Trees

Trees in Data Structure

Nitin Prakash
Master's in computer science
Northern Arizona University
Flagstaff Arizona Coconino
np679@nau.edu

Sai Sunandh Ramayanam
Master's in computer science
Northern Arizona University
Flagstaff Arizona Coconino
sr2984@nau.edu

Katta Lakshmi Prasanna
Master's in computer science
Northern Arizona University
Flagstaff Arizona Coconino
klp468@nau.edu

ABSTRACT

Trees are fundamental data structures in computer science, essential for various applications ranging from databases to network systems. This paper presents a comprehensive overview of tree data structures, beginning with basic definitions and terminology such as nodes, edges, root, leaves, and depth. We delve into the distinction between balanced and unbalanced trees, illustrating their significance in ensuring efficient data operations. The real-world applicability of tree structures is highlighted through practical examples.

A significant portion of this study is dedicated to tree traversal techniques—specifically in-order, pre-order, and post-order traversals—comparing recursive and iterative approaches. We explore expression trees and their utility in evaluating expressions, offering insights into their operational mechanisms.

Advancing into complex tree structures, we discuss binary heaps, detailing their definitions, essential operations like insertion and deletion, and their role in sorting algorithms and various applications. The paper also explores self-balancing trees, with a focus on Red-Black Trees and AVL Trees, examining their balancing criteria and rotation mechanisms. B-Trees are presented in the context of their applications in databases and file systems, emphasizing their unique structural features.

Lastly, we delve into Huffman Trees, discussing their critical role in data compression techniques. This comprehensive study aims to elucidate the intricacies of tree data structures, providing a foundational understanding that is crucial for both academic research and practical implementations in computer science.

1. INTRODUCTION:

In the fields of mathematics and computer science, a tree is a hierarchical data structure. It is made up of nodes with edges connecting them. Apart from the top node, known as the root, which has no parents, every node in a tree can have zero or more children. Data is frequently stored in a tree's nodes, and the relationships between those nodes are represented by their connections. For organizing and illustrating hierarchical systems, trees are very helpful.

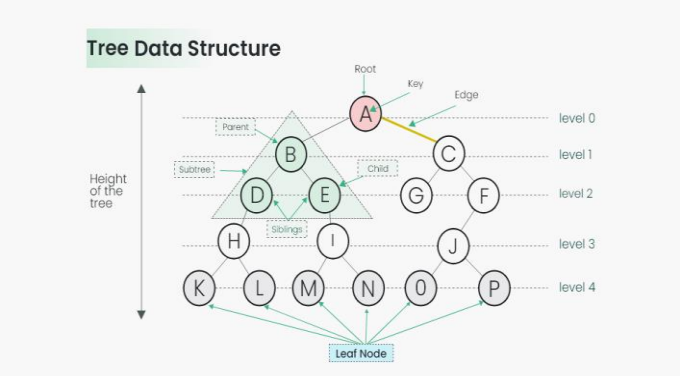

Figure 1.1 Tree Data Structure

1. **Root**: In a tree, the root is the highest node. It acts as the gateway to all of the tree's components.

2. **Node**: A node is any one of a tree's constituent elements. A node can have zero or more child nodes and holds data.

3. **Edge**: A link between two nodes is called an edge. It outlines the connections between nodes and the path we can take through the tree.

4. **Parent and Child**: A node in a tree may have one or more child nodes under it as parents. The nodes linked to a node are termed its offspring, while the node from which an edge originates is referred to as the parent.

5. **Leaf:** A leaf node is an unbroken node. It is the last node of a branch.

6. **Subtree:** A subtree is a section of a tree that consists of every node and all its offspring.

7. **Depth:** The distance along the path that connects a node at the root to that node in a tree is its depth. The depth of the root node is zero, that of its direct offspring is one, and so forth.

8. **Height**: The distance along the longest path from a root to a leaf node determines a tree's height. It shows the general height or depth of the tree.

Types of Trees:

1. **BINARY TREE:** A binary tree consists of at most two nodes, the left node and the right node. A binary tree can be used for a variety of purposes, such as binary search tree and expression tree.

2. **BINARY SEARCH TREE**: A binary search tree (BST) is a binary search tree with the following properties: The left child of the node contains values that are less than (or equal to) the value of the node. The right child of the node has values that are greater than the value of the value of the same node. This property makes the search and sorting operation efficient.

3. **AVL TREE:** An AVL tree (Self-Balancing Binary Search Tree) is a type of binary search tree where the height of each node in the AVL tree differs by a balance factor of not more than one.

4. **B-TREE**: A B-Tree is a self-balanced tree structure that is used to store a sorted list or a map of data. B-Trees are used in many databases and file systems to store and retrieve data efficiently.

5. **TRIE**: A trie (Prefix Tree) is a data structure that is like a tree. It stores a dynamic collection of strings or a key-value pair. It is used for autocompletion and dictionary implementation.

6. **BINARY HEAP**: A binary heap is a full binary tree that has the 'heap' property. It is used in many priority queues and in heap sort algorithms.

Trees in computer science and in data structures are referred to as balanced and unbalanced trees. These terms refer to how well a tree balances its subtrees. The balance between subtrees has a major influence on the performance of different operations on the tree.

1.1. Balanced Tree:

A balanced tree is a tree structure where the heights of subtrees for any given node do not differ significantly. In other words, the tree is designed to ensure that the maximum depth of the tree remains relatively small, which leads to efficient operations.

Common types of balanced trees include:

AVL Tree: A self-balancing binary search tree where the heights of the left and right subtrees of any node differ by at most one.

Red-Black Tree: Another self-balancing binary search tree that maintains balance by enforcing specific rules during insertion and deletion operations.

Benefits of balanced trees:

Efficient search, insertion, and deletion operations with time complexity O(log n) for most operations.

Predictable and controlled tree height, which ensures consistent performance.

1.2. Unbalanced Tree:

An unbalanced tree, also known as an imbalanced or skewed tree, is a tree structure in which the heights of subtrees for some nodes differ significantly. This lack of balance can result in inefficient operations, particularly when the tree degenerates into a long chain-like structure.

Unbalanced trees can occur in various situations, such as:

When elements are inserted in sorted order into a binary search tree (resulting in a degenerate tree with a height of n, where n is the number of elements).

In the worst-case scenario for a basic binary tree without balancing mechanisms.

Drawbacks of unbalanced trees:

Inefficient search, insertion, and deletion operations with a time complexity of O(n) in the worst case.

Performance degradation and the potential for the tree to become a linked list-like structure.

To ensure efficient operations, especially for search, insertion, and deletion, it is essential to use balanced trees when building data structures. AVL trees and Red-Black trees are examples of self-balancing trees that maintain their balance during operations. Unbalanced trees can still be useful in certain situations, but they should be carefully managed to avoid performance issues. Balancing mechanisms help keep the tree height under control, ensuring that the tree remains efficient regardless of the order in which elements are inserted or removed.

In-order, pre-order, and post-order are three common methods for traversing and exploring the nodes of a binary tree. These traversal techniques are used to visit all the nodes of a tree, each with a different order of visiting the parent and its children's nodes. Here's an explanation of each traversal method:

2. TREE TRAVERSAL TECHNIQUES:

2.1. **In-order Traversal**: In an in-order traversal, the nodes of a binary tree are visited in the following order:

- Visit the left subtree.
- Visit the current (root) node.
- Visit the right subtree.

