

# Lecture 6. Purely Functional Data structures

## Functional Programming

Frank Staals



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

# Goals

- ▶ Know the difference between persistent (purely functional) and ephemeral data structures,
- ▶ Be able to use persistent data structures,
- ▶ Define and work with custom data types



# Data Structures in Memory

- ▶ What does `x:xs` look like in memory?



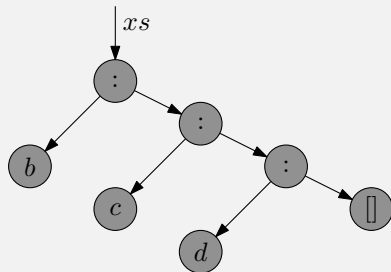
# Data Structures in Memory

- ▶ What does  $x:xs$  look like in memory?
- ▶ Suppose that  $xs = b:c:d:[]$  for some  $b,c$  and  $d$



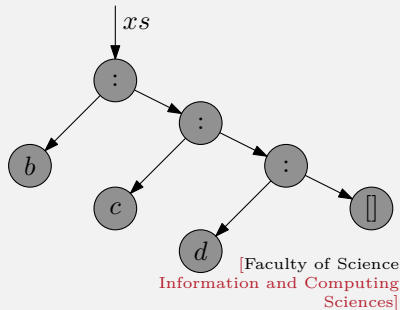
# Data Structures in Memory

- What does `xs = b:c:d:[]` look like in memory?



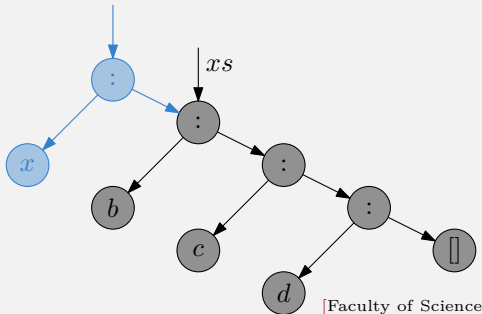
# Data Structures in Memory

- What does  $x:xs$  look like in memory?



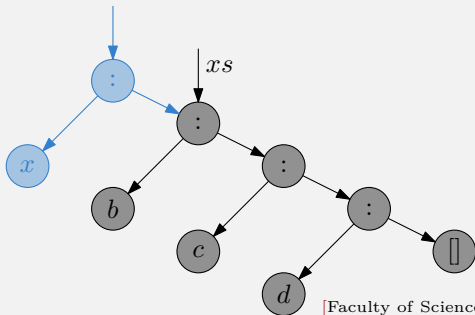
# Data Structures in Memory

- What does  $x:xs$  look like in memory?



# Data Structures in Memory

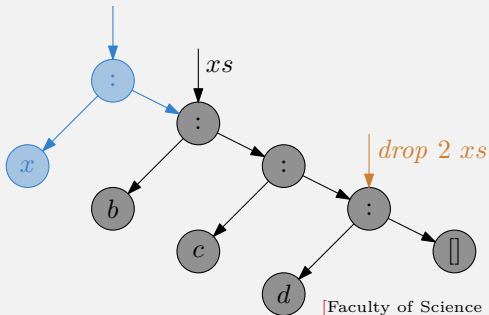
- What does `drop 2 xs` look like in memory?





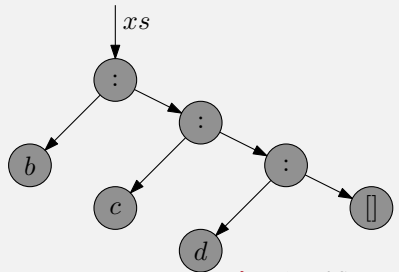
# Data Structures in Memory

- What does `drop 2 xs` look like in memory?



# Data Structures in Memory

- What does `1 ++ xs` look like in memory?

