Lazy Search Trees

Introduction

Rafał Włodarczyk

INA 4

October 23, 2025

Overview

A Brief History of Tree Data Structures

These tree data structures had the biggest impact on the field:

- **Binary Search Tree (1960s)** $O(\log n)$ average-case query, $O(n)$ worst-case query.
- **AVL Tree (1962)** Self-balancing, $O(\log n)$ worst-case for insert and query.
- **Red-Black Tree (1972)** Self-balancing, $O(\log n)$ worst-case for insert and query, often with fewer rotations than AVL.
- **Splay Tree (1985)** Self-adjusting, performance optimized for sequences of operations. Frequently queried keys are closer to the root node.
- **Treap (1989)** Randomized, combines properties of binary search trees and heaps.

Building on research into Deferred Tree Structures, the **Lazy Search Tree (2020)** introduces postponed updates and efficiency for specific query patterns.

Problem Domain

Assume the comparison model and consider a dynamic multiset S of (current) size $|S| = n$. Our goal is to efficiently support two types of operations:

- order-based operations: rank, select, membership, predecessor, successor, minimum, and maximum.
- dynamic operations: insert, delete, change-key, split, and merge.

Example: if we only want to support minimum and maximum element we are left with a **priority queue**. We therefore aim for a generalization of priority queues, which provide all of the order-based operations from above.

Performance-wise, we want to get as close to a dictionary implemented with a hash table, while supporting the order-based operations - ultimately achieving what is known as a **sorted dictionary**.

Sorted Dictionary Interface

The paper proposes the following interface to define a sorted dictionary:

- **Construction(S)** Construct a sorted dictionary on the set S.
- **Insert(e)** Add (k, v) to S, where k is a comparable key.
- **RankBasedQuery(r)** A general rank-based query on S. Example. r-th smallest element.
- **Delete(ptr)** Delete the element pointed to by `ptr` from S.
- **ChangeKey(ptr, k')** Change the key of the element pointed to by `ptr` to k'.
- **Split(r)** Split S at rank r, returning two sorted dictionaries T_1 and T_2 of r and $n - r$ elements, respectively, which satisfy $(\forall x \in T_1)\,(\forall y \in T_2)\,(x \leq y)$.
- **Merge($\mathbf{T_1}$, $\mathbf{T_2}$)** Merge sorted dictionaries T_1 and T_2 and return the result, which satisfies $(\forall x \in T_1)\,(y \in T_2)\,(x \leq y)$.

Gaps

We maintain a set of m gaps $\{\Delta_i\}$, $1 \leq i \leq m$, where each gap Δ_i contains a bag of elements. The gaps satisfy **the total order property** - $(\forall x \in \Delta_i)\,(y \in \Delta_{i+1})\,(x \leq y)$.

Example 1. Take an example multiset $S = \{(1, a), (1, a), (3, c), (4, d)\}$, composed of pairs (k_i, v_i), we can represent it as a set of gaps, which satisfy the total order on k_i.

$$\Delta_1 = \{(1, a)\} \quad \Delta_2 = \{(1, a)\} \quad \Delta_3 = \{(3, c)\} \quad \Delta_4 = \{(4, d)\}$$

Example 2. We could also define it as:

$$\Delta_1 = \{(1, a), (1, a)\} \quad \Delta_2 = \{(3, c), (4, d)\}$$

Gap insertion and deletion

Example 3. Let's insert a new element $(k, v) = (3, b)$ to gaps from Example 1. We must maintain the total order property, therefore we find a gap, such that:

$$(\forall k_i \in \Delta_i)\,(\forall k_{i+1} \in \Delta_{i+1})\; k_i \leq k < k_{i+1}$$

Which in this case is Δ_2 or Δ_3. We insert the element into the gap, resulting in:

$$\Delta_1 = \{(1, a)\} \quad \Delta_2 = \{(1, a)\} \quad \Delta_3 = \{(3, c), (3, b)\} \quad \Delta_4 = \{(4, d)\}$$

Alternatively:

$$\Delta_1 = \{(1, a)\} \quad \Delta_2 = \{(1, a), (3, b)\} \quad \Delta_3 = \{(3, c)\} \quad \Delta_4 = \{(4, d)\}$$

The paper specifies both are valid, and the choice is left to the implementation.

