基于 COQ 证明的
二叉堆操作的算法验证

刘嘉奇 **韩潇申** **薛轩宇**

唐亦恺

实验环境

- Coq 8.15.12

1 概述

本报告旨在研究二叉堆的操作及其合法性证明，具体任务包括以下几个方面：

1. **定义局部破坏的二叉堆合法性性质**：首先，我们需要定义局部破坏的二叉堆的性质，然后需要定义当某节点向下移动或向上移动时，该二叉树仍然满足局部破坏性。对于向下移动，要求在全树的合法性保持不变；对于向上移动，同样需要保证整个树的合法性。我们将详细描述这些操作，并证明它们在执行时保持集合不变性和合法性。

2. **定义插入与删除最小元操作**：接下来，我们将定义二叉堆中的插入操作与删除最小元操作（默认二叉堆为小根堆），并证明这两个操作的正确性。

3. **定义并证明"满二叉树"性质**：最后，我们将定义"满二叉树"性质，并证明前述的插入和删除操作能够保持该性质。在证明过程中，将使用 Coq 的形式化方法，确保所有操作在执行后仍然符合满二叉树的结构要求。

本任务通过 Coq 语言进行形式化验证，确保二叉堆操作的正确性，特别是合法性和结构性质的保持。通过该报告，我们展示了如何在形式化验证中使用 Coq 进行复杂的数据结构操作验证，并提供了相应的理论证明过程。

2 难点

2.1 节点的唯一性

具体原因是现在的 BinaryTree Z Z 的定义方式是把点当作一个整数来考虑，并且最初定义了堆是 <= 的小顶堆, 这样就直接导致树中会出现两个 value 一样的节点。再加上**现在的节点就是一个整数**这个条件，我们没有办法获取节点之间的拓扑关系。而且现有的边的定义也是基于节点的 value 来判断的，所以在考虑 = 的情况下它就没法区分两

个 value 相等的点究竟是不是同一个节点。比如：
考虑两个节点 a, b。二者的 value 一样，父子的 value 都一样，但是他们分布在 root 节点的两个不同分支上。故而最初的定义没法说明 a 和 b 是两个不同的节点（根本原因是没法表示他的拓扑性质），最后导致很多地方的证明无法进行下去。我们针对这个问题尝试使用一些显示的语言来表述，大概思路为：

- 如果 x 既有左儿子又有右儿子，那么左右儿子不相等

- ne$_l$: $forally1y2, step_l bty1y2- > y1 <> y2$;

2.2 递归类型的定义

我们在尝试定义“满二叉树”性质时，需要用到节点的 num、index 和 depth 信息，但是初始的二叉树是用集合的方式定义的，无法通过递推或递归的方式得到给定点的 num、index 和 depth 信息。这使我们的初轮定义陷入了窘境。

在与助教的多次交互讨论中，我们发现用 Inductive 类型定义这几个量，可以很好的补充我们节点信息的缺失。例如 Index 的定义：

```
Inductive Index (bt: BinTree Z Z): Z -> Z -> Prop :=
  | index_invalid:
      forall v,
      ~BinaryTree.vvalid Z Z bt v ->
      Index bt v 0
  | index_root:
      forall v,
      BinaryTree.vvalid Z Z bt v ->
      Root bt v ->
      Index bt v 1
  | index_left:
      forall v cl d1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_l bt v cl ->
      Index bt v d1 ->
      Index bt cl (2 * d1)
  | index_right:
      forall v cr d1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_r bt v cr ->
      Index bt v d1 ->
      Index bt cr (2 * d1 + 1%Z).
```

Listing 1: BinaryTree 模块定义

我们通过对节点的多种情况进行分类讨论，可以很轻松地做到从根节点（根节点 index 为 1，无节点 index 则为 0）出发，每个左子的 index 会等于他父节点的两倍，而右子的 index 会等于他父节点的两倍加 1。以这样一种 Inductive 的方式，我们可以定义出二叉树上所有节点的 index。

3 定义部分

3.1 BinaryTree 的定义

```
Module BinaryTree.

Record BinaryTree (Vertex Edge: Type) := {
  vvalid : Vertex -> Prop;   (* 验证顶点有效性 *)
  evalid : Edge -> Prop;     (* 验证边有效性 *)
  src : Edge -> Vertex;      (* 给定边, 返回起始顶点 *)
  dst : Edge -> Vertex;      (* 给定边, 返回目标顶点 *)
  go_left: Edge -> Prop;     (* 判断给定边是否是指向左子树的边 *)
}.
```

Listing 2: BinaryTree 模块定义

该定义表示了一个带有两个参数 `Vertex` 和 `Edge` 的 `BinaryTree` 记录类型，包含了以下字段：

- `vvalid`：验证一个顶点是否有效。
- `evalid`：验证一条边是否有效。
- `src`：给定边，返回边的起点。
- `dst`：给定边，返回边的终点。
- `go_left`：判断某条边是否指向左子树。

GO_RIGHT 定义

```
Definition go_right (V E: Type) (bt: BinaryTree V E) (e: E): Prop :=
  ~ go_left _ _ bt e. (* go_right 是 go_left 的补集 *)
```

Listing 3: go_right 定义

`go_right` 函数表示如果一条边 `e` 不是指向左子树的边，那么它就是指向右子树的边。通过取反操作实现。

符号表示法的定义

```
Notation "bt '.(vvalid)'" := (vvalid _ _ bt) (at level 1).
Notation "bt '.(evalid)'" := (evalid _ _ bt) (at level 1).
Notation "bt '.(src)'" := (src _ _ bt) (at level 1).
Notation "bt '.(dst)'" := (dst _ _ bt) (at level 1).
Notation "bt '.(go_left)'" := (go_left _ _ bt) (at level 1).
Notation "bt '.(go_right)'" := (go_right _ _ bt) (at level 1).
```

Listing 4: 符号表示法的定义

这些定义通过符号表示法简化了对二叉树中各个字段的访问，例如 `bt.(vvalid)` 代表了 `vvalid` 函数对二叉树 `bt` 的应用。

STEP_AUX 定义

```
Record step_aux {V E: Type} (bt: BinaryTree V E) (e: E) (x y: V): Prop := (*V&E可以是传入的任意类型，相当于C++中的template*)
{
  step_evalid: bt.(evalid) e;
  step_src_valid: bt.(vvalid) x;
  step_dst_valid: bt.(vvalid) y;
  step_src: bt.(src) e = x;
  step_dst: bt.(dst) e = y;
}.
```

Listing 5: step_aux 定义

step_aux 是一个记录类型，描述了给定二叉树 bt 中的一条边 e 和顶点 x、y 满足的条件。它包含了以下字段：

- step_evalid：验证边 e 是否有效。
- step_src_valid：验证顶点 x 是否有效。
- step_dst_valid：验证顶点 y 是否有效。
- step_src：验证边 e 的起点是否为 x。
- step_dst：验证边 e 的终点是否为 y。

STEP_L 定义

```
Definition step_l {V E: Type} (bt: BinaryTree V E) (x y: V): Prop := (*e=x->y && x -> left -> y*)
  exists e, step_aux bt e x y /\ bt.(go_left) e.
```

Listing 6: step_l 定义

step_l 定义表示在二叉树 bt 中，存在一条边 e 连接顶点 x 和顶点 y，使得顶点 y 是顶点 x 的左子，并且满足边 e 是指向左子树的一条边。

类似的，step_r 和 step_u 分别代表到右子和到父节点的定义。

唯一性定义：LEGAL

```
Record legal {V E: Type} (bt: BinaryTree V E): Prop :=
{
  step_l_unique: forall x y1 y2, step_l bt x y1 -> step_l bt x y2 -> y1 = y2;
  step_r_unique: forall x y1 y2, step_r bt x y1 -> step_r bt x y2 -> y1 = y2;
  step_u_unique: forall x y1 y2, step_u bt x y1 -> step_u bt x y2 -> y1 = y2;
}.
```

Listing 7: legal

该记录定义了三个唯一性约束：

- step_l_unique：每个顶点最多有一个左子节点。
- step_r_unique：每个顶点最多有一个右子节点。
- step_u_unique：每个顶点最多有一个父节点。

唯一性补充定义：LEGAL_FS

```
Record legal_fs {V E: Type} (bt: BinaryTree V E): Prop :=
{
  ne_l:  forall y1 y2, step_l bt y1 y2 -> y1 <> y2;
  ne_r:  forall y1 y2, step_r bt y1 y2 -> y1 <> y2;
  ne_childchild: forall y y1 y2, step_l bt y y1 -> step_r bt y y2 -> y1 <> y2;
}.
```

Listing 8: legal_fs

该记录定义了以下不等式约束：

- ne_l：左子节点不相等。
- ne_r：右子节点不相等。
- ne_childchild：左右子节点不同。

反向定义：U_L_R

```
Record u_l_r {V E: Type} (bt: BinaryTree V E): Prop :=
{
  l_u:  forall y1 y2, step_l bt y1 y2 -> step_u bt y2 y1;
  r_u:  forall y1 y2, step_r bt y1 y2 -> step_u bt y2 y1;
  u_lr: forall y1 y2, step_u bt y1 y2 -> (step_l bt y2 y1 \/ step_r bt y2 y1);
}.
```

Listing 9: u_l_r

该记录定义了以下操作关系：

- l_u：如果一个节点是另一个节点的左子节点，那么它的父节点也可以通过 step_u 找到。
- r_u：如果一个节点是另一个节点的右子节点，那么它的父节点也可以通过 step_u 找到。
- u_lr：如果一个节点是另一个节点的父节点，那么该父节点要么是该节点的左子节点，要么是右子节点。

3.2 Heap 小顶堆定义

在 Coq 中，`Heap` 记录类型用于定义一个小顶堆，具体定义如下：

```
Record Heap (h: BinTree Z Z): Prop := (* 小顶堆 *)
{
  heap_l: forall x y: Z, BinaryTree.step_l h x y -> (x < y)%Z;
  heap_r: forall x y: Z, BinaryTree.step_r h x y -> (x < y)%Z;
  heap_legality: BinaryTree.legal h /\ BinaryTree.legal_fs h /\ BinaryTree.u_l_r h;
}.
```

Listing 10: 小顶堆定义

该定义包含三个主要部分：

- `heap_l`：对于每一对父节点和左子节点 `x` 和 `y`，如果存在从 `x` 到 `y` 的左子节点边 (`step_l`)，则 `x < y`。即父节点的值小于左子节点。
- `heap_r`：对于每一对父节点和右子节点 `x` 和 `y`，如果存在从 `x` 到 `y` 的右子节点边 (`step_r`)，则 `x < y`。即父节点的值小于右子节点。
- `heap_legality`：二叉堆的合法性，包含三项要求：
  - `BinaryTree.legal h`：二叉树的合法性。
  - `BinaryTree.legal_fs h`：二叉树的额外合法性约束（例如节点唯一性等）。
  - `BinaryTree.u_l_r h`：节点结构合法性（例如父子节点的关系）。

通过这些条件，`Heap` 记录类型确保了树的每个节点都符合小顶堆的要求，同时保证了树的结构是合法的。

3.3 节点定义

在 Coq 中，我们定义了不同类型的二叉树节点，并通过 'Prop' 类型表示它们的性质。以下是各个定义的详细说明：

```
Definition Leaf (V E: Type) (bt: BinTree V E) (v: V): Prop :=
  bt.(vvalid) v ->
  (~ exists y, BinaryTree.step_l bt v y) /\
  (~ exists y, BinaryTree.step_r bt v y).
```

