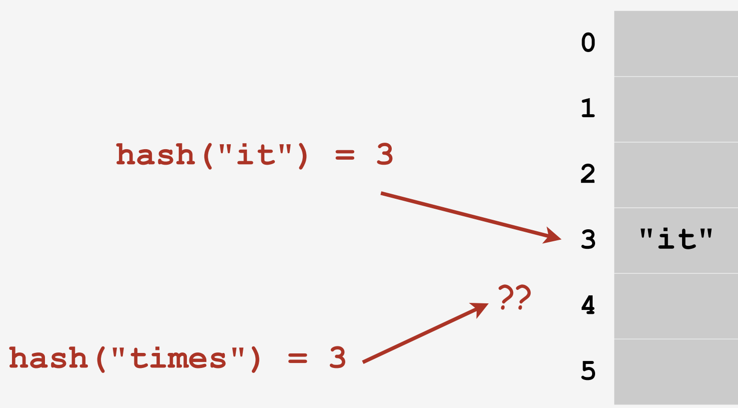
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Collision. Two distinct keys hashing to same index.

- Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount (quadratic) of memory.
- Coupon collector + load balancing ⇒ collisions will be evenly distributed.

Challenge. Deal with collisions efficiently.

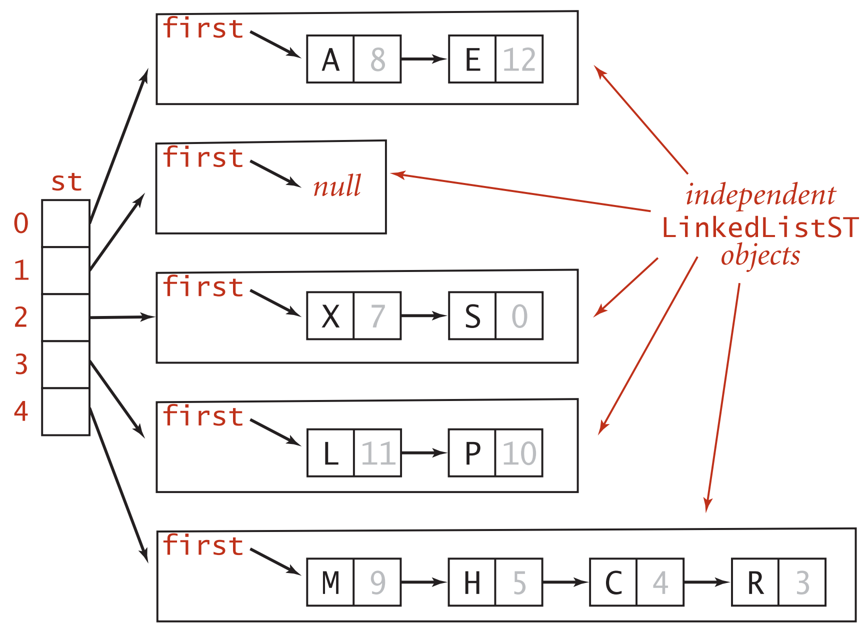


Separate chaining ST

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: only need to search i^{th} chain.

key	hash	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12



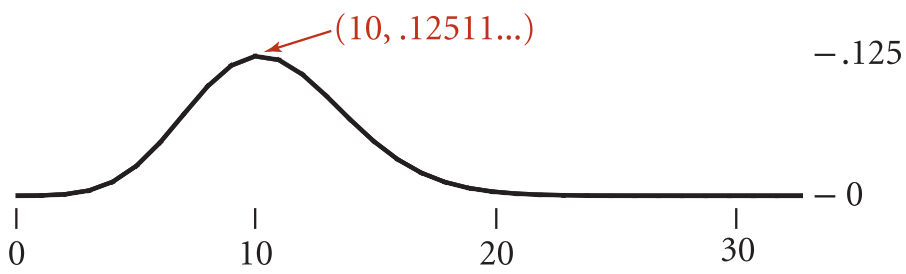
Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int N;                  // number of key-value pairs
   private int M;                  // hash table size
   private LinkedListST[] st;      // array of STs
   public SCHashST()
   {  this(997);  }

