Kasdi Merbah University - Ouargla

Faculty of New Technologies of Information and Communication

Department of Computer Sciences and Information Technology

ACADEMIC MASTER

Domain: Mathematics and Computing Sciences

Field: Computing Science

Major: fundamental

Title

CONCURRENT SKIP LIST SET TO A CONCURRENT AVL TREE ALGORITHM

Submitted by: Kalboddine Mohammed Charif Kabdi

President:	Bachir Said
Examiner:	Hamida Djediai
Supervisor:	Saleh Easchi

Academic Year: 2019/2020

Dedication

To my parents

To my family and my friends for all their support

To all my instructors teachers for all the instructing instructions and guidance

Contents

Table of Contents

الملخص

تتميز قائمة التخطي المتزامنة بالفهرسة أو المستوى، والتي تتحكم في عملية البحث بسرعة وبشكل متزامن، ولكن عندما تتعامل قائمة التخطي المتزامنة هذه مع الكثير من البيانات، تصبح الفهرسة المتعلقة بها غير متوازنة، مما يؤدي إلى نوع من البطء في تنفيذ معظم العمليات. السؤال الذي يطرح نفسه هل يمكننا حل مشكلة التوازن هذه؟

للإجابة على هذا، أخذنا في الاعتبار فرضيتين، إما اعادة برمجة الفهرسة بطريقة تجعلها أكثر توازنا أو استبدالها بهيكل بيانات آخر.

ولأسباب عديدة أخذنا الفرضية الثانية لأننا نعتقد أنها الخيار الأفضل المتاح امامنا من أجل تحسين جودة الخدمات المقدمة سواء في قواعد البيانات أو أي مكون لنظام يتطلب السرعة والدقة فإن بنية بيانات شجرة AVL المتزامنة هي الخوارزمية اللازمة لبناء نظام متكامل يعتمد على الدقة والسرعة بشكل متزامن.

وعليه فان هذا المشروع يحتاج كلاً من بنية البيانات وتقنيات التزامن من أجل إنجازه

على عكس خوارزميات التزامن الأخرى، تقدم هذه الخوارزمية احسن تقنيات متوفرة في البرمجة التزامنية التي وفرتها لغة جافا وتشمل هذه: خيط واحد مسؤول عن إعادة التوازن، بالإضافة الى :

شجرة AVL متزامنة تتعامل مع القيم الصحيحة.

يبدأ الخيط الدوري المسؤول عن إعادة التوازن بعد 500 عملية.

مفهوم الخيط الآمن الذي قدمه بريان جويت في كتاب التزامن عمليًا (concurrency in practice by brian Goetz).

قفل متفائل كأسلوب مهم للتعامل مع المزامنات المختلفة.

Abstract

The concurrent skip list Characterized by indexation or level, which Contributes to search faster in different operation concurrently, but when this concurrent skip list handle lot of data, the indexation related to it becomes unbalanced, which causes some kind of slowness in most operations execution. the question that posse itself can we solve this balancing problem?

To answer this, we took two hypotheses into consideration, either making a balanced indexation level or replace it with another data structure.

for many reason we took the second hypotheses becouse We believe that it is the best choice.

In order to improve the quality of services provided in databases or any system components that require speed and accuracy, the Synchronous AVL Tree Data Structure is the most concurrent algorithm needed to build an integrated system based on accuracy and speed.

This project manages both data structure and concurrency techniques in order to make efficient and hight performance tasks.

In contrast to other concurrency algorithms, this algorithm offers high techniques. These include: one thread responsible for rebalancing Issues, our solution is:

- A Concurrent AVL tree which deal with integer values.
- A periodic thread responsible for rebalancing start after 500 operation.
- A re-entrant lock for avoiding deadlock did b the thread itself.
- A thread safe concept Which was submitted by brian Goetz in concurrency in practice book.
- An optimistic lock as important technique to handle various synchronizations.

Keyword: AVL Tree, BST, SKIPLIST, Concurrency, Balanced Tree, Algorithm, Optimistic concurrency.

Abstract

La "skip list" synchronisée Caractérisée par l'indexation ou le niveau, contribue à rechercher plus rapidement dans différentes opérations simultanément, mais lorsque cette skip list simultanée gère beaucoup de données, l'indexation qui lui y est associée devient déséquilibrée, ce qui entraîne une certaine lenteur dans l'exécution de la plupart des opérations. La question qui se pose peut-on résoudre ce problème d’équilibrage ?

Pour y répondre, nous avons pris en considération deux hypothèses, soit en établissant un niveau d'indexation équilibré, soit en le remplaçant par une autre structure de données.

Nous avons retenu la seconde hypothèse car nous pensons que c'est le meilleur choix.

Afin d'améliorer la qualité des services fournis dans les bases de données ou dans tout composant système nécessitant vitesse et précision, la structure de données de l'arbre de recherche AVL synchrone est l'algorithme le plus simultané nécessaire pour construire un système intégré basé sur la précision et la vitesse.

Ce projet gère à la fois la structure des données et les techniques de concurrence afin de réaliser des tâches performantes et efficaces.

Contrairement à d'autres algorithmes de concurrence, cet algorithme offre des techniques élevées.

Ceux-ci incluent : un thread responsable du rééquilibrage de l'arbre, notre solution est :

• Un arbre AVL simultané qui traite des valeurs entières.

• Un thread périodique chargé du rééquilibrage démarre après 500 opérations.

• Un verrou (lock) rentrant pour éviter le blocage fait par le thread lui-même.

• Un concept thread-safe qui a été soumis par Brian Goetz en concurrency in practice livre.

• Un verrou optimiste comme technique importante pour gérer diverses synchronisations

List of Abbreviations

BST	**Binary Search Tree**
AVL	**Adelson-Velsky and Landis**
BF	**Balance Factor**
RMW	**Read-Modify-Write**
JVM	**Java Virtuel Machine**
JMM	**Java Memory Model**
API	**application programming interface**
CPU	**central processing unit**
IDE	**integrated development environment**
POM	**Project Object Model**
XML	**Extensible Markup Language**
JMH	**Java Microbenchmark Harness**
CAS	**Compare And Swap**

General Introduction

In our present time, technology has become an integrated part of human life, as it has facilitated the movement, transfer and processing of data, so that any man can receive calls or talk via phone, make a direct broadcast, see the stock market exchange, do bank transactions or buy via the Internet.

This would never happen through a lot of research conducted on small components that control large systems, represented by data structures consists in organizing, and storing format that enables efficient access and modification on data.

However, the data structures alone are not enough because the world's requirements, need speedy processing Also, in storing, extracting and processing data, here we define the term concurrent data structures, which store and organize the data for accessing by multiple threads.

There are a lot of data structures that have been programmed and many works have been presented about them, including LinkedList, Stack, Queue, Tree.

The data structures aforementioned created some problems, including the slow reading and writing when a huge amount of data was entered into so that the degree of its complexity reached O(n).

This problem has been solved by a dynamic algorithm skip list written by Professor William Pugh in 1989, its degree of complexity reached O(Log N) in almost all operations, and so far some programming languages and databases still use this algorithm as:

Apache Portable Runtime, MemSQL DB, Cyrus IMAP server, nessDB and ConcurrentSkipListSet.

The skip list Characterized by what is called indexation or level, which Contributes to search faster in different operation, but when entering large amounts of data, the indexation related to it becomes unbalanced, which causes some kind of slowness in most operations execution. So, we are wondering if this problem can be solved and how can we do it concurrently?

Data structure

Introduction

A data structure is a special way of organizing and storing data in a computer so that it can be used efficiently. Array, LinkedList, Stack, Queue, Tree, Graph etc are all data structures that stores the data in a special way so that we can access and use the data efficiently.

We used to deal with data structure in proportion to the problem we are facing, in this chapter we tackle the definition of the data structure and see the two types of data structure that we are going to build our work on.

1.Overview on data structure

A.Definition of data structure

The term data structure is used to denote a particular way of organizing data for particular types of operation. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data usage of data structures

There are different types of data structures suited to different kind of applications, some of them are highly connected to specific tasks for example relation databases commonly use balancing self-tree as B-tree or red-black-tree indexes for data retrieval. Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services.

2.Skip list data structure

A.Skip list

Skip list is a relatively young data structure invented by William Pugh, it's a probabilistic data structure maintains a dynamic set of N element in O(log n) time per operation in expectation and with height probability result.

The element in skip list is kept sorted by key. It allows a quick search, insertions and deletion of elements with simple algorithms.

Basically, skip list is a linked list with additional pointers such that intermediate node can be skipped, it uses a random number generator to make some decisions.

a. Reasons for Probabilistic

The level of every new node to be inserted is chosen randomly with a probability distribution that keeps approximately the proportions of nodes of level. The complexity of operation (search, insert and delate) takes a O(log n) of time. But in the worst-case skip list operation take O(n) of time complexity when all or almost all the nodes are given level 1 at insertion in that case skip list becomes an ordinary linked list.

The skip list may be viewed as tree where the head is the root, the nodes levels correspond to levels in tree, level 1 node are the leaves.

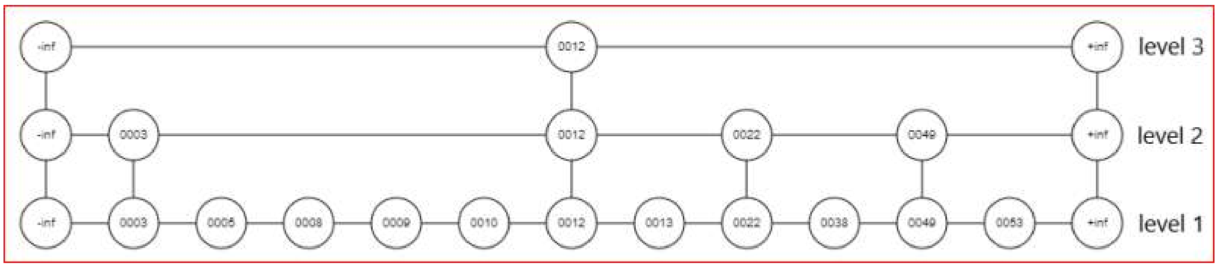


Figure 1 skip list data structure

b.Searching

Searching for an element x is to travers forward pointers that do not overshoot the node containing the element being searched for (figure), When no more progress can be made at the current level of forward pointers, the search moves down to the next level.

When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

Figure 2 Searching algorithm in skip list

Example

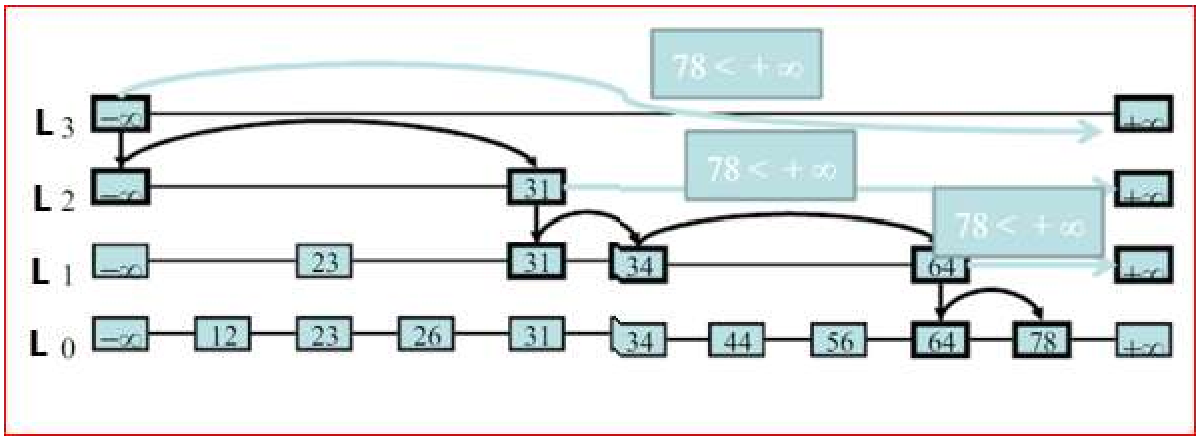


Figure 3 example of searching in skip list

3.Introduction to Binary tree data structure

A.Definition of Binary Tree

a.Binary tree

Binary Tree is a hierarchical data structure in which each node has zero, one, or at the most, two children. Each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer represents the topmost node in the tree. Each node in the data structure is directly connected to arbitrary number of nodes on either side, referred to as children. A null pointer represents the binary tree. There is no particular order to how the nodes are to be organized in the binary tree. Nodes with no children nodes are called leaf nodes, or external nodes.

b.Binary search tree

A binary search tree (BST) is a binary tree with special property which is given as:

Key which is a data stored at the node.

Left child subtree and right child subtree are pointers to the parent node.

∀ A, B node, if B ∊ left subtree of A then key of B ≤ the key of A and if B ∊ right subtree of A the key of B ≥key of A.

c.difference between binary tree and binary search tree

1. Definition of Binary Tree and Binary Search Tree

Binary Tree is a hierarchical data structure in which a child can have maximum two child nodes; each node contains a left pointer, a right pointer and a data element. There's no particular order to how the nodes should be organized in the tree. Binary Search Tree, on the other hand, is an ordered binary tree in which there is a relative order to how the nodes should be organized.

2. *Structure of Binary Tree and Binary Search Tree*

The top node in the tree represents the root pointer in a binary tree, and the left and the right pointers represent the smaller trees on either side. It’s a specialized form of tree which represents data in a tree structure. on the other hand, The Binary search tree, is a type of binary tree in which all the nodes in the left subtree are less than or equal to the value of the root node and that of the right subtree are equal or greater than to the value of the root node.

