Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Applied Functional Programming

USCS 2009

Andres Löh

Department of Information and Computing Sciences
Utrecht University

August 20, 2009

C2. Data structures

Overview

- Persistent data structures
- Arrays
- Trees, sets and finite maps
- Other useful data structures

C2.1 Persistent data structures

Imperative vs. functional style

Given a finite map (associative map, dictionary) m.

Imperative style:

> m.put $(42, \texttt{"Mordor"}); \ldots$

Functional style:

> **let** $\mathsf{m}' = \mathsf{insert}\ 42\ \texttt{"Gondor"}\ \mathsf{m}$ **in** $\ldots$

What is the difference?

Imperative vs. functional style

Given a finite map (associative map, dictionary) m.

Imperative style:

> m.put $(42, \texttt{"Mordor"}); \ldots$

Functional style:

> **let** $\mathsf{m}' = \mathsf{insert}\ 42\ \texttt{"Gondor"}\ \mathsf{m}$ **in** $\ldots$

What is the difference?

Imperative: destructive update

Functional: creation of a new value

Persistent data structures

Imperative languages:

- many operations make use of destructive updates
- after an update, the old version of the data structure is no longer available

Functional languages:

- most operations create a new data structure
- old versions are still available

Data structures where old version remain accessible are called **persistent**.

Persistent data structures

- In functional languages, most data structures are (automatically) persistent.
- In imperative languages, most data structures are not persistent (**ephemeral**).
- It is generally possible to also use ephemeral data structures in functional or persistent data structures in imperative languages.

Persistent data structures

- In functional languages, most data structures are (automatically) persistent.
- In imperative languages, most data structures are not persistent (**ephemeral**).
- It is generally possible to also use ephemeral data structures in functional or persistent data structures in imperative languages.

How do persistent data structures work?