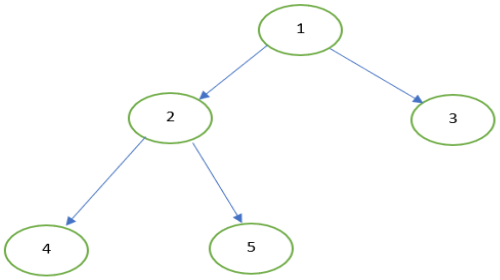

Figure 2.1.1 Tree

Order of Visiting: Left, Root, Right

The in-order traversal starts from the leftmost node of the tree, then moves to the current (root) node, and finally explores the right subtree.

1. Visit Node 4 (Left Subtree of 2)

Move to the leftmost node in the left subtree of 2.

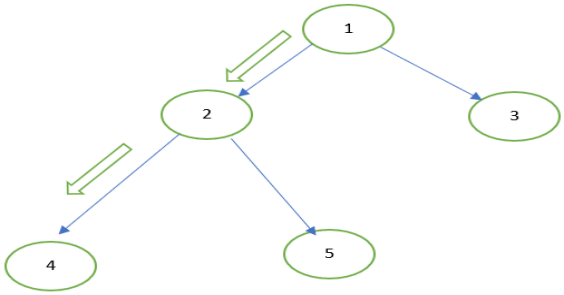

Figure 2.1.2 visit Node 4

2. Visit Node 2 (Root of the Subtree)

Move to the root of the subtree, which is 2.

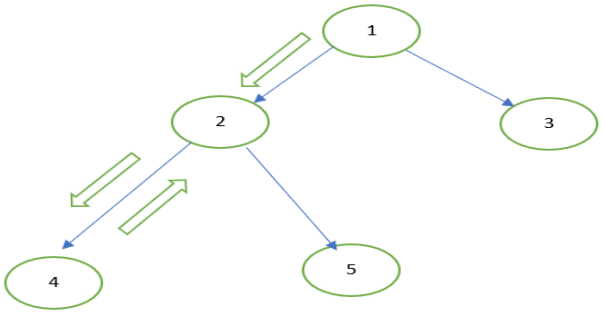

Figure 2.1.3 Visit Node 2

3. Visit Node 5 (Right Subtree of 2)

Move to the right subtree of 2 and visit the leftmost node, which is 5.

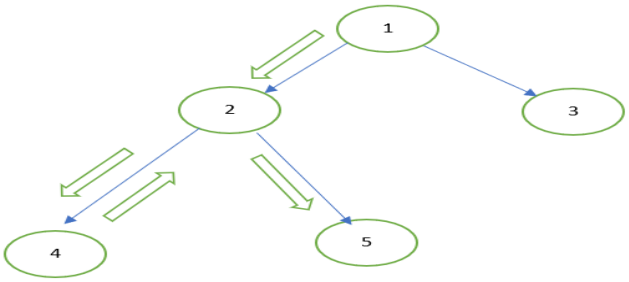

Figure 2.1.4 Visit Node 5

4. Visit Node 1 (Root of the Main Tree)

Move to the root of the entire tree, which is 1.

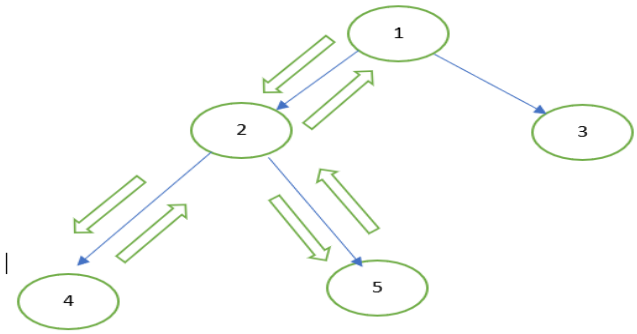

Figure 2.1.5 Visit Node 1

5. Visit Node 3 (Right Subtree of 1)

Move to the right subtree of 1 and visit the leftmost node, which is 3.

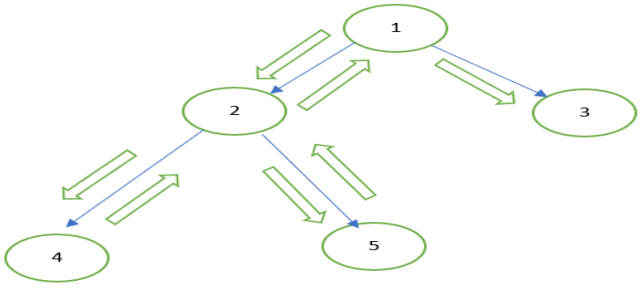

Figure 2.1.6 Visit Node 3

In-Order Traversal (Output: 4, 2, 5, 1, 3)

In other words, we start at the leftmost node, visit the left subtree, then the current node, and finally the right subtree. This traversal is commonly used in binary search trees (BST) to visit nodes in ascending order, as it visits nodes in sorted order. In-order traversal is typically implemented using a recursive function.

2.2 Pre-order Traversal: In a pre-order traversal, the nodes of a binary tree are visited in the following order:

- Visit the current (root) node.
- Visit the left subtree.
- Visit the right subtree.

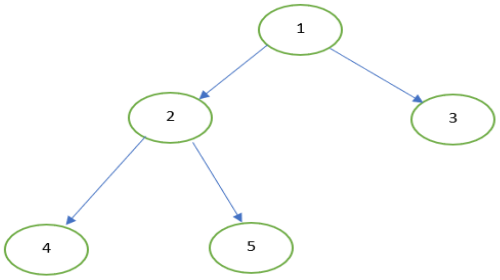

Figure 2.2.1 Tree

1. Visit Node 1 (Root of the Tree)

- Start at the root of the tree, which is 1.

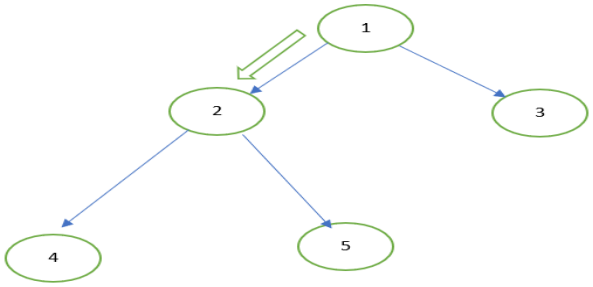

Figure 2.2.2 Visit Node 2

2. Visit Node 2 (Left Subtree of 1)

- Move to the left subtree of 1 and visit the root, which is 2.

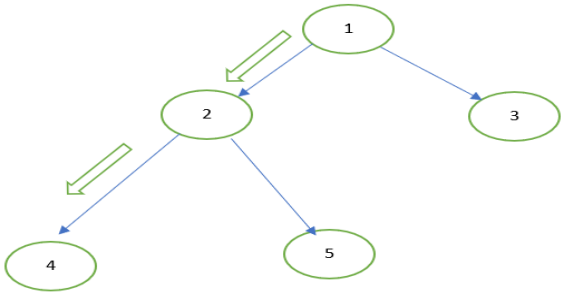

Figure 2.2.3 Visit Node 4

3. Visit Node 4 (Left Subtree of 2)

- Move to the left subtree of 2 and visit the root, which is 4.

