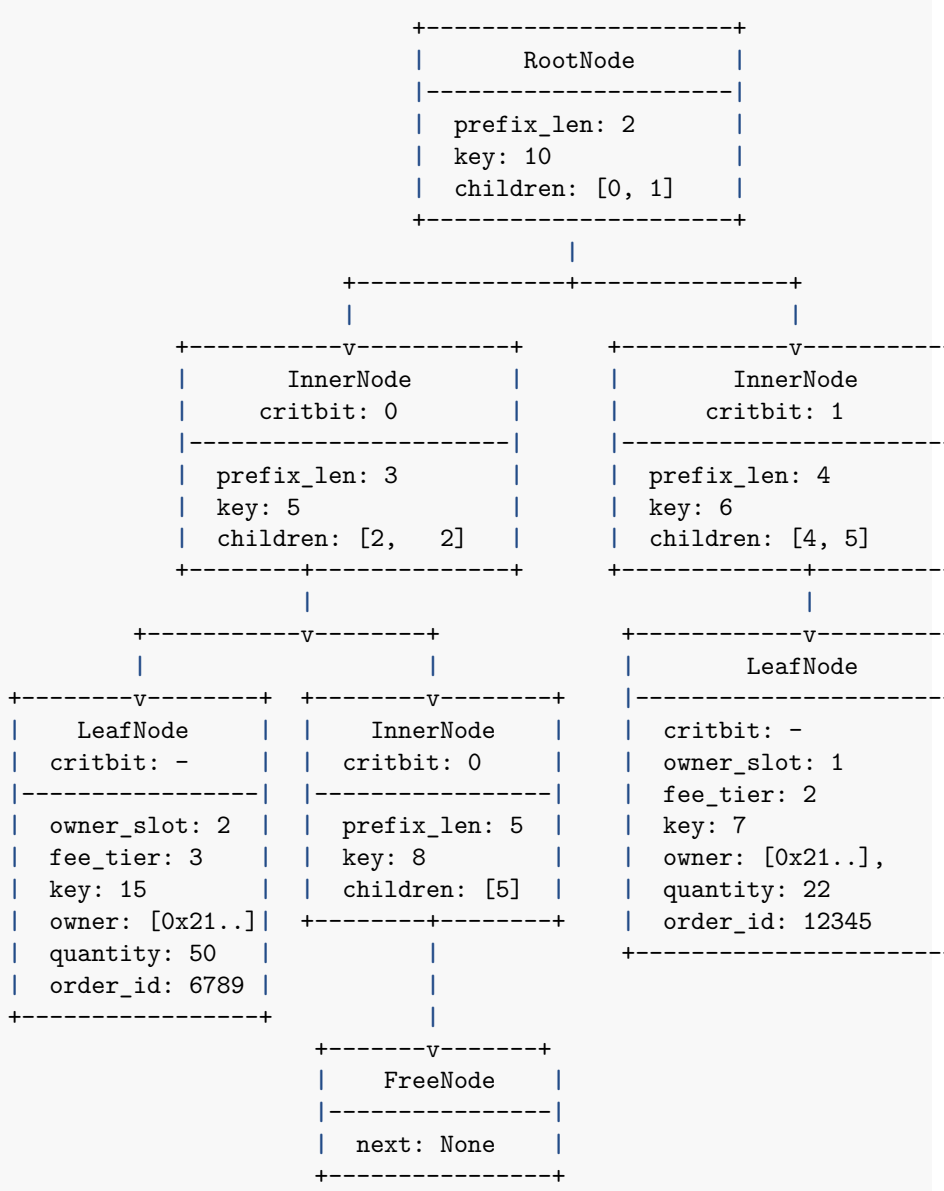
# Chapter 4: Understanding the Critbit Process in OpenBook v1




## Introduction

Welcome to the fourth chapter of this series where we explore the universe of web3 in Rust, particularly focusing on the Solana blockchain. This chapter goes

over each component of the Critbit process, exploring their roles, interactions, and implementation details within the OpenBook v1 Dex.

