A Project Report

On

Concurrent Implementation of Red Black Trees.

BY

**A.Vinay Kumar : 18XJ1A0502**

**A.Sai Charan : 18XJ1A0503**

**M.Abhinav Reddy : 18XJ1A0528**

Under the supervision of

**DR DR Praveen kumar Alapati**

**SUBMITTED IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS OF**

**PR 301: 3rd Year Team Project**

cid:image003.png@01D6594A.2604D070

**ECOLE CENTRALE SCHOOL OF ENGINEERING**

**HYDERABAD**

**(May 2021)**

ACKNOWLEDGMENTS

First and foremost we want to express gratitude toward our instructor and guide PROF. PRAVEEN KUMAR ALAPATI who gave us his significant ideas and thought when we needed them. We would likewise express gratitude toward to every one of them who aided us to finish this project. We are immensely grateful to all associated with this project as without their inspiration and significant idea it would not have been possible to develop the project inside the recommended time.

cid:image003.png@01D6594A.2604D070

**Ecole Centrale School of Engineering**

**Hyderabad**

Certificate

This is to certify that the project report entitled “ **Concurrent Implementation of Red Black Trees”** submitted by **Mr. A Vinay Kumar(18XJ1A0502), Mr. A Sai Charan (18XJ1A0503)** and **Mr. M Abhinav Reddy(18XJ1A0528)** in partial fulfillment of the requirements of the course PR 301, Project Course, embodies the work done by him/her under my supervision and guidance.

**(Prof . Praveen kumar Alapati)**

Ecole Centrale School of Engineering, Hyderabad.

Date:

ABSTRACT

We present concurrent Red Black Tree algorithms that explicitly maintain logical ordering information in the data structure, permitting clean separation from its physical tree layout. We capture logical ordering using intervals, with the property that an item belongs to the tree if and only if the item is an endpoint of some interval. We are thus able to construct efficient, synchronization-free and intuitive lookup operations.

We implemented our Red Black Tree using Logical Ordering and evaluated the running time for insertion ,Deletion and Contains Operations.

CONTENTS

Title page………………………………………………………… …….1

Acknowledgements……………………………………………………..2

Certificate…………………………………………………………….....3

Abstract………………………………………………………………....4

1.Chapter 1 Introduction……………………………………………......6

2.Chapter 2 Project Proposals…………………………………………..8

3.Chapter 3 Background and Related work ……………………………12

4.Chapter 4 Implementation…………………………………………….16

5.Chapter 5 Results………………………………………………………17

6.Chapter 6 Conclusion………………………………………………….18

7.Future Work ……………………………………………………………19

8.Chapter 7 References…………………………….…………………….10

**CHAPTER 1: INTRODUCTION**

**1.1 Importance of Concurrent DataStructuers**

In Recent past , chip manufactuers are moving towards simultaneous multhithreaded architechture because of it’s advantages in simultaneously utilizing and sharing of multiple resources , such as , ALUs ,Memory hierarchy etc. Concurrent data structures are a fundamental building block for leveraging modern multi-core processors. Recent years have seen rising interest in scalable and efficient concurrent algorithms for data structure.In this paper we will discuss about concurrent algorithm for Red Black Tree.

**1.2 Real world Applications of Red Black Tree**

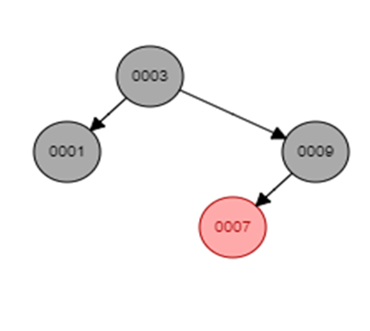
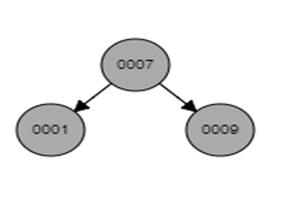
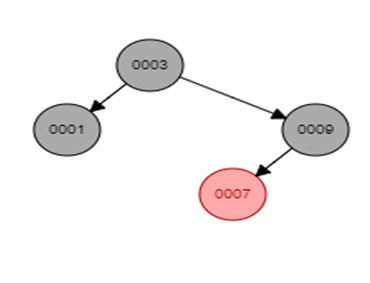
Red Black trees are used in many real-world libraries as the foundations for sets and dictionaries and are common in the Linux kernel. For example in a process schedulers or for keeping track of the virtual memory segments for a process. They are also used in map, multimap, multiset from C++ STL and java.util.TreeMap , java.util.TreeSet from Java. Besides they are use in K-mean clustering algorithm for reducing time complexity. MySQL uses Red-Black Trees for indexes on tables.As these Red Black Trees are used in schedulers it is very important to implement concurrently and efficiently.

