Introduction - Review

Advanced Algorithms and Data Structures - Lecture 1

Venanzio Capretta

Monday 30 September 2019

School of Computer Science, University of Nottingham

The Maximum Subarray Problem

Maximum Subarray Example

Consider the following list of natural numbers:

$$[4, -2, 3, -7, 5, 2, -6, 8, -4, 3, -2, 1]$$

Problem: Find the sublist with maximum sum

Maximum Subarray Example

Consider the following list of natural numbers:

$$[\underbrace{4, -2, 3}_{5}, -7, 5, 2, -6, 8, -4, 3, -2, 1]$$

Some sublists:

start index	end index	sum
0	2	5

Problem: Find the sublist with maximum sum

Maximum Subarray Example

Consider the following list of natural numbers:

$$[4, -2, 3, -7, 5, 2, -6, \underbrace{8, -4, 3, -2, 1}_{6}]$$

Some sublists:

start index	end index	sum
0	2	5
7	11	6

Problem: Find the sublist with maximum sum

Maximum Subarray Example

Consider the following list of natural numbers:

$$[4, -2, 3, -7, \underbrace{5, 2}_{7}, -6, 8, -4, 3, -2, 1]$$

Some sublists:

start index	end index	sum
0	2	5
7	11	6
4	5	7

Problem: Find the sublist with maximum sum

Maximum Subarray Example

Consider the following list of natural numbers:

$$[4, -2, 3, -7, \underbrace{5, 2, -6, 8}_{9}, -4, 3, -2, 1]$$

Some sublists:

start index	end index	sum
0	2	5
7	11	6
4	5	7

Problem: Find the sublist with maximum sum

In this case it is the list with:

$$\text{start index} = 4 \quad \text{end index} = 7 \quad \text{sum} = 9$$

Maximum Subarray Example

Consider the following list of natural numbers:

$$[4, -2, 3, -7, \underbrace{5, 2, -6, 8}_{9}, -4, 3, -2, 1]$$

Some sublists:

start index	end index	sum
0	2	5
7	11	6
4	5	7

Problem: Find the sublist with maximum sum

In this case it is the list with:

$$\text{start index} = 4 \quad \text{end index} = 7 \quad \text{sum} = 9$$

Write an algorithm to compute the maximum sublist

(Applications: gene sequence analysis, computer vision, data mining)

Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a brute force algorithm that checks all sublists:

Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a brute force algorithm that checks all sublists:

- Create the list of all sublists

  $[[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \ldots, [3, -2, 1], [-2], [-2, 1], [1]]$

Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a brute force algorithm that checks all sublists:

- Create the list of all sublists

$$[[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \ldots, [3, -2, 1], [-2], [-2, 1], [1]]$$

- Compute the sum of all sublists

$$[4, 2, 5, -2, \ldots, 2, -2, -1, 1]$$

Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a brute force algorithm that checks all sublists:

- Create the list of all sublists

$$[[4],[4,-2],[4,-2,3],[4,-2,3,-7],\ldots,[3,-2,1],[-2],[-2,1],[1]]$$

- Compute the sum of all sublists

$$[4,2,5,-2,\ldots,2,-2,-1,1]$$

- Take the maximum

$$\text{maximum}\,[4,2,5,-2,\ldots,2,-2,-1,1]=9$$

Brute Force Algorithm

We will look at several algorithms for the Maximum Subarray Problem

The first is a brute force algorithm that checks all sublists:

- Create the list of all sublists

  $$[[4], [4, -2], [4, -2, 3], [4, -2, 3, -7], \ldots, [3, -2, 1], [-2], [-2, 1], [1]]$$

- Compute the sum of all sublists

  $$[4, 2, 5, -2, \ldots, 2, -2, -1, 1]$$

- Take the maximum

  $$\text{maximum}\,[4, 2, 5, -2, \ldots, 2, -2, -1, 1] = 9$$

  This is very inefficient (cubic complexity)

  Small improvement: reuse the sums already computed

Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
                     else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                    else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

The type of the function `maxSub` says that

`maxSub` is a function that maps a list to a triple (i, j, s)

- i is the index of the first element of the sublist
- j is the index of the last element of the sublist
- s is the sum of the sublist

Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                    else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

If the input is a singleton list `[x]`
then the singleton is the maximum list

- 0: index of `x`, first element of the sublist
- 0: `x` is also the last element of the sublist
- x: the sum of `[x]` is x

Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                     else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

For longer lists we split the computation in two parts

- Sublists that contain the first element of `xs`
  let $(0, i_0, s_0)$ be the maximum of them
- Sublists that do not contain the first element
  Then they are sublists of `(tail xs)`
  let (i, j, s) be the recursive max sublist of `(tail xs)`
- Choose the larger of the two

Haskell Code

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                     else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

Ausiliary functions:

- `sums` computes the progressive sums of lists starting at the beginning
  `sums [1,-2,3,5] = [(0,1),(1,-1),(2,2),(3,7)]` because
  - the sum of `[1]` (indices 0 and 0) is 1 $\Rightarrow (0,1)$
  - the sum of `[1,-2]` (indices 0 and 1) is -1 $\Rightarrow (1,-1)$
  - the sum of `[1,-2,3]` (indices 0 and 2) is 2 $\Rightarrow (2,2)$
  - the sum of `[1,-2,3,5]` (indices 0 and 3) is 7 $\Rightarrow (3,7)$
- `argMax snd` selects the element
  with the maximum second component (the sum)

Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

A: NO, you just need to understand the code

Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

A: NO, you just need to understand the code

- I give examples and solutions in Haskell
  You must be able to understand my code
  I only use basic Haskell (no Advanced Functional Programming)

Do I need to know Haskell?

Q: Am I supposed to program in Haskell?

A: NO, you just need to understand the code

- I give examples and solutions in Haskell
  You must be able to understand my code
  I only use basic Haskell (no Advanced Functional Programming)
- You don't need to program in Haskell yourself
  Use your favourite programming language
  The textbook has pseudocode in imperative style

Complexity of the Algorithm

Exercise: What is the complexity of the `maxSub` algorithm?
(How long does it take to to compute on an input of size n?)

PRETTY BAD (we'll see how bad)

Q: Are there more efficient algorithms?
YES: We will see two of them

Complexity of the Algorithm

Exercise: What is the complexity of the `maxSub` algorithm?
(How long does it take to to compute on an input of size n?)

PRETTY BAD (we'll see how bad)

Q: Are there more efficient algorithms?
YES: We will see two of them

But First:
We see the Outline of the course
We review the basics of computational complexity

Introduction and Prerequisites

What is this course about?

Course Contents:

What is this course about?

Course Contents:

- Advanced Data Structures
  How to store data efficiently
  Graphs, Search Threes, Networks, Heaps

What is this course about?

Course Contents:

- Advanced Data Structures
  How to store data efficiently
  Graphs, Search Threes, Networks, Heaps
- Advanced Algorithms
  Efficient Algorithms to solve important problems
  Fast search, route planning, network flow, scheduling

What is this course about?

Course Contents:

- Advanced Data Structures
  How to store data efficiently
  Graphs, Search Threes, Networks, Heaps
- Advanced Algorithms
  Efficient Algorithms to solve important problems
  Fast search, route planning, network flow, scheduling
- Programming Techniques
  Divide-and-Conquer, Dynamic Programming

What is this course about?

COURSE CONTENTS:

- Advanced Data Structures
  How to store data efficiently
  Graphs, Search Threes, Networks, Heaps
- Advanced Algorithms
  Efficient Algorithms to solve important problems
  Fast search, route planning, network flow, scheduling
- Programming Techniques
  Divide-and-Conquer, Dynamic Programming
- Complexity Analysis
  The Master Method, Amortized Complexity

Special Topics

Extra Subjects: Trendy Structures/Algorithms

- RSA Public-Key Cryptosystem
- Neural Networks (the Gradient-Descent Algorithm)
- Page Rank (the Google Search Algorithm)

Prerequisites

- Discrete Math (MCS)
  IA Ch.3 and Appendices A and B
- Basic Algorithms and Data Structures (ACE)
  stacks, lists, trees (IA Ch.10)
  sorting (IA Ch.2)
  elements of computational complexity
- Programming skills
  In *some* programming language: C/C++, Java, Python, Haskell
  You need to understand Haskell code, but you don't have to write it

Complexity Classes

Running Times

We measure the complexity of an algorithm
by the time it takes to execute:

$$T(n)$$

n is the size of the input

$T(n)$ is the time it takes to run on inputs of size n

Running Times

We measure the complexity of an algorithm
by the time it takes to execute:

$$T(n)$$

n is the size of the input

$T(n)$ is the time it takes to run on inputs of size n

It may take different times for different inputs of the same size n.

In general we mean the worst-case running time.
(Sometimes we're interested in the average running time.)

Running Times

We measure the complexity of an algorithm
by the time it takes to execute:

$$T(n)$$

n is the size of the input

$T(n)$ is the time it takes to run on inputs of size n

It may take different times for different inputs of the same size n.

In general we mean the worst-case running time.
(Sometimes we're interested in the average running time.)

We don't mean exact running time
(that depends on the implementation and machine)
but a measure of the number of elementary computation steps

How to measure input and time

Strictly speaking we shuld measure:

- input size by the number of bits the input takes in memory
- running time by the number of basic hardware operations executed

How to measure input and time

Strictly speaking we shuld measure:

- input size by the number of bits the input takes in memory
- running time by the number of basic hardware operations executed

In complexity theory we're usually more relaxed

- input size is often measured by memory locations, for example lists are measured by their lengths, trees by the number of nodes
- running time is measured by assuming that certain elementary operations (for example arithmetic, logical, pointer operations) take constant time (which is usually false!)

How to measure input and time

Strictly speaking we shuld measure:

- input size by the number of bits the input takes in memory
- running time by the number of basic hardware operations executed

In complexity theory we're usually more relaxed

- input size is often measured by memory locations, for example lists are measured by their lengths, trees by the number of nodes
- running time is measured by assuming that certain elementary operations (for example arithmetic, logical, pointer operations) take constant time (which is usually false!)

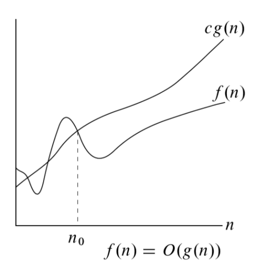
Important is not the exact running time, but the Complexity Class:
the algorithms runs in linear, or quadratic, ... or exponential time

Big-O notation

The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$

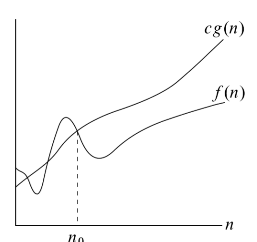
Big-O notation

The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$



Big-O notation

The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$

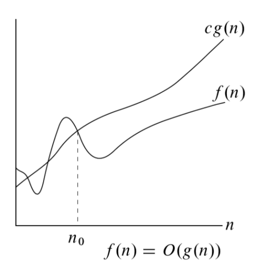


$f(n) = O(g(n))$

- there is a multiple c of $g(n)$ such that

Big-O notation

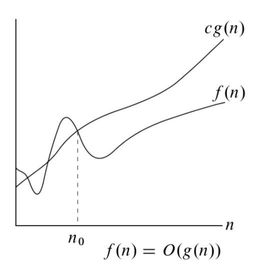
The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$



- there is a multiple c of $g(n)$ such that
- after a certain size n_0

Big-O notation

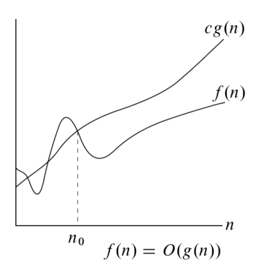
The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$



- there is a multiple c of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays below $cg(n)$

Big-O notation

The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$

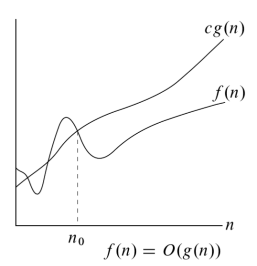


- there is a multiple c of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays below $cg(n)$

$f(n) = O(g(n))$ means: there is a costant c and a number n_0 such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$

Big-O notation

The notation $f(n) = O(g(n))$ intuitively means that
the function $f(n)$ grows at most as $g(n)$



- there is a multiple c of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays below $cg(n)$

$f(n) = O(g(n))$ means: there is a costant c and a number n_0 such that for all $n \geq n_0, 0 \leq f(n) \leq cg(n)$

Formally we should write $f \in O(g)$ where $O(g)$ is the set of functions

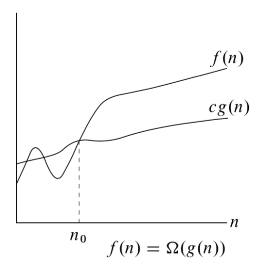
$$O(g) = \{f \mid \exists c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$

Big-Ω notation

The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$

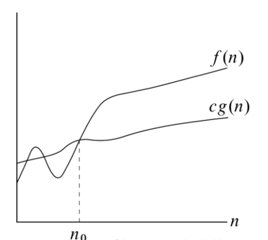
Big-Ω notation

The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$



Big-Ω notation

The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$

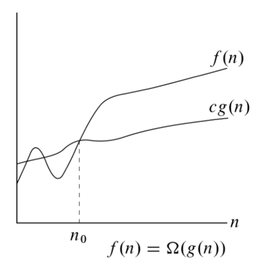


$f(n) = \Omega(g(n))$

- there is a multiple c of $g(n)$ such that

Big-Ω notation

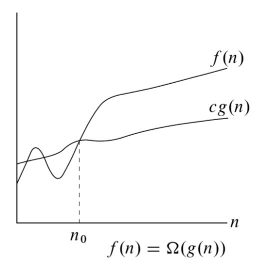
The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$



- there is a multiple c of $g(n)$ such that
- after a certain size n_0

Big-Ω notation

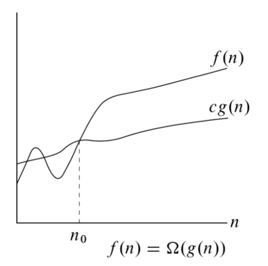
The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$



- there is a multiple c of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays above $cg(n)$

Big-Ω notation

The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$

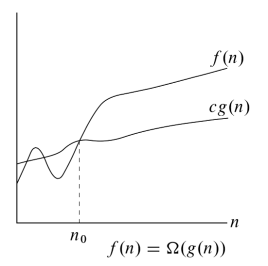


- there is a multiple c of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays above $cg(n)$

$f(n) = \Omega(g(n))$ means: there is a costant c and a number n_0 such that for all $n \geq n_0, 0 \leq cg(n) \leq f(n)$

Big-Ω notation

The notation $f(n) = \Omega(g(n))$ intuitively means that
the function $f(n)$ grows at least as $g(n)$



- there is a multiple c of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays above $cg(n)$

$f(n) = \Omega(g(n))$ means: there is a costant c and a number n_0 such that for all $n \geq n_0, 0 \leq cg(n) \leq f(n)$

Formally we should write $f \in \Omega(g)$ where $\Omega(g)$ is the set of functions

$$\Omega(g) = \{f \mid \exists c, \exists n_0, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$

Θ notation

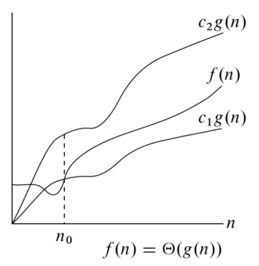
$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$

the function $f(n)$ grows like $g(n)$

Θ notation

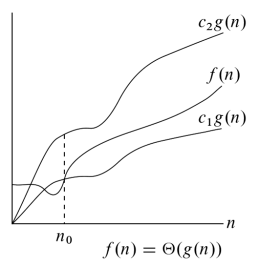
$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$

the function $f(n)$ grows like $g(n)$



Θ notation

$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$
the function $f(n)$ grows like $g(n)$

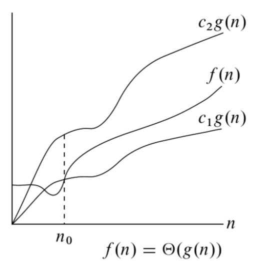


- there are two multiples c_1, c_2 of $g(n)$ such that

Θ notation

$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$

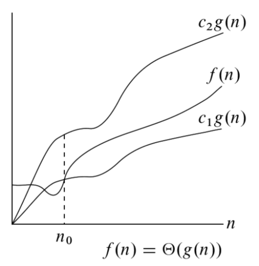
the function $f(n)$ grows like $g(n)$



- there are two multiples c_1, c_2 of $g(n)$ such that
- after a certain size n_0

Θ notation

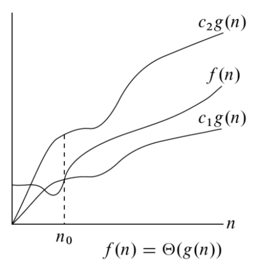
$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$
the function $f(n)$ grows like $g(n)$



- there are two multiples c_1, c_2 of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays between $c_1 g(n)$ and $c_2 g(n)$

Θ notation

$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$
the function $f(n)$ grows like $g(n)$

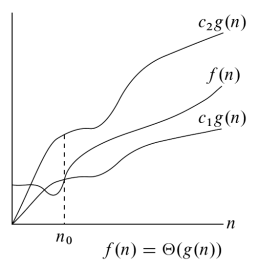


- there are two multiples c_1, c_2 of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays between $c_1 g(n)$ and $c_2 g(n)$

$f(n) = \Theta(g(n))$ means: there are costants c_1, c_2 and a number n_0 such that
for all $n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

Θ notation

$\Theta(g(n))$ is the combination of $O(g(n))$ and $\Omega(g(n))$
the function $f(n)$ grows like $g(n)$



- there are two multiples c_1, c_2 of $g(n)$ such that
- after a certain size n_0
- $f(n)$ stays between $c_1 g(n)$ and $c_2 g(n)$

$f(n) = \Theta(g(n))$ means: there are costants c_1, c_2 and a number n_0 such that
for all $n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

Formally we should write $f \in \Theta(g)$ where

$$\Theta(g) = O(g) \cap \Omega(g)$$

o and ω classes

There are also "small" versions of O and Ω where the inequalities hold strictly

o and ω classes

There are also "small" versions of O and Ω
where the inequalities hold strictly

$f(n) = o(g(n))$ means $f(n)$ grows slower than $g(n)$

o and ω classes

There are also "small" versions of O and Ω
where the inequalities hold strictly

$f(n) = o(g(n))$ means $f(n)$ grows slower than $g(n)$
for every costant $c > 0$ there exists a number n_0
such that for all $n \geq n_0, 0 \leq f(n) < cg(n)$

$$o(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

o and ω classes

There are also "small" versions of O and Ω
where the inequalities hold strictly

$f(n) = o(g(n))$ means $f(n)$ grows slower than $g(n)$
for every costant $c > 0$ there exists a number n_0
such that for all $n \geq n_0, 0 \leq f(n) < cg(n)$

$$o(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

$f(n) = \omega(g(n))$ means $f(n)$ grows faster than $g(n)$

o and ω classes

There are also "small" versions of O and Ω
where the inequalities hold strictly

$f(n) = o(g(n))$ means $f(n)$ grows slower than $g(n)$
for every costant $c > 0$ there exists a number n_0
such that for all $n \geq n_0, 0 \leq f(n) < cg(n)$

$$o(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq f(n) < cg(n)\}$$

$f(n) = \omega(g(n))$ means $f(n)$ grows faster than $g(n)$
for every costant $c > 0$ there exists a number n_0
such that for all $n \geq n_0, 0 \leq cg(n) < f(n)$

$$\omega(g) = \{f \mid \forall c, \exists n_0, \forall n \geq n_0, 0 \leq cg(n) < f(n)\}$$

Complexity of Maximum Subarray

Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0 ≥ s then (0,j0,s0)
                      else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                     else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

When the input has length $n = 1$, it is a singleton $[x]$

We immediately return the result $(0, 0, x)$

Constant time: just write the output

$$T(1) = c_0$$

Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                     else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

For inputs with length $n > 1$, we must compute

- `(sums xs)` linear time: traverses the input list
- `argMax snd ...` linear time: traverses the sums
- `maxSub (tail xs)` recursive call

$$T(n) = c_1 n + T(n-1)$$

Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                     else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

$$
\begin{aligned}
T(1) &= c_0 \\
T(n) &= c_1 n + T(n-1)
\end{aligned}
$$

Analysis of the Naive Algorithm

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = if s0≥s then (0,j0,s0)
                    else (i+1,j+1,s)
  where (j0,s0) = argMax snd (sums xs)
        (i,j,s) = maxSub (tail xs)
```

$$T(1) = c_0$$
$$T(n) = c_1 n + T(n-1)$$

So the complexity is:

$$\begin{aligned} T(n) &= c_1 n + T(n-1) = c_1 n + c_1(n-1) + T(n-2) \\ &= c_1 n + c_1(n-1) + \cdots c_1 2 + T(1) \\ &= c_1 n + c_1(n-1) + \cdots c_1 2 + c_0 \\ &= c_1 \sum_{i=2}^{i=n} i + c_0 = c_1(n(n+1)/2 - 1) + c_0 \\ &= \Theta(n^2) \end{aligned}$$

More Efficient Algorithms?

Exercise: Write a better algorithm for the Maximum Subarray Problem

Two ideas/strategies:

More Efficient Algorithms?

Exercise: Write a better algorithm for the Maximum Subarray Problem

Two ideas/strategies:

1. A divide-and-conquer algorithm:
   - Split the list in two halves
   - Compute separately the maximum subarray of both halves
   - Compute the maximum *cross-over* subarray

   This has complexity $O(n \log n)$

More Efficient Algorithms?

Exercise: Write a better algorithm for the Maximum Subarray Problem

Two ideas/strategies:

1. A divide-and-conquer algorithm:
   - Split the list in two halves
   - Compute separately the maximum subarray of both halves
   - Compute the maximum *cross-over* subarray

   This has complexity $O(n \log n)$
2. A linear time algorithm (see Ex. 4.1-5 in IA)
   - Traverse the list from left to right
   - Keep track of the maximum subarray seen so far
   - and the maximum subarray ending at the last seen element

   This has complexity $O(n)$

The Master Method

Advanced Algorithms and Data Structures - Lecture 2A

Venanzio Capretta

Monday 7 October 2019

School of Computer Science, University of Nottingham

Maximum Array - Divide and Conquer

Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

$$l = [4, -2, 3, -7, 5, 2, -3, 4, -8, 6, -2, 1]$$

Maximum Array - Divide and Conquer

Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

$$l_1 = [4, -2, 3, -7, 5, 2\ ][\ -3, 4, -8, 6, -2, 1] = l_2$$

- Split the input array in two halves

Maximum Array - Divide and Conquer

Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

$$l_1 = [4, -2, 3, -7, \underbrace{5, 2}_{\texttt{maxSub}\, l_1} \;][\; -3, 4, -8, \underbrace{6}_{\texttt{maxSub}\, l_2}, -2, 1] = l_2$$

- Split the input array in two halves
- Compute the maximum subarray of each half

Maximum Array - Divide and Conquer

Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

$$l_1 = [4, -2, 3, -7, \overbrace{\underbrace{5, 2}_{\texttt{maxSub}\ l_1} \;\;][\; -3, 4}^{\text{cross-over}}, -8, \underbrace{6}_{\texttt{maxSub}\ l_2}, -2, 1] = l_2$$

- Split the input array in two halves
- Compute the maximum subarray of each half
- Compute the maximum *cross-over* subarray

Maximum Array - Divide and Conquer

Back to the Maximum Array Problem.

We solve it in a recursive way, similar to Merge Sort:

$$l_1 = [4, -2, 3, -7, \overbrace{\underbrace{5, 2}_{\texttt{maxSub}\ l_1} \quad][\; -3, 4}^{\text{cross-over}}, -8, \underbrace{6}_{\texttt{maxSub}\ l_2}, -2, 1] = l_2$$

- Split the input array in two halves
- Compute the maximum subarray of each half
- Compute the maximum *cross-over* subarray

The result is the maximum of the three partial subproblems

Maximum Array DC in Haskell