To start, let's explore the Crit-Bit tree data structure.

### 1. Crit-Bit Trees

The **Critbit** module in OpenBook v1 implements a **CritBit Tree**, short for "Critical Bit Tree". This type of tree structure excels in storing key-value pairs where keys are Strings and can be compared bitwise (like IP addresses or cryptographic hashes), making it particularly suitable for applications requiring fast prefix-based searches and updates[1]. The primary components of Critbit include **Inner Nodes**, **Leaf Nodes**, and **Free Nodes**, each serving a different role in the organization and operation of the data structure.

**Node Representation**:

```
pub type NodeHandle = u32;

#[derive(IntoPrimitive, TryFromPrimitive)]
#[repr(u32)]
enum NodeTag {
    Uninitialized = 0,
    InnerNode = 1,
    LeafNode = 2,
    FreeNode = 3,
    LastFreeNode = 4,
}
```

- `NodeHandle`: A handle to refer to nodes in the tree.
- `NodeTag`: Tags used to differentiate between different node types (Inner, Leaf, Free, LastFree).

**Node Structures**: - `InnerNode`: Represents internal nodes of the crit-bit tree.

```
struct InnerNode {
    tag: u32,
    prefix_len: u32,
    key: u128,
    children: [u32; 2],
    _padding: [u64; 5],
}
```

- `prefix_len`: Length of the common prefix with the parent node.
- `key`: The key associated with the node.
- `children`: Pointers to child nodes.

[1] Binary key comparisons efficiency: Binary Search Trees

- `LeafNode`: Represents leaf nodes storing actual data, the order in the tree.

```
struct LeafNode {
    tag: u32,
    owner_slot: u8,
    fee_tier: u8,
    padding: [u8; 2],
    key: u128,
    owner: [u64; 4],
    quantity: u64,
    client_order_id: u64,
}
```

**Reference**: openbook-dex/program

The **Inner Node** in Critbit contains metadata about the binary prefix length it handles, the key it represents, and pointers to its child nodes. This metadata allows efficient traversal of the tree based on the bits of the search key, ensuring quick lookup times even with large datasets[2]. On the other hand, **Leaf Nodes** store actual data entries, such as the order itself. These nodes hold specific information important to the decentralized exchange operations, including the owner details, quantities, and client order IDs[3].

```
struct FreeNode {
    tag: u32,
    next: u32,
    _padding: [u64; 8],
}
```

- `tag` and `next`: Markers to identify and link free nodes.
- `_padding`: Padding for alignment and structure integrity.

**Reference**: openbook-dex/program

Complementing these are **Free Nodes**, which manage memory reclamation and recycling within the tree structure. These nodes keep track of available memory slots, ensuring efficient space utilization and allocation as nodes are inserted, updated, or removed from the tree[4].

The overall design of Critbit in OpenBook v1 revolves around optimizing memory usage, facilitating fast data access, and ensuring scalability as the decentralized exchange platform grows.

### 2. InnerNode Structure Operations

As previously mentioned, the `InnerNode` structure manages internal nodes within the crit-bit tree. It defines methods for traversal and child node manipulation.

[2]Optimization techniques for tree traversal: Tree Traversal
[3]More about Tree data structure terminology such as Leaf Nodes: Leaf Nodes
[4]Memory management in tree structures: Trees in Memory Management

The `InnerNode` struct:

```
impl InnerNode {
    fn walk_down(&self, search_key: u128) -> (NodeHandle, bool) {
        let crit_bit_mask = (1u128 << 127) >> self.prefix_len;
        let crit_bit = (search_key & crit_bit_mask) != 0;
        (self.children[crit_bit as usize], crit_bit)
    }
}
```

- `walk_down`: Determines the next child node based on the crit-bit comparison with the search key.

