21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2023**

Lecture 2: Red-Black Tree & B+ Tree

Lecturer: Deshi Ye *Scribe: D. Ye*

2.1 Overview

Reb black is a balanced binary search tree, it was invented by Rudolf Bayer in 1972. Part of the lecture is based on Textbook [1].

Definition 2.1 *A red-black tree is a binary search tree that satisfies the following red-black properties:*

1. *Every node is either red or black.*
2. *The root is black.*
3. *Every leaf (NIL) is black.*
4. *If a node is red, then both its children are black.*
5. *For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.*

One example of Red-black tree is shown in Fig. 2.1.

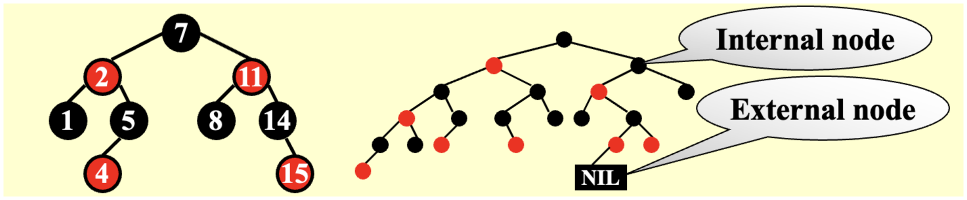

Figure 2.1: Example of Red black tree

Definition 2.2 *The* **black-height** *of any node x, denoted by $bh(x)$, is the number of black nodes on any simple path from x (x not included) down to a leaf. The black-height of a tree is the black height of the root, i.e., $bh(Tree) = bh(root)$.*

Theorem 2.3 *A red-black tree with N internal nodes has height at most $2\ln(N + 1)$.*

Proof: Let $sizeof(x)$ be the number of internal nodes in the subtree rooted at x. Let $h(x)$ be the height of x. We show by induction that $sizeof(x) \geq 2^{bh(x)} - 1$.

If $h(x) = 0$, x is NULL and $bh(x) = 0$, then $sizeof(x) = 2^0 - 1 = 0$. If $bh(x) = 1$, the tree has at least one black node, it true that $sizeof(x) \geq 2^1 - 1 = 1$.

Suppose it is true for all x with $h(x) \leq k$ and $bh(x) \geq 1$.

For any node x, let y be one of $x's$ child. we have $bh(y) = bh(x)$ if $x's$ child is a red node and $bh(y) = bh(x) - 1$ if y is a black node.

For x with $h(x) = k+1$, Since $h(y) \leq k$ due to y is a child of x, and then $sizeof(y) \geq 2^{bh(y)} - 1 \geq 2^{bh(x)-1} - 1$.

Since $bh(x) \geq 1$, we know x has two children. Let z be another child of x, again, we have $sizeof(z) \geq 2^{bh(x)-1} - 1$.

Thus, $sizeof(x) = 1 + sizeof(y) + sizeof(z) \geq 1 + 2(2^{bh(x)-1} - 1) = 2^{bh(x)} - 1$.

On the other hand, we know that $bh(x) \geq h(x)/2$. Since for every red node, both of its children must be black, hence on any simple path from the root to a leaf, at least half the nodes (root not included) must be black.

Denote h to be the height of the tree T with N nodes, then $sizeof(root) = N \geq 2^{bh(T)} - 1 \geq 2^{h/2} - 1$, and we have $h \leq 2\ln(N + 1)$, and the theorem follows.

■

2.2 Insert

Insert a new node x in the Red-black tree T, the top-down algorithm [1] is described as below. Using binary search tree algorithm insert the node x in T and color it red. Let z be the current node that might break rule 4. Initially, $z = x$. If z's parent is black, the algorithm stops, and it is already a Red-black tree. Otherwise, kept applying the following cases to adjust the new tree such that it is a red-black tree.

Case 1 [Color Flip] If z's uncle y is red. Flip z's parent A to black, and flip y to black, and flip z's grandfather node C to red (if C is not a root), and the algorithm stops if C is the root, otherwise it becomes the new node z. See Fig. 2.2 for illustration.

