Exploring and Comparing Different Merging Algorithms for Balanced Binary Search Trees

December 8, 2025

IB Computer Science (Group 4) Extended Essay
Word Count: 3179

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 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 or disjoint set unions (DSU).

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. The merging of some data structures like linked lists, DSUs or heaps (especially leftist heaps) is simple and straightforward. However, these easy-to-merge data structures can lack of properties like random accessing (for linked lists or heaps) or ordered property (for DSUs). Balanced BSTs, however, supports a wide range of operations in just $O(\log n)$ time, and is widely used in theoretical researches, competitions and real life applications.

This essay will focus on investigating the theoretical time complexity (usually described using big-O notation, showing the asymptotic upper bound (Cormen et al., 2022)) and actual performance of merging algorithms of balanced BSTs, which are one of the most common and powerful 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: $\forall u, v \in X, (u \le v \wedge v \le u) \Leftrightarrow u = v.$
2. Totality: $\forall u, v \in X, u \le v \vee v \le u.$
3. Transitivity: $\forall u, v, w \in X, (u \le v \wedge v \le w) \Rightarrow u \le w.$

We say $P = (X, \le)$ is a total order. For example $P = (\mathbb{R}, \le)$, where $\le$ is numerical comparison, is a total order. The set of finite strings and lexicographical order comparison is also a total order. But $P = (\{S : S \subset \mathbb{R}\}, \subset)$ (i.e. all the real number sets and the subset relation) 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 acyclic graph.
  - The *root* of a tree is a designated vertex that has no parent. For every other vertex, exactly one of its neighbors is its parent. It can be easily proven that once the root is chosen, the tree is determined.
  - The *children* of a vertex are all its neighbors except for its parent. It can be easily proven that the parent of all the children of a vertex is the vertex itself.
  - The *leaf* of a tree is a vertex that has no children.

  - The *subtree* of a vertex v is the union of v and the subtree of all its children, or only the vertex itself in the case that v is a leaf.
  - The *depth* of a tree or a subtree is the maximum number of edges on the simple path from the root to one of the leaves.

- 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, scapegoat trees and red-black trees.

In this essay, we will focus on AVL trees (named after Adelson-Velsky and Landis). 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]. AVL trees are chosen because their properties make it easy to implement and analyze. AVL trees are known to have simple, straightforward balancing condition. Unlike splay or treap, each operation on an AVL tree have a predictable time complexity for every operation, as the shape of the tree is relatively static. All the rebalance operations happen locally, and no case-heavy analysis (like in red-black tree) or amortized operations (like in splay tree) are needed.

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 $n \geq m$.

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

[1] We assume the maximum depth difference is 1 in this essay, which might not be the case in early studies.

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 an 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 $T_2$
procedure InsertionBasedMerge($T_1, T_2$)
    for all elements $x$ in $T_2$ (in-order traversal) do
        $T_1$. Insert($x$)
    return $T_1$
```

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($T_1, T_2$)
    $A_1 \leftarrow$ InOrderTraversal($T_1$)
    $A_2 \leftarrow$ InOrderTraversal($T_2$)
    $A \leftarrow$ MergeSortedArrays($A_1, A_2$)
    $T \leftarrow$ 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 negligible 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, Stockmayer and Yao has proven that in term of number of comparisons, 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 redundant 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 $T_2$
        while $successor$ not empty and $key(x) > key(\text{top}(successor))$ do
            repeat
                pop($path$)
            until top($path$) = top($successor$)
            pop($successor$)
        $p \leftarrow \text{top}(path)$
        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
                    $p.right \leftarrow x$; break
                else
```

```
                    $p \leftarrow p.right$
            push($p$, $path$)
        while $path$ not empty do
            $s \leftarrow$ 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,

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

$$\begin{aligned}
\log(\binom{n+m}{n}) &= \log(\frac{(n+m)!}{n!}) \quad (4)\\
&= (n+m)\log(n+m) - n\log n - m\log m + O(\log(n+m)) \quad (5)\\
&= n\log(1+\frac{m}{n}) + m\log(1+\frac{n}{m}) + O(\log(n+m)) \quad (6)
\end{aligned}$$

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 comparison cost, which is assumed to be $O(1)$ in this case.

We can also consider two extreme cases, when $m \approx n$, and when $m << n$. In the first case, as m approaches n, the time complexity of the algorithm will become $O(n\log(1+1)) = O(n)$. This is significantly smaller than the insertion-based algorithm, which is $O(m\log(n)) \approx O(n\log n)$ and the same as the in-order traversal algorithm's $O(m+n)$, which is $O(n+n) = O(n)$ in this case. In the second case, as m approaches $O(1)$ the time complexity is $O(\log n)$, which is the same as the insertion-based algorithm and significantly more optimal than the in-order traversal algorithm.

The theorem and the extreme-case analysis both showed that Tarjan and Brown's algorithm is the optimal algorithm for merging two instances of ordered data structures.

3 Hypothesis

The cost for all the algorithms should increase as n increases. Insertion-based algorithm perform well when α is small while in-order transversal algorithm performs well when α is large. Brown and Tarjan's algorithm is the optimal algorithm for merging two instances of ordered data structures in general cases.

4 Experiment Design

4.1 Variables

4.1.1 Independent variables

The independent variables of this experiment are:

- $\alpha = \frac{n}{m} \in \{2^0, 2^3, 2^6, 2^9, 2^{12}\}$.
- $n \in \{2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, 2^{20}\}$.

It can be found that $m = \frac{n}{\alpha}$. 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 compared to m. We are more interested in these two properties, rather than the scale of one particular part of data.

α and n are both uniformly chosen in the log scale, since it can better help us investigate performance of the algorithm in varying data sizes. The change is expected to be insignificant if they were to be chosen in the linear scale.

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-complexity 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 comparisons, helping us to analyze the performance when the comparison is not actually $O(1)$ (e.g. lexicographical order can be more costly to compare.)

4.1.3 Controlled Variables

The test environment is controlled, as is listed in the Appendix

4.2 Procedure

The procedure of the experiment is as follows:

Testing Procedure