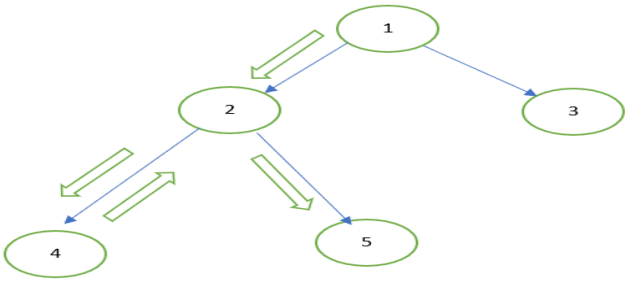

Figure 2.2.4 Visit Node 5

4. Visit Node 5 (Right Subtree of 2)

- Move to the right subtree of 2 and visit the root, which is 5.

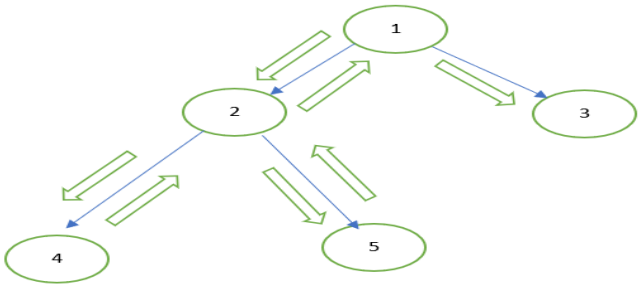

Figure 2.2.5 Visit Node 3

5. Visit Node 3 (Right Subtree of 1)

- Move to the right subtree of 1 and visit the root, which is 3.

Pre-Order Traversal: 1 2 4 5 3

This traversal starts at the root node, then moves to the left subtree and finally the right subtree. Pre-order traversal is often used in parsing expression trees and building a copy of a tree.

2.3. Post-order Traversal: In a post-order traversal, the nodes of a binary tree are visited in the following order:

- Visit the left subtree.
- Visit the right subtree.
- Visit the current (root) node.

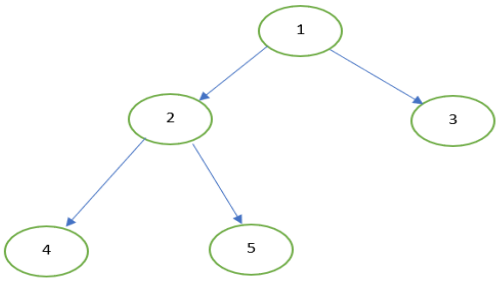

Figure 2.3.1 Tree

1. Visit Node 4 (Left Subtree of 2)

- Move to the left subtree of 2 and visit the leftmost node, which is 4.

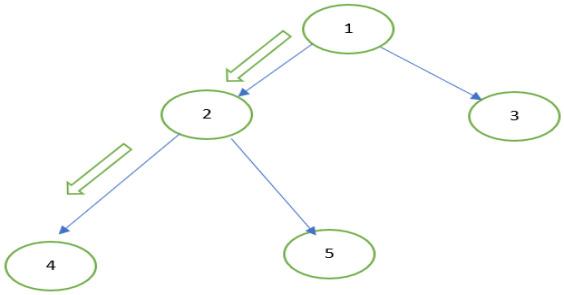

Figure 2.3.2 Visit Node 4

2. Visit Node 5 (Right Subtree of 2)

- Move to the right subtree of 2 and visit the leftmost node, which is 5.

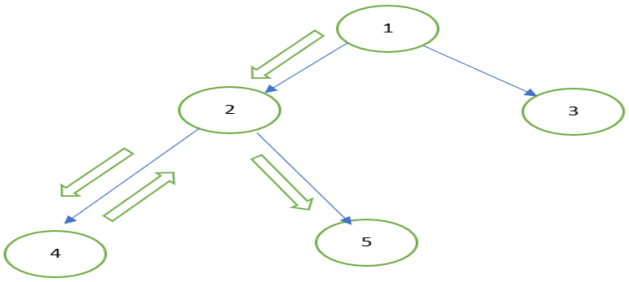

Figure 2.3.3 Visit Node 5

3. Visit Node 2 (Root of the Subtree)

- Visit the root of the subtree, which is 2.

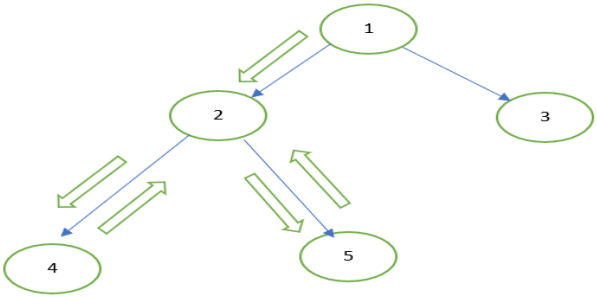

Figure 2.3.4 Visit Node 2

4. Visit Node 3 (Right Subtree of 1)

- Move to the right subtree of 1 and visit the leftmost node, which is 3.

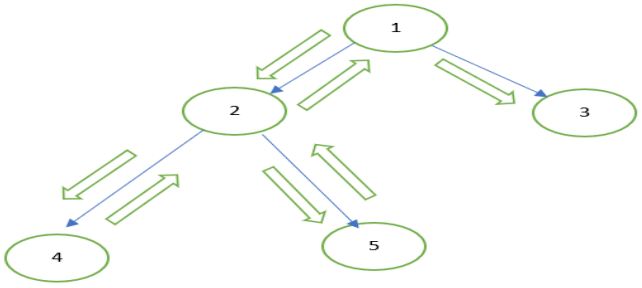

Figure 2.3.5 Visit Node 3

5. Visit Node 1 (Root of the Tree)

- Visit the root of the entire tree, which is 1.

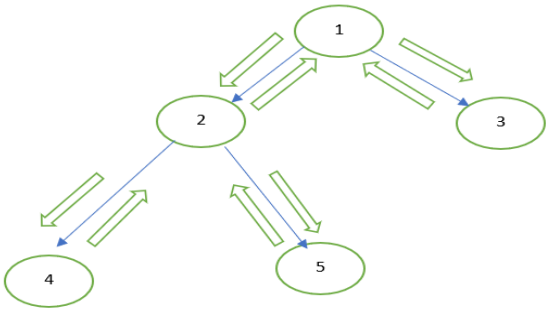

Figure 2.3.6 Visit Node 1

Post-Order Traversal: 4 5 2 3 1

Post-order traversal starts at the left subtree, then moves to the right subtree, and finally visits the current node. It is often used when we need to perform some action after visiting both left and right subtrees before moving to the parent node.

2.4 Recursive vs Iterative approaches:

As of now, the tree traversal techniques are explained. But, to traverse a tree using a technique, we can traverse using recursive or iterative approach. To differentiate in simple terms, an iterative function is one that loops to repeat some part of the code, and a recursive function is one that calls itself again to repeat the code.

Iteration is faster and more space-efficient than recursion. So why do we even need recursion? The reason is simple — it's easier to code a recursive approach for a given problem. Try doing in order tree traversal using recursion and iteration both.

Strengths and Weaknesses of Iteration:

Strengths:

a) Iteration can be used to repeatedly execute a set of statements without the overhead of function calls and without using stack memory.

b) Iteration is faster and more efficient than recursion.

c) It's easier to optimize iterative codes, and they generally have polynomial time complexity.

d) They are used to iterate over the elements present in data structures like an array, set, map, etc.

e) If the iteration count is known, we can use for loops; else, we can use while loops, which terminate when the controlling condition becomes false.

Weaknesses:

a) In loops, we can go only in one direction, i.e., we can't go or transfer data from the current state to the previous state that has already been executed.

b) It's difficult to traverse trees/graphs using loops.

c) Only limited information can be passed from one iteration to another, while in recursion, we can pass as many parameters as we need.

