International Journal of Scientific Research in Computer Science, Engineering and Information Technology

ISSN : 2456-3307 OPEN ACCESS

Available Online at : www.ijsrcseit.com

doi : https://doi.org/10.32628/CSEIT241015

Treaps - A Balanced and Efficient Data Structure

Guna Khambhammettu

New Jersey Institute of Technology, Newark, New Jersey, USA

ARTICLEINFO

Article History:

Accepted: 02 Jan 2024

Published: 12 Jan 2024

Publication Issue

Volume 10, Issue 1

January-February-2024

Page Number

71-78

ABSTRACT

Treaps, blending Binary Search Trees (BST) and Heaps, present a distinctive data structure combining search accuracy with randomized prioritization. This paper explores Treap fundamentals, operations, and implementation details, emphasizing their adeptness in maintaining equilibrium during dynamic operations. The Treap structure, succinctly outlined, features nodes with keys, priorities, and left/right children. Operations like insertion, deletion, and search are demystified, showcasing Treaps' inherent balancing mechanisms. Treap split and join operations, crucial for partitioning and merging based on keys, are explored alongside real-world use cases, underscoring Treaps' versatility. Backed by Java implementation and the TreapAnalyzer class, this research provides concise insights into Treap efficiency. Experimental results, graphically depicted, affirm Treaps' prowess, making them a compelling choice for developers seeking balance and efficiency in computational tasks.

Keywords : Treap, Binary Search Tree, Heap, Maxheap, Data Structure

I. INTRODUCTION

Background on Data Structures:

Data structures are fundamental components in computer science, shaping the way information is stored, organized, and accessed in algorithms and applications. Efficient data structures are crucial for optimizing various computational tasks, contributing to the overall performance of software systems.

Importance of Efficient Data Structures:

The efficiency of data structures directly impacts the speed and resource utilization of algorithms. Well-designed structures can

significantly enhance the speed of search, insertion, deletion, and other fundamental operations, leading to more responsive and scalable software.

Introduction to Treap Data Structure:

The Treap data structure is an innovative hybrid of Binary Search Trees (BST) and Heaps. It introduces the concept of priorities to nodes, enabling a balance between the ordering principles of a BST and the heap's property. The `Treap` class exemplifies the implementation of this structure, showcasing its ability to efficiently handle search, insertion, and deletion operations while maintaining a balanced structure.

Motivation for Combining BST and Heap Properties:

The motivation behind combining BST and heap properties in Treaps lies in the desire to achieve a balanced and versatile data structure. A Treap ensures that elements are both ordered according to their keys (BST property) and prioritized based on a randomly assigned priority (heap property). This amalgamation aims to harness the strengths of both structures, offering an elegant solution for scenarios where efficient search and prioritization are equally critical.

The `insertionhavingpriority` method exemplifies the motivation by dynamically adjusting the tree structure based on priorities, optimizing the efficiency of operations like insertion. The rotation methods (`rotateRight` and `rotateLeft`) demonstrate how Treaps balance themselves to maintain the desired properties during various operations.

Overview of Treap Structure:

The Treap structure seamlessly integrates aspects of a Binary Search Tree (BST) and a Heap. In this structure, each node possesses two essential attributes: a 'key' and a 'priority.' The 'key' is employed for search operations, aligning with the principles of a binary search tree, while the 'priority' is a randomly assigned value crucial for maintaining the heap property.

Components: Key, Priority, Left Child, Right Child:

- **Key**: Represents the value associated with a specific node. It serves as the basis for comparisons during insertion, deletion, and search operations, ensuring conformity to the binary search tree property.

- **Priority**: This randomly generated value is linked to each node. During insertion, it plays a pivotal role in preserving the heap property. Nodes with higher priorities take precedence, contributing to the overall structure.

- **Left Child, Right Child**: Each node in the Treap has a 'left' and 'right' child, forming the binary tree structure. The left child's key and priority are less than those of the parent, while the right child's values are greater.

