S8.1: Introdution to Purely Functional Data Structures

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Purely Functional Data Structures

- Read chapters 1, 2, 3, and 5 of Chris Okasaki's book.
- Much of our discussion will use the figures from the text. Thus, these slides are rather incomplete.

Chris's amotivation

- "To rectify this imbalance, this book describes data structures from a functional point of view", on
- "However, there is on aspect of functional programming that no amount of cleverness on the part of the compiler writer is likely to mitigate — the use of inferior or inappropriate data structures."

Okasaki, page 1.

Two basic challenges

1. no mutation (updating) of data
2. data structures are *persistent* not just *ephemeral* as in imperative languages.

Persistence

- *New* data structure share some components with *old* data structures.
- In building them, they will contain parts of the old ones.
- Start by considering lists.
- ML structures and signatures
  Review figures on page 8.
- List append
  - in imperative setting - Fig 2.4
  - in functional setting - Fig 2.5 and code at bottom of page 9.
- Similar case for `update` - code on page 10, figure 2.6

Trees

- We can implement sets (Fig 2.7) using binary trees.
- See `member` and also in `insert` how persistence is achieved via sharing (Fig 2.8).

Some sample data structures: Chap 3

- Leftist Heaps
- Binomial Heaps
- Red-Black Trees

Our aim is to avoid the pointer nightmare one encounters in imperative languages.

Leftist Heaps

- useful if you only need to know the *minimal* element in a set or map.
- a.k.a. priority queue or heap
- See Figure 3.1

Leftist Heaps

```
type Heap
  = E
  | T of int * Elem.T * Heap * Heap

let rank t = match t with
  | E -> 0
  | T (r, _, _, _) -> r

let makeT (x, a, b) =
     if rank a >= rank b
     then T ((rank b)+1, x, a, b)
     else T ((rank a)+1, x, b, a)
```

Merging Leftist Heaps

```
let rec merge t1 t2 = match t1, t2 with
  | h, E -> h
  | E, h -> h
  | h1 as T(_,x,a1,b1), h2 as T(_,y,a2,b2) ->
     if Elem.leq (x,y)
     then makeT x a1 (merge b1 h2)
     else makeT y a2 (merge h1 b2)
```

Why does `merge` run in $O(log\ n)$ time?

Leftist Heaps

With `merge`, the rest are easy.

```
let insert x h = merge (T(1,x,E,E)) h

let findMin (T(_, x, _, _)) = x

let deleteMin (T(_, x, a, b)) = merge a b
```


Leftist Heaps - almost all on one

```
type Heap = E | T of int * Elem.T * Heap * Heap
let rank t = match t with
  | E -> 0
  | T (r, _, _, _) -> r
let makeT (x, a, b) =
     if rank a >= rank b
     then T (rank b+1, x, a, b)
     else T (rank a+1, x, b, a)
let rec merge t1 t2 = match t1, t2 with
  | h, E -> h
  | E, h -> h
  | h1 as T(_,x,a1,b1), h2 as T(_,y,a2,b2) ->
     if Elem.leq (x,y)
     then makeT x a1 (merge b1 h2)
     else makeT y a2 (merge h1 b2)
let insert x h = merge (T(1,x,E,E)) h
```

Binomial Heaps

- made up of *binomial trees*
- binomial trees:
  - binomial tree of rank 0 is a single node
  - binomial tree of rank $r+1$ links two binomial trees of rank r,
    makes one tree the left most child of the other.
- see figure 3.3
- binomial tree of rank r has 2^r nodes

Binomial Heaps

An alternate definition:

- a binomial heap of rank r is a node with r children, $t_1, ..., t_r$, which each t_i is a binomial tree of rank $r - i$.
- Does this hold for your drawings of rank 4 and rank 5 trees?

