Functional Data Structures

"I want to return this wallet. I can't seem to keep any money in it."
Functional Data Structures

imperative
spend money

functional
spend money
undo
A functional data structure preserves old versions after updates.

Old versions are unaffected by updates if they are still accessible (e.g. by variables).
Functional DS: Topics

(1) Functional vs. Imperative Data Structures
   • Persistence
   • Some simple examples: lists, trees
   • Red-black trees

(2) Amortization
   • Banker’s method & physicist’s method
   • Examples: queues

(4) Amortization for Persistent Data Structures
   • Role of lazy evaluation
   • Advanced example: catenable lists
Example: List Append in an Imperative Language

xs = [1, 2, 3]
ys = [7, 8]
zs := xs ++ ys

⇒ last(xs) = 3 (or 8?)
length(xs) = 5

- Dangerous side effects, inconsistencies
- Lists are **not persistent**: old versions are invalidated by updates
+ $O(1)$ time and space requirement
Compare: List Append in Haskell

- No bad side effects
- Lists are persistent: old versions remain valid after updates
- O(|xs|) time and space requirement

xs = [1,2,3]
ys = [7,8]
zs = xs ++ ys
⇒ last(xs) = 3
    length(xs) = 3
Model of Functional Data Structures

**data** \( T = C \ldots | D \ldots \)

\[ x = C (D \ldots) \ldots \]

\( x \) is represented as a pointer data structure (tree/graph) in the heap.

*Update a subterm \( y \):*

\[ \Rightarrow \text{update a copy} \text{ of the corresponding cell } y \text{ in memory (i.e. node in tree)} \]

\[ \Rightarrow \text{copy} \text{ all nodes on the path from the root to } y \]

\[ \Rightarrow \text{the rest of the data structure is shared among the two versions} \]
Example: Binary Search Trees

\[
\begin{aligned}
\text{insert} & :: \text{Ord } a \Rightarrow a \to \text{Tree } a \to \text{Tree } a \\
\text{insert } x \text{ Leaf} & = \text{Node } x \text{ Leaf Leaf} \\
\text{insert } x (\text{Node } y \text{ l } r) \mid x \leq y & = \text{Node } y (\text{insert } x \text{ l}) \text{ r} \\
\mid \text{True} & = \text{Node } y \text{ l} (\text{insert } x \text{ r})
\end{aligned}
\]

\[
t = \text{Node } 4 (\text{Node } 2 ...) (\text{Node } 7 ...)
\]

\[
u = \text{insert } 5 t
\]
On ‘Persistence’

“Traditional” data structures:
Updates destroy old versions of data structures

**Persistent** data structures:
Access to old versions of data structures are possible

**Applications:**
- Editors (undo operations)
- Computational geometry (algorithms)
- Data types in declarative languages

**Degrees of persistence:**

1. **no persistence** (one version)
2. **partial persistence** (updates only on last version)
3. **full persistence** (updates on all versions)
4. **confluent persistence** (…)
5. **controlled persistence** (named choices …)
Red-Black Trees

data Color = R | B

data Tree a = Leaf | Node Color a (Tree a) (Tree a)

Invariants:

1. No red node has a red child
2. All paths (root→leaf) contain the same number of black nodes

⇒ longest path (alternating black and red nodes) is at most twice as long as the shortest path (black nodes only)

⇒ depth of a red-black tree with n nodes is \( \leq 2(\log (n+1)) \)

Worst Case:

(a) \( n+1 \geq 2^d \) \( \Rightarrow \) \( d \leq \log (n+1) \)
(b) depth \( \leq 2d \)

⇒ depth \( \leq 2(\log (n+1)) \)
Insertion into Red-Black Tree

data Color = R | B
data Tree a = Leaf | Node Color a (Tree a) (Tree a)

insert :: Ord a => a -> Tree a -> Tree a
insert x = blackRoot . ins

  where ins Leaf = Node R x Leaf Leaf
       ins (Node c y l r) | x<y = balance c y (ins l) r
                       | True = balance c y l (ins r)

blackRoot (Node _ x l r) = Node B x l r

balance B z (Node R y (Node R x a b) c) d = Node R y (Node B x a b) (Node B z c d)
balance B z (Node R x a (Node R y b c)) d = Node R y (Node B x a b) (Node B z c d)
balance B x a (Node R z (Node R y b c) d) = Node R y (Node B x a b) (Node B z c d)
balance B x a (Node R y b (Node R z c d)) = Node R y (Node B x a b) (Node B z c d)
balance c x l r = Node c x l r
Balancing Trees

1. l l
2. l r
3. r l
4. r r
Balancing Trees (cont'd)

Node B z
(Node R y
(Node R x a b)
c))

d

Node B z
(Node R x
 a
(Node R y b c))

d

Node R y
(Node B x a b)
(Node B z c d)

