Exercise Session 07 – Functors, Lambdas

Data Structures and Algorithms

These slides are based on those of the course, but were adapted and extended by the teaching assistant Adel Gavranović

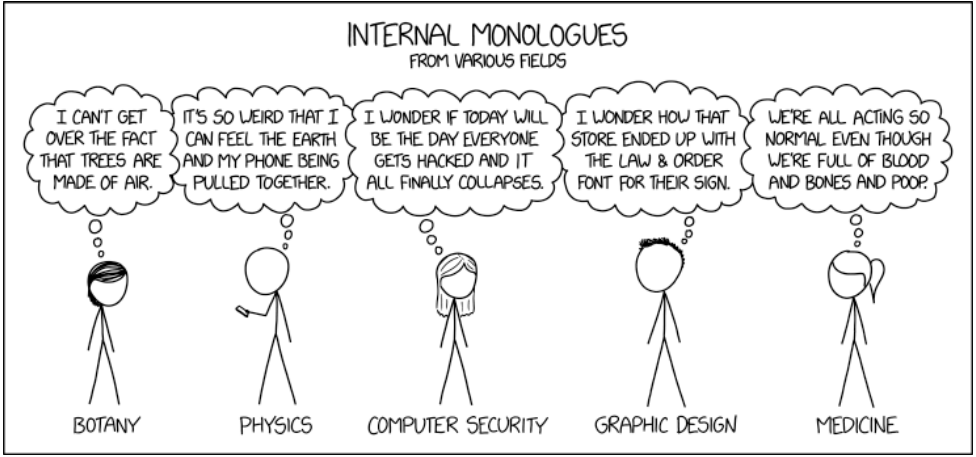
Today's Schedule

- Intro
- Follow-up
- Learning Objectives
- Quadtrees
- Code-Example
- Higher Order Functions
  - Function Signature Notation
- Tree Recap
  - Recap 2-3 Trees
  - Recap Red-Black Trees
- Outro

n.ethz.ch/~agavranovic

- Exercise Session Material
- Adel's Webpage
- Mail to Adel

Comic of the Week



▸ xkcd

1. Intro

Intro

Intro

- Welcome back!

2. Follow-up

Follow-up from last exercise session

Follow-up from last exercise session

- Red-Black tree from last session (S06)

Questions regarding **code** expert from your side?

3. Learning Objectives

Learning Objectives

∈ {High Ord function}

Quadtrees

- ☐ Understand what *quadtrees* are where they are used
- ☐ Understand the minimization problem behind *quadtrees*

Red-Black Trees

- ☐ Be able to perform basic operations on the most common trees, in particular Red-Black trees

Functors

- ☐ Know what *Functors* are
- ☐ Understand how *Functors* work
- ☐ Know where and how to use *Functors*

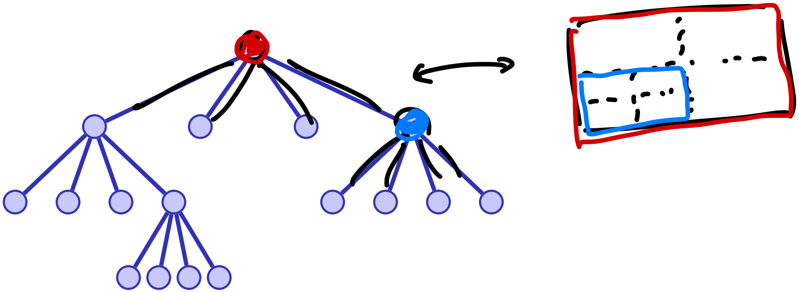
Lambdas

- ☐ Know what *Lambdas* are
- ☐ Understand how *Lambdas* work
- ☐ Know where and how to use *Lambdas*

4. Quadtrees

Quadtrees

Quadtrees are trees where each node has at most **four** children.



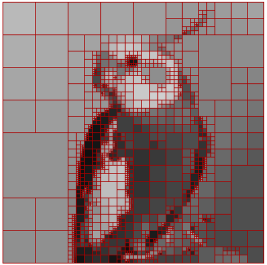
Main application: Image processing.

Quadtrees for Image Compression

Insight: (1) Divide image recursively into four regions, (2) map the regions to nodes in a quadtree und (3) assign each leaf the average color of its region.

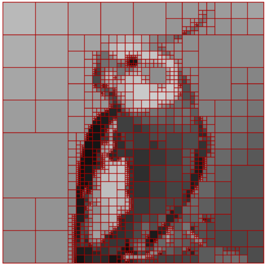
Quadtrees for Image Compression

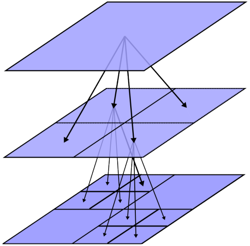
Insight: (1) Divide image recursively into four regions, (2) map the regions to nodes in a quadtree und (3) assign each leaf the average color of its region.

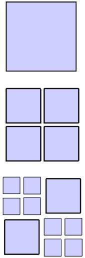


Quadtrees for Image Compression

Insight: (1) Divide image recursively into four regions, (2) map the regions to nodes in a quadtree und (3) assign each leaf the average color of its region.

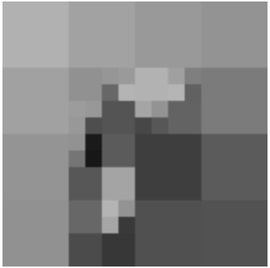






Quadtrees for Image Compression

When and where to stop the recursion?



Too early?

Quadtrees for Image Compression

When and where to stop the recursion?

Is this better?

