Multidimensional Binary Search Trees

Martin Rau

December 14, 2021

Abstract

This entry provides a formalization of multidimensional binary trees, also known as k-d trees. It includes a balanced build algorithm as well as the nearest neighbor algorithm and the range search algorithm. It is based on the papers "Multidimensional binary search trees used for associative searching" [1] and "An Algorithm for Finding Best Matches in Logarithmic Expected Time" [2].

Contents

1 Definition of the k-d Tree

theory *KD-Tree*
imports
Complex-Main
HOL−Analysis.Finite-Cartesian-Product
HOL−Analysis.Topology-Euclidean-Space
begin

A k-d tree is a space-partitioning data structure for organizing points in a k-dimensional space. In principle the k-d tree is a binary tree. The leafs hold the k-dimensional points and the nodes contain left and right subtrees as well as a discriminator v at a particular axis k. Every node divides the space into two parts by splitting along a hyperplane. Consider a node n with associated discriminator v at axis k. All points in the left subtree must have a value at axis k that is less than or equal to v and all points in the right subtree must have a value at axis k that is greater than v.

Deviations from the papers:

The chosen tree representation is taken from [2] with one minor adjustment. Originally the leafs hold buckets of points of size b. This representation fixes the bucket size to $b = 1$, a single point per Leaf. This is only a minor adjustment since the paper proves that $b = 1$ is the optimal bucket size for minimizing the running time of the nearest neighbor algorithm [2], only simplifies building the optimized k-d trees [2] and has little influence on the search algorithm [1].

type-synonym $'k\ point = (real,\ 'k)\ vec$

lemma *dist-point-def*:
fixes $p_0 :: ('k::finite)\ point$
shows $dist\ p_0\ p_1 = sqrt\ (\sum k \in UNIV.\ (p_0\$k - p_1\$k)^2)$
unfolding *dist-vec-def L2-set-def dist-real-def* **by** *simp*

datatype $'k\ kdt =$
Leaf 'k point
| *Node 'k real 'k kdt 'k kdt*

1.1 Definition of the k-d Tree Invariant and Related Functions

fun $set\text{-}kdt :: 'k\ kdt \Rightarrow ('k\ point)\ set$ **where**
$set\text{-}kdt\ (Leaf\ p) = \{\ p\ \}$
| $set\text{-}kdt\ (Node\ \text{-}\ \text{-}\ l\ r) = set\text{-}kdt\ l \cup set\text{-}kdt\ r$

definition $spread :: ('k::finite) \Rightarrow 'k\ point\ set \Rightarrow real$ **where**
$spread\ k\ P = (if\ P = \{\}\ then\ 0\ else\ let\ V = (\lambda p.\ p\$k)\ `\ P\ in\ Max\ V - Min\ V)$

definition $widest\text{-}spread\text{-}axis :: ('k::finite) \Rightarrow 'k\ set \Rightarrow 'k\ point\ set \Rightarrow bool$ **where**

$widest\text{-}spread\text{-}axis\ k\ K\ ps \longleftrightarrow (\forall k' \in K.\ spread\ k'\ ps \le spread\ k\ ps)$

fun *invar* :: ('k::finite) kdt ⇒ bool **where**
$invar\ (Leaf\ p) \longleftrightarrow True$
$|\ invar\ (Node\ k\ v\ l\ r) \longleftrightarrow (\forall p \in set\text{-}kdt\ l.\ p\$k \le v) \land (\forall p \in set\text{-}kdt\ r.\ v < p\$k)$
$\land$
$widest\text{-}spread\text{-}axis\ k\ UNIV\ (set\text{-}kdt\ l \cup set\text{-}kdt\ r) \land invar\ l \land invar\ r$

fun *size-kdt* :: 'k kdt ⇒ nat **where**
$size\text{-}kdt\ (Leaf\ \text{-}) = 1$
$|\ size\text{-}kdt\ (Node\ \text{-}\ \text{-}\ l\ r) = size\text{-}kdt\ l + size\text{-}kdt\ r$

fun *height* :: 'k kdt ⇒ nat **where**
$height\ (Leaf\ \text{-}) = 0$
$|\ height\ (Node\ \text{-}\ \text{-}\ l\ r) = max\ (height\ l)\ (height\ r) + 1$

fun *min-height* :: 'k kdt ⇒ nat **where**
$min\text{-}height\ (Leaf\ \text{-}) = 0$
$|\ min\text{-}height\ (Node\ \text{-}\ \text{-}\ l\ r) = min\ (min\text{-}height\ l)\ (min\text{-}height\ r) + 1$

definition *balanced* :: 'k kdt ⇒ bool **where**
$balanced\ kdt \longleftrightarrow height\ kdt - min\text{-}height\ kdt \le 1$

fun *complete* :: 'k kdt ⇒ bool **where**
$complete\ (Leaf\ \text{-}) = True$
$|\ complete\ (Node\ \text{-}\ \text{-}\ l\ r) \longleftrightarrow complete\ l \land complete\ r \land height\ l = height\ r$

lemma *invar-l*:
$invar\ (Node\ k\ v\ l\ r) \Longrightarrow invar\ l$
by *simp*

lemma *invar-r*:
$invar\ (Node\ k\ v\ l\ r) \Longrightarrow invar\ r$
by *simp*

lemma *invar-l-le-k*:
$invar\ (Node\ k\ v\ l\ r) \Longrightarrow \forall p \in set\text{-}kdt\ l.\ p\$k \le v$
by *simp*

lemma *invar-r-ge-k*:
$invar\ (Node\ k\ v\ l\ r) \Longrightarrow \forall p \in set\text{-}kdt\ r.\ v < p\$k$
by *simp*

lemma *invar-set*:
$set\text{-}kdt\ (Node\ k\ v\ l\ r) = set\text{-}kdt\ l \cup set\text{-}kdt\ r$
by *simp*

1.2 Lemmas adapted from *HOL−Library.Tree* to k-d Tree

lemma *size-ge0*[*simp*]:
  0 < size-kdt kdt
  by (*induction kdt*) *auto*

lemma *eq-size-1*[*simp*]:
  size-kdt kdt = 1 $\longleftrightarrow$ ($\exists$ *p. kdt = Leaf p*)
  apply (*induction kdt*)
  apply (*auto*)
  using *size-ge0 nat-less-le* **apply** *blast+*
  done

lemma *eq-1-size*[*simp*]:
  1 = size-kdt kdt $\longleftrightarrow$ ($\exists$ *p. kdt = Leaf p*)
  using *eq-size-1* **by** *metis*

lemma *neq-Leaf-iff*:
  ($\nexists$ *p. kdt = Leaf p*) = ($\exists$ *k v l r. kdt = Node k v l r*)
  by (*cases kdt*) *auto*

lemma *eq-height-0*[*simp*]:
  height kdt = 0 $\longleftrightarrow$ ($\exists$ *p. kdt = Leaf p*)
  by (*cases kdt*) *auto*

lemma *eq-0-height*[*simp*]:
  0 = height kdt $\longleftrightarrow$ ($\exists$ *p. kdt = Leaf p*)
  by (*cases kdt*) *auto*

lemma *eq-min-height-0*[*simp*]:
  min-height kdt = 0 $\longleftrightarrow$ ($\exists$ *p. kdt = Leaf p*)
  by (*cases kdt*) *auto*

lemma *eq-0-min-height*[*simp*]:
  0 = min-height kdt $\longleftrightarrow$ ($\exists$ *p. kdt = Leaf p*)
  by (*cases kdt*) *auto*

lemma *size-height*:
  size-kdt kdt $\leq$ *2* ^ *height kdt*
proof(*induction kdt*)
  case (*Node k v l r*)
  show *?case*
  proof (*cases height l* $\leq$ *height r*)
    case *True*
    have *size-kdt* (*Node k v l r*) = *size-kdt l* + *size-kdt r* **by** *simp*
    also have ... $\leq$ *2* ^ *height l* + *2* ^ *height r* **using** *Node.IH* **by** *arith*
    also have ... $\leq$ *2* ^ *height r* + *2* ^ *height r* **using** *True* **by** *simp*
    also have ... = *2* ^ *height* (*Node k v l r*)
      using *True* **by** (*auto simp*: *max-def mult-2*)
    finally show *?thesis* **.**

next
case *False*
have *size-kdt* (*Node k v l r*) $=$ *size-kdt l* $+$ *size-kdt r* **by** *simp*
also have $\ldots \leq$ *2* $\hat{}$ *height l* $+$ *2* $\hat{}$ *height r* **using** *Node.IH* **by** *arith*
also have $\ldots \leq$ *2* $\hat{}$ *height l* $+$ *2* $\hat{}$ *height l* **using** *False* **by** *simp*
finally show *?thesis* **using** *False* **by** (*auto simp*: *max-def mult-2*)
qed
qed *simp*

lemma *min-height-le-height*:
min-height kdt $\leq$ *height kdt*
by (*induction kdt*) *auto*

lemma *min-height-size*:
2 $\hat{}$ *min-height kdt* $\leq$ *size-kdt kdt*
proof(*induction kdt*)
case (*Node k v l r*)
have (*2*::*nat*) $\hat{}$ *min-height* (*Node k v l r*) $\leq$ *2* $\hat{}$ *min-height l* $+$ *2* $\hat{}$ *min-height r*
by (*simp add*: *min-def*)
also have $\ldots \leq$ *size-kdt* (*Node k v l r*) **using** *Node.IH* **by** *simp*
finally show *?case* **.**
qed *simp*

lemma *complete-iff-height*:
complete kdt $\longleftrightarrow$ (*min-height kdt* $=$ *height kdt*)
apply (*induction kdt*)
apply *simp*
apply (*simp add*: *min-def max-def*)
by (*metis le-antisym le-trans min-height-le-height*)