Example: Haskell lists

$$[1, 2, 3, 4]$$

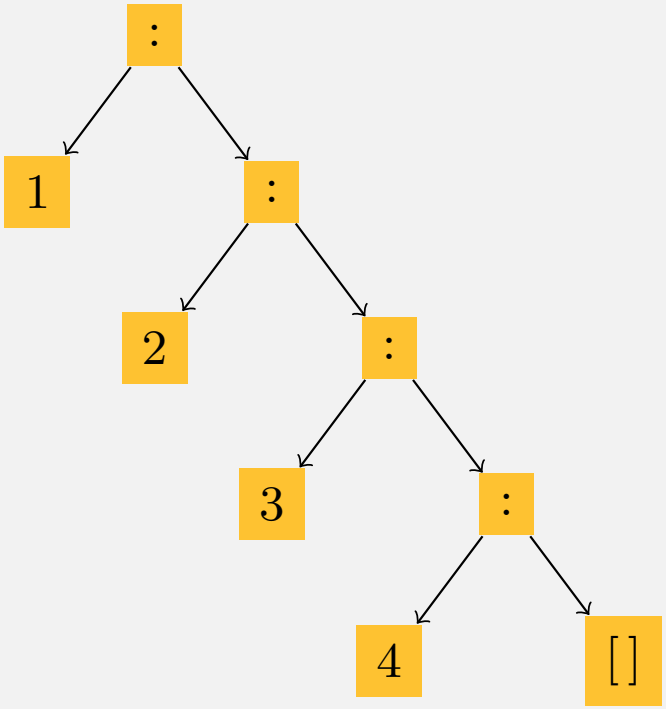
Example: Haskell lists

$[1, 2, 3, 4]$ is syntactic sugar for $1 : (2 : (3 : (4 : [\,])))$

Example: Haskell lists

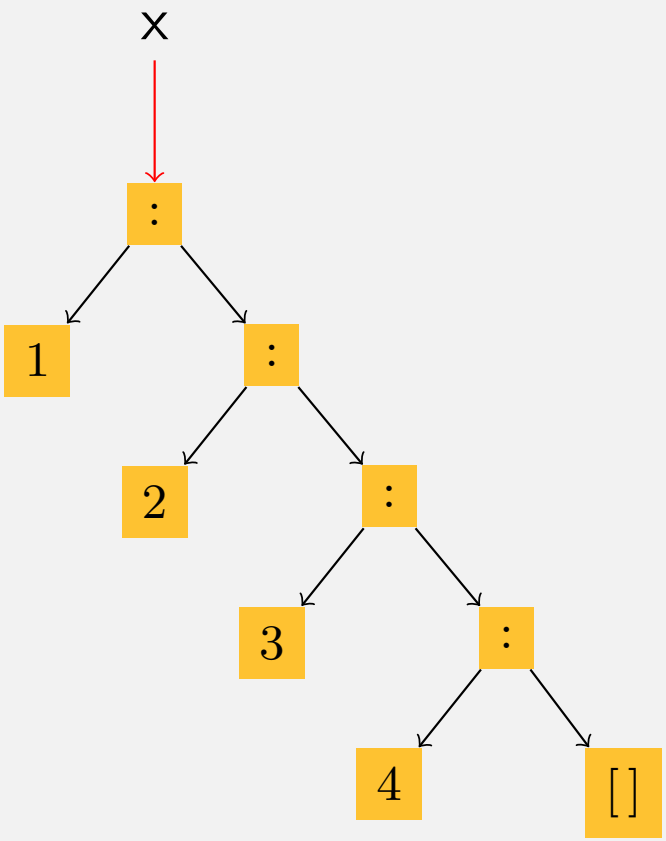
$[1, 2, 3, 4]$ is syntactic sugar for $1 : (2 : (3 : (4 : [\,])))$

Representation in memory:



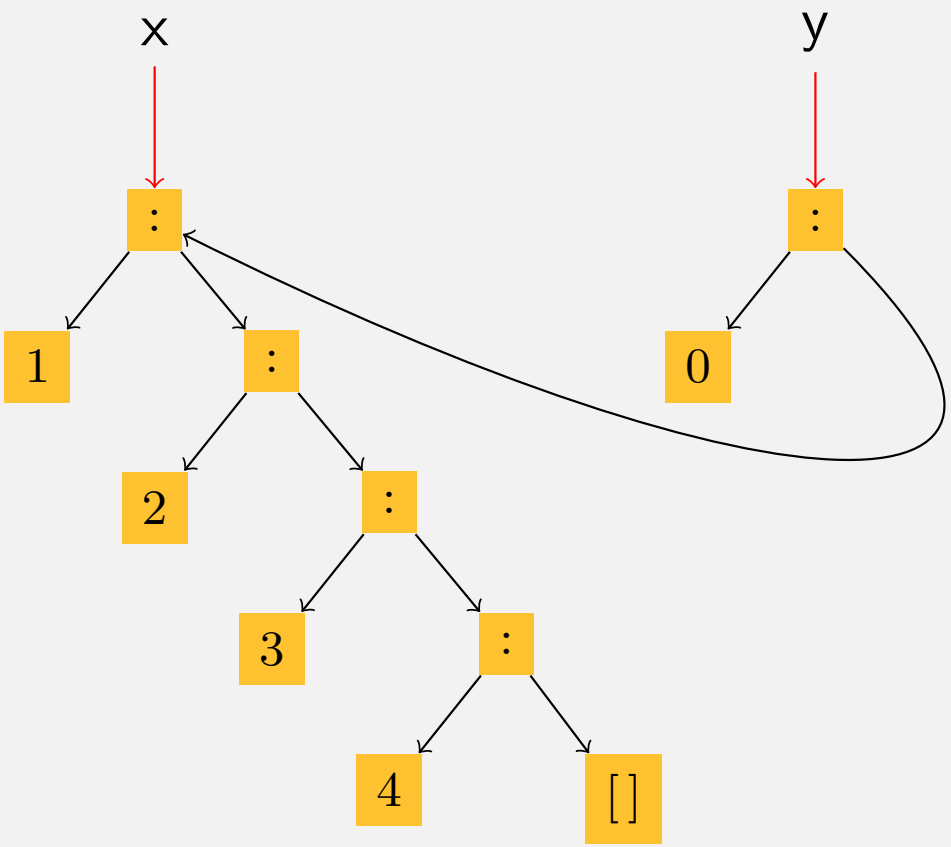
Lists are persistent

$$\textbf{let}\ \mathsf{x} = [1, 2, 3, 4]; \mathsf{y} = 0 : \mathsf{x}; \mathsf{z} = \mathsf{drop}\ 3\ \mathsf{y}\ \textbf{in} \ldots$$



