

Advanced Data Structures

Conrado Martínez
U. Politècnica de Catalunya

March 11, 2024





Consent for reproduction, modification, or use, whether total or partial, of these slides for exclusively educational purposes is automatically granted, both at the Polytechnic University of Catalonia (UPC) and at any other educational institution, whether higher or not, in any country, as long as: a) due credit is given to the author or authors; b) the copyright, reproduction, modification, and use rights, whether total or partial, applicable to derivative materials, are the same as those expressed here. Please send an email to [conrado
-at- cs.upc.edu](mailto:conrado-at-cs.upc.edu) if you are interested in reproducing, modifying, or using these slides, whether in whole or in part, for any purpose, including educational purposes.

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: Binary Search Trees
- 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):

- 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 #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

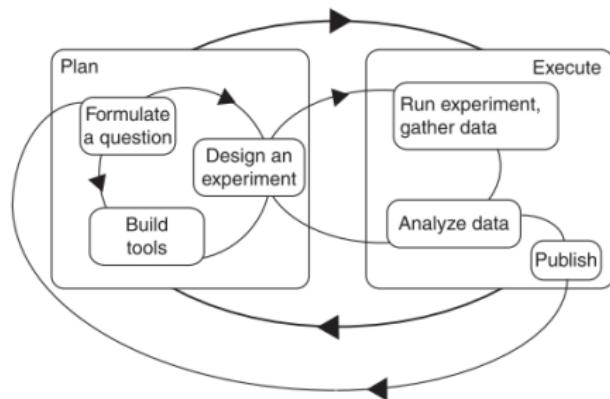
- Benchmarks are not experiments; they are observations, much like observing the behavior of animals in the wild.
- They are a nice source of observational data, to be explained and complemented by experiments (under controlled conditions).
- They are also good (although far from complete!) to check 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: Binary Search Trees
- Probabilistic Tools

3 Amortized Analysis

Part I

Techniques

- 1 Experimental Algorithmics
- 2 Probabilistic Analysis of Algorithms
 - The Continuous Master Theorem
 - Example: Binary Search Trees
 - 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: Analyzing Binary Search Trees

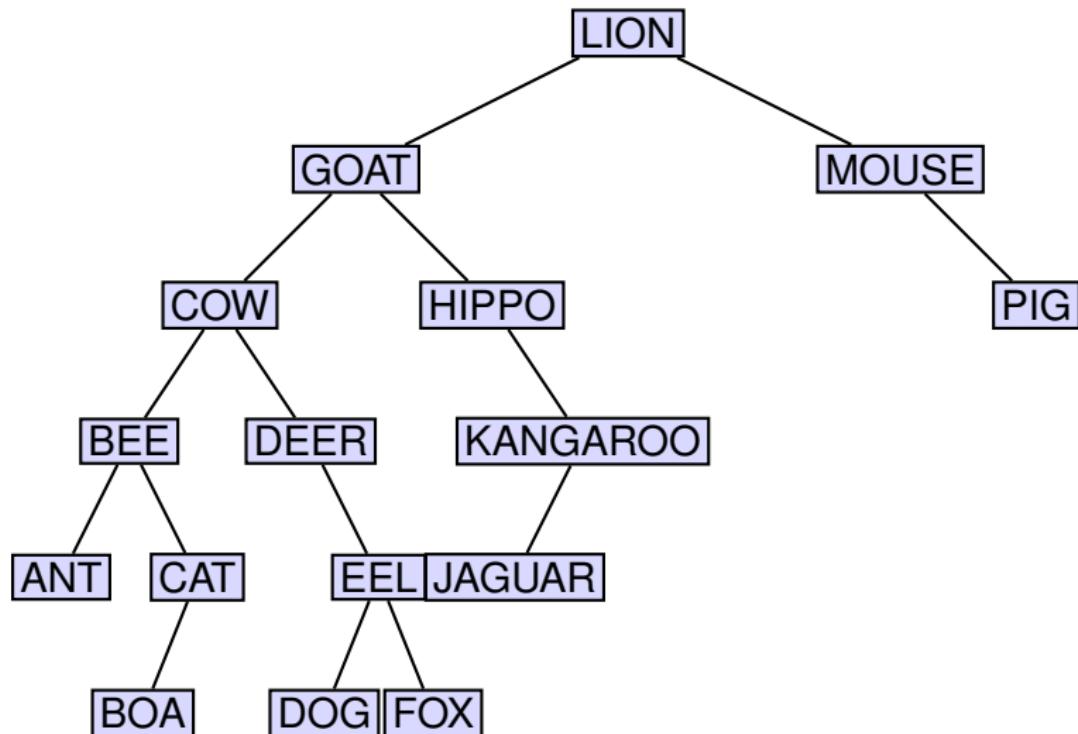
Definition

A **binary search tree** (BST, for short) T is a binary tree that is either empty, or it contains at least one element x at its root, and

- ① The left and right subtrees of T , L and R , respectively, are binary search trees.
- ② For all elements y in L , $\text{KEY}(y) < \text{KEY}(x)$, and for all elements z in R , $\text{KEY}(z) > \text{KEY}(x)$.

Example: Analyzing Binary Search Trees

A binary search tree for a set of 16 strings



Traversals in BSTs

Lemma

An *inorder traversal* of a binary search tree visits all the elements it contains in ascending order of their keys.

Traversals are often implemented via **iterators**; then the class offers methods like `begin` and `end` (among others) for the traversals.

Search in BSTs

Let us consider now the search algorithm in a BST. Because of their recursive definition of BST, we will start with a recursive algorithm.

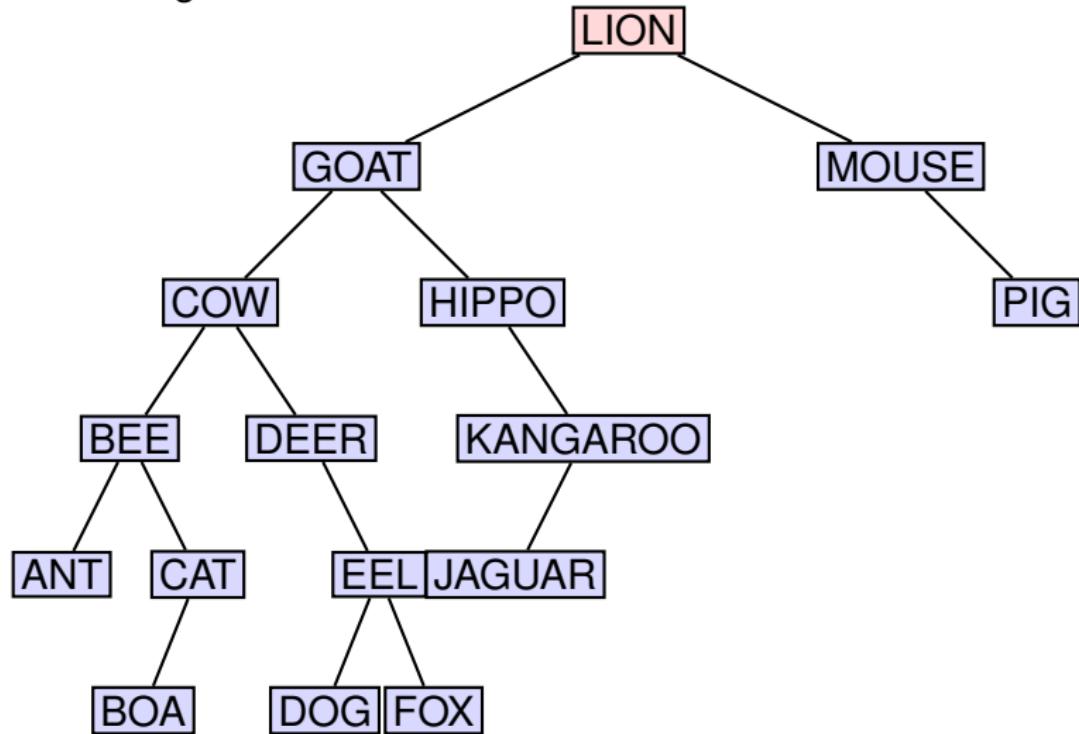
Let T be a BST representing the dictionary and let k be the key we are looking for. If $T = \square$ then k is not in the dictionary and we need only to signal this event in some convenient way (e.g. we return `false`). If T is not empty we will need to check the relation between k and the key of the element x stored at the root of T .

Search in BSTs

If $k = \text{KEY}(x)$ the search is successful and we can stop it there, returning x or the information associated to x which we care about. If $k < \text{KEY}(x)$, following the definition of BSTs, if there exists an element in T with key k it must be stored in the left subtree of T ; hence, we must make a recursive call on the left subtree of T to continue the search. Analogously, if $k > \text{KEY}(x)$ then the search must recursively continue in the right subtree of T .

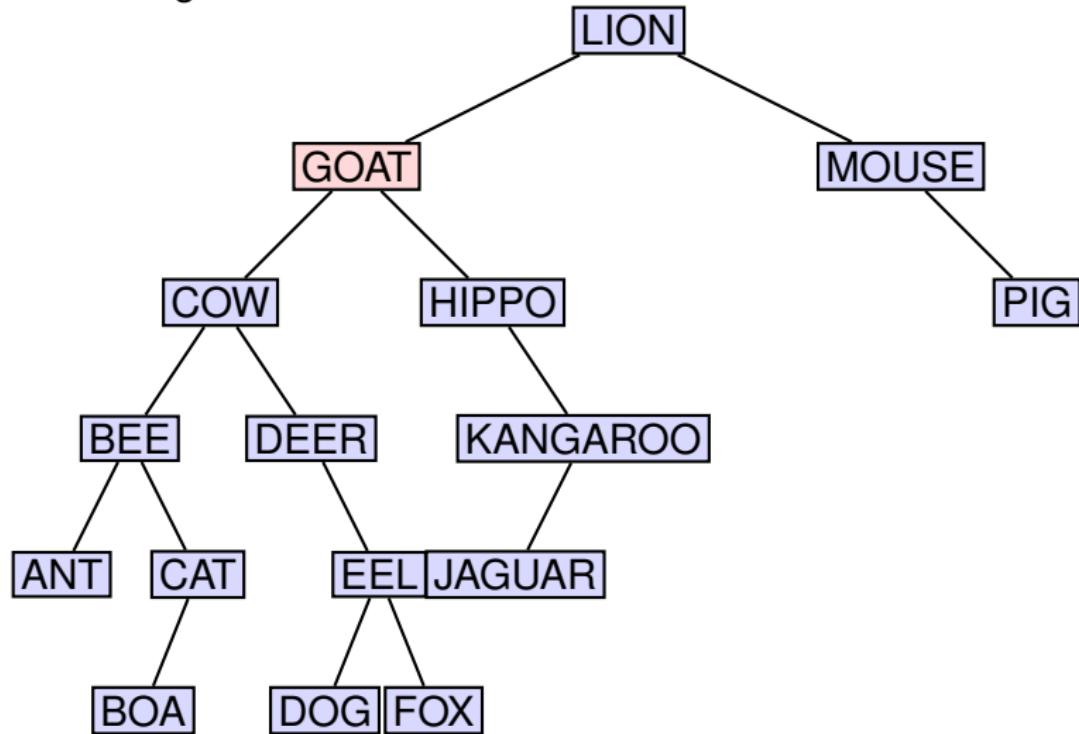
Searchin BSTs: An Example

Searching for DOG



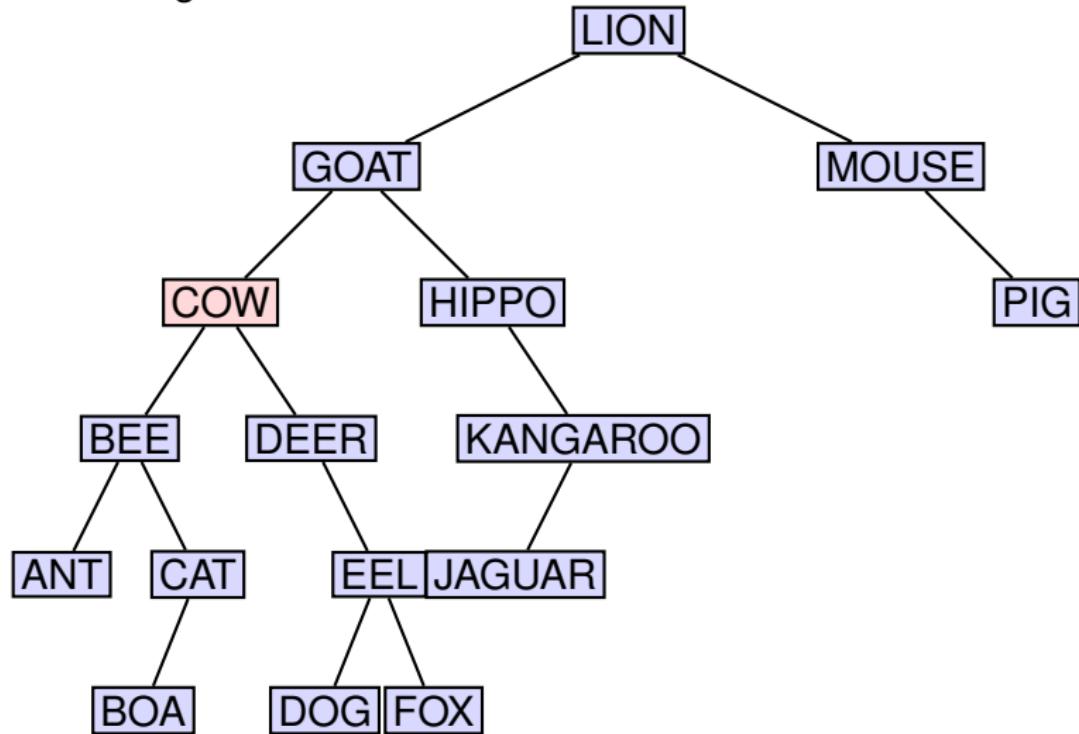
Search in BSTs: An Example

Searching for DOG



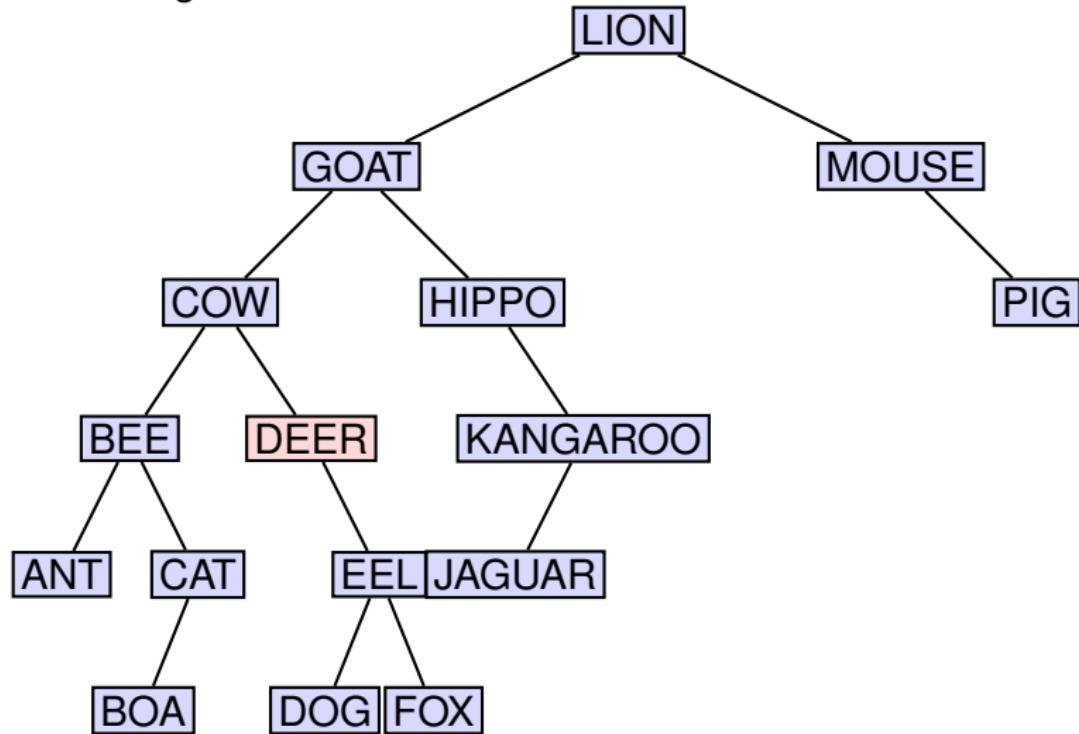
Search in BSTs: An Example

Searching for DOG



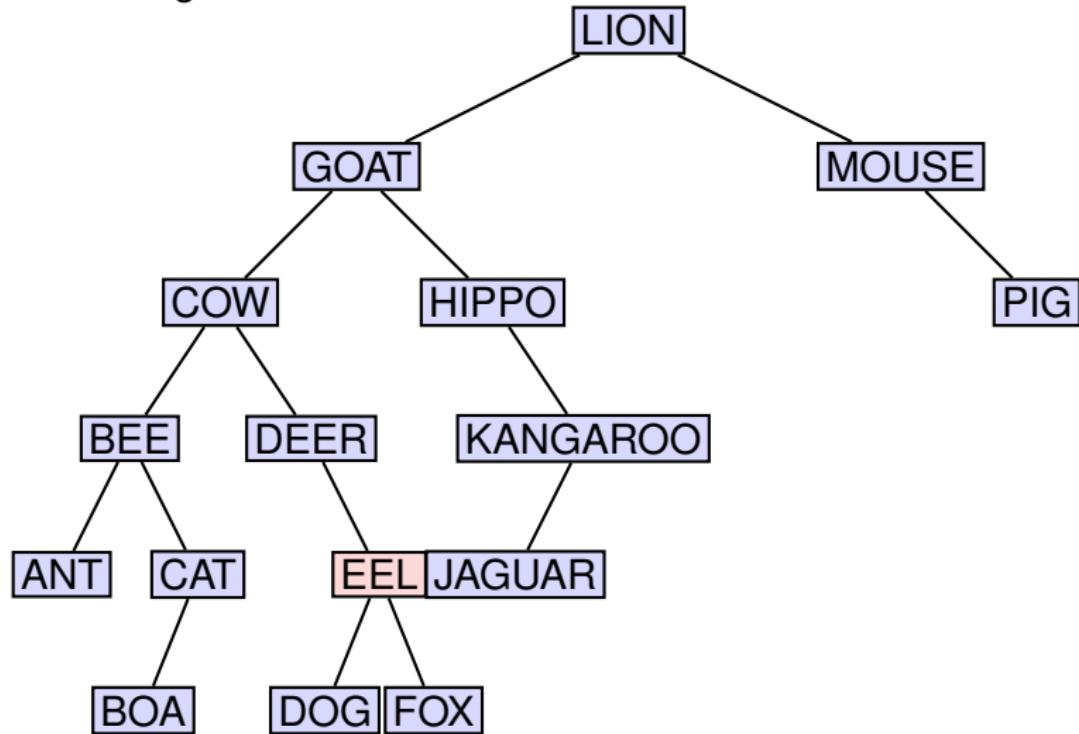
Search in BSTs: An Example

Searching for DOG



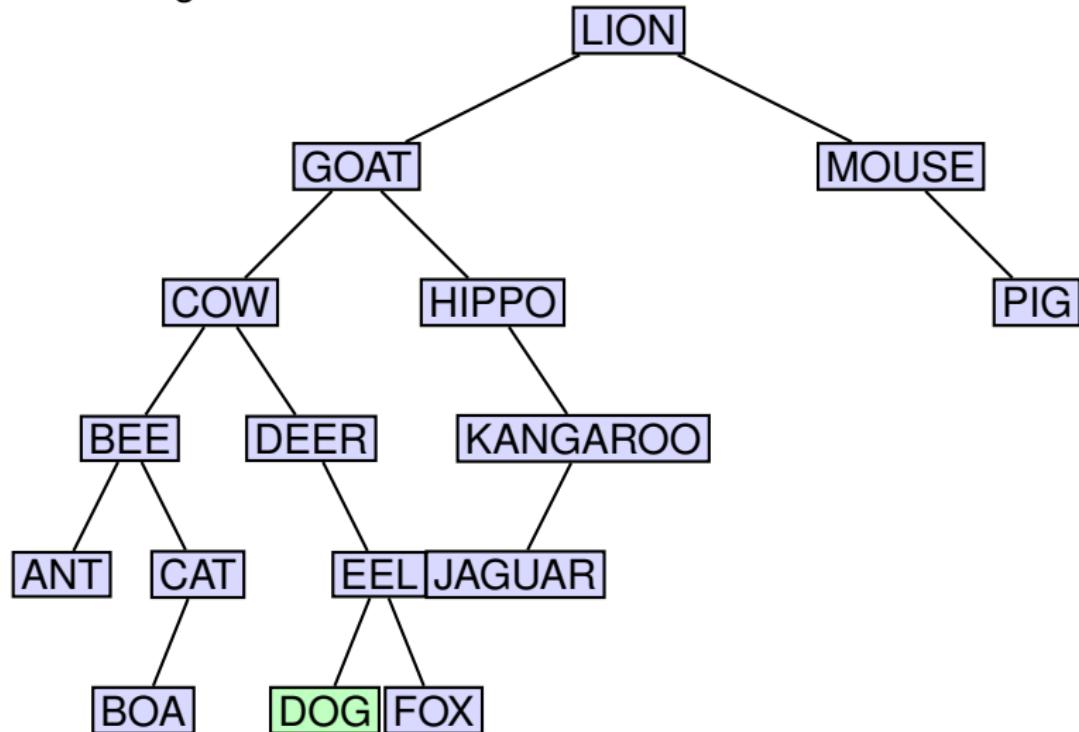
Search in BSTs: An Example

Searching for DOG



Search in BSTs: An Example

Searching for DOG



Insertions in BSTs

The insertion algorithm is also very easy and we can design it by using the same reasoning as for the search algorithm: if the new key to be inserted is smaller than the key at the root insert in the left subtree; otherwise, insert into the right subtree. When the current subtree is empty (*a leaf*), it is replaced by a new node containing the given key and value and empty subtrees. If the given key already appears in the tree then no new element is inserted, but we might change the value associated to the key to a new given value.

The Cost of Searches and Updates in BSTs

A BST of size n can be equivalent to a linked list, with one node with no children and $n - 1$ nodes with only one non-empty subtree. Such a tree can be built by inserting n elements in ascending or descending order of their keys into an initially empty BST.

Thus in the height of a BST can be as high as n ; in the opposite extreme a perfectly balanced BST of size n has height at least $\lceil \log_2(n + 1) \rceil$.

For a BST of height h the worst-case cost of a search, an insertion or a deletion has cost $\Theta(h)$. Hence, the worst-case for search and update operations in a BST of size n is $\Theta(n)$.

The Cost of Searches and Updates in BSTs

However, on average, the height of a random BST is $\Theta(\log n)$, and thus the average cost of the basic operations is $\Theta(\log n)$ too.

By a **random BST** of size n , we mean a tree built by n insertions, with all possible $n!$ orderings of the n insertions being equally likely.

For successful searches we will assume that the sought element is any of the n elements in the tree; for unsuccessful searches (and insertions) we will assume that the search ends at any of the $n + 1$ empty subtrees (leaves) with identical probability.

The Cost of Searches and Updates in BSTs

Moreover, we will consider as our measure of cost only the number of comparisons made between the given key and the keys of the nodes that we examine. The cost of the operation will be proportional to this number of comparisons. Let $C(n)$ be the expected number of comparisons made by a successful search on a random BST of size n , and $C(n, k)$ the expected number of comparisons conditioned to the root storing the element with the k -th smallest key. Then

$$C(n) = \sum_{1 \leq k \leq n} C(n; k) \times \mathbb{P}[\text{root is the } k\text{-th}] .$$

The Cost of Searches and Updates in BSTs

$$\begin{aligned}C(n) &= \frac{1}{n} \sum_{1 \leq k \leq n} C(n; k) \\&= 1 + \frac{1}{n} \sum_{1 \leq k \leq n} \left(\frac{1}{n} \cdot 0 + \frac{k-1}{n} \cdot C(k-1) + \frac{n-k}{n} \cdot C(n-k) \right) \\&= 1 + \frac{1}{n^2} \sum_{0 \leq k < n} (k \cdot C(k) + (n-1-k) \cdot C(n-1-k)) \\&= 1 + \frac{2}{n^2} \sum_{0 \leq k < n} k \cdot C(k).\end{aligned}$$

An alternative way to analyze the cost of successful searches is to consider the so-called **Internal Path Length** (IPL). Given a BST, its IPL is the sum of the depths from the root to every other node in the tree. The *External Path Length* (EPL) is defined similarly and it is used in the analysis of unsuccessful searches/insertions.

The Cost of Searches and Updates in BSTs

If $I(n)$ denotes the expected value of the IPL of a random BST of size n then

$$C(n) = 1 + \frac{I(n)}{n}$$

The expected value of the IPL satisfies the following recurrence:

$$I(n) = n - 1 + \frac{2}{n} \sum_{0 \leq k < n} I(k), \quad I(0) = 0. \quad (1)$$

Indeed, every node except the root of the tree contributes to the total IPL one unit plus its contribution to the IPL of the subtree of the root to which the node belongs.

Note also that the IPL of a tree does not only give us the cost of successful searches in that tree; it is also the cost of building that tree starting from an empty tree.

The Cost of Searches and Updates in BSTs

Recurrence (1) is identical to the recurrence for the cost of quicksort!!