Case 2 [Single rotaiton] If z's uncle y is black, and z is an inner child. Make a left rotation on z's parent A in the oppesite direction such that A becomes the new z and it is an outer child. Fig. 2.3 illustrates it. Then it goes to Case 3.

Case 3 [Single rotation] If z's uncle y is black, and z is an outer child. See Fig. 2.4 for details. Make a rotation on z's grandfather C in the direction of the parent such that B becomes the new root of the subtree, and flip the color of parent B and grandfather C. The algorithm stops.

Note that the insertion algorithm stops when it applies case 2 or case 3.

Analysis: Case 2 makes a single rotation, and goes to case 3, actually, one can do it by a double rotation. Case 3 does a single rotation and a few color flips. The case 1 only flips color. In all, the insertion algorithm will finally adjust a tree to a Red-black tree, and it spends at most 2 rotations. Both the insertion of BST and color flipping will cost $O(\log n)$ time for a n-node tree.

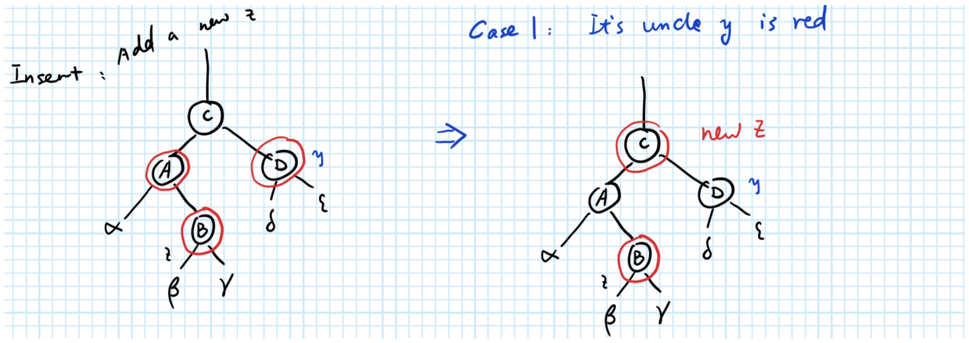

Figure 2.2: Case 1: color flip and one step goes up

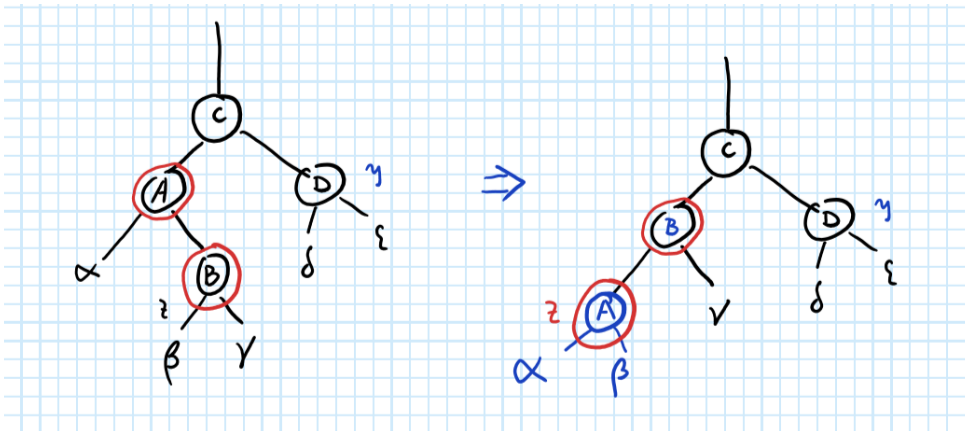

Figure 2.3: Case 2: If z's uncle is black, and z is an inner child

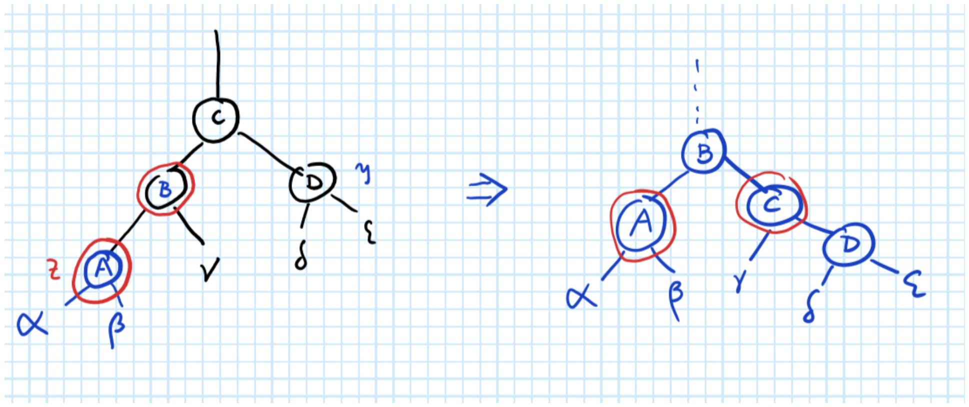

