Bahnasy Tree: A Dynamic Sequence with SPF-Guided Splitting, Local Overflow Repair, and Lazy Range Operations

Mostafa Bahnasy

Abstract

We present the Bahnasy Tree, a dynamic multi-way tree representing a sequence and supporting index-based access, insertion, deletion, range queries, and lazy range updates. Each internal node stores the sizes of its child subtrees, enabling routing by lower-bounding prefix sums. To control fanout, the data structure uses a threshold parameter T and splits nodes using a branching factor derived from the smallest prime factor (SPF) of the node size, with fallback to binary splitting. To repair local overflow at leaf level, we regroup leaves when the leaf-parent degree exceeds cT for a fixed constant $c \geq 1$. A rebuild-after-K-splits policy is used to prevent long-term drift under adversarial edits. We prove correctness via invariants and provide complexity bounds in terms of the current height H, together with a typical post-rebuild specialization where $H \approx \log_T N$. We empirically evaluate the SumAdd instantiation (range add + range sum) against an implicit treap (fully dynamic workloads) and a lazy segment tree (fixed-length workloads).

1 Introduction

Dynamic sequence problems require maintaining an ordered list of elements under a mixture of operations: queries and updates on ranges, point updates, and structural edits (insertions and deletions). Well-known solutions include implicit treaps for fully dynamic sequences and segment trees for fixed-length arrays. This paper studies a simple multi-way alternative: the *Bahnasy Tree*. It routes by storing child subtree sizes and uses SPF-guided splitting to create bounded fanout, combined with local overflow repair and occasional rebuilding.

Contributions.

- A multi-way dynamic sequence representation with routing by prefix sums of subtree sizes.
- An SPF-guided branching heuristic for global construction and local overflow repair.
- A generic aggregate/lazy framework (sum/min/xor/or/and and corresponding lazies).
- Correctness arguments and complexity bounds in terms of current height H.
- Benchmarks for SumAdd (range add + range sum) versus implicit treap and lazy segment tree.

2 Related Work

Implicit treaps maintain sequence order using implicit keys and randomized priorities and support split/merge, making them a strong baseline for fully dynamic sequences with range aggregates and lazy updates. **Segment trees** provide efficient range query/update on fixed-length arrays but do not natively support insertion/deletion. **B-trees/B$^+$-trees** and **ropes** represent sequences with bounded fanout and are widely used in systems. Bahnasy Tree shares the multi-way partitioning idea, but uses SPF-guided branching and a lightweight local overflow repair based on regrouping leaves, plus a rebuild policy to mitigate adversarial drift.

3 Preliminaries

3.1 Sequence ADT

We represent a sequence $A[1..N]$ (1-indexed) supporting:

- PointSet(i, x): set $A[i] \leftarrow x$.
- RangeQuery(l, r): query an aggregate on $A[l..r]$.
- RangeApply(l, r, Δ): apply a lazy update to each element in $A[l..r]$.
- InsertAt(i, x): insert x so it becomes the new element at position i (shifting the old $i, i+1, \ldots$ right).
- EraseAt(i): remove the element at position i (shifting right part left).

3.2 Policy interface (aggregate + lazy)

A policy specifies: (i) an aggregate set $(\mathcal{A}, \oplus, e)$ (associative, identity e), (ii) a lazy set $(\mathcal{L}, \circ, \mathsf{id})$ (associative, identity id), and (iii) an apply function ApplyAgg$(\text{agg}, \text{lazy}, \ell)$ that updates a segment aggregate given its length ℓ, compatible with $\oplus$.

Definition 1 (SumAdd policy (evaluated))**.** *Elements are integers. Aggregate is segment sum:* $(\mathcal{A}, \oplus, e) = (\mathbb{Z}, +, 0)$. *Lazy update is an integer* Δ *meaning "add* Δ *to every element in the segment". Applying to a segment of length* ℓ *is* ApplyAgg$(s, \Delta, \ell) = s + \Delta \cdot \ell$. *Lazy composition is addition and identity is* 0.

Other supported policies. The same interface supports alternative aggregates and lazies (e.g., range add + range min, xor/xor, or/or, and/and). Our empirical evaluation focuses on SumAdd.

