Functional data structures

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau

Important comment on sources: Most code, text, and illustrations (modulo rephrasing or refactoring) have been extracted from the „Handbook of Data Structures and Applications“, Chapter 40 „Functional Data Structures“ by Chris Okasaki. At the time of writing (these slides), the handbook is freely available online: http://www.e-reading-lib.org/bookreader.php/138822/Mehta_-_Handbook_of_Data_Structures_and_Applications.pdf

Further sources are cited on individual slides.

Data structure

http://101companies.org/wiki/Data_structure

Headline

A particular way of storing and organizing data in a computer

Illustration

See linked lists as a simple example of an imperative data structure.

See immutable lists as a simple example of a functional data structure.

Resources

- Wikipedia
  - this ***sameAs*** http://en.wikipedia.org/wiki/Data structure

A **functional data structure** is a data structure that is suitable for implementation in a functional programming language, or for coding in an ordinary language like C or Java using a functional style. Functional data structures are closely related to **persistent** data structures and **immutable** data structures.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Stacks — a simple example

Stacks

- `empty`: a constant representing the empty stack.
- `push`(x,s): push the element x onto the stack s and return the new stack.
- `top`(s): return the top element of s.
- `pop`(s): remove the top element of s and return the new stack.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A functional data structure for stacks in *Haskell*

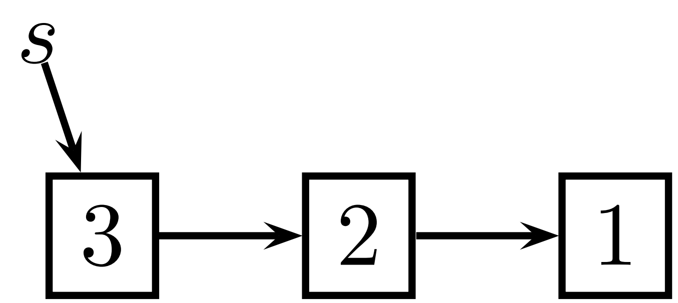
```
data Stack = Empty | Push Int Stack

empty = Empty
push x s  = Push x s
top (Push x s) = x
pop (Push x s) = s
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

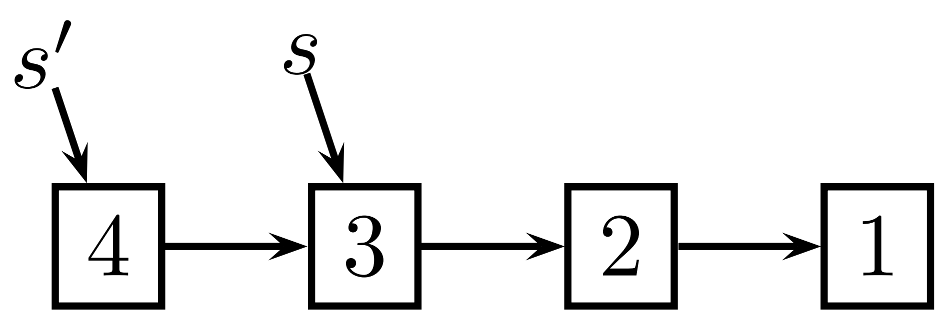
The „functional“ *push* operation

$$s' = \texttt{push}(4, s)$$

(Before)



(After)

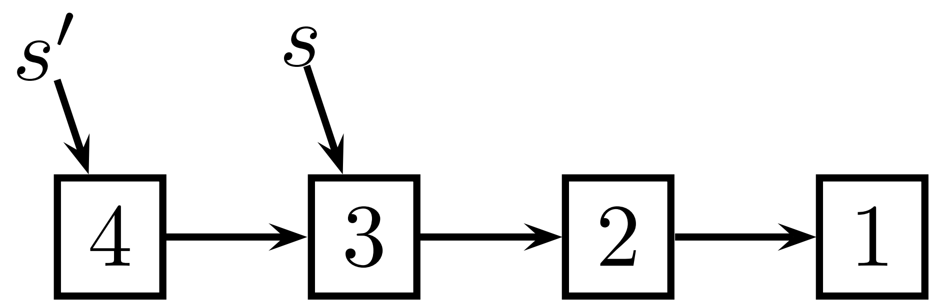


Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

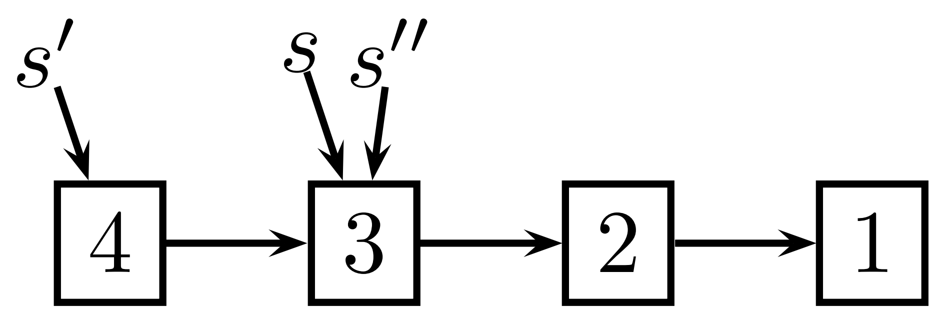
The „functional“ *pop* operation

$$s'' = \texttt{pop}(s')$$

(Before)



(After)



Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A functional data structure for stacks in *Java*

```
public class Stack {
  private int elem;
  private Stack next;
  public static final Stack empty = null;
  public static Stack push(int x,Stack s) {
    return new Stack(x,s);
  }
  public static int top(Stack s) { return s.elem; }
  public static Stack pop(Stack s) { return s.next; }
  private Stack(int elem, Stack next) {
    this.elem = elem;
    this.next = next;
  }
}
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A **non-**functional data structure for stacks in *Java*

```java
public class Stack {
    private class Node {
        private int elem;
        private Node next;
    }
    private Node first;
    public Stack() {} // "empty"
    public void push(int x) {
        Node n = new Node();
        n.elem = x;
        n.next = first;
        first = n;
    }
    public int top() { return first.elem; }
    public void pop() { first = first.next; }
}
```

Terminology
&
characteristics

A **functional data structure** is a data structure that is suitable for implementation in a functional programming language, or for coding in an ordinary language like C or Java using a functional style. Functional data structures are closely related to **persistent** data structures and **immutable** data structures.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

persistent

immutable

functional

- The term **persistent data structures** refers to the general class of data structures in which an update does not destroy the previous version of the data structure, but rather creates a new version that co-exists with the previous version. See the handbook (Chapter 31) for more details about persistent data structures.

