

# Comparing different data structures in merging efficiency.

Changhui (Eric) Zhou

May 22, 2025

word count: ???

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Theory</b>	<b>2</b>
2.1	Data structure terminology . . . . .	2
2.2	Optimality . . . . .	3
	<b>References</b>	<b>14</b>

# 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 pursuit 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 affecting 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 investigating 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 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,

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

which means

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

Using the definition of combination number,

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

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

## Appendix

Listing 1: VectorSet

```

1  #include <vector>
2  #include <algorithm>
3
4  template <typename T>
5  class VectorSet {
6  private:
7      std::vector<T> elements;
8  }
```

```

9 public:
10     VectorSet() = default;
11     VectorSet(const std::vector<T>& vec, bool sorted = 0){
12         elements = vec;
13         if(!sorted)
14             std::sort(elements.begin(), elements.end());
15     }
16
17     // Inserts an element into the set if not already present
18     void insert(const T& value) {
19         auto it = std::lower_bound(elements.begin(), elements.end
20             (), value);
21         if (it == elements.end() || *it != value) {
22             elements.insert(it, value);
23         }
24     }
25
26     // Removes an element from the set if present
27     void erase(const T& value) {
28         auto it = std::lower_bound(elements.begin(), elements.end
29             (), value);
30         if (it != elements.end() && *it == value) {
31             elements.erase(it);
32         }
33     }
34
35     // Checks if an element exists in the set
36     bool contains(const T& value) const {
37         return std::binary_search(elements.begin(), elements.end()
38             , value);
39     }
40
41     // Returns the number of elements in the set
42     size_t size() const {
43         return elements.size();
44     }
45
46     // Checks if the set is empty
47     bool empty() const {
48         return elements.empty();
49     }

```

```

46     }
47
48     // Allows iteration over the elements (read-only)
49     typename std::vector<T>::const_iterator begin() const {
50         return elements.begin();
51     }
52
53     typename std::vector<T>::const_iterator end() const {
54         return elements.end();
55     }
56
57     // Clear all elements from the set
58     void clear() {
59         elements.clear();
60     }
61
62     // Merge another VectorSet into this one (union operation)
63     void merge(const VectorSet<T>& other) {
64         std::vector<T> merged;
65         merged.reserve(elements.size() + other.elements.size());
66
67         auto it1 = elements.begin(), end1 = elements.end();
68         auto it2 = other.elements.begin(), end2 = other.elements.
69             end();
70
71         while (it1 != end1 && it2 != end2) {
72             if (*it1 < *it2) {
73                 merged.push_back(*it1);
74                 ++it1;
75             } else if (*it2 < *it1) {
76                 merged.push_back(*it2);
77                 ++it2;
78             } else { // Equal elements
79                 merged.push_back(*it1);
80                 ++it1;
81                 ++it2;
82             }
83         }
84
85         // Add remaining elements from either vector

```

```

85     merged.insert(merged.end(), it1, end1);
86     merged.insert(merged.end(), it2, end2);
87
88     elements = std::move(merged);
89 }
90
91 // Merge-and-assign operator
92 VectorSet<T>& operator+=(const VectorSet<T>& other) {
93     merge(other);
94     return *this;
95 }
96 };

```

Listing 2: AVLSet

```

1  #include <vector>
2  #include <algorithm>
3  #include <memory>
4  #include <list>
5  #include <functional>
6  #include <stack>
7
8  template <typename T, typename Compare = std::less<T>>
9  class AVLSet {
10 private:
11     struct Node {
12         T key;
13         Node* left;
14         Node* right;
15         int height;
16
17         template <typename... Args>
18         Node(Args&&... args)
19             : key(std::forward<Args>(args)...),
20               left(nullptr),
21               right(nullptr),
22               height(1) {}
23     };
24
25     Node* root;
26     size_t size;

```

```

27 Compare comp;
28
29 // Memory pool management
30 std::list<Node> node_storage;
31 std::vector<Node*> free_nodes;
32
33 // Memory pool operations
34 Node* create_node(const T& key) {
35     if (!free_nodes.empty()) {
36         Node* node = free_nodes.back();
37         free_nodes.pop_back();
38         *node = Node(key);
39         return node;
40     }
41     node_storage.emplace_back(key);
42     return &node_storage.back();
43 }
44
45 Node* create_node(T&& key) {
46     if (!free_nodes.empty()) {
47         Node* node = free_nodes.back();
48         free_nodes.pop_back();
49         *node = Node(std::move(key));
50         return node;
51     }
52     node_storage.emplace_back(std::move(key));
53     return &node_storage.back();
54 }
55
56 void recycle_node(Node* node) {
57     free_nodes.push_back(node);
58 }
59
60 // Helper functions for merging
61 int height(Node* node) const {
62     return node ? node->height : 0;
63 }
64
65 void update_height(Node* node) {
66     node->height = 1 + std::max(height(node->left), height(

```



```

        node->right));
67     }
68
69     Node* rotate_right(Node* y) {
70         Node* x = y->left;
71         Node* T2 = x->right;
72
73         x->right = y;
74         y->left = T2;
75
76         update_height(y);
77         update_height(x);
78
79         return x;
80     }
81
82     Node* rotate_left(Node* x) {
83         Node* y = x->right;
84         Node* T2 = y->left;
85
86         y->left = x;
87         x->right = T2;
88
89         update_height(x);
90         update_height(y);
91
92         return y;
93     }
94
95     int balance_factor(Node* node) const {
96         return node ? height(node->left) - height(node->right) :
97             0;
98     }
99
100     Node* balance(Node* node) {
101         update_height(node);
102         int bf = balance_factor(node);
103
104         // Left Heavy
105         if (bf > 1) {

