(Nearest) Neighbors You Can Rely On

Formally Verified k-d Tree Construction and Search in Coq

Nadeem Abdul Hamid
Berry College, Georgia, USA

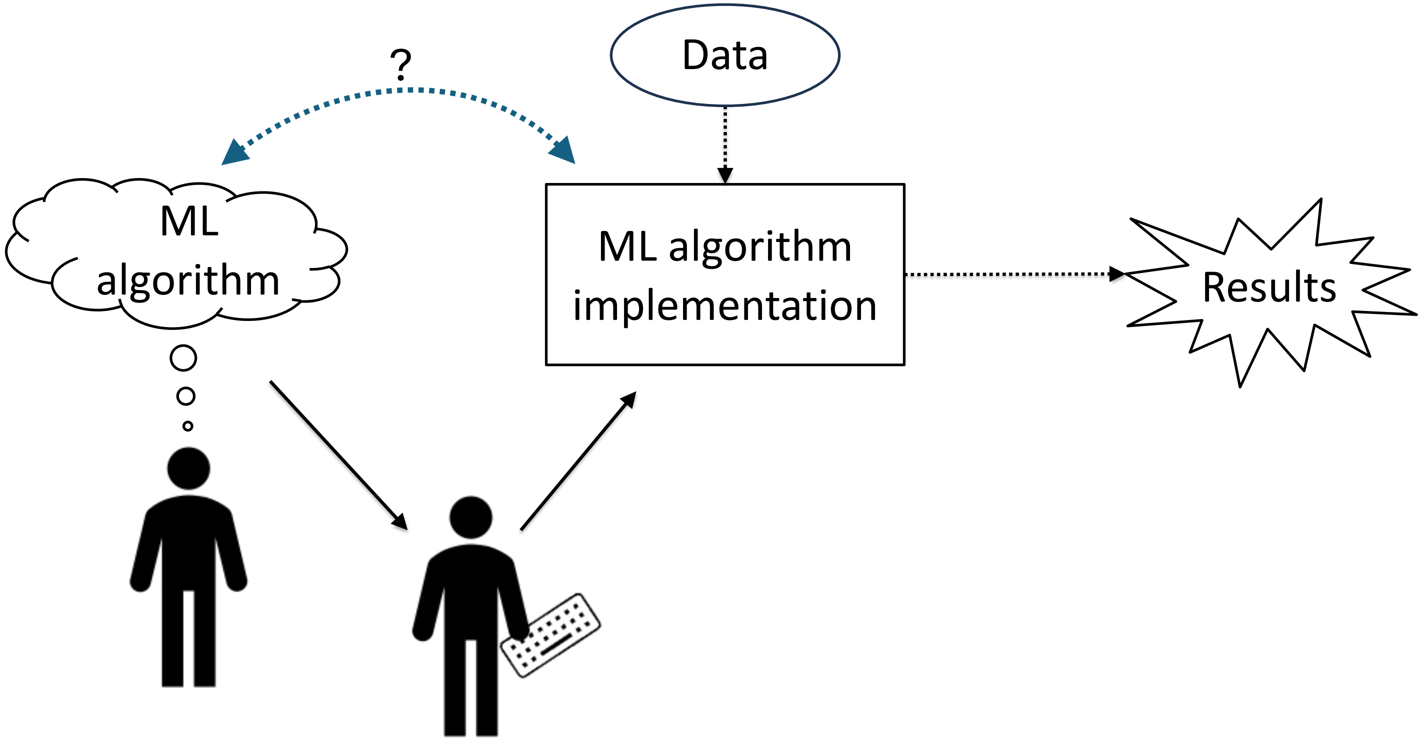
TRUST LABORATORY
Transformative Reliability Underlying Software Technology

