~ Lecture V ~

Keywords:
sorting: insertion sort, quick sort, merge sort; parametric sorting; queues; signatures; structures; functors; generic sorting.

References:

- [MLWP]
  Chapter 2, § Introduction to modules
  Chapter 3, § Sorting: A case study
  Chapter 7, § Three representations of queues
- [PFDS]
  Section 5.2, § Queues

Sorting

Insertion sort

Insertion sort works by inserting the items, one at a time, into a sorted list.

```
fun ins( x , []) = [x]
  | ins( x , h::t ) =
      if x <= h then x::h::t
      else h::ins(x,t) ;

val sort = foldl ins [] ;

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

Insertion sort takes $O(n^2)$ comparisons in average and in the worst case.

Sorting

Quick sort

Quick sort works by picking up a pivot value and partitioning the list into two lists: values less than or equal to the pivot and values greater than the pivot. The sort is applied recursively to the two partitions and the resulting (sorted) lists are concatenated.

```
fun sort [] = []
  | sort (h::t) =
      case List.partition ( fn x => x <= h ) t of
        (left,right) => sort left @ h::sort right ;

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

Quick sort takes $O(n \log n)$ comparisons on average and $O(n^2)$ in the worst case.

Quick sort

without List.partition and @

```
fun quick( [] , sorted ) = sorted
  | quick( [x] , sorted ) = x::sorted
  | quick ( h::t, sorted ) =
      let fun part( left , right , [] ) =
                quick( left , h::quick(right,sorted) )
            | part( left , right , h1::t1 ) =
                if h1 <= h
                then part(h1::left,right,t1)
                else part(left,h1::right,t1)
      in
        part( [] , [] , t )
      end ;
fun sort l = quick( l, [] ) ;
```

Sorting

Merge sort

Merge sort is another divide-and-conquer sorting algorithm. It works by dividing the input into two halves, sorting each of them, and then merging them in order.

The implementation below is *top-down*; it sorts one half of the list before starting with the other half. A *bottom-up* approach is also possible; starting with individual elements and merging them into larger lists until the sort is complete.

Merge sort takes $O(n \log n)$ comparisons on average and in the worst case.


```
fun merge( [] , l2 ) = l2
  | merge( l1 , [] )  = l1
  | merge( l1 as h1::t1 , l2 as h2::t2 ) =
      if h1 <= h2 then h1::merge( t1 , l2 )
      else h2::merge( l1 , t2 ) ;
fun sort [] = []
  | sort [x] = [x]
  | sort l = let
                val n = length l div 2
              in
                merge( sort( List.take(l,n) ) ,
                       sort( List.drop(l,n) ) )
              end ;
```

? Why do we need two base cases? What happens if we declare `val n = 1` in the `sort`ing function?

Merge sort

without `take` *and* `drop`

```
fun sort l =
    let fun msort( 0 , l ) = ( [] , l )
          | msort( 1 , h::t ) = ( [h] , t )
          | msort( n , l ) =
            let val (sl1,l1) = msort( (n+1) div 2 , l )
                val (sl2,l2) = msort( n div 2 , l1 )
            in  ( merge(sl1,sl2) , l2 )  end
    in
      case msort( length l , l ) of
        (sl,_) => sl
    end ;
```


Parametric sorting

```
fun msort comp [] = []
  | msort comp [x] = [x]
  | msort comp l =
      let fun merge( [] , l2 ) = l2
            | merge( l1 , [] )  = l1
            | merge( l1 as h1::t1 , l2 as h2::t2 ) =
                if comp(h1,h2)
                then h1 :: merge( t1 , l2 )
                else h2 :: merge( l1 , t2 )
          val n = length l div 2
      in
        merge( msort comp ( List.take(l,n) ) ,
               msort comp ( List.drop(l,n) ) )
      end ;
```

```
> val 'a msort
    = fn : ('a * 'a -> bool) -> 'a list -> 'a list
```

Two sample specialisations:

```
val leqintmsort = msort op<= ;
```

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

```
load"Real" ;
val geqrealmsort = msort Real.>= ;
```

```
> val geqrealmsort = fn : real list -> real list
```


Queues

Queue is an abstract data type allowing for the insertion and removal of elements in a first-in-first-out (FIFO) discipline. It provides the following operations:

```
signature QUEUE = sig
type 'a t                       (* type of queues *)
val empty: 'a t                 (* the empty queue *)
val null: 'a t -> bool          (* test for empty queue *)
val enq: 'a -> 'a t -> 'a t     (* add to end *)
val deq: 'a t -> 'a t           (* remove from front *)
val hd: 'a t -> 'a              (* return the front element *)
end ;
```



```
> signature QUEUE =
  /\t.
    {type 'a t = 'a t,
     val 'a empty : 'a t,
     val 'a null : 'a t -> bool,
     val 'a enq : 'a -> 'a t -> 'a t,
     val 'a deq : 'a t -> 'a t,
     val 'a hd : 'a t -> 'a}
```


An implementation

A fast and simple implementation of queues can be done with an ordered pair of lists. The first list contains the front elements of the queue in order and the second list contains the rear elements in reverse order.