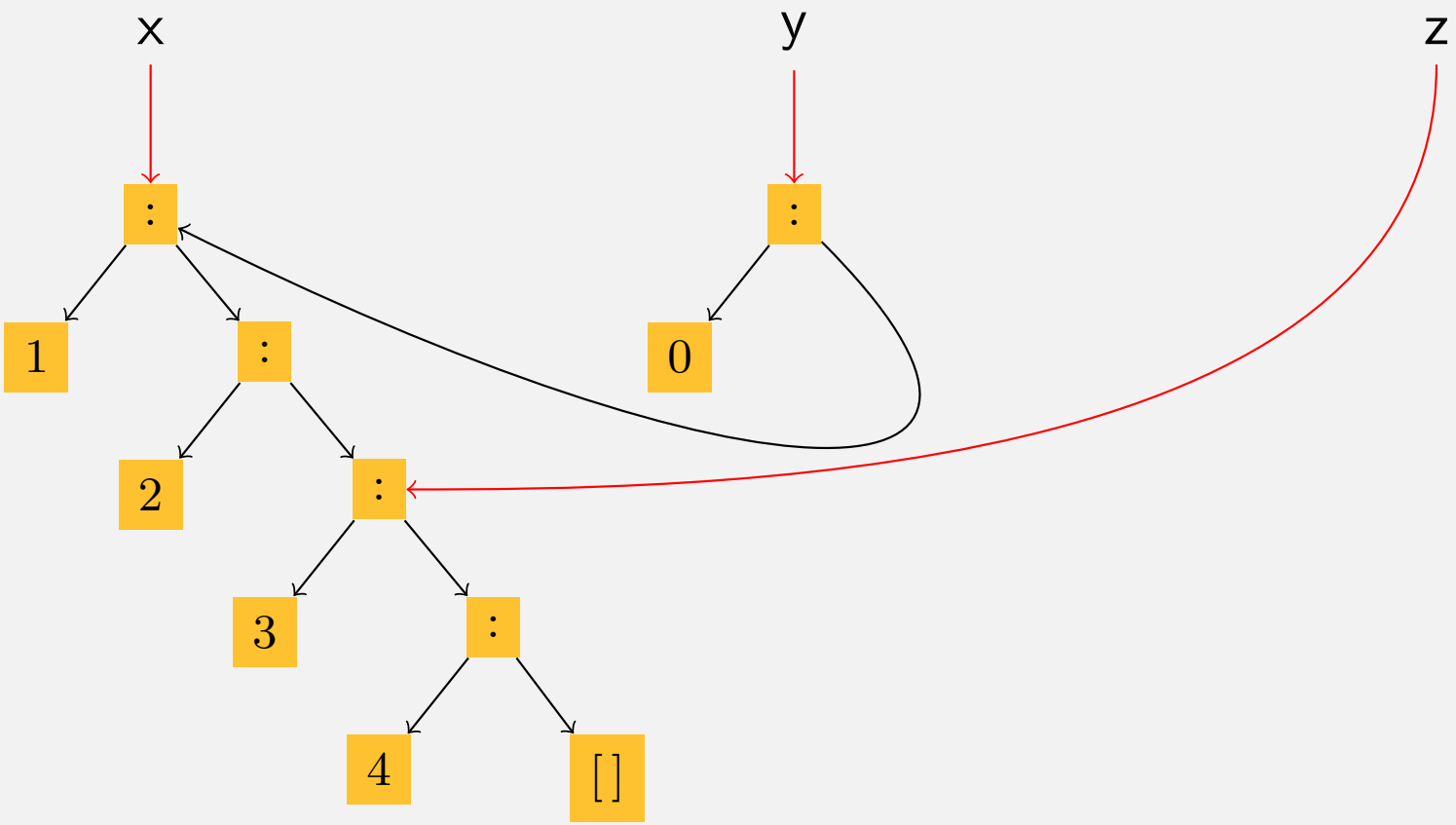
Lists are persistent

let $\mathsf{x} = [1, 2, 3, 4]; \mathsf{y} = 0 : \mathsf{x}; \mathsf{z} = \mathsf{drop}\ 3\ \mathsf{y}$ **in** ...