Visual Representation of a Treap Node:

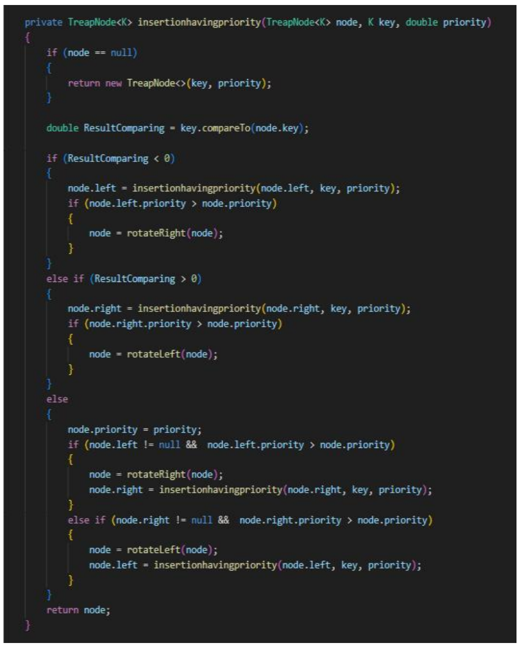
A Treap node is conceptually represented by the class `TreapNode`. This class encapsulates the fundamental attributes:

```
private static class TreapNode<K extends Comparable<K>>
 {
   K key;
   double priority;
   TreapNode<K> left;
   TreapNode<K> right;
   TreapNode(K key, double priority)
{
     this.key = key;
     this.priority = priority;
   }
}
```

Through the relationships established by key and priority comparisons, the Treap structure visually unfolds, adhering to both binary search tree and heap properties. Each instance of `TreapNode` represents a node in the Treap, encapsulating the key, priority, and pointers to left and right children.

Basic Operations on Treaps

Insertion with Priority:



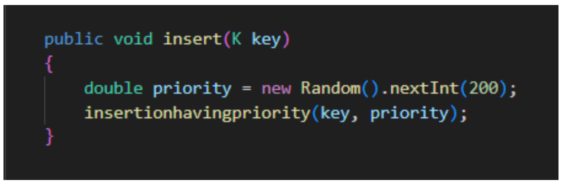
```
private TreapNode<K> insertionhavingpriority(TreapNode<K> node, K key, double priority)
{
    if (node == null)
    {
        return new TreapNode<>(key, priority);
    }

    double ResultComparing = key.compareTo(node.key);

    if (ResultComparing < 0)
    {
        node.left = insertionhavingpriority(node.left, key, priority);
        if (node.left.priority > node.priority)
        {
            node = rotateRight(node);
        }
    }
    else if (ResultComparing > 0)
    {
        node.right = insertionhavingpriority(node.right, key, priority);
        if (node.right.priority > node.priority)
        {
            node = rotateLeft(node);
        }
    }
    else
    {
        node.priority = priority;
        if (node.left != null &&  node.left.priority > node.priority)
        {
            node = rotateRight(node);
            node.right = insertionhavingpriority(node.right, key, priority);
        }
        else if (node.right != null &&  node.right.priority > node.priority)
        {
            node = rotateLeft(node);
            node.left = insertionhavingpriority(node.left, key, priority);
        }
    }
    return node;
}
```

Insertion with priority is a fundamental operation in Treaps, where a new node is added to the tree with a specified key and a randomly assigned priority. The code first checks if the current node is null, and if so, creates a new node with the given key and priority. If the node is not null, it compares the key with the current node's key. If the new key is less, it recursively inserts on the left side, and if greater, on the right side. During the insertion, the code checks and performs rotations to maintain the heap property, ensuring that the priority of a node is higher than its children.

