Comparing different data structures in merging efficiency.

Changhui (Eric) Zhou

May 21, 2025

word count: ???

Contents

1 Introduction

A *data structure* is a particular way of organising data in a computer so that it can be used effectively (GeeksforGeeks, n.d.). 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: VectorSet

```
#include <vector>
#include <algorithm>

template <typename T>
class VectorSet {
private:
    std::vector<T> elements;

```

```
public:
    VectorSet() = default;
    VectorSet(const std::vector<T>& vec, bool sorted = 0){
        elements = vec;
        if(!sorted)
            std::sort(elements.begin(), elements.end());
    }

    // Inserts an element into the set if not already present
    void insert(const T& value) {
        auto it = std::lower_bound(elements.begin(), elements.end(), value);
        if (it == elements.end() || *it != value) {
            elements.insert(it, value);
        }
    }

    // Removes an element from the set if present
    void erase(const T& value) {
        auto it = std::lower_bound(elements.begin(), elements.end(), value);
        if (it != elements.end() && *it == value) {
            elements.erase(it);
        }
    }

    // Checks if an element exists in the set
    bool contains(const T& value) const {
        return std::binary_search(elements.begin(), elements.end(), value);
    }

    // Returns the number of elements in the set
    size_t size() const {
        return elements.size();
    }

    // Checks if the set is empty
    bool empty() const {
        return elements.empty();
```

```cpp
    }

    // Allows iteration over the elements (read-only)
    typename std::vector<T>::const_iterator begin() const {
        return elements.begin();
    }

    typename std::vector<T>::const_iterator end() const {
        return elements.end();
    }

    // Clear all elements from the set
    void clear() {
        elements.clear();
    }

     // Merge another VectorSet into this one (union operation)
     void merge(const VectorSet<T>& other) {
        std::vector<T> merged;
        merged.reserve(elements.size() + other.elements.size());

        auto it1 = elements.begin(), end1 = elements.end();
        auto it2 = other.elements.begin(), end2 = other.elements.
            end();

        while (it1 != end1 && it2 != end2) {
            if (*it1 < *it2) {
                merged.push_back(*it1);
                ++it1;
            } else if (*it2 < *it1) {
                merged.push_back(*it2);
                ++it2;
            } else { // Equal elements
                merged.push_back(*it1);
                ++it1;
                ++it2;
            }
        }

        // Add remaining elements from either vector
```

```
        merged.insert(merged.end(), it1, end1);
        merged.insert(merged.end(), it2, end2);

        elements = std::move(merged);
    }

    // Merge-and-assign operator
    VectorSet<T>& operator+=(const VectorSet<T>& other) {
        merge(other);
        return *this;
    }
};
```

Listing 2: AvlSet

```
#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;

    // 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();
    }

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

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

    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) {
```

```cpp
            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;
        traverse_in_order(node->left, f);
        f(node->key);
        traverse_in_order(node->right, f);
    }

    void insert_merge(const T& key) {
        if (!root) {
            root = create_node(key);
            size++;
            return;
        }

        std::vector<Node*> path;
        std::vector<Node*> successor;
        Node* current = root;
        bool inserted = false;

        // Climb up to find the insertion path
        while (true) {
            path.push_back(current);
            if (comp(key, current->key)) {
                if (!current->left) break;
                successor.push_back(current);
                current = current->left;
```

```
            } else if (comp(current->key, key)) {
                if (!current->right) break;
                current = current->right;
            } else {
                // Duplicate, do not insert
                return;
            }
        }

        // Insert the new node
        Node* newNode = create_node(key);
        if (comp(key, current->key)) {
            current->left = newNode;
        } else {
            current->right = newNode;
        }
        size++;

        // Retrace the path to update heights and balance
        while (!path.empty()) {
            Node* node = path.back();
            path.pop_back();
            node = balance(node);

            if (!path.empty()) {
                if (path.back()->left == node) {
                    path.back()->left = node;
                } else {
                    path.back()->right = node;
                }
            } else {
                root = node;
            }
        }
    }

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

    // Move operations
```

```cpp
    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() {
        node_storage.clear();
        free_nodes.clear();
        root = nullptr;
        size = 0;
    }

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

    size_t get_size() const {
        return size;
    }
```

```cpp

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

    void merge(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_merge(key);
        });

        other.clear();
    }

private:
    // Example of modified insert implementation
    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);
    }

    // Example of modified remove implementation
    Node* remove(Node* node, const T& key) {
        if (!node) return nullptr;

        if (comp(key, node->key)) {
            node->left = remove(node->left, key);
        } else if (comp(node->key, key)) {
            node->right = remove(node->right, key);
        } else {
            // Node deletion with recycling
            if (!node->left || !node->right) {
                Node* temp = node->left ? node->left : node->right
                    ;
                recycle_node(node);
                size--;
                node = temp;
            } else {
                Node* temp = findMin(node->right);
                node->key = std::move(temp->key);
                node->right = remove(node->right, temp->key);
            }
        }

        return node ? balance(node) : nullptr;
    }

    public:
    void insert(const T& val){
        insert(root, val);
    }
    void remove(const T& val){
        remove(root, val);
    }
};
```

References

GeeksforGeeks. (n.d.). *Introduction to data structures.* Retrieved from https://www.geeksforgeeks.org/introduction-to-data-structures/ ([Accessed April 13, 2025])