**Reference**: openbook-dex/program

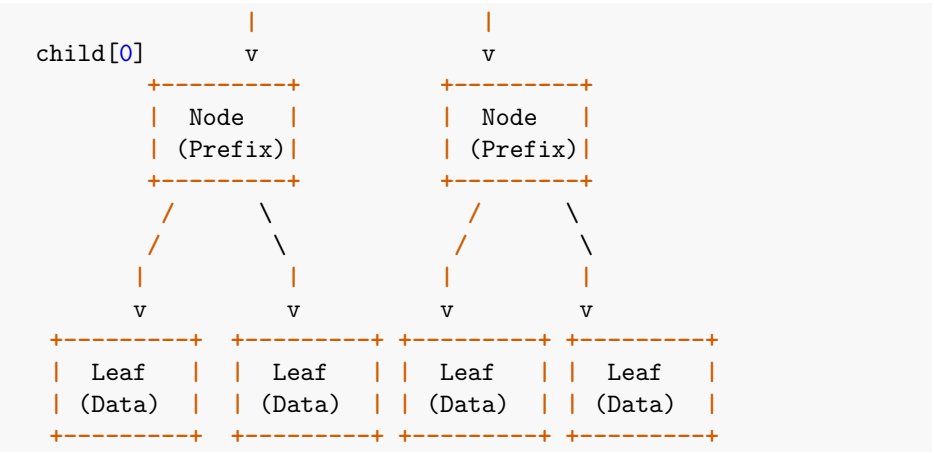
let's visually understand the inner working of this method using the following diagram:

```
InnerNode::walk_down

   +---+---+---+---+---+---+---+---+
   | 1 | 0 | 0 | ..... | 0 | 0 | 0 | crit_bit_mask = (1u128 << 127) >> 2
   +---+---+---+---+---+---+---+---+
                 |
                 v
               +-----+                          +---+---+---+---+---+---+---+---+
               | AND | ------------> crit_bit | 0 | 0 | 0 | ..... | 0 | 0 | 0 | search_key
               +-----+                          +---+---+---+---+---+---+---+---+
                  ^
                  |
   +---+---+---+---+---+---+---+---+
   | 0 | 0 | 0 | ..... | 0 | 1 | 1 | search_key = 3
   +---+---+---+---+---+---+---+---+

return child[crit_bit], crit_bit = child[0], false


                     Root Node
                        |
                        v
                    +---------+
                    |   Node  |
                    | (Prefix)|
                    +---------+
                     /       \
                    /         \
                   /           \
            crit_bit=0    crit_bit=1
```




### 3. LeafNode Structure Operations

The `LeafNode` structure represents the leaf nodes storing actual data within the crit-bit tree.

```
impl LeafNode {
    fn fee_tier(&self) -> FeeTier {
        // ...snip...
    }

    fn price(&self) -> NonZeroU64 {
        // ...snip...
    }

    fn order_id(&self) -> u128 {
        // ...snip...
    }

    fn quantity(&self) -> u64 {
        // ...snip...
    }

    fn set_quantity(&mut self, quantity: u64) {
        // ...snip...
    }
    // ...snip...

    fn owner(&self) -> [u64; 4] {
        // ...snip...
    }
```

```
    fn owner_slot(&self) -> u8 {
        // ...snip...
    }

    fn client_order_id(&self) -> u64 {
        // ...snip...
    }
}
```

**Reference**: openbook-dex/program

### 4. FreeNode Structure and Operations

The `FreeNode` structure manages nodes that are marked as free in the crit-bit tree, allowing efficient node reuse.

The `FreeNode` struct:

```
struct FreeNode {
    tag: u32,
    next: u32,
    _padding: [u64; 8],
}
```

- `tag` and `next`: Markers to identify and link free nodes.
- `_padding`: Padding for alignment and structure integrity.

