Comparing Implementations of Optimal Binary Search Trees

Corianna Jacoby and Alex King
Tufts University
May 2017

Introduction

In this paper we sought to put together a practical comparison of the optimality of static binary search tree implementations. Therefore, we implemented five different binary search tree algorithms and compared static operations. We look at construction time and expected depth to determine the best implementation based on number of searches before rebuilding. We used real world data, Herman Melville's *Moby Dick*, to generate realistic probability distributions using word frequency. However, we also wanted to observe the performance of the algorithms in more extreme cases, so we additionally generated some edge test cases to test particular types of behavior.

We implemented a suite of static binary tree implementations. Since we were looking at optimality, we started with Knuth's 1970 optimal binary tree implementation, which guarantees an optimal implementation given probabilities for all keys and leaves in the tree. In the same paper Knuth mentions two potential methods for approximately optimal trees with reduced build time. One of which was fully proven and demonstrated by Kurt Mehlhorn, and the other of which was deemed less optimal. We implemented both of these methods to get a full look at approximate methods. We also implemented two non-optimal methods, which did not consider the probabilities of keys and leaves during the construction process. We implemented AVL trees, which approach optimality through simple balancing. We also implemented an unbalanced binary search tree as a baseline test, where values were simply inserted in random order. Each of these algorithms is further described at a high level with resources listed for further study.

Algorithms Studied

Knuth's Algorithm for Optimal Trees

In his 1970 paper "Optimal Binary Search Trees", Donald Knuth proposes a method to find the optimal binary search tree with a given set of values and the probability of looking up each value and searching for a value not in the tree between each consecutive keys. This method relies on dynamic programming to get a construction time of $O(n^2)$. While this is slow, this method is guaranteed to produce a tree with the shortest possible expected search depth.

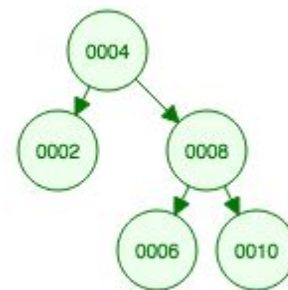
This method works by first generating a table that calculates the probability of each complete subtree. A complete subtree is one that contains all the values between two keys within the tree. Using this table, the algorithm then, for every complete subtree, adds the overall probability of the subtree to the minimum composition of the two subtrees that would exist for every possible root. In this process the root that is picked is also recorded. At the end the value for the whole tree is the expected search depth and the saved roots can be used to form the tree.

This method has a construction time of $O(n^3)$, but Knuth includes a speedup that results in the reported time of $O(n^2)$. This speedup comes when choosing the root for each subtree. Instead of looking at every possible root, it only looks at the roots between or equal to the roots of the subtree missing the smallest and largest nodes. This generates a telescoping effect which allows this portion of the algorithm to then average to constant time as it results in O(n) time divided by *n* subtrees, giving O(1) on average.

This will result in the tree with the best possible expected search depth, but because this is not necessarily correlated with a balanced tree, it does not guarantee a tree of logarithmic height. On average, however, it guarantees at most logarithmic lookup as it's expected depth is less than or equal to that of any other tree formation. As this is a static method, insertion beyond the original construction and deletion are not allowed.

The tree to the right has been generated with the following information. The key and leaf probabilities visually represent the values they represent, with the keys probabilities directly below the key they represent and the leaves in the corresponding gap between values.

Key Values: [2, 4, 6, 8, 10]
Key probabilities: [.17, .15, .13, .06, .08]
Leaf probabilities: [.02, .12, .02, .04, .15, .06]



Calculated Expected Search Depth: 1.33

Resources for Further Research:
[Optimum Binary Search Trees by Knuth]

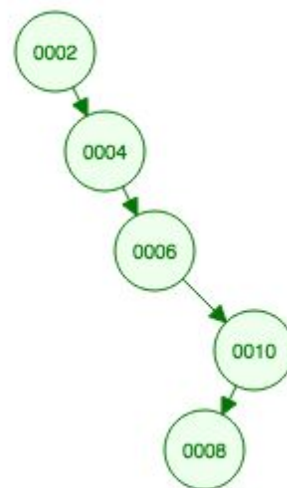
Knuth's "Rule 1" for Approximately Optimal Trees

Knuth's paper "Optimal Binary Search Trees" also mentions two rules of thumb that may lead to approximately optimal search trees in $o(n^2)$ time. The first rule described is quite simple: for every given probability range, always select the root to be the key with the highest probability,

then continue on the left and right sub-ranges to create left and right children. The intuition behind this heuristic makes sense: this approach will naturally force high-probability keys to appear close to the root of the tree, possibly decreasing the expected search depth for the tree. However, it fails to consider how all other keys may be partitioned, possibly leading to a non-optimal tree. In Kurt Mehlhorn's paper "Nearly Optimal Binary Search Trees" (1975), the author provides a short proof to demonstrate why this rule may lead to bad search trees.

