Generic Programming Project
Title : Generic Red Black Tree
Project ID : 4

Sai Eashwar K S PES1201801910

Monisha Chandra PES1201802102

Red Black Tree :

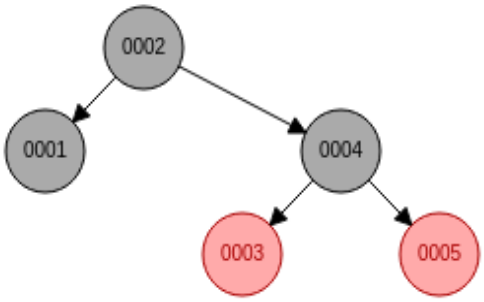
A red black tree is a special kind of self-balancing binary search tree. To ensure that the tree remains balanced, extra information called “colour” is stored in each node of the tree.

A red black is a binary search tree that has the following properties:

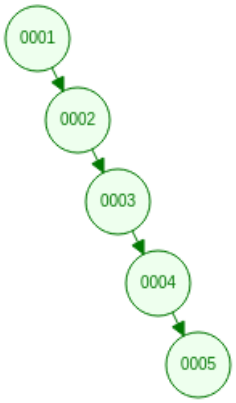
1. Each node is either red or black.
2. All NIL leaves are considered black.
3. If a node is red, then both its children are black.
4. Every path from a given node to any of its descendant NIL leaves goes through the same number of black nodes.

When the tree is modified, the new tree is rearranged with the help of rotations and repainted to keep the tree balanced. The rebalancing is not perfect, but guarantees searching in O(log n) time, where n is the number of nodes of the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O(log n) time.

Comparison of inserting 1, 2, 3, 4, 5 to a red black tree and a binary search tree.



Red black tree



Binary search tree

The main differences between red black trees and binary search trees lie in the insert and delete operations.

Insert

Algorithm:

1. Create a new node x with its colour set as red and perform standard binary search tree insertion.
2. If x is the root, change its colour to black. Insertion is complete.
3. If the colour of x's parent is red
    a. If x's Uncle is red
        i. Change colour of x's parent and uncle to black
        ii. Change the colour of x's grandparent to red.
        iii. Let new x be x's grandparent. Repeat steps 2 & 3 for new x.
    b. If x's Uncle is black
        i. Left left case (parent is to the left of grandparent, x is to the left of parent)
            1. Right rotate grandparent
            2. Swap colours of grandparent and parent
        ii. Left right case (parent is to the left of grandparent, x is right of parent)
            1. Left rotate parent
            2. Right rotate grandparent
            3. Swap colours of grandparent and x
        iii. Right right case (parent is right of grandparent, x is right of parent)
            1. Left rotate grandparent
            2. Swap colours of grandparent and parent
        iv. Right left case (parent is right of grandparent, x is left of parent)
            1. Right rotate parent
            2. Left rotate grandparent
            3. Swap colours of grandparent and x

Delete

To help with deletion, we use the concept of *double black.* When a black node is deleted and replaced by a black child, the child is marked double black. This would've violated Property 4 and rebalancing must be done.

Algorithm:

1. Perform standard binary search tree deletion. While performing, we end up having to delete only the leaf nodes or nodes with one child. This is because, even if we were to delete the internal nodes, they would be replaced by their

successor and delete would be called recursively on the successor. Let the node to be deleted be x and its child be y (x can be leaf, then y is NIL) which will be replacing x.

2. If x or y is red, make the replaced node's colour black, hence the black height doesn't change.
3. If both x and y are black, mark y as double black. We have to remove this double black.
    - 3.1. Do the following while y is black and it is not root. Let s be the sibling of y.
        - 3.1.1. If sibling is red
            - 3.1.1.1. Mark its parent as red.
            - 3.1.1.2. Mark the sibling as black.
            - 3.1.1.3. If the sibling is a left child, perform right rotation on the parent.
            - 3.1.1.4. If the sibling is a right child, perform left rotation on the parent.
            - 3.1.1.5. y would've moved up. Go to 3.1.
    - 3.2. If sibling is black
        - 3.2.1. If the sibling has a red child, let the red child be called r. There are 4 cases possible
            - 3.2.1.1. Left Left case : s is a left child and r is a left child (or both children of s are red).
                - 3.2.1.1.1. Change r's colour to black.
                - 3.2.1.1.2. Change s's colour to its parent's colour.
                - 3.2.1.1.3. Perform right rotation on s's parent.
            - 3.2.1.2. Right Left case : s is a right child and r is left child
                - 3.2.1.2.1. Change r's colour to that of s's parent.
                - 3.2.1.2.2. Perform right rotation on s.
                - 3.2.1.2.3. Perform left rotation on parent.
            - 3.2.1.3. Left Right case : s is a left child and r is a right child
                - 3.2.1.3.1. Change r's colour to that of s.
                - 3.2.1.3.2. Perform left rotation on s.
                - 3.2.1.3.3. Perform right rotation on parent.
            - 3.2.1.4. Right Right case : s is a right child and r is a right child
                - 3.2.1.4.1. Change r's colour to that of s.
                - 3.2.1.4.2. Change s's colour to that of its parent.
                - 3.2.1.4.3. Perform left rotation on parent.
        - 3.2.2. If both the children of s are black