Strengths and Weaknesses of Recursion:

Strengths:

a) It's easier to code the solution using recursion when the solution of the current problem is dependent on the solution of smaller similar problems.

b) cRecursive codes are smaller and easier to understand.

c) We can pass information to the next state in the form of parameters and return information to the previous state in the form of the return value.

d) It's a lot easier to perform operations on trees and graphs using recursion.

Weaknesses:

a) The simplicity of recursion comes at the cost of time and space efficiency.

b) It is much slower than iteration due to the overhead of function calls and control shift from one function to another.

c) It requires extra memory on the stack for each recursive call. This memory gets deallocated when function execution is over.

d) It is difficult to optimize a recursive code, and they generally have higher time complexity than iterative codes due to overlapping subproblems.

To compare, Iterative programming and recursive programming each have their advantages and disadvantages. Iterative programming is often more efficient for solving problems that require repetitive iterations, and it can be easier to read and understand. However, iterative programming can become complex for some problems and may require more lines of code than recursive programming. Recursive programming, on the other hand, can be more efficient for solving problems with recursive structures, and it often requires less code than iterative programming.

Usage of either of these techniques is a trade-off between time complexity and size of code. If time complexity is the point of focus, and the number of recursive calls would be large, it is better to use iteration. However, if time complexity is not an issue and shortness of code is, recursion would be the way to go.

2.5 Expression Trees and Evaluating Expressions

Definition: Expression Trees are binary trees used to represent expressions. Each leaf node in an expression tree represents an operand, and each internal node represents an operator. This structure facilitates the evaluation of arithmetic expressions and the conversion between different expression notations.

Evaluating Expressions: To evaluate an expression using an expression tree, one performs a post-order traversal (left-right-root). During this traversal, each operator node combines the values of its child nodes according to its operator.

Example: Consider the expression 3 + ((5 + 9) * 2). The corresponding expression tree would be:

- The root node is the + operator, representing the addition of 3 and the result of the multiplication operation.
- The left child of the root is **3**.
- The right child of the root is a * node, representing the multiplication.
- The left child of the * node is another + node, representing the addition of 5 and 9.
- The right child of the * node is **2**.
- The left child of the + node under the * node is **5**.
- The right child of the + node under the * node is **9**.

Evaluating this tree involves first evaluating **5 + 9** to get **14**, then 14*2 will be 28 and then add the result with **3** to obtain **31**.

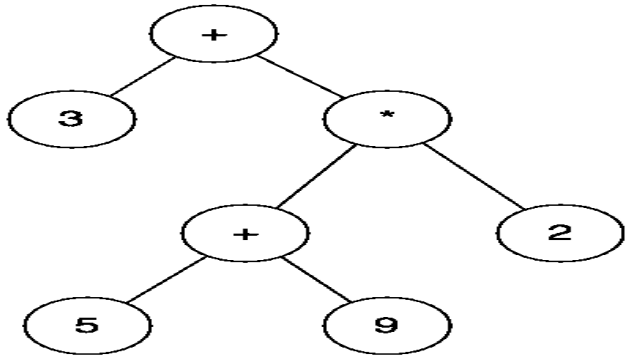

Figure 2.5.1 Tree structure for 3 + ((5 + 9) * 2).