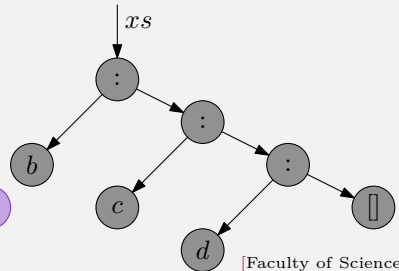
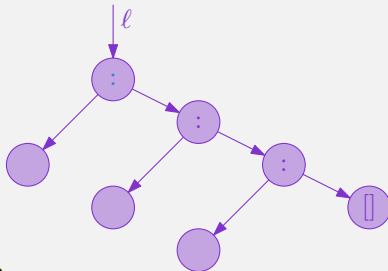


[Faculty of Science  
Information and Computing  
Sciences]



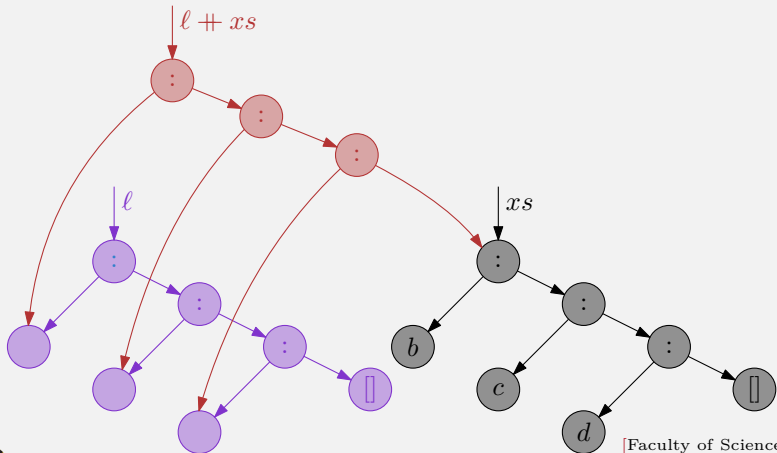
# Data Structures in Memory

- What does `1 ++ xs` look like in memory?



# Data Structures in Memory

- What does `l ++ xs` look like in memory?



# Persistent vs Ephemeral

- ▶ Data structures in which old versions are available are **persistent** data structures.
- ▶ Traditional data structures are **ephemeral**.



# Persistent vs Ephemeral

- ▶ Advantages of persistent data structures:
  - ▶ Convenient to have both old and new:
    - ▶ Separation of concerns;
    - ▶ Compute subexpressions independently
  - ▶ Output may contain old versions (i.e. tails)



# Can we get this for other data structures?

Yes\*!



Universiteit Utrecht

[Faculty of Science  
Information and Computing  
Sciences]

# Can we get this for other data structures?

Yes\*!

[\*] for a lot of them





# Successor Data Structure

- ▶ Store an set  $S$  of ordered elements s.t. we can efficiently find successor of a query  $q$ .
- ▶ The successor of  $q$  is the smallest element in  $S$  larger or equal to  $q$ .



# Implementing a Successor DS: Try 1, Lists

- ▶ Idea: Use an (unordered) list

```
type Successor a = [a]
```

- ▶ What should the type of our `succOf` function be?



# Implementing a Successor DS: Try 1, Lists

- ▶ Idea: Use an (unordered) list

```
type Successor a = [a]
```

- ▶ What should the type of our succOf function be?

```
succOf :: Ord a => a -> Successor a -> Maybe a
```



# Implementing a Successor DS: Try 1, Lists

```
succOf      :: Ord a => a -> Successor a -> Maybe a
succOf q s  = minimum' [ x | x <- s, x >= q]
  where
    minimum' [] = Nothing
    minimum' xs = Just (minimum xs)
```



# Implementing a Successor DS: Try 1, Lists

```
succOf      :: Ord a => a -> Successor a -> Maybe a
succOf q s  = minimum' [ x | x <- s, x >= q]
  where
    minimum' [] = Nothing
    minimum' xs = Just (minimum xs)
```

► Running time:  $O(n)$



# Implementing a Successor DS: Try 2, Ordered Lists

- Idea: Use an **ordered** list.

```
succOf q [] = Nothing
succOf q (x:s) | x < q = succOf q s
               | otherwise = Just x
```



# Implementing a Successor DS: Try 2, Ordered Lists

- Idea: Use an **ordered** list.

```
succOf q [] = Nothing
succOf q (x:s) | x < q = succOf q s
               | otherwise = Just x
```

- Does not really help: running time is still  $O(n)$ .



# Implementing a Successor DS: Try 2, Ordered Lists

- ▶ Idea: Use an **ordered** list.

```
succOf q [] = Nothing
succOf q (x:s) | x < q = succOf q s
               | otherwise = Just x
```

- ▶ Does not really help: running time is still  $O(n)$ .
- ▶ We need a better data structure.





