Key Independent Optimality

John Iacono[*]

Abstract

A new form of optimality for comparison based static dictionaries is introduced. This type of optimality, key-independent optimality, is motivated by applications that assign key values randomly. It is shown that any data structure that is key-independently optimal is expected to execute any access sequence where the key values are assigned arbitrarily to unordered data as fast as any offline binary search tree algorithm, within a multiplicative constant. Asymptotically tight upper and lower bounds are presented for key-independent optimality. Splay trees are shown to be key-independently optimal.

1 Introduction

The problem of executing a sequence of operations on a binary search tree is one of the oldest and most fundamental problems in computer science. This paper examines the simplest form of this problem, where there is a sequence A of m accesses (successful searches) to a tree containing a static set of n items. For simplicity, neither insertions nor deletions are considered, and it is assumed the access sequence is not short (at least $n \log n$). The binary search trees are of the standard type, and allow unit cost edge traversals and rotations.

There has been a long history both of structures for this problem, and of ways of bounding the runtime of a structure as a function of the access sequence. A brief history of runtime characterizations will be presented, along with a description of associated structures. This history will motivate the main result.

First, a few notes on notation and terminology. As previously stated, we are only interested in standard binary search trees that store n items and execute an access sequence A containing m items $a_1, a_2, \ldots a_m$. The access sequence A will often be an implicit parameter of the functions that we will define. When we need to make this explicit, a subscript is used. Since insertions and deletions are not under consideration in this work, we assume without loss of generality, that the data stored is simply the integers from 1 to n. By $\log x$ we mean $\max(1, \log_2 x)$. A standard binary search tree structure is a binary search tree storing n items subject to the following conditions: A pointer to one node in the structure is maintained. At unit cost the pointer may be moved to the parent, left child, or right child of the current node. Also at unit cost, a single left or right rotation may be performed. We say that a binary search structure executes an access sequence A if, after executing all of the pointer movements and rotations required by the structure, A is a subsequence of the sequence formed by data in all of the nodes touched by the pointer.

Property 1 *A binary search tree structure has the $O(\log n)$ runtime property if it executes A in time $O(m \log n)$.*

The $O(\log n)$ runtime property is expected to be optimal if the accesses in A are independently drawn at random from a uniform distribution. It is also known from [11] that there exist some access sequences which no binary search structure can execute in time $o(m \log n)$. All data structures discussed in this paper have the $O(\log n)$ runtime property. The simplest such structure would be a static balanced binary search tree.

Property 2 *A data structure has the static finger property at finger g if the time to execute A is $O(\sum_{i=1}^{m} \log |g - a_i|)$.*

Recall that an item's value is also its rank, and thus $|g - a_i|$ is the rank difference between the finger g and a_i. There exist specialized structures [5] that given g, have the static finger property at g. Splay trees [10] have the static finger property for any given g, without being explicitly initialized with g.

Property 3 *Let $f(x)$ be the number of accesses to x in A. A data structure has the static optimality property if it executes A in time $O(\sum_{i=1}^{n} f(i) \log \frac{m}{f(i)})$.*

Optimal search trees [9] have the static optimality property, however, their construction requires that the access frequencies $\frac{m}{f(i)}$ be known. Splay trees have the static optimality property [10] and do not require any distribution-specific parameterization. Search trees with the static optimality property are expected to be optimal if A consists of a sequence of independent accesses drawn from a fixed (static) distribution. "Optimal" search trees and other structures with the static optimality property are not necessarily optimal for deterministic access sequences, or randomly generated access sequences where one access is dependent on the next. It has been shown that any data structure with the static optimality property also has the static finger property, for any choice of g [8].

[*]Department of Computer and Informaion Science. 5 MetroTech Center, Brookyn NY 11201, USA. http://john.poly.edu. Research partially supported by NSF grants CCF-0430849 and CCF-9732689.