lemma *size-if-complete*:
complete kdt $\Longrightarrow$ *size-kdt kdt* $=$ *2* $\hat{}$ *height kdt*
by (*induction kdt*) *auto*

lemma *complete-if-size-height*:
size-kdt kdt $=$ *2* $\hat{}$ *height kdt* $\Longrightarrow$ *complete kdt*
proof (*induction height kdt arbitrary*: *kdt*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc h*)
hence $\nexists$ *p*. *kdt* $=$ *Leaf p*
by *auto*
then obtain *k v l r* **where** [*simp*]: *kdt* $=$ *Node k v l r*
using *neq-Leaf-iff* **by** *metis*
have *1*: *height l* $\leq$ *h* **and** *2*: *height r* $\leq$ *h* **using** *Suc*(*2*) **by**(*auto*)
have *3*: $\neg$ *height l* $<$ *h*
proof
assume *0*: *height l* $<$ *h*
have *size-kdt kdt* $=$ *size-kdt l* $+$ *size-kdt r* **by** *simp*

```
    also have ... ≤ 2 ^ height l + 2 ^ height r
      using size-height[of l] size-height[of r] by arith
    also have  ... < 2 ^ h + 2 ^ height r using 0 by (simp)
    also have  ... ≤ 2 ^ h + 2 ^ h using 2 by (simp)
    also have ... = 2 ^ (Suc h) by (simp)
    also have ... = size-kdt kdt using Suc(2,3) by simp
    finally have size-kdt kdt < size-kdt kdt .
    thus False by (simp)
  qed
  have 4: ¬ height r < h
  proof
    assume 0: height r < h
    have size-kdt kdt = size-kdt l + size-kdt r by simp
    also have ... ≤ 2 ^ height l + 2 ^ height r
      using size-height[of l] size-height[of r] by arith
    also have  ... < 2 ^ height l + 2 ^ h using 0 by (simp)
    also have  ... ≤ 2 ^ h + 2 ^ h using 1 by (simp)
    also have ... = 2 ^ (Suc h) by (simp)
    also have ... = size-kdt kdt using Suc(2,3) by simp
    finally have size-kdt kdt < size-kdt kdt .
    thus False by (simp)
  qed
  from 1 2 3 4 have *: height l = h height r = h by linarith+
  hence size-kdt l = 2 ^ height l size-kdt r = 2 ^ height r
    using Suc(3) size-height[of l] size-height[of r] by (auto)
  with * Suc(1) show ?case by simp
qed

lemma complete-if-size-min-height:
  size-kdt kdt = 2 ^ min-height kdt ⟹ complete kdt
proof (induct min-height kdt arbitrary: kdt)
  case 0 thus ?case by auto
next
  case (Suc h)
  hence ∄ p. kdt = Leaf p
    by auto
  then obtain k v l r where [simp]: kdt = Node k v l r
    using neq-Leaf-iff by metis
  have 1: h ≤ min-height l and 2: h ≤ min-height r using Suc(2) by (auto)
  have 3: ¬ h < min-height l
  proof
    assume 0: h < min-height l
    have size-kdt kdt = size-kdt l + size-kdt r by simp
    also note min-height-size[of l]
    also(xtrans) note min-height-size[of r]
    also(xtrans) have (2::nat) ^ min-height l > 2 ^ h
        using 0 by (simp add: diff-less-mono)
    also(xtrans) have (2::nat) ^ min-height r ≥ 2 ^ h using 2 by simp
    also(xtrans) have (2::nat) ^ h + 2 ^ h = 2 ^ (Suc h) by (simp)
```

also have ... = *size-kdt kdt* **using** *Suc*(*2*,*3*) **by** *simp*
finally show *False* **by** (*simp add*: *diff-le-mono*)
qed
have *4*: ¬ *h* < *min-height r*
proof
assume *0*: *h* < *min-height r*
have *size-kdt kdt* = *size-kdt l* + *size-kdt r* **by** *simp*
also note *min-height-size*[*of l*]
also(*xtrans*) **note** *min-height-size*[*of r*]
also(*xtrans*) **have** (*2*::*nat*) ^ *min-height r* > *2* ^ *h*
using *0* **by** (*simp add*: *diff-less-mono*)
also(*xtrans*) **have** (*2*::*nat*) ^ *min-height l* ≥ *2* ^ *h* **using** *1* **by** *simp*
also(*xtrans*) **have** (*2*::*nat*) ^ *h* + *2* ^ *h* = *2* ^ (*Suc h*) **by** (*simp*)
also have ... = *size-kdt kdt* **using** *Suc*(*2*,*3*) **by** *simp*
finally show *False* **by** (*simp add*: *diff-le-mono*)
qed
from *1 2 3 4* **have** ∗: *min-height l* = *h min-height r* = *h* **by** *linarith*+
hence *size-kdt l* = *2* ^ *min-height l size-kdt r* = *2* ^ *min-height r*
using *Suc*(*3*) *min-height-size*[*of l*] *min-height-size*[*of r*] **by** (*auto*)
with ∗ *Suc*(*1*) **show** *?case*
by (*simp add*: *complete-iff-height*)
qed

lemma *complete-iff-size*:
complete kdt ⟷ *size-kdt kdt* = *2* ^ *height kdt*
using *complete-if-size-height size-if-complete* **by** *blast*

lemma *size-height-if-incomplete*:
¬ *complete kdt* ⟹ *size-kdt kdt* < *2* ^ *height kdt*
by (*meson antisym-conv complete-iff-size not-le size-height*)

lemma *min-height-size-if-incomplete*:
¬ *complete kdt* ⟹ *2* ^ *min-height kdt* < *size-kdt kdt*
by (*metis complete-if-size-min-height le-less min-height-size*)

lemma *balanced-subtreeL*:
balanced (*Node k v l r*) ⟹ *balanced l*
by (*simp add*: *balanced-def*)

lemma *balanced-subtreeR*:
balanced (*Node k v l r*) ⟹ *balanced r*
by (*simp add*: *balanced-def*)

lemma *balanced-optimal*:
assumes *balanced kdt size-kdt kdt* ≤ *size-kdt kdt′*
shows *height kdt* ≤ *height kdt′*
proof (*cases complete kdt*)
case *True*
have (*2*::*nat*) ^ *height kdt* ≤ *2* ^ *height kdt′*

```
  proof -
    have 2 ^ height kdt = size-kdt kdt
      using True by (simp add: complete-iff-height size-if-complete)
    also have ... ≤ size-kdt kdt' using assms(2) by simp
    also have ... ≤ 2 ^ height kdt' by (rule size-height)
    finally show ?thesis .
  qed
  thus ?thesis by (simp)
next
  case False
  have (2::nat) ^ min-height kdt < 2 ^ height kdt'
  proof -
    have (2::nat) ^ min-height kdt < size-kdt kdt
      by(rule min-height-size-if-incomplete[OF False])
    also have ... ≤ size-kdt kdt' using assms(2) by simp
    also have ... ≤ 2 ^ height kdt'  by(rule size-height)
    finally have (2::nat) ^ min-height kdt < (2::nat) ^ height kdt' .
    thus ?thesis .
  qed
  hence *: min-height kdt < height kdt' by simp
  have min-height kdt + 1 = height kdt
    using min-height-le-height[of kdt] assms(1) False
    by (simp add: complete-iff-height balanced-def)
  with * show ?thesis by arith
qed
```

1.3 Lemmas adapted from *HOL−Library.Tree-Real* to *k*-d Tree

```
lemma size-height-log:
  log 2 (size-kdt kdt) ≤ height kdt
  by (simp add: log2-of-power-le size-height)

lemma min-height-size-log:
  min-height kdt ≤ log 2 (size-kdt kdt)
  by (simp add: le-log2-of-power min-height-size)

lemma size-log-if-complete:
  complete kdt ⟹ height kdt = log 2 (size-kdt kdt)
  using complete-iff-size log2-of-power-eq by blast

lemma min-height-size-log-if-incomplete:
  ¬ complete kdt ⟹ min-height kdt < log 2 (size-kdt kdt)
  by (simp add: less-log2-of-power min-height-size-if-incomplete)

lemma min-height-balanced:
  assumes balanced kdt
  shows min-height kdt = nat(floor(log 2 (size-kdt kdt)))
proof cases
  assume *: complete kdt
```

  hence *size-kdt kdt* = *2* ^ *min-height kdt*
    by (*simp add*: *complete-iff-height size-if-complete*)
  from *log2-of-power-eq*[*OF this*] **show** *?thesis* **by** *linarith*
next
  assume ∗: ¬ *complete kdt*
  hence *height kdt* = *min-height kdt* + *1*
    using *assms min-height-le-height*[*of kdt*]
    by(*auto simp add*: *balanced-def complete-iff-height*)
  hence *size-kdt kdt* < *2* ^ (*min-height kdt* + *1*)
    by (*metis* ∗ *size-height-if-incomplete*)
  hence *log 2* (*size-kdt kdt*) < *min-height kdt* + *1*
    using *log2-of-power-less size-ge0* **by** *blast*
  thus *?thesis* **using** *min-height-size-log*[*of kdt*] **by** *linarith*
qed

lemma *height-balanced*:
  assumes *balanced kdt*
  shows *height kdt* = *nat*(*ceiling*(*log 2* (*size-kdt kdt*)))
proof *cases*
  assume ∗: *complete kdt*
  hence *size-kdt kdt* = *2* ^ *height kdt*
    by (*simp add*: *size-if-complete*)
  from *log2-of-power-eq*[*OF this*] **show** *?thesis*
    by *linarith*
next
  assume ∗: ¬ *complete kdt*
  hence ∗∗: *height kdt* = *min-height kdt* + *1*
    using *assms min-height-le-height*[*of kdt*]
    by(*auto simp add*: *balanced-def complete-iff-height*)
  hence *size-kdt kdt* ≤ *2* ^ (*min-height kdt* + *1*) **by** (*metis size-height*)
  from *log2-of-power-le*[*OF this size-ge0*] *min-height-size-log-if-incomplete*[*OF* ∗]
∗∗
  show *?thesis* **by** *linarith*
qed