```
Require: Sets of parameter values $A$ (for $\alpha$) and $N$ (for $n$).
Ensure: A CSV file results.csv containing time and comparison
        counts for each test.
procedure TEST($A, N$)
    Initialize random seed based on current time.
    Open results.csv and write header.
    for all Algorithm $F$ in {"Insertion-based", "In-order traversal",
    "Brown and Tarjan's"} do
        for all $\alpha \in A$ do
            for all $n \in N$ do
                for trial $\leftarrow$ 1 to 20 do
                    Randomly generate two AVL trees $S_1$ and $S_2$ of
                    sizes $n$ and $n/\alpha$.
                    Record time $t_1$.
                    $S_1 \leftarrow F(S_1, S_2)$
                    Record time $t_2$ and $\Delta t \leftarrow t_2 - t_1$.
                    Record number of comparisons.
                    Append $(F, \alpha, n, \text{trial}, \Delta t, \text{comparisons})$ to
                    results.csv.
    Close results.csv.
```

5 Results

5.1 Raw data

The output is `seed = 524042979266800` and a 2701 row csv file. The output is too redundant to be shown here, and part of it is shown in Table 1.

The result is reproducible with the same seed since the random number generator (*std::mt19937*) is deterministic (*std::mersenne_twister_engine — cppreference.com*, 2024).

Table 1: Raw data (part of)

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

5.2 Processed data

The processed data is also too redundant to be shown in this table, and part of it is shown in Table 2 and Table 3. To analyze the effect of α and n of the cost of the algorithm respectively, the cost is plotted against α and n, with the other parameters fixed, respectively in Figure 1 and Figure 3.

Table 2: Processed data for $\alpha = 64$ (time and comparisons vs. n).

n	Method	Time [ms]	Comparisons
4096	Insertion-based	0.022 ± 0.005	1154 ± 30
8192	Insertion-based	0.048 ± 0.013	2492 ± 42
[. . .]			
524288	Insertion-based	7.970 ± 1.105	233756 ± 486
1048576	Insertion-based	16.645 ± 2.057	492018 ± 375
4096	In-order traversal	0.369 ± 0.211	4121 ± 84
8192	In-order traversal	0.617 ± 0.198	8269 ± 123
[. . .]			
524288	In-order traversal	52.595 ± 5.530	532392 ± 128
1048576	In-order traversal	101.494 ± 6.763	1064745 ± 107
4096	Brown & Tarjan's	0.030 ± 0.003	699 ± 22
8192	Brown & Tarjan's	0.084 ± 0.081	1416 ± 40
16384	Brown & Tarjan's	0.133 ± 0.054	2830 ± 79
32768	Brown & Tarjan's	0.269 ± 0.061	5688 ± 92
65536	Brown & Tarjan's	0.645 ± 0.205	11382 ± 142
131072	Brown & Tarjan's	1.469 ± 0.553	22810 ± 145
262144	Brown & Tarjan's	3.519 ± 1.226	45675 ± 286
524288	Brown & Tarjan's	7.283 ± 0.823	91383 ± 384
1048576	Brown & Tarjan's	14.626 ± 1.152	182779 ± 555

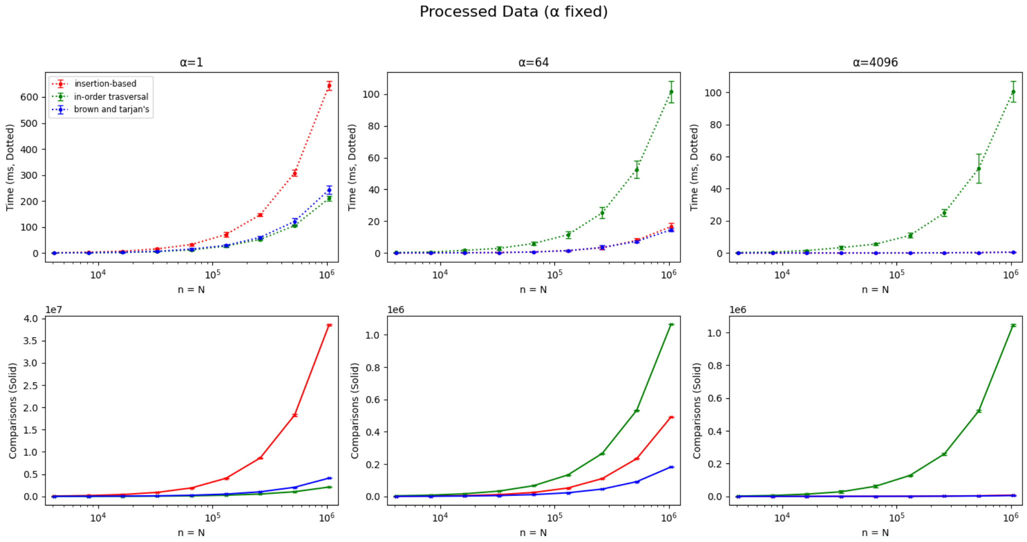

