4 Persistent Disjoint Sets

This challenge is inspired by a paper of Sylvain Conchon and Jean-Christophe Filliâtre [1].

A disjoint set (aka union-find) data structure maintains a partition of a finite set. The data structure supports two operations: finding the class of a given element of the set and updating the data structure by taking the union of two classes. Each class is uniquely identified by one of its members, the class' *representative*.

For simplicity, we assume that the underlying set is $[0, \ldots, n)$ for some natural number n that is chosen when the data structure is initialized. Our goal is to implement a persistent version of the data structure. That is, the union operation does not modify the structure in place, but instead returns an updated version. This suggests the following minimal OCaml signature for the operations of the data structure:

```
val create : int → dset
val find : dset → int → int
val union : dset → int → int → dset
```

Our implementation follows the classical optimal imperative algorithm [2, 3], which represents each class as an inverted tree whose nodes are the elements of the class (see Fig. 1). The tree's root is the representative of the class. The inverted trees for all classes are stored in an array `parent` of size n that assigns each element to its parent in the tree of its class. A class representative is its own parent, identifying a root.

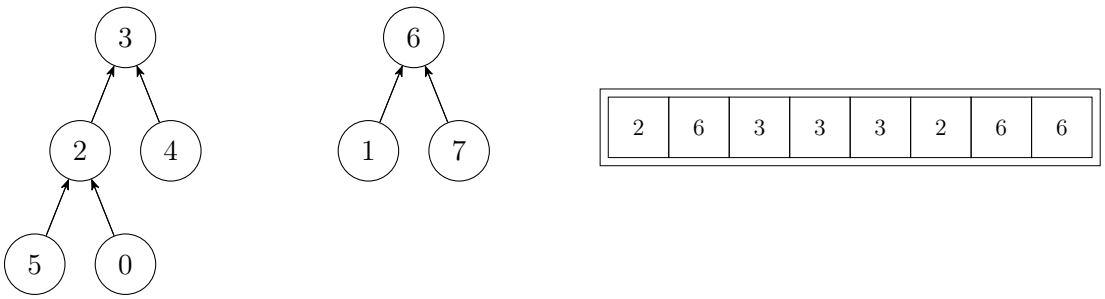

Figure 1: State of a disjoint set structure with 8 elements and two classes

A `find ds x` operation follows the parent pointers until it reaches the root `r`, which is the representative that the operation will return. Crucial for the efficiency of the algorithm is the *path compression optimization*: when the recursion stack of the traversal is unwound, `find` reroutes the parent pointer of each node on the path directly to the root `r`. This guarantees that subsequent `find` operations on these nodes will terminate in constant time.

A `union ds x y` operation uses `find` to identify the representatives `rx` and `ry` of `x`'s and `y`'s classes and then lets `rx` become the new parent of `ry` or vice versa. To decide between these two alternatives, the data structure maintains a second array `size` that assigns each element to the number of its descendants (i.e. the size of its subtree). The new root is chosen to be `rx` if and only if its size is greater than or equal to that of `ry`.

To obtain a persistent data structure, we build on the persistent arrays from Challenge 3 but otherwise follow the classical imperative algorithm. The representation of the type `dset` is as follows:

```
type dset = { mutable parent: int parray; size: int parray }
```

```
let create n =
 let rec init a i =
  if i < n then init (PArray.set a i i) (i + 1)
  else a
 in
 let parent = init (PArray.create n 0) 0 in
 let size = PArray.create n 0 in
 { parent; size }

let find ds x =
 let rec find_aux x =
  let p = PArray.get ds.parent x in
  if p = x then p else
   let parent, rx = find_aux p in
   let parent = PArray.set parent x rx in
   parent, rx
 in
 let parent, rx = find_aux x in
 ds.parent ← parent;
 rx

let union ds x y =
 let rx = find ds x in
 let ry = find ds y in
 if rx = ry then ds else
  let sx = PArray.get ds.size x in
  let sy = PArray.get ds.size y in
  let r1, r2 =
   if sx >= sy then rx, ry else ry, rx
  in
  let parent = PArray.set ds.parent r2 r1 in
  let size = PArray.set ds.size r1 (sx + sy) in
  { parent; size }
```

Figure 2: Implementation of persistent disjoint set data structure

An implementation of the three operations that follows the above description is given in Fig. 2.

Tasks

(a) Define the persistent disjoint set data structure in your verification tool of choice and implement the `create`, `find`, and `union` operations.

(b) Prove that the implementations of the operations are memory-safe / crash-free. That is, you should prove that all pointer accesses are safe and that all calls to persistent array operations satisfy the preconditions needed to guarantee memory safety of these calls (cf. Challenge 3). In particular, all array accesses should be within the bounds of the array.

(c) Prove the functional correctness of the three operations.

(d) Prove that all operations terminate.

```
let find ds x =
 let rec find_aux parent x p =
  if p = x then parent, p
  else
   let pp = PArray.get parent p in
   let parent = PArray.set parent x pp in
   find_aux parent p pp
 in
 let p = PArray.get parent x in
 let parent, rx = find_aux x p in
 ds.parent ← parent;
 rx
```

Figure 3: Implementation of `find` with path splitting

Extra Tasks

An alternative to the path compression optimization is *path splitting*. Here, `find` sets the parent pointer of every node along the path to that of its grandparent. These updates are performed during the upward traversal of the path, yielding a tail-recursive implementation, as shown in Fig. 3. This implementation is still asymptotically optimal [4].

(e) Implement the variant of `find` with path splitting in your verification tool of choice.

(f) Prove that the variant is memory safe / crash safe.

(g) Verify that `find` with path splitting is functionally correct.

(h) Prove that `find` with path splitting always terminates.

References

[1] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In Claudio V. Russo and Derek Dreyer, editors, *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*, pages 37–46. ACM, 2007.

[2] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[3] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[4] Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.