lemma *balanced-Node-if-wbal1*:
  assumes *balanced l balanced r size-kdt l* = *size-kdt r* + *1*
  shows *balanced* (*Node k v l r*)
proof −
  from *assms*(*3*) **have** [*simp*]: *size-kdt l* = *size-kdt r* + *1* **by** *simp*
  have *nat* ⌈*log 2* (*1* + *size-kdt r*)⌉ ≥ *nat* ⌈*log 2* (*size-kdt r*)⌉
    by(*rule nat-mono*[*OF ceiling-mono*]) *simp*
  hence *1*: *height*(*Node k v l r*) = *nat* ⌈*log 2* (*1* + *size-kdt r*)⌉ + *1*
    using *height-balanced*[*OF assms*(*1*)] *height-balanced*[*OF assms*(*2*)]
    by (*simp del*: *nat-ceiling-le-eq add*: *max-def*)
  have *nat* ⌊*log 2* (*1* + *size-kdt r*)⌋ ≥ *nat* ⌊*log 2* (*size-kdt r*)⌋
    by(*rule nat-mono*[*OF floor-mono*]) *simp*
  hence *2*: *min-height*(*Node k v l r*) = *nat* ⌊*log 2* (*size-kdt r*)⌋ + *1*
    using *min-height-balanced*[*OF assms*(*1*)] *min-height-balanced*[*OF assms*(*2*)]

```
    by (simp)
  have size-kdt r ≥ 1 by (simp add: Suc-leI)
  then obtain i where i: 2 ^ i ≤ size-kdt r size-kdt r < 2 ^ (i + 1)
    using ex-power-ivl1[of 2 size-kdt r] by auto
  hence i1: 2 ^ i < size-kdt r + 1 size-kdt r + 1 ≤ 2 ^ (i + 1) by auto
  from 1 2 floor-log-nat-eq-if[OF i] ceiling-log-nat-eq-if[OF i1]
  show ?thesis by(simp add:balanced-def)
qed

lemma balanced-sym:
  balanced (Node k v l r) ⟹ balanced (Node k' v' r l)
  by (auto simp: balanced-def)

lemma balanced-Node-if-wbal2:
  assumes balanced l balanced r abs(int(size-kdt l) − int(size-kdt r)) ≤ 1
  shows balanced (Node k v l r)
proof −
  have size-kdt l = size-kdt r ∨ (size-kdt l = size-kdt r + 1 ∨ size-kdt r = size-kdt
l + 1) (is ?A ∨ ?B)
    using assms(3) by linarith
  thus ?thesis
  proof
    assume ?A
    thus ?thesis using assms(1,2)
      apply(simp add: balanced-def min-def max-def)
      by (metis assms(1,2) balanced-optimal le-antisym le-less)
  next
    assume ?B
    thus ?thesis
      by (meson assms(1,2) balanced-sym balanced-Node-if-wbal1)
  qed
qed

end
```

2 Building a balanced k-d Tree from a List of Points

```
theory Build
imports
  KD-Tree
  Median-Of-Medians-Selection.Median-Of-Medians-Selection
begin
```

Build a balanced k-d Tree by recursively partition the points into two lists. The partitioning criteria will be the median at a particular axis k. The left list will contain all points p with $p \ \$ \ k \leq median$. The right list will contain all points with median at axis $median < p \ \$ \ k$. The left and right list differ in length by one or none. The axis k will the widest spread axis.

2.1 Auxiliary Lemmas

lemma *length-filter-mset-sorted-nth*:
  assumes *distinct xs n* $<$ *length xs sorted xs*
  shows $\{\#\ x \in\#\ \mathit{mset\ xs}.\ x \leq \mathit{xs}\ !\ n\ \#\} = \mathit{mset}\ (\mathit{take}\ (n + 1)\ \mathit{xs})$
  using *assms*
proof (*induction xs arbitrary*: *n rule*: *list.induct*)
  case (*Cons x xs*)
  thus *?case*
  proof (*cases n*)
    case *0*
    thus *?thesis*
      using *Cons.prems*(*1*,*3*) *filter-mset-is-empty-iff* **by** *fastforce*
  next
    case (*Suc n′*)
    thus *?thesis*
      using *Cons* **by** *simp*
  qed
qed *auto*

lemma *length-filter-sort-nth*:
  assumes *distinct xs n* $<$ *length xs*
  shows *length* (*filter* ($\lambda x.\ x \leq$ *sort xs* ! *n*) *xs*) = *n* + *1*
proof −
  have *length* (*filter* ($\lambda x.\ x \leq$ *sort xs* ! *n*) *xs*) = *length* (*filter* ($\lambda x.\ x \leq$ *sort xs* ! *n*) (*sort xs*))
    by (*simp add*: *filter-sort*)
  also have ... = *size* (*mset* (*filter* ($\lambda x.\ x \leq$ *sort xs* ! *n*) (*sort xs*)))
    using *size-mset* **by** *metis*
  also have ... = *size* ($\{\#\ x \in\#$ *mset* (*sort xs*). $x \leq$ *sort xs* ! *n* $\#\}$)
    using *mset-filter* **by** *simp*
  also have ... = *size* (*mset* (*take* (*n* + *1*) (*sort xs*)))
    using *length-filter-mset-sorted-nth assms sorted-sort distinct-sort length-sort* **by** *metis*
  finally show *?thesis*
    using *assms*(*2*) **by** *auto*
qed

2.2 Widest Spread Axis

definition *calc-spread* :: (*′k*::*finite*) ⇒ *′k point list* ⇒ *real* **where**
  calc-spread k ps = (*case ps of* [] ⇒ *0* | *ps* ⇒
    let ks = *map* ($\lambda p.\ p\$k$) (*tl ps*) *in*
    fold max ks ((*hd ps*)\$*k*) − *fold min ks* ((*hd ps*)\$*k*)
)

fun *widest-spread* :: (*′k*::*finite*) *list* ⇒ *′k point list* ⇒ *′k* × *real* **where**
  widest-spread [] - = *undefined*
| *widest-spread* [*k*] *ps* = (*k*, *calc-spread k ps*)
| *widest-spread* (*k* # *ks*) *ps* = (

let (k', s') = *widest-spread ks ps in*
let s = *calc-spread k ps in*
if $s \leq s'$ *then* (k', s') *else* (k, s)
)

lemma *calc-spread-spec*:
calc-spread k ps = *spread k* (*set ps*)
using *Max.set-eq-fold*[*of* (*hd ps*)$k] *Min.set-eq-fold*[*of* (*hd ps*)$k]
by (*auto simp*: *Let-def spread-def calc-spread-def split*: *list.splits*, *metis set-map*)

lemma *widest-spread-calc-spread*:
ks ≠ [] ⟹ (*k*, *s*) = *widest-spread ks ps* ⟹ *s* = *calc-spread k ps*
by (*induction ks ps rule*: *widest-spread.induct*) (*auto simp*: *Let-def split*: *prod.splits if-splits*)

lemma *widest-spread-axis-Un*:
shows *widest-spread-axis k K P* ⟹ *spread* k' *P* ≤ *spread k P* ⟹ *widest-spread-axis k* ($K \cup \{ k' \}$) *P*
and *widest-spread-axis k K P* ⟹ *spread k P* ≤ *spread* k' *P* ⟹ *widest-spread-axis* k' ($K \cup \{ k' \}$) *P*
unfolding *widest-spread-axis-def* **by** *auto*

lemma *widest-spread-spec*:
(*k*, *s*) = *widest-spread ks ps* ⟹ *widest-spread-axis k* (*set ks*) (*set ps*)
proof (*induction ks ps arbitrary*: *k s rule*: *widest-spread.induct*)
case (*3* k_0 k_1 *ks ps*)
obtain K' S' **where** K'-*def*: (K', S') = *widest-spread* (k_1 # *ks*) *ps*
by (*metis surj-pair*)
hence *IH*: *widest-spread-axis* K' (*set* (k_1 # *ks*)) (*set ps*)
using *3.IH* **by** *blast*
hence *0*: S' = *spread* K' (*set ps*)
using K'-*def widest-spread-calc-spread calc-spread-spec* **by** *blast*
define *S* **where** *S* = *calc-spread* k_0 *ps*
hence *1*: *S* = *spread* k_0 (*set ps*)
using *calc-spread-spec* **by** *blast*
show *?case*
proof (*cases* $S \leq S'$)
case *True*
hence *widest-spread-axis* K' (*set* (k_0 # k_1 # *ks*)) (*set ps*)
using *0 1 widest-spread-axis-Un*(*1*)[*OF IH*, *of* k_0] **by** *auto*
thus *?thesis*
using *True* K'-*def S-def 3.prems* **by** (*auto split*: *prod.splits*)
next
case *False*
hence *widest-spread-axis* k_0 (*set* (k_0 # k_1 # *ks*)) (*set ps*)
using *0 1 widest-spread-axis-Un*(*2*)[*OF IH*, *of* k_0] *3.prems*(*1*) **by** *auto*
thus *?thesis*
using *False* K'-*def S-def 3.prems* **by** (*auto split*: *prod.splits*)
qed

qed (*auto simp*: *widest-spread-axis-def*)