Figure 1: Processed data (α fixed)

Table 3: Processed data for $n = 65536$ (time and comparisons vs. α).

α	Method	Time [ms]	Comparisons
1	Insertion-based	33.072 ± 3.972	1901789 ± 15394
8	Insertion-based	4.135 ± 0.998	207622 ± 4350
64	Insertion-based	0.712 ± 0.313	24630 ± 132
512	Insertion-based	0.111 ± 0.036	3073 ± 46
4096	Insertion-based	0.019 ± 0.007	386 ± 20
1	In-order traversal	12.241 ± 2.432	131068 ± 4
8	In-order traversal	6.462 ± 1.191	73722 ± 6
64	In-order traversal	5.985 ± 1.271	66486 ± 174
512	In-order traversal	5.822 ± 1.712	65059 ± 1384
4096	In-order traversal	5.536 ± 0.600	61662 ± 7352
1	Brown & Tarjan's	15.597 ± 3.442	256442 ± 246
8	Brown & Tarjan's	2.729 ± 0.518	56023 ± 258
64	Brown & Tarjan's	0.645 ± 0.205	11382 ± 142
512	Brown & Tarjan's	0.151 ± 0.080	1991 ± 50
4096	Brown & Tarjan's	0.027 ± 0.013	312 ± 15

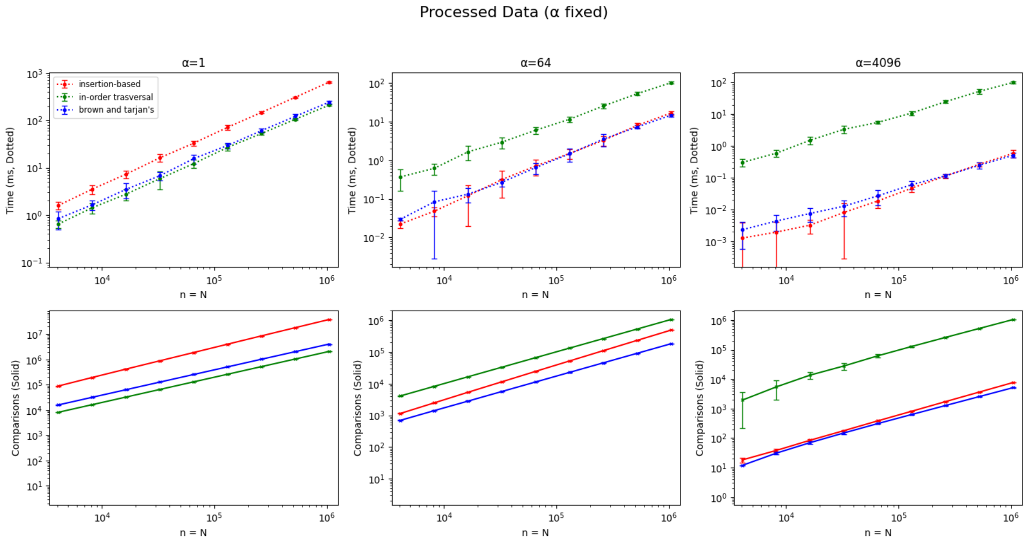

Figure 2: Processed data (α fixed) (both axis on log scale)

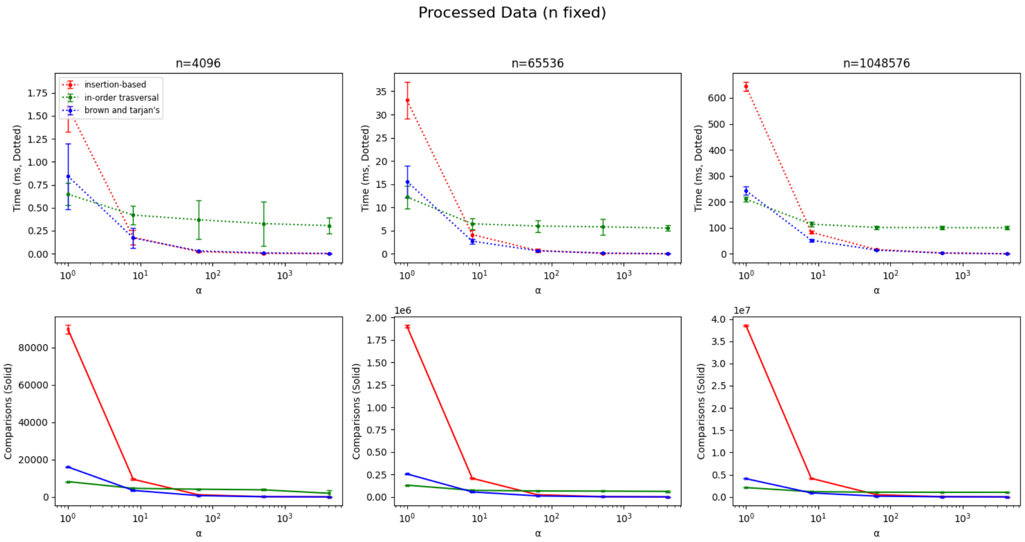

Figure 3: Processed data (n fixed)

6 Discussion and Conclusion

6.1 Correlation between n and cost

We can find clear trends from the graphs. In Figure 1, we can see that the cost of the algorithm all increases as the total data size n increases. It can be noticed that if we draw both axes in logarithmic scale, as is shown in Figure 2, the linear trend is more obvious. When α is fixed, the cost of three algorithms all increases linearly with the total data size n.

However, their rate of increase differs when α is different. For experiment where $\alpha = 1$, which means the sizes of both tree are the same, we can notice that both the time cost and the number of comparison of insertion-based algorithm is significantly higher than the other two algorithms, shown by a clear skyrocketing trend in the graph.

It can be also noticed that the despite having very significant disadvantage in number of comparison, the insertion based algorithm does not perform as bad as in the time. This might be because iteration through the smaller tree, which is obviously costly in this case as the tree takes up half the data, significantly costs time but not comparison.

On the contrary, for experiment where $\alpha = 64$, which means a small tree is being inserted to a significantly larger tree, the in-order traversal algorithm become the most costly, with both the time cost and number of comparisons skyrocketing. It can also be noticed that Brown and Tarjan's algorithm have smaller number of comparisons than the insertion-based algorithm, but similar cost in time. This might be because the cost of maintaining the path is quite significant. This implies if the cost of comparison is larger, like comparing the lexicographical order of the strings, Brown and Tarjan's algorithm's advantage will be more obvious. These trends are even more significant when $\alpha = 4096$.