3. *Operation of Binary Tree and Binary Search Tree*

Binary tree can be treated as node that has two children and one parent. Common operations that can be performed on a binary tree are insertion, deletion ets. Binary search trees are more of sorted binary trees that allows for fast and efficient lookup, insertion, and deletion of items. Unlike binary trees, binary search trees keep their keys sorted, so lookup usually implements binary search for operations.

4. *Types of Binary Tree and Binary Search Tree*

There are different types of binary trees, the common being the Complete Binary Tree, and Extended Binary Tree. Some common types of binary search trees include AVL trees, Red-Black trees etc.

B.Type of Binary search tree

As we know that BST has a complexity of O (log n) in searching but unfortunately, this is not entirely guaranteed for example:

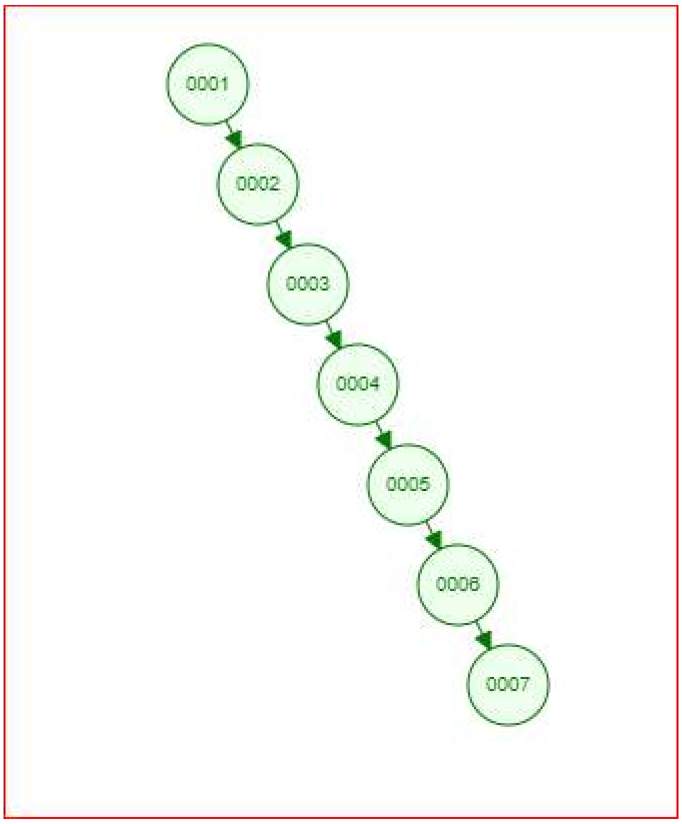


Figure 4 A BST tree(Special case)

This special case is a binary search tree to in which search proceeds the same as in a linked list, so we obliged to consider the balance of a binary search tree which has subtree that are almost equal in size and depth.

From this point we can two classification for BST:

a.Imbalanced tree
Which is a BST which in the worst case will take O(n) complexity in searching.

b.Self-balancing binary search tree
defined as a BST in which the height of right and subtree of node differ by not more than 1.

There is a different implementation for balanced tree as AVL tree, red-black-tree…etc.

4.Self-balancing AVL BST

A.AVL tree

The AVL tree (**A**delson-**V**elsky and **L**andis) is a self-balancing binary search tree for every internal node *v* of *T*, the heights of the children of *v* can differ by at most 1 example:

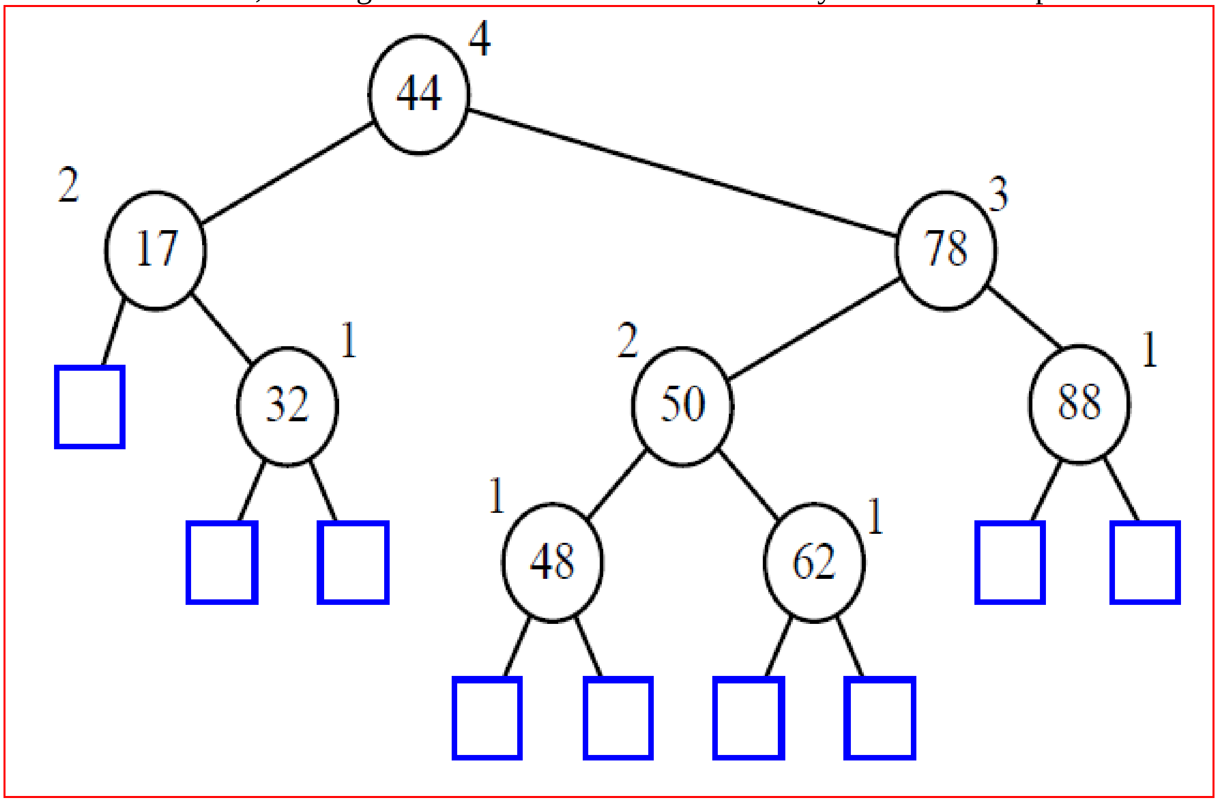
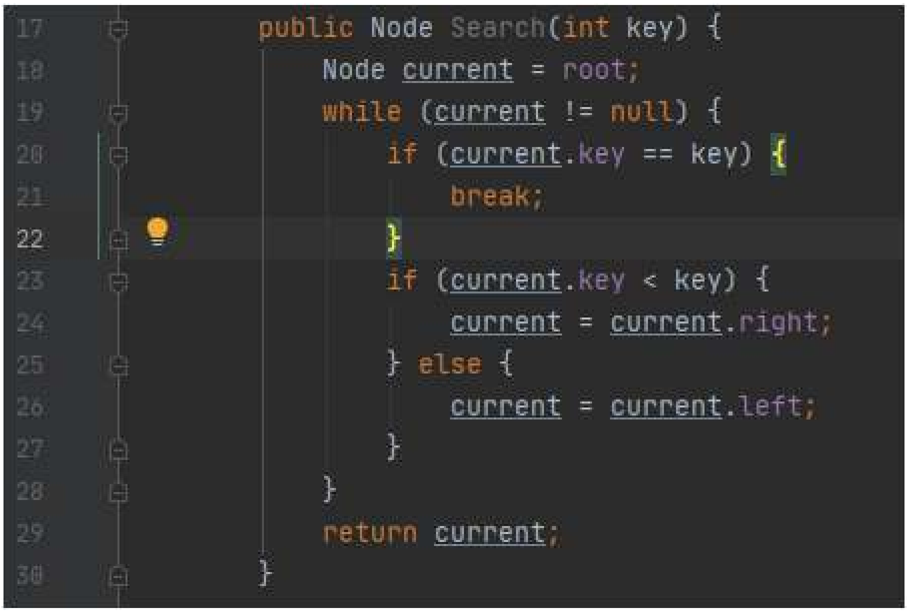


Figure 5 balanced AVL tree

B.Operations

a.Search
Searching operation in AVL tree is the same in any binary search tree, start from the root and compare the key with value of current node return the node if equal or go to the next right node if the key is greater otherwise continue the search from the left child.

Time complexity of search operation take O(log n) in the worst case



```java
public Node Search(int key) {
    Node current = root;
    while (current != null) {
        if (current.key == key) {
            break;
        }
        if (current.key < key) {
            current = current.right;
        } else {
            current = current.left;
        }
    }
    return current;
}
```

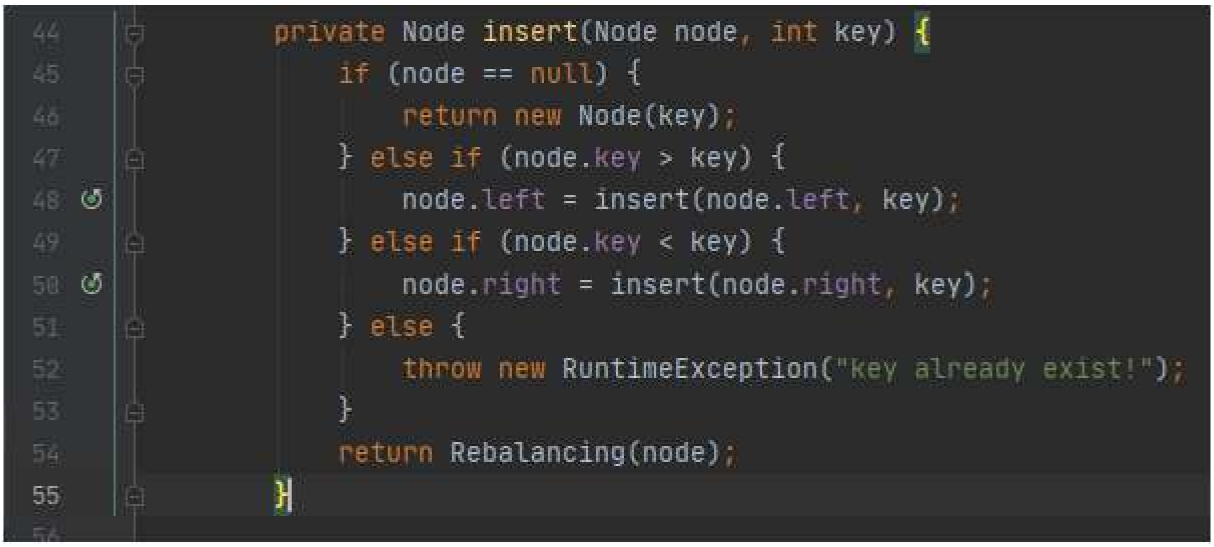
Figure 6 search method in sequential

b.Insert

We check If the tree is empty, then the node is inserted as the root of the tree. In case the tree has not been empty then recursively we go down the root searching for a sub-tree mark on a null because the new inserted node always replaces a NULL reference (left or right).

After this insertion, the tree may become unbalanced, only ancestors of the newly inserted node are unbalanced. So, it is necessary to check and update the balance factor for each node

If the balance factor after insertion is bigger than 1 or smaller than -1 here, we do some rotation to balance the tree, and if not, we keep it as it is.



```java
private Node insert(Node node, int key) {
    if (node == null) {
        return new Node(key);
    } else if (node.key > key) {
        node.left = insert(node.left, key);
    } else if (node.key < key) {
        node.right = insert(node.right, key);
    } else {
        throw new RuntimeException("key already exist!");
    }
    return Rebalancing(node);
}
```

