In this exercise, you shall develop a trie data structure for keys of type `"a list"` (instead of `"bool list"`).

Thus, a node needs to store a map from `"a"` to the next trie.

In a first step, we encode the map as `@"a" = @"b" option;`

datatype "a trie = Leaf | Node bool "("a:"a trie) trie23"
text "The refined trie datatype:

datatype 'a trie = Leaf | Node bool ("\xax'a trie") trie23"

- Define an invariant for trie and an abstraction function to trie.
- Then define membership, insertion, and deletion, and show that they behave correctly wrt. the abstract trie. Finally, combine the correctness lemmas to get a set interface based on 2-3-trie tries.

fun trie_inv :: "'a:linorder trie ⇒ bool" where
  "trie_inv _ = undefined"

fun trie_u :: "'a:linorder trie ⇒ 'a trie" where
  "trie_u _ = undefined"

fun isin :: "'a:linorder trie ⇒ 'a list ⇒ bool" where
  "isin _ _ = undefined"

---

text "ExerciseSheet(11)(15.9.2018)"

Exercise(Tries with 2-3-trees)

In this exercise, you shall develop a trie data structure for keys of type @"\a list" instead of @"\a list".

Thus, a node needs to store a map from @"\a" to the next trie.

In a first step, we encode the map as @"\a ⇒ 'b option".

datatype 'a trie = Leaf | Node bool ("\a ⇒ 'a trie option")

- Define and prove correct membership, insertion and deletion (without shrinking the trie).

fun isin :: "'a trie ⇒ 'a list ⇒ bool" where
  "isin _ _ = undefined"

---

type 'a list' Instead of @"\a list".

Thus, a node needs to store a map from @"\a" to the next trie.

In a first step, we encode the map as @"\a ⇒ 'b option".

datatype 'a trie = Leaf | Node bool ("\a ⇒ 'a trie option")

text - Define and prove correct membership, insertion and deletion (without shrinking the trie).

fun isin :: "'a trie ⇒ 'a list ⇒ bool" where
  "isin _ _ = undefined"
type 'a list' (instead of #(typ "a list").

Thus, a node needs to store a map from 'a to the next trie.

In a first step, we encode the map as #(typ 'a -> 'b option).

datatype 'a trie = Leaf | Node bool 'a -> 'a trie option

text - Define and prove correct membership, insertion and deletion (without shrinking the trie).

fun isin : 'a trie -> 'a list -> bool
where
  "isin _ = undefined"
fun ins : 'a list -> 'a trie -> 'a trie
where
  "ins _ = undefined"

lemma ins_correct: "isin (ins as t) bs = (as=bs v isin t bs)"

| "isin (Node b m) [] | b" | = b |
| "isin (Node b m) (k:ms) 
  (case m k of None | False | Some t = isin t ks)" |

term "m:k=t1"

term "m:k=Some t"

term "fun upd m k (Some t)"

fun ins : "a list -> 'a trie -> 'a trie
where
  "ins _ = undefined"

lemma ins_correct: "isin (ins as t) bs = (as=bs v isin t bs)"

| if k then r else l = k |

fun insert : bool list -> trie -> trie
where
  insert [] Leaf = Node True (Leaf Leaf) |
  insert [] (Node b r l) = Node True l r |
| insert (k:ms) Leaf = 
  Node False (if k then Leaf, insert ks Leaf) |
| insert (k:ms) (Node b r l) = 
  Node False (if k then Leaf, insert ks Leaf) |
| insert (k:ms) (Node b (l,r)) = 
  Node False (if k then Leaf, insert ks Leaf) |
| Node b (if k then l, insert ks r) |
| else (insert ks l, r) |

lemma insin.insert : "isin (insert as t) bs = (as=bs v isin t bs)"

consts

insert : bool list -> trie -> trie

Found termination order: "(l: size list (and p)) <mlex> (l: size list size (fst p)) <mlex> ()

|
fun ins :: Type a => a list => a trie => a trie

where

| ins [] Leaf = Node True Map.empty |
| ins (Node b m) = Node True (b, m) |

lemma ins_correct: "isin (ins as t) bs = (as\#bs ∨ isin t bs)"
fun ins :: "'a list => 'a trie => 'a trie"
  where
  "ins [] Leaf = Node True Leaf.empty" 
  "ins [] Node True m = Node True m" 
  "ins [k]s Leaf = Node False [k-\ins ks Leaf]" 
  "ins [k]s (Node b m) = 
    case b of 
      None => Node b (mk-\ins ks Leaf) 
      Some t => Node b (mk-\ins ks t)"