4 Bahnasy Tree

4.1 Node representation

A Bahnasy Tree is an ordered rooted tree. Each node v stores:

- $\mathsf{size}(v)$: number of leaves (sequence elements) in its subtree.
- $\mathsf{agg}(v)$: aggregate of values in its subtree.
- $\mathsf{lazy}(v)$: a pending lazy update for the entire subtree.
- ordered children $(c_1, \ldots, c_k)$.
- prefix sizes $p_0 = 0$ and $p_j = \sum_{t=1}^{j} \mathsf{size}(c_t)$.

A leaf is a node with no children and $\mathsf{size} = 1$. A *leaf-level parent* is a node whose children are leaves.

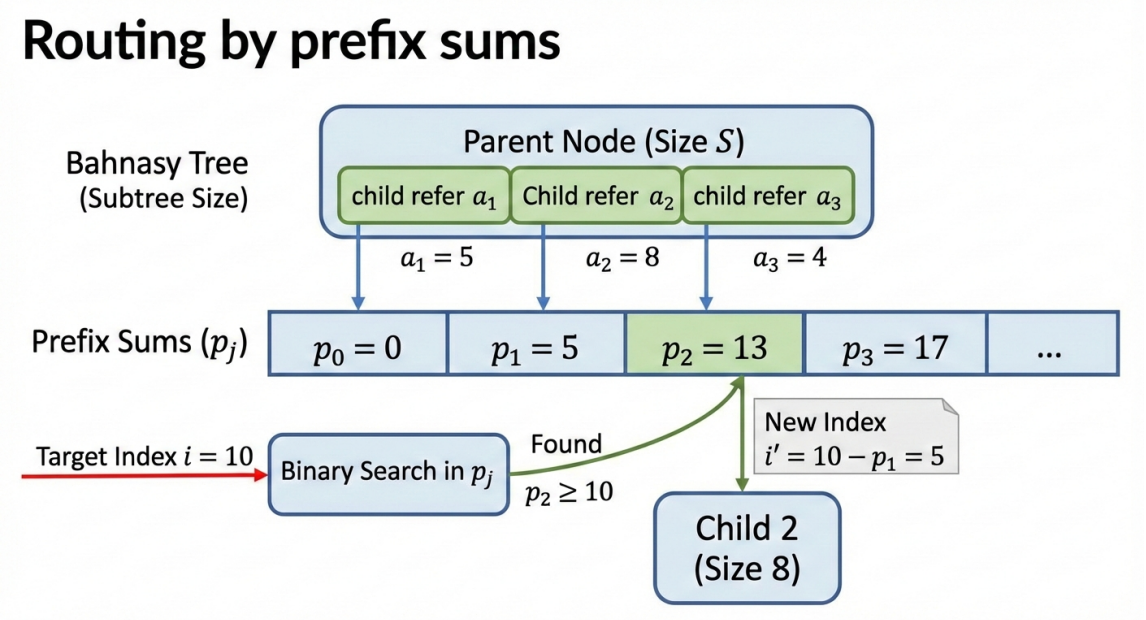

Figure 1: Routing by lower-bounding prefix sums of child subtree sizes.

4.2 Core invariants

Definition 2 (Structural invariants)**.** *For every internal node v with children $(c_1, \ldots, c_k)$:*

1. ***Order/contiguity.*** *Leaves under v appear in the sequence order and children partition a contiguous range.*
2. ***Size.*** $\mathsf{size}(v) = \sum_{j=1}^{k} \mathsf{size}(c_j)$*; leaves have size* 1*.*
3. ***Prefix.*** $p_j = \sum_{t=1}^{j} \mathsf{size}(c_t)$*.*

Definition 3 (Aggregate/lazy invariant)**.** $\mathsf{agg}(v)$ *equals the aggregate of the leaf values in v's subtree after applying all deferred updates along the path;* $\mathsf{lazy}(v)$ *encodes a deferred update to be applied to all leaves in the subtree.*

4.3 Routing by prefix sums

To locate index $i \in [1, \mathsf{size}(v)]$ at node v, find the smallest j with $p_j \geq i$, then recurse into c_j with local index $i' = i - p_{j-1}$ (Figure 1).