Deletion happens similarly, we find the gap containing the element to be deleted, and remove it from the bag - if the bag is empty, we remove the gap itself.

Intervals

Within each gap Δ_i, elements are further partitioned into intervals. Each interval is defined by a pair of keys (k_l, k_r). The interval only contains a key k if $k_l \leq k \leq k_r$.
Example. For the gap Δ_t:

$$\Delta_t = \{(1, a), (1, b), (2, c), (3, d)\}$$

We can present the following interval representation:

$$\mathcal{I} = \{(1, 2), (3, 3)\}$$

The paper proves there are at most $4 \log(|\Delta_i|)$ intervals in each Δ_i gap.

Lazy Search Tree

The data structure has two levels:

- **The Gaps** $\{\Delta_i\}$ at the top level. Within each **gap** $\{\Delta_i\}$, there are:
- **The Intervals** $\mathcal{I}_{i,1}, \mathcal{I}_{i,2}, \ldots, \mathcal{I}_{i,\ell_i}$, where ℓ_i is # of intervals in gap Δ_i.

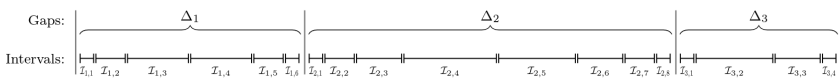

Figure: Visualization of a Lazy Search Tree.

We can store the gaps using a globally biased $(2, B)$-Tree and the intervals as a sorted array.

LST - Construction

Let us insert the sequence: $5, 12, 3, 8, 15, 6$.

- **Initial** $\{\}$.
- **Add 5** $\Delta_1 = \{5\}$
- **Add 12** $\Delta_1 = \{5, 12\}$
- **Add 3** $\Delta_1 = \{3\}, \Delta_2 = \{5, 12\}$
- **Add 8** $\Delta_1 = \{3\}, \Delta_2 = \{5, 8, 12\}$
- **Add 15** $\Delta_1 = \{3\}, \Delta_2 = \{5, 8, 12, 15\}$
- **Add 6** $\Delta_1 = \{3\}, \Delta_2 = \{5, 6, 8, 12, 15\}$

Construction inevitably runs in $O(n)$ time.

LST - Insertion

We either create a new gap Δ_1 or find the smallest i, such that $(\forall k \in \Delta_i)\,(\text{key} \geq k)$

```
void insert(const T &key) {
    if (empty()) {
        gap r_gap = gap(key);
        gap_ds.insert(r_gap);
    } else {
        gap& r_gap = gap_ds.lower_bound_or_last(gap(key));
        r_gap.insert(key);
    }
    ++lst_size;
}
```

Lower bound or last gap - Δ_i can be found with binary search in $O\left(\log \frac{n}{\Delta_i}\right)$. There are at most $O(\log \Delta_i)$ intervals in Δ_i. Binary search to find an interval then runs in $O(\log \log \Delta_i)$ time. Therefore the whole insert runs in $O\left(\log \frac{n}{\Delta_i} + \log \log \Delta_i\right)$.

LST - Insertion Example with Intervals

Assume LST: $\Delta_1 = \{3\}, \Delta_2 = \{5, 6, 8, 12, 15\}$, with intervals
$\mathcal{I}_{1,1} = [3, 3], \mathcal{I}_{2,1} = [5, 6, 8], \mathcal{I}_{2,2} = [12, 15]$ and we want to insert key $= 9$.

1. Find the relevant gap. The `lower_bound_or_last` function selects Δ_2 as the first target gap, since $9 > 3$.
2. Find the interval within gap. Within Δ_2, 9 falls between the elements of $\mathcal{I}_{2,1}$ and $\mathcal{I}_{2,2}$.
3. Place 9 in Δ_2 in its sorted position. Update gaps or intervals depending on the situation - must maintain the total order property and not degenerate the structure.

We end up with LST: $\Delta_1 = \{3\}, \Delta_2 = \{5, 6, 8, 9, 12, 15\}$, with intervals
$\mathcal{I}_{1,1} = [3, 3], \mathcal{I}_{2,1} = [5, 6, 8]$, $\mathcal{I}_{2,2} = [9, 9]$, $\mathcal{I}_{2,3} = [12, 15]$).