Listing 11: Leaf 节点定义

- `Leaf` 定义了一个节点 `v` 是叶子节点的条件。如果节点 `v` 有效（即 `vvalid v`），并且不存在指向它的左子节点和右子节点（即，`step_l` 和 `step_r` 不成立），则该节点为叶子节点。

```
Definition Node (V E: Type) (bt: BinTree V E) (v: V): Prop :=
  ~ Leaf V E bt v.
```

Listing 12: Node 节点定义

- `Node` 定义了一个节点 `v` 是普通节点的条件。普通节点即非叶子节点，因此 `Node` 的定义为 `Leaf`，即节点 `v` 不是叶子节点。

```
Definition Lson_only_Node (V E: Type) (bt: BinTree V E) (v: V): Prop :=
  (~ exists y, BinaryTree.step_r bt v y) /\
  exists yl, BinaryTree.step_l bt v yl.
```

Listing 13: 仅有左子节点的节点定义

- `Lson_only_Node` 定义了一个节点 `v` 仅有左子节点的条件。该节点没有右子节点（即，`step_r` 不成立），但它有一个左子节点（即，存在一个 `yl`，满足 `step_l`）。

```
Definition Rson_only_Node (V E: Type) (bt: BinTree V E) (v: V): Prop :=
  (~ exists y, BinaryTree.step_l bt v y) /\
  exists yr, BinaryTree.step_r bt v yr.
```

Listing 14: 仅有右子节点的节点定义

- `Rson_only_Node` 定义了一个节点 `v` 仅有右子节点的条件。该节点没有左子节点（即，`step_l` 不成立），但它有一个右子节点（即，存在一个 `yr`，满足 `step_r`）。

```
Definition Lson_Rson_Node (V E: Type) (bt: BinTree V E) (v: V): Prop :=
  (exists y, BinaryTree.step_l bt v y ) /\
  exists y', BinaryTree.step_r bt v y'.
```

Listing 15: 有左子节点和右子节点的节点定义

- `Lson_Rson_Node` 定义了一个节点 `v` 同时拥有左子节点和右子节点的条件。该节点存在一个左子节点（即，`step_l` 成立），同时也存在一个右子节点（即，`step_r` 成立）。

3.4

集合不变性定义

```
Definition Abs (h: BinTree Z Z) (X: Z -> Prop): Prop :=
  X == h.(vvalid).
```

Listing 16: Abs 定义

`Abs` 定义了一个抽象条件，描述了一个集合 `X` 和二叉树 `h` 的顶点有效性（`vvalid`）的等价关系。具体来说，`X` 应当等于二叉树中顶点的有效性集合。即，`X` 包含所有有效顶点，并且不包含无效顶点。

连通性定义

```
Definition bintree_connected {V E: Type} (bt: BinTree V E): Prop :=
  exists root: V,
    bt.(vvalid) root /\
    (~ exists e, bt.(evalid) e /\ bt.(dst) e = root) /\
    (forall v,
      bt.(vvalid) v ->
      v <> root ->
```

```
      exists e, bt.(evalid) e /\ bt.(dst) e = v).
```

Listing 17: bintree_connected 定义

`bintree_connected` 定义了二叉树的连通性条件，要求：

- 存在一个根节点 `root`，使得 `vvalid root` 成立。
- 根节点没有父节点（即不存在一条边指向根节点）。
- 对于每个有效顶点 `v`（且 `v` 不是根节点），存在一条有效的边，从某个节点到达 `v`。

3.5 局部破坏性定义

局部破坏的堆定义

```
Record PartialHeap (h: BinTree Z Z): Prop := {
  (* 最多存在一个节点 v 违反堆的性质 *)
  exists_violation: forall v1 v2: Z,
    (* 如果 v1 和 v2 都违反堆的性质 *)
    (h.(vvalid) v1 /\
      exists y1: Z, (BinaryTree.step_l h v1 y1 \/ BinaryTree.step_r h v1 y1) /\ (v1 >
    y1)%Z) ->
    (h.(vvalid) v2 /\
      exists y2: Z, (BinaryTree.step_l h v2 y2 \/ BinaryTree.step_r h v2 y2) /\ (v2 >
    y2)%Z) ->
    (* 那么 v1 和 v2 必须是同一个节点 *)
    v1 = v2;
  partial_heap_legality: BinaryTree.legal h /\ BinaryTree.legal_fs h /\ BinaryTree.
    u_l_r h;
}.
```

Listing 18: PartialHeap 定义

- **exists_violation**：表示如果二叉树中的两个节点 $v1$ 和 $v2$ 都违反了堆的性质（即它们是有效节点并且存在一个子节点小于它们），则这两个节点必须是同一个节点。这意味着在堆中最多只能有一个节点违反堆的性质。
- **partial_heap_legality**：保证二叉树符合合法性条件，要求树的结构和节点关系满足堆的基本合法性约束（由 $BinaryTree.legal$、$BinaryTree.legal\_fs$ 和 $BinaryTree.u\_l\_r$ 表示）。

严格局部破坏的堆定义

```
Record StrictPartialHeap (h: BinTree Z Z): Prop := {
  (* 存在一个节点 v 违反堆的性质 *)
  exists_violation_strict: exists v: Z,
    (h.(vvalid) v /\ (* v 必须是合法节点 *)
    exists y: Z,
```

```
    ((BinaryTree.step_l h v y \/ BinaryTree.step_r h v y) /\ (v > y)%Z)) /\
    (
      forall v2: Z, (h.(vvalid) v /\ (* v 必须是合法节点 *)
      exists y: Z,
      ((BinaryTree.step_l h v2 y \/ BinaryTree.step_r h v2 y) /\ (v2 > y)%Z)) ->
      (v2 = v)
    );
  strict_partial_heap_legality: BinaryTree.legal h /\ BinaryTree.legal_fs h /\
    BinaryTree.u_l_r h;
}.
```

Listing 19: StrictPartialHeap 定义

在上述定义中，StrictPartialHeap 描述了一个严格局部破坏的堆。与 PartialHeap 不同，StrictPartialHeap 强制要求堆中最多 ** 只有一个节点 ** 违反堆的性质。具体来说：

- **存在违反堆的节点**：在堆中存在一个节点 v，它违反了堆的性质，即存在一个子节点 y 使得 $v > y$，且 v 必须是一个合法节点。
- **仅存在一个违反堆的节点**：对于所有其他可能违反堆性质的节点 v_2，如果 v_2 也违反了堆性质，且存在一个子节点 y_2 满足 $v_2 > y_2$，则 v_2 必须与 v 相同，即堆中最多只有一个违反堆性质的节点。
- **堆合法性条件**：strict_partial_heap_legality 确保二叉树符合堆的基本合法性要求（通过 BinaryTree.legal、BinaryTree.legal_fs 和 BinaryTree.u_l_r）。

为了方便证明，我们还引入了 StrictPartialHeap1，StrictPartialHeap2，StrictPartialHeap3 的定义，分别代表同时有左右子节点，并且比两个子节点都大、有左孩子且大于左孩子，左孩子如果存在则不大于左孩子、有右孩子且大于右孩子，左孩子如果存在则不大于左孩子。

Root 定义

```
Definition Root (h: BinTree Z Z) (v: Z): Prop :=
  h.(vvalid) v ->
  (~ exists y, BinaryTree.step_u h v y).
```

Listing 20: Root 定义

`Root` 定义表示，如果二叉树 `h` 中节点 `v` 是有效的（由 `vvalid` 判断），并且不存在从节点 `v` 出发的上一步边（通过 `BinaryTree.step_u` 函数表示），那么节点 `v` 就是该树的根节点。

preserve_vvalid 定义

```
Definition preserve_vvalid (bt bt': BinTree Z Z) : Prop :=
  BinaryTree.vvalid _ _ bt' == BinaryTree.vvalid _ _ bt.
```

Listing 21: preserve_vvalid 定义

preserve_vvalid 定义表示，在二叉树 bt 和 bt' 之间，如果节点的有效性（由 vvalid 判断）保持不变，即 bt 和 bt' 对应的节点有效性相等，那么该定义为真。

PRESERVE_EVALID 定义

```
Definition preserve_evalid (bt bt': BinTree Z Z) : Prop :=
  BinaryTree.evalid _ _ bt' == BinaryTree.evalid _ _ bt.
```

Listing 22: preserve_evalid 定义

preserve_evalid 定义表示，在二叉树 bt 和 bt' 之间，如果边的有效性（由 evalid 判断）保持不变，即 bt 和 bt' 对应的边有效性相等，那么该定义为真。

SWAP_SRC 定义

```
Definition swap_src (v1 v2: Z) (bt bt': BinTree Z Z) : Prop :=
  (forall e,
    BinaryTree.src _ _ bt' e =
      if Z.eq_dec (BinaryTree.src _ _ bt e) v1 then v2
      else if Z.eq_dec (BinaryTree.src _ _ bt e) v2 then v1
      else BinaryTree.src _ _ bt e) /\

  (* Ensure step relations are updated accordingly *)
  (forall x y,
    BinaryTree.step_l bt x y <->
    BinaryTree.step_l bt' (if Z.eq_dec x v1 then v2 else if Z.eq_dec x v2 then v1 else x)
                        (if Z.eq_dec y v1 then v2 else if Z.eq_dec y v2 then v1 else y)) /\

  (forall x y,
    BinaryTree.step_r bt x y <->
    BinaryTree.step_r bt' (if Z.eq_dec x v1 then v2 else if Z.eq_dec x v2 then v1 else x)
                        (if Z.eq_dec y v1 then v2 else if Z.eq_dec y v2 then v1 else y)).
```

Listing 23: swap_src 定义

swap_src 定义表示，在二叉树 bt 和 bt' 之间，源节点 v1 和 v2 交换后：1. 对于每一条边 e，如果边的源节点是 v1，则将其修改为 v2，如果源节点是 v2，则修改为 v1，否则保持不变。2. 左边 step_l 和右边 step_r 的关系也相应更新，确保交换后的节点和边保持一致。

SWAP_DST 定义

```
Definition swap_dst (v1 v2: Z) (bt bt': BinTree Z Z) : Prop :=
  forall e, BinaryTree.dst _ _ bt' e =
    if Z.eq_dec (BinaryTree.dst _ _ bt e) v1 then v2
    else if Z.eq_dec (BinaryTree.dst _ _ bt e) v2 then v1
    else BinaryTree.dst _ _ bt e.
```

Listing 24: swap_dst 定义

swap_dst 定义表示，在二叉树 bt 和 bt' 之间，目标节点 v1 和 v2 交换后：对于每一条边 e，如果边的目标节点是 v1，则将其修改为 v2，如果目标节点是 v2，则修改为 v1，否则保持不变。

SWAP_NODES 定义

```
Definition swap_nodes (v1 v2: Z) : StateRelMonad.M (BinTree Z Z) unit :=
  fun (bt: BinTree Z Z) (_:unit) (bt': BinTree Z Z) =>
    preserve_vvalid bt bt' /\
    preserve_evalid bt bt' /\
    swap_src v1 v2 bt bt' /\
    swap_dst v1 v2 bt bt' /\
    BinaryTree.go_left _ _ bt' = BinaryTree.go_left _ _ bt.
```

Listing 25: swap_nodes 定义

swap_nodes 定义表示一个状态变换操作，它接受一个二叉树 bt 和目标二叉树 bt'，并确保以下条件：1. preserve_vvalid：保持节点的有效性不变。2. preserve_evalid：保持边的有效性不变。3. swap_src：交换源节点 v1 和 v2。4. swap_dst：交换目标节点 v1 和 v2。5. go_left：确保二叉树的 go_left 操作在变换后仍保持一致。

