On Functional Programming

Functional Data Structures

Daniil Berezun

2022

New concepts

- *Immutable* data structures
- *Persistent* data structures

Remarks

- We can use *old* nodes (*share*) in new version of the data structure
- Non-persistent data structures are called *ephemeral*

Linked List

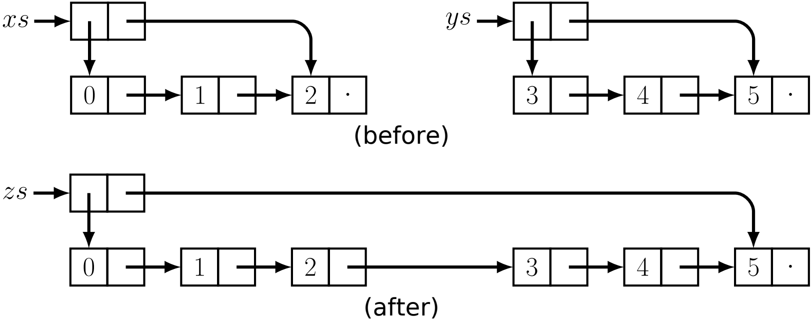
Definition (Linked List)

Who knows?

Definition (List) [One of possible definitions]

A data structure such that from some predefined side (for example, list head) deletion and insertion of element has complexity $O(1)$

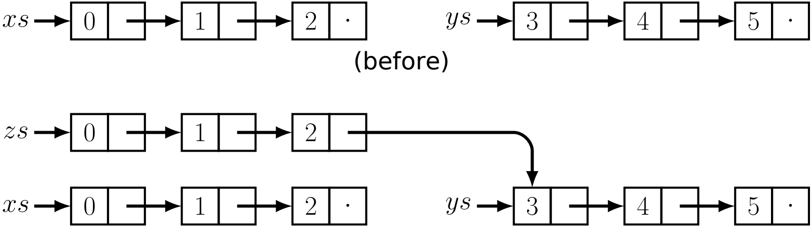
List Concatenation in the Imperative Paradigm



Concatenation of lists xs and ys in the imperative paradigm

- Destroys argument lists xs and ys (one can't use them further)
- Complexity: $O(1)$

Pure Functional Lists Concatenation



(before)

(after)

Execution of `zs = xs ++ ys` in functional world

- xs and ys remain intact
- we copied **a lot** but the first list only

Pure Functional Lists Concatenation — 2

How to implement concatenation ++ of lists xs and ys?

- If xs is empty then ys is the answer
- Otherwise xs consists of h as a head and tl as a tail then the answer is a list with head h and tail tl++ys.

Complexity: $O(length(xs))$

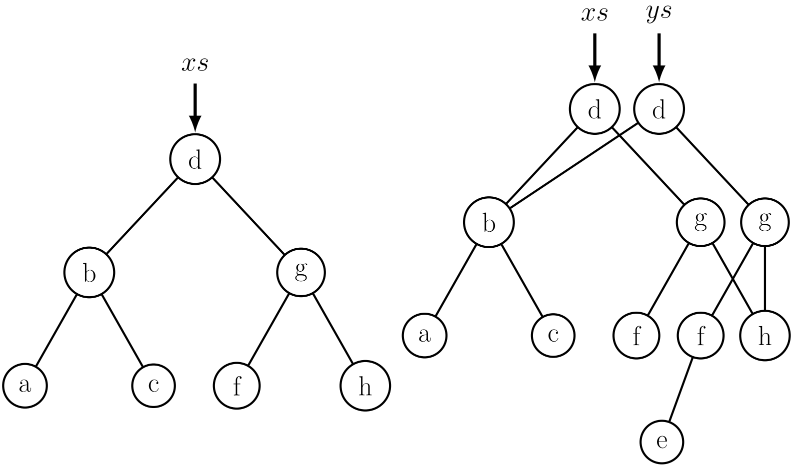
```
(++) []     ys = ys
(++) (h:tl) ys = h : (tl ++ ys)
```

How to update the n-th list element?

```
update []   i y = error "i is greater than list length"
update x:xs 0 y = y:xs
update x:xs i y = x : update xs (i-1) y
```

- $O(n)$ ... very sad ;(
- We copy the element being modified **and** all elements that have direct or indirect pointers to it

