~ Lecture VIII ~

Keywords:

tree-based data structures; binary search trees; red/black trees; flexible functional arrays; heaps; priority queues.

References:

- [MLWP, Chapters 4 and 7]
- [PFDS, Chapters 2(§2), 3(§3), and 5(§2)]

Binary search trees

A *binary search tree* is a binary tree with nodes where data items are stored with the property that,

> for every node in the tree, the elements of the left subtree are smaller than or equal to that of the node which in turn is smaller than the elements of the right subtree.

Thus,

> the inorder listing of a binary search tree is a sorted list.

Applications:

1. Binary search trees offer a simple way to represent *sets*; in which case, to eliminate repetitions, it is natural to impose the extra condition that elements of left subtrees are strictly smaller than their respective roots.

   When balanced, they admit $O(n \log n)$ runtime for all basic operations.

2. Binary search trees can also be easily extended to act as *dictionaries*, mapping keys to values.

We can test membership in a binary search tree using `lookup`:

```
fun lookup x empty = false
  | lookup x ( node(v,l,r) )
      = ( x = v )
          orelse
            ( if x < v then (lookup x l)
              else ( lookup x r ) ) ;

val lookup = fn : int -> int tree -> bool
```

To insert a new value, we just need to find its proper place:

```
fun insert x empty = node( x , empty , empty )
  | insert x ( node(v,l,r) )
      = if x <= v then node( v , insert x l , r )
        else node( v , l , insert x r ) ;

val insert = fn : int -> int tree -> int tree
```

We could thus sort a list by the following procedure:

```
val sort
= inorder o ( foldl ( fn(x,t) => insert x t ) empty ) ;

val sort = fn : int list -> int list
```

Red/Black trees

Binary search trees are simple and perform well on random or unordered data, but they perform poorly on ordered data, degrading to $O(n)$ performance for common operations. Red/black trees are a popular family of *balanced* binary search trees.

Every node in a red/black tree is colored either red or black.

```
datatype
  'a RBtree
    = O
    | R of 'a * 'a RBtree * 'a RBtree
    | B of 'a * 'a RBtree * 'a RBtree ;
```

NB: Empty nodes are considered to be black.

We insist that every red/black tree satisfies the following two *balance invariants*:

1. No red node has a red child.
2. Every path from the root to an empty node contains the same number of black nodes.

Together, these invariants ensure that the longest possible path, one with alternating black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only.

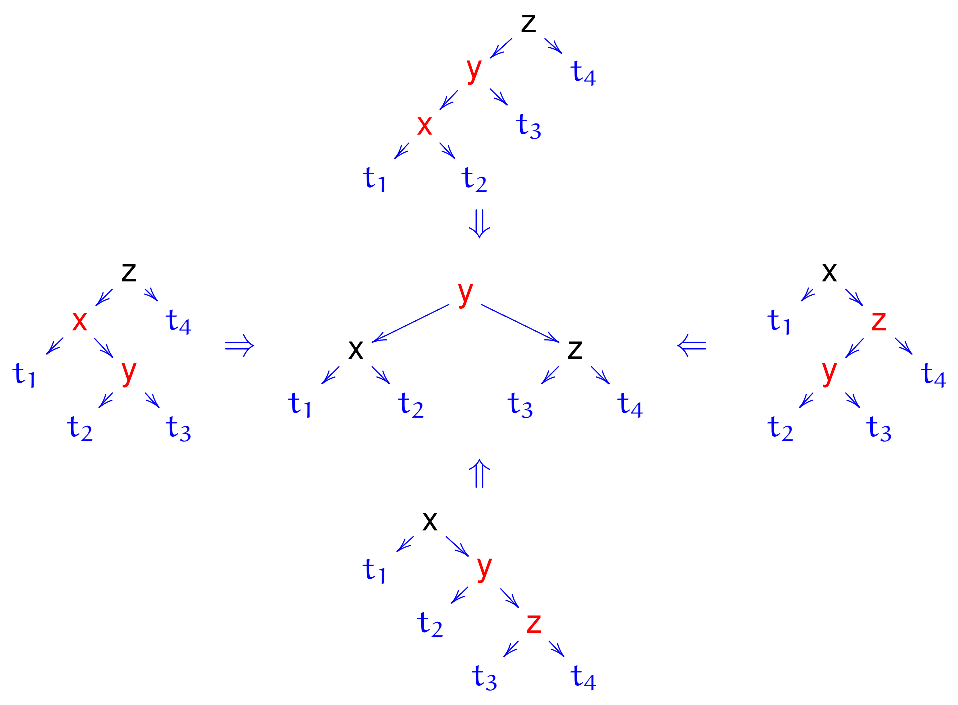
Red/Black tree insert

```
fun RBinsert x t
  = let fun ins( O ) = R( x , O , O )                          (*1*)
          | ins( R( y , l , r ) )
            = if x <= y then R( y , ins l , r )
              else R( y , l , ins r )
          | ins( B( y , l , r ) )
            = if x <= y
              then BALANCE( B( y , ins l , r ) )               (*3*)
              else BALANCE( B( y , l , ins r ) ) ;             (*3*)
    in case  ins t  of
            R(x',l,r) => B(x',l,r)                              (*2*)
          | t' => t'
    end ;
```

