SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

TABLE OF CONTENTS INDEX ROADMAP

BINOM

BINOMIAL QUEUES

Implementation and correctness proof of fast mergeable priority queues using binomial queues.

Operation `empty` is constant time, `insert`, `delete_max`, and `merge` are logN time. (Well, except that comparisons on `nat` take linear time. Read the Extract chapter to see what can be done about that.)

Required Reading

Binomial Queues http://www.cs.princeton.edu/~appel/Binom.pdf by Andrew W. Appel, 2016.

Binomial Queues http://www.cs.princeton.edu/~appel/BQ.pdf Section 9.7 of *Algorithms 3rd Edition in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, and Searching*, by Robert Sedgewick. Addison-Wesley, 2002.

The Program

```
Require Import Perm.
Require Import Priqueue.

Module BinomQueue <: PRIQUEUE.

Definition key := nat.

Inductive tree : Type :=
| Node: key → tree → tree → tree
| Leaf : tree.
```

A priority queue (using the binomial queues data structure) is a list of trees. The `i`'th element of the list is either `Leaf` or it is a power-of-2-heap with exactly `2^i` nodes.

This program will make sense to you if you've read the Sedgewick reading; otherwise it is rather mysterious.

```
Definition priqueue := list tree.

Definition empty : priqueue := nil.

Definition smash (t u: tree) : tree :=
  match t , u with
  | Node x t1 Leaf, Node y u1 Leaf ⇒
                  if x >? y then Node x (Node y u1 t1) Leaf
                               else Node y (Node x t1 u1) Leaf
  | _ , _ ⇒ Leaf (* arbitrary bogus tree *)
  end.

Fixpoint carry (q: list tree) (t: tree) : list tree :=
  match q, t with
  | nil, Leaf ⇒ nil
  | nil, _ ⇒ t :: nil
  | Leaf :: q', _ ⇒ t :: q'
  | u :: q', Leaf ⇒ u :: q'
  | u :: q', _ ⇒ Leaf :: carry q' (smash t u)
 end.

Definition insert (x: key) (q: priqueue) : priqueue :=
     carry q (Node x Leaf Leaf).

Eval compute in fold_left (fun x q ⇒insert q x)
[3;1;4;1;5;9;2;3;5] empty.

  = [Node 5 Leaf Leaf;
     Leaf;
     Leaf;
     Node 9
        (Node 4 (Node 3 (Node 1 Leaf Leaf) (Node 1 Leaf Leaf))
           (Node 3 (Node 2 Leaf Leaf) (Node 5 Leaf Leaf)))
        Leaf]
   : priqueue

Fixpoint join (p q: priqueue) (c: tree) : priqueue :=
  match p, q, c with
    [], _ , _ ⇒ carry q c
  | _, [], _ ⇒ carry p c
  | Leaf::p', Leaf::q', _ ⇒ c :: join p' q' Leaf
  | Leaf::p', q1::q', Leaf ⇒ q1 :: join p' q' Leaf
  | Leaf::p', q1::q', Node _ _ _ ⇒ Leaf :: join p' q' (smash c
q1)
  | p1::p', Leaf::q', Leaf ⇒ p1 :: join p' q' Leaf
  | p1::p', Leaf::q',Node _ _ _ ⇒ Leaf :: join p' q' (smash c
p1)
  | p1::p', q1::q', _ ⇒ c :: join p' q' (smash p1 q1)
 end.
```

```
Fixpoint unzip (t: tree) (cont: priqueue → priqueue) : priqueue
:=
  match t with
  | Node x t₁ t₂ ⇒ unzip t₂ (fun q ⇒ Node x t₁ Leaf :: cont q)
  | Leaf ⇒ cont nil
  end.

Definition heap_delete_max (t: tree) : priqueue :=
  match t with
    Node x t₁ Leaf ⇒ unzip t₁ (fun u ⇒ u)
  | _ ⇒ nil (* bogus value for ill-formed or empty trees *)
  end.

Fixpoint find_max' (current: key) (q: priqueue) : key :=
  match q with
  | [] ⇒ current
  | Leaf::q' ⇒ find_max' current q'
  | Node x _ _ :: q' ⇒ find_max' (if x >? current then x else
current) q'
  end.

Fixpoint find_max (q: priqueue) : option key :=
  match q with
  | [] ⇒ None
  | Leaf::q' ⇒ find_max q'
  | Node x _ _ :: q' ⇒ Some (find_max' x q')
 end.

Fixpoint delete_max_aux (m: key) (p: priqueue) : priqueue *
priqueue :=
  match p with
  | Leaf :: p' ⇒ let (j,k) := delete_max_aux m p' in (Leaf::j,
k)
  | Node x t₁ Leaf :: p' ⇒
       if m >? x
       then (let (j,k) := delete_max_aux m p'
             in (Node x t₁ Leaf::j,k))
       else (Leaf::p', heap_delete_max (Node x t₁ Leaf))
  | _ ⇒ (nil, nil) (* Bogus value *)
  end.

Definition delete_max (q: priqueue) : option (key * priqueue) :=
  match find_max q with
  | None ⇒ None
  | Some m ⇒ let (p',q') := delete_max_aux m q
                             in Some (m, join p' q' Leaf)
  end.

Definition merge (p q: priqueue) := join p q Leaf.
```