However, there was no mention of the expected *practical* performance of this algorithm, so we sought to implement it alongside several other algorithms to see how it would compare. The algorithm was quite straightforward to code: for every sub-range of probabilities, the highest probability was tracked, a node was created for the key associated with that probability, and its left- and right-hand children were assigned to the result of iterations on the left and right subranges. In the event that multiple keys shared the same maximum probability, the median index was picked as the root. This helped to provide more balance in the event that many keys in a given range shared the same probability.

We predicted that the runtime of this algorithm would be O(n*log(n)) in the average case, as it appears to have similar behavior to quicksort with randomized pivot selection. In each case, an initial array of size n is searched/partitioned in O(n) time, and then the same operations are recursively performed on subdivisions of the array until a base case is reached. In each case, the split may be good for efficient performance (a nearly even split) or bad (a split near one of the edges). A randomized analysis of quicksort demonstrates that the algorithm is expected to run in O(n*log(n)) time. Thus, we expected that if probability values were roughly randomly distributed throughout the sorted list of keys, the runtime for Knuth's "Rule 1" algorithm would also be O(n*log(n)). Because the algorithm must operate on a list of probabilities sorted by key, there is no randomization or shuffling we can do to improve our confidence that the algorithm will run efficiently.



The tree to the right has been generated with the following information. The key and leaf probabilities visually represent the values they represent, with the keys probabilities directly below the key they represent and the leaves in the corresponding gap between values.
Key Values: [2, 4, 6, 8, 10]
Key probabilities: [.17, .15, .13, .06, .08]
Leaf probabilities: [.02, .12, .02, .04, .15, .06]

Calculated Expected Search Depth: 1.99

Resources for Further Research:
[Optimum Binary Search Trees by Knuth]

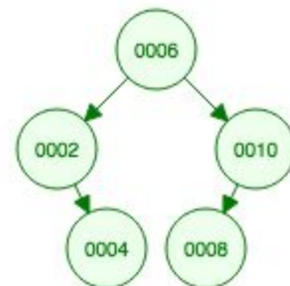
Knuth's "Rule 2" (Mehlhorn's Approximation Algorithm)

Knuth's second suggested rule was more interesting: for every given probability range, always select the root that equalizes the sums of weights to its left and right as much as possible. This also has clear intuition behind it. If each node in the tree can roughly split the weight in half, no single key will ever be too far out of reach. Even if the *shape* of the tree is not balanced, its *weight distribution* will be.

Mehlhorn's paper outlines an algorithm for building a tree this way. For every sub-range of probabilities, a pointer is initialized at each end of the array, each pointing to a candidate root. The sum of weights to the left and right of each pointer is tracked, given that the total sum of every probability range is always known (the initial call of the algorithm will have a probability sum of 1, and sub-range sums can be propagated forward after that). As the pointers move towards the center of the array, weight differences are updated in O(1) time, and as soon as one pointer detects that its weight balance is becoming *more* uneven instead of *less* uneven, the root leading to the best weight split has been found. A node is created, and the algorithm continues to build left and right children.

The use of two pointers aids the runtime analysis of the algorithm, which Mehlhorn demonstrates to be O(n*log(n)). He also provides an upper bound on the expected tree height for trees created with this algorithm, which he cites as 2 + 1.44 * H, where H is the entropy of the distribution. This is in comparison to the lower bound on expected height of the optimal tree, which is 0.63 * H. Therefore, depending on the size of our distribution, we expected that Mehlhorn's approximation algorithm would not build trees with expected heights exceeding 2-3x the optimal height. Given that Mehlhorn's algorithm boasted an O(n*log(n)) runtime, we anticipated this algorithm being very compelling for practical usage.

