Red-Black Tree

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

October 14, 2024

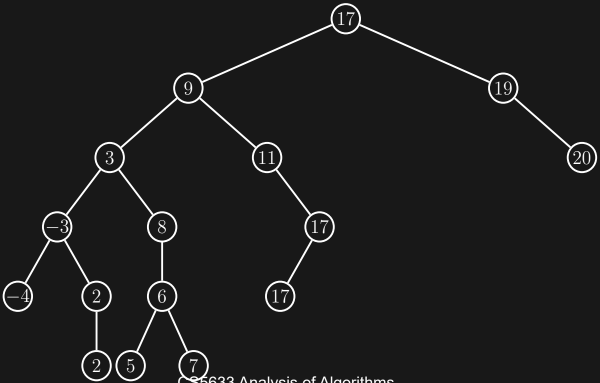
Binary Search Tree

Searching Algorithms

- We will now consider data structures which are used to store and lookup data in an efficient manner.
- We may want these data structures to be dynamic, that is, we should be able to insert and delete elements as needed.
- We want to be able to perform various operations on the data structure. For example:
  - Search
  - Max
  - Min
  - Insert
  - Delete
- We want these operations to be computed as quickly as possible.

Binary Search Tree

- A **binary search tree** is a special type of binary tree in which each node stores a value such that the following properties hold for a node x:
  1. For each node y in the left subtree of x, we have $y \leqslant x$.
  2. For each node y in the right subtree of x, we have $y \geqslant x$.
- An example of binary search tree (figure by Martin Thoma):

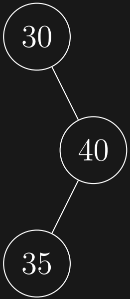


Using Binary Search Tree

- If a binary tree with n nodes has height $O(\log n)$, then we can find any node in the tree (or determine that it isn't there) in time $O(\log n)$.
- Since we want the tree to be dynamic, a “bad case” sequence of inserts can leave us with a tree of height n.
- We will consider trees which will “balance themselves” in the event that we see a bad sequence of inserts (or deletions).

A Bad Binary Search Tree

- A binary search tree may be slewed to have a height of $O(n)$. For example,



- A skewed binary tree takes $O(n)$ time to perform any type of search operations.

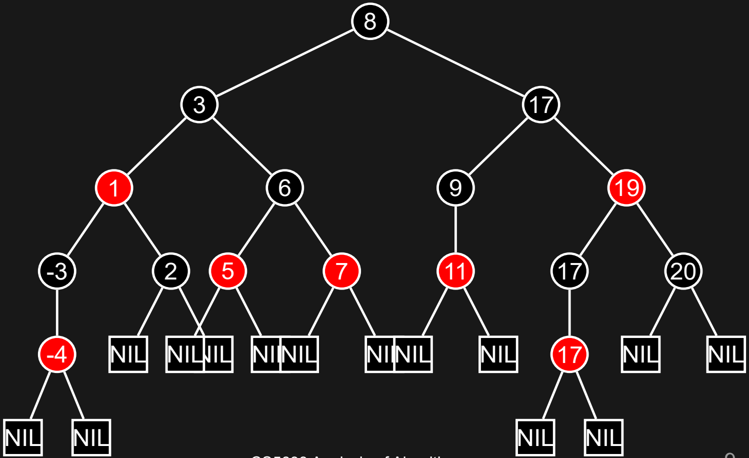
Red-Black Tree Basics

The Motivation for Red-Black Tree

- Red-black tree is a special binary tree that always has a $O(\lg n)$ height.
- With $O(\lg n)$ height, most operations on Red-black tree has a run time of $O(\lg n)$.
- Besides being a binary search tree, Red-black tree also has the following requirements:
  1. Every node has an extra field recording its color.
  2. A node is either marked with red or black.
  3. Root node is always black.
  4. Leaves are *Nil* nodes and are always black.
  5. If a node is red, its both children mush be black
  6. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