That's no coincidence. The recursion tree associated to a quicksort execution is a BST of size n : each node holds the pivot used at the corresponding recursive call. Now, every element in the tree has been compared (not being the pivot) against every pivot selected in the sequence of recursive calls leading from the initial call to the recursive call where the element is finally chosen as the pivot. That is, every element is compared to all its proper ancestors in the BSTs, or in other terms, the number of comparisons it gets involved (not as a pivot) is its depth in the BST. Hence $I(n) = Q(n)$, where $Q(n)$ is the expected number of comparisons made by quicksort to sort an array of size n .

The Cost of Searches and Updates in BSTs

We apply CMT to solve the IPL recurrence for random BSTs 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,

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

The Cost of Searches and Updates in BSTs

Since $H_n = \ln n + \mathcal{O}(1)$ we have

$$\begin{aligned} I(n) &= Q(n) = 2n \ln n + \mathcal{O}(n) \\ &= 1.386 \dots n \log_2 n + \mathcal{O}(n) \end{aligned}$$

For a successful search in a BST of size n , the average number of comparisons is then

$$C(n) = 1 + \frac{I(n)}{n} = 2 \ln n + \mathcal{O}(1).$$

The Continuous Master Theorem

To learn more:

-  [S. Roura](#)
Improved Master Theorems for Divide-and-Conquer Recurrences
J. of the ACM 48(2):170–205, 2001
-  [C. Martínez and S. Roura](#)
Optimal Sampling Strategies in Quicksort and Quickselect
SIAM J. on Computing 31(3):683–705, 2001

Part I

Techniques

- 1 Experimental Algorithmics
- 2 Probabilistic Analysis of Algorithms
 - The Continuous Master Theorem
 - Example: Binary Search Trees
 - 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 ($X \geq 0$). For any $a > 0$

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

Markov's Inequality

Proof.

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

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

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

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



N.B. If X is strictly positive then $\mathbb{P}[X > a] < \mathbb{E}[X] / a$.

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

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

In general,

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

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 $1/\log n$:

$$\begin{aligned}\mathbb{P}[|q_n - \mathbb{E}[q_n]| \geq c \mathbb{E}[q_n]] &\leq \frac{\left(7 - \frac{2\pi^2}{3}\right) n^2 + o(n^2)}{4c^2 n^2 \ln^2 n + o(n^2 \log^2 n)} \\ &= \frac{\left(7 - \frac{2\pi^2}{3}\right)}{4c^2 \ln^2 n} + o(1/\log n)\end{aligned}$$

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}

Probabilistic Tools

To learn more:

-  R. Motwani and P. Raghavan
Randomized Algorithms
Cambridge Univ. Press, 1995
-  M. Mitzenmacher and E. Upfal
Probability and Computing: An Introduction to Randomized Algorithms and Probabilistic Analysis
Cambridge Univ. Press, 2005
-  T. Cormen, C. Leiserson, R. Rivest and C. Stein
Introduction to Algorithms, 3rd edition
The MIT Press, 2009

Part I

Techniques

1 Experimental Algorithmics

2 Probabilistic Analysis of Algorithms

- The Continuous Master Theorem
- Example: Binary Search Trees
- 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 n times (increments $\pmod{2^k}$). The counter has k bits. How many bit flips are needed?

Counter value	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_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 n times (increments $\pmod{2^k}$). The counter has k bits. How many bit flips are needed?

Counter value	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_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.



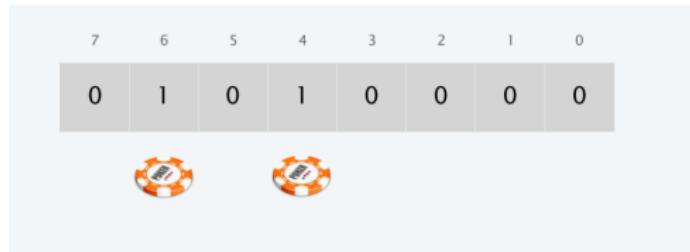
Source: Kevin Wayne

(<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:

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

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 copying $v'[j] := v[j]$ and use the other credit for the copying of $v[j - n/2]$ to $v'[j - n/2]$

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 a unused slot $v'[j + n]$ 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

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

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

Amortized Analysis

To learn more:

-  T. Cormen, C. Leiserson, R. Rivest and C. Stein
Introduction to Algorithms, 3rd ed
The MIT Press, 2009
-  M. T. Goodrich and R. Tamassia
Algorithm Design and Applications
John Wiley & Sons, 2015

Part II

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

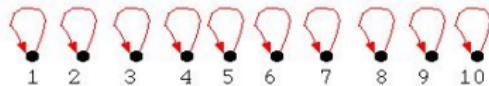
We can avoid scanning the entire array to perform a union; but we will still have to change the representative of all the elements in one block to point to the representative of the other block, and this has linear cost in the worst-case too.

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

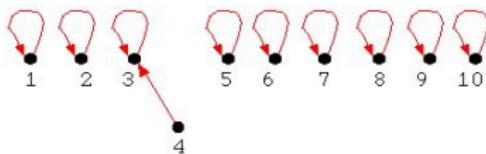
make(10)

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



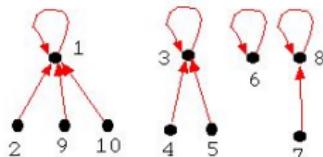
union(3, 4)

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

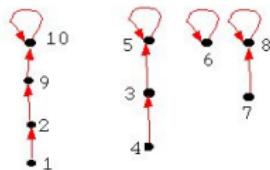
make(10); union(4, 3)

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



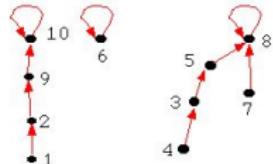
union(1,2); union(4,5); union(1,9);
union(2, 10); union(7,8)

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[rj] += P[ri]; // <= union-by-weight
            // P[rj] = min(P[rj], P[ri]-1); // <= union-by-rank
            P[ri] = rj;
        } else {
            // rj is the smallest/shortest
            P[ri] += P[rj]; // <= union-by-weight
            // P[ri] = min(P[ri], P[rj]-1); // <= union-by-rank
            P[rj] = ri;
        }
        --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:

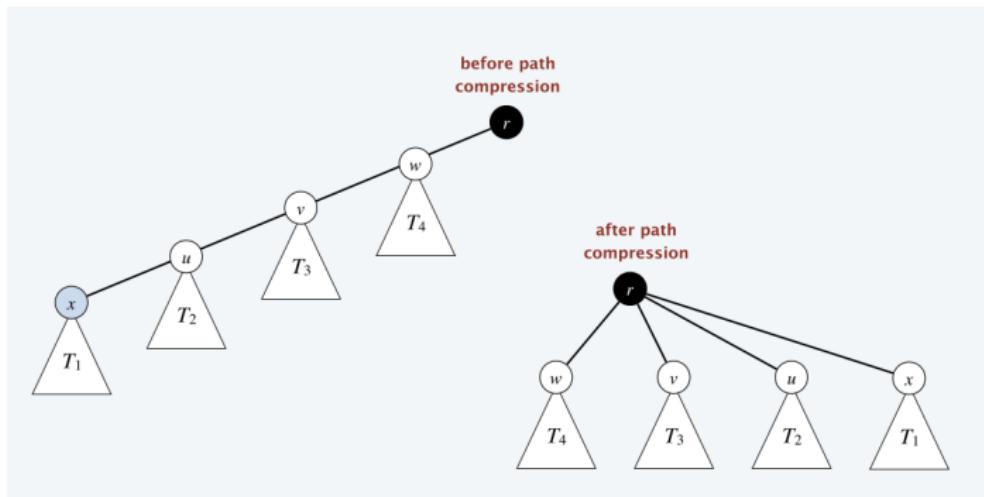
- ➊ In **full path compression**, we traverse the path from i to its representative twice: first to determine that ri is such representative; second, to set $P[k] := ri$ for all k along the path, as all these items have ri as their representative; this only doubles the cost of $\text{FIND}(i)$.
- ➋ In **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 grandparent; 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 grandparent.

Path compression: full path compression

Make every node point to its representative.



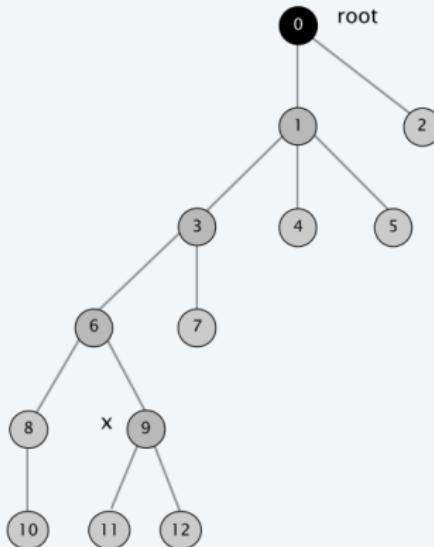
Path compression: full path compression

```
// 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];

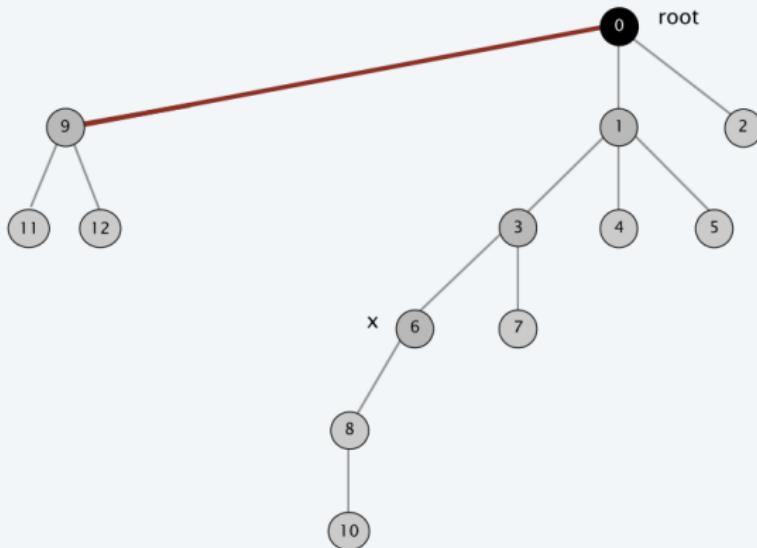
    // traverse the path again making everyone point to ri
    while (P[i] >= 0) {
        int aux = i;
        i = P[i];
        P[aux] = ri;
    }
    return ri;
}

// 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];
    }
}
```

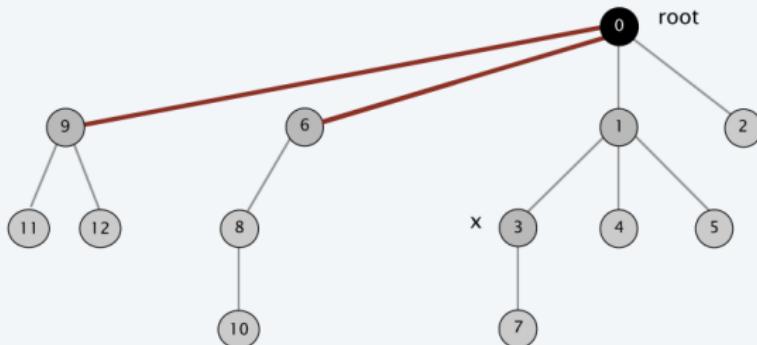
Path compression: full path compression



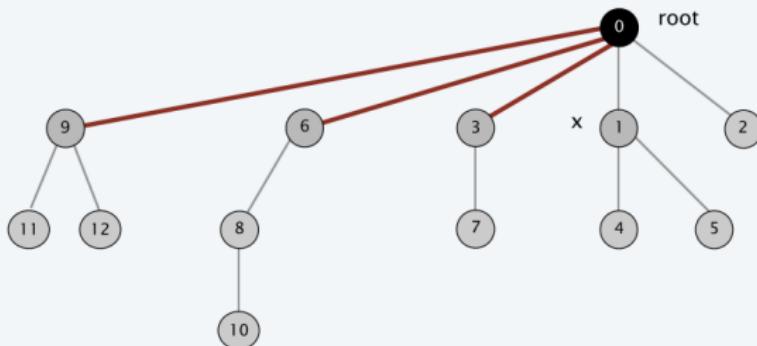
Path compression: full path compression



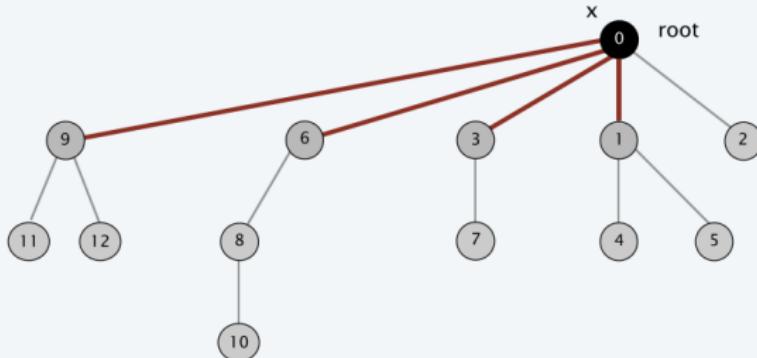
Path compression: full path compression



Path compression: full path compression

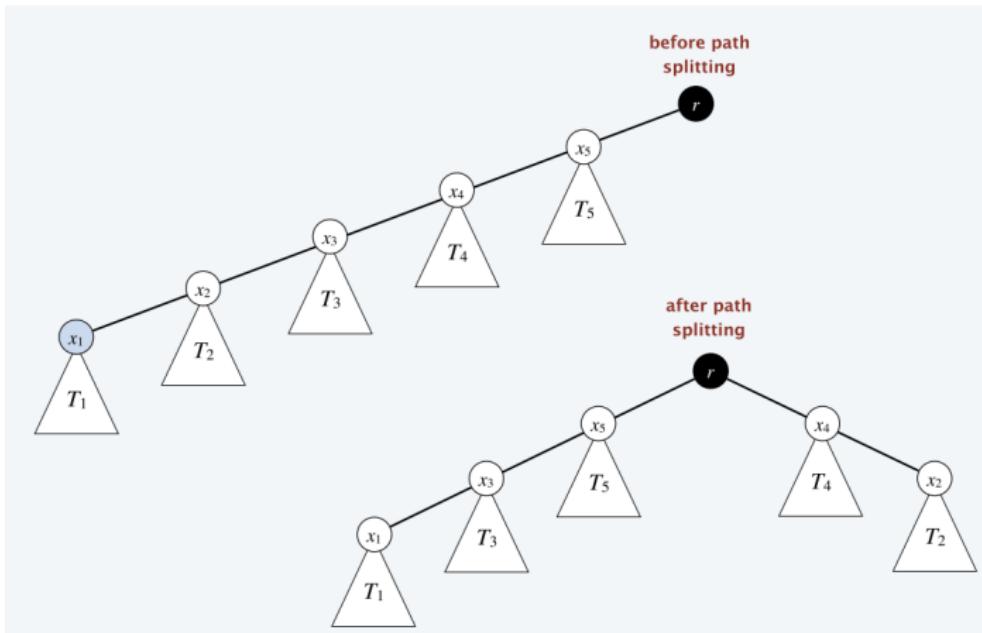


Path compression: full path compression



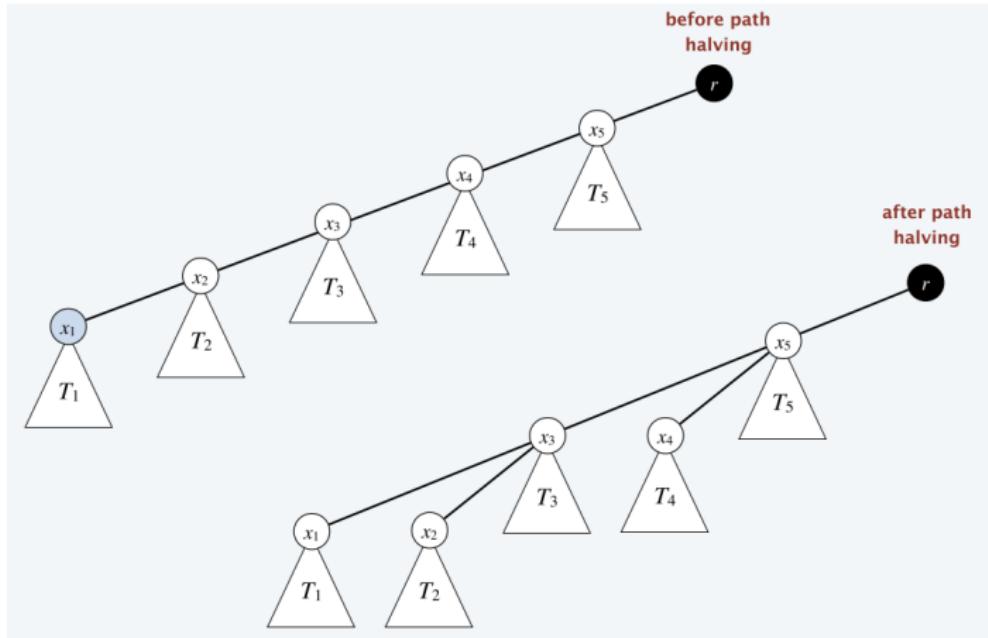
Path compression: path splitting

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



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).



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]) {
            rank[rj] = max(rank[rj], 1+rank[ri]);
            P[ri] = rj;
        } else {
            rank[ri] = max(rank[ri], 1+rank[rj]);
            P[rj] = ri;
        }
    }
}
```

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

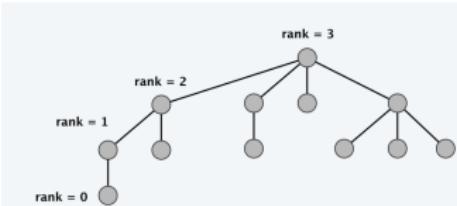
Proof of Property #1

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

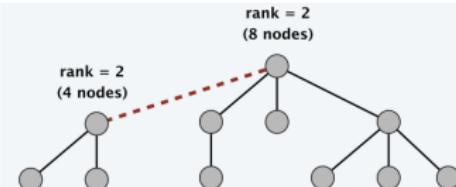
By induction on k . Base: If $k = 0$ then x is the root of a tree with only one node, so the number of descendants is $\geq 2^k$.

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

Since the number of descendants of a root node of rank is $\geq 2^k$, but $2^k \leq n$, it follows that $k \leq \log_2 n$, for otherwise we would have a contradiction.

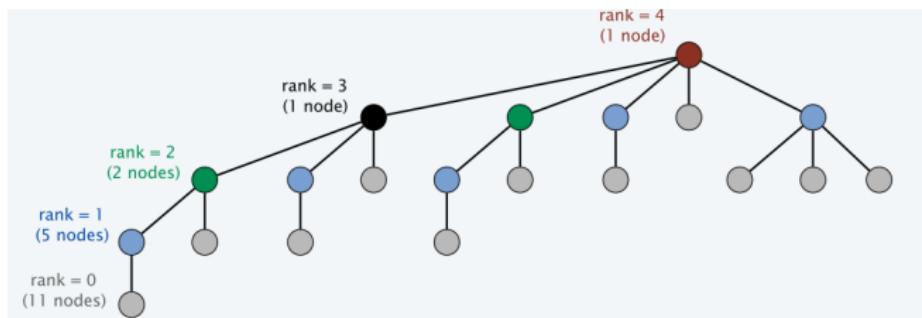


Amortized analysis of Union-Find

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.

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 $2 \uparrow\uparrow k = \underbrace{2^{2^{\dots^2}}}_{k \text{ exponentiations}}$. 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 u such that $\text{rank}(\text{parent}(u))$ and $\text{rank}(u)$ are in the same group to pay 1 credit to follow and update the parent pointer during the FIND.

- Then $\text{rank}(\text{parent}(u))$ 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 u has been updated 2^x times by FIND operations.
- Once the parent's rank belongs to a higher group than the rank of u , the situation will remain, as $\text{rank}(u)$ (hence the group) never changes and $\text{rank}(\text{parent}(u))$ never decreases.

Therefore, u 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.

Acknowledgements

These slides have been substantially inspired by those of Kevin Wayne for his course on *Algorithm Design*:

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

In particular, several pictures in my slides are from Kevin Wayne's slides.

Disjoint Sets

To learn more:

-  **S. Dasgupta, C. Papadimitriou and U. Vazirani**
Algorithms
McGraw-Hill, 2006
-  **T. Cormen, C. Leiserson, R. Rivest and C. Stein**
Introduction to Algorithms, 4th ed
The MIT Press, 2022

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 III

Priority Queues

- 4 Introduction
- 5 Binary Heaps & Heapsort
- 6 Binomial Queues
- 7 Fibonacci Heaps

Part III

Priority Queues

4

Introduction

5

Binary Heaps & Heapsort

- Heapsort

6

Binomial Queues

7

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 III

Priority Queues

4

Introduction

5

Binary Heaps & Heapsort

- Heapsort

6

Binomial Queues

7

Fibonacci Heaps

Heaps

Definition

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

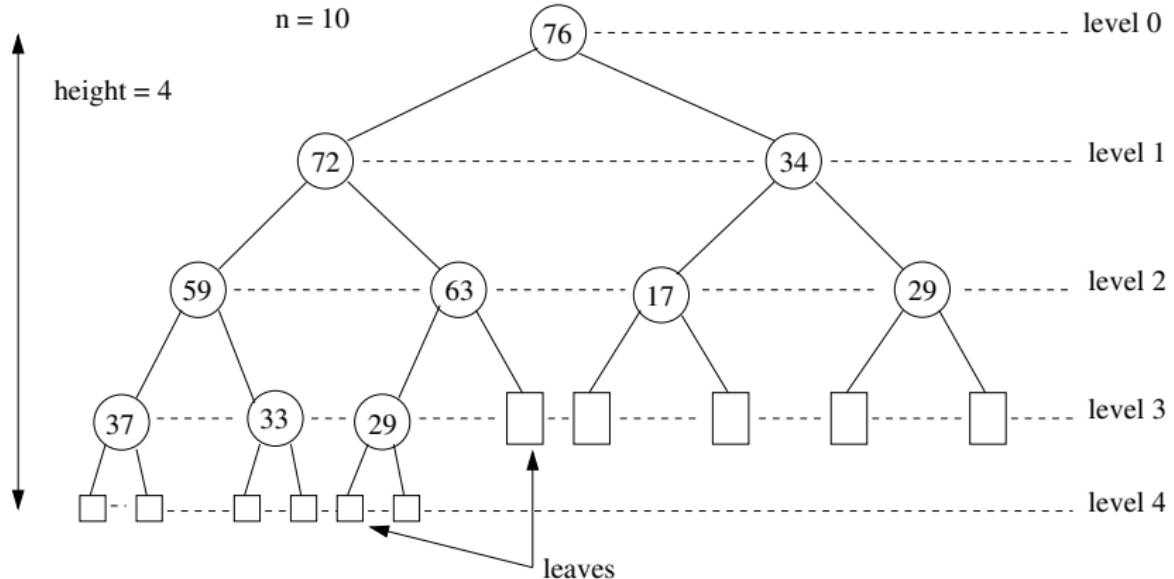
- ① All empty subtrees are located in the last two levels of the tree.
- ② If a node has an empty left subtree then its right subtree is also empty.
- ③ The priority of any element is larger or equal than the priority of any element in its descendants.

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.

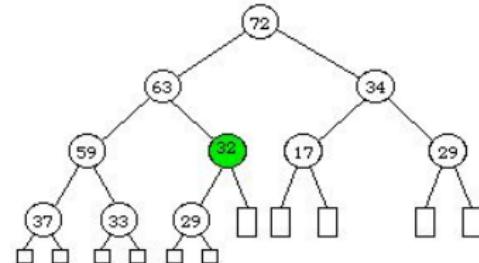
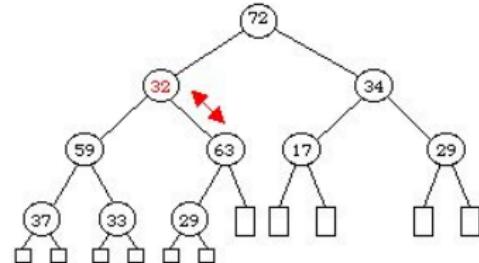
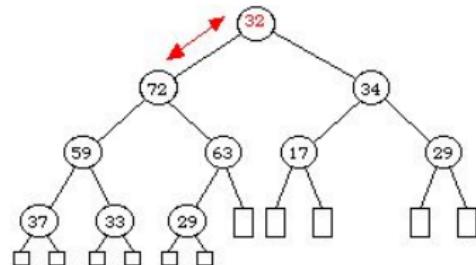
Heaps: Removing the maximum

- 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 Ele, typename Prio>
bool PriorityQueue<Ele,Prio>::empty() const {
    return nelems == 0;
}

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

template <typename Ele, typename Prio>
Prio PriorityQueue<Ele,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 III

Priority Queues

- 4 Introduction
- 5 Binary Heaps & Heapsort
 - Heapsort
- 6 Binomial Queues
- 7 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 $\mathcal{O}(n \log n)$.

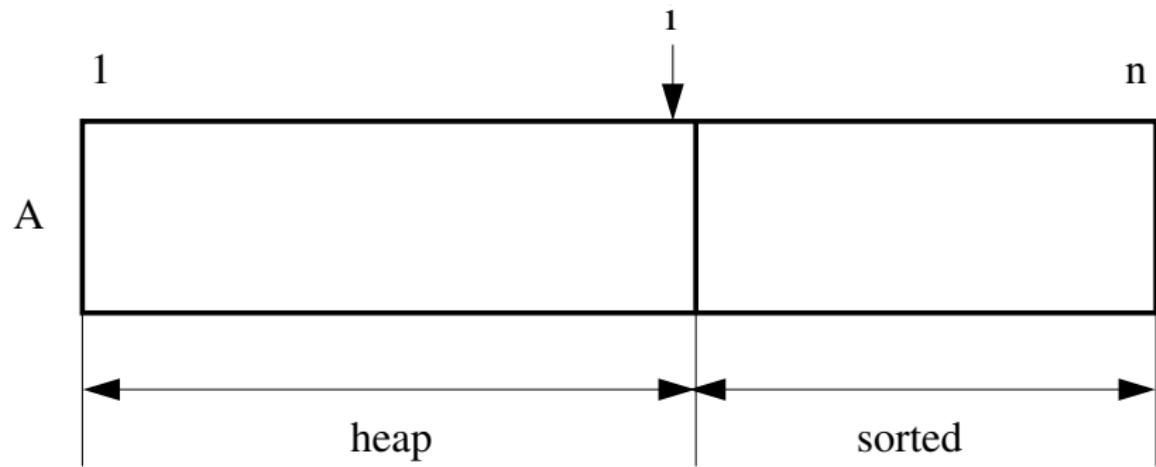
Heapsort

