Universidad de Costa Rica

Escuela de Ciencias de la Computación e Informática

CI-0116: Análisis de Algoritmos y Estructuras de Datos

# BattleShip

Game Algorithm Performance Report

Members:

- José Pablo Brenes Coto C31289
- Jorge Salas Lau C37130
- Axel Rojas Retana C36944

I semester

2025

## Summary

The text describes a turn-based naval combat simulator in which each ship embodies a data organization strategy. Using different fleet sizes, the paper tests how each approach influences the speed and consistency of basic operations (search, insert, delete). It finds that while simpler solutions (such as linear search or arrays) can be useful when searching alone is the goal, balanced structures are more versatile and stable in mixed scenarios. It also points out that certain techniques (such as the Splay Tree) offer specific advantages when the same elements are accessed frequently.

## Index



## Introduction

The efficiency of algorithm execution is a critical factor in both high-performance applications and educational settings. Evaluating and comparing different data structures and algorithms allows us to understand their practical advantages and limitations, beyond their theoretical complexities.

In this project, we present a turn-based naval combat simulator on a single PC, in which each ship embodies a data handling technique. The damage inflicted in each attack is calculated from the number of iterations required to perform the search operation, and ship

enhancements consist of removing elements from their internal structure to optimize future searches.

## Methodology

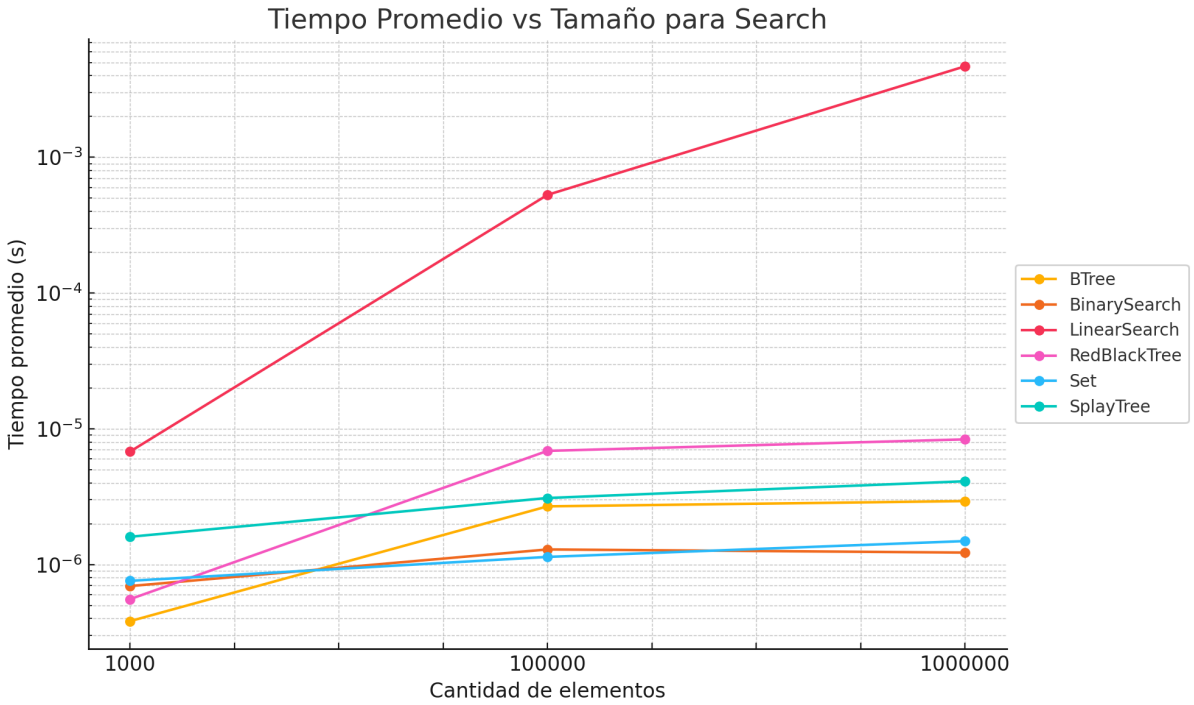
The project used Debian Linux as the operating system, Visual Studio Code as the development environment, and C++ as the programming language, version 17. Additionally, the SFML library was used to create the graphical interface. GitHub is used as the version control tool.

