Optimal Binary Search Trees

Tobias Nipkow and Dániel Somogyi
Technical University Munich

March 17, 2025

Abstract

This article formalizes recursive algorithms for the construction of optimal binary search trees given fixed access frequencies. We follow Knuth [1], Yao [4] and Mehlhorn [2].

The algorithms are memoized with the help of an AFP entry for memoization [3], thus yielding dynamic programming algorithms.

Contents

1 Introduction

These theories formalize algorithms for the construction of optimal binary search trees from fixed access frequencies for a fixed list of items. The work

is based on the original article by Knuth [1] and the textbook by Mehlhorn [2, Part III, Chapter 4].

Initially the algorithms are expressed as naive recursive functions and have exponential complexity. Nevertheless we already refer to them as the cubic (Section 3) and the quadratic algorithm (Section 5), their running times of their fully memoized dynamic programming versions. In Section 7 the algorithms are memoized with the help of an existing framework [3].

1.1 Data Representation

Instead of labeling our BSTs with (ascending) keys $x_i < \ldots < x_j$ we label them with the indices of the actual keys, some interval of integers. Functions taking two integer arguments i and j construct or analyze trees such that $inorder\ t = [i..j]$.

The access frequencies are given by two tables (functions) a and b:

$a\,k$ $(i \le k \le j+1)$ is the frequency of (failing) searches with a key in the interval (x_{k-1}, x_k).

$b\,k$ $(i \le k \le j)$ is the frequency of (successful) searches with key x_k.

2 Weighted Path Length of BST

theory *Weighted_Path_Length*
imports *HOL−Library.Tree*
begin

This theory presents two definitions of the *weighted path length* of a BST, the objective function we want to minimize, and proves them equivalent. Function *Wpl* is the intuitive global definition that sums a over all leaves and b over all nodes, taking their depth (= number of comparisons to reach that point) into account. Function *wpl* is a recursive definition and thus suitable for the later dynamic programming approaches to building a BST with the minimal weighted path length.

lemma *inorder_upto_split*:
  assumes *inorder* ⟨*l*,*k*,*r*⟩ = [*i*..*j*]
  shows *inorder l* = [*i*..*k*−1] *inorder r* = [*k*+1..*j*] *i* ≤ *k k* ≤ *j*
proof −
  have *k*: *k*∈*set*[*i*..*j*] **using** *assms* **by** (*metis set_inorder tree.set_intros*(2))
  have [*i*..*k*−1] @ *k* # [*k*+1..*j*] = [*i*..*j*]
    using *k upto_rec*1 *upto_split*1 **by** (*metis atLeastAtMost_iff set_upto*)
  also have ... = *inorder l* @ *k* # *inorder r* **using** *assms* **by** *auto*
  finally have *inorder l* = [*i*..*k*−1] ∧ *inorder r* = [*k*+1..*j*] (**is** *?A* ∧ *?B*)
    by(*auto simp*: *append_Cons_eq_iff*)
  thus *?A ?B* **by** *auto*
  show *i* ≤ *k k* ≤ *j* **using** *k* **by** *auto*
qed

fun *incr2* :: *int* × *nat* ⇒ *int* × *nat* **where**
incr2 (*x*,*n*) = (*x*, *n* + 1)

fun *leaves* :: *int* ⇒ *int tree* ⇒ (*int* ∗ *nat*) *set* **where**
leaves i Leaf = {(*i*,0)} |
leaves i (*Node l k r*) = *incr2* ‘ (*leaves i l* ∪ *leaves* (*k*+1) *r*)

fun *nodes* :: *int tree* ⇒ (*int* ∗ *nat*) *set* **where**
nodes Leaf = {} |
nodes (*Node l k r*) = {(*k*,1)} ∪ *incr2* ‘ (*nodes l* ∪ *nodes r*)

lemma *finite_nodes*: *finite* (*nodes t*)
by(*induction t*) *auto*

lemma *finite_leaves*: *finite* (*leaves i t*)
by(*induction i t rule*: *leaves.induct*) *auto*

lemma *notin_nodes*0: (*k*, 0) ∉ *nodes t*
by(*induction t*) *auto*

lemma *sum_incr2*: *sum f* (*incr2* ‘ *A*) = *sum* (λ*xy*. *f*(*fst xy*,*snd xy*+1)) *A*
proof −
  have *sum f* (*incr2* ‘ *A*) = *sum* (*f o incr2*) *A*
    by(*subst sum.reindex*)(*auto simp*: *inj_on_def*)
  also have *f o incr2* = (λ*xy*. *f*(*fst xy*,*snd xy*+1))
    by(*auto simp*: *fun_eq_iff*)
  finally show *?thesis* **by** *simp*
qed

lemma *fst_nodes*: *fst* ‘ *nodes t* = *set_tree t*
apply(*induction t*)
 apply *simp*
apply (*fastforce simp*: *image_def set_eq_iff ball_Un*)
done

lemma *fst_leaves*: ⟦ *inorder t* = [*i*..*j*]; *i* ≤ *j*+1⟧ ⟹ *fst* ‘ *leaves i t* = {*i*..*j*+1}
proof(*induction t arbitrary*: *i j*)
  case *Leaf*
  then show *?case* **by** *auto*
next
  case (*Node t*1 *k t*2)
  note *inorder* = *inorder_upto_split*[*OF Node.prems*(1)]
  show *?case*
    using *Node.IH*(1)[*OF inorder*(1)] *Node.IH*(2)[*OF inorder*(2)] *inorder*(3,4)
Node.prems(2)
    by (*fastforce simp*: *image_def set_eq_iff bex_Un*)
qed

lemma *sum_leaves*: ⟦ *inorder t* = [*i*..*j*]; *i* ≤ *j*+1 ⟧ ⟹
(∑ *x*∈*leaves i t*. (*f*(*fst x*) :: *nat*)) = *sum f* {*i*..*j*+1}
proof(*induction t arbitrary*: *i j*)
case *Leaf*
hence *i* = *j*+1 **by** *simp*
thus *?case* **by** *simp*
next
case (*Node l k r*)
note *inorder* = *inorder_upto_split*[*OF Node.prems*(1)]
let *?Ll* = *leaves i l* **let** *?Lr* = *leaves* (*k*+1) *r* **let** *?L* = *?Ll* ∪ *?Lr*
have *fst* ‘ *?Ll* ∩ *fst* ‘ *?Lr* = {} **using** *inorder*
by(*simp add*: *fst_leaves del*: *set_inorder add*: *set_inorder*[*symmetric*])
hence *l*0: *?Ll* ∩ *?Lr* = {} **by** *auto*
have {*i*..*j*+1} = {*i*..*k*} ∪ {*k*+1..*j*+1} **using** *inorder*(3,4) **by** *auto*
thus *?case*
using *Node.IH*(1)[*OF inorder*(1)] *Node.IH*(2)[*OF inorder*(2)] *inorder*(3,4) *Node.prems*(2)
by(*simp add*: *sum_incr*2 *sum_Un_nat finite_leaves l*0)
qed

lemma *sum_nodes*: *inorder t* = [*i*..*j*] ⟹
(∑ *xy*∈*nodes t*. (*f*(*fst xy*) :: *nat*)) = *sum f* {*i*..*j*}
proof(*induction t arbitrary*: *i j*)
case *Leaf* **thus** *?case* **by** *simp*
next
case (*Node l k r*)
note *inorder* = *inorder_upto_split*[*OF Node.prems*(1)]
let *?Nl* = *nodes l* **let** *?Nr* = *nodes r* **let** *?N* = *?Nl* ∪ *?Nr*
have (*fst* ‘ *?Nl*) ∩ (*fst* ‘ *?Nr*) = {} **using** *inorder*(1,2)
by(*simp add*: *fst_nodes del*: *set_inorder add*: *set_inorder*[*symmetric*])
hence *n*0: *?Nl* ∩ *?Nr* = {} **by** *auto*
have (∑ *xy*∈*nodes*(*Node l k r*). (*f* (*fst xy*) :: *nat*))
= (∑ *xy*∈*insert* (*k*, *Suc* 0) (*incr*2 ‘ (*nodes l* ∪ *nodes r*)). *f*(*fst xy*))
by(*simp*)
also have ... = *f k* + (∑ *xy*∈(*incr*2 ‘ (*nodes l* ∪ *nodes r*)). *f*(*fst xy*))
by(*subst sum.insert*, *auto simp*: *finite_nodes notin_nodes*0)
also have ... = *sum f* {*i*..*j*}
proof −
have {*i*..*j*} = {*i*..*k*−1} ∪ {*k*} ∪ {*k*+1..*j*} **using** *inorder*(3,4) **by** *auto*
thus *?thesis*
using *Node.IH*(1)[*OF inorder*(1)] *Node.IH*(2)[*OF inorder*(2)] *inorder*(3,4)
by(*simp add*: *sum_incr*2 *sum_Un_nat finite_nodes n*0)
qed
finally show *?case* **.**
qed

locale *wpl* =
fixes *w* :: *int* ⇒ *int* ⇒ *nat*
begin

fun *wpl* :: *int* ⇒ *int* ⇒ *int tree* ⇒ *nat* **where**
wpl i j Leaf = 0 |
wpl i j (*Node l k r*) = *wpl i* (*k*−1) *l* + *wpl* (*k*+1) *j r* + *w i j*

end

locale *Wpl* =
fixes *a b* :: *int* ⇒ *nat*
begin

definition *Wpl* :: *int* ⇒ *int tree* ⇒ *nat* **where**
Wpl i t = *sum* (λ(*k*,*c*). *c* ∗ *b k*) (*nodes t*) + *sum* (λ(*k*,*c*). *c* ∗ *a k*) (*leaves i t*)