Figure 2.4: Case 3: If z's uncle is black, and z is an outer child.

2.3 Delete

Delete a node x in Red-black tree: The detailed algorithm is given as below.

- Apply BST deletion algorithms to delete x
- If node x is a red-node, it is done. Otherwise, applying the rules (4 cases) in Fig. 2.5, Fig. 2.6, Fig. 2.7, Fig. 2.8, and Fig. 2.9 to rebalance the black height.

We list the 4 cases and the way of recoloring and rotations as below.

Case 1 $x's$ sibling is red.

- flip the color of its parent and sibling.
- make a rotation at its parent in the direction of the deleted node x.
- move to one of the Cases 2-4.

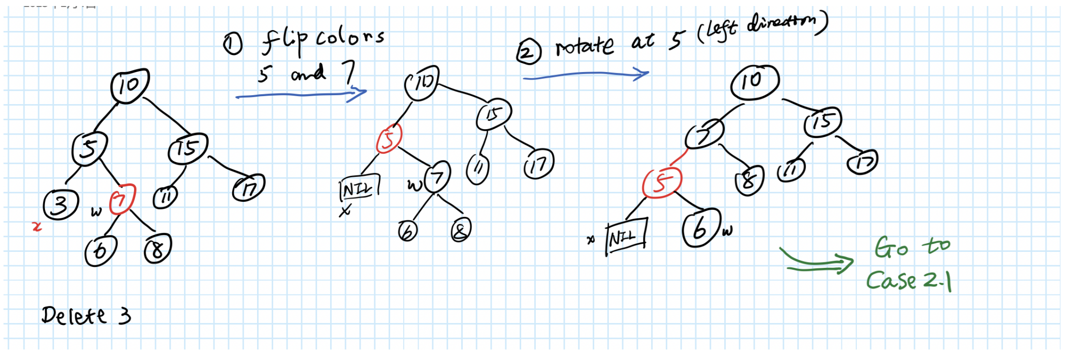

Figure 2.5: Case 1

Case 2.1 $x's$ sibling w is black, and both $w's$ children are black. Moreover, $x's$ parent is red.

- flip the color of $x's$ parent and its sibling w. Algorithm stops.

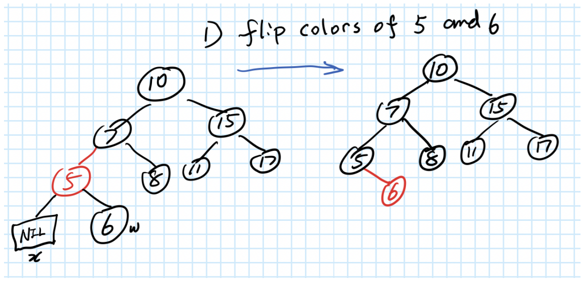

Figure 2.6: Case 2.1

Case 2.2 $x's$ sibling w is black, and both $w's$ children are black. Moreover, $x's$ parent is black.

- flip the color of $x's$ sibling's color to red.
- $x's$ parent becomes the new deletion node z, and the recursive continues.

Case 3 $x's$ sibling w is black, w has one red child, and “outer nephew” is black.