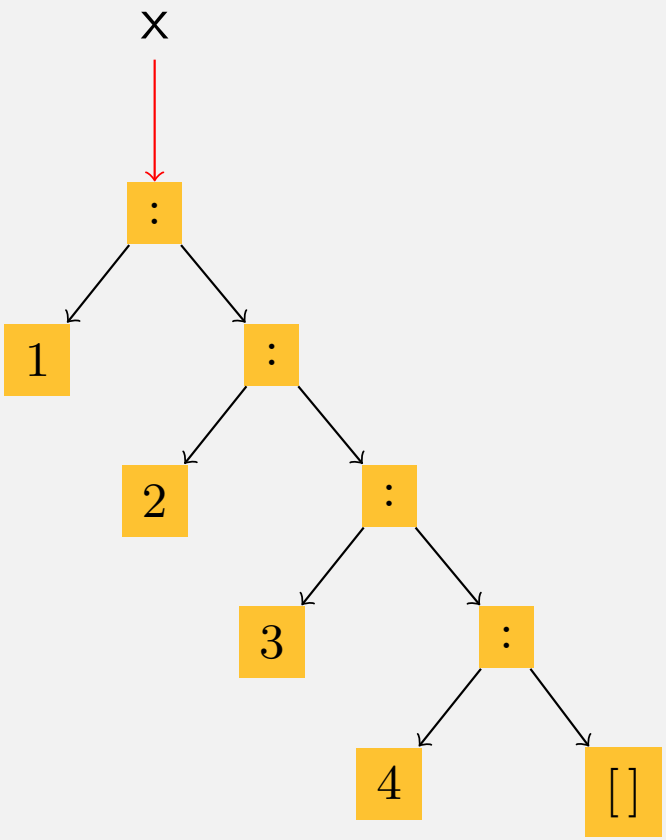
Lists are persistent

let $\mathsf{x} = [1, 2, 3, 4]; \mathsf{y} = 0 : \mathsf{x}; \mathsf{z} = \mathsf{drop}\ 3\ \mathsf{y}$ **in** ...



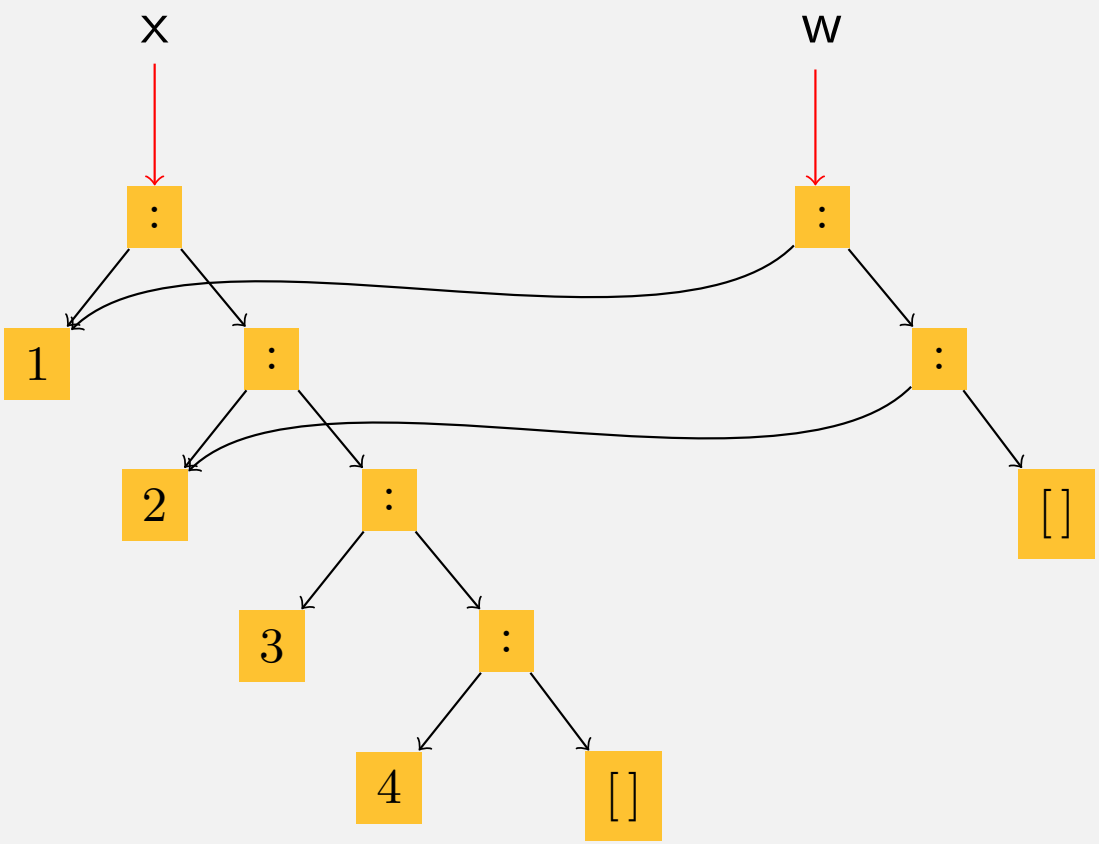
Lists are persistent – contd.