- The term **immutable data structures** emphasizes a particular implementation technique for achieving persistence, in which memory devoted to a particular version of the data structure, once initialized, is never altered.

- The term **functional data structures** emphasizes the language or coding style in which persistent data structures are implemented. Functional data structures are always immutable, except in a technical sense discussed (related to laziness and memoization).

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Functional programming specifics related to data structures

- **Immutability** as opposed to imperative variables
- **Recursion** as opposed to control flow with loops
- **Garbage collection** as opposed to malloc/dealloc
- **Pattern matching**

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Perceived advantages of functional data structures

- **Fewer bugs** as data cannot change suddenly
- **Increased sharing** as defensive cloning is not needed
- **Decreased synchronization** as a consequence

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Sets — another example

Sets

```
data Set e s = Set {
  empty :: s e,
  insert :: e -> s e -> s e,
  search :: e -> s e -> Bool
}
```

Let's look at different implementations of this signature!

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A naive, equality- and list-based implementation of sets in *Haskell*

```
set :: Eq e => Set e []
set = Set {
  empty = [],
  insert = \e s ->
    case s of
      [] -> [e]
      s'@(e':s'') ->
        if e==e'
          then s'
          else e':insert set e s'',
  search = \e s ->
    case s of
      [] -> False
      (e':s') -> e==e' || search set e s'
}
```

The time complexity is embarrassing: insertion and search takes time proportional to the size of the set.

Sets based on binary search trees in *Haskell*

```
data BST e = Empty | Node (BST e) e (BST e)

set :: Ord e => Set e BST
set = Set {

  empty = Empty,

  insert = ...,

  search = ...

}
```

That is, we go for another implementation with, hopefully, better time complexity.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Sets based on binary search trees in *Haskell*

The running time of search is proportional to the length of the search path — just like in a non-persistent implementation.

```
search = \e s ->
  case s of
    Empty -> False
    (Node s1 e' s2) ->
      if e<e'
        then search set e s1
        else if e>e'
          then search set e s2
          else True
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Sets based on binary search trees in *Haskell*

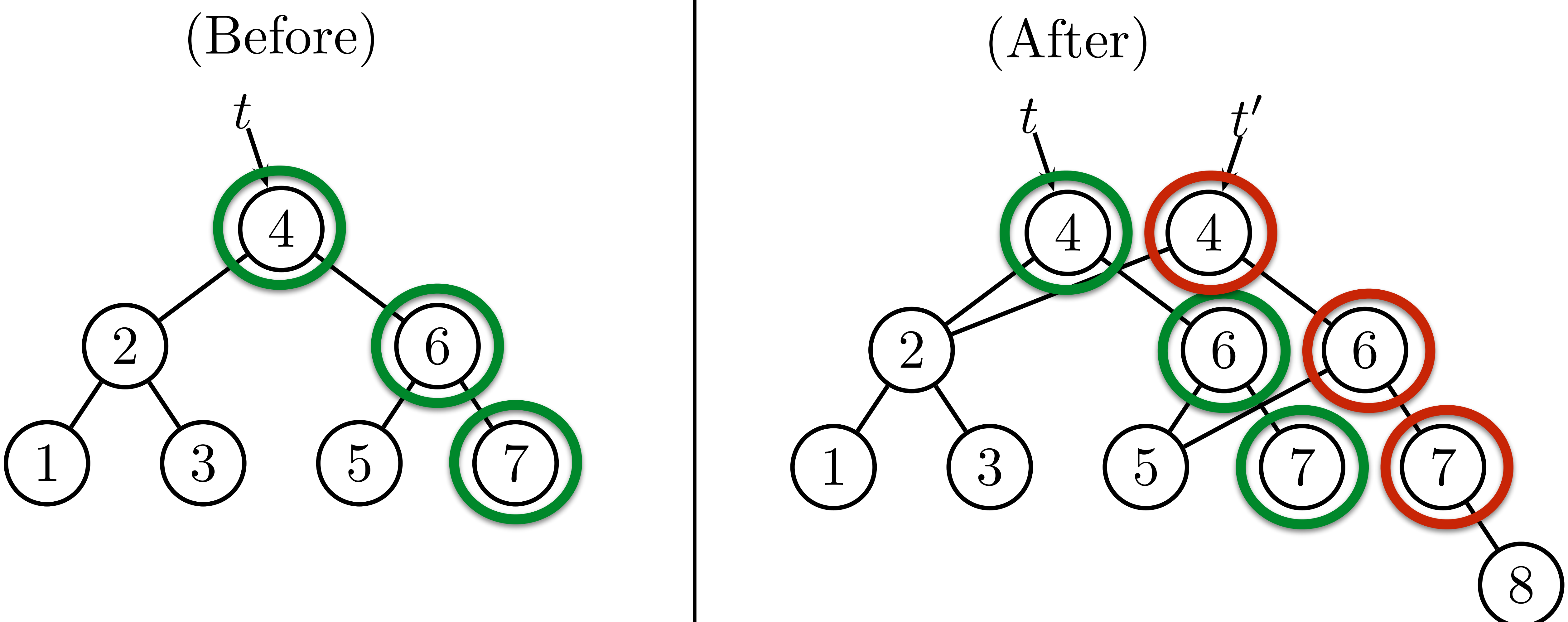
The running time of insert is also proportional to the length of the search path.