**1.3 Key Challenges**

A Red Black Tree data structure supports the operations insert, delete, and contains with their standard meaning. Any correct BST algorithm must preserve two invariants: (i) the Red Black Tree does not contain duplicate keys, and (ii) the tree follows the standard Red Black Tree structural layout.A key challenge in designing correct and efficient concurrent Red Black Tree algorithms is to devise a scalable design for the lookup operation. This operation is invoked by all three operations (insert,Delete,Contains) to check whether a given element exists in the tree. In this project we have designed a lookup operation using logical order.

To illustrate the Difficulty for Lookups in Red Black Tree,while the tree is mutated with concurrent opertions .for example if T1 performing contains(7) in figure 2 and T2 performing delete(3)

|  |  |
| --- | --- |
| T1(contains(7)) | T2(delete(3)) |
| Intial pointer is at 3 then Pointer moves to 9 now if T2 start then T1 Suspended  Resumes it’s operation now pointer is on 9 as shown in figure 2,which is last internal node so,it can’t find 7 But 7 is present . | Perform Delete(3)  After deleting 3 our tree look like figure 3  completed |

****In this paper we present simple intuitive look up operation and this look up operation is lock free.we discuss the idea below.

T1

T1

Figure 3

Figure 2

Figure 1

**1.4 Key Idea**

We Discuss idea through Example,consider Tree as shown in Figure 1 , This Res Black Tree represents a set of integers {1,3,7,9} where elements can be ordered with there key values 1<3<7<9.The logical ordering of elements can be maintained explicitly in the data structure, and this important property enables us to find the successor and predecessor of a node without traversing pointers along the tree layout. Because logical ordering is stable under layout manipulations (such as balancing), lookup operations can proceed concurrently with operations that mutate the layout.

|  |  |
| --- | --- |
| T1(Contains(7)) | T2(Delete(3)) |
| Intial pointer is at 3 then Pointer moves to 9 now if T2 start then T1 Suspended the  However, when the tree is equipped with ordering, after T1 thread reaches node 9 and learns that its left child is null, it looks up the predecessor of 9, finding that 7 < 9 is in the ordered set, meaning that 7 is in the tree, and thus contains(7) correctly returns true. | Perform Delete(3)  After deleting 3 our tree look like figure 3  Completed The logical order look like {1,7,9} |

For our example, these are the pairs: (−∞, 1),(1, 7),(7, 9),(9, ∞). These pairs can be viewed as intervals where a key belongs to the set if it is an endpoint of some interval and does not belong to the set otherwise.

**CHAPTER 2 : PROJECT PROPOSAL**

Why Red-Black Tree?

a)Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.

The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that the height of the tree remains O(log n)

after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations. The height of a

Red-Black tree is always O(log n) where n is the number of nodes in the tree.

why red black tree is prefered over avl tree?

a)The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.

So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred.

**2.1 Problem Defination :**

Concurrent data structures are a fundamental building block for leveraging modern multi-core processors. Recent years have seen rising interest in scalable and efficient concurrent algorithms for data structure,so it is very important to implement concurrent datastructuers efficiently .As Red black Tree Datastructure used in many real world applications like used in schedulers and are also used in map, multimap, multiset from C++ STL and java.util.TreeMap , java.util.TreeSet from Java.red black tree operations (insertion , deletion ,contains) time complexity is O(logn) , implementing concurrent red black trees efficiently can fast up the schedulers time and other running times which uses multimap, multiset from C++.our main aim is to implement red black trees concurrently .

**CHAPTER 3 : BACKGROUND AND RELATED WORK**

**3.1 What is Red Black Tree ?**

A Red black tree is BST(Binary Search Tree) where each node stores extra information about it’s color , which can be either Red or Black. Each node of the tree now contains the attributes color, key, left, right, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves, As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL . For a red-black tree T , the sentinel T.nil is an object with the same attributes as an ordinary node in the tree. Its color attribute is BLACK, and its other attributes—p, left, right, and key—can take on arbitrary values. All pointers to NIL are replaced by pointers to the sentinel T.nil.

