Experimental Analysis of Sequential Search Heuristics

Muhammad Arsal Asif (asifma@myumanitoba.ca)

November 21, 2019

Abstract

In this paper, the performance of sequential search heuristics is analyzed. The literature review of sequential search heuristics is done, resulting in the selection of Move-to-Front, Transpose, and K-in-a-Row for this paper. The described literature review is also used to identify previously known trends regarding the chosen heuristics. Markov chain model is selected for access sequence generation. The access sequences are divided into four categories based on different probabilities. In this regard, probabilities are selected manually, and by using Zipfian distribution. All heuristics are tested with generated access sequences on implementations of lists and trees. Linked lists, Binary search trees, and Splay trees are used for this purpose. Access costs, rotation costs, and total costs are reported. Finally, analyses of results from the experiments are done.

1 Introduction

Any data structure is used to store records (or keys). A sequential search on a data structure is defined as sequentially moving through records until the searched key is found. As such, the process involves traversal of the data structure from the start till the location where the key is located.

Any optimal offline algorithm knows the sequence in advance, and as such, it can sort data structure to reduce overall access cost. In contrast, online algorithms are unaware of access sequences beforehand, and they rely on search heuristics to improve efficiency.

1.1 Sequential Search Heuristics

Sequential search heuristics are search heuristics defined to search sequential data structures efficiently. The goal of any search heuristic is to reduce the overall access cost of any sequence σ. This is done by exploiting locality in data. The simple idea is that any accessed item is likely to be reaccessed. As such, different heuristics are derived from promoting accessed items.

Various search heuristics are introduced in literature over the years. The most popular search heuristic is *Move-to-Front* (MTF) which was introduced by McCabe in 1965 [1]. MTF states that on each access to an item, move it to the front of the data structure. Considering the principle of locality, if an accessed item appears multiple times stacked together in a sequence, the subsequent accesses to that item will cost zero if it was moved to the front on the first access and if no item was accessed since its last access.

Another popular heuristic is *Transpose* [1]. Transpose involves swapping the accessed item with its predecessor (an item that appears before it). Rivest proved Transpose to have a lower cost than MTF for some special distributions [2]. Bitner et al. also backed this conjecture for given distributions [3]. Similar

heuristics to Transpose are identified in literature under different names: Simple Exchange [4], Move-to-Parent [5], Swap-with-Parent [5]. All the following heuristics work same as Transpose, i.e., swapping the item with the predecessor.

Frequency Count [6] is a popular heuristic focusing on the number of accesses (frequency) to any item. It works by having a separate data structure keeping count of accesses to each item. Frequency Count is noted to work well, but it compromises on space complexity.

Some of the other popular heuristics are:

- Move-to-Rear: Move an accessed item to the rear [7].
- Move-ahead-k: Move an accessed item k positions ahead [2].
- Wait c, Move-and-Clear: Wait for c accesses to an item, use either MTF or Transpose on accessed item after c accesses, then clear the counter of c to zero [8].

Gonnet et al. [9] gave two interesting heuristics: *K-in-a-Batch* and *K-in-a-Row*. For K-in-a-Batch, requests are grouped into batches of size k and a requested item is moved (using Transpose or MTF) if it is requested k times in a batch. K-in-a-Row requires waiting for k consecutive accesses to an item and then performing MTF or Transpose on it. It works similar to heuristic by Bitner [8], but moves item only on consecutive accesses.

Search heuristics are often defined with the data structure in mind. Some heuristics are noted to perform better on trees, some on heaps and some on lists. Swap-with-Parent [5], for example, considers its application on heaps. It is also noted to work best on unary trees. K-in-a-Batch or K-in-a-Row is noted to outperform MTF on lists but not on trees [9].

In this project, Move-to-Front [1], Transpose [2], and K-in-a-Row [9] are considered. Gonnet et al. demonstrated K-in-a-Row with a value of $K = 2$, and for this project, the same value is used. For the remainder of the paper, this heuristic will be referred to as 2-in-a-Row.

1.2 Data Structures

1.2.1 Linked List

Linked List is the most common data structure used to implement lists. It is one of the chosen data structures for implementation in this project. Implementing the MTF, Transpose and 2-in-a-Row search heuristics results in *self-organizing* linked lists.

1.2.2 Binary Search Trees

Binary Search Tree (BST) is the classical static tree data structure for sequential search on items. On access to an item, if the searched value is greater than the current node, search moves onto the right subtree. If the searched value is less than the current node, it moves to the left subtree. If the searched value is equal to the value of the node, the current node is returned. An item is inserted into a binary search tree in the same way.

1.2.3 Splay Trees

Sleator and Tarjan [10] introduced Splay trees. The intuition behind splay trees is *self-organizing* data structures. For that purpose, splay trees use *splay* as a restructuring heuristic. Splaying is done by performing

a series of rotations to restructure the tree. On each access to an item, the item is searched through binary search then moved to the front (MTF heuristic) through restructuring by splaying. Sleator and Tarjan define splay trees to perform splaying on insertion, deletion, and search. However, for the scope of this project, we will consider splaying only on search. It is because this project's scope is limited to the sequential search complexity, and for a fair comparison, it was necessary for each algorithm to start with the same tree.

1.3 Related Work

Compared to theoretical analysis, less work is done on experimental analysis of sequential search heuristics. Most of the literature on experimental analysis is based on either comparing data structures (which inherently use these heuristics) [11] [12] or comparing a classical search heuristic such as MTF with a newly introduced heuristic [13] [14] [15] [16] [17] [18]. Another part of the literature is focused solely on comparing search heuristics on one data structure such as lists or trees [19] [20]. Sometimes, the analysis is also focused on an application such as data compression [21]. As such, these experimental analyses are not directly related to the one in this paper.

More closely related to this project are analyses by Rivest [2], Bitner [8], Bentley and McGeoch [6]. Rivest briefly shows a result of comparing transpose with MTF. Bitner also compares the two heuristics along with the heuristic introduced in the paper on a sequence of 1000 requests maximum. Both papers consider Zipfian distributions [22] for their analyses. Bentley and McGeoch used a dictionary of words (which also follows Zipf's law) to compare MTF, Transpose and Frequency Count. Their results show that MTF is better than Transpose, which is in contradiction to the probabilistic analysis by Rivest. Note that these papers are dated, and at the time of their writing, the computational resources available for a large-scale experimental analysis were not sufficient. However, their theoretical analyses are of more significance and are considered in the results section of this paper.