```
insert = \e s ->
  case s of
    Empty -> Node Empty e Empty
    (Node s1 e' s2) ->
      if e<e'
        then Node (insert set e s1) e' s2
        else if e>e'
          then Node s1 e' (insert set e s2)
          else Node s1 e' s2,
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Operations of functional data structures involve ***path copying***

For example: $t' = \texttt{insert}(8, t)$



Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Benchmark results

https://github.com/101companies/101repo/tree/master/concepts/Functional_data_structure/Set

```
benchmarking NaiveSet/insert
mean: 5.673453 ms, lb 5.610866 ms, ub 5.836548 ms, ci 0.950
std dev: 480.9444 us, lb 228.4352 us, ub 986.8636 us, ci 0.950
found 16 outliers among 100 samples (16.0%)
  4 (4.0%) high mild
  12 (12.0%) high severe
variance introduced by outliers: 72.809%
variance is severely inflated by outliers
```

```
benchmarking BinarySearchTree/insert
mean: 241.3734 us, lb 240.6849 us, ub 242.4783 us, ci 0.950
std dev: 4.375792 us, lb 3.020795 us, ub 7.339799 us, ci 0.950
found 35 outliers among 100 samples (35.0%)
  15 (15.0%) low severe
  5 (5.0%) low mild
  2 (2.0%) high mild
  13 (13.0%) high severe
variance introduced by outliers: 11.315%
variance is moderately inflated by outliers
```

Insert is (much) faster with binary search trees.

Benchmark results

https://github.com/101companies/101repo/tree/master/concepts/Functional_data_structure/Set

```
benchmarking NaiveSet/search
mean: 38.35384 us, lb 36.66107 us, ub 40.54014 us, ci 0.950
std dev: 9.812249 us, lb 8.019951 us, ub 11.71828 us, ci 0.950
found 10 outliers among 100 samples (10.0%)
  10 (10.0%) high mild
variance introduced by outliers: 96.775%
variance is severely inflated by outliers
```

Search is (much) faster with binary search trees.

```
benchmarking BinarySearchTree/search
mean: 1.606348 us, lb 1.576601 us, ub 1.645087 us, ci 0.950
std dev: 172.8071 ns, lb 139.6882 ns, ub 203.6180 ns, ci 0.950
found 16 outliers among 100 samples (16.0%)
  15 (15.0%) high severe
variance introduced by outliers: 82.070%
variance is severely inflated by outliers
```

Discussion of binary search trees

- „Of course“, a balanced variation would be needed:
  - AVL trees
  - Red-black trees
  - 2-3 trees
  - Weight-balanced trees
- Path copying still applies
  - Time complexity Ok
  - Space complexity Ok because of garbage collection

Priority queues — a tougher example

Priority queues

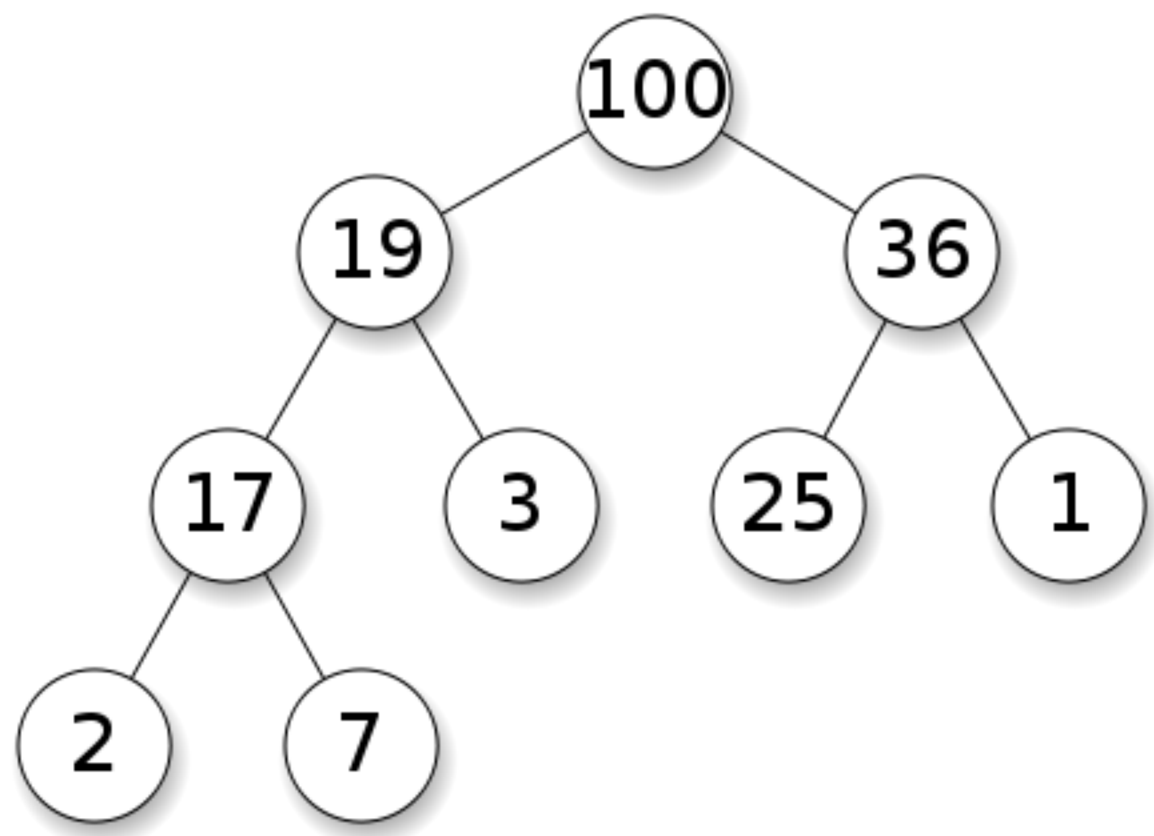
- `empty`: a constant representing the empty heap.
- `insert(`x`,`h`)`: insert the element x into the heap h and return the new heap.
- `findMin(`h`)`: return the minimum element of h.
- `deleteMin(`h`)`: delete the minimum element of h and return the new heap.
- `merge(`h_1`,`h_2`)`: combine the heaps h_1 and h_2 into a single heap and return the new heap.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Heaps: an efficient implementation of priority queues

- A tree structure with keys at the nodes.
- Max-heap: maximum key value always at the root.
- Min-heap: minimum key value always at the root.
- Note:
  - No particular order on the children.
  - Heaps are essentially *partially* ordered trees.

