An analysis of Treap

Felipe Vaiano Calderan *

Federal University of São Paulo (UNIFESP)

July 2022

Abstract

Treap is a binary search tree that bets on the uniformity of a random distribution of items at the time of insertion. The bet is that by entering items in this way, a relatively balanced tree can be built, without the extra cost of algorithmically calculating and balancing the tree. It is also worth mentioning that treap has this name because it is a fusion of tree and heap, since the conformation of the items in the tree follows the rules of both binary search trees and binary heap trees. In this article, we will analyze the complexity and CPU time of the operations of inserting, searching and removing items from the treap tree, in addition to comparing it with other search methods, specifically linear, binary, binary tree, AVL tree search and red-black tree.

Contents

*fvcalderan@gmail.com

1 Introduction

It is extremely uncommon for data entries to be alone or completely segregated from each other. What you see in practice are data sets referring to measurements and/or enumerations on a common topic, such as: a set of students in a school, Bitcoin values in the last 6 months or the different characteristics of copper crystals. The sets can then be processed by algorithms or functions for arbitrary purposes, like visualization or prediction. This shows that sets are as relevant in computer science as they are in mathematics.

Of course, sets need to be constructed at some point, and over time the elements that make up

such sets can change through insertions and deletions. Furthermore, it is quite common that algorithms need to check the existence of certain items in the set or obtain values associated with the index demarcated by the element. Structures that serve this purpose are known as dictionaries [5].

Searching for items in sets can be the most time-consuming operation of many programs, so a myriad of algorithms has been developed over the years aiming to organize the sets so that searches are less costly. Formally, a search operation can be defined as a function $f(x)$ that searches for the key x in the set f [7]. The function then returns the element associated with the key x, if it exists, or null/error if the element is not present.

One of the most common frameworks for this purpose are binary search trees (BST). Such trees are linked items, where each internal node has 2 children: a left child, whose key value is less than the parent's key value, and a right child, whose value is greater. Children are roots for other binary lookup trees unless they are leaves. This allows for paths of nodes to be followed so that the values of the keys of the nodes get closer and closer to the searched key, similarly to how it is done in an array binary search.

The main challenge with this approach is to insert the items in a way that makes the tree as low as possible (no matter how wide). This is because the worst case scenario to fetch an element by its key is to walk from the root to the farthest leaf possible, that is, it is an operation of complexity $O(h)$. If any insertion policy is applied, there may be cases where inserting items of the set $\mathcal{N}$ in ascending/descending order results in a tree where $h = |\mathcal{N}|$ (a degenerated or pathological tree [8]).

The best-case scenario for a BST structure is for it to be perfectly balanced (minimum height possible) [6], but in practice, balanced trees (heights of the left and right subtrees of any node have a maximum difference of 1) are ideal enough. To avoid the generation of pathological or quasi-pathological trees, different policies were developed to keep the initial formation of the BST and their subsequent maintenance (insertion and removal of items) balanced. Adelson-Velsky and Landis (AVL) [9] and the Red-Black [5] trees are two very famous implementations of this idea. For every node addition and, they use logical and arithmetical procedures to recognize when rotations should be applied to the nodes to achieve structure balancing.

Unfortunately, these operations to analyze potential rotations to keep the tree balanced at all costs are not free, as they add up increasing the general complexity of the algorithm. To deal with this problem, there are algorithms that provide a compromise of structural balance and performance. One of these algorithms is the Treap, which bets on the uniformity of a random distribution of items to build a relatively balanced tree.

2 Treap

Cormen et al. [5] shows that the complexity of searching for a key in a randomly built BST is expected to be of order $O(logn)$ by proving Theorem 1. This complexity for the search operation is the same on a perfectly balanced tree. This is very interesting, but a problem exists: there is no way to guarantee that the items inputted have a randomly ordered key, and even if you could shuffle the initial insertions, the subsequent insertions may not be random.

Theorem 1 *The expected height of a randomly built BST on n keys is* $O(logn)$

To solve this problem, Aragon and Seidel [1] proposed the Treap (a portmanteau of tree and heap). It is a binary tree where each node typically stores everything a binary tree search would store (pointers to children and an item with some content and a key), but also a priority value. This priority value is assigned at insertion time, so the risk of losing the randomness is not present. The nodes are organized in a BST fashion in relation to their keys, but the priority values follow a heap pattern, that is, considering a min-heap, the children will always have smaller priority values than their parent. Figure 1 exhibits an example of a Treap.

When inserting or removing nodes, the procedure is identical to the standard BST, but the algorithm needs to take the priority values into consideration, since they can end up disrespecting the heap rule.

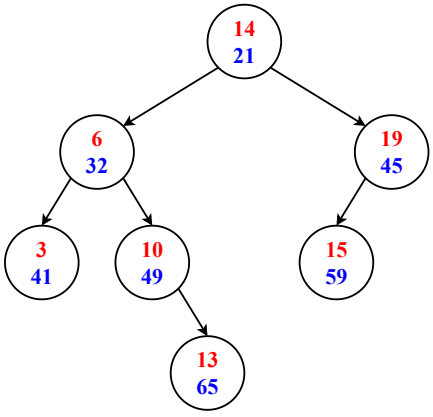

Figure 1: An example of a Treap. Red values are the keys and the blue ones are the priority. This Treap uses a Min-Heap to set up the priority values

If this happens, rotations need to be performed to correct the tree. It is also worth mentioning that, in addition to node insertion and deletion operations, the usage of Treaps allow for fast union, intersection, and difference operations on ordered sets [2], which can be an interesting feature for many applications.

The space complexity of the Treap is $O(n)$, since all the data and metadata needed are inside the nodes. The following subsections analyze the time complexity of inserting, searching for, and deleting elements from the tree. Table 1 displays a summary of best, average and worst-case scenarios complexities for these operations.

Complexity	Best	Average	Worst
Space	$O(logn)$	$O(logn)$	$O(n)$
Insertion	$O(logn)$	$O(logn)$	$O(n)$
Search	$O(logn)$	$O(logn)$	$O(n)$
Deletion	$O(logn)$	$O(logn)$	$O(n)$

Table 1: Summary of the complexities of operations on the Treap

2.1 Insertion

Inserting an item in a Treap can be thought as inserting an item in a BST and then inserting an item in a binary heap [4]. The analyses below take this fact as a basis for calculations.

The **best-case** scenario is when the tree is perfectly balanced. Both BST's and heap's insertion time complexity is $O(logn)$, since the only thing needed is to traverse the tree until the correct spot for the insertion is reached. Therefore, the complexity for inserting an element, in the best-case scenario, in a Treap is $T(n) = 2 \cdot O(logn) = O(logn)$.

Using Theorem 1, it is possible to conclude that the **average-case** scenario for the insertion operation has a complexity of $T(n) = 2O(logn) = O(logn)$, since the expected height of the BST (and heap, by consequence) is $O(logn)$. Then the same logic as above applies.

Finally, the **worst-case** scenario happens when the tree ends up being pathological, which is increasingly unlikely as the tree grows, but is always possible. To insert an element in a BST of with this structure, the complexity is $O(n)$. Although heaps are almost complete (therefore cannot be pathological), the Treap takes primarily the BST structure into consideration, so the worst-case is inherited from the BST complexity, that is, $O(n)$.

The rotations are irrelevant to the complexity, since elements can go from leaf to root in a time complexity of $O(1)$.

2.2 Search

The best, average and worst-case scenarios for searching a key is the exact same as inserting, since the cost of inserting an item is primarily dependent on the cost of traversing the BST looking for the right place to insert it, considering its key. The heap part of the logic is irrelevant for the search operation, since the heap deals with the priority values, not the keys.

2.3 Deletion

Deletion procedure works by searching for the item to be deleted by its key, if it's found, delete it and restructure the tree so that it respects Treap rules. Since deletion can be thought as the insertion operation backwards [1], the complexities for the best,

average and worst-case scenarios are, again, the same as the insertion ones.

