import Mathlib

universe u

namespace project

structure HeapNodeAux (α : Type u) (h : Type u) where

  val : α

  rank : Nat

  children : h

-- A `Heap` is a forest of binomial trees.

inductive Heap (α : Type u) : Type u where

  | heap (ns : List (HeapNodeAux α (Heap α))) : Heap α

  deriving Inhabited

open Heap

-- A `BinTree` is a binomial tree. If a `BinTree` has rank `k`, its children

-- have ranks between `0` and `k - 1`. They are ordered by rank. Additionally,

-- the value of each child must be greater than or equal to the value of its

-- parent node.

abbrev BinTree α := HeapNodeAux α (Heap α)

def Heap.nodes : Heap α → List (BinTree α)

  | heap ns => ns

@[simp]

theorem Heap.nodes\_def : nodes (heap xs) = xs := rfl

variable {α : Type u}

def hRank : List (BinTree α) → Nat

  | []   => 0

  | h::\_ => h.rank

def isEmpty : Heap α → Bool

  | heap [] => true

  | \_       => false

def empty : Heap α :=

  heap []

def singleton (a : α) : Heap α :=

  heap [{ val := a, rank := 1, children := heap [] }]

-- Combine two binomial trees of rank `r`, creating a binomial tree of rank

-- `r + 1`.

@[specialize] def combine (le : α → α → Bool) (n₁ n₂ : BinTree α) : BinTree α :=

  if le n₂.val n₁.val then

     { n₂ with rank := n₂.rank + 1, children := heap $ n₂.children.nodes ++ [n₁] }

  else

     { n₁ with rank := n₁.rank + 1, children := heap $ n₁.children.nodes ++ [n₂] }

-- Merge two forests of binomial trees. The forests are assumed to be ordered

-- by rank and `mergeNodes` maintains this invariant.

@[specialize] def mergeNodes (le : α → α → Bool) : List (BinTree α) → List (BinTree α) → List (BinTree α)

  | [], h  => h

  | h,  [] => h

  | f@(h₁ :: t₁), s@(h₂ :: t₂) =>

    if h₁.rank < h₂.rank then h₁ :: mergeNodes le t₁ s

    else if h₂.rank < h₁.rank then h₂ :: mergeNodes le t₂ f

    else

      let merged := combine le h₁ h₂

      let r      := merged.rank

      if r != hRank t₁ then

        if r != hRank t₂ then merged :: mergeNodes le t₁ t₂ else mergeNodes le (merged :: t₁) t₂

      else

        if r != hRank t₂ then mergeNodes le t₁ (merged :: t₂) else merged :: mergeNodes le t₁ t₂

termination\_by \_ h₁ h₂ => h₁.length + h₂.length

decreasing\_by simp\_wf; simp\_arith [\*]

@[specialize] def merge (le : α → α → Bool) : Heap α → Heap α → Heap α

  | heap h₁, heap h₂ => heap (mergeNodes le h₁ h₂)

@[specialize] def head? (le : α → α → Bool) : Heap α → Option α

  | heap []      => none

  | heap (h::hs) => some $

    hs.foldl (init := h.val) fun r n => if le r n.val then r else n.val

@[inline] def head [Inhabited α] (le : α → α → Bool) (h : Heap α) : α :=

  head? le h |>.getD default

@[specialize] def findMin (le : α → α → Bool) : List (BinTree α) → Nat → BinTree α × Nat → BinTree α × Nat

  | [],    \_,   r          => r

  | h::hs, idx, (h', idx') => if le h'.val h.val then findMin le hs (idx+1) (h', idx') else findMin le hs (idx+1) (h, idx)

    -- It is important that we check `le h'.val h.val` here, not the other way

    -- around. This ensures that head? and findMin find the same element even

    -- when we have `le h'.val h.val` and `le h.val h'.val` (i.e. le is not

    -- irreflexive).

def deleteMin (le : α → α → Bool) : Heap α → Option (α × Heap α)

  | heap [] => none

  | heap [h] => some (h.val, h.children)

  | heap hhs@(h::hs) =>

    let (min, minIdx) := findMin le hs 1 (h, 0)

    let rest          := hhs.eraseIdx minIdx

    let tail          := merge le (heap rest) min.children

    some (min.val, tail)

@[inline] def tail? (le : α → α → Bool) (h : Heap α) : Option (Heap α) :=

  deleteMin le h |>.map (·.snd)