2 Methodology

2.1 Zipf's Law

George Kingsley Zipf proposed Zipf's law in 1935 [23] [24]. Zipf's law relates to word frequencies in the English language. Zipf's law is an empirical law in mathematical statistics. It states that the frequency of an item is inversely proportional to its rank. Various corpus of text such as the Calgary and Canterbury corpus [25] were created based on Zipf's law. Zipf distribution (based on Zipf's law) can be used to generate probabilities following Zipf's law for any input data.

2.2 Markov Chain

Markov chain is a system for defining states with transitions and assigning probabilities to each transition [26] [27]. Any input data can be assigned to states in the Markov chain, along with probabilities of going from one state to another. Transitions can be between states or to themselves (self-looping). Markov Chain is widely studied in the case of MTF and Move-to-Root search heuristic [28] [29] [30] [31] [32]. Rivest used Markov chains for probabilistic analysis on MTF and Transpose [2]. Similarly, Bitner [8] also used Markov chain.

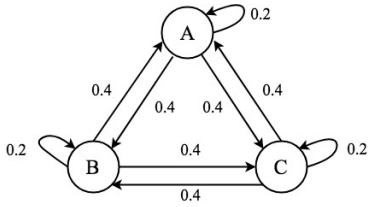

Figure 1: Markov Chain

Figure 1 shows a simple Markov chain model with 3 states. The probability of going from one state to another is 0.4, whereas the probability of self-looping is 0.2.

2.3 Input and Access Sequence Generation

Choice of input data and access sequence is crucial to the experiment. After carefully running the experiment on multiple input data sizes, a sequence of integers from 1 to 1000 $\langle 1, 2, 3, \ldots 998, 999, 1000 \rangle$ were selected. On each run, the list of integers is randomly shuffled before any further processing on it. The random shuffling guarantees a more balanced tree.

After generation of the list of integers, we move on to the generation of access sequence. Integers are considered as states of the Markov chain model and probabilities assigned to them to generate access sequences. Every time a state is selected, the value of that state (in this case, an integer from input data) is added to the access sequence.

For each experiment, access sequences of sizes $\langle 100000, 200000, 300000, 400000, 500000, 600000,$ $700000, 800000, 900000, 1000000 \rangle$ were generated.

For this project, four types of access sequence data were derived using the Markov chain model. All four types of sequences have high locality. The sections below describe the four types.

2.3.1 High Self Loop

This type of access sequence is referred to as *High Self Loop* in this paper. For this sequence, a probability of 0.9 was assigned to self-loop, and a probability of 0.1/999 was assigned on each transition from one state to another. This results in 0.1 probability of going from one state to another and 0.9 probability of looping at the same state. The result is that there is a high amount of self-looping before moving to another state. As such, access sequences consist of large amounts of consecutive accesses before a different item appears in the sequence.

2.3.2 Medium Self Loop

This type of access sequence is referred to as *Medium Self Loop* in this paper. For this sequence, a probability of 0.5 was assigned to self-loop, and a probability of 0.5/999 was assigned on each transition from one state to another. This results in 0.5 probability of going from one state to another and 0.5 probability of looping at the same state. Access sequences of this type of data also consist of more consecutive accesses. However, the consecutive accesses are less in comparison to the high self-loop sequence.

2.3.3 Low Self Loop

This type of access sequence is referred to as *Low Self Loop* in this paper. For this sequence, a probability of 0.1 was assigned to self-loop and probability of 0.9/999 was assigned on each transition from one state to another. This results in 0.9 probability of going from one state to another and 0.1 probability of looping at the same state. Access sequences of this type of data have considerably less consecutive accesses compared to the other two types mentioned before. However, it still has high locality when compared with a uniform distribution.

2.3.4 Zipfian

This type of access sequence is referred to as *Zipfian* in this paper. For this sequence, probabilities of transitioning from one state to another were determined by Zipfian distribution. Rank of each state defines the probability of another state transitioning to it. The probability of self-looping for each state is same as the probability of another state transitioning to it. Lower ranked states are visited more frequently, and higher ranked states have a very low probability. A common practice to verify if probabilities and rank of input data follow Zipfian distribution is to plot a log-log chart. Figure 2 shows the log-log chart of log(rank) plotted against log(probabilities) for the selected input data.

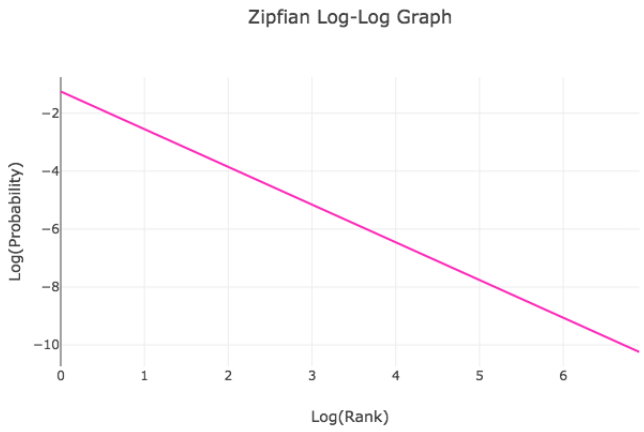

Figure 2: Log-Log chart

2.4 Data Structures and Implementation

All three chosen heuristics are implemented on linked lists. For MTF, the accessed item is moved to the front of the linked list. For Transpose, the accessed item is swapped with the parent node in the linked list. For 2-in-a-Row, on each consecutive access to an accessed item, it is moved to the front of the linked list.

Splay trees are implemented for the experiment on trees. Splay trees inherently follow MTF rule, and as such, no extra steps were required for it. For transpose, splaying was done on the subtree having the parent of the accessed node as root, this results in accessed node getting swapped with parent node through one

rotation. 2-in-a-Row also used the splaying. On each consecutive access to an item, 2-in-a-Row performed splay on the tree to move the accessed item to root.

2.4.1 Optimal Offline Algorithm

Any analysis of online algorithms is incomplete without a static optimal offline algorithm (OPT) [33]. The implementation of OPT is as studied in class. OPT knows the access sequence in advance, and as such, it knows the frequency of accesses to any item. The list of input data is initially sorted by items with decreasing frequency of accesses to minimize access cost for OPT. For both linked list and trees, the input data is inserted in this way.

OPT is implemented by static linked list sequential searching. For trees, OPT does not use splay trees, but rather a static binary search tree (BST).

2.4.2 Nuts and Bolts

Java was used for all implementation and experiments.
`ZipfDistribution` was used to generate probabilities of input data based on Zipfian distribution.
`EnumeratedIntegerDistribution` was used to select state transitions based on given probabilities. Both of these classes can be found in Java Common Maths library [34]. Linked list, binary search tree, and splay tree are implemented in this project. Results of the experiments were written to CSV files. The output CSV files were then accessed through python code to generate plots and figures for results section of this paper. Plotly library of python was used for this purpose [35]. For more implementation details, the code can be accessed through the GitHub link in references [36].