5 SPF-guided splitting

5.1 Branching factor

Let $\mathrm{SPF}(n)$ denote the smallest prime factor of n (and $\mathrm{SPF}(1) = 1$).

Definition 4 (Branching factor)**.** *Fix a threshold $T \geq 2$. For a node containing n elements, define the branching factor $S(n)$ by:*

- *if* $\mathrm{SPF}(n) \leq T$*, then set* $S(n) \leftarrow \mathrm{SPF}(n)$*;*
- *otherwise set* $S(n) \leftarrow 2$*.*

In particular, if n is prime and $n > T$, then $S(n) = 2$.

5.2 Global construction

To build a tree for n elements:

- if $n \leq T$, create n leaves as children;
- else set $S \leftarrow S(n)$ and partition n into S consecutive parts with sizes $g = \lfloor n/S \rfloor$ repeated $S - 1$ times and $g + r$ for the last part, where $r = n \bmod S$; recurse for each child.

After rebuild, every internal node has degree at most T.

6 Local overflow repair at leaf level

Definition 5 (Local split threshold)**.** *Let $c \geq 1$ be a fixed constant. A leaf-level parent v with d leaf-children is considered too wide when $d > cT$. When this happens, we perform a* local split *by inserting one intermediate layer and regrouping the leaves.*

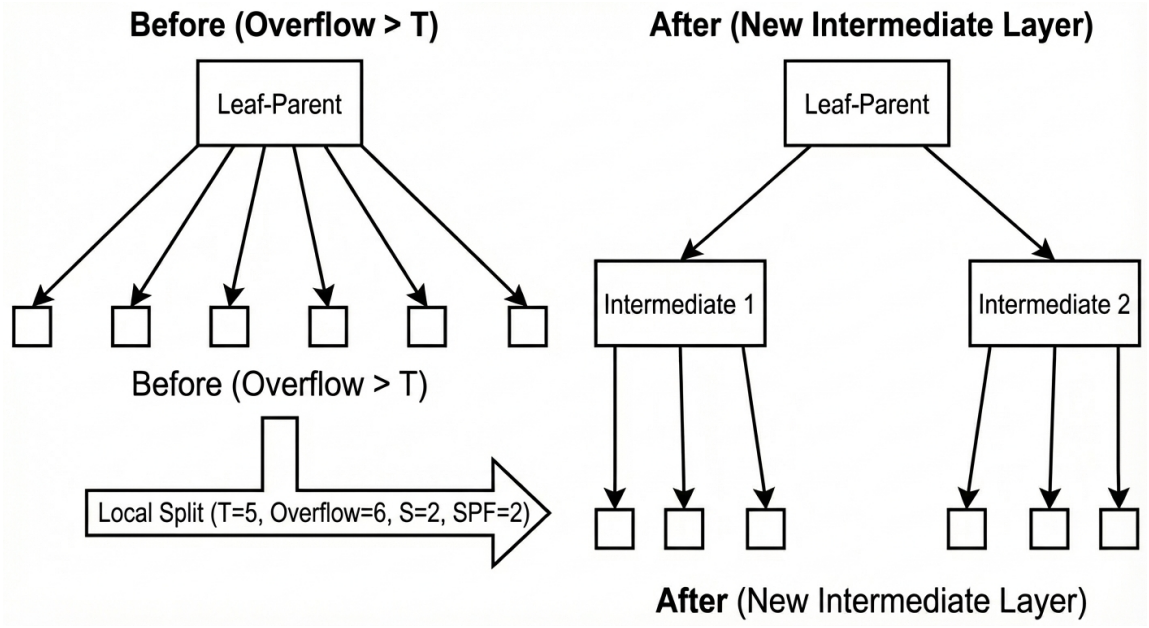

Figure 2: Local split at leaf level: regrouping leaves by inserting an intermediate layer.

7 Algorithms (SumAdd pseudocode)

In pseudocode, we write nodes with fields: size, agg, lazy, children array, and prefix array. We assume each node can mark its prefix array as "dirty" after edits and rebuild it on demand.