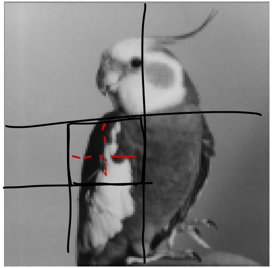
Quadtrees for Image Compression

When and where to stop the recursion?

Question: Should we stop when each node is mapped to a pixel?

Quadtrees for Image Compression

When and where to stop the recursion?



Answer: We would get the original image but gain no storage efficiency.

Quadtrees for Image Compression

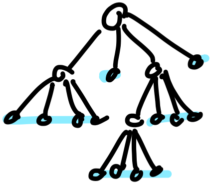
We want

- as close approximation as possible, und
- as few nodes as possible.

Quadtrees for Image Compression

We want

- as close approximation as possible, und
- as few nodes as possible.



This can be expressed as an optimization problem:

$$H_\gamma(T, \boldsymbol{y}) := \gamma \cdot \underbrace{|L(T)|}_{\text{Number of leaves}} + \underbrace{\sum_{r \in L(T)} \|\boldsymbol{y}_r - \boldsymbol{\mu}_r\|_2^2}_{\text{Cumulative approximation error of all leaves}}$$

where T is a quadtree, $\boldsymbol{y}$ is the image data, and $\gamma \geq 0$ is a regularization parameter. For a given γ we seek the optimal solution $\arg\min_T H_\gamma(T, \boldsymbol{y})$.

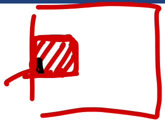
Quadtrees for Image Compression

we wanna minimize this!

$$H_\gamma(T, \boldsymbol{y}) := \gamma \cdot \underbrace{|L(T)|}_{\text{Number of leaves}} + \underbrace{\sum_{r \in L(T)} \|\boldsymbol{y}_r - \boldsymbol{\mu}_r\|_2^2}_{\text{Cumulative approximation error of all leaves}}$$

Question: What is the effect of a low value of γ?

Quadtrees for Image Compression



$$H_\gamma(T, \boldsymbol{y}) := \gamma \cdot \underbrace{|L(T)|}_{\text{Number of leaves}} + \underbrace{\sum_{r \in L(T)} \|\boldsymbol{y}_r - \boldsymbol{\mu}_r\|_2^2}_{\text{Cumulative approximation error of all leaves}}$$

Question: What is the effect of a low value of γ?

Answer: Improves the approximation at the expense of increasing the size of the quadtree.

Algorithm: Minimize($\boldsymbol{y}$,r,γ)

Input: Image data $\boldsymbol{y} \in \mathbb{R}^S$, rectangle $r \subset S$, regularization $\gamma > 0$.
Output: $\min_T \gamma|L(T)| + \|\boldsymbol{y} - \boldsymbol{\mu}_{L(T)}\|_2^2$

if $|r| = 0$ **then return** 0
$m \leftarrow \gamma + \sum_{s \in r}(y_s - \mu_r)^2$
if $|r| > 1$ **then**
  Split r into r_{ll},r_{lr},r_{ul},r_{ur}
  $m_1 \leftarrow \mathrm{Minimize}(y, r_{ll}, \gamma)$; $m_2 \leftarrow \mathrm{Minimize}(y, r_{lr}, \gamma)$
  $m_3 \leftarrow \mathrm{Minimize}(y, r_{ul}, \gamma)$; $m_4 \leftarrow \mathrm{Minimize}(y, r_{ur}, \gamma)$
  $m' \leftarrow m_1 + m_2 + m_3 + m_4$
else
  $m' \leftarrow \infty$
if $m' < m$ **then** $m \leftarrow m'$
return m

Code-Example

Quadtrees on **code** expert

- Region-Point Quadtree ← codeexpert code examples

6. Higher Order Functions

Motivation

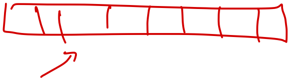
- Overarching goal: make code generic, thus reusable

Motivation

- Overarching goal: make code generic, thus reusable
- Templates so far: make code parametric in the data it operates on, e.g.
  - `Pair<T>` for all types `T`
  - `print<C>` for all iterable containers `C`

Motivation

- Overarching goal: make code generic, thus reusable
- Templates so far: make code parametric in the data it operates on, e.g.
  - `Pair<T>` for all types `T`
  - `print<C>` for all iterable containers `C`
- Now: make code parametric in the algorithms it uses, e.g.
  - `filter(container, predicate)`
  - `apply(signal, transformation/filter)`
  - `leader_election(participants, protocol)`
  - `navigation_system(map, shortest_path_algorithm)`
  - `Button("Save").onClick(handle_click_event)`



Callables and Higher-Order Functions

// return true if n is even
even(n)

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);

  return data;
}
```

Callables and Higher-Order Functions

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);

  return data;
}
```

- pred must be callable (applicable, invocable), i.e., something function-like

.... (n)

Callables and Higher-Order Functions

```cpp
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);

  return data;
}
```

- `pred` must be callable (applicable, invocable), i.e., something function-like
- In C++:
  - free or member function
  - lambda function
  - functor (object with `operator()`)
  - `std::function` object
  - function pointers [not discussed]

Callables and Higher-Order Functions

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;

  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);

  return data;
}
```

- `pred` must be callable (applicable, invocable), i.e., something function-like
- In C++:
  - free or member function
  - lambda function
  - functor (object with `operator()`)
  - `std::function` object
  - function pointers [not discussed]

Functions taking or returning functions are called **higher-order functions**.

C++Functors

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;

  AtLeast(T m): min(m) {};

  bool operator()(T i) const {
    return min <= i;
  }
};
```

C++Functors

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;

