Advanced FP

Functional Data Structures 2

Daniil Berezun

2023

Plan for Today: More Purely Functional Data Structures

- Numerical Representation
  - Binary (Dense and Sparse)
    - Binary Random-Access Lists
    - Binary Heaps
  - Segmented
  - Skew Binary
    - Skew Binary Random-Access Lists
    - Skew Binary Heaps

- Data-Structural Bootstrapping
  - Structural Decomposition
    - Bootstrapped Queues
  - Structural Abstraction
    - Heaps with efficient merging

```
datatype a List = Nil
  | Cons of a × a List
fun tail (Cons (x,xs)) = xs
fun app (Nil, ys) = ys
    | (Cons (x,xs), ys) =
       Cons (x, app(xs,ys))
```

```
datatype Nat = Zero
  | Succ of Nat
fun pred (Succ  n) = n
fun plus (Zero, n) = n
       | (Succ m, n) =
         Succ (plus(m,n))
```

- Unary — inc $O(1)$, add $O(n)$
- Binary — inc $O(\log n)$, add $O(\log n)$

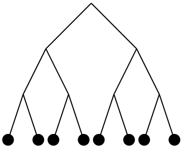
```
(* Dense *)
datatype Digit = Z | O
type Nat = Digit list

fun inc [] = [O]
| Z::ds = O::ds
| O::ds = Z::inc ds (* carry *)
fun dec [O] = []
| O::ds = Z::ds
| Z::ds = O::dec ds (* borrow *)
fun add (ds , []) = ds
| ([], ds) = ds
| (d::ds1, Z::ds2) = d::add(ds1,ds2)
| (Z::ds1, d::ds2) = d::add(ds1,ds2)
| (O::ds1, O::ds2) =
  Z::inc(add(ds1,ds2)) (* carry *)
```

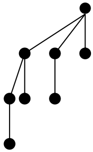
```
(* Sparse *)
type Nat = int list (*weights,2^i*)

fun carry (w, []) = [w]
| (w, ws as w'::r) = if w < w'
  then w::ws
  else carry(2*w,r)
fun borrow (w, ws as w'::r) =
  if w = w' then r
  else w:borrow(2*w,ws)
fun inc ws = carry  (1, ws)
fun dec ws = borrow (1, ws)
fun add (ws,[]) = ws
| ([],ws) = ws
| (m as w::ws, n as q::qs) =
  if   w < q then w::add(ws,n)
  elif q < w then q::add(n,qs)
  else carry(2*w,add(ws,qs))
```

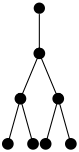
Reminder: Binary Representations



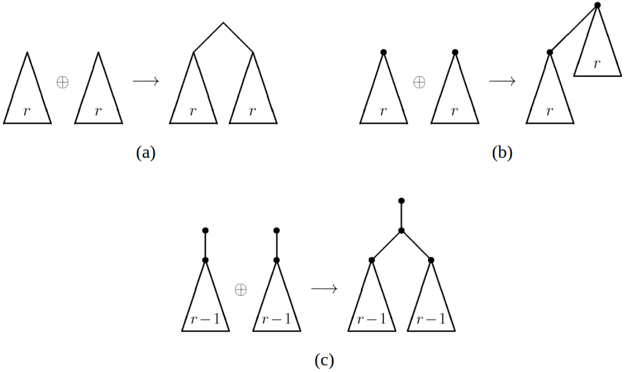
Complete binary leaf tree



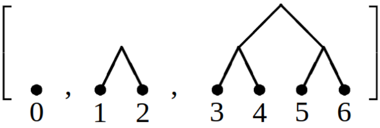
Binomial tree



Pennant



Linking trees




```
(* dense *)                    (* tree size and two kids *)
datatype a T      = L of a | N of int × a T × a T
datatype a D      = Z | O of a Tree
type     a RList = a D list
```

```
(* Dense num inc *)

fun inc [] = [O]
| (Z::ds) = O::ds
| (O::ds) = Z:: inc ds
```

```
(* cons in RA Lists *)
fun cons (x, ts) = insT (L x, ts)
fun insT (t, []) = [O t]
| (t , Z::ts) = O t ::ts
| (t1, O t2 :: ts) =
  Z :: insT (link (t1, t2), ts)
```

