

Sonic Airdrop Contracts

Security Review

Cantina Managed review by:

Saw-mon and Natalie, Lead Security Researcher
Cccz, Security Researcher

July 18, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Critical Risk . . . 4
  - 3.1.1 `segment` is incorrectly updated with a normalised quantity . . . 4
- 3.2 High Risk . . . 6
  - 3.2.1 Making limit orders may cost excessive funds . . . 6
  - 3.2.2 `cancelAndMakeLimitOrders()` may not return funds . . . 6
  - 3.2.3 Excess NFTs are gotten when adding order book fails . . . 8
  - 3.2.4 Users can convert NFT to `quoteToken` when claiming orders . . . 9
  - 3.2.5 Attacker can provide `S` to steal the equal amount of quoteToken . . . 11
  - 3.2.6 Stale value of `tombstoneOffset` is used when the `price` is updated in `_addToBookSide` 13
- 3.3 Medium Risk . . . 14
  - 3.3.1 A new limit order might get added before an older limit order . . . 14
  - 3.3.2 `cancelAndMakeLimitOrders()` may fail due to insufficient balance of the user . . . 17
  - 3.3.3 Rounding down when calculating fees will result in users failing to claim tokens . . . 19
- 3.4 Low Risk . . . 20
  - 3.4.1 No boundary checks for `instantClaimBps` . . . 20
  - 3.4.2 `next` and `prev` can be called on a removed tree node . . . 21
  - 3.4.3 Parent of the `EMPTY` node might not be `EMPTY` . . . 21
  - 3.4.4 Off-by-one issue in `SonicAirdropNFT.balanceOf()` . . . 24
  - 3.4.5 An existing airdrop season with `startTime == 0` can be re-added . . . 25
  - 3.4.6 Tradable timestamp regions do not match in `setTokenIdInfos` and `refreshIsTradeable` . . . 26
  - 3.4.7 Unsafe cast from `uint256` to `uint88` . . . 27
  - 3.4.8 Potential overflow when increasing `claimableTokenInfo.amount` . . . 27
- 3.5 Gas Optimization . . . 27
  - 3.5.1 `remainingSegment` can be reused when calculating `storedNormalizedQuantity` . . . 27
  - 3.5.2 The `if` block when `numSegmentsFullyConsumed != 0` can be simplified . . . 28
- 3.6 Informational . . . 28
  - 3.6.1 Analysis of `BokkyPooBahsRedBlackTreeLibrary` . . . 28
  - 3.6.2 Changes made to `BokkyPooBahsRedBlackTreeLibrary` . . . 61
  - 3.6.3 Typos, comments, minor recommendations, . . . 62
  - 3.6.4 The too many orders hit check is incorrect . . . 64
  - 3.6.5 `_claimNFTs` assumes the `_nft` would revert on `0` token id transfers . . . 65
  - 3.6.6 The owner can indirectly change the `priceTick` of a `tokenId` . . . 66
  - 3.6.7 `_addToBookSide` can be simplified . . . 67
  - 3.6.8 `_allOrdersAtPriceSide` can be simplified . . . 71
  - 3.6.9 Invariants of Price Segments . . . 72

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Paintswap is an open NFT marketplace, NFT launchpad, VRF and financial NFT order book on Sonic. They are also the developers of the Web3 game Estfor Kingdom.

From Apr 29th to May 15th the Cantina team conducted a review of sonic-airdrop-contracts on commit hash a3016e5f. The team identified a total of **29** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	1	1	0
High Risk	6	6	0
Medium Risk	3	3	0
Low Risk	8	6	2
Gas Optimizations	2	2	0
Informational	9	7	2
Total	**29**	**25**	**4**

3 Findings

3.1 Critical Risk

3.1.1 `segment` is incorrectly updated with a normalised quantity

Severity: Critical Risk

Context: SonicOrderBook.sol#L694-L697

Description: When the `segment` needs to be updated with a new normalised quantity, the following expression is used:

```
segment = bytes32(
  (uint256(segment) & ~(uint256(0xffffff) << (slot * SEGMENT_SIZE + ORDER_ID_SIZE))) | // clear out the old
  ↪  value
    (newNormalizedQuantity << (slot * SEGMENT_SIZE + ORDER_ID_SIZE)) // plugin the new value
);
```

When clearing out the old value the constant `0xffffff` (`type(uint24).max`) is used which is incorrect since the normalised quantity of a slot occupies `4` bytes or `32` (`NORMALIZED_QUANTITY_SIZE`) bits (its type is `uint32`). And so upon the clearing out operation the last byte of the old normalised quantity is not cleared out and it gets `OR`ed with `newNormalizedQuantity`:

```
// storedNormalizedQuantity = AA_BB_CC_DD

AA_BB_CC_DD -->
AA_00_00_00 -->
AA_00_00_00 | newNormalizedQuantity
```

The final value would always be greater or equal to `AA_00_00_00 = storedNormalizedQuantity | 0xffffff`. Thus, a minimum amount of `AA_00_00_00` will be retained in the normalised quantity indefinitely here. This would actually allow a bad actor to drain the order book both from `NFT` tokens and also quote tokens. This is the strategy. Sell and Buy orders are usually such that the lowest sell price p_s is higher than the highest buy price p_b for the unmatched orders:

$$p_b < p_s$$

So a bad actor A can do the following:

1. Pick a price p such that $p_b < p < p_s$ and create a buy limit order with a quantity such that its normalised value has the upper byte non-zero for example `0xff_??_??_??` and transfer the required amount of quote tokens to the orderbook (only need to transfer once).
2. A then can create a sell limit order with for the same token id and the same price as in step 1. such that it would only eat into the order from step 1. , for example its quantity can be only off by 1 from the quantity provided in step 1. Since the order in the previous step is the only order that exists for that price, the sell limit order gets matched with the buy limit order but only partially and it would trigger the bug above, ie. the normalised quantity would still have at least `0xff_00_00_00` value left in it.
3. A can claim its `NFT` airdrop that it has provided in Step 2. since it has matched an order against another order made by the same account. Also it can claim the quote token for the above match. So the net amount of `NFT` airdrop token taken from A would be `0` but it would gain the equivalent amount of quote token based on the price picked in step 1.
4. Repeat step 2. and 3. till draining the whole orderbook. At the very end if some amounts are not exactly matching in terms of balances of the orderbook A can donate the difference before performing the last iteration some the amounts requested from claiming the tokens and the balance of the orderbook match.

Proof of Concept:

```
it("POC", async function () {
  const {orderBook, tokenId, alice, priceTick} = await loadFixture(deployContractsFixture);
  const price = ethers.parseEther("0.01");
  let quantity = ethers.parseEther("167772.16"); // 0xffffff + 1
  let limitOrders = [
    {
      side: OrderSide.Buy,
      tokenId,
      price,
      quantity
    }
  ];

  await orderBook.limitOrders(limitOrders, RETURN_WRAPPED_SONIC);
  let orders = await orderBook.allOrdersAtPrice(OrderSide.Buy, tokenId, price);
  console.log(orders);  // 167772.16e18
  quantity = ethers.parseEther("0.01");
  limitOrders = [
    {
      side: OrderSide.Sell,
      tokenId,
      price,
      quantity
    }
  ];
  await orderBook.limitOrders(limitOrders, RETURN_WRAPPED_SONIC);
  orders = await orderBook.allOrdersAtPrice(OrderSide.Buy, tokenId, price);
  console.log(orders); // 335544.31e18

});
```

Recommendation: Use the constant `0xff_ff_ff_ff` (`type(uint32).max`) to clear out the old value. Moreover it would be best to:

1. Create a library to work with segments and slots. This way the security of the library and the order book can be separated and verified independently.
2. Define a named constant for the value used for `0xff_ff_ff_ff`. `QUANTITY_MASK` is already defined and can be reused here.

```
library SegmentSlotLib {
  //...
  function updateNQ(
    bytes32 segment,
    uint256 slot,
    uint32 normalizedQuantity
  ) internal pure returns (bytes32 newSegment) {
    assembly ("memory-safe") {
      // we are assuming here that the addition and multiplication below
      // would not overflow since `NUM_SLOTS_PER_SEGMENT` is vert small
      let shiftAmount := add(ORDER_ID_SIZE, mul(slot, SEGMENT_SIZE))

      /*   ...   slot_j   ... = segment
       *   ... [ nq, oid] ... -->
       *   ... [  0, oid] ... -->
       *   ... [nq', oid] ...
       */
      newSegment := or(
        // clear out the old normalized quantity value
        and(
          segment,
          not(shl(shiftAmount, QUANTITY_MASK))
        ),
        // plug in the new value
        shl(shiftAmount, normalizedQuantity)
      )
    }
  }
}
```

Paintswap: Fixed in PR 27. Have made 2 libraries SegmentLibrary and SlotLibrary which detail with a lot of the bit manipulation and it makes the orderbook code more manageable and easier to understand. Have renamed some of the variable names which were not clear like `offsetIndex` is now `slotIndex`, `slot`

is now used to mean actual slot bytes now but it is used with `uint256` to prevent segment/slots being called with incorrect library.

Cantina Managed: Issue has been fixed, although the other issues regarding:

- Check for remaining segment.
- Stale tombstone.

are still present and care needs to be taken when the PR fixes for those and also this PR get merged into the same branch.

3.2 High Risk

3.2.1 Making limit orders may cost excessive funds

Severity: High Risk

Context: SonicOrderBook.sol#L543

Description: When making the limit order, if the order number at the current price exceeds the maximum order number , the protocol adjusts the price of the created limit order.

```
// Check if last segment is full
if (
  (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT >= _maxOrdersPerPrice &&
  lastSegmentFilled
) {
  // Loop until we find a suitable place to put this
  while (true) {
    unchecked {
      price = SafeCast.toUint64(uint128(int128(uint128(price)) + priceTick));
    }
```

For example, when a user makes a limit order on the Buy side, if the order at the current price is full, the price of the order will be lowered, which means that the user will provide a lower price to make the limit order. The problem here is that when calculating the token amount that the user needs to pay, the protocol does not take into account that the price of the order has been adjusted, and still uses the old price to calculate the user's cost, which causes the user to spend more tokens to make the limit order, and these excess tokens will be lost.

```
if (orders[i].side == OrderSide.Buy) {
  unchecked {
    quoteTokensToUs += cost + (uint256(orders[i].price) * quantityAddedToBook) / BASE_DECIMAL_DIVISOR;
  }
```

Recommendation: It is recommended to consider the return value of `_addToBookSide()` when calculating the cost of the user making the limit order.

Paintswap: Fixed in PR 19. Have done the recommendation of returning the new price.

Cantina Managed: Fix verified.

3.2.2 `cancelAndMakeLimitOrders()` may not return funds

Severity: High Risk

Context: SonicOrderBook.sol#L288-L290

Description: In `cancelAndMakeLimitOrders()`, it will first cancel orders and then create new limit orders:

```
function cancelAndMakeLimitOrders(
  uint48[] calldata orderIds,
  CancelOrder[] calldata orders,
  LimitOrder[] calldata newOrders,
  bool returnWrappedS
) external payable nonReentrant {
  address sender = _msgSender();

  (uint256 amountRefunded, uint256[] memory nftIdsFromUs, uint256[] memory nftAmountsFromUs) = _cancelOrders(
    sender,
    orderIds,
    orders
  );
  (uint256 fees, uint256 quoteTokensToUs, uint256 quoteTokensFromUs) = _limitOrders(sender, newOrders);

  if (nftIdsFromUs.length != 0) {
    _safeBatchTransferNFTsFromUs(sender, nftIdsFromUs, nftAmountsFromUs);
  }

  bool quoteTokenIsS = _quoteToken == _wS;
  uint256 value = msg.value;
  if (quoteTokenIsS) {
    value += amountRefunded;
  }

  _resolveQuoteTokens(sender, fees, quoteTokensToUs, quoteTokensFromUs, value, returnWrappedS);
}
```

The problem here is that the quoteToken returned for canceled orders is only returned if the quoteToken is wS, otherwise the funds are frozen in the contract.

Proof of Concept:

```
it("POC", async function () {
  const {orderBook, tokenId, owner, quoteToken} = await loadFixture(deployContractsFixture);

  // Set up order book
  const price = ethers.parseEther("8");
  const quantity = ethers.parseEther("100");
  const initBal = await quoteToken.balanceOf(owner);
  console.log(initBal);
  await orderBook.limitOrders(
    [
      {
        side: OrderSide.Buy,
        tokenId,
        price,
        quantity
      }
    ],
    RETURN_WRAPPED_SONIC
  );
  const bal1 = await quoteToken.balanceOf(owner);
  console.log(initBal - bal1); // 800e18
  const newOrder = {
    side: OrderSide.Buy,
    tokenId,
    price: price,
    quantity: quantity + ethers.parseEther("2")
  };
  await orderBook.cancelAndMakeLimitOrders(
    [1],
    [{side: OrderSide.Buy, tokenId, price}],
    [newOrder],
    RETURN_WRAPPED_SONIC
  );
  const bal2 = await quoteToken.balanceOf(owner);
  console.log(initBal - bal2); // 1616e18
});
```

Recommendation: It is recommended to subtract the `amountRefunded` from the `quoteTokensToUs` first and add any leftover to the `quoteTokensFromUs`.

```
  function cancelAndMakeLimitOrders(
    uint48[] calldata orderIds,
    CancelOrder[] calldata orders,
    LimitOrder[] calldata newOrders,
    bool returnWrappedS
  ) external payable nonReentrant {
    address sender = _msgSender();

    (uint256 amountRefunded, uint256[] memory nftIdsFromUs, uint256[] memory nftAmountsFromUs) = _cancelOrders(
      sender,
      orderIds,
      orders
    );
    (uint256 fees, uint256 quoteTokensToUs, uint256 quoteTokensFromUs) = _limitOrders(sender, newOrders);

    if (nftIdsFromUs.length != 0) {
      _safeBatchTransferNFTsFromUs(sender, nftIdsFromUs, nftAmountsFromUs);
    }
+   uint256 value = msg.value;
+   if (quoteTokensToUs > amountRefunded) {
+     quoteTokensToUs -= amountRefunded;
+   } else {
+     quoteTokensFromUs += amountRefunded - quoteTokensToUs;
+     quoteTokensToUs = 0;
+   }
-   bool quoteTokenIsS = _quoteToken == _wS;
-   if (quoteTokenIsS) {
-     value += amountRefunded;
-   }

    _resolveQuoteTokens(sender, fees, quoteTokensToUs, quoteTokensFromUs, value, returnWrappedS);
  }
```

Paintswap: Fixed in PR 20. Have done the recommendation, and renamed `amountRefunded` → `quoteTokensFromCancelling` to be clearer. Finding "`cancelAndMakeLimitOrders()` may fail due to insufficient balance of the user" had some comments about returning the quote tokens before limit orders. It was done this way to be more gas efficient for buy orders, although it does increase complexity, but will keep it how it is given it's already been reviewed in this finding and added a comment about what is and not returned after cancelling.

Cantina Managed: Fix verified.

3.2.3 Excess NFTs are gotten when adding order book fails

Severity: High Risk

Context: SonicOrderBook.sol#L549

Description: When matching orders in `_limitOrders()`, `failedQuantity` is not taken into account for the NFT amount returned to the user.

```
if (cost != 0) {
  // Transfer the NFTs taken from the order book straight to the taker
  nftIdsFromUs[nftLengthFromUs] = orders[i].tokenId;
  nftAmountsFromUs[nftLengthFromUs++] = orders[i].quantity - quantityAddedToBook;
}
```

Consider that a user intends to purchase 100 NFTs, 90 are traded, and the remaining 10 are less than minQuantity, so `quantityAddedToBook` is 0 and `failedQuantity` is 10. At this time, the user should be charged the value of 90 NFTs and 90 NFTs should be sent to the user. But since the `failedQuantity` is not subtracted from `orders.quantity` when calculating `nftAmountsFromUs`, this causes the protocol to return 100 NFTs to the user.

Proof of Concept: In the following test, Alice spends `56e18` `quoteTokens` to get `10e18` instead of `7e18` NFTs.

```
it("POC", async function () {
  const {orderBook, tokenId, alice, sonicAirdropNFT, priceTick, quoteToken} = await
  ↪  loadFixture(deployContractsFixture);
  const minQuantity = ethers.parseEther("5");
  const isTradeable = true;
  await orderBook.setTokenIdInfos([tokenId],[{priceTick, minQuantity, isTradeable}]);
```

```
  const price = ethers.parseEther("8");
  let quantity = ethers.parseEther("7");
  let limitOrders = [
    {
      side: OrderSide.Sell,
      tokenId,
      price : price + ethers.parseEther("1"),
      quantity
    },
    {
      side: OrderSide.Sell,
      tokenId,
      price,
      quantity
    }
  ];

  await orderBook.limitOrders(limitOrders, RETURN_WRAPPED_SONIC);

  quantity = ethers.parseEther("10");
  limitOrders = [
    {
      side: OrderSide.Buy,
      tokenId,
      price: price,
      quantity
    }
  ]
  let NftBalBefore = await sonicAirdropNFT.balanceOf(alice, tokenId);
  let CoinBalBefore = await quoteToken.balanceOf(alice);

  await orderBook.connect(alice).limitOrders(limitOrders, RETURN_WRAPPED_SONIC);
  let NftBalAfter = await sonicAirdropNFT.balanceOf(alice, tokenId);
  let CoinBalAfter = await quoteToken.balanceOf(alice);

  console.log(NftBalAfter-NftBalBefore); // 10e18
  console.log(CoinBalBefore-CoinBalAfter); // 56e18
});
```

Recommendation:

```
- nftAmountsFromUs[nftLengthFromUs++] = orders[i].quantity - quantityAddedToBook;
+ nftAmountsFromUs[nftLengthFromUs++] = orders[i].quantity - quantityAddedToBook - failedQuantity;
```

Paintswap: Fixed in PR 21.

Cantina Managed: Fix verified.

3.2.4 Users can convert NFT to `quoteToken` when claiming orders

Severity: High Risk

Context: SonicOrderBook.sol#L303-L316

Description: `_tokenClaimables` distinguishes between `quoteToken` and NFT based on `tokenId`.

```
_tokenClaimables[newOrderId] = ClaimableTokenInfo({
  maker: sender,
  tokenId: uint8(OrderSide.Buy == side ? tokenId : 0),
  amount: 0
});
```

The problem here is that the tokenId is not checked to be 0 when claiming tokens, resulting in users being able to convert NFTs to quoteTokens for claiming.

```
function _claimTokens(address sender, uint48[] calldata orderIds, bool returnWrappedS) private {
  require(orderIds.length <= MAX_CLAIMABLE_ORDERS, ClaimingTooManyOrders());

  uint256 amount;
  for (uint256 i = 0; i < orderIds.length; ++i) {
    ClaimableTokenInfo storage claimableTokenInfo = _tokenClaimables[orderIds[i]];
    uint256 claimableAmount = claimableTokenInfo.amount;
    require(claimableAmount != 0, NothingToClaim());
    require(claimableTokenInfo.maker == sender, NotMaker());

    claimableTokenInfo.amount = 0;
    amount += claimableAmount;
  }
```

In addition to breaking the asset composition in the order book, this is equal to unlocking the funds in the NFT in advance.

Proof of Concept: In the test the user converted the `10e18` NFT to `10e18` `quoteToken`.

```
it("POC", async function () {
  const {orderBook, owner, alice, tokenId} = await loadFixture(deployContractsFixture);

  // Set up order book
  const price = ethers.parseEther("8");
  const quantity = ethers.parseEther("100");
  await orderBook.limitOrders(
    [
      {
        side: OrderSide.Buy,
        tokenId,
        price,
        quantity
      }
    ],
    RETURN_WRAPPED_SONIC
  );
  await orderBook.connect(alice).limitOrders(
    [
      {
        side: OrderSide.Sell,
        tokenId,
        price,
        quantity: ethers.parseEther("10")
      }
    ],
    RETURN_WRAPPED_SONIC
  );

  const orderId = 1;
  expect(await orderBook.tokensClaimable([orderId])).to.eq(0);
  expect(await orderBook.tokensClaimable([orderId + 1])).to.eq(0);
  expect((await orderBook.nftsClaimable([orderId]))[0]).to.eq(ethers.parseEther("10"));
  expect((await orderBook.nftsClaimable([orderId + 1]))[0]).to.eq(0);

  // claim as the maker
  await expect(orderBook.claimTokens([orderId], false))
    .to.emit(orderBook, "ClaimedTokens")
    .withArgs(owner.address, [orderId], ethers.parseEther("10"));
});
```

Recommendation:

```
  for (uint256 i = 0; i < orderIds.length; ++i) {
    ClaimableTokenInfo storage claimableTokenInfo = _tokenClaimables[orderIds[i]];
    uint256 claimableAmount = claimableTokenInfo.amount;
    require(claimableAmount != 0, NothingToClaim());
+   require(claimableTokenInfo.tokenId == 0, NothingToClaim());
    require(claimableTokenInfo.maker == sender, NotMaker());
```

Paintswap: Fixed in PR 22. Have made the change. Something else which might be confusing is that `setTokenIdInfos()` is allowed to set info on token id 0. Given it has special conditions in the order book for the claimables it makes sense to prevent setting that, which I have done.

Cantina Managed: Fix verified.

