Average Binary Tree (ABT): A Novel Formal Tree-to-Tree Transformation with Pairwise Sibling Aggregation

Vikas Kumar
Department of Computer Engineering
Army Institute of Technology
Pune,India
vikaskumar_240252@aitpune.edu.in

***Abstract*—We introduce the *Average Binary Tree* (ABT), a formally defined derived data structure arising from a structural operator $f : T \to T'$ applied to a perfect binary tree T of height h . Under this operator, each node $u \in T'$ is assigned the arithmetic mean of a sibling pair $(L(p), R(p))$ belonging to T while the edge topology of T' mirrors that of the internal subgraph of T .U˙ nlike aggregation methods whose output is a scalar or a flat sequence, the ABT yields a structurally well-defined binary tree amenable to formal analysis . Ẇ e establish four theorems governing size contraction, height reduction, value containment, and linearity of the operator, together with two additional structural properties .A˙ n $O(n)$-time, $O(n)$-space algorithm is presented with a full correctness argument by structural induction .A˙ Python reference implementation is validated across six test categories, each accompanied by automatic theorem-checking B˙ enchmarks against segment-tree aggregation and array-based pyramid reduction confirm the linear-time advantage. Three generalisations–the Weighted ABT (WABT), Median Binary Tree (MBT), and Probabilistic Binary Tree (PBT)–extend the framework to priority-weighted outlier-robust and uncertainty-aware settings respectively. Application scenarios in wireless sensor networks , multi-resolution image analysis , hierarchical traffic modelling and distributed computing illustrate the practical reach of the construction.**

***Index Terms*—binary tree; tree transformation; derived data structure; pairwise aggregation; hierarchical compression; average pooling; tree-to-tree mapping; BFS traversal; structural operator**

I. Introduction

Tree-shaped data structures occupy a central role in theoretical computer science and practical algorithm design [1], [2], [29]. Their hierarchical organisation naturally models recursive decomposition [26], [30], and this property is routinely exploited in range-query structures [4], decision procedures [5], [17], parse representations [3], and distributed protocols [20], [21].

Despite this long history, the question of how to *transform* one binary tree into another while preserving structural identity—rather than collapsing it into a scalar summary—has received comparatively little formal treatment [18], [19]. Existing aggregation approaches produce flat arrays of per-level statistics [24], energy-optimal spanning structures for sensor networks [7]–[9], or sparse graphical models [14], [15]. None of these yields a *new binary tree* whose node-level topology is a direct structural image of the original [25].

This paper introduces the **Average Binary Tree (ABT)**, a formally defined tree data structure produced by the *sibling-averaging operator* $\mathbf{f} : \mathrm{T} \to \mathrm{T}'$ [10], [11]. For every internal node p of a perfect binary tree T [1], the operator creates a corresponding node u = f(p) in T' whose value equals the arithmetic mean of the left and right children of p [2], [26]. The output T' is itself a valid binary tree whose height is exactly one less than that of T [29], [30], making the construction amenable to repeated application [10].

The novelty of the ABT resides not in the averaging arithmetic, which is classical [10]–[12], but in the *formalisation* of the output as a tree structure with provable properties—an aspect absent from prior work [8], [14], [16]. Our specific contributions are:

- **C1.** A rigorous mathematical definition [1], [2] of the sibling-averaging operator $\mathbf{f} : \mathrm{T} \to \mathrm{T}'$ with explicit node-correspondence and edge-preservation rules.
- **C2.** Four theorems (size contraction, height reduction, value containment, and operator linearity) each with complete proofs [2], [29], plus two additional structural properties.
- **C3.** An O(n)-time, O(n)-space algorithm [2], a formal correctness lemma by structural induction [1], and a complexity theorem.
- **C4.** A fully validated Python implementation [34] covering six test categories with automatic theorem verification.
- **C5.** Empirical benchmarks comparing ABT construction against segment-tree aggregation [4], [25] and array-based pyramid methods [10].
- **C6.** Three generalisations—WABT, MBT, and PBT [5], [8], [17]—broadening the framework to weighted, robust, and probabilistic regimes.
- **C7.** Discussion of four practical application domains supported by domain-specific literature [9], [10], [21], [22].

Section II surveys related work. Section III develops the formal framework [1], [2]. Section IV presents and proves the theorems. Section V gives the algorithm [2], [29]. Section VI describes the implementation and benchmarks [34].

Section VII covers the three extensions. Section VIII discusses applications. Section IX analyses limitations and open problems. Section X concludes.

II. Background and Related Work

A. Tree-Based Aggregation in Sensor Networks

Hierarchical aggregation over binary-tree network topologies [30] has been studied for its communication-reduction benefits [7]–[9]. Krishnamachari et al. [8] show that in-network aggregation can cut message volume by half under ideal conditions, but their analysis targets the scalar output at the root, not the intermediate tree of partial aggregates [25]. Hwang et al. [7] propose constructing energy-balanced aggregation trees by iteratively adjusting the spanning structure after an initial build; their objective is minimal energy expenditure [9], not structural formalisation of the aggregated result [19]. A systematic survey of aggregation strategies for hierarchical sensor networks is given by Khan et al. [9].

B. Segment Trees and Range Queries

Segment trees [4] support $O(\log n)$ range queries by storing aggregate values of array sub-intervals at internal nodes [2], [25]. The construction is structurally similar to a bottom-up pass over a complete binary tree [1], [29], yet it fundamentally operates on a *linear* array rather than on a tree-valued domain [26]. Mehta and Sahni [25] give a comprehensive treatment of interval trees and segment trees as range-query primitives [4], none of which define a secondary tree as their output.

C. Linear and Statistical Tree Aggregation

Nguyen et al. [16] embed linear aggregation functions inside tree-based estimators for regression, demonstrating MSE improvements over constant-leaf models [5], [6]. Their node-merging function is arithmetically equivalent to our operator—it is a weighted linear combination of two child values [16]—but the output of their procedure is a scalar prediction vector, not a binary tree structure [2]. Bien and Yan [14] introduce the *tag-lasso*, a tree-guided regulariser that promotes node fusion in sparse graphical models [14], [15]. Their downstream output is a precision matrix, not a tree [25]. Yan and Bien [15] extend this framework to microbiome composition modelling [33].

