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

SS 2022
8. 7. 2022

Functional Data Structures

Exercise Sheet 10

Exercise 10.1 Tries with 2-3-trees

In the lecture, tries stored child nodes with an abstract map. We want to 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 ≡ Leaf"*
abbreviation *"inv23 t ≡ complete t ∧ sorted1 (inorder t)"*

The refined trie datatype

datatype *'a trie' = Nd' bool "('a×'a trie') tree23"*

Define an invariant for trie' and an abstraction function to trie. Based on the original tries, 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.

You will need a lemma like the following for termination:

lemma *lookup_size_aux*[*termination_simp*]:
  "lookup m k = Some v ⟹ size (v::'a trie') < Suc (size_tree23 (λx. Suc (size (snd x))) m)"

fun *trie'_inv* :: *"'a::linorder trie' ⇒ bool"*
fun *trie'_α* :: *"'a::linorder trie' ⇒ 'a trie"*
definition *empty'* :: *"'a trie'"* **where**
[*simp*]: *"empty' = Nd' False empty23"*

fun *isin'* :: *"'a::linorder trie' ⇒ 'a list ⇒ bool"*
fun *insert'* :: *"'a::linorder list ⇒ 'a trie' ⇒ 'a trie'"*
fun *delete'* :: *"'a::linorder list ⇒ 'a trie' ⇒ 'a trie'"*
definition *set'* :: *"'a::linorder trie' ⇒ 'a list set"* **where**
[*simp*]: *"set' t = set (trie'_α t)"*

lemmas *map23_thms*[*simp*] = *M.map_empty Tree23_Map.M.map_update Tree23_Map.M.map_delete*
  Tree23_Map.M.invar_empty Tree23_Map.M.invar_update Tree23_Map.M.invar_delete
  M.invar_def M.inorder_update M.inorder_inv_update sorted_upd_list

interpretation *S'*: *Set*
where *empty = empty'* **and** *isin = isin'* **and** *insert = insert'* **and** *delete = delete'*
and *set = set'* **and** *invar = trie'_inv*

Exercise 10.2 Bootstrapping a Priority Queue

Given a generic priority queue implementation with $O(1)$ *empty*, *is_empty* operations, $O(f_1\ n)$ insert, and $O(f_2\ n)$ *get_min* and *del_min* operations.

Derive an implementation with $O(1)$ *get_min*, and the asymptotic complexities of the other operations unchanged!

Hint: Store the current minimal element! As you know nothing about f_1 and f_2, you must not use *get_min*/*del_min* in your new *insert* operation, and vice versa!

For technical reasons, you have to define the new implementations type outside the locale!

```
datatype ('a,'s) bs_pq =

locale Bs_Priority_Queue =
  orig: Priority_Queue where
    empty = orig_empty and
    is_empty = orig_is_empty and
    insert = orig_insert and
    get_min = orig_get_min and
    del_min = orig_del_min and
    invar = orig_invar and
    mset = orig_mset
  for orig_empty orig_is_empty orig_insert orig_get_min orig_del_min orig_invar
  and orig_mset :: "'s ⇒ 'a::linorder multiset"
begin
```

In here, the original implementation is available with the prefix *orig*, e.g.

```
  term orig_empty term orig_invar
  thm orig.invar_empty

  definition empty :: "('a,'s) bs_pq"
  fun is_empty :: "('a,'s) bs_pq ⇒ bool"
  fun insert :: "'a ⇒ ('a,'s) bs_pq ⇒ ('a,'s) bs_pq"
  fun get_min :: "('a,'s) bs_pq ⇒ 'a"
  fun del_min :: "('a,'s) bs_pq ⇒ ('a,'s) bs_pq"
  fun invar :: "('a,'s) bs_pq ⇒ bool"
  fun mset :: "('a,'s) bs_pq ⇒ 'a multiset"
  lemmas [simp] = orig.is_empty orig.mset_get_min orig.mset_del_min
    orig.mset_insert orig.mset_empty
    orig.invar_empty orig.invar_insert orig.invar_del_min
```

Show that your new implementation satisfies the priority queue interface!

```
  sublocale Priority_Queue
    where empty = empty
    and is_empty = is_empty
    and insert = insert
    and get_min = get_min
    and del_min = del_min
```

and *invar* = *invar*
and *mset* = *mset*
apply *unfold_locales*
proof *goal_cases*

Homework 10.1 Constructing an lheap from a List of Elements

Submission until Thursday, 14. 7. 2022, 23:59pm.

The naive solution of starting with the empty heap and inserting the elements one by one can be improved by repeatedly merging heaps of roughly equal size. Start by turning the list of elements into a list of singleton heaps. Now make repeated passes over the list, merging adjacent pairs of heaps in each pass (thus halving the list length) until only a single heap is left. It can be shown that this takes linear time.

Define a function *heap_of_list* :: *'a list* ⇒ *'a lheap* and prove its functional correctness.

Start with a function to merge pairs of adjacent heaps, and show that it halves the length:

fun *merge_adjacent* :: *"'a::ord lheap list ⇒ 'a lheap list"*
lemma *length_merge_adjacent*[*simp*]: *"length (merge_adjacent ts) = (length ts + 1) div 2"*

Then define a function to merge a list of *lheap*s, and use it for the final definition:

fun *merge_forest* :: *"'a::ord lheap list ⇒ 'a lheap"*
definition *heap_of_list* :: *"'a::ord list ⇒ 'a lheap"*
lemma *mset_heap_of_list*: *"mset_tree (heap_of_list xs) = mset xs"*
lemma *heap_heap_of_list*: *"heap (heap_of_list xs)"*
lemma *ltree_ltree_of_list*: *"ltree (heap_of_list xs)"*

Homework 10.2 Be Original!

Submission until Thursday, 21. 7. 2022, 23:59pm.

Develop a nice Isabelle formalisation yourself!

- This homework goes in parallel this week and exclusively next week. You should choose your topic until next week, and have some key concepts formalized.
- The homework will yield 15 points (for minimal solutions). Additionally, up to 10 bonus points may be awarded for particularly nice/original/etc solutions.
- You may develop a formalisation from all areas, not only functional data structures.
- Document your solution well, such that it is clear what you have formalised and what your main theorems state!
- Set yourself a time frame and some intermediate/minimal goals. Your formalisation needs not be universal and complete.

- You are encouraged to discuss the realisability of your project with us!
- If you can't think of a topic on your own, here are a few suggestions (though they will score lower on creativity): 1-2 trees (deletion, height), sparse matrices, skew binary numbers, arbitrary precision arithmetic (on lists of bits), interval lists (extension to those of the tutorial), spatial data structures (quad-trees, oct-trees), Fibonacci heaps, etc.
- Use the submission system to submit your (temporary) result as a single Isabelle theory. This submission does not need to (and in fact, won't be able to) get the status "passed".