3.2.5 Attacker can provide S to steal the equal amount of quoteToken

Severity: High Risk

Context: SonicOrderBook.sol#L1226-L1251

Description: In `_resolveQuoteTokens()`, it assumes `quoteToken` is `wS`. When `quoteToken` is not `wS`, attacker can provide `S` to steal the equal amount of `quoteTokens`.

```
function _resolveQuoteTokens(
  address sender,
  uint256 fees,
  uint256 quoteTokensToUs,
  uint256 quoteTokensFromUs,
  uint256 value,
  bool returnWrappedS
) private {
  if (value > quoteTokensToUs) {
    // Refund to the user
    unchecked {
      quoteTokensFromUs += value - quoteTokensToUs;
    }
    value = quoteTokensToUs;
  }

  if (quoteTokensToUs != 0) {
    _safeTransferCoinsToUs(sender, value, quoteTokensToUs);
  }

  if (quoteTokensFromUs != 0) {
    _safeTransferCoinsFromUs(sender, quoteTokensFromUs, returnWrappedS);
  }

  _sendFees(fees);
}
```

For example, when the user provides NFT to fulfill the order, the protocol will send `quoteToken` to the user. Consider the attacker calling `marketOrder()` with 500 `S`, in `_resolveQuoteTokens()`, `quoteTokensToUs` is `0`, `quoteTokensFromUs` is 1000, and `msg.value` is 500.

```
function marketOrder(MarketOrder calldata order, bool returnWrappedS) external payable nonReentrant {
  // ...
  } else {
    require(cost >= order.totalCost, TotalCostConditionNotMet());

    // Transfer tokens to the seller if any have sold
    quoteTokensFromUs = cost - fees;
    quoteTokensToUs = 0;
  }

  _resolveQuoteTokens(sender, fees, quoteTokensToUs, quoteTokensFromUs, msg.value, returnWrappedS);
```

Then in `_resolveQuoteTokens()`, `quoteTokensFromUs` will increase to 1500, i.e. the contract returns the quoteToken to the user as excess `S`.

```
function _resolveQuoteTokens(
  address sender,
  uint256 fees,
  uint256 quoteTokensToUs,
  uint256 quoteTokensFromUs,
  uint256 value,
  bool returnWrappedS
) private {
  if (value > quoteTokensToUs) {
    // Refund to the user
    unchecked {
      quoteTokensFromUs += value - quoteTokensToUs;
    }
    value = quoteTokensToUs;
  }

  if (quoteTokensToUs != 0) {
    _safeTransferCoinsToUs(sender, value, quoteTokensToUs);
  }

  if (quoteTokensFromUs != 0) {
    _safeTransferCoinsFromUs(sender, quoteTokensFromUs, returnWrappedS);
  }

  _sendFees(fees);
}
```

Considering that `quoteToken` might be USDC and `wS` is priced at about 0.5 USDC, it is profitable for the attacker to convert `S` to USDC 1:1.

Proof of Concept:

```
it("POC", async function () {
  const {orderBook, alice, tokenId, quoteToken} = await loadFixture(deployContractsFixture);

  // Set up order book
  const price = ethers.parseEther("8");
  const quantity = ethers.parseEther("100");
  await orderBook.limitOrders(
    [
      {
        side: OrderSide.Buy,
        tokenId,
        price,
        quantity
      }
    ],
    RETURN_WRAPPED_SONIC
  );
  let CoinBalBefore = await quoteToken.balanceOf(alice);
  await orderBook.connect(alice).marketOrder(
    {
      side: OrderSide.Sell,
      tokenId,
      totalCost: 0, // total totalCost is too high
      quantity: ethers.parseEther("10")
    },
    RETURN_WRAPPED_SONIC,
    {value: ethers.parseEther("700")}
  );
  let CoinBalAfter = await quoteToken.balanceOf(alice);
  console.log(CoinBalAfter-CoinBalBefore); // 779.76e18
});
```

Recommendation:

```
   function _resolveQuoteTokens(
     address sender,
     uint256 fees,
     uint256 quoteTokensToUs,
     uint256 quoteTokensFromUs,
     uint256 value,
     bool returnWrappedS
   ) private {
-    if (value > quoteTokensToUs) {
+    if (_quoteToken == _wS && value > quoteTokensToUs) {
       // Refund to the user
       unchecked {
         quoteTokensFromUs += value - quoteTokensToUs;
       }
       value = quoteTokensToUs;
-    }
+    } else if (_quoteToken != _wS){
+     require(value == 0);
+    }

     if (quoteTokensToUs != 0) {
       _safeTransferCoinsToUs(sender, value, quoteTokensToUs);
     }

     if (quoteTokensFromUs != 0) {
       _safeTransferCoinsFromUs(sender, quoteTokensFromUs, returnWrappedS);
     }

     _sendFees(fees);
   }
```

Paintswap: Fixed in PR 25.

Cantina Managed: Fix verified.

3.2.6 Stale value of `tombstoneOffset` is used when the `price` is updated in `_addToBookSide`

Severity: High Risk

Context: SonicOrderBook.sol#L972

Description: When the current `price` level has reached its capacity, we enter the `while` loop where the `price` is incremented in each iteration by `priceTick`. If this new `price` already exists in the `tree` we would want to check whether this new `price` level has reached its capacity or not. If it has not reached its capacity, it would be a suitable price level where we can `break` out of the loop. But there is a catch, the check for the capacity at this new `price` level uses the old stale `tombstoneOffset` value from the user provided original price which might not be equal to the `tombstoneOffset` value of the current new `price`.

Recommendation: To avoid mistakes like above, it would be best to abstract away this and other utility operations into utility functions. For example for this case we can define:

```
/**
 * @dev Returns the used capacity of a specific price level in the market.
 * The used capacity is determined by the number of segments at that price level,
 * minus the number of tombstones (inactive slots), and then multiplied by the number of slots per segment.
 *
 * @param tree The Red-Black Tree containing the prices and their associated nodes.
 * @param segmentsPriceMap A mapping from each price to its segments, where each segment is an array of bytes32
 ↪  representing the slots at that price level.
 * @param price The price level for which to retrieve the used capacity.
 *
 * @return uint256 The used capacity of the specified price level in the market.
 */
function _getPriceLevelUsedCapacity(
  BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
  mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
  uint64 price
) internal view returns (uint256) {
  // below already checks whether the price exists in the tree
  uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;
  return (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT;
}
```

Then the following patch can be applied:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index b3bb01a..cbda90b 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -947,7 +947,6 @@ contract SonicOrderBook is
       if (!tree.exists(price)) {
         tree.insert(price);
       } else {
-        uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;
         // Check if this would go over the max number of orders allowed at this price level
         bool lastSegmentFilled = (
           (uint256(
  @@ -957,7 +956,7 @@ contract SonicOrderBook is

         // Check if last segment is full
         if (
-          (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT >= _maxOrdersPerPrice &&
+          _getPriceLevelUsedCapacity(tree, segmentsPriceMap, price) >= _maxOrdersPerPrice &&
           lastSegmentFilled
         ) {
           // Loop until we find a suitable place to put this
  @@ -969,7 +968,7 @@ contract SonicOrderBook is
               tree.insert(price);
               break;
             } else if (
-              (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT < _maxOrdersPerPrice
↪  ||
+              _getPriceLevelUsedCapacity(tree, segmentsPriceMap, price) < _maxOrdersPerPrice ||
               (
                 (uint256(
                   segmentsPriceMap[price][segmentsPriceMap[price].length - 1] >>
```

The value computed by `_getPriceLevelUsedCapacity` and also the current implementation is not exact as the segment pointed by `tombstoneOffset` of set of a `price` might have some canceled or matched orders, so the actual value is possibly up to `2` units less than this number. One can calculate the exact number but it would require more gas. Moreover since this was not picked up by test cases. New regression test cases need to be added to cover the issue mentioned in this finding.

Paintswap: Fixed in PR 30. Have done the recommendation to make a `_getPriceLevelUsedCapacity` utility function.

Cantina Managed: Fix verified.

3.3 Medium Risk

3.3.1 A new limit order might get added before an older limit order

Severity: Medium Risk

Context: SonicOrderBook.sol#L994-L1005

Description: Let's consider this case that for a `side`, `tokenId` and a `price` the corresponding segments look like below:

```
    slots                       segment indices
--------<--------------------+-----------------
// ... [Q, O] [Q, O] [Q, O] ...   0
// ... [Q, O] [Q, O] [Q, O] ...   1
// ...
// ... slot_k slot_m slot_n ...   segments.length - 1 == tombstoneOffset
```

It is possible that the order corresponding to `slot_n` gets matched fully and thus one ends up with:

```
    slots                       segment indices
--------<--------------------+-----------------
// ... [Q, O] [Q, O] [Q, O] ...   0
// ... [Q, O] [Q, O] [Q, O] ...   1
// ...
// ... slot_k slot_m [0, O] ...   segments.length - 1 == tombstoneOffset
```

and note that the order for `slot_m` was added before the order for `slot_k` and the next orders should be added after these slots and not before since orders are supposed to be matched in a `FIFO` fashion. But here in this `remainingSegment` context gets calculated as the `slot_n` which is `0` and thus the new upcoming order gets inserted before `slot_k` and `slot_m`. The mistake here is that as the name suggests `remainingSegment` needs to represent the remaining of the last segment and not just a slot portion of it (note that masking with `& SEGMENT_MASK` is used).

Proof of Concept: Apply the following test patch where one can show case a new limit order can actually be placed in the inactive tombstoned area of the price segments:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index b3bb01a..6fef482 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -479,6 +479,14 @@ contract SonicOrderBook is
       }
     }

+    function getSegments(OrderSide side, uint256 tokenId, uint64 price) external view returns (bytes32[]
↪  memory) {
+      if (side == OrderSide.Buy) {
+        return _bidsAtPrice[tokenId][price];
+      } else {
+        return _asksAtPrice[tokenId][price];
+      }
+    }
+
     /// @notice Get all orders at a specific price level
     /// @param side The side of the order book to get orders from
     /// @param tokenId The token ID to get orders for
  diff --git a/test/sonicOrderBook/SonicOrderBook.ts b/test/sonicOrderBook/SonicOrderBook.ts
  index b7aaf51..be252f1 100644
-  -- a/test/sonicOrderBook/SonicOrderBook.ts
+  ++ b/test/sonicOrderBook/SonicOrderBook.ts
  @@ -4472,6 +4472,94 @@ describe("SonicOrderBook", function () {
       });
     });

+    it("PoC - order placed in the inactive tombstoned area", async function () {
+      const {orderBook, owner, tokenId, quoteToken, sonicAirdropNFT, priceTick, initialCoins} =
+        await loadFixture(deployContractsFixture);
+
+      await quoteToken.mint(owner, initialCoins * 30n);
+      await quoteToken.approve(orderBook, initialCoins * 30n);
+
+      const price = ethers.parseEther("1");
+      const quantity = ethers.parseEther("10");
+
+      await sonicAirdropNFT.mintSpecificId(tokenId, quantity * 40n);
+
+      async function logSegments(side: OrderSide, tokenId: number, price: bigint) {
+        let segments = await orderBook.getSegments(side, tokenId, price);
+        console.log(segments);
+      }
+
+      await logSegments(OrderSide.Sell, tokenId, price);
+
+      await orderBook.limitOrders(
+        [
+          {
+            side: OrderSide.Sell,
+            tokenId,
+            price: price,
+            quantity
+          },
+          {
+            side: OrderSide.Sell,
+            tokenId,
+            price: price,
+            quantity
+          },
+          {
+            side: OrderSide.Sell,
+            tokenId,
+            price: price,
```

```
+            quantity
+          }
+        ],
+        RETURN_WRAPPED_SONIC
+      );
+
+      await logSegments(OrderSide.Sell, tokenId, price);
+
+      await orderBook.limitOrders(
+        [
+          {
+            side: OrderSide.Buy,
+            tokenId,
+            price: price,
+            quantity: quantity
+          }
+        ],
+        RETURN_WRAPPED_SONIC
+      );
+
+      await logSegments(OrderSide.Sell, tokenId, price);
+
+      await orderBook.limitOrders(
+        [
+          {
+            side: OrderSide.Buy,
+            tokenId,
+            price: price,
+            quantity: quantity * 2n
+          }
+        ],
+        RETURN_WRAPPED_SONIC
+      );
+
+      await logSegments(OrderSide.Sell, tokenId, price);
+
+      await orderBook.limitOrders(
+        [
+          {
+            side: OrderSide.Sell,
+            tokenId,
+            price: price,
+            quantity
+          },
+        ],
+        RETURN_WRAPPED_SONIC
+      );
+
+      await logSegments(OrderSide.Sell, tokenId, price);
+    });
+
     it("System test (many orders)", async function () {
       const {orderBook, owner, tokenId, quoteToken, sonicAirdropNFT, priceTick, initialCoins} =
         await loadFixture(deployContractsFixture);
```

and run:

```
yarn test -- --grep "PoC"
```

```
Result(0) []
Result(1) [
  '0x0000000003e80000000000003000003e80000000000002000003e8000000000001'
]
Result(1) [
  '0x0000000003e80000000000003000003e8000000000002000000000000000000000'
]
Result(1) [
  '0x0000000003e80000000000003000003e8000000000002000000000000000000000'
]
Result(1) [
  '0x0000000003e80000000000003000003e80000000000002000003e8000000000004'
]
```

As you can see from the logs, order `4` is inserted in the inactive tombstoned area.

Recommendation: Make sure `remainingSegment` is calculated as:

```
uint256 remainingSegment = uint256(lastSegment) >> (slot * SEGMENT_SIZE)
```

and thus the check `remainingSegment == 0` with this new calculated value makes sure that `... slot_k slot_m slot_n == 0` which means that there are no other orders in this last segment from `slot_n` onward.

Paintswap: Fixed in PR 31.

Cantina Managed: Fix verified.

3.3.2 `cancelAndMakeLimitOrders()` may fail due to insufficient balance of the user

Severity: Medium Risk

Context: SonicOrderBook.sol#L275-L284

Description: In `cancelAndMakeLimitOrders()`, it will transfer the NFTs in the canceled order to the user after `_limitOrders()`.

```
(uint256 amountRefunded, uint256[] memory nftIdsFromUs, uint256[] memory nftAmountsFromUs) = _cancelOrders(
  sender,
  orderIds,
  orders
);
(uint256 fees, uint256 quoteTokensToUs, uint256 quoteTokensFromUs) = _limitOrders(sender, newOrders);

if (nftIdsFromUs.length != 0) {
  _safeBatchTransferNFTsFromUs(sender, nftIdsFromUs, nftAmountsFromUs);
}
```

Since at the end of `_limitOrders()`, NFTs may be transferred from User to Us, this may cause the transfer of NFTs in `_limitOrders()` in `cancelAndMakeLimitOrders()` to fail due to insufficient NFT balance of the user.

```
if (nftLengthToUs != 0) {
  assembly ("memory-safe") {
    mstore(nftIdsToUs, nftLengthToUs)
    mstore(nftAmountsToUs, nftLengthToUs)
  }
  _safeBatchTransferNFTsToUs(sender, nftIdsToUs, nftAmountsToUs);
}
```

Proof of Concept:

```
  it("POC", async function () {
    const {orderBook, tokenId, priceTick, owner} = await loadFixture(deployContractsFixture);

    // initialQuantity 120e18
    const price = ethers.parseEther("8");
    const quantity = ethers.parseEther("60");

    // make 60e18 limit order
    await orderBook.limitOrders(
      [
        {
          side: OrderSide.Sell,
          tokenId,
          price,
          quantity
        }
      ],
      RETURN_WRAPPED_SONIC
    );

    const newOrder = {
      side: OrderSide.Sell,
      tokenId,
      price: price - priceTick,
      quantity: quantity + ethers.parseEther("2")
    };

    // cancel and make new 62e18 limit order, failed due to ERC1155InsufficientBalance
    await orderBook.cancelAndMakeLimitOrders(
      [1],
      [{side: OrderSide.Sell, tokenId, price}],
      [newOrder],
      RETURN_WRAPPED_SONIC
    );
  });
});
```

Recommendation: It is recommended to move the `_limitOrders()` call to after the NFT transfer:

```
  (uint256 amountRefunded, uint256[] memory nftIdsFromUs, uint256[] memory nftAmountsFromUs) = _cancelOrders(
    sender,
    orderIds,
    orders
  );
- (uint256 fees, uint256 quoteTokensToUs, uint256 quoteTokensFromUs) = _limitOrders(sender, newOrders);

  if (nftIdsFromUs.length != 0) {
    _safeBatchTransferNFTsFromUs(sender, nftIdsFromUs, nftAmountsFromUs);
  }

+ (uint256 fees, uint256 quoteTokensToUs, uint256 quoteTokensFromUs) = _limitOrders(sender, newOrders);
```

Paintswap: Fixed in PR 32.

Cantina Managed: The fix in the PR 32 does not cover the below edge case:

Another point to notice here is that the `sender` might have also counted on receiving `amountRefunded` from canceling its orders before making the call to `_limitOrders`. There is a slight difference if the `amountRefunded` was transferred first before calling `_limitOrders`, it could in some cases be possible for the `sender` to perform some on-chain actions (for example making some `NFT` transfers off of the current orderbook, either using a different order book or just simple transfers to change its configuration of the `NFT` tokens it holds). If `sender` relied on this assumption its call to `cancelAndMakeLimitOrders` could potentially fail due to its assumption that.

```
cancelAndMakeLimitOrders =
1. cancelOrders
2. limitOrders
```

which is not entirely true based on above (the edge cases above includes cross-orderbook reentrancy) after after the above fix. So to keep this equivalence `amountRefunded` should be transferred out first then call `_limitOrders`.

3.3.3 Rounding down when calculating fees will result in users failing to claim tokens

Severity: Medium Risk

Context: SonicOrderBook.sol#L709-L710

Description: For USDC with 6 decimals, `priceTick` is 0.0001e6, i.e. 100, `QUANTITY_TICK` is 1e16, and the minimum cost must be a multiple of 100*10e16/10e18 = 1, so no rounding happens when calculating the cost.

```
const priceTick = ethers.parseUnits("0.0001", decimals);
```

However, when calculate `fees`, it would be tradeCost * 30 / 10000. So for the fee, rounding is highly likely to happen.

```
} else {
  uint256 fees = _calcFees(tradeCost);
  claimableTokenInfo.amount += uint88(tradeCost - fees);
}
```

Making it serious is in `_takeFromOrderBookSide()`, the protocol calculates the fee on each order, but collects the fee on all cost. Since the calculation of the fee is rounded multiple times, the collection of the fee is only rounded once, which will cause the collected fee to be greater than the calculated fee, so the user will fail to collect the token due to insufficient balance.

```
  } else {
    uint256 fees = _calcFees(tradeCost);
    claimableTokenInfo.amount += uint88(tradeCost - fees);
  }
// ...
function marketOrder(MarketOrder calldata order, bool returnWrappedS) external payable nonReentrant {
  // Must fufill the order and be below the total cost (or above depending on the side)
  uint256 quoteTokensToUs;
  uint256 quoteTokensFromUs;
  address sender = _msgSender();

  uint48[] memory orderIdsPool = new uint48[](MAX_ORDERS_HIT);
  uint256[] memory quantitiesPool = new uint256[](MAX_ORDERS_HIT);

  uint256 cost = _makeMarketOrder(sender, order, orderIdsPool, quantitiesPool);
  bool isBuy = order.side == OrderSide.Buy;
  uint256 fees = _calcFees(cost);
```

Proof of Concept:

```
it("POC", async function () {
  const {orderBook, owner, alice, dev, tokenId, quoteToken, ONE_QUOTE_TOKEN, priceTick} =
    await loadFixture(deployContractsFixture);
  const sellOrdesCount = ethers.parseUnits("5",0);
  const price = ONE_QUOTE_TOKEN + priceTick; // 1.0001 USDC
  const quantity = ethers.parseEther("1") + ethers.parseEther("0.01"); // 1.01e18
  const totalCost = (price * quantity * sellOrdesCount) / ethers.parseEther("1");
  await quoteToken.mint(alice, totalCost);
  await quoteToken.connect(alice).approve(orderBook, totalCost);

  await orderBook.connect(owner).limitOrders(
    [
      {
        side: OrderSide.Sell,
        tokenId,
        price,
        quantity
      },{
        side: OrderSide.Sell,
        tokenId,
        price,
        quantity
      },{
        side: OrderSide.Sell,
        tokenId,
        price,
        quantity
      },{
        side: OrderSide.Sell,
```

```
        tokenId,
        price,
        quantity
      },{
        side: OrderSide.Sell,
        tokenId,
        price,
        quantity
      }
    ],
    DONT_RETURN_WRAPPED_SONIC
  );
  await orderBook.connect(alice).limitOrders(
    [
      {
        side: OrderSide.Buy,
        tokenId,
        price,
        quantity : quantity * sellOrdesCount
      }
    ],
    DONT_RETURN_WRAPPED_SONIC
  );
  console.log(totalCost); // 5050505
  const tc = await orderBook.tokensClaimable([1,2,3,4,5]);
  console.log(tc);  // 5035355
  const bal = await quoteToken.balanceOf(orderBook);
  console.log(bal); // 5035354
  await orderBook.connect(owner).claimTokens([1,2,3,4,5], DONT_RETURN_WRAPPED_SONIC); // reverted with custom
  ↪  error 'ERC20InsufficientBalance
});
```

Recommendation: One option is to round up when calculating fees. Another option is to make the collected and calculated fees consistent, always on each order.

Paintswap: Fixed in PR 38. We are now accumulating all the fees individually rather than from the total cost.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 No boundary checks for `instantClaimBps`

Severity: Low Risk

Context: SonicAirdropNFT.sol#L325

Description: When the owner of `SonicAirdropNFT` calls `addSeason(...)` no boundary checks are performed for `instantClaimBps`. If the provided value is greater than `SCALING_FACTOR` the calculation for `locked` amount would underflow in an `unchecked` block:

```
uint128 instant;
uint128 locked;
unchecked {
  instant = (amount * theSeason.instantClaimBps) / SonicMetadata.SCALING_FACTOR;
  locked = amount - instant; // <<< potential underflow
}
```

Recommendation: Although it could be highly unlikely that the owner would provide out of range value for `instantClaimBps` just like the other checks for the other parameters, it would be best to add a check for `instantClaimBps` in `addSeason(...)` to make sure it could not be greater than `SonicMetadata.SCALING_FACTOR`. Moreover one can even add a check against a contract-level constant `MAX_INSTANT_CLAIM_BPS` which itself is less than `SonicMetadata.SCALING_FACTOR`:

```
instantClaimBps < MAX_INSTANT_CLAIM_BPS < SonicMetadata.SCALING_FACTOR
```

Paintswap: Fixed in PR 18.