This function extends the insert function for unbalanced search trees in three significant ways.

1. When we create a new node, we initially color it red.
2. We force the final root to be black, regardless of the color returned by `ins`.
3. We include calls to a `BALANCE` function, that massages its arguments as indicated in the figure on the following page to enforce the balance invariants.

   NB: We allow a single red-red violation at a time, and percolate this violation up the search path towards the root during rebalancing.



```
fun BALANCE
  ( B( z , R( y , R( x , t1 , t2 ) , t3 ) , t4 )
  | B( z , R( x , t1 , R( y , t2 , t3 ) ) , t4 )
  | B( x , t1 , R( y , t2 , R( z , t3 , t4 ) ) )
  | B( x , t1 , R( z , R( y , t2 , t3 ) , t4) )
    ) = R( y , B( x , t1 , t2 ) , B( z , t3 , t4 ) )
  | BALANCE t
      = t ;

val BALANCE = fn : 'a RBtree -> 'a RBtree
```

Functional arrays

A *functional* array provides a mapping from an initial segment of natural numbers to elements, together with `lookup` and `update` operations.

A *flexible* array augments the above operations to insert or delete elements from either end of the array.

```
signature UpperFlexARRAY =
sig  type 'a t
     exception Subscript
     val empty: 'a t
     val null: 'a t -> bool
     val length: 'a t -> int
     val lookup: 'a t * int -> 'a
     val update: 'a t * int * 'a -> 'a t
     val shrink: 'a t -> 'a t              end ;
```

We will provide an implementation based on the following tree-based data structure.

```
structure TreeArrayMod =
struct
  abstype 'a t = O | V of 'a * 'a t * 'a t with
  exception Subscript ;
  val empty = O ;
  fun lookup( V(v,tl,tr) , k )
    = if k = 1 then v
      else if k mod 2 = 0
           then lookup( tl , k div 2 )
           else lookup( tr , k div 2 )
    | lookup( O , _ )
      = raise Subscript ;
```

```
  fun update( O , k , v )
        = if k = 1 then ( V(v,O,O) , 1 )
          else raise Subscript
    | update( V(x,tl,tr) , k , v )
        = if k = 1 then ( V(v,tl,tr) , 0 )
          else if k mod 2 = 0
               then let
                 val (t,i)
                   = update(tl,k div 2,v)
               in  ( V(x,t,tr) , i )  end
               else let
                 val (t,i)
                   = update(tr,k div 2,v)
               in  ( V(x,tl,t) , i )  end ;
```

```
  fun delete( O , n )
    = raise Subscript
    | delete( V(v,tl,tr) , n )
      = if n = 1 then O
        else if n mod 2 = 0
             then V( v , delete(tl,n div 2) , tr )
             else V( v , tl , delete(tr,n div 2) ) ;
  end ;
end ;
```

An implementation of *upper flexible* functional arrays follows.