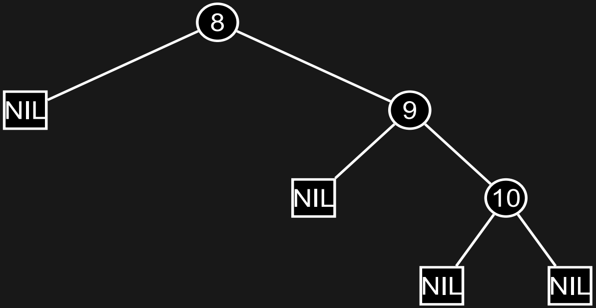
An Example of Red-Black Tree

- Note how this tree obeys the 5th and 6th requirements of Red-black tree (figure by Martin Thoma).



A Bad Red-Black Tree

- This tree violates the 6th requirement.

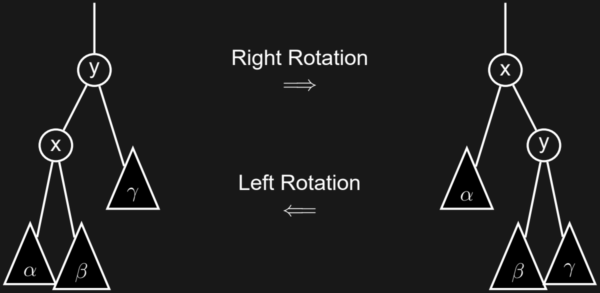


- This bad tree illustrates how 6th requirement eliminates the existence of a heavily skewed binary tree.

Red-Black Tree Maintenance: Insert

Red-Black Tree Rotation

- Before discussing the insert operation, we need to learn one operation in Red-black tree: Rotation.



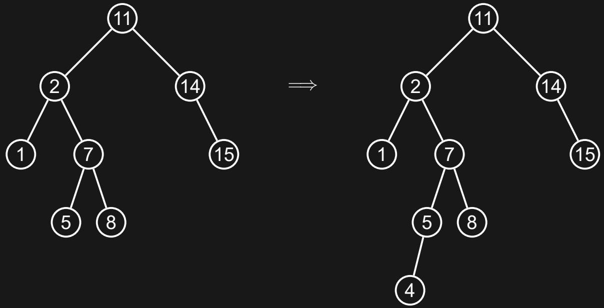
- The rotation operation is used later in Red-black tree insertion.

Maintaining a Red-Black Tree

- To ensure a Red-black tree always satisfies its requirements, special care must be taken when inserting new nodes or deleting old nodes from the tree.
  - Mostly, the goal is to ensure 5th and 6th requirements are met.
- The basic insert and delete operations are the same as the binary search tree (BST) insert and delete. We add a fix-up step after the BST insert/delete to ensure R-B tree's requirements are met.

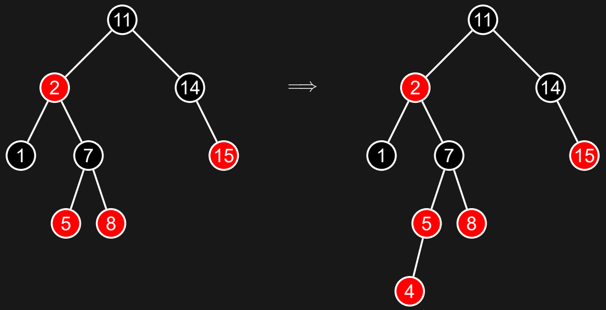
Binary Search Tree Insert

- The general algorithm traverses the tree to look for a leaf node to insert the new value.
- The new value will always be inserted into the tree as a leaf node.
- For example, inserting a 4 into the following tree gives:



Red-Black Tree Insert New Node

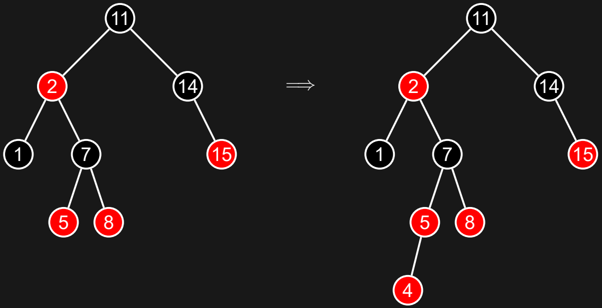
- Red-black tree insert first it performs a BST insert to insert a new node.
- The newly inserted node is marked as Red.
- For example, inserting a 4 into the following tree gives (Nil leaves are omitted):



- Note the new node “4” is red.

