EPFL

Formal verification of Treaps

Anton Paramonov &
Giorgio Seguini

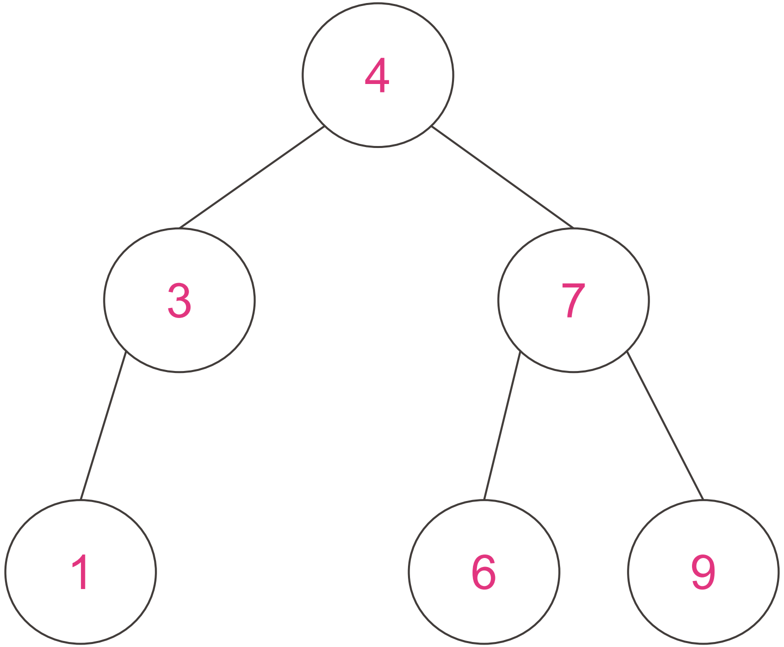
Fall semester 2022

Preliminaries

Binary search tree
Heap
Treap
Isabelle

Binary search tree (BST)

Binary search tree

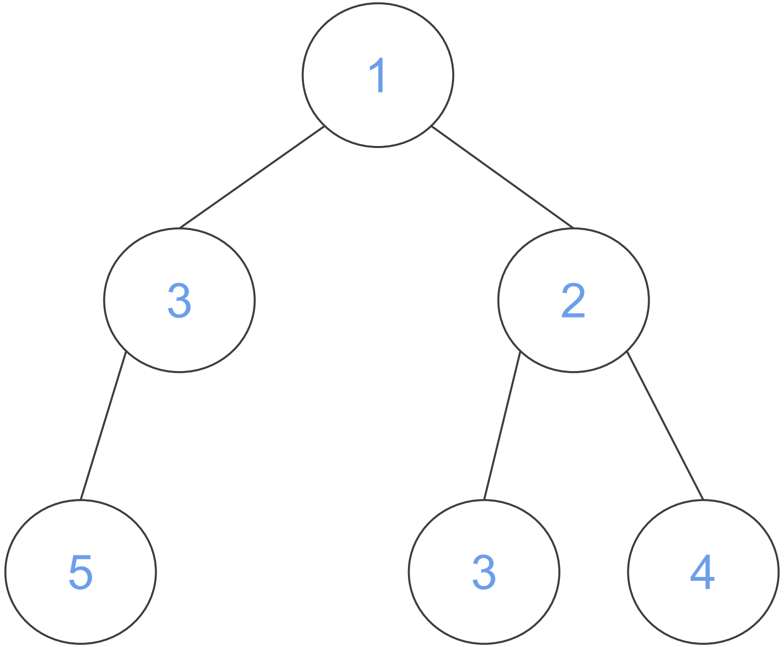


- Put(x)
- Delete(x)
- Search(x)

Complexity Θ(height)