There is also abnormal large error bar when both axes are drawn in log scale, which results from the exaggeration of the error bar at small values, since $\log x$ is more steep than x when x is small.

6.2 Correlation between α and cost

.

We can also find the advantage area of each algorithm in Figure 3. As is shown in the graph, insertion-based algorithm shows the least consistent performance when α differs, with significant higher cost when $\alpha = 1$ and have lower cost when α is bigger. In-order transversal algorithm performs the best when α is small, but the cost reduces very slowly when α is bigger. This make it the algorithm with the most consistent performance. Brown and Tarjan's algorithm performs slightly inferior to in-order transversal algorithm when α is small, and performs almost as well as the insertion-based algorithm when α is large. This made it the best algorithm overall. It can be also noticed that three algorithms' performance is quite close when $\alpha \approx 8$.

The trend is almost the same throughout the graphs except for different scales, α is the only factor determining which algorithm is the optimal.

The error bars when n is small are significantly larger. The apparent instability for small n reflects the fact that the algorithm's asymptotic behavior is masked by constant factors and system-level effects.

7 Evaluation

The experiment is conducted successfully with sufficient data and supports our hypothesis. Test data incorporated a large number of trials and a wide range of independent variables, making the experiment more robust. The results are reproducible and consistent with the hypothesis. However, there is still room for improvement.

Limitations and improvements:

- **Costly `new` and `delete` operations**: For the simplicity of implementation, `new` and `delete` operations are used to create and free nodes in the tree. Repetitive `new` and `delete` operations are expensive, risky and inconsistent with modern C++ standard (Meyers, 2014). A better implementation can be modern pointers and/or memory pool to avoid extra time cost.

- **Cache warmup and memory fragmentation control**: First several trials may be slower due to lack of cache warmup, later trials may be slower due to memory fragmentation. A better implementation is to run several warmup trials and use memory pool to avoid memory fragmentation.

- **Regular independent variables**: To avoid floating point calculation when calculating data size, the powers of 2 are used as the sizes of the sequence generated. However, this results in a complete binary tree after tree construction, which is not always the case in practice. A better implementation is to use more independent variables and generate them randomly.