Red-Black Tree Insert Maintenance

- Marking a new node red may violate the 5th requirement – a red node's children must be black.
- In the previous insert example, both nodes "4" and "5" are red, violating the 5th requirement:

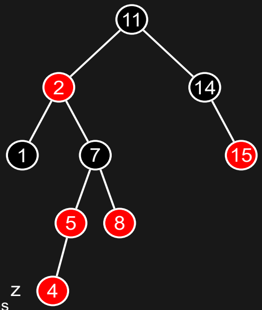


- Fix-ups must be conducted on the tree to fix the violations (no fix-ups if no violations).

Red-Black Tree Insert Maintenance: Case 1

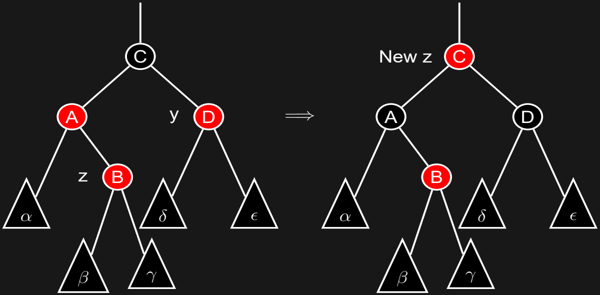
- Let the changed node be “z”.
  - Initially, the newly-inserted node is “z”.
  - As we will fix-up the tree recursively, other nodes may become “z”. The following discussion treat “z” as an arbitrary node, not just the newly-inserted leaf node.
- Depends on node “z”’s location and its neighbor nodes, there are three cases we need to consider and handle.

- In Case 1: z’s uncle is red.
- In the previous insert example, new node “4”’s uncle node “8” is also red, satisfying case 1.



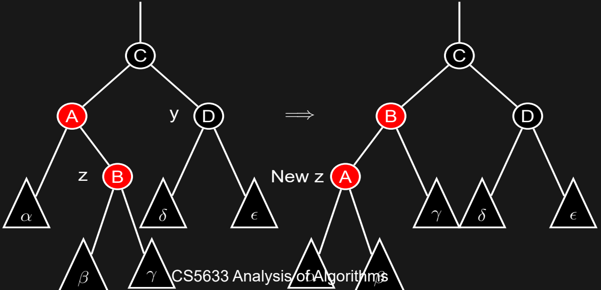
R-B Tree Insert Maintenance: Case 1 cont.

- For case 1, do the following operations:
  - Mark node “z”’s (node “B” in the figure) parent (node “A”) and uncle (node “D”) as black.
  - Mark node “z”’s grandparent (node “C” in the figure) as “red”. This ensures that node “A” and “B” still satisfy the 6th requirement.
  - Let node “z”’s grandparent be the new “z”, and recursively fix-up this new “z.”



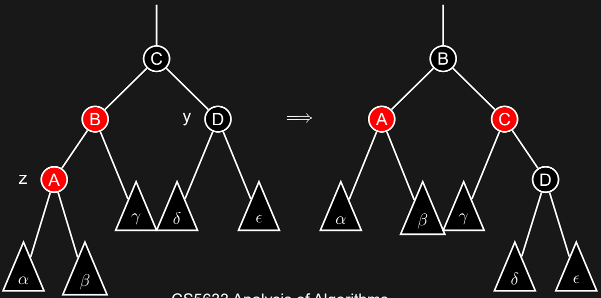
R-B Tree Insert Maintenance: Case 2

- Case 2: node "z" (node "B" in the figure) has right black uncle (node "y", aka node "D", rotation will be opposite if the uncle is left) and is a right child.
- For case 2, do the following operations:
  - Perform a left rotation on node "z" and its parent (node "A").
  - Let node "z"'s parent (node "A") be the new "z". This new "z" is for the following Case 3 operations.
  - Also perform the operations for Case 3. Case 2 always turns into a Case 3.