**Reference**: openbook-dex/program

### 5. Other Critbit Operations

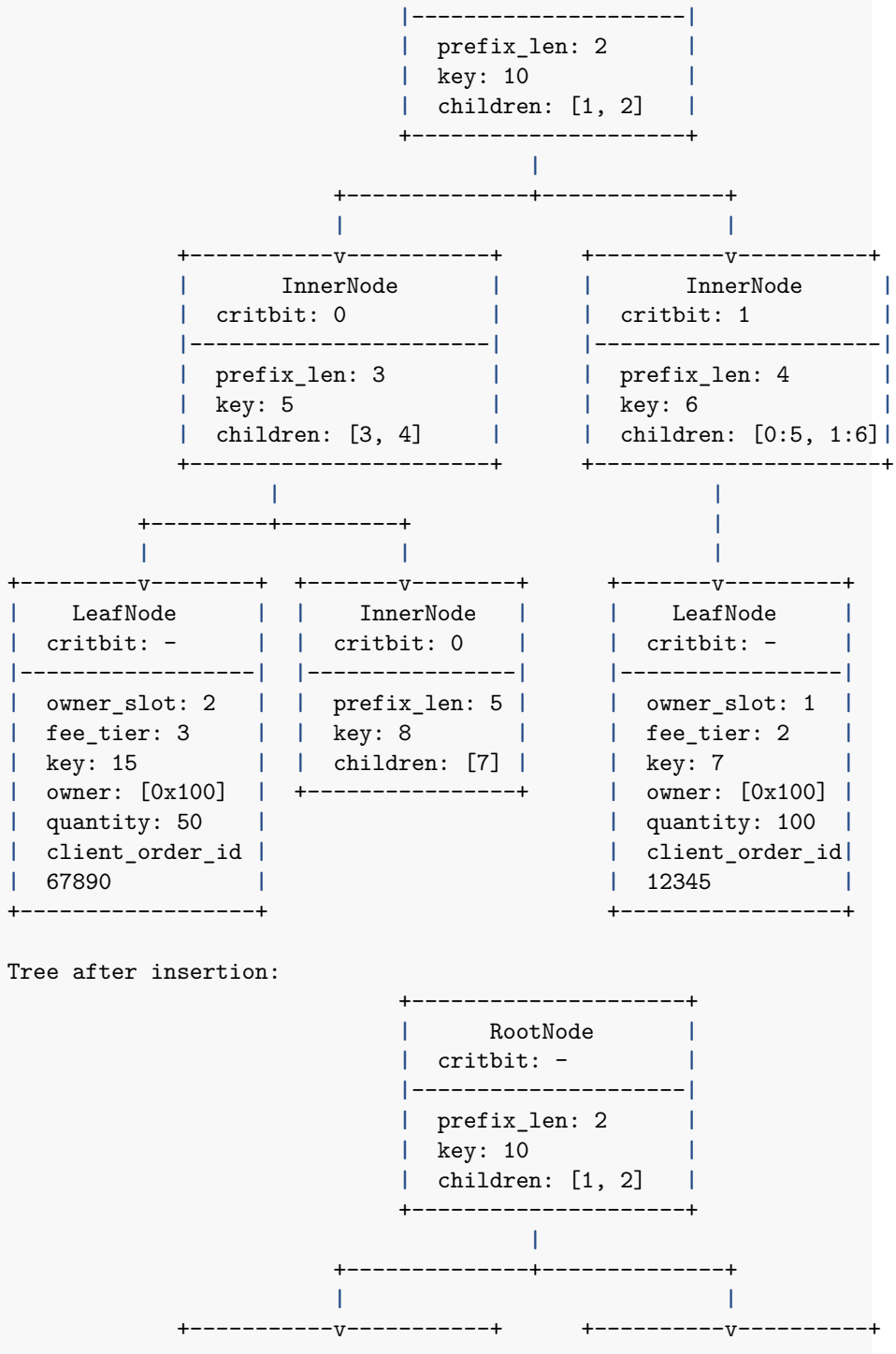
Now, let's visualize how `insert_leaf`, `find_by_key`, `remove_by_key`, `find_min`, and `find_max` work by showing how they interact with the crit-bit tree structure.

**5.1 `insert_leaf`** This method inserts a new leaf node into the tree. let's have a look at the process of traversing the tree and finding the correct position for the new leaf node.

