Documentation of First Assignment from DSA

It contains:

Code description,

Comparison and analysis

Presenter:

Chesnykov Vladyslav

Place:

STU BA FIIT

Software:

IntelliJ IDEA

Computer:

Hardware Model	ASUSTeK COMPUTER INC. ASUS TUF Dash F15 FX516PR_FX516PR
Memory	16.0 GiB
Processor	11th Gen Intel® Core™ i7-11370H × 8
Graphics	Software Rendering / Mesa Intel® Xe Graphics (TGL GT2)
Disk Capacity	512.1 GB

==========

Provided implementation in Java:
AVL Tree, Splay Tree, HT with Open Addressing, HT with Separated Chains.
STU BA, FIIT.
Chesnykov Vladyslav.

- Implementation of AVL Tree: *(I describing main points of my code, all other will be covered in comments in code document)*

Insert function:

In my code you can find algorithm where I in this part of code making an update of the node's height:

```
1 usage
public void updateHeight(Node node) {
    node.height = Math.max(getHeight(node.left), getHeight((node.right))) + 1;
}
```

It works as increasing of the maximum value of last nodes of right and left sub-trees.

Then my algorithm by recursion searching the place for new node:

```
} else if(node.key < key) {
    node.right = insert(node.right, key, data);
    if(getBalanceCoef(node) < -1) {
```

And here:

```
} else if(node.key > key) {
    node.left = insert(node.left, key, data);
    if(getBalanceCoef(node) > 1) {
```

Here I made all sorts of rotation if right sub-tree is bigger then left:

```
if(getBalanceCoef(node) < -1) {
    if (key > node.right.key) {
        node = rotateWithRight(node);
    } else {
        node.right = rotateWithLeft(node.right);
        node = rotateWithRight(node);
    }
```

And on this picture you can see all sorts of rotations if left sub-tree is bigger then right:

```
if(getBalanceCoef(node) > 1) {
    if (key < node.left.key) {
        node = rotateWithLeft(node);
    } else {
        node.left = rotateWithRight(node.left);
        node = rotateWithLeft(node);
    }
```

Delete function:

```
} else {
    if(node.left == null || node.right == null) {
        Node temp = null;
        if (node.left == null) {
            temp = node.right;
        } else {
            temp = node.left;
        }
        node = temp;
```

If node left child or right (or both of them) will be null, then we will rise the node which is not null (or return null).

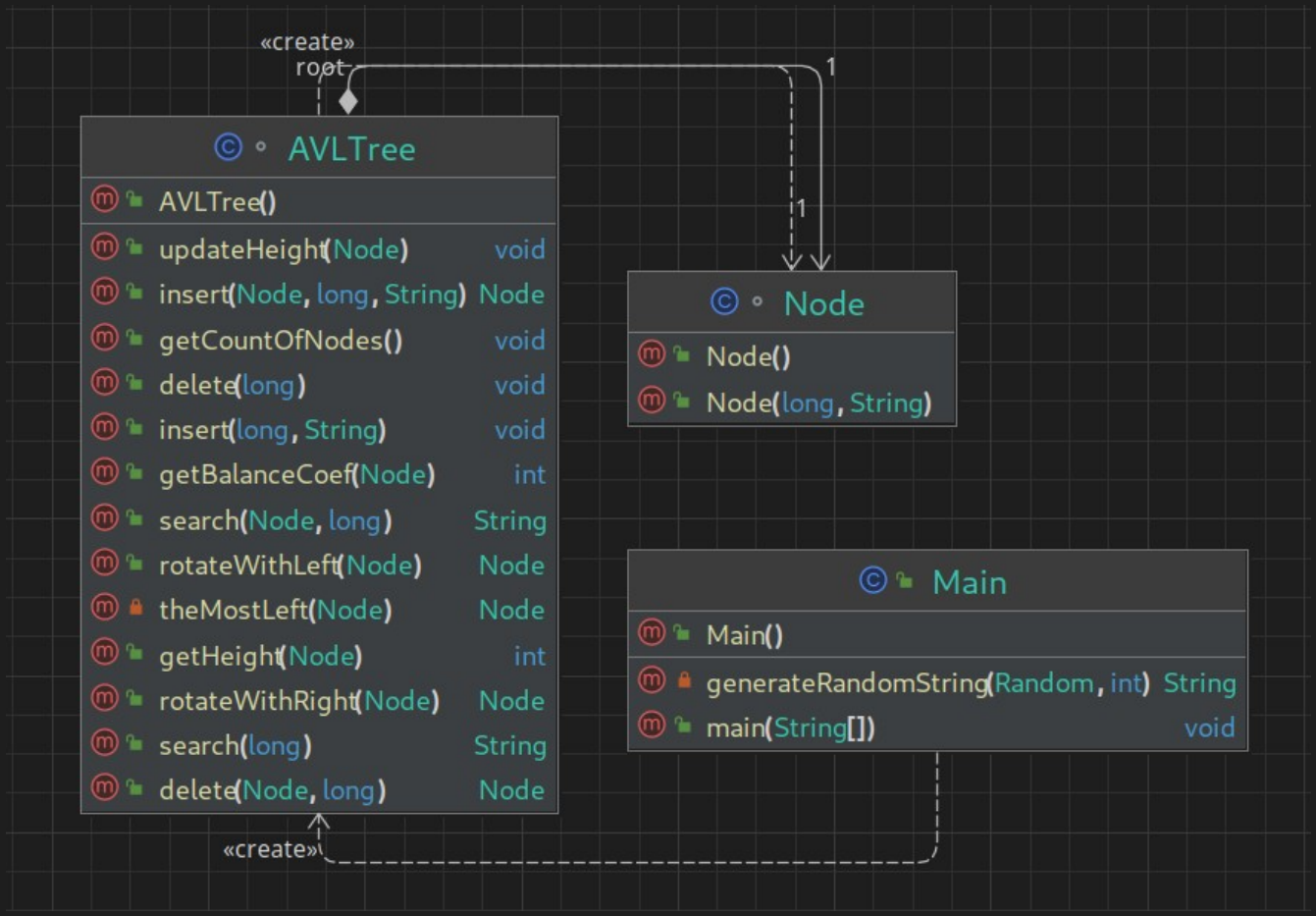
If all the children exist, then we will find the most left child of right sub-tree, because of the conception of BST, as all right elements is bigger then parent element, and all left are less. And that is why this element must be the nearest to parent element.

```
    } else {
        Node temp = theMostLeft(node.right);
        node.key = temp.key;
        node.data = temp.data;
        node.right = delete(node.right,temp.key);
    }
}
return node;
```

Search function:

```
while (node != null && data.equals("Is not found...")) {
    if (node.key > key) {
        node = node.left;
    } else if (node.key < key) {
        node = node.right;
    } else {
        data = node.data;
        break;
    }
    data = search(node, key);
}
```

In this cycle I use recursion for finding the element which is match to the key user write.



- Implementation of Splay Tree: *(I describing main points of my code, all other will be covered in comments in code document)*

  Insert function:

```
public void insert(long key, String data) { root = insert(root, key, data); }
3 usages
public Node insert(Node node, long key, String data) {
    if(node == null) {
        node = new Node(key, data);
    } else if(node.key > key) {
        node.left = insert(node.left, key, data);
    } else if(node.key < key) {
        node.right = insert(node.right, key, data);
    }
    return splay(node, key);
}
```

In recursion I searching the place for my element as in any BST, then I use splay function.

  Delete function:

```
public Node delete(Node node, long key) {
    if (node == null) {
        return null;
    }
    node = splay(node, key);
    if (node.key != key) {
        return node;
    }
    if (node.left == null) {
        node = node.right;
    } else {
        Node rightSubtree = node.right;
        node = node.left;
        node = splay(node, key);
        node.right = rightSubtree;
    }
    return node;
}
```

On the picture you can see that code will find the element and with the help of splay function will rise the element on the top to the root. There I chose the variant where left child will be put on the top, as a new root, and its right sub-tree will be exchanged with the previous root's right sub-tree.

Search function:

```
public void search(long key) {
    root = search(root, key);
    if (check) {
        if (root.key == key) {
            System.out.println(root.data);
        } else {
            System.out.println("This node must be deleted, or does not exist.");
        }
    } else
        System.out.println(root.data);
}
```

By recursion will find the element and print its data (or print that it is not exist).

```
public Node search(Node node, long key) { return splay(node, key); }
```

And will make a splay rotation will found element.

