VISVESWARAYA TECHNOLOGICAL UNIVERSITY

BELGAUM

SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING, MYSORE-570006

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Report on

IMPLEMENTATION OF RED-BLACK TREE

Guidance of

N.R. PRASHANTH
Professor
Department of CS&E, SJCE, Mysore.

Done By:

VIKRAM T.V.
4th Semester,
Computer Science and Engineering,
S.J.C.E, Mysore

Contents

1 Introduction

Red black tree, a variant of binary tree, is determined by the color assigned to each of the node. The color may be either 'red' or 'black'. Knowing the color of a node helps us to balance the tree.

2 Properties and Rotation

A binary tree is said to be a red black tree if it satisfies the following properties.

- Every node is either red or black.
- The root is black.
- Every null leaf is black.
- If a node is red, then both its children are black.
- For each node, all paths from the node to descendant leaves contain the same number of black nodes (known as 'black height').

In order to balance the tree using the color properties, we may need to change color of the nodes and adjust the pointers. Adjusting pointers is done by **rotation**, which prevents the binary search property of the tree. There are two kinds of rotation - left rotation and right rotation. Consider a tree with internal nodes 'x' and 'y' and 'A', 'B', 'C' as leaf nodes. The left and right rotations are complementary and are as shown in the figure.

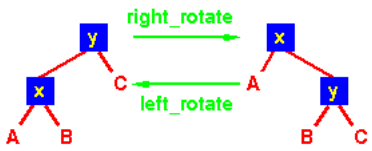

Figure 1: Rotation of a tree

3 The Algorithm

The algorithm for insertion and deletion involves coloring and rotation to preserve the red black properties of the tree.

3.1 Insertion

Insertion of a new node to the tree involves two stages. First, the new node 'z' is colored 'red' and inserted as if it were an ordinary binary search tree. Next, the rotations are done to 'fixup' the insertion and can be one of the following cases:

- Case 1: Uncle[z] is also 'red'.
  In this case, both parent[z] and uncle[z] are colored 'black' and grandparent[z] is colored 'red'. Finally assign 'z' to its grandparent.
- Case 2: Uncle[z] is 'black'.
  Case 2a: If parent[z] is left child of its parent and 'z' is a *right child* of its parent, then left rotate at parent[z] (and perform case 3).
  Case 2b: if parent[z] is right child of its parent and 'z' is a *left child* of its parent, then right rotate at parent[z] (and perform case 3).
  These operations have not changed any colors.
- Case 3: Uncle[z] is again 'black', but may have resulted from case 2 after rotation.
  Assign parent[z] with 'black', grandParent[z] with 'red' and perform right rotation for case 2a or left rotation for case 2b at grandParent[z].

Perform the above cases until parent[z] remains 'red'. Finally, assign the color[T] as 'black'.

The above cases are represented by the following algorithm; where color of node 'z' is represented as color[z], left and right children as left[z] and right[z] repectively.

Algorithm 1 RBT Insert

1: **rbtInsert (T, z)**
2: insert node 'z' as if it were inserted to an ordinary binary search tree.
3: color[z] ← RED
4: rbtInsertFixup (T, z)

3.2 Deletion

The deletion of a node from a red black tree is also same as the deletion of a node from an ordinary binary search tree. Three cases arise, with 'z' being the node to be deleted, 'y' - the node to be spliced out and 'x' - the child of 'y'.

- Case 1: When 'z' is a leaf node, just delete it.
- Case 2: When 'z' has one child, then splice out 'z' and attach the child[z] to parent[z]. In the above two cases, 'z' equals 'y'.
- Case 3: When 'z' has both the children, then identify node 'y' as the successor of 'z' and

Algorithm 2 RBT Insert Fixup

```
1: rbtInsertFixup (T, z)
2: while color[parent[z]] = RED do
3:   if parent[z] = left[grandparent[z]] then
4:     y ← right[grandparent[z]]
5:     if color[y] = RED then
6:       color[parent[z]] ← BLACK
7:       color[y] ← BLACK
8:       color[grandparent[z]] ← RED
9:       z ← grandparent[z]
10:    else
11:      if z = right[parent[z]] then
12:        z ← parent[z]
13:        leftRotate (T, z)
14:      end if
15:      color[parent[z]] ← BLACK
16:      color[grandparent[z]] ← RED
17:      rightRotate (T, grandparent[z])
18:    end if
19:  else
20:    (same as then clause with "right" and "left"
       exchanged)
21:  end if
22: end while
23: color[T] ← BLACK
```

splice out 'y' by attaching child[y] to parent[y]. Also add the information contents of 'y' to 'z'.