- Have added `maxInstantClaimAvailableBps` param to the SonicAirdropNFT constructor and checking that is below the scaling factory and now checking that `instantClaimBps` is below or equal to it.

- Have moved `SCALING_FACTOR` to a shared file (and renamed to `NFT_SCALING_FACTOR`) as referencing it from a metadata library didn't make sense when it's used for actual logic too.
- Fixed up some incorrect use of "percent" to "basis points" in comments too.

Cantina Managed: Fix verified.

3.4.2 `next` and `prev` can be called on a removed tree node

Severity: Low Risk

Context: BokkyPooBahsRedBlackTreeLibrary.sol#L54-L78

Description: With the changes made to the `BokkyPooBahsRedBlackTreeLibrary` the removed nodes retain their:

- `left`.
- `right`.
- `red`.
- `tombstoneOffset`.

values and thus if `next` and `prev` since existence checks are missing from these functions they might return non-EMPTY nodes when they are called with some removed nodes.

Recommendation: Currently these functions are not used in the codebase (order book), but to avoid potential future mistakes it might be a good to either:

1. Add existence checks to `next` and `prev` or...
2. Upon removal of a node also clear its `left` and `right` childs. (Note the children of a removed node might still exist in the tree).

Paintswap: Fixed in PR 23. I have removed `next` and `prev` functions. I don't think nodes which are removed are used anywhere and then when they get re-added to the tree all the properties are reset, so to save gas will just leave them.

Cantina Managed:

1. Has been applied by removing those functions.
2. Has been acknowledged.

3.4.3 Parent of the `EMPTY` node might not be `EMPTY`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When removing a node `n` from a tree T there are some edges cases where the `probe` ends up being the `EMPTY` node connected to a non-EMPTY node P. In these cases in the remove flow the call to `removeFixup` is made with the parameters `removeFixup(T, EMPTY)`. This is needed to fix up invariant 4. of the red black tree which is broken. After the call `removeFixup` ends, the parent of the `EMPTY` node is never cleared up.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.29;

import {Test} from "forge-std/Test.sol";
import {BokkyPooBahsRedBlackTreeLibrary as RBT} from
↪  "../contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol";

/**
 * In the test diagrams the nodes N with `o` N(o) represent
 * black nodes and the ones with `x`, N(x) represent red nodes
 */
contract BokkyPooBahsRedBlackTreeLibraryTest is Test {
    using RBT for RBT.Tree;
```

```
RBT.Tree public tree;
uint64 private constant EMPTY = 0;
function setUp() public {

}

/**
 *        1(o)
 *          \
 *          2(x) <--- remove
 */
function testRemove1() public {
    tree.root = 1;
    tree.nodes[1] = RBT.Node({
        parent: EMPTY,
        left: EMPTY,
        right: 2,
        red: false,
        tombstoneOffset: 0
    });
    tree.nodes[2] = RBT.Node({
        parent: 1,
        left: EMPTY,
        right: EMPTY,
        red: true,
        tombstoneOffset: 0
    });

    tree.remove(2);

    _assertSameBlackPathNumber(tree.root);
    _assertEMPTYNodeIsIsolatedAndBlack();
    _assertNoRedNodesAreConnected();
}

/**
 *        1(o)
 *          \
 *          2(x) <--- remove
 *
 * Generally this corresponds to the following situation
 *          |
 *          p
 *          |
 *          n <--- remove
 *         /  \
 *
 *
 * There is another general case that is not tested here:
 *
 *         p
 *         |
 *         k <--- remove
 *        / \
 *       L0  R0
 *           /
 *         L1
 *         /
 *       ...
 *       /
 *      L_{n-1} <-- p
 *     /
 *    Ln <-- c
 *   / \
 *      <-- probe
 */
function testRemove() public {
    tree.insert(1);
    tree.insert(2);
    tree.remove(2);

    _assertSameBlackPathNumber(tree.root);
    _assertEMPTYNodeIsIsolatedAndBlack();
    _assertNoRedNodesAreConnected();
}
```

```
/**
 *
 *            |
 *           (o)
 *           / \
 *
 */
function _assertEMPTYNodeIsIsolatedAndBlack() internal {
    // EMPTY should have EMPTY parent at all times
    // otherwise one needs to make sure, the parent of an EMPTY
    // Node is not accessed by mistake.

    _assertEMPTYNodeIsNotRed();
    _assertEMPTYNodeHasEMPTYLeftChild();
    _assertEMPTYNodeHasEMPTYRightChild();
    _assertEMPTYNodeHasEMPTYParent();
    _assertEMPTYNodeHashZeroTombstoneOffset();
}

function _assertEMPTYNodeIsNotRed() internal {
    vm.assertEq(tree.nodes[EMPTY].red, false, "EMPTY node is RED");
}

function _assertRootIsNoteRed() internal {
    vm.assertEq(tree.nodes[tree.root].red, false, "ROOT node is RED");
}

function _assertEMPTYNodeHasEMPTYParent() internal {
    vm.assertEq(tree.nodes[EMPTY].parent, EMPTY, "EMPTY node has a non-EMPTY parent");
}

function _assertEMPTYNodeHasEMPTYLeftChild() internal {
    vm.assertEq(tree.nodes[EMPTY].left, EMPTY, "EMPTY node has a non-EMPTY left child");
}

function _assertEMPTYNodeHasEMPTYRightChild() internal {
    vm.assertEq(tree.nodes[EMPTY].right, EMPTY, "EMPTY node has a non-EMPTY right child");
}

function _assertEMPTYNodeHashZeroTombstoneOffset() internal {
    vm.assertEq(tree.nodes[EMPTY].tombstoneOffset, 0, "EMPTY node has a non-ZERO `tombstoneOffset`");
}

function _assertNoRedNodesAreConnected() internal {
    uint64 root = tree.root;
    RBT.Node storage rootNode = tree.nodes[root];
    // the root node should always be BLACK
    vm.assertEq(rootNode.red, false, "root node is RED");

    _assertNoRedNodesAreConnectedInTheSubTree(root);
}
function _assertNoRedNodesAreConnectedInTheSubTree(uint64 key) internal {
    if (key == EMPTY) {
        _assertEMPTYNodeIsIsolatedAndBlack();
        return;
    }

    RBT.Node storage currNode = tree.nodes[key];
    uint64 left = currNode.left;
    uint64 right = currNode.right;
    RBT.Node storage leftChild = tree.nodes[left];
    RBT.Node storage rightChild = tree.nodes[right];

    if (currNode.red) {
        vm.assertEq(leftChild.red, false, "Two RED nodes are connected");
        vm.assertEq(rightChild.red, false, "Two RED nodes are connected");
    }

    _assertNoRedNodesAreConnectedInTheSubTree(left);
    _assertNoRedNodesAreConnectedInTheSubTree(right);
}

function _assertSameBlackPathNumber(uint64 key) internal returns (uint256 num) {
    if (key == EMPTY) {
        return 1;
    }
```

```
        RBT.Node storage currNode = tree.nodes[key];
        uint256 leftPathValue = _assertSameBlackPathNumber(currNode.left);
        uint256 rightPathValue = _assertSameBlackPathNumber(currNode.right);

        vm.assertEq(leftPathValue, rightPathValue, "Number of BLACK nodes from the current node to its leaf
        ↪  nodes do NOT match.");

        return currNode.red ? leftPathValue : leftPathValue + 1;
    }
}
```

Recommendation: Although when one inspects the library and considers all the edges cases, one can see that the `parent` of the `EMPTY` node is only queried in `removeFixup`'s first iteration of the `while` loop :

```
function removeFixup(Tree storage self, uint64 key) private {
  uint64 cursor;
  while (key != self.root && !self.nodes[key].red) {
    uint64 keyParent = self.nodes[key].parent; // <--- key could be `EMPTY`
```

It might be best to clear this value for the `EMPTY` node:

```
  diff --git a/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
  ↪  b/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
  index 07bb4bc..783e369 100644
-  -- a/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
+  ++ b/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
  @@ -172,6 +172,9 @@ library BokkyPooBahsRedBlackTreeLibrary {
       }
       // Don't delete the node, so that we can re-use the tombstone offset if re-adding this price
       self.nodes[cursor].parent = EMPTY;
+
+      // reset the EMPTY node's parent. Might be cheaper to `delete` vs reseting the parent field
+      delete self.nodes[EMPTY];
     }

     function treeMinimum(Tree storage self, uint64 key) private view returns (uint64) {
```

Paintswap: Acknowledged.

Cantina Managed: Acknowledged.

3.4.4 Off-by-one issue in `SonicAirdropNFT.balanceOf()`

Severity: Low Risk

Context: SonicAirdropNFT.sol#L371-L373

Description: When `_now() == lockedBurnTime`, in `tisTheSeason()`, it is valid for unlocking:

```
} else if (what == For.Unlocking) {
  uint40 startTime = theSeason.startTime;
  require(startTime <= _now(), SeasonClaimNotStarted(season, startTime));
  uint40 lockedBurnTime = theSeason.lockedBurnTime;
  require(lockedBurnTime >= _now(), SeasonUnlocksEnded(season, lockedBurnTime));
```

But the user cannot unlock because balanceOf is 0 now.

```
if (_seasons[Season(seasonId)].lockedBurnTime <= _now()) {
  return 0;
}
```

Recommendation: It is recommended to change to:

```
- if (_seasons[Season(seasonId)].lockedBurnTime <= _now()) {
+ if (_seasons[Season(seasonId)].lockedBurnTime < _now()) {
    return 0;
  }
```

Paintswap: Fixed in PR 33.

Cantina Managed: Fix verified.

3.4.5 An existing airdrop season with `startTime == 0` can be re-added

Severity: Low Risk

Context: SonicAirdropNFT.sol#L302-L313, SonicOrderBook.sol#L384, SonicOrderBook.sol#L1284-L1287

Description: If the owner of `SonicAirdropNFT` calls `addSeason` with `startTime` equal to `0` and it can call `addSeason` again to modify the `_seasons[season]` storage for the `season` in question. This can potentially affect all the calculations regarding token accountings. Some other minor issue are that for an added `season` with `startTime == 0`,

1. Calls to `_getURI` fails due to the following check:

```
require(season != Season.NONE && theSeason.startTime != 0, SeasonInvalid(season));
```

2. When adding token ids infos on an order book the following check will be skipped:

```
SeasonData memory seasonData = _nft.getSeasonData(Season(tokenIds[i]));
if (seasonData.startTime != 0 && tokenIdInfos[i].isTradeable) {
  // If this season exists make sure it hasn't ended yet if setting tradeable to true
  require(seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME > block.timestamp,
  ↪  SeasonLocked());
}
```

3. `_tokenIdInfos[tokenId].isTradeable` cannot be turned off by calling `refreshIsTradeable` due to the following check:

```
require(seasonData.startTime != 0, SeasonDoesntExist(tokenId));
```

Proof of Concept: Add the following test patch:

```
diff --git a/test/sonicNFT/SonicAirdropNFT.ts b/test/sonicNFT/SonicAirdropNFT.ts
index 7d9eb49..701d697 100644
-- a/test/sonicNFT/SonicAirdropNFT.ts
++ b/test/sonicNFT/SonicAirdropNFT.ts
@@ -230,6 +230,41 @@ describe("Sonic Airdrop NFT", () => {
          .withArgs(season);
      });

+     it("PoC - an existing season with startTime == 0 can be re-added", async () => {
+       const {sonicAirdropNFT, user1, user2} = await loadFixture(fixture);
+
+       async function addSeason3(user: SignerWithAddress, sonicAirdropNFT: SonicAirdropNFT, timeBuffer:
↪  number) {
+         const now = ((await ethers.provider.getBlock("latest")) as Block).timestamp;
+         const futureTime = now + timeBuffer;
+         const airdropAmount = ethers.parseEther("1000");
+
+         // Create merkle tree for future season
+         const claimData = [[user.address, airdropAmount.toString()]];
+         const tree = StandardMerkleTree.of(claimData, ["address", "uint128"]);
+
+         // Add future season
+         await sonicAirdropNFT.addSeason(
+           3, // <-- season 3
+           airdropAmount,
+           0, // <-- startTime = 0
+           futureTime + 50000,
+           futureTime + 100000,
+           futureTime + 150000,
+           2500,
+           tree.root,
+           {value: airdropAmount}
+         );
+       }
+
+       await expect(
+         addSeason3(user1, sonicAirdropNFT, 100000)
+       ).to.be.not.reverted;
+
+       await expect(
+         addSeason3(user2, sonicAirdropNFT, 200000)
+       ).to.be.not.reverted;
+     });
```

```
+
       it("should revert with InvalidAirdropAmount when value doesn't match", async () => {
         const {sonicAirdropNFT} = await loadFixture(fixture);
         const now = ((await ethers.provider.getBlock("latest")) as Block).timestamp;
```

and run

```
yarn test -- grep "PoC"
```

Recommendation: When `addSeason` is called make sure `startTime` is non-zero:

```
  diff --git a/contracts/SonicAirdropNFT.sol b/contracts/SonicAirdropNFT.sol
  index 8b84e4f..7bf1a3f 100644
-  -- a/contracts/SonicAirdropNFT.sol
+  ++ b/contracts/SonicAirdropNFT.sol
  @@ -49,6 +49,7 @@ contract SonicAirdropNFT is ISonicAirdropNFT, UUPSUpgradeable, Ownable2StepUpgra
     // Errors
     error SeasonInvalid(Season season);
     error SeasonExists(Season season);
+    error StartTimeCannotBeZero();
     error InvalidAirdropAmount();
     error InvalidClaimTime();
     error InvalidMatureTime();
  @@ -310,6 +311,7 @@ contract SonicAirdropNFT is ISonicAirdropNFT, UUPSUpgradeable, Ownable2StepUpgra
       bytes32 merkleRoot
     ) external payable override onlyOwner {
       _checkSeasonIsNotNone(season);
+      require(startTime != 0, StartTimeCannotBeZero());
       require(_seasons[season].startTime == 0, SeasonExists(season));
       require(airdropAmount != 0 && airdropAmount == msg.value, InvalidAirdropAmount());
       require(startTime < claimsBurnTime, InvalidClaimTime());
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index b3bb01a..daed5fe 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -1278,13 +1278,14 @@ contract SonicOrderBook is
           InvalidMinQuantity()
         );
         require(!(existingTick != 0 && newTick != 0 && existingTick != newTick), TickCannotBeChanged());
-        _tokenIdInfos[tokenIds[i]] = tokenIdInfos[i];

         SeasonData memory seasonData = _nft.getSeasonData(Season(tokenIds[i]));
         if (seasonData.startTime != 0 && tokenIdInfos[i].isTradeable) {
           // If this season exists make sure it hasn't ended yet if setting tradeable to true
           require(seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME > block.timestamp,
           ↪  SeasonLocked());
         }
+
+        _tokenIdInfos[tokenIds[i]] = tokenIdInfos[i];
       }

       emit SetTokenIdInfos(tokenIds, tokenIdInfos);
```

Note in the above patch we also have moved the storage update for `_tokenIdInfos[tokenIds[i]]` after all the required checks.

Paintswap: Fixed in PR 35.

Cantina Managed: Fix verified.

3.4.6 Tradable timestamp regions do not match in `setTokenIdInfos` and `refreshIsTradeable`

Severity: Low Risk

Context: SonicOrderBook.sol#L386, SonicOrderBook.sol#L1286

Description: In `setTokenIdInfos` the allowed tradable region is defined by:

```
seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME > block.timestamp
```

which means we should be able to flip `_tokenIdInfos[tokenId].isTradeable` to `false` when:

```
seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME <= block.timestamp
```

But the inequality checked in `refreshIsTradeable` is a strict inequality versus the above one which would also allow the case `seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME == block.timestamp`.

Recommendation: Make sure the boundary cases in `setTokenIdInfos` and `refreshIsTradeable` are complementary (ie. one of the two inequalities need to include the case where `seasonData.lockedBurnTime - SET_TRADEABLE_THRESHOLD_LOCKED_BURN_TIME` and `block.timestamp` are equal).

Paintswap: Fixed in PR 39.

Cantina Managed: Fix verified.

3.4.7 Unsafe cast from `uint256` to `uint88`

Severity: Low Risk

Context: SonicOrderBook.sol#L710

Description: `tradeCost - fees` has the type `uint256` but it gets casted down to `uint88` to be added to `claimableTokenInfo.amount`. For sell orders with high prices that get matched (ie, `tradeCost - fees` does not fit in `uint88`) not all the transferred quote tokens by the buyer can be claimed by the seller. The extra amount will be locked in the contract.

Recommendation: Some analysis is required here to see at what prices and for what realistic quote tokens this could happen. But it is recommend to perform a safe cast here.

Paintswap: Fixed in PR 42. Have added a safe cast

```
claimableTokenInfo.amount += SafeCast.toUint88(tradeCost - fees);
```

Cantina Managed: Fix verified.

3.4.8 Potential overflow when increasing `claimableTokenInfo.amount`

Severity: Low Risk

Context: SonicOrderBook.sol#L710

Description: `claimableTokenInfo.amount` is of type `uint88`, and supports a maximum of `309485009e18`. A very edge case is if a quoteToken with a very low price is used, for example, the market price is `S:quoteToken = 2500`, the user creates a Sell order with the price set to `2000e18` (which is the lowest price), the quantity is `154743`, and then immediately takes `154742`, leaving `1`.

Since the price is very low, `getLowestAsk()` always returns it, however when other users take it, it will fail due to overflow.

Recommendation: If on mainnet, ETH/DAI would make it true, but the attacker will have the disadvantage of losing fees and having funds occupied. And since we seem to only choose wS or USDC as quoteToken, it is recommended to just document this issue.

Paintswap: Acknowledged. To be safe I am using `SafeCast.toUint88(tradeCost - fees);` to solve the subtle overflow as it's possible in the future that other base/quote tokens are used. If `amount` itself overflows and reverts that's fine. See PR 43.

Cantina Managed: Acknowledged.

3.5 Gas Optimization

3.5.1 `remainingSegment` can be reused when calculating `storedNormalizedQuantity`

Severity: Gas Optimization

Context: SonicOrderBook.sol#L678

Description: `remainingSegment` can be reused when calculating `storedNormalizedQuantity`.

Recommendation:

```
uint32 storedNormalizedQuantity = uint32(uint256(remainingSegment >> ORDER_ID_SIZE));
```

Paintswap: Fixed in PR 26.

Cantina Managed: Fix verified.

3.5.2 The `if` block when `numSegmentsFullyConsumed != 0` can be simplified

Severity: Gas Optimization

Context: SonicOrderBook.sol#L731-L739

Description: The `if` block when `numSegmentsFullyConsumed != 0` can be simplified.

Recommendation: Note that `node` is a storage pointer:

```
BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree[tokenId].getNode(bestPrice);
```

1. And upon its retrieval it has been checked whether it exists in the `tree[tokenId]` or not. So it is not necessary to perform the same existence check when editing the node. When `tree[tokenId].edit(bestPrice, ...)` is called `node.tombstoneOffset` gets updated and thus the inner `if` block can be simplified to:

- Option 1:

```
if (numSegmentsFullyConsumed != 0) {
  tree[tokenId].edit(bestPrice, uint56(numSegmentsFullyConsumed));

  // Consumed all orders at this price level, so remove it from the tree
  if (node.tombstoneOffset == segments.length) {
    tree[tokenId].remove(bestPrice);
  }
}
```

- Option 2: Due to (1)we can remove the `edit` function from the library and just performs the update in this call site (the only spot `edit` has been used currently):

```
if (numSegmentsFullyConsumed != 0) {
  uint256 newTombstoneOffset = node.tombstoneOffset + numSegmentsFullyConsumed;
  node.tombstoneOffset = uint56(newTombstoneOffset); // <<< best to perform safe cast here.

  // Consumed all orders at this price level, so remove it from the tree
  if (newTombstoneOffset == segments.length) {
    tree[tokenId].remove(bestPrice);
  }
}
```

Paintswap: Fixed in PR 29. Have done a hybrid change. Kept the edit function to keep all state changes inside the library. But have removed the precondition and pass in newTombstoneOffset for gas efficiency.

Cantina Managed: Fix verified.

3.6 Informational

3.6.1 Analysis of `BokkyPooBahsRedBlackTreeLibrary`

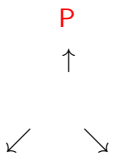
Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The following invariants need to be satisfied for the tree T:

1. Every node is either red or black (●●).
2. All null (EMPTY) nodes are considered black ●.
3. A red node does not have a red child ●$\longrightarrow \times$ ●.
4. Every path from a given node to any of its leaf nodes goes through the same number of black nodes.
5. The root node is black ●. (optional for the general definition but it is enforced in this implemeneta-tion).
6. If a parent node P has a (left/right) child node C, then C's parent is P.

7. If a child node C has a parent node P then, either the left or right child of P is C.
8. Child nodes of a parent node are distinct unless they are both the **empty** node ∅.
9. Empty node's left/right children are **empty**.
10. Empty node's parent should be **empty**. *Although in some rare cases when removing a node from the tree, the tree could end up in a state where the empty node has a **non** empty parent!*.



11. When rotating around a node `n`, the node and its cursor `c` must not be empty nodes:

```
function insert(Tree storage self, uint64 key) internal {
  require(key != EMPTY, KeyCannotBeZero());
  // k =/=
  uint64 cursor = EMPTY;
  uint64 probe = self.root; // could be  for an  tree.