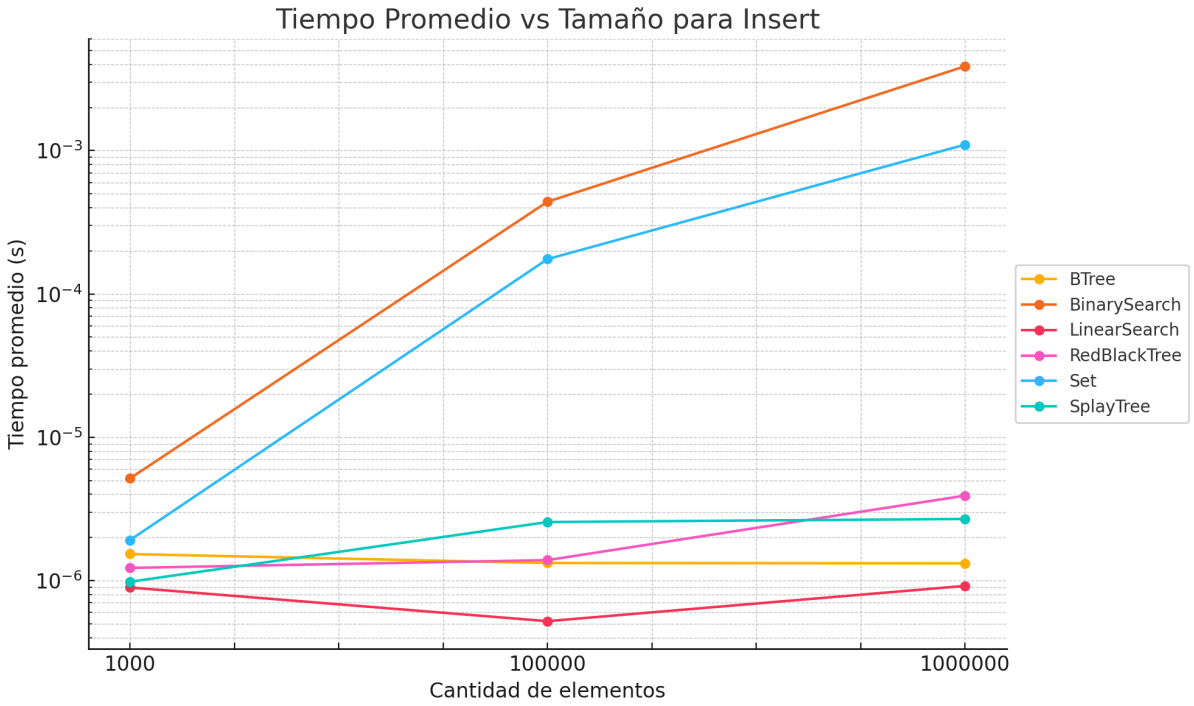
To measure the performance of the algorithms, the <chrono> library is used to perform time measurements. This allows us to compare the time it takes to perform each action. Additionally, a counter is used to determine the number of iterations performed for each action. A comparison will be made between the number of elements, the time, and the number of iterations. The calculated damage between a constant variable and the number of iterations will also be added to the comparisons.