7.1 Lazy utilities: apply, push, pull

Algorithm 1 SumAdd utilities

procedure ApplyToNode(v, Δ)
    $\mathsf{agg}(v) \leftarrow \mathsf{agg}(v) + \Delta \cdot \mathsf{size}(v)$
    $\mathsf{lazy}(v) \leftarrow \mathsf{lazy}(v) + \Delta$
end procedure
procedure Push(v)
    if v has no children **then return**
    end if
    if $\mathsf{lazy}(v) = 0$ **then return**
    end if
    for all children c of v **do**
        ApplyToNode($c, \mathsf{lazy}(v)$)
    end for
    $\mathsf{lazy}(v) \leftarrow 0$
end procedure
procedure Pull(v)
    $\mathsf{agg}(v) \leftarrow 0$
    for all children c of v **do**
        $\mathsf{agg}(v) \leftarrow \mathsf{agg}(v) + \mathsf{agg}(c)$
    end for
end procedure

7.2 Prefix maintenance and child selection

Algorithm 2 Prefix rebuild and child selection

```
1: procedure RebuildPrefix(v)
2:     p_0 ← 0
3:     for j = 1 to degree(v) do
4:         p_j ← p_{j−1} + size(c_j)
5:     end for
6: end procedure
7: procedure ChooseChildByIndex(v, i)                ▷ i is 1-indexed within v
8:     ensure prefix sizes are up to date (rebuild if dirty)
9:     return smallest j such that p_j ≥ i           ▷ lower_bound on prefix sums
10: end procedure
```

7.3 Range query and range apply

Algorithm 3 RangeQuery(v, l, r) (SumAdd)

```
1: procedure RangeQuery(v, l, r)                     ▷ [l, r] are 1-indexed within v
2:     if v has no children then return 0
3:     end if
4:     if r < 1 or l > size(v) then return 0
5:     end if
6:     l ← max(l, 1); r ← min(r, size(v))
7:     if l > r then return 0
8:     end if
9:     if l = 1 and r = size(v) then return agg(v)
10:    end if
11:    Push(v)
12:    if v is leaf-level parent then
13:        ans ← 0
14:        for t = l to r do ans ← ans + agg(leaf child t)
15:        end for
16:        return ans
17:    end if
18:    j_L ← ChooseChildByIndex(v, l); j_R ← ChooseChildByIndex(v, r)
19:    let prefix sizes be (p_0, ..., p_k)
20:    if j_L = j_R then
21:        return RangeQuery(c_{j_L}, l − p_{j_L}, r − p_{j_L})
22:    end if
23:    ans ← RangeQuery(c_{j_L}, l − p_{j_L}, size(c_{j_L}))
24:    for j = j_L + 1 to j_R − 1 do
25:        ans ← ans + agg(c_j)
26:    end for
27:    ans ← ans+ RangeQuery(c_{j_R}, 1, r − p_{j_R})
28:    return ans
29: end procedure
```

Algorithm 4 RangeApply(v, l, r, Δ) (SumAdd)

```
1: procedure RangeApply(v, l, r, Δ)
2:     if v has no children then return
3:     end if
4:     if r < 1 or l > size(v) then return
5:     end if
6:     l ← max(l, 1); r ← min(r, size(v))
7:     if l > r then return
8:     end if
9:     if l = 1 and r = size(v) then
10:        ApplyToNode(v, Δ); return
11:    end if
12:    Push(v)
13:    if v is leaf-level parent then
14:        for t = l to r do
15:            agg(leaf child t) ← agg(leaf child t) + Δ
16:        end for
17:        Pull(v)
18:        return
19:    end if
20:    j_L ← ChooseChildByIndex(v, l); j_R ← ChooseChildByIndex(v, r)
21:    let prefix sizes be (p_0, ..., p_k)
22:    for j = j_L to j_R do
23:        L ← max(1, l − p_j)
24:        R ← min(size(c_j), r − p_j)
25:        if L ≤ R then RangeApply(c_j, L, R, Δ)
26:        end if
27:    end for
28:    Pull(v)
29: end procedure
```