@[inline] def tail (le : α → α → Bool) (h : Heap α) : Heap α :=

  tail? le h |>.getD empty

partial def toList (le : α → α → Bool) (h : Heap α) : List α :=

  match deleteMin le h with

  | none          => []

  | some (hd, tl) => hd :: toList le tl

partial def toArray (le : α → α → Bool) (h : Heap α) : Array α :=

  go #[] h

  where

    go (acc : Array α) (h : Heap α) : Array α :=

      match deleteMin le h with

      | none => acc

      | some (hd, tl) => go (acc.push hd) tl

partial def toListUnordered : Heap α → List α

  | heap ns => ns.bind fun n => n.val :: toListUnordered n.children

partial def toArrayUnordered (h : Heap α) : Array α :=

  go #[] h

  where

    go (acc : Array α) : Heap α → Array α

      | heap ns => Id.run do

        let mut acc := acc

        for n in ns do

          acc := acc.push n.val

          acc := go acc n.children

        return acc

inductive All (P: α → Prop) : List α → Prop

| nil: All P []

| cons: P x → All P xs → All P (x :: xs)

mutual

  inductive IsBinTree : BinTree α → Prop where

  | C: IsRankedTree 1 a.rank a.children.nodes → IsBinTree a

  /-IsRankedTree n m ts :<=> the list ts contains the children of the parentnode of a binomial tree, IsRankedTree

  assures that the order of the rank of the children is n, n+1, n+2,.....,m-1 and if n = m, then ts is empty-/

  inductive IsRankedTree : Nat → Nat → List (BinTree α)  → Prop where

  | nil : IsRankedTree n n []

  | cons : t.rank = n  → IsRankedTree (n + 1) m ts → IsBinTree t → IsRankedTree n m (t::ts)

end

/-IsHeap rank [t₁,...,tₙ] :<=> Each tree in the list t₁ upto tₙ should have a smaller rank than the tree

that follows so tₙ.rank < tₙ₊₁. Also each tree in the list should be a binomial tree so IsBinTree t should hold for each tree t-/

inductive IsHeapForest' : Nat → List (BinTree α) → Prop where

| nil : IsHeapForest' rank []

| cons : rank < t.rank → IsBinTree t → IsHeapForest' t.rank ts → IsHeapForest' rank (t::ts)

abbrev IsHeapForest : List (BinTree α) → Prop := IsHeapForest' 0

/-IsHeap (heap [ts]) holds if IsHeapForest 0 ts holds, 0 is used because every binomial tree has a rank higher than 0-/

inductive IsHeap : Heap α → Prop where

| intro : IsHeapForest ts → IsHeap (heap ts)

mutual

  inductive IsSearchTree (le : α → α → Bool) : BinTree α → Prop where

  | C : IsMinTree le a.val a.children.nodes → IsSearchTree le a

  /-IsMinHeap le ns val :<=> assures that val(value of parent node) is less or equal than the value

  of the nodes in ns(the children). Maintains minimum heap property-/

  inductive IsMinTree (le : α → α → Bool) : α → List (BinTree α) → Prop where

  | nil : IsMinTree le val []

  | cons : le val n.val → IsMinTree le val ns → IsSearchTree le n → IsMinTree le val (n::ns)

end

/-IsSearchForrest le (heap [t₁,...,tₙ]) :<=> IsMinHeap holds if for each tree t in the list t₁ upto tₙ, IsSearchTree le t holds-/

inductive IsMinHeap (le : α → α → Bool) : Heap α → Prop where

| nil : IsMinHeap le (heap [])

| cons : IsSearchTree le n → IsMinHeap le (heap ns) → IsMinHeap le (heap (n::ns))

theorem heap\_empty : IsHeap (@empty α) := by

  repeat constructor

theorem min\_heap\_empty : IsMinHeap le (@empty α) := by

  constructor

theorem singleton\_ranked : IsHeap (singleton a) := by

  constructor

  constructor

  . dsimp

    apply Nat.lt\_succ\_self

  . constructor

    dsimp

    constructor

  . constructor

theorem singleton\_min\_heap : IsMinHeap le (singleton a) := by

  constructor

  . constructor

    dsimp

    constructor

  . constructor

theorem IsRankedTree\_append (h : IsRankedTree n m xs) (ha: IsBinTree a) (hrank: a.rank = m) :

  IsRankedTree n (m + 1) (xs ++ [a]) := by

  induction xs generalizing n

  case nil =>

    dsimp

    cases h

    constructor

    assumption

    constructor

    assumption

  case cons b xs ih =>

    cases h

    constructor

    . assumption

    . apply ih

      assumption

    . assumption

theorem combine\_trees\_IsBinTree (le : α → α → Bool) (a b : BinTree α) :

