BSTs

-Items to the left of a given node are smaller

-Items to the right of a given node are larger

-external nodes: leaves, internal nodes: nodes that are not leaves (root is included)

-In the best case to search an element it takes O(1). Happens when the node we search is the root.

-In the average case to search an element (traversal), insertion, deletion all take O(h) time complexity where the height h is log n. -->when balanced

-In the worst case binary trees degenerate to lists with linear performance. it takes O(h) time complexity where the height h is n (the number of elements in the tree)--> when most unbalanced/skewed/degenerate (see slide 414)

-Insertion and deletion operations take per se O(1) time (but we still have to traverse to the position where we want to insert/delete the element so we have to add up that complexity)

-Insertion: done normally according to the properties of the BST

-Deletion: depends on how many children a node has:

3 cases:

1) Node t is a leaf node: since it has no children we simply remove it.

2a) Node t has only a right child: replace value of the node t with its right child and remove/free the right child

2b) Node t has only a left child: replace node t with its left child and remove/free the left child

3) Node t has two children: we find the minimum value in the right subtree (inorder successor) or the largest in the left subtree (inorder predecessor), replace the node t with the value of the inorder successor/predecessor and remove the inorder successor/predecessor

This procedure can also be performed with the greatest node in the left subtree (instead of the inorder successor)

Red black trees

Properties

-they are balanced Binary Search Trees (can't be skewed, if skewed we apply rotatins to rebalance them)

-nodes are either red or black

-the root and leaves (NIL) are black

-if a node is red, then all its children are black

-all paths from a node to its NIL descendants contain the same number of black nodes (node from where we start is not counted, NIL leaves are counted)-->black height of a node x

-Black height of tree = black-height of root (number of black nodes from root (excluded) to leaves (NIL included))

-external nodes: leaves, internal nodes: nodes that are not leaves (root is included)

-Smallest possible number of internal nodes in a rb tree with black-height k: (2^k) - 1

-Largest possible number of internal nodes in a rb tree with black-height k: (2^2k) - 1

Extra notes

-Red black trees guaranteed maximum height of 2log(n+1) -> h = O(log n)

-nodes require one storage bit to keep track of color

-O(h) for deletion, insertion and search/traversal, worst case h = log(n). (best case O(1))

-Very popular because of the worst-case guarantee

<u>Rotations</u>

-alters the structure of a tree by rearranging the positions of the nodes

-used for maintaining the properties of a red-black tree when they are violated by other operations such as insertion and deletion.

-do not affect the order of elements

-If in the left side there are too many nodes, we do a right rotation to balance the tree

-If in the right side there are too many nodes, we do a left rotation to balance the tree

-Have a time complexity of O(1): we are only changing 6 pointers

<u>Left Rotation</u>

The arrangement of the nodes on the right is transformed into the arrangements on the left node.

Procedure LR(p):

-If p's right child has a left subtree, we assign p as the parent of the left subtree of p's right child

-If the parent of p is NULL, set the root of the right subtree as the root of the tree.

Else if p is the left child of the parent, make root of right subtree as the left child of parent.

Else assign p's right child as the right child of p's parent

-Make the root of the right subtree as the parent of p

<u>Right rotation</u>

The arrangement of the nodes on the left is transformed into the arrangemenets on the right node

Procedure RR(p):

-If p's left child has a right subtree, we assign p as the parent of the right subtree of its left child

-If the parent of p is NULL, make p's left child as the root of the tree.

Else if p is the right child of its parent, make p's left child as the right child of p's parent.

Else assign p's left child as the left child of p's parent.

-Make p's left child as the parent of p.

Operations with Red and Black Trees

-insert and remove (both with time complexities O(h))

-Insert and remove may result in violation of red-black tree properties (to fix it we use the procedure for the different cases)

Insertions

-when inserting, we have to make sure that the properties of the red-black tree stay preserved

For each element we want to insert: we first insert the element in the right position according to the properties of the BST and we color it red,

and only after that we look at which case it is.

4 cases after we insert an element t and color it red:

Notation:

t: new node

p: parent of the element we inserted

g: parent of p or grandparent of t

u: uncle of t

-case 0: p is a left child and the color of p is black: no property violated so nothing to do

-case 1: p is a left child and t's uncle is red: set color of p and u as black and the color of g as red, then we set t = g (tree is now balanced, but g is now red so the rb property may be violated (propagate upwards with t = g since we arrive at root and then if it is red we set is as black since the root can't be red)

-case 2: p is a left child and t's uncle is black and t is a right child: LR(p) and then we set t = p

-> two adjacent red nodes are still there so we get case 3

-case 3: p is a left child and t's uncle is black and t is a left child: we set p->col = black, g->col = red and then we RightRotate(g) --> Tree rooted at p is now balanced

Mirror Cases: In case that p is a right child, all the above cases are handled similarly: exchange "left" and right (e.g if RR(g) change to LR(g) and viceversa)

Final Observations on Insertion:

-if two red nodes are adjacent, one does either restructure (with one or two rotations) and stop or propagate the adjacent red nodes upwards in the tree

-The number of propagations is max log n --> bound by O(log(n))

-The running time of insertion + fixing the color is thus O(log(n))

Deletions

Assumptions: x is a left child and x replaces the deleted node s

Notation:

-p: parent of x, b: brother of x, m: left child of brother, n: right child of brother case 0: if color of deleted node was red we don't have to do anything, else we simply set the color of x back to black

case 1: x is black and x's brother is red: b->col = black, p->col = red and then we LeftRotate(p) and we set b = m. Note that one gets either case 2,3 or 4 after rotating.

case 2: x's brother b is black and both m and n are black (left & right children of brother): b->col = red and we set x = p. Black height of both subtrees is reduced by one and we stop the deletion if p is red (we just set p->col = black)

case 3: x's brother b is black, m is red and n is black: m->col = black, b->col = red and we RR(b) and we set b = m. One gets in this way case 4.

case 4: x's brother b is black and b's right child n is red: b->col = p->col , p->col = n->col = black, and then we LeftRotate(p). The tree rooted at b is now balanced.

Mirror cases: in case that x is a right child, all the above cases are handled similarly: exchange "left" and right (e.g if LR(p) change to RR(p) and viceversa)

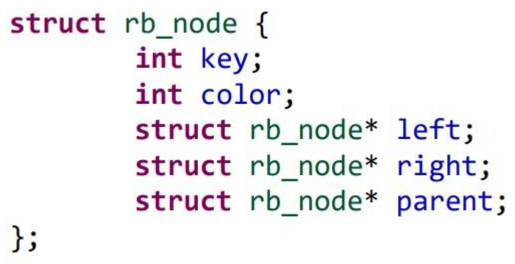
Final Observations on Deletion:

-The intuitive idea is to perform a color compensation (find a red node nearby and change its color into black)

-2 cases: restructing, recoloring: restructing resolves the problem locally, recoloring propagates it upwards

-The number of recolorings is max log n -> bound by O(log n)

-The running time of deletion is thus O(log n)



```
struct rb_node {
        int key;
        int color;
        struct rb_node* left;
        struct rb_node* right;
        struct rb_node* parent;
};
```

Rb trees implementation contains pointer to parent because we also need access to the grandparent of the node when inserting according to the different cases (and propagating upwards)