Characterization Predicates

`t` is a complete binary tree of depth `n`, with every key <= `m`

```
Fixpoint pow2heap' (n: nat) (m: key) (t: tree) :=
 match n, m, t with
    0, m, Leaf ⇒ True
  | 0, m, Node _ _ _ ⇒ False
  | S _, m,Leaf ⇒ False
  | S n', m, Node k l r ⇒
       m ≥ k ∧ pow2heap' n' k l ∧ pow2heap' n' m r
 end.
```

t is a power-of-2 heap of depth n

```
Definition pow2heap (n: nat) (t: tree) :=
  match t with
    Node m t1 Leaf ⇒ pow2heap' n m t1
  | _ ⇒ False
  end.
```

l is the ith tail of a binomial heap

```
Fixpoint priq' (i: nat) (l: list tree) : Prop :=
   match l with
  | t :: l' ⇒ (t=Leaf ∨ pow2heap i t) ∧ priq' (S i) l'
  | nil ⇒ True
 end.
```

q is a binomial heap

```
Definition priq (q: priqueue) : Prop := priq' 0 q.
```

Proof of Algorithm Correctness

Various Functions Preserve the Representation Invariant

...that is, the priq property, or the closely related property pow2heap.

Exercise: 1 star (empty_priq)

```
Theorem empty_priq: priq empty.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (smash_valid)

```
Theorem smash_valid:
       ∀ n t u, pow2heap n t → pow2heap n u → pow2heap (S n)
(smash t u).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (carry_valid)

```
Theorem carry_valid:
           ∀ n q, priq' n q →
           ∀ t, (t=Leaf ∨ pow2heap n t) → priq' n (carry q t).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, optional (insert_valid)

```
Theorem insert_priq: ∀ x q, priq q → priq (insert x q).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars, optional (join_valid)

```
(* This proof is rather long, but each step is reasonably straightforward.
    There's just one induction to do, right at the beginning. *)
Theorem join_valid: ∀ p q c n, priq' n p → priq' n q → (c=Leaf ∨
pow2heap n c) → priq' n (join p q c).
(* FILL IN HERE *) Admitted.
```

☐

```
Theorem merge_priq: ∀ p q, priq p → priq q → priq (merge p q).
Proof.
 intros. unfold merge. apply join_valid; auto.
Qed.
```

Exercise: 5 stars, optional (delete_max_Some_priq)

```
Theorem delete_max_Some_priq:
      ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
(* FILL IN HERE *) Admitted.
```

☐

The Abstraction Relation

`tree_elems t l` means that the keys in t are the same as the elements of l (with repetition)

```
Inductive tree_elems: tree → list key → Prop :=
| tree_elems_leaf: tree_elems Leaf nil
| tree_elems_node: ∀ bl br v tl tr b,
           tree_elems tl bl →
           tree_elems tr br →
           Permutation b (v::bl++br) →
           tree_elems (Node v tl tr) b.
```

Exercise: 3 stars (priqueue_elems)

Make an inductive definition, similar to `tree_elems`, to relate a priority queue "l" to a list of all its elements.