3 Results

The five sections below describe the results. The first four sections are according to four types of access sequence data selected before. Each section consists of charts of access costs, rotation costs, total cost (access cost + rotation cost) plotted against varying sequence lengths on both lists and trees. Each section also consists of charts of input data along with their frequency (in percentage) in access sequence. To supplement these figures, tables with values that were plotted on these charts are added to the appendix. Additionally, an analysis comparing results on all four types of access sequence data is done in the last section.

3.1 High Self Loop

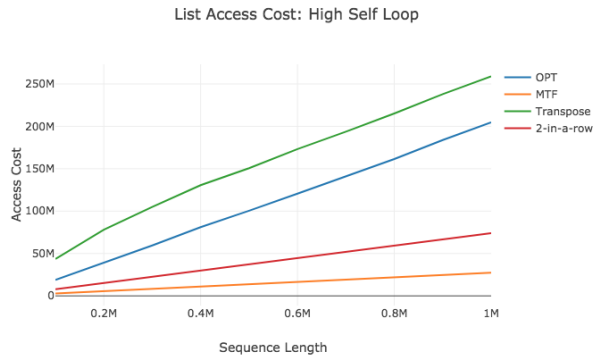

Figure 3: List Access Cost: High Self Loop

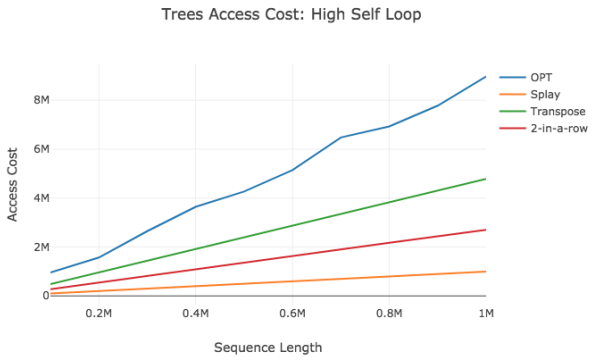

Figure 4: Trees Access Cost: High Self Loop

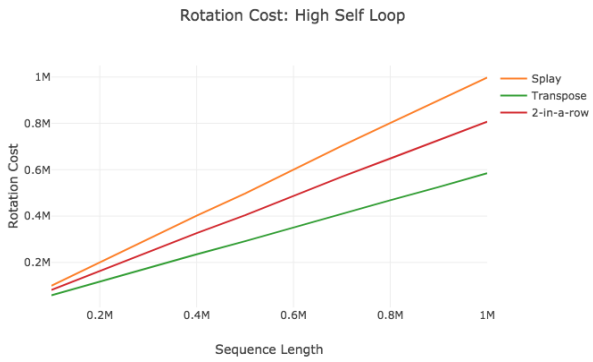

Figure 5: Rotation Cost: High Self Loop

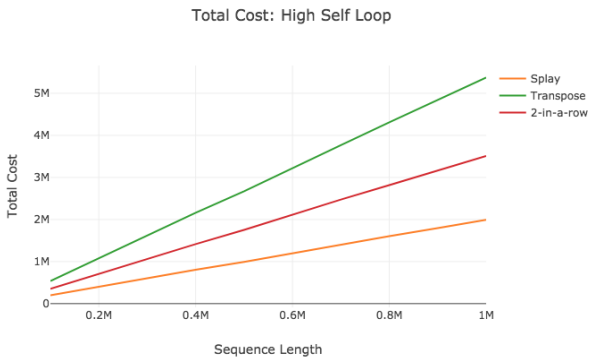

Figure 6: Total Cost: High Self Loop

3.1.1 List Access Cost

Figure 3 shows the chart of access costs on lists for all four algorithms. OPT costs roughly seven times more than MTF, whereas Transpose costs roughly nine times more. 2-in-a-Row performs fairly better than Transpose and OPT but still costs more than twice as much as MTF. Cost of each algorithm increases linearly. However, we can see a far more increase in cost in both OPT and Transpose as compared to MTF and 2-in-a-Row. We can say that as access sequence size increases, the performance difference between OPT and MTF increases greatly. Note that access costs are in factors of millions. Consider the case where access sequence size is 1 million, the difference between the cost of OPT and MTF is $\approx$ 178 million, which is huge. MTF outperforms all other three algorithms.

3.1.2 Trees Access Cost

Figure 4 shows the chart of access costs on trees for all four algorithms. A similar trend to performance on lists can be observed here for 2-in-a-Row and Splay. However, it is noteworthy to see that highest access cost (sequence size 1 million) for any of the algorithms is less than 10 million, which is a great improvement from the highest access cost of over 250 million, incurred on lists. We can say that using trees instead of lists greatly improved performance.

Another noteworthy observation here is that Transpose performs significantly better than OPT on trees, whereas it performed worse on lists.

3.1.3 Rotation Cost

The performance measure of self-organizing trees includes measuring rotation costs, in addition to access costs. In any practical scenario, performance depends on CPU time that any algorithm uses. If an algorithm has low access cost, but very high rotation cost, it will still use a lot of CPU time. For this reason, it is necessary to look at rotation costs of the three algorithms on trees.

Figure 5 shows the rotation costs. The trend here is fairly straightforward. Splay has the most rotation cost, as it splays on every access. 2-in-a-Row splays on every consecutive access, and as such, it has less rotation cost than Splay but still considerably high. Transpose has the least rotation cost as it performs only one rotation per access.

3.1.4 Total Cost

Figure 6 shows the total costs. We can see that adding both access and rotation cost, Splay still outperforms the other two algorithms. Access costs are in order of millions whereas rotation costs are less than a million. As such, the addition of both still favors Splay trees.

3.1.5 Frequency of Input Data

Figure 7: Frequency: High Self Loop

Figure 7 shows the frequency of input data in each of the access sequence sizes. Recall that OPT sorts input data by decreasing frequency. Looking at Figure 7, we can see that OPT gains no clear advantage by performing this operation when frequencies of input data are not skewed to favor some input data over others. Since OPT started with a tree with items of lower frequency further down the tree, consecutive access to these items increase access cost significantly.

Notice the slight discrepancy (spikes) in access costs of OPT on trees. Various sequence and input data sizes were experimented on to see if trends were similar. For each experiment, there was always some discrepancy. It can be attributed to the fact that access sequences are created on run-time based on the probability of going from one state to another. Since the probability of self-looping is higher than the probability of going from one state to another, it is possible that some items having less frequency in the final access sequence appeared a large number of times consecutively. It is equally likely that they appeared fewer times. The key idea here is not the frequency of items in generated access sequence, but rather their consecutive accesses.