Property 4 *Let $l(i,x)$ be j if a_j is the index of the last access to x in the subsequence $a_1, a_2, \dots a_{i-1}$. If x is not accessed in $a_1, a_2, \dots a_{i-1}$, we define $l(i,x)$ to be 1. Let $w(i,x)$ be the number of distinct items accessed in the subsequence $a_{l(i,x)+1}, a_{l(i,x)+2} \dots a_i$. We use $w(a_i)$ to denote $w(i,a_i)$. We say a data structure has the working set property if it can execute A in time $O(\sum_{i=1}^m \log w(a_i))$*

Splay trees have the working set property [10]. It has been shown that the working set property implies the static optimality property [6], and thus also the static finger property. A data structure that is not a tree was introduced in [7] that has worst-case $O(\log w(a_i))$ runtime per access. As it was also shown that similar worst-case results are impossible to obtain for the static finger property and the static optimality property [8], the working-set property can be viewed as the most natural of the three previous properties.

Property 5 *A data structure with the dynamic finger property executes A in time $O(\sum_{i=2}^m \log |a_{i-1} - a_i|)$*

Level-linked trees of Brown and Tarjan [1] were shown to have the dynamic finger property. However, they do not meet the definition of a standard binary search tree structure, because there are additional pointers that the trees must be augmented with. However, it was shown that splay trees have the dynamic finger property [3, 2]. The dynamic finger property implies the static finger property.

Neither the dynamic finger property nor the working set property imply each other, and these are known to be the two best analyses of splay trees, and search trees in general. We now leave the proven properties of binary search trees and continue on to conjectured properties.

Property 6 *A data structure with the unified property executes an access sequence A in time $O(\sum_{i=1}^m \min_j \log(w(i,j) + |a_i - j|))$.*

The unified property [7] is an attempt to unify the dynamic finger property and the working set property. It implies the dynamic finger property when $j = a_{i-1}$ and the working set property when $j = a_i$. Splay trees are conjectured to have the unified property, but so far the only structure with the unified property, the unified structure [7], is not a binary search tree. It has been known that the unified property is not a tight bound on the runtime of splay trees [4], as access sequences exist that splay trees execute faster than the unified property would indicate.

Property 7 *We define $OPT(A)$ to be the fastest any binary search tree structure can execute access sequence A. We allow the binary search structure to choose the initial tree. A data structure has the dynamic optimality property if it executes A in time $O(OPT(A))$*

The dynamic optimality conjecture [10] simply states that splay trees have the dynamic optimality property. That is, it is conjectured that no binary search tree, even offline and with infinite preprocessing, can execute *any* A by more than a multiplicative constant faster than splay trees. Note that the choice of an initial tree can not asymptotically change $O(OPT(A))$ for $m = \Omega(n)$.

So, in summary, the best binary search tree structure appears to be the splay tree. The splay tree's runtime has not yet been tightly analyzed, and there are currently two "best" bounds on how fast a splay tree executes A (the working set property and the dynamic finger property), neither of which implies the other. Even the conjectured explicit bound of the unified property is not equal to the dynamic optimality conjecture. So far the only proven forms of optimality for search trees are quite old, and require independent accesses from static distributions to generate entropy-based lower bounds [9]. These lower bounds apply not specifically to trees but to any comparison based structure. On the other hand, dynamic optimality, while powerful remains but a conjecture. The purpose of this paper is to propose and *prove* a new form of optimality for binary search trees.

We define a random bijection b from n to n by choosing a random permutation of the integers $1, 2, \dots n$, and defining $b(i)$ to be the ith element of the permutation. The permutation is chosen at random with all $n!$ permutation being equally likely.

Property 8 *We define $KIOPT(A) = E[OPT(b(A))]$, where b is a random bijection from n to n and $b(A) = b(a_1), b(a_2), \dots b(a_m)$. A data structure is key independently optimal if it executes A in time $O(KIOPT(A))$.*

In this definition we introduce a new form of optimality for binary search trees, *key-independent optimality*. Our definition is motivated by a particular type of application, where there is no correlation between the rank distance between two key values and their relative likelihood of a proximate access. This could occur for several reasons.