```
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);
    }
}
```

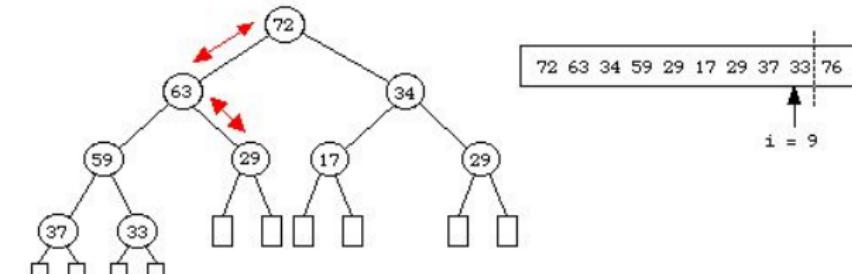
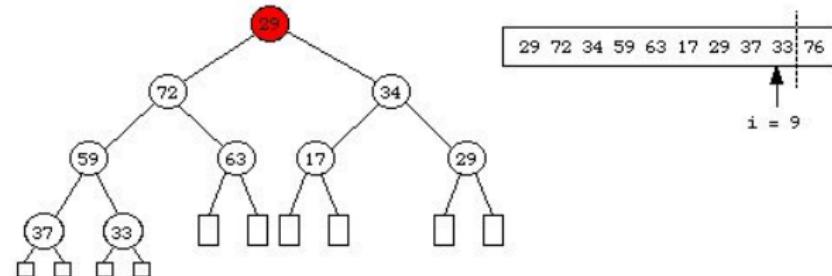
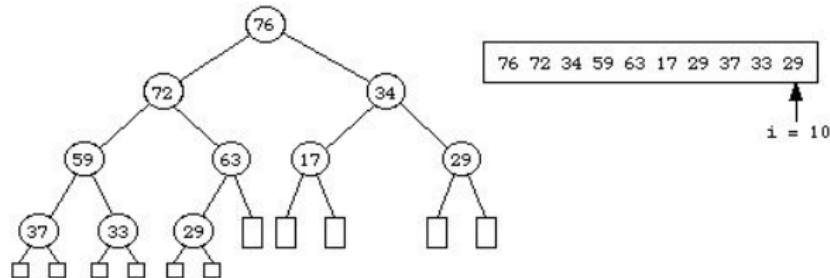
Heapsort



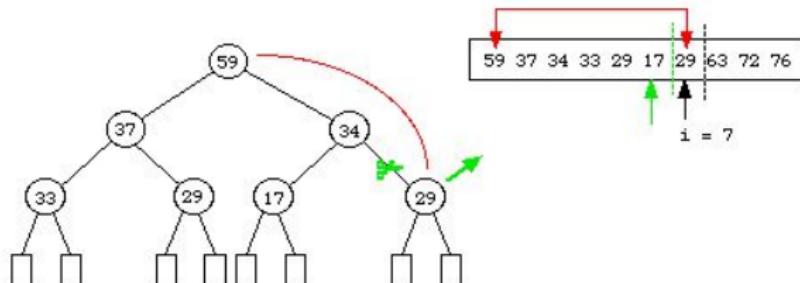
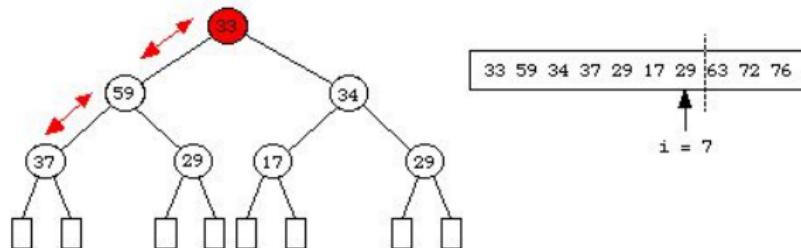
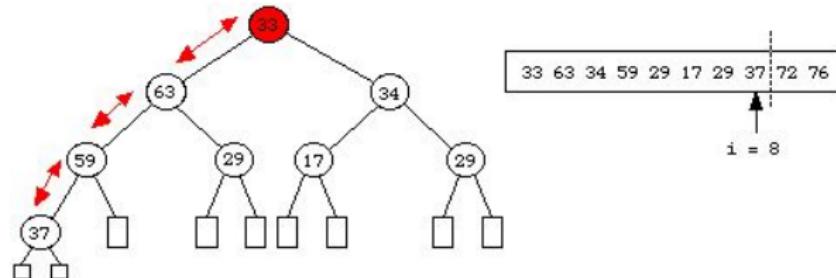
$$A[1] \leq A[i+1] \leq A[i+2] \leq \dots \leq A[n]$$

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

Heapsort



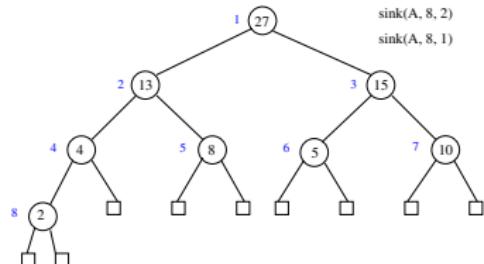
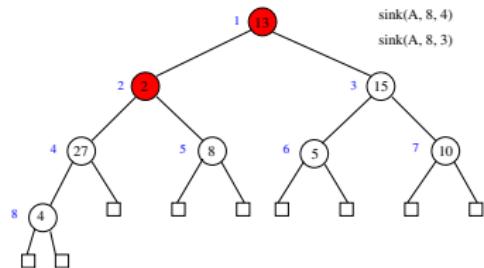
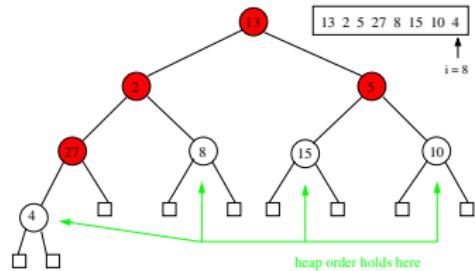
Heapsort



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 III

Priority Queues

4

Introduction

5

Binary Heaps & Heapsort

- Heapsort

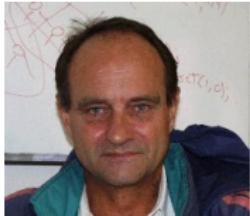
6

Binomial Queues

7

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);

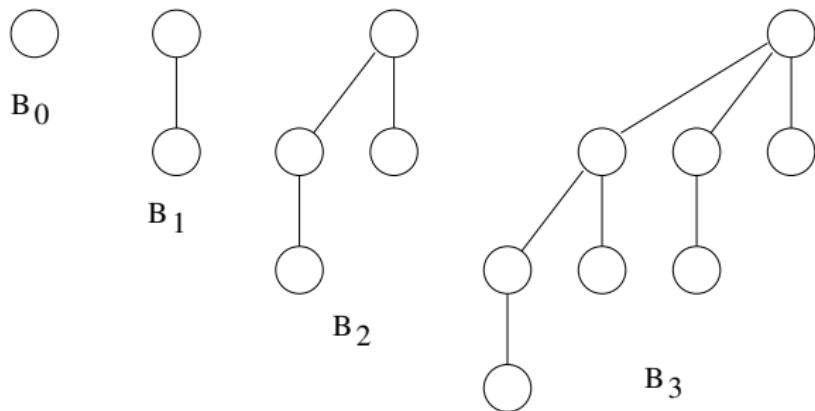
    // Returns true if and only if the queue is empty
    bool empty() const throw();

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

    ...
};
```

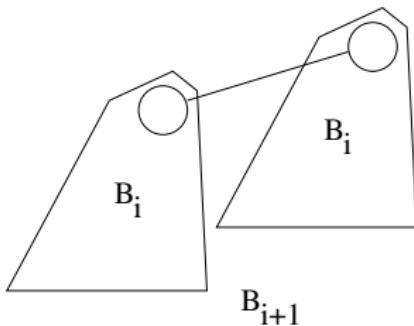
Binomial Queues

- 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 .

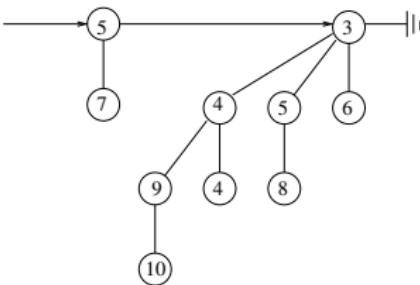


- The size of B_i is 2^i ; indeed $|B_0| = 2^0 = 1$, $|B_{i+1}| = 2 \cdot |B_i| = 2 \cdot 2^i = 2^{i+1}$
- A binomial tree of order i has exactly $\binom{i}{k}$ descendants at level k (the root is at level 0); hence their name
- A binomial tree of order i has height $i = \log_2 |B_i|$

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

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

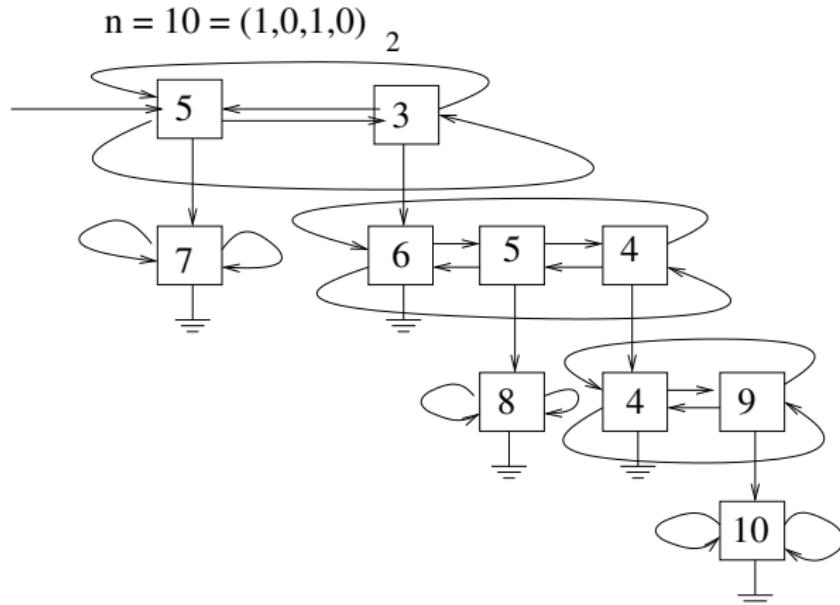


- A binomial queue for n elements contains at most $\lceil \log_2(n + 1) \rceil$ binomial trees
- The n elements of the binomial queue are stored in the binomial trees in such a way that **each binomial tree satisfies the heap property**: the priority of the element at any given node is \leq than the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

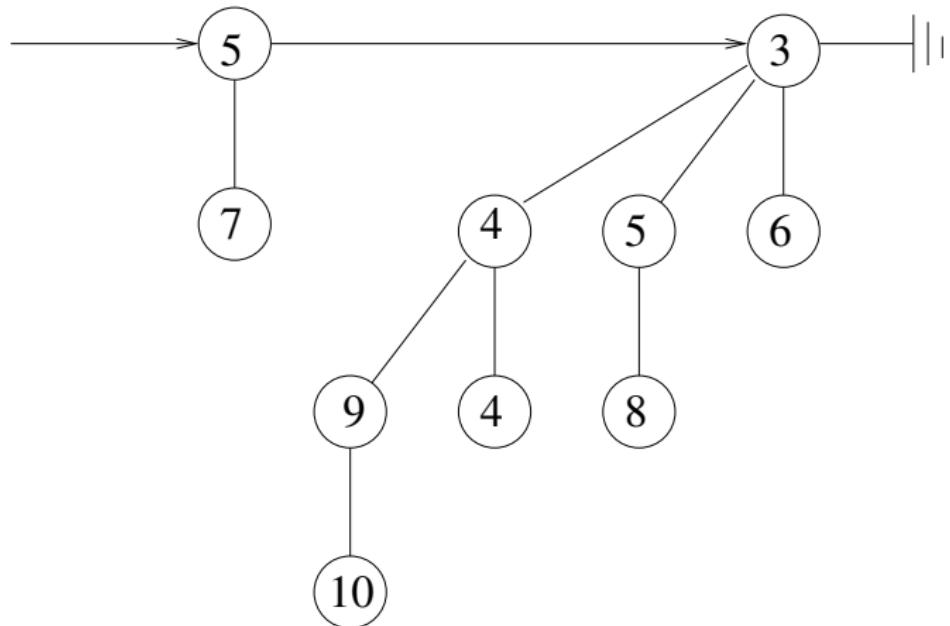
- To delete an element of minimum priority from a queue Q , we start locating such an element, say x ; it must be at the root of some B_i
- The root of B_i is detached from Q and thus B_i is no longer part of the original queue Q ; the list of x 's children is a binomial queue Q' with $2^i - 1$ elements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

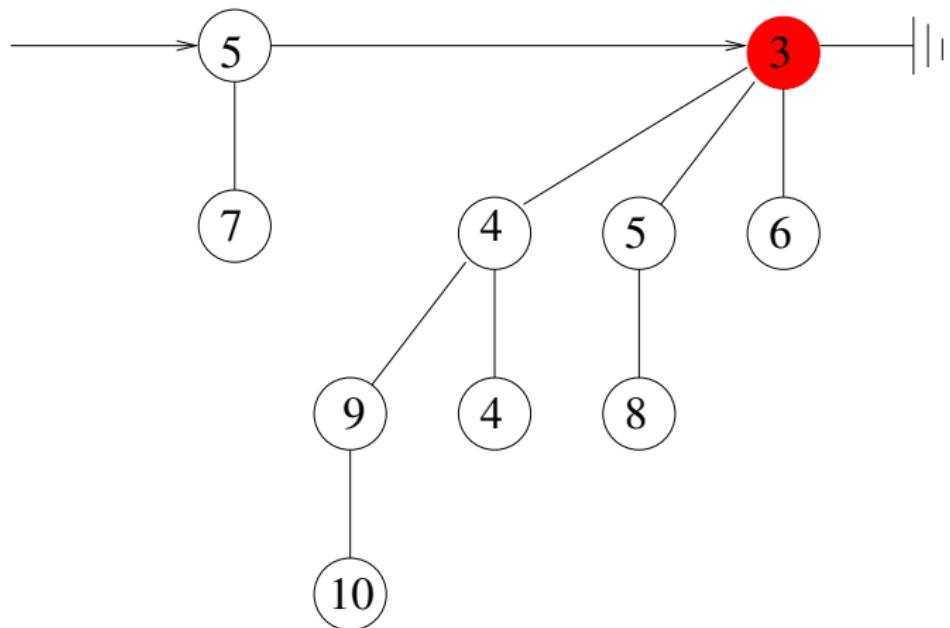
Binomial Queues

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



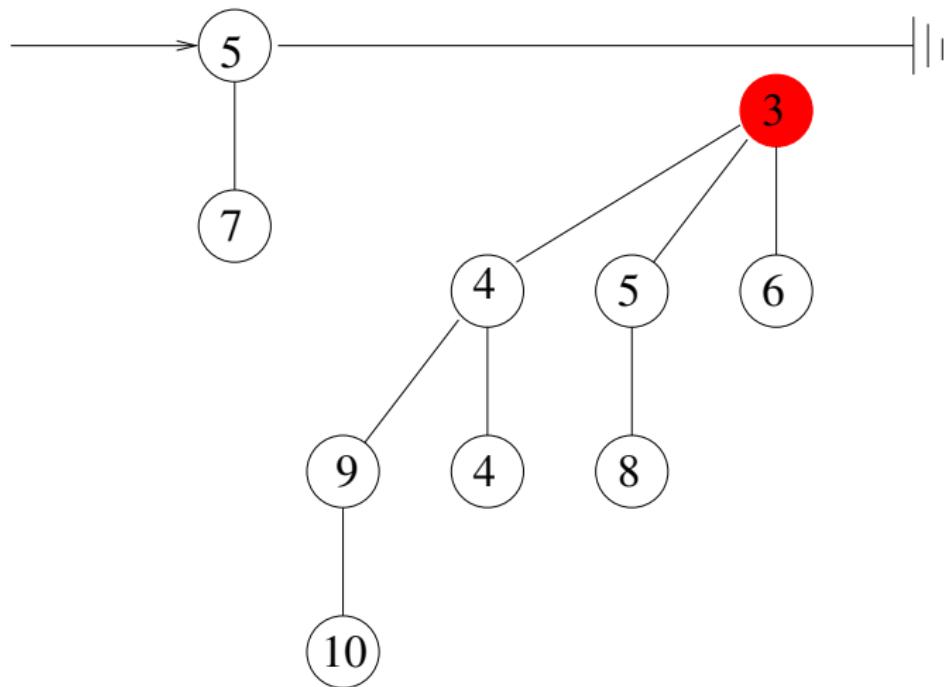
Binomial Queues

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



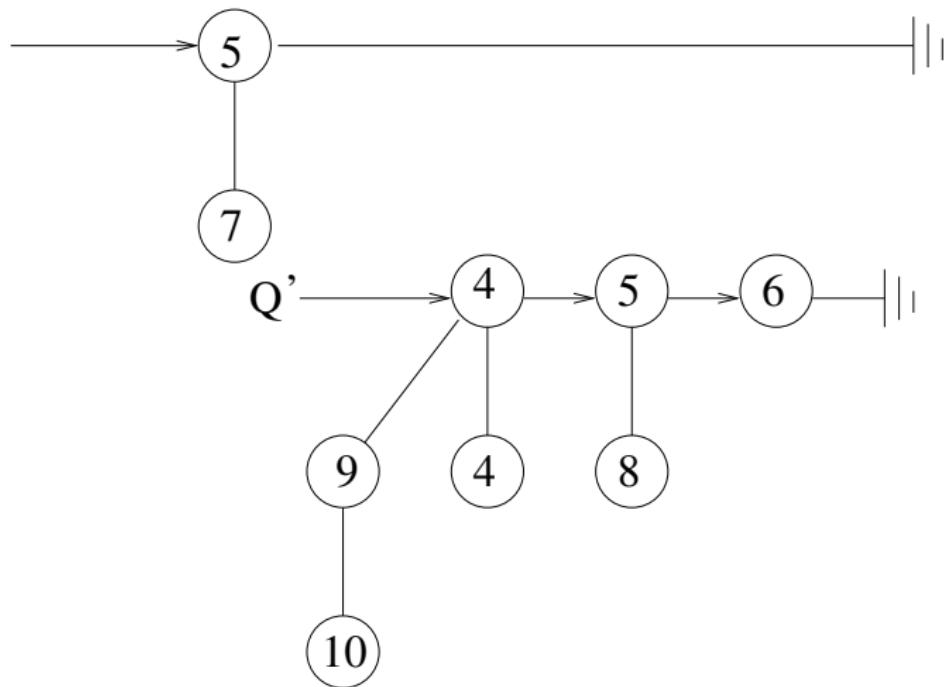
Binomial Queues

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



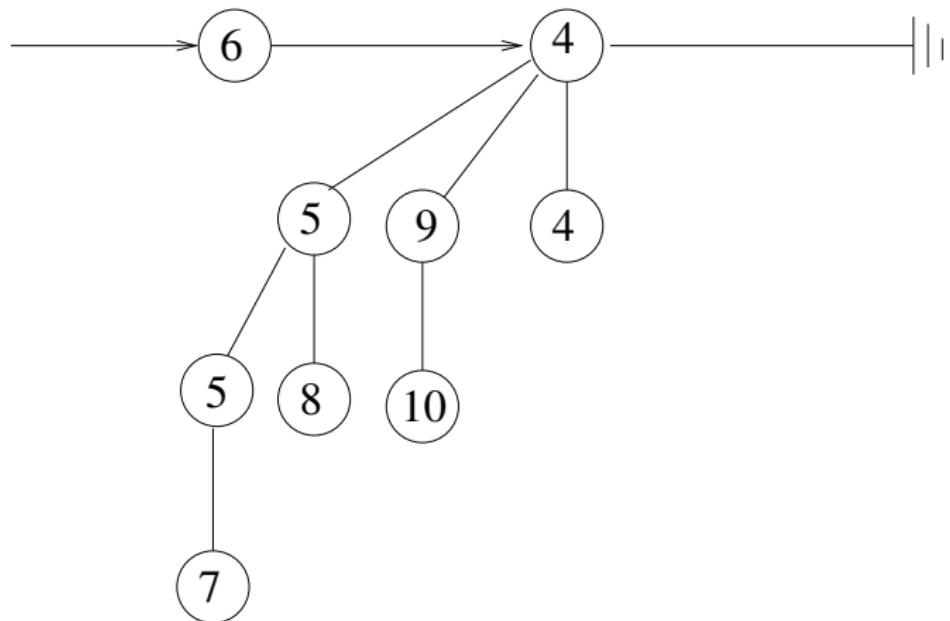
Binomial Queues

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



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

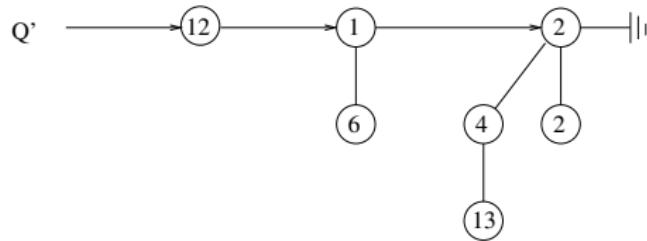
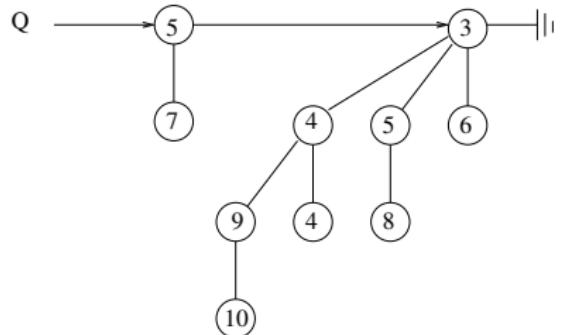
Binomial Queues

