Functional Data Structures


Purely functional

(car | cdr)

never modify
only create new pairs
only DAGs

(Atomic values: Integers, Chars, Float, Bool, ....)

**Strict evaluation**
Evaluate list now

**Lazy evaluation/memoization**
First add element when head needed and return function
*lazy* incrementing the rest

Example

\[
\text{inc}(() ) = () \\
\text{inc}(e::L') = (e+1)::\text{inc}(L')
\]
List operations

- `makelist(x)`
- `push(x,L)`
- `pop(L)`
- `inject(x,L)`
- `eject(L)`
- `catenate(K,L)`

Diagram:

```
x_1 -> x_2 -> x_3 -> x_4 -> x_5
  ^  |  |  |  |
  push head pop  inject eject
```

Categories:
- Stack
- Queue
- Dequeue
- Catenable lists
- Catenable deques
Catenable lists (slow)

\[
\begin{align*}
\text{cat}((), L) &= L \\
\text{cat}(e::K, L) &= e::\text{cat}(K, L)
\end{align*}
\]

O(length 1st list)

List reversal

\[
\begin{align*}
\text{rev}(L) &= \text{rev}'(L, ()) \\
\text{rev}'(() , T) &= T \\
\text{rev}'(e::L, T) &= \text{rev}'(L, e::T)
\end{align*}
\]

O(|L|)

Bad if expensive operation repeated

Queues (Head, Tail)

Ex: ((1,2,3),(5,4)) \equiv [1,2,3,4,5]

\[
\begin{align*}
\text{infect}(e, (H, T)) &= (H, e::T)
\end{align*}
\]

O(1)

Version 1

\[
\begin{align*}
\text{pop}((e::H, T)) &= (e, (H, T)) \\
\text{pop}(() , T) &= (e, (T',())), \text{ where } e::T' = \text{rev}(T)
\end{align*}
\]

O(1) amortized

Strict

\(\Phi = |T|\)

Version 2 (Invariant |H|\(\geq|T|\))

\[
\begin{align*}
\text{pop}((e::H, T)) &= (e, (H, T)) \quad \text{if } |H|>|T| \\
&= (e, (\text{cat}(H, \text{rev}(T)), ())) \quad \text{if } |H|\leq|T|
\end{align*}
\]

\[
\begin{align*}
\text{infect}(e, (H, T)) &= (H, e::T) \quad \text{if } |T|>|H| \quad \text{if } |H|>|T| \\
&= (\text{cat}(H, \text{rev}(e::T)), ()) \quad \text{if } |H|\leq|T|
\end{align*}
\]

O(1) amortized

Lazy

Good

\[
\begin{align*}
\text{cat}((), L) & = L \quad \text{lazy evaluation} \rightarrow \text{recursive call first} \\
\text{cat}(e::K, L) & = e::\text{cat}(K, L) \\
\text{rev}(L) & = \text{rev'}(L, ()) \quad \text{lazy evaluation} \rightarrow \text{everything} \\
\text{rev'}((), T) & = T \quad \text{evaluated when 1st element accessed} \\
\text{rev'}(e::L, T) & = \text{rev'}(L, e::T) \\
\text{inject}(e, (H, T)) & = (H, e::T) \\
\end{align*}
\]

**TRICK** In \(\text{cat}(H, \text{rev}(T))\) the cost for \(\text{rev}(T)\) is paid by the subsequent pops (with no reversals) from the \(H\) part of the catenation. All pops deleting from \(H\) pays \(O(1)\) for doing \(O(1)\) work of the reverse.

**Version 2 (Invariant \(|H| \geq |T|\))**

\[
\begin{align*}
pop\left((e::H, T)\right) & = (e, (H, T)) \quad \text{if } |H| > |T| \\
& = (e, (\text{cat}(H, \text{rev}(T)), ()) \quad \text{if } |H| \leq |T|
\end{align*}
\]

setup expensive lazy evaluation

cheap operations
amortize cost of upcoming expensive lazy evaluation

execute expensive

only one of $Q_5$ or $Q_5$ is expensive
Real-time Queues i.e. strict worst-case $O(1)$ time


- incremental version of the amortized solution

$|E| + d \leq |D| + |A|/2$

$|F| \geq (2|B| + |A| + |C|)/3$

- deques...
Queues


Catenable lists


Functional Concatenable Search Trees


- Search, update $O(\log n)$
- Catenation $O(1)$

Open problems
- Split $O(\log n)$ ?
- Finger search trees with $O(1)$ time catenation ?
- Search trees with $O(1)$ space per update ?