  IsBinTree a → IsBinTree b → a.rank = b.rank → IsBinTree (combine le a b) := by

    intros ha hb hab

    constructor

    unfold combine

    split

    case inl =>

      dsimp

      cases hb

      apply IsRankedTree\_append

      repeat assumption

    case inr =>

      dsimp

      cases ha

      apply IsRankedTree\_append

      repeat assumption

      apply Eq.symm

      assumption

theorem IsMinTree\_append (h : IsMinTree le m xs) (ha : IsSearchTree le a) (hba: le m a.val = true) :

  IsMinTree le m (xs ++ [a]) := by

    induction xs with

    | nil =>

      dsimp

      constructor <;> assumption

    | cons \_ \_ ih =>

      cases h

      constructor

      . assumption

      . dsimp

        apply ih

        assumption

      . assumption

variable {le : α → α → Bool} (not\_le\_le : ∀ x y, ¬ le x y → le y x)

theorem combine\_trees\_IsSearchTree (a b : BinTree α) :

  IsSearchTree le a → IsSearchTree le b → IsSearchTree le (combine le a b) := by

    intros ha hb

    constructor

    unfold combine

    split

    case inl =>

      dsimp

      cases hb

      apply IsMinTree\_append

      repeat assumption

    case inr =>

      dsimp

      cases ha

      apply IsMinTree\_append

      repeat assumption

      apply not\_le\_le

      assumption

theorem temp : mergeNodes le [] h = h := by

  unfold mergeNodes

  split

  . rfl

  . rfl

  . rename\_i f

    cases f

theorem temp2 : mergeNodes le h [] = h := by

  unfold mergeNodes

  split

  . rfl

  . rfl

  . rename\_i f

    cases f

theorem IsHeapForest'\_weaken : IsHeapForest' n xs → m ≤ n → IsHeapForest' m xs := by

  intros h hle

  cases h with

  | nil => constructor

  | cons h1 h2 h3 =>

    constructor <;> try assumption

    . apply Nat.lt\_of\_le\_of\_lt hle h1

theorem rank\_combine : h₁.rank = h₂.rank → (combine le h₁ h₂).rank = h₁.rank + 1:= by

intro h₃

unfold combine

split

. dsimp

  simp

  apply Eq.symm

  assumption

. dsimp

theorem IsHeapForest'\_strengthen (h : IsHeapForest' r (t :: ts)) (hpos : t.rank > 0) : IsHeapForest' (t.rank - 1) (t :: ts) := by

  cases h

  constructor <;> try assumption

  . apply Nat.sub\_lt\_self

    . decide

    . assumption

theorem IsHeapForest'\_lowerbound : rx < ry → IsHeapForest' ry ts → IsHeapForest' rx ts := by

intros h₁ h₂

cases h₂

. constructor

. constructor

  . apply lt\_trans h₁

    assumption

  . assumption

  . assumption

theorem IsHeapForest'\_strengthen2 : IsHeapForest' rx ts → ry < hRank ts → IsHeapForest' ry ts := by

intros h₁ h₂

cases h₁

. constructor

. constructor

  repeat assumption

theorem hrank\_of\_cons : hRank (t :: ts) = t.rank := by

rfl

theorem hrank\_of\_empty (ts : List (BinTree α)) : ts = [] → hRank ts = 0 := by

intros h₁

rw[h₁]

rfl

theorem Nat.min\_eq (x y : Nat) : min x y = if x ≤ y then x else y := rfl

theorem Bool.not\_eq\_false' (x : Bool) : (!x) = false ↔ x = true := by

  cases x <;> simp

--lemma to proof min (hRank t1) (hRank t2) <= hRank (mergeNodes ...)

theorem min\_hRank\_mergeNodes (ht₁ : IsHeapForest' r t1) (ht₂ : IsHeapForest' r t2) : min (hRank t1) (hRank t2) ≤ hRank (mergeNodes le t1 t2) :=

match t1, t2 with

| [], h  => by

  rw[Nat.min\_eq]

  split

  . rw[temp]

    assumption

  . rw[temp]

    apply le\_rfl

| h,  [] => by

  rw[temp2]

  rw[Nat.min\_eq]

  split

  . apply le\_rfl

  . simp at \*

    apply Nat.le\_of\_lt

    assumption

| (h₁ :: t₁), (h₂ :: t₂) => by

  unfold mergeNodes

  split

  . repeat rw[hrank\_of\_cons]

    rw[Nat.min\_eq]

    split

    . apply le\_rfl

    . simp at \*

      apply Nat.le\_of\_lt

      assumption

  . split

    . repeat rw[hrank\_of\_cons]

      rw[Nat.min\_eq]

