

CSU22011: ALGORITHMS AND DATA STRUCTURES I

Lecture 1: Module Overview & Introduction

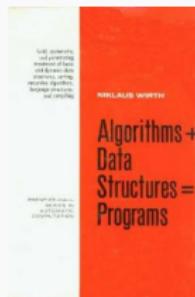
Vasileios Koutavas

School of Computer Science and Statistics



Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

→ **Objective:** learn to solve computational problems **efficiently**



“Algorithms + Data Structures = Programs”
— Niklaus Wirth



- **Algorithm:** The steps to **correctly** perform a task that answers a **general**¹ computational problem

- **Algorithm:** The steps to **correctly** perform a task that answers a **general**¹ computational problem
 - What is the median age of all people in Ireland?

- **Algorithm:** The steps to **correctly** perform a task that answers a **general**¹ computational problem
 - What is the median age of all people in Ireland?
 - What is the quickest route from here to the airport?

- **Algorithm:** The steps to **correctly** perform a task that answers a **general**¹ computational problem
 - What is the median age of all people in Ireland?
 - What is the quickest route from here to the airport?

¹Algorithms answering the above questions should work when the population of Ireland changes & for other destinations!

- **Algorithm:** The steps to **correctly** perform a task that answers a **general**¹ computational problem
 - What is the median age of all people in Ireland?
 - What is the quickest route from here to the airport?
- **Data Structures:** The ways to store the information needed for the algorithm.
 - Array, Linked List, Hash Table, Binary Tree, etc.

¹Algorithms answering the above questions should work when the population of Ireland changes & for other destinations!

ALGORITHMS + DATA STRUCTURES = PROGRAMS

Programs

→ must be **correct**

Programs

- must be **correct**
 - Mathematical **guarantee** of correctness only through **Formal Verification** (see CSU44004)

Programs

- must be **correct**
 - Mathematical **guarantee** of correctness only through **Formal Verification** (see CSU44004)
 - **Confidence** of correctness through **Testing**
 - in CSU22011 (and in the SW industry): **Unit Testing**

Programs

- must be **correct**
 - Mathematical **guarantee** of correctness only through **Formal Verification** (see CSU44004)
 - **Confidence** of correctness through **Testing**
 - in CSU22011 (and in the SW industry): **Unit Testing**
 - Other kinds of testing: integration testing, validation testing, user testing, etc.

Programs

- must be **correct**
 - Mathematical **guarantee** of correctness only through **Formal Verification** (see CSU44004)
 - **Confidence** of correctness through **Testing**
 - in CSU22011 (and in the SW industry): **Unit Testing**
 - Other kinds of testing: integration testing, validation testing, user testing, etc.
- must be **efficient**

WHAT IS EFFICIENCY?

Is this a good measure of a program's efficiency?

- How long it takes the program to run on my laptop

WHAT IS EFFICIENCY?

Is this a good measure of a program's efficiency?

- How long it takes the program to run on my laptop
- How long it takes the program to run on the fastest computer

WHAT IS EFFICIENCY?

Is this a good measure of a program's efficiency?

- How long it takes the program to run on my laptop
- How long it takes the program to run on the fastest computer
- How long it takes the program to run on the slowest computer

WHAT IS EFFICIENCY?

Is this a good measure of a program's efficiency?

- How long it takes the program to run on my laptop
- How long it takes the program to run on the fastest computer
- How long it takes the program to run on the slowest computer
- How long it takes the program to run with the largest input

WHAT IS EFFICIENCY?

Is this a good measure of a program's efficiency?

- How long it takes the program to run on my laptop
- How long it takes the program to run on the fastest computer
- How long it takes the program to run on the slowest computer
- How long it takes the program to run with the largest input
- How long it takes the program to run with the smallest input

WHAT IS EFFICIENCY?

Established measure: how well the program **scales** to larger inputs

- When I **double** the input size my program takes **the same** time to run on the same computer (*constant running time*).
- When I **double** the input size my program takes **twice** the time to run on the same computer (*linear running time*).
- ...

WHAT IS EFFICIENCY?

Established measure: how well the program **scales** to larger inputs

- When I **double** the input size my program takes **the same** time to run on the same computer (*constant running time*).
- When I **double** the input size my program takes **twice** the time to run on the same computer (*linear running time*).
- ...

We also care about **memory needed**:

- When I **double** the input size my program needs **the same** amount of memory to run (*constant memory space*).
- When I **double** the input size my program takes **twice** the amount of memory to run (*linear memory space*).
- ...

WHAT IS EFFICIENCY?

Established measure: how well the program **scales** to larger inputs

- When I **double** the input size my program takes **the same** time to run on the same computer (*constant running time*).
- When I **double** the input size my program takes **twice** the time to run on the same computer (*linear running time*).
- ...

We also care about **memory needed**:

- When I **double** the input size my program needs **the same** amount of memory to run (*constant memory space*).
- When I **double** the input size my program takes **twice** the amount of memory to run (*linear memory space*).
- ...

Scalability may sometimes seem crude but at least allows us to **compare algorithms**

- Learn **the most common A&DS** that every CS graduate must know.
 - Example algorithms: Merge Sort, Union-Find, Dijkstra's Shortest Path Tree, ...

The Software Engineer's toolbox



The Software Engineer's toolbox



- Learn **the most common A&DS** that every CS graduate must know.
 - Example algorithms: Merge Sort, Union-Find, Dijkstra's Shortest Path Tree, ...
- Identify which known algorithms/data structures **best fit** specific problems
 - Example: What is the best algorithm for finding the median?
 - It depends:^a QuickSelect, MedianOfMedians, IntroSelect, using SoftHeaps

The Software Engineer's toolbox



- Learn **the most common A&DS** that every CS graduate must know.
 - Example algorithms: Merge Sort, Union-Find, Dijkstra's Shortest Path Tree, ...
- Identify which known algorithms/data structures **best fit** specific problems
 - Example: What is the best algorithm for finding the median?
 - It depends:^a QuickSelect, MedianOfMedians, IntroSelect, using SoftHeaps
- Learn **Abstract Data Types**: interfaces of A&DS

The Software Engineer's toolbox



- Learn **the most common A&DS** that every CS graduate must know.
 - Example algorithms: Merge Sort, Union-Find, Dijkstra's Shortest Path Tree, ...
- Identify which known algorithms/data structures **best fit** specific problems
 - Example: What is the best algorithm for finding the median?
 - It depends:^a QuickSelect, MedianOfMedians, IntroSelect, using SoftHeaps
- Learn **Abstract Data Types**: interfaces of A&DS
- Practice **implementing A&DS**

^a<https://www.quora.com/What-is-the-most-efficient-algorithm-to-find-the->

The Computer Scientist's toolbox



- Learn to **evaluate** new algorithms
 - Efficiency: **calculate** the running time and memory usage
 - how well they scale
 - **Measuring aspects:** Worst-case, average-case, amortised, experimental performance
 - **Measuring systems:** big-O notation, tilde notation, cost models
 - Correctness: **rigorous testing** and some informal correctness arguments (see Unit Testing, test coverage)

CSU22011 LOGISTICS

Primary website: <https://mymodule.tcd.ie>

EXAMPLE PROBLEM

A software engineer was asked to design an algorithm which will input two **unsorted** arrays of integers, A (of size N) and B (also of size N), and will output **true** when all integers in A are present in B.

A software engineer was asked to design an algorithm which will input two **unsorted** arrays of integers, A (of size N) and B (also of size N), and will output **true** when all integers in A are present in B.

The engineer came up with **two** alternatives. The first is:

```
boolean isContained1(int[] A, int[] B) {  
    boolean AInB = true;  
    for (int i = 0; i < A.length; i++) {  
        boolean iInB = linearSearch(B, A[i]);  
        AInB = AInB && iInB;  
    }  
    return AInB;  
}
```

A software engineer was asked to design an algorithm which will input two **unsorted** arrays of integers, A (of size N) and B (also of size N), and will output **true** when all integers in A are present in B. The engineer came up with **two** alternatives:

The second is:

```
boolean isContained2(int[] A, int[] B) {  
    int[] C = new int[B.length];  
    for (int i = 0; i < B.length; i++) { C[i] = B[i] }  
    sort(C); // heapsort  
    boolean AInC = true;  
    for (int i = 0; i < A.length; i++) {  
        boolean iInC = binarySearch(C, A[i]);  
        AInC = AInC && iInC;  
    }  
    return AInC;  
}
```

- (a) Calculate the **worst-case running time** of each of the two implementations.
- (b) For each implementation, how much **extra memory space** is it required to store copies of the elements in A and B? You should take into account any copies made within the methods `sort`, `linearSearch`, and `binarySearch`.
- (c) Find an implementation which is more efficient than both of the engineer's implementation.

WHY?

VIDEO

<http://www.youtube.com/embed/vSi6YoTPWLw?rel=0&start=8&end=165>

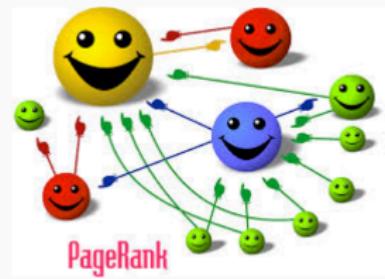
WHY?

- To get a technology job
- <http://www.careercup.com>



WHY?

- To create the “New Google”
<http://en.wikipedia.org/wiki/PageRank>



WHY?

- To make science

[http://ocw.mit.edu/courses/biological-engineering/](http://ocw.mit.edu/courses/biological-engineering/20-482j-foundations-of-algorithms-and-computational-techniques-in)

[20-482j-foundations-of-algorithms-and-computational-techniques-in](#)

Foundations of Algorithms and Computational Techniques in Systems Biology

[COURSE HOME](#)

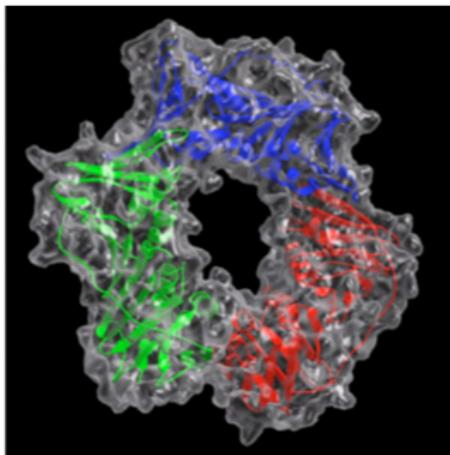


[SYLLABUS](#)

[LECTURE NOTES](#)

[ASSIGNMENTS](#)

[DOWNLOAD COURSE MATERIALS](#)



Instructor(s)

Prof. Bruce Tidor

Prof. Jacob White

MIT Course Number

20.482J / 6.581J

As Taught In

Spring 2006

Level

Graduate

Translated Versions

简体字

[CITE THIS COURSE](#)

The role of the Rad checkpoint complex was inferred from the 3-D structure predicted by comparative modeling at Lawrence Livermore National Laboratory. The Rad complex delays cell division to allow time for DNA repair to take place. (Image courtesy of the U.S. Department of Energy Genomics: GTL Program.)

WHY?

- To win big on the stock market

[http://www.theguardian.com/business/2012/oct/21/
superstar-traders-lost-magic](http://www.theguardian.com/business/2012/oct/21/superstar-traders-lost-magic)

One theory for the decline of the superstar trader is the rise of the analytical nerd and computerised algorithmic trading. Schmidt says: "The superstars are confronted with a changing market. The punting around is not working. You now need to be either a traditional long-term stock picker, a very short-term person working on algorithms, or a combination of both. There is no future for guys like Coffey."

WHY?

→ To rule the world!

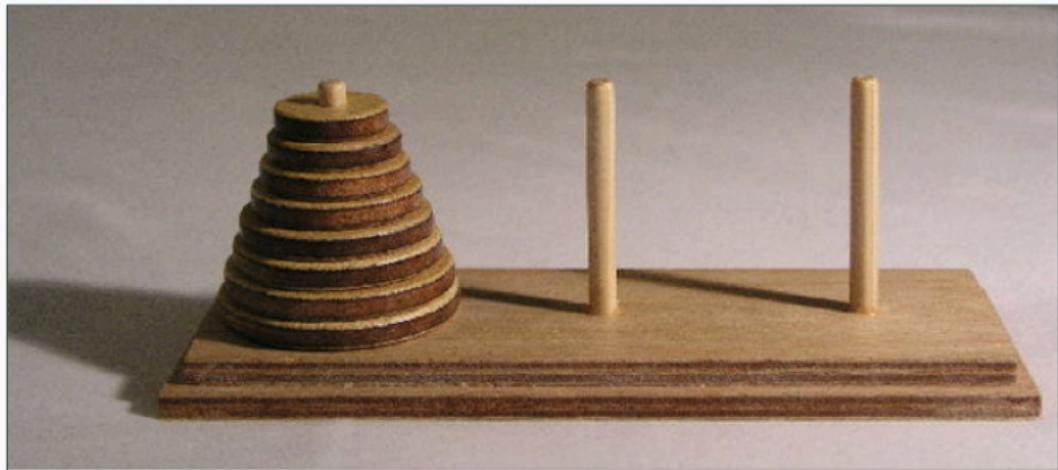
[http://www.theguardian.com/science/2013/jul/01/
how-algorithms-rule-world-nsa](http://www.theguardian.com/science/2013/jul/01/how-algorithms-rule-world-nsa)

The screenshot shows a news article from theguardian.com. At the top, there's a navigation bar with links for News, Sport, Comment, Culture, Business, Money, and Life & style. Below the navigation bar, a breadcrumb trail shows the article's path: News > Science > Mathematics. The main title of the article is "How algorithms rule the world". The subtitle reads: "The NSA revelations highlight the role sophisticated algorithms play in sifting through masses of data. But more surprising is their widespread use in our everyday lives. So should we be more wary of their power?"

WHY?

→ For fun!

http://en.wikipedia.org/wiki/Tower_of_Hanoi



CSU22011: ALGORITHMS AND DATA STRUCTURES I

Lecture 2: Analysis of Algorithms

Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

LAST LECTURE

Programs = Algorithms + Data structures.

- Must be **correct**
- Must be **efficient**

LAST LECTURE

Programs = Algorithms + Data structures.

- Must be **correct**
- Must be **efficient**
 - What is efficiency?
 - **Analysis of Algorithms**: How to evaluate efficiency?

LAST LECTURE

Programs = Algorithms + Data structures.

- Must be **correct**
- Must be **efficient**
 - What is efficiency?
 - **Analysis of Algorithms**: How to evaluate efficiency?
 - Established measure of efficiency (from now on **performance**): how the program **scales** to larger inputs. Measure the effect of **doubling** the input size to
 - the **running time** of the algorithm
 - the **memory footprint** of the algorithm

LAST LECTURE

Programs = Algorithms + Data structures.

- Must be **correct**
- Must be **efficient**
 - What is efficiency?
 - **Analysis of Algorithms**: How to evaluate efficiency?
 - Established measure of efficiency (from now on **performance**): how the program **scales** to larger inputs. Measure the effect of **doubling** the input size to
 - the **running time** of the algorithm
 - the **memory footprint** of the algorithm
 - We want to be able to **analyse algorithms** w.r.t. their performance.

WHY ANALYSE ALGORITHMS?

- **Good programmer:** to predict the performance of our programs.
- **Good client:** to choose between alternative algorithms/implementations.
- **Good manager:** to provide guarantees to clients / avoid client complaints.
- **Good scientist:** to understand the nature of computing.

EXAMPLE: LINEAR SEARCH

Linear Search: simple search in an array:

```
boolean linearSearch1(int[] ar, int s) {  
    boolean found = false;  
    for (int i = 0; i < ar.length; i++) {  
        if (ar[i] == s) found = true;  
    }  
    return found;  
}
```

How can we evaluate how well the running time of this algorithm scales to larger inputs?

¹We will see a number of approaches how to evaluate running times

EXAMPLE: LINEAR SEARCH

Linear Search: simple search in an array:

```
boolean linearSearch1(int[] ar, int s) {  
    boolean found = false;  
    for (int i = 0; i < ar.length; i++) {  
        if (ar[i] == s) found = true;  
    }  
    return found;  
}
```

How can we evaluate how well the running time of this algorithm scales to larger inputs?

- Evaluate¹ its running time for various sizes of array **ar**. Then calculate the **rate of growth** of the running time with respect to the input size (stay tuned).

¹We will see a number of approaches how to evaluate running times

EXAMPLE: LINEAR SEARCH 2

Improved Linear Search: return as soon as possible.

```
boolean linearSearch2(int[] ar, int s) {  
    for (int i = 0; i < ar.length; i++) {  
        if (ar[i] == s) return true;  
    }  
    return false;  
}
```

Consider inputs arrays of size 100. Is the running time the same?

EXAMPLE: LINEAR SEARCH 2

Improved Linear Search: return as soon as possible.

```
boolean linearSearch2(int[] ar, int s) {  
    for (int i = 0; i < ar.length; i++) {  
        if (ar[i] == s) return true;  
    }  
    return false;  
}
```

Consider inputs arrays of size 100. Is the running time the same?

NO! Even for the same input size, some inputs will require more time than others.

- best case inputs; e.g: `ar=[1,2,...,100]`, `s = 1`
- worst case inputs; e.g: `ar=[1,2,...,100]`, `s = 101`
- everything in between (sometimes we talk about average case)

EXAMPLE: LINEAR SEARCH 2

Improved Linear Search: return as soon as possible.

```
boolean linearSearch2(int[] ar, int s) {  
    for (int i = 0; i < ar.length; i++) {  
        if (ar[i] == s) return true;  
    }  
    return false;  
}
```

Consider inputs arrays of size 100. Is the running time the same?

NO! Even for the same input size, some inputs will require more time than others.

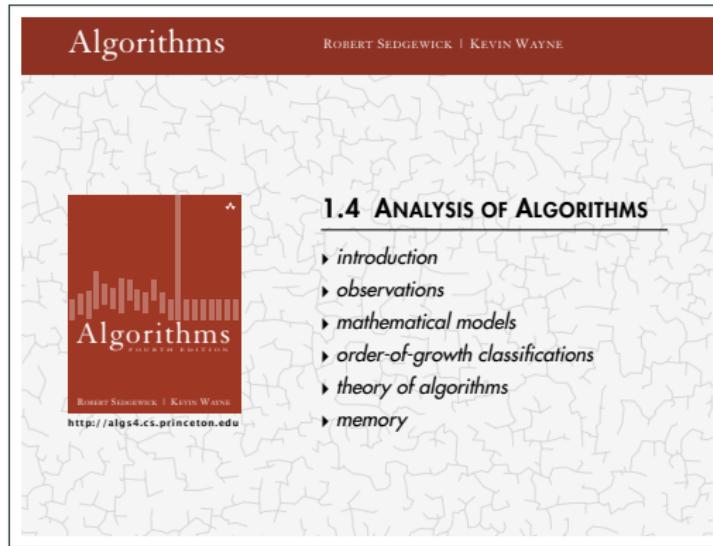
- best case inputs; e.g: `ar=[1,2,...,100]`, `s = 1`
- worst case inputs; e.g: `ar=[1,2,...,100]`, `s = 101`
- everything in between (sometimes we talk about average case)

In this module (and usually in programming practice) we focus on the **worst-case inputs**.

EVALUATING RUNNING TIMES

In the course of the next lectures we will see the following methods for evaluating how an algorithm's running time scales:

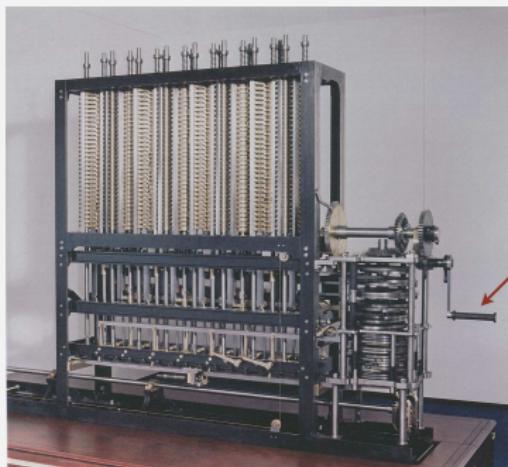
1. **Scientific method:** measure running times through experiments
2. **Mathematical methods:** consider a convenient model of computation and:
 - sum up the number of program steps for **worst-case** inputs of size N (N is a variable).
 - calculate the number of program steps for **worst-case** inputs of size N , when N nears infinity.This is the **asymptotic** (worst-case) running time — notation big-O, Ω , Θ .



- Parts from S&W 1.4
- Estimate the performance of algorithms by
 - Experiments & Observations
 - Precise Mathematical Calculations

Running time

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



how many times do you
have to turn the crank?

Analytic Engine

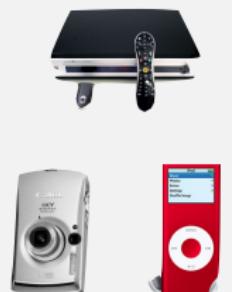
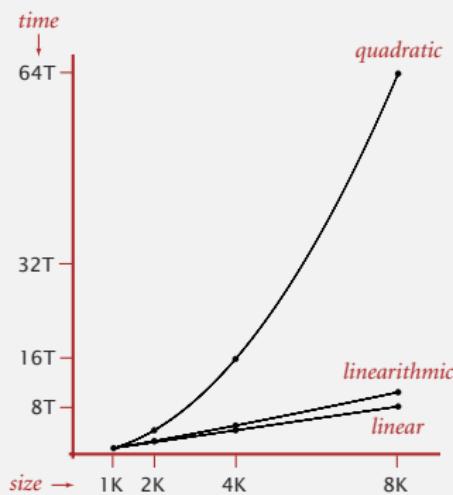
Some algorithmic successes

Discrete Fourier transform.

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



Friedrich Gauss
1805



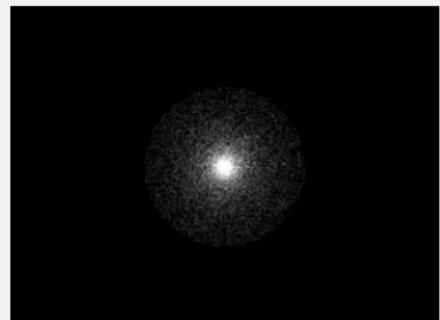
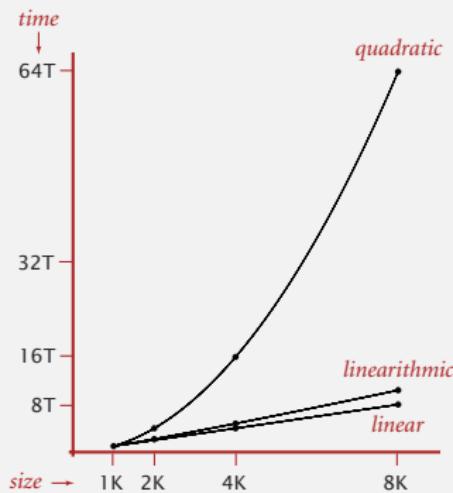
Some algorithmic successes

N-body simulation.

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

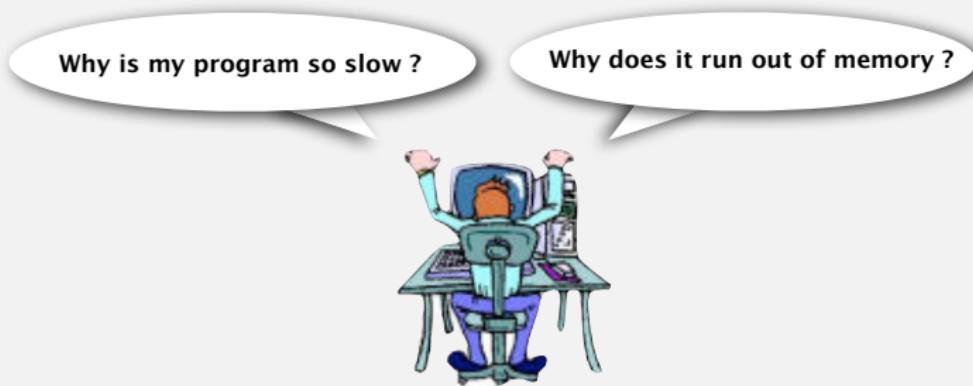


Andrew Appel
PU '81



The challenge

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



Insight. [Knuth 1970s] Use **scientific method** to understand performance.

Scientific method applied to analysis of algorithms

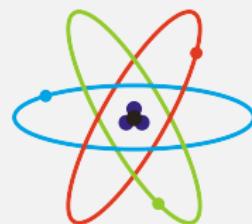
A framework for predicting performance and comparing algorithms.

Scientific method.

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

Principles.

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



Feature of the natural world. Computer itself.

SCIENTIFIC APPROACH:
EVALUATING PERFORMANCE BY
EXPERIMENTS

Example: 3-SUM

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

```
% more 8ints.txt
8
30 -40 -20 -10 40 0 10 5
% java ThreeSum 8ints.txt
4
```

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



Context. Deeply related to problems in computational geometry.

3-SUM: brute-force algorithm

```
public class ThreeSum
{
    public static int count(int[] a)
    {
        int N = a.length;
        int count = 0;
        for (int i = 0; i < N; i++)
            for (int j = i+1; j < N; j++)
                for (int k = j+1; k < N; k++)           ← check each triple
                    if (a[i] + a[j] + a[k] == 0)          ← for simplicity, ignore
                        count++;                      integer overflow
        return count;
    }

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

QUESTION

Suppose we care only about 100-element arrays.

Q. What is a **worst-case input** for ThreeSum?

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

QUESTION

Suppose we care only about 100-element arrays.

Q. What is a **worst-case input** for ThreeSum?

A. They are all worst case inputs. For-loops always run to the end.

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

Measuring the running time

Q. How to time a program?

A. Manual.



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



70

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



528

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



4039

Measuring the running time

Q. How to time a program?

A. Automatic.

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

```
Stopwatch()
```

create a new stopwatch

```
double elapsedTime()
```

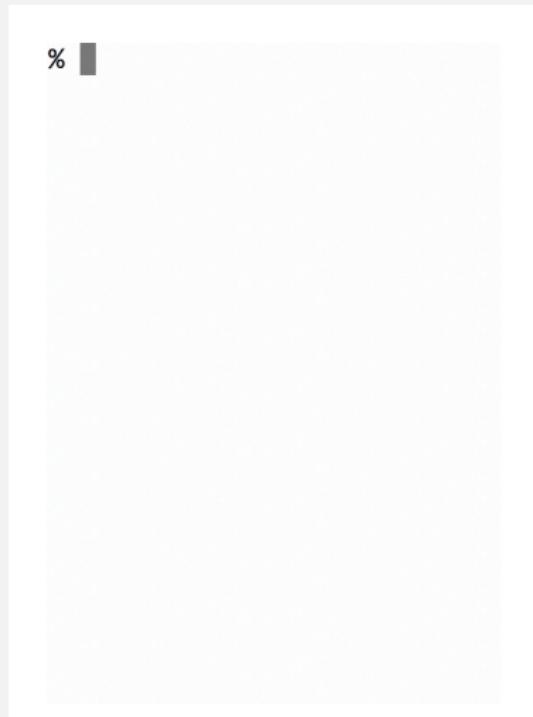
time since creation (in seconds)

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

Empirical analysis

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

%



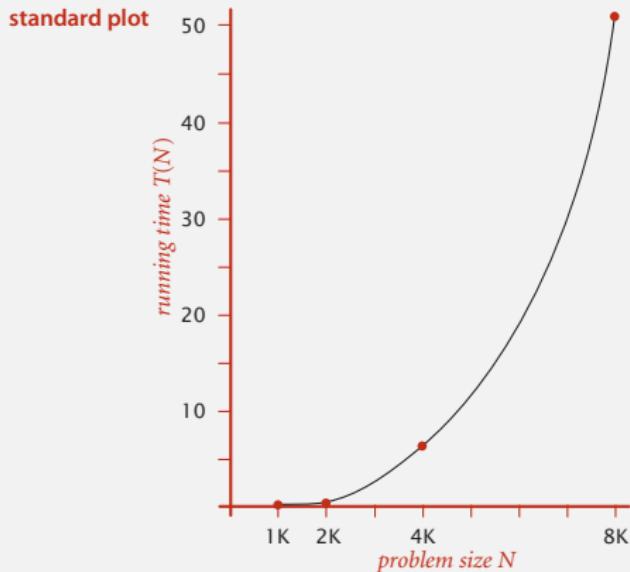
Empirical analysis

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

N	time (seconds) [†]
250	0.0
500	0.0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?

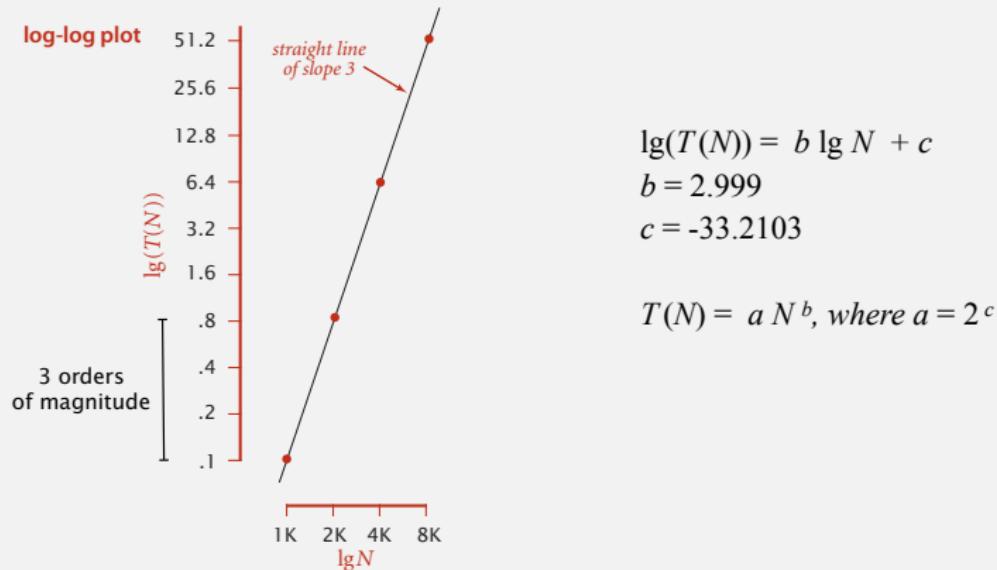
Data analysis

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



Data analysis

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



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

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

power law

slope

Try out the scientific analysis through experiments:

[https://docs.google.com/spreadsheets/d/
1WnihyK6g1pYdcT2ndZ0qNNRkTitXkWKnOrTgCnM-bw8/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1WnihyK6g1pYdcT2ndZ0qNNRkTitXkWKnOrTgCnM-bw8/edit?usp=sharing)

Prediction and validation

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

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

Predictions.

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

Observations.

N	time (seconds) †
8,000	51.1
8,000	51.0
8,000	51.1
16,000	410.8

validates hypothesis!

Doubling hypothesis

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

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

N	time (seconds) †	ratio	lg ratio
250	0.0		-
500	0.0	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8.0	3.0 ← lg (6.4 / 0.8) = 3.0
8,000	51.1	8.0	3.0

$$\begin{aligned}\frac{T(2N)}{T(N)} &= \frac{a(2N)^b}{aN^b} \\ &= 2^b\end{aligned}$$

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

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

Caveat. Cannot identify logarithmic factors with doubling hypothesis.

Doubling hypothesis

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

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

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

N	time (seconds) \dagger
8,000	51.1
8,000	51.0
8,000	51.1

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

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

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



almost identical hypothesis
to one obtained via linear regression

Experimental algorithmics

System independent effects.

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

System dependent effects.

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

Bad news. Difficult to get precise measurements.

Good news. Much easier and cheaper than other sciences.



e.g., can run huge number of experiments

This was the **scientific approach** to algorithm analysis.

In the **mathematical approach** we do **calculations** instead of experiments.

CSU22011: ALGORITHMS AND DATA STRUCTURES

Lecture 3: Mathematical Approach to the Analysis of Algorithms

Vasileios Koutavas

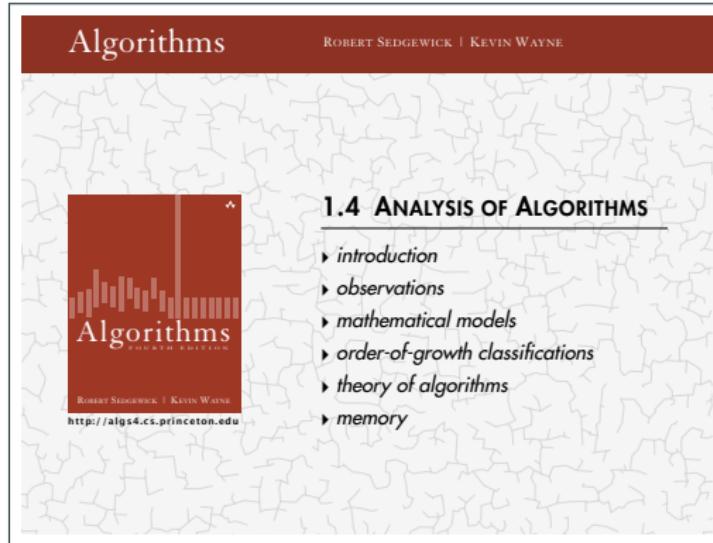


School of Computer Science and Statistics
Trinity College Dublin

LAST LECTURE

Analysis of Algorithms: evaluates the efficiency of our programs

- How the **running time/memory footprint** of the algorithm **scales** when the input size increases.
 - Express running time as a function $T(N)$, where N is the size of the input.
- For any given input size N , we will be focusing on the **worst-case** inputs (the worst case value of $T(N)$)
- **Scientific method:** measure running times through experiments and discover $T(N)$



Mathematical methods – consider a model of computation and count the number of program steps for worst-case inputs of size N.

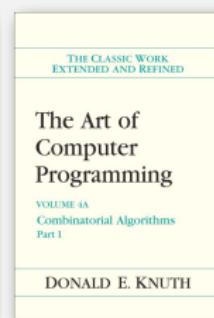
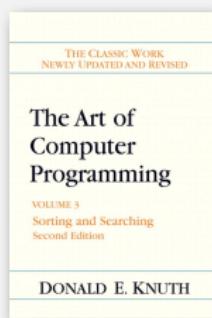
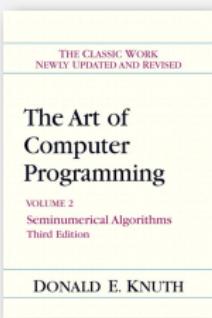
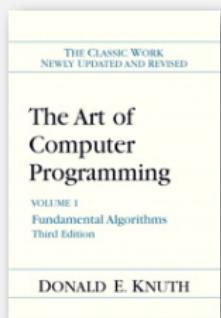
- Parts from S&W 1.4
- Evaluate the performance of algorithms by Mathematical Calculations

MATHEMATICAL APPROACH:
EVALUATING PERFORMANCE BY
CALCULATIONS

Mathematical models for running time

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

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



Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

Challenge. How to estimate constants.

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	<code>Math.sin(theta)</code>	91.3
arctangent	<code>Math.atan2(y, x)</code>	129.0
...

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

Cost of basic operations

Observation. Most primitive operations take constant time.

operation	example	nanoseconds \dagger
variable declaration	<code>int a</code>	c_1
assignment statement	<code>a = b</code>	c_2
integer compare	<code>a < b</code>	c_3
array element access	<code>a[i]</code>	c_4
array length	<code>a.length</code>	c_5
1D array allocation	<code>new int[N]</code>	$c_6 N$
2D array allocation	<code>new int[N][N]</code>	$c_7 N^2$

Caveat. Non-primitive operations often take more than constant time.

 novice mistake: abusive string concatenation

Example: 1-SUM

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

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

N array accesses

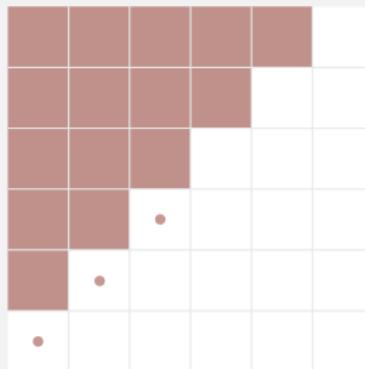
operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2N$

Example: 2-SUM

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

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

Pf. [n even]



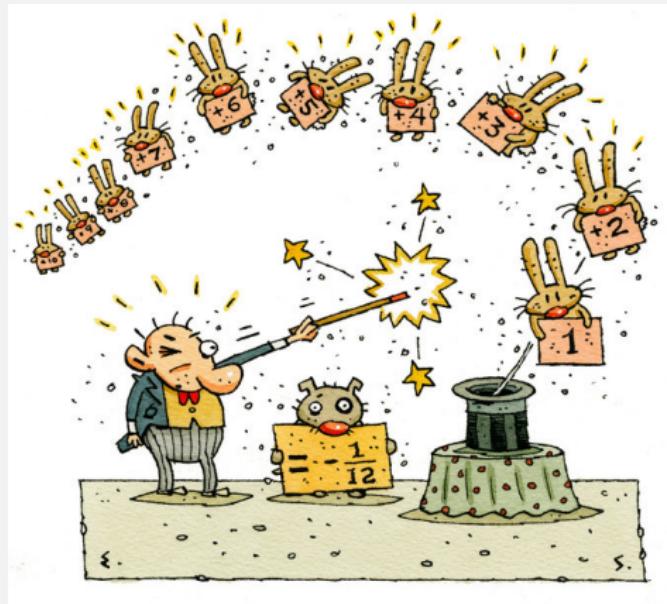
$$\begin{aligned} 0 + 1 + 2 + \dots + (N - 1) &= \frac{1}{2} N(N - 1) \\ &= \binom{N}{2} \end{aligned}$$

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

half of
square half of
diagonal

String theory infinite sum

$$1 + 2 + 3 + 4 + \dots = -\frac{1}{12}$$



<http://www.nytimes.com/2014/02/04/science/in-the-end-it-all-adds-up-to.html>

Example: 2-SUM

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

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

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

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



tedious to count exactly

Example: 2-SUM

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

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

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

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

}

tedious to count exactly

“best case” vs “worst case” input of size N . We are

CALCULATING PRECISE RUNNING TIME

$$T(N) = c_1A(N) + c_2B(N) + c_3C(N) + c_4D(N) + c_5E(N)$$

Where

c_1 :cost of array access

$A(N)$:number of array accesses

c_2 :cost of integer addition

$B(N)$:number of integer additions

c_3 :cost of integer comparison

$C(N)$:number of integer comparisons

c_4 :cost of increment

$D(N)$:number of increments

c_5 :cost of assignment

$E(N)$:number of assignments

(functions of the input size N)

CALCULATING PRECISE RUNNING TIME

$$T(N) = c_1A(N) + c_2B(N) + c_3C(N) + c_4D(N) + c_5E(N)$$

Where

c_1 :cost of array access

$A(N)$:number of array accesses

c_2 :cost of integer addition

$B(N)$:number of integer additions

c_3 :cost of integer comparison

$C(N)$:number of integer comparisons

c_4 :cost of increment

$D(N)$:number of increments

c_5 :cost of assignment

$E(N)$:number of assignments

(functions of the input size N)

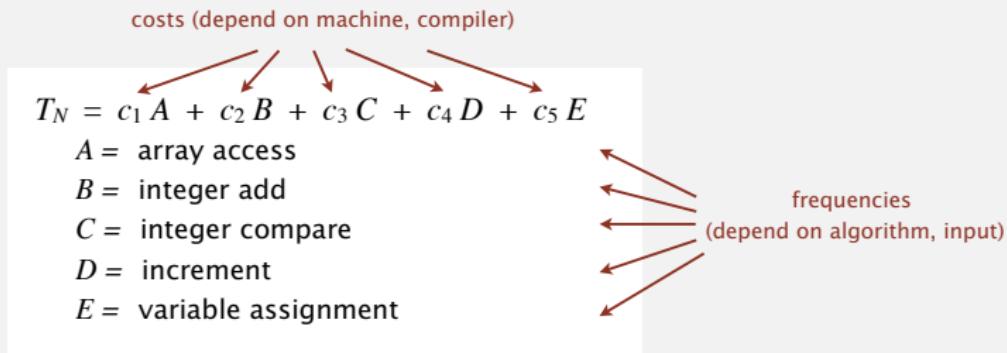
Q. Advantages / Disadvantages over scientific method?

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

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



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

We will simplify calculations using approximations based on
Cost Models

COST MODEL 1: ALL CONSTANT COSTS = 1

COST MODEL 1: ALL CONSTANT COSTS = 1

- New generation computers have smaller constants than previous generation

$$c_i = 1$$

$$T_N = A + B + C + D + E$$

Where

A :number of array accesses

B :number of integer additions

C :number of integer comparisons

D :number of increments

E :number of assignments

NON-CONSTANT COSTS

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + "ABCDEFG";`
- Method calls: `max = Collections.max(myList);`

NON-CONSTANT COSTS

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + "ABCDEFG";`
 - the cost of this operation is linear to the size of `str`
- Method calls: `max = Collections.max(myList);`

NON-CONSTANT COSTS

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + "ABCDEFG";`
 - the cost of this operation is linear to the size of `str`
 - when efficiency is important use `StringBuilder`
- Method calls: `max = Collections.max(myList);`

NON-CONSTANT COSTS

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + "ABCDEFG";`
 - the cost of this operation is linear to the size of `str`
 - when efficiency is important use `StringBuilder`
- Method calls: `max = Collections.max(myList);`
 - the cost of this operation is the cost of running the algorithm in `Collections.max` with an input of size `myList.size()`

Example: 2-SUM

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

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

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

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

}

tedious to count exactly

Estimate performance by adding up frequencies

COST MODEL 2: ONLY HIGHEST ORDER TERMS COUNT

Simplification 2: tilde notation

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

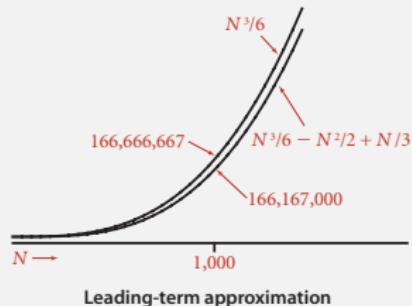
Ex 1. $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$

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

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

discard lower-order terms

(e.g., $N = 1000$: 166.67 million vs. 166.17 million)



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

Simplification 2: tilde notation

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

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

Estimate performance by adding up simplified frequencies

COST MODEL 3: COUNT ONLY SOME OPERATIONS

Simplifying the calculations

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

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(National Physical Laboratory, Teddington, Middlesex)

[Received 4 November 1947]

SUMMARY

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



Simplification 1: cost model

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

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

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

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

cost model = array accesses
(we assume compiler/JVM do not optimize any array accesses away!)

Performance estimate = (array accesses) $\times c_{\text{arraccess}}$

DON'T OVER-SIMPLIFY!

Careful!

Make sure that the operations you are not counting add up to a factor **lower** than the operations you do count.

COMBINATIONS OF COST MODELS

COMBINATIONS OF COST MODELS

Each cost model makes a **simplification** in the calculation of running time.

⇒ **approximation** of running time.

We can even **combine** the assumptions of different cost models.

Example: 2-SUM

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

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

"inner loop"

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

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

Performance estimate = simplified number of array accesses

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

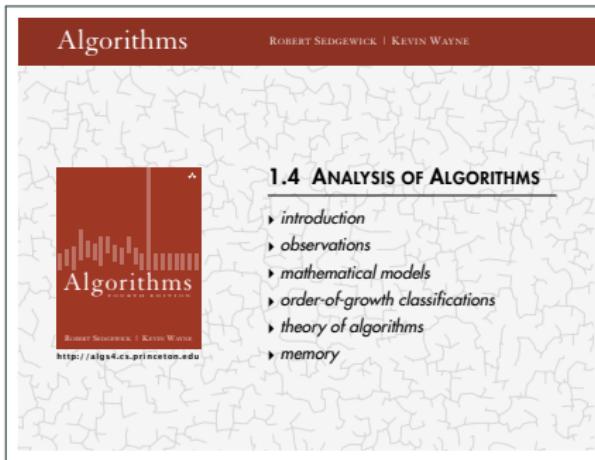
CSU22010: ALGORITHMS AND DATA STRUCTURES

Lecture 4: Order of Growth, Asymptotic Notation, Memory Performance

Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin



- Estimate the performance of algorithms by
 - Experiments & Observations
 - Precise Mathematical Calculations
 - Approximate Mathematical Calculations using Cost Models
 - Every basic operation costs 1 time unit
 - Keep only the higher-order terms
 - Count only some operations
- This Lecture: Classification according to running time order of growth

Doubling hypothesis

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

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

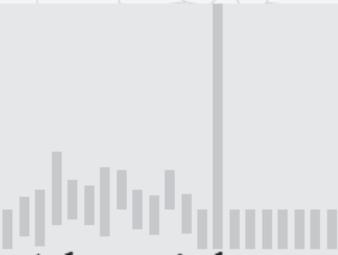
N	time (seconds) †	ratio	lg ratio
250	0.0		-
500	0.0	4.8	2.3
1,000	0.1	6.9	2.8
2,000	0.8	7.7	2.9
4,000	6.4	8.0	3.0 ← lg (6.4 / 0.8) = 3.0
8,000	51.1	8.0	3.0

$$\begin{aligned}\frac{T(2N)}{T(N)} &= \frac{a(2N)^b}{aN^b} \\ &= 2^b\end{aligned}$$

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

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

Caveat. Cannot identify logarithmic factors with doubling hypothesis.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Common order-of-growth classifications

Definition. If $f(N) \sim c g(N)$ for some constant $c > 0$, then the **order of growth** of $f(N)$ is $g(N)$.

- Ignores leading coefficient.
- Ignores lower-order terms.

Ex. The order of growth of the **running time** of this code is N^3 .