7.4 Point set

Algorithm 5 PointSet(v, i, x) (SumAdd)

```
1: procedure PointSet(v, i, x)
2:     if v has no children or i < 1 or i > size(v) then return
3:     end if
4:     Push(v)
5:     if v is leaf-level parent then
6:         agg(leaf child i) ← x
7:         Pull(v)
8:         return
9:     end if
10:    j ← ChooseChildByIndex(v, i); let prefix be (p_0, ..., p_k)
11:    PointSet(c_j, i − p_j, x)
12:    Pull(v)
13: end procedure
```

7.5 Insert, erase, local split

Algorithm 6 InsertAt(v, i, x) (returns whether a local split occurred)

```
1: procedure InsertAt(v, i, x)
2:     clamp i into [1, size(v) + 1]
3:     Push(v)
4:     size(v) ← size(v) + 1
5:     agg(v) ← agg(v) + x
6:     if v is leaf-level parent then
7:         insert new leaf of value x into ordered child list at position i
8:         mark prefix of v dirty
9:         if degree(v) > cT then
10:            LocalSplit(v)
11:            return true
12:        end if
13:        return false
14:    end if
15:    j ← ChooseChildByIndex(v, i); let prefix be (p_0, ..., p_k)
16:    did ← InsertAt(c_j, i − p_j, x)
17:    mark prefix dirty; Pull(v)
18:    return did
19: end procedure
```

Algorithm 7 EraseAt(v, i)

```
1: procedure EraseAt(v, i)
2:     if v has no children or i < 1 or i > size(v) then return
3:     end if
4:     Push(v)
5:     if v is leaf-level parent then
6:         remove leaf child at position i (shift in vector)
7:         size(v) ← size(v) − 1
8:         Pull(v)
9:         mark prefix dirty; return
10:    end if
11:    j ← ChooseChildByIndex(v, i); let prefix be (p_0, ..., p_k)
12:    EraseAt(c_j, i − p_j)
13:    if c_j becomes empty, remove it from child list
14:    size(v) ← size(v) − 1
15:    mark prefix dirty; Pull(v)
16: end procedure
```

Algorithm 8 LocalSplit(v) for a leaf-level parent

```
1: procedure LocalSplit(v)
2:     let n ← degree(v)
3:     compute s ← S(n)
4:     g ← ⌊n/s⌋, r ← n mod s
5:     move old leaf children into an array old in order; clear v's children
6:     idx ← 1
7:     for t = 1 to s do
8:         cnt ← g; if t = s then cnt ← g + r
9:         create new intermediate node m with size 0 and agg 0
10:        for j = 1 to cnt do
11:            move old[idx] as a child of m; idx ← idx + 1
12:            update size(m) and agg(m) from moved leaf
13:        end for
14:        mark prefix of m dirty
15:        append m as a child of v
16:    end for
17:    recompute size(v) and agg(v); mark prefix dirty
18: end procedure
```

7.6 Rebuild-after-K-splits

We rebuild after a global split counter reaches K.

Algorithm 9 CollectLeaves and Rebuild

```
1: procedure CollectLeaves(v, a)
2:     if v has no children then return
3:     end if
4:     Push(v)
5:     if v is leaf-level parent then
6:         for all leaf children u of v do append agg(u) to a
7:         end for
8:     else
9:         for all children c of v do CollectLeaves(c, a)
10:        end for
11:    end if
12: end procedure
13: procedure Rebuild(root)
14:    a ← empty array
15:    CollectLeaves(root, a)
16:    delete old tree
17:    build new skeleton by global SPF splitting on |a| with threshold T
18:    fill leaf values from a (in order) and recompute aggregates bottom-up
19:    splitCnt ← 0
20: end procedure
```

8 Correctness

Lemma 1 (Prefix routing correctness)**.** *Let a node v satisfy the prefix invariant. For any $i \in [1, \mathsf{size}(v)]$, the smallest j with $p_j \geq i$ is unique and satisfies $p_{j-1} < i \leq p_j$. Therefore routing to child c_j with local index $i - p_{j-1}$ reaches the correct leaf.*

Proof sketch. Prefix sums are nondecreasing and end at $p_k = \mathsf{size}(v)$, so such a j exists. Uniqueness follows from monotonicity. The inequality $p_{j-1} < i \le p_j$ means the first p_{j-1} leaves are in earlier children and the next leaves start in c_j, so $i - p_{j-1}$ is the correct local index.