3 Implementation

The complete code for this project (the search algorithms and the functions to execute the various tests) is too extensive to display in details here, therefore, this paper focuses on the general structure of the project and the Treap.

All the algorithms and testing procedures were written in the C language, since the resulting binary compiled by GCC runs very fast. No 3rd party libraries were used in the implementation.

3.1 Data Structures

Looking at the code and extracting the data structures in a more abstract way, we have 3 core structures:

- `TreapNode`: each node in a Treap. It contains the item (of type `TreapItem`, priority values and pointers to the children, which are `TreapNodes` themselves.
- `TreapItem`: each `TreapNode` contains an item that stores the key and content to be indexed by the tree. In the real program, there is no content, since it is irrelevant to the tests ran.
- `Dictionary`: an abstraction for the root of the Treap. It can also store metadata, if needed, like the size of the dictionary. It is superfluous in this specific case.

The naming scheme is different from the one used in the source code, for abstraction, styling, and language purposes. To avoid confusion, Table S1 is provided in the Supplementary Information describing the correspondence between the name schemes. Figure 2 shows the UML diagram related to the abstract data types described.

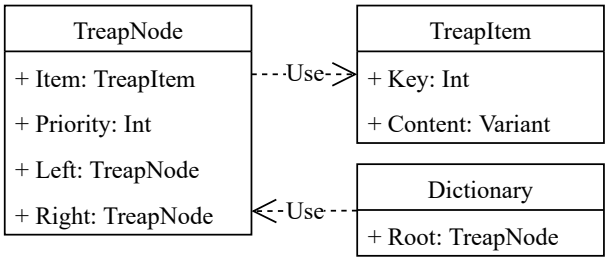

Figure 2: UML diagram for the data structures present in the Treap program

3.2 Main Function

Before discussing the main function, it is important to inform that the sorting algorithms are segregated into different files and each one of them have a main function. All the main functions are virtually the same. The project is organized this way to avoid dealing with renaming the data types for each algorithm (since they have different information in the tree nodes, for example).

The Main function takes care of the input and output of the program, as well as calling and timing the insertion, search, and deletion procedures. The input is composed of 2 pieces of information: the path of the binary file containing the integers values to populate the array of elements to be inserted/searched/deleted from the tree and the size of this array. The output is displayed in the standard output formatted as a CSV file [10]. The columns are: the algorithm name (`algorithm`), array file path (`file`), size of the array (`size`), number of comparisons when adding (`add_c`), searching for (`look_c`) and deleting (`del_c`) elements and the respective CPU times for the operations (`add_t`, `look_t` and `del_t`), respectively.

The CPU times were obtained through the use of the function `clock_t clock(void)`[1] from the `time.h` Unix library. This function returns an approximation of processor time used by the program. To find the value in seconds, the `clock_t` value is divided by the `CLOCK_PER_SEC` definition. Figure 3 shows the Main function's flowchart.

[1] `man 3 clock`

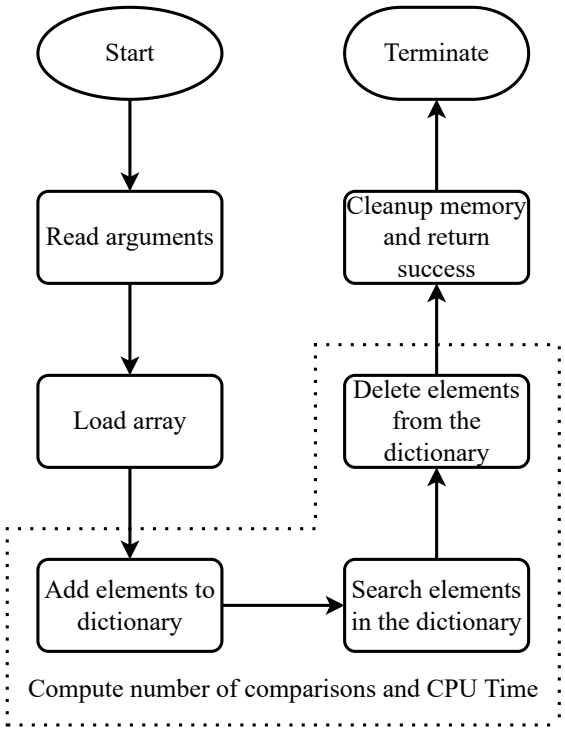

Figure 3: Main function's flowchart

3.3 Search Algorithms

In total, six different search algorithms are implemented in the project: linear, binary, binary tree, AVL tree, Red-Black tree and Treap. The insertion, search, and deletion functionalities of each algorithm are abstracted away by a Dictionary type, which has 3 core functions: insert, search and remove. This lets the program have a common interface to talk to the algorithms, no matter which one it is.

All the implementations can be found inside the repository provided in the Supplementary Information. In this paper, only the Treap implementation will be discussed, since it is the algorithm being analyzed.

The functions here follow a different name scheme from the source code provided, for the same reasons as the data structures. To avoid confusion, Table S1 is provided in the Supplementary Information describing the correspondence between the different schemes.

The implemented Treap has 6 core functions: `InitializeNode`, `LeftRotation`, `RightRotation`, `Insert Node`, `SearchNode` and `RemoveNode`. High-level versions of these functions are described here.

Algorithm 1 InitializeNode

Require: TreapNode N, TreapItem x
Ensure: N is properly initialized and contains x
  Allocate memory for N
  $N.Item \leftarrow x$
  $N.Priority \leftarrow$ Random
  $N.Left \leftarrow NIL$
  $N.Right \leftarrow NIL$
  return N

Algorithm 1 is used to create new nodes and, consequently, to create a new tree, since if the node is the first one, it is the root. It has complexity $O(1)$, because it only attributes values to the allocated TreapNode properties.

Algorithm 2 LeftRotation

Require: TreapNode N
Ensure: A left rotation is performed
  $M \leftarrow N.Right$
  $N.Right \leftarrow M.Left$
  $M.Left \leftarrow N$
  $N \leftarrow M$
  $M \leftarrow N.Left$
  return N

The `RightRotation` algorithm is identical to Algorithm 2 swapping `Right` for `Left` and vice-versa. A rotation is a very straightforward operation with time and space complexity $O(1)$, since it basically consists of pointer manipulation.

Algorithm 3 guarantees that an Item is correctly inserted into the Treap, by performing the rotations needed to make the whole tree follow the Treap rules. This algorithm has an average complexity of $O(logn)$, as discussed in Section 2.1. Even though it calls `InitializeNode` and `LeftRotation` (or `RightRotation`), those operations are $O(1)$, thus, they make no difference in asymptotic terms.

Algorithm 3 InsertNode

Require: TreapNode N, Item x
Ensure: Item x is inserted in the N-rooted tree
  if N is NIL **then**
    return InitializeNode(N)
  end if
  if $x.Key < N.Item.Key$ **then**
    $N.Left \leftarrow$ InsertNode($N.Left$, $Item.Key$)
    if $N.Left.Priority < N.Priority$ **then**
      $N \leftarrow$ RotateRight(N)
    end if
  else
    $N.Right \leftarrow$ InsertNode($N.Right$, $Item.Key$)
    if $N.Right.Priority < N.Priority$ **then**
      $N \leftarrow$ RotateLeft(N)
    end if
  end if
  return N

The Search Node function, exhibited in Algorithm 4 is the exact same as of a conventional BST. The search is performed by recursively accessing the $Left$ and $Right$ children of each node. If at any point, the node with the correct Key is found, it is returned. The search can also fail if a NIL node is reached, in which case NIL is returned. The time complexity of this function directly dictates the complexities of `InsertNode` and `RemoveNode`. It is $O(logn)$ in the best and average-case scenarios and $O(n)$ in the worst-case one.