Node B x
(Node R z
(Node R y b c)
d)

Node B x
(Node R y
 b
(Node R z c d))

Functional Data Structures
Amortized Worst Case

‘worst’ worst case:
always assume maximal time

$$n \cdot O(n) \rightarrow O(n^2)$$

amortized worst case:
$$c \cdot O(n) + n \cdot O(1) \rightarrow O(n)$$

Advantages of amortized analysis:
• more realistic analyses
• new design opportunities
• easier implementations
• faster data structures in practice

Self-adjusting DS (splay trees)
Example: Implementing a Queue with 2 Stacks

Naive queue implementation based on lists

\[
\begin{align*}
\text{head} \ (x:xs) & = x \\
\text{tail} \ (x:xs) & = xs \\
\text{snoc} \ [\ ] \ y & = [y] \\
\text{snoc} \ (x:xs) \ y & = x:\text{snoc} \ xs \ y
\end{align*}
\]

\(O(n)\)

Idea:

Represent a queue \(q\) by a pair of lists \((f,r)\) such that

\[q = f ++ \text{reverse} \ r\]

\[
\begin{array}{c}
q \\
[1,2,3,4] \\
\downarrow \\
\text{snoc} \ q \ 5 \\
[1,2,3,4,5] \\
\downarrow \\
[1,2,5,4,3]
\end{array}
\]

\[
\begin{align*}
\text{snoc} \ (f,r) \ x & = \text{chk} \ (f,x:r) \\
\text{tail} \ (x:f,r) & = \text{chk} \ (f,r) \\
\text{chk} \ ([],r) & = (\text{reverse} \ r,[]) \\
\text{chk} \ (f,r) & = (f,r)
\end{align*}
\]

\(O(n)\) ‘sometimes’
The Banker’s Method

**Idea:** For each operation $i$, define amortized costs:

- $a_i$: amortized costs
- $t_i$: actual costs

Each operation $i$ gets $a_i$ credits.

Operation $i$ is:

- **cheap** ⇔ $t_i < a_i$
- **neutral** ⇔ $t_i = a_i$
- **expensive** ⇔ $t_i > a_i$

Cheap operations save $a_i - t_i$ credits. These are associated with a location in the data structure.

Neutral operations spend their own credits.

Expensive operations require in addition to their own credits $t_i - a_i$ credits that were saved by cheap operations.

**Show:** all available credits suffice for all operations:

$$\sum a_i \geq \sum t_i$$

In other words: there are enough credits to pay all actual costs.
Banker’s Analysis of Queue Implementation

1 credit \hspace{1cm} \text{head} \ (x:f,r) = x
1 credit \hspace{1cm} \text{tail} \ (x:f,r) = \text{chk} \ (f,r)
2 credits \hspace{1cm} \text{snoc} \ (f,r) \ x = \text{chk} \ (f,x:r)

\text{amortized costs} \ a_i

\text{snoc} \text{ saves 1 credit. } \\
\text{This credit is associated with each element } x \text{ added to } r.

\text{head and tail (for } |f| > 1 \text{) need only 1 credit.}

\text{tail (for } |f| = 1 \text{) requires } |r|+1 \text{ credits.}

\text{tail can use its own credit and the } |r| \text{ credits associated with the elements of } r \text{ that were saved by } \text{snoc}.

\text{Therefore: } \sum a_i \geq \sum t_i

\Rightarrow \text{tail runs in } O(1) \text{ amortized worst case time.}
The Physicist’s Method

Idea: Define potential for data structure \( \Phi: D \rightarrow \mathbb{R} \)

Operation \( i \) maps \( d_{i-1} \) to \( d_i \), then:

\[
a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})
\]

\[
\sum t_i = \sum (a_i + \Phi(d_i) - \Phi(d_{i-1}))
= \sum a_i + \Phi(d_0) - \Phi(d_n)
\]

Show: Potential is always positive.
That is, with \( \Phi(d_0) = 0 \), \( \forall i: \Phi(d_i) \geq 0 \).