ALGORITHM 2.5.1: Expression Tree

```
Let t be the expression tree
If t is not null then
    If t.value is operand then
        Return t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

Expression trees are data structures that represent expressions as trees. They have several benefits, including:

1. Complex expressions: Expression trees can be used to make complex expressions.
2. Associativity: Expression trees can be used to determine the associativity of each operator in an expression.
3. Expression evaluation: Expression trees can be used to solve infix, prefix, and postfix expression evaluation.
4. Code manipulation: Expression trees allow code to be manipulated during run time.
5. Dynamic code building: Expression trees allow code to be built dynamically at runtime.

Some other uses of expression trees include:

- Producing equation formatters
- Symbolic manipulation of equations

3. AVL Trees:

Definition: The AVL Tree, named after its inventors Adelson-Velsky and Landis, represents a pinnacle of ingenuity in computer science. It's a self-balancing binary search tree, where the height difference between the left and right subtrees of any node is always one or zero. This subtle yet powerful balance criterion is the key to its efficiency, ensuring O(log n) time complexity for essential operations like search, insertion, and deletion.

The heart of an AVL Tree's magic lies in its balancing act. Whenever a node's balance - the height difference between its left and right subtree - strays beyond the -1 to 1 range, the tree performs a graceful dance of rotations to restore balance. These rotations, the core of AVL Tree's algorithm, come in four main types:

3.1 Types of Rotations:

1. **Single Right Rotation (LL Rotation):**

   This is employed when the left subtree of an unbalanced node gets a new left child, tilting the balance.

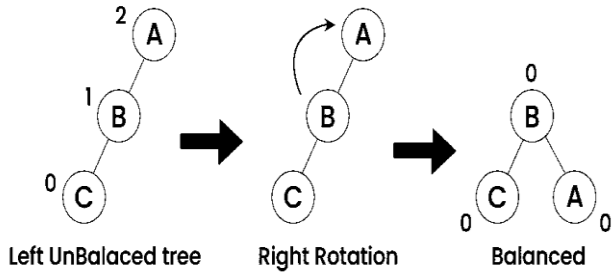

Figure 3.1.1: Right Rotation Tree

2. **Single Left Rotation (RR Rotation):**

   The opposite scenario, where a right child is added to the right subtree of an unbalanced node.

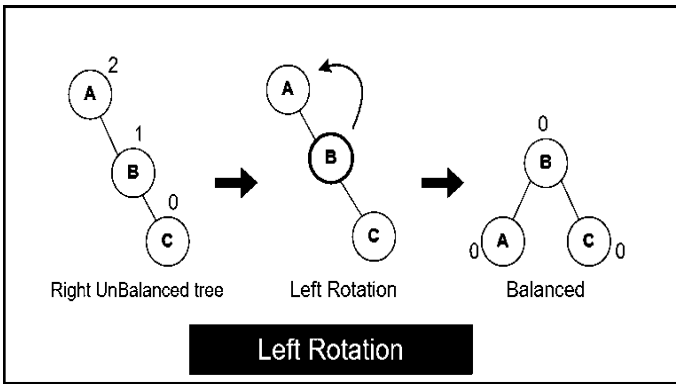

Figure 3.1.2: Left Rotation Tree

3. **Left-Right Rotation (LR Rotation):**

   A double maneuver starting with a left rotation on the left child, followed by a right rotation on the unbalanced node.

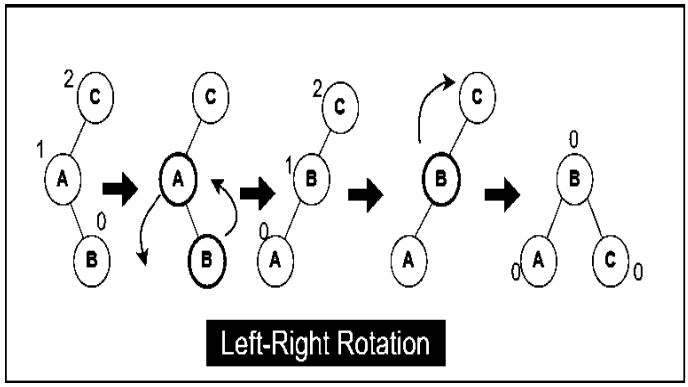

Figure 3.1.3: Left-Right Rotation Tree

4. **Right-Left Rotation (RL Rotation):**

   Another double act, but this time beginning with a right rotation on the right child, then a left rotation on the unbalanced node.

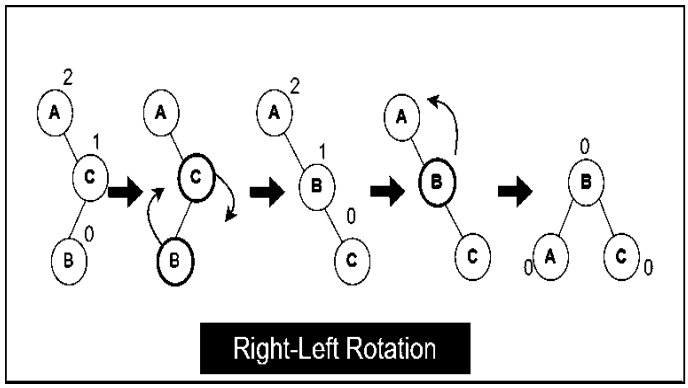

Figure 3.1.4: Right- Left Rotation Tree

ALGORITHM 3.1: AVL Tree

The following steps are involved in performing the insertion operation of an AVL Tree −

Step 1 − Create a node

Step 2 − Check if the tree is empty

Step 3 − If the tree is empty, the new node created will become the root node of the AVL Tree.

Step 4 − If the tree is not empty, we perform the Binary Search Tree insertion operation and check the balancing factor of the node in the tree.

Step 5 − Suppose the balancing factor exceeds ±1, we apply suitable rotations on the said node and resume the insertion from Step 4.

3.2 Insertion:

The insertion process in an AVL Tree is a step-by-step journey of meticulous checks and balances. Starting with creating a node, it proceeds to check if the tree is empty. If not, it follows the traditional binary search tree insertion path, constantly monitoring the balance factor. When this factor steps out of the -1 to 1 range, the appropriate rotations are elegantly executed to maintain the tree's equilibrium.

3.2.1 Insertion Example

For instance, constructing an AVL tree with integers 1 to 7, we see these principles in action.

Figure 3.2.1.1

Starting with the first element 1, we create a node and measure the balance, i.e., 0.

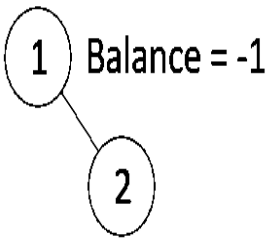

Figure 3.2.1.2

Since both the binary search property and the balance factor are satisfied, we insert another element into the tree.

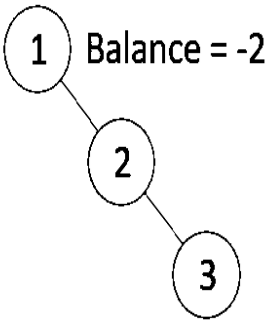

Figure 3.2.1.3

The balance factor for the two nodes are calculated and is found to be -1 (Height of left subtree is 0 and height of the right subtree is 1). Since it does not exceed 1, we add another element to the tree.

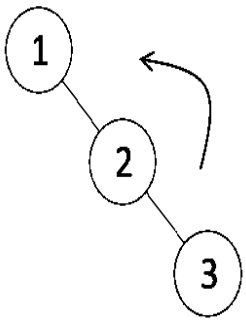

Figure 3.2.1.4

Now, after adding the third element, the balance factor exceeds 1 and becomes 2. Therefore, rotations are applied. In this case, the RR rotation is applied since the imbalance occurs at two right nodes.

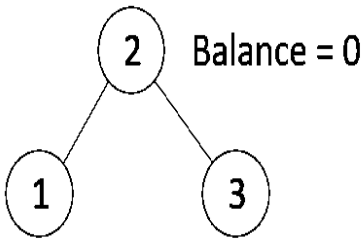

Figure 3.2.1.5

The tree is rearranged as above.

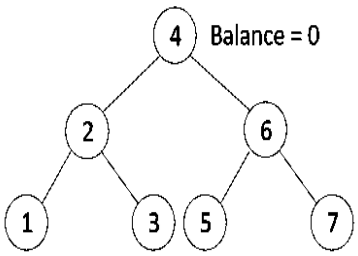

Figure 3.2.1.6

Similarly, the next elements are inserted and rearranged using these rotations. After rearrangement, we achieve the tree as above.

3.3 Deletion

Deletion in AVL Trees, although similar in its concern for balance, varies based on the node's status - a leaf, having one child, or two children. Each scenario demands a different approach but shares the common goal of maintaining the delicate balance. If the balance is disturbed post-deletion, rotations are again the saviors, realigning the tree to its stable state. Deletion in the AVL Trees take place in three different scenarios −

1. **Deletion of a Leaf Node:** This scenario seems straightforward but holds hidden complexity. Deleting a leaf node doesn't upset the binary search tree's order, but it can unbalance the delicate equilibrium of an AVL Tree. Here, the tree's innate intelligence shines through. It evaluates the new structure and, if necessary, performs rotations to rebalance itself. It's akin to a skilled tightrope walker, who subtly adjusts their balance with each step to prevent a fall.