R-B Tree Insert Maintenance: Case 3

- Case 3: node "z" (node "A" in the figure) has right black uncle (node "y", aka node "D") and is a left child.
- For case 3, do the following operations:
  - Perform a right rotation on node "z"'s parent (node "B") and grand parent (node "C").
  - Mark node "z"'s parent (node "B") as black, and mark "z"'s old grandparent (node "C") as red.
  - Note that after fixing up Case 3, the tree already satisfies all requirements, i.e., not more fix-ups to do.



R-B Tree Insert Fixup Algorithm

- The algorithm for Insert Fixup is:

Algorithm 1: Red-black Tree Insert-Fixup.

```
Function Insert_Fixup(node z)
    if node z is black then
        return; // nothing to do;
    if node z satisfies Case 1 then
        perform Case 1 operations;
        z = z's grandparent; // get the new z;
        Insert_Fixup(z); // recursively fix up grandparent;
    else
        if node z satisfies Case 2 then
            perform Case 2 operations;
            z = z's parent; // get the new z;
        perform Case 3 operations;
    mark root node as black; // a special case where z is the root; no
      need to change other parts of the tree;
    return;
    ;
```

R-B Tree Insert_Fixup and Insert Run Time

- Case 2 and Case 3 terminate the algorithm.
- Recursion only happens in Case 1.
- The recursion will be performed at most $O(\lg n)$ times.
  - Each time a recursion happens, node "z" moves up two level in the tree.
  - The tree has $O(\lg n)$ height. Therefore, there are at most $O(\lg n)$ recursion.
- Based on the above, the run time of Insert_Fixup is $O(\lg n)$.
- A BST insert has $O(\lg n)$ time on R-B tree. Therefore the total run time for R-B insert, including BST insert and Insert_Fixup, is $O(\lg n)$.

Red-Black Tree Maintenance: Delete

Red-Black Tree Delete

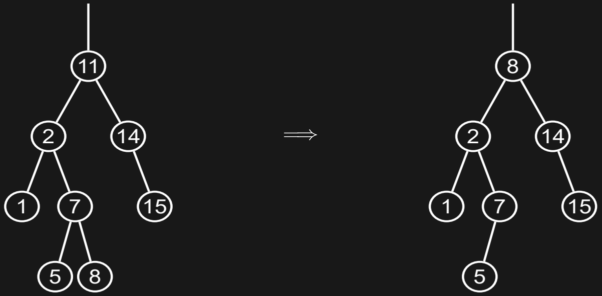
- Another common operation on a search tree is deleting a node.
- Similar to insertion, Red-black tree deletion also uses BST delete algorithm first. After deleting a node, fix-up operations are performed to ensure the new tree satisfies all requirements.

Binary Search Tree Delete

- First, search the tree to locate the node to delete.
- If the node is a leaf: simply remove it.
- If the node has one child: replace the node with its child and remove the child node.
- If the node has two children: find the predecessor (or successor) of the node, replace the node with its predecessor (or successor), and remove the predecessor or successor.
  - You should know in-order, pre-order and post-order tree traversal. For BST, in-order traversal gives an ascending list of the nodes.

Binary Search Tree Delete Example 1

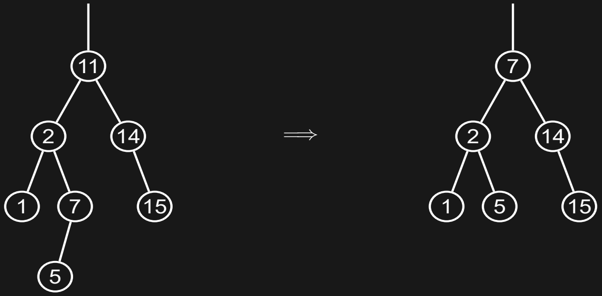
- For example, deleting 11 from the tree gives:



- Note the node 8 is used to replace node 11, and the original node 8 is removed from the tree.