   public SCHashST(int M)
   {  // Create M sequential-search-with-linked-list STs.
      this.M = M;
      st = new LinkedListST[M];
      for (int i = 0; i < M; i++)
         st[i] = new LinkedListST();
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M; }

   public Value get(Key key)
   {  return (Value) st[hash(key)].get(key);  }

   public void put(Key key, Value value)
   {  st[hash(key)].put(key, value);  }

   public Iterable<Key> keys()
   {  return st[i].keys());  }

}
```

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.



Binomial distribution $(N = 10^4,\ M = 10^3,\ \alpha = 10)$

Consequence. Number of compares 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 ~ N/5 $\Rightarrow$ constant-time ops.

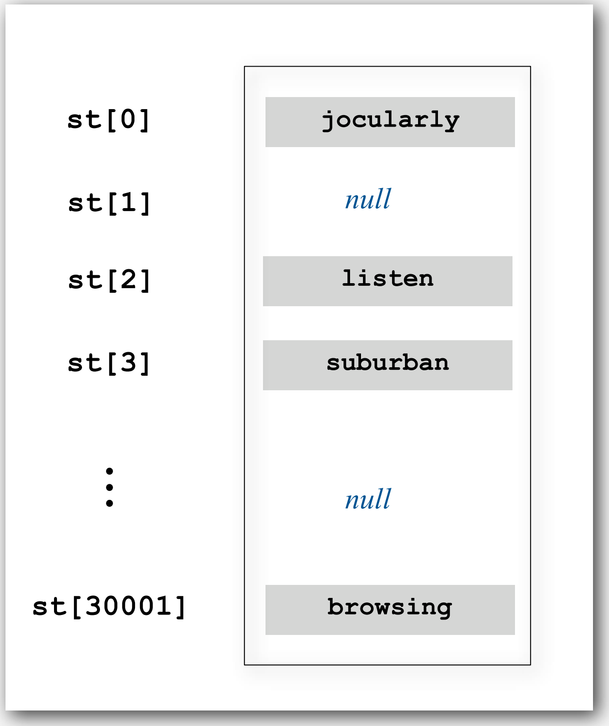
M times faster than sequential search

- hash functions
- separate chaining
- **linear probing**
- applications

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing (M = 30001, N = 15000)

Linear probing

Use an array of size M > N.

- Hash: map key to integer i between 0 and M-1.
- Insert: put in slot i if free; if not try i+1, i+2, etc.
- Search: search slot i; if occupied but no match, try i+1, i+2, etc.

-	-	-	S	H	-	-	A	C	E	R	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

-	-	-	S	H	-	-	A	C	E	R	I	-
0	1	2	3	4	5	6	7	8	9	10	11	12