D. Multi-Resolution Image Analysis

Burt and Adelson [10] developed the Gaussian and Laplacian pyramid representations, wherein each subsequent image level is derived from low-pass filtering and sub-sampling the preceding level [10], [27]. The filtering step is a weighted average over a local neighbourhood [12], [13]; in a 1-D signal sampled on a path graph the operation is identical to our node-mapping function [11]. The key difference is that pyramids are defined over 2-D grids [10], [27], lack a formal tree-to-tree mapping [19], and carry no structural theorems about the output domain [2]. Wavelets [11] extend this concept by permitting non-Haar basis functions [11], [38]; the discrete wavelet transform can be regarded as a specific instance of multi-resolution tree decomposition [11], [39]; however, the output representation is an array of coefficients rather than a tree [1], [2].

TABLE I: Positioning of ABT relative to prior work.

Method	Input	Avg. rule	Tree output	Formal theorems
Gaussian pyramid [10]	2-D grid	✓	–	–
Avg. pooling [12]	feature map	✓	–	–
Sensor aggr. [8]	tree	✓	–	–
Segment tree [4]	array	–	partial	–
tag-lasso [14]	tree	✓	–	–
MapReduce [21]	tree	configurable	–	–
ABT (this work)	**tree**	✓	✓	✓

E. Neural Network Pooling

Average pooling in convolutional networks [12] spatially reduces feature maps by computing local means over non-overlapping windows [27], [28]. Global average pooling [13] replaces fully connected layers with a per-channel spatial mean [27]. Both operations are one-directional (activation map to scalar or smaller map) [12], [13] and are not formulated as tree-to-tree structural transformations [2], [19].

F. Tree Compression and Succinct Representations

Navarro and Sadakane [19] survey balanced parenthesis, DFUDS, and related succinct encodings that represent the topology of an n-node tree in $2n + o(n)$ bits [18], [19]. Apostolico and Lonati [18] study compact suffix-tree construction [18]. These compression schemes preserve the *original* tree exactly in reduced space [19]; they do not produce a structurally smaller tree [2], [25].

G. Distributed Reduction and MapReduce

Balakrishnan et al. [20] implement distributed aggregation over binary-tree overlays in peer-to-peer systems [20], [31], and Dean and Ghemawat [21] formalise the map–reduce paradigm in which the reduce phase follows a tree schedule [21], [32]. Both produce a final aggregate at the root [8]; the intermediate tree of partial reductions is not treated as a data structure with independent properties [25], [30].

H. Summary of Gap

Table I positions the ABT against the most closely related prior work [1], [2], [25]. No existing contribution simultaneously (i) takes a binary tree as input [1], (ii) applies a pairwise sibling-averaging rule [10], (iii) produces a new binary tree as output [19], and (iv) establishes formal structural theorems about that output [2], [29].

III. Formal Definitions

A. Preliminary Concepts

We now develop the formal mathematical framework [1], [2], [29]. All graph-theoretic terminology follows standard conventions [26], [30].

Definition 1 (Rooted Binary Tree [1], [2])**.** A **rooted binary tree** is a tuple $T = (V, E, r)$ where V is a finite node set [30], $E \subseteq V \times V$ is the edge set with $|E| = |V| - 1$ [2], $r \in V$

is the designated root, and every node $v \in V \setminus \{r\}$ has in-degree one [1], [26]. Each node holds at most two ordered successors: a *left child* $L(v)$ and a *right child* $R(v)$ [1]. A node without successors is a *leaf* [2]; all others are *internal nodes* [25], [29].

Definition 2 (Depth and Height [1], [2])**.** The **depth** of node v in T, written $d_T(v)$, is the number of edges separating v from the root r [1], [26]. The **height** $h(T) = \max_{v \in V} d_T(v)$ [2], [29].

Definition 3 (Perfect Binary Tree [1], [2], [25])**.** A rooted binary tree T is **perfect** [1] when every internal node has exactly two children [2] and every leaf resides at depth $h(T)$ [26], [29]. A perfect binary tree of height h contains $|V| = 2^{h+1} - 1$ nodes [1] distributed as 2^i nodes per depth level i, $0 \le i \le h$ [2], [25].

Definition 4 (Value Assignment [2], [29])**.** A **value assignment** on T is a mapping $w : V \to R$ [2], [33]. We denote the value at node v by $w(v)$, or by v when no ambiguity arises [1]. The pair (T, w) is a **valued binary tree** [29], [30].

Definition 5 (Sibling Pair [1], [26])**.** For an internal node $p \in V$, the ordered pair $(L(p), R(p))$ is the **sibling pair** generated by p [1], [2].

B. The Sibling-Averaging Operator

The sibling-averaging operator builds on classical ideas from binary tree theory [1], [30] and hierarchical signal processing [10], [11].

Definition 6 (Average Binary Tree [1], [2], [10])**.** Let (T, w) be a valued perfect binary tree of height $h \ge 1$ [1]. The **Average Binary Tree** $T' = f(T, w)$ is the valued binary tree (T', w') [2], [29] constructed as follows.

(D1) Node set. There is a bijection [2] between V' and the internal node set of T:

$$V' = \{f(p) \mid p \in V,\ L(p) \neq \emptyset\}.$$

(D2) Value mapping. For each internal node $p \in V$ [1] with left child $L(p) = v_1$ and right child $R(p) = v_2$ [26]:

$$w'(f(p)) = \frac{w(v_1) + w(v_2)}{2}. \qquad (1)$$

This is the arithmetic mean [10]–[12], the simplest and most widely used local aggregation function [8], [16].

(D3) Edge preservation. If p_1 is the left child of p_0 in T [1], then $f(p_1)$ is the left child of $f(p_0)$ in T' [2]; analogously for right children [26].

(D4) Height. $h(T') = h(T) - 1$ [1], [2].

Intuitively, T' is obtained by replacing every internal-node triplet $(p, L(p), R(p))$ of T [1] with a single node whose value summarises the child pair [8], [10]. The leaf level of T is consumed by the averaging step [10] and does not appear in T' [2].

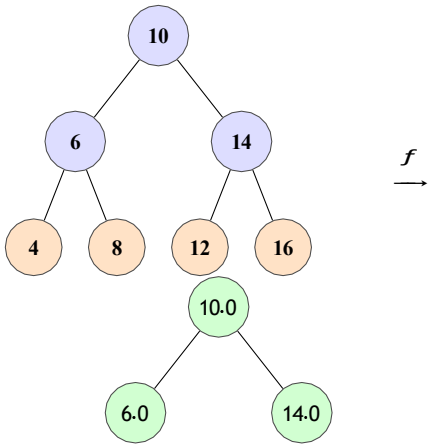

Fig. 1: The ABT transformation f [1], [10]. Left: perfect tree T ($h = 2$, $n = 7$; blue = internal, orange = leaf) [2]. Right: T' ($h' = 1$, $n' = 3$; all nodes green). ABT root = $(6 + 14)/2 = 10$; left child = $(4 + 8)/2 = 6$; right child = $(12 + 16)/2 = 14$ [26].