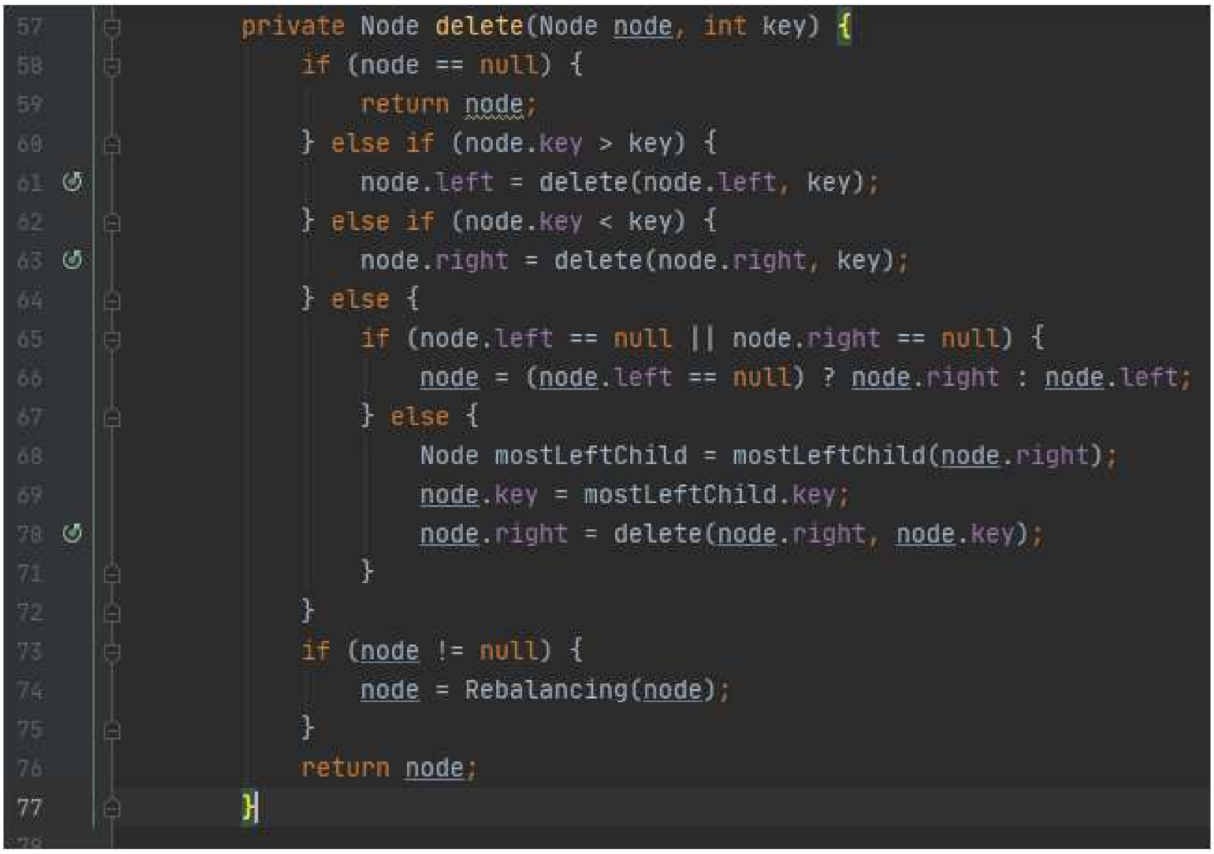
Figure 7 insert method in sequential

c.Delete

Deleting operation in avl-tree starts with finding the node (n) should be deleted then we have to introduce the new candidate to be its replacement in the tree. If n is a leaf, the candidate is empty we just replace it with a null. If n has only one child, this child is the candidate, but if Z has two children, the process is a bit more complicated.

Assume the right child of n called A. First, we find the leftmost node of the A and call it B. Then, we set the new value of Z equal to X ‘s value and continue to delete B from A.

Finally, we call the rebalance method at the end to keep the BST an AVL Tree.



```
    private Node delete(Node node, int key) {
        if (node == null) {
            return node;
        } else if (node.key > key) {
            node.left = delete(node.left, key);
        } else if (node.key < key) {
            node.right = delete(node.right, key);
        } else {
            if (node.left == null || node.right == null) {
                node = (node.left == null) ? node.right : node.left;
            } else {
                Node mostLeftChild = mostLeftChild(node.right);
                node.key = mostLeftChild.key;
                node.right = delete(node.right, node.key);
            }
        }
        if (node != null) {
            node = Rebalancing(node);
        }
        return node;
    }
```

Figure 8 deletion method in sequential

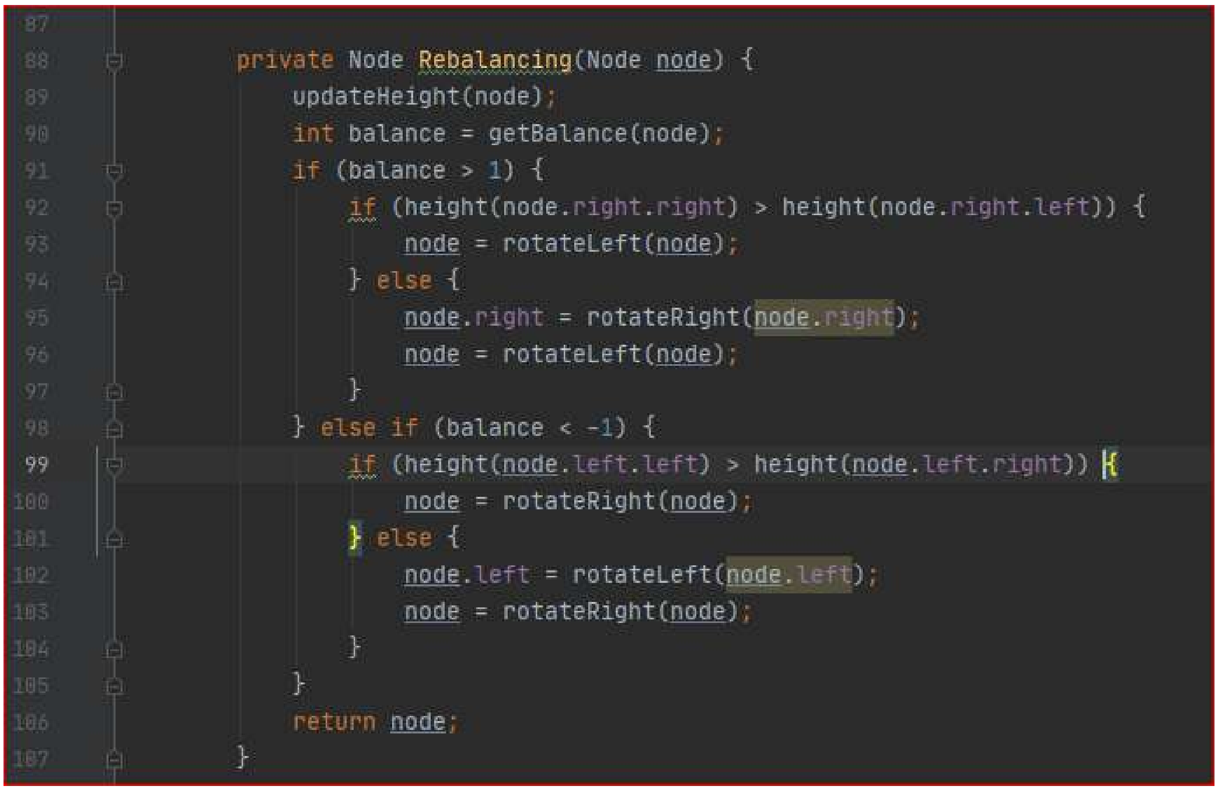
C.Rotation

After performing every operation like deletion or insertion, we need to check if the tree is still balanced here, we recalculate the balance factor of every node in the tree. If every node satisfies the balance factor condition which is less than 2 then we conclude the operation. Otherwise, we must make it balanced.

We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

There are four rotations which are classified into two types:

Simple rotation and double rotation, this implementation in figure shows how to do rebalancing of each operation.



```
        private Node Rebalancing(Node node) {
            updateHeight(node);
            int balance = getBalance(node);
            if (balance > 1) {
                if (height(node.right.right) > height(node.right.left)) {
                    node = rotateLeft(node);
                } else {
                    node.right = rotateRight(node.right);
                    node = rotateLeft(node);
                }
            } else if (balance < -1) {
                if (height(node.left.left) > height(node.left.right)) {
                    node = rotateRight(node);
                } else {
                    node.left = rotateLeft(node.left);
                    node = rotateRight(node);
                }
            }
            return node;
        }
```

Figure 9 rebalancing method in sequential

a.Balance factor

In order to maintain the balance of the tree during the writing operations, we attach to each node a balance factor, which is a variable used to check whether any node in the AVL have violated balanced property of tree or not. If any node in AVL Tree has violated the AVL-ness property then restore its property by performing a set of manipulations on the tree. So variable represents the difference between the height of the right sub-tree and the left sub-tree, and each node has a balance factor (BF) between 2<BF<-2.

Balance factor = height (Right sub-tree) - height (Left sub-tree)

b.Simple rotation

The tree starts its operation of balancing itself at least When the balance factor equal or biggest than 2, A simple rotation

1. Left rotation

If a tree becomes unbalanced like in figure 10, when a node is inserted into the right subtree of the right subtree, we perform a single left rotation which the node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of **B**.

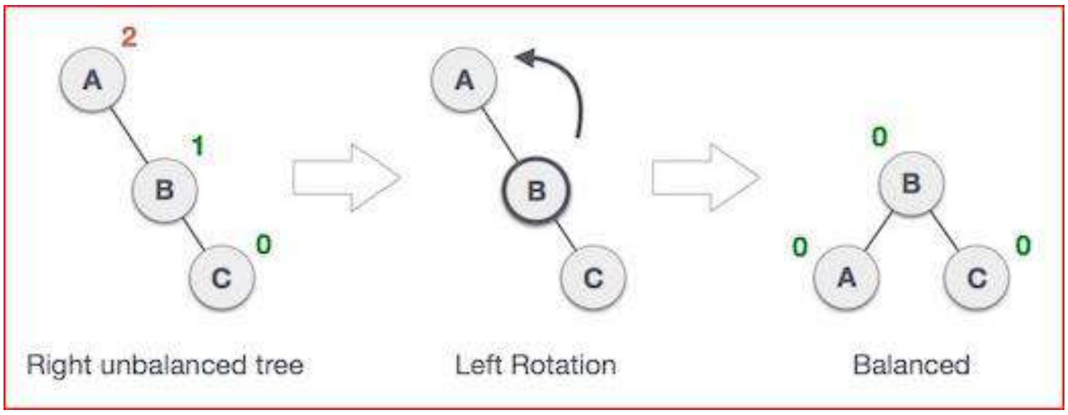


Figure 10 left rotation

2. Right rotation

In figure 11 a right rotation is a mirror of the left rotation operation described above. if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation in which the node **C** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the right rotation by making **C** the right-subtree of **B** as in figure.

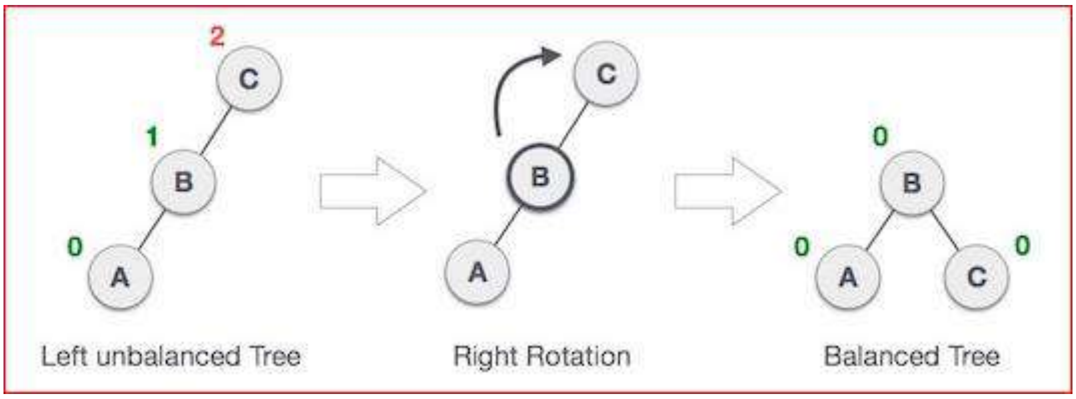


Figure 11 right rotation

c.double rotation

When we have the left-right and right-left cases, a single rotation is not sufficient to rebalance the tree. In this case, we need to perform a double rotation, which consists of two single rotations. A left followed by right rotation in left right case or a right rotation followed by left rotation in right left case.

1. left right rotation

A left-right rotation (figure 12) case happens when a node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B** here we get the left rotation case. We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree as figure shown.

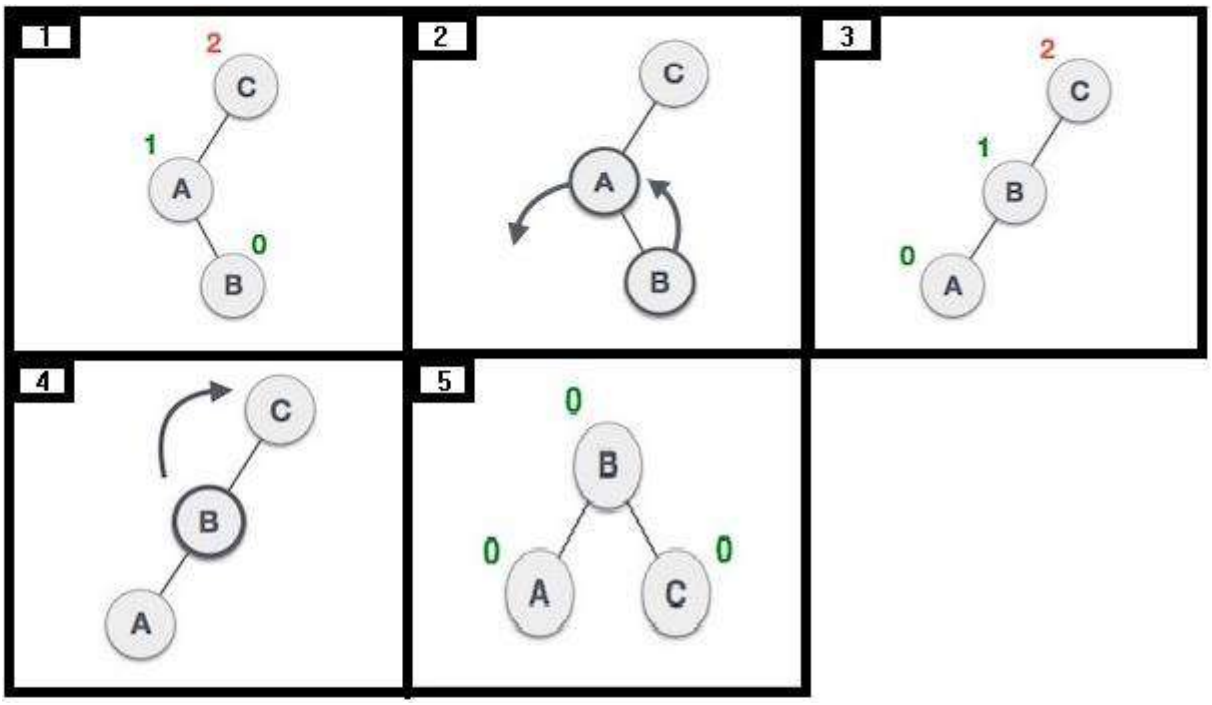


Figure 12 left right rotation

2. right left rotation

In this case, we have a combination of right rotation followed by left rotation.