Binary RA Lists --- 2

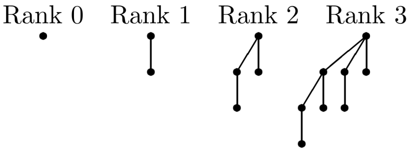
```
(* Dense num dec *)
fun dec [O] = []
| (O::ds) = Z:: ds
| (Z::ds) = O:: dec ds
```

```
(* RA List del (bt = borrowTree) *)
fun bT [O t ] = (t, [])
| (O t :: ts) = (t, Z::ts)
| (Z   :: ts) =
  let (N(_,t1,t2),ts') = bT ts
  in (t1, O t2 :: ts')
fun head = fst . bT
fun tail = snd . bT
```

```
fun lookup
  (Z ::ts, i)   = lookup (ts, i)
| (O t ::ts, i) =
  if   i < size t
  then lookupTree (t, i)
  else lookup (ts, i - size t)
fun lookupTree (L x , 0) = x
| (N (w, t1, t2), i) =
  if   i < w div 2
  then lookupTree (t1, i)
  else lookupTree (t2, i - w/2)
```

```
fun update
  (Z   :: ts, i, y) = Z:: update(ts,i,y)
| (O t :: ts, i, y) =
  if   i < size t
  then O (updateTree (t, i, y)) :: ts
  else O t :: update(ts, i- size t, y)
fun updateTree (L x, 0, y) = L y
| updateTree (N (w,t1,t2), i, y) =
  if   i < w/2
  then N (w, updateTree(t1, i, y), t2)
  else N (w, t1, updateTree(t2,i-w/2,y))
```

- cons, head, tail, lookup, update --- $O(\log n)$




```
(* sparse *)
datatype Tree = N of int × Elem.T × Tree list
type     Heap = Tree list
```

```
(* assert: r2 == r *)
fun link (t1 as N(r,x1,c1), t2 as N(r2,x2,c2)) =
  if   Elem.leq (x1, x2)
  then N (r+1, x1, t2 :: c1)
  else N (r+1, x2, t1 :: c2)
```

Binomial Heaps (Queues) --- 2

```
(* sparse num  inc *)
fun carry (w, []) = [w]
| (w, ws as w'::r) =
    if w < w'
    then w :: ws
    else carry (2 * w, r)
fun inc ws = carry (1, ws)
```

```
(* BH insert *)
fun insT (t , []) = [t]
| (t1, ts as t2 :: rs) =
  if rank t1 < rank t2
  then t1 :: ts
  else insT (link (t1, t2), rs)
fun ins (x,ts) = insT (N(0,x,[]), ts)
```

```
(* sparse num add *)
fun add
  (ws,[]) = ws
| ([],ws) = ws
| add (m as w::ws, n as q::qs) =
  if   w < q then w :: add(ws,n)
  elif q < w then q :: add(n,qs)
  else carry (2*w, add (ws, qs))
```

```
(* merge BHs *)
fun merge
  (ts1, [] ) = ts1
| ([] , ts2) = ts2
| (t1::ts1, t2::ts2) =
  if   w < q then t1 ::merge(ts1,t2::ts2)
  elif q < w then t2 ::merge(t1::ts1,ts2)
  else insT (link(t1,t2), merge(ts1,ts2))
  where w = rank t1, q rank t2
```

```
fun findMin
  [t] = root t
| (t::ts) =
  let val x = root t
      val y = findMin ts
  in if Elem.leq (x, y)
     then x else y
```

```
fun deleteMin ts =
  let fun getMin [t] = (t, [])
        | getMin (t :: ts) =
          let (t', ts') = getMin ts
          in if Elem.leq (root t, root t')
             then (t , ts)
             else (t', t :: ts')
      (Node (_,x,ts1), ts2) = getMin ts
  in merge (rev ts1, ts2)
```

```
(* Segmented Numbers;
   last block (if any) always contains 1s *)
datatype DigitBlock = Z of int | O of int
type     Nat        = DigitBlock list

fun zeros (i, []) = []                          (* O(1) *)
| (i , Z j :: bs) = Z (i+j) :: bs
| (0, bs) = bs
| (i, bs) = Z i :: bs
fun ones (i , O j :: bs) = O (i+j) :: bs (* O(1) *)
| (0, bs) = bs
| (i, bs) = O i :: bs

fun inc [] = [O 1]                              (* O(1) *)
| (Z i :: bs) = ones (1, zeros (i-1, bs))
| (O i :: bs) = Z i :: inc bs
fun dec (O i :: bs) = zeros (1, ones (i-1, bs))
| (Z i :: bs) = O i :: dec bs                   (* O(1) *)
```

```
(* sparse; weights in increasing order *)
type Nat = int list

fun inc (ws as a::b::rs) =  (* O(1) *)
  if a = b then (1+a+b)::rs else 1::ws
| ws = 1 :: ws

fun dec (1::ws) = ws        (* O(1) *)
| (w::ws) = (w div 2)::(w div 2)::ws
```