insert I
hash(I) = 11

-	-	-	S	H	-	-	A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

insert N
hash(N) = 8

Linear probing: trace of standard indexing client

key	hash	value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0							S 0									
E	10	1							S 0				E 1					
A	4	2					A 2		S 0				E 1					
R	14	3					A 2		S 0				E 1				R 3	
C	5	4					A 2	C 5	S 0				E 1				R 3	
H	4	5					A 2	C 5	S 0	H 5			E 1				R 3	
E	10	6					A 2	C 5	S 0	H 5			E 6				R 3	
X	15	7					A 2	C 5	S 0	H 5			E 6				R 3	X 7
A	4	8					A 8	C 5	S 0	H 5			E 6				R 3	X 7
M	1	9		M 9			A 8	C 5	S 0	H 5			E 6				R 3	X 7
P	14	10	P 10	M 9			A 8	C 5	S 0	H 5			E 6				R 3	X 7
L	6	11	P 10	M 9			A 8	C 5	S 0	H 5	L 11		E 6				R 3	X 7
E	10	12	P 10	M 9			A 8	C 5	S 0	H 5	L 11		E 12				R 3	X 7

entries in red are new

entries in gray are untouched

keys in black are probes

probe sequence wraps to 0

keys[]

vals[]

Linear probing ST implementation

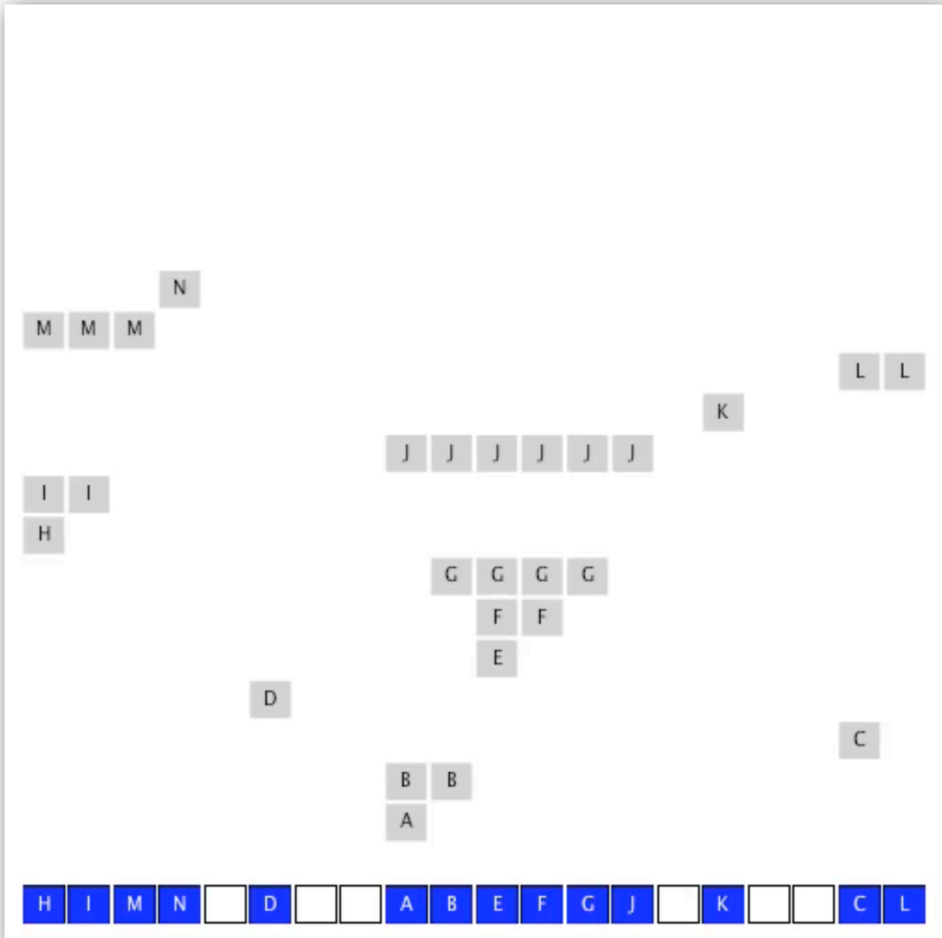
```
public class LinearProbingST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   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 (key.equals(keys[i]))
             break;
      vals[i] = val;
      keys[i] = key;
   }

   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
code omitted

Clustering

Cluster. A contiguous block of items.