- 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 $C = \emptyset$

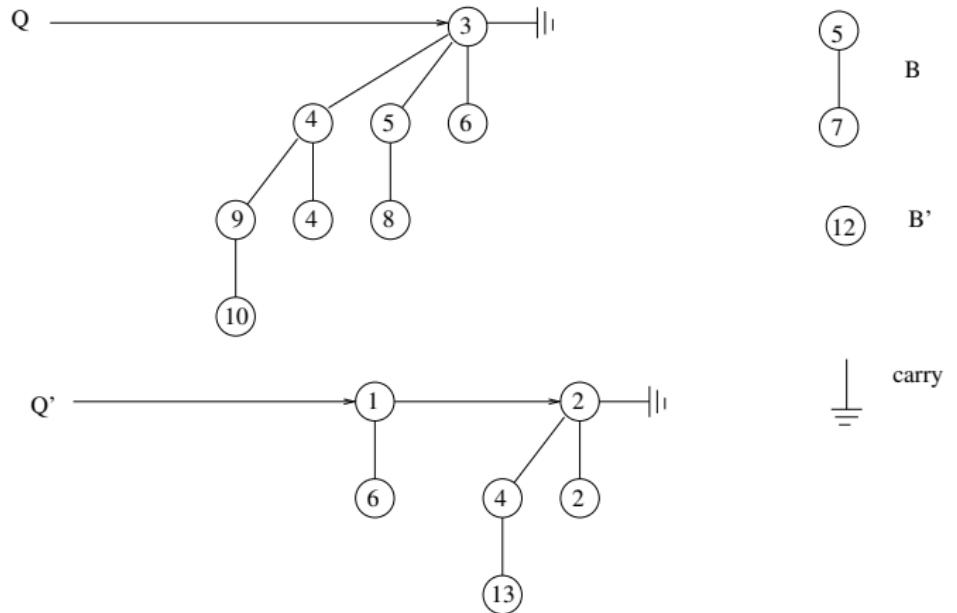
Binomial Queues

- 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

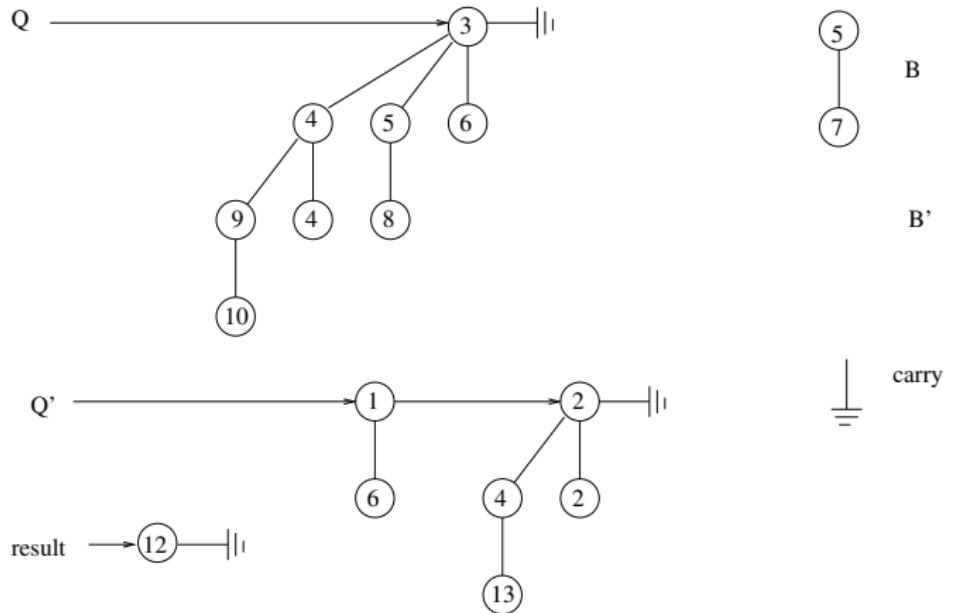
Binomial Queues



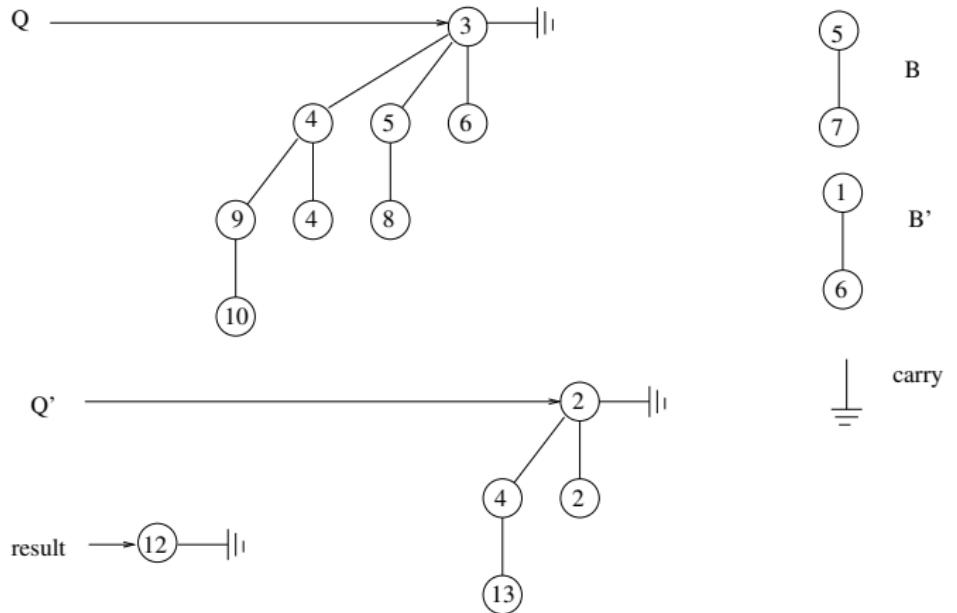
Binomial Queues



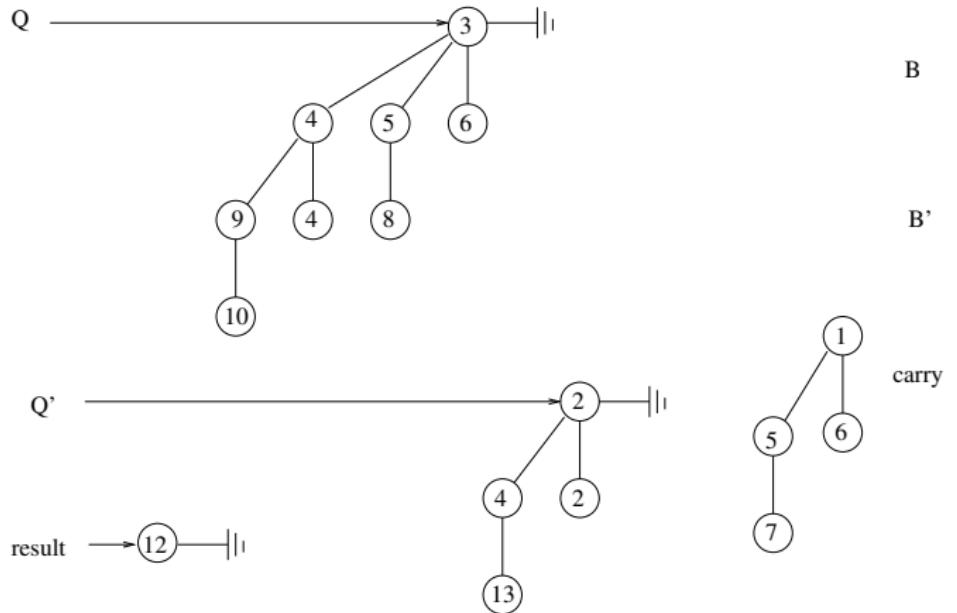
Binomial Queues



Binomial Queues

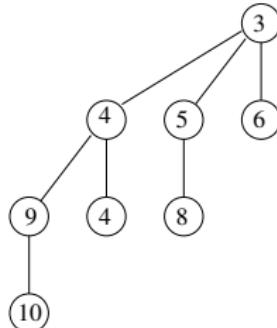


Binomial Queues



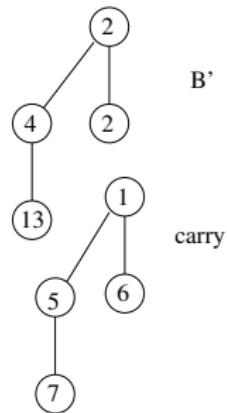
Binomial Queues

Q → ||



Q' → ||

B



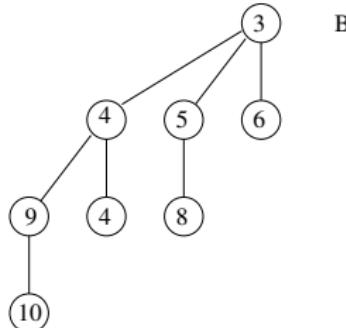
B'

carry

result → 12 → ||

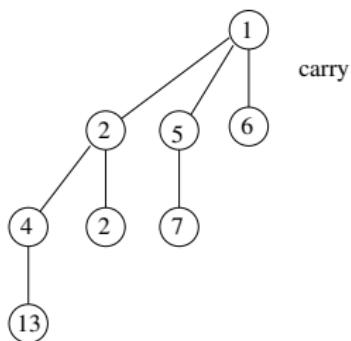
Binomial Queues

Q → ||



B

Q' → ||



B'

result → 12 → ||

carry

Binomial Queues

Q —————||—

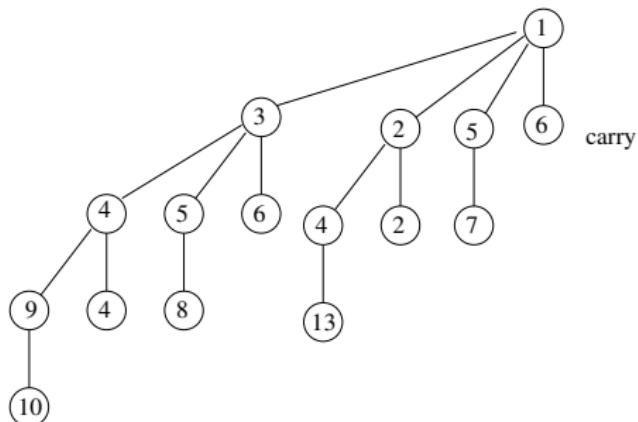
B

B'

Q' —————||—

result → 12 —————||—

carry



Binomial Queues

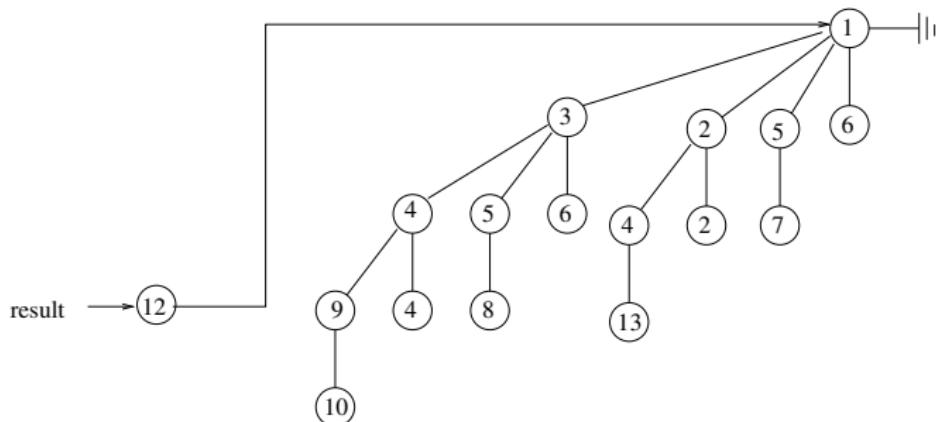
Q —————|||

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, 2nd ed
MIT Press, 2001.

Part III

Priority Queues

- 4 Introduction
- 5 Binary Heaps & Heapsort
 - Heapsort
- 6 Binomial Queues
- 7 Fibonacci Heaps

Introduction



M. Freedman R.E. Tarjan

- *Fibonacci heaps* were introduced in 1986 by M. Freedman and R.E. Tarjan; they guarantee that a sequence of m operations (`insert`, `extract_min` and `decrease_prio`) involving n items (n insertions) has cost $\mathcal{O}(m + n \log n)$
- For $m = \Omega(n \log n)$, this implies constant amortized cost per operation
- For example, applying them for Dijkstra's shortest paths algorithms reduces cost from $\mathcal{O}(m \log n)$ to $\mathcal{O}(m + n \log n)$. It also implies improvements in other graph algorithms

Introduction



M. Freedman R.E. Tarjan

- *Fibonacci heaps* were introduced in 1986 by M. Freedman and R.E. Tarjan; they guarantee that a sequence of m operations (`insert`, `extract_min` and `decrease_prio`) involving n items (n insertions) has cost $\mathcal{O}(m + n \log n)$
- For $m = \Omega(n \log n)$, this implies constant amortized cost per operation
- For example, applying them for Dijkstra's shortest paths algorithms reduces cost from $\mathcal{O}(m \log n)$ to $\mathcal{O}(m + n \log n)$. It also implies improvements in other graph algorithms

Introduction



M. Freedman R.E. Tarjan

- *Fibonacci heaps* were introduced in 1986 by M. Freedman and R.E. Tarjan; they guarantee that a sequence of m operations (`insert`, `extract_min` and `decrease_prio`) involving n items (n insertions) has cost $\mathcal{O}(m + n \log n)$
- For $m = \Omega(n \log n)$, this implies constant amortized cost per operation
- For example, applying them for Dijkstra's shortest paths algorithms reduces cost from $\mathcal{O}(m \log n)$ to $\mathcal{O}(m + n \log n)$. It also implies improvements in other graph algorithms

Introduction

- Fibonacci heaps are like binomial queues, but less rigid
- Binomial queues's invariant requires **at all moments** that the n elements are organized into binomial trees of order $0, 1, \dots, \log_2 n$, no more than one tree of order i , for any i , a binomial tree of order i holding 2^i elements
- In a `decrease_prio` the element sifts up, until the heap order is reestablished in its binomial tree
- Fibonacci allows several binomial trees of the same *rank* until **consolidation** (triggered by `extract_min`)

Introduction

- Fibonacci heaps are like binomial queues, but less rigid
- Binomial queues's invariant requires **at all moments** that the n elements are organized into binomial trees of order $0, 1, \dots, \log_2 n$, no more than one tree of order i , for any i , a binomial tree of order i holding 2^i elements
- In a `decrease_prio` the element sifts up, until the heap order is reestablished in its binomial tree
- Fibonacci allows several binomial trees of the same *rank* until **consolidation** (triggered by `extract_min`)

Introduction

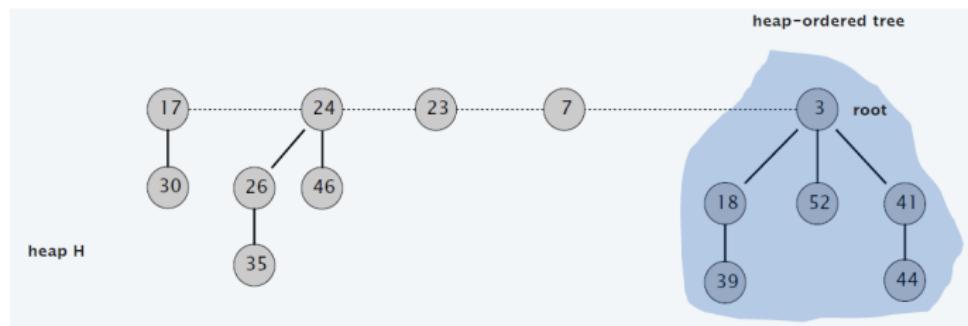
- Fibonacci heaps are like binomial queues, but less rigid
- Binomial queues's invariant requires **at all moments** that the n elements are organized into binomial trees of order $0, 1, \dots, \log_2 n$, no more than one tree of order i , for any i , a binomial tree of order i holding 2^i elements
- In a `decrease_prio` the element sifts up, until the heap order is reestablished in its binomial tree
- Fibonacci allows several binomial trees of the same *rank* until **consolidation** (triggered by `extract_min`)

Introduction

- Fibonacci heaps are like binomial queues, but less rigid
- Binomial queues's invariant requires **at all moments** that the n elements are organized into binomial trees of order $0, 1, \dots, \log_2 n$, no more than one tree of order i , for any i , a binomial tree of order i holding 2^i elements
- In a `decrease_prio` the element sifts up, until the heap order is reestablished in its binomial tree
- Fibonacci allows several binomial trees of the same *rank* until **consolidation** (triggered by `extract_min`)

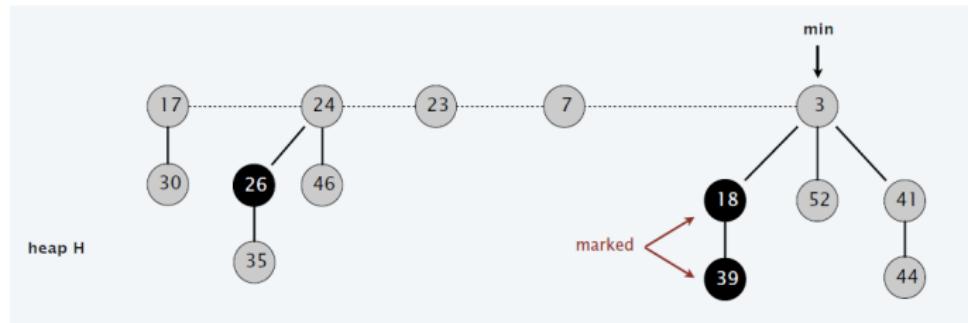
Fibonacci heaps

- A set of **heap ordered** trees: each child's priority smaller than parent's
- A set of **marked** nodes



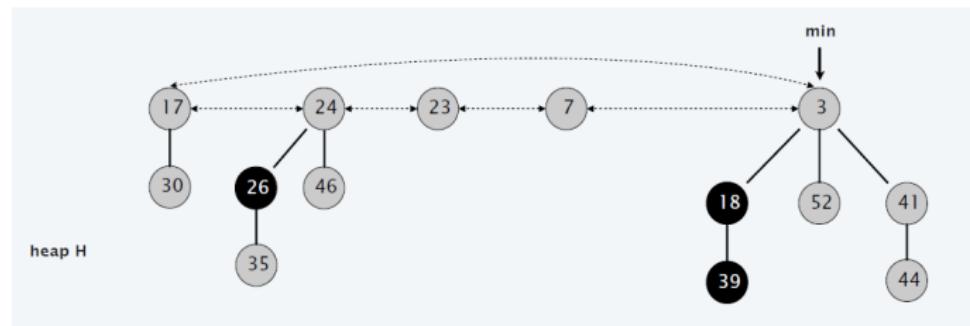
Fibonacci heaps

- A set of **heap ordered** trees: each child's priority smaller than parent's
- A set of **marked** nodes



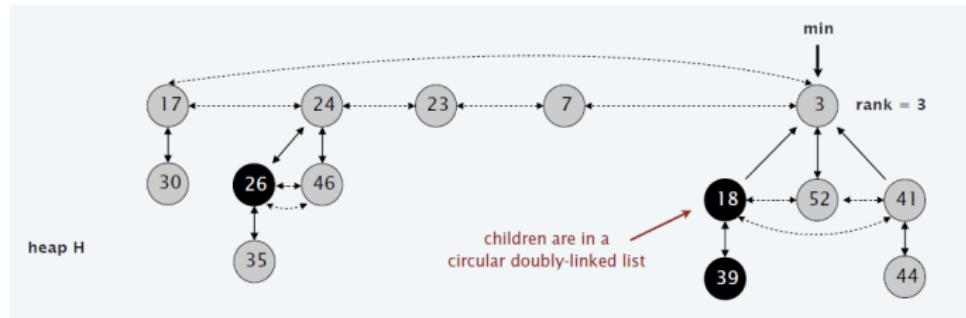
Heap representation

- A pointer `min` to a node with minimum priority
- A **root list**, doubly-linked circular list of tree roots



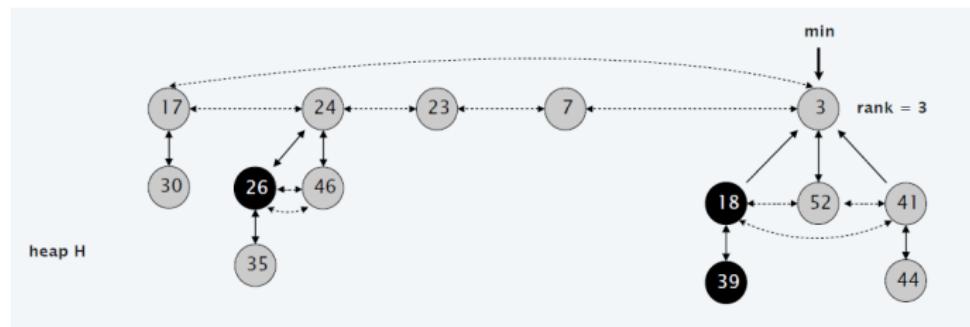
Node representation

- A pointer to its parent
- A pointer to one of its children
- Pointers to left and right sibling (circularly)
- Rank = number of children
- Whether the node is marked



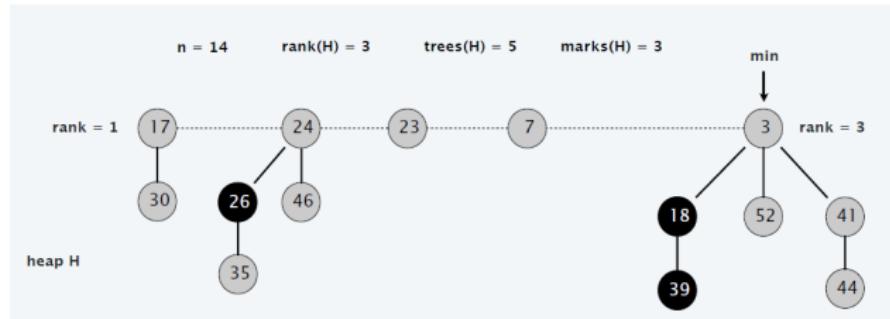
Constant time ops

- Find element of minimum prio
- Merge two root lists
- Find rank of a node
- Add or remove element from a root list
- **Cut** a subtree and merge into root list
- **Link** a subtree (its root) to the root of another



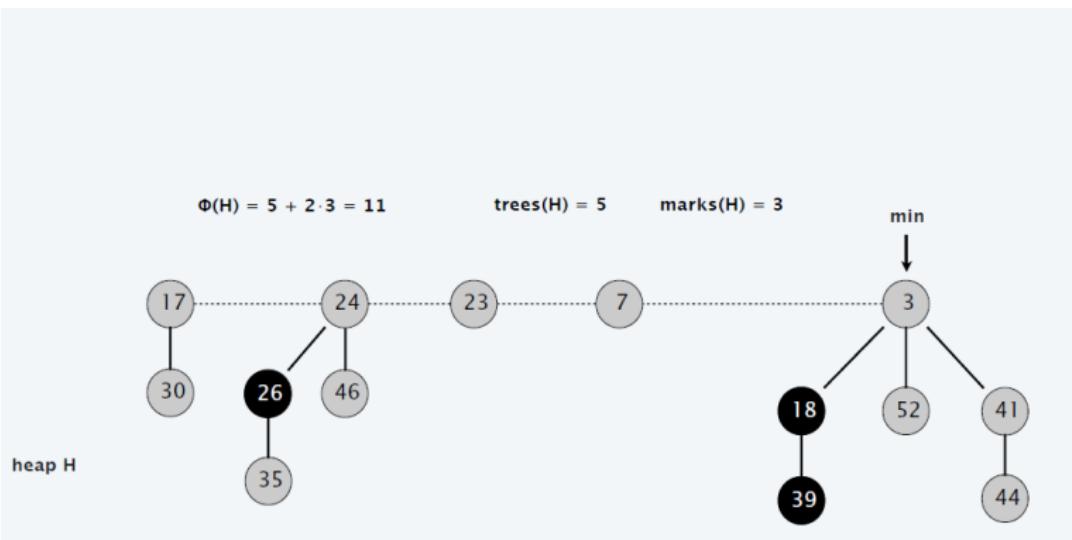
Summary & notations

Notation	Definition
n	number of nodes (<i>size</i>)
$\text{rank}(x)$	number of children of node x
$\text{rank}(H)$	max rank rank of any node in heap H
$\text{trees}(H)$	number of trees in heap H
$\text{marks}(H)$	number of marked nodes in heap H



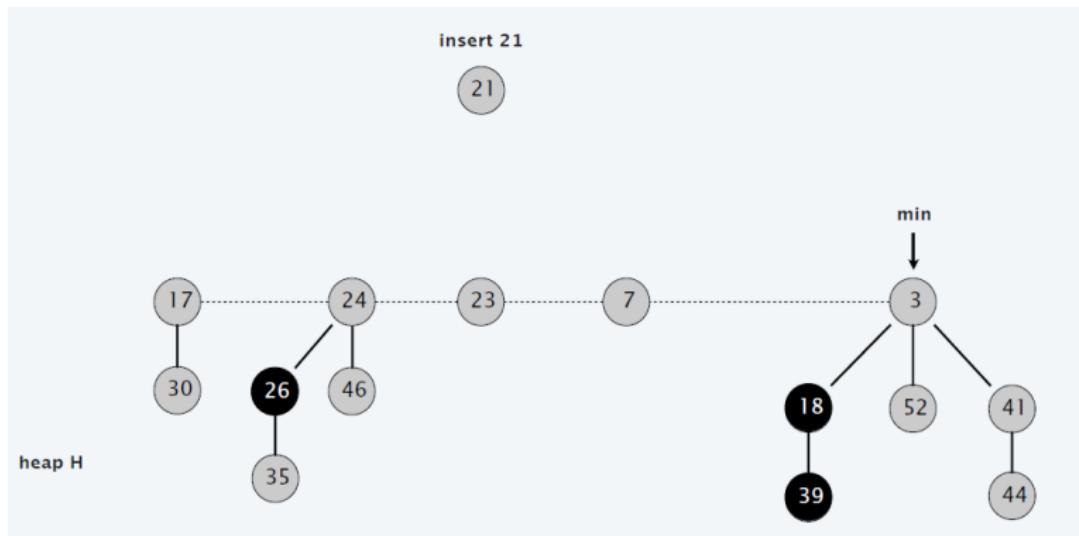
Potential function

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



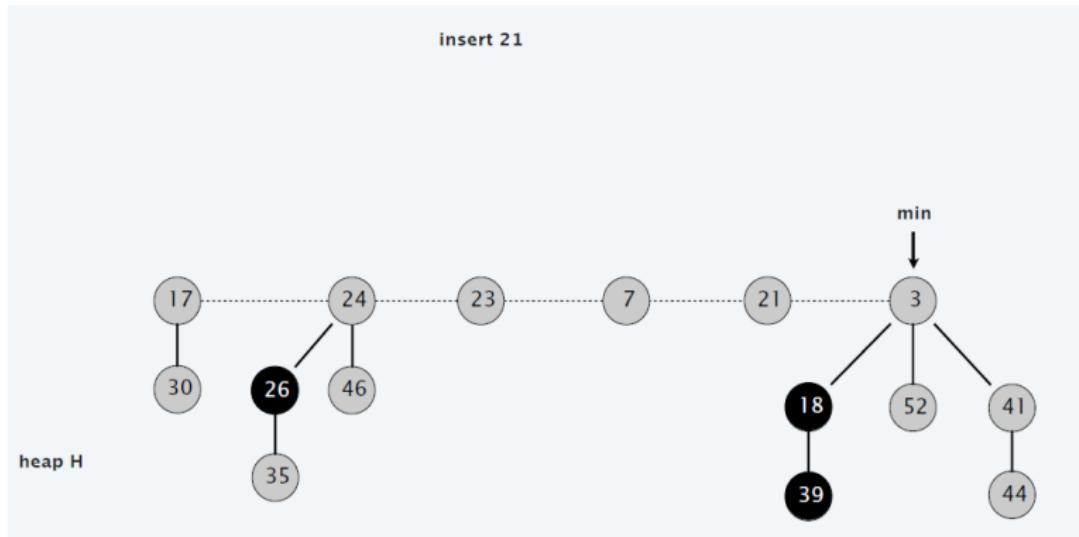
Insertion

- 1 Create a new tree with a single node
- 2 Add it to root list and update `min` if needed