Skew Binary Random-Access Lists

```
datatype a Tree  = L of a | N of a × a Tree × a Tree
type     a RList = (int × a Tree) list
```

```
(* skew numbers inc *)
fun inc (ws as a::b::rs) =
  if a = b
  then (1+a+b)::rs
  else 1::ws
| ws = 1 :: ws
```

```
(* skew bin num dec *)
fun dec (1::ws) = ws
| (w::ws) = (w/2)::(w/2)::ws
```

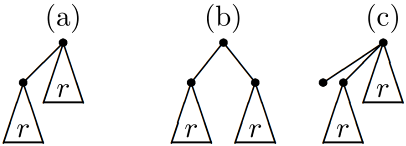
```
(* tail of Skew Binary RA Lists *)
fun tail    ((1, L _) :: ts) = ts
| ((w, N (x, t1, t2)) :: ts) =
  (w/2,t1) :: (w/2,t2) :: ts
(* head of Skew Binary RA Lists *)
fun head  ((1, L x) :: _) = x
| ((w, N (x, _, _)) :: _) = x
```

```
(* Skew Binary RA lists cons *)
fun cons (x, ws as (w1,t1)::(w2,t2)::rs)=
  if w1 = w2
  then (1+w1+w2, N (x,t1,t2))::rs
  else (1, L x)::ws
| (x , ts) = (1, L x) :: ws
```

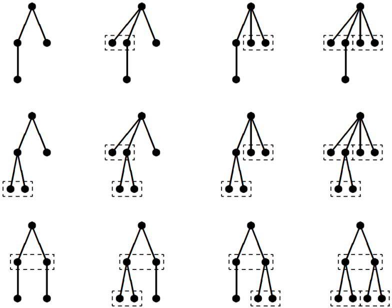
```
fun lookup ((w, t) :: ts, i) =
  if i < w then lookupTree(w,t,i)
  else lookup (ts, i - w)
fun lookupTree (1, L x, 0) = x
| (w, N(x,t1,t2), 0) = x
| (w, N(x,t1,t2), i) = if i < w/2
  then lookupTree (w/2, t1, i-1)
  else lookupTree (w/2, t2, i-1-w/2)
```

- cons, head, tail --- $O(1)$
- lookup, update --- $O(\log n)$

Skew Binary Trees



(a) simple (b) type A skew (c) a type B skew



skew binomial trees of rank 2

Skew Binary Heaps --- 2

```
(* first two defs *)
datatype Tree = N of int × Elem.T × Tree list
(* last def; we use this one for simplicity *)
datatype Tree = N of int × Elem.T × Elem.T list × Tree list
```

```
fun link (t1 as N(r,x1,xs1,c1),
          t2 as N(_,x2,xs2,c2)) =
  if   Elem.leq (x1, x2)
  then N (r+1, x1, xs1, t2::c1)
  else N (r+1, x2, xs2, t1::c2)
```

```
(* skew numbers inc *)
fun inc (ws as w1::w2::rs) =
  if w1 = w2
  then (1+w1+w2) :: rs
  else 1::ws
| ws = 1 :: ws
```

```
fun norm [] = []
| (t::ts) = insTree (t, ts)
fun merge (ts1, ts2) =
  mergeTrees (norm ts1, norm ts2)
```

```
fun skewLink (x, t1, t2) =
  let N (r, y, ys, c) = link (t1, t2)
  in if   Elem.leq (x, y)
     then N(r, x, y :: ys, c)
     else N(r, y, x :: ys, c)
```

```
(* insert in skew binomial tree *)
fun insert (x, ts as t1 :: t2 :: rs) =
  if   rank t1 = rank t2
  then skewLink (x, t1, t2) :: rs
  else N (0, x, [], []) :: ts
| (x, ts) = N (0, x, [], []) :: ts
```

```
(* exactly like ordinary BH *)
fun findMin [t] = root t
| (t :: ts) = let x = root t
                  y = findMin ts
  in if Elem.leq (x, y) then x else y
```