As you can see in the definition of `tree_elems`, a `tree` relates to *any* permutation of its keys, not just a single permutation. You should make your `priqueue_elems` relation behave similarly, using (basically) the same technique as in `tree_elems`.

```
Inductive priqueue_elems: list tree → list key → Prop :=
             (* FILL IN HERE *)
.
```

☐

```
Definition Abs (p: priqueue) (al: list key) := priqueue_elems p
al.
```

Sanity Checks on the Abstraction Relation

Exercise: 2 stars (tree_elems_ext)

Extensionality theorem for the tree_elems relation

```
Theorem tree_elems_ext: ∀ t e1 e2,
  Permutation e1 e2 → tree_elems t e1 → tree_elems t e2.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (tree_perm)

```
Theorem tree_perm: ∀ t e1 e2,
  tree_elems t e1 → tree_elems t e2 → Permutation e1 e2.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (priqueue_elems_ext)

To prove `priqueue_elems_ext`, you should almost be able to cut-and-paste the proof of `tree_elems_ext`, with just a few edits.

```
Theorem priqueue_elems_ext: ∀ q e1 e2,
  Permutation e1 e2 → priqueue_elems q e1 → priqueue_elems q e2.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (abs_perm)

```
Theorem abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (can_relate)

```
Lemma tree_can_relate: ∀ t, ∃ al, tree_elems t al.
Proof.
(* FILL IN HERE *) Admitted.

Theorem can_relate: ∀ p, priq p → ∃ al, Abs p al.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Various Functions Preserve the Abstraction Relation

Exercise: 1 star (empty_relate)

```
Theorem empty_relate: Abs empty nil.
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (smash_elems)

Warning: This proof is rather long.

```
Theorem smash_elems: ∀ n t u bt bu,
                     pow2heap n t → pow2heap n u →
                     tree_elems t bt → tree_elems u bu →
                     tree_elems (smash t u) (bt ++ bu).
(* FILL IN HERE *) Admitted.
```

☐

Optional Exercises

Some of these proofs are quite long, but they're not especially tricky.

Exercise: 4 stars, optional (carry_elems)

```
Theorem carry_elems:
      ∀ n q, priq' n q →
      ∀ t, (t=Leaf ∨ pow2heap n t) →
      ∀ eq et, priqueue_elems q eq →
                        tree_elems t et →
                        priqueue_elems (carry q t) (eq++et).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, optional (insert_elems)

```
Theorem insert_relate:
        ∀ p al k, priq p → Abs p al → Abs (insert k p) (k::al).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 4 stars, optional (join_elems)

```
Theorem join_elems:
                ∀ p q c n,
                    priq' n p →
                    priq' n q →
                    (c=Leaf ∨ pow2heap n c) →
                  ∀ pe qe ce,
                             priqueue_elems p pe →
                             priqueue_elems q qe →
                             tree_elems c ce →
                              priqueue_elems (join p q c)
(ce++pe++qe).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, optional (merge_relate)

```
Theorem merge_relate:
    ∀ p q pl ql al,
       priq p → priq q →
       Abs p pl → Abs q ql → Abs (merge p q) al →
       Permutation al (pl++ql).
```

```
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 5 stars, optional (delete_max_None_relate)

```
Theorem delete_max_None_relate:
        ∀ p, priq p → (Abs p nil ↔ delete_max p = None).
(* FILL IN HERE *) Admitted.
```

□

Exercise: 5 stars, optional (delete_max_Some_relate)

```
Theorem delete_max_Some_relate:
  ∀ (p q: priqueue) k (pl ql: list key), priq p →
   Abs p pl →
   delete_max p = Some (k,q) →
   Abs q ql →
   Permutation pl (k::ql) ∧ Forall (ge k) ql.
(* FILL IN HERE *) Admitted.
```

□

With the following line, we're done! We have demonstrated that Binomial Queues are a correct implementation of mergeable priority queues. That is, we have exhibited a `Module BinomQueue` that satisfies the `Module Type PRIQUEUE`.

```
End BinomQueue.
```

Measurement.

Exercise: 5 stars, optional (binom_measurement)

Adapt the program (but not necessarily the proof) to use Ocaml integers as keys, in the style shown in Extract. Write an ML program to exercise it with random inputs. Compare the runtime to the implementation from Priqueue, also adapted for Ocaml integers. □