Red-Black Trees

Pre-requisite Knowledge: - Binary Search Trees - Rotations (conceptual). The implementation of a rotation will have accompanying exmplanation in the instructions file.

A red-black tree is a variation on binary search trees (BSTs). The variations ensures that the BST stays balanced, avoiding a worst-case BST where all the nodes get added to one side. Since the tree is balanced, the height of the tree becomes O(lg n), where n represents the number of nodes. Operations on RB-trees take O(lg n) time in worst case.

The variation that allows the BST to be balanced comes from adding an additional bit of info to each node in the tree: an attribute colour (red or black). The rest of the BST properties remain (node value, left node, right node, parent. . .).

There are 5 properties that need to be satisfied in order to have an RB-tree: 1. Every node must be one of red or black (i.e. each node needs to have the attribute colour.) 2. The root of the tree is black. 3. All the leaves (called Nil) are black. 4. If a node is red, its children must be black. Therefore there can't be consecutive red nodes. 5. For each node, all paths from the node to its descendant leaves have the same number of black nodes.

The black height of a node is defined as the number of black nodes (including a leaf) on the path from the node to a descendant leaf.

The operations we would want to perform on an RB-tree are similar to those of a regular BST: insert & delete. We also want a method that will check if a tree is a valid RB-tree.

Data type definition

To start implementing a functional red-black tree in OCaml, we must first define 2 data types. The first datatype will be a color type, that can be either Black or Red. The second data type is the same as a regular recursive defintion of a BST, however we add to it a colour for every node, and we call the leaves Nil instead of a leaf.

1. A node in a red-black tree can either be Red or Black. This is defined as a simple variant type:

```
type color = Red | Black
```

2. The tree is a recursive data structure with the following definition:

```
type 'a rb_tree =
    | Nil
    | Node of color * 'a * 'a rb_tree * 'a rb_tree
```

Nil represents a leaf node and is always considered black. The node contains: - A color (Red or Black). - A value of type 'a (the data stored in the node). - A

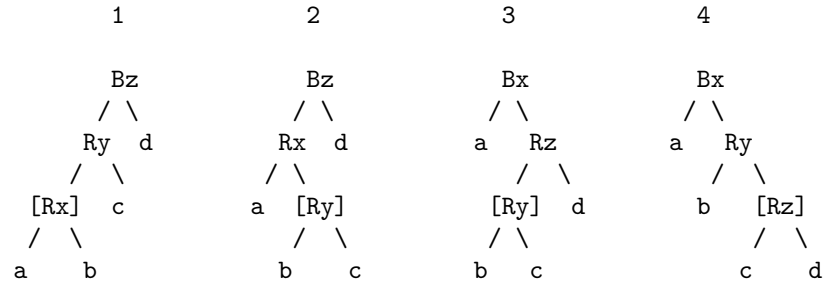
left subtree and a right subtree, both of type 'a rb_tree.

This structure allows us to represent the red-black tree's key properties, including node colors, hierarchical relationships, and the recursive nature of the tree.

Insertion:

Inserting a node into an RB-tree should not break the RB-trees properties. In non-functional programming, this is the procedure for implementing insertion in RB-trees: 1. Use regular BST insert to a node, x, into RB-tree, T. 2. Colour x red. 3. Use re-colouring and rotations to restore the RB-tree properties.

However, in functional programming, we follow the same first 2 steps, but the third step is different. Okasaki's algorithm is an elegant solution for the insertion and balancing of a node into a red-black tree. Since we are inserting a red node, then there are some possible conflicts, and, some impossible conflicts, of the 5 properties. For instance, it is impossible for a node to have no colour, it is impossible for the black-height of any node to change, since we are adding a red node, and the leaves will always be black. However, if we inserted the root, then now the root is red, and needs to be black. If we did not insert the root, then it is possible we inserted a red node as the childe of another red node. So, this is the procedure to fix the tree properties: 1. If the parent of the inserted node is black, then we are done with insertion. 2. If the parent of the inserted node is red, then we must fix the tree. Note that it is impossible for the parent node to be the root, because the root is black, and thus we can say that the inserted node has a grandparent. Then, there are 4 cases that can arise:



In the above cases, the notation Bz indicated that there is a node z that is blac, the capital R indicates a red node, and then a, b, c, and d are all subtrees, possibly empty, meaning we do not care about the specific nodes in them. The nodes marked with [] indicate that they are the node that violates the RB-tree property. By performing one rotation on one of the above cases, we are left with a section of the tree that no longer violates the properties, as seen below. However, it is possible that we simply moved the problem up, and we will need to balance once more.

Now, if Ry does not have a parent, this means that it is the root of the tree. Thus, we need to make it Black, and then a node has been successfully. However, if Ry has a parent, we need to re-check and make sure that we have not created a new violation of the RB-tree properties. Specifically, since there can't be an issue with the black height, the problem would be that Ry's parent is a red node. So, we need to recursively travel up the tree, fixing these violations, until we reach the root node, where we will finally have a valid RB-tree.

