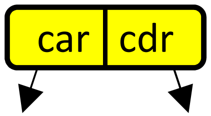
Functional Data Structures

[C. Okasaki, *Simple and efficient purely functional queues and deques*, J. of Functional Programming, 5(4), 583-592, 1995]
[H. Kaplan, R. Tarjan, *Purely functional, real-time deques with catenation*, Journal of the ACM, 46(5), 577-603, 1999]

Purely functional



never modify
only create new pairs
only DAGs

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

Strict evaluation
Evaluate list now

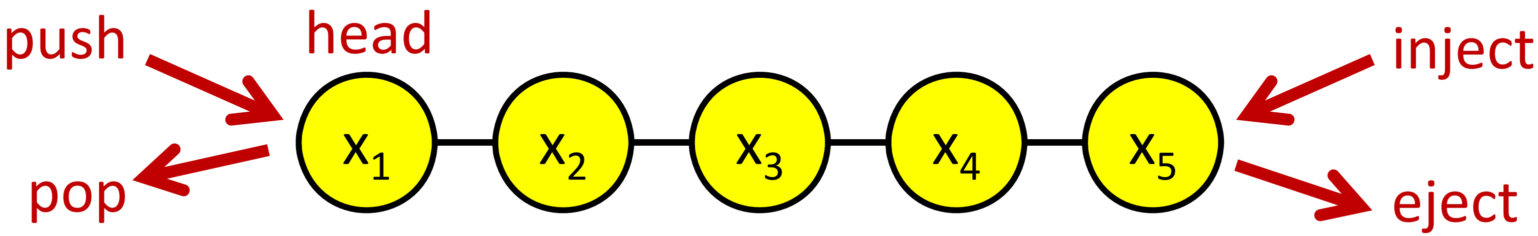
Lazy evaluation/memoization
First add element when head needed and return function *lazy* incrementing the rest

Example

```
inc(())     = ()
inc(e::L’) = (e+1)::inc(L’)
```

List operations

	Stack	Queue	Deque	Catenable lists	Catenable deques
makelist(x)	★	★	★	★	★
push(x,L)	★		★	★	★
pop(L)	★	★	★	★	★
inject(x,L)		★	★		★
eject(L)			★		★
catenate(K,L)				★	★



Catenable lists (slow)

```
cat((),L)      = L
cat(e::K,L)    = e::cat(K,L)
```

O(length 1st list)

List reversal

```
rev(L)         = rev'(L,())
rev'((),T)     = T
rev'(e::L,T)   = rev'(L,e::T)
```

O(|L|)

Bad if expensive operation repeated

Queues (Head,Tail) Ex: ((1,2,3),(5,4)) ≡ [1,2,3,4,5]

```
inject(e,(H,T)) = (H,e::T)
```

O(1)

Version 1

```
pop((e::H,T)) = (e,(H,T))
pop(((),T))   = (e,(T',())) where e::T' = rev(T)
```

Strict
O(1) amortized
$\Phi = |T|$

Version 2 (Invariant |H|≥|T|)

```
pop((e::H,T)) = (e,(H,T))                 if |H|>|T|
              = (e,(cat(H,rev(T)),()) if |H|≤|T|
Inject(e,(H,T)) = (H,e::T) if |T|>|H|   if |H|>|T|
                = (cat(H,rev(e::T)),()) if |H|≤|T|
```

Lazy **Good**
O(1) amortized

[C. Okasaki, *Simple and efficient purely functional queues and deques*, J. of Functional Programming, 5(4), 583-592, 1995]

```
cat((),L)      = L
cat(e::K,L)    = e::cat(K,L)
```

lazy evaluation → recursive call first evaluated when 1st element accessed

```
rev(L)         = rev'(L,())
rev'((),T)     = T
rev'(e::L,T)   = rev'(L,e::T)
```

lazy evaluation → everything evaluated when 1st element accessed

```
inject(e,(H,T)) = (H,e::T)
```