Splay function:

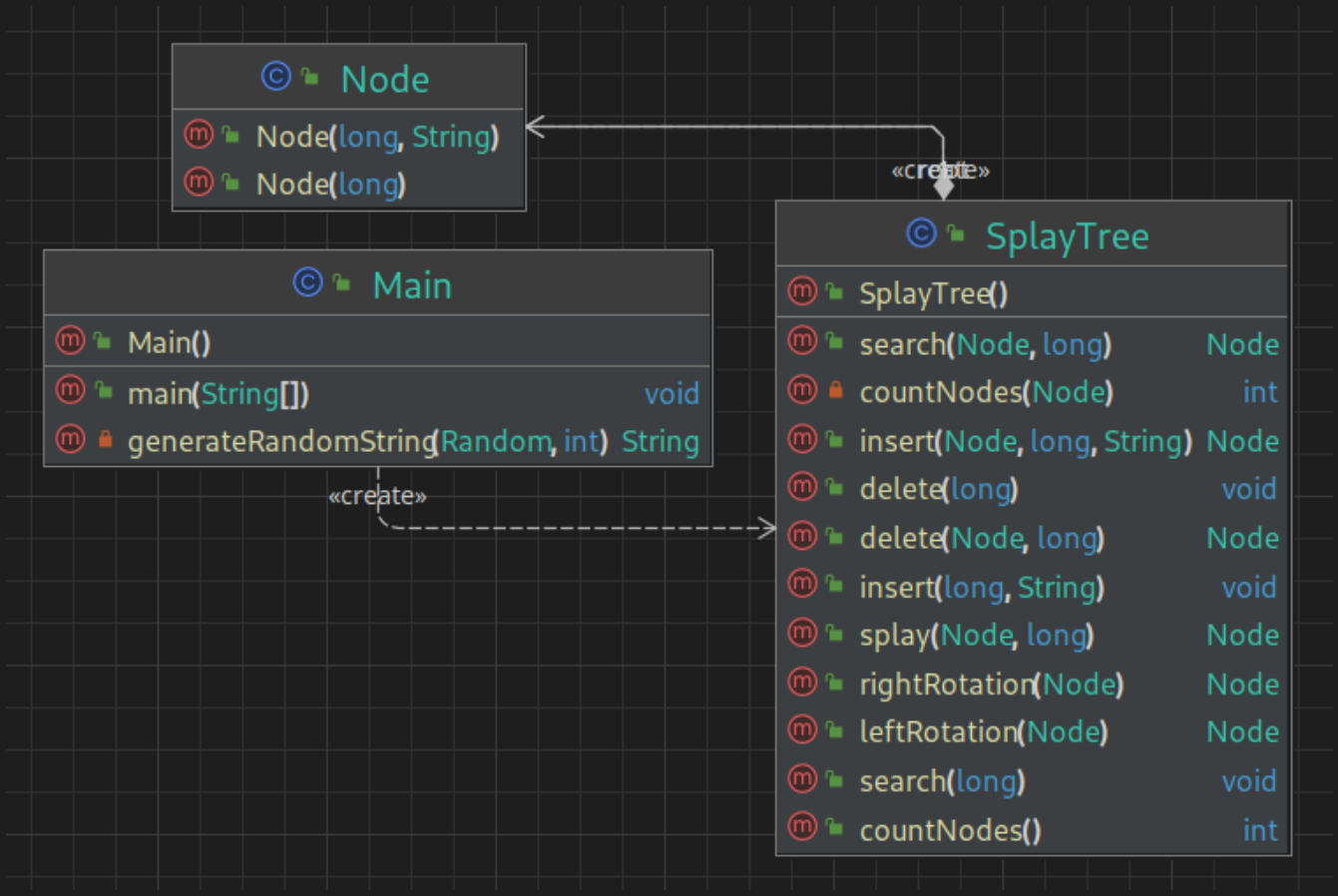
This part will check if the node has the same key user entered, then its node will be returned 'check' factor will be changed to 'true'.

```
if(node == null || node.key == key) {
    check = true;
    return node;
}
```

And this part of code will return in each checking if node.key has more or less value of user's key.

```
if (node.key > key) {
    if(node.left == null) {
        check = true;
        return node;
    }
    if(node.left.key > key) {
        node.left.left = splay(node.left.left, key);
        if (node.left.left != null) {
            node = rightRotation(node);
        }
    } else if (node.left.key < key){
        node.left.right = splay(node.left.right, key);
        if (node.left.right != null) {
            node = leftRotation(node);
        }
    }
} else {
    if(node.right == null) {
        check = true;
        return node;
    }
    if (node.right.key > key) {
        node.right.left = splay(node.right.left, key);
        if(node.right.left != null) {
            node = rightRotation(node);
        }
    } else if (node.right.key < key) {
        node.right.right = splay(node.right.right, key);
        if(node.right.right != null) {
            node = leftRotation(node);
        }
    }
    if (node.right == null) {
        check = true;
        return node;
    } else {
        return leftRotation(node);
```

Here you can see, that we have 2 steps of comparison: first, when we figure out the key value of **node.left** is bigger then user's key or less(and in every case we have 2 additional checks **node.left.left** and **node.left.right**, or **node.right.left** and **node.right.right**). For each variant we have its own sort of rotation. And by recursion we make the amount of rotation which is needed for rising the element on the top of tree.



- Implementation of HT with Open Addressing: *(I describing main points of my code, all other will be covered in comments in code document)*

This is my Hash function:

```
public int getIndex(String key) {
    int hash = 5381;
    for (int i = 0; i < key.length(); i++) {
        hash = ((hash << 5) + hash) + key.charAt(i);
    }
    return Math.abs(hash) % capacity;
}
```

This hash function makes hash code from the string we have first any number, for instance 5381, then in binary format I move the number on 5 position, and then add binary code of 5381 and my character binary code. It provides me good hash code. And then divide it by capacity of my array.

For making me code faster I made main array not as ArrayList of LinkedList:

```
public class HashTable {
    15 usages
    Node[] arrayHash;
    11 usages
```

On the right picture is algorithm of searching free node in array, as in any Open Addressing hash-table.

And if the count of nodes (**'size'**) is bigger then **overloadIndex*capacity**, then hash-table will be **increased and rehashed**.

```
if (arrayHash[index] != null) {
    while (arrayHash[index] != null) {
        index++;
        if (index == capacity) {
            index = 0;
        }
    }
}
arrayHash[index] = newItem;
size++;
```

With the help of 'temp' I can increase capacity of my main array, and reinitialize all of the nodes. And then insert all the element from 'temp' into new array.

```
public void rehashIncrease() {
    Node[] temp = arrayHash;
    arrayHash = new Node[2 * capacity];
    capacity *= 2;
    size = 0;
    for (Node node : temp) {
        if (node != null) {
            insert(node.key, node.data);
        }
    }
}
```

Delete function:

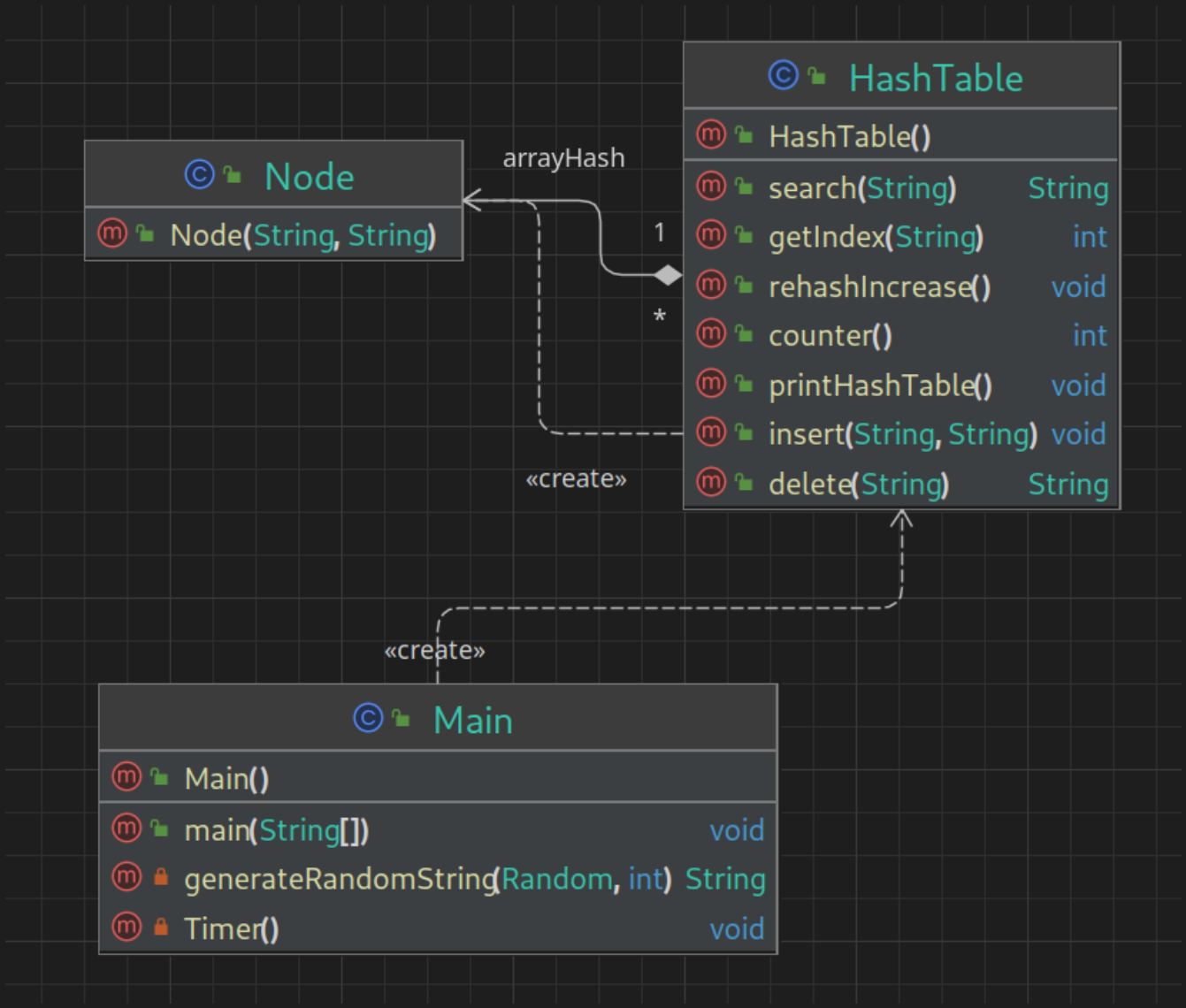
My delete function is repeating insert part in searching the element I need to delete, but if in 'insert' I put the value into the node, then in delete I set the node's value to null.

```
if (arrayHash[index].key.equals(key)) {
    arrayHash[index] = null;
    size--;
    return "Node deleted.";
}
```

Search function:

In my search function I have ‘String data’ where must be save value of searching element if it will be found. And all the code the same as in ‘delete’ or ‘insert’.

```
while (arrayHash[index] != null) {
    if (arrayHash[index].key.equals(key)) {
        return arrayHash[index].data;
    }
```



- Implementation of HT with Separated Chains: *(I describing main points of my code, all other will be covered in comments in code document)*

  Insert function:

This part of code contains the algorithm of searching place to insert the element:

```
public void insert(typeKey key, typeData data) {
    int index = getIndex(key);
    Node<typeKey, typeData> head = arrayHash.get(index);
    while(head != null) {
        if(head.key.equals(key)) {
            head.data = data;
            return;
        }
        head = head.next;
    }
```

Code will find the place through hash function:

```
public int getIndex(typeKey key) {
    int hashCode = key.hashCode();
    while (hashCode < size) {
        hashCode += size;
    }
    return hashCode % size;
}
```

Here I used standart library for getting the hash from the object.

After this condition, as in HT with Open Addressing hash-table will be increased and rehashed:

```java
public void rehashIncrease() {
    ArrayList<Node<typeKey, typeData>> temp = arrayHash;
    arrayHash = new ArrayList<Node<typeKey, typeData>>( initialCapacity: 2 * size);
    for (int i = 0; i < 2 * size; i++) {
        arrayHash.add(null);
    }
    countOfNodes = 0;
    size *= 2;
    for (Node<typeKey, typeData> head : temp) {
        while (head != null) {
            insert(head.key, head.data);
            head = head.next;
        }
    }
}
```

All the same as in 'HT_OA', but with using the ability of choosing your own type.

Delete function:

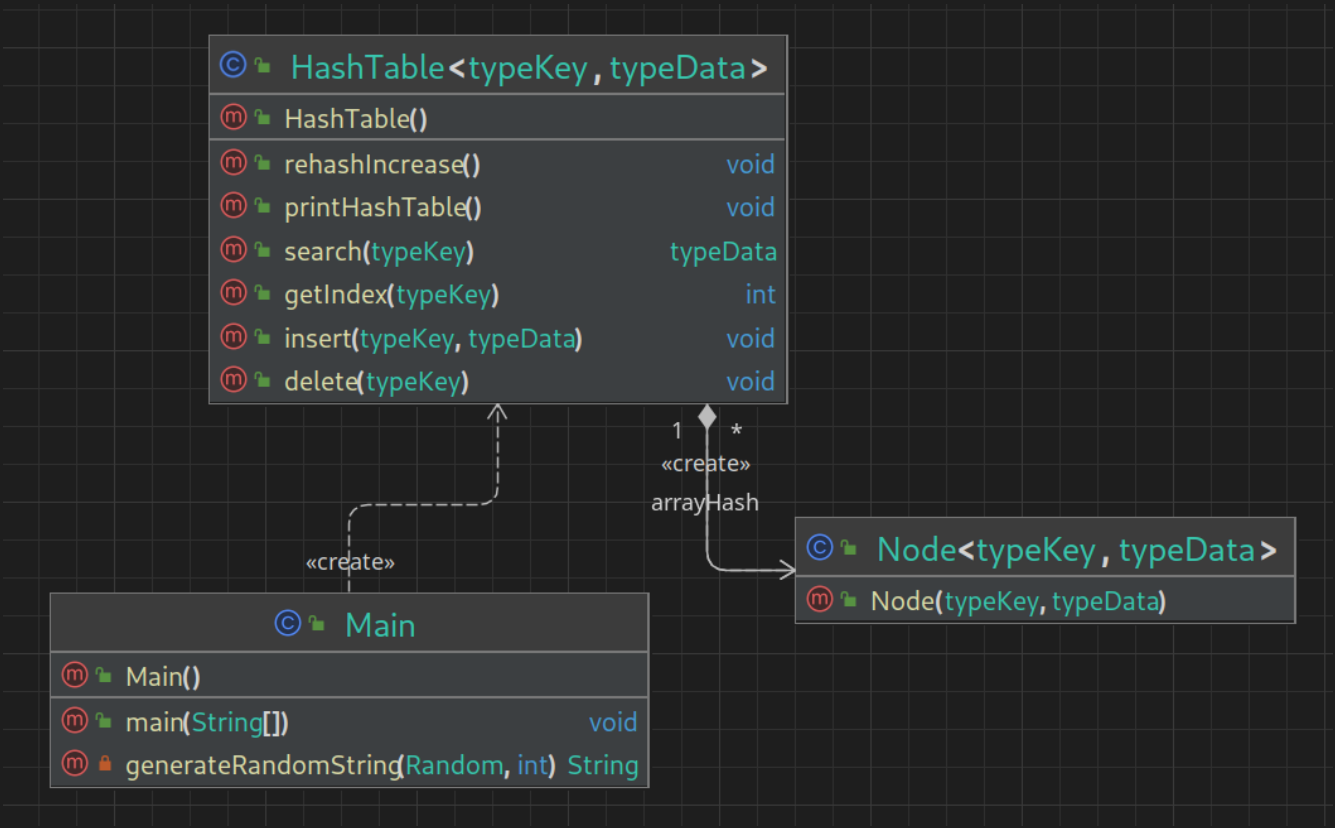
Here I searching the element and set it to null:

```java
for (int i = 0; i < arrayHash.size(); i++) {
    if (head != null) {
        if(head.key.equals(key)) {
            if (prev == null) {
                arrayHash.set(removingIndex, head.next);
            } else {
                prev.next = head.next;
            }
            countOfNodes--;
            return;
        }
        prev = head;
        head = head.next;
```

Search function:

This 'while' will find the element we searching by comparing its key with user's key.

```java
while(temp != null) {
    if(temp.key.equals(key)) {
        return temp.data;
    }
    temp = temp.next;
```



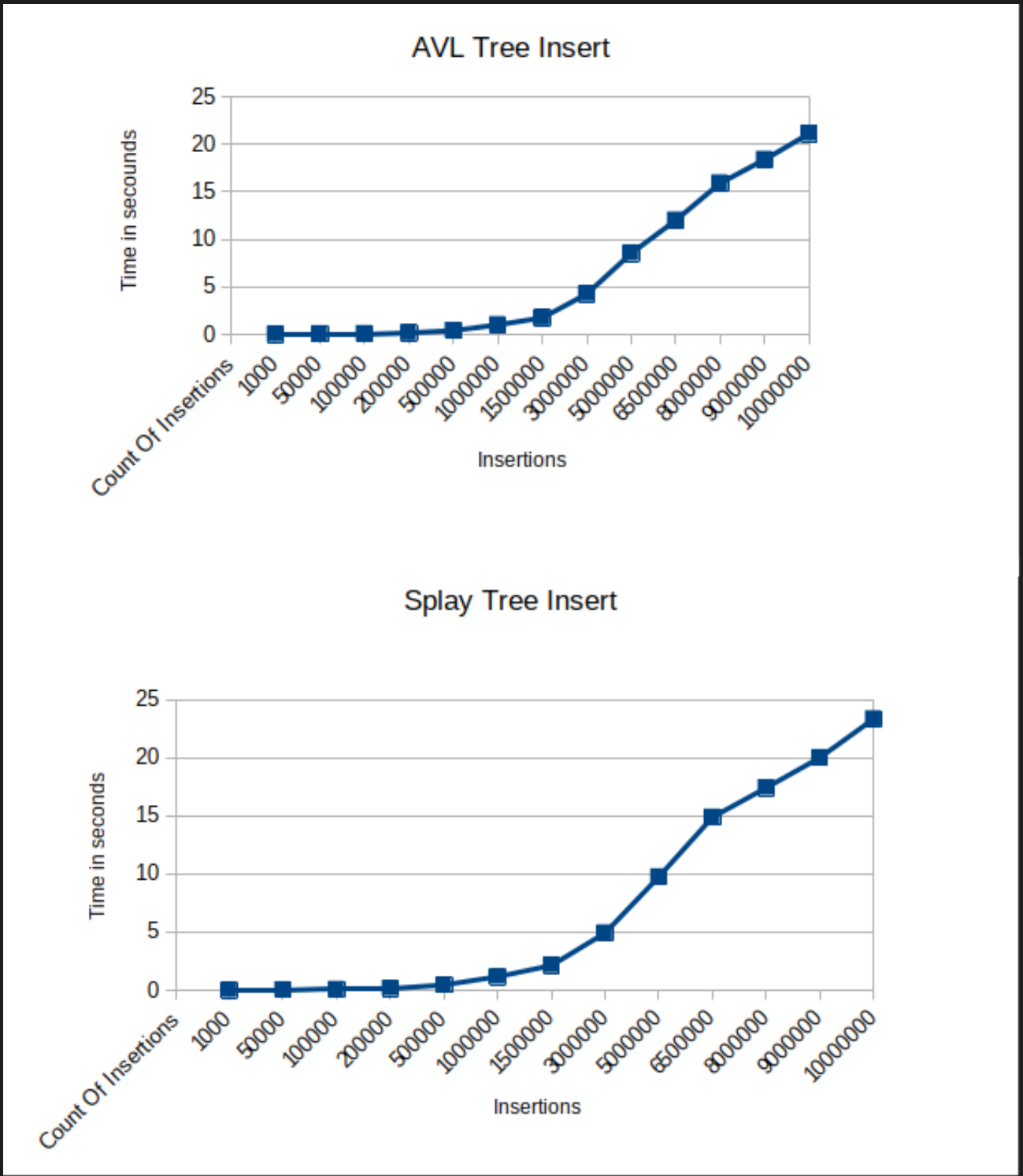
Comparisons & conclusions

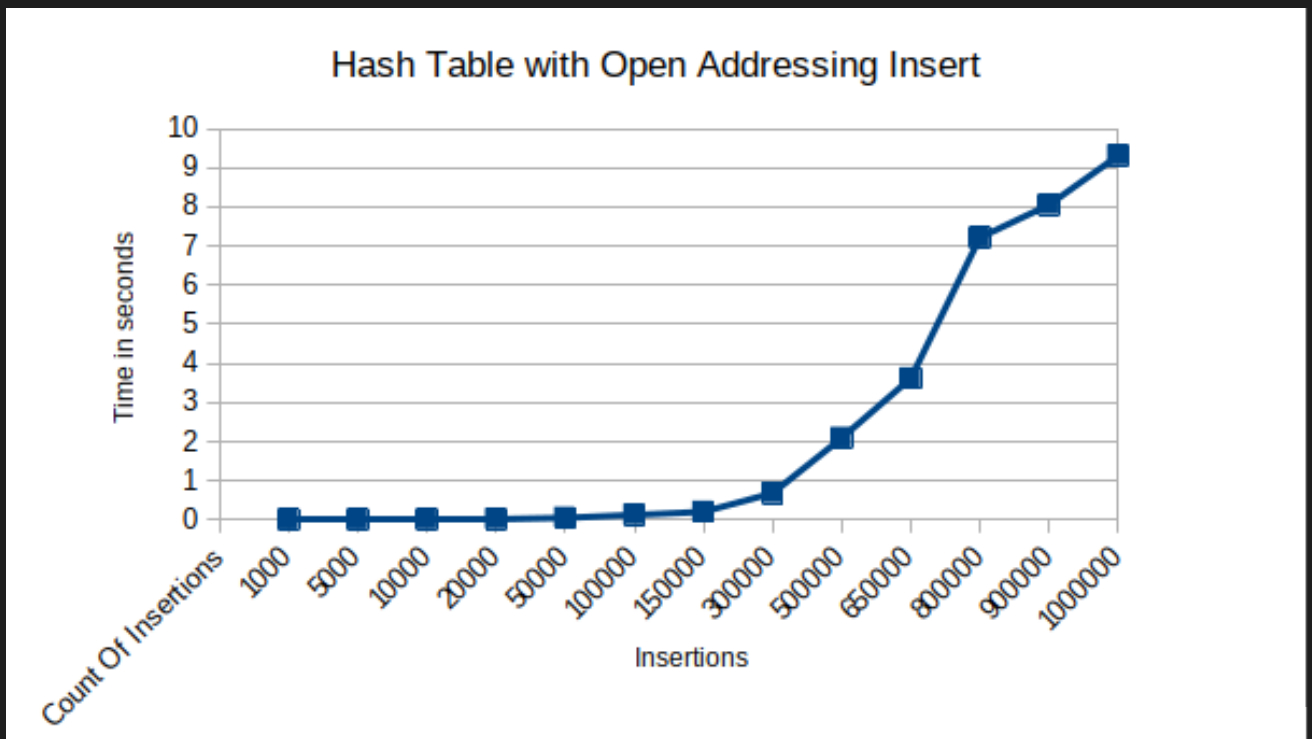
Performance and Memory Cost Comparison of Node Operations in a Data Structure