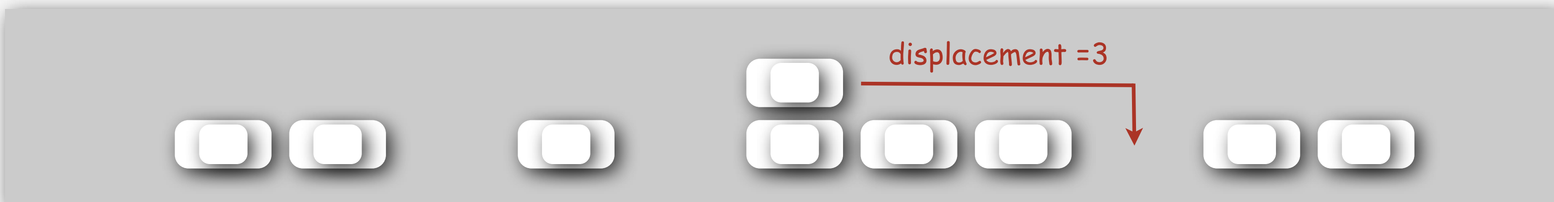
Observation. New keys likely to hash into middle of big clusters.



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, …

Q. What is mean displacement of a car?



Empty. With M/2 cars, mean displacement is ~ 3/2.

Full. With M cars, mean displacement is ~ $\sqrt{\pi M / 8}$

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of probes in a hash table of size M that contains N = α M keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit — search miss / insert

Pf. [Knuth 1962] A landmark in analysis of algorithms.

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: α = N/M < 1/2 $\Rightarrow$ constant-time ops.

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	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
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	`compareTo()`
hashing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

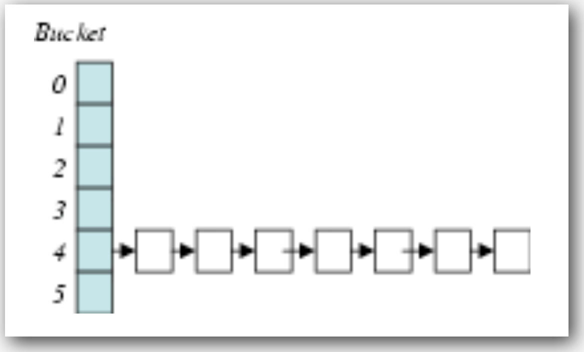
* under uniform hashing assumption

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 to find two keys that hash to same value.

Ex. $\underbrace{\text{MD4, MD5, SHA-0, SHA-1, SHA-2}}_{\text{known to be insecure}}$, WHIRLPOOL, RIPEMD-160.

```
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.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate chaining variant)

- Hash to two positions, put key in shorter of the two chains.
- Reduces average 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.

Hashing vs. balanced trees

Hashing.

- 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 trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black trees: `java.util.TreeMap`, `java.util.TreeSet`.
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`.

- hash functions
- separate chaining
- linear probing
- **applications**

Set API

Mathematical set. A collection of distinct keys.

public class SET<Key extends Comparable<Key>>		
	`SET()`	*create an empty set*
`void`	`add(Key key)`	*add the key to the set*
`boolean`	`contains(Key key)`	*is the key in the set?*
`void`	`remove(Key key)`	*remove the key from the set*
`int`	`size()`	*return the number of keys in the set*
`Iterator<Key>`	`iterator()`	*iterator through keys in the set*

Q. How to implement?

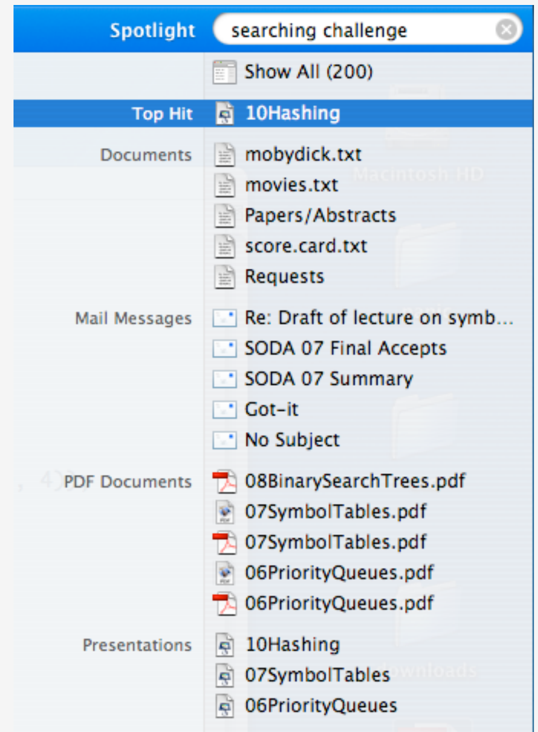
Searching challenge 5

Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.

Which searching method to use?

- Hashing
- Red-black-trees
- Doesn't matter much.



Index for a PC or the web

Solution. Symbol table with:

- Key = query string.
- Value = set of pointers to files.