definition *w* :: *int* ⇒ *int* ⇒ *nat* **where**
w i j = *sum a* {*i*..*j*+1} + *sum b* {*i*..*j*}

sublocale *wpl* **where** *w* = *w* **.**

lemma *inorder t* = [*i*..*j*] ⟹ *wpl i j t* = *Wpl i t*
proof(*induction t arbitrary*: *i j*)
case *Leaf* **thus** *?case* **by**(*simp add*: *Wpl_def*)
next
case (*Node l k r*)
let *?b* = λ(*k*,*c*). *c* ∗ *b k* **let** *?a* = λ(*k*,*c*). *c* ∗ *a k*
note *inorder* = *inorder_upto_split*[*OF Node.prems*]
let *?Nl* = *nodes l* **let** *?Nr* = *nodes r* **let** *?N* = *?Nl* ∪ *?Nr*
let *?Ll* = *leaves i l* **let** *?Lr* = *leaves* (*k*+1) *r* **let** *?L* = *?Ll* ∪ *?Lr*
have (*fst* ‘ *?Nl*) ∩ (*fst* ‘ *?Nr*) = {} **using** *inorder*(1,2)
by(*simp add*: *fst_nodes del*: *set_inorder add*: *set_inorder*[*symmetric*])
hence *n*0: *?Nl* ∩ *?Nr* = {} **by** *auto*
have *fst* ‘ *?Ll* ∩ *fst* ‘ *?Lr* = {} **using** *inorder*
by(*simp add*: *fst_leaves del*: *set_inorder add*: *set_inorder*[*symmetric*])
hence *l*0: *?Ll* ∩ *?Lr* = {} **by** *auto*
have *wpl i j* (*Node l k r*) = *Wpl i l* + *Wpl* (*k* + 1) *r* + *w i j*
using *Node.IH inorder* **by**(*simp*)
also have . . . = *sum ?b* (*nodes l*) + *sum ?a* (*leaves i l*) +
sum ?b (*nodes r*) + *sum ?a* (*leaves* (*k*+1) *r*) + *w i j*
by (*simp add*: *Wpl_def*)
also have . . . = (*sum ?b* (*nodes l*) + *sum ?b* (*nodes r*))
+ (*sum ?a* (*leaves i l*) + *sum ?a* (*leaves* (*k*+1) *r*)) + *w i j*
by(*simp add*: *algebra_simps*)
also have . . . = *sum ?b ?N* + *sum ?a ?L* + *w i j*
by(*simp add*: *sum_Un_nat finite_nodes finite_leaves l*0 *n*0)
also have . . . = *sum ?b ?N* + *sum ?a ?L* + *sum a* {*i*..*j*+1} + *sum b* {*i*..*j*}
by (*simp add*: *w_def*)
also have . . . = *sum ?b ?N* + *sum b* {*i*..*j*} + (*sum ?a ?L* + *sum a* {*i*..*j*+1})
by(*simp add*: *algebra_simps*)
also have *sum ?a ?L* + *sum a* {*i*..*j*+1} = *sum ?a* (*incr*2 ‘ *?L*)
proof −

```
    have {i..j+1} = {i..k} ∪ {k+1..j+1} using inorder(3,4) by auto
    thus ?thesis using inorder(3,4)
      by (simp add: sum_incr2 split_def sum.distrib sum_Un_nat finite_leaves l0
        sum_leaves[OF inorder(1)] sum_leaves[OF inorder(2)])
  qed
  also have sum ?b ?N + sum b {i..j}
    = sum ?b ?N + sum b ({i..k−1} ∪ {k+1..j}) + b k
  proof −
    have {i..j} = {k} ∪ {i..k−1} ∪ {k+1..j} using inorder(3,4) by auto
    thus ?thesis by simp
  qed
  also have sum ?b ?N + sum b ({i..k−1} ∪ {k+1..j}) = sum ?b (incr2 ‘ ?N)
    by (simp add: sum_incr2 split_def sum.distrib sum_Un_nat finite_nodes n0
        sum_nodes[OF inorder(1)] sum_nodes[OF inorder(2)])
  also have sum ?b (incr2 ‘ ?N) + b k = sum ?b ({(k,1)} ∪ incr2 ‘ ?N)
    by(simp, subst sum.insert, auto simp add: finite_nodes notin_nodes0)
  also have sum ?b ({(k,1)} ∪ incr2 ‘ ?N) + sum ?a (incr2 ‘ ?L) = Wpl i ⟨l,k,r⟩
    by(simp add: Wpl_def)
  finally show ?case .
qed

end

end
```

3 Optimal BSTs: The ‘Cubic’ Algorithm

```
theory Optimal_BST
imports Weighted_Path_Length Monad_Memo_DP.OptBST
begin
```

3.1 Function *argmin*

Function *argmin* was moved to *Monad_Memo_DP.argmin*. It iterates over a list and returns the rightmost element that minimizes a given function:

```
argmin ?f (?x # ?xs) =
(if ?xs = [] then ?x
 else let m = argmin ?f ?xs in if ?f ?x < ?f m then ?x else m)
```

An optimized version that avoids repeated computation of $f\ x$:

```
fun argmin2 :: ('a ⇒ ('b::linorder)) ⇒ 'a list ⇒ 'a * 'b where
argmin2 f (x#xs) =
  (let fx = f x
   in if xs = [] then (x, fx)
      else let mfm = argmin2 f xs
           in if fx < snd mfm then (x,fx) else mfm)
```

lemma *argmin2_argmin*: $xs \neq [] \Longrightarrow argmin2\ f\ xs = (argmin\ f\ xs, f(argmin\ f\ xs))$
by (*induction xs*) (*auto simp*: *Let_def*)

lemma *argmin_argmin2*[*code*]: $argmin\ f\ xs = (if\ xs = []\ then\ undefined\ else\ fst(argmin2\ f\ xs))$
apply(*auto simp*: *argmin2_argmin*)
apply (*meson argmin.elims list.distinct*(1))
done

lemma *argmin_in*: $xs \neq [] \Longrightarrow argmin\ f\ xs \in set\ xs$
using *argmin_forall*[*of xs* $\lambda x.\ x \in set\ xs$] **by** *blast*

lemma *argmin_pairs*: $xs \neq [] \Longrightarrow$
$(argmin\ f\ xs, f\ (argmin\ f\ xs)) = argmin\ snd\ (map\ (\lambda x.\ (x, f\ x))\ xs)$
by (*induction f xs rule*:*argmin.induct*) (*auto*, *smt snd_conv*)

lemma *argmin_map*: $xs \neq [] \Longrightarrow argmin\ c\ (map\ f\ xs) = f(argmin\ (c\ o\ f)\ xs)$
by(*induction xs*) (*simp_all add*: *Let_def*)

3.2 The 'Cubic' Algorithm

We hide the details of the access frequencies a and b by working with an abstract version of function w definied above (summing a and b). Later we interpret w accordingly.

locale *Optimal_BST* =
fixes $w :: int \Rightarrow int \Rightarrow nat$
begin

3.2.1 Functions *wpl* and *min_wpl*

sublocale *wpl* **where** $w = w$ **.**

Function *min_wpl i j* computes the minimal weighted path length of any tree t where $inorder\ t = [i..j]$. It simply tries all possible indices between i and j as the root. Thus it implicitly constructs all possible trees.

declare *conj_cong* [*fundef_cong*]
function *min_wpl* :: $int \Rightarrow int \Rightarrow nat$ **where**
min_wpl i j =
(*if* $i > j$ *then* 0
else Min $((\lambda k.\ min\_wpl\ i\ (k-1) + min\_wpl\ (k+1)\ j)\ `\ \{i..j\}) + w\ i\ j$)
by *auto*
termination by (*relation measure* $(\lambda(i,j).\ nat(j-i+1))$) *auto*
declare *min_wpl.simps*[*simp del*]

Note that for efficiency reasons we have pulled $+\ w\ i\ j$ out of *Min*. In the lemma below this is reversed because it simplifies the proofs. Similar optimizations are possible in other functions below.

```
lemma min_wpl_simps[simp]:
  i > j ⟹ min_wpl i j = 0
  i ≤ j ⟹ min_wpl i j =
     Min ((λk. min_wpl i (k−1) + min_wpl (k+1) j + w i j) ‘ {i..j})
by(auto simp add: min_wpl.simps[of i j] Min_add_commute)

lemma upto_split1:
  ⟦ i ≤ j;  j ≤ k ⟧ ⟹ [i..k] = [i..j−1] @ [j..k]
proof (induction j rule: int_ge_induct)
  case base thus ?case by (simp add: upto_rec1)
next
  case step thus ?case using upto_rec1 upto_rec2 by simp
qed
```

Function *local.min_wpl* returns a lower bound for all possible BSTs:

```
theorem min_wpl_is_optimal:
  inorder t = [i..j] ⟹ min_wpl i j ≤ wpl i j t
proof(induction i j t rule: wpl.induct)
  case 1
  thus ?case by(simp add: upto.simps split: if_splits)
next
  case (2 i j l k r)
  then show ?case
  proof cases
    assume i > j thus ?thesis by(simp)
  next
    assume [arith]: ¬ i > j

    note inorder = inorder_upto_split[OF 2.prems]

    let ?M = (λk. min_wpl i (k−1) + min_wpl (k+1) j + w i j) ‘ {i..j}
    let ?w = min_wpl i (k−1) + min_wpl (k+1) j + w i j

    have aux_min:Min ?M ≤ ?w
    proof (rule Min_le)
      show finite ?M by simp
      show ?w ∈ ?M using inorder(3,4) by simp
    qed

    have min_wpl i j = Min ?M by(simp)
    also have ... ≤ ?w by (rule aux_min)
    also have ... ≤ wpl i (k−1) l + wpl (k+1) j r + w i j
      using inorder(1,2) 2.IH by simp
    also have ... = wpl i j ⟨l,k,r⟩ by simp
    finally show ?thesis .
  qed
qed
```

Now we show that the lower bound computed by *local.min_wpl* is the wpl of an optimal tree that can be computed in the same manner.