lemma ins_correct: "\isn (ins as t) bs = (as\bs \ \isin t bs)"

proof (prove)
  goal (1 subgoal):
  1. ex10.tmpl.\isn (ex10_tmpl.\isn as t) bs = (as = bs \ ex10_tmpl.\isin t bs)"
case n k of 
| Some t => node b (mk' - ins ks Leaf)

lemma ins_correct: "isin (ins as t) bs = (as' = bs ∨ ins t bs)"
proof (prove)

goal (1 subgoal):
1. ex10_tpl.isin (ex10_tpl.isn as t) bs = (as = bs ∨ ex10_tpl.isin t bs)

proof (prove)

goal (2 subgoals):
1. isin [Node Leaf empty] bs = [[]] = bs
2. isin (Node Leaf bs) = [[]] = bs ∨ isin (Node Leaf bs)
3. ∀ k. isin [ex10_tpl.isn ks Leaf] bs = (ks = bs ∨ isin Leaf bs)

proof (prove)

goal (6 subgoals):
1. isin (ex10_tpl.isn [Leaf]) bs = [[]] = bs ∨ isin Leaf bs
2. isin (Node Leaf bs) = [[]] = bs ∨ isin (Node Leaf bs)
3. isin (Node Leaf bs) = [[]] = bs ∨ isin (Node Leaf bs)
lemma Isin_correct: "isin (ins as t) bs = (as @ bs \& isin t bs)"
apply (induction as \x t arbitrary: bs rules: Isin_induct)
apply (auto splitI \x)
done

proof (prove)
goal (4 subgoals):
1. \(\forall s. case bs of [] \Rightarrow \text{True} | k \# \text{ks} \Rightarrow \text{case Name of Name \Rightarrow \text{False} | some t \Rightarrow isin t \text{ks} \Rightarrow \text{[] = bs}
2. \(\forall b. isin (exit,\text{tmpl.ins} [] (Node b m)) \Rightarrow \text{[\text{[]} = bs \lor isin (Node b m) \Rightarrow bs)
3. \(\forall k k s. bs.
4. \(\forall k k s. bs.
5. \text{lemma "isin (delete as t) bs = (as \& bs \lor isin t bs)"}
apply (induction as \x t arbitrary: bs rules: delete.induct)

consts
delete :: "bool list \Rightarrow trie \Rightarrow trie"
Found termination order: "(op, size \& (and \& p)) <\max> (\)"
then show "bij_betw F A A" using  'bij_betw_def[set f "A \cup (\emptyset)" A] by blast

subsection "Function Updating:

definition fun_upd \( f \cdot \{ \langle a, b \rangle \} \cdot a \mapsto b \) where "fun_upd \( f \cdot a \cdot b \) = (\lambda x. If x = a then b else f x)"

proof (prove)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

```proof (prove)

  goal [1 subgoal]:
  1. isin (elt0_impl.delete as t) bs = (as \neq bs \land isin t bs)

  apply (induction as t arbitrary bs rules: delete.induct)
  apply (auto split: list.split option.split)
  done
```

lemma delete_correct: "isin (delete as t) bs = (as \neq bs \land isin t bs)"

apply (induction as t arbitrary bs rules: delete.induct)
apply (auto split: list.split option.split)

text - Now refine the trie data structure to use 2-3 trees for the map.

Note: To make the provided interface more usable, we introduce some abbreviations here:

abbreviation "empty23" = Trie23.Leaf
abbreviation "inv23 t = bal t \ sorted1 (inorder t)"

lemma map23_thens = map_empty_map_update_map_delete
  invar_empty invar_update invar_delete

lemma theorems:
  lookup \ = \ Map.empty
  bal t \ sorted1 (inorder t) \ = \ lookup (update t 0 0 t) \ = \ lookup (update t 0 0 t) \ = \ lookup (update t 0 0 t)
  bal t \ sorted1 (inorder t) \ = \ lookup (Trie23.Map.delete 0 0 t) \ = \ lookup (Trie23.Map.delete 0 0 t) \ = \ (lookup (Trie23.Map.delete 0 0 t) (\ x := \ None))
  bal t \ sorted1 (inorder t) \ = \ bal (update t 0 0 t) \ = \ sorted1 (inorder (update t 0 0 t))
  bal t \ sorted1 (inorder t) \ = \ bal (Trie23.Map.delete 0 0 t) \ = \ sorted1 (inorder (Trie23.Map.delete 0 0 t))