Finally, Algorithm 5 removes an element from the Treap and has an average time complexity of $O(logn)$. This is the longest and most complex function implemented. It searches for the item to be removed, then, if it is found, the removal occurs. When there is a node removal, the tree may need to be rebalanced (rotations may need to be performed). If the wanted node is not found, NIL is returned.

Those are all the core functions that compose the Treap. This tree removes much of the complexity that AVL and Red-Black add when inserting and removing nodes. For the complexity analyses and implementation in the C language of Treap's set operation algorithms, the reader is referred to Blelloch and Reid-Miller's paper [2].

Algorithm 4 SearchNode

Require: TreapNode N, Item x
Ensure: x is returned, if found
  if N is NIL **then**
    return NIL
  end if
  if $N.Item.Key = x.Key$ **then**
    return N
  end if
  if $x.Key < N.Item.Key$ **then**
    return SearchNode($N.Left$, x)
  else
    return SearchNode($N.Dir$, x)
  end if

3.4 Array Generation Tool

In Subsection 3.2, it was mentioned that the program loads a binary file into an array. This file needs to be generated in the first place, so an array generation tool was built for this purpose. More details are available in the Comb Sort analysis paper [3].

4 Computational Environment

4.1 Computer Specifications

The environment used to compile the program and run the tests has the following specifications:

Hardware:

- **CPU:** 1 Core of AMD EPYC 7551
- **RAM:** 1GiB DIMM RAM

Software:

- **OS:** Ubuntu 20.04.4 LTS
- **KERNEL:** Linux 5.13.0-1018-oracle
- **GCC:** 9.4.0

Algorithm 5 RemoveNode

Require: TreapNode N, Item x
Ensure: x is removed, if found
  if N is NIL **then**
    return NIL
  end if
  if $N.Item.Key = x.Key$ **then**
    if $N.Left = NIL$ and $N.Right = NIL$ **then**
      return NIL
    else if $N.Left = NIL$ **then**
      return $N.Right$
    else if $N.Right = NIL$ **then**
      return $N.Left$
    else
      $NLP \leftarrow N.Left.Priority$
      $NRP \leftarrow N.Right.Priority$
      if $NLP < NRP$ **then**
        $N \leftarrow$ RotateRight(N)
        $N \leftarrow$ RemoveNode($N.Right$, x)
      else
        $N \leftarrow$ RotateLeft(N)
        $N \leftarrow$ RemoveNode($N.Left$, x)
      end if
    end if
  else if $x.Key < N.Item.Key$ **then**
    $N.Left \leftarrow$ RemoveNode($N.Left$, x)
  else
    $N.Right \leftarrow$ RemoveNode($N.Right$, x)
  end if
  return N

4.2 Array Set

The arrays generated have the following characteristics:

- 10 seeds for the random number generator
- 4 modes: ordered, inversely ordered, almost ordered and random
- 6 sizes: 10, 100, 1000, 10000, 100000 and 1000000 elements

where almost ordered arrays have 1% of their elements (when possible) at the wrong places.

The array set contains the combination of all the above characteristics, adding up to $10 \cdot 4 \cdot 6 = 240$ different arrays.

5 Tests

In this section, different charts displaying the time taken and number of key comparisons made by each algorithm are presented. The values shown in these charts are given by $T_m = I_m + S_m + D_m$, where m can be either c or t (c is the number of key comparisons and t is the CPU time taken) and I is insertions, S is searches and D is deletions. In other words, it's the sum of the 3 operations. Also, the values presented are the average between the value received from all 10 different seeds of the random number generator. Charts for each operation individually were generated and are in the Supplementary Information for this article.

5.1 CPU Time and Comparisons for each array type

5.2 Ordered

From Figures 4 and S1, it is possible to see that, for the ordered arrays, even though the Treap ends up making more comparisons than the AVL tree, Red-Black tree and binary search, it actually performs better than all of them (specially comparing with the binary search) in terms of CPU time consumption.

The competition between AVL, Red-Black and Treap is going to be a common theme here, since they all perform similarly. On the other hand, linear search, binary search and binary tree search tend to be slower.

It is worth noting that the charts are in log_10 scale, so even though in Figure 4 it seems like the binary search is not much worse than the Red-Black tree, it is orders of magnitude worse.

5.2.1 Inversely Ordered

Looking at the inversely ordered arrays (Figures 5 and S2, It is possible to see a very similar pattern to

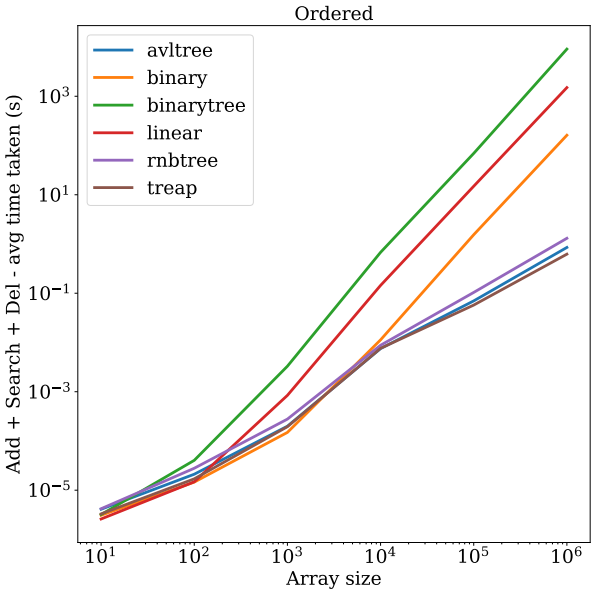

Figure 4: Average CPU time taken (Ordered arrays)

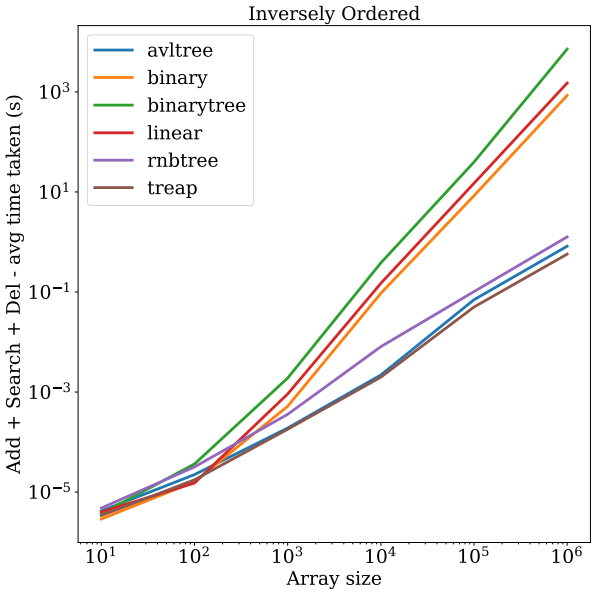

Figure 5: Average CPU time taken (Inversely ordered arrays)

ordered arrays. Treap makes slightly more key comparisons than the AVL tree, but ends up performing better in terms of CPU time (being, overall, the fastest). Again, AVL, Red-Black and Treap are very close.

5.2.2 Almost ordered

In the case of almost ordered arrays, depicted in Figures 6 and S3, Treap does not beat AVL, neither in terms of comparisons, nor in CPU time. AVL is the best in this category, but not by much. Treap and Red-Black actually switch places in relation to better and worse than one another as the size of the arrays grow. When the biggest size tested is reached, they end up extremely close, Treap being the best between the two in terms of CPU time, but worst in key comparisons.

5.2.3 Randomly Ordered

Figure 7 shows that, for completely random arrays, the best algorithm, in terms of CPU time, is actually the BST, which makes sense, since all the sets before this one lead to the worst-case scenario for this tree (they are sequential or almost sequential, therefore generate pathological trees).