Implementing binomial heaps

```
type Tree = Node of int * Elem.t * Tree list
```

- Trees in list are in decreasing order of rank
- Elements are stored in "heap order"

```
let link (t1 as Node (r, x1, c1)
          t2 as Node (_, x2, c2)) =
    if x1 <= x2
    then Node (r+1, x1, t2 :: c1)
    else Node (r+1, x2, t1 :: c2)
```

- try linking two sample trees of rank 0, then 1, then 2
- we only link trees of the same rank
- what is the ML representation of all of these?

Binomial Heap

- is a list of heap-ordered binomial trees
- `type Heap = Tree list`
  stored in order of increasing rank
- Recall, a binomial tree contains exactly 2^r nodes
- A binomial heap with n elements corresponds to a binary representation of the number n.
- What does a binomial heap with 5 elements look like?
- What does a binomial heap with 21 elements look like?
- Draw these.

Functions on binomial heaps

```
let rank (Node (r, x, c)) = r

let rec insTree t1 t2 = match t1, t2 with
  | t, [] ->= [t]
  | t, ts as t' :: ts' ->
      if rank t < rank t'
      then t :: ts
      else insTree (link (t, t')) ts'

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

Insert values 30, 20, 10, 40 into [ ], do this using ML data structures instead of just the drawings.

merging heaps

```
let rec merge trees1 trees2 = match trees1, trees2 witl
  | ts1, [] -> ts1
  | [], ts2 -> ts2
  | ts1 as t1::ts1', ts2 as t2::ts2' ->
      if rank t1 < rank t2
      then t1 :: merge ts1' ts2
      else if rank t2 < rank t1
           then t2 :: merge ts1 ts2'
           else insTree (link(t1, t2))
                        (merge ts1' ts2'))
```

Step through lists in increasing rank, link trees of equal rank

linking is like carrying in binary arithmetic

Exercise: merge [10, 20-30] with [15]

Minimum

```
let rec removeMinTree trees = match trees with
  | [t] -> (t, [])
  | t :: ts ->
        let (t', ts') = removeMinTree ts
        in if root t < root t'
            then (t, ts)
            else (t', t::ts')
        end

let findMin ts = let val (t, _) = removeMinTree ts
                 in root t end
```


Delete minimum

```
let deleteMin ts =
  let (Node (_, _, ts1), ts2) = removeMinTree ts
  in merge (rev ts1, ts2)
  end
```

Why do we reverse `ts1` here?

Red-black trees

- binary trees perform well with random or unordered data, but poorly with ordered data
  operations degenerate from $O(log\ n)$ time to $O(n)$
- red-black trees, popular form of balanced binary tree.
- each node is colored red or black
- ```
  type Color = R | B
  type Tree = E
            | T of Color * Tree * int * Tree
  ```
- empty nodes (`E`) are black
  ```

Red-black trees

- Invariants:
  - no red node has a red child
  - all paths from root have same number of black nodes
- longest path is one with alternating red-black nodes
- shortest will have only black nodes
- length of longest is no more than twice the length of the shortest

Membership

```
let rec member elem tree = match elem, tree with
  | x, E -> false
  | x, T(_, a, y, b) ->
      if x < y
      then member x a
      else
      if x > y
      then member x b
      else true
```

- as expected
- colors don't matter

Insertion

```
let insert x s =
  let rec ins t = match t with
        | E -> T(R, E, x, E)
        | s as T(color, a, y, b) ->
            if x < y
            then balance color (ins a) y b
            else
            if x > y
            then balance color a y (ins b)
            else s
        in let T(_, a, y, b) = ins s
  in T(B, a, y, b)
  end
```

Balance

```
let balance c t1 v t2 = match color, t1, v, t2 with
  | B, T(R, T(R,a,x,b), y,c), z, d
  | B, T(R, a,x, T(R,b,y,c)), z, d
  | B, a, x, T(R, T(R,b,y,c), z, d)
  | 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 body = T body
```

called on insertion of `x` in tree
`ins (s as T(color, a, y, b))`
as either
`balance(color, ins a, y, b)`
or
`balance(color, a, y, ins b)`