let $\mathsf{x} = [1, 2, 3, 4]; \mathsf{w} = \mathsf{take}\ 2\ \mathsf{x}$ **in** ...



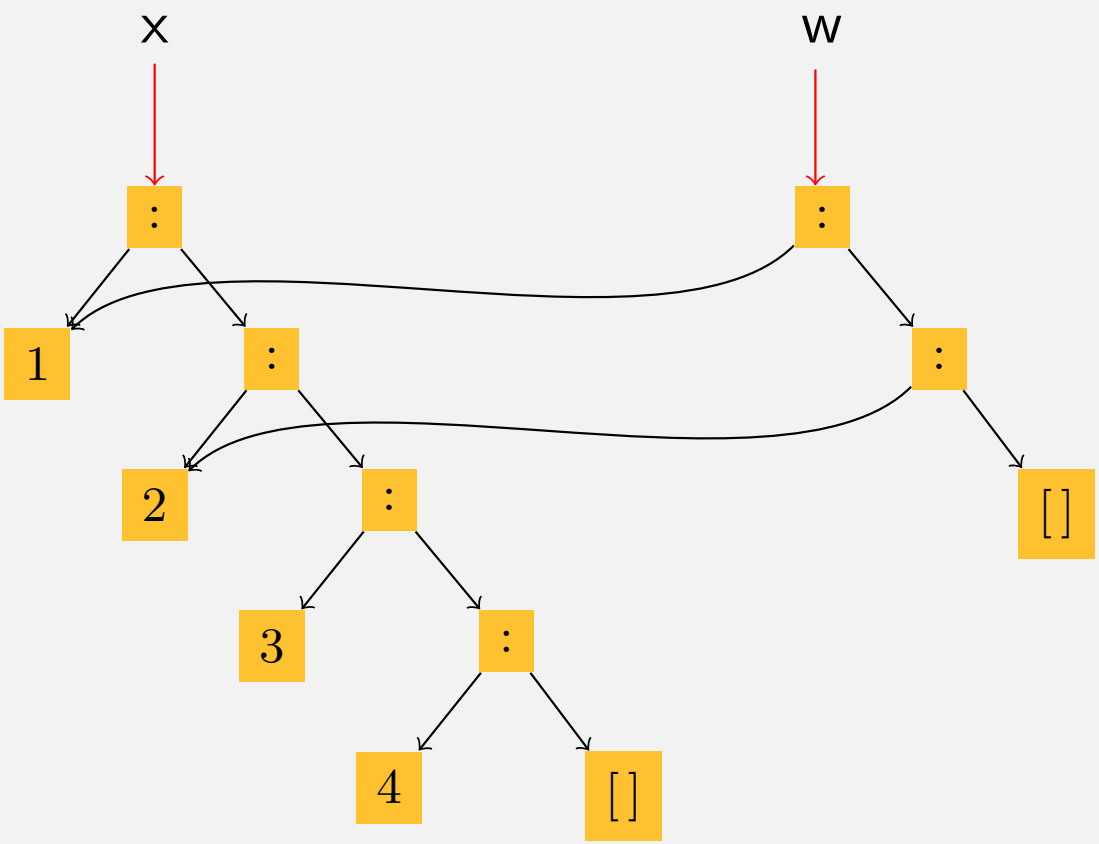
Lists are persistent – contd.

let x = [1, 2, 3, 4]; w = take 2 x **in** ...