In this case, the BST wins by generating trees that were sufficiently balanced (as discussed before, randomly created trees tend to automatically balance themselves) without the expense of manually balancing them every insertion or deletion.

An argument could be made that Treap should have performed just as well as the BST, since it also works with randomness. This is not the case, since, although the Treap performs pretty much the same amount of key comparisons as the BST (Figure S4), it also has the overhead of organizing the tree according to BST and Heap properties. This leads to extra workload comparing priority values, which end up increasing the CPU time. In this case, it performed worse than AVL and Red-Black, as well.

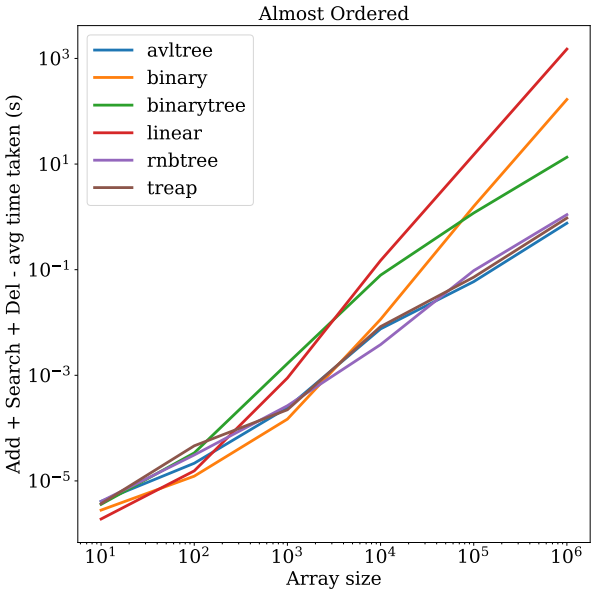

Figure 6: Average CPU time taken (Almost ordered arrays)

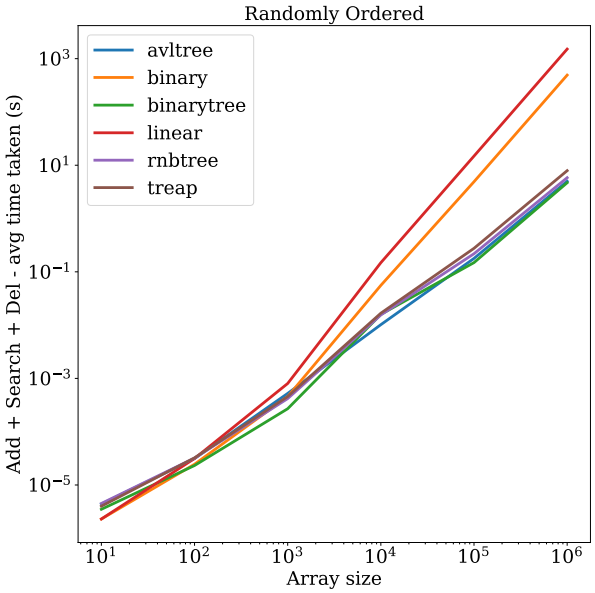

Figure 7: Average CPU time taken (Randomly ordered arrays)

5.2.4 All arrays

When looking at the general performance of the algorithms, the Treap performs extremely close (being slightly worse) to the Red-Black tree, both in number of comparisons and CPU time. The AVL tree is the overall best choice for the type of tests ran in this paper.

Analyzing Figures S6, S7 and S8 from the Supplementary Information, it is visible that the Treap does an overall worse job than the AVL tree. It does beat the Red-Black on insertion operations, but by a very small margin.

Two interesting observations are that the binary search is very efficient performing search operations, while the BST performs poorly at it, but when looking at deletion operations, it is the opposite. One other fact to be observed is that the linear search performs badly in every single test executed (it is always $O(logn)$), so this search algorithm is only recommended for very small arrays, and even then, the binary search may be an overall better choice.

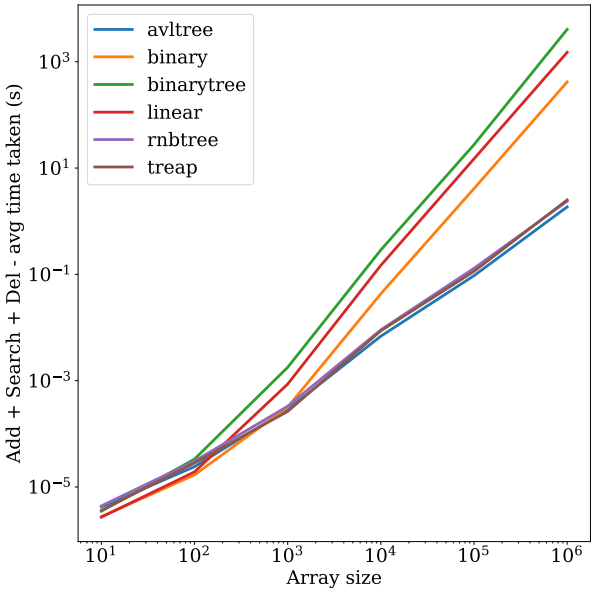

Figure 8: Average CPU time taken (All arrays)

6 Conclusion

Searching elements in a set is a very common task in computer science (since it is a part of many other larger algorithms), hence, the more optimized it is, the better. Having this in mind, many search algorithms were developed, Treap being one of them.

Treap, which is a tree with both BST and Heap properties, uses the fact that completely random distributions tend to be uniform, therefore inserting elements from a distribution like this lead to a relatively balanced tree. This saves time by not running tree-balancing operations, like AVL and Red-black trees do.

The tests ran show that the Treap beats every other algorithm tested when the arrays are ordered or inversely ordered. It also does a very good job keeping up with the best algorithms in almost ordered and randomly ordered arrays, therefore, Treap's bet on randomness is not only supported mathematically, but also empirically.

Given the results, it is recommended to use Treap for ordered and inversely ordered arrays, AVL tree for almost ordered arrays and conventional BST for randomly ordered arrays. If there is no way to know how the items are distributed beforehand, it is recommended to use the AVL tree, because it has the best performance, on average.

References

[1] Cecilia R Aragon and Raimund Seidel. Randomized search trees. In *FOCS*, volume 30, pages 540–545, 1989.

[2] Guy E Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, 1998.

[3] Felipe V Calderan. An analysis of comb sort. 2022.

[4] Igor Carpanese. A visual introduction to treap data structure (part i: The basics). *Medium*, 2020.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[6] RA Frost and MM Peterson. A short note on binary search trees. *The Computer Journal*, 25(1):158–158, 1982.

[7] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[8] Anand K. Parmar. Different types of binary tree with colorful illustrations. *Medium*, 2020.

[9] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.

[10] Y Shafranovich. Common format and mime type for comma-separated values (csv) files. 2005.