```
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++;
```

Typical usage. With running times.

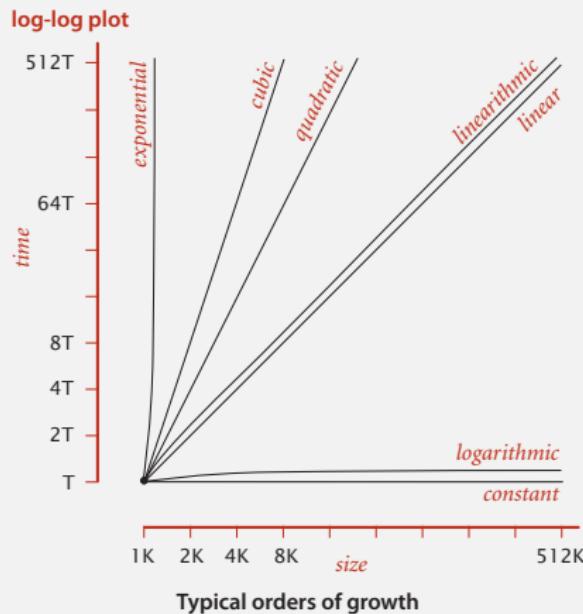
→ where leading coefficient
depends on machine, compiler, JVM, ...

Common order-of-growth classifications

Good news. The set of functions

$1, \log N, N, N \log N, N^2, N^3,$ and 2^N

suffices to describe the order of growth of most common algorithms.



Common order-of-growth classifications

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

Example: 3-SUM

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

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

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

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

- Count only array accesses
- Cost of each array access: 1 time unit
- use tilde notation

Order of Growth: N^3

EXAMPLE: BINARY SEARCH

Binary search demo

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

Binary search. Compare key against middle entry.

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



successful search for 33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Diagram illustrating a successful binary search for the key 33 in a sorted array. The array elements are 6, 13, 14, 25, 33, 43, 51, 53, 64, 72, 84, 93, 95, 96, 97. The search range is initially defined by indices 0 and 14. A red arrow labeled 'lo' points to index 4, where the value 33 is found. Another red arrow labeled 'hi' points to index 14, indicating the search space has been narrowed to the element at index 4.

Binary search: Java implementation

Trivial to implement?

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

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

one "3-way compare"

Invariant. If key appears in the array a[], then $a[lo] \leq key \leq a[hi]$.

Binary search: mathematical analysis

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

Def. $T(N) = \# \text{ key compares to binary search a sorted subarray of size } \leq N$.

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

\uparrow \uparrow
left or right half possible to implement with one
(floored division) 2-way compare (instead of 3-way)

Pf sketch. [assume N is a power of 2]

$$\begin{aligned} T(N) &\leq T(N/2) + 1 && [\text{given}] \\ &\leq T(N/4) + 1 + 1 && [\text{apply recurrence to first term}] \\ &\leq T(N/8) + 1 + 1 + 1 && [\text{apply recurrence to first term}] \\ &\vdots \\ &\leq T(N/N) + 1 + 1 + \dots + 1 && [\text{stop applying, } T(1) = 1] \\ &= 1 + \lg N \end{aligned}$$

LOGARITHMS

- The base of the logarithm contributes only a constant factor to the running time.

$$\log_a N = \frac{\log_b N}{\log_b a} = c \cdot \log_b N$$

where $c = 1/\log_b a$ is a constant (does not depend on N).

- EX. BINS (Binary search) runs in $T_{\text{BINS}}(N) \sim \lg N$ ¹ time.

Suppose SUPERBINS, a faster algorithm for binary search, that runs in $T_{\text{superbin}}(N) \sim \log_{16} N$ time.

Then we would have $T_{\text{SUPERBINS}}(N) \sim \frac{1}{\log_2 16} \lg N = \frac{1}{4} \lg N \sim \frac{1}{4} T_{\text{BINS}}(N)$.

Although the faster algorithm runs in 1/4 of the time of binary search, it still has the **same order of growth**:

- When the input size **doubles**, the running time increases by the same amount:

$$\frac{T_{\text{SUPERBINS}}(2N)}{T_{\text{SUPERBINS}}(N)} = \frac{\frac{1}{4} T_{\text{BINS}}(2N)}{\frac{1}{4} T_{\text{BINS}}(N)} = \frac{T_{\text{BINS}}(2N)}{T_{\text{BINS}}(N)}$$

¹ $\lg N$ is notation for $\log_2 N$

Example: 3-SUM

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

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

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

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

Can we do better?

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

Algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers $a[i]$ and $a[j]$, binary search for $-(a[i] + a[j])$.

What is the order of growth?

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10 0 5 10 30 40

binary search

(-40, -20) 60

(-40, -10) 50

(-40, 0) 40

(-40, 5) 35

(-40, 10) 30

:

:

(-20, -10) 30

:

:

(-10, 0) 10

:

:

(10, 30) -40

(10, 40) -50

(30, 40) -70

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

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

Algorithm.

- Step 1: Sort the N (distinct) numbers.
- Step 2: For each pair of numbers $a[i]$ and $a[j]$, binary search for $-(a[i] + a[j])$.

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

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

Remark. Can achieve N^2 by modifying binary search step.

input

30 -40 -20 -10 40 0 10 5

sort

-40 -20 -10 0 5 10 30 40

binary search

(-40, -20) 60

(-40, -10) 50

(-40, 0) 40

(-40, 5) 35

(-40, 10) 30

:

:

(-20, -10) 30

:

:

(-10, 0) 10

:

:

(10, 30) -40

(10, 40) -50

(30, 40) -70

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

Comparing programs

Hypothesis. The sorting-based $N^2 \log N$ algorithm for 3-SUM is significantly faster in practice than the brute-force N^3 algorithm.

N	time (seconds)
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1

`ThreeSum.java`

N	time (seconds)
1,000	0.14
2,000	0.18
4,000	0.34
8,000	0.96
16,000	3.67
32,000	14.88
64,000	59.16

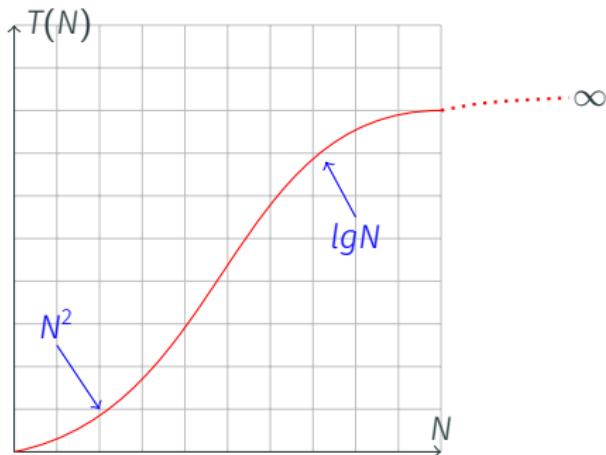
`ThreeSumDeluxe.java`

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

ASYMPTOTIC NOTATION

ASYMPTOTIC NOTATION

- In the Theory of Algorithms we are interested in the **order of growth** of runtime $T(N)$, expressed as a function of the input size N .
- $T(N)$ may not be a simple function and can have **different orders of growth** for different values of N .



- Rationale of asymptotic notation:
 - larger N 's are more important than smaller ones
 - consider the order of growth when $N \rightarrow \infty$ (**asymptotic order of growth**)

ASYMPTOTIC NOTATION

→ $\Theta(g(N))$: the **set** of functions with **asymptotic order of growth** $g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Theta(N^2)$.

We write $T(N) = \Theta(N^2)$ [†]

[†]Abuse of notation, means $T(N) \in \Theta(N^2)$.

ASYMPTOTIC NOTATION

- $\Theta(g(N))$: the **set** of functions with **asymptotic order of growth** $g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Theta(N^2)$.

We write $T(N) = \Theta(N^2)$ [†]

- $O(g(N))$: the **set** of functions with **asymptotic order of growth** $\leq g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $O(N^3)$.

We write $T(N) = O(N^3)$ [†]

[†]Abuse of notation, means $T(N) \in \Theta(N^2)$.

→ $\Theta(g(N))$: the **set** of functions with **asymptotic order of growth** $g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Theta(N^2)$.

We write $T(N) = \Theta(N^2)$ [†]

→ $O(g(N))$: the **set** of functions with **asymptotic order of growth** $\leq g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $O(N^3)$.

We write $T(N) = O(N^3)$ [†]

→ $\Omega(g(N))$: the **set** of functions with **asymptotic order of growth** $\geq g(N)$.

EX. The (worst case) running time $T(N)$ of Insertion Sort is in $\Omega(N)$.

We write $T(N) = \Omega(N)$ [†]

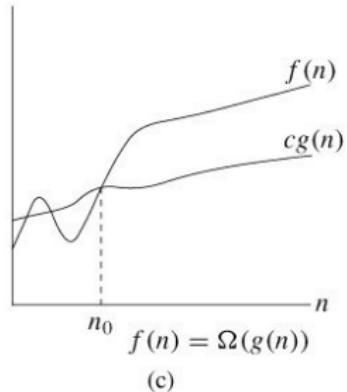
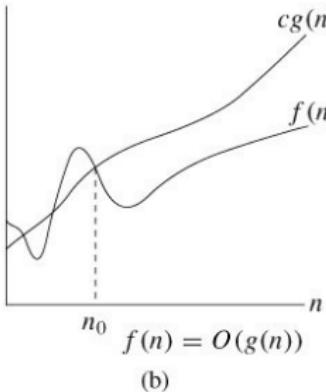
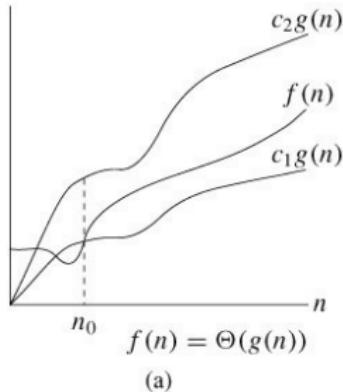
[†]Abuse of notation, means $T(N) \in \Theta(N^2)$.

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ ⋮	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ ⋮	develop lower bounds

FORMAL DEFINITIONS OF ASYMPTOTIC NOTATION

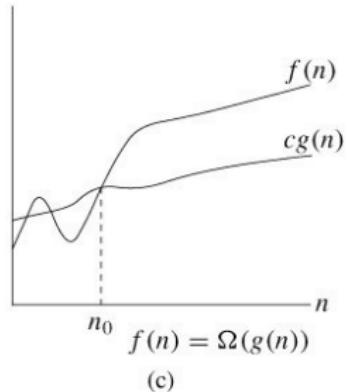
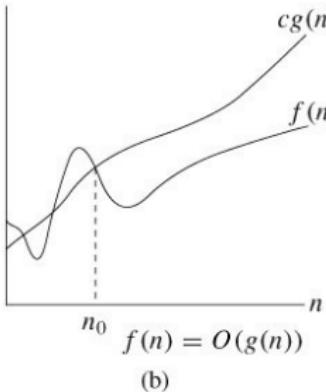
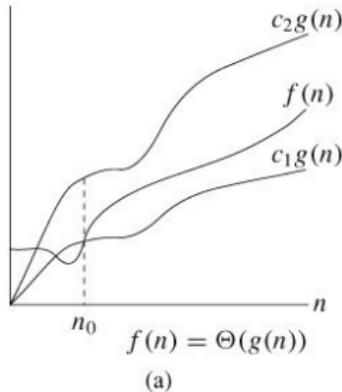
(image CLRS book)



$\rightarrow \Theta(g(N)) = \left\{ f(N) : \text{there exist positive constants } c_1, c_2, N_0 \text{ such that } 0 \leq c_1 g(N) \leq f(N) \leq c_2 g(N) \text{ for all } N \geq N_0 \right\}$

FORMAL DEFINITIONS OF ASYMPTOTIC NOTATION

(image CLRS book)

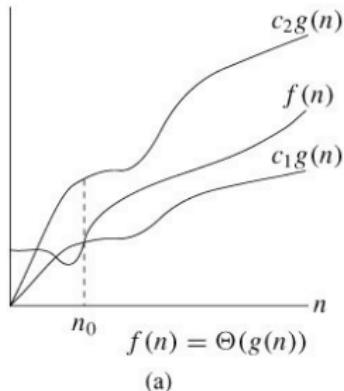


$\rightarrow \Theta(g(N)) = \left\{ f(N) : \text{there exist positive constants } c_1, c_2, N_0 \text{ such that } 0 \leq c_1 g(N) \leq f(N) \leq c_2 g(N) \text{ for all } N \geq N_0 \right\}$

$\rightarrow O(g(N)) = \left\{ f(N) : \text{there exist positive constants } c, N_0 \text{ such that } 0 \leq f(N) \leq c g(N) \text{ for all } N \geq N_0 \right\}$

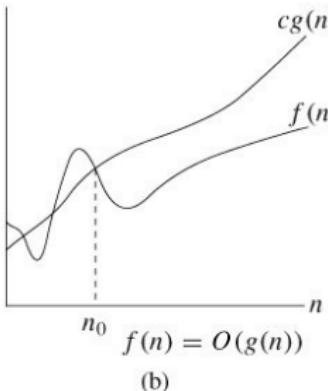
FORMAL DEFINITIONS OF ASYMPTOTIC NOTATION

(image CLRS book)



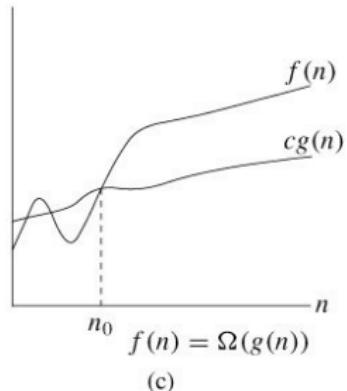
$$f(n) = \Theta(g(n))$$

(a)



$$f(n) = O(g(n))$$

(b)



$$f(n) = \Omega(g(n))$$

(c)

$\rightarrow \Theta(g(N)) = \left\{ f(N) : \text{there exist positive constants } c_1, c_2, N_0 \text{ such that } 0 \leq c_1 g(N) \leq f(N) \leq c_2 g(N) \text{ for all } N \geq N_0 \right\}$

$\rightarrow O(g(N)) = \left\{ f(N) : \text{there exist positive constants } c, N_0 \text{ such that } 0 \leq f(N) \leq c g(N) \text{ for all } N \geq N_0 \right\}$

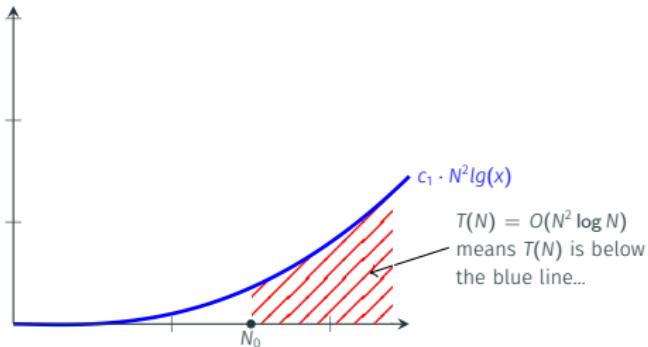
$\rightarrow \Omega(g(N)) = \left\{ f(N) : \text{there exist positive constants } c, N_0 \text{ such that } 0 \leq c_1 g(N) \leq f(N) \text{ for all } N \geq N_0 \right\}$

EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorithm` has an asymptotic running time $T(N) = O(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = \Omega(N^2 \log N)$?
- $T(N) = \Theta(N^2 \log N)$?

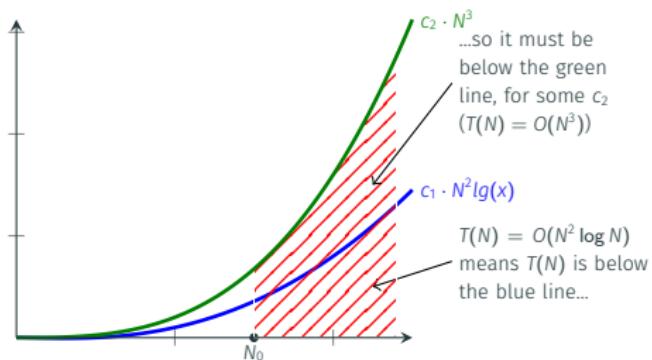


EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorithm` has an asymptotic running time $T(N) = O(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$? YES
- $T(N) = O(N^2)$? NO
- $T(N) = \Omega(N)$? NO
- $T(N) = \Omega(N^3)$? NO
- $T(N) = \Omega(N^2 \log N)$? NO
- $T(N) = \Theta(N^2 \log N)$? NO

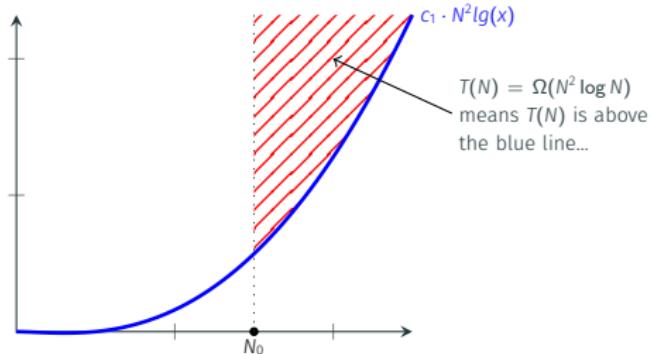


EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorithm` has an asymptotic running time $T(N) = \Omega(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = O(N^2 \log N)$?
- $T(N) = \Theta(N^2 \log N)$?

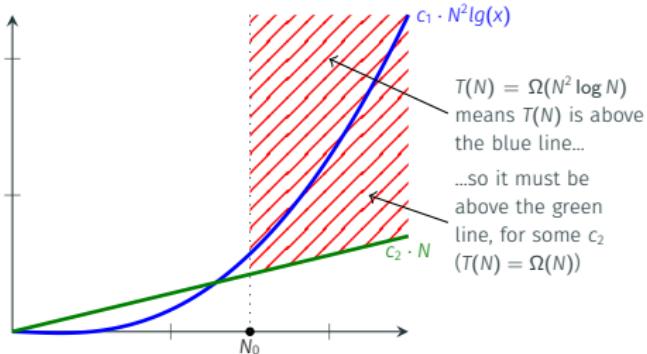


EXERCISE: NECESSARY CONCLUSIONS

Suppose myAlgorithm has an asymptotic running time $T(N) = \Omega(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$? NO
- $T(N) = O(N^2)$? NO
- $T(N) = \Omega(N)$? YES
- $T(N) = \Omega(N^3)$? NO
- $T(N) = O(N^2 \log N)$? NO
- $T(N) = \Theta(N^2 \log N)$? NO

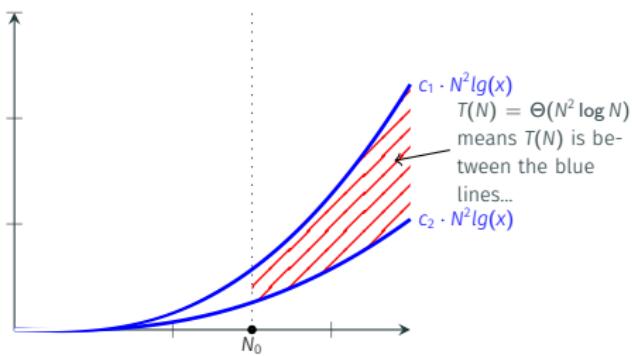


EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorithm` has an asymptotic running time $T(N) = \Theta(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = \Omega(N^2 \log N)$?
- $T(N) = O(N^2 \log N)$?

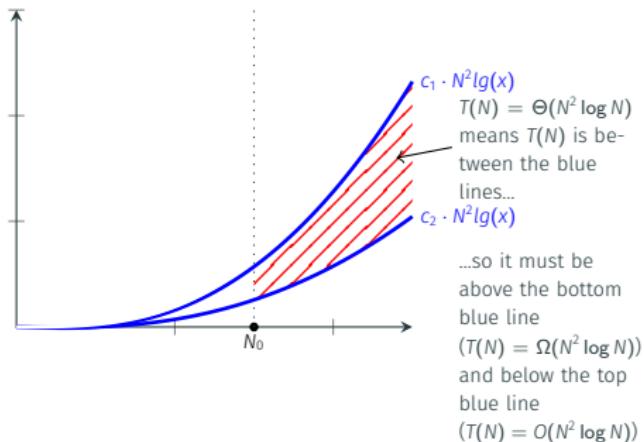


EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorithm` has an asymptotic running time $T(N) = \Theta(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$?
- $T(N) = O(N^2)$?
- $T(N) = \Omega(N)$?
- $T(N) = \Omega(N^3)$?
- $T(N) = \Omega(N^2 \log N)$? YES
- $T(N) = O(N^2 \log N)$? YES

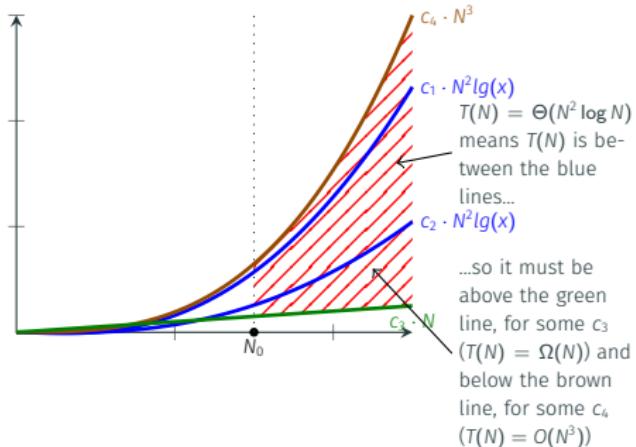


EXERCISE: NECESSARY CONCLUSIONS

Suppose `myAlgorithm` has an asymptotic running time $T(N) = \Theta(N^2 \log N)$

Does that necessarily mean

- $T(N) = O(N^3)$? YES
- $T(N) = O(N^2)$? NO
- $T(N) = \Omega(N)$? YES
- $T(N) = \Omega(N^3)$? NO
- $T(N) = \Omega(N^2 \log N)$? YES
- $T(N) = O(N^2 \log N)$? YES



- Tilde notation: an approximate model
- Asymptotic notation $O/\Theta/\Omega$: order of growth when $N \rightarrow \infty$

- Tilde notation: an approximate model
- Asymptotic notation $O/\Theta/\Omega$: order of growth when $N \rightarrow \infty$
- Common mistake 1: interpret $O/\Theta/\Omega$ as worst/average/best case running times
- Common mistake 2: interpret asymptotic notation as an approximate model

- Tilde notation: an approximate model
- Asymptotic notation $O/\Theta/\Omega$: order of growth when $N \rightarrow \infty$
- Common mistake 1: interpret $O/\Theta/\Omega$ as worst/average/best case running times
- Common mistake 2: interpret asymptotic notation as an approximate model
- Book S&W: uses the Tilde-notation.
- This module: use asymptotic notation.
 - Easy calculations because of the properties of asymptotic notation

PROPERTIES OF ASYMPTOTIC NOTATION (1)

Simplify: Constant coefficients are equivalent to 1;
Keep only highest order term

Ex:

$$\rightarrow \Theta(1) = \Theta(2) = \Theta(300)$$

$$\rightarrow \Theta(N^2) = \Theta(10N^2) = \Theta(100N^2)$$

$$\rightarrow \Theta(\lg N) = \Theta(\log_{10} N) = \Theta(\log_{20} N)$$

$$\rightarrow \Theta(N + \log N) = \Theta(N)$$

$$\rightarrow \Theta(10N^2 + 5\log N + 30) = \Theta(N^2)$$

And the same for O and Ω

Always use the simplest forms!

Addition: keep only the highest order terms

Theorem

$$\Theta(f(N)) + \Theta(g(N)) = \Theta(f(N)) \quad \text{when } f(N) \geq_{\infty} g(N)$$

Ex:

$$\rightarrow \Theta(1) + \Theta(1) = \Theta(1)$$

$$\rightarrow \Theta(N^2) + \Theta(N \log N) = \Theta(N^2)$$

$$\rightarrow \Theta(\log N) + \Theta(N \log N) = \Theta(N \log N)$$

And the same for O and Ω

Multiplication: multiply inner functions

Theorem

$$\Theta(f(N)) \times \Theta(g(N)) = \Theta(f(N) \times g(N))$$

Ex:

$$\rightarrow \Theta(1) \times \Theta(1) = \Theta(1)$$

$$\rightarrow \Theta(N^2) \times \Theta(N \log N) = \Theta(N^3 \log N)$$

$$\rightarrow \Theta(\log N) \times \Theta(2^N) = \Theta(2^N \log N)$$

And the same for O and Ω

EXAMPLE: BINARY SEARCH

```
1 public static int binarySearch(int[] a, int key) {  
2     int lo = 0, hi = a.length-1;  
3     while (lo <= hi) {  
4         int mid = lo + (hi - lo)/2;  
5         if      (key < a[mid]) hi = mid - 1;  
6         else if (key > a[mid]) lo = mid + 1;  
7         else return mid;  
8     }  
9     return -1;  
10 }
```

Asymptotic (worst case) analysis:

EXAMPLE: BINARY SEARCH

```
1 public static int binarySearch(int[] a, int key) {  
2     int lo = 0, hi = a.length-1;  
3     while (lo <= hi) {  
4         int mid = lo + (hi - lo)/2;  
5         if      (key < a[mid]) hi = mid - 1;  
6         else if (key > a[mid]) lo = mid + 1;  
7         else return mid;  
8     }  
9     return -1;  
10 }
```

Asymptotic (worst case) analysis:

Line 2: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_2 = \Theta(1)$

Lines 3 – 8: executed $\Theta(\log N)$ times, each execution takes $\Theta(1)$ time $\Rightarrow T_{3-8} = \Theta(\log N) \times \Theta(1) = \Theta(\log N)$

Lines 9 – 10: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_{9-10} = \Theta(1)$

EXAMPLE: BINARY SEARCH

```
1 public static int binarySearch(int[] a, int key) {  
2     int lo = 0, hi = a.length-1;  
3     while (lo <= hi) {  
4         int mid = lo + (hi - lo)/2;  
5         if      (key < a[mid]) hi = mid - 1;  
6         else if (key > a[mid]) lo = mid + 1;  
7         else return mid;  
8     }  
9     return -1;  
10 }
```

Asymptotic (worst case) analysis:

Line 2: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_2 = \Theta(1)$

Lines 3 – 8: executed $\Theta(\log N)$ times, each execution takes $\Theta(1)$ time $\Rightarrow T_{3-8} = \Theta(\log N) \times \Theta(1) = \Theta(\log N)$

Lines 9 – 10: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_{9-10} = \Theta(1)$

Total Running time: $T(N) = T_2 + T_{3-8} + T_{9-10} = \Theta(1) + \Theta(\log N) + \Theta(1) = \Theta(\log N)$

EXAMPLE: INSERTION SORT

```
1 public void insertion_sort(int[] a) {  
2     for (int j = 1; j < a.length; j++) {  
3         int i = j - 1;  
4         while(i >= 0 && a[i] > a[i+1]) {  
5             int temp = a[i];  
6             a[i] = a[i+1];  
7             a[i+1] = temp;  
8             i--;  
9         }  
10    }  
11 }
```

Asymptotic (worst case) analysis:

EXAMPLE: INSERTION SORT

```
1 public void insertion_sort(int[] a) {  
2     for (int j = 1; j < a.length; j++) {  
3         int i = j - 1;  
4         while(i >= 0 && a[i] > a[i+1]) {  
5             int temp = a[i];  
6             a[i] = a[i+1];  
7             a[i+1] = temp;  
8             i--;  
9         }  
10    }  
11 }
```

Asymptotic (worst case) analysis:

Lines 2,3,10: executed $\Theta(N)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_{2,3,10} = \Theta(N)$

EXAMPLE: INSERTION SORT

```
1 public void insertion_sort(int[] a) {  
2     for (int j = 1; j < a.length; j++) {  
3         int i = j - 1;  
4         while(i >= 0 && a[i] > a[i+1]) {  
5             int temp = a[i];  
6             a[i] = a[i+1];  
7             a[i+1] = temp;  
8             i--;  
9         }  
10    }  
11 }
```

Asymptotic (worst case) analysis:

Lines 2,3,10: executed $\Theta(N)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_{2,3,10} = \Theta(N)$

Lines 4 – 9: executed $\Theta(\log N^2)$ times, each execution takes $\Theta(1)$ time $\Rightarrow T_{4-9} = \Theta(N^2) \times \Theta(1) = \Theta(N^2)$

Line 11: executed $\Theta(1)$ times, execution takes $\Theta(1)$ time $\Rightarrow T_{11} = \Theta(1)$

Total Running time: $T(N) = T_{2,3,10} + T_{4-9} + T_{11} = \Theta(N) + \Theta(N^2) + \Theta(1) = \Theta(N^2)$

EXERCISE: ASYMPTOTIC ANALYSIS

A software engineer was asked to design an algorithm which will input two **unsorted** arrays of integers, A (of size N) and B (also of size N), and will output **true** when all integers in A are present in B. The engineer came up with **two** alternatives. **Which one is better?**

```
1 boolean isContained1(int[] A, int[] B) {  
2     boolean AInB = true;  
3     for (int i = 0; i < A.length; i++) {  
4         boolean iInB = linearSearch(B, A[i]);  
5         AInB = AInB && iInB;  
6     }  
7     return AInB;  
8 }
```

```
1 boolean isContained2(int[] A, int[] B) {  
2     int[] C = new int[B.length];  
3     for (int i = 0; i < B.length; i++) { C[i] = B[i] }  
4     sort(C); // heapsort  
5     boolean AInC = true;  
6     for (int i = 0; i < A.length; i++) {  
7         boolean iInC = binarySearch(C, A[i]);  
8         AInC = AInC && iInC;  
9     }  
10    return AInC;  
11 }
```

EXERCISE: COMPARISONS

Write the following asymptotic order of growths in ascending order, from the most to the least efficient, using $<$ or $=$ to show the equivalences and inequivalences between them.

$$\Theta(N \log N)$$

$$\Theta(N)$$

$$\Theta(N^2 + 3N + 1)$$

$$\Theta(1)$$

$$\Theta(5N)$$

$$\Theta(N^3 + \log N)$$

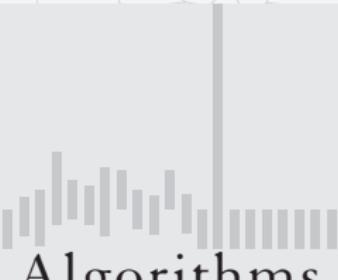
$$\Theta(N^2)$$

$$\Theta(10)$$

$$\Theta(10N^3 + 2 \lg(N))$$

$$\Theta(10 \lg(N))$$

$$\Theta(2^N)$$



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ ***theory of algorithms***
- ▶ *memory*

Types of analyses

~~Best case.~~ Lower bound on cost.

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

~~Worst case.~~ Upper bound on cost.

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

~~Average case.~~ Expected cost for random input.

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



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

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

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

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

Ex 2. Comparisons for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

Theory of algorithms

Goals.

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

Approach.

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

Upper bound. Performance guarantee of algorithm for any input.

Lower bound. Proof that no algorithm can do better. (for worst case inputs)

Optimal algorithm. Lower bound = upper bound (to within a constant factor).

Theory of algorithms: example 1

Goals.

- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = *“Is there a 0 in the array?”*

Upper bound.

A specific algorithm.

- Ex. Brute-force algorithm for 1-SUM: Look at every array entry.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound.

Proof that no algorithm can do better.

- Ex. Have to examine all N entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$. (for worst case inputs)

Optimal algorithm.

- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.

Theory of algorithms: example 2

Goals.

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

Upper bound.

A specific algorithm.

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

Theory of algorithms: example 2

Goals.

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

Upper bound.

A specific algorithm.

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

Lower bound.

Proof that no algorithm can do better.

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

Open problems.

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

Algorithm design approach

Start.

- Develop an algorithm.
- Prove a lower bound.

Gap?

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

Golden Age of Algorithm Design.

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

Caveats.

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



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*
- ▶ *memory*

Basics

Bit. 0 or 1.

NIST

most computer scientists



Byte. 8 bits.

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

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



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

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



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



Typical memory usage for primitive types and arrays

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

primitive types

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

one-dimensional arrays

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

two-dimensional arrays

Typical memory usage for objects in Java

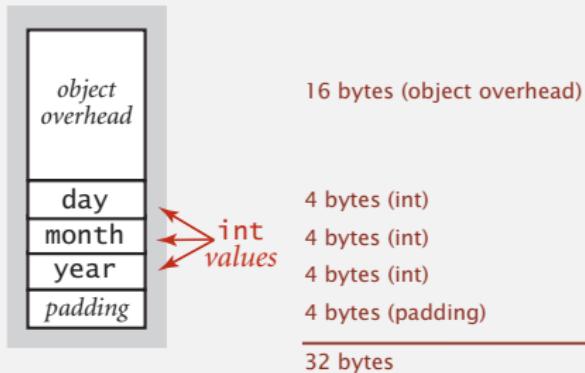
Object overhead. 16 bytes.

Reference. 8 bytes.

Padding. Each object uses a multiple of 8 bytes.

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

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



Typical memory usage summary

Total memory usage for a data type value:

- Primitive type: 4 bytes for int, 8 bytes for double, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

+ 8 extra bytes per inner class object
(for reference to enclosing class)

Shallow memory usage: Don't count referenced objects.

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

Example

Q. How much memory does WeightedQuickUnionUF use as a function of N ?

Use tilde notation to simplify your answer.

```
public class WeightedQuickUnionUF
{
    private int[] id;
    private int[] sz;
    private int count;

    public WeightedQuickUnionUF(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;
    }
    ...
}
```

16 bytes
(object overhead)

8 + (4N + 24) bytes each
(reference + int[] array)

4 bytes (int)

4 bytes (padding)

8N + 88 bytes

A. $8N + 88 \sim 8N$ bytes.

Turning the crank: summary

Empirical analysis.

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

Mathematical analysis.

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



Scientific method.

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

CS2011: ALGORITHMS AND DATA STRUCTURES I

Lecture 5: Doubly Linked Lists

Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Java collections library

public interface **java.util.List<E>**

boolean **add(int index, E element)**

Inserts the specified element at the specified position in this list (optional operation).

E **remove(int index)**

Removes the element at the specified position in this list

...

Implementations:

- **LinkedList** doubly-linked list implementation
- **ArrayList** resizable-array implementation

Example client

```
public static void main(String[] args)
{
    StackOfStrings buffer = new StackQueue<String>();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("<"))
            StdOut.print(buffer.pop());
        else if (s.equals(">"))
            StdOut.print(buffer.dequeue());
        else
            stack.enqueue(s);
    }
}
```

pop: return and remove the **most recent** element

dequeue: return and remove the **least recent** element

enqueue: add an element

Example client

```
public static void main(String[] args)
{
    StackOfStrings buffer = new StackQueue<String>();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("<"))
            StdOut.print(buffer.pop());
        else if (s.equals(">"))
            StdOut.print(buffer.dequeue());
        else
            stack.enqueue(s);
    }
}
```

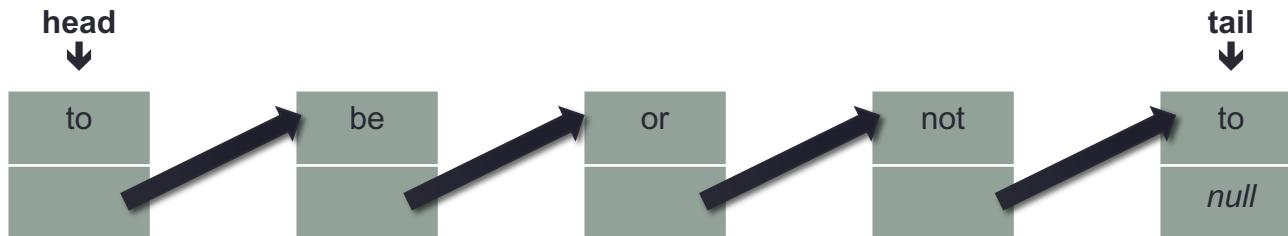
Input: “judge me by my size do > you > < < < < ? >”



Example client

```
public static void main(String[] args)
{
    StackOfStrings buffer = new StackQueue<String>();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("<"))
            StdOut.print(buffer.pop());
        else if (s.equals(">"))
            StdOut.print(buffer.dequeue());
        else
            stack.enqueue(s);
    }
}
```

Can we implement StackQueue efficiently using a **linked list**?



Example client

```
public static void main(String[] args)
{
    StackOfStrings buffer = new StackQueue<String>();
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readString();
        if (s.equals("<"))
            StdOut.print(buffer.pop());
        else if (s.equals(">"))
            StdOut.print(buffer.dequeue());
        else
            stack.enqueue(s);
    }
}
```

Can we implement StackQueue efficiently using a **linked list**?

- enqueue and push to the same end of the LL (will run in $\Theta(1)$)
- pop from the same end of the LL & dequeue from the other end
 - one of dequeue/pop will run in $\Theta(1)$ and the other in $\Theta(N)$

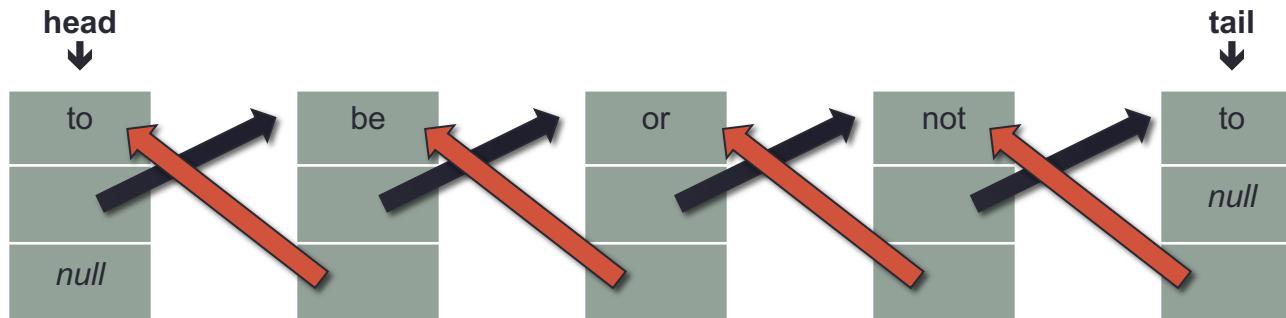
we can do better

Doubly Linked Lists

(Java implementation in Assignment 2)

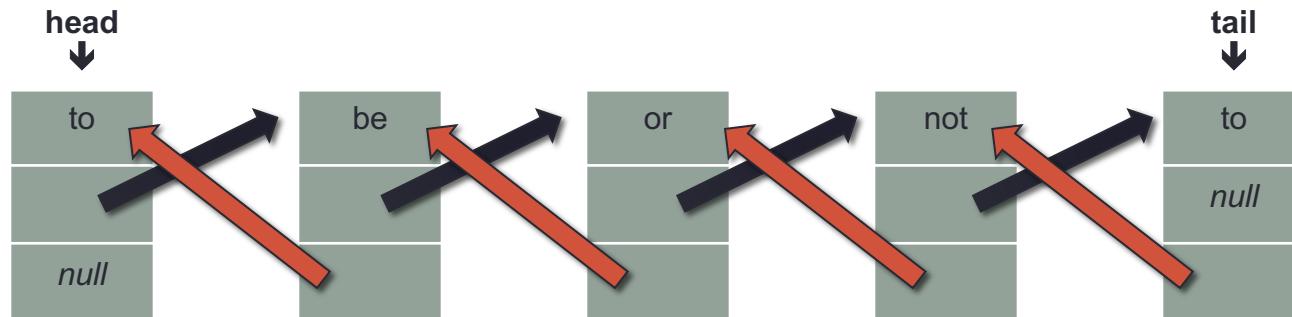
<https://www.scss.tcd.ie/Vasileios.Koutavas/teaching/cs2010/mt1819/assignment-2/>

Doubly Linked Lists (DLL)



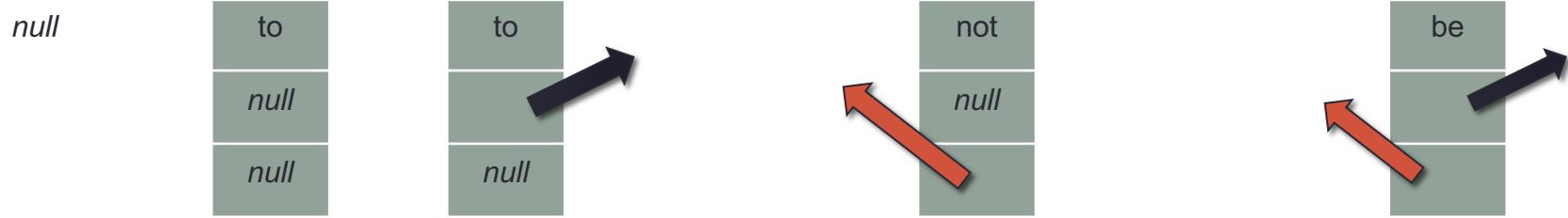
```
class DLLNode {  
    String item;  
    DLLNode next;  
    DLLNode prev;  
}
```

Doubly Linked Lists (DLL)



```
class DLLNode {
    String item;
    DLLNode next;
    DLLNode prev;
}
```

Kinds of nodes inside a DLL:



Empty
DLL

The **only**
node of a
DLL with 1
node

The **first**
node of a
DLL with
>1 node

The **last**
node of a
DLL with
>1 node

A **middle**
node of a
DLL with
>2 items

Doubly Linked Lists

class **DLLofString**

 DoublyLinkedList()

void	insertFirst(String s)	<i>inserts s at the head of the list</i>
String	getFirst()	<i>returns string at the head of the list</i>
boolean	deleteFirst()	<i>removes string at the head of the list</i>
void	insertLast(String s)	<i>inserts s at the end of the list</i>
String	getLast(String s)	<i>returns string at the end of the list</i>
boolean	deleteLast()	<i>removes string at the end of the list</i>
void	insertBefore(int pos, String s)	<i>inserts s before position pos</i>
String	get(int pos)	<i>returns string at position pos</i>
boolean	deleteAt(int pos)	<i>deletes string at position pos</i>

- Interface somewhat different than that of `java.util.List`.
- How to make interface generic?
 - class `DoublyLinkedList<T>`

Doubly Linked Lists

class DLLofString

 DoublyLinkedList()

void insertFirst(String s)

inserts s at the head of the list

String getFirst()

returns string at the head of the list

boolean deleteFirst()

removes string at the head of the list

void insertLast(String s)

inserts s at the end of the list

String getLast(String s)

returns string at the end of the list

boolean deleteLast()

removes string at the end of the list

void insertBefore(int pos, String s)

inserts s before position pos

String get(int pos)

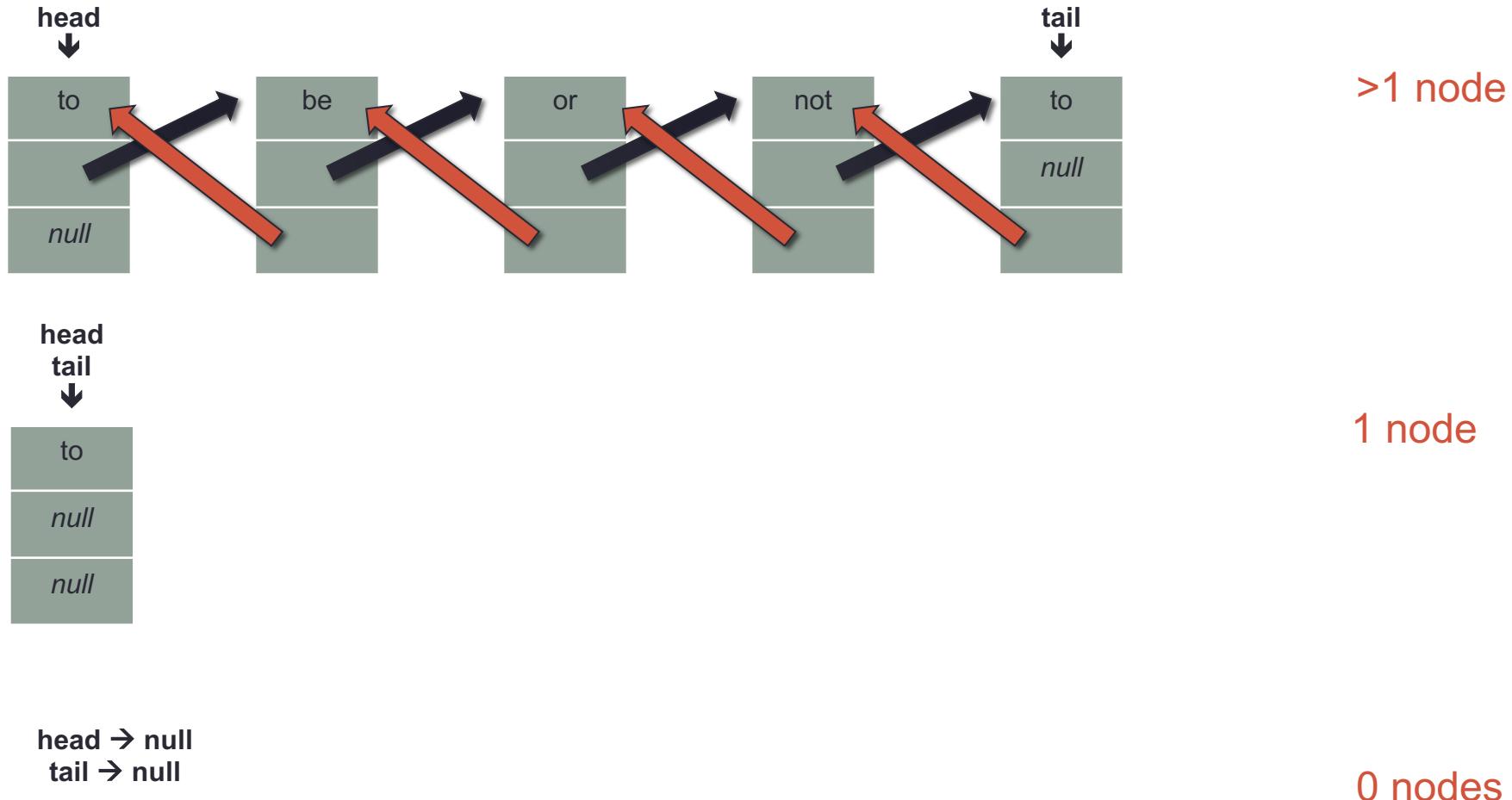
returns string at position pos

boolean deleteAt(int pos)

deletes string at position pos

- Interface somewhat different than that of java.util.List.
- How to make interface generic?
 - class DoublyLinkedList<T>

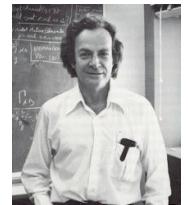
Three main cases to deal with



```
/**  
 * Inserts an element at the end of the doubly linked list  
 * @param data : The new data of class T that needs to be added to the list  
 * @return none  
 *  
 */  
public void insertLast( T data )
```

void insertLast("be")

DLL of size 0	?
DLL of size 1	?
DLL of size > 1	?

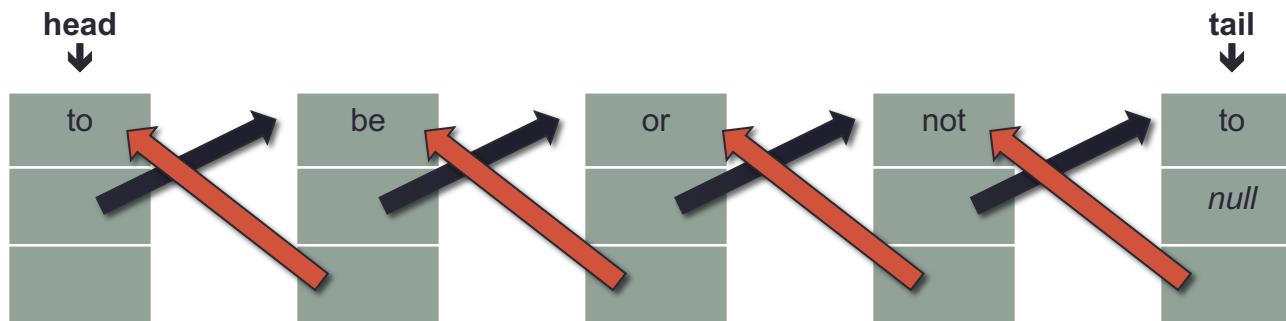


Before you write any code
consider simple examples,
but not too simple!

(paraphrase of Richard Feynman)

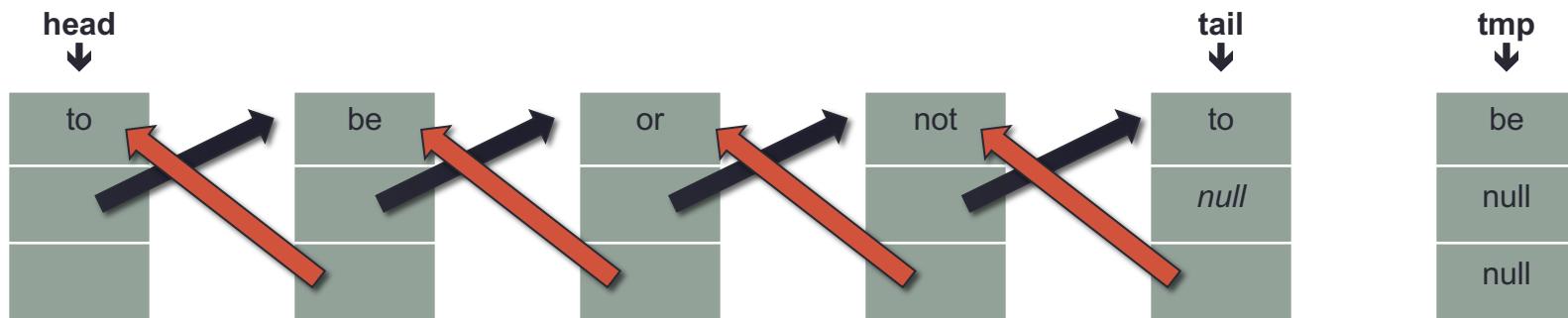
insertLast("be")

>1 node



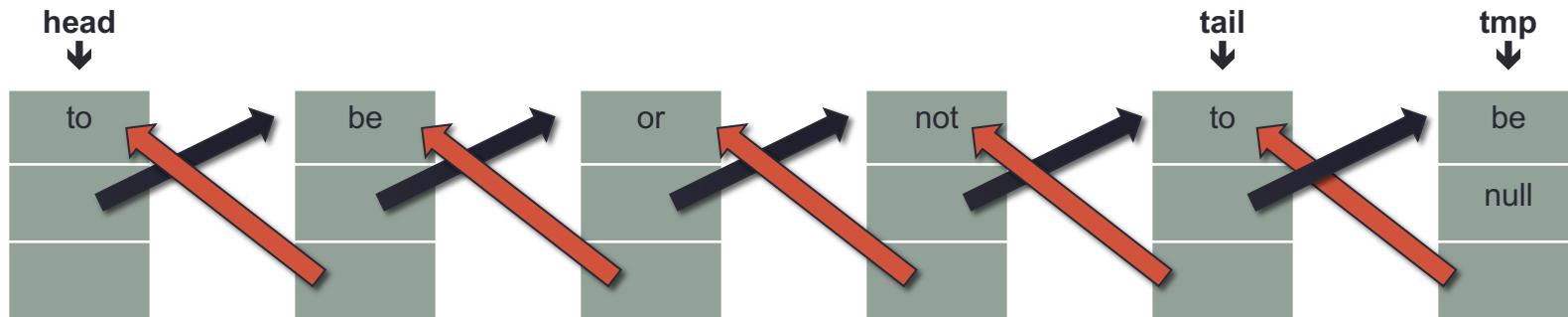
insertLast("be")

>1 node



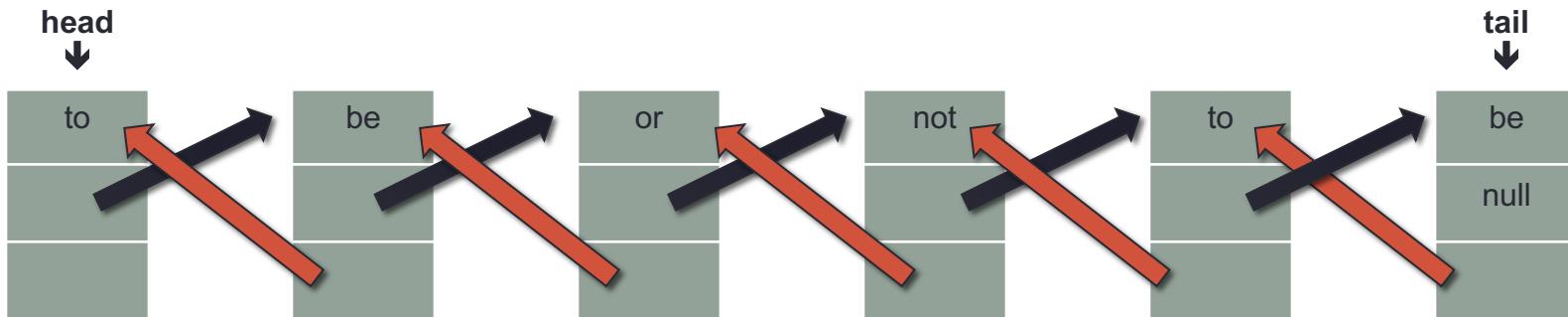
insertLast("be")

>1 node



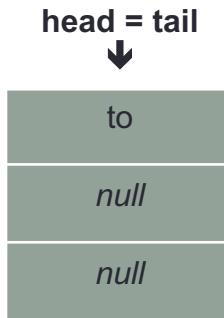
insertLast("be")

>1 node



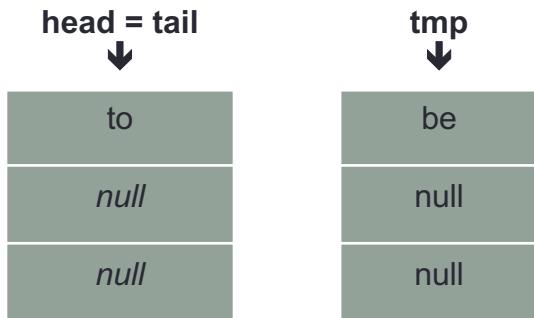
insertLast("be")

1 nodes



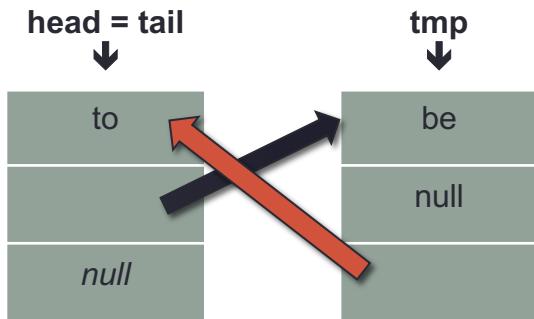
insertLast("be")

1 node



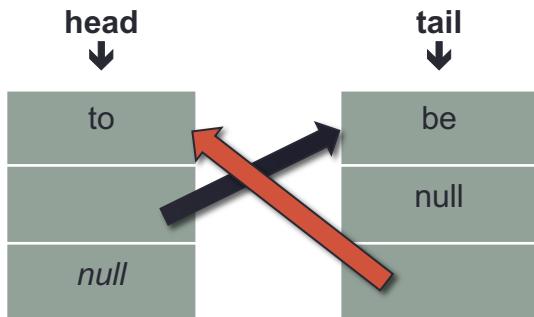
insertLast("be")

1 node



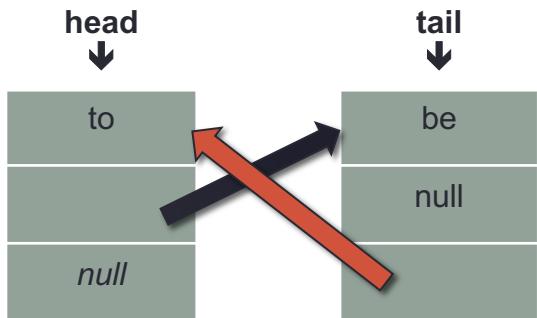
insertLast("be")

1 node



insertLast("be")

1 node



* Same as the case with >1 node

insertLast("be")

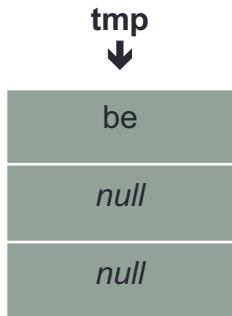
0 nodes

head → null tail → null

insertLast("be")

0 nodes

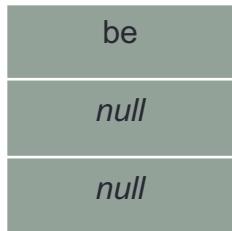
head → null tail → null



insertLast("be")

0 nodes

head = tail
↓



```
/**  
 * Inserts an element at the beginning of the doubly linked list  
 * @param data : The new data of class T that needs to be added to the list  
 * @return none  
 *  
 */  
public void insertFirst( T data )
```

how can we implement this?

void insertFirst("be")

DLL of size 0	?	write down an example for each case
DLL of size 1	?	
DLL of size > 1	?	

cases for size of the DLL