The tree to the right has been generated with the following information. The key and leaf probabilities visually represent the values they represent, with the keys probabilities directly below the key they represent and the leaves in the corresponding gap between values.



Key Values: [2, 4, 6, 8, 10]
Key probabilities: [.17, .15, .13, .06, .08]
Leaf probabilities: [.02, .12, .02, .04, .15, .06]

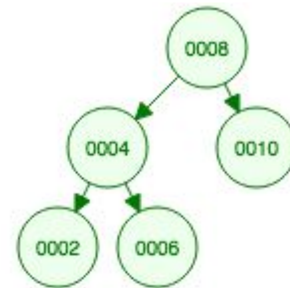
Calculated Expected Search Depth: 1.41

Resources for Further Research:
[Nearly Optimal Binary Search Trees by Mehlhorn]

AVL Tree

AVL trees are balanced binary search trees, invented by Adelson-Velsky and Landis in 1962. Instead of balancing the tree by looking at probability weights as with Knuth and Mehlhorn, an AVL tree simply balances height. We were curious to see how a height-focused tree would fare when measured alongside the above weight-focused algorithms. An AVL tree is simply one where for all nodes in the tree the height of its children differs by at most 1. This property is maintained through rotations back up the tree when a new node is inserted. The tree will only ever have to perform rotations at one place for any new insertion. Through this property, we can determine that the depth of an AVL tree will be at most 1.4*log(n). All operations, lookup, insertion, and deletion, are O(log(n)) in an AVL tree. When using the AVL tree our values were randomly shuffled before being inserted.

The tree to the right has been generated to use on the following dataset. The key and leaf probabilities visually represent the values they represent, with the keys probabilities directly below the key they represent and the leaves in the corresponding gap between values.
Key Values: [2, 4, 6, 8, 10]
Key probabilities: [.17, .15, .13, .06, .08]
Leaf probabilities: [.02, .12, .02, .04, .15, .06]



The key values were inserted in the following random order as the algorithm did not use probabilities:
[8, 10, 6, 4, 2]

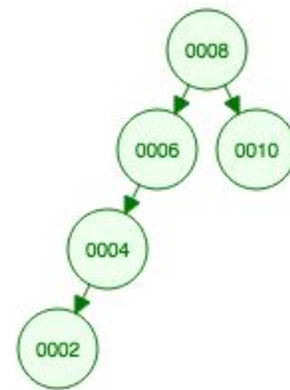
Calculated Expected Search Depth: 1.44

Resources for Further Research:
[AVL Tree and Its Operations by Nair, Oberoi, and Sharma]
[Visualization]

Binary Search Tree (Baseline)

The expected depth of a randomly built basic binary search tree is O(log(n)) (Cormen et al. section 12.4). Furthermore, we saw in lecture that the expected max depth upper bound has a leading constant of ~3, which is not much larger than the upper bound leading constant of 1.4 for AVL trees. In practice, we found that the depth constant for randomly built trees was around 2.5, certainly falling within the proven upper bound. With this information in mind, we sought to include a simple baseline benchmark to compare against more the more intentional optimal binary search tree algorithms, as well as the AVL tree. Here, we insert all *n* keys into the tree in random order, taking O(n*log(n)) time.

The tree to the right has been generated to use on the following dataset. The key and leaf probabilities visually represent the values they represent, with the keys probabilities directly below the key they represent and the leaves in the corresponding gap between values.
Key Values: [2, 4, 6, 8, 10]
Key probabilities: [.17, .15, .13, .06, .08]
Leaf probabilities: [.02, .12, .02, .04, .15, .06]



The key values were inserted in the following random order as the algorithm did not use probabilities:
[8, 10, 6, 4, 2]

Calculated Expected Search Depth: 1.73

Results and Discussion

Our tests used portions of the novel *Moby Dick* to form a word frequency distribution.