Supplementary Information

Type	Name in the paper	Name in the source code
ADT	TreapNode	TNo
ADT	TreapItem	TItem
ADT	Dictionary	TDicionario
Function	InitializeNode	TNo_Inicia
Function	LeftRotation	RotacionaEsquerda
Function	RightRotation	RotacionaDireita
Function	InsertNode	InsereRecursivo
Function	SearchNode	PesquisaRecursiva
Function	RemoveNode	RetiraRecursivo

Table S1: Name scheme translation table for the abstract data types and functions in the paper and in the source code

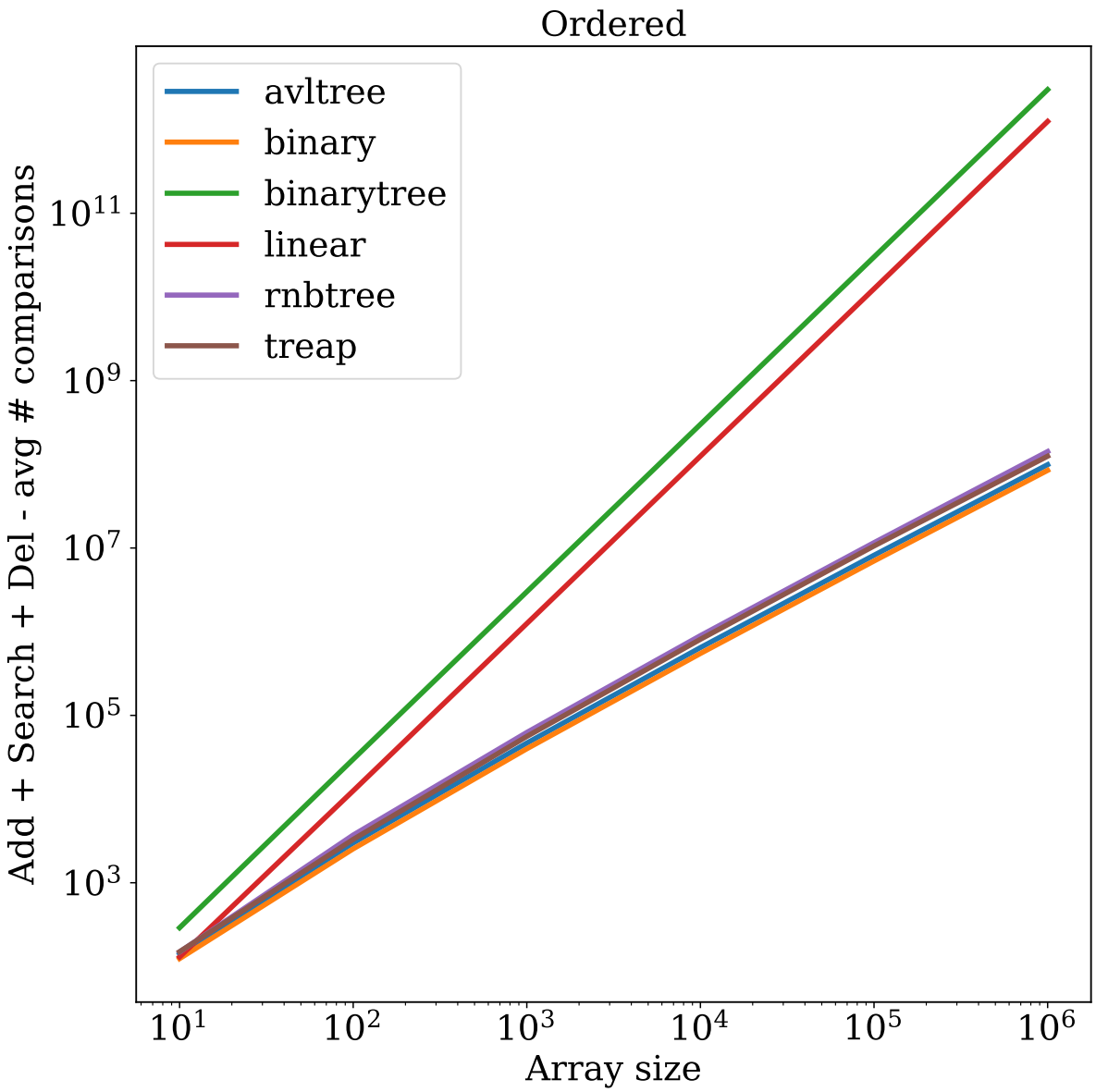

Figure S1: Average number of key comparisons (Ordered arrays)

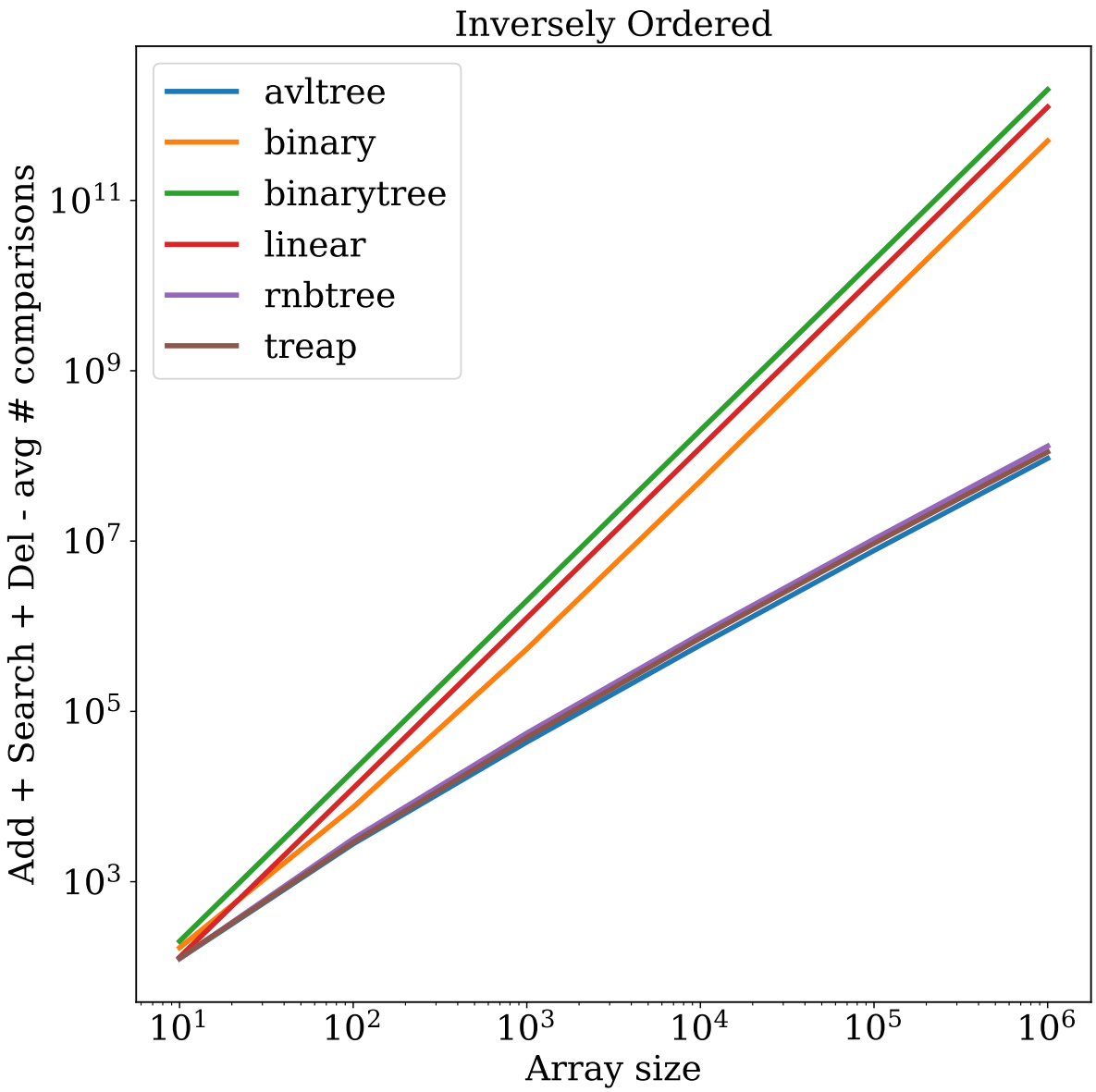

Figure S2: Average number of key comparisons (Inversely ordered arrays)