As we see in figure 13, a node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A** then we get the right rotation because of the left-subtree of the left-subtree. Now we shall do a right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

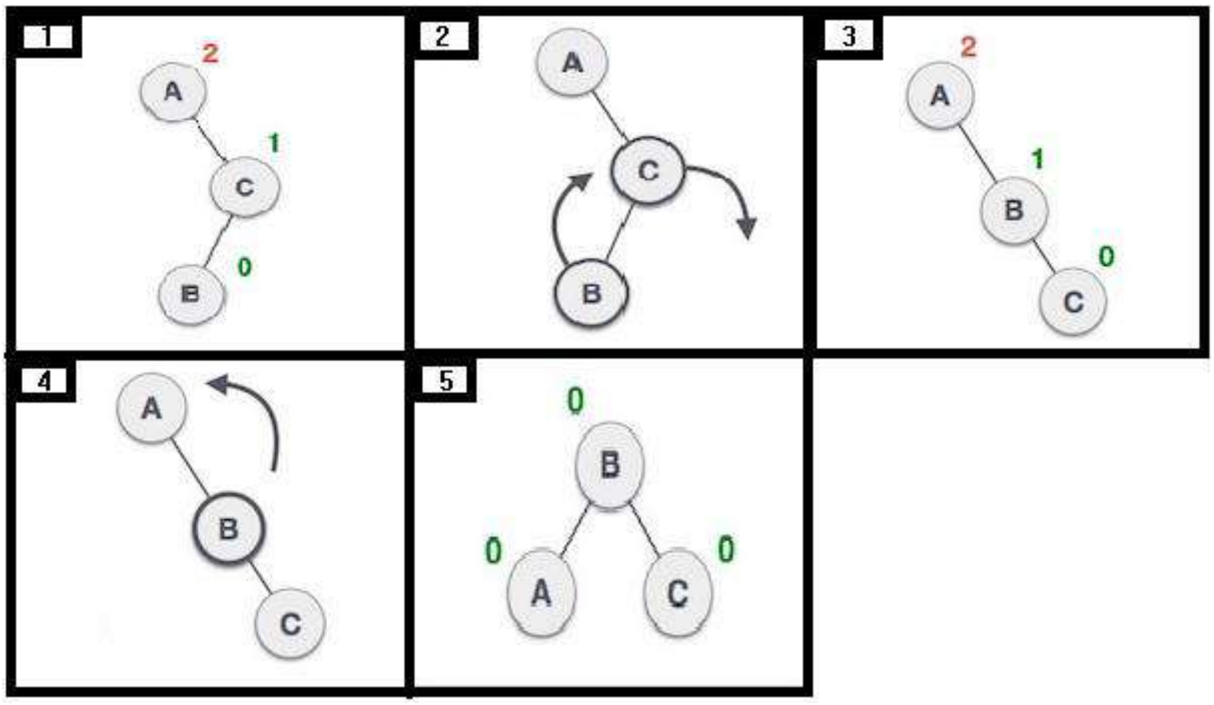


Figure 13 right left rotation

D.Runtime complexity

In the worst case, most of operations equal to height of avl tree and height always take O(log(n)) time where n is the number of nodes.

The rotation operations take O(c) where c is a constant as only few pointers are being changed there.

E.Conclusion:

After understanding these data structure and in order to complete shedding light on the basic note and approach to understand and built our work, we must investigate some concepts of concurrency approach in the next chapter.

Concurrency approach

Introduction

Concurrency is a broad approach so that we can explain it in all the details but we can say that concurrency is all about structuring and designing algorithm. It is widely used in computer science today to solve problems efficiently. One such example is that of bank operations in which someone deposits and withdraws cash from their bank account at the same time.

In this chapter, we will present the important points and details that must be précised when dealing with concurrency.

2.Overview on concurrency approach

F.What is concurrency

The concurrency is coordination and management of independent lines of execution. These executions can be truly parallel or simply be managed by interleaving. They can communicate via shared memory or message passing.

Rob Pike's definition: "the composition of independently executing computations"

Examples of Concurrency

Real world contains actors that execute independently of, but communicate with, each other. In modelling the world, many parallel executions have to be composed and coordinated, and that's where the study of concurrency comes in. Consider:

- Banking systems.
- Railway Networks (SCADA systems).
- Multiplayer games.
- Well-coordinated teams or societies.

a.Hardware examples:

- A processor can have multiple cores (multicore).
- A computer can have multiple processors.
- A network can have multiple computers.

b.Software examples:

multiprogramming, or running a lot of programs concurrently (the O.S. has to multiplex their execution on available processors). For example:

- listening to streaming audio.
- having a clock running.
- chatting.
- broadcast live.

- Simulation programs.
- High volume servers (multiple clients are serviced at the same time (possibly in parallel, possible with interleaved execution) to give the appearance of responsiveness).

3.The difference between concurrency and parallel programming

(intended to be used in the computer science world is a way to build things it's a composition of independently executing things (as)typically function and we usually express those as the interacting processes like thread or co-routines) or we can say it's a composition of independently executing processes parallelism is simultaneous execution of multiple things

Concurrency	**parallel**

It is about dealing many processes at once.	It is about doing a lot of processes at once.

Concurrency is a way to stricture thing so that u can may be use parallelism to do better job but it doesn’t need to be.

In addition to that in concurrency there are three concepts as a fundamental to a correct concurrent programming. And when a concurrent program is not written correctly the errors fall into one of these categories:

c. Atomicity

The atomicity deals with which action have indivisible effects, it is usually thought of in term of mutual exclusion.it is a property of grouping multiple operations into an all or nothing unit of work, which can succeed only if all individual operations succeed.
In generally we use atomicity used with RMW (read from memory, modify, write it back to memory) actions.

d.Ordering

Ordering constraints describe what order things are seen to occur. You only get intuitive ordering constraints by synchronizing correctly.

Generally, it is happening when two thread gets invoked on the same object in another. The Thread one sow a part updating by thread two for example thread two updating two variable a and b but thread one sow the change on variable b before the change to a.

e.Visibility

Refers to shared action between threads, if an action in one thread is visible to another thread then the result of that action can be observed by the second thread.in order to guarantee that you have to use some forms of synchronization given by the virtual machine like to make sure that the second thread sees the action did by the first thread.

4.The types of concurrency technique

G.Synchronisation

Synchronization is the capability to control the access of multiple threads to any shared resource and Grant better option where we want to allow only one thread to access the shared resource, in which java implement it with a concept called monitors which allow only one thread to own a monitor at a given time. there are many types to handle thread synchronization as using:

1. *Mutex:*

The mutex (In fact, the term mutex is short for mutual exclusion) also known as spinlock is the simplest synchronization tool that is used to protect critical regions and thus prevent race conditions. That is a thread must acquire a lock before entering into a critical section (In critical section multi threads share a common variable, updating a table, writing a file and so on), it releases the lock when it leaves critical section.

2. *Read Write Lock:*

Read-write-lock allow one writer OR multiple parallel readers. If a reader is reading then a writer request is delayed until the readers complete. If a writer is changing data, all new reads are blocked. All readers will always be reading the same data. If you have a writer changing data a lot, this causes your readers to be continually blocking. The delay on the writer is also high due to a potentially high amount of parallel readers that need to exit.

3. *Lock Free:*

called non-blocking These are structures that don't use a mutex or any locks at all, and can have multiple readers and multiple writers at the same time and thread cannot cause failure or suspension or any interruption of any other thread.

An interesting part of lock free is that all threads are working on the same set - if thread 1 reads a value, then thread 2 writes the same value, the next read from thread 1 will show the new value. There are some times where this is really useful as a property when you need a single view of the world between all threads, and your program can tolerate data changing between reads.

4. *Wait Free:*

Wait free is a specialisation of lock free, where the writer/ reader has guaranteed characteristics about the time they will wait to read or write data. This is very detailed and subtle, only affecting real time systems that have strict deadline and performance requirements.

5. *concurrently readable:*

A concurrently readable structure allows one writer and multiple parallel readers. An interesting property is that when the reader “begins” to read, the view for that reader is guaranteed not to change until the reader completes. This means that the structure is transactional.

5.Overview on java concurrency

When the Java language was programmed, Its developers maker assured of making compatible with concurrency approach for that they program many concept for that as multi-threading concept, here we will show some features and guarantees provided by the Java language

A.Thread in java (Multi-threading)

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler.it can be used to perform complicated tasks in the background without interrupting the main program. In java language threads are an inescapable feature, they can simplify the development of complex systems by turning complicated asynchronous code into straight-line code.

a.Thread safety

a class is thread-safe when it continues to behave correctly when accessed from multiple threads.

A mechanism applicable to multi-threading used to manage access to shared state and protect shared data from uncontrolled concurrent access. There are various implementation strategies to improve thread safety which is guaranteed to be free of race-conditions when many threads have a simultaneously execution in here, we have to define race-conditions and how to avoiding it.

Race-conditions

a thread writes a variable that might next be read by another thread or reads a variable that might have last been written by another thread if both threads do not use synchronization; code with data races has no useful defined semantics under the Java Memory Model. Not all race conditions are data *races, and not all data races are race conditions, but they both can cause concurrent programs to fail in unpredictable ways*

A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing. The most common type of race condition is check-then-act, where a potentially stale observation is used to make a decision on what to do next. and the type condition is RMW, as like incrementing a value to avoid race-condition we have to use some technique as:

1. *Re-entrancy*

Re-entrancy means that locks are acquired on a per-thread rather than per-invocation basis Re-entrancy is implemented by associating with each lock an acquisition count and an owning thread. When the count is zero, the lock is considered unheld. When a thread acquires a previously unhold lock, the JVM records the owner and sets the acquisition count to one. If that same thread acquires the lock again, the count is incremented, and when the owning thread exits the block, the synchronized count is decremented. When the count reaches zero, the lock is released.

2. *Thread-local storage*

The meaning of thread local storage is that each thread has its own private copy of variables.

3. *Immutable objects*

Immutable object cannot be changed after creating they only allow to read it. Mutable operations can then be implemented in such a way that they create new objects instead of modifying existing ones.

4. *Serializable object*

It is a mechanism of converting the state of an object into a byte stream then store it in a file or memory, It used for detecting changes in time-varying data.

5. *Mutual execution*

It is a concept used with a critical section, a piece of code in which processes or threads access a shared resource. Only one thread owns the mutex at a time, when a thread tries to acquire a mutex, it gains the mutex if it is available, otherwise the thread is set to sleep condition. otherwise When it holds a resource, it has to lock the mutex from other threads to prevent concurrent access of the resource. the mutex can be enforced at both the hardware and software levels.

6. Atomic operations

Atomic operations mean that thread access a shared data which cannot be interrupted by other threads, when the operation is atomic, you have a guarantee that the shared data is always kept in a valid state, no matter how other thread access it.

b.Java memory model (JMM)

It’s very important to understand the **JMM** to design a concurrent program with high performance because the java memory model define the guaranties that all of the java compiler, virtual machine, OS and hardware are obliged to be compliant with, which make it easier for developers to design/implement concurrent programs that are reliable and correspondent designing specification, in other side the **JMM** defines a partial ordering on program actions or event (lock/unlock, read/write) called happens-before. If event X happens-before Y, it means X’s results are visible to Y. But between thread there are no visibility guarantees if you don't use SYNCHERONIZED.

B.Synchronization and volatile in java

a.synchronization

Synchronization in java is the capability to control the access of multiple threads to any shared resource. Java provide a great implementation for concurrency, this implemented using a concept called monitors also known as monitor lock or intrinsic lock in order to manage synchronization, each object in Java is associated with a monitor, which a thread can lock/unlock. this concurrency **API** have different synchronization mechanisms that allow to define a critical section to access shared resources, the most important synchronization mechanisms is:

Synchronization keyword: who allows you to define critical section in a method or a block of code.

1. Lock interfaces*:*

It works like synchronized blocks except locks can be more sophisticated than Java’s synchronized blocks. Locks allow more flexible structuring of synchronized code. There is different type of lock as:

Reentrantlock:

Which means that locks are bound to the current thread. A thread can safely acquire the same lock multiple times without running into deadlocks.

Re-entrant-read-write-lock:

Which separate write operations from read operations. The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable. the read-lock can allow threads to read simultaneously as long as no threads hold the write-lock.

2. Semaphore*:*

Java provide semaphore to control the number of concurrent threads accessing to a specific resource

b.Volatile

Volatile keyword is a controller memory order provide by the **JVM**, it used to mark a java variable as being stored in main memory, and When a field or variable declared volatile the compiler is put a notice that this variable is shared and that operations on it should not be reordered with other memory operations. that means every read/write of a volatile variable will be read/write from the computer s main memory not the **CPU** cache and that provide memory visibility.

c.Deadlock

These are some of the guidelines using which we can avoid most of the deadlock situations.

