Generic Programming

Mini Project

Adarsh S 1PI14CS008
Shashank TD 1PI14IS060

Red-Black Trees

Introduction:

Although Binary Search Trees are efficient datastructures to store data, they tend to change from O(log n) to O(n) when data is stored in ascending or descending order.Hence, we use balanced Binary Search Trees such as AVL and Red-Black trees to implement the same, which guarantee O(log n) for insertion, deletion and search even in the worst case.

In Red-Black trees,the folllowing properties are to be maintained always:
i)root is always black
ii)parent and child cannot have red as color(Red-Red conflict)
iii)the number of black nodes in any path from root to null should all be the same.
Every Red Black Tree with n nodes has height<= 2log(n+1)

These trees have widely used in today's applications and is more importantly used in standard libraries
Java: java.util.TreeMap , java.util.TreeSet .
C++ STL: map, multimap, multiset.
Linux kernel: completely fair scheduler, linux/rbtree.h

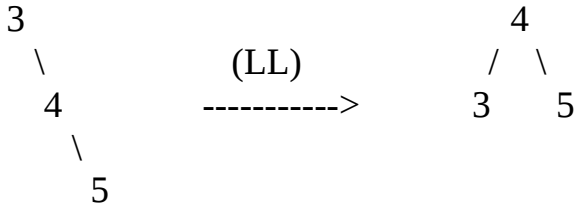
We have made a generic implementation of the above trees, such that it can store integers,double, float, char, string etc. based on comparison, because ultimately, it will be stored as a Binary Search Tree.
We have provided methods of insertion, deletion , searching and use of iterators to get an inorder traversal of the tree.

Mostly our code consists of 2 classes: the Node class with members such as data,left, right and additional fields for red black trees such as parent and color, and the Tree class having all the method declarations and implementations(It is here that we declare root variable as well).The main method then offer a list of methods to the user and call the methods accordingly.we have also provided a user defined predicate and will go through them during insertion/search. For now, we have used comparison and return -1,1 or 0 for <,> and == conditional operators in the red black tree, and string comparisons based on string length.

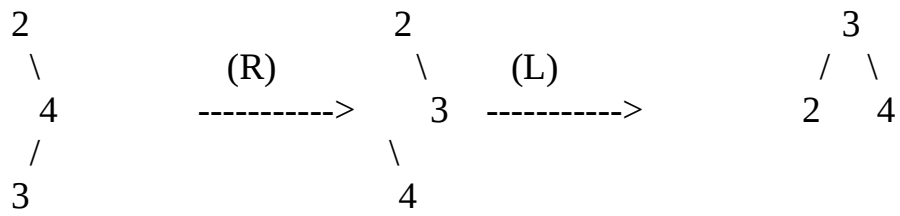
Making the Tree generic was not tough because ultimately any type that supports < ,> and == operators will be supported in the tree as well, as a Bianr Search Tree only requires those conditional operators.

Although the implementation of the trees differ, they primarily are based on 4 types of rotations- LL,LR,RR,RL.

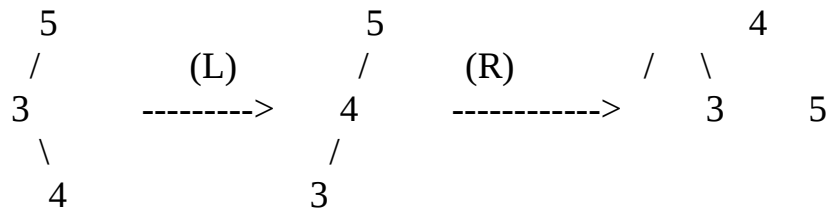
LL rotation:



RL:



LR:



RR:

Red Black Trees:

Insertion:

Upon standard BST insertion, and then we check for Red-Red violation, as an inserted node will always be of red color.

i) If the inserted node is root, we change it’s color to black.

ii) If the inserted node is the root's children, we do not have to change it's color at all,as the root will be black anyway.
When the inserted node is grandchild of root or lower, then check for the color of it's uncle node, and depending on whether it is :
- Red, then we change the color of the parent and uncle to black and the grandparent to red.
- Black,then we have 4 subcases based on:
- Parent node is Left child(L) of grandparent and inserted node is parent's left child(L)- Right rotation
- Parent node is right child (R) of grandparent and inserted node is parent's right child(L)-Left Right rotation
- Parent node is right child(R) of grandparent and inserted node is right child(R) of parent-Left rotation
- Parent node is right child of grandparent and inserted node is left child of parent-Right Left rotation
In all the cases, we swap the colors of the parent and grandparent.

Deletion:

Unlike checking the color of the uncle node in insertion, here we check the color of the sibling node to decide the case of rotation and recoloring.Deletion is slightly trickier than insertion because when we delete a black node, it's left and right subtrees now have one lesser black node and hence the property of number of black nodes being same is violated.
First, we perform standard BST deletion ,i.e. searching for the node to be deleted, deleting it and then replacing it with the next node in the inorder sequence(the left most child in the right subtree else right most child in left subtree, if we are connsidering successor).
i) If the node to be deleted is red, or the node to be replaced is red, then we replace the node by the next node in the inorder sequence and then recolor it to black.
ii)However, if both the node to be deleted and the node to be replaced is black, then we will get a case of double black, wherin we have to convert it into single black. If the node to be deleted was a leaf we can convert it back to single black, because we assume the null left and right node pointers as black only.
Else, we check the sibling s:
- If sibling is black and atleast one of the sibling's child is red, then we have 4 subcases:
    - Child is left child(L) of sibling which is the left child(L) of parent, or both children being red, in which case we perform Right Rotation.
    - Sibling is Left child(L) of parent and red node child is it's right child(R) so we perform LR rotation,i.e we convert the right child to left child, swap the child and it's parent and then perform R rotation.
    - Sibling is right child of parent and red node child is right child of sibling, here we perform L rotation.
    -Sibling is right child of parent and red node child is left child of sibling, so we perform RL rotation.

ii)if sibling is black and both its children are black,
    -if parent is red, we convert it into black.(red+double black=single black)
    -If parent is black, we recolor,i.e. we change the sibling's color to red and then recursively send the parent (now containing a double black) to remove violations.

iii)if sibling is red,and both its children black

-If sibling is left child of it’s parent, then we make R rotation

-If it is right child then we make an L rotation.

Perhaps the most intersting feature of our project is the use iteratord to get the elements of the tree in an order. For this, we send the left most child’s pointer of the tree to the iterator function, which then increments the pointer and since we have overloaded the increment operator, this should give us the pointer to the next successor node in the inOrder sequence. Similarly, another function also works for reverse access of a tree,i.e. we send the pointer to the right most child to the function, and then decrement it, wherin the decrement operator is overloaded so as to return the pointer to the predecessor node.So in total, we can say bidirectional iterator is supported here.
This is slightly tricky because to find the next/ previous node wrt a given node we have to handle fairly complex logic, although the process would become simpler if we perhaps stored them in an array, like Array implementation of BST.
We have provided methods for both pre increment and post increment.

For future enhancements, we thought of combining AVL and Red-Black trees or atleast some of it’s methods to maybe implement an even more efficient datastructure.

Thank You.