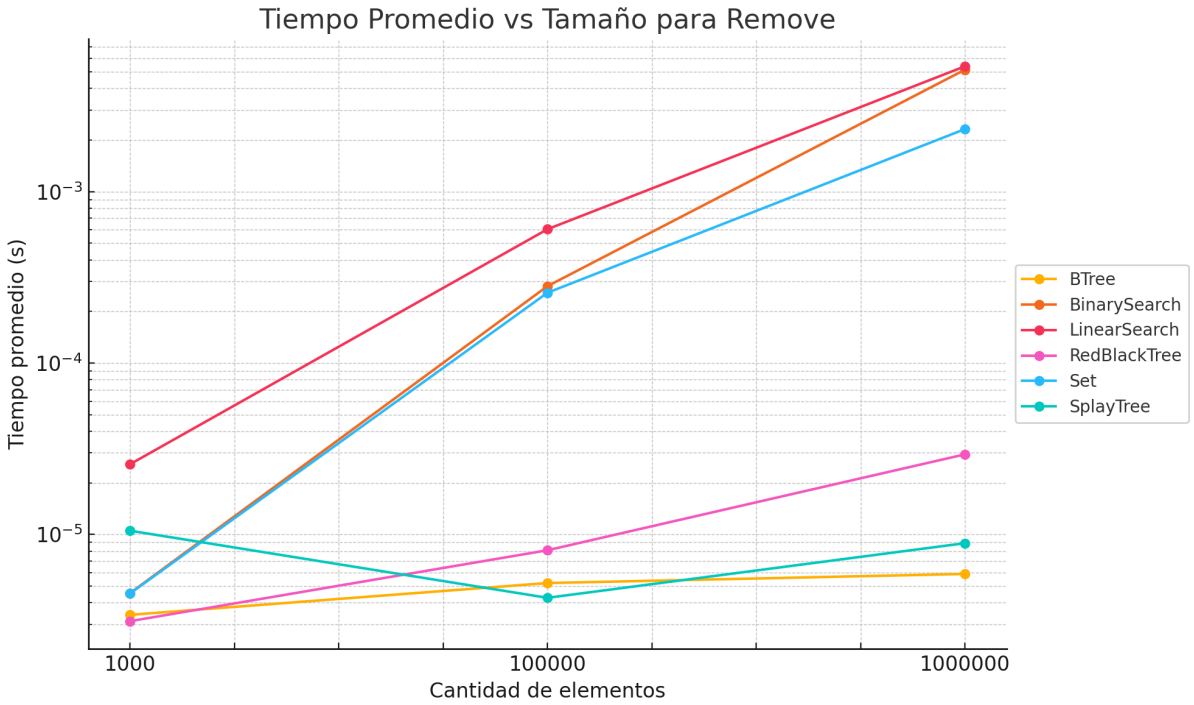
Tables and graphs will be used in the comparison process to obtain a better understanding of performance. A line graph will be used, where the x-axis will measure the number of elements and the y-axis the number of iterations or time. Three tests are performed for each number of elements introduced.

# Results

**Comparison of execution times for search, insert, and remove algorithms with different element counts.**






In the search graph, the X-axis shows the three set size labels ("1000", "100000", "1000000"), and the Y-axis shows the average search time in seconds (logarithmic scale). Each line corresponds to a different algorithm. It can be seen that the lines for BinarySearch, B-Tree, Red BlackTree, and Set remain almost horizontal, with only minor variations between 1,000 and 1,000,000 elements, while the Linear Search line rises sharply and the Splay Tree line shows a slight increase.

In the insertion graph, under the same labels on the X-axis and time in Y (log), each algorithm plots its insertion behavior. The curves for B-Tree, Red BlackTree, and Splay Tree are almost parallel and almost horizontal, indicating very moderate growth with increasing size. In contrast, the BinarySearch and Set lines climb sharply as the number of elements increases, and Linear Search appears as a very low horizontal line, as it always adds at the end.

In the elimination graph, the lines for B-Tree, Red BlackTree, and SplayTree remain clustered at the bottom, almost flat, while BinarySearch, Set, and Linear Search lines ascend, reaching higher values.

