Technische Universität München
Institut für Informatik
Prof. Tobias Nipkow, Ph.D.
Fabian Huch

SS 2022
24. 6. 2022

Functional Data Structures

Exercise Sheet 8

Exercise 8.1 Joining 2-3-Trees

Implement and prove correct a function to combine two 2-3-trees of equal height, such that the inorder traversal of the resulting tree is the concatenation of the inorder traversal of the arguments, and the height of the result is either the height of the arguments, or has increased by one. Use *'a upI* to return the result, similar to *Tree23_Set.ins*:

```
fun join :: "'a tree23 ⇒ 'a tree23 ⇒ 'a upI"

lemma join_inorder:
  assumes "height t1 = height t2"
    and "complete t1" "complete t2"
   shows "inorder (treeI (join t1 t2)) = (inorder t1 @ inorder t2)"

lemma join_complete:
  assumes "height t1 = height t2"
    and "complete t1" "complete t2"
   shows "complete (treeI (join t1 t2)) ∧ hI (join t1 t2) = height t2"
```

Hints:

- Try to use automatic case splitting (*auto split*: ...) instead of explicit case splitting via Isar (There will be dozens of cases).
- To find bugs in your join function, or isolate the case where your automatic proof does not (yet) work, use Isar to perform the induction proof case by case.

Exercise 8.2 Bounding the Size of 2-3-Trees

Show that for 2-3-trees, we have:

$$log_3 (s(t) + 1) \leq h(t) \leq log_2 (s(t) + 1)$$

Hint: It helps to first raise the two sides of the inequation to the 2nd/3rd power. Use sledgehammer and find-theorems to search for the appropriate lemmas.

```
lemma height_bound_upper: "complete t ⟹ height t ≤ log 2 (size t + 1)"
```

lemma *height_bound_lower*:
  assumes *"complete t"*
  shows *"log 3 (size t + 1) ≤ height t"*

Homework 8.1 Deletion from a disjoint interval tree

Submission until Thursday, June 30, 23:59pm.

An interval tree is a tree whose nodes each contain an interval of elements from an ordered type. We can define its type as follows:

datatype *'a itree = iLeaf | iNode ('a itree) ('a × 'a) ('a itree)*

The following are two useful functions for interval trees: one returning the set of intervals in the tree, and another returning the set of elements in the tree.

set_itree2 iLeaf = {}

set_itree2 (iNode l (low, high) r) = {low..high} ∪ (set_itree2 l ∪ set_itree2 r)

set_itree3 iLeaf = {}

set_itree3 (iNode l (low, high) r) = {(low, high)} ∪ (set_itree3 l ∪ set_itree3 r)

An ordered disjoint interval tree is an interval tree such that:

- The lower end of an interval in a node is strictly greater than the higher end of every interval in the left subtree.
- The higher end of an interval in a node is strictly smaller than the lower end in every interval in the right subtree.
- The interval in every node has a lower end that is smaller than or equal than its upper end.

Recursively define an invariant for an interval tree that formalises the above conditions.

fun *ibst* :: *"'a::linorder itree ⇒ bool"*

Define and verify a delete function for interval trees. That function should: i) only take an element (i.e. not an interval) and delete it from the tree, ii) exploit the fact that the tree is ordered, and iii) be implemented using an appropriate join function for interval trees.

fun *delete* :: *"int ⇒ int itree ⇒ int itree"*

Hint: this function has to deal with three cases.

- if the element is equal to the two ends of an interval, in which case the interval should be completely removed from the tree,
- if the element is equal to one of the ends of an interval, in which case the interval has to be appropriately shrinked, and

- if the element lies with an interval, in which case the interval as to be split into two. One way to deal with that situation is to let the left subinterval inherit the position of the original interval, and position the right subinterval to be the left most leaf in the right subtree.

value *"delete 3 (iNode (iNode (iNode iLeaf (Interval 0 0) iLeaf) (Interval 1 1) (iNode iLeaf (Interval 2 4) iLeaf)) (Interval (5::nat) 6) (iLeaf))*
= iNode (iNode (iNode iLeaf (Interval 0 0) iLeaf) (Interval 1 1) (iNode iLeaf (Interval 2 2) (iNode iLeaf (Interval 4 4) iLeaf))) (Interval 5 6) iLeaf"

Prove that the function removes the correct element from the tree.

lemma *delete_set_minus*: *"ibst t $\Longrightarrow$ set_itree2 (delete x t) = (set_itree2 t) − {x}"*

Prove that the resulting interval tree conforms to the invariant. Hint: you might want to define a function that returns an list of intervals in an interval tree and a predicate characterising the sortedness of that list. Proving that *delete* preserves the invariant should reduce to arguing about the sortedness of this list.

lemma *ibst_delete*: *"ibst t $\Longrightarrow$ ibst (delete x t)"*

Homework 8.2 Insertion 1-2 Trees

Submission until Thursday, June 30, 23:59pm.

Similar to a 2-3 tree, we can construct search trees that consist of 1- and and 2-nodes and maintains completeness. To achieve a logarithmic height, 1-nodes may not be chained, i.e. the full invariant is::

invar $\langle\rangle$ = *True*

invar $\langle t\rangle$ = (*case t of* $\langle\rangle \Rightarrow$ *True* | $\langle x\rangle \Rightarrow$ *False* | $\langle l, x, r\rangle \Rightarrow$ *height l = height r* $\wedge$ *invar l* $\wedge$ *invar r*)

invar $\langle l, uu, r\rangle$ = (*height l = height r* $\wedge$ *invar l* $\wedge$ *invar r*)

Define an insert function, similar to 2-3 trees (start by copying that function). Instead of the three-node constructor, use an auxiliary *merge* function, which may be recursive again:

fun *merge* :: *"'a tree12 ⇒ 'a ⇒ 'a tree12 ⇒ 'a ⇒ 'a tree12 ⇒ 'a upI"*
fun *ins* :: *"'a::linorder ⇒ 'a tree12 ⇒ 'a upI"*

Show that merge retains the inorder:

lemma *inorder_merge[simp]*:
"inorder(treeI(merge l a m b r)) = (inorder l) @ a # (inorder m) @ b # (inorder r)"

Homework 8.3 Insertion 1-2 Trees II (Bonus))

Submission until Thursday, June 30, 23:59pm.

This is a bonus exercise worth 4 points.

Show that insert retains the invariant:

theorem *invar_ins*: “$invar\ t \Longrightarrow invar\ (treeI(ins\ x\ t)) \wedge hI\ (ins\ x\ t) = height\ t$”