# Implementing a Successor DS: Try 3, BSTs

- Idea: Use a binary search tree (BST).

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
  deriving (Show,Eq)
```

```
type Successor a = Tree a
```



# Implementing a Successor DS: Try 3, BSTs

- ▶ Idea: Use a binary search tree (BST).

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)
  deriving (Show,Eq)
```

```
type Successor a = Tree a
```

- ▶ Can we list all elements in a `Tree a`?
- ▶ Can we test if a `t :: Tree a` is a BST?



# Warmup: Listing The elements of a Tree

```
elems          :: Tree a -> [a]
elems Leaf     = []
elems (Node l x r) = elems l ++ [x] ++ elems r
```



# Warmup: Testing if a Tree is a BST?

```
isBST :: Ord a => Tree a -> Bool
isBST Leaf = True
isBST (Node l x r) = all (<= x) (elems l)
                    && all (>= x) (elems r)
                    && isBST l && isBST r
```

- ▶ This implementation uses  $O(n^2)$  time.
- ▶ Exercise: write an implementation that runs in  $O(n)$  time.



# Implementing a Successor DS: Queries

```
succOf q Leaf = Nothing
succOf q (Node l x r) | x < q = succOf q r
                      | otherwise =
    case succOf q l of
      Nothing -> Just x
      Just sq  -> Just sq
```



# Implementing a Successor DS: Queries

```
succOf q Leaf = Nothing
succOf q (Node l x r) | x < q = succOf q r
                      | otherwise =
    case succOf q l of
      Nothing -> Just x
      Just sq  -> Just sq
```

Nice if the input tree happens to be balanced, i.e. of height  $O(\log n)$



# Making Balanced Trees

- ▶ Suppose that the input is a sorted list, how to build a balanced tree?



# Making Balanced Trees

- Suppose that the input is a sorted list, how to build a balanced tree?

```
buildBalanced    :: [a] -> Tree a
buildBalanced [] = Leaf
buildBalanced xs = Node l x r
  where
    h = length xs `div` 2
    (ls,x:rs) = splitAt h xs

    l = buildBalanced ls
    r = buildBalanced rs
```

- Running time:  $O(n \log n)$ .





# Dynamic Successor: Insert

- Can we add new elements to the set  $S$ ?



# Dynamic Successor: Insert

- Can we add new elements to the set  $S$ ?

```
insert          :: Ord a => a -> Tree a -> Tree a
insert x Leaf   = Node Leaf x Leaf
insert x t@(Node l y r)
  | x < y       = Node (insert x l) y r
  | x == y      = t
  | otherwise   = Node l y (insert x r)
```



# Dynamic Successor: Insert

- Can we add new elements to the set  $S$ ?

```
insert          :: Ord a => a -> Tree a -> Tree a
insert x Leaf    = Node Leaf x Leaf
insert x t@(Node l y r)
  | x < y        = Node (insert x l) y r
  | x == y       = t
  | otherwise    = Node l y (insert x r)
```

- Not just insert  $x$   $l$ !
- Note that we are building new trees!

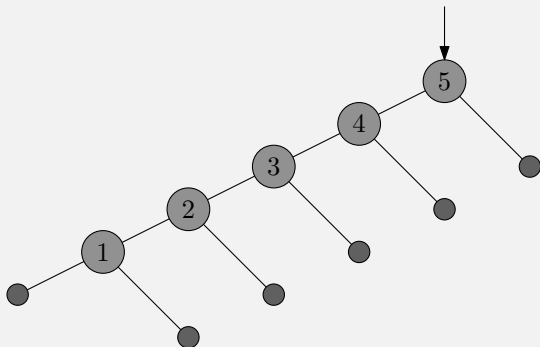


# May unbalance the tree

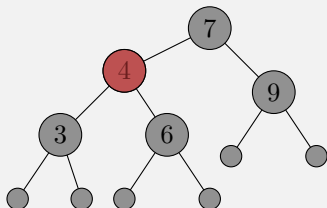
- Repeatedly inserting elements unbalances the tree

```
> foldr insert Leaf [1..5]
```

```
Node (Node (Node (Node (Node Leaf 1 Leaf) 2 Leaf) 3 Leaf) 4 Leaf) 5 Leaf
```



# Self balancing trees: Red Black Trees

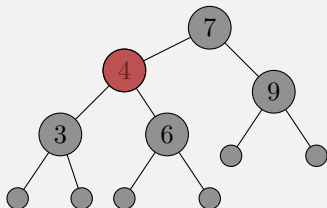


► Properties:

- 1) leaves are black
- 2) root is black
- 3) red nodes have black children
- 4) for any node, all paths to leaves have the same number of black children.