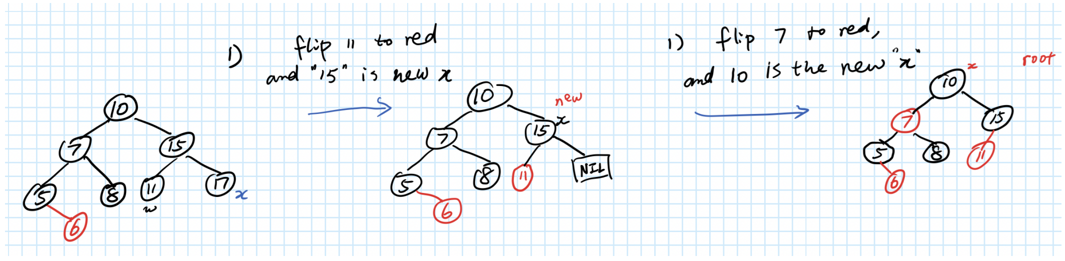

Figure 2.7: Case 2.2

- flip the color of w and its "inner nephew".
- rotate at $x's$ sibling w to the outer direction.

Algorithm stops.

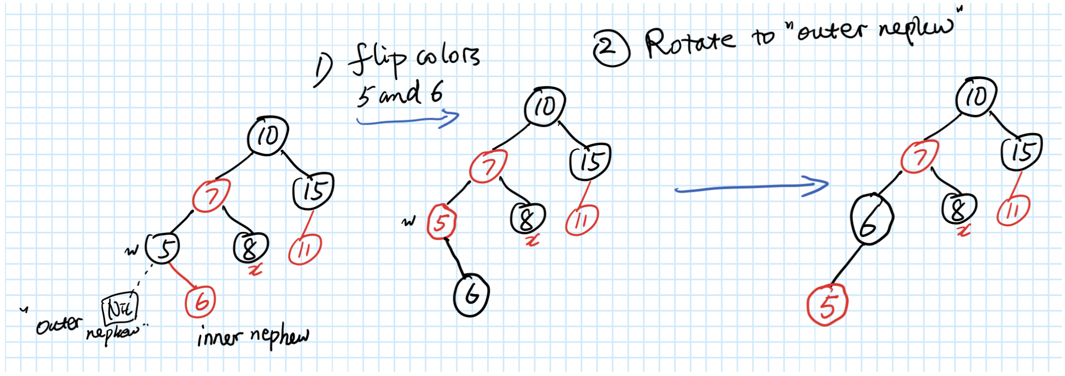

Figure 2.8: Case 3

Case 4 $x's$ sibling w is black, w has at least one red child, and "outer nephew" is red.

- recolor the sibling w in the parent's color; recolor the parent to black and flip the outer nephew to black.
- rotate at the parent node in the direction of the deleted node.

Algorithm stops.

Remark: Only Case 2.2 will make a further recursive. It is worth noticing that there is no rotation in Case 2.2. In each case, there is at most rotation. Hence, there are at most 3 rotations during the whole deletion operation (Case 1 to Case 3, and to Case 4). However, the recoloring might require $O(\log n)$.

2.4 B+ tree

B+ tree is a balanced search tree, an extension of the binary search tree.

