SOFTWARE FOUNDATIONS

VOLUME 3: VERIFIED FUNCTIONAL ALGORITHMS

PRIQUEUE

PRIORITY QUEUES

A *priority queue* is an abstract data type with the following operations:

- `empty: priqueue`
- `insert: key → priqueue → priqueue`
- `delete_max: priqueue → option (key * priqueue)`

The idea is that you can find (and remove) the highest-priority element. Priority queues have applications in:

- Discrete-event simulations: The highest-priority event is the one whose scheduled time is the earliest. Simulating one event causes new events to be scheduled in the future.
- Sorting: *heap sort* puts all the elements in a priority queue, then removes them one at a time.
- Computational geometry: algorithms such as *convex hull* use priority queues.
- Graph algorithms: Dijkstra's algorithm for finding the shortest path uses a priority queue.

We will be considering *mergeable* priority queues, with one additional operator:

- `merge: priqueue → priqueue → priqueue`

The classic data structure for priority queues is the "heap", a balanced binary tree in which the the key at any node is *bigger* than all the keys in nodes below it. With heaps, `empty` is constant time, `insert` and `delete_max` are logN time. But `merge` takes NlogN time, as one must take all the elements out of one queue and insert them into the other queue.

Another way to do priority queues is by *balanced binary search trees* (such as red-black trees); again, `empty` is constant time, `insert` and `delete_max` are logN time, and `merge` takes NlogN time, as one must take all the elements out of one queue and insert them into the other queue.

In the *Binom* chapter we will examine an algorithm in which `empty` is constant time, `insert`, `delete_max`, and `merge` are logN time.

In *this* chapter we will consider a much simpler (and slower) implementation, using unsorted lists, in which:

- `empty` takes constant time
- `insert` takes constant time
- `delete_max` takes linear time
- `merge` takes linear time

Module Signature

This is the "signature" of a correct implementation of priority queues where the keys are natural numbers. Using `nat` for the key type is a bit silly, since the comparison function Nat.ltb takes linear time in the value of the numbers! But you have already seen in the Extract chapter how to define these kinds of algorithms on key types that have efficient comparisons, so in this chapter (and the Binom chapter) we simply won't worry about the time per comparison.

```
Require Import Perm.

Module Type PRIQUEUE.
  Parameter priqueue: Type.
  Definition key := nat.

  Parameter empty: priqueue.
  Parameter insert: key → priqueue → priqueue.
  Parameter delete_max: priqueue → option (key * priqueue).
  Parameter merge: priqueue → priqueue → priqueue.

  Parameter priq: priqueue → Prop.
  Parameter Abs: priqueue → list key → Prop.
  Axiom can_relate: ∀ p, priq p → ∃ al, Abs p al.
  Axiom abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
  Axiom empty_priq: priq empty.
  Axiom empty_relate: Abs empty nil.
  Axiom insert_priq: ∀ k p, priq p → priq (insert k p).
  Axiom insert_relate:
        ∀ p al k, priq p → Abs p al → Abs (insert k p) (k::al).
  Axiom delete_max_None_relate:
        ∀ p, priq p → (Abs p nil ↔ delete_max p = None).
  Axiom delete_max_Some_priq:
      ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
  Axiom 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.
  Axiom merge_priq: ∀ p q, priq p → priq q → priq (merge p q).
  Axiom 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).
End PRIQUEUE.
```

Take some time to consider whether this is the right specification! As always, if we get the specification wrong, then proofs of "correctness" are not so useful.