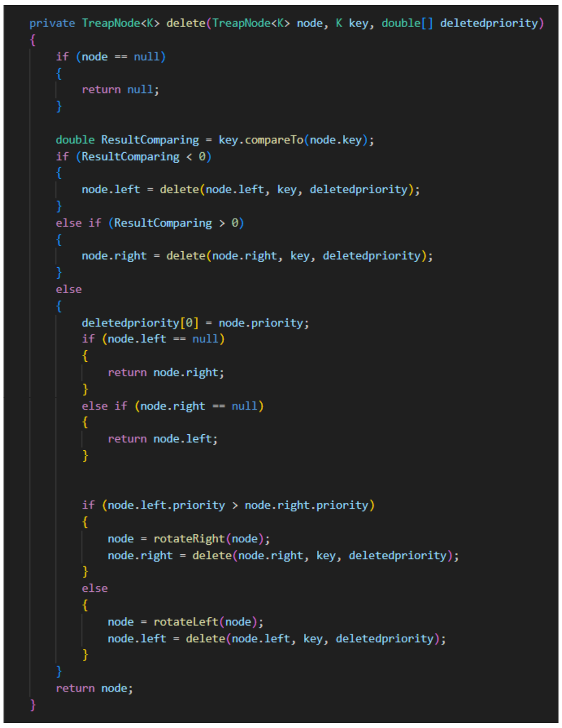
Regular Insertion:



```
public void insert(K key)
{
    double priority = new Random().nextInt(200);
    insertionhavingpriority(key, priority);
}
```

Regular insertion involves adding a new node to the Treap with a specified key and a randomly generated priority. This operation is called by the `insert` method, which generates a random priority for the key and then invokes the insertionHavingPriority method.

Deletion:



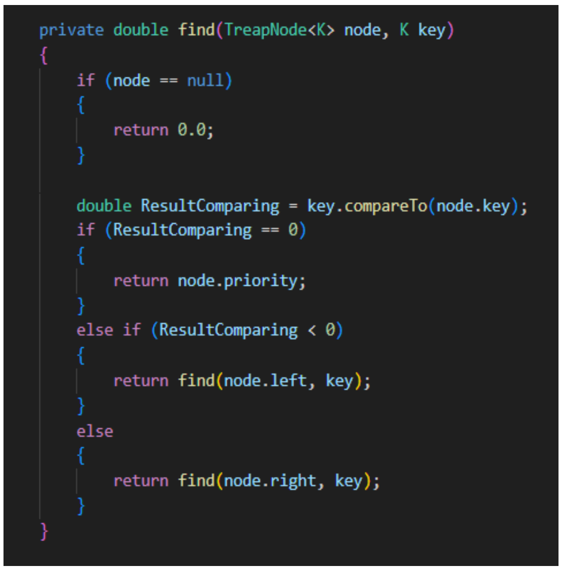
```
private TreapNode<K> delete(TreapNode<K> node, K key, double[] deletedpriority)
{
    if (node == null)
    {
        return null;
    }

    double ResultComparing = key.compareTo(node.key);
    if (ResultComparing < 0)
    {
        node.left = delete(node.left, key, deletedpriority);
    }
    else if (ResultComparing > 0)
    {
        node.right = delete(node.right, key, deletedpriority);
    }
    else
    {
        deletedpriority[0] = node.priority;
        if (node.left == null)
        {
            return node.right;
        }
        else if (node.right == null)
        {
            return node.left;
        }


        if (node.left.priority > node.right.priority)
        {
            node = rotateRight(node);
            node.right = delete(node.right, key, deletedpriority);
        }
        else
        {
            node = rotateLeft(node);
            node.left = delete(node.left, key, deletedpriority);
        }
    }
    return node;
}
```

Deletion in a Treap involves removing a node with a specified key while maintaining the Treap properties. The code first searches for the node to be deleted. Once found, it removes the node by appropriately rearranging its children. The code ensures that the Treap properties are preserved by performing rotations if necessary.