Example: Trees



- Usually, the number of nodes to be copied is at most $\log_2 n$

On Concatenation Associativity

In theory list concatenation is associative

$$(((a_1 \mathbin{+\!\!+} a_2) \mathbin{+\!\!+} a_3) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} a_n) \equiv (a_1 \mathbin{+\!\!+} (a_2 \mathbin{+\!\!+} (a_3 \mathbin{+\!\!+} (\ldots \mathbin{+\!\!+} a_n))))$$

In practise left-had side is much slower than right-hand side

Note for developers

Sometimes, for an efficient implementation one need to redesign algorithms in a way such that shorter lists are concatenated with longer lists. Ideally, always concatenate one element with a list.

On Amortized Time Analysis

Standard complexity notation $O(\cdot)$ – worst case estimation

But actually, we may have more freedom:

- Let's perform $n+1$ action
- Most of actions will be "cheap": $O(1)$
- One "expensive" action: for example, $O(n)$
- Standard assymptotic compexity: $O(n)$
- Average complexity of performing n actions (*amortized time complexity*) can be $O(1)$ for an action

$$a = \frac{\sum_{i=1}^{n} t_i}{n}$$

This additional freedom degree sometimes allows a simpler and more efficient implementation to be designed

Banker’s Method

Definition (*Accumulated Savings*)

A difference between total current amortized cost and total current fair value

- **NB:** accumulated savings must be **non-negative**
- I.e. “expensive” operations may take place iff accumulated savings are enough to cover theis additional cost

$$a_i = t_i + c_i - \bar{c}_i$$

where t_i — fair cost, c_i — credit amount provided by action i, $\bar{c}_i$ — amount of credit spent by action i

- Each credit unit must be allocated before being spent
- Credit cannot be used twice
- $\sum c_i \geq \sum \bar{c}_i \Rightarrow \sum a_i \geq \sum t_i$
- Amortized compexity is $n * O(f(n,m)) \Leftrightarrow \forall n . a_i = O(f(n,m))$ $\Rightarrow a = \frac{\sum_{i=1}^{n} a_i}{n} = \frac{n*O(f(n,m))}{n} = O(f(n,m))$

Pure Functional Queues

Interface:

- `empty: queue -> bool`
- `enqueue: queue * int -> queue`
- `head: queue -> int`
- `tail: queue -> queue`

Simplest implementation

Via a pair of lists, `f` and `r`

- `f` (front) contains the head elements of the queue in the initial (correct) order,
- `r` (reversed) consist of tail elements in reverse order

For example, queue `=[1;2;3;4;5;6]` can be represented as two lists `f=[1;2;3]` and `r=[6;5;4]`

The Queue Invariant

Question: When to move elements from the front to reversed list?

Definition (Queue Invariant)

List `f` may become empty iff list `r` is also empty (i.e., the queue is empty)

Otherwise `head` is $O(n)$

Pure Functional Queue and Banker’s method

Definition (Invariant)

Each element in the *tail* list is associated with one credit unit

- Each `enqueue` call performs the only real computational step and emits *edditional* credit unit for an element in the tail list

  amortized complexity is 2

- `tail`, if no list inversion happend, preforms one step and spends no credit units

  amortized compelxity is 1

- `tail`, if list reverse happends, performs $(m+1)$ steps, where m is a tail list length, and spends m credit units

  amortized complexity is $m+1-m=1$

Conclusion

- In case of purely functional queue, function `tail` worst case complexity is $O(n)$ and amortized — $O(1)$
- Good if one do not need persistency, and amortized performance is good enough for the problem
- Lazy evaluations $+$ amortized compexity $=$ persistent queues with a very good amortized complexity

Lazy Evaluations

Lazy Evaluations

Delays the evaluation of an expression until its value is needed (*non-strict evaluation*)

Memoization of lazy evaluations

Ones the value of expression is needed, evaluate it and *memoize* (remember, *sharing*) the result; If it will be needed further, just return the memoized result

Lazy Lists (Streams)

Definition (Stream)

is a list but evaluations of sublists are delayed

Example: Stream of all possible natural numbers

Notation

Add an element *x* to the tail *xs*: $*Cons x xs*
Empty stream: $*Nil*
Delay *f*: $*f*