**3.2 Red Black Tree Properties**

A red-black tree must satisfies the following red-black properties:

1. Every node is either red or black.

2. The root is black.

3. Every leaf (NULL) 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

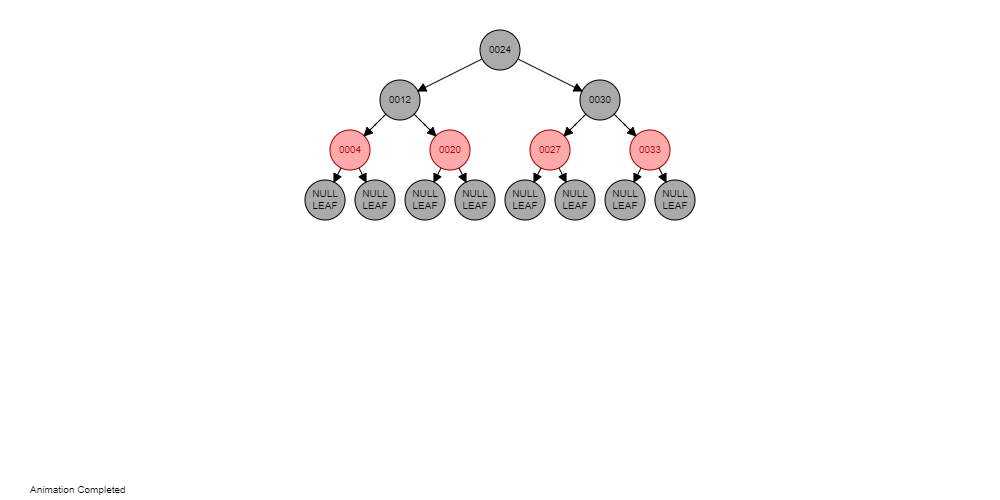
same number of black nodes.

Figure 4 Example of red black tree

Every property mentioned above are satisfying in the Figure 4 , This example of Red black .

**3.3 Operations on Red Black Tree:**

Red Black Tree supports three operations

* Insert
* Delete
* Contains

**3.3.1 Insert Opertion**

We can insert a node into an n-node red-black tree in O(logn) time, to insert node z into the tree T as if it were an ordinary binary search tree, and then we color red.

When we colour our inserted node Z there will be some conflicts , if z’s parent colour is black then there will be no-coflict because all properties mentioned in 3.2 are satisfied after inserting . if z’s parent is red then the property 4 mention in 3.2 is not satisfying so,to maintain the properties we need to do some rotations and recoloring.