3.2.2 Function *opt_bst*

This is the functional equivalent of the standard cubic imperative algorithm. Unless it is memoized, the complexity is again exponential. The pattern of recursion is the same as for *local.min_wpl* but instead of the minimal weight it computes a tree with the minimal weight:

function *opt_bst* :: *int* ⇒ *int* ⇒ *int tree* **where**
opt_bst i j =
  (*if i* > *j then Leaf*
   else argmin (*wpl i j*) [⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩. *k* ← [*i*..*j*]])
by *auto*
termination by (*relation measure* (λ(*i*,*j*) . *nat*(*j*−*i*+1))) *auto*
declare *opt_bst.simps*[*simp del*]

corollary *opt_bst_simps*[*simp*]:
  i > *j* ⟹ *opt_bst i j* = *Leaf*
  i ≤ *j* ⟹ *opt_bst i j* =
    (*argmin* (*wpl i j*) [⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩. *k* ← [*i*..*j*]])
by(*auto simp add*: *opt_bst.simps*[*of i j*])

As promised, *local.opt_bst* computes a tree with the minimal wpl:

theorem *wpl_opt_bst*: *wpl i j* (*opt_bst i j*) = *min_wpl i j*
proof(*induction i j rule*: *min_wpl.induct*)
  case (1 *i j*)
  show *?case*
  proof *cases*
    assume *i* > *j*
    thus *?thesis* **by**(*simp*)
  next
    assume [*arith*]: ¬ *i* > *j*
    let *?ts* = [⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩. *k* ← [*i*..*j*]]
    let *?M* = ((λ*k*. *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j* + *w i j*) ‘ {*i*..*j*})
    have 1: *?ts* ≠ [] **by** (*auto simp add*: *upto.simps*)
    have *wpl i j* (*opt_bst i j*) = *wpl i j* (*argmin* (*wpl i j*) *?ts*) **by** *simp*
    also have ... = *Min* (*wpl i j* ‘ (*set ?ts*))
      by(*rule argmin_Min*[*OF* 1])
    also have ... = *Min ?M*
    proof (*rule arg_cong*[**where** *f*=*Min*])
      show *wpl i j* ‘ (*set ?ts*) = *?M* **using** 1.*IH*
        by (*force simp*: *Bex_def image_iff* 1.*IH*)
    qed
    also have ... = *min_wpl i j* **by** *simp*
    finally show *?thesis* **.**
  qed
qed

corollary *opt_bst_is_optimal*:
  inorder t = [*i*..*j*] ⟹ *wpl i j* (*opt_bst i j*) ≤ *wpl i j t*
by (*simp add*: *min_wpl_is_optimal wpl_opt_bst*)

3.2.3 Function *opt_bst_wpl*

Function *local.opt_bst* is simplistic because it computes the wpl of each tree anew rather than returning it with the tree. That is what *opt_bst_wpl* does:

function *opt_bst_wpl* :: *int* ⇒ *int* ⇒ *int tree* × *nat* **where**
opt_bst_wpl i j =
(*if i* > *j then* (*Leaf*, 0)
else argmin snd [*let* (*t*1,*c*1) = *opt_bst_wpl i* (*k*−1);
(*t*2,*c*2) = *opt_bst_wpl* (*k*+1) *j*
in (⟨*t*1,*k*,*t*2⟩, *c*1 + *c*2 + *w i j*). *k* ← [*i*..*j*]])
by *auto*
termination
by (*relation measure* (λ(*i*,*j*). *nat*(*j*−*i*+1)))(*auto*)
declare *opt_bst_wpl.simps*[*simp del*]

Function *opt_bst_wpl* returns an optimal tree and its wpl:

lemma *opt_bst_wpl_eq_pair*:
opt_bst_wpl i j = (*opt_bst i j*, *wpl i j* (*opt_bst i j*))
proof(*induction i j rule*: *opt_bst_wpl.induct*)
case (1 *i j*)
note [*simp*] = *opt_bst_wpl.simps*[*of i j*]
show *?case*
proof *cases*
assume *i* > *j* **thus** *?thesis* **using** 1.*prems* **by** *auto*
next
assume ¬ *i* > *j*
thus *?thesis* **by** (*simp add*: *argmin_pairs comp_def* 1.*IH cong*: *list.map_cong_simp*)
qed
qed

corollary *opt_bst_wpl_eq_pair'*: *opt_bst_wpl i j* = (*opt_bst i j*, *min_wpl i j*)
by (*simp add*: *opt_bst_wpl_eq_pair wpl_opt_bst*)

end

end

4 Quadrangle Inequality

theory *Quadrilateral_Inequality*
imports *Main*
begin

definition *is_arg_min_on* :: ('*a* ⇒ ('*b*::*linorder*)) ⇒ '*a set* ⇒ '*a* ⇒ *bool* **where**
is_arg_min_on f S x = (*x* ∈ *S* ∧ (∀ *y* ∈ *S*. *f x* ≤ *f y*))

definition *Args_min_on* :: (*int* ⇒ ('*b*::*linorder*)) ⇒ *int set* ⇒ *int set* **where**
Args_min_on f I = {*k*. *is_arg_min_on f I k*}

lemmas *Args_min_simps* = *Args_min_on_def is_arg_min_on_def*

lemma *is_arg_min_on_antimono*: **fixes** $f :: \_ \Rightarrow \_::order$
shows $[\![$ *is_arg_min_on f S x*; $f\ y \leq f\ x$; $y \in S$ $]\!] \Longrightarrow$ *is_arg_min_on f S y*
by (*metis antisym is_arg_min_on_def*)

lemma *ex_is_arg_min_on_if_finite*: **fixes** $f :: 'a \Rightarrow 'b :: linorder$
shows $[\![$ *finite S*; $S \neq \{\}$ $]\!] \Longrightarrow \exists x.$ *is_arg_min_on f S x*
unfolding *is_arg_min_on_def* **using** *ex_min_if_finite*[*of f ' S*] **by** *fastforce*

locale *QI* =
fixes *c_k* :: *int* ⇒ *int* ⇒ *int* ⇒ *nat*
fixes *c* :: *int* ⇒ *int* ⇒ *nat*
and *w* :: *int* ⇒ *int* ⇒ *nat*
assumes *QI_w*: $[\![i \leq i'; i' < j; j \leq j']\!] \Longrightarrow$
w i j + *w i' j'*≤ *w i' j* + *w i j'*
assumes *monotone_w*: $[\![i \leq i'; i' < j; j \leq j']\!] \Longrightarrow$ *w i' j* ≤ *w i j'*
assumes *c_def*: $i < j \Longrightarrow$ *c i j* = *Min* ((*c_k i j*) ' {*i*+1..*j*})
assumes *c_k_def*: $[\![\ i < j;\ \ k \in \{i+1..j\}\ ]\!] \Longrightarrow$
c_k i j k = *w i j* + *c i* (*k*−1) + *c k j*
begin

abbreviation *mins i j* ≡ *Args_min_on* (*c_k i j*) {*i*+1..*j*}

definition *K i j* ≡ (*if i* = *j then i else Max* (*mins i j*))

lemma *c_def_rec*:
$i < j \Longrightarrow$ *c i j* = *Min* ((λ*k*. *c i* (*k*−1) + *c k j* + *w i j*) ' {*i*+1..*j*})
using *c_def c_k_def* **by** (*auto simp*: *algebra_simps image_def*)

lemma *mins_subset*: *mins i j* ⊆ {*i*+1..*j*}
by (*auto simp*: *Args_min_simps*)

lemma *mins_nonempty*: $i < j \Longrightarrow$ *mins i j* ≠ {}
using *ex_is_arg_min_on_if_finite*[*OF finite_atLeastAtMost_int*, *of i*+1 *j c_k i j*]
by(*auto simp*: *Args_min_simps*)

lemma *finite_mins*: *finite*(*mins i j*)
by(*simp add*: *finite_subset*[*OF mins_subset*])

lemma *is_arg_min_on_Min*:
assumes *finite A is_arg_min_on f A a* **shows** *Min* (*f ' A*) = *f a*
proof −
from *assms*(2) **have** *f ' A* ≠ {}
by (*fastforce simp*: *is_arg_min_on_def*)
thus *?thesis* **using** *assms* **by** (*simp add*: *antisym is_arg_min_on_def*)

qed

lemma *c_k_with_K*: $i < j \Longrightarrow c\ i\ j = c\_k\ i\ j\ (K\ i\ j)$
using *Max_in*[*of mins i j*] *finite_mins*[*of i j*] *mins_nonempty*[*of i j*]
is_arg_min_on_Min[*of* $\{i{+}1..j\}$ *c_k i j*]
by (*auto simp*: *Args_min_simps c_def K_def*)

lemma *K_subset*: **assumes** $i \le j$ **shows** $K\ i\ j \in \{i..j\}$ **using** *mins_subset K_def*
proof *cases*
assume $i = j$
thus *?thesis*
using *K_def* **by** *auto*
next
assume $\neg i = j$
hence $K\ i\ j \in \{i{+}1..j\}$ **using** *mins_subset K_def* $\langle i \le j \rangle$
by (*metis Max_in finite_mins less_le mins_nonempty subsetCE*)

