**Exercises**

**Exercise 4.1** Height-Preserving In-Order Join

Write a function that joins two binary trees such that

- The in-order traversal of the new tree is the concatenation of the in-order traversals of the original tree
- The new tree is at most one higher than the highest original tree

Hint: Once you got the function right, proofs are easy!

```functional
fun join :: "'a tree ⇒ 'a tree ⇒ 'a tree"
lemma "inorder (join t1 t2) = inorder t1 @ inorder t2"
lemma "height (join t1 t2) ≤ max (height t1) (height t2) + 1"
```

**Exercise 4.2** Enumerate Elements in Interval

Write a function to in-order enumerate all elements of a BST in a given interval. I.e., `in_range t u v` shall enumerate all elements `x` with `u ≤ x ≤ v`. Write a recursive function that does not descend into nodes that definitely contain no elements in the given range.

```functional
fun in_range :: "'a::linorder tree ⇒ 'a ⇒ 'a ⇒ 'a list"
lemma "bst t =⇒ set (in_range t u v) = {x ∈ set_tree t. u ≤ x ∧ x ≤ v}"
lemma "bst t =⇒ in_range t u v = filter (λx. u ≤ x ∧ x ≤ v) (inorder t)"
```

**Exercise 4.3** Pretty Printing of Binary Trees

Define a function that checks whether two binary trees have the same structure. The values at the nodes may differ.

```functional
fun bin_tree2 :: "'a tree ⇒ 'b tree ⇒ bool"
```
While this function itself is not very useful, the induction rule generated by the function package is! It allows simultaneous induction over two trees:

```
print_statement bin_tree2_induct
```

Binary trees can be uniquely pretty-printed by emitting a symbol L for a leaf, and a symbol N for a node. Each N is followed by the pretty-prints of the left and right tree. No additional brackets are required!

```
datatype 'a tchar = L | N 'a

fun pretty :: "'a tree ⇒ 'a tchar list"
```

Show that pretty-printing is actually unique, i.e., no two different trees are pretty-printed the same way. Hint: Auxiliary lemma. Simultaneous induction over both trees.

```
lemma pretty_unique: "pretty t = pretty t' ⇒ t=t'"
```

### Homework 4  Delete Minimum

**Submission until Friday, May 26, 11:59am.**

Define a function to return and delete the minimum element from a non-empty BST. You may omit the equation for the empty tree.

```
fun del_min :: "'a tree ⇒ 'a∗ 'a tree" where
```

Show that your function preserves the search tree property. Hint: An auxiliary lemma of the form \( t \neq \text{Leaf} \implies \text{del_min } t = (x,t') \implies \ldots \) relating set_tree t and set_tree t' may be helpful.

```
lemma "[t\neq Leaf; bst t] ⇒ bst (snd (del_min t))"
```

Show that your function returns the first element of the inorder traversal, and a tree whose inorder traversal corresponds to the tail of the original inorder traversal. Hint: You may need auxiliary lemmas. Some subgoals may require more forceful methods than auto.

```
lemma "t\neq Leaf ⇒ del_min t = (x,t')
      ⇒ case inorder t of (y#ys) ⇒ x=y ∧ inorder t' = ys"
```

Define a function \( \text{del_min2} \) that uses so called continuations: The second argument is a function which is applied to the result. (i.e. the minimum element and the tree of remaining elements)

Do not use \( \text{del_min} \) for the definition, but define \( \text{del_min2} \) recursively, passing down (modified) continuations.
fun del_min2 :: “‘a tree ⇒ (‘a ⇒ ‘a tree ⇒ ‘a tree) ⇒ ‘a tree”

Show that your function definition corresponds to del_min

lemma del_min2,del_min:
“t ≠ Leaf ⇒ del_min2 t f = (let (a,t') = del_min t in f a t')”