If the z’s parent color is red then we have 2 cases they are

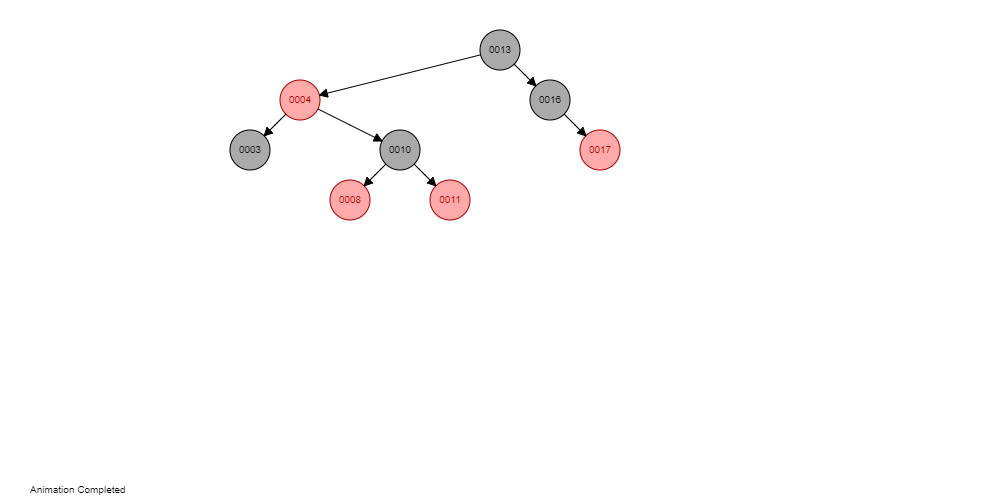
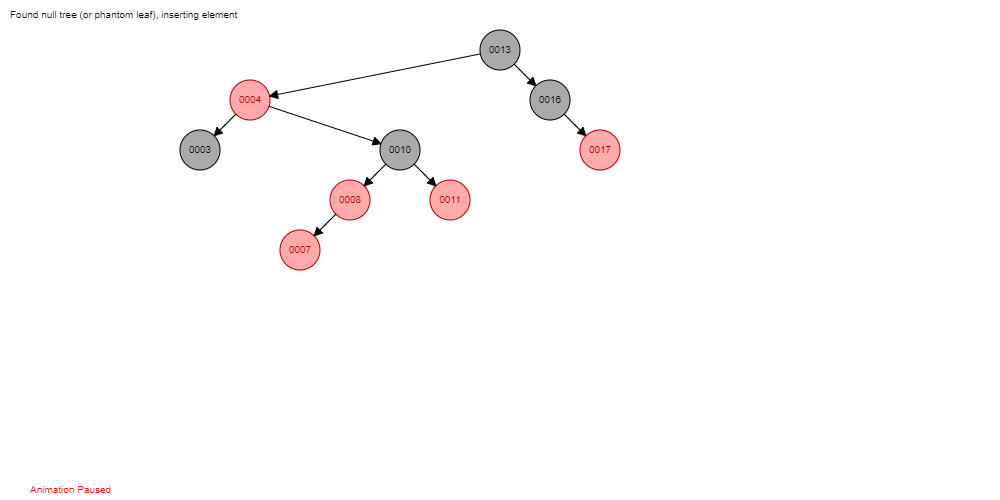
1. z’s parent is left child
2. z’s parent is right child

These both cases are symmetric and each of the above cases we have 3 more cases .

If z’s parent is left child then

1. z’s uncle y is red
2. z’s uncle y is black and z is right child
3. z’s uncle y is black and z is left child

And if z’s parent is right child the cases are same as above.

Now we will look into example for each cases.

y

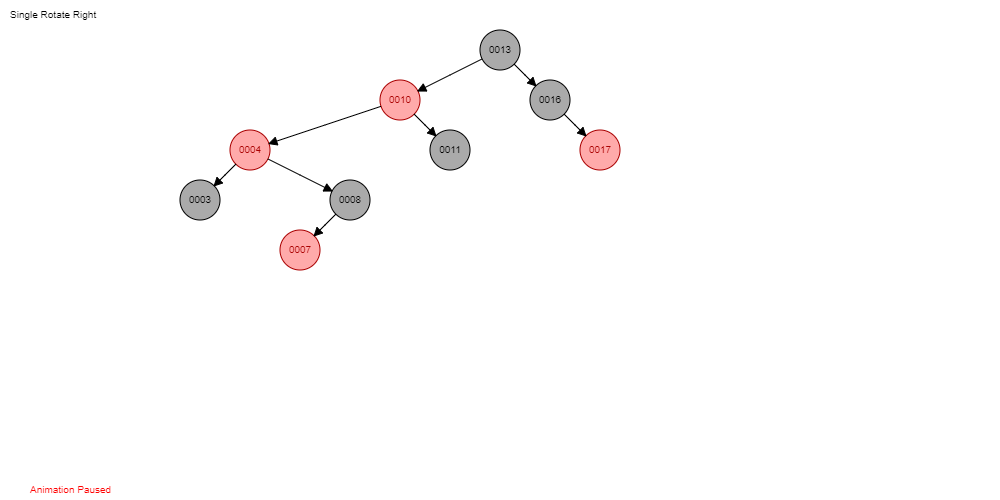
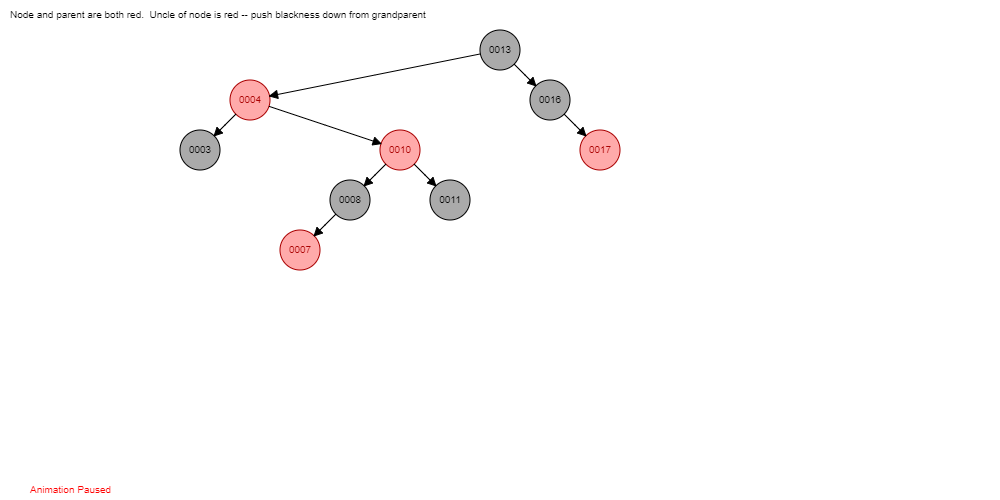
Z