NB: The implementation can be extended to also provide *downwards* flexibility in logarithmic time. Consider this as an exercise, or consult [MLWP, 4.15].

```
structure TreeArray : UpperFlexARRAY =
struct
  abstype 'a t = A of int * 'a TreeArrayMod.t with
  exception Subscript ;
  val empty = A( 0 , TreeArrayMod.empty ) ;
  fun null( A(l,_) ) = l = 0 ;
  fun length( A(l,_) ) = l ;
fun lookup( A(l,t) , k )
  = if l = 0 orelse ( k < 1 andalso k > l )
    then raise Subscript
    else TreeArrayMod.lookup(t,k) ;

fun update( A(l,t) , k , v )
  = if 1 <= k andalso k <= l+1
    then let
           val (u,i) = TreeArrayMod.update( t , k , v )
         in
           A( l+i , u )
         end
    else raise Subscript ;
```

```
  fun shrink( A(l,t) )
    = if l = 0 then empty
      else A( l-1 , TreeArrayMod.delete(t,l) ) ;
  end ;
end ;
```

Priority queues using heaps

A *priority queue* is an ordered collection of items. Items may be inserted in any order, but only the *highest priority* item (typically taken to be that with *lower numerical value*) may be seen or deleted.

```
signature PRIORITY_QUEUE =
sig  exception Size
     type item
     type t
     val empty: t
     val null: t -> bool
     val insert: item -> t -> t
     val min: t -> item
     val delmin: t -> t          end ;
```

A *heap* is a binary tree in which the labels are arranged so that every label is less than or equal to all labels below it in the tree. This *heap condition* puts the labels in no strict order, but does put the least label at the root.

The following *functional priority queues* are based on flexible arrays that are heaps.

```
structure Heap: PRIORITY_QUEUE =
struct
  exception Size ;
  type item = real ;
  abstype 'a tree
    = O | V of 'a * 'a tree * 'a tree with
  type t = item tree ;
```

```
val empty = O ;

fun null O = true
  | null _ = false ;

fun insert (w:item) O
    = V( w , O , O )
  | insert w ( V( v , l , r ) )
    = if w <= v then V( w , insert v r , l )
      else V( v , insert w r , l ) ;

fun min( V(v,_,_) ) = v
  | min O = raise Size ;
```

```
local
  exception Impossible ;

  fun leftrem( V( v , O , O ) ) = ( v, O )
    | leftrem( V( v , l , r ) )
      = let
          val (w,t) = leftrem l
          in
            ( w , V(v,r,t) )
        end
    | leftrem _ = raise Impossible ;
```

```
  fun siftdown( w:item , O , O ) = V( w , O , O )
    | siftdown( w , t as V(v,O,O) , O )
        = if w <= v then V( w , t , O )
          else V( v , V(w,O,O) , O )
    | siftdown( w , l as V(u,ll,lr) , r as V(v,rl,rr) )
        = if w <= u andalso w <= v then V(w,l,r)
          else if u <= v then V( u , siftdown(w,ll,lr) ,r)
               else V( v , l , siftdown(w,rl,rr) )
    | siftdown _ = raise Impossible ;
```

```
  in
    fun delmin O = raise Size
      | delmin( V(v,O,_) ) = O
      | delmin( V(v,l,r) )
        = let
            val (w,t) = leftrem l
          in
            siftdown( w , r , t )
          end
  end ;

  end ;
end ;
```

Heap sort

```
fun heapTOlist h
  = if Heap.null h then []
    else (Heap.min h) :: heapTOlist(Heap.delmin h) ;

val sort
  = heapTOlist
      o foldl (fn(v,h) => Heap.insert v h) Heap.empty ;
```