2. **Deletion of a Node with One Child:** This is where the AVL Tree's adaptability is further tested. Upon deleting a node with a single offspring, the tree doesn't just remove the node; it assimilates the child's value into the vacated position. This maneuverer ensures the tree's structural integrity. However, this change can tilt the balance, compelling the tree to execute rotations to regain its poise. It's a sophisticated dance of values and structures, ensuring the tree remains both a binary search tree and a balanced AVL Tree.
3. **Deletion of a Node with Two Children:** This is the most intricate of the scenarios, requiring a nuanced approach. The tree identifies the inorder successor of the node to be deleted, a leaf node ideally positioned to maintain the binary search order. This successor's value supplants the original node's value. Then, the successor node, now redundant, is deleted. However, this action might disrupt the balance, necessitating a careful application of balance algorithms.

Each of these deletion processes demonstrates the AVL Tree's inherent genius in maintaining balance and order. It's not just about removing a node; it's about preserving the tree's fundamental properties - its binary search capability and its balance. This intricate choreography of deletions and adjustments exemplifies the elegance and efficiency of AVL Trees, making them a cornerstone of efficient data handling in computer science.

Such a nuanced approach to maintaining balance in AVL Trees underscores their significance in the realm of algorithms and data structures. Their ability to self-regulate and adapt ensures optimal performance, a quality that is indispensable in the fast-paced world of technology. This dynamic interplay of structure and balance in AVL Trees provides a rich field of study and application, underscoring their enduring relevance and utility.

3.3.1 Deletion Example

Using the same tree given above, let us perform deletion in three scenarios −

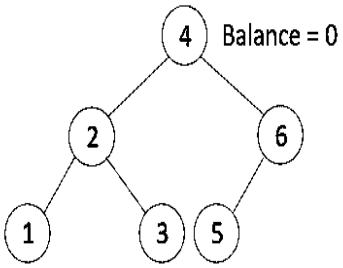

Figure 3.3.1.1

AVL Tree example for Deletion.

Deleting element 7 from the tree above −

Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree

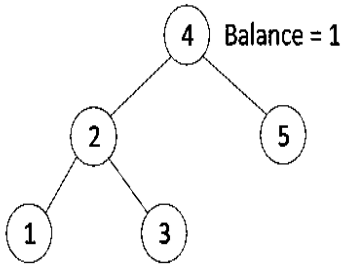

Figure 3.3.1.2

After deleting 7 from tree.

Deleting element 6 from the output tree achieved −

However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node: node 5.

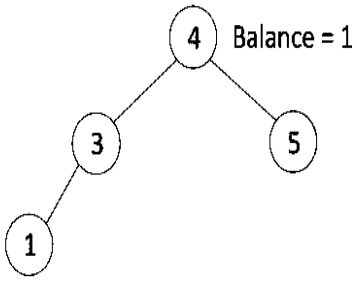

Figure 3.3.1.3

After deleting 6 from tree.

The balance of the tree becomes 1, and since it does not exceed 1 the tree is left as it is. If we delete the element 5 further, we will have to apply the left rotations; either LL or LR since the imbalance occurs at both 1-2-4 and 3-2-4.

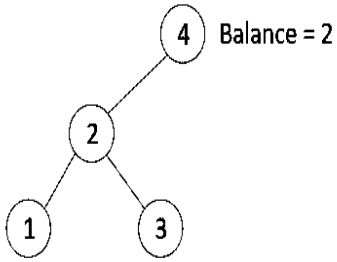

Figure 3.3.1.4

After deleting 5 from tree.

The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here).

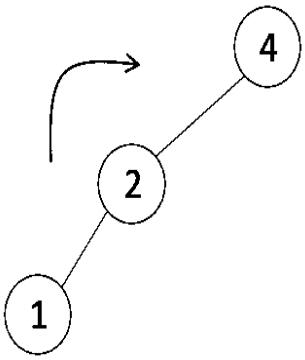

Figure 3.3.1.5

After deleting 5 from tree, we need to apply LL rotation to balance the tree.

Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4.

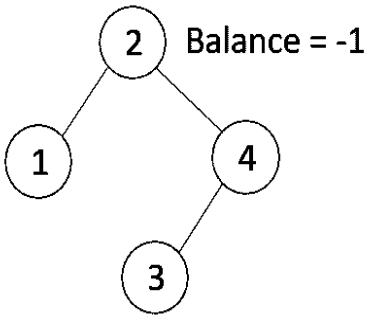

Figure 3.3.1.6

After to balancing the tree.

Deleting element 2 from the remaining tree −

As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor.

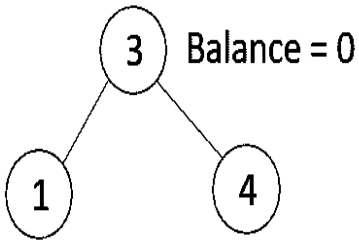

Figure 3.3.1.7

After deleting 3 from tree.

The balance of the tree remains 1, therefore we leave the tree as it is without performing any rotations.

3.3 Applications of AVL Trees:

1. **Database Indexing**: AVL Trees shine in indexing large records in databases. Their self-balancing nature ensures that search operations remain swift and efficient, a critical requirement in database management.
2. **In-Memory Collections**: For sets and dictionaries, where in-memory speed is crucial, AVL Trees offer an optimal solution. Their balanced structure enables quick access and modification, key for real-time data handling.
3. **Database Applications**: Particularly in scenarios where data lookups are frequent, but insertions and deletions are rare, AVL Trees provide an ideal balance between maintenance and access speed.
4. **Optimized Search Software**: Any software that prioritizes search efficiency, be it in file systems or data analysis tools, can benefit significantly from the AVL Tree's rapid lookup capabilities.
5. **Gaming and Corporate Applications**: From complex storylines in gaming to data-driven decision-making in corporate environments, AVL Trees offer a robust framework for handling intricate data structures efficiently.

Advantages of AVL Trees:

1. **Self-Balancing Nature**: The ability of AVL Trees to maintain balance autonomously is a marvel. It ensures optimal height and, hence, optimal search and access times.
2. **Non-Skewed Structure**: Unlike some other tree structures, AVL Trees are guaranteed to be non-skewed, which directly contributes to their efficiency.
3. **Efficient Lookups**: When compared to Red-Black Trees, AVL Trees typically offer faster lookups, making them preferable in situations where search time is critical.
4. **Superior Searching Time Complexity**: Compared to binary trees and other data structures, AVL Trees offer better search time complexity, which is a significant advantage in data-intensive applications.
5. **Height Efficiency**: The height of an AVL Tree is capped at log(N), with N being the total number of nodes. This limitation ensures that the tree always remains balanced and efficient.

Disadvantages of AVL Trees:

1. **Complex Implementation: The** sophistication of AVL Trees comes at a cost – they are challenging to implement, requiring a deep understanding of tree structures and rotations.