Then amortized costs are an upper bound on the accumulated actual costs:

\[
\sum a_i \geq \sum t_i
\]
Physicist’s Analysis of Queue Implementation

Define $\Phi(f,r) = |r|$

Thus:

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

$$= t_i + |r_i| - |r_{i-1}|$$

$$\Delta \Phi$$

- **snoc** increases $|r|$ by 1
- **snoc** increases potential by 1

$$[\Delta \Phi = 1 \Rightarrow a_i = 2]$$

- **head** and **tail** (for $|f|>1$) do not change $r$
- **head** and **tail** do not change the potential

$$[\Delta \Phi = 0 \Rightarrow a_i = 1]$$

- **tail** (for $|f_{i-1}|=1$) reduces $|r|$ by $|r|$
- **tail** decreases potential by $|r|$

$$t_i = |r_{i-1}| + 1$$

$$[\Delta \Phi = -|r_{i-1}| \Rightarrow a_i = t_i + \Delta \Phi$$

$$a_i = |r_{i-1}| + 1 - |r_{i-1}| = 1]$$

Therefore: $\Phi$ is always positive and thus $\sum a_i \geq \sum t_i$

$\Rightarrow$ **tail** runs in $O(1)$ amortized worst-case time.
O(1) Amortized Deques

**Idea:**
Treat \( f \) and \( r \) in a symmetric way: keep \( f \) and \( r \) non-empty if queue contains 2 or more elements.

\[
\begin{align*}
\text{head} \ (x:f,r) &= x \\
\text{head} \ (_,[x]) &= x \\
\text{tail} \ (x:f,r) &= \text{chk} \ (f,r) \\
\text{snoc} \ (f,r) x &= \text{chk} \ (f,x:r) \\
\text{last} \ (f,x:r) &= x \\
\text{last} \ ([x],_) &= x \\
\text{init} \ (f,x:r) &= \text{chk} \ (f,r) \\
\text{cons} \ x \ (f,r) &= \text{chk} \ (x:f,r)
\end{align*}
\]

\[
\begin{align*}
\text{chk} \ ([],r@(_,_::_)) &= \text{split reverse id} \ r \\
\text{chk} \ (f@(_,_:_),[]) &= \text{split id reverse f} \\
\text{chk} \ (f,r) &= (f,r)
\end{align*}
\]

\[
\text{split} \ f \ g \ l = (f \ xs,g \ ys) \\
\text{where} \ (xs,ys) = (\text{take} \ k \ l,\text{drop} \ k \ l) \\
k &= \text{length} \ l \div 2
\]

When one list becomes empty, split the other list in half and reverse one part.
Physicist’s Analysis of Deque Implementation

Define $\Phi(f,r) = \text{abs} (|f|-|r|)$

Thus:

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

$$= t_i + \text{"change in length diff"}$$

Define $\Phi(f,r) = \text{abs} (|f|-|r|)$

Thus:

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

$\Phi$ is always positive and thus $\sum a_i \geq \sum t_i$

$\Rightarrow$ all operations run in $O(1)$ amortized worst case time.

- **snoc** and **cons** increase $\Phi$ by 1
  $\Rightarrow a_i = 2$

- **tail** ($|f|>1$), and **init** ($|r|>1$) increase $\Phi$ by 1
  $\Rightarrow a_i = 2$

- **head** and **last** don't change $\Phi$
  $\Rightarrow a_i = 1$

- **tail** (for $|f|=1$): $t_i = |r|$, $\Delta \Phi = -|r|$
  $\Rightarrow a_i = |r|+1-|r| = 1$

- **init** (for $|r|=1$): $t_i = |f|$, $\Delta \Phi = -|f|$
  $\Rightarrow a_i = |f|+1-|f| = 1$
Amortization & Persistent Data Structures

The Bad News!

(Traditional) amortization breaks down in the case of persistent data structures because an expensive operation can be repeated many times.

Example:

Let \( q = ([x],r) \) where \(|r| = n\).

For \( 1 \leq i \leq n \) compute: \( q_i = \text{tail } q \) \( \{ O(n^2) \} \)

\( \Rightarrow \) (a new) computation of \text{reverse} is forced \( n \) times.
The \( n \) credits in \( r \) do not suffice to amortize all costs.

Problem: Credits can be spent only once.

Solution: Memoization & early, incremental computation

Employ lazy evaluation
Lazy vs. Eager Evaluation

cf. CS 581

eager evaluation

\[
e_1 \downarrow \lambda v. e' \quad e_2 \downarrow l' \quad [v \mapsto l']e' \downarrow l
\]

\[
e_1 \quad e_2 \downarrow l
\]

normal order evaluation

\[
e_1 \downarrow \lambda v. e' \quad [v \mapsto e_2]e' \downarrow l
\]

\[
e_1 \quad e_2 \downarrow l
\]

... add memoization to get lazy evaluation
O(1) Queues under Persistence

Idea:
Start computation of reverse early enough so that each operation sequence that forces reverse has paid for it completely.