BERRY COLLEGE

SAC-SVT 2024, Ávila, Spain

Implementing Machine Learning Algorithms

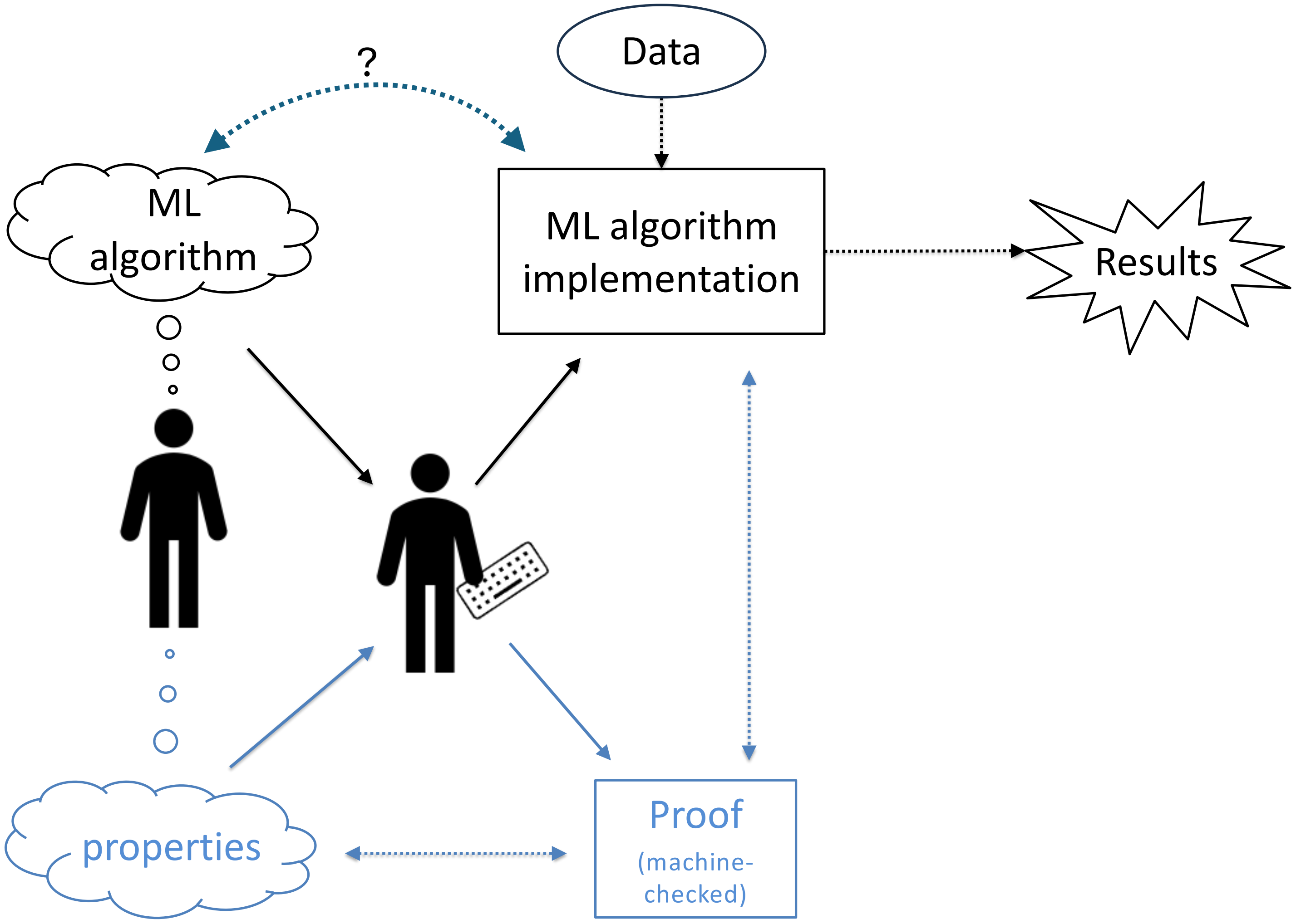
- Gap between the mathematical model and mechanics of implementation



Implementing Machine Learning Algorithms

- Gap between the mathematical model and mechanics of implementation

- (Big Picture) Context for this work: *Development of verified implementations of ML systems*



Focus: KNN (*K-nearest-neighbors*) Search

- Does the program code for an ML algorithm faithfully implement the mathematical description?
- Focus on the mechanics of the algorithm, not meta-theoretical properties
  - That an implementation correctly finds the closest neighbors to a query
  - Not that those closest neighbors have some statistical properties (Future work)

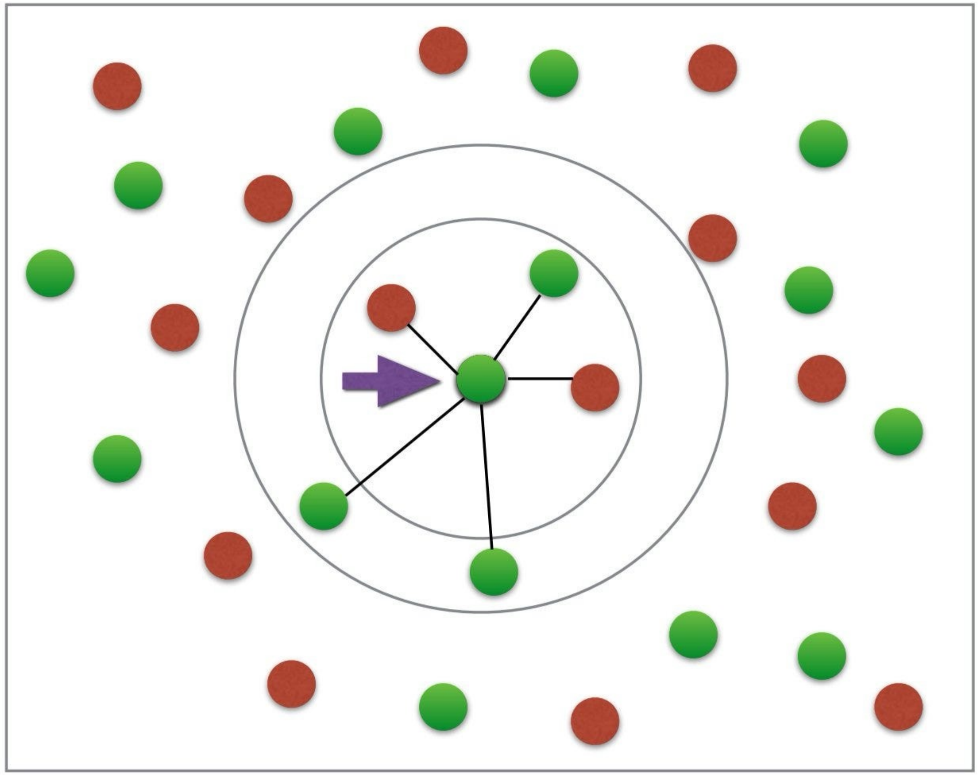


Image: datacamp.com

KNN Search

- One of the oldest, well-known, widely used <u>classification</u> algorithms
  - Assigns class labels to observations based on previously seen data
  - Can also be used for regression
- Applied in a wide variety of domains (not just ML)
- Popularity can be attributed to its simplicity, ease of implementation, and high accuracy rates
- Although, there are known limitations of KNN search
  - (curse of dimensionality; scaling to large data sets)