1. *Avoid deadlock by breaking circular wait condition:*

In order to do that, you can make arrangement in the code to impose the ordering on acquisition and release of locks. If lock will be acquired in a consistent order and released in just opposite order, there would not be a situation where one thread is holding a lock which is acquired by other and vice-versa.

2. *Avoid Nested Locks:*

This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It’s almost impossible to get a deadlock situation if you are working with only one object lock.

3. *Lock Only What is Required:*

You should acquire lock only on the resources you have to work on, if we are only interested in one of its fields, then we should lock only that *specific field not complete object*.

4. *Avoid waiting indefinitely:*

You can get a deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it’s always *best to use join with maximum time you want to wait for the thread to finish.*

6.Conclusion

In this chapter, we learned about some important points in synchronization, And we knew that there is no doubt that the Java language has provided us with many advantages that support concurrency programming and in the previous chapter we mentioned about what we want to know about data structure, in the next chapter.

Depending on these two chapters, we can start to explain and find out how the implementation in the next chapter is to be build.

.

Implementation

1.Introduction

In this chapter we discuss solution about the problem of concurrent skip list Which can be summarized by unbalancing level that made kind of a weakness Exploration by taking a time to access to the data due to level indexes who are not balanced Especially when we deal with concurrent data structure because if the data structure work with locks it may Disrupts(interrupt) other operation during the exploration (in pessimistic lock case).

2.Solution

a.idea

By investigating and analyzing the topic, we found that this problem solved in two ways:

1. making a balanced indexation level

making the indexation level balanced is a good idea ,but we must put into consideration that this implementation takes a long time so it can be achieved because we have to deal with Nodes of a LinkedList or tree that can be rotated concurrently (change its references link in the LinkedList case) until reaching the node we want in the last level.

2. replace it with another data structure

Replacing it with another concurrent data structure as concurrent balanced **AVL** tree that can do some rotation concurrently when it needs to and each node contain a data.

b.Starting the solution

We have taken the second as a solution to this problem so we proposed a concurrent algorithm based on AVL tree, rather than the more popular red-black tree, because balance AVL trees are less complex than balance red-black trees. The AVL balance condition is stricter.
This algorithm implemented by optimistic locking (non-blocking algorithm) and a periodic rebalancing thread which rebalances when an update threshold is reached.

c.tools

This part specified to offering tools and libraries that we used to build the algorithm.

1. *IntelliJ IDE*

is one of the most powerful and popular Integrated Development Environments (IDE) for Java. It was developed and is maintained by JetBrains, and is available in community and ultimate edition. This feature-rich IDE enables rapid development and helps in improving code quality.

2. *maven repository*

is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

Maven's streamlined, XML-based configuration model enables developers to rapidly describe or grasp the outlines of any Java-based project, which makes starting and sharing new projects a snap. Maven also supports test-driven development, long-term project maintenance, and its declarative configuration and wide range of plugins make it a popular for developers.

3. *JMH benchmark*

(the Java Microbenchmark Harness) is a toolkit that helps you implement Java microbenchmarks correctly. JMH is developed by the same people who implement the Java virtual machine it takes care of the things like JVM warm-up and code-optimization paths, making benchmarking as simple as possible, harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages.

3.Our algorithm

A.introducing the algorithm

We present our concurrent tree algorithm through which we intend to compete the concurrent skip list set made by java as an integer object that supports four methods:

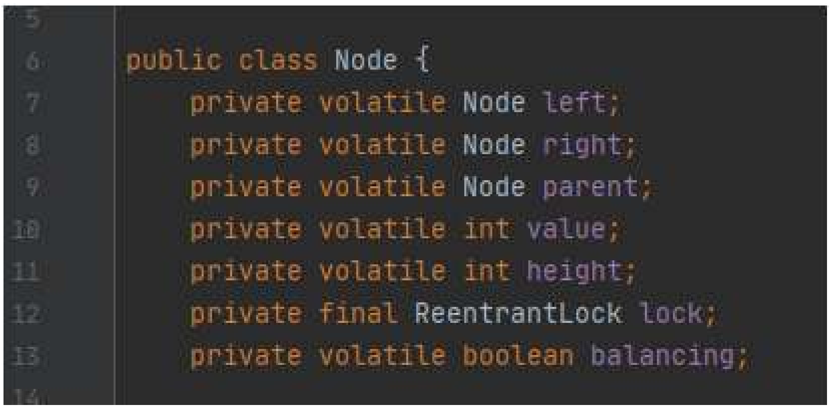
A. Insert
B. Search
C. Remove
D. StartPeriodicRebalancingThread
E. PeriodicRebalancing.

The insert (int)operation is a Boolean method takes an integer value as a parameter and returning true when inserting this integer value, the search method takes an integer value as parameter and returning true if the current value is in the tree false if not, remove method take integer value as parameter and return true if the integer value has been delated successfully false if not exist in the tree, the size method returning int value as the size of the tree, the minNode method take a three nodes as one object which is grandfather node and the parent node and the child node and returning a new object representing the most three minimum nodes, the StartPeriodicRebalancingThread method doesn't return anything it takes an object representing the concurrent AVL tree class and start a new runnable thread to call the periodic Rebalancing method to start rebalancing the tree, peroidicRebalancing method take an object representing the concurrent AVL tree and start rebalancing the tree.

a.The data structure: Node

The nodes that compose our tree store their own height and value, each node have an object representing its own lock and a Boolean balancing set it as false and using it when the rebalancing thread start rebalancing the nodes.

Figure 14 shows the fields of a node. All fields are mutable. height, parent, left, right, value, lock, and balancing may only be changed while their enclosing node is locked. parent may only be changed while the parent node is locked (both the old and the new parent must be locked).



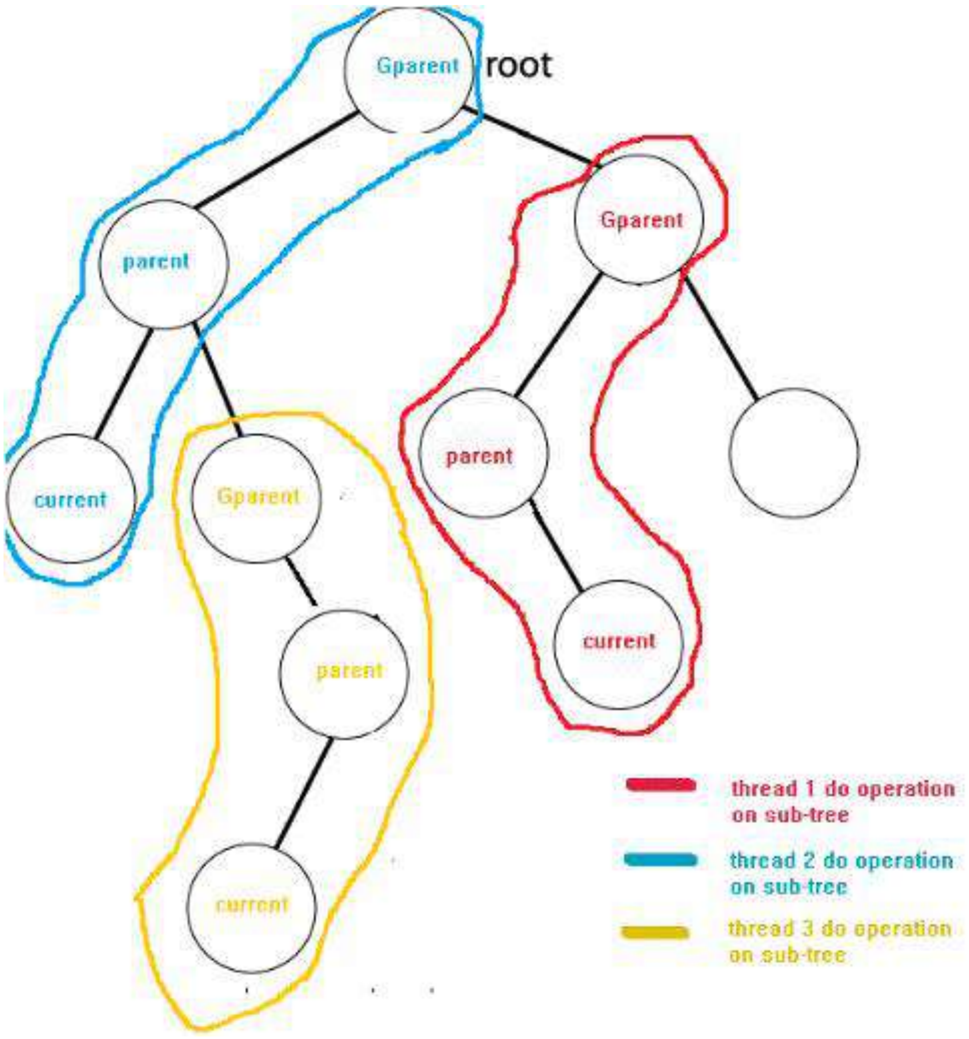
```
public class Node {
    private volatile Node left;
    private volatile Node right;
    private volatile Node parent;
    private volatile int value;
    private volatile int height;
    private final ReentrantLock lock;
    private volatile boolean balancing;
```

Figure 14 node structure

b.subtree nodes:

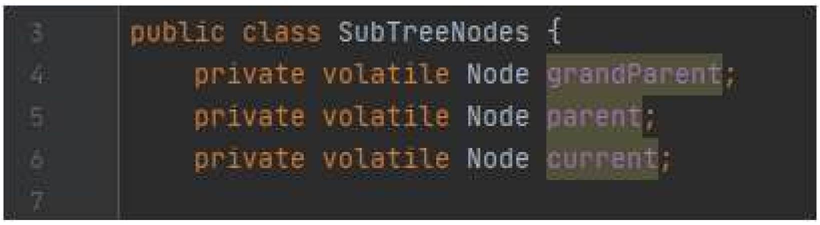
the sub tree node in figure 15 takes three nodes as a grandparent node, parent node, child node as showing in figure 16, the tree node is mutable object and that give as a guaranty that provide memory visibility when change happening by threads.

This technique used to help threads to handling the sub-trees in order to using them as groups in most of operations.



Gparent
root
parent
current
Gparent
parent
current
Gparent
parent
current
thread 1 do operation on sub-tree
thread 2 do operation on sub-tree
thread 3 do operation on sub-tree

Figure 15 threads dealing with group of sub tree structure



```java
public class SubTreeNodes {
    private volatile Node grandParent;
    private volatile Node parent;
    private volatile Node current;
```

Figure 16 group of sub tree declaration

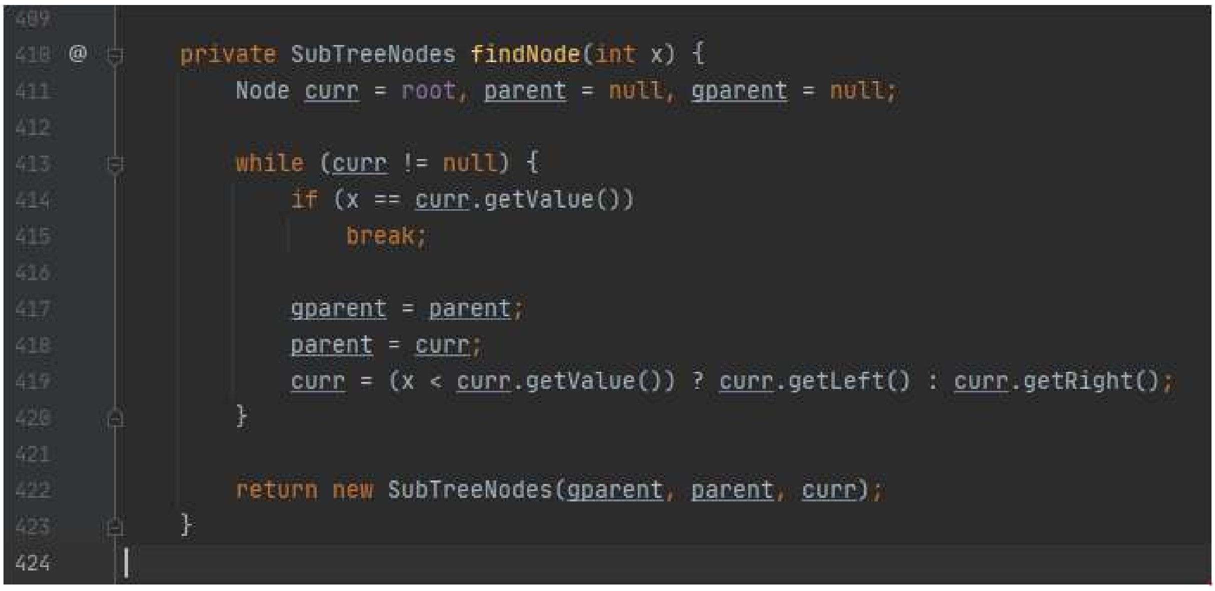
1. *lock handled by threads*

The technique of lock that we used in our implementation distinguished from synchronized method that we win the time that takes by the thread who access these methods and get the lock from three node at time, that allows other thread to do its operation in other of nodes.

c.findNode:

the findNode method showed in figure 18,designed to take an integer value, then start with the root node in the tree as a start point then test while the current value is not null get its value and test it if the integer value that was given are equal the current value break the operation, else assigned the grandparent node is the parent node and the parent node assigned to the current node then test again if the value node are less than the current node, we take the left node as a new current node, else we take the right node as the figure 19 showing.

