Exploring different merging algorithms for balanced trees and their time complexity optimization.

Changhui (Eric) Zhou

October 7, 2025

word count: ???

Contents

1 Introduction

A *data structure* is a way to store and organize data in order to facilitate access and modifications (Cormen, Leiserson, Rivest, & Stein, 2022). Designing and choosing more efficient data structures has always been a great persuit for computer scientists, for optimal data structures can save huge amount of computing resources, especially in face of large amount of data. Basic data structures include ordered data structures like arrays, linked lists and binary search trees and unordered data structures like hashtables.

For ordered data structures, merging two or more instances of them while maintaining its ordered property may be frequently used in practice. For example, to investigate the factors affacting the school grade, data from different schools may be grouped and merged according to various factors. The efficiency of combination varies significantly based on the data structure itself and the algorithm used in the process.

This essay will focus on invesitgating the theoretical time complexity (need definitions aa) and actual performance of merging algorithms of different data structures, namely arrays, and BSTs, which are the most commonly used data structure in real life.

Research question: How does different algorithm affect the efficiency of merging two instances of balanced search trees?

2 Theory

2.1 Data structure terminology

When a homogeneous relation (a binary relation between two elements) $\leq$ on a set of elements X satisfies:

1. Antisymmetry: $\forall u, v \in X, (u \leq v \wedge v \leq u) \Leftrightarrow u = v.$
2. Totality: $\forall u, v \in X, u \leq v \vee v \leq u.$
3. Transitivity: $\forall u, v, w \in X, (u \leq v \wedge v \leq w) \Rightarrow u \leq w.$

We say $P = (X, \leq)$ is a total order. For example $P = (\mathbb{R}, \leq)$, where $\leq$ is numerical comparison, is a total order. The set of finite strings and lexographical order comparison is also a total order. But $P = (\{S : S \subset \mathbb{R}\}, \subset)$ is not a total order.

Ordered data structures can store elements that satisfies a total order while maintaining their order. In C++, arrays, vectors, linked lists and sets can be ordered data structures, but unordered sets (hashtables) are not ordered data structures.

Definition

Ordered data structures are data structures that can store elements that satisfies a total order while maintaining their order.

2.2 Balanced binary search trees

- A *graph* $G = (V, E)$ is the combination of the vertex set V and the edge set E.
- A *tree* $T = (V, E)$ is a connected acylic graph.
- A *binary tree* is a tree that has no more than two children for each node.
- A *binary search tree (BST)* is a binary tree, whose nodes contain values under a total order, that has the following property: For any node v, all nodes in its left subtree are less than v, and all nodes in its right subtree are greater than v (Cormen et al., 2022).

Generally speaking, a balanced BST is a BST whose depth or the cost of iterating from the root to one specific leaf is strictly, expectedly or amortized $O(\log n)$. There are different kinds of balanced BSTs, like splay tree, treap, AVL trees and red-black trees. This essay will focus on AVL trees. AVL trees are a type of self-balancing binary search trees, which adjusts its shape through rotations and maintain the difference of the depths of two subtrees at most one (Karlton, Fuller, Scroggs, & Kaehler, 1976)[1].

In this essay, we will assume that there are two instances of AVL trees T_1 and T_2 to be merged. Without losing generality, we will assume T_1 has n elements and T_2 has m elements and $n \geq m$.

[1]In fact, this kind of BST was refered to as HB[1], but AVL trees nowadays are mainly HB[1]

2.3 Insertion-based merge

One of the basic operations supported by a balanced BST is insertion, where one element is added to the tree and the order of the tree is automatically maintained. In fact, merging two instances of BSTs can be reduced to a sequence of insertions to a balanced BST. To be more specific, we iterate through all the elements in T_2, insert them one by one into T_1, that would be m operations with each having time complexity of $O(\log n)$, resulting in a overall complexity of $O(m \log n)$.