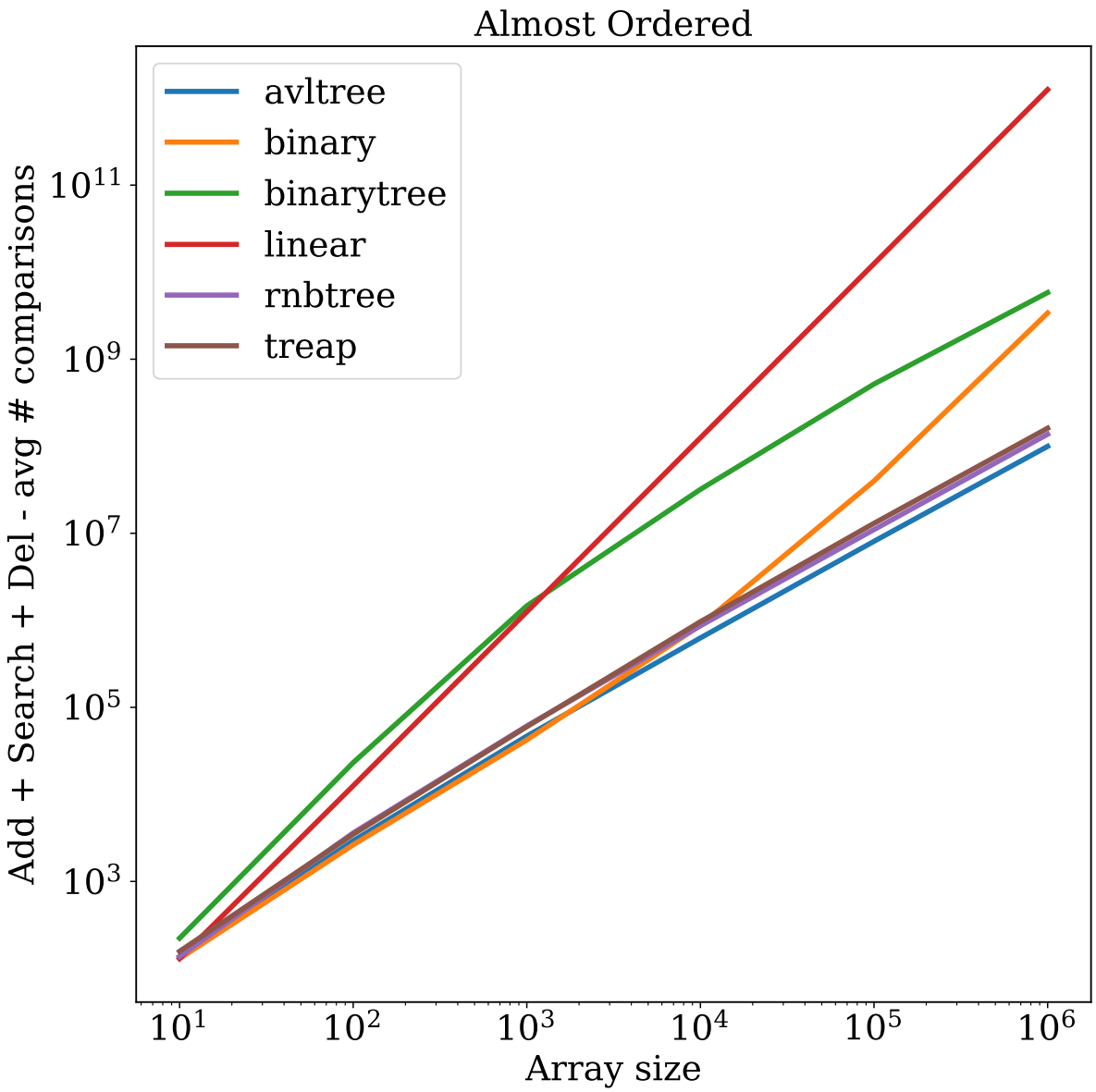

Figure S3: Average number of key comparisons (Almost ordered arrays)

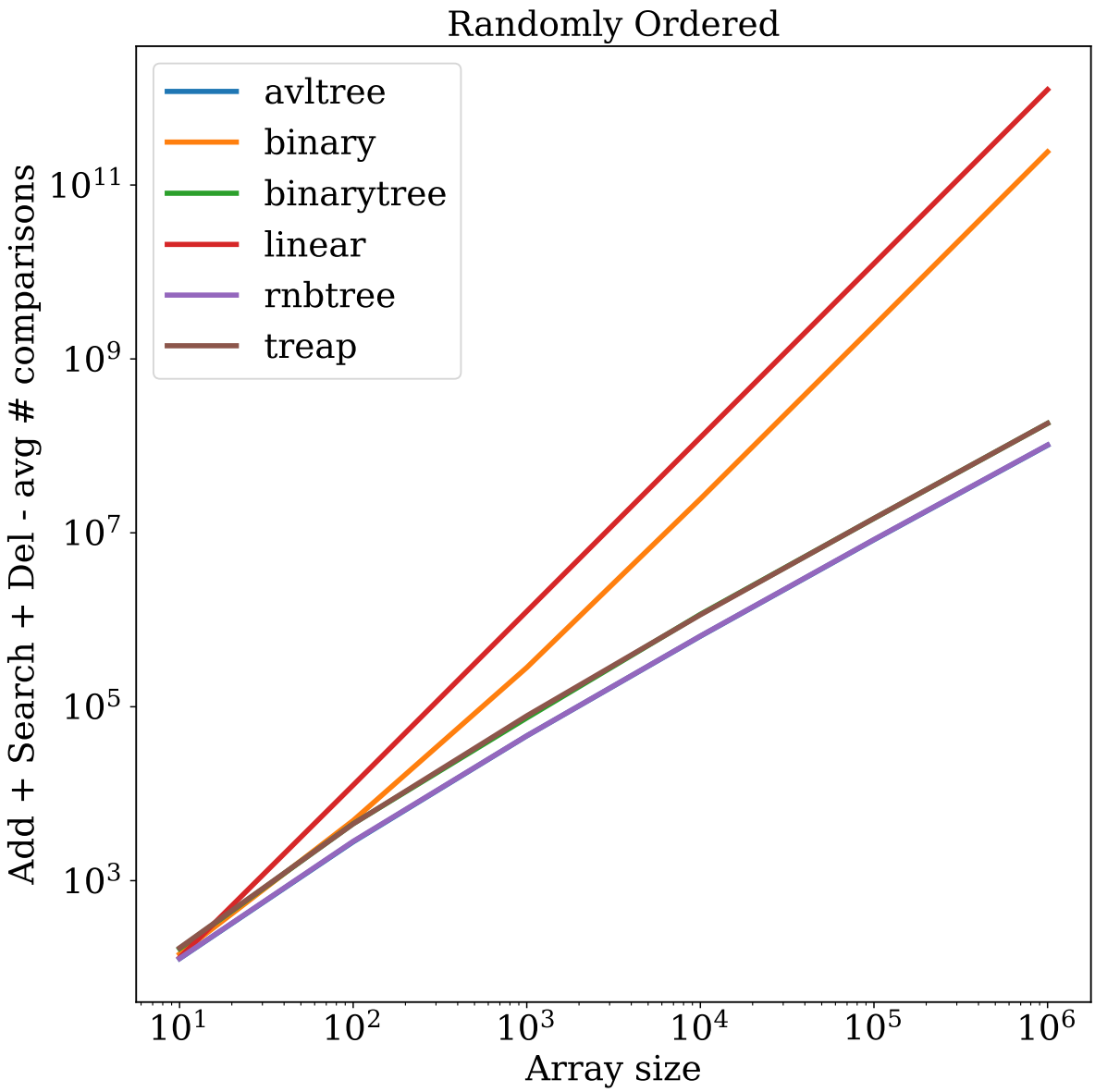

Figure S4: Average number of key comparisons (Randomly ordered arrays)

