Interface for module Heap

1. Heaps

```
module type Ordered = sig
  type t
  val compare : t -> t -> int
end

exception EmptyHeap
```

2. Imperative implementation.

```
module Imperative(X : Ordered) : sig
```

Type of imperative heaps. (In the following n refers to the number of elements in the heap)

```
  type t
```

create c creates a new heap, with initial capacity of c

```
  val create : int -> t
```

is_empty h checks the emptiness of h

```
  val is_empty : t -> bool
```

add x h adds a new element x in heap h; size of h is doubled when maximum capacity is reached; complexity $O(log(n))$

```
  val add : t -> X.t -> unit
```

maximum h returns the maximum element of h; raises *EmptyHeap* when h is empty; complexity $O(1)$

```
  val maximum : t -> X.t
```

remove h removes the maximum element of h; raises *EmptyHeap* when h is empty; complexity $O(log(n))$

```
  val remove : t -> unit
```

pop_maximum h removes the maximum element of h and returns it; raises *EmptyHeap* when h is empty; complexity $O(log(n))$

```
  val pop_maximum : t -> X.t
```

usual iterators and combinators; elements are presented in arbitrary order

```
  val iter : (X.t -> unit) -> t -> unit
```

val $fold$: $(X.t \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$

end

3. Persistent implementation.

module type $PersistentSig$ = sig

heap elements

type elt

Type of functional heaps

type t

The empty heap

val $empty$: t

$add\ x\ h$ returns a new heap containing the elements of h, plus x; complexity $O(log(n))$

val add : $elt \rightarrow t \rightarrow t$

$maximum\ h$ returns the maximum element of h; raises $EmptyHeap$ when h is empty; complexity $O(1)$

val $maximum$: $t \rightarrow elt$

$remove\ h$ returns a new heap containing the elements of h, except the maximum of h; raises $EmptyHeap$ when h is empty; complexity $O(log(n))$

val $remove$: $t \rightarrow t$

usual iterators and combinators; elements are presented in arbitrary order

val $iter$: $(elt \rightarrow unit) \rightarrow t \rightarrow unit$

val $fold$: $(elt \rightarrow \alpha \rightarrow \alpha) \rightarrow t \rightarrow \alpha \rightarrow \alpha$

end

module $Persistent(X : Ordered)$: $PersistentSig$ with type $elt = X.t$

Module Heap

4. Heaps

module type $Ordered$ = sig
  type t
  val $compare$: $t \rightarrow t \rightarrow int$
end

```
exception EmptyHeap
```

5. Imperative implementation

```
module Imperative(X : Ordered) = struct
```

The heap is encoded in the array *data*, where elements are stored from 0 to $size - 1$. From an element stored at i, the left (resp. right) subtree, if any, is rooted at $2\times i+1$ (resp. $2\times i+2$).

```
  type t = { mutable size : int; mutable data : X.t array }
```

When *create n* is called, we cannot allocate the array, since there is no known value of type $X.t$; we'll wait for the first addition to do it, and we remember this situation with a negative size.

```
  let create n =
    if n ≤ 0 then invalid_arg "create";
    { size = − n; data = [||] }

  let is_empty h = h.size ≤ 0
```

resize doubles the size of *data*

```
  let resize h =
    let n = h.size in
    assert (n > 0);
    let n′ = 2 × n in
    let d = h.data in
    let d′ = Array.create n′ d.(0) in
    Array.blit d 0 d′ 0 n;
    h.data ← d′

  let add h x =
    (∗ first addition: we allocate the array ∗)
    if h.size < 0 then begin
      h.data ← Array.create (− h.size) x; h.size ← 0
    end;
    let n = h.size in
    (∗ resizing if needed ∗)
    if n ≡ Array.length h.data then resize h;
    let d = h.data in
    (∗ moving x up in the heap ∗)
    let rec moveup i =
      let fi = (i − 1) / 2 in
      if i > 0 ∧ X.compare d.(fi) x < 0 then begin
```

```
        d.(i) ← d.(fi);
        moveup fi
      end else
        d.(i) ← x
    in
    moveup n;
    h.size ← n + 1

  let maximum h =
    if h.size ≤ 0 then raise EmptyHeap;
    h.data.(0)

  let remove h =
    if h.size ≤ 0 then raise EmptyHeap;
    let n = h.size − 1 in
    h.size ← n;
    let d = h.data in
    let x = d.(n) in
    (* moving x down in the heap *)
    let rec movedown i =
      let j = 2 × i + 1 in
      if j < n then
        let j =
          let j' = j + 1 in
          if j' < n ∧ X.compare d.(j') d.(j) > 0 then j' else j
        in
        if X.compare d.(j) x > 0 then begin
          d.(i) ← d.(j);
          movedown j
        end else
          d.(i) ← x
      else
        d.(i) ← x
    in
    movedown 0

  let pop_maximum h = let m = maximum h in remove h; m

  let iter f h =
    let d = h.data in
    for i = 0 to h.size − 1 do f d.(i) done
```

```
let fold f h x0 =
  let n = h.size in
  let d = h.data in
  let rec foldrec x i =
    if i ≥ n then x else foldrec (f d.(i) x) (succ i)
  in
  foldrec x0 0
end
```