Heap

Heap



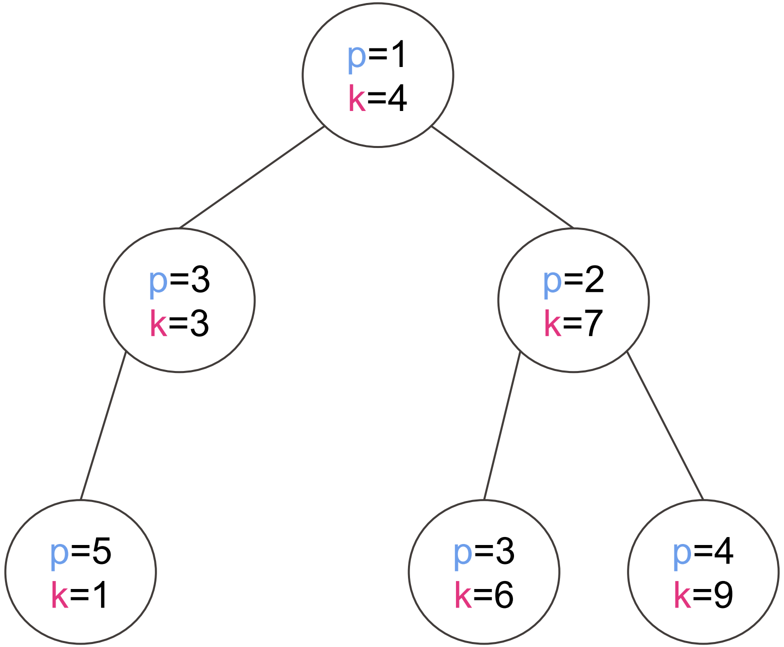
- Put(x)
- Pop() → min

Complexity Θ(height)

Treap

Anton Paramonov & Giorgio Seguini

Treap



A *treap* is a binary tree in which every node contains both a key element and an associated priority (a real number). A treap must be a BST w. r. t. the keys elements and a heap w. r. t. the priorities.

- Put(k, p)
- Delete(k, p)
- Search(k, p)
- Pop() → (k, min_p)

Anton Paramonov & Giorgio Seguini

Anton Paramonov & Giorgio Seguini

BST from Treap

BST from Treap

$$\text{Put}_{BST}(k) = \text{Put}_{Treap}(k, \text{random}())$$

$$\text{Delete}_{BST}(k) = \text{Delete}_{Treap}(k)$$

don't need priority

Remember, we care about the height

$$E[\textit{height}(\text{BST from Treap})] = \Theta(\ln n)$$

Anton Paramonov & Giorgio Seguini

Isabelle

Isabelle

- Higher-order logic theorem proving environment
- Datatypes, inductive definitions and recursive functions with complex pattern matching
- Large theory library: elementary number theory, analysis, algebra, set theory…
- Proof text naturally understandable for both humans *and* computers

Previous work

Verified Analysis of Random Binary Tree Structures

Manuel Eberl, Max W. Haslbeck & Tobias Nipkow (2020)

Quick overview

- Probability theory in Isabelle
- Quicksort
- Random Binary Search Trees
- Treaps

What was already done

- Treaps

```
definition treap :: "('k::linorder * 'p::linorder) tree ⇒ bool" where
"treap t = (bst (map_tree fst t) ∧ heap (map_tree snd t))"
```

- Insertion function definition

```
fun ins :: "'k::linorder ⇒ 'p::linorder ⇒ ('k × 'p) tree
  ⇒ ('k × 'p) tree" where
```

They left it to us

change the tree at all, so we can just drop any duplicates from the list without changing the result. Similarly, the uniqueness property of treaps means that after deleting an element, the resulting treap is exactly the same as if the element had never been inserted in the first place, so even though we only analyse the case of insertions without duplicates, this extends to any sequence of insertion and deletion operations (although we do not show this explicitly).

The main result shall be that after inserting a certain number of distinct elements into the treap and then forgetting their priorities, we get a BST that is distributed identically to a random BST with the same elements, i.e. the treap behaves as if we had inserted the elements

Anton Paramonov & Giorgio Seguini

Our work

Insert analysis
Delete analysis

Insert Analysis

Anton Paramonov & Giorgio Seguini

Cont predicate

```
fun cont:: "'k ⇒ 'p ⇒ ('k × 'p) tree ⇒ bool" where
"cont k p ⟨⟩ = False" |
"cont k p ⟨l, (k1, p1), r⟩ = ((k = k1 ∧ p = p1) ∨ (cont k p l) ∨ (cont k p r))"
```

Anton Paramonov & Giorgio Seguini

Cont results

```
lemma ins_cont:
  assumes  "treap t"
  shows "cont k p (ins k p t) ∨ k ∈ keys t"
```

```
lemma cont_ins_same: "⟦treap t; cont k p t⟧ ⟹ ins k p t = t"
```