Lists are persistent – contd.

let x = [1, 2, 3, 4]; w = take 2 x **in** . . .



Nodes are automatically copied as needed.

Implementation of persistent data structures

- Modifications of an existing structure take place by creating new nodes and pointers.
- Sometimes, parts of a structure have to be copied, because the old version must not be modified.

Of course, we want to copy as little as possible, and reuse as much as possible.

Excursion: Vacuum

Vacuum is a tool by Matt Morrow.

It generates graph representations of Haskell values.

The representations can then be visualized.

Excursion: Vacuum

Vacuum is a tool by Matt Morrow.

It generates graph representations of Haskell values.

The representations can then be visualized.

```
System.Vacuum.Cairo⟩ view (let x = [1, 2] in x ++ x)
```

C2.2 Arrays

Persistence and complexity

Some data structures show unexpected (i.e., bad) behaviour when used in a persistent setting:

Haskell arrays:

> **let** x = listArray $(0, 4)$ $[1, 2, 3, 4, 5]$ **in** x // $[(2, 13)]$

Persistence and complexity

Some data structures show unexpected (i.e., bad) behaviour when used in a persistent setting:

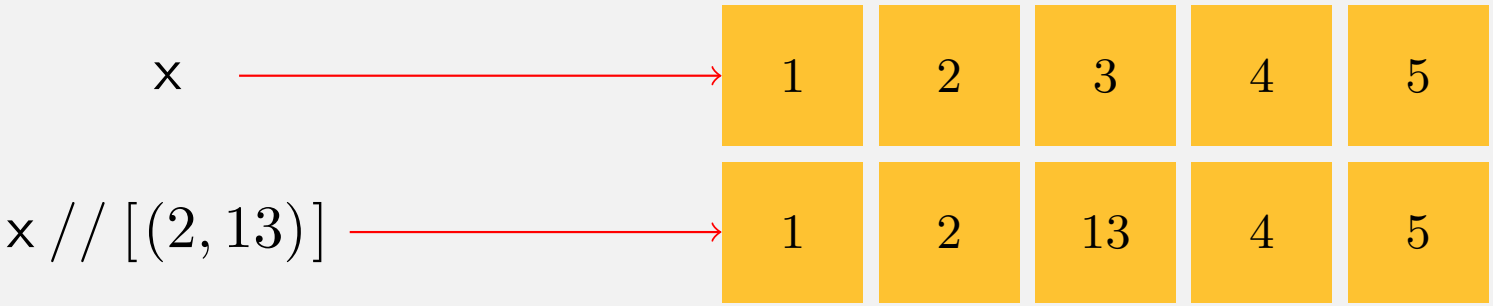
Haskell arrays:

> **let** x = listArray $(0, 4)$ $[1, 2, 3, 4, 5]$ **in** x // $[(2, 13)]$

How expensive is the update operation?

- In an imperative language, we expect O(1), i.e., constant time.

Arrays



- Arrays are stored in a contiguous block of memory.
- This allows O(1) access to each element.
- A destructive update is also possible in O(1).
- But if a persistent update is desired, the whole array must be copied, which takes O(n), i.e., linear time.

Advice on arrays

Generally, do not use them.

Arrays are useful only if you have some essentially constant data you want quick access to.

C2.3 Trees

Trees

- Arrays and hash tables are expensive in a functional (persistent) setting, because it is impossible to share substructures between different versions.
- Tree-shaped structures, however, are generally very suitable in a functional setting. Reuse of subtrees is easy to achieve. Most functional data structures therefore are (different sorts of) trees.

Trees

- Arrays and hash tables are expensive in a functional (persistent) setting, because it is impossible to share substructures between different versions.
- Tree-shaped structures, however, are generally very suitable in a functional setting. Reuse of subtrees is easy to achieve. Most functional data structures therefore are (different sorts of) trees.

Lists are trees, too – just a very peculiar variant.

Lists

- There is a lot of syntactic sugar for lists in Haskell. Thus, lists are used for a lot of different purposes.
- Lists are the default data structure in functional languages much as arrays are in imperative languages.
- However, lists support only a few operations efficiently.

