Lab Session 3
Skew Heaps

Software Verification

Nova School of Science and Technology
Mário Pereira `mjp.pereira@fct.unl.pt`

Version of September 23, 2025

The goal of this lab session is to derive the proof of several Lemmas that ensure the correctness of a Skew Heap implementation.

1 Correctness Proof of Skew Heaps

Consider the Rocq implementation of Skew Heaps given in the Appendix A.

Question 1. Give a Rocq definition for the `le_root` relation, given by the following rules:

$$\text{(LE\_ROOT\_EMPTY)}\ \frac{x : \texttt{nat}}{\texttt{le\_root}\ x\ \texttt{Empty}} \qquad \frac{x \leq y}{\texttt{le\_root}\ x\ (\texttt{Node}\ l\ y\ r)}\ \text{(LE\_ROOT\_NODE)}$$

□

Question 2. Give a Rocq definition for the `is_heap` relation, given by the following rules:

$$\frac{}{\texttt{is\_heap Empty}}\ \text{(IS\_HEAP\_EMPTY)}$$

$$\frac{\texttt{is\_heap}\ l \qquad \texttt{is\_heap}\ r \qquad \texttt{le\_root}\ x\ l \qquad \texttt{le\_root}\ x\ r}{\texttt{is\_heap}\ (\texttt{Node}\ l\ x\ r)}\ \text{(IS\_HEAP\_NODE)}$$

□

Question 3. Prove the following Lemma:

```
Lemma create_correct: is_heap create.
```
Rocq

□

Question 4. Prove the following Lemma:

```
Lemma merge_correct : ∀ t h1 h2,
  is_heap h1 → is_heap h2 →
  t = (h1, h2) →
  is_heap (merge t).
```
Rocq

□

Question 5. Prove the following Lemma:

```
Lemma add_correct: ∀ x h,                                    Rocq
  is_heap h →
  is_heap (add x h).
```

□

Question 6. Prove the following Lemma:

```
Lemma remove_min_correct: ∀ h,                               Rocq
  is_heap h →
  match remove_min h with
  | None ⇒ h = BinTree.Empty
  | Some h' ⇒ is_heap h'
  end.
```

□

A Rocq Implementation of Skew Heaps

```
Module BinTree.

  Inductive bin_tree : Type :=
  | Empty
  | Node (l: bin_tree) (e: nat) (r: bin_tree).

  Fixpoint size (t: bin_tree) : nat :=
    match t with
    | Empty => 0
    | Node l _ r => 1 + size l + size r
    end.

End BinTree.

Definition heap : Type := BinTree.bin_tree.

Definition create : heap := BinTree.Empty.

Definition size_two (t: heap * heap) : nat :=
  let (h1, h2) := t in
  BinTree.size h1 + BinTree.size h2.

Function merge (t: heap * heap) {measure size_two t} : heap :=
  match t with
  | (BinTree.Empty, t2) => t2
  | (t1, BinTree.Empty) => t1
  | (BinTree.Node l1 x1 r1 as h1, BinTree.Node l2 x2 r2 as h2) =>
      if x1 <=? x2 then
        BinTree.Node (merge (r1, h2)) x1 l1
      else
        BinTree.Node (merge (r2, h1)) x2 l2
  end.
Proof.
  - intros. simpl in *. lia.
  - intros. simpl in *. lia.
Defined.

Definition add (x: nat) (h: heap) : heap :=
  merge ((BinTree.Node BinTree.Empty x BinTree.Empty), h).

Definition remove_min (h: heap) : option heap :=
  match h with
  | BinTree.Empty => None
  | BinTree.Node l _ r => Some (merge (l, r))
  end.
```