More in depth implementation explanation is given in instructions.md.

Time complexity analysis: There are at most O(log n) recursions, since the height is bounded by O(log n), and the at each recursive step, there is a call to balance which is done in O(1), then the final time comlpexity of the insert function is O(log n)

Remark: The functional implementation is slightly different than the more commonly found implemenations. Online simulators may implement different algorithms, thus doing the same sequence of inserts in the functional implementation and the other implementation might result in different RB-tree. However, the functional programming one remains valid as none of the 5 properties are violated after insertion.

Deletion:

Deleting a node from a red-black tree is more complex than insertion because we need to ensure that the red-black properties are maintained even when removing a node. The key challenge lies in preserving the black height property, which requires special handling when removing a black node.

In functional programming, the process follows these general steps:

Overview of the Process

1. Locate the Node to Delete:

- Start with a standard binary search tree traversal to locate the node with the value to be deleted.
- Once located, handle the node according to its child structure:
- No Children (Leaf Node): Remove it directly.
- One Child: Replace the node with its child.
- Two Children: Replace the node's value with the smallest value in its right subtree (the in-order successor), then recursively delete the successor node.

2. Recoloring and Balancing:

- If the deleted node is red, no rebalancing is needed because red nodes do not affect the black height.

- If the deleted node is black, rebalancing is necessary to maintain the black height and ensure all paths from a node to its leaves have the same number of black nodes.

3. Handle Double-Black:

- Removing a black node can create a "double-black" situation, where a subtree is missing one black node.
- To fix this, we apply a series of transformations (rotations and recoloring) to redistribute the black height and ensure the red-black properties are restored.

4. Restore Red-Black Properties:

- Using recoloring and rotations, ensure:
- The root is black.
- There are no consecutive red nodes.
- The black height is consistent across all paths.

Details of the Balancing Cases

Balancing after deletion involves handling specific configurations, depending on the sibling (or "brother") of the double-black node:

1. Sibling is Red:

- If the sibling of the double-black node is red, rotate the sibling up, recolor it black, and continue rebalancing.

2. Sibling is Black and Has Two Black Children:

- If the sibling is black and both of its children are black, recolor the sibling red. This reduces the black height issue one level up, requiring recursive balancing on the parent.

3. Sibling is Black and Has a Red Child:

- If the sibling has at least one red child, perform rotations and recoloring to redistribute the black height.

4. Sibling is Black with One Red Child on the "Outside":

- Perform a single rotation (or double rotation, if necessary) to fix the imbalance, and recolor nodes appropriately.

Recursive Fixing:

- The rebalancing process starts at the point of deletion and propagates upward toward the root if necessary.
- At each step, we adjust the structure and recolor nodes to maintain the red-black properties.

Final Adjustments:

1. Ensure the Root is Black:

- After all transformations, check the root of the tree. If it is red, recolor it black.

2. Return the Updated Tree:

- The final tree will satisfy all red-black properties.

Time Complexity Analysis:

- Search for the Node to Delete: O(log n) , since the height of the tree is bounded by O(log n) .
- Balancing Operations: Each balancing operation (rotations and recoloring) takes O(1) , and there can be at most O(log n) such operations (one per level of the tree).
- Overall Complexity: O(log n) .

Check Validity:

The function is_valid ensures that a given tree adheres to the five key properties of a red-black tree. To validate the tree, we check each property recursively and return a failure as soon as one is violated. If the entire tree satisfies all properties, it is a valid red-black tree. Here's how we can systematically verify validity:

The Five Properties of Red-Black Trees

1. Every node is either red or black. (In this implementation, the color type ensures this by design.)
2. The root of the tree is black. (Implicitly checked by ensuring the black height is consistent across all paths.)
3. All leaves (Nil nodes) are black. (This is true by design since Nil represents a black node.)
4. If a node is red, its children must be black. (This is explicitly checked by the no_red_with_red_child function.)
5. For each node, all paths to descendant leaves must have the same number of black nodes. (This is verified by the black_height function.)

How the Validation Works

1. Black Height Check (black_height):

- Recursively calculates the number of black nodes along any path from the root to a leaf.
- Ensures that all paths have the same black height.
- If any discrepancy is found (e.g., unequal heights or invalid subtree), it returns 0, indicating failure.

2. Red-Child Check (no_red_with_red_child):

- Traverses the tree recursively.
- Ensures that if a node is red, both of its children (if any) are black.

- Returns false immediately if a red node with a red child is detected.

3. Combine Results:

- The tree is valid if and only if:
- The black height check passes (returns a positive number).
- No red node has red children.

Thus, to check the validity of a red-black tree:

1. Recursively ensure that all paths have the same number of black nodes (black_height).
2. Ensure no red node has a red child (no_red_with_red_child).
3. If both checks pass, the tree is valid; otherwise, it is not.

This method efficiently validates the tree by combining these checks during a single traversal of the tree, making it O(n) in time complexity for a tree with n nodes.

For more information, check out our project's GitHub repo: https://github.com/JeremiasZimmerman213/COMP3
RBTree-Project