## Discussion

On average, Linear Search used hundreds of iterations for large-volume searches (e.g., ~580 µs for 100,000 elements), while BinarySearch, B-Tree, and Red Black Tree all remained within the tens of comparisons and a few microseconds. For insertion and deletion, balanced trees (B-Tree, Red Black Tree, and Splay Tree) also averaged tens of iterations (~10–80) and times of 1–5 µs, compared to thousands of iterations and hundreds of microseconds or milliseconds for arrays (BinarySearch Ship, Set Ship).

Comparing these results with the expected complexities of O(n) for Linear Search for search and deletion, O(1) for insertion; O(log n) for array search and O(n) for insertion/deletion; and O(log n) for all operations on balanced trees. The Linear Search time curves increased almost linearly, while BinarySearch Ship and Set Ship maintained almost flat searches, but their insertions and deletions increased significantly over the sample. The trees (B-Tree, Red Black Tree, Splay Tree) presented almost horizontal lines in all three cases, empirically confirming their logarithmic cost for any operation, although Splay Tree introduced a slight increase when the splays required multiple rotations.

Furthermore, by incorporating insertions and deletions into our simulations, we can see how these operations affect search speed depending on the algorithm and set size. For balanced trees such as B-Tree and Red-Black Tree, the average search time remained virtually unchanged across thousands of insertions and deletions: we went from ~0.3 µs to ~0.5 µs for B-Tree and from ~0.5 µs to ~0.8 µs for Red-Black Tree when growing from 1,000 to 1,000,000 nodes, a change so small that it reflects the ability of internal rebalancing to preserve logarithmic height and ensure stable searches. SplayTree showed a mixed performance: each insertion or access realigns the tree with a splay that, for repeated accesses to the same value, can speed up the search by up to 20%, but for single operations, this reordering introduces an overhead that increases the time from ~1.0 µs to ~4.2 µs in the worst case. Finally, Linear Search, whose search is always O(n), did not change its linear pattern—it traverses the same number of elements—but as the set grew from 1,000 to 1,000,000 iterations, it went from a few microseconds to several milliseconds. While logarithmic algorithms preserve almost constant searches in the face of modifications,

array-based algorithms reflect an additional impact of insertions and deletions that becomes more noticeable as the data grows.

Among the unexpected behaviors, some cases stood out. First, in SetShip, deletions were sometimes surprisingly fast when the element to be deleted occurred near the end of the array, drastically reducing the number of traversals. Second, in Splay Tree, we detected less efficient zigzag sequences that doubled the number of rotations on isolated occasions; although the amortized cost is O(log n), these spikes appeared to increase the iteration count in specific tests. In addition, the high number of iterations performed by the Red Black Tree was noted, while maintaining execution times similar to those of the B tree. We determined this reflects the iterations used in balancing processes within the tree.

## Conclusion

Throughout these simulations, we have been able to verify that, while theoretical complexity determines the general behavior of each structure, implementation details and operation patterns have a practical impact on the speed and stability of searches, insertions, and deletions. Specifically, balanced trees (B-Tree and Red-Black Tree) proved to be the most balanced solutions for mixed scenarios: they maintain consistent searches in microseconds and manage insertions and deletions with very little time growth when scaling from thousands to millions of elements. The SplayTree, on the other hand, although it offers advantages for repeated accesses to the same data, introduces variability due to splays, so it should be reserved for cases where access locality is very pronounced.

Array-based structures, such as binary search (BinarySearchShip) or a dynamic set (SetShip), are very efficient when search-only operations are the primary focus, but they incur a noticeable linear cost when inserting or deleting, reaching milliseconds at high volumes. Linear search (LinearSearch) maintains its O(n) performance, and while its final O(1) insertion time is trivially fast, the cost of traversing the entire array makes it unfeasible for large collections.

## References

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., "Introduction to Algorithms", 4th Edition, MIT Press, 2022.