```
maxSub :: [Int] → (Int,Int,Int)
maxSub [x] = (0,0,x)
maxSub xs = let mid = length xs `div` 2
                (xs1,xs2) = splitAt mid xs
                (i1,j1,max1) = maxSub xs1
                (i2,j2,max2) = maxSub xs2
                (i3,j3,max3) = maxCross xs1 xs2
            in if max1 ≥ max2 && max1 ≥ max3
               then (i1,j1,max1)
               else if max2 ≥ max3
                    then (i2+mid,j2+mid,max2)
                    else (i3,j3+mid,max3)
```

`maxCross` is an auxiliary functions that finds the maximum *crossover* sublist, with `i3` the start index in `xs1` and `j3` the end index in `xs2`

It has linear complexity in the sum of the lengths of `xs1` and `xs2`

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function `maxCross` has linear time complexity, $c_2 n$

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function `maxCross` has linear time complexity, $c_2 n$
- Determining the largest among the three partial results `max1`, `max2`, and `max3` and returning the corresponding output takes constant time d

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function `maxCross` has linear time complexity, $c_2 n$
- Determining the largest among the three partial results `max1`, `max2`, and `max3` and returning the corresponding output takes constant time d
- Finally the two recursive calls `maxSub xs1` and `maxSub xs2` will each take time $T(n/2)$ because `xs1` and `xs1` have half the size of `xs`

Recursive Equations for Time Complexity

Let's determine the time complexity $T(n)$ of this algorithm.

Singleton list ($n = 1$): return output in constant time: $T(1) = c_0$

For longer lists, the algorithm performs several steps:

- Splitting the list into two halves also takes linear time, $c_1 n$
- The auxiliary function `maxCross` has linear time complexity, $c_2 n$
- Determining the largest among the three partial results `max1`, `max2`, and `max3` and returning the corresponding output takes constant time d
- Finally the two recursive calls `maxSub xs1` and `maxSub xs2` will each take time $T(n/2)$ because `xs1` and `xs1` have half the size of `xs`

Putting all the components together we get (with $c = c_1 + c_2$):

$$T(n) = 2T(n/2) + c_1 n + c_2 n + d = 2T(n/2) + cn + d$$

Simplifying the Equations

Strictly speaking, if the length n of the list is not even, the splitting is not exact: we get a sublist of length $\lfloor n/2 \rfloor$ and one of length $\lceil n/2 \rceil$
The exact equation is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$$

But the approximation does not influence the resulting complexity class

Simplifying the Equations

Strictly speaking, if the length n of the list is not even, the splitting is not exact: we get a sublist of length $\lfloor n/2 \rfloor$ and one of length $\lceil n/2 \rceil$

The exact equation is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$$

But the approximation does not influence the resulting complexity class

The exact value of the constant factors is not relevant

The larger component dominates

So in $cn + d$ we just need to consider that this term is linear

Simplifying the Equations

Strictly speaking, if the length n of the list is not even, the splitting is not exact: we get a sublist of length $\lfloor n/2 \rfloor$ and one of length $\lceil n/2 \rceil$
The exact equation is

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn + d$$

But the approximation does not influence the resulting complexity class

The exact value of the constant factors is not relevant
The larger component dominates
So in $cn + d$ we just need to consider that this term is linear

We can rewrite the equation using complexity classes for the terms:

$$T(1) = \Theta(1)$$
$$T(n) = 2T(n/2) + \Theta(n)$$

Solving Recursive Equations

Three methods to solve a recursive equation:

- Substitution Method: make a guess on the complexity class, verify and derive the parameters by recursion
- Recursion Tree Method: Draw a tree with all the recursive calls of the function and add up all the steps in each node
- Master Method: A general theorem that gives you the complexity class depending on the form of the equation

Solving Recursive Equations

Three methods to solve a recursive equation:

- Substitution Method: make a guess on the complexity class, verify and derive the parameters by recursion
- Recursion Tree Method: Draw a tree with all the recursive calls of the function and add up all the steps in each node
- Master Method: A general theorem that gives you the complexity class depending on the form of the equation

Let's apply all three to the simplify system of equations

$$T(1) = 1$$
$$T(n) = 2T(n/2) + n$$

The solution will be the same as for the equations for the Maximum Subarray algorithm (and merge sort)

Substitution Method

Guess the solution:

Since it is the same equation as for merge sort, we guess that

$$T(n) = O(n \log n)$$

Substitution Method

Guess the solution:

Since it is the same equation as for merge sort, we guess that

$$T(n) = O(n \log n)$$

By the definition of O-notation, this means that

There exists a factor c and a starting size n_0 such that:

$$T(n) \leq cn \log n \qquad \text{for } n \geq n_0$$

Substitution Method

Guess the solution:

Since it is the same equation as for merge sort, we guess that

$$T(n) = O(n \log n)$$

By the definition of O-notation, this means that

There exists a factor c and a starting size n_0 such that:

$$T(n) \leq cn \log n \qquad \text{for } n \geq n_0$$

Let's check that this works for the inductive step:

Assume that it is true for values smaller than n

Prove that it also must hold for n:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2c\tfrac{n}{2}\log\tfrac{n}{2} + n \qquad \text{by Induction Hypothesis} \\ &= cn(\log n - \log 2) + n = cn(\log n - 1) + n \\ &= cn\log n - cn + n \leq cn \log n \qquad \text{if } c \geq 1 \end{aligned}$$

Substitution Method - Base Case

The base case is more problematic:
We have $T(1) = 1$, we can't prove $T(1) \leq c1 \log 1 = 0$

Substitution Method - Base Case

The base case is more problematic:
We have $T(1) = 1$, we can't prove $T(1) \leq c1 \log 1 = 0$

But we can choose any starting point n_0
For example

$$\begin{aligned} T(2) &= 2T(1) + 2 = 4 \\ &\leq c2 \log 2 = 2c \qquad \text{if } c \geq 2 \end{aligned}$$

Substitution Method - Base Case

The base case is more problematic:
We have $T(1) = 1$, we can't prove $T(1) \leq c1 \log 1 = 0$

But we can choose any starting point n_0
For example

$$\begin{aligned} T(2) &= 2T(1) + 2 = 4 \\ &\leq c2 \log 2 = 2c \qquad \text{if } c \geq 2 \end{aligned}$$

So everything works if we choose $n_0 = 2$ and $c = 2$

We proved that $T(n) = O(n \log n)$

(We've been a bit simplistic: $n/2$ is not guaranteed to be an integer.
Either assume that n is a power of two, or replace $n/2$ with $\lfloor n/2 \rfloor$)

Recursion Tree Method

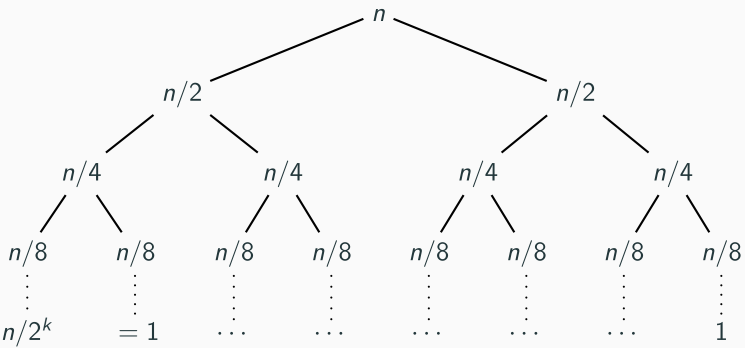
We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$ Children: two calls $T(n/2)$ And so on

Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

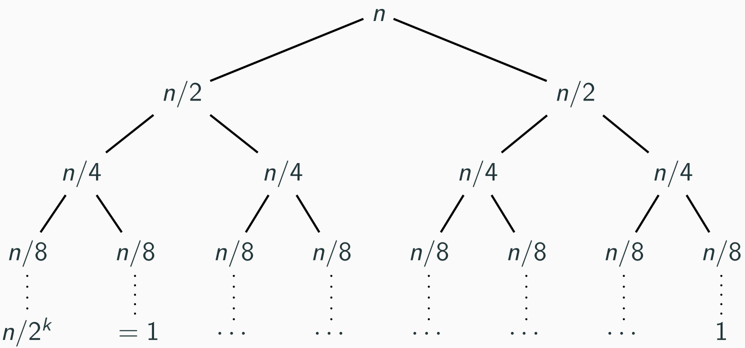
Root: $T(n)$ Children: two calls $T(n/2)$ And so on



Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$ Children: two calls $T(n/2)$ And so on

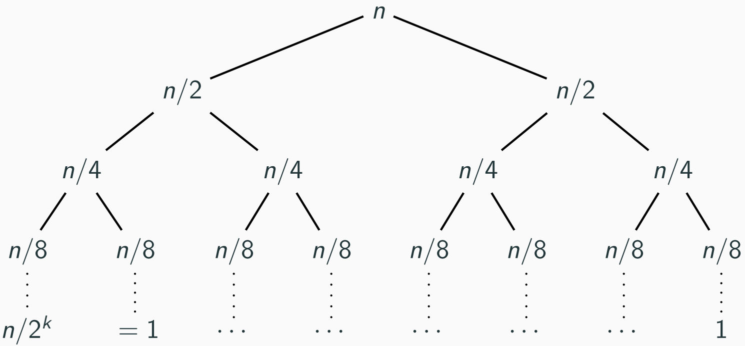


What is the depth k?

Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$ Children: two calls $T(n/2)$ And so on

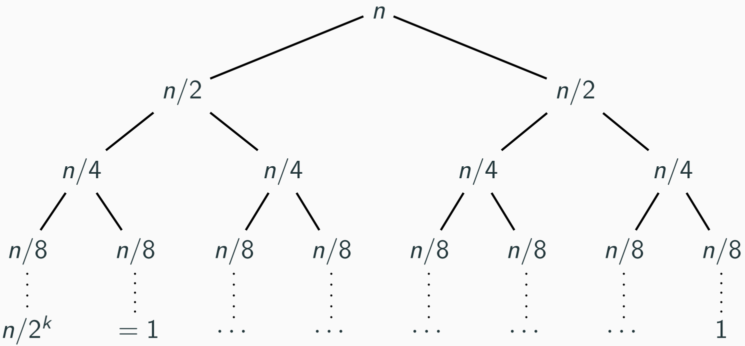


What is the depth k? $k = \log n$

Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$ Children: two calls $T(n/2)$ And so on



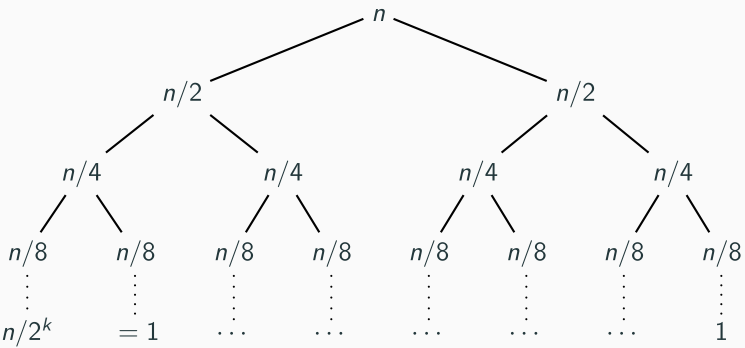
What is the depth k? $k = \log n$

How many computation steps do we do at each node?

Recursion Tree Method

We construct a tree of recursive calls, labelled with arguments

Root: $T(n)$ Children: two calls $T(n/2)$ And so on



What is the depth k? $k = \log n$

How many computation steps do we do at each node? At level j, $n/2^j$

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level j there are 2^j nodes with argument $n/2^j$

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level j there are 2^j nodes with argument $n/2^j$
- The recursive equation for those nodes gives

$$T(n/2^j) = 2\,T(n/2^{j+1}) + n/2^j$$

So the computation steps for each node is $n/2^j$

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level j there are 2^j nodes with argument $n/2^j$
- The recursive equation for those nodes gives

$$T(n/2^j) = 2T(n/2^{j+1}) + n/2^j$$

  So the computation steps for each node is $n/2^j$
- Adding up all the steps at level j we get: $2^j n/2^j = n$

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level j there are 2^j nodes with argument $n/2^j$
- The recursive equation for those nodes gives

$$T(n/2^j) = 2T(n/2^{j+1}) + n/2^j$$

  So the computation steps for each node is $n/2^j$
- Adding up all the steps at level j we get: $2^j n/2^j = n$

So there are a total of n computation steps at each level
and there are $\log n$ levels
Total number of steps: $n \log n$

Recursion Tree Method, Calculation

Let's sum up all the computation steps:

- There are $k = \log n$ levels in the tree
- At each level j there are 2^j nodes with argument $n/2^j$
- The recursive equation for those nodes gives

$$T(n/2^j) = 2\,T(n/2^{j+1}) + n/2^j$$

  So the computation steps for each node is $n/2^j$
- Adding up all the steps at level j we get: $2^j n/2^j = n$

So there are a total of n computation steps at each level
and there are $\log n$ levels
Total number of steps: $n \log n$

This shows that $T(n) = \Theta(n \log n)$

The Master Method

The Master Method generalizes the recursion tree techniques
to algorithms with different number of recursive calls
with different sizes of argumens

The Master Method

The Master Method generalizes the recursion tree techniques
to algorithms with different number of recursive calls
with different sizes of argumens

The Maximum Subarray algorithm (and Merge Sort) had:

- Two recursive calls
- Each with an argument of half size, $n/2$
- A linear non-recursive part

This leads to the equation: $T(n) = 2T(n/2) + cn$

The Master Method

The Master Method generalizes the recursion tree techniques
to algorithms with different number of recursive calls
with different sizes of argumens

The Maximum Subarray algorithm (and Merge Sort) had:

- Two recursive calls
- Each with an argument of half size, $n/2$
- A linear non-recursive part

This leads to the equation: $T(n) = 2T(n/2) + cn$

A more general recursive program could have:

- Any number (a) of recursive calls
- Each with an argument of size n/b
- A non-recursive part given by a function $f(n)$

This leads to the equation $T(n) = aT(n/b) + f(n)$

Master Method: Recursion Trees

If we draw the recursion tree:

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j:

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree:

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree: $\log_b n$

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree: $\log_b n$

What is the total number of nodes?

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree: $\log_b n$

What is the total number of nodes?

- 1 node at level 0 (root)

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree: $\log_b n$

What is the total number of nodes?

- 1 node at level 0 (root)
- a nodes at level 1

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree: $\log_b n$

What is the total number of nodes?

- 1 node at level 0 (root)
- a nodes at level 1
- a^2 nodes at level 2

Master Method: Recursion Trees

If we draw the recursion tree:

- Number of children for each node: a
- Arguments at level j: n/b^j
- Depth of tree: $\log_b n$

What is the total number of nodes?

- 1 node at level 0 (root)
- a nodes at level 1
- a^2 nodes at level 2
- a^j nodes at level j

There are $k = \log_b n$ levels, total number of nodes:

$$1 + a + a^2 + a^3 + \cdots a^{\log_b n}$$

This is a geometric series (see IA Appendix A)

Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^k = \frac{a^{k+1} - 1}{a - 1}$$

Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^k = \frac{a^{k+1} - 1}{a - 1} = \Theta(a^k) = \Theta(a^{\log_b n})$$

Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^k = \frac{a^{k+1}-1}{a-1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$$

Compare with the non-recursive part $f(n)$:

Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^k = \frac{a^{k+1}-1}{a-1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$$

Compare with the non-recursive part $f(n)$:

- If the non-recursive part grows slower than the number of nodes:

$$f(n) = O(n^{\log_b a - \epsilon}) \qquad \text{for some } \epsilon > 0$$

the recursive part dominates: $T(n) = \Theta(n^{\log_b a})$

Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^k = \frac{a^{k+1}-1}{a-1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$$

Compare with the non-recursive part $f(n)$:

- If the non-recursive part grows slower than the number of nodes:

$$f(n) = O(n^{\log_b a - \epsilon}) \qquad \text{for some } \epsilon > 0$$

  the recursive part dominates: $T(n) = \Theta(n^{\log_b a})$
- If they are of the same class: $f(n) = \Theta(n^{\log_b a})$
  each level adds $n^{\log_b a}$ computation steps (check the math)
  There are $\log_a n$ levels, so: $T(n) = \Theta(n^{\log_b a} \log n)$

Master Method: Computation Steps

Total number of nodes:

$$\sum_{j=0}^{j=k} a^k = \frac{a^{k+1}-1}{a-1} = \Theta(a^k) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$$

Compare with the non-recursive part $f(n)$:

- If the non-recursive part grows slower than the number of nodes:

$$f(n) = O(n^{\log_b a - \epsilon}) \qquad \text{for some } \epsilon > 0$$

  the recursive part dominates: $T(n) = \Theta(n^{\log_b a})$

- If they are of the same class: $f(n) = \Theta(n^{\log_b a})$
  each level adds $n^{\log_b a}$ computation steps (check the math)
  There are $\log_a n$ levels, so: $T(n) = \Theta(n^{\log_b a} \log n)$
- If the non-recursive part grows faster than the number of nodes:

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \qquad \text{for some } \epsilon > 0$$

  (plus some other condition)
  the non-recursive part dominates: $T(n) = \Theta(f(n))$

Application of the MM to Maximum Array

In the case of the Maximum Array algorithm (and Merge Sort):

$$T(1) = c_0$$
$$T(n) = 2T(n/2) + c_1 n + c_2$$

Application of the MM to Maximum Array

In the case of the Maximum Array algorithm (and Merge Sort):

$$T(1) = c_0$$
$$T(n) = 2T(n/2) + c_1 n + c_2$$

We have $a = 2$, $b = 2$, $f(n) = c_1 n + c_2$

Application of the MM to Maximum Array

In the case of the Maximum Array algorithm (and Merge Sort):

$$T(1) = c_0$$
$$T(n) = 2T(n/2) + c_1 n + c_2$$

We have $a = 2$, $b = 2$, $f(n) = c_1 n + c_2$

We must compare $f(n)$ with $n^{\log_b a} = n^{\log_2 2} = n$

We have $f(n) = \Theta(n)$, so we're in the second case

Application of the MM to Maximum Array

In the case of the Maximum Array algorithm (and Merge Sort):

$$T(1) = c_0$$
$$T(n) = 2T(n/2) + c_1 n + c_2$$

We have $a = 2,\ b = 2,\ f(n) = c_1 n + c_2$

We must compare $f(n)$ with $n^{\log_b a} = n^{\log_2 2} = n$

We have $f(n) = \Theta(n)$, so we're in the second case

Conclusion $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$

Binary Search Trees

Advanced Algorithms and Data Structures - Lecture 2B

Venanzio Capretta

Monday 7 October 2019

School of Computer Science, University of Nottingham

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$
with algorithms to perform these operations:

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$ with algorithms to perform these operations:

- Search a key in the collection

$$\texttt{search}\ 4\ \{7, 5, 1, 10, 4, 6, 9\} = \text{true}$$
$$\texttt{search}\ 3\ \{7, 5, 1, 10, 4, 6, 9\} = \text{false}$$

  In concrete applications, every number/key would be associated with a value and the search function would return that value

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$ with algorithms to perform these operations:

- Search a key in the collection

  $$\texttt{search}\ 4\ \{7, 5, 1, 10, 4, 6, 9\} = \text{true}$$
  $$\texttt{search}\ 3\ \{7, 5, 1, 10, 4, 6, 9\} = \text{false}$$

  In concrete applications, every number/key would be associated with a value and the search function would return that value

- Insert an new element in the collection

  $$\texttt{insert}\ 3\ \{7, 5, 1, 10, 4, 6, 9\} = \{7, 5, 1, 10, 4, 6, 9, 3\}$$

  (where it is inserted depends on the representation)

Dynamic Sets

A dynamic set is a representation of a collection of elements (keys) from an ordered set, example: $\{7, 5, 1, 10, 4, 6, 9\}$ with algorithms to perform these operations:

- Search a key in the collection

$$\texttt{search}\ 4\ \{7, 5, 1, 10, 4, 6, 9\} = \text{true}$$
$$\texttt{search}\ 3\ \{7, 5, 1, 10, 4, 6, 9\} = \text{false}$$

  In concrete applications, every number/key would be associated with a value and the search function would return that value

- Insert an new element in the collection

$$\texttt{insert}\ 3\ \{7, 5, 1, 10, 4, 6, 9\} = \{7, 5, 1, 10, 4, 6, 9, 3\}$$

  (where it is inserted depends on the representation)

- Delete an element from the collection

$$\texttt{delete}\ 4\ \{7, 5, 1, 10, 4, 6, 9\} = \{7, 5, 1, 10, 6, 9\}$$

Dictionaries

In practice elements of a dynamic sets will be pairs:

A key used for searching, a value to be returned

Such a dynamic set is also called a dictionary

Example: In a database of students, the key could be the ID number, the value the name of the student (and all other relevant data)

$$\{(7, \text{Monica}), (5, \text{Richard}), (1, \text{Fang}), (10, \text{Wei}),$$
$$(4, \text{Jan}), (6, \text{Femke}), (9, \text{Clara})\}$$

Dictionaries

In practice elements of a dynamic sets will be pairs:

A key used for searching, a value to be returned

Such a dynamic set is also called a dictionary

Example: In a database of students, the key could be the ID number, the value the name of the student (and all other relevant data)

$$\{(7, \text{Monica}), (5, \text{Richard}), (1, \text{Fang}), (10, \text{Wei}),$$
$$(4, \text{Jan}), (6, \text{Femke}), (9, \text{Clara})\}$$

Searching the data base with a key returns the value:

$$\texttt{search}\, 6\, \{(7, \text{Monica}), \ldots, (9, \text{Clara})\} = \text{Femke}$$

Dictionaries

In practice elements of a dynamic sets will be pairs:
A key used for searching, a value to be returned
Such a dynamic set is also called a dictionary

Example: In a database of students, the key could be the ID number, the value the name of the student (and all other relevant data)

$$\{(7, \text{Monica}), (5, \text{Richard}), (1, \text{Fang}), (10, \text{Wei}), \\ (4, \text{Jan}), (6, \text{Femke}), (9, \text{Clara})\}$$

Searching the data base with a key returns the value:

$$\texttt{search}\, 6\, \{(7, \text{Monica}), \ldots, (9, \text{Clara})\} = \text{Femke}$$

For the study of the algorithms just the keys are relevant
I will describe the algorithms just using a set of keys
Exercise: Extend them to include the values

Implementations

Possible representations are:

Implementations

Possible representations are:

- Unordered List: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list

Implementations

Possible representations are:

- UNORDERED LIST: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list
- ORDERED LIST: $[1, 4, 5, 6, 7, 9, 10]$
  Searching still takes $O(n)$ time
  Just a bit better: I may stop early

Implementations

Possible representations are:

- UNORDERED LIST: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list
- ORDERED LIST: $[1, 4, 5, 6, 7, 9, 10]$
  Searching still takes $O(n)$ time
  Just a bit better: I may stop early
- RANDOM-ACCESS ORDERED ARRAY
  Fast search in $O(\log n)$ time
  But insertion and deletion still take $O(n)$ time
  Resizing of the array and shifting elements is necessary and costly

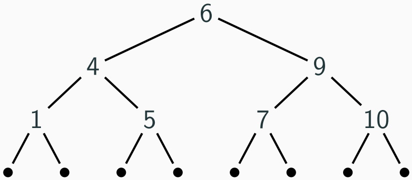
Implementations

Possible representations are:

- Unordered List: $[7, 5, 1, 10, 4, 6, 9]$
  Searching takes $O(n)$ time
  We must scan the whole list
- Ordered List: $[1, 4, 5, 6, 7, 9, 10]$
  Searching still takes $O(n)$ time
  Just a bit better: I may stop early
- Random-Access Ordered Array
  Fast search in $O(\log n)$ time
  But insertion and deletion still take $O(n)$ time
  Resizing of the array and shifting elements is necessary and costly

Can we find a data structure for which all three operations are efficient?

Binary Search Trees



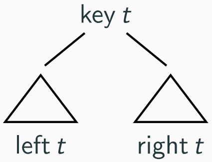
The operations of search, insert, delete
can be done in $O(k)$ time
where k is the depth of the tree

But tricky to keep k small: $k \sim O(\log n)$

There are more advanced variants that guarantee this: Red-Black Trees

The BST Property

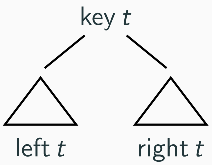
For every tree t, we use the notation:



(key t)	element in the root node
(left t)	left subtree of t
(right t)	right subtree of t

The BST Property

For every tree t, we use the notation:



(key t)	element in the root node
(left t)	left subtree of t
(right t)	right subtree of t

The defining property of Binary Search Trees is:

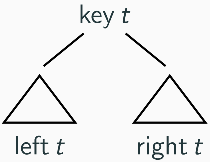
All elements in (left t) are smaller than (key t) and

All elements in (right t) are lareger than (key t)

(We assume that there are no repeated keys)

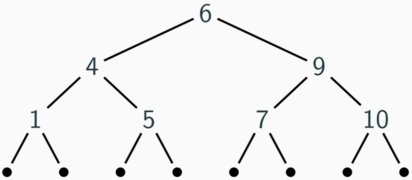
The BST Property

For every tree t, we use the notation:



(key t)	element in the root node
(left t)	left subtree of t
(right t)	right subtree of t

The defining property of Binary Search Trees is:
All elements in (left t) are smaller than (key t) and
All elements in (right t) are lareger than (key t)
(We assume that there are no repeated keys)



$4, 1, 5 < 6$
$9, 7, 10 > 6$
recursively:
$1 < 4, 5 > 4$
$7 < 9, 10 > 9$

Trees with key-value pairs

In practical applications:
nodes will contain pairs $\langle k, v \rangle$ of
a key k and a value v

The key is used for searching
The value is the information we want to store

Trees with key-value pairs

In practical applications:
nodes will contain pairs $\langle k, v \rangle$ of
a key k and a value v

The key is used for searching
The value is the information we want to store

For example

- In a dictionary: keys = words; values = definitions
- In an address book: keys = names; values = addresses

Trees with key-value pairs

In practical applications:
nodes will contain pairs $\langle k, v \rangle$ of
a key k and a value v

The key is used for searching
The value is the information we want to store

For example

- In a dictionary: keys = words; values = definitions
- In an address book: keys = names; values = addresses

For the sake of the definition of the algorithms, we only use keys
Exercise: modify the algorithms with key-value pairs

The type of BSTs

BST is a type with two constructors:

The type of BSTs

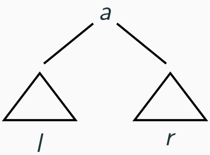
BST is a type with two constructors:

- Nil for leaves / empty trees

The type of BSTs

BST is a type with two constructors:

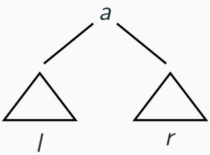
- Nil for leaves / empty trees
- Node for internal nodes
  containing elements from an ordered type Key
  (Node $a\ l\ r$) is the tree



The type of BSTs

BST is a type with two constructors:

- Nil for leaves / empty trees
- Node for internal nodes
  containing elements from an ordered type Key
  (Node a l r) is the tree



In functional programming it is defined as this inductive data type