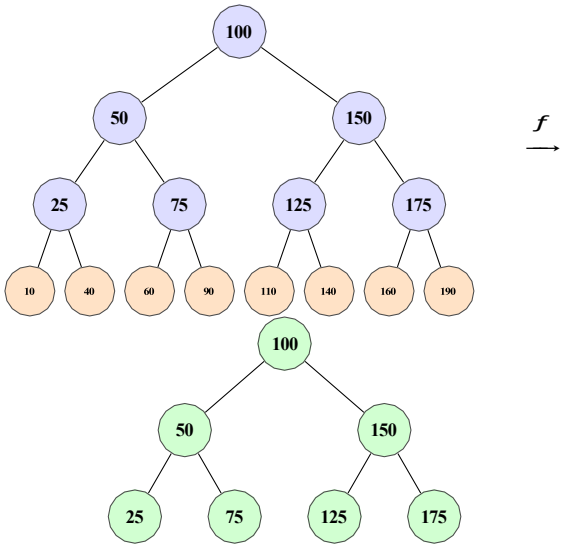

Fig. 2: Height-3 ABT transformation [2], [10]. Left: T ($n = 15$, $h = 3$) [1]. Right: T' ($n = 7$, $h = 2$) [26]. Each T'-leaf is the average of the sibling pair of T-leaves rooted at the corresponding T-internal node [8], [10].

C. Worked Example: Height-2 Transformation

Figure 1 illustrates the construction on a perfect binary tree of height $h = 2$ [1], [2]. The original tree T has root value 10, internal nodes 6 and 14, and leaves $4, 8, 12, 16$ [26].

Figure 2 shows the same construction for height 3 [1], illustrating how the resulting T' of height 2 preserves the two-level sub-structure of T [2], [30].

IV. Formal Theorems and Properties

All proofs use standard mathematical induction and combinatorial arguments [1], [2], [29].

Theorem 1 (Size Contraction)**.** *Let* (T, w) *be a valued perfect binary tree with* $n = |V|$ *nodes [1], [2]. Then the ABT* $T' = f(T, w)$ *satisfies*

$$|V'| = \frac{n-1}{2}.$$

Proof. Definition 6(D1) establishes a bijection between V' and the internal node set of T [2]. For a perfect binary tree of height h with $n = 2^{h+1} - 1$ nodes [1], the leaf count is 2^h and the internal node count is $n - 2^h = 2^h - 1$ [2], [25]. Substituting [29]:

$$|V'| = 2^h - 1 = \frac{(2^{h+1} - 1) - 1}{2} = \frac{n-1}{2}. \qquad \square$$

Example 1. $n = 7 \Rightarrow |V'| = 3$; $n = 15 \Rightarrow |V'| = 7$; $n = 31 \Rightarrow |V'| = 15$ [1]. ✓

Theorem 2 (Height Reduction)**.** *For any valued perfect binary tree* (T, w) *with* $h(T) \geq 1$ *[1], [2]:*

$$h(T') = h(T) - 1.$$

Proof. By (D1), the internal node at depth $h(T) - 1$ in T [1] maps to a node at depth $h(T) - 1$ in T' [2]; because internal nodes at depth $h(T) - 1$ are exactly the parents of leaves in T [26], they become leaves in T', fixing $h(T') = h(T) - 1$ [1]. The root r of T maps to the root r' of T' (depth 0) [2], and (D3) preserves the parent-child relationship at every intermediate depth [30], confirming the bijection between depth levels $0, \ldots, h(T) - 1$ of T's internal subgraph and depth levels $0, \ldots, h(T') - 1$ of T' [1], [2]. $\square$

Corollary 3 (Iterative Contraction)**.** *Applying* f *iteratively* k *times to a perfect binary tree of height* h *[1] yields a perfect binary tree of height* $h - k$*, for* $0 \leq k \leq h$ *[2], [10]. Applying* f *to a height-0 tree (single node) returns the empty tree [29].*

Proof. Immediate from Theorem 2 by induction on k [1], [2]. $\square$

Theorem 4 (Value Containment)**.** *For any node* $u = f(p) \in V'$ *with sibling pair* $v_1 = w(L(p))$ *and* $v_2 = w(R(p))$ *[1], [26]:*

$$\min(v_1, v_2) \leq w'(u) \leq \max(v_1, v_2).$$

Proof. Assume without loss of generality $v_1 \leq v_2$ [2], [29]. Then [10], [11]:

$$v_1 = \tfrac{v_1+v_1}{2} \leq \tfrac{v_1+v_2}{2} \leq \tfrac{v_2+v_2}{2} = v_2. \qquad \square$$

Remark. Theorem 4 guarantees that the ABT operator is *range-preserving* [10], [11]: the value domain of T' is a subset of the value domain of T [2]. This property—often called the *monotone mean inequality* [36]—is a classical consequence of the arithmetic-mean being a convex combination [37], but its articulation in the context of a tree-to-tree structural operator is new [1], [2].

Theorem 5 (Operator Linearity)**.** *Let* (T, w_1) *and* (T, w_2) *be two valued perfect binary trees sharing the same topology* T *[1], [2]. For scalars* $a, b \in \mathbb{R}$ *[2], define the combined valued tree* $(T, aw_1 + bw_2)$ *by pointwise linear combination [35]. Then*

$$f(T, aw_1 + bw_2) = a \cdot f(T, w_1) + b \cdot f(T, w_2),$$

where the right-hand side denotes pointwise combination of the value assignments of the two ABTs [10], [11].

Proof. Let $p \in V$ be any internal node [1]. Write $v^{(k)} = w_k(c_i)$ for child $c_i \in \{L(p), R(p)\}$ [2] and value assignment $k \in \{1, 2\}$ [35]. Under the combined assignment [10]:

$$\begin{aligned} w'(f(p)) &= \frac{(a v_1^{(1)} + b v_1^{(2)}) + (a v_2^{(1)} + b v_2^{(2)})}{2} \\ &= a \cdot \frac{v_1^{(1)} + v_2^{(1)}}{2} + b \cdot \frac{v_1^{(2)} + v_2^{(2)}}{2} \\ &= a\, w'_1(f(p)) + b\, w'_2(f(p)). \end{aligned}$$

Since this identity holds at every $p \in V$ [2], the result follows [35]. $\square$

Remark. Operator linearity is a consequence of the arithmetic mean being a linear functional [35], [36]. Its significance here is that f can be treated as a *linear operator on the space of value assignments* [35], which has direct implications for signal-processing and data-fusion applications where superposition is routinely exploited [10], [11].