- **O(1) comparison of `int`**: Only integer values whose comparison is $O(1)$ are put into the binary search trees. In many cases, the comparison can be more costly like in lexicographical order, floating point number comparison, big numbers or their ratios, etc. In these cases, the cost of logic or memory operations can be more insignificant compared to the cost of comparison, which means reducing the number of comparison is of greater priority. This can result in significantly different performance under the same data scale, as is discussed before. It would be better if more data types other than integer are used.

- **Fully shuffled dataset**: Only cases where the data is fully shuffled are considered. In many cases, the data is not purely random. Some algorithm may run well for random data but not every trial, like treap without random priority. When merging two instances of ordered data structures, elements in one instance may be significantly smaller than the other, in which case the performance can be different. Cases like this should be investigated as well.

- **Limited trials and independent variable range**: The experiment is conducted with a limited number of trials and a limited range of independent variables, due to the time limit and poor performance of the device. Some properties that can only be found with large n or certain dataset may not be investigated. It would be better if more trials and larger independent variable ranges are used with a better device.

- **No multithreading considered**: This essay did not discuss possible multithreading improvement of the algorithms.

Acknowledgements

ChatGPT (OpenAI, 2025) was used to generate an initial C++ code skeleton that aligns with the modern C++ standard. All analysis, implementation, and algorithmic development and testing were performed by me.

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.

Meyers, S. (2014). *Effective Modern C++: 42 specific ways to improve your use of C++11 and C++14*. Retrieved from `http://cds.cern.ch/record/1953418`

std::mersenne_twister_engine — cppreference.com. (2024). `https://en.cppreference.com/w/cpp/numeric/random/mersenne\_twister\_engine`. (Last modified 21 August 2024)

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

Appendix A Test environment

Table 4: 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)
Compiling Command:	`g++ test.cpp -o test -std=c++17 -O0 -Wall`
Interpreter:	Python 3.12.2 (graphing)

Appendix B Algorithm implementation

Listing 1: avl_set.h

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