Most of the operations are reading operations so we decide to lock only the writing and let the reading without any kind of lock that enables us to profit more performance and by using a volatile field in value we guarantee that the current value is the last version of it.



```java
    private SubTreeNodes findNode(int x) {
        Node curr = root, parent = null, gparent = null;

        while (curr != null) {
            if (x == curr.getValue())
                break;

            gparent = parent;
            parent = curr;
            curr = (x < curr.getValue()) ? curr.getLeft() : curr.getRight();
        }

        return new SubTreeNodes(gparent, parent, curr);
    }
```

Figure 17 finding method

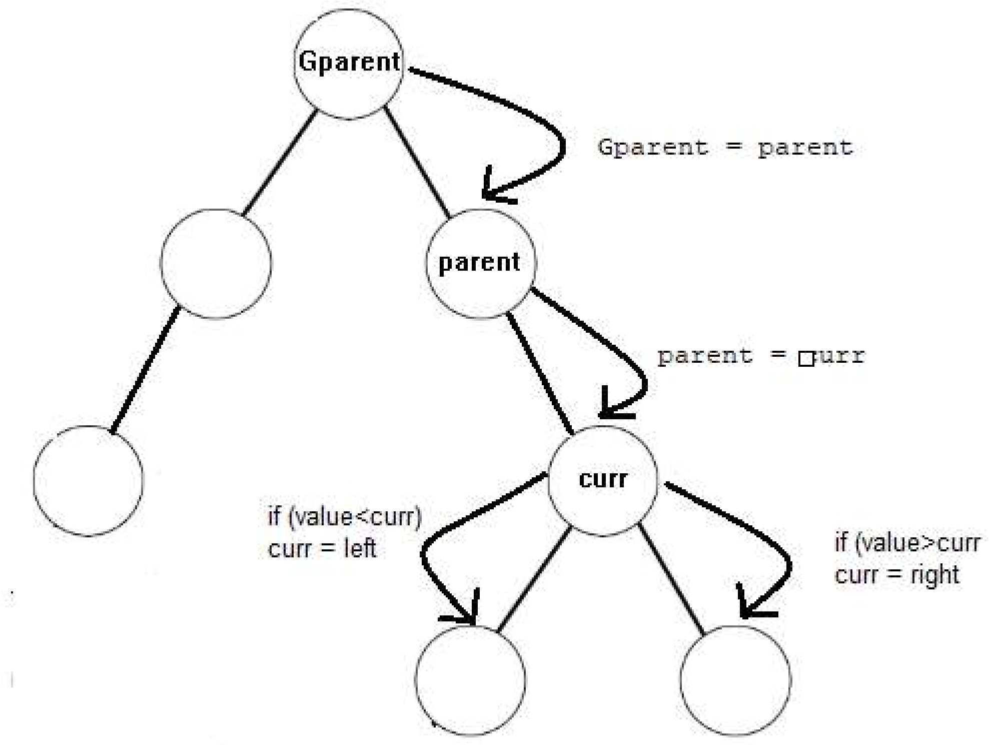


Figure 18 finding with groups of sub tree

d.Insert

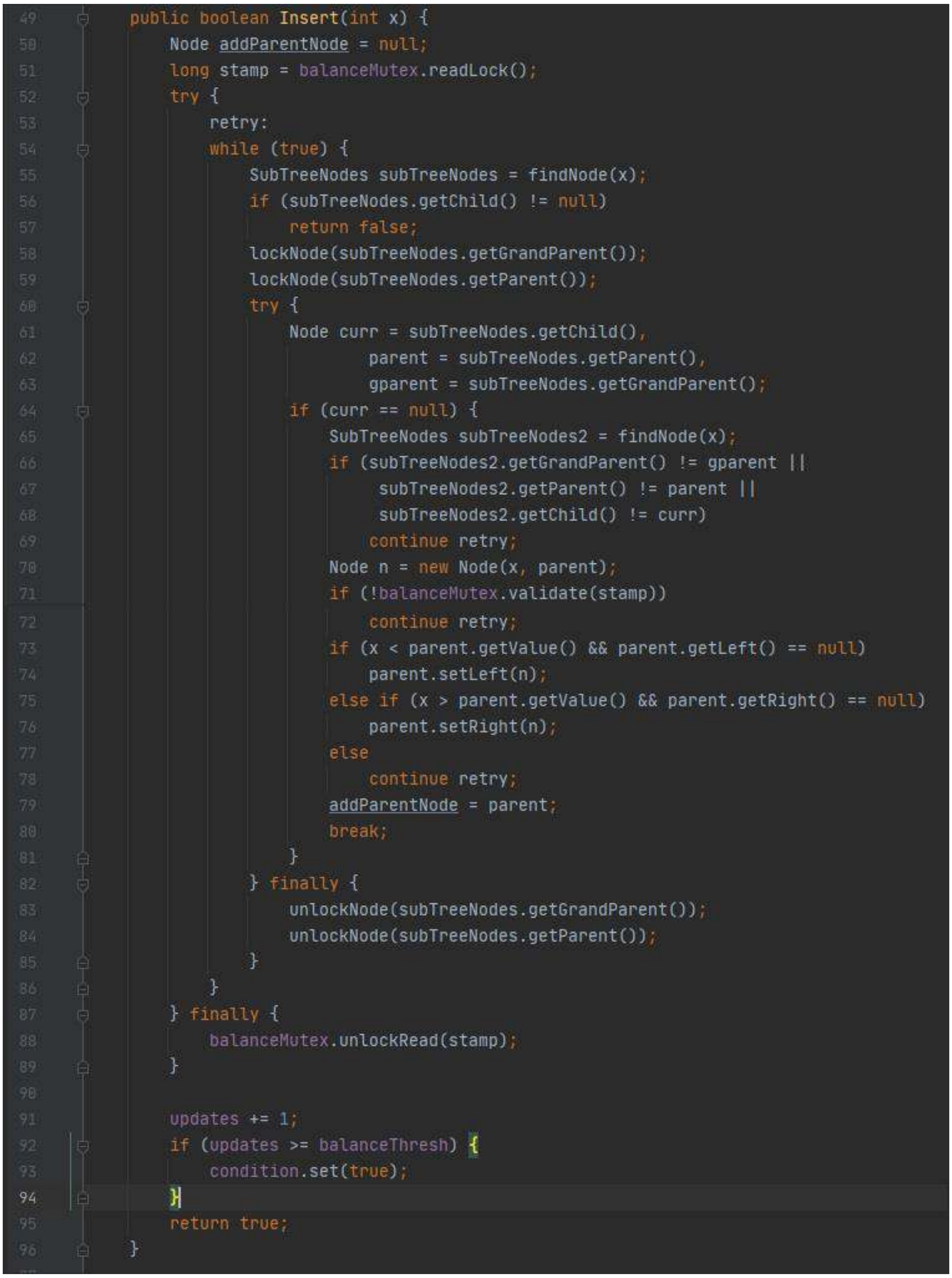
Insertion process as showing in figure 20 starts with taking a stamped read lock which is a long value used for allow optimistic locking for read operations.

After taking the read lock, a searching process starts looking for a place so we add this integer by calling find method, if the integer value is already existing in the tree, the method return false, else we get the lock for the grandparent node and the parent node and we testing if the current node equal null.

We make sure that we are in the real grandparent node and the parent node and we create a new node holding the value X that we want to add, then we test the parent node if this value X is bigger than the parent value we set the new node on the left of the parent node, and if the X value is less than the parent value we set the new node which hold this value X in the left of the parent node.

After setting the new node, we get back the grand parent and the parent lock.

Finally, we unlock the read stamped lock and updating a value that represent the counter for the rebalancing thread, if this counter reach 500 we set the Boolean condition as true to starting the balancing process.



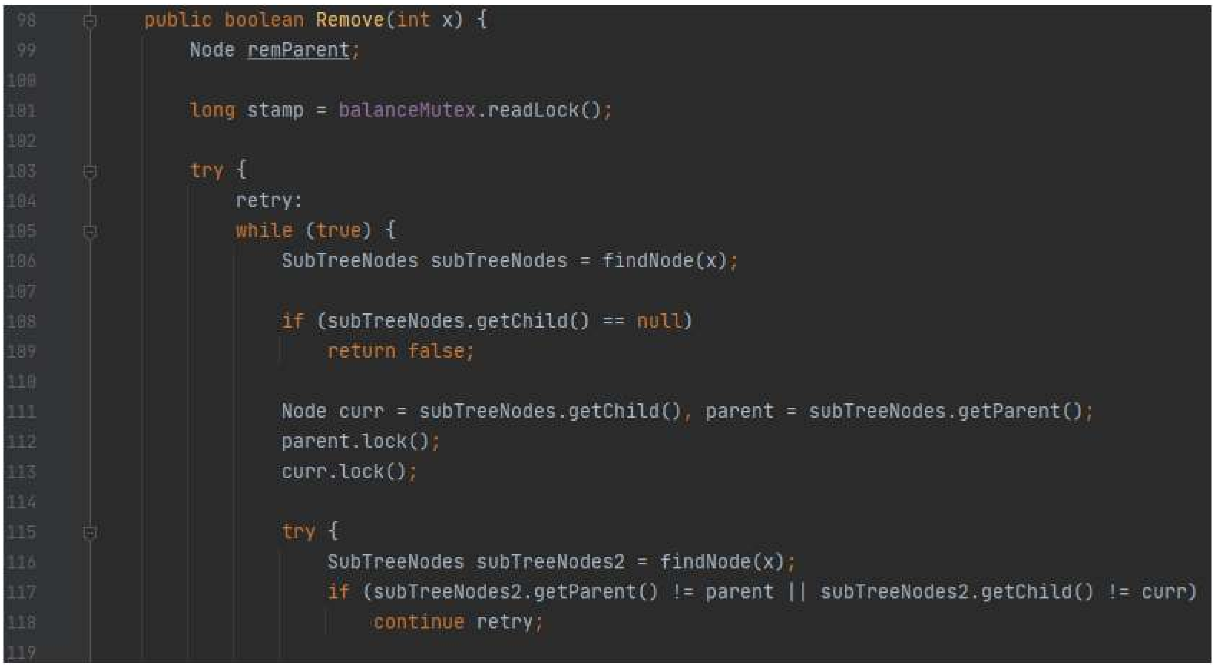
```java
public boolean Insert(int x) {
    Node addParentNode = null;
    long stamp = balanceMutex.readLock();
    try {
        retry:
        while (true) {
            SubTreeNodes subTreeNodes = findNode(x);
            if (subTreeNodes.getChild() != null)
                return false;
            lockNode(subTreeNodes.getGrandParent());
            lockNode(subTreeNodes.getParent());
            try {
                Node curr = subTreeNodes.getChild(),
                        parent = subTreeNodes.getParent(),
                        gparent = subTreeNodes.getGrandParent();
                if (curr == null) {
                    SubTreeNodes subTreeNodes2 = findNode(x);
                    if (subTreeNodes2.getGrandParent() != gparent ||
                         subTreeNodes2.getParent() != parent ||
                         subTreeNodes2.getChild() != curr)
                        continue retry;
                    Node n = new Node(x, parent);
                    if (!balanceMutex.validate(stamp))
                        continue retry;
                    if (x < parent.getValue() && parent.getLeft() == null)
                        parent.setLeft(n);
                    else if (x > parent.getValue() && parent.getRight() == null)
                        parent.setRight(n);
                    else
                        continue retry;
                    addParentNode = parent;
                    break;
                }
            } finally {
                unlockNode(subTreeNodes.getGrandParent());
                unlockNode(subTreeNodes.getParent());
            }
        }
    } finally {
        balanceMutex.unlockRead(stamp);
    }

    updates += 1;
    if (updates >= balanceThresh) {
        condition.set(true);
    }
    return true;
}
```

Figure 19 insertion method

e.remove

Removing process as showing in figure 21 start with taking a stamped read lock that we already explained hereinbefore.

After taking the read lock, a searching process start for finding the node that we want to remove it by calling find method, if the node is null returning false, otherwise we get the lock for the parent node and the current node.