Operations on lists

```
[]       :: [a]                          -- O(1)
(:)      :: a → [a] → [a]                -- O(1)
head     :: [a] → a                      -- O(1)
tail     :: [a] → [a]                    -- O(1)
snoc     :: [a] → a → [a]                -- O(n)
snoc     = λxs x → xs ++ [x]
(!!)     :: [a] → Int → a                -- O(n)
(++)     :: [a] → [a] → [a]              -- O(m) (first list)
reverse  :: [a] → [a]                    -- O(n)
splitAt  :: Int → [a] → ([a], [a])       -- O(n)
union    :: Eq a ⇒ [a] → [a] → [a]       -- O(mn)
elem     :: Eq a ⇒ a → [a] → Bool        -- O(n)
```

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

Can be used to implement finite maps and sets efficiently, and persistently.

What is better than lists?

Are there functional data structures that support a more efficient lookup operation than lists?

Yes, balanced search trees.

Can be used to implement finite maps and sets efficiently, and persistently.

Question

What is a (binary) search tree?

Finite maps

- A finite map is a function with a finite domain (type of keys).
- Useful for a wide variety of applications (tables, environments, arrays (!)).
- Inefficient representation: **type** Map a b = [(a, b)].
- Can be implemented efficiently using balanced search trees:
  - Available in Data.Map and Data.IntMap for Int as key type.
  - Requires the keys to be ordered.
  - Keys are stored ordered in the tree, so that efficient lookup is possible.
  - Inserting and removing elements can trigger rotations to rebalance the tree.

Sets

- Sets are a special case of finite maps: essentially,

```
type Set a = Map a ()
```

- A specialized set implementation is available in Data.Set and Data.IntSet, but the idea is the same as for finite maps.

Finite map interface

This is an excerpt from the functions available in Data.Map:

```
data Map k a   -- abstract

insert  :: (Ord k) ⇒ k → a → Map k a → Map k a      -- O(log n)
lookup  :: (Ord k) ⇒ k → Map k a → Maybe a          -- O(log n)
delete  :: (Ord k) ⇒ k → Map k a → Map k a          -- O(log n)
update  :: (Ord k) ⇒ (a → Maybe a) →
                     k → Map k a → Map k a          -- O(log n)
union   :: (Ord k) ⇒ Map k a → Map k a → Map k a    -- O(m + n)
member  :: (Ord k) ⇒ k → Map k a → Bool             -- O(log n)
size    :: Map k a → Int                            -- O(1)
map     :: (a → b) → Map k a → Map k b              -- O(n)
```

The interface for Set is very similar.

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later.

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later.

A map is either a leaf called Tip,

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later.

A map is either a leaf called Tip, or a binary node called Bin

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later.

A map is either a leaf called Tip, or a binary node called Bin containing the size of the tree

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later.

A map is either a leaf called Tip, or a binary node called Bin containing the size of the tree the key and the value

Implementation

In the following, we will sketch the implementation as it is available in Data.Map:

```
data Map k a = Tip
             | Bin !Size (Map k a) k a (Map k a)
type Size = Int
```

The ! is a strictness annotation for extra efficiency. More about that later.

A map is either a leaf called Tip, or a binary node called Bin containing the size of the tree the key and the value and a left and right subtree.

Creating finite maps

```
empty :: Map k a
empty = Tip

singleton :: k → a → Map k a
singleton k x = bin Tip k x Tip
```

Creating finite maps

```
empty :: Map k a
empty = Tip

singleton :: k → a → Map k a
singleton k x = bin Tip k x Tip
```

The function bin is an example of a **smart constructor**.

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

In this case: the Size argument of Bin should always reflect the actual size of the tree.

```
bin :: Map k a → k → a → Map k a → Map k a
bin l k x r = Bin (size l + size r + 1) l k x r
```

Smart constructors

Smart constructors are wrappers around constructors that help to ensure that invariants of the data structure are maintained.

