Adaptive Binary Search Trees

Jonathan Carlyle Derryberry

CMU-CS-09-180

December 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Daniel Sleator, Chair
Guy Blelloch
Gary Miller
Seth Pettie, U. Michigan

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant number CCR-0122581. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: binary search trees, adaptive algorithms, splay trees, Unified Bound, dynamic optimality, BST model, lower bounds, partial-sums

Abstract

A ubiquitous problem in the field of algorithms and data structures is that of searching for an element from an ordered universe. The simple yet powerful binary search tree (BST) model provides a rich family of solutions to this problem. Although BSTs require $\Omega(\lg n)$ time per operation in the worst case, various *adaptive* BST algorithms are capable of exploiting patterns in the sequence of queries to achieve tighter, input-sensitive, bounds that can be $o(\lg n)$ in many cases. This thesis furthers our understanding of what is achievable in the BST model along two directions.

First, we make progress in improving instance-specific *lower bounds* in the BST model. In particular, we introduce a framework for generating lower bounds on the cost that any BST algorithm must pay to execute a query sequence, and we show that this framework generalizes previous lower bounds. This suggests that the optimal lower bound in the framework is a good candidate for being tight to within a constant factor of the optimal BST algorithm for each input. Further, we show that lower bounds in this framework are also valid lower bounds for instances of the partial-sums problem in a restricted model of computation, which suggests that augmented BSTs may be the most efficient way of maintaining sums over ranges of an array when the entries of the array can be updated throughout time.

Second, we improve the input-sensitive *upper bounds* that are known to be achievable in the BST model by introducing two new BST algorithms, skip-splay and cache-splay. These two algorithms are the first BSTs that are known to have running times that have nontrivial competitiveness to Iacono's Unified Bound, which is a generalization of the dynamic finger and working set bounds. Skip-splay is a simple algorithm that is nearly identical to splaying, and it achieves a running time that is within additive $O(\lg \lg n)$ per operation of the Unified Bound. Cache-splay is a slightly more complicated splay-based algorithm that is the first BST to achieve the Unified Bound to within a constant factor.

Acknowledgments

I would first like to thank my thesis committee. My advisor Danny Sleator was especially helpful in suggesting topics to explore, helping me focus on my best ideas, and supporting my progress as a graduate student in general and on this thesis in particular. Additionally, I would like to thank Gary Miller, Guy Blelloch, and my external committee member Seth Pettie for their suggestions and feedback along the way.

Also, I thank my friends and office mates for giving me advice and encouragement through the years as I navigated my way through the graduate program.

Additionally, I thank Weijia for tolerating my long hours as I worked toward finishing this thesis, and for providing occasional refreshing distractions.

Finally, I thank my family for being so patient as I worked toward finishing. I would like to particularly thank my father, whom I miss dearly. I wish he could have been here to see me finish.

Contents

List of Figures

Chapter 1

Introduction

The search problem is one of the simplest problems in all of theoretical computer science. Abstractly, and informally, the search problem can be stated as follows. Given a set of convex regions that partition some space in addition to a point in that space, return the region that contains that point. Of course, if we only had one query, we could use a naïve brute force algorithm and simply test each region one by one to see whether it contained the point. However, we are usually concerned with serving a *sequence* $\sigma = \sigma_1 \cdots \sigma_m$ of queries where each σ_j represents a query to a point σ_j in the search space. When executing such a sequence of queries, it usually makes sense to organize the input into a *data structure* that helps speed up each query.

To illustrate this, consider the case of one-dimensional search, in which the regions are all individual line segments. For ease of theoretical analysis, we will make the simplifying assumption that each region is a point, or *key*, from the set $S = \{1, \ldots, n\}$. In reality, we would generally want to support keysets that were not consecutive integers, allow queries to points that were in between the keys of S, and support insertion and deletion of keys. Suppose a search algorithm used no data structure so that the elements of S were stored in memory cells that were scattered throughout memory with no organization whatsoever. Then, for each query σ_j, we would have to compare σ_j to each of the n members of S to ensure that we found the queried element. However, if we simply sorted the members of S, then we could rule out half of the remaining members of S with a single comparison during each step of computation.