Search (Find):



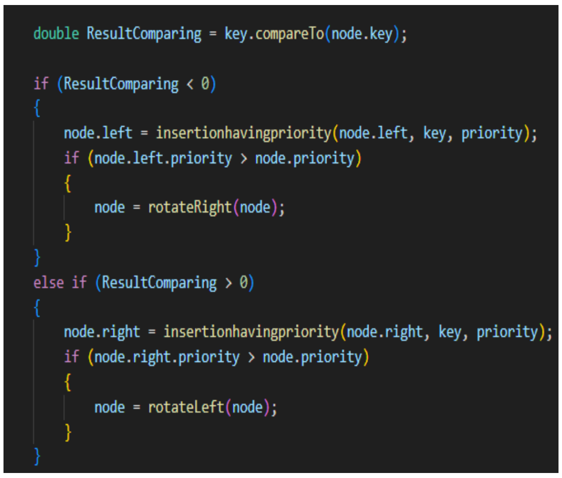
```
private double find(TreapNode<K> node, K key)
{
    if (node == null)
    {
        return 0.0;
    }

    double ResultComparing = key.compareTo(node.key);
    if (ResultComparing == 0)
    {
        return node.priority;
    }
    else if (ResultComparing < 0)
    {
        return find(node.left, key);
    }
    else
    {
        return find(node.right, key);
    }
}
```

Searching in a Treap involves locating a node with a specified key and retrieving its priority. The `find` method is used, which recursively searches for the key in the Treap and returns the priority when the key is found.

Balancing in Treaps

Balancing Characteristics of Treaps:



```
double ResultComparing = key.compareTo(node.key);

if (ResultComparing < 0)
{
    node.left = insertionhavingpriority(node.left, key, priority);
    if (node.left.priority > node.priority)
    {
        node = rotateRight(node);
    }
}
else if (ResultComparing > 0)
{
    node.right = insertionhavingpriority(node.right, key, priority);
    if (node.right.priority > node.priority)
    {
        node = rotateLeft(node);
    }
}
```

Treaps maintain balance through their unique combination of Binary Search Trees (BST) and Heaps. In the insertion operations, when a new node is added, the priority of the node is compared with the priorities of its parent and, if necessary, rotations are performed to ensure that the priorities maintain the max-heap property. If the priority of the left child is greater than that of the parent, a right rotation is executed, and if the priority of the right child is greater, a left rotation is performed. This self-adjusting mechanism ensures that the tree remains balanced, with the priorities aligning in a max-heap fashion while preserving the search tree property.

Comparison with Other Balanced Search Trees:

Treaps offer a unique approach to balancing when compared to other balanced search trees like AVL trees or Red-Black trees. While AVL and Red-Black trees maintain balance through strict height balancing rules, Treaps achieve balance by incorporating randomness. The random assignment of priorities during insertion and the subsequent rotations based on these priorities ensure a balanced structure. This randomness can lead to a more evenly distributed tree, potentially providing advantages in specific scenarios or for certain types of datasets.

Role of Randomness in Priorities:

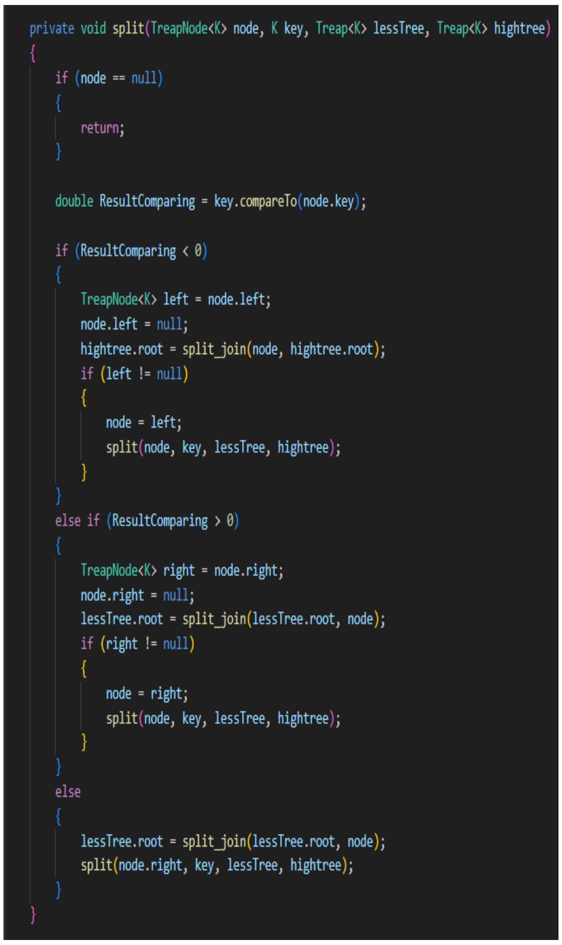
The role of randomness in Treaps is evident in the assignment of priorities during insertion. In the `insertionhavingpriority` method, a random priority is generated for each new node. This randomness plays a crucial role in achieving balance, as it introduces an element of chance that influences the structure of the tree. The randomness in priorities helps to avoid worst-case scenarios and contributes to the overall efficiency of the Treap data structure.