Interesting subtleties

Anton Paramonov & Giorgio Seguini

```
lemma cont_ins_same: "⟦treap t; cont k p t⟧ ⟹ ins k p t = t"
```

- Proof is done by the induction on *ins* function

```
fun ins :: "'k::linorder ⇒ 'p::linorder ⇒ ('k × 'p) tree ⇒ ('k × 'p) tree" where
"ins k p Leaf = ⟨Leaf, (k,p), Leaf⟩" |
"ins k p ⟨l, (k1,p1), r⟩ =
```

- Proof is split into cases on relation between k and the root key

- Proof is split into cases on relation between k and the root key

```
obtain l2 k2 p2 r2 where get_p2: "ins k p r = ⟨l2, (k2, p2), r2⟩"
  by (metis ins_neq_Leaf neq_Leaf_iff prod.collapse)
```

Delete Analysis

Merge function

```
fun merge :: "('k::linorder × 'p::linorder) tree ⇒ ('k × 'p) tree
  ⇒ ('k × 'p) tree" where
"merge t Leaf = t" |
"merge Leaf t = t" |
"merge ⟨l1, (k1,p1), r1⟩ ⟨l2, (k2, p2), r2⟩ =
 (if p1 < p2 then
    ⟨l1, (k1,p1), merge r1 ⟨l2, (k2, p2), r2 ⟩⟩
  else
     ⟨merge ⟨l1, (k1,p1), r1⟩ l2, (k2, p2), r2⟩)
"
```

```
lemma merge_treap:
  "⟦treap l; treap r ;(∀k' ∈ keys l. ∀k'' ∈ keys r. k' < k'') ⟧
   ⟹ treap (merge l r)"
```

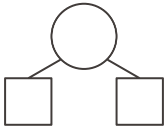

Anton Paramonov & Giorgio Seguini

EPFL

Delete function

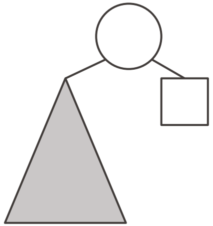
```
fun del:: "'k::linorder ⇒  ('k ×'p::linorder) tree ⇒ ('k × 'p) tree" where
"del k Leaf = Leaf" |
```

```
"del k ⟨Leaf, (k1,p1), Leaf⟩ =
(if k = k1 then Leaf
else ⟨Leaf, (k1,p1), Leaf⟩)" |
```

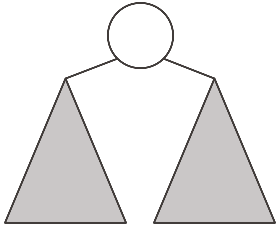


```
"del k ⟨l1, (k1,p1), Leaf⟩  =
(if k = k1 then del k l1
else ⟨del k l1, (k1,p1), Leaf⟩)" |
```

```
"del k ⟨Leaf, (k1,p1), r1⟩  =
(if k = k1 then del k r1
else ⟨Leaf, (k1,p1), del k r1⟩)" |
```



```
"del k ⟨l1, (k1,p1), r1⟩  =
(if k = k1 then
merge (del k l1) (del k r1)
else ⟨del k l1, (k1,p1), del k r1⟩)
"
```



Anton Paramonov & Giorgio Seguini

Delete correctness

Anton Paramonov & Giorgio Seguini

```
lemma treap_del:
"⟦treap t ⟧ ⟹  k ∉ keys (del k t)"
```

```
lemma treap_del2:
"⟦treap t ⟧ ⟹ (keys (del k t)) =  keys t - {k}"
```

```
lemma treap_del3:
"⟦treap t ⟧ ⟹  (prios (del k t)) ⊆  prios t"
```

```
lemma treap_del4:
"⟦treap t ⟧ ⟹  treap (del k t)"
```

```
lemma treap_del5:
"⟦treap t; k ∉ keys t ⟧ ⟹  t = (del k t)"
```

Towards Delete correctness

Anton Paramonov & Giorgio Seguini

```
lemma treap_union:
  assumes   "treap l"  "treap r"
    "∀kl ∈ keys l. kl < k"       "∀kr ∈ keys  r. k < kr"
    "∀pl ∈ prios l. p ≤ pl"       "∀pr ∈ prios r. p ≤ pr"
  shows "treap  ⟨l, (k, p), r⟩"
```

```
lemma merge_treap_key_preserve:
" keys (merge t1 t2) = keys  t1 ∪ keys t2"
```

```
lemma merge_treap_prios_preserve:
" prios (merge t1 t2) = prios  t1 ∪ prios t2"
```