Implementation

```
Module List_Priqueue <: PRIQUEUE.
```

Now we are responsible for providing `Definitions` of all those `Parameters`, and proving `Theorems` for all those `Axioms`, so that the values in the `Module` match the types in the `Module Type`. If we try to `End List_Priqueue` before everything is provided, we'll get an error. Uncomment the next line and try it!

```
(* End List_Priqueue. *)
```

Some Preliminaries

A copy of the `select` function from Selection.v, but getting the max element instead of the min element:

```
Fixpoint select (i: nat) (l: list nat) : nat * list nat :=
match l with
| nil ⇒ (i, nil)
| h::t ⇒ if i >=? h
               then let (j, l') := select i t in (j, h::l')
               else let (j,l') := select h t in (j, i::l')
end.
```

Exercise: 3 stars (select_perm_and_friends)

```
Lemma select_perm: ∀ i l,
  let (j,r) := select i l in
   Permutation (i::l) (j::r).
Proof.
(* Copy your proof from Selection.v, and change one character. *)
intros i l; revert i.
induction l; intros; simpl in *.
(* FILL IN HERE *) Admitted.

Lemma select_biggest_aux:
  ∀ i al j bl,
    Forall (fun x ⇒ j ≥ x) bl →
    select i al = (j,bl) →
    j ≥ i.
Proof.
(* Copy your proof of select_smallest_aux from Selection.v, and edit. *)
(* FILL IN HERE *) Admitted.

Theorem select_biggest:
  ∀ i al j bl, select i al = (j,bl) →
```

```
     Forall (fun x ⇒ j ≥ x) bl.
Proof.
(* Copy your proof of select_smallest from Selection.v, and edit. *)
intros i al; revert i; induction al; intros; simpl in *.
(* FILL IN HERE *) admit.
bdestruct (i >=? a).
*
destruct (select i al) eqn:?H.
(* FILL IN HERE *) Admitted.
```

☐

The Program

```
Definition key := nat.

Definition priqueue := list key.

Definition empty : priqueue := nil.
Definition insert (k: key)(p: priqueue) := k::p.
Definition delete_max (p: priqueue) :=
  match p with
  | i::p' ⇒ Some (select i p')
  | nil ⇒ None
  end.
Definition merge (p q: priqueue) : priqueue := p++q.
```

Predicates on Priority Queues

The Representation Invariant

In this implementation of priority queues as unsorted lists, the representation invariant is trivial.

```
Definition priq (p: priqueue) := True.
```

The abstraction relation is trivial too.

```
Inductive Abs': priqueue → list key → Prop :=
Abs_intro: ∀ p, Abs' p p.

Definition Abs := Abs'.
```

Sanity Checks on the Abstraction Relation

```
Lemma can_relate : ∀ p, priq p → ∃ al, Abs p al.
Proof.
  intros. ∃ p; constructor.
Qed.
```

When the `Abs` relation says, "priority queue `p` contains elements `al`", it is free to report the elements in any order. It could even relate `p` to two different lists `al` and `bl`, as long as one is a permutation of the other.

```
Lemma abs_perm: ∀ p al bl,
   priq p → Abs p al → Abs p bl → Permutation al bl.
Proof.
intros.
inv H0. inv H1. apply Permutation_refl.
Qed.
```

Characterizations of the Operations on Queues

```
Lemma empty_priq: priq empty.
Proof. constructor. Qed.

Lemma empty_relate: Abs empty nil.
Proof. constructor. Qed.

Lemma insert_priq: ∀ k p, priq p → priq (insert k p).
Proof. intros; constructor. Qed.

Lemma insert_relate:
    ∀ p al k, priq p → Abs p al → Abs (insert k p) (k::al).
Proof. intros. unfold insert. inv H0. constructor. Qed.

Lemma delete_max_Some_priq:
      ∀ p q k, priq p → delete_max p = Some(k,q) → priq q.
Proof. constructor. Qed.
```

Exercise: 2 stars (simple_priq_proofs)

```
(* GRADE_THEOREM 0.5: delete_max_None_relate *)
Lemma delete_max_None_relate:
  ∀ p, priq p →
      (Abs p nil ↔ delete_max p = None).
Proof.
(* FILL IN HERE *) Admitted.

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

Lemma merge_priq:
  ∀ p q, priq p → priq q → priq (merge p q).
Proof. intros. constructor. Qed.

(* GRADE_THEOREM 0.5: delete_max_Some_relate *)
Lemma 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.
```

□

```
End List_Priqueue.
```