2.3 Fast Axis Median

definition *axis-median* :: $('k::\mathit{finite}) \Rightarrow 'k$ *point list* $\Rightarrow$ *real* **where**
axis-median k ps = (*let n* = (*length ps* − *1*) *div 2 in fast-select n* (*map* ($\lambda p.\ p\$k$) *ps*))

lemma *length-filter-le-axis-median*:
assumes *0* < *length ps* $\forall k$. *distinct* (*map* ($\lambda p.\ p\$k$) *ps*)
shows *length* (*filter* ($\lambda p.\ p\$k \leq$ *axis-median k ps*) *ps*) = (*length ps* − *1*) *div 2* + *1*
proof −
let *?n* = (*length ps* − *1*) *div 2*
let *?ps* = *map* ($\lambda p.\ p\$k$) *ps*
let *?m* = *fast-select ?n ?ps*
have *0*: *?n* < *length ?ps*
using *assms*(*1*) **by** (*auto*, *linarith*)
have *1*: *distinct ?ps*
using *assms*(*2*) **by** *blast*
have *?m* = *select ?n ?ps*
using *fast-select-correct*[*OF 0*] **by** *blast*
hence *length* (*filter* ($\lambda p.\ p\$k \leq$ *axis-median k ps*) *ps*) =
length (*filter* ($\lambda p.\ p\$k \leq$ *sort ?ps* ! *?n*) *ps*)
unfolding *axis-median-def* **by** (*auto simp add*: *Let-def select-def simp del*: *fast-select.simps*)
also have ... = *length* (*filter* ($\lambda v.\ v \leq$ *sort ?ps* ! *?n*) *?ps*)
by (*induction ps*) (*auto*, *metis comp-apply*)
also have ... = *?n* + *1*
using *length-filter-sort-nth*[*OF 1 0*] **by** *blast*
finally show *?thesis* **.**
qed

definition *partition-by-median* :: $('k::\mathit{finite}) \Rightarrow 'k$ *point list* $\Rightarrow 'k$ *point list* $\times$ *real* $\times\ 'k$ *point list* **where**
partition-by-median k ps = (
let m = *axis-median k ps in*
let (*l*, *r*) = *partition* ($\lambda p.\ p\$k \leq m$) *ps in*
(*l*, *m*, *r*)
)

lemma *set-partition-by-median*:
(*l*, *m*, *r*) = *partition-by-median k ps* $\Longrightarrow$ *set ps* = *set l* $\cup$ *set r*
unfolding *partition-by-median-def* **by** (*auto simp*: *Let-def*)

lemma *filter-partition-by-median*:
assumes (*l*, *m*, *r*) = *partition-by-median k ps*
shows $\forall p \in$ *set l*. $p\$k \leq m$
and $\forall p \in$ *set r*. $\neg p\$k \leq m$

using *assms* **unfolding** *partition-by-median-def* **by** (*auto simp*: *Let-def*)

lemma *sum-length-partition-by-median*:
assumes (l, m, r) = *partition-by-median k ps*
shows *length ps* = *length l* + *length r*
using *assms sum-length-filter-compl*[*of* ($\lambda p.$ *p* \$ *k* $\leq$ *axis-median k ps*)]
unfolding *partition-by-median-def* **by** (*simp add*: *Let-def o-def*)

lemma *length-l-partition-by-median*:
assumes *0* < *length ps* $\forall k.$ *distinct* (*map* ($\lambda p.$ *p*\$*k*) *ps*) (l, m, r) = *partition-by-median k ps*
shows *length l* = (*length ps* − *1*) *div 2* + *1*
using *assms* **unfolding** *partition-by-median-def* **by** (*auto simp*: *Let-def length-filter-le-axis-median*)

corollary *lengths-partition-by-median-1*:
assumes *0* < *length ps* $\forall k.$ *distinct* (*map* ($\lambda p.$ *p*\$*k*) *ps*) (l, m, r) = *partition-by-median k ps*
shows *length l* − *length r* $\leq$ *1*
and *length r* $\leq$ *length l*
and *0* < *length l*
and *length r* < *length ps*
using *length-l-partition-by-median*[*OF assms*] *sum-length-partition-by-median*[*OF assms*(*3*)] **by** *auto*

corollary *lengths-partition-by-median-2*:
assumes *1* < *length ps* $\forall k.$ *distinct* (*map* ($\lambda p.$ *p*\$*k*) *ps*) (l, m, r) = *partition-by-median k ps*
shows *0* < *length r*
and *length l* < *length ps*
proof −
have ∗: *0* < *length ps*
using *assms*(*1*) **by** *auto*
show *0* < *length r length l* < *length ps*
using *length-l-partition-by-median*[*OF* ∗ *assms*(*2*,*3*)] *sum-length-partition-by-median*[*OF assms*(*3*)]
using *assms*(*1*) **by** *linarith*+
qed

lemmas *length-partition-by-median* =
sum-length-partition-by-median length-l-partition-by-median
lengths-partition-by-median-1 lengths-partition-by-median-2

2.4 Building the Tree

function (*domintros*, *sequential*) *build* :: (*'k*::*finite*) *list* ⇒ *'k point list* ⇒ *'k kdt*
where
build - [] = *undefined*
| *build* - [*p*] = *Leaf p*
| *build ks ps* = (

```
    let (k, -) = widest-spread ks ps in
    let (l, m, r) = partition-by-median k ps in
    Node k m (build ks l) (build ks r)
  )
  by pat-completeness auto

lemma build-domintros3:
  assumes (k, s) = widest-spread ks (x # y # zs) (l, m, r) = partition-by-median
k (x # y # zs)
  assumes build-dom (ks, l) build-dom (ks, r)
  shows build-dom (ks, x # y # zs)
proof −
  {
    fix k s l m r
    assume (k, s) = widest-spread ks (x # y # zs) (l, m, r) = partition-by-median
k (x # y # zs)
    hence build-dom (ks, l) build-dom (ks, r)
      using assms by (metis Pair-inject)+
  }
  thus ?thesis
    by (simp add: build.domintros(3))
qed

lemma build-termination:
  assumes ∀ k. distinct (map (λp. p$k) ps)
  shows build-dom (ks, ps)
  using assms
proof (induction ps rule: length-induct)
  case (1 xs)
  consider (A) xs = [] | (B) ∃ x. xs = [x] | (C) ∃ x y zs. xs = x # y # zs
    by (induction xs rule: induct-list012) auto
  then show ?case
  proof cases
    case C
    then obtain x y zs where xyzs-def: xs = x # y # zs
      by blast
    obtain k s where ks-def: (k, s) = widest-spread ks xs
      by (metis surj-pair)
    obtain l m r where lmr-def: (l, m, r) = partition-by-median k xs
      by (metis prod-cases3)
    note defs = xyzs-def ks-def lmr-def
    have ∀ k. distinct (map (λp. p $ k) l) ∀ k. distinct (map (λp. p $ k) r)
      using lmr-def unfolding partition-by-median-def
      by (auto simp: Let-def 1.prems distinct-map-filter)
    moreover have length l < length xs length r < length xs
     using length-partition-by-median(8)[OF - 1.prems] length-partition-by-median(6)[OF
- 1.prems]
      using defs by auto
    ultimately have build-dom (ks, l) build-dom (ks, r)
```

using *1.IH* **by** *blast+*
thus *?thesis*
using *build-domintros3 defs* **by** *blast*
qed (*auto intro*: *build.domintros*)
qed

lemma *build-psimp-1*:
ps = [*p*] $\Longrightarrow$ *build k ps* = *Leaf p*
by (*simp add*: *build.domintros*(*2*) *build.psimps*(*2*))

lemma *build-psimp-2*:
assumes (*k*, *s*) = *widest-spread ks* (*x* # *y* # *zs*) (*l*, *m*, *r*) = *partition-by-median k* (*x* # *y* # *zs*)
assumes *build-dom* (*ks*, *l*) *build-dom* (*ks*, *r*)
shows *build ks* (*x* # *y* # *zs*) = *Node k m* (*build ks l*) (*build ks r*)
proof −
have *0*: *build-dom* (*ks*, *x* # *y* # *zs*)
using *assms build-domintros3* **by** *blast*
thus *?thesis*
using *build.psimps*(*3*)[*OF 0*] *assms*(*1*,*2*) **by** (*auto split*: *prod.splits*)
qed

lemma *length-xs-gt-1*:
1 < *length xs* $\Longrightarrow$ $\exists$ *x y ys*. *xs* = *x* # *y* # *ys*
by (*cases xs*, *auto simp*: *neq-Nil-conv*)

lemma *build-psimp-3*:
assumes *1* < *length ps* (*k*, *s*) = *widest-spread ks ps* (*l*, *m*, *r*) = *partition-by-median k ps*
assumes *build-dom* (*ks*, *l*) *build-dom* (*ks*, *r*)
shows *build ks ps* = *Node k m* (*build ks l*) (*build ks r*)
using *build-psimp-2 length-xs-gt-1 assms* **by** *blast*

