Concurrent Implementation of Red Black Trees

Teja Sri Dharma Reddy Vanukuri[1], Bala Akash Mutthavarapu[2], Sai Teja Vadranam[3], Shaik Sohail[4], K.B.V.Brahma Rao[5], S.Hrushi Kesava Raju[6]

[1] C.S. Student, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation (KLEF), Vaddeswaram, Guntur Andhra Pradesh, India. *klucse2000031042@gmail.com*

[2] C.S. Student, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation (KLEF), Vaddeswaram, Guntur Andhra Pradesh, India. *klucse2000030663@gmail.com*

[3] C.S.Student, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation (KLEF), Vaddeswaram, Guntur Andhra Pradesh, India. *klucse2000031054@gmail.com*

[4] C.S.Student, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation (KLEF), Vaddeswaram, Guntur Andhra Pradesh, India. *klucse2000031733@gmail.com*

[5] Professor, Department of Computer Science & Engineering, Koneru Lakshmaiah Education Foundation, Vaddeswaram, Guntur Andhra Pradesh, India. *brahmaraokbv@kluniversity.in*

[6] Professor, Department of Computer Science & Engineering, Koneru Lakshmaiah Education Foundation, Vijayawada, India. *hkesavaraju@kluniversity.in*

Abstract: We offer Red Black Tree concurrent algorithms that logical ordering information is clearly maintained in the data structure, allowing clear separation from its physical tree architecture. With the property that an item only belongs to the tree if and only if it is an endpoint of an interval, we represent logical ordering using intervals. Thus, we are able to design lookup operations that are quick, simple, and devoid of synchronization. In this paper we implemented our Red Black Tree using Logical Ordering and evaluated the running time for insertion, Deletion and Contains Operations.

Keywords: Red black trees , leaf, node, delete, insert.

1. INTRODUCTION

1.1 Importance of Concurrent Data Structures

In Recent past, chip manufactures are moving towards simultaneous multithreaded architecture because of it's advantages in are simultaneously utilizing and sharing of multiple resources , such as , ALUs ,Memory hierarchy etc. Concurrent data structures are an essential building component for taking use of contemporary multi-core CPUs. In recent years, there has been a surge in interest in scalable and efficient concurrent data structure techniques. In this work, we will look at a concurrent method for the 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

The operations insert, delete, and contains are supported by a Red Black Tree data structure with their usual meanings. The Red Black Tree has to avoid duplicate keys and adhere to the Red Black Tree's typical structural arrangement in order for the BST algorithm to be correct.Making the lookup procedure scalable is a fundamental difficulty in creating correct and effective concurrent Red Black Tree algorithms. Each of the three operations (insert, delete, and contains) call this operation to determine whether a particular element is present in the tree. In this project, we used logical order to build a lookup operation.

To illustrate the Difficulty for Lookups in Red Black Tree, while the tree is mutated with concurrent operation's .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

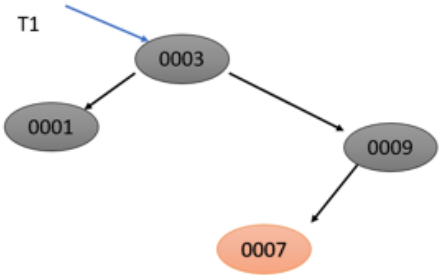

Figure 1

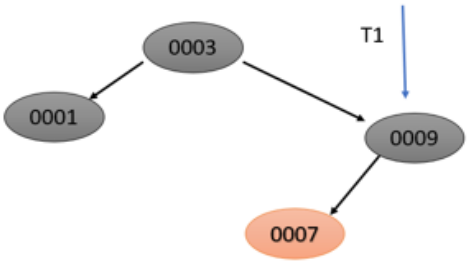

Figure 2

Figure 3

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

1.4 Key Idea

We discuss an idea using the example of a tree in Figure 1. This Res Black Tree depicts a collection of integers 1, 3, 7, and 9, where elements can be arranged according to their key values of 1<3<7<9.We may discover the successor and predecessor of a node without traversing pointers throughout the tree architecture thanks to the essential attribute that the logical ordering of elements can be maintained explicitly in the data structure. Lookup operations can be carried out simultaneously with operations that change the layout because logical ordering remains stable under layout modifications (such as balancing).