```
/**  
 * Inserts an element in the doubly linked list  
 * @param pos : The integer location at which the new data should be  
 *      inserted in the list. We assume that the first position in the list  
 *      is 0 (zero). If pos is less than 0 then add to the head of the list.  
 *      If pos is greater or equal to the size of the list then add the  
 *      element at the end of the list.  
 * @param data : The new data of class T that needs to be added to the list  
 * @return none  
 *  
 */  
public void insertBefore( int pos, T data )
```

void insertBefore(pos, “be”)

	pos == 0 (insert at the head of the DLL)	0 < pos < DLL.size -1 (insert in the middle of the DLL)	pos == DLL.size -1	pos < 0 (insert at the head of the DLL)	pos ≥ DLL.size (insert at the end of the DLL)
DLL of size 0					
DLL of size 1					
DLL of size > 1					

void insertBefore(pos, “be”)

	pos == 0 (insert at the head of the DLL)	0 < pos < DLL.size -1 (insert in the middle of the DLL)	pos == DLL.size -1	pos < 0 (insert at the head of the DLL)	pos ≥ DLL.size (insert at the end of the DLL)
DLL of size 0	insertFirst("be")				
DLL of size 1	insertFirst("be")				
DLL of size > 1	insertFirst("be")				

void insertBefore(pos, “be”)

	pos == 0 (insert at the head of the DLL)	0 < pos < DLL.size -1 (insert in the middle of the DLL)	pos == DLL.size -1	pos < 0 (insert at the head of the DLL)	pos ≥ DLL.size (insert at the end of the DLL)
DLL of size 0	insertFirst(“be”)	<i>not possible</i> $0 < pos < -1$			
DLL of size 1	insertFirst(“be”)	<i>not possible</i> $0 < pos < 0$			
DLL of size > 1	insertFirst(“be”)	?			

void insertBefore(pos, “be”)

	pos == 0 (insert at the head of the DLL)	0 < pos < DLL.size -1 (insert in the middle of the DLL)	pos == DLL.size -1	pos < 0 (insert at the head of the DLL)	pos ≥ DLL.size (insert at the end of the DLL)
DLL of size 0	insertFirst(“be”)	<i>not possible</i> $0 < pos < -1$	insertFirst(“be”) (here pos == -1)		
DLL of size 1	insertFirst(“be”)	<i>not possible</i> $0 < pos < 0$	insertFirst(“be”) (here pos == 0)		
DLL of size > 1	insertFirst(“be”)	?	?		

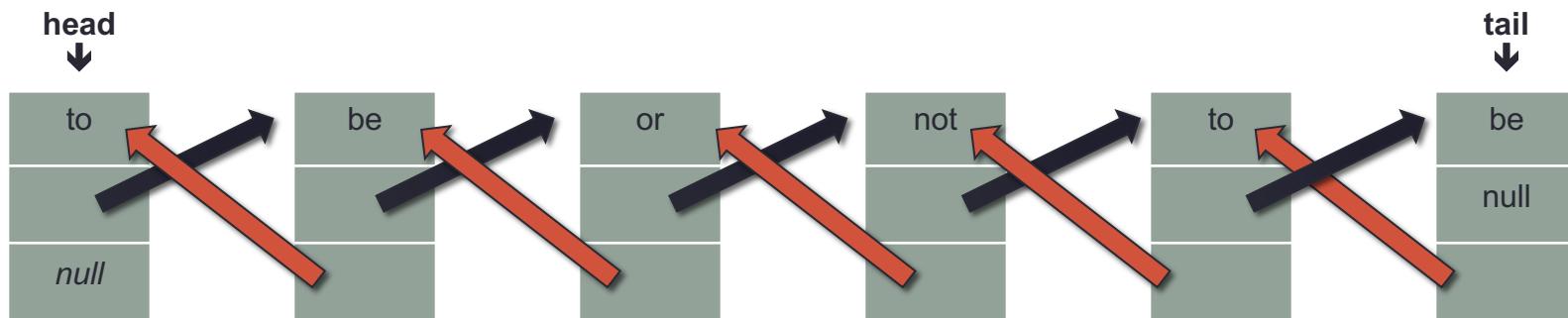
void insertBefore(pos, “be”)

	pos == 0 (insert at the head of the DLL)	0 < pos < DLL.size -1 (insert in the middle of the DLL)	pos == DLL.size -1	pos < 0 (insert at the head of the DLL)	pos ≥ DLL.size (insert at the end of the DLL)
DLL of size 0	insertFirst("be")	<i>not possible</i> $0 < pos < -1$	insertFirst("be") (here pos == -1)	insertFirst("be")	InsertLast("be")
DLL of size 1	insertFirst("be")	<i>not possible</i> $0 < pos < 0$	insertFirst("be") (here pos == 0)	insertFirst("be")	InsertLast("be")
DLL of size > 1	insertFirst("be")	?	?	insertFirst("be")	InsertLast("be")

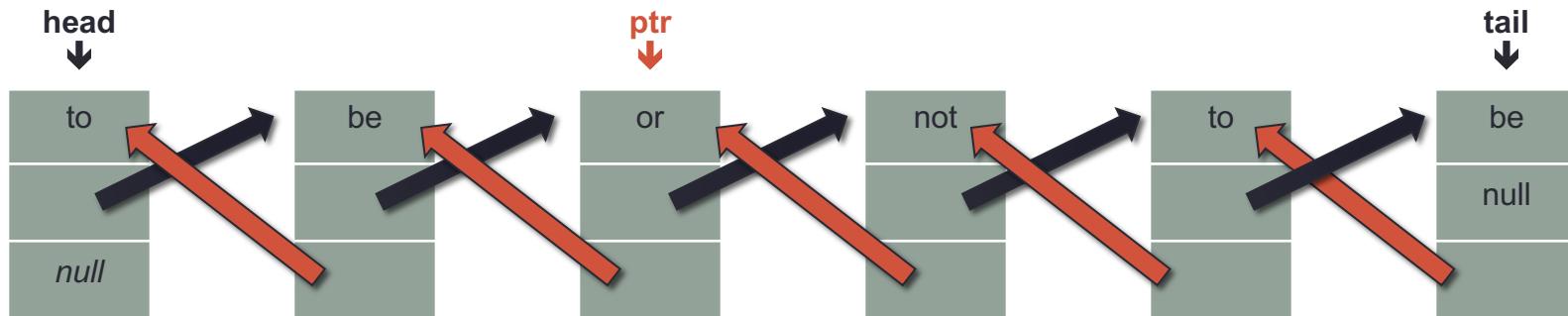
void insertBefore(pos, “be”)

	pos == 0 (insert at the head of the DLL)	0 < pos < DLL.size -1 (insert in the middle of the DLL)	pos == DLL.size -1	pos < 0 (insert at the head of the DLL)	pos ≥ DLL.size (insert at the end of the DLL)
DLL of size 0	insertFirst("be")	<i>not possible 0 < pos < -1</i>	insertFirst("be") (here pos == -1)	insertFirst("be")	InsertLast("be")
DLL of size 1	insertFirst("be")	<i>not possible 0 < pos < 0</i>	insertFirst("be") (here pos == 0)	insertFirst("be")	InsertLast("be")
DLL of size > 1	insertFirst("be")	?	same as case to the left	insertFirst("be")	InsertLast("be")

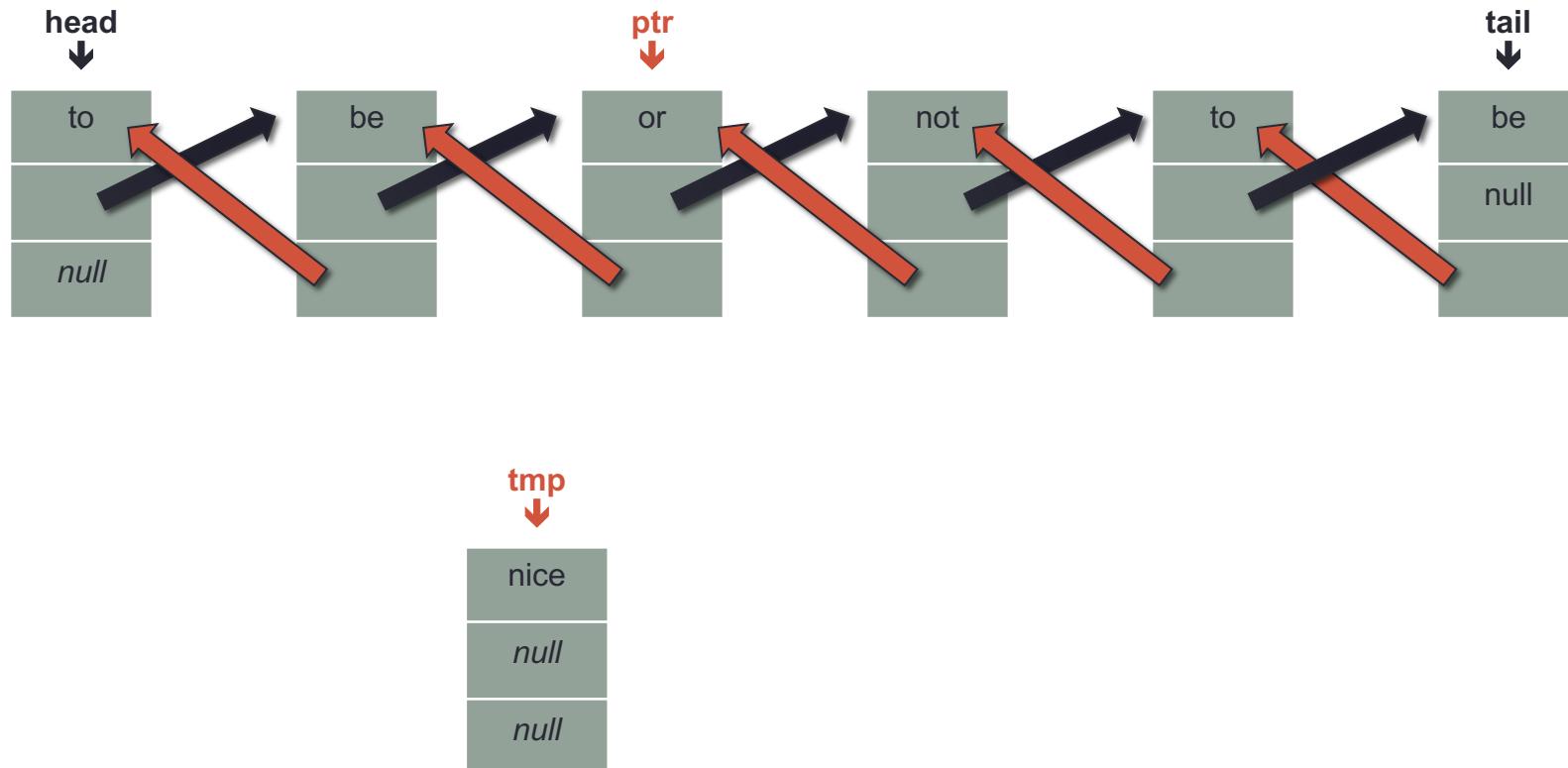
insertBefore(2, “nice”)



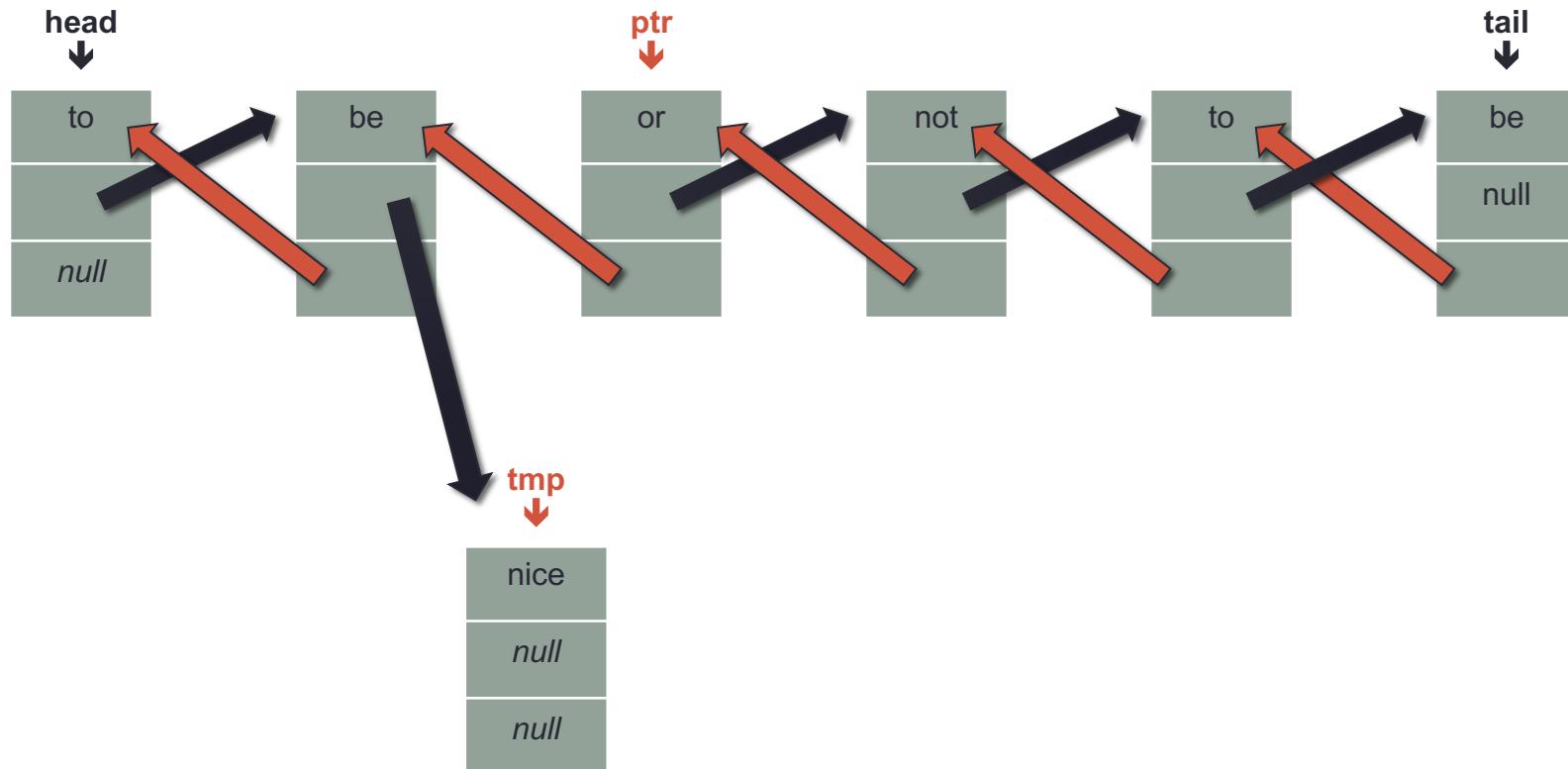
insertBefore(2, “nice”)



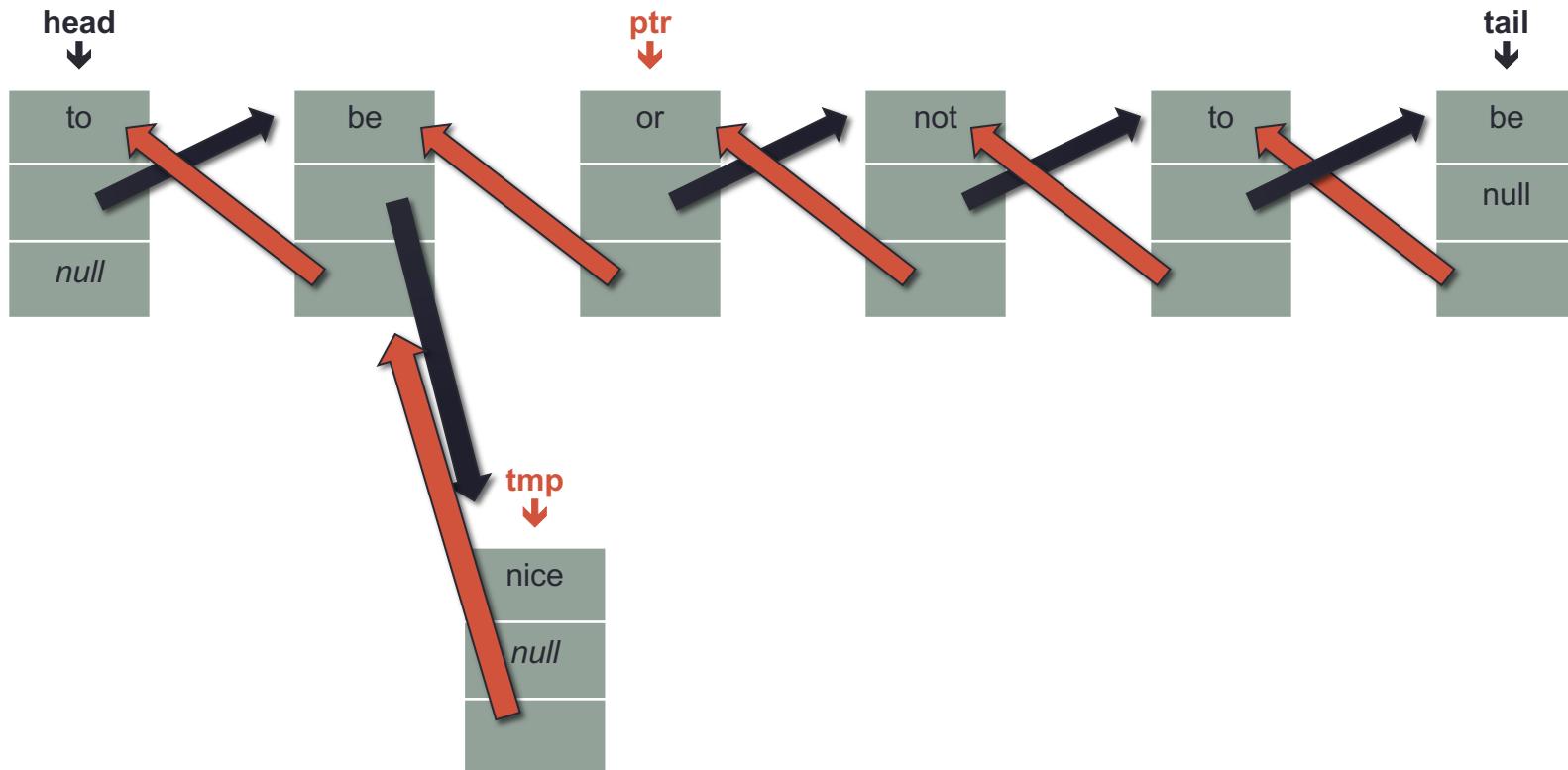
insertBefore(2, "nice")



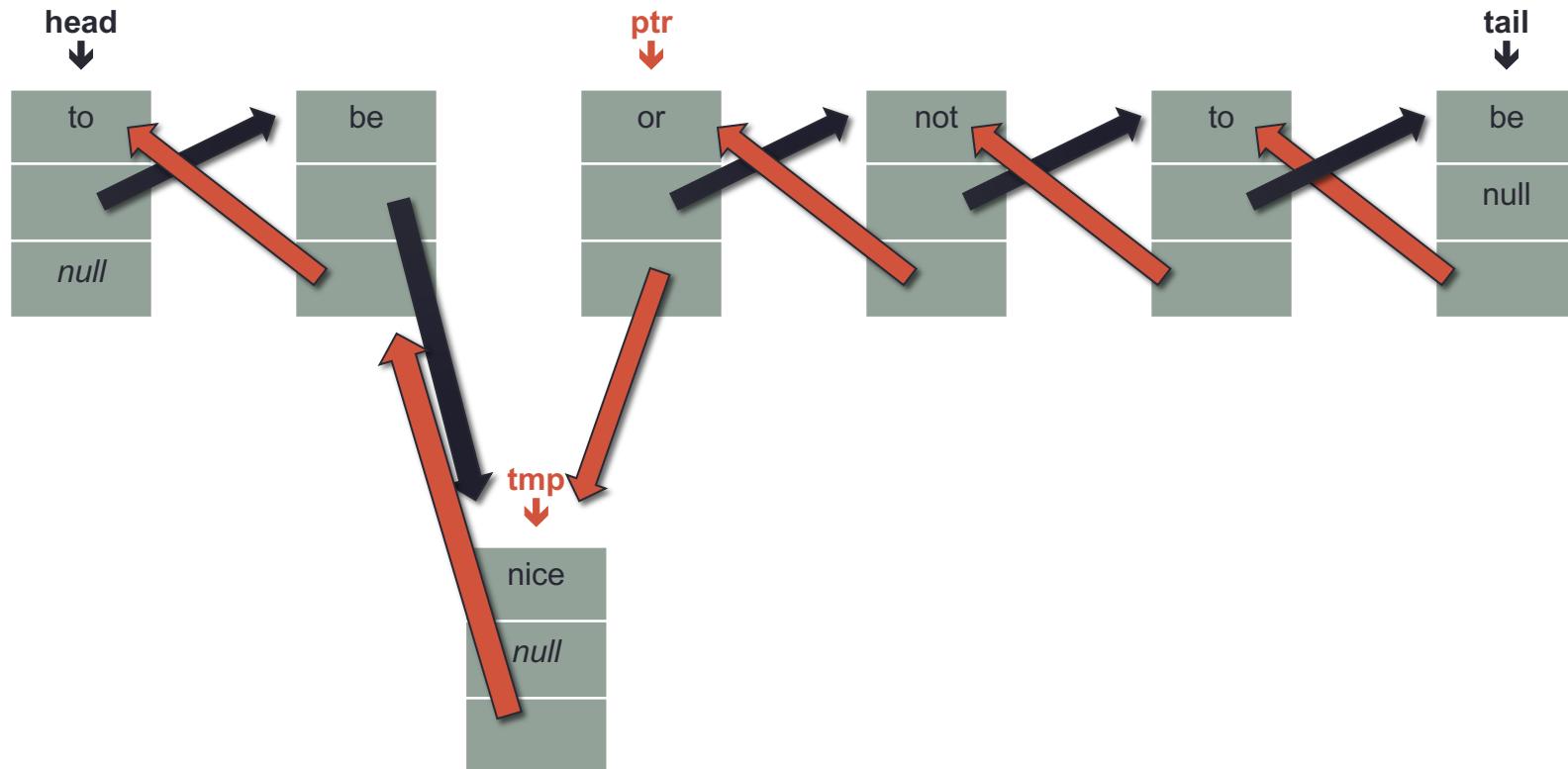
insertBefore(2, "nice")



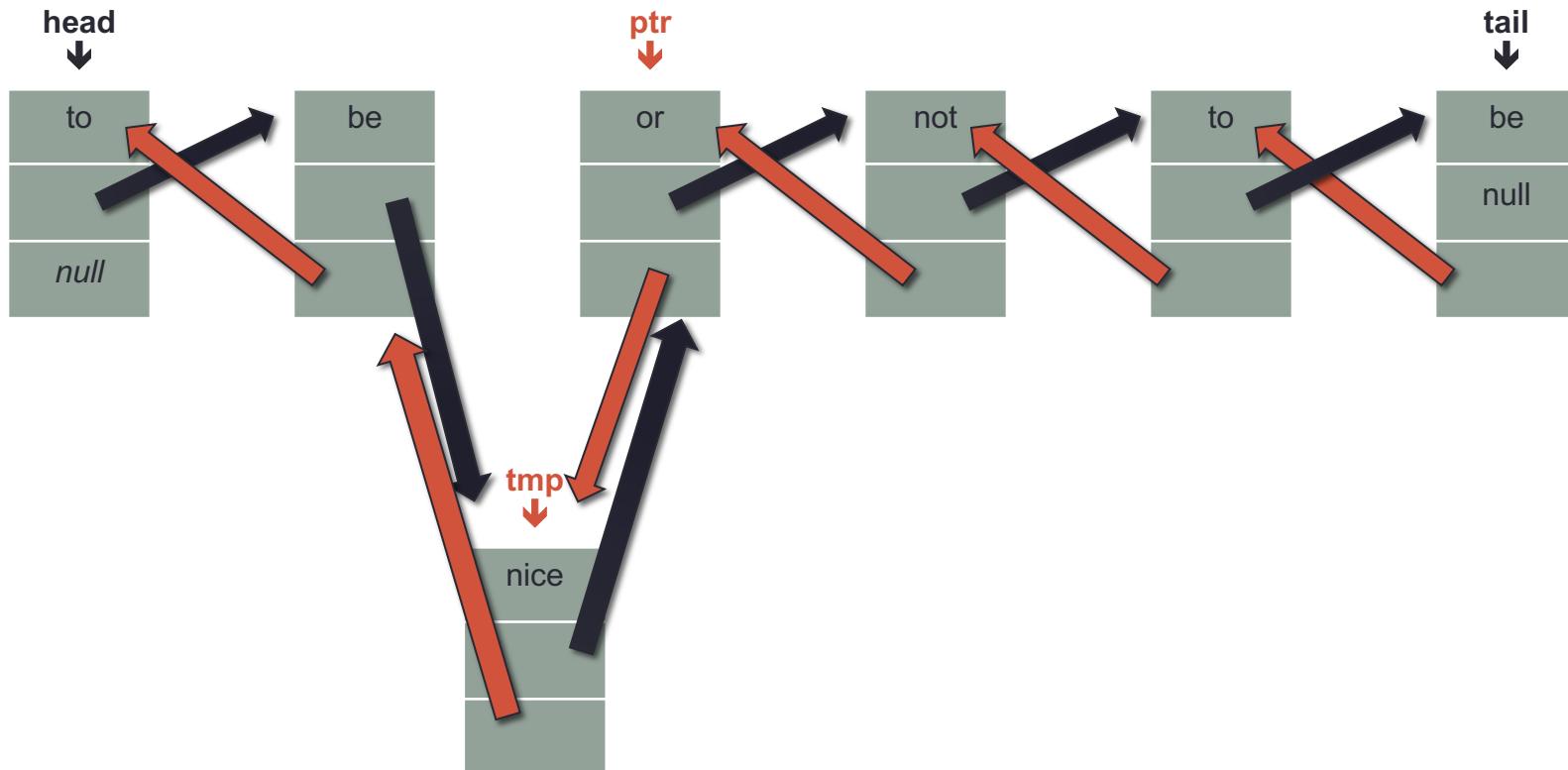
insertBefore(2, “nice”)



insertBefore(2, "nice")



insertBefore(2, "nice")



Moral of the story

Before you implement an operation of a complex Data Structure (DS):

- For each different meaningful **state** your DS may be in:
 - For each different meaningful **input** to your DS operation:
 - Create one example **on paper**
 - Work out the example **on paper**
 - **Reuse** code from previous, simpler examples
 - **Group cases** whose examples require the same steps
 - **ONLY then write code**

CSU22011: ALGORITHMS AND DATA STRUCTURES I

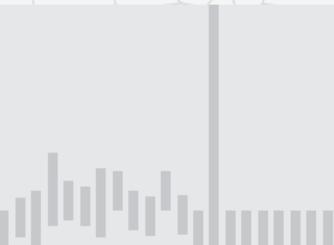
Lecture 6: Java Generics & Iterators

Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

JAVA GENERICS



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans,

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.

(Java 1.5 released Sep 2004)



Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans,

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.



Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfVans,

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, StackOfInts, StackOfVans,

Attempt 3. Java generics.

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

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

type parameter

compile-time error

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

Generic stack: linked-list implementation

```
public class LinkedStackOfStrings
{
    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 FixedCapacityStackOfStrings
{
    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]; }
}
```

the way it should be

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

    public FixedCapacityStack(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]; }
}
```

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

Generic stack: array implementation

```
public class FixedCapacityStackOfStrings
{
    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]; }
}
```

the way it is

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

    public FixedCapacityStack(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 ugly cast

Unchecked cast

```
% javac FixedCapacityStack.java
```

```
Note: FixedCapacityStack.java uses unchecked or unsafe operations.
```

```
Note: Recompile with -Xlint:unchecked for details.
```

```
% javac -Xlint:unchecked FixedCapacityStack.java
```

```
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
```

```
found   : java.lang.Object[]
```

```
required: Item[]
```

```
    a = (Item[]) new Object[capacity];  
           ^
```

```
1 warning
```

Q. Why does Java make me cast (or use reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about **type erasure** and **covariant arrays**.



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.

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

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

GENERIC ITERATORS



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

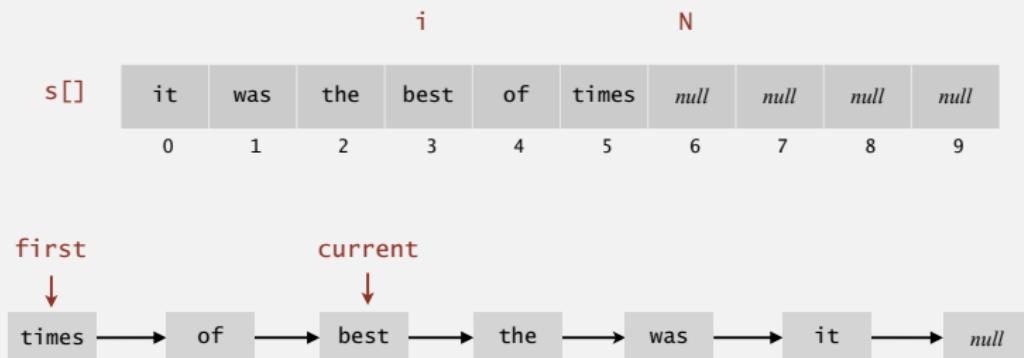
<http://algs4.cs.princeton.edu>

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *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 implement the `java.lang.Iterable` interface.

Iterators

Q. What is an `Iterable` ?

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

`java.lang.Iterable interface`

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

Q. What is an `Iterator` ?

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

`java.util.Iterator interface`

```
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 (shorthand)

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

equivalent code (longhand)

```
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;
        }
    }
}
```

throw UnsupportedOperationException
throw NoSuchElementException
if no more items in iteration



Stack iterator: array implementation

```
import java.util.Iterator;

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

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

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

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

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

Iteration: concurrent modification

Q. What if client modifies the data structure while iterating?

A. A fail-fast iterator throws a `java.util.ConcurrentModificationException`.

concurrent modification

```
for (String s : stack)
    stack.push(s);
```

Q. How to detect?

A.

- Count total number of `push()` and `pop()` operations in Stack.
- Save counts in *Iterator subclass upon creation.
- If, when calling `next()` and `hasNext()`, the current counts do not equal the saved counts, throw exception.

GENERIC COMPARISONS

COMPARISONS

Design Challenge: Add a `search` method in the Stack ADT.

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

    private class Node
    {
        Item item;
        Node next;
    }

    ...
    boolean search(Item searchObj)
    {
        // we need to compare searchObj to other items in the Stack
        ...
    }
}
```

`Item` needs to at least implement the comparable interface.

JAVA'S COMPARABLE INTERFACE

```
public interface Comparable<T>
{
    int compareTo(T o);
    // Compares this object with objects of class T.
}
```

i.compareTo(o) returns:

- 0 if i = o
- >0 if i > o
- <0 if i < o

Comparable is a **parametric** interface because it doesn't know a priori the type T.

COMPARISONS

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

    private class Node
    {
        Item item;
        Node next;
    }

    ...
    boolean search(Item searchObj)
    {
        for (Item i : this)
        {
            if (i.compareTo(searchObj) == 0) return true;
        }
        return false;
    }
}
```

CSU22011: ALGORITHMS AND DATA STRUCTURES I

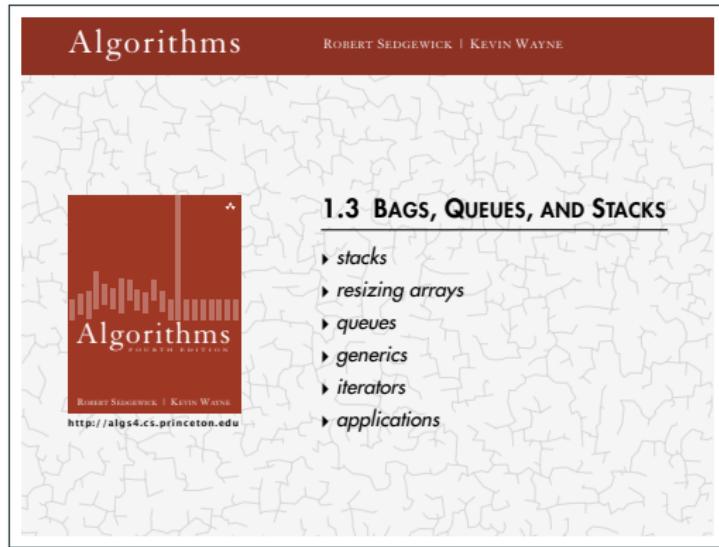
Lecture 7: Abstract Data Types - Stack & Queue

Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

THIS LECTURE



- Abstract Data Types
- Stacks and Queues *
- S&W 1.2 and 1.3

ABSTRACT DATA TYPES

ABSTRACT DATA TYPE

Example:

public class Counter	
Counter(String id)	<i>create a counter named id</i>
void increment()	<i>increment the counter by one</i>
int tally()	<i>number of increments since creation</i>
String toString()	<i>string representation</i>

A **Data Type** is

- A set of values
 - in example: all counter objects at state 0, 1, 2, ...
- A set of operations on those values
 - in example: **constructor**, **increment**, **tally**, **toString**

ABSTRACT DATA TYPE

Example:

public class Counter	
Counter(String id)	<i>create a counter named id</i>
void increment()	<i>increment the counter by one</i>
int tally()	<i>number of increments since creation</i>
String toString()	<i>string representation</i>

A **Data Type** is

- A set of values
 - in example: all counter objects at state 0, 1, 2, ...
- A set of operations on those values
 - in example: **constructor**, **increment**, **tally**, **toString**

An **Abstract Data Type** (ADT) is

- A Data Type whose implementation is unknown to the **client** of the ADT

ABSTRACT DATA TYPE

Example:

public class Counter	
Counter(String id)	<i>create a counter named id</i>
void increment()	<i>increment the counter by one</i>
int tally()	<i>number of increments since creation</i>
String toString()	<i>string representation</i>

A Data Type is

- A set of values
 - in example: all counter objects at state 0, 1, 2, ...
- A set of operations on those values
 - in example: **constructor**, **increment**, **tally**, **toString**

An Abstract Data Type (ADT) is

- A Data Type whose implementation is unknown to the client of the ADT

An Application Programming Interface (API) is

- A list and informal description of the operations of an ADT

WHO IS THE CLIENT OF AN ADT?

Example:

```
test client → public static void main(String[] args)
{
    Counter heads = new Counter("heads");
    Counter tails = new Counter("tails");
    heads.increment();
    heads.increment();
    tails.increment();
    StdOut.println(heads + " " + tails);
    StdOut.println(heads.tally() + tails.tally());
}
```

create and initialize objects

invoke constructor

automatically invoke toString()

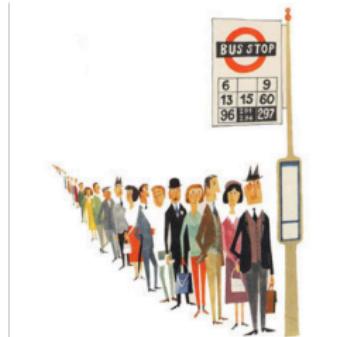
object name

invoke

→ **Client:** the rest of the program, using the ADT

STACKS & QUEUES

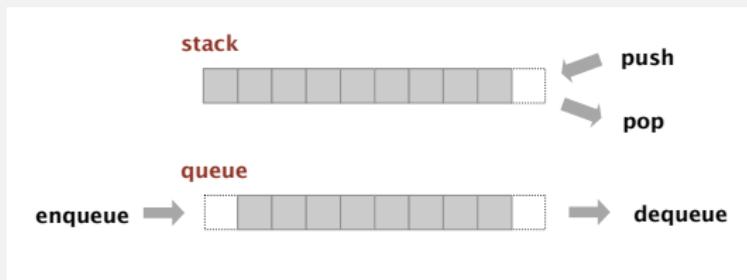
STACKS & QUEUES



Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: **insert**, **remove**, **iterate**, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. ← LIFO = "last in first out"

Queue. Examine the item least recently added. ← FIFO = "first in first out"

Client, implementation, interface

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find,

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ 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.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

Stack API

Warmup API. Stack of strings data type.

```
public class StackOfStrings
```

```
    StackOfStrings()
```

create an empty stack

```
    void push(String item)
```

insert a new string onto stack

```
    String pop()
```

*remove and return the string
most recently added*

```
    boolean isEmpty()
```

is the stack empty?

```
    int size()
```

number of strings on the stack

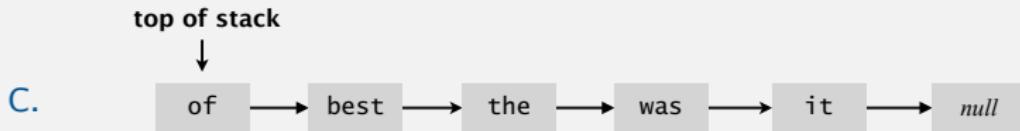
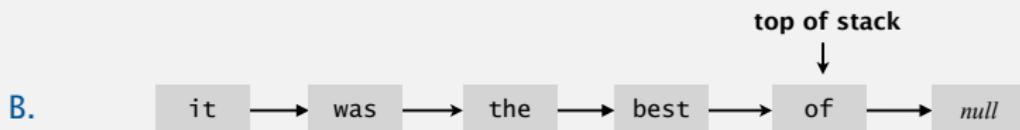
push pop



Warmup client. Reverse sequence of strings from standard input.

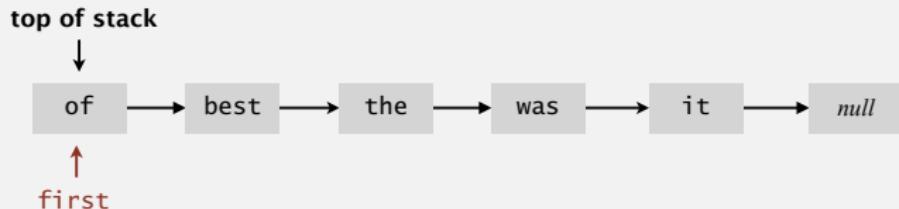
How to implement a stack with a linked list?

- A. Can't be done efficiently with a singly-linked list.



Stack: linked-list implementation

- Maintain pointer `first` to first node in a singly-linked list.
- Push new item before `first`.
- Pop item from `first`.



STACK: LINKED LIST IMPLEMENTATION

<http://dsvproject.github.io/dsvproject/code/stackLinkedList.html>

Stack pop: linked-list implementation

inner class

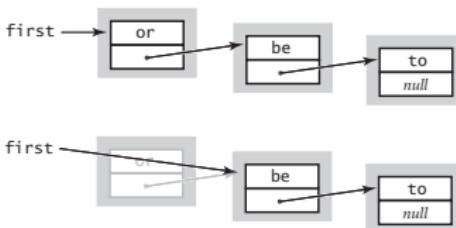
```
private class Node  
{  
    String item;  
    Node next;  
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

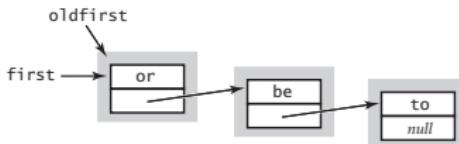
Stack push: linked-list implementation

inner class

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

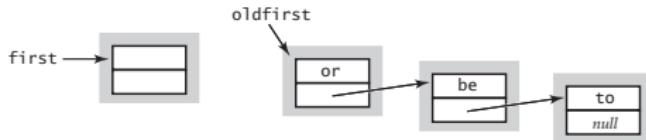
save a link to the list

```
Node oldfirst = first;
```



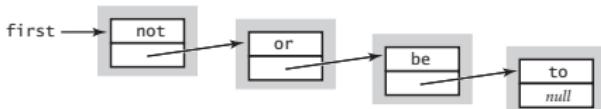
create a new node for the beginning

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



set the instance variables in the new node

```
first.item = "not";  
first.next = oldfirst;
```



Stack: linked-list implementation in Java

```
public class LinkedStackOfStrings
{
    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;
    }
}
```

private inner class
(access modifiers for instance
variables don't matter)

Stack: linked-list implementation performance

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with N items uses $\sim 40N$ bytes.

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



Remark. This accounts for the memory for the stack
(but not the memory for strings themselves, which the client owns).

How to implement a fixed-capacity stack with an array?

- A. Can't be done efficiently with an array.

B.

it	was	the	best	of	times	null	null	null	null
0	1	2	3	4	5	6	7	8	9

top of stack



C.

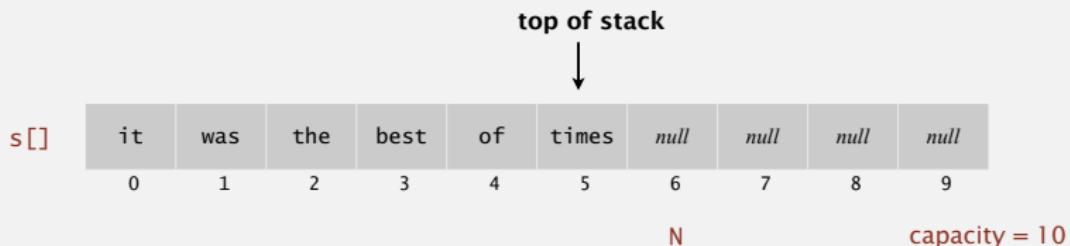
times	of	best	the	was	it	null	null	null	null
0	1	2	3	4	5	6	7	8	9

top of stack



Fixed-capacity stack: array implementation

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



Defect. Stack overflows when N exceeds capacity. [stay tuned]

STACK: ARRAY IMPLEMENTATION

<http://dsvproject.github.io/dsvproject/code/stackArray.html>

Fixed-capacity stack: array implementation

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

    public FixedCapacityStackOfStrings(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]; }
}
```

a cheat
(stay tuned)

use to index into array;
then increment N

decrement N;
then use to index into array

Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

Loitering. Holding a reference to an object when it is no longer needed.

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

loitering

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

this version avoids "loitering":
garbage collector can reclaim memory for
an object only if no outstanding references

A BETTER ARRAY IMPLEMENTATION OF STACKOFSTRINGS



Stack: resizing-array implementation

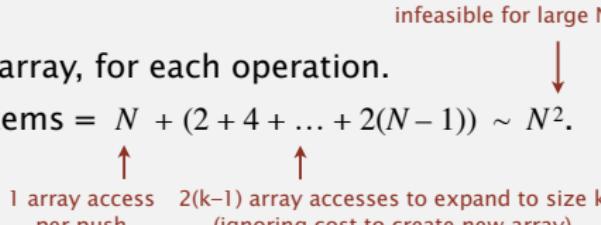
Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

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

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first N items = $N + (2 + 4 + \dots + 2(N-1)) \sim N^2$.


Challenge. Ensure that array resizing happens infrequently.

INCREASE SIZE BY 1

We start with an empty array of size 1

Array accesses (AC):

- 1st push: 1 AC (store new item)
- 2nd push: 1 AC + 2 AC (read-write previous item(s) to new array)
- 3rd push: 1 AC + 4 AC
- 4th push: 1 AC + 6 AC
- ...
- Nth push: $1 \text{ AC} + 2(N - 1) \text{ AC}$

~ N^2 array accesses to insert N items starting from the empty stack

Stack: resizing-array implementation

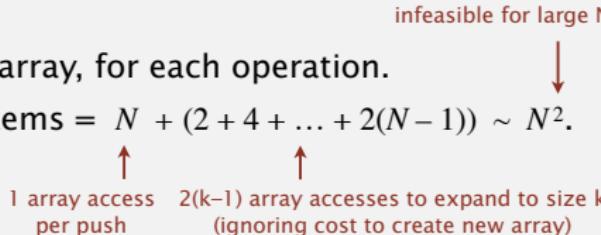
Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

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

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first N items = $N + (2 + 4 + \dots + 2(N-1)) \sim N^2$.


infeasible for large N

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-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 ResizingArrayStackOfStrings()
{   s = new String[1]; }

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

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

Array accesses to insert first $N = 2^i$ items. $N + (2 + 4 + 8 + \dots + N) \sim 3N$.

↑
1 array access
per push

↑
k array accesses to double to size k
(ignoring cost to create new array)

Stack: resizing-array implementation

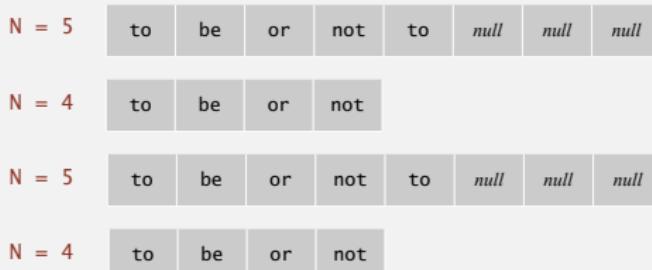
Q. How to shrink array?

First try.

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

Too expensive in worst case.

- Consider push-pop-push-pop... sequence when array is full.
- Each operation takes time proportional to N .



Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

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

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

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

AMORTIZED RUNNING TIME

Assume implementation of ADT with operations A, B, C

How do we calculate the amortized running time of the operations of the ADT?

- Consider the ADT to be **empty**
- Consider **all feasible sequences of N operations**
 - A, A, A, ...
 - A, B, A, C, ...
 - ...
- calculate total running time for each of these sequences take the **largest one** (worst case sequence)
- Amortized running time of ADT operations = worst-case running time of N operations / N

Stack resizing-array implementation: performance

Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.

Proposition. Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M .

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

doubling and
halving operations

order of growth of running time
for resizing stack with N items

RESIZING-ARRAY IMPLEMENTATION OF STACKOFSTRINGS ADT

Operations (push/pop/construct/resize) have
 $O(1)$ amortized running time

Stack resizing-array implementation: memory usage

Proposition. Uses between $\sim 8N$ and $\sim 32N$ bytes to represent a stack with N items.

- $\sim 8N$ when full.
- $\sim 32N$ when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
    private String[] s; ← 8 bytes × array size
    private int N = 0;
    ...
}
```

Remark. This accounts for the memory for the stack
(but not the memory for strings themselves, which the client owns).

Stack implementations: resizing array vs. linked list

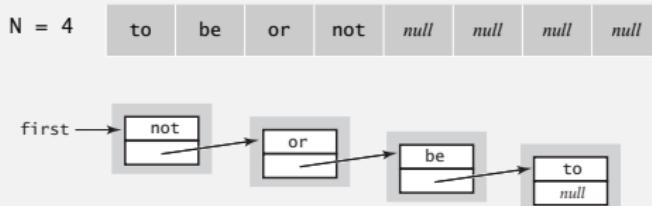
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

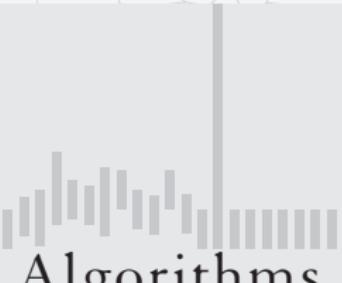
Linked-list implementation.

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

Resizing-array implementation.

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





Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.3 BAGS, QUEUES, AND STACKS

- ▶ *stacks*
- ▶ *resizing arrays*
- ▶ *queues*
- ▶ *generics*
- ▶ *iterators*
- ▶ *applications*

Queue API

```
public class QueueOfStrings
```

```
    QueueOfStrings()
```

create an empty queue

```
    void enqueue(String item)
```

insert a new string onto queue

```
    String dequeue()
```

*remove and return the string
least recently added*

```
    boolean isEmpty()
```

is the queue empty?

```
    int size()
```

number of strings on the queue

enqueue

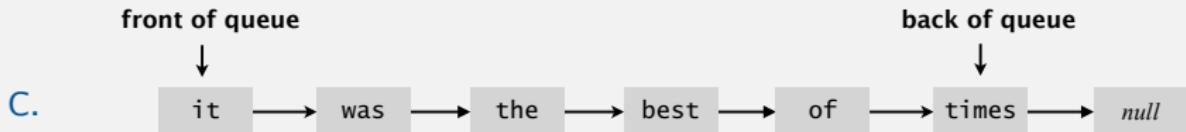


dequeue



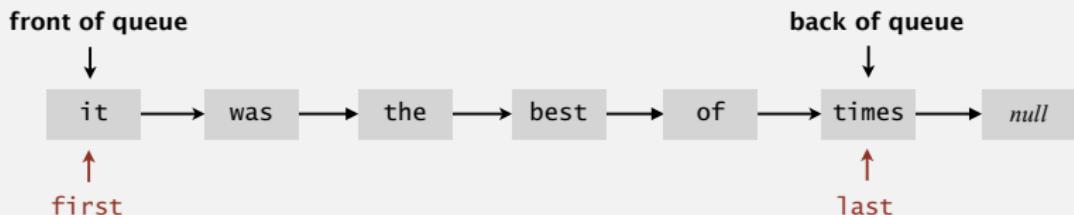
How to implement a queue with a linked list?

- A. Can't be done efficiently with a singly-linked list.



Queue: linked-list implementation

- Maintain one pointer `first` to first node in a singly-linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.



Visualisation:

<http://www.cs.usfca.edu/~galles/visualization/QueueLL.html>

Queue dequeue: linked-list implementation

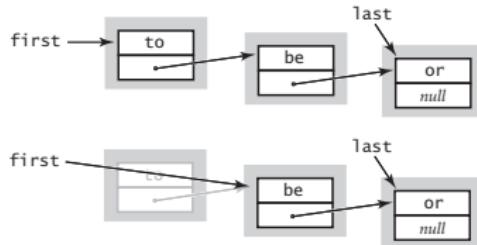
inner class

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

save item to return
`String item = first.item;`

delete first node

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



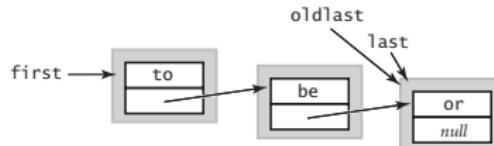
return saved item
`return item;`

Remark. Identical code to linked-list stack pop().

Queue enqueue: linked-list implementation

save a link to the last node

```
Node oldlast = last;
```

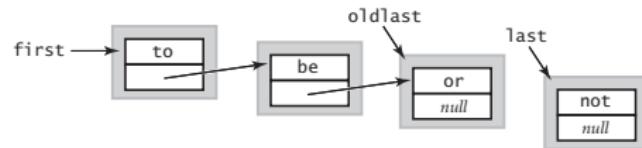


inner class

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

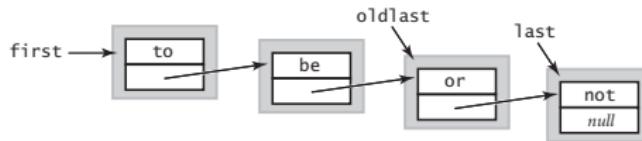
create a new node for the end

```
last = new Node();  
last.item = "not";
```



link the new node to the end of the list

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



Queue: linked-list implementation in Java

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

    private class Node
    { /* same as in LinkedStackOfStrings */ }

    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;
    }
}
```

special cases for
empty queue



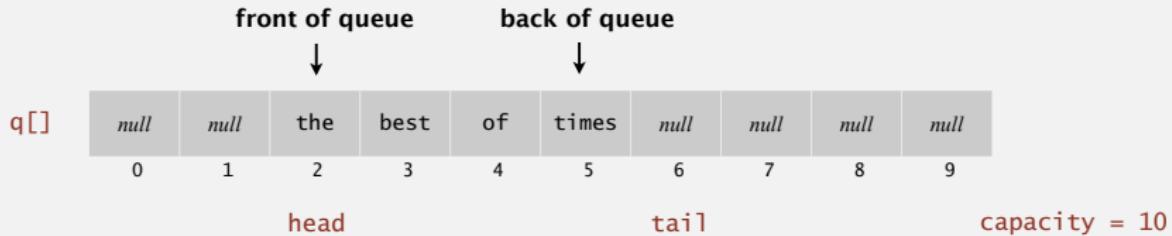
How to implement a fixed-capacity queue with an array?

- A. Can't be done efficiently with an array.



Queue: resizing-array implementation

- Use array $q[]$ to store items in queue.
- $\text{enqueue}()$: add new item at $q[\text{tail}]$.
- $\text{dequeue}()$: remove item from $q[\text{head}]$.
- Update head and tail modulo the capacity.
- Add resizing array.



Q. How to resize?

CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 8: Priority Queues

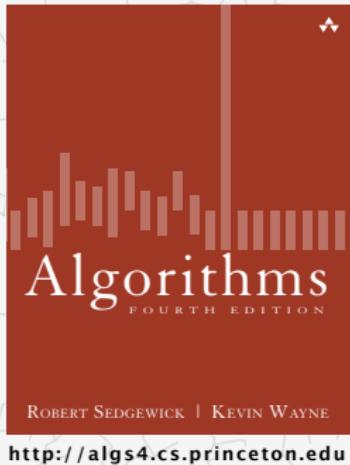
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



2.4 PRIORITY QUEUES

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

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



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

2.4 PRIORITY QUEUES

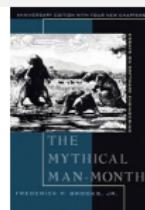
- ▶ *API and elementary implementations*
- ▶ *binary heaps*
- ▶ *heapsort*
- ▶ *event-driven simulation*

Collections

A **collection** is a data types that store groups of items.

data type	key operations	data structure
stack	PUSH, POP	<i>linked list, resizing array</i>
queue	ENQUEUE, DEQUEUE	<i>linked list, resizing array</i>
priority queue	INSERT, DELETE-MAX	<i>binary heap</i>
symbol table	PUT, GET, DELETE	<i>BST, hash table</i>
set	ADD, CONTAINS, DELETE	<i>BST, hash table</i>

“Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.” — Fred Brooks



Priority queue

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

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

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

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

Priority queue API

Requirement. Generic items are Comparable.

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

Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [online median in data stream]
- Operating systems. [load balancing, interrupt handling]
- Computer networks. [web cache]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.

Priority queue client example

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

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

N huge, M large



Constraint. Not enough memory to store N items.

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

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



sort key

Priority queue client example

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

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

N huge, M large

Constraint. Not enough memory to store N items.

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

use a min-oriented pq

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

pq contains largest M items

Priority queue client example

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

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

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

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 P

A sequence of operations on a priority queue

Priority queue: unordered array implementation

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

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

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

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

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

no generic array creation

less() and exch()
similar to sorting methods
(but don't pass pq[])

should null out entry
to prevent loitering

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 of running time for priority queue with N items



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

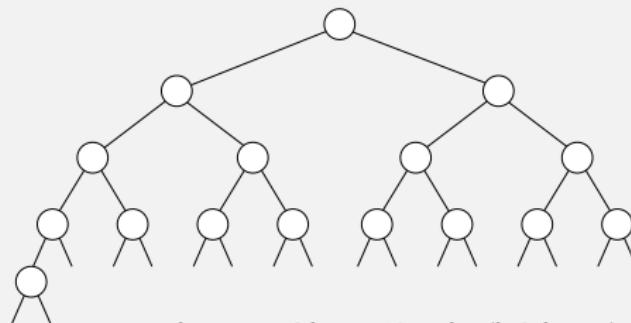
2.4 PRIORITY QUEUES

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

Complete binary tree

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

Complete tree. Perfectly balanced, except for bottom level.



Property. Height of complete tree with N nodes is $\lceil \lg N \rceil$.

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

A complete binary tree in nature



Hyphaene Compressa - Doum Palm

© Shlomit Pinter

Binary heap representations

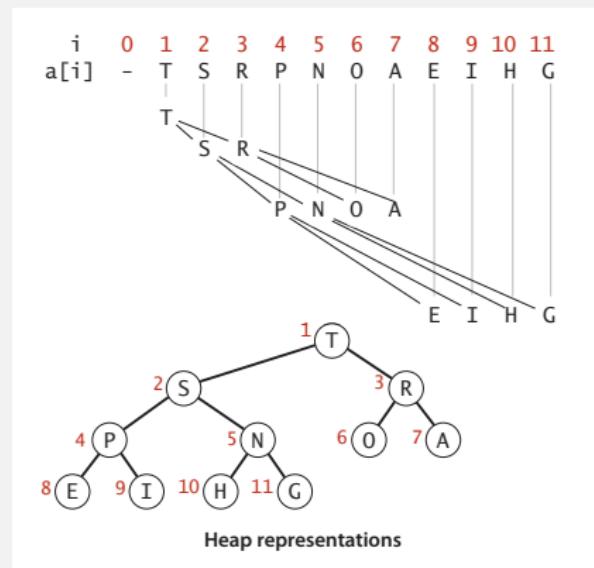
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

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

Array representation.

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

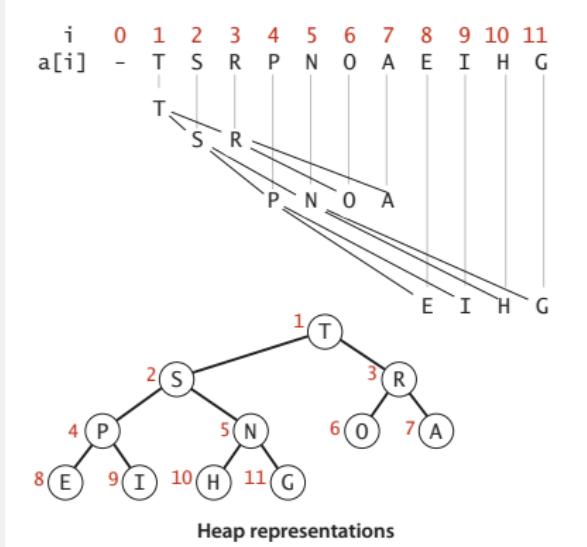


Binary heap properties

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

Proposition. Can use array indices to move through tree.

- Parent of node at k is at $k/2$.
- Children of node at k are at $2k$ and $2k+1$.
- left subtree of k is empty if $2k>N$.
- right subtree of k is empty if $(2k+1)>N$.
- k is a leaf node if $2k>N$.



CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 9: Binary Heap – Heapsort

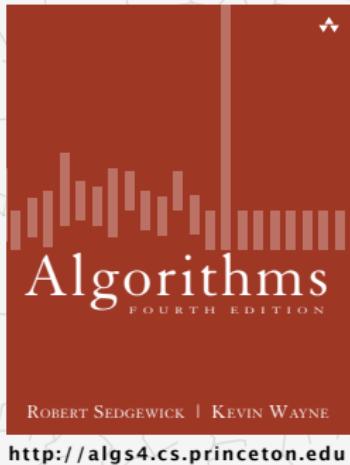
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



2.4 PRIORITY QUEUES

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

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



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

2.4 PRIORITY QUEUES

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

LAST LECTURE

Priority queue

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

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Randomized queue. Remove a random item.

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

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

Priority queue API

Requirement. Generic items are Comparable.

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

Priority queue client example

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

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

N huge, M large



Constraint. Not enough memory to store N items.

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

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



Priority queue client example

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

- Fraud detection: isolate \$\$ transactions.
- NSA monitoring: flag most suspicious documents.

N huge, M large

Constraint. Not enough memory to store N items.

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

use a min-oriented pq

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

pq contains largest M items

Priority queue client example

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

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

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

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 P

A sequence of operations on a priority queue

Priority queue: unordered array implementation

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

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

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

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

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

no generic array creation

less() and exch()
similar to sorting methods
(but don't pass pq[])

should null out entry
to prevent loitering

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 of running time for priority queue with N items

TODAY

TODAY

- Implementation of binary heaps
- Practical improvements of binary heaps
- Heapsort



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

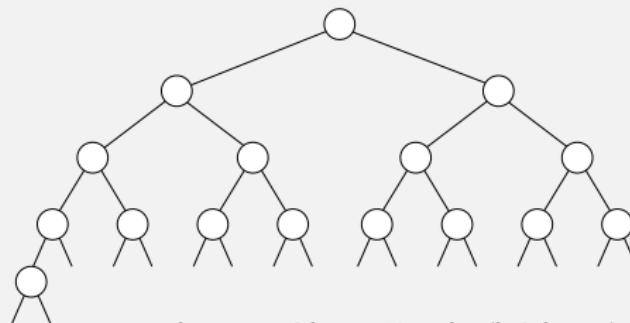
2.4 PRIORITY QUEUES

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

Complete binary tree

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

Complete tree. Perfectly balanced, except for bottom level.



Property. Height of complete tree with N nodes is $\lceil \lg N \rceil$.

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

A complete binary tree in nature



Hyphaene Compressa - Doum Palm

© Shlomit Pinter

Binary heap representations

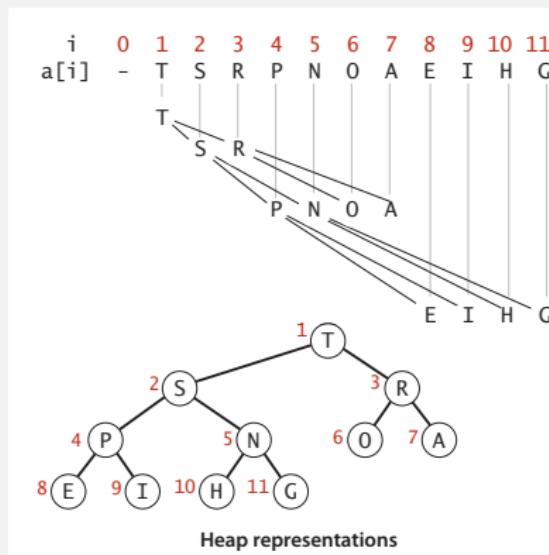
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

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

Array representation.

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

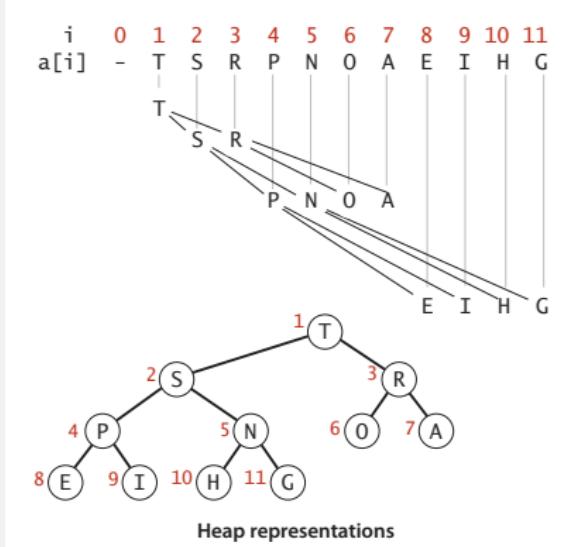


Binary heap properties

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

Proposition. Can use array indices to move through tree.

- Parent of node at k is at $k/2$.
- Children of node at k are at $2k$ and $2k+1$.
- left subtree of k is empty if $2k>N$.
- right subtree of k is empty if $(2k+1)>N$.
- k is a leaf node if $2k>N$.

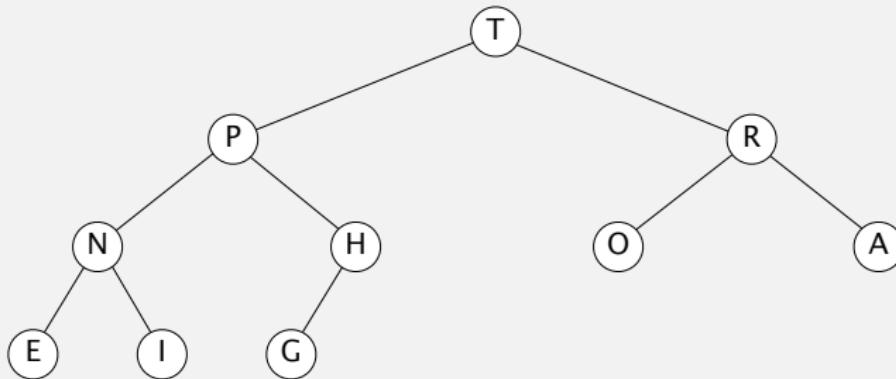


Binary heap demo

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

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

heap ordered



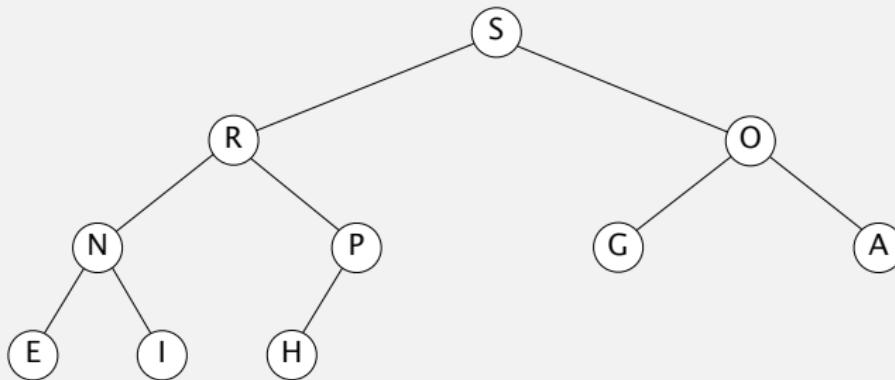
T	P	R	N	H	O	A	E	I	G	
---	---	---	---	---	---	---	---	---	---	--

Binary heap demo

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

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

heap ordered



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

Promotion in a heap

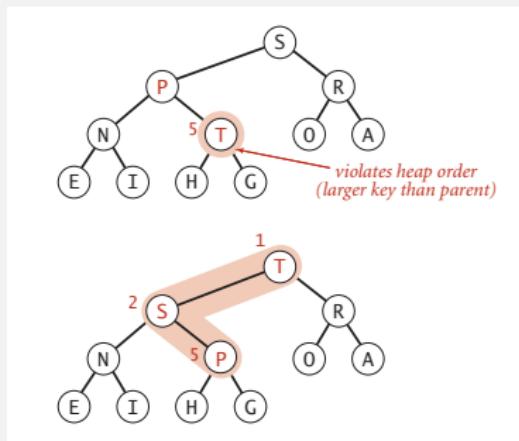
Scenario. Child's key becomes **larger** key than its parent's key.

To eliminate the violation:

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

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

parent of node at k is at k/2



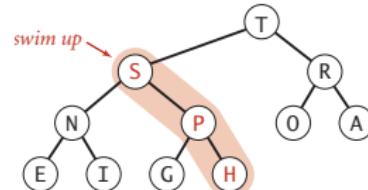
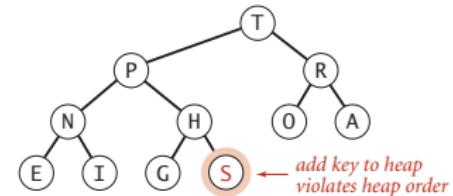
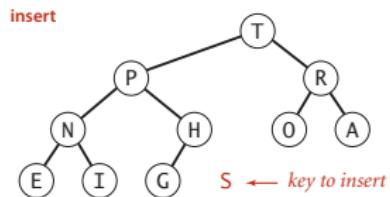
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

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

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

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



Demotion in a heap

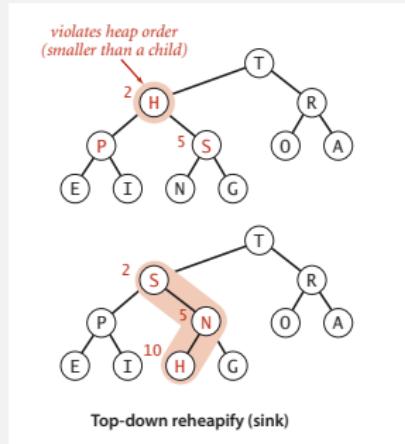
Scenario. Parent's key becomes **smaller** than one (or both) of its children's.