2. **High Constant Factors**: Certain operations in AVL Trees have high constant factors, which can impact performance in specific scenarios.
3. **Less Popular than Red-Black Trees**: Despite their advantages, AVL Trees are often overshadowed by Red-Black Trees, which are more commonly used in standard libraries due to their simpler implementation and good enough balancing.
4. **Complicated Insertion and Removal**: The strict balance requirement of AVL Trees means that insertions and removals can be complex, involving multiple rotations.
5. **Intensive Balancing Process**: The process of maintaining balance, while beneficial for search operations, requires additional processing, which can be a drawback in environments where insertions and deletions are frequent.

4. HUFFMAN CODING:

Huffman coding is one of the application or implementation of Trees concept. When transmitting the data from source to destination, along with the transmission techniques, the encryption also plays a vital role. Huffman coding is mainly known for “Loss Less Data compression”, which is a class of data compression that allows the original data to be perfectly reconstructed from the compressed data with no loss of information. Huffman coding is first introduced by David A. Huffman, in a research paper named: A Method for the Construction of Minimum-Redundancy Codes", which summary states that it is an optimum method of coding an ensemble of messages consisting of a finite number of members is developed. A minimum-redundancy code is one constructed in such a way that the average number of coding digits per message is minimized.

The other reason that Huffman code is famous for is "optimal prefix code", which is a type of code system distinguished by its possession of the "prefix property", which requires that there is no whole code word in the system that is a prefix (initial segment) of any other code word in the system. Prefix codes are also known as prefix-free codes, prefix condition codes and instantaneous codes. Although Huffman coding is just one of many algorithms for deriving prefix codes, prefix codes are also widely referred to as "Huffman codes", even when the code was not produced by a Huffman algorithm. The term comma-free code is sometimes also applied as a synonym for prefix-free codes but in most mathematical books and articles a comma-free code is used to mean a self-synchronizing code, a subclass of prefix codes. Hence, after encoding the data with Huffman coding, the generated coding satisfies the prefix code property and hence there will be no ambiguity when decoding. This ensures the efficiency of the transmission. The example for prefix codes is as demonstrated in the following.

Let us imagine the encoding for four symbols a, b, c, d is 0, 1, 00 and 01, which does not follow the prefix code property. And, let the transmitted encrypted message at the destination is 0011. Now, to retrieve the original symbols, if we consider 0 as ‘a’ and 1 as ‘b’, the decoded symbols sequence will be ‘aabb’. But, if we consider 00 as ‘c’ and 01 as ‘d’, the transmitted sequence could also be ‘acb’ or ‘cbb’ or ‘adb’. As there are number of possibilities for the encoded messages could be, there is ambiguity in retrieving the original transmitted message at the destination or sink end. Now, let us consider the other example encoding for four symbols a, b, c, d as 00, 11, 010, 011 and let the transmitted encrypted message be same: 0011. If we decode this, the only possibility for this is “ab”. There is no other decoded message except this. In this way, there is no disambiguity in the decoding process involving Huffman codes and this ensures the efficiency of the decryption.

There is interesting history behind the Huffman codes. In Wikipedia, it is described as: “In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code. Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient. In doing so, Huffman outdid Fano, who had worked with Claude Shannon to develop a similar code. Building the tree from the bottom-up guaranteed optimality, unlike the top-down approach of Shannon–Fano coding.”

Now, let’s go through the Huffman codes generation and the procedure of encoding and decoding. The main idea behind the Huffman coding is to assign “variable length codes” to input characters and lengths or sizes of assigned codes are based on frequencies of the corresponding characters. That means, the less frequent character will have highest length or size, whereas the most frequent character will have lowest length or size after encoding. This algorithm is well-known as “Greedy approach”. First, we need to build the Huffman tree in bottom-up manner.

The step-by-step procedure to produce Huffman tree and codes are explained below:

Input is an array of unique characters along with their frequency of occurrences.

output is Huffman Tree.

a) Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

b) Extract two nodes with the minimum frequency from the min heap.

c) Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

d) Repeat steps #b) and #c) until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let’s build the Huffman tree with the above steps: Consider the input as “HelloWorld”. Now let’s first develop the table of frequencies and then build the Huffman tree and finally observe the generated Huffman code for every character.

The table of frequencies as follows:

Character	H	e	l	o	W	r	d
Frequency	1	1	3	2	1	1	1

Based on the above steps in algorithm, the demonstration of each step is:

Step 1: Consider all characters as leaf nodes, with corresponding frequencies as capacities of the nodes.