Small, Medium and Large Frequency Distributions

Small (2,011 keys)

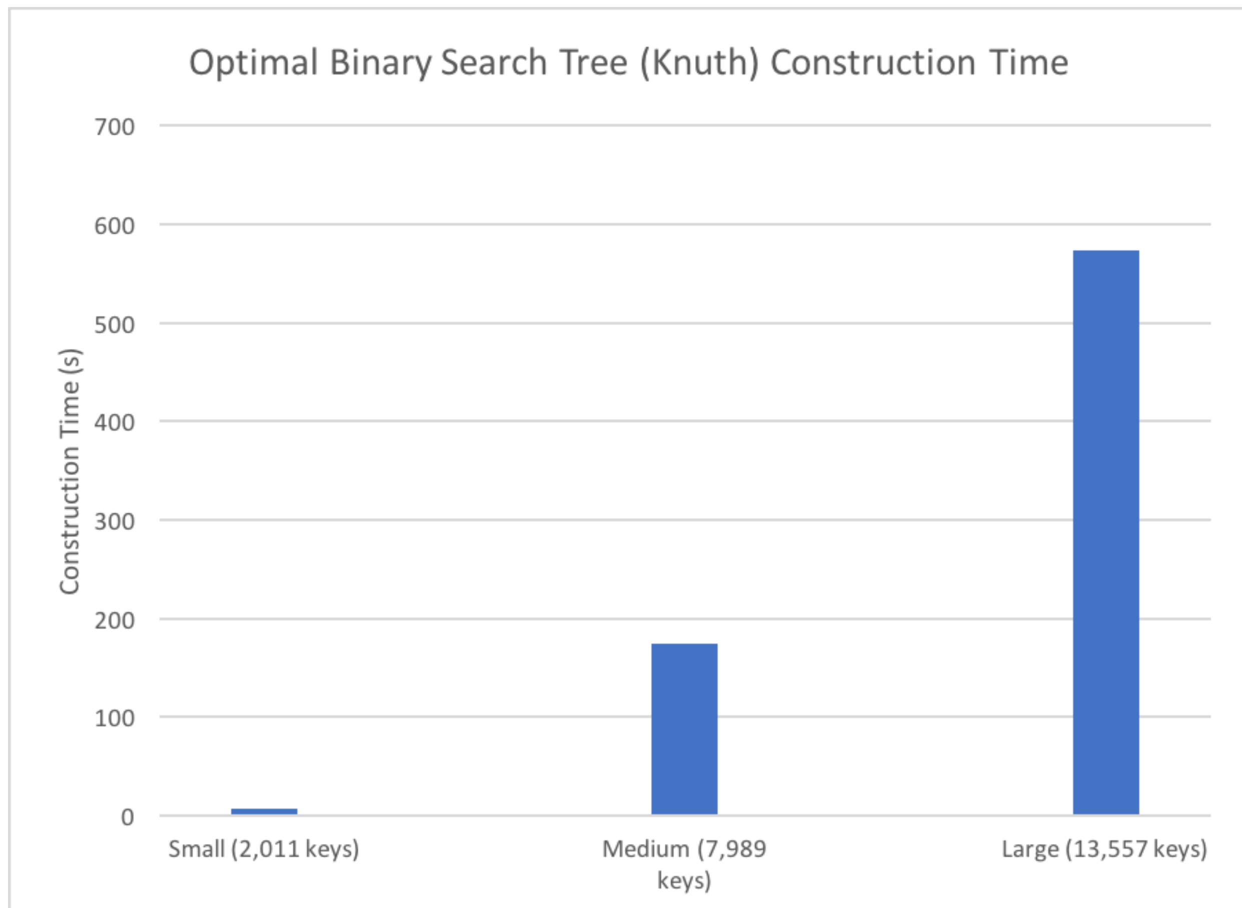
	Knuth O(n^2)	**Mehlhorn**	**Rule 1**	**AVL**	**BST**
Build Time (s)	6.949	0.030	0.023	0.060	0.013
Expected Search Depth	7.43	7.94	9.29	9.73	13.81