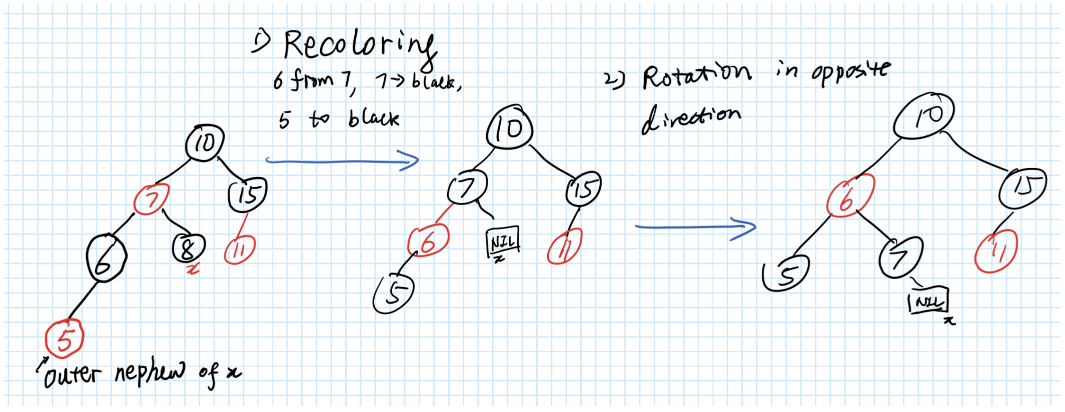

Figure 2.9: Case 4

Definition 2.4 *A B+ tree of order M is a tree with the following structural properties:*

(1) The root is either a leaf or has between 2 and M children.

(2) All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.

(3) All leaves are at the same depth. Assume each nonroot leaf also has between $\lceil M/2 \rceil$ and M keys.

B+ tree is to store all data in leaf nodes, and interior nodes keep the key value as the index.

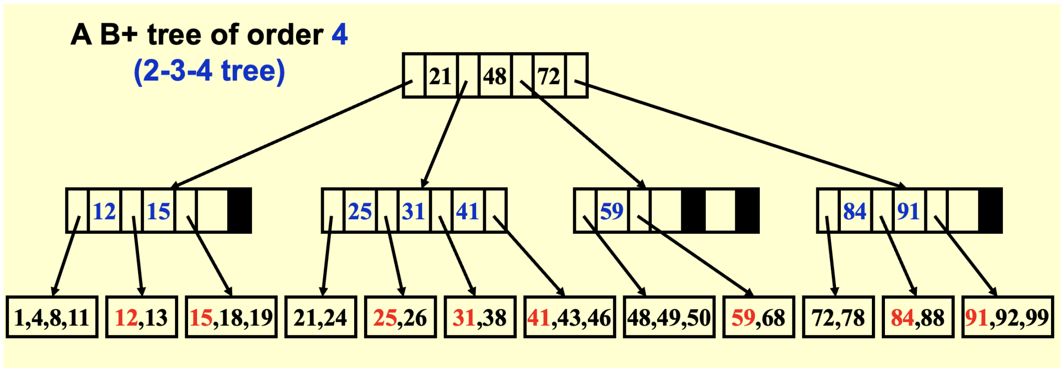

Figure 2.10: Example: A B+ tree of order 4

In Fig. 2.10, we give an example of B+ tree of order 4. All data will be stored at the leaf nodes, and the smallest value in each leaf node (except the node in each of the most left branches) will be a key (with red font in Fig. 2.10).

- All the actual data are stored at the leaves.
- Each interior node contains M pointers to the children.
- $M - 1$ smallest key values in the subtrees except the 1st one.

- Each interior node has k children (k pointers), where $k \leq M$. From the 2nd to kth pointer, the smallest value of its children will be the key stored at this node. Thus, Each interior node has at most $M-1$ key values. The keys serve as speration values for the subtree. For example, if an internal node has k children, then it must have $k-1$ keys, such that all values stored in the left-most subtree will be no greater than key_0, where key_0 is the smallest key value. The values stored in the left-most + 1 subtree will be between key_0 and key_1, and so on.

The pseudo-code of INSERT a node in the B+ tree of order M is given as below.

```
Btree  Insert ( ElementType X,  Btree T )
{
   Search from root to leaf for X and find the proper leaf node;
   Insert X;
   While ( this node has no enough space  ) {
         split it into 2 nodes with ceil((M+1)/2) and floor((M+1)/2) keys,
         if (this node is the root)
            create a new root with two children;
         Copy the new key to the parent node
         /* (M+1)/2 th key when it is a leaf, M/2 th key when it is a interior node
         check its parent;
         /* if its parent has M keys, add M/2 key to the parent node */
   }
}
```

The running time of INSERT operation is $T(M,N) = O((M/\log M)\log N)$, the depth of $O(\log_{M/2} N)$. Searching operation is $O(\log N) = O(\log_{M/2} N)\log M$.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.