3.2 Medium Self Loop

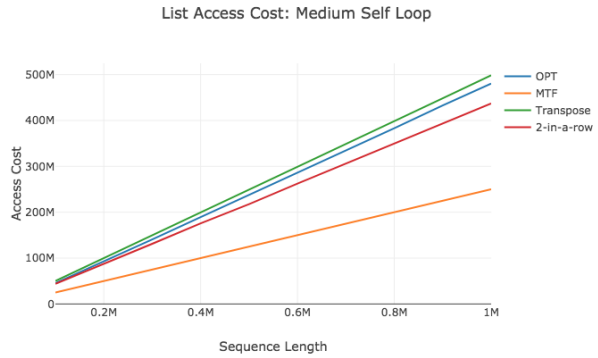

Figure 8: List Access Cost: Medium Self Loop

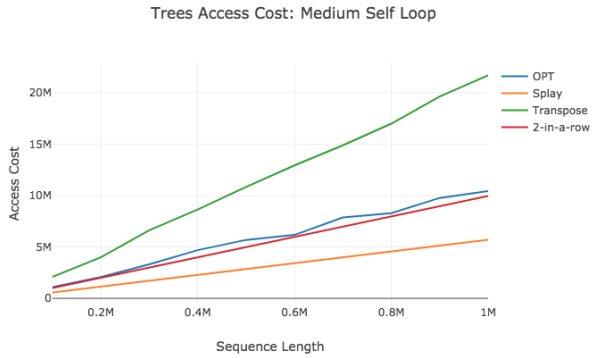

Figure 9: Trees Access Cost: Medium Self Loop

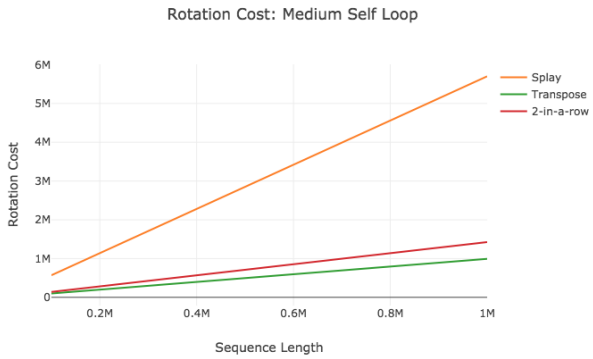

Figure 10: Rotation: Medium Self Loop

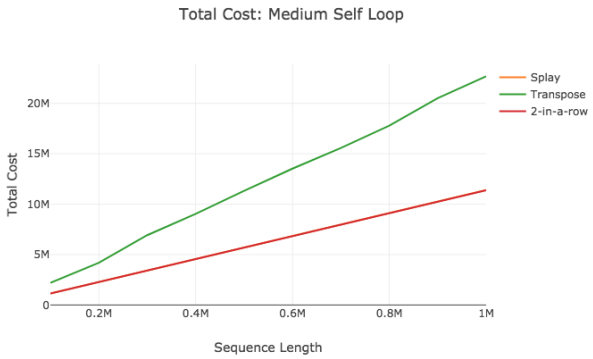

Figure 11: Total Cost: Medium Self Loop

3.2.1 List Access Cost

Figure 8 shows the chart of access costs on lists for all four algorithms. It follows similar trend to what we saw on High Self Loop. However, access cost of each algorithm has almost doubled.

3.2.2 Trees Access Cost

Figure 9 shows the chart of access costs on trees for all four algorithms. Access cost for OPT on this type of access sequence stayed same as High Self Loop (around 10 million). However, the access cost of all other algorithms increased significantly. Due to this, Transpose had worse cost than all other algorithms. Despite the increase in access costs from the previous type of sequence and cost of OPT staying the same, Splay still performs the best.

3.2.3 Rotation Cost

Figure 10 shows the rotation costs. A similar trend to High Self Loop can be seen here. The exception is rotation cost of Splay. Rotation cost of Splay is significantly high on this access sequence compared to high self-loop as there are less consecutive accesses now, and as such, requiring more splaying.

3.2.4 Total Cost

Figure 11 shows the total costs. We can see that adding both access and rotation cost, Splay still outperforms the other two algorithms. Access costs are in order of millions whereas rotation costs are less than a million. Note that for Splay trees, rotation cost equals access cost. On each access, for each level accessed, Splay tree performs a left or right rotation on that level. The plot of Splay is shadowed behind the line of 2-in-a-Row in the above figure, but adding numbers through tables in the appendix, it is clear that the total cost of Splay trees is close to 2-in-a-Row.

3.2.5 Frequency of Input Data

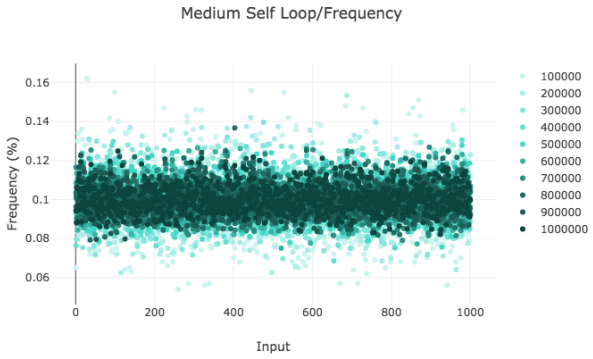

Figure 12: Frequency: Medium Self Loop

Figure 12 shows frequency of input data in access sequences generated through Medium Self Loop. This explains why OPT performs similarly to before on this type of access sequence data on trees. Previously, the frequency was more skewed, resulting in a less balanced tree, whereas it is now more concentrated around the same point, resulting in a more balanced tree. As explained before, in the case of High Self Loop, OPT suffered when less frequent items had more consecutive accesses, but OPT performed better when more frequent items had more consecutive accesses. In this case, the frequency is less skewed, but consecutive accesses are also significantly reduced, as such the advantage and disadvantage of OPT are both balanced, so it performs similarly as before.