  AtLeast(T m): min(m) {};

  bool operator()(T i) const {
    return min <= i;
  }
};
```

- A functor
  - is it's an object that implements `operator()`
  - combines state (since an object) with callability (with the `operator()`)

creates funct.

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};
selection1 = filter(data, AtLeast(-1));
// = {-1,0,1,2,4,5}
selection2 = filter(data, AtLeast(4));
// = {4,5}
```

auto f = AtLeast(42); ; f(n);

C++Functors

```
// generic filter function
template <typename C, typename P>
C filter(const C& src_data, P pred) {
  C data;
  for (const auto& e : src_data)
    if (pred(e)) data.push_back(e);
  return data;
}
```

```
// stateful predicate as functor
template <typename T>
struct AtLeast {
  T min;

  AtLeast(T m): min(m) {};

  bool operator()(T i) const {
    return min <= i;
  }
};
```

- A functor
  - is it's an object that implements `operator()`
  - combines state (since an object) with callability (with the `operator()`)
- Objects of type `AtLeast<T>` are callable with one `T` argument.

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};
selection1 = filter(data, AtLeast(-1));
// = {-1,0,1,2,4,5}
selection2 = filter(data, AtLeast(4));
// = {4,5}
```

Lambda Expressions Translate to Functors

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};

auto selection1 = filter(data, [](int e) { return -2 <= e; });
auto selection2 = filter(data, [](int e) { return e != 0; });
```

```
struct lambda1 {
  bool operator()(int e) const {
    return -2 <= e;
  }
};

struct lambda2 {
  bool operator()(int e) const {
    return e != 0;
  }
};
```

Lambda Expressions Translate to Functors

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};

auto selection1 = filter(data, [](int e) { return -2 <= e; });
auto selection2 = filter(data, [](int e) { return e != 0; });
```

```
struct lambda1 {
  bool operator()(int e) const {
    return -2 <= e;
  }
};

struct lambda2 {
  bool operator()(int e) const {
    return e != 0;
  }
};
```

- C++compiler generates functors from lambda expressions

Lambda Expressions Translate to Functors

```
std::vector<int> data = {-1,0,1,2,-2,4,5,-3};

auto selection1 = filter(data, [](int e) { return -2 <= e; });
auto selection2 = filter(data, [](int e) { return e != 0; });
```

```
struct lambda1 {
  bool operator()(int e) const {
    return -2 <= e;
  }
};

struct lambda2 {
  bool operator()(int e) const {
    return e != 0;
  }
};
```

- C++compiler generates functors from lambda expressions
- Lambdas are not essential, but "merely" convenient

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{-> R}}_{\text{return type}} \; \underbrace{\{\ \texttt{stmt}\ \}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\{\ \texttt{stmt}\ \}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\{\ \texttt{stmt}\ \}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced
- `[=]` all necessary variables are automatically copied

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \ \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \ \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \ \underbrace{\texttt{\{ stmt \}}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced
- `[=]` all necessary variables are automatically copied
- `[&, x]` all necessary variables are referenced, except `x`, which is copied

Lambda Expression Syntax

Most general shape:

$$\underbrace{[e_1, \ldots, e_n]}_{\text{captures}} \; \underbrace{(T_1 x_1, \ldots, T_n x_n)}_{\text{parameters}} \; \underbrace{\texttt{->}\ \texttt{R}}_{\text{return type}} \; \underbrace{\{\ \texttt{stmt}\ \}}_{\text{body}}$$

Captures declare context variables the lambda's body can access. Syntax examples:

- `[]` no context access
- `[x]` `x` is copied (and `const`)
- `[&x]` `x` is accessible by reference
- `[x, &y]` `x` is copied, `y` is referenced
- `[&]` all necessary variables are automatically referenced
- `[=]` all necessary variables are automatically copied
- `[&, x]` all necessary variables are referenced, except `x`, which is copied
- `[=, &x]` all necessary variables are copied, except `x`, which is referenced

Functors

1. Write down the functor that corresponds to the lambda
2. Use the functor in the filter(...) expression

```
unsigned count = 0;
int min = 3;
std::vector<int> data = {4,-2,0};
data = filter(data, [&, min](int e) {
  ++count; return min <= e;
});
```

Functors

1. Write down the functor that corresponds to the lambda
2. Use the functor in the filter(...) expression

```cpp
unsigned count = 0;
int min = 3;
std::vector<int> data = {4,-2,0};
data = filter(data, [&, min](int e) {
  ++count; return min <= e;
});
```

Solution

```cpp
class lambda1 {
  unsigned& count;
  int min;
public:
  lambda1(unsigned& c, int m):
    count(c), min(m) {}
  bool operator()(int e) const {
    ++count;
    return min <= e;
  }
};
```

```cpp
unsigned count = 0;
int min = 3;
std::vector<int> data = {4,-2,0};

data = filter(data, lambda1(count, min));
```

Functors

- Observe that the lambda now uses the `auto` type placeholder for its argument

```
data = filter(data, [](auto e) { return 0 <= e; });
```

Functors

- Observe that the lambda now uses the `auto` type placeholder for its argument

```
data = filter(data, [](auto e) { return 0 <= e; });
```

- Question: How is this reflected by the generated functor?

Functors

- Observe that the lambda now uses the `auto` type placeholder for its argument

```
data = filter(data, [](auto e) { return 0 <= e; });
```

- Question: How is this reflected by the generated functor?
- Solution:

```
class lambda2 {
public:
  lambda2() {}

  template <typename T>
  bool operator()(T e) const {
    return 0 <= e;
  }
};
```

6.1 Function Signature Notation

not exam relevant

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean
  - $f_3 : A \times (A \to B) \to B$ "higher-order function" (with two arguments)

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean
  - $f_3 : A \times (A \to B) \to B$ "higher-order function" (with two arguments)
  - $f_4 : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$ higher-order function involving vectors

Function Signature Notation

- In the context of functional programming, function signatures are often expressed in a mathematics-inspired notation
- Convention today: upper-case letters denote type parameters, lower-case names denote concrete types
- Examples:
  - $f_1 : A \to \text{int}$ function from any type to integer
  - $f_2 : A \times A \times A \to \text{bool}$ function from three A's to boolean
  - $f_3 : A \times (A \to B) \to B$ "higher-order function" (with two arguments)
  - $f_4 : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$ higher-order function involving vectors
  - $f_5 : (A \times A \to B) \times A \to ((A \to B) \to \text{bool})$ taking and returning a function

Function Signature Notation: Example 1

- Task: Write down a function with signature $f_2 : A \times A \rightarrow \text{bool}$

Function Signature Notation: Example 1

- Task: Write down a function with signature $f_2 : A \times A \rightarrow \text{bool}$
- Solution:

```
template <typename A>
bool eq(A a1, A a2) {
  return a1 == a2;
}
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 1:

```
template <typename A, typename F>
auto apply1(A a, F a_to_b) {
  return a_to_b(a);
}

int i1 = apply1('a', [](char c) { return c - 65; });
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 1:

```
template <typename A, typename F>
auto apply1(A a, F a_to_b) {
  return a_to_b(a);
}

int i1 = apply1('a', [](char c) { return c - 65; });
```

- Observations
  - type parameter `B` is only implicitly given, as `F`'s return type
  - template type parameters inferred at call-site

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \rightarrow B) \rightarrow B$

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 2:

```
template <typename A, typename B>
B apply2(A a, std::function<B(A)> a_to_b) {
  return a_to_b(a);
}

int i2 = apply2('a', std::function([](char c) { return c - 65; }));
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- Solution 2:

```
template <typename A, typename B>
B apply2(A a, std::function<B(A)> a_to_b) {
  return a_to_b(a);
}

int i2 = apply2('a', std::function([](char c) { return c - 65; }));
```

- Observations
  - type parameter `B` is explicit
  - but we need to wrap the lambda in a `std::function`
  - template type parameters inferred at call-site

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- ~~Solution~~ Attempt 3

```
template <typename A, typename F, typename B>
B apply3(A a, F a_to_b) {
  return a_to_b(a);
}

int i3 = apply3<char, ???, int>('a', [](char c) { return c - 65; });
```

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- ~~Solution~~ Attempt 3

```
template <typename A, typename F, typename B>
B apply3(A a, F a_to_b) {
  return a_to_b(a);
}

int i3 = apply3<char, ???, int>('a', [](char c) { return c - 65; });
```

- Observations
  - type parameter `B` is explicit
  - but not directly connected to return type of `F`

Function Signature Notation: Example 2

- Task: Write down a function with signature $f_2 : A \times (A \to B) \to B$
- ~~Solution~~ Attempt 3

```
template <typename A, typename F, typename B>
B apply3(A a, F a_to_b) {
  return a_to_b(a);
}

int i3 = apply3<char, ???, int>('a', [](char c) { return c - 65; });
```

- Observations
  - type parameter `B` is explicit
  - but not directly connected to return type of `F`
- Problem: At call-site, `B` can't be inferred. We can explicitly instantiate `B` - but now we'd have to do that for `F` as well, which we can't.

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind([](auto s1, auto s2) { return s1 + s2; }, planet);
```

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind([](auto s1, auto s2) { return s1 + s2; }, planet);
```

- Question: how to use `f`?

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind([](auto s1, auto s2) { return s1 + s2; }, planet);
```

- Question: how to use `f`?
- Answer:

```
std::cout << f(" is the fourth planet from the sun.");
```

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind([](auto s1, auto s2) { return s1 + s2; }, planet);
```

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind([](auto s1, auto s2) { return s1 + s2; }, planet);
```

- Question: What would happen if the capture were `[&]` instead of `[=]`?

Function Signature Notation: Example 3

- Task: Write down a function with signature
  $f_2 : A \times (A \times A \to B) \to (A \to B)$
- Solution:

```
template <typename A, typename F>
auto bind(A a1, F aa_to_b) {
  return [=](A a2) { return aa_to_b(a1, a2); };
}

std::string planet = "Mars";
auto f = bind([](auto s1, auto s2) { return s1 + s2; }, planet);
```

- Question: What would happen if the capture were `[&]` instead of `[=]`?
- Answer: The returned lambda would capture argument `a1` by reference, but `a1` is removed from memory when the call to `bind()` terminates. Calling `f` would thus result in undefined behaviour.

A Prominent Higher Order Function

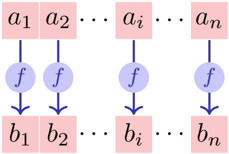
- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$

A Prominent Higher Order Function

- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?

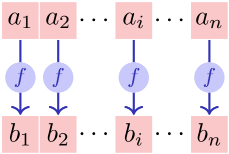
A Prominent Higher Order Function

- Consider the function m : `vec<A>` $\times (A \to B) \to$ `vec<B>`
- Given the signature above, what could function m do?
- Visual hint:



A Prominent Higher Order Function

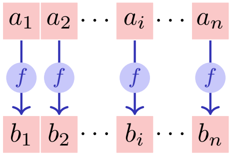
- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?
- Visual hint:



- Task: Implement the function in C++

A Prominent Higher Order Function

- Consider the function $m : \texttt{vec<A>} \times (A \to B) \to \texttt{vec<B>}$
- Given the signature above, what could function m do?
- Visual hint:



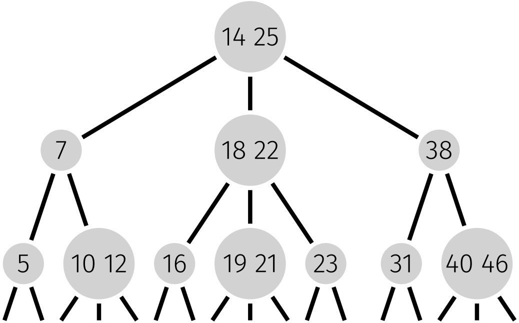
- Task: Implement the function in C++
- Solution:

```cpp
template <typename A, typename B>
std::vector<B> map(std::vector<A> as, std::function<B(A)> f) {
  std::vector<B> result;
  for (const auto& a : as)
    result.push_back(f(a));
  return result;
}
```

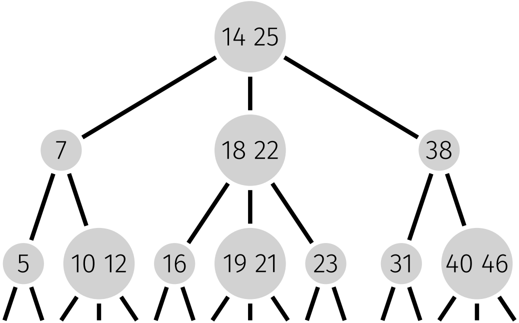
7. Tree Recap

7.1 Recap 2-3 Trees

Searching

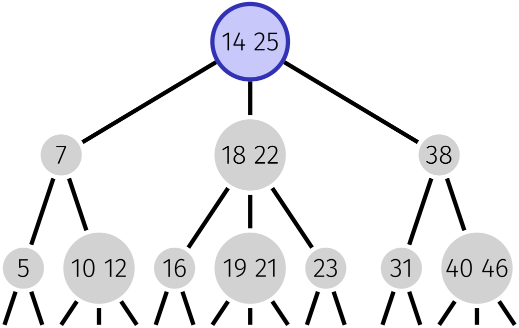


Searching



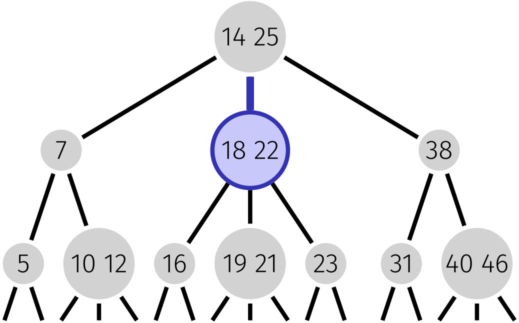
$\texttt{search}(23)$

Searching



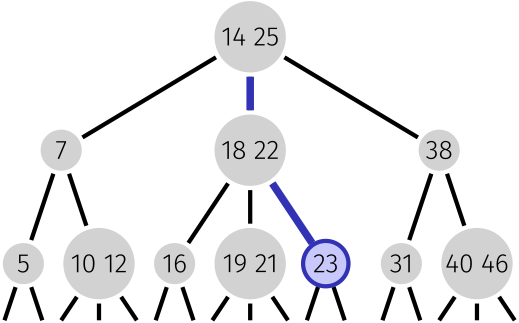
$\texttt{search}(23)$

Searching



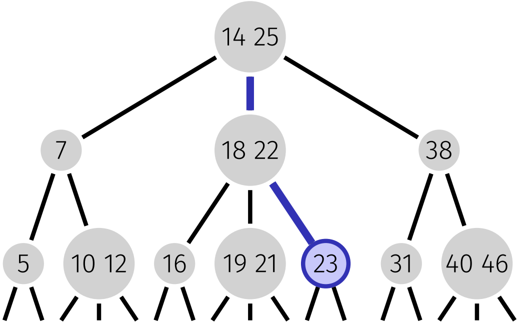
$\texttt{search}(23)$

Searching



$\texttt{search}(23)$

Searching

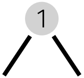


$\texttt{search}(23) \rightarrow \texttt{found}$

2-3 Tree: Insertion

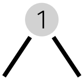
Insert the keys $1, \ldots, 7$ into an (initially empty) 2-3-tree. Draw the tree after every step (`split`/`propagate`, `join`, $\ldots$).

2-3 Tree: Insertion

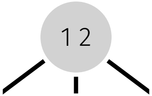


`insert(1)`:
new node

2-3 Tree: Insertion

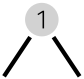


`insert(1)`:
new node

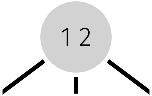


`insert(2)`:
`join`

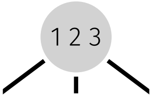
2-3 Tree: Insertion



`insert(1)`:
new node

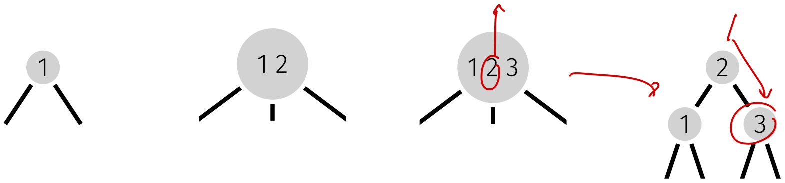


`insert(2)`:
`join`



`insert(3)`:
4-node

2-3 Tree: Insertion



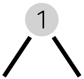
`insert(1)`: new node

`insert(2)`: join

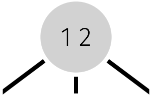
`insert(3)`: 4-node

`insert(3)`: split/propagate

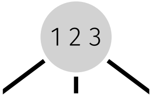
2-3 Tree: Insertion



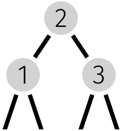
`insert(1)`:
new node



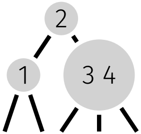
`insert(2)`:
`join`



`insert(3)`:
4-node

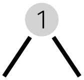


`insert(3)`:
`split/propagate`

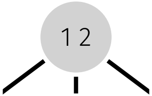


`insert(4)`:
`join`

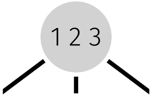
2-3 Tree: Insertion



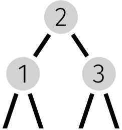
`insert(1)`:
new node



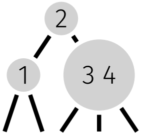
`insert(2)`:
`join`



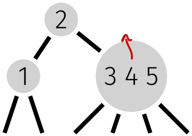
`insert(3)`:
4-node



`insert(3)`:
`split/propagate`

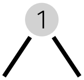


`insert(4)`:
`join`

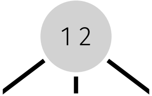


`insert(5)`:
4-node

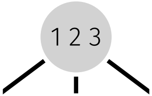
2-3 Tree: Insertion



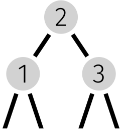