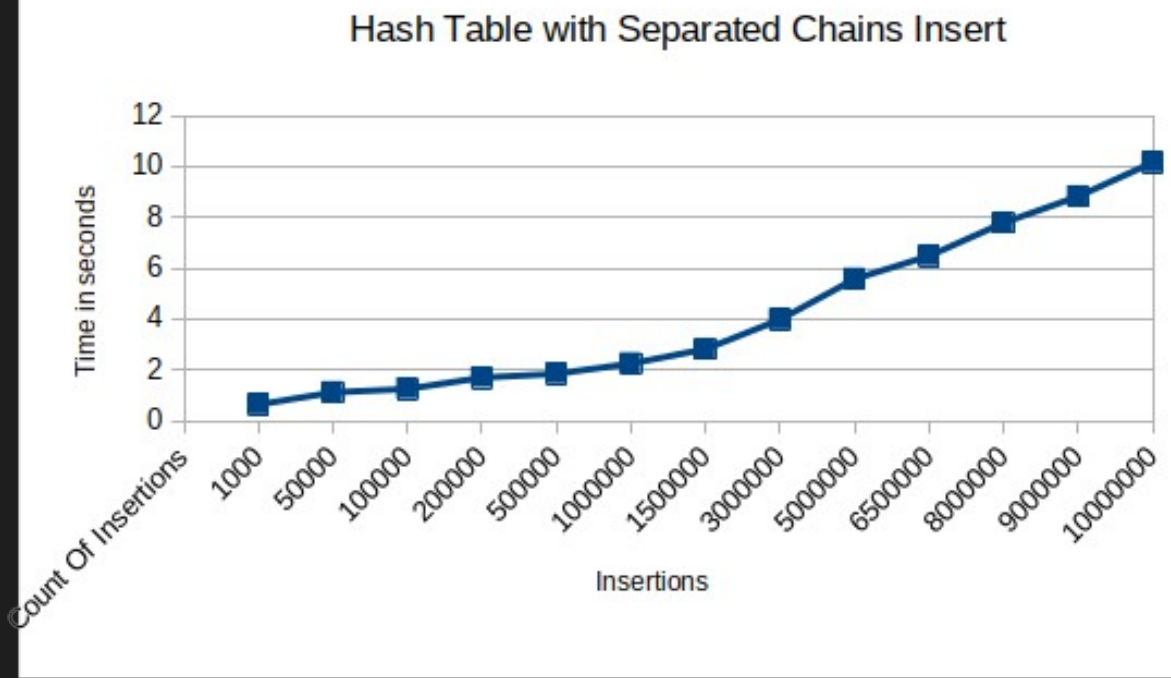
This study aims to compare the performance and CPU workload of individual node operations, including insertion, search, and deletion, in a data structure. The comparison was conducted on individual random subsets of data, which were common to all stages of verification. Additionally, the CPU cost of each function was compared. The study also investigated the effects of certain circumstances, such as the non-existence of a node during search and deletion operations, on the performance of the data structure.

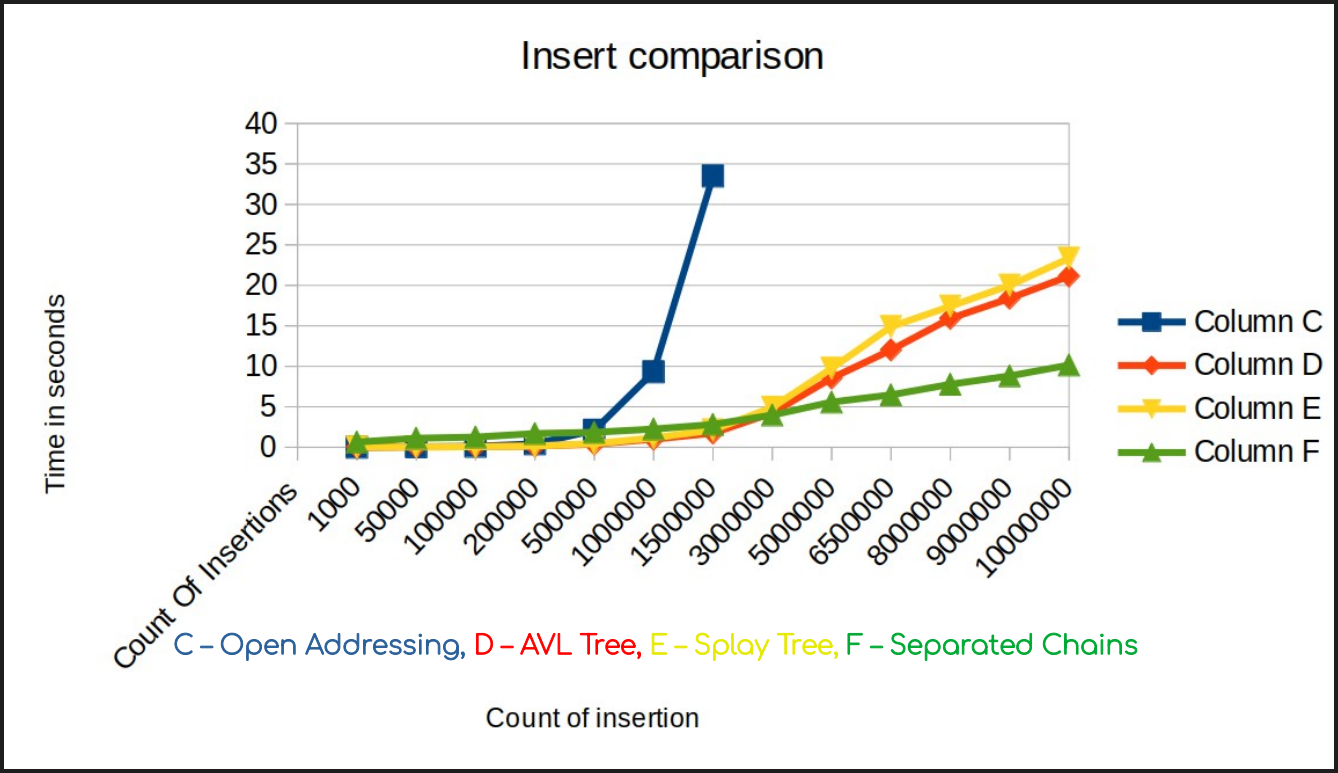
Investigation:

Insert




Hash Table with Open Addressing Insert
Time in seconds
0
1
2
3
4
5
6
7
8
9
10
Count Of Insertions
1000
5000
10000
20000
50000
100000
150000
300000
500000
650000
800000
900000
1000000
Insertions


Hash Table with Separated Chains Insert
Time in seconds
0
2
4
6
8
10
12
Count Of Insertions
1000
50000
100000
200000
500000
1000000
1500000
3000000
5000000
6500000
8000000
9000000
10000000
Insertions



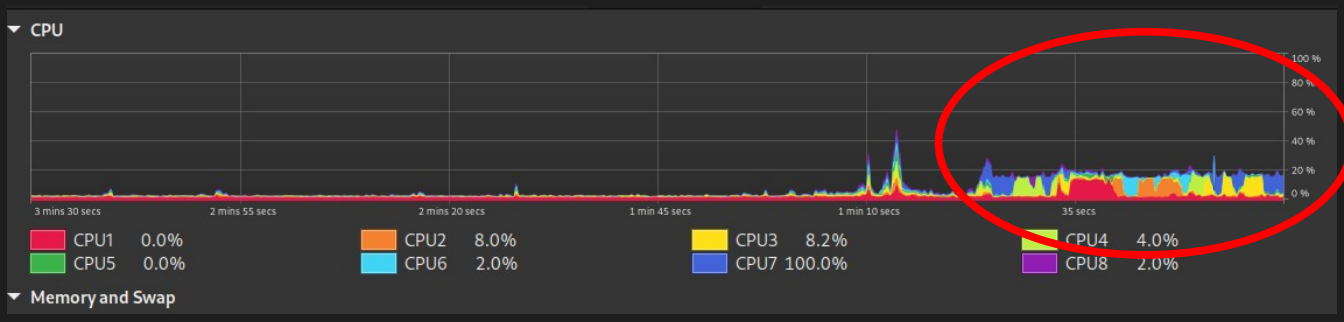
Conclusion:

The primary objective of the project was to assess the ability to construct code for functional data structures. Another key aspect was to evaluate the preservation of a confirmed logarithmic function when utilizing an alternative graph to represent the implementation of any of the suggested functions. The aforementioned images illustrate that the logarithmic function is indeed maintained, albeit not entirely. Our findings indicate that the Splay tree implementation and separated chains provided the most effective reproduction of insert function.