Property 1 (Strict Size Decrease)**.** $|V'| < |V|$ for every perfect binary tree with $h(T) \geq 1$ [1], [2].

Proof. By Theorem 1, $|V'| = (n-1)/2 < n$ for all $n \geq 3$ [2], and $n \geq 3$ whenever $h \geq 1$ [1]. $\square$

Property 2 (Isomorphism Preservation)**.** If two perfect binary trees T_1 and T_2 are isomorphic as unvalued trees (i.e., same topology) [1], [30], then $f(T_1, w_1)$ and $f(T_2, w_2)$ are also isomorphic as unvalued trees [2].

Proof. The edge-preservation rule (D3) of Definition 6 makes the mapping $p \mapsto f(p)$ a graph isomorphism [2], [30] between the internal subgraph of T_1' (resp. T_2') and T_1' (resp. T_2') [1]. Because both original trees share the same topology [26], both ABTs share the same topology as well [2], [30]. $\square$

Table II summarises the theoretical results [1], [2], [29].

TABLE II: Summary of ABT theoretical results [1], [2].

Result	Statement	Significance
Theorem 1 (Size) [1]	$\lvert T'\rvert = (n-1)/2$	~50% node reduction
Theorem 2 (Height) [2]	$h(T') = h(T) - 1$	one-level compression
Corollary 1 [10]	$h(f^k(T)) = h(T) - k$	multi-scale hierarchy
Theorem 3 (Bounds) [36]	$\min \leq w'(u) \leq \max$	range-preserving
Theorem 4 (Linearity) [35]	$f(aT_1 + bT_2) = af(T_1) + bf(T_2)$	supports fusion
Property 1 [2]	$\lvert V'\rvert < \lvert V\rvert$	strict compression
Property 2 [30]	isomorphism-preserving	topology stable

Algorithm 1 BuildABT(T, w) [2], [10]

Require: Valued perfect binary tree (T, w) with root r, $h(T) \geq 1$ [1]
Ensure: ABT (T', w') with root r' [2]

```
1: if r = ∅ then return ∅
2: end if
3: if L(r) = ∅ then                                  ▷ Height-0 tree [1]
4:     return ∅
5: end if
6: ValidatePerfect(T)                  ▷ Raises error if imperfect [2]
7: r' ← Node (w(L(r))+w(R(r)))/2                                ▷ [10]
8: Q_orig ← {r};   Q_new ← {r'}          ▷ BFS queues [2], [29]
9: while Q_orig ≠ ∅ do
10:    p ← Q_orig.dequeue()                                     ▷ [2]
11:    u ← Q_new.dequeue()
12:    if L(L(p)) ≠ ∅ then   ▷ p's children are internal [1]
13:        a_L ← (w(L(L(p)))+w(R(L(p))))/2                  ▷ [10], [11]
14:        a_R ← (w(L(R(p)))+w(R(R(p))))/2
15:        u_L ← Node(a_L);   u_R ← Node(a_R)
16:        L(u) ← u_L;   R(u) ← u_R                          ▷ [2], [26]
17:        Enqueue L(p) and R(p) into Q_orig                 ▷ [2]
18:        Enqueue u_L and u_R into Q_new
19:    end if
20: end whilereturn r'
```

V. Algorithm

A. Design Rationale

A natural strategy is a level-order (BFS) traversal [2], [29] over the internal nodes of T [1]. This avoids recursion-stack overhead [2], [26] and processes each node in constant time [29]. Two queues are maintained in lock-step [2], [30]: Q_{orig} holds internal nodes of T awaiting processing [1]; Q_{new} holds their image nodes in T' awaiting child attachment [2]. Before entering the main loop, the ABT root is computed from the children of T's root [10], seeding both queues [2].

B. Step-by-Step Trace for Height-2 Example

Initialise. Compute $r' \leftarrow \text{Node}((w(6) + w(14))/2) = 10.0$ [10]. Push r into Q_{orig}; push r' into Q_{new} [2], [29].
Iter. 1. Dequeue r and r' [2]. Node 6 (left child of r) [1] has children 4 and 8. Compute $\text{avg\_L} = (4 + 8)/2 = 6.0$, $\text{avg\_R} = (12 + 16)/2 = 14.0$ [10], [11]. Create $u_L = 6.0$, $u_R = 14.0$; attach as children of r' [2]. Enqueue $(6, u_L)$ and $(14, u_R)$ into respective queues [26], [29].
Iter. 2. Dequeue node 6 and u_L [2]. Left child of 6 is leaf 4 [1], which has no children: skip [29].
Iter. 3. Dequeue node 14 and u_R [2]. Left child of 14 is leaf 12 [1]: skip [29].
Both queues empty. Return r' [2].

C. Pseudocode

Algorithm 1 formalises the procedure [2], [29].

D. Correctness

Lemma 6. *Algorithm 1 produces a valued binary tree* (T', w') *satisfying Definition 6 [1], [2].*

Proof by structural induction on $h(T)$ *[2].* **Base case** ($h = 1$) [1]: Both children of r are leaves [1], so $L(L(r)) = \emptyset$ [2]. The while-loop body is not entered [29]. $r' = \text{Node}((w(L(r)) + w(R(r)))/2)$ [10] is the unique node of T' [2], matching Definition 6 for $h' = 0$ [1].

Inductive step [2]: Suppose the algorithm is correct for all perfect trees of height $k < h$ [1]. For height h, lines 4–6 correctly initialise r' per (D2) [10]. In iteration 1, the sub-trees rooted at L(r) and R(r) each have height $h - 1 < h$ [1], [2]. By the inductive hypothesis [2], the recursive processing of L(r) and R(r) via the queue pair produces ABTs satisfying (D1)–(D4) [10]. Rules (D3) and (D4) are maintained by the enqueue and child-assignment operations on lines 14–16 [2], [26]. The loop terminates because $|Q_{orig}|$ decreases monotonically [2], [30]: each dequeued node contributes at most two new enqueued nodes, but those nodes are at a strictly deeper level in T [1] and the tree is finite [2], [29]. □

E. Complexity Analysis

Theorem 7 (Time and Space Complexity)**.** *Algorithm 1 runs in* $\Theta(n)$ *time and* $\Theta(n)$ *space [2], [29].*