该定义用来在保持某些性质（如节点和边的有效性）不变的前提下，交换两个节点的源和目标位置，同时保证其他结构性质（如左子树关系）也不发生变化。

MOVE_UP 定义

```
Definition move_up (v: Z): StateRelMonad.M (BinTree Z Z) unit :=
  fun (bt1: BinTree Z Z) (_: unit) (bt2: BinTree Z Z) =>
    (* 检查节点 v 是合法的 *)
    (exists parent, BinaryTree.step_u bt1 v parent ->
      (* 使用新的 swap_nodes_rel 交换节点 *)
      (swap_nodes v parent) bt1 tt bt2).
```

Listing 26: move_up 定义

move_up 定义表示一个状态变换操作，其目标是将节点 v 向上移动。操作首先通过检查节点 v 是否有父节点（通过 BinaryTree.step_u 关系）来验证节点是否合法。如果存在父节点 parent，则使用 swap_nodes 操作交换节点 v 和父节点 parent。这种状态变换操作是通过 StateRelMonad.M 来实现的，它在执行过程中将 bt1 变换为新的二叉树 bt2。

MOVE_DOWN 定义

```
Definition move_down (v: Z): StateRelMonad.M (BinTree Z Z) unit :=
  fun (bt1: BinTree Z Z) (_: unit) (bt2: BinTree Z Z) =>
    (exists child,
      (BinaryTree.step_l bt1 v child \/ BinaryTree.step_r bt1 v child) ->
      (swap_nodes v child) bt1 tt bt2).
```

Listing 27: move_down 定义

move_down 定义表示一个状态变换操作，其目标是将节点 v 向下移动。操作首先检查节点 v 是否有左子节点或右子节点（通过 BinaryTree.step_l 或 BinaryTree.step_r 关系）。如果存在子节点 child，则使用 swap_nodes 操作交换节点 v 和子节点 child。该操作通过 StateRelMonad.M 实现状态变换，最终将变换后的树保存在 bt2 中。

MOVE_UP_IN_PARTIAL_HEAP 定义

```
Definition move_up_in_partial_heap: StateRelMonad.M (BinTree Z Z) unit :=
  fun (bt1: BinTree Z Z) (_: unit) (bt2: BinTree Z Z) =>
    (* 首先确保输入是一个 PartialHeap *)
    PartialHeap bt1 /\
    (* 如果是完整的堆，保持不变 *)
    ((Heap bt1 /\ bt1 = bt2) \/
    (* 如果是 StrictPartialHeap2，找到其唯一的违反堆性质的节点 *)
    (StrictPartialHeap2 bt1 /\
      exists v yl,
        BinaryTree.step_l bt1 v yl /\ (v > yl)%Z /\
        (swap_nodes v yl) bt1 tt bt2) \/
    (* 如果是 StrictPartialHeap3，类似处理 *)
    (StrictPartialHeap3 bt1 /\
      exists v yr,
        (* 从 StrictPartialHeap3 的定义中获取违反性质的节点 v *)
        BinaryTree.step_r bt1 v yr /\ (v > yr)%Z /\
        (forall yl, BinaryTree.step_l bt1 v yl -> (v < yl)%Z) /\
        (swap_nodes v yr) bt1 tt bt2)).
```

Listing 28: move_up_in_partial_heap 定义

move_up_in_partial_heap 定义表示一个状态变换操作，其目标是对部分堆（PartialHeap）进行向上移动操作。首先，该操作确保输入的树 bt1 是一个部分堆（PartialHeap）。然后，基于堆的类型，进行以下变换：1. 如果输入是一个完整堆（Heap），则树不发生变化。2. 如果输入是一个严格部分堆 2（StrictPartialHeap2），则找到违反堆性质的唯一节点 v 和其左子节点 yl，并交换 v 和 yl。3. 如果输入是一个严格部分堆 3（StrictPartialHeap3），则找到违反堆性质的节点 v 和其右子节点 yr，并交换 v 和 yr。

该操作通过 StateRelMonad.M 实现状态变换，最终返回变换后的二叉树 bt2。

MOVE_DOWN_IN_PARTIAL_HEAP 定义

```
Definition move_down_in_partial_heap: StateRelMonad.M (BinTree Z Z) unit :=
  fun (bt1: BinTree Z Z) (_: unit) (bt2: BinTree Z Z) =>
    (* 首先确保输入是一个 PartialHeap *)
    PartialHeap bt1 /\
    (* 如果是完整的堆，保持不变 *)
    ((Heap bt1 /\ bt1 = bt2) \/
    (* 如果是 StrictPartialHeap1，交换 v 和较小的子节点 *)
    (StrictPartialHeap1 bt1 /\
      exists v yl yr,
        BinaryTree.step_l bt1 v yl /\
        BinaryTree.step_r bt1 v yr /\
```

```
        (v > yl)%Z /\ (v > yr)%Z /\
        (swap_nodes v (if (yl < yr)%Z then yl else yr)) bt1 tt bt2)).
```

Listing 29: move_down_in_partial_heap 定义

`move_down_in_partial_heap` 定义表示一个状态变换操作，其目标是对部分堆（PartialHeap）进行向下移动操作。首先，该操作确保输入的树 `bt1` 是一个部分堆（`PartialHeap`）。然后，基于堆的类型，进行以下变换：1. 如果输入是一个完整堆（`Heap`），则树不发生变化。2. 如果输入是一个严格部分堆 1（`StrictPartialHeap1`），则找到违反堆性质的节点 `v` 和其左右子节点 `yl` 和 `yr`，并交换 `v` 和较小的子节点（即 `yl` 或 `yr`）。

该操作通过 `StateRelMonad.M` 实现状态变换，最终返回变换后的二叉树 `bt2`。

Depth 定义

```
Inductive Depth (bt: BinTree Z Z): Z -> Z -> Prop :=
  | depth_invalid:
      forall v,
      ~BinaryTree.vvalid Z Z bt v ->
      Depth bt v 0
  | depth_leaf:
      forall v,
      BinaryTree.vvalid Z Z bt v ->
      (forall y, ~BinaryTree.step_l bt v y) ->
      (forall y, ~BinaryTree.step_r bt v y) ->
      Depth bt v 1
  | depth_left:
      forall v cl d1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_l bt v cl ->
      (~exists cr, BinaryTree.step_r bt v cr) ->
      Depth bt cl d1 ->
      Depth bt v (d1 + 1%Z)
  | depth_right:
      forall v cr d1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_r bt v cr ->
      (~exists cl, BinaryTree.step_l bt v cl) ->
      Depth bt cr d1 ->
      Depth bt v (d1 + 1%Z)
  | depth_left_right:
      forall v cl cr d1 d2,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_l bt v cl ->
      BinaryTree.step_r bt v cr ->
      Depth bt cl d1 ->
      Depth bt cr d2 ->
      Depth bt v ((Z.max d1 d2) + 1%Z).
```

Listing 30: Depth 定义

Depth 定义表示二叉树中某个节点 v 的深度。根据树结构和节点的位置，深度可以通过以下几种方式定义：

1. depth_invalid：如果节点 v 在二叉树中无效（即 BinaryTree.vvalid 不成立），则其深度为 0。2. depth_leaf：如果节点 v 是一个叶子节点（即没有左子树和右子树），且节点有效，那么深度为 1。3. depth_left：如果节点 v 有左子树且没有右子树，且节点有效，深度由左子树的深度加 1 得到。4. depth_right：如果节点 v 有右子树且没有左子树，且节点有效，深度由右子树的深度加 1 得到。5. depth_left_right：如果节点 v 同时有左子树和右子树，且节点有效，深度由左右子树的深度的最大值加 1 得到。

该定义使用归纳法来递归地计算每个节点的深度，并考虑节点的不同类型（无效、叶子节点、左子树、右子树、左右子树）。

Index 定义

```
Inductive Index (bt: BinTree Z Z): Z -> Z -> Prop :=
  | index_invalid:
      forall v,
      ~BinaryTree.vvalid Z Z bt v ->
      Index bt v 0
  | index_root:
      forall v,
      BinaryTree.vvalid Z Z bt v ->
      Root bt v ->
      Index bt v 1
  | index_left:
      forall v cl d1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_l bt v cl ->
      Index bt v d1 ->
      Index bt cl (2 * d1)
  | index_right:
      forall v cr d1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_r bt v cr ->
      Index bt v d1 ->
      Index bt cr (2 * d1 + 1%Z).
```

Listing 31: Index 定义

Index 定义表示二叉树中某个节点 v 的索引值，索引值与二叉树的结构密切相关。根据节点的不同位置，索引值的定义如下：

1. index_invalid：如果节点 v 在树中无效（不满足 vvalid），则该节点的索引值为 0。2. index_root：如果节点 v 是根节点（满足 Root），且 v 在树中有效，则索引值为 1。3. index_left：如果节点 v 的左子树为 cl，且节点 v 在树中有效，且左子树的索引为 d1，则节点 v 的索引为 2 * d1。4. index_right：如果节点 v 的右子树为 cr，且节点 v 在树中有效，且右子树的索引为 d1，则节点 v 的索引为 2 * d1 + 1。

通过递归定义，`Index` 可以计算出每个节点的索引，根节点的索引为 1，而左右子树的节点根据父节点的索引进行计算。

NumNodes 定义

```
Inductive NumNodes (bt: BinTree Z Z): Z -> Z -> Prop :=
  | num_nodes_invalid:
      forall v,
      ~BinaryTree.vvalid Z Z bt v ->
      NumNodes bt v 0
  | num_nodes_leaf:
      forall v,
      BinaryTree.vvalid Z Z bt v ->
      (forall y, ~BinaryTree.step_l bt v y) ->
      (forall y, ~BinaryTree.step_r bt v y) ->
      NumNodes bt v 1
  | num_nodes_left:
      forall v cl n1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_l bt v cl ->
      (~exists cr, BinaryTree.step_r bt v cr) ->
      NumNodes bt cl n1 ->
      NumNodes bt v (n1 + 1%Z)
  | num_nodes_right:
      forall v cr n1,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_r bt v cr ->
      (~exists cl, BinaryTree.step_l bt v cl) ->
      NumNodes bt cr n1 ->
      NumNodes bt v (n1 + 1%Z)
  | num_nodes_left_right:
      forall v cl cr n1 n2,
      BinaryTree.vvalid Z Z bt v ->
      BinaryTree.step_l bt v cl ->
      BinaryTree.step_r bt v cr ->
      NumNodes bt cl n1 ->
      NumNodes bt cr n2 ->
      NumNodes bt v (n1 + n2 + 1%Z).
```

Listing 32: NumNodes 定义

`NumNodes` 定义表示二叉树中某个节点 `v` 的子树节点数。具体来说，节点数的定义依赖于节点的位置和结构，分为以下几种情况：

1. `num_nodes_invalid`：如果节点 `v` 在树中无效（不满足 `vvalid`），则节点数为 0。2. `num_nodes_leaf`：如果节点 `v` 是一个叶子节点（没有左或右子节点），且在树中有效，则节点数为 1。3. `num_nodes_left`：如果节点 `v` 有左子树 `cl`，且没有右子树，则节点数为左子树节点数加 1。4. `num_nodes_right`：如果节点 `v` 有右子树 `cr`，且没有左子树，则节点数为右子树节点数加 1。5. `num_nodes_left_right`：如果节点 `v` 同时有左子树 `cl` 和右子树 `cr`，则节点数为左子树节点数、右子树节点数加上节点本身 1 的和。

这些规则通过递归地计算子树的节点数来定义每个节点的节点数。

MaxIndex 定义

```
Definition MaxIndex (bt: BinTree Z Z) (index: Z): Prop :=
  exists largest_v,
  Index bt largest_v index ->
  forall v index_v,
  v <> largest_v ->
  Index bt v index_v ->
  (index > index_v)%Z.
```