thus *?thesis* **by** *auto*
qed

lemma *lemma_2*:
$[\![\ l = nat\ (j' - i);\ \ i \le i';\ \ i' \le j;\ \ j \le j'\ ]\!]$
$\Longrightarrow c\ i\ j + c\ i'\ j' \le c\ i\ j' + c\ i'\ j$
proof(*induction l arbitrary*: *i i' j j' rule*:*less_induct*)
case (*less l*)
show *?case*
proof *cases*
assume $l \le 1$
hence $i = i' \vee j = j'$ **using** *less.prems* **by** *linarith*
thus *?case* **by** *auto*
next
assume $\neg\ l \le 1$
show *?case*
proof *cases*
assume $i \ge i'$ **thus** *?thesis* **using** *less.prems* **by** *auto*
next
assume $\neg i \ge i'$
hence $i < i'$ **by** *simp*
show *?thesis*
proof *cases*
assume $j \ge j'$ **thus** *?thesis* **using** *less.prems* **by** *auto*
next
assume $\neg\ j \ge j'$
show *?thesis*

proof *cases*
assume $i' = j$
let *?k* $= K\ i\ j'$
have *?k* $\in \{i{+}1..j'\}$

```
      unfolding K_def
     using mins_subset Max_in[OF finite_mins mins_nonempty] less.prems
‹¬ i' ≤ i›
      by (smt subsetCE)
    show ?thesis

    proof cases
      assume ?k ≤ j

      have a: c i j ≤ w i j + c i (?k−1) + c ?k j
      proof −
        have c i j = Min ((λk. c i (k−1) + c k j + w i j) ` {i+1..j})
          using c_def_rec ‹¬ i' ≤ i› ‹i' = j› by auto
        also have ... ≤ c i (?k−1) + c ?k j + w i j
          using ‹?k∈{i+1..j'}› ‹?k ≤ j› by simp
        finally show ?thesis by simp
      qed

      have nat (j' − ?k) < l using ‹?k∈{i+1..j'}› less.prems by simp
      hence b: c ?k j + c j j' ≤ c ?k j' + c j j
        using ‹?k ≤ j› less.prems
          less.IH [where i = ?k and i' = j and j = j and j' = j', OF _ refl]
        by auto

      have c i j + c i' j' = c i j + c j j' by (simp add: ‹i' = j›)
      also have ...≤ w i j + c i (?k−1) + c ?k j + c j j'
        using a by auto
      also have ... ≤ w i j'+ c i (?k−1) + c ?k j + c j j'
        using less.prems monotone_w ‹i < i'› by simp
      also have ... ≤ w i j'+ c i (?k−1) + c ?k j' + c j j
        using b by auto
      also have ... = c i j' + c j j using ‹?k∈{i+1..j'}›
        by(simp add: c_k_def c_k_with_K)
      finally show ?thesis by(simp add: ‹i' = j›)
    next

      assume ¬ ?k ≤ j
      hence ?k ∈ {j+1..j'} using ‹?k ∈ {i+1..j'}› by auto
      have a: c j j' ≤ w j j' + c j (?k−1) + c ?k j'
      proof −
       have c j j' = Min ((λk. c j (k−1) + c k j' + w j j') ` {j+1..j'})
         using c_def_rec ‹¬ j' ≤ j› by auto
       also have ... ≤ c j (?k−1) + c ?k j' + w j j'
         using ‹?k ∈ {j+1..j'}› by simp
       finally show c j j' ≤ w j j' + c j (?k−1) + c ?k j' by simp
      qed

      have nat ((?k−1) −i) < l using ‹?k ∈ {i+1..j'}› less.prems by simp
      hence b: c i j + c j (?k−1) ≤ c i (?k−1) + c j j
```

```
      using less.prems ‹¬ ?k ≤ j›
        less.IH[where i=i and i'=j and j=j and j'=(?k−1), OF _ refl]
      by auto

    have c i j + c i' j' = c i j + c j j' by(simp add: ‹i' = j›)
    also have ... ≤ w j j' + c j (?k−1) + c ?k j' + c i j
      using a by simp
    also have ... ≤ w i j'+ c j (?k−1) + c ?k j' + c i j
      using less.prems monotone_w ‹?k ∈ {j+1..j'}› by simp
    also have ... ≤ w i j'+ c i (?k−1) + c ?k j' + c j j
      using b by simp
    also have ... ≤ c i j' + c j j
      using ‹?k∈{i+1..j'}› by (simp add: c_k_def c_k_with_K)
    finally show ?thesis by(simp add: ‹i' = j›)
  qed
next

  assume i' ≠ j
  let ?y = K i' j
  let ?z = K i j'
  have ?y ∈ {i'+1..j}
 using mins_subset less.prems ‹i' ≠ j› Max_in[OF finite_mins mins_nonempty]
    unfolding K_def by (metis le_less subsetCE)
  have ?z ∈ {i+1..j'}
 using mins_subset less.prems ‹i' ≠ j› Max_in[OF finite_mins mins_nonempty]
    unfolding K_def by (smt subsetCE)
  have w_mon: w i' j' + w i j ≤ w i' j + w i j'
    using less.prems QI_w ‹i' ≠ j› by force

  have i' < j' i < j using ‹i' ≠ j› less.prems by auto
  show ?thesis

  proof cases
    assume ?z ≤ ?y
    have ?y ∈ {i'+1..j'} using less.prems ‹?y ∈ {i'+1..j}› by simp
     have ?z ∈ {i+1..j} using ‹?z ∈ {i+1..j'}› ‹?z ≤ ?y› ‹?y ∈ {i'+1..j}›
by simp

    have a: c i' j' ≤ w i' j' + c i' (?y−1) + c ?y j'
    proof −
      have c i' j' = Min((λk. c i' (k−1) + c k j' + w i' j') ‘ {i'+1..j'})
        by (simp add: c_def_rec[OF ‹i' < j'›])
      also have ... ≤ w i' j' + c i' (?y−1) + c ?y j'
        using ‹?y ∈ {i'+1..j'}› by simp
      finally show ?thesis .
    qed

    have b: c i j ≤ w i j + c i (?z−1) + c ?z j
    proof −
```

```
    have c i j = Min ((λk. c i (k−1) + c k j + w i j)  ‘ {i+1..j})
      using ‹i < j› by (simp add: c_def_rec)
    also have ... ≤ w i j + c i (?z−1) + c ?z j
      using ‹?z ∈ {i+1..j}› by simp
    finally show ?thesis .
  qed

  have nat (j' − ?z) < l using ‹?z ∈ {i+1..j}› less.prems by simp
  hence IH_step: c ?z j + c ?y j' ≤ c ?z j' + c ?y j
    using ‹?z ≤ ?y› ‹j ≤ j'› ‹?y ∈ {i'+1..j}›
    less.IH[where i = ?z and i' = ?y and j = j and j' = j', OF _ refl]
    by simp

  have c i' j' + c i j
    ≤ w i' j + w i j' + c i' (?y−1) + c i (?z−1) + c ?y j' + c ?z j
    using a b w_mon by simp
  also have ... ≤ w i j' + w i' j +  c i' (?y−1) + c i (?z−1) + c ?y j +
c ?z j'
    using IH_step by auto
  also have ... = c i j' + c i' j using ‹?z ∈ {i+1..j'}› ‹?y ∈ {i'+1..j}›
    by(simp add: c_k_def c_k_with_K)
  finally show ?thesis by linarith
next

  assume ¬?z ≤ ?y

  have ?y ∈ {i+1..j} using less.prems ‹?y ∈ {i'+1..j}› by simp
  have ?z ∈ {i'+1..j'} using ‹?z ∈ {i+1..j'}› ‹¬?z ≤ ?y› ‹?y ∈ {i'+1..j}›
    by simp

  have a: c i' j' ≤ w i' j' + c i' (?z−1) + c ?z j'
  proof −
    have  c i' j' =  Min ((λk. c i' (k−1) + c k j' + w i' j') ‘ {i'+1..j'})
      using ‹i' < j'› by(simp add: c_def_rec)
    also have ... ≤ w i' j' + c i' (?z−1) + c ?z j'
      using ‹?z ∈ {i'+1..j'}› by simp
    finally show ?thesis .
  qed

  have b: c i j ≤ w i j + c i (?y−1) + c ?y j
  proof −
    have c i j = Min ((λk. c i (k−1) + c k j + w i j)  ‘ {i+1..j})
      using ‹i < j› by(simp add: c_def_rec)
    also have ... ≤ w i j + c i (?y−1) + c ?y j
      using ‹?y ∈ {i+1..j}› by simp
    finally show ?thesis .
  qed

  have nat (?z − 1 − i) < l using ‹?z ∈ {i'+1..j'}› less.prems by simp
```

```
          hence IH_Step:  c i (?y−1) + c i' (?z−1) ≤ c i' (?y−1) + c i (?z−1)
            using ‹?y ∈ {i'+1..j}› ‹¬?z ≤ ?y› ‹i ≤ i'›
              less.IH[where i=i and i'=i' and j=?y−1 and j'=?z−1, OF _ refl]
            by simp

          have c i' j' + c i j
            ≤ w i' j +  w i j' + c i' (?z−1) + c i (?y−1) + c ?z j' + c ?y j
            using a b w_mon by simp
          also have ... ≤ w i' j +  w i j' + c i (?z−1) + c i' (?y−1) + c ?z j' +
c ?y j
            using IH_Step by auto
          also have ... = c i j' + c i' j using ‹?z ∈ {i + 1..j'}› ‹?y ∈ {i' + 1..j}›