      split

      . assumption

      . apply le\_rfl

    . dsimp

      split

      . split

        . simp at \*

          have i : h₁.rank = h₂.rank

          apply LE.le.antisymm <;> assumption

          repeat rw[hrank\_of\_cons]

          rw[rank\_combine i]

          rw[Nat.min\_eq]

          split <;> apply Nat.le\_succ

        . repeat rw[hrank\_of\_cons]

          have h₃ : IsHeapForest' r (combine le h₁ h₂ :: t₁) := by

            simp at \*

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            constructor

            . rw[rank\_combine i] at \*

              cases ht₁

              apply Nat.lt\_succ\_of\_lt

              assumption

            . cases ht₁

              cases ht₂

              apply combine\_trees\_IsBinTree <;> assumption

            . cases ht₁

              rw[rank\_combine i] at \*

              rename\_i a

              cases a

              . constructor

              . rename\_i z x c v b n

                constructor

                . rw[hrank\_of\_cons] at \*

                  unfold bne at n

                  rw[Nat.not\_beq\_eq\_true\_eq] at n

                  have l : h₁.rank + 1 ≤  z.rank

                  apply Nat.succ\_le\_of\_lt b

                  apply lt\_of\_le\_of\_ne <;> assumption

                repeat assumption

          have h₄ : IsHeapForest' r t₂ := by

            cases ht₂

            apply IsHeapForest'\_weaken

            . assumption

            . apply LT.lt.le

              assumption

          have ih := min\_hRank\_mergeNodes h₃ h₄

          transitivity (min (hRank (combine le h₁ h₂ :: t₁)) (hRank t₂))

          . rw[hrank\_of\_cons] at \*

            simp at \*

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            rw[rank\_combine i] at \*

            repeat rw[Nat.min\_eq]

            split

            . split

              . apply Nat.le\_succ

              . simp at \*

                rename\_i e f g

                unfold bne at e

                rw [Bool.not\_eq\_false', beq\_iff\_eq] at e

                simp [← e] at g

            . split

              . apply Nat.le\_succ

              . rename\_i e f g

                unfold bne at e

                rw [Bool.not\_eq\_false', beq\_iff\_eq] at e

                simp [← e] at g

          . apply ih

      . split

        . repeat rw[hrank\_of\_cons]

          have h₃ : IsHeapForest' r t₁ := by

            cases ht₁

            apply IsHeapForest'\_weaken

            . assumption

            . apply LT.lt.le

              assumption

          have h₄ : IsHeapForest' r (combine le h₁ h₂ :: t₂) := by

            simp at \*

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            constructor

            . cases ht₁

              rw[rank\_combine i] at \*

              apply Nat.lt\_succ\_of\_lt

              assumption

            . cases ht₁

              cases ht₂

              apply combine\_trees\_IsBinTree <;> assumption

            . rw[rank\_combine i] at \*

              cases ht₂

              rename\_i a

              cases a

              . constructor

              . rename\_i h₃ h₄ h₅ h₆ h₇ h₈ h₉

                rw[hrank\_of\_cons] at \*

                constructor

                . rw[←i] at h₈

                  unfold bne at h₉

                  rw[Nat.not\_beq\_eq\_true\_eq] at h₉

                  apply lt\_of\_le\_of\_ne <;> assumption

                repeat assumption

          have ih := min\_hRank\_mergeNodes h₃ h₄

          transitivity (min (hRank t₁) (hRank (combine le h₁ h₂ :: t₂)))

          . rw[hrank\_of\_cons] at \*

            simp at \*

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            rw[rank\_combine i] at \*

            repeat rw[Nat.min\_eq]

            split

            . split

              . cases ht₁

                rename\_i a

                cases a

                . contradiction

                . rw[hrank\_of\_cons]

                  rename\_i b c d

                  apply Nat.le\_of\_lt

                  assumption

              . apply Nat.le\_succ

            . split

              . cases ht₁

                rename\_i a

                cases a

                . contradiction

                . rw[hrank\_of\_cons]

                  rename\_i b c d

                  apply Nat.le\_of\_lt

                  assumption

              . apply Nat.le\_succ

          . apply ih

        . repeat rw[hrank\_of\_cons]

          simp at \*

          have i : h₁.rank = h₂.rank

          apply LE.le.antisymm <;> assumption

          rw[rank\_combine i]

          rw[Nat.min\_eq]

          split <;> apply Nat.le\_succ

termination\_by \_  => t1.length + t2.length

decreasing\_by simp\_wf; simp\_arith [\*]