  // find where k needs to be inserted
  while (probe != EMPTY) {
    cursor = probe;
    if (key < probe) {
      probe = self.nodes[probe].left;
    } else {
      probe = self.nodes[probe].right;
    }
  }

  /* Case E.
  * probe = cursor =  for an EMPTY tree
  */

  /* Case N.
  * probe =
  * cursor =/=
  */

  self.nodes[key] = Node({
    parent: cursor,
    left: EMPTY,
    right: EMPTY,
    red: true,
    tombstoneOffset: self.nodes[key].tombstoneOffset
  });
  if (cursor == EMPTY) {
    self.root = key;
  } else if (key < cursor) {
    self.nodes[cursor].left = key;
  } else {
    self.nodes[cursor].right = key;
  }

  /* Case E.
  * probe = cursor =  for an EMPTY tree
  *          <-- c
  *         |
  *        k(x) <-- root
  *        /  \
  *
  */

  /* Case N.L
  * probe =
  * cursor =/=
  *
  *            c
  *           /
  *        k(x)
  *        /  \
```

```
  *
  */

  /* Case N.R
  * probe =
  * cursor =/=
  *
  *        c
  *         \
  *        k(x)
  *        /  \
  *
  */

  // if c is also RED or k(x) is a root we need to fix up the tree going upwards.
  insertFixup(self, key);
}

function insertFixup(Tree storage self, uint64 key) private {
  uint64 cursor;
  while (key != self.root && self.nodes[self.nodes[key].parent].red) {
    /* 1. at this point k should always be a red non-root node. So:
    *         p(x)
    *          |
    *         k(x)
    *    which is a violation. 2 red nodes should not be connected. Inv 3. is violated
    * 2. root of a tree should be a black node. An invariant that should not be broken.
    * 3. Inv 4. is statisfied and we need to make sure the tree modifications would not
    *    break this invariant.
    */
    uint64 keyParent = self.nodes[key].parent;
    // we know that p (keyParent) is not EMPTY since k (key) is not a root
    // but `self.nodes[keyParent].parent`, aka the grandparent, could be EMPTY
    if (keyParent == self.nodes[self.nodes[keyParent].parent].left) {
      /* Assuming there are no EMPTY nodes in the tree, if we end up here we know that
      * the grandparent p' is also not EMPTY
      *           p'
      *         /   \
      *      p(x)    c
      *       |
      *      k(x)
      */
      cursor = self.nodes[self.nodes[keyParent].parent].right;
      if (self.nodes[cursor].red) {
        /* Case LR (I.)
        *           p'
        *         /   \
        *      p(x)    c(x)
        *       |
        *      k(x)
        */
        self.nodes[keyParent].red = false;
        self.nodes[cursor].red = false;
        self.nodes[self.nodes[keyParent].parent].red = true;
        key = self.nodes[keyParent].parent;
        /* Case LR (I.) (will continue)
        *         p'(x) <-- k' (continue the loop with this new value)
        *         /   \
        *      p(o)    c(o)
        *       |
        *      k(x)
        */
      } else {
        /* Case LB
        *           p'
        *         /   \
        *      p(x)    c(o)
        *       |
        *      k(x)
        */
        if (key == self.nodes[keyParent].right) {
          /* Case LBR (II.)
          *           p'
          *         /   \
          *      p(x)    c(o)
          *         \
```

```
  *         k(x)
  */
  key = keyParent;
  /* Case LBR (II.)
  *                p'
  *               /   \
  *     k' --> p(x)   c(o)
  *             /  \
  *           A   k(x)
  *               /  \
  *              B    C
  *
  * can be rotated left since p, k are NOT EMPTY.
  */
  rotateLeft(self, key);
  /* Case LBR (II.)
  *                  p'
  *                /   \
  *              k(x)  c(o)
  *              /  \
  *   k' --> p(x)   C
  *          /   \
  *         A     B
  */
} else {
  /* Case LBL (III.)
  * added this empty block to just leave comments
  *           p'
  *         /   \
  *      p(x)   c(o)
  *      /
  *   k(x)
  */
}
keyParent = self.nodes[key].parent;
self.nodes[keyParent].red = false;
self.nodes[self.nodes[keyParent].parent].red = true;

/* Case LBR (II.)
*               p'(x)
*               /   \
*             k(o)  c(o)
*             /  \
*   k' --> p(x)   C
*          /   \
*         A     B
*
* can be rotated left since p', k are NOT EMPTY.
*/

/* Case LBL (III.)
*         p'(x)
*         /   \
*      p(o)   c(o)
*      /
*   k(x)
*
* can be rotated left since p', p are NOT EMPTY.
*/

rotateRight(self, self.nodes[keyParent].parent);

/* Case LBR (II.) (loop will stop in this case)
*
*                k(o)
*               /   \
*     k' --> p(x)   p'(x)
*            / \     /  \
*           A   B   C   c(o)
*/

/* Case LBL (III.) (loop will stop in this case)
*
*          p(o)
*         /   \
*      k(x)   p'(x)
```

```
      *                \
      *                c(o)
      */
    }
  } else {
    // even if the `self.nodes[keyParent].parent` was EMPTY we could end up here.
    cursor = self.nodes[self.nodes[keyParent].parent].left;

    /* Case (*). self.nodes[keyParent].parent == EMPTY ()
    *             <-- p'
    *          /   \
    *  c -->     p(x)
    *              |
    *             k(x)
    * this case should NOT be possible since otherwise p(x) would be the
    * root of the tree which is a red node. But the root is always black.
    */

    if (self.nodes[cursor].red) {
      /* Case RR (IV.)
      *            p'
      *          /   \
      *      c(x)    p(x)
      *               |
      *              k(x)
      */
      self.nodes[keyParent].red = false;
      self.nodes[cursor].red = false;
      self.nodes[self.nodes[keyParent].parent].red = true;
      key = self.nodes[keyParent].parent;

      /* Case RR (IV.) (continue the loop)
      *          p'(x) <-- k' (continue the loop with this new value)
      *         /    \
      *      c(o)    p(o)
      *               |
      *              k(x)
      */
    } else {
      /* Case (*). self.nodes[keyParent].parent == EMPTY ()
      *             <-- p'
      *          /   \
      *  c -->     p(x)
      *              |
      *             k(x)
      * this case should not be possible since otherwise p(x) would be the
      * root of the tree which is a red node. But the root is always black.
      */
      if (key == self.nodes[keyParent].left) {
        /* Case (*'). self.nodes[keyParent].parent == EMPTY ()
        *             <-- p'
        *          /   \
        *  c -->     p(x)
        *              /
        *             k(x)
        * this case should not be possible since otherwise p(x) would be the
        * root of the tree which is a red node. But the root is always black.
        */

        /* Case RBL (V.)
        *            p'
        *          /   \
        *      c(o)    p(x)
        *               /
        *              k(x)
        */

        key = keyParent;

        /* Case (*', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
        *             <-- p'
        *          /   \
        *  c -->     p(x) <-- k'
        *              /  \
        *           k(x)   C
        *           /  \
```

```
  *         A    B
  *
  * this case should not be possible since otherwise p(x) would be the
  * root of the tree which is a red node. But the root is always black.
  */

  /* Case RBL (V.)
  *           p'
  *         /   \
  *      c(o)   p(x) <-- k'
  *              /  \
  *           k(x)   C
  *           /  \
  *         A    B
  *
  * can be rotated left since p, k are NOT EMPTY.
  */

  rotateRight(self, key);

  /* Case (*', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
  *            <-- p'
  *         /   \
  *  c -->     k(x)
  *             /  \
  *            A   p(x) <-- k'
  *                /  \
  *               B    C
  * this case should not be possible since otherwise p(x) would be the
  * root of the tree which is a red node. But the root is always black.
  */

  /* Case RBL (V.)
  *           p'
  *         /   \
  *      c(o)   k(x)
  *             /  \
  *            A   p(x) <-- k'
  *                /  \
  *               B    C
  */
} else {
  /* Case (*'', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
  *            <-- p'
  *         /   \
  *  c -->     p(x)
  *               \
  *               k(x)
  * this case should not be possible since otherwise p(x) would be the
  * root of the tree which is a red node. But the root is always black.
  */

  /* Case RBR (VI.)
  *           p'
  *         /   \
  *      c(o)   p(x)
  *               \
  *               k(x)
  */
}

/* Case (*', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
*            <-- p'
*         /   \
*  c -->     k(x)
*             /  \
*            A   p(x) <-- k'
*                /  \
*               B    C
* this case should not be possible since otherwise p(x) would be the
* root of the tree which is a red node. But the root is always black.
*/

/* Case RBL (V.)
*           p'
*         /   \
```

```
*      c(o)   k(x)
*              /  \
*             A   p(x) <-- k'
*                 /  \
*                B    C
*/

/* Case (*'', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
*            <-- p'
*         /   \
*  c -->     p(x)
*               \
*               k(x)
* this case should not be possible since otherwise p(x) would be the
* root of the tree which is a red node. But the root is always black.
*/

/* Case RBR (VI.)
*           p'
*         /   \
*      c(o)   p(x)
*               \
*               k(x)
*/

keyParent = self.nodes[key].parent;
self.nodes[keyParent].red = false;
self.nodes[self.nodes[keyParent].parent].red = true; // potentially marking the EMPTY node red !!

/* Case (*', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
*          (x) <-- p'
*         /   \
*  c -->     k(o) <-- p
*             /  \
*            A   p(x) <-- k'
*                /  \
*               B    C
* this case should not be possible since otherwise the original p(x) would be the
* root of the tree which is a red node. But the root is always black.
*/

/* Case RBL (V.)
*          p'(x)
*         /   \
*      c(o)   k(o) <-- p
*             /  \
*            A   p(x) <-- k'
*                /  \
*               B    C
*
* can be rotated left since p', k are NOT EMPTY.
*/

/* Case (*'', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
*          (x) <-- p'
*         /   \
*  c -->     p(o) <-- p
*               \
*               k(x)
* this case should not be possible since otherwise the original p(x) would be the
* root of the tree which is a red node. But the root is always black.
*/

/* Case RBR (VI.)
*          p'(x)
*         /   \
*      c(o)   p(o) <-- p
*               \
*               k(x)
* can be rotated left since p', p are NOT EMPTY.
*/

rotateLeft(self, self.nodes[keyParent].parent);

/* Case (*', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
*          (x) <-- p'
```

```
        *         /   \
        *  c -->     k(o) <-- p
        *             /  \
        *            A   p(x) <-- k'
        *                /  \
        *               B    C
        * the above stays unchanged but `self.root` becomes EMPTY ()
        * this case should not be possible since otherwise the original p(x) would be the
        * root of the tree which is a red node. But the root is always black.
        */

        /* Case RBL (V.) (terminate)
        *          k(o) <-- p
        *         /   \
        *      p'(x)   p(x) <-- k' (loop will stop in this case)
        *      /  \     /  \
        *   c(o)   A   B    C
        */

        /* Case (*'', Impossible Case) self.nodes[keyParent].parent == EMPTY ()
        *          (x) <-- p'
        *         /   \
        *  c -->     p(o) <-- p
        *              \
        *               k(x)
        * the above stays unchanged but `self.root` becomes EMPTY ()
        * this case should not be possible since otherwise the original p(x) would be the
        * root of the tree which is a red node. But the root is always black.
        */

        /* Case RBR (VI.) (terminate)
        *          p(o) <-- p
        *         /   \
        *      p'(x)   k(x)  (loop will stop in this case)
        *      /
        *    c(o)
        */
      }
    }
  }

  // it is very important to keep the root of the tree black!
  self.nodes[self.root].red = false;
}

function remove(Tree storage self, uint64 key) internal {
  require(key != EMPTY, KeyCannotBeZero());
  require(exists(self, key), KeyDoesntExist());
  /* k =/=  and k is either the root or has a non-EMPTY parent. */
  uint64 probe;
  uint64 cursor;
  if (self.nodes[key].left == EMPTY || self.nodes[key].right == EMPTY) {
    cursor = key;

    /* Case EE, both are EMPTY
    *         k <-- c
    *        / \
    *
    */

    /* Case ER, left is EMPTY right is not
    *         k <-- c
    *        / \
    *         R(x)
    *          / \
    *
    */

    /* Case LE, right is EMPTY left is not
    *         k <-- c
    *        / \
    *     L(x)
    *      / \
    *
    */
  } else {
```

```
  cursor = self.nodes[key].right;
  /* Case LR.00, both non-EMPTY
  *         k
  *        / \
  *       L0  R0 <-- c
  */

  while (self.nodes[cursor].left != EMPTY) {
    cursor = self.nodes[cursor].left;
  }

  /* Case LR(00), both non-EMPTY
  *         k
  *        / \
  *       L0  R0 <-- c
  *           /
  *
  */

  /* Case LR(n0), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  *         k
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      Ln <-- c
  *     /
  *
  */
}
if (self.nodes[cursor].left != EMPTY) {
  /* Case LE, right is EMPTY left is not
  *         k <-- c
  *        / \
  *     L(x)
  *      / \
  *
  */
  probe = self.nodes[cursor].left;
  /* Case LE, right is EMPTY left is not
  *                  k <-- c
  *                 / \
  *   probe --> L(x)
  *              / \
  *
  */
} else {
  probe = self.nodes[cursor].right;

  /* Case EE, both are EMPTY
  *         k <-- c
  *        / \
  *           <-- probe
  */

  /* Case ER, left is EMPTY right is not
  *         k <-- c
  *        / \
  *         R(x) <-- probe
  *          / \
  *
  */

  /* Case LR(00.E), both non-EMPTY
  *         k
  *        / \
  *       L0  R0 <-- c
  *           / \
  *              <-- probe
  */

  /* Case LR(00.N), both non-EMPTY
  *         k
```

```
  *        / \
  *       L0  R0 <-- c
  *           / \
  *             R1(x) <-- probe
  *              /  \
  *
  */

  /* Case LR(n0.E), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  *         k
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      Ln <-- c
  *     /  \
  *          <-- probe
  */

  /* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  *         k
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      Ln <-- c
  *     /  \
  *        R_{n+1}(x) <-- probe
  *        / \
  *
  */
}

/* Case LEE, right is EMPTY left is not
*
*                  |
*                  k(o) <-- c, root
*                 / \
*   probe --> L(x)
*             / \
*
*/

/* Case LEP, right is EMPTY left is not
*                  p
*                  |
*                  k(o) <-- c
*                 / \
*   probe --> L(x)
*             / \
*
*/

/* Case EEE, both are EMPTY
*
*         |
*         k <-- c, root
*        / \
*           <-- probe
*/

/* Case EEP, both are EMPTY
*         p
*         |
*         k <-- c
*        / \
*           <-- probe
*/

/* Case ERE, left is EMPTY right is not
```

```
*
*         |
*         k(o) <-- c, root
*        / \
*         R(x) <-- probe
*          / \
*
*/

/* Case ERP, left is EMPTY right is not
*         p
*         |
*         k(o) <-- c
*        / \
*         R(x) <-- probe
*          / \
*
*/

/* Case LR(00.E), both non-EMPTY
*         k
*        / \
*       L0  R0 <-- c
*           / \
*              <-- probe
*/

/* Case LR(00.N), both non-EMPTY
*         k
*        / \
*       L0  R0(o) <-- c
*           / \
*             R1(x) <-- probe
*              /  \
*
*/

/* Case LR(n0.E), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
*         k
*        / \
*       L0  R0
*           /
*         L1
*         /
*       ...
*       /
*      L_{n-1} <-- p
*     /
*    Ln <-- c
*   / \
*      <-- probe
*/

/* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
*         k
*        / \
*       L0  R0
*           /
*         L1
*         /
*       ...
*       /
*      Ln(o) <-- c
*     /  \
*        R_{n+1}(x) <-- probe
*        / \
*
*/

uint64 yParent = self.nodes[cursor].parent;
self.nodes[probe].parent = yParent;
if (yParent != EMPTY) {
  if (cursor == self.nodes[yParent].left) {
    self.nodes[yParent].left = probe;
  } else {
    self.nodes[yParent].right = probe;
```

```
  }
} else {
  self.root = probe;
}

/* Case LEE, k still points on the left to L(x)
*
*                       /
*                     k(o) <-- c
*                    /  / \
* root, probe --> L(x)
*                   / \
*
*/

/* Case LEP, k still points on the left to L(x)
*                        p
*                        /
*                    p   k(o) <-- c
*                    /  / \
*       probe --> L(x)
*                   / \
*
*/

/* Case EEE,
*
*          /
*          k <-- c
*         / \
*            <-- probe, root
*/

/* Case EEP, this is potentially a BAD case since EMPTY's parent would not be EMPTY
*                p
*                /
*          c --> k  p
*               / \ /
*                   <-- probe
*/

/* Case ERE, k still points on the right to R(x)
*
*               /
*      c --> k(o)
*              / \ /
*               R(x) <-- probe, root
*                / \
*
*/

/* Case ERP, k still points on the right to R(x)
*               p
*               /
*      c --> k(o) p
*              / \ /
*               R(x) <-- probe
*                / \
*
*/

/* Case LR(00.E), R0's parent is still k, but k on the right points to EMPTY
*          k               k
*         / \              /
*        L0              R0 <-- c
*                         / \
*                            <-- probe
*/

/* Case LR(00.N), k and R1(x) points to each other and R0 points to k and R1(x)
*          k                          k
*         / \                         /
*        L0  R1(x) <-- probe         R0(o) <-- c
*            / \                     / \
*                                    R1(x) <-- probe
*                                      /  \
```

```
*
*/

/* Case LR(n0.E), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1}
*          k
*         / \
*        L0  R0
*            /
*          L1
*          /
*        ...
*        /
*      L_{n-1} <-- p               l_{n-1} <-- p
*     /                              |
*      <-- probe                    Ln <-- c
*                                   / \
*                                      <-- probe
*/

/* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1} and Ln would still point to R_{n+1}(x) on the right
*              k
*             / \
*            L0  R0
*                /
*              L1
*              /
*            ...
*            /
*          L_{n-1} <-- p               L_{n-1} <-- p
*         /                             |
*        R_{n+1}(x) <-- probe           Ln(o) <-- c
*       / \                             / \
*                                      R_{n+1}(x)
*/

bool doFixup = !self.nodes[cursor].red;

/* Case LEE, k still points on the left to L(x)
*
*                        |
*                       k(o) <-- c
*                      |  / \
* root, probe --> L(x)
*                    / \
*
*/

/* Case LEP, k still points on the left to L(x)
*                         p
*                         |
*                    p    k(o) <-- c
*                    |   / \
*       probe --> L(x)
*                   / \
*
*/

/* Case EEEB,
*
*          |
*          k(o) <-- c
*         / \
*            <-- probe, root
*/

/* Case EEER (not possbile the root is always BLACK o)
*
*          |
*          k(x) <-- c
*         / \
*            <-- probe, root
*/

/* Case EEPB, this is potentially a BAD case since EMPTY's parent would not be EMPTY
```

```
*               p
*               |
*       c --> k(o) p
*              / \ |
*                  <-- probe
*/

/* Case EEPR, this is potentially a BAD case since EMPTY's parent would not be EMPTY
*               p
*               |
*       c --> k(x) p
*              / \ |
*                  <-- probe
*/

/* Case ERE, k still points on the right to R(x)
*
*              |
*      c --> k(o)
*             / \ |
*              R(x) <-- probe, root
*               / \
*
*/

/* Case ERP, k still points on the right to R(x)
*              p
*              |
*      c --> k(o) p
*             / \ |
*              R(x) <-- probe
*               / \
*
*/

/* Case LR(00.E)B, R0's parent is still k, but k on the right points to EMPTY
*         k              k
*        / \             |
*       L0             R0(o) <-- c
*                       / \
*                          <-- probe
*/

/* Case LR(00.E)R, R0's parent is still k, but k on the right points to EMPTY
*         k              k
*        / \             |
*       L0             R0(x) <-- c
*                       / \
*                          <-- probe
*/

/* Case LR(00.N), k and R1(x) points to each other and R0 points to k and R1(x)
*         k                          k
*        / \                         |
*       L0  R1(x) <-- probe         R0(o) <-- c
*           / \                     / \
*                                   R1(x) <-- probe
*                                     /  \
*
*/

/* Case LR(n0.E)B, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1}
*         k
*        / \
*       L0  R0
*           /
*         L1
*         /
*       ...
*       /
*      L_{n-1} <-- p              l_{n-1} <-- p
*     /                             |
*      <-- probe                   Ln(o) <-- c
*                                  / \
*                                     <-- probe
```

```
*/

/* Case LR(n0.E)R, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1}
*         k
*        / \
*       L0  R0
*           /
*         L1
*         /
*       ...
*       /
*      L_{n-1} <-- p              l_{n-1} <-- p
*     /                             /
*      <-- probe                   Ln(x) <-- c
*                                  / \
*                                     <-- probe
*/

/* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1} and Ln would still point to R_{n+1}(x) on the right
*             k
*            / \
*           L0  R0
*               /
*             L1
*             /
*           ...
*           /
*          L_{n-1} <-- p              L_{n-1} <-- p
*         /                           /
*        R_{n+1}(x) <-- probe         Ln(o) <-- c
*       / \                           / \
*                                    R_{n+1}(x)
*/

if (cursor != key) {
  /* Case LR(00.E)B, R0's parent is still k, but k on the right points to EMPTY
  *         k              k
  *        / \             /
  *       L0             R0(o) <-- c
  *                      / \
  *                         <-- probe
  */

  /* Case LR(00.E)R, R0's parent is still k, but k on the right points to EMPTY
  *         k              k
  *        / \             /
  *       L0             R0(x) <-- c
  *                      / \
  *                         <-- probe
  */

  /* Case LR(00.N), k and R1(x) points to each other and R0 points to k and R1(x)
  *         k                        k
  *        / \                       /
  *       L0  R1(x) <-- probe       R0(o) <-- c
  *           / \                   / \
  *                                 R1(x) <-- probe
  *                                   /  \
  *
  */