Remark

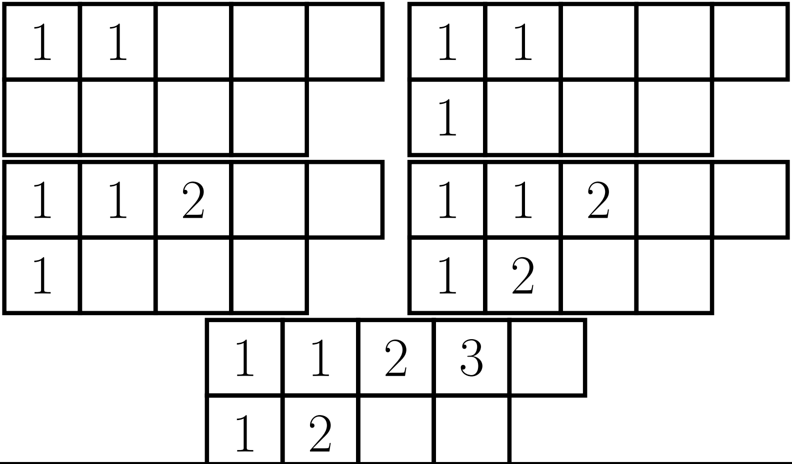
Stream may be both finite and infinite;
One never knows until the end appears

Example: Fibonacci Numbers

Consider function `zip` : *stream* × *stream* → *stream*, which sums streams element by element

A stream of Fibonacco numbers:

$$\mathit{fibs} \equiv \texttt{\$Cons(1,\$Cons(1,zip(fibs,tail(fibs))))}$$



1	1			

1	1			
1				

1	1	2		
1				

1	1	2		
1	2			

1	1	2	3	
1	2			

Banker's Queue Improvement

Remark

This implementation has amortized complexity $O(1)$ and is persistent

1. Use streams instead of lists
2. Store stream lengths explicitly
3. Invariant: $|f| \geq |r|$

If streams f and r have the same length, define f as $f +\!\!+\ \mathit{reverse}(r)$

Reverse

- Lazy evaluation ⇒ delayed until needed
- Memoization ⇒ computed only ones

Scheduling

Problem Statement

- We produce *n* “cheap” steps
- Then, one “expensive” step $O(n)$
- Thus, we can only state amortized complexity

An Idea: *scheduling*

Instead of one “expensive” step let’s perform *n* smaller steps with constant complexity. Performing each “cheap” step, we will also perform one of this “smaller” steps.

Real-time Queue

Reminder: banker's queue: we relied on calculation of $\mathtt{f} +\!\!+ \mathit{reverse}(\mathtt{r})$

Now let's instead use a special function `rotate`

$$\mathtt{rotate}(\mathtt{f}, \mathtt{r}, a) = \mathtt{f} +\!\!+ \mathit{reverse}(\mathtt{r}) +\!\!+ a$$

Third parameter is an accomulator which stores partially computed result of $\mathit{reverse}(\mathtt{r})$

Obviously

$$\mathit{rotate}(\mathtt{f}, \mathtt{r}, \$\mathit{Nil}) = \mathtt{f} +\!\!+ \mathit{reverse}(\mathtt{r})$$

When to Reorder the Queue?

Let's reorder queue when $|\texttt{r}| = |\texttt{f}| + 1$
This ratio will be maintained throughout the rebuilding

Let's prove it by induction on the length of front $|\texttt{f}|$

Base:

$$\begin{aligned} \mathit{rotate}(\texttt{\$Nil}, \texttt{\$Cons(y,\$Nil)}, a) &\equiv \texttt{\$Nil} +\!\!+ \mathit{reverse}(\texttt{\$Cons(y,\$Nil)}) +\!\!+ a \\ &\equiv \texttt{\$Cons(y,a)} \end{aligned}$$

Induction step:

$$\begin{aligned} &\mathit{rotate}(\texttt{\$Cons(x,f)}, \texttt{\$Cons(y,r)}, a) \\ &\quad\equiv \texttt{\$Cons(x,f)} +\!\!+ \mathit{reverse}(\texttt{\$Cons(y,r)}) +\!\!+ a \\ &\quad\equiv \texttt{\$Cons(x,f } +\!\!+ \texttt{reverse(\$Cons(y,r)) } +\!\!+ \texttt{a)} \\ &\quad\equiv \texttt{\$Cons(x,f } +\!\!+ \texttt{reverse(r) } +\!\!+ \texttt{\$Cons(y,a))} \\ &\quad\equiv \texttt{\$Cons(x,rotate (f, r, \$Cons(y,a)))} \end{aligned}$$

Conclusion

Queue\ Operation	enqueue	head	tail
Banker’s	$O(1)^*$	$O(1)^*$	$O(1)^*$
Real-time	O(1)	O(1)	O(1)

Amortized estimations are marked as c^*

TODO

- Sets?