theorem hRank\_mergeNodes\_cons (ht₁ : IsHeapForest' r (u :: y)) (ht₂ : IsHeapForest' r (c :: z)) : u.rank = c.rank → u.rank + 1 ≤ hRank (mergeNodes le (u :: y) (c :: z)) := by

intros h₁

unfold mergeNodes

split

. have h₂ : u.rank ≠ c.rank := by

    apply ne\_of\_lt

    assumption

  contradiction

. split

  . have h₂ : c.rank ≠ u.rank := by

      apply ne\_of\_lt

      assumption

    have : c.rank = u.rank := by

      apply Eq.symm h₁

    contradiction

  . dsimp

    split

    . split

      . rw[hrank\_of\_cons]

        rw[rank\_combine h₁] at \*

        apply le\_rfl

      . have h₂ : IsHeapForest' r (combine le u c :: y) := by

          constructor

          . rw[rank\_combine h₁] at \*

            cases ht₁

            rename\_i h₂ h₃

            apply Nat.lt\_succ\_of\_lt h₂

          . cases ht₁

            cases ht₂

            apply combine\_trees\_IsBinTree <;> assumption

          . cases ht₁

            cases ht₂

            rename\_i h₂ h₃ h₄ h₅

            cases h₂

            . constructor

            . rename\_i a b g q t x

              rw[rank\_combine h₁, hrank\_of\_cons] at \*

              unfold bne at x

              rw[Nat.not\_beq\_eq\_true\_eq] at x

              have m : u.rank + 1 ≤ a.rank := by

                apply Nat.succ\_le\_of\_lt t

              constructor

              . apply lt\_of\_le\_of\_ne <;> assumption

              . assumption

              . assumption

        have h₃ : IsHeapForest' r z := by

          cases ht₂

          apply IsHeapForest'\_weaken

          . assumption

          . apply Nat.le\_of\_lt

            assumption

        have h₄ : min (hRank (combine le u c :: y)) (hRank z) ≤ hRank (mergeNodes le (combine le u c :: y) z) := by

          apply min\_hRank\_mergeNodes h₂ h₃

        repeat rw[hrank\_of\_cons] at \*

        rw[Nat.min\_eq] at h₄

        rw[rank\_combine h₁] at \*

        split at h₄

        . assumption

        . unfold bne at \*

          rw[Nat.not\_beq\_eq\_true\_eq] at \*

          simp at \*

          rename\_i h₅ h₆

          rw[h₅]

          assumption

    . split

      . have h₂ : IsHeapForest' r (combine le u c :: z) := by

          constructor

          . rw[rank\_combine h₁] at \*

            cases ht₂

            rename\_i h₂ h₃

            rw[← h₁] at h₂

            apply Nat.lt\_succ\_of\_lt h₂

          . cases ht₁

            cases ht₂

            apply combine\_trees\_IsBinTree <;> assumption

          . cases ht₁

            cases ht₂

            rename\_i a

            cases a

            . constructor

            . rename\_i b g q t x d

              rw[rank\_combine h₁, hrank\_of\_cons] at \*

              unfold bne at d

              rw[Nat.not\_beq\_eq\_true\_eq] at d

              have m : c.rank + 1 ≤ b.rank := by

                apply Nat.succ\_le\_of\_lt x

              constructor

              . rw[←h₁] at m

                apply lt\_of\_le\_of\_ne <;> assumption

              . assumption

              . assumption

        have h₃ : IsHeapForest' r y := by

          cases ht₁

          apply IsHeapForest'\_weaken

          . assumption

          . apply Nat.le\_of\_lt

            assumption

        have h₄ : min (hRank y) (hRank (combine le u c :: z)) ≤ hRank (mergeNodes le y (combine le u c :: z)) := by

          apply min\_hRank\_mergeNodes h₃ h₂

        repeat rw[hrank\_of\_cons] at \*

        rw[Nat.min\_eq] at h₄

        rw[rank\_combine h₁] at \*

        split at h₄

        . unfold bne at \*

          rw[Nat.not\_beq\_eq\_true\_eq] at \*

          simp at \*

          rename\_i h₅ h₆

          rw[h₅]

          assumption

        . assumption

      . rw[hrank\_of\_cons]

        rw[rank\_combine h₁]

        apply le\_rfl

theorem IsHeapForest'\_of\_IsHeapForest' : IsBinTree r → y < r.rank → IsHeapForest' r.rank x → IsHeapForest' y (r :: x) := by

intros h₁ h₂ h₃

constructor <;> assumption

theorem IsHeap\_merge (hxs : IsHeapForest' rx xs) (hys : IsHeapForest' ry ys) :  IsHeapForest' (min rx ry) (mergeNodes le xs ys) :=