One reason could be because the key values were assigned randomly to otherwise unordered data. We will prove (starting in the next paragraph) that if this is the case, no binary search tree data structure is expected to execute any *fixed* access sequence on unordered data faster than $KIOPT(A)$ after the key values have been assigned. It is important to note that other than picking the total ordering of the key values, the access sequence is fixed. Unlike previous optimality results, there is no requirement that the access sequence be created by independent drawings from a distribution. One can think of many such applications where key values are generated arbitrarily.

We now formally prove the claim of the previous paragraph. Let U be a set of n unordered elements. Let $A' = a'_1, a'_2, \ldots a'_m$ be a fixed sequence of m elements of U, representing an access sequence. Let b' be a random bijection that maps the elements U to the totally ordered set $X = 1, 2, \ldots n$. Let $b'(A') = b'(a'_1), b'(a'_2), \ldots b'(a'_m)$. Thus $b'(A')$ represents the accesses to U after a random ordering has been imposed on the elements, and $OPT(b'(A'))$ is the fastest that such a sequence can be executed on a binary search tree.

Lemma 1 $E[OPT(b'(A'))] = KIOPT(b'(A'))$

By the definition of $KIOPT$, $KIOPT(b'(A')) = E[OPT(b(b'(A)))]$. Both b and b' were defined to be random bijections. The random bijections were defined by randomly chosen permutations. We observe that randomly permuting the integers $1, 2, \ldots n$ twice, by choosing the permutation uniformly, has the same effect as randomly permuting once. Thus composing the two random bijections yields one random bijection and completes the proof of the lemma.

Now having shown that key-independent optimality is a useful notion, we now present the positive result that we are able to tightly bound $KIOPT(A)$.

Theorem 1 *The working set property and key-independent optimality property are asymptotically the same.* $KIOPT(A) = \Theta(\sum_{i=1}^{m} \log w(a_i))$

The proof of this theorem is the subject of Section 2.

Corollary 1 *Splay trees are key-independently optimal*

This is immediate, since splay trees have the working set property. Thus, if the key values in a given application are assigned randomly, then no binary search tree can be expected to execute A more than a constant multiplicative factor faster than splay trees, even with complete foreknowledge of the access sequence, and infinite preprocessing time.

2 Proof of main result

The proof of Theorem 1 is split into upper and lower bounds.

Lemma 2 $KIOPT(A) = O(\sum_{i=1}^{m} \log w(a_i))$

Proof: As previously stated, splay trees were shown in Sleator and Tarjan [10] to have the working set property. Thus splay trees execute A in time $O(\sum_{i=1}^{m} \log w(a_i))$. Since the definition of $w(x)$ does not take into account the key values at all, $\sum_{i=1}^{m} \log w(a_i) = \sum_{i=1}^{m} \log w(b(a_i))$ for every bijection b, including, of course, a randomly chosen one. Thus since splay trees are a binary search tree structure that will execute $b(A)$ in time $O(\sum_{i=1}^{m} \log w(a_i))$, this is an upper bound on $KIOPT(A)$. $\square$

The lower bound is based upon the second lower bound of Wilber found in [11]. The result of Wilber and two technical lemmas are presented before the proof of the lower bound.

Define the working set of a sequence at i, $W(A, i)$ to be the sequence $a_{i-1}, a_{i-2}, \ldots a_j$ where a_j is the previous access in A to the item accessed in a_i. If the item accessed in a_i has not been accessed before, $W(A, i)$ is the sequence $a_{i-1}, a_{i-2}, \ldots a_1, a_i$. Define the induced tree of a sequence $X = x_1, x_2, \ldots x_i$, $T(X)$ to be the binary search tree formed as follows: Insert $x_1, x_2, \ldots x_i$ into an initially empty non-balanced binary search tree. If an item is already in the tree, the insertion does nothing. Let $P(X)$ be the search path for the last item inserted in $T(X)$, expressed as a sequence of L's and R's, representing whether the search path moves left or right at each node, starting with the root. Let $l(X)$ be the the length of $P(X)$. Let $v(X)$ be the number of alternations between L's and R's in the sequence $P(X)$. Wilber's lower bound can be stated as follows:

Theorem 2 $OPT(A) = \Omega(\sum_{i=1}^{m} v(W(A, i)))$.