  /* Case LR(n0.E)B, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  * Ln's parent would still be L_{n-1}
  *         k
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      L_{n-1} <-- p              l_{n-1} <-- p
  *     /                             /
  *      <-- probe                   Ln(o) <-- c
```

```
*                                / \
*                                   <-- probe
*/

/* Case LR(n0.E)R, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1}
*          k
*         / \
*        L0  R0
*            /
*          L1
*          /
*        ...
*        /
*      L_{n-1} <-- p              l_{n-1} <-- p
*     /                             /
*      <-- probe                  Ln(x) <-- c
*                                 / \
*                                    <-- probe
*/

/* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1} and Ln would still point to R_{n+1}(x) on the right
*              k
*             / \
*            L0  R0
*                /
*              L1
*              /
*            ...
*            /
*          L_{n-1} <-- p              L_{n-1} <-- p
*         /                            /
*        R_{n+1}(x) <-- probe         Ln(o) <-- c
*       / \                           / \
*                                    R_{n+1}(x)
*/

replaceParent(self, cursor, key);
self.nodes[cursor].left = self.nodes[key].left;
self.nodes[self.nodes[cursor].left].parent = cursor;
self.nodes[cursor].right = self.nodes[key].right;
self.nodes[self.nodes[cursor].right].parent = cursor;
self.nodes[cursor].red = self.nodes[key].red;
(cursor, key) = (key, cursor);

/* Case LR(00.E)B, R0's parent is p, but k on the right points to EMPTY
*          p
*          |
*        k(?) <-- c'     p
*        / \             |
*       L0             R0(?) <-- k'
*                       / \
*                      L0   <-- probe
*/

/* Case LR(00.E)R, R0's parent is p, but k on the right points to EMPTY
* R0 was originally RED R0(x)
*          p
*          |
*        k(?) <-- c'     p
*        / \             |
*       L0             R0(?) <-- k'
*                       / \
*                      L0   <-- probe
*/

/* Case LR(00.N), k points to p, L0, and R1(x)
*          p
*          |
*          k(?) <-- c'               p
*         / \                        |
*        L0  R1(x) <-- probe        R0(?) <-- k'
*            / \                    / \
*                                L0   R1(x) <-- probe
*                                     /  \
```

```
  *
  */

  /* Case LR(n0.E)B, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  * k points to p, L0, and R0, Ln was originally BLACK Ln(o)
  * k(?) <-- c'
  *
  *         p
  *         |
  *         Ln(?) <-- k'
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      L_{n-1}
  *     /
  *      <-- probe
  *
  *
  */

  /* Case LR(n0.E)R, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  * k points to p, L0, and R0, Ln was originally RED Ln(x)
  * k(?) <-- c'
  *
  *         p
  *         |
  *         Ln(?) <-- k'
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      L_{n-1}
  *     /
  *      <-- probe
  *
  *
  */

  /* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  * Ln's parent would still be L_{n-1} and Ln would still point to R_{n+1}(x) on the right
  * k still points to p, L0, and R0
  * Ln was originally BLACK Ln(o)
  * k <-- c'
  *             p
  *             |
  *             Ln(?) <-- k'
  *            / \
  *           L0  R0
  *               /
  *             L1
  *             /
  *           ...
  *           /
  *          L_{n-1} <-- p
  *         /
  *        R_{n+1}(x) <-- probe
  *       / \
  *
  */

}
if (doFixup) {
  /* Case LEE, k still points on the left to L(x)
  *
  *                       |
  *                      k(o) <-- c
  *                    |  / \
  * root, probe --> L(x)
  *                   / \
```

```
*
*/

/* Case LEP, k still points on the left to L(x)
*                        p
*                        /
*                   p   k(o) <-- c
*                   /  / \
*       probe --> L(x)
*                  / \
*
*/

/* Case EEEB,
*
*         /
*         k(o) <-- c
*        / \
*           <-- probe, root
*/

/* Case EEPB, this is potentially a BAD case since EMPTY's parent would not be EMPTY
*               p
*               /
*       c --> k(o) p
*              / \ /
*                 <-- probe
*/

/* Case ERE, k still points on the right to R(x)
*
*              /
*      c --> k(o)
*             / \ /
*              R(x) <-- probe, root
*               / \
*
*/

/* Case ERP, k still points on the right to R(x)
*              p
*              /
*      c --> k(o) p
*             / \ /
*              R(x) <-- probe
*               / \
*
*/

/* Case LR(00.E)B, R0's parent is still k, but k on the right points to EMPTY
* R0 was originally BLACK R0(o)
*         p
*         /
*        k(?) <-- c'     p
*        / \             /
*       L0             R0(?) <-- k'
*                       / \
*                      L0   <-- probe
*/

/* Case LR(00.N), k and R1(x) points to each other and R0 points to k and R1(x)
* R0 was originally BLACK R0(o)
*         p
*         /
*         k(?) <-- c'                p
*        / \                         /
*       L0  R1(x) <-- probe         R0(?) <-- k'
*           / \                     / \
*                                L0   R1(x) <-- probe
*                                     /  \
*
*/

/* Case LR(n0.E)B, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* k points to p, L0, and R0, Ln was originally BLACK Ln(o)
* k(?) <-- c'
```

```
  *
  *         p
  *         |
  *         Ln(?) <-- k'
  *        / \
  *       L0  R0
  *           /
  *         L1
  *         /
  *       ...
  *       /
  *      L_{n-1}
  *     /
  *      <-- probe
  *
  *
  */

  /* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
  * Ln's parent would still be L_{n-1} and Ln would still point to R_{n+1}(x) on the right
  * k still points to p, L0, and R0
  * Ln was originally BLACK Ln(o)
  * k <-- c'
  *             p
  *             |
  *             Ln(?) <-- k'
  *            / \
  *           L0  R0
  *               /
  *             L1
  *             /
  *           ...
  *           /
  *          L_{n-1} <-- p
  *         /
  *        R_{n+1}(x) <-- probe
  *       / \
  *
  */
  removeFixup(self, probe);
}
// Don't delete the node, so that we can re-use the tombstone offset if re-adding this price
self.nodes[cursor].parent = EMPTY;

/* Case LEE, k still points on the left to L(o)
* k(o) still points to L(o)
*
*
*
*                    |
* root, probe --> L(o)
*                   / \
*
*/

/* Case LEP, k still points on the left to L(o)
* k(o) still points to L(o)
* no fix up needed besides changing the color of L from RED to BLACK since L was originally RED
*
*                    p
*                    |
*       probe --> L(o)
*                   / \
*
*
* Although now both Inv 3. and Inv 4. might be broken.
*/

/* Case EEEB,
*
*         |
*         k(o) <-- c
*        / \
*           <-- probe, root
*/
```

```
/* Case EEER (not possbile the root is always BLACK o)
*
*         |
*         k(x) <-- c
*        / \
*           <-- probe, root
*/

/* Case EEPB, this is potentially a BAD case since EMPTY's parent would not be EMPTY
*                  needs a fix up:
*                 ---------------
*               |
*       c --> k(o) p
*              / \ |
*                  <-- probe
*/

/* Case EEPR, this is potentially a BAD case since EMPTY's parent would not be EMPTY
*
*               |
*       c --> k(x) p
*              / \ |
*                  <-- probe
*/

/* Case ERE, k still points on the right to R(o)
* does not need a fix up besides changing the color of R from RED to BLACK
*
*              |
*      c --> k(o)
*             / \ |
*              R(o) <-- probe, root
*               / \
*
*/

/* Case ERP, k still points on the right to R(o)
* * does not need a fix up besides changing the color of R from RED to BLACK
*
*              |
*      c --> k(o) p
*             / \ |
*              R(o) <-- probe
*               / \
*
*/

/* Case LR(OO.E)B
*                      below needs to be fixed up
*         |             --------------------------
*        k(?) <-- c'     p
*        / \             |
*       LO              RO(?) <-- k'
*                        / \
*                       LO   <-- probe
*/

/* Case LR(OO.E)R
* RO was originally RED RO(x)
*                      below does NOT need to be fixed up
*         |             --------------------------
*         |
*        k(?) <-- c'     p
*        / \             |
*       LO              RO(?) <-- k'
*                        / \
*                       LO   <-- probe
*/

/* Case LR(OO.N), k still points to LO and R1(o)
*                      the fix up below was only changing the color of R1 from RED to BLACK
*         |             --------------------------
*         k(?) <-- c'                p
*        / \                         |
*       LO  R1(o) <-- probe         RO(?) <-- k'
*           / \                     / \
```

```
*                               L0   R1(o) <-- probe
*                                     /  \
*
*/

/* Case LR(n0.E)B, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* k points to , L0, and R0, Ln was originally BLACK Ln(o)
*
*  |
* k(?) <-- c'
*
* below needs a fix up:
*         p
*         |
*         Ln(?) <-- k'
*        / \
*       L0  R0
*           /
*         L1
*         /
*       ...
*       /
*      L_{n-1}
*     /
*      <-- probe
*
*
*/

/* Case LR(n0.E)R, R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* k points to , L0, and R0, Ln was originally RED Ln(x)
*
*  |
* k(?) <-- c'
*
* below does NOT need a fix up:
*         p
*         |
*         Ln(?) <-- k'
*        / \
*       L0  R0
*           /
*         L1
*         /
*       ...
*       /
*      L_{n-1}
*     /
*      <-- probe
*
*
*/

/* Case LR(n0.N), R0, L0, ..., Ln non-EMPTY, n in {1,2, ... }
* Ln's parent would still be L_{n-1} and Ln would still point to R_{n+1}(x) on the right
* k still points to , L0, and R0
* Ln was originally BLACK Ln(o)
*
* |
* k <-- c'
*
* below needs a fix up:
*             p
*             |
*             Ln(?) <-- k'
*            / \
*           L0  R0
*               /
*             L1
*             /
*           ...
*           /
*          L_{n-1} <-- p
*         /
*        R_{n+1}(x) <-- probe
*       / \
```

```
  *
  */


}

function removeFixup(Tree storage self, uint64 key) private {
  uint64 cursor;
  while (key != self.root && !self.nodes[key].red) {
    /* k(o) , key is a black node. It could be an EMPTY node .
    * If we end up in this while loop, the assumption is that the paths from the parent
    * to the node k have one less BLACK nodes compared to all other paths, ie Inv 4.
    * is broken by just one node. The operations in the while loop try to fix this
    * in the subtree where changes happen and push the inconsistency up the tree
    * till it eventually gets satisfied.
    * The the very beginning when entering this while loop from the `remove` function we have:
    * k =
    *               p          p
    *               |          |
    *                <-- k     c
    *
    * where the EMPTY node's parent is marked to be `p`.
    * In general if `c` is the sibling of `k` since Inv 4. is broken by 1 we have that:
    * n(k) + 1 + 1 = n(c) + 1? (1? depends on the color of c)
    * where n(X) = number of black nodes from X to any of its leaf nodes not including X
    */


    uint64 keyParent = self.nodes[key].parent;
    /*  k(o) -- p
    *
    *         p
    *         |
    *        k(o)
    */
    if (key == self.nodes[keyParent].left) {
      /* k(o) -- left -- p
      *
      *         p
      *      /    \
      *    k(o)    c
      *           / \
      *          A   G
      *         / \
      *        B   D
      *       / \
      *      E   F
      * p, the keyParent, should not be EMPTY here unless invariants are broken for the tree
      */

      cursor = self.nodes[keyParent].right;
      /* k(o) -- left -- p -- right -- c
      *         p
      *      /    \
      *    k(o)    c      (c could NOT be EMPTY, since n(k) + 1 + 1 = n(c) + 1?)
      *           / \
      *          A   G
      *         / \
      *        B   D
      *       / \
      *      E   F
      *
      * n(k) + 1 + 1 = n(c) + 1?
      */

      if (self.nodes[cursor].red) {
        /* Case R.
        *
        *       p(o)           (color of p is determined by c(x) since 2 REDs cannot be connected)
        *      /   \
        *    k(o)  c(x)        (c is NOT EMPTY, since n(k) + 1 + 1 = n(c))
        *          / \
        *        A(o)  G(o)    (A/G could NOT be EMPTY, otherwise Inv 4. (-1) would be broken. And their colors
        ↪  are determined by c(x))
        *         / \
        *        B   D         (n(k) + 1 = n(B) + 1? = n(D) + 1?) here 1? depends on the color of the
        ↪  corresponding node in the summation
```

```
    *       / \
    *      E   F
    *
    * Since Inv 4. is broken by 1 we know:
    * n(k) + 1 = n(A) = n(G)
    * where n(X) = number of black nodes from X to any of its leaf nodes not including X
    */

    self.nodes[cursor].red = false;
    self.nodes[keyParent].red = true;
    /* Case R.
    *
    *       p(x)
    *      /   \
    *    k(o)  c(o)         (c is NOT EMPTY, since n(k) + 1 + 1 = n(c))
    *           / \
    *         A(o)  G(o)    (A/G could NOT be EMPTY, since n(k) + 1 = n(A) = n(G))
    *         / \
    *        B   D          (n(k) + 1 = n(B) + 1? = n(D) + 1?)
    *       / \
    *      E   F
    *
    * can be rotated left since p, c are NOT EMPTY
    */

    rotateLeft(self, keyParent);
    /* Case R.
    *
    *          c(o)
    *         /   \
    *       p(x)   G(o)    (G could NOT be EMPTY, n(k) + 1 = n(G))
    *       /  \
    *    k(o)  A(o)        (A could NOT be EMPTY, n(k) + 1 = n(A))
    *          / \
    *         B   D        (n(k) + 1 = n(B) + 1? = n(D) + 1?)
    *        / \
    *       E   F
    */

    cursor = self.nodes[keyParent].right;
    /* Case R.
    *
    *          c(o)
    *         /   \
    *       p(x)  G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
    *       /  \
    *    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(A))
    *          / \
    *         B   D          (n(k) + 1 = n(B) + 1? = n(D) + 1?)
    *        / \
    *       E   F
    */
  } else {
    /* Case B. (II)
    *         p
    *      /    \
    *    k(o)   c(o)         (c could NOT be EMPTY, n(k) + 1 = n(c))
    *           / \
    *          A   G         (n(k) + 1 = n(A) + 1? = n(G) + 1?)
    *         / \
    *        B   D
    *       / \
    *      E   F
    */
  }

  if (!self.nodes[self.nodes[cursor].left].red && !self.nodes[self.nodes[cursor].right].red) {
    /* Case R.BB (I.0)
    *
    *          c(o)
    *         /   \
    *       p(x)   G(o)      (G could NOT be EMPTY, since n(k) + 1 = n(G))
    *       /  \
    *    k(o)   A(o) <-- c' (A could NOT be EMPTY, since n(k) + 1 = n(A))
    *          / \
    *        B(o) D(o)       (n(k) = n(B) = n(D))
```

```
      *        / \
      *       E   F
      */

      /* Case B.BB (II.0)
      *
      *         p
      *      /    \
      *    k(o)   c(o)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
      *           / \
      *        A(o) G(o)       (n(k) = n(A) = n(G))
      *         / \
      *        B   D
      *       / \
      *      E   F
      */

      self.nodes[cursor].red = true;
      key = keyParent;

      /* Case R.BB (I.0) (this case terminates). But currently Inv 3. is violated
      *          But will be corrected outside of the while loop
      *
      *             c(o)
      *            /   \
      *   k' --> p(x)   G(o)      (G could NOT be EMPTY, since n(k) + 1 = n(G))
      *          /  \
      *       k(o) A(x) <-- c' (A could NOT be EMPTY, since n(k) + 1 = n(A))
      *            / \
      *           B(o) D(o)     (n(k) = n(B) = n(D))
      *           / \
      *          E   F
      */

      /* Case B.BB (II.0) (might continue)
      *
      *         p <-- k'
      *      /    \
      *    k(o)   c(x)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
      *           / \
      *        A(o) G(o)       (n(k) = n(A) = n(G))
      *         / \
      *        B   D
      *       / \
      *      E   F
      */

      /* Case B.BB.PB (will continue if not root)
      *
      *        p(o) <-- k'     (Inv 4. would be broken by 1 if this loop continues, ie n(p) + 1 = n(sibling of
      ↪  p))
      *      /    \
      *    k(o)   c(x)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
      *           / \
      *        A(o) G(o)       (n(k) = n(A) = n(G))
      *         / \
      *        B   D
      *       / \
      *      E   F
      */

      /* Case B.BB.PR (will terminate) But currently Inv 3. is violated
      *          But will be corrected outside of the while loop
      *
      *        p(x) <-- k'
      *      /    \
      *    k(o)   c(x)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
      *           / \
      *        A(o) G(o)       (n(k) = n(A) = n(G))
      *         / \
      *        B   D
      *       / \
      *      E   F
      */
    } else /*  cases in this block terminate */ {
      /* Case R.R? (I.1)
```

```
*
*          c(o)
*         /   \
*       p(x)   G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
*       /  \
*    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(A))
*          / \
*        B(x) D       (n(k) + 1 = n(B) = n(D) + 1?)
*        / \
*       E   F
*/

/* Case R.?R (I.2)
*
*          c(o)
*         /   \
*       p(x)   G(o)      (G could NOT be EMPTY, since n(k) + 1 = n(G))
*       /  \
*    k(o)   A(o) <-- c' (A could NOT be EMPTY, since n(k) + 1 = n(A))
*          / \
*         B  D(x)         (n(k) + 1 = n(B) + 1? = n(D))
*        / \
*       E   F
*/

/* Case B.R? (II.1)
*
*         p
*      /    \
*    k(o)   c(o)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
*           / \
*        A(x)  G         (n(k) + 1 = n(A) = n(G) + 1?)
*         / \
*        B   D
*       / \
*      E   F
*/

/* Case B.?R (II.2)
*
*         p
*      /    \
*    k(o)   c(o)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
*           / \
*          A   G(x)      (n(k) + 1 = n(A) = n(G) + 1?)
*         / \
*        B   D
*       / \
*      E   F
*/

if (!self.nodes[self.nodes[cursor].right].red) {
  /* Case R.RB (I.1.1)
  *
  *          c(o)
  *         /   \
  *       p(x)   G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
  *       /  \
  *    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(A))
  *          / \
  *        B(x) D(o)       (n(k) + 1 = n(B) = n(D) + 1) B is not EMPTY
  *        / \
  *       E   F
  */

  /* Case B.RB (II.1.1)
  *
  *         p
  *      /    \
  *    k(o)   c(o)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
  *           / \
  *        A(x)  G(o)      (n(k) + 1 = n(A) = n(G) + 1 = n(D) + 1?) A is not EMPTY
  *         / \
  *        B   D
  *       / \
  *      E   F
```

```
    */

    self.nodes[self.nodes[cursor].left].red = false;
    self.nodes[cursor].red = true;

    /* Case R.RB (I.1.1)
    *
    *          c(o)
    *         /   \
    *       p(x)   G(o)     (G could NOT be EMPTY)
    *       /  \
    *    k(o)  A(x) <-- c' (A could NOT be EMPTY)
    *          / \
    *        B(o) D(o)      (n(k) + 1 = n(B) = n(D) + 1)  B is not EMPTY
    *        / \
    *       E   F           (n(k) + 1 = n(E) + 1? = n(F) + 1?)
    *
    * can be rotated right since A, B are NOT EMPTY.
    */

    /* Case B.RB (II.1.1)
    *
    *         p
    *      /    \
    *    k(o)   c(x)         (c could NOT be EMPTY)
    *           / \
    *        A(o)  G(o)      (n(k) + 1 = n(A) = n(G) + 1) A is not EMPTY
    *         / \
    *        B   D           (n(k) + 1 = n(B) + 1? = n(D) + 1?)
    *       / \
    *      E   F
    *
    * can be rotated right since c, A are NOT EMPTY.
    */

    rotateRight(self, cursor);
    cursor = self.nodes[keyParent].right;

    /* Case R.RB (I.1.1)
    *
    *          c(o)
    *         /   \
    *       p(x)   G(o)          (G could NOT be EMPTY)
    *       /  \
    *    k(o)  B(o) <-- c''      (B is not EMPTY)
    *          / \
    *         E  A(x) <-- c'     (A could NOT be EMPTY)
    *            /  \
    *           F   D(o)         (n(k) + 1 = n(E) + 1? = n(F) + 1? = n(D) + 1)
    */

    /* Case B.RB (II.1.1)
    *
    *         p
    *      /    \
    *    k(o)   A(o) <-- c'      (A is not EMPTY)
    *           /   \
    *          B    c(x)         (c could NOT be EMPTY)
    *         / \   /  \
    *        E   F D   G(o)      (n(k) + 1 = n(G) + 1 = n(D) + 1? = n(B) + 1?)
    */
} else {
    /* Case R.RR (I.1.2)
    *
    *          c(o)
    *         /   \
    *       p(x)   G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
    *       /  \
    *    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(A))
    *           / \
    *        B(x) D(x)       (n(k) + 1 = n(B) = n(D))
    *        / \
    *       E   F
    */

    /* Case R.?R (I.2)
```

```
  *
  *          c(o)
  *         /   \
  *       p(x)   G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
  *       /  \
  *    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(G))
  *          / \
  *         B  D(x)        (n(k) + 1 = n(B) + 1? = n(D))
  *        / \
  *       E   F
  */

  /* Case B.RR (II.1.2)
  *
  *         p
  *      /    \
  *    k(o)   c(o)         (c could NOT be EMPTY, n(k) + 1 = n(c))
  *           / \
  *        A(x)  G(X)      (n(k) + 1 = n(A) = n(G))
  *         / \
  *        B   D
  *       / \
  *      E   F
  */

  /* Case B.?R (II.2)
  *
  *         p
  *      /    \
  *    k(o)   c(o)         (c could NOT be EMPTY, n(k) + 1 = n(c))
  *           / \
  *          A   G(x)      (n(k) + 1 = n(A) + 1? = n(G))
  *         / \
  *        B   D
  *       / \
  *      E   F
  */
}