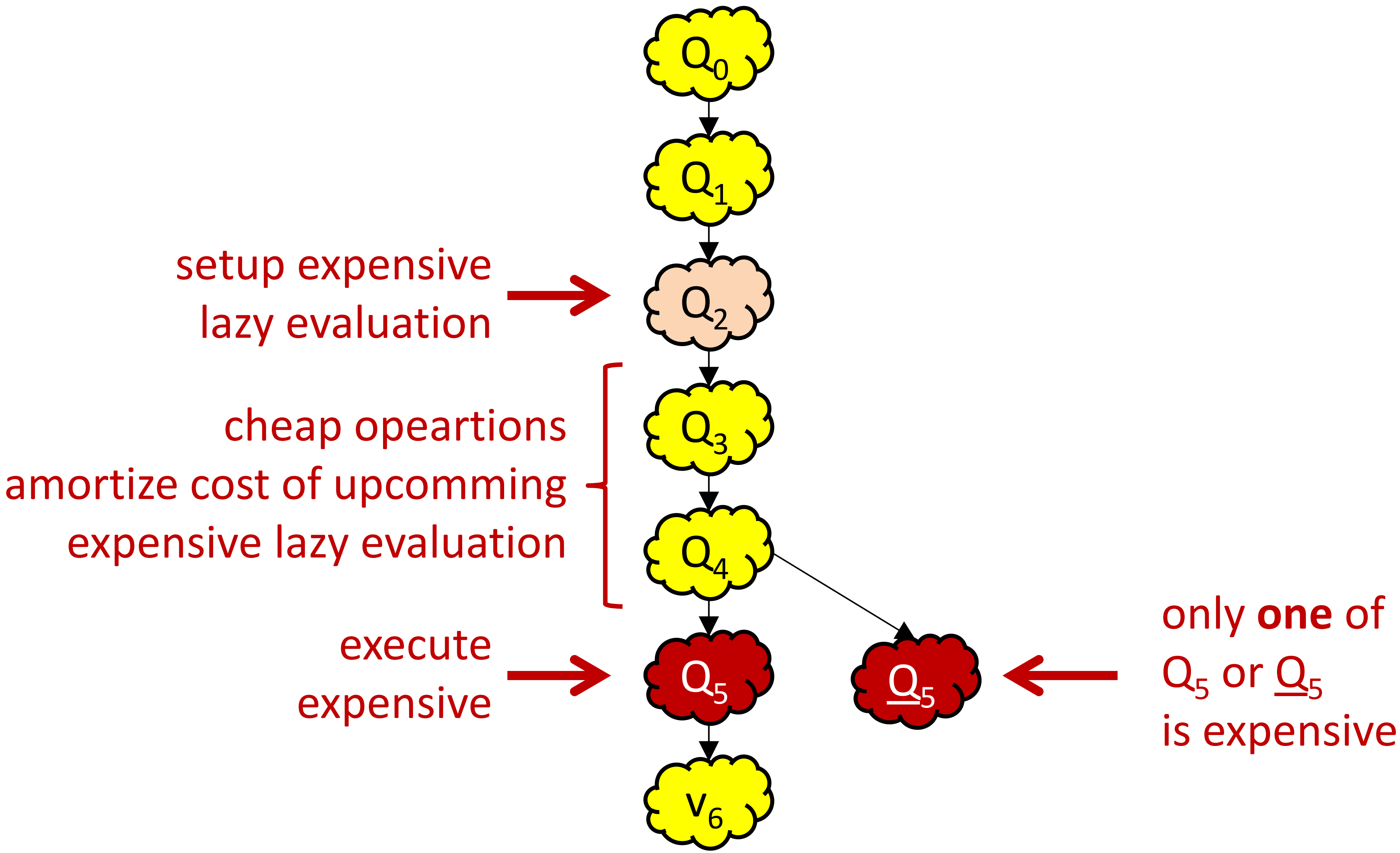
> **TRICK** In `cat(H,rev(T))` the cost for rev (T) is paied by the subsequent pops (with no reversals) from the H part of the catenation. All pops deleting from H pays O(1) for doing O(1) work of the reverse.

Version 2 (Invariant |H|≥|T|)

lazy evaluation

```
pop((e::H,T)) = (e,(H,T))               if |H|>|T|
              = (e,(cat(H,rev(T)),()) if |H|≤|T|
```

[C. Okasaki, *Simple and efficient purely functional queues and deques*, J. of Functional Programming, 5(4), 583-592, 1995]