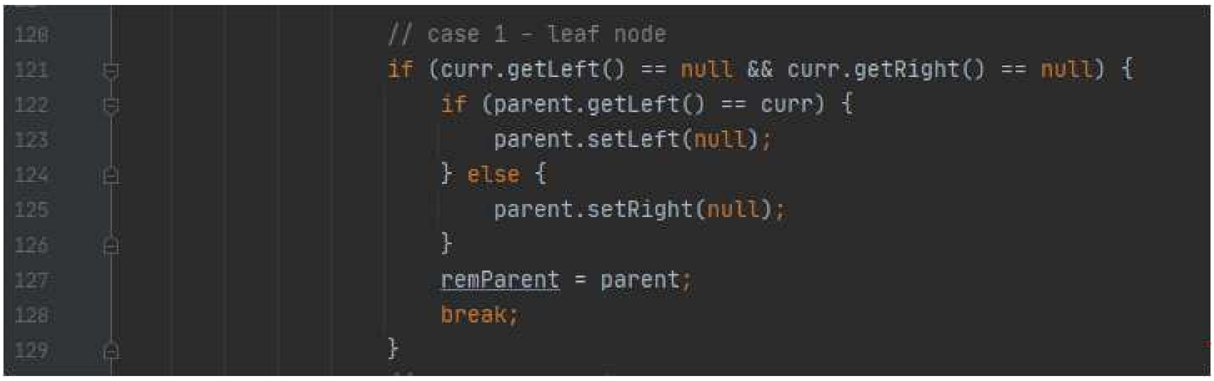


```java
public boolean Remove(int x) {
    Node remParent;

    long stamp = balanceMutex.readLock();

    try {
        retry:
        while (true) {
            SubTreeNodes subTreeNodes = findNode(x);

            if (subTreeNodes.getChild() == null)
                return false;

            Node curr = subTreeNodes.getChild(), parent = subTreeNodes.getParent();
            parent.lock();
            curr.lock();

            try {
                SubTreeNodes subTreeNodes2 = findNode(x);
                if (subTreeNodes2.getParent() != parent || subTreeNodes2.getChild() != curr)
                    continue retry;

```

Figure 20 removing method

1. *Case 1: Leaf node*

If the current node that we want to remove is a leaf, we do another test to see if the left node of the parent node is the current if it is directly, we set the parent left node to null, otherwise we set the parent right node to null and break the operation and return.



```
// case 1 - leaf node
if (curr.getLeft() == null && curr.getRight() == null) {
    if (parent.getLeft() == curr) {
        parent.setLeft(null);
    } else {
        parent.setRight(null);
    }
    remParent = parent;
    break;
}
```

Figure 21 first case of removing

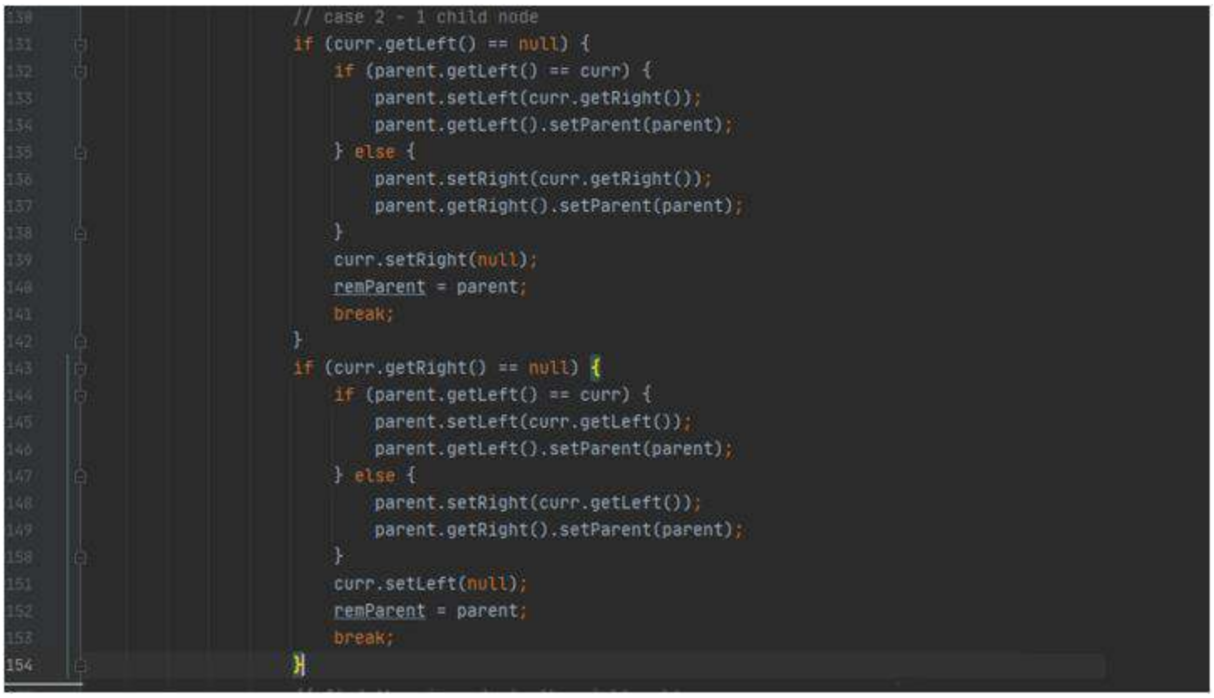
2. *Case 2 :1 child node*

There is another case where the current node that we want to remove has a child, in this case we test if the left node of current node is null this will give us two choices , we test if the left parent equal current node in this case we set the right node of the current node in the left of the parent node, otherwise set the right node of the current in the right of parent node and setting null in the right of current node.

The next case ,the right node of current node is null this will give us two choices again, we test if the left parent equal current node in this case we set the left node of the current node in the left of the parent node, otherwise set the left node of the current in the right of parent node and setting null in the right of current node.

After that we unlock the current node and the parent node.

Finally, we unlock the read stamped lock and updating a value that represent the counter for the rebalancing thread, as we explained in the previous lecture.



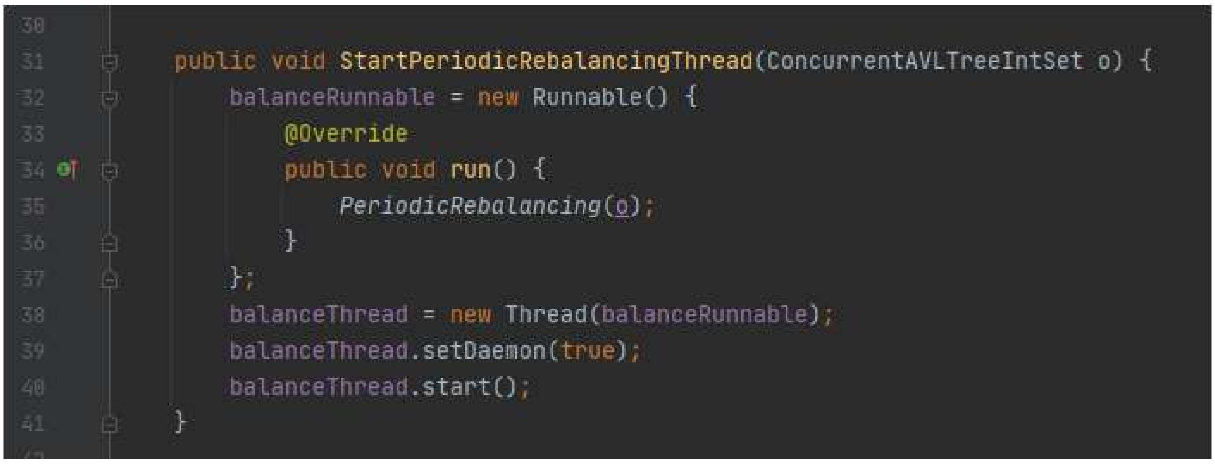
```java
// case 2 - 1 child node
if (curr.getLeft() == null) {
    if (parent.getLeft() == curr) {
        parent.setLeft(curr.getRight());
        parent.getLeft().setParent(parent);
    } else {
        parent.setRight(curr.getRight());
        parent.getRight().setParent(parent);
    }
    curr.setRight(null);
    remParent = parent;
    break;
}
if (curr.getRight() == null) {
    if (parent.getLeft() == curr) {
        parent.setLeft(curr.getLeft());
        parent.getLeft().setParent(parent);
    } else {
        parent.setRight(curr.getLeft());
        parent.getRight().setParent(parent);
    }
    curr.setLeft(null);
    remParent = parent;
    break;
}
```

Figure 22 second case of removing

f.StartPeriodicRebalancingThread:

The StartPeriodicRebalancingThread is a method that allows to start a Runnable thread to rebalancing the tree after every 500 insert/remove operation happened, we call in by adding an atomic Boolean condition assigned by false.

In every remove or insert operation there is a counter that counts until 500 operation, than assign condition as true this will call this method to start balancing the tree.



```java
public void StartPeriodicRebalancingThread(ConcurrentAVLTreeIntSet o) {
    balanceRunnable = new Runnable() {
        @Override
        public void run() {
            PeriodicRebalancing(o);
        }
    };
    balanceThread = new Thread(balanceRunnable);
    balanceThread.setDaemon(true);
    balanceThread.start();
}
```

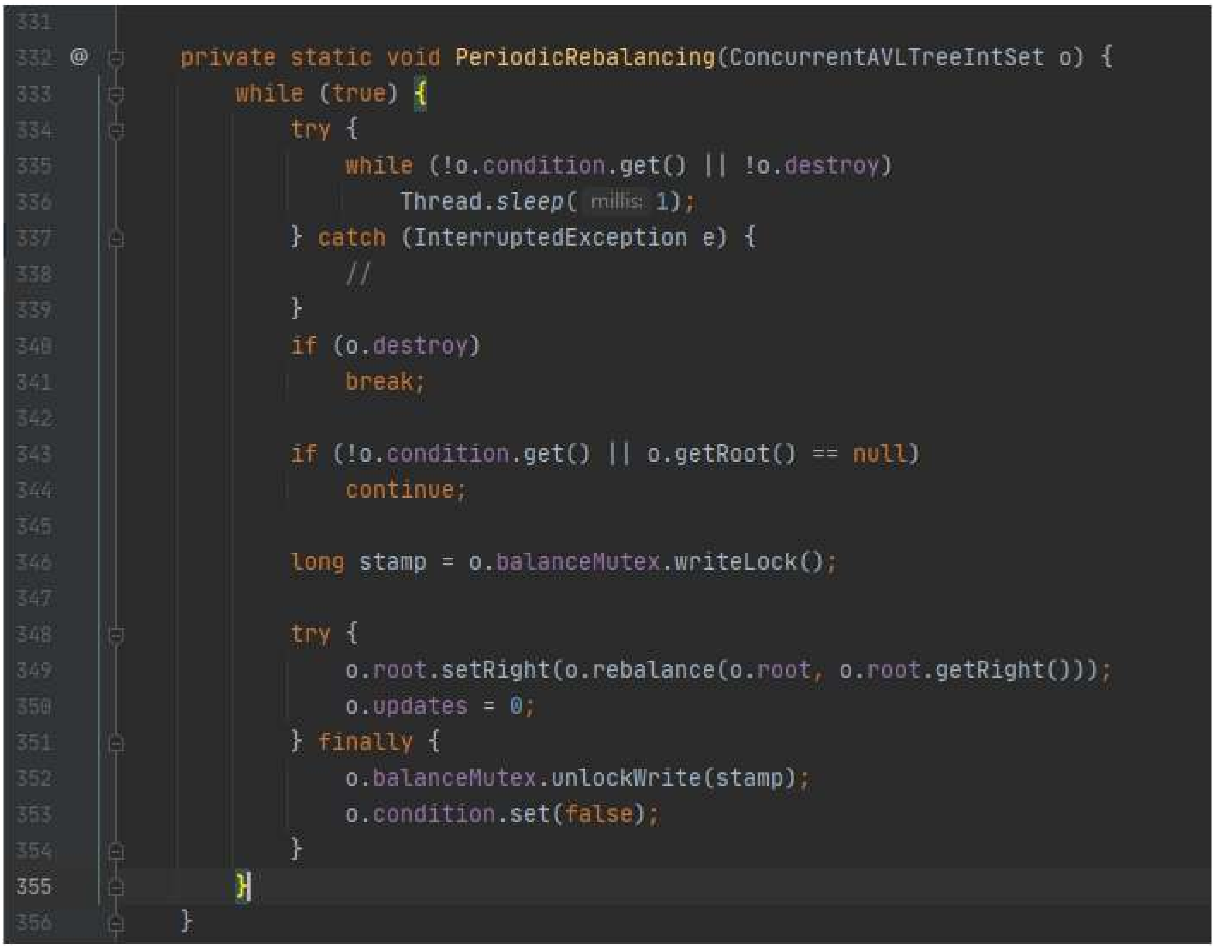
Figure 23 start Periondic Rebalance Thread

g.Periodic Rebalancing

The periodic rebalancing process starting by the StartPeriodicRebalancingThread when the condition setting true, it testing the condition value if it is true, then creating an object represented a sequence of bytes that includes a write lock.

This object takes a priority in processing because it already in bytes, after writing the lock it call rebalance method to rebalancing the tree.