T1(Contains(7))	T2(Delete(3))
Initial pointer is at 3 then Pointer moves to 9 now if T2 start then T1 Suspended the If the tree has ordering, however, and T1 thread reaches node 9 and discovers that its left child is null, it searches up the predecessor of 9, discovering that 7< 9 is in the ordered set, indicating that 7 is in the tree, and as a result 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}

These are the pairs that make up our example: (-∞,1),(1, 7),(7, 9),(9,∞). These pairings can be thought of as intervals where a key only joins the set if it is the endpoint of an interval and otherwise does not.

2. PROJECT PROPOSAL

Why Red-Black Tree?

a) The majority of BST operations (such as search, max, min, insert, delete, etc.), where h is the BST's height, take O(h) time. For a skewed Binary tree, the cost of these operations can increase to O(n). We can provide an upper bound of O(log n) for all of these operations if we make sure that the height of the tree stays O(log n) after each insertion and deletion. A Red-Black tree's height is always O(log n), where n is the tree's node count.

Why red black tree is preferred over AVL tree?

a) Compared to Red-Black Trees, AVL Trees are more evenly distributed, although they may result in more rotations during insertion and deletion .Red-Black trees are therefore chosen if your application includes frequent insertions and removals.

2.1 Problem Definition

A crucial building component for utilizing contemporary multi-core computers is concurrent data structures. It is crucial to implement concurrent data structures effectively since there has been an increase in interest in scalable and effective concurrent algorithms for data structures in recent years. As Red black Tree Data structure 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.

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. Currently, the properties color, key, left, right, and p are present in each node of the tree. A node's associated pointer attribute has the value NIL if either its parent or child are not present. These NILs will be regarded as leaf pointing indicators. We utilize a single sentinel to represent NIL out of convenience when dealing with boundary conditions in red-black tree code. The sentinel T.nil for a red-black tree T is an object having the same properties as a typical tree node. Its key, p, left, and right attributes can all have arbitrary values, and its color attribute is BLACK. Pointers to the sentinel T.nil are used in place of all pointers to NIL.

3.2 Red Black Tree Properties

A red-black tree must satisfy 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.