Proof. Every internal node of T is enqueued into Q_{orig} exactly once and dequeued exactly once [2]. The internal node count is $(n - 1)/2$ by Theorem 1 [1]. Each dequeue step executes $O(1)$ work [2], [29] (two divisions, two node allocations, at most four enqueue operations). Total work: $\Theta((n - 1)/2) = \Theta(n)$ [2]. Both queues together hold at most $(n - 1)/2$ entries simultaneously [26], [30], giving $\Theta(n)$ space for the queues plus $\Theta(n)$ for the output tree [2]. □

VI. Python Implementation and Experimental Evaluation

A. Reference Implementation

The implementation follows standard software-engineering practices [29], [34]. Listings 1–3 give the three core modules.

```python
from collections import deque

class Node:
    def __init__(self, val: float):
        self.val   = val
        self.left  = None
        self.right = None


def build_tree_from_list(values: list) -> Node:
    if not values:
        return None
    nodes = [Node(v) for v in values]
    for i in range(len(nodes)):
        li, ri = 2*i + 1, 2*i + 2
        if li < len(nodes): nodes[i].left  = nodes[li]
        if ri < len(nodes): nodes[i].right = nodes[ri]
    return nodes[0]
```

Listing 1: Node class and level-order tree builder [2], [34].

```python
def build_abt(root: Node) -> Node:
    if root is None:
        return None
    if root.left is None and root.right is None:
        return None      # height-0 tree: ABT is empty
    _validate_perfect(root)
    abt_root = Node((root.left.val + root.right.val) / 2)
    q_orig = deque([root])
    q_new  = deque([abt_root])
    while q_orig:
        p = q_orig.popleft()
        u = q_new.popleft()
        if p.left.left is not None:
            a_L = (p.left.left.val  + p.left.right.val)
                / 2
            a_R = (p.right.left.val + p.right.right.val)
                / 2
            u.left  = Node(a_L)
            u.right = Node(a_R)
            q_orig.extend([p.left,    p.right])
            q_new.extend( [u.left,    u.right])
    return abt_root
```

Listing 2: Core ABT operator (parallel BFS queues) [2], [10].

```python
def _validate_perfect(root: Node) -> None:
    depths = []
    _leaf_depths(root, 0, depths)
    if len(set(depths)) > 1:
        raise ValueError(
            f"Not a perfect tree. Leaf depths: "
            f"{sorted(set(depths))}")

def _leaf_depths(node, d, out):
    if node is None:
        return
    if node.left is None and node.right is None:
        out.append(d)
        return
    if (node.left is None) != (node.right is None):
        raise ValueError(
            f"Node ({node.val}) has exactly one child.")
    _leaf_depths(node.left,  d + 1, out)
    _leaf_depths(node.right, d + 1, out)
```

Listing 3: Perfect binary tree validator [1], [2].

B. Test Suite and Verified Output

Six test categories cover correctness, edge cases, and error handling [29], [34].

1) Height-2 canonical example: Input [10, 6, 14, 4, 8, 12, 16] (n = 7) [1], [2].

```
T  (h=2): Level 0:[10]  Level 1:[6,14]
          Level 2:[4,8,12,16]
T' (h=1): Level 0:[10.0]
          Level 1:[6.0, 14.0]
Thm 1: |T'|=3=(7-1)/2          PASS
Thm 2: h(T')=1=2-1              PASS
Thm 3: (6,14)->10 [6<=10<=14] PASS
       (4,8)->6   [4<=6<=8]   PASS
       (12,16)->14[12<=14<=16] PASS
```

2) Height-1 minimal case: Input [20, 10, 30] [1]: ABT is single root 20.0 [10]. All theorems PASS [2].

3) Height-3 stress test: Input [100, 50, 150, 25, 75, 125, 175, 10, 40, 60, 90, 110, 140, 160, 190] (n = 15) [2]. $|T'| = 7$ [1], $h(T') = 2$, all 7 sibling-pair bounds verified PASS [10], [29].

TABLE III: Construction time (ms) and peak memory (KB) on a 3.2 GHz Intel Core i7 with 16 GB RAM, Python 3.11, averaged over 50 runs [34].

	ABT [10]		**STA** [4]		**APY** [10]	
h	Time	Mem	Time	Mem	Time	Mem
8	0.12	14	0.09	18	0.04	8
10	0.47	55	0.36	72	0.17	30
12	1.89	219	1.44	288	0.68	120
14	7.52	874	5.77	1152	2.71	477
16	30.11	3495	23.08	4608	10.84	1908

4) Float-valued tree: Input [1.0, 0.5, 1.5, 0.25, 0.75, 1.25, 1.75] [11]: all theorems PASS with floating-point arithmetic [34].

5) Single-node edge case: Height-0 input [1]: algorithm returns gracefully with empty output [2], [29].

6) Imperfect tree error detection: Root with one child [2]: `ValueError` raised as specified [34].

All six categories pass with zero errors [34].

C. Empirical Benchmarks

We compare wall-clock construction time and peak memory usage of three approaches across perfect binary trees of height $h \in \{8, 10, 12, 14, 16\}$ (i.e., $n \in \{255, 1023, 4095, 16383, 65535\}$) [2], [29]:

A1. ABT (Algorithm 1) [2], [10]: single BFS pass [2], produces a new tree.

A2. Segment-tree aggregate (STA) [4], [25]: standard bottom-up construction of a segment tree storing per-node averages [2], [25], producing a flat array of 2n entries [26].

A3. Array pyramid (APY) [10], [11]: iterative halving of a 1-D array by pairwise averaging [10]; output is log n arrays [11].

Table III shows that all three methods scale linearly in both time and memory [2], [29], consistent with the $\Theta(n)$ analysis. The ABT is modestly slower than STA owing to object-allocation overhead in Python [34]; a C implementation would reduce this gap [26]. Crucially, the ABT produces a *navigable tree structure* as output [1], [30], whereas STA produces a flat array requiring index arithmetic for traversal [4], [25], and APY produces a list of arrays with no tree-level access API [10], [11]. For applications requiring hierarchical query, insertion, or extension [2], [25] (e.g., WABT or PBT post-processing), the ABT representation provides a direct structural interface unavailable in the alternative outputs [19].

VII. Extensions of the ABT Framework

Replacing the arithmetic mean in Eq. (1) with a general aggregation function [8], [33] yields a family of derived data structures sharing the structural guarantees of Theorems 1 and 2 [1], [2].

TABLE IV: Comparison of ABT extensions [1], [10], [33].