For instance, the queue with integers $[-2..2]$ is represented by any one of the following:

```
( [] , [2,1,0,~1,~2] )          ( [~2,~1,0] , [2,1] )
( [~2] , [2,1,0,~1] )           ( [~2,~1,0,1] , [2] )
( [~2,~1] , [2,1,0] )           ( [~2,~1,0,1,2] , [] )
```

The head of the queue is the head of the first list, so `hd` returns this element and `deq` removes it. To add an element to the queue, we just add it to the beginning of the second list.

To ensure that `hd` always succeds on a non-empty queue, we must maintain the *invariant* that if the first list is empty then so is the second one. When the first list is exhausted, we move the reverse of the second list to the front. This needs to happen in `enq` when the queue is empty, and in `deq` when the first list is a singleton.


```
structure Queue : QUEUE =
  struct
    type 'a t =  'a list * 'a list ;
    val empty = ([],[]) ;
    fun null( ([],[]) ) = true
      | null _ = false ;
    fun enq x ([],_) = ( [x] , [] )
      | enq x (front,back) = ( front , x::back ) ;
    fun deq( (_::[],back) ) = ( rev back , [] )
      | deq( (_::rest,back) ) = ( rest , back )
      | deq( _ ) = empty ;
    fun hd( (head::rest,_) ) = head ;
  end ;
```



```
> structure Queue :
  {type 'a t = 'a list * 'a list,
   val 'a deq :
     'a list * 'a list -> 'a list * 'a list,
   val 'a empty : 'a list * 'a list,
   val 'a enq :
     'a -> 'a list * 'a list -> 'a list * 'a list,
   val 'a hd : 'a list * 'a list -> 'a,
   val 'a null : 'a list * 'a list -> bool}
```

```
fun buildq F
  = foldl ( fn(f,q) => f q ) Queue.empty F ;
```

```
val buildq = fn :
      ('a Queue.t -> 'a Queue.t) list -> 'a Queue.t
```


Generic orders

```
signature ORDER =
  sig
    type t
    val leq: t * t -> bool
  end ;
```

```
> signature ORDER
    = /\t.{type t = t, val leq : t * t -> bool}
```

```
structure LeqIntOrder =
  struct
    type t = int ;
    val leq = op<= ;
  end ;
load"Real" ;
structure LeqRealOrder =
  struct
    type t = real ;
    val leq = Real.<= ;
  end ;
> structure LeqIntOrder :
    {type t = int, val leq : int * int -> bool}
> structure LeqRealOrder :
    {type t = real, val leq : real * real -> bool}
```



```
functor Op( O: ORDER ) : ORDER =
  struct
    type t = O.t ;
    fun leq(x,y) = O.leq(y,x) ;
  end ;
> functor Op :
  !t.
   {type t = t, val leq : t * t -> bool}
    -> {type t = t, val leq : t * t -> bool}
structure GeqIntOrder = Op(LeqIntOrder) ;
structure GeqRealOrder = Op(LeqRealOrder) ;
> structure GeqIntOrder :
    {type t = int, val leq : int * int -> bool}
> structure GeqRealOrder :
    {type t = real, val leq : real * real -> bool}
```


Generic sorting

```
signature SORTER =
  sig
    type t
    val sort: t list -> t list
  end ;

> signature SORTER =
    /\t.{type t = t, val sort : t list -> t list}

functor MergeSort (O: ORDER) : SORTER =
```



```
  struct (* MergeSort *)
    type t = O.t ;
    fun merge( [] , l2 ) = l2
      | merge( l1 , [] )  = l1
      | merge( l1 as h1::t1 , l2 as h2::t2 ) =
          if O.leq(h1,h2) then h1::merge( t1 , l2 )
          else h2::merge( l1 , t2 ) ;
    fun sort [] = []
      | sort [x] = [x]
      | sort l = let val n = length l div 2 in
                   merge( sort( List.take(l,n) ) ,
                          sort( List.drop(l,n) ) )
                 end ;
  end (* MergeSort *) ;
```

```
> functor MergeSort :
  !t.
   {type t = t, val leq : t * t -> bool}
     -> {type t = t, val sort : t list -> t list}
structure LeqIntMergeSort = MergeSort( LeqIntOrder ) ;
structure GeqRealMergeSort = MergeSort( GeqRealOrder ) ;
> structure LeqIntMergeSort :
    {type t = int, val sort : int list -> int list}
> structure GeqRealMergeSort :
    {type t = real, val sort : real list -> real list}
- LeqIntMergeSort.sort ;
> val it = fn : int list -> int list
- GeqRealMergeSort.sort ;
> val it = fn : int real -> real list
```