To eliminate the violation:

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

why not smaller child?
↓

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



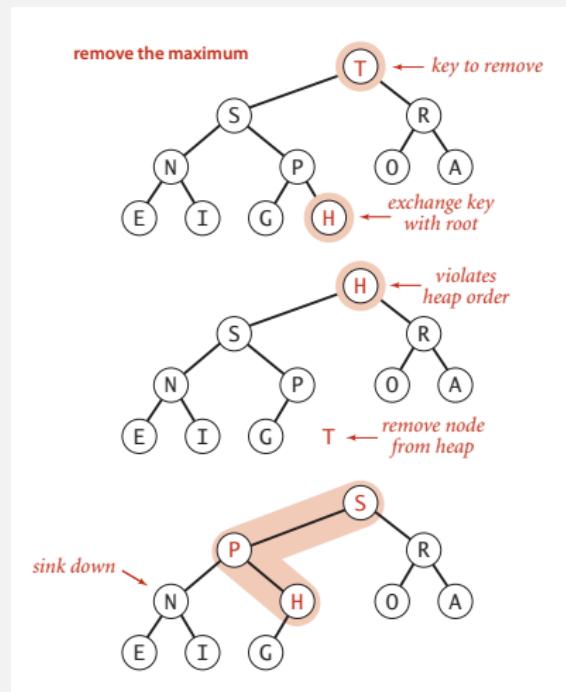
Power struggle. Better subordinate promoted.

Delete the maximum in a heap

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

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

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



Binary heap: 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]; } ← fixed capacity  
          (for simplicity)

    public boolean isEmpty()
    {   return N == 0; }
    public void insert(Key key)
    public Key delMax()
    {   /* see previous code */ }

    private void swim(int k)
    private void sink(int k) ← PQ ops
    {   /* 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; } ← heap helper functions

}
```

array helper functions

Priority queues implementation cost summary

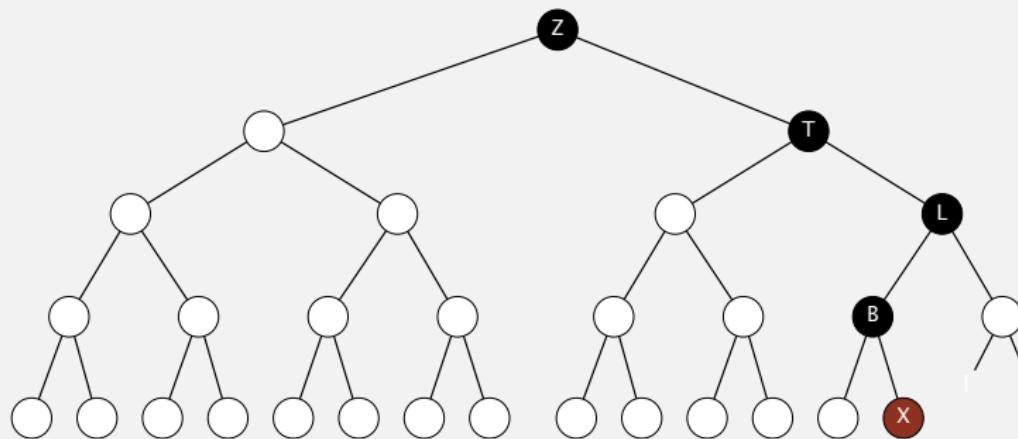
implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	$\log N$	$\log N$	1

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

Binary heap: practical improvements

Half-exchanges in sink and swim.

- Reduces number of array accesses.
- Worth doing.



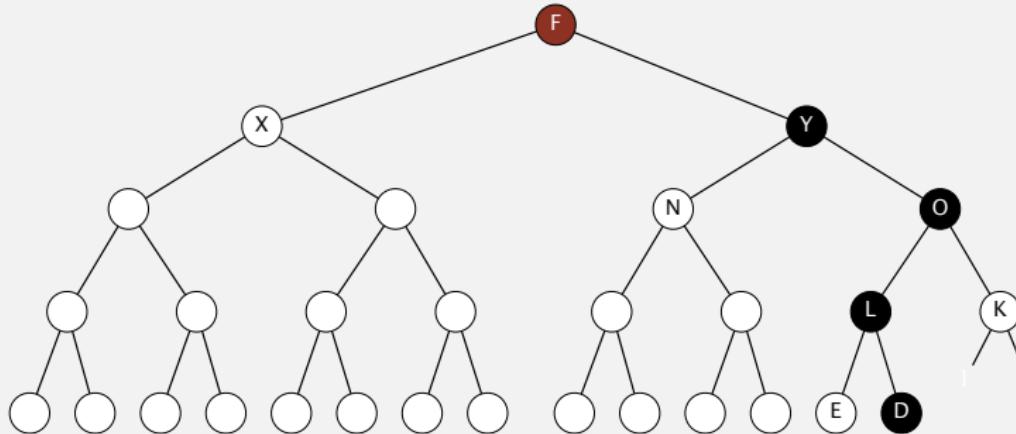
Binary heap: practical improvements

Floyd's sink-to-bottom trick.

- Sink key at root all the way to bottom. ← 1 compare per node
- Swim key back up. ← some extra compares and exchanges
- Fewer compares; more exchanges.
- Worthwhile depending on cost of compare and exchange.



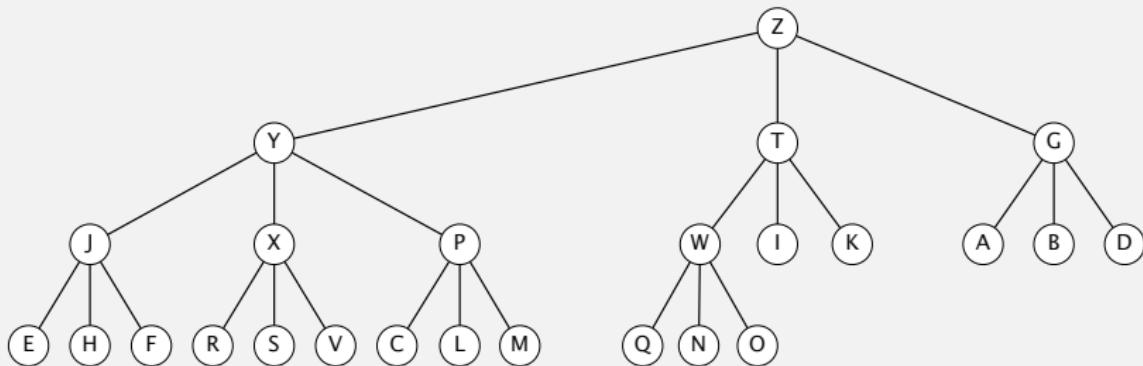
R. W. Floyd
1978 Turing award



Binary heap: practical improvements

Multiway heaps.

- Complete d -way tree.
- Parent's key no smaller than its children's keys.
- Swim takes $\log_d N$ compares; sink takes $d \log_d N$ compares.
- Sweet spot: $d = 4$.

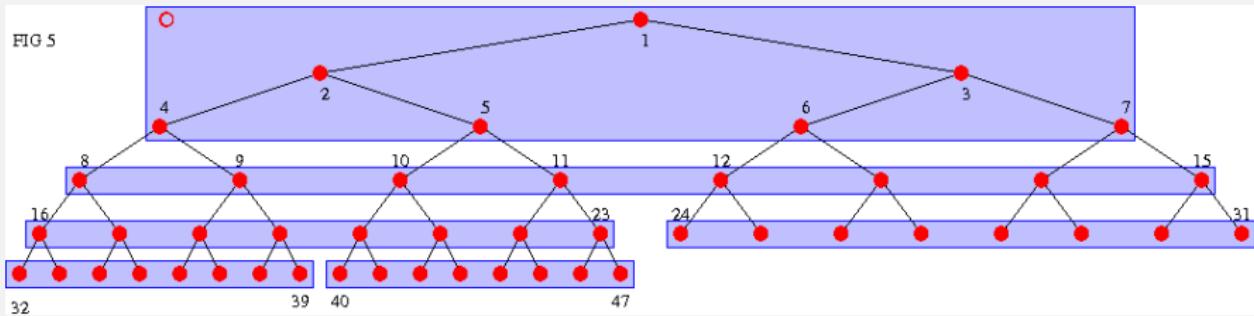


3-way heap

Binary heap: practical improvements

Caching. Binary heap is not cache friendly.

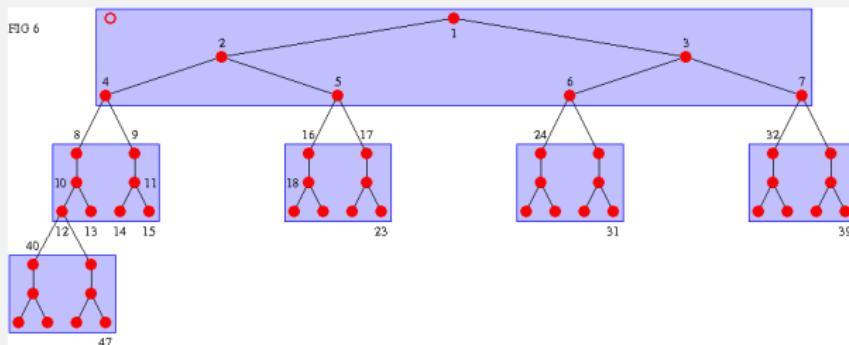
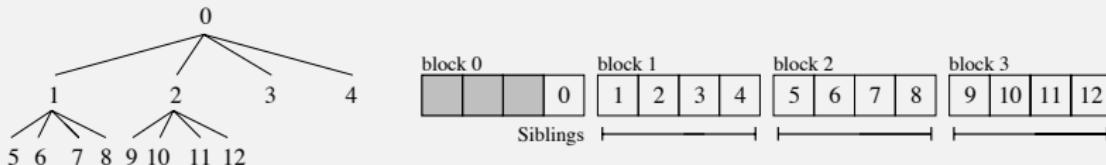
FIG 5



Binary heap: practical improvements

Caching. Binary heap is not cache friendly.

- Cache-aligned d -heap.
- Funnel heap.
- B-heap.
- ...



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
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	$\log N^\dagger$	1
Brodal queue	1	$\log N$	1
impossible	1	1	1

← why impossible?

† amortized

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

Binary heap considerations

Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to log N
amortized time per op
(how to make worst case?)

Minimum-oriented priority queue.

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

Other operations.

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

can implement efficiently with `sink()` and `swim()`
[stay tuned for Prim/Dijkstra]

Immutability of keys.

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

Immutability: implementing in Java

Data type. Set of values and operations on those values.

Immutable data type. Can't change the data type value once created.

```
public final class Vector {  
    private final int N;  
    private final double[] data;  
  
    public Vector(double[] data) {  
        this.N = data.length;  
        this.data = new double[N];  
        for (int i = 0; i < N; i++)  
            this.data[i] = data[i];  
    }  
    ...  
}
```

← can't override instance methods

← instance variables private and final

← defensive copy of mutable
instance variables

← instance methods don't change
instance variables

Immutable. String, Integer, Double, Color, Vector, Transaction, Point2D.

Mutable. StringBuilder, Stack, Counter, Java array.

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can't change the data type value once created.

Advantages.

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



Disadvantage. Must create new object for each data type value.

“Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible.”

— Joshua Bloch (Java architect)





Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

2.4 PRIORITY QUEUES

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

Sorting with a binary heap

Q. What is this sorting algorithm?

```
public void sort(String[] a)
{
    int N = a.length;
    MaxPQ<String> pq = new MaxPQ<String>();
    for (int i = 0; i < N; i++)
        pq.insert(a[i]);
    for (int i = N-1; i >= 0; i--)
        a[i] = pq.delMax();
}
```

Q. What are its properties?

A. $N \log N$, extra array of length N , not stable.

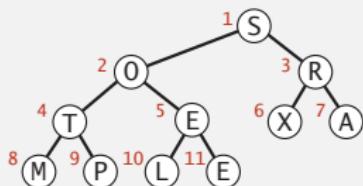
Heapsort intuition. A heap is an array; do sort in place.

Heapsort

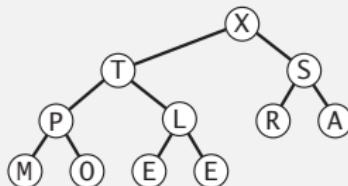
Basic plan for in-place sort.

- View input array as a complete binary tree.
- Heap construction: build a max-heap with all N keys.
- Sortdown: repeatedly remove the maximum key.

keys in arbitrary order



build max heap
(in place)



sorted result
(in place)



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

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

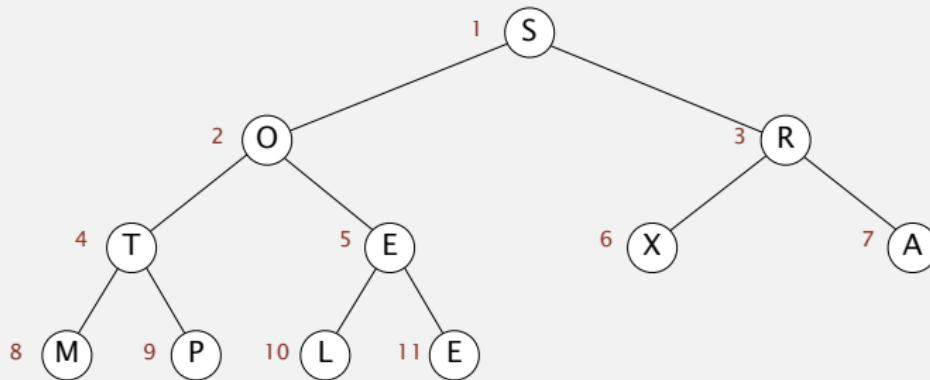
1 2 3 4 5 6 7 8 9 10 11
A E E L M O P R S T X

Heapsort demo

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

we assume array entries are indexed 1 to N

array in arbitrary order



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

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order

A

E

E

L

M

O

P

R

S

T

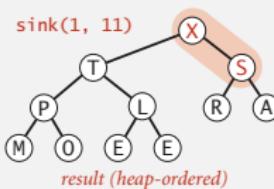
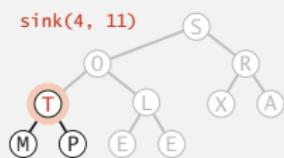
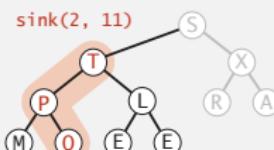
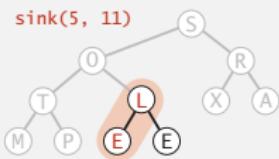
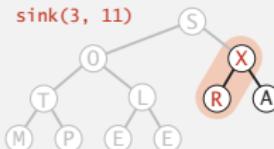
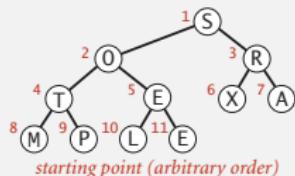
X

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

Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

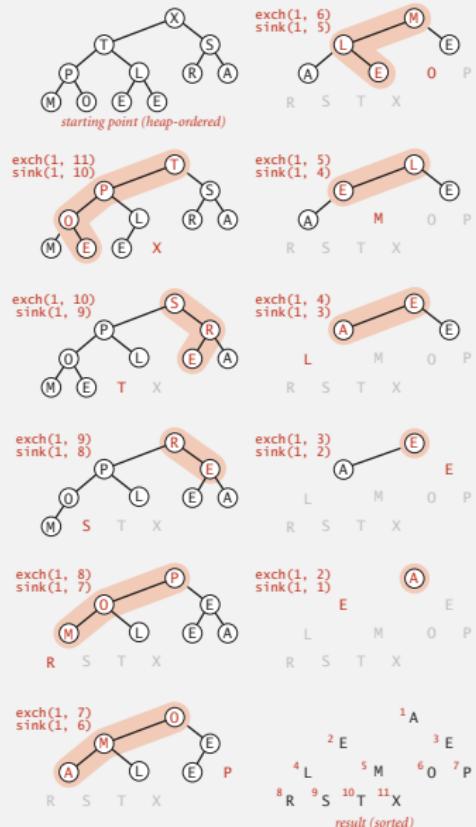


Heapsort: sortdown

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```



Heapsort: Java implementation

```
public class Heap
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int k = N/2; k >= 1; k--)
            sink(a, k, N);
        while (N > 1)
        {
            exch(a, 1, N);
            sink(a, 1, --N);
        }
    }  
        but make static (and pass arguments)  
private static void sink(Comparable[] a, int k, int N)
{ /* as before */ }  
  
private static boolean less(Comparable[] a, int i, int j)
{ /* as before */ }  
  
private static void exch(Object[] a, int i, int j)
{ /* as before */ }  
}  
        but convert from 1-based  
        indexing to 0-base indexing
```

Heapsort: trace

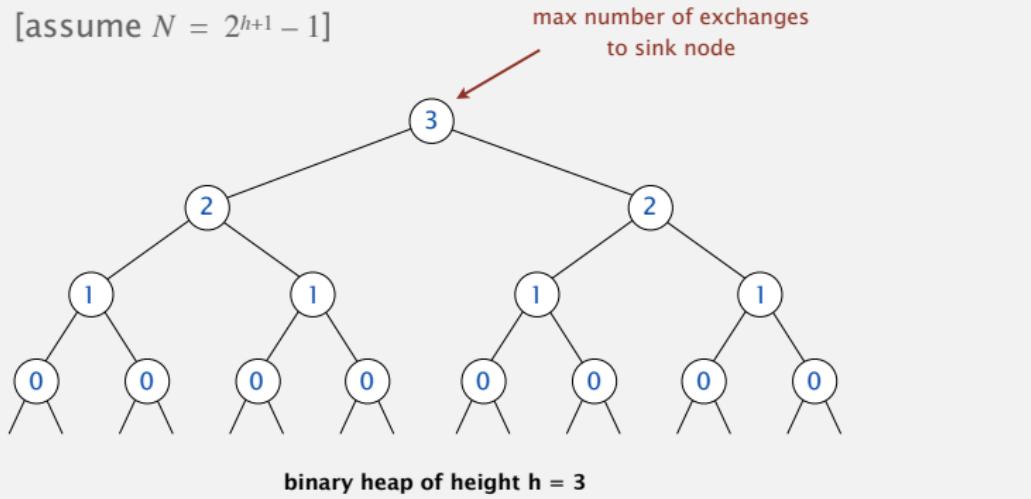
N	k	0	1	2	3	4	5	6	7	8	9	10	11
		S	O	R	T	E	X	A	M	P	L	E	
<i>initial values</i>		S	O	R	T	L	X	A	M	P	E	E	
11	5	S	O	R	T	L	X	A	M	P	E	E	
11	4	S	O	R	T	L	X	A	M	P	E	E	
11	3	S	O	X	T	L	R	A	M	P	E	E	
11	2	S	T	X	P	L	R	A	M	O	E	E	
11	1	X	T	S	P	L	R	A	M	O	E	E	
<i>heap-ordered</i>		X	T	S	P	L	R	A	M	O	E	E	
10	1	T	P	S	O	L	R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

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Pf sketch. [assume $N = 2^{h+1} - 1$]



$$h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \dots + 2^h(0) \leq 2^{h+1} \\ = N$$

a tricky sum
(see COS 340)

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and $\leq N$ exchanges.

Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.

algorithm can be improved to $\sim 1N \lg N$

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← $N \log N$ worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, but:

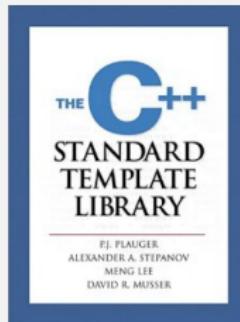
- Inner loop longer than quicksort's.
- Makes poor use of cache.
- Not stable.

advanced tricks for improving

Goal. As fast as quicksort in practice; $N \log N$ worst case, in place.

Introsort.

- Run quicksort.
- Cutoff to heapsort if stack depth exceeds $2 \lg N$.
- Cutoff to insertion sort for $N = 16$.



Introspective Sorting and Selection Algorithms

David R. Musser^a
Computer Science Department
Rensselaer Polytechnic Institute, Troy, NY 12180
musser@cs.rpi.edu

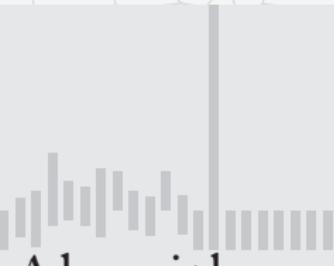
Abstract

Quicksort is the preferred large-scale sorting algorithm in many contexts, due to its average complexity (for randomly distributed inputs) is $O(N \log N)$, and it is in fact faster than most other sorting algorithms on most inputs. Its downside is that its worst-case time bound is $O(N^2)$. Previous attempts to protect against the worst case by imposing the way quicksort handles ties have been unsatisfactory. This paper proposes a new approach that is much more robust—so robust that it might as well be heapsort, which has an $O(N \log N)$ worst-case time bound but less the average 2-to-5 slowdown from quicksort. A similar algorithm with efficient algorithms for finding the k th largest element, insertion partitioning, and for selecting a fan sorted in almost linear time, is another algorithm with interesting properties. Using template as the “glue,” yields a sorting algorithm that is just as fast as quicksort in the average case but also has an $O(N \log N)$ worst case time bound. For selection, a hybrid of Horne’s two algorithms, which is linear on average but quadratic in the worst case, is used. The paper also discusses the reasons why this new algorithm in practice yet has a linear worst-case time bound. Also discussed are issues of implementing the new algorithms as generic algorithms and accurately measuring their performance in the framework of the C++ Standard Template Library.

In the wild. C++ STL, Microsoft .NET Framework.

Sorting algorithms: summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✓	✓	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✓		$N \log_3 N$?	$c N^{3/2}$	tight code; subquadratic
→ merge		✓	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✓	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
quick	✓		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✓		N	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
→ heap	✓		N	$2 N \lg N$	$2 N \lg N$	$N \log N$ guarantee; in-place
?	✓	✓	N	$N \lg N$	$N \lg N$	holy sorting grail



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

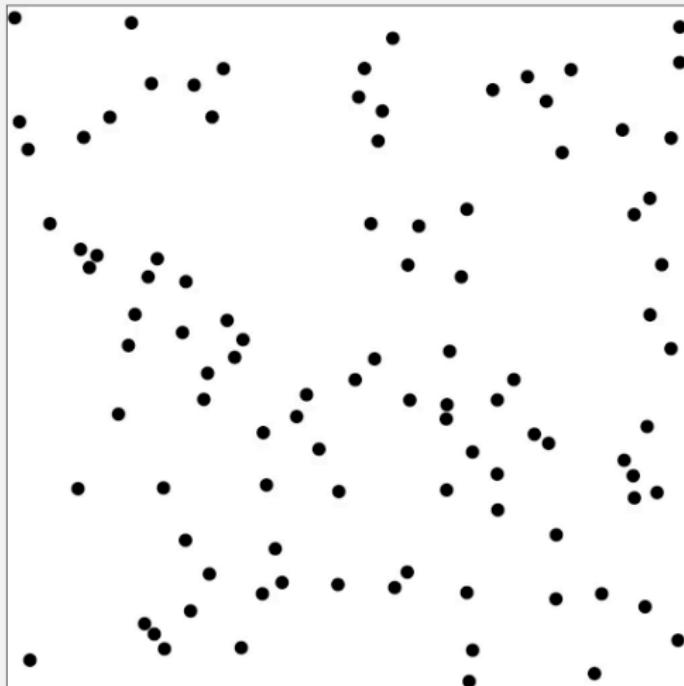
<http://algs4.cs.princeton.edu>

2.4 PRIORITY QUEUES

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

Molecular dynamics simulation of hard discs

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

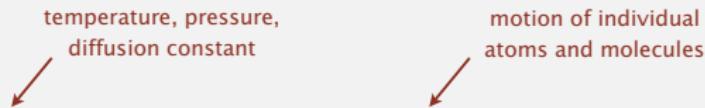


Molecular dynamics simulation of hard discs

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

Hard disc model.

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



Significance. Relates macroscopic observables to microscopic dynamics.

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

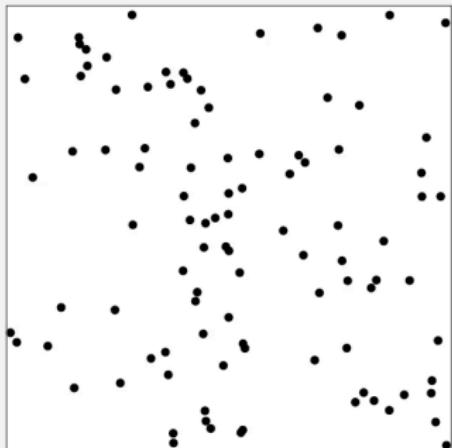
Warmup: bouncing balls

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

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

main simulation loop

```
% java BouncingBalls 100
```



Warmup: bouncing balls

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

    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); }
}
```

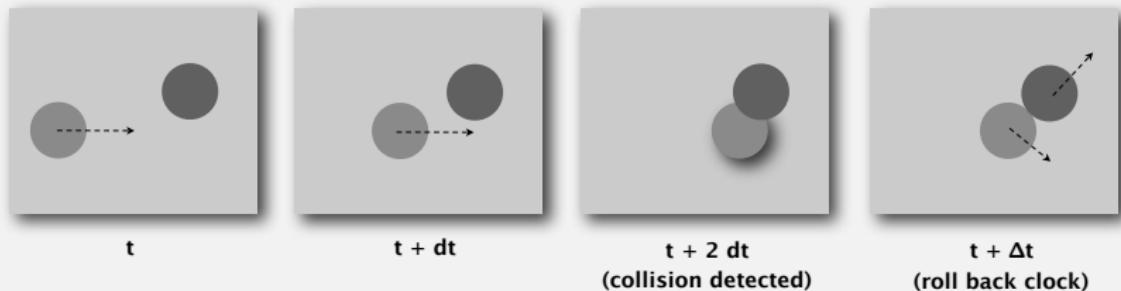


Missing. Check for balls colliding with each other.

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

Time-driven simulation

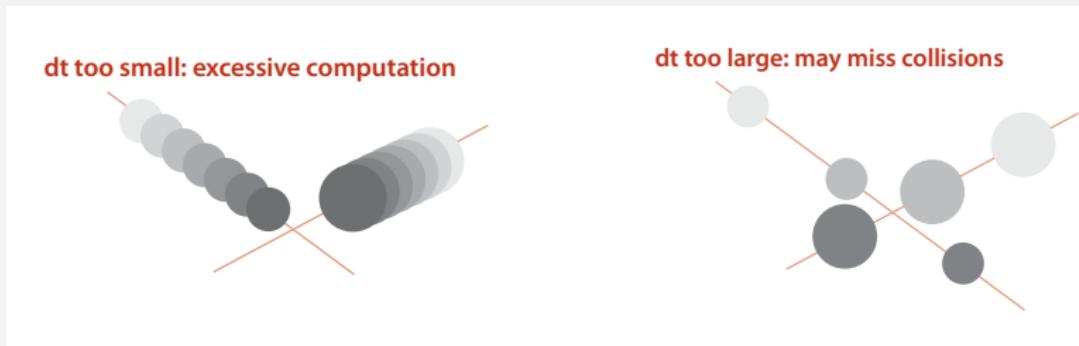
- Discretize time in quanta of size dt .
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.



Time-driven simulation

Main drawbacks.

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



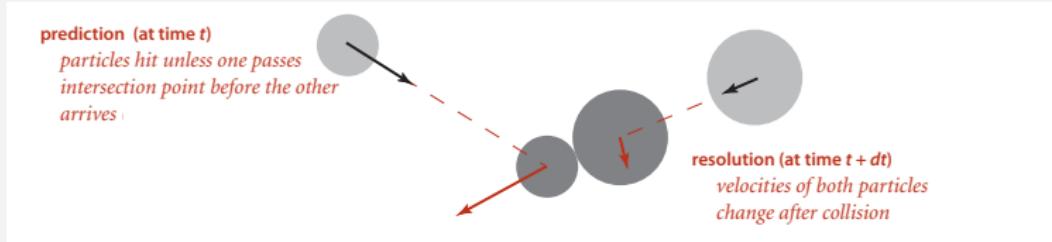
Event-driven simulation

Change state only when something happens.

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

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

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



Particle-wall collision

Collision prediction and resolution.

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

prediction (at time t)

$$dt \equiv \text{time to hit wall}$$
$$= \text{distance}/\text{velocity}$$
$$= (1 - s - r_x)/v_x$$

resolution (at time $t + dt$)

$$\text{velocity after collision} = (-v_x, v_y)$$
$$\text{position after collision} = (1 - s, r_y + v_y dt)$$

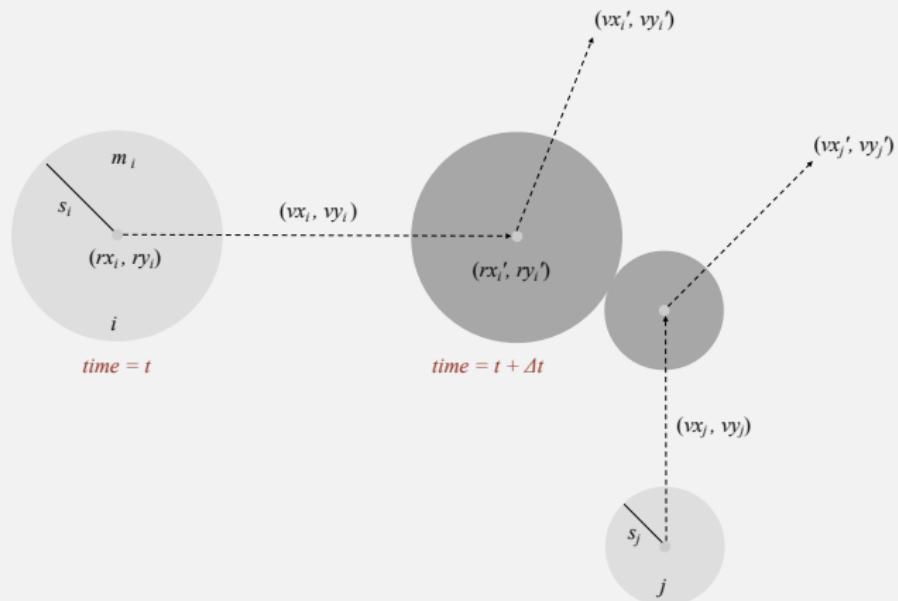
wall at $x = 1$

Predicting and resolving a particle-wall collision

Particle-particle collision prediction

Collision prediction.

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



Particle-particle collision prediction

Collision prediction.

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

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

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

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

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

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

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

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

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

Particle-particle collision resolution

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

$$vx_i' = vx_i + Jx / m_i$$

$$vy_i' = vy_i + Jy / m_i$$

$$vx_j' = vx_j - Jx / m_j$$

$$vy_j' = vy_j - Jy / m_j$$

Newton's second law
(momentum form)

$$J_x = \frac{J \Delta r_x}{\sigma}, \quad J_y = \frac{J \Delta r_y}{\sigma}, \quad J = \frac{2m_i m_j (\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

impulse due to normal force

(conservation of energy, conservation of momentum)

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

Particle data type skeleton

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

    public Particle(...) { }

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

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

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

}
```

predict collision
with particle or wall

resolve collision
with particle or wall

Particle-particle collision and resolution implementation

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

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

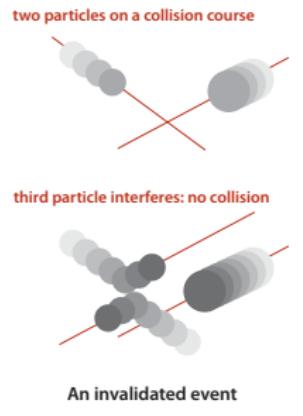
Collision system: event-driven simulation main loop

Initialization.

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



"potential" since collision may not happen if some other collision intervenes



Main loop.

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

Event data type

Conventions.

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

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

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

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

    public boolean isValid()
    {   }
}
```

invalid if
intervening
collision

Collision system implementation: skeleton

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

    public CollisionSystem(Particle[] particles) { }

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

    private void redraw() { }

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

Collision system implementation: main event-driven simulation loop

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

← initialize PQ with collision events and redraw event

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

← get next event

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

← update positions and time

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

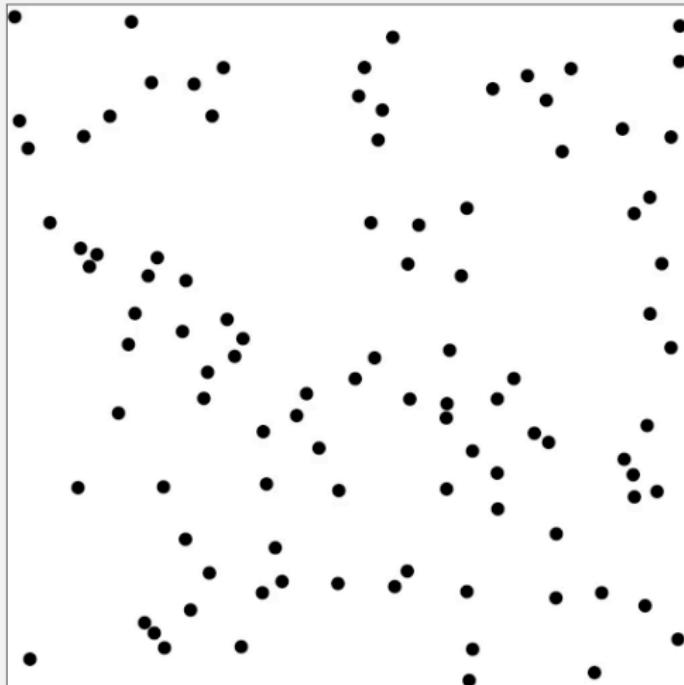
← process event

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

← predict new events based on changes

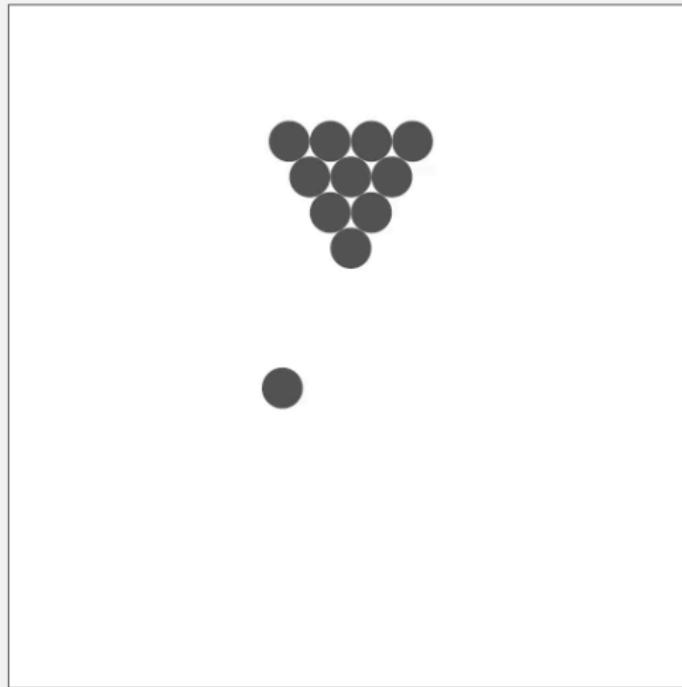
Particle collision simulation example 1

```
% java CollisionSystem 100
```



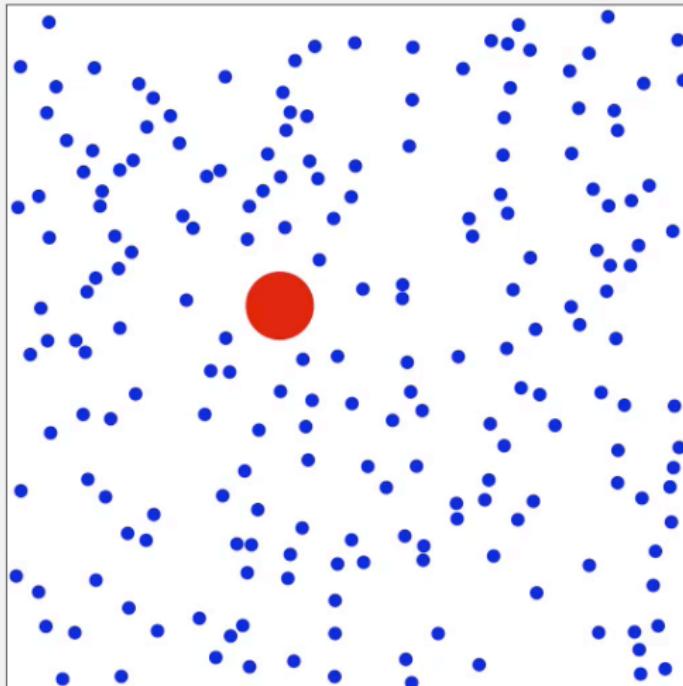
Particle collision simulation example 2

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



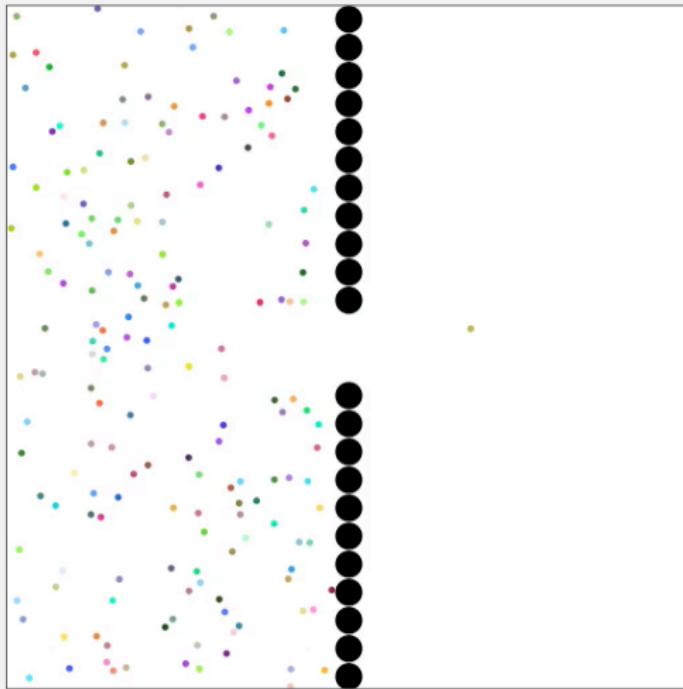
Particle collision simulation example 3

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



Particle collision simulation example 4

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



CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 10: Recursion vs Iteration

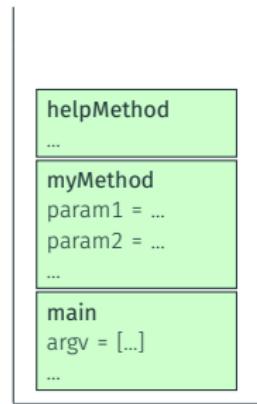
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

HOW METHODS EXECUTE

- **Call stack**: is a stack maintained by the Java runtime system
- One **call stack frame** (aka activation record) for each **running instance** of a method: contains all information necessary to execute the method
 - **references** to parameter values and local objects, return address etc.
- Objects themselves are stored in another part of memory: the **heap**
- every time a method is called, a new stack frame is pushed on the call stack.
- every time a method returns, the top-most stack frame is popped.



RECURSION

Recursion: when something is defined in terms of itself.

Infinite Recusion



Well-founded Recursion



RECURSION

Principle: A method is **recursive** when its definition **calls the method itself**.

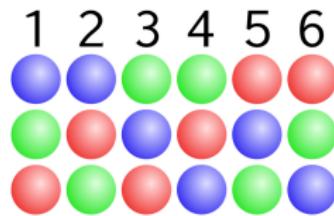
A correct recursive method should be **well-founded**: it should terminate/must end up at a **base case**.

Classic example: factorial – in math written as: $n!$

Math definition:

$$0! = 1$$

$$n! = n \cdot (n - 1)! \quad \text{when } n > 0$$



RECURSION

Principle: A method is **recursive** when its definition **calls the method itself**.

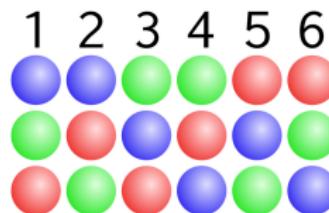
A correct recursive method should be **well-founded**: it should terminate/must end up at a **base case**.

Classic example: factorial – in math written as: $n!$

Math definition:

$$0! = 1$$

$$n! = n \cdot (n - 1)! \quad \text{when } n > 0$$



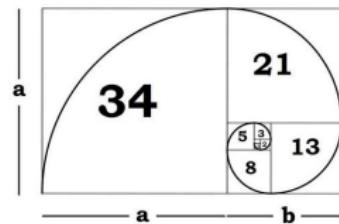
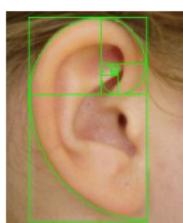
Classic example: Fibonacci numbers

Math definition:

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2) \quad \text{when } n > 1$$



RECURSION

Principle: A method is **recursive** when its definition **calls the method itself**.

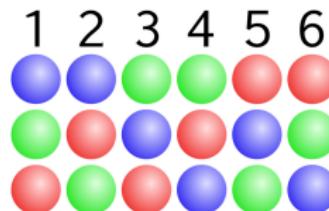
A correct recursive method should be **well-founded**: it should terminate/must end up at a **base case**.

Classic example: factorial – in math written as: $n!$

Math definition:

$$0! = 1$$

$$n! = n \cdot (n - 1)! \quad \text{when } n > 0$$



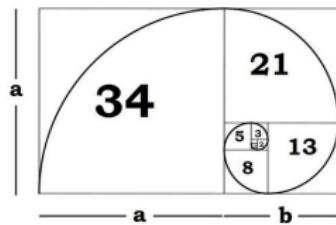
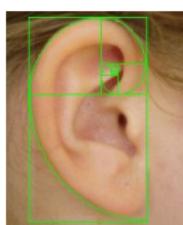
Classic example: Fibonacci numbers

Math definition:

$$\text{fib}(0) = 1$$

$$\text{fib}(1) = 1$$

$$\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2) \quad \text{when } n > 1$$



It is convenient to implement recursive math definitions using recursive methods.

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {
```

```
}
```

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.
- follows the **divide and conquer** approach:

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.
- follows the **divide and conquer** approach:
 - break the implementation of **fac(n)** into smaller and smaller parts: **fac(n-1)**

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.
- follows the **divide and conquer** approach:
 - break the implementation of **fac(n)** into smaller and smaller parts: **fac(n-1)**
 - only deal with the **smallest possible cases** (here when $n = 0$): the **base cases**

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.
- follows the **divide and conquer** approach:
 - break the implementation of **fac(n)** into smaller and smaller parts: **fac(n-1)**
 - only deal with the **smallest possible cases** (here when $n = 0$): the **base cases**
 - the rest of the cases are the **recursive cases** (here when $n > 0$)

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.
- follows the **divide and conquer** approach:
 - break the implementation of **fac(n)** into smaller and smaller parts: **fac(n-1)**
 - only deal with the **smallest possible cases** (here when $n = 0$): the **base cases**
 - the rest of the cases are the **recursive cases** (here when $n > 0$)
 - specify how smaller solutions **compose** into the solutions of recursive cases

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Closely matches the mathematical definition.
- follows the **divide and conquer** approach:
 - break the implementation of **fac(n)** into smaller and smaller parts: **fac(n-1)**
 - only deal with the **smallest possible cases** (here when $n = 0$): the **base cases**
 - the rest of the cases are the **recursive cases** (here when $n > 0$)
 - specify how smaller solutions **compose** into the solutions of recursive cases
 - it is usually a **top-down calculation**

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

→ Q: asymptotic worst-case running time? (# recursive calls * cost of each call)

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

→ Q: asymptotic worst-case running time? (# recursive calls * cost of each call)
A: $\Theta(n)$ time

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

- Q: asymptotic worst-case running time? (# recursive calls * cost of each call)
A: $\Theta(n)$ time
- Q: memory space for call stack frames? (max # frames on call stack)

IMPLEMENTATION OF FACTORIAL

Recursive implementation:

```
int fac(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

→ Q: asymptotic worst-case running time?

(# recursive calls * cost of each call)

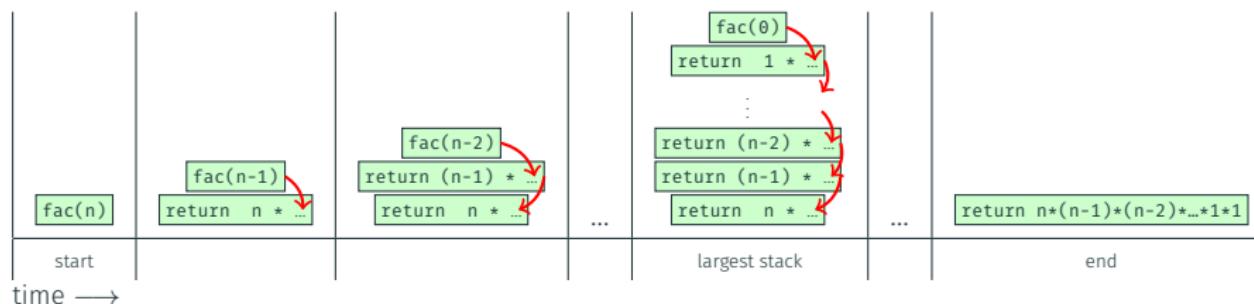
A: $\Theta(n)$ time

→ Q: memory space for call stack frames?

(max # frames on call stack)

A: $\Theta(n)$ space

All this call stack space is needed because of the `return n * ...`



IMPLEMENTATION OF FACTORIAL

Recursive implementation using **accumulator**: (H/W: can you implement the accumulator version **bottom-up**?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return facAcc(n-1, acc * n);
}
```

IMPLEMENTATION OF FACTORIAL

Recursive implementation using **accumulator**: (H/W: can you implement the accumulator version **bottom-up**?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return facAcc(n-1, acc * n);
}
```

→ Q: asymptotic worst-case running time?

IMPLEMENTATION OF FACTORIAL

Recursive implementation using **accumulator**: (H/W: can you implement the accumulator version **bottom-up**?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return facAcc(n-1, acc * n);
}
```

→ Q: asymptotic worst-case running time?

$\Theta(n)$

IMPLEMENTATION OF FACTORIAL

Recursive implementation using **accumulator**: (H/W: can you implement the accumulator version **bottom-up**?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return facAcc(n-1, acc * n);
}
```

- Q: asymptotic worst-case running time?
- Q: memory space for call stack frames?

$\Theta(n)$

IMPLEMENTATION OF FACTORIAL

Recursive implementation using **accumulator**: (H/W: can you implement the accumulator version **bottom-up**?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return facAcc(n-1, acc * n);
}
```

- Q: asymptotic worst-case running time? $\Theta(n)$
- Q: memory space for call stack frames?
 - In Java $\Theta(n)$ for stack space
 - In other, mainly functional, languages (ML, Lisp, Haskell, ...) the compiler runs this using $\Theta(1)$ stack space.
- Only the top-most stack frame is necessary because every function call simply returns the inner result: `return facAcc(n-1, acc * n)`
- This is called **tail recursion**

IMPLEMENTATION OF FACTORIAL

Recursive implementation using **accumulator**: (H/W: can you implement the accumulator version **bottom-up**?)

```
int fac(int n) { return facAcc(n, 1); }

int facAcc(int n, int acc) {
    if (n == 0)
        return acc;
    else
        return facAcc(n-1, acc * n);
}
```

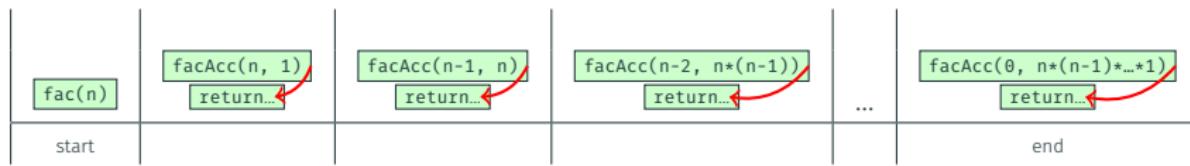
- Q: asymptotic worst-case running time? $\Theta(n)$
- Q: memory space for call stack frames?

→ In Java $\Theta(n)$ for stack space

→ In other, mainly functional, languages (ML, Lisp, Haskell, ...) the compiler runs this using $\Theta(1)$ stack space.

Only the top-most stack frame is necessary because every function call simply returns the inner result: `return facAcc(n-1, acc * n)`

This is called **tail recursion**



IMPLEMENTATION OF FACTORIAL

From tail recursive implementation → iterative implementation:

```
int fac(int n) { return facAcc(n, 1); }           int fac(int n) {  
int facAcc(int n, int acc) {                      int acc = 1;  
    if (n == 0)                                    for ( ; !(n == 0); n--) {  
        return acc;                                acc = acc * n;  
    else  
        return facAcc(n-1, acc * n);  
}                                                 }  
}                                                 return acc;  
}
```

IMPLEMENTATION OF FACTORIAL

From tail recursive implementation → iterative implementation:

```
int fac(int n) { return facAcc(n, 1); }           int fac(int n) {  
int facAcc(int n, int acc) {                      int acc = 1;  
    if (n == 0)                                    for ( ; !(n == 0); n--) {  
        return acc;                                acc = acc * n;  
    else  
        return facAcc(n-1, acc * n);  
}                                                 }  
}                                                 return acc;  
}
```

→ Running time of iterative implementation: $\Theta(n)$

IMPLEMENTATION OF FACTORIAL

From tail recursive implementation → iterative implementation:

```
int fac(int n) { return facAcc(n, 1); }           int fac(int n) {  
int facAcc(int n, int acc) {                      int acc = 1;  
    if (n == 0)                                    for ( ; !(n == 0); n--) {  
        return acc;                                acc = acc * n;  
    else  
        return facAcc(n-1, acc * n);  
}                                              }  
                                              return acc;  
}
```

→ Running time of iterative implementation: $\Theta(n)$

→ Stack space of iterative implementation: $\Theta(1)$

In functional languages this simple translation is done by the compiler!

FIBONACCI NUMBERS

Math definition:

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2) \quad \text{when } n > 1$$

Recursive implementation:

FIBONACCI NUMBERS

Math definition:

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2) \quad \text{when } n > 1$$

Recursive implementation:

```
int fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

FIBONACCI NUMBERS

Math definition:

$$fib(0) = 1$$

$$fib(1) = 1$$

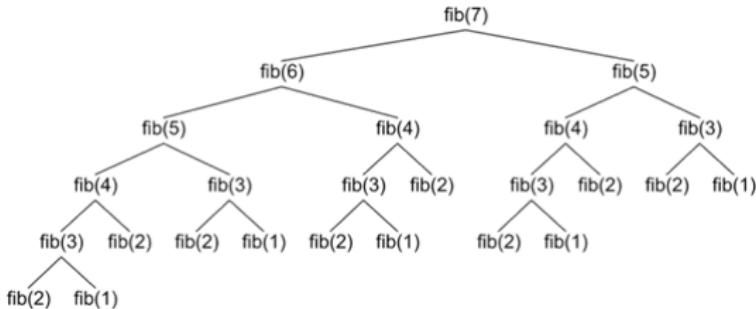
$$fib(n) = fib(n - 1) + fib(n - 2) \quad \text{when } n > 1$$

Recursive implementation:

```
int fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

→ Running time: non-tight upper bound: $O(2^n)$ tight bound: $\Theta(fib(n))$

→ # of recursive calls of $fib(n)$ is the size of the binary tree of recursive calls with n levels ($\leq 2^n$); however this is not a complete tree; thus $O(2^n)$.



FIBONACCI NUMBERS

Math definition:

$$fib(0) = 1$$

$$fib(1) = 1$$

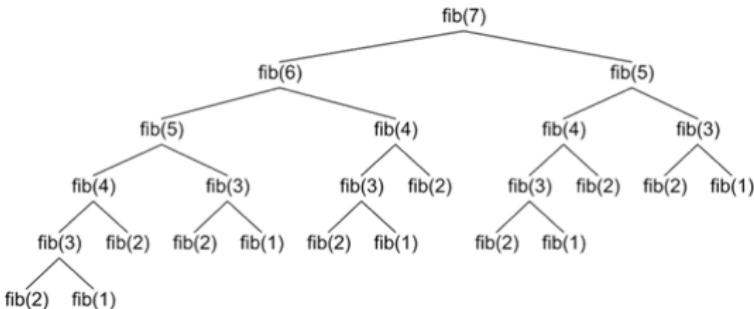
$$fib(n) = fib(n - 1) + fib(n - 2) \quad \text{when } n > 1$$

Recursive implementation:

```
int fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

→ Running time: non-tight upper bound: $O(2^n)$ tight bound: $\Theta(fib(n))$

→ # of recursive calls of $fib(n)$ is the size of the binary tree of recursive calls with n levels ($\leq 2^n$); however this is not a complete tree; thus $O(2^n)$.



→ Call stack space: $\Theta(n)$

FIBONACCI NUMBERS

Math definition:

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2) \quad \text{when } n > 1$$

Recursive implementation with accumulator (tail recursion):

```
int fib(int n) { return fibAcc(n, 1, 1); }

int fibAcc(int n, int last, int secondToLast) {
    if (n <= 1) return last;
    else return fibAcc(n-1, last + secondToLast, last);
}
```

- This is much trickier! Read it again off-line and understand why it works.
- It is a **bottom-up calculation** using two accumulators.

FIBONACCI NUMBERS

Math definition:

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2) \quad \text{when } n > 1$$

Recursive implementation with accumulator (tail recursion):

```
int fib(int n) { return fibAcc(n, 1, 1); }

int fibAcc(int n, int last, int secondToLast) {
    if (n <= 1) return last;
    else return fibAcc(n-1, last + secondToLast, last);
}
```

- This is much trickier! Read it again off-line and understand why it works.
- It is a **bottom-up calculation** using two accumulators.
- Running time: $\Theta(n)$
- Call stack space: $\Theta(n)$ in Java and $\Theta(1)$ in other languages.

FIBONACCI NUMBERS

Tail-recursive implementation → iterative implementation

```
int fib(int n) { return fibAcc(n, 1, 1); }

int fibAcc(int n, int last, int secondToLast) {
    if (n <= 1) return last;
    else
        return fibAcc(n-1, last + secondToLast, last);
}
```

```
int fib(int n) {
    int last = 1; int secondToLast = 1;
    for ( ; !(n <= 1); n--) {
        int tmpLast = last;
        last = last + secondToLast;
        secondToLast = tmpLast;
    }
    return last;
}
```

- Worst Case Asymptotic Running time: $\Theta(n)$
- Call stack space: $\Theta(1)$

GREATEST COMMON DIVISOR (GCD)

$\text{gcd}(x,y)$ is the largest number n such that $x \% n = y \% n = 0$.

For simplicity assume $x \leq y$.

GREATEST COMMON DIVISOR (GCD)

$\text{gcd}(x,y)$ is the largest number n such that $x \% n = y \% n = 0$.

For simplicity assume $x \leq y$.

Attempt 1: try all numbers n from $+\infty$ down to 1 until you find one that has the above property.

GREATEST COMMON DIVISOR (GCD)

$\text{gcd}(x,y)$ is the largest number n such that $x \% n = y \% n = 0$.

For simplicity assume $x \leq y$.

Attempt 1: try all numbers n from $+inf$ down to 1 until you find one that has the above property.

Attempt 2: try all numbers n from x down to 1 until you find one that has the above property (because gcd will necessarily be $\leq \min(x,y)$).

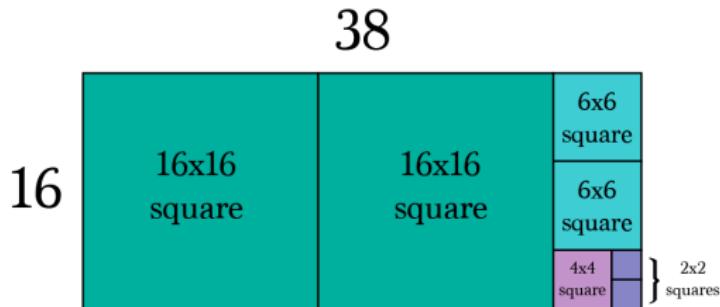
GREATEST COMMON DIVISOR (GCD)

$\text{gcd}(x,y)$ is the largest number n such that $x \% n = y \% n = 0$.

For simplicity assume $x \leq y$.

Attempt 3: Euclid's algorithm a Divide & Conquer approach:

Euclid's Theorem: $\text{gcd}(x,y) = \text{gcd}(y, x \% y)$.



The base case here is $\text{gcd}(x,0) = x$ (why is this the base case?).

→ Q: How many iterations in the worst case?

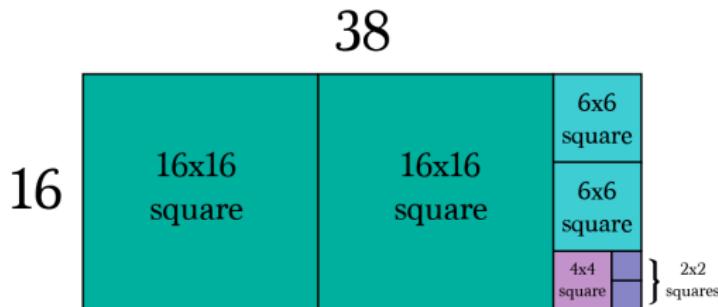
GREATEST COMMON DIVISOR (GCD)

$\text{gcd}(x,y)$ is the largest number n such that $x \% n = y \% n = 0$.

For simplicity assume $x \leq y$.

Attempt 3: Euclid's algorithm a Divide & Conquer approach:

Euclid's Theorem: $\text{gcd}(x,y) = \text{gcd}(y, x \% y)$.



The base case here is $\text{gcd}(x,0) = x$ (why is this the base case?).

- Q: How many iterations in the worst case?
- **Gabriel Lame's Theorem (1844):** #iterations < $5 \cdot h$
Where $h = \text{digits of } \min(x,y)$ (here this is x) in base 10.
- A: $O(\lg(x))$

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of **correctness** (Divide and Conquer is more obvious and similar to mathematical induction)

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of **correctness** (Divide and Conquer is more obvious and similar to mathematical induction)
- Iterative algorithms usually have easier proofs of **memory usage** (and in Java use less memory)

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of **correctness** (Divide and Conquer is more obvious and similar to mathematical induction)
- Iterative algorithms usually have easier proofs of **memory usage** (and in Java use less memory)
- Recursive algorithms are **sometimes easier to implement** when we have complex data structures.
 - because the compiler maintains a stack of previous calls for us.
 - We will use recursive implementations for most operations over **trees**.

RECURSION VS ITERATION: WHICH ONE TO CHOOSE?

- It **depends** on the benefits vs the drawbacks each implementation gives us in each case.
- Recursive algorithms usually have easier proofs of **correctness** (Divide and Conquer is more obvious and similar to mathematical induction)
- Iterative algorithms usually have easier proofs of **memory usage** (and in Java use less memory)
- Recursive algorithms are **sometimes easier to implement** when we have complex data structures.
 - because the compiler maintains a stack of previous calls for us.
 - We will use recursive implementations for most operations over **trees**.
- We *could* have used recursive implementations for operations over lists & arrays but:
 - they are not simpler than the iterative implementations
 - Java will need $O(n)$ call stack space to execute them.

HOMEWORK (OPTIONAL)

Homework 1: Implement Euclid's algorithm using

- A recursive method.
- An iterative method.

Homework 2: give recursive implementations for:

- binary search over an array
- linear search over a linked list

Homework 3: Implement **search** on a Binary Search Tree (next lecture) using recursion and compare with the iterative version in the book.

Homework 4:** give an iterative implementation for method **put** on binary search tree.

CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 11: Symbol Table ADT

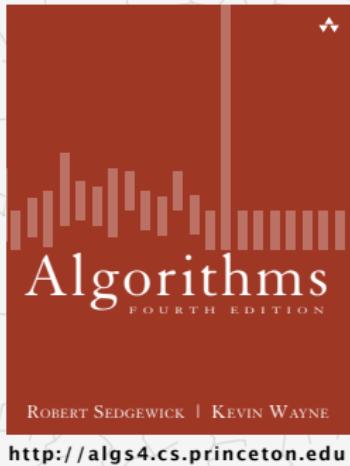
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

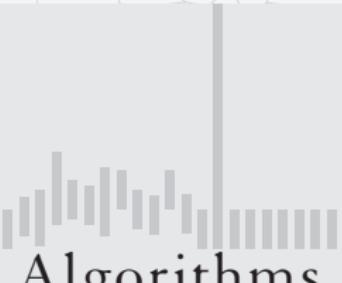
ROBERT SEDGEWICK | KEVIN WAYNE



3.1 SYMBOL TABLES

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

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



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.1 SYMBOL TABLES

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

Symbol tables

Key-value pair abstraction.

- **Insert** a value with specified key.
- Given a key, **search** for the corresponding value.

Ex. DNS lookup.

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

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

↑
key

↑
value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address	domain name	IP address
reverse DNS	find domain name	IP address	domain name
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and $N - 1$.

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

every array is an associative array every object is an associative array table is the only primitive data structure

```
hasNiceSyntaxForAssociativeArrays["Python"] = true  
hasNiceSyntaxForAssociativeArrays["Java"] = false
```

legal Python code

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```
public class ST<Key, Value>
```

```
    ST()
```

create an empty symbol table

```
    void put(Key key, Value val)
```

put key-value pair into the table $\leftarrow a[key] = val$

```
    Value get(Key key)
```

value paired with key $\leftarrow a[key]$

```
    boolean contains(Key key)
```

is there a value paired with key?

```
    void delete(Key key)
```

remove key (and its value) from table

```
    boolean isEmpty()
```

is the table empty?

```
    int size()
```

number of key-value pairs in the table

```
    Iterable<Key> keys()
```

all the keys in the table

Conventions

- Values are not null. ← Java allows null value
- Method get() returns null if key not present.
- Method put() overwrites old value with new value.

Intended consequences.

- Easy to implement contains().

```
public boolean contains(Key key)
{   return get(key) != null; }
```

- Can implement lazy version of delete().

```
public void delete(Key key)
{   put(key, null); }
```

Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are Comparable, use compareTo().
- Assume keys are any generic type, use equals() to test equality.
- Assume keys are any generic type, use equals() to test equality; use hashCode() to scramble key.

specify Comparable in API.

built-in to Java
(stay tuned)

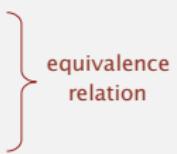
Best practices. Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

Equality test

All Java classes inherit a method `equals()`.

Java requirements. For any references x , y and z :

- Reflexive: $x.equals(x)$ is true.
 - Symmetric: $x.equals(y)$ iff $y.equals(x)$.
 - Transitive: if $x.equals(y)$ and $y.equals(z)$, then $x.equals(z)$.
 - Non-null: $x.equals(null)$ is false.
- 

Default implementation. $(x == y)$

do x and y refer to
the same object?



Customized implementations. `Integer`, `Double`, `String`, `java.io.File`, ...

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy.

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

    public boolean equals(Date that)
    {

        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

check that all significant
fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

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

```
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...
    @Override
    public boolean equals(Object y)
    {
```

Use `@Override` annotation
<http://docs.oracle.com/javase/tutorial/java/annotations/predefined.html>

must be Object.
Why? Experts still debate.

```
        if (y == this) return true;
```

optimize for true object equality

```
        if (y == null) return false;
```

check for null

```
        if (y.getClass() != this.getClass())
            return false;
```

objects must be in the same class
(religion: `getClass()` vs. `instanceof`)

```
        Date that = (Date) y;
```

cast is guaranteed to succeed

```
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
```

check that all significant
fields are the same

```
}
```

Equals design

"Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type and cast.
- Compare each significant field:
 - if field is a primitive type, use ==
 - if field is an object, use equals()
 - if field is an array, apply to each entry

but use Double.compare() with double
(or otherwise deal with -0.0 and NaN)

apply rule recursively

can use Arrays.deepEquals(a, b)
but not a.equals(b)

Best practices.

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make compareTo() consistent with equals().

e.g., cached Manhattan distance

x.equals(y) if and only if (x.compareTo(y) == 0)

ST test client for traces

Build ST by associating value i with i^{th} string from standard input.

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

output

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

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

ST test client for analysis

Frequency counter. Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
```

```
% java FrequencyCounter 1 < tinyTale.txt
it 10
```

```
% java FrequencyCounter 8 < tale.txt
business 122
```

```
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

tiny example
(60 words, 20 distinct)

real example
(135,635 words, 10,769 distinct)

real example
(21,191,455 words, 534,580 distinct)

Frequency counter implementation

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]);
        ST<String, Integer> st = new ST<String, Integer>();           ← create ST
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (word.length() < minlen) continue;                         ← ignore short strings
            if (!st.contains(word)) st.put(word, 1);
            else                      st.put(word, st.get(word) + 1);
        }
        String max = "";
        st.put(max, 0);
        for (String word : st.keys())
            if (st.get(word) > st.get(max))
                max = word;
        StdOut.println(max + " " + st.get(max));
    }
}
```

← read string and update frequency

← print a string with max freq



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.1 SYMBOL TABLES

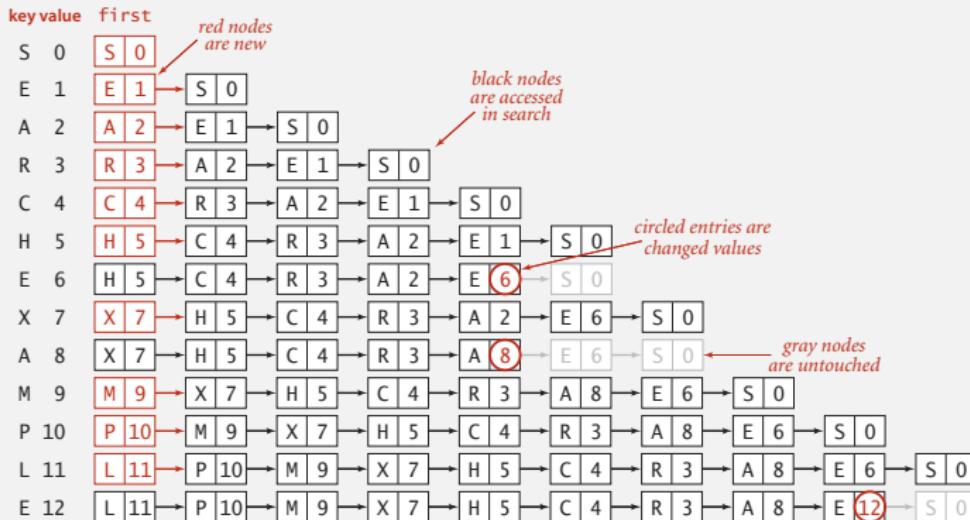
- ▶ API
- ▶ *elementary implementations*
- ▶ *ordered operations*

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	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	<code>equals()</code>

Challenge. Efficient implementations of both search and insert.

Binary search in an ordered array

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

Rank helper function. How many keys $< k$?

keys[]									
0	1	2	3	4	5	6	7	8	9
A	C	E	H	L	M	P	R	S	X

successful search for P

lo	hi	m		A	C	E	H	L	M	P	R	S	X
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 $\text{keys}[m] = P$: return 6

unsuccessful search for Q

lo	hi	m		A	C	E	H	L	M	P	R	S	X
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 $\text{lo} > \text{hi}$: return 7

Binary search: Java implementation

```
public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}
```

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

Binary search: trace of standard indexing client

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

keys[]										N	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	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$N / 2$	N	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	<code>compareTo()</code>

Challenge. Efficient implementations of both search and insert.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.1 SYMBOL TABLES

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

Examples of ordered symbol table API

	keys	values
min()	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13)	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00)	09:03:13	Chicago
	09:10:11	Seattle
select(7)	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00)	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00)	09:35:21	Chicago
	09:36:14	Seattle
max()	09:37:44	Phoenix

size(09:15:00, 09:25:00) is 5

rank(09:10:25) is 7

Ordered symbol table API

```
public class ST<Key extends Comparable<Key>, Value>
```

...

Key min()

smallest key

Key max()

largest key

Key floor(Key key)

largest key less than or equal to key

Key ceiling(Key key)

smallest key greater than or equal to key

int rank(Key key)

number of keys less than key

Key select(int k)

key of rank k

void deleteMin()

delete smallest key

void deleteMax()

delete largest key

int size(Key lo, Key hi)

number of keys between lo and hi

Iterable<Key> keys()

all keys, in sorted order

Iterable<Key> keys(Key lo, Key hi)

keys between lo and hi, in sorted order

Binary search: ordered symbol table operations summary

	sequential search	binary search
search	N	$\log N$
insert / delete	N	N
min / max	N	1
floor / ceiling	N	$\log N$
rank	N	$\log N$
select	N	1
ordered iteration	$N \log N$	N

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

CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 13: Binary Search Trees

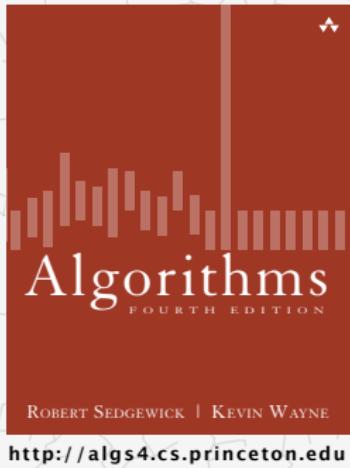
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



3.2 BINARY SEARCH TREES

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

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



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.2 BINARY SEARCH TREES

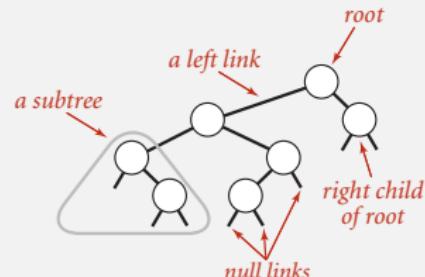
- ▶ **BSTs**
- ▶ *ordered operations*
- ▶ *deletion*

Binary search trees

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

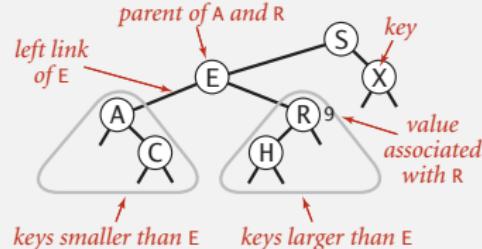
A binary tree is either:

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



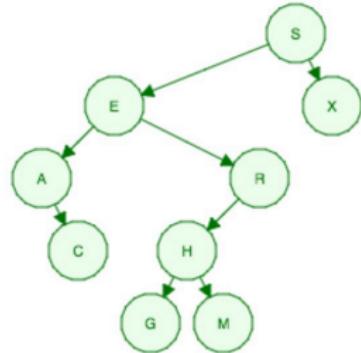
Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



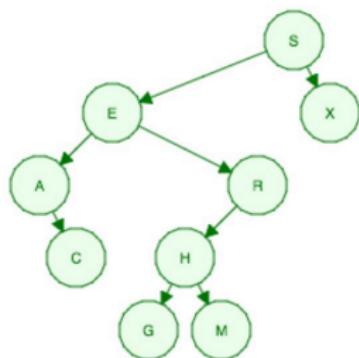
BINARY TREES: MORE DEFINITIONS

- **leaves** of tree: the nodes with no child nodes
- **height** of tree: the maximum number of **links** from the root to a leaf
- **levels** of tree: the maximum number of **nodes** from the root to a leaf (incl. root and leaf)
- **size** of tree: the number of nodes in the tree
- **depth** of a node: the number of **links** from the root to this node.



BINARY TREES: MORE DEFINITIONS

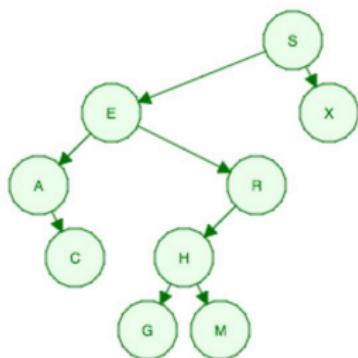
- **leaves** of tree: the nodes with no child nodes
- **height** of tree: the maximum number of **links** from the root to a leaf
- **levels** of tree: the maximum number of **nodes** from the root to a leaf (incl. root and leaf)
- **size** of tree: the number of nodes in the tree
- **depth** of a node: the number of **links** from the root to this node.



- Q: how many leaves in this tree?
- Q: what is the height of this tree?
- Q: how many levels in this tree?
- Q: what is the size of this tree?
- Q: what is the depth of 'H'?

BINARY TREES: MORE DEFINITIONS

- **leaves** of tree: the nodes with no child nodes
- **height** of tree: the maximum number of **links** from the root to a leaf
- **levels** of tree: the maximum number of **nodes** from the root to a leaf (incl. root and leaf)
- **size** of tree: the number of nodes in the tree
- **depth** of a node: the number of **links** from the root to this node.

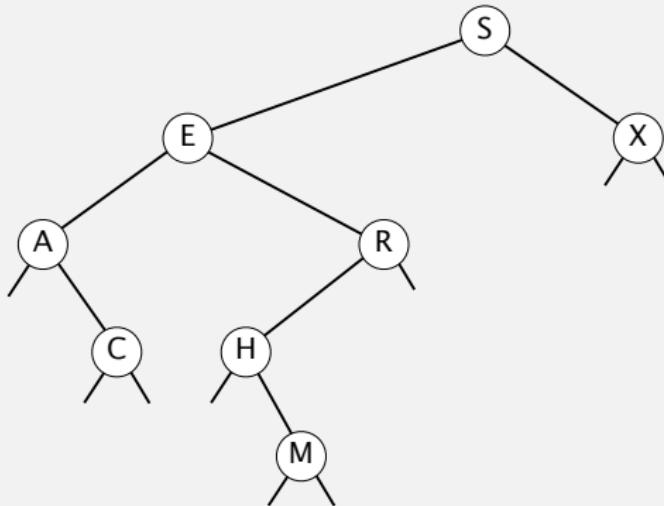


- Q: how many leaves in this tree? 4
- Q: what is the height of this tree? 4
- Q: how many levels in this tree? 5
- Q: what is the size of this tree? 9
- Q: what is the depth of 'H'? 3

Binary search tree demo

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

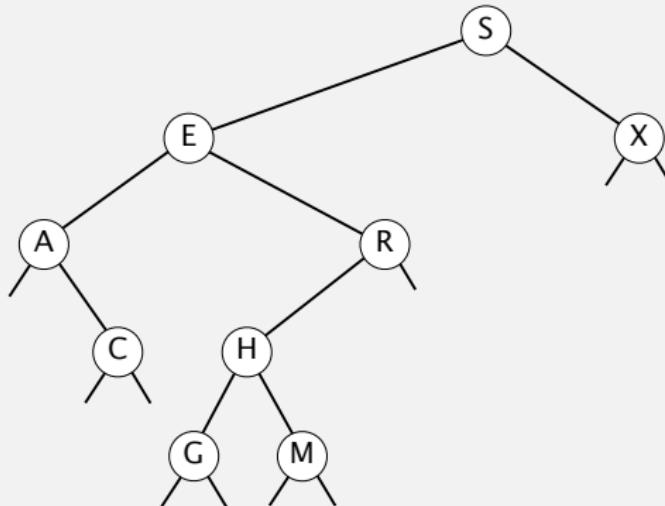
successful search for H



Binary search tree demo

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

insert G



BST representation in Java

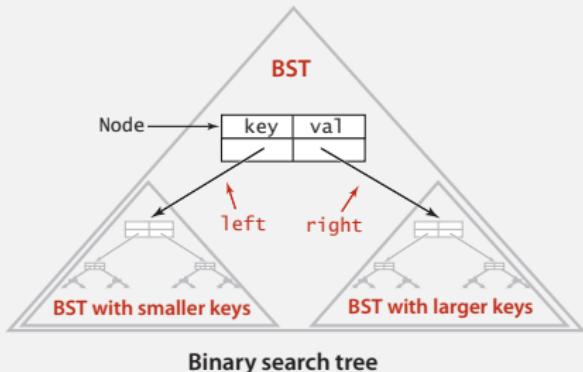
Java definition. A BST is a reference to a root Node.

A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.



```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```



Key and Value are generic types; Key is Comparable

BST implementation (skeleton)

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

    private class Node
    { /* see previous slide */ }

    public void put(Key key, Value val)
    { /* see next slides */ }

    public Value get(Key key)
    { /* see next slides */ }

    public void delete(Key key)
    { /* see next slides */ }

    public Iterable<Key> iterator()
    { /* see next slides */ }

}
```

BST search: Java implementation

Get. Return value corresponding to given key, or `null` if no such key.

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

Cost. Number of compares is equal to $1 + \text{depth of node}$.

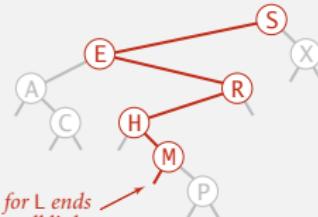
BST insert

Put. Associate value with key.

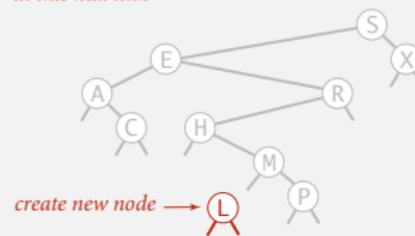
Search for key, then two cases:

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

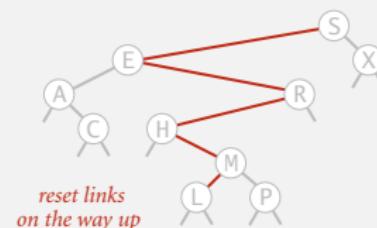
inserting L



search for L ends
at this null link



create new node → L



reset links
on the way up

Insertion into a BST

BST insert: Java implementation

Put. Associate value with key.

```
public void put(Key key, Value val)
{   root = put(root, key, val);   }

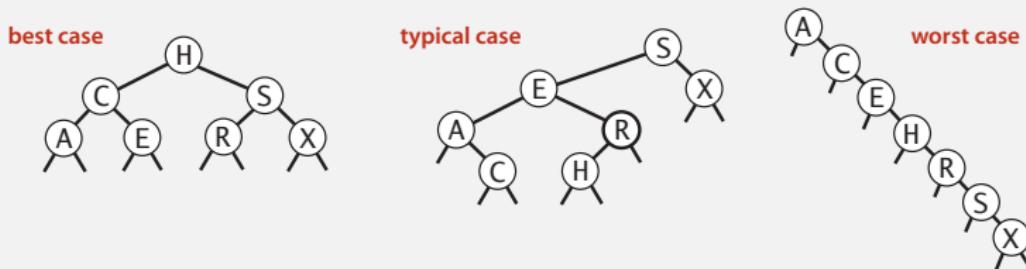
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if      (cmp  < 0)
        x.left  = put(x.left,  key, val);
    else if (cmp  > 0)
        x.right = put(x.right, key, val);
    else if (cmp == 0)
        x.val = val;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

concise, but tricky,
recursive code;
read carefully!

Cost. Number of compares is equal to $1 + \text{depth of node}$.

Tree shape

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



Bottom line. Tree shape depends on order of insertion.

BST insertion: random order visualization

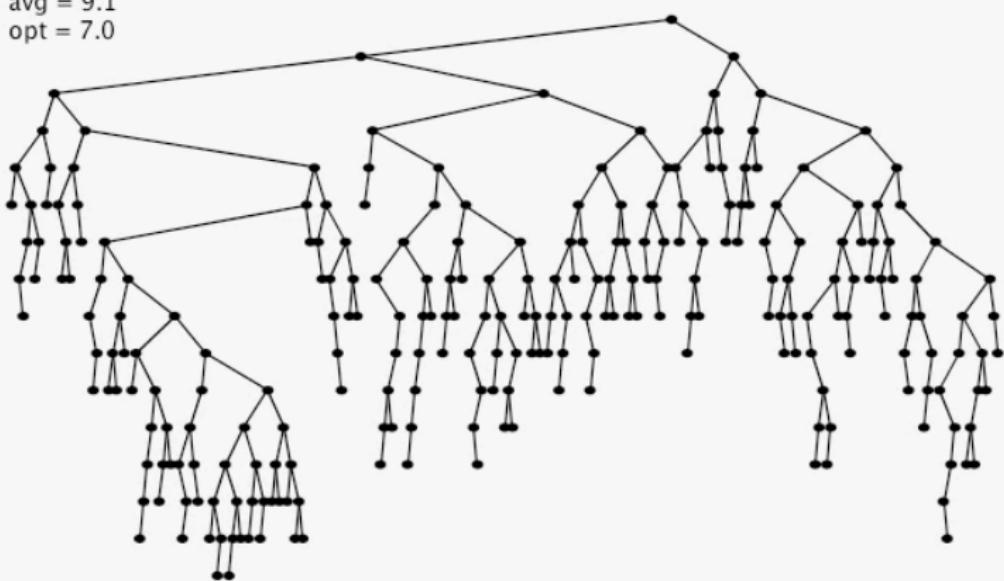
Ex. Insert keys in random order.

$N = 255$

$\text{max} = 16$

$\text{avg} = 9.1$

$\text{opt} = 7.0$



BSTs: mathematical analysis

Proposition. If N distinct keys are inserted into a BST in **random** order, the expected number of compares for a search/insert is $\sim 2 \ln N$.

Pf. 1-1 correspondence with quicksort partitioning.

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

How Tall is a Tree?

Bruce Reed
CNRS, Paris, France
reed@moka.ccr.jussieu.fr

ABSTRACT

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

But... Worst-case height is N .

[exponentially small chance when keys are inserted in random order]

ST implementations: summary

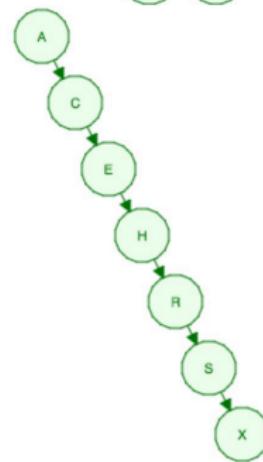
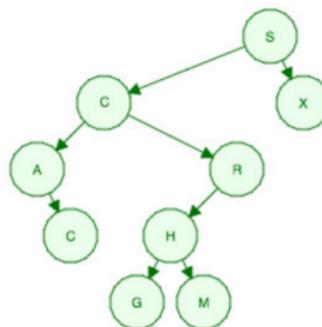
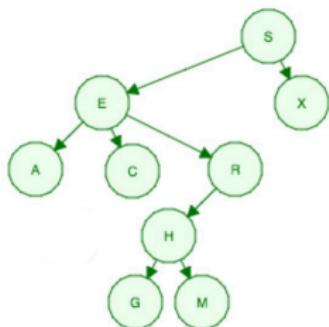
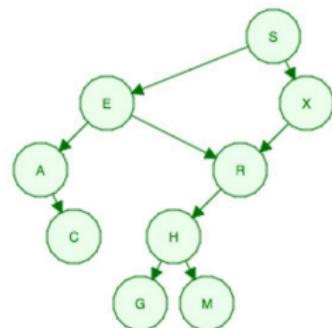
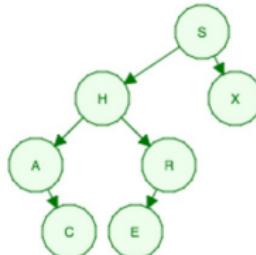
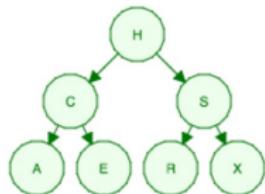
implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$\frac{1}{2}N$	N	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2}N$	<code>compareTo()</code>
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	<code>compareTo()</code>



Why not shuffle to ensure a (probabilistic) guarantee of $4.311 \ln N$?

QUIZ

Q: Which of the following are Binary Search Trees? Why?



CS2010: ALGORITHMS AND DATA STRUCTURES

Lecture 14: Binary Search Trees (2)

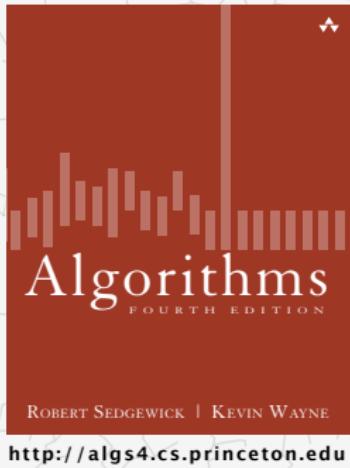
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ROBERT SEDGEWICK | KEVIN WAYNE

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

3.2 BINARY SEARCH TREES

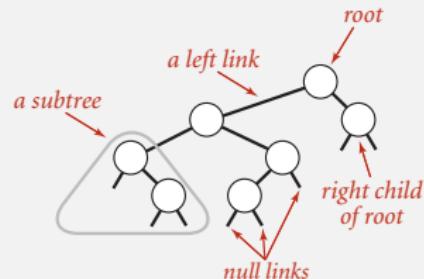
- ▶ *BSTs*
- ▶ *ordered operations*
- ▶ *deletion*

Binary search trees

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

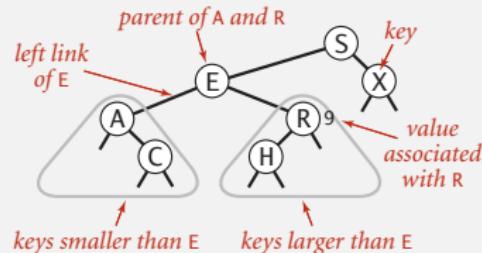
A binary tree is either:

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



Symmetric order. Each node has a key, and every node's key is:

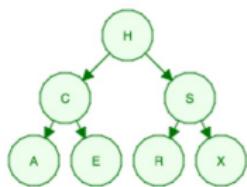
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



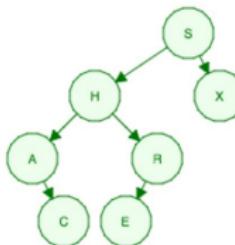
QUIZ

Q: Which of the following are Binary Search Trees? Why?

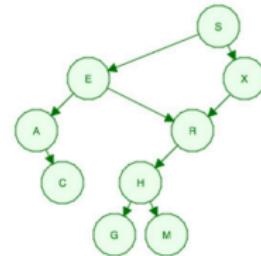
(1)



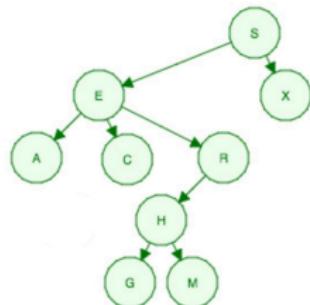
(2)



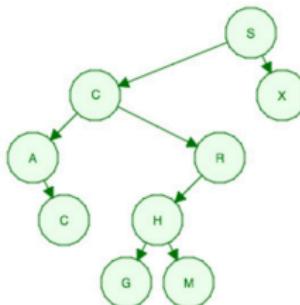
(3)



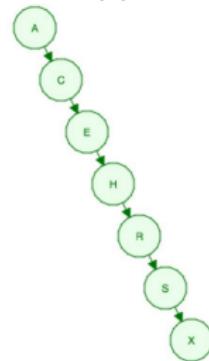
(4)



(5)



(6)



ST implementations: summary

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	$\frac{1}{2}N$	N	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	$\lg N$	$\frac{1}{2}N$	<code>compareTo()</code>
BST	N	N	$1.39 \lg N$	$1.39 \lg N$	<code>compareTo()</code>



Why not shuffle to ensure a (probabilistic) guarantee of $4.311 \ln N$?



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

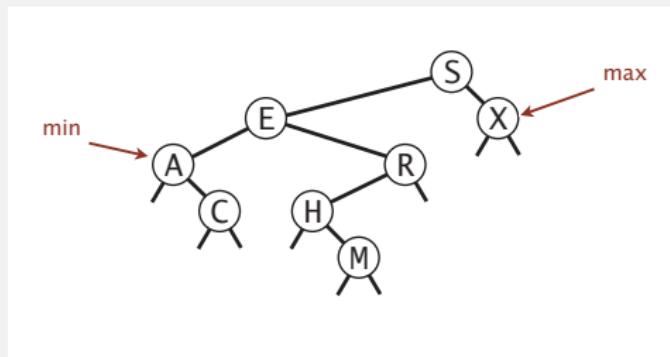
3.2 BINARY SEARCH TREES

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

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

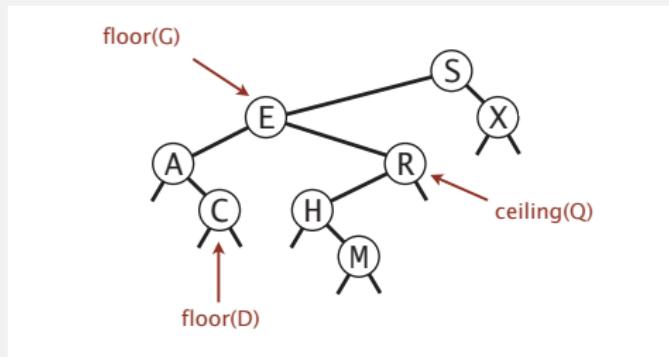


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key \leq a given key.

Ceiling. Smallest key \geq a given key.



Q. How to find the floor / ceiling?

Computing the floor

Case 1. [k equals the key in the node]

The floor of k is k .

Case 2. [k is less than the key in the node]

The floor of k is in the left subtree.

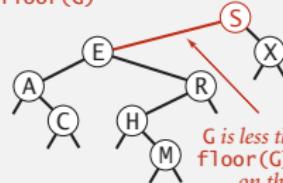
Case 3. [k is greater than the key in the node]

The floor of k is in the right subtree

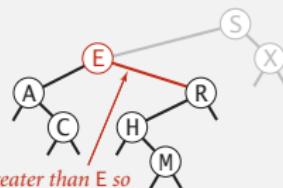
(if there is any key $\leq k$ in right subtree);

otherwise it is the key in the node.

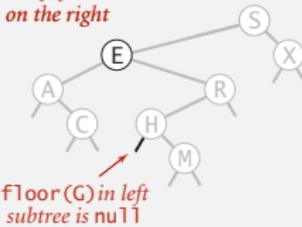
finding floor(G)



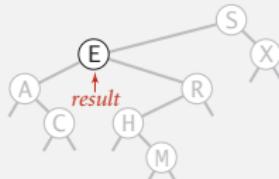
*G is less than S so
floor(G) must be
on the left*



*G is greater than E so
floor(G) could be
on the right*



*floor(G) in left
subtree is null*



result

Computing the floor

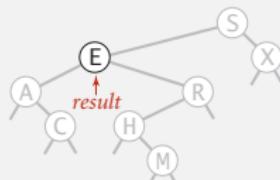
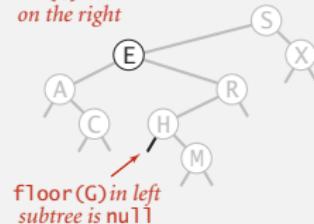
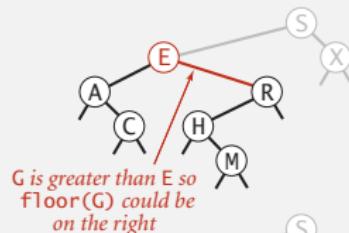
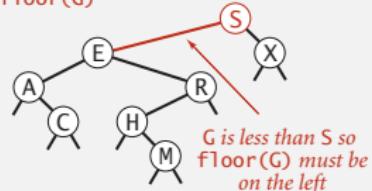
```
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;

    if (cmp < 0)  return floor(x.left, key);

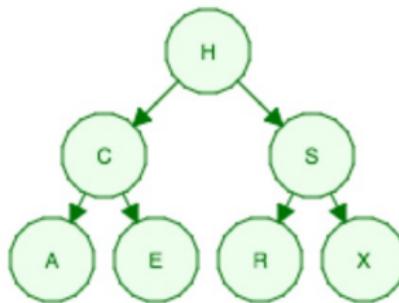
    Node t = floor(x.right, key);
    if (t != null) return t;
    else           return x;
}
```

finding floor(G)



RANK AND SELECT

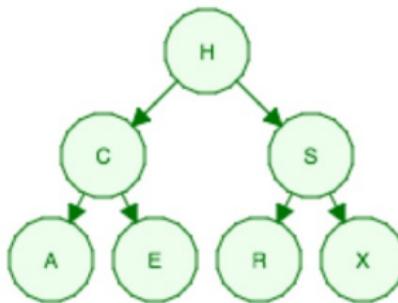
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



- Q: what is the rank of 'S'?

RANK AND SELECT

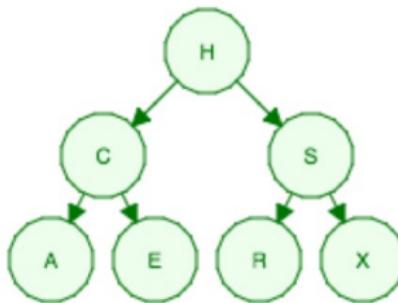
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



→ Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)

RANK AND SELECT

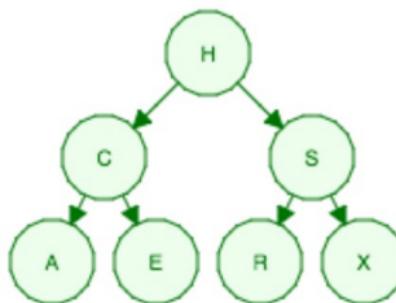
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



- Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- Q: what key has rank 4?

RANK AND SELECT

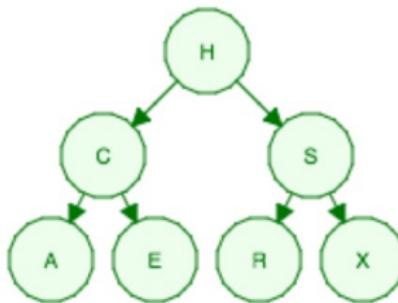
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



- Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- Q: what key has rank 4? 'R' (4 keys less than 'R' in the tree)

RANK AND SELECT

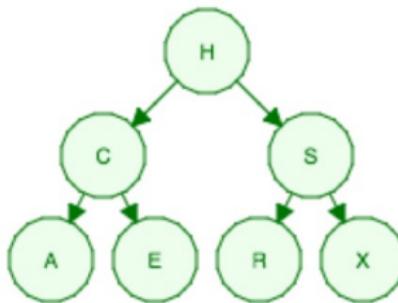
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



- Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- Q: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- Q: what is the rank of 'Q' ('Q' is not in the tree)?

RANK AND SELECT

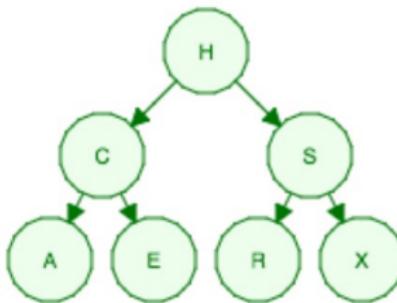
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



- Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- Q: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- Q: what is the rank of 'Q' ('Q' is not in the tree)? 4 (4 keys in the tree are less than 'Q')

RANK AND SELECT

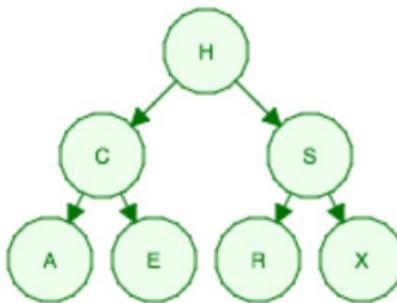
- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?



- Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- Q: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- Q: what is the rank of 'Q' ('Q' is not in the tree)? 4 (4 keys in the tree are less than 'Q')
- Q: what key has rank 7?

RANK AND SELECT

- `rank(Key k)`: how many keys less than k ?
- `select(int n)`: what key has rank n ?

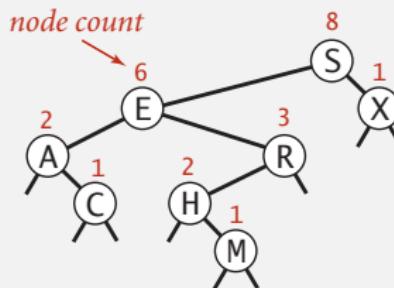


- Q: what is the rank of 'S'? 5 (5 keys less than 'S' in the tree)
- Q: what key has rank 4? 'R' (4 keys less than 'R' in the tree)
- Q: what is the rank of 'Q' ('Q' is not in the tree)? 4 (4 keys in the tree are less than 'Q')
- Q: what key has rank 7? no key in the tree has this rank (no key has 7 keys smaller than it in the tree)

Rank and select

Q. How to implement rank() and select() efficiently?

A. 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.



BST implementation: subtree counts

```
private class Node  
{  
    private Key key;  
    private Value val;  
    private Node left;  
    private Node right;  
    private int count;  
}
```

number of nodes in subtree

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

```
private int size(Node x)  
{  
    if (x == null) return 0;  
    return x.count;  }  
          ok to call  
          when x is null
```

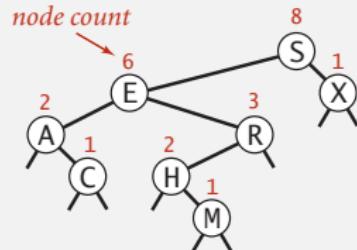
```
private Node put(Node x, Key key, Value val)  
{  
    if (x == null) return new Node(key, val, 1);  
    int cmp = key.compareTo(x.key);  
    if (cmp < 0) x.left = put(x.left, key, val);  
    else if (cmp > 0) x.right = put(x.right, key, val);  
    else if (cmp == 0) x.val = val;  
    x.count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

initialize subtree
count to 1

Rank

Rank. How many keys $< k$?

Easy recursive algorithm (3 cases!)



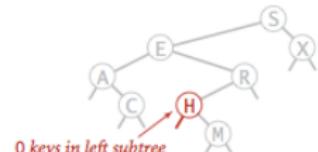
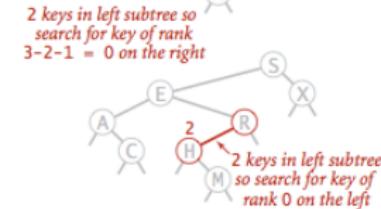
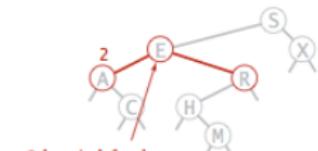
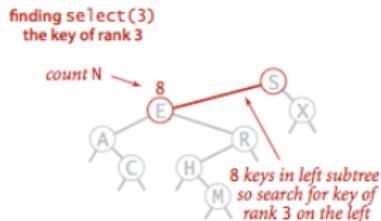
```
public int rank(Key key)
{   return rank(key, root);  }

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

SELECT

Select. Find the key with rank n .

```
public Key select(int n) {  
    if (n < 0 || n >= size()) return null;  
    Node x = select(root, n);  
    return x.key;  
}  
  
private Node select(Node x, int n) {  
    if (x == null) return null;  
    int t = size(x.left);  
    if (t > n) return select(x.left, n);  
    else if (t < n) return select(x.right, n-t-1);  
    else  
        return x;  
}
```



Selection in a BST

TREE TRAVERSALS

Task: Process all nodes of the tree.

Purpose: To print all nodes, to add all nodes in a datastructure (e.g. queue), etc.

Three kinds of traversals:

→ **inorder:** for each node:

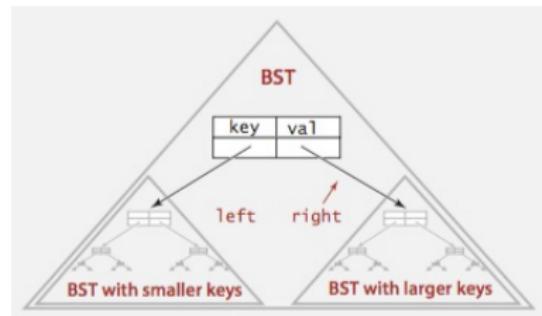
1. traverse the left subtree
2. **process the node**
3. traverse the right subtree

→ **preorder:** for each node:

1. **process the node**
2. traverse the left subtree
3. traverse the right subtree

→ **postorder:** for each node:

1. traverse the left subtree
2. traverse the right subtree
3. **process the node**

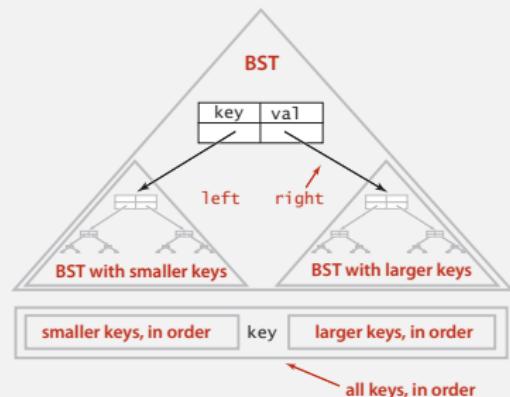


Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

```
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```



Property. Inorder traversal of a BST yields keys in ascending order.

BST: ordered symbol table operations summary

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

h = height of BST
(proportional to $\log N$)
if keys inserted in random order

Worst case: $h = O(N)$

order of growth of running time of ordered symbol table operations



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.2 BINARY SEARCH TREES

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

ST implementations: summary

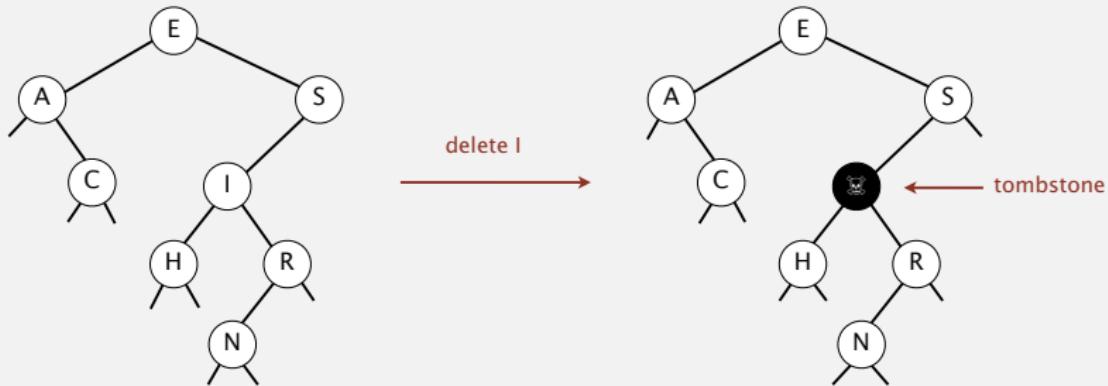
implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$???	✓	<code>compareTo()</code>

Next. Deletion in BSTs.

BST deletion: lazy approach

To remove a node with a given key:

- Set its value to null.
- Leave key in tree to guide search (but don't consider it equal in search).



Cost. $\sim 2 \ln N'$ per insert, search, and delete (if keys in random order), where N' is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) overload.

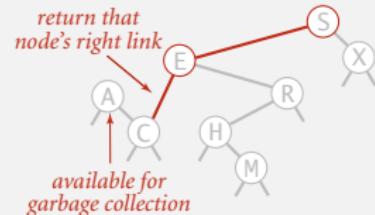
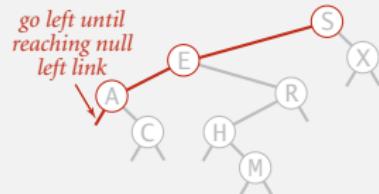
Deleting the minimum

To delete the minimum key:

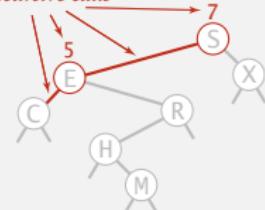
- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

```
public void deleteMin()
{   root = deleteMin(root);  }

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



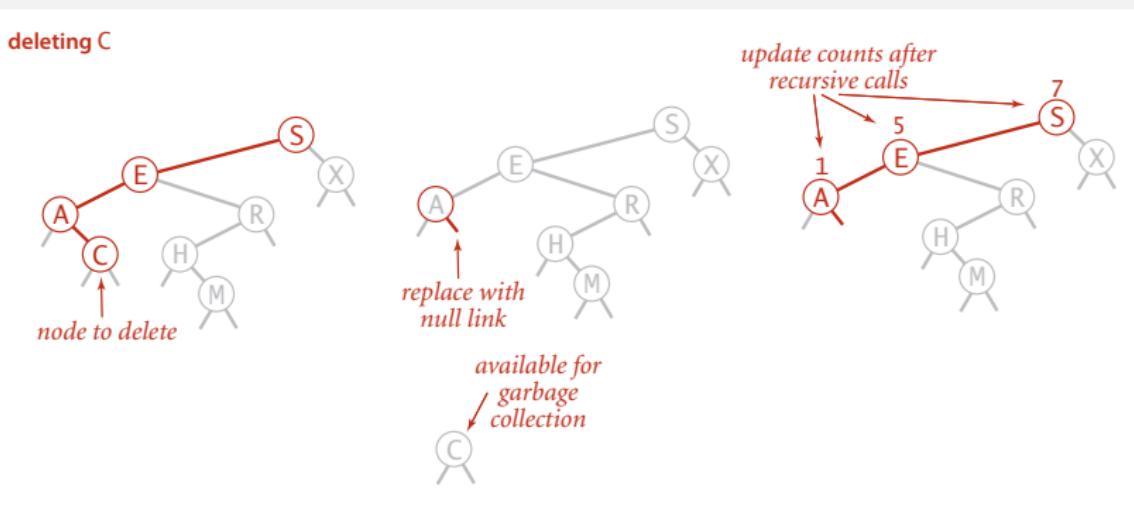
update links and node counts
after recursive calls



Hibbard deletion

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

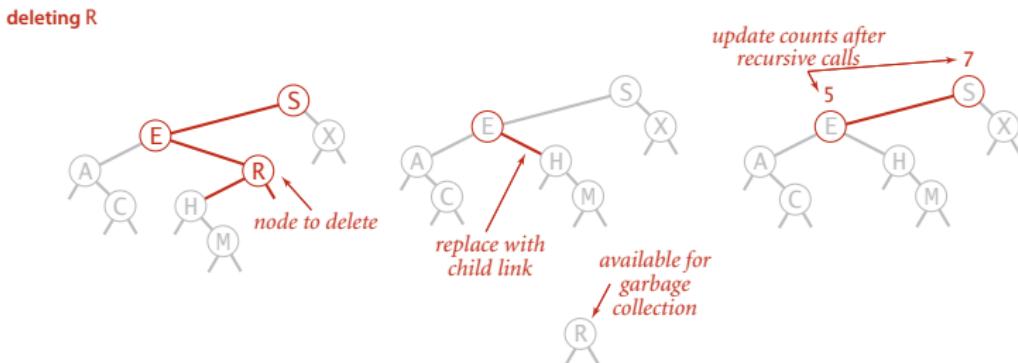
Case 0. [0 children] Delete t by setting parent link to null.



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.



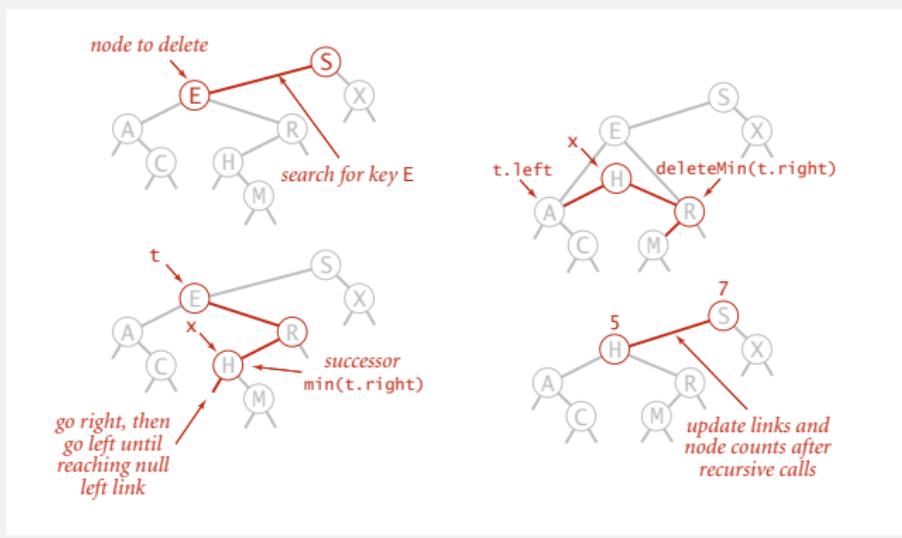
Hibbard deletion

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

Case 2. [2 children]

- Find successor x of t .
- Delete the minimum in t 's right subtree.
- Put x in t 's spot.

← x has no left child
← but don't garbage collect x
← still a BST



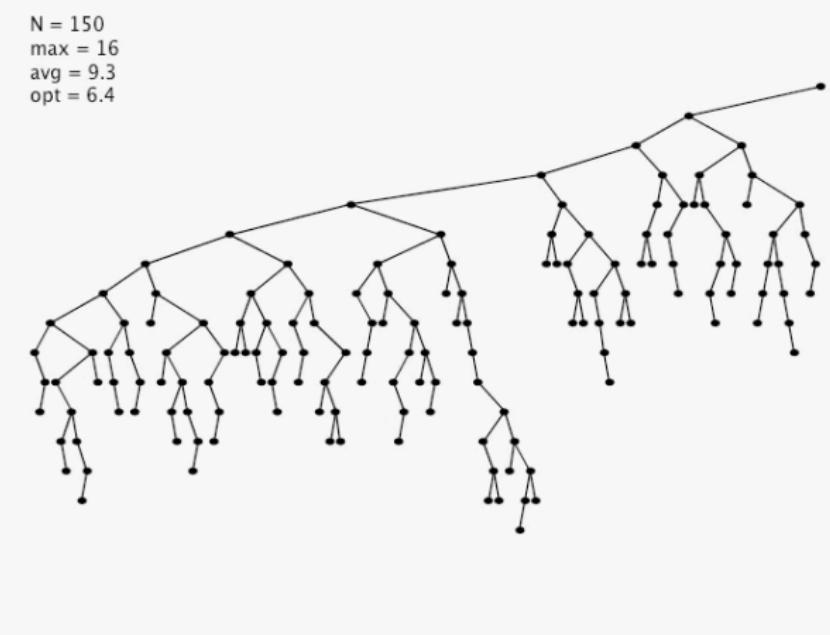
Hibbard deletion: Java implementation

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

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = delete(x.left,  key);           ← search for key
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;                      ← no right child
        if (x.left  == null) return x.right;                     ← no left child
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);                          ← replace with successor
        x.left = t.left;
    }
    x.count = size(x.left) + size(x.right) + 1;             ← update subtree counts
    return x;
}
```

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.

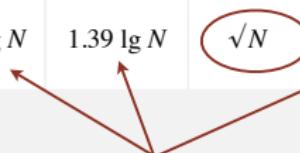


Surprising consequence. Trees not random (!) $\Rightarrow \sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>



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

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

CS2010: ALGORITHMS AND DATA STRUCTURES

Lectures 15: Balanced Binary Search Trees

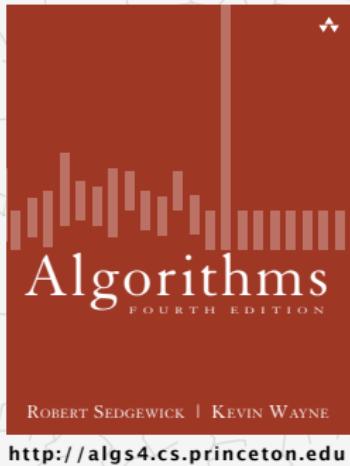
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ROBERT SEDGEWICK | KEVIN WAYNE

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

3.3 BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ red-black BSTs
- ▶ B-trees

Symbol table review

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black BSTs, B-trees.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.3 BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ red-black BSTs
- ▶ B-trees

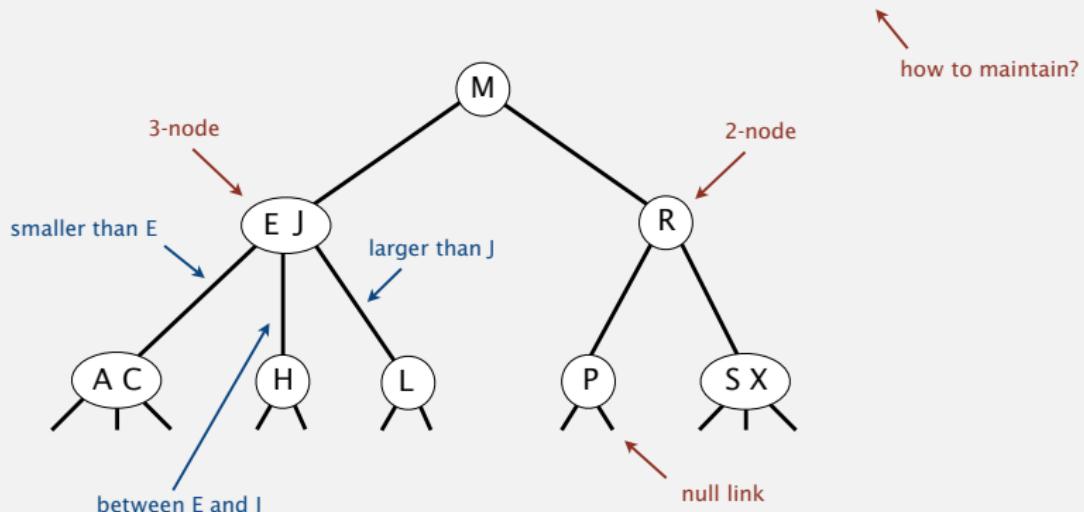
2-3 tree

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.



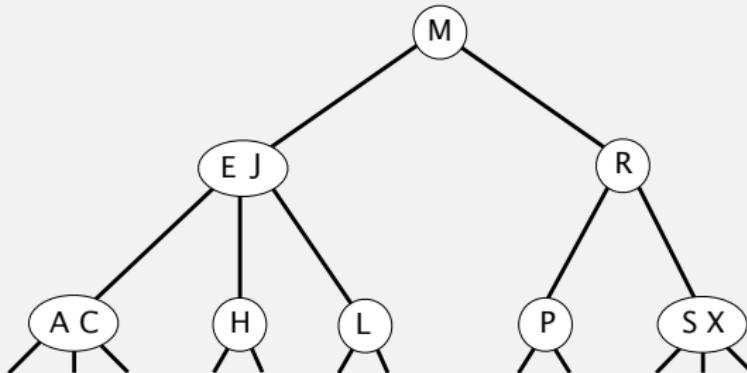
2-3 tree demo

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).



search for H

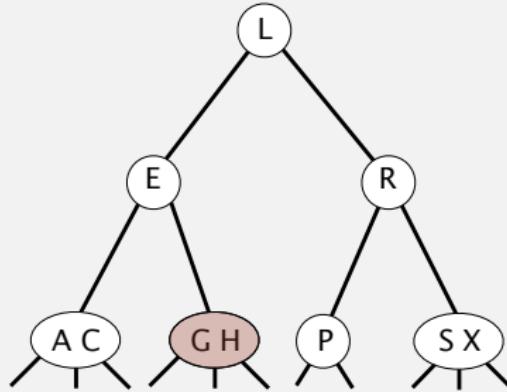
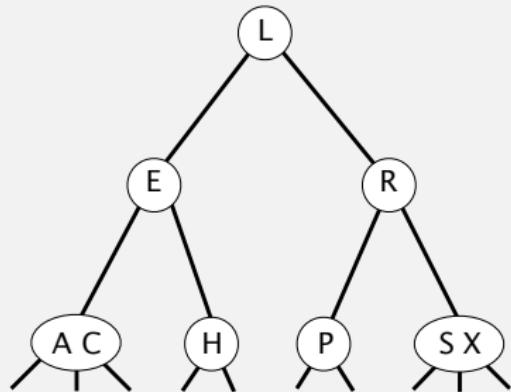


Insertion into a 2-3 tree

Insertion into a 2-node at bottom.

- Add new key to 2-node to create a 3-node.

insert G

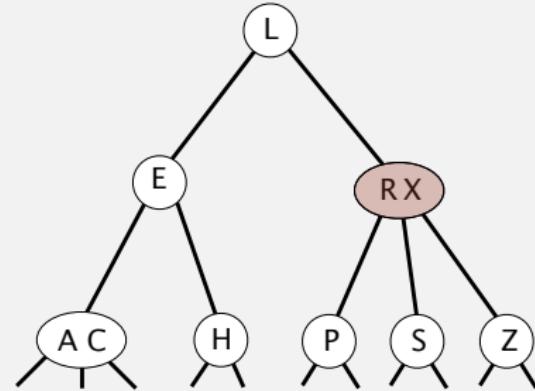
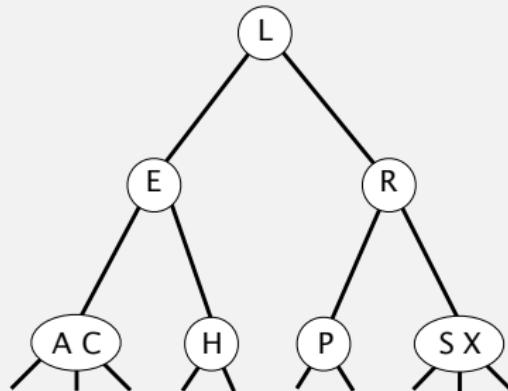


Insertion into a 2-3 tree

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert Z



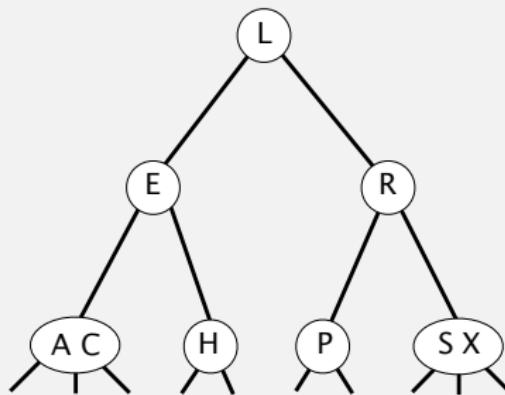
2-3 tree construction demo

insert S



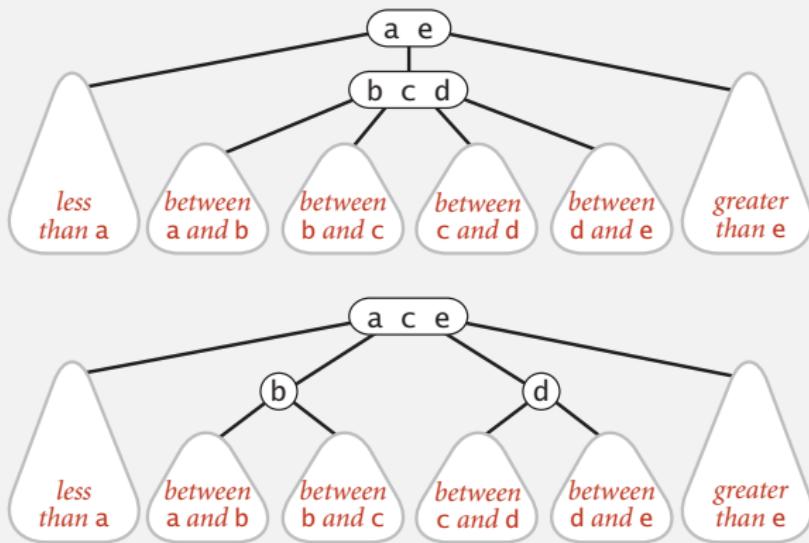
2-3 tree construction demo

2-3 tree



Local transformations in a 2-3 tree

Splitting a 4-node is a **local** transformation: constant number of operations.



Global properties in a 2-3 tree

Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.

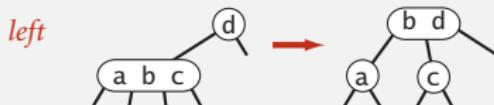
root



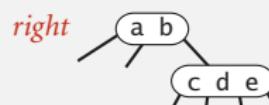
parent is a 3-node



parent is a 2-node



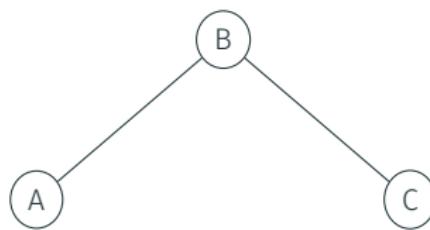
right



QUESTION

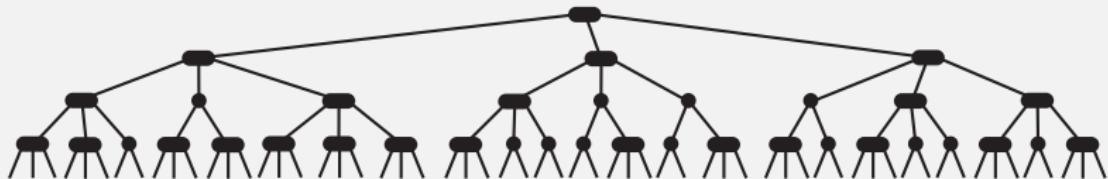
Insert the keys: A, B, C

Q: which of the following trees do we get?



2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

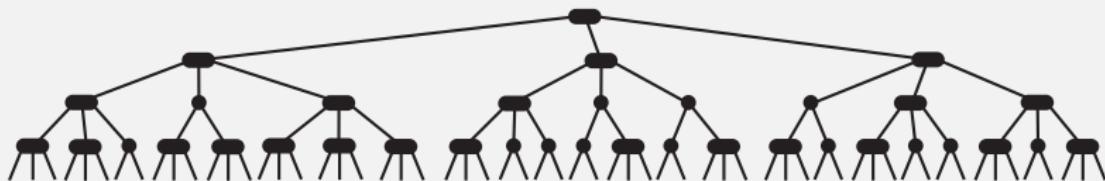


Tree height.

- 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.

Bottom line. Guaranteed logarithmic performance for search and insert.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✓	<code>compareTo()</code>



 constant c 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.

fantasy code

```
public void put(Key key, Value val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChildKey();
        if (x.is4Node()) x.split();
    }
    if      (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
```

Bottom line. Could do it, but there's a better way.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

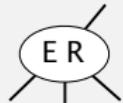
<http://algs4.cs.princeton.edu>

3.3 BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ red-black BSTs
- ▶ B-trees

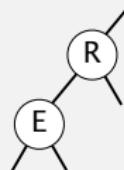
How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?



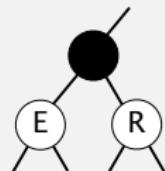
Approach 1: regular BST.

- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.



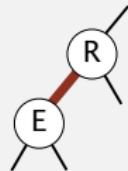
Approach 2: regular BST with "glue" nodes.

- Wastes space, wasted link.
- Code probably messy.



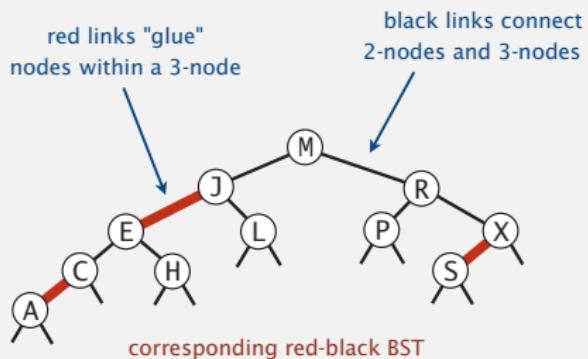
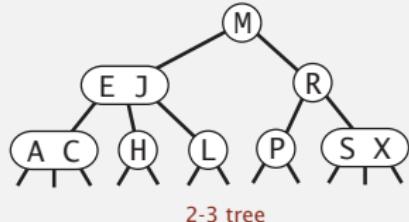
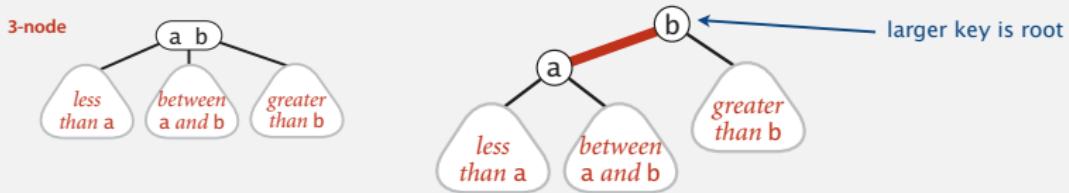
Approach 3: regular BST with red "glue" links.

- Widely used in practice.
- Arbitrary restriction: red links lean left.



Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.

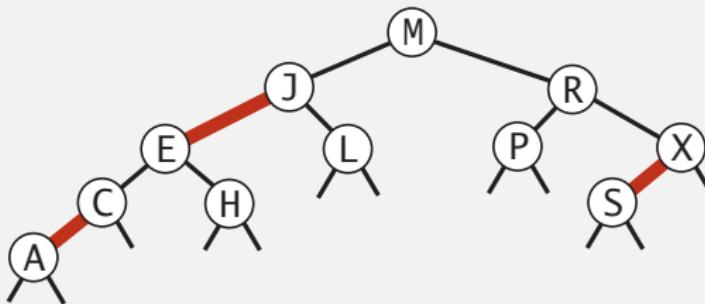


An equivalent definition

A BST such that:

- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

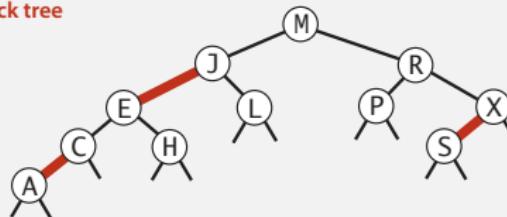
"perfect black balance"



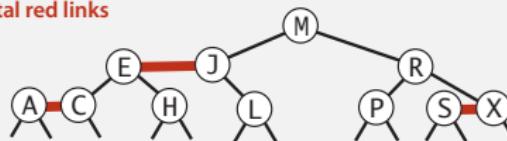
Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1-1 correspondence between 2-3 and LLRB.

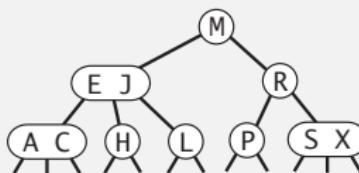
red–black tree



horizontal red links



2-3 tree

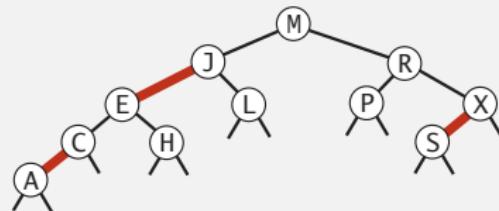


Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).

but runs faster
because of better balance

```
public Val get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```



Remark. Most other ops (e.g., floor, iteration, selection) are also identical.

Red-black BST representation

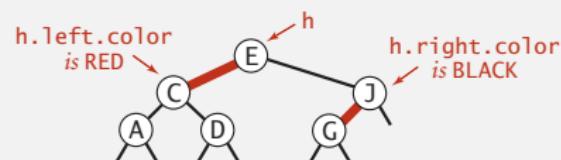
Each node is pointed to by precisely one link (from its parent) \Rightarrow
can encode color of links in nodes.

```
private static final boolean RED  = true;
private static final boolean BLACK = false;

private class Node
{
    Key key;
    Value val;
    Node left, right;
    boolean color; // color of parent link
}

private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

null links are black



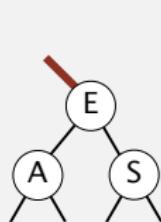
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

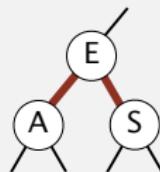
During internal operations, maintain:

- Symmetric order.
- Perfect black balance.

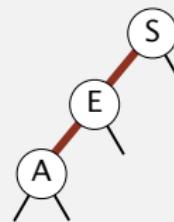
[but not necessarily color invariants]



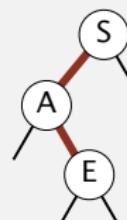
right-leaning
red link



two red children
(a temporary 4-node)



left-left red
(a temporary 4-node)



left-right red
(a temporary 4-node)

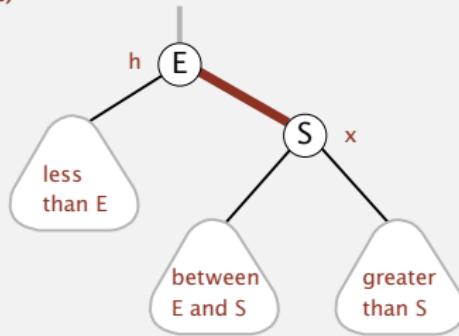
How? Apply elementary red-black BST operations: rotation and color flip.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(before)



```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

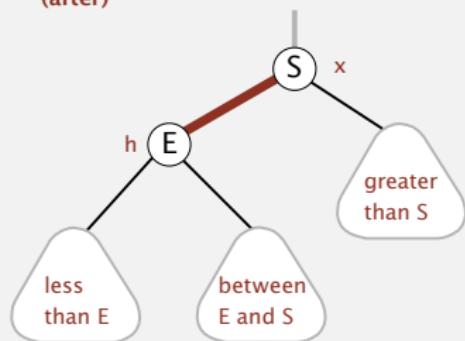
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left

(after)



```
private Node rotateLeft(Node h)
{
    assert isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

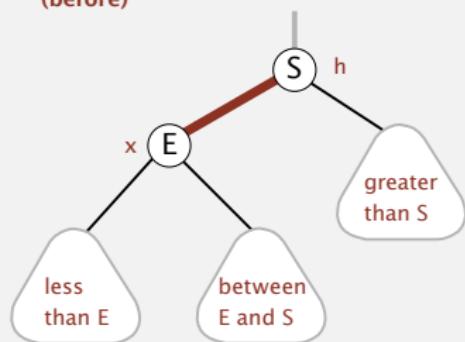
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(before)



```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

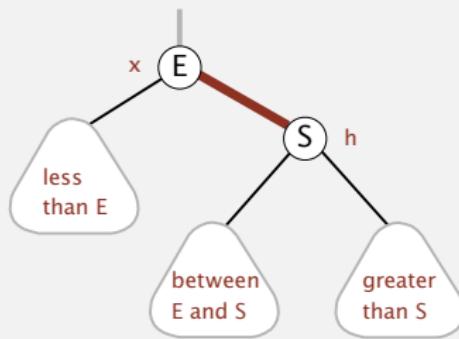
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right

(after)

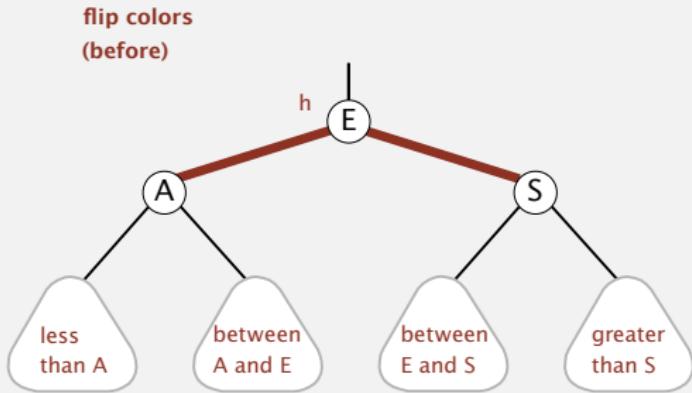


```
private Node rotateRight(Node h)
{
    assert isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

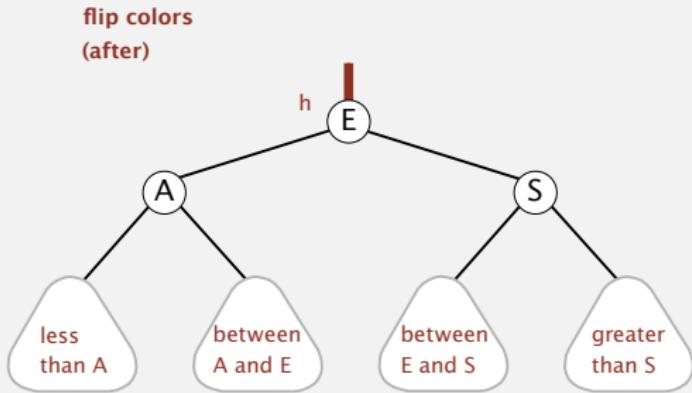


```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.



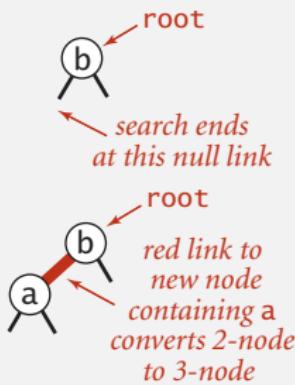
```
private void flipColors(Node h)
{
    assert !isRed(h);
    assert isRed(h.left);
    assert isRed(h.right);
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

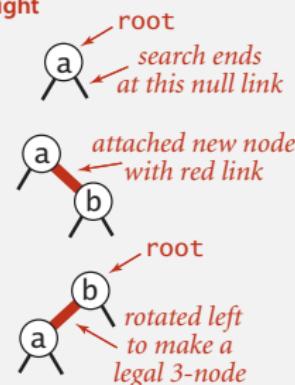
Insertion in a LLRB tree

Warmup 1. Insert into a tree with exactly 1 node.

left



right

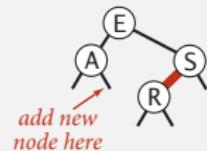


Insertion in a LLRB tree

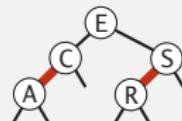
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- If new red link is a right link, rotate left. ← to fix color invariants

insert C



right link red
so rotate left



Insertion in a LLRB tree

Warmup 2. Insert into a tree with exactly 2 nodes.

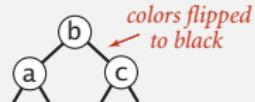
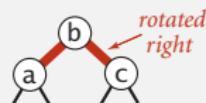
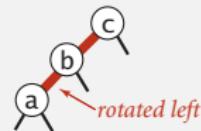
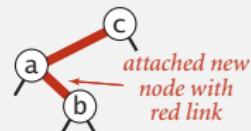
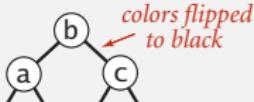
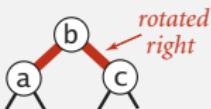
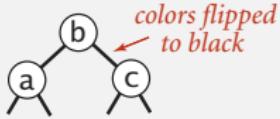
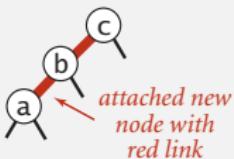
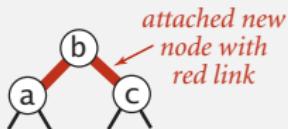
larger



smaller



between



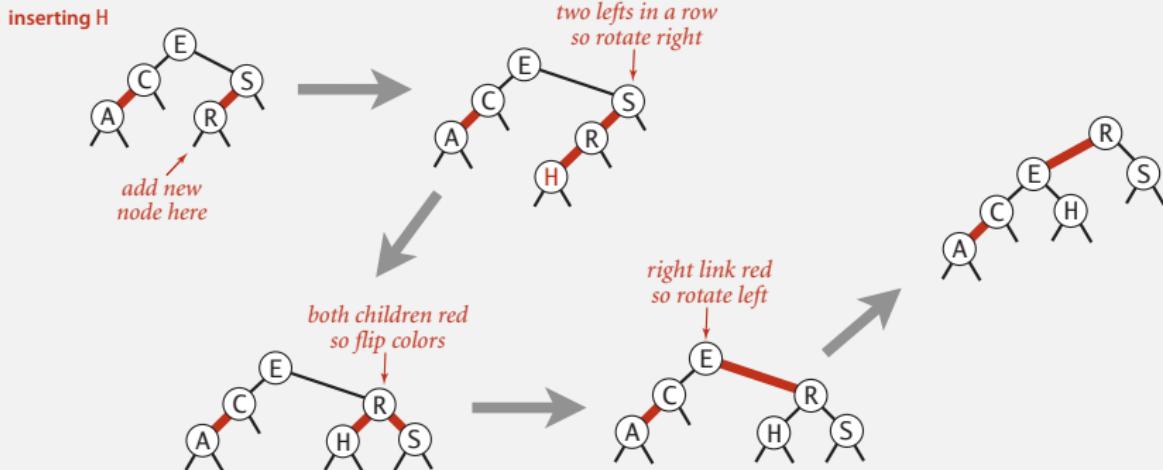
Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red.
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

to maintain symmetric order
and perfect black balance

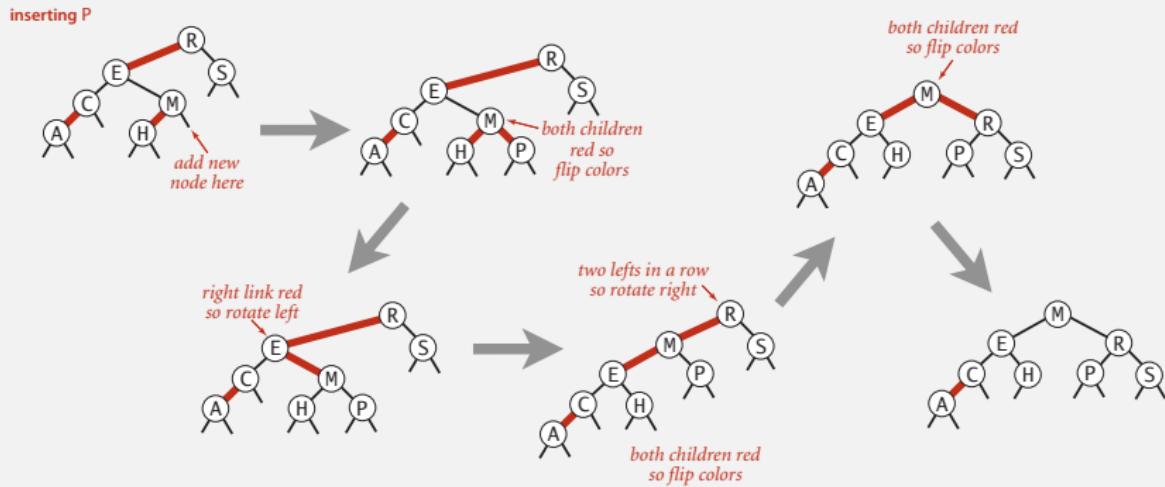
to fix color invariants



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. ← to maintain symmetric order and perfect black balance
- 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). ← to fix color invariants



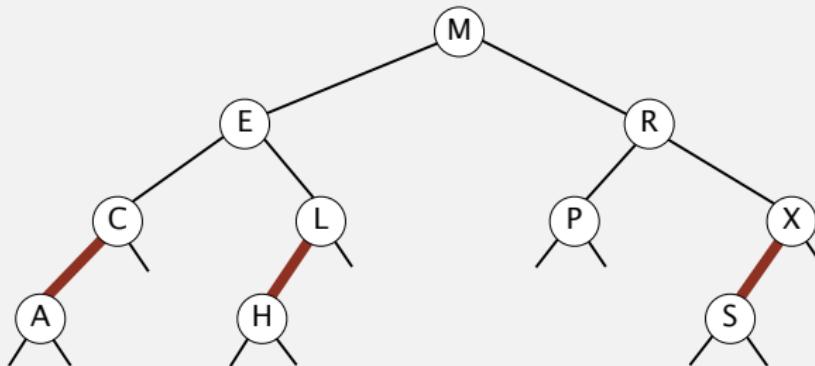
Red-black BST construction demo

insert S



Red-black BST construction demo

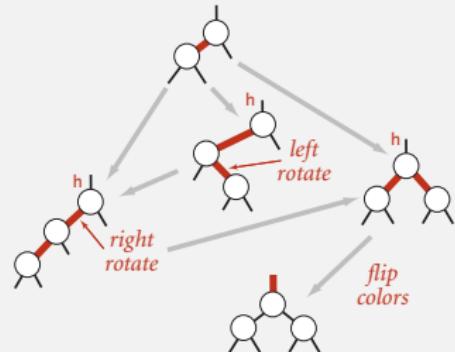
red-black BST



Insertion in a LLRB tree: Java implementation

Same code for all 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); ← insert at bottom
    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else if (cmp == 0) h.val   = val;

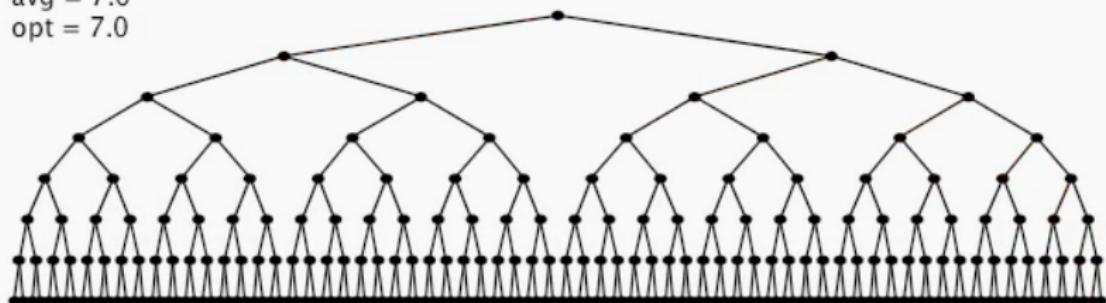
    if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h); ← lean left
    if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h); ← balance 4-node
    if (isRed(h.left)  && isRed(h.right))       flipColors(h); ← split 4-node

    return h;
}
```

only a few extra lines of code provides near-perfect balance

Insertion in a LLRB tree: visualization

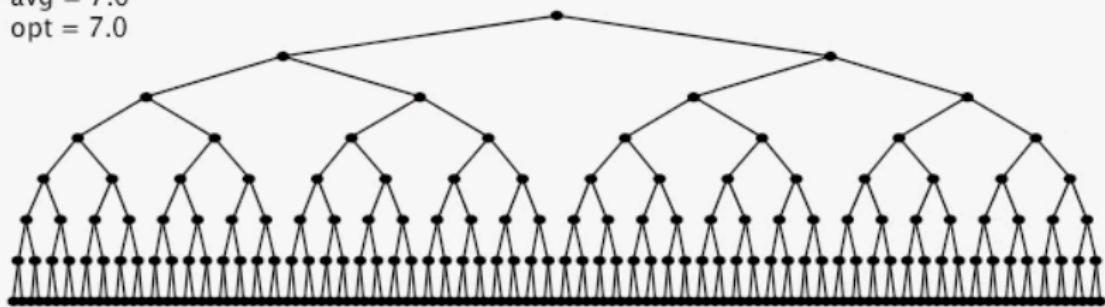
N = 255
max = 8
avg = 7.0
opt = 7.0



255 insertions in ascending order

Insertion in a LLRB tree: visualization

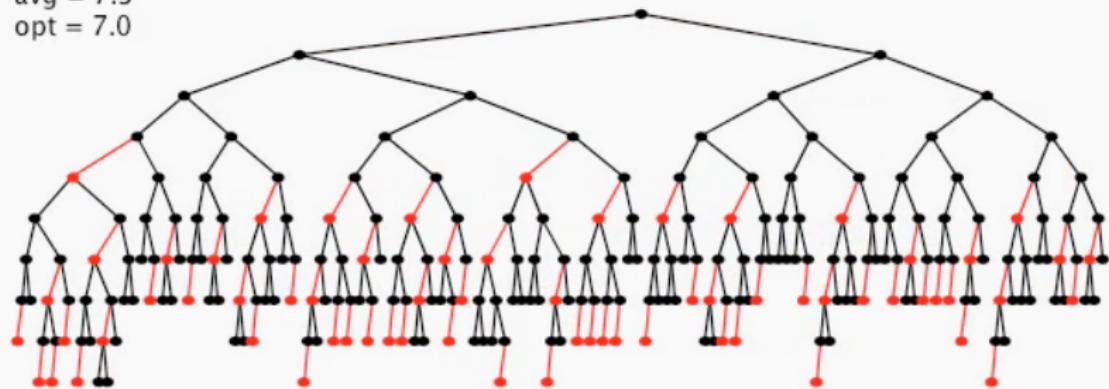
N = 255
max = 8
avg = 7.0
opt = 7.0



255 insertions in descending order

Insertion in a LLRB tree: visualization

N = 255
max = 10
avg = 7.3
opt = 7.0



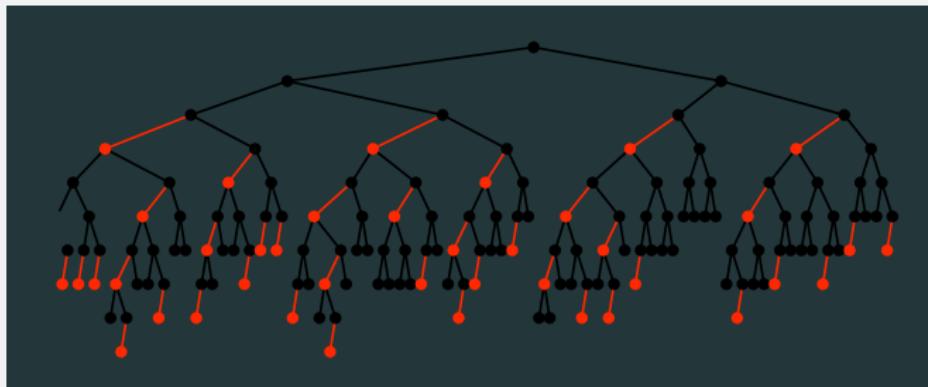
255 random insertions

Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



Property. Height of tree is $\sim 1.0 \lg N$ in typical applications.

ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
2–3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N^*$	$1.0 \lg N^*$	$1.0 \lg N^*$	✓	<code>compareTo()</code>

* exact value of coefficient unknown but extremely close to 1

War story: why red-black?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.
- ...



Xerox Alto

A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Leo J. Guibas
*Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University*

Robert Sedgewick*
*Program in Computer Science
Brown University
Providence, R. I.*

ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its

War story: red-black BSTs

Telephone company contracted with database provider to build real-time database to store customer information.

Database implementation.

- Red-black BST search and insert; Hibbard deletion.
- Exceeding height limit of 80 triggered error-recovery process.

allows for up to 2^{40} keys

Extended telephone service outage.

- Main cause = height bounded exceeded!
- Telephone company sues database provider.
- Legal testimony:

Hibbard deletion
was the problem



“If implemented properly, the height of a red-black BST with N keys is at most $2 \lg N$. ” — expert witness





Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.3 BALANCED SEARCH TREES

- ▶ 2-3 search trees
- ▶ red-black BSTs
- ▶ B -trees

File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).



Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

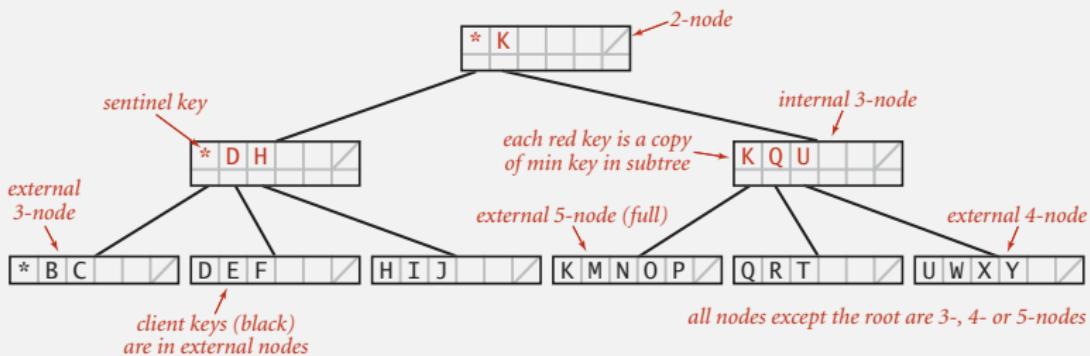
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

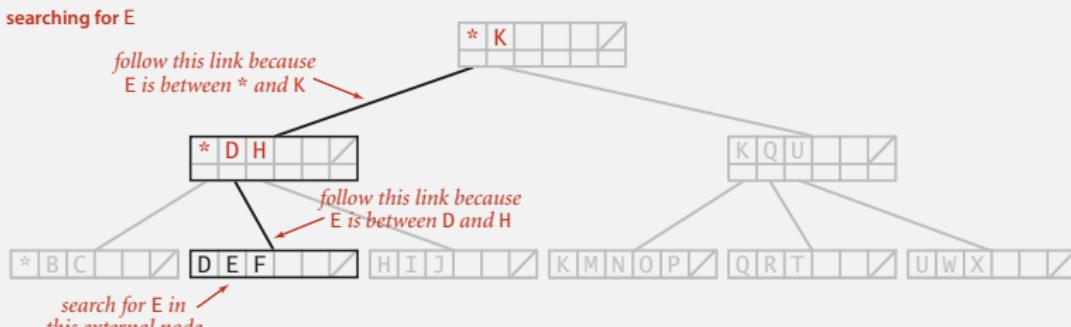
choose M as large as possible so that M links fit in a page, e.g., $M = 1024$



Anatomy of a B-tree set ($M = 6$)

Searching in a B-tree

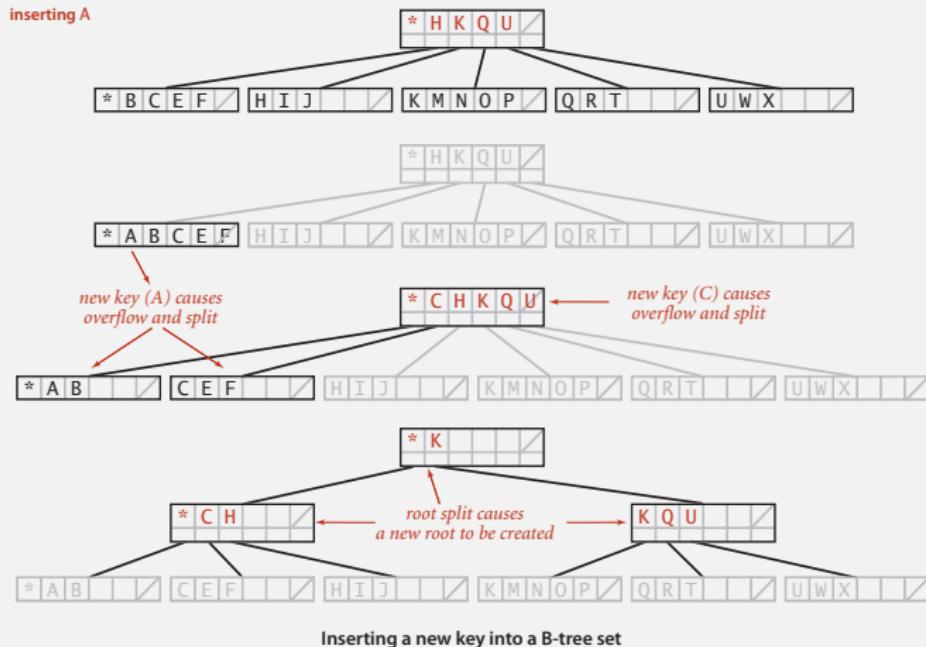
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set ($M = 6$)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Balance in B-tree

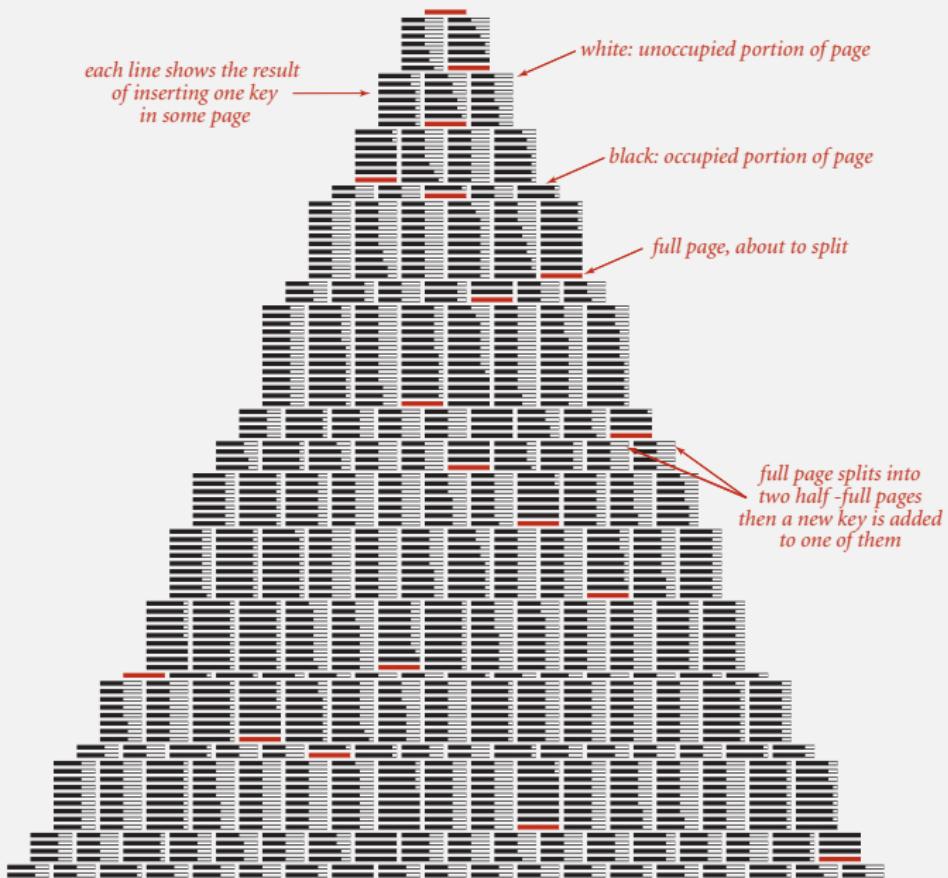
Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

Pf. All internal nodes (besides root) have between $M/2$ and $M - 1$ links.

In practice. Number of probes is at most 4. $\leftarrow M = 1024; N = 62 \text{ billion}$
 $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Building a large B tree



Balanced trees in the wild

Red-black trees are widely used as system symbol tables.

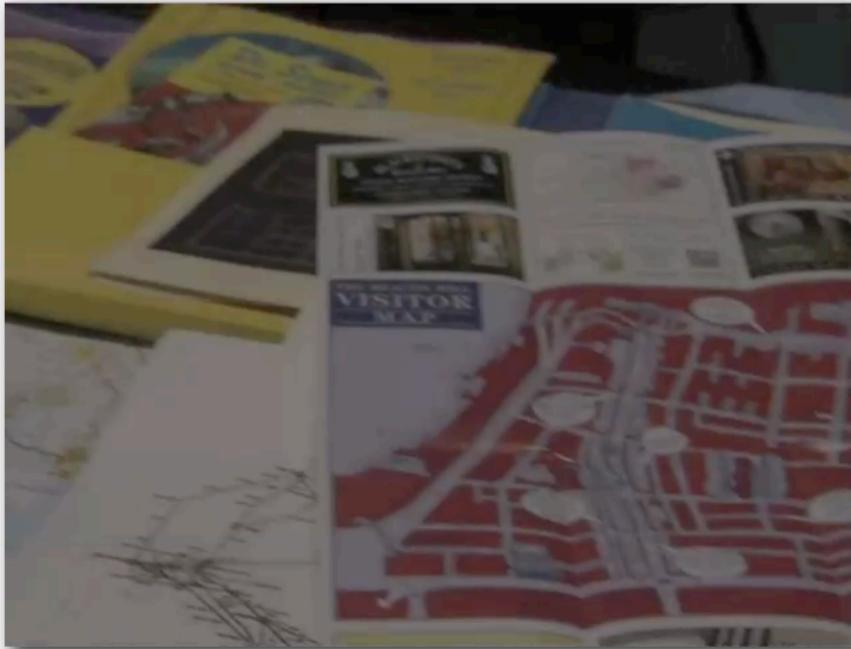
- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: `map`, `multimap`, `multiset`.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.
- Emacs: conservative stack scanning.

B-tree variants. B+ tree, B*tree, B# tree, ...

B-trees (and variants) are widely used for file systems and databases.

- Windows: NTFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

Red-black BSTs in the wild



*Common sense. Sixth sense.
Together they're the
FBI's newest team.*

Red-black BSTs 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?

CS2010: ALGORITHMS AND DATA STRUCTURES

Lectures 15-16: Hashtables

Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Symbol tables

Key-value pair abstraction.

- **Insert** a value with specified key.
- Given a key, **search** for the corresponding value.

Ex. DNS lookup.

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

domain name	IP address
www.cs.princeton.edu	128.112.136.11
www.princeton.edu	128.112.128.15
www.yale.edu	130.132.143.21
www.harvard.edu	128.103.060.55
www.simpsons.com	209.052.165.60

↑
key

↑
value

Symbol table applications

application	purpose of search	key	value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
financial account	process transactions	account number	transaction details
web search	find relevant web pages	keyword	list of page names
compiler	find properties of variables	variable name	type and value
routing table	route Internet packets	destination	best route
DNS	find IP address	domain name	IP address
reverse DNS	find domain name	IP address	domain name
genomics	find markers	DNA string	known positions
file system	find file on disk	filename	location on disk

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```
public class ST<Key, Value>
```

```
    ST()
```

create an empty symbol table

```
    void put(Key key, Value val)
```

put key-value pair into the table $\leftarrow a[key] = val$

```
    Value get(Key key)
```

value paired with key $\leftarrow a[key]$

```
    boolean contains(Key key)
```

is there a value paired with key?

```
    void delete(Key key)
```

remove key (and its value) from table

```
    boolean isEmpty()
```

is the table empty?

```
    int size()
```

number of key-value pairs in the table

```
    Iterable<Key> keys()
```

all the keys in the table

Conventions

- Values are not null. ← Java allows null value
- Method get() returns null if key not present.
- Method put() overwrites old value with new value.

Intended consequences.

- Easy to implement contains().

```
public boolean contains(Key key)
{   return get(key) != null; }
```

- Can implement lazy version of delete().

```
public void delete(Key key)
{   put(key, null); }
```

Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are Comparable, use compareTo().
- Assume keys are any generic type, use equals() to test equality.
- Assume keys are any generic type, use equals() to test equality; use hashCode() to scramble key.

specify Comparable in API.

built-in to Java
(stay tuned)

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

Equality test

All Java classes inherit a method `equals()`.

Java requirements. For any references x , y and z :

- Reflexive: $x.equals(x)$ is true.
 - Symmetric: $x.equals(y)$ iff $y.equals(x)$.
 - Transitive: if $x.equals(y)$ and $y.equals(z)$, then $x.equals(z)$.
 - Non-null: $x.equals(null)$ is false.
- 

Default implementation. $(x == y)$

do x and y refer to
the same object?



Customized implementations. `Integer`, `Double`, `String`, `java.io.File`, ...

User-defined implementations. Some care needed.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.1 SYMBOL TABLES

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

Examples of ordered symbol table API

	keys	values
min()	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13	Houston
get(09:00:13)	09:00:59	Chicago
	09:01:10	Houston
floor(09:05:00)	09:03:13	Chicago
	09:10:11	Seattle
select(7)	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
keys(09:15:00, 09:25:00)	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
ceiling(09:30:00)	09:35:21	Chicago
	09:36:14	Seattle
max()	09:37:44	Phoenix

size(09:15:00, 09:25:00) is 5

rank(09:10:25) is 7

Ordered symbol table API

```
public class ST<Key extends Comparable<Key>, Value>
```

...

Key min()

smallest key

Key max()

largest key

Key floor(Key key)

largest key less than or equal to key

Key ceiling(Key key)

smallest key greater than or equal to key

int rank(Key key)

number of keys less than key

Key select(int k)

key of rank k

void deleteMin()

delete smallest key

void deleteMax()

delete largest key

int size(Key lo, Key hi)

number of keys between lo and hi

Iterable<Key> keys()

all keys, in sorted order

Iterable<Key> keys(Key lo, Key hi)

keys between lo and hi, in sorted order

Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>

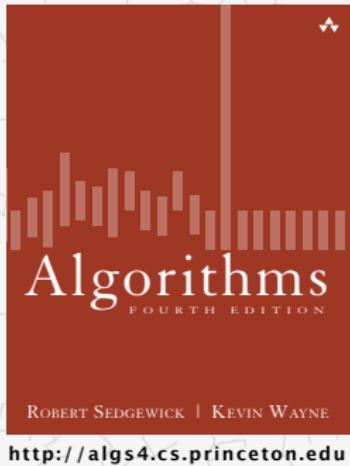
Q. Can we do better?

A. Yes, but with different access to the data.

Will NOT support ordered operations.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ROBERT SEDGEWICK | KEVIN WAYNE

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

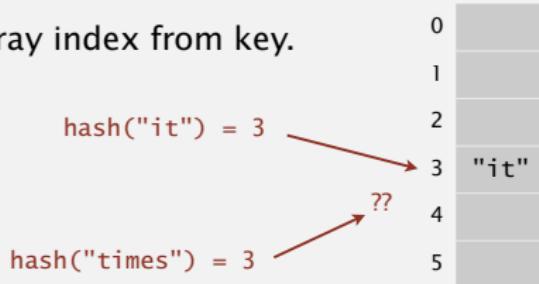
3.4 HASH TABLES

- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ *context*

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.
- Space and time limitations: hashing (the real world).



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

3.4 HASH TABLES

- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ *context*

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

- Efficiently computable.
- Each table index equally likely for each key.



thoroughly researched problem,
still problematic in practical applications



Ex 1. Phone numbers.

- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.

- Bad: first three digits. ← 573 = California, 574 = Alaska
(assigned in chronological order within geographic region)
- Better: last three digits.

Practical challenge. Need different approach for each key type.

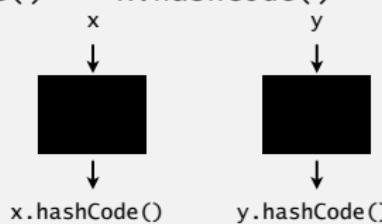
Java's hash code conventions

All Java classes inherit a method `hashCode()`, which returns a 32-bit int.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.

Requirement `x.hashCode() == x.hashCode()`



Default implementation. Memory address of `x`.

Legal (but poor) implementation. Always return 17.

Customized implementations. Integer, Double, String, File, URL, Date, ...

User-defined types. Users are on their own.

Implementing hash code: integers, booleans, and doubles

Java library implementations

```
public final class Integer
{
    private final int value;
    ...
    public int hashCode()
    {   return value;   }
}
```

```
public final class Boolean
{
    private final boolean value;
    ...
    public int hashCode()
    {
        if (value) return 1231;
        else      return 1237;
    }
}
```

```
public final class Double
{
    private final double value;
    ...
    public int hashCode()
    {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Warning: -0.0 and +0.0 have different hash codes

Implementing hash code: strings

Java library implementation

```
public final class String
{
    private final char[] s;
    ...

    public int hashCode()
    {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash);
        return hash;
    }
}
```

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

ith character of s

- Horner's method to hash string of length L : L multiplies/adds.
- Equivalent to $h = s[0] \cdot 31^{L-1} + \dots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0$.

Ex. `String s = "call";
int code = s.hashCode();` \leftarrow $3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$
 $= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$
(Horner's method)

Implementing hash code: strings

Performance optimization.

- Cache the hash value in an instance variable.
- Return cached value.

```
public final class String
{
    private int hash = 0;           ← cache of hash code
    private final char[] s;
    ...

    public int hashCode()
    {
        int h = hash;             ← return cached value
        if (h != 0) return h;
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * h);
        hash = h;                 ← store cache of hash code
        return h;
    }
}
```

Q. What if hashCode() of string is 0?

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
    private final String who;
    private final Date when;
    private final double amount;

    public Transaction(String who, Date when, double amount)
    { /* as before */ }

    ...

    public boolean equals(Object y)
    { /* as before */ }

    public int hashCode()
    {
        int hash = 17;                                ← nonzero constant
        hash = 31*hash + who.hashCode();                ← for reference types,
                                                       use hashCode()
        hash = 31*hash + when.hashCode();              ← for primitive types,
                                                       use hashCode()
                                                       of wrapper type
        hash = 31*hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

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 wrapper type `hashCode()`.
- If field is `null`, return 0.
- If field is a reference type, use `hashCode()`. ← applies rule recursively
- If field is an array, apply to each entry. ← or use `Arrays.deepHashCode()`

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Keys are bitstring; "universal" hash functions exist.

Basic rule. Need to use the whole key to compute hash code;
consult an expert for state-of-the-art hash codes.

Modular hashing

Hash code. An int between -2^{31} and $2^{31} - 1$.

Hash function. An int between 0 and $M - 1$ (for use as array index).

↑
typically a prime or power of 2

```
private int hash(Key key)
{   return key.hashCode() % M; }
```

bug



```
private int hash(Key key)
{   return Math.abs(key.hashCode()) % M; }
```

1-in-a-billion bug

↑
hashCode() of "polygenelubricants" is -2^{31}

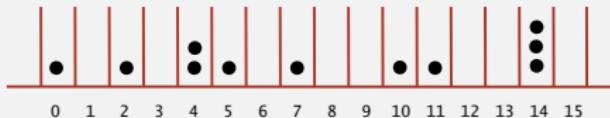
```
private int hash(Key key)
{   return (key.hashCode() & 0xffffffff) % M; }
```

correct

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sim \sqrt{\pi M / 2}$ tosses.

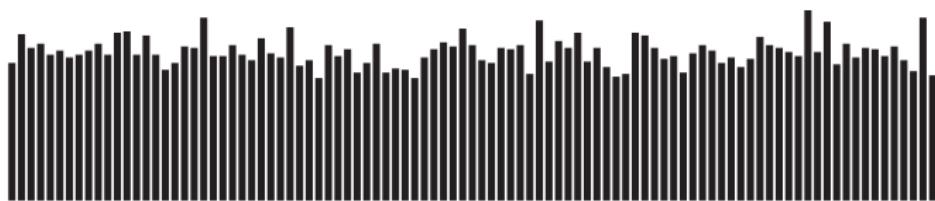
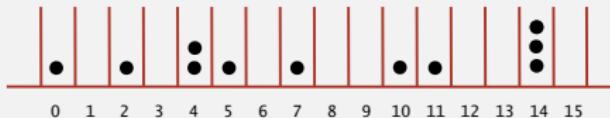
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Uniform hashing assumption

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and $M - 1$.

Bins and balls. Throw balls uniformly at random into M bins.



Hash value frequencies for words in Tale of Two Cities ($M = 97$)

Java's String data uniformly distribute the keys of Tale of Two Cities



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

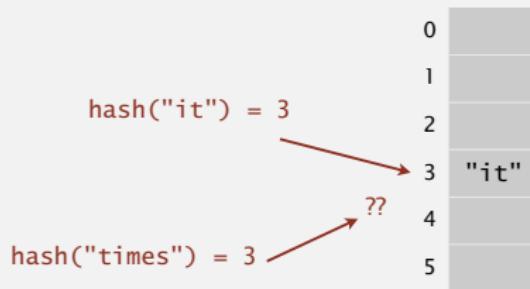
3.4 HASH TABLES

- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ *context*

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem \Rightarrow can't avoid collisions unless you have a ridiculous (quadratic) amount of memory.
- Coupon collector + load balancing \Rightarrow collisions are evenly distributed.



Challenge. Deal with collisions efficiently.

Separate-chaining symbol table

Use an array of $M < N$ linked lists. [H. P. Luhn, IBM 1953]

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain (if not already there).
- Search: need to search only i^{th} chain.

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

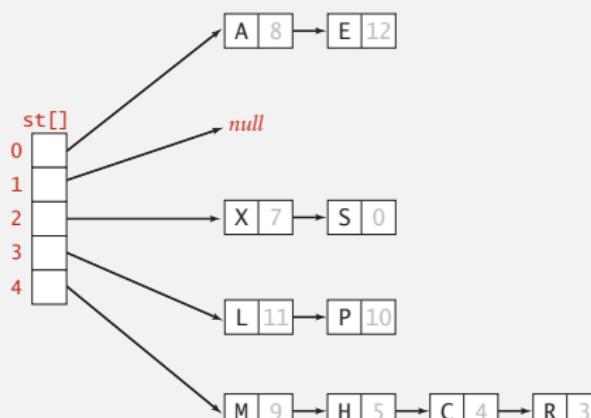
A 0 8

M 4 9

P 3 10

L 3 11

E 0 12



Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;                      // number of chains
    private Node[] st = new Node[M];          // array of chains

    private static class Node
    {
        private Object key;      ← no generic array creation
        private Object val;     ← (declare key and value of type Object)
        private Node next;
        ...
    }

    private int hash(Key key)
    {   return (key.hashCode() & 0xffffffff) % M;   }

    public Value get(Key key) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }
}
```

array doubling and
halving code omitted

Separate-chaining symbol table: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;                      // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

    private int hash(Key key)
    {   return (key.hashCode() & 0xffffffff) % M;   }

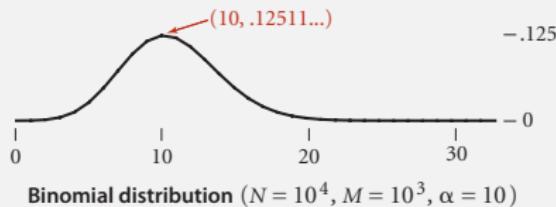
    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }

}
```

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. 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 probes for search/insert is proportional to N/M .

- M too large \Rightarrow too many empty chains.
- M too small \Rightarrow chains too long.
- Typical choice: $M \sim N/4 \Rightarrow$ constant-time ops.

equals() and hashCode()

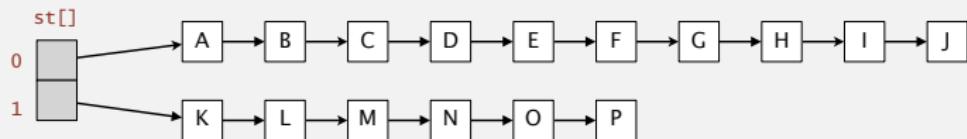
\uparrow
M times faster than
sequential search

Resizing in a separate-chaining hash table

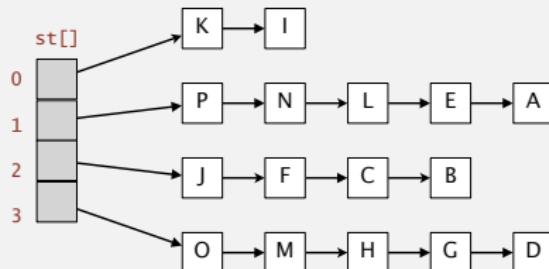
Goal. Average length of list $N / M = \text{constant}$.

- Double size of array M when $N / M \geq 8$.
- Halve size of array M when $N / M \leq 2$.
- Need to rehash all keys when resizing. ← x.hashCode() does not change
but hash(x) can change

before resizing



after resizing

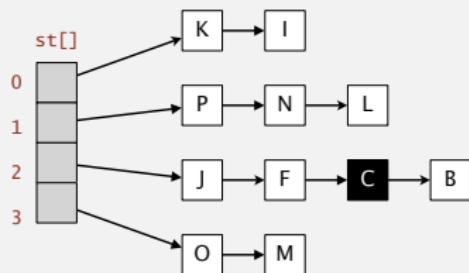


Deletion in a separate-chaining hash table

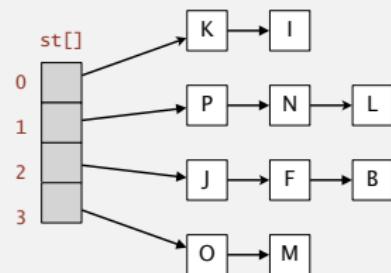
Q. How to delete a key (and its associated value)?

A. Easy: need only consider chain containing key.

before deleting C



after deleting C



Symbol table implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>
separate chaining	N	N	N	$3\text{-}5^*$	$3\text{-}5^*$	$3\text{-}5^*$		<code>equals()</code> <code>hashCode()</code>

* under uniform hashing assumption



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

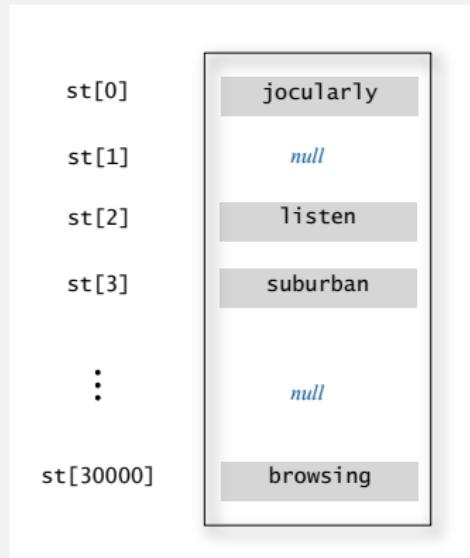
3.4 HASH TABLES

- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ *context*

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rochester-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.



linear probing ($M = 30001, N = 15000$)

Linear-probing hash table demo

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

linear-probing hash table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]																

M = 16



Linear-probing hash table demo

Hash. Map key to integer i between 0 and $M-1$.

Search. Search table index i ; if occupied but no match, try $i+1$, $i+2$, etc.

search K

hash(K) = 5

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E			R	X

$M = 16$

K

search miss
(return null)

Linear-probing hash table summary

Hash. Map key to integer i between 0 and $M-1$.

Insert. Put at table index i if free; if not try $i+1, i+2$, etc.

Search. Search table index i ; if occupied but no match, try $i+1, i+2$, etc.

Note. Array size M **must be** greater than number of key-value pairs N .

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
st[]	P	M			A	C	S	H	L		E				R	X

$M = 16$

Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];

    private int hash(Key key) { /* as before */ }

    private void put(Key key, Value val) { /* next slide */ }

    public Value get(Key key)
    {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M)
            if (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
```

array doubling and
halving code omitted

Linear-probing symbol table: Java implementation

```
public class LinearProbingHashST<Key, Value>
{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];

    private int hash(Key key) { /* as before */ }

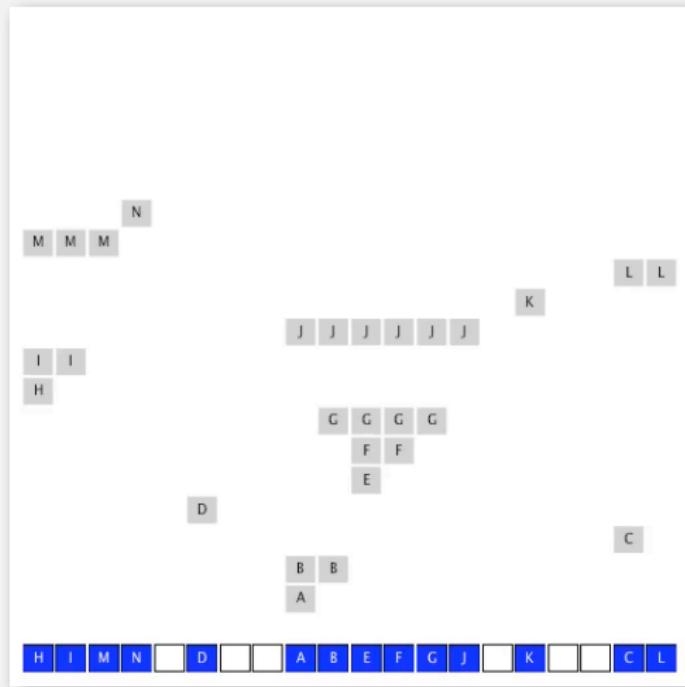
    private Value get(Key key) { /* previous slide */ }

    public void put(Key key, Value val)
    {
        int i;
        for (i = hash(key); keys[i] != null; i = (i+1) % M)
            if (keys[i].equals(key))
                break;
        keys[i] = key;
        vals[i] = val;
    }
}
```

Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.



Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.

Each desires a random space i : if space i is taken, try $i + 1, i + 2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M/2$ cars, mean displacement is $\sim 3/2$.

Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$$

search hit

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$$

search miss / insert

Pf.

*[My first analysis of an algorithm, originally
done during summer 1961 in Madison.]*

8354

NOTES ON "OPEN" ADDRESSING.
D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- M too large \Rightarrow too many empty array entries.
- M too small \Rightarrow search time blows up.
- Typical choice: $\alpha = N/M \sim 1/2$.

probes for search hit is about 3/2
probes for search miss is about 5/2

Resizing in a linear-probing hash table

Goal. Average length of list $N / M \leq \frac{1}{2}$.

- Double size of array M when $N / M \geq \frac{1}{2}$.
- Halve size of array M when $N / M \leq \frac{1}{8}$.
- Need to rehash all keys when resizing.

before resizing

	0	1	2	3	4	5	6	7
keys[]		E	S			R	A	
vals[]		1	0			3	2	

after resizing

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]				A		S			E				R			
vals[]					2	0				1				3		

Deletion in a linear-probing hash table

Q. How to delete a key (and its associated value)?

A. Requires some care: can't just delete array entries.

before deleting S

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C	S	H	L		E				R	X
vals[]	10	9			8	4	0	5	11		12				3	7

doesn't work, e.g., if $\text{hash}(H) = 4$

after deleting S ?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
keys[]	P	M			A	C		H	L		E				R	X
vals[]	10	9			8	4		5	11		12				3	7



ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$\frac{1}{2}N$	N	$\frac{1}{2}N$		<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	<code>compareTo()</code>
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	\sqrt{N}	✓	<code>compareTo()</code>
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$	$1.0 \lg N$	$1.0 \lg N$	✓	<code>compareTo()</code>
separate chaining	N	N	N	3-5 *	3-5 *	3-5 *		<code>equals()</code> <code>hashCode()</code>
linear probing	N	N	N	3-5 *	3-5 *	3-5 *		<code>equals()</code> <code>hashCode()</code>

* under uniform hashing assumption



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

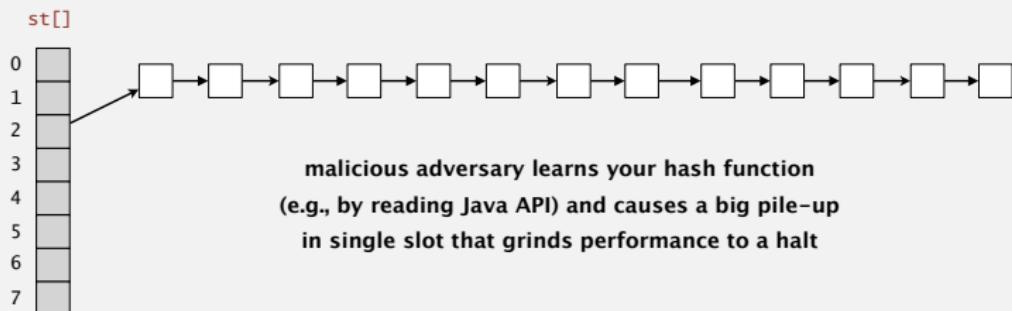
<http://algs4.cs.princeton.edu>

3.4 HASH TABLES

- ▶ *hash functions*
- ▶ *separate chaining*
- ▶ *linear probing*
- ▶ **context**

War story: algorithmic complexity attacks

- Q. Is the uniform hashing assumption important in practice?
- A. Obvious situations: aircraft control, nuclear reactor, pacemaker.
- A. Surprising situations: denial-of-service attacks.



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.

War story: algorithmic complexity attacks

A Java bug report.

Jan Lieskovsky 2011-11-01 10:13:47 EDT

Description

Julian Wälde and Alexander Klink reported that the String.hashCode() hash function is not sufficiently collision resistant. hashCode() value is used in the implementations of HashMap and Hashtable classes:

<http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html>
<http://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.html>

A specially-crafted set of keys could trigger hash function collisions, which can degrade performance of HashMap or Hashtable by changing hash table operations complexity from an expected/average $O(1)$ to the worst case $O(n)$. Reporters were able to find colliding strings efficiently using equivalent substrings and meet in the middle techniques.

This problem can be used to start a denial of service attack against Java applications that use untrusted inputs as HashMap or Hashtable keys. An example of such application is web application server (such as tomcat, see [bug #750524](#)) that may fill hash tables with data from HTTP request (such as GET or POST parameters). A remote attack could use that to make JVM use excessive amount of CPU time by sending a POST request with large amount of parameters which hash to the same value.

This problem is similar to the issue that was previously reported for and fixed in e.g. perl:

http://www.cs.rice.edu/~scrosby/hash/CrosbyWallach_UsenixSec2003.pdf

Algorithmic complexity attack on Java

Goal. Find family of strings with the same hash code.

Solution. The base-31 hash code is part of Java's string API.

key	hashCode()
"Aa"	2112
"BB"	2112

key	hashCode()
"AaAaAaAa"	-540425984
"AaAaAaBb"	-540425984
"AaAaBBAa"	-540425984
"AaAaBBBB"	-540425984
"AaBBAaAa"	-540425984
"AaBBAaBB"	-540425984
"AaBBBBAa"	-540425984
"AaBBBBBB"	-540425984

key	hashCode()
"BBAaAaAa"	-540425984
"BBAaAaBB"	-540425984
"BBAaBBAa"	-540425984
"BBAaBBBB"	-540425984
"BBBBAaAa"	-540425984
"BBBBAaBB"	-540425984
"BBBBBBAA"	-540425984
"BBBBBBBB"	-540425984

2^N strings of length $2N$ that hash to same value!

Diversion: one-way hash functions

One-way hash function. "Hard" to find a key that will hash to a desired value (or two keys that hash to same value).

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160,

known to be insecure

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

/* prints bytes as hex string */
```

Applications. Digital fingerprint, message digest, storing passwords.

Caveat. Too expensive for use in ST implementations.

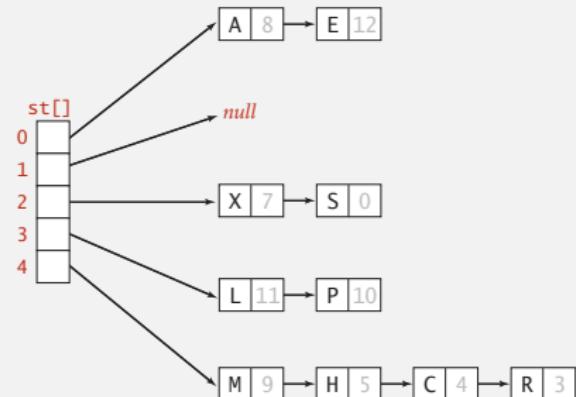
Separate chaining vs. linear probing

Separate chaining.

- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.



keys[]	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
vals[]	P	M			A	C	S	H	L		E			R	X	
	10	9			8	4	0	5	11		12			3	7	

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. [separate-chaining variant]

- Hash to two positions, insert key in shorter of the two chains.
- Reduces expected length of the longest chain to $\log \log N$.

Double hashing. [linear-probing variant]

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.
- More difficult to implement delete.

Cuckoo hashing. [linear-probing variant]

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst-case time for search.



Hash tables vs. balanced search trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus $\log N$ compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

CS2010: ALGORITHMS AND DATA STRUCTURES

Lectures 17: Union-Find

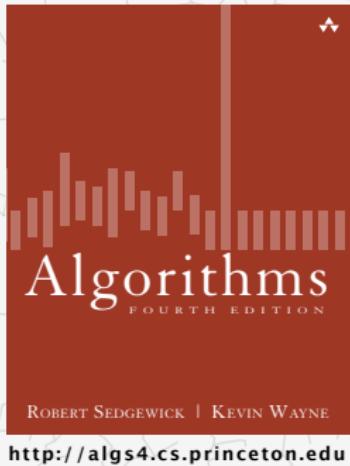
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

- ▶ *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 not.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Dynamic connectivity problem

Given a set of N objects, support two operations:

- Connect two objects.
- Is there a path connecting the two objects?

connect 4 and 3

connect 3 and 8

connect 6 and 5

connect 9 and 4

connect 2 and 1

are 0 and 7 connected? ✗

are 8 and 9 connected? ✓

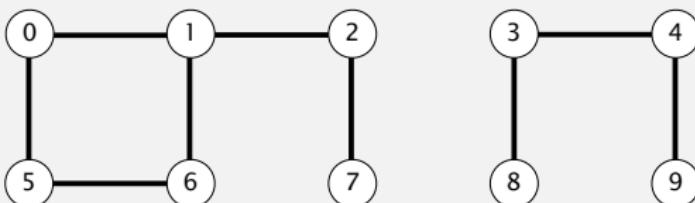
connect 5 and 0

connect 7 and 2

connect 6 and 1

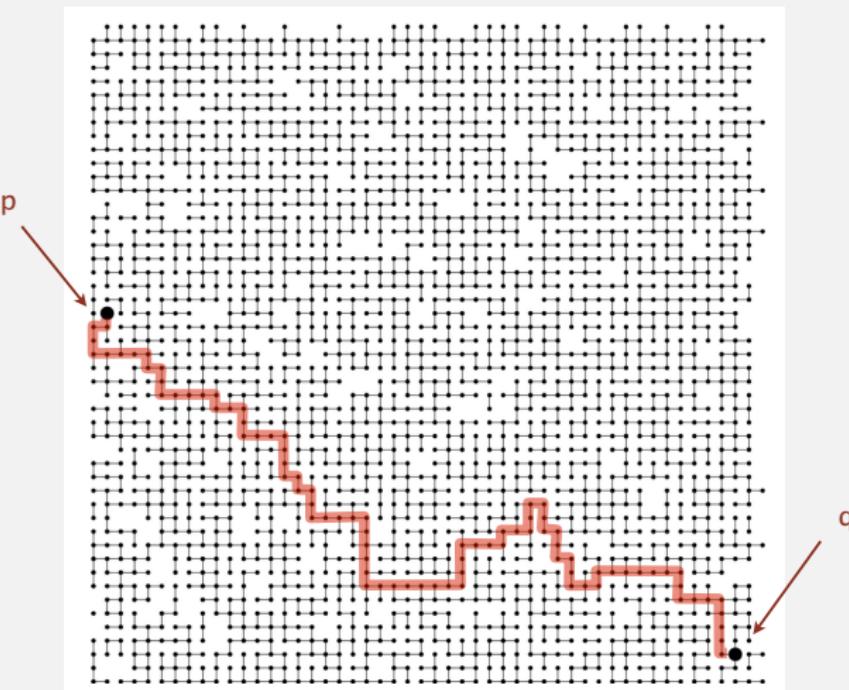
connect 1 and 0

are 0 and 7 connected? ✓



A larger connectivity example

Q. Is there a path connecting p and q ?



A. Yes.

Modeling the objects

Applications involve manipulating objects of all types.

- Pixels in a digital photo.
- Computers in a network.
- Friends in a social network.
- Transistors in a computer chip.
- Elements in a mathematical set.
- Variable names in a Fortran program.
- 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.



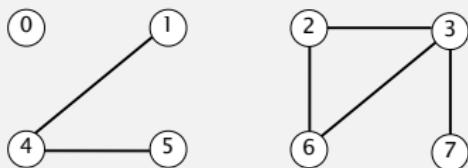
can use symbol table to translate from site
names to integers: stay tuned (Chapter 3)

Modeling the connections

We assume "is connected to" is an equivalence relation:

- Reflexive: p is connected to p .
- Symmetric: if p is connected to q , then q is connected to p .
- Transitive: if p is connected to q and q is connected to r ,
then p is connected to r .

Connected component. Maximal **set** of objects that are mutually connected.



$\{ 0 \} \{ 1 4 5 \} \{ 2 3 6 7 \}$

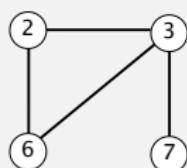
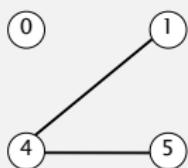
3 connected components

Implementing the operations

Find. In which component is object p ?

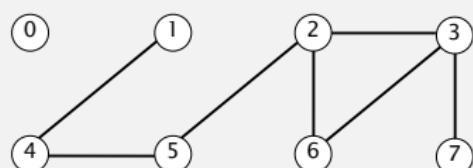
Connected. Are objects p and q in the same component?

Union. Replace components containing objects p and q with their union.



union(2, 5)

A large red arrow points from the initial state of three components to the merged state where components 2 and 5 have been joined.



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

Three red arrows point from the labels below to the three components: {0}, {1, 4, 5}, and {2, 3, 6, 7}. The label "3 connected components" is centered below the arrows.

3 connected components

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

Two red arrows point from the labels below to the two components: {0} and {1, 2, 3, 4, 5, 6, 7}. The label "2 connected components" is centered below the arrows.

2 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.
- Union and find operations may be intermixed.

```
public class UF
```

```
UF(int N)
```

*initialize union-find data structure
with N singleton objects (0 to $N - 1$)*

```
void union(int p, int q)
```

add connection between p and q

```
int find(int p)
```

component identifier for p (0 to $N - 1$)

```
boolean connected(int p, int q)
```

are p and q in the same component?

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

1-line implementation of connected()

Dynamic-connectivity client

- Read in number of objects N from standard input.
- Repeat:
 - read in pair of integers from standard input
 - if they are not yet connected, connect them and print out pair

```
public static void main(String[] args)
{
    int N = StdIn.readInt();
    UF uf = new UF(N);
    while (!StdIn.isEmpty())
    {
        int p = StdIn.readInt();
        int q = StdIn.readInt();
        if (!uf.connected(p, q))
        {
            uf.union(p, q);
            StdOut.println(p + " " + q);
        }
    }
}
```

```
% more tinyUF.txt
```

```
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

already connected



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

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

Quick-find [eager approach]

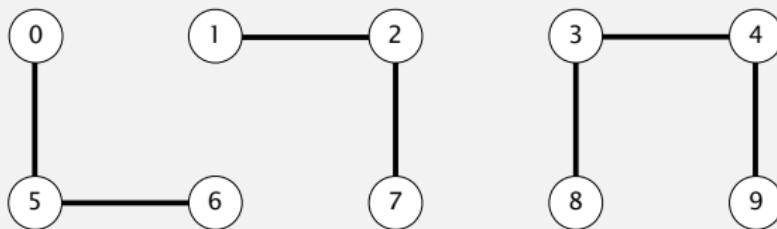
Data structure.

- Integer array $\text{id}[]$ of length N .
- Interpretation: $\text{id}[p]$ is the id of the component containing p .

if and only if

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	0	1	1	8	8	0	0	1	8	8

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected



Quick-find [eager approach]

Data structure.

- Integer array $\text{id}[]$ of length N .
- Interpretation: $\text{id}[p]$ is the id of the component containing p .

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	0	1	1	8	8	0	0	1	8	8

Find. What is the id of p ?

$\text{id}[6] = 0; \text{id}[1] = 1$
6 and 1 are not connected

Connected. Do p and q have the same id?

Union. To merge components containing p and q , change all entries whose id equals $\text{id}[p]$ to $\text{id}[q]$.

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	1	1	1	8	8	1	1	1	8	8

problem: many values can change

after union of 6 and 1

Quick-find demo



0

1

2

3

4

5

6

7

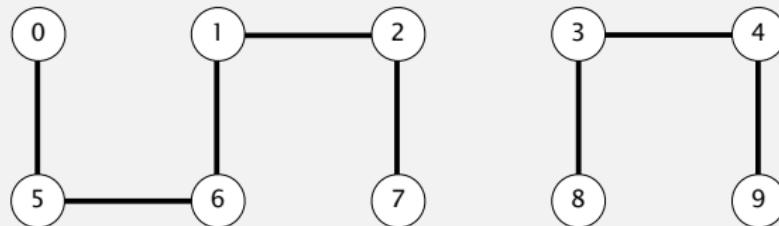
8

9

0 1 2 3 4 5 6 7 8 9

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

Quick-find demo



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

Quick-find: Java implementation

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

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

    public int find(int p)
    {   return id[p];   }

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

set id of each object to itself
(N array accesses)

return the id of p
(1 array access)

change all entries with $\text{id}[p]$ to $\text{id}[q]$
(at most $2N + 2$ array accesses)

Quick-find is too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find	connected
quick-find	N	N	1	1

order of growth of number of array accesses

Union is too expensive. It takes N^2 array accesses to process a sequence of N union operations on N objects.

quadratic

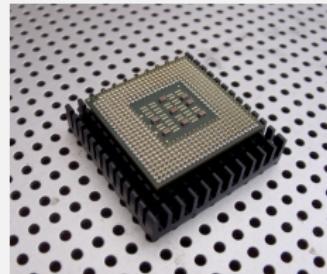


Quadratic algorithms do not scale

Rough standard (for now).

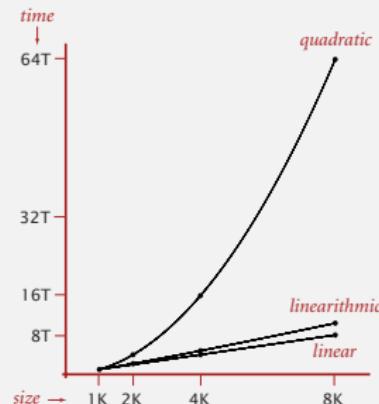
- 10^9 operations per second.
- 10^9 words of main memory.
- Touch all words in approximately 1 second.

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!



Quadratic algorithms don't scale with technology.

- New computer may be 10x as fast.
- But, has 10x as much memory ⇒ want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

1.5 UNION-FIND

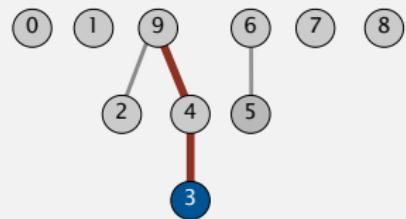
- ▶ *dynamic connectivity*
- ▶ *quick find*
- ▶ *quick union*
- ▶ *improvements*
- ▶ *applications*

Quick-union [lazy approach]

Data structure.

- Integer array $\text{id}[]$ of length N .
- Interpretation: $\text{id}[i]$ is parent of i .
keep going until it doesn't change
(algorithm ensures no cycles)
- Root of i is $\text{id}[\text{id}[\text{id}[\dots\text{id}[i]\dots]]]$.

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	0	1	9	4	9	6	6	7	8	9



parent of 3 is 4

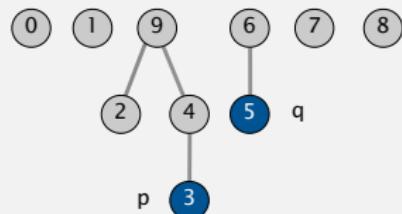
root of 3 is 9

Quick-union [lazy approach]

Data structure.

- Integer array $\text{id}[]$ of length N .
- Interpretation: $\text{id}[i]$ is parent of i .
- Root of i is $\text{id}[\text{id}[\text{id}[\dots\text{id}[i]\dots]]]$.

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	0	1	9	4	9	6	6	7	8	9



Find. What is the root of p ?

Connected. Do p and q have the same root?

root of 3 is 9

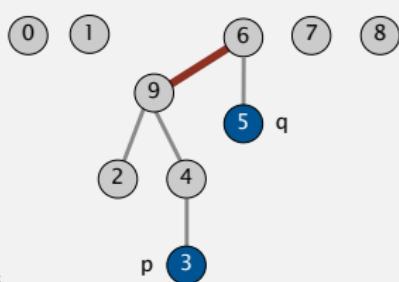
root of 5 is 6

3 and 5 are not connected

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

	0	1	2	3	4	5	6	7	8	9
$\text{id}[]$	0	1	9	4	9	6	6	7	8	6

only one value changes



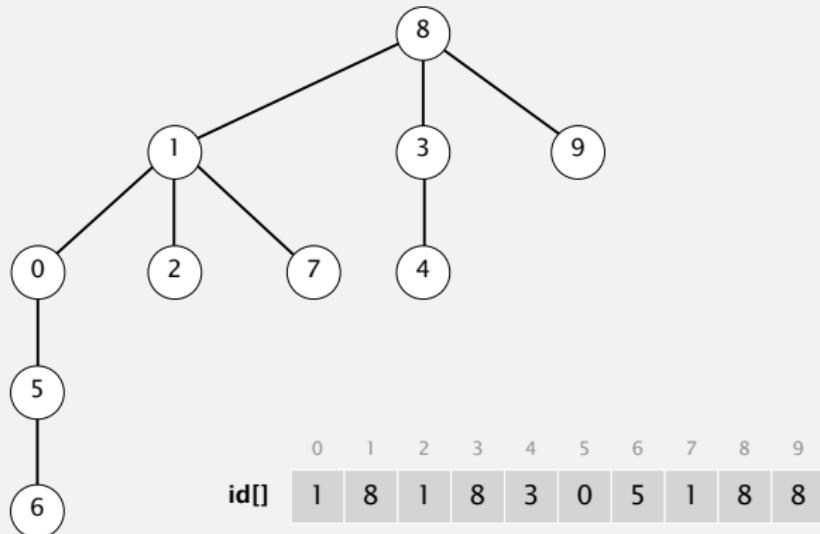
Quick-union demo



0 1 2 3 4 5 6 7 8 9

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

Quick-union demo



Quick-union: Java implementation

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

    public QuickUnionUF(int N)
    {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i; ← set id of each object to itself
    }

    public int find(int i)
    {
        while (i != id[i]) i = id[i]; ← chase parent pointers until reach root
        return i;
    }

    public void union(int p, int q)
    {
        int i = find(p);
        int j = find(q);
        id[i] = j; ← change root of p to point to root of q
    }
}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

Quick-union is also too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N [†]	N	N

← worst case

† includes cost of finding roots

Quick-find defect.

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

Quick-union defect.

- Trees can get tall.
- Find/connected too expensive (could be N array accesses).



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

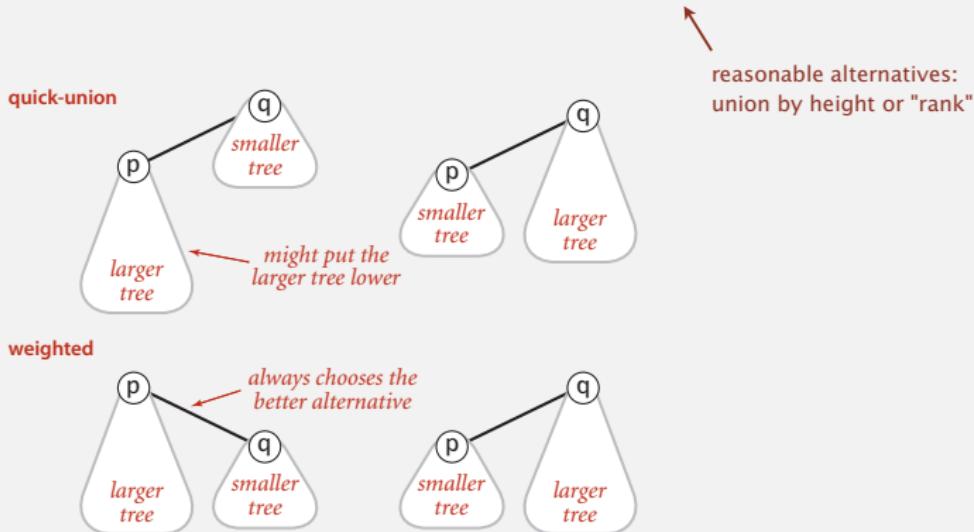
1.5 UNION-FIND

- ▶ *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 tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.



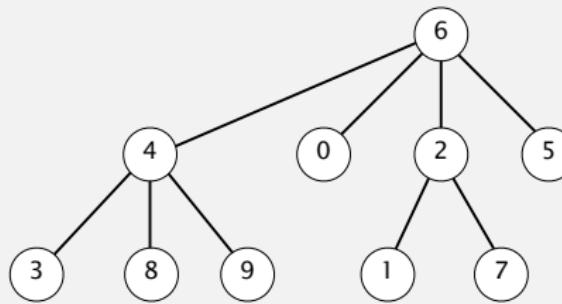
Weighted quick-union demo



0 1 2 3 4 5 6 7 8 9

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

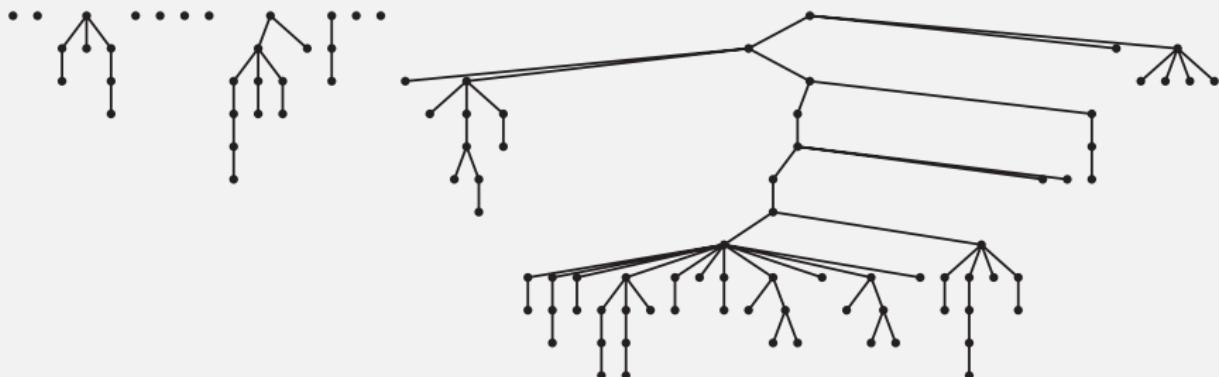
Weighted quick-union demo



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

Quick-union and weighted quick-union example

quick-union



average distance to root: 5.11

weighted



average distance to root: 1.52

Quick-union and weighted quick-union (100 sites, 88 union() operations)

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/connected. Identical to quick-union.

Union. Modify quick-union to:

- Link root of smaller tree to root of larger tree.
- Update the $sz[]$ array.

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

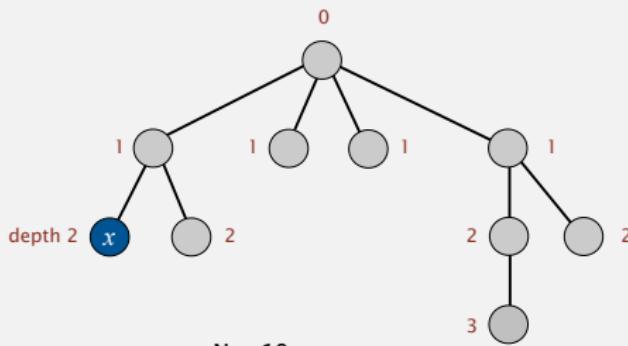
Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p .
- Union: takes constant time, given roots.

\lg = base-2 logarithm

Proposition. Depth of any node x is at most $\lg N$.



Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p .
- Union: takes constant time, given roots.

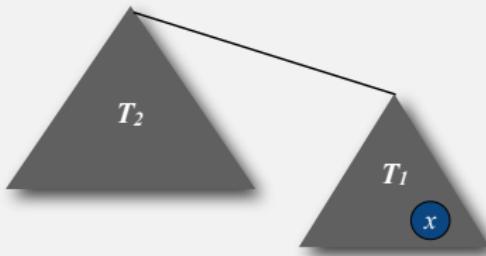
\lg = base-2 logarithm

Proposition. Depth of any node x is at most $\lg N$.

Pf. What causes the depth of object x to increase?

Increases by 1 when 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. Why?



Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p .
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most $\lg N$.

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N^{\dagger}	N	N
weighted QU	N	$\lg N^{\dagger}$	$\lg N$	$\lg N$

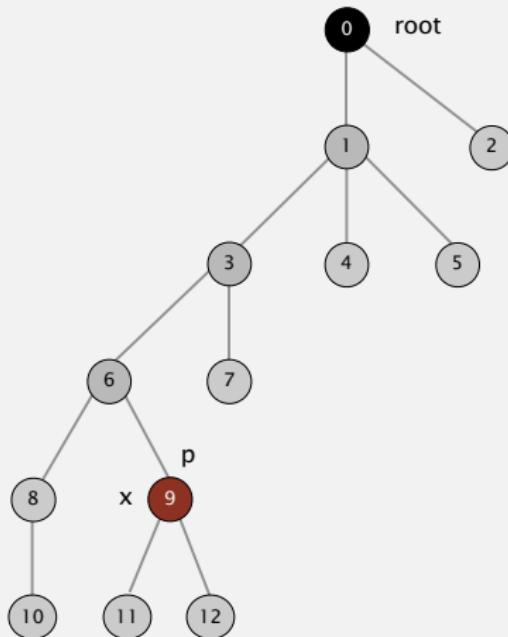
\dagger includes cost of finding roots

Time/space tradeoff: Weighted QU uses $O(N)$ more space to improve running time.

- Q. Stop at guaranteed acceptable performance?
A. No, easy to improve further.

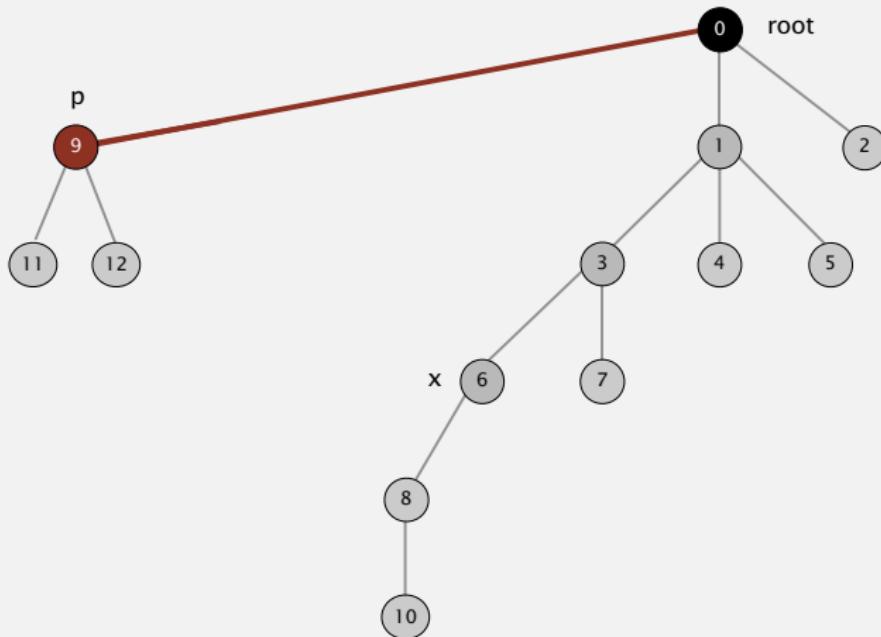
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



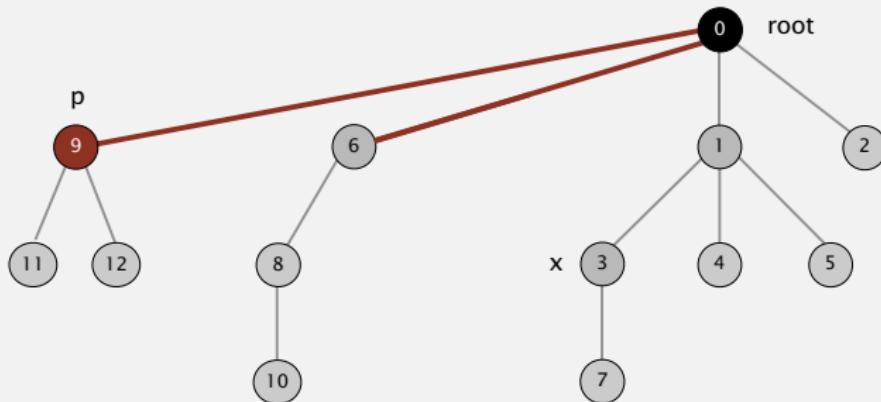
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



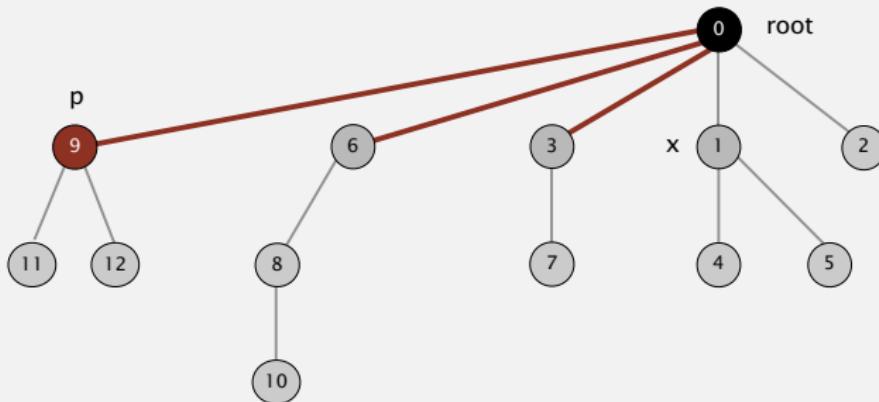
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



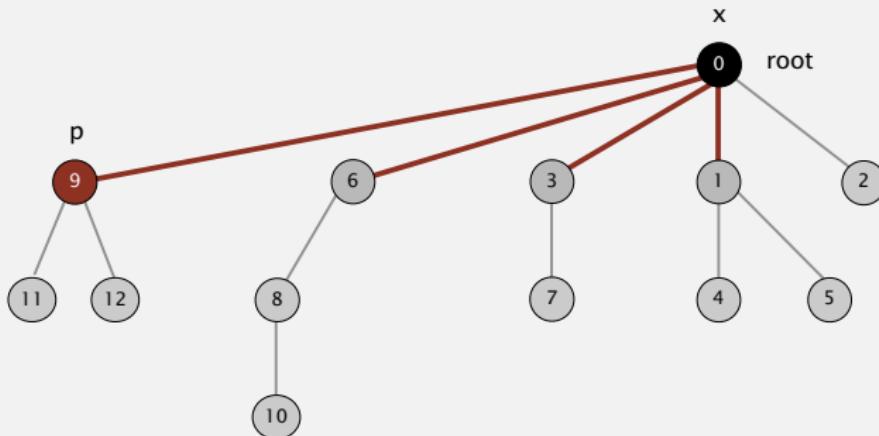
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



Improvement 2: path compression

Quick union with path compression. Just after computing the root of p , set the $\text{id}[]$ of each examined node to point to that root.



Bottom line. Now, `find()` has the side effect of compressing the tree.

Path compression: Java implementation

Two-pass implementation: add second loop to `find()` to set the `id[]` of each examined node to the root.

Simpler one-pass variant (path halving): Make every other node in path point to its grandparent.

```
public int find(int i)
{
    while (i != id[i])
    {
        id[i] = id[id[i]]; ← only one extra line of code !
        i = id[i];
    }
    return i;
}
```

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

Weighted quick-union with path compression: amortized analysis

Proposition. [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of M union–find ops on N objects makes $\leq c(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M \alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

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

iterated \lg function

Linear-time algorithm for M union–find ops on N objects?

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

Amazing fact. [Fredman-Saks] No linear-time algorithm exists.

in "cell-probe" model of computation

Summary

Key point. Weighted quick union (and/or path compression) 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	$N + M \lg^* N$

order of growth for M union-find operations on a set of N objects

Ex. [10⁹ unions and finds with 10⁹ objects]

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



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

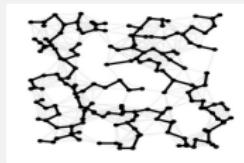
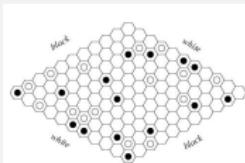
<http://algs4.cs.princeton.edu>

1.5 UNION-FIND

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

Union-find applications

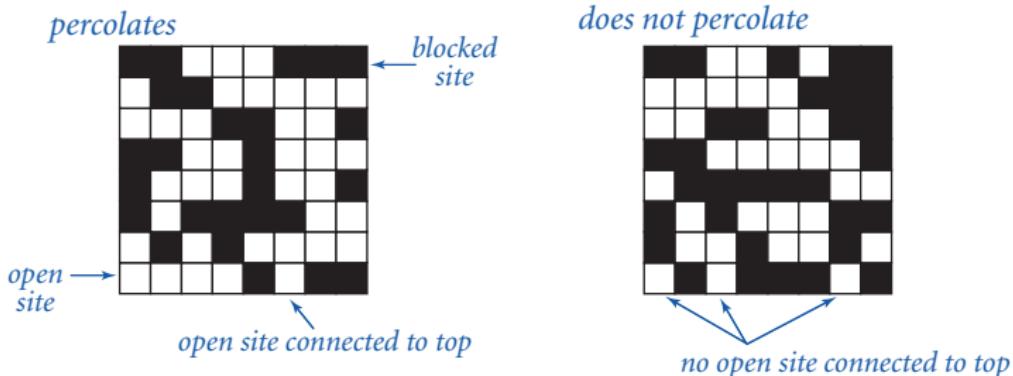
- Percolation.
- Games (Go, Hex).
- ✓ Dynamic connectivity.
- Least common ancestor.
- Equivalence of finite state automata.
- Hoshen-Kopelman algorithm in physics.
- Hinley-Milner polymorphic type inference.
- Kruskal's minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.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



Percolation

An abstract model for many physical systems:

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



Percolation

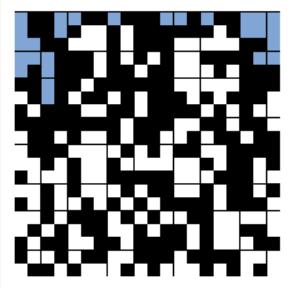
An abstract model for many physical systems:

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

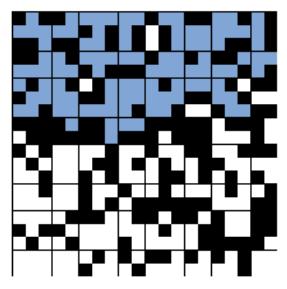
model	system	vacant site	occupied site	percolates
electricity	material	conductor	insulated	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Likelihood of percolation

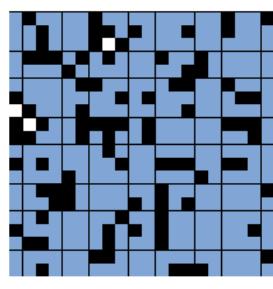
Depends on grid size N and site vacancy probability p .



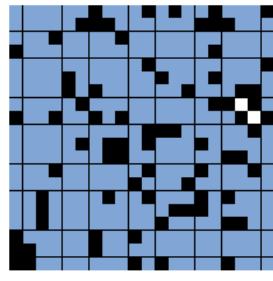
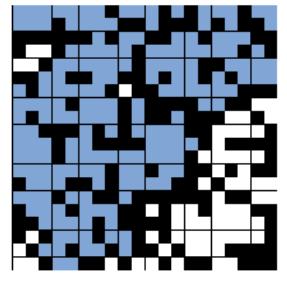
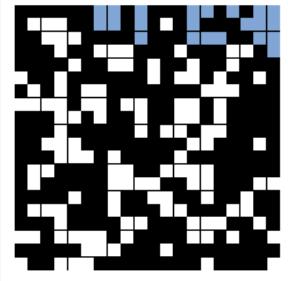
p low (0.4)
does not percolate



p medium (0.6)
percolates?



p high (0.8)
percolates

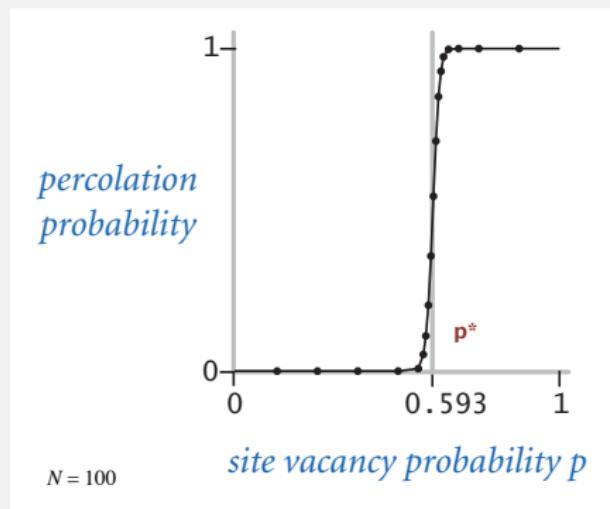


Percolation phase transition

When N is large, theory guarantees a sharp threshold p^* .

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

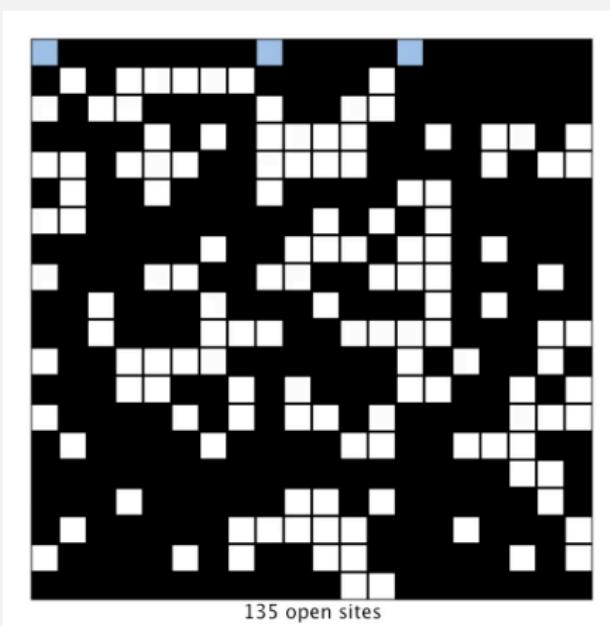
Q. What is the value of p^* ?



Monte Carlo simulation

- Initialize all sites in an N -by- N grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates p^* .

$N = 20$



full open site
(connected to top)

empty open site
(not connected to top)

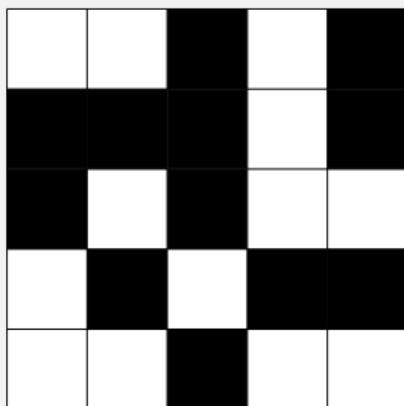
blocked site

Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an N -by- N system percolates?

A. Model as a **dynamic connectivity** problem and use **union-find**.

$N = 5$



open site

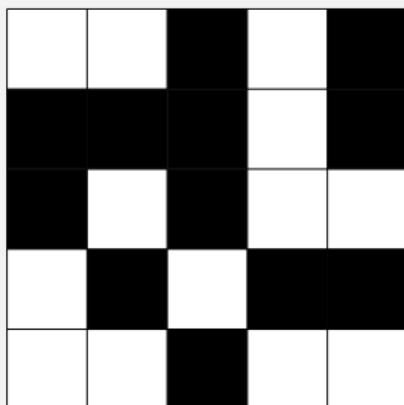
blocked site

Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an N -by- N system percolates?

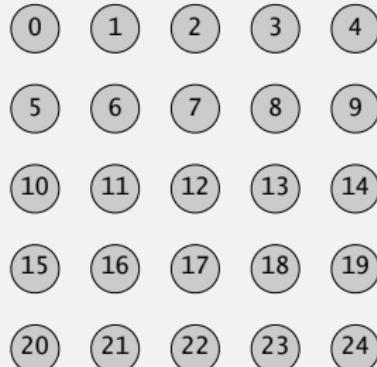
- Create an object for each site and name them 0 to $N^2 - 1$.

$N = 5$



open site

blocked site

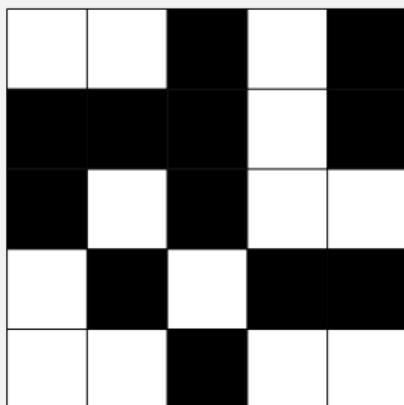


Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an N -by- N system percolates?

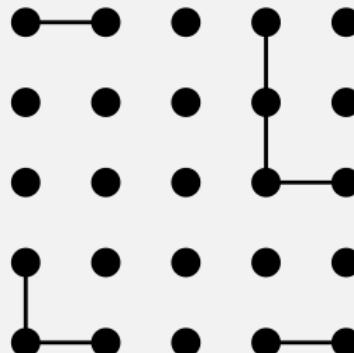
- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component iff connected by open sites.

$N = 5$



open site

blocked site



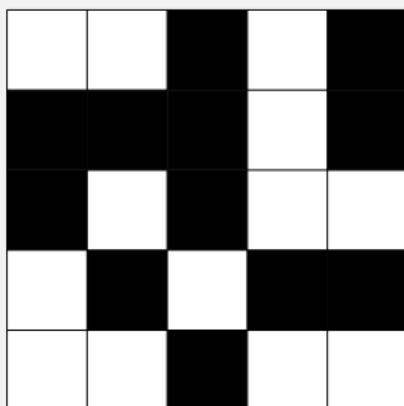
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an N -by- N system percolates?

- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component iff connected by open sites.
- Percolates iff any site on bottom row is connected to any site on top row.

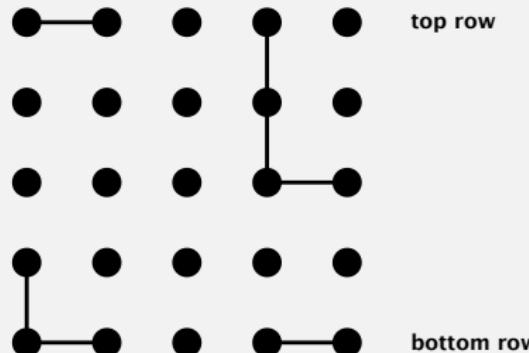
brute-force algorithm: N^2 calls to connected()

$N = 5$



open site

blocked site



top row

bottom row

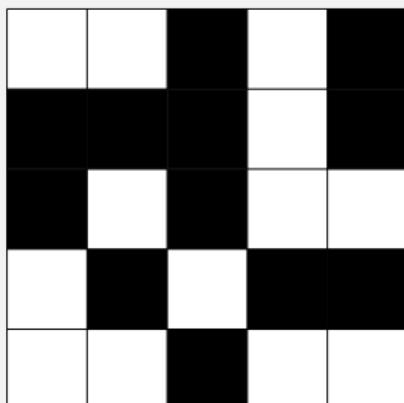
Dynamic connectivity solution to estimate percolation threshold

Clever trick. Introduce 2 virtual sites (and connections to top and bottom).

- Percolates iff virtual top site is connected to virtual bottom site.

more efficient algorithm: only 1 call to connected()

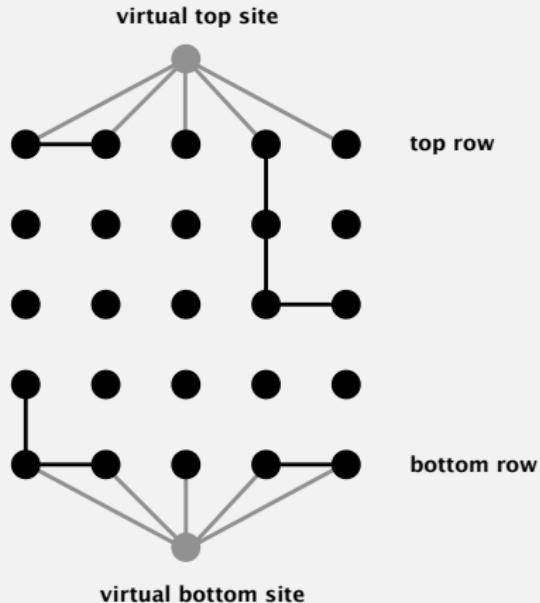
$N = 5$



open site

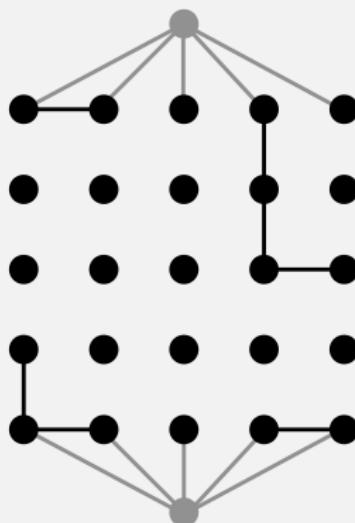
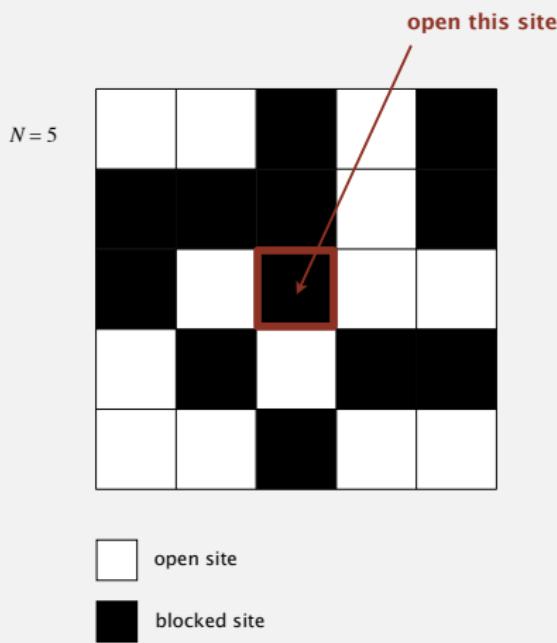


blocked site



Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

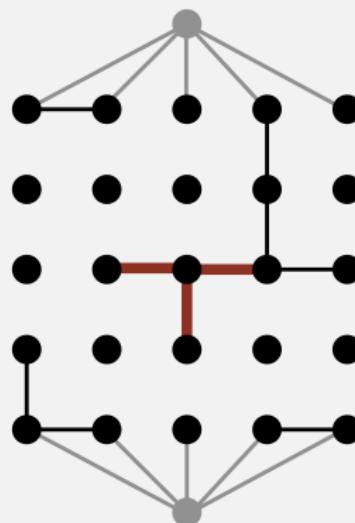
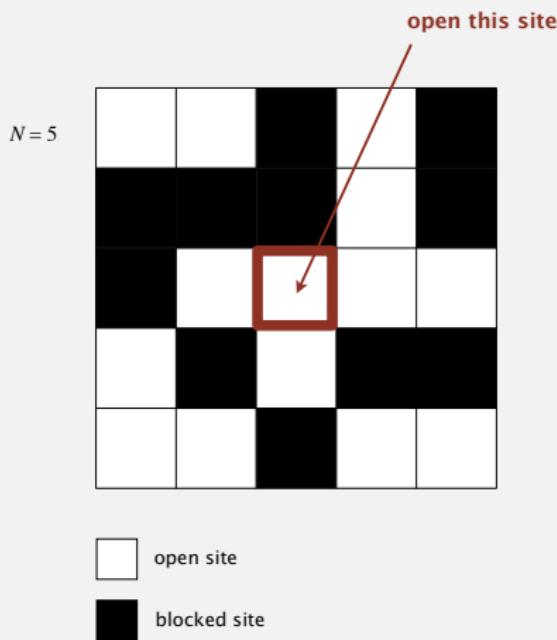


Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

A. Mark new site as open; connect it to all of its adjacent open sites.

up to 4 calls to union()

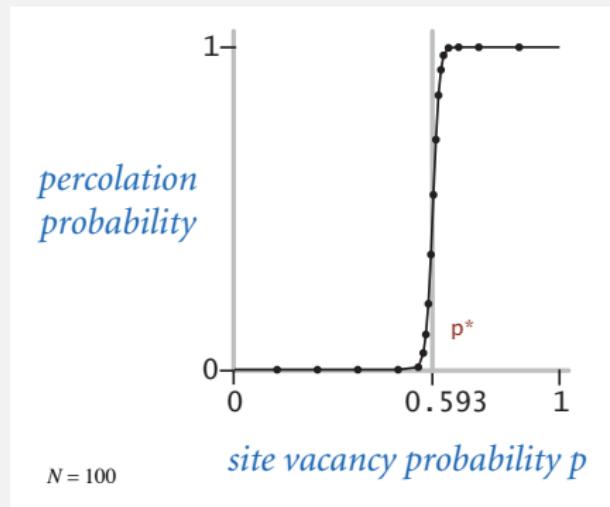


Percolation threshold

Q. What is percolation threshold p^* ?

A. About 0.592746 for large square lattices.

constant known only via simulation



Fast algorithm enables accurate answer to scientific question.