In this case: the Size argument of Bin should always reflect the actual size of the tree.

```
bin :: Map k a → k → a → Map k a → Map k a
bin l k x r = Bin (size l + size r + 1) l k x r
```

```
size :: Map k a → Int
size Tip                = 0
size (Bin sz _ _ _ _) = sz
```

Finding an element

```
lookup :: Ord k ⇒ k → Map k a → Maybe a
lookup k Tip                  = Nothing
lookup k (Bin _ l kx x r) =
  case compare k kx of
    LT → lookup k l
    GT → lookup k r
    EQ → Just x
```

Finding an element

```
lookup :: Ord k ⇒ k → Map k a → Maybe a
lookup k Tip                  = Nothing
lookup k (Bin _ l kx x r) =
  case compare k kx of
    LT → lookup k l
    GT → lookup k r
    EQ → Just x
```

Comparing two elements:

```
compare :: Ord a ⇒ a → a → Ordering
data Ordering = LT | EQ | GT
```

Inserting an element

```
insert :: Ord k ⇒ k → a → Map k a → Map k a
insert kx x Tip = singleton kx x-- insert new value
insert kx x (Bin sz l ky y r) =
  case compare kx ky of
    LT → balance (insert kx x l) ky y r
    GT → balance l ky y (insert kx x r)
    EQ → Bin sz l kx x r-- overwrite old value
```

Inserting an element

```
insert :: Ord k ⇒ k → a → Map k a → Map k a
insert kx x Tip = singleton kx x-- insert new value
insert kx x (Bin sz l ky y r) =
  case compare kx ky of
    LT → balance (insert kx x l) ky y r
    GT → balance l ky y (insert kx x r)
    EQ → Bin sz l kx x r-- overwrite old value
```

The function balance is an even smarter constructor with the same type as bin:

```
balance :: Map k a → k → a → Map k a → Map k a
```

Balancing the tree

We could just define

> balance = bin

and that would actually be correct.

Question

What is the problem, and when does it arise?

Balancing approach

If the height of the two subtrees is not too different, we just use Bin.

Otherwise, we perform a rotation.

Balancing approach

If the height of the two subtrees is not too different, we just use Bin.

Otherwise, we perform a rotation.

Rotation

A rearrangement of the tree that preserves the search tree property.

Rotation

```
rotateL :: Map a b → a → b → Map a b → Map a b
rotateL l k x r@(Bin _ ly _ _ ry)
    | size ly < ratio * size ry = singleL l k x r
    | otherwise = doubleL l k x r
rotateL _ _ _ Tip = error "rotateL Tip"
```

Depending on the shape of the tree, either a simple (single) or a more complex (double) rotation is performed.

Rotation – contd.

```
singleL :: Map a b → a → b → Map a b → Map a b
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
    bin (bin t1 k1 x1 t2) k2 x2 t3
```

Rotation – contd.

```
singleL :: Map a b → a → b → Map a b → Map a b
singleL t1 k1 x1 (Bin _ t2 k2 x2 t3) =
  bin (bin t1 k1 x1 t2) k2 x2 t3
```

```
doubleL :: Map a b → a → b → Map a b → Map a b
doubleL t1 k1 x1 (Bin _ (Bin _ t2 k2 x2 t3) k3 x3 t4) =
  bin (bin t1 k1 x1 t2) k2 x2 (bin t3 k3 x3 t4)
```

C3. Other useful data structures

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

A universal data structure. A good choice for a wide range of applications.

Finger trees

- A balanced persistent tree structure.
- Supports search tree operations in logarithmic time.
- Supports cons and snoc in $O(1)$.
- Supports logarithmic splitting and union.

A universal data structure. A good choice for a wide range of applications.

Available in Data.Sequence and in an extended version in the fingertree package on Hackage.

Byte strings

Haskell strings are lists of characters.

That is a lot of overhead per character. For every character, a cons-cell is stored.

Byte strings have a more compact representation (array-like).

Byte strings

Haskell strings are lists of characters.

That is a lot of overhead per character. For every character, a cons-cell is stored.

Byte strings have a more compact representation (array-like).

To prevent the typical array problems, a form of fusion is used.

Byte strings

Haskell strings are lists of characters.

That is a lot of overhead per character. For every character, a cons-cell is stored.

Byte strings have a more compact representation (array-like).

To prevent the typical array problems, a form of fusion is used.

Available as Data.ByteString.

More on Hackage

On Hackage, there are several additional libraries for data structures.

Some examples: heaps, priority search queues, heterogeneous lists, zippers, tries, graphs, quadtrees, ...

Summary

- It is important to keep persistence in mind when thinking about functional data structures.
- Arrays should be used only in very specific and rare circumstances.
- Lists are ok for stack-like use or simple traversals.
- Good general-purpose data structures are sets, finite maps and finger trees.