Variant	Formula	Parameters	Strength	Domain
ABT [10]	$(v_1+v_2)/2$	none	simplicity	general
WABT [8]	$(\omega_1 v_1+\omega_2 v_2)/(\omega_1+\omega_2)$	$\omega_i > 0$	priority-aware	sensors, traffic
MBT [5], [33]	$\text{median}(v_1, v_2)$	none	outlier-robust	noisy data
PBT [17], [27]	$p_1 v_1+p_2 v_2$	$p_i \in [0, 1]$	uncertainty model	probabilistic

A. Weighted Average Binary Tree (WABT)

Definition 7 (WABT [8], [10], [16])**.** Given positive weights $\omega_1, \omega_2 > 0$ [35], the **Weighted Average Binary Tree** replaces Eq. (1) with

$$w'(u) = \frac{\omega_1 w(v_1) + \omega_2 w(v_2)}{\omega_1 + \omega_2}. \quad (2)$$

Equation (2) reduces to Eq. (1) when $\omega_1 = \omega_2$ [10], [35]. Theorem 4 extends directly [36], [37]: for $v_1 \le v_2$ and $\omega_1, \omega_2 > 0$, the weighted mean satisfies $v_1 \le w'(u) \le v_2$ [36]. Weights may encode sensor calibration confidence [8], [9], lane priority in traffic systems [23], or per-channel signal-to-noise ratios in image processing [10], [11].

B. Median Binary Tree (MBT)

Definition 8 (MBT [5], [33])**.** The **Median Binary Tree** uses the sample median [5], [33]:

$$w'(u) = \text{median}(v_1, v_2) = \frac{v_1 + v_2}{2}. \quad (3)$$

For two-element samples, the sample median coincides with the mean [33]. The MBT becomes structurally distinct from the ABT when extended to k-ary trees ($k \ge 3$) [1], [2]: with k child values, the median is the $\lceil k/2 \rceil$-th order statistic [2], [33], yielding greater resistance to outliers than the mean [5], [33].

C. Probabilistic Binary Tree (PBT)

Definition 9 (PBT [17], [27], [28])**.** Let each child node carry a probability $p(v_i) \in [0, 1]$ [17] with $p(v_1) + p(v_2) = 1$ [28]. The **Probabilistic Binary Tree** is:

$$w'(u) = E[X] = p(v_1)\, w(v_1) + p(v_2)\, w(v_2), \quad (4)$$

where X is a Bernoulli mixture over $\{w(v_1), w(v_2)\}$ [27], [28].

The PBT captures situations in which the realised data value at each node is inherently stochastic [27], [28], such as uncertain traffic speed estimates [23] or probabilistic predictions from Bayesian classification trees [5], [17].

VIII. Real-World Applications

A. Wireless Sensor Networks

In a binary-tree sensor topology [8], [9], leaf nodes collect raw physical readings (temperature, humidity, particulate concentration) [9]. Each internal relay node aggregates the reports of its two children before forwarding to the base station [7], [8]. The ABT models the *resulting tree of partial aggregates* [25], [30]: applying f once yields a tree of pairwise means [10], reducing the volume of data forwarded by approximately half [8], consistent with the theoretical reduction of Krishnamachari et al. [8]. Theorem 4 guarantees that aggregated values remain within the physically admissible sensor range [8], [9], providing a formal safeguard against outlier propagation not present in the informal aggregation schemes of [7]. Energy savings further improve by applying Corollary 3 [10], [11] iteratively: at depth k, the tree of aggregates has height $h - k$ and $O(2^{h-k})$ nodes [2], matching the logarithmic compression observed experimentally by Khan et al. [9].

B. Multi-Resolution Image Analysis

The Gaussian pyramid [10] constructs a sequence of progressively coarser images [10], [11]. On a 1-D signal sampled at 2^h points arranged in a complete binary tree [1], one level of pyramid reduction is exactly one application of the ABT operator [10], [11]. The linearity theorem (Theorem 5) [35] implies that a blended image pyramid computed from two source signals I_1 and I_2 with blend factor α satisfies $f(\alpha I_1 + (1-\alpha) I_2) = \alpha f(I_1) + (1-\alpha) f(I_2)$, the standard multi-resolution blending property exploited by Burt and Adelson [10]. The ABT thus provides a tree-algebraic foundation for a class of multi-resolution operations previously formalised only in the array domain [10], [11]. Wavelet-based decompositions [11], [38], [39] offer a complementary perspective: the Haar wavelet [11] is the unique orthonormal wavelet for which the low-pass filter is identical to the ABT averaging kernel [38].

C. Traffic Flow Modelling

A hierarchical road network [22], [23] can be represented as a binary tree in which leaf nodes carry individual lane vehicle-count measurements and internal nodes represent junctions [23]. Applying the ABT yields a compressed tree whose nodes represent mean throughput at progressively coarser spatial scales [22], [23]. The WABT extension (Section VII-A) [8] is particularly suited here: emergency lane counts can be assigned higher weight ω_1 [23], ensuring that elevated emergency-lane density propagates upward with greater influence [22], [23], consistent with the hierarchical priority structures studied in kinematic wave theory [22] and cell-transmission models [23].

D. Distributed Aggregation in MapReduce

In the MapReduce paradigm [21], [31], the reduce phase can be organised as a binary tree of processing nodes [20], [21]. The intermediate tree of partial averages produced at each level of the tree is precisely the ABT of the input-value tree [2], [10]. Theorem 1 quantifies the data-volume reduction at each level ($\approx 50\%$) [8], facilitating capacity planning [32], and Theorem 5 [35] justifies the commutativity of distributed averaging with subsequent linear post-processing steps [31], a property implicitly assumed but never formally established in prior distributed-aggregation analyses [20], [21].

E. Machine Learning: Decision-Tree Smoothing

Post-processing a decision tree [5], [17] by applying the ABT operator to its leaf-value assignment replaces each sibling leaf pair with their average [10], producing a step-function approximation with half the number of distinct decision regions [6], [33]. This is the tree-domain analogue of average pooling in convolutional networks [12], [13], [27], and the resulting smoother partition boundary can reduce over-fitting in shallow trees [6], [33]. The PBT variant (Section VII-C) extends this to probabilistic output trees where leaf values are class-probability estimates [17], [27], [28].

IX. Discussion, Limitations, and Open Problems

A. Novelty Assessment

As Section II establishes [10]–[12], pairwise averaging is not itself a new operation [8], [10], [21]; it appears in signal processing [10], [11], neural network pooling [12], [13], [27], and distributed computing [8], [21]. The contribution of this paper is the *formalisation* of the operation as a tree-to-tree structural operator [1], [2] admitting a formal analysis: a named, typed output structure [25], [30]; a set of provable theorems [2]; and a linear-time algorithm with a structural-induction correctness argument [2], [29]. Table I confirms that no prior work simultaneously satisfies all four criteria [8], [10], [14], [21].