Figure 6

Figure 5

Insert(7)

After inserting 7, Tree shown in (Figure 6 )violating the property 4 mentioned in 3.2 . z’s uncle y color is red so it fall under case I.we need to follow the steps (5-8) mentioned in fixInsert

****

y

Z

Figure 8(Case 3)

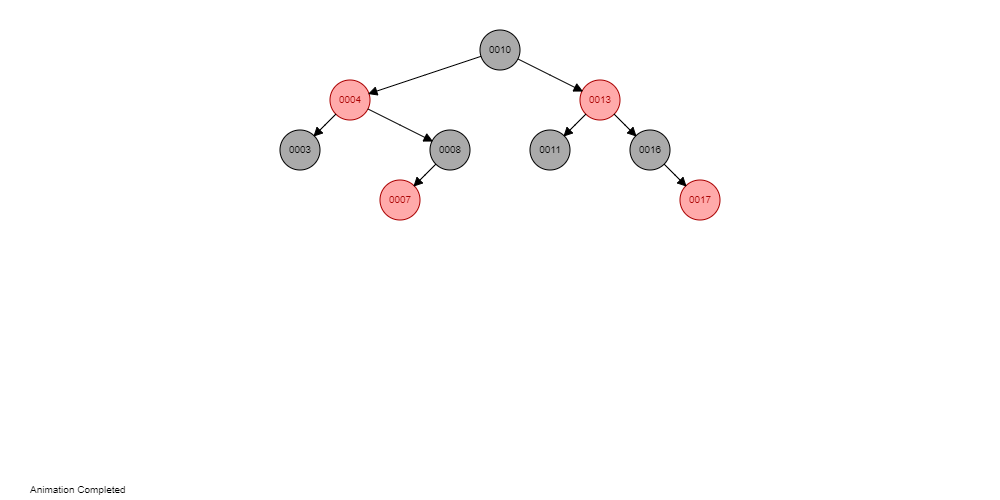
Figure 7(Case 2)

z’s uncle y is black and z is right child

Z

y

z’s uncle y is black and z is left child

****

Tree in figure-9 satisfies all properties mentioned in 3.2.

Figure 9

Z

Maintaining the Predecessor and Successor We now describe how pred and succ are maintained. In the following, we use Zk to denote a node with key k. Insert In a Red Black Tree, a new node, Z, is inserted as a child of either its predecessor, p, or its successor, s. Thus, N can access and set p and s using its parent’s pred and succ pointers.For example consider figure 5 we are inserting 7, z’s parent in 8 and 8’s succ and pred values are Z10 and Z4  so Z7 pred is 8’s pred i.e Z4 , Z7 succ is Z8.

* + 1. **Delete Opertion**

We can Delete a node from red black tree in O(logn).when we are deleting a node from a tree we need to make sure all red black tree properties must satisfy .