Skew Binary Heaps --- 2

```
(* ordinary binomial heaps *)
fun deleteMin ts = let
  fun getMin [t] = (t, [])
  | (t :: ts) = let
    let (t', ts') = getMin ts in
    if Elem.leq (root t, root t')
    then (t , ts)
    else (t', t :: ts')
  (N (_,x,ts1), ts2) = getMin ts
in merge (rev ts1, ts2)
```

```
(* skew binomial heaps *)
fun deleteMin ts = let
  fun getMin [t] = (t, [])
    | getMin (t :: ts) =
      let val (t', ts') = getMin ts in
      if Elem.leq (root t, root t')
      then (t, ts)
      else (t', t :: ts')
   (Node (_, x, xs, c), ts') = getMin ts
  fun insertAll ([], ts) = ts
  | (x::xs,ts) = insertAll(xs,insert(x,ts))
in insertAll (xs, mergeTrees (rev c,
                              norm ts'))
```

- insert --- $O(1)$
- merge, findMin, deleteMin --- same as in BH --- $O(\log n)$
- *Global Root* optimization: findMin $\rightarrow O(1)$

$$
\begin{array}{lcll}
findMin'\ (\langle x,q\rangle) & = & x & \\
insert'\ (y,\langle x,q\rangle) & = & \langle x, insert(y,q)\rangle & \text{if } x \leq y \\
insert'\ (y,\langle x,q\rangle) & = & \langle y, insert(x,q)\rangle & \text{if } y < x \\
merge'\ (\langle x_1,q_1\rangle,\langle x_2,q_2\rangle) & = & \langle x_1, insert(x_2, merge(q_1,q_2))\rangle & \text{if } x_1 \leq x_2 \\
merge'\ (\langle x_1,q_1\rangle,\langle x_2,q_2\rangle) & = & \langle x_2, insert(x_1, merge(q_1,q_2))\rangle & \text{if } x_2 < x_1 \\
deleteMin'\ (\langle x,q\rangle) & = & \langle findMin(q), deleteMin(q)\rangle &
\end{array}
$$

Data-Structural Bootstrapping: Structural Decomposition

Uniformly Recursive Data Types

```
datatype a List = Nil | Cons of a × a List
datatype a Tree = Leaf of a | Node of a Tree × a Tree
```

Non-Uniform Data Types

```
datatype a Seq = Empty | Seq of a × (a × a) Seq
```

```
(* Lists *)
fun sizeL Nil = 0
| (Cons (x, xs)) = 1 + sizeL xs
```

```
(* Seqs *)
fun sizeS Empty = 0
| (Seq (x, ps)) = 1 + 2 * sizeS ps
```

✗ type inference is undecidable

Eliminating Polymorphic Recursion

```
datatype a ElemOrPair = Elem of a | Pair of a ElemOrPair × a ElemOrPair
datatype a Seq        = Empty     | Seq  of a ElemOrPair × a Seq
```

✗ not exactly the same

✗ type safety loss

Bootstrapped Queue

Reminder: Banker's queue

front stream F is replaced by $F +\!\!+ \; reverse\ R$

$$(\cdots((f +\!\!+ \text{reverse } r_1) +\!\!+ \text{reverse } r_2) \cdots +\!\!+ \text{reverse } r_k)$$

Queue Decomposition: f, r, m

```
datatype a Queue = Empty | Queue of
  {F: a list, M: a list susp Queue, LenFM : int, R: a list, LenR : int}
```