            by(simp add: c_k_def c_k_with_K)
          finally show ?thesis by linarith
        qed
      qed
    qed
  qed
 qed
qed

corollary QI': assumes i < k  k ≤ k'  k' ≤ j   c_k i j k' ≤ c_k i j k
shows c_k i (j+1) k' ≤ c_k i (j+1) k
proof −
  have c k j + c k' (j+1) ≤ c k' j + c k (j+1)
    using lemma_2[of _ j+1 k k' j] assms(1−3) by fastforce

  hence c_k i j k + c_k i (j+1) k' ≤ c_k i j k' + c_k i (j+1) k
    using assms(1−3) c_k_def by simp

  thus c_k i (j+1) k' ≤ c_k i (j+1) k
    using assms(4) by simp
qed

corollary QI'': assumes i+1 < k k ≤ k' k' ≤ j+1 c_k i (j+1) k' ≤ c_k i (j+1)
k
shows c_k (i+1) (j+1) k' ≤ c_k (i+1) (j+1) k
proof −
  have c i k + c (i + 1) k' ≤ c i k' + c (i + 1) k
    using lemma_2[of _ k' i i+1 k] assms(1,2) by fastforce

  hence c_k i (j+1) k + c_k (i+1) (j+1) k' ≤ c_k i (j+1) k' + c_k (i+1) (j+1)
k
    using c_k_def assms(1−3) lemma_2 by simp

  thus c_k (i+1) (j+1) k' ≤ c_k (i+1) (j+1) k
    using assms(4) by simp
qed
```

```
lemma lemma_3_1: assumes i ≤ j shows K i j ≤ K i (j+1)
proof cases
  assume i = j
  thus ?thesis
    by (metis K_def K_subset atLeastAtMost_iff less_add_one less_le)
next
  assume i≠j
  hence i < j using ‹i ≤ j› by simp

  let ?k = K i (j+1)
  have K i j ∈ {i+1..j} using K_def
     by (metis Max_in ‹i < j› mins_nonempty[OF ‹i < j›] finite_mins less_le
mins_subset subsetCE)

  have i < j+1 using ‹i < j› by linarith
  hence K i (j+1) ∈ {i+1..j+1}
    by (metis Max_in K_def mins_nonempty[OF ‹i < j+1›] finite_mins less_le
mins_subset subsetCE)

  have ∗: is_arg_min_on (c_k i (j+1)) {i+1..j+1} ?k
  proof −
    have K i (j+1) ∈ mins i (j+1) using finite_mins mins_nonempty ‹i < j›
K_def by fastforce
    thus is_arg_min_on (c_k i (j+1)) {i+1..j+1} (K i (j+1))
      unfolding Args_min_simps by blast
  qed
  show ?thesis
  proof cases
    assume ?k = j+1 thus ?thesis using ‹K i j ∈ {i+1..j}› by simp
  next
    assume ?k ≠ j+1
    hence ?k ∈ {i+1..j} using ‹K i (j+1) ∈ {i+1..j+1}› by auto
    have i≠j i≠j+1 using ‹i < j› by auto
    hence K_simps: K i j = Max (mins i j) K i (j+1) = Max (mins i (j+1))
      unfolding K_def by auto
    show ?thesis unfolding K_simps
    proof (rule Max.boundedI[OF finite_mins mins_nonempty[OF ‹i < j›]])
      fix k′ assume k′: k′ ∈ mins i j
      show k′ ≤ Max (mins i (j+1))
      proof (rule ccontr)
        assume ~ k′ ≤ Max (mins i (j+1))
        have c_k i (j+1) k′ ≤ c_k i (j+1) ?k unfolding K_simps
        proof (rule QI′)
          show i < Max (mins i (j+1))
            using ‹K i (j + 1) ∈ {i+1..j + 1}› K_simps by auto
          show Max (mins i (j+1)) ≤ k′ using ‹~ k′ ≤ Max (mins i (j+1))›
            by linarith
          show k′ ≤ j using mins_subset atLeastAtMost_iff k′ by blast
```

```
        show c_k i j k' ≤ c_k i j (Max (mins i (j + 1)))
          using k' ‹?k ∈ {i+1..j}› by(simp add: K_simps Args_min_simps)
      qed

      hence is_arg_min_on (c_k i (j+1)) {i+1..j+1} k'
        apply(rule is_arg_min_on_antimono[OF ∗])
        using mins_subset k' by fastforce
      hence k' ∈ mins i (j+1) using k' by (auto simp: Args_min_on_def)
      thus False using finite_mins ‹¬ k' ≤ Max (mins i (j + 1))› by auto
    qed
  qed
 qed
qed

lemma lemma_3_2: assumes i ≤ j shows K i (j+1) ≤ K (i+1) (j+1)
proof cases
 assume i = j
 thus ?thesis
   by (metis K_def K_subset atLeastAtMost_iff less_add_one less_le)
next
 assume i ≠ j
 hence i < j using ‹i ≤ j› by simp
 let ?k = K (i+1) (j+1)
 have K i (j+1) ∈ {i+1..j+1} unfolding K_def
  by (metis Max_in ‹i < j› finite_mins less_irrefl mins_nonempty mins_subset
subsetCE zless_add1_eq)

 have i+1 < j+1 using ‹i < j› by linarith
 hence K (i+1) (j+1) ∈ {i+1+1..j+1}
  using mins_nonempty[OF ‹i+1 < j+1›] mins_subset Max_in K_def finite_mins
    by (metis atLeastatMost_empty atLeastatMost_empty_iff2 contra_subsetD
empty_subsetI less_add_one psubsetI)

 have ∗: is_arg_min_on (c_k (i+1)(j+1)) {i+1+1..j+1} ?k
 proof −
  have K (i+1) (j+1) ∈ mins (i+1) (j+1)
    using finite_mins mins_nonempty ‹i + 1 < j + 1› unfolding K_def
    by (metis Max_in not_less_iff_gr_or_eq)
  thus is_arg_min_on (c_k (i+1) (j+1)) {i+1+1..j+1} (K (i+1) (j+1))
    unfolding Args_min_on_def by blast
 qed
 show ?thesis
 proof cases
  assume ?k = j+1 thus ?thesis using ‹K i (j+1) ∈ {i+1..j+1}› by simp
 next
  assume ?k ≠ j+1
  hence ?k ∈ {i+1+1..j} using ‹K (i+1) (j+1) ∈ {i+1+1..j+1}› by auto

  have  i≠j+1 i+1 ≠ j+1 using ‹i < j› by auto
```

```
    hence K_simps: K i (j+1) = Max (mins i (j+1))
                   K (i+1) (j+1) = Max (mins (i+1) (j+1))
      unfolding K_def by auto
    have i < j+1 using ‹i+1 < j+1› by simp

    show ?thesis unfolding K_simps
    proof (rule Max.boundedI[OF finite_mins mins_nonempty[OF ‹i < j+1›]])
      fix k' assume k': k' ∈ mins i (j+1)
      show k' ≤ Max (mins (i + 1) (j + 1))
      proof (rule ccontr)
        assume ~ k' ≤ Max (mins (i+1)(j+1))
        have c_k (i+1) (j+1) k' ≤ c_k (i+1) (j+1) ?k unfolding K_simps
          thm QI'[of i+1 Max(mins (i+1)(j+1)) k' j]
        proof (rule QI'')
          show i+1  < Max (mins (i+1)(j+1))
            using ‹K (i+1) (j+1) ∈ {i+1+1..j+1}› K_simps
            by auto
          show Max (mins (i + 1) (j + 1)) ≤ k'
            using ‹~ k' ≤ Max (mins (i+1)(j+1))› K_simps by linarith
          show k' ≤ j+1
            using mins_subset k' by fastforce
          show c_k i (j+1) k' ≤ c_k i (j+1) (Max (mins (i + 1) (j + 1)))
            using k' ‹?k ∈ {(i+1)+1..j + 1}› K_simps
            by(simp add: Args_min_simps)
        qed

        hence is_arg_min_on (c_k (i+1) (j+1)) {i+1+1..j+1} k'
          apply(rule is_arg_min_on_antimono[OF *])
          using mins_subset k' K_simps ‹?k ∈ {i+1+1..j}›
            ‹¬ k' ≤ Max (mins (i + 1) (j + 1))› atLeastAtMost_iff
          by force
        hence k' ∈ mins (i+1) (j+1) by (simp add: k' Args_min_on_def)
        thus False using finite_mins ‹¬ k' ≤ Max (mins (i+1)(j+1))› Max_ge
          by blast
      qed
    qed
  qed
qed

lemma lemma_3: assumes i ≤ j
  shows K i j ≤ K i (j+1) K i (j+1) ≤ K (i+1) (j+1)
using assms lemma_3_1 lemma_3_2 by blast+

end

end
```

5 Optimal BSTs: The 'Quadratic' Algorithm

theory *Optimal_BST2*
imports
Optimal_BST
Quadrilateral_Inequality
begin

Knuth presented an optimization of the previously known cubic dynamic programming algorithm to a quadratic one. A simplified proof of this optimization was found by Yao [4]. Mehlhorn follows Yao closely. The core of the optimization argument is given abstractly in theory *Optimal_BST.Quadrilateral_Inequality*. In addition we first need to establish some more properties of *argmin*.