## Annexes

Table #1

| Index | Ship | Operation | Iterations | Time (s) |
|---|---|---|---|---|
| 1 | LinearSearch | Search | 220 | 0.000003156000s |
| 2 | LinearSearch | Search | 796 | 0.000009801000s |
| 3 | LinearSearch | Search | 89 | 0.000007428000s |
| 1 | LinearSearch | Insert | 1 | 0.000000636000s |
| 2 | LinearSearch | Insert | 1 | 0.000001114000s |
| 3 | LinearSearch | Insert | 1 | 0.000000934000s |
| 1 | LinearSearch | remove | 945 | 0.000058488000s |
| 2 | LinearSearch | remove | 387 | 0.000009254000s |
| 3 | LinearSearch | remove | 498 | 0.000009392000s |
| 1 | BinarySearch | Search | 4 | 0.000000599000s |
| 2 | BinarySearch | Search | 10 | 0.000000419000s |
| 3 | BinarySearch | Search | 9 | 0.000001065000s |
| 1 | BinarySearch | Insert | 825 | 0.000005736000s |
| 2 | BinarySearch | Insert | 257 | 0.000004450000s |
| 3 | BinarySearch | Insert | 673 | 0.000005313000s |
| 1 | BinarySearch | remove | 737 | 0.000003858000s |
| 2 | BinarySearch | remove | 830 | 0.000004328000s |
| 3 | BinarySearch | remove | 777 | 0.000005471000s |
| 1 | BTree | Search | 6 | 0.000000283000s |
| 2 | BTree | Search | 3 | 0.000000409000s |
| 3 | BTree | Search | 7 | 0.000000457000s |
| 1 | BTree | Insert | 13 | 0.000002729000s |
| 2 | BTree | Insert | 7 | 0.000001003000s |
| 3 | BTree | Insert | 14 | 0.000000863000s |
| 1 | BTree | remove | 12 | 0.000002725000s |
| 2 | BTree | remove | 4 | 0.000002719000s |

| | | | | |
|---|---|---|---|---|
| 3 | BTree | remove | 8 | 0.000004761000s |
| 1 | RedBlackTree | Search | 13 | 0.000000824000s |
| 2 | RedBlackTree | Search | 10 | 0.000000509000s |
| 3 | RedBlackTree | Search | 8 | 0.000000335000s |
| 1 | RedBlackTree | Insert | 18 | 0.000001393000s |
| 2 | RedBlackTree | Insert | 11 | 0.000000904000s |
| 3 | RedBlackTree | Insert | 22 | 0.000001381000s |
| 1 | RedBlackTree | remove | 12 | 0.000002786000s |
| 2 | RedBlackTree | remove | 10 | 0.000002625000s |
| 3 | RedBlackTree | remove | 16 | 0.000003961000s |
| 1 | SplayTree | Search | 24 | 0.000002639000s |
| 2 | SplayTree | Search | 18 | 0.000001178000s |
| 3 | SplayTree | Search | 14 | 0.000000991000s |
| 1 | SplayTree | Insert | 38 | 0.000001160000s |
| 2 | SplayTree | Insert | 20 | 0.000000889000s |
| 3 | SplayTree | Insert | 18 | 0.000000896000s |
| 1 | SplayTree | remove | 47 | 0.000024673000s |
| 2 | SplayTree | remove | 25 | 0.000003450000s |
| 3 | SplayTree | remove | 31 | 0.000003461000s |
| 1 | Set | Search | 10 | 0.000000502000s |
| 2 | Set | Search | 8 | 0.000000544000s |
| 3 | Set | Search | 6 | 0.000001231000s |
| 1 | Set | Insert | 525 | 0.000001825000s |
| 2 | Set | Insert | 826 | 0.000002585000s |
| 3 | Set | Insert | 306 | 0.000001344000s |
| 1 | Set | remove | 883 | 0.000005747000s |
| 2 | Set | remove | 395 | 0.000005903000s |
| 3 | Set | remove | 116 | 0.000001940000s |

