

Advanced Data Structures

Conrado Martínez
U. Politècnica de Catalunya

March 14, 2025



Part I

Techniques

- 1 Experimental Algorithmics
- 2 Probabilistic Analysis of Algorithms
- 3 Amortized Analysis

Part I

Techniques

1 Experimental Algorithmics

2 Probabilistic Analysis of Algorithms

- The Continuous Master Theorem
 - Example #1: Quicksort
 - Example #2: Quickselect
- Probabilistic Tools

3 Amortized Analysis

Experimental Algorithmics: Introduction

- What is Experimental Algorithmics?
- Why doing experiments (with algorithms)?

Experimental Algorithmics: Introduction

Goals of the analysis of algorithms (and data structures):

- ➊ To predict the amount of computational resources needed by an algorithm, in terms of simple parameters, e.g., *size*.
- ➋ To compare the performance of competing alternative solutions.
- ➌ To help and guide the design of new algorithms or variants of existing ones.
- ➍ To explain the observed performance of algorithms.

Experimental Algorithmics: Introduction

Goals of the analysis of algorithms (and data structures):

- ① To predict the amount of computational resources needed by an algorithm, in terms of simple parameters, e.g., *size*.
- ② To compare the performance of competing alternative solutions.
- ③ To help and guide the design of new algorithms or variants of existing ones.
- ④ To explain the observed performance of algorithms.

Experimental Algorithmics: Introduction

Goals of the analysis of algorithms (and data structures):

- ① To predict the amount of computational resources needed by an algorithm, in terms of simple parameters, e.g., *size*.
- ② To compare the performance of competing alternative solutions.
- ③ To help and guide the design of new algorithms or variants of existing ones.
- ④ To explain the observed performance of algorithms.

Experimental Algorithmics: Introduction

Goals of the analysis of algorithms (and data structures):

- ① To predict the amount of computational resources needed by an algorithm, in terms of simple parameters, e.g., *size*.
- ② To compare the performance of competing alternative solutions.
- ③ To help and guide the design of new algorithms or variants of existing ones.
- ④ To explain the observed performance of algorithms.

Experimental Algorithmics: Introduction

- Goals #1 and #4 are “scientific” goals.
- Goals #2 and #3 are “engineering” goals.

Experimental Algorithmics: Introduction

- Goals #1 and #4 are “scientific” goals.
- Goals #2 and #3 are “engineering” goals.

Experimental Algorithmics: Introduction

At a deep level:

- Goals #1 (predict) and #4 (explain) are identical to the main goals of any other science
- We look for quantitative predictions, the use of mathematical models that provide measurable and precise descriptions of the behavior of a system (which ultimately explain it)
- The rôle of experiments in Computer Science is the rôle they play in the **scientific method**.

Experimental Algorithmics: Introduction

At a deep level:

- Goals #1 (predict) and #4 (explain) are identical to the main goals of any other science
- We look for quantitative predictions, the use of mathematical models that provide measurable and precise descriptions of the behavior of a system (which ultimately explain it)
- The rôle of experiments in Computer Science is the rôle they play in the **scientific method**.

Experimental Algorithmics: Introduction

At a deep level:

- Goals #1 (predict) and #4 (explain) are identical to the main goals of any other science
- We look for quantitative predictions, the use of mathematical models that provide measurable and precise descriptions of the behavior of a system (which ultimately explain it)
- The rôle of experiments in Computer Science is the rôle they play in the **scientific method**.

Intro to EA

- ➊ Experiments are the source of (controlled) observations, a very fruitful starting point for the scientific endeavour
- ➋ They help us develop hypotheses and intuitions about the behavior of algorithms (*pilot studies*)
- ➌ They serve to test the hypotheses and refine them
- ➍ They are the ultimate yardstick for the utility of our computational and mathematical models
- ➎ Exhaustive experiments provide compelling evidence to support the hypotheses and validate mathematical models (*workhorse studies*)

Intro to EA

- ➊ Experiments are the source of (controlled) observations, a very fruitful starting point for the scientific endeavour
- ➋ They help us develop hypotheses and intuitions about the behavior of algorithms (**pilot studies**)
- ➌ They serve to test the hypotheses and refine them
- ➍ They are the ultimate yardstick for the utility of our computational and mathematical models
- ➎ Exhaustive experiments provide compelling evidence to support the hypotheses and validate mathematical models (**workhorse studies**)

Intro to EA

- ➊ Experiments are the source of (controlled) observations, a very fruitful starting point for the scientific endeavour
- ➋ They help us develop hypotheses and intuitions about the behavior of algorithms (**pilot studies**)
- ➌ They serve to test the hypotheses and refine them
- ➍ They are the ultimate yardstick for the utility of our computational and mathematical models
- ➎ Exhaustive experiments provide compelling evidence to support the hypotheses and validate mathematical models (**workhorse studies**)

Intro to EA

- ➊ Experiments are the source of (controlled) observations, a very fruitful starting point for the scientific endeavour
- ➋ They help us develop hypotheses and intuitions about the behavior of algorithms (**pilot studies**)
- ➌ They serve to test the hypotheses and refine them
- ➍ They are the ultimate yardstick for the utility of our computational and mathematical models
- ➎ Exhaustive experiments provide compelling evidence to support the hypotheses and validate mathematical models (**workhorse studies**)

Intro to EA

- ➊ Experiments are the source of (controlled) observations, a very fruitful starting point for the scientific endeavour
- ➋ They help us develop hypotheses and intuitions about the behavior of algorithms (**pilot studies**)
- ➌ They serve to test the hypotheses and refine them
- ➍ They are the ultimate yardstick for the utility of our computational and mathematical models
- ➎ Exhaustive experiments provide compelling evidence to support the hypotheses and validate mathematical models (**workhorse studies**)

Intro to EA

- ➊ Experiments can be used to check to what extent the conclusions drawn from the models apply in (real life?) situations where some of our assumptions do not hold, e.g., randomness of the input, independence, etc.
- ➋ If your model does not apply, try to find an explanation for failure
- ➌ Correctness is not an absolute concept in natural sciences: the claim that the Earth is an sphere is not correct, but it more “correct” than the claim it is plane! Use experiments to quantify the “correctness” of your model

Intro to EA

- ➊ Experiments can be used to check to what extent the conclusions drawn from the models apply in (real life?) situations where some of our assumptions do not hold, e.g., randomness of the input, independence, etc.
- ➋ If your model does not apply, try to find an explanation for failure
- ➌ Correctness is not an absolute concept in natural sciences: the claim that the Earth is an sphere is not correct, but it more “correct” than the claim it is plane! Use experiments to quantify the “correctness” of your model

Intro to EA

- ➊ Experiments can be used to check to what extent the conclusions drawn from the models apply in (real life?) situations where some of our assumptions do not hold, e.g., randomness of the input, independence, etc.
- ➋ If your model does not apply, try to find an explanation for failure
- ➌ Correctness is not an absolute concept in natural sciences: the claim that the Earth is an sphere is not correct, but it more “correct” than the claim it is plane! Use experiments to quantify the “correctness” of your model

Intro to EA

- ➊ **Simulations** are closely related to experiments but a simulation produces (numerical) data according to a theoretical model
- ➋ Simulations are very useful to investigate when the asymptotic regime starts, estimate the magnitude of hidden constants, etc.
- ➌ For us it is often difficult to draw a line between experiments and simulations: the algorithms (\equiv nature) might be seen as models themselves!

Intro to EA

- ➊ **Simulations** are closely related to experiments but a simulation produces (numerical) data according to a theoretical model
- ➋ Simulations are very useful to investigate when the asymptotic regime starts, estimate the magnitude of hidden constants, etc.
- ➌ For us it is often difficult to draw a line between experiments and simulations: the algorithms (\equiv nature) might be seen as models themselves!

Intro to EA

- ➊ **Simulations** are closely related to experiments but a simulation produces (numerical) data according to a theoretical model
- ➋ Simulations are very useful to investigate when the asymptotic regime starts, estimate the magnitude of hidden constants, etc.
- ➌ For us it is often difficult to draw a line between experiments and simulations: the algorithms (\equiv nature) might be seen as models themselves!

Intro to EA: Do's and Don'ts

- ➊ Experiments should be set up with a falsifiable hypothesis (that's what the scientific method requires!)
- ➋ If not, they're fine for the exploratory phase, but not much more . . . They might be good to illustrate your point, but be aware of their limited value
- ➌ Experiments must be reproducible: better report artifact-independent measures!

Intro to EA: Do's and Don'ts

- ① Experiments should be set up with a falsifiable hypothesis (that's what the scientific method requires!)
- ② If not, they're fine for the exploratory phase, but not much more . . . They might be good to illustrate your point, but be aware of their limited value
- ③ Experiments must be reproducible: better report artifact-independent measures!

Intro to EA: Do's and Don'ts

- ① Experiments should be set up with a falsifiable hypothesis (that's what the scientific method requires!)
- ② If not, they're fine for the exploratory phase, but not much more . . . They might be good to illustrate your point, but be aware of their limited value
- ③ Experiments must be reproducible: better report artifact-independent measures!

Intro to EA: Do's and Don'ts

- Empirical comparative studies (“running races”) can raise your adrenaline but have **little or no explicative power** ...
- They are OK from the engineering perspective
- Comparing two variants A' and A'' of an algorithm is useful from the scientific point of view; the differences in performance could hopefully be explained in terms of the (small) differences between A' and A''

Intro to EA: Do's and Don'ts

- Empirical comparative studies (“running races”) can raise your adrenaline but have **little or no explicative power** ...
- They are OK from the engineering perspective
- Comparing two variants A' and A'' of an algorithm is useful from the scientific point of view; the differences in performance could hopefully be explained in terms of the (small) differences between A' and A''

Intro to EA: Do's and Don'ts

- Empirical comparative studies (“running races”) can raise your adrenaline but have **little or no explicative power** ...
- They are OK from the engineering perspective
- Comparing two variants A' and A'' of an algorithm is useful from the scientific point of view; the differences in performance could hopefully be explained in terms of the (small) differences between A' and A''

Intro to EA: Do's and Don'ts

- CPU time depends on many factors, including the instrument of measure (the computer)!
- A serious scientific study of CPU time and other machine-dependent features must analyze and explain each of the factors involved: run the experiments for different architectures, programming languages, ...
- Studies of CPU time versus input size alone are more often than not useless; we need experiments to reveal the dependence of CPU time on several parameters
- Experiments at that level of instantiation (see later) are difficult to reproduce

Intro to EA: Do's and Don'ts

- CPU time depends on many factors, including the instrument of measure (the computer)!
- A serious scientific study of CPU time and other machine-dependent features must analyze and explain each of the factors involved: run the experiments for different architectures, programming languages, ...
- Studies of CPU time versus input size alone are more often than not useless; we need experiments to reveal the dependence of CPU time on several parameters
- Experiments at that level of instantiation (see later) are difficult to reproduce

Intro to EA: Do's and Don'ts

- CPU time depends on many factors, including the instrument of measure (the computer)!
- A serious scientific study of CPU time and other machine-dependent features must analyze and explain each of the factors involved: run the experiments for different architectures, programming languages, ...
- Studies of CPU time versus input size alone are more often than not useless; we need experiments to reveal the dependence of CPU time on several parameters
- Experiments at that level of instantiation (see later) are difficult to reproduce

Intro to EA: Do's and Don'ts

- CPU time depends on many factors, including the instrument of measure (the computer)!
- A serious scientific study of CPU time and other machine-dependent features must analyze and explain each of the factors involved: run the experiments for different architectures, programming languages, ...
- Studies of CPU time versus input size alone are more often than not useless; we need experiments to reveal the dependence of CPU time on several parameters
- Experiments at that level of instantiation (see later) are difficult to reproduce

Intro to EA: Do's and Don'ts

- Benchmarks are not experiments; they are observations, much like observing the behavior of animals in the wild.
- They are a nice source of observational data, to be explained and complemented by experiments (under controlled conditions).
- They are also good (although far from complete!) to check the utility of our mathematical models; of course it's very nice when your predictions explain well “real-life” phenomena
- Good predictions for real-life data \implies understanding what is the “structure” of these instances \implies we can generate synthetic instances that mimick well real-life instances but that we can control at will

Intro to EA: Do's and Don'ts

- Benchmarks are not experiments; they are observations, much like observing the behavior of animals in the wild.
- They are a nice source of observational data, to be explained and complemented by experiments (under controlled conditions).
- They are also good (although far from complete!) to check the utility of our mathematical models; of course it's very nice when your predictions explain well “real-life” phenomena
- Good predictions for real-life data \implies understanding what is the “structure” of these instances \implies we can generate synthetic instances that mimick well real-life instances but that we can control at will

Intro to EA: Do's and Don'ts

- Benchmarks are not experiments; they are observations, much like observing the behavior of animals in the wild.
- They are a nice source of observational data, to be explained and complemented by experiments (under controlled conditions).
- They are also good (although far from complete!) to check the utility of our mathematical models; of course it's very nice when your predictions explain well “real-life” phenomena
- Good predictions for real-life data \implies understanding what is the “structure” of these instances \implies we can generate synthetic instances that mimick well real-life instances but that we can control at will

Intro to EA: Do's and Don'ts

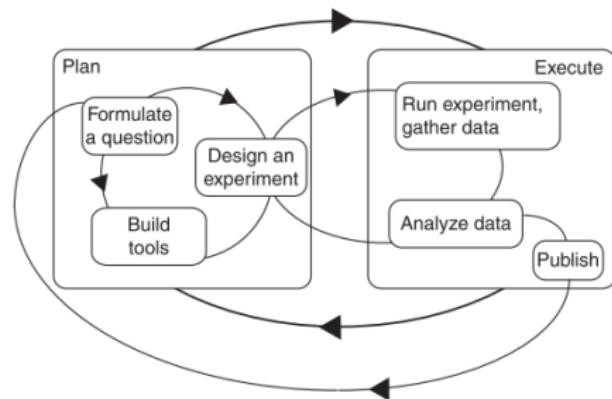
- Benchmarks are not experiments; they are observations, much like observing the behavior of animals in the wild.
- They are a nice source of observational data, to be explained and complemented by experiments (under controlled conditions).
- They are also good (although far from complete!) to check the utility of our mathematical models; of course it's very nice when your predictions explain well “real-life” phenomena
- Good predictions for real-life data \implies understanding what is the “structure” of these instances \implies we can generate synthetic instances that mimick well real-life instances but that we can control at will

Intro to EA

- Levels / scales of instantiation
 - Algorithmic paradigms, algorithmic schemes, metaheuristics
 - Algorithms (e.g., # of comparisons)
 - Source code/programs (e.g., # of instructions)
 - Processes (e.g., CPU time elapsed)
- Test subject vs. test program—**often different!**

Intro to EA

- The experimental process



Intro to EA

Experimental goals

- reproducibility / replication
- efficiency
- relevance / utility

Intro to EA

- Pilot studies vs. the *workhorse*
- Spurious results & artifacts: bugs, external factors, biased “randomness”, round-up & fixed-point arithmetic, . . .

Experimental Setup

- Input generation (\leftarrow reproducibility, efficiency, relevance!)
- Measures: prefer abstract, machine-independent quantities, conduct a separate study to relate abstract measures to machine-dependent measures (e.g., CPU time)
- Instrumentation: does measuring change results? where and when to measure?
- Carrying out the experiments: test subject vs. test program

Experimental Setup

- Input generation (\leftarrow reproducibility, efficiency, relevance!)
- Measures: prefer abstract, machine-independent quantities, conduct a separate study to relate abstract measures to machine-dependent measures (e.g., CPU time)
- Instrumentation: does measuring change results? where and when to measure?
- Carrying out the experiments: test subject vs. test program

Experimental Setup

- Input generation (\leftarrow reproducibility, efficiency, relevance!)
- Measures: prefer abstract, machine-independent quantities, conduct a separate study to relate abstract measures to machine-dependent measures (e.g., CPU time)
- Instrumentation: does measuring change results? where and when to measure?
- Carrying out the experiments: test subject vs. test program

Experimental Setup

- Input generation (\leftarrow reproducibility, efficiency, relevance!)
- Measures: prefer abstract, machine-independent quantities, conduct a separate study to relate abstract measures to machine-dependent measures (e.g., CPU time)
- Instrumentation: does measuring change results? where and when to measure?
- Carrying out the experiments: test subject vs. test program

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Longer trials to choose long-term asymptotic regime
 - Larger sample sizes to reduce variance
 - Use multiple runs to check consistency

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report "raw" data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don't use "lossy" performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, "narrow" indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report “raw” data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don’t use “lossy” performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, “narrow” indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report “raw” data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don’t use “lossy” performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, “narrow” indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report “raw” data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don’t use “lossy” performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, “narrow” indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report “raw” data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don’t use “lossy” performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, “narrow” indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report “raw” data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don’t use “lossy” performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, “narrow” indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

- Tune the code to generate inputs and to do the experiments
- Some items of advice for a good experimental study
 - More trials, larger samples
 - Larger inputs to show long-term/asymptotic regime
 - Do not summarize prematurely. Report “raw” data. Data analysis is a different task to be done by a different tool
 - Find a trade-off between reporting too much or not enough data points (efficiency, utility)
 - Don’t use “lossy” performance measures; you can use them in the data analysis & visualization phase. E.g., you are interested in $R = X/Y$, your experiment should report X and Y values.
 - Prefer focused, “narrow” indicators, e.g., counts for a specific kind of elementary operation vs. a count of executed instructions.

Experimental Setup

Apply variance-reduction techniques (VRT):

- Use the same inputs for variants of an algorithm to be compared (reduces variance and saves times!), measure $\Delta = X - X'$

$$\mathbb{V}[\Delta] = \mathbb{V}[X] + \mathbb{V}[X'] - 2\text{Cov}(X, X')$$

- Use a control variate X' , another measure with known expected value μ' and positively correlated with the measure of your interest X :

$$Y = X - c(X' - \mu), \mathbb{E}[Y] = \mathbb{E}[X]$$

Optimal value for $c = \text{Cov}(X, Y') / \mathbb{V}[X']$; estimate a good value for c with some small pilot study

Experimental Setup

Apply variance-reduction techniques (VRT):

- Conditional expectation VRT (conditional Monte Carlo):
split state generation from cost measurement and make
many (all?) measurements on the same state.
Very usual in empirical studies of data structures
 - ➊ build the random data structure
 - ➋ measure some unique parameter that conveys info about
the costs (IPL/EPL in search trees, path length/list length in
hash tables, ...)

Data Analysis

- Curve fitting: don't overfit, use theory to provide a reasonable guess / ground model
- Visualization of data: prefer plots to table to convey information, but don't put too much information; be watchful with misleading plots
- Testing hypotheses: use statistical tools, go beyond averages; collect as much data from the experiments as possible, to be processed later

Examples

- Deletions in binary search trees \implies Eppinger (1983)
- Percolation in random graphs \implies Sedgewick's slides for *Algorithms*
- Cache performance of quicksort \implies LaMarca & Ladner (1999)
- Some further examples \implies check the shared documentation folder *Examples*

Experimental Algorithmics

To learn more:



Catherine C. McGeoch
A Guide to Experimental Algorithmics.
Cambridge Univ. Press, 2012

Experimental Algorithmics

To learn more:

-  [J. L. Eppinger](#)
An Empirical Study of Insertion and Deletion in Binary Search Trees.
Comm. ACM 26(9):663-669, 1983.
-  [A. LaMarca and R. E. Ladner](#)
The Influence of Caches on the Performance of Sorting.
J. Algorithms 31(1):66–104, 1999.

Part I

Techniques

1 Experimental Algorithmics

2 Probabilistic Analysis of Algorithms

- The Continuous Master Theorem
 - Example #1: Quicksort
 - Example #2: Quickselect
- Probabilistic Tools

3 Amortized Analysis

Part I

Techniques

- 1 Experimental Algorithmics
- 2 Probabilistic Analysis of Algorithms
 - The Continuous Master Theorem
 - Example #1: Quicksort
 - Example #2: Quickselect
 - Probabilistic Tools
- 3 Amortized Analysis

The Continuous Master Theorem

CMT considers divide-and-conquer recurrences of the following type:

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j, \quad n \geq n_0$$

for some positive integer n_0 , a function t_n , called the *toll function*, and a sequence of weights $\omega_{n,j} \geq 0$. The weights must satisfy two conditions:

- ① $W_n = \sum_{0 \leq j < n} \omega_{n,j} \geq 1$ (at least one recursive call).
- ② $Z_n = \sum_{0 \leq j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1$ (the size of the subinstances is a fraction of the size of the original instance).

The next step is to find a *shape function* $\omega(z)$, a continuous function approximating the discrete weights $\omega_{n,j}$.

The Continuous Master Theorem

Definition

Given the sequence of weights $\omega_{n,j}$, $\omega(z)$ is a shape function for that set of weights if

① $\int_0^1 \omega(z) dz \geq 1$

② there exists a constant $\rho > 0$ such that

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z) dz \right| = \mathcal{O}(n^{-\rho})$$

A simple trick that works very often, to obtain a convenient shape function is to substitute j by $z \cdot n$ in $\omega_{n,j}$, multiply by n and take the limit for $n \rightarrow \infty$.

$$\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n}$$

The Continuous Master Theorem

The extension of discrete functions to functions in the real domain is immediate, e.g., $j^2 \rightarrow z^2$. For binomial numbers one might use the approximation

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

The continuation of factorials to the real numbers is given by Euler's Gamma function $\Gamma(z)$ and that of harmonic numbers by Ψ function: $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$.

For instance, in quicksort's recurrence all Wright are equal: $\omega_{n,j} = \frac{2}{n}$. Hence a simple valid shape function is $\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n} = 2$.

The Continuous Master Theorem

Theorem (Roura, 1997)

Let F_n satisfy the recurrence

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j,$$

with $t_n = \Theta(n^a(\log n)^b)$, for some constants $a \geq 0$ and $b > -1$, and let $\omega(z)$ be a shape function for the weights $\omega_{n,j}$. Let $\mathcal{H} = 1 - \int_0^1 \omega(z)z^a dz$ and $\mathcal{H}' = -(b+1)\int_0^1 \omega(z)z^a \ln z dz$. Then

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{if } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{if } \mathcal{H} = 0 \text{ and } \mathcal{H}' \neq 0, \\ \Theta(n^\alpha) & \text{if } \mathcal{H} < 0, \end{cases}$$

where $x = \alpha$ is the unique non-negative solution of the equation

$$1 - \int_0^1 \omega(z)z^x dz = 0.$$

Example #1: QuickSort

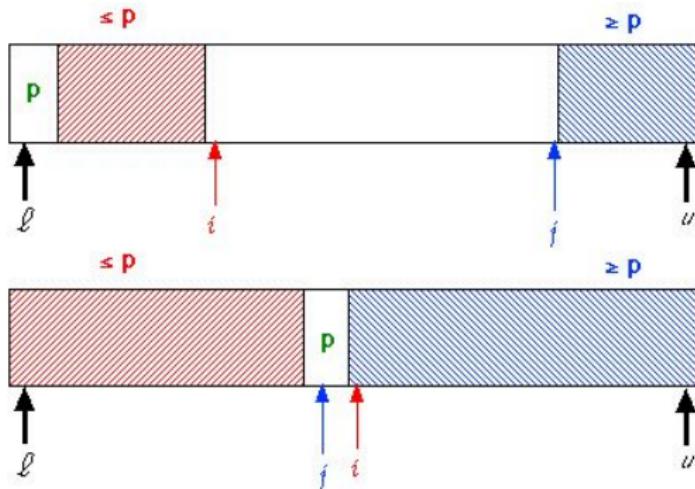


C.A.R. Hoare (1934–)

QUICKSORT (Hoare, 1962) is a sorting algorithm using the divide-and-conquer principle too, but contrary to the previous examples, it does not guarantee that the size of each subinstance will be a fraction of the size of the original given instance.

QuickSort

The basis of *Quicksort* is the procedure PARTITION: given an element p , called the **pivot**, the (sub)array is rearranged as shown in the picture below.



QuickSort

PARTITION puts the pivot in its final place. Hence it suffices to recursively sort the subarrays to its left and to its right. While *divide* is simple and *combine* does most of the work in MERGESORT, in QUICKSORT it happens just the contrary: *divide* consists in partitioning the array, and we have nothing to do to *combine* the solutions obtained from the recursive calls. The QUICKSORT algorithm for a subarray $A[\ell..u]$ is

```
procedure QUICKSORT( $A, \ell, u$ )
Ensure: Sorts subarray  $A[\ell..u]$ 
  if  $u - \ell + 1 \leq M$  then
    use a simple sorting method, e.g., insertion sort
  else
    PARTITION( $A, \ell, u, k$ )
     $\triangleright A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$ 
    QUICKSORT( $(A, \ell, k - 1)$ )
    QUICKSORT( $(A, k + 1, u)$ )
  end if
end procedure
```

QuickSort

Instead of using a simple sorting method each time we reach a subarray with M or less elements, we can postpone the sorting of small subarrays until the end:

`QUICKSORT(A , 1, $A.\text{SIZE}()$)`

`INSERTSORT(A , 1, $A.\text{SIZE}()$)`

Since the array is almost sorted upon completion of the call to `QUICKSORT`, the last step using `INSERTSORT` needs only linear time ($\Theta(n)$), where $n = A.\text{SIZE}()$.

The typical optimal choice for the cut-off value M is around 20 to 25.

QuickSort

```
template <class T>
void partition(vector<T>& v, int l, int u, int& k);

template <class T>
void quicksort(vector<T>& v, int l, int u) {
    if (u - l + 1 > M) {
        int k;
        partition(v, l, u, k);
        quicksort(v, l, k-1);
        quicksort(v, k+1, u);
    }
}

template <class T>
void quicksort(vector<T>& v) {
    quicksort(v, 0, v.size()-1);
    insertion_sort(v, 0, v.size()-1);
}
```

QuickSort

There are many ways to do the partition; not them all are equally good. Some issues, like repeated elements, have to be dealt with carefully. Bentley & McIlroy (1993) discuss a very efficient partition procedure, which works seamlessly even in the presence of repeated elements. Here, we will examine a basic algorithm, which is reasonably efficient.

We will keep two indices i and j such that $A[\ell + 1..i - 1]$ contains elements less than or equal to the pivot p , and $A[j + 1..u]$ contains elements greater than or equal to the pivot p . The two indices scan the subarray locations, i from left to right, j from right to left, until $A[i] > p$ and $A[j] < p$, or until they cross ($i = j + 1$).

QuickSort

procedure PARTITION(A, ℓ, u, k)

Require: $\ell \leq u$

Ensure: $A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$

$i := \ell + 1; j := u; p := A[\ell]$

while $i < j + 1$ **do**

while $i < j + 1 \wedge A[i] \leq p$ **do**

$i := i + 1$

end while

while $i < j + 1 \wedge A[j] \geq p$ **do**

$j := j - 1$

end while

if $i < j + 1$ **then**

$A[i] := A[j]$

$i := i + 1; j := j - 1$

end if

end while

$A[\ell] := A[j]; k := j$

end procedure

QuickSort

```
template <class T>
void partition(vector<T>& v, int l, int u, int& k) {
    int i = l+1;
    int j = u;
    T pv = v[i]; // simple choice for pivot
    while (i < j+1) {
        while (i < j+1 and v[i] <= pv) ++i;
        while (i < j+1 and pv <= v[j]) --j;
        if (i < j + 1) {
            swap(v[i], v[j]);
            ++i; --j;
        }
    };
    swap(v[l], v[j]);
    k = j;
}
```

The Cost of QuickSort

The worst-case cost of QUICKSORT is $\Theta(n^2)$, hence not very attractive. But it only occurs if all or most recursive call one of the subarrays contains very few elements and the other contains almost all. That would happen if we systematically choose the first element of the current subarray as the pivot and the array is already sorted!

The cost of the partition is $\Theta(n)$ and we would have then

$$\begin{aligned} Q(n) &= \Theta(n) + Q(n - 1) + Q(0) \\ &= \Theta(n) + Q(n - 1) = \Theta(n) + \Theta(n - 1) + Q(n - 2) \\ &= \dots = \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{0 \leq i \leq n} i\right) \\ &= \Theta(n^2). \end{aligned}$$

The Cost of QuickSort

However, on average, there will be a fraction of the elements that are less than the pivot (and will be to its left) and a fraction of elements that are greater than the pivot (and will be to its right). It is for this reason that `QUICKSORT` belongs to the family of divide-and-conquer algorithms, and indeed it has a good average-case complexity.

The Cost of QuickSort

To analyze the performance of QUICKSORT it only matters the relative order of the elements to be sorted, hence we can safely assume that the input is a permutation of the elements 1 to n . Furthermore, we can concentrate in the number of comparisons between the elements since the total cost will be proportional to that number of comparisons.

The Cost of QuickSort

Let us assume that all $n!$ possible input permutations are equally likely and let q_n be the expected number of comparisons to sort the n elements. Then

$$\begin{aligned} q_n &= \sum_{1 \leq j \leq n} \mathbb{E}[\# \text{ compar.} \mid \text{pivot is the } j\text{-th}] \times \Pr\{\text{pivot is the } j\text{-th}\} \\ &= \sum_{1 \leq j \leq n} (n - 1 + q_{j-1} + q_{n-j}) \times \frac{1}{n} \\ &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq j \leq n} (q_{j-1} + q_{n-j}) \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq j < n} q_j \end{aligned}$$

To solve this recurrence the *continuous master theorem* (CMT).

Solving QuickSort's Recurrence

We apply CMT to quicksort's recurrence with the set of weights $\omega_{n,j} = 2/n$ and toll function $t_n = n - 1$. As we have already seen, we can take $\omega(z) = 2$, and the CMT applies with $a = 1$ and $b = 0$. All necessary conditions to apply CMT are met. Then we compute

$$\mathcal{H} = 1 - \int_0^1 2z \, dz = 1 - z^2 \Big|_{z=0}^{z=1} = 0,$$

hence we will have to apply CMT's second case and compute

$$\mathcal{H}' = - \int_0^1 2z \ln z \, dz = \frac{z^2}{2} - z^2 \ln z \Big|_{z=0}^{z=1} = \frac{1}{2}.$$

Finally,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1.386 \dots n \log_2 n + o(n \log n). \end{aligned}$$

Example #2: QuickSelect

The **selection** problem is to find the j -th smallest element in a given set of n elements. More specifically, given an array $A[1..n]$ of size $n > 0$ and a **rank** j , $1 \leq j \leq n$, the selection problem is to find the j -th element of A if it were in ascending order.

For $j = 1$ we want to find the minimum, for $j = n$ we want to find the maximum, for $j = \lceil n/2 \rceil$ we are looking for the median, etc.

QuickSelect

The problem can be trivially but inefficiently (because it implies doing much more work than needed!) solved with cost $\Theta(n \log n)$ sorting the array. Another solution keeps an unsorted table of the j smallest elements seen so far while scanning the array from left to right; it has cost $\Theta(j \cdot n)$, and using clever data structures the cost can be improved to $\Theta(n \log j)$. This is not a real improvement with respect the first trivial solution if $j = \Theta(n)$.

QUICKSELECT (Hoare, 1962), also known as FIND and as *one-sided* QUICKSORT, is a variant of QUICKSORT adapted to select the j -th smallest element out of n .

QuickSelect

Assume we partition the subarray $A[\ell..u]$, that contains the elements of ranks ℓ to u , $\ell \leq j \leq u$, with respect some pivot p . Once the partition finishes, suppose that the pivot ends at position k .

Then $A[\ell..k - 1]$ contains the elements of ranks ℓ to $(k - 1)$ in A and $A[k + 1..u]$ contains the elements of ranks $(k + 1)$ to u . If $j = k$ we are done since we have found the sought element. If $j < k$ then we need to recursively continue in the left subarray $A[\ell..k - 1]$, whereas if $j > k$ then the sought element must be located in the right subarray $A[k + 1..u]$.

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

9	5	10	12	3	1	11	15	7	2	8	13	6	4	14
---	---	----	----	---	---	----	----	---	---	---	----	---	---	----

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

9	5	10	12	3	1	11	15	7	2	8	13	6	4	14
---	---	----	----	---	---	----	----	---	---	---	----	---	---	----

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

7	5	4	6	3	1	8	2	9	15	11	13	12	10	14
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

pivot ends at position $k = 9 > j$

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

7	5	4	6	3	1	8	2	9	15	11	13	12	10	14
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

1	5	4	2	3	6	8	7	9	15	11	13	12	10	14
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

pivot ends at position $k = 6 > j$

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

1	5	4	2	3	6	8	7	9	15	11	13	12	10	14
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

2	3	1	4	5	6	8	7	9	15	11	13	12	10	14
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

pivot ends at position $k = 4 = j \Rightarrow \text{DONE!}$

QuickSelect

```
procedure QUICKSELECT( $A, \ell, j, u$ )
```

Ensure: Returns the $(j + 1 - \ell)$ -th smallest element in $A[\ell..u]$,

$\ell \leq j \leq u$

```
if  $\ell = u$  then
```

```
    return  $A[\ell]$ 
```

```
end if
```

```
PARTITION( $A, \ell, u, k$ )
```

```
if  $j = k$  then
```

```
    return  $A[k]$ 
```

```
end if
```

```
if  $j < k$  then
```

```
    return QUICKSELECT( $A, \ell, j, k - 1$ )
```

```
else
```

```
    return QUICKSELECT( $A, k + 1, j, u$ )
```

```
end if
```

```
end procedure
```

QuickSelect

Our implementation of QUICKSELECT uses **tail recursion**, it is hence straightforward to derive an efficient iterative solution that needs no auxiliary memory space.

In the worst-case, the cost of QUICKSELECT is $\Theta(n^2)$. However, its average cost is $\Theta(n)$, with the proportionality constant depending on the ratio j/n . Knuth (1971) proved that $C_n^{(j)}$, the expected number of comparisons to find the smallest j -th element among n is:

$$\begin{aligned} C_n^{(j)} &= 2((n+1)H_n - (n+3-j)H_{n+1-j} \\ &\quad - (j+2)H_j + n+3) \end{aligned}$$

The maximum average cost corresponds to finding the median ($j = \lfloor n/2 \rfloor$); then we have

$$C_n^{(\lfloor n/2 \rfloor)} = 2(\ln 2 + 1)n + o(n).$$

QuickSelect

Let us now consider the analysis of the expected cost C_n when j takes any value between 1 and n with identical probability.

Then

$$C_n = n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \mathbb{E}[\text{remaining number of comp.} \mid \text{pivot is the } k\text{-th element}] ,$$

as the pivot will be the k -th smallest element with probability $1/n$ for all k , $1 \leq k \leq n$.

QuickSelect

The probability that $j = k$ is $1/n$, then no more comparisons are needed since we would be done. The probability that $j < k$ is $(k - 1)/n$, then we will have to make C_{k-1} comparisons.

Similarly, with probability $(n - k)/n$ we have $j > k$ and we will then make C_{n-k} comparisons. Thus

$$\begin{aligned}C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\&= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq k < n} \frac{k}{n} C_k.\end{aligned}$$

Applying the CMT with the shape function

$$\lim_{n \rightarrow \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

we obtain $\mathcal{H} = 1 - \int_0^1 2z^2 dz = 1/3 > 0$ and $C_n = 3n + o(n)$.

Part I

Techniques

- 1 Experimental Algorithmics
- 2 Probabilistic Analysis of Algorithms
 - The Continuous Master Theorem
 - Example #1: Quicksort
 - Example #2: Quickselect
 - Probabilistic Tools
- 3 Amortized Analysis

Linearity of Expectations

For any random variables X and Y , independent or not,

$$\mathbb{E}[aX + bY] = a \mathbb{E}[X] + b \mathbb{E}[Y].$$

If X and Y are independent then $\mathbb{V}[X + Y] = \mathbb{V}[X] + \mathbb{V}[Y]$.

Indicator variables

It is often useful to introduce **indicator** random variables

$X_i = \mathbb{I}_{A_i}$ such that $X_i = 1$ if the event A_i is true and $X_i = 0$ if the event A_i is false. Let $p_i = \mathbb{P}[\text{event } A_i \text{ happens}]$. Then the X_i are Bernoulli random variables with

$$\mathbb{E}[X_i] = \mathbb{P}[X_i = 1] = p_i$$

In many cases we can express or bound a random variable X as a linear combination of indicator random variables and then exploit linearity of expectations to derive $\mathbb{E}[X]$.

Union Bound

For any sequence (finite or denumerable) of events $\{A_i\}_{i \geq 0}$

$$\mathbb{P}\left[\bigcup_{i \geq 0} A_i\right] \leq \sum_{i \geq 0} \mathbb{P}[A_i]$$

Markov's Inequality

Theorem

Let X be a positive random variable. For any $a > 0$

$$\mathbb{P}[X > a] \leq \frac{\mathbb{E}[X]}{a}$$

Markov's Inequality

Proof.

Let A be the event $X > a$. Then for the indicator random variable \mathbb{I}_A we have

$$\mathbb{P}[X > a] = \mathbb{P}[\mathbb{I}_A = 1] = \mathbb{E}[\mathbb{I}_A],$$

but $a \cdot \mathbb{I}_A < X$, therefore $a \mathbb{E}[\mathbb{I}_A] < \mathbb{E}[X]$ and

$$\mathbb{P}[X > a] < \frac{\mathbb{E}[X]}{a}.$$



Markov's Inequality

Example

Suppose we throw a fair coin n times. Let H_n denote number of heads in the n throws. We have $\mathbb{E}[H_n] = n/2$. Using Markov's inequality

$$\mathbb{P}[H_n > 3n/4] \leq \frac{n/2}{3n/4} = \frac{2}{3}.$$

In general,

$$\mathbb{P}[X > c \cdot \mathbb{E}[X]] \leq \frac{\mathbb{E}[X]}{c \mathbb{E}[X]} = \frac{1}{c}.$$

Chebyshev's Inequality

Theorem

Let X be a positive random variable.

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq a] \leq \frac{\mathbb{V}[X]}{a^2}$$

Corollary

$$\mathbb{P}[|X - \mathbb{E}[X]| \geq c \cdot \sigma_X] \leq \frac{1}{c^2},$$

with $\sigma_X = \sqrt{\mathbb{V}[X]}$, the standard deviation.

Chebyshev's Inequality

Proof.

We have

$$\begin{aligned}\mathbb{P}[|X - \mathbb{E}[X]| \geq a] &= \mathbb{P}[(X - \mathbb{E}[X])^2 \geq a^2] \\ &\leq \frac{E(X - \mathbb{E}[X])^2}{a^2} = \frac{\mathbb{V}[X]}{a^2}\end{aligned}$$



Chebyshev's Inequality

Example

Again H_n = the number of heads in n throws of a fair coin.

Since $H_n \sim \text{Binomial}(n, 1/2)$, $\mathbb{E}[H_n] = n/2$ and $\mathbb{V}[H_n] = n/4$.

Using Chebyshev's inequality

$$\mathbb{P}[H_n > 3n/4] \leq \mathbb{P}\left[|X - \frac{n}{2}| \geq \frac{n}{4}\right] \leq \frac{\mathbb{V}[H_n]}{(n/4)^2} = \frac{4}{n}.$$

Chebyshev's Inequality

Example

The expected number of comparisons $\mathbb{E}[q_n]$ in standard quicksort is $2n \ln n + o(n \log n)$. It can be shown that

$\mathbb{V}[q_n] = \left(7 - \frac{2\pi^2}{3}\right) n^2 + o(n^2)$. Hence, the probability that we deviate more than c times from the expected value goes to 0 as $1/\log n$:

$$\begin{aligned}\mathbb{P}[|q_n - \mathbb{E}[q_n]| \geq c \mathbb{E}[q_n]] &\leq \frac{\left(7 - \frac{2\pi^2}{3}\right) n^2 + o(n^2)}{2c^2 n^2 \ln^2 n + o(n^2 \log^2 n)} \\ &= \frac{\left(7 - \frac{2\pi^2}{3}\right)}{2c \ln n} + o(1/\log n)\end{aligned}$$

Jensen's Inequality

Theorem

If f is a convex function then

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$$

Example

For any random variable X , $\mathbb{E}[X^2] \geq (\mathbb{E}[X])^2$, since $f(x) = x^2$ is convex.

Chernoff Bounds

Theorem

Let $\{X_i\}_{i=0}^n$ be **independent Bernoulli trials**, with $\mathbb{P}[X_i = 1] = p_i$. Then, if $X = \sum_{i=1}^n X_i$, and $\mu = \mathbb{E}[X]$, we have

- ① $\mathbb{P}[X \leq (1 - \delta)\mu] \leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^\mu$, for $\delta \in (0, 1)$.
- ② $\mathbb{P}[X \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu$ for any $\delta > 0$.

Chernoff Bounds

Corollary (Corollary 1)

Let $\{X_i\}_{i=0}^n$ be **independent** Bernouilli trials, with $\mathbb{P}[X_i = 1] = p_i$. Then if $X = \sum_{i=1}^n X_i$, and $\mu = \mathbb{E}[X]$, we have

- ① $\mathbb{P}[X \leq (1 - \delta)\mu] \leq e^{-\mu\delta^2/2}$, for $\delta \in (0, 1)$.
- ② $\mathbb{P}[X \geq (1 + \delta)\mu] \leq e^{-\mu\delta^2/3}$, for $\delta \in (0, 1]$.

Corollary (Corollary 2)

Let $\{X_i\}_{i=0}^n$ be **independent** Bernouilli trials, with $\mathbb{P}[X_i = 1] = p_i$. Then if $X = \sum_{i=1}^n X_i$, $\mu = \mathbb{E}[X]$ and $\delta \in (0, 1)$, we have

$$\mathbb{P}[|X - \mu| \geq \delta\mu] \leq 2e^{-\mu\delta^2/3}.$$

Chernoff Bounds

Back to an old example: We flip n times a fair coin, we wish an upper bound on the probability of having at least $\frac{3n}{4}$ heads.

Recall Let $H_n \sim \text{Binomial}(n, 1/2)$, then,

$$\mu = \mathbb{E}[H_n] = n/2, \mathbb{V}[H_n] = n/4.$$

We want to bound $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right]$.

- **Markov:** $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right] \leq \frac{\mu}{3n/4} = 2/3.$
- **Chebyshev:** $\mathbb{P}\left[H_n \geq \frac{3n}{4}\right] \leq \mathbb{P}\left[|H_n - \frac{n}{2}| \geq \frac{n}{4}\right] \leq \frac{\mathbb{V}[H_n]}{(n/4)^2} = \frac{4}{n}.$
- **Chernoff:** Using Cor. 1.2,

$$\begin{aligned}\mathbb{P}\left[H_n \geq \frac{3n}{4}\right] &= \mathbb{P}\left[H_n \geq (1 + \delta)\frac{n}{2}\right] \Rightarrow (1 + \delta)\frac{3}{2} \Rightarrow \delta = \frac{1}{2} \\ &\Rightarrow \mathbb{P}\left[H_n \geq \frac{3n}{4}\right] \leq e^{-\mu\delta^2/3} = e^{-\frac{n}{24}}\end{aligned}$$

Example

- If $n = 100$, Cheb. = 0.04, Chernoff = 0.0155
- If $n = 10^6$, Cheb. = 4×10^{-6} , Chernoff = 2.492×10^{-18095}

Part I

Techniques

1 Experimental Algorithmics

2 Probabilistic Analysis of Algorithms

- The Continuous Master Theorem
 - Example #1: Quicksort
 - Example #2: Quickselect
- Probabilistic Tools

3 Amortized Analysis

Amortized Analysis

In **amortized analysis** we find the (worst/best/average) cost C_n of a sequence of n operations; the **amortized cost** per operation is

$$a_n = \frac{C_n}{n}$$

Sometimes we compute the cost $C(n_1, \dots, n_k)$ of a sequence involving n_1 operations of type 1, n_2 operations of type 2, ... The amortized cost is then

$$A(n_1, \dots, n_k) = \frac{C(n_1, \dots, n_k)}{n_1 + \dots + n_k}$$

Amortized Analysis

Amortized cost is interesting when we consider that a sequence of operations must be performed, and some are expensive, but some are cheap; bounding the total cost by n times the cost of the most expensive operation is overly pessimistic.

A first example: Binary counter

Suppose we have a counter that we initialize to 0 and increment it ($\bmod 2$) n times. The counter has k bits. How many bit flips are needed?

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf>)

Theorem

Starting from 0, a sequence of n increments makes $\mathcal{O}(nk)$ bit flips.

Proof

Any increment flips $\mathcal{O}(k)$ bits.



A first example: Binary counter

Suppose we have a counter that we initialize to 0 and increment it ($\bmod 2$) n times. The counter has k bits. How many bit flips are needed?

Counter value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf>)

Theorem

Starting from 0, a sequence of n increments makes $\mathcal{O}(nk)$ bit flips.

Proof

Any increment flips $\mathcal{O}(k)$ bits.



Aggregate method

- Determine (an upper bound on) the number $N(c)$ of operations of cost c in the sequence. Then the cost of the sequence is $\leq \sum_{c>0} c \cdot N(c)$.
- An alternative is to count how many operations $N'(c)$ have cost $\geq c$, then the cost of the sequence is $\leq \sum_{c>0} N'(c)$.

Aggregate method

In the binary counter problem, we observe that bit 0 flips n times, bit 1 flips $\lfloor n/2 \rfloor$, bit 2 flips $\lfloor n/4 \rfloor$ times, ...

Theorem

Starting from 0, a sequence of n increments makes $\Theta(n)$ bit flips.

Proof

Each increment flips at least 1 bit, thus we make at least $\Omega(n)$ flips. On the other hand, the total cost is also $\mathcal{O}(n)$. Indeed

$$\sum_{j=0}^{k-1} \left\lfloor \frac{n}{2^j} \right\rfloor \leq n \sum_{j=0}^{k-1} \frac{1}{2^j} < n \sum_{j=0}^{\infty} \frac{1}{2^j} = n \frac{1}{1 - (1/2)} = 2n.$$



Accounting method (banker's viewpoint)

We associate “charges” to different operations, these charges may be smaller or larger than the actual cost.

- When the charge or **amortized cost** \hat{c}_i of an operation is larger than the actual cost c_i then the difference is seen as credits that we store in the data structure to pay for future operations.
- When the amortized cost \hat{c}_i is smaller than c_i the difference must be covered from the credits stored in the data structure.
- The initial data structure D_0 has 0 credits.

Invariant: For all ℓ ,

$$\sum_{i=1}^{\ell} (\hat{c}_i - c_i) \geq 0,$$

that is, at all moments, there must be a positive number of credits in the data structure.

Accounting method (banker's viewpoint)

Theorem

The total cost of processing a sequence of n operations starting with D_0 is bounded by the sum of amortized costs.

Proof

$$\text{Invariant} \implies \sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i \quad \leftarrow \text{total cost.}$$



Accounting method (banker's viewpoint)

In the binary counter problem, we charge 2 credits every time we flip a bit from 0 to 1, we pay 1 credit every time we flip a bit from 1 to 0.

We consider that each time we flip a bit from 0 to 1 we store one credit in the data structure and use the other credit to pay the flip. When the bit is flip from 1 to 0, we use the stored credit for that. Thus 1-bits all store 1 credit each, whereas 0-bits do not store credits.



Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf>)

Accounting method (banker's viewpoint)

In the binary counter problem, we charge 2 credits every time we flip a bit from 0 to 1, we pay 1 credit every time we flip a bit from 1 to 0.

We consider that each time we flip a bit from 0 to 1 we store one credit in the data structure and use the other credit to pay the flip. When the bit is flip from 1 to 0, we use the stored credit for that. Thus 1-bits all store 1 credit each, whereas 0-bits do not store credits.



Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf>)

Accounting method (banker's viewpoint)

In the binary counter problem, we charge 2 credits every time we flip a bit from 0 to 1, we pay 1 credit every time we flip a bit from 1 to 0.

We consider that each time we flip a bit from 0 to 1 we store one credit in the data structure and use the other credit to pay the flip. When the bit is flip from 1 to 0, we use the stored credit for that. Thus 1-bits all store 1 credit each, whereas 0-bits do not store credits.



Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf>)

Accounting method (banker's viewpoint)

Theorem

Starting from 0, a sequence of n increments makes $\Theta(n)$ bit flips.

Proof

Every increment flips at least one bit, thus $\sum_i c_i \geq n$. On the other hand every increment flips a 0-bit to a 1-bit once (the rightmost 0 in the counter before the increment is the only 0-bit flipped). Hence $\hat{c}_i = 2$ because all the other flips during the i -th increment are from 1-bits to 0-bits, and their amortized cost is 0. Thus

$$\sum_i \hat{c}_i = 2n \geq \sum_i c_i.$$

As the number of credits per bit is ≥ 0 then the number of credits stored at the counter are ≥ 0 at all times, that is, the invariant is preserved. □

Potential method (physicist's viewpoint)

In the **potential method** we define a **potential** function Φ that associates a non-negative real to every possible configuration D of the data structure.

- ① $\Phi(D) \geq 0$ for all possible configurations D of the data structure.
- ② $\Phi(D_0) = 0 \leftarrow$ the potential of the initial configuration is 0
- D_i = configuration of the data structure after i -th operation
- c_i = actual cost of the i -th operation in the sequence
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) =$
amortized cost of the i -th operation, it is the actual cost c_i of the operation plus the change in potential
 $\Delta\Phi_i = \Phi(D_i) - \Phi(D_{i-1}).$

Potential method (physicist's viewpoint)

Theorem

The total cost of processing a sequence of n operations starting with D_0 is bounded by the sum of amortized costs.

Proof

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n c_i + \Delta\Phi_i \\&= \sum_{i=1}^n c_i + \sum_{i=1}^n (\Phi(D_i) - \Phi(D_{i-1})) \\&= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) = \sum_{i=1}^n c_i + \Phi(D_n) \geq \sum_{i=1}^n c_i,\end{aligned}$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.



Potential method (physicist's viewpoint)

For the binary counter problem, we will take

$\Phi(D)$ = number of 1-bits in the binary counter D . Notice that $\Phi(D_0) = 0$ and $\Phi(D) \geq 0$ for all D .

- The actual cost c_i of the i -th increment is $\leq 1 + p$, where p , $0 \leq p \leq k$ is the position of the rightmost 0-bit. We flip the p 1's to the right of the rightmost 0-bit, then the rightmost 0-bit (except when the counter is all 1's and we reset it, then the cost is p).
- The change in potential is $\leq 1 - p$ because we add one 1-bit (flipping the rightmost 0-bit to a 1-bit, except if $p = k$) and we flip p 1-bits to 0-bits, those to the right of the rightmost 0-bit. Hence

$$\hat{c}_i = c_i + \Delta\Phi_i \leq 1 + p + (1 - p) = 2,$$

$$\implies \sum_i \hat{c}_i \leq 2n.$$

Stacks with multi-pop

Example

Suppose we have a **stack** that supports:

- **PUSH(x)**
- **POP():** pops the top of the stack and returns it, stack must be non-empty
- **MPOP(k):** pops k items, the stack must contain at least k items

The cost of **PUSH** and **POP** is $\mathcal{O}(1)$ and the cost of **MPOP(k)** is $\Theta(k) = \mathcal{O}(n)$ (n = size of the stack), but saying that the worst-case cost of a sequence of N stack operations is $\mathcal{O}(N^2)$ is too pessimistic!

Stacks with multi-pop

Example

Accounting: Assign 2 credits to each PUSH. One is used to do the operation and the other credit to pop (with pop or multi-pop) the element at a later time. The total number of credits in the stack = size of the stack.

- $\hat{c}_{\text{PUSH}} = 2$.
- $\hat{c}_{\text{POP}} = \hat{c}_{\text{MPOP}} = 0$.

$$\implies 2N \geq \sum_{i=1}^N \hat{c}_i \geq \sum_{i=1}^N c_i$$

Stacks with multi-pop

Example

Potential: $\Phi(S) = \text{size}(S)$. Then: (1) $\Phi(S_0) = 0$, (2) $\Phi(S) \geq 0$ for all stacks S .

- $\hat{c}_{\text{PUSH}} = 1 + \Delta\Phi_i = 2$.
- $\hat{c}_{\text{POP}} = 1 + \Delta\Phi_i = 1 + (-1) = 0$.
- $\hat{c}_{\text{MPOP}} = k + \Delta\Phi_i = k + (-k) = 0 \quad \leftarrow |S_{i-1}| \geq k$

$$\implies 2N \geq \sum_{i=1}^N \hat{c}_i \geq \sum_{i=1}^N c_i$$

Dynamic arrays

Example

We often use **dynamic arrays** (a.k.a. **vector**s in C++), the array dynamically grows as we add items (using `v.push_back(x)`, say).

A common way to implement dynamic arrays is to allocate an array of some size from dynamic memory; in a given moment, we use only part of the array, we have then

- **size**: number of elements in the array
- **capacity**: number of memory cells in the array,
 $\text{size} \leq \text{capacity}$

Dynamic arrays

Example

When a new element has to be added and $n = \text{size} = \text{capacity}$ a new array with double capacity is allocated from dynamic memory, the contents of the old array copied into the new and the old array freed back to dynamic memory, with total cost $\Theta(n)$. The program sets the array name (a pointer) to point to the new array instead of pointing to the old.

This procedure is called **resizing**, and it implies that a single `push_back` can be very costly if it has to invoke a resizing to accomplish its task.

Dynamic arrays

Example

How much does it cost to fill a dynamic array using n push_back's?

Aggregate:

- The cost c_i of the i -th push_back is $\Theta(1)$ except if $i = 2^k + 1$ for some k , $0 \leq k \leq \log_2(n - 1)$.
- When $i = 2^k + 1$, it triggers a resizing with cost $\Theta(i)$.

Total cost:

$$\begin{aligned}\sum_{i=1}^n c_i &= \Theta\left(\sum_{i:i \neq 2^k+1} 1 + \sum_{i:i=2^k+1} i\right) \\ &= n - \Theta(\log n) + \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} (2^k + 1) \\ &\leq n - \Theta(\log n) + \Theta(\log n) + (2^{\log_2(n-1)+1} - 1) = \Theta(n).\end{aligned}$$

Dynamic arrays

Example

How much does it cost to fill a dynamic array using n `push_back`'s?

Accounting:

- Charge 3 credits to the assignment $v[j] := x$ in which we add x to the first unused array slot j ; every `push_back` does it, sometimes a resizing is also needed. Use 1 credit for the assignment, and store the remaining 2 credits in slot j .
- When resizing an array v of size n to an array v' with capacity $2n$, each $j \in v[n/2..n-1]$ stores 2 credits; use one credit for the creation of $v'[j+n]$, the copying $v'[j] := v[j]$ and the destruction of $v[j]$; and use the other credit for the creation of $v'[j+n/2]$, the copying of $v[j-n/2]$ to $v'[j-n/2]$ and the destruction of $v[j-n/2]$

Dynamic arrays

1	2	3	4
---	---	---	---



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



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----



Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/AmortizedAnalysis.pdf>)

Dynamic arrays

Example

How much does it cost to fill a dynamic array using n `push_back`'s?

Accounting: The total number of credits in the dynamic array v is $2 \times (\text{size}(v)/2) = \text{size}(v)$, therefore always ≥ 0 .

$$\sum_{i=1}^n \hat{c}_i = 3n \geq \sum_{i=1}^n c_i.$$

Dynamic arrays

Example

Instead of 3 credits for the assignment $v[j] := x$ we might charge some other constant quantity $c \geq 3$ so that we use 1 credit for the assignment $v[j] := x$ proper, and we store $c - 1$ credits at every used slot j in the upper half of v ; these $c - 1$ credits will be used to pay for the copying of $v[j]$ and of $v[j - n/2]$, but also the creation of unused slots $v'[j + n/2]$ and $v'[j + n]$ in the new array and the destruction of $v[j - n/2]$ and $v[j]$ in the old array, if such construction/destruction costs need to be taken into account.

Dynamic arrays

Example

How much does it cost to fill a dynamic array using n `push_back`'s?

Potential:

- When there is no resizing: $c_i = 1$.
- When there is resizing: $c_i = 1 + k \cdot \text{capacity}(v_i)$, for some constant k , v_i is the dynamic array after the i -th `push_back`
- $\Phi(v) = 2k(2 \cdot \text{size}(v) - \text{capacity}(v) + 1)$

N.B. We will take $k = 1/2$ to simplify the calculations

Dynamic arrays

Example

How much does it cost to fill a dynamic array using n `push_back`'s?

Potential:

- When there is no resizing: $\text{capacity}(v_i) = \text{capacity}(v_{i-1})$,

$$\begin{aligned}\Phi(v_i) - \Phi(v_{i-1}) &= 2(\text{size}(v_{i-1}) + 1) - \text{capacity}(v_{i-1}) + 1 \\ &\quad - \{2 \cdot \text{size}(v_{i-1}) - \text{capacity}(v_{i-1}) + 1\} \\ &= 2,\end{aligned}$$

and $\hat{c}_i = c_i + \Delta\Phi_i = 3$.

Dynamic arrays

Example

How much does it cost to fill a dynamic array using n `push_back`'s?

Potential:

- When there is resizing:

$$\text{capacity}(v_i) = 2 \cdot \text{capacity}(v_{i-1}) = 2 \cdot \text{size}(v_{i-1}),$$

$$\begin{aligned}\Phi(v_i) - \Phi(v_{i-1}) &= 2(\text{size}(v_{i-1}) + 1) - 2\text{capacity}(v_{i-1}) + 1 \\ &\quad - \{2 \cdot \text{size}(v_{i-1}) - \text{capacity}(v_{i-1}) + 1\} \\ &= 2 - \text{capacity}(v_{i-1}),\end{aligned}$$

and

$$\hat{c}_i = c_i + \Delta \Phi_i = 1 + \text{capacity}(v_{i-1}) + 2 - \text{capacity}(v_{i-1}) = 3$$

$$\implies \sum_{i=1}^n \hat{c}_i = 3n \geq \sum_{i=1}^n c_i.$$

Part II

Dictionaries

- 4 Hash Tables
- 5 Red-Black Trees

Part II

Dictionaries

4 Hash Tables

5 Red-Black Trees

Hash Tables

A **hash table** (cat: *taula de dispersió*, esp: *tabla de dispersión*) allows us to store a set of elements (or pairs $\langle key, value \rangle$) using a **hash function** $h : K \implies I$, where I is the set of indices or addresses into the table, e.g., $I = [0..M - 1]$.

Ideally, the hash function h would map every element (their keys) to a distinct address of the table, but this is hardly possible in a general situation, and we should expect to find **collisions** (different keys mapping to the same address) as soon as the number of elements stored in the table is $n = \Omega(\sqrt{M})$.

Hash Tables

If the hash function evenly “spreads” the keys, the hash table will be useful as there will be a small number of keys mapping to any given address of the table.

Given two distinct keys x and y , we say that they are **synonyms**, also that they **collide** if $h(x) = h(y)$.

A fundamental problem in the implementation of a dictionary using a hash table is to design a **collision resolution strategy**.

Hash Tables

```
template <typename T> class Hash {
public:
    int operator()(const T& x) const;
};

template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
public:
    ...
private:
    struct node {
        Key _k;
        Value _v;
        ...
    };
    nat _M; // capacity of the table
    nat _n; // number of elements (size) of the table
    double _alpha_max; // max. load factor
    HashFunct<Key> h;

    // open addressing
    vector<node> _Thash; // an array with pairs <key,value>

    // separate chaining
    // vector<list<node>> _Thash; // an array of lists of synonyms

    int hash(const Key& k) {
        return h(k) % _M;
    }
};
```

Hash Functions

A good hash function h must enjoy the following properties

- ➊ It is easy to compute
- ➋ It must evenly spread the set of keys K : for all i , $0 \leq i < M$

$$\frac{\#\{k \in K \mid h(k) = i\}}{\#\{k \in K\}} \approx \frac{1}{M}$$

Hash Functions

In our implementation, the class `Hash<T>` overloads operator `()` so that for an object `h` of the class `Hash<T>`, `h(x)` is the result of “applying” `h` to the object `x` of class `T`. The operation returns a positive integer.

The private method `hash` in class `Dictionary` computes

$$h(x) \% _M$$

to obtain a valid position into the table, an index between 0 and $_M - 1$.

Hash Functions

```
// specialization of the template for T = string
template <> class Hash<string> {
public:
    int operator()(const string& x) const {
        int s = 0;
        for (int i = 0; i < x.length(); ++i)
            s = s * 37 + x[i];
        return s;
    }

// specialization of the template for T = int
template <> class Hash<int> {
    static long const MULT = 31415926;
public:
    int operator()(const int& x) const {
        long y = ((x * x * MULT) << 20) >> 4;
        return y;
    }
};
```

Other sophisticated hash functions use weighted sums or non-linear transformations (e.g., they square the number represented by the k central bits of the key).

Collision Resolution

Collision resolution strategies can be grouped into two main families. By historical reasons (not very logically) they are called

- **Open hashing:** separate chaining, 2-way chaining, coalesced hashing, ...
- **Open addressing:** linear probing, double hashing, quadratic hashing, cuckoo hashing, ...

Separate Chaining

In separate chaining, each slot in the hash table has a pointer to a linked list of synonyms.

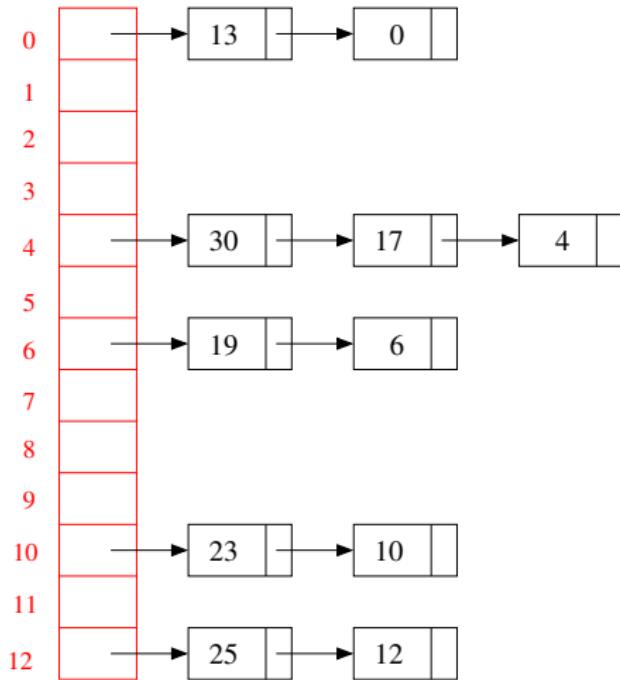
```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>
class Dictionary {
    ...
private:
    struct node {
        Key _k;
        Value _v;
        ...
    };
    vector<list<node>> _Thash; // array of linked lists of synonyms
    int _M;           // capacity of the table
    int _n;           // number of elements
    double _alpha_max; // max. load factor

    list<node>::const_iterator lookup_sep_chain(const Key& k, int i) const ;
    void insert_sep_chain(const Key& k,
                          const Value& v);
    void remove_sep_chain(const Key& k) ;
};
```

Separate Chaining

$$M = 13 \quad X = \{0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30\}$$

$$h(x) = x \bmod M$$



Separate Chaining

For insertions, we access the appropriate linked list using the hash function, and scan the list to find out whether the key was already present or not. If present, we modify the associated value; if not, a new node with the pair $\langle \text{key}, \text{value} \rangle$ is added to the list.

Since the lists contain very few elements each, the simplest and more efficient solution is to add elements to the front. There is no need for double links, sentinels, etc. Sorting the lists or using some other sophisticated data structure instead of linked lists does not report real practical benefits.

Separate Chaining

Searching is also simple: access the appropriate linked list using the hash function and sequentially scan it to locate the key or to report unsuccessful search.

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert(const Key& k,
                                              const Value& v) {
    insert_sep_chain(k, v);
    if (_n / _M > _alpha_max)
        // the current load factor is too large, raise here an exception or
        // resize the table and rehash
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::insert_sep_chain(
    const Key& k, const Value& v) {
    int i = hash(k);
    list<node>::const_iterator p = lookup_sep_chain(k, i);
    // insert as first item in the list
    // if not present
    if (p == _thash[i].end()) {
        _thash[i].push_back(node(k, v));
        ++_n;
    }
    else
        p->_v = v;
}
```

Separate Chaining

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
void Dictionary<Key,Value,HashFunct>::lookup(const Key& k,
                                             bool& exists, Value& v) const {
    int i = hash(k);
    list<node>::const_iterator p = lookup_sep_chain(k, i);
    if (p == _thash[i].end())
        exists = false;
    else {
        exists = true;
        v = p -> _v;
    }
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
list<Dictionary<Key,Value,HashFunct>::node>::const_iterator
Dictionary<Key,Value,HashFunct>::lookup_sep_chain(const Key& k,
                                                int i) const {
    list<node>::const_iterator p = _Thash[i].begin();
    // sequential search in the i-th list of synonyms
    while (p != _thash[i].end() and p-> _k != k)
        ++p;

    return p;
}
```

The Cost of Separate Chaining

Let n be the number of elements stored in the hash table. On average, each linked list contains $\alpha = n/M$ elements and the cost of lookups (either successful or unsuccessful), of insertions and of deletions will be proportional to α . If α is a small constant value then the cost of all basic operations is, on average, $\Theta(1)$. However, it can be shown that the expected length of the largest synonym list is $\Theta(\log n / \log \log n)$. The value α is called **load factor**, and the performance of the hash table will be dependent on it.

The Cost of Separate Chaining

- $L_n^{(i)}$: the number of elements hashing to the i -th list, $0 \leq i < M$, after the insertion of n items.
- Standard assumption: the probability that the j -th inserted item hashes to position i , $0 \leq i < M$, is $1/M$
- The random variables $L_n^{(i)}$, $0 \leq i < M$, are **not** independent, but they are identically distributed
- Set $L_n := L_n^{(0)}$. Let $Y_j = 1$ iff the j -th inserted item goes to list 0, and $Y_j = 0$ otherwise.

$$L_n = Y_1 + \dots + Y_n$$

$$\begin{aligned}\mathbb{E}[L_n] &= \mathbb{E}[Y_1 + \dots + Y_n] = \mathbb{E}[Y_1] + \dots + \mathbb{E}[Y_n] \\ &= 1/M + \dots + 1/M = n/M = \alpha\end{aligned}$$

The Cost of Separate Chaining

- Cost of unsuccessful search $U_n \approx$ cost of insertion of the $(n + 1)$ -th item

$$\begin{aligned}\mathbb{E}[U_n] &= \sum_{0 \leq i < M} \mathbb{E}[U_n | \text{search in list } i] \cdot \mathbb{P}[\text{search in list } i] \\ &= \frac{1}{M} \sum_{0 \leq i < M} \mathbb{E}[U_n | \text{search in list } i] = \frac{1}{M} \sum_{0 \leq i < M} (1 + \mathbb{E}[L_n^{(i)}]) \\ &= 1 + \alpha\end{aligned}$$

The Cost of Separate Chaining

- Cost of successful search of a random item $S_n \approx$ cost of deletion of a random item

$$\begin{aligned}\mathbb{E}[S_n] &= \sum_{0 \leq i < M : L_n^{(i)} > 0} \mathbb{E}[S_n | \text{search in list } i] \cdot \mathbb{P}[\text{search in list } i] \\ &= \sum_{0 \leq i < M : L_n^{(i)} > 0} \left(\sum_{\ell > 0} \frac{\ell + 1}{2} \mathbb{P}[L_n^{(i)} = \ell] \right) \cdot \frac{L_n^{(i)}}{n} \\ &= \sum_{0 \leq i < M : L_n^{(i)} > 0} \frac{1 + \alpha}{2} \frac{L_n^{(i)}}{n} \\ &= \frac{1 + \alpha}{2} \sum_{0 \leq i < M : L_n^{(i)} > 0} \frac{L_n^{(i)}}{n} = \frac{1 + \alpha}{2}\end{aligned}$$

The Cost of Separate Chaining

- The Poisson model: in order to avoid the dependence between $L_n^{(i)}$ we can consider a Poisson random model in which “balls” (items) are thrown into “bins” (slots in the hash table) at a rate α , then the length of each list $\mathcal{L}_i \sim \text{Poisson}(\alpha)$ is independent of all other
- We have, for instance, $\mathbb{E}[\mathcal{L}_i] = \alpha = \mathbb{E}[L_n^{(i)}]$
- In general we can make our computations in the easier Poisson model then (rigorously) transfer these results to the “exact model”

The Cost of Separate Chaining

- Let $L_n^* = \max\{L_n^{(0)}, \dots, L_n^{(M-1)}\}$. This random variable gives the worst-case cost of search, insertions and deletions
- An important identity for positive discrete r.v.

$$\mathbb{E}[X] = \sum_{k \geq 0} k \mathbb{P}[X = k] = \sum_{k \geq 0} \mathbb{P}[X > k]$$

The Cost of Separate Chaining

- In the Poisson model, we have M i.i.d. Poisson r.v. \mathcal{L}_i , all with parameter $\alpha = n/M$, giving the length of the i -th list, $0 \leq i < M$
- Then for $\mathcal{L}^* = \max_{0 \leq i < M} \{\mathcal{L}_i\}$ we have

$$\begin{aligned}\mathbb{P}[\mathcal{L}^* \leq k] &= \prod_i \mathbb{P}[\mathcal{L}_i \leq k] \\ &= \left(\sum_{0 \leq j \leq k} \frac{\alpha^j e^{-\alpha}}{j!} \right)^M\end{aligned}$$

and

$$\mathbb{E}[\mathcal{L}^*] = \sum_{k>0} \left(1 - \left(\sum_{0 \leq j \leq k} \frac{\alpha^j e^{-\alpha}}{j!} \right)^M \right)$$

- But this path leads us nowhere.

The Cost of Separate Chaining

We will try a different way:

- Compute (or give useful bounds) for the median of \mathcal{L}^* , i.e., the value of j such that $\mathbb{P}[\mathcal{L}^* \leq j] = 1/2$
- Show that the expectation (mean) of \mathcal{L}^* is close to its median, namely we show that

$$\frac{\mathbb{E}[\mathcal{L}^*]}{j} \rightarrow 1$$

if n is large enough (and $\alpha = n/M$ is kept constant)

The Cost of Separate Chaining

For which value j do we have

$$\mathbb{P}[\mathcal{L}^* \leq j] = \prod_i \mathbb{P}[\mathcal{L}_i \leq j] = \left(\sum_{0 \leq k \leq j} \frac{\alpha^k e^{-\alpha}}{k!} \right)^M = 1/2?$$

The summation

$$\sum_{0 \leq k \leq j} \frac{\alpha^k}{k!} \approx e^\alpha - \frac{\alpha^{j+1}}{(j+1)!},$$

hence

$$\mathbb{P}[\mathcal{L}^* \leq j] \approx \left(1 - \frac{\alpha^{j+1}}{(j+1)!} e^{-\alpha} \right)^M$$

The Cost of Separate Chaining

We want j such that

$$\left(1 - \frac{\alpha^{j+1}}{(j+1)!} e^{-\alpha}\right)^M = \frac{1}{2}$$

Taking natural logs on both sides

$$M \ln \left(1 - \frac{\alpha^{j+1} e^{-\alpha}}{(j+1)!}\right) = -\ln 2$$

Since $\ln(1-x) \sim x + x^2/2 + O(x^3)$

$$\left(\frac{\alpha^{j+1} e^{-\alpha}}{(j+1)!} + \dots\right) \approx \frac{-\ln 2}{M}$$

The Cost of Separate Chaining

Hence

$$\frac{\alpha^{j+1} e^{-\alpha}}{(j+1)!} = \Theta(1/M)$$

- $\alpha \rightarrow 1$ implies $(j+1)! = \frac{M}{e \ln 2}$, that is, $j = \Gamma^{(-1)}(M/(e \ln 2))$.
- For $\alpha < 1$ we also have $j = \Theta(\Gamma^{(-1)}(M))$.
- Since $\Gamma^{(-1)}(n) \sim \ln n / \ln \ln n$, and $n = \alpha M$ we have that the median j of \mathcal{L}^* is $j = \Theta(\log n / \log \log n)$.

(see next slide for definitions and remarks)

The Cost of Separate Chaining

Note:

- $\Gamma^{(-1)}$ = inverse of the Gamma function $\Gamma(z)$
- Γ generalizes factorials to complex numbers
 $(\Gamma(z + 1) = z\Gamma(z))$.
- Since $\ln n! \sim n \ln n - n + O(1)$ (Stirling's approximation)
we can easily prove $\Gamma^{(-1)}(n) \sim \ln n / \ln \ln n$ if $n \rightarrow \infty$.

For the rest of the proof (showing that the expected value of \mathcal{L}^* has the same order of growth as its median) you can check Section 2.2 in [Gon81]

d-way Chaining

Azar, Broder, Karlin and Upfal [ABKU99] have shown the following important result

Theorem

Suppose n balls are sequentially placed in $m \geq n$ bins, so that for each ball $d \geq 2$ random bins are chosen and the ball is placed in the least full bin —with ties broken arbitrarily. Then with high probability, as $n \rightarrow \infty$, the fullest bin contains

$$(1 + o(1)) \ln \ln n / \ln d + \Theta(m/n)$$

balls.

d-way Chaining

This result has many applications, not only for data structures design. In the context of hashing, the hashing scheme suggested by this result is very straightforward:

- To insert an item x , compute $i = h_1(x)$ and $j = h_2(x)$ with two (or in general d) independent hash functions and insert x in the synonym list which is shorter, i.e., list i if $L_i \leq L_j$ and vice-versa
- To search (or delete) an item x compute $i = h_1(x)$ and $j = h_2(x)$ and search for x in both lists (why? why not only the shortest?) Clearly if x is present it must be in one of these two lists

d -way Chaining

In **d -way chaining** we basically multiply by d the expected costs of all operations, as compared to separate chaining, as we need to evaluate d hash functions and search in that many lists. We also need to keep the size of each list.

It is very easy to show that with d -way chaining the expected length of each list is $\alpha = n/m$ like in ordinary separate chaining. However:

- the variance of each $L_n^{(i)}$ is smaller than in separate chaining
- the expected longest list has length $\Theta(\log \log n)$, a huge improvement w.r.t. the $\Theta(\log n / \log \log n)$ in separate chaining

For those interested in the details (in particular the proof of the result) check [ABKU99].

Open Addressing

In **open addressing**, synonyms are stored in the hash table. Searches and insertions probe a sequence of positions until the given key or an empty slot is found. The sequence of probes starts in position $i_0 = h(k)$ and continues with i_1, i_2, \dots . The different open addressing strategies use different rules to define the sequence of probes. The simplest one is **linear probing**:

$$i_1 = i_0 + 1, i_2 = i_1 + 1, \dots,$$

taking modulo M in all cases.

Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct = Hash>

class Dictionary {
    ...
private:
    struct node {
        Key _k;
        Value _v;
        bool _free;
        // constructor for class node
        node(const Key& k, const Value& v, bool free = true);
    };
    vector<node> _Thash; // array of nodes
    int _M;           // capacity of the table
    int _n;           // number of elements
    double _alpha_max; // max. load factor (must be < 1)

    int lookup_linear_probing(const Key& k) const ;
    void insert_linear_probing(const Key& k,
                               const Value& v);
    void remove_linear_probing(const Key& k) ;
};
```

Linear Probing

M = 13

X = { 0, 4, 6, 10, 12, 13, 17, 19, 23, 25, 30 }

$h(x) = x \bmod M$ (incremento 1)

0	0
1	
2	
3	
4	4
5	
6	6
7	
8	
9	
10	10
11	
12	12



0	0	occupied
1	13	occupied
2		free
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8		free
9		free
10	10	occupied
11	23	occupied
12	12	occupied

0	0	occupied
1	13	occupied
2	25	occupied
3		free
4	4	occupied
5	17	occupied
6	6	occupied
7	19	occupied
8	30	occupied
9		free
10	10	occupied
11	23	occupied
12	12	occupied

+ { 0, 4, 6, 10, 12 }

+ { 13, 17, 19, 23 }

+ { 25, 30 }

Linear Probing

```
template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup(
    const Key& k,
    bool& exists, Value& v) const {
    int i = lookup_linear_probing(k);
    if (not _Thash[i]._free and _Thash[i]._k == k) {
        exists = true; v = _Thash[i]._v;
    }
    else
        exists = false;
}

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::lookup_linear_probing(
    const Key& k) const {
    int i = hash(k);
    int visited = 0; // this is only necessary if
                      // _n == _M, otherwise there is at least
                      // a free position
    while (not _Thash[i]._free and _Thash[i]._k != k
           and visited < _M) {
        ++visited;
        i = (i + 1) % _M;
    }
    return i;
}
```

Deletions in Open Addressing

There is no general solution for true deletions in open addressing tables. It is not enough to mark the position of the element to be removed as “free”, since later searches might report as not present some element which is stored in the table.

The general technique that can be used is **lazy deletions**. Each slot can be free, occupied or **deleted**. Deleted slots can be used to store there a new element, but they are not free and searches must pass them over and continue.

Deletions in Linear Probing

For linear probing, we can do true deletions. The deletion algorithm must continue probing the positions after the removed element, and moving to the emptied slot any element whose hash address is equal (or smaller in the cyclic order) to the address of the emptied slot. Moving an element creates a new emptied slot, and the procedure is repeated until an empty slot is found. In our implementation we will use the function $\text{displ}(j, i)$ which gives us the distance between j e i in the cyclic order: if $j > i$ we must turn around position $M - 1$ and go back to position 0.

```
int displ(j, i, M) {
    if (i >= j)
        return i - j;
    else
        return M + (i - j);
}
```

Deletions in Linear Probing

```
// we assume _n < _M

template <typename Key, typename Value,
          template <typename> class HashFunct>
int Dictionary<Key,Value,HashFunct>::remove_linear_probing(
    const Key& k) const {
    int i = lookup_linear_probing(k);
    if (not _Thash[i]._free) {
        // _Thash[i] is the element to remove
        int free = i; i = (i + 1) % _M; int d = 1;
        while (not _Thash[i]._free) {
            int i_home = hash(_Thash[i]._k);
            if (displ(i_home, i, _M) >= d) {
                _Thash[free] = _Thash[i]; free = i; d = 0;
            }
            i = (i + 1) % _M; ++d;
        }
        _Thash[free]._free = true; --_n;
    }
}
```

Other Open Addressing Schemes

As we have already mentioned different probe sequences give us different **open addressing** strategies. In general, the sequence of probes is given by

$$\begin{aligned} i_0 &= h(x), \\ i_j &= i_{j-1} \oplus \Delta(j, x), \end{aligned}$$

where $x \oplus y$ denotes $x + y \pmod M$.

Other Open Addressing Schemes

- ① Linear Probing: $\Delta(j, x) = 1$ (or a constant); $i_j = h(x) \oplus j$
- ② Quadratic Hashing: $\Delta(j, x) = a \cdot j + b$;
 $i_j = h(x) \oplus (Aj^2 + Bj + C)$; constants a and b must be carefully chosen to guarantee that the probe sequence will ultimately explore all the table if necessary
- ③ Double Hashing: $\Delta(j, x) = h_2(x)$ for a second independent hash function h_2 such that $h_2(x) \neq 0$; $i_j = h(x) \oplus j \cdot h_2(x)$
- ④ Uniform Hashing: i_0, i_1, \dots is a random permutation of $\{0, \dots, M - 1\}$
- ⑤ Random Probing: i_0, i_1, \dots is a random sequence such that $0 \leq i_k < M$, for all k , and it contains every value in $\{0, \dots, M - 1\}$ at least once

Other Open Addressing Schemes

Uniform Hashing and Random Probing are completely impractical algorithms; they are interesting as idealizations —they do not suffer from **clustering**

- Linear Probing suffers **primary clustering**. There are only M distinct probe sequences, the M circular permutations of $0, 1 \dots, M - 1$
- Quadratic Hashing and other methods with $H(j, x) = f(j)$ (a non-constant function only of j) behave almost as the schemes with **secondary clustering**: two keys such that $h(x) = h(y)$ will probe exactly the same sequence of slots, but if a key x probes i_j in the j -th step and y probes i'_k in the k -th step then i_{j+1} and i'_{k+1} will be probably different
- Double Hashing is even better and generalizations, they exhibit **secondary** (more generally **k -ary clustering**) as they depend on $(k - 1)$ evaluations of independent hash functions

Other Open Addressing Schemes

- In linear probing two keys will have the same probe sequence with probability $1/M$; in an scheme with secondary clustering that probability drops to $1/M(M - 1)$
- The average performance of schemes with k -ary clustering, $k \geq 2$, is close to that of uniform hashing (no clustering)
- Random probing also approximates well the performance of uniform hashing

The Cost of Open Addressing

We will focus in the following parameters (we assume M is fixed):

- ① \mathcal{U}_n : number of probes in an **unsuccessful search** that starts at a random slot in a table with n items
- ② $S_{n,i}$: number of probes in the **successful** search of the i -th inserted item when the table contains n items, $1 \leq i \leq n$

We will actually be more interested in $S_n := S_{n,U_n}$ where U_n is a random uniform value in $\{1, \dots, n\}$, that is, S_n is the cost of a successful search of a random item in a table with n items

The Cost of Open Addressing

- The cost of the $(n + 1)$ -th insertion is given by \mathcal{U}_n
- With the FCFS insertion policy (see next slides), an item will be inserted where the unsuccessful search terminated and never be moved from there, hence

$$\mathcal{S}_{n,i} \stackrel{\mathcal{D}}{=} \mathcal{U}_{i-1}$$

where $\stackrel{\mathcal{D}}{=}$ denotes equal distribution

The Cost of Open Addressing

Consider random probing. What is $U_n = \mathbb{E}[\mathcal{U}_n]$?

With one probe we land in an empty slot and we are done.

Probability is $(1 - \alpha)$. If the first place is occupied, probability α , we probe a second slot, which is empty with probability $1 - \alpha$. And so on. Thus

$$\begin{aligned} U_n &= 1 \times (1 - \alpha) + 2 \times \alpha \cdot (1 - \alpha) + 3 \times \alpha^2 \cdot (1 - \alpha) \\ &= \sum_{k>0} k\alpha^{k-1} \cdot (1 - \alpha) = (1 - \alpha) \sum_{k>0} \frac{d(\alpha^k)}{d\alpha} \\ &= (1 - \alpha) \frac{d}{d\alpha} \sum_{k>0} \alpha^k = \frac{1}{1 - \alpha}. \end{aligned}$$

The Cost of Open Addressing

And for the expected successful search we have

$$S_n = \mathbb{E}[S_n] = \frac{1}{n} \sum_{1 \leq i \leq n} \mathbb{E}[S_{n,i}] = \frac{1}{n} \sum_{1 \leq i \leq n} \mathbb{E}[U_{i-1}] = \frac{1}{n} \sum_{1 \leq i \leq n} U_{i-1}$$

Using Euler-McLaurin

$$S_n = \frac{1}{\alpha M} \sum_{1 \leq i \leq n} U_{i-1} = \frac{1}{\alpha} \int_0^\alpha \frac{1}{1-\beta} d\beta = \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$$

The Cost of Open Addressing

The actual expected costs of hashing with uniform hashing (and thus of quadratic hashing, double hashing) are slightly different from those of random probing, a few small corrections must be introduced:

- $U_n = 1/(1 - \alpha) - \alpha - \ln(1 - \alpha)$
- $S_n = 1/\alpha \int_0^\alpha U(\beta) d\beta = 1 - \alpha/2 - \ln(1 - \alpha)$ (*)

The Cost of Open Addressing

The analysis of linear probing turns out to be more challenging than one could think at first.

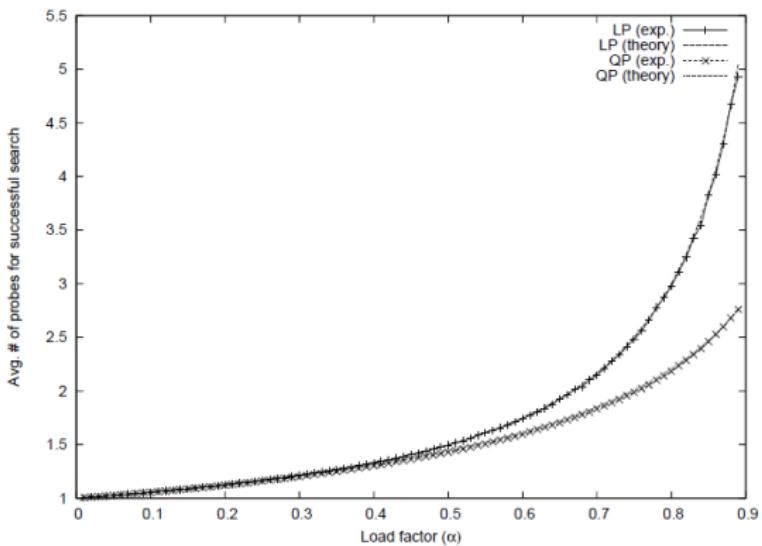
- The average cost of unsuccessful search is

$$U_n = \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right)$$

- The average cost of successful search is

$$S_n = \frac{1}{\alpha} \int_0^\alpha U(\beta) d\beta = \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \quad (**)$$

The Cost of Open Addressing



Comparison of experimental vs. theoretical expected cost of successful search in linear probing and quadratic hashing

Insertion Policies in Open Addressing

- The standard insertion policy in case of a collision is FCFS (first-come-first-served): the item x that occupies an slot remains there, and the colliding item y continues with its probe sequence
- But other policies are also possible and have been proposed in the literature:
 - LCFS (last-come-first-served): y kicks out x , x continues with its probe sequence
 - Ordered hashing: If $x \leq y$, x remains and y continues, and the other way around otherwise
 - Robin Hood: the item farthest away from its home location stays, the other continues, ties are resolved arbitrarily

Insertion Policies in Open Addressing

All these strategies lead to the same average successful search cost as FCFS, but:

- the variance is significantly reduced
- most importantly, the expected worst case is reduced from $\Theta(\log n)$ to $\Theta(\log \log n)$

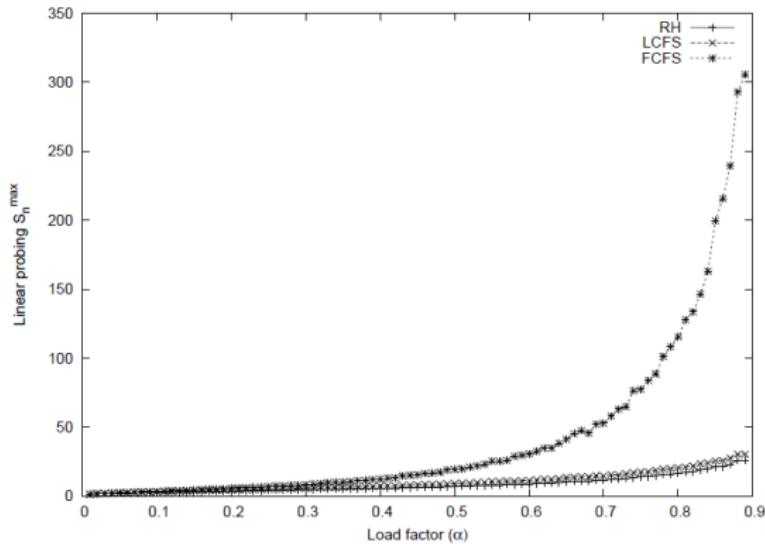
Think for instance in linear probing. The length of clusters will be the same for FCFS, LCFS, OH and RH, and the **sum of the distances** of all items to their respective home locations also changes but the distribution of distances to home location will vastly differ—like in the two sums below

$$1 + 3 + 7 = 3 + 4 + 4$$

Insertion Policies in Open Addressing

- Both ordered hashing and Robin Hood have the very nice feature that, given a set X of items to be inserted the final table is always the same, irrespective of the order in which item are inserted.
- This invariance with respect the prder of insertions notably simplifies some analysis.
- The unsuccessful search cost in OH and RH can be greatly improved; no need to continue until an empty slot is found (why?)
- But the insertion cost is the same, we either stop at an empty location or we kick out some item to insert the new item, but then we must continue

Insertion Policies in Open Addressing



Comparison of the maximum expected cost of a successful search in linear probing with FCFS (standard), LCFS and RH

Cuckoo Hashing

In **cuckoo hashing** we have **two** tables T_1 and T_2 of size M each, and two hash functions $h_1, h_2 : 0 \rightarrow M - 1$.

We can insert in such table $n < M$ items: the load factor $\alpha = n/2M$ must be strictly less than 1/2

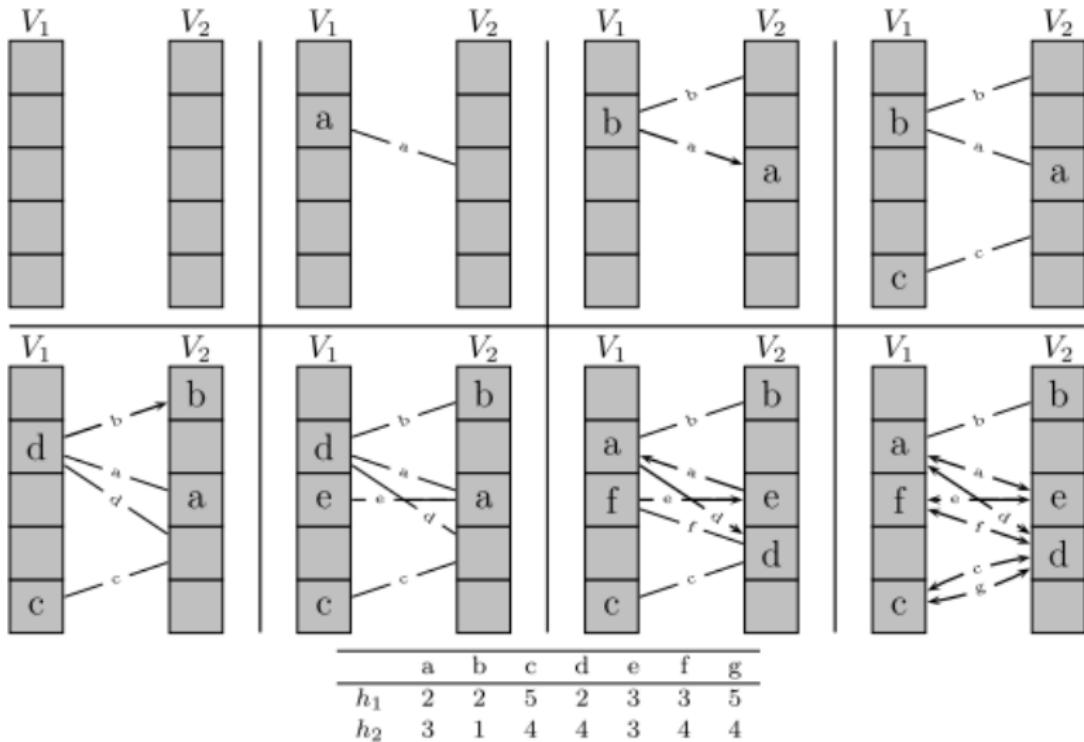
Cuckoo Hashing



To insert a new item x , we probe slot $T_1[h_1(x)]$, if it is empty, we put x there and stop. Otherwise if y sits already in that slot, then x **kicks out** y — x is put in $T_1[h_1(x)]$ and y moves to $T_2[h_2(y)]$. If that slot in T_2 is empty, we're done, but if some z occupies $T_2[h_2(y)]$, then y is put in its second “nest” and z is kicked out to $T_1[h_1(z)]$, and so on.

These “kicks out” give the name to this strategy. If this procedure succeeds to insert n keys then each key x **can only appear in one of its two nests**: $T_1[h_1(x)]$ or $T_2[h_2(x)]$, nowhere else!

Cuckoo Hashing



Cuckoo Hashing

```
// Representation of the dictionary with cuckoo hashing
struct node {
    Key _k;
    Value _v;
    bool _free;
};
vector<node> _T1, _T2;
int _M, _n;
Hash<Key> _h1, _h2;
...
```

Cuckoo Hashing

```
void lookup(const Key& k, bool& exists, Value& v) const {
    exists = false;
    node& n1 = _T1[_h1(x)];
    if (not n1._free and n1._key == k) {
        exists = true; v = n1._v;
    } else {
        node& n2 = _T2[_h2(x)];
        if (not n2._free and n2._key == k) {
            exists = true; v = n2._v;
        }
    }
}
```

Only **two** probes are necessary in the worst-case! To delete we
localate with ≤ 2 probes the key to remove and mark the slot as
free.

Cuckoo Hashing

- The insertion of an item x can fail because we enter in an infinite loop of items each kicking out the next in the cycle
 - ...
- The solution to the problem: nuke the table! Draw **two new hash functions**, and **rehash** everything again with the two new functions.
- This rehashing is clearly quite costly; moreover, we don't have a guarantee that the new functions will succeed where the old failed!

We will see, however, that **insertion has expected amortized constant cost**, or equivalently, that the expected cost of n insertions is $\Theta(n)$

Cuckoo Hashing

```
void insert(const Key& k, const Value& v) {
    if (_n == _M - 1) { // resize and rehash, cannot insert >= _M items
    }
    // _n < _M - 1
    if (''k in the table'') { // update v and return
    }
    node x = { k, v, false };
    for (int i = 1; i <= MaxIter(_n, _M);++i) {
        // we can take MaxIter = 2n, we should never see
        // more than 2n vertices unless we're in an infinite loop
        swap(x, _T1[_h1(x._k)]);
        if (x._free) return;
        swap(x, _T2[_h2(x._k)]);
        if (x._free) return;
    }
    // failure!! rehash and try again:
    rehash(); insert(k,v);
}
```

Cuckoo Hashing

We say that an insertion is *good* if it does not run into a infinite loop (our implementation protects from ∞ -loops by bounding the number of iterations).

A “high-level analysis” of the cost of insertions follows from:

- ➊ The expected number of steps/iterations in a good insertion is $\Theta(1)$
- ➋ The probability that the insertion of an item is not good is $O(1/n^2)$

Cuckoo Hashing

- ④ By the union bound, the probability that we fail to make n consecutive good insertions is $O(1/n)$
- ⑤ The expected total cost of making n good insertions—conditioned on the event that we can make them—is $n \times \Theta(1) = \Theta(n)$

Cuckoo Hashing

- ① The expected number of times we need to rehash a set of n items until we can insert all with good insertions is given by a **geometric r.v.** with probability of success $1 - O(1/n)$:

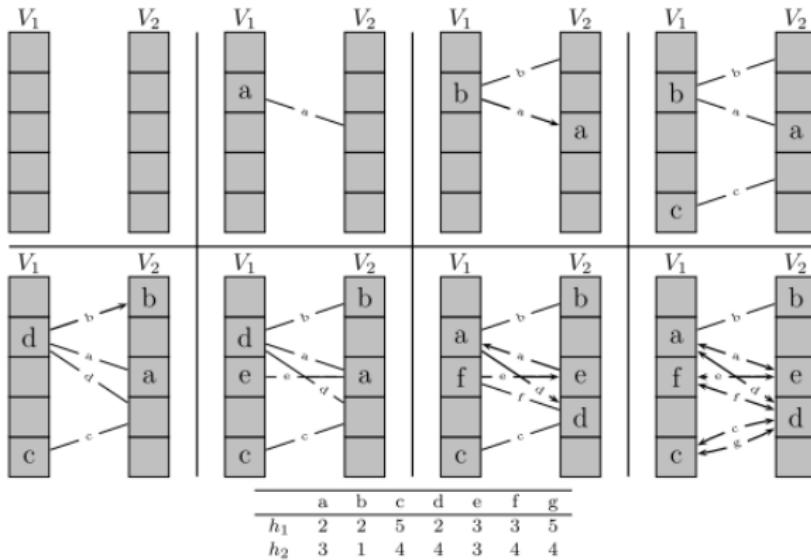
$$\mathbb{E}[\# \text{ rehashes}] = \frac{1}{1 - O(1/n)} = 1 + O(1/n)$$

- ② Each rehash plus the attempt to insert with good insertions the n items has expected cost $\Theta(n)$
- ③ By Wald's lemma, the expected cost of the insertion will be

$$\mathbb{E}[\#\text{rehashes}] \times \mathbb{E}[\text{cost of rehash}] = (1 + O(1/n)) \times O(n) = O(n)$$

Cuckoo Hashing

To prove facts #1 (good insertion needs expected $O(1)$ time) and #2 (probability of a good insertion is $1 - O(1/n^2)$) we formulate the problem in **graph theoretic** terms.



Cuckoo Hashing

Cuckoo graph:

- Vertices: $V = \{v_{1,i}, v_{2,i} \mid 0 \leq i < M\} =$
the set of $2M$ slots in the tables
- Edges: If $T_1[j]$ is occupied by x then there's an edge
 $(v_{1,j}, v_{2,h_2(x)})$, where $v_{\ell,j}$ is the vertex associated to $T_\ell[j]$; x
is the label of the edge. If $T_2[k]$ is occupied by y then there
is an edge $(v_{2,k}, v_{1,h_1(y)})$ with label y .

This is a labeled directed “bipartite” multigraph—all edges go
from $v_{1,j}$ to $v_{2,k}$ or from $v_{1,k}$ to $v_{2,j}$.

Cuckoo Hashing

Consider the connected components of the cuckoo graph. A component can be either a tree (no cycles), unicyclic (exactly one cycle—with trees “hanging”) or **complex** (two or more cycles). Trees with k nodes have exactly $k - 1$ edges, unicycles have exactly k edges and complex components have $> k$ edges.

- Fact 1: An insertion that creates a complex component is not good \implies if the cuckoo graph contains no complex components then all insertions were good
- Fact 2: the expected time of a good insertion is bounded by the expected diameter of the component in which we make the insertion (also by the size)

Cuckoo Hashing

Then we convert the analysis to that of the cuckoo graph as a random bipartite graph with $2M$ vertices and $n = (1 - \epsilon)M$ edges—each item gives us an edge.

This is a very “sparse” graph, but if the density n/M grew to $1/2$ there will be an complex component with very high probability (a similar thing happens in random Erdős-Renyi graphs).

Cuckoo Hashing

The most detailed analysis of the cuckoo graph has been made by Drmota and Kutzelnigg (2012). They prove, among many other things:

- ➊ The probability that the cuckoo graph contains no complex component is

$$1 - h(\epsilon) \frac{1}{M} + O(1/M^2)$$

We do not reproduce their explicit formula for $h(\epsilon)$ here
($h(\epsilon) \rightarrow \infty$ as $\epsilon \rightarrow 0$)

- ➋ The expected number of steps in n good insertions is

$$\leq n \cdot \min \left(4, \frac{\ln(1/\epsilon)}{1 - \epsilon} \right) + O(1)$$

These two results prove the two Facts that we needed for our analysis

Cuckoo Hashing

- Several variants of Cuckoo hashing have appeared in the literature, for instance, using $d > 2$ tables and d hash functions. With such d -Cuckoo Hashing higher load factor, approaching 1 can be achieved
- An interesting variant puts all items in one single table, all the $d \geq 2$ hash functions map keys into the range $0..M - 1$; the load factor n/M must be below some threshold α_d . We need to know which function was used to put the item at an occupied location—easily using $\log_2 d$ bits.

Hash Tables

To learn more:

-  Y. Azar, A. Broder, A. Karlin and E. Upfal.
Balanced Allocations
SIAM J. Computing, 29(1):180-200, 1999.
-  Gaston H. Gonnet
Expected Length of the Longest Probe Sequence in Hash
Code Searching
Journal of the ACM, 28 (2): 289–304, 1981.
-  Leo J. Guibas
The Analysis of Hashing Techniques That Exhibit k -ary
Clustering
Journal of the ACM, 25 (4): 544–555, 1978.

Hash Tables

To learn more:

-  [Donald E. Knuth](#)
The Art of Computer Programming. Volume 3: Sorting and
Searching (2nd ed)
[Addison-Wesley, Reading, MA, 1998.](#)
-  [Rasmus Pagh and Flemming F. Rodler](#)
Cuckoo Hashing
[*Journal of Algorithms* 51:122-144, 2004](#)

Hash Tables

To learn more:

-  [L. Devroye and P. Morin](#)
Cuckoo hashing: further Analysis
Information Proc. Letters 86:215–219, 2003
-  [M. Drmota and R. Kutzelnigg](#)
A precise analysis of Cuckoo Hashing
ACM Transactions on Algorithms 8(2):1–36, 2012

Part II

Dictionaries

4 Hash Tables

5 Red-Black Trees

Red-black Trees

- Red-black trees were introduced by Guibas and Sedgewick in 1978 as a practical implementation for so-called 2-3-4 trees (stay tuned! more on 2-3-4 trees later!)
- Red-black trees constitute a simple implementation of balanced trees, which does only require one bit per nodes as balancing information (color of the nodes)
- They provide logarithmic cost guarantees for all search & update operations, like other well known balanced search trees, e.g., AVLs
- They've become a *standard de facto* in the industry; for example, they are the most common way to implement C++ STL sets and maps

Red-black trees

Definition

A *red-black tree* T is a (full) binary tree —that is, every internal node has exactly two children, every external node (leaves) has degree 0— such that

- ① It stores a collection X of n items in its n internal nodes, one item per node, and it is a **binary search tree** for X
- ② Every node is either **red** or **black**
- ③ All leaves are **black**
- ④ No **red** node has a **red** child \equiv every **red** node has two **black** children
- ⑤ For any node x , the number of **black** nodes in any path from x to one of its descendant leaves is the same.
- ⑥ The root is **black** *this is only by convention; not a necessary condition

Red-black trees

- $h(x)$ = distance from node x to the farthest descendant leaf = **height** of x
- $h(T) := h(\text{root}(T))$ = height of the red-black tree T
- $\text{bh}(x)$ = number of black nodes in any path from node x to any of its descendant leaves = **black height** of x
- $\text{bh}(T) := \text{bh}(\text{root}(T))$

Red-black trees

- $h(x)$ = distance from node x to the farthest descendant leaf = **height** of x
- $h(T) := h(\text{root}(T))$ = height of the red-black tree T
- $\text{bh}(x)$ = number of black nodes in any path from node x to any of its descendant leaves = **black height** of x
- $\text{bh}(T) := \text{bh}(\text{root}(T))$

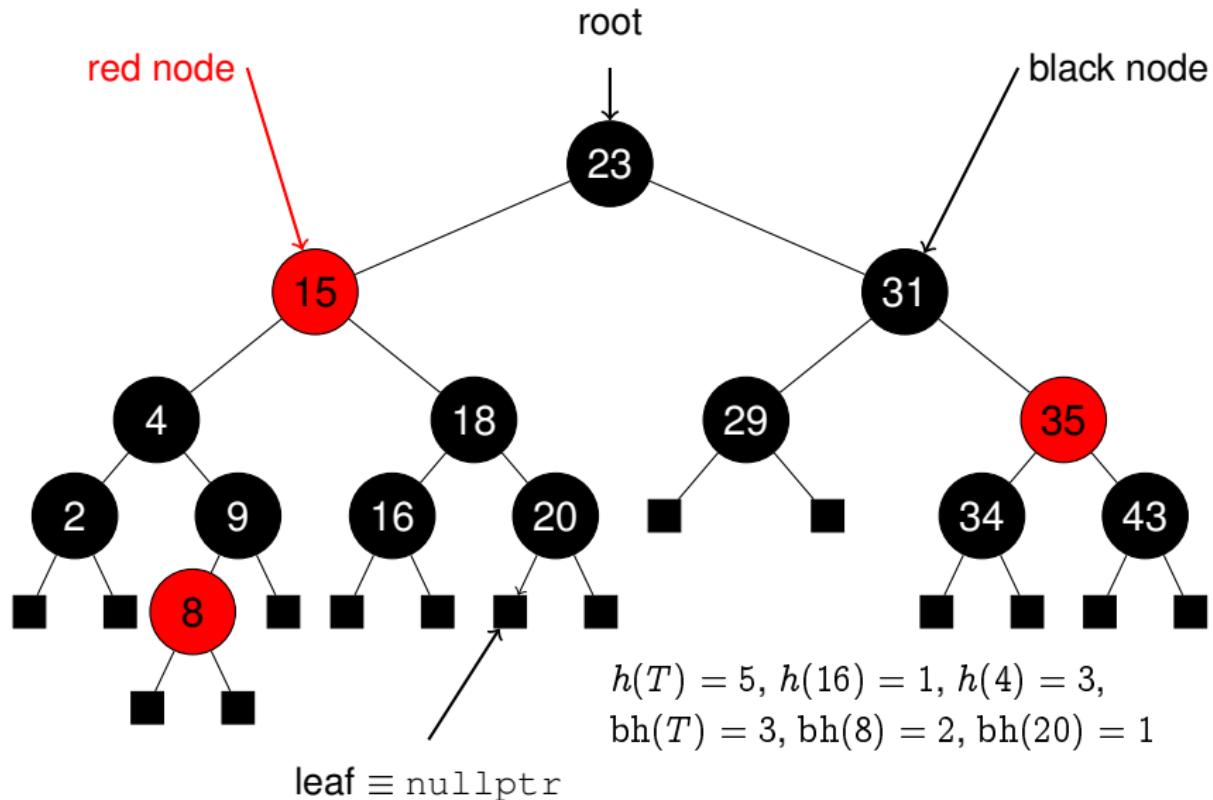
Red-black trees

- $h(x)$ = distance from node x to the farthest descendant leaf = **height** of x
- $h(T) := h(\text{root}(T))$ = height of the red-black tree T
- $\text{bh}(x)$ = number of black nodes in any path from node x to any of its descendant leaves = **black height** of x
- $\text{bh}(T) := \text{bh}(\text{root}(T))$

Red-black trees

- $h(x) =$ distance from node x to the farthest descendant leaf = **height** of x
- $h(T) := h(\text{root}(T)) =$ height of the red-black tree T
- $\text{bh}(x) =$ number of black nodes in any path from node x to any of its descendant leaves = **black height** of x
- $\text{bh}(T) := \text{bh}(\text{root}(T))$

Red-black trees



The Height of Red-black Trees

Lemma

For any red-black tree T , $h(T) \leq 2\lg(|T| + 1)$

Proof

We prove first, by induction, that the size of the subtree T_x rooted at x is at least $2^{\text{bh}(x)} - 1$.

- Basis of induction: $h(x) = 0$. This can only be true if x is a leaf, then $\text{bh}(x) = 0$ and $|T_x| = 0 = 2^0 - 1$.

The Height of Red-black Trees

Proof (cont'd)

- Inductive step: $h(x) > 0$. Let y and z denote the roots of the left and right subtrees of T_x .
 - If y is red, then $\text{bh}(y) = \text{bh}(x)$; if y is black then $\text{bh}(y) = \text{bh}(x) - 1$. In any case, $2^{\text{bh}(y)} \geq 2^{\text{bh}(x)-1}$
 - $h(y) < h(x)$

Likewise for z . By induction,

$$\begin{aligned}|T_x| &= 1 + |T_y| + |T_z| \geq 1 + 2^{\text{bh}(y)} - 1 + 2^{\text{bh}(z)} - 1 \\ &\geq 2^{\text{bh}(x)-1} + 2^{\text{bh}(x)-1} - 1 = 2^{\text{bh}(x)} - 1.\end{aligned}$$

The Height of Red-black Trees

Proof (cont'd)

- Inductive step: $h(x) > 0$. Let y and z denote the roots of the left and right subtrees of T_x .
 - If y is red, then $\text{bh}(y) = \text{bh}(x)$; if y is black then $\text{bh}(y) = \text{bh}(x) - 1$. In any case, $2^{\text{bh}(y)} \geq 2^{\text{bh}(x)-1}$
 - $h(y) < h(x)$

Likewise for z . By induction,

$$\begin{aligned}|T_x| &= 1 + |T_y| + |T_z| \geq 1 + 2^{\text{bh}(y)} - 1 + 2^{\text{bh}(z)} - 1 \\ &\geq 2^{\text{bh}(x)-1} + 2^{\text{bh}(x)-1} - 1 = 2^{\text{bh}(x)} - 1.\end{aligned}$$

The Height of Red-black Trees

Proof (cont'd)

Consider any red-black tree T of height h and size n . No red node can have a red child, hence at least half of the nodes in the path from the root to the farthest leaf must be black:

$$\text{bh}(T) \geq \lceil h/2 \rceil$$

Then,

$$n = |T| \geq 2^{\text{bh}(T)} - 1 \geq 2^{\lceil h/2 \rceil} - 1$$

That is,

$$n + 1 \geq 2^{\lceil h/2 \rceil},$$

Taking $\lg \equiv \log_2$ and multiplying by 2

$$2 \lg(n + 1) \geq 2 \left\lceil \frac{h}{2} \right\rceil \geq h$$



Insertions in Red-black Trees

How to insert a new element x into read-black tree T ?

- ① Search for x in T (using the standard search algorithm for BSTs) and replace the leaf where search ends by the new internal node x
- ② Paint x in red

Properties 4 and 6 might be invalid because of the insertion; all the other properties hold since the new node x is red and inserted according to the BST invariant

Insertions in Red-black Trees

To restore property 4 we shall perform several rotations (see next slides) & recolorations until the property 4 holds. If, by the end of the process, the root becomes **red**, then trivially restore property 6 setting

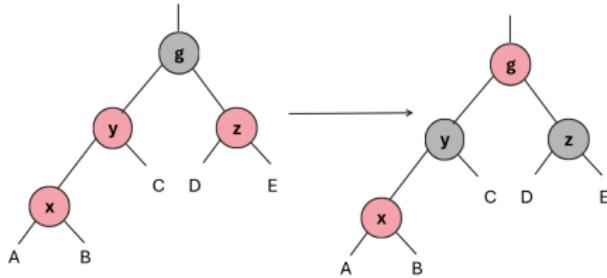
```
root -> color = black;
```

- ③ If x has no parent, or the parent of x is **black**, then we're done! Reestablish property 6 (if need be)
- ④ Otherwise, x has a **red** parent y , and then we have two cases depending on the color of the “uncle” z of x . Node z is y 's sibling; y has a sibling because it cannot be the root, we assume that y is **red**

Insertions in Red-black Trees

Case (a): z is red $\implies x$'s grandpa g must be **black** since both y and z are red

Recolor y and z black, and make g red.

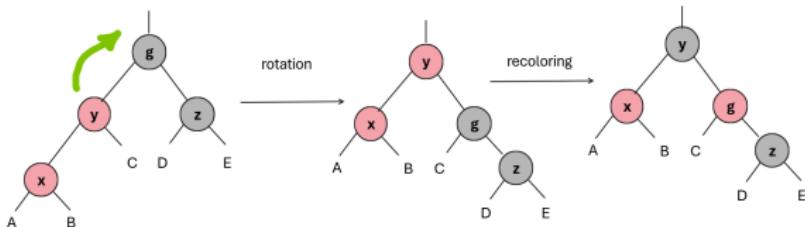


Go back to step 3, now g playing the role of x (violation of property 4 has been removed or propagated upwards)

Insertions in Red-black Trees

Case (b1): z is black x and y are both left children (the symmetric case, both x and y are right children, is dealt with analogously with RR rotation)

Do a simple LL rotation at g and recolor y and g

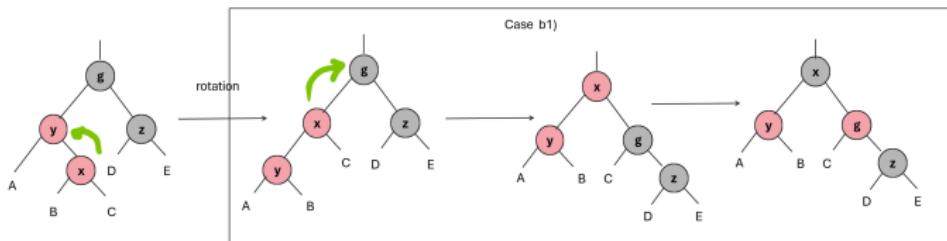


No further rotations/recolorations are necessary as all properties have been restored

Insertions in Red-black Trees

Case (b2): z is black x is the right child of y which is the left child of g (the symmetric case, x is left child, y is right child, is dealt with analogously)

Do a simple rotation at y and reduce to case (b1)
≡ do a double rotation LR at g and recolor



No further rotations/recolorations are necessary
as all properties have been restored

Insertions in Red-black Trees

```
template <typename Key, typename Value>
class Dictionary {
public:
    ...
    void lookup(const Key& k,
                bool& exists, Value& v) const;
    void insert(const Key& k,
                const Value& v);
    void remove(const Key& k);
    ...
private:
    enum node_color {black, red};
    struct node_rb_tree {
        Key _k;
        Value _v;
        node_color _color;
        node_rb_tree* _left;
        node_rb_tree* _right;
        // constructor for class node_rb_tree
        node_rb_tree(const Key& k, const Value& v,
                     node_color _color = red,
                     node_avl* left = nullptr,
                     node_avl* right = nullptr);
    };
    node_rb_tree* root;
    ...
    static node_rb_tree* rb_tree_insert(node_rb_tree* p,
                                        const Key& k, const Value& v);
    static node_rb_tree* rotate_LL(node_rb_tree* p);
    static node_rb_tree* rotate_LR(node_rb_tree* p);
    ...
};
```

Insertions in Red-black Trees

```
template <typename Key, typename Value>
static int Dictionary<Key,Value>::color(
    node_rb_tree* p) {
    if (p == nullptr)
        return black;
    else
        return p -> _color;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_rb_tree*
Dictionary<Key,Value>::rotate_LL(node_rb_tree* p) {
    node_rb_tree* q = p -> _left;
    p -> _left = q -> _right;
    q -> _right = p;
    return q;
}

template <typename Key, typename Value>
Dictionary<Key,Value>::node_rb_tree*
Dictionary<Key,Value>::rotate_LR(node_rb_tree* p) {
    p -> _left = rotate_RR(p -> _left);
    return rotate_LL(p);
}
...
}
```

Insertions in Red-black Trees

```
template <typename Key, typename Value>
void Dictionary<Key,Value>::insert(const Key& k,
    const Value& v) {
    root = rb_tree_insert(root, k, v);
    root -> _color = black;
};
```

Insertions in Red-black Trees

```
template <typename Key, typename Value>
Dictionary<Key,Value>::node_rb_tree*
Dictionary<Key,Value>::rb_tree_insert(node_rb_tree* p,
    const Key& k, const Value& v) {
    if (p == nullptr)
        return new node_rb_tree(k, v);
    if (k < p->_k) {
        p->_left = rb_tree_insert(p->_left, k, v);
        // check if rotation/recoloring is needed
        node_rb_tree* y = p->_left; // y can't be nullptr
        node_rb_tree* z = p->_right;
        node_rb_tree* x = k < y->_k ? y->_left : y->_right;
        if (color(x) == red and color(y) == red) {
            if (color(z) == red) // case (a)
                else if (k < y->_k) // case (b1)
                else // y->_k < k, case (b2)
            }
        }
    else if (p->_k < k) { // symmetric case
        p->_right = rb_tree_insert(p->_right, k, v);
        ...
    }
    else // p->_k == k
        p->_v = v;
    return p;
}
```

Insertions in Red-black Trees

```
// case (a)
y -> _color = z -> _color = black;
p -> _color = red;

// case (b1)
p -> _color = red;
p = rotate_LL(p); // p points now to y
p -> _color = black;

// case (b2)
p -> _color = red;
p = rotate_LR(p);
p -> _color= black;
```

Red-black Trees: Summary

Operation	Worst-case Cost	# of Rotations
Lookup	$\Theta(\log n)$	None
Insertion	$\Theta(\log n)$	2*
Deletion	$\Theta(\log n)$	3

(*) A double rotation counts as 2 rotations

Part III

Disjoint Sets

Disjoint Sets

A set of *disjoint sets* or *partition* Π of a non-empty set \mathcal{A} is a collection of non-empty subsets $\Pi = \{\mathcal{A}_1, \dots, \mathcal{A}_k\}$ such that

- ① $i \neq j \implies \mathcal{A}_i \cap \mathcal{A}_j = \emptyset$
- ② $\mathcal{A} = \bigcup_{1 \leq i \leq k} \mathcal{A}_i$

Each \mathcal{A}_k is often called *block* or *class*; we might see a partition as an equivalence relation and each \mathcal{A}_k as one of its equivalence classes.

Disjoint Sets

Given a partition Π of \mathcal{A} , it induces an equivalence relation \equiv_Π

$$x \equiv_\Pi y \iff \text{there is } \mathcal{A}_i \in \Pi \text{ such that } x, y \in \mathcal{A}_i$$

Conversely, an equivalence relation of a non-empty set \mathcal{A} induces a partition $\Pi = \{\mathcal{A}_x\}_{x \in \mathcal{A}}$, with

$$\mathcal{A}_x = \{y \in \mathcal{A} \mid y \equiv x\}.$$

Disjoint Sets

Without loss of generality we will assume that the *support* \mathcal{A} of the disjoint sets is $\{1, \dots, n\}$ (or $\{0, 1, \dots, n - 1\}$). If that were not the case, we can have a **dictionary** to map the actual elements of \mathcal{A} into the range $\{1, \dots, n\}$.

We shall also assume that \mathcal{A} is **static**, that is, no elements are added or removed. Efficient representations for partitions of a dynamic set can be obtained with some extra but small effort.

Disjoint Sets

Two fundamental operations supported by a DISJOINTSETS abstract data type are:

- ➊ Given i and j , determine if the items i and j belong to the same block (class), or not. Alternatively, given an item i **find** the representative of the block (class) to which i belongs; i and j belong to the same block
 $\iff \text{Find}(i) = \text{Find}(j)$
- ➋ Given i and j , perform the **union** (a.k.a. **merge**) of the blocks of i and j into a single block; the operation might require i and j to be the representatives of their respective blocks

It is because of these two operations that these data structures are usually called **union-find** sets or **merge-find** sets (mfsets, for short).

Union-Find

```
class UnionFind {
public:
    // Creates the partition {{0}, {1}, ..., {n-1}}
    UnionFind(int n);

    // Returns the representative of the class to which
    // i belongs (should be const, but it is not to
    // allow path compression)
    int Find(int i);

    // Performs the union of the classes with representatives
    // ri and rj, ri ≠ rj
    void Union(int ri, int rj);

    // Returns the number of blocks in the union-find set
    int nr_blocks() const;
    ...
};
```

Implementation #1: Quick-find

- We represent the partition with a vector P :
 $P[i] =$ the representative of the block of i
- Initially, $P[i] = i$ for all i , $1 \leq i \leq n$
- $\text{FIND}(i)$ is trivial: just return $P[i]$
- For the union of two blocks with representatives ri and rj , simply scan the vector P and set all elements in the block of ri now belong to the block of rj , that is, set $P[k] := rj$ whenever $P[k] = ri$ (or vice-versa, transfer elements in the block rj to block ri)
- $\text{FIND}(i)$ is very cheap ($\Theta(1)$), but $\text{UNION}(ri, rj)$ is very expensive ($\Theta(n)$).

Implementation #1: Quick-find

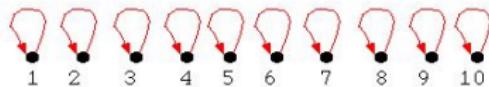
We can avoid scanning the entire array to perform a union; but we will still have to change the representative of all the elements in one block to point to the representative of the other block, and this has linear cost in the worst-case too.

Despite it is not very natural in this case, it is very convenient to think of the union-find set as a collection of trees, one tree per block, and see $P[i]$ as a pointer to the parent of i in its tree; $P[i] = i$ indicates that i is the root of the tree— i is the representative of the block. With *quick-find*, all trees have height 1 (blocks with a single item) or 2 (the representative is the root and all other items in the block are its children).

Implementation #1: Quick-find

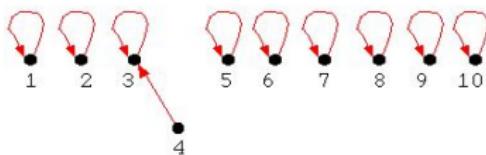
make(10)

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10



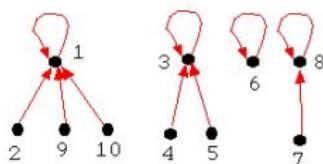
union(3, 4)

1	2	3	3	5	6	7	8	9	10
1	2	3	3	5	6	7	8	9	10



union(1, 2); union(4, 5); union(1, 9);
union(2, 10); union(8, 7)

1	1	3	3	3	6	8	8	1	1
1	1	3	3	3	6	8	8	1	1



Implementation #2: Quick-union

In **quick-union**, to merge two blocks with representatives ri and rj , it is enough to set $P[ri] := rj$ or $P[rj] := ri$. That makes **UNION(ri, rj)** trivial and cheap (cost is $\Theta(1)$).

If we allow **UNION(i, j)** with whatever i and j , we must find the corresponding representatives ri and rj , check that they are different and proceed as above. The operation can now be costly, but that's because of the calls to **FIND**.

A call **FIND(i)** can be expensive in the worst-case, it is proportional to the maximum height of the tree that contains i , and that can be as much as $\Theta(n)$.

Implementation #2: Quick-union

```
class UnionFind {
    ...
private:
    vector<int> P;
    int nr_blocks;
};

UnionFind::UnionFind(int n) : P(vector<int>(n)) {
    // constructor
    for (int j = 0; j < n; ++j)
        P[j] = j;
    nr_blocks = n;
}
void UnionFind::Union(int i, int j) {
    int ri = Find(i); int rj = Find(j);
    if (ri != rj) {
        P[ri] = rj; --nr_blocks;
    }
}
int UnionFind::Find(int i) {
    while (P[i] != i) i = P[i];
    return i;
}
```

Implementation #2: Quick-union

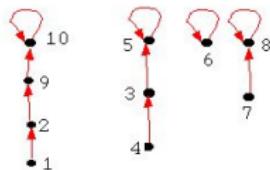
make(10); union(4, 3)

1	2	3	3	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10



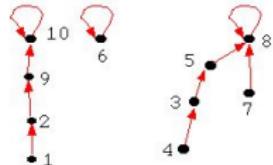
union(1,2); union(4,5); union(1,9);
union(2, 10); union(7,8)

2	9	5	3	5	6	8	8	10	10
1	2	3	4	5	6	7	8	9	10



union(4,7);

2	9	5	3	8	6	8	8	10	10
1	2	3	4	5	6	7	8	9	10



Implementation #3: Union by weight or by rank

To overcome the problem of unbalanced trees (leading to trees which are too high) it is enough to make sure that in a union

- ① The smaller tree becomes the child of the bigger tree (**union-by-weight**), or
- ② The tree with smaller rank becomes the child of the tree with larger rank (**union-by-rank**)

Unless we use **path compression** (stay tuned!) $rank \equiv height$.

Implementation #3: Union by weight or by rank

- To implement one of these two strategies we will need to know, for each block, its size (number of elements) or its rank (=height).
- We can use an auxiliary array to store that information. But we can avoid the extra space as follows: if i is the representative of its block, instead of setting $P[i] := i$ to mark it as the root we can have
 - 1 $P[i] = -\text{the size of the tree rooted at } i$
 - 2 $P[i] = -\text{the rank of the tree rooted at } i$

We use the negative sign to indicate that i is the root of a tree.

Implementation #3: Union by weight or by rank

```
class UnionFind {
    ...
private:
    vector<int> P;
    int nr_blocks;
};

UnionFind::UnionFind(int n) : P(vector<int>(n)) {
    // constructor
    for (int j = 0; j < n; ++j)
        P[j] = -1; // all items are roots of trees of size 1 (or rank 1)
    nr_blocks = n;
}
int UnionFind::Find(int i) {
    // P[i] < 0 when i is a root
    while (P[i] > 0) i = P[i];
    return i;
}
...
...
```

Implementation #3: Union by weight or by rank

```
void UnionFind::Union(int i, int j) {
    int ri = Find(i); int rj = Find(j);
    if (ri != rj) {
        if (P[ri] >= P[rj]) {
            // ri is the smallest/shortest
            P[ri] = rj;
            P[rj] += P[ri]; // <= union-by-weight
            // P[rj] = min(P[rj], P[ri]-1); // <= union-by-rank
        } else {
            // rj is the smallest/shortest
            P[rj] = ri;
            P[ri] += P[rj]; // <= union-by-weight
            // P[ri] = min(P[ri], P[rj]-1); // <= union-by-rank
        }
        --nr_blocks;
    }
}
```

Implementation #3: Union by weight or by rank

Lemma

The height of a tree that represents a block of size k is $\leq 1 + \log_2 k$, using union-by-weight.

Proof

We prove it by induction. If $k = 1$ the lemma is obviously true, the height of a tree of one element is 1. Let T be a tree of size k resulting from the union-by-weight of two trees T_1 and T_2 of sizes r and s , respectively, assume $r \leq s < k = r + s$. Then T has been obtained putting T_1 as child of T_2 .

Implementation #3: Union by weight or by rank

Proof (cont'd)

By inductive hypothesis, $\text{height}(T_1) \leq 1 + \log_2 r$ and $\text{height}(T_2) \leq 1 + \log_2 s$. The height of T is that of T_2 unless $\text{height}(T_1) = \text{height}(T_2)$, then $\text{height}(T) = \text{height}(T_1) + 1$. That is,

$$\begin{aligned}\text{height}(T) &= \max(\text{height}(T_2), \text{height}(T_1) + 1) \\ &\leq 1 + \max(\log_2 s, 1 + \log_2 r) \\ &= 1 + \max(\log_2 s, \log_2(2r)) \\ &\leq 1 + \log_2 k,\end{aligned}$$

since $s \leq k$ and $2r \leq r + s = k$.



Implementation #3: Union by weight or by rank

An analogous lemma can be proved if we perform union by rank.

We might be satisfied with union-by-rank or union-by-weight, but we can improve even further the cost of FIND applying some **path compression** heuristic.

Path Compression

While we look for the representative of i in a $\text{FIND}(i)$, we follow the pointers from i up to the root, and we could make the pointers along that path change so that the path becomes shorter and therefore we may speed up future calls to FIND . There are several heuristics for path compression:

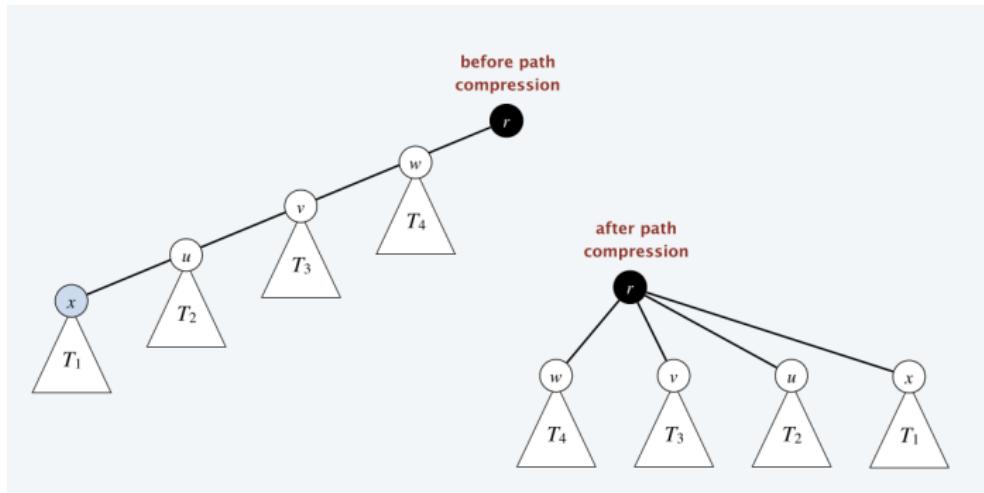
- ➊ In **full path compression**, we traverse the path from i to its representative twice: first to determine that ri is such representative; second, to set $P[k] := ri$ for all k along the path, as all these items have ri as their representative; this only doubles the cost of $\text{FIND}(i)$.
- ➋ In **path splitting**, we maintain two consecutive items in the path i_1 and $i_2 = P[i_1]$, then when we go up in the tree we make $P[i_1] := P[i_2]$; at the end of this traversal, all k along the path, except the root and its immediate child, will point to the element that was previously their grand-parent; we reduce the length of the path roughly by half.

Path Compression

- ③ In **path halving**, we traverse the path from i to its representative, making every other node point to its grand-parent.

Path compression: full path compression

Make every node point to its representative.



Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf>)

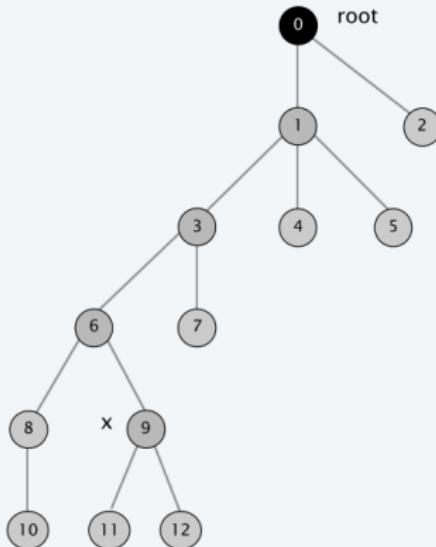
Path compression: full path compression

```
// iterative full path compression
// with the convention that P[i] = -the rank of i if P[i] < 0
int UnionFind::Find(int i) {
    int ri = i;
    // P[ri] < 0 when ri is a root
    while (P[ri] > 0) ri = P[ri];

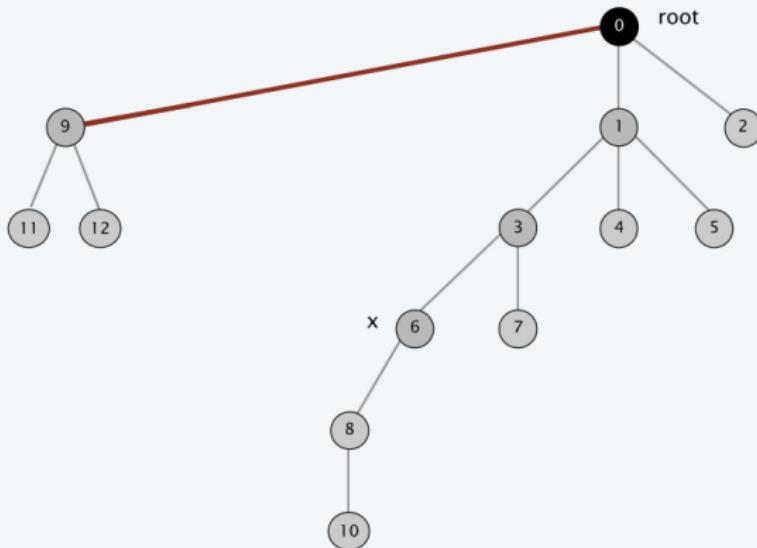
    // traverse the path again making everyone point to ri
    while (P[i] > 0) {
        int aux = i;
        i = P[i];
        P[aux] = ri;
    }
    return ri;
}

// recursive full path compression
// with the convention that P[i] = -the rank of i if P[i] < 0
int UnionFind::Find(int i) {
    if (P[i] < 0) return i;
    else {
        P[i] = Find(P[i]);
        return P[i];
    }
}
```

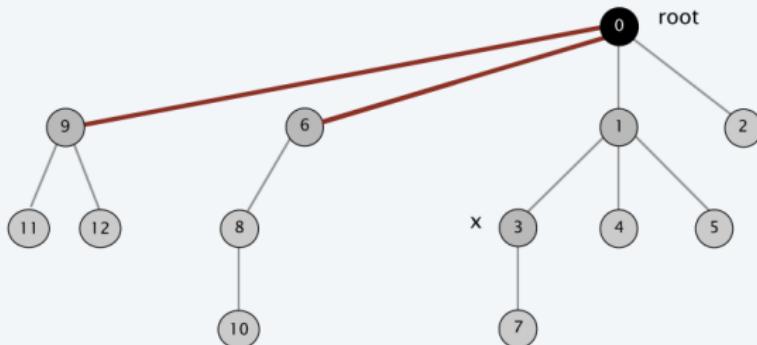
Path compression: full path compression



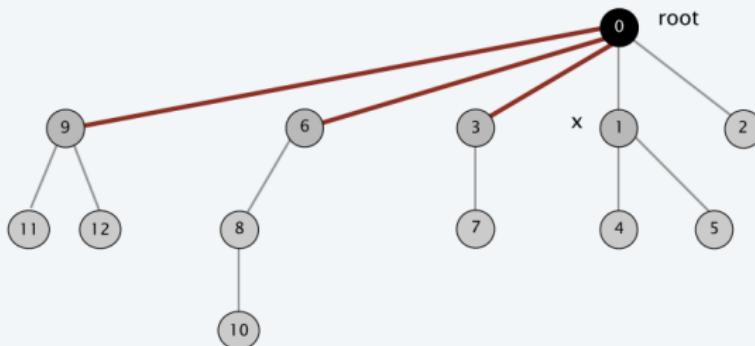
Path compression: full path compression



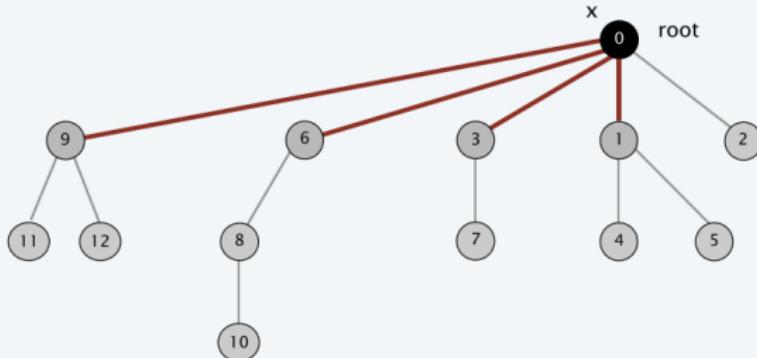
Path compression: full path compression



Path compression: full path compression

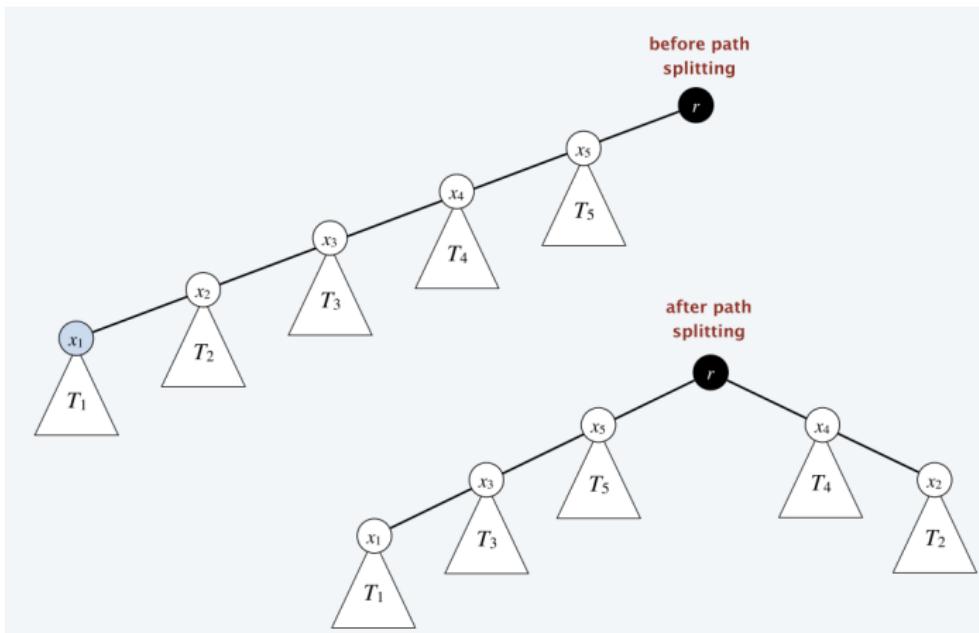


Path compression: full path compression



Path compression: path splitting

Make every node point to its grandparent (except if it is the root or a child of the root).

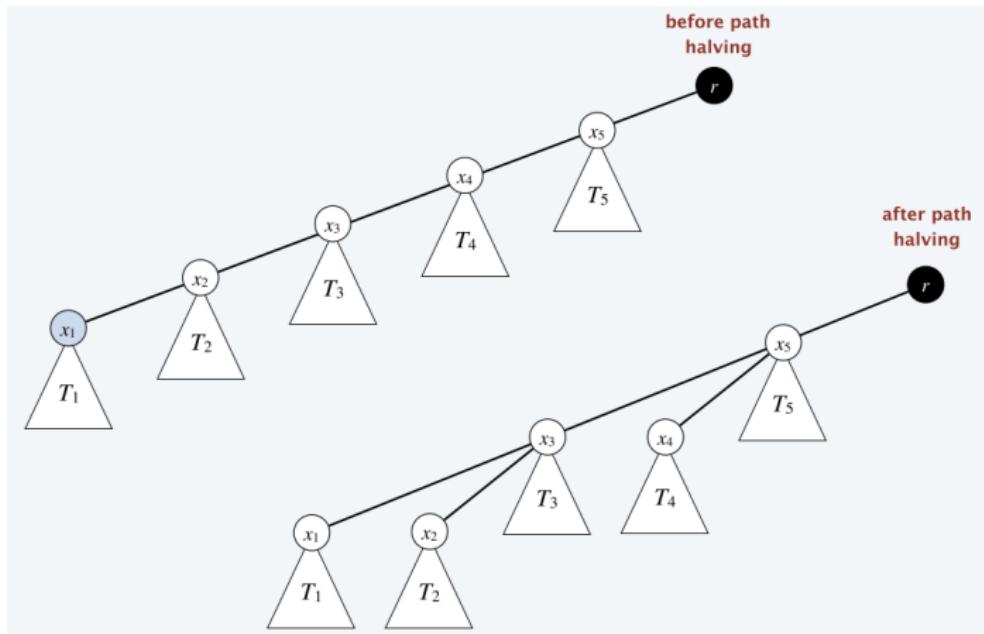


Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf>)

Path compression: path halving

Make every other node in the path point to its grandparent (except if it is the root or a child of the root).



Source: Kevin Wayne

(<https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/UnionFind.pdf>)

Amortized analysis of Union-Find

The analysis of Union-Find with union by weight (or by rank) using some path compression heuristic must be amortized: the union of two representatives (roots) is always cheap, and the cost of any FIND is bounded by $\mathcal{O}(\log n)$, but if we apply many FIND's the trees become bushier, and we approach rather quickly the situation of *Quick-Find* while we avoid costly UNION's.

In what follows we will analyze the cost of a sequence of m intermixed UNIONS and FINDS performed in a Union-Find data structure with $n \leq m$ elements, using **union-by-rank** and **full path compression**.

Similar results hold for the various combinations of union-by-weight/union-by-rank and path compression heuristics.

Amortized analysis of Union-Find

- Observation #1. Path compression does not change the rank of any node, hence $\text{rank}(x) > \text{height}(x)$ for any node x .
- In what follows we assume that we initialize the rank of all nodes to 0 and keep the ranks in a different array, so we will have:

```
int UnionFind::Find(int i) {
    if (P[i] == i) return i;
    else {
        P[i] = Find(P[i]);
        return P[i];
    }
}

void UnionFind::Union(int i, int j) {
    int ri = Find(i); int rj = Find(j);
    if (ri != rj) {
        if (rank[ri] <= rank[rj]) {
            P[ri] = rj;
            rank[rj] = max(rank[rj], 1+rank[ri]);
        } else {
            P[rj] = ri;
            rank[ri] = max(rank[ri], 1+rank[rj]);
        }
    }
}
```

Amortized analysis of Union-Find

Proposition

The tree roots, node ranks and elements within a tree are the same with or without path compression.

Proof

Path compression only changes some parent pointers, nothing else. It does not create new roots, does not change ranks or move elements from one tree to another. □

Amortized analysis of Union-Find

Properties:

- ① If x is not a root node then $\text{rank}(x) < \text{rank}(\text{parent}(x))$.
- ② If x is not a root node then its rank will not change.
- ③ Let $r_t = \text{rank}(\text{parent}(x))$ at time t . If at time $t + 1$ x changes its parent then $r_t < r_{t+1}$
- ④ A root node of rank k has $\geq 2^k$ descendants.
- ⑤ The rank of any node is $\leq \lceil \log_2 n \rceil$
- ⑥ For any $r \geq 0$, the Union-Find data structure contains at most $n/2^r$ of rank r

Amortized analysis of Union-Find

All the six properties hold for Union-Find with union-by-rank. By the previous proposition, properties #2, #4, #5 and #6 immediately hold for any variant using path compression

Only properties #1 and #3 might not hold as path compression makes changes to parent pointers. However, they still hold if we are doing path compression.

Amortized analysis of Union-Find

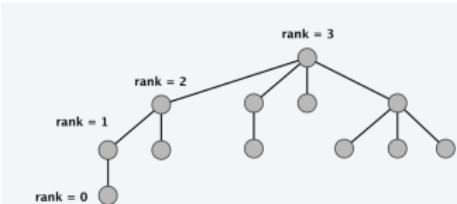
Proof of Property #1

A node of rank k can only be created by joining two nodes of rank $k - 1$. Path compression can't change ranks. However it might change the parent of x ; in that case, x will point to some ancestor of its previous parent, hence $\text{rank}(x) < \text{rank}(\text{parent}(x))$ at all times.



Proof of Property #2

The rank of a node can only change in union-by-rank if x was a root and becomes a non-root. Once a root becomes a non-root it will never become a root node again. Path compression never changes ranks and never changes roots.



Amortized analysis of Union-Find

Proof of Property #3

When the parent of x changes it is because either

- ① x becomes a non-root and union-by-rank guarantees that $r_t = \text{rank}(\text{parent}(x)) = \text{rank}(x)$ and $r_{t+1} > r_t$ as x becomes a child of a node whose rank is larger than r_t
- ② x is a non-root at time t but path compression changes its parent. Because x will be pointing to some ancestor of its parent then $r_{t+1} > r_t$ (because of Property #1)



Amortized analysis of Union-Find

Proof of Property #4

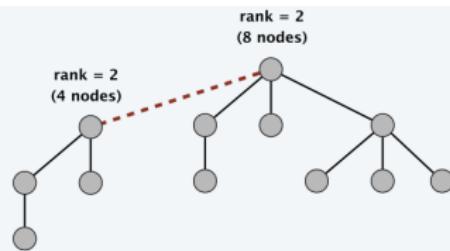
By induction on k .

Base: If $k = 0$ then x is the root of a tree with only one node, so the number of descendants is $\geq 2^k$.

Inductive hypothesis: a node x of rank k can only get that rank because of the union of two nodes of rank $k - 1$, hence x was the root of one of the trees involved and its rank was $k - 1$ before the union. By hypothesis, each tree contained $\geq 2^{k-1}$ and the result must then contain $\geq 2^k$ nodes. □

Proof of Property #5

Immediate from Properties #1 and #4. □

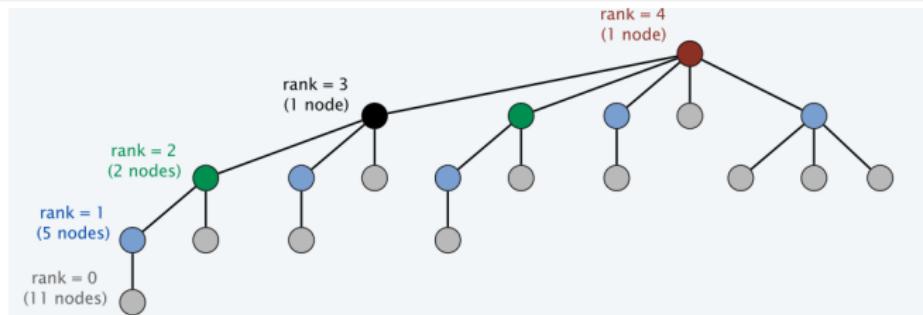


Amortized analysis of Union-Find

Proof of Property #6

Because of Property #4, any node x of rank k is the root of a subtree with $\geq 2^k$ nodes. Indeed, if x is a root that is the statement of the property. Else, inductively, because x had the property just before becoming a non-root; since neither its rank nor the set of descendants can change afterwards, the property is also true for non-root nodes. Because of Property #1, two distinct nodes of rank k can't be one ancestor of the other and then they can't have common ancestors.

Therefore, there can be at most $n/2^r$ nodes of rank r .



Amortized analysis of Union-Find

Definition

The **iterated logarithm** function is

$$\lg^* x = \begin{cases} 0, & \text{if } x \leq 1 \\ 1 + \lg^*(\lg x), & \text{otherwise} \end{cases}$$

We consider only logarithms base 2, hence write $\lg \equiv \log_2$.

n	$\lg^* n$	
(0, 1]	0	
(1, 2]	1	
(2, 4]	2	$\lg^* n \leq 5$ in this Universe.
(4, 16]	3	
(16, 65536]	4	
(65536, $2^{65536}]$	5	

Amortized analysis of Union-Find

Given k , let $2 \uparrow\uparrow k = \underbrace{2^2}_{k \text{ exponentiations}}^{\cdot 2^{\cdot \cdot \cdot}}.$ Inductively: $2 \uparrow\uparrow 0 = 1$, and
 $2 \uparrow\uparrow k = 2^{2 \uparrow\uparrow (k-1)}.$ Then $\lg(2 \uparrow\uparrow k) = k.$ Define groups

$$G_0 = \{1\}$$

$$G_1 = \{2\}$$

$$G_2 = \{3, 4\}$$

$$G_3 = \{5, \dots, 16\}$$

$$G_4 = \{17, \dots, 65536\}$$

$$G_5 = \{65537, \dots, 2^{65536}\}$$

$$\dots = \dots$$

$$G_k = \{1 + 2 \uparrow\uparrow (k-1), \dots, 2 \uparrow\uparrow k\}$$

For any $n > 0$, n belongs to $G_{\lg^* n}.$ The rank of any node in a Union-Find data structure of n elements will belong to one of the first $\lg^* n$ groups (as all ranks are between 0 and $\lg n$).

Amortized analysis of Union-Find

Accounting scheme: We assign credits during a UNION to the node that ceases to be a root; if its rank belongs to group G_k we assign $2 \uparrow\uparrow k$ to the item.

Proposition

The number of credits assigned in total among all nodes is $\leq n \lg^ n$.*

Proof

By Property #6, the number of nodes with rank $\geq x + 1$ is at most

$$\frac{n}{2^{x+1}} + \frac{n}{2^{x+2}} + \dots \leq \frac{n}{2^x}$$

Consider nodes that belong (their ranks) to group

$G_k = \{x + 1, \dots, 2^x\}$ ($x = 2 \uparrow\uparrow (k - 1)$). As the group contains $\leq 2^x$ nodes the number of credits assigned to nodes in the group is $\leq n$. All the ranks belong in the first $\lg^* n$ groups, hence the total number of credits is $\leq n \lg^* n$. □

Amortized analysis of Union-Find

The cost of **Union** is constant. We need to find the amortized cost of **Find**. The actual cost is the number of parent pointers followed:

- ① $\text{parent}(x)$ is a root \implies this is true for at most one of the nodes visited during the execution of a FIND,
- ② $\text{rank}(\text{parent}(x))$ belongs to a group G_j higher than $\text{rank}(x)$
 \implies this might happen at most for $\lg^* n$ visited x 's during the execution of a FIND
- ③ $\text{rank}(\text{parent}(x))$ and $\text{rank}(x)$ belong to the same group
 \implies see next slide

Amortized analysis of Union-Find

We make any node x such that $\text{rank}(\text{parent}(x))$ and $\text{rank}(x)$ are in the same group to pay 1 credit to follow and update the parent pointer during the FIND.

- Then $\text{rank}(\text{parent}(x))$ strictly increases (Property #1).
- If the node was in group $G = \{x + 1, \dots, 2^x\}$ then it had 2^x credits to spend and the rank of its parent will belong to a higher group before x has been updated 2^x times by FIND operations.
- Once the parent's rank belongs to a higher group than the rank of x , the situation will remain, as $\text{rank}(x)$ (hence the group) never changes and $\text{rank}(\text{parent}(x))$ never decreases.

Therefore x has enough credits to pay for all FIND's in which it gets involved before it becomes a node in Case #2.

Amortized analysis of Union-Find

Theorem

Starting from an initial Union-Find data structures for n elements with n disjoint blocks, any sequence of $m \geq n$ UNION and FIND using union-by-rank and full path compression have total cost $\mathcal{O}(m \lg^ n)$.*

Proof

The amortized cost of FIND is $\mathcal{O}(\lg^* n)$, and that of any UNION is constant, hence the sequence of m operations has total cost $\mathcal{O}(m \lg^* n)$. □

Amortized analysis of Union-Find



- 1972: Fischer: $\mathcal{O}(m \log \log n)$
- 1973: Hopcroft & Ullman: $\mathcal{O}(m \lg^* n)$
- 1975: Tarjan: $\mathcal{O}(m \alpha(m, n))$. Ackermann's inverse $\alpha(m, n)$ is an extremely slowly growing function
 $\alpha(m, n) \ll \lg^* n$
- 1984: Tarjan & van Leeuwen: $\mathcal{O}(m \alpha(m, n))$. For all combinations of union-by-weight/rank and path compression heuristics (full/splitting/halving).
- 1989: Fredman & Saks: $\Omega(m \alpha(m, n))$. A non-trivial lower bound for amortized complexity of Union-Find in the **cell probe** model.

Disjoint Sets

To learn more:

-  Michael J. Fischer
Efficiency of Equivalence Algorithms
Symposium on Complexity of Computer Computations,
IBM Thomas J. Watson Research Center, 1972.
-  J.E. Hopcroft and J.D. Ullman
Set Merging Algorithms
SIAM J. Computing 2(4):294–303, 1973.
-  Robert E. Tarjan
Efficiency of a Good But Not Linear Set Union Algorithm
J. ACM 22(2):215–225, 1975.

Disjoint Sets

To learn more:

-  Robert E. Tarjan and Jan van Leeuwen
Worst-Case Analysis of Set Union Algorithms
J. ACM 31(2):245–281, 1984.
-  Michael L. Fredman and Michael E. Saks
The Cell Probe Complexity of Dynamic Data Structures
Proc. 21st Symp. Theory of Computing (STOC), p.
345–354, 1989.
-  Z. Galil and G. Italiano
Data Structures and Algorithms for Disjoint Set Union
Problems
ACM Computing Surveys 23(3):319–344, 1991.

Part IV

Priority Queues

- 6 Introduction
- 7 Binary Heaps & Heapsort
- 8 Binomial Queues
- 9 Fibonacci Heaps

Part IV

Priority Queues

6

Introduction

7

Binary Heaps & Heapsort

- Heapsort

8

Binomial Queues

9

Fibonacci Heaps

Priority Queues: Introduction

A **priority queue** (cat: *cua de prioritat*; esp: *cola de prioridad*) stores a collection of elements, each one endowed with a value called its **priority**.

Priority queues support the insertion of new elements and the query and removal of an element of minimum (or maximum) priority.

Introduction

```
template <typename Elemt, typename Prio>
class PriorityQueue {
public:
    ...
    // Adds an element x with priority p to the priority queue.
    void insert(cons Elemt& x, const Prio& p);

    // Returns an element of minimum priority; throws an
    // exception if the queue is empty.
    Elemt min() const;

    // Returns the priority of an element of minimum priority; throws an
    // exception if the queue is empty.
    Prio min_prio() const;

    // Removes an element of minimum priority; throws an
    // exception if the queue is empty.
    void remove_min();

    // Returns true iff the priority queue is empty
    bool empty() const;
};
```

Priority Queues: Introduction

```
// We have two arrays Weight and Symb with the atomic
// weights and the symbols of n chemical elements, e.g.,
// Symb[i] = "C" y Weight[i] = 12.2, for some i.
// We use a priority queue to sort the information in alphabetic
// ascending order

PriorityQueue<double, string> P;
for (int i = 0; i < n; ++i)
    P.insert(Weigth[i], Symb[i]);
int i = 0;
while(not P.empty()) {
    Weight[i] = P.min();
    Symb[i] = P.min_prio();
    ++i;
    P.remove_min();
}
```

Priority Queues: Introduction

- Several techniques that used for the implementation of dictionaries can also be used for priority queues (not hash tables).
- For instance, balanced search trees such as AVLs can be used to implement a PQ with cost $\mathcal{O}(\log n)$ for insertions and deletions

Part IV

Priority Queues

6

Introduction

7

Binary Heaps & Heapsort

- Heapsort

8

Binomial Queues

9

Fibonacci Heaps

Heaps

Definition

A **heap** is a binary tree such that

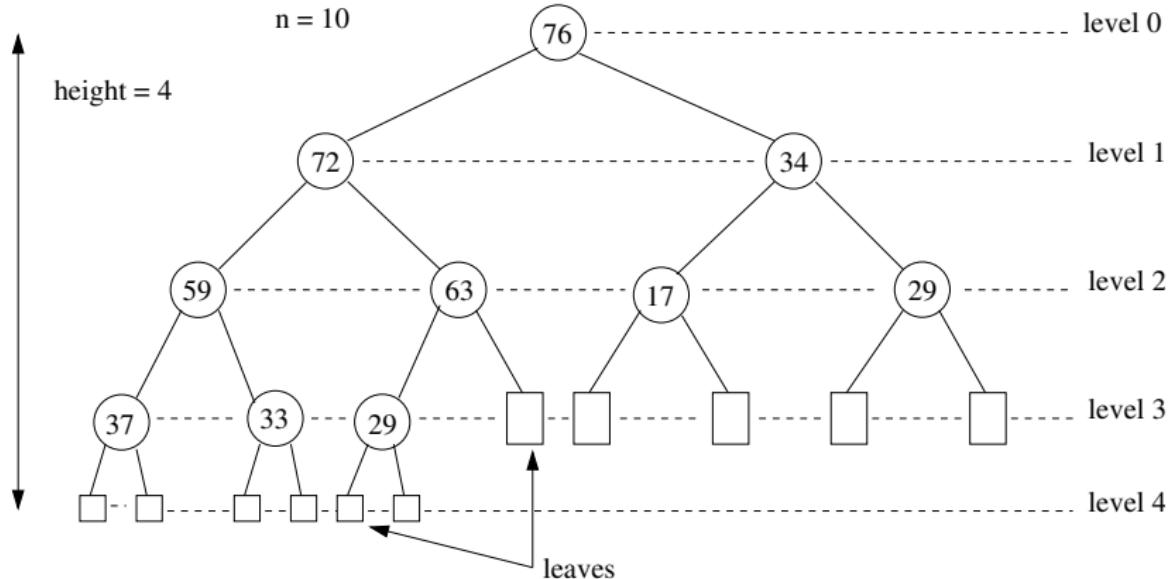
- ① All empty subtrees are located in the last two levels of the tree.
- ② If a node has an empty left subtree then its right subtree is also empty.
- ③ The priority of any element is larger or equal than the priority of any element in its descendants.

Heaps

Because of properties 1–2 in the definition, a heap is a **quasi-complete** binary tree. Property #3 is called **heap order** (for **max-heaps**).

If the priority of an element is smaller or equal than that of its descendants then we talk about **min-heaps**.

Heaps



Heaps

Proposition

- ① *The root of a max-heap stores an element of maximum priority.*
- ② *A heap of size n has height*

$$h = \lceil \log_2(n + 1) \rceil.$$

If heaps are used to implement a PQ the query for a max/min element and its priority is trivial: we need only to examine the root of the heap.

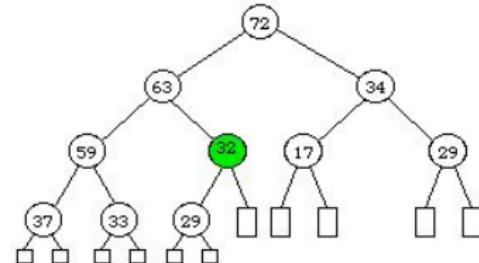
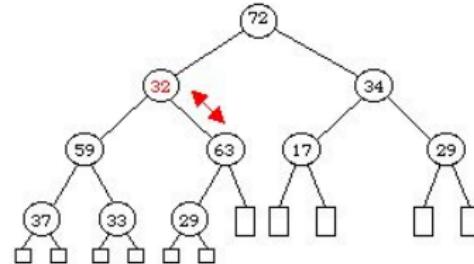
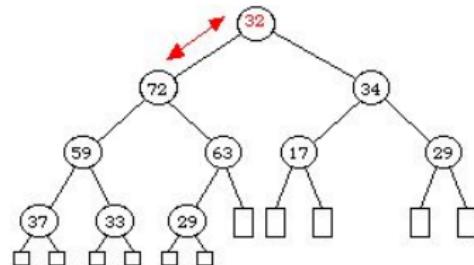
Heaps: Removing the maximum

- ➊ Replace the root of the heap with the last element (the rightmost element in the last level)
- ➋ Reestablish the invariant (heap order) **sinking** the root:
The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

Heaps: Removing the maximum

- ① Replace the root of the heap with the last element (the rightmost element in the last level)
- ② Reestablish the invariant (heap order) **sinking** the root:
The function `sink` exchanges a given node with its largest priority child, if its priority is smaller than the priority of its child, and repeats the same until the heap order is reestablished.

Heaps: Removing the maximum



Heaps: Adding a new element

- ① Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
- ② Reestablish the heap order **sifting up** (a.k.a. *floating*) the new added element:

The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

Heaps: Adding a new element

- ① Add the new element as rightmost node of the last level of the heap (or as the first element of a new deeper level)
- ② Reestablish the heap order **sifting up** (a.k.a. *floating*) the new added element:

The function `siftup` compares the given node to its father, and they are exchanged if its priority is larger than that of its father; the process is repeated until the heap order is reestablished.

The Cost of Heaps

Since the height of a heap is $\Theta(\log n)$, the cost of removing the maximum and the cost of insertions is $\mathcal{O}(\log n)$.

We can implement heaps with dynamically allocated nodes, and three pointers per node (left, right, father) ... But it is much easier and efficient to implement heaps with vectors!

Since the heap is a quasi-complete binary tree this allows us to avoid wasting memory: the n elements are stored in the first n components of the vector, which implicitly represent the tree.

Implementing Heaps

To make the rules easier we will use a vector A of size $n + 1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

- ➊ $A[1]$ contains the root
- ➋ If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i + 1 \leq n$ then $A[2i + 1]$ contains the right subtree of $A[i]$
- ➌ If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

To make the rules easier we will use a vector A of size $n + 1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

- ➊ $A[1]$ contains the root
- ➋ If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i + 1 \leq n$ then $A[2i + 1]$ contains the right subtree of $A[i]$
- ➌ If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

To make the rules easier we will use a vector A of size $n + 1$ and discard $A[0]$. Resizing can be used to allow unlimited growth.

- ① $A[1]$ contains the root
- ② If $2i \leq n$ then $A[2i]$ contains the left child of $A[i]$ and if $2i + 1 \leq n$ then $A[2i + 1]$ contains the right subtree of $A[i]$
- ③ If $i \geq 2$ then $A[i/2]$ contains the father of $A[i]$

Implementing Heaps

```
template <typename Ele, typename Prio>
class PriorityQueue {
public:
    ...
private:
    // Component of index 0 is not used
    vector<pair<Ele, Prio> > h;
    int nelems;

    void siftup(int j) throw();
    void sink(int j) throw();
};
```

Implementing Heaps

```
template <typename Elemt, typename Prio>
bool PriorityQueue<Elemt,Prio>::empty() const {
    return nelems == 0;
}

template <typename Elemt, typename Prio>
Elemt PriorityQueue<Elemt,Prio>::min() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    return h[1].first;
}

template <typename Elemt, typename Prio>
Prio PriorityQueue<Elemt,Prio>::min_prio() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    return h[1].second;
}
```

Implementing Heaps

```
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::insert(cons Elemt& x,
                                         cons Prio& p) {
    ++nelems;
    h.push_back(make_pair(x, p));
    siftup(nelems);
}

template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::remove_min() const {
    if (nelems == 0) throw EmptyPriorityQueue;
    swap(h[1], h[nelems]);
    --nelems;
    h.pop_back();
    sink(1);
}
```

Implementing Heaps

```
// Cost: O(log(n/j))
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::sink(int j) {

    // if j has no left child we are at the last level
    if (2 * j > nelems) return;

    int minchild = 2 * j;
    if (minchild < nelems and
        h[minchild].second > h[minchild + 1].second)
        ++minchild;

    // minchild is the index of the child with minimum priority
    if (h[j].second > h[minchild].second) {
        swap(h[j], h[minchild]);
        sink(minchild);
    }
}
```

Implementing Heaps

```
// Cost: O(log j)
template <typename Elemt, typename Prio>
void PriorityQueue<Elemt,Prio>::siftup(int j) {

    // if j is the root we are done
    if (j == 1) return;

    int father = j / 2;
    if (h[j].second < h[father].second) {
        swap(h[j], h[father]);
        siftup(father);
    }
}
```

Part IV

Priority Queues

- 6 Introduction
- 7 Binary Heaps & Heapsort
 - Heapsort
- 8 Binomial Queues
- 9 Fibonacci Heaps

Heapsort

Heapsort (Williams, 1964) sorts an array of n elements building a heap with the n elements and extracting them, one by one, from the heap (cif. our example of the atomic weights and chemical symbols).

The originally given array is used to build the heap; heapsort works **in-place** and only some constant auxiliary memory space is needed.

Since insertions and deletions in heaps have cost $\mathcal{O}(\log n)$ the cost of the algorithm is $\mathcal{O}(n \log n)$.

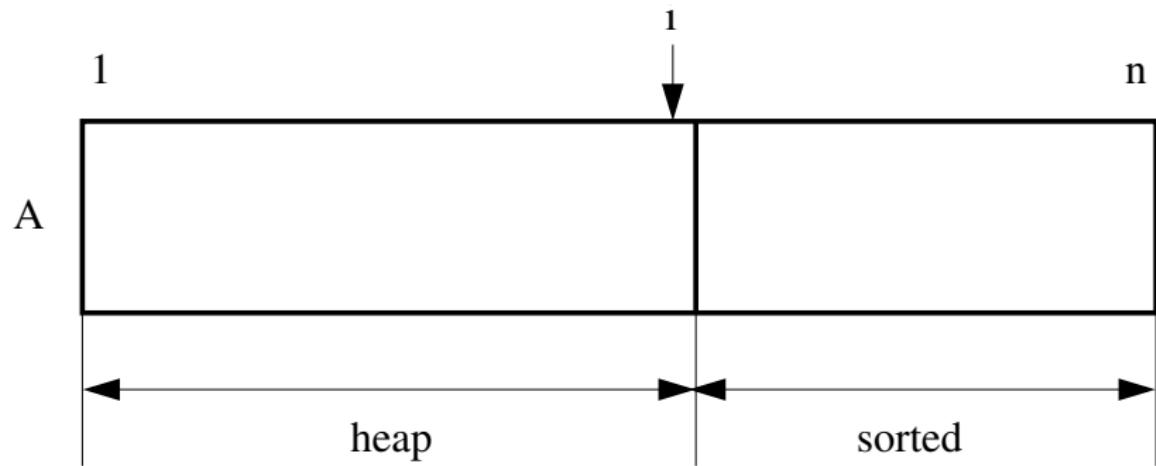
Heapsort

```
template <typename Elem>
void sink(Elem v[], int sz, int pos);

// Sort v[1..n] in ascending order
// (v[0] is unused)
template <typename Elem>
void heapsort(Elem v[], int n) {
    make_heap(v, n);
    for (int i = n; i > 0; --i) {
        // extract largest element from the heap
        swap(v[1], v[i]);

        // sink the root to reestablish max-heap order
        // in the subarray v[1..i-1]
        sink(v, i-1, 1);
    }
}
```

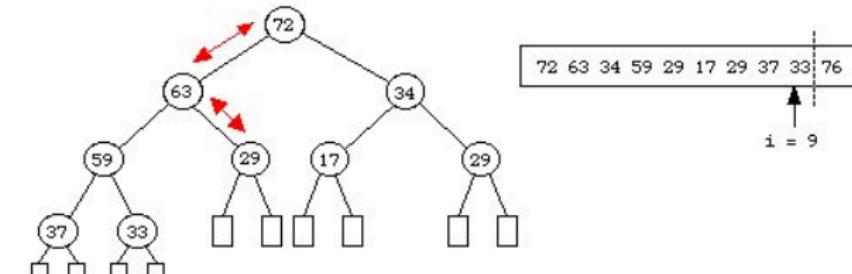
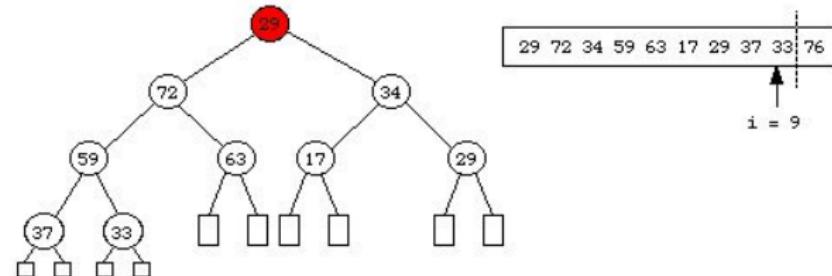
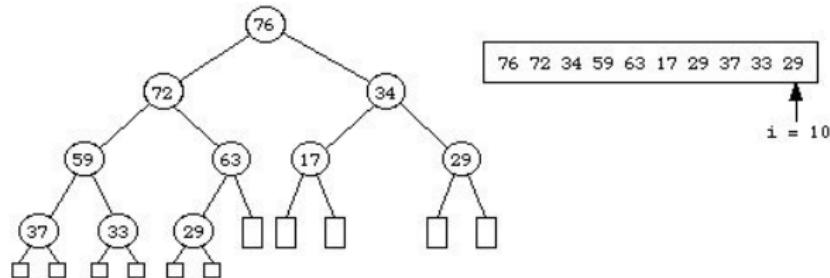
Heapsort



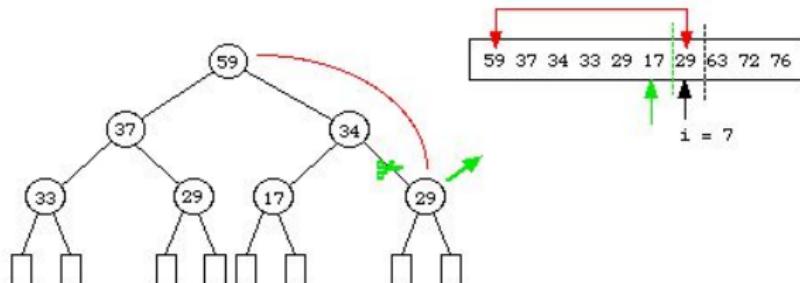
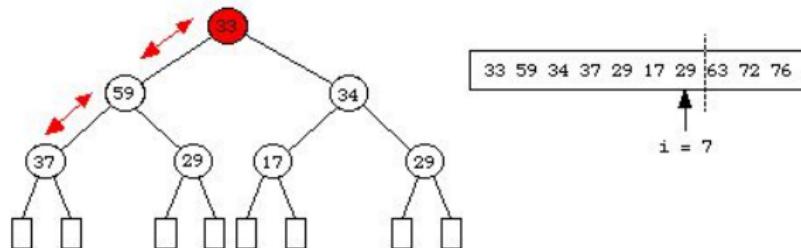
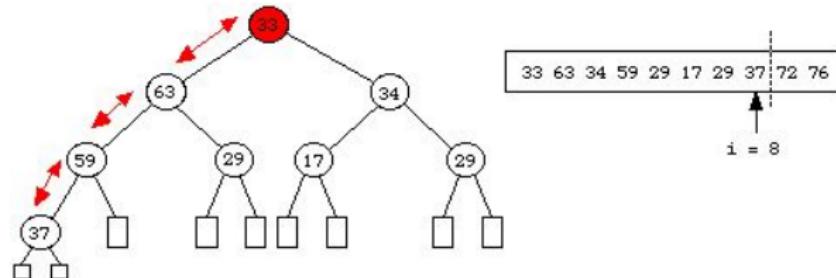
$$A[1] \leq A[i+1] \leq A[i+2] \leq \dots \leq A[n]$$

$$A[1] = \max_{1 \leq k \leq i} A[k]$$

Heapsort



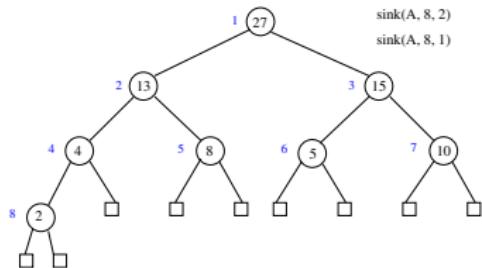
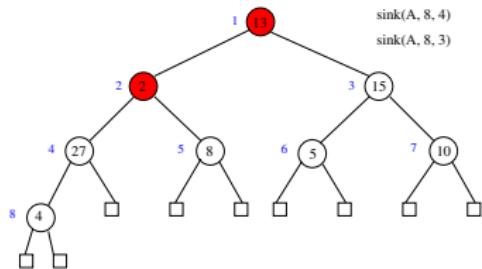
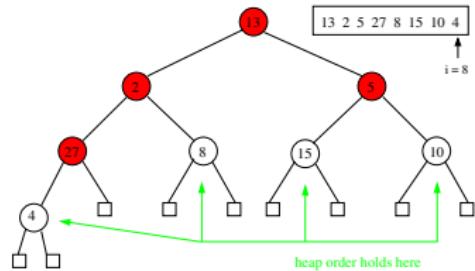
Heapsort



Heapify

```
// Establish (max) heap order in the
// array v[1..n] of Elem's; Elem == priorities
// this is a.k.a. as heapify
template <typename Elemt>
void make_heap(Elemt v[], int n) {
    for (int i = n/2; i > 0; --i)
        sink(v, n, i);
}
```

Heapify



The Cost of Heapsort

Let $H(n)$ be the worst-case cost of heapsort and $B(n)$ the cost `make_heap`. Since the worst-case cost of `sink($v, i - 1, 1$)` is $\mathcal{O}(\log i)$ we have

$$\begin{aligned} H(n) &= B(n) + \sum_{i=1}^{i=n} \mathcal{O}(\log i) \\ &= B(n) + \mathcal{O}\left(\sum_{1 \leq i \leq n} \log_2 i\right) \\ &= B(n) + \mathcal{O}(\log(n!)) = B(n) + \mathcal{O}(n \log n) \end{aligned}$$

A rough analysis of $B(n)$ shows that $B(n) = \mathcal{O}(n \log n)$ since it makes $\Theta(n)$ calls to `sink`, each one with cost $\mathcal{O}(\log n)$. Hence, $H(n) = \mathcal{O}(n \log n)$; actually, $H(n) = \Theta(n \log n)$ in any case if all elements are different.

The Cost of Heapify

A refined analysis of $B(n)$:

$$\begin{aligned} B(n) &= \sum_{1 \leq i \leq \lfloor n/2 \rfloor} \mathcal{O}(\log(n/i)) \\ &= \mathcal{O}\left(\log \frac{n^{n/2}}{(n/2)!}\right) \\ &= \mathcal{O}\left(\log(2e)^{n/2}\right) = \mathcal{O}(n) \end{aligned}$$

Since $B(n) = \Omega(n)$, we conclude $B(n) = \Theta(n)$.

The Cost of Heapify

Alternative proof: Let $h = \lceil \log_2(n + 1) \rceil$ the height of the heap.
Level $h - 1 - k$ contains at most

$$2^{h-1-k} < \frac{n+1}{2^k}$$

elements; in the worst-case each one will sink down to level
 $h - 1$ with cost $\mathcal{O}(k)$

The Cost of Heapify

$$\begin{aligned} B(n) &= \sum_{0 \leq k \leq h-1} \mathcal{O}(k) \frac{n+1}{2^k} \\ &= \mathcal{O}\left(n \sum_{0 \leq k \leq h-1} \frac{k}{2^k}\right) \\ &= \mathcal{O}\left(n \sum_{k \geq 0} \frac{k}{2^k}\right) = \mathcal{O}(n), \end{aligned}$$

since

$$\sum_{k \geq 0} \frac{k}{2^k} = 2.$$

In general, if $0 < |r| < 1$,

$$\sum_{k \geq 0} k \cdot r^k = \frac{r}{(1-r)^2}.$$

The Cost of Heapify

Despite $H(n) = \Theta(n \log n)$, the refined analysis of $B(n)$ is important: using a *min-heap* we can get the smallest k elements in an array in ascending order with cost:

$$\begin{aligned} S(n, k) &= B(n) + k \cdot \mathcal{O}(\log n) \\ &= \mathcal{O}(n + k \log n). \end{aligned}$$

If $k = \mathcal{O}(n / \log n)$ then $S(n, k) = \mathcal{O}(n)$.

Part IV

Priority Queues

6

Introduction

7

Binary Heaps & Heapsort

- Heapsort

8

Binomial Queues

9

Fibonacci Heaps

Binomial Queues



J. Vuillemin

- A **binomial queue** is a data structure that efficiently supports the standard operations of a **priority queue** (`insert`, `min`, `extract_min`) and additionally it supports the **melding** (merging) of two queues in time $\mathcal{O}(\log n)$.
- Note that melding two ordinary heaps takes time $\mathcal{O}(n)$.
- Binomial queues (aka *binomial heaps*) were invented by J. Vuillemin in 1978.

```
template <typename Elemt, typename Prio>
class PriorityQueue {
public:
    PriorityQueue() throw(error);
    ~PriorityQueue() throw();
    PriorityQueue(const PriorityQueue& Q) throw(error);
    PriorityQueue& operator=(const PriorityQueue& Q) throw(error);

    // Add element x with priority p to the priority queue
    void insert(cons Elemt& x, const Prio& p) throw(error)

    // Returns an element of minimum priority. Throws an exception if
    // the priority queue is empty
    Elemt min() const throw(error);

    // Returns the minimum priority in the queue. Throws an exception
    // if the priority queue is empty
    Prio min_prio() const throw(error);

    // Removes an element of minimum priority from the queue. Throws
    // an exception if the priority queue is empty
    void remove_min() throw(error);

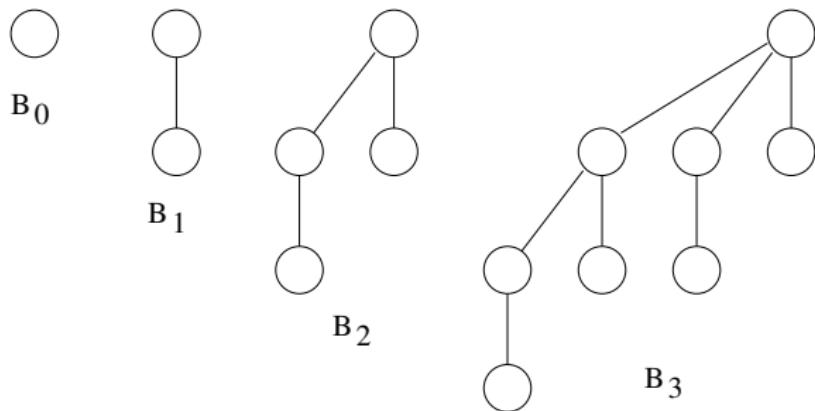
    // Returns true if and only if the queue is empty
    bool empty() const throw();

    // Melds (merges) the priority queue with the priority queue Q;
    // the priority queue Q becomes empty
    void meld(PriorityQueue& Q) throw();

    ...
};
```

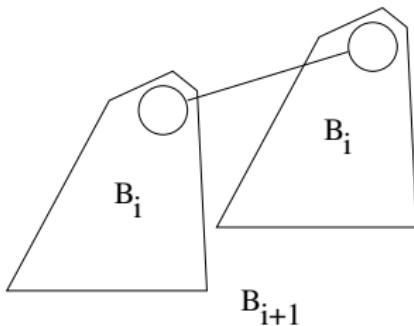
Binomial Queues

- A binomial queue is a collection of **binomial trees**.
- The binomial tree of order i (called B_i) contains 2^i nodes



Binomial Queues

- A binomial tree of order $i + 1$ is (recursively) built by planting a binomial tree B_i as a child of the root of another binomial tree B_i .



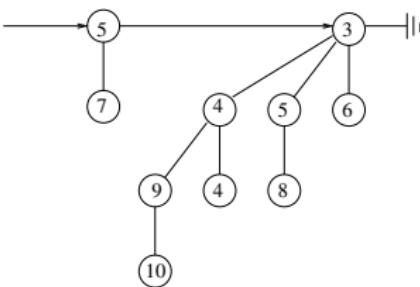
- The size of B_i is 2^i ; indeed $|B_0| = 2^0 = 1$,
 $|B_{i+1}| = 2 \cdot |B_i| = 2 \cdot 2^i = 2^{i+1}$
- A binomial tree of order i has exactly $\binom{i}{k}$ descendants at level k (the root is at level 0); hence their name
- A binomial tree of order i has height $i = \log_2 |B_i|$

Binomial Queues

- Let $(b_{k-1}, b_{k-2}, \dots, b_0)_2$ be the binary representation of n . Then a binomial queue for a set of n elements contains b_0 binomial trees of order 0, b_1 binomial trees of order 1, \dots , b_j binomial trees of order j , \dots

$$n = 10 = (1,0,1,0)_2$$

esempio sbagliato:
andrebbe sorted
dall'order dei children.
L'albero 5,7 andrebbe
a destra di quello con
3

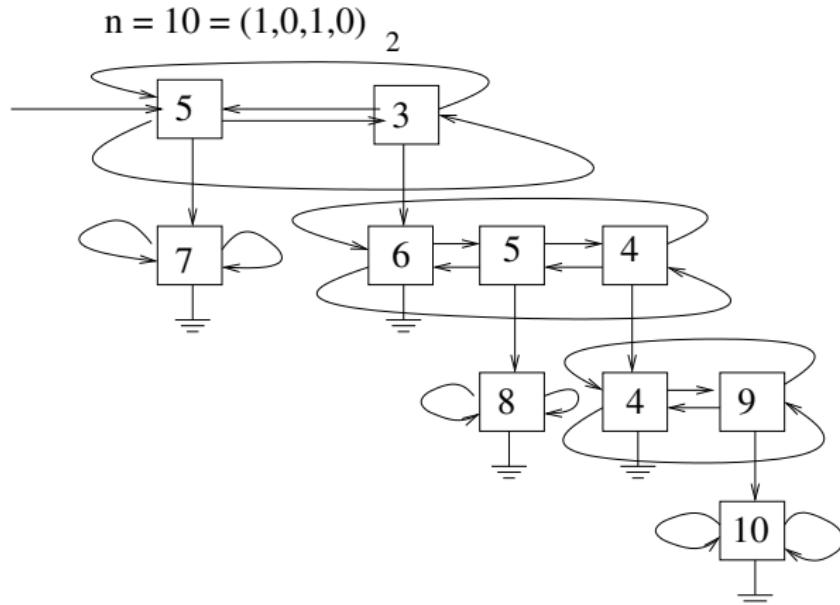


- A binomial queue for n elements contains at most $\lceil \log_2(n + 1) \rceil$ binomial trees
- The n elements of the binomial queue are stored in the binomial trees in such a way that **each binomial tree satisfies the heap property**: the priority of the element at any given node is \leq than the priority of its descendants

Binomial Queues

- Each node in the binomial queue will store an `Elem` and its priority (any type that admits a total order)
- Each node will also store the order of the binomial subtree of which the node is the root
- We will use the usual *first-child/next-sibling* representation for general trees, with a twist: the list of children of a node will be double linked and circularly closed
- We need thus three pointers per node: `first_child`, `next_sibling`, `prev_sibling`
- The binomial queue is simply a pointer to the root of the first binomial tree
- We will impose that all lists of children are in increasing order

Binomial Queues



Binomial Queues

```
template <typename Elem, typename Prio>
class PriorityQueue {
    ...
private:
    struct node_bq {
        Elem _info;
        Prio _prio;
        int _order;
        node_bq* _first_child;
        node_bq* _next_sibling;
        node_bq* _prev_sibling;
        node_bq(const Elem& x, const Prio& p, int order = 0) : _info(x), _prio(p),
                                                               _order(order), _first_child(NULL) {
            _next_sibling = _prev_sibling = this;
        };
        node_bq* _first;
        int _nelems;
    };
}
```

Binomial Queues

- To locate an element of minimum priority it is enough to visit the roots of the binomial trees; the minimum of each binomial tree is at its root because of the heap property.
- Since there are at most $\lceil \log_2(n + 1) \rceil$ binomial trees, the methods `min()` and `min_prio()` take $\mathcal{O}(\log n)$ time and both are very easy to implement.

Binomial Queues

- We can also keep a pointer to the root of the element with minimum priority, and update it after each insertion or removal, when necessary. The complexity of updates does not change and `min()` and `min_prio()` take $\mathcal{O}(1)$ time

Binomial Queues

```
static node_bq* min(node_bq* f) const throw(error) {
    if (f == NULL) throw error(EmptyQueue);
    Prio minprio = f -> _prio;
    node_bq* minelem = f;
    node_bq* p = f-> _next_sibling;
    while (p != f) {
        if (p -> _prio < minprio) {
            minprio = p -> _prio;
            minelem = p;
        };
        p = p -> _next_sibling;
    }
    return minelem;
}

Elem min() const throw(error) {
    return min(_first) -> _info;
}

Prio min_prio() const throw(error) {
    return min(_first) -> _prio;
}
```

Binomial Queues

- To insert a new element x with priority p , a binomial queue with just that element is trivially built and then the new queue is melded with the original queue
- If the cost of melding two queues with a total number of items n is $M(n)$, then the cost of insertions is $\mathcal{O}(M(n))$

Binomial Queues

```
void insert(const Elem& x, const Prio& p) throw(error) {
    node_bq* nn = new node_bq(x, p);
    _first = meld(_first, nn);
    ++_nelems;
}
```

Binomial Queues

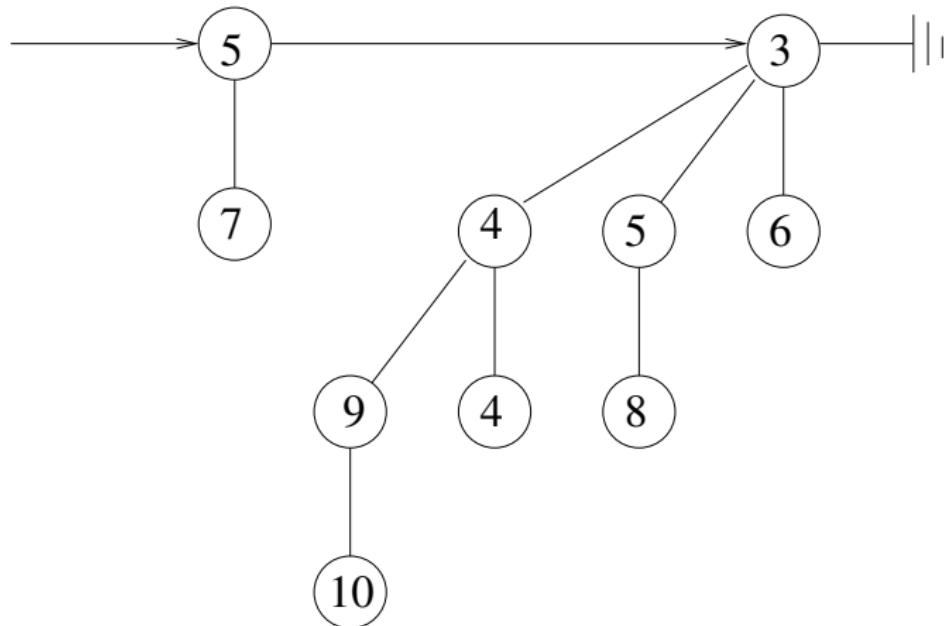
- To delete an element of minimum priority from a queue Q , we start locating such an element, say x ; it must be at the root of some B_i
- The root of B_i is detached from Q and thus B_i is no longer part of the original queue Q ; the list of x 's children is a binomial queue Q' with $2^i - 1$ elements
- The queue Q' has i binomial trees of orders 0, 1, 2, ... up to $i - 1$

$$1 + 2 + \dots + 2^{i-1} = 2^i - 1$$

- The queue $Q \setminus B_i$ is then melded with Q'

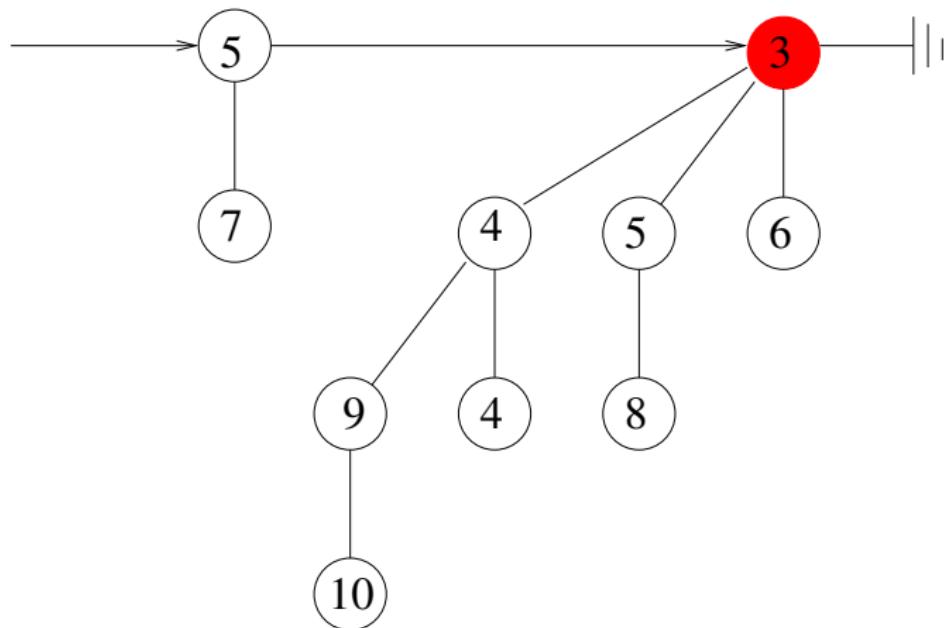
Binomial Queues

$$n = 10 = (1,0,1,0)_2$$



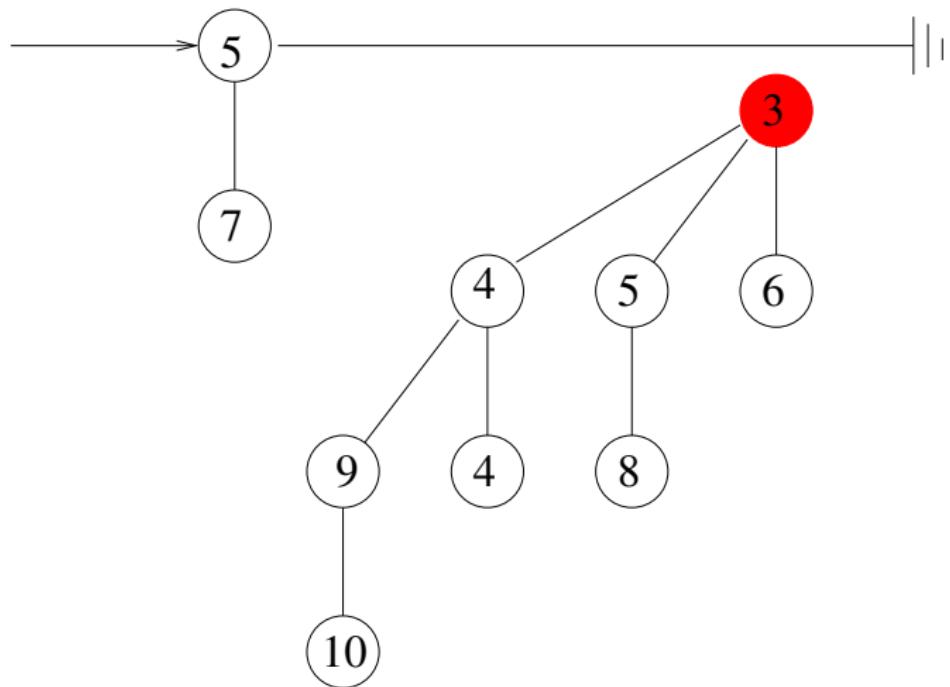
Binomial Queues

$$n = 10 = (1,0,1,0)_2$$



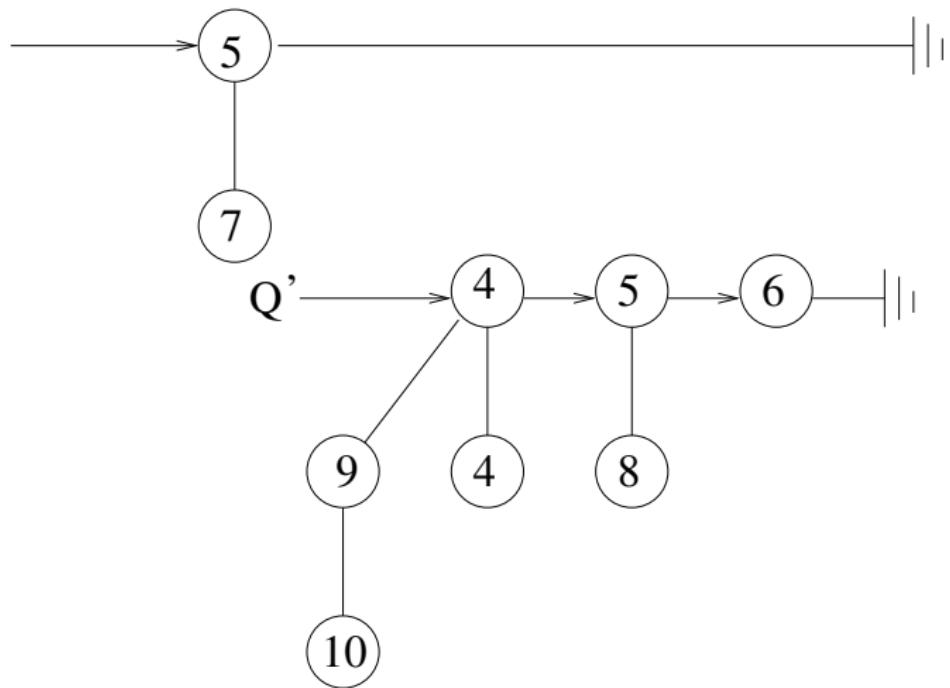
Binomial Queues

$$n = 10 = (1,0,1,0)_2$$



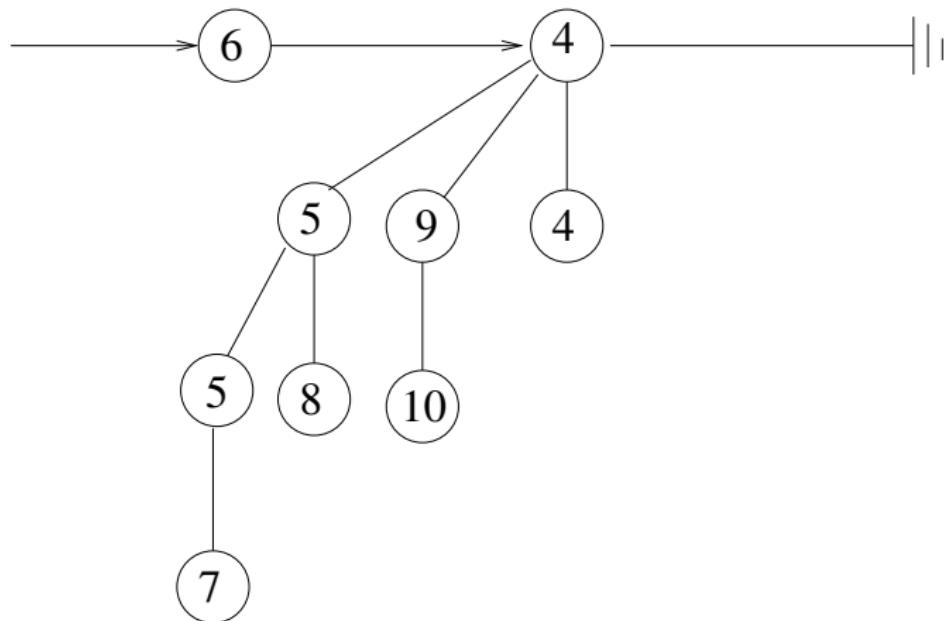
Binomial Queues

$$n = 10 = (1,0,1,0)_2$$



Binomial Queues

$$n = 9 = (1,0,0,1)_2$$



Binomial Queues

```
void remove_min() throw(error) {
    node_bq* m = min(_first);
    node_bq* children = m -> _first_child;
    if (m != m -> _next_sibling) { // there is more than one
        // binomial tree
        m -> _prev_sibling -> _next_sibling = m -> _next_sibling;
        m -> _next_sibling -> _prev_sibling = m -> _prev_sibling;
    } else {
        _first = NULL;
    }
    node_bq* qaux = m -> _first_child;
    m -> _first_child = m -> _next_sibling = m -> _prev_sibling = NULL;
    delete m;
    _first = meld(_first, qaux);
    --_nelems;
}
```

Binomial Queues

- The cost of extracting an element of minimum priority:
 - To locate the minimum priority has cost $\mathcal{O}(\log n)$
 - Melding $Q \setminus B_i$ and Q' has cost $\mathcal{O}(M(n))$, since
$$|Q \setminus B_i| + |Q'| = n - 2^i + 2^i - 1 = n - 1$$
- In total: $\mathcal{O}(\log n + M(n))$

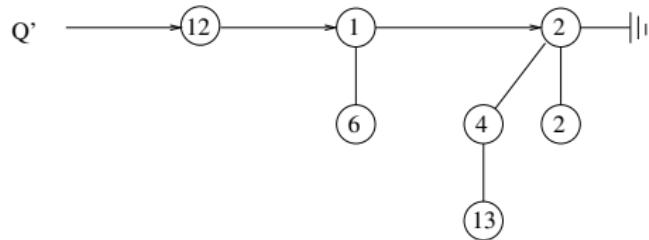
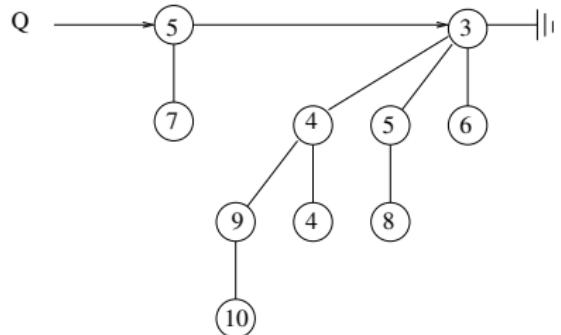
Binomial Queues

- Melding two binomial queues Q and Q' is very similar to the addition of two binary numbers bitwise
- The procedure iterates along the two lists of binomial trees; at any given step we consider two binomial trees B_i and B'_j , and a *carry* $C = B''_k$ or $C = \emptyset$

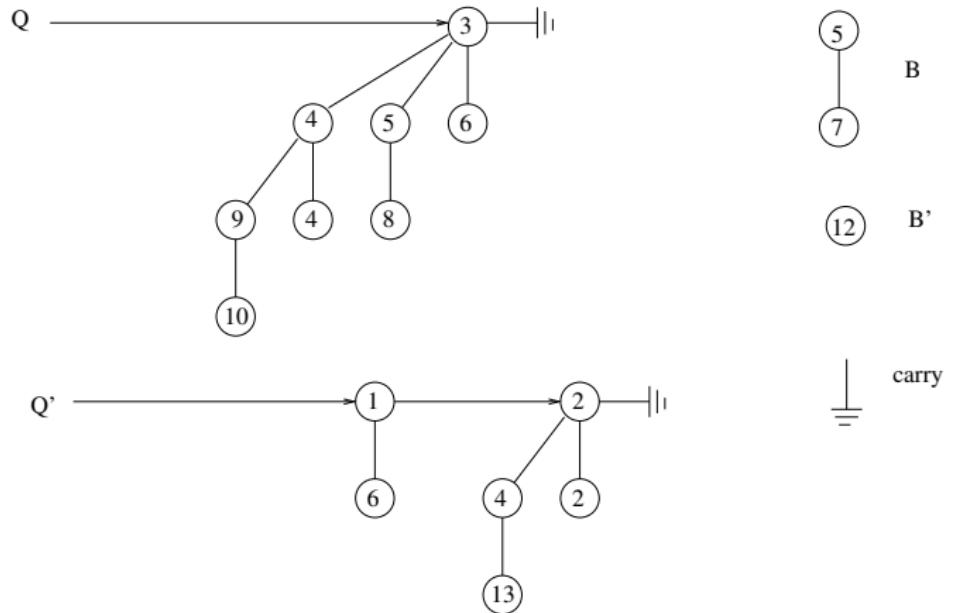
Binomial Queues

- Let $r = \min(i, j, k)$.
 - If there is only one binomial tree in $\{B_i, B'_j, C\}$ of order r , put that binomial tree in the result and advance to the next binomial tree in the corresponding queue (or set $C = \emptyset$)
 - If exactly two binomial trees in $\{B_i, B'_j, C\}$ are of order r , set $C = B_{r+1}$ by joining the two binomial trees (while preserving the heap property), remove the binomial trees from the respective queues, and advance to the next binomial tree where appropriate
 - If the three binomial trees are of order r , put B''_k in the result, remove B_i from Q and B'_j from Q' , set $C = B_{r+1}$ by joining B_i and B'_j , and advance in both Q and Q' to the next binomial trees

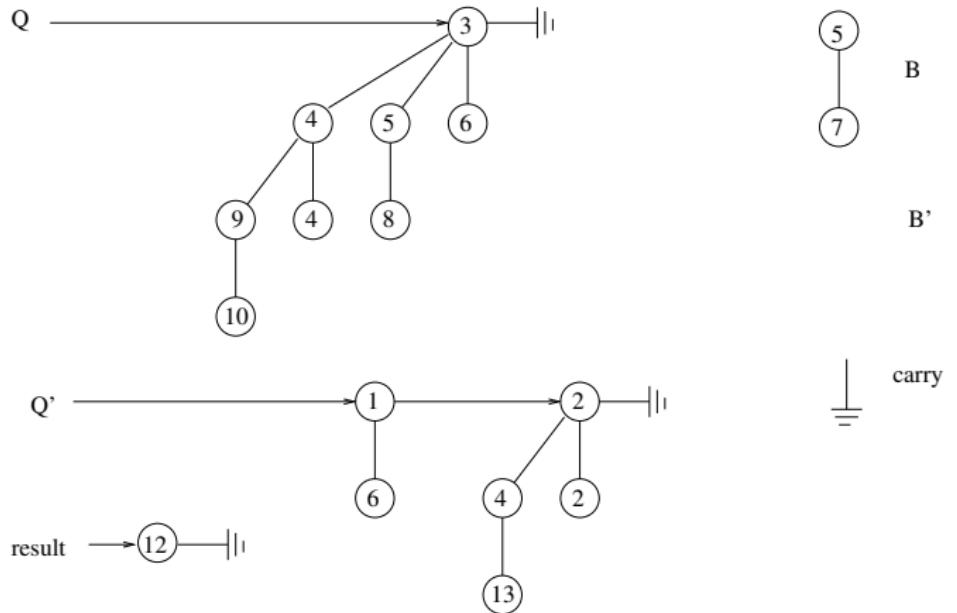
Binomial Queues



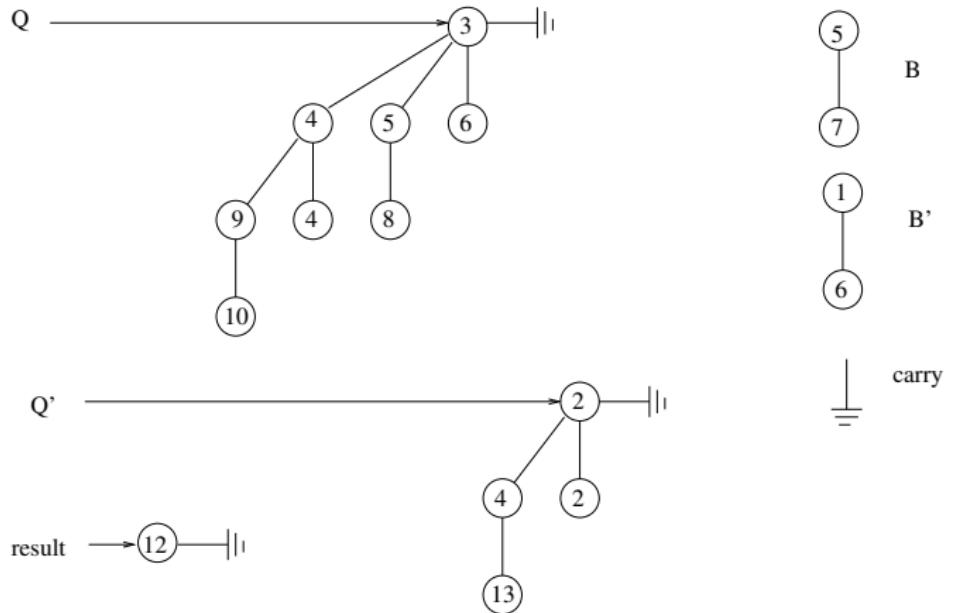
Binomial Queues



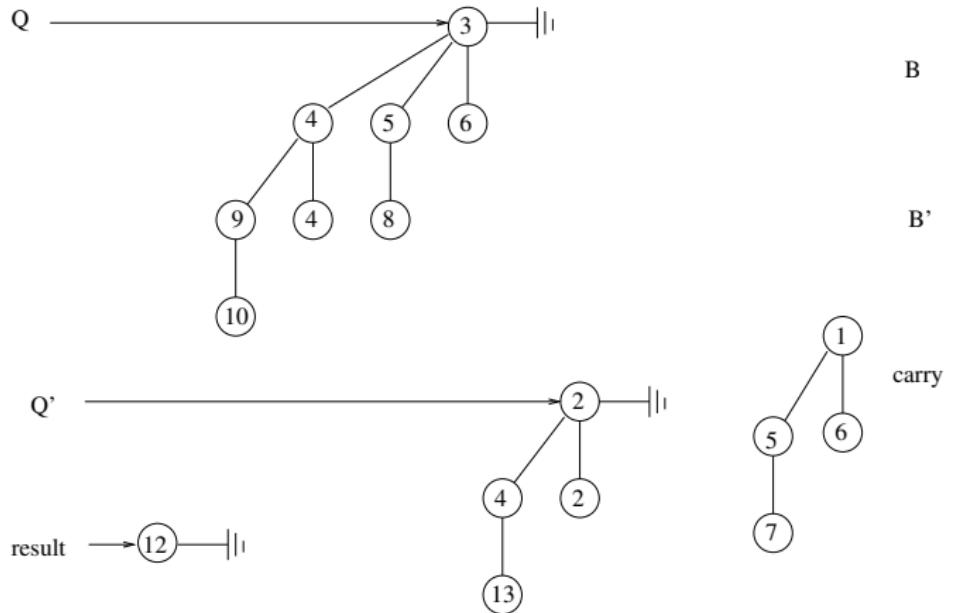
Binomial Queues



Binomial Queues

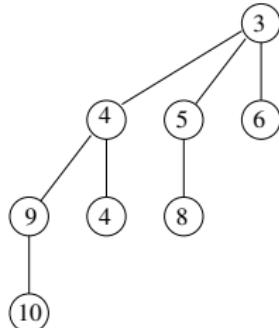


Binomial Queues



Binomial Queues

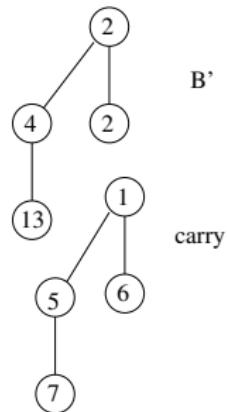
Q → ||



Q' → ||

result → 12 → ||

B

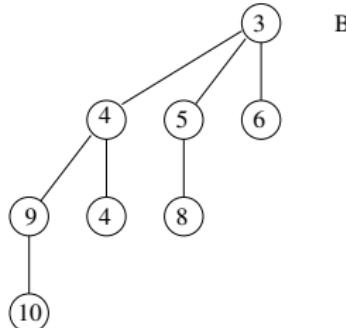


B'

carry

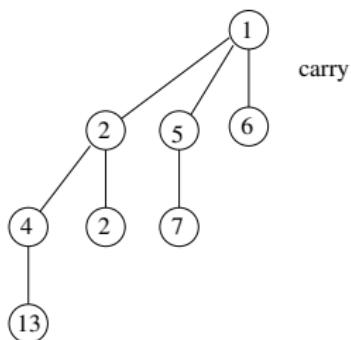
Binomial Queues

Q → ||



B

Q' → ||



B'

result → 12 → ||

carry

Binomial Queues

Q —————||—

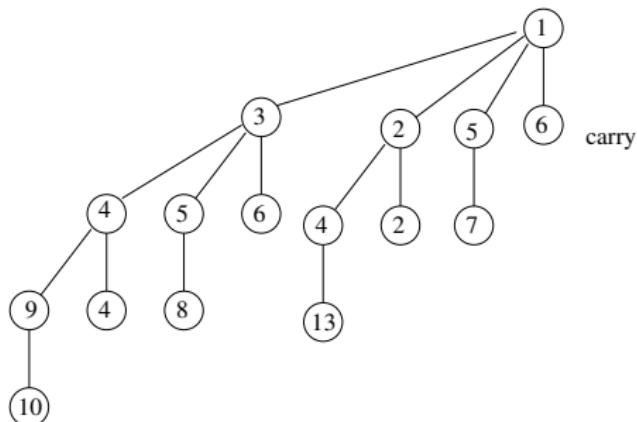
B

B'

Q' —————||—

result → 12 —————||—

carry



Binomial Queues

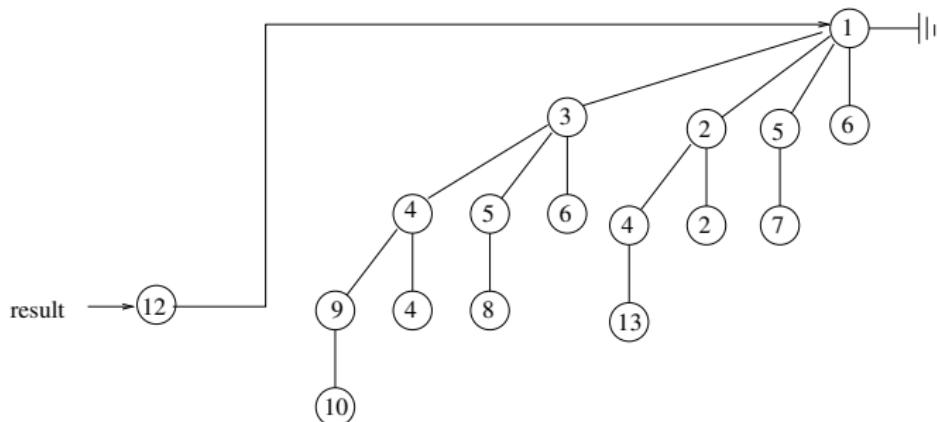
Q —————|||

B

Q' —————|||

B'

carry



Binomial Queues

```
// removes the first binomial tree from the binomial queue q
// and returns it; if the queue q is empty, returns NULL: cost: Theta(1)
static node_bq* pop_front(node_bq*& q) throw() {

    // adds the binomial queue b (typically consisting of a single tree)
    // at the end of the binomial queue q;
    // does nothing if b == NULL; cost: Theta(1)
    static void append(node_bq*& q, node_bq* b) throw() {

        // melds Q and Qp, destroying the two binomial queues
        static node_bq* meld(node_bq*& Q, node_bq*& Qp) throw() {
            node_bq* B = pop_front(Q);
            node_bq* Bp = pop_front(Qp);
            node_bq* carry = NULL;
            node_bq* result = NULL;
            while (non-empty(B, Bp, carry) >= 2) {
                node_bq* s = add(B, Bp, carry);
                append(result, s);
                if (B == NULL) B = pop_front(Q);
                if (Bp == NULL) Bp = pop_front(Qp);
            }
            // append the remainder t othe result
            append(result, Q);
            append(result, Qp);
            append(result, carry);
            return result;
        }
    }
}
```

Binomial Queues

```
static node_bq* add(node_bq*& A, node_bq*& B, node_bq*& C) throw() {
    int i = order(A); int j = order(B); int k = order(C);
    int r = min(i, j, k);
    node_bq* a, b, c;
    a = b = c = NULL;
    if (i == r) { a = A; A = NULL; }
    if (j == r) { b = B; B = NULL; }
    if (k == r) { c = C; C = NULL; }
    if (a != NULL and b == NULL and c == NULL) {
        return a;
    }
    if (a == NULL and b != NULL and c == NULL) {
        return b;
    }
    if (a == NULL and b == NULL and c != NULL) {
        return c;
    }
    if (a != NULL and b != NULL and c == NULL) {
        C = join(a, b);
        return NULL;
    }
    if (a != NULL and b == NULL and c != NULL) {
        C = join(a,c);
        return NULL;
    }
    if (a == NULL and b != NULL and c != NULL) {
        C = join(b,c);
        return NULL;
    }
    // a != NULL and b != NULL and c != NULL
    C = join(a,b);
    return c;
}
```

Binomial Queues

```
static int order(node_bq* q) throw() {
    // no binomial queue will ever be of order as high as 256 ...
    // unless it had 2^256 elements, more than elementary particles in
    // this Universe; to all practical purposes 256 = infinity
    return q == NULL ? 256 : q -> _order;
}

// plants p as rightmost child of q or q as rightmost child of p
// to obtain a new binomial tree of order + 1 and preserving
// the heap property
static node_bq* join(node_bq* p, node_bq* q) {
    if (p -> _prio <= q -> _prio) {
        push_back(p -> _first_child, q);
        ++p -> _order;
        return p;
    } else {
        push_back(q -> _first_child, p);
        ++q -> _order;
        return q;
    }
}
```

Binomial Queues

- Melding two queues with ℓ and m binomial trees each, respectively, has cost $\mathcal{O}(\ell + m)$ because the cost of the body of the iteration is $\mathcal{O}(1)$ and each iteration removes at least one binomial tree from one of the queues
- Suppose that the queues to be melded contain n elements in total; hence the number of binomial trees in Q is $\leq \log n$ and the same is true for Q' , and the cost of `meld` is $M(n) = \mathcal{O}(\log n)$
- The cost of inserting a new element is $\mathcal{O}(M(n))$ and the cost of removing an element of minimum priority is

$$\mathcal{O}(\log n + M(n)) = \mathcal{O}(\log n)$$

Binomial Queues

- Note that the cost of inserting an item in a binomial queue of size n is $\Theta(\ell_n + 1)$ where ℓ_n is the weight of the rightmost zero in the binary representation of n .
- The cost of n insertions

$$\begin{aligned} \sum_{0 \leq i < n} \Theta(\ell_i + 1) &= \sum_{r=1}^{\lceil \log_2(n+1) \rceil} \Theta(r) \cdot \frac{n}{2^r} \\ &\leq n \Theta \left(\sum_{r \geq 0} \frac{r}{2^r} \right) = \Theta(n), \end{aligned}$$

as $\approx n/2^r$ of the numbers between 0 and $n - 1$ have their rightmost zero at position r , and the infinite series in the last line above is bounded by a positive constant

- This gives a $\Theta(1)$ amortized cost for insertions

Binomial Queues

To learn more:

-  [J. Vuillemin](#)
A Data Structure for Manipulating Priority Queues.
Comm. ACM 21(4):309–315, 1978.
-  [T. Cormen, C. Leiserson, R. Rivest and C. Stein.](#)
Introduction to Algorithms, 2e.
MIT Press, 2001.

Part IV

Priority Queues

6

Introduction

7

Binary Heaps & Heapsort

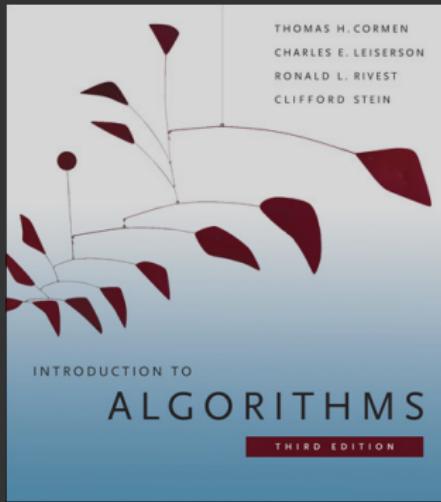
- Heapsort

8

Binomial Queues

9

Fibonacci Heaps



FIBONACCI HEAPS

- ▶ *preliminaries*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

Lecture slides by Kevin Wayne

<http://www.cs.princeton.edu/~wayne/kleinberg-tardos>

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Ahead. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

Fibonacci heaps

Theorem. [Fredman-Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving n INSERT operations takes $O(m + n \log n)$ time.

this statement is a bit weaker
than the actual theorem

Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms

MICHAEL L. FREDMAN

University of California, San Diego, La Jolla, California

AND

ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

Abstract. In this paper we develop a new data structure for implementing heaps (priority queues). Our structure, *Fibonacci heaps* (abbreviated *F-heaps*), extends the binomial queues proposed by Vuillemin and studied further by Brown. F-heaps support arbitrary deletion from an n -item heap in $O(\log n)$ amortized time and all other standard heap operations in $O(1)$ amortized time. Using F-heaps we are able to obtain improved running times for several network optimization algorithms. In particular, we obtain the following worst-case bounds, where n is the number of vertices and m the number of edges in the problem graph:

- (1) $O(n \log n + m)$ for the single-source shortest path problem with nonnegative edge lengths, improved from $O(nm \log_{n+2} n^2)$;
- (2) $O(n^2 \log n + nm)$ for the all-pairs shortest path problem, improved from $O(nm \log_{n+2} n^2)$;
- (3) $O(n^2 \log n + nm)$ for the assignment problem (weighted bipartite matching), improved from $O(nm \log_{n+2} n^2)$;
- (4) $O(m\beta(m, n))$ for the minimum spanning tree problem, improved from $O(m \log \log_{n+2} n)$, where $\beta(m, n) = \min \{l \mid \log^l n \leq m/n\}$. Note that $\beta(m, n) \leq \log^* n$ if $m \geq n$.

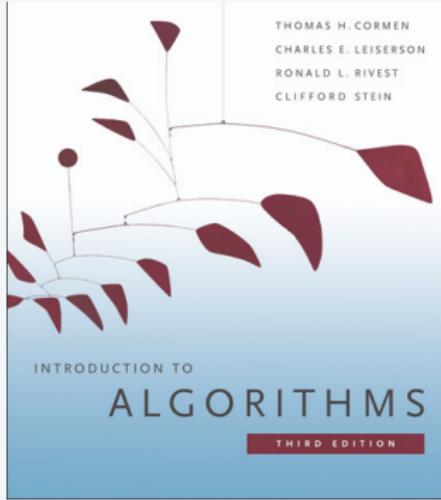
Of these results, the improved bound for minimum spanning trees is the most striking, although all the results give asymptotic improvements for graphs of appropriate densities.

Fibonacci heaps

Theorem. [Fredman-Tarjan 1986] Starting from an empty Fibonacci heap, any sequence of m INSERT, EXTRACT-MIN, and DECREASE-KEY operations involving n INSERT operations takes $O(m + n \log n)$ time.

History.

- Ingenious data structure and application of amortized analysis.
- Original motivation: improve Dijkstra's shortest path algorithm from $O(m \log n)$ to $O(m + n \log n)$.
- Also improved best-known bounds for all-pairs shortest paths, assignment problem, minimum spanning trees.



SECTION 19.1

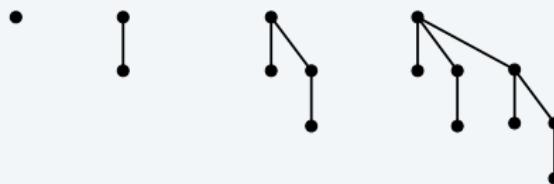
FIBONACCI HEAPS

- ▶ *structure*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

Fibonacci heaps

Basic idea.

- Similar to binomial heaps, but less rigid structure.
- Binomial heap: **eagerly** consolidate trees after each INSERT; implement DECREASE-KEY by repeatedly exchanging node with its parent.



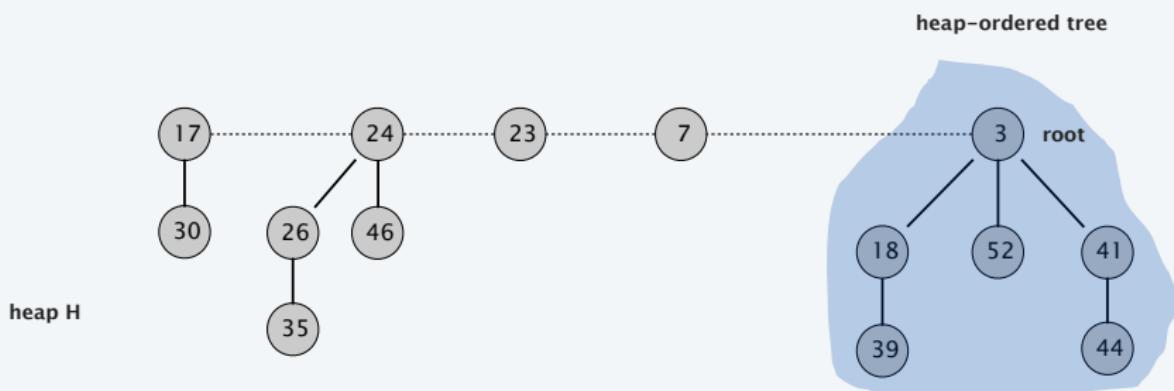
- Fibonacci heap: **lazily** defer consolidation until next EXTRACT-MIN; implement DECREASE-KEY by cutting off node and splicing into root list.

Remark. Height of Fibonacci heap is $\Theta(n)$ in worst case, but it doesn't use sink or swim operations.

Fibonacci heap: structure

- Set of **heap-ordered** trees.

↑
each child no smaller
than its parent

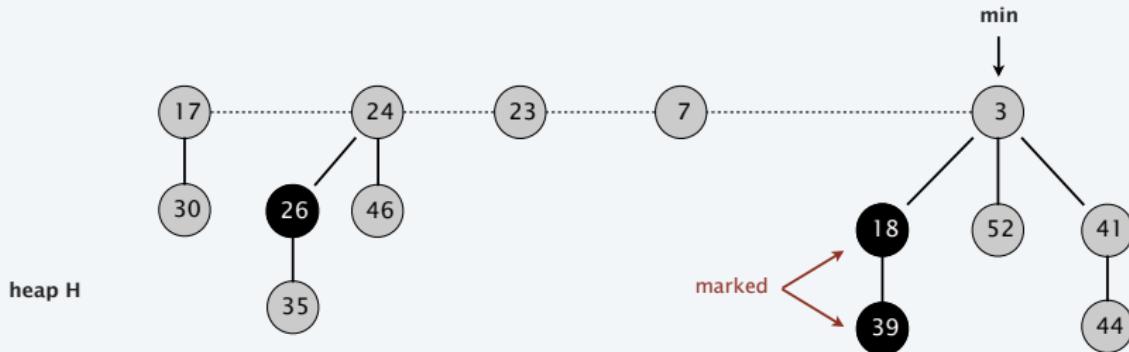


Fibonacci heap: structure

- Set of heap-ordered trees.
- Set of marked nodes.



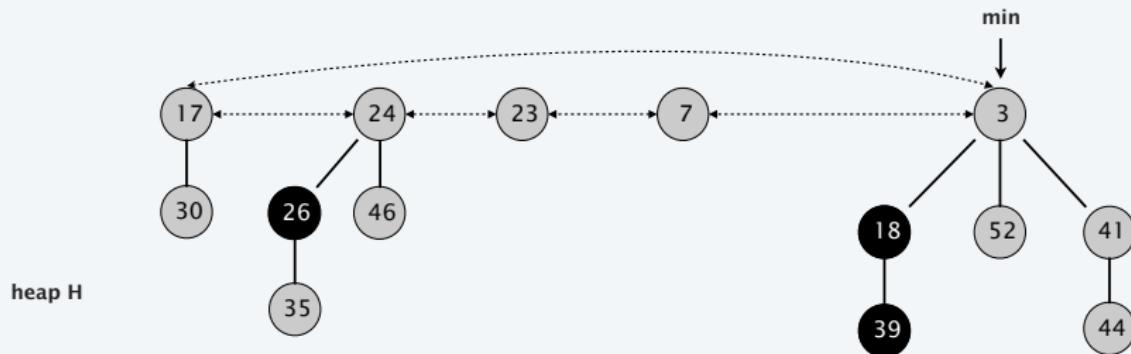
used to keep trees bushy
(stay tuned)



Fibonacci heap: structure

Heap representation.

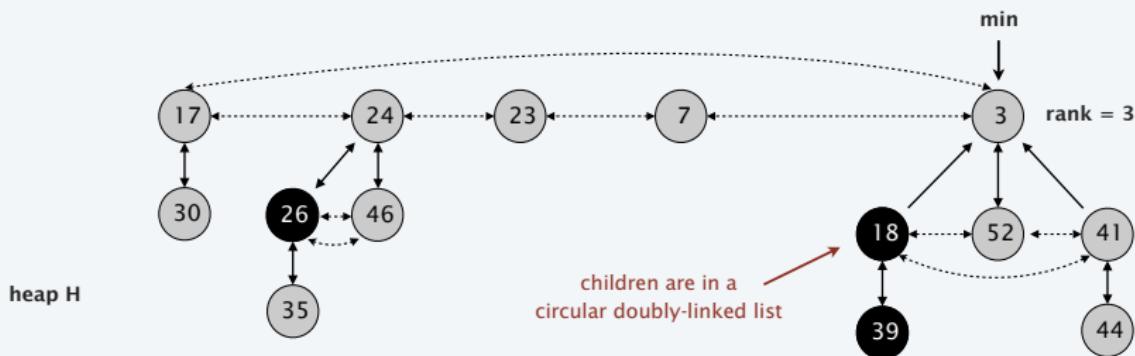
- Store a pointer to the minimum node.
- Maintain tree roots in a circular, doubly-linked list.



Fibonacci heap: representation

Node representation. Each node stores:

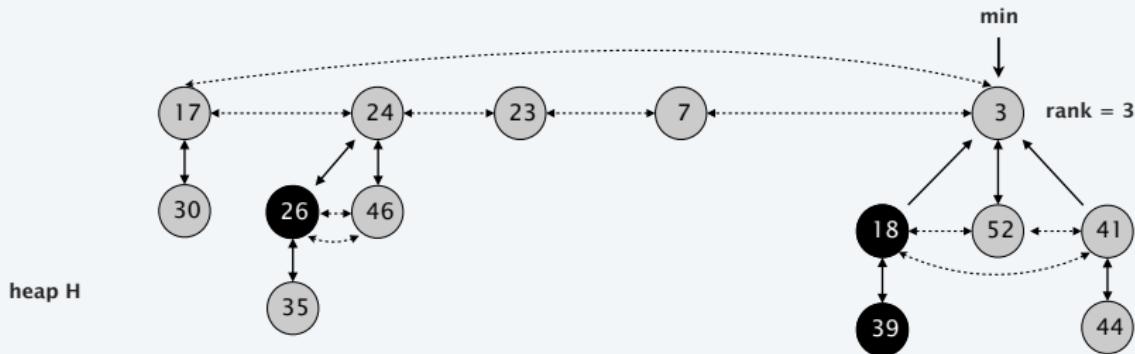
- A pointer to its parent.
- A pointer to any of its children.
- A pointer to its left and right siblings.
- Its rank = number of children.
- Whether it is marked.



Fibonacci heap: representation

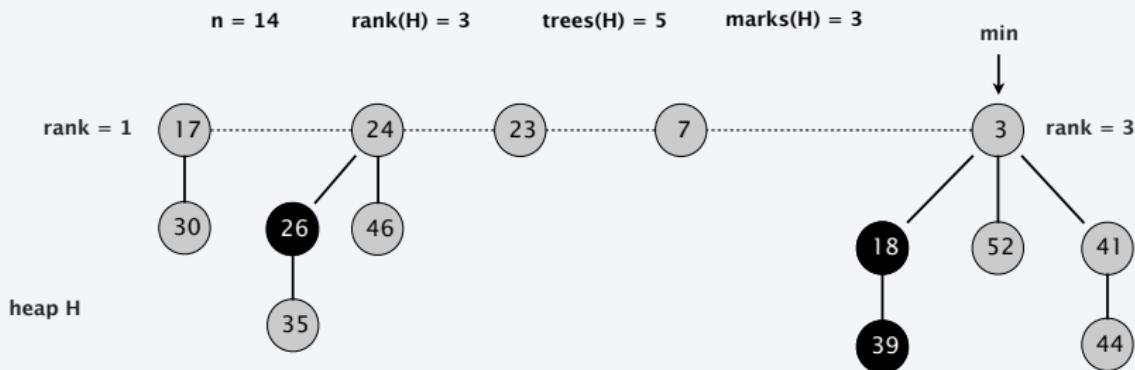
Operations we can do in constant time:

- Find the minimum element.
- Merge two root lists together.
- Determine rank of a root node.
- Add or remove a node from the root list.
- Remove a subtree and merge into root list.
- Link the root of one tree to root of another tree.



Fibonacci heap: notation

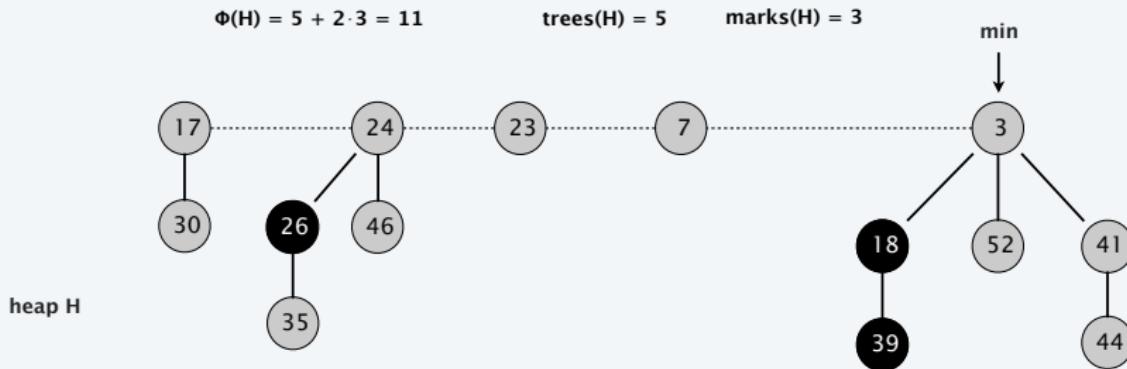
notation	meaning
n	number of nodes
$rank(x)$	number of children of node x
$rank(H)$	max rank of any node in heap H
$trees(H)$	number of trees in heap H
$marks(H)$	number of marked nodes in heap H

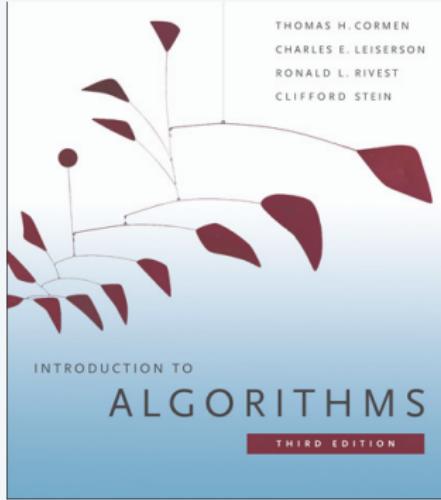


Fibonacci heap: potential function

Potential function.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$





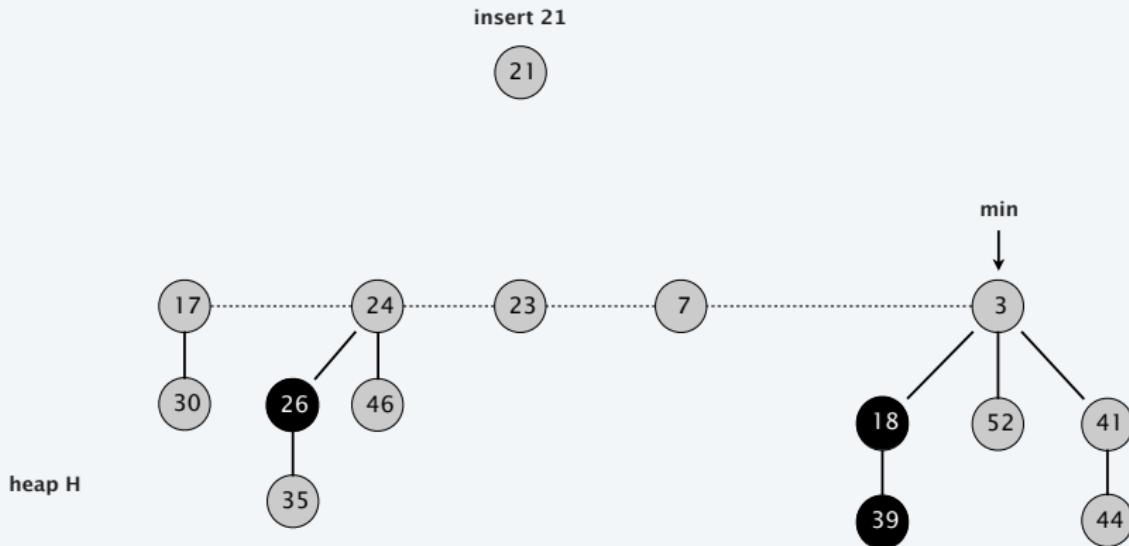
SECTION 19.2

FIBONACCI HEAPS

- ▶ *preliminaries*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

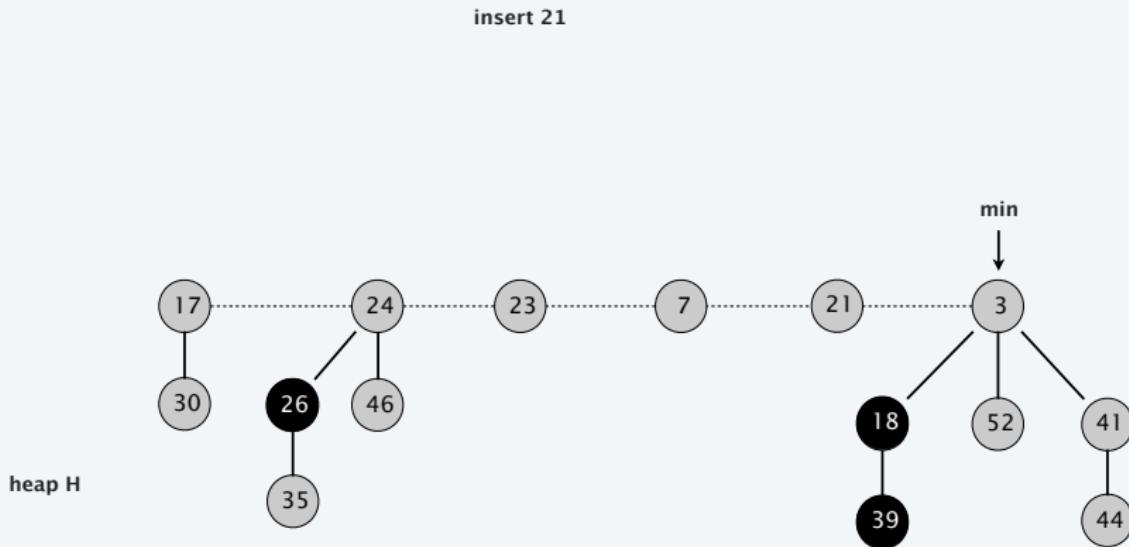
Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).



Fibonacci heap: insert

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary).



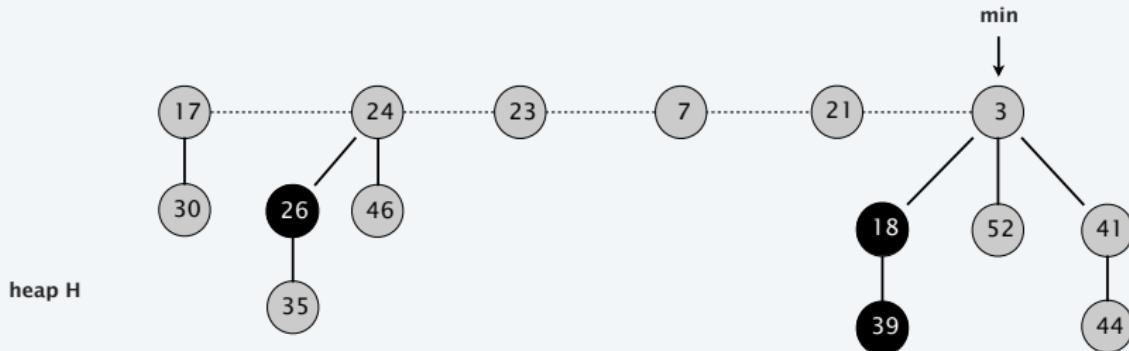
Fibonacci heap: insert analysis

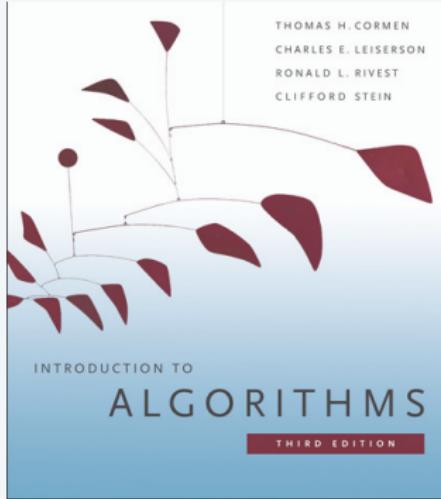
Actual cost. $c_i = O(1)$.

Change in potential. $\Delta\Phi = \Phi(H_i) - \Phi(H_{i-1}) = +1$. ← one more tree; no change in marks

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$





SECTION 19.2

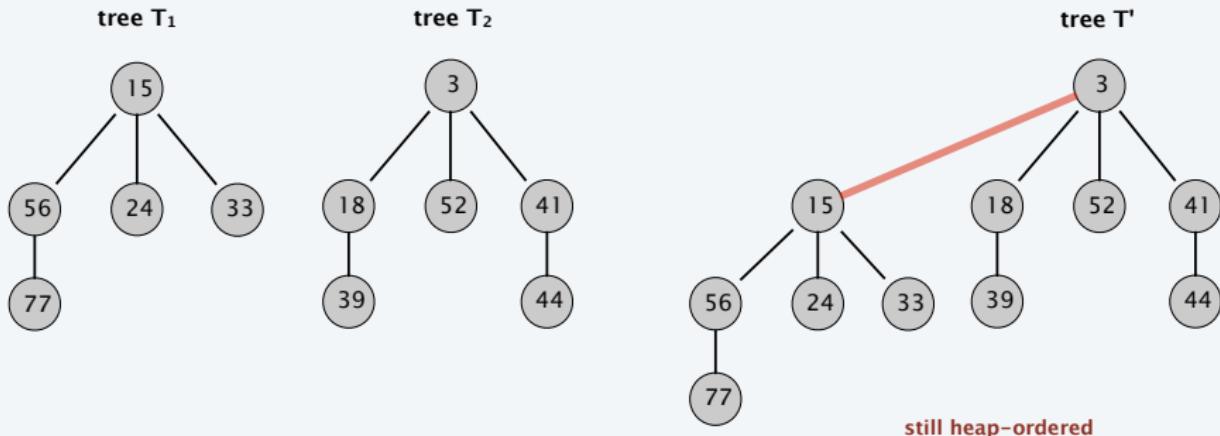
FIBONACCI HEAPS

- ▶ *preliminaries*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

Linking operation

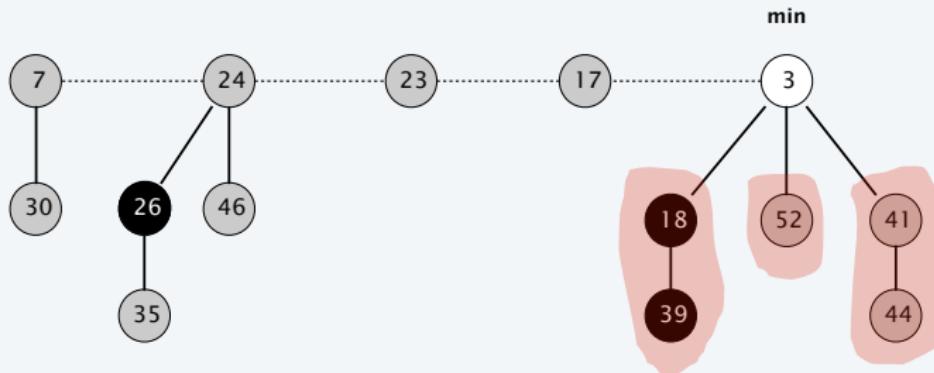
Useful primitive. Combine two trees T_1 and T_2 of rank k .

- Make larger root be a child of smaller root.
- Resulting tree T' has rank $k + 1$.



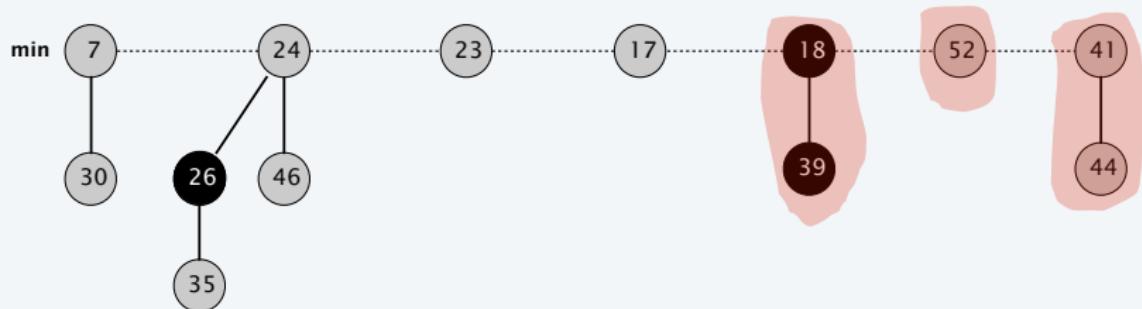
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



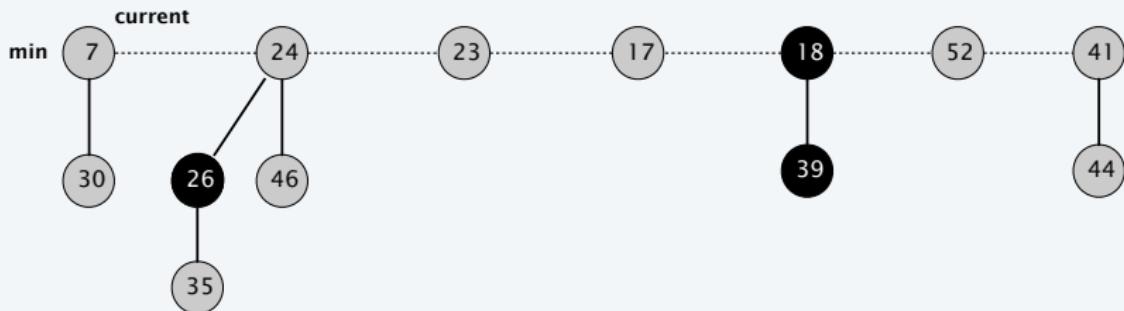
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



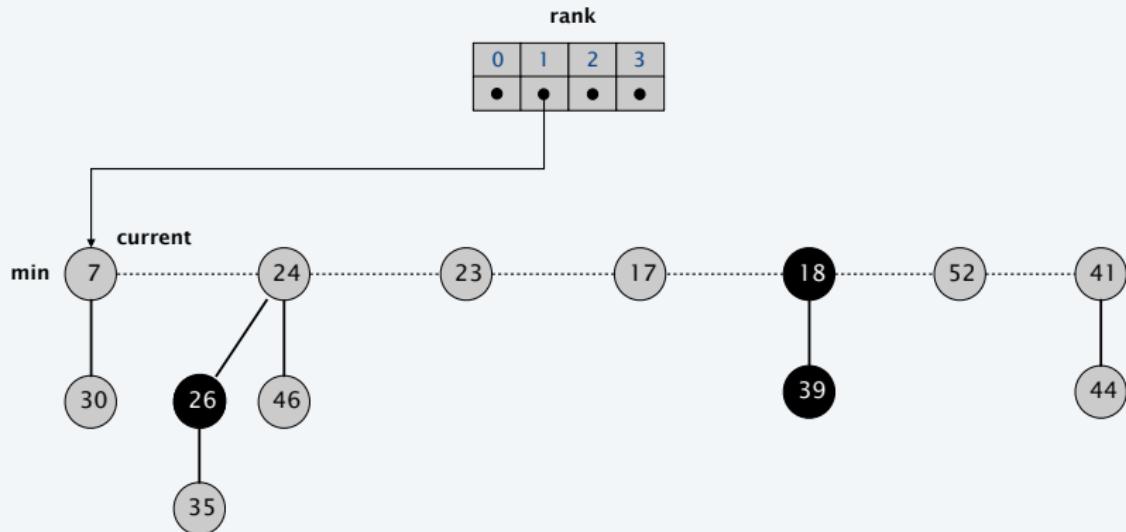
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



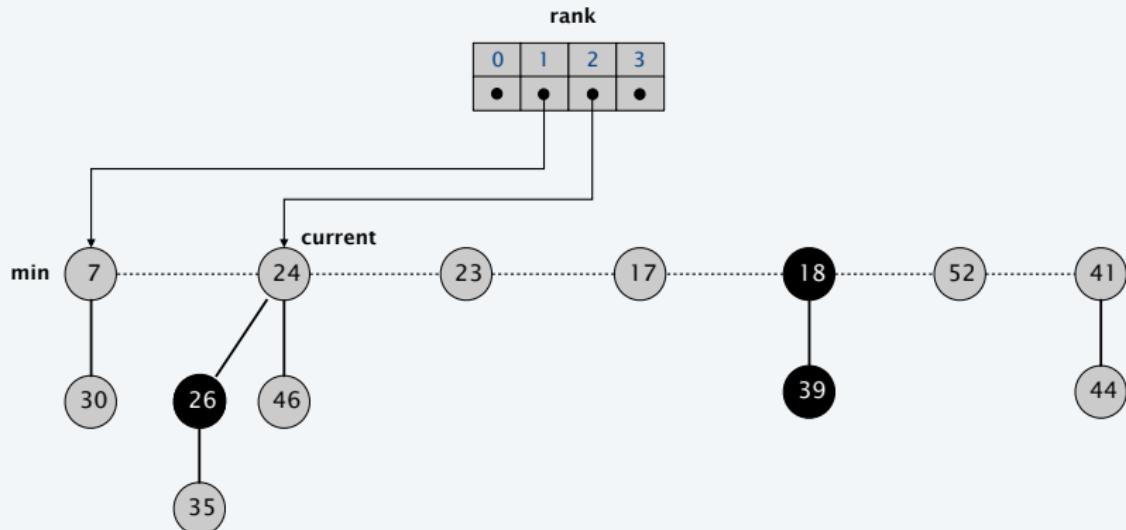
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



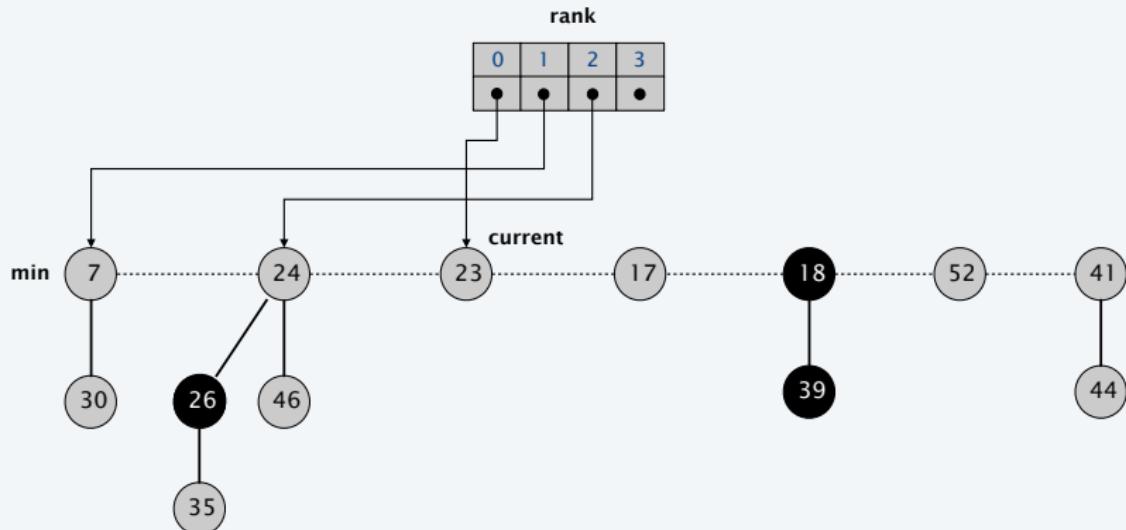
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



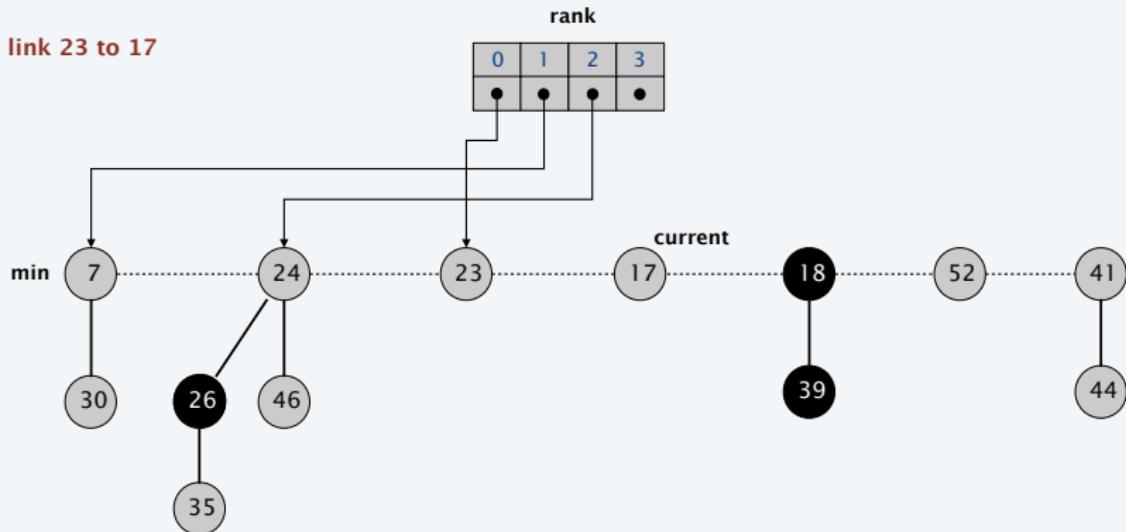
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



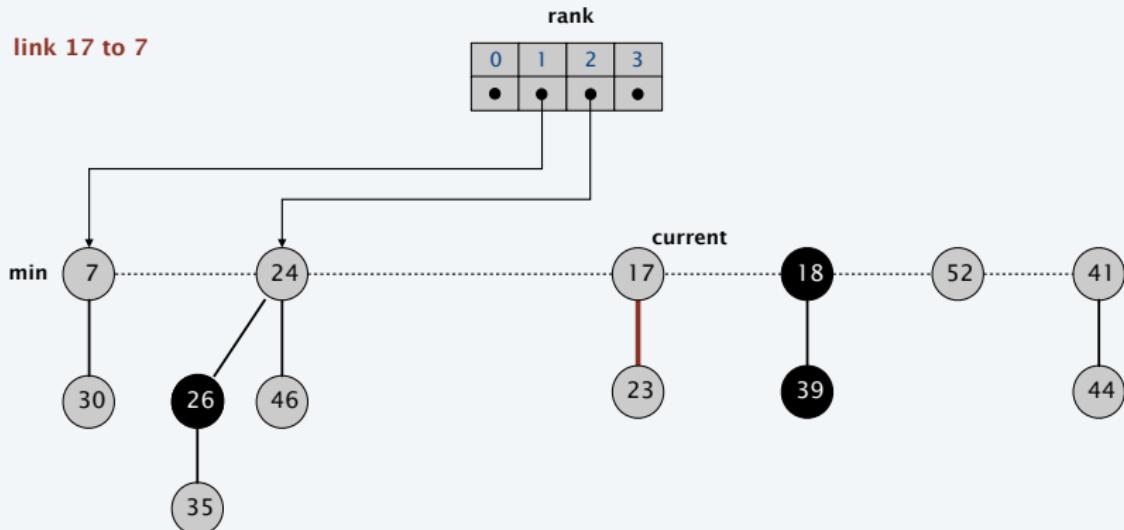
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
 - Consolidate trees so that no two roots have same rank.



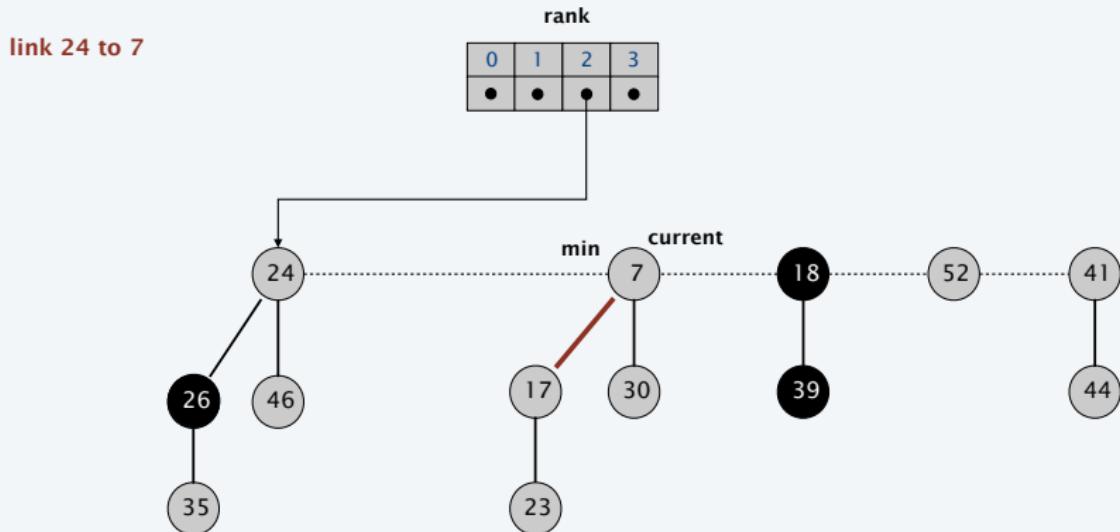
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



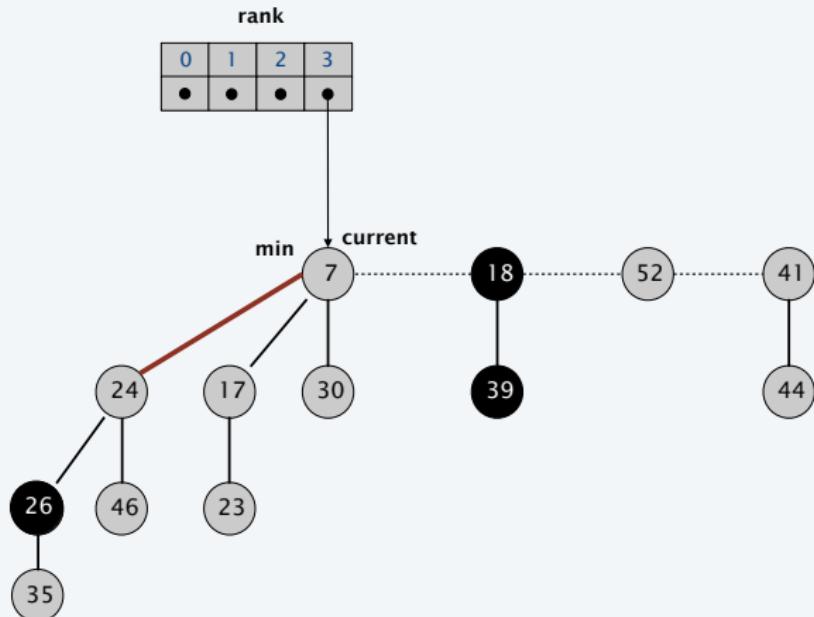
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



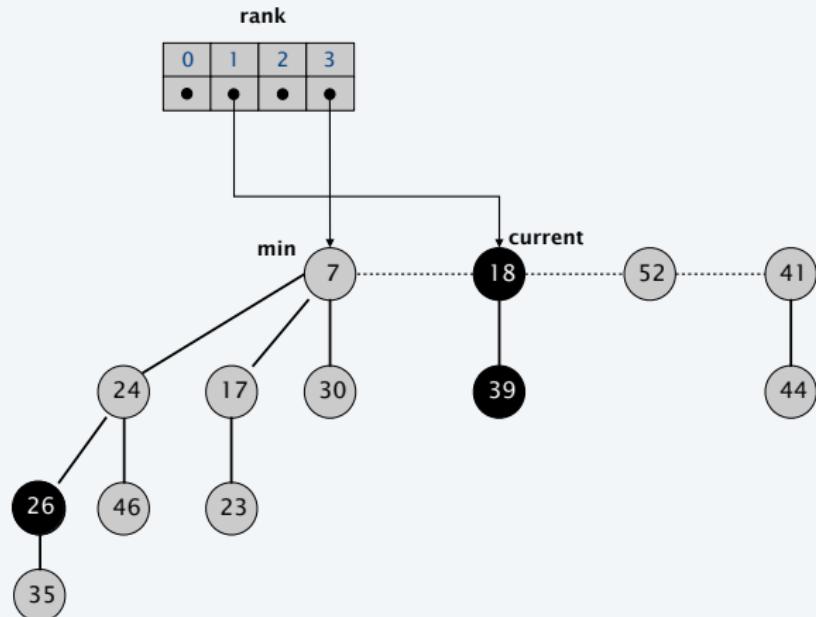
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



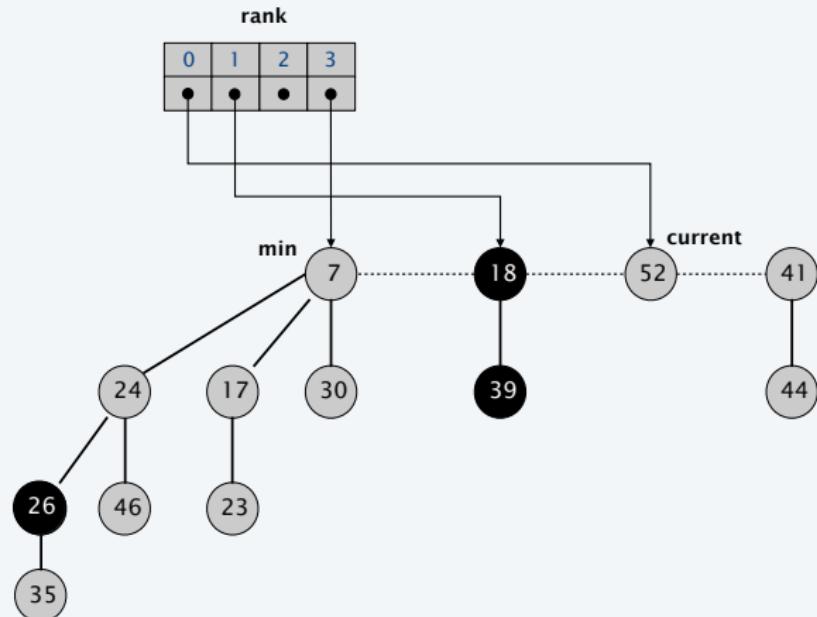
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



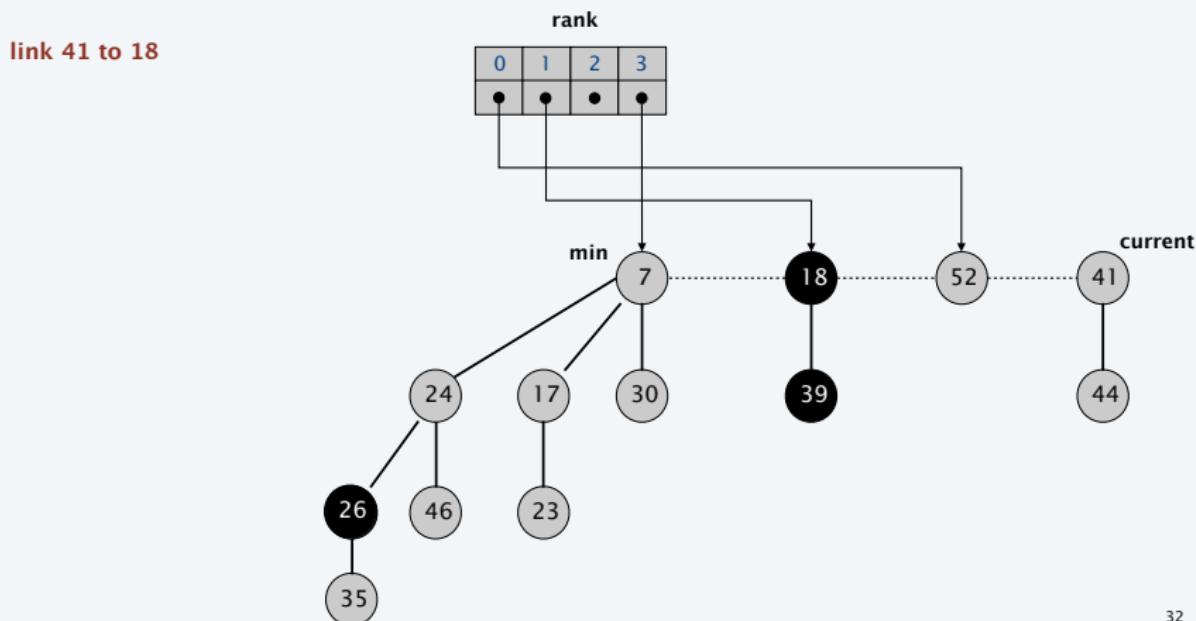
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



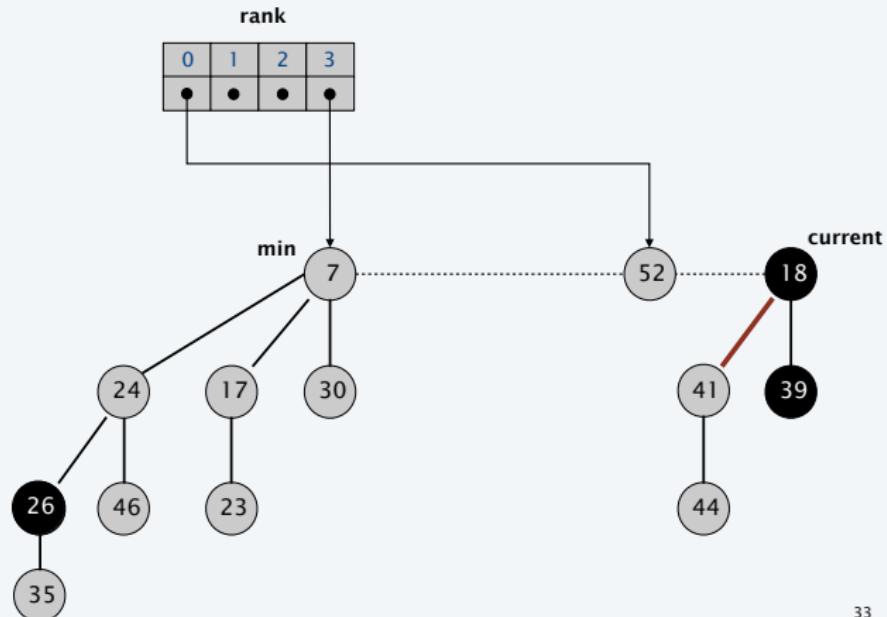
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



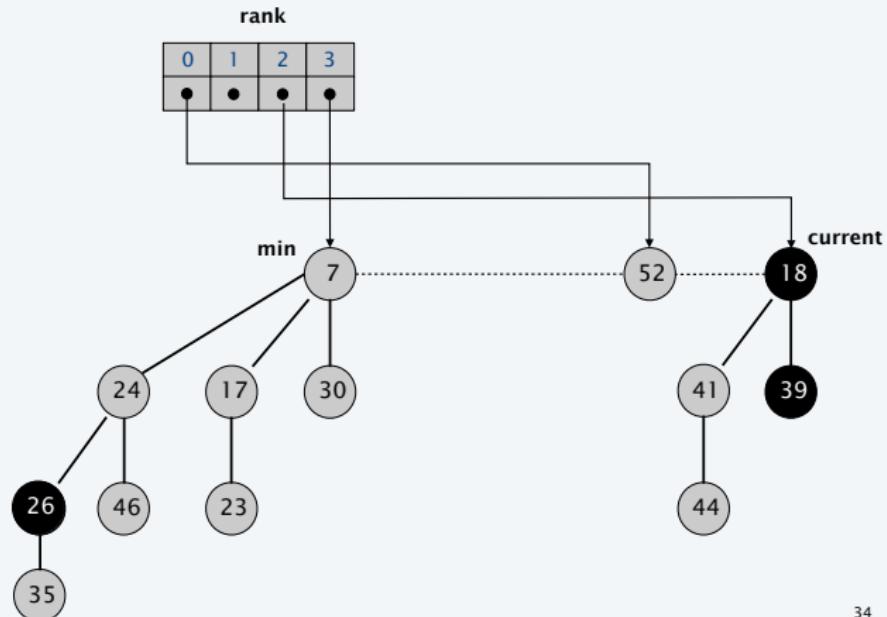
Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

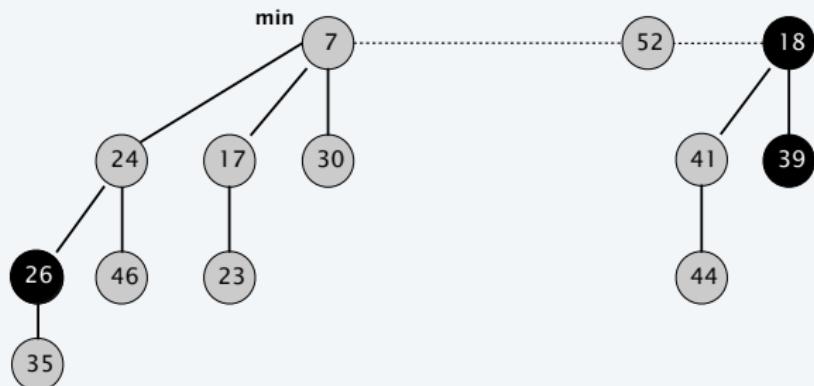
- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.



Fibonacci heap: extract the minimum

- Delete min; meld its children into root list; update min.
- Consolidate trees so that no two roots have same rank.

stop (no two trees have same rank)



Fibonacci heap: extract the minimum analysis

Actual cost. $c_i = O(\text{rank}(H)) + O(\text{trees}(H))$.

- $O(\text{rank}(H))$ to meld min's children into root list. ← $\leq \text{rank}(H)/\text{children}$
- $O(\text{rank}(H)) + O(\text{trees}(H))$ to update min. ← $\leq \text{rank}(H) + \text{trees}(H) - 1$ root nodes
- $O(\text{rank}(H)) + O(\text{trees}(H))$ to consolidate trees. ← number of roots decreases by 1 after each linking operation

bound on $\text{trees}(H') - \text{trees}(H)$

Change in potential. $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$.

- No new nodes become marked.
- $\text{trees}(H') \leq \text{rank}(H') + 1$. ← no two trees have same rank after consolidation

Amortized cost. $O(\log n)$.

- $\hat{c}_i = c_i + \Delta\Phi = O(\text{rank}(H)) + O(\text{rank}(H'))$.
- The rank of a Fibonacci heap with n elements is $O(\log n)$.

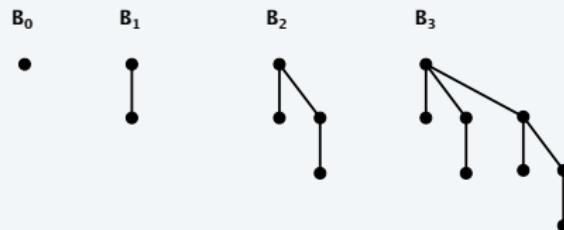
↑
Fibonacci lemma
(stay tuned)

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Fibonacci heap vs. binomial heaps

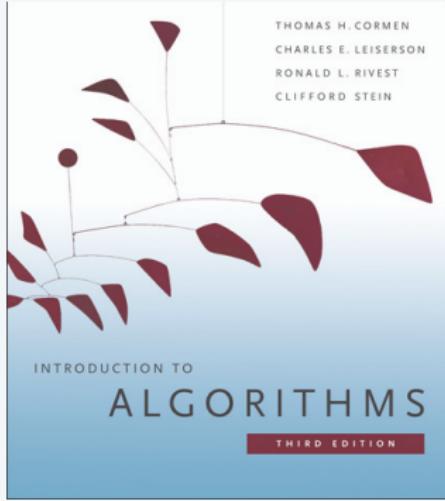
Observation. If only INSERT and EXTRACT-MIN operations, then all trees are binomial trees.

we link only trees of equal rank



Binomial heap property. This implies $\text{rank}(H) \leq \log_2 n$.

Fibonacci heap property. Our DECREASE-KEY implementation will not preserve this property, but we will implement it in such a way that $\text{rank}(H) \leq \log_{\phi} n$.



SECTION 19.3

FIBONACCI HEAPS

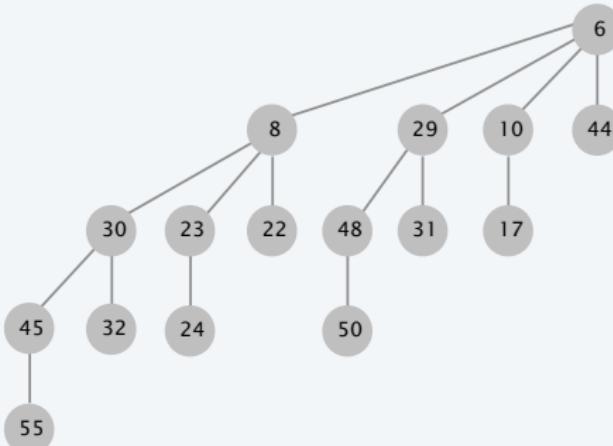
- ▶ *preliminaries*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

Fibonacci heap: decrease key

Intuition for decreasing the key of node x .

- If heap-order is not violated, decrease the key of x .
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of x from 30 to 7

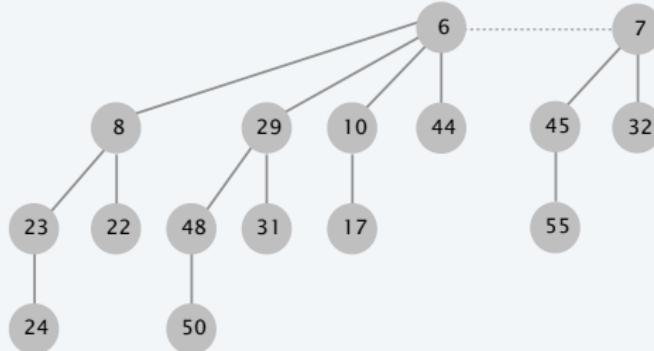


Fibonacci heap: decrease key

Intuition for decreasing the key of node x .

- If heap-order is not violated, decrease the key of x .
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of x from 23 to 5



Fibonacci heap: decrease key

Intuition for decreasing the key of node x .

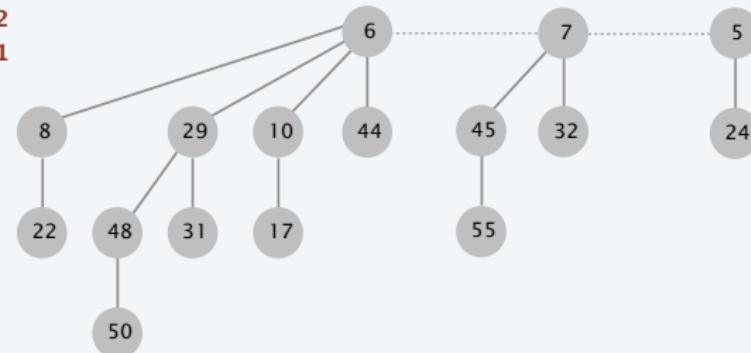
- If heap-order is not violated, decrease the key of x .
- Otherwise, cut tree rooted at x and meld into root list.

decrease-key of 22 to 4

decrease-key of 48 to 3

decrease-key of 31 to 2

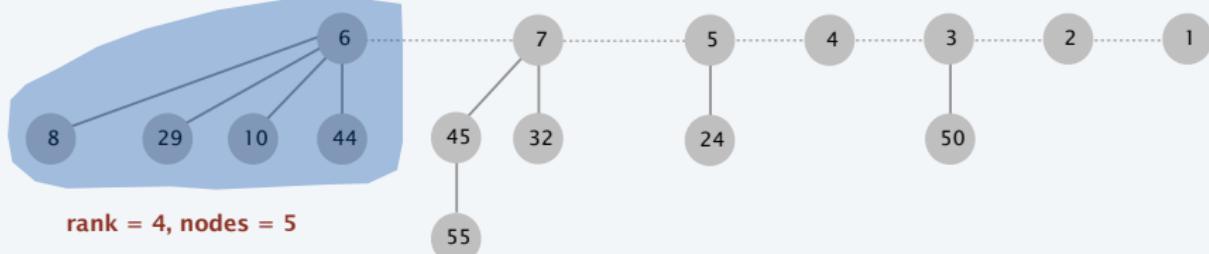
decrease-key of 17 to 1



Fibonacci heap: decrease key

Intuition for decreasing the key of node x .

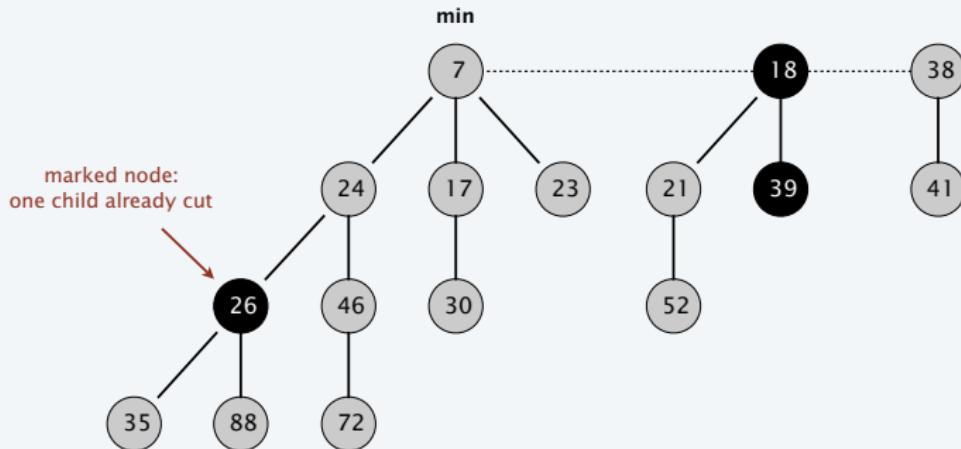
- If heap-order is not violated, decrease the key of x .
- Otherwise, cut tree rooted at x and meld into root list.
- Problem: number of nodes not exponential in rank.



Fibonacci heap: decrease key

Intuition for decreasing the key of node x .

- If heap-order is not violated, decrease the key of x .
- Otherwise, cut tree rooted at x and meld into root list.
- Solution: as soon as a node has its second child cut, cut it off also and meld into root list (and unmark it).

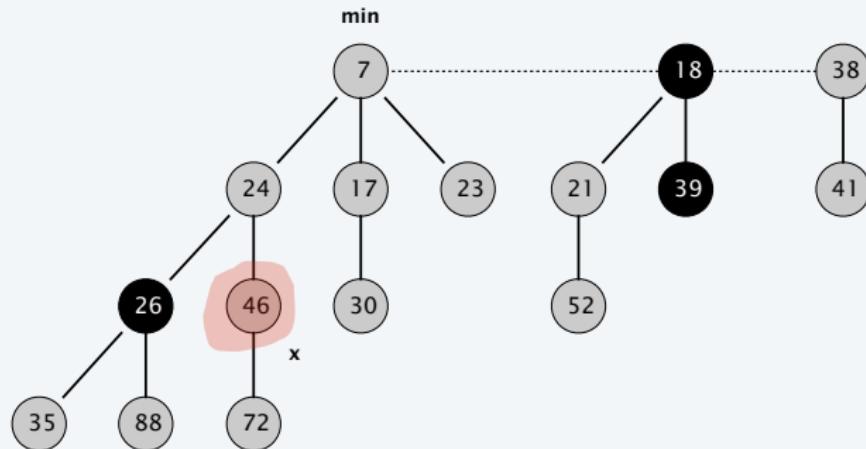


Fibonacci heap: decrease key

Case 1. [heap order not violated]

- Decrease key of x .
- Change heap min pointer (if necessary).

decrease-key of x from 46 to 29

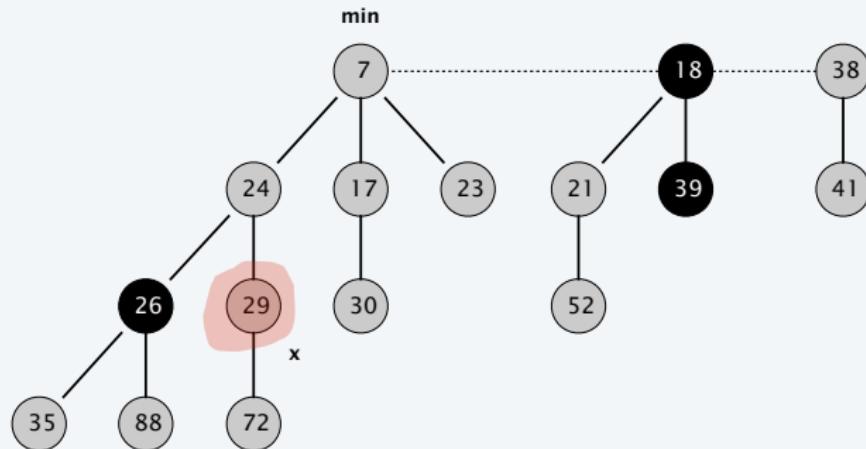


Fibonacci heap: decrease key

Case 1. [heap order not violated]

- Decrease key of x .
- Change heap min pointer (if necessary).

decrease-key of x from 46 to 29

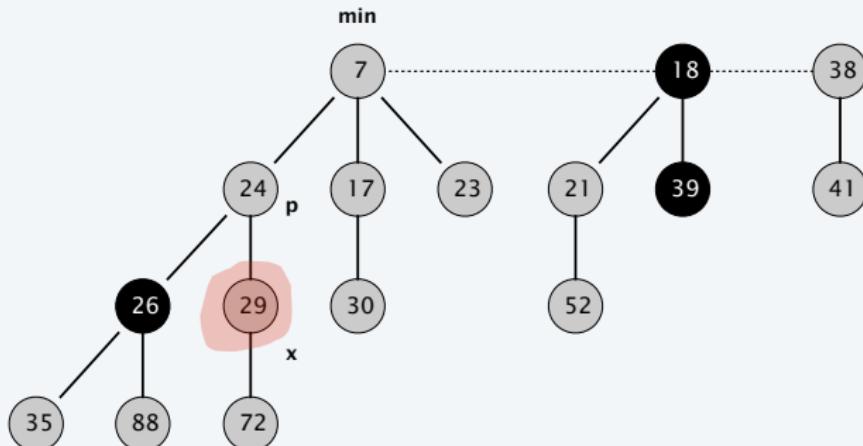


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

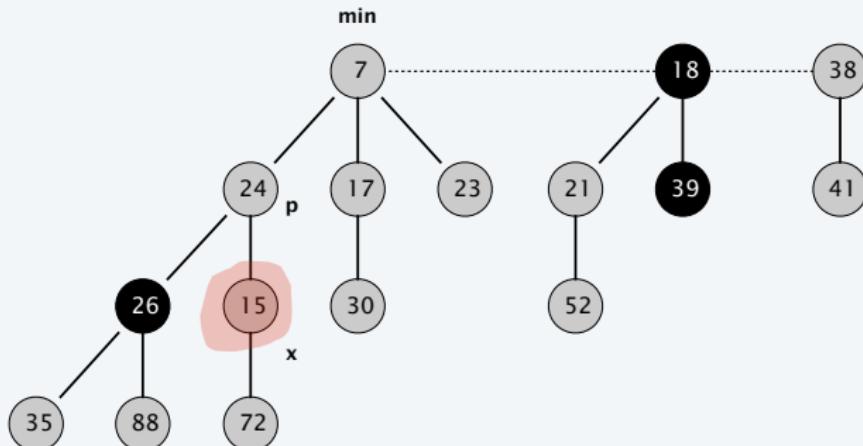


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

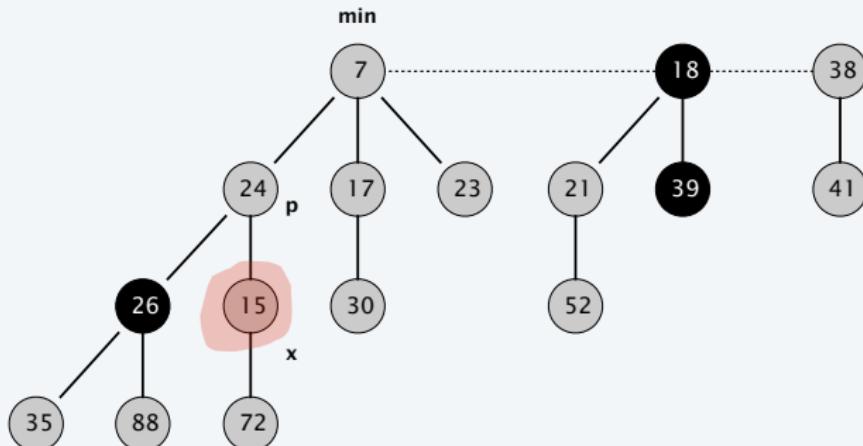


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

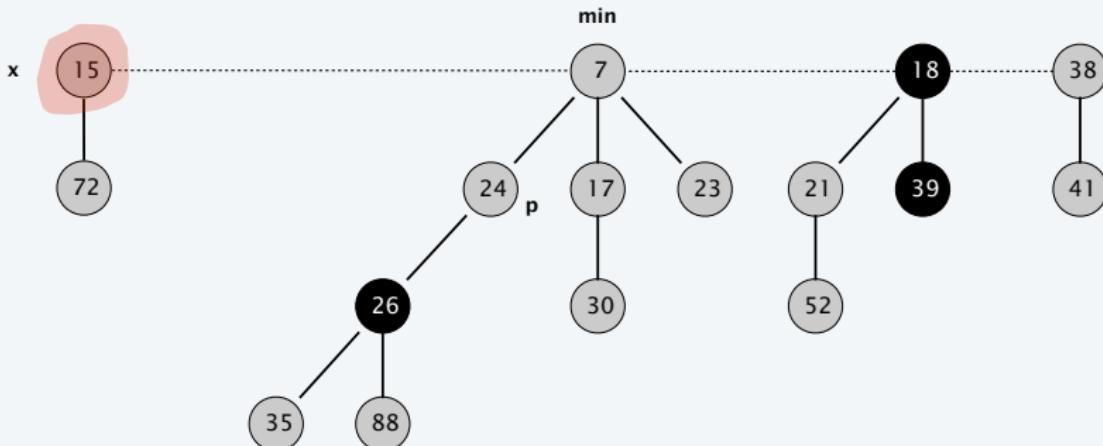


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

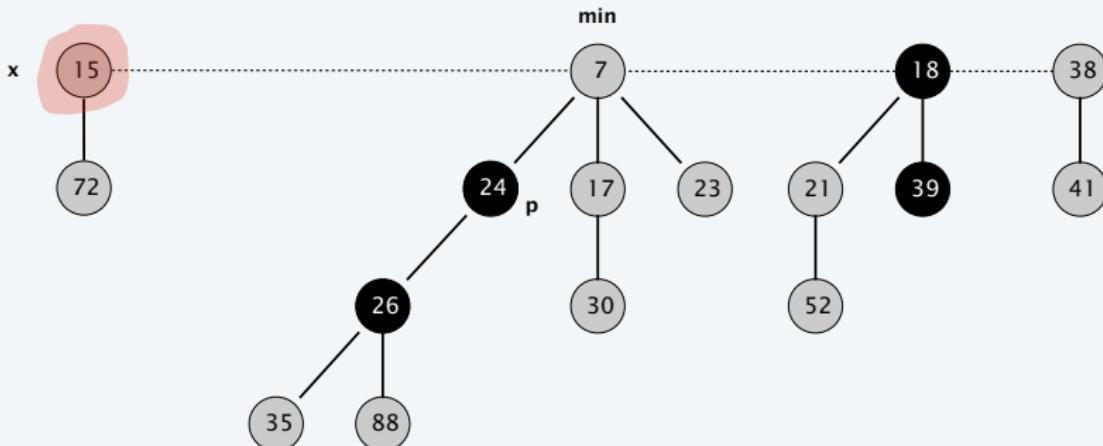


Fibonacci heap: decrease key

Case 2a. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 29 to 15

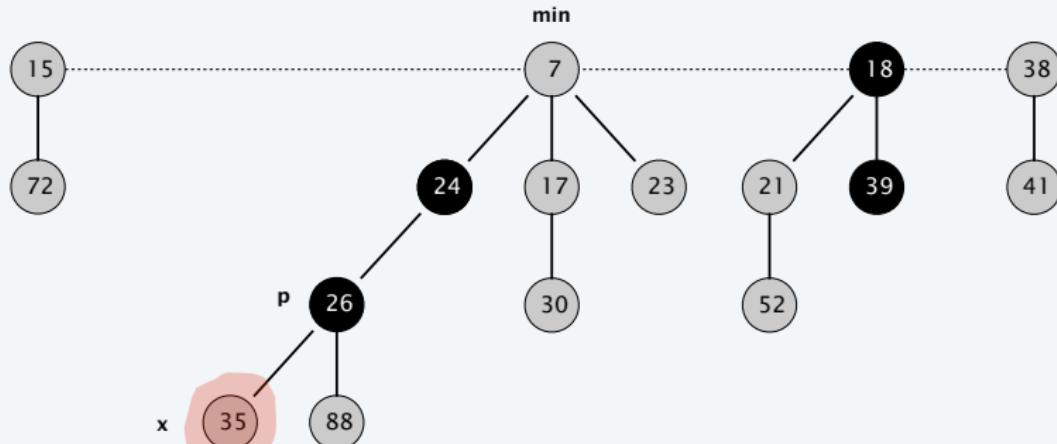


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

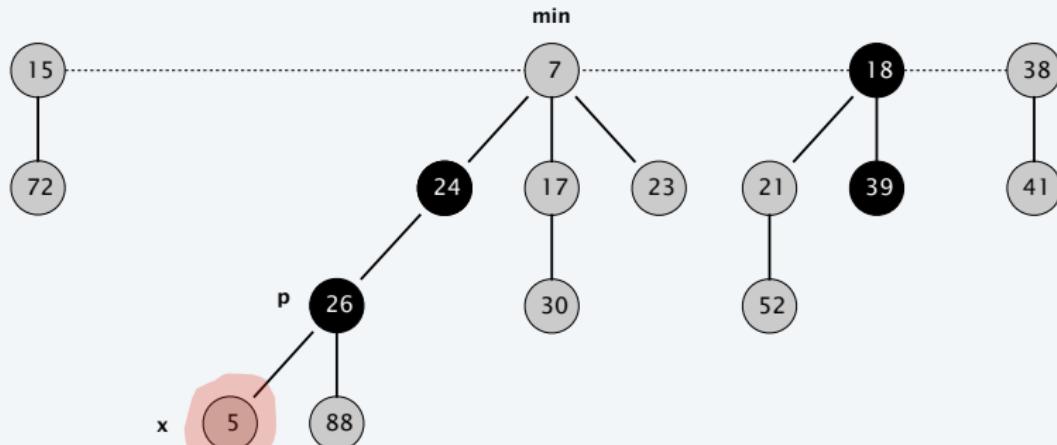


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

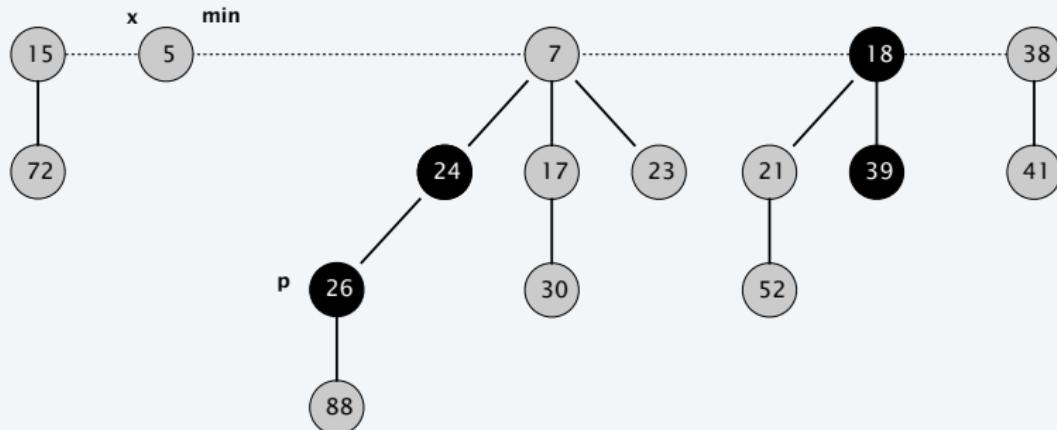


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

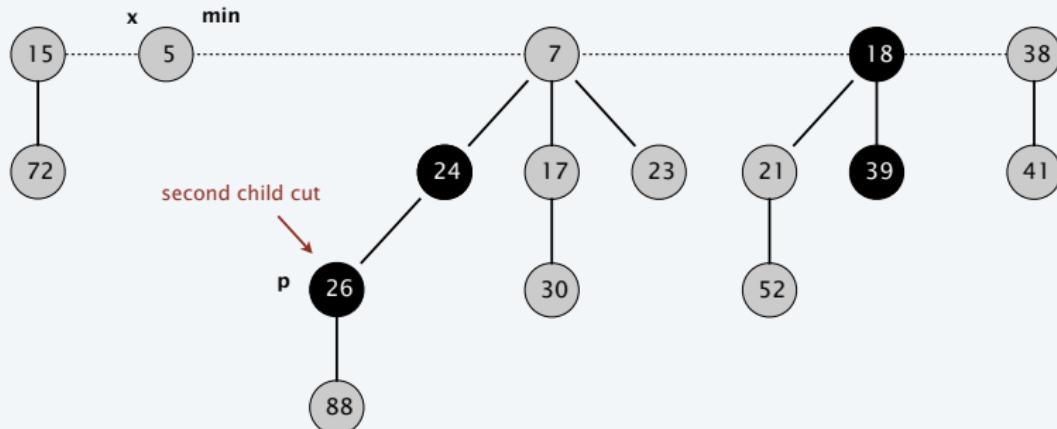


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

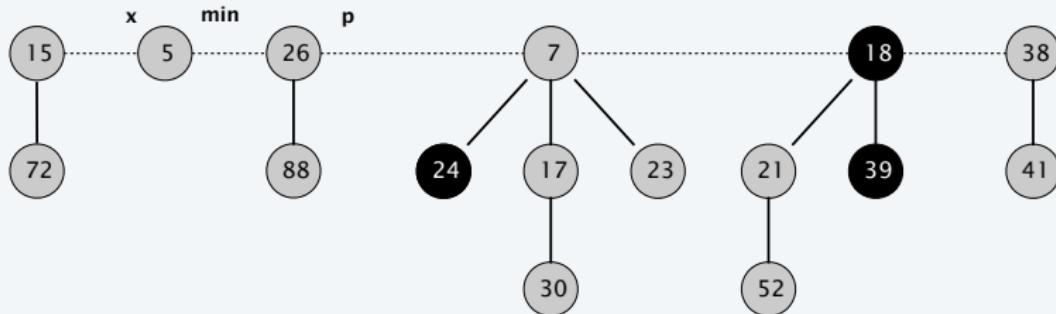


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

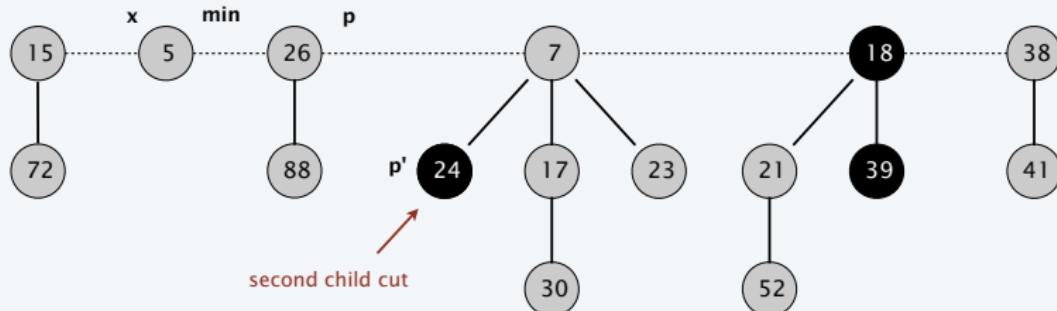


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5

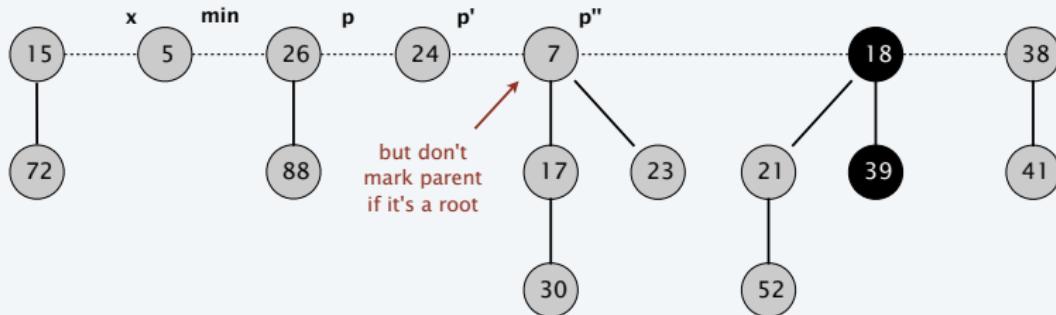


Fibonacci heap: decrease key

Case 2b. [heap order violated]

- Decrease key of x .
- Cut tree rooted at x , meld into root list, and unmark.
- If parent p of x is unmarked (hasn't yet lost a child), mark it;
Otherwise, cut p , meld into root list, and unmark
(and do so recursively for all ancestors that lose a second child).

decrease-key of x from 35 to 5



Fibonacci heap: decrease key analysis

Actual cost. $c_i = O(c)$, where c is the number of cuts.

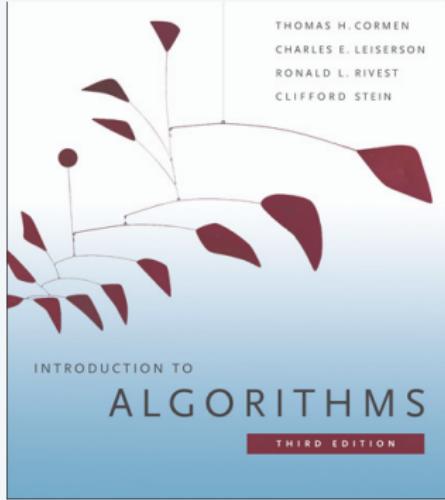
- $O(1)$ time for changing the key.
- $O(1)$ time for each of c cuts, plus melding into root list.

Change in potential. $\Delta\Phi = O(1) - c$.

- $trees(H') = trees(H) + c$.
- $marks(H') \leq marks(H) - c + 2$. ← each cut (except first) unmarks a node
last cut may or may not mark a node
- $\Delta\Phi \leq c + 2 \cdot (-c + 2) = 4 - c$.

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = trees(H) + 2 \cdot marks(H)$$



SECTION 19.4

FIBONACCI HEAPS

- ▶ *preliminaries*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

Analysis summary

Insert. $O(1)$.

Delete-min. $O(\text{rank}(H))$ amortized.

Decrease-key. $O(1)$ amortized.

Fibonacci lemma. Let H be a Fibonacci heap with n elements.

Then, $\text{rank}(H) = O(\log n)$.

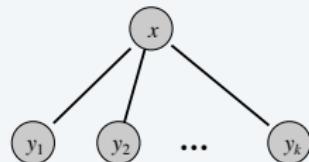
number of nodes is
exponential in rank



Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k , and let y_1, \dots, y_k denote its current children in the order in which they were linked to x . Then:

$$\text{rank}(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



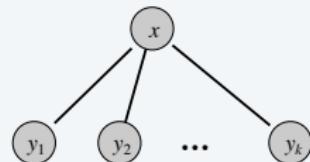
Pf.

- When y_i was linked into x , x had at least $i - 1$ children y_1, \dots, y_{i-1} .
- Since only trees of equal rank are linked, at that time
 $\text{rank}(y_i) = \text{rank}(x) \geq i - 1$. this is why we need the marking
- Since then, y_i has lost at most one child (or y_i would have been cut).
- Thus, right now $\text{rank}(y_i) \geq i - 2$. ■

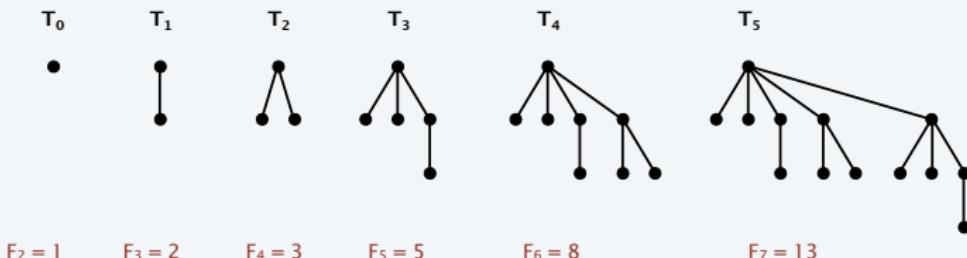
Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k , and let y_1, \dots, y_k denote its current children in the order in which they were linked to x . Then:

$$\text{rank}(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



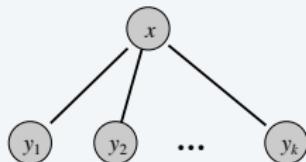
Def. Let T_k be smallest possible tree of rank k satisfying property.



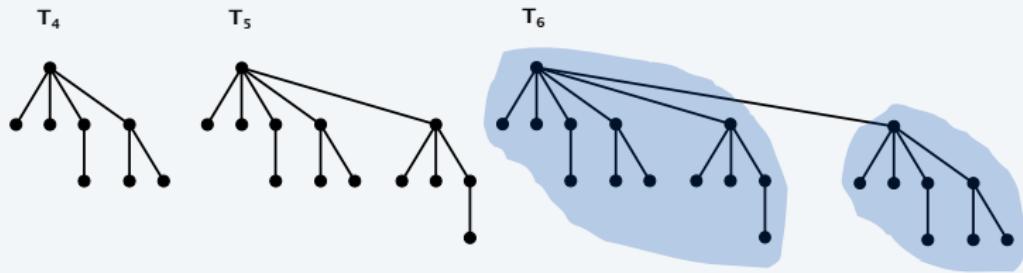
Bounding the rank

Lemma 1. Fix a point in time. Let x be a node of rank k , and let y_1, \dots, y_k denote its current children in the order in which they were linked to x . Then:

$$rank(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i \geq 2 \end{cases}$$



Def. Let T_k be smallest possible tree of rank k satisfying property.



$$F_6 = 8$$

$$F_7 = 13$$

$$F_8 = F_6 + F_7 = 8 + 13 = 21$$

Bounding the rank

Lemma 2. Let s_k be minimum number of elements in any Fibonacci heap of rank k . Then $s_k \geq F_{k+2}$, where F_k is the k^{th} Fibonacci number.

Pf. [by strong induction on k]

- Base cases: $s_0 = 1$ and $s_1 = 2$.
- Inductive hypothesis: assume $s_i \geq F_{i+2}$ for $i = 0, \dots, k-1$.
- As in Lemma 1, let y_1, \dots, y_k denote its current children in the order in which they were linked to x .

$$\begin{aligned} s_k &\geq 1 + 1 + (s_0 + s_1 + \dots + s_{k-2}) && (\text{Lemma 1}) \\ &\geq (1 + F_1) + F_2 + F_3 + \dots + F_k && (\text{inductive hypothesis}) \\ &= F_{k+2}. \blacksquare && (\text{Fibonacci fact 1}) \end{aligned}$$

Bounding the rank

Fibonacci lemma. Let H be a Fibonacci heap with n elements.

Then, $\text{rank}(H) \leq \log_{\phi} n$, where ϕ is the golden ratio $= (1 + \sqrt{5}) / 2 \approx 1.618$.

Pf.

- Let H is a Fibonacci heap with n elements and rank k .
- Then $n \geq F_{k+2} \geq \phi^k$.

$$\begin{array}{ccc} \uparrow & & \uparrow \\ \text{Lemma 2} & & \text{Fibonacci} \\ & & \text{Fact 2} \end{array}$$

- Taking logs, we obtain $\text{rank}(H) = k \leq \log_{\phi} n$. ■

Fibonacci fact 1

Def. The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Fibonacci fact 1. For all integers $k \geq 0$, $F_{k+2} = 1 + F_0 + F_1 + \dots + F_k$.

Pf. [by induction on k]

- Base case: $F_2 = 1 + F_0 = \text{apple}^1$
- Inductive hypothesis: assume $F_{k+1} = 1 + F_0 + F_1 + \dots + F_{k-1}$.

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} && \text{(definition)} \\ &= F_k + (1 + F_0 + F_1 + \dots + F_{k-1}) && \text{(inductive hypothesis)} \\ &= 1 + F_0 + F_1 + \dots + F_{k-1} + F_k. \blacksquare && \text{(algebra)} \end{aligned}$$

Fibonacci fact 2

Def. The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Fibonacci fact 2. $F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$.

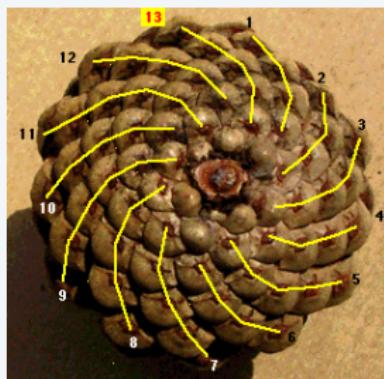
Pf. [by induction on k]

- **Base cases:** $F_2 = 1 \geq 1$, $F_3 = 2 \geq \phi$.
- **Inductive hypotheses:** assume $F_k \geq \phi^k$ and $F_{k+1} \geq \phi^{k+1}$

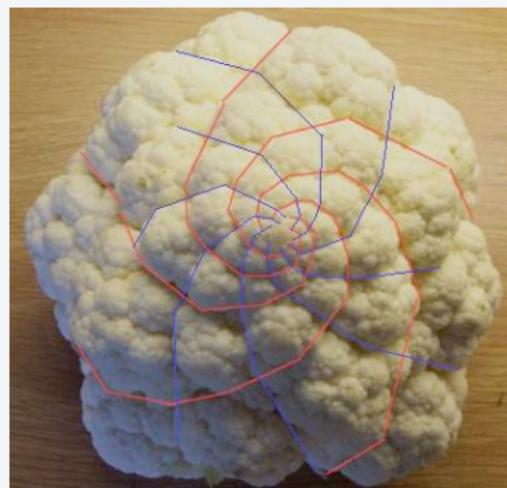
$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} && \text{(definition)} \\ &\geq \phi^{k-1} + \phi^{k-2} && \text{(inductive hypothesis)} \\ &= \phi^{k-2}(1 + \phi) && \text{(algebra)} \\ &= \phi^{k-2} \phi^2 && (\phi^2 = \phi + 1) \\ &= \phi^k. \blacksquare && \text{(algebra)} \end{aligned}$$

Fibonacci numbers and nature

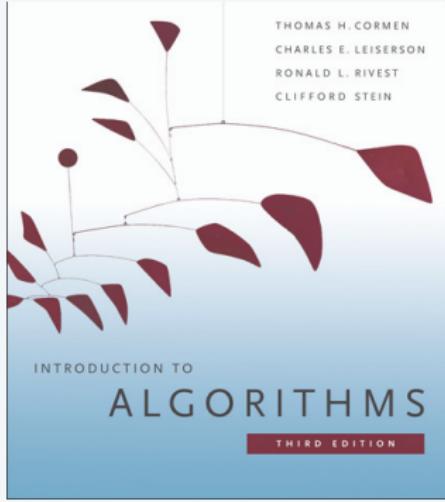
Fibonacci numbers arise both in nature and algorithms.



pinecone



cauliflower



FIBONACCI HEAPS

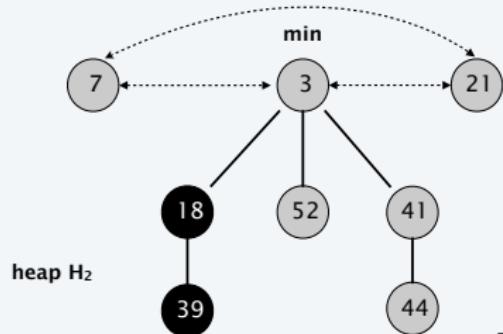
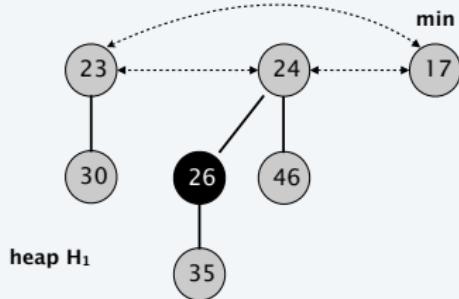
- ▶ *preliminaries*
- ▶ *insert*
- ▶ *extract the minimum*
- ▶ *decrease key*
- ▶ *bounding the rank*
- ▶ *meld and delete*

SECTION 19.2, 19.3

Fibonacci heap: meld

Meld. Combine two Fibonacci heaps (destroying old heaps).

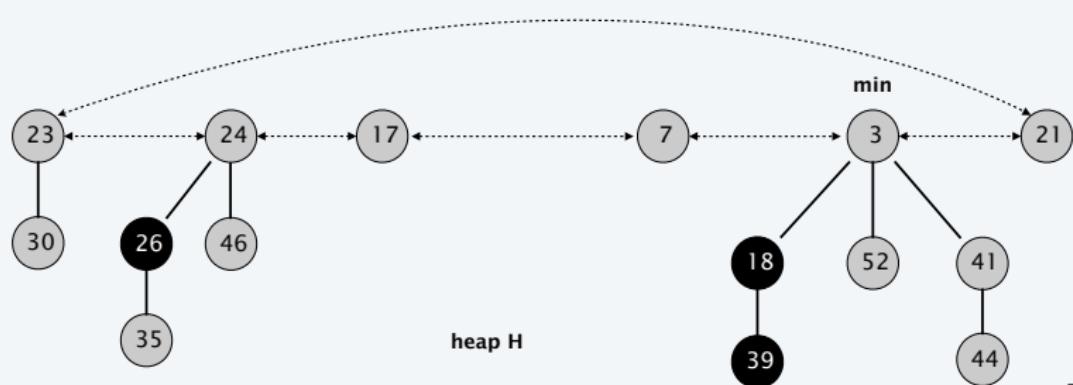
Recall. Root lists are circular, doubly-linked lists.



Fibonacci heap: meld

Meld. Combine two Fibonacci heaps (destroying old heaps).

Recall. Root lists are circular, doubly-linked lists.



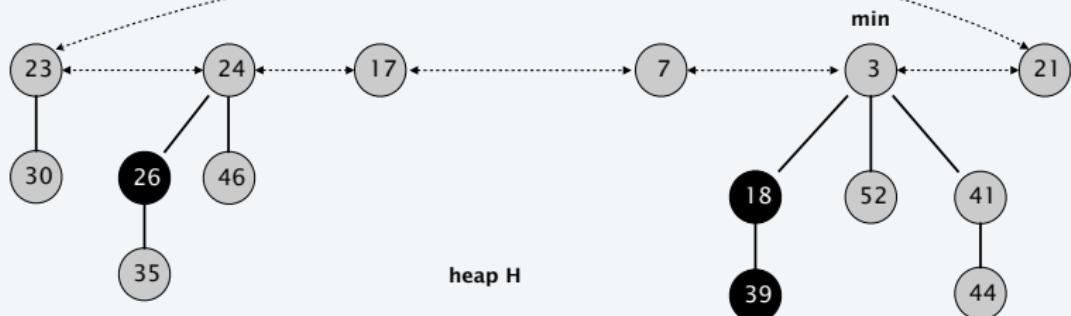
Fibonacci heap: meld analysis

Actual cost. $c_i = O(1)$.

Change in potential. $\Delta\Phi = 0$.

Amortized cost. $\hat{c}_i = c_i + \Delta\Phi = O(1)$.

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$



Fibonacci heap: delete

Delete. Given a handle to an element x , delete it from heap H .

- $\text{DECREASE-KEY}(H, x, -\infty)$.
- $\text{EXTRACT-MIN}(H)$.

Amortized cost. $\hat{c}_i = O(\text{rank}(H))$.

- $O(1)$ amortized for DECREASE-KEY .
- $O(\text{rank}(H))$ amortized for EXTRACT-MIN .

$$\Phi(H) = \text{trees}(H) + 2 \cdot \text{marks}(H)$$

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	amortized Fibonacci heap †
MAKE-HEAP amortized	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† amortized

Accomplished. $O(1)$ INSERT and DECREASE-KEY, $O(\log n)$ EXTRACT-MIN.

PRIORITY QUEUES

- ▶ *binary heaps*
- ▶ *d-ary heaps*
- ▶ *binomial heaps*
- ▶ *Fibonacci heaps*
- ▶ *advanced topics*

Heaps of heaps

- b-heaps.
- Fat heaps.
- 2-3 heaps.
- Leaf heaps.
- Thin heaps.
- Skew heaps.
- Splay heaps.
- Weak heaps.
- Leftist heaps.
- Quake heaps.
- Pairing heaps.
- Violation heaps.
- Run-relaxed heaps.
- Rank-pairing heaps.
- Skew-pairing heaps.
- Rank-relaxed heaps.
- Lazy Fibonacci heaps.



Brodal queues

Q. Can we achieve same running time as for Fibonacci heap but with worst-case bounds per operation (instead of amortized)?

Theory. [Brodal 1996] Yes.

Worst-Case Efficient Priority Queues*

Gerth Stølting Brodal†

Abstract

An implementation of priority queues is presented that supports the operations MAKEQUEUE, FINDMIN, INSERT, MELD and DECREASEKEY in worst case time $O(1)$ and DELETEMIN and DELETE in worst case time $O(\log n)$. The space requirement is linear. The data structure presented is the first achieving this worst case performance.

Practice. Ever implemented? Constants are high (and requires RAM model).

Strict Fibonacci heaps

Q. Can we achieve same running time as for Fibonacci heap but with worst-case bounds per operation (instead of amortized) in **pointer model**?

Theory. [Brodal-Lagogiannis-Tarjan 2002] Yes.

Gerth Stølting Brodal
MADALGO^{*}
Dept. of Computer Science
Aarhus University
Åbogade 34, 8200 Aarhus N
Denmark
gerth@cs.au.dk

George Lagogiannis
Agricultural University
of Athens
Iera Odos 75, 11855 Athens
Greece
lagogiann@hua.gr

Robert E. Tarjan[†]
Dept. of Computer Science
Princeton University
and HP Labs
35 Olden Street, Princeton
New Jersey 08540, USA
ret@cs.princeton.edu

ABSTRACT

We present the first pointer-based heap implementation with time bounds matching those of Fibonacci heaps in the worst case. We support make-heap, insert, find-min, meld and decrease-key in worst-case $O(1)$ time, and delete and delete-min in worst-case $O(\lg n)$ time, where n is the size of the heap. The data structure uses linear space.

A previous, very complicated, solution achieving the same time bounds in the RAM model made essential use of arrays and extensive use of redundant counter schemes to maintain balance. Our solution uses neither. Our key simplification is to discard the structure of the smaller heap when doing a meld. We use the pigeonhole principle in place of the redundant counter mechanism.

Fibonacci heaps: practice

- Q. Are Fibonacci heaps useful in practice?
- A. They are part of LEDA and Boost C++ libraries.
(but other heaps seem to perform better in practice)



Pairing heaps

Pairing heap. A self-adjusting heap-ordered general tree.

The Pairing Heap: A New Form of Self-Adjusting Heap

Michael L. Fredman^{1,4}, Robert Sedgewick^{2,5}, Daniel D. Sleator³, and Robert E. Tarjan^{2,3,6}

Abstract. Recently, Fredman and Tarjan invented a new, especially efficient form of heap (priority queue) called the *Fibonacci heap*. Although theoretically efficient, Fibonacci heaps are complicated to implement and not as fast in practice as other kinds of heaps. In this paper we describe a new form of heap, called the *pairing heap*, intended to be competitive with the Fibonacci heap in theory and easy to implement and fast in practice. We provide a partial complexity analysis of pairing heaps. Complete analysis remains an open problem.

Theory. Same amortized running times as Fibonacci heaps for all operations except DECREASE-KEY.

- $O(\log n)$ amortized. [Fredman et al. 1986]
- $\Omega(\log \log n)$ lower bound on amortized cost. [Fredman 1999]
- $2\sqrt{O(\log \log n)}$ amortized. [Pettie 2005]

Pairing heaps

Pairing heap. A self-adjusting heap-ordered general tree.

Practice. As fast as (or faster than) the binary heap on some problems.
Included in GNU C++ library and LEDA.

The image shows the cover of a research paper titled "Pairing Heaps: Experiments and Analysis". The cover is white with black text. At the top left, it says "RESEARCH CONTRIBUTION". Below that, it lists the journal title "Algorithms and Data Structures" and the editor "G. Scott Graham". The main title "Pairing Heaps: Experiments and Analysis" is centered in large, bold, black font. Below the title, the authors' names "JOHN T. STASKO and JEFFREY SCOTT VITTER" are listed. The abstract and body text are visible at the bottom of the page.

ABSTRACT: The pairing heap has recently been introduced as a new data structure for priority queues. Pairing heaps are extremely simple to implement and seem to be competitive with other priority queues. It has been conjectured that they achieve the same amortized times bounds as Fibonacci heaps. Only one operation, *decrease_key*, is slower than *min* and *find* for all other operations, where n is the size of the priority queue at the time of the operation. We provide empirical evidence that supports this conjecture. The most prominent algorithmic application of pairing heaps is in the *kruskal* algorithm for minimum spanning trees. We prove that, assuming no *decrease_key* operations are performed, it achieves the same associated time bounds as Fibonacci heaps.

and practical importance from their use in solving a wide range of combinatorial problems, including job scheduling, minimal spanning tree, shortest path, and graph traversal.

Priority queues support the operations *insert*, *find_min*, and *delete_min*; additional operations often include *decrease_key* and *delete*. The *insert*, (c) *operator* and *decrease_key* operations are supported by the *pairing heap* queue. The *find_min* operation returns the item with minimum key value. The *delete_min* operation returns the item with minimum key value and removes it from the priority queue. The *decrease_key*, (d) *operator* decreases item i 's key value by d . The *delete*(i) operation removes item i from the priority queue. The *decrease_key* and *delete* operations require that a pointer to the location in the

Priority queues performance cost summary

operation	linked list	binary heap	binomial heap	pairing heap †	Fibonacci heap †	Brodal queue
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$2\sqrt{O(\log \log n)}$	$O(1)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$

† amortized

Priority queues with integer priorities

Assumption. Keys are integers between 0 and C .

Theorem. [Thorup 2004] There exists a priority queue that supports `INSERT`, `FIND-MIN`, and `DECREASE-KEY` in constant time and `EXTRACT-MIN` and `DELETE-KEY` in either $O(\log \log n)$ or $O(\log \log C)$ time.

Available online at www.sciencedirect.com
 SCIENCE @ DIRECT[®]
Journal of Computer and System Sciences 69 (2004) 330–353
<http://www.elsevier.com/locate/jcss>

Integer priority queues with decrease key in constant time and
the single source shortest paths problem

Mikkel Thorup
AT&T Labs Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA
Received 22 July 2003; revised 8 March 2004
Available online 20 July 2004

Abstract

We consider Fibonacci heap style integer priority queues supporting find-min, insert, and decrease key operations in constant time. We present a deterministic linear space solution that with n integer keys supports delete in $O(\log \log n)$ time. If the integers are in the range $[0, N]$, we can also support delete in $O(\log \log N)$ time.

Priority queues with integer priorities

Assumption. Keys are integers between 0 and C .

Theorem. [Thorup 2004] There exists a priority queue that supports `INSERT`, `FIND-MIN`, and `DECREASE-KEY` in constant time and `EXTRACT-MIN` and `DELETE-KEY` in either $O(\log \log n)$ or $O(\log \log C)$ time.

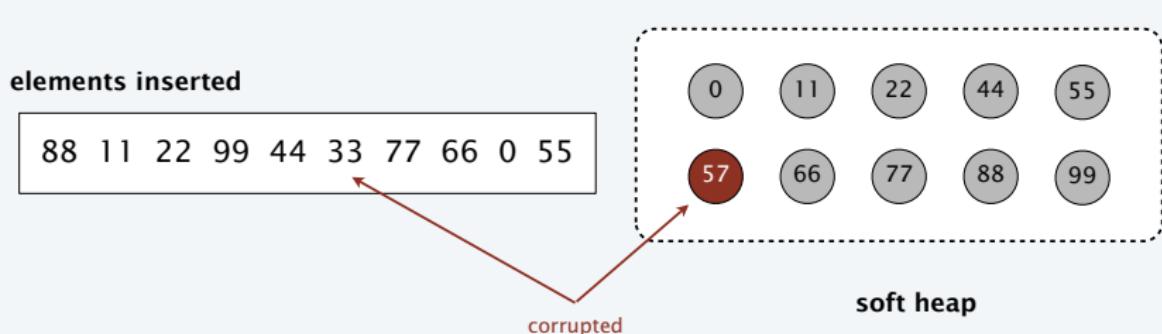
Corollary 1. Can implement Dijkstra's algorithm in either $O(m \log \log n)$ or $O(m \log \log C)$ time.

Corollary 2. Can sort n integers in $O(n \log \log n)$ time.

Computational model. Word RAM.

Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).



Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

Representation.

- Set of binomial trees (with some subtrees missing).
- Each node may store several elements.
- Each node stores a value that is an **upper bound** on the original keys.
- Binomial trees are heap-ordered with respect to these values.

Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

Theorem. [Chazelle 2000] Starting from an empty soft heap, any sequence of n **INSERT**, **MIN**, **EXTRACT-MIN**, **MELD**, and **DELETE** operations takes $O(n)$ time and at most 10% of its elements are corrupted at any given time.

The Soft Heap: An Approximate Priority Queue with Optimal Error Rate

BERNARD CHAZELLE

Princeton University, Princeton, New Jersey, and NEC Research Institute

Abstract. A simple variant of a priority queue, called a *soft heap*, is introduced. The data structure supports the usual operations: insert, delete, meld, and findmin. Its novelty is to beat the logarithmic bound on the complexity of a heap in a comparison-based model. To break this information-theoretic barrier, the entropy of the data structure is reduced by artificially raising the values of certain keys. Given any mixed sequence of n operations, a soft heap with error rate ϵ (for any $0 < \epsilon \leq 1/2$) ensures that, at any time, at most ϵn of its items have their keys raised. The amortized complexity of each operation is constant, except for insert, which takes $O(\log 1/\epsilon)$ time. The soft heap is optimal for any value of ϵ in a comparison-based model. The data structure is purely pointer-based. No arrays are used and no numeric assumptions are made on the keys. The main idea behind the soft heap is to move items across the data structure not individually, as is customary, but in groups, in a data-structuring equivalent of “car pooling.” Keys must be raised as a result, in order to preserve the heap ordering of the data structure. The soft heap can be used to compute exact or approximate medians and percentiles optimally. It is also useful for approximate sorting and for computing minimum spanning trees of general graphs.

Soft heaps

Goal. Break information-theoretic lower bound by allowing priority queue to corrupt 10% of the keys (by increasing them).

Q. Brilliant. But how could it possibly be useful?

Ex. Linear-time deterministic selection. To find k^{th} smallest element:

- Insert the n elements into soft heap.
- Extract the minimum element $n / 2$ times.
- The largest element deleted $\geq 4n / 10$ elements and $\leq 6n / 10$ elements.
- Can remove $\geq 5n / 10$ of elements and recur.
- $T(n) \leq T(3n / 5) + O(n) \Rightarrow T(n) = O(n)$. ▀

Soft heaps

Theorem. [Chazelle 2000] There exists an $O(m \alpha(m, n))$ time deterministic algorithm to compute an MST in a graph with n nodes and m edges.

Algorithm. Borůvka + nongreedy + divide-and-conquer + soft heap + ...

A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity

BERNARD CHAZELLE

Princeton University, Princeton, New Jersey, and NEC Research Institute

Abstract. A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m, n))$, where α is the classical functional inverse of Ackermann's function and n (respectively, m) is the number of vertices (respectively, edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.