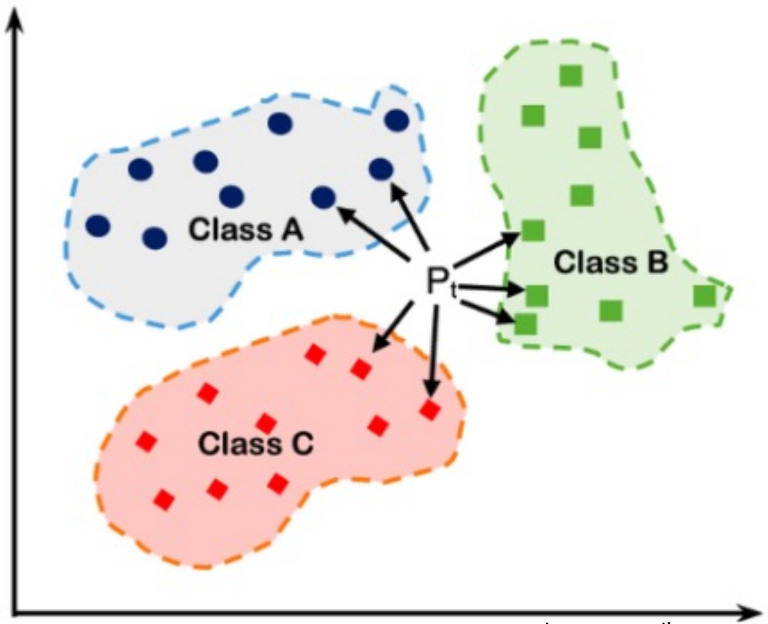


Image: medium.com

Our Results

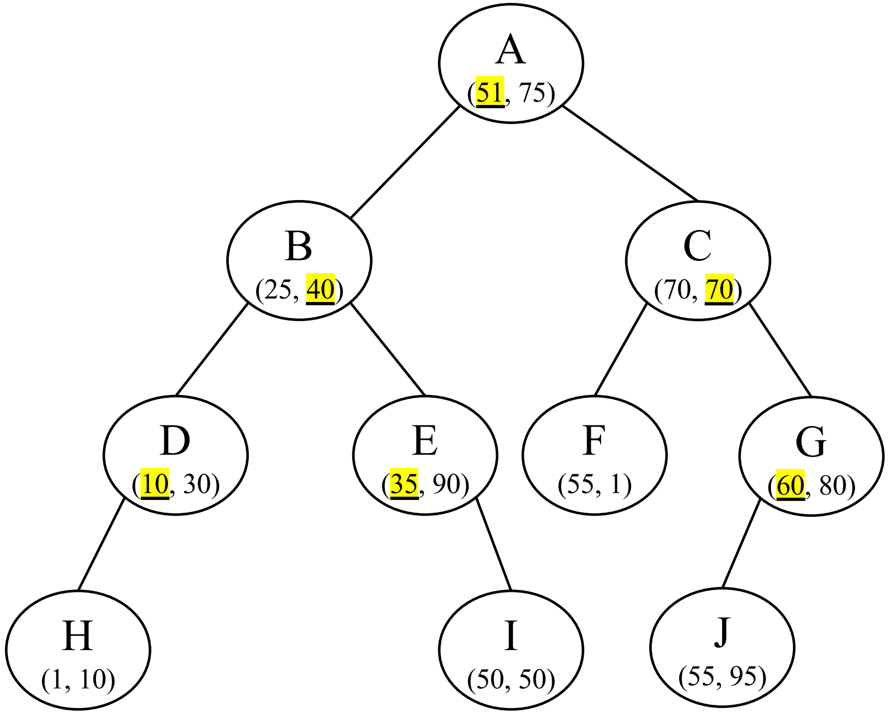
Formally verified (machine-checkable) implementation of a **KNN search algorithm** in the **Coq proof assistant**

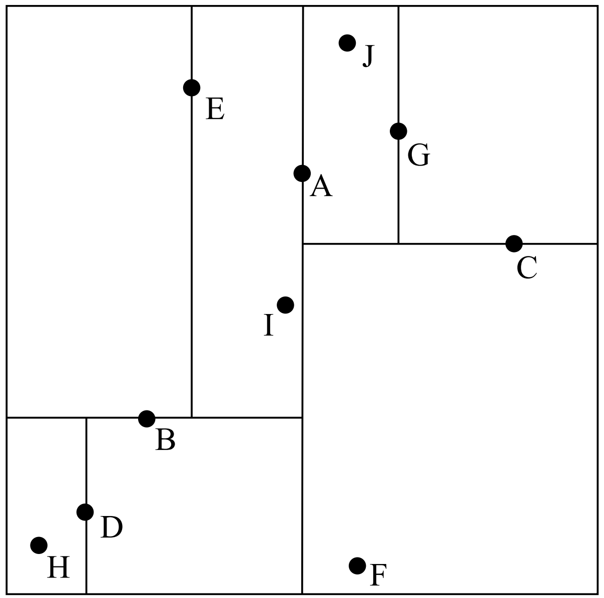
- Implementing/Integrating previously-verified data structures
  - ***k*-d trees** (new)
  - bounded priority queue (adapted)

- And algorithms
  - **Quick-select** median finding
  - Generalized ***K*-nearest neighbors search**

k-d trees

- Binary tree
- Nodes: *k*-dimensional data points
- Each level partitioned based on one of *k* dimensions
- Each subtree associated with an (implicit) bounding box
- Enables sub-linear NN search complexity through branch-and-bound






```
I: (50, 50)
A: (51, 75)
B: (25, 40)
C: (70, 70)
J: (55, 95)
H: (1, 10)
G: (60, 80)
F: (55, 1)
E: (35, 90)
D: (10, 30)
```