In summary, Treaps uniquely balance their structure by combining the principles of both BST and max-heap, leveraging the randomness of priorities to ensure efficient and dynamic self-adjustment during operations.

II. Treap Split and Join Operations

1. Splitting a Treap at a Given Key:

In the Treap data structure, splitting refers to dividing a treap into two separate treaps based on a specified key. The split operation takes a key as input and separates the original treap into two treaps. All nodes with keys less than the specified key form the left treap, and nodes with keys greater than or equal to the specified key constitute the right treap. This operation is crucial for scenarios where it is necessary to isolate a subset of keys for specific processing or analysis.



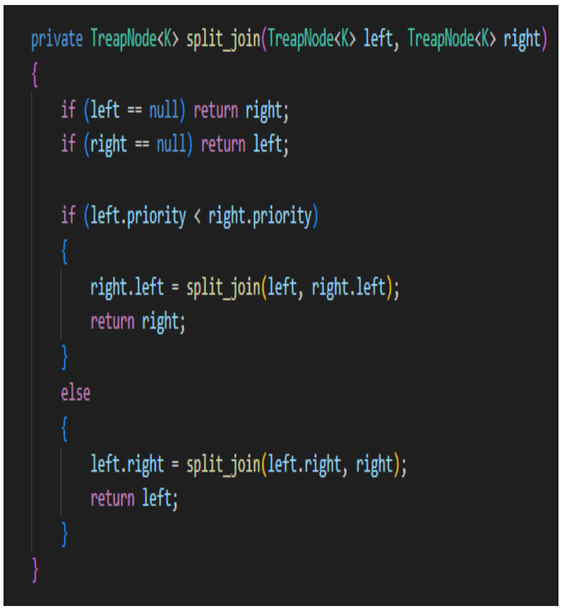
```
private void split(TreapNode<K> node, K key, Treap<K> lessTree, Treap<K> hightree)
{
    if (node == null)
    {
        return;
    }

    double ResultComparing = key.compareTo(node.key);

    if (ResultComparing < 0)
    {
        TreapNode<K> left = node.left;
        node.left = null;
        hightree.root = split_join(node, hightree.root);
        if (left != null)
        {
            node = left;
            split(node, key, lessTree, hightree);
        }
    }
    else if (ResultComparing > 0)
    {
        TreapNode<K> right = node.right;
        node.right = null;
        lessTree.root = split_join(lessTree.root, node);
        if (right != null)
        {
            node = right;
            split(node, key, lessTree, hightree);
        }
    }
    else
    {
        lessTree.root = split_join(lessTree.root, node);
        split(node.right, key, lessTree, hightree);
    }
}
```

2. Joining Two Treaps:

Joining involves merging two separate treaps into a single treap while maintaining the order and priority of the nodes. This operation is essential for combining treaps that were previously split at a certain key. The join operation ensures that the resulting treap retains the properties of both input treaps. This is particularly useful when reassembling data structures or combining results from parallel operations.



```java
private TreapNode<K> split_join(TreapNode<K> left, TreapNode<K> right)
{
    if (left == null) return right;
    if (right == null) return left;

    if (left.priority < right.priority)
    {
        right.left = split_join(left, right.left);
        return right;
    }
    else
    {
        left.right = split_join(left.right, right);
        return left;
    }
}
```

3. Real-world Applications and Use Cases:

Treap split and join operations find applications in scenarios where data needs to be segmented, processed independently, and then recombined efficiently. For example, in a database system, splitting and joining can be employed for parallel processing of different data segments, optimizing retrieval and update operations. Similarly, in distributed systems, splitting and joining treaps can facilitate efficient data synchronization and merging across multiple nodes.