```
data BST = Nil | Node Key BST BST
```

Component Functions

Functions to give the components
Defined by pattern-matching

Component Functions

Functions to give the components

Defined by pattern-matching

```
key :: BST → Key
key Nil = undefined
key (Node a l r) = a
```

Component Functions

Functions to give the components

Defined by pattern-matching

```
key :: BST → Key
key Nil = undefined
key (Node a l r) = a
```

```
left :: BST → BST
left Nil = undefined
left (Node a l r) = l
```

Component Functions

Functions to give the components

Defined by pattern-matching

```
key :: BST → Key
key Nil = undefined
key (Node a l r) = a
```

```
left :: BST → BST
left Nil = undefined
left (Node a l r) = l
```

```
right :: BST → BST
right Nil = undefined
right (Node a l r) = r
```

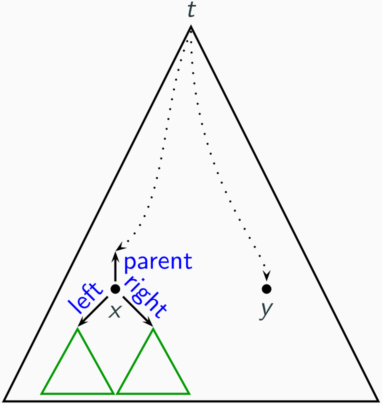
Imperative Implementation

In imperative programming, we use pointers to nodes, with methods giving the two children and the parent (See Ch.12 of IA)

(We can do it in functional programming too: using paths or defining them directly with those fields (Exercise))

Global Objects, Local Pointers

Work with a global tree t and with pointers x, y to subtrees/nodes



We can move around the tree with operations on pointers

- (parent x) the immediate precursor of x; Nil if x is the root
- (left x) and (right x) the children of x; Nil if they are leaves

Searching

Searching a tree t for a key k is done by following a single path
At each node x:

- If $k = \mathsf{key}\, x$, we have found it!
- If $k < \mathsf{key}\, x$, go to the left child of x;
- If $k > \mathsf{key}\, x$, go to the right child of x.

Searching

Searching a tree t for a key k is done by following a single path
At each node x:

- If $k = \text{key}\, x$, we have found it!
- If $k < \text{key}\, x$, go to the left child of x;
- If $k > \text{key}\, x$, go to the right child of x.

Functional/Recursive Version:

```
search :: Key → BST → Bool
search k Nil = False
search k (Node x l r) =
     (k == x) || if (k < x) then search k l
                            else search k r
```

Imperative/Iterative Version

```
search (t,k):
  x := t
  while x /= Nil and k /= (key x)
    if k < (key x) then x := (left x)
                   else x := (right x)
  return k == (key x)
```

Imperative/Iterative Version

```
search (t,k):
  x := t
  while x /= Nil and k /= (key x)
    if k < (key x) then x := (left x)
                   else x := (right x)
  return k == (key x)
```

This algorithm just returns a Boolean value:
true if k is present in the tree, false otherwise

Exercise: Modify the algorithm for trees containing key-values pairs;
if the key is found, it must return the corresponding value.

Imperative/Iterative Version

```
search (t,k):
  x := t
  while x /= Nil and k /= (key x)
    if k < (key x) then x := (left x)
                    else x := (right x)
  return k == (key x)
```

This algorithm just returns a Boolean value:
true if k is present in the tree, false otherwise

Exercise: Modify the algorithm for trees containing key-values pairs; if the key is found, it must return the corresponding value.

Complexity: The search algorithm starts at the root and follows a specific path, until it reaches a node that matches the search key or a leaf. The time complexity is $O(h)$ where h is the height of the tree.

In-Order Traversal

Given a BST, generate a list containing all its elements in order

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller thant the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller thant the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

```
treeList :: BST → [Key]
treeList Nil = []
treeList (Node x l r) = treeList l ++ x : treeList r
```

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller thant the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

```
treeList :: BST → [Key]
treeList Nil = []
treeList (Node x l r) = treeList l ++ x : treeList r
```

Complexity: The algorithm visits every node just once.
It runs in $O(n)$ time, where n is the number of elements.

In-Order Traversal

Given a BST, generate a list containing all its elements in order

Since the elements in the left child are smaller thant the node key and the elements in the right child are larger:

- First output (recursively) all elements of the left child
- Then output the node key
- Finally output (recursively) all elements of the right child

```
treeList :: BST → [Key]
treeList Nil = []
treeList (Node x l r) = treeList l ++ x : treeList r
```

Complexity: The algorithm visits every node just once.
It runs in $O(n)$ time, where n is the number of elements.

Exercise: Implement the `insert` operation
Hint: Insert in a leaf in the correct position

Minimum and Maximum

The minimum element in a binary search tree is the leftmost one, the maximum is the rightmost

```
minimum :: BST → Maybe Key
minimum Nil = Nothing
minimum (Node x Nil _) = Just x
minimum (Node x l _) = minimum l
```

Minimum and Maximum

The minimum element in a binary search tree is the leftmost one, the maximum is the rightmost

```
minimum :: BST → Maybe Key
minimum Nil = Nothing
minimum (Node x Nil _) = Just x
minimum (Node x l _) = minimum l
```

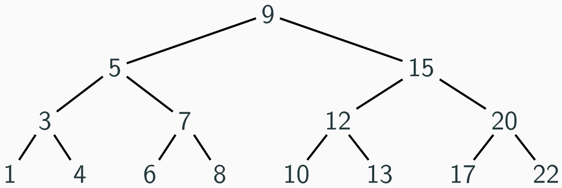
In iterative style:

```
minimum (x):
  while (left x) /= Nil
    x := (left x)
  return (key x)
```

The maximum is defined similarly, going right instead of left.

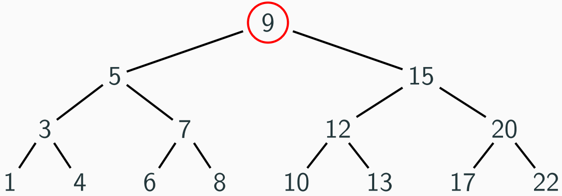
Complexity: $O(h)$ where h is the height of the tree.

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



Delete

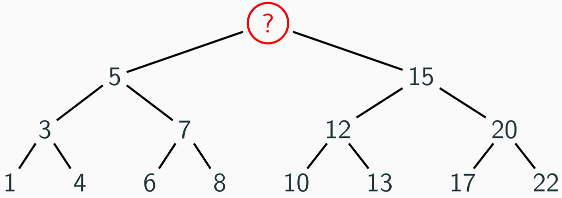
When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



To delete the root 9:

Delete

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property

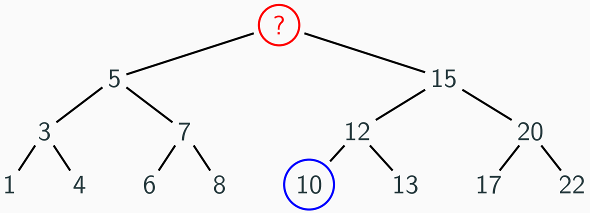


To delete the root 9:

- Remove the root

Delete

When we delete an element, we search for it, extract it and replace it with another element that preserves the BST property



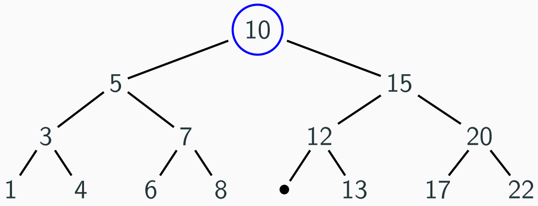
To delete the root 9:

- Remove the root
- Find the minimum of the right child

(We could also do it with the maximum of the left child)

Delete

When we delete an element, we search for it, extract it
and replace it with another element that preserves the BST property



To delete the root 9:

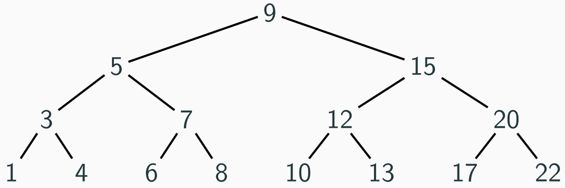
- Remove the root
- Find the minimum of the right child
- Place it at the root

(We could also do it with the maximum of the left child)

Predecessor and Successor

Extra Operations:

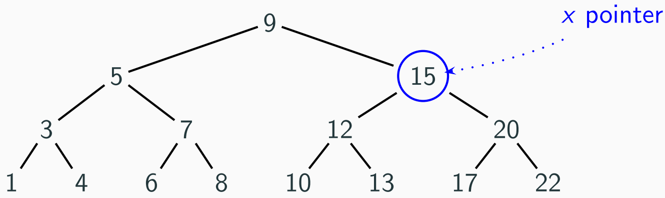
Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



Predecessor and Successor

Extra Operations:

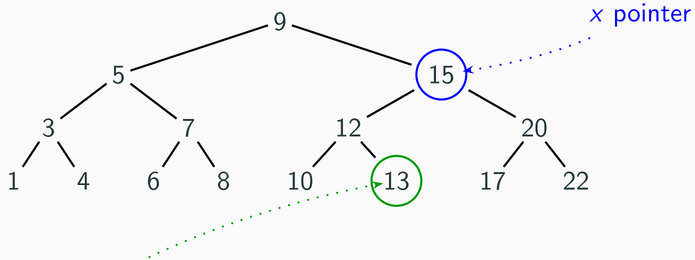
Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



Predecessor and Successor

Extra Operations:

Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.

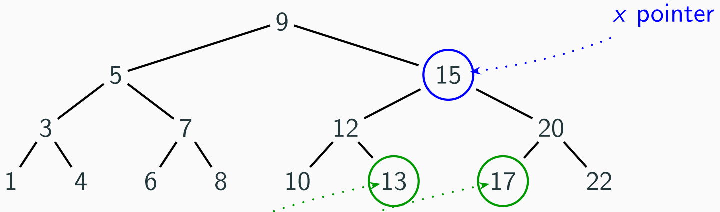


- The predecessor is the maximum of the left subtree of x;

Predecessor and Successor

Extra Operations:

Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.

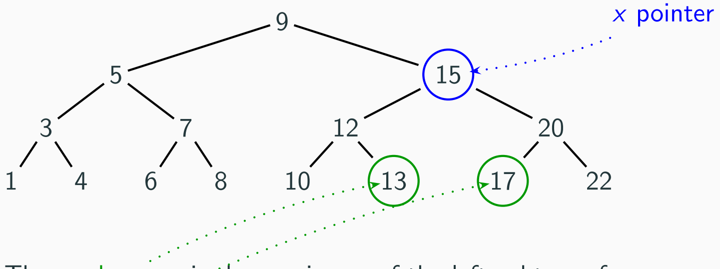


- The predecessor is the maximum of the left subtree of x;
- The successor is the minimum of the right subtree of x;

Predecessor and Successor

Extra Operations:

Find the element in the tree that is immediately lower (or higher) than the one at the given pointer.



- The predecessor is the maximum of the left subtree of x;
- The successor is the minimum of the right subtree of x;

Exercise: What if the node doesn't have a left child (or a right child)?

Red-Black Trees

Advanced Algorithms and Data Structures - Lecture 3

Venanzio Capretta

Monday 14 October 2019

School of Computer Science, University of Nottingham

BST: Keeping the Balance

Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where h is the height of the tree

If the tree is balanced, $h = O(\log n)$ where n is the number of elements

BST: Keeping the Balance

Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where h is the height of the tree

If the tree is balanced, $h = O(\log n)$ where n is the number of elements

However, there is no guarantee that the tree is balanced
The operations `insert` and `delete` can change the balance of the tree

BST: Keeping the Balance

Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where h is the height of the tree

If the tree is balanced, $h = O(\log n)$ where n is the number of elements

However, there is no guarantee that the tree is balanced
The operations `insert` and `delete` can change the balance of the tree

For an efficient representation we must
implement `insert` and `delete` so that they preserve the balance
Not easy

There are several ways to do it

BST: Keeping the Balance

Binary Search Trees implement the basic operations of Dynamic Sets in $O(h)$ time, where h is the height of the tree

If the tree is balanced, $h = O(\log n)$ where n is the number of elements

However, there is no guarantee that the tree is balanced
The operations `insert` and `delete` can change the balance of the tree

For an efficient representation we must
implement `insert` and `delete` so that they preserve the balance
Not easy

There are several ways to do it

Red-Black Trees:

- Not perfect balance
- Some paths may be twice as long as others
- Still guarantees that the height is $O(\log n)$

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satifying these properties:

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satifying these properties:

1. Every node contains and extra color value: **Red** or **Black**
   (basically a Boolean value)

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satifying these properties:

1. Every node contains and extra color value: **Red** or **Black** (basically a Boolean value)
2. The root (and leaves) are black

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satifying these properties:

1. Every node contains and extra color value: **Red** or **Black**
   (basically a Boolean value)
2. The root (and leaves) are black
3. The children of a red node are black
   (no consecutive red nodes)

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

A Red-Black Tree is a BST satifying these properties:

1. Every node contains and extra color value: **Red** or **Black**
   (basically a Boolean value)
2. The root (and leaves) are black
3. The children of a red node are black
   (no consecutive red nodes)
4. For each node, every path from it to a leaf has the same number of black nodes

Red-Black Trees Definition

Idea:

- Color the nodes of a BST either **Red** or **Black**
- When computing the height, only count black nodes
- Keep the number of red nodes low: no consecutive red nodes

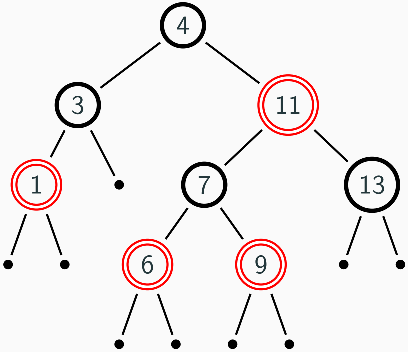
A Red-Black Tree is a BST satifying these properties:

1. Every node contains and extra color value: **Red** or **Black**
   (basically a Boolean value)
2. The root (and leaves) are black
3. The children of a red node are black
   (no consecutive red nodes)
4. For each node, every path from it to a leaf has the same number of black nodes

Black-height of a node:
The number of black nodes in any path from the node to any leaf

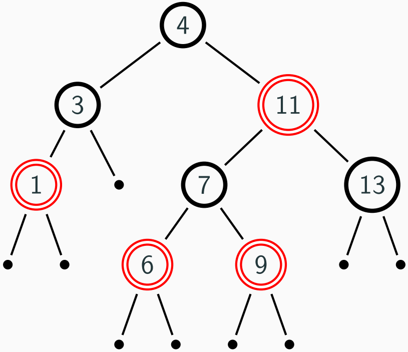
Example of Red-Black Tree

Example (red nodes have double circles)



Example of Red-Black Tree

Example (red nodes have double circles)

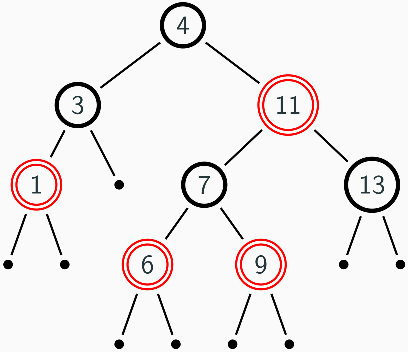


All paths from root to a leaf contain two black nodes: black-height $= 2$

- Shortest paths: only black nodes, eg: $4, 3, \cdot$
- Longest paths: alternating black and red , eg: $4, 11, 7, 9, \cdot$

Example of Red-Black Tree

Example (red nodes have double circles)



All paths from root to a leaf contain two black nodes: black-height $= 2$

- Shortest paths: only black nodes, eg: $4, 3, \cdot$
- Longest paths: alternating black and red , eg: $4, 11, 7, 9, \cdot$

Longest paths at most twice as long as shortest

Definition of Data Type

Definition of the type of Red-Black trees in Haskell

Similar to Binary Search Trees, with extra field for color

```
data Color = Red | Black
data RBTree = Leaf | Node Color RBTree Key RBTree
```

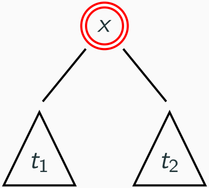
Definition of Data Type

Definition of the type of Red-Black trees in Haskell

Similar to Binary Search Trees, with extra field for color

```
data Color = Red | Black
data RBTree = Leaf | Node Color RBTree Key RBTree
```

The term `(Node Red t1 x t2)` represents the tree



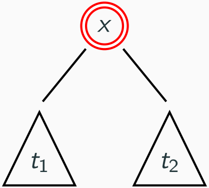
Definition of Data Type

Definition of the type of Red-Black trees in Haskell

Similar to Binary Search Trees, with extra field for color

```
data Color = Red | Black
data RBTree = Leaf | Node Color RBTree Key RBTree
```

The term `(Node Red t1 x t2)` represents the tree



Type definition does not guarantee elements are correct Red-Black trees

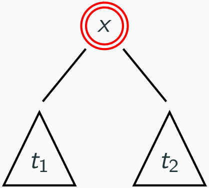
Definition of Data Type

Definition of the type of Red-Black trees in Haskell

Similar to Binary Search Trees, with extra field for color

```
data Color = Red | Black
data RBTree = Leaf | Node Color RBTree Key RBTree
```

The term `(Node Red t1 x t2)` represents the tree



Type definition does not guarantee elements are correct Red-Black trees

Ensure that the properties are satisfied when you create and modify trees:

The element must be a correct Binary Search Tree and

It must satisfy the extra Red-Black properties

Height of Trees

If a R-B tree contains n elements, then
the maximum length of a path is $2\log(n+1)$

Even if the tree is not perfectly balanced, its height is $O(\log n)$

Therefore the running time for searching is $O(\log n)$

Height of Trees

If a R-B tree contains n elements, then
the maximum length of a path is $2\log(n+1)$

Even if the tree is not perfectly balanced, its height is $O(\log n)$

Therefore the running time for searching is $O(\log n)$

But we must modify the insertion and deletion algorithms
to preserve the R-B properties

Height of Trees

If a R-B tree contains n elements, then
the maximum length of a path is $2\log(n+1)$

Even if the tree is not perfectly balanced, its height is $O(\log n)$

Therefore the running time for searching is $O(\log n)$

But we must modify the insertion and deletion algorithms
to preserve the R-B properties

Idea:

- Insert and delete as for regular binary search trees
- Always color new nodes red when you insert
- Rotate and recolor to restore the properties

Height of Trees

If a R-B tree contains n elements, then
the maximum length of a path is $2\log(n+1)$

Even if the tree is not perfectly balanced, its height is $O(\log n)$

Therefore the running time for searching is $O(\log n)$

But we must modify the insertion and deletion algorithms
to preserve the R-B properties

Idea:

- Insert and delete as for regular binary search trees
- Always color new nodes red when you insert
- Rotate and recolor to restore the properties

We define an auxiliary function `balance` that rotates a tree when there are two consecutive red nodes in one of its children

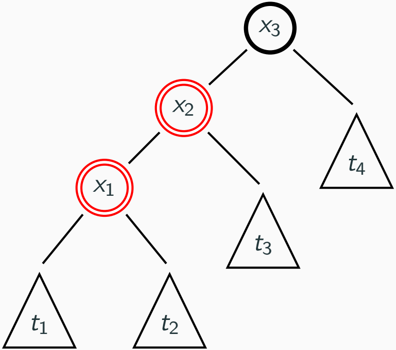
Balance Rotation I

Assume that the top node is **black**,

but there are two consecutive red nodes under it

There are four cases, according to the position of the red nodes

First Case:



BST property: $t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4$

Rotate and Change Colors

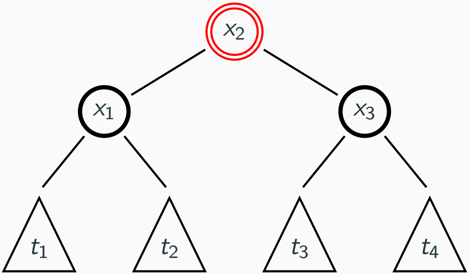
Balance Rotation I

Assume that the top node is **black**,

but there are two consecutive red nodes under it

There are four cases, according to the position of the red nodes

First Case:



BST property: $t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4$

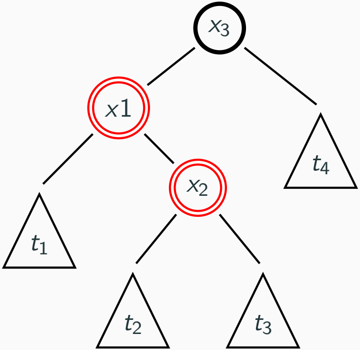
The black-height of every node remains the same

No consecutive red nodes any more

(but there may be above if the parent is red)

Balance Rotation II

Second Case:

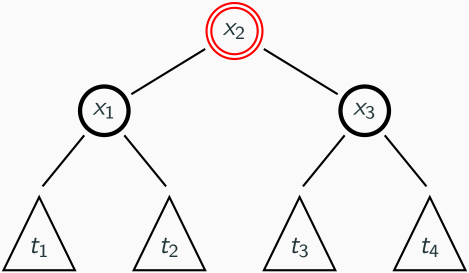


BST property: $t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4$

Rotate and Change Colors

Balance Rotation II

Second Case:



BST property: $t_1 < x_1 < t_2 < x_2 < t_3 < x_3 < t_4$

If the consecutive red nodes are in the right child

rotate symmetrically in the other direction

Balance Rotation III

Haskell program that fixes one double occurrence of red nodes:
It receives the input tree already divided into
color, left-child, key, right-child

```
balance :: Color → RBTree → Key → RBTree → RBTree
balance Black (Node Red (Node Red t1 x1 t2) x2 t3) x3 t4
  = NodeRB Red (Node Black t1 x1 t2) x2 (Node Black t3 x3 t4)

...

balance Black t1 x1 (Node Red t2 x2 (Node Red t3 x3 t4))
  = Node Red (Node Black t1 x1 t2) x2 (Node Black t3 x3 t4)
```

Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply `balance` to fix consecutive reds
- At the end, if the root is red, make it black

Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply `balance` to fix consecutive reds
- At the end, if the root is red, make it black

```
ins :: Key → RBTree → RBTree
ins a Leaf = Node Red Leaf a Leaf
ins a t@(Node color t1 x t2)
          | a < x = balance color (ins a t1) x t2
          | a > x = balance color t1 x (ins a t2)
          | otherwise = t
```

Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply `balance` to fix consecutive reds
- At the end, if the root is red, make it black

```
ins :: Key → RBTree → RBTree
ins a Leaf = Node Red Leaf a Leaf
ins a t@(Node color t1 x t2)
          | a < x = balance color (ins a t1) x t2
          | a > x = balance color t1 x (ins a t2)
          | otherwise = t
```

The result satisfies all the R-B properties

Except: Its root could be red

Insertion

Insert a new element into a R-B tree by:

- Insert in place of a leaf as in BSTs
- Initially color the new node red
- Recursively apply `balance` to fix consecutive reds
- At the end, if the root is red, make it black

```
ins :: Key → RBTree → RBTree
ins a Leaf = Node Red Leaf a Leaf
ins a t@(Node color t1 x t2)
          | a < x = balance color (ins a t1) x t2
          | a > x = balance color t1 x (ins a t2)
          | otherwise = t
```

The result satisfies all the R-B properties

Except: Its root could be red - just paint it black:

```
insert :: Key → RBTree → RBTree
insert a tree = blackRoot (ins a tree)
```

Insert Observations

Let's say that a tree is weakly R-B if it satisfies all the R-B properties except that its root may be red and one of its children may also be red (so there could be two consecutive red nodes at the top.

Insert Observations

Let's say that a tree is weakly R-B if it satisfies all the R-B properties except that its root may be red and one of its children may also be red (so there could be two consecutive red nodes at the top.

Observation:

- If `t` is a weakly R-B tree, then also `(ins a t)` is a weakly R-B tree

Insert Observations

Let's say that a tree is weakly R-B if it satisfies all the R-B properties except that its root may be red and one of its children may also be red (so there could be two consecutive red nodes at the top.

Observation:

- If `t` is a weakly R-B tree, then also `(ins a t)` is a weakly R-B tree
- If `t` is a weakly R-B tree, then we can turn it into a fully R-B tree by painting its root black

This will increase the black-height by one,
but since we do it at the root,
all paths will increase their black-lenghts equally.

Complexity of Insert

The function `balance` only rearranges the two two levels of the tree,
So it runs in constant time

Complexity of Insert

The function `balance` only rearranges the two two levels of the tree,
So it runs in constant time

So `insert` still runs in $O(h)$ time
where h is the height of the tree

Complexity of Insert

The function `balance` only rearranges the two two levels of the tree,
So it runs in constant time

So `insert` still runs in $O(h)$ time
where h is the height of the tree

The height of a R-B tree is $h = O(\log n)$

Complexity of Insert

The function `balance` only rearranges the two two levels of the tree,
So it runs in constant time

So `insert` still runs in $O(h)$ time
where h is the height of the tree

The height of a R-B tree is $h = O(\log n)$

So `insert` runs in $O(\log n)$ time

Deletion

Deleting an element is a bit more complicated than inserting it

Deletion may cause a subtree to decrese its black-height

Then we must apply some rotations to rebalance it

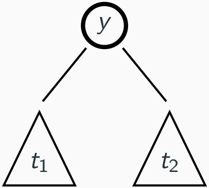
Deletion

Deleting an element is a bit more complicated than inserting it

Deletion may cause a subtree to decrese its black-height

Then we must apply some rotations to rebalance it

For example, if we `delete x` from the tree



in the case that $x < y$, we delete it from t_1
This may cause the black-height of t_1 to decrease
while the black-height of t_2 is unchanged

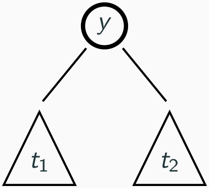
Deletion

Deleting an element is a bit more complicated than inserting it

Deletion may cause a subtree to decrese its black-height

Then we must apply some rotations to rebalance it

For example, if we `delete x` from the tree



in the case that $x < y$, we delete it from t_1
This may cause the black-height of t_1 to decrease
while the black-height of t_2 is unchanged

Define rebalancing functions for
when one child has a black-height larger by one than the other

Delete: Simultaneously Defined Functions

Delete: Simultaneously Defined Functions

- `delete :: Key -> RBTree -> RBTree`
  (`delete x t`) looks for key `x` in the tree `t`; if it finds it, it deletes it and rebalances the tree so the R-B properties hold

Delete: Simultaneously Defined Functions

- `delete :: Key -> RBTree -> RBTree`
  `(delete x t)` looks for key `x` in the tree `t`; if it finds it, it deletes it and rebalances the tree so the R-B properties hold
- `del :: Key -> RBTree -> RBTree`
  Preliminary version of `delete`: the result satisfies the R-B properties except it may have two consecutive red nodes at the top

Delete: Simultaneously Defined Functions

- `delete :: Key -> RBTree -> RBTree`
  (`delete x t`) looks for key `x` in the tree `t`; if it finds it, it deletes it and rebalances the tree so the R-B properties hold
- `del :: Key -> RBTree -> RBTree`
  Preliminary version of `delete`: the result satisfies the R-B properties except it may have two consecutive red nodes at the top
- `delL :: Key -> RBTree -> RBTree`
  Deletes an element from the left child

Delete: Simultaneously Defined Functions

- `delete :: Key -> RBTree -> RBTree`
  (`delete x t`) looks for key `x` in the tree `t`; if it finds it, it deletes it and rebalances the tree so the R-B properties hold
- `del :: Key -> RBTree -> RBTree`
  Preliminary version of `delete`: the result satisfies the R-B properties except it may have two consecutive red nodes at the top
- `delL :: Key -> RBTree -> RBTree`
  Deletes an element from the left child
- `balL :: Key -> RBTree -> RBTree`
  Rebalances a tree when the black-height of the left child is one shorter than the right

Delete: Simultaneously Defined Functions

- `delete :: Key -> RBTree -> RBTree`
  (`delete x t`) looks for key `x` in the tree `t`; if it finds it, it deletes it and rebalances the tree so the R-B properties hold
- `del :: Key -> RBTree -> RBTree`
  Preliminary version of `delete`: the result satisfies the R-B properties except it may have two consecutive red nodes at the top
- `delL :: Key -> RBTree -> RBTree`
  Deletes an element from the left child
- `balL :: Key -> RBTree -> RBTree`
  Rebalances a tree when the black-height of the left child is one shorter than the right
- `delR, balR :: Key -> RBTree -> RBTree`
  Like `delL` and `balL`, but on the right

Delete: Simultaneously Defined Functions

- `delete :: Key -> RBTree -> RBTree`
  (`delete x t`) looks for key `x` in the tree `t`; if it finds it, it deletes it and rebalances the tree so the R-B properties hold
- `del :: Key -> RBTree -> RBTree`
  Preliminary version of `delete`: the result satisfies the R-B properties except it may have two consecutive red nodes at the top
- `delL :: Key -> RBTree -> RBTree`
  Deletes an element from the left child
- `balL :: Key -> RBTree -> RBTree`
  Rebalances a tree when the black-height of the left child is one shorter than the right
- `delR, balR :: Key -> RBTree -> RBTree`
  Like `delL` and `balL`, but on the right
- `fuse :: RBTree -> RBTree -> RBTree`
  merges two trees t_1 and t_2 when all elements of t_1 are smaller than all elements of t_2

Balancing Left I

Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child

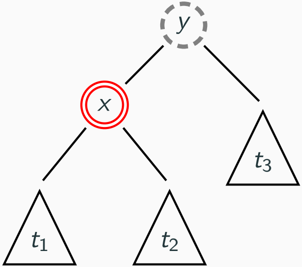
There can be three cases (color of the root irrelevant or obvious)

Balancing Left I

Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child

There can be three cases (color of the root irrelevant or obvious)

First Case (left child has a red node):



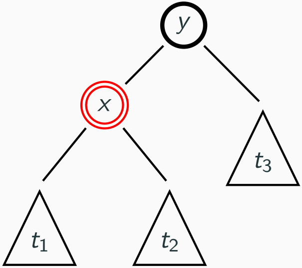
y must be black

Balancing Left I

Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child

There can be three cases (color of the root irrelevant or obvious)

First Case (left child has a red node):



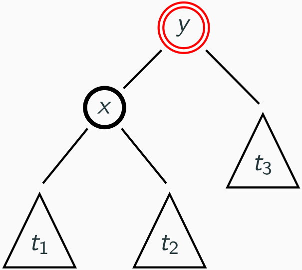
y must be black We swap the colors of x and y

Balancing Left I

Suppose we have a tree in which the black-height of the left child is one less than the black-height of the right child

There can be three cases (color of the root irrelevant or obvious)

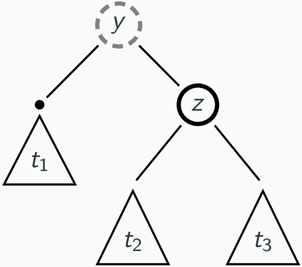
First Case (left child has a red node):



y must be black We swap the colors of x and y
The black-height of the right child decreases by 1,
the black-height of the left child is unchanged
(There could now be two red nodes at the top)

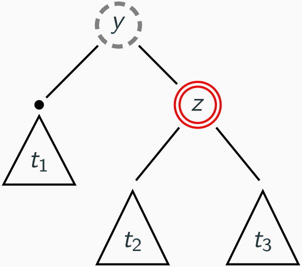
Balancing Left II

Second Case (left child black or leaf, right black):



Balancing Left II

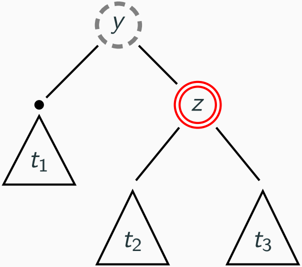
Second Case (left child black or leaf, right black):



Just repaint z red

Balancing Left II

Second Case (left child black or leaf, right black):

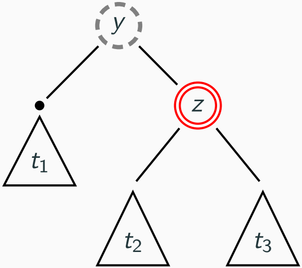


Just repaint z red

We decreased the black-height of the right child

Balancing Left II

Second Case (left child black or leaf, right black):



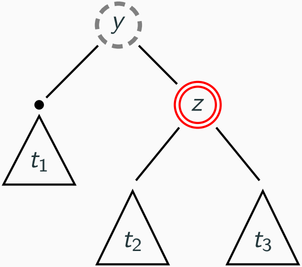
Just repaint z red

We decreased the black-height of the right child

But we may have created consecutive red nodes on the right

Balancing Left II

Second Case (left child black or leaf, right black):



Just repaint z red

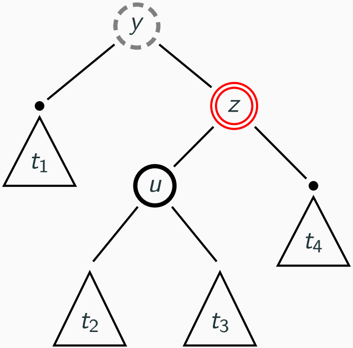
We decreased the black-height of the right child

But we may have created consecutive red nodes on the right

Apply `balance` to fix this problem

Balancing Left III

Third Case (left child black or leaf, right red):

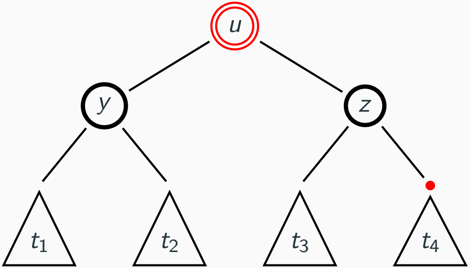


There must be at least a black node under z for the right child to have higher black-height

t_4 must have a black top node

Balancing Left III

Third Case (left child black or leaf, right red):



There must be at least a black node under z for the right child to have higher black-height

t_4 must have a black top node

We might have created consecutive red nodes in the right child

Apply `balance` to the right child

Balance Left in Haskell

If we put the three cases together we obtain the function
to rebalance when the left child has black-height smaller by 1

Balance Left in Haskell

If we put the three cases together we obtain the function
to rebalance when the left child has black-height smaller by 1

The input tree is give in its three components
No need to specify the color of the root

Balance Left in Haskell

If we put the three cases together we obtain the function
to rebalance when the left child has black-height smaller by 1

The input tree is give in its three components
No need to specify the color of the root

```
balL :: RBTree → Key → RBTree → RBTree
balL (Node Red t1 x t2) y t3
   = Node Red (Node Black t1 x t2) y t3
balL t1 y (Node Black t2 z t3)
   = balance Black t1 y (Node Red t2 z t3)
balL t1 y (Node Red (Node Black t2 u t3) z t4)
   = Node Red (Node Black t1 y t2)
              u
              (balance Black t3 z (redRoot t4))
```

Delete Left

Deleting a key from the left child (when $x < y$)

Delete Left

Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black
the deletion will decrease the black height
so we must rebalance with `balL`

Delete Left

Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black
the deletion will decrease the black height
so we must rebalance with `balL`

Otherwise (top node red)
the black-height stays the same and we don't need to rebalance

Delete Left

Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black
the deletion will decrease the black height
so we must rebalance with `balL`

Otherwise (top node red)
the black-height stays the same and we don't need to rebalance

```
delL :: Key → RBTree → Key → RBTree → RBTree
delL x t1 y t2 =
  if (color t1) == Black
  then balL (del x t1) y t2
  else NodeRB Red (del x t1) y t2
```

Delete Left

Deleting a key from the left child (when $x < y$)

If the initial top node of the child is black
the deletion will decrease the black height
so we must rebalance with `balL`

Otherwise (top node red)
the black-height stays the same and we don't need to rebalance

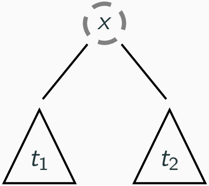
```
delL :: Key → RBTree → Key → RBTree → RBTree
delL x t1 y t2 =
  if (color t1) == Black
  then balL (del x t1) y t2
  else NodeRB Red (del x t1) y t2
```

Define similar functions `balR` and `delR`
to rebalance and delete on the right

Fuse

In the case when $x = y$, we must delete the root of the tree

If we delete x from



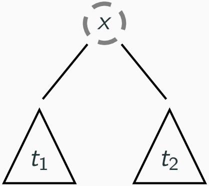
We're left with the orphan trees t_1 and t_2

We must put them back together into a single tree

Fuse

In the case when $x = y$, we must delete the root of the tree

If we delete x from



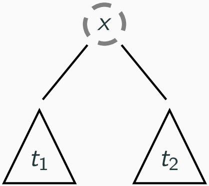
We're left with the orphan trees t_1 and t_2
We must put them back together into a single tree

The strategy that we used with Binary Search Trees
of replacing the deleted node with the minimum of the right child
doesn't work any more, because it may disrupt the R-B properties

Fuse

In the case when $x = y$, we must delete the root of the tree

If we delete x from



We're left with the orphan trees t_1 and t_2
We must put them back together into a single tree

The strategy that we used with Binary Search Trees
of replacing the deleted node with the minimum of the right child
doesn't work any more, because it may disrupt the R-B properties

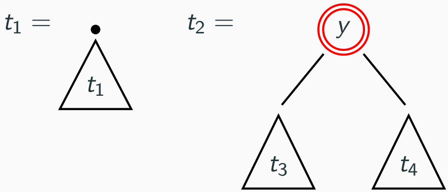
We must come up with a cleverer way of fusing t_1 and t_2

```
fuse ::  RBTree -> RBTree -> RBTree
```

We know that all elements of t_1 are smaller than all elements of t_2

Fuse: Different Color

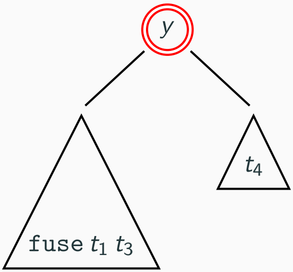
If the two trees have top nodes of different color



We can choose the red one as new top node

Fuse: Different Color

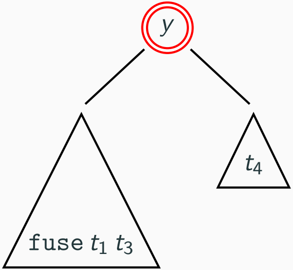
If the two trees have top nodes of different color



We can choose the red one as new top node

Fuse: Different Color

If the two trees have top nodes of different color

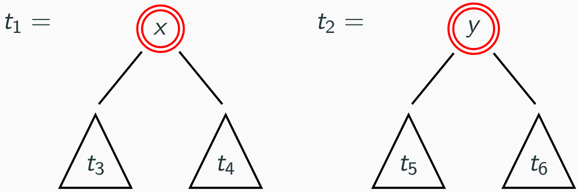


We can choose the red one as new top node

Similarly when the first is red and the second is black

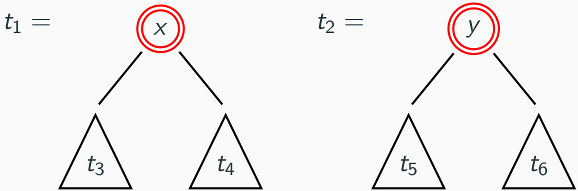
Fuse: Both Red

If both trees have a red top node



Fuse: Both Red

If both trees have a red top node

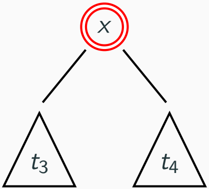


First we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

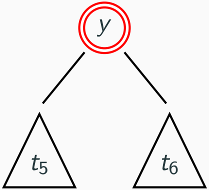
Fuse: Both Red

If both trees have a red top node

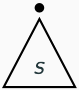
$t_1 =$



$t_2 =$



$\texttt{fuse}\ t_4\ t_5 =$

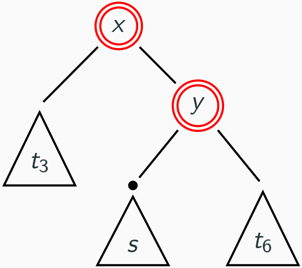


First we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

If s has a black top node,

Fuse: Both Red

If both trees have a red top node

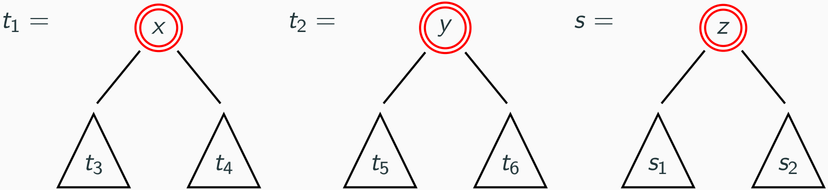


First we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

If s has a black top node, we put it under y

Fuse: Both Red

If both trees have a red top node

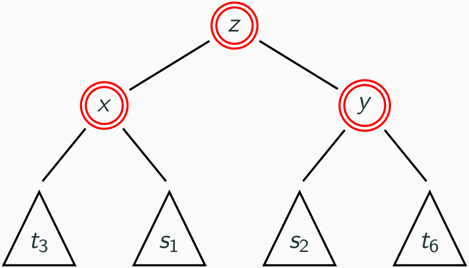


First we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\, t_4\, t_5$

If s has a red top node,

Fuse: Both Red

If both trees have a red top node



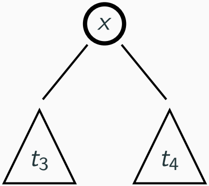
First we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

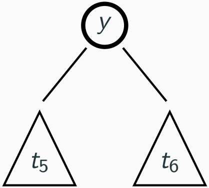
If s has a red top node, we use its node as new root

There are double red nodes on both sides, but
the top node will be recolored black either by `balL` or `balR` or `delete`,
according to where we deleted: left, right, or root

Fuse: Both Black

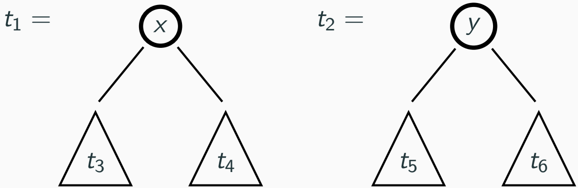
If both trees have a black top node

$t_1 =$ 

$t_2 =$ 

Fuse: Both Black

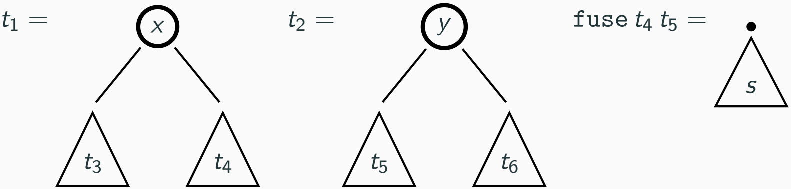
If both trees have a black top node



Again we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\, t_4\, t_5$

Fuse: Both Black

If both trees have a black top node

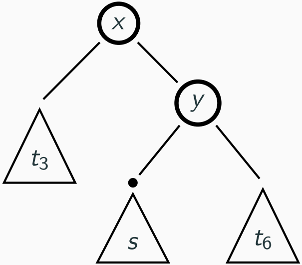


Again we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

If s has a black top node,

Fuse: Both Black

If both trees have a black top node



Again we recursively fuse the *middle subtrees*: $s =$ `fuse` $t_4\ t_5$

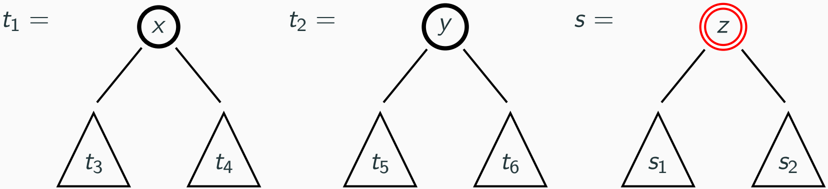
If s has a black top node, we put it under y

But this time the right subtree has increased black-height

We must apply `balL`

Fuse: Both Black

If both trees have a black top node

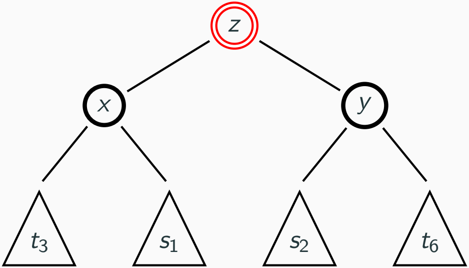


Again we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

If s has a red top node,

Fuse: Both Black

If both trees have a black top node



Again we recursively fuse the *middle subtrees*: $s = \texttt{fuse}\ t_4\ t_5$

If s has a red top node, we use it as new root

The main delete function

Having defined all the auxiliary functions, we can now simply implement the main delete function:

```
delete :: Key → RBTree → RBTree
delete x t = blackRoot (del x t)