If we want to delete node z , we need to check wether node z has one child or two childs,if that node has one child then transplant with it’s child and if it has two children then we need to find it’s successor and then transplant,There wil only conflict when we are deleting node z with color black ,if it’s red directly we can delete.

y is z initially

Finally, if node y was black, we might have introduced one or more violations of the red-black properties, and so we call FIXDELETE in to restore the red-black properties. If y was red, the red-black properties still hold when y is removed or moved, for the following reasons:

1. No black-heights in the tree have changed.

2. No red nodes have been made adjacent. Because y takes z’s place in the tree, along with z’s color, we cannot have two adjacent red nodes at y’s new position in the tree. In addition, if y was not z’s right child, then y’s original right child x replaces y in the tree. If y is red, then x must be black, and so replacing y by x cannot cause two red nodes to become adjacent.

3. Since y could not have been the root if it was red, the root remains black

If node y was black, three problems may arise, which the call of fixDelete will remedy. First, if y had been the root and a red child of y becomes the new root, we have violated property 2. Second, if both x and x.p are red, then we have violated property 4. Third, moving y within the tree causes any simple path that previously contained y to have one fewer black node. Thus, property 5 is now violated by any ancestor of y in the tree.

If x color is black then

1. X is left child
2. x right child

in each case we 4 more cases which are symmetric.

If x is left child

1. x’s sibilling w is red
2. x’s sibling w is black, and both of w’s children are black
3. x’s sibling w is black, w’s left child is red, and w’s right child is black
4. x’s sibling w is black, and w’s right child is red.

Maintaing predecessor and successor , We now describe how pred and succ are maintained.if Zk is deleted node with key k then maintaining pred and succ value shown below

(Zk.pred).succ = Zk.succ and (Zk.succ).pred = Zk.pred

* + 1. **Contains**

Contains take O(logn) to serch for a key .

In a concurrent setting, such traversal may lead to incorrect results due to concurrent mutations of the tree. To avoid this problem, we rely on the following observation: to determine whether k is present in the tree, it is enough to have two keys, k1, k2, such that

1. k1 and k2 are in the tree,
2. (ii) for every ˜k ∈ (k1, k2),˜k is not in the tree, and

(iii) k ∈ [k1, k2].

Using the logical ordering and the above observation, we can determine whether a key k is in the tree as follows:

• If k was found during traversal, then k is in the tree.

• If k was not found, then we must find two keys, k1 and k2, that are in the tree and such that k ∈ (k1, k2). The search for k terminates when it reaches a node of value ˜k that lay at the end of the scanned path. If there are no concurrent updates, ˜k is either k’s predecessor or successor; thus,

one of the following holds:

(i) k ∈ (pred(˜k),˜k), or

(ii) k ∈ (˜k,succ(˜k)).

In the presence of concurrent updates, k1 and k2 must be found, which will be done via the pred and succ pointers.

**CHAPTER 4:IMPLEMENTATION**

**4.1 The Node Data Structure**

class **Node** {

**int** data;

**Node** parent;

**Node** left;

**Node** right;

**int** color;//1 red 0 black

**Node** pred;

**Node** succ;

}

This is the node structure our tree , In addition to the fields of a standard Red Black Tree node, it contains pointers to the predecessor (pred) and successor(succ) of the node.

**4.2 search and Contains**

**4.2.1 search**

This method is used in all three operation insert,delete and contains.search(k) will return a node with the value is equal to k(if exsist). if k value doesn’t exsist in the tree it will return sentinel node(TNULL).

**searchTreeHelper**(Node node, int key) {

**if** (node == TNULL || key == node.data) {

**return** node;

}

**if** (key < node.data) {

**return** **searchTreeHelper**(node.left, key);

}

**return** **searchTreeHelper**(node.right, key);

}

**Search**(int k){

**return** **searchTreeHelper**(this.root,k)

}

**4.2.2 contains**

**contains**(int key){

**Node** node = **search**(key);

**while**(node.data > key){

node = node.pred;}

**while**(node.data<key){

node = node.succ; }

**return** (node.data == key);}

The contains operation,begins by calling search(k). If the returned node has key k, then contains returns true. If the node has a key different than k, then two nodes are required to determine whether k is in the tree, Nk1 and Nk2, that hold k ∈ (k1, k2] and succ(Nk1 ) = Nk2 . If k2 = k (i.e., k was found), To obtain Nk1 and Nk2 , the contains operation uses the node returned by search. It then traverses using the pred field until reaching the first node whose key is not greater than k. Once discovered, it scans nodes using the succ field, until reaching a node with a key equal to or greater than k. The last iteration of this loop reads Nk1 ’s succ field and saves Nk2 as required.