Left-oriented heap — 1

Priority Queue (or *heap*)

Data structure that supports efficient access to the minimum element

Remark

Note *order* relation in the heap signature (unlike sets)

Left-oriented heap

- *leftist property*: rank of any left subtree is not less than rank of its right sister node
- *rank* is a length of the *right spane*
- Thus, right spane is a shortest path to a list
- Implementation via *heap-ordered* trees, i.e. element in a node is less or equal to all elements in subtrees
- minimum is always in the root

Left-oriented heap — 2

```haskell
data Heap k a = Leaf | Node k a (Heap k a) (Heap k a)
```

```haskell
data Heap k a = Leaf | Node k a (Heap k a) (Heap k a)

makeTree :: (Num k, Ord k) =>
  a -> Heap k a -> Heap k a -> Heap k a
makeTree x a b
  | rank a >= rank b = Node (rank b + 1) x a b
  | otherwise        = Node (rank a + 1) x b a
```

```haskell
data Heap k a = Leaf | Node k a (Heap k a) (Heap k a)

makeTree :: (Num k, Ord k) =>
  a -> Heap k a -> Heap k a -> Heap k a
makeTree x a b
  | rank a >= rank b = Node (rank b + 1) x a b
  | otherwise        = Node (rank a + 1) x b a

merge :: (Num k, Ord k, Ord a) =>
  Heap k a -> Heap k a -> Heap k a -- O(log_2 n)
merge h Leaf = h
merge Leaf h = h
```

Binominal Heap

Def [Binominal Tree] inductive

- rank 0 — singleton node
- rank $r+1$ is *linking* of two binominal trees of rank r such that one if them becomes a left most child of another

Def [Binominal Tree] alternative

Binominal heap of rank r is a node with r descendants $t_1, \ldots, t_r$:
$\forall i \,.\, rank(t_i) = r - i$

- binominal tree of rank r has exactly 2^r elements

TODO: picture 3.3 from page 30

Binominal Tree

```haskell
data Tree a = Node Int a [Tree a]
```

```haskell
link t1@(Node r x1 c1) t2@(Node _ x2 c2)
  | x1 <= x2  = Node (r+1) x1 (t2:c1)
  | otherwise = Node (r+1) x2 (t1:c2)
```

Binominal Heap — 1

```haskell
type Heap a = [Tree a]
```

```haskell
rank (Node r x c) = r
root (Node r x c) = x
insTree t [] = [t]
insTree t ts@(t':ts')
   | rank t < rank t' = t:ts
   | otherwise        = insTree (link t t') ts'

insert x ts = insTree (Node 0 x []) ts
```

Binominal Heap — 1

```haskell
merge (t,  []) = t
merge ([], t ) = t
merge (ts1@(t1:ts1'), ts2@(t2:ts2'))
  | rank t1 < rank t2 = t1 : merge (ts1', ts2)
  | rank t2 < rank t1 = merge (ts1, ts2')
  | otherwise         = insTree (link t1 t2)
                                (merge (ts1', ts2'))
```

```haskell
removeMinTree [t] = (t, [])
removeMinTree (t:ts)
  | root t <= root t' = (t , ts  )
  | otherwise         = (t', t:ts)
 where (t', ts') = removeMinTree ts

findMin ts = root t where (t, _) = removeMinTree ts

deleteMin ts = merge (reverse ts1, ts2) where
  (Node _ x ts1, ts2) = removeMinTree ts
```