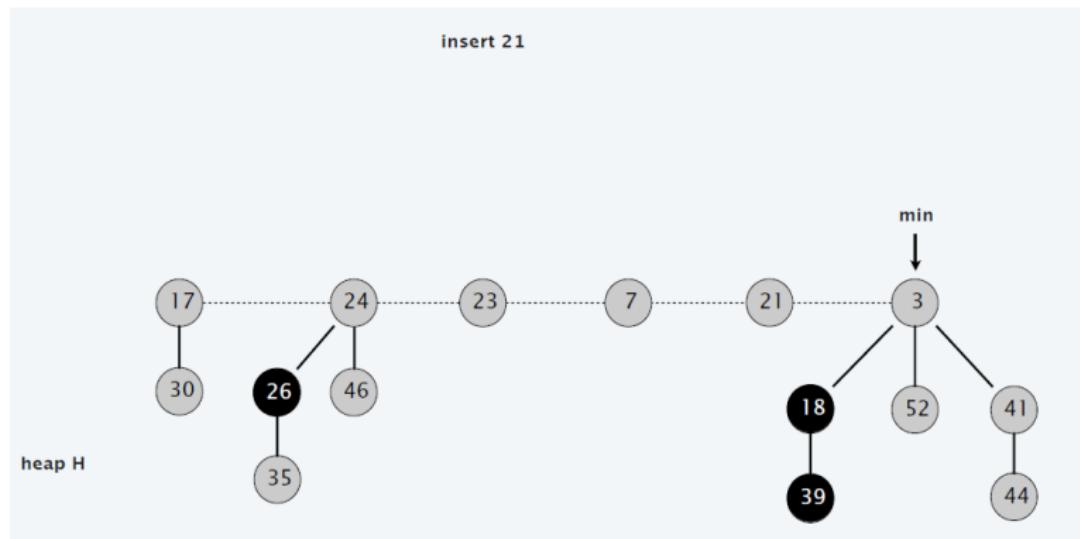
Insertion

- 1 Create a new tree with a single node
- 2 Add it to root list and update `min` if needed



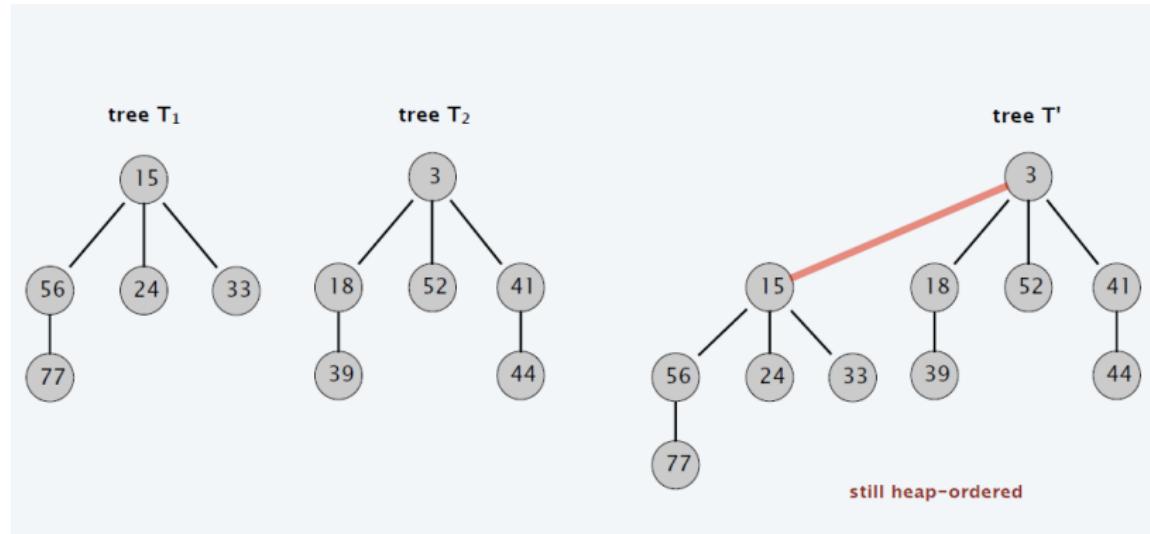
Insertion: Analysis

- ① Actual cost: $c_{\text{INS}} = \mathcal{O}(1)$
- ② Change in potential: $\Delta\Phi = 1$, one more tree, no changes in marks
- ③ Amortized cost: $\hat{c}_{\text{INS}} = c_{\text{INS}} + \Delta\Phi = c_{\text{INS}} + 1 = \mathcal{O}(1)$



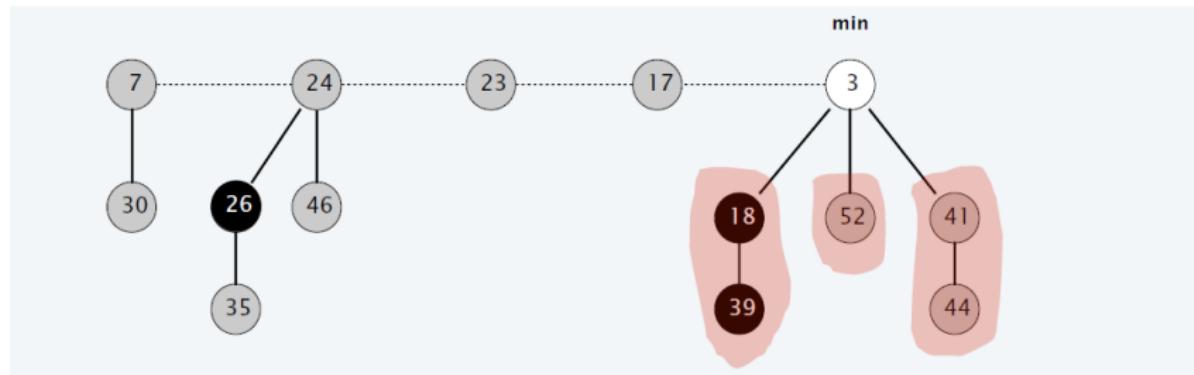
Extract minimum

We will use the following **linking** primitive. Given two trees T_1 and T_2 of rank k , the root with larger priority becomes a child of the root with smaller priority. The resulting tree T' has rank $k + 1$.



Extract minimum

- ① Delete \min , meld its children with the root list and update \min
- ② **Consolidate** the root lists so that no two trees have the same rank



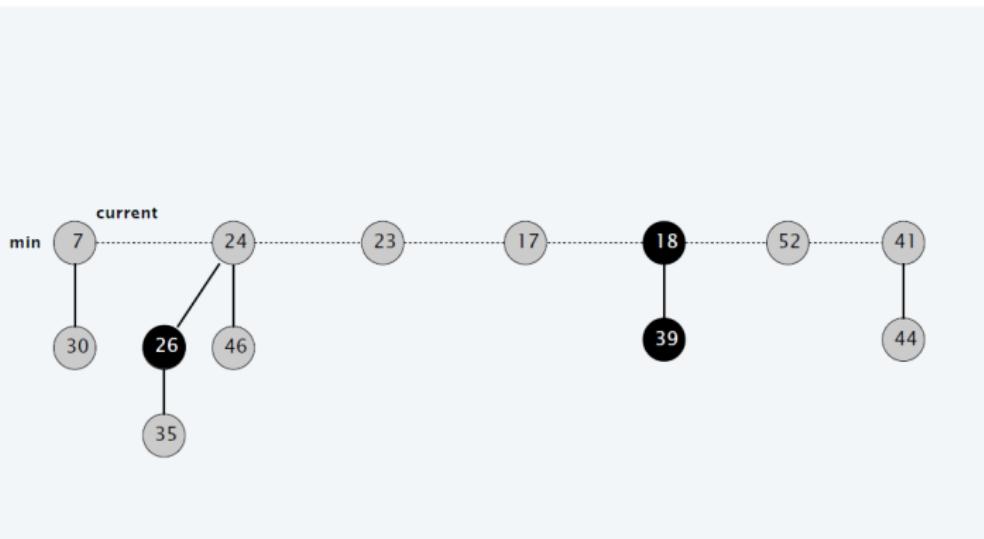
Extract minimum

- ① Delete \min , meld its children with the root list and update \min
- ② **Consolidate** the root lists so that no two trees have the same rank



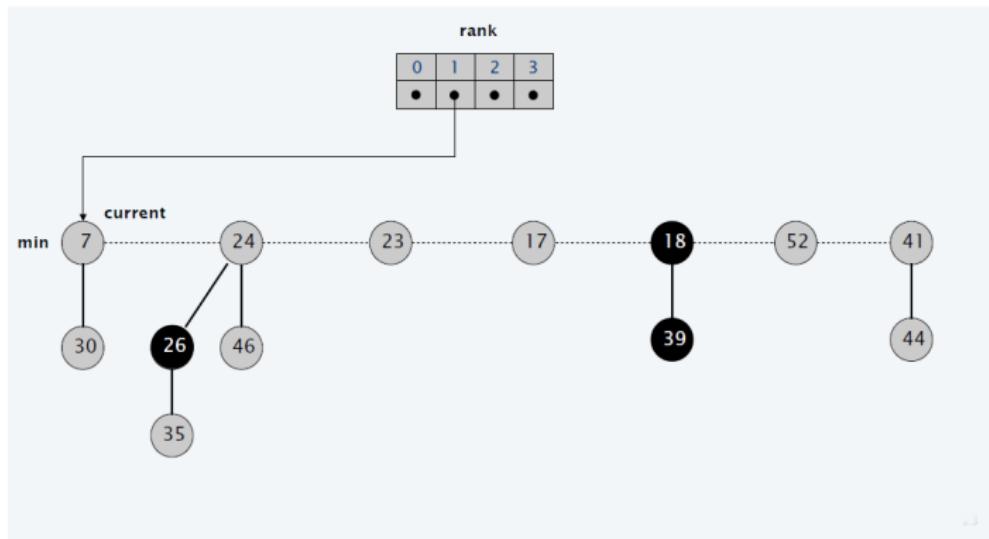
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



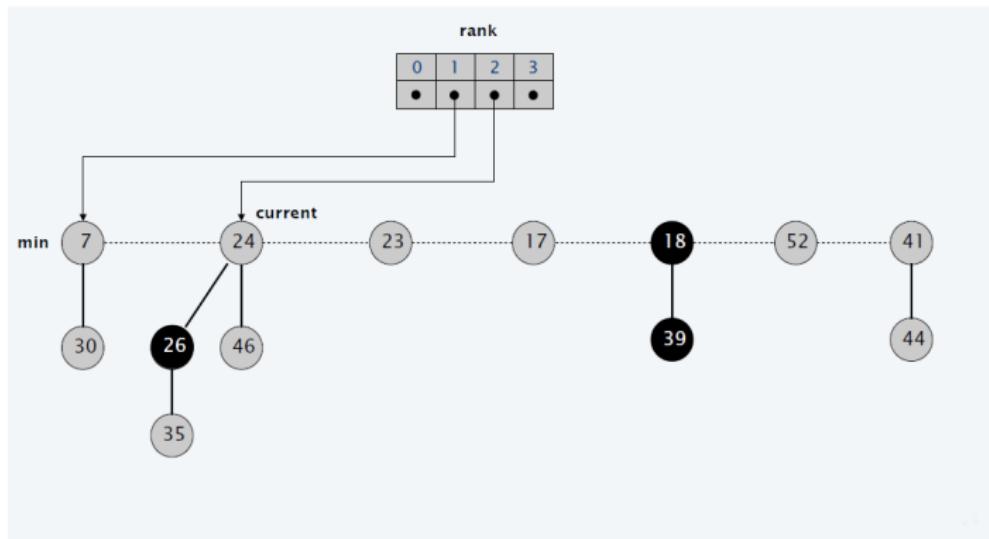
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



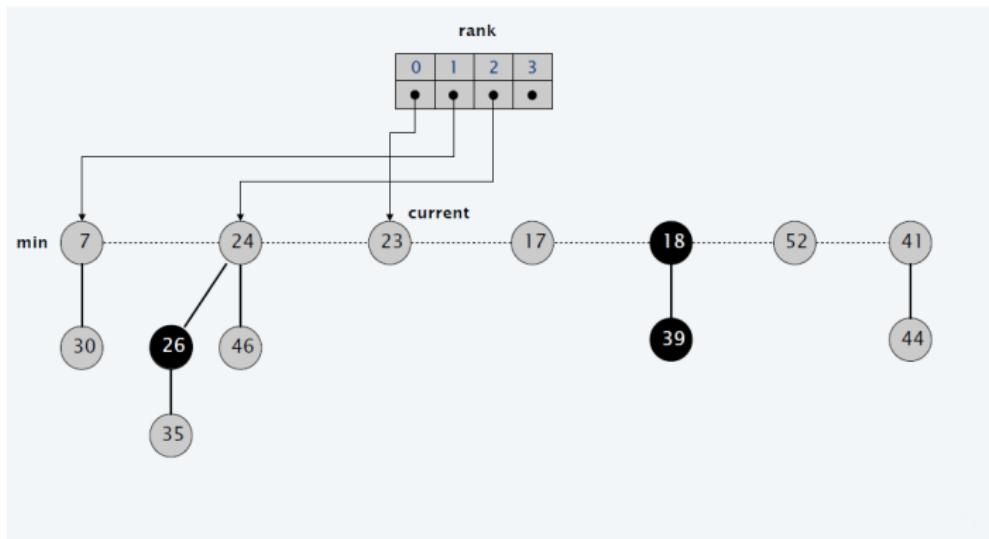
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



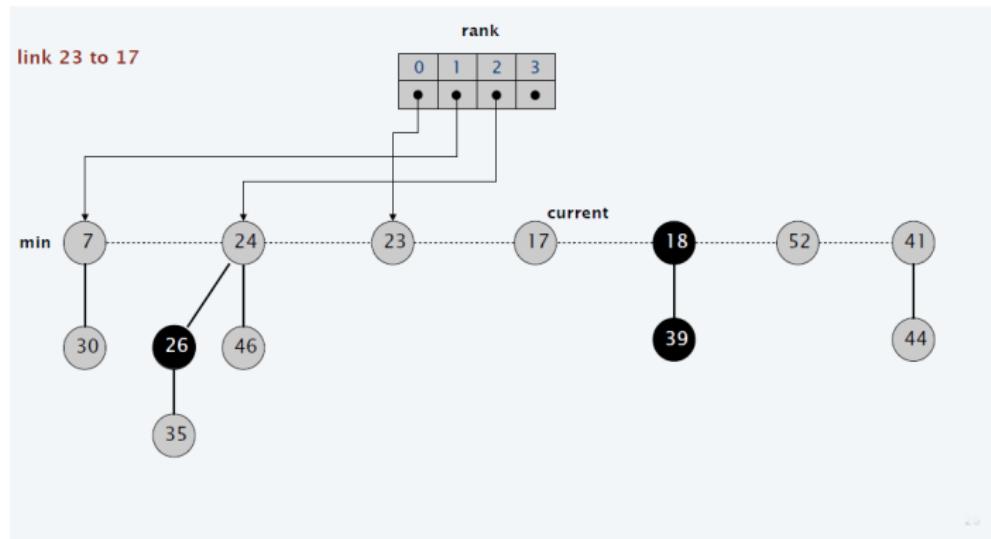
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



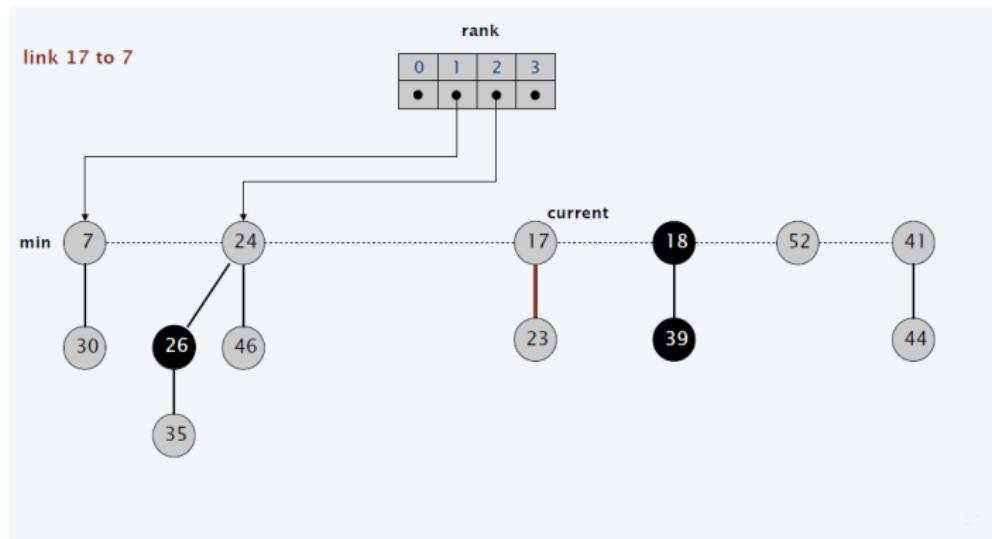
Consolidate

- 1 Delete \min , meld its children with the root list and update \min
- 2 **Consolidate** the root lists so that no two trees have the same rank



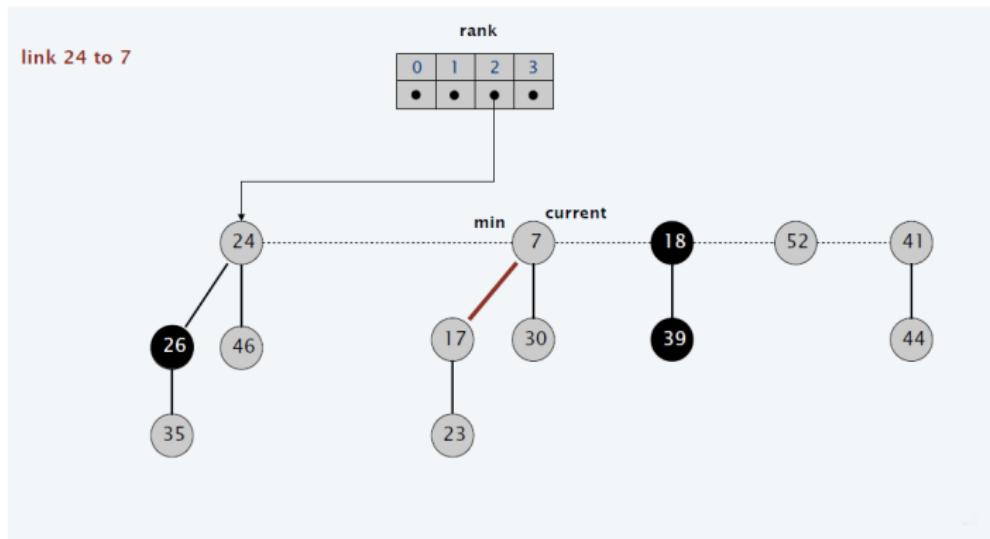
Consolidate

- 1 Delete \min , meld its children with the root list and update \min
- 2 **Consolidate** the root lists so that no two trees have the same rank



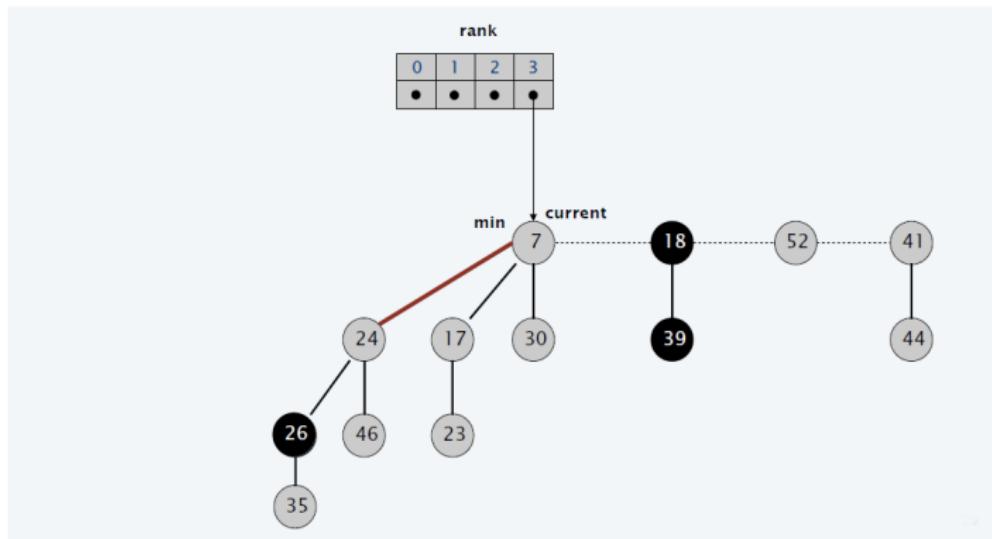
Consolidate

- ① Delete min , meld its children with the root list and update min
- ② **Consolidate** the root lists so that no two trees have the same rank



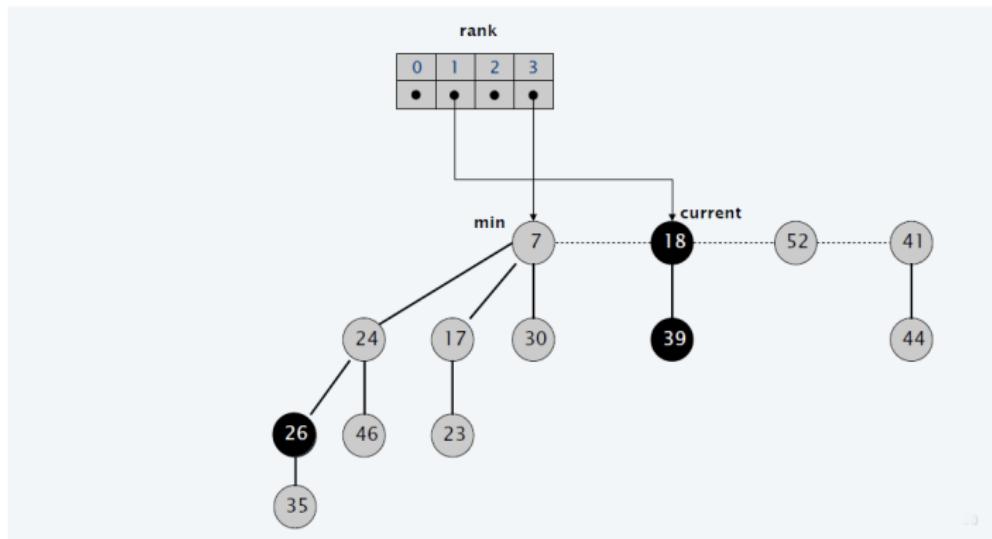
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



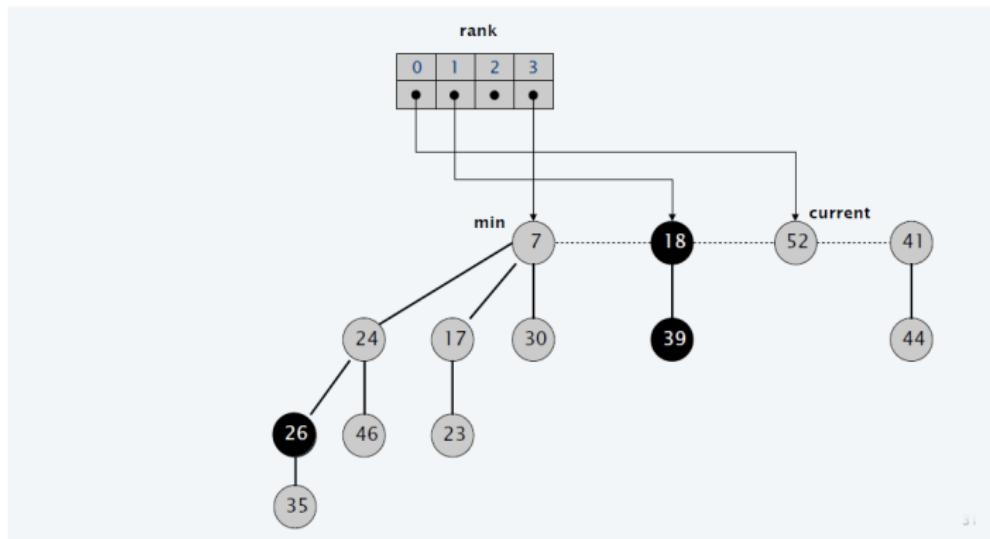
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



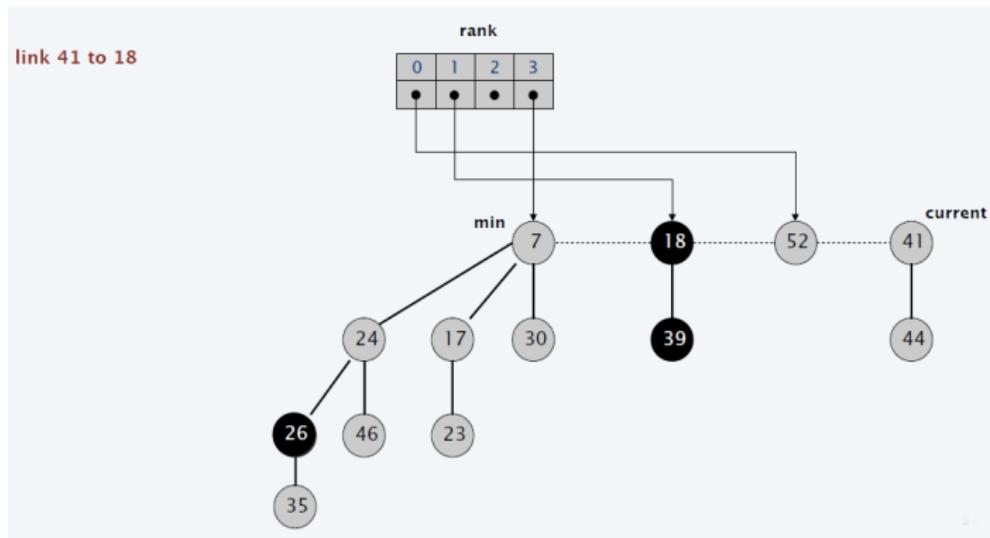
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