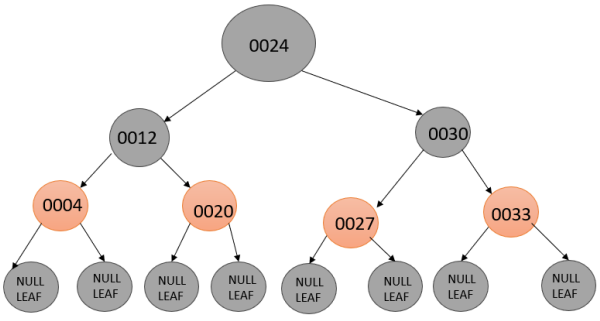

Figure 4 Example of red black tree

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 Operation

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

I. z's uncle y is red.

II. z's uncle y is black, and z is right child

III. 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 examples for each cases.

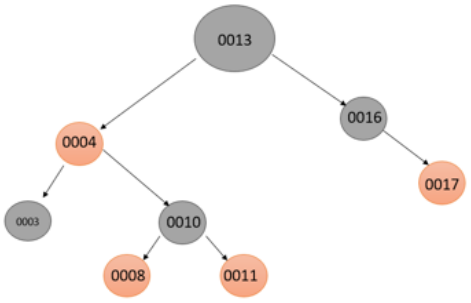

Figure 5

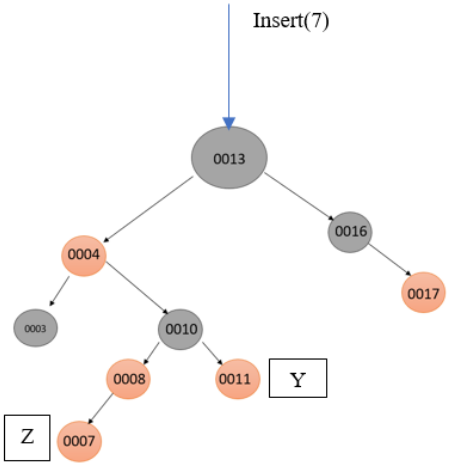

Figure 6

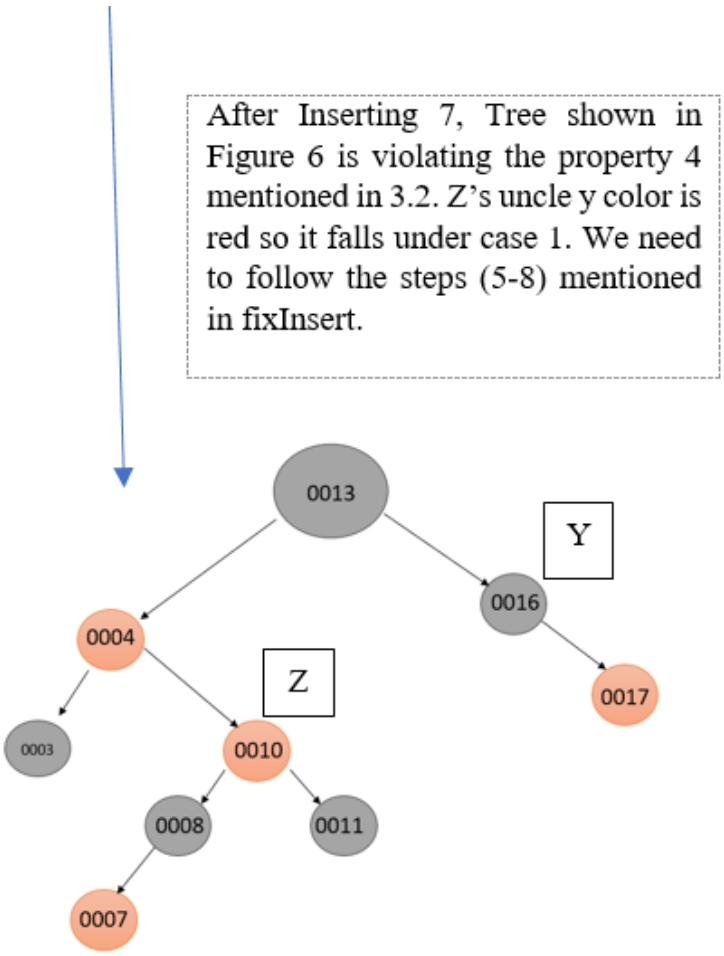

Figure 7 (Case 2)

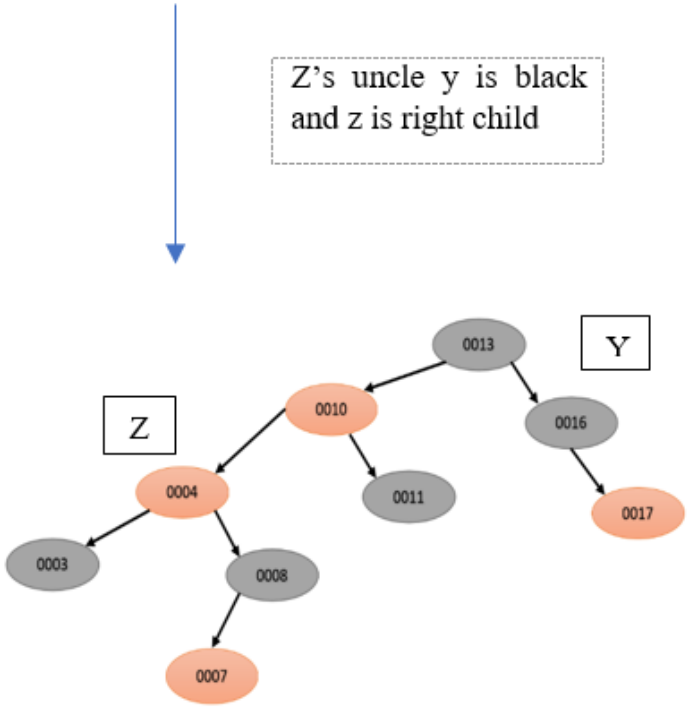

Figure 8 (Case 3)