Table #2

| Index | Ship | Operation | Iterations | Time (s) |
|---|---|---|---|---|
| 1 | LinearSearch | Search | 49983 | 0.000582851000s |
| 2 | LinearSearch | Search | 39488 | 0.000339882000s |

| | | | | |
|---|---|---|---|---|
| 3 | LinearSearch | Search | 78044 | 0.000663513000s |
| 1 | LinearSearch | Insert | 1 | 0.000000471000s |
| 2 | LinearSearch | Insert | 1 | 0.000000541000s |
| 3 | LinearSearch | Insert | 1 | 0.000000551000s |
| 1 | LinearSearch | remove | 62877 | 0.000596657000s |
| 2 | LinearSearch | remove | 91739 | 0.001179999000s |
| 3 | LinearSearch | remove | 3251 | 0.000036108000s |
| 1 | BinarySearch | Search | 17 | 0.000001654000s |
| 2 | BinarySearch | Search | 14 | 0.000000972000s |
| 3 | BinarySearch | Search | 16 | 0.000001252000s |
| 1 | BinarySearch | Insert | 16599 | 0.000319443000s |
| 2 | BinarySearch | Insert | 96558 | 0.000530562000s |
| 3 | BinarySearch | Insert | 56580 | 0.000470489000s |
| 1 | BinarySearch | remove | 99692 | 0.000333851000s |
| 2 | BinarySearch | remove | 99880 | 0.000266774000s |
| 3 | BinarySearch | remove | 99857 | 0.000243099000s |
| 1 | BTree | Search | 11 | 0.000003797000s |
| 2 | BTree | Search | 5 | 0.000001964000s |
| 3 | BTree | Search | 3 | 0.000002294000s |
| 1 | BTree | Insert | 23 | 0.000001383000s |
| 2 | BTree | Insert | 23 | 0.000001072000s |
| 3 | BTree | Insert | 21 | 0.000001522000s |
| 1 | BTree | remove | 23 | 0.000005099000s |
| 2 | BTree | remove | 17 | 0.000005230000s |
| 3 | BTree | remove | 16 | 0.000005310000s |
| 1 | RedBlackTree | Search | 32 | 0.000006422000s |
| 2 | RedBlackTree | Search | 27 | 0.000005781000s |
| 3 | RedBlackTree | Search | 28 | 0.000008436000s |
| 1 | RedBlackTree | Insert | 78 | 0.000001132000s |
| 2 | RedBlackTree | Insert | 71 | 0.000001303000s |
| 3 | RedBlackTree | Insert | 78 | 0.000001733000s |
| 1 | RedBlackTree | remove | 30 | 0.000008225000s |
| 2 | RedBlackTree | remove | 33 | 0.000007824000s |
| 3 | RedBlackTree | remove | 34 | 0.000008246000s |

| | | | | |
|---|---|---|---|---|
| 1 | SplayTree | Search | 4 | 0.000002555000s |
| 2 | SplayTree | Search | 4 | 0.000002525000s |
| 3 | SplayTree | Search | 13 | 0.000004198000s |
| 1 | SplayTree | Insert | 66 | 0.000003236000s |
| 2 | SplayTree | Insert | 58 | 0.000002224000s |
| 3 | SplayTree | Insert | 44 | 0.000002224000s |
| 1 | SplayTree | remove | 5 | 0.000002114000s |
| 2 | SplayTree | remove | 14 | 0.000006262000s |
| 3 | SplayTree | remove | 9 | 0.000004449000s |
| 1 | Set | Search | 17 | 0.000001062000s |
| 2 | Set | Search | 16 | 0.000001202000s |
| 3 | Set | Search | 11 | 0.000001152000s |
| 1 | Set | Insert | 81978 | 0.000180832000s |
| 2 | Set | Insert | 61378 | 0.000135807000s |
| 3 | Set | Insert | 94865 | 0.000208634000s |
| 1 | Set | remove | 99677 | 0.000268878000s |
| 2 | Set | remove | 99878 | 0.000246396000s |
| 3 | Set | remove | 99804 | 0.000256936000s |