`insert(1)`:
new node



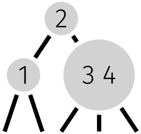
`insert(2)`:
`join`



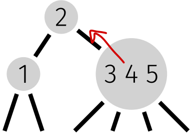
`insert(3)`:
4-node



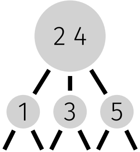
`insert(3)`:
`split/propagate`



`insert(4)`:
`join`

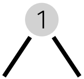


`insert(5)`:
4-node

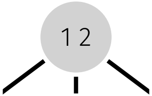


`insert(5)`:
`split/propagate`

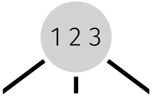
2-3 Tree: Insertion



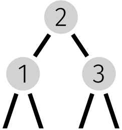
`insert(1)`:
new node



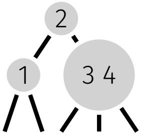
`insert(2)`:
`join`



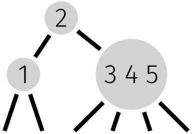
`insert(3)`:
4-node



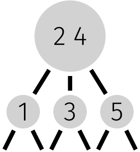
`insert(3)`:
`split/propagate`



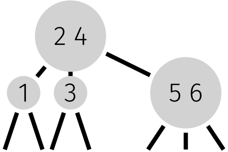
`insert(4)`:
`join`



`insert(5)`:
4-node

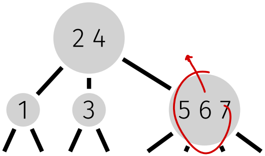


`insert(5)`:
`split/propagate`



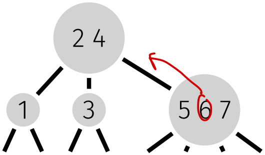
`insert(6)`:
`join`

2-3 Tree: Insertion

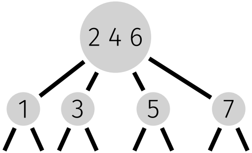


`insert(7)`:
4-node

2-3 Tree: Insertion

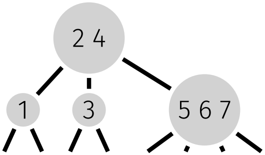


`insert(7)`:
4-node

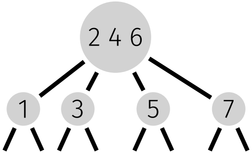


`insert(7)`:
`split/propagate`

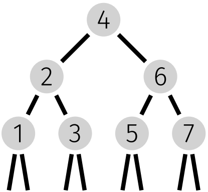
2-3 Tree: Insertion



`insert(7)`:
4-node

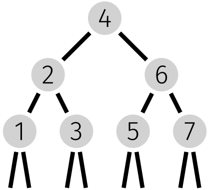


`insert(7)`:
`split/propagate`



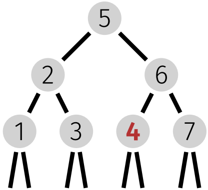
`insert(7)`:
`split/propagate`

2-4 Tree: Deletion

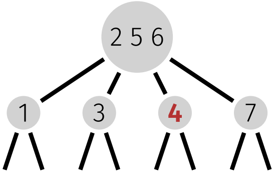


Delete key 4 from the resulting tree.

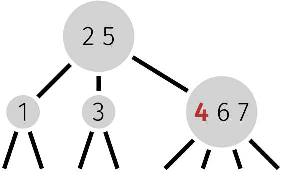
2-4 Tree: Deletion



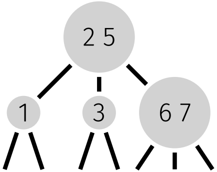
1. swap



2. create 4-node at root



3. combine with sibling

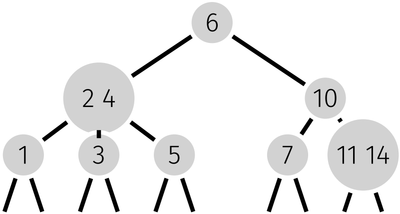


4. delete key

7.2 Recap Red-Black Trees

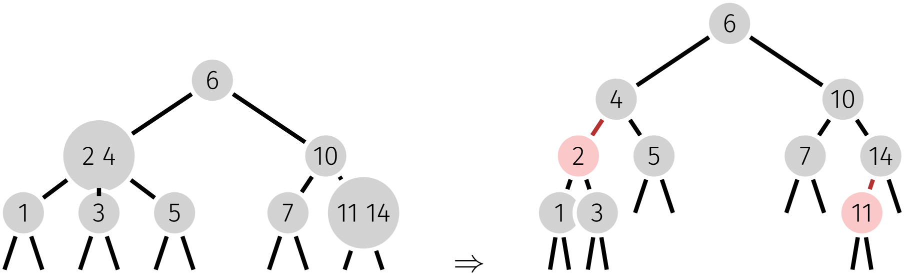
Red-Black Trees

Draw the following 2-3 tree as a red-black tree.



Red-Black Trees

Draw the following 2-3 tree as a red-black tree.



Red-Black Trees: True or False?

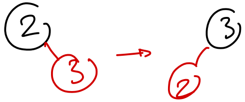
1. right spine (path going right from root) has length $\lceil \log_2(n+1) \rceil$.

Red-Black Trees: True or False?

1. right spine (path going right from root) has length $\lceil \log_2(n+1) \rceil$.
   Correct, since there are no right-leaning red edges and we have perfect black balance.

Red-Black Trees: True or False?