Interesting subtleties

```
lemma treap_del4:
"⟦treap t ⟧ ⟹  treap (del k t)"
```

- Prove by induction - consider the case:

```
"del k ⟨l1, (k1,p1), Leaf⟩
```

- Split into two cases
- First case trivial

```
"treap_l = del k ⟨l1_l, l1_k, l1_r⟩"
```

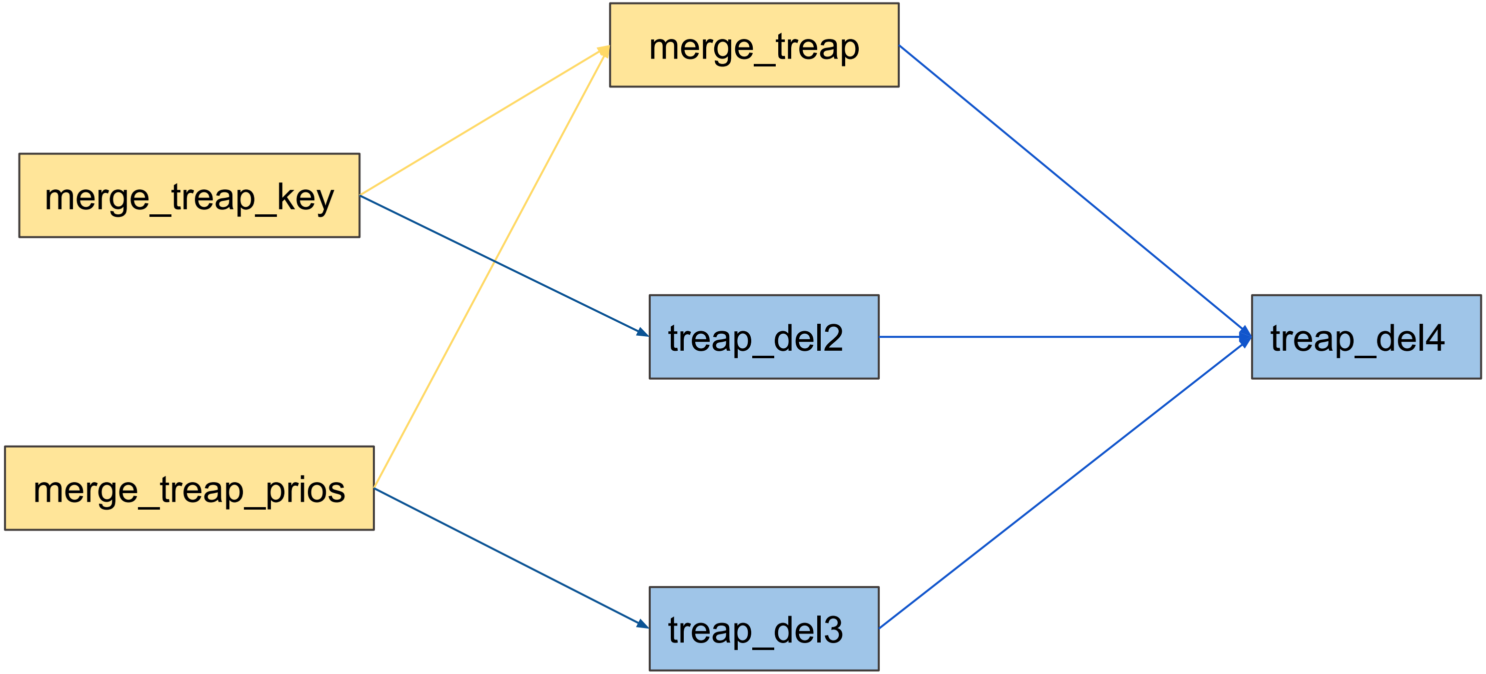
- Second case:

```
treap_del2[of l1 k]
```

```
have keys_ok: "∀k'∈keys treap_l. k' < k1"  by auto
```

```
treap_union[of treap_l Leaf k1 p1]
```

Delete lemmas usage



Thank you!

We verified formal properties of treaps using Isabelle

Duplicate insertion

Element deletion