3.3 Low Self Loop

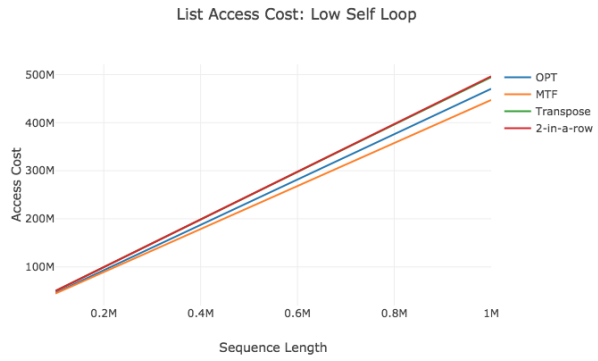

Figure 13: List Access Cost: Low Self Loop

Figure 14: Trees Access Cost: Low Self Loop

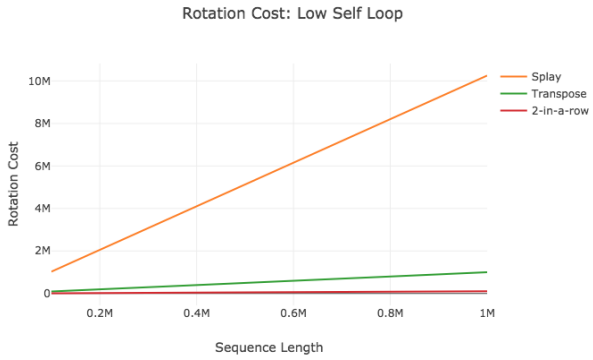

Figure 15: Rotation Cost: Low Self Loop

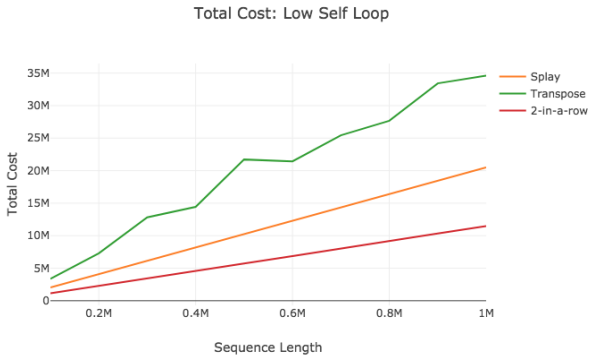

Figure 16: Total Cost: Low Self Loop

3.3.1 List Access Cost

Figure 13 shows the chart of access costs on lists for all four algorithms. Access sequence has less frequent consecutive accesses now, and as such, the costs of algorithms start to converge, and the cost of OPT gets closer to best performing algorithm MTF. Notice that the cost of OPT is still close to the cost of OPT in the previous two access sequence data. However, the cost of other algorithms, especially MTF increased significantly. MTF's advantage was due to consecutive accesses, and with far less consecutive accesses now, MTF does not perform as well. Cost of 2-in-a-Row and Transpose also increased, but there is not a significant difference.

3.3.2 Trees Access Cost

Figure 14 shows the chart of access costs on trees for all four algorithms. Cost of OPT still stays around 10 million for the largest sequence, and it follows the same trend as the previous two types of access sequences. Costs of other algorithms, however, are significantly increased. Cost of Splay has increased almost tenfold from the High Self Loop variant. There is more discrepancy in the cost of Transpose now. There are considerably less consecutive accesses now. Transpose swaps accessed node with the parent and accessed node changes more frequently. As such, transpose gets an unbalanced structure for trees every time. For some iterations, it can be very unbalanced, for some it can be more balanced, thereby causing spikes in costs. OPT's discrepancy is due to the similar reason as described in previous sections.

3.3.3 Rotation Cost

Figure 15 shows the rotation costs. Rotation cost for Transpose and 2-in-a-Row are similar to the trend for Medium Self Loop. However, Splay did almost twice as many rotations now. Compared to high and medium self-loop, Splay performs more rotations on this type of access sequences as there are less consecutive accesses now.

3.3.4 Total Cost

Figure 16 shows the total costs. Contrary to previous two types of access sequences, Splay has surpassed 2-in-a-Row in total cost. This is attributed to Splay performing far more rotations on Low Self Loop. Total cost for Transpose varied a lot. Again, it is because of the facts mentioned in earlier sections.

3.3.5 Frequency of Input Data

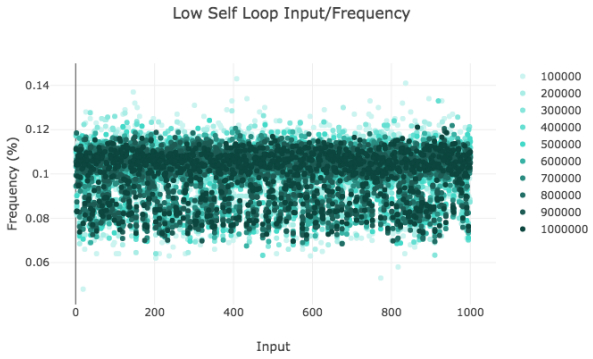

Figure 17: Frequency: Low Self Loop

Figure 17 shows the frequency of input data in access sequences generated through Low Self Loop. Frequency is similarly skewed to that of input on Medium Self Loop. As such, the analysis of frequency on Medium Self Loop applies here as well.