Example of a (complete) binary max-heap with node keys being integers from 1 to 100



Source: http://en.wikipedia.org/wiki/Heap_(data_structure)#mediaviewer/File:Max-Heap.svg

A complete binary tree of size *N* has height O(log *N*).

Signature of *heaps*

```
data Heap e t = Heap {
  empty :: t e,
  insert :: e -> t e -> t e,
  findMin :: t e -> Maybe e,
  deleteMin :: t e -> Maybe (t e),
  merge :: t e -> t e -> t e
}
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A tree-based representation type for *heaps*

```
data Tree e
  = Empty
  | Node e (Tree e) (Tree e)
   deriving (Eq, Show)

leaf e = Node e Empty Empty
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

This is not yet „optimal“.

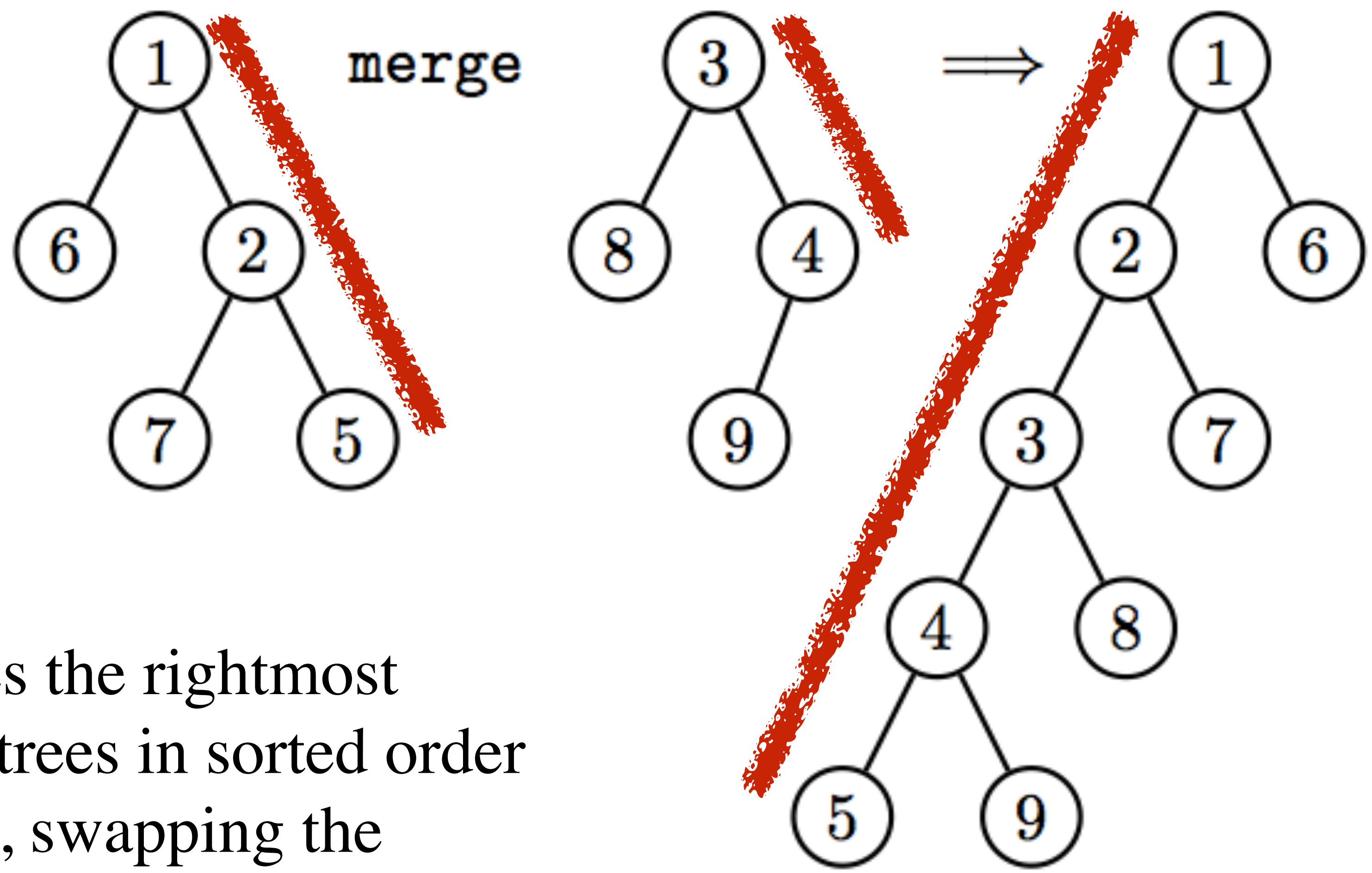
```
heap = Heap {
  empty = Empty,
  insert = \x t -> merge' (Node x Empty Empty) t,
  findMin = \t -> case t of
    Empty -> Nothing
    (Node x _ _) -> Just x,
  deleteMin = \t -> case t of
    Empty -> Nothing
    (Node _ l r) -> Just (merge' l r),
  merge = \l r -> case (l, r) of
    (Empty, t) -> t
    (t, Empty) -> t
    (t1@(Node x1 l1 r1), t2@(Node x2 l2 r2)) ->
      if x1 <= x2
        then Node x1 (merge' l1 r1) t2
        else Node x2 t1 (merge' l2 r2)
}
  where merge' = merge heap
```