**4.2.3 Insert**

**insert**(**int** key) {

**Node** node = new **Node**();

**Node** y = **search**(k);

**Node** temp;

node.parent = y;

**if** (y == null) {

root = node;

node.pred=TNULL;

node.succ=TNULL;

} **else** **if** (node.data < y.data) {

y.left = node;

temp = y.pred;

y.pred = node ;

node.succ = y;

node.pred = temp;

} **else** {

y.right = node;

temp = y.succ;

y.succ = node;

node.pred = y;

node.succ = temp;

}

**if** (node.parent == null) {

node.color = 0;

**return**;

}

**if** (node.parent.parent == null) {

**return**;

}

**fixInsert**(node);

}

**fixInsert**(**Node** k) {

**Node** u;

**while** (k.parent.color == 1) {

**if** (k.parent == k.parent.parent.left) {

u = k.parent.parent.right;

**if** (u.color == 1) {

u.color = 0; //**case1**

k.parent.color = 0; //**case1**

k.parent.parent.color = 1; //**case1**

k = k.parent.parent; //**case1**

} **else** {

**if** (k == k.parent.right) {

k = k.parent; //**case2**

leftRotate(k); //**case2**

}

k.parent.color = 0; //**case3**

k.parent.parent.color = 1; //**case3**

rightRotate(k.parent.parent); //**case3**

}

}

} **else** {

(same as then clause with “right” and “left” exchanged)

}

if (k == root) {

break;

}

}

root.color = 0;

}

When insert is called it calls search(k) if the Node with key k exsist it will return or it will resume it’s process and each and every cases(case 1, case 2,case 3) are illustrated with examples clearly in section in 3.3.1.

we capture the logical ordering via a set of intervals: {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}. With each interval (p, s). The intervals can be split, upon insertion.upon splitting an interval (p, s) to (p, k) and (k, s).

An update of (p, s) to (p, k),(k, s) is applied as follows:

1. Nk is created with pred set to Np and succ set to Ns.

2. Np’s succ and Ns’s pred are updated to Nk.

**4.2.4 Deletion:**

Delete(int key){

Node x, y;

Node z = searchTree(key);

Node node = this.root

if (z.data != key) {

return;

}

Node temp =z;

Node k1,k2;

k1=temp.pred;

k2=temp.succ;

if(k1!=TNULL && k2!=TNULL){

k1.succ=k2;

k2.pred=k1;

}

else if(k1==TNULL){

k2.pred=k1;

}

else if(k2==TNULL){

k1.succ=k2;

}

y = z;

contuning right side

int yOriginalColor = y.color;

if (z.left == TNULL) {

x = z.right;

rbTransplant(z, z.right);

}

else if (z.right == TNULL) {

x = z.left;

rbTransplant(z, z.left);

}

else {

y = z.succ;

yOriginalColor = y.color;

x = y.right;

if (y.parent == z) {

x.parent = y;

} else {

rbTransplant(y, y.right);

y.right = z.right;

y.right.parent = y;

}

rbTransplant(z, y);

y.left = z.left;

y.left.parent = y;

y.color = z.color;

}

if (yOriginalColor == 0) {

fixDelete(x);

}

}

rbTransplant(Node u, Node v) {

if (u.parent == null) {

root = v;

} else if (u == u.parent.left) {

u.parent.left = v;

} else {

u.parent.right = v;

}

v.parent = u.parent;

}

The rbtransplant is used to exchange the deleted node with it’s right or left child if deleted node has one child and if node has two children we exchange with it’s successor.

fixDelete(Node x) {

Node s;

while (x != root && x.color == 0) {

if (x == x.parent.left) {

s = x.parent.right;

if (s.color == 1) {

s.color = 0; //case 1

x.parent.color = 1; //case 1

leftRotate(x.parent); //case 1

s = x.parent.right; //case 1

}

if (s.left.color == 0 && s.right.color == 0) {

s.color = 1; //case 2

x = x.parent; //case 2

} else {

if (s.right.color == 0) {

s.left.color = 0; //case 3

s.color = 1; //case 3

rightRotate(s); //case 3

s = x.parent.right; //case 3

}

s.color = x.parent.color; //case 4

x.parent.color = 0; //case 4

s.right.color = 0; //case 4

leftRotate(x.parent); //case 4

x = root;

}

} else {

(same as then clause with “right” and “left” exchanged)

}

x.color = 0;

}

we capture the logical ordering via a set of intervals: {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}. With each interval (p, s). The intervals can be merged , Upon merging two intervals (p, k),(k, s) to (p, s).