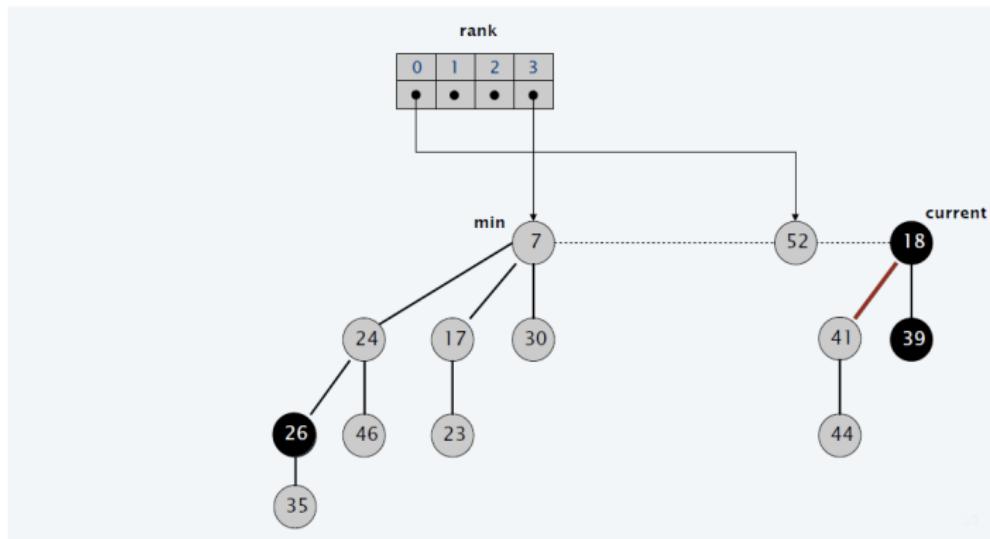
Consolidate

- 1 Delete \min , meld its children with the root list and update \min
- 2 **Consolidate** the root lists so that no two trees have the same rank



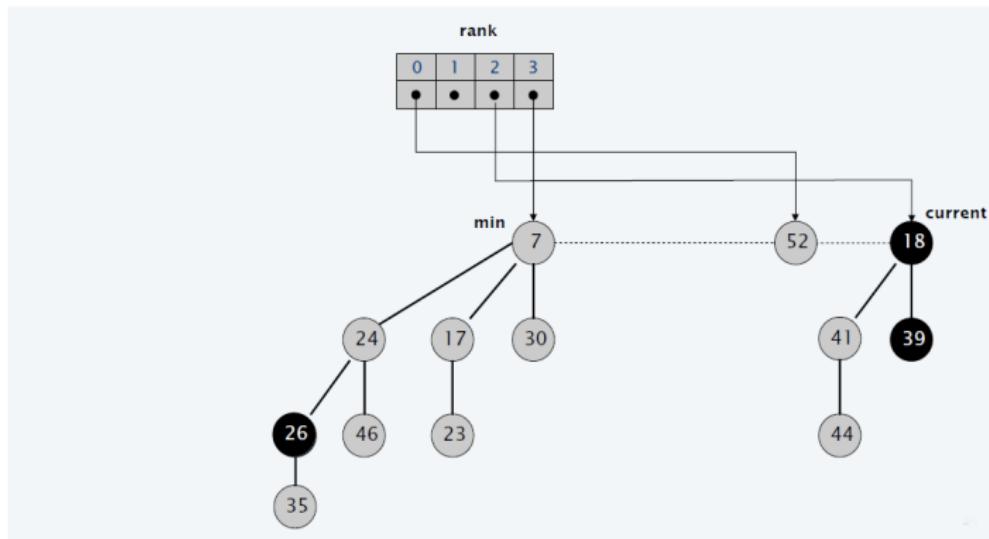
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



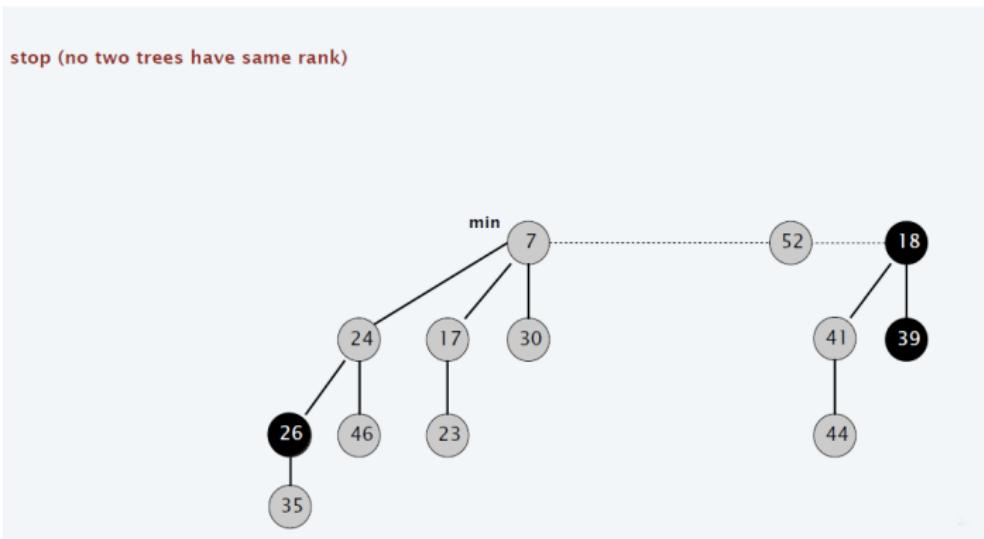
Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



Consolidate

- 1 Delete min , meld its children with the root list and update min
- 2 **Consolidate** the root lists so that no two trees have the same rank



Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld `min`'s children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update `min`
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
- ➌ Amortized cost: $\mathcal{O}(\log n)$

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
- ➌ Amortized cost: $\mathcal{O}(\log n)$

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$

- ➌ Amortized cost: $\mathcal{O}(\log n)$

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - New marks created by moving children of \min from one tree to another
 - $\Delta\Phi \leq \text{rank}(H) + 1 - \text{trees}(H)$
- ➌ Amortized cost: $\mathcal{O}(\log n)$

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - $\text{trees}(H') \leq \text{rank}(H') + 1 \leftarrow$ no two trees have the same rank after consolidate
- ➌ Amortized cost: $\mathcal{O}(\log n)$

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - $\text{trees}(H') \leq \text{rank}(H') + 1 \leftarrow$ no two trees have the same rank after consolidate
- ➌ Amortized cost: $\mathcal{O}(\log n)$

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - $\text{trees}(H') \leq \text{rank}(H') + 1 \leftarrow \text{no two trees have the same rank after consolidate}$
- ➌ Amortized cost: $\mathcal{O}(\log n)$
 - $\hat{c}_{EM} = c_{EM} + \Delta\Phi = \mathcal{O}(\text{rank}(H')) + \mathcal{O}(\text{rank}(H))$
 - The rank of any element is at most $\mathcal{O}(\log n)$ (from the previous slide)

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - $\text{trees}(H') \leq \text{rank}(H') + 1 \leftarrow \text{no two trees have the same rank after consolidate}$
- ➌ Amortized cost: $\mathcal{O}(\log n)$
 - $\hat{c}_{EM} = c_{EM} + \Delta\Phi = \mathcal{O}(\text{rank}(H')) + \mathcal{O}(\text{rank}(H))$
 - The rank of any Fibonacci heap is $\mathcal{O}(\log n)$ (Fibonacci Lemma)

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - $\text{trees}(H') \leq \text{rank}(H') + 1 \leftarrow \text{no two trees have the same rank after consolidate}$
- ➌ Amortized cost: $\mathcal{O}(\log n)$
 - $\hat{c}_{EM} = c_{EM} + \Delta\Phi = \mathcal{O}(\text{rank}(H')) + \mathcal{O}(\text{rank}(H))$
 - The rank of any Fibonacci heap is $\mathcal{O}(\log n)$ (Fibonacci Lemma)

Extract minimum: Analysis

- ➊ Actual cost: $c_{EM} = \mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$
 - $\mathcal{O}(\text{rank}(H))$ to meld \min 's children in root list
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to update \min
 - $\mathcal{O}(\text{rank}(H)) + \mathcal{O}(\text{trees}(H))$ to consolidate
- ➋ Change in potential: $\Delta\Phi \leq \text{rank}(H') + 1 - \text{trees}(H)$
 - No new marks created
 - $\text{trees}(H') \leq \text{rank}(H') + 1 \leftarrow \text{no two trees have the same rank after consolidate}$
- ➌ Amortized cost: $\mathcal{O}(\log n)$
 - $\hat{c}_{EM} = c_{EM} + \Delta\Phi = \mathcal{O}(\text{rank}(H')) + \mathcal{O}(\text{rank}(H))$
 - The rank of any Fibonacci heap is $\mathcal{O}(\log n)$ (**Fibonacci Lemma**)

Fibonacci heaps vs. binomial heaps

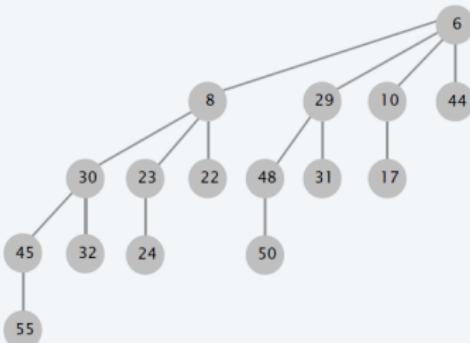
- If only `insert` and `extract_min` operations are performed then all trees in a FH are binomial trees (because we only link trees of equal rank \equiv order).
- Binomial heap property implies $\text{rank}(H) \leq \log_2 n$.
- Fibonacci heap property does not guarantee that the trees are binomial (because the way we implement `decr_prio`); we will guarantee however that $\text{rank}(H) \leq \log_\phi n$, with $\phi = (1 + \sqrt{5})/2 \approx 1.618 \dots$

Decrease priority

First approximation for decreasing the priority of a node x

- ① If heap-order is not violated, update the priority of x
- ② Otherwise, cut tree rooted at x and meld into root list

decrease-key of x from 30 to 7

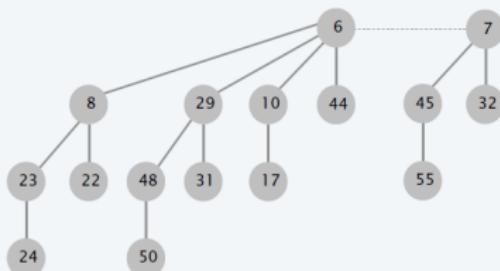


Decrease priority

First approximation for decreasing the priority of a node x

- 1 If heap-order is not violated, update the priority of x
- 2 Otherwise, cut tree rooted at x and meld into root list

decrease-key of x from 23 to 5

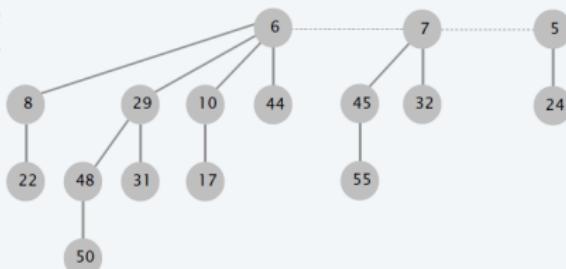


Decrease priority

First approximation for decreasing the priority of a node x

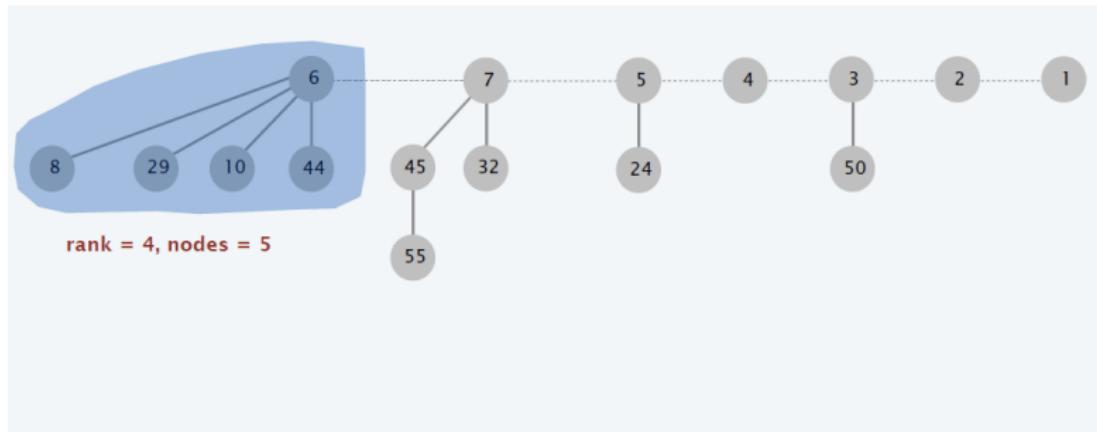
- 1 If heap-order is not violated, update the priority of x
- 2 Otherwise, cut tree rooted at x and meld into root list

```
decrease-key of 22 to 4
decrease-key of 48 to 3
decrease-key of 31 to 2
decrease-key of 17 to 1
```



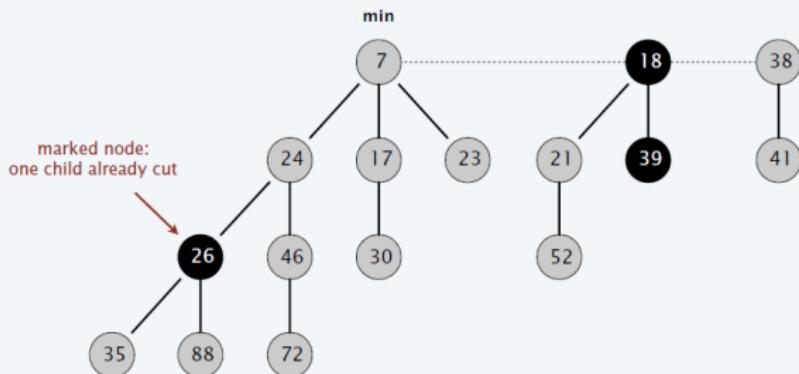
Decrease priority

A problem: number of nodes not exponential w.r.t. rank



Decrease priority

Solution: when a node gets two children cut, cut it too, and meld into the root list; apply iteratively until no more cuts are needed. We will use **marks** to keep track when a node has lost a child. When a node is cut because of losing 2nd child, it gets unmarked

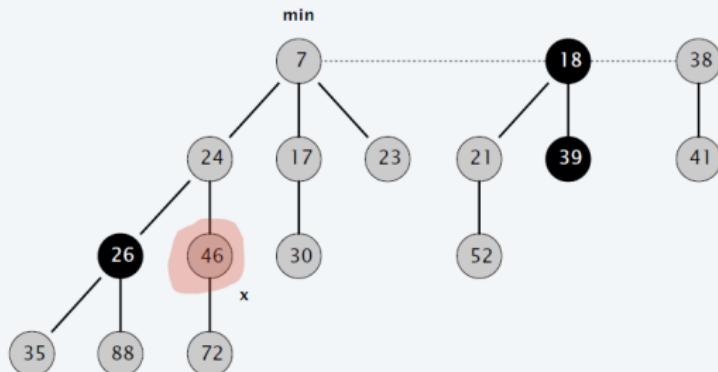


Decrease priority

Case 1 Heap-order is not violated

- Decrease the priority of x
- Update \min if necessary

decrease-key of x from 46 to 29

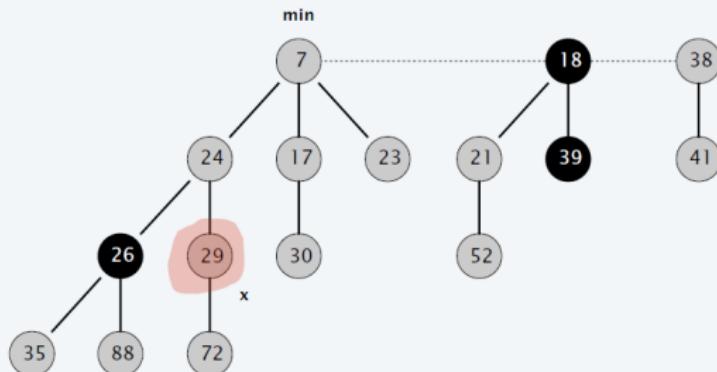


Decrease priority

Case 1 Heap-order is not violated

- Decrease the priority of x
- Update \min if necessary

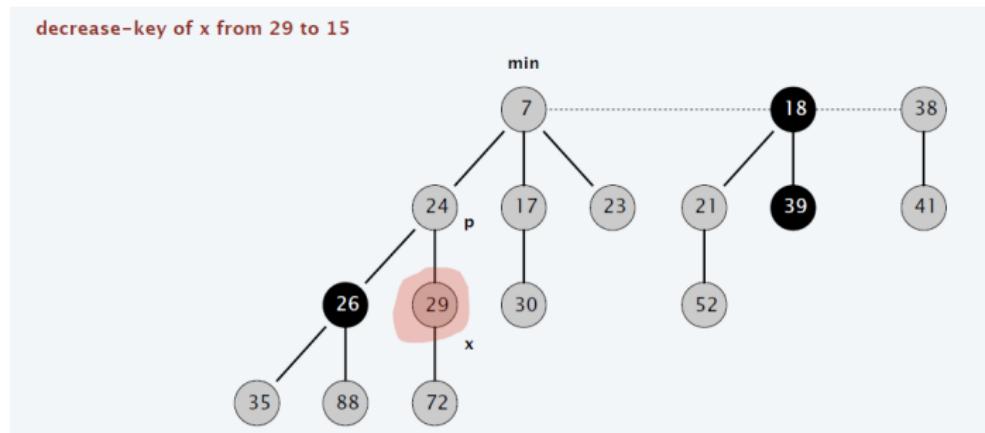
decrease-key of x from 46 to 29



Decrease priority

Case 2a Heap-order violated

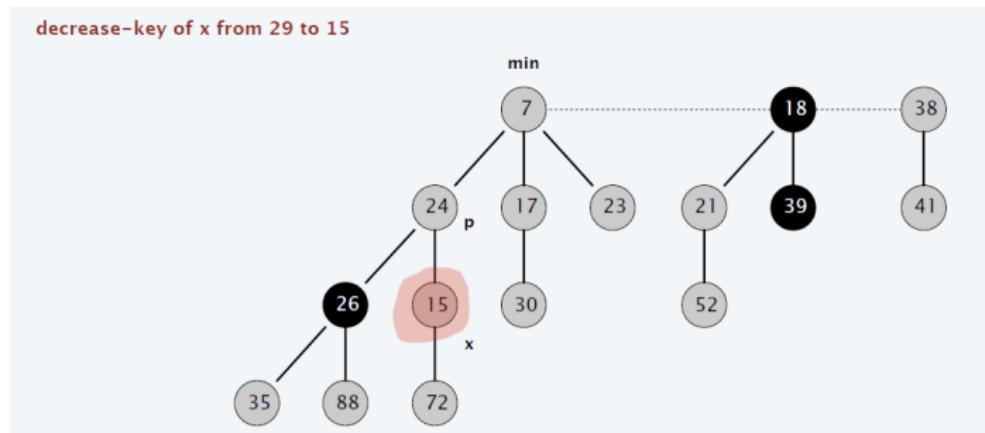
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is unmarked, mark it



Decrease priority

Case 2a Heap-order violated

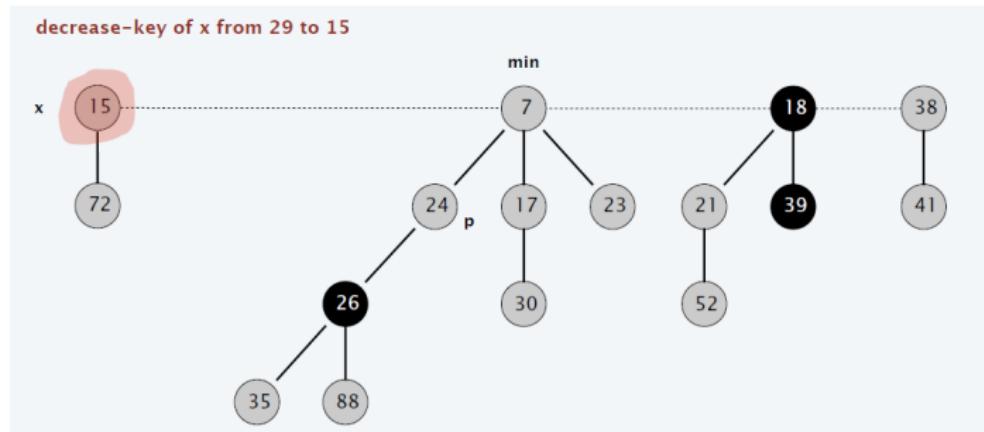
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is unmarked, mark it



Decrease priority

Case 2a Heap-order violated

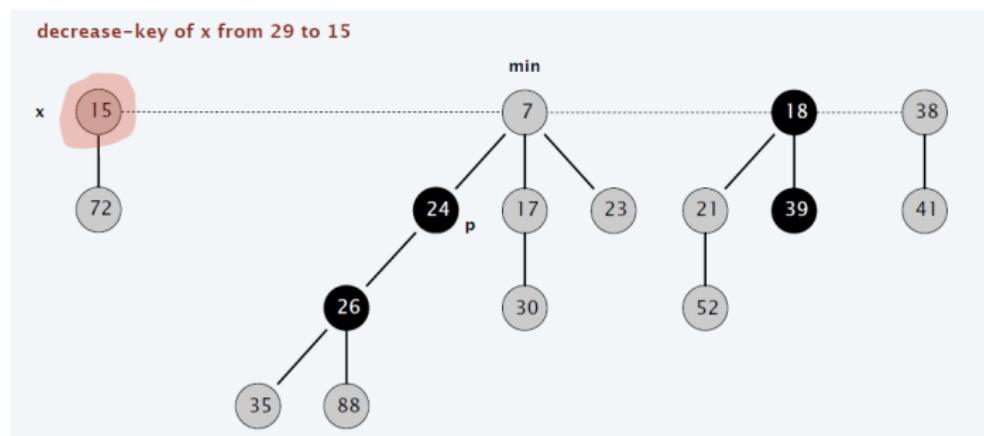
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is unmarked, mark it



Decrease priority

Case 2a Heap-order violated

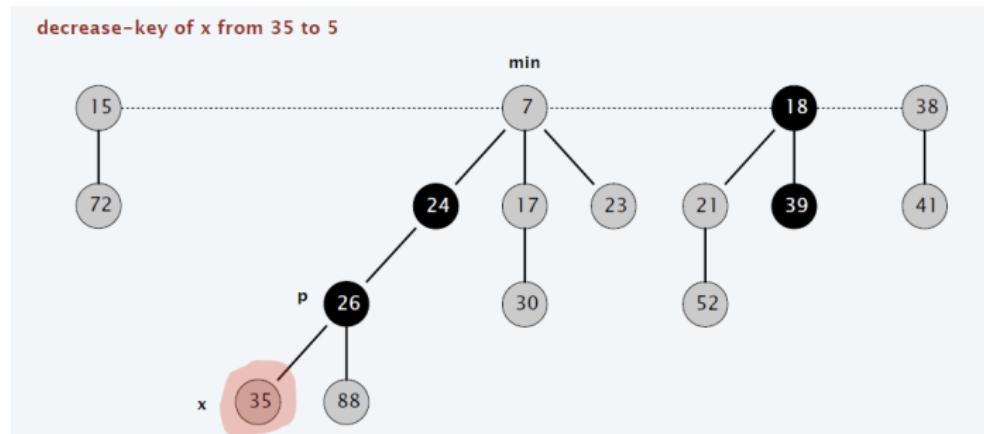
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is unmarked, mark it



Decrease priority

Case 2b Heap-order violated

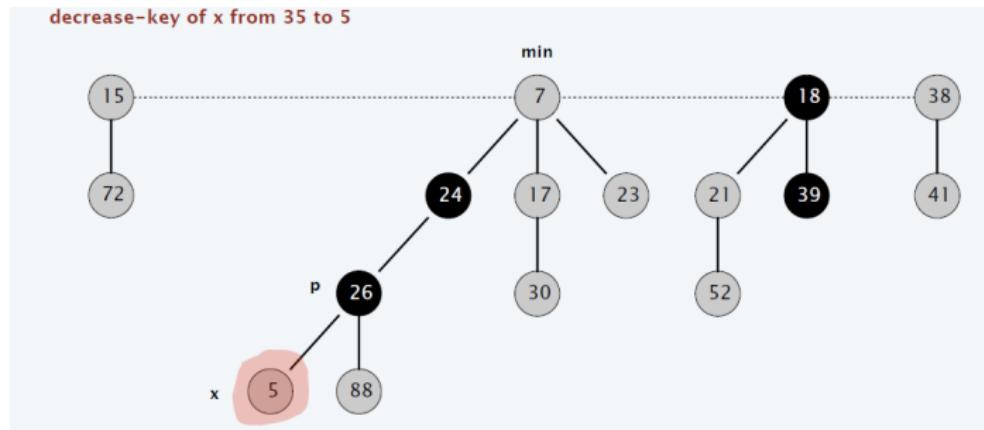
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root