lemmas *build-psimps*[*simp*] = *build-psimp-1 build-psimp-3*

2.5 Main Theorems

theorem *set-build*:
0 < *length ps* $\Longrightarrow$ $\forall$ *k*. *distinct* (*map* (λ*p*. *p*$*k*) *ps*) $\Longrightarrow$ *set ps* = *set-kdt* (*build ks ps*)
proof (*induction ps rule*: *length-induct*)
case (*1 ps*)
show *?case*
proof (*cases 1* < *length ps*)
case *True*
obtain *k s* **where** *ks-def*: (*k*, *s*) = *widest-spread ks ps*
by (*metis surj-pair*)
obtain *l m r* **where** *lmr-def*: (*l*, *m*, *r*) = *partition-by-median k ps*
by (*metis prod-cases3*)

```
    have D: ∀k. distinct (map (λp. p$k) l) ∀k. distinct (map (λp. p$k) r)
      using lmr-def unfolding partition-by-median-def
      by (auto simp: 1.prems(2) Let-def distinct-map-filter)
    moreover have length l < length ps 0 < length l
                  length r < length ps 0 < length r
      using length-partition-by-median(8)[OF True 1.prems(2)]
            length-partition-by-median(5)[OF 1.prems(1) 1.prems(2)]
            length-partition-by-median(6)[OF 1.prems(1) 1.prems(2)]
            length-partition-by-median(7)[OF True 1.prems(2)]
            lmr-def by blast+
    ultimately have set l = set-kdt (build ks l) set r = set-kdt (build ks r)
      using 1.IH by blast+
    moreover have set ps = set l ∪ set r
      using lmr-def unfolding partition-by-median-def by (auto simp: Let-def)
    moreover have build ks ps = Node k m (build ks l) (build ks r)
      using build-psimp-3[OF True ks-def lmr-def] build-termination D by blast
    ultimately show ?thesis
      by simp
  next
    case False
    thus ?thesis
      using 1.prems by (cases ps) auto
  qed
qed

theorem invar-build:
  0 < length ps ⟹ ∀k. distinct (map (λp. p$k) ps) ⟹ set ks = UNIV ⟹ invar
(build ks ps)
proof (induction ps rule: length-induct)
  case (1 ps)
  show ?case
  proof (cases 1 < length ps)
    case True
    obtain k s where ks-def: (k, s) = widest-spread ks ps
      by (metis surj-pair)
    obtain l m r where lmr-def: (l, m, r) = partition-by-median k ps
      by (metis prod-cases3)
    have D: ∀k. distinct (map (λp. p$k) l) ∀k. distinct (map (λp. p$k) r)
      using lmr-def unfolding partition-by-median-def
      by (auto simp: 1.prems(2) Let-def distinct-map-filter)
    moreover have length l < length ps 0 < length l
                  length r < length ps 0 < length r
      using length-partition-by-median(8)[OF True 1.prems(2)]
            length-partition-by-median(5)[OF 1.prems(1) 1.prems(2)]
            length-partition-by-median(6)[OF 1.prems(1) 1.prems(2)]
            length-partition-by-median(7)[OF True 1.prems(2)]
            lmr-def by blast+
    ultimately have invar (build ks l) invar (build ks r)
      using 1.IH 1.prems(3) by blast+
```

moreover have $\forall p \in set\ l.\ p\$k \leq m$ $\forall p \in set\ r.\ m < p\$k$
using *filter-partition-by-median*(*1*)[*OF lmr-def*]
filter-partition-by-median(*2*)[*OF lmr-def*] **by** *auto*
moreover have *widest-spread-axis k UNIV* (*set l* ∪ *set r*)
using *widest-spread-spec*[*OF ks-def*] *1.prems*(*3*) *set-partition-by-median*[*OF lmr-def*] **by** *simp*
moreover have *build ks ps* = *Node k m* (*build ks l*) (*build ks r*)
using *build-psimp-3*[*OF True ks-def lmr-def*] *build-termination D* **by** *blast*
ultimately show *?thesis*
using *set-build*[*OF* ‹*0* < *length l*› *D*(*1*)] *set-build*[*OF* ‹*0* < *length r*› *D*(*2*)] **by** *simp*
next
case *False*
thus *?thesis*
using *1.prems* **by** (*cases ps*) *auto*
qed
qed

theorem *size-build*:
0 < *length ps* ⟹ ∀ *k. distinct* (*map* (λ*p. p*\$*k*) *ps*) ⟹ *size-kdt* (*build ks ps*) = *length ps*
proof (*induction ps rule*: *length-induct*)
case (*1 ps*)
show *?case*
proof (*cases 1* < *length ps*)
case *True*
obtain *k s* **where** *ks-def*: (*k*, *s*) = *widest-spread ks ps*
by (*metis surj-pair*)
obtain *l m r* **where** *lmr-def*: (*l*, *m*, *r*) = *partition-by-median k ps*
by (*metis prod-cases3*)
have *D*: ∀ *k. distinct* (*map* (λ*p. p*\$*k*) *l*) ∀ *k. distinct* (*map* (λ*p. p*\$*k*) *r*)
using *lmr-def* **unfolding** *partition-by-median-def*
by (*auto simp*: *1.prems*(*2*) *Let-def distinct-map-filter*)
moreover have *length l* < *length ps* *0* < *length l*
length r < *length ps* *0* < *length r*
using *length-partition-by-median*(*8*)[*OF True 1.prems*(*2*)]
length-partition-by-median(*5*)[*OF 1.prems*(*1*) *1.prems*(*2*)]
length-partition-by-median(*6*)[*OF 1.prems*(*1*) *1.prems*(*2*)]
length-partition-by-median(*7*)[*OF True 1.prems*(*2*)]
lmr-def **by** *blast+*
ultimately have *size-kdt* (*build ks l*) = *length l* *size-kdt* (*build ks r*) = *length r*
using *1.IH* **by** *blast+*
moreover have *build ks ps* = *Node k m* (*build ks l*) (*build ks r*)
using *build-psimp-3*[*OF True ks-def lmr-def*] *build-termination D* **by** *blast*
ultimately show *?thesis*
using *length-partition-by-median*(*1*)[*OF lmr-def*] **by** *simp*
next
case *False*
thus *?thesis*

using *1.prems* **by** (*cases ps*) *auto*
qed
qed

theorem *balanced-build*:
0 < length ps ⟹ ∀k. distinct (map (λp. p$k) ps) ⟹ balanced (build ks ps)
proof (*induction ps rule*: *length-induct*)
case (*1 ps*)
show *?case*
proof (*cases 1 < length ps*)
case *True*
obtain *k s* **where** *ks-def*: (*k, s*) = *widest-spread ks ps*
by (*metis surj-pair*)
obtain *l m r* **where** *lmr-def*: (*l, m, r*) = *partition-by-median k ps*
by (*metis prod-cases3*)
have *D*: *∀k. distinct (map (λp. p$k) l) ∀k. distinct (map (λp. p$k) r)*
using *lmr-def* **unfolding** *partition-by-median-def*
by (*auto simp*: *1.prems(2) Let-def distinct-map-filter*)
moreover have *length l < length ps 0 < length l*
length r < length ps 0 < length r
using *length-partition-by-median(8)[OF True 1.prems(2)]*
length-partition-by-median(5)[OF 1.prems(1) 1.prems(2)]
length-partition-by-median(6)[OF 1.prems(1) 1.prems(2)]
length-partition-by-median(7)[OF True 1.prems(2)]
lmr-def **by** *blast+*
ultimately have *IH*: *balanced (build ks l) balanced (build ks r)*
using *1.IH* **by** *blast+*
have *build ks ps = Node k m (build ks l) (build ks r)*
using *build-psimp-3[OF True ks-def lmr-def] build-termination D* **by** *blast*
moreover have *length r + 1 = length l ∨ length r = length l*
using *length-partition-by-median(1)[OF lmr-def]*
length-partition-by-median(3)[OF 1.prems(1) 1.prems(2) lmr-def]
length-partition-by-median(4)[OF 1.prems(1) 1.prems(2) lmr-def]
by *linarith*
ultimately show *?thesis*
using *balanced-Node-if-wbal1[OF IH] balanced-Node-if-wbal2[OF IH]*
size-build[OF ‹0 < length l› D(1)] size-build[OF ‹0 < length r› D(2)]
by *auto*
next
case *False*
thus *?thesis*
using *1.prems* **by** (*cases ps*) (*auto simp*: *balanced-def*)
qed
qed

lemma *complete-if-balanced-size-2powh*:
assumes *balanced kdt size-kdt kdt = 2 ^ h*
shows *complete kdt*
proof (*rule ccontr*)

assume ¬ *complete kdt*
hence $2 \,\hat{}\, (min\text{-}height\ kdt) < size\text{-}kdt\ kdt\ size\text{-}kdt\ kdt < 2 \,\hat{}\, height\ kdt$
by (*simp-all add*: *min-height-size-if-incomplete size-height-if-incomplete*)
hence $height\ kdt - min\text{-}height\ kdt > 1$
using *assms*(*2*) **by** *simp*
hence ¬ *balanced kdt*
using *balanced-def* **by** *force*
thus *False*
using *assms*(*1*) **by** *simp*
qed

theorem *complete-build*:
$length\ ps = 2 \,\hat{}\, h \Longrightarrow \forall k.\ distinct\ (map\ (\lambda p.\ p\$k)\ ps) \Longrightarrow complete\ (build\ k\ ps)$
by (*simp add*: *balanced-build complete-if-balanced-size-2powh size-build*)

corollary *height-build*:
assumes $length\ ps = 2 \,\hat{}\, h\ \forall k.\ distinct\ (map\ (\lambda p.\ p\$k)\ ps)$
shows $h = height\ (build\ k\ ps)$
using *complete-build*[*OF assms*] *size-build*[*OF - assms*(*2*)] **by** (*simp add*: *assms*(*1*) *complete-iff-size*)

end

3 Range Searching

theory *Range-Search*
imports
KD-Tree
begin

Given two k-dimensional points p_0 and p_1 which bound the search space, the search should return only the points which satisfy the following criteria:

For every point p in the resulting set:
For every axis k:
$p_0\ \$\ k \leq p\ \$\ k \wedge p\ \$\ k \leq p_1\ \$\ k$

For a *2*-d tree a query corresponds to selecting all the points in the rectangle that has p_0 and p_1 as its defining edges.