```

```

105         if (balance_factor(node->left) < 0)
106             node->left = rotate_left(node->left);
107         return rotate_right(node);
108     }
109     // Right Heavy
110     if (bf < -1) {
111         if (balance_factor(node->right) > 0)
112             node->right = rotate_right(node->right);
113         return rotate_left(node);
114     }
115     return node;
116 }
117
118 template <typename Func>
119 void traverse_in_order(Node* node, Func f) const {
120     if (!node) return;
121     traverse_in_order(node->left, f);
122     f(node->key);
123     traverse_in_order(node->right, f);
124 }
125
126 void insert_merge(const T& key) {
127     if (!root) {
128         root = create_node(key);
129         size++;
130         return;
131     }
132
133     std::vector<Node*> path;
134     std::vector<Node*> successor;
135     Node* current = root;
136     bool inserted = false;
137
138     // Climb up to find the insertion path
139     while (true) {
140         path.push_back(current);
141         if (comp(key, current->key)) {
142             if (!current->left) break;
143             successor.push_back(current);
144             current = current->left;

```

```

145         } else if (comp(current->key, key)) {
146             if (!current->right) break;
147             current = current->right;
148         } else {
149             // Duplicate, do not insert
150             return;
151         }
152     }
153
154     // Insert the new node
155     Node* newNode = create_node(key);
156     if (comp(key, current->key)) {
157         current->left = newNode;
158     } else {
159         current->right = newNode;
160     }
161     size++;
162
163     // Retrace the path to update heights and balance
164     while (!path.empty()) {
165         Node* node = path.back();
166         path.pop_back();
167         node = balance(node);
168
169         if (!path.empty()) {
170             if (path.back()->left == node) {
171                 path.back()->left = node;
172             } else {
173                 path.back()->right = node;
174             }
175         } else {
176             root = node;
177         }
178     }
179 }
180
181 public:
182     AVLSet() : root(nullptr), size(0), comp(Compare()) {}
183
184     // Move operations

```

```

185     AVLSet(AVLSet&& other) noexcept
186         : root(other.root),
187           size(other.size),
188           node_storage(std::move(other.node_storage)),
189           free_nodes(std::move(other.free_nodes)) {
190         other.root = nullptr;
191         other.size = 0;
192     }
193
194     AVLSet& operator=(AVLSet&& other) noexcept {
195         if (this != &other) {
196             clear();
197             root = other.root;
198             size = other.size;
199             node_storage = std::move(other.node_storage);
200             free_nodes = std::move(other.free_nodes);
201             other.root = nullptr;
202             other.size = 0;
203         }
204         return *this;
205     }
206
207     // Disable copy operations
208     AVLSet(const AVLSet&) = delete;
209     AVLSet& operator=(const AVLSet&) = delete;
210
211     void clear() {
212         node_storage.clear();
213         free_nodes.clear();
214         root = nullptr;
215         size = 0;
216     }
217
218     bool empty() const {
219         return size == 0;
220     }
221
222     size_t get_size() const {
223         return size;
224     }

```

```

225
226     template <typename Func>
227     void traverse_in_order(Func f) const {
228         traverse_in_order(root, f);
229     }
230
231     void merge(AVLSet&& other) {
232         if (other.empty()) return;
233
234         if (size < other.size) {
235             // Swap to merge smaller into larger
236             std::swap(root, other.root);
237             std::swap(size, other.size);
238             std::swap(node_storage, other.node_storage);
239             std::swap(free_nodes, other.free_nodes);
240         }
241
242         // Insert all elements from the smaller tree (now 'other')
243         // into this
244         other.traverse_in_order([this](const T& key) {
245             this->insert_merge(key);
246         });
247
248         other.clear();
249     }
250
251     private:
252     // Example of modified insert implementation
253     Node* insert(Node* node, const T& key) {
254         if (!node) {
255             size++;
256             return create_node(key);
257         }
258
259         if (comp(key, node->key)) {
260             node->left = insert(node->left, key);
261         } else if (comp(node->key, key)) {
262             node->right = insert(node->right, key);
263         } else {
264             return node;
265         }

```

```

264     }
265
266     return balance(node);
267 }
268
269 // Example of modified remove implementation
270 Node* remove(Node* node, const T& key) {
271     if (!node) return nullptr;
272
273     if (comp(key, node->key)) {
274         node->left = remove(node->left, key);
275     } else if (comp(node->key, key)) {
276         node->right = remove(node->right, key);
277     } else {
278         // Node deletion with recycling
279         if (!node->left || !node->right) {
280             Node* temp = node->left ? node->left : node->right
281                 ;
282             recycle_node(node);
283             size--;
284             node = temp;
285         } else {
286             Node* temp = findMin(node->right);
287             node->key = std::move(temp->key);
288             node->right = remove(node->right, temp->key);
289         }
290     }
291
292     return node ? balance(node) : nullptr;
293 }
294
295 public:
296 void insert(const T& val){
297     insert(root, val);
298 }
299 void remove(const T& val){
300     remove(root, val);
301 }
302 };

```

## References

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