3.2.2.1. Change s's colour to red.
3.2.2.2. If the parent is black, recur 3.1 on the parent.
3.2.2.3. Else, make the parent's colour black.

3.3. If y is root, make it black and stop.

Implementation

Template parameters

- T : Type of the data stored in the tree
- Compare : A binary predicate that takes two arguments and returns bool.

Member types

Member type	**Definition**	**Notes**
value_type	The first template parameter T	
value_compare	The second template parameter Compare	Defaults to less<value_type>
reference	const value_type&	const because
pointer	const value_type*	
iterator	A bidirectional iterator to const value_type	
difference_type	ptrdiff_t	

Note : the Iterator class in class RBT handles reverse iterator explicitly, it doesn't use std::reverse_iterator

Public functions (interface)

- **Class RBT<T, Compare>**

Constructors

RBT()	Default constructor
RBT(const RBT<T, Compare> & rhs)	Copy constructor

RBT(initializer_list<T> init_list)	Braced initialiser list constructor
template <typename InputIterator> RBT(InputIterator first, InputIterator last)	Range constructor
RBT<T, Compare> &operator=(const RBT<T, Compare> &)	Copy assignment

Destructor

~RBT()	Destructor

Iterators

Iterator begin() const	Return iterator to the first inorder element
Iterator end() const	Return iterator to nullptr (conceptually beyond the last node)
Iterator rbegin() const	Return a reverse Iterator to the last inorder element
Iterator rend() const	Return a reverse iterator to nullptr (conceptually beyond the first inorder element)

Capacity

int size()	Return the number of nodes
int leaf_count()	Return the number of leaves
int height()	Return height of the tree

Modifiers

void insert(T data)	Insert data into the tree
void remove(T data)	Delete the node containing data if it exists

Operations

Iterator find(T data) const	If data is present, return Iterator to that node, else return nullptr
vector<T> get_inorder() const	Returns a vector of data contained in inorder

vector<T> get_postorder() const	Returns a vector of data contained in postorder
vector<T> get_preorder() const	Returns a vector of data contained in preorder
Iterator max()	Return iterator to the max node (rightmost node)
Iterator min()	Return iterator to the min node (leftmost node)
template <typename T2, typename Compare2> friend RBT<T2, Compare2> &operator+(const RBT<T2, Compare2> &t1, const RBT<T2, Compare2> &t2)	Performs addition of trees (return a new tree with elements of t1 and t2 inserted)
void print_level_order()	Prints the data in level order

- **Operations of class Iterator of RBT**

Iterator(Node<T> *node_ptr, int is_reverse = 0)	Constructor
operator bool() const	Returns true if iterator points to a node, else false
const T& operator*()	Return the data contained by the node pointed to by the iterator
bool operator==(const Iterator& rhs)	Return true if both the iterators point to the same node, else false
bool operator!=(const Iterator& rhs)	Return negation of operator==
Iterator& operator++()	Pre increment. Make iterator point to its inorder successor and return it
Iterator operator++(int)	Post increment
Iterator& operator--()	Pre decrement. Make iterator point to its inorder predecessor and return it
Iterator operator--(int)	Post decrement

Genericity

Our implementation of red black trees are generic i.e., the tree can have nodes that can store any type of data. The two template parameters of the tree are T and Compare where T is the datatype of the data being stored and Compare is a functor class using which the users can provide custom predicates used while inserting and searching for data in the tree. The default type of Compare is std::less<T>.

Execution

List of files :

1. rbt.h
   Contains all the necessary classes (Node, RBT and iterator) and their definitions.
   Note : if DEBUG flag is true, the rotations performed during insert are displayed
2. client.cpp
   Client code, includes rbt.h.
3. make.mk
   A build automation tool

Steps for execution:

1. $ make -f make.mk (Linux)
   $ ./a.out
   OR
2. Compile client.cpp and run the executable

References

1. https://en.wikipedia.org/wiki/Red%E2%80%93black_tree
2. https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
3. https://www.cs.usfca.edu/~galles/visualization/BST.html
4. https://www.geeksforgeeks.org/red-black-tree-set-2-insert/
5. https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/