/* Case R.RB (I.1.1)
*
*          c(o)
*         /   \
*       p(x)   G(o)         (G could NOT be EMPTY)
*       /  \
*    k(o)  B(o) <-- c''     (B is not EMPTY)
*          / \
*         E  A(x) <-- c'    (A could NOT be EMPTY)
*            /  \
*           F   D(o)        (n(k) + 1 = n(E) + 1? = n(F) + 1? = n(D) + 1)
*
* can be rotated left since p, B are NOT EMPTY.
*/

/* Case B.RB (II.1.1)
*
*        p(?)
*      /    \
*    k(o)   A(o) <-- c'     (A is not EMPTY)
*           /   \
*          B    c(x)       (c could NOT be EMPTY)
*         / \   /  \
*        E   F D   G(o)     (n(k) + 1 = n(G) + 1 = n(D) + 1? = n(B) + 1?)
*
* can be rotated left since p, A are NOT EMPTY.
*/

/* Case R.RR (I.1.2)
*
*          c(o)
*         /   \
*       p(x)   G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
*       /  \
*    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(A))
*          / \
*        B(x) D(x)       (n(k) + 1 = n(B) = n(D))
```

```
*        / \
*       E   F
*
* can be rotated left since p, A are NOT EMPTY.
*/

/* Case R.?R (I.2)
*
*          c(o)
*         /   \
*       p(x)   G(o)      (G could NOT be EMPTY, n(k) + 1 = n(G))
*       /  \
*    k(o)   A(o) <-- c' (A could NOT be EMPTY, n(k) + 1 = n(G))
*          / \
*         B  D(x)        (n(k) + 1 = n(B) + 1? = n(D))
*        / \
*       E   F
*
* can be rotated left since p, A are NOT EMPTY.
*/

/* Case B.RR (II.1.2)
*
*         p
*      /    \
*    k(o)   c(o)         (c could NOT be EMPTY, n(k) + 1 = n(c))
*           / \
*        A(x)  G(X)      (n(k) + 1 = n(A) = n(G))
*         / \
*        B   D
*       / \
*      E   F
*
* can be rotated left since p, c are NOT EMPTY.
*/

/* Case B.?R (II.2)
*
*         p
*      /    \
*    k(o)   c(o)         (c could NOT be EMPTY, n(k) + 1 = n(c))
*           / \
*          A   G(x)      (n(k) + 1 = n(A) + 1? = n(G))
*         / \
*        B   D
*       / \
*      E   F
*
* can be rotated left since p, c are NOT EMPTY.
*/

self.nodes[cursor].red = self.nodes[keyParent].red;
self.nodes[keyParent].red = false;
self.nodes[self.nodes[cursor].right].red = false;
rotateLeft(self, keyParent);

/* Case R.RB (I.1.1) (will terminate)
*
*              c(o)
*             /   \
*   c'' --> B(x)   G(o)     (B/G could NOT be EMPTY)
*           /  \
*        p(o)   A(o) <-- c' (A could NOT be EMPTY)
*        / \     / \
*     k(o)  E   F  D(o)     (n(k) + 1 = n(E) + 1? = n(F) + 1? = n(D) + 1 = n(G))
*/

/* Case B.RB (II.1.1) (will terminate)
*
*          A(?) <-- c'      (A is not EMPTY)
*         /    \
*       p(o)    c(o)        (c could NOT be EMPTY)
*       /  \     / \
*    k(o)  B    D   G(o)    (n(k) + 1 = n(G) + 1 = n(D) + 1? = n(B) + 1?)
*         / \
*        E   F
```

```
      */

      /* Case R.RR (I.1.2) (will terminate)
      *
      *              c(o)
      *             /   \
      *    c' --> A(x)   G(o)      (A/G could NOT be EMPTY) (n(k) + 1 = n(G))
      *           /  \
      *        p(o)  D(o)          (n(k) + 1 = n(B) = n(D) = n(G))
      *        /  \
      *     k(o)  B(x)
      *           / \
      *          E   F
      */

      /* Case R.?R (I.2) (will terminate)
      *
      *              c(o)
      *             /   \
      *    c' --> A(x)   G(o)      (A/G could NOT be EMPTY)
      *           /  \
      *        p(o)  D(o)          (n(k) + 1 = n(B) + 1? = n(D) = n(G))
      *        /  \
      *     k(o)   B
      *           / \
      *          E   F
      */

      /* Case B.RR (II.1.2) (will terminate)
      *
      *          c(?)            (c could NOT be EMPTY, n(k) + 1 = n(c))
      *        /     \
      *      p(o)    G(o)        (n(k) + 1 = n(A) = n(G))
      *      /  \
      *    k(o) A(x)
      *         / \
      *        B   D
      *       / \
      *      E   F
      */

      /* Case B.?R (II.2) (will terminate)
      *
      *          c(?)            (c could NOT be EMPTY, n(k) + 1 = n(c))
      *        /     \
      *      p(o)    G(o)        (n(k) + 1 = n(A) + 1? = n(G))
      *      /  \
      *    k(o)  A
      *         / \
      *        B   D
      *       / \
      *      E   F
      */

      key = self.root; // cause the while loop to break
    }
  } else /* mirrored cases */ {
    cursor = self.nodes[keyParent].left;
    if (self.nodes[cursor].red) {
      self.nodes[cursor].red = false;
      self.nodes[keyParent].red = true;
      rotateRight(self, keyParent);
      cursor = self.nodes[keyParent].left;
    }
    if (!self.nodes[self.nodes[cursor].right].red && !self.nodes[self.nodes[cursor].left].red) {
      self.nodes[cursor].red = true;
      key = keyParent;
    } else {
      if (!self.nodes[self.nodes[cursor].left].red) {
        self.nodes[self.nodes[cursor].right].red = false;
        self.nodes[cursor].red = true;
        rotateLeft(self, cursor);
        cursor = self.nodes[keyParent].left;
      }
      self.nodes[cursor].red = self.nodes[keyParent].red;
      self.nodes[keyParent].red = false;
```

```
      self.nodes[self.nodes[cursor].left].red = false;
      rotateRight(self, keyParent);
      key = self.root;
    }
  }

  // cases that might continue:

  /* Case B.BB.PB (will continue if not root)
  *
  *        p(o) <-- k'    (Inv 4. would be broken by 1 if this loop continues, ie n(p) + 1 = n(sibling of p))
  *      /    \
  *    k(o)   c(x)        (c could NOT be EMPTY, since n(k) + 1 = n(c))
  *           / \
  *        A(o) G(o)      (n(k) = n(A) = n(G))
  *         / \
  *        B   D
  *       / \
  *      E   F
  */

  /* Case mirrored(B.BB.PB) */
}

/* Case R.BB (I.0) Currently Inv 3. is violated
*
*             c(o)
*            /   \
*   k' --> p(x)   G(o)     (G could NOT be EMPTY, since n(k) + 1 = n(G))
*           /  \
*       k(o) A(x) <-- c' (A could NOT be EMPTY, since n(k) + 1 = n(A))
*             / \
*           B(o) D(o)     (n(k) = n(B) = n(D))
*           / \
*          E   F
*/

/* Case B.BB.PR Currently Inv 3. is violated
*
*        p(x) <-- k'
*      /    \
*    k(o)   c(x)        (c could NOT be EMPTY, since n(k) + 1 = n(c))
*           / \
*        A(o) G(o)      (n(k) = n(A) = n(G))
*         / \
*        B   D
*       / \
*      E   F
*/

/* Case R.RB (I.1.1)
*  k' --> root
*
*              c(o)
*             /   \
*   c'' --> B(x)   G     (G could NOT be EMPTY, otherwise Inv 4. would be broken)
*           /  \
*        p(o)   A(o) <-- c' (A could NOT be EMPTY, otherwise Inv 4. would be broken)
*        / \     / \
*     k(o)  E   F  D(o)
*/

/* Case B.RB (II.1.1)
*  k' --> root
*
*          A(?) <-- c'
*         /    \
*       p(o)    c(o)        (c could NOT be EMPTY, otherwise Inv 4. would be broken)
*       /  \     / \
*    k(o)  B    D   G(o)
*         / \
*        E   F
*/

/* Case R.RR (I.1.2)
* k' --> root
```

```
*
*              c(o)
*             /   \
*    c' --> A(x)   G(o)      (A/G could NOT be EMPTY) (n(k) + 1 = n(G))
*           /  \
*        p(o)  D(o)          (n(k) + 1 = n(B) = n(D) = n(G))
*        /  \
*     k(o)  B(x)
*           / \
*          E   F
*/

/* Case R.?R (I.2)
* k' --> root
*
*              c(o)
*             /   \
*    c' --> A(x)   G(o)      (A/G could NOT be EMPTY)
*           /  \
*        p(o)  D(o)          (n(k) + 1 = n(B) + 1? = n(D) = n(G))
*        /  \
*     k(o)   B
*           / \
*          E   F
*/

/* Case B.RR (II.1.2)
* k' --> root
*
*          c(?)         (c could NOT be EMPTY, n(k) + 1 = n(c))
*        /     \
*      p(o)   G(o)      (n(k) + 1 = n(A) = n(G))
*      /  \
*    k(o) A(x)
*         / \
*        B   D
*       / \
*      E   F
*/

/* Case B.?R (II.2)
* k' --> root
*
*          c(?)         (c could NOT be EMPTY, n(k) + 1 = n(c))
*        /     \
*      p(o)   G(o)      (n(k) + 1 = n(A) + 1? = n(G))
*      /  \
*    k(o)  A
*         / \
*        B   D
*       / \
*      E   F
*/

/* mirrored cases */

self.nodes[key].red = false;

/* Case R.BB (I.0).
*
*             c(o)
*            /   \
*   k' --> p(o)   G(o)      (G could NOT be EMPTY, since n(k) + 1 = n(G))
*          /  \
*       k(o) A(x) <-- c' (A could NOT be EMPTY, since n(k) + 1 = n(A))
*            / \
*           B(o) D(o)     (n(k) = n(B) = n(D))
*           / \
*          E   F
*/

/* Case B.BB.PR
*
*        p(o) <-- k'
*      /     \
*    k(o)   c(x)         (c could NOT be EMPTY, since n(k) + 1 = n(c))
```

```
*           / \
*         A(o) G(o)       (n(k) = n(A) = n(G))
*          / \
*         B   D
*        / \
*       E   F
*/

/* Case R.RB (I.1.1)
*  k' --> root(o)
*
*              c(o)
*             /   \
*   c'' --> B(x)   G(o)     (B/G could NOT be EMPTY)
*           /  \
*         p(o)   A(o) <-- c' (A could NOT be EMPTY)
*         / \     / \
*     k(o)  E   F  D(o)     (n(k) + 1 = n(E) + 1? = n(F) + 1? = n(D) + 1 = n(G))
*/

/* Case B.RB (II.1.1)
*  k' --> root(o)
*
*           A(?) <-- c'      (A is not EMPTY)
*          /    \
*        p(o)    c(o)        (c could NOT be EMPTY)
*        /  \     / \
*    k(o)  B    D   G(o)     (n(k) + 1 = n(G) + 1 = n(D) + 1? = n(B) + 1?)
*         / \
*        E   F
*/

/* Case R.RR (I.1.2)
* k' --> root(o)
*
*              c(o)
*             /   \
*    c' --> A(x)   G(o)     (A/G could NOT be EMPTY) (n(k) + 1 = n(G))
*           /  \
*         p(o)  D(o)         (n(k) + 1 = n(B) = n(D) = n(G))
*         /  \
*     k(o)  B(x)
*           / \
*          E   F
*/

/* Case R.?R (I.2)
* k' --> root(o)
*
*              c(o)
*             /   \
*    c' --> A(x)   G(o)     (A/G could NOT be EMPTY)
*           /  \
*         p(o)  D(o)         (n(k) + 1 = n(B) + 1? = n(D) = n(G))
*         /  \
*     k(o)   B
*           / \
*          E   F
*/

/* Case B.RR (II.1.2)
* k' --> root(o)
*
*           c(?)         (c could NOT be EMPTY, n(k) + 1 = n(c))
*          /    \
*       p(o)   G(o)      (n(k) + 1 = n(A) = n(G))
*       /  \
*    k(o) A(x)
*         / \
*        B   D
*       / \
*      E   F
*/

/* Case B.?R (II.2)
* k' --> root(o)
```

```
  *
  *           c(?)        (c could NOT be EMPTY, n(k) + 1 = n(c))
  *         /    \
  *       p(o)   G(o)     (n(k) + 1 = n(A) + 1? = n(G))
  *       /  \
  *    k(o)  A
  *          / \
  *         B   D
  *        / \
  *       E   F
  */

  /* mirrored cases */
}

/* Assumptions:
* assumes key (k) and cursor (c) exist otherwise some
* tree invariants could break.
* p = keyParent
* c = cursor
* L = cursorLeft
*
* Case 1. if k =  then self.root becomes
* Case 2. if (p, k, c, L) = (p, k, , ) then tree graph breaks
* Case 3. if (p, k, c, L) = (, k, , ) then self.root becomes
*/
function rotateLeft(Tree storage self, uint64 key) private {
  uint64 cursor = self.nodes[key].right;
  uint64 keyParent = self.nodes[key].parent;
  uint64 cursorLeft = self.nodes[cursor].left;
  self.nodes[key].right = cursorLeft;
  if (cursorLeft != EMPTY) {
    self.nodes[cursorLeft].parent = key;
  }
  self.nodes[cursor].parent = keyParent;
  if (keyParent == EMPTY) {
    self.root = cursor;
  } else if (key == self.nodes[keyParent].left) {
    self.nodes[keyParent].left = cursor;
  } else {
    self.nodes[keyParent].right = cursor;
  }
  self.nodes[cursor].left = key;
  self.nodes[key].parent = cursor;
}

/* Assumptions:
* assumes key (k) and cursor (c) exist otherwise some
* tree invariants could break.
* p = keyParent
* c = cursor
* R = cursorRight
*
* Case 1. if k =  then self.root becomes
* Case 2. if (p, k, c, R) = (p, k, , ) then tree graph breaks
* Case 3. if (p, k, c, R) = (, k, , ) then self.root becomes
*/
function rotateRight(Tree storage self, uint64 key) private {
  uint64 cursor = self.nodes[key].left;
  uint64 keyParent = self.nodes[key].parent;
  uint64 cursorRight = self.nodes[cursor].right;
  self.nodes[key].left = cursorRight;
  if (cursorRight != EMPTY) {
    self.nodes[cursorRight].parent = key;
  }
  self.nodes[cursor].parent = keyParent;
  if (keyParent == EMPTY) {
    self.root = cursor;
  } else if (key == self.nodes[keyParent].right) {
    self.nodes[keyParent].right = cursor;
  } else {
    self.nodes[keyParent].left = cursor;
  }
  self.nodes[cursor].right = key;
  self.nodes[key].parent = cursor;
}
```

Paintswap: Acknowledged.

Cantina Managed: Acknowledged.

3.6.2 Changes made to `BokkyPooBahsRedBlackTreeLibrary`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Here are the summary of changes that are made to the original implementation of `BokkyPooBahsRedBlackTreeLibrary`:

1. `uint $\rightarrow$ uint64`.
2. A new field `tombstoneOffset` has been added to the `Node` struct.
3. Custom errors have been added (`KeyCannotBeZero` and `KeyDoesntExist`).
4. `solc` pragma has been changed from `^0.6.0` to `^0.8.29`.
5. `getNode` has been modified to return the node itself instead of individual components.
6. `edit` function has been added to increment the `tombstoneOffset` of an existing node in the tree.
7. `insert` and `remove` have been slightly modified.

Here are the diff of the main changes:

```
  diff --git a/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
  ↪  b/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
  index aae199e..07bb4bc 100644
-  -- a/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
+  ++ b/contracts/libraries/BokkyPooBahsRedBlackTreeLibrary.sol
  @@ -22,6 +22,7 @@ library BokkyPooBahsRedBlackTreeLibrary {
       uint64 parent;
       uint64 left;
       uint64 right;
+      uint56 tombstoneOffset; // Number of deleted segments, for gas efficiency
       bool red;
     }

  @@ -87,17 +88,19 @@ library BokkyPooBahsRedBlackTreeLibrary {
    function getEmpty() internal pure returns (uint) {
      return EMPTY;
    }
-    function getNode(
-      Tree storage self,
-      uint64 key
-    ) internal view returns (uint64 _returnKey, uint64 _parent, uint64 _left, uint64 _right, bool _red) {
-      require(exists(self, key));
-      return (key, self.nodes[key].parent, self.nodes[key].left, self.nodes[key].right, self.nodes[key].red);
+
+    function getNode(Tree storage self, uint64 key) internal view returns (Node storage node) {
+      require(exists(self, key), KeyDoesntExist());
+      return self.nodes[key];
+    }
+
+    function edit(Tree storage self, uint64 key, uint56 extraTombstoneOffset) internal {
+      require(exists(self, key), KeyDoesntExist());
+      self.nodes[key].tombstoneOffset += extraTombstoneOffset;
    }

    function insert(Tree storage self, uint64 key) internal {
      require(key != EMPTY, KeyCannotBeZero());
-      require(!exists(self, key));
      uint64 cursor = EMPTY;
      uint64 probe = self.root;
      while (probe != EMPTY) {
  @@ -112,7 +115,8 @@ library BokkyPooBahsRedBlackTreeLibrary {
        parent: cursor,
        left: EMPTY,
        right: EMPTY,
-        red: true
+        red: true,
+        tombstoneOffset: self.nodes[key].tombstoneOffset
```

```
      });
      if (cursor == EMPTY) {
        self.root = key;
@@ -166,7 +170,8 @@ library BokkyPooBahsRedBlackTreeLibrary {
      if (doFixup) {
        removeFixup(self, probe);
      }
-     delete self.nodes[cursor];
+     // Don't delete the node, so that we can re-use the tombstone offset if re-adding this price
+     self.nodes[cursor].parent = EMPTY;
    }

    function treeMinimum(Tree storage self, uint64 key) private view returns (uint64) {
```

Important notes:

- `insert`:
  1. Upon insertion of a node `n` its previous `tombstoneOffset` gets loaded back in it. |.
  2. Checking whether the node `n` already exists in the tree `T` is skipped/removed. This is dealt with higher up now in `_addToBookSide` (1) and `_addToBookSide` (2) where the following pattern is used:

     ```
     if (!tree.exists(price)) {
         tree.insert(price);
         // ...
     ```

- `remove`: `self.nodes[cursor]` does not get completely deleted but only its parents is set to the `EMPTY` node. Thus it can still retain its left/right child, color, and `tombstoneOffset`. Extra care needs to be taken to make sure this disconnected/removed node is not queried. calls to `exists(T, n)` are in place for `getNode`, `edit` and `remove` which would disallow `n` to be this removed node (unless the removed node ends up being a root after removal, but this case can be proven to not ever happen). *Although, the functions `next` and `prev` can be called on this removed node, but their currently not used in the code base.* Upon re-insertion among the retained left/right child, color, and `tombstoneOffset` values of the removed node, only `tombstoneOffset` and the others are set to their own default values.

Paintswap: Acknowledged.

Cantina Managed: Acknowledged.

3.6.3 Typos, comments, minor recommendations,

Severity: Informational

Context: SonicOrderBook.sol

Description/Recommendation:

- Typos:
  - SonicOrderBook.sol#L41: `maxOrdesrsPerPrice` → `maxOrdersPerPrice`.
  - SonicOrderBook.sol#L400: `the/` → `the`.
  - SonicOrderBook.sol#L355: `Convience` → `Convenience`.
- Minor Recommendations:
  - BokkyPooBahsRedBlackTreeLibrary.sol#L88: `uint` → `uint64`.
  - SonicOrderBook.sol#L31: This line can be removed:

    ```
    - using BokkyPooBahsRedBlackTreeLibrary for BokkyPooBahsRedBlackTreeLibrary.Node;
    ```

    `BokkyPooBahsRedBlackTreeLibrary` does not have any functions with the first input variable as `Node`.
  - SonicOrderBook.sol#L228: `limitOrders` is defined as a `public` function although it is not used within the same contract. This could be due to gas optimisation, if not it can be made `external`.
  - SonicOrderBook.sol#L823: Rename `remainingSegment` → `slotData`.

- SonicOrderBook.sol#L522: `priceTick` could have been queried internally in `_makeLimitOrder` instead of passing it as an argument.
- SonicOrderBook.sol: Define a getter function `getSegments`. It might be helpful to debug issues.

```
function getSegments(OrderSide side, uint256 tokenId, uint64 price) external view returns
↪  (bytes32[] memory) {
  if (side == OrderSide.Buy) {
    return _bidsAtPrice[tokenId][price];
  } else {
    return _asksAtPrice[tokenId][price];
  }
}
```

- SonicOrderBook.sol#L666, SonicOrderBook.sol#L838, SonicOrderBook.sol#L1087, SonicOrderBook.sol#L1117: Inconsistent unpacking of `orderId`:

```
orderId = uint48(data); // method 1
orderId = uint48(data & ORDER_MASK); // method 2
```

- SonicOrderBook.sol#L541: Instead of using `orders[i]` one can use the already defined parameter `order`
- SonicOrderBook.sol#L1045, SonicOrderBook.sol#L1054: Un-safe casts from `uint128` to `int128`.
- SonicOrderBook.sol#L857: The next line when `tree.getNode(price)` is called, the existence of the `price` key in the `tree` is checked. So the check on this line is redundant.