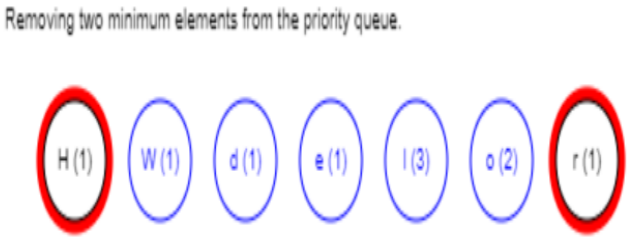

Figure 4.1 Step 1

Step 2: Take the two nodes with minimum capacities (H-1, r-1) after sorting the frequencies and add the capacities as the new parent of those two nodes, that is 1+1=2.

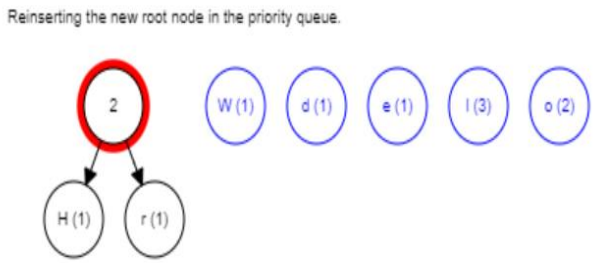

Figure 4.2 Step 2

Step 3: Again, sort the frequencies and the select the first 2 minimum capacity nodes as in the following figure. W-1 and d-1.

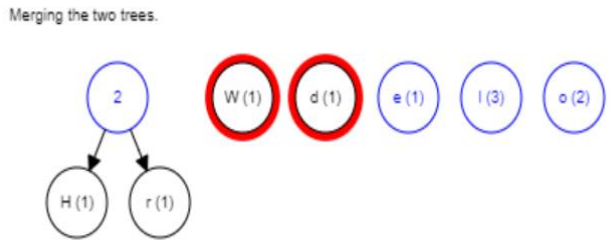

Figure 4.3 Step 3

Step 4: Repeat the same step and add the selected nodes to get the parent node with capacity as the sum of the two selected nodes, that is 1+1=2.

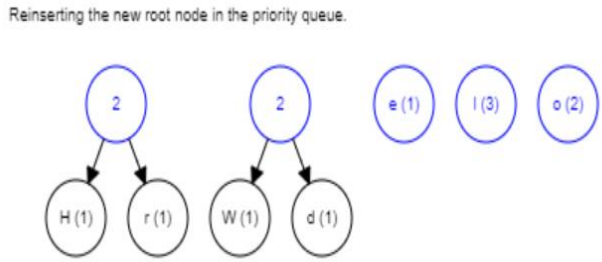

Figure 4.4 Step 4

Step 5: After sorting all the recent nodes, select the two minimum capacity nodes. e-1 and node with capacity 2.

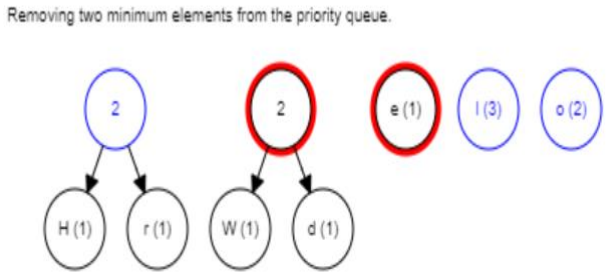

Figure 4.5 Step 5

Step 6: Add the selected nodes to get the parent node with calculated capacity as the sum of those two. 2+1=3.

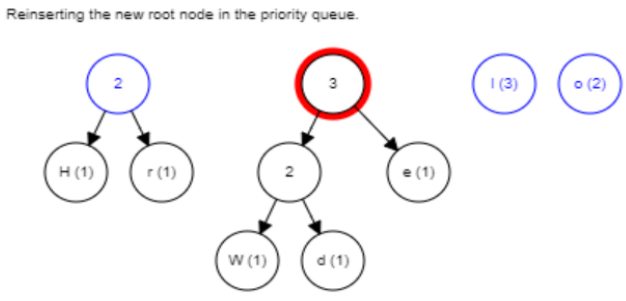

Figure 4.6 Step 6

Step 7: Repeat the same step again and select the minimum two capacity nodes. 2, 2 as shown in the following:

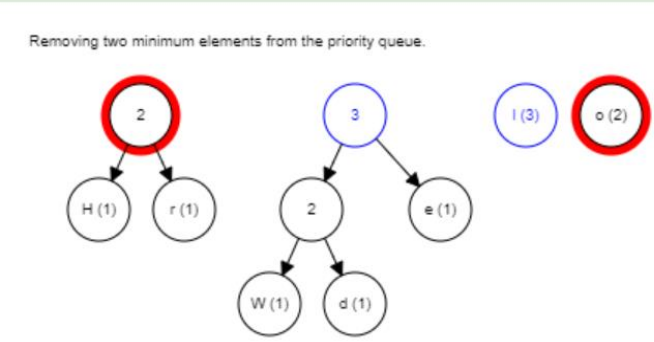

Figure 4.7 Step 7

Step 8: Merging the selected two nodes as a parent node of those two nodes.

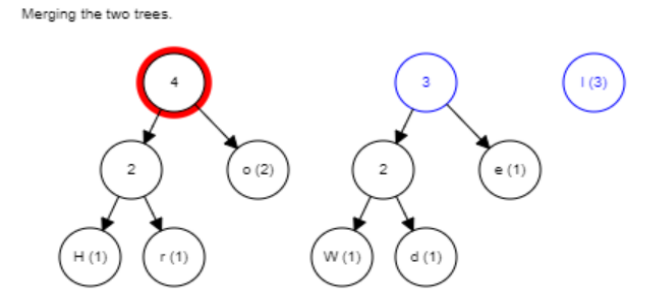

Figure 4.8 Step 8

Step 9: After sorting the updated node capacities, selecting the first two minimum, that are 3,3.

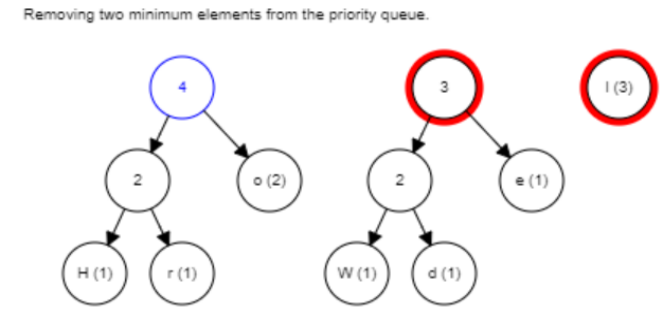

Figure 4.9 Step 9

Step 10: Adding up the two selected nodes and merging them up to a parent node, with capacity is the addition of those two nodes.

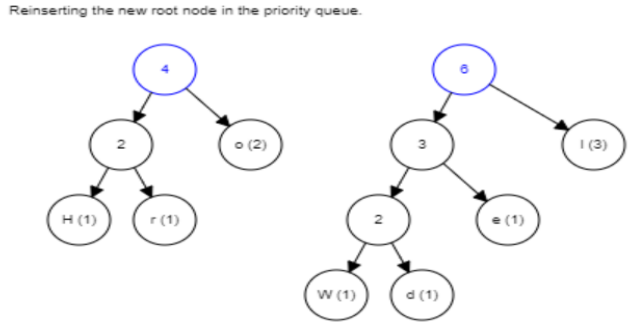

Figure 4.10 Step 10

Step 11: Now, there is only two nodes with capacities 4,6 left and so select these two nodes.

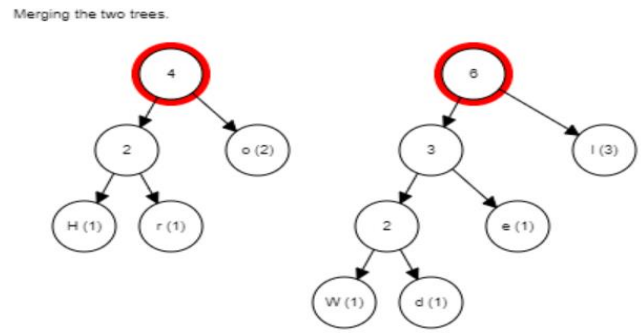

Figure 4.11 Step 11

Step 12: Finally adding the last two nodes to get the parent node of the entire tree, with capacity 4+6=10 = Sum of frequencies of all the characters = Sum of all node capacities.

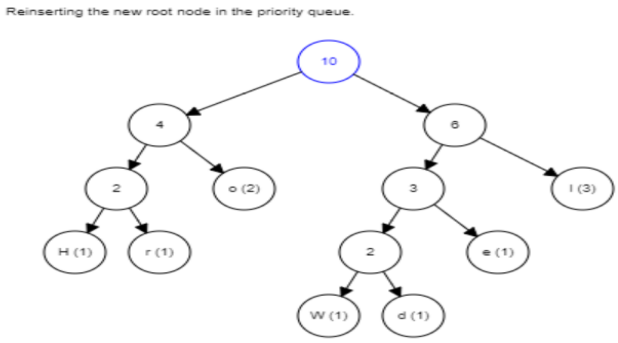

Figure 4.12 Step 12

The generated Huffman tree of given input is:

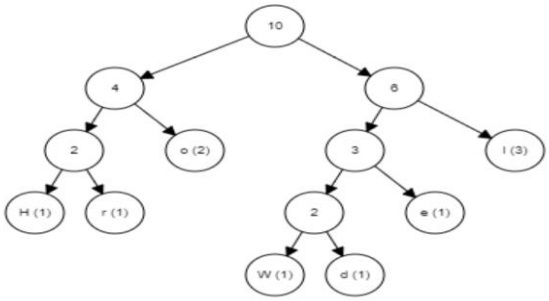

Figure 4.13 Final Huffman Tree

To get the Huffman code for each character, we need to give 0s and 1s for the left and right branch of each subtree.

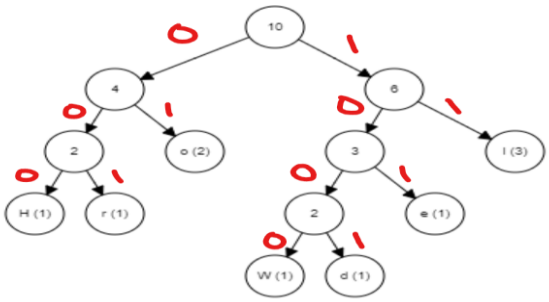

Figure 4.14 Huffman tree with assigned 0s, 1s.

To get Huffman code of each character, the procedure is to traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.

Character (Frequency)	Huffman code
H	000
e	101
l	11
o	01
W	1000
r	001
d	1001

If we observe, the generated Huffman codes follows the properties of Variable length, Prefix-free and Lossless data compression. The character with lowest frequency has the highest length and the character with highest frequency has the lowest length of all Huffman codes.

The most frequent character is the closest node to the root, and hence has the lowest length.

Time complexity: **O(nlogn)** where n is the number of unique characters. If there are n nodes, extractMin() is called 2*(n – 1) times. extractMin() takes O(logn) time as it calls minHeapify(). So, the overall complexity is O(nlogn).

Huffman Decoding:

To decode the encoded data, we require the Huffman tree. We iterate through the binary encoded data. To find character corresponding to current bits, we use the following simple steps:

a) We start from the root and do the following until a leaf is found.

b) If the current bit is 0, we move to the left node of the tree.

c) If the bit is 1, we move to right node of the tree.

d) If during the traversal, we encounter a leaf node, we print the character of that particular leaf node and then again continue the iteration of the encoded data starting from step 1.

In this way, we can decode the transmitted characters from Huffman tree.

KEYWORDS

Data Structures, Tree Traversal, Binary Trees, Balanced Trees, AVL Trees, Huffman Trees, Algorithm Efficiency, Recursion and Iteration, Data Compression, Tree Rotations, Graph Theory, Computational Complexity, Real-world Applications of Trees, Software Engineering Memory Management

ACKNOWLEDGMENTS

We extend our heartfelt appreciation to the contributors who played an integral role in the conception and completion of this ACM paper on tree data structures. A special acknowledgment is owed to

our Team members, whose expertise and unwavering commitment significantly shaped the research, writing, and presentation of the comprehensive exploration of tree data structures.

Our gratitude extends to our professor Dr. Wolf Dieter Otte, for his invaluable support and resources, which empowered us to delve into the nuances of tree structures and offer meaningful insights into their practical applications.

This collaborative effort involved the dedicated contributions of our Team members, and we recognize and appreciate their academic and practical input. Their collective commitment has greatly enhanced the substance and caliber of this paper, establishing it as a valuable resource for both academic research and practical implementations in the dynamic field of computer science.

REFERENCES

[1] Tutorial Points: https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm.
[2] Geeks for Geeks: https://www.geeksforgeeks.org/introduction-to-avl-tree/
[3] Ian Editor (Ed.). 2007. *The title of book one* (1st. ed.). The name of the series one, Vol. 9. University of Chicago Press, Chicago. DOI: https://doi.org/10.1007/3-540-09237-4.
[4] David Kosiur. 2001. *Understanding Policy-Based Networking* (2nd. ed.). Wiley, New York, NY
[5] https://en.wikipedia.org/wiki/Huffman_coding
[6] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/