6. Persistent implementation

```
module type PersistentSig = sig
  type elt
  type t
  val empty : t
  val add : elt → t → t
  val maximum : t → elt
  val remove : t → t
  val iter : (elt → unit) → t → unit
  val fold : (elt → α → α) → t → α → α
end
```

```
module Persistent(X : Ordered) = struct
```

Heaps are encoded as complete binary trees, i.e., binary trees which are full expect, may be, on the bottom level. These trees also enjoy the heap property, namely the value of any node is greater or equal than those of its left and right subtrees.

The representation invariant is the following: the number of nodes in the left subtree is equal to the number of nodes in the right subtree, or exceeds it by exactly once. In the first case, we use the constructor *Same* and in the second the constructor *Diff*. Then it can be proved that $2\hat{}(h-1) \le n \le 2\hat{}h$ when n is the number of elements and h the height of the tree.

```
type elt = X.t

type t =
  | Empty
  | Same of t × X.t × t (* same number of elements on both sides *)
  | Diff of t × X.t × t (* left has n + 1 nodes and right has n *)

let empty = Empty
```

```
let rec add x = function
  | Empty ->
      Same (Empty, x, Empty)
  (* insertion to the left *)
  | Same (l, y, r) ->
      if X.compare x y > 0 then Diff (add y l, x, r) else Diff (add x l, y, r)
  (* insertion to the right *)
  | Diff (l, y, r) ->
      if X.compare x y > 0 then Same (l, x, add y r) else Same (l, y, add x r)

let maximum = function
  | Empty -> raise EmptyHeap
  | Same (_, x, _) | Diff (_, x, _) -> x
```

extracts one element on the bottom level of the tree, while maintaining the representation invariant

```
let rec extract_last = function
  | Empty -> raise EmptyHeap
  | Same (Empty, x, Empty) -> x, Empty
  | Same (l, x, r) -> let y, r' = extract_last r in y, Diff (l, x, r')
  | Diff (l, x, r) -> let y, l' = extract_last l in y, Same (l', x, r)
```

removes the topmost element of the tree and inserts a new element x

```
let rec descent x = function
  | Empty ->
      assert false
  | Same (Empty, _, Empty) ->
      Same (Empty, x, Empty)
  | Diff (Same (_, z, _) as l, _, Empty) ->
      if X.compare x z > 0 then Diff (l, x, Empty)
      else Diff (Same (Empty, x, Empty), z, Empty)
  | Same (l, _, r) ->
      let ml = maximum l in
      let mr = maximum r in
      if X.compare x ml > 0 && X.compare x mr > 0 then
        Same (l, x, r)
      else
        if X.compare ml mr > 0 then
          Same (descent x l, ml, r)
        else
          Same (l, mr, descent x r)
```

```
  | Diff (l, _, r) →
      let ml = maximum l in
      let mr = maximum r in
      if X.compare x ml > 0 ∧ X.compare x mr > 0 then
        Diff (l, x, r)
      else
        if X.compare ml mr > 0 then
          Diff (descent x l, ml, r)
        else
          Diff (l, mr, descent x r)

let remove = function
  | Empty → raise EmptyHeap
  | Same (Empty, x, Empty) → Empty
  | h → let y, h' = extract_last h in descent y h'

let rec iter f = function
  | Empty → ()
  | Same (l, x, r) | Diff (l, x, r) → iter f l; f x; iter f r

let rec fold f h x0 = match h with
  | Empty → x0
  | Same (l, x, r) | Diff (l, x, r) → fold f l (fold f r (f x x0))

end
```