An index-based specification of *argmin* expressing that the last minimal list-element is picked:

lemma *argmin_takes_last*: $xs \neq [] \Longrightarrow$
argmin f xs = *xs* ! *Max* $\{i.\ i < length\ xs \wedge (\forall x \in set\ xs.\ f(xs!i) \leq f\ x)\}$
(**is** _ $\Longrightarrow$ _ = _ ! *Max* (*?M xs*))
proof(*induction xs*)
case (*Cons x xs*)
show *?case*
proof *cases*
assume *xs* = [] **thus** *?thesis* **by**(*simp cong*: *conj_cong*)
next
assume 0: *xs* $\neq$ []
show *?thesis*
proof *cases*
assume 1: $\forall u \in set\ xs.\ f\ x < f\ u$
hence 2: *?M* (*x*#*xs*) = {0}
by (*fastforce simp*: *not_less*[*symmetric*] *less_Suc_eq_0_disj*)
have *f x* < *f* (*argmin f xs*) **using** 0 1 *argmin_Min*[*of xs f*] **by** *auto*
with 1 *Cons.prems* **show** *?case* **by**(*subst* 2) (*auto simp*: *Let_def*)
next
assume 1: $\neg(\forall u \in set\ xs.\ f\ x < f\ u)$
have 2: $\neg$ *f x* < *f* (*argmin f xs*) **using** 1 *argmin_Min*[*of xs f*] 0 **by** *auto*
have *argmin f xs* : $\{u \in set\ xs.\ \forall x \in set\ xs.\ f\ u \leq f\ x\}$
using 0 *argmin_Min*[*of xs f*] **by** (*simp add*: *argmin_in*)
hence $\{u \in set\ xs.\ \forall x \in set\ xs.\ f\ u \leq f\ x\} \neq \{\}$ **by** *blast*
hence *ne*: *?M xs* $\neq$ {} **by**(*auto simp*: *in_set_conv_nth*)
have *Max* (*?M* (*x*#*xs*)) = *Max* (*?M xs*) + 1
proof (*cases* $\exists u \in set\ xs.\ f\ u < f\ x$)
case *True*
hence *?M* (*x*#*xs*) = (+) 1 ' *?M xs*
by (*auto simp*: *nth_Cons'* *image_def less_Suc_eq_0_disj*)
thus *?thesis*
using *mono_Max_commute*[*of* (+) 1 *?M xs*] *ne* **by** (*auto simp*: *mono_def*)
next

```
      case False
      hence ∗: ?M (x#xs) = insert 0 ((+) 1 ‘ ?M xs)
        using 1 by (auto simp: nth_Cons′ image_def less_Suc_eq_0_disj)
      hence Max (?M (x#xs)) = Max ((+) 1 ‘ ?M xs) using Max_insert ne by
simp
    thus ?thesis using mono_Max_commute[of (+) 1 ?M xs] ne by (auto simp:
mono_def)
   qed
   with Cons 2 0 show ?case by auto
  qed
 qed
qed simp

lemma Min_ex: ⟦ finite F; F ≠ {} ⟧ ⟹ ∃ m ∈ F. ∀ n ∈ F. m ≤ (n::_::linorder)
using eq_Min_iff[of F Min F] by (fastforce)
```

A consequence of *argmin_takes_last*:

```
lemma argmin_Max_Args_min_on: assumes [arith]: i ≤ j
shows argmin f [i..j] = Max (Args_min_on f {i..j})
proof −
  let ?min = λk. ∀ n ∈ {i..j}. f([i..j]!k) ≤ f n
  let ?M = {k. k < nat(j−i+1) ∧ ?min k}
  let ?Max = Max ?M
  have ?M ≠ {} using Min_ex[of f ‘ {i..j}]
    apply(auto simp add: nth_upto)
    apply(rule_tac x=nat (m−i) in exI)
    by simp
  hence ?Max < nat(j−i+1) by(simp add: nth_upto)
  hence 1: i + int ?Max ≤ j by linarith
  have argmin f [i..j] = [i..j] ! ?Max
    using argmin_takes_last[of [i..j] f] by simp
  also have ... = i + int ?Max using 1 by(simp add: nth_upto)
  also have ... = i + Max(int ‘ {k. k < nat(j−i+1) ∧ ?min k})
    using ‹?M ≠ {}› by (simp add: monoI mono_Max_commute)
  also have ... = Max ((λx. i + x) ‘ (int ‘ {k. k < nat(j−i+1) ∧ ?min k}))
    using ‹?M ≠ {}› by (simp add: monoI mono_Max_commute)
  also have (λx. i + x) ‘ (int ‘ {k. k < nat(j−i+1) ∧ ?min k}) =
   {k. is_arg_min_on f {i..j} k}
     apply(auto simp: is_arg_min_on_def Ball_def nth_upto image_def cong:
conj_cong)
    apply(rule_tac x = x−i in exI)
    apply auto
    apply(rule_tac x = nat(x−i) in exI)
    by auto
  finally show ?thesis by(simp add: Args_min_simps)
qed
```

As a consequence of *argmin_Max_Args_min_on* the following lemma allows us to justify the restriction of the index range of *argmin* used below in the optimized (quadratic) algorithm.

lemma *argmin_red_ivl*:
assumes $i \le i'$ *argmin f* $[i..j] \in \{i'..j'\}$ $j' \le j$
shows *argmin f* $[i'..j'] =$ *argmin f* $[i..j]$
proof −
have *ij*[*arith*]: $i \le j$ **using** *assms* **by** *simp*
have *ij'*[*arith*]: $i' \le j'$ **using** *assms* **by** *simp*
from *Min_ex*[*of f* ' $\{i..j\}$] **have** *m*: $\exists\, m \in \{i..j\}.\ \forall\, n \in \{i..j\}.\ f\, m \le f\, n$ **by** *auto*
note $* =$ *argmin_Max_Args_min_on*[*OF ij*, *of f*]
note $** =$ *argmin_Max_Args_min_on*[*OF ij'*, *of f*]
let *?M* = *Args_min_on f* $\{i..j\}$
let *?M'* = *Args_min_on f* $\{i'..j'\}$
have *M*: *finite ?M ?M* $\neq \{\}$
using *m* **by** (*fastforce simp*: *Args_min_simps simp del*: *atLeastAtMost_iff*)+
have *Max ?M* $\in$ *?M* **by** (*simp add*: *M*)
have *Max ?M* $\in$ *?M'* **using** *Max_in*[*OF M*] *assms* $*$ **by**(*auto simp*: *Args_min_simps*)
have *?M'* $\subseteq$ *?M* **using** ‹*Max ?M* $\in$ *?M*› ‹*Max ?M* $\in$ *?M'*› *assms*(1,3)
by(*force simp add*: *Args_min_simps Ball_def*)
have *finite ?M'* **using** *M*(1) ‹*?M'* $\subseteq$ *?M*› *infinite_super* **by** *blast*
hence *Max ?M* $\le$ *Max ?M'* **by** (*simp add*: ‹*Max ?M* $\in$ *?M'*›)
have *Max ?M'* $\le$ *Max ?M* **using** *Max.subset_imp*[*OF* ‹*?M'* $\subseteq$ *?M*› _ *M*(1)]
‹*Max ?M* $\in$ *?M'*› **by** *auto*
thus *?thesis* **using** $*$ $**$ ‹*Max ?M* $\le$ *Max ?M'*› **by** *force*
qed

fun *root*:: *'a tree* $\Rightarrow$ *'a* **where**
root $\langle$_,*r*,_$\rangle$ = *r*

Now we can formulate and verify the improved algorithm. This requires two assumptions on the weight function *w*.

locale *Optimal_BST2* = *Optimal_BST* +
assumes *monotone_w*: $[\![i \le i'; i' \le j; j \le j']\!] \Longrightarrow w\ i'\ j \le w\ i\ j'$
assumes *QI_w*: $[\![i \le i'; i' \le j; j \le j']\!] \Longrightarrow w\ i\ j + w\ i'\ j' \le w\ i'\ j + w\ i\ j'$
begin

When finding an optimal tree for $[i..j]$ the optimization consists in reducing the search for the root from $[i..j]$ to $[root\ (opt\_bst2\ i\ (j-1))..root\ (opt\_bst2\ (i+1)\ j)]$:

function *opt_bst2* :: *int* $\Rightarrow$ *int* $\Rightarrow$ *int tree* **where**
opt_bst2 i j =
(*if i* > *j then Leaf else*
if i = *j then Node Leaf i Leaf else*
let left = *root* (*opt_bst2 i* (*j*−1)) *in*
let right= *root* (*opt_bst2* (*i*+1) *j*) *in*
argmin (*wpl i j*) [⟨*opt_bst2 i* (*k*−1), *k*, *opt_bst2* (*k*+1) *j*⟩. *k* ← [*left..right*]])
by *auto*

The termination of *opt_bst2* is not completely obvious. We first need to establish some functional properties of the terminating computations. We start by showing that the root of the returned tree is always between *left*

and *right*. This is essentially equivalent to proving that *left* $\leq$ *right* because otherwise *argmin* is applied to [], which is undefined.

```
lemma left_le_right:
 opt_bst2_dom(i,j) ⟹
  (i=j ⟶ root(opt_bst2 i j) = i) ∧
  (i<j ⟶ root(opt_bst2 i j) ∈ {root(opt_bst2 i (j−1)) .. root(opt_bst2 (i+1) j)})
proof (induction rule: opt_bst2.pinduct)
  case (1 i j)
  let ?left = root (opt_bst2 i (j−1))
  let ?right = root (opt_bst2 (i+1) j)
  let ?f =(λk. ⟨opt_bst2 i (k − 1), k, opt_bst2 (k + 1) j⟩)
  show ?case
  proof cases
    assume i > j thus ?thesis by auto
  next
    assume [arith]: ¬ i > j
    show ?thesis
    proof cases
      assume i = j thus ?thesis using opt_bst2.psimps[OF 1.hyps] by simp
    next
      assume [arith]: i ≠ j
      have left_le_right: ?left ≤ ?right
      proof cases
        assume [arith]: i = j−1
        have l: root (opt_bst2 i (j − 1)) = i using 1.IH(1) by auto
        have r: root (opt_bst2 (i+1) j) = j using 1.IH(2) by auto
        show ?thesis using l r by auto
      next
        assume ¬ i = j−1
        hence[arith]: i < j−1 by arith
        have ?left ≤ root (opt_bst2 (i + 1) (j − 1)) using 1.IH(1) by auto
        also have ... ≤ root (opt_bst2 (i+1) j) using 1.IH(2) by auto
        finally have ?left ≤ ?right .
        thus ?thesis by auto
      qed

      let ?lambda = λt. root t ∈ {?left .. ?right}
      show ?thesis
      using argmin_forall[of ‹map ?f [?left..?right]› ‹?lambda› ‹wpl i j›] left_le_right
        by (fastforce simp add: opt_bst2.psimps[OF 1.hyps])
    qed
  qed
qed
```