```
ST<String, SET<File>> st = new ST<String, SET<File>>();
for (File file : filesystem)
{
   In in = new In(file);
   String[] words = in.readAll().split("\\s+");
   for (int i = 0; i < words.length; i++)
   {
       String s = words[i];
       if (!st.contains(s))
          st.put(s, new SET<File>());
       SET<File> files = st.get(s);
       files.add(file);
   }
}
```

build index

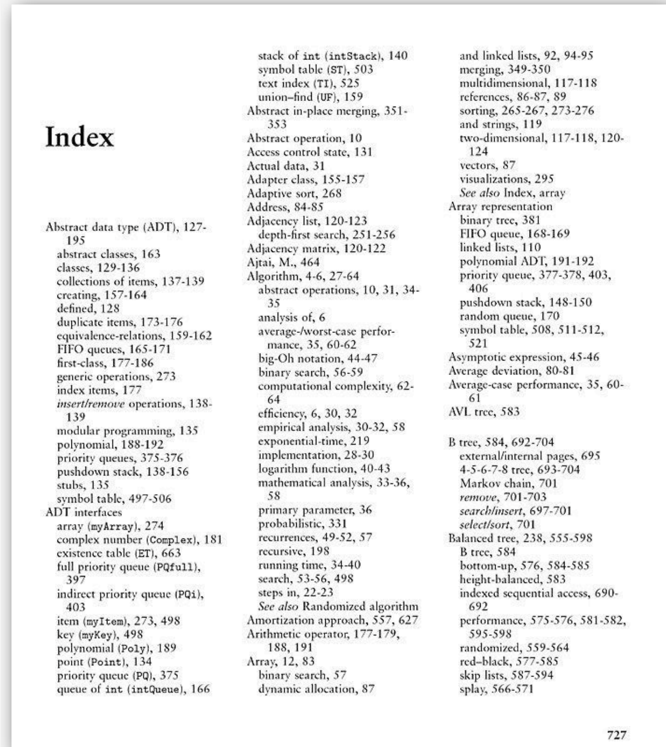
```
SET<File> files = st.get(query);
for (File file : files) ...
```

process lookup request

Searching challenge 6

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.



Index

727

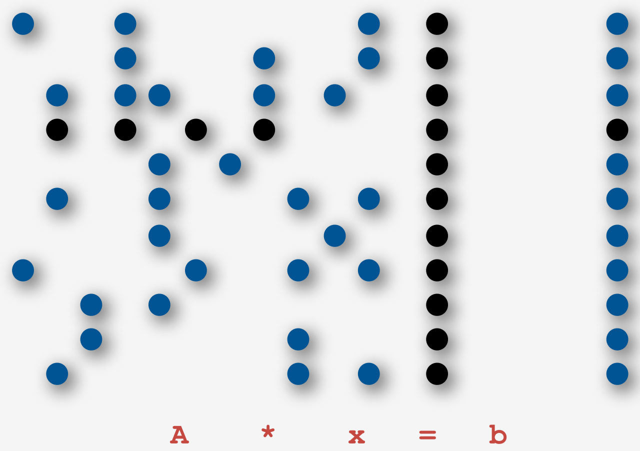
Which searching method to use?

1. Hashing
2. Red-black-tree
3. Doesn't matter much.

Searching challenge 2

Problem. Sparse matrix-vector multiplication.

Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.



Matrix-vector multiplication (standard implementation)

$$\begin{array}{ccc} \texttt{a[][]} & \texttt{x[]} & & \texttt{b[]} \\ \begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} & \begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} & = & \begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix} \end{array}$$