Algorithm: Insertion-Based Merge

Require: Two balanced binary search trees T_1 and T_2

Ensure: A single balanced binary search tree containing all elements from T_1 and T_2

```
1: procedure InsertionBasedMerge(T1, T2)
2:     for all elements x in T2 (in-order traversal) do
3:         T1.Insert(x)
4:     return T1
```

This algorithm performs well when m/n is small, as the overall time complexity will be mainly $O(\log n)$. However, when m and n are relatively at the same scale, the overall time complexity will be close to $O(n \log n)$.

2.4 In-order traversal merge

Another way to merge two instances of BSTs is to utilize the property that each instance is already in-order. To combine them, we can view this process as merging two sorted subarrays into a new array, just like a merge sort. The iteration and new array construction process take $O(n+m)$ time. With proper construction function, we can create a balanced BST in linear time out of a sorted array. Therefore, the overall time complexity is $O(n+m)$.

Algorithm: In-order traversal merge

Require: Two balanced binary search trees T_1 and T_2
Ensure: A single balanced binary search tree containing all elements from T_1 and T_2

```
procedure MergeSortBasedMerge(T1, T2)
    A1 ← InOrderTraversal(T1)
    A2 ← InOrderTraversal(T2)
    A ← MergeSortedArrays(A1, A2)
    T ← BuildBalancedBST(A)
    return T
```

This algorithm performs well when m/n is large, as the overall time complexity will be approximately $O(n)$. However, when m is pretty negeligible compared to n, a full iteration over T_1 will be still needed and the overall time complexity will still be $O(n)$, which wastes a lot of time.

In fact, it Stockmayer and Yao has proven that in term of number of comparisions, this algorithm is optimal when $m \leq n \leq \lfloor 3m/2 \rfloor + 1$ (Stockmeyer & Yao, 1980). This algorithm, however, does not perform well outside this range.

2.5 Brown and Tarjan's merging algorithm (1979)

In 1979, Brown and Tarjan proposed another algorithm based on the two merging algorithm mentioned above. It utilized both the tree structure for fast insertion-place location and the ordered property to reduce redundent operations. The algorithm again chooses the T_1 as the base tree and view the merging process as m insertions to a balanced BST of size n. However, the property that the inserted objects themselves are sorted helped to make the algorithm more efficient. Instead of iterating from the root, each insertion starts with the ending position of the last insertion, as it can be already told that the next insertion will happen to the right of the last insertion.

To be more specific, the algorithm keeps a stack called *path* and a stack called *successor*. The former is used to record the path from the root to the current node, while the latter records all the nodes on the *path* that is larger than the current node (that means they are on the right side of the current node their left children is visited on the *path*). Each insertion, instead of starting from the root, starts from the last node on the *successor* that is smaller than the node to be inserted. Keep extending the *path* and *successor* during insertion. And the path shrinks back after the insertion, until a rebalance operation is triggered or we know that there need no rebalancing at all.

It is worth noticing that the rebalance operation may make the initial path unusable. In this case, we can simply dispose of the path under the rotated node start next insertion there (Brown & Tarjan, 1979).

Algorithm: Brown and Tarjan's Merging Algorithm

Require: Two balanced binary search trees T_1 (size n) and T_2 (size m), where $n \geq m$