del :: Key → RBTree → RBTree
del x LeafRB = LeafRB
del x (NodeRB _ t1 y t2)
   | x<y = delL x t1 y t2      -- delete from left child
   | x>y = delR x t1 y t2      -- delete from right child
   | otherwise = fuse t1 t2    -- delete root, fuse children
```

Dynamic Programming

Advanced Algorithms and Data Structures - Lecture 4

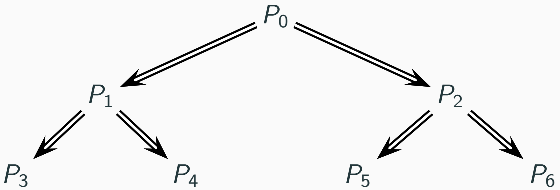
Venanzio Capretta

Monday 21 October 2019

School of Computer Science, University of Nottingham

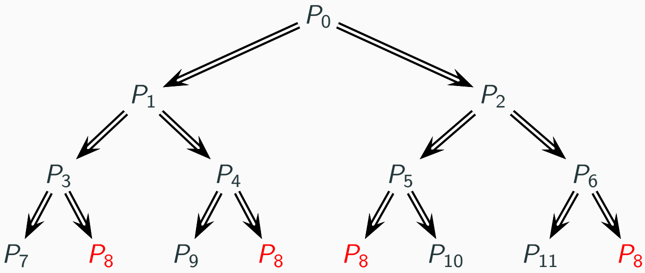
Repeated Subproblems

- Divide-and-Conquer:
  Split the problem into smaller subproblems - solve them recursively



Repeated Subproblems

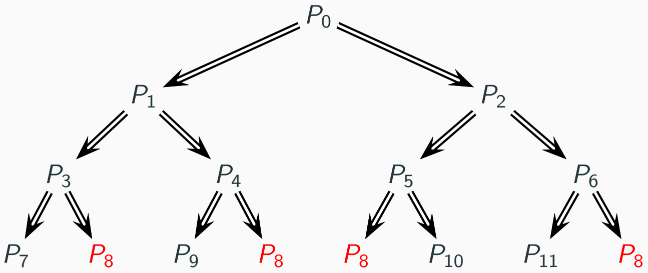
- Divide-and-Conquer:
  Split the problem into smaller subproblems - solve them recursively



- We may hit the same subproblem in different branches

Repeated Subproblems

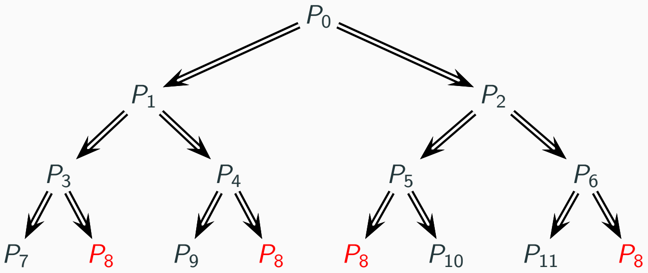
- Divide-and-Conquer:
  Split the problem into smaller subproblems - solve them recursively



- We may hit the same subproblem in different branches
- Divide-and-Conquer would recompute P_8 four times

Repeated Subproblems

- Divide-and-Conquer:
  Split the problem into smaller subproblems - solve them recursively

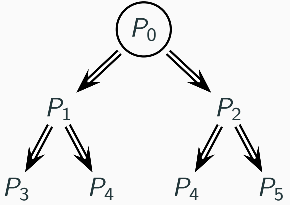


- We may hit the same subproblem in different branches
- Divide-and-Conquer would recompute P_8 four times
- Dynamic programming:
  Remember the solution of P_8 after the first time

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



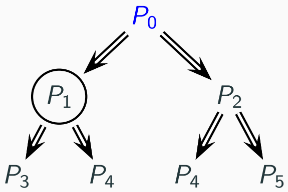
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?					

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



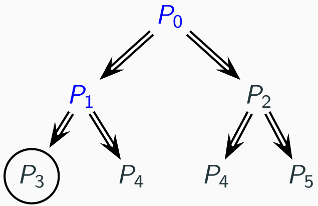
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?	?				

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



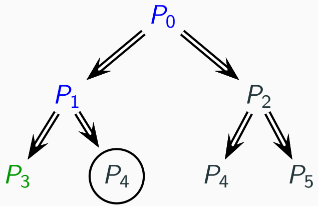
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?	?		?		

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



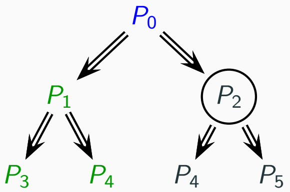
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?	?		S_3	?	

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



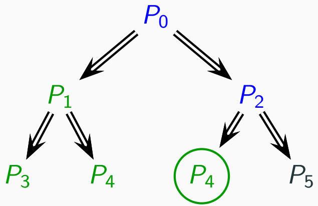
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?	S_1	?	S_3	S_4	

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



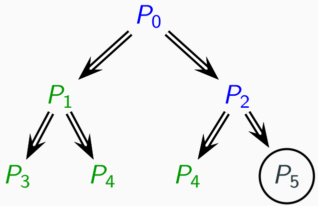
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?	S_1	?	S_3	S_4	

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table



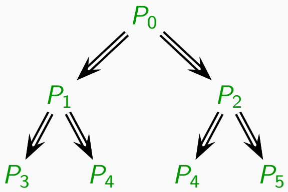
Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	?	S_1	?	S_3	S_4	?

The table is a global variable

Table Building

Dynamic Programming idea:

- Keep a table of already computed subproblems
- Look up a subproblem in the table before recomputing
- New subproblem? Compute the solution and add it to the table

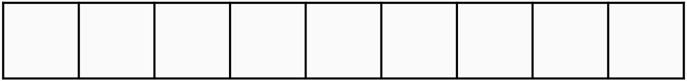


Problem	P_0	P_1	P_2	P_3	P_4	P_5
Solution	S_0	S_1	S_2	S_3	S_4	S_4

The table is a global variable

The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

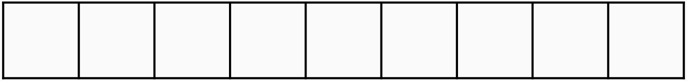


The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

We have a table of prices for pieces of different length

You must cut the rod to maximize the total price of the pieces



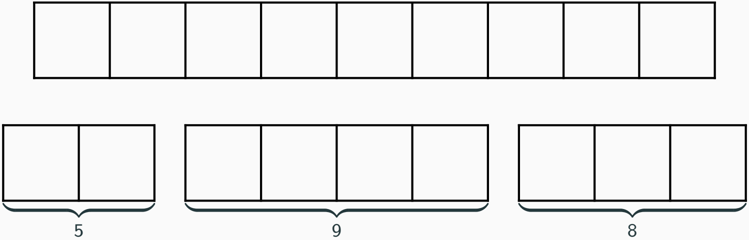
length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

We have a table of prices for pieces of different length

You must cut the rod to maximize the total price of the pieces



If we split it in $9 = 2 + 4 + 3$, price: $5 + 9 + 8 = 22$

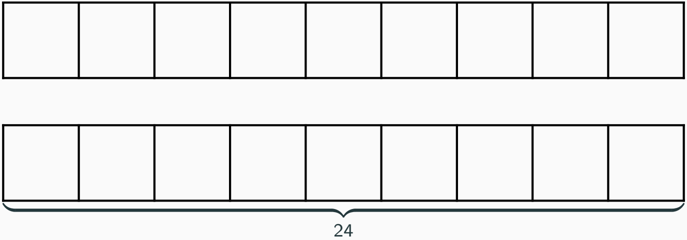
length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

We have a table of prices for pieces of different length

You must cut the rod to maximize the total price of the pieces



If we split it in $9 = 2 + 4 + 3$, price: $5 + 9 + 8 = 22$

If we don't split it, price: 24

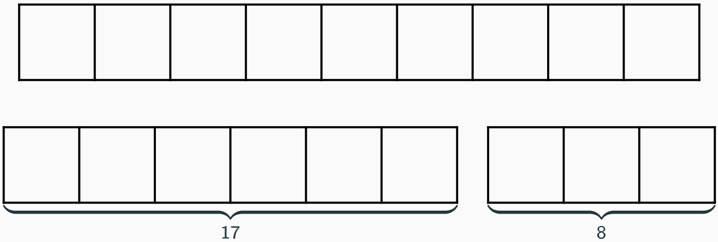
length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

We have a table of prices for pieces of different length

You must cut the rod to maximize the total price of the pieces



If we split it in $9 = 2 + 4 + 3$, price: $5 + 9 + 8 = 22$

If we don't split it, price: 24

If we split it in $9 = 6 + 3$, price: $17 + 8 = 25$

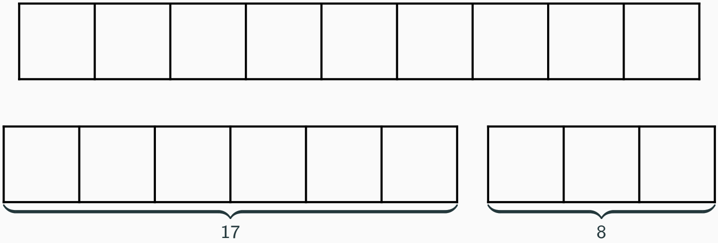
length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

The Rod-Cutting Problem

Cut a rod into pieces maximizing their total price

We have a table of prices for pieces of different length

You must cut the rod to maximize the total price of the pieces



If we split it in $9 = 2 + 4 + 3$, price: $5 + 9 + 8 = 22$

If we don't split it, price: 24

If we split it in $9 = 6 + 3$, price: $17 + 8 = 25$ (maximum)

length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30

Optimal Cut Equations (1)

We can adopt a divide-and-conquer strategy:

- First do one cut, you get two smaller rods
- Then apply the algorithm recursively to them

Optimal Cut Equations (1)

We can adopt a divide-and-conquer strategy:

- First do one cut, you get two smaller rods
- Then apply the algorithm recursively to them

Equations expressing the price r_n of an optimal cut of a rod of length n:

$$
\begin{aligned}
r_1 &= p_1 \\
r_n &= \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\}
\end{aligned}
$$

We cut the rod of length n into two rods of length i and $n - i$
in all possible ways (explicitely consider the uncut price p_n)

Optimal Cut Equations (2)

We can improve the algorithm by taking the first cut to be definitive:
The first half will not be further cut,
so we don't need a recursive call for it:

$$r_0 = 0$$
$$r_n = \max_{i=1\ldots n}(p_i + r_{n-i})$$

This takes care also of

- r_1 (it automatically gives p_1)
- the uncut option when $i = n$

Observation:

Possible improvement: assume that the first cut is the largest:
Cutting $9 = 3 + 6$ is equivalent to $9 = 6 + 3$
Order of cuts is unimportant: only consider the second one

But we don't follow this path (exercise: try)
We'll look at a better algorithm using Dynamic Programming

Naive Algorithm in Haskell

```haskell
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- Type: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices
    (`pr!!n` is the price of a rod of length `n`)
  - `n :: Int`, the length of the rod we want to cut

Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- Type: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices (`pr!!n` is the price of a rod of length `n`)
  - `n :: Int`, the length of the rod we want to cut
- A rod of length 0 cannot be cut and gives a total price of 0

Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- Type: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices (`pr!!n` is the price of a rod of length `n`)
  - `n :: Int`, the length of the rod we want to cut
- A rod of length 0 cannot be cut and gives a total price of 0
- For longer rods, we create the list of all possible prices
  - Make one cut and recursively compute the price of the optimal cut of the remaining part
  - Take the maximum of all possibilities

Naive Algorithm in Haskell

```
maxCut :: [Int] → Int → Int
maxCut pr 0 = 0
maxCut pr n = maximum [pr!!k + maxCut pr (n-k) | k ← [1..n]]
```

- Type: `maxCut` takes two inputs:
  - `pr :: [Int]`, the list of prices
    (`pr!!n` is the price of a rod of length `n`)
  - `n :: Int`, the length of the rod we want to cut
- A rod of length 0 cannot be cut and gives a total price of 0
- For longer rods, we create the list of all possible prices
  - Make one cut and recursively compute the price of the optimal cut of the remaining part
  - Take the maximum of all possibilities

Exercise: Modify it so it returns the cuts (the list of ks)

Complexity of Naive Algorithm

The Complexity is Exponential: $T(n) = O(2^n)$
(See IA for the formal derivation)

Problem:
We recompute several times optimal cuts for the same length
Eg when computing `maxCut pr 9`, among the possibilities we have
$9 = 5 + 4$, $9 = 3 + 2 + 4$, $9 = 4 + 1 + 4$ etc
The optimal solution for a rod of length 4 is recomputed each time.

Idea: keep a table with the optimal prices already computed and look up in it before recomputing.

DP Solution: Imperative

We construct a global array/table `bestCut`

that cointain the optimal cut for every length:

`bestCut!!i` = total price of optimal cut for a rod of length `i`

DP Solution: Imperative

We construct a global array/table `bestCut`
that cointain the optimal cut for every length:
`bestCut!!i` = total price of optimal cut for a rod of length `i`

Two ways of constructing the table:

- Top-Dow Memoization: Initialize all entries in the table with a dafault value ($-\infty$); implement like Divide-and-Conquer, but always check if the result is already in the table; if it is not, compute it and put it in the table

DP Solution: Imperative

We construct a global array/table `bestCut`
that cointain the optimal cut for every length:
`bestCut!!i` = total price of optimal cut for a rod of length `i`

Two ways of constructing the table:

- Top-Dow Memoization: Initialize all entries in the table with a dafault value ($-\infty$); implement like Divide-and-Conquer, but always check if the result is already in the table; if it is not, compute it and put it in the table
- Bottom-Up Method: Systmatically compute all the values in the table in order: `bestCut!!0`, `bestCut!!1`, ..., `bestCut!!n`; when computing `bestCut!!i`, we already know all the previous values are in the table

Bottom-Up is efficient if we know in advance that we need to compute all the values in the table

DP and lazy evaluation

In Functional Programming:

- Declarative Style: We can just define the table of values, without worrying about the order in which it is computed and when values will be available
- Lazy Evaluation: Entries of the table will be computed when needed and they persist for further calls

```
maxCutD :: [Int] → Int → Int
maxCutD pr n = last bestCut
  where bestCut = 0:[ maximum [ pr!!i + bestCut!!(k-i)
                                | i←[1..k] ]
                    | k ← [1..n] ]
```

Ingredients for Dynamic Programming

Ingredients for Dynamic Programming

- Optimal Substructure
  The optimal solution to an instance of the problem (eg cutting a rod of length n) contains optimal solutions of some subproblems (cutting rods of shorter length)

Ingredients for Dynamic Programming

- Optimal Substructure
  The optimal solution to an instance of the problem (eg cutting a rod of length n) contains optimal solutions of some subproblems (cutting rods of shorter length)
- Overlapping Subproblems Different branches of the computation of an optimal solution require to compute the same subproblem several times

Graph Algorithms

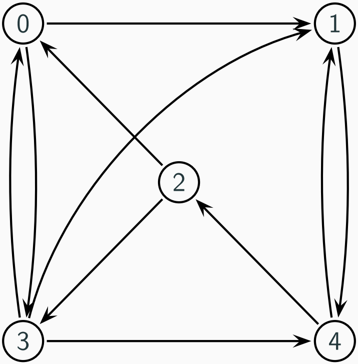
Advanced Algorithms and Data Structures - Lecture 5

Venanzio Capretta

Monday 28 October 2019

School of Computer Science, University of Nottingham

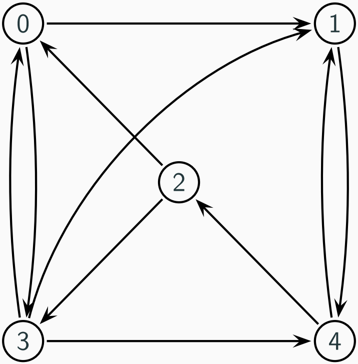
Directed Graphs



A (directed) graph consists of

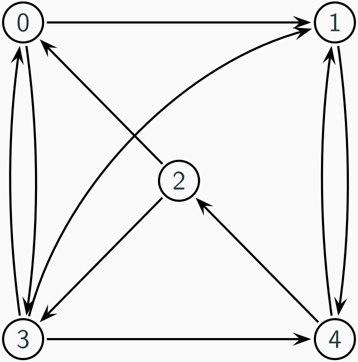
- a set of vertices: $\{0, 1, 2, 3, 4\}$
- a set of edges between the vertices: $\{(0, 1), (0, 3), (1, 4), (2, 0), (2, 3), (3, 0), (3, 1), (3, 4), (4, 1), (4, 2)\}$

Edge Representation



There are several ways of representing graphs as data structures:

Edge Representation



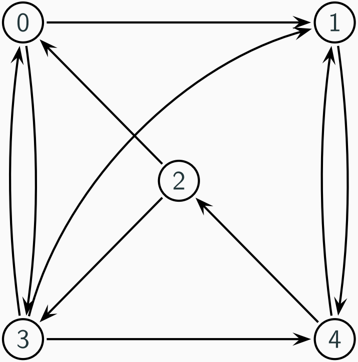
There are several ways of representing graphs as data structures:

List of edges:

$$[(0,1),(0,3),(1,4),(2,0),(2,3),(3,0),(3,1),(3,4),(4,1),(4,2)]$$

We assume the set of vertices is implicit:
the vertices are the ones given as source or targets of edges

Adjacency List



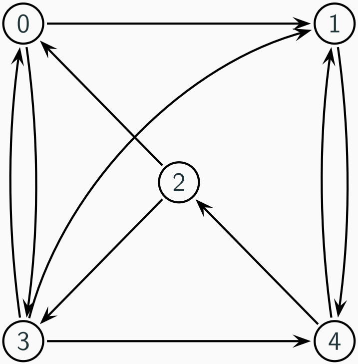
$0 \rightarrow [1, 3]$
$1 \rightarrow [4]$
$2 \rightarrow [0, 3]$
$3 \rightarrow [0, 1, 4]$
$4 \rightarrow [1, 2]$

Adjacency List:

For every vertex $i \rightarrow$ a list of vertices j for which there is an edge (i, j)

If the vertices are numbered $\{0, \ldots, n-1\}$,
we can leave the source unspecified (it's the index in the list)
List of lists: $[[1, 3], [4], [0, 3], [0, 1, 4], [1, 2]]$

Adjacency Matrix



	0	1	2	3	4
0	false	true	false	true	false
1	false	false	false	false	true
2	true	false	false	true	false
3	true	true	false	false	true
4	false	true	true	false	false

Adjacency Matrix: An $n \times n$ matrix of Booleans

The (i, j) entry is true if there is an edge from i to j

Space Complexity

The amount of memory necessary to store a graph depends on the representation

- With an djacency list we need $\Theta(V + E)$ space
  where V is the number of vertices and E is the number of edges
- With an adjacency matrix we need $\Theta(V^2)$ space
  independently of the number of edges

Space Complexity

The amount of memory necessary to store a graph depends on the representation

- With an djacency list we need $\Theta(V + E)$ space
  where V is the number of vertices and E is the number of edges
- With an adjacency matrix we need $\Theta(V^2)$ space
  independently of the number of edges

Which one is more convenient depends on the number of edges:

- Sparse Graphs:
  the number of edges is much smaller than the possible maximum V^2
  It is more convenient to use a adjacency list
- Dense Graphs:
  the number of edges is close to the possible maximum V^2
  It is more convenient to use a adjacency matrix

Space Complexity

The amount of memory necessary to store a graph depends on the representation

- With an djacency list we need $\Theta(V + E)$ space
  where V is the number of vertices and E is the number of edges
- With an adjacency matrix we need $\Theta(V^2)$ space
  independently of the number of edges

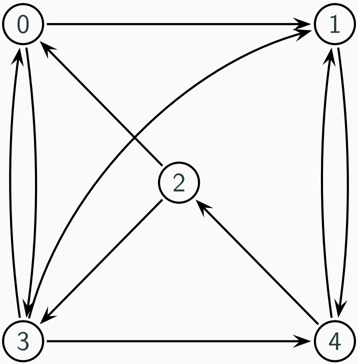
Which one is more convenient depends on the number of edges:

- Sparse Graphs:
  the number of edges is much smaller than the possible maximum V^2
  It is more convenient to use a adjacency list
- Dense Graphs:
  the number of edges is close to the possible maximum V^2
  It is more convenient to use a adjacency matrix

Exercise: Write conversion functions between the two representations

Minimum Length Problem

Given two vertices i and j in a graph,
find a path from i to j with the least number of edges



From 0 to 3:

There is a path of length 4: $0 \to 1 \to 4 \to 2 \to 3$

But the direct path has length 1: $0 \to 3$

Dynamic Programming for Minimum Path

We may solve the problem efficiently using Dynamic Programming

Verify that the conditions for DP are met:

Optimal Substructure

Suppose a path $\pi : i \rightsquigarrow j$ goes through an intermediate vertex k:

$$\underbrace{i \overset{\pi_1}{\rightsquigarrow} k \overset{\pi_2}{\rightsquigarrow} j}_{\pi}$$

If π is a minimum path from i to j, then

π_1 is a minimum path from i to k and

π_2 is a minimum path from k to j

Overlapping Subproblems

Overlapping Subproblems

The same subproblem may occur in different branches of the computation:

$$i \dashrightarrow k \dashrightarrow j$$

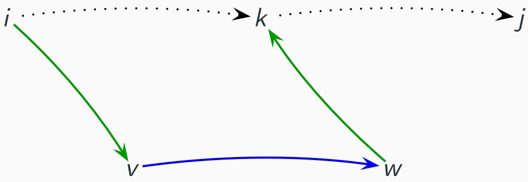
I'm trying to find a minimum path from i to j

I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Overlapping Subproblems

Overlapping Subproblems

The same subproblem may occur in different branches of the computation:



I'm trying to find a minimum path from i to j

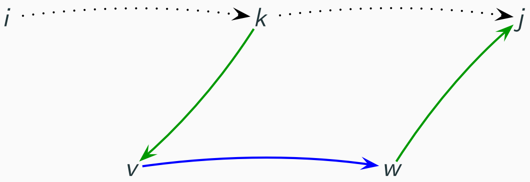
I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Computing $i \rightsquigarrow k$ may involve paths going from v to w

Overlapping Subproblems

Overlapping Subproblems

The same subproblem may occur in different branches of the computation:



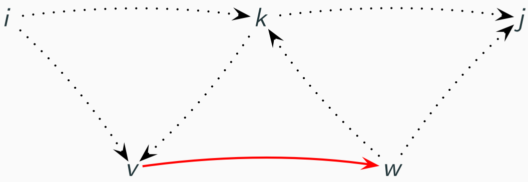
I'm trying to find a minimum path from i to j
I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Computing $i \rightsquigarrow k$ may involve paths going from v to w
Computing $k \rightsquigarrow j$ may also involve paths going from v to w (not both)

Overlapping Subproblems

Overlapping Subproblems

The same subproblem may occur in different branches of the computation:



I'm trying to find a minimum path from i to j

I use an intermediate vertex k; subprobems: $i \rightsquigarrow k$, $k \rightsquigarrow j$

Computing $i \rightsquigarrow k$ may involve paths going from v to w

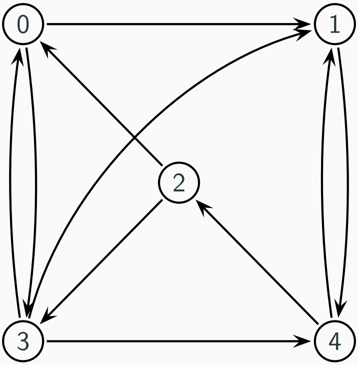
Computing $k \rightsquigarrow j$ may also involve paths going from v to w (not both)

The subproblem $v \rightsquigarrow w$ is recomputed several times

Exercise: Write a DP algorithm to solve the shortest path problem

Longest Path Problem

Similar problem: Find the longest simple path between two nodes
(simple = contains no cycles)



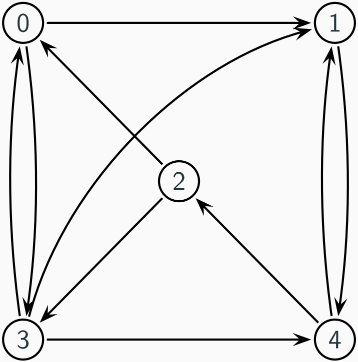
Longest Path from 0 to 3, length 4: $0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3$

With cycles we could make it as long as we want, ex length 8:

$0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3$

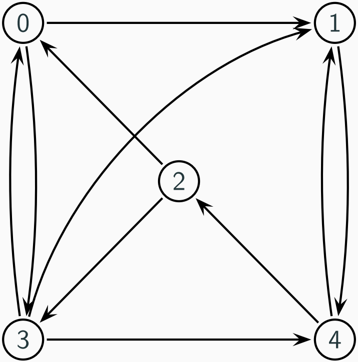
DP for Maximum Length?

Can DP also be applied to this problem? Optimal Substrcture?



DP for Maximum Length?

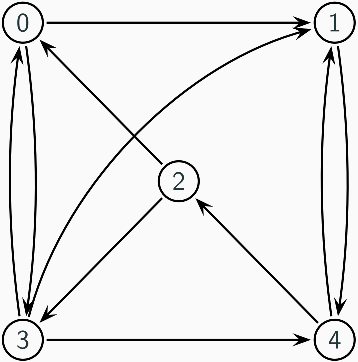
Can DP also be applied to this problem? Optimal Substrcture?



- Optimal solution for $0 \rightsquigarrow 3$: $0 \to 1 \to 4 \to 2 \to 3$
- It goes through 1, subproblems: $0 \rightsquigarrow 1$ and $1 \rightsquigarrow 3$

DP for Maximum Length?

Can DP also be applied to this problem? Optimal Substrcture?



- Optimal solution for $0 \rightsquigarrow 3$: $0 \to 1 \to 4 \to 2 \to 3$
- It goes through 1, subproblems: $0 \rightsquigarrow 1$ and $1 \rightsquigarrow 3$
- Optimal solution for $0 \rightsquigarrow 1$: $0 \to 3 \to 4 \to 1$
- Optimal solution for $1 \rightsquigarrow 3$: $1 \to 4 \to 2 \to 3$
  We can't put the subproblem together: cycles!

No DP for Maximum Length

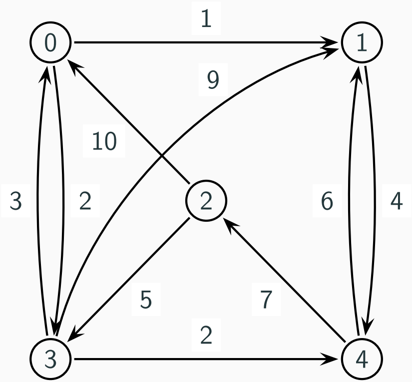
The Maximum Length Problem does not have Optimal Substrcture

We can't apply Dynamic Programming to find an efficient algorithm

In fact, this is an NP-complete problem

Weighted Graphs

We assign to every edge a weight:



Every edge is assign a real number, its weight

We can easily modify the adjacency list and adjacency matrix representations to include weights.