# Self balancing trees: Red Black Trees



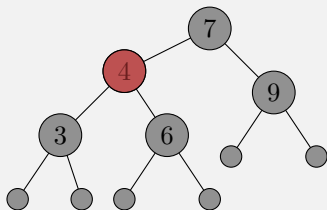
► Properties:

- 1) leaves are black
- 2) root is black
- 3) red nodes have black children
- 4) for any node, both children have the same **blackheight**

- **blackHeight** of a node = number of black children on any path from that node to its leaves.



# Self balancing trees: Red Black Trees



- ▶ Properties:
  - 1) leaves are black
  - 2) root is black
  - 3) red nodes have black children
  - 4) for any node, both children have the same **blackheight**
- ▶ Support queries and updates in  $O(\log n)$  time.



# Red Black Trees in Haskell

```
data Color = Red | Black deriving (Show,Eq)
```

```
data RBTREE a = Leaf  
              | Node Color (RBTREE a) a (RBTREE a)  
              deriving (Show,Eq)
```

- ▶ Enforces property 1. Other properties are more difficult to enforce in the type.





# Implementing Queries and Inserts

- ▶ succOf more or less the same as before.
- ▶ Insert:
  - ▶ Make sure black heights remain ok by replacing a black leaf by a red node.
  - ▶ The only issue is red,red violations.
  - ▶ Allow red,red violations with the root, but not below that.
  - ▶ Recolor the root black at the end.



# Implementing Insert

```
insert    :: Ord a => a -> RBTREE a -> RBTREE a  
insert x = blackenRoot . insert' x
```

```
insert'   :: Ord a => a -> RBTREE a -> RBTREE a
```



# Implementing Insert

```
insert    :: Ord a => a -> RBTREE a -> RBTREE a  
insert x = blackenRoot . insert' x
```

```
insert' :: Ord a => a -> RBTREE a -> RBTREE a
```

```
blackenRoot :: RBTREE a -> RBTREE a  
blackenRoot Leaf = Leaf  
blackenRoot (Node _ l y r) = Node Black l y r
```



# Implementing Insert'

```
insert' :: Ord a => a -> RBTREE a -> RBTREE a
insert' x Leaf = Node Red Leaf x Leaf
```



# Implementing Insert'

```
insert' :: Ord a => a -> RBTREE a -> RBTREE a
insert' x Leaf = Node Red Leaf x Leaf

insert' x t@(Node c l y r)
  | x < y      = balance c (insert' x l) y r
  | x == y     = t
  | otherwise  = balance c l y (insert' x r)
```



# Implementing Insert'

```
insert' :: Ord a => a -> RBTREE a -> RBTREE a
insert' x Leaf = Node Red Leaf x Leaf
```

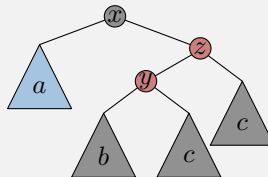
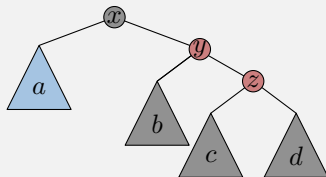
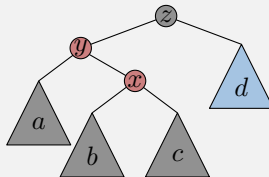
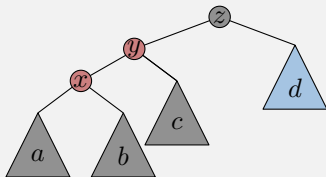
```
insert' x t@(Node c l y r)
  | x < y      = balance c (insert' x l) y r
  | x == y     = t
  | otherwise  = balance c l y (insert' x r)
```

```
balance :: Color -> RBTREE a -> a -> RBTREE a
          -> RBTREE a
```



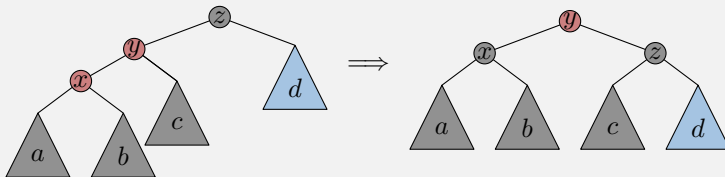
# Rebalancing

- ▶ The only potential issue is two red nodes near the root.
- ▶ There are only four configurations:



# Rebalancing

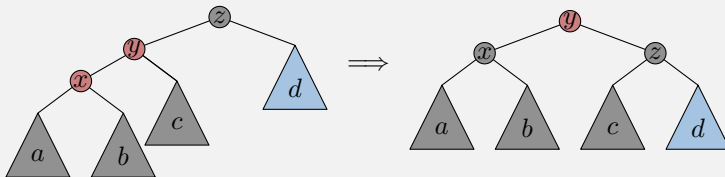
- Make the root red, and its children black:





# Rebalancing

- Make the root red, and its children black:



```
balance Black (Node Red (Node Red a x b) y c) z d =  
  Node Red (Node Black a x b) y (Node Black c z d)
```



## Rebalancing code

- ▶ Other cases are symmetric:

```
balance Black (Node Red (Node Red a x b) y c) z d =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance Black (Node Red a x (Node Red b y c)) z d =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance Black a x (Node Red (Node Red b y c) z d) =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance Black a x (Node Red b y (Node Red c z d)) =  
    Node Red (Node Black a x b) y (Node Black c z d)
```



## Rebalancing code

- Other cases are symmetric:

```
balance Black (Node Red (Node Red a x b) y c) z d =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance Black (Node Red a x (Node Red b y c)) z d =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance Black a x (Node Red (Node Red b y c) z d) =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance Black a x (Node Red b y (Node Red c z d)) =  
    Node Red (Node Black a x b) y (Node Black c z d)
```

```
balance c l x r  
    Node c l x r
```



# Deleting

- ▶ What if we also want to remove elements from  $S$ ?



# Deleting

- ▶ What if we also want to remove elements from  $S$ ?
- ▶ Possible in  $O(\log n)$  time with Red-Black trees, but a bit more messy.



# Data structures in the Haskell Standard Library

- ▶ Self balancing BST Implementation available in `Data.Set`
- ▶ Often useful to store additional information: `Data.Map`.

```
lookup :: Ord k => k -> Map k v -> Maybe v
```



# Data structures in the Haskell Standard Library

- ▶ Self balancing BST Implementation available in `Data.Set`
- ▶ Often useful to store additional information: `Data.Map`.

```
lookup :: Ord k => k -> Map k v -> Maybe v
```

- ▶ Finite Sequences: `Data.Sequence`, allow fast access to front and back.



# Data structures in the Haskell Standard Library

- ▶ Self balancing BST Implementation available in `Data.Set`
- ▶ Often useful to store additional information: `Data.Map`.

```
lookup :: Ord k => k -> Map k v -> Maybe v
```

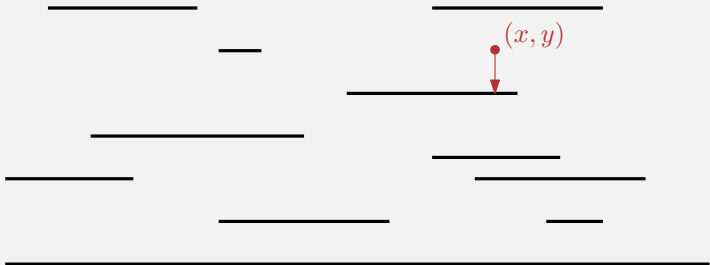
- ▶ Finite Sequences: `Data.Sequence`, allow fast access to front and back.
- ▶ All these data structures are persistent.





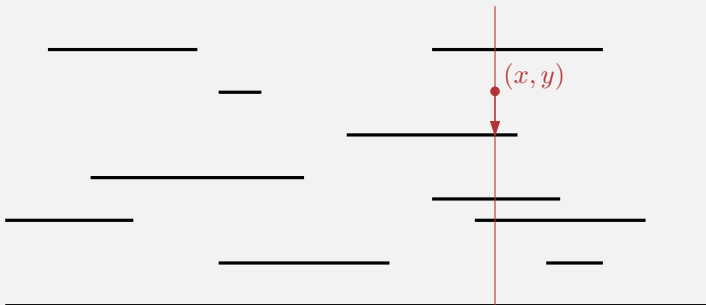
# Example Application: Point Location

- Can we quickly find the platform directly below Mario at  $(x, y)$ ?