Ensure: A single balanced BST containing all elements from T_1 and T_2

procedure FastMerge(T_1, T_2)
    Initialize stack $path \leftarrow \{\text{root}(T_1)\}$
    Initialize empty stack *successor*
    $height \leftarrow \text{height}(T_1)$
    for all nodes x in T_2 (in-order traversal) **do**
        Detach x from T_2
        ▷ — *Step 1: Adjust path to maintain PathPredicate* —
        while *successor* not empty **and** $key(x) > key(\text{top}(successor))$ **do**
            repeat
                pop($path$)
            until $\text{top}(path) = \text{top}(successor)$
            pop($successor$)
        ▷ — *Step 2: Search down from last successor and insert x* —
        $p \leftarrow \text{top}(path)$
        while True **do**
            if $key(x) < key(p)$ **then**
                if $p.left = Nil$ **then**
                    $p.left \leftarrow x$; **break**
                else
                    push(p, $successor$); $p \leftarrow p.left$
            else
                if $p.right = Nil$ **then**

```
                p.right ← x; break
            else
                p ← p.right
        push(p, path)
    ▷ — Step 3: Adjust balance factors and rebalance if needed —
    while path not empty do
        s ← pop(path)
        if tree at s is unbalanced then
            REBALANCE(s); break
        else
            Update balance factor of s
        if top(successor) = s then
            pop(successor)
    return root of T_1 (now merged)
```

2.6 Optimality

When merging two instances of size n and m respectively, there are in total $\binom{n+m}{n}$ possible outcomes. According to the decision tree theory, each of them corresponds to a decision tree leaf node. Since the merging algorithm is comparison based, the decision tree has to be a binary tree (i.e. Each node has at most two children). The height of the decision tree is therefore no lower than $O(\log_2(\binom{n+m}{n}))$.

According to Sterling's approximation,

$$n! \approx \sqrt{2\pi n}(\frac{n}{e})^n \tag{1}$$

which means

$$O(\log(n!)) = O(\frac{1}{2}\log(2\pi n) + n\log n - n\log e) \quad = O(n\log n - n + O(logn)) \tag{2}$$

Using the definition of combination number,

$$\binom{n+m}{m} = \frac{(n+m)!}{n!m!} \tag{3}$$

$$\log(\binom{n+m}{n}) = \log(\frac{(n+m)!}{n!}) \tag{4}$$
$$= (n+m)\log(n+m) - n\log n - m\log m + O(\log(n+m)) \tag{5}$$
$$= n\log(1+\frac{m}{n}) + m\log(1+\frac{n}{m}) + O(\log(n+m)) \tag{6}$$

Since $m \leq n$ we have $n\log(1+m/n) \leq n \cdot (\frac{m}{n}) = m$, therefore the first term is $O(m)$. This means the first term should be neglected as $m\log(1+\frac{n}{m})$ is the dominant term compared to $O(m)$.

Since $m\log(1+\frac{n}{m})$ can be written as $m\log(n+m) - m\log(m)$, where the first term is more dominant than $O(\log(n+m))$, the third term should be neglected as well.

We can get the overall expression

$$\boxed{\log\binom{n+m}{n} = O(m\log(1+\frac{n}{m}))} \tag{7}$$

Theorem

The optimal time complexity of merging two instances of ordered data structures is $O(m\log(1+\frac{n}{m}))$, multiplied by the comparision cost with is assumed to be $O(1)$ in this case.

3 Hypothesis

4 Experiment Design

4.1 Variables

4.1.1 Independent variables

The independent variables of this experiment are:

- $\alpha = n/m \in \{1.0, 9.0, 99.0, 999.0\}$.
- $N = n + m \in \{10^4, 10^5, 10^6, 10^7, 10^8\}$

From the definition equation, we can find property shown in Equation 8,

$$n = \frac{\alpha N}{1+\alpha}, m = \frac{N}{1+\alpha} \tag{8}$$

which means for each pair of α and N, there is a corresponding pair of n and m. However, n and m are not chosen as the independent variables because α can better represent the **relative scale** or **balance** of n and m, while N can better represent the **total scale** of data. We are more interested in these two properties, rather than the scale of one particular part of data.

α and N are both almost uniformly chosen in the lographic scale, since it can better help us investigate the varying data size in application. The reason why α are not chosen in $10^k (k \in \mathrm{Z}^+)$ but $10^k - 1 (k \in \mathrm{Z}^+)$ is to avoid floating point calculation and error when choosing n and m.