If the spliced out node 'y' is black, call the delete fixup function at 'x' (child[y]) to adjust the pointers and colors of the tree. This is because, due to the deletion of the black node, black heights and other red black properties mentioned in the Properties section might have changed.

Delete Fixup

Four cases arise depending on the color of x's sibling 'w' and its children. The below cases are when 'x' is left child of its parent.

- Case 1: x's sibling 'w' is red.
  Because 'w' is red, its children are black. Change color[w] to black and parent[x] to red. Left rotate at parent[x] and the new sibling of 'x' is now right[parent[x]]. This results in case 2, 3 or 4.

- Case 2: x's sibling 'w' is black, and both of w's children are black.
  Adjusting black height is done by coloring red to 'w' and moving up 'x' to its parent. As we just change color and move up, this case results in looping till we reach root or some other case.

- Case 3: x's sibling 'w' is black, w's left child is red, and w's right child is black.
  Color the left child of 'w' with black and 'w' with red. Right rotate at 'w', which results in a new 'w', sibling of 'x'.

- Case 3: x's sibling 'w' is black, and w's right child is red.
  Assign w's color with color of parent[x]. Now color of parent[x] and right child of 'w' is assigned with black. Left rotate at parent[x]. Assigning 'x' with the root of the tree stops the iteration.

Finally, assign 'x' with black color. The above cases are symmetric with 'x' being a right child of its parent, the left and right are exchanged. The delete fixup routine is repeated till 'x' is not root and its color is black.

The above deletion cases is given in algorithms 3 and 4. 'z' is the node to be deleted.

Algorithm 3 RBT Delete

```
1: rbtDelete (T, z)
2: delete node 'z' as if it were deleted from an ordi-
   nary binary search tree, with 'y' as the node to
   be spliced out and 'x' as the child of 'y'. Finally
   the spliced out node is returned.
3: if color[y] = BLACK then
4:   rbtDeleteFixup (T, x)
5: end if
6: return 'y'
```

4 Time Complexity

Insertion into the tree having 'n' nodes takes O(lg n) time. The while loop in the fixup after insertion loops only for case 1 and the 'z' pointer moves up by two levels and terminates for case 2 or case 3 and takes O(lg n) time. Thus, insertion takes O(lg n) time. For each insertion fixup, it takes no more than two rotations.

Deletion from the tree having 'n' nodes takes O(lg n) time. The fixup after deletion is repeated for case 2 and for other cases there can be atmost three rotations. From case 2, the pointer 'x' moves up at most O(lg n) times with no rotations and fixup takes O(lg

n) time with at most three rotations. Thus, deletion also takes O(lg n) time.

5 Sample Run

A sample run of the above algorithm would result in a tree as shown in figure 1. The nodes with color red are circled with red color and with black are colored with black color.

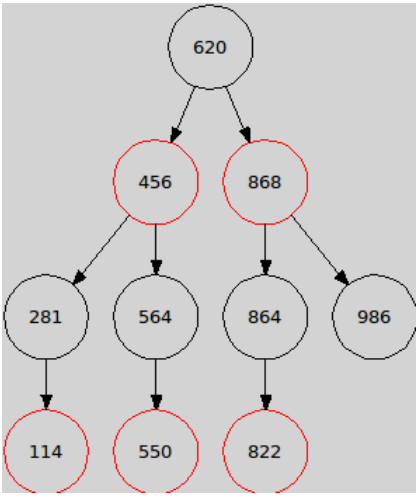

Figure 2: Red Black Tree

The time complexity for insertion and deletion of the nodes are shown in figure 2. The insertions are done randomly and deletions are done recursively at the root rather than choosing a random node.