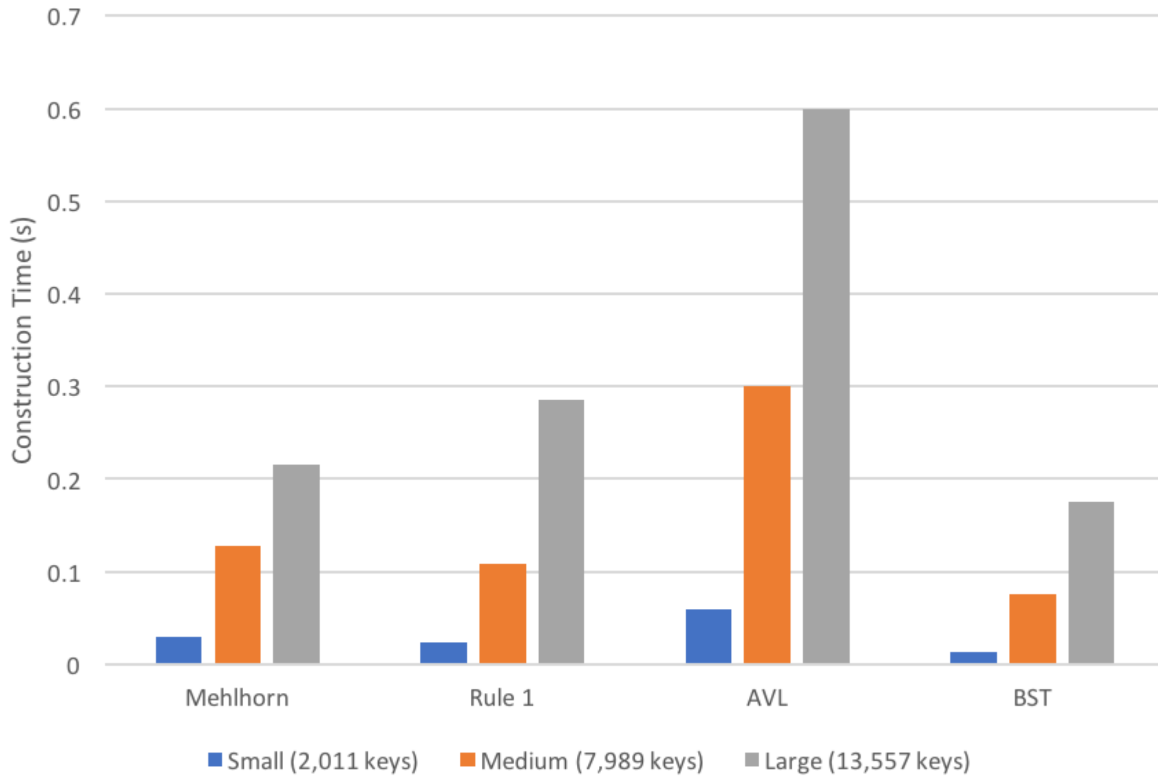
Medium (7,989 keys)

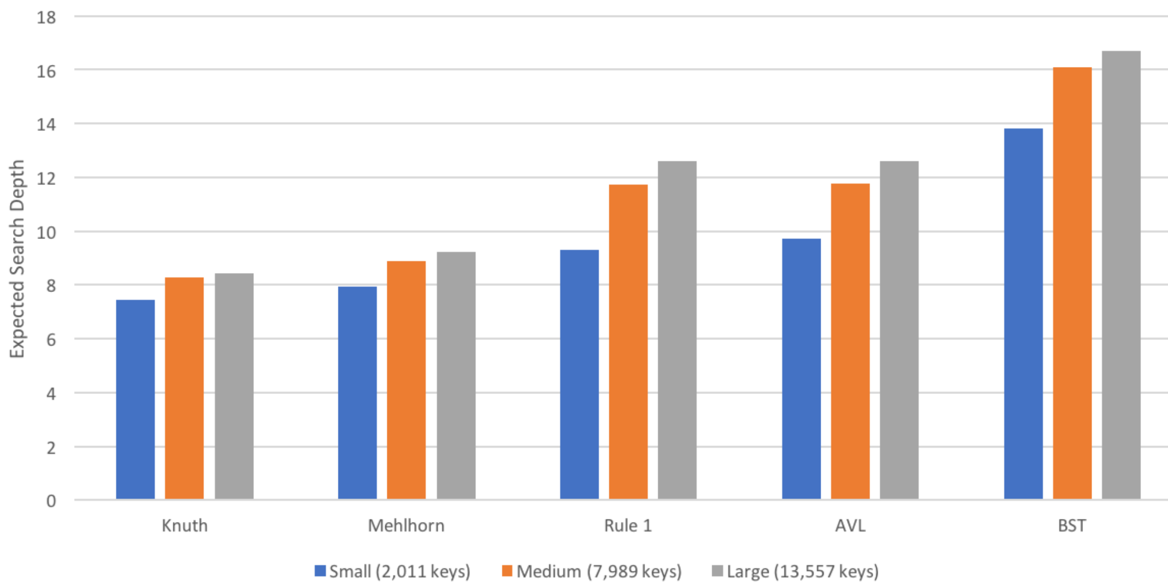
	Knuth $O(n^2)$	**Mehlhorn**	**Rule 1**	**AVL**	**BST**
Build Time (s)	173.649	0.128	0.109	0.300	0.075
Expected Search Depth	8.27	8.88	11.73	11.77	16.10