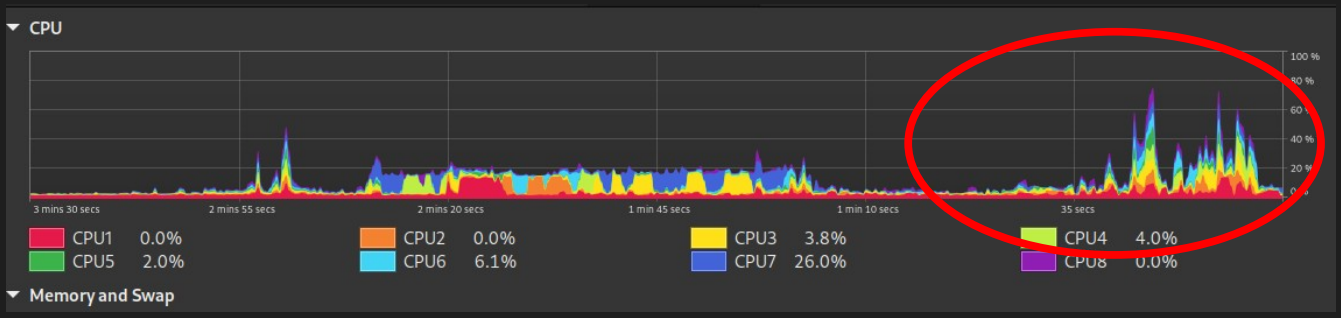
CPU usage:

For 10_000_000 inserted elements:

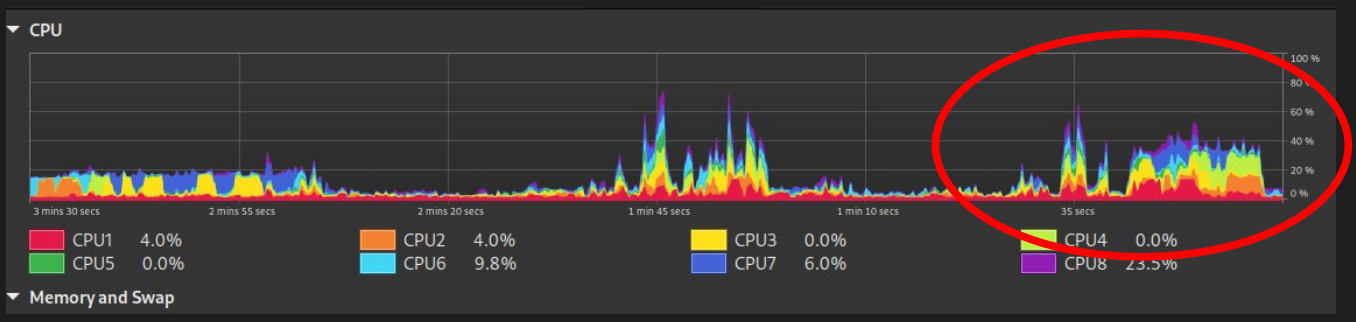
- HT with Open Addressing:



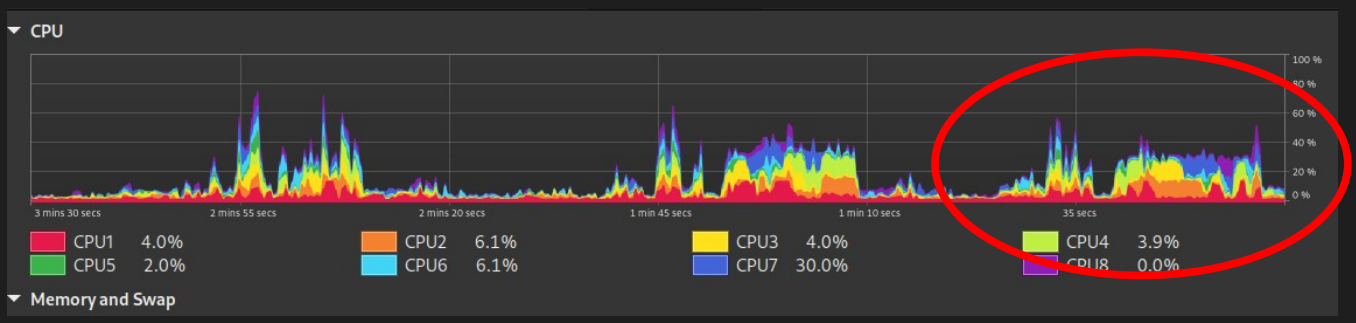
- HT with Separated Chains:



- AVL Tree:



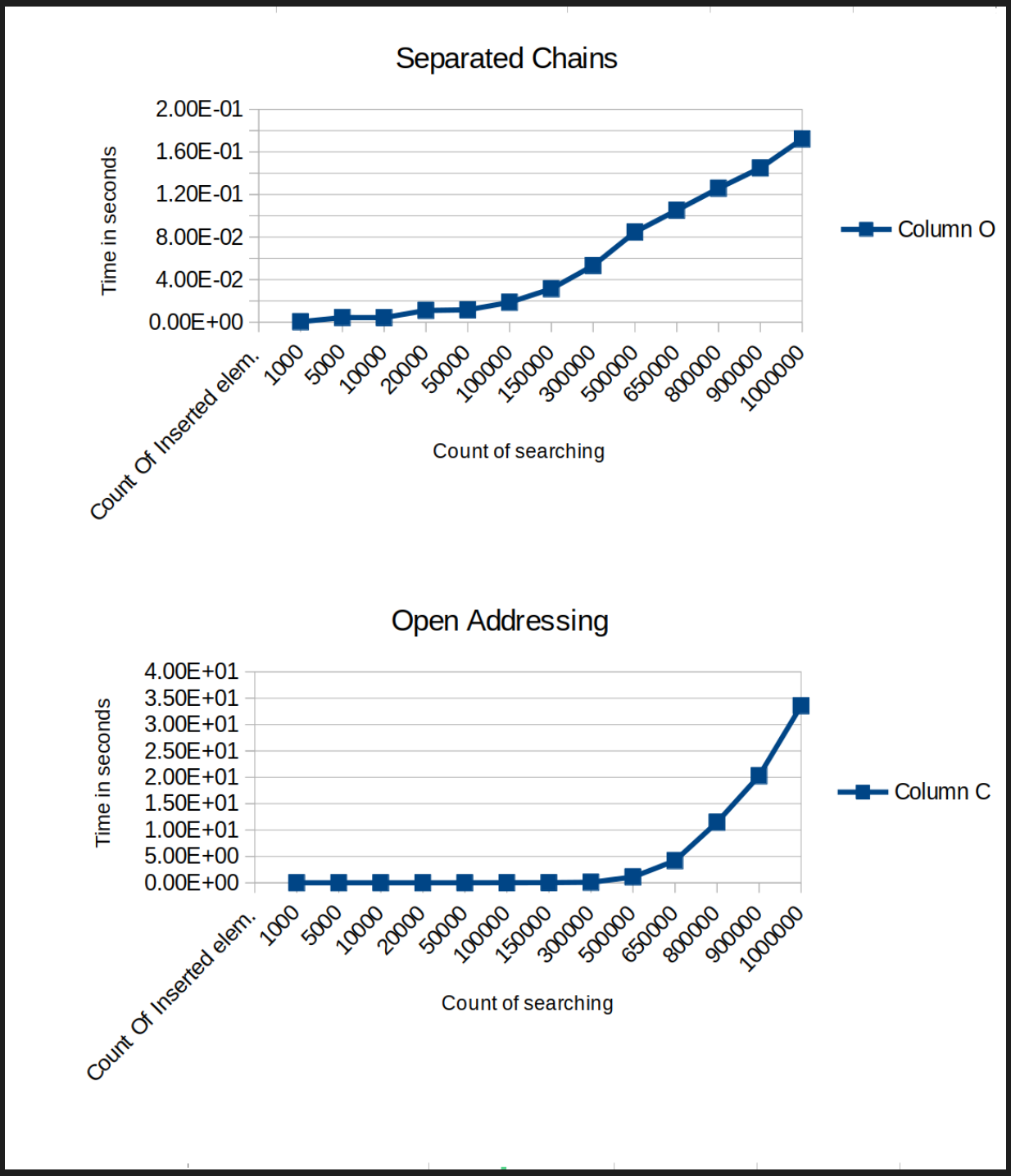
- Splay Tree:



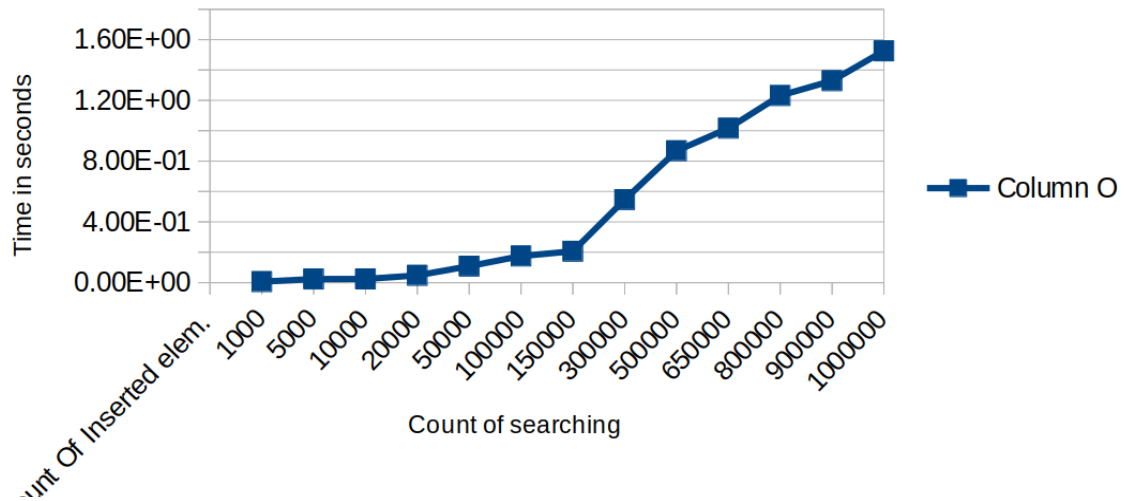
Conclusion:

Based on the observed data represented in the graphical outputs, it is evident that the employment of AVL tree structure results in the most advantageous outcome. This is due to the remarkable performance metrics exhibited by the processor, with a significant proportion of the indicators reaching optimal levels of 75%. However, in contrast, the implementation of Hash table with open addressing appears to be inefficient in terms of the insertion process, as it fails to utilize the processor's full capacity despite its inherent potential