Algorithm 4 RBT Delete Fixup

```
1:  rbtDeleteFixup (T, x)
2:  while x ≠ T and color[x] = BLACK do
3:    if x = left[parent[x]] then
4:      w ← right[parent[x]]
5:      if color[w] = RED then
6:        color[w] ← BLACK
7:        color[parent[x]] ← RED
8:        leftRotate (T, parent[x])
9:        w ← right[parent[x]]
10:     end if
11:     if color[left[w]] = BLACK and
        color[right[w]] = BLACK then
12:       color[w] ← RED
13:       x ← parent[x]
14:     else
15:       if color[right[w]] = BLACK then
16:         color[left[w]] ← RED
17:         color[w] ← RED
18:         rightRotate (T, w)
19:         w ← right[parent[x]]
20:       end if
21:       color[w] ← color[parent[x]]
22:       color[parent[x]] ← BLACK
23:       color[right[w]] ← BLACK
24:       leftRotate (T, parent[x])
25:       x ← T
26:     end if
27:   else
28:     (same as then clause with "right" and "left"
        exchanged)
29:   end if
30: end while
31: color[x] ← BLACK
```

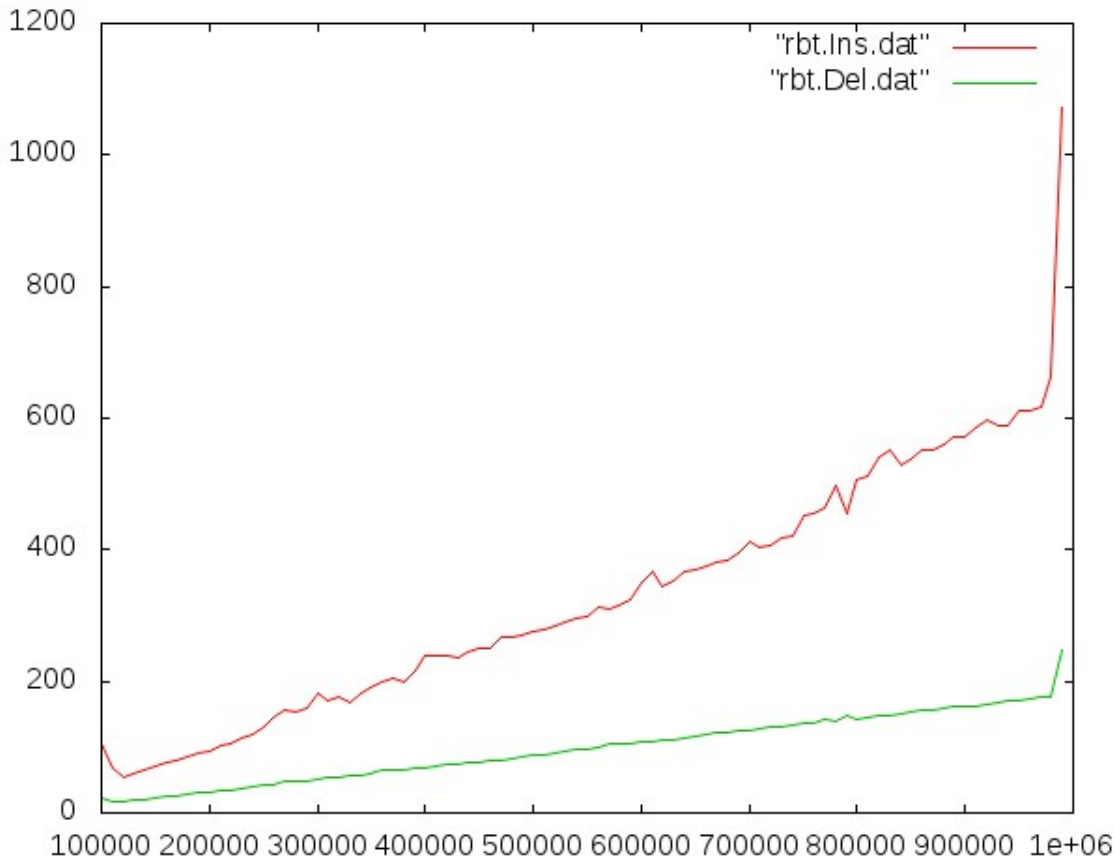

Figure 3: Red Black Tree: Insertion vs Deletion

References

[1] Cormen, Thomas H et al, 2005, *Introduction to Algorithms—Second Edition*, Prentice-Hall of India.