Lemma 2 (Lazy propagation correctness (SumAdd))**.** *ApplyToNode and Push preserve the aggregate/lazy invariant for SumAdd.*

Proof sketch. Adding Δ to each element in a subtree of size ℓ increases the sum by $\Delta\ell$. ApplyToNode applies exactly this change to $\mathsf{agg}(v)$ and composes $\mathsf{lazy}(v)$. Push forwards the pending update to all children; the represented leaf values and hence subtree sums are unchanged.

Lemma 3 (Local split preserves order and content)**.** *Local split does not change the in-order sequence of leaves, hence it preserves the represented sequence.*

Proof sketch. The procedure moves leaves into buckets in left-to-right order and only inserts an intermediate layer. No permutation occurs, so the in-order leaf sequence is identical before and after splitting.

Lemma 4 (Local split preserves invariants)**.** *After local split, the structural invariants and aggregate/lazy invariant hold.*

Proof sketch. Each new intermediate node covers a consecutive block of the old leaves, so contiguity/order holds. Sizes add up by construction. Aggregates are recomputed from children (and lazies are unchanged except pushed before restructuring), so sums remain correct.

Theorem 1 (Operation correctness)**.** *Assuming invariants hold before an operation, RangeQuery, RangeApply, PointSet, InsertAt, EraseAt, LocalSplit, and Rebuild preserve invariants and implement the intended sequence ADT semantics.*

Proof sketch. Routing correctness follows from Lemma 1. Range procedures recurse only into children whose blocks intersect the query, and use exact aggregates on fully covered nodes, hence return/apply correct results given Lemma 2. Insert/erase update exactly one leaf position in the in-order sequence and adjust sizes/aggregates on the path. LocalSplit correctness is Lemmas 3–4. Rebuild flattens leaves in order and reconstructs a valid tree for the same array.

9 Complexity Analysis

Let N be current sequence length and let H be current height (number of internal levels on a root-to-leaf path). Let $d(v)$ denote the degree of node v.

9.1 Always-correct bounds (parameterized by current height)

Proposition 1 (Routing cost)**.** *At a node v, child selection by lower-bounding prefix sums costs $O(\log d(v))$. Thus Find/routing costs $O\Big(\sum_{\ell=1}^{H} \log d(v_\ell)\Big)$ along the path. If visited degrees are bounded by $O(cT)$, routing is $O(H \log(cT))$.*

Proposition 2 (Insert/erase cost between rebuilds)**.** *Insert/erase performs one routing plus a leaf-level vector insertion/erasure and possibly a local split. If leaf-parent degrees are bounded by $O(cT)$, the vector shift and regrouping cost is $O(cT)$. Thus insert/erase are $O(H \log(cT) + cT)$ between rebuilds.*

Proposition 3 (Range query/apply cost (safe bound))**.** *At each visited internal node, the algorithm may aggregate or update all fully covered children between two boundary children. Therefore the per-node overhead is $O(\log d(v) + d(v))$ in the worst case, yielding a safe bound $O(H(\log(cT) + cT))$ when visited degrees are $O(cT)$.*

9.2 Post-rebuild specialization (typical height)

After rebuild, global splitting yields fanout at most T, and in near-uniform shapes height is typically close to $H \approx \log_T N$.

Corollary 1 (Typical post-rebuild routing)**.** *Under $H \approx \log_T N$ and degrees bounded by T, routing is $O(\log_T N \cdot \log T) = O(\log N)$ by change of base.*

9.3 Rebuild cost and heuristic rebuild overhead

Proposition 4 (Rebuild cost)**.** *Flattening leaves costs $O(N)$, and rebuilding creates $O(N)$ total nodes across levels (geometric shrink), so* Rebuild *costs $O(N)$ time and $O(N)$ memory.*

Proposition 5 (Heuristic rebuild overhead)**.** *Assume adversarial growth is dominated by repeated local splits in a hot region and that each additional split requires $\Omega(T)$ concentrated insertions (e.g., in repeated worst-case fallback to binary splitting). If rebuild is triggered after K splits, then over Q operations the number of rebuilds is $O(Q/(KT))$, giving total rebuild time $O(NQ/(KT))$. With $K = \Theta(T)$ this becomes $O(NQ/T^2)$.*