```cpp
#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));
    }

    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;
    }

    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;
```

```cpp
            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.
    */

    void linearmerge(AVLSet&& other) {
        if (other.empty()) return;
        std::vector<T> q1 = items();
        std::vector<T> q2 = other.items();
        std::vector<T> all_elements;
        all_elements.reserve(q1.size() + q2.size());
        std::merge(q1.begin(), q1.end(), q2.begin(), q2.end(),
                 std::back_inserter(all_elements), comp_);
        delete_tree(root);
        delete_tree(other.root);
        root = nullptr;
        other.root = nullptr;
        root = build(all_elements.begin(), all_elements.end());
        size = all_elements.size();
    }

    /**
     * Merge two sets in O(M log(N)) 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); }
        other.traverse_in_order([this](const T& key) {
            this->insert(key);
        });
        other.clear();
    }

    /**
     * Merge two sets in O(M log(1+N/M)) time.
     * @param other The AVLSet to be merged into this set.
     */
    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) {
            while (!successor.empty() && !comp_(x, (*successor.back())->key)) {
                Node** succLink = successor.back();
                while (!path.empty() && path.back() != succLink) path.pop_back();
                successor.pop_back();
            }

            Node** curLink = path.empty() ? &root : path.back();
            Node* p = *curLink;
            if (!p) {
                *curLink = create_node(x);
                path.push_back(curLink);
            } else {
                for (;;) {
                    path.push_back(curLink);
                    if (comp_(x, p->key)) {
                        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 {
                        if (p->right == nullptr) {
                            p->right = create_node(x);
                            path.push_back(&(p->right));
                            break;
                        } else {
                            curLink = &(p->right);
                            p = *curLink;
                        }
                    }
                }
            }
            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);
                    *link = newsub;
                    while (!path.empty() && path.back() != link) path.pop_back();
                    break;
                }

                if (bf == 0) {
                    break;
                }
            }
        }
        #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
    }

};
```

Listing 2: test.cpp

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <fstream>
#include "avl_set.h"

typedef std::shared_ptr<std::size_t> CounterPtr;

struct SharedCountingComp {
    CounterPtr counter;

    SharedCountingComp() : counter(std::make_shared<std::size_t>(0)) {}
    SharedCountingComp(CounterPtr c) : counter(std::move(c)) {}

    bool operator()(int a, int b) const {
        ++(*counter);
        return a < b;
    }
    size_t getcount() const { return *counter; }
};

const std::string methodnames[] = {"Insertion-based", "In-order trasversal", "Brown and Tarjan’s"};

std::tuple<double, size_t> testmerge(int type, std::mt19937_64& rnd, size_t q1 = 5e5, size_t q2 = 5e5) {
    CounterPtr sharecounter = std::make_shared<std::size_t>(0);
    auto comp = SharedCountingComp(sharecounter);
    std::vector<int> vec(q1 + q2);
    iota(vec.begin(), vec.end(), 0);
    std::shuffle(vec.begin(), vec.end(), rnd);
    auto s1 = AVLSet<int, SharedCountingComp>(comp), s2 = AVLSet<int, SharedCountingComp>(comp);
    sort(vec.begin()+0, vec.begin()+q1);
    s1.construct(vec.begin() + 0, vec.begin() + q1);
    sort(vec.begin() + q1 + 1, vec.begin() + q1 + q2);
    s2.construct(vec.begin() + q1 + 1, vec.begin() + q1 + q2);
    const auto t1 = std::chrono::steady_clock::now();
    if(type == 1)
        s1.simplemerge(std::move(s2));
    else if (type == 2)
        s1.linearmerge(std::move(s2));
    else if (type == 3)
        s1.brownmerge(std::move(s2));
    const auto t2 = std::chrono::steady_clock::now();
    double ms = std::chrono::duration<double, std::milli>(t2 - t1).count();
    return std::make_tuple(ms, s1.comparator().getcount());
}

int main() {
    auto seedx = std::chrono::steady_clock::now().time_since_epoch().count();
    std::mt19937_64 rnd(seedx);

    std::ofstream outfile("results.csv");
    outfile << "method,alpha,N,trial,time_ms,comparisons\n"; // CSV header

    for (int type : {1, 2, 3}) {
        for (int alpha : {1, 1<<3, 1<<6, 1<<9, 1<<12}) {
            for (long long n : {1<<12, 1<<13, 1<<14, 1<<15, 1<<16, 1<<17, 1<<18, 1<<19,
                 1<<20}) {
                for (int trial = 1; trial <= 20; trial++) {
                    auto [time, comp] = testmerge(type, rnd, n, n / alpha);
                    outfile << methodnames[type - 1] << ","
                            << alpha << ","
                            << n << ","
                            << trial << ","
                            << time << ","
                            << comp << "\n";
                }
            }
        }
    }

    outfile.close();
    std::cout << "seed = " << seedx << std::endl;
    return 0;
}
```