Let $R(i)$ be a random permutation of $1, 2, \ldots i$. Recall that b is a random bijection.

Lemma 3 $E[v(W(b(A), i))] = E[v(R(w_A(a_i)))]$

Proof:

We will now define a method for creating a compressed representation of $X = x_1, x_2, \ldots x_i$ by removing items from X to form $C(X)$ according to the following rule: if for some $j < k < i$, if $a_j = a_k$, remove a_k. We refer to this new sequence as the compressed sequence of X, and the length of $C(X)$ as $c(X)$. Since we only remove those items that were not used to construct $T(X)$ because they were duplicates, $T(X) = T(C(X))$ and therefore $v(X) = v(C(X))$. Using this fact, $E[v(W(b(A), i))] = E[v(C(W(b(A), i)))]$.

Since the constructions of W and C only use equality tests, it can be observed that $C(W(b(A), i)) = b(C(W(A, i)))$. Thus, $E[v(W(b(A), i))] = E[v(C(W(b(A), i)))] = E[v(b(C(W(A, i))]$.

Since a compressed sequence $C(X)$ is a set of distinct values, then a compressed sequence of $b(C(X))$ is a sequence of distinct values passed through a random bijection, which is just a random permutation of length $c(X)$. Since we are in the comparison model, we may replace $b(C(X))$ with $R(c(X))$, a random permutation of the first $1, 2, \ldots c(X)$. Thus, $E[v(W(b(A), i))] = E[v(b(C(W(A, i))] = E[v(R(c(W(A, i))))]$.

Next we observe that $C(W(A, i))$ contains exactly the distinct items accessed since the last access to a_i plus a_i itself. From the definition of w note that $c(W(A, i)) = w_A(a_i)$. Thus, $E[v(W(b(A), i))] = E[v(R(c(W(A, i))))] = E[v(R(w_A(a_i)))]$.

□

It is well known that $E[l(R(i))] = \Theta(\log i)$. However, we are concerned not with the expected length of the path $P(R(i))$, but rather the expected number of alternations in this path, $E[v(R(i))]$. This is certainly $O(\log i)$, but showing that it is $\Omega(\log i)$ requires further argument.

Lemma 4 *If $R(i)$ is a random permutation of i values.* $E[v(R(i))] = \Theta(\log i)$

Proof: Intuitively, if we look at the path of the last insertion in a randomly built binary search tree, at each step we would expect to keep going in the same direction (left or right) half the time, and change direction about half the time. We formally prove this with a symmetry argument.

Given a BST tree T we define the reversal of T, $\rho(T)$, to be a new tree formed by switching the left and right children of every node whose depth is even (consider the root to have depth 0). If a root-to-leaf path for an element in T has length l and m alternations between going to the left or to the right, observe that the root-to-leaf path for the same element in $\rho(T)$ also has length l but has $l - m - 1$ alternations (the -1 is from the fact that an alternation can not occur during the first step of a search). Thus, $v(X) + v(\beta(X)) = l(X) - 1$.

We now define a bijection $\beta_X(j)$ as follows, where $X = x_1, x_2, \ldots x_i$ is a permutation of $1, 2, \ldots i$. Let $\beta_X(j) = k$ if j is the kth node in an inorder traversal of $\rho(T(X))$. We define $\beta(X)$ to be $\beta_X(x_1), \beta_X(x_2), \ldots \beta_X(x_i)$.