  match xs, ys with

  | [], h  => by

    rw [temp]

    apply IsHeapForest'\_weaken hys

    apply min\_le\_right

  | h,  [] =>  by

    cases hxs

    . constructor

    . constructor

      . rw[Nat.min\_eq]

        split

        . assumption

        . rename\_i f j

          simp at j

          apply Nat.lt\_trans j

          assumption

      . assumption

      . assumption

  | (h₁ :: t₁), (h₂ :: t₂) => by

    unfold mergeNodes

    split

    . constructor

      . cases hxs

        rw[Nat.min\_eq]

        split

        . assumption

        . rename\_i f j h

          simp at h

          apply Nat.lt\_trans h

          assumption

      . cases hxs

        assumption

      . cases hxs

        . rename\_i f g j p

          have : h₁.rank = min h₁.rank (h₂.rank - 1)

          . have a : h₁.rank ≤ (h₂.rank - 1)

            . apply Nat.le\_pred\_of\_lt f

            have : min h₁.rank (h₂.rank - 1) = h₁.rank

            . apply min\_eq\_left a

            rw[this]

          rw [this]; clear this

          apply IsHeap\_merge (xs := t₁)

          . assumption

          . apply IsHeapForest'\_strengthen

            . assumption

            . cases g

              . rename\_i a

                have l : h₁.rank ≥ 0

                . cases h₁.rank

                  . simp

                  . simp

                have o : 0 ≤ h₁.rank

                apply ge.le l

                have i : 0 < h₂.rank

                apply lt\_of\_le\_of\_lt o f

                assumption

    . split

      . constructor

        . rw[Nat.min\_eq]

          split

          . cases hys

            apply Nat.lt\_of\_le\_of\_lt <;> assumption

          . cases hys

            assumption

        . cases hys

          assumption

        . have k : min h₂.rank (h₁.rank -1) = h₂.rank

          have a : h₂.rank ≤ (h₁.rank - 1)

          apply Nat.le\_pred\_of\_lt

          assumption

          apply min\_eq\_left

          assumption

          rw[←k]

          apply IsHeap\_merge

          . cases hys

            assumption

          . constructor

            . cases hxs

              apply Nat.pred\_lt

              simp

              rename\_i a b

              cases rx <;> apply Nat.not\_eq\_zero\_of\_lt; repeat assumption

            . cases hxs

              assumption

            . cases hxs

              assumption

      . dsimp

        split

        . split

          . rename\_i f g v x

            simp at f g

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            constructor

            . have o : (combine le h₁ h₂).rank = h₁.rank + 1

              apply rank\_combine i

              rw[o]

              rw[Nat.min\_eq]

              split

              . cases hxs

                apply Nat.lt\_succ\_of\_lt

                assumption

              . cases hys

                rw[i]

                apply Nat.lt\_succ\_of\_lt

                assumption

            . cases hxs

              cases hys

              apply combine\_trees\_IsBinTree <;> assumption

            . rw[rank\_combine i] at \*

              cases hxs

              rename\_i hh₁ hrx ht₁

              cases hys

              rename\_i hh₂ hry ht₂

              have ih := IsHeap\_merge ht₁ ht₂

              have o : min (hRank t₁) (hRank t₂) ≤ hRank (mergeNodes le t₁ t₂) := by

                rw[←i] at ht₂

                apply min\_hRank\_mergeNodes ht₁ ht₂

              by\_cases (t₁ = [] ∧ t₂ = [])

              . have he₁ := h.left

                have he₂ := h.right

                simp[he₁, he₂]

                constructor

              . apply IsHeapForest'\_strengthen2

                . apply ih

                . rw[Nat.min\_eq] at o

                  split at o

                  . cases ht₁

                    . rw[temp] at \*

                      cases ht₂

                      . simp at h

                      . rename\_i u y t q a w

                        rw[hrank\_of\_cons] at \*

                        rw[←i] at a

                        unfold bne at x

                        rw[Nat.not\_beq\_eq\_true\_eq] at x

                        have n : h₁.rank + 1 ≤ u.rank

                        apply Nat.succ\_le\_of\_lt a

                        apply lt\_of\_le\_of\_ne <;> assumption

                    . rename\_i u y t q a w

                      rw[hrank\_of\_cons] at \*

                      have l : h₁.rank < hRank (mergeNodes le (u :: y) t₂)