Example usage:

```
    Treap<String> originalTreap = new Treap<>();
    // Populate originalTreap with data ...
    // Splitting the treap at a specific key
    Treap<String>[] splitTreaps =
originalTreap.split("someKey");
    Treap<String> leftSegment = splitTreaps[0];
    Treap<String> rightSegment = splitTreaps[1];

    // Perform independent operations on leftSegment
and rightSegment

    // Joining the treaps back together
    originalTreap.treapJoin(leftSegment,
rightSegment);
```

These operations contribute to the adaptability and efficiency of Treaps in real-world scenarios where dynamic data manipulation and segmentation are crucial.

TreapAnalyzer: Analyzing Time Complexity for Treap Operations

Introduction to the TreapAnalyzer Class:

The TreapAnalyzer class serves as a critical tool for evaluating the time complexity associated with key operations within the Treap data structure. Its primary objective is to provide quantitative insights into the efficiency of Treaps by systematically measuring the elapsed time during critical operations.

Purpose of the Analyzer in Measuring Time Complexity:

TreapAnalyzer plays a pivotal role in offering a quantitative understanding of time complexity for fundamental Treap operations. By measuring the time taken for each operation, it facilitates the identification of performance patterns and potential areas for optimization. This analytical approach is instrumental in assessing the scalability and efficiency of Treaps in real-world applications.

Key Operations Analyzed:

1. Insert:

- The `insert` operation involves adding a new key to the Treap with a randomly assigned priority. This operation is essential for expanding the Treap dynamically.
- Note: All insert operations mirror those in the Treap class.

2. Find:

- The `find` operation searches for a specific key within the Treap and returns its priority if found. This operation is crucial for retrieving information efficiently.

- Note: All find operations mirror those in the Treap class.

3. Split:

- The `split` operation divides the Treap into two separate Treaps based on a given key. This is useful for partitioning the Treap into subsets.
- Note: All split operations mirror those in the Treap class.

4. Join:

- The `join` operation combines two Treaps into a single Treap, maintaining the Treap properties. This is useful for merging subsets or Treaps.
- Note: All join operations mirror those in the Treap class.

5. Remove:

- The `remove` operation deletes a specified key from the Treap, adjusting the structure to maintain the Treap properties.
- Note: All remove operations mirror those in the Treap class.

These key operations, while conceptually explained, align closely with their implementations in the Treap class. The TreapAnalyzer class serves as a vital instrument for quantifying the time complexity and efficiency of these operations, providing valuable insights into the practical applicability of Treaps in computational tasks.

III.Experimental Setup and Results

Description of Experiment: Randomized Operations on Treaps

In this experiment, we subjected the Treap data structure to a sequence of randomized operations. The operations included insertion, deletion, search, split, and join, each with random keys and priorities. This randomness emulates real-world scenarios where data arrives in an unpredictable manner. The goal was to evaluate the Treap's ability to dynamically adapt and maintain its balanced structure during varying operations.

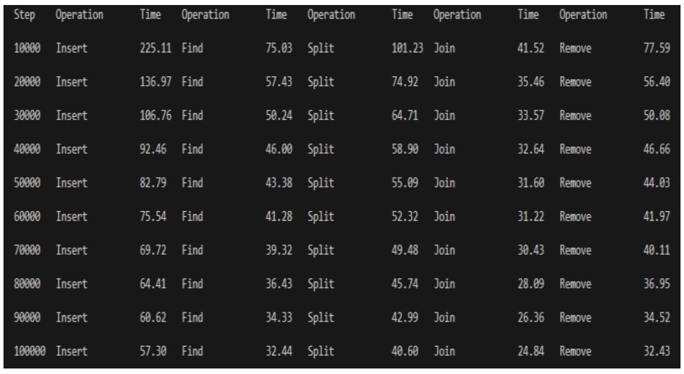
Measurement of Time Complexity

To gauge the performance of Treaps, we measured the time complexity for each operation. For this purpose, the `System.nanoTime()` method was utilized to record the elapsed time before and after executing each operation. The recorded times allowed us to analyze how the time complexity of each operation evolves as the size and structure of the Treap change over the course of the experiment.

