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

Changhui (Eric) Zhou

July 23, 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 theoratical 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 ordered data structures?

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. 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.

2.2 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

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

Using the definition of combination number,

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

which is approximately $O(n\log_2(\frac{n}{m}))$ (provided $n \leq m$).

Appendix

Listing 1: AvlSet

```cpp
#include <vector>
#include <algorithm>
#include <memory>
#include <list>
#include <functional>
#include <stack>

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;

    // Memory pool management
    std::list<Node> node_storage;
    std::vector<Node*> free_nodes;

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

    // Memory pool operations
    Node* create_node(const T& key) {
        if (!free_nodes.empty()) {
            Node* node = free_nodes.back();
            free_nodes.pop_back();
            *node = Node(key);
            return node;
        }
        node_storage.emplace_back(key);
        return &node_storage.back();
    }
```

```

    Node* create_node(T&& key) {
        if (!free_nodes.empty()) {
            Node* node = free_nodes.back();
            free_nodes.pop_back();
            *node = Node(std::move(key));
            return node;
        }
        node_storage.emplace_back(std::move(key));
        return &node_storage.back();
    }

    // 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 recycle_node(Node* node) {
        free_nodes.push_back(node);
    }

    void update_height(Node* node) {
```

```
        node->height = 1 + std::max(height(node->left), height(
            node->right));
    }

    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;
    }

    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;
        }
    }

    void recycle_tree(Node* node) {
        if (!node) return;
        std::stack<Node*> stack;
        Node* current = node;
        Node* last_visited = nullptr;

        while (current || !stack.empty()) {
            if (current) {
                stack.push(current);
```

```
                current = current->left;
            } else {
                Node* top = stack.top();
                if (top->right && top->right != last_visited) {
                    current = top->right;
                } else {
                    recycle_node(top);
                    last_visited = top;
                    stack.pop();
                }
            }
        }
    }

    void insert_with_path(std::vector<Node*>& path, const T& key)
        {

    }

public:
    AVLSet() : root(nullptr), size(0), comp(Compare()) {}

    // Move operations
    AVLSet(AVLSet&& other) noexcept
        : root(other.root),
          size(other.size),
          node_storage(std::move(other.node_storage)),
          free_nodes(std::move(other.free_nodes)) {
        other.root = nullptr;
        other.size = 0;
    }

    AVLSet& operator=(AVLSet&& other) noexcept {
        if (this != &other) {
            clear();
            root = other.root;
            size = other.size;
            node_storage = std::move(other.node_storage);
            free_nodes = std::move(other.free_nodes);
            other.root = nullptr;
```

```
            other.size = 0;
        }
        return *this;
    }

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

    void clear() {
        recycle_tree(root);
        node_storage.clear();
        free_nodes.clear();
        root = nullptr;
        size = 0;
    }

    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;
    }

    /* Merge two sets in O(N+M) time.
    * Some additional space may be costed.
```

```
    * But it does not affect the result of the experiment.
    */

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

        other.free_nodes.clear();

        // 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
        recycle_tree(root);
        recycle_tree(other.root);
        root = nullptr;
        other.root = nullptr;
        size = 0;
        other.size = 0;

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

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

        if (size < other.size) {
            // Swap to merge smaller into larger
            std::swap(root, other.root);
            std::swap(size, other.size);
            std::swap(node_storage, other.node_storage);
            std::swap(free_nodes, other.free_nodes);
```

```
        }

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

        other.clear();
    }

    private:

    Node* insert(Node* node, const T& key) {
        if (!node) {
            size++;
            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:
    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
```

```
        recycle_tree(root);
        root = nullptr;
        // Build new tree
        root = build(bg, ed);
        size = static_cast<size_t>(ed - bg);
    }
};
```

References

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