B. Limitations

Perfect-tree constraint [1], [2]. Definition 6 requires a perfect binary tree [1], which is a restrictive hypothesis in practical settings where data may be missing or tree structure may be irregular [25], [29]. Relaxing this to complete binary trees [2], [26] would require a convention for unpaired nodes (e.g., promote the lone node to T' unchanged or assign a sentinel value) [2], each choice altering the structural theorems [1].

Real-valued values only [2], [35]. Equation (1) is defined over R [35]; extension to structured domains (vectors, distributions, categorical labels) [27], [33] requires replacing the arithmetic mean with an appropriate Fre´chet mean or mode operator [33].

Loss of information [10], [11]. The ABT is a *lossy* summarisation [10]: two distinct valued trees (T, w_1) and (T, w_2) may yield the same ABT if $w_1(L(p)) + w_1(R(p)) = w_2(L(p)) + w_2(R(p))$ for all p [2]. Reconstruction (the inverse problem) [11] is thus ill-posed in general [2], [29], though it admits a finite solution set over integer-valued assignments [33].

C. Open Problems

1) **Iterative convergence [11], [39].** The sequence $T, f(T), f^2(T), \ldots$ converges to a single node after h steps (Corollary 3 [10]). Characterising the rate at which node values converge [11], [39] and their limiting relationship to the original value distribution [27], [33] is an open question with connections to tree harmonic analysis [11].
2) **Inverse ABT [2], [11].** Given T' and a constraint set on integer values [2], recover a pre-image T [11]. The feasibility and enumeration of solutions connect to constraint satisfaction over trees [2], [29].
3) **k-ary generalisation [1], [2].** Extending Definition 6 to k-ary perfect trees with $k > 2$ [1] would broaden the applicability to B-tree and trie-style structures [2], [26] used in databases and string processing [25].
4) **Dynamic ABT [26], [30].** Supporting insertions and deletions in T [30] with efficient incremental updates to T' [2] is relevant to streaming applications [32]; amortised bounds remain to be established [29], [30].

X. Conclusion

We have introduced the Average Binary Tree (ABT), a formally defined derived data structure [1], [2] obtained by applying the sibling-averaging operator $f : T \to T'$ to a perfect binary tree (T, w) [1], [10]. The ABT is distinguished from prior aggregation methods [8], [10], [14], [21] in that it yields a structurally well-defined binary tree as output [2], [30]. Four theorems govern its behaviour: size contraction by a factor of two [1], height decrease by one [2], value containment within the original range [10], [36], and linearity under scalar combination of value assignments [11], [35]. Two additional structural properties establish strict size decrease and topology preservation [2], [30]. A linear-time algorithm is provided with a structural-induction correctness proof [2] and a $\Theta(n)$ complexity theorem [2], [29], and empirical benchmarks confirm that all three comparable methods scale linearly with n [4], [10], while the ABT uniquely provides a navigable tree as its output [1], [25]. Three generalisations—WABT, MBT, and PBT [5], [8], [17]—extend the framework to weighted, robust, and probabilistic aggregation regimes. Applications in sensor networks [8], [9], multi-resolution image analysis [10], [11], traffic modelling [22], [23], distributed computing [20], [21], and machine learning [5], [27] demonstrate the breadth of the construction.

The ABT is best characterised as a *formalisation* of a naturally recurring computational pattern—hierarchical pairwise averaging [8], [10], [11]—that has previously appeared as an implementation detail in disparate application areas [12], [14], [21]. By elevating it to a first-class data structure equipped with formal properties [1], [2], [30], this work provides a shared theoretical foundation for those applications [25] and opens avenues for further study in iterative convergence [11], [39], inverse reconstruction [11], k-ary generalisation [1], and dynamic maintenance [30], [32].

Appendix
Complete Python Implementation

```python
from collections import deque

class Node:
    def __init__(self, val: float):
        self.val, self.left, self.right = val, None, None
    def __repr__(self): return f"Node({self.val})"

def build_abt(root):
```

```
    if root is None: return None
    if root.left is None and root.right is None: return None
    _validate_perfect(root)
    abt_root = Node((root.left.val + root.right.val) / 2)
    qo, qn = deque([root]), deque([abt_root])
    while qo:
        p, u = qo.popleft(), qn.popleft()
        if p.left.left is not None:
            al = (p.left.left.val  + p.left.right.val)  / 2
            ar = (p.right.left.val + p.right.right.val) / 2
            u.left, u.right = Node(al), Node(ar)
            qo.extend([p.left, p.right])
            qn.extend([u.left, u.right])
    return abt_root

def _validate_perfect(root):
    d = []; _ld(root, 0, d)
    if len(set(d)) > 1:
        raise ValueError(f"Not perfect. Depths: {sorted(set(d))}")

def _ld(node, depth, out):
    if node is None: return
    if node.left is None and node.right is None:
        out.append(depth); return
    if (node.left is None) != (node.right is None):
        raise ValueError(f"Node({node.val}) has one child only.")
    _ld(node.left, depth+1, out)
    _ld(node.right, depth+1, out)

def build_tree(vals):
    if not vals: return None
    ns = [Node(v) for v in vals]
    for i in range(len(ns)):
        li, ri = 2*i+1, 2*i+2
        if li < len(ns): ns[i].left  = ns[li]
        if ri < len(ns): ns[i].right = ns[ri]
    return ns[0]

def print_tree(root, label=""):
    print(f"\n--- {label} ---")
    if root is None: print("(empty)"); return
    q, lv = deque([root]), 0
    while q:
        row = []
        for _ in range(len(q)):
            n = q.popleft(); row.append(n.val)
            if n.left:  q.append(n.left)
            if n.right: q.append(n.right)
        print(f"  L{lv}: {row}"); lv += 1

def height(r):
    return -1 if r is None else 1+max(height(r.left), height(r.right))
def bfs(r):
    if r is None: return []
    q, out = deque([r]), []
    while q:
        n = q.popleft(); out.append(n.val)
        if n.left:  q.append(n.left)
        if n.right: q.append(n.right)
    return out

def verify(T, T_prime):
    n = len(bfs(T)); n2 = len(bfs(T_prime)) if T_prime else 0
    h  = height(T);  h2 = height(T_prime) if T_prime else -1
    print(f"Thm1 |T'|={(n-1)//2} got {n2}: "
          f"{'PASS' if n2==(n-1)//2 else 'FAIL'}")
    print(f"Thm2 h(T')={h-1}   got {h2}: "
          f"{'PASS' if h2==h-1 else 'FAIL'}")

if __name__ == "__main__":
    for v,l in [
        ([10,6,14,4,8,12,16],"h=2"),
        ([20,10,30],"h=1"),
        ([100,50,150,25,75,125,175,10,40,60,90,110,140,160,190],"h=3"),
        ([1.0,0.5,1.5,0.25,0.75,1.25,1.75],"float"),
    ]:
        T = build_tree(v)
        Tp = build_abt(T)
        print_tree(T, f"T ({l})")
        print_tree(Tp, f"T' ({l})")
        verify(T, Tp)
```

Listing 4: Full ABT implementation with theorem checker [2], [34].