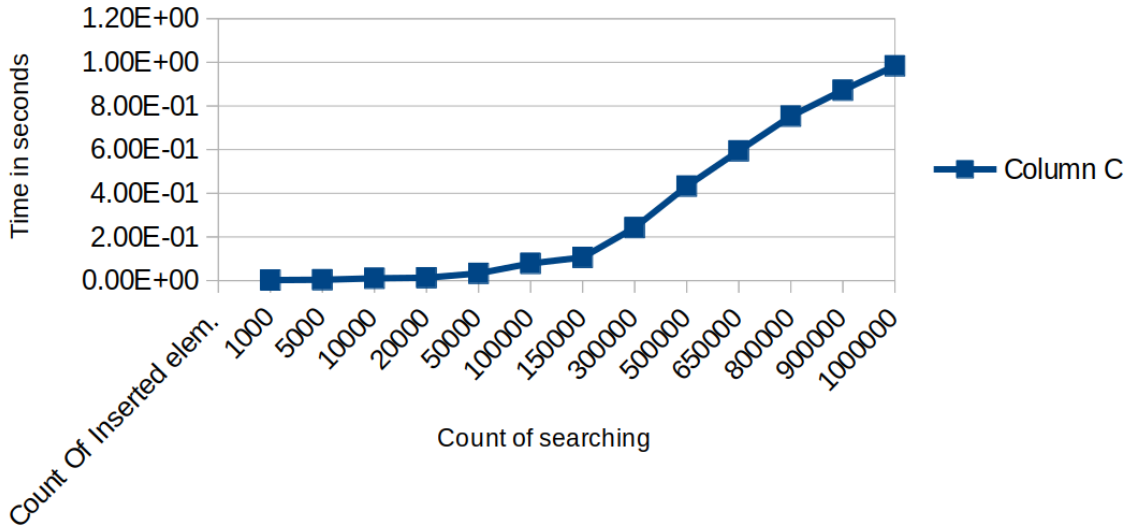
Search



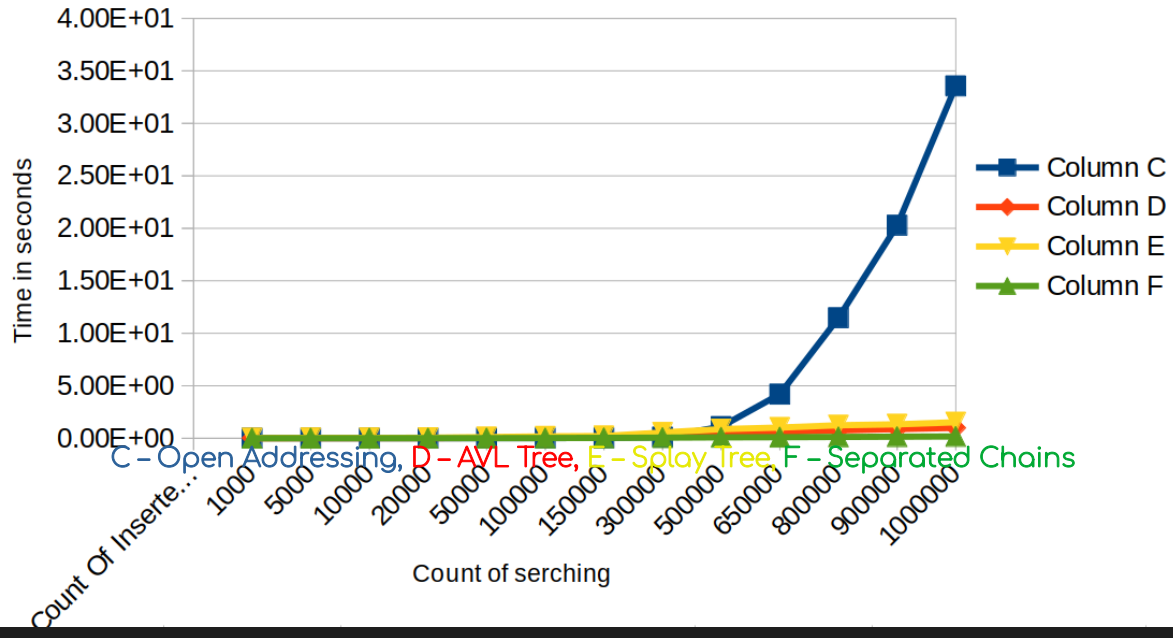
Splay Tree



AVL Tree



Search comparison

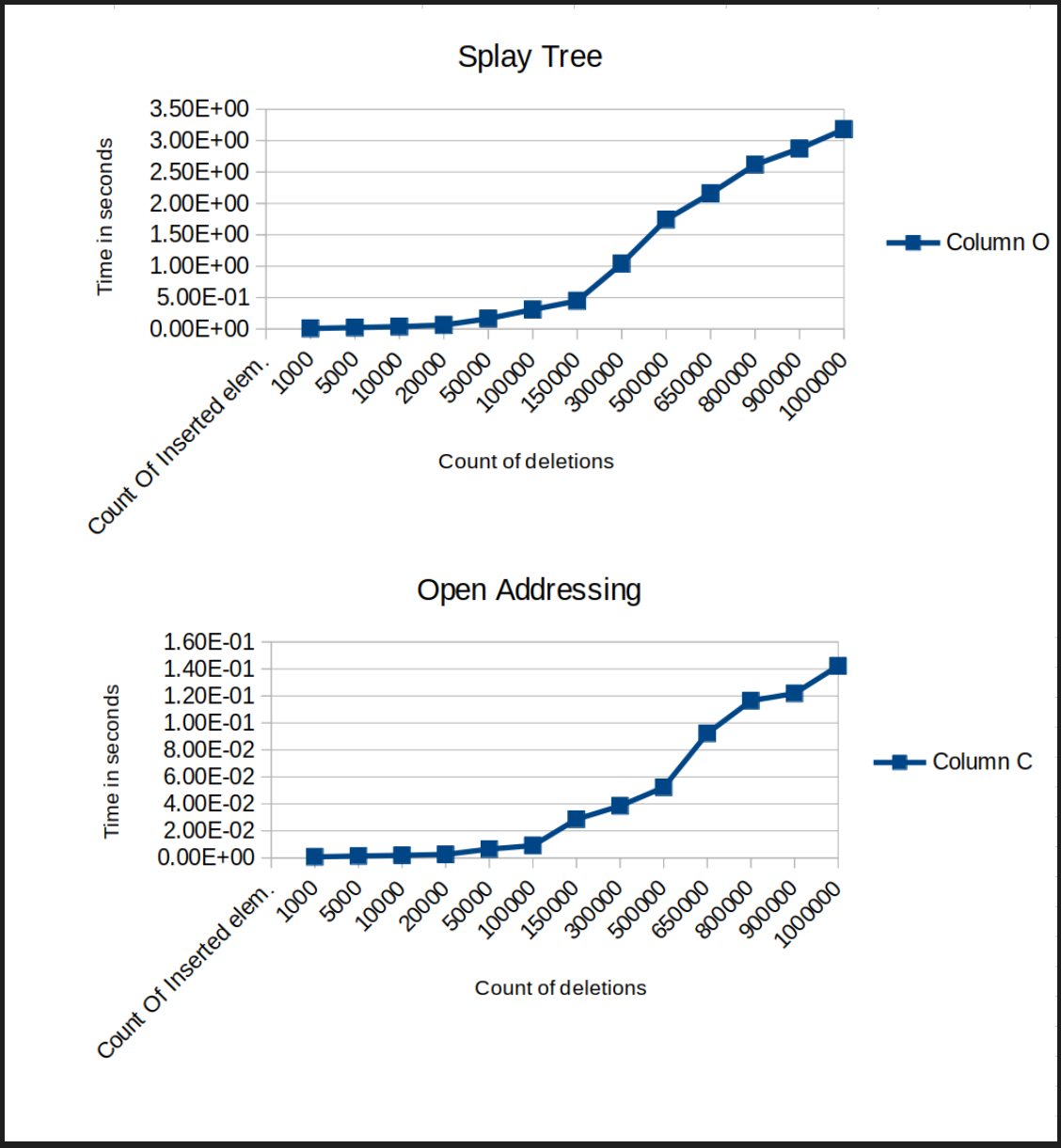


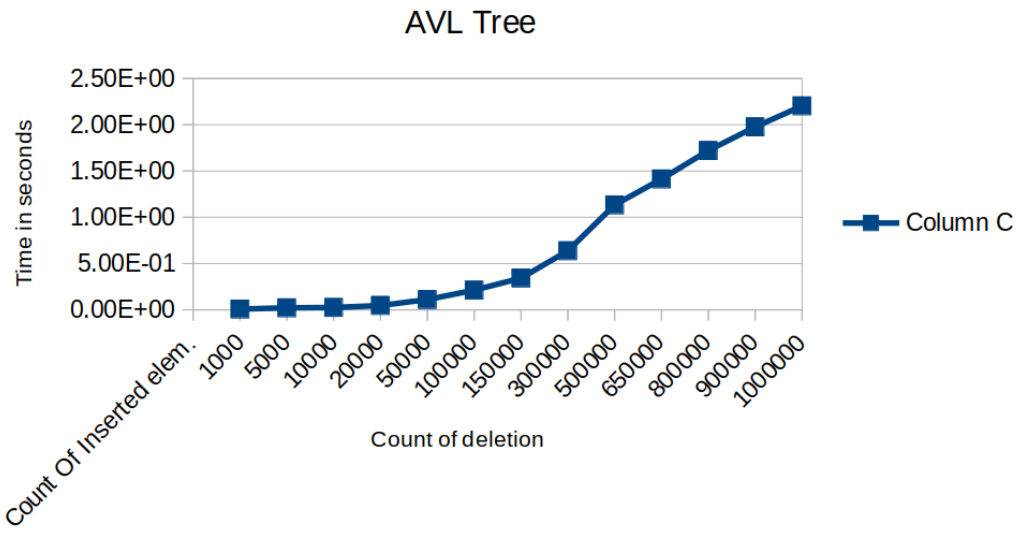
(On the picture above C – Open Addressing, D – AVL Tree, E – Splay Tree, F – Separated Chains)

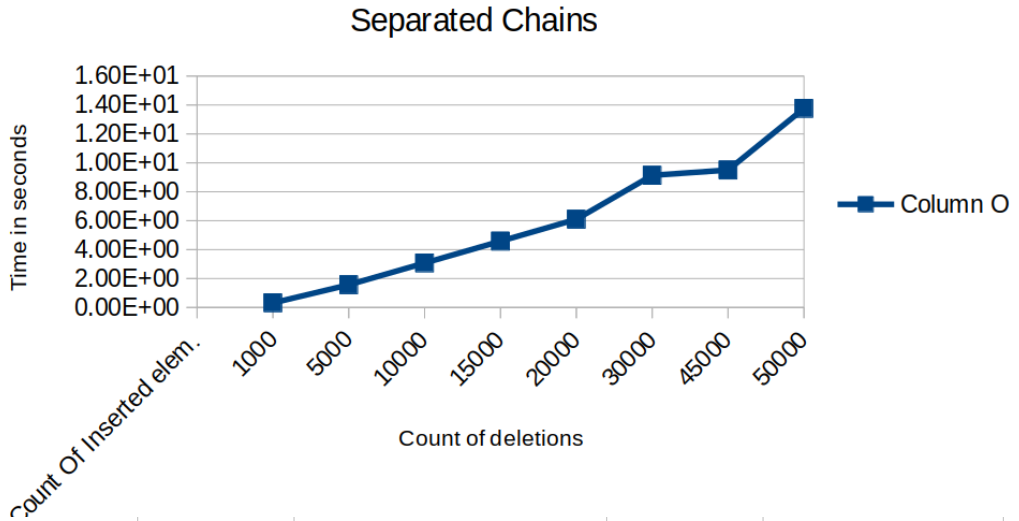
Conclusion:

The objective of this study was to examine the variation in element search behavior among different algorithms. Our results indicate that search speed is affected by table completeness, with open addressing showing notably slow search speeds. The algorithmic approach of examining all filled cells to locate a suitable key placement is a typical and reasonable technique. The fastest search performance was observed with the separated chains algorithm. Additionally, an experiment was conducted to investigate the speed of single searches across varying table occupancy levels. Our findings reveal that all implementations are well-suited for custom searches and significantly outperform linear traversal of node fields.

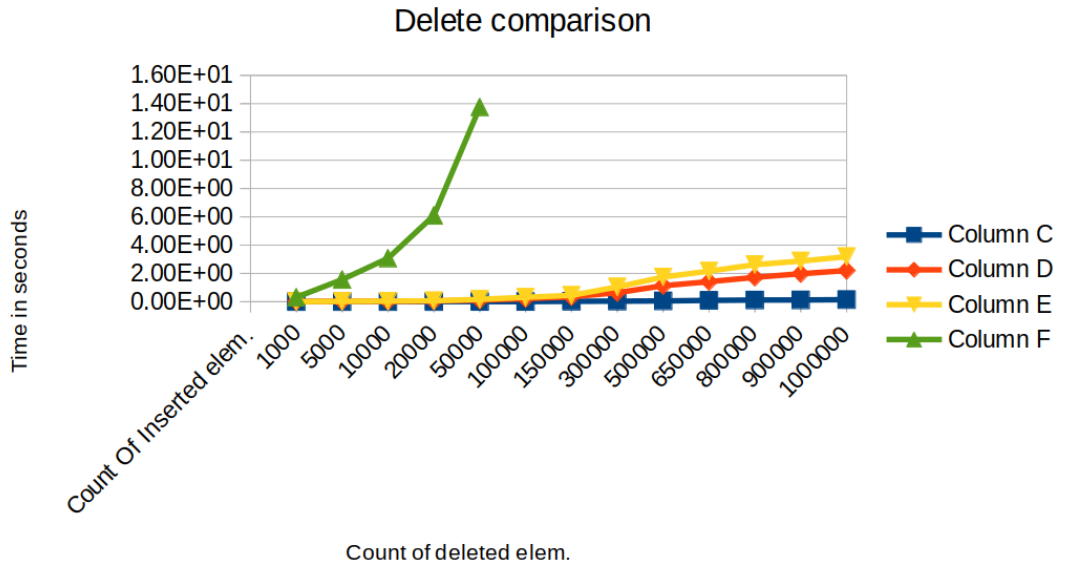
Delete







C – Open Addressing, D – AVL Tree, E – Splay Tree, F – Separated Chains



Conclusion:

Upon examining the graphs, it is evident that the separated chain implementation is the least optimal, as it displays significant sensitivity to the table's fullness. Conversely, other implementations, which involve element removal, have demonstrated higher efficacy. Notably, open addressing has been shown to be the most effective approach.

Main conclusion:

Data structures are essential tools in computer science and are used to store and organize data efficiently. The performance of data structures is an important consideration when selecting the appropriate structure for a given task. This study compares the performance and memory cost of individual node operations, namely insertion, search, and deletion, in a data structure. The study also investigates the effects of certain circumstances on the performance of the data structure.


hallelujah