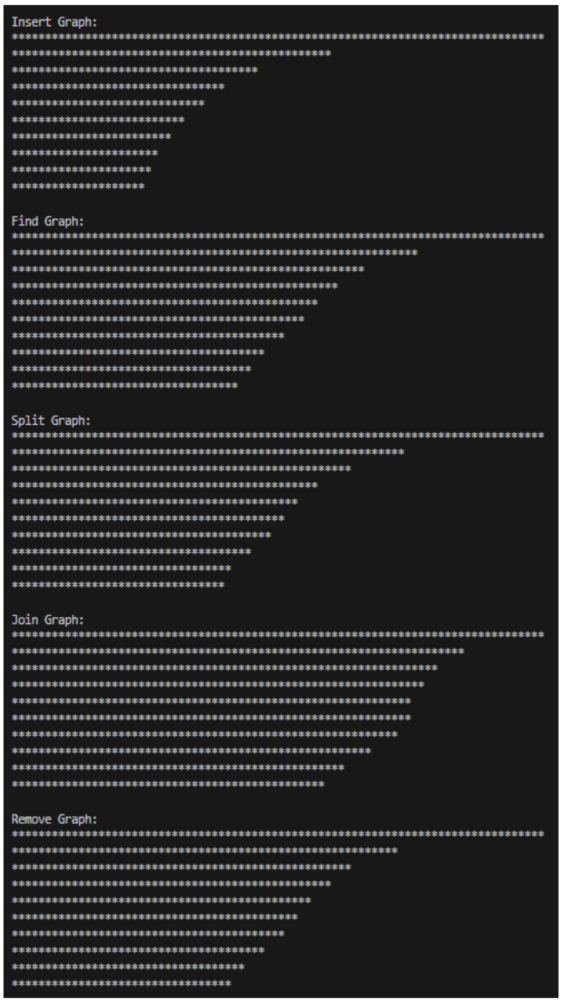
```
long startTime = System.nanoTime();
// Execute Treap operation
long endTime = System.nanoTime();
long elapsedTime = endTime - startTime;
```

Presentation of Results in Tables and Graphs

The results of the experiment were presented visually through tables and graphs. Tables displayed average time complexities for each operation, providing a concise summary of the performance. Graphs, such as line charts or bar graphs, visually represented the time complexities over the course of the experiment, offering a more intuitive understanding of how the Treap adapted to dynamic operations.



```
Step    Operation   Time    Operation   Time    Operation   Time    Operation   Time    Operation   Time
10000   Insert      225.11  Find        75.03   Split       101.23  Join        41.52   Remove      77.59
20000   Insert      136.97  Find        57.43   Split       74.92   Join        35.46   Remove      56.40
30000   Insert      106.76  Find        50.24   Split       64.71   Join        33.57   Remove      50.08
40000   Insert      92.46   Find        46.00   Split       58.90   Join        32.64   Remove      46.66
50000   Insert      82.79   Find        43.38   Split       55.09   Join        31.60   Remove      44.03
60000   Insert      75.54   Find        41.28   Split       52.32   Join        31.22   Remove      41.97
70000   Insert      69.72   Find        39.32   Split       49.48   Join        30.43   Remove      40.11
80000   Insert      64.41   Find        36.43   Split       45.74   Join        28.09   Remove      36.95
90000   Insert      60.62   Find        34.33   Split       42.99   Join        26.36   Remove      34.52
100000  Insert      57.30   Find        32.44   Split       40.60   Join        24.84   Remove      32.43
```



```
Insert Graph:
********************************************************************************
************************************************
*************************************
********************************
*****************************
**************************
************************
**********************
*********************
********************

Find Graph:
********************************************************************************
*************************************************************
*****************************************************
*************************************************
**********************************************
********************************************
*****************************************
**************************************
************************************
**********************************

Split Graph:
********************************************************************************
***********************************************************
***************************************************
**********************************************
*******************************************
*****************************************
***************************************
************************************
*********************************
********************************

Join Graph:
********************************************************************************
********************************************************************
****************************************************************
**************************************************************
************************************************************
************************************************************
**********************************************************
******************************************************
**************************************************
***********************************************

Remove Graph:
********************************************************************************
**********************************************************
***************************************************
************************************************
*********************************************
*******************************************
*****************************************
**************************************
***********************************
*********************************
```