Large (13,557 keys)

	Knuth $O(n^2)$	**Mehlhorn**	**Rule 1**	**AVL**	**BST**
Build Time (s)	573.640	0.216	0.285	0.599	0.176
Expected Search Depth	8.43	9.22	12.62	12.60	16.70




Approximate Binary Search Tree Construction Times
Construction Time (s)
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Mehlhorn
Rule 1
AVL
BST
Small (2,011 keys)
Medium (7,989 keys)
Large (13,557 keys)


Expected Search Depths of (Approximate) Optimal Binary Search Trees
Expected Search Depth
18
16
14
12
10
8
6
4
2
0
Knuth
Mehlhorn
Rule 1
AVL
BST
Small (2,011 keys)
Medium (7,989 keys)
Large (13,557 keys)

Several patterns are apparent. First, Knuth's optimal algorithm consistently makes the best trees (having the smallest expected depth) and consistently takes the most time to build. Its $O(n^2)$ runtime separates it from its competitors, which all run in O(n*log(n)) time. Mehlhorn's approximation algorithm performs most impressively, coming extremely close to the best possible expected depth in all cases and taking orders of magnitude less time to run. In the large distribution benchmark in particular, Mehlhorn's algorithm runs over 2,700x faster than the optimal algorithm does. Memory usage of these two algorithms is also quite different, with Knuth's algorithm having $O(n^2)$ space complexity and Mehlhorn's having O(n) space complexity. We did a single memory usage benchmark on the large dataset. Knuth's algorithm used 3.47GB of memory, while Mehlhorn's used only 34.16MB of memory, roughly 100x less.

Our prediction that Knuth's "Rule 1" heuristic would run in O(n*log(n)) time was proved correct, based on how the runtime scales with its neighboring algorithms. Interestingly, "Rule 1" does not result in exceedingly bad trees. In fact, Rule 1 tends to produce trees that are at most 50% worse than the optimal arrangement. In Mehlhorn's original paper, he provides a short example of why the rule will not lead to nearly optimal trees before dismissing it and focusing on the better heuristic. As it turns out, the intuition of putting popular nodes near the root is not entirely illogical.

Finally, the performance of the two traditional binary search trees is also interesting to analyze. The unbalanced binary search tree builds faster than any of the other algorithms, evidently due to minimal overhead in the algorithm (*n* nodes are inserted into a growing tree with no additional bookkeeping necessary). The expected depths of the unbalanced tree tend to be roughly twice as deep as the optimal tree. We predicted the depth would still be logarithmic, but it is interesting to see that there is only a constant of 2 separating the expected search performance for the most optimal tree and a randomly constructed tree.

Unlike the unbalanced binary search tree, the AVL tree *guarantees* logarithmic depth with an expected depth of 1.4*log(n). Interestingly, the AVL tree's expected depth tends to fall roughly halfway between that of the optimal tree and the unbalanced tree. Its overall performance is quite similar to Rule 1.

When benchmarking the average search times for each algorithm, we developed a rule of thumb for calculating real CPU search time based on depth: *half expected depth, in microseconds*. For example, A tree with an expected depth of 12.5 would tend to have an average search time close to 6.25 microseconds, with some variability. By using this estimation, we can calculate when, if at all, it makes sense to invest the time to build a truly optimal tree using Knuth's algorithm.

	Mehlhorn	**Rule 1**	**AVL**	**BST**
Searches before optimal tree becomes ideal (13,557 keys)	~1.45 billion	~274 million	~275 million	~139 million

A higher score by this metric indicates a more practical algorithm. These calculations demonstrate that it takes a high number of searches before the cost of the optimal algorithm begins to pay off. Mehlhorn's algorithm performs the best.