```
- require(tree.exists(price), OrderNotFoundInTree(orderId, price));
```

- SonicOrderBook.sol#L1134-L1135: At this point, `nextSegmentIndex` would be `index` and `nextOffsetIndex` would be `slot` since we can prove that `slot` is less than `NUM_SLOTS_PER_SEGMENT`.
- SonicOrderBook.sol#L303-L304: Unlike `_claimNFTs`, `_claimTokens` is missing the following check:

```
require(orderIds.length != 0, NothingToClaim());
```

  It would be best if both used the same patterns.

- SonicOrderBook.sol#L952-L956: Define a new constant that only masks the last possible slot in a segment:

```
uint256 private constant LAST_SLOT_MASK = SEGMENT_MASK << ((NUM_SLOTS_PER_SEGMENT - 1) *
↪  SEGMENT_SIZE);
// ...

/**
* @dev Retrieves the last slot of a specific price level in the market.
*
* @param segmentsPriceMap A mapping from each price to its segments, where each segment is an
↪  array of bytes32 representing the slots at that price level.
* @param price The price level for which to retrieve the last slot.
*
* @return uint256 The last slot of the specified price level in the market (it is only masked
↪  off but not shifted down).
*/
function _getLastSlot(
  mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
  uint64 price
) internal view returns (uint256) {
  bytes32 lastSegment = segmentsPriceMap[price][segmentsPriceMap[price].length - 1];
  return uint256(lastSegment) & LAST_SLOT_MASK;
}
```

  All these operations regarding segment and slots can be moved to a utility library. For example the above suggestion also applies to SonicOrderBook.sol#L974-L977:

```
diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
index b3bb01a..43e1506 100644
--- a/contracts/SonicOrderBook.sol
+++ b/contracts/SonicOrderBook.sol
@@ -949,11 +949,7 @@ contract SonicOrderBook is
      } else {
        uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;
        // Check if this would go over the max number of orders allowed at this price level
-       bool lastSegmentFilled = (
-         (uint256(
-           segmentsPriceMap[price][segmentsPriceMap[price].length - 1] >>
↪  ((NUM_SLOTS_PER_SEGMENT - 1) * SEGMENT_SIZE)
-         ) & SEGMENT_MASK)
-       ) != 0;
+       bool lastSegmentFilled = _getLastSlot(segmentsPriceMap, price) != 0;

        // Check if last segment is full
        if (
@@ -970,13 +966,7 @@ contract SonicOrderBook is
              break;
            } else if (
              (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT <
              ↪  _maxOrdersPerPrice ||
-             (
-               (uint256(
-                 segmentsPriceMap[price][segmentsPriceMap[price].length - 1] >>
-                   ((NUM_SLOTS_PER_SEGMENT - 1) * SEGMENT_SIZE)
-               ) & SEGMENT_MASK)
-             ) ==
-             0
+             _getLastSlot(segmentsPriceMap, price) == 0
            ) {
              // There are segments available at this price level or the last segment is not
              ↪  filled yet
              break;
```

Paintswap: Fixed in PR 44.

Cantina Managed: Most of the mentioned findings have been fixed in the PR 44 except the following:

- SonicOrderBook.sol#L823: rename `remainingSegment` → `slotData`.
- SonicOrderBook.sol#L522: `priceTick` could have been queried internally in `_makeLimitOrder` instead of passing it as an argument.
- SonicOrderBook.sol: Define a getter function `getSegments`. It might be helpful to debug issues.
- SonicOrderBook.sol#L666, SonicOrderBook.sol#L838, SonicOrderBook.sol#L1087, SonicOrderBook.sol#L1117: inconsistent unpacking of `orderId`:

```
orderId = uint48(data); // method 1
orderId = uint48(data & ORDER_MASK); // method 2
```

- SonicOrderBook.sol#L1134-L1135: At this point, `nextSegmentIndex` would be `index` and `nextOffsetIndex` would be `slot` since we can prove that `slot` is less than `NUM_SLOTS_PER_SEGMENT`.
- SonicOrderBook.sol#L952-L956: Define a new constant that only masks the last possible slot in a segment.

3.6.4 The too many orders hit check is incorrect

Severity: Informational

Context: SonicOrderBook.sol#L713-L717

Description: `orderIdsPool` and `quantitiesPool` are of length `MAX_ORDERS_HIT`.

```
uint48[] memory orderIdsPool = new uint48[](MAX_ORDERS_HIT);
uint256[] memory quantitiesPool = new uint256[](MAX_ORDERS_HIT);
```

The first time the check below fails:

```
require(numberOfOrders < MAX_ORDERS_HIT, TooManyOrdersHit());
```

We should have `numberOfOrders == MAX_ORDERS_HIT` which means before `numberOfOrders` getting updated it was `MAX_ORDERS_HIT - 1` which is an allowed index to populate `orderIdsPool` and `quantitiesPool`:

```
orderIdsPool[MAX_ORDERS_HIT - 1] = orderId;
quantitiesPool[MAX_ORDERS_HIT - 1] = quantityNFTClaimable;
numberOfOrders = MAX_ORDERS_HIT - 1 + 1;

require(MAX_ORDERS_HIT < MAX_ORDERS_HIT, TooManyOrdersHit()); // reverts
```

Recommendation: Change the check to (non-strict inequality):

```
require(numberOfOrders <= MAX_ORDERS_HIT, TooManyOrdersHit());
```

To allow the last elements of `orderIdsPool` and `quantitiesPool` to be populated. Moreover the `require` statement could be moved higher up inside the `for` loop so that after the last element is populated, if one tries to compute more values for an extra element, `revert` happens earlier (this would just be a gas optimisation for an unwanted case).

Paintswap: Fixed in PR 28. Moved the check higher.

Cantina Managed: Fix verified.

3.6.5 `_claimNFTs` assumes the `_nft` would revert on `0` token id transfers

Severity: Informational

Context: SonicOrderBook.sol#L331-L353

Description: Currently, the `0` token id in the order book represent quote tokens and not airdrop locked `NFT` token id. `_claimNFTs` does not check whether the `claimableTokenInfo.tokenId` is `0` or not and relies on the assumption that calling `_nft` would revert on `0` token id transfers. This is true in the current implementation as one cannot mint a non-zero quantity for locked airdrop `NFT` for `0` (season `None`. This is prevented by a series of checks in the `SonicAirdropNFT` contract.

Recommendation: It is best to `revert` in `SonicOrderBook` directly and not rely on the series of assumptions from `SonicAirdropNFT`. Besides the important checks added in the following patch a few minor adjustment is also made:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index b3bb01a..012a6b4 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -67,6 +67,7 @@ contract SonicOrderBook is
     error SeasonLocked();
     error InvalidQuantity();
     error InvalidMinQuantity();
+    error CannotClaimZeroNFTTokenId();

     enum OrderSide {
       Buy,
  @@ -334,16 +335,20 @@ contract SonicOrderBook is

       uint256[] memory nftAmountsFromUs = new uint256[](orderIds.length);
       uint256[] memory tokenIds = new uint256[](orderIds.length);
-      for (uint256 i = 0; i < tokenIds.length; ++i) {
+      for (uint256 i = 0; i < orderIds.length; ++i) {
         uint48 orderId = orderIds[i];
         ClaimableTokenInfo storage claimableTokenInfo = _tokenClaimables[orderId];
+        require(claimableTokenInfo.maker == sender, NotMaker());
+
         uint256 tokenId = claimableTokenInfo.tokenId;
-        tokenIds[i] = tokenId;
+        require(tokenId != 0, CannotClaimZeroNFTTokenId()); // <-- new check added
+
         uint256 claimableAmount = claimableTokenInfo.amount;
         require(claimableAmount != 0, NothingToClaim());
-        require(_tokenClaimables[orderId].maker == sender, NotMaker());

+        tokenIds[i] = tokenId;
         nftAmountsFromUs[i] = claimableAmount;
+
         claimableTokenInfo.amount = 0;
       }
```

Paintswap: Fixed in PR 34.

Cantina Managed: Fix verified.

3.6.6 The owner can indirectly change the `priceTick` of a `tokenId`

Severity: Informational

Context: SonicOrderBook.sol#L1280-L1281

Description: The owner can indirectly change the `priceTick` of a `tokenId` that has already been set:

1. Change the `priceTick` to `0`.
2. Change it back to a non-zero value.

This also has been noticed in the current test cases:

```
it("Tick change constraints", async function () {
  const {orderBook, tokenId, priceTick, minQuantity} = await loadFixture(deployContractsFixture);
  // Cannot be changed if set to a new one
  await expect(
    orderBook.setTokenIdInfos([tokenId], [{priceTick: priceTick + 1n, minQuantity, isTradeable: true}])
  ).to.be.revertedWithCustomError(orderBook, "TickCannotBeChanged");
  // Can be set to 0 to remove it from the book.
  await expect(orderBook.setTokenIdInfos([tokenId], [{priceTick: 0, minQuantity, isTradeable: true}])).to.not.be
    .reverted;

  // And then can be changed from 0 (although not recommended unless it's back to the original one!)
  await expect(orderBook.setTokenIdInfos([tokenId], [{priceTick: priceTick, minQuantity, isTradeable:
  ↪  true}])).to.not
    .be.reverted;
});
```

Changing the `priceTick` can create the following issues:

If the grids created by the old and new `priceTick`s don't match:

1. Then the newly inserted limit orders might fall in a direction that would make it match before the older limit orders in the same direction if:
    1. The chosen price node by the user is already full and...
    2. The new price grid is a finer grid.
2. Then the newly inserted limit orders might fall in a direction that would make it match later than expected if:
    1. The chosen price node by the user is already full and...
    2. With the older price tick, the order would have been inserted in a direction that would have been matched earlier.

Recommendation: Add a warning to the NatSpec for `setTokenIdInfos` mentioning this issue or enforce that if `existingTick != 0` then `existingTick == newTick` (ie only allowing `minQuantity` and `isTradeable` to be changed for a `tokenId` with a non-zero `existingTick`).

Paintswap: Fixed in PR 36. Have removed the ability to set the price tick to 0 now to prevent this issue from occurring. It was being used as a check that the token id exists, but we can just rely on the existing `isTradeable` flag for that.

Cantina Managed: Fix verified.

3.6.7 `_addToBookSide` can be simplified

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `_addToBookSide` can be simplified by defining new `private` and `internal` functions. Moreover these functions can be grouped into utility libraries.

Recommendation: New implementation:

```
/**
* @dev Returns the used capacity of a specific price level in the market.
* The used capacity is determined by the number of segments at that price level,
* minus the number of tombstones (inactive slots), and then multiplied by the number of slots per segment.
*
* @param tree The Red-Black Tree containing the prices and their associated nodes.
* @param segmentsPriceMap A mapping from each price to its segments, where each segment is an array of bytes32
↪  representing the slots at that price level.
* @param price The price level for which to retrieve the used capacity.
*
* @return uint256 The used capacity of the specified price level in the market.
*/
function _getPriceLevelUsedCapacity(
  BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
  mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
  uint64 price
) internal view returns (uint256) {
  // below already checks whether the price exists in the tree
  uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;
  return (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT;
}

/**
* @dev Assumes that `segments.length > 0`.
*/
function _lastSegmentLastSlot(bytes32[] storage segments) private view returns (uint256) {
  return (uint256(segments[segments.length - 1]) >> ((NUM_SLOTS_PER_SEGMENT - 1) * SEGMENT_SIZE)) &
  ↪  SEGMENT_MASK;
}

function _segmentsHasSpaceForMoreOrders(
  BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
  mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
  uint64 price
) private returns (bool) {
  if (!tree.exists(price)) {
    tree.insert(price);
    return true;
```

```
  }

  return (
    _getPriceLevelUsedCapacity(tree, segmentsPriceMap, price) < _maxOrdersPerPrice ||
  _lastSegmentLastSlot(segmentsPriceMap[price]) == 0
  );
}

function _nextPrice(
  uint64 price,
  int128 priceTick // - for buy, + for sell
) private pure returns (uint64) {
  unchecked {
    return SafeCast.toUint64(uint128(int128(uint128(price)) + priceTick));
  }
}

/**
* @dev Assumes `orderId` and `normalizedQuantity` do NOT have dirty upper bits.
*/
function _packOrderIdAndNormalizedQuantity(
  uint256 orderId,
  uint256 normalizedQuantity
) private pure returns (bytes32 packed) {
  assembly ("memory-safe") {
    packed := or(shl(ORDER_ID_SIZE, normalizedQuantity), orderId)
  }
}

function _addOrderToEndOfSegments(
  bytes32[] storage segments,
  uint256 orderId,
  uint256 normalizedQuantity
) private {
  bytes32 packed = _packOrderIdAndNormalizedQuantity(orderId, normalizedQuantity);
  // Read last one. Decide if we can add to the existing segment or if we need to add a new segment
  if (segments.length != 0) {
    bytes32 lastSegment = segments[segments.length - 1];
    // Are there are free entries in this segment
    for (uint256 slotShiftAmount = 0; slotShiftAmount < NUM_SLOTS_PER_SEGMENT * SEGMENT_SIZE; slotShiftAmount
    ↪  += SEGMENT_SIZE) {
      uint256 remainingSegment = uint256(lastSegment) >> slotShiftAmount;
      if (remainingSegment == 0) {
        // Found free entry one, so add to an existing segment
        segments[segments.length - 1] = (packed << slotShiftAmount) | lastSegment;
        return;
      }
    }
  }

  segments.push(packed);
}



function _addToBookSide(
  mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
  BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
  uint64 price,
  uint256 orderId,
  uint256 normalizedQuantity,
  int128 priceTick // - for buy, + for sell
) private returns (uint64) {
  // Loop until we find a suitable price level to insert the order
  while (!_segmentsHasSpaceForMoreOrders(tree, segmentsPriceMap, price)) {
    price = _nextPrice(price, priceTick);
  }

  _addOrderToEndOfSegments(segmentsPriceMap[price], orderId, normalizedQuantity);

  return price;
}
```

Copy the patch below to a file an apply it with `git apply path/to/patch/file`:

```
diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
```

```
  index b3bb01a..3d1a491 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -935,81 +935,111 @@ contract SonicOrderBook is
       }
     }

-    function _addToBookSide(
+    /**
+     * @dev Returns the used capacity of a specific price level in the market.
+     * The used capacity is determined by the number of segments at that price level,
+     * minus the number of tombstones (inactive slots), and then multiplied by the number of slots per segment.
+     *
+     * @param tree The Red-Black Tree containing the prices and their associated nodes.
+     * @param segmentsPriceMap A mapping from each price to its segments, where each segment is an array of
↪  bytes32 representing the slots at that price level.
+     * @param price The price level for which to retrieve the used capacity.
+     *
+     * @return uint256 The used capacity of the specified price level in the market.
+     */
+    function _getPriceLevelUsedCapacity(
+      BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
       mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
+      uint64 price
+    ) internal view returns (uint256) {
+      // below already checks whether the price exists in the tree
+      uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;
+      return (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT;
+    }
+
+    /**
+     * @dev Assumes that `segments.length > 0`.
+     */
+    function _lastSegmentLastSlot(bytes32[] storage segments) private view returns (uint256) {
+      return (uint256(segments[segments.length - 1]) >> ((NUM_SLOTS_PER_SEGMENT - 1) * SEGMENT_SIZE)) &
↪  SEGMENT_MASK;
+    }
+
+    function _segmentsHasSpaceForMoreOrders(
       BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
-      uint64 price,
-      uint256 orderId,
-      uint256 normalizedQuantity,
-      int128 priceTick // - for buy, + for sell
-    ) private returns (uint64) {
-      // Add to the bids section
+      mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
+      uint64 price
+    ) private returns (bool) {
       if (!tree.exists(price)) {
         tree.insert(price);
-      } else {
-        uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;
-        // Check if this would go over the max number of orders allowed at this price level
-        bool lastSegmentFilled = (
-          (uint256(
-            segmentsPriceMap[price][segmentsPriceMap[price].length - 1] >> ((NUM_SLOTS_PER_SEGMENT - 1) *
↪  SEGMENT_SIZE)
-          ) & SEGMENT_MASK)
-        ) != 0;
-
-        // Check if last segment is full
-        if (
-          (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT >= _maxOrdersPerPrice &&
-          lastSegmentFilled
-        ) {
-          // Loop until we find a suitable place to put this
-          while (true) {
-            unchecked {
-              price = SafeCast.toUint64(uint128(int128(uint128(price)) + priceTick));
-            }
-            if (!tree.exists(price)) {
-              tree.insert(price);
-              break;
-            } else if (
-              (segmentsPriceMap[price].length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT < _maxOrdersPerPrice
↪  ||
```

```
-              (
-                (uint256(
-                  segmentsPriceMap[price][segmentsPriceMap[price].length - 1] >>
-                    ((NUM_SLOTS_PER_SEGMENT - 1) * SEGMENT_SIZE)
-                ) & SEGMENT_MASK)
-              ) ==
-              0
-            ) {
-              // There are segments available at this price level or the last segment is not filled yet
-              break;
-            }
-          }
-        }
+        return true;
+      }
+
+      return (
+        _getPriceLevelUsedCapacity(tree, segmentsPriceMap, price) < _maxOrdersPerPrice ||
+      _lastSegmentLastSlot(segmentsPriceMap[price]) == 0
+      );
+    }
+
+    function _nextPrice(
+      uint64 price,
+      int128 priceTick // - for buy, + for sell
+    ) private pure returns (uint64) {
+      unchecked {
+        return SafeCast.toUint64(uint128(int128(uint128(price)) + priceTick));
       }
+    }

+    /**
+    * @dev Assumes `orderId` and `normalizedQuantity` do NOT have dirty upper bits.
+    */
+    function _packOrderIdAndNormalizedQuantity(
+      uint256 orderId,
+      uint256 normalizedQuantity
+    ) private pure returns (bytes32 packed) {
+      assembly ("memory-safe") {
+        packed := or(shl(ORDER_ID_SIZE, normalizedQuantity), orderId)
+      }
+    }
+
+    function _addOrderToEndOfSegments(
+      bytes32[] storage segments,
+      uint256 orderId,
+      uint256 normalizedQuantity
+    ) private {
+      bytes32 packed = _packOrderIdAndNormalizedQuantity(orderId, normalizedQuantity);
       // Read last one. Decide if we can add to the existing segment or if we need to add a new segment
-      bytes32[] storage segments = segmentsPriceMap[price];
-      bool pushToEnd = true;
       if (segments.length != 0) {
         bytes32 lastSegment = segments[segments.length - 1];
         // Are there are free entries in this segment
-        for (uint256 slot = 0; slot < NUM_SLOTS_PER_SEGMENT; ++slot) {
-          uint256 remainingSegment = ((uint256(lastSegment) >> (slot * SEGMENT_SIZE)) & SEGMENT_MASK);
+        for (uint256 slotShiftAmount = 0; slotShiftAmount < NUM_SLOTS_PER_SEGMENT * SEGMENT_SIZE;
↪  slotShiftAmount += SEGMENT_SIZE) {
+          uint256 remainingSegment = uint256(lastSegment) >> slotShiftAmount;
           if (remainingSegment == 0) {
             // Found free entry one, so add to an existing segment
-            bytes32 newSegment = lastSegment |
-              (bytes32(orderId) << (slot * SEGMENT_SIZE)) |
-              (bytes32(normalizedQuantity) << (slot * SEGMENT_SIZE + ORDER_ID_SIZE));
-            segments[segments.length - 1] = newSegment;
-            pushToEnd = false;
-            break;
+            segments[segments.length - 1] = (packed << slotShiftAmount) | lastSegment;
+            return;
           }
         }
       }

-      if (pushToEnd) {
-        bytes32 segment = bytes32(orderId) | (bytes32(normalizedQuantity) << ORDER_ID_SIZE);
```

```
-        segments.push(segment);
+      segments.push(packed);
+    }
+
+
+
+    function _addToBookSide(
+      mapping(uint256 price => bytes32[]) storage segmentsPriceMap,
+      BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
+      uint64 price,
+      uint256 orderId,
+      uint256 normalizedQuantity,
+      int128 priceTick // - for buy, + for sell
+    ) private returns (uint64) {
+      // Loop until we find a suitable price level to insert the order
+      while (!_segmentsHasSpaceForMoreOrders(tree, segmentsPriceMap, price)) {
+        price = _nextPrice(price, priceTick);
       }

+      _addOrderToEndOfSegments(segmentsPriceMap[price], orderId, normalizedQuantity);
+
       return price;
     }
```

Paintswap: Fixed in PR 37.

Cantina Managed: Fix verified.

3.6.8 `_allOrdersAtPriceSide` can be simplified

Severity: Informational

Context: SonicOrderBook.sol#L816-L844

Description: `_allOrdersAtPriceSide` can be simplified. Currently there is an asymmetry when calculating the slots not used. The current implementation only considers the very beginning at the segment pointed to by `tomestoneOffset` but does not consider the unused slots at the very last segment.