3.4 Zipfian

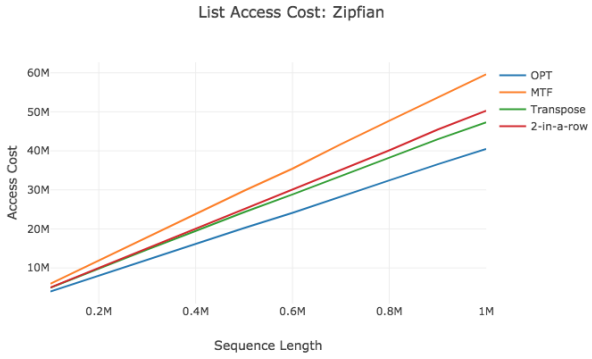

Figure 18: List Access Cost: Zipfian

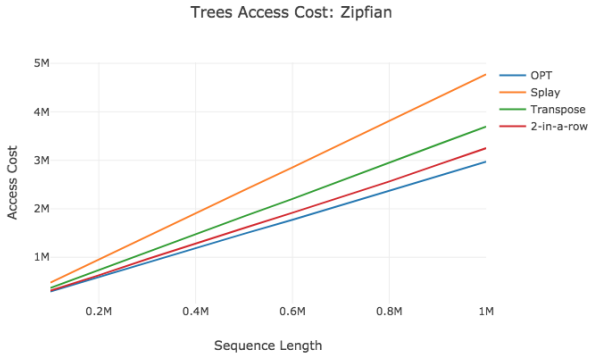

Figure 19: Trees Access Cost: Zipfian

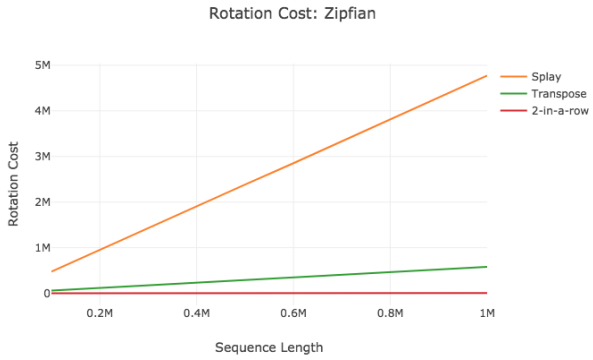

Figure 20: Rotation Cost: Zipfian

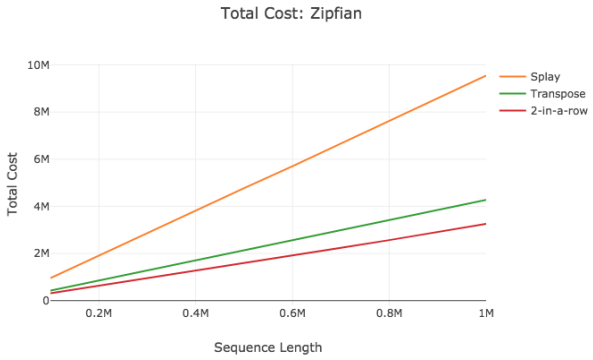

Figure 21: Total Cost: Zipfian

3.4.1 List Access Cost

Figure 18 shows the chart of access costs on lists for all four algorithms. OPT outperforms all other algorithms. As explained before, Zipfian distribution assigns the probability of items based on their rank resulting in a frequency distribution which is more skewed towards the lower ranked item. OPT gains advantage by sorting list initially based on frequency. Contrary to previous results, MTF performs worse than the other three algorithms. MTF brings every accessed item to the front, regardless of its frequency, as such, it brings lower frequency items to the front on each access as well. Thereby, increasing its cost greatly. Transpose performs better than both 2-in-a-Row and MTF because less frequent items are not immediately brought to the front by Transpose.

3.4.2 Trees Access Cost

Figure 19 shows the chart of access costs on trees for all four algorithms. OPT performs best on trees as well. An interesting observation here is that 2-in-a-Row performed better on trees than Transpose, as compared to Transpose performing better on lists. This is in line with the conjectures made in literature, as explained in the introduction section of this paper, that some heuristics perform better on trees, and some on lists. Another interesting observation is that Transpose performs better than MTF on both lists and trees, supporting the probabilistic analysis of Rivest [2]. As noted before, Rivest also used a Zipfian distribution.

3.4.3 Rotation Cost

Figure 20 shows the rotation costs. Splay incurs a much higher rotation cost than both Transpose and 2-in-a-Row. The reasons for this are as described in previous sections.

3.4.4 Total Cost

Figure 21 shows the total costs. A clear trend follows here. Since both rotation and access cost of Splay is high, it has the highest total cost.

3.4.5 Frequency of Input Data

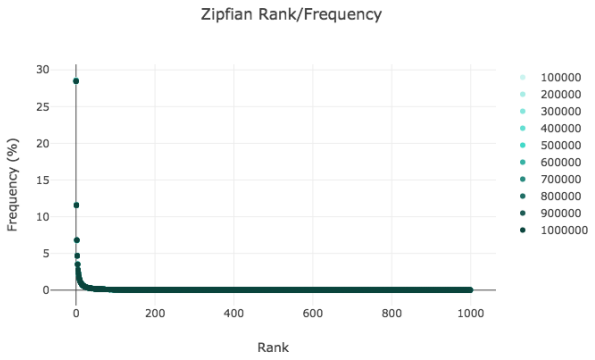

Figure 22: Frequency: Zipfian

Figure 22 shows the frequency of the rank of data. Zipfain frequencies are based on rank (in this case index of input data in the list), we can see that lower ranked items have a high frequency and it decreases as we move towards higher ranked items.

3.5 Putting It All Together

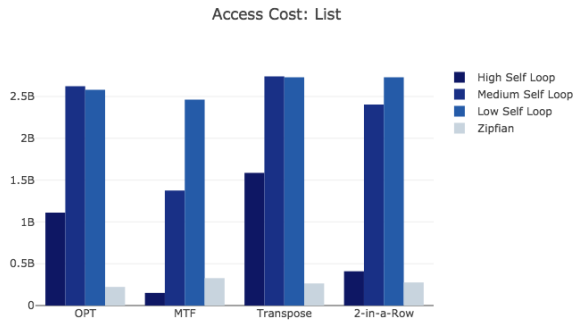

Figure 23: Access Cost: List

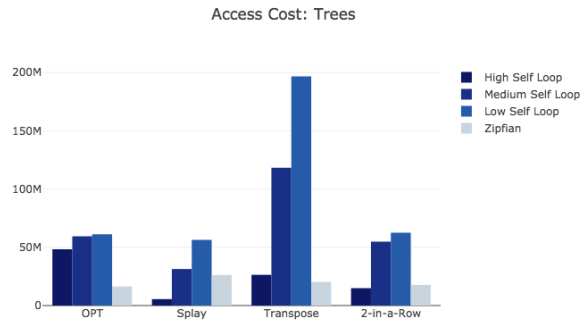

Figure 24: Access Cost: Trees

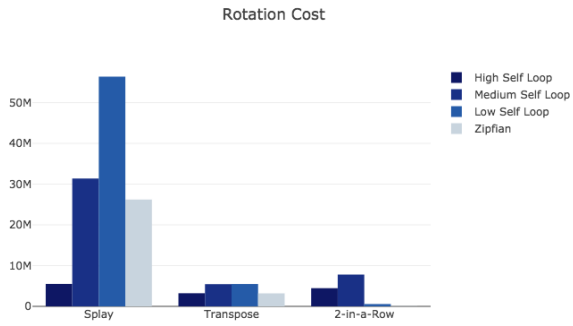

Figure 25: Rotation Cost

Figure 23 shows the sum of all access costs of all algorithms on all four types of access sequence data on lists. As we saw in previous sections, access costs of all algorithms start to converge as we go from high to medium to low self-loop. The gap closes between all four algorithms, and we move towards less difference in access costs. There is more than a billion difference in sum of all access costs for high self-loop, but it closes to 0.2 billion on low self-loop. Zipfian has the least access cost than other three access sequences, except for MTF having a higher cost of Zipfian distribution than high self-loop.

Figure 24 shows the sum of all access costs of all algorithms on all four types of access sequence data on trees. Access costs on trees for any algorithm are a lot less than on lists. The difference in costs is also very less, in order the of millions as compared to billions that we saw for lists. Similar to lists, access costs start to converge as we go towards low self-loop. However, the access cost for Transpose increases significantly

due to the reasons described before. Zipfian has the same trend as we saw for lists.

Figure 25 shows the sum of all rotation costs of all algorithms on all four types of access sequence data on trees. Splay has a significantly high rotation cost regardless of input. As we move closer to low self-loop, the rotation cost for Splay increases. Rotation cost for Transpose does not increase as much. For Zipfian, 2-in-a-Row has minimal rotation cost.

4 Conclusion

Detailed analysis of selected heuristics on all types of access sequences is done. Some notable findings are performance differences between different types of access sequences. Move-to-Front, a popular heuristic, performed significantly better on the self-loop type of access sequences, whereas it performed significantly worse on Zipfian distribution. Access sequences using Zipfian distributions are generally considered in literature [2] [8] and the findings of this paper support the claim for the performance of these heuristics on these distributions. However, in the case of self-loop, the findings contradict with what is well-known in the literature. The papers comparing these heuristics are generally focused on comparing them on selected specific distributions, but we saw in the results that performance is highly dependant on the type of access sequence data.

Secondly, performance also depends highly on the type of data structure used. All heuristics performed significantly well on trees, regardless of the type of access data. No single heuristic can be declared as universally best performing due to these factors.

In conclusion, if aim to use a sequential search heuristic, it is essential to identify what type of access sequence data can be expected. For example, Zipf's law is popular in English text, and if the access sequence is known to be English text, then perhaps it is better to use 2-in-a-Row or Transpose. In another case, if the access sequence is known to have large consecutive accesses to the same item with more uniform frequency distribution, perhaps using MTF is the best choice. Implementation can be done on trees if space complexity is not an issue, as trees are shown to be more efficient on all heuristics.

For future work, different probability distributions can be explored to be used with Markov models. There are many different sequential search heuristics, and future experiments can include them. Data structures other than lists and trees, such as heaps, can be explored.

References

[1] John McCabe. On serial files with relocatable records. *Operations Research*, 13(4):609–618, 1965.

[2] Ronald Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19(2):63–67, 1976.

[3] James Richard Bitner. *Heuristics that dynamically alter data structures to reduce their access time.* PhD thesis, University of Illinois at Urbana-Champaign, 1976.

[4] Brian Allen and Ian Munro. Self-organizing binary search trees. *Journal of the ACM (JACM)*, 25(4):526–535, 1978.

[5] John Oommen and Juan Dong. Generalized swap-with-parent schemes for self-organizing sequential linear lists. In *International Symposium on Algorithms and Computation*, pages 414–423. Springer, 1997.

[6] Jon L Bentley and Catherine C McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, 1985.

[7] Vladimir Estivill-Castro. Move-to-end is best for double-linked lists. In *Computing and Information, 1992. Proceedings. ICCI'92., Fourth International Conference on*, pages 84–87. IEEE, 1992.

[8] James R Bitner. Heuristics that dynamically organize data structures. *SIAM Journal on Computing*, 8(1):82–110, 1979.

[9] Gaston H Gonnet, J Ian Munro, and Hendra Suwanda. Toward self-organizing linear search. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 169–174. IEEE, 1979.

[10] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[11] Douglas W Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[12] Yiqun Zhao. *An Experimental Study of Dynamic Biased Skip Lists*. PhD thesis, 2017.

[13] Abdelrahman Amer and B John Oommen. A novel framework for self-organizing lists in environments with locality of reference: Lists-on-lists. *The Computer Journal*, 50(2):186–196, 2006.

[14] Rakesh Mohanty and Ashirbad Mishra. Performance evaluation of a novel most recently used frequency count (mrufc) list accessing algorithm. In *Proceedings of International Conference on Advances in Computing*, pages 267–276. Springer, 2013.

[15] Debashish Rout. Experimental analysis of novel variant of mfk algorithm. *International Journal of Science, Engineering and Technology Research (IJSETR)*, 2(12), 2013.

[16] Debashish Rout. Experimental analysis of novel variant of bit algorithm. *Journal of Computer and Mathematical Sciences 4, volume=4, number=6, year=2013*.

[17] Rakesh Mohanty, Shiba Prasad Dash, Burle Sharma, and Sangita Patel. Performance evaluation of a proposed variant of frequency count (vfc) list accessing algorithm. *arXiv preprint arXiv:1206.6185*, 2012.

[18] Rakesh Mohanty, Tirtharaj Dash, Biswadeep Khan, and Shiba Prasad Dash. An experimental study of a novel move-to-front-or-middle (mfm) list update algorithm. In *International Conference on Applied Algorithms*, pages 187–197. Springer, 2014.

[19] Donald R Fletcher. *An Investigation of Self-Organizing Binary Search Trees*. PhD thesis, 1977.

[20] Jim Bell and Gopal Gupta. An evaluation of self-adjusting binary search tree techniques. *Software: Practice and Experience*, 23(4):369–382, 1993.

[21] Reza Dorrigiv, Alejandro López-Ortiz, and J Ian Munro. Experimental evaluation of list update algorithms for data compression. *School of Computer Science, University of Waterloo, Ontario, Canada, Tech. Rep. CS–2007–38*, 2007.

[22] Zipfś Law. `https://www.britannica.com/topic/Zipfs-law`. Accessed: 2018-12-09.

[23] George K Zipf. The psychology of language. 1935.

[24] George K Zipf. Human behavior and the principle of least effort. 1949.

[25] Ian H Witten, Ian H Witten, Alistair Moffat, Timothy C Bell, Timothy C Bell, and Timothy C Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[26] Bertrand M. Hochwald and Predrag R. Jelenkovic. State learning and mixing in entropy of hidden markov processes and the gilbert-elliott channel. *IEEE Transactions on Information Theory*, 45(1):128–138, 1999.

[27] Markov Chain. `https://brilliant.org/wiki/markov-chains/`. Accessed: 2018-12-09.

[28] Edward G Coffman Jr and Predrag Jelenković. Performance of the move-to-front algorithm with markov-modulated request sequences. *Operations Research Letters*, 25(3):109–118, 1999.

[29] Robert P Dobrow. The move-to-root rule for self-organizing trees with markov dependent requests. *Stochastic Analysis and Applications*, 14:73–88, 1996.

[30] RM Phatarfod and AJ Pryde. On some multi-request move-to-front heuristics. *Journal of applied probability*, 35(4):911–918, 1998.

[31] Robert P Dobrow and James Allen Fill. On the markov chain for the move-to-root rule for binary search trees. *The Annals of Applied Probability*, pages 1–19, 1995.

[32] Eliane R Rodrigues. The performance of the move-to-front scheme under some particular forms of markov requests. *Journal of applied probability*, 32(4):1089–1102, 1995.

[33] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. cambridge university press, 2005.

[34] Java Common Maths. `http://commons.apache.org/proper/commons-math`. Accessed: 2018-12-09.

[35] Plotly. `https://plot.ly/`. Accessed: 2018-12-09.