Uneven Frequency Distribution Between Keys and Leaves

To see the effects of different kinds of patterns of frequency distribution we generated a set where the probability of searching for something in the tree was significantly higher than that of searching for something that was not in the tree and vice versa. To generate the set with high probability keys we simply took our dataset and put every value into the tree and set every alpha, or leaf, probability to zero as every value was present. To generate a set with high leaf probabilities, or high probability of searching for values not in the tree, we simply adjusted the way we accumulated our alpha and beta values. Instead of adding one value to the tree for every two values in our dataset, we added one value to the tree for every three values in our dataset. This created search gaps, or alpha values, with about twice the probability than each of the keys on average.

High Key (9,547 keys)

	Knuth $O(n^2)$	**Mehlhorn**	**Rule 1**	**AVL**	**BST**
Build Time	260.766	0.162	0.130	0.408	0.111
Expected Search Depth	7.33	8.37	9.72	11.45	15.47

High Leaf (5,326 keys)

	Knuth $O(n^2)$	Mehlhorn	Rule 1	AVL	BST
Build Time	65.773	0.095	0.081	0.224	0.065
Expected Search Depth	8.07	8.65	10.99	11.45	14.648

As expected, Knuth's optimal algorithm presents the best expected depth and the slowest build time for both datasets while the unbalanced implementation presented the worst expected depth and fastest build times. In the dataset with high key probability, Mehlhorn's algorithm was further off the optimal tree than it typically is. The reason for this is still unclear. However it still had the next best expected depth, by a healthy margin. The times for the four O(n*log(n)) trees were still relatively consistent among themselves, with the exception of the AVL tree which was noticeably slower.

In the dataset with high leaf probability, Mehlhorn's approximation was much closer to its usual difference from Knuth. However, Knuth's "Rule 1" heuristic had a much larger expected depth than the optimized algorithms, which fit our predictions. As this heuristic only looks at key probabilities, it is less equipped to deal with the case where searches frequently terminate at leaves. We would hypothesize that if we purposefully introduced great variation in the weights of leaves while keeping key weights low, or simply induce even greater difference between the leaf and key weights, it would perform even worse.

Uniform Frequency Distribution

Uniform (3,000 keys)

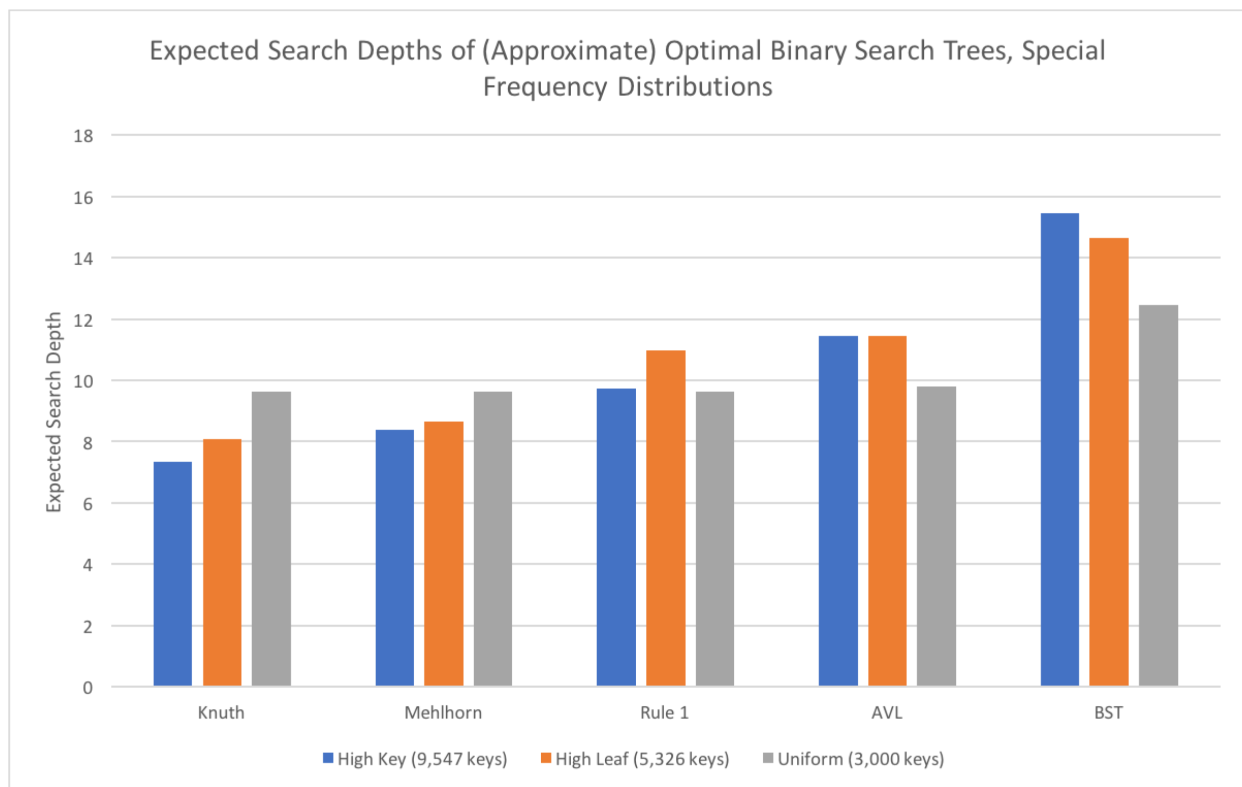
	Knuth $O(n^2)$	Mehlhorn	Rule 1	AVL	BST
Build Time	13.406	0.053	0.039	0.090	0.020
Expected Search Depth	9.639	9.639	9.64	9.8	12.45