Weighted Graph Representations

- Adjacency List

  The entries in the list are pairs of target-vertices and edge-weights

$$
\begin{aligned}
0 &\rightarrow [(1,1.0),(3,2.0)] \\
1 &\rightarrow [(4,4.0)] \\
2 &\rightarrow [(0,10.0),(3,5.0)] \\
3 &\rightarrow [(0,3.0),(1,9.0),(4,2.0)] \\
4 &\rightarrow [(1,6.0),(2,7.0)]
\end{aligned}
$$

$$
\begin{aligned}
&[[(1,1.0),(3,2.0)] \\
&\ [(4,4.0)] \\
&\ [(0,10.0),(3,5.0)] \\
&\ [(0,3.0),(1,9.0),(4,2.0)], \\
&\ [(1,6.0),(2,7.0)]]
\end{aligned}
$$

Weighted Graph Representations

- Adjacency List

  The entries in the list are pairs of target-vertices and edge-weights

$$\begin{aligned} 0 &\to [(1, 1.0), (3, 2.0)] \\ 1 &\to [(4, 4.0)] \\ 2 &\to [(0, 10.0), (3, 5.0)] \\ 3 &\to [(0, 3.0), (1, 9.0), (4, 2.0)] \\ 4 &\to [(1, 6.0), (2, 7.0)] \end{aligned}$$

$$\begin{array}{l} [[(1, 1.0), (3, 2.0)] \\ \;[(4, 4.0)] \\ \;[(0, 10.0), (3, 5.0)] \\ \;[(0, 3.0), (1, 9.0), (4, 2.0)], \\ \;[(1, 6.0), (2, 7.0)]] \end{array}$$

- Adjecency Matrix

  The entries in the matrix are weights instead of Booleans

	0	1	2	3	4
0	·	1.0	·	2.0	·
1	·	·	·	·	4.0
2	10.0	·	·	5.0	·
3	3.0	9.0	·	·	2.0
4	·	6.0	7.0	·	·

Shortest Path Problems

Shortest path problem

Find a path such that the sum of the weights of its edges has the minimum possible value

We assume the weights to be non-negative

(If we allow negatives, findind the shortest is as hard as the longest path)

The version with no weights is a special case: all edges have weight 1.0

Shortest Path Problems

Shortest path problem

Find a path such that the sum of the weights of its edges has the minimum possible value

We assume the weights to be non-negative

(If we allow negatives, findind the shortest is as hard as the longest path)

The version with no weights is a special case: all edges have weight 1.0

Two versions:

- Single-Source Shortest Paths
  Fix a source vertex,
  find the shortest paths from that source to all vertices

Shortest Path Problems

Shortest path problem

Find a path such that the sum of the weights of its edges has the minimum possible value

We assume the weights to be non-negative

(If we allow negatives, findind the shortest is as hard as the longest path)

The version with no weights is a special case: all edges have weight 1.0

Two versions:

- Single-Source Shortest Paths
  Fix a source vertex,
  find the shortest paths from that source to all vertices
- All-Pairs Shortest Paths
  Find the shortest path between all pairs of two vertices

Relaxation

In the solution of the single-source shortest paths problem

- We call $w_{i,j}$ the weight of an edge from i to j;
  If there is no edge $w_{i,j} = \infty$
- We keep an estimate dist_i of
  the minimum length of a path from the source s to the vertex i

Relaxation

In the solution of the single-source shortest paths problem

- We call $w_{i,j}$ the weight of an edge from i to j;
  If there is no edge $w_{i,j} = \infty$
- We keep an estimate dist_i of
  the minimum length of a path from the source s to the vertex i

We will use an auxiliary relaxation algorithm to update the distances:

- Suppose we have estimated dist_i without using the vertex k
  (That is, our estimate of dist_i uses paths that don't include k)
- If at one point we found the minimum distance dist_k,
  (so dist_i is just an estimate, while dist_k is the correct value)
- We can use k to ipdate the estimate dist_i

Relaxation

In the solution of the single-source shortest paths problem

- We call $w_{i,j}$ the weight of an edge from i to j;
  If there is no edge $w_{i,j} = \infty$
- We keep an estimate dist_i of
  the minimum length of a path from the source s to the vertex i

We will use an auxiliary relaxation algorithm to update the distances:

- Suppose we have estimated dist_i without using the vertex k
  (That is, our estimate of dist_i uses paths that don't include k)
- If at one point we found the minimum distance dist_k,
  (so dist_i is just an estimate, while dist_k is the correct value)
- We can use k to ipdate the estimate dist_i

Relaxation: If $\text{dist}_k + w_{k,i} < \text{dist}_i$ then update $\text{dist}_i \leftarrow \text{dist}_k + w_{k,i}$

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance dist_i has been estimated but not yet fixed

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance dist_i has been estimated but not yet fixed

This will be a Priority Queue
A data type which represent a set of keys (vertices)
with values (estimated distances)
supporting the following operations:

- Insert a new element in the queue with associated value
- Extract the element with the minimum value
- Update the value of an element in the list

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance dist_i has been estimated but not yet fixed

This will be a Priority Queue
A data type which represent a set of keys (vertices)
with values (estimated distances)
supporting the following operations:

- Insert a new element in the queue with associated value
- Extract the element with the minimum value
- Update the value of an element in the list

The algorithm iterates extracting the vertex with the minimum distance
and updating the remaining vertex-distances using relaxation

Priority Queues

In our algorithm we will keep a queue of vertices
whose distance dist_i has been estimated but not yet fixed

This will be a Priority Queue
A data type which represent a set of keys (vertices)
with values (estimated distances)
supporting the following operations:

- Insert a new element in the queue with associated value
- Extract the element with the minimum value
- Update the value of an element in the list

The algorithm iterates extracting the vertex with the minimum distance and updating the remaining vertex-distances using relaxation

For now we can use a naive representation of queues as list of pairs or (balanced) search trees

We will see efficient tree representations (Heaps) in future lectures:
Leftist Heaps, Fibonacci Heaps

Dijkstra's Algorithm

Let the source vertex be s

Keep a vector dist that, for every vertex i, contain an approximation dist_i of the length of the shortest path from s to i

Keep an queue Q of edges whose distance from s has not yet been fully computed

DIJKSTRA'S ALGORITHM:

- Initialize the distance: $\text{dist}_i = \infty$ for all i, except $\text{dist}_s = 0.0$
- Initialize the queue: $Q = V$ all vertices
- Repeat while Q is not empty
  - Extract from Q the vertex i with the minimum dist_i
  - Relax the distances of all remaining elements of Q using i

All-pairs shortest path

To compute the minimum distances between all pairs of vertices
We could apply Dijkstra's algorithm repeatedly,
running the source through all vertices

All-pairs shortest path

To compute the minimum distances between all pairs of vertices
We could apply Dijkstra's algorithm repeatedly,
running the source through all vertices

A better method is the Floyd-Warshall Algorithm
It uses a form of Dynamic Programming
It works also with negative weights
as long as there are no negative cycles

All-pairs shortest path

To compute the minimum distances between all pairs of vertices
We could apply Dijkstra's algorithm repeatedly,
running the source through all vertices

A better method is the Floyd-Warshall Algorithm
It uses a form of Dynamic Programming
It works also with negative weights
as long as there are no negative cycles

Idea: Use an growing set of intermediate vertices
to construct better and better paths

The intermediate vertices of a path $i_0 \to i_1 \to \cdots \to i_{m-1} \to i_m$
are $\{i_1, \ldots, i_{m-1}\}$

Floyd-Warshall Algorithm

Let V_k be the set of vertices $\{0, \ldots, k-1\}$
So $V_0 = \emptyset$, $V_1 = \{0\}$, $V_2 = \{0, 1\}$, etc.
V_n is the set of all vertices

For every k, we compute the minimum distances $\text{dist}_{i,j}^{(k)}$ of a path from i to j that uses only elements of V_k as intermediate vertices

Floyd-Warshall Algorithm

Let V_k be the set of vertices $\{0, \ldots, k-1\}$
So $V_0 = \emptyset$, $V_1 = \{0\}$, $V_2 = \{0, 1\}$, etc.
V_n is the set of all vertices

For every k, we compute the minimum distances $\text{dist}_{i,j}^{(k)}$ of a path from i to j that uses only elements of V_k as intermediate vertices

- $\text{dist}_{i,j}^{(0)} = w_{i,j}$ ($\text{dist}_{i,j}^{(0)} = \infty$ if there is no edge)
- A minimum path from i to j that only uses intermediate vertices from V_{k+1} either goes through k or not
  - If it doesn't go through k, then it only uses V_k and $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,j}^{(k)}$
  - If it goes through k, then it is made of a path from i to k and a path from k to j; these paths do not use k as internal vertex, so $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)}$
- So $\text{dist}_{i,j}^{(k+1)} = \min(\text{dist}_{i,j}^{(k)}, \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)})$

Floyd-Warshall Algorithm

Let V_k be the set of vertices $\{0, \ldots, k-1\}$
So $V_0 = \emptyset$, $V_1 = \{0\}$, $V_2 = \{0, 1\}$, etc.
V_n is the set of all vertices

For every k, we compute the minimum distances $\text{dist}_{i,j}^{(k)}$ of a path from i to j that uses only elements of V_k as intermediate vertices

- $\text{dist}_{i,j}^{(0)} = w_{i,j}$ ($\text{dist}_{i,j}^{(0)} = \infty$ if there is no edge)
- A minimum path from i to j that only uses intermediate vertices from V_{k+1} either goes through k or not
  - If it doesn't go through k, then it only uses V_k and $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,j}^{(k)}$
  - If it goes through k, then it is made of a path from i to k and a path from k to j; these paths do not use k as internal vertex, so $\text{dist}_{i,j}^{(k+1)} = \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)}$
- So $\text{dist}_{i,j}^{(k+1)} = \min(\text{dist}_{i,j}^{(k)}, \text{dist}_{i,k}^{(k)} + \text{dist}_{k,j}^{(k)})$

Floyd-Warshall Algorithm: Use the previous recursive equations to construct a sequence of matrices $(\text{dist}_{i,j}^{(k)})_{i,j=0\ldots n-1}$ for $k = 0 \ldots n$
Return $(\text{dist}_{i,j}^{(n)})_{i,j=0\ldots n-1}$

Amortized Complexity

Advanced Algorithms and Data Structures - Lecture 7

Venanzio Capretta

Monday 11 November 2019

School of Computer Science, University of Nottingham

Amortized Analysis

Some data structures have operations with high worst-case complexity,
but when we do a sequence of operations,
the average cost is small:
one costly operation can be compensated by many cheap ones

Amortized Analysis assigns to each operation

- An amortized cost that
- may be smaller than actual cost
- but takes into account a way of averaging computation steps over several operations.

Amortized Cost

Amortized cost must be defined so that the total amortized cost of a sequence of operations is larger or equal to the actual cost:
We perform a sequence of operations on the data:

$$f_1, f_2, f_3, \cdots, f_m$$

Each operation f_i has an actual cost t_i
We assign to it an amortized cost a_i
We must guarantee that

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

So the amortized complexity is an overestimation of the actual complexity

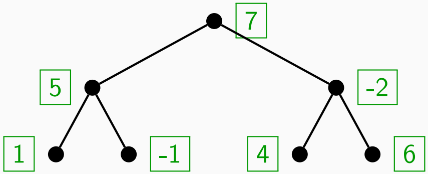
Accounting method

There are two main methods of amortized analysis:
the accounting method and the potential method

THE ACCOUNTING METHOD (also called *banker's method*)

We imagine that every location in the data structure has a store
where we can save *credits*, virtual time steps
that can be used at a different time

For example, a tree structure will store credits in each node:



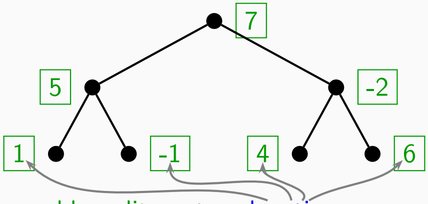
Accounting method

There are two main methods of amortized analysis:
the accounting method and the potential method

THE ACCOUNTING METHOD (also called *banker's method*)

We imagine that every location in the data structure has a store where we can save *credits*, virtual time steps that can be used at a different time

For example, a tree structure will store credits in each node:



Every operation can add credits c_i to a location or use credits $\overline{c_i}$ from a location

Credit Accounts

Amortized cost of operation f_i:

$$a_i = t_i + c_i - \overline{c_i}$$

where

- t_i is the actual time cost of f_i
- c_i is the number of credits allocated by operation f_i
- $\overline{c_i}$ is the number of credits spent by operation f_i

For the total amortized cost to be an overestimate of actual cost

$$\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$$

The overall credit must always be positive (never in debt):

$$\sum_{i=1}^{m} c_i \geq \sum_{i=1}^{m} \overline{c_i}$$

Potential Method

Also called *physicist's method*

We associate a potential function to a data structure D

$$\phi : D \to \mathbb{R}_{\geq 0}$$

Intuitively, the potential gives us some *complexity for free*:
We can compensate for a costly operation by using some of the potential

Cheap operations may increase the potential, so it can be used later

Usually we define ϕ so that the initial (empty) data structure has potential zero

Variation of Potential

Amortized cost of an operation f_i:

$$a_i = t_i + \phi(d_i) - \phi(d_{i-1})$$

- t_i is the actual cost
- $d_{i-1} \in D$ is the state of the data structure before operation f_i
- $d_i \in D$ is the state of the data structure after operation f_i
- So $\phi(d_i) - \phi(d_{i-1})$ is the change of potential

The actual cost is $t_i = a_i + \phi(d_{i-1}) - \phi(d_i)$
(The amortized cost minus the change of potential
ie, me must spend actual time to charge the potential)

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\sum_{i=1}^{m} t_i$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i))$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\begin{aligned}\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i)) \\ &= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} (\phi(d_{i-1}) - \phi(d_i)) \quad \text{(telescoping summation)}\end{aligned}$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m}(a_i + \phi(d_{i-1}) - \phi(d_i)) \\
&= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m}(\phi(d_{i-1}) - \phi(d_i)) \quad \text{(telescoping summation)} \\
&= \sum_{i=1}^{m} a_i + \phi(d_0) - \phi(d_m)
\end{aligned}$$

Sequence of Operations

If we perform several operation in sequence,
starting with the data structure in state d_0

$$d_0 \xmapsto{f_1} d_1 \xmapsto{f_2} d_2 \xmapsto{f_3} \cdots \xmapsto{f_m} d_m$$

The total actual cost is:

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} (a_i + \phi(d_{i-1}) - \phi(d_i)) \\
&= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} (\phi(d_{i-1}) - \phi(d_i)) \quad \text{(telescoping summation)} \\
&= \sum_{i=1}^{m} a_i + \phi(d_0) - \phi(d_m)
\end{aligned}$$

If the initial potential is zero, $\phi(d_0) = 0$, then $\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i - \phi(d_m)$

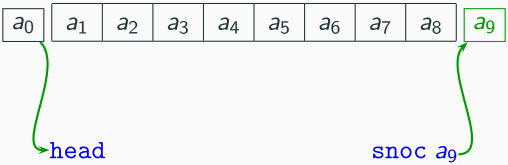
and the actual cost is smaller than the amortized cost: $\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i$

FIFO Queues

A simple example of use of amortized analysis is the data structure of First In First Our (FIFO) Queues

Lists of elements of some type A with operations:

- Insert a new element at the end (snoc)
- Get an element from the front (head)



(The word `snoc` is the inverse of `cons`, which is the usual operation to add an element in front of a list)

Queue Type

The data type `Queue` is required to have the following methods:

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness
- `snoc :: Queue -> A -> Queue`
  Adds an element at the end of the queue

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness
- `snoc :: Queue -> A -> Queue`
  Adds an element at the end of the queue
- `head :: Queue -> A`
  Read the first element of the queue

Queue Type

The data type `Queue` is required to have the following methods:

- `empty :: Queue`
  The queue with not elements
- `isEmpty :: Queue -> Bool`
  Test for emptiness
- `snoc :: Queue -> A -> Queue`
  Adds an element at the end of the queue
- `head :: Queue -> A`
  Read the first element of the queue
- `extract :: Queue -> (A,Queue)`
  Remove the first element of the queue,
  return it together with the tail

Imperative Implementation

In imperative programming we can realize queues as doubly-linked lists:

This allows us to perform all operations in constant time

Imperative Implementation

In imperative programming we can realize queues as doubly-linked lists:

This allows us to perform all operations in constant time

In functional programming
(or imperative programming if we want to save on pointers),
we can achieve constant amortized cost
by representing a queue as a pair of lists

$$([a_0, a_1], [a_3, a_2])$$

(the second part is inverted)

Functional Implementation

```
Queue = ([A], [A])
```

A queue is split in two parts:

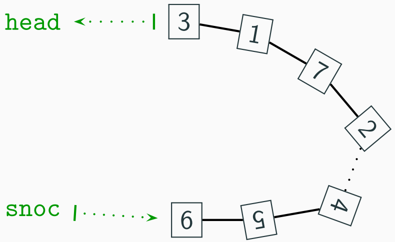
- A front portion and
- A rear portion, which is reversed

The queue

3	1	7	2	4	5	9

can be represented as $([3, 1, 7, 2], [9, 5, 4])$

Imagine that the queue is *bent* to present both ends to the user:



Different Representations

The representation is not unique

The same queue has alternative representations:

$$([3,1,7],[9,5,4,2]) \qquad ([3,1,7,2,4,5],[9]) \qquad \text{etc.}$$

Different Representations

The representation is not unique

The same queue has alternative representations:

$$([3,1,7],[9,5,4,2]) \qquad ([3,1,7,2,4,5],[9]) \qquad \text{etc.}$$

It is possible that one of the two portions is empty:

$$([3,1,7,2,4,5,9],[]) \qquad ([],[9,5,4,2,7,1,3])$$

Different Representations

The representation is not unique

The same queue has alternative representations:

$$([3, 1, 7], [9, 5, 4, 2]) \qquad ([3, 1, 7, 2, 4, 5], [9]) \qquad \text{etc.}$$

It is possible that one of the two portions is empty:

$$([3, 1, 7, 2, 4, 5, 9], []) \qquad ([], [9, 5, 4, 2, 7, 1, 3])$$

All operations can be executed in constant time

Except `head` and `extract` when the front list is empty

In that case, we must first reverse the rear list and then extract:

$$
\begin{array}{cl}
([], [9, 5, 4, 2, 7, 1, 3]) & \\
\downarrow & \text{reverse the rear} \quad O(n) \\
([3, 1, 7, 2, 4, 5, 9], []) & \\
\downarrow & \texttt{extract} \\
(3, ([1, 7, 2, 4, 5, 9], [])) &
\end{array}
$$

Implementation of Insertion and Extraction

```
snoc (f,r) x = (f, x:r)
```

Add the new element at the front of the rear list

Remember that the rear list is inverted

Implementation of Insertion and Extraction

```
snoc (f,r) x = (f, x:r)
```

Add the new element at the front of the rear list

Remember that the rear list is inverted

```
extract (x:f, r) = (x, (f,r))
extract ([],[])  = error "Empty Queue"
extract ([],r)   = extract (reverse r, [])
```

In the last case we can assume that the rear list is not empty, so the recursive call to `extract` will hit the first case

Implementation of Insertion and Extraction

```
snoc (f,r) x = (f, x:r)
```

Add the new element at the front of the rear list
Remember that the rear list is inverted

```
extract (x:f, r) = (x, (f,r))
extract ([],[])  = error "Empty Queue"
extract ([],r)   = extract (reverse r, [])
```

In the last case we can assume that the rear list is not empty, so the recursive call to `extract` will hit the first case

The last case of `extract` has cost $O(n)$ because we must reverse the rear list. But after that, we can extract the next n elements in constant time.

We can show that all operations have constant amortized cost

Potential for Queues

The potential function for queues is the length of rear list:

$$\phi(\texttt{f}, \texttt{r}) = \texttt{length}\ \texttt{r}$$

Potential for Queues

The potential function for queues is the length of rear list:

$$\phi(\texttt{f}, \texttt{r}) = \texttt{length}\,\texttt{r}$$

Analysis of the amortized cost of extract:

- First case: `extract (s:f,r)`

Potential for Queues

The potential function for queues is the length of rear list:

$$\phi\,(\texttt{f}, \texttt{r}) = \texttt{length}\,\texttt{r}$$

Analysis of the amortized cost of extract:

- First case: `extract (s:f,r)`

$$\begin{array}{rcccccc}
a & = & t & + & \phi\,(\texttt{f}, \texttt{r}) & - & \phi\,(\texttt{x} : \texttt{f}, \texttt{r}) \\
 & & \| & & \| & & \| \\
 & & 1 & & \texttt{length}\,\texttt{r} & & \texttt{length}\,\texttt{r} \\
 & & \uparrow & & & & \\
 & & \multicolumn{5}{l}{\text{just one step to get the head}} \\
 & = & 1 & & & &
\end{array}$$

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rcl}
a & = & \underset{\substack{\| \\ \texttt{length}\,\texttt{r} + 1 \\ \uparrow \\ \text{invert } \texttt{r} \text{ and take the head}}}{t} \quad + \quad \underset{\substack{\| \\ 0}}{\phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}), [])} \quad - \quad \underset{\substack{\| \\ \texttt{length}\,\texttt{r}}}{\phi([], \texttt{r})} \\
 & = & 1
\end{array}
$$

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rcl}
a & = & \underbrace{t}_{\substack{\texttt{length r} + 1 \\ \uparrow \\ \text{invert } \texttt{r} \text{ and take the head}}} + \underbrace{\phi(\texttt{tail}(\texttt{reverse r}), [])}_{0} - \underbrace{\phi([], \texttt{r})}_{\texttt{length r}} \\
& = & 1
\end{array}
$$

The amortized cost is 1 in all cases of `extract`

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rcccccc}
a & = & t & + & \phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}),[]) & - & \phi([],\texttt{r}) \\
 & & \| & & \| & & \| \\
 & & \texttt{length}\,\texttt{r}+1 & & 0 & & \texttt{length}\,\texttt{r} \\
 & & \uparrow & & & & \\
 & & \text{invert } \texttt{r} \text{ and take the head} & & & & \\
 & = & 1 & & & &
\end{array}
$$

The amortized cost is 1 in all cases of `extract`

Amortized cost of `snoc`:
It adds one element to the rear list:

- One actual step of computation
- The potential increases by one

So the amortized cost is 2.

Amortized Analysis of `extract`

- Third case: `extract ([],r)`

$$
\begin{array}{rcccccc}
a & = & \underset{\substack{\| \\ \texttt{length}\,\texttt{r} + 1 \\ \uparrow \\ \text{invert } \texttt{r} \text{ and take the head}}}{t} & + & \underset{\substack{\| \\ 0}}{\phi(\texttt{tail}(\texttt{reverse}\,\texttt{r}), [])} & - & \underset{\substack{\| \\ \texttt{length}\,\texttt{r}}}{\phi([], \texttt{r})} \\
 & = & 1 & & & &
\end{array}
$$

The amortized cost is 1 in all cases of `extract`

Amortized cost of `snoc`:
It adds one element to the rear list:

- One actual step of computation
- The potential increases by one

So the amortized cost is 2.

All operations have $O(1)$ amortized cost

Priority Queues - Heaps

Advanced Algorithms and Data Structures - Lecture 7-B

Venanzio Capretta

Monday 11 November 2019

School of Computer Science, University of Nottingham

Priority Queues

A priority queue or heap is a collection of keys/values that can be access in order. We are not interested in searching the collection. We just want to extract the minimum element efficiently.

Specification:

The type `Heap` must have the following methods:

Priority Queues

A priority queue or heap is a collection of keys/values that can be access in order. We are not interested in searching the collection. We just want to extract the minimum element efficiently.

Specification:
The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values

Priority Queues

A priority queue or heap is a collection of keys/values that can be access in order. We are not interested in searching the collection. We just want to extract the minimum element efficiently.

Specification:
The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty

Priority Queues

A priority queue or heap is a collection of keys/values that can be access in order. We are not interested in searching the collection. We just want to extract the minimum element efficiently.

Specification:
The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty
- `insert :: Key -> Heap -> Heap`
  Adds a new element to the heap

Priority Queues

A priority queue or heap is a collection of keys/values that can be access in order. We are not interested in searching the collection. We just want to extract the minimum element efficiently.

Specification:
The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty
- `insert :: Key -> Heap -> Heap`
  Adds a new element to the heap
- `minimum :: Heap -> Key`
  Returns the smallest element of the heap

Priority Queues

A priority queue or heap is a collection of keys/values that can be access in order. We are not interested in searching the collection. We just want to extract the minimum element efficiently.

Specification:
The type `Heap` must have the following methods:

- `empty :: Heap`
  The heap containing no values
- `isEmpty :: Heap -> Bool`
  Tests if the heap is empty
- `insert :: Key -> Heap -> Heap`
  Adds a new element to the heap
- `minimum :: Heap -> Key`
  Returns the smallest element of the heap
- `extract :: Heap -> (Key,Heap)`
  Removes the smallest element and returns it together with the rest of the heap

Extra Operations

We may also need a function to merge two heaps into one
(used as auxiliary to other methods, for example extraction):

```
union :: Heap -> Heap -> Heap
```

Another useful operation consists in decreasing a key in the heap:

```
decreaseKey :: Heap -> Element -> Key -> Heap
```

Assuming that the new key is smaller than the key of the element

This is useful when using heaps for the queue in Dijktra's algorithm
(the relaxation step)

Inefficient Realizations

We will see several implementation of the heap specification and analyze the complexity of the methods

The first naive instantiations can be:

- (Unordered) Lists
- Ordered Lists
- Binary Search Trees

Inefficient Realizations

We will see several implementation of the heap specification and analyze the complexity of the methods

The first naive instantiations can be:

- (Unordered) Lists
- Ordered Lists
- Binary Search Trees

Implementing heaps as (unordered) lists we have:

- `empty = []`, complexity $\Theta(1)$
- `isEmpty`: test if the list is `[]`, complexity $\Theta(1)$
- `insert x h = x :: h`, complexity $\Theta(1)$
- `minimum`
  Search the list for the least element, complexity $\Theta(n)$
- `extract`
  Find minimum, relink the parts before and after it, complexity $\Theta(n)$

Ordered Lists and Binary Search Trees

Ordered Lists and Binary Search Trees

- Implementing heaps as ordered lists:
  The minimum is now the first element of the list: `minimum` and `extract` can be done in $\Theta(1)$
  But `insert` must put the new element in the right place, the complexity is $\Theta(n)$

Ordered Lists and Binary Search Trees

- Implementing heaps as ordered lists:
  The minimum is now the first element of the list: `minimum` and `extract` can be done in $\Theta(1)$
  But `insert` must put the new element in the right place, the complexity is $\Theta(n)$
- Implementing heaps as (balanced) binary search trees `insert`, `minimum`, and `extract` can be done in $O(\log n)$

Ordered Lists and Binary Search Trees

- Implementing heaps as ordered lists:
  The minimum is now the first element of the list: `minimum` and `extract` can be done in $\Theta(1)$
  But `insert` must put the new element in the right place, the complexity is $\Theta(n)$
- Implementing heaps as (balanced) binary search trees `insert`, `minimum`, and `extract` can be done in $O(\log n)$

Since we're not interested in searching the heap, but only in finding the minimum, binary search trees are an overkill. We will see:

- (Leftist) Heaps:
  `minimum` in $\Theta(1)$; `insert` and `extract` in $\Theta(\log n)$
- Fibonacci Heaps:
  `minimum`, `insert`, `union`, `decrease` in $O(1)$ amortized complexity; `extract` in $O(\log n)$

Heap Sort

Using Heaps for Sorting:

There is a correspondence between realizations of the heap data structure and sorting algorithms

$$
\begin{tikzcd}
\text{Lists} & & \text{Heaps} \\
l \arrow[rr, bend left, "\texttt{listHeap}"] \arrow[dd, dotted, "\texttt{sort}"] & & \\
& & h \arrow[dll, bend left, "\texttt{heapList}"] \\
l' & &
\end{tikzcd}
$$

Conversion Between Lists and Heaps

```
listHeap :: [Key] → Heap
listHeap [] = empty
listHeap (x:xs) = insert x (listHeap xs)
```

```
heapList :: Heap → [Key]
heapList h = if (isEmpty h)
             then []
             else let (x,h') = extract h
                  in (x:heapList h')
```

These are generic conversion functions

For specific heap implementations there may be more efficient algorithms

```
sort :: [Key] → [Key]
sort = heapList ∘ listHeap
```

For some implementations of heaps there may be ad hoc versions of `listHeap` and `heapList` that are more efficient

- For unordered lists, `listHeap` is just the identity
- For ordered lists, `heapList` is just the identity

For some implementations of heaps there may be ad hoc versions of `listHeap` and `heapList` that are more efficient

- For unordered lists, `listHeap` is just the identity
- For ordered lists, `heapList` is just the identity

Different heap realizations correspond to different sorting algorithms

- With unordered lists, we get selection sort
- With ordered lists, we get insertion sort

Binary Heaps

We implement heaps as binary trees

The minimum element will always be at the root of the tree

Binary Heaps

We implement heaps as binary trees

The minimum element will always be at the root of the tree

- Heap Property: The key in every node is smaller or equal to all the keys in its children

Binary Heaps

We implement heaps as binary trees

The minimum element will always be at the root of the tree

- Heap Property: The key in every node is smaller or equal to all the keys in its children
- Balance: The number of elements in the left and right children differ by at most one

Binary Heaps

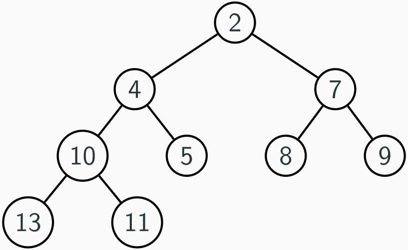
We implement heaps as binary trees

The minimum element will always be at the root of the tree

- Heap Property: The key in every node is smaller or equal to all the keys in its children
- Balance: The number of elements in the left and right children differ by at most one

Stronger condition in AI - the tree is complete: every lever, except the last, is full, and elements in the last level are as far left as possible

A complete heap:

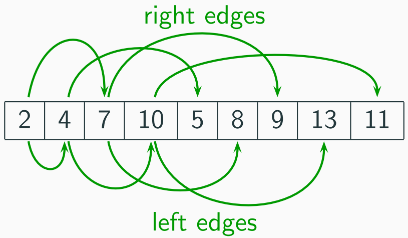


Implementation as an Array

Complete Binary Heaps can be easily implemented as arrays, with the parent-child link implicit (by indexing), instead of using pointers

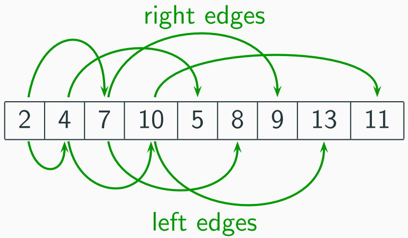
Implementation as an Array

Complete Binary Heaps can be easily implemented as arrays, with the parent-child link implicit (by indexing), instead of using pointers



Implementation as an Array

Complete Binary Heaps can be easily implemented as arrays, with the parent-child link implicit (by indexing), instead of using pointers



This is an example of implicit data structure

It makes it easy to add an element *"at the end"*, which is needed for insertion, and *"get the last element"*, which is needed for elimination.