Let's make our heaps self-adjusting.
We swap arguments of merge.
These are so-called skew heaps.

```
heap = Heap {
  empty = Empty,
  insert = \x t -> merge' (Node x Empty Empty) t,
  findMin = \t -> case t of
    Empty -> Nothing
    (Node x _ _) -> Just x,
  deleteMin = \t -> case t of
    Empty -> Nothing
    (Node _ l r) -> Just (merge' r l),
  merge = \l r -> case (l, r) of
    (Empty, t) -> t
    (t, Empty) -> t
    (t1@(Node x1 l1 r1), t2@(Node x2 l2 r2)) ->
      if x1 <= x2
        then Node x1 (merge' t2 r1) l1
        else Node x2 (merge' t1 r2) l2
}
  where merge' = merge heap
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Merging two skew heaps



Merge interleaves the rightmost paths of the two trees in sorted order (on the left path), swapping the children of nodes along the way.

Without swapping, the rightmost path would get „too“ long.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A functional data structure for skew heaps in *Java*

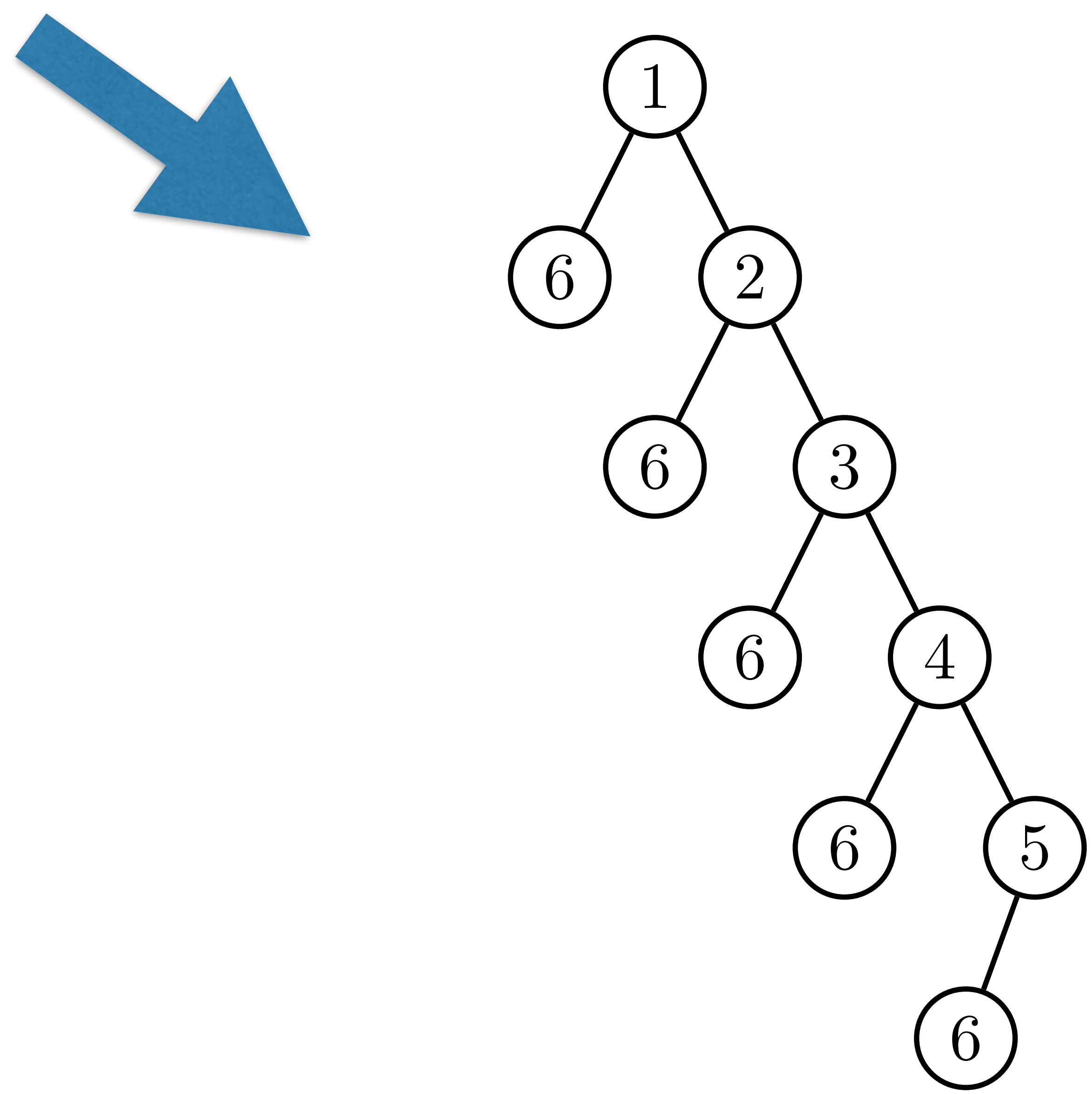
```java
public class Skew {
    public static final Skew empty = null;
    public static Skew insert(int x,Skew s) { return merge(new Skew(x,null,null),s); }
    public static int findMin(Skew s) { return s.elem; }
    public static Skew deleteMin(Skew s) { return merge(s.left,s.right); }
    public static Skew merge(Skew s,Skew t) {
        if (t == null) return s;
        else if (s == null) return t;
        else if (s.elem < t.elem)
            return new Skew(s.elem,merge(t,s.right),s.left);
        else
            return new Skew(t.elem,merge(s,t.right),t.left);
    }
    private int elem;
    private Skew left,right;
    private Skew(int elem, Skew left, Skew right) {
        this.elem = elem; this.left = left; this.right = right;
    }
}
```

We will need to revise this implementation.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