Listing 33: MaxIndex 定义

`MaxIndex` 定义表示在二叉树中，存在一个节点 `largest_v`，其索引值为 `index`，并且对于任何其他节点 `v` 和其索引值 `index_v`，如果 `v` 不是 `largest_v`，则 `index` 总是大于 `index_v`。

该定义确保了 `largest_v` 是具有最大索引值的节点。

FullHeap 定义

```
Record FullHeap (bt: BinTree Z Z) :=
  {
    full_heap_heap: Heap bt;
    full_heap_connected: bintree_connected bt;
    full_heap_full: exists root root_num max_index,
      Root bt root /\
      NumNodes bt root root_num /\
      MaxIndex bt max_index /\
      max_index = root_num
  }.
```

Listing 34: FullHeap 定义

`FullHeap` 是一个包含三部分信息的记录类型：

1. `full_heap_heap`：表示 `bt` 是一个堆（符合 `Heap` 定义）。2. `full_heap_connected`：表示二叉树是连通的，符合 `bintree_connected`。3. `full_heap_full`：包含一个存在的根节点 `root`，根节点的节点数 `root_num` 和最大索引值 `max_index`，并且满足以下条件：- `root` 是根节点（`Root bt root`）。- 根节点的子树节点数是 `root_num`（`NumNodes bt root root_num`）。- `max_index` 是具有最大索引的节点，且 `max_index` 等于 `root_num`（`MaxIndex bt max_index` 和 `max_index = root_num`）。

这个记录类型的目的是定义一个完全符合堆性质、连通并且具有最大索引的二叉树结构。

INSERT__NOT__SWAP 定义

```
Definition insert_not_swap (v: Z): StateRelMonad.M (BinTree Z Z) Z :=
  fun (bt1: BinTree Z Z) (v_return: Z) (bt2: BinTree Z Z) =>
    exists root,
    (* v不在原树中 *)
```

```
  ~ BinaryTree.vvalid Z Z bt1 v /\
  (* v_return是v的父节点 *)
  BinaryTree.step_u bt2 v v_return /\
  (* v是最末尾节点 *)
  Root bt2 root /\
  NumNodes bt2 root = Index bt2 v /\
  (* 原树中的所有节点在新树中保持不变 *)
  (forall v1, BinaryTree.vvalid Z Z bt1 v1 ->
    BinaryTree.vvalid Z Z bt2 v1 /\
    Index bt2 v1 = Index bt1 v1) /\
  (* 原树中节点间的关系保持不变 *)
  (forall v1 v2, BinaryTree.step_l bt1 v1 v2 <->
    BinaryTree.step_l bt2 v1 v2) /\
  (forall v1 v2, BinaryTree.step_r bt1 v1 v2 <->
    BinaryTree.step_r bt2 v1 v2).
```

Listing 35: insert_not_swap 定义

`insert_not_swap` 定义表示在二叉树中插入节点 `v` 时，保持原树的结构不变，并且在新树中执行插入操作。具体来说：

1. `v` 不在原树中，即原树中没有节点 `v`。2. `v_return` 是新树中 `v` 的父节点。3. `v` 是新树中的最末尾节点。4. 新树中所有节点的有效性和索引值保持与原树一致。5. 原树中的子树关系在新树中保持不变，即 `step_l` 和 `step_r` 关系保持一致。

此操作保证了插入操作仅仅是将 `v` 插入到合适的位置，而不改变树中其他节点的结构。

DELETE_ROOT 定义

```
Definition delete_root: StateRelMonad.M (BinTree Z Z) Z :=
 fun (bt1: BinTree Z Z) (v_return: Z) (bt2: BinTree Z Z) =>
  exists root last,
  (* 树不为空 *)
  Root bt1 root /\
  (* last是最后一个节点 *)
  BinaryTree.vvalid Z Z bt1 last /\
  (forall v i1 i2, BinaryTree.vvalid Z Z bt1 v ->
    Index bt1 v i1 -> Index bt1 last i2 -> (i1 <= i2)%Z) /\
  (* 返回被删除的根节点值 *)
  v_return = root /\
  (* 新树中last替换root位置 *)
  (forall v1 v2, BinaryTree.step_l bt2 v1 v2 <->
    (if Z.eq_dec v1 last
      then BinaryTree.step_l bt1 root v2
      else if Z.eq_dec v2 last
          then False
          else BinaryTree.step_l bt1 v1 v2)) /\
  (forall v1 v2, BinaryTree.step_r bt2 v1 v2 <->
    (if Z.eq_dec v1 last
      then BinaryTree.step_r bt1 root v2
      else if Z.eq_dec v2 last
          then False
          else BinaryTree.step_r bt1 v1 v2)) /\
  (* 原树中除root和last外的节点在新树中保持不变 *)
```

```
  (forall v, BinaryTree.vvalid Z Z bt1 v ->
    v <> root -> v <> last ->
    BinaryTree.vvalid Z Z bt2 v) /\
  (* 更新索引 *)
  (forall v, BinaryTree.vvalid Z Z bt2 v ->
    (if Z.eq_dec v last
      then Index bt2 v = Index bt1 root
      else Index bt2 v = Index bt1 v).
```

Listing 36: delete_root 定义

`delete_root` 定义表示在二叉树中删除根节点并更新树结构。具体来说：

1. 树不为空，存在根节点 `root` 和最后一个节点 `last`。2. 通过约束保证 `last` 是最后一个节点，并且所有节点的索引满足顺序关系。3. 被删除的根节点值 `v_return` 等于 `root`。4. 在新树中，`last` 替换了 `root` 位置，同时更新了左右子树之间的连接关系（边关系）。5. 除去根节点和最后一个节点之外，原树中的所有节点在新树中的有效性保持不变。6. 更新新树中的节点索引，确保 `last` 的索引与根节点的索引相等。

STATE__BASED__HEAP 定义

```
Definition state_based_Heap {bt: BinTree Z Z} (state: tree_state bt) : Prop :=
  Heap (tree bt state).  (* 使用 tree_state 中的 tree 字段进行堆性质的检查 *)
```

Listing 37: state__based__Heap 定义

`state_based_Heap` 定义表示通过 `tree_state` 中的 `tree` 字段来检查堆的性质。它利用 `Heap` 来验证二叉树是否符合堆的要求。

TEST__HEAP 定义

```
Definition test_heap {bt: BinTree Z Z} (s: tree_state bt): StateRelMonad.M (tree_state
    bt) unit :=
  fun s1 (_: unit) s2 => s1 = s2 /\ state_based_Heap s1.
```

Listing 38: test__heap 定义

`test_heap` 定义表示一个状态变换的操作，测试当前的 `tree_state` 是否满足堆性质。该操作接受当前状态 `s1` 和目标状态 `s2`，并要求两者相等，同时验证 `s1` 是否符合堆的性质（通过 `state_based_Heap`）。

INSERT__BODY 定义

```
Definition insert_body (bt: BinTree Z Z)
: StateRelMonad.M (BinTree Z Z) (ContinueOrBreak (BinTree Z Z) (BinTree Z Z)) :=
StateRelMonadOp.choice
  (StateRelMonadOp.test (fun state => Heap state) ;;
    StateRelMonadOp.break bt)
  (StateRelMonadOp.test (fun state => StrictPartialHeap state) ;;
    move_up_in_partial_heap;;
```

```
    StateRelMonadOp.continue bt).
```

Listing 39: insert_body 定义

`insert_body` 定义表示插入操作的主体部分。它基于当前状态来决定操作的下一步：1. 如果当前状态是一个堆（`Heap state`），则立即终止操作，返回当前树（`StateRelMonadOp.break bt`）。2. 如果当前状态是严格的部分堆（`StrictPartialHeap state`），则执行向上移动操作（`move_up_in_partial_heap`），然后继续进行插入操作，返回更新后的树（`StateRelMonadOp.continue bt`）。

DELETE_BODY 定义

```
Definition delete_body (bt: BinTree Z Z)
: StateRelMonad.M (BinTree Z Z) (ContinueOrBreak (BinTree Z Z) (BinTree Z Z)) :=
StateRelMonadOp.choice
  (StateRelMonadOp.test (fun state => Heap state) ;;
    StateRelMonadOp.break bt)
  (StateRelMonadOp.test (fun state => StrictPartialHeap state) ;;
    move_down_in_partial_heap;;
    StateRelMonadOp.continue bt).
```

Listing 40: delete_body 定义

`delete_body` 定义表示删除操作的主体部分。它基于当前状态来决定操作的下一步：1. 如果当前状态是一个堆（`Heap state`），则立即终止操作，返回当前树（`StateRelMonadOp.break bt`）。2. 如果当前状态是严格的部分堆（`StrictPartialHeap state`），则执行向下移动操作（`move_down_in_partial_heap`），然后继续进行删除操作，返回更新后的树（`StateRelMonadOp.continue bt`）。

INSERT_HEAP 定义

```
Definition insert_heap (v: Z): StateRelMonad.M (BinTree Z Z) Z :=
  fun (bt1: BinTree Z Z) (v_return: Z) (bt2: BinTree Z Z) =>
    exists bt_mid bt_tmp,
    (* First insert the node at the last position *)
    insert_not_swap v bt1 v_return bt_mid /\
    (* Then repeatedly apply insert_body until the heap property is restored *)
    repeat_break insert_body bt_mid bt_tmp bt2 bt2 ->
    (* Final tree bt2 should be a heap *)
    Heap bt2.
```

Listing 41: insert_heap 定义

`insert_heap` 定义表示在堆中插入一个节点并恢复堆的性质。它分为几个步骤：1. 首先，将节点 `v` 插入到树的最后位置（使用 `insert_not_swap`）。2. 然后，重复应用 `insert_body` 直到恢复堆的性质，直到树的结构符合堆的要求（通过 `repeat_break insert_body`）。3. 最终，返回的树 `bt2` 必须是一个有效的堆（`Heap bt2`）。

DELETE_HEAP 定义

```
Definition delete_heap (v: Z): StateRelMonad.M (BinTree Z Z) Z :=
  fun (bt1: BinTree Z Z) (v_return: Z) (bt2: BinTree Z Z) =>
    exists bt_mid bt_tmp,
    (* First delete the root node *)
    delete_root bt1 v_return bt_mid /\
    (* Then repeatedly apply delete_body until the heap property is restored *)
    repeat_break delete_body bt_mid bt_tmp bt2 bt2 ->
    (* Final tree bt2 should be a heap *)
    Heap bt2.
```

Listing 42: delete_heap 定义

`delete_heap` 定义表示在堆中删除一个节点并恢复堆的性质。它分为几个步骤：1. 首先，删除堆的根节点（使用 `delete_root`）。2. 然后，重复应用 `delete_body`，直到堆的性质得到恢复（通过 `repeat_break delete_body`）。3. 最终，返回的树 `bt2` 必须是一个有效的堆（`Heap bt2`）。

4 引理的提出及证明

NODE_TRIPLE

```
Lemma Node_triple :
  forall (V E: Type) (bt: BinTree V E) (v: V),
    Node V E bt v  <-> Lson_only_Node V E bt v \/ Rson_only_Node V E bt v \/
     Lson_Rson_Node V E bt v.
Proof.
```

Listing 43: Node_triple 引理

该引理证明了二叉树中节点的分类性质。具体来说，给定二叉树 bt 和节点 v，节点 v 要么是普通节点（即非叶子节点），要么属于下列三种情况之一：

- **只有左子节点**（$Lson\_only\_Node$）：节点 v 有一个左子节点，但没有右子节点；
- **只有右子节点**（$Rson\_only\_Node$）：节点 v 有一个右子节点，但没有左子节点；
- **同时有左子节点和右子节点**（$Lson\_Rson\_Node$）：节点 v 同时有左子节点和右子节点。