4.1.2 Dependent variables

The dependent variable of the experiment is the efficiency of the algorithm. To be more specific, the efficiency is measured by the **clock time** taken by the algorithm t, as well as the **number of comparison operations** c.

These two variables can both represent the cost and measure the efficiency of the algorithm, but they have different foci and advantage over theoretical time-compexity analysis. The clock time t can reveal the invisible cost neglected term during big-O analysis, better representing the real world efficiency, while the comparison operations c can better represent the actual number of comparisions, helping us to analyze the performance when the comparison is not actually $O(1)$ (e.g. lexographical order can be more costly to compare.)

4.1.3 Controlled Variables

5 Results

5.1 Raw data

The output is `seed = 524042979266800` and a 2701 row csv file, part of which is shown in Table 1.

Table 1: Raw data

Method	α	N	Trial	Time (ms)	Comparisons
Insertion-based	1	4096	1	2.0169	90036
Insertion-based	1	4096	2	1.5414	91083
Insertion-based	1	4096	3	1.5598	91498
Insertion-based	1	4096	4	1.5160	90642
Insertion-based	1	4096	5	1.5823	89572
		[...]			
Brown and Tarjan's	4096	1048576	16	0.5161	5079
Brown and Tarjan's	4096	1048576	17	0.5049	5175
Brown and Tarjan's	4096	1048576	18	0.5001	5088
Brown and Tarjan's	4096	1048576	19	0.4834	5147
Brown and Tarjan's	4096	1048576	20	0.5141	5180

Appendix A Test environment

Table 2: Test environment

Device:	Laptop
CPU:	Intel(R) Core(TM) Ultra 7 155H (1.40 GHz)
Memory:	32GB
OS:	Windows 11 24H2 26100.6584
Compiler (C++):	g++ 15.2.0 (algorithm implementation and timing)
Interpreter:	Python 3.12.2 (graphing)