An update of (p, k),(k, s) to (p, s) is applied as follows:

1. Nk is removed .This also serves as an indication that (k, s) is removed.

2. Np’s succ is set to Ns and Ns’s pred is set to Np

**CHAPTER 5:RESULTS**

The table give below is the average running time for insertion deletion and search when number of nodes to be inserted or deleted are varying .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number of nodes | Insertion(in ns) | Inserion(in ms) | Deletion(in ns) | Deletion(in ms) |
| 10 | 39750 | 0 | 47550 | 0 |
| 100 | 134650 | 0 | 258100 | 0 |
| 1000 | 2506950 | 2 | 1735950 | 1 |
| 10000 | 4602750 | 4 | 7394600 | 6.5 |
| 100000 | 40218100 | 40 | 22777700 | 22 |
| 1000000 | 381963800 | 381.5 | 108951650 | 108.5 |

|  |  |  |
| --- | --- | --- |
| Number of nodes | Contains (in ns) | Contains(in ms) |
| 10 | 8700 | 0 |
| 100 | 53800 | 0 |
| 1000 | 1533800 | 1 |
| 10000 | 1321100 | 1 |
| 100000 | 11248300 | 11 |
| 1000000 | 75664300 | 75 |

**CHAPTER 6:CONCLUSION**

We presented sequential implementation of red black trees using logical ordering. The core idea behind our algorithms is the notion of logical ordering, which is explicitly maintained in the tree. We leveraged this idea to design an intuitive, simple lookup operation, which is also lock-free. We have implemented our algorithms and we have calculated running time of the different operations (insert, delete and contains) shown in Results section(Chapter 5).

In a concurrent setting, normal(general) traversal may lead to incorrect results due to concurrent mutations of the tree. To address this challenge, some concurrent trees maintain all values in the leaves thus never changing the location of an element yet others use some form of notification such as version numbers or node marking to detect concurrent updates during lookup. While these approaches differ on the exact details of how they synchronize, they all base their synchronization on the tree layout. Using logical ordering we have designed simple lookup operation which is lock-free (illustrated with example in 1.3 and 1.4 sections).

**FUTURE WORK:**

We have implemented sequential implemetation of red black trees using logical ordering , In future we will be implementing concurrent red black tree via logical ordering .

Our synchronization is based on locks, where each node can be locked in two separate layouts:

• The tree ordering layout

• The tree physical layout

Each update operation is applied in four steps:

1. Acquire ordering layout locks.

2. Acquire physical layout locks.

3. Update the ordering layout and release ordering locks.

4. Update the physical layout and release physical locks.

Locking the node in the tree’s physical layout prevents concurrent updates to the node’s physical layout information, that is, the node’s children, parent and color information.we will reentrant lock to lock a particular node.

**Insertion:**

we capture the logical ordering via a set of intervals: {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}. With each interval (p, s). The intervals can be split, upon insertion.upon splitting an interval (p, s) to (p, k) and (k, s).

An update of (p, s) to (p, k),(k, s) is applied as follows:

1. (p, s)’s lock is acquired.

2. Nk is created with pred set to Np and succ set to Ns.

3. Np’s succ and Ns’s pred are updated to Nk.

**Deletion:**

we capture the logical ordering via a set of intervals: {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}. With each interval (p, s). The intervals can be merged , Upon merging two intervals (p, k),(k, s) to (p, s).

An update of (p, k),(k, s) to (p, s) is applied as follows:

1. (p, k)’s and (k, s)’s locks are acquired.

2. Nk is removed .This also serves as an indication that (k, s) is removed.

3. Np’s succ is set to Ns and Ns’s pred is set to Np

**REFERENCES:**

1.ppopp14-trees Practical Concurrent Binary Search Trees via Logical Ordering

2. Introduction\_to\_algorithms-3rd Edition by THOMAS H.CORMEN

3. <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

**4.** Fast Concurrent Lock-Free Binary Search Trees Aravind Natarajan Neeraj Mittal

5. <http://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Red-Black-Trees>