Functional Realization

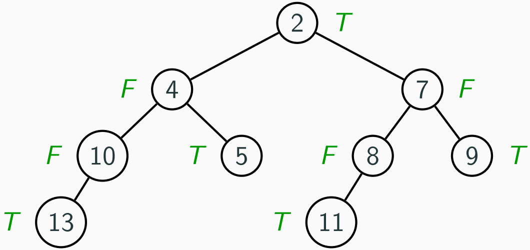
In functional programming, where tree structures are more natural, we use balanced trees.

Each node has a Boolean Flag:

- true if left and right children have the same number of elements
- false if the left child has one more element than the right

```
data BinTree = Empty
             | Node Bool Key BinTree BinTree
```

Functional Example



Formally:

`Node True 2 (Node False 4 ...)  (Node False 7 ...)`

(Leaves are not shown)

It's slightly different from the previous (complete) version:

To keep the balance, 11 is on the right side

Insertion

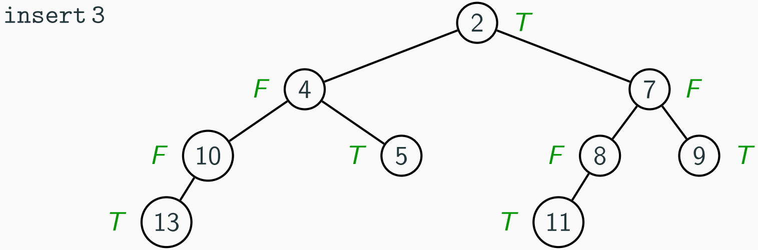
To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

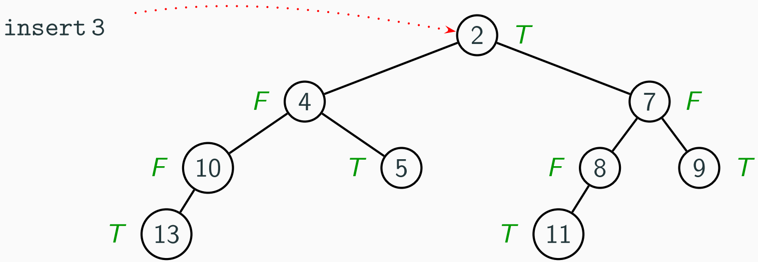


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

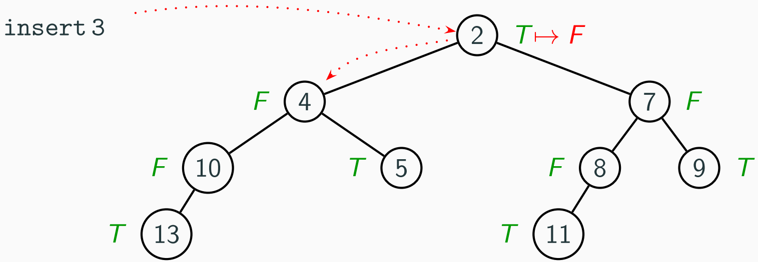


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

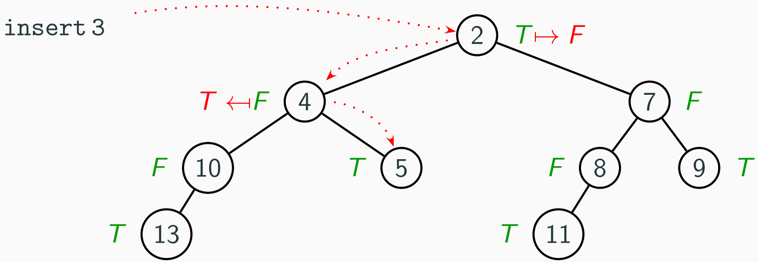


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

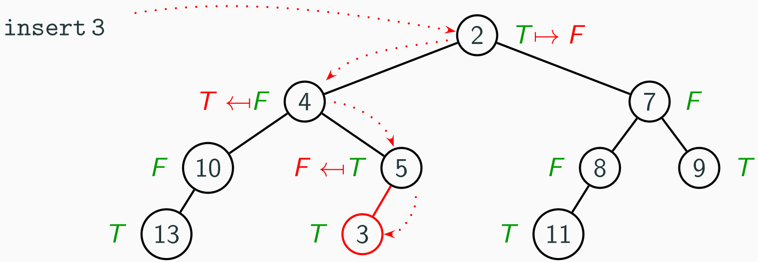


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Insertion

To insert a new element: (Similar for imperative version)

- First place it at the bottom, preserving the balance
- Then move it up, swapping it with higher elements that are bigger

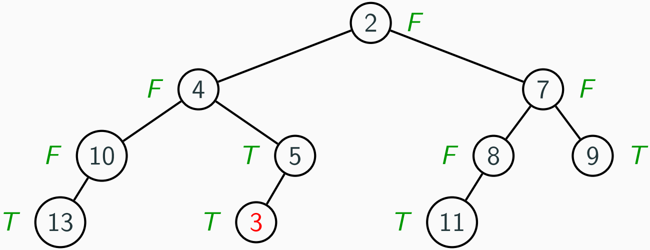


- If the node is true, go left
- If the node is false, go right
- Flip the Boolean flag

Swapping

We must now fix the Heap Property:

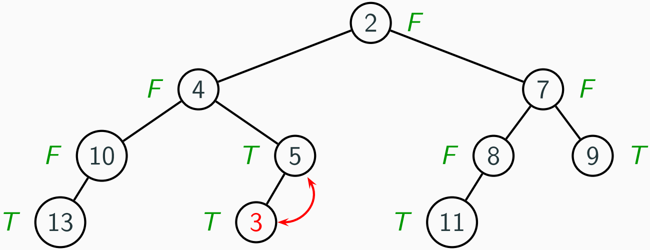
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

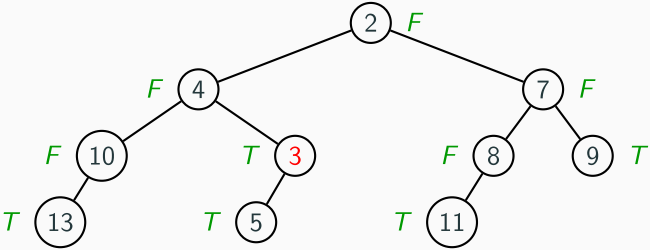
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

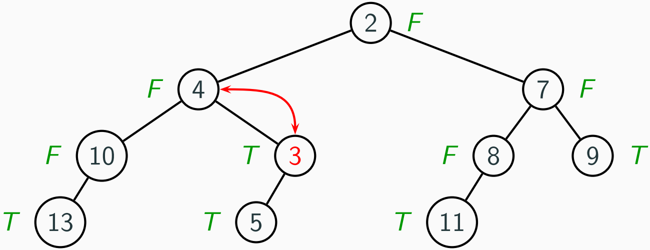
Move the new element upward until it gets to the right place



Swapping

We must now fix the Heap Property:

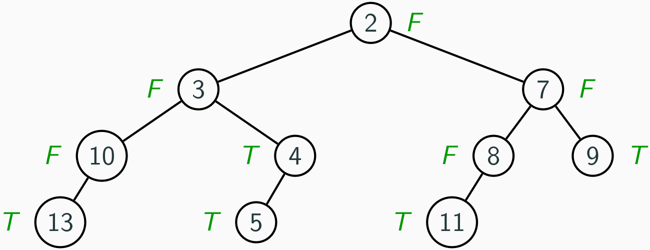
Move the new element upward until it gets to the right place



Swapping

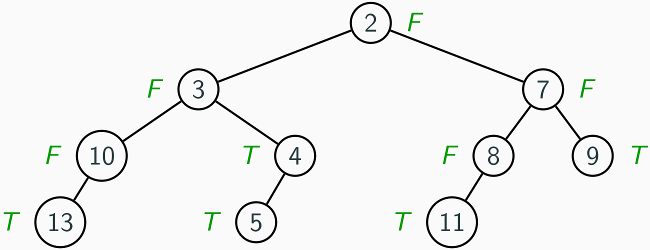
We must now fix the Heap Property:

Move the new element upward until it gets to the right place



Swapping

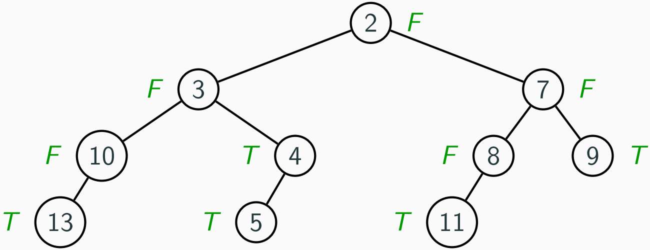
We must now fix the Heap Property:
Move the new element upward until it gets to the right place



In functional programming we can optimize the two phases:
Do only one pass of the tree, swap as you move down

Swapping

We must now fix the Heap Property:
Move the new element upward until it gets to the right place



In functional programming we can optimize the two phases:
Do only one pass of the tree, swap as you move down

Complexity: The depth of the tree is $O(\log n)$
So the complexity of `insert` is $O(\log n)$

Haskell Version

The previous version is good for the array implementation
Placing the element *"at the bottom"* is easy:
Just add it at the end of the array

In functional programming, we can do the swapping as we descend the tree:

```
insert :: Key → BinHeap → BinHeap
insert x Empty = Node True x Empty Empty
insert x (Node True y h1 h2) = Node False (min x y)
                                   (insert (max x y) h1)
                                   h2
insert x (Node False y h1 h2) = Node True (min x y)
                                    h1
                                    (insert (max x y) h2)
```

`minimum` is just the root of the tree

Extraction

`extract` is more complicated:

- We extract the minimum, which is the root
- We are left with the two children; We need to merge them

Extraction

`extract` is more complicated:

- We extract the minimum, which is the root
- We are left with the two children; We need to merge them

Idea:

- Recursively extract the minimum from one of the children (choose which so the balance is preserved)
- Use it as the new root
- Swap it down inside the other child if necessary

Extraction

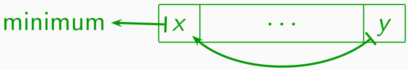
`extract` is more complicated:

- We extract the minimum, which is the root
- We are left with the two children; We need to merge them

Idea:

- Recursively extract the minimum from one of the children (choose which so the balance is preserved)
- Use it as the new root
- Swap it down inside the other child if necessary

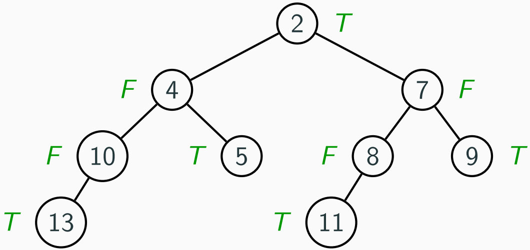
(In the array representation, choose the last element as the new root:



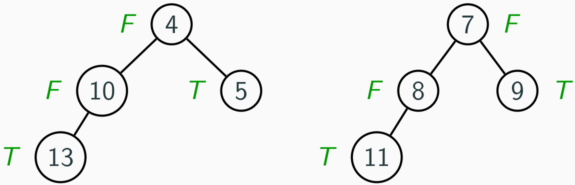
Then move it down if necessary)

Example Extraction

Extract the minimum from this tree:

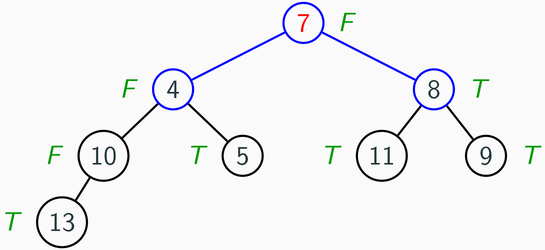


The minimum is 2; we are left with the children



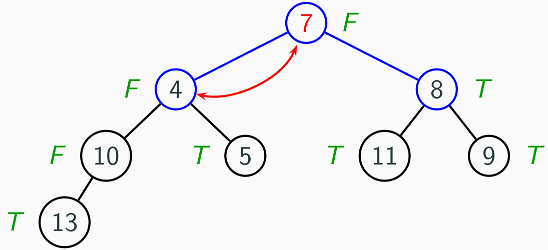
Merging

Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Merging

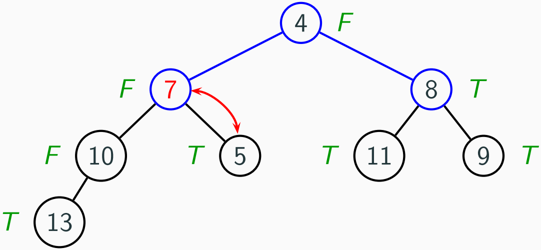
Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Then we *swap down* the new root to move it to the right place

Merging

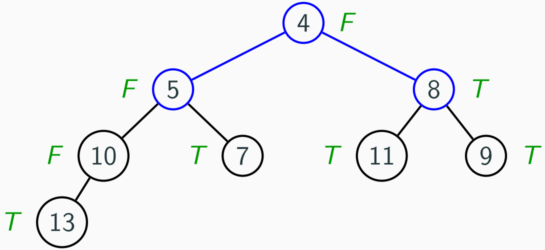
Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Then we *swap down* the new root to move it to the right place

Merging

Since the two trees have the same number of elements
(the Boolean value at the root was true)
we recursively extract from the right tree
and use its minimum as the new root:



Then we *swap down* the new root to move it to the right place

Swapping Function

Auxiliary function to move an element down to the right position:

```
siftDown :: Key → BinHeap → (Key,BinHeap)
siftDown x Empty = (x, Empty)
siftDown x h@(Node b y h1 h2) =
  if x>y then (y,downHeap (Node b x h1 h2))
         else (x,h)

downHeap :: BinHeap → BinHeap
downHeap Empty = Empty
downHeap (Node b x h1 h2) =
  if h1 ≤ h2
  then let (x',h1') = siftDown x h1
       in (Node b x' h1' h2)
  else let (x',h2') = siftDown x h2
       in (Node b x' h1 h2')
```

The order relation $\leq$ on trees is true if the root of `h1` is smaller than the root of `h2` or `h2` is empty (if `h1` empty, then `h2` empty, by balance)

Extraction in Haskell

Finally we can implement extraction:

```
extract :: BinTree → (Key,BinTree)
extract (Node b x Empty Empty) = (x,Empty)
extract (Node True x h1 h2) =
  let (y,h2') = extract h2
      (z,h1') = siftDown y h1
  in  (x, Node False z h1' h2')
extract (Node False x h1 h2) =
   ...     (similar to previous case)
```

Complexity

The complexity of `siftDown` and `downHeap` is the same:
`siftDown` just calls `downHeap` after making a constant-time operation

$$T_0(n) = T_0(n/2) + c_0$$

- The recursive call to `siftDown` is on either `h1` or `h2` which have half of the elements (plus or minus 1)
- At each call we do a constant number of extra steps
- Therefore $T_0(n) = \Theta(\log n)$

Complexity

The complexity of `siftDown` and `downHeap` is the same:
`siftDown` just calls `downHeap` after making a constant-time operation

$$T_0(n) = T_0(n/2) + c_0$$

- The recursive call to `siftDown` is on either `h1` or `h2` which have half of the elements (plus or minus 1)
- At each call we do a constant number of extra steps
- Therefore $T_0(n) = \Theta(\log n)$

Complexity of `extract`:

$$T_1 = T_1(n/2) + T_0(n/2) + c_1 = T_1(n/2) + \Theta(\log n)$$

- The call to `siftDown` gives the term $T_0(n/2)$
- Therefore $T_1(n) = \Theta(\log n)$

Fibonacci Heaps

Advanced Algorithms and Data Structures - Lecture 8

Venanzio Capretta

Monday 18 November 2019

School of Computer Science, University of Nottingham

Complexity of Heap Operations

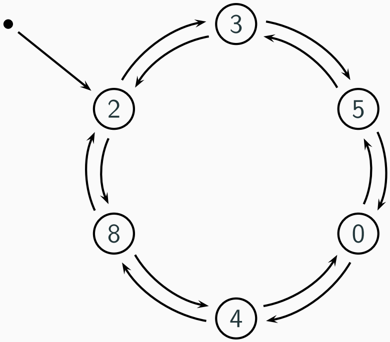
Summary of the complexity of basic heap operation
for several kinds of heaps:
(Fibonacci Heap extraction is amortized complexity)

	`insert`	`minimum`	`extract`	`union`
`Binary`	$O(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
`Leftist`	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
`Binomial`	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
`Fibonacci`	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$

Amortized Complexityi s a method of analyzing the running time
that takes into account a whole sequence of operations:
Some operation have a long running time, some have a short running time
Amortized complexity refers to the average time of the whole sequence

Wheels

As a first step towards the definition of Fibonacci Heaps
we study doubly-linked circular lists (wheels)



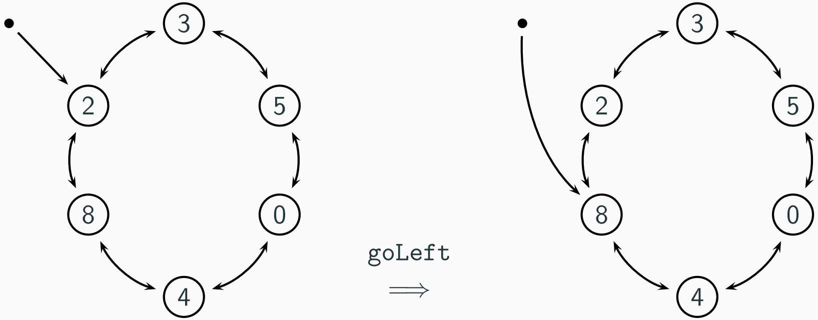
(We use the notation $\lbrack\!\!\phi\, 2, 3, 5, 0, 4, 8 \,\phi\!\!\rbrack$)

A sequence of values linked in a circle with a head pointer
to one value (2 in the example)
and operations to move the pointer, insert and delete elements

Wheel Operations: Move

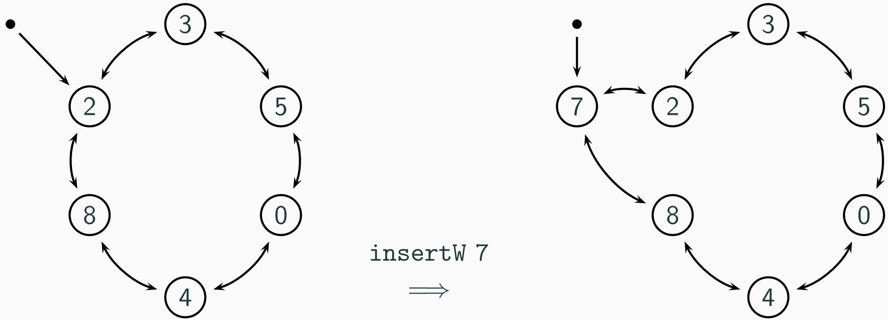
goRight and goLeft

move the head pointer clockwise (to 3) or anti-clockwise (to 8):



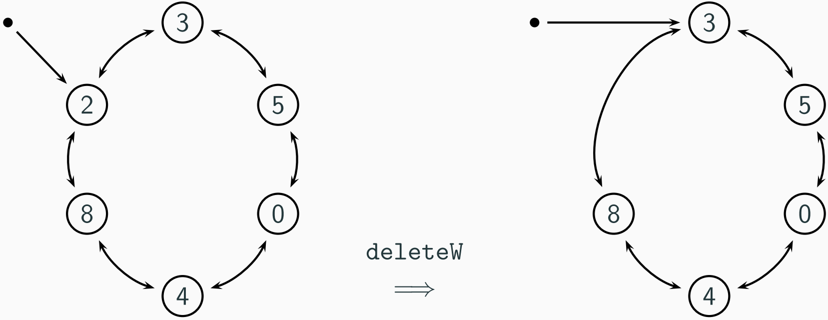
Wheel Operations: Insert

`insert` a new element just before the pointer:



Wheel Operations: Delete

`delete` the element pointed at:
(and move the pointer clockwise)



Implementation of Wheels

In imperative programming, you can implement wheels using pointers going back and forth between every pair of elements.
The complexity of each operation is $\Theta(1)$.

In functional programming, you can implement wheels by a pair of lists, similarly to what we have done for FIFO queues.
All operations can be programmed with amortized complexity $\Theta(1)$.

Hint: Since we can move both left and right, the *best* state of the structure is when both lists have the same length. This should be the state with the highest potential.

Fibonacci Heaps

A Fibonacci Heap is a kind of fractal wheel:

Fibonacci Heaps

A Fibonacci Heap is a kind of fractal wheel:
It's a wheel of values in which every element is in turn connected with another Fibonacci heap

The structure is similar to a Binary or Leftist Heap in which each node is replaced by a wheel

Fibonacci Heaps

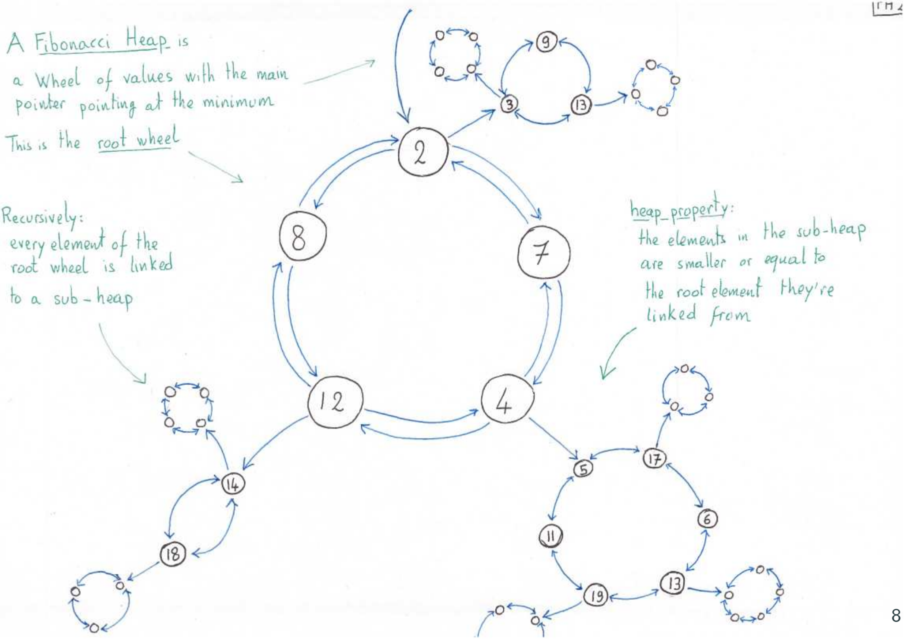
A Fibonacci Heap is a kind of fractal wheel:
It's a wheel of values in which every element is in turn connected with another Fibonacci heap

The structure is similar to a Binary or Leftist Heap in which each node is replaced by a wheel

Formally a Fibonacci Heap has:

- A root wheel of values, with the main pointer pointing to the smallest of them
  The values don't need to be ordered
- Each element of the root wheel is in turn connected to a sub-heap
  (The sub-heap could be empty)
- Heap Property: The elements of the sub-heap are larger or equal to the root element they're linked from

A Fibonacci Heap



Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

When we delete an element,
we must traverse the wheel to find the new minimum
We take advantage of this traversal to "consolidate" the heap

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

When we delete an element,
we must traverse the wheel to find the new minimum
We take advantage of this traversal to "consolidate" the heap

The degree of an element is
the length of the main wheel of the sub-heap it is linked to

A heap is consolidated if all its root elements have different degrees

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

When we delete an element,
we must traverse the wheel to find the new minimum
We take advantage of this traversal to "consolidate" the heap

The degree of an element is
the length of the main wheel of the sub-heap it is linked to

A heap is consolidated if all its root elements have different degrees

Q: If the heap and all the sub-heaps are consolidated,
What is the minimum number of elements the heap
as a function of the length of the root wheel?