Decrease priority

Case 2b Heap-order violated

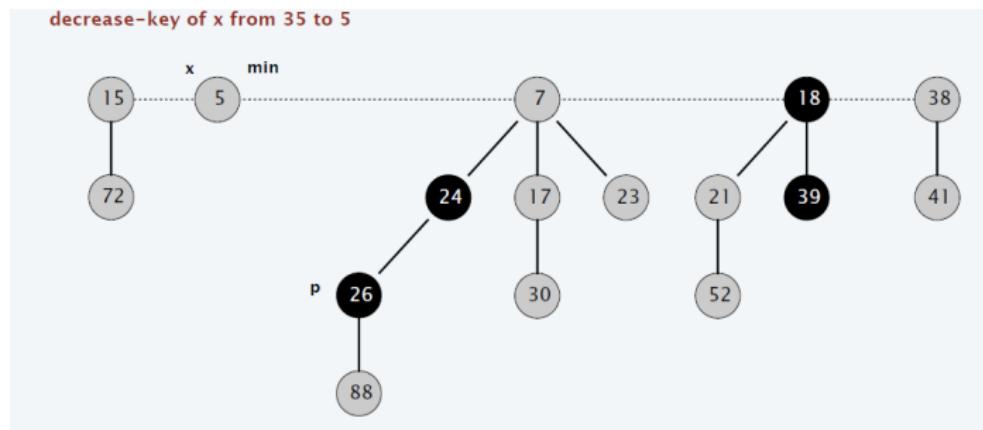
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root



Decrease priority

Case 2b Heap-order violated

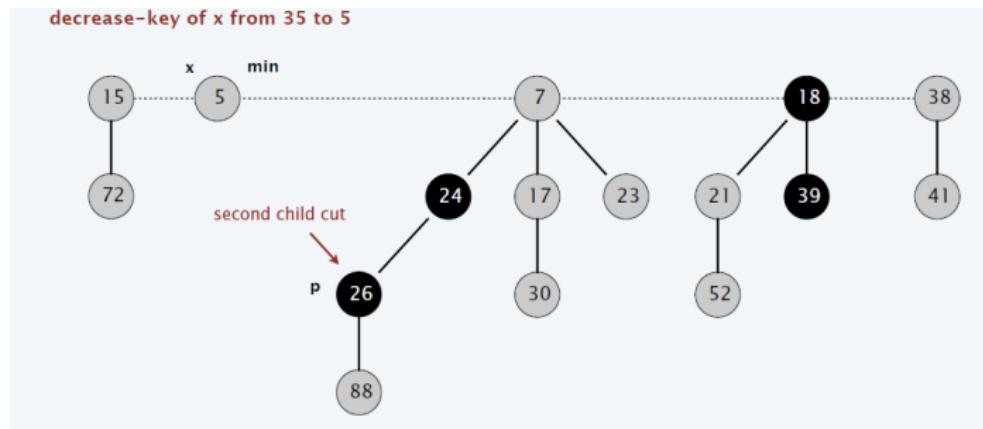
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root



Decrease priority

Case 2b Heap-order violated

- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root

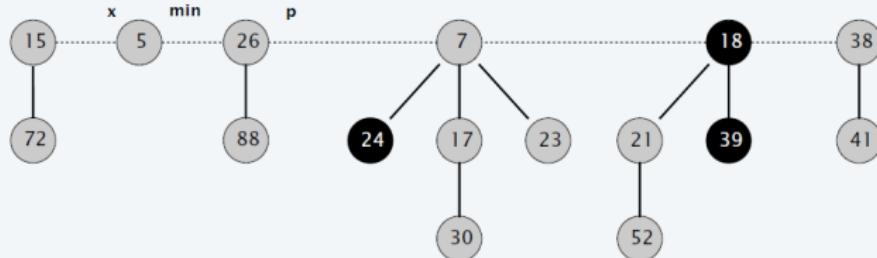


Decrease priority

Case 2b Heap-order violated

- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root

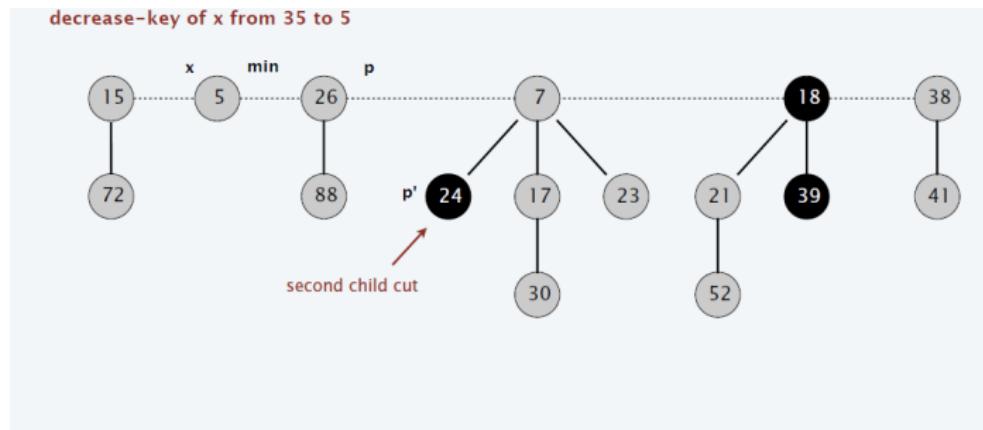
decrease-key of x from 35 to 5



Decrease priority

Case 2b Heap-order violated

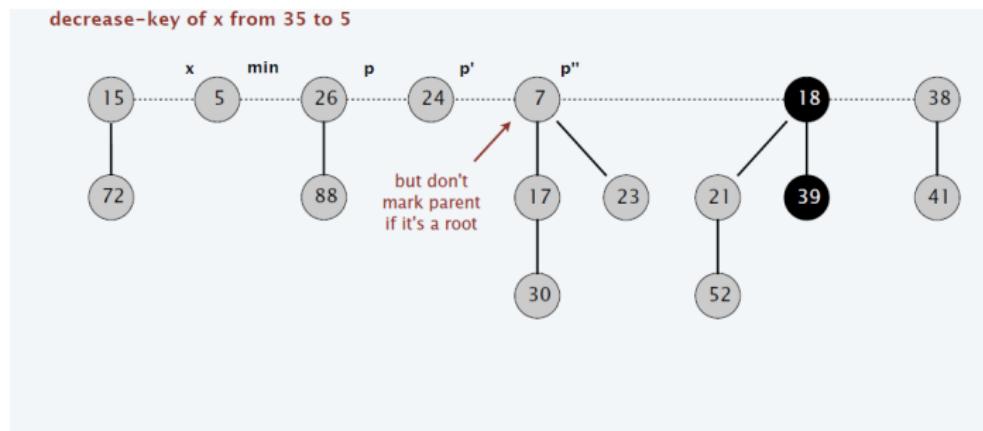
- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root



Decrease priority

Case 2b Heap-order violated

- Decrease the priority of x
- Cut tree rooted at x , remove mark (if it had a mark) and meld into root list
- If parent p of x is marked apply previous step recursively until the ancestor is not marked or the root



Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$

- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
 - $\text{trees}(H') = \text{trees}(H) + c$
 - $\Phi(H') = \Phi(H) + c - \mathcal{O}(1)$
 - Decreasing the priority of a node by 1 increases its weight by 1, so the total weight of the tree decreases by c .
 - The potential Φ is defined as the sum of the weights of the nodes.
- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
 - $\text{trees}(H') = \text{trees}(H) + c$
 - $\text{marks}(H') \leq \text{marks}(H) - c + 2 \leftarrow$ each cut except the first removes a mark, the last cut might or might not mark a node
 - $\Delta\Phi \leq c + 2(2 - c) = 4 - c$
- Amortized cost: $\bar{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
 - $\text{trees}(H') = \text{trees}(H) + c$
 - $\text{marks}(H') \leq \text{marks}(H) - c + 2 \leftarrow$ each cut except the first removes a mark, the last cut might or might not mark a node
 - $\Delta\Phi \leq c + 2(2 - c) = 4 - c$
- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
 - $\text{trees}(H') = \text{trees}(H) + c$
 - $\text{marks}(H') \leq \text{marks}(H) - c + 2 \leftarrow \text{each cut except the first removes a mark, the last cut might or might not mark a node}$
 - $\Delta\Phi \leq c + 2(2 - c) = 4 - c$
- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
 - $\text{trees}(H') = \text{trees}(H) + c$
 - $\text{marks}(H') \leq \text{marks}(H) - c + 2 \leftarrow \text{each cut except the first removes a mark, the last cut might or might not mark a node}$
 - $\Delta\Phi \leq c + 2(2 - c) = 4 - c$
- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Decrease priority: Analysis

- Actual cost: $c_{\text{DECR}} = \mathcal{O}(c)$, where c is the number of cuts
 - $\mathcal{O}(1)$ for decreasing prio + update \min
 - $\mathcal{O}(1)$ for each cut, including melding the tree in the root list
- Change in potential: $\Delta\Phi \leq 1 - c$
 - $\text{trees}(H') = \text{trees}(H) + c$
 - $\text{marks}(H') \leq \text{marks}(H) - c + 2 \leftarrow \text{each cut except the first removes a mark, the last cut might or might not mark a node}$
 - $\Delta\Phi \leq c + 2(2 - c) = 4 - c$
- Amortized cost: $\hat{c}_{\text{DECR}} = c_{\text{DECR}} + \Delta\Phi = \mathcal{O}(1)$

Summary of the analysis

- ➊ Insert: $\mathcal{O}(1)$
- ➋ Extract minimum: $\mathcal{O}(\text{rank}(H))$ amortized
- ➌ Decrease priority: $\mathcal{O}(1)$ amortized

Lemma (Fibonacci Lemma)

Let H be a Fibonacci heap with n items. Then

$$\text{rank}(H) \leq \log_{\phi} n \approx 1.44 \log_2 n$$

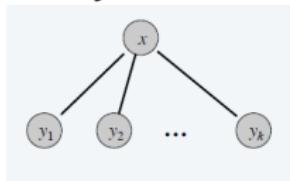
Proof of Fibonacci Lemma

Lemma (Lemma #1)

Consider a sequence of operations leading to a Fibonacci heap H , and consider a node x with rank k , and let y_1, \dots, y_k be its children, indexed in the order in which they were linked to x .

Then

$$\text{rank}(y_i) \geq \begin{cases} 0 & \text{if } i = 1, \\ i - 2 & \text{if } i \geq 2. \end{cases}$$

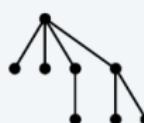


Proof

- When y_i was linked to x , x had already $\geq i - 1$ (it might have lost a few later) children
- Only trees of equal rank are linked so $\text{rank}(y_i) = \text{rank}(x) \geq i - 1$
- Node y_i has lost at most one child, otherwise it would have been cut. Hence $\text{rank}(y_i) \geq i - 2$

Proof of Fibonacci Lemma

Let T_k be the smallest possible tree of rank k satisfying the previous lemma.

 T_0  T_1  T_2  T_3  T_4  T_5 

$F_2 = 1$

$F_3 = 2$

$F_4 = 3$

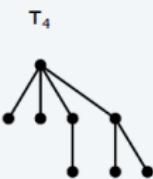
$F_5 = 5$

$F_6 = 8$

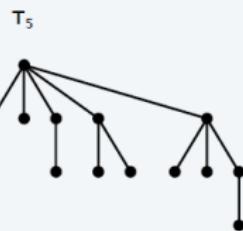
$F_7 = 13$

Proof of Fibonacci Lemma

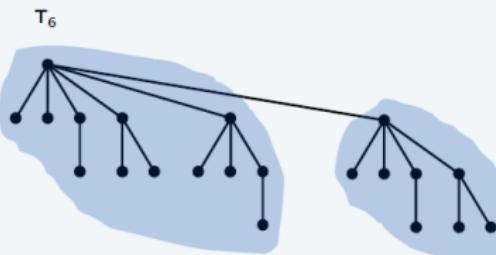
Let T_k be the smallest possible tree of rank k satisfying the previous lemma.



$$F_6 = 8$$



$$F_7 = 13$$



$$F_8 = F_6 + F_7 = 8 + 13 = 21$$

Proof of Fibonacci Lemma

Lemma (Lemma #2)

Let $s_k = |T_k|$. Then $s_k \geq F_{k+2}$, the $(k+2)$ -th Fibonacci number.

Proof

- Basis of induction: $s_0 = 1 \geq F_2$, $s_1 = 2 \geq F_3$
 $(F_0 = 0, F_1 = 1, F_2 = F_0 + F_1 = 1, F_3 = F_1 + F_2 = 2, \dots)$
- Inductive step:

$$\begin{aligned}s_k &\stackrel{\text{Lemma } \#1}{\geq} 1 + 1 + s_0 + \cdots + s_{k-2} \\&\geq 1 + F_1 + F_2 + \cdots + F_k = F_{k+2}.\end{aligned}$$

Exercise: Prove $F_{k+2} = 1 + F_0 + F_1 + \cdots + F_k$.



Proof of Fibonacci Lemma

Lemma (Lemma #3)

For any $k \geq 0$, the $(k + 2)$ -th Fibonacci number satisfies

$F_{k+2} \geq \phi^k$, with $\phi = (1 + \sqrt{5})/2$ (**golden ratio**), the largest root of $\phi^2 - \phi - 1 = 0$.

Proof

By induction on k .

- Basis: $F_2 = 1 = \phi^0$, $F_3 = 2 \geq \phi$.
- Inductive step:

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \geq \phi^{k-1} + \phi^{k-2} \\ &= \phi^{k-2}(\phi + 1) = \phi^{k-2} \cdot \phi^2 = \phi^k. \end{aligned}$$



Proof of Fibonacci Lemma

Fibonacci Lemma

Let k be the rank of H . In the worst case, there is only one tree in H with k children and by definition

$$n \geq s_k \geq^{\text{Lemma #2}} F_{k+2} \geq^{\text{Lemma #3}} \phi^k$$

Hence

$$k = \text{rank}(H) \leq \log_\phi n \approx 1.44 \log_2 n$$



Other operations

Meld: Concatenate the root lists. Cost: $\mathcal{O}(1)$. Amortized cost is also constant since $\Delta\phi = 0$.

Delete key: Decrease priority of the element to $-\infty$ and then extract_min. Amortized cost is $\mathcal{O}(1 + \text{rank}(H)) = \mathcal{O}(\log n)$.

Make-heap: Builds initial heap inserting n items: cost is $\mathcal{O}(n)$.

To learn more

-  M.L. Fredman and R. E. Tarjan
Fibonacci Heaps and their Used to Improve Network Optimization Algorithms
J. of the ACM 34(3):596–615, 1987
-  T. Cormen, C. Leiserson, R. Rivest and C. Stein.
Introduction to Algorithms, 3rd ed
The MIT Press, 2009

Part IV

Data Structures for Strings

8 Tries

9 Suffix Trees

Part IV

Data Structures for Strings

8

Tries

9

Suffix Trees

Tries

We often deal with keys which are sequences of symbols (characters, decimal digits, bits, . . .). Such a decomposition is usually very natural and can be exploited to implement efficient dictionaries.

Moreover, we usually want, besides lookups and updates, operations in which the keys as symbol sequences matter: for example, we might want an operation that, given a collection of words C and some word w , returns all words in C that contain w as a subsequence. Or as a prefix, or a suffix, etc.

Tries

Consider a finite alphabet $\Sigma = \{\sigma_1, \dots, \sigma_m\}$ with $m \geq 2$ symbols. We denote Σ^* , as usual in the literature, the set of all strings that can be formed with symbols from Σ . Given two strings u and v in Σ^* we write $u \cdot v$ for the string which results from the concatenation of u and v . We will use λ to denote the empty string or string of length 0.

Definition

Given a finite set of strings $X \subset \Sigma^*$, all of identical length, the *trie* T of X is an m -ary tree recursively defined as follows:

- ① If X contains a single element x or none, then T is a tree consisting on a single node that contains x or is empty.
- ② If $|X| \geq 2$, let T_i be the trie for the subset

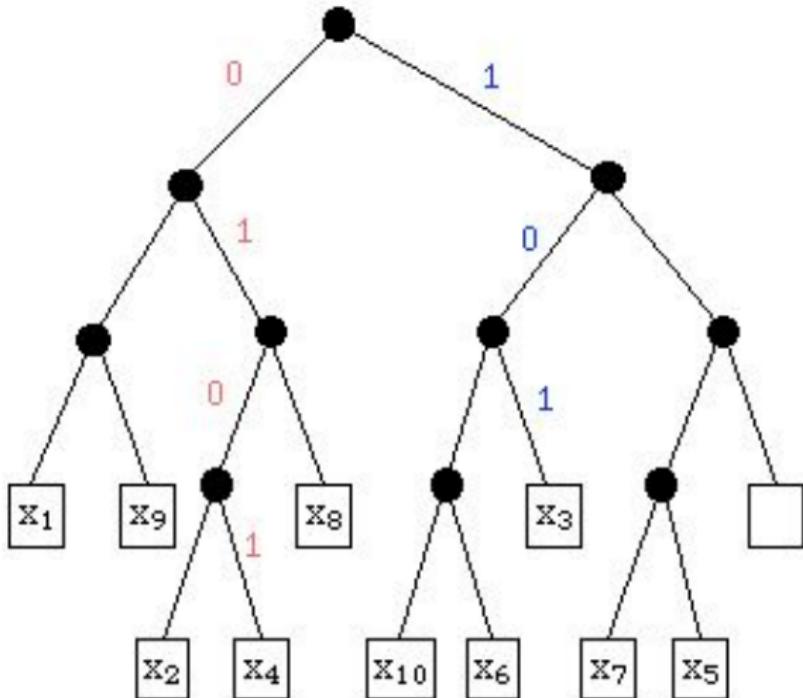
$$X_i = \{y \mid x = \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$$

Then the trie T for X is the m -ary tree consisting of a root \circ and m subtrees T_1, T_2, \dots, T_m .

Tries

$m=2$

$x_1 = 000101$
 $x_2 = 010001$
 $x_3 = 101000$
 $x_4 = 010101$
 $x_5 = 110101$
 $x_6 = 100111$
 $x_7 = 110001$
 $x_8 = 011111$
 $x_9 = 001110$
 $x_{10} = 100001$



Tries

Lemma

If the edges in the trie T for X are labelled in such a way that the edge connecting the root of T with subtree T_i has label σ_i , $1 \leq i \leq m$, then the sequence of labels in the path from the root to a non-empty leaf that contains x form the shortest prefix that univoquely identifies x , that is, the shortest prefix of x which is not shared by any other element of X

Lemma

Let p be the sequence of labels in a path from the root of the trie T to some node v (either internal or leaf); the subtree rooted at v is a trie for the subset of all strings in X starting with the prefix p (and no other strings)

Tries

Lemma

Given a finite set $X \subset \Sigma^$ of strings of equal length, the trie T for X is unique; in particular, T does not depend on the order in which we “present” or “insert” the elements of X*

Lemma

The height of a trie T is the minimum length of prefixes needed to distinguish two elements in X ; in other words, the length of the longest prefix which is common to ≥ 2 elements in X ; of course, if ℓ is the length of the string in X then

$$\text{height}(T) \leq \ell$$

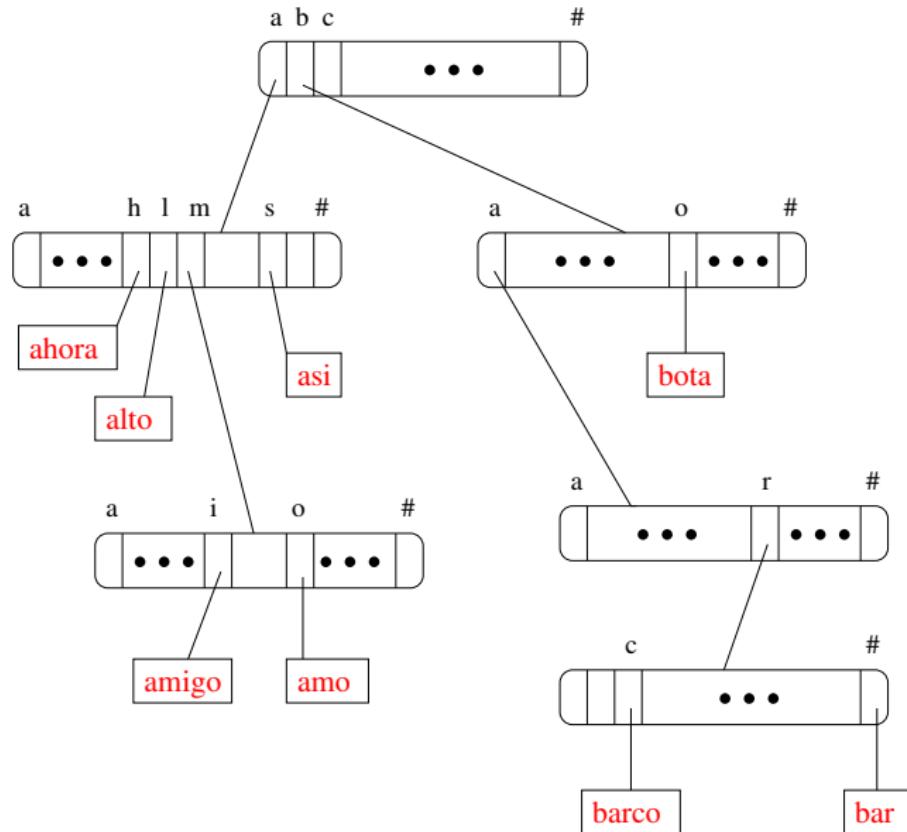
Tries

Our definition of tries requires all strings being of the same length; that's too restrictive. What we actually need is that no string in X is a proper prefix of another string in X .

The standard solution to the problem is to extend Σ with a special symbol (e.g. $\ddot{\imath}$) to mark the end of strings. If we append $\ddot{\imath}$ to the end of all strings in X , then no (marked) string is a proper prefix of another string. The modest price to pay is to work with an alphabet of $m + 1$ symbols

Tries

$X = \{\text{ahora, alto, amigo, amo, asi, bar, barco, bota, ...}\}$



Tries

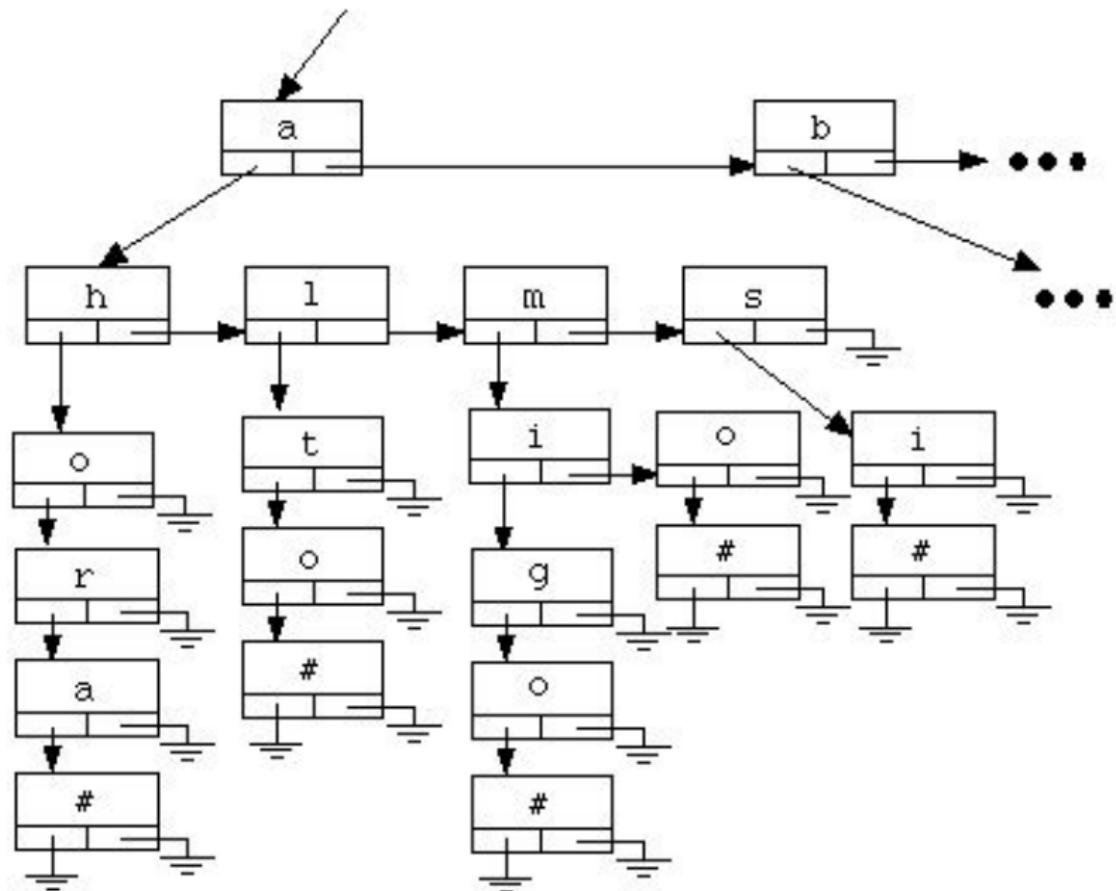
In order to implement tries we can use standard techniques to implement m -ary trees, namely, use an **array of pointers** for each internal node or use the first child-next sibling representation.