How intervals are found in code

```
int getIntervalIdx(const T &key) {
  int lo = last_left_idx, hi, mult;
  bool init = key <= intervals[last_left_idx]->get_max();
  mult = (init) ? -1 : 1;

  exponential_search(lo, hi, mult, init, key, intervals);
  return binary_search(lo, hi, init, key, intervals);
}
```

Paper states it is optimized to provide $O(1)$ average case insert, $O(\log \log \Delta_i)$ worst case.

LST - Query

When performing a query, we first identify the gap Δ_i that contains the rank element r. Formally, we find i such that:

$$\sum_{j=1}^{i-1} |\Delta_j| < r \leq \sum_{j=1}^{i} |\Delta_j|.$$

We then answer the query using the elements within Δ_i.
During this process, we can restructure the gaps by splitting Δ_i into two new gaps, Δ'_i and Δ'_{i+1}, keeping the total order property. The split is performed so that the rank element r is the largest in Δ'_i or the smallest in Δ'_{i+1}. Specifically, after the split, either

$$|\Delta'_i| + \sum_{j=1}^{i-1} |\Delta_j| = r \quad \text{or} \quad |\Delta'_i| + \sum_{j=1}^{i-1} |\Delta_j| = r - 1.$$

This ensures that the structure remains consistent and supports efficient future queries.

LST - Update on Query

After each query we can perform an update.

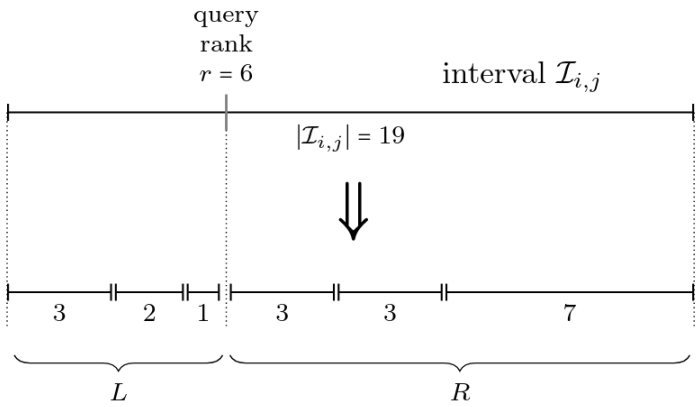

Figure: Visualization of an interval update.

LST - Count function

```
bool membership(const T &key) {
  return intervals[getIntervalIdx(key)]->membership(key);
}

int count(const T &key) {
    if (empty()) return false;
    gap &r_gap = gap_ds.lower_bound_or_last(gap(key));
    bool result = r_gap.membership(key);

    pair<gap, gap> new_gaps = r_gap.restructure(key, 2);
    gap_ds.erase(r_gap);
    if (!new_gaps.first.empty())  gap_ds.insert(new_gaps.first);
    if (!new_gaps.second.empty()) gap_ds.insert(new_gaps.second);
    return result;
}
```

LST - Chained Queries

The Lazy Search Tree adapts its efficiency based on query distribution.

- **General Case**. Over a sequence of n insertions and q distinct queries, the total complexity is $O(B + \min(n \log\log n, n \log q))$. $B = \Sigma_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$ is defined in the abstract. A time bound of $O(n \log q + q \log n)$ holds.
- **Few Queries**. If q is small (e.g., $q = O(1)$), lazy search trees can achieve near-linear time for the sequence.
- **Clustered Queries**. For q/k queries each requesting k consecutive keys, with uniform insertions in between, the total cost is $O(n \log(q/k) + \min(n \log\log n, n \log q))$ for insertions and $O(\log n)$ for successive queries within a batch after the first query.
- **Priority Queue**. If all queries are for the minimum element, the Lazy Search Tree functions as a priority queue. Insertions take $O(\log\log n)$ time, and each query takes $O(\log n)$ time.

References

1. Sandlund, Bryce, and Sebastian Wild. "Lazy search trees." 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS). IEEE, 2020. https://arxiv.org/abs/2010.08840
2. Sandlund, Bryce. "Lazy Search Trees (GitHub repository)." https://github.com/brycesandlund/lazy-search-trees

Further read:

1. Rysgaard, Casper Moldrup, and Sebastian Wild. "Lazy B-Trees." arXiv preprint arXiv:2507.00277 (2025). https://arxiv.org/abs/2507.00277