Queue operations

```
fun enqueue (Empty, x) = Queue {[x],Empty,1,[],0}
| (Queue {f,m,lenFM,r,lenR}, x) = queue {f,m,lenFM,x::r,lenR+1}
fun head (Queue {x::f,...}) = x
fun tail (Queue {x::f,m,lenFM,r,lenR}) = queue {f,m,lenFM-1,r,lenR}

fun queue (q as {f,m,lenFM,r,lenR}) =
  if lenR <= lenFM then checkF q
  else checkF {f, M = snoc (m , $rev r), lenFM+lenR, [],0}
fun checkF {[],Empty,...} = Empty
| {[],m,l1,r,l2} = Queue {F = force (head m), M = tail m, l1,r,l2}
| q = Queue q
```

- enqueue and tail --- $O_{am}(\log^* n)$

15

- Idea: collections that contain other collections as elements and supports efficient `join` function
- Given `a C` with `insert : a x a C -> a C`
- Derive *bootstrapped type* `a B` supporing

```
insertB : a   × a B -> a B
joinB   : a B × a B -> a B
unitB   : a         -> a B

fun insertB (x , b ) = joinB  (unitB x, b)
fun joinB   (b1, b2) = insert (b1, b2)
```

- First attempt: `datatype a B = B of (a B) C` $a\ B \sim (a\ B)\ C$
- Second attempt: `datatype a B = B of a x (a B) C`
- Final attempt: `datatype a B = Empty | B of a x (a B) C`

16

```
datatype a B = Empty | B of a × (a B) C

fun unitB x = B (x , empty)

(* templates *)
fun insertB (x, Empty   ) = B (x, empty)
          | (x, B (y, c)) = B (x, insert (unitB y, c))
fun joinB (b    , Empty) = b
        | (Empty    , b) = b
        | (B (x, c), b) = B (x , insert (b, c))
```

Bootstrapping: Heaps With Efficient Merging

Let P_α be primitive heaps (priority queues)

Finding out correct bootstrapped type

1. $BP_\alpha = P_{P_\alpha}$
2. $BP_\alpha = P_{BP_\alpha}$
3. $BP_\alpha = \alpha \times P_{BP_\alpha}$
4. $BP_\alpha = \{empty\} + R_\alpha$ where $R_\alpha = \alpha \times P_{R_\alpha}$

Bootstrapped functions

$$
\begin{array}{lcll}
findMin' \ (\langle x, q\rangle) & = & x & \\
insert' \ (x, q) & = & merge'(\langle x, empty\rangle, q) & \\
merge' \ (\langle x_1, q_1\rangle, \langle x_2, q_2\rangle) & = & \langle x_1, insert(\langle x_2, q_2\rangle, q_1)\rangle & \text{if } x_1 \leq x_2 \\
merge' \ (\langle x_1, q_1\rangle, \langle x_2, q_2\rangle) & = & \langle x_2, insert(\langle x_1, q_1\rangle, q_2)\rangle & \text{if } x2 < x1 \\
deleteMin' \ (\langle x, q\rangle) & = & \langle y, merge(q_1, q_2)\rangle & \\
 & & \text{where } \langle y, q_1\rangle = findMin(q) & \\
 & & \qquad\quad q_2 = deleteMin(q) &
\end{array}
$$

Bootstrapped skew binomial heaps

- ⋗ `deleteMin` --- $O(\log n)$
- ✓ all other operations $O(1)$

Conclusion

Queues

	enqueue	head	tail
Banker's	$O_{am}(1)$	$O_{am}(1)$	$O_{am}(1)$
Physicist's	$O_{am}(1)$	$O_{am}(1)$	$O_{am}(1)$
BT	$O_{am}(\log^* n);\ O_{am}(1)$	$O(1)$	$O_{am}(\log^* n);\ O_{am}(1)$
Real-time	$O(1)$	$O(1)$	$O(1)$

Random-Access Lists

	enqueue	head	tail	lookup	update
Binary	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Skew Binary	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$

Heaps (Priority Queues)

	insert	merge	findMin	deleteMin
BH	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Skew BH	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Rooted SBH	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$
BT Skew BH	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$

Questions?

Hometask (deadline: 2 weeks)

1. Dijkstra's algorithm via BT SBH
2. Use Banker's method to implement "BankersDeque"
   all operations $O_{am}(1)$
3. * Lists supporting efficient catenation
   all operations $O_{am}(1)$

* is optional