Appendix B Algorithm implementation

Listing 1: AvlSet

```
#include <vector>
#include <algorithm>
#include <memory>
#include <list>
#include <functional>
#include <stack>
#ifdef DEBUG
#include <assert.h>
#endif

template <typename T, typename Compare = std::less<T>>
```

```
class AVLSet {
private:
    struct Node {
        T key;
        Node* left;
        Node* right;
        int height;

        template <typename... Args>
        Node(Args&&... args)
            : key(std::forward<Args>(args)...),
              left(nullptr),
              right(nullptr),
              height(1) {}
    };

    Node* root;
    size_t size;
    Compare comp_;

    // Helper functions for merging
    int height(Node* node) const {
        return node ? node->height : 0;
    }

    Node* create_node(const T& key) {
        ++size;
        return new Node(key);
    }

    Node* create_node(T&& key) {
        ++size;
        return new Node(std::move(key));
    }

    // Generates a balanced subtree in O(n) time out from a ordered sequence.
    // Returns the root node pointer.
    // *Preconditions
    // keys have to be ordered
    // _RandAccIt is the random access iterator
    // (*bg) and (*ed) should be of type T

    template<typename _RandAccIt>
    Node* build(const _RandAccIt& bg, const _RandAccIt& ed){
        if(bg == ed) return nullptr;
        auto it = bg;
        if(++it == ed) return create_node(*bg);
        it = bg + (ed - bg) / 2; // The same as (bg + ed)/2 but avoids overflow problems
        auto cur = create_node(*it);
        cur->left = build(bg, it);
        cur->right = build(it + 1, ed);
        update_height(cur);
        return cur;
    }

    void update_height(Node* node) {
        node->height = 1 + std::max(height(node->left), height(node->right));
```

```cpp
    }

    void delete_tree(Node* node) {
        if (!node) return;
        delete_tree(node->left);
        delete_tree(node->right);
        --size;
        delete node;
    }

    Node* rotate_right(Node* y) {
        Node* x = y->left;
        Node* T2 = x->right;

        x->right = y;
        y->left = T2;

        update_height(y);
        update_height(x);

        return x;
    }

    Node* rotate_left(Node* x) {
        Node* y = x->right;
        Node* T2 = y->left;

        y->left = x;
        x->right = T2;

        update_height(x);
        update_height(y);

        return y;
    }

    int balance_factor(Node* node) const {
        return node ? height(node->left) - height(node->right) : 0;
    }

    Node* balance(Node* node) {
        update_height(node);
        int bf = balance_factor(node);

        // Left Heavy
        if (bf > 1) {
            if (balance_factor(node->left) < 0)
                node->left = rotate_left(node->left);
            return rotate_right(node);
        }
        // Right Heavy
        if (bf < -1) {
            if (balance_factor(node->right) > 0)
                node->right = rotate_right(node->right);
            return rotate_left(node);
        }
        return node;
```

```cpp
    }

    template <typename Func>
    void traverse_in_order(Node* node, Func f) const {
        if (!node) return;
        std::stack<Node*> stack;
        Node* current = node;
        while (current || !stack.empty()) {
            while (current) {
                stack.push(current);
                current = current->left;
            }
            current = stack.top();
            stack.pop();
            f(current->key);
            current = current->right;
        }
    }

    Node* insert(Node* node, const T& key) {
        if (!node) {
            return create_node(key);
        }

        if (comp_(key, node->key)) {
            node->left = insert(node->left, key);
        } else if (comp_(node->key, key)) {
            node->right = insert(node->right, key);
        } else {
            return node;
        }

        return balance(node);
    }

public:
    AVLSet(Compare Comp = Compare()) : root(nullptr), size(0), comp_(Comp) {}
    Compare comparator() const { return comp_; }
    // Move operations
    AVLSet(AVLSet&& other) noexcept
        : root(other.root),
          size(other.size){
        other.root = nullptr;
        other.size = 0;
    }

    AVLSet& operator=(AVLSet&& other) noexcept {
        if (this != &other) {
            clear();
            root = other.root;
            size = other.size;
            other.root = nullptr;
            other.size = 0;
        }
        return *this;
    }

```

```
    // Disable copy operations
    AVLSet(const AVLSet&) = delete;
    AVLSet& operator=(const AVLSet&) = delete;

    void clear() {
        delete_tree(root);
        root = nullptr;
    }

    bool empty() const {
        return size == 0;
    }

    size_t get_size() const {
        return size;
    }

    template <typename Func>
    void traverse_in_order(Func f) const {
        traverse_in_order(root, f);
    }

    std::vector<T> items() const {
        std::vector<T> result;
        traverse_in_order([&result](const T& key) {
            result.push_back(key);
        });
        return result;
    }


    void swap_with(AVLSet& other) {
        std::swap(root, other.root);
        std::swap(size, other.size);
    }

    void insert(const T& val){
        root = insert(root, val);
    }
    void remove(const T& val){
        root = remove(root, val);
    }
    template<typename RandAccIt>
    void construct(RandAccIt bg, RandAccIt ed){
        // Clear the current tree
        delete_tree(root);
        root = nullptr;
        // Build new tree
        root = build(bg, ed);
        size = static_cast<size_t>(ed - bg);
    }


    /** Merge two sets in O(N+M) time.
    * Some additional space may be costed.
    * But it does not affect the result of the experiment.
    * @param other The AVLSet to be merged into this set.
```

```cpp
    */

    void linearmerge(AVLSet&& other) {
        if (other.empty()) return;

        // Get elements from both trees
        std::vector<T> q1 = items();
        std::vector<T> q2 = other.items();
        std::vector<T> all_elements;
        all_elements.reserve(q1.size() + q2.size());

        // Merge sorted vectors
        std::merge(q1.begin(), q1.end(), q2.begin(), q2.end(),
                   std::back_inserter(all_elements), comp_);

        // Recycle both trees
        delete_tree(root);
        delete_tree(other.root);
        root = nullptr;
        other.root = nullptr;

        // Build new tree
        root = build(all_elements.begin(), all_elements.end());
        size = all_elements.size();
    }

    /**
     * Merge two sets in O(N log(M)) time.
     * @param other The AVLSet to be merged into this set.
     */
    void simplemerge(AVLSet&& other) {
        if (other.empty()) return;

        if (size < other.size) { swap_with(other); }

        // Insert all elements from the smaller tree (now 'other') into this
        other.traverse_in_order([this](const T& key) {
            this->insert(key);
        });

        other.clear();
    }

    void brownmerge(AVLSet&& other) {
        if (other.empty()) return;
        if (size < other.size) swap_with(other);

        std::vector<T> elems = other.items();
        other.clear();

        // stacks of pointers-to-links (Node**). Each points to some parent->left or
            parent->right or &root
        std::vector<Node**> path;
        std::vector<Node**> successor;

        path.push_back(&root);