1. right spine (path going right from root) has length $\lceil \log_2(n+1) \rceil$.
   Correct, since there are no right-leaning red edges and we have perfect black balance.
2. the number of red edges is at most the number of black edges.



Red-Black Trees: True or False?

1. right spine (path going right from root) has length $\lceil \log_2(n+1) \rceil$.
   Correct, since there are no right-leaning red edges and we have perfect black balance.
2. the number of red edges is at most the number of black edges.
   Wrong, a tree with 2 nodes and one edge must have a red edge but not black edge.

Red-Black Trees: True or False?

1. right spine (path going right from root) has length $\lceil \log_2(n+1) \rceil$.
   Correct, since there are no right-leaning red edges and we have perfect black balance.
2. the number of red edges is at most the number of black edges.
   Wrong, a tree with 2 nodes and one edge must have a red edge but not black edge.
3. All nodes in the left subtree of a node are smaller than the node.

Red-Black Trees: True or False?

1. right spine (path going right from root) has length $\lceil \log_2(n+1) \rceil$.
   Correct, since there are no right-leaning red edges and we have perfect black balance.
2. the number of red edges is at most the number of black edges.
   Wrong, a tree with 2 nodes and one edge must have a red edge but not black edge.
3. All nodes in the left subtree of a node are smaller than the node.
   Correct, since a red-black tree is a search tree.

Red-Black Trees: Insertion

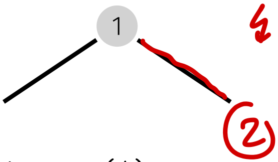
Insert the numbers $1, \ldots, 7$ into an (initially empty) red-black tree and draw the tree after every step.

Red-Black Trees: Insertion

Insert the numbers $1, \ldots, 7$ into an (initially empty) red-black tree and draw the tree after every step.

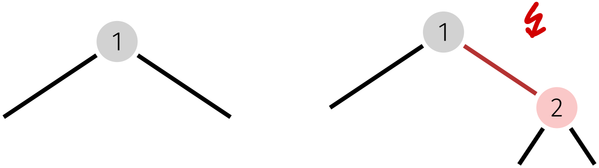
Compare your steps with your result for the 2-3 tree before.

Red-Black Trees: Insertion



`insert(1)`

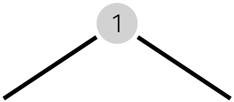
Red-Black Trees: Insertion



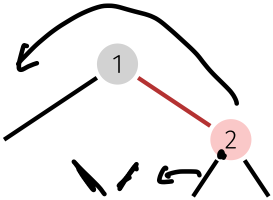
`insert(1)`

`insert(2): add`

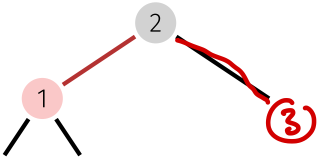
Red-Black Trees: Insertion



`insert(1)`

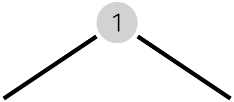


`insert(2): add`

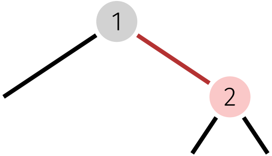


`insert(2): rotate_left`
(since right child red)

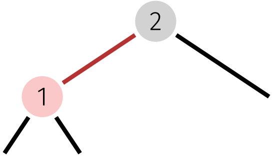