The shown tree is an unbalanced skew heap generated by inserting the listed numbers.

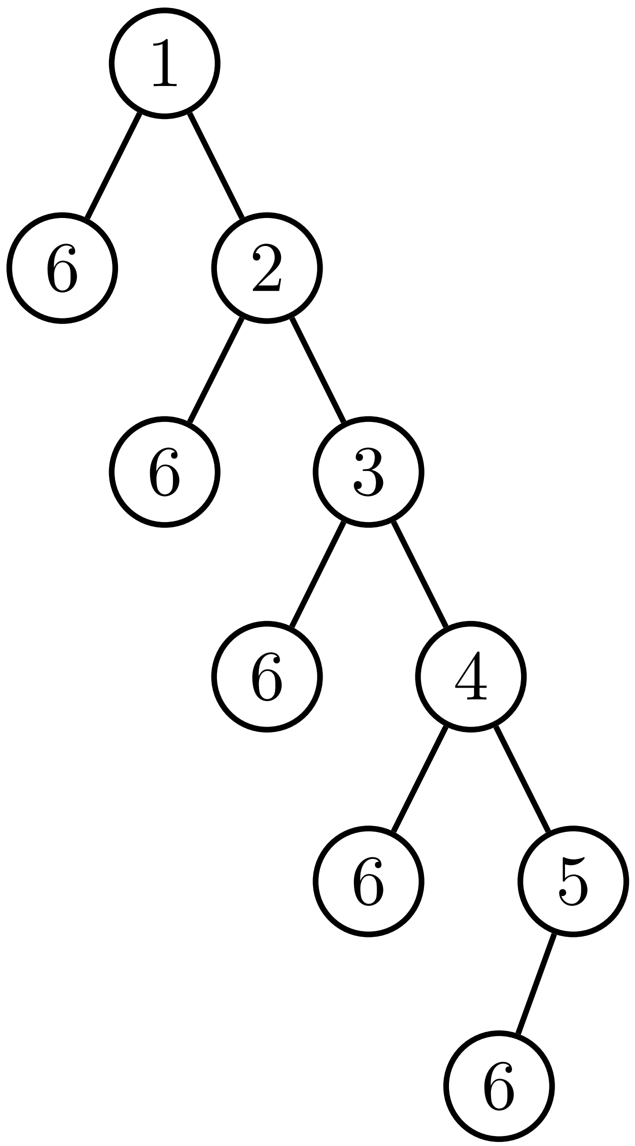
[5, 6, 4, 6, 3, 6, 2, 6, 1, 6]



Skew heaps are not balanced, and individual operations can take linear time in the worst case.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Complexity of operation sequences



Inserting a new element such as 7 into this unbalanced skew heap would take linear time. However, in spite of the fact that any one operation can be inefficient, **the way that children are regularly swapped keeps the operations efficient „in average“.** Insert, deleteMin, and merge run in logarithmic (amortized) time.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Amortization

SIAM J. COMPUT.
Vol. 15, No. 1, February 1986

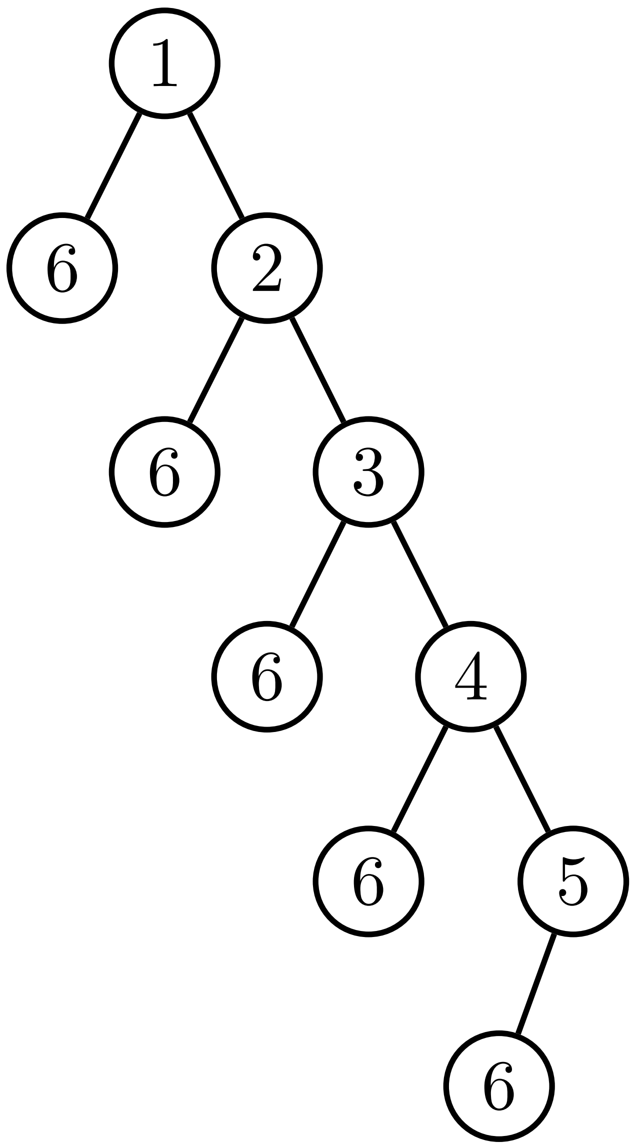
SELF-ADJUSTING HEAPS*

DANIEL DOMINIC SLEATOR† AND ROBERT ENDRE TARJAN†

Abstract. In this paper we explore two themes in data structure design: *amortized computational complexity* and *self-adjustment*. We are motivated by the following observations. In most applications of data structures, we wish to perform not just a single operation but a sequence of operations, possibly having correlated behavior. By averaging the running time per operation over a worst-case sequence of operations, we can sometimes obtain an overall time bound much smaller than the worst-case time per operation multiplied by the number of operations. We call this kind of averaging *amortization*.
...

Available online: https://www.cs.cmu.edu/~sleator/papers/adjusting-heaps.pdf

Persistence may break amortized bounds.

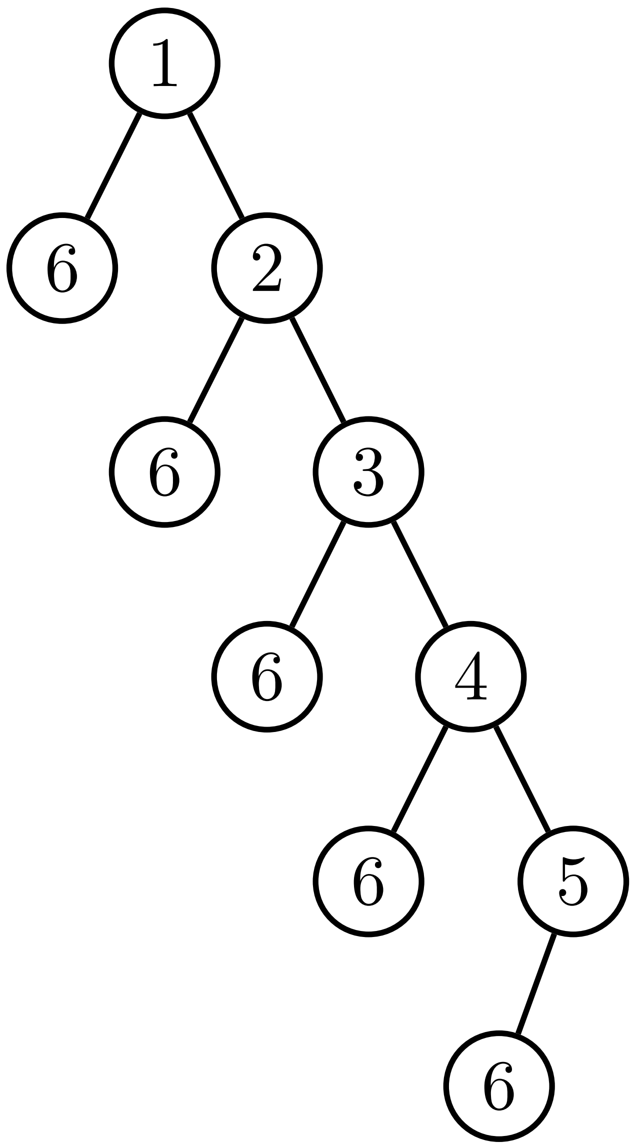


However, naively incorporating path copying causes the logarithmic amortized bounds to degrade to the linear worst-case bounds.