Now we can bound the result of *opt_bst2* easily:

```
lemma root_opt_bst2_bound:
  opt_bst2_dom (i,j) ⟹ i ≤ j ⟹ root (opt_bst2 i j) ∈ {i..j}
proof(induction i j rule:opt_bst2.pinduct)
  case (1 i j)
```

show *?case* **using** 1.*prems* 1.*IH*(1,2) *left_le_right*[*OF* 1.*hyps*] **by** *force*
qed

Now termination follows easily:

lemma *opt_bst2_dom*: ∀ *args*. *opt_bst2_dom args*
by (*relation measure* (λ(*i*,*j*). *nat* (*j*−*i*+1))) (*auto dest*: *root_opt_bst2_bound*)

termination by(*rule opt_bst2_dom*)

declare *opt_bst2.simps*[*simp del*]

abbreviation *min_wpl3 i j k* ≡ *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j* + *w i j*

The correctness proof [4] is based on a general theory of 'quatrilateral inequalities' developed in locale QI that we now instantiate:

interpretation *QI*
where
c = λ*i j*. *min_wpl* (*i*+1) *j*
and *c_k* = λ*i j*. *min_wpl3* (*i*+1) *j*
and *w* = λ*i j*. *w* (*i*+1) *j*
proof (*standard*, *goal_cases*)
case (1 *i i′ j j′*)
thus *?case* **using** *QI_w* **by** *simp*
next
case (2 *i i′ j j′*)
thus *?case* **using** *monotone_w* **by** *simp*
next
case (3 *i j*)
thus *?case* **by** *simp*
next
case (4 *i j k*)
show *?case* **by** *simp*
qed

lemma *K_argmin*: *i* < *j* ⟹ *K i j* = *argmin* (*min_wpl3* (*i*+1) *j*) [*i*+1..*j*]
by(*simp add*: *K_def argmin_Max_Args_min_on Args_min_on_def*)

theorem *opt_bst2_opt_bst*: *opt_bst2 i j* = *opt_bst i j*
proof (*induction i j rule*: *opt_bst2.induct*)
case (1 *i j*)
show *?case*
proof *cases*
assume *i* ≥ *j* **thus** *?thesis* **by**(*cases i*=*j*) (*auto simp*: *opt_bst2.simps*)
next
assume [*arith*]: ¬ *i* ≥ *j*
let *?c* = λ*k*. *min_wpl i* (*k*−1) + *min_wpl* (*k*+1) *j* + *w i j*
let *?opt* = λ*k*. ⟨*opt_bst i* (*k*−1), *k*, *opt_bst* (*k*+1) *j*⟩
have 1: *i* ≤ *K* (*i*−1) (*j*−1)
using *argmin_in*[*of* [*i*..*j*−1]] **by**(*auto simp add*: *K_argmin*)

```
    have 2: argmin ?c [i..j] ∈ {K (i−1) (j−1)..K i j}
      using lemma_3[of i−1 j−1] by(simp add: K_argmin)
    have 3: K i j ≤ j using argmin_in[of [i+1..j]] by(auto simp: K_argmin)
    have ∗: argmin ?c [K (i−1) (j−1)..K i j] = argmin ?c [i..j]
      by(rule argmin_red_ivl[OF 1 2 3])
    have opt_bst2 i j =
     argmin (wpl i j) (map ?opt [root(opt_bst2 i (j−1))..root(opt_bst2 (i+1) j)])
      using [[simp_depth_limit=3]]
     by(simp add: 1.IH(3,4)[OF _ _ refl refl] opt_bst2.simps[of i j] cong: list.map_cong_simp)
   also have ... = argmin (wpl i j) (map ?opt [root(opt_bst i (j−1))..root(opt_bst
(i+1) j)])
      by (simp add: 1.IH(1,2))
    also have root(opt_bst i (j−1)) = K (i−1) (j−1)
      by(simp add: argmin_map wpl_opt_bst comp_def K_argmin)
    also have root(opt_bst (i+1) j) = K i j
      by(simp add: argmin_map wpl_opt_bst comp_def K_argmin)
    also have argmin (wpl i j) (map ?opt [K (i − 1) (j − 1)..K i j])
       = ?opt (argmin (wpl i j o ?opt) [K (i−1) (j−1)..K i j])
      using lemma_3[of i−1 j−1] by(simp add: argmin_map)
    also have ... = ?opt (argmin ?c [K (i−1) (j−1)..K i j])
      by(simp add: comp_def wpl_opt_bst)
    also have ... = ?opt(argmin ?c [i..j])
      by (simp add: ∗)
    also have ... = ?opt(argmin (wpl i j o ?opt) [i..j])
      by(simp add: comp_def wpl_opt_bst)
    also have ... = argmin (wpl i j) (map ?opt [i..j])
      by(simp add: argmin_map)
    also have ... = opt_bst i j
      by simp
    finally show ?thesis .
  qed
qed

corollary opt_bst2_is_optimal: wpl i j (opt_bst2 i j) = min_wpl i j
by (simp add: opt_bst2_opt_bst wpl_opt_bst)


function opt_bst_wpl2 :: int ⇒ int ⇒ int tree × nat where
opt_bst_wpl2 i j =
  (if i > j then (Leaf,0) else
   if i = j then (Node Leaf i Leaf, w i i) else
   let l = root(fst(opt_bst_wpl2 i (j−1)));
       r = root(fst(opt_bst_wpl2 (i+1) j)) in
     argmin snd
       [let (tl,wl) = opt_bst_wpl2 i (k−1); (tr,wr) = opt_bst_wpl2 (k+1) j
        in (⟨tl, k, tr⟩, wl + wr + w i j) . k ← [l..r]])
by auto

lemma left_le_right2:
```

```
 opt_bst_wpl2_dom(i,j) ⟹
  (i=j ⟶ root(fst(opt_bst_wpl2 i j)) = i) ∧
  (i<j ⟶ root(fst(opt_bst_wpl2 i j)) ∈
    {root(fst(opt_bst_wpl2 i (j−1))) .. root(fst(opt_bst_wpl2 (i+1) j))})
proof (induction rule: opt_bst_wpl2.pinduct)
  case (1 i j)
  let ?l = root (fst(opt_bst_wpl2 i (j−1)))
  let ?r = root (fst(opt_bst_wpl2 (i+1) j))
  let ?f =λk. let (tl,wl) = opt_bst_wpl2 i (k−1); (tr,wr) = opt_bst_wpl2 (k+1) j
              in (⟨tl, k, tr⟩, wl + wr + w i j)
  show ?case
  proof cases
    assume i > j thus ?thesis by auto
  next
    assume [arith]: ¬ i > j
    show ?thesis
    proof cases
      assume i = j thus ?thesis using opt_bst_wpl2.psimps[OF 1.hyps] by simp
    next
      assume [arith]: i ≠ j
      have left_le_right: ?l ≤ ?r
      proof cases
        assume [arith]: i = j−1
        have l: root (fst(opt_bst_wpl2 i (j−1))) = i using 1.IH(1) by auto
        have r: root (fst(opt_bst_wpl2 (i+1) j)) = j using ‹i = j−1› 1.IH(2)
          by auto
        show ?thesis using l r by auto
      next
        assume ¬ i = j−1
        hence[arith]: i < j−1 by arith
        have ?l ≤ root (fst(opt_bst_wpl2 (i+1) (j−1))) using 1.IH(1) by auto
        also have ... ≤ root (fst(opt_bst_wpl2 (i+1) j)) using 1.IH(2) by auto
        finally have ?l ≤ ?r .
        thus ?thesis by auto
      qed

      let ?P = λt. root (fst t) ∈ {?l .. ?r}
      show ?thesis
        using argmin_forall[of ‹map ?f [?l..?r]› ?P snd] left_le_right
        by (fastforce simp add: opt_bst_wpl2.psimps[OF 1.hyps] split: prod.splits)
    qed
  qed
qed
```

Now we can bound the result of *opt_bst_wpl2*:

```
lemma root_opt_bst_wpl2_bound:
  opt_bst_wpl2_dom (i,j) ⟹ i ≤ j ⟹ root (fst(opt_bst_wpl2 i j)) ∈ {i..j}
proof(induction i j rule:opt_bst_wpl2.pinduct)
  case (1 i j)
```

```
  show ?case using  1.prems 1.IH(1)  1.IH(2)[OF _ _ refl] left_le_right2[OF
1.hyps]
    by fastforce
qed
```