引理的证明基于节点 v 不是叶子节点的定义（即满足 $Node$ 的定义），并且对节点的子节点情况进行分类，得出结论。

NODE_TRIPLE 证明思路

- **-> 方向（从 Node 到三种情况之一）**：假设节点 v 是一个普通节点（Node），即 v 不是叶子节点，并且存在子节点。通过双重否定法（NNPP），假设 v 不属

于三种情况之一（Lson_only_Node，Rson_only_Node，Lson_Rson_Node），最终得到矛盾。然后检查 v 是否有右子节点，如果有右子节点，则它属于 Lson_Rson_Node；如果没有右子节点，再检查是否有左子节点。如果有左子节点，则 v 属于 Lson_only_Node。

- **<- 方向（从三种情况之一到 Node）**：如果节点 v 属于 Lson_only_Node、Rson_only_Node 或 Lson_Rson_Node 中的某一类，则证明 v 是一个普通节点（Node），即 v 不是叶子节点，且具有子节点。

STRICT_PARTIAL_HEAP_CLASSIFICATION

```
Theorem strict_partial_heap_classification:
  forall h: BinTree Z Z,
    StrictPartialHeap h -> StrictPartialHeap1 h \/ StrictPartialHeap2 h \/
     StrictPartialHeap3 h.
Proof.
```

Listing 44: strict_partial_heap_classification 定理

该定理证明了严格局部破坏堆（StrictPartialHeap）的分类。具体来说，给定二叉树 h，如果 h 满足 StrictPartialHeap 条件，那么它必定满足下列三种情况之一：

- 属于 StrictPartialHeap1 分类。
- 属于 StrictPartialHeap2 分类。
- 属于 StrictPartialHeap3 分类。

引理的证明过程通过将 StrictPartialHeap 堆结构分为这三种子类型之一来完成。通过这一分类，可以更细致地描述不同类型的局部堆破坏情况。

STRICT_PARTIAL_HEAP_CLASSIFICATION 证明思路

- **基础假设**：假设二叉树 h 满足 StrictPartialHeap 条件，首先通过堆的定义得到存在违反堆性质的节点 x_0。
- **分类讨论**：对于节点 x_0，首先检查它的子节点是否违反堆性质。
  - 如果 x_0 有右子节点违反堆性质（$x_0 > k$），则它属于 StrictPartialHeap1。在这种情况下，我们进一步讨论节点 x_0 和它的子节点，确保它满足 StrictPartialHeap1 的条件。
  - 如果没有右子节点违反堆性质，但左子节点违反堆性质，则它属于 StrictPartialHeap2。我们通过进一步分析节点 x_0 的结构得出这一结论。
  - 如果既有左子节点又有右子节点违反堆性质，则将其分类为 StrictPartialHeap3。这种情况涉及到复杂的子树结构的分析，通过相应的推理得出。

INVERSE__STRICT__PARTIAL__HEAP__CLASSIFICATION

```
Theorem inverse_strict_partial_heap_classification:
  forall h: BinTree Z Z,
    StrictPartialHeap1 h \/ StrictPartialHeap2 h \/ StrictPartialHeap3 h ->
     StrictPartialHeap h.
Proof.
```

Listing 45: inverse__strict__partial__heap__classification 定理

该定理证明了 StrictPartialHeap1、StrictPartialHeap2 和 StrictPartialHeap3 中任意一种堆的分类条件可以推导出 StrictPartialHeap。具体来说，给定二叉树 h，如果 h 满足以下三种情况之一：

- 属于 StrictPartialHeap1 分类。
- 属于 StrictPartialHeap2 分类。
- 属于 StrictPartialHeap3 分类。

那么我们可以得出 h 满足 StrictPartialHeap 条件。换句话说，StrictPartialHeap 的定义可以通过这三种分类之一的条件来反推。

INVERSE__STRICT__PARTIAL__HEAP__CLASSIFICATION 证明思路

- **分类讨论**：
  - **对于 StrictPartialHeap1**：假设堆满足 StrictPartialHeap1，即存在一个违反堆性质的节点 x，且该节点只有左子节点违反堆性质。通过堆的定义，构造该节点，并验证所有违反性质的节点满足堆条件。
  - **对于 StrictPartialHeap2**：假设堆满足 StrictPartialHeap2，即存在一个违反堆性质的节点 x，该节点左子节点违反堆性质。逐步分析堆中各节点的性质，确保其符合堆的合法性定义。
  - **对于 StrictPartialHeap3**：假设堆满足 StrictPartialHeap3，即存在违反堆性质的节点 x，且该节点有左子节点和右子节点都违反堆性质。通过类似的方法逐步验证堆的合法性。
- **堆的合法性**：在每种情况下，通过逐步分析子节点和父节点的关系，确保最终堆满足 StrictPartialHeap 的定义，即存在一个节点违反堆性质，且该节点违反的性质符合堆的定义。

PARTIAL__HEAP__CLASSIFICATION

```
Theorem partial_heap_classification:
  forall h: BinTree Z Z,
    PartialHeap h -> StrictPartialHeap h \/ Heap h.
```

Listing 46: partial__heap__classification 定理

该定理表明，如果一个二叉树 h 满足 PartialHeap 条件，那么它要么满足 StrictPartialHeap，要么满足 Heap 条件。证明的过程分为两部分：

- **有局部破坏的情况**：如果树 h 存在违反堆性质的节点，则进一步检查其是否满足 StrictPartialHeap 条件。通过对违反堆性质的节点的分析，可以得出其满足 StrictPartialHeap 的条件。
- **无局部破坏的情况**：如果树 h 不存在违反堆性质的节点，则进一步分析其是否满足 Heap 条件。通过排除法以及与 heap 的定义相结合，最终得出 h 满足 Heap 条件。

PARTIAL__HEAP__CLASSIFICATION 证明思路

- **PartialHeap 的定义**：PartialHeap 是指堆的某些性质被破坏，但最多只有一个节点违反堆的性质。通过对 PartialHeap 的定义进行分析，证明它要么是 StrictPartialHeap，要么是正常的 Heap。
- **分类讨论**：
  - 如果堆存在违反堆性质的节点，并且这些节点满足严格的条件，那么它属于 StrictPartialHeap。
  - 如果堆满足堆的所有基本性质（没有违反堆性质的节点或只有少数节点违反堆性质），那么它属于 Heap。
- **堆的合法性**：在每种情况下，通过逐步验证堆的合法性，确保堆满足 StrictPartialHeap 或 Heap 的定义。

THEOREM EQ__PH__SHH

```
Theorem eq_PH_SHH:
  forall h: BinTree Z Z,
    PartialHeap h <-> StrictPartialHeap h \/ Heap h.
Proof.
  intros.
  split.
  apply partial_heap_classification.
  apply inverse_partial_heap_classification.
Qed.
```

Listing 47: eq__PH__SHH 定理

该定理表明 PartialHeap h 与 StrictPartialHeap h Heap h 等价。也就是说，如果一个堆是部分堆（PartialHeap），那么它要么是严格部分堆（StrictPartialHeap），要么是一个普通堆（Heap）。反之，若堆满足严格部分堆或普通堆的条件，也就意味着它是一个部分堆。

Theorem eq_PH_SHH 证明思路

- **证明方向 1**（PartialHeap h -> StrictPartialHeap h Heap h）：
  - 使用 `partial_heap_classification` 引理，证明如果一个堆是部分堆（PartialHeap），则它必须是严格部分堆（StrictPartialHeap）或普通堆（Heap）。
- **证明方向 2**（StrictPartialHeap h Heap h -> PartialHeap h）：
  - 使用 `inverse_partial_heap_classification` 引理，证明如果堆满足严格部分堆或普通堆的条件，那么它就是一个部分堆（'PartialHeap'）。

Theorem eq_SH_SH123

```
Theorem eq_SH_SH123:
  forall h: BinTree Z Z,
    StrictPartialHeap h <-> StrictPartialHeap1 h \/ StrictPartialHeap2 h \/
     StrictPartialHeap3 h.
Proof.
  intros.
  split.
  apply strict_partial_heap_classification.
  apply inverse_strict_partial_heap_classification.
Qed.
```

Listing 48: eq_SH_SH123 定理

该定理表明 StrictPartialHeap h 与 StrictPartialHeap1 h StrictPartialHeap2 h StrictPartialHeap3 h 等价。也就是说，若堆是严格部分堆（StrictPartialHeap），它要么符合 StrictPartialHeap1，要么符合 StrictPartialHeap2，要么符合 StrictPartialHeap3。反之，若堆符合这三种分类之一的条件，那么它就满足严格部分堆的定义。

Theorem eq_SH_SH123 证明思路

- **证明方向 1**（StrictPartialHeap h -> StrictPartialHeap1 h StrictPartialHeap2 h StrictPartialHeap3 h）：
  - 使用 `strict_partial_heap_classification` 引理，证明如果一个堆是严格部分堆（StrictPartialHeap），则它符合 StrictPartialHeap1、StrictPartialHeap2 或 StrictPartialHeap3 之一的条件。
- **证明方向 2**（StrictPartialHeap1 h StrictPartialHeap2 h StrictPartialHeap3 h -> StrictPartialHeap h）：
  - 使用 `inverse_strict_partial_heap_classification` 引理，证明如果堆符合 StrictPartialHeap1、StrictPartialHeap2 或 StrictPartialHeap3 中的任意一种条件，那么它就是严格部分堆（StrictPartialHeap）。

Lemma simplify_if

```
Lemma simplify_if :
  forall (v yr : Z),
    (if Z.eq_dec v v then yr else if Z.eq_dec v yr then v else v) = yr.
Proof.
  intros v yr.

  (* Destruct the first Z.eq_dec v v *)
  destruct (Z.eq_dec v v) as [H_eq | H_neq].

  - (* Case 1: v = v (which is always true) *)
    (* The expression simplifies to yr *)
    reflexivity.

  - (* Case 2: v <> v (impossible, contradiction) *)
    exfalso. apply H_neq. reflexivity.
Qed.
```

Listing 49: simplify_if 引理

该引理简化了一个包含两层条件判断的表达式。根据 Z.eq_dec v v 始终为真，第一层条件判断总是成立，因此整个表达式简化为 yr。

Lemma simplify_if2

```
Lemma simplify_if2 :
  forall (v yr : Z),
    (if Z.eq_dec yr v
     then yr
     else if Z.eq_dec yr yr then v else yr) = v.
Proof.
  intros v yr.

  (* Destruct the first Z.eq_dec yr v *)
  destruct (Z.eq_dec yr v) as [H_eq | H_neq].

  - (* Case 1: yr = v *)
    (* In this case, the expression simplifies to yr, which is v *)
    lia.

  - (* Case 2: yr <> v *)
    (* The second if is always true, so the expression simplifies to v *)
    destruct (Z.eq_dec yr yr) as [H_eq' | H_neq'].
    + (* Case 2a: yr = yr (trivially true) *)
      reflexivity.
    + (* Case 2b: yr <> yr (impossible, contradiction) *)
      exfalso. apply H_neq'. reflexivity.
Qed.
```

Listing 50: simplify_if2 引理

该引理同样简化了一个包含两层条件判断的表达式。首先判断 yr 是否等于 v，若等于则简化为 yr。然后判断 yr 是否等于 yr，该判断始终成立，最终简化为 v。