```

```
        for (const T& x : elems) {
            // ==== CLIMB/RETRACT: pop successors while their key <= x (i.e. not (x < succ_key)) ====
            while (!successor.empty() && !comp_(x, (*successor.back())->key)) {
                Node** succLink = successor.back();
                // pop path entries until top == succLink
                while (!path.empty() && path.back() != succLink) path.pop_back();
                successor.pop_back();
            }

            // ==== DESCEND from current finger (path.back()) to insertion point ====
            Node** curLink = path.empty() ? &root : path.back();
            Node* p = *curLink;
            if (!p) {
                // empty subtree (rare because root existed), insert directly
                *curLink = create_node(x);
                path.push_back(curLink);
            } else {
                for (;;) {
                    path.push_back(curLink); // link that points to p
                    if (comp_(x, p->key)) {
                        // go left; mark this link as a successor (we turned left here)
                        if (p->left == nullptr) {
                            p->left = create_node(x);
                            path.push_back(&(p->left));
                            break;
                        } else {
                            successor.push_back(curLink);
                            curLink = &(p->left);
                            p = *curLink;
                        }
                    } else {
                        // go right
                        if (p->right == nullptr) {
                            p->right = create_node(x);
                            path.push_back(&(p->right));
                            break;
                        } else {
                            curLink = &(p->right);
                            p = *curLink;
                        }
                    }
                }
            }

            // ==== REBALANCE upward using the link-stack; attach rotated subtree via *link ====
            while (!path.empty()) {
                Node** link = path.back();
                Node* s = *link;
                path.pop_back();

                if (!successor.empty() && successor.back() == link) successor.pop_back();

                update_height(s);
                int bf = balance_factor(s);

```

```
            if (std::abs(bf) > 1) {
                Node* newsub = balance(s); // returns new root of this subtree
                *link = newsub; // reattach correctly via the link

                // retract path until the remaining entries are consistent with this
                     rotation
                while (!path.empty() && path.back() != link) path.pop_back();
                break; // stop climbing after performing rotation
            }

            if (bf == 0) {
                // height didn't increase => stop climbing
                break;
            }
            // else continue climbing
        }
    }
    #ifdef DEBUG
    auto items_after = items();
    for (size_t i = 1; i < items_after.size(); ++i)
        assert(!comp_(items_after[i], items_after[i-1]));
    #endif
}

};
```

References

Brown, M. R., & Tarjan, R. E. (1979). A fast merging algorithm. *Journal of the ACM (JACM)*, *26*(2), 211–226.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (Fourth ed.). MIT Press.

Karlton, P. L., Fuller, S. H., Scroggs, R., & Kaehler, E. (1976). Performance of height-balanced trees. *Communications of the ACM*, *19*(1), 23–28.

Stockmeyer, P. K., & Yao, F. F. (1980). On the optimality of linear merge. *SIAM Journal on Computing*, *9*(1), 85–90.