                      apply lt\_of\_lt\_of\_le <;> assumption

                      unfold bne at v

                      rw[Nat.not\_beq\_eq\_true\_eq] at v

                      have n : h₁.rank + 1 ≤ u.rank

                      apply Nat.succ\_le\_of\_lt a

                      have m : h₁.rank + 1 < u.rank

                      apply lt\_of\_le\_of\_ne <;> assumption

                      apply lt\_of\_lt\_of\_le <;> assumption

                  . cases ht₂

                    . rw[temp2] at \*

                      cases ht₁

                      . simp at h

                      . rename\_i u y t q a w

                        rw[hrank\_of\_cons] at \*

                        unfold bne at v

                        rw[Nat.not\_beq\_eq\_true\_eq] at v

                        have n : h₁.rank + 1 ≤ u.rank

                        apply Nat.succ\_le\_of\_lt a

                        apply lt\_of\_le\_of\_ne <;> assumption

                    . rename\_i u y t q a w

                      rw[hrank\_of\_cons] at \*

                      rw[←i] at a

                      have n : h₁.rank + 1 ≤ u.rank

                      apply Nat.succ\_le\_of\_lt a

                      unfold bne at x

                      rw[Nat.not\_beq\_eq\_true\_eq] at x

                      have m : h₁.rank + 1 < u.rank := by

                        apply lt\_of\_le\_of\_ne <;> assumption

                      apply lt\_of\_lt\_of\_le <;> assumption

          . apply IsHeap\_merge (ys := t₂)

            rename\_i f g v x

            simp at f g

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            . constructor

              . cases hxs

                . rw[rank\_combine i]

                  have t : h₁.rank  < h₁.rank + 1

                  apply Nat.lt\_succ\_self

                  rename\_i o p a

                  apply lt\_trans p t

              . cases hxs

                cases hys

                apply combine\_trees\_IsBinTree <;> assumption

              . cases hxs

                rename\_i w l q

                have k : (combine le h₁ h₂).rank = h₁.rank + 1

                apply rank\_combine i

                simp only [k] at \*; clear k

                cases q

                . constructor

                . constructor

                  . rename\_i e w q y a

                    have b : h₁.rank + 1 ≤ e.rank

                    apply Nat.succ\_le\_of\_lt a

                    apply lt\_of\_le\_of\_ne b

                    rw[hrank\_of\_cons] at v

                    unfold bne at v

                    rw[Nat.not\_beq\_eq\_true\_eq] at v

                    assumption

                  . assumption

                  . assumption

            . cases hys

              rename\_i f g

              apply IsHeapForest'\_lowerbound f g

        . split

          . apply IsHeap\_merge (xs := t₁)

            . cases hxs

              rename\_i f g

              apply IsHeapForest'\_lowerbound f g

            . constructor

              . rename\_i f g v x

                simp at f g

                have i : h₁.rank = h₂.rank

                apply LE.le.antisymm <;> assumption

                rw[rank\_combine i] at \*

                cases hys

                rw[i]

                apply Nat.lt\_succ\_of\_lt

                assumption

              . cases hxs

                cases hys

                rename\_i f g v x b a o p q z

                simp at f g

                have i : h₁.rank = h₂.rank

                apply LE.le.antisymm <;> assumption

                apply combine\_trees\_IsBinTree <;> assumption

              . rename\_i f g v x

                simp at f g

                have i : h₁.rank = h₂.rank

                apply LE.le.antisymm <;> assumption

                rw[rank\_combine i] at \*

                cases hys

                rename\_i q y P

                cases P

                . constructor

                . rename\_i l t r w c

                  rw[←i] at c

                  have b : h₁.rank + 1 ≤ l.rank

                  apply Nat.succ\_le\_of\_lt c

                  rw[hrank\_of\_cons] at x

                  constructor

                  . unfold bne at x

                    rw[Nat.not\_beq\_eq\_true\_eq] at x

                    apply lt\_of\_le\_of\_ne <;> assumption

                  repeat assumption

          . rename\_i f g v x

            simp at f g

            have i : h₁.rank = h₂.rank

            apply LE.le.antisymm <;> assumption

            constructor

            . rw[Nat.min\_eq]

              split

              . rw[rank\_combine i] at \*

                cases hxs

                apply Nat.lt\_succ\_of\_lt

                assumption

              . rw[rank\_combine i] at \*

                cases hys

                rw[i]

                apply Nat.lt\_succ\_of\_lt

                assumption

            . cases hxs

              cases hys

              apply combine\_trees\_IsBinTree <;> assumption

            . have i : h₁.rank = h₂.rank

              apply LE.le.antisymm <;> assumption

              rw[rank\_combine i] at \*

              cases hxs

              rename\_i hh₁ hrx ht₁

              cases hys

              rename\_i hh₂ hry ht₂

              have ih := IsHeap\_merge ht₁ ht₂

              have o : min (hRank t₁) (hRank t₂) ≤ hRank (mergeNodes le t₁ t₂) := by

                rw[←i] at ht₂

                apply min\_hRank\_mergeNodes ht₁ ht₂

              by\_cases (t₁ = [] ∧ t₂ = [])