Recommendation: It would be best to simply the codebase to remove this asymmetry. Moreover a few optimisations have been applied in the provided patch. Copy and apply the following patch `git apply /path/to/patch/file`:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index b3bb01a..06e9808 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -813,33 +813,22 @@ contract SonicOrderBook is
       if (!tree.exists(price)) {
         return orders;
       }
-      BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNode(price);
-      uint256 tombstoneOffset = node.tombstoneOffset;

-      uint256 numInSegmentDeleted;
-      {
-        uint256 segment = uint256(segments[tombstoneOffset]);
-        for (uint256 slot; slot < NUM_SLOTS_PER_SEGMENT; ++slot) {
-          uint256 remainingSegment = (segment >> (slot * SEGMENT_SIZE)) & SEGMENT_MASK;
-          if (remainingSegment == 0) {
-            ++numInSegmentDeleted;
-          } else {
-            break;
-          }
-        }
-      }
+      uint256 tombstoneOffset = tree.getNode(price).tombstoneOffset;

-      orders = new Order[]((segments.length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT - numInSegmentDeleted);
+      orders = new Order[]((segments.length - tombstoneOffset) * NUM_SLOTS_PER_SEGMENT);
       uint256 numberOfEntries;
-      for (uint256 i = numInSegmentDeleted; i < orders.length + numInSegmentDeleted; ++i) {
-        uint256 segment = uint256(segments[i / NUM_SLOTS_PER_SEGMENT + tombstoneOffset]);
-        uint256 slot = i % NUM_SLOTS_PER_SEGMENT;
```

```
-        uint256 slotData = (segment >> (slot * SEGMENT_SIZE)) & SEGMENT_MASK;
-        uint48 id = uint48(slotData & ORDER_MASK);
-        if (id != 0) {
-          uint32 normalizedQuantity = uint32((slotData >> ORDER_ID_SIZE) & QUANTITY_MASK);
-          orders[numberOfEntries] = Order(_tokenClaimables[id].maker, uint88(normalizedQuantity *
↪  QUANTITY_TICK), id);
-          ++numberOfEntries;
+
+      for(uint256 segmentIdx = tombstoneOffset; segmentIdx < segments.length; ++segmentIdx) {
+        uint256 segment = uint256(segments[segmentIdx]);
+        for (uint256 slotShiftAmount = 0; slotShiftAmount < NUM_SLOTS_PER_SEGMENT * SEGMENT_SIZE;
↪  slotShiftAmount += SEGMENT_SIZE) {
+          uint256 slotData = (segment >> slotShiftAmount) & SEGMENT_MASK;
+          uint48 id = uint48(slotData & ORDER_MASK);
+          if (id != 0) {
+            uint32 normalizedQuantity = uint32((slotData >> ORDER_ID_SIZE) & QUANTITY_MASK);
+            orders[numberOfEntries] = Order(_tokenClaimables[id].maker, uint88(normalizedQuantity *
↪  QUANTITY_TICK), id);
+            ++numberOfEntries;
+          }
        }
      }
```

Paintswap: Fixed in PR 40.

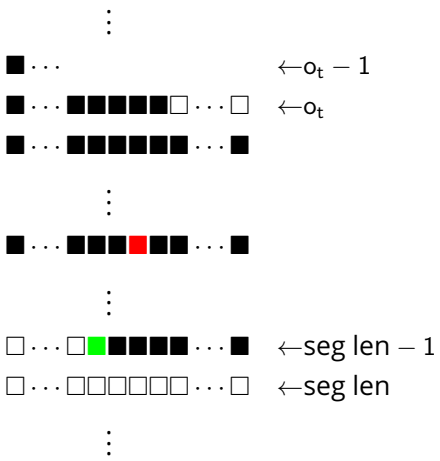
Cantina Managed: Fix verified.

3.6.9 Invariants of Price Segments

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description:



- □ represents empty orders with 0 order id and normalized quantity.
- ■ represent orders where both the order id and the normalized quantity are non-zero.

Invariants:

1. It is very important that empty orders that are not tombstonned, ie □, do not have a corresponding maker. `_tokenClaimables[0]` should always be empty. This would guarantee that empty orders □ cannot be cancelled. Otherwise canceling a possible empty order on the segment pointed to by o_t could cause the last segment to be popped out.
2. The following configuration in the active area of the segments when all orders are drawn on a line should also not be allowed $\cdots$■□■$\cdots$, it should be proved that this cannot happen.
3. If an order has an order id 0 then its normalized quantity should also be 0 (is not allowed) and vice versa (is not allowed).
4. Orders are inserted/added at the end.

5. Orders are matched based on their order id. The smaller order ids get matched first (FIFO).
6. The last segment of an existing price node has at least one non-empty order:

$$\cdots\blacksquare\cdots \leftarrow \text{seg len} - 1$$

   For the case of 3 slots per segment the following cases are possible:

$$\blacksquare\blacksquare\blacksquare \leftarrow \text{seg len} - 1$$
$$\blacksquare\blacksquare\square \leftarrow \text{seg len} - 1 = o_t$$
$$\blacksquare\square\square \leftarrow \text{seg len} - 1 = o_t$$
$$\square\blacksquare\blacksquare \leftarrow \text{seg len} - 1$$
$$\square\blacksquare\square \leftarrow \text{seg len} - 1 = o_t$$
$$\square\square\blacksquare \leftarrow \text{seg len} - 1$$

7. The segment pointed to by o_t (`tombstoneOffset`) if the price node exists in the tree, then this segment would have at least one non-empty order:

$$\blacksquare\blacksquare\blacksquare \leftarrow o_t$$
$$\blacksquare\blacksquare\square \leftarrow o_t$$
$$\blacksquare\square\square \leftarrow o_t$$
$$\square\blacksquare\blacksquare \leftarrow o_t = \text{seg len} - 1$$
$$\square\blacksquare\square \leftarrow o_t = \text{seg len} - 1$$
$$\square\square\blacksquare \leftarrow o_t = \text{seg len} - 1$$

8. For price nodes that do not belong to the tree T, we should have the following configuration:
   1. The following:

$$\square\cdots\square \leftarrow o_t = \text{seg len}$$

      i.e. all the orders belonging to the segment pointed to by o_t, they should all be empty orders.
   2. In some edge cases when matching orders we could end up in states where:

$$\square\cdots\square \leftarrow o_t = \text{seg len} - 1$$

      ie. the last empty segment is not popped out. This can happen if before matching the last segement looked like one of the below configurations and the order would fill the whole price segments:

$$\square\blacksquare\blacksquare \leftarrow \text{seg len} - 1$$
$$\square\blacksquare\square \leftarrow \text{seg len} - 1$$
$$\square\square\blacksquare \leftarrow \text{seg len} - 1$$

      The important line is SonicOrderBook.sol#L750. As an example at this point SonicOrderBook.ts#L4843 in the `SonicOrderBook > System test ...` the `_bidsAtPrice[tokenId][price]` looks like:

```
0000000004b00000000000180000032000000000001300000000000000000000
000000000000000000000000000000000000000000000000000000000000000000  o_t, seg.len -1
```

9. Either $o_t = 0$ or if $o_t > 0$ we should have that the last order in the segment pointed to by $o_t - 1$ is a non-empty order:

$$\blacksquare\blacksquare\blacksquare \leftarrow o_t - 1$$
$$\blacksquare\blacksquare\square \leftarrow o_t - 1$$
$$\blacksquare\square\square \leftarrow o_t - 1$$
$$\blacksquare\square\blacksquare \leftarrow o_t - 1$$

case 9.4 should not be possible (Need to double check, but it would not break the following invariant). But overall it is important that the following cases are not possible, otherwise for non-existing price node with the below configuration the next added order would get inserted in the tombstoned area and would not ever get matched:

$$\square\square\square \leftarrow o_t - 1$$
$$\square\blacksquare\blacksquare \leftarrow o_t - 1$$
$$\square\blacksquare\square \leftarrow o_t - 1$$
$$\square\square\blacksquare \leftarrow o_t - 1$$

10. Orders belonging to segments pointed to by $k \geq$ seg len should all be empty orders (unless there is a storage collision if k is a really big number). This is a more general invariant compared to Inv 8.:

$$\square \cdots \square \leftarrow k \geq \text{seg len}$$

method	Inv 1.	Inv 2.	Inv 3.	Inv 4.	Inv 5.	Inv 6.	Inv 7.	Inv. 8	Inv. 9	Inv 10.
canceling orders	✓	✓	✓	N/A	✓	✓	✓	✓	✓	✓
placing limit orders	✓	×	✓	×	×	✓	✓	✓	increase	✓
matching orders	✓	?	✓	N/A	✓	✓	✓	✓ if Inv 10 is true	✓	✓

The ✓ above means that if the invariant was satisfied before calling the method, it will be satisfied after the call has been done (in some edge cases it would only mean the invariant is satisfied after the call has been done, but it might not be before the call for example at the very beginning).

The invariants broken × for *placing limit orders* can be fixed by fixing "A new limit order might get added before an older limit order" for Inv 9. means that although the invariant would be satisfied if it was satisfied before, it would still be possible to place orders in the inactive tombstoned area, but again fixing "A new limit order might get added before an older limit order" would fix this issue.

Recommendation: Applying the following patch for matching order follows makes it easier to verify the invariants for this flow:

```
  diff --git a/contracts/SonicOrderBook.sol b/contracts/SonicOrderBook.sol
  index b3bb01a..8246632 100644
-  -- a/contracts/SonicOrderBook.sol
+  ++ b/contracts/SonicOrderBook.sol
  @@ -639,7 +639,7 @@ contract SonicOrderBook is
       }

       bool isTakingFromBuy = side == OrderSide.Buy;
-      uint256 numberOfOrders;
+      uint256 numberOfOrders = 0;
       while (quantityRemaining != 0) {
         uint64 bestPrice = isTakingFromBuy ? getHighestBid(tokenId) : getLowestAsk(tokenId);
         if (bestPrice == 0 || (isTakingFromBuy ? bestPrice < price : bestPrice > price)) {
  @@ -647,125 +647,190 @@ contract SonicOrderBook is
           break;
         }

-        // Loop through all at this order
-        uint256 numSegmentsFullyConsumed = 0;
```

```
-        bytes32[] storage segments = segmentsAtPrice[tokenId][bestPrice];
-        BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree[tokenId].getNode(bestPrice);
-
-        bool eatIntoLastOrder;
-        uint256 numOrdersWithinLastSegmentFullyConsumed;
-        bytes32 segment;
-        uint256 lastSegment;
-        for (uint256 i = node.tombstoneOffset; i < segments.length && quantityRemaining != 0; ++i) {
-          lastSegment = i;
-          segment = segments[i];
-          uint256 numOrdersWithinSegmentConsumed;
-          bool wholeSegmentConsumed;
-          for (uint256 slot; slot < NUM_SLOTS_PER_SEGMENT && quantityRemaining != 0; ++slot) {
-            uint256 remainingSegment = uint256(segment >> (slot * SEGMENT_SIZE));
-            uint48 orderId = uint48(remainingSegment);
-            if (orderId == 0) {
-              // Check if there are any order left in this segment
-              if (remainingSegment != 0) {
-                // Skip this order in the segment as it's been deleted
-                ++numOrdersWithinLastSegmentFullyConsumed;
-                continue;
-              } else {
-                break;
-              }
-            }
+        uint256 currentCost = 0;
+        (quantityRemaining, currentCost, numberOfOrders) = _consumePriceSegments(
+          tree[tokenId],
+          segmentsAtPrice[tokenId][bestPrice],
+          bestPrice,
+          isTakingFromBuy,
+          quantityRemaining,
+          orderIdsPool,
+          quantitiesPool,
+          numberOfOrders
+        );

-            uint32 storedNormalizedQuantity = uint32(uint256(segment >> (slot * SEGMENT_SIZE +
↪  ORDER_ID_SIZE)));
-            uint256 quantityL3 = storedNormalizedQuantity * QUANTITY_TICK;
-            uint256 quantityNFTClaimable;
-            if (quantityRemaining >= quantityL3) {
-              // Consume this whole order
-              unchecked {
-                quantityRemaining -= quantityL3;
-
-                // Is the last one in the segment being fully consumed?
-                wholeSegmentConsumed = slot == (NUM_SLOTS_PER_SEGMENT - 1);
-              }
-              ++numOrdersWithinSegmentConsumed;
-              quantityNFTClaimable = quantityL3;
-            } else {
-              // Eat into the order
-              uint256 newNormalizedQuantity = (quantityL3 - quantityRemaining) / QUANTITY_TICK;
-              segment = bytes32(
-                (uint256(segment) & ~(uint256(0xffffff) << (slot * SEGMENT_SIZE + ORDER_ID_SIZE))) |
-                  (newNormalizedQuantity << (slot * SEGMENT_SIZE + ORDER_ID_SIZE))
-              );
-              quantityNFTClaimable = quantityRemaining;
-              quantityRemaining = 0;
-              eatIntoLastOrder = true;
-            }
-            uint256 tradeCost = (quantityNFTClaimable * bestPrice) / BASE_DECIMAL_DIVISOR;
-            cost += tradeCost;
-
-            ClaimableTokenInfo storage claimableTokenInfo = _tokenClaimables[orderId];
-            if (isTakingFromBuy) {
-              claimableTokenInfo.amount += uint88(quantityNFTClaimable);
-            } else {
-              uint256 fees = _calcFees(tradeCost);
-              claimableTokenInfo.amount += uint88(tradeCost - fees);
-            }
+        cost += currentCost;
+      }

-            orderIdsPool[numberOfOrders] = orderId;
```

```
-            quantitiesPool[numberOfOrders] = quantityNFTClaimable;
-            ++numberOfOrders;
+      if (numberOfOrders != 0) {
+        assembly ("memory-safe") {
+          mstore(orderIdsPool, numberOfOrders)
+          mstore(quantitiesPool, numberOfOrders)
+        }

-            require(numberOfOrders < MAX_ORDERS_HIT, TooManyOrdersHit());
+        emit OrdersMatched(sender, orderIdsPool, quantitiesPool);
+      }
+    }
+
+    function _consumePriceSegments(
+      BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
+      bytes32[] storage segments,
+      uint64 bestPrice,
+      bool isTakingFromBuy,
+      uint256 quantityRemaining,
+      uint48[] memory orderIdsPool,
+      uint256[] memory quantitiesPool,
+      uint256 numberOfOrders
+    ) private returns (
+      uint256 newQunatityRemaining,
+      uint256 cost,
+      uint256 newNumberOfOrders
+    ) {
+      BokkyPooBahsRedBlackTreeLibrary.Node storage node = tree.getNode(bestPrice);
+
+      uint256 numOrdersFullyConsumed = 0;
+      bytes32 segment;
+      uint256 lastConsumedSegmentIndex;
+
+      for (uint256 i = node.tombstoneOffset; i < segments.length && quantityRemaining != 0; ++i) {
+        lastConsumedSegmentIndex = i;
+        segment = segments[i];
+
+        for (uint256 slot = 0; slot < NUM_SLOTS_PER_SEGMENT && quantityRemaining != 0; ++slot) {
+          uint256 remainingSegment = uint256(segment >> (slot * SEGMENT_SIZE));
+          if (remainingSegment == 0) {
+            // only should hit this branch on the last segment
+            break; // No more orders in this segment
           }

-          if (wholeSegmentConsumed) {
-            ++numSegmentsFullyConsumed;
-            numOrdersWithinLastSegmentFullyConsumed = 0;
-          } else {
-            numOrdersWithinLastSegmentFullyConsumed += numOrdersWithinSegmentConsumed;
-            if (eatIntoLastOrder) {
-              break;
-            }
+          uint48 orderId = uint48(remainingSegment);
+          if (orderId == 0) {
+            // Skip this order in the segment as it's been deleted
+            ++numOrdersFullyConsumed;
+            continue;
+          }
+
+          uint256 orderConsumed = 0;
+          uint256 quantityNFTClaimable = 0;
+          uint256 tradeCost = 0;
+          (segment, quantityRemaining, orderConsumed, quantityNFTClaimable, tradeCost) = _consumeOrder(
+            bestPrice,
+            segment,
+            slot,
+            orderId,
+            quantityRemaining,
+            isTakingFromBuy
+          );
+
+          cost += tradeCost;
+          numOrdersFullyConsumed += orderConsumed;
+
+          orderIdsPool[numberOfOrders] = orderId;
+          quantitiesPool[numberOfOrders] = quantityNFTClaimable;
```

```
+          ++numberOfOrders;
+
+          require(numberOfOrders < MAX_ORDERS_HIT, TooManyOrdersHit());
          }
+      }

-        if (numSegmentsFullyConsumed != 0) {
-          uint256 tombstoneOffset = node.tombstoneOffset;
-          tree[tokenId].edit(bestPrice, uint56(numSegmentsFullyConsumed));
+      _updateSegmentsAndTree(
+        tree,
+        node,
+        segments,
+        bestPrice,
+        segment,
+        lastConsumedSegmentIndex,
+        numOrdersFullyConsumed
+      );

-          // Consumed all orders at this price level, so remove it from the tree
-          if (numSegmentsFullyConsumed == segments.length - tombstoneOffset) {
-            tree[tokenId].remove(bestPrice);
-          }
+      newQunatityRemaining = quantityRemaining;
+      newNumberOfOrders = numberOfOrders;
+
+    }
+
+    function _consumeOrder(
+      uint256 bestPrice,
+      bytes32 segment,
+      uint256 slot,
+      uint48 orderId,
+      uint256 quantityRemaining,
+      bool isTakingFromBuy
+    ) private returns (
+      bytes32 newSegment,
+      uint256 newQuantityRemaining,
+      uint256 orderConsumed,
+      uint256 quantityNFTClaimable,
+      uint256 tradeCost
+    ) {
+      uint32 storedNormalizedQuantity = uint32(uint256(segment >> (slot * SEGMENT_SIZE + ORDER_ID_SIZE)));
+      uint256 quantityL3 = storedNormalizedQuantity * QUANTITY_TICK;
+      if (quantityRemaining >= quantityL3) {
+        // Consume this whole order
+        unchecked {
+          quantityRemaining -= quantityL3;
          }
+        quantityNFTClaimable = quantityL3;
+        orderConsumed = 1;
+      } else {
+        // Eat into the order
+        uint256 newNormalizedQuantity = (quantityL3 - quantityRemaining) / QUANTITY_TICK;
+        segment = bytes32(
+          (uint256(segment) & ~(uint256(0xff_ff_ff_ff) << (slot * SEGMENT_SIZE + ORDER_ID_SIZE))) |
+            (newNormalizedQuantity << (slot * SEGMENT_SIZE + ORDER_ID_SIZE))
+        );
+        quantityNFTClaimable = quantityRemaining;
+        quantityRemaining = 0;
+      }
+      tradeCost = (quantityNFTClaimable * bestPrice) / BASE_DECIMAL_DIVISOR;

-        if (eatIntoLastOrder || numOrdersWithinLastSegmentFullyConsumed != 0) {
-          // This segment wasn't completely filled before
-          if (numOrdersWithinLastSegmentFullyConsumed != 0) {
-            for (uint256 i; i < numOrdersWithinLastSegmentFullyConsumed; ++i) {
-              segment &= _clearOrderMask(i);
-            }
-          }
-          if (uint256(segment) == 0) {
-            // All orders in the segment are consumed, delete from tree
-            tree[tokenId].remove(bestPrice);
-          }
+      ClaimableTokenInfo storage claimableTokenInfo = _tokenClaimables[orderId];
+      if (isTakingFromBuy) {
```

```
+        claimableTokenInfo.amount += uint88(quantityNFTClaimable);
+      } else {
+        uint256 fees = _calcFees(tradeCost);
+        claimableTokenInfo.amount += uint88(tradeCost - fees);
+      }

-          segments[lastSegment] = segment;
+      newSegment = segment;
+      newQuantityRemaining = quantityRemaining;
+    }

-          if (eatIntoLastOrder) {
-            break;
-          }
+    function _updateSegmentsAndTree(
+      BokkyPooBahsRedBlackTreeLibrary.Tree storage tree,
+      BokkyPooBahsRedBlackTreeLibrary.Node storage bestPriceNode,
+      bytes32[] storage segments,
+      uint64 bestPrice,
+      bytes32 segment,
+      uint256 lastSegmentIndex,
+      uint256 numOrdersFullyConsumed
+    ) private {
+      uint256 numSegmentsFullyConsumed = numOrdersFullyConsumed / NUM_SLOTS_PER_SEGMENT;
+      uint256 numOrdersWithinLastSegmentFullyConsumed = numOrdersFullyConsumed % NUM_SLOTS_PER_SEGMENT;
+
+      if (numSegmentsFullyConsumed != 0) {
+        tree.edit(bestPrice, uint56(numSegmentsFullyConsumed));
+
+        // Consumed all orders at this price level, so remove it from the tree
+        if (bestPriceNode.tombstoneOffset == segments.length) {
+          tree.remove(bestPrice);
        }
      }
+
+      for (uint256 i; i < numOrdersWithinLastSegmentFullyConsumed; ++i) {
+        segment &= _clearOrderMask(i);
+      }

-      if (numberOfOrders != 0) {
-        assembly ("memory-safe") {
-          mstore(orderIdsPool, numberOfOrders)
-          mstore(quantitiesPool, numberOfOrders)
-        }
+      if (uint256(segment) == 0) {
+        // All orders in the segment are consumed, delete from tree
+        tree.remove(bestPrice);

-        emit OrdersMatched(sender, orderIdsPool, quantitiesPool);
+        // Question: Perhaps should pop out the last segment here?
+        // segments.pop(); // casues a test to fail !! 1) SonicOrderBook --> System test (many orders):
+      } else {
+        // segments[lastSegmentIndex] = segment;
      }
+
+      segments[lastSegmentIndex] = segment;
    }

    function _takeFromOrderBook(
```

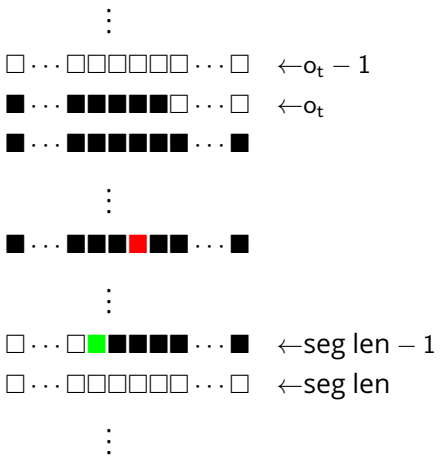
Paintswap: Fixed in PR 41.

Cantina Managed: The recommendation with some modifications has been applied in PR 41. The new implementation breaks invariant 9. The new invariant is that all the orders in segments before the tombstone offset of a price node are all deleted/zeroed out:

Either $o_t = 0$ or if $o_t > 0$ we should have:

$$\ldots$$
$$\square\square\square \leftarrow o_t - 1$$

So the new shape of the price segment would be roughly:


$\leftarrow o_t - 1$
$\leftarrow o_t$
$\leftarrow$seg len $- 1$
$\leftarrow$seg len