Note that $T(X)$ is, by definition, a binary search tree, and $\rho(T(X))$ is not (except where the tree is a single node). However, if we define $\rho'(T(X))$ to be the tree formed by replacing each element i of $\rho(T(X))$ with $b_X(i)$, $\rho'(X)$ is a binary search tree. In fact, $\rho'(X) = T(\beta(X))$. We also note $\rho'(\rho'(T(X))) = T(X)$, and thus $\beta(\beta(X)) = X$.

Let $S(i)$ be the set of all permutations of $1, 2, \ldots i$. Let $S_L(i)$ be the subset of all $X \in S(i)$ where $P(X)$ begins with an L, and define $S_R(i)$ in the same manner. The symmetric bijection β imposes a perfect matching on all of the permutations of P where the sets P_L and P_R each contain exactly one permutation from each matching. Note that for any $X \in P_L$, $\beta(X) \in P_R$ and vice-versa.

All of these observations are now combined, starting with the definition of $E[v(R(i))]$.

$E[v(R(i))] = \sum_{X \in S(i)} \frac{v(X)}{n!}$

$E[v(R(i))] = \sum_{X \in S_L(i)} \frac{v(X)+v(\beta(X))}{n!}$

Since $v(X) + v(\beta(X)) = l(X) - 1$,

$E[v(R(i))] = \sum_{X \in S(i)} \frac{l(X)}{2n!} - \frac{1}{2}$

Using the definition of $E[l(X)]$ gives

$E[v(R(i))] = \frac{E[l(X)]}{2} - \frac{1}{2}$

Since $E[l(X)]$ is $\Theta(\log i)$,

$E[v(R(i))] = \Theta(\log i)$

□

Lemma 5 $KIOPT(A) = \Omega(\sum_{i=1}^{m} \log w(a_i))$

Proof:
Starting from the definition of key-independent optimality:

$KIOPT(A) = E[OPT(b(A))]$

From Wilber's Bound:

$KIOPT(A) = \Omega(\sum_{i=1}^{m} E[v(W(b(A), i))])$

From Lemma 3:

$KIOPT(A) = \Omega(\sum_{i=1}^{m} E[v(R(w_A(a_i))])$

From Lemma 4:

$KIOPT(A) = \Omega(\sum_{i=1}^{m} \log w_A(a_i))$

$\square$

Theorem 1 *The working set property and key-independent optimality are asymptotically the same.* $KIOPT(A) = \Theta(\sum_{i=1}^{m} \log w(a_i))$.

Proof: Immediate from Lemma 2 and Lemma 5.

$\square$

3 Further Work

As Wilber's bound seems quite powerful, it would be of interest to see whether or not it is equivalent to the dynamic optimality property. If one were to try to prove non-equivalence, the simplest way would be to come up with a sequence A that splay trees execute faster than Wilber's lower bound on this sequence.

References

[1] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput.*, 9:594–614, 1980.

[2] R. Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM J. Comp.*, 30(1):44–85, 2000.

[3] R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting log n-block sequences. *SIAM J. Comp.*, 30(1):1–43, 2000.

[4] M. L. Fredman, 2001. Private Communication.

[5] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing*, pages 49–60, 1977.

[6] J. Iacono. New upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory (LNCS 1851)*, pages 32–45, 2000.

[7] J. Iacono. Alternatives to splay trees with $o(\log n)$ worst-case access times. In *Symposium on Discrete Algorithms*, pages 516–522, 2001.

[8] J. Iacono. *Distribution Sensitive Data Structures*. PhD thesis, Rutgers, The State University of New Jersey, Graduate School, New Brunswick, 2001.

[9] D. E. Knuth. Optimum binary search trees. *Acta Inf.*, 1:14–25, 1971.

[10] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *JACM*, 32:652–686, 1985.

[11] R. Wilbur. Lower bounds for accessing binary search trees with rotations. In *Proc. 27th Symp. on Foundations of Computer Science*, pages 61–69, 1986.