Even in this one-dimensional case of the search problem, there is a rich and seemingly endless abundance of possibilities for how to create such a data structure for helping an algorithm serve queries quickly. One family of such data structures is *binary search trees* (BSTs), which comprise a set of nodes, each representing a key from the set S. These

nodes are linked together into a rooted binary tree with the keys in symmetric order, and this tree can be modified by the BST algorithm during the sequence of queries it is executing. Each query begins at the root of the tree with a comparison to the root's key, and proceeds to one of the root's children according to the result of the comparison. See Chapter 2 for a formal description of the BST model that we will use in this thesis.

Even with the constraints imposed on BSTs, substantial flexibility remains for the design of a BST algorithm. As mentioned above, not only can a BST algorithm use any valid initial BST, but it can also adjust the structure of this tree during the sequence of queries if doing so seems likely to speed up the responses to future queries. For example, if one particular region of keyspace were accessed several times in a row, the BST algorithm may move the portion of the tree that corresponds to that region closer to the root in order to accelerate future queries to that region.

Any search algorithm that tries to make such an improvement is called an *adaptive* algorithm. If the queries are uniformly at random and the search algorithm is online so that it does not know what the future queries are, then such attempts to "guess future queries" will not speed up the algorithm on average. However, suppose that the queries of the sequence are correlated with each other so that the conditional probability distribution for the current query, depending on all previous queries, is highly skewed. In this case, it is in principle possible for an online search algorithm to serve queries faster than when queries are uniformly at random. Such correlation might be expected to exist in actual sequences of queries if the queries are being generated by another algorithm that is scanning across the data in some regular manner rather than probing completely at random.

To exploit such correlation, we do not need prior knowledge of the distribution from which queries are drawn. As long as we make some prior assumptions for the type of correlation that might exist, we may be able to ensure that we can exploit the correlation if the actual sequence of queries is generated by such a distribution. This motivates bounding the running time for serving a query sequence with a function that depends on the amount of correlation that appears in the sequence of queries. We will call such a bound an *adaptive bound*.

There are many examples of such adaptive bounds as well as data structures that provably meet these bounds. For a query to key x at time j, the *working set bound* is defined to be $\lg w(x, j)$, where $w(x, j)$ is the number of distinct keys queried during the period of time between the most recent query to x and time j, assuming such a query exists. A data structure whose running time is $O(\lg w(\sigma_j, j))$ for each query σ_j exploits the possibility that recently-queried keys may be more likely to be queried than other keys. Alternatively, the *dynamic finger bound* for query σ_j with $j > 1$ is defined to be $\lg(|\sigma_j - \sigma_{j-1}| + 1)$. A data structure whose running time is $O(\lg(|\sigma_j - \sigma_{j-1}| + 1)$ exploits the possibility that

most queries might be near to the previous query. Chapter 4 will discuss these bounds and others in greater detail, and will also discuss previous work toward provably achieving such bounds with various data structures. Chapters 5 and 6 will discuss BST algorithms that achieve a substantially richer bound called the Unified Bound [37, 16] that subsumes both the working set bound and the dynamic finger bound.

The above adaptive bounds are useful in that they clearly specify what the cost bound is for each query sequence, and they each have an intuitive interpretation and quantitative meaning. However, such formulaic bounds are limited in that, by themselves, they say nothing about whether we have reached the limit of what is achievable in a particular model of computation. One could imagine a seemingly endless quest for achieving better – or at least different – adaptive bounds that attempt to capture any type of query correlation we can imagine. No matter how many of these bounds we proved for a particular algorithm, we would never know for sure whether it performed well on *all* inputs for which a speedup was possible, or know if such a universally adaptive algorithm even existed.

To accomplish this goal, we need to use *competitive analysis*, which compares the performance of an algorithm on a specific input against the performance of all other algorithms on that same input. The competitive ratio of an algorithm A is roughly defined to be the maximum ratio, across all inputs, between the running time of A and the fastest algorithