Pragmatic Usage of Trees

Marcel Jerzyk

A report in the Field of Software Engineering

The 4th Task List of the subject of Algorithms and Data Structures

Politechnika Wrocławska

19 May 2019

Abstract

Many wise people invented and established many algorithms and data structures. These are already well written and optimize and ready to use by the “common folk”. The problem lies in the understanding of weaknesses and preponderances that they provide. The goal of this paper is to conclude pragmatic application of Binary Search Tree and its other, modified versions: Red-Black Tree & Splay Tree. When implementing a tree we have to ask: Is this going to be the best performing tree in this case of use and with this data set? In this context, best performing is defined by the fastest performance in the most common usage of the tree.

Based on own implementation of these listed above with different kinds of data sets (variations of unique sorted sequences, two randomly generated string lists and two common language texts – “Lord of The Rings” book by J.R.R. Tolkien and a script of “The King James Bible” from Bill McGinnis Ministries - "Feeding His Sheep") the knowledge and expectations were subjected to real trial. Analysis of the trees shows that RBT is the best and most versatile tree type. The results indicate that there is very narrow, input oriented and operation specific room to even consider using other tree type.

Frontispiece

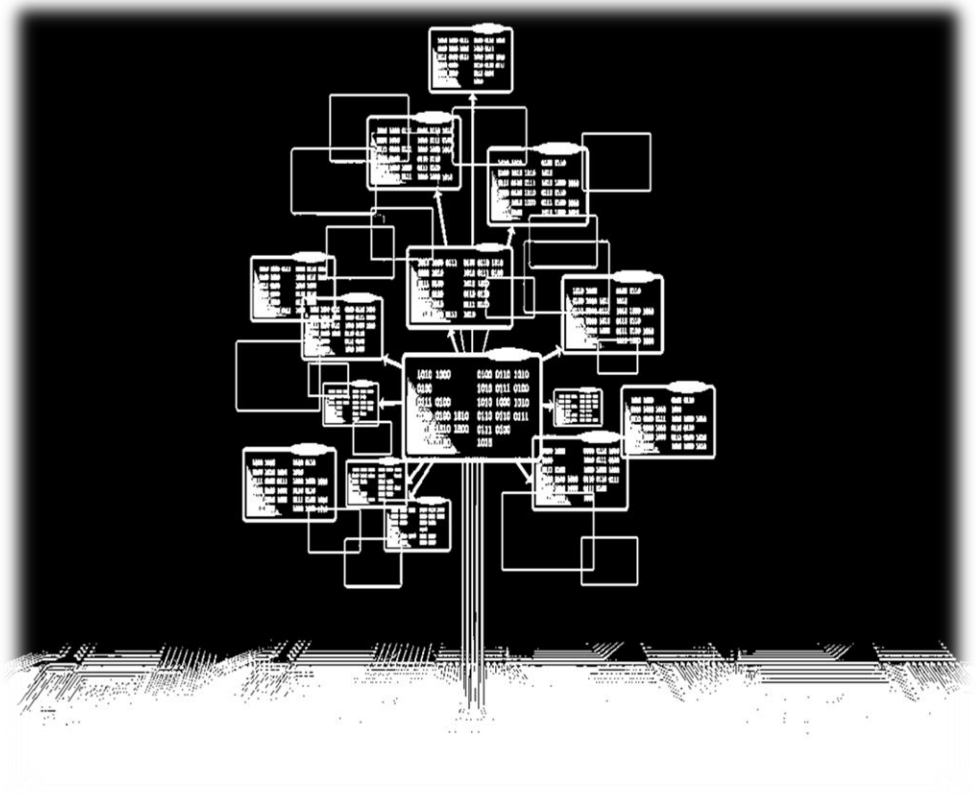


Table of Contents

Frontispiece iv

List of Tables vi

List of Figures vii

Chapter I. Introduction 1

Chapter II. Sample Data 2

Chapter III. Trees 4

- Binary Search Tree (BST) 5
- Red-Black Tree (RBT) 9
- Splay Tree 14

Chapter IV. The Final Overview 18

Chapter V. Results 22

Chapter V. References 24

List of Tables

Table 1. Binary Search Tree Time Averages. 6

Table 2. Binary Search Tree Comparisons 7

Table 3. Binary Search Tree Modifications 8

Table 4. Red Black Tree Time Averages 11

Table 5. Red Black Tree Comparisons 12

Table 6. Red Black Tree Modifications 13

Table 7. Splay Tree Time Averages 15

Table 8. Splay Tree Comparisons 16

Table 9. Splay Tree Modifications 17

Table 10. aspell & aspellReversed time performance (sorted lists) 18

Table 11. aspellRandom & randomStrings time performance (random input).................19

Table 12. kjb & rndStrings1500k time performance (big files).......................................20

Table 13. lotr time performance (English language)..21

List of Figures & Info’s

Figure 1. Simple Unordered Tree. 4

Figure 2. Binary Search Tree 5

Info 1. BST...6

Info 2. RBT 9

Figure 3. Red-Black Tree...10

Info 3. SPlay..14

Figure 4. Sorted Average Results.. 22s

Chapter I.

Introduction

Given the task to construct and code: Binary Search Tree (for short: BST), Red-Black Tree (RBT) and Splay tree, only these three will be considered during the analysis. The language for that wasn't chosen arbitrarily so the choice fell on Java as it's common in the programming world and it's a strongly typed language. In the trees there are implemented measurements for the amount of comparisons, modifications and time which takes to insert, search and delete a key/node from the tree. The most important factor on which this paper emphasis is the time which can be as well understood as the performance. Of course the lower time that takes to do an action the better.

Every algorithm is easily accessible by simply calling the right method, importing a package or just copying it from the internet. There's no need for an average coder to grasp the full knowledge of how a type of a tree works and all the mechanisms that stands behind it (though certainly useful). Desired conclusions are to be as short and as easy to comprehend by anyone who looks at them to easily decide which tree he should implement in his situation.

Chapter II.

Sample Data

To get relatively good results of the test trees had to handle five different sample text files and each of them had different properties. That fills up the possibilities of potential situations that can be faced and gives good all-around understanding of the tree.

First sample is “aspell.txt” which consist of 126 000 unique sorted sequences in alphabetical order without reps. The next two are variations of this file – it’s reverse and randomized one.

In real life randomized list like this could mimic a user database which must be made up with unique names and that’s sorted by the date of registration and an alphabetically ordered lists could be found in e-commerce shops that store their products by name. These are only two examples but these two types of lists are spread all around so they are definitely a good objects to run the tests on. The reversed version can be a an indicator to check if algorithms are implemented properly as this have the same properties as it’s unreversed original.

“kjb.txt” is fourth sample that features real language. Real language fallows two interesting rules that is interesting to mention – The Pareto Principle and Zipf’s Law. In short: Zipf’s Law says that frequency of used words is logarithmic and so the most popular word can repeat a lot and The Pareto Principle indicates that 20% of all the words appears 80% of time. That’s why we are dealing with something special if the sample text is a real life language instead of completely randomized strings or arbitrary chosen unique words. “lotr.txt” is fifth text file that’s another reference to comparison and the difference from “kjb” doesn’t include the formatting that a script does have.

The last two samples are previously mentioned randomized strings of length 3. First version has 55 000 lines which were completely randomly generated and the second one is the same as first one but repeated 27,25 times to hit 1 500 000 number of lines. They were created to check how the trees will perform on strings that doesn’t have any human-driven properties resulting from all the language features that we have had developed and as well on large amount of repetitions.

Chapter III.

Trees

Tree is a widely used abstract data structure that simulates hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes. A tree data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

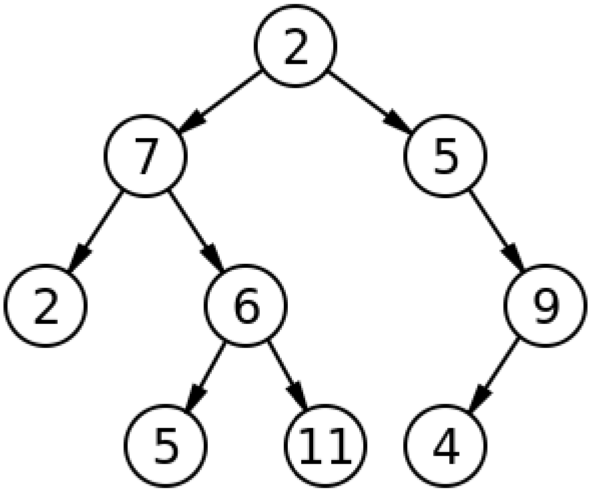


Figure 1: Simple Unordered Tree

Binary Search Tree (BST)

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key and right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

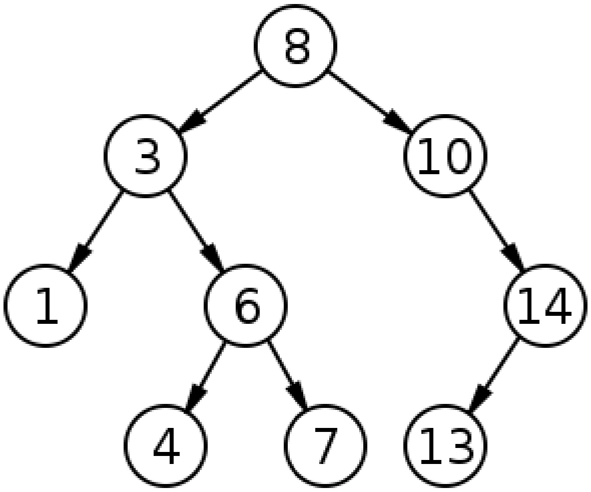
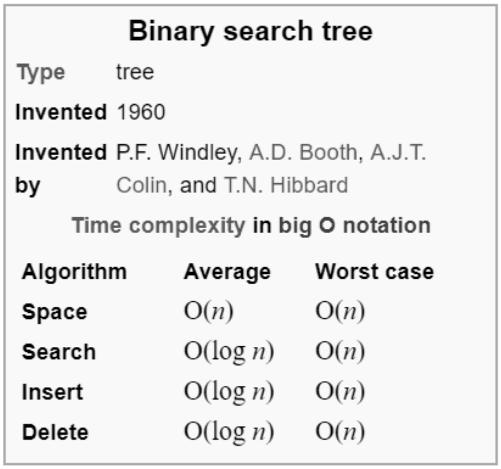


Figure 2: Binary Search Tree

With these two rather simple but yet powerful rules we get a core structure that is foundation for handful of other trees that use BST methods of insertion, search, deletion etc. and just modifying them a bit to suit their own intentions.

BST does have average time of O(logn) on all examined situations as it can be expected from a tree-like structure – logn represents how (more or less) the tree will expand in height. The problem arise when to the structure is kept being added a new maximal or minimal key so it expands in height with every iteration giving O(n) in the worst case. To make this tree effective new elements should preferably be average compared to the rest.



Binary search tree

Type	tree
Invented	1960
Invented by	P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Info 1: BST

Table 1. Binary Search Tree Time Averages

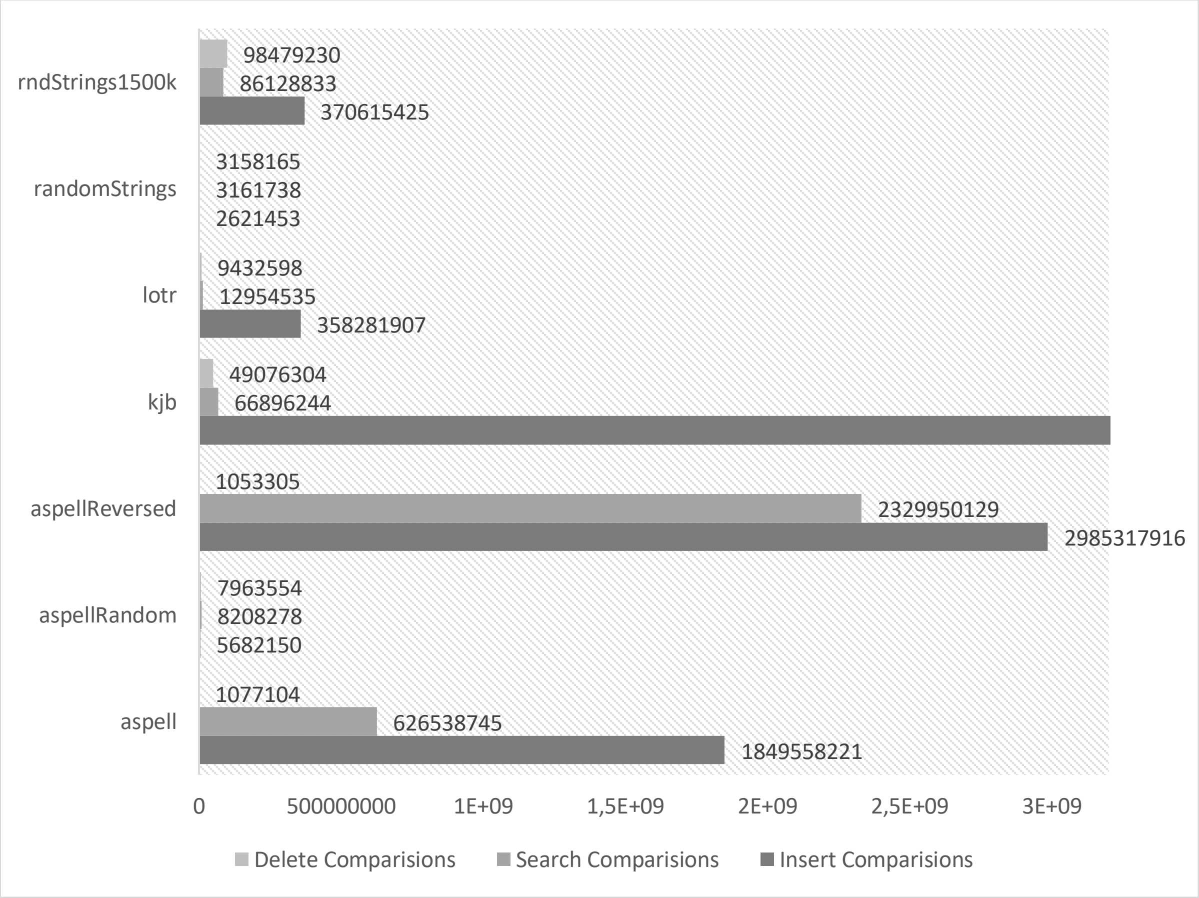
Text File	Insert Time	Search Time	Delete Time
aspell.txt	76,164s	74,003s	0,009s
aspellReversed.txt	85,837s	82,316s	0,009s
aspellRandom.txt	0,099s	0,140s	0,096s
kjb.txt	2102,015s	0,371s	0,314s
lotr.txt	8,566s	0,94s	0,06s
randomStrings.txt	0,028s	0,028s	0,025s
randomStrings1500k.txt	7,769s	0,715s	0,999s

Averages taken from 10 tests sample size.

At this moment without comparison how other trees can handle the task it’s hard to judge this tree. Extremely large insertion time on kjb.txt is due to the file being huge (~4MB) with a lot of words in it. This can overshadow aspell.txt and aspellReversed.txt large insertion times but it’s worth noting that this is definitely big.

As mentioned in the Data Samples chapter, language consist of certain rules which we can observe in the differences between search times – files with completely unique values keep their Search Time about as good as Insertion Time while the book & script keep their search time low even despite enormous size in the case of kjb.txt.

Table 2. Binary Search Tree Comparisons

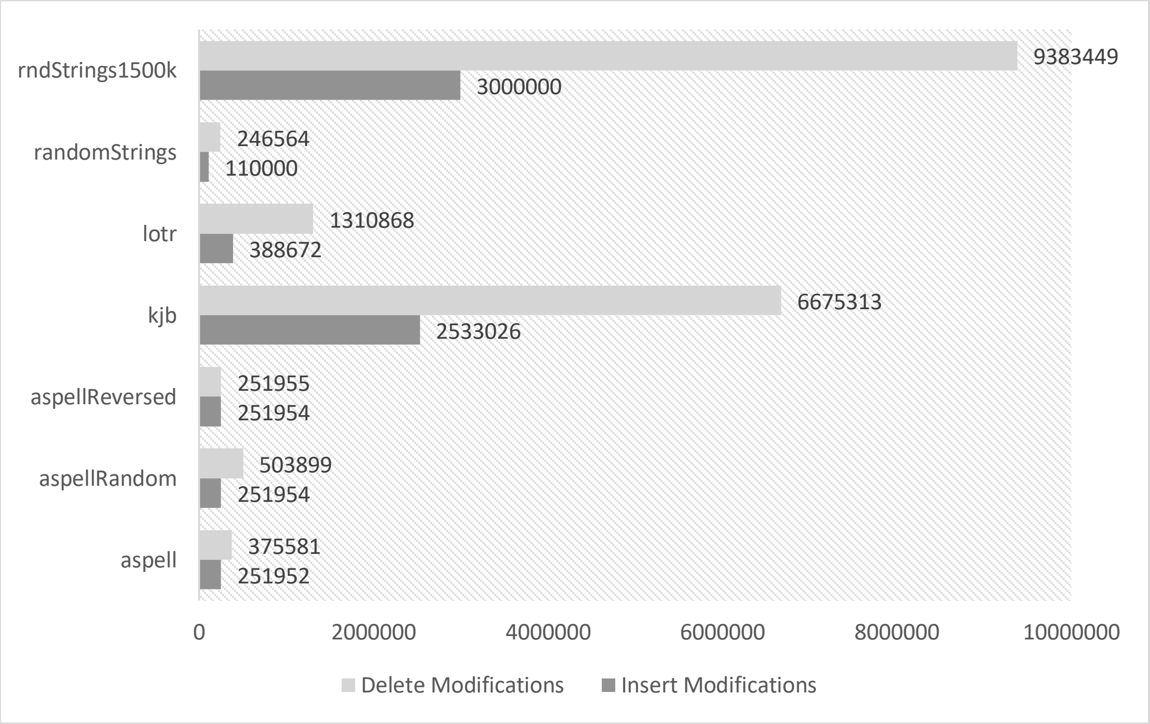


Averages taken from 10 tests sample size.

Pay attention to the aspell – file performed much better in overall comparisions than its reversed counterpart and gained only slight edge in overall time performance (~8,5s). The kjb had of course gigantic amount of insertions comparisions putting the amount literally "off the charts" (3 977 851 022).

So far deducting from the Tables it is certain that insertion is the biggest performance offender as it takes a lot of insertion comparisons to add another node to the tree and if data wasn't sorted – other operations will have decent effective time. We assured that with rndStrings1500k file which is about 12 times bigger with about 27 repetitions and it's search time is low in contrary to alphabetically sorted files.

Table 3. Binary Search Tree Modifications



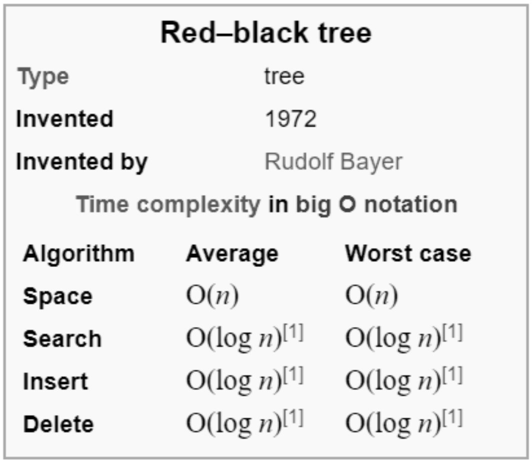
Averages taken from 10 tests sample size.

From gathered information we can conclude that BST works very well with randomized files (aspellRandom.txt & randomStrings.txt) and struggles a lot with inserting continuously max/min keys to the tree. There is slight performance decrease if in a very large sample with reps delete operation is made due to the amount of modifications needed.

At this point the interest lies in improving the insertion. It is indeed possible because BST doesn't have any balancing properties between throwing keys to the left or right with its simple rule and depending on sample data the tree can even end up as a linked list in worst case scenario. An example of self-balancing tree to keep its height always to logn is the Red-Black Tree.

Red-Black Tree (RBT)

Red Black Tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra information about it's color (red or black). These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.



Red–black tree

Type	tree
Invented	1972
Invented by	Rudolf Bayer

Time complexity in big O notation

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(\log n)$[1]	$O(\log n)$[1]
Delete	$O(\log n)$[1]	$O(\log n)$[1]

Info 2: RBT

Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged and repainted to restore the coloring properties and it can perform that efficiently.

The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to a previously mention binary search tree. To store it's color each node has "color bit" which can be 0/1 depending on a color.

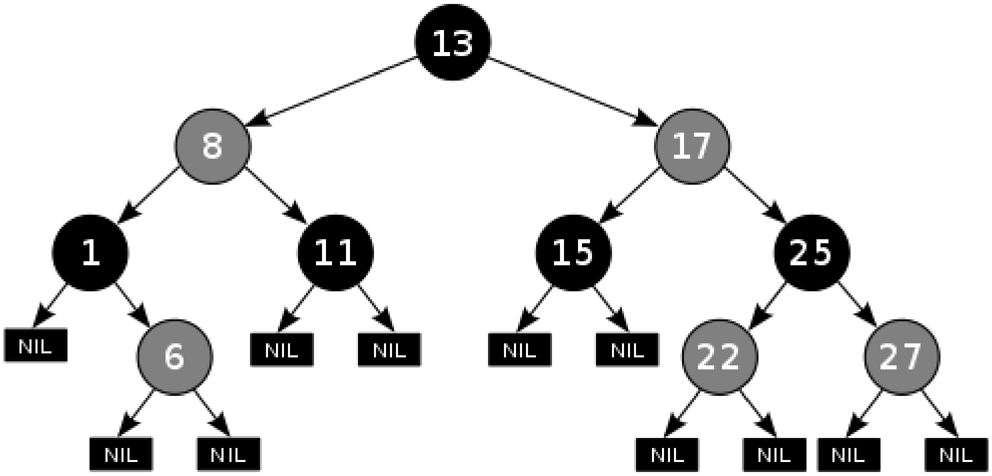


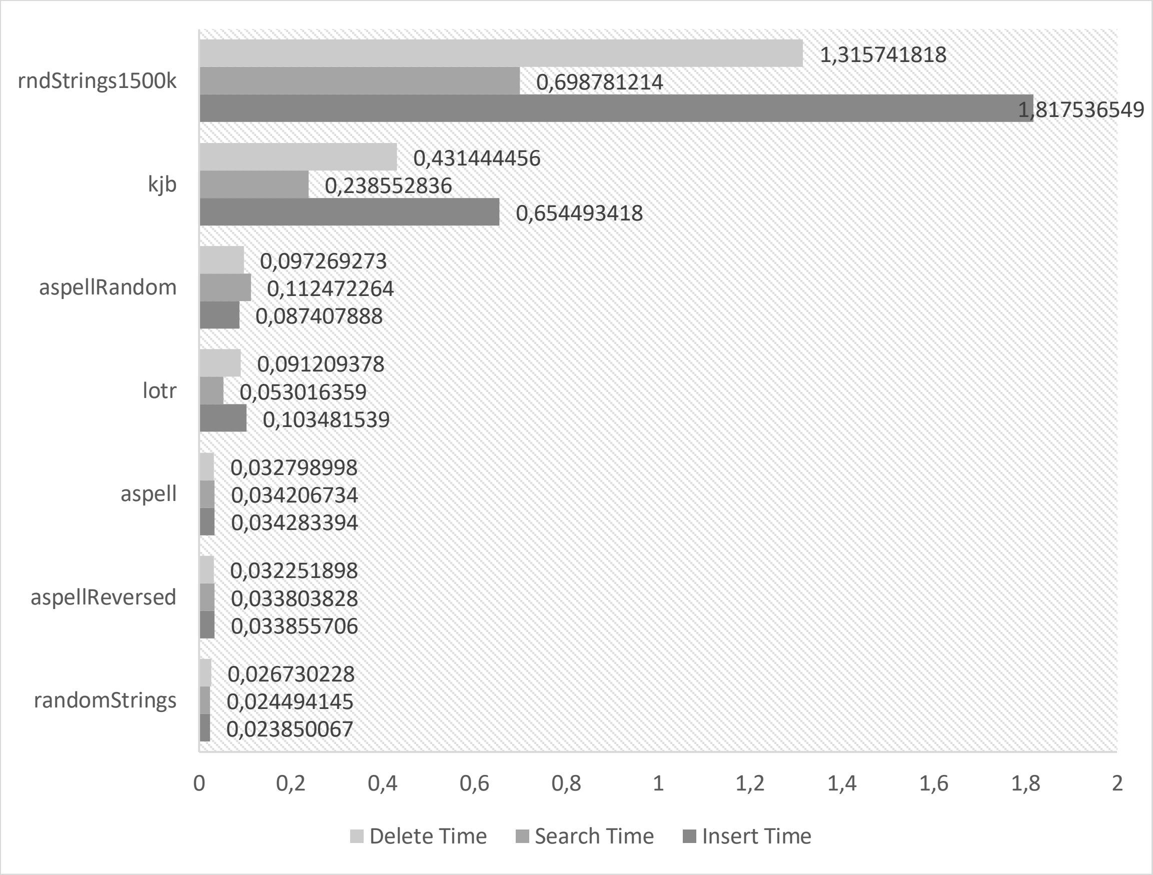
Figure 3: Red-Black Tree

Properties:

1. Each node is either red or black.
2. The root is black. This rule is sometimes omitted. Since the root can always be changed from red to black, but not necessarily vice versa, this rule has little effect on analysis.
3. All leaves (NIL) are black.
4. If a node is red, then both its children are black.
5. Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes.

Given these properties we assure resolving the previously mentioned problem of BST which was the struggle of continues min or maxes added to the structure.

Table 4. Red Black Tree Time Averages

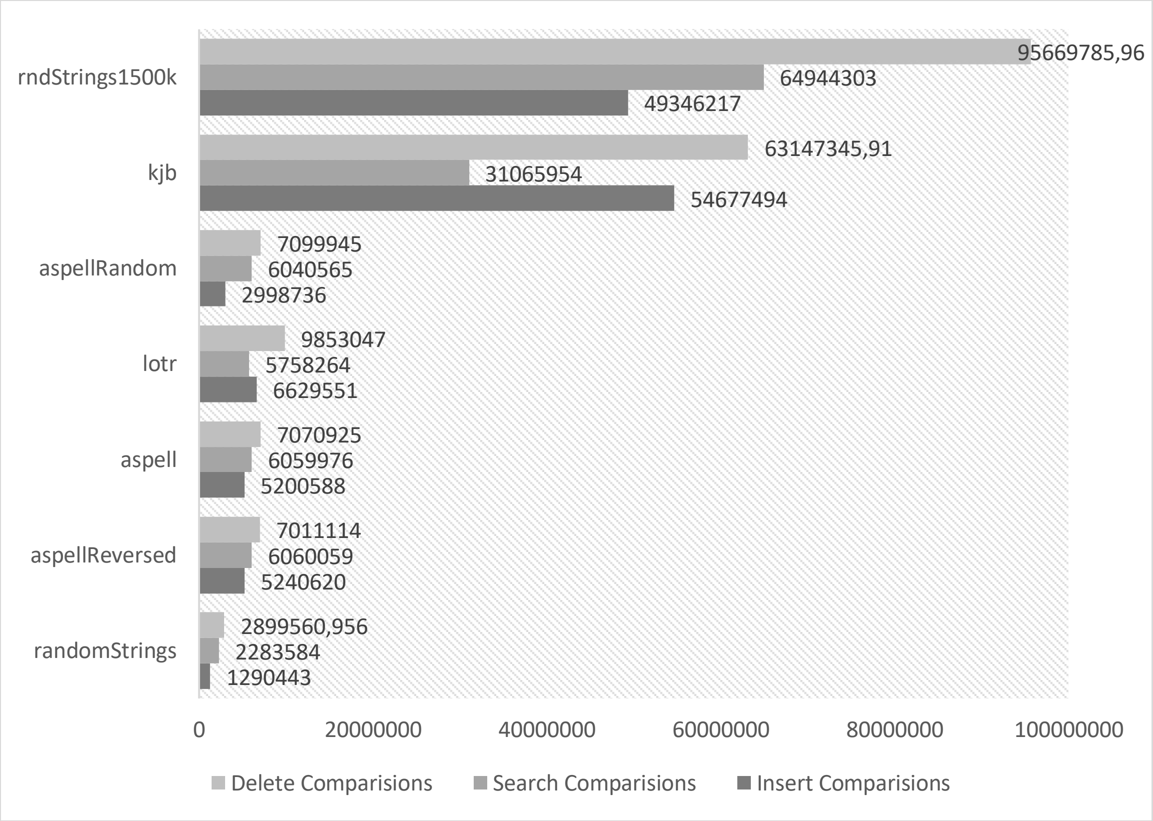


Averages taken from 160 tests sample size.

The results are outstanding in comparison to BST. RBT greatly outperformed the classic tree in all aspects besides Search Time for rndStrings1500.txt (almost twice as long). KJB file insertion time problem is now non-existent and same goes for alphabetically sorted samples. None of tested operations exceeded 2s mark. Contrary to

BST the randomized version of aspell took longer to proceed than presorted ones (which achieved identical time results).

Table 5. Red Black Tree Comparisons

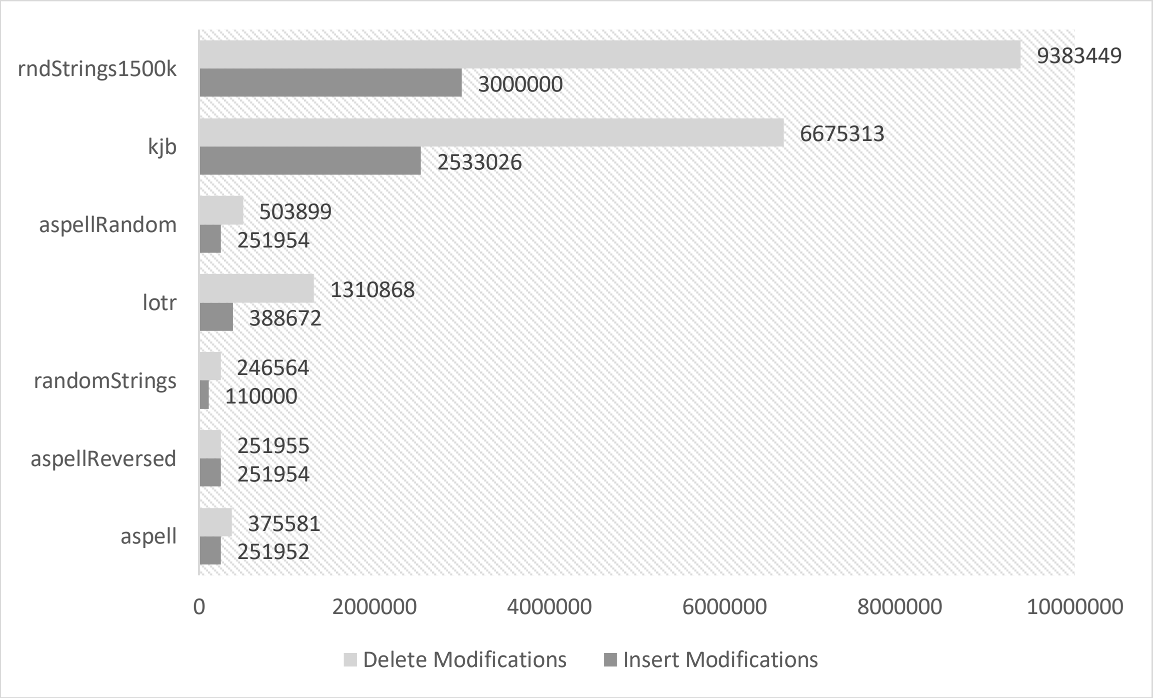


Averages taken from 160 tests sample size.

In this tree amount of comparisons for insertion doesn't overwhelm two other inspected operations. Again we see a limit up to 1,0E8 where BST exceeded far beyond that number. Based on this graph the biggest struggle for RBT are simply the biggest sample files which require a lot of comparisons to manage deletion. This is a very good sign because now if one would like to look for improvements he should rather do that by reducing the sample input rather than using different algorithm as it is overall very effective.

Type of input (if it’s language, sorted or random) doesn’t seem to have meaningful impact at least on this samples. Maybe on much bigger files (let’s say over 100MB) differences could be more prevalent.

Table 6. Red Black Tree Modifications

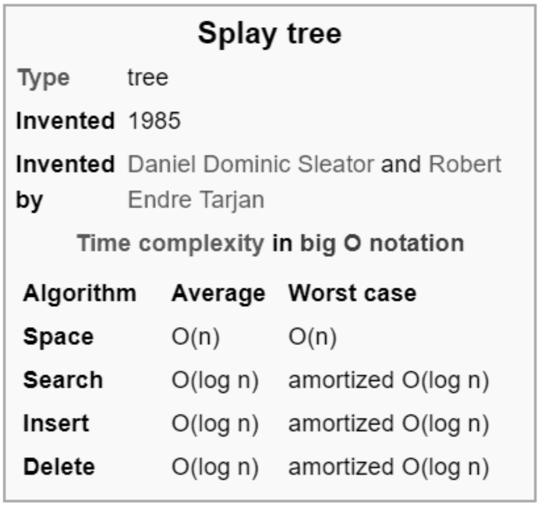


Averages taken from 160 tests sample size.

Nothing new to observe from the amount of modifications. Seems like RBT is a perfect answer if we don’t know for what kind of input it will be used as it will work very well regardless. What if one know what will be the tree used for and would like to have special properties that suits his needs? For example: if one would like to quickly access the last used node? For that reason we got a Splay Tree.

Splay Tree

A Splay Tree is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time.



Splay tree

Type	tree
Invented	1985
Invented by	Daniel Dominic Sleator **and** Robert Endre Tarjan

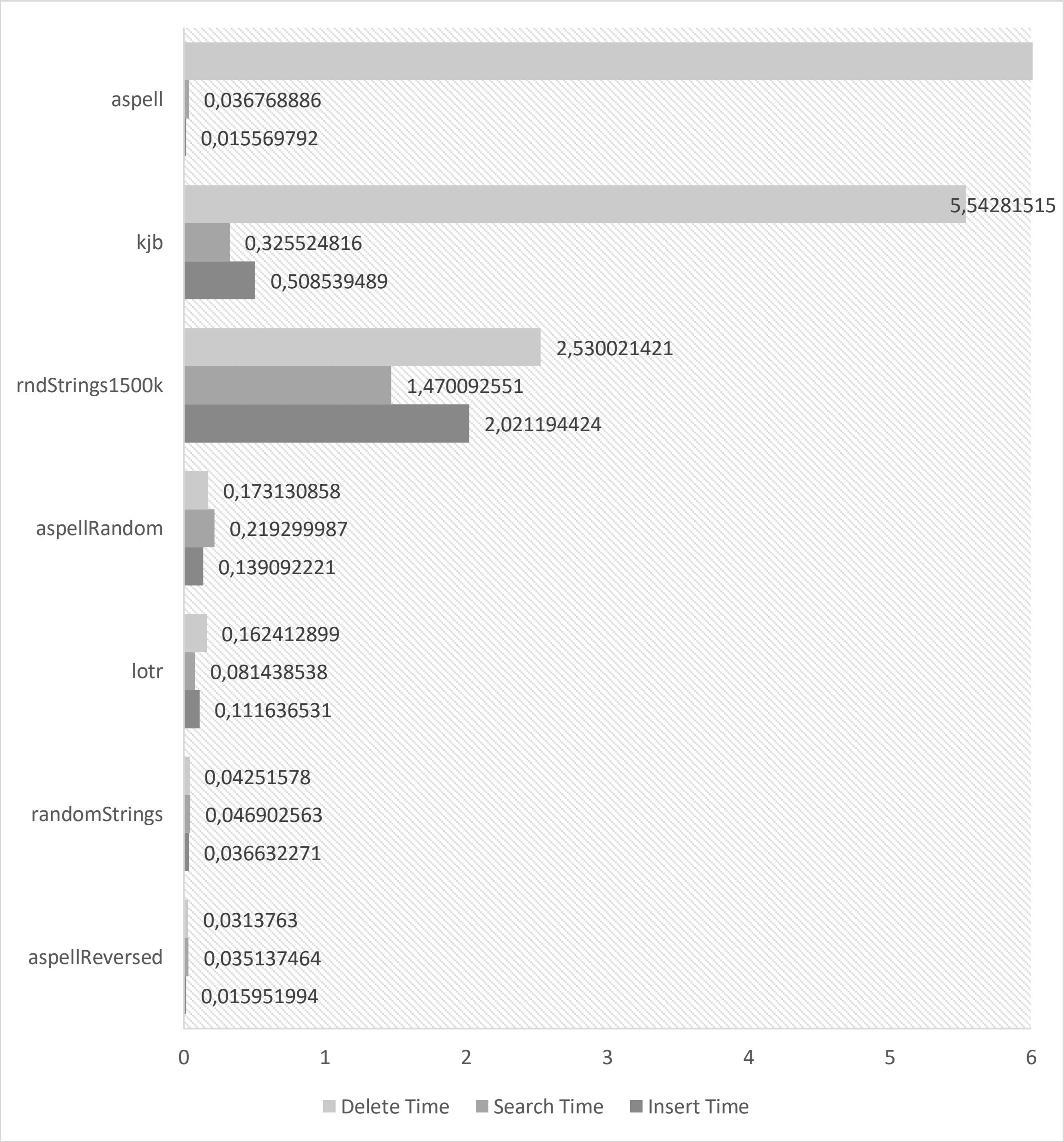
Time complexity **in big O notation**

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	amortized O(log n)
Insert	O(log n)	amortized O(log n)
Delete	O(log n)	amortized O(log n)

Info 3: Splay

All normal operations on a binary search tree are combined with one basic operation, called splaying. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this with the basic search operation is to first perform a standard binary tree search for the element in question, and then use tree rotations similar to RBT (which are called Zig for right rotation and Zag for left rotation) to bring the element to the top.

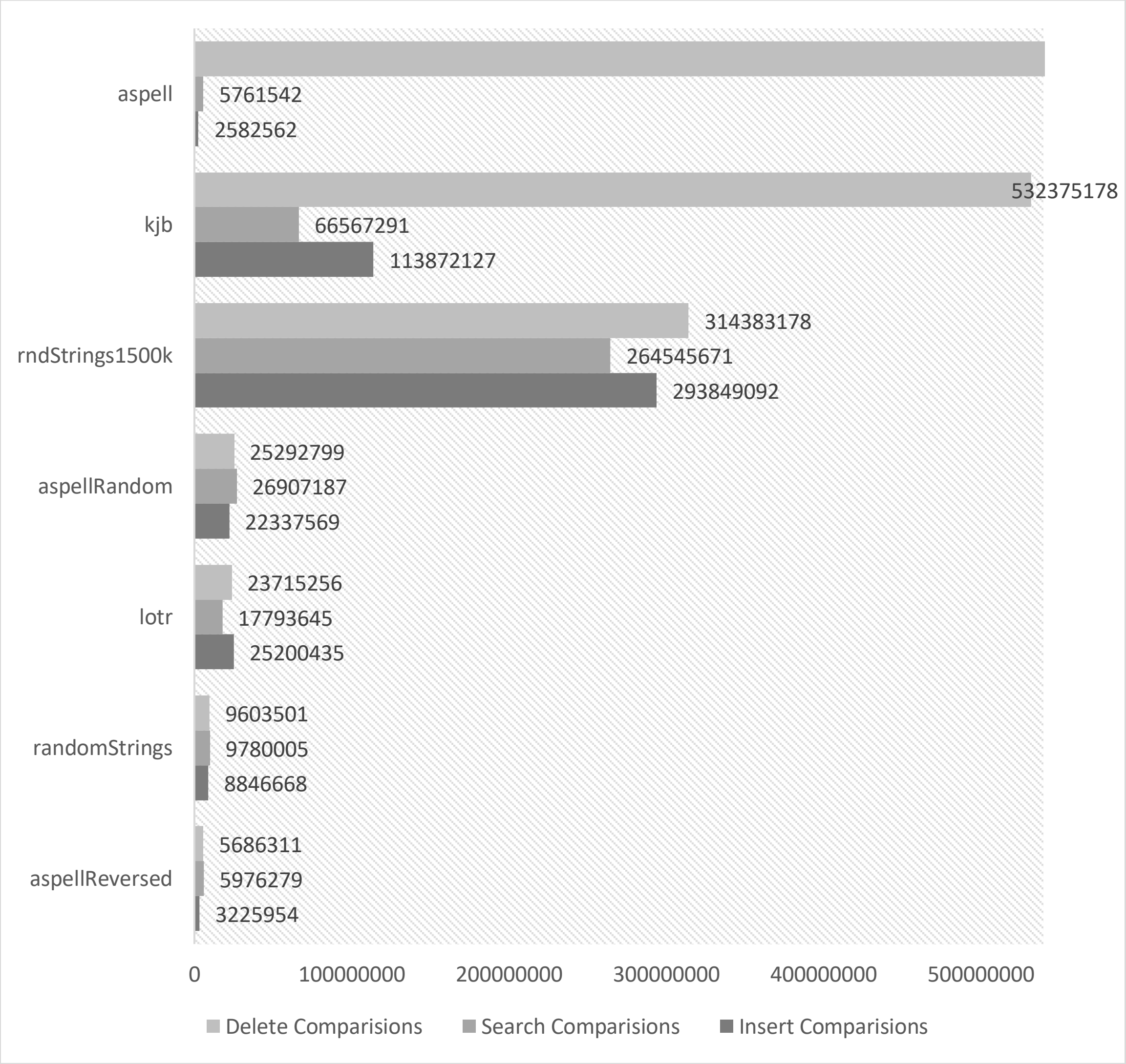
Table 7. Splay Tree Time Averages



Averages taken from 160 tests sample size.

Value for deletion in aspell is off the charts (~69s). It struggles from the same problem as BST.

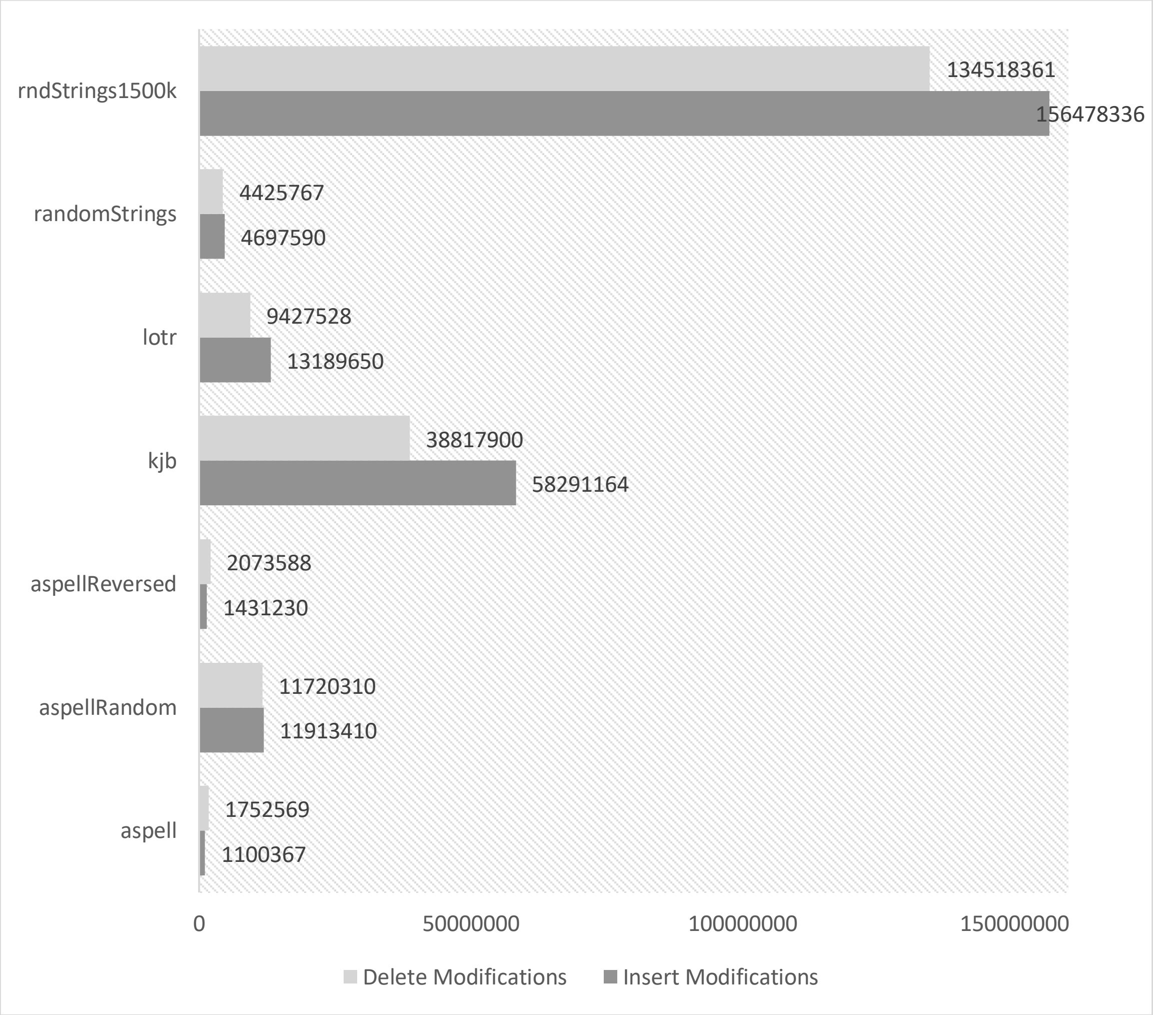
Table 8. Splay Tree Comparisons



Averages taken from 160 tests sample size.

This tree needs a lot of comparisons for kjb and aspell which was prevalent in the Time Average Table (7.). In aspell (in the opposite to it's reversed counterpart) delete need to traverse through the whole tree to proceed (after performing search from [1..n], 1 ends up at the very end).

Table 9. Splay Tree Modifications



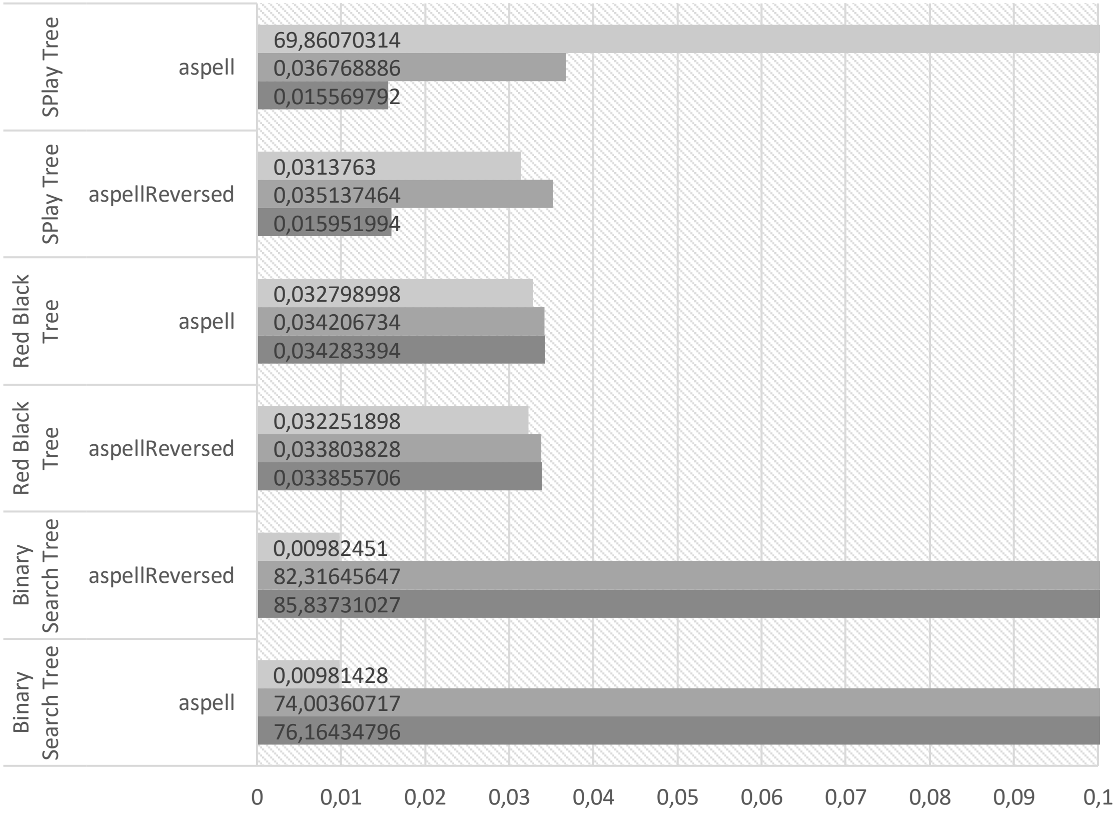
Averages taken from 160 tests sample size.

Chapter IV.

The Final Overview

In Chapter III we took a closer look how each tree manages each of the sample data. Now let’s compare them with each other to get final observations before giving the verdict on use situations for them.

Table 10. aspell & aspellReversed time performance (sorted lists)

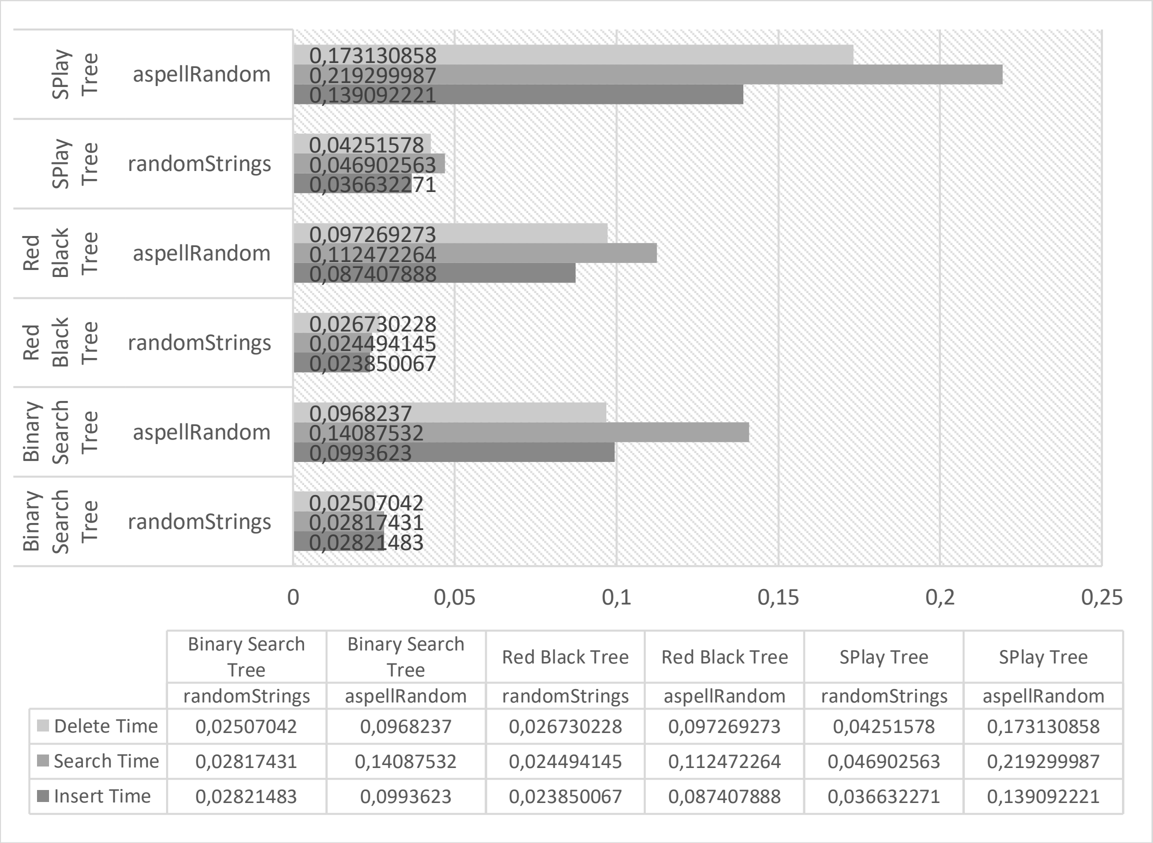


	Binary Search Tree	Binary Search Tree	Red Black Tree	Red Black Tree	SPlay Tree	SPlay Tree
	aspell	aspellReversed	aspellReversed	aspell	aspellReversed	aspell
▨ Delete Time	0,00981428	0,00982451	0,032251898	0,032798998	0,0313763	69,86070314
▨ Search Time	74,00360717	82,31645647	0,033803828	0,034206734	0,035137464	0,036768886
▨ Insert Time	76,16434796	85,83731027	0,033855706	0,034283394	0,015951994	0,015569792

Note: Time scale is set max to 0,1s

In terms of alphabetical insertions BST is a big no-no regardless the quickest deletion times. Because splay fails in terms of time performance at deletion for one version of the alphabetical input, RBT scores a point here. Worth to mention before proceeding: insertion is fastest in Splay.

Table 11. aspellRandom & randomStrings time performance (random input)

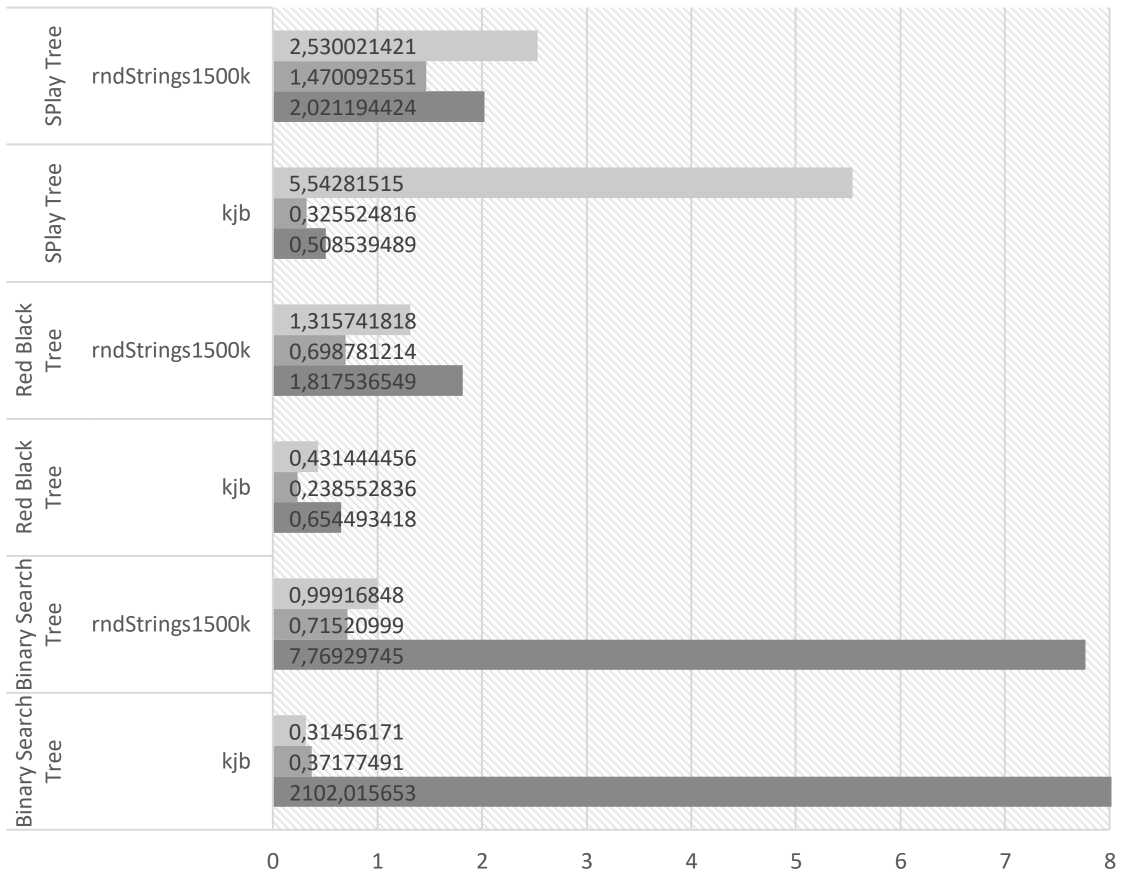


	Binary Search Tree	Binary Search Tree	Red Black Tree	Red Black Tree	SPlay Tree	SPlay Tree
	randomStrings	aspellRandom	randomStrings	aspellRandom	randomStrings	aspellRandom
■ Delete Time	0,02507042	0,0968237	0,026730228	0,097269273	0,04251578	0,173130858
■ Search Time	0,02817431	0,14087532	0,024494145	0,112472264	0,046902563	0,219299987
■ Insert Time	0,02821483	0,0993623	0,023850067	0,087407888	0,036632271	0,139092221

Note: Time scale is set max to 0,25s

Random inputs are actually managed quite well in all of trees. Splay had the worst performance out of the three for both samples. RBT beats BST by a tiny amount (~0,001s - 0,03s) in every aspect besides deletion of both randomStrings and aspellRandom where it’s the other way round.

Table 12. kjb & rndStrings1500k time performance (big files)

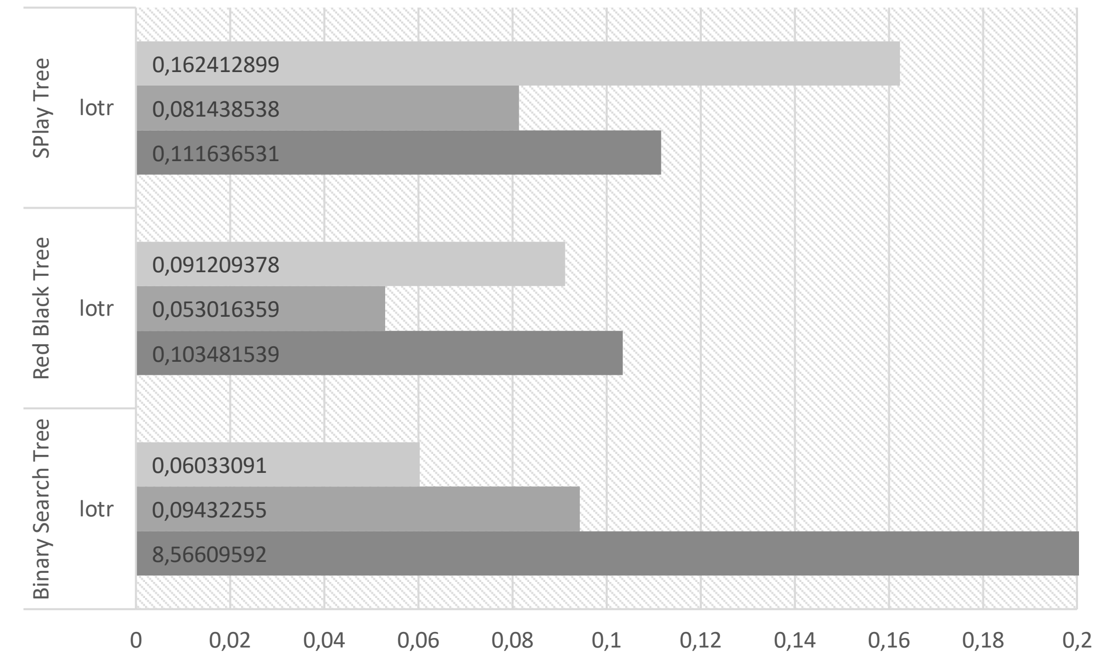


	Binary Search Tree	Binary Search Tree	Red Black Tree	Red Black Tree	SPlay Tree	SPlay Tree
	kjb	rndStrings1500k	kjb	rndStrings1500k	kjb	rndStrings1500k
■ Delete Time	0,31456171	0,99916848	0,431444456	1,315741818	5,54281515	2,530021421
■ Search Time	0,37177491	0,71520999	0,238552836	0,698781214	0,325524816	1,470092551
■ Insert Time	2102,015653	7,76929745	0,654493418	1,817536549	0,508539489	2,021194424

Note: Time scale is set max to 8s

BST have unacceptable time performance for insertions in kjb (~2102s), however with that sample file splay found a very, very narrow niche for itself. If the interest lies in insertion time then the splay is the tree to go. Not using RBT for this case sacrifices much better delete time (~3,7s) and search (~0,15s) for a bit faster insertion (~0,1s). For rndString1500k the fastest is RBT.

Table 13. lotr time performance (English language)



	Binary Search Tree	Red Black Tree	SPlay Tree
	lotr	lotr	lotr
▨ Delete Time	0,06033091	0,091209378	0,162412899
▨ Search Time	0,09432255	0,053016359	0,081438538
▨ Insert Time	8,56609592	0,103481539	0,111636531

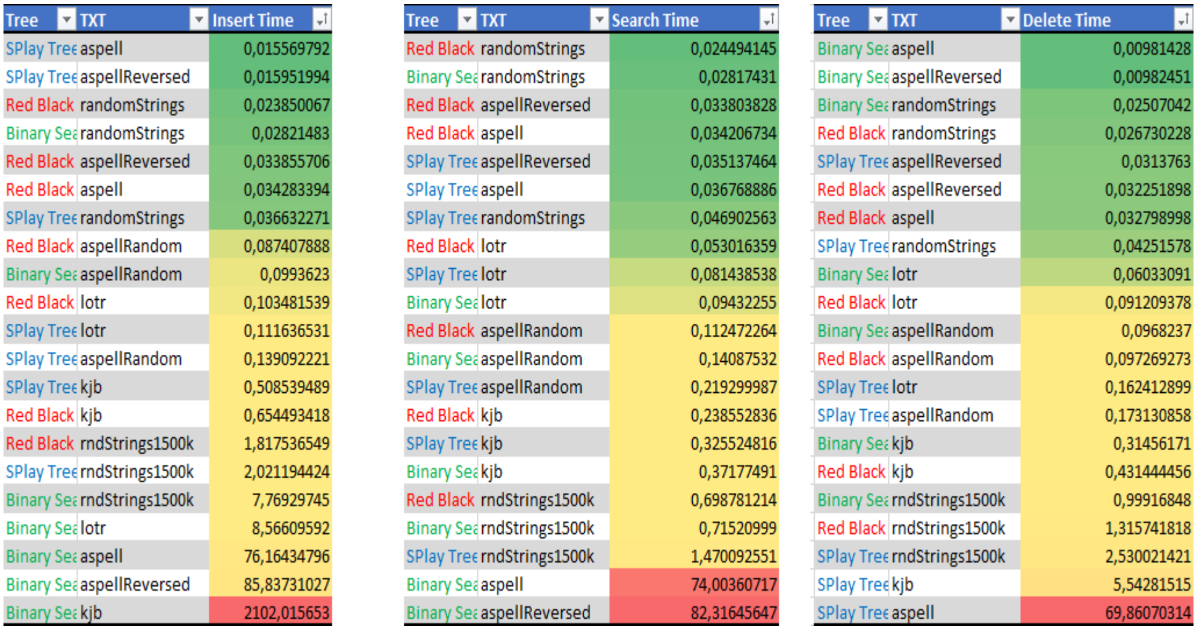
Note: Time scale is set max to 0,2s

Again Splay doesn’t have anything to be for which he could be praised as better time performance for operations can be found in BST and RBT. As well once again better deletion time is achieved by BST and insertion with search is quicker in RBT.

Chapter V.

Results

Summarizing gathered data it turns out that Red Black Tree is the overall winner in terms of global performance. Not only it will perform faster than two other candidates but it will also easily work any type of inserted input (so without a certain niche or without having a clue what type of data will be inserted into the structure – it's the best versatile choice).



Tree	TXT	Insert Time
SPlay Tree	aspell	0,015569792
SPlay Tree	aspellReversed	0,015951994
Red Black	randomStrings	0,023850067
Binary Sea	randomStrings	0,02821483
Red Black	aspellReversed	0,033855706
Red Black	aspell	0,034283394
SPlay Tree	randomStrings	0,036632271
Red Black	aspellRandom	0,087407888
Binary Sea	aspellRandom	0,0993623
Red Black	lotr	0,103481539
SPlay Tree	lotr	0,111636531
SPlay Tree	aspellRandom	0,139092221
SPlay Tree	kjb	0,508539489
Red Black	kjb	0,654493418
Red Black	rndStrings1500k	1,817536549
SPlay Tree	rndStrings1500k	2,021194424
Binary Sea	rndStrings1500k	7,76929745
Binary Sea	lotr	8,56609592
Binary Sea	aspell	76,16434796
Binary Sea	aspellReversed	85,83731027
Binary Sea	kjb	2102,015653

Tree	TXT	Search Time
Red Black	randomStrings	0,024494145
Binary Sea	randomStrings	0,02817431
Red Black	aspellReversed	0,033803828
Red Black	aspell	0,034206734
SPlay Tree	aspellReversed	0,035137464
SPlay Tree	aspell	0,036768886
SPlay Tree	randomStrings	0,046902563
Red Black	lotr	0,053016359
SPlay Tree	lotr	0,081438538
Binary Sea	lotr	0,09432255
Red Black	aspellRandom	0,112472264
Binary Sea	aspellRandom	0,14087532
SPlay Tree	aspellRandom	0,219299987
Red Black	kjb	0,238552836
SPlay Tree	kjb	0,325524816
Binary Sea	kjb	0,37177491
Red Black	rndStrings1500k	0,698781214
Binary Sea	rndStrings1500k	0,71520999
SPlay Tree	rndStrings1500k	1,470092551
Binary Sea	aspell	74,00360717
Binary Sea	aspellReversed	82,31645647

Tree	TXT	Delete Time
Binary Sea	aspell	0,00981428
Binary Sea	aspellReversed	0,00982451
Binary Sea	randomStrings	0,02507042
Red Black	randomStrings	0,026730228
SPlay Tree	aspellReversed	0,0313763
Red Black	aspellReversed	0,032251898
Red Black	aspell	0,032798998
SPlay Tree	randomStrings	0,04251578
Binary Sea	lotr	0,06033091
Red Black	lotr	0,091209378
Binary Sea	aspellRandom	0,0968237
Red Black	aspellRandom	0,097269273
SPlay Tree	lotr	0,162412899
SPlay Tree	aspellRandom	0,173130858
Binary Sea	kjb	0,31456171
Red Black	kjb	0,431444456
Binary Sea	rndStrings1500k	0,99916848
Red Black	rndStrings1500k	1,315741818
SPlay Tree	rndStrings1500k	2,530021421
SPlay Tree	kjb	5,54281515
SPlay Tree	aspell	69,86070314

Figure 4. Sorted average results

By time performance on all operations

BST and Splay are viable options if it is sure what kind of operations will be used the most and what kind of data will be given.

Analysis conclude these final instructions:

- Use Splay if input will be always in alphabetical order (increasing or decreasing) and the data won’t be deleted with search operation used sparingly.
- Use BST if there was mysterious case found in which insertion and search is irrelevant and all that matters is deletion time.
- Otherwise use RBT

References

http://wikipedia.com/ *Subpages about BST, RBT, Splay Tree*. Pages as of 19.05.2019

http://geeksforgeeks.com/ *Subpages about BST, RBT, Splay Tree*. Pages as of 19.05.2019