With a fully uniform set of probabilities, all keys with equal probability and leaves set to 0, the optimal tree is a fully balanced tree. A complete tree with 2^n nodes will have a max depth of $n - 1$, thus our tree of size 3,000 should have a max depth of $log(3000) - 1 = \sim 10.55$. Our results in the best-performing cases give an average height of roughly one less than this max height, which

is exactly what one would expect for a fully balanced tree, given that half of the tree's nodes exist on the bottom level.

Given this, all the algorithms produce relatively similar results, very close to the optimal, except for the unbalanced tree implementation, whose depth is still fully based on the random input order. As we would expect, in this case the AVL tree performs the closest it ever does to optimal, as it tries to optimize the tree simply through balance. Given the numbers, Mehlhorn's approximation implementation, also does unsurprisingly well, only .001 off in expected depth from the optimal. Knuth's "Rule 1" heuristic, however, originally did extremely poorly. This was due to the fact that it simply takes max probability as root, and in our original implementation it set the first max node it encountered, which resulted in a linked list implementation in this set. This caused us to alter the design and have it take the median max node. This then resulting in Rule 1 behaving identically in Mehlhorn in this case and giving identical expected depths.

The construction times of these algorithms followed the same patterns set the previous tests, and what was intuitive knowing the algorithms run time and propensity for overhead. This makes an improved "Rule 1" heuristic the best for a fully even tree, given its slightly smaller build time, but Mehlhorn's algorithm would probably given a case where values make be close in probability but not completely equal.



Conclusion

The optimality and performance characteristics of Knuth's dynamic programming algorithm were known before beginning this project. What surprised us was that all of the approximation algorithms perform better than anticipated, leading us to question when Knuth's algorithm is actually worth its investment of time and space resources. Our calculations demonstrated that for a frequency distribution with only ~13,000 keys, one would need to anticipate searching the binary search tree ~1.45 billion times before Knuth's algorithm began to beat Mehlhorn's algorithm in terms of overall investment of time. This equates to searching the structure 46 times per second for an entire year. Perhaps large enterprise-grade systems would hit this target. However, because these are statically optimal trees, any change to the frequency distribution would require a new tree to be built to retain optimality. Depending on the application, this may or may not be viable.

Especially when considering the $O(n^2)$ memory usage of Knuth's algorithm, it becomes clear that Mehlhorn's algorithm is the most practical algorithm to create approximately optimal binary search trees. Mehlhorn's algorithm is easy to implement, and it provides great search performance given its build time. For computing contexts where structures like hash tables are not viable, Mehlhorn's algorithm leads to a very efficient structure for storing large sets of keys with known search probabilities.

References

Cormen, Thomas H, et al. Introduction to Algorithms. 2nd ed. ed., Cambridge, Mass., MIT Press, 2001.

Knuth, D. E. "Optimum Binary Search Trees." Acta Informatica 1 (1971): 14-25. Web.

Mehlhorn, Kurt. "Nearly Optimal Binary Search Trees." Acta Informatica 5.4 (1975): 287-295. Web.

Appendix

Code from this project is available on GitHub at https://github.com/alexvking/comp150_bsts.