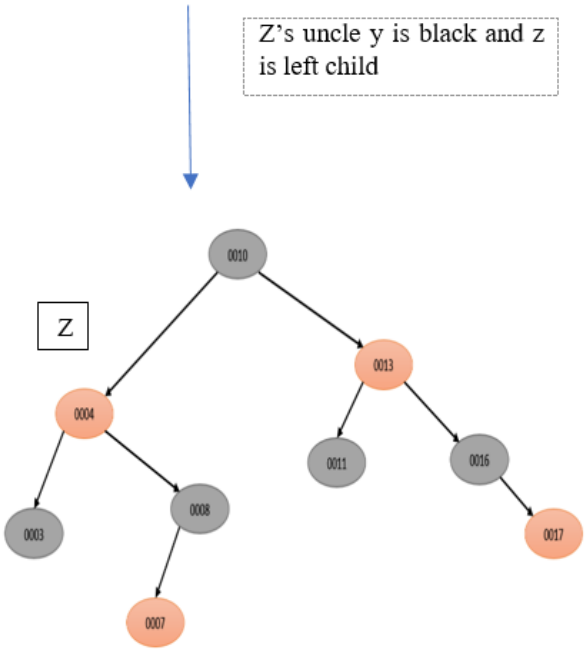

Figure 9

Tree in Figure 9 satisfies all properties mentioned in 3.2

Keeping the Successor and Decedent We will now go through how pred and succ are kept up. The node with key k is indicated in the following by the symbol Zk. Insert A new node, Z, is added to a Red Black Tree as a child of either its predecessor, p, or its succeeding node, s. N can therefore use its parent's pred and succ pointers to access and set p and s. For instance, have a look at picture 5, where we are adding 7. Z's parent is 8 and Z's succ and pred values are Z10 and Z4, respectively.

3.3.2 Delete Operation

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 last but not least, if node y had been black, we might have introduced one or more violations of the red-black characteristics, so we call FIXDELETE to fix them. When y is removed or moved, the red-black qualities still apply if y is red for the following reasons:

1. The tree's black heights have not changed.
2. There are no nearby red nodes. We cannot have two neighboring red nodes at y's new position in the tree since y inherits both z's position and color. Furthermore, if y wasn't z's right child, x, who was originally y's right child, takes y's position in the tree. Since x must be black if y is red, y must be replaced by x cannot cause two red nodes to become adjacent.
3. The root is still black because y could not have been the root if it had been red.

Three issues could develop if node y was black, which the call to fixDelete will repair. First, we have violated property if y had been the root and a red child of y became new root. 2. If x and x.p are both red, we have broken the property.4.Third, each sample path that previously contained y has one less dark node because of the movement of y within the tree. Any ancestor of y in the tree is now in violation of property 5 as a result.

If x color is black then

1. X is left child
2. X right child\

In each case we have 4 more cases which are symmetric.

If x is left child

I. x's sibilling w is red

II. x's sibling w is black, and both of w's children are black

III. x's sibling w is black, w's left child is red, and w's right child is black

IV. x's sibling w is black, and w's right child is red.

Maintain predecessor and successor, we now describe how pred and succ are maintained. If Z_k is deleted node with key k then maintaining pred and succ value shown below

(Z_k.pred).succ = Z_k.succ and (Z_k.succ).pred = Z_k.pred.

3.3.3 Contains

Contains take O(log n) to search for a key .

Due to concurrent tree modifications in a concurrent environment, such a traversal may produce inaccurate results. We base our solution on the following finding to get around this issue: Two keys, k1, k2, are sufficient to establish whether k is present in the tree.

(i) k1 and k2 are in the tree,

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

(iii) (iii) k ∈ [k1, k2].

We may determine whether a key k is in the tree by using the logical ordering and the observation mentioned earlier as follows:

• If k was discovered during traversing, k is present in the tree.

• If k was not discovered, then we need to locate two keys in the tree, k1 and k2, such that k (k1, k2). When it arrives at a node with value k that was at the end of the scanned path, the search for k comes to an end. One of the following is true if there are no concurrent updates and k is either k's predecessor or successor:

(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.

4. IMPLEMENTATION

4.1 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 of our tree; it includes references to the predecessor (pred) and successor (succ) of the node in addition to the fields found in a typical Red Black Tree node

4.2 Search and Contains

4.2.1 Search

```
Search(int k){
 return searchTreeHelper(this.root,k)
}
```

```
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);
}
```

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).

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, N_{k1} and N_{k2}, that hold $k \in (k1, k2]$ and $succ(N_{k1}) = N_{k2}$. If k2 = k (i.e., k was found), To obtain Nk1 and Nk2 , The node that search returned is used in the contains operation. Once it has reached the first node whose key is not greater than k, it continues to traverse using the pred field. Once found, it uses the succ field to scan nodes until it finds one with a key equal to or greater than k. This loop's final iteration stores Nk2 as needed after reading the succ field from Nk1.