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

The example in the previous page is written:

$$\mathtt{FHeap}\ \lbrace\!\!\lbrace (2,3,\mathtt{h}_1),(7,0,\mathtt{emptyW}),(4,6,\mathtt{h}_2),(12,2,\mathtt{h}_3),(8,0,\mathtt{emptyW}) \rbrace\!\!\rbrace$$

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

The example in the previous page is written:

$$\mathtt{FHeap}\ \lbag (2, 3, \mathtt{h}_1), (7, 0, \mathtt{emptyW}), (4, 6, \mathtt{h}_2), (12, 2, \mathtt{h}_3), (8, 0, \mathtt{emptyW}) \rbag$$

The head element is 2, it has degree 3, because its sub-heap h_1 has three elements in its root wheel

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

The example in the previous page is written:

$$\mathtt{FHeap} \lbrace (2,3,\mathtt{h}_1),(7,0,\mathtt{emptyW}),(4,6,\mathtt{h}_2),(12,2,\mathtt{h}_3),(8,0,\mathtt{emptyW}) \rbrace$$

The head element is 2, it has degree 3, because its sub-heap h_1 has three elements in its root wheel

The element 7 has degree 0 because its sub-heap is empty

(This is not a consolidated heap: Two nodes with the same degree 0)

Heap Operations

- The empty heap simply has an empty wheel:

```
emptyH = FHeap ⟦ ⟧
```

Heap Operations

- The empty heap simply has an empty wheel:

```
emptyH = FHeap ⦃ ⦄
```

- The minimum is just the head of the wheel:

```
minimum (FHeap w) = first (headW w)
```

Heap Operations

- The empty heap simply has an empty wheel:

```
emptyH = FHeap ⦃ ⦄
```

- The minimum is just the head of the wheel:

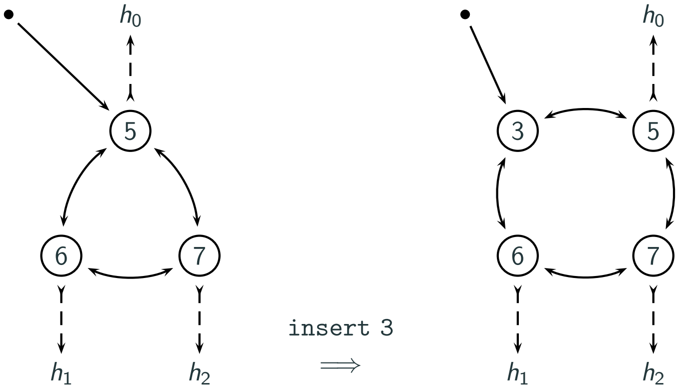
```
minimum (FHeap w) = first (headW w)
```

- Insertion adds the new element to the wheel with empty sub-heap, but must move the head right if the inserted element is bigger than the previous head:

```
insertH x h@(FHeap w) =
  if (isEmptyW w)
  then FHeap ⦃(x, 0, emptyH)⦄
  else if x ≤ minimum h
       then FHeap (insertW (x,0,emptyH) w)
       else FHeap (goRight (insertW (x,0,emptyH) w))
```

Insertion Example 1

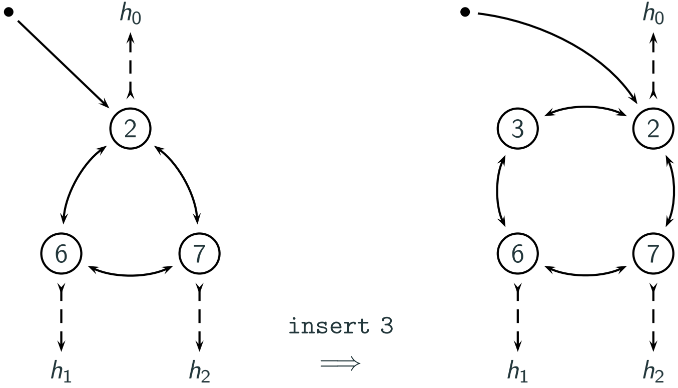
For example, if we insert 3 into a heap with minimum 5
3 becomes the new minimum:



Insertion Example 2

But if we insert 3 into a heap with minimum 2

2 remains the minimum:

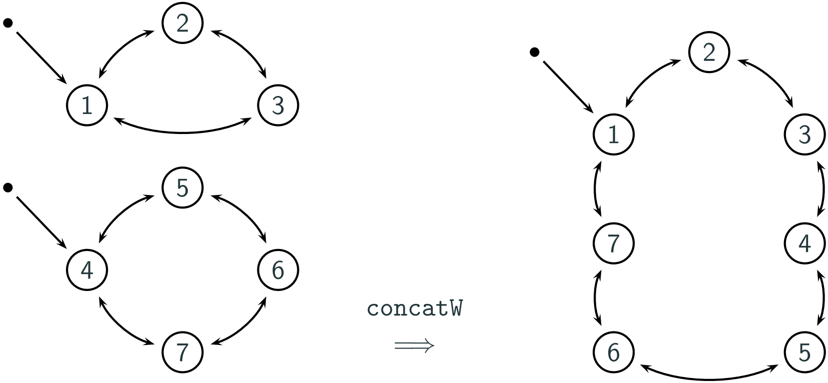


Union 1

Union of two heaps is done by simply concatenating the corresponding wheels

We must make sure that the head pointer points to the minimum of the heads of the two heaps

Exercise: Implement the concatenation of two wheels:



Union 2

```
union h1@(FHeap w1) h2@(FHeap w2) =
  if isEmpty h1 then h2 else
  if isEmpty h2 then h1 else
  if minimum h1 ≤ minimum h2
    then FHeap (concatW w1 w2)
    else FHeap (concatW w2 w1)
```

We compare the minimums of the two heaps
and we concatenate the wheels so that
the smaller one becomes the new minimum

Union 2

```
union h1@(FHeap w1) h2@(FHeap w2) =
  if isEmpty h1 then h2 else
  if isEmpty h2 then h1 else
  if minimum h1 ≤ minimum h2
    then FHeap (concatW w1 w2)
    else FHeap (concatW w2 w1)
```

We compare the minimums of the two heaps
and we concatenate the wheels so that
the smaller one becomes the new minimum

We implemented most heap operations
in a naive way, without worrying about the structure of the heap

The only operation that rearranges the heap is extraction

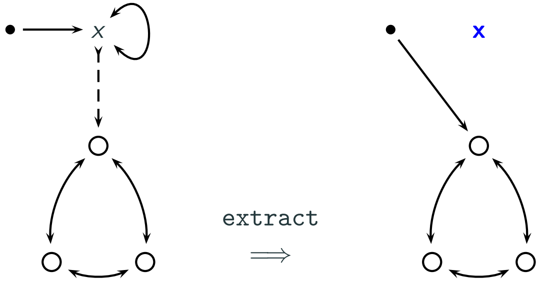
Extraction 1

When we extract the minimum from a heap:

Extraction 1

When we extract the minimum from a heap:

- If it was the only element of the wheel, the new heap is its sub-heap



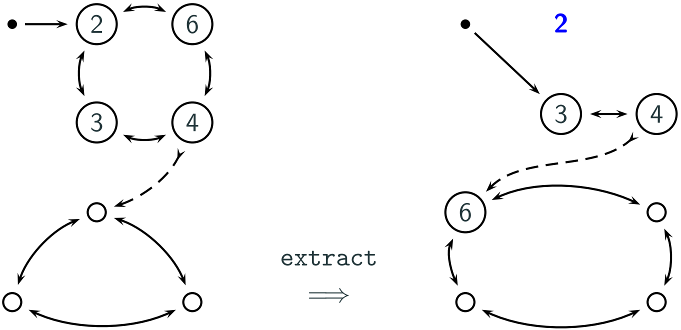
Extraction 2

- If there are other elements in the root wheel, we must
  - Concatenate the sub-heap of the extracted element with the remaining root wheel
  - Traverse the root wheel to find the new minimum
  - Take advantage of this traversal to restructure (consolidate) the heap

Extraction 2

- If there are other elements in the root wheel, we must
  - Concatenate the sub-heap of the extracted element with the remaining root wheel
  - Traverse the root wheel to find the new minimum
  - Take advantage of this traversal to restructure (consolidate) the heap

It can happen that an element that was originally on the root wheel is moved into some of the sub-heaps:



Extraction 3

So extraction removes the minimum, concatenates its sub-heap with the root wheel, and then consolidates:

```
extract :: FibHeap → (Key,FibHeap)
extract (FHeap w) =
  let ((x,FHeap wx), w’) = extractW w
  in  (x, consolidate (FHeap (concatenateW wx w’)))
```

(The code is sketchy, to make it work you must add a couple of details)

Consolidation consists in reorganizing the structure of the heap while at the same time finding the new minimum.

Consolidation 1

We use an array A in which we place
the nodes/sub-heaps from the root wheel

$A[d]$ will contain either nothing or a single node/sub-heap with degree d

Consolidation 1

We use an array A in which we place
the nodes/sub-heaps from the root wheel

$A[d]$ will contain either nothing or a single node/sub-heap with degree d

If we try to add a new node/heap to $A[d]$, but the place is already taken we link the two nodes/heaps into a single one with degree $d+1$ and try to put it in $A[d+1]$ (if that place is free)

Consolidation 1

We use an array A in which we place
the nodes/sub-heaps from the root wheel

$A[d]$ will contain either nothing or a single node/sub-heap with degree d

If we try to add a new node/heap to $A[d]$, but the place is already taken we link the two nodes/heaps into a single one with degree $d+1$ and try to put it in $A[d+1]$ (if that place is free)

Define a type of nodes/sub-heap that explicitly contains the degree:

```
type Node = (Key,Int,FibHeap)
```

Note: then a Fibonacci Heap can be defined just as a wheel of nodes:

```
data FibHeap = FHeap (Wheel Node)
```

Consolidation 2

Linking two nodes:

insert the larger one as child of the smaller

```
link :: Node → Node → Node
link x@(kx,dx,hx) y@(ky,dy,hy) =
  if kx ≤ ky
  then (kx, dx+1, FHeap (insertN y hx))
  else (ky, dx+1, FHeap (insertN x hy))
```

(`insertN` should insert the node with its subheap,
making sure to still point at the minimum)

Consolidation 2

Linking two nodes:

insert the larger one as child of the smaller

```
link :: Node → Node → Node
link x@(kx,dx,hx) y@(ky,dy,hy) =
  if kx ≤ ky
  then (kx, dx+1, FHeap (insertN y hx))
  else (ky, dx+1, FHeap (insertN x hy))
```

(`insertN` should insert the node with its subheap,
making sure to still point at the minimum)

Now we use an array A of nodes

(In functional programming we can use Finite Maps)

Let us call `NArray` its type

Consolidation 2

Linking two nodes:

insert the larger one as child of the smaller

```
link :: Node → Node → Node
link x@(kx,dx,hx) y@(ky,dy,hy) =
  if kx ≤ ky
  then (kx, dx+1, FHeap (insertN y hx))
  else (ky, dx+1, FHeap (insertN x hy))
```

(`insertN` should insert the node with its subheap,
making sure to still point at the minimum)

Now we use an array A of nodes

(In functional programming we can use Finite Maps)

Let us call `NArray` its type

$A[d]$ is either empty or contains a node of degree d

Let us denote by $A[d \mapsto x]$

the array A where the entry $A[d]$ has been changed to x

Consolidation 3

Inserting a new node into the array will require checking if its degree is already taken:

```
insNA :: Node → NArray → NArray
insNA x@(kx,dx,hx) A =
  if A[dx] is undefined
    then A[dx ↦ x]
    else insNA (link x A[dx]) A
```

Note that if the degree `dx` in `A` is already occupied
We link `x` with the occupier `A[dx]` before inserting
We know this generates a new node of degree `dx+1`

Consolidation 3

We now transform a wheel of nodes into an array
by extracting and inserting them one by one

```
makeNA :: (Wheel Node) → NArray
makeNA w =
  if (isEmpty w)
    then emptyArray
    else let (x,w') = extractW w
         in  insNA x (makeNA w')
```

Consolidation 4

Once we have an array of nodes, stored by degree
we put them back together into a wheel

```
wheelNA :: NArray → (Wheel Node)
```

This works by starting from and empty wheel
and adding the elements from the array one by one
inserting them into the wheel with the following function
(Details depends on implementation,
with Haskell's finite maps we can use `foldr`)

Consolidation 4

Once we have an array of nodes, stored by degree
we put them back together into a wheel

```
wheelNA :: NArray → (Wheel Node)
```

This works by starting from and empty wheel
and adding the elements from the array one by one
inserting them into the wheel with the following function
(Details depends on implementation,
with Haskell's finite maps we can use `foldr`)

```
insNode x w =
  if (isEmpty w) or (x ≤ head w)
    then (insertW x w)
    else (goRight (insertW x w))
```

The last line guarantees that we are still pointing at the minimum

Consolidation 5

Finally we can put all the steps together:

```
consolidate :: FibHeap → FibHeap
consolidate (FHeap w) = wheelNA (makeNA w)
```

Consolidation 5

Finally we can put all the steps together:

```
consolidate :: FibHeap → FibHeap
consolidate (FHeap w) = wheelNA (makeNA w)
```

Complexity

All operations except `extract` are trivially $\Theta(1)$

We can show that `extract` runs in $O(\log n)$ amortized time
This depends on the relation between:

- the number elements of the heap
- the length of the root wheel

in a consolidated heap (see earlier exercise)

See IA for the definition of the potential function and the proof

Leftist Heaps

Summary of complexity of operations on Binary Heaps:

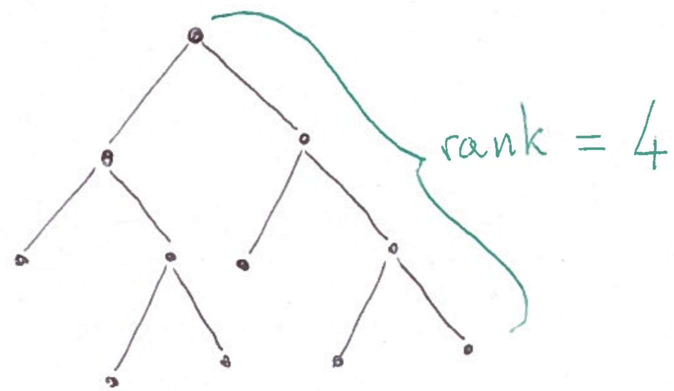
- insert: $O(\log n)$
- minimum: $O(1)$
- extract: $O(\log n)$

We haven't seen how to perform union.
The naive way is to extract the elements from one heap and insert them into the second.
The best algorithm can do it in $O(n)$.

Leftist Heaps are a structure for which we can do union in $O(\log n)$

rank of a binary tree: length of its rightmost spine

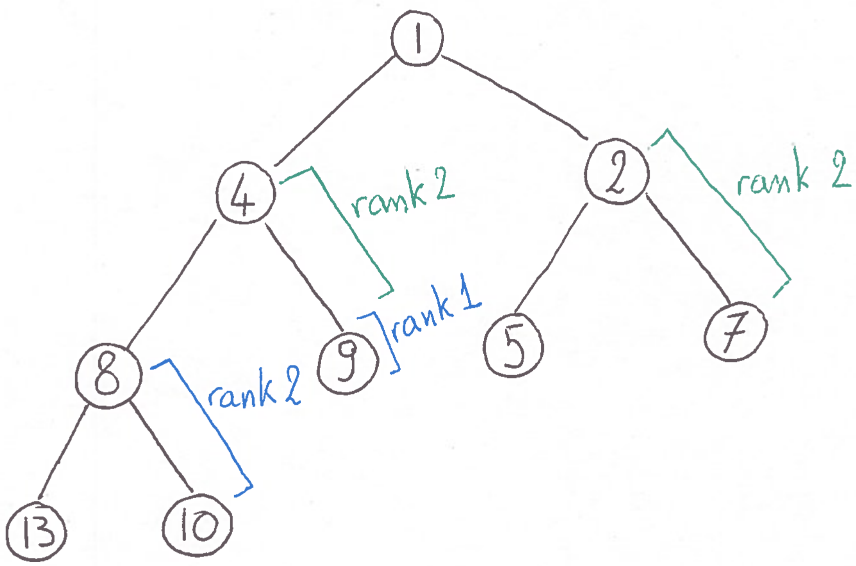
ex:



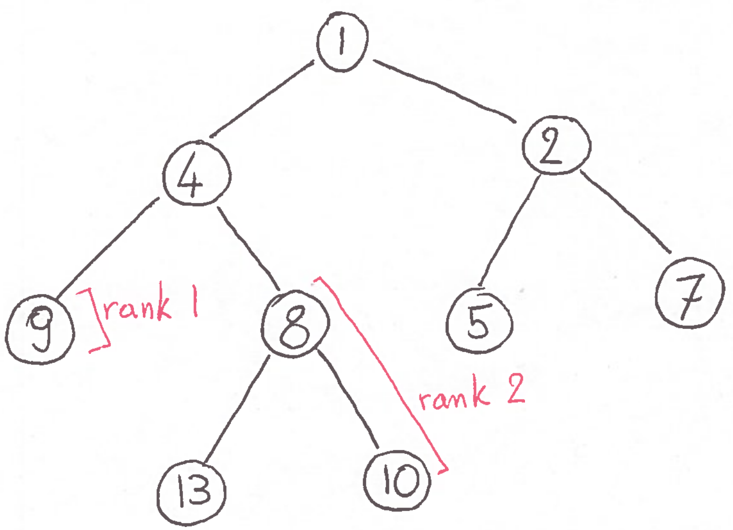
A leftist heap is a binary tree such that:

- It has the heap property: every node is smaller or equal to its children;
- For every node, the rank of the left child is larger or equal to the rank of the right child.

Example:
This is a correct Leftist Heap:



This is not a correct Leftist Heap:

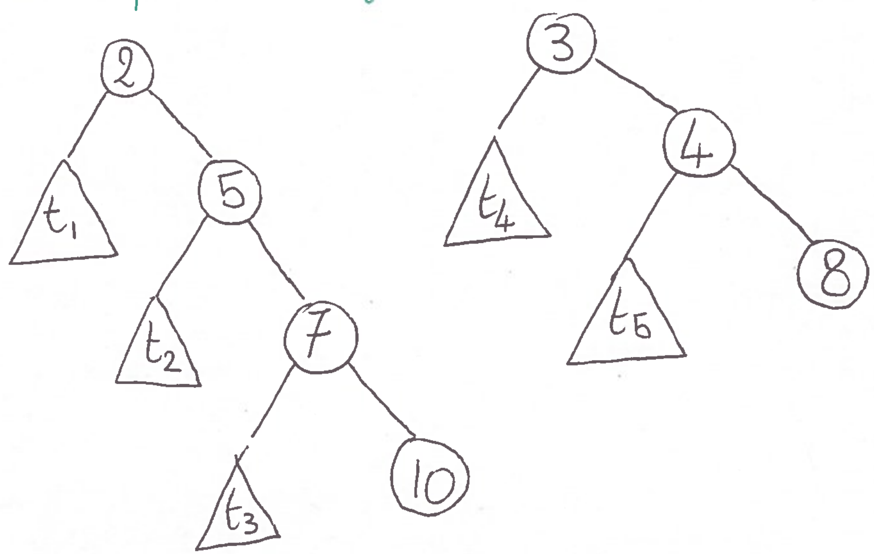


So a leftist heap is allowed to be unbalanced to the left.

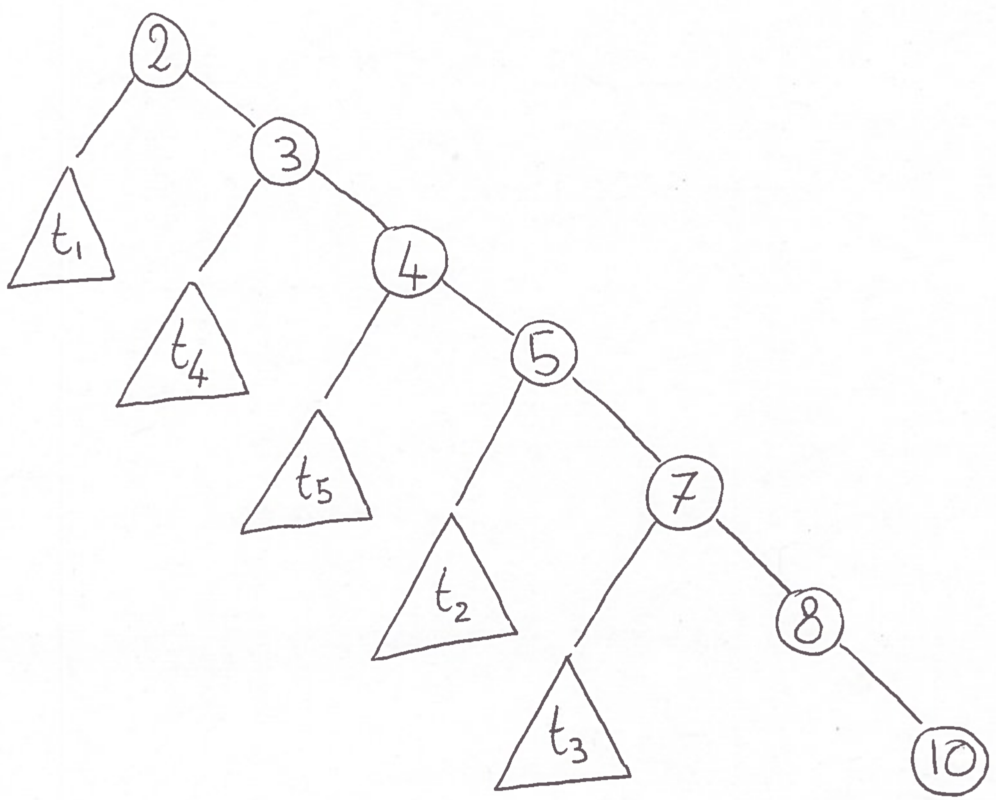
This makes it easy to implement union.

Idea:
- merge the two trees along the right spine (as in mergesort for list)
- then rearrange the children to preserve the leftist property: when the right child has higher rank than the left child, swap them

Example: Merge these two trees



Merge them along the right spines:



Then travel up the right spine and fix the violations of the leftist property by swapping children

We represent Leftists Heaps by keepint track of the rank in each node

```
data LHeap =
  Empty | Node N Key LHeap LHeap
```

N: a natural number giving the rank of the tree

The auxiliary function to "repair" the leftist property, constructs a Leftist tree from a key and the two children:

```
makeLH :: Key → LHeap → LHeap → LHeap
makeLH x h1 h2 =
  if (rank h1) ≥ (rank h2)
    then Node (rank h2+1) x h1 h2
    else Node (rank h1+1) x h2 h1
```

makeLH puts the child with the higher rank on the left, that with the lower rank on the right.

The new rank is one more than the rank of the right child.

Union of two Leftist Heaps:
merge them along the right spines
apply makeLH at each step:

```
union :: LHeap ⟶ LHeap ⟶ LHeap
union h1 Empty = h1
union Empty h2 = h2
union h1@(Node r1 x1 h11 h12) h2@(Node r2 x2 h21 h22)
  = if x1 ≤ x2
      then makeLH x1 h11 (union h12 h2)
      else makeLH x2 h21 (union h1 h22)
```

What is the complexity of union?

makeLH has complexity $O(1)$
because it performs a fixed set of operations with no recursive calls

union makes one recursive call where
- one of the arguments remains the same
- the other is replaced by its right child.

So union does recursion along the right spine. The length of the right spine is the rank.

Observation:
Because of the leftist property, the rank is $O(\log n)$
(Exercise: prove this)

So the complexity of union is $O(\log n)$
(if n is a bound of the size of the two arguments)

Other operations can be defined in terms of union, so they also have complexity $O(\log n)$

```
insert x h = union (Node 1 x Empty Empty) h

extract (Node r x h1 h2) =
        (x, union h1 h2)
```