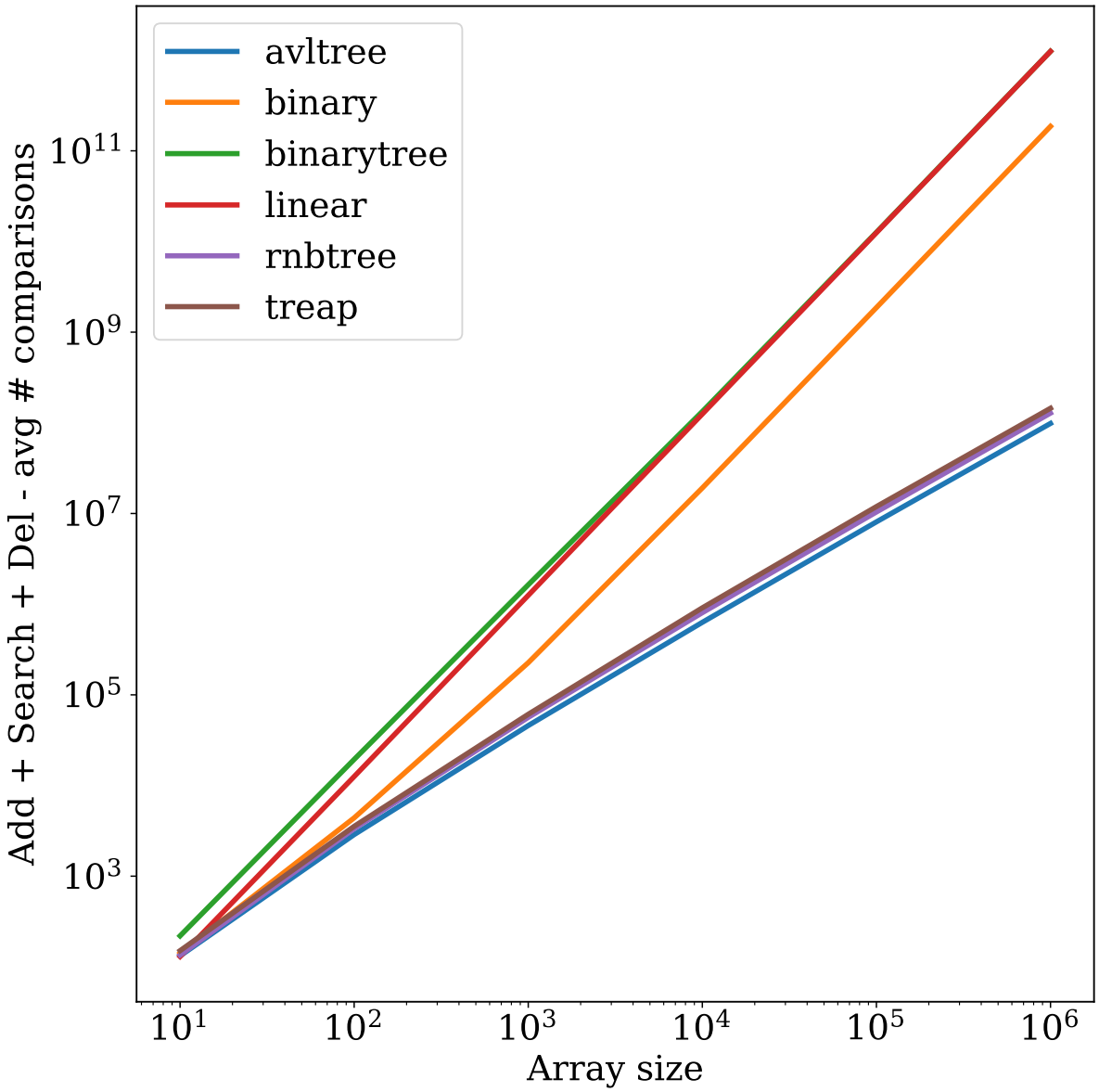

Figure S5: Average number of key comparisons (All arrays)

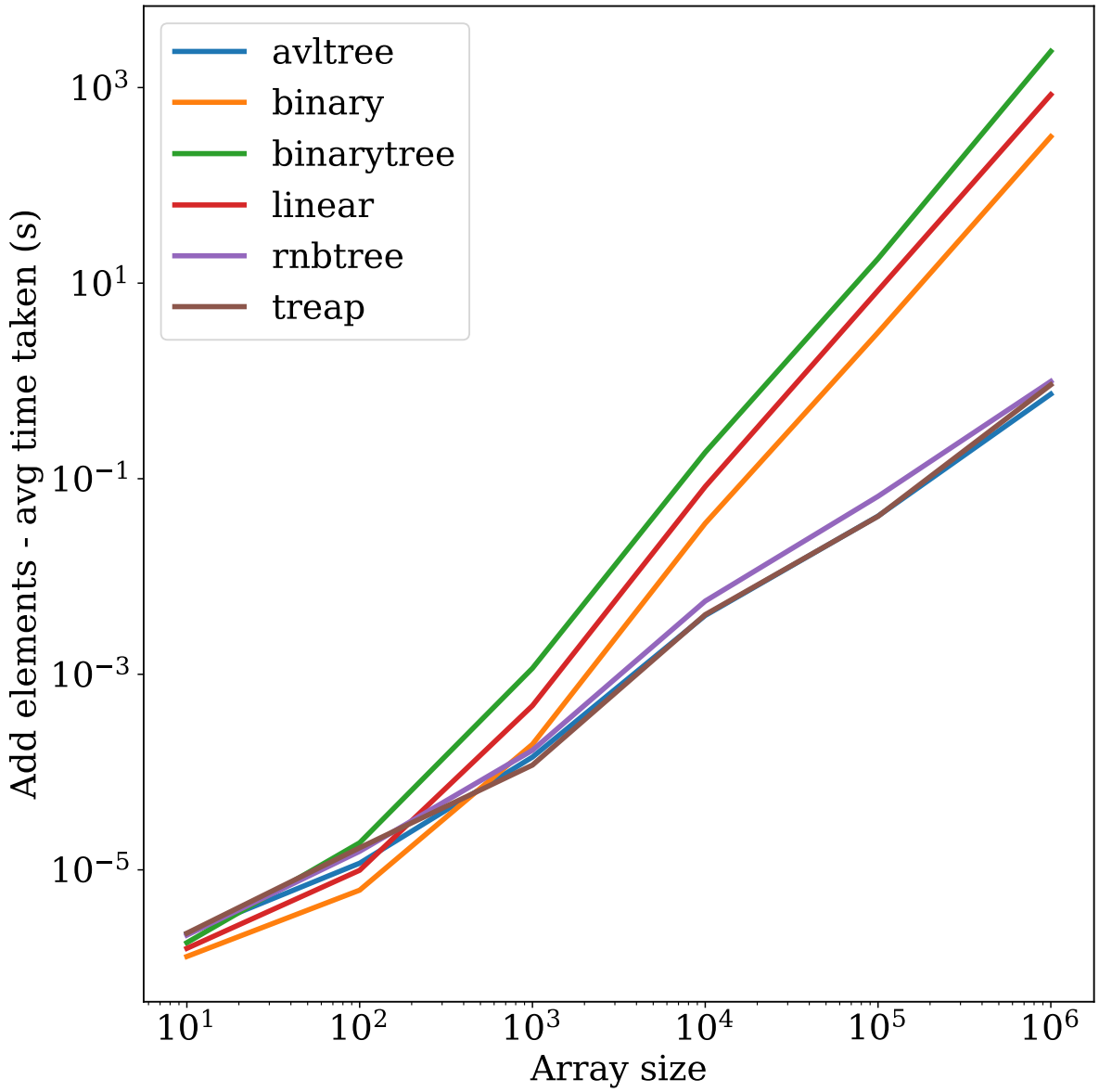

Figure S6: Average of CPU time taken on insertion operations (All arrays)