If using arrays of pointers, the pointers give us access to the root node of every non-empty subtree of the current node, and the symbols of Σ can be used to indices into the array of pointers (eventually with some easy bijective mapping $f : \Sigma \rightarrow \{0, \dots, m - 1\}$). Leaves that are not empty can contain the remaining suffix of the element, the prefix given by the path from the root to the leave.

Tries

When using first child-next sibling, each node stores a symbol c ; the pointer to the first child points to the root of the trie of words that have c at that level, while the next sibling points to the node giving us access to some other trie, now with some other symbol d . It is usual that since the symbols in Σ typically admit a natural total order then the list of children of a node in the trie are increasingly sorted according to that order.

Tries



Tries

Despite it is more costly in space to use nodes to store full string, symbol by symbol, instead of storing suffixes once a leaf is reached, it is advantageous to avoid different types of nodes, different types of pointers or forcing pointers of one type point to nodes of some other type, using wasteful unions,...

```
// We assume that the class Key supports
// x.length() = length() >= 0 of key x
int Key::length() const;

// x[i] = i-th symbol of key x;
// throws an exception if i < 0 or i >= x.length()
template <typename Symbol>
Symbol Key::operator[](const Key& x, int i);

template <typename Symbol, typename Key, typename Value>
class DigitalDictionary {
public:
    ...
private:
    struct trie_node {
        Symbol _c;
        trie_node* _first_child;
        trie_node* _next_sibl;
        Value _v;
    };
    trie_node* root;
    ...
};
```

Tries

```
template <typename Symbol,
          typename Key,
          typename Value>
void DigitalDictionary<Symbol,Key,Value>::lookup(
    const Key& k, bool& exists, Value& v) const {
    trie_node* p = _lookup(root, k, 0);
    if (p == nullptr)
        exists = false;
    else {
        exists = true;
        v = p -> _v;
    }
}
```

Tries

```
// Pre: p points to the root of the subtree that contains
// all elements such that their first i-1 symbols
// coincide with the first i-1 symbols of the key k
// Post: returns a pointer to the node that stores the value
// associated to $k$ if $\text{\color{red} pair\{k,v\}}$ belongs to the dictionary
// and a null pointer if not such pair exists
// Cost: ;{\color{red} \bigOh(|k|\cdot m)}}
template <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::trie_node*
DigitalDictionary<Symbol,Key,Value>::_lookup(trie_node* p,
                                             const Key& k, int i) const {
    if (p == nullptr)    return nullptr;
    if (i == k.length()) return p;
    if (p->_c > k[i])  return nullptr;
    if (p->_c < k[i])
        return _lookup(p->_next_sibl, k, i);
    // p->_c == k[i]
    return _lookup(p->_first_child, k, i+1);
}
```

Tries

```
template <typename Symbol,
          typename Key,
          typename Value>
void DigitalDictionary<Symbol,Key,Value>::insert(
    const Key& k, const Value& v) {
    _root = _insert(root, k, 0);
}
```

Tries

```
// Pre: p points to the root of the subtree that contains
// all elements such that their first i-1 symbols
// coincide with the first i-1 symbols of the key k
// Post: returns a pointer to the root of the tree resulting from
// the insertion of the pair $\pair{k[i..],v}$ in the subtree
// Cost: ;{\color{red} \$\bigOh(|k| \cdot m)}}
template <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::trie_node*
DigitalDictionary<Symbol,Key,Value>::_insert(trie_node* p,
                                           const Key& k, int i) const {
    if (i == k.length()) {
        if (p == nullptr) p = new trie_node;
        p->_c = Symbol(); // Symbol() is the end-of-string symbol
                            // e.g. Symbol() == '\0' or Symbol() == '\sharp'
        p->_v = v;
        return p;
    }
    if (p == nullptr or p->_c > k[i]) {
        trie_node* p = new trie_node;
        p->_next_sibl = p;
        p->_c = k[i];
        p->_first_child = _insert(nullptr, k, i+1);
        return p;
    }
    if (p->_c < k[i])
        p->_next_sibl = _insert(p->_next_sibl, k, i);
    else // p->_c == k[i]
        p->_first_child = _insert(p->_first_child, k, i+1);
    return p;
}
```

Ternary Search Trees

One alternative to implement tries is to represent the trie nodes as binary search trees with pointers to roots of subtrees, instead of as array of pointers to roots, or as linked lists of pointers to roots (of non-empty subtrees).

The new data structure, invented by Bentley and Sedgewick (1997), is called an **ternary search tree** (TST). It tries to combine the efficiency in space of list-tries (we avoid the large number of null pointers when using arrays) and the efficiency in time (we avoid the linear “scans” when using lists to navigate to the appropriate subtree).

Nodes in TSTs have a symbol c and 3 pointers each: pointers to the left and right child of the node in the BST that represents the trie “node”, and a central pointer to the root of the subtree with symbol c at that level.

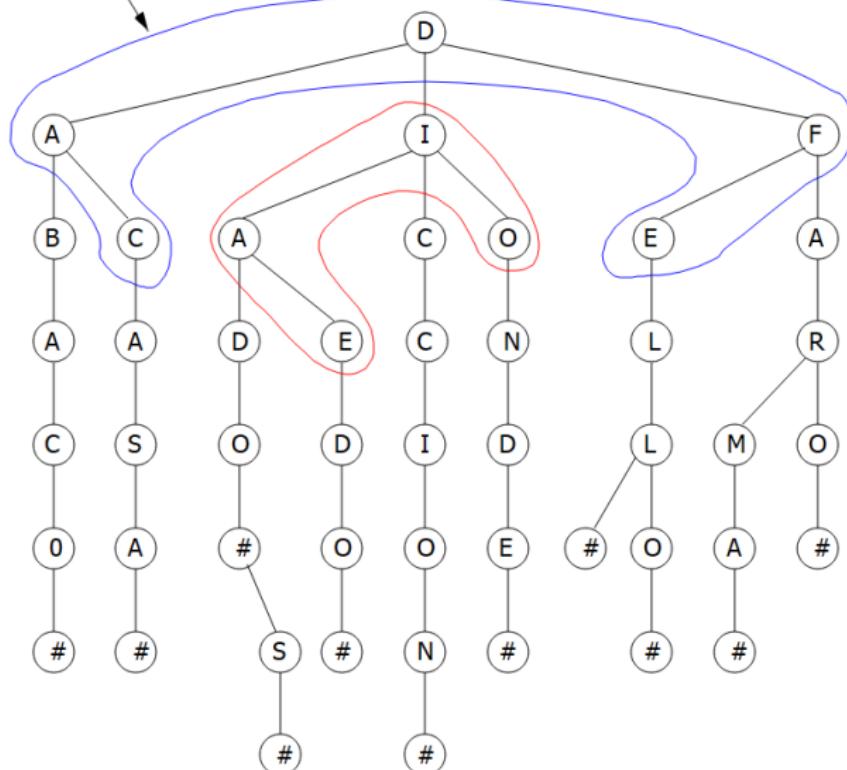
Ternary Search Trees

```
template <typename Symbol,
          typename Key,
          typename Value>
class DigitalDictionary {
public:
    ...
    void lookup(const Key& k, bool& exists, Value& v) const;
    void insert(const Key& k, const Value& v);
    ...
private:
    struct tst_node {
        Symbol _c;
        tst_node* _left;
        tst_node* _cen;
        tst_node* _right;
        Value _v;
    };
    tst_node* root;
    ...
static tst_node* _lookup(tst_node* t,
                        int i, const Key& k, const Value& v);
static tst_node* _insert(tst_node* t,
                        int i, const Key& k, const Value& v);
    ...
};
```

Ternary Search Trees

X = {DICCION, DADO, DADOS, DEDO, DONDE, ABACO, CASA, FARO, FAMA, ELLO, EL}

a "node" in the trie



Ternary Search Trees

```
// Pre: p points to the root of the subtree that contains
// all elements such that their first i-1 symbols
// coincide with the first i-1 symbols of the key k
// Post: returns a pointer to the node that stores the value
// associated to $k$ if $\pair{k,v}$ belongs to the dictionary
// and a null pointer if not such pair exists
// Expected cost: ;{\color{red} \bigOh{|k| \log m}};
template <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::trie_node*
DigitalDictionary<Symbol,Key,Value>::_lookup(tst_node* p,
                                             const Key& k, int i) const {
    if (p == nullptr)    return nullptr;
    if (i == k.length()) return p;
    if (k[i] < p->_c > k[i]) return _lookup(p->_left, k, i);
    if (k[i] == p->_c)   return _lookup(p->_cen, k, i+1);
    if (p->_c < k[i])   return _lookup(p->_right, k, i);
}
```

Ternary Search Trees

```
template <typename Symbol,
          typename Key,
          typename Value>
void DigitalDictionary<Symbol,Key,Value>::insert(
    const Key& k, const Value& v) {
    // Symbol() is the end-of-string symbol
    // e.g. Symbol() == '\0' or Symbol() == '\sharp'
    k[k.length()] = Symbol(); // add end-of-string
    root = _insert(root, 0, k, v);
}
```

Ternary Search Trees

```
template <typename Symbol,
          typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>::::tst_node*
DigitalDictionary<Symbol,Key,Value>::::_insert(
    tst_node* t, int i,
    const Key& k, const Value& v) {

    if (t == nullptr) {
        t = new tst_node;
        t->_left = t->_right = t->_cen = nullptr;
        t->_c = k[i];
        if (i < k.length() - 1) {
            t->_cen = _insert(t->_cen, i + 1, k, v);
        } else { // i == k.length() - 1; k[i] == Symbol()
            t->_v = v;
        }
    } else {
        if (t->_c == k[i])
            t->_cen = _insert(t->_cen, i + 1, k, v);
        if (k[i] < t->_c)
            t->_left = _insert(t->_left, i, k, v);
        if (t->_c < k[i])
            t->_right = _insert(t->_right, i, k, v);
    }
    return t;
}
```

Performance of Tries

There are several measures of the performance of tries in terms of space and time of the different operations.

For example, in tries using arrays of pointers and considering an extended alphabet with $m + 1$ symbols a tree for a set of n elements will contain $\geq n$ leaves; how many of them?

A common model to study the average behavior of tries (array, list or TST) is to consider that the n strings are produced by some memoryless source (so that the r -th symbol of the element is symbol σ_i with a probability p_i irresp. of the previous symbols and r) or some Markovian model there is a probability $p_{i,j}$ that some symbol is σ_i given that the preceding sysmbol was σ_j

Performance of Tries

Theorem (Clément, Flajolet, Vallée (1998))

*The **external path length** (EPL) in a random trie of n elements produced by a random source S is*

$$\frac{C_S}{H_S} n \log n + o(n \log n)$$

where both C_S and H_S are constants depending on the random source S ; moreover C_S depends on the specific implementation of the trie (array, list, TST).

The EPL is the sum of the length of all paths from the root of the trie to all leaves in the trie; a random search (successful or unsuccessful) will cost, on average $C_S / H_S \log n$

Performance of Tries

For example, if the source is memoryless with probabilities p_1, \dots, p_m , for the symbols of the alphabet ($\sum_i p_i = 1$) then $H_S = -\sum_i p_i \log p_i$ is the **entropy** of the source and

Type	C_S
Array	1
List	$\sum_i (i - 1)p_i$
TST	$2 \sum_{i < j} \frac{p_i p_j}{p_i + \dots + p_j}$

When all $p_i = 1/m$ we have that $H_S = \log m$ and hence the cost of random searches is

Type	Cost of random search
Array	$\log_m n$
List	$\approx \frac{m}{2} \log_m n$
TST	$\approx 2 \ln m \log_m n = 2 \ln n$

Performance of Tries

Let N denote the total number of symbols (including end-of-string, if needed) required for our n strings. The shared prefixes will provide for a more compact representation of the set of strings and thus we expect to need $\ll N$ nodes (assuming we store one symbol per node like in list-tries and TSTs).

On the other hand, it has been shown that to store n strings in a trie we will need, on average, n/H_S internal nodes (Knuth 1968, Regnier 1988), hence the average number of pointers is $n \cdot (m + 1)/H_S$ (array-tries), $2n/H_S$ (list-tries) and $3n/H_S$ (TSTs).

Performance of Tries

For example, for a memoryless source with m symbols all equally likely, we will need on average $n / \ln m$ nodes, e.g., $n / \ln 2$ nodes in a binary trie.

In list-tries and TSTs the number of nodes coincides, in the worst-case, with N as each node holds one symbol. But in practice the common prefixes will be exploited giving a reduction by a factor of $1/H_S$ ($\approx 1 / \ln m$ for string made out of equally likely independent symbols)

Patricia (a.k.a. Compressed Tries)

When an internal node in a trie has only one non-empty subtree we say that such a node is **redundant**. In a **compressed trie** or **Patricia** (*Practical Algorithm To Retrieve Information Coded in Alphanumeric*, D. Morrison, 1968) chains of redundant nodes are substituted by a single node, and edges labeled by subsequence of one or more symbols.

In a Patricia the n strings are stored in the n leaves, and by definition there are no empty leaves with one or more non-empty siblings.

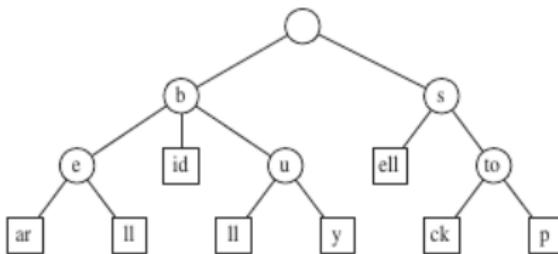
Patricia

To implement Patricia we can use the first child/next sibling or the TST representation, but instead of storing one symbol per node, we can store a substring of ≥ 1 symbols, or keep skip attributes that indicate which position has to be examined next if we move to a descendant node in the trie.

Thus during a search, if we reach a node x at level i which we have matched the first $i - 1$ symbols of the given key and the node has the substring $w = w_0 \dots w_{j-1}$ as a label, then we will have to see if $k[i - 1..i + j - 2]$ matches w , and if so, we “descend”, otherwise we will continue looking for the appropriate subtree or declare the search unsuccessful—this will depend on whether we are compacting a list-trie or a TST.

Patricia

- In a list-Patricia, the first child pointer will point to the subtree that contains all strings with prefix $k[0..i - 2] \cdot w$, and the next sibling will give us access to the subtrees that store words with a prefix $\geq k[0..i - 2] \cdot w'_0$, where w'_0 is the successor of w_0 in Σ in the alphabetic order.
- In a bst-Patricia, the central child contains all strings with prefix $k[0..i - 2] \cdot w$, the left child all strings with a smaller prefix, the right child all strings with a larger prefix



A Patricia for the set $X = \{\text{bear}, \text{bell}, \dots, \text{stock}, \text{stop}\}$.

Source: M. T. Goodrich & R. Tamassia

Patricia

Despite Patricia saves on empty leaves and it might be slightly more efficient to store common infixes in the internal nodes instead of the full expanded infixes, one symbol per node, but the major advantage of Patricia occurs when the strings are externally stored and Patricia is an index into the external storage.

For example, if we have an array of strings $S[0..M - 1]$, we can designate the substring between positions i and j of $S[k]$ by a triplet $\langle k; i, j \rangle$; we can build our Patricia storing such triplets in the nodes instead of symbols or subsequences of symbols.

Patricia

	0	1	2	3	4
$S[0] =$	s	e	e		
$S[1] =$	b	e	a	r	
$S[2] =$	s	e	l	l	
$S[3] =$	s	t	o	c	k

	0	1	2	3
$S[4] =$	b	u	l	l
$S[5] =$	b	u	y	
$S[6] =$	b	i	d	

	0	1	2	3
$S[7] =$	h	e	a	r
$S[8] =$	b	e	l	l
$S[9] =$	s	t	o	p

(a)



(b)

Patricia

```
template <typename Symbol, typename Key,
          typename Value>
DigitalDictionary<Symbol,Key,Value>:::patricia_node*
DigitalDictionary<Symbol,Key,Value>:::_lookup(patricia_node* p,
                                             const Key& k, int i) const {
    if (p == nullptr)    return nullptr;
    if (i == k.length()) return p;
    triplet x = p -> _x; // x=(idx,first,last)
    int len = x.last - x.first + 1; // length of the subsequence
    if (k[i..i+len-1] < S[x.idx][first..last])  return nullptr;
    if (S[idx][first..last] < k[i..i+len-1])
        return _lookup(p -> _next_sibl, k, i);
    // S[idx][first..last] == k[i..i+len-1]
    return _lookup(p -> _first_child, k, i+len);
}
```

Patricia

The lookup algorithm in Patricia has cost $\mathcal{O}(\ell \cdot m)$ in the worst-case, where ℓ is the length of the longest string in the set, and m the size of the alphabet. If we combine TST & compression we get expected cost $\mathcal{O}(\ell \cdot \log m)$ for searches. Likewise the cost of insertions and deletions will be like that of search.

Inverted files

One interesting application of tries and Patricia is **inverted files** (a.k.a. **inverted indices**).

Suppose we have a large collection of documents D_1, \dots, D_T . For each document we extract the unique set of words (**vocabulary, index terms**) of each document (we eventually record the positions of the document at which each unique word occurs). It is also frequent to remove common words such as pronouns, articles, connectives, ... known as **stopwords**.

Inverted files

We then proceed to insert/update, one by one, each index term of each document, in a trie (or TST or Patricia). When a word appears in several different documents we will keep track in a **occurrence list**. Because of their sheer volume, occurrence lists will be typically stored in secondary memory, and they won't be kept in any particular order.

When we process a word w from document D_i , we consider three cases

- ① w is a stopword: discard it and proceed to the next word/term in D_i (or start processing a new document)
- ② w was already in the inverted file: use the (compressed) trie to located the occurrence list and add a reference to document D_i to the list associated to word w ; or
- ③ w wasn't yet in the inverted list: create a new occurrence list with (w, D_i) , append the new occurrence list to the set of occurrence lists, and add a link from the (compressed) trie to the new occurrence list

Inverted files

Inverted indices are used in search engines to retrieve relevant documents as follows.

- ① The user query Q is normalized and stopwords removed
 - ② For each term/word t in Q , use the compressed trie (in main memory) to get the link to the corresponding occurrence list for t and retrieve it from secondary memory
 - ③ Intersect (*) the occurrence lists and sort the resulting set of documents according to some *relevance parameter* (e.g. the **PageRank** when the documents are web pages)
- (*) This is what we do when it is assumed that we want the subset of documents that contain **all** the terms in Q ; in some cases, the user will use operators or will allow a certain degree of mismatch, or we will have to produce results discarding terms that do not appear in the index

Inverted files

If $|V_i|$ is the size of the vocabulary V_i of D_i , without stopwords, then we will be building a (compressed) trie for

$$N = \left| \bigcup_{i=1}^T V_i \right|$$

words/terms, and the space that it will occupy will be roughly $\Theta(N/H_S)$, H_S being the empirical entropy for Σ , based upon the set of documents. On the other hand, we will have to store the N words and their respective occurrence lists somewhere else.

To construct the inverted file we will have to perform $D = \sum_{1 \leq i \leq T} |D_i|$ insertions/updates in the index and each such operation will have cost $\mathcal{O}(\ell)$, where ℓ is the length of the longest word in the collection of documents.

Inverted files

The cost of processing a query will be the cost of searching the Q terms in the (compressed) trie ($\mathcal{O}(|Q| \cdot \ell)$) plus the cost of merging the $|Q|$ occurrence lists and sorting the final result by relevance.

In practical situations the occurrence lists can be long, but not extremely long (words that **are not** stopwords do not appear in a significative fraction of the documents in the collection!). This is even more true for the final result (the “merging” of all occurrence lists) and the cost of sorting will be small compared to the others, hence the cost will be $\ll \Theta(|Q| \cdot T + T \log T)$, indeed, it will be close to $\Theta(|Q|)$, as the length of the occurrence lists can be thought as $\mathcal{O}(1)$

Tries

To learn more:

-  [D. E. Knuth](#)
The Art of Computer Programming, Volume 3: Sorting and
Searching, 2nd ed
[Addison-Wesley, 1998](#)
-  [M. T. Goodrich and R. Tamassia](#)
Algorithm Design and Applications
[John Wiley & Sons, 2015](#)

Tries

To learn more:

-  [J. L. Bentley and R. Sedgewick](#)
Fast algorithms for sorting and searching strings
Proc. SODA, pp. 360–369, 1997
-  [J. Clément, Ph. Flajolet and B. Vallée](#)
The Analysis of Hybrid Trie Structures
Proc. SODA, pp. 531–539, 1998

Part IV

Data Structures for Strings

8

Tries

9

Suffix Trees

Suffix Trees

A **suffix tree** (or *suffix trie*) is simply a trie for all the suffixes of a string, the **text**, $T[0..n - 1]$.

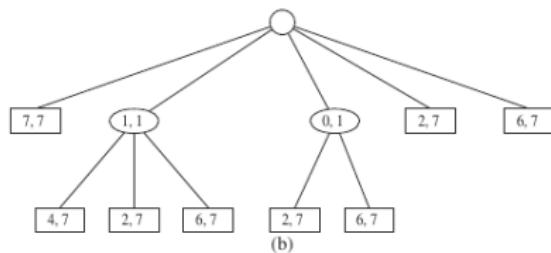
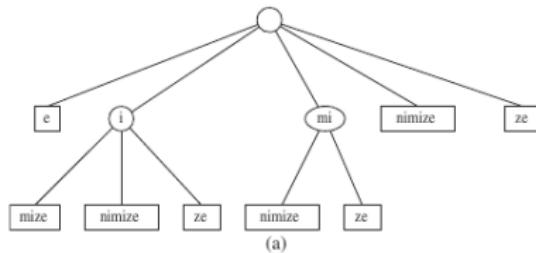
Thus we will form a trie with the M suffixes of the text T . Since the length of the text is n there are The total number of (proper) suffixes to store is n ($n + 1$ if we count the empty suffix) and the total number of symbols involved is

$$\leq \sum_{i=0}^n n - i = \frac{n(n+1)}{2}$$

as the suffix $T[i..n - 1]$ has length $n - i$, $0 \leq i \leq n$

Suffix Tries

A compact representation of the trie, storing the pair (i, j) to represent a substring $T[i..j]$ will be most convenient; in any case, using Patricia for the suffixes guarantees that the space used is $\Theta(n)$.



(a) the suffix trie for the text $T = \text{minimize}$

(b) the suffix trie in compact form

Suffix Tries

A naïve approach to the construction of suffix tries would need $\Theta(n^2)$ in the worst case—assuming that we use the array of pointers representation with $\Theta(1)$ cost to find which children to use in the next level, otherwise an extra factor m or $\log m$ appears

However there exist several linear-time algorithms for the construction of suffix tries

- ➊ Weiner, 1973 (*position trees*)
- ➋ McCreight, 1976 (more space efficient than Weiner's)
- ➌ Ukkonen, 1995 (same bounds as McCreight's, simpler)

They won't be covered here

Suffix Tries

Suffix tries have many applications. The most immediate one is the substring matching problem. We are given a text $T[0..n - 1]$ and a pattern $P[0..k - 1]$, typically $k \ll n$ and we need to show if P occurs as a substring of T , and if so, where.

Well known algorithms like Knuth-Morris-Pratt (KMP) or Boyer-Moore (BM) —there many other— solve this important problem in time by preprocessing the pattern in time $\Theta(k)$ and then scanning the text in time $\Theta(n)$, giving a total cost $\Theta(n + k)$ in the worst-case

Suffix Tries

The same bound is achieved with suffix tries: but we invest time $\Theta(n)$ in the preprocessing of the text (\rightarrow build the suffix trie!) and then search the pattern in the suffix trie with cost $\Theta(k)$.

The great news is that we can search many patterns very efficiently, the cost of building the suffix trie was paid once, the search of each pattern is lightning fast. This simply not possible with KMP, BM and many of the string matching algorithms since they preprocess each pattern and will need to scan the text with cost $\Theta(n)$ for every pattern. Unless we were given all the patterns in advance, in which case we can preprocess the whole set of p patterns at once (with cost $\Theta(k \cdot p + n)$ instead of cost $\Theta(k + n) \cdot p$). But this is not the case in many situations where the patterns to be searched are not known in advance.

Suffix tries

```
int k = P.size();
int j = 0;
suffix_trie_node* p = T.root;
do {
    bool fin = true;
    for(q:children of p) {
        int i = q.first;
        if (P[j] == T[i]) {
            int len = q.last - i + 1;
            if (k <= len) {
                // suffix is shorter than node label
                if (P[j..j+k-1] == T[i..i+k-1])
                    return 'match at i-j'
                else
                    return 'P not a substring of T'
            } else { // k > len
                if (P[j..j+len-1]==T[i..i+len-1])
                    k -= len; j += len; p = q;
                fin = false;
                break; // end the for(q:children of p) loop
            }
        }
    }
} while (not fin and p is not a leaf);
return 'P not a substring of T';
```

Suffix Trees

To learn more:



[Dan Gusfield](#)

Algorithms on Strings, Trees & Sequences

Cambridge Univ. Press, 1997

Acknowledgements

These slides are the outcome of my readings, numerous discussions with colleagues, inspiration from other similar courses (in particular, I owe particular thanks to Kevin Wayne's slides on Algorithm Design), and the comments and insight of my students along many years in which I have taught this material.

Special thanks go to: Manuel Joey Becklas, Felix Mühlenberend, and Hugo Sanz for pointing out several typos and errors. Their help to improve these slides is highly appreciated.