RB-Trees — 1

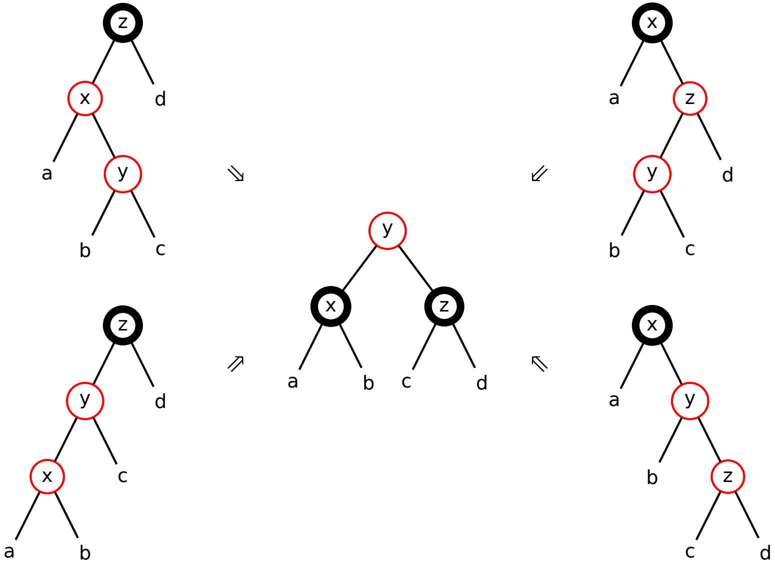
```haskell
data Colour = R | B
data Tree a = E | T Colour (Tree a) a (Tree a)

member x E = False
member x (T _ a y b)
  | x < y     = member x a
  | x > y     = member x b
  | otherwise = True
```

```haskell
insert x s = T B a y b where -- root is always black
  ins E = T R E x E  -- new node is red
  ins s@(T colour a y b)
    | x < y     = balance colour (ins a) y b
    | x > y     = balance colour a       y (ins b)
    | otherwise = s
  T _ a y b = ins s
```

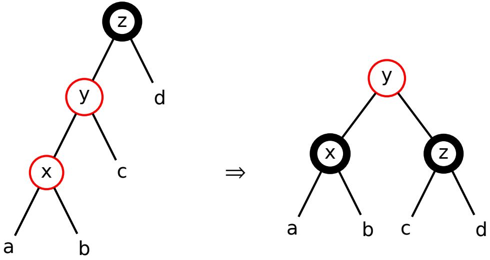
RB-Trees — 2

```
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```


z
x
d
a
y
b
c
x
a
z
y
d
b
c
⇘
⇙
y
x
z
a
b
c
d
⇗
⇖
z
y
d
x
c
a
b
x
a
y
b
z
c
d

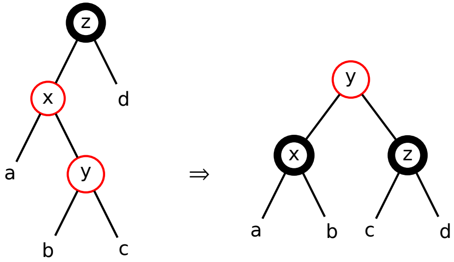
Balance (1/4)

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```



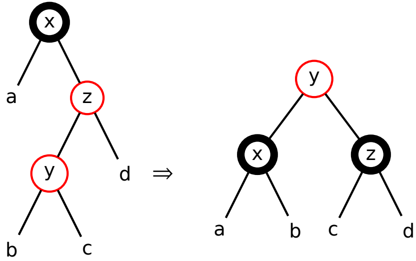
Balance (2/4)

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```



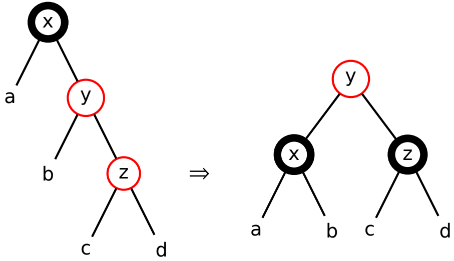
Balance (3/4)

```
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```



Balance (4/4)

```haskell
balance :: Colour -> Tree a -> a -> Tree a -> Tree a
balance B (T R (T R a x b) y c) z d
  = T R (T B a x b) y (T B c z d)
balance B (T R a x (T R b y c)) z d
  = T R (T B a x b) y (T B c z d)
balance B a x (T R (T R b y c) z d)
  = T R (T B a x b) y (T B c z d)
balance B a x (T R b y (T R c z d))
  = T R (T B a x b) y (T B c z d)
balance c t1 a t2 = T c t1 a t2
```



Questions?