Red-Black Trees: Insertion



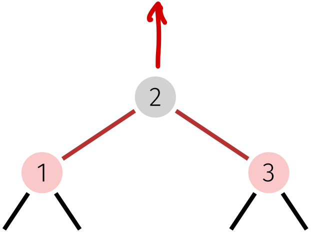
`insert(1)`



`insert(2): add`

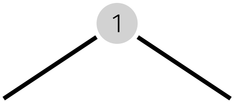


`insert(2): rotate_left`
(since right child red)

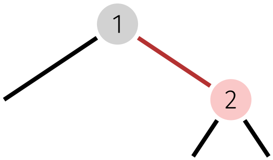


`insert(3): add`

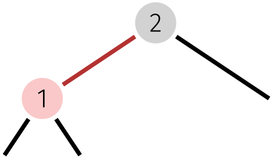
Red-Black Trees: Insertion



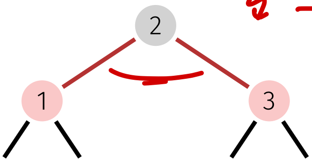
`insert(1)`



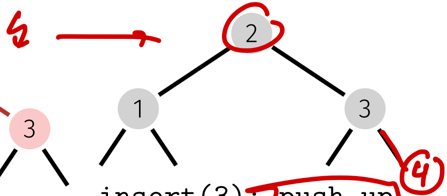
`insert(2): add`



`insert(2): rotate_left`
(since right child red)

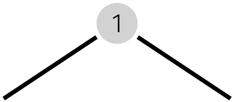


`insert(3): add`

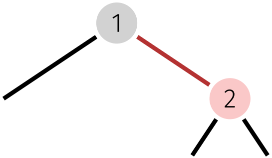


`insert(3): push_up`
(two red children)

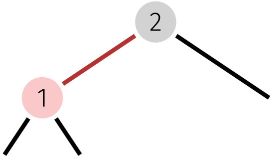
Red-Black Trees: Insertion



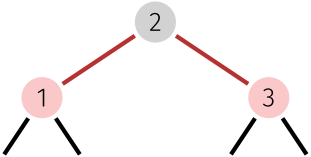
`insert(1)`



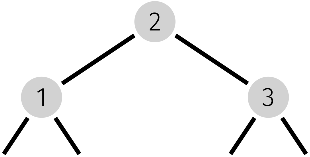
`insert(2): add`



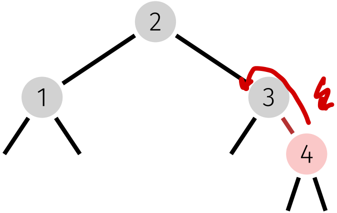
`insert(2): rotate_left`
(since right child red)



`insert(3): add`

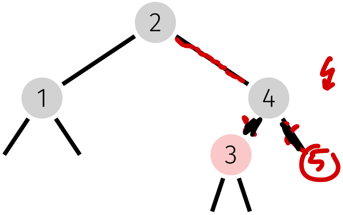


`insert(3): push_up`
(two red children)



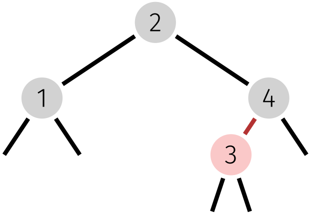
`insert(4): add`

Red-Black Trees: Insertion

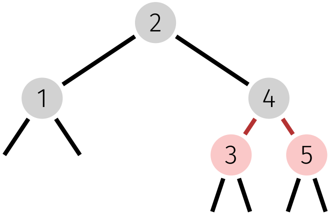


`insert(4): rotate_left`
(since right child red)

Red-Black Trees: Insertion

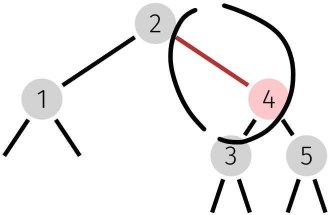


`insert(4): rotate_left`
(since right child red)



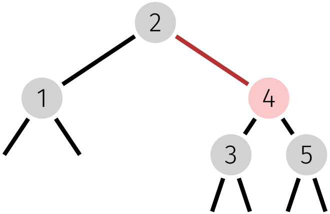
`insert(5): add`

Red-Black Trees: Insertion

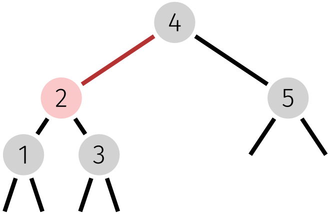


`insert(5): push_up`
(two children red)

Red-Black Trees: Insertion

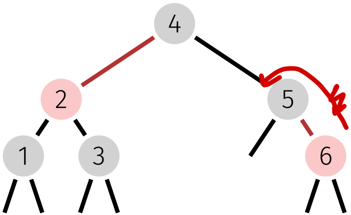


`insert(5): push_up`
(two children red)



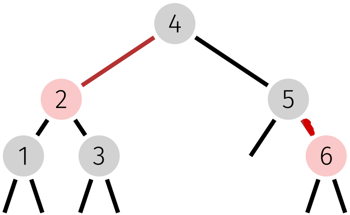
`insert(5): rotate_left`
(since right child red)

Red-Black Trees: Insertion

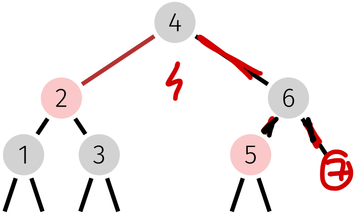


`insert(6): add`

Red-Black Trees: Insertion

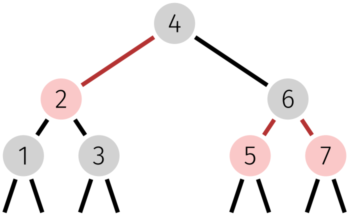


`insert(6): add`



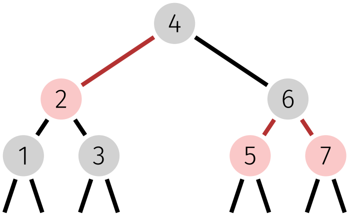
`insert(6): rotate_left`
(since right child red)

Red-Black Trees: Insertion

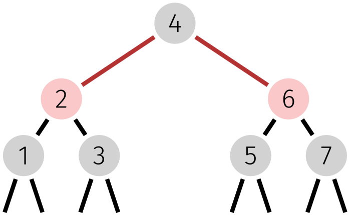


`insert(7): add`

Red-Black Trees: Insertion

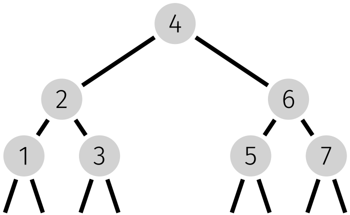


`insert(7): add`



`insert(7): push_up`
(since two children red)

Red-Black Trees: Insertion



`insert(7): push_up`

(since both children red)

8. Outro

General Questions?

See you next time

Have a nice week!

PVK Demand Analysis

- **Chemie**
- **D&A**