data type a trie' = Leaf | Node `bool a trie' (a trie) trie23

data type a trie = Trie23 | Node `bool a trie' (a trie) trie23

lemma map23_thens:
  lookup empty23 = Map.empty
  inv23 t \ = \ lookup (update t 0 0 t) \ = \ lookup (update t 0 0 t) \ = \ (lookup (update t 0 0 t) (\ x := \ None))
  inv23 empty23
  inv23 t \ = \ inv23 (update t 0 0 t) \ = \ inv23 (Trie23.Map.delete 0 0 t)
Theorem map32_theo:
  \[
  \begin{align*}
  \text{lookup empty32} &= \text{Map.empty} \\
  \text{inv23 \ h} &= \text{lookup (update \ h \ h' \ \text{empty32})} \\
  \text{inv23 \ h} &= \text{lookup (update \ h \ h' \ \text{empty32})} \\
  \text{inv23 \ h} &= \text{empty32} \\
  \text{inv23 \ h} &= \text{lookup (update \ h \ h' \ \text{empty32})} \\
  \end{align*}
  \]

The Refined Trie Datatype:

```plaintext
datatype 'a trie = Leaf | Node (bool * 'a trie) trie23
```

Define an invariant for trie' and an abstraction function to trie'.
Then define membership, insertion, and deletion, and show that they behave correctly wrt. the abstract trie'. Finally, combine the correctness lemmas to get a set interface based on 2-3-tree tries.
It is meant to store keys of the same length only. Thus, the Node constructor stores inner nodes, and there are two types of leaves, LeafF if this path is not in the set, and LeafT if it is in the set.

Define an invariant is_true N that states that all keys in N have length  \( N \) and that there are no supernfluous nodes, i.e., no nodes of the form Node (LeafF, LeafF).

fun is_true = "not ⇒ true ⇒ true"  

Hint: The following should evaluate to true!

value "is_true 3 LeafF"  
value "is_true 2 (Node (LeafF, Node (LeafF, LeafF)))"  

Whereas these should be false

value "is_true 3 LeafF" — Wrong key length  
value "is_true 2 (Node (LeafF, Node (LeafF, LeafF)))" — Wrong key length  
value "is_true 1 (Node (LeafF, Node (LeafF, LeafF)))" — Supernfluous node  

Define membership, insert, and delete functions, and prove them correct!

fun mem :: "true ⇒ bool list ⇒ bool"  
fun isin :: "bool list ⇒ true ⇒ true"  

Define membership, insert, and delete functions, and prove them correct!

value "is_true 3 LeafF" — Wrong key length  
value "is_true 2 (Node (LeafF, Node (LeafF, LeafF)))" — Wrong key length  
value "is_true 1 (Node (LeafF, Node (LeafF, LeafF)))" — Supernfluous node  

Define membership, insert, and delete functions, and prove them correct!

fun mem :: "true ⇒ bool list ⇒ bool"  
fun isin :: "bool list ⇒ true ⇒ true"  

Note that Booleans are ordered by False < True, and that we imported List.Record, which defines a lexicographic ordering on lists, if the elements are ordered.

value "[True, True, False] < [True, True, True, True]"  

**Homework 10.3 Be Original!**

**Submission until Friday, 13. 7. 2017, 11:59am.**

Develop a nice Isabelle formalization yourself!

- This homework goes in parallel to other homeworks for the rest of the lecture period. From next sheet on, we will reduce regular homework load a bit, such that you have a time-frame of 3 weeks with reduced regular homework load.
Homework 10.3 Be Original!

Submission until Friday, 13. 7. 2017, 11:59pm.
Develop a nice Isabelle formalization yourself!

- This homework goes in parallel to other homeworks for the rest of the lecture period. From next sheet on, we will reduce regular homework load a bit, such that you have a time-frame of 3 weeks with reduced regular homework load.

- Document your solution, such that it is clear what you have formalized and what your main theorems state!
- Set yourself a time frame and some intermediate/minimal goals. Your formalization needs not be universal and complete after 3 weeks.
- You are welcome to discuss the realizability of your project with the tutor!
- In case you should need inspiration to find a project: Sparse matrices, skew binary numbers, arbitrary precision arithmetic (on lists of bits), interval data structures (e.g. interval lists), spatial data structures (quad-trees, oct-trees), Fibonacci heaps, prefix tries/arrays and BWT, etc.

- This homework will yield 15 points (for minimal solutions). Additionally, up to 15 bonus points may be awarded for particularly nice/original/etc solutions.
- You may develop a formalization from all areas, not only functional data structures.
- Document your solution, such that it is clear what you have formalized and what your main theorems state!
- Set yourself a time frame and some intermediate/minimal goals. Your formalization needs not be universal and complete after 3 weeks.
- You are welcome to discuss the realizability of your project with the tutor!