To see this, consider repeated insertion of large elements into a tree. Each insertion could be applied to the original tree. Thus, each insertion would have linear costs resulting also in average linear costs.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Impact of laziness

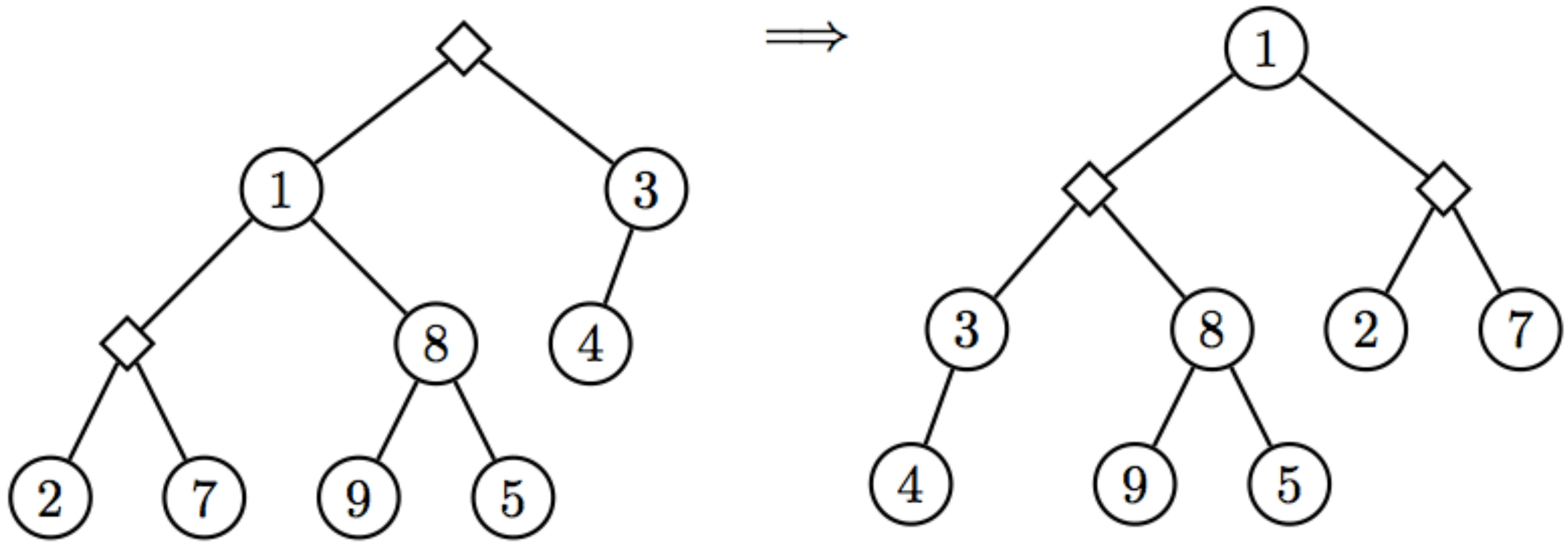


If we benchmark the Haskell implementation, we do not observe linear behavior though! Instead, the operations appear to retain their logarithmic amortized bounds, even under persistent usage. This pleasant result is a consequence of a fortuitous interaction between path copying and lazy evaluation.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

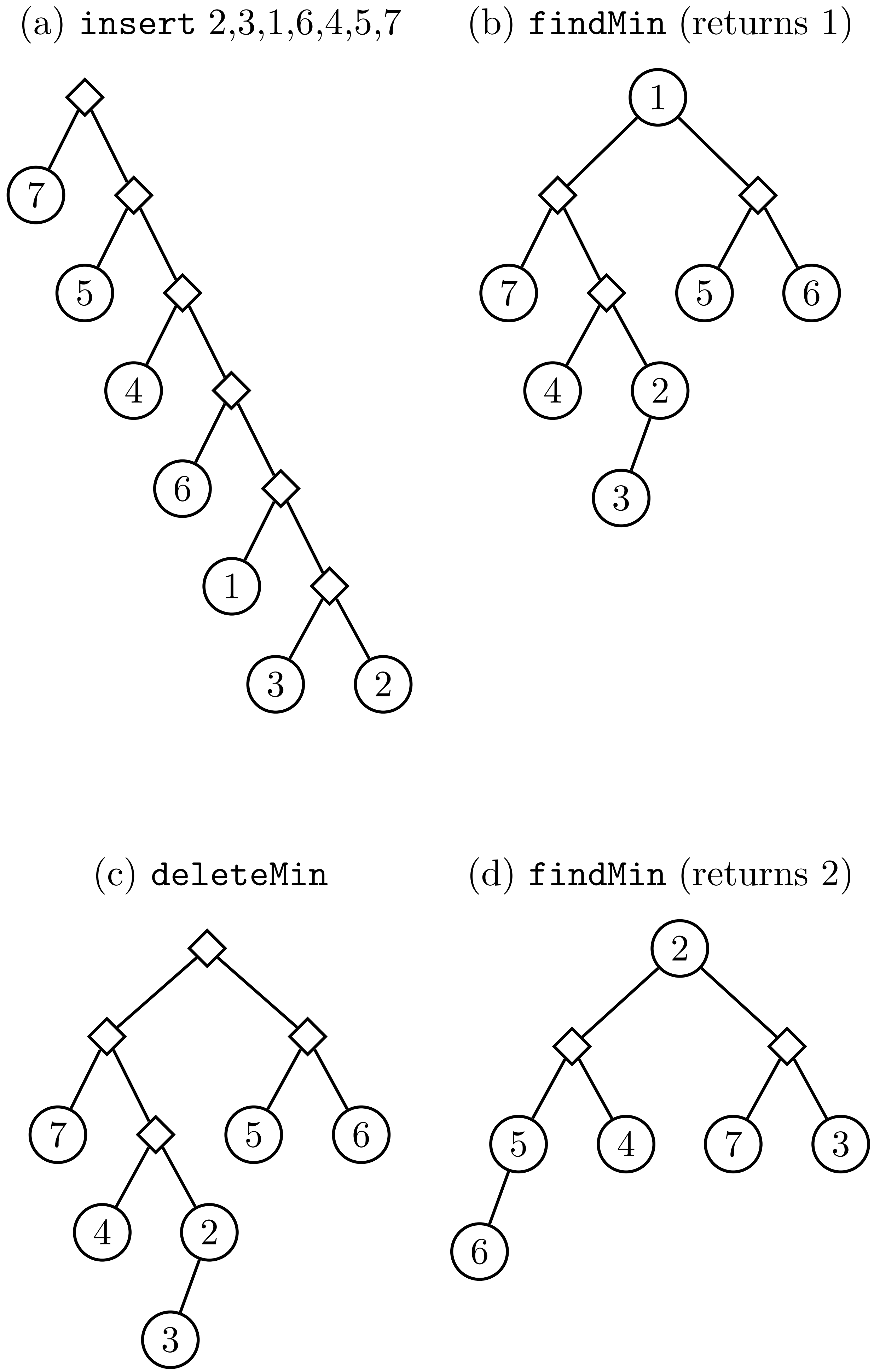
Pending merge

findMin



Under lazy evaluation, operations such as *merge* are not actually executed until their results are needed. Instead, a new kind of node that we might call a pending merge (see the diamonds) is automatically created. The pending merge lays dormant until some other operation such as *findMin* needs to know the result. Then and only then is the pending merge executed. The node representing the pending merge is overwritten with the result so that it cannot be executed twice. (This is benign mutation.)

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.



A sequence of operations

Pending merges do not affect the end results of those steps. After all the pending merges have been executed, the final tree is identical to the one produced by skew heaps without lazy evaluation. (Printing the tree would execute all pending nodes!) Some functional languages allow this kind of mutation, known as memoization, because it is invisible to the user, except in terms of efficiency.

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

A Java implementation with pending merges