Lemma simplify_if3

```
Lemma simplify_if3 :
  forall (x v yr: Z),
    (x <> v) /\ (x <> yr) /\ (v <> yr) ->
    (if Z.eq_dec x v then yr else if Z.eq_dec x yr then v else x) = x.
Proof.
  intros.
  destruct H. destruct H0.
  destruct (Z.eq_dec x v) as [H_eq | H_neq].
  - (* Case 1: x = v (contradiction with x <> v) *)
    exfalso. apply H_neq. exact H_eq.
  - (* Case 2: x <> v *)
    destruct (Z.eq_dec x yr) as [H_eq' | H_neq'].
    + (* Case 2a: x = yr (contradiction with x <> yr) *)
      exfalso. apply H_neq'. exact H_eq'.
    + (* Case 2b: x <> yr *)
      (* Both conditions are false, so the expression simplifies to x *)
      reflexivity.
Qed.
```

Listing 51: simplify_if3 引理

该引理简化了三层条件判断的表达式。首先判断 x 是否等于 v，如果等于则产生矛盾。接着判断 x 是否等于 yr，若都不成立，则表达式最终简化为 x。

simplify_if 证明思路

- **对于 simplify_if**：
  - 第一层条件 Z.eq_dec v v 总是成立，因此表达式直接简化为 yr。
- **对于 simplify_if2**：
  - 首先判断 yr 是否等于 v，如果是，则简化为 yr；否则，再判断 yr 是否等于 yr，该判断永远成立，最后简化为 v。
- **对于 simplify_if3**：
  - 判断 x 是否等于 v 和 yr，如果都不等，则最终简化为 x。

Theorem swap_nodes_up_edge

```
Theorem swap_nodes_up_edge:
  forall bt1 bt2 (v parent : Z),
    BinaryTree.step_u bt1 v parent ->
    (swap_nodes v parent) bt1 tt bt2 ->
    BinaryTree.step_u bt2 parent v.
Proof.
```

Listing 52: swap_nodes_up_edge 引理

该定理表明，当一个二叉树中的节点 v 和其父节点 $parent$ 在树中交换时，交换后的树仍然满足上升操作。具体地，如果在树 $bt1$ 中存在一个上升操作（step_u），并且通过 swap_nodes 操作交换了节点 v 和其父节点 $parent$ 后，得到树 $bt2$，那么树 $bt2$ 也应当存在从 $parent$ 到 v 的上升操作。

THEOREM SWAP_NODES_UP_EDGE 证明思路

- 通过对交换操作的定义进行展开，首先我们要验证在交换后的树 $bt2$ 中，边的有效性是否得以保留。
- 接着，我们需要证明交换操作没有破坏源节点和目标节点的有效性。具体来说，我们要证明原本从 v 到 $parent$ 的边的有效性在交换操作后得以保持。
- 最后，通过逻辑推理，结合已有条件，证明交换操作不会破坏树的结构，并且在交换后，$bt2$ 中仍然保持着从 $parent$ 到 v 的边。

THEOREM SWAP_NODES_LEFT_EDGE

```
Theorem swap_nodes_left_edge:
  forall bt1 bt2 (v parent : Z),
    BinaryTree.step_l bt1 v parent ->
    (swap_nodes v parent) bt1 tt bt2 ->
    BinaryTree.step_l bt2 parent v.
Proof.
```

Listing 53: swap_nodes_left_edge 引理

该定理表明，当一个二叉树中的节点 v 和其父节点 $parent$ 在树中交换时，交换后的树仍然满足左子树的边操作（step_l）。具体地，如果在树 $bt1$ 中存在一个从 v 到 $parent$ 的左子树边操作，并且通过 swap_nodes 操作交换了节点 v 和其父节点 $parent$ 后，得到树 $bt2$，那么树 $bt2$ 中也应当存在从 $parent$ 到 v 的左子树边操作。

THEOREM SWAP_NODES_LEFT_EDGE 证明思路

- 首先，展开交换操作的定义，证明交换后的树 $bt2$ 中的边有效性是否得以保留。
- 然后，证明交换操作没有破坏源节点和目标节点的有效性。具体来说，要证明原本从 v 到 $parent$ 的左子树边的有效性在交换操作后得以保持。
- 最后，结合已有条件，证明交换操作不会破坏树的结构，确保在交换后，$bt2$ 中仍然保持着从 $parent$ 到 v 的左子树边操作。

THEOREM SWAP_NODES_RIGHT_EDGE

```
Theorem swap_nodes_right_edge:
  forall bt1 bt2 (v parent : Z),
    BinaryTree.step_r bt1 v parent ->
    (swap_nodes v parent) bt1 tt bt2 ->
```

```
    BinaryTree.step_r bt2 parent v.
Proof.
```

Listing 54: swap_nodes_right_edge 引理

该定理表明，当一个二叉树中的节点 v 和其父节点 $parent$ 在树中交换时，交换后的树仍然满足右子树的边操作（step_r）。具体地，如果在树 $bt1$ 中存在一个从 v 到 $parent$ 的右子树边操作，并且通过 swap_nodes 操作交换了节点 v 和其父节点 $parent$ 后，得到树 $bt2$，那么树 $bt2$ 中也应当存在从 $parent$ 到 v 的右子树边操作。

Theorem swap_nodes_right_edge 证明思路

- 首先，展开交换操作的定义，证明交换后的树 $bt2$ 中的边有效性是否得以保留。
- 然后，证明交换操作没有破坏源节点和目标节点的有效性。具体来说，要证明原本从 v 到 $parent$ 的右子树边的有效性在交换操作后得以保持。
- 最后，通过逻辑推理，结合已有条件，证明交换操作不会破坏树的结构，并且在交换后，$bt2$ 中仍然保持着从 $parent$ 到 v 的右子树边操作。

Theorem swap_nodes_other_same_l

```
Theorem swap_nodes_other_same_l:
  forall bt1 bt2 (v1 v2 : Z),
    BinaryTree.legal_fs bt1 ->
    BinaryTree.u_l_r bt1 ->
    (BinaryTree.step_l bt1 v1 v2 \/ BinaryTree.step_r bt1 v1 v2 \/ BinaryTree.step_u
     bt1 v1 v2) ->
    (swap_nodes v1 v2) bt1 tt bt2 ->
    (forall v3 v4 : Z, v3 <> v1 -> v3 <> v2 -> v4 <> v1 -> v4 <> v2 -> BinaryTree.
     step_l bt1 v3 v4 -> BinaryTree.step_l bt2 v3 v4).
Proof.
```

Listing 55: swap_nodes_other_same_l 引理

该定理表明，若二叉树 $bt1$ 满足合法性条件（legal_fs）并且符合 u_l_r 条件，并且存在从 $v1$ 到 $v2$ 的左子树、右子树或上升操作（step_l、step_r、step_u），则交换操作（swap_nodes v1 v2）不会影响其他不相关的节点之间的左子树操作。也就是说，如果交换了节点 $v1$ 和 $v2$，那么在交换后的树 $bt2$ 中，所有与 $v1$ 和 $v2$ 不相关的节点 $v3$ 和 $v4$ 之间的左子树操作（step_l）依然成立。

Theorem swap_nodes_other_same_l 证明思路

- 首先，证明交换操作不影响与 $v1$ 和 $v2$ 不相关的节点 $v3$ 和 $v4$ 之间的操作。
- 然后，分析交换操作的定义，确保在交换后，二叉树中的结构和节点之间的关系得以保留。
- 最后，通过逻辑推理，结合已有条件，证明即使交换了 $v1$ 和 $v2$，其他不相关的节点之间的左子树操作依然是成立的。

Theorem swap_nodes_other_same_r

```
Theorem swap_nodes_other_same_r:
  forall bt1 bt2 (v1 v2 : Z),
    BinaryTree.legal_fs bt1 ->
    BinaryTree.u_l_r bt1 ->
    (BinaryTree.step_l bt1 v1 v2 \/ BinaryTree.step_r bt1 v1 v2 \/ BinaryTree.step_u
     bt1 v1 v2) ->
    (swap_nodes v1 v2) bt1 tt bt2 ->
    (forall v3 v4 : Z, v3 <> v1 -> v3 <> v2 -> v4 <> v1 -> v4 <> v2 -> BinaryTree.
     step_r bt1 v3 v4 -> BinaryTree.step_r bt2 v3 v4).
Proof.
```

Listing 56: swap_nodes_other_same_r 引理

该定理表明，若二叉树 $bt1$ 满足合法性条件（legal_fs）并且符合 u_l_r 条件，并且存在从 $v1$ 到 $v2$ 的左子树、右子树或上升操作（step_l、step_r、step_u），则交换操作（swap_nodes v1 v2）不会影响其他不相关的节点之间的右子树操作。也就是说，如果交换了节点 $v1$ 和 $v2$，那么在交换后的树 $bt2$ 中，所有与 $v1$ 和 $v2$ 不相关的节点 $v3$ 和 $v4$ 之间的右子树操作（step_r）依然成立。

Theorem swap_nodes_other_same_r 证明思路

- 首先，我们使用已知的合法性条件（legal_fs）和 u_l_r 条件，验证交换操作对树的影响。
- 然后，我们需要证明交换操作不会改变节点之间的右子树操作（step_r），即若树 $bt1$ 中存在一个从节点 $v3$ 到 $v4$ 的右子树操作，那么在树 $bt2$ 中，交换操作后同样会保持这一操作。
- 通过推理，我们能够确认在交换操作后，所有与 $v1$ 和 $v2$ 不相关的右子树操作依然有效，从而得出结论。

Theorem swap_nodes_other_same_u

```
Theorem swap_nodes_other_same_u:
  forall bt1 bt2 (v1 v2 : Z),
    BinaryTree.legal_fs bt1 ->
    BinaryTree.u_l_r bt1 ->
    (BinaryTree.step_l bt1 v1 v2 \/ BinaryTree.step_r bt1 v1 v2 \/ BinaryTree.step_u
     bt1 v1 v2) ->
    (swap_nodes v1 v2) bt1 tt bt2 ->
    (forall v3 v4 : Z, v3 <> v1 -> v3 <> v2 -> v4 <> v1 -> v4 <> v2 -> BinaryTree.
     step_u bt1 v3 v4 -> BinaryTree.step_u bt2 v3 v4).
Proof.
```

Listing 57: swap_nodes_other_same_u 引理

该定理表明，若二叉树 $bt1$ 满足合法性条件（legal_fs）并且符合 u_l_r 条件，并且存在从 $v1$ 到 $v2$ 的左子树、右子树或上升操作（step_l、step_r、step_u），则交换操作

(swap_nodes v1 v2) 不会影响其他不相关的节点之间的上升操作 (step_u)。也就是说，如果交换了节点 $v1$ 和 $v2$，那么在交换后的树 $bt2$ 中，所有与 $v1$ 和 $v2$ 不相关的节点 $v3$ 和 $v4$ 之间的上升操作依然成立。

Theorem swap_nodes_other_same_u 证明思路

- 首先，假设树 $bt1$ 满足 legal_fs 和 u_l_r 条件，并且存在从 $v1$ 到 $v2$ 的某种操作 (step_l、step_r 或 step_u)。
- 接着，我们通过交换操作 (swap_nodes v1 v2) 得到树 $bt2$，并假设交换后树的状态依然符合二叉树的有效性要求。
- 最后，我们需要证明，在交换后的树 $bt2$ 中，所有与 $v1$ 和 $v2$ 不相关的节点之间的上升操作 (step_u) 不会受到影响，仍然成立。

Lemma preserve_partial_heap_after_swap_strict2

```
Lemma preserve_partial_heap_after_swap_strict2:
  forall bt1 bt2 (v yl : Z),
    StrictPartialHeap2 bt1 ->
    BinaryTree.step_l bt1 v yl ->
    (v > yl)%Z ->
    (swap_nodes v yl) bt1 tt bt2 ->
    PartialHeap bt2.
Proof.
```