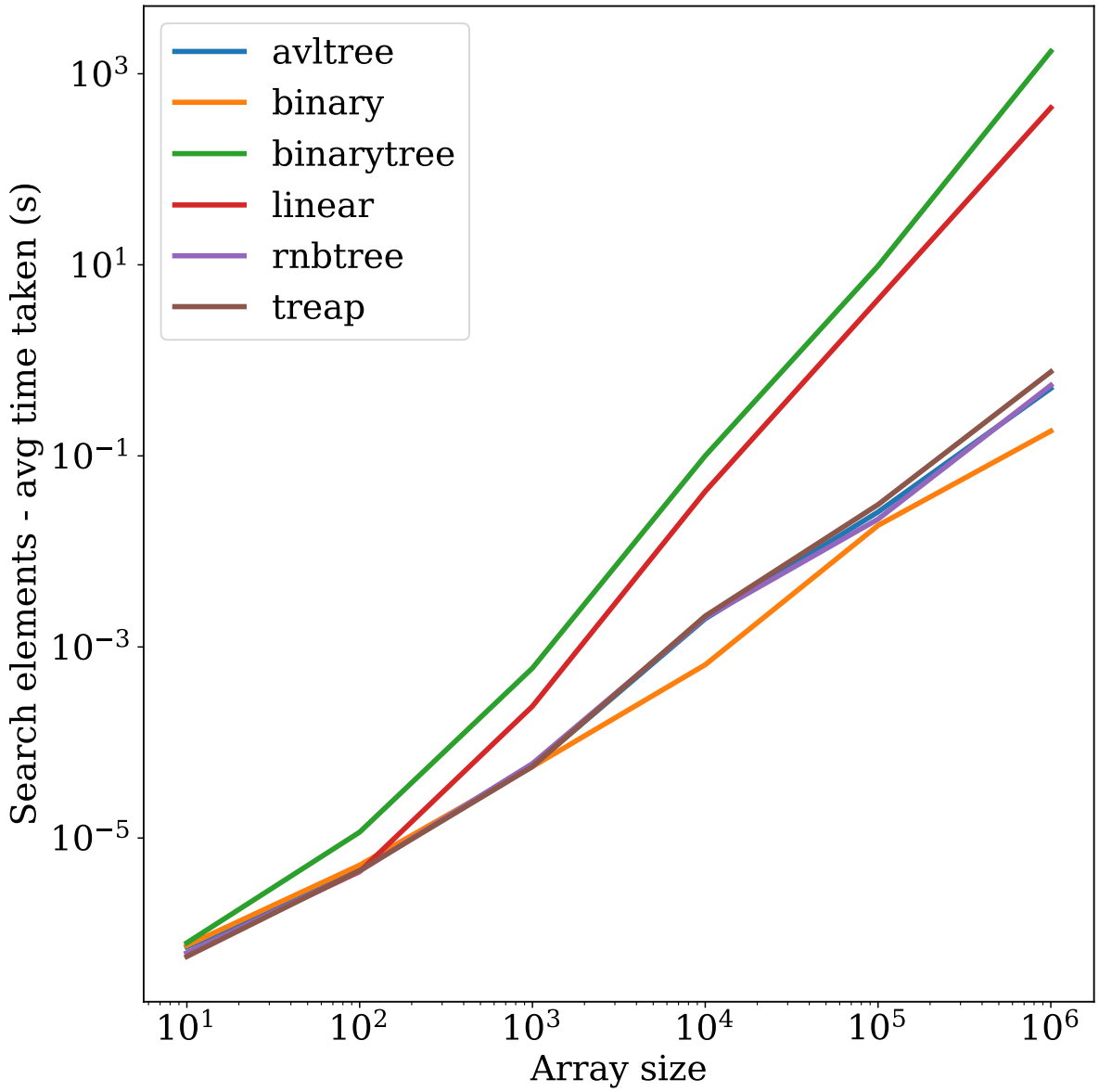

Figure S7: Average of CPU time taken on search operations (All arrays)

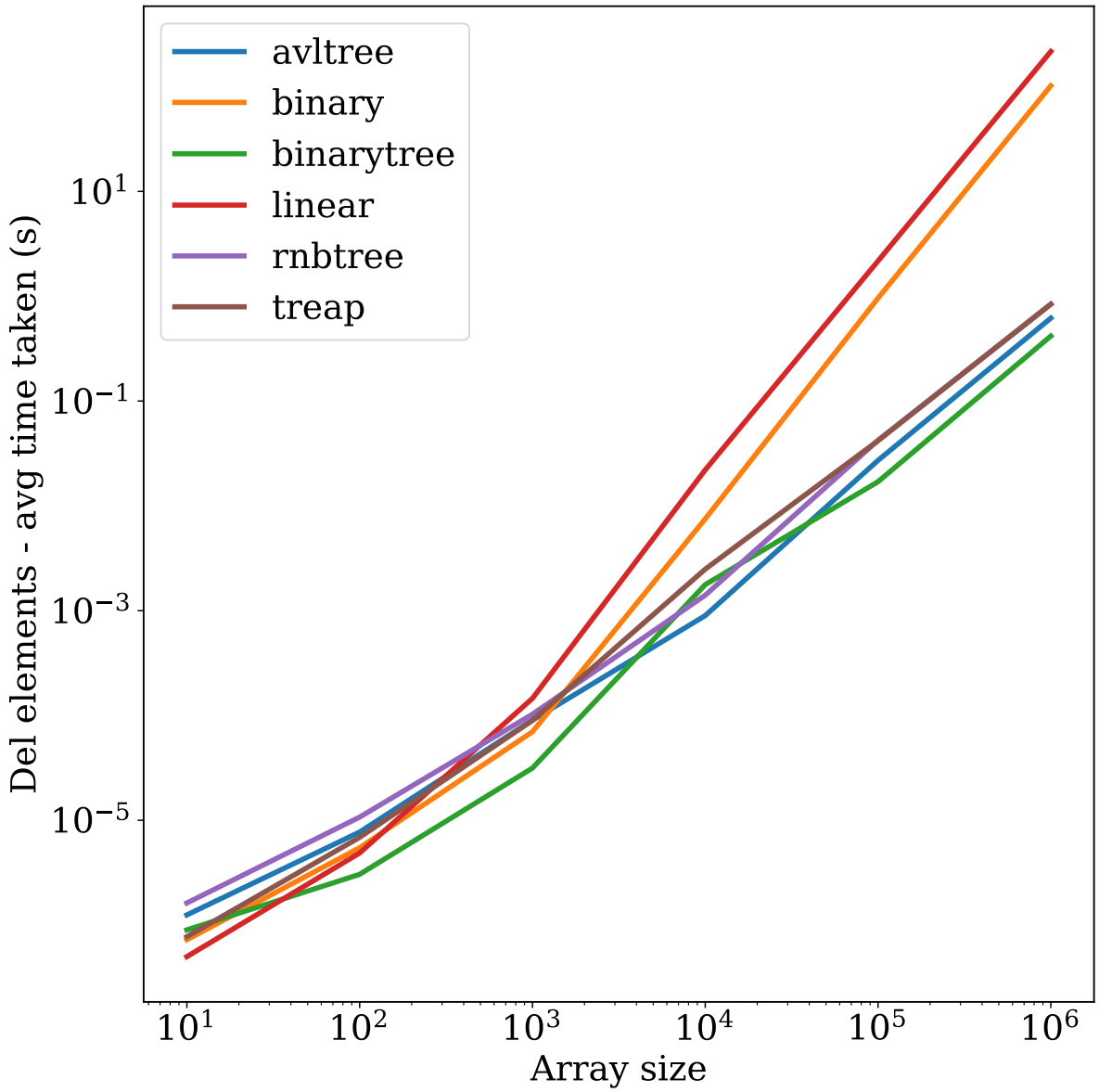

Figure S8: Average of CPU time taken on deletion operations (All arrays)

The complete source code for the project can be found here:
https://github.com/fvcalderan/TreapSearch_analysis.