[36] Github. `https://github.com/arsalasif/online-algo-project`. Accessed: 2018-12-09.

A Supplementary Tables for Plotted Graphs

Sequence Length	OPT	MTF	Transpose	2-in-a-Row
100000	18840487	2824874	43565449	7747610
200000	39374774	5573957	78028732	15161209
300000	59495178	8295773	104893923	22549460
400000	81068304	11103060	130575668	30206794
500000	100401139	13666431	150601610	37085479
600000	120580988	16464541	173151147	44625678
700000	140882949	19222042	193686750	52188531
800000	161393711	21866960	215176911	59135758
900000	183800505	24700347	238040197	66901486
1000000	204611155	27279198	258914409	74026985

Figure 26: List Access Cost: High Self Loop Table

Sequence Length	OPT	Splay	Transpose	2-in-a-Row
100000	959379	99451	477872	270336
200000	1569260	201016	957303	544994
300000	2647567	302525	1448942	819826
400000	3642666	401883	1925595	1089356
500000	4267548	497020	2381199	1348244
600000	5147682	601205	2878290	1627661
700000	6477961	702804	3358625	1905028
800000	6933217	801825	3841224	2170240
900000	7778557	898868	4310200	2435454
1000000	8976328	997295	4787586	2704259

Figure 27: Trees Access Cost: High Self Loop Table

Sequence Length	Splay	Transpose	2-in-a-Row
100000	99451	58296	81074
200000	201016	117410	162853
300000	302525	177146	245485
400000	401883	234986	326202
500000	497020	291586	403829
600000	601205	350761	485706
700000	702804	410566	569918
800000	801825	468531	648524
900000	898868	525526	728455
1000000	997295	584570	807652

Figure 28: Rotation Cost: High Self Loop Table

Sequence Length	OPT	MTF	Transpose	2-in-a-Row
100000	45181377	25037762	49985488	43869843
200000	92400780	50055451	99410032	86950060
300000	140745038	74810017	149911712	131125448
400000	189445179	100445129	199497506	175737103
500000	237597205	124844653	249383833	217703784
600000	285923170	149666045	298890123	262370924
700000	333773441	174699883	348935223	305621406
800000	383132440	199725354	397343309	349578537
900000	432676243	225110448	448288484	393047862
1000000	480459617	249787664	498271996	436940376

Figure 29: List Access Cost: Medium Self Loop Table

Sequence Length	OPT	Splay	Transpose	2-In-a-Row
100000	1094974	571332	2092669	999818
200000	2070488	1141488	3983621	1989622
300000	3312390	1707701	6626879	2993286
400000	4686023	2286893	8635254	3995694
500000	5683363	2848810	10825968	4978754
600000	6165330	3416822	12928586	5986175
700000	7864355	3987261	14878057	6974268
800000	8275628	4555334	16984774	7967173
900000	9763336	5135119	19625839	8976999
1000000	10437986	5699093	21677383	9961813

Figure 30: Trees Access Cost: Medium Self Loop Table

Sequence Length	Splay	Transpose	2-In-a-Row
100000	571332	98990	141357
200000	1141488	198093	284820
300000	1707701	297038	426025
400000	2286893	396203	570976
500000	2848810	495547	710918
600000	3416822	594148	856792
700000	3987261	693307	994839
800000	4555334	792528	1137303
900000	5135119	891462	1280042
1000000	5699093	990859	1421655

Figure 31: Rotation Cost: Medium Self Loop Table

Sequence Length	OPT	MTF	Transpose	2-In-a-Row
100000	45694019	44678913	49890008	49819762
200000	92911403	89486890	99600715	99271419
300000	140031936	134265936	149443450	149061691
400000	187133098	178971894	199434852	198561266
500000	234074123	223673131	248259122	248301018
600000	281621768	268450716	297735039	297369130
700000	328841877	313041497	347177558	348108169
800000	375949782	358106686	396246588	396970817
900000	423078157	403275426	445988265	446698269
1000000	470602768	447379530	494702337	496282336

Figure 32: List Access Cost: Low Self Loop Table

Sequence Length	OPT	Splay	Transpose	2-In-a-Row
100000	1078117	1023652	3258317	1137117
200000	1988643	2049596	7097258	2292454
300000	3477220	3076516	12512374	3393088
400000	5007981	4099214	14030290	4544920
500000	5384597	5123575	21216475	5662333
600000	6490153	6145490	20840393	6795022
700000	7820440	7170143	24731857	7978365
800000	8588838	8198408	26872933	9088620
900000	10514697	9227191	32526553	10236896
1000000	10765886	10250619	33610216	11386097

Figure 33: Trees Access Cost: Low Self Loop Table

Sequence Length	Splay	Transpose	2-in-a-Row
100000	1023652	99688	10219
200000	2049596	199350	21061
300000	3076516	298999	30289
400000	4099214	398618	40162
500000	5123575	498311	50522
600000	6145490	598119	63684
700000	7170143	697719	70569
800000	8198408	797522	83341
900000	9227191	897187	91900
1000000	10250619	996730	102247

Figure 34: Rotation Cost: Low Self Loop Table

Sequence Length	OPT	MTF	Transpose	2-in-a-Row
100000	3969533	5960464	4954443	4982156
200000	8036409	11954286	9858099	9984502
300000	12037803	17802496	14684913	14982873
400000	16146580	23794133	19476282	20061183
500000	20228952	29775526	24286445	25135712
600000	24114479	35444610	28825452	29996027
700000	28334606	41709142	33550005	35106956
800000	32433918	47732243	38223661	40127316
900000	36554729	53707338	42978784	45469050
1000000	40504183	59594368	47311234	50258352

Figure 35: List Access Cost: Zipfian Table

Sequence Length	OPT	Splay	Transpose	2-in-a-Row
100000	294772	476911	368600	311374
200000	593923	955550	740238	628028
300000	888959	1428436	1103979	963870
400000	1185859	1901414	1474500	1289013
500000	1485342	2385005	1845414	1601918
600000	1774464	2852591	2204904	1919368
700000	2072981	3336039	2579552	2235353
800000	2377250	3820654	2960986	2563114
900000	2675309	4302218	3325867	2908312
1000000	2972686	4772296	3695062	3250934

Figure 36: Trees Access Cost: Zipfian Table

Sequence Length	Splay	Transpose	2-in-a-Row
100000	476911	57808	435
200000	955550	116205	878
300000	1428436	174191	1376
400000	1901414	230939	2286
500000	2385005	289895	2585
600000	2852591	347594	3189
700000	3336039	405914	3611
800000	3820654	464200	4523
900000	4302218	521669	4732
1000000	4772296	579471	5229

Figure 37: Rotation Cost: Zipfian Table