```
...
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

Vector representations

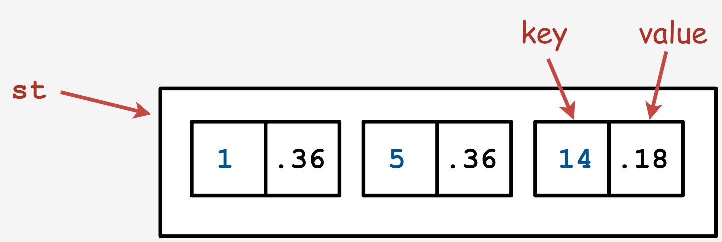
1D array (standard) representation.

- 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;
   public SparseVector()
    {  v = new HashST<Integer, Double>();  }

   public void put(int i, double x)
    {  v.put(i, x);  }

   public double get(int i)
   {
      if (!v.contains(i)) return 0.0;
      else return v.get(i);
   }
   public Iterable<Integer> indices()
   {  return v.keys();  }

   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : v.indices())
           sum += that[i]*this.get(i);
       return sum;
   }
}
```

`HashST` because order not important

empty ST represents all 0s vector

`a[i] = value`

`return a[i]`

dot product is constant time for sparse vectors

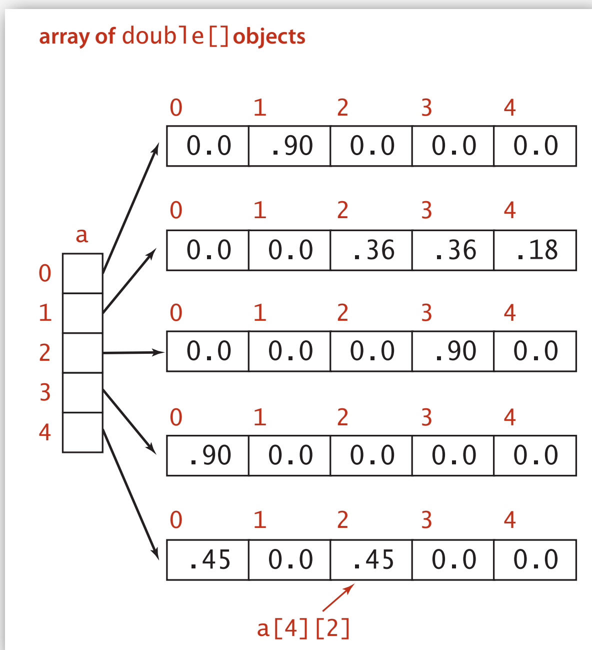
Matrix representations

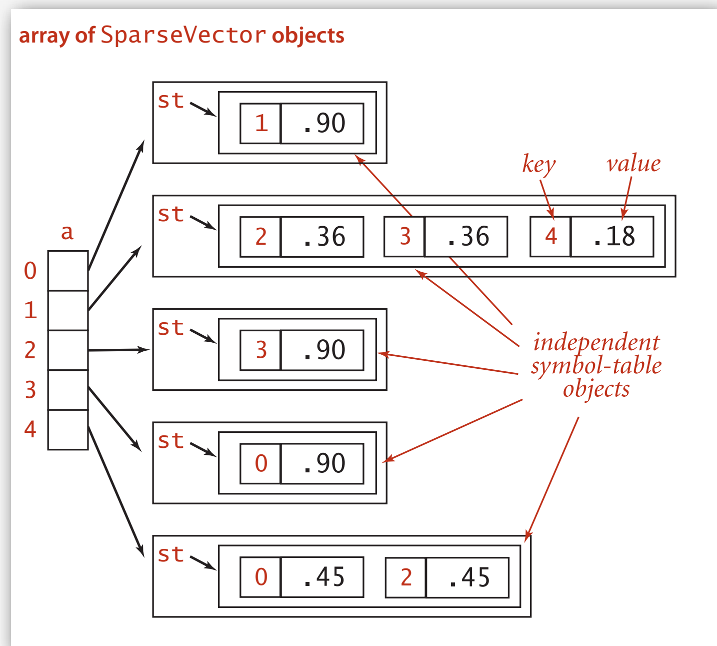
2D array (standard) representation: Each row of matrix is an array.

- Constant time access to elements.
- Space proportional to N^2.

Sparse 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

$$\begin{array}{cccc} \texttt{a[][]} & \texttt{x[]} & & \texttt{b[]} \\ \begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix} & \begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix} & = & \begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix} \end{array}$$