3.1 Rectangle Definition

lemma *cbox-point-def*:
fixes p_0 :: (*'k::finite*) *point*
shows $cbox\ p_0\ p_1 = \{\ p.\ \forall k.\ p_0\$k \leq p\$k \wedge p\$k \leq p_1\$k\ \}$
proof −
have $cbox\ p_0\ p_1 = \{\ p.\ \forall k.\ p_0 \cdot axis\ k\ 1 \leq p \cdot axis\ k\ 1 \wedge p \cdot axis\ k\ 1 \leq p_1 \cdot axis\ k\ 1\ \}$

unfolding *cbox-def* **using** *axis-inverse* **by** *auto*
also have ... = { $p.\ \forall k.\ p_0\$k \cdot 1 \leq p\$k \cdot 1 \wedge p\$k \cdot 1 \leq p_1\$k \cdot 1$ }
using *inner-axis*[*of - - 1*] **by** (*smt Collect-cong*)
also have ... = { $p.\ \forall k.\ p_0\$k \leq p\$k \wedge p\$k \leq p_1\$k$ }
by *simp*
finally show *?thesis* **.**
qed

3.2 Search Function

fun *search* :: (*'k::finite*) *point* ⇒ *'k point* ⇒ *'k kdt* ⇒ *'k point set* **where**
search p_0 p_1 (*Leaf p*) = (*if p* ∈ *cbox* p_0 p_1 *then* { *p* } *else* {})
| *search* p_0 p_1 (*Node k v l r*) = (
if $v < p_0\$k$ *then*
search p_0 p_1 *r*
else if $p_1\$k < v$ *then*
search p_0 p_1 *l*
else
search p_0 p_1 *l* ∪ *search* p_0 p_1 *r*
)

3.3 Auxiliary Lemmas

lemma *l-empty*:
assumes *invar* (*Node k v l r*) $v < p_0\$k$
shows *set-kdt l* ∩ *cbox* p_0 p_1 = {}
proof −
have $\forall p \in$ *set-kdt l*. $p\$k < p_0\$k$
using *assms* **by** *auto*
hence $\forall p \in$ *set-kdt l*. $p \notin$ *cbox* p_0 p_1
using *cbox-point-def leD* **by** *blast*
thus *?thesis* **by** *blast*
qed

lemma *r-empty*:
assumes *invar* (*Node k v l r*) $p_1\$k < v$
shows *set-kdt r* ∩ *cbox* p_0 p_1 = {}
proof −
have $\forall p \in$ *set-kdt r*. $p_1\$k < p\$k$
using *assms* **by** *auto*
hence $\forall p \in$ *set-kdt r*. $p \notin$ *cbox* p_0 p_1
using *cbox-point-def leD* **by** *blast*
thus *?thesis* **by** *blast*
qed

3.4 Main Theorem

theorem *search-cbox*:
assumes *invar kdt*
shows *search* p_0 p_1 *kdt* = *set-kdt kdt* ∩ *cbox* p_0 p_1

using *assms l-empty r-empty* **by** (*induction kdt*) (*auto*, *blast+*)

end

4 Nearest Neighbor Search on the k-d Tree

theory *Nearest-Neighbors*
imports
KD-Tree
begin

Verifying nearest neighbor search on the k-d tree. Given a k-d tree and a point p, which might not be in the tree, find the points ps that are closest to p using the Euclidean metric.

4.1 Auxiliary Lemmas about *sorted-wrt*

lemma
assumes *sorted-wrt f xs*
shows *sorted-wrt-take*: *sorted-wrt f* (*take n xs*)
and *sorted-wrt-drop*: *sorted-wrt f* (*drop n xs*)
proof −
have *sorted-wrt f* (*take n xs* @ *drop n xs*)
using *assms* **by** *simp*
thus *sorted-wrt f* (*take n xs*) *sorted-wrt f* (*drop n xs*)
using *sorted-wrt-append* **by** *blast+*
qed

definition *sorted-wrt-dist* :: ($'k$::*finite*) *point* ⇒ $'k$ *point list* ⇒ *bool* **where**
sorted-wrt-dist $p \equiv$ *sorted-wrt* ($\lambda p_0\ p_1.\ dist\ p_0\ p \leq dist\ p_1\ p$)

lemma *sorted-wrt-dist-insort-key*:
sorted-wrt-dist p ps ⟹ *sorted-wrt-dist p* (*insort-key* ($\lambda q.\ dist\ q\ p$) *q ps*)
by (*induction ps*) (*auto simp*: *sorted-wrt-dist-def set-insort-key*)

lemma *sorted-wrt-dist-take-drop*:
assumes *sorted-wrt-dist p ps*
shows $\forall p_0 \in set\ (take\ n\ ps).\ \forall p_1 \in set\ (drop\ n\ ps).\ dist\ p_0\ p \leq dist\ p_1\ p$
using *assms sorted-wrt-append*[*of - take n ps drop n ps*] **by** (*simp add*: *sorted-wrt-dist-def*)

lemma *sorted-wrt-dist-last-take-mono*:
assumes *sorted-wrt-dist p ps* $n \leq$ *length ps* $0 < n$
shows *dist* (*last* (*take n ps*)) $p \leq$ *dist* (*last ps*) *p*
using *assms* **unfolding** *sorted-wrt-dist-def* **by** (*induction ps arbitrary*: *n*) (*auto simp add*: *take-Cons'*)

lemma *sorted-wrt-dist-last-insort-key-eq*:
assumes *sorted-wrt-dist p ps insort-key* ($\lambda q.\ dist\ q\ p$) *q ps* $\neq$ *ps* @ [*q*]

shows *last* (*insort-key* ($\lambda q.$ *dist q p*) *q ps*) = *last ps*
using *assms* **unfolding** *sorted-wrt-dist-def* **by** (*induction ps*) (*auto*)

lemma *sorted-wrt-dist-last*:
assumes *sorted-wrt-dist p ps*
shows $\forall$ *q* $\in$ *set ps. dist q p* $\leq$ *dist* (*last ps*) *p*
proof (*cases ps* = [])
case *True*
thus *?thesis* **by** *simp*
next
case *False*
then obtain *ps' p'* **where** [*simp*]:*ps* = *ps'* @ [*p'*]
using *rev-exhaust* **by** *blast*
hence *sorted-wrt-dist p* (*ps'* @ [*p'*])
using *assms* **by** *blast*
thus *?thesis*
unfolding *sorted-wrt-dist-def* **using** *sorted-wrt-append*[*of - ps'* [*p'*]] **by** *simp*
qed

4.2 Neighbors Sorted wrt. Distance

definition *upd-nbors* :: *nat* $\Rightarrow$ (*'k*::*finite*) *point* $\Rightarrow$ *'k point* $\Rightarrow$ *'k point list* $\Rightarrow$ *'k point list* **where**
upd-nbors n p q ps = *take n* (*insort-key* ($\lambda q.$ *dist q p*) *q ps*)

lemma *sorted-wrt-dist-nbors*:
assumes *sorted-wrt-dist p ps*
shows *sorted-wrt-dist p* (*upd-nbors n p q ps*)
proof −
have *sorted-wrt-dist p* (*insort-key* ($\lambda q.$ *dist q p*) *q ps*)
using *assms sorted-wrt-dist-insort-key* **by** *blast*
thus *?thesis*
by (*simp add*: *sorted-wrt-dist-def sorted-wrt-take upd-nbors-def*)
qed

lemma *sorted-wrt-dist-nbors-diff*:
assumes *sorted-wrt-dist p ps*
shows $\forall$ *r* $\in$ *set ps* $\cup$ {*q*} − *set* (*upd-nbors n p q ps*). $\forall$ *s* $\in$ *set* (*upd-nbors n p q ps*). *dist s p* $\leq$ *dist r p*
proof −
let *?ps'* = *insort-key* ($\lambda q.$ *dist q p*) *q ps*
have *set ps* $\cup$ { *q* } = *set ?ps'*
by (*simp add*: *set-insort-key*)
moreover have *set ?ps'* = *set* (*take n ?ps'*) $\cup$ *set* (*drop n ?ps'*)
using *append-take-drop-id set-append* **by** *metis*
ultimately have *set ps* $\cup$ { *q* } − *set* (*take n ?ps'*) $\subseteq$ *set* (*drop n ?ps'*)
by *blast*
moreover have *sorted-wrt-dist p ?ps'*
using *assms sorted-wrt-dist-insort-key* **by** *blast*

```
  ultimately show ?thesis
    unfolding upd-nbors-def using sorted-wrt-dist-take-drop by blast
qed

lemma sorted-wrt-dist-last-upd-nbors-mono:
  assumes sorted-wrt-dist p ps n ≤ length ps 0 < n
  shows dist (last (upd-nbors n p q ps)) p ≤ dist (last ps) p
proof (cases insort-key (λq. dist q p) q ps = ps @ [q])
  case True
  thus ?thesis
    unfolding upd-nbors-def using assms sorted-wrt-dist-last-take-mono by auto
next
  case False
  hence last (insort-key (λq. dist q p) q ps) = last ps
    using sorted-wrt-dist-last-insort-key-eq assms by blast
  moreover have dist (last (upd-nbors n p q ps)) p ≤ dist (last (insort-key (λq.
dist q p) q ps)) p
     unfolding upd-nbors-def using assms sorted-wrt-dist-last-take-mono[of p in-
sort-key (λq. dist q p) q ps]
    by (simp add: sorted-wrt-dist-insort-key)
  ultimately show ?thesis
    by simp
qed
```

4.3 The Recursive Nearest Neighbor Algorithm

```
fun nearest-nbors :: nat ⇒ ('k::finite) point list ⇒ 'k point ⇒ 'k kdt ⇒ 'k point
list where
  nearest-nbors n ps p (Leaf q) = upd-nbors n p q ps
| nearest-nbors n ps p (Node k v l r) = (
    if p$k ≤ v then
      let candidates = nearest-nbors n ps p l in
      if length candidates = n ∧ dist p (last candidates) ≤ dist v (p$k) then
        candidates
      else
        nearest-nbors n candidates p r
    else
      let candidates = nearest-nbors n ps p r in
      if length candidates = n ∧ dist p (last candidates) ≤ dist v (p$k) then
        candidates
      else
        nearest-nbors n candidates p l
  )
```

4.4 Auxiliary Lemmas

```
lemma cutoff-r:
  assumes invar (Node k v l r)
  assumes p$k ≤ v dist p c ≤ dist (p$k) v
  shows ∀ q ∈ set-kdt r. dist p c ≤ dist p q
```

proof *standard*
  fix *q*
  assume $*$: $q \in$ *set-kdt r*
  have *dist p c* $\leq$ *dist* $(p\$k)$ *v*
    using *assms*(3) **by** *blast*
  also have ... $\leq$ *dist* $(p\$k)$ *v* $+$ *dist v* $(q\$k)$
    by *simp*
  also have ... $=$ *dist* $(p\$k)$ $(q\$k)$
    using $*$ *assms*$(1,2)$ *dist-real-def* **by** *auto*
  also have ... $\leq$ *dist p q*
    using *dist-vec-nth-le* **by** *blast*
  finally show *dist p c* $\leq$ *dist p q* **.**
qed

lemma *cutoff-l*:
  assumes *invar* (*Node k v l r*)
  assumes $v \leq p\$k$ *dist p c* $\leq$ *dist v* $(p\$k)$
  shows $\forall$ $q \in$ *set-kdt l*. *dist p c* $\leq$ *dist p q*
proof *standard*
  fix *q*
  assume $*$: $q \in$ *set-kdt l*
  have *dist p c* $\leq$ *dist v* $(p\$k)$
    using *assms*(3) **by** *blast*
  also have ... $\leq$ *dist v* $(p\$k)$ $+$ *dist* $(q\$k)$ *v*
    by *simp*
  also have ... $=$ *dist* $(p\$k)$ $(q\$k)$
    using $*$ *assms*$(1,2)$ *dist-real-def* **by** *auto*
  also have ... $\leq$ *dist p q*
    using *dist-vec-nth-le* **by** *blast*
  finally show *dist p c* $\leq$ *dist p q* **.**
qed

4.5 The Main Theorems

lemma *set-nns*:
  set (*nearest-nbors n ps p kdt*) $\subseteq$ *set-kdt kdt* $\cup$ *set ps*
  apply (*induction kdt arbitrary*: *ps*)
  apply (*auto simp*: *Let-def upd-nbors-def set-insort-key*)
  using *in-set-takeD set-insort-key* **by** *fastforce*

lemma *length-nns*:
  length (*nearest-nbors n ps p kdt*) $=$ *min n* (*size-kdt kdt* $+$ *length ps*)
  by (*induction kdt arbitrary*: *ps*) (*auto simp*: *Let-def upd-nbors-def*)

lemma *length-nns-gt-0*:
  $0 < n \Longrightarrow 0 <$ *length* (*nearest-nbors n ps p kdt*)
  by (*induction kdt arbitrary*: *ps*) (*auto simp*: *Let-def upd-nbors-def*)

lemma *length-nns-n*:

```
  assumes (set-kdt kdt ∪ set ps) − set (nearest-nbors n ps p kdt) ≠ {}
  shows length (nearest-nbors n ps p kdt) = n
  using assms
proof (induction kdt arbitrary: ps)
  case (Node k v l r)
  let ?nnsl = nearest-nbors n ps p l
  let ?nnsr = nearest-nbors n ps p r
  consider (A) p$k ≤ v ∧ length ?nnsl = n ∧ dist p (last ?nnsl) ≤ dist v (p$k)
        | (B) p$k ≤ v ∧ ¬(length ?nnsl = n ∧ dist p (last ?nnsl) ≤ dist v (p$k))
        | (C) v < p$k ∧ length ?nnsr = n ∧ dist p (last ?nnsr) ≤ dist v (p$k)
        | (D) v < p$k ∧ ¬(length ?nnsr = n ∧ dist p (last ?nnsr) ≤ dist v (p$k))
    by argo
  thus ?case
  proof cases
    case B
    let ?nns = nearest-nbors n ?nnsl p r
    have length ?nnsl ≠ n ⟶ (set-kdt l ∪ set ps − set (nearest-nbors n ps p l) =
{})
      using Node.IH(1) by blast
    hence length ?nnsl ≠ n ⟶ (set-kdt r ∪ set ?nnsl − set ?nns ≠ {})
      using B Node.prems by auto
    moreover have length ?nnsl = n ⟶ ?thesis
      using B by (auto simp: length-nns)
    ultimately show ?thesis
      using B Node.IH(2) by force
  next
    case D
    let ?nns = nearest-nbors n ?nnsr p l
    have length ?nnsr ≠ n ⟶ (set-kdt r ∪ set ps − set (nearest-nbors n ps p r)
= {})
      using Node.IH(2) by blast
    hence length ?nnsr ≠ n ⟶ (set-kdt l ∪ set ?nnsr − set ?nns ≠ {})
      using D Node.prems by auto
    moreover have length ?nnsr = n ⟶ ?thesis
      using D by (auto simp: length-nns)
    ultimately show ?thesis
      using D Node.IH(1) by force
  qed auto
qed (auto simp: upd-nbors-def min-def set-insort-key)

lemma sorted-nns:
  sorted-wrt-dist p ps ⟹ sorted-wrt-dist p (nearest-nbors n ps p kdt)
  using sorted-wrt-dist-nbors by (induction kdt arbitrary: ps) (auto simp: Let-def)

lemma distinct-nns:
  assumes invar kdt distinct ps set ps ∩ set-kdt kdt = {}
  shows distinct (nearest-nbors n ps p kdt)
  using assms
proof (induction kdt arbitrary: ps)
```

case (*Node k v l r*)
let *?nnsl* = *nearest-nbors n ps p l*
let *?nnsr* = *nearest-nbors n ps p r*
have *set ps* ∩ *set-kdt l* = {} *set ps* ∩ *set-kdt r* = {}
using *Node.prems*(*3*) **by** *auto*
hence *DCLR*: *distinct ?nnsl distinct ?nnsr*
using *Node invar-l invar-r* **by** *blast+*
have *set ?nnsl* ∩ *set-kdt r* = {} *set ?nnsr* ∩ *set-kdt l* = {}
using *Node.prems*(*1*,*3*) *set-nns* **by** *fastforce+*
hence *distinct* (*nearest-nbors n ?nnsl p r*) *distinct* (*nearest-nbors n ?nnsr p l*)
using *Node.IH*(*1*,*2*) *Node.prems*(*1*,*2*) *DCLR invar-l invar-r* **by** *blast+*
thus *?case*
using *DCLR* **by** (*auto simp add*: *Let-def*)
qed (*auto simp*: *upd-nbors-def distinct-insort*)

lemma *last-nns-mono*:
assumes *invar kdt sorted-wrt-dist p ps n* ≤ *length ps 0* < *n*
shows *dist* (*last* (*nearest-nbors n ps p kdt*)) *p* ≤ *dist* (*last ps*) *p*
using *assms*
proof (*induction kdt arbitrary*: *ps*)
case (*Node k v l r*)
let *?nnsl* = *nearest-nbors n ps p l*
let *?nnsr* = *nearest-nbors n ps p r*
have *n* ≤ *length ?nnsl n* ≤ *length ?nnsr*
using *Node.prems*(*3*) **by** (*simp-all add*: *length-nns*)
hence *dist* (*last* (*nearest-nbors n ?nnsl p r*)) *p* ≤ *dist* (*last ?nnsl*) *p*
dist (*last* (*nearest-nbors n ?nnsr p l*)) *p* ≤ *dist* (*last ?nnsr*) *p*
using *sorted-nns Node invar-l invar-r* **by** *blast+*
hence *dist* (*last* (*nearest-nbors n ?nnsl p r*)) *p* ≤ *dist* (*last ps*) *p*
dist (*last* (*nearest-nbors n ?nnsr p l*)) *p* ≤ *dist* (*last ps*) *p*
using *Node.IH*(*1*)[*of ps*] *Node.IH*(*2*)[*of ps*] *Node.prems invar-l length-nns-gt-0*
by *auto*
thus *?case*
using *Node* **by** (*auto simp add*: *Let-def*)
qed (*auto simp*: *sorted-wrt-dist-last-upd-nbors-mono*)

theorem *dist-nns*:
assumes *invar kdt sorted-wrt-dist p ps set ps* ∩ *set-kdt kdt* = {} *distinct ps 0* < *n*
shows ∀ *q* ∈ *set-kdt kdt* ∪ *set ps* − *set* (*nearest-nbors n ps p kdt*). *dist* (*last* (*nearest-nbors n ps p kdt*)) *p* ≤ *dist q p*
using *assms*
proof (*induction kdt arbitrary*: *ps*)
case (*Node k v l r*)

let *?nnsl* = *nearest-nbors n ps p l*
let *?nnsr* = *nearest-nbors n ps p r*

have *IHL*: ∀ *q* ∈ *set-kdt l* ∪ *set ps* − *set ?nnsl*. *dist* (*last ?nnsl*) *p* ≤ *dist q p*

using *Node.IH*(*1*) *Node.prems invar-l invar-set* **by** *auto*
have *IHR*: $\forall\, q \in$ *set-kdt r* $\cup$ *set ps* $-$ *set ?nnsr*. *dist* (*last ?nnsr*) *p* $\leq$ *dist q p*
using *Node.IH*(*2*) *Node.prems invar-r invar-set* **by** *auto*

have *SORTED-L*: *sorted-wrt-dist p ?nnsl*
using *sorted-nns Node.prems*(*2*) **by** *blast*
have *SORTED-R*: *sorted-wrt-dist p ?nnsr*
using *sorted-nns Node.prems*(*2*) **by** *blast*

have *DISTINCT-L*: *distinct ?nnsl*
using *Node.prems distinct-nns invar-set invar-l* **by** *fastforce*
have *DISTINCT-R*: *distinct ?nnsr*
using *Node.prems distinct-nns invar-set invar-r*
by (*metis inf-bot-right inf-sup-absorb inf-sup-aci*(*3*) *sup.commute*)

consider (*A*) *p*\$*k* $\leq$ *v* $\wedge$ *length ?nnsl* $=$ *n* $\wedge$ *dist p* (*last ?nnsl*) $\leq$ *dist v* (*p*\$*k*)
| (*B*) *p*\$*k* $\leq$ *v* $\wedge$ $\neg$(*length ?nnsl* $=$ *n* $\wedge$ *dist p* (*last ?nnsl*) $\leq$ *dist v* (*p*\$*k*))
| (*C*) *v* $<$ *p*\$*k* $\wedge$ *length ?nnsr* $=$ *n* $\wedge$ *dist p* (*last ?nnsr*) $\leq$ *dist v* (*p*\$*k*)
| (*D*) *v* $<$ *p*\$*k* $\wedge$ $\neg$(*length ?nnsr* $=$ *n* $\wedge$ *dist p* (*last ?nnsr*) $\leq$ *dist v* (*p*\$*k*))
by *argo*
thus *?case*
proof *cases*
case *A*
hence $\forall\, q \in$ *set-kdt r*. *dist* (*last ?nnsl*) *p* $\leq$ *dist q p*
using *Node.prems*(*1*,*2*) *cutoff-r* **by** (*metis dist-commute*)
thus *?thesis*
using *IHL A* **by** *auto*
next
case *B*

let *?nns* $=$ *nearest-nbors n ?nnsl p r*

have *set ?nnsl* $\subseteq$ *set-kdt l* $\cup$ *set ps set ps* $\cap$ *set-kdt r* $=$ {}
using *set-nns Node.prems*(*1*,*3*) **by** (*simp add*: *set-nns disjoint-iff-not-equal*)+
hence *set ?nnsl* $\cap$ *set-kdt r* $=$ {}
using *Node.prems*(*1*) **by** *fastforce*
hence *IHLR*: $\forall\, q \in$ *set-kdt r* $\cup$ *set ?nnsl* $-$ *set ?nns*. *dist* (*last ?nns*) *p* $\leq$ *dist*
q p
using *Node.IH*(*2*)[*OF - SORTED-L - DISTINCT-L Node.prems*(*5*)] *Node.prems*(*1*)
invar-r **by** *blast*

have $\forall\, q \in$ *set ps* $-$ *set ?nnsl*. *dist* (*last ?nns*) *p* $\leq$ *dist q p*
proof *standard*
fix *q*
assume $*$: *q* $\in$ *set ps* $-$ *set ?nnsl*

hence *length ?nnsl* $=$ *n*
using *length-nns-n* **by** *blast*
hence *LAST*: *dist* (*last ?nns*) *p* $\leq$ *dist* (*last ?nnsl*) *p*

using *last-nns-mono SORTED-L invar-r Node.prems*(*1*,*2*,*5*) **by** (*metis order-refl*)
have *dist* (*last ?nnsl*) *p* ≤ *dist q p*
using *IHL* ∗ **by** *blast*
thus *dist* (*last ?nns*) *p* ≤ *dist q p*
using *LAST* **by** *argo*
qed
hence *R*: ∀ *q* ∈ *set-kdt r* ∪ *set ps* − *set ?nns*. *dist* (*last ?nns*) *p* ≤ *dist q p*
using *IHLR* **by** *auto*

have ∀ *q* ∈ *set-kdt l* − *set ?nnsl*. *dist* (*last ?nns*) *p* ≤ *dist q p*
proof *standard*
fix *q*
assume ∗: *q* ∈ *set-kdt l* − *set ?nnsl*

hence *length ?nnsl* = *n*
using *length-nns-n* **by** *blast*
hence *LAST*: *dist* (*last ?nns*) *p* ≤ *dist* (*last ?nnsl*) *p*
using *last-nns-mono SORTED-L invar-r Node.prems*(*1*,*2*,*5*) **by** (*metis order-refl*)
have *dist* (*last ?nnsl*) *p* ≤ *dist q p*
using *IHL* ∗ **by** *blast*
thus *dist* (*last ?nns*) *p* ≤ *dist q p*
using *LAST* **by** *argo*
qed
hence *L*: ∀ *q* ∈ *set-kdt l* − *set ?nns*. *dist* (*last ?nns*) *p* ≤ *dist q p*
using *IHLR* **by** *blast*

show *?thesis*
using *B R L* **by** *auto*
next
case *C*
hence ∀ *q* ∈ *set-kdt l*. *dist* (*last ?nnsr*) *p* ≤ *dist q p*
using *Node.prems*(*1*,*2*) *cutoff-l* **by** (*metis dist-commute less-imp-le*)
thus *?thesis*
using *IHR C* **by** *auto*
next
case *D*

let *?nns* = *nearest-nbors n ?nnsr p l*

have *set ?nnsr* ⊆ *set-kdt r* ∪ *set ps set ps* ∩ *set-kdt l* = {}
using *set-nns Node.prems*(*1*,*3*) **by** (*simp add*: *set-nns disjoint-iff-not-equal*)+
hence *set ?nnsr* ∩ *set-kdt l* = {}
using *Node.prems*(*1*) **by** *fastforce*
hence *IHRL*: ∀ *q* ∈ *set-kdt l* ∪ *set ?nnsr* − *set ?nns*. *dist* (*last ?nns*) *p* ≤ *dist q p*
using *Node.IH*(*1*)[*OF - SORTED-R - DISTINCT-R Node.prems*(*5*)] *Node.prems*(*1*) *invar-l* **by** *blast*

have $\forall q \in$ *set ps* $-$ *set ?nnsr*. *dist* (*last ?nns*) *p* $\leq$ *dist q p*
proof *standard*
fix *q*
assume $*$: $q \in$ *set ps* $-$ *set ?nnsr*

hence *length ?nnsr* = *n*
using *length-nns-n* **by** *blast*
hence *LAST*: *dist* (*last ?nns*) *p* $\leq$ *dist* (*last ?nnsr*) *p*
using *last-nns-mono SORTED-R invar-l Node.prems*(*1*,*2*,*5*) **by** (*metis order-refl*)
have *dist* (*last ?nnsr*) *p* $\leq$ *dist q p*
using *IHR* $*$ **by** *blast*
thus *dist* (*last ?nns*) *p* $\leq$ *dist q p*
using *LAST* **by** *argo*
qed
hence *R*: $\forall q \in$ *set-kdt l* $\cup$ *set ps* $-$ *set ?nns*. *dist* (*last ?nns*) *p* $\leq$ *dist q p*
using *IHRL* **by** *auto*

have $\forall q \in$ *set-kdt r* $-$ *set ?nnsr*. *dist* (*last ?nns*) *p* $\leq$ *dist q p*
proof *standard*
fix *q*
assume $*$: $q \in$ *set-kdt r* $-$ *set ?nnsr*

hence *length ?nnsr* = *n*
using *length-nns-n* **by** *blast*
hence *LAST*: *dist* (*last ?nns*) *p* $\leq$ *dist* (*last ?nnsr*) *p*
using *last-nns-mono SORTED-R invar-l Node.prems*(*1*,*2*,*5*) **by** (*metis order-refl*)
have *dist* (*last ?nnsr*) *p* $\leq$ *dist q p*
using *IHR* $*$ **by** *blast*
thus *dist* (*last ?nns*) *p* $\leq$ *dist q p*
using *LAST* **by** *argo*
qed
hence *L*: $\forall q \in$ *set-kdt r* $-$ *set ?nns*. *dist* (*last ?nns*) *p* $\leq$ *dist q p*
using *IHRL* **by** *blast*

show *?thesis*
using *D R L* **by** *auto*
qed
qed (*auto simp*: *sorted-wrt-dist-nbors-diff upd-nbors-def*)

4.6 Nearest Neighbors Definition and Theorems

definition *nearest-neighbors* :: *nat* $\Rightarrow$ (*'k*::*finite*) *point* $\Rightarrow$ *'k kdt* $\Rightarrow$ *'k point list*
where
nearest-neighbors n p kdt = *nearest-nbors n* [] *p kdt*

theorem *length-nearest-neighbors*:

length (*nearest-neighbors n p kdt*) = *min n* (*size-kdt kdt*)
by (*simp add*: *length-nns nearest-neighbors-def*)

theorem *sorted-wrt-dist-nearest-neighbors*:
sorted-wrt-dist p (*nearest-neighbors n p kdt*)
using *sorted-nns* **unfolding** *nearest-neighbors-def sorted-wrt-dist-def* **by** *force*

theorem *set-nearest-neighbors*:
set (*nearest-neighbors n p kdt*) $\subseteq$ *set-kdt kdt*
unfolding *nearest-neighbors-def* **using** *set-nns* **by** *force*

theorem *distinct-nearest-neighbors*:
assumes *invar kdt*
shows *distinct* (*nearest-neighbors n p kdt*)
using *assms* **by** (*simp add*: *distinct-nns nearest-neighbors-def*)

theorem *dist-nearest-neighbors*:
assumes *invar kdt nns* = *nearest-neighbors n p kdt*
shows $\forall$ *q* $\in$ (*set-kdt kdt* $-$ *set nns*). $\forall$ *r* $\in$ *set nns*. *dist r p* $\leq$ *dist q p*
proof (*cases 0* < *n*)
case *True*
have $\forall$ *q* $\in$ *set-kdt kdt* $-$ *set nns*. *dist* (*last nns*) *p* $\leq$ *dist q p*
using *nearest-neighbors-def dist-nns*[*OF assms*(*1*), *of p* [], *OF - - - True*] *assms*(*2*)
by (*simp add*: *nearest-neighbors-def sorted-wrt-dist-def*)
hence $\forall$ *q* $\in$ *set-kdt kdt* $-$ *set nns*. $\forall$ *n* $\in$ *set nns*. *dist n p* $\leq$ *dist q p*
using *assms*(*2*) *sorted-wrt-dist-nearest-neighbors*[*of p n kdt*] *sorted-wrt-dist-last*[*of p nns*] **by** *force*
thus *?thesis*
using *nearest-neighbors-def* **by** *blast*
next
case *False*
hence *length nns* = *0*
using *assms*(*2*) **unfolding** *nearest-neighbors-def* **by** (*auto simp*: *length-nns*)
thus *?thesis*
by *simp*
qed

end

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[2] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.