Listing 58: preserve_partial_heap_after_swap_strict2 引理

该引理表明，若二叉树 $bt1$ 满足 StrictPartialHeap2 条件，并且存在从节点 v 到 yl 的左子树操作 (step_l)，并且满足 $v > yl$，则交换节点 v 和 yl 后，二叉树 $bt2$ 仍然满足 PartialHeap 条件。

Lemma preserve_partial_heap_after_swap_strict2 证明思路

- 首先，我们需要验证交换操作不会破坏二叉树的结构。具体来说，交换操作后，树的约束条件应当保持不变。
- 其次，我们需要证明，即使交换了节点 v 和 yl，树仍然保持 PartialHeap 的属性。为了完成这一点，我们要确保交换操作不会影响树中其他节点的顺序和约束条件。

Lemma strict3_property

```
Lemma strict3_property:
 forall (bt1 : BinTree Z Z) (v yr : Z),
   (* 假设 bt1 是 StrictPartialHeap3 *)
   StrictPartialHeap3 bt1 ->
   BinaryTree.step_r bt1 v yr ->
   (forall v' yl : Z,
```

```
      BinaryTree.step_l bt1 v' yl -> (v' < yl)%Z) ->
  (v > yr)%Z ->
  (forall (v1 v2 : Z),
  v1 <> v ->
  v2 <> yr ->
  BinaryTree.step_r bt1 v1 v2 ->
  (v1 < v2)%Z).
```

Listing 59: strict3_property 引理

该引理表明，如果二叉树 $bt1$ 满足严格的部分堆属性 StrictPartialHeap3，并且存在一个从节点 v 到节点 yr 的右边操作（step_r），那么我们可以得出结论：对于所有不同于 v 和 yr 的节点 $v1$ 和 $v2$，如果从 $v1$ 到 $v2$ 存在右边操作（step_r），则 $v1$ 必定小于 $v2$。

Lemma strict3_property 证明思路

- 证明首先依赖于 StrictPartialHeap3 对应的二叉树性质，这些性质包括树的合法性和边的有效性。
- 然后，我们需要根据给定的 $step\_r$ 操作，逐步推导出对于任意的节点 $v1$ 和 $v2$，如果它们之间存在右边操作，则 $v1$ 必定小于 $v2$。
- 在推导过程中，需要证明一个反证法的推理，首先假设 $v1 \geq v2$，然后通过与 StrictPartialHeap3 的性质冲突来得到矛盾。
- 最终证明 $v1$ 和 $v2$ 必定满足所要求的关系 $(v1 < v2)$。

Theorem uni

```
Theorem uni:
  forall (bt1 : BinTree Z Z) (v x: Z),
    (* 假设 bt1 是 StrictPartialHeap3 *)
    BinaryTree.legal bt1 ->
    BinaryTree.legal_fs bt1 ->
    (v = x) ->
    (forall v1: Z, BinaryTree.step_l bt1 v v1 -> BinaryTree.step_l bt1 x v1)
    /\ (forall v2: Z, BinaryTree.step_r bt1 v v2 -> BinaryTree.step_r bt1 x v2)
    /\ (forall v3: Z, BinaryTree.step_u bt1 v v3 -> BinaryTree.step_u bt1 x v3).
```

Listing 60: uni 引理

该引理表明，如果 $v = x$ 且二叉树 $bt1$ 满足合法性条件 $BinaryTree.legal$ 和 $BinaryTree.legal\_fs$，那么在 $bt1$ 中，从 v 到任何子节点的操作（包括左边操作、右边操作和上升操作）与从 x 到这些子节点的操作是等效的。

Theorem uni 证明思路

- 由于 $v = x$，对于每个涉及 v 的操作，都会直接通过替换为 x 来证明它们等价。

- 通过证明对于所有的左边操作、右边操作和上升操作，都能保持相同的关系，从而完成该定理的证明。
- 对于每种操作类型，直接通过 substitution 进行简化，得出结论。

Lemma strict3_property2

```
Lemma strict3_property2:
  forall (bt1 : BinTree Z Z) (v yr x: Z),
    (* 假设 bt1 是 StrictPartialHeap3 *)
    StrictPartialHeap3 bt1 ->
    BinaryTree.legal bt1 ->
    BinaryTree.legal_fs bt1 ->
    BinaryTree.u_l_r bt1 ->
    BinaryTree.step_r bt1 v yr ->
    BinaryTree.step_l bt1 yr x ->
    (x <> v)%Z.
```

Listing 61: strict3_property2 引理

该引理表明，在满足以下条件的二叉树 $bt1$ 中：$bt1$ 是 StrictPartialHeap3、$bt1$ 合法且满足 u_{lr} 属性，同时存在从 v 到 yr 的右边操作（$step_r$）和从 yr 到 x 的左边操作（$step_l$），那么 x 和 v 必然不相等，即 $x \neq v$。

Lemma strict3_property2 证明思路

- 通过利用 $StrictPartialHeap3$ 的性质，可以推导出在树中，从一个节点到另一个节点的操作关系和节点之间的相对位置。
- 假设从 v 到 yr 是右边操作，从 yr 到 x 是左边操作，根据这些操作的定义和二叉树的结构，可以推导出 x 和 v 之间不可能相等。
- 通过对操作类型的分析，利用树的结构性质证明 $x \neq v$。

Lemma strict3_property3

```
Lemma strict3_property3:
  forall (bt1 : BinTree Z Z) (v yr: Z),
    (* 假设 bt1 是 StrictPartialHeap3 *)
    StrictPartialHeap3 bt1 ->
    BinaryTree.legal bt1 ->
    BinaryTree.legal_fs bt1 ->
    BinaryTree.u_l_r bt1 ->
    BinaryTree.step_r bt1 v yr ->
    (exists yl : Z, BinaryTree.step_l bt1 v yl).
```

Listing 62: strict3_property3 引理

该引理表明，假设 $bt1$ 是一个满足 StrictPartialHeap3 条件的二叉树，并且满足 v 到 yr 的右边操作（$step_r$），则存在一个节点 yl，使得从 v 到 yl 存在一个左边操作（$step_l$）。

Lemma strict3_property3 证明思路

- 通过分析二叉树 $bt1$ 的结构和 $StrictPartialHeap3$ 的性质，结合 $step_r$ 操作的定义，可以推导出从节点 v 出发存在另一个节点 yl，使得从 v 到 yl 存在左边操作（$step_l$）。
- 使用 $BinaryTree.u_{lr}$ 和其他合法性条件，证明在树中必然存在这样的左边操作。
- 最终，结合存在性量词，得出结论：存在一个节点 yl，满足从 v 到 yl 的左边操作。

Lemma preserve_legality_after_swap

```
Lemma preserve_legality_after_swap:
  forall bt1 bt2 (v yr : Z),
    (* 假设 bt1 是 StrictPartialHeap3 *)
    StrictPartialHeap3 bt1 ->
    (* v 与 yr 就是"唯一违反堆性质"的父与其右孩子 *)
    BinaryTree.step_r bt1 v yr ->
    (swap_nodes v yr) bt1 tt bt2 ->
    (* 结论: 交换后 bt2 仍然是 PartialHeap *)
    BinaryTree.legal bt2 /\ BinaryTree.legal_fs bt2 /\ BinaryTree.u_l_r bt2.
```

Listing 63: preserve_legality_after_swap 引理

该引理表明，在一个满足 StrictPartialHeap3 条件的二叉树 $bt1$ 中，如果存在一对节点 v 和 yr，使得 v 是其右孩子 yr 的父节点，并且 $step_r$ 操作成立，那么在交换节点 v 和 yr 后，得到的树 $bt2$ 仍然保持合法性，即满足以下条件：

- $bt2$ 是合法的二叉树（$BinaryTree.legalbt2$）。
- $bt2$ 满足树的完整性（$BinaryTree.legal_fsbt2$）。
- $bt2$ 满足左、右操作规则（$BinaryTree.u_{lr}bt2$）。

Lemma preserve_legality_after_swap 证明思路

- 首先，假设树 $bt1$ 满足 StrictPartialHeap3 条件，且 v 和 yr 满足右边操作（$step_r$）。我们通过分析交换操作的影响来推导结论。
- 交换节点后，新的树 $bt2$ 应该保持原有的合法性，特别是要证明交换操作没有破坏树的完整性或左右操作规则。
- 最终，结合合法性条件，得出结论：交换节点后，树 $bt2$ 仍然是合法的二叉树，并且满足左、右操作规则。

Lemma preserve__partial__heap__after__swap__strict3

```
Lemma preserve_partial_heap_after_swap_strict3:
  forall bt1 bt2 (v yr : Z),
    (* 假设 bt1 是 StrictPartialHeap3 *)
    StrictPartialHeap3 bt1 ->
    (* BinaryTree.legal_fs bt1 ->
    BinaryTree.legal_fs bt2 ->
    BinaryTree.u_l_r bt1->
    BinaryTree.u_l_r bt2->
    BinaryTree.legal bt1 ->
    BinaryTree.legal bt2 -> *)
    (* v 与 yr 就是"唯一违反堆性质"的父与其右孩子 *)
    BinaryTree.step_r bt1 v yr ->
    (v > yr)%Z ->
    (swap_nodes v yr) bt1 tt bt2 ->
    (* 结论: 交换后 bt2 仍然是 PartialHeap *)
    PartialHeap bt2.
```

Listing 64: preserve__partial__heap__after__swap__strict3 引理

该引理表明，在一个满足 StrictPartialHeap3 条件的二叉树 $bt1$ 中，如果存在一对节点 v 和 yr，使得 v 是其右孩子 yr 的父节点，并且 $step_r$ 操作成立，且 $v > yr$，则在交换节点 v 和 yr 后，得到的树 $bt2$ 仍然保持 PartialHeap 性质。具体地：

- 交换操作通过 *swap__nodes* 实现，交换前后树的合法性和其他相关性质会得到保持。
- 交换后的树 $bt2$ 满足 PartialHeap 的定义，即树的结构满足严格的堆性质。

Lemma preserve__partial__heap__after__swap__strict3 证明思路

- 首先，假设树 $bt1$ 满足 StrictPartialHeap3 条件，并且给定了父节点 v 和右孩子 yr，同时保证 $v > yr$。
- 接着，通过对交换操作的分析，我们需要证明交换后得到的树 $bt2$ 仍然符合 PartialHeap 的定义。
- 通过结合 *preserve__legality__after__swap* 引理，我们可以推导出交换操作不会破坏树的合法性和堆结构，从而得出结论 $bt2$ 仍然是 PartialHeap。

Theorem swap__swap__nodes

```
Theorem swap_swap_nodes:
  forall bt1 bt2 (v1 v2 : Z),
    (swap_nodes v1 v2) bt2 tt bt1 ->
    (swap_nodes v1 v2) bt1 tt bt2.
```

Listing 65: swap__swap__nodes 定理

该定理说明了交换节点操作是可逆的：若通过交换节点 $v1$ 和 $v2$ 得到树 $bt1$，那么通过再次交换 $v1$ 和 $v2$ 可以恢复到原始树 $bt2$。