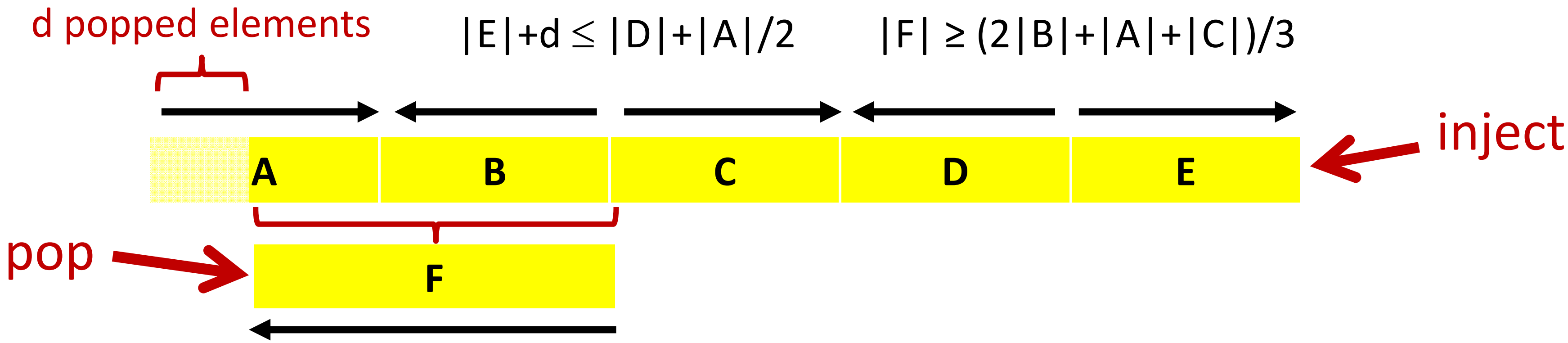


Q_0
Q_1
setup expensive lazy evaluation
Q_2
cheap opeartions amortize cost of upcomming expensive lazy evaluation
Q_3
Q_4
execute expensive
Q_5
$\underline{Q}_5$
only **one** of Q_5 or $\underline{Q}_5$ is expensive
V_6

Real-time Queues i.e. strict worst-case O(1) time

[R. Hood, R. Melville, *Real-time queue operations in pure Lisp*. Information Processing Letters, 13, 50-54, 1981]

- incremental version of the amortized solution




```
                                  d   F   A   B   C   D   E
makelist(x)                     = (0,(x),(),(x),(),(),())
inject(x,(d,F,A,B,C,D,E)) = f(f(f(d,F,A,B,C,D,x::E)))
pop((d,x::F,A,B,C,D,E))   = f(f(f(f(d+1,F,A,B,C,D,E))))

f(d,F,(),B,x::C,D,E)  = (d,F,(),B,C,x::D,E)
f(d,F,A,x::B,(),D,E)  = (d,F,x::A,B,(),D,E)
f(d,F,x::A,(),(),D,E) = (d,F,A,(),(),x::D,E) if |A|>d
f(d,F,A,(),(),D,E)    = (0,D,(),D,E,(),())   if |A|=d
```

- deques...

Queues

[R. Hood, R. Melville, *Real-time queue operations in pure Lisp*. Inf.ormation Processing Letters, 13, 50-54, 1981]

Strict, worst-case O(1)

[C. Okasaski, *Simple and efficient purely functional queues and deques*. Journal of Functional Programming 5,4, 583-592, 1995]

Lazy, amortized O(1)

Catenable lists

[S.R. Kosaraju, *Real-time simulation of concatenable double-ended queues by double-ended queues*, Proc. 11th Annual ACM Symposium on Theory of Computing, 346-351, 1979]

[S.R. Kosaraju, *An optimal RAM implementation of catenable min double-ended queues*, Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 195-203, 1994]

Not confluently persistent

[J.R. Driscoll , D.D.K. Sleator , R.E. Tarjan, *Fully persistent lists with catenation*, Journal of the ACM, 41(5), 943-959, 1994]

O(loglog k)

[A.L. Buchsbaum , R.E. Tarjan, *Confluently persistent deques via data-structural bootstrapping*, Journal of Algorithms, 18(3), 513-547, 1995]

$2^{O(\log^* k)}$

O(log* k)

Not funtional

[H. Kaplan, R. Tarjan, *Purely functional, real-time deques with catenation*, Journal of the ACM, 46(5), 577-603, 1999]

Strict, worst-case O(1)

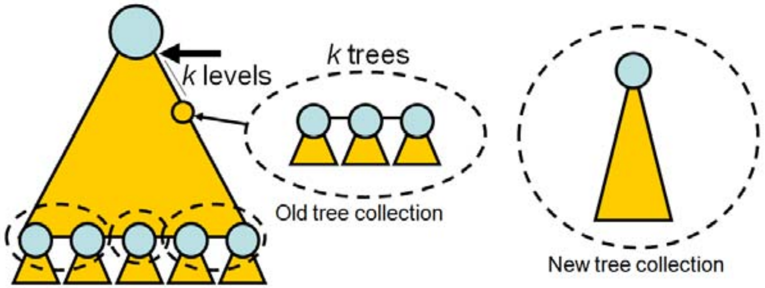
[H. Kaplan, C. Okasaki, R.E. Tarjan, *Simple Confluently Persistent Catenable Lists*, SIAM Journal of Computing 30(3), 965-977 (2000)]

Lazy, amortized O(1)

Functional Concatenable Search Trees

[G.S. Brodal, C.Makris, K. Tsichlas, *Purely Functional Worst Case Constant Time Catenable Sorted Lists*, In Proc. 14th Annual European Symposium on Algorithms, LNCS 4168, 172-183, 2006]

- 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 ?