This process end after rebalancing all node needed to rebalance in the tree, then it releases the writing lock and set the condition to false by using CAS (Compare and Swap).



```

    private static void PeriodicRebalancing(ConcurrentAVLTreeIntSet o) {
        while (true) {
            try {
                while (!o.condition.get() || !o.destroy)
                    Thread.sleep( millis: 1);
            } catch (InterruptedException e) {
                //
            }
            if (o.destroy)
                break;

            if (!o.condition.get() || o.getRoot() == null)
                continue;

            long stamp = o.balanceMutex.writeLock();

            try {
                o.root.setRight(o.rebalance(o.root, o.root.getRight()));
                o.updates = 0;
            } finally {
                o.balanceMutex.unlockWrite(stamp);
                o.condition.set(false);
            }
        }
    }
```

Figure 24 periodic rebalance

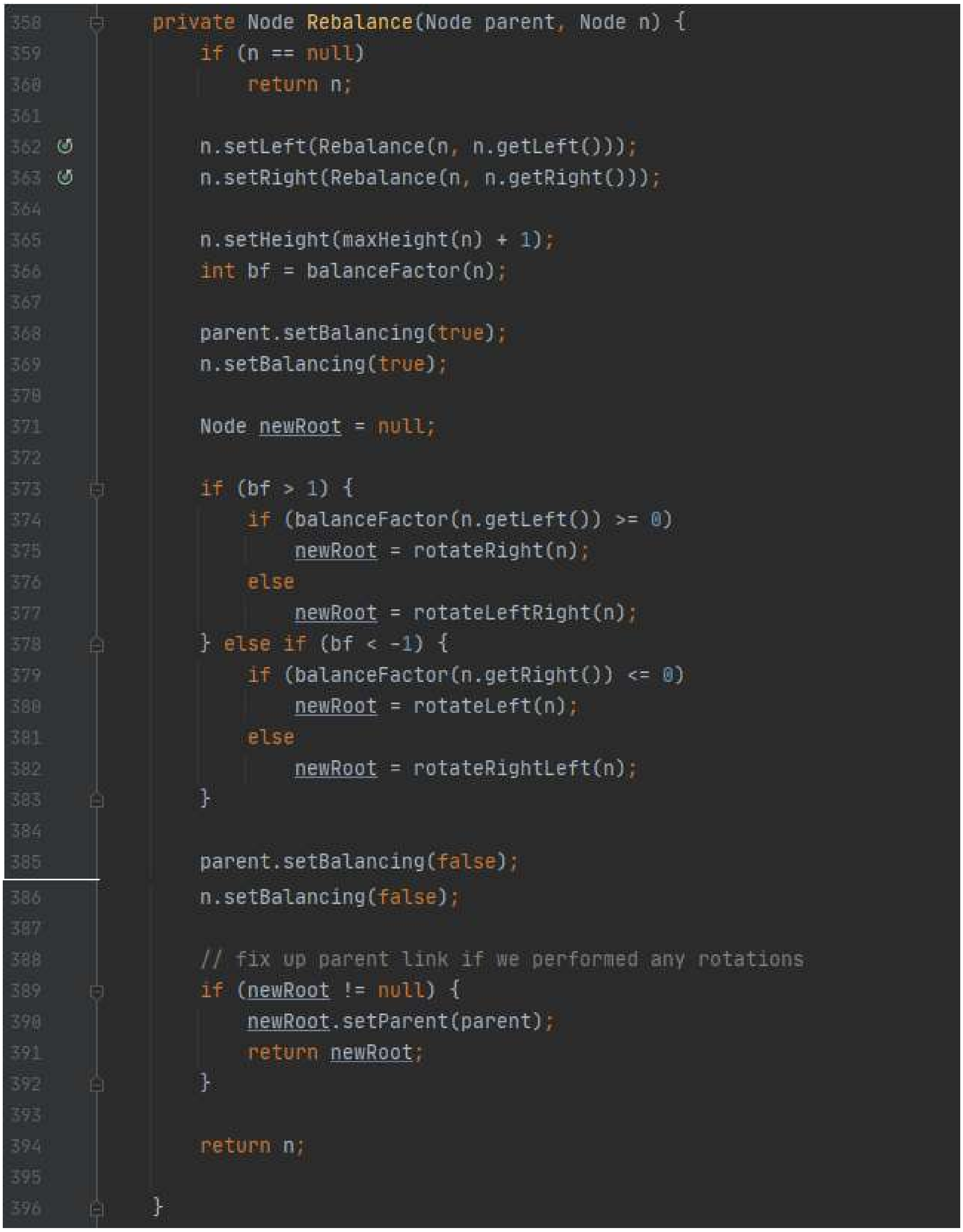
h.Rebalance

The rebalancing method showed in figure 27 start after a call by the periodic process, this method takes the root node after that it calling herself recursively by calling the left sub-tree then the right sub-tree.

For each call we set the Boolean value balancing as true for each current and parent node, this Boolean value Aimed at preventing any thread to read or write for the current and parent node that we want to rebalance, After that it count the height of each node then count the balance factor of each parent node, if the balance factor is bigger than one, and the balance factor of left node greater or equal zero we perform a right rotation, else we do a left right rotation.

If the BF < -1, and the balance factor of right node smaller or equal zero we perform a left rotation, else we do a right left rotation.

After that, we reset the Boolean value balancing as false so that will allow the write/read on them.



```java
private Node Rebalance(Node parent, Node n) {
    if (n == null)
        return n;

    n.setLeft(Rebalance(n, n.getLeft()));
    n.setRight(Rebalance(n, n.getRight()));

    n.setHeight(maxHeight(n) + 1);
    int bf = balanceFactor(n);

    parent.setBalancing(true);
    n.setBalancing(true);

    Node newRoot = null;

    if (bf > 1) {
        if (balanceFactor(n.getLeft()) >= 0)
            newRoot = rotateRight(n);
        else
            newRoot = rotateLeftRight(n);
    } else if (bf < -1) {
        if (balanceFactor(n.getRight()) <= 0)
            newRoot = rotateLeft(n);
        else
            newRoot = rotateRightLeft(n);
    }

    parent.setBalancing(false);
    n.setBalancing(false);

    // fix up parent link if we performed any rotations
    if (newRoot != null) {
        newRoot.setParent(parent);
        return newRoot;
    }

    return n;

}
```

Figure 25 rebalance method

B.Benchmarking

We evaluate the performance of our algorithm, the benchmarked implementation of our algorithm is written in Java.by the JMH benchmark library, unfortunately for a resources and time constraints we did not get a strict an accurate calculation to compare it with previous researching results.

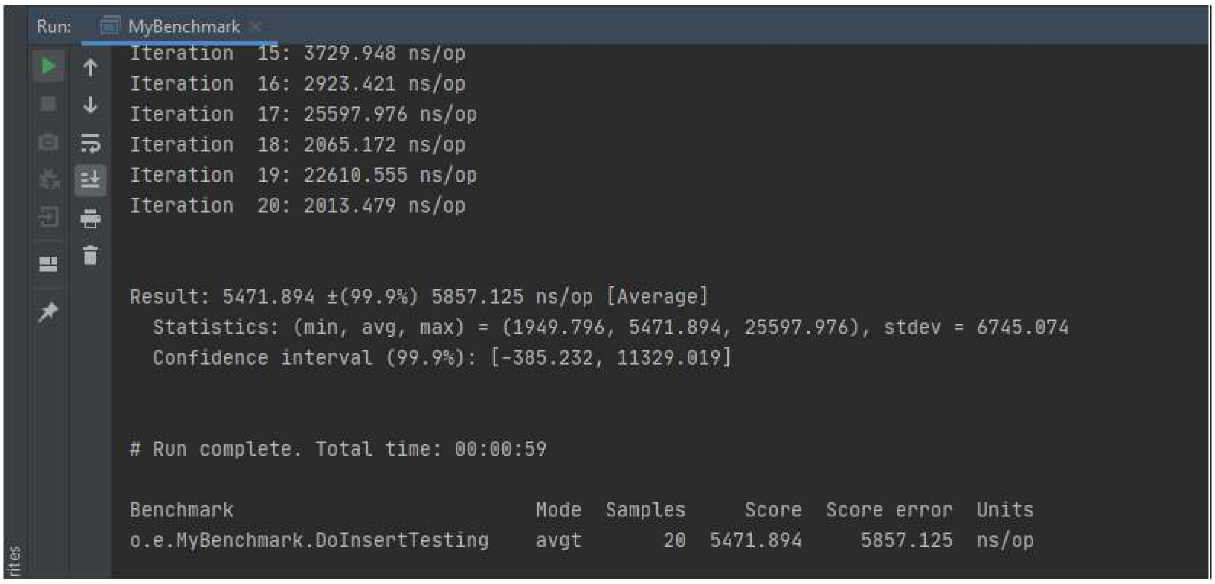
Experiments were run on a HP ProBook with 4 core 1.6 GHz Intel Core i5-8250U processor, and 8GB of RAM. Hyper-Threading was enabled, we ran our experiments in windows 10.

.

We get an average time for insertion as:

0.000385 MS /operation in minimum.

0.005471 milli second/operation in the average time of insertions.



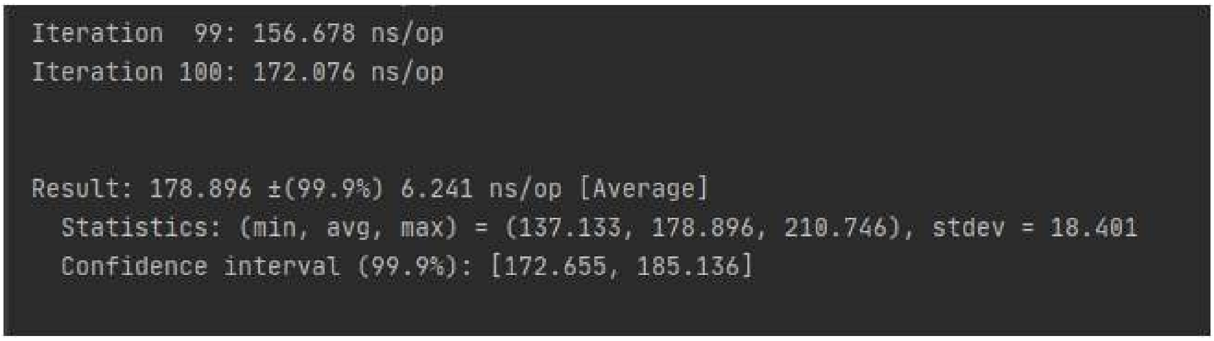
```
Iteration  15: 3729.948 ns/op
Iteration  16: 2923.421 ns/op
Iteration  17: 25597.976 ns/op
Iteration  18: 2065.172 ns/op
Iteration  19: 22610.555 ns/op
Iteration  20: 2013.479 ns/op


Result: 5471.894 ±(99.9%) 5857.125 ns/op [Average]
  Statistics: (min, avg, max) = (1949.796, 5471.894, 25597.976), stdev = 6745.074
  Confidence interval (99.9%): [-385.232, 11329.019]


# Run complete. Total time: 00:00:59

Benchmark                          Mode  Samples     Score  Score error  Units
o.e.MyBenchmark.DoInsertTesting    avgt       20  5471.894     5857.125  ns/op
```

Figure 26 Benchmark result

In the deletion we get an average time of

0.000178 MS /operation in minimum.

0.000185 milli second/operation in the average time of delating.



```
Iteration  99: 156.678 ns/op
Iteration 100: 172.076 ns/op


Result: 178.896 ±(99.9%) 6.241 ns/op [Average]
  Statistics: (min, avg, max) = (137.133, 178.896, 210.746), stdev = 18.401
  Confidence interval (99.9%): [172.655, 185.136]
```

C.Further Work

a.Optimization with VarHandler:

A VarHandle is a dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure. Access to such variables is supported under various access modes, including plain read/write access, volatile read/write access, and compare-and-swap.

VarHandles are immutable and have no visible state. VarHandles cannot be subclassed by the user.

b.Optimization with CAS (Compare And Swap):

The compare-and-swap (CAS) instruction is an uninterruptible instruction that reads a memory location, compares the read value with an expected value, and stores a new value in the memory location when the read value matches the expected value. Otherwise, nothing is done. The actual microprocessor instruction may differ somewhat (e.g., return true if CAS succeeded or false otherwise instead of the read value).

Or Optimization with Wait-Free and these two techniques.

Conclusion

We may have tolerated with deadlocks and, we left it with the JVM dealing, in the same time we obeyed the rules that said:

- don't hold several locks at once. If you do, always acquire the locks in the same order.
- don't execute foreign code while holding a lock.
- use interruptible locks.

For time purpose we couldn’t Complete it to handling with object, we advise Who want to complete this project to read the further work and implement it by using comparable and comparator interface.

Finally, we can say that this implementation delivers high performance and good scalability especially when we are using optimistic locking and a periodic rebalancing thread to balance the tree.

This algorithm is faster than the ConcurrentSkipListSet that provide by Java.

General Conclusion

As a result, we can say that our work which depended on AVL BST data structure and java concurrency techniques is a worthwhile search, we get good results and offered high performance in this algorithm because we used a High speed and quality technologies Offred by ORACLE developers, these kinds of algorithms are developing rapidly due to ongoing research by ORACLE and University researchers.

Table of Figures

Bibliography

A Skip List Cookbook by William Pugh

Data Structures in Java for the Principled Programmer 7 edition by Duane A. Bailey.

Introduction to Process Synchronization Using the *Java* Language by Harry J. Foxwell.

Introduction to Algorithms *Third Edition by* Thomas H. Cormen, Charles E. Leiserson,Ronald L. Rivest

Insertion and Deletion In AVL Tree by Dr. Erich Kaltofen

Java Concurrency in Practice by brian Goetz**.**

Paper Linked Lists: Locking, Lock-Free by Maurice Herlihy

ppopp207-bronson: A Practical Concurrent Binary Search Tree by Nathan G. Bronson Jared Casper Hassan Chafi Kunle Olukotun

Transactions and Concurrency Control by Vlad Mihalcea

The Java® Language Specification *Java SE 8 Edition by* James Gosling,Bill

Joy,Guy Steele,Gilad Bracha,Alex Buckley.

Webography

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks
https://docs.oracle.com/javase/jndi/tutorial/objects/storing/serial.html#:~:text=To%20serialize%20an%20object%20means,io.
https://algorithms.tutorialhorizon.com/avl-tree-insertion/

https://www.programiz.com/dsa/avl-tree

https://dzone.com/articles/how-detect-java-deadlocks

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html

https://fy.blackhats.net.au/blog/html/2019/12/29/concurrency_2_concurrently_readable_structures.html