Now termination follows easily:

```
lemma opt_bst_wpl2_dom: ∀ args. opt_bst_wpl2_dom args
by (relation measure (λ(i,j). nat (j−i+1))) (auto dest: root_opt_bst_wpl2_bound)

termination by(rule opt_bst_wpl2_dom)

declare opt_bst_wpl2.simps[simp del]

lemma opt_bst_wpl2_eq_pair:
  opt_bst_wpl2 i j = (opt_bst2 i j, wpl i j (opt_bst2 i j))
proof(induction i j rule: opt_bst_wpl2.induct)
  case (1 i j)
  note [simp] = opt_bst2.simps[of i j] opt_bst_wpl2.simps[of i j]
  show ?case
  proof cases
    assume i > j thus ?thesis using 1.prems by (simp)
  next
    assume [arith]: ¬ i > j
    show ?case
    proof cases
      assume [arith]: i = j
      show ?thesis by(simp) (simp add: ‹i = j›)
    next
      assume [arith]: i ≠ j
      let ?l = root (opt_bst2 i (j−1)) let ?r = root (opt_bst2 (i+1) j)
      have ∗: ?l ≤ ?r
      using left_le_right[of i j] by (fastforce simp: opt_bst2_opt_bst opt_bst2_dom)
      let ?f = λk. case opt_bst_wpl2 i (k−1) of
            (l,wl) ⇒ case opt_bst_wpl2 (k+1) j of
                        (r,wr) ⇒ (⟨l,k,r⟩, wl + wr + w i j)
      let ?g = λk. (⟨opt_bst2 i (k−1), k, opt_bst2 (k+1) j⟩,
                  wpl i (k−1) (opt_bst2 i (k−1)) + wpl (k+1) j (opt_bst2 (k+1) j)
+ w i j)
      have fg: ?f k = ?g k if k: k ∈ {?l..?r} for k
      proof −
        have 1: opt_bst_wpl2 i (k−1) = (opt_bst2 i (k−1), wpl i (k−1) (opt_bst2
i (k−1)))
          using k 1.IH(3) by(simp add: 1.IH(1,2))
        have 2: opt_bst_wpl2 (k+1) j = (opt_bst2 (k+1) j, wpl (k+1) j (opt_bst2
(k+1) j))
          using 1 k 1.IH(4) by(simp add: 1.IH(1,2))
        show ?thesis using 1 2 by(simp)
      qed
      have opt_bst_wpl2 i j =
```

```
      argmin snd (map ?f [root(fst(opt_bst_wpl2 i (j−1)))..root(fst(opt_bst_wpl2
(i+1) j))])
       by(simp)
     also have ... = argmin snd (map ?f [?l..?r])
       by(simp add: 1.IH(1,2))
     also have ... = argmin snd (map ?g [?l..?r])
       using fg by (simp cong: list.map_cong_simp)
     also have ... = (opt_bst2 i j, wpl i j (opt_bst2 i j)) using ∗
       by(simp add: argmin_pairs comp_def)
     finally show ?thesis .
   qed
  qed
qed

corollary opt_bst_wpl2_eq_pair′: opt_bst_wpl2 i j = (opt_bst i j, min_wpl i j)
by (simp add: opt_bst_wpl2_eq_pair opt_bst2_opt_bst wpl_opt_bst)

end

end


theory Optimal_BST_Examples
imports HOL−Library.Tree
begin
```

Example by Mehlhorn:

```
definition a_ex1 :: int ⇒ nat where
a_ex1 i = [4,0,0,3,10] ! nat i

definition b_ex1 :: int ⇒ nat where
b_ex1 i = [1,3,3,0] ! nat i

definition t_opt_ex1 :: int tree where
t_opt_ex1 = ⟨⟨⟨⟩, 0, ⟨⟩, 1, ⟨⟩⟩, 2, ⟨⟨⟩, 3, ⟨⟩⟩⟩
```

Example by Knuth:

```
definition a_ex2 :: int ⇒ nat where
a_ex2 i = 0

definition b_ex2 :: int ⇒ nat where
b_ex2 i = [32,7,69,13,6,15,10,8,64,142,22,79,18,9] ! nat i

definition t_opt_ex2 :: int tree where
t_opt_ex2 =
  ⟨
   ⟨
    ⟨⟨⟩, 0, ⟨⟨⟩, 1, ⟨⟩⟩⟩,
    2,
```

```
⟨
 ⟨
  ⟨⟨⟩, 3, ⟨⟨⟩, 4, ⟨⟩⟩⟩,
  5,
  ⟨⟨⟩, 6, ⟨⟨⟩, 7, ⟨⟩⟩⟩
 ⟩,
 8,
 ⟨⟩
⟩
⟩,
9,
⟨⟨⟨⟩, 10, ⟨⟩⟩,
 11,
 ⟨⟨⟩, 12, ⟨⟨⟩, 13, ⟨⟩⟩
 ⟩
⟩
⟩
```

end

6 Code Generation (unmemoized)

theory *Optimal_BST_Code*
imports
  *Optimal_BST*2
  Optimal_BST_Examples
begin

locale *Wpl_Optimal_BST* = *Wpl a b* + *Optimal_BST* **where** *w* = *Wpl.w a b*
for *a b*

locale *Wpl_Optimal_BST*2 = *Wpl a b* + *Optimal_BST*2 **where** *w* = *Wpl.w a b*
for *a b*

global_interpretation *Wpl_Optimal_BST* + *Wpl_Optimal_BST*2
defines *wpl_ab* = *wpl* **and** *opt_bst_ab* = *opt_bst* **and** *opt_bst*2*_ab* = *opt_bst*2
proof (*standard*, *unfold Wpl.w_def*, *goal_cases*)
  case (1 *i i' j j'*)
 thus *?case* **by** (*simp add*: *add_mono_thms_linordered_semiring*(1) *sum_mono*2)
next
  note *un*1 = *ivl_disj_un_two*(7)[*symmetric*]
  note *un*2 = *ivl_disj_un_two*(8)[*symmetric*]
  case (2 *i i' j j'*)
  have {*i*..<*i'*} ∩ {*j*<..*ub*} = {} **for** *ub* **using** ‹*i'* ≤ *j*› **by** *auto*
  with 2 **show** *?case*
    using *un*2[*of i' j j'*] *un*1[*of i i' j*] *un*1[*of i i' j'*]
          *un*2[*of i' j j'*+1] *un*1[*of i i' j*+1] *un*1[*of i i' j'*+1]
    by (*simp add*: *sum_Un_nat algebra_simps ivl_disj_int Int_Un_distrib*)

qed

Examples:

lemma *opt_bst_ab a_ex1 b_ex1 0 3 = t_opt_ex1*
by *eval*

lemma *opt_bst2_ab a_ex2 b_ex2 0 13 = t_opt_ex2*
by *eval*

end

7 Memoization

theory *Optimal_BST_Memo*
imports
Optimal_BST
Monad_Memo_DP.State_Main
HOL−Library.Product_Lexorder
HOL−Library.RBT_Mapping
Optimal_BST_Examples
begin

This theory memoizes the recursive algorithms with the help of our generic memoization framework. Note that currently only the tree building (function *Optimal_BST.opt_bst*) is memoized but not the computation of *w*.

global_interpretation *Wpl*
where *a = a* **and** *b = b* **for** *a b*
defines *w_ab = w* **and** *wpl_ab = wpl.wpl w_ab* **.**

First we express *argmin* via *fold*. Primarily because we have a monadic version of *fold* already. At the same time we improve efficiency.

lemma *fold_argmin*: *fold* (λ*x* (*m*,*fm*). *let fx = f x in if fx ≤ fm then* (*x*,*fx*) *else* (*m*,*fm*)) *xs* (*x*,*f x*)
= (*argmin f* (*x*#*xs*), *f*(*argmin f* (*x*#*xs*)))
by (*induction xs arbitrary*: *x*) (*auto simp*: *Let_def split*: *prod.split*)

lemma *argmin_fold*: *argmin f xs* = (*case xs of* [] ⇒ *undefined* |
x#*xs* ⇒ *fst*(*fold* (λ*x* (*m*,*fm*). *let fx = f x in if fx ≤ fm then* (*x*,*fx*) *else* (*m*,*fm*)) *xs* (*x*,*f x*)))
apply(*auto simp*:*fold_argmin split*: *list.split*)
apply (*meson argmin.elims list.distinct*(1))
done

The actual memoization of the cubic algorithm:

context *Optimal_BST*
begin

memoize_fun opt_bst_m: opt_bst **with_memory** $dp\_consistency\_mapping$
monadifies (*state*) $opt\_bst.simps$[*unfolded argmin_fold*]

thm $opt\_bst_m'.simps$

memoize_correct
by *memoize_prover*

lemmas [*code*] = $opt\_bst_m.memoized\_correct$

end

Code generation:

global_interpretation *Optimal_BST*
where $w = w\_ab\ a\ b$
rewrites $wpl.wpl\ (w\_ab\ a\ b) = wpl\_ab\ a\ b$ **for** $a\ b$
defines $opt\_bst\_ab = opt\_bst$ **and** $opt\_bst\_ab' = opt\_bst_m'$
by(*simp add*: *wpl_ab_def*)

Examples:

lemma $opt\_bst\_ab\ a\_ex1\ b\_ex1\ 0\ 3 = t\_opt\_ex1$
by *eval*

lemma $opt\_bst\_ab\ a\_ex2\ b\_ex2\ 0\ 13 = t\_opt\_ex2$
by *eval*

end

References

[1] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.

[2] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984.

[3] S. Wimmer, S. Hu, and T. Nipkow. Monadification, memoization and dynamic programming. *Archive of Formal Proofs*, 2018. http://isa-afp.org/entries/Monad_Memo_DP.html, Formal proof development.

[4] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *STOC*, pages 429–435. ACM, 1980.