              . have he₁ := h.left

                have he₂ := h.right

                simp[he₁, he₂]

                constructor

              . apply IsHeapForest'\_strengthen2

                . apply ih

                . rw[Nat.min\_eq] at o

                  split at o

                  . cases ht₂

                    . rw[temp2] at \*

                      cases ht₁

                      . contradiction

                      . rename\_i u y t q a w

                        rw[hrank\_of\_cons] at \*

                        unfold bne at \*

                        rw[Nat.not\_beq\_eq\_true\_eq] at \*

                        simp at \*

                        contradiction

                    . rename\_i u y t q a w

                      cases ht₁

                      . rw[temp] at \*

                        rw[hrank\_of\_cons] at \*

                        unfold bne at \*

                        rw[Nat.not\_beq\_eq\_true\_eq] at \*

                        simp at \*

                        contradiction

                      . rename\_i h₃ h₄ h₅ h₆ h₇

                        repeat rw[hrank\_of\_cons] at \*

                        unfold bne at \*

                        rw[Nat.not\_beq\_eq\_true\_eq] at \*

                        simp at \*

                        have ht₁ : IsHeapForest' h₁.rank (h₃ :: h₄) := by

                          apply IsHeapForest'\_of\_IsHeapForest' <;> assumption

                        have ht₂ : IsHeapForest' h₁.rank (u :: y) := by

                          rw[←i] at a

                          apply IsHeapForest'\_of\_IsHeapForest' <;> assumption

                        have h₈ : h₃.rank + 1 ≤ hRank (mergeNodes le (h₃ :: h₄) (u :: y))

                        apply hRank\_mergeNodes\_cons ht₁ ht₂

                        rw[x] at v

                        apply Eq.symm v

                        rw[v]

                        apply Nat.lt\_of\_succ\_le h₈

                  . cases ht₂

                    . rw[temp2] at \*

                      cases ht₁

                      . contradiction

                      . rw[hrank\_of\_cons] at \*

                        rename\_i h₃ h₄ h₅ h₆ h₇ h₈

                        unfold bne at v

                        rw[Nat.not\_beq\_eq\_true\_eq] at v

                        simp at v

                        contradiction

                    . rename\_i h₃ h₄ h₅ h₆ h₇ h₈

                      rw[hrank\_of\_cons] at \*

                      unfold bne at x

                      rw[Nat.not\_beq\_eq\_true\_eq] at x

                      simp at x

                      cases ht₁

                      . contradiction

                      . rename\_i m n k t q a

                        rw[hrank\_of\_cons] at \*

                        unfold bne at v

                        rw[Nat.not\_beq\_eq\_true\_eq] at v

                        simp at v

                        have ht₁ : IsHeapForest' h₁.rank (n :: k) := by

                          apply IsHeapForest'\_of\_IsHeapForest' <;> assumption

                        have ht₂ : IsHeapForest' h₁.rank (h₃ :: h₄) := by

                          rw[←i] at h₇

                          apply IsHeapForest'\_of\_IsHeapForest' <;> assumption

                        have h₈ : n.rank + 1 ≤ hRank (mergeNodes le (n :: k) (h₃ :: h₄))

                        apply hRank\_mergeNodes\_cons ht₁ ht₂

                        rw[x] at v

                        apply Eq.symm v

                        rw[v]

                        apply Nat.lt\_of\_succ\_le h₈

termination\_by \_  => xs.length + ys.length

decreasing\_by simp\_wf; simp\_arith [\*]

--maybe add more conclusions

-- add isheap stuff

theorem deleteMin\_non\_empty\_IsHeap (h₁ : IsHeap (heap xs)) (h₂ : IsMinHeap le (heap xs)): deleteMin le (heap xs) = some (y, (heap ys)) → IsHeap (heap ys) ∧ IsMinHeap le (heap ys) :=

  match xs with

  | [] => by

    unfold deleteMin

    dsimp

    sorry

  | [h] => by

    unfold deleteMin

    dsimp

    intro h₃

    rw[Option.some\_inj] at h₃

    apply And.intro

    . constructor

      sorry

    . cases h₂

      sorry

  | (h::hs) => by

    sorry

theorem deleteMin\_empty\_IsHeap :  deleteMin le xs = none → isEmpty xs := by

  intros h₁

  sorry