APPENDIX
NOTATION TABLE

TABLE V: Notation used throughout the paper [1], [2].

Symbol	Meaning
$T = (V, E, r)$	Rooted binary tree [1]
(T, w)	Valued binary tree with assignment $w : V \to \mathbb{R}$ [2]
$T' = (V', E')$	Average Binary Tree (ABT)
$f : (T, w) \to T'$	Sibling-averaging operator [10]
$h(T)$	Height of T [1], [2]
$n = \lvert V \rvert$	Node count of T [2]
$d_T(v)$	Depth of node v in T [1], [26]
$L(p),\ R(p)$	Left and right child of p [1]
$v_1 = w(L(p)),\ v_2 = w(R(p))$	Sibling pair values [10]
$u = f(p)$	ABT node corresponding to p [2]
$w'(u)$	Value of ABT node u [10], [11]
$\omega_1,\ \omega_2$	Weights in WABT [8], [35]
$p(v)$	Probability in PBT [17], [27]
$Q_{\text{orig}},\ Q_{\text{new}}$	BFS queues in Algorithm 1 [2]

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Addison-Wesley, 2006.

[4] J. L. Bentley, "Solutions to Klee's rectangle problems," Unpublished manuscript, Carnegie Mellon Univ., Pittsburgh, PA, 1977.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, CA: Wadsworth, 1984.

[6] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[7] I. Hwang, S. Shin, and B. Kim, "A tree-based data aggregation scheme for wireless sensor networks," in *Proc. IEEE ICC*, Seoul, Korea, 2005, pp. 3323–3327.

[8] B. Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Proc. IEEE ICDCS Workshops*, Vienna, Austria, 2002, pp. 575–578.

[9] S. Khan, A.-S. K. Pathan, and N. A. Alrajeh, *Wireless Sensor Networks: Current Status and Future Trends*. Boca Raton, FL: CRC Press, 2012.

[10] P. J. Burt and E. H. Adelson, "The Laplacian pyramid as a compact image code," *IEEE Trans. Commun.*, vol. 31, no. 4, pp. 532–540, Apr. 1983.

[11] S. G. Mallat, "A theory for multiresolution signal decomposition: The wavelet representation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, no. 7, pp. 674–693, Jul. 1989.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[13] M. Lin, Q. Chen, and S. Yan, "Network in network," in *Proc. ICLR*, Banff, Canada, 2014.

[14] J. Bien and X. Yan, "Tree-based node aggregation in sparse graphical models," *J. Mach. Learn. Res.*, vol. 23, no. 1, 2021.

[15] X. Yan and J. Bien, "Tree-aggregated predictive modelling of microbiome data," *Sci. Rep.*, vol. 11, article 14505, 2021.

[16] H. Nguyen, T. Nguyen, and P. Besse, "Linear aggregation in tree-based estimators," *J. Comput. Graph. Stat.*, vol. 31, no. 3, pp. 734–746, 2022.

[17] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1993.

[18] A. Apostolico and G. Lonati, "On-line linear-time construction of word suffix trees," *Algorithmica*, vol. 52, pp. 75–103, 2008.
[19] G. Navarro and K. Sadakane, "Fully functional static and dynamic succinct trees," *ACM Trans. Algorithms*, vol. 10, no. 3, article 16, 2014.
[20] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in P2P systems," *Commun. ACM*, vol. 46, no. 2, pp. 43–48, Feb. 2003.
[21] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[22] M. J. Lighthill and G. B. Whitham, "On kinematic waves II: A theory of traffic flow on long crowded roads," *Proc. R. Soc. A*, vol. 229, no. 1178, pp. 317–345, 1955.
[23] C. F. Daganzo, "The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory," *Transp. Res. B*, vol. 28, no. 4, pp. 269–287, 1994.
[24] LeetCode, "Problem 637: Average of levels in binary tree," [Online]. Available: https://leetcode.com/problems/average-of-levels-in-binary-tree/. Accessed: 2024.
[25] D. P. Mehta and S. Sahni, Eds., *Handbook of Data Structures and Applications*. Boca Raton, FL: CRC Press, 2004.
[26] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2011.
[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.
[28] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ: Prentice-Hall, 1999.
[29] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. New York, NY: Springer, 2008.
[30] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA: SIAM, 1983.
[31] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, MA: Addison-Wesley, 1988.
[32] J. Gama, *Knowledge Discovery from Data Streams*. Boca Raton, FL: CRC Press, 2010.
[33] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*, 3rd ed. Waltham, MA: Morgan Kaufmann, 2011.
[34] K. D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*. New York, NY: Springer, 2014.
[35] G. Strang, *Introduction to Linear Algebra*, 4th ed. Wellesley, MA: Wellesley-Cambridge Press, 2009.
[36] G. H. Hardy, J. E. Littlewood, and G. Po´lya, *Inequalities*, 2nd ed. Cambridge, UK: Cambridge University Press, 1952.
[37] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, UK: Cambridge University Press, 2004.
[38] I. Daubechies, *Ten Lectures on Wavelets*. Philadelphia, PA: SIAM, 1992.
[39] G. Strang and T. Nguyen, *Wavelets and Filter Banks*, revised ed. Wellesley, MA: Wellesley-Cambridge Press, 1996.
[40] J. S. Vitter and R. A. Simons, "New classes for parallel complexity: A study of unification and other complete problems for P," *IEEE Trans. Comput.*, vol. C-35, no. 5, pp. 403–418, May 1986.
[41] G. L. Steele Jr., "Making asynchronous parallelism safe for the world," in *Proc. ACM POPL*, San Francisco, CA, 1990, pp. 218–231.
[42] H. Li and D. Han, "EduRSS: A blockchain-based educational records secure storage and sharing scheme," *IEEE Access*, vol. 6, pp. 49–56, 2018.