Evaluation of reverse needs n steps, but will happen only after m applications of tail.

type Queue a = (Int,[a],Int,[a])

head (m,x:f,n,r) = x

tail (m,x:f,n,r) = chk (m-1,f,n,r)

snoc (m,f,n,r) x = chk (m,f,n+1,x:r)

chk q@(m,f,n,r) | m>=n = q

| True = (m+n,f++reverse r,0,[])

lengths of f and r
Amortization under Persistence

Compute:

\[ q_i = \text{tail } q_{i-1} \]

(for \(0 < i < m + 2\))

\[
q = (m, [x_1, ..., x_m], m, [y_1, ..., y_m])
\]

\[
q_1 = \text{chk} (m-1, [x_2, ..., x_m], m, [y_1, ..., y_m])
\]

\[
= (m-1 + m, [x_2, ..., x_m] + \text{reverse } [y_1, ..., y_m], 0, [])
\]

\[
q_2 = (2m-2, [x_3, ..., x_m] + \text{reverse } [y_1, ..., y_m], 0, [])
\]

... \[
q_m = (m, [], + \text{reverse } [y_1, ..., y_m], 0, [])
\]

\[
q_{m+1} = (m-1, [y_{m-1}, ..., y_1], 0, [])
\]
Elements in Functional DS

(1) **Exploit Amortization**
   ⇒ more flexibility in defining operations
   (Banker's method & physicist's method)

(2) **Employ Lazy Evaluation**
   ⇒ make amortization work in the presence of persistent data structures

(3) **Apply 'Scheduling'**
   ⇒ systematically transform amortized data structures into worst-case data structures

(4) **Special Techniques**
   ⇒ DS bootstrapping, structural decomposition, ...
Catenable Lists

Catenable lists extend the usual lists with an efficient append function \texttt{cat}.

\[
\begin{align*}
\text{empty} & \colon \text{Cat} \ a \\
\text{head} & \colon \text{Cat} \ a \rightarrow a \\
\text{tail} & \colon \text{Cat} \ a \rightarrow \text{Cat} \ a \\
\text{cons} & \colon a \rightarrow \text{Cat} \ a \rightarrow \text{Cat} \ a \\
\text{snoc} & \colon \text{Cat} \ a \rightarrow a \rightarrow \text{Cat} \ a \\
\text{cat} & \colon \text{Cat} \ a \rightarrow \text{Cat} \ a \rightarrow \text{Cat} \ a
\end{align*}
\]

\[
\begin{array}{l}
\text{just like } []
\end{array}
\]

\[
\begin{array}{l}
\text{O(1) version of } (++)
\end{array}
\]

\textbf{snoc} can be simulated by \texttt{cons} and \texttt{cat}:

\[
\text{snoc } c \ x = \text{cat } c \ (\text{cons } x \ \text{empty})
\]
DS Bootstrapping

(1) Structural Abstraction:

Regard a collection as a single object
⇒ cat can be reduced to snoc

(2) Structural Decomposition:

Repeated decomposition into smaller parts
⇒ implementation can be reduced to trivial cases

```haskell
data Cat a = E | C a (Queue (Cat a))
```

```haskell
data Queue a = Q ([a],[a])
```

- Catenable list \( l \) is represented as a tree \( t \), such that \( l = \text{preorder} \ t \)
- Children of a node are stored in a queue (with \( O(1) \) operations)
CatList Operations

```haskell
empty :: Cat a
empty = E

head :: Cat a -> a
head (C x _) = x

cat :: Cat a -> Cat a -> Cat a
cat c E = c
cat E c = c
cat c d = link c d

link (C x q) d = C x (Q.snoc q d)

cons :: a -> Cat a -> Cat a
cons x c = cat (C x Q.empty) c

snoc :: Cat a -> a -> Cat a
snoc c x = cat c (C x Q.empty)

import qualified Queue as Q

data Cat a = E | C a (Q.Queue (Cat a))
```

Diagram:
```
\begin{tikzpicture}
  \node (x) at (0,0) {x};
  \node (d) at (1,0) {d};
  \node (c) at (2,0) {c};
  \draw[->] (x) -- (d);
  \draw[->] (x) -- (c);
\end{tikzpicture}
```
**CatList: Tail**

\[ \text{tail} :: \text{Cat} \ a \rightarrow \text{Cat} \ a \]
\[ \text{tail} \ (C \ x \ q) = \text{linkAll} \ q \]

\[
\text{linkAll} \ q \mid \text{Q.isEmpty} \ r = t \\
\mid \text{otherwise} \quad = \text{link} \ t \ (\text{linkAll} \ r) \]

\[ \text{where} \ (t,r) = (\text{Q.head} \ q, \text{Q.tail} \ q) \]

Diagram:
- **q** = \{ a, b, c, d \}
- **r** = Q.tail q
- **t** = Q.head q
- **tail** (C x q) = tail (C x q) -> tail (link (C x q) d)

Graphical representation of the tail function and its application to a CatList.
The Study of Data Structures...

Features

Efficiency
The Study of Data Structures ...

... is enriched by using (concepts of) functional languages