```
..
SparseVector[] a;
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);
```

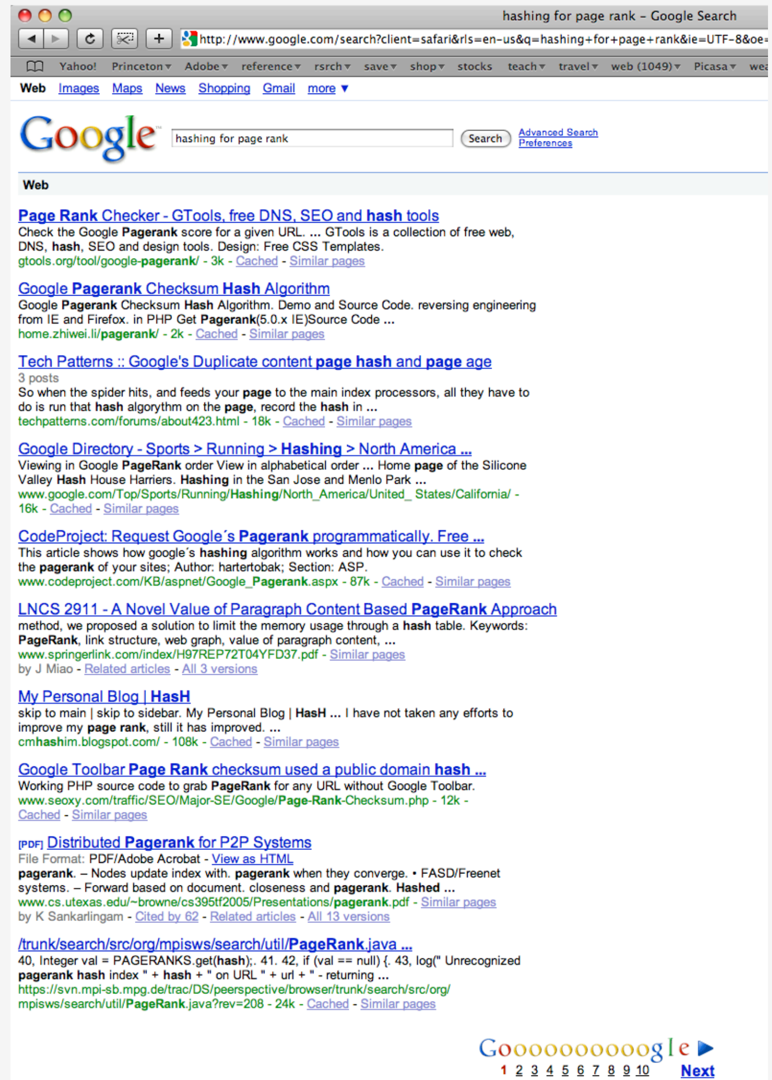
one loop
linear running time
for sparse matrix

Searching challenge 7

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.