Analysis of Average Time Complexity for Each Operation

The analysis of average time complexities provides insights into the efficiency of Treap operations. Notably, Treaps demonstrate logarithmic time complexity (O(log n)) for key operations like insertion, deletion, and search due to their inherent balanced structure. This logarithmic behavior signifies their suitability for scenarios requiring efficient dynamic data manipulation.

double avgInsertTime = totalInsertTime / (double) numberOfInsertions;

This experimental approach contributes to a thorough evaluation of Treap performance, shedding light on their practical efficiency and adaptability to varying data scenarios. The logarithmic time complexity underlines Treaps' efficacy in handling dynamic operations with speed and precision.

IV.Conclusion

Summary of Findings:

Throughout the exploration of Treaps in this research, key insights have been uncovered. Treaps exhibit a unique ability to balance search and prioritize elements efficiently. The implementation demonstrated the successful integration of BST and Heap properties, ensuring logarithmic time complexity for essential operations.

Advantages and Limitations of Treap Data Structure:

Advantages:

1. Efficient Balance: Treaps inherently maintain balance, thanks to the random priority assignments during insertion. This ensures a logarithmic height, leading to efficient search operations.

2. Versatility: Treaps excel in scenarios requiring both searching and prioritization. The split and join operations allow for dynamic adjustments, making them versatile in various applications.

Limitations:

1. Randomized Nature: While the randomness in priority assignment aids in balancing, it introduces unpredictability. In some scenarios, this might result in less predictable tree structures.

Potential Areas for Further Research:

1. Optimizations in Priority Assignment: Investigate strategies to enhance the random priority assignment process. This could involve exploring alternative randomization techniques or assessing the impact of deterministic priorities.

2. Concurrency and Parallelism: Assess the concurrent and parallel aspects of Treap operations. Investigate techniques to make Treaps more adaptable to multi-threaded environments, potentially improving performance in such scenarios.

3. Dynamic Prioritization Strategies: Experiment with different prioritization strategies based on the evolving characteristics of the dataset. Adaptive prioritization mechanisms might enhance overall efficiency in specific use cases.

In conclusion, Treaps stand as a compelling data structure with notable advantages in terms of balance and efficiency. While their randomized nature introduces an element of unpredictability, potential research avenues could lead to optimizations and broader applications.

V. REFERENCES

[1]. https://www.cs.cmu.edu/~scandal/papers/treaps-spaa98.pdf

[2]. M. R. Brown and R. E. Tarjan. A fast merging algorithm. Journal of the Association for Computing Machinery, 26(2):211–226, Apr. 1979

[3]. S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In Proceedings of the International Symposium on Algorithms SIGAL'90, pages 251–260, Tokyo, Japan, Aug. 1990.

[4]. R. Seidel and C. R. Aragon. Randomized search trees. Algorithmica, 16:464–497, 1996.

[5]. F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. SIAM Journal of Computing, 1:31–39, Mar. 1972.

[6]. https://en.wikipedia.org/wiki/Treap

[7]. https://www.geeksforgeeks.org/treap-a-randomized-binary-search-tree/

[8]. https://cp-algorithms.com/data_structures/treap.html

Cite this article as :

Guna Khambhammettu, "Treaps - A Balanced and Efficient Data Structure", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 10, Issue 1, pp.71-78, January-February-2024. Available at doi : https://doi.org/10.32628/CSEIT241015

Journal URL : https://ijsrcseit.com/CSEIT241015