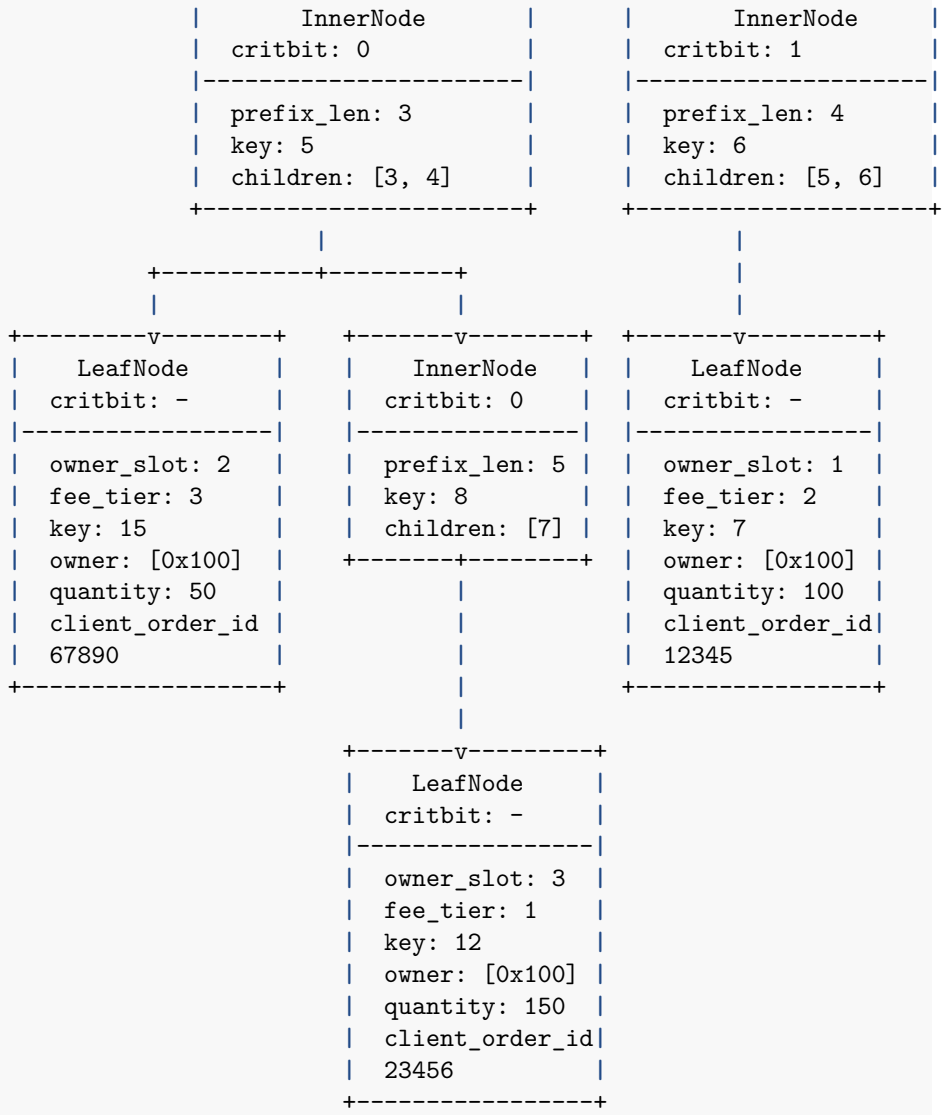
```
Insert Leaf (key: 12, owner: [0x100], quantity: 150)

1. Start at RootNode
2. Traverse based on critbit until the correct position is found
3. Insert the new leaf node

Tree before insertion:
                              +---------------------+
                              |      RootNode       |
                              |  critbit: -         |
```




Tree after insertion:



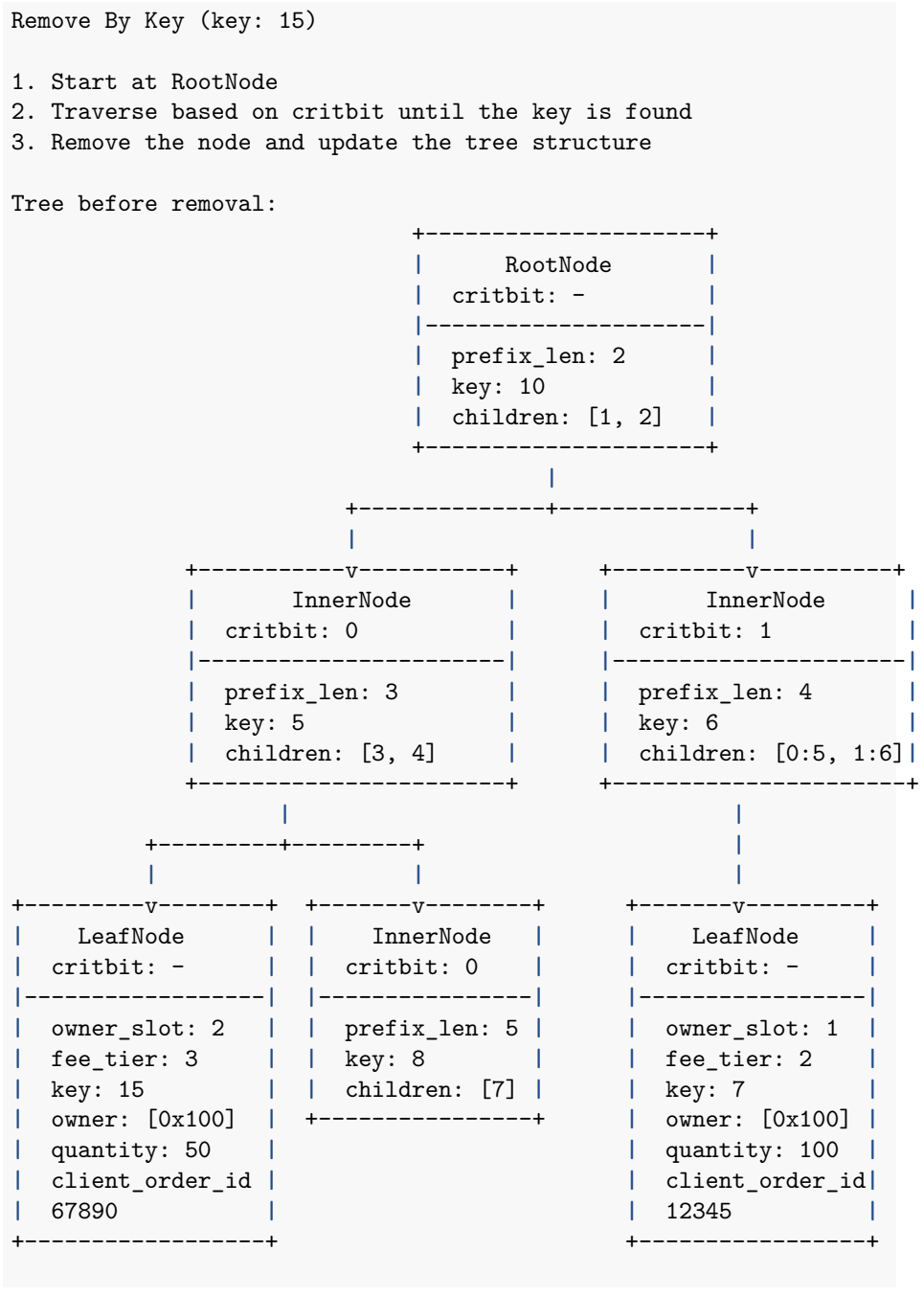

### 5.2 `find_by_key`

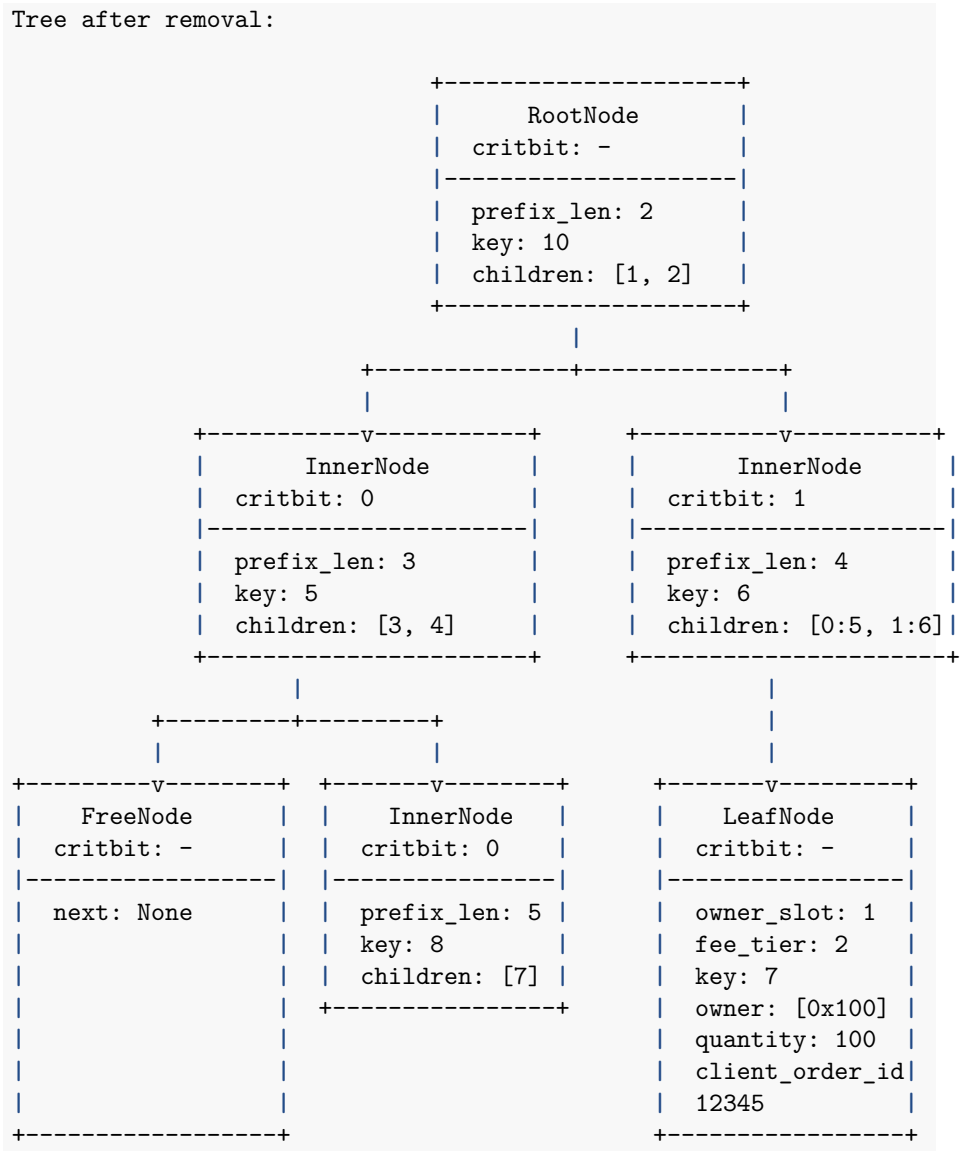
This method searches the tree for a specific key and retrieves the corresponding leaf node.

1. Start at RootNode
2. Traverse based on critbit until the key is found or a leaf node is reached

### 5.3 remove_by_key

This method removes a node from the tree based on the specified key.






### 5.4 `find_min` and `find_max`

These methods find the minimum and maximum keys in the tree, respectively.

Find Min

1. Start at RootNode
2. Traverse to the leftmost node (critbit: 0) until a leaf node is found

Find Max

1. Start at RootNode
2. Traverse to the rightmost node (critbit: 1) until a leaf node is found

---

## Conclusion

In this chapter, we have explored the detailed implementation and workings of the crit-bit tree in OpenBook V1. From node structures to operations, each aspect contributes to an efficient and robust crit-bit tree implementation suitable for various applications requiring fast key lookups and efficient memory usage.