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.

CSU22010: ALGORITHMS AND DATA STRUCTURES

Lecture 18: Geometric Applications of Binary Search Trees

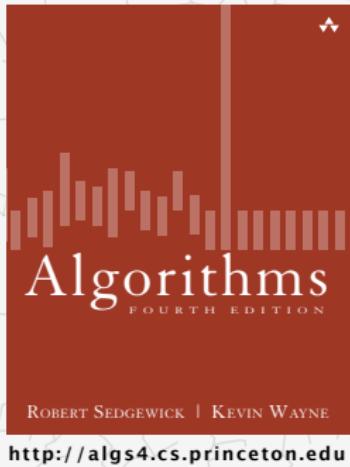
Vasileios Koutavas



School of Computer Science and Statistics
Trinity College Dublin

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

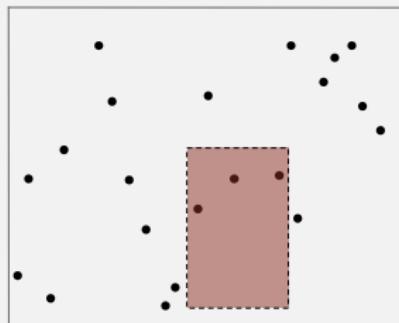


GEOMETRIC APPLICATIONS OF BSTs

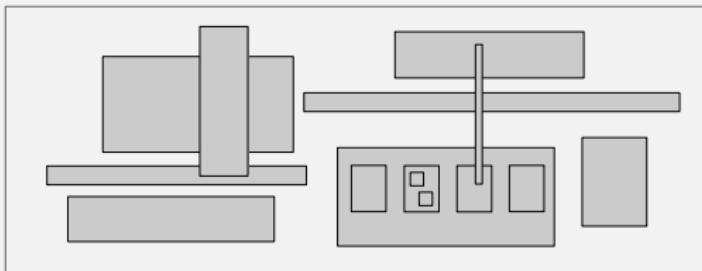
- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Overview

This lecture. Intersections among **geometric objects**.



2d orthogonal range search



orthogonal rectangle intersection

Applications. CAD, games, movies, virtual reality, databases, GIS,

Efficient solutions. **Binary search trees** (and extensions).



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

1d range search

Extension of ordered symbol table.

- Insert key-value pair.
- Search for key k .
- Delete key k .
- Range search: find all keys between k_1 and k_2 .
- Range count: number of keys between k_1 and k_2 .

Application. Database queries.

Geometric interpretation.

- Keys are point on a line.
- Find/count points in a given 1d interval.



insert B	B
insert D	B D
insert A	A B D
insert I	A B D I
insert H	A B D H I
insert F	A B D F H I
insert P	A B D F H I P
search G to K	H I
count G to K	2

1d range search: elementary implementations

Unordered list. Fast insert, slow range search.

Ordered array. Slow insert, binary search for k_1 and k_2 to do range search.

order of growth of running time for 1d range search

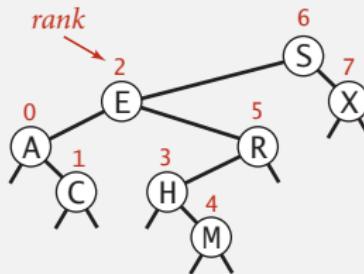
data structure	insert	range count	range search
unordered list	1	N	N
ordered array	N	$\log N$	$R + \log N$
goal	$\log N$	$\log N$	$R + \log N$

N = number of keys

R = number of keys that match

1d range count: BST implementation

1d 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

Proposition. Running time proportional to $\log N$.

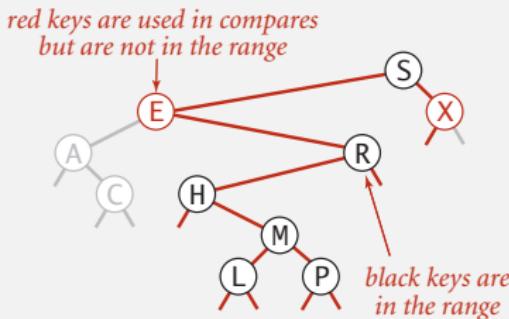
Pf. Nodes examined = search path to lo + search path to hi .

1d range search: BST implementation

1d range search. Find all keys between lo and hi .

- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).

searching in the range $[F \dots T]$



Proposition. Running time proportional to $R + \log N$.

Pf. Nodes examined = search path to lo + search path to hi + matches.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

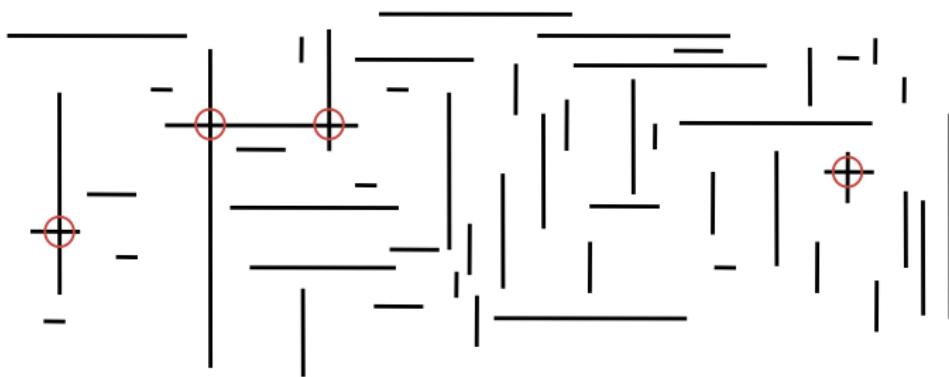
<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Orthogonal line segment intersection

Given N horizontal and vertical line segments, find all intersections.



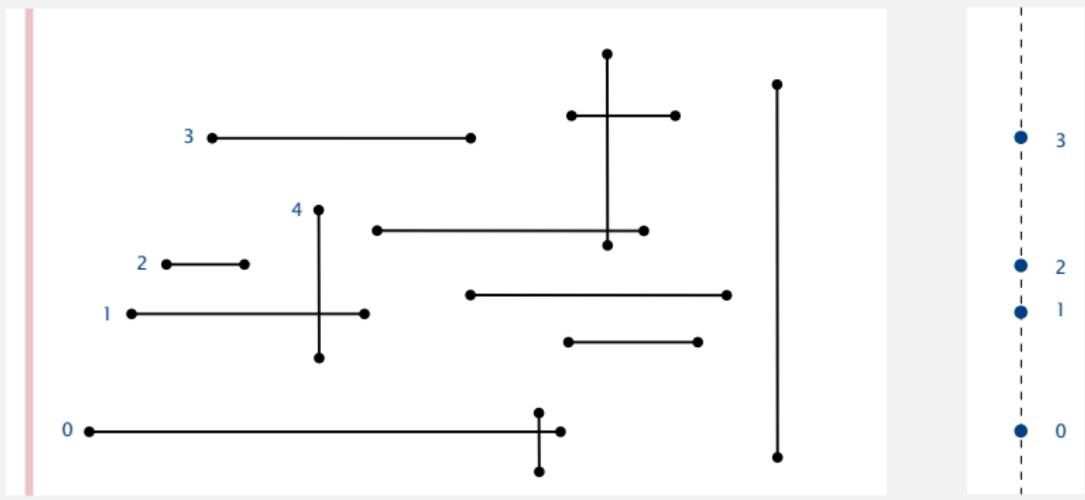
Quadratic algorithm. Check all pairs of line segments for intersection.

Nondegeneracy assumption. All x - and y -coordinates are distinct.

Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

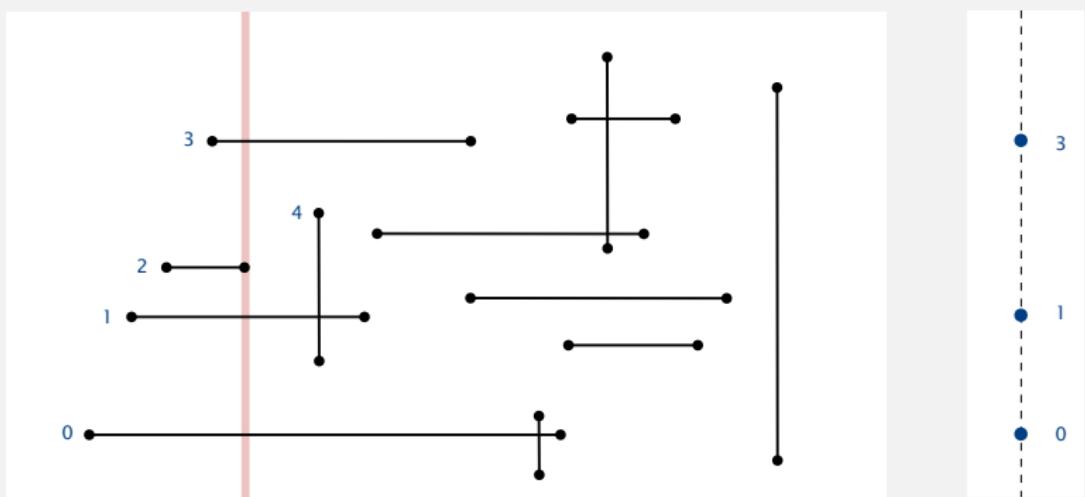
- x -coordinates define events.
- h -segment (left endpoint): insert y -coordinate into BST.



Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

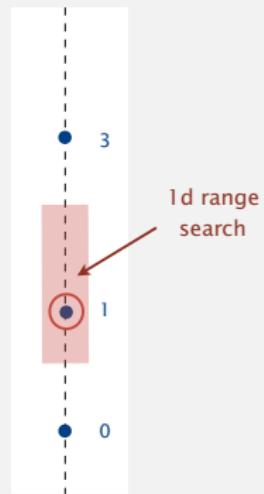
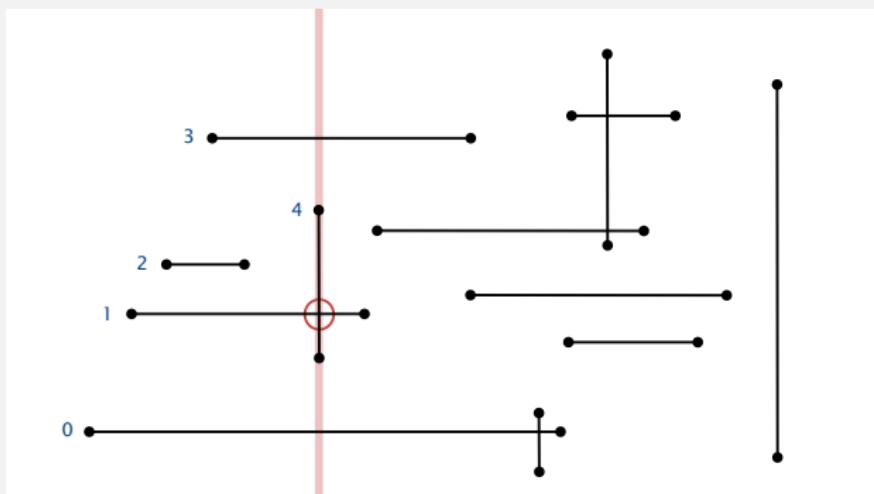
- x -coordinates define events.
- h -segment (left endpoint): insert y -coordinate into BST.
- h -segment (right endpoint): remove y -coordinate from BST.



Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

- x -coordinates define events.
- h -segment (left endpoint): insert y -coordinate into BST.
- h -segment (right endpoint): remove y -coordinate from BST.
- v -segment: range search for interval of y -endpoints.



Orthogonal line segment intersection: sweep-line analysis

Proposition. The sweep-line algorithm takes time proportional to $N \log N + R$ to find all R intersections among N orthogonal line segments.

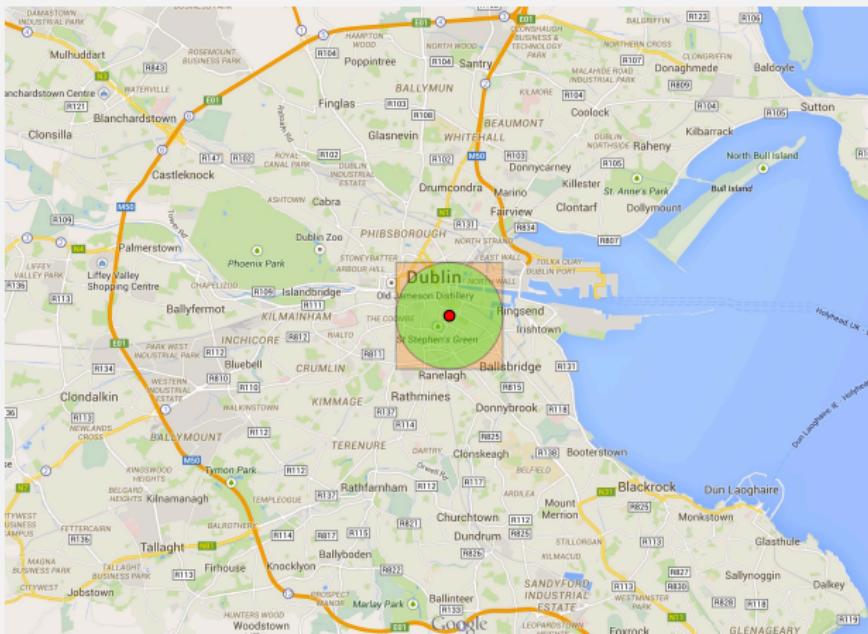
Pf.

- Put x -coordinates on a PQ (or sort). $\leftarrow N \log N$
- Insert y -coordinates into BST. $\leftarrow N \log N$
- Delete y -coordinates from BST. $\leftarrow N \log N$
- Range searches in BST. $\leftarrow N \log N + R$

Bottom line. Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.

OTHER GEOMETRIC QUERIES

- Q: what bus stops are within 500m from my location?
Q: what petrol stations are within 2km from Trinity College?



We need a data structure that can support 2-D queries.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

2-d orthogonal range search

Extension of ordered symbol-table to 2d keys.

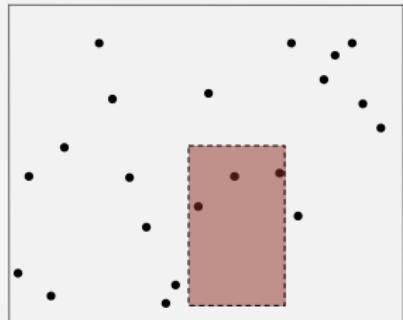
- Insert a 2d key.
- Delete a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range.
- Range count: number of keys that lie in a 2d range.

Applications. Networking, circuit design, databases, ...

Geometric interpretation.

- Keys are point in the plane.
- Find/count points in a given *h-v* rectangle

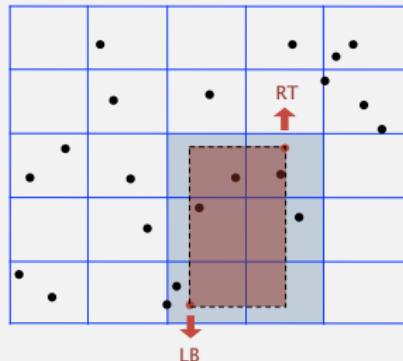
↑
rectangle is axis-aligned



2d orthogonal range search: grid implementation

Grid implementation.

- Divide space into M -by- M grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add (x, y) to list for corresponding square.
- Range search: examine only squares that intersect 2d range query.



2d orthogonal range search: grid implementation analysis

Space-time tradeoff.

- Space: $M^2 + N$.
- Time: $1 + N/M^2$ per square examined, on average.

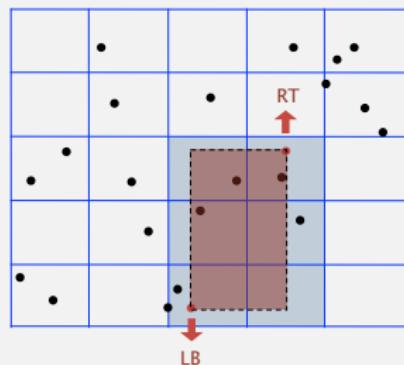
Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: \sqrt{N} -by- \sqrt{N} grid.

Running time. [if points are evenly distributed]

- Initialize data structure: N .
- Insert point: 1.
- Range search: 1 per point in range.

choose $M \sim \sqrt{N}$



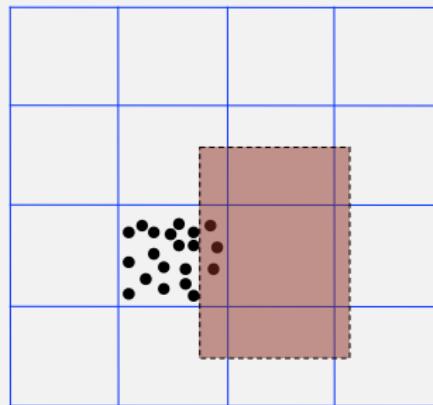
Clustering

Grid implementation. Fast, simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

- Lists are too long, even though average length is short.
- Need data structure that adapts gracefully to data.

Ex: Many buses stop in the centre of a city, few in the outskirts.



Clustering

Grid implementation. Fast, simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.



13,000 points, 1000 grid squares



half the squares are empty

half the points are
in 10% of the squares

Space-partitioning trees

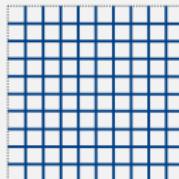
Use a **tree** to represent a recursive subdivision of 2d space.

Grid. Divide space uniformly into squares.

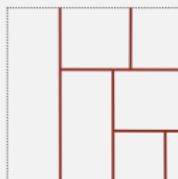
2d tree. Recursively divide space into two halfplanes.

Quadtree. Recursively divide space into four quadrants.

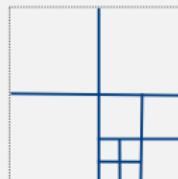
BSP tree. Recursively divide space into two regions.



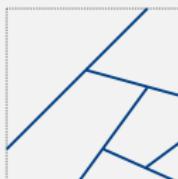
Grid



2d tree



Quadtree

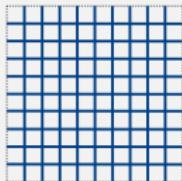


BSP tree

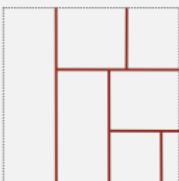
Space-partitioning trees: applications

Applications.

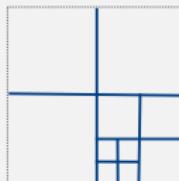
- Ray tracing.
- 2d range search.
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- Nearest neighbor search.
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.



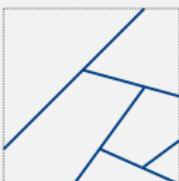
Grid



2d tree



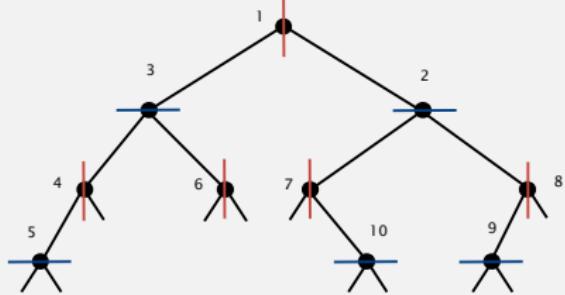
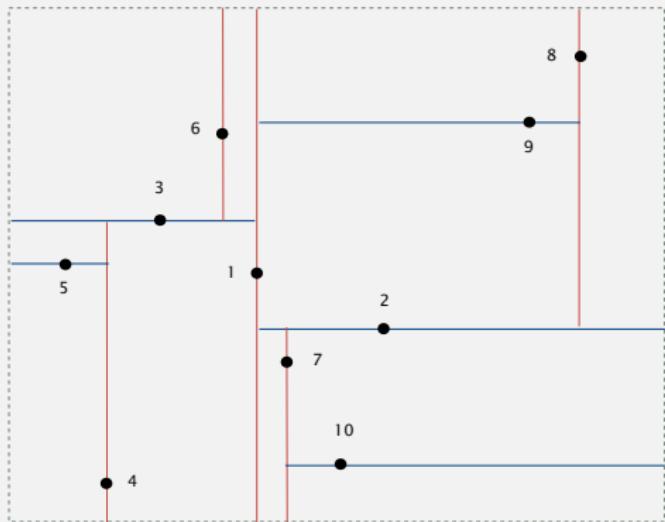
Quadtree



BSP tree

2d tree construction

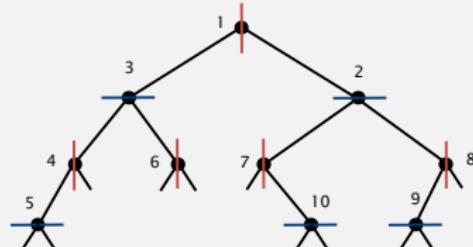
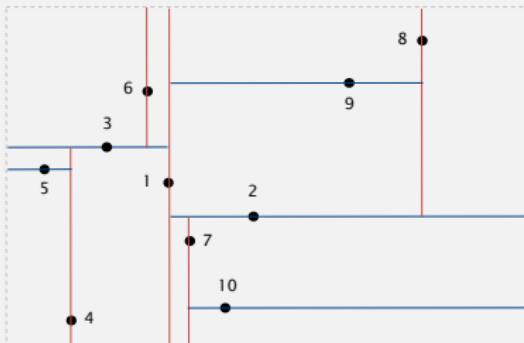
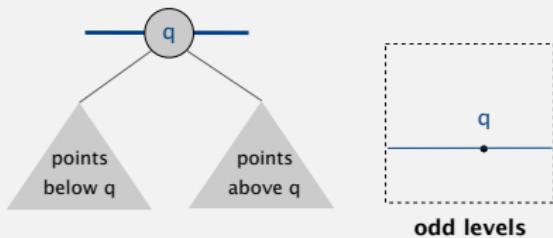
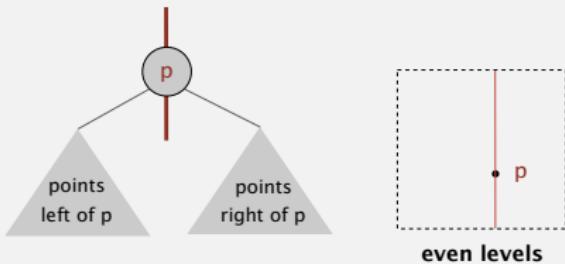
Recursively partition plane into two halfplanes.



2d tree implementation

Data structure. BST, but alternate using x - and y -coordinates as key.

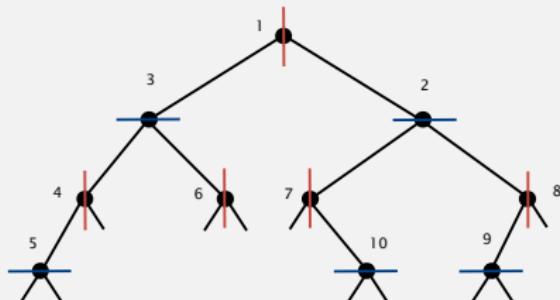
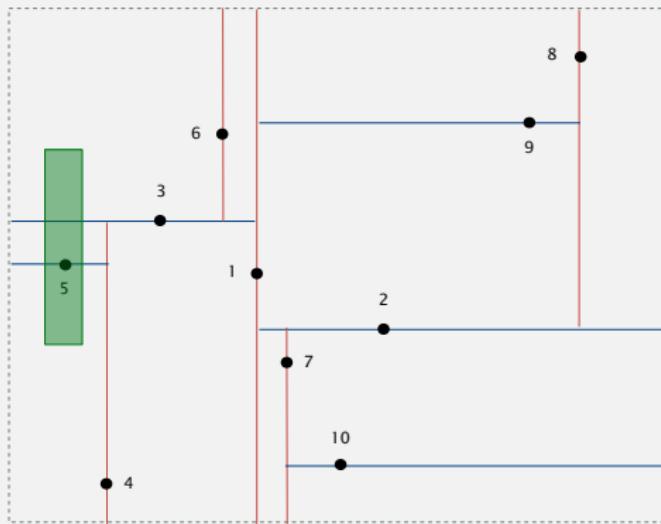
- Search gives rectangle containing point.
- Insert further subdivides the plane.



2d tree demo: range search

Goal. Find all points in a query axis-aligned rectangle.

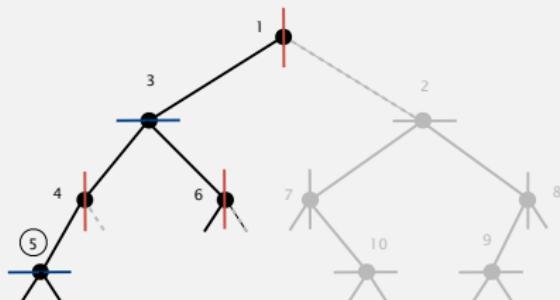
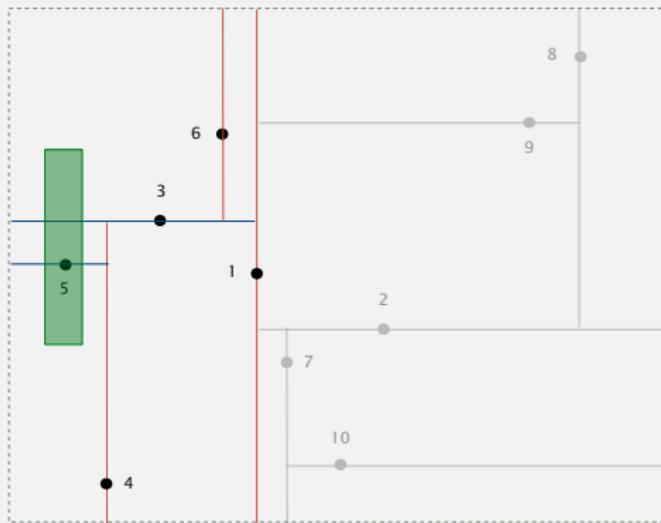
- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).



2d tree demo: range search

Goal. Find all points in a query axis-aligned rectangle.

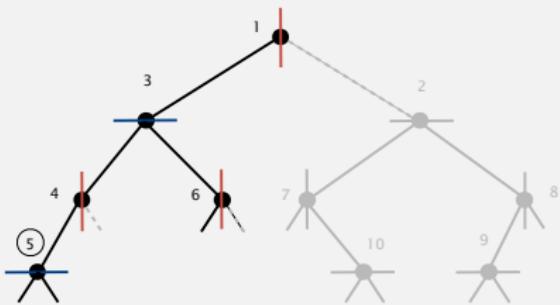
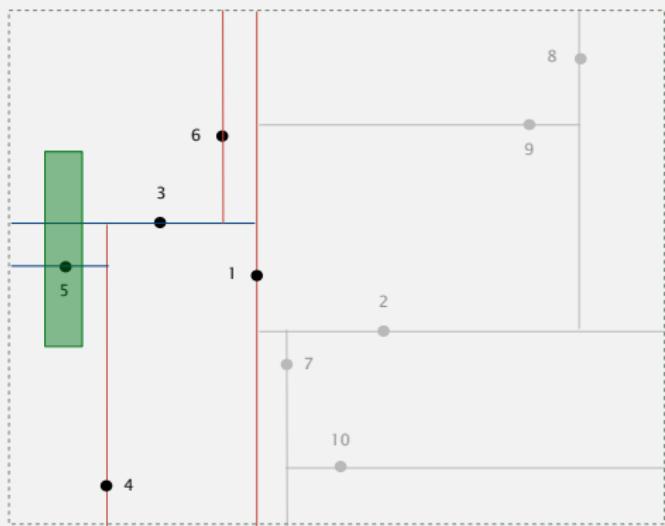
- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).



Range search in a 2d tree analysis

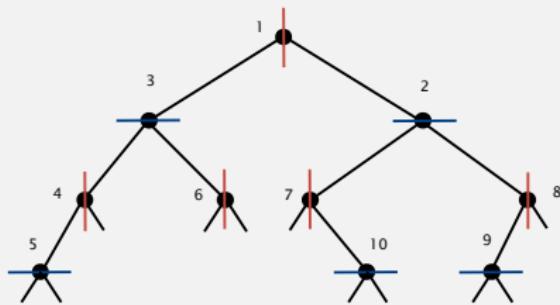
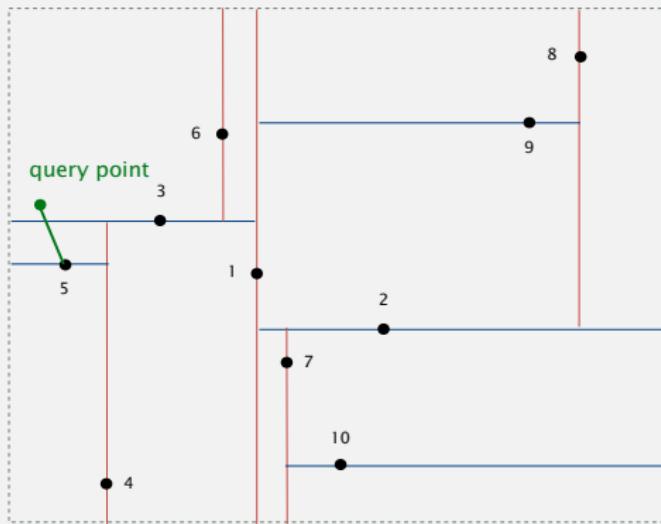
Typical case. $R + \log N$.

Worst case (assuming tree is balanced). $R + \sqrt{N}$.



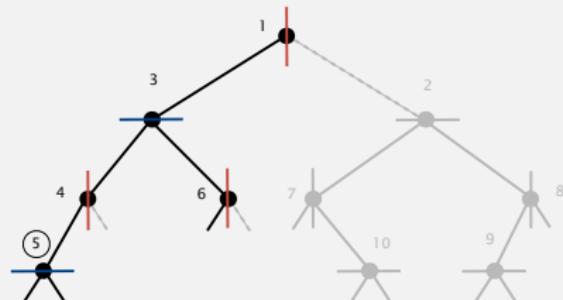
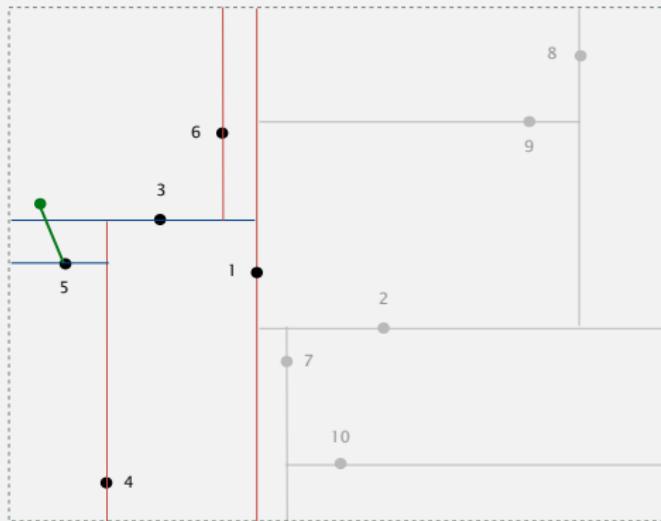
2d tree demo: nearest neighbor

Goal. Find closest point to query point.



2d tree demo: nearest neighbor

- Check distance from point in node to query point.
- Recursively search left/bottom (if it could contain a closer point).
- Recursively search right/top (if it could contain a closer point).
- Organize method so that it begins by searching for query point.

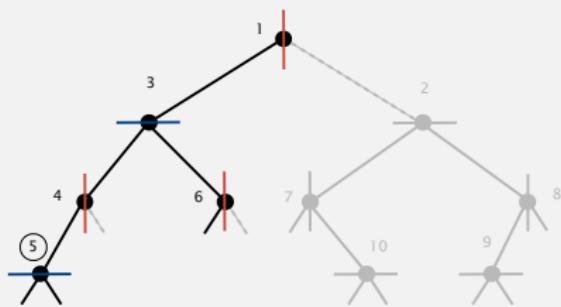
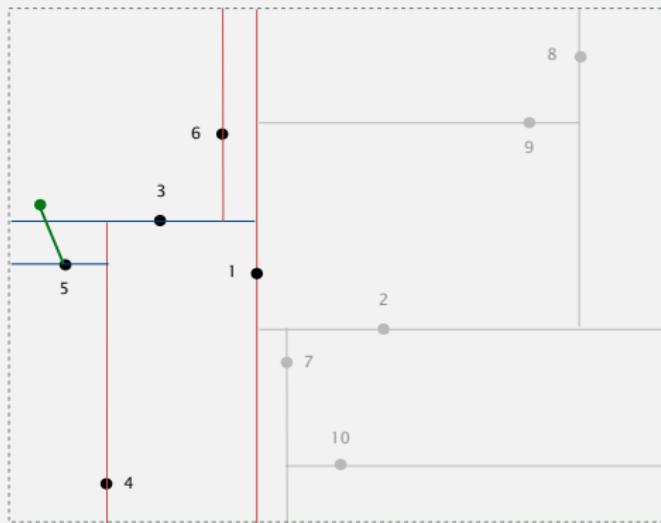


nearest neighbor = 5

Nearest neighbor search in a 2d tree analysis

Typical case. $\log N$.

Worst case (even if tree is balanced). N .



nearest neighbor = 5

Flocking birds

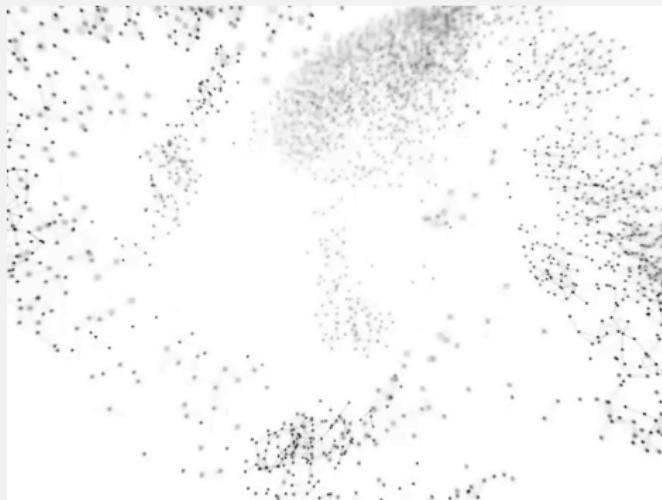
Q. What "natural algorithm" do starlings, migrating geese, starlings, cranes, bait balls of fish, and flashing fireflies use to flock?



Flocking boids [Craig Reynolds, 1986]

Boids. Three simple rules lead to complex emergent flocking behavior:

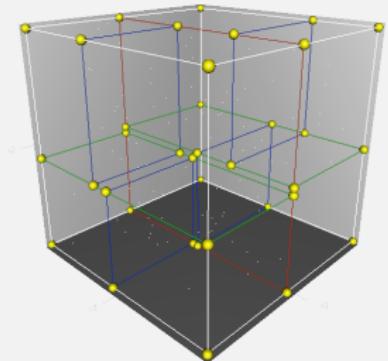
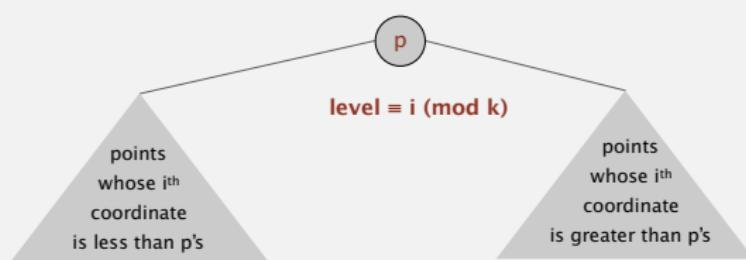
- Collision avoidance: point away from k nearest boids.
- Flock centering: point towards the center of mass of k nearest boids.
- Velocity matching: update velocity to the average of k nearest boids.



Kd tree

Kd tree. Recursively partition k -dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.



Efficient, simple data structure for processing k -dimensional data.

- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad in an algorithms class!



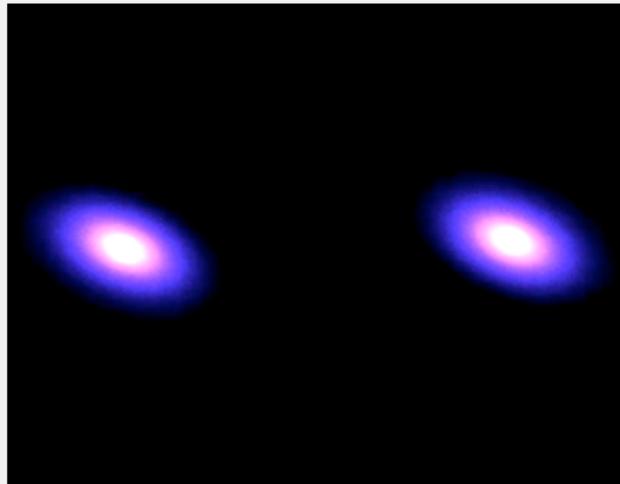
Jon Bentley

N-body simulation

Goal. Simulate the motion of N particles, mutually affected by gravity.

Brute force. For each pair of particles, compute force: $F = \frac{G m_1 m_2}{r^2}$

Running time. Time per step is N^2 .



http://www.youtube.com/watch?v=ua7YlN4eL_w

Appel's algorithm for N-body simulation

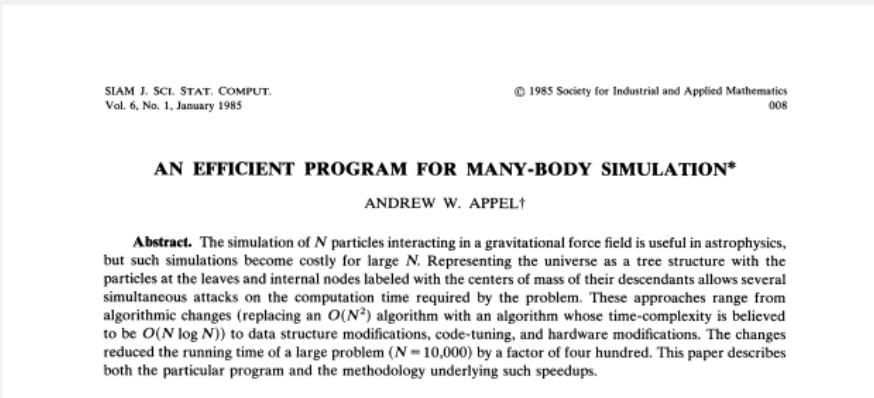
Key idea. Suppose particle is far, far away from cluster of particles.

- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and **center of mass** of aggregate.



Appel's algorithm for N-body simulation

- Build 3d-tree with N particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large.



Impact. Running time per step is $N \log N \Rightarrow$ enables new research.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

1d interval search

1d interval search. Data structure to hold set of (overlapping) intervals.

- Insert an interval (lo, hi) .
- Search for an interval (lo, hi) .
- Delete an interval (lo, hi) .
- **Interval intersection query:** given an interval (lo, hi) , find all intervals (or one interval) in data structure that intersects (lo, hi) .

Q. Which intervals intersect $(9, 16)$?

A. $(7, 10)$ and $(15, 18)$.



1d interval search API

```
public class IntervalST<Key extends Comparable<Key>, Value>
```

```
    IntervalST()
```

create interval search tree

```
    void put(Key lo, Key hi, Value val)
```

put interval-value pair into ST

```
    Value get(Key lo, Key hi)
```

value paired with given interval

```
    void delete(Key lo, Key hi)
```

delete the given interval

```
    Iterable<Value> intersects(Key lo, Key hi)
```

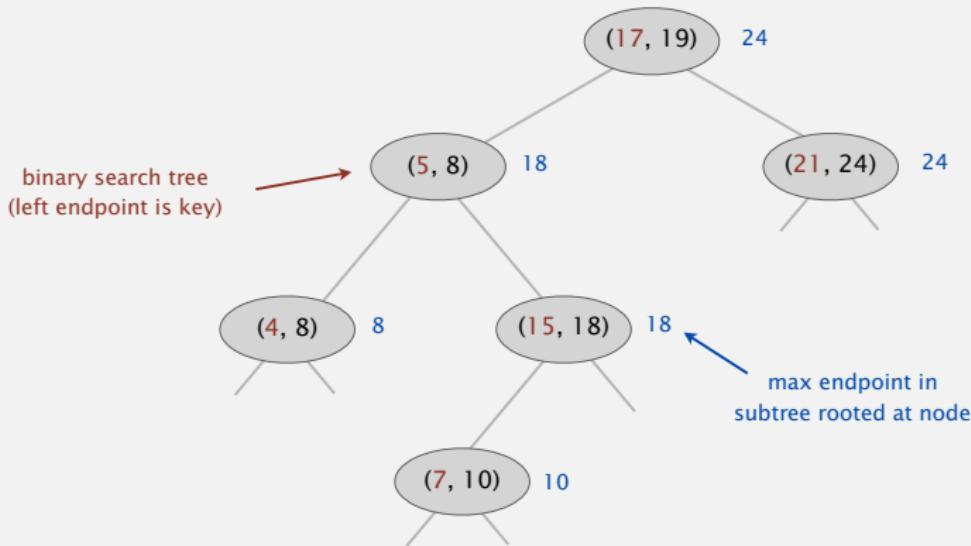
all intervals that intersect (lo, hi)

Nondegeneracy assumption. No two intervals have the same left endpoint.

Interval search trees

Create BST, where each node stores an interval (lo, hi).

- Use left endpoint as BST key.
- Store max endpoint in subtree rooted at node.



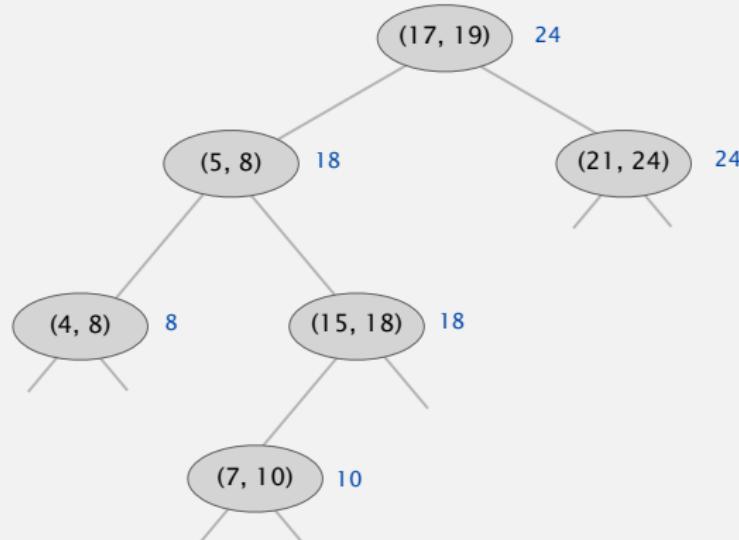
Interval search tree demo: insertion

To insert an interval (lo , hi):

- Insert into BST, using lo as the key.
- Update max in each node on search path.



insert interval (16, 22)



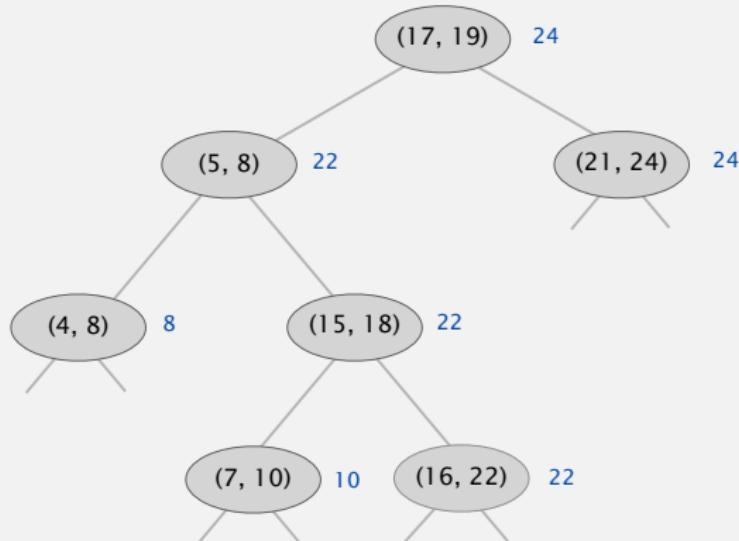
Interval search tree demo: insertion

To insert an interval (lo , hi):

- Insert into BST, using lo as the key.
- Update max in each node on search path.



insert interval (16, 22)

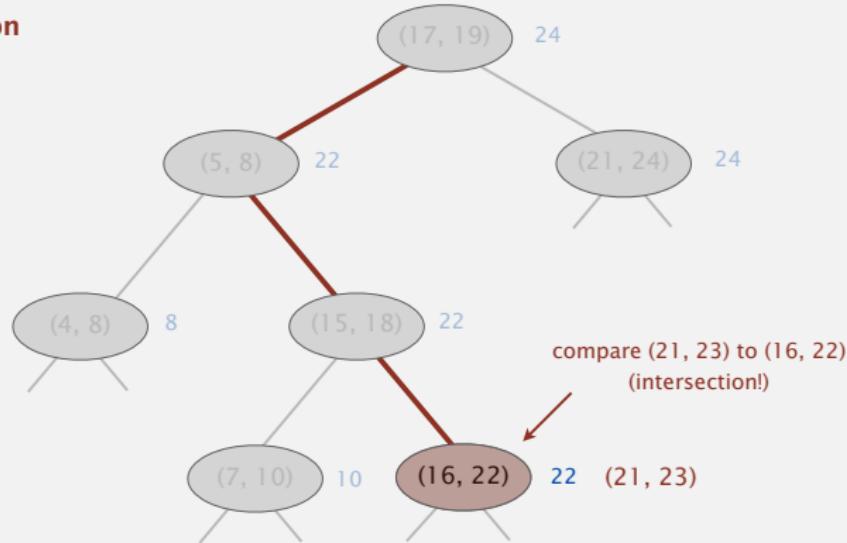


Interval search tree demo: intersection

To search for any one interval that intersects query interval (lo , hi):

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

interval intersection
search for (21, 23)



Search for an intersecting interval: implementation

To search for any one interval that intersects query interval (lo, hi):

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

```
Node x = root;
while (x != null)
{
    if      (x.interval.intersects(lo, hi)) return x.interval;
    else if (x.left == null)                  x = x.right;
    else if (x.left.max < lo)                x = x.right;
    else                                     x = x.left;
}
return null;
```

Search for an intersecting interval: analysis

To search for any one interval that intersects query interval (lo, hi) :

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

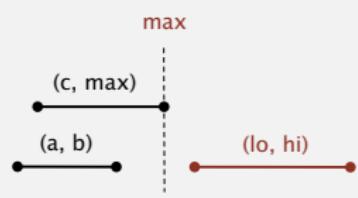
Case 1. If search goes **right**, then no intersection in left.

Pf. Suppose search goes right and left subtree is non empty.

- Since went right, we have $\max < lo$.
- For any interval (a, b) in left subtree of x ,
we have $b \leq \max < lo$.

definition of max reason for going right

- Thus, (a, b) will not intersect (lo, hi) .



left subtree of x

right subtree of x

Search for an intersecting interval: analysis

To search for any one interval that intersects query interval (lo, hi) :

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

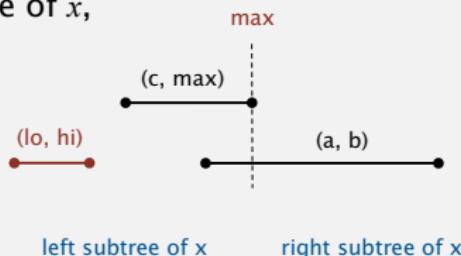
Case 2. If search goes **left**, then there is either an intersection in left subtree or no intersections in either.

Pf. Suppose no intersection in left.

- Since went left, we have $lo \leq max$.
- Then for any interval (a, b) in right subtree of x ,
 $hi < c \leq a \Rightarrow$ no intersection in right.

no intersections
in left subtree

intervals sorted
by left endpoint



Interval search tree: analysis

Implementation. Use a red-black BST to guarantee performance.



easy to maintain auxiliary information
($\log N$ extra work per operation)

operation	brute	interval search tree	best in theory
insert interval	1	$\log N$	$\log N$
find interval	N	$\log N$	$\log N$
delete interval	N	$\log N$	$\log N$
find any one interval that intersects (lo, hi)	N	$\log N$	$\log N$
find all intervals that intersects (lo, hi)	N	$R \log N$	$R + \log N$

order of growth of running time for N intervals



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

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

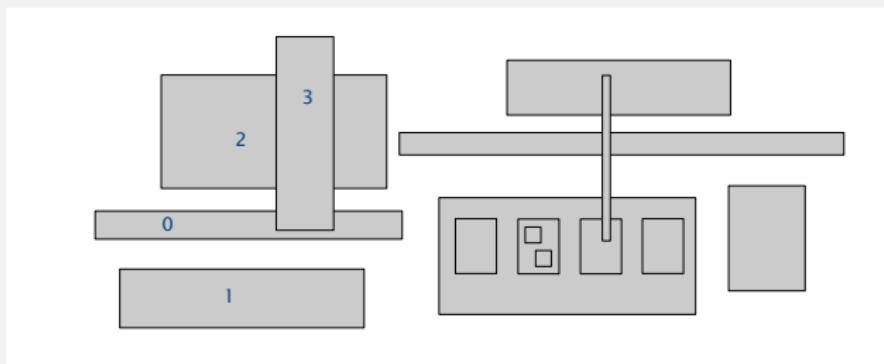
GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Orthogonal rectangle intersection

Goal. Find all intersections among a set of N orthogonal rectangles.

Quadratic algorithm. Check all pairs of rectangles for intersection.



Non-degeneracy assumption. All x - and y -coordinates are distinct.

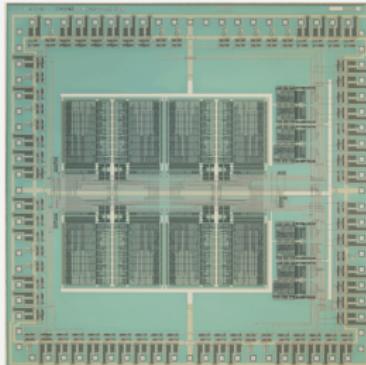
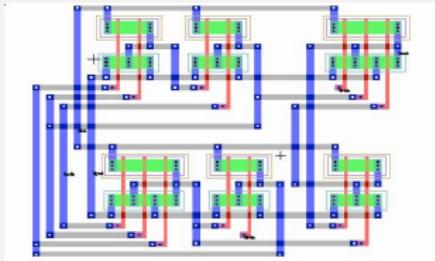
Microprocessors and geometry

Early 1970s. microprocessor design became a **geometric** problem.

- Very Large Scale Integration (VLSI).
- Computer-Aided Design (CAD).

Design-rule checking.

- Certain wires cannot intersect.
- Certain spacing needed between different types of wires.
- Debugging = orthogonal rectangle intersection search.



Algorithms and Moore's law

"Moore's law." Processing power doubles every 18 months.

- $197x$: check N rectangles.
- $197(x+1.5)$: check $2N$ rectangles on a $2x$ -faster computer.



Gordon Moore

Bootstrapping. We get to use the faster computer for bigger circuits.

But bootstrapping is not enough if using a quadratic algorithm:

- $197x$: takes M days.
- $197(x+1.5)$: takes $(4M)/2 = 2M$ days. (!)

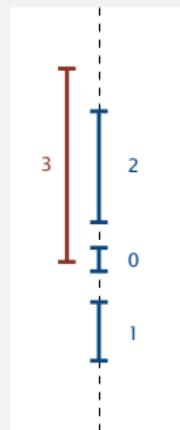
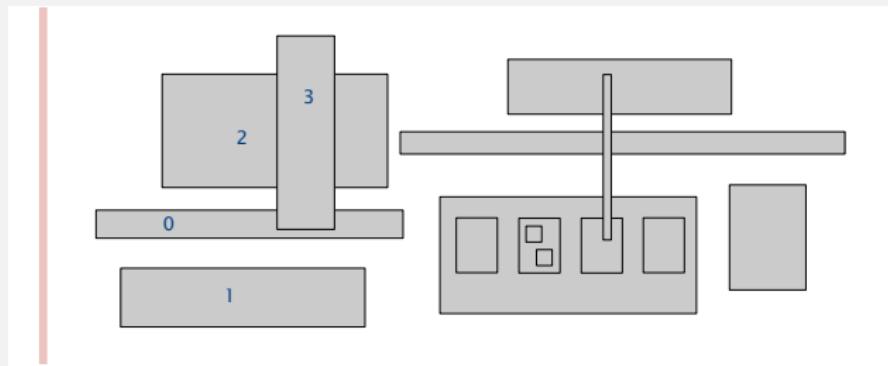


Bottom line. Linearithmic algorithm is **necessary** to sustain Moore's Law.

Orthogonal rectangle intersection: sweep-line algorithm

Sweep vertical line from left to right.

- x -coordinates of left and right endpoints define events.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using y -intervals of rectangle).
- Left endpoint: interval search for y -interval of rectangle; insert y -interval.
- Right endpoint: remove y -interval.



y -coordinates

Orthogonal rectangle intersection: sweep-line analysis

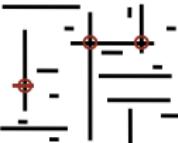
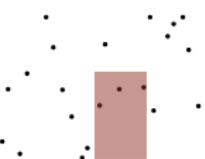
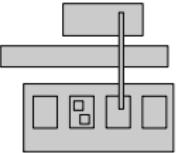
Proposition. Sweep line algorithm takes time proportional to $N \log N + R \log N$ to find R intersections among a set of N rectangles.

Pf.

- Put x -coordinates on a PQ (or sort). $\leftarrow N \log N$
- Insert y -intervals into ST. $\leftarrow N \log N$
- Delete y -intervals from ST. $\leftarrow N \log N$
- Interval searches for y -intervals. $\leftarrow N \log N + R \log N$

Bottom line. Sweep line reduces 2d orthogonal rectangle intersection search to 1d interval search.

Geometric applications of BSTs

problem	example	solution
1d range search	BST
2d orthogonal line segment intersection		sweep line reduces to 1d range search
kd range search		kd tree
1d interval search		interval search tree
2d orthogonal rectangle intersection		sweep line reduces to 1d interval search