9.4 Choosing $T \approx N^{1/3}$ and the empirical best range

Two dominating terms in edit-heavy worst-style behavior are: (i) local leaf-level work $O(Q \cdot T)$ and (ii) rebuild overhead $O(NQ/T^2)$. Balancing these suggests choosing T to minimize $T + N/T^2$, yielding $T = \Theta(N^{1/3})$ and overall heuristic time $\Theta(QN^{1/3})$, plus routing/range-query costs.

Empirically, the best exponent depends on workload mix. In our tests, good values often fall in $T \in [N^{0.27}, N^{0.39}]$, where smaller T tends to help query-heavy workloads and larger T tends to help insertion-heavy workloads.

10 Implementation Details

The benchmarked C++ implementation uses an SPF sieve up to a fixed bound and falls back to $S(n) = 2$ above it. Threshold selection uses $T \approx \sqrt[3]{N}$ at (re)build time, with rounding to $2^k - 1$ to reduce overhead. Rebuild triggers after $K = \max(50, 2T)$ local splits. Benchmarks used $c = 4$ (local split at degree $> 4T$).

11 Experiments

11.1 Setup

All benchmarks use $N = Q = 2 \cdot 10^5$. Experiments were run on a Windows machine with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80 GHz (4 physical cores, 8 logical processors). All programs were compiled using MinGW-w64 `g++` 15.1.0 with flags `-std=c++17 -O2 -pipe`. Each benchmark suite contains multiple input test files; we report the average wall-clock runtime over all files.

11.2 Baselines and fairness

We compare against:

- **Implicit treap (dynamic):** supports range add, range sum, point set, insertion, and deletion.
- **Lazy segment tree (fixed-length):** supports range add and range sum on a fixed-length array.

Table 1: Benchmark summary (SumAdd). Segment tree is evaluated only on fixed-length suites.

Test suite	Bahnasy variant	Reference	Tests	Bahnasy avg	Ref avg
All operations	dynamic Bahnasy (rebuild)	implicit treap	105	0.202s	0.193s
Point update + query	Bahnasy (fixed-length)	segment tree	18	0.303s	0.218s
Range update + query	Bahnasy (fixed-length)	segment tree	12	0.274s	0.236s

Because a segment tree does not natively support insertion/deletion without rebuilding, we evaluate it only on fixed-length suites. For the full "all operations" suite, we compare Bahnasy Tree only against the treap baseline.

11.3 Workloads and I/O format

We use the SumAdd operation set. For the fully dynamic suite, operations are:

- `op=1`: point set (i, x)
- `op=2`: range sum query (l, r)
- `op=3`: range add (l, r, Δ)
- `op=4`: insert (i, x)
- `op=5`: erase (i)

Fixed-length suites contain only the subset of operations relevant to the generator (e.g., range add + range sum).

11.4 Correctness checking

For each test file, Bahnasy outputs were compared against the corresponding baseline implementation. No mismatches were observed in the reported runs.

11.5 Results

On fixed-length workloads, the segment tree benefits from tight $O(\log N)$ structure and small constants. Bahnasy Tree targets fully dynamic workloads where inserts and deletes are frequent; there, the main baseline is the implicit treap.

12 Limitations and Future Work

The structure can exhibit increased height under adversarial insertion patterns between rebuilds. Future work includes (i) more formal amortized guarantees under explicit rebuild triggers, (ii) alternative deterministic balancing rules, (iii) cache-aware node layouts, and (iv) deamortized rebuilding.

13 Conclusion

We presented Bahnasy Tree, a multi-way dynamic sequence structure using prefix-size routing, SPF-guided splitting, local overflow repair at degree $> cT$, and rebuild-after-K-splits. A heuristic choice $T \approx N^{1/3}$ balances local edit overhead and rebuild cost, and experiments suggest a practical exponent range of 0.27–0.39.

References

[1] C. Seidel and M. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[3] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software—Practice & Experience*, 25(12):1315–1330, 1995.