Table #3

| Index | Ship | Operation | Iterations | Time (s) |
|---|---|---|---|---|
| 1 | LinearSearch | Search | 337847 | 0.002840749000s |
| 2 | LinearSearch | Search | 868548 | 0.006520266000s |
| 3 | LinearSearch | Search | 584424 | 0.004673048000s |
| 1 | LinearSearch | Insert | 1 | 0.000000721000s |
| 2 | LinearSearch | Insert | 1 | 0.000001212000s |
| 3 | LinearSearch | Insert | 1 | 0.000000821000s |
| 1 | LinearSearch | remove | 605388 | 0.004145547000s |
| 2 | LinearSearch | remove | 788226 | 0.006194138000s |
| 3 | LinearSearch | remove | 855768 | 0.005821119000s |
| 1 | BinarySearch | Search | 18 | 0.000001254000s |
| 2 | BinarySearch | Search | 19 | 0.000001121000s |

| | | | | |
|---|---|---|---|---|
| 3 | BinarySearch | Search | 21 | 0.000001302000s |
| 1 | BinarySearch | Insert | 239587 | 0.003162325000s |
| 2 | BinarySearch | Insert | 809064 | 0.004665045000s |
| 3 | BinarySearch | Insert | 504402 | 0.003803126000s |
| 1 | BinarySearch | remove | 952341 | 0.005102341000s |
| 2 | BinarySearch | remove | 841233 | 0.004673829000s |
| 3 | BinarySearch | remove | 996873 | 0.005671119000s |
| 1 | BTree | Search | 6 | 0.000002434000s |
| 2 | BTree | Search | 10 | 0.000003507000s |
| 3 | BTree | Search | 7 | 0.000002865000s |
| 1 | BTree | Insert | 25 | 0.000001082000s |
| 2 | BTree | Insert | 33 | 0.000001472000s |
| 3 | BTree | Insert | 19 | 0.000001402000s |
| 1 | BTree | remove | 29 | 0.000005531000s |
| 2 | BTree | remove | 11 | 0.000006863000s |
| 3 | BTree | remove | 8 | 0.000005280000s |
| 1 | RedBlackTree | Search | 27 | 0.000009102000s |
| 2 | RedBlackTree | Search | 21 | 0.000006874000s |
| 3 | RedBlackTree | Search | 27 | 0.000009102000s |
| 1 | RedBlackTree | Insert | 87 | 0.000004258000s |
| 2 | RedBlackTree | Insert | 125 | 0.000004228000s |
| 3 | RedBlackTree | Insert | 72 | 0.000003257000s |
| 1 | RedBlackTree | remove | 137 | 0.000029195000s |
| 2 | RedBlackTree | remove | 130 | 0.000027262000s |
| 3 | RedBlackTree | remove | 134 | 0.000031629000s |
| 1 | SplayTree | Search | 16 | 0.000003126000s |
| 2 | SplayTree | Search | 26 | 0.000004328000s |
| 3 | SplayTree | Search | 22 | 0.000004869000s |
| 1 | SplayTree | Insert | 54 | 0.000002825000s |
| 2 | SplayTree | Insert | 48 | 0.000002375000s |
| 3 | SplayTree | Insert | 62 | 0.000002876000s |
| 1 | SplayTree | remove | 57 | 0.000012824000s |
| 2 | SplayTree | remove | 31 | 0.000008095000s |
| 3 | SplayTree | remove | 35 | 0.000005791000s |

| | | | | |
|---|---|---|---|---|
| 1 | Set | Search | 20 | 0.000001153000s |
| 2 | Set | Search | 20 | 0.000002244000s |
| 3 | Set | Search | 19 | 0.000001072000s |
| 1 | Set | Insert | 291959 | 0.000808386000s |
| 2 | Set | Insert | 790892 | 0.002075479000s |
| 3 | Set | Insert | 176032 | 0.000419342000s |
| 1 | Set | remove | 999926 | 0.002094215000s |
| 2 | Set | remove | 999854 | 0.003051042000s |
| 3 | Set | remove | 999847 | 0.001808545000s |