```java
public class Skew {
    private int elem;
    private Skew left,right;
    private boolean pendingMerge;
    public static final Skew empty = null;
    public static Skew insert(int x,Skew s) {
        return merge(new Skew(x,null,null),s);
    }
    public static int findMin(Skew s) {
        executePendingMerge(s);
        return s.elem;
    }
    public static Skew deleteMin(Skew s) {
        executePendingMerge(s);
        return merge(s.left,s.right);
    }
    public static Skew merge(Skew s,Skew t) {
        if (t == null) return s;
        else if (s == null) return t;
        else return new Skew(s,t); // create a pending merge
    }
    private Skew(int elem, Skew left, Skew right) { ... }
    private Skew(Skew left,Skew right) { ... } // create a pending merge
    private static void executePendingMerge(Skew s) { ... }
}
```

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

```java
private Skew(int elem, Skew left, Skew right) {
    this.elem = elem;
    this.left = left;
    this.right = right;
    pendingMerge = false;
}
private Skew(Skew left,Skew right) { // create a pending merge
    this.left = left;
    this.right = right;
    pendingMerge = true;
}
private static void executePendingMerge(Skew s) {
    if (s != null && s.pendingMerge) {
        Skew s1 = s.left, s2 = s.right;
        executePendingMerge(s1);
        executePendingMerge(s2);
        if (s2.elem < s1.elem) {
            Skew tmp = s1;
            s1 = s2; s2 = tmp;
        } s.elem = s1.elem;
        s.left = merge(s2,s1.right);
        s.right = s1.left;
        s.pendingMerge = false;
    }
}
```

A Java implementation with pending merges

Source: Chapter 40: Functional Data Structures by C. Okasaki. In: Handbook of Data Structures and Applications. Chapman & Hall/CRC.

Summary

- Functional DS are persistent and in „functional style“.
- We looked at stacks, sets, and heaps.
- Functional and „non“-f. DS can be equally efficient.
- Lazy evaluation includes memoization.
- Have a look at methods of amortized analysis*!*