Binary Search Tree Delete Example 2

- For example, deleting 11 from the tree gives:



- Note the node 7 is used to replace node 11. The original node 7 is removed from the tree. Node 5 is used to take node 7's place when node 7 is removed.
- Also note that the node actually got removed always have at most one child. For R-B tree, this means at most one non-Nil child.

Red-Black Tree Delete Fix-up

- If the node removed from the tree is a red node, no fix-up needs to be done. The tree still satisfies all requirements.

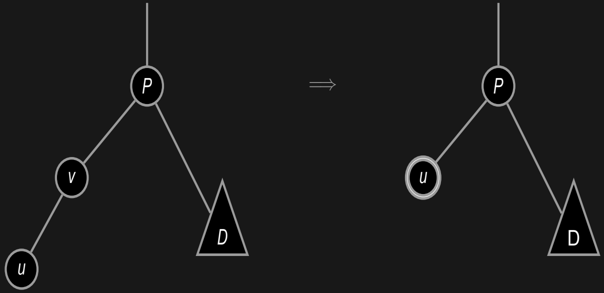
- If the node removed is a black node, it will certainly violates the 6th requirement. That is, all paths go through that black node will have their black height reduced by 1.
  - A fix-up is required for this scenario.
  - Similar to insert, there are four cases to consider.

Red-Black Tree Delete Fix-up

- If the node removed from the tree is a red node, no fix-up needs to be done. The tree still satisfies all requirements.
  - Removing a red node does not append a red node to another red node because the removed red node only has black children.
  - Removing a red node does not change the black height of any other nodes.
- If the node removed is a black node, it will certainly violates the 6th requirement. That is, all paths go through that black node will have their black height reduced by 1.
  - A fix-up is required for this scenario.
  - Similar to insert, there are four cases to consider.

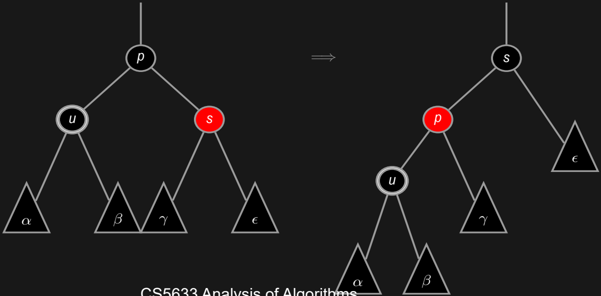
R-B Tree Delete Fix-up Node Removal

- Before we examine the cases, a special operation is conducted on the removed black node:
  - Remove the target black node and mark its child as double black.
  - Recall that initially, the removed node has at most one non-Nil child.
  - An illustration of the double black node, node *v* is removed and node *u* is marked double black:



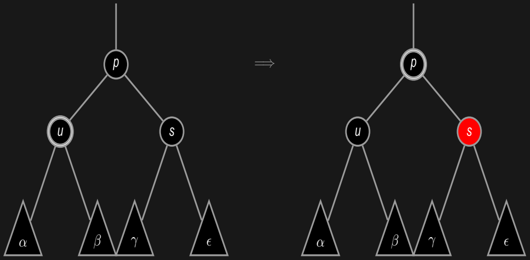
R-B Tree Delete Fix-up Node Case 1

- Case 1: the double black node *u* has a red sibling.
  - A left rotation (rotation is opposite if the sibling is on the left) on node *u*'s parent (node *p* in the figure) and *u*'s sibling (node *s*).
  - Mark *u*'s parent (node *p*) as red to maintain black height on its branch.
  - *u* remains double black, and recursively fix up *u*. Case 1 may be followed by case 2, 3, 4.
  - Note that subtree γ's black height should be 2-level larger than α or β due to *u* being double black.