# Example Application: Point Location

- Can we quickly find the platform directly below Mario at  $(x, y)$ ?

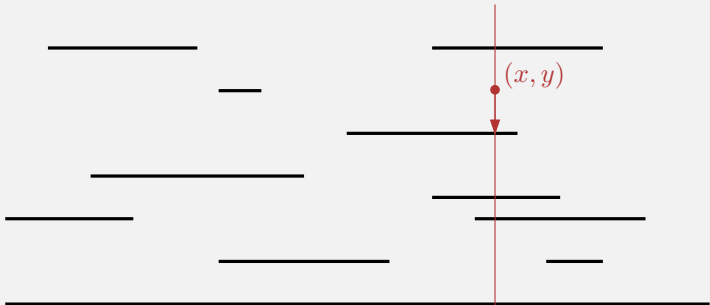


- Easy if we had the platforms intersecting the vertical line at  $x$  in a Set or Map: find predecessor of  $y$ .



# Example Application: Point Location

- ▶ Can we quickly find the platform directly below Mario at  $(x, y)$ ?

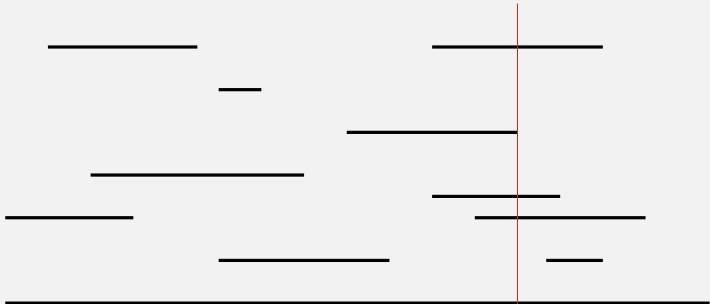


- ▶ What happens when vertical line starts/stops to intersect a platform?



# Example Application: Point Location

- Can we quickly find the platform directly below Mario at  $(x, y)$ ?

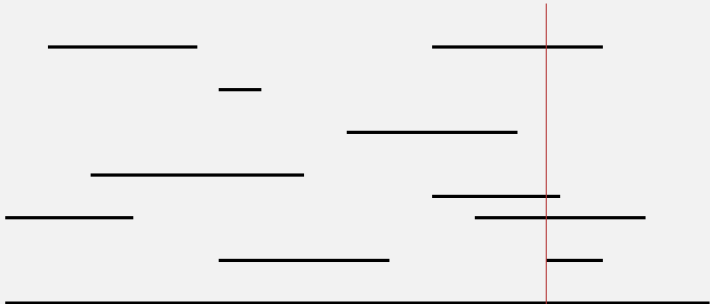


- What happens when vertical line starts/stops to intersect a platform?



# Example Application: Point Location

- ▶ Can we quickly find the platform directly below Mario at  $(x, y)$ ?



- ▶ What happens when vertical line starts/stops to intersect a platform?



# Example Application: Point Location

- ▶ Can we quickly find the platform directly below Mario at  $(x, y)$ ?
- ▶ What happens when vertical line starts/stops to intersect a platform?



# Example Application: Point Location

- ▶ Can we quickly find the platform directly below Mario at  $(x, y)$ ?
- ▶ What happens when vertical line starts/stops to intersect a platform?
- ▶ Add or remove a platform from the Set



# Example Application: Point Location

- ▶ Can we quickly find the platform directly below Mario at  $(x, y)$ ?
- ▶ What happens when vertical line starts/stops to intersect a platform?
- ▶ Add or remove a platform from the Set
- ▶ Since Set is persistent, old versions remain in tact. Store them in a Map.





# Example Application: Point Location

- ▶ Can we quickly find the platform directly below Mario at  $(x, y)$ ?
- ▶ What happens when vertical line starts/stops to intersect a platform?
- ▶ Add or remove a platform from the Set
- ▶ Since Set is persistent, old versions remain intact. Store them in a Map.
- ▶ To answer a query: go to the version at time  $x$  using a successor query, and find predecessor of  $y$ .



# Homework: Verifying Red-Black Tree Properties

- ▶ Write a function `validRBTree :: RBTree a -> Bool` that checks if a given `RBTree a` satisfies all red-black tree properties.