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;
 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;
}
```

Through a series of intervals, {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}, we are able to represent the logical ordering. With each (p, s) interval. When two intervals (p, k), (k, s), and (p, s) are combined, the intervals can be combined.

The following is the update of (p, k),(k, s) to (p, s):

1. Nk is taken out.This also shows that (k, s) has been eliminated.

2. Np's succ and Ns's pred are both set to Np.

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

No. of nodes	Insertion (in ns)	Insertion (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

6. CONCLUSION

We presented sequential implementation of red black trees using logical ordering. The notion of logical ordering, which is explicitly maintained in the tree, is the fundamental concept behind our algorithms. We used this concept as a springboard to create a lock-free, logical, and straightforward lookup method. We have put our algorithms into practice and calculated the running times for the various operations (insert, delete, and contains), which are displayed in the Results section of Chapter 5.

Due to simultaneous tree modifications in a concurrent context, normal(general) traversal may produce inaccurate results. In order to overcome this difficulty, some concurrent trees keep all values in the leaves, never changing an element's position, while others employ some kind of signal, like version numbers or node marking, to identify concurrent updates while doing a lookup. The tree layout serves as the common synchronization point for all of these systems, despite differences in the specifics of how they synchronize. We have created a straightforward, lock-free lookup process using logical ordering (examples are provided in sections 1.3 and 1.4).

7. FUTURE WORK

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

We use locks to synchronize, and there are two different ways to lock a node:

• 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.

By locking a node in the physical layout of the tree, concurrent alterations to the node's children, parents, and color information are prevented. We'll use a re-entrant lock to secure a specific node.

Insertion

Through a series of intervals, {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}. we are able to represent the logical ordering. With each (p, s) interval. When inserted, the intervals can be divided.after dividing a gap (p, s) into (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

Through a series of intervals, {(p, s) | Np, Ns ∈ tree ∧ ∀k ∈ (p, s), Nk ∈/ tree}. we are able to represent the logical ordering. With each (p, s) interval. When inserted, the intervals can be divided.after dividing a gap (p, s) into (p, k) and (k, 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

8. REFERENCES

[1] Rudolf Bayer. Symmetric binary B-trees: data structure and maintenance algorithms. Acta Informatica, 1:290–306, 1972.

[2] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 257–268, Bangalore, India, January 2010. ACM

[3] Karimov, Elshad. (2020). Red–Black Tree. 10.1007/978-1-4842-5769-2_12.

[4] W. PUGH, Skip lists: A probabilistic alternative to balanced trees, in "Algorithms and Data Structures, Proceedings WADS '89" (F. Dehne, J.-R. Sack, and N. Santoro, Eds.), pp. 437-449, Lecture Notes in Computer Science, Vol. 382, Springer-Verlag, Berlin, 1989.

[5] L. J. GUmAS AND R. SEDGEWICK, A dichromatic framework for balanced trees, in "Proceedings, Nineteenth Annual Symposium on Foundations of Computer Science," pp. 8-21, IEEE Comput. Soe. Press, Long Beach, CA, 1978.

[6] H. T. KUNG AND P. L. LEHMAN, A concurrent database manipulation problem: Binary search trees, in "Proceedings, Fourth International Conference on Very Large Data Bases," p. 498, IEEE Cornput. Soc. Press, Long Beach, 1978 (abstract); full version in Concurrent manipulation of binary search trees, ACM Trans. Database Systems 5 (1980), 354-382

[7] Besa, Juan & Eterovic, Yadran. (2013). A Concurrent Red Black Tree. Journal of Parallel and Distributed Computing. 73. 434–449. 10.1016/j.jpdc.2012.12.010.

[8] Mailund, Thomas. (2021). Red-Black Search Trees. 10.1007/978-1-4842-7077-6_15.

[9] Baldan, Paolo & Corradini, Andrea & Esparza, Javier & Heindel, Tobias & König, Barbara & Kozioura, Vitali. (2012). Verifying red-black trees.

[10] Elmasry, Amr & Kahla, Mostafa & Ahdy, Fady & Hashem, Mahmoud. (2019). Red-Black Trees with Constant Update Time. Acta Informatica. 56. 10.1007/s00236-019-00335-9.

[11] Hanke, S. & Ottmann, Th & Soisalon-Soininen, E.. (2006). Relaxed Balanced Red-Black Trees. 10.1007/3-540-62592-5_72.

[12] Kahrs, Stefan. (2001). Red-black trees with types. Journal of Functional Programming. 11. 10.1017/S0956796801004026.

[13] Djamel Eddine, Zegour. (2021). Improving the Red-Black tree delete algorithm. 10.21203/rs.3.rs-1194654/v2.

DOI: https://doi.org/10.15379/ijmst.v10i1.2618