SWAP_SWAP_NODES 证明思路

- 首先，我们使用 *swap_nodes_eq* 引理来处理该问题。这个引理表明，交换节点是可逆的，因此我们可以得到结论 $(swap\_nodes\ v1\ v2)\ bt2\ tt\ bt1 \rightarrow (swap\_nodes\ v1\ v2)\ bt1\ tt\ bt2$。
- 接下来，我们分解假设，假设 $swap\_nodes\ v1\ v2\ bt2\ tt\ bt1$ 成立，我们需要证明 $swap\_nodes\ v1\ v2\ bt1\ tt\ bt2$。
- 在证明过程中，我们逐一证明了 *preserve_vvalid*、*preserve_evalid* 以及 *swap_src* 等条件的对称性。
- 具体地，*vvalid* 和 *evalid* 是等价关系，因此可以利用对称性直接证明它们在交换前后保持不变。
- 对于源节点交换的证明，我们通过交换节点的对称性来推导源节点的交换关系。

SWAP_SWAP_NODES 证明步骤

1. **使用 *swap_nodes_eq* 引理：**通过应用 *swap_nodes_eq*，我们能够反向证明节点交换操作的对称性。
2. **分解假设：**将假设 $swap\_nodes\ v1\ v2\ bt2\ tt\ bt1$ 分解为多个子假设，涉及到 *vvalid*、*evalid* 和源节点交换的相关性。
3. **证明 *preserve_vvalid* 和 *preserve_evalid*：**利用 *vvalid* 和 *evalid* 的等价性，我们可以直接通过对称性证明这些关系在交换操作前后保持不变。
4. **证明 *swap_src*：**对源节点交换的对称性进行证明，确保源节点交换关系在交换前后保持一致。

LEMMA LSON_ONLY_UP

```
Lemma Lson_only_up:
  forall bt1 bt2 (x x0 : Z),
    PartialHeap bt1 ->
    (x > x0)%Z ->
    Lson_only_Node Z Z bt1 x0 ->
    (swap_nodes x x0) bt1 tt bt2 ->
    StrictPartialHeap bt2 \/ Heap bt2.
Proof.
```

Listing 66: Lson_only_up 引理

LSON_ONLY_UP 证明思路

该引理讨论了在交换节点 x 和 $x0$ 后，树 $bt2$ 是否依然满足堆性质。根据 *Lson_only_Node* 的性质，交换操作后可以推导出 *StrictPartialHeap* 或 *Heap* 的性质。

Lemma Rson_only_up

```
Lemma Rson_only_up:
  forall bt1 bt2 (x x0 : Z),
    PartialHeap bt1 ->
    (x > x0)%Z ->
    Rson_only_Node Z Z bt1 x0 ->
    (swap_nodes x x0) bt1 tt bt2 ->
    StrictPartialHeap bt2 \/ Heap bt2.
Proof.
```

Listing 67: Rson_only_up 引理

Rson_only_up 证明思路

与上一个引理类似，$Rson\_only\_Node$ 是交换操作的前提，意味着交换节点 x 和 $x0$ 后，树 $bt2$ 满足堆性质。该引理说明通过交换操作可以得到两种可能的结论，$StrictPartialHeap$ 或 $Heap$。

Lemma Lson_Rson_up

```
Lemma Lson_Rson_up:
  forall bt1 bt2 (x x0 : Z),
    PartialHeap bt1 ->
    (x > x0)%Z ->
    Lson_Rson_Node Z Z bt1 x0 ->
    (swap_nodes x x0) bt1 tt bt2 ->
    StrictPartialHeap bt2 \/ Heap bt2.
Proof.
```

Listing 68: Lson_Rson_up 引理

Lson_Rson_up 证明思路

该引理综合了 $Lson\_only\_Node$ 和 $Rson\_only\_Node$ 的性质，进一步表明交换节点 x 和 $x0$ 后，树 $bt2$ 仍然可能满足堆性质。结论同样是 $StrictPartialHeap$ 或 $Heap$。

Lemma Leaf_up

```
Lemma Leaf_up:
  forall bt1 bt2 (x x0 : Z),
    PartialHeap bt1 ->
    (x > x0)%Z ->
    Leaf Z Z bt1 x0 ->
    (swap_nodes x x0) bt1 tt bt2 ->
    StrictPartialHeap bt2 \/ Heap bt2.
Proof.
```

Listing 69: Leaf_up 引理

Leaf_up 证明思路

该引理考虑交换操作作用于叶节点的情况。通过交换节点 x 和 $x0$ 后，我们可以推导出树 $bt2$ 满足 $StrictPartialHeap$ 或 $Heap$ 中的某一性质。

5 主要定理的证明

Lemma Abs_swap_nodes

```
Lemma Abs_swap_nodes:
  forall bt1 bt2 v yr X,
    swap_nodes v yr bt1 tt bt2 ->
    Abs bt1 X ->
    Abs bt2 X.
```

Listing 70: Abs_swap_nodes 引理

该引理表明，当节点 v 和 yr 在树 $bt1$ 和 $bt2$ 中交换时，若树 $bt1$ 满足集合不变性 Abs，则树 $bt2$ 也会满足同样的性质 Abs。具体地，如果交换操作 $swap\_nodes$ 将节点 v 和 yr 从树 $bt1$ 交换为树 $bt2$，并且树 $bt1$ 满足集合不变性 Abs，那么树 $bt2$ 也必然满足相同的性质。证明过程如下：

- 交换操作首先将节点 v 和 yr 交换，并且通过 $swap\_nodes$ 的定义，我们可以得知交换后的树 $bt2$ 在 $vvalid$ 集合上的性质保持不变。
- 使用 $preserve\_vvalid$ 的定义，我们得出树 $bt1$ 和树 $bt2$ 的 $vvalid$ 集合相等。
- 最后，基于 $bt1$ 满足 Abs 的假设，我们可以得出 $bt2$ 同样满足 Abs。

Theorem move_up_in_partial_heap_preserves_partial_heap

```
Theorem move_up_in_partial_heap_preserves_partial_heap:
  forall bt1 bt2,
    PartialHeap bt1 ->
    move_up_in_partial_heap bt1 tt bt2 ->
    PartialHeap bt2.
Proof.
intros.
destruct H0.
destruct H1.
- destruct H1 as [_ H1].
  rewrite <- H1.
  exact H.
- destruct H1.
  -- destruct H1.
     destruct H2.
     destruct H2.
     destruct H2.
     destruct H3.
     apply inverse_partial_heap_classification.
     destruct (classic (Node Z Z bt1 x0)) as [H_is_node | H_is_leaf].
```

```
    + apply Node_triple in H_is_node.
      destruct H_is_node.
      ++ specialize (Lson_only_up bt1 bt2 x x0).
         intro.
         tauto.
      ++ destruct H5.
         +++ specialize (Rson_only_up bt1 bt2 x x0).
            tauto.
         +++ specialize (Lson_Rson_up bt1 bt2 x x0).
            tauto.
    + right.
      assert (Leaf Z Z bt1 x0).
      {
        unfold Node in H_is_leaf.
        tauto.
      }
      specialize (Leaf_up bt1 bt2 x x0).
      tauto.
 -- (* 证明所有左子节点满足堆性质 *)
    destruct H1.
    destruct H2.
    destruct H2.
    destruct H2.
    destruct H3.
    apply inverse_partial_heap_classification.
    destruct (classic (Node Z Z bt1 x0)) as [H_is_node | H_is_leaf].
    + apply Node_triple in H_is_node.
      destruct H_is_node.
      ++ specialize (Lson_only_up bt1 bt2 x x0).
         intro.
         tauto.
    ++ ...
```

Listing 71: move__up__in__partial__heap__preserves__partial__heap

证明思路

- ** 第一步：** 通过解构 move__up__in__partial__heap 的假设，来处理树的不同部分。

- ** 第二步：** 使用逆向的部分堆分类（inverse__partial__heap__classification）来验证节点是否为内节点或叶节点。

- ** 第三步：** 通过分别应用 Lson__only__up、Rson__only__up 和 Leaf__up 引理，确保树的结构在操作后仍满足部分堆（Partial Heap）的性质。

- ** 第四步：** 对左子节点和右子节点的处理通过分类和递归应用，确保满足堆的定义。

MOVE__DOWN__IN__PARTIAL__HEAP__PRESERVES__PARTIAL__HEAP

命题:

```
forall bt1 bt2,
```

```
PartialHeap bt1 ->
move_down_in_partial_heap bt1 tt bt2 ->
PartialHeap bt2.
```

证明思路：

1. **假设：** 首先假设 $bt1$ 是一个部分堆（PartialHeap），并且通过操作 $move\_down\_in\_partial\_heap$ 从 $bt1$ 得到 $bt2$。
2. **情况分析：** - 我们分析操作后的树结构。如果树结构没有发生变化，仅仅是标签发生了变化，那么部分堆性质显然被保留。- 如果树结构发生了变化，我们继续分析不同类型的节点：
   - **节点情况：** 如果当前节点不是叶子节点，我们应用诸如 Lson_only_up 和 Rson_only_up 等属性来确保堆性质得到维护。同时，对于左右子节点同时存在的情况，使用 Lson_Rson_up 处理。
   - **叶子节点情况：** 如果当前节点是叶子节点，直接应用 Leaf_up 进行证明。
3. **通过分类讨论，最后得出结论：** 根据所有的情况分析，我们得出 $bt2$ 仍然满足部分堆的性质。

结论： 因此，$move\_down\_in\_partial\_heap$ 操作在部分堆中执行后，仍然保持部分堆性质。

定理：MOVE_UP_IN_PARTIAL_HEAP 变换保持节点集合不变

命题：

$$\forall bt1\, bt2\, X,\ \mathrm{PartialHeap}(bt1) \to \mathrm{move\_up\_in\_partial\_heap}(bt1, tt, bt2) \to \mathrm{Abs}(bt1, X) \to \mathrm{Abs}(bt2, X)$$

证明思路：

1. **引入假设** 假设 $bt1$ 是一个符合 PartialHeap 的二叉树，通过 move_up_in_partial_heap 操作得到树 $bt2$。同时假设 $bt1$ 满足 Abs 定义，即 $Abs(bt1, X)$。

2. **展开 move_up_in_partial_heap 的定义**展开 move_up_in_partial_heap 操作的定义，分析其对树的结构变化的影响。特别是关注该操作是否会影响节点集合 X 的结构。

3. **分类讨论** - 如果涉及节点交换（例如使用 Abs_swap_nodes），我们可以利用该引理推导出节点集合 X 保持不变。- 如果有其他类型的树结构调整（如父节点和子节点的交换），我们需要逐步分析并确保这些操作不会破坏节点集合的正确性。

4. **结论：** 经过以上分析，我们证明操作后的树 $bt2$ 仍然满足 $Abs(bt2, X)$，从而 $Abs(bt2, X)$ 始终成立。

定理 MOVE__DOWN__IN__PARTIAL__HEAP__PRESERVES__SET__OF__NODES

命题：

```
forall bt1 bt2 X,
    PartialHeap bt1 ->
    move_down_in_partial_heap bt1 tt bt2 ->
    Abs bt1 X ->
    Abs bt2 X.
```

证明思路：

1. **引入假设：** 假设 $bt1$ 是一个部分堆（PartialHeap），并且通过 move__down__in__partial__heap 操作得到 $bt2$。假设 $Abs(bt1, X)$ 成立，即 $bt1$ 和 X 之间的关系有效。
2. **展开 move__down__in__partial__heap 的定义：** 将 move__down__in__partial__heap 的定义展开，分析树的变化。我们需要通过节点交换或结构改变来确保 $Abs(bt2, X)$ 成立。
3. **分类讨论：**
   - **节点交换情况：** 如果交换了节点，则可以使用类似 Abs__swap__nodes 的引理来确保交换前后的树仍然满足 Abs 定义。
   - **结构变化情况：** 如果树的结构发生了变化，分析每个步骤后节点集合是否仍然保持不变。
4. **结论：** 无论哪种情况，通过适当的引理应用和推导，最终证明 $Abs(bt2, X)$ 成立，即树的节点集合保持不变。