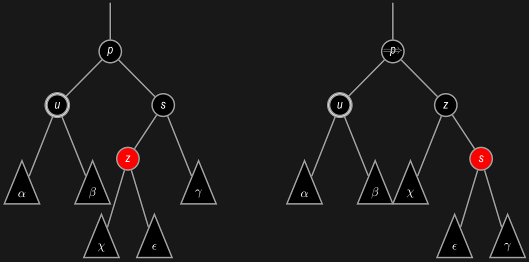
R-B Tree Delete Fix-up Node Case 2

- Case 2: the double black node *u* has a black sibling, and the black sibling has two black children.
  - Mark *u*'s parent (node *p* in the figure) as double black if the parent originally is black; or mark the parent black if the parent originally is red.
  - Mark *u*'s sibling (node *s* in the figure) as red.
  - Mark *u*'s as (single) black. The above three color changes maintain black heights on all branches.
  - Recursively fix up *p*, if *p* is double black.



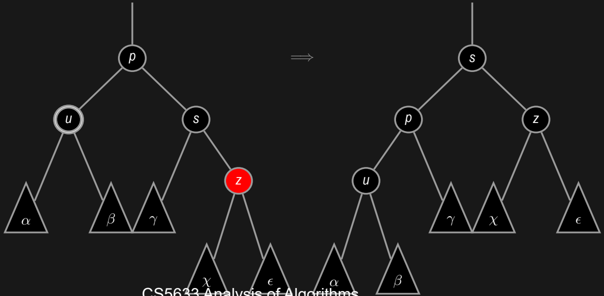
R-B Tree Delete Fix-up Node Case 3

- Case 3: the double black node u has a <u>black</u> sibling, and the black sibling 's left child is red, right child is black.
  - Do a right rotation on u's sibling (node z) and its red child (node s).
  - Recursively fix up the new double black node u. Case 3 is followed by case 4.
  - Note that the root of sub tree γ must be black, as in this case s has only one red child. The root of sub tree γ being black allows s to be marked red.



R-B Tree Delete Fix-up Node Case 4

- Case 4: the double black node *u* has a black sibling, and the black sibling's right child is red. The operations to handle this case are:
  - A left rotation on node *u*'s parent (node *p* in the figure) and *u*'s sibling (node *s*).
  - Mark the black sibling's red child (node *z*) as black to maintain the black heights on its branch.
  - Remove *u*'s double black, and the tree is fixed up.
  - Note that subtree γ's black height should be 1-level higher than α or β due to *u* being double black.



R-B Tree Delete Fixup Algorithm

- The algorithm for Insert Fixup is:

Algorithm 2: Red-black Tree Delete-Fixup.

```
Function Delete_Fixup(double black node u)
    if node u satisfies Case 1 then
        perform Case 1 operations;
        Delete_Fixup(z); // recursively fix up, to case 2, 3, or 4;
    else if node u satisfies Case 2 then
        perform Case 2 operations;
        mark u black;
        mark u's parent double-black;
        z = u's parent;
        Delete_Fixup(z);
    else if node u satisfies Case 3 then
        perform Case 3 operations;
        Delete_Fixup(z); // recursively fix up, to case 4;
    else if node u satisfies Case 4 then
        perform Case 4 operations;
        mark root node as black; // a corner case;
        return;
```

The Run Time of R-B Tree Delete_Fixup and Delete

- As case 4 terminates the algorithm, it only can execute once.
- As case 3 eventually converts to case 4, so it only can execute once.
- Case 1 converts to case 2, 3, 4.
  - It has constant execution time.
  - When it coverts to case 3 or 4, it can execute once.
  - When it converts to case 2, it can be viewed as a constant extra cost for case 2.
- Case 2 is the only case that actually occurs recursively.
  - Each call to case 2 goes up in the tree by one level.
  - Therefore, the recursion will be performed at most $O(\lg n)$ times.

The Run Time of R-B Tree Delete_Fixup and Delete cont.

- That is, the run time of Delete_Fixup has $O(\lg n)$ run time.
- BST delete has a cost of $O(\lg n)$. Therefore the total delete cost of R-B tree is $O(\lg n)$.

What Makes Red-Black Tree Balanced?

- It is easy to see that the 6th requirement makes Red-black tree balanced.
  - It is completely OK to have a Red-black tree with only black nodes.
- Then, what is the use of red node?
  - The red nodes are used to facilitate the fix-up procedure.