Lowercase ***k*** = dimension of data points;
Uppercase ***K*** = number of neighbors

Bounding boxes

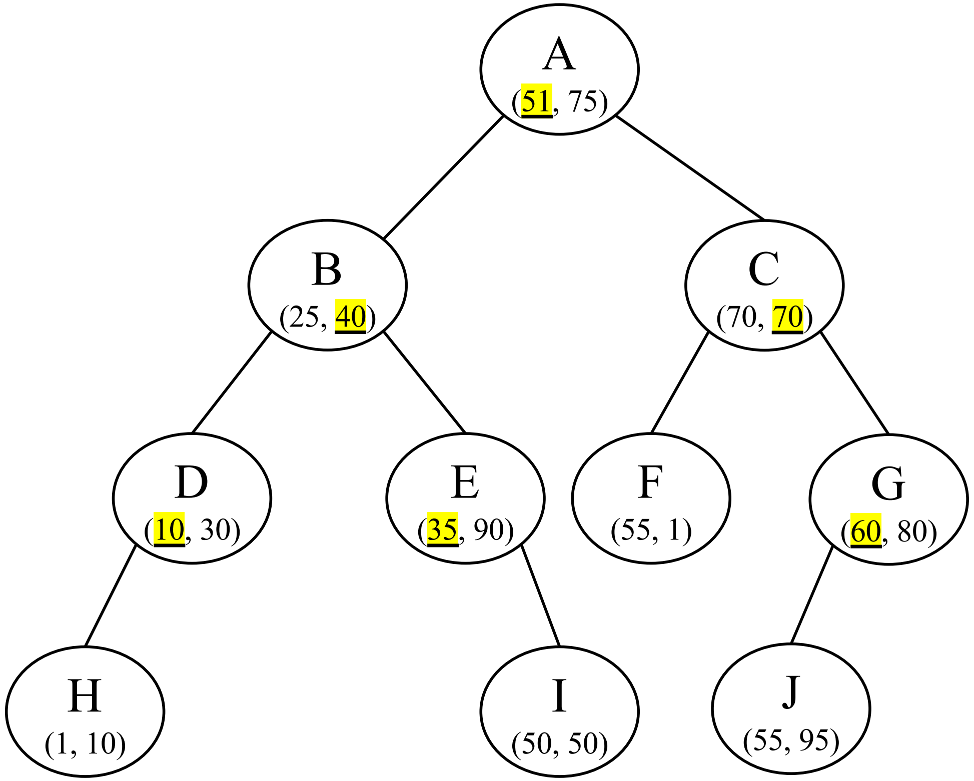
- Implicit in implementation; Crucial to correctness

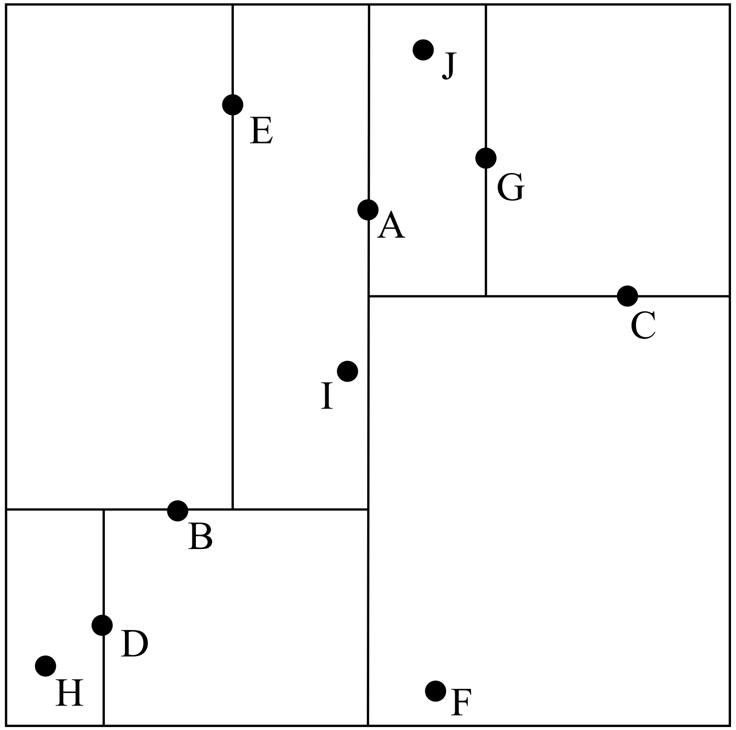
$[(x_{min}, y_{min}), (x_{max}, y_{max})]$

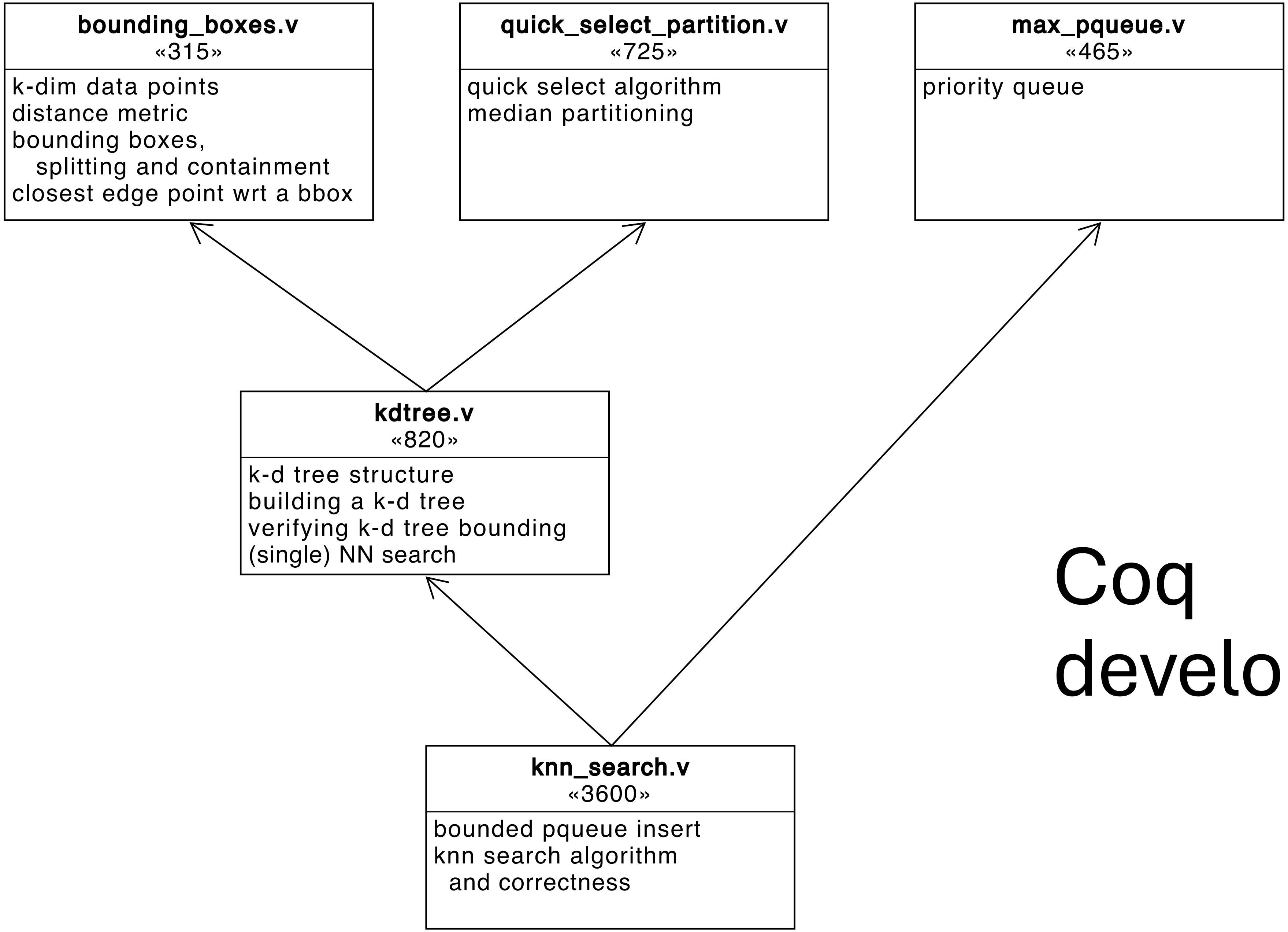
A - $[(-\infty, -\infty), (+\infty, +\infty)]$

…

I - $[(35, 40), (51, +\infty)]$





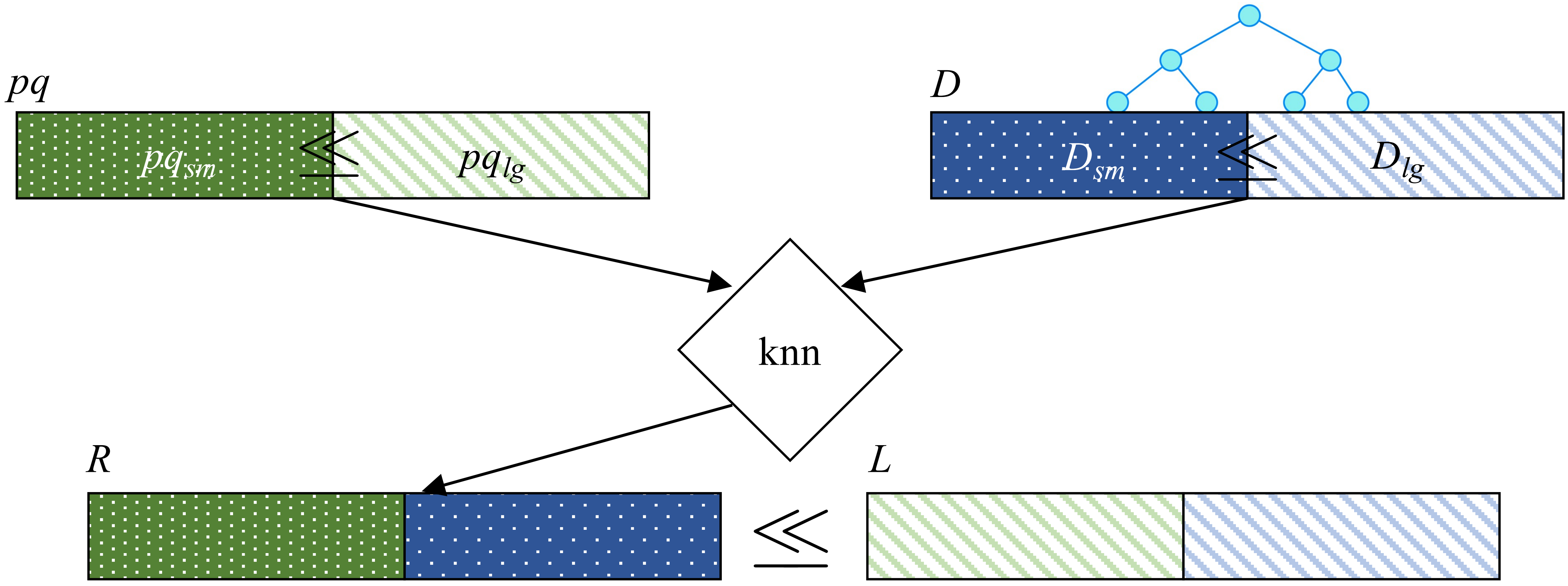


Coq developments

Final Theorem

```
Theorem knn_search_build_kdtree_correct :
 forall (K:nat) (k : nat) (data : list datapt),    // Preconditions:
   0 < K ->                                       // at least one neighbor sought
   0 < length data ->                             // data is non-empty
   0 < k ->                                       // dimension space is non-empty
   (forall v' : datapt,                           // all data points well-formed (k-dim)
          In v' data -> length v' = k) ->
   forall tree query result,
      tree = (build_kdtree k data) ->              // If: the k-d tree built from data
      knn_search K k tree query = result ->        // produces result for a query point,
      exists leftover,                             // Then:
         length result = min K (length data)       // the result is length (at most) K,
         /\ Permutation data (result ++ leftover)  // and is a sub-list of data,
         /\ all_in_leb (sum_dist query) result leftover. // and everything in
                 // result is closer in distance to the query than all the leftover part of data.
```

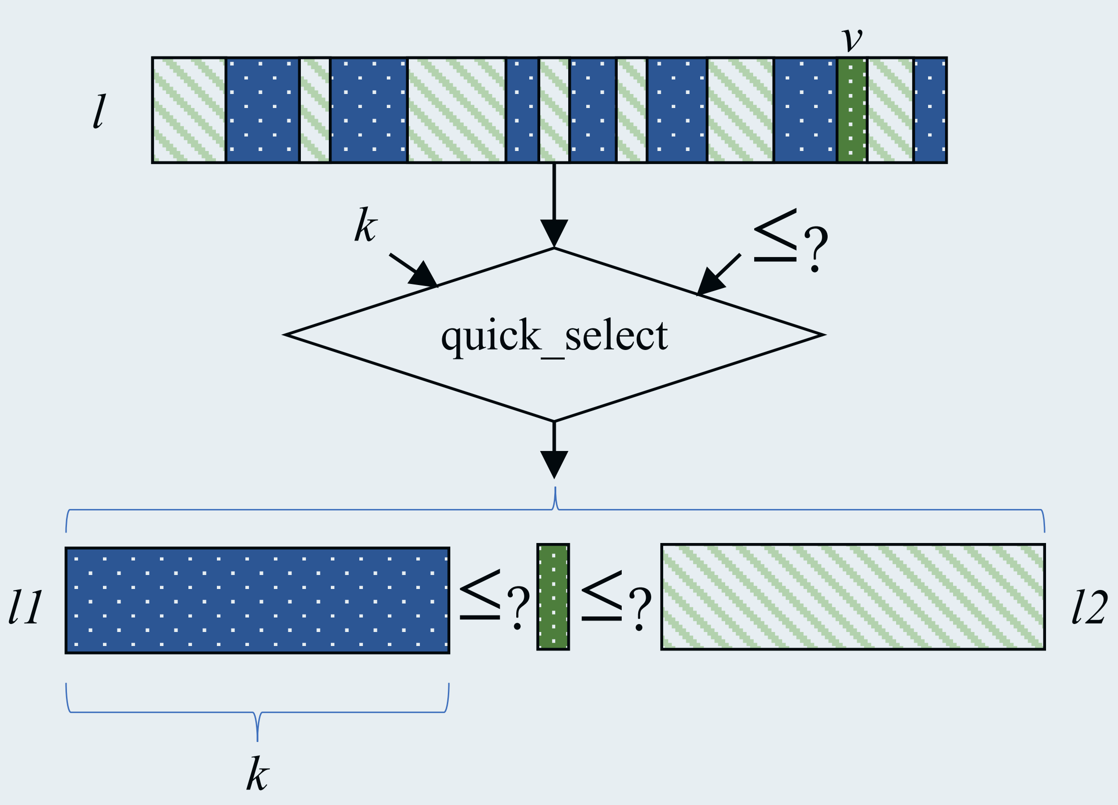
Partitions Induced by the *knn* Function



Quickselect

- Used to build the initial *k*-d tree

```
Theorem quick_select_exists_correct :
 forall (X:Set) (k:nat) (l:list X)
        (le:X -> X -> bool),
   le_props le ->
   k < length l ->
   exists l1 v l2,
     quick_select k l le = Some (l1,v,l2) /\
     Permutation l (l1 ++ v :: l2) /\
     length l1 = k /\
     (forall x, In x l1 -> le x v = true) /\
     (forall x, In x l2 -> le v x = true).
```



Future Work

- ~~Abstract the distance metric~~
- *Automate permutations reasoning*
- *Implement, specify, & verify a KNN-based classification algorithm*

- Port to verified C implementation
- Extend to modern variants of KNN (e.g. approximation, etc.)

Acknowledgements

Matthew Bowker

Jessica Herring

Bernny Velasquez

Thank you!

Questions?

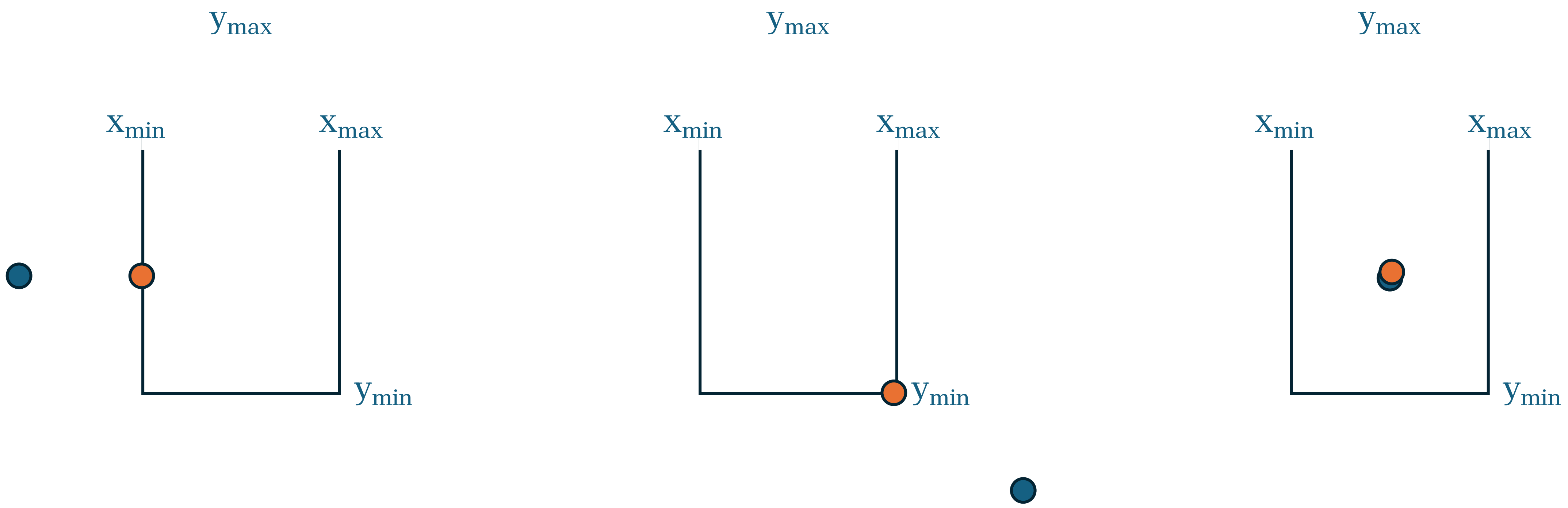
nadeem@acm.org

```
Fixpoint knn (K:nat) (k:nat) (tree:kdtree) (bb:bbox) (query:datapt)
        (pq:priqueue datapt (sum_dist query)) : priqueue datapt (sum_dist query)
:= match tree with
   | mt_tree => pq
   | node ax pt lft rgt =>
     let body (pq':priqueue datapt (sum_dist query)) :=
        let dx := nth ax pt 0 in
        let bbs := bb_split bb ax dx in
          if (ith_leb ax pt query)
          then (knn K k rgt (snd bbs) query (knn K k lft (fst bbs) query pq’))
          else (knn K k lft (fst bbs) query (knn K k rgt (snd bbs) query pq'))
     in
     match (peek_max _ _ pq) with
        | None => body (insert_bounded K _ _ pt pq)
        | Some top => if (K <=? (size _ _ pq))
                         && ((sum_dist query top) <?
                               (sum_dist query (closest_edge_point query bb)))
                      then pq
                      else body (insert_bounded K _ _ pt pq)
        end
   end.
```

```
Definition knn_search (K:nat) (k:nat) (tree:kdtree) (query:datapt) : list datapt
  :=
  pq_to_list
      (knn K k tree (mk_bbox (repeat None k) (repeat None k))
             query
             (empty datapt (sum_dist query))).
```

Closest Enclosed Point (cep)

LEMMA 4.6 (CEP_MIN_DIST).
Given a point q and bounding box B, $\forall p \in B$, $\delta_q(\mathrm{cep}(q, B)) \leq \delta_q(p)$.



```
Definition insert_bounded (K:nat) A key (e:A) (pq:priqueue A key)
                                                         : priqueue A key :=
  let updpq := (insert A key e pq)
  in
    if K <? (size A key updpq) then
      match delete_max _ key updpq with
      | None => updpq (* should never happen *)
      | Some (_ , updpq') => updpq’
      end
     else updpq.
```

```
Lemma insert_bounded_preserve_max
   : forall (K : nat) (A : Type) (key : A -> nat) (e : A)
            (pq : priqueue A key) (lst : list A),
     priq A key pq ->
     Abs A key pq lst ->
     size A key pq = K -> size A key (insert_bounded K A key e pq) = K.
```

```
Proof.
    unfold insert_bounded; intros.
    rewrite insert_size with (al:=lst); auto.
    rewrite HK.
    replace (K <? 1 + K) with true.
    2: { destruct (K <? 1 + K) eqn:Hk; auto; split_andb_leb; lia. }
    pose proof (insert_delete_max_some _ _ e _ _ Hpriq Habs) as (k, (q, Hd)).
    rewrite Hd.
    apply delete_max_Some_size with (p:=(insert A key e pq)) (k:=k) (pl:=e::lst)
    ; auto.
    rewrite <- HK.
    eapply insert_size; eauto.
Qed.
```

```
insert_bounded_preserve_max =
(fun (K : nat) (A : Type) (key : A -
> nat) (e : A) (pq : priqueue A key) (lst : list A) (Hpriq : priq A key pq) (Habs : Abs A key pq lst) (HK : size A key pq = K) => eq_ind_r (fun n : nat => size A key (i
f K <? n then match delete_max A key (insert A key e pq) with
| Some (_, updpq') => updpq'
| None => insert A key e pq
end else insert A key e pq) = K) (eq_ind_r (fun n : nat => size A key (if K <? 1 + n then match delete_max A key (insert A key e pq) with
| Some (_, updpq') => updpq'
| None => insert A key e pq
end else insert A key e pq) = K) (let H : true = (K <? 1 + K) := let b := K <? 1 + K in let Hk : (K <? 1 + K) = b := eq_refl in (if b as b0 return ((K <? 1 + K) = b0 -
> true = b0) then fun _ : (K <? 1 + K) = true => eq_refl else fun Hk0 : (K <? 1 + K) = false => let H : forall x y : nat, (x <? y) = false -
> y <= x := fun x y : nat => match Nat.ltb_ge x y with
| conj x0 _ => x0
end in let Hk1 : 1 + K <= K := H K (1 + K) Hk0 in let Hk2 : BinInt.Z.le (BinInt.Z.add (BinNums.Zpos BinNums.xH) (BinInt.Z.of_nat K)) (BinInt.Z.of_nat K) := ZifyClasses.
rew_iff (1 + K <= K) (BinInt.Z.le (BinInt.Z.add (BinNums.Zpos BinNums.xH) (BinInt.Z.of_nat K)) (BinInt.Z.of_nat K)) (ZifyClasses.mkrel nat BinNums.Z le BinInt.Z.of_nat
BinInt.Z.le Znat.Nat2Z.inj_le (1 + K) (BinInt.Z.add (BinNums.Zpos BinNums.xH) (BinInt.Z.of_nat K)) (ZifyClasses.mkapp2 nat nat nat BinNums.Z BinNums.Z BinNums.Z Nat.add
 BinInt.Z.of_nat BinInt.Z.of_nat BinInt.Z.of_nat BinInt.Z.add Znat.Nat2Z.inj_add 1 (BinNums.Zpos BinNums.xH) eq_refl K (BinInt.Z.of_nat K) eq_refl) K (BinInt.Z.of_nat K
) eq_refl) Hk1 in let HK0 : BinInt.Z.of_nat (size A key pq) = BinInt.Z.of_nat K := ZifyClasses.rew_iff (size A key pq = K) (BinInt.Z.of_nat (size A key pq) = BinInt.Z.o
f_nat K) (ZifyClasses.mkrel nat BinNums.Z eq BinInt.Z.of_nat eq (fun x y : nat => iff_sym (Znat.Nat2Z.inj_iff x y)) (size A key pq) (BinInt.Z.of_nat (size A key pq)) eq
_refl K (BinInt.Z.of_nat K) eq_refl) HK in let cstr : BinInt.Z.le BinNums.Z0 (BinInt.Z.of_nat (size A key pq)) := Znat.Nat2Z.is_nonneg (size A key pq) in let cstr0 : Bi
nInt.Z.le BinNums.Z0 (BinInt.Z.of_nat K) := Znat.Nat2Z.is_nonneg K in let __arith : forall (__p1 : Prop) (__x1 : BinNums.Z), BinInt.Z.le (BinInt.Z.add (BinNums.Zpos Bin
Nums.xH) __x1) __x1 -
> __p1 := fun (__p1 : Prop) (__x1 : BinNums.Z) => let __wit := [] in let __varmap := VarMap.Elt __x1 in let __ff := Tauto.IMPL (Tauto.A Tauto.isProp {| RingMicromega.Fl
hs := EnvRing.PEadd (EnvRing.PEc (BinNums.Zpos BinNums.xH)) (EnvRing.PEX BinNums.xH); RingMicromega.Fop := RingMicromega.OpLe; RingMicromega.Frhs := EnvRing.PEX BinNums
.xH |} tt) None (Tauto.X Tauto.isProp __p1) in ZMicromega.ZTautoChecker_sound __ff __wit (eq_refl <: ZMicromega.ZTautoChecker __ff __wit = true) (VarMap.find BinNums.Z0
 __varmap) in __arith (true = false) (BinInt.Z.of_nat K) Hk2) Hk in eq_ind true (fun b : bool => size A key (if b then match delete_max A key (insert A key e pq) with
| Some (_, updpq') => updpq'
| None => insert A key e pq
end else insert A key e pq) = K) (let H0 : exists (k : A) (q : priqueue A key), delete_max A key (insert A key e pq) = Some (k, q) := insert_delete_max_some A key e pq
lst Hpriq Habs in match H0 with
| ex_intro _ x x0 => (fun (k : A) (H1 : exists q : priqueue A key, delete_max A key (insert A key e pq) = Some (k, q)) => match H1 with
| ex_intro _ x1 x2 => (fun (q : priqueue A key) (Hd : delete_max A key (insert A key e pq) = Some (k, q)) => eq_ind_r (fun o : option (A * priqueue A key) => size A key
 match o with
| Some (_, updpq') => updpq'
| None => insert A key e pq
end = K) (delete_max_Some_size A key K (insert A key e pq) q k (e :: lst) lst (insert_priq A key e pq Hpriq) (insert_relate A key pq lst e Hpriq Habs) (eq_ind (size A k
ey pq) (fun K0 : nat => size A key (insert A key e pq) = S K0) (insert_size A key pq lst e Hpriq Habs) K HK) Hd) Hd) x1 x2
end) x x0
end) (K <? 1 + K) H) HK) (insert_size A key pq lst e Hpriq Habs)) : forall (K : nat) (A : Type) (key : A -
> nat) (e : A) (pq : priqueue A key) (lst : list A), priq A key pq -> Abs A key pq lst -> size A key pq = K -> size A key (insert_bounded K A key e pq) = K
: forall (K : nat) (A : Type) (key : A -> nat) (e : A) (pq : priqueue A key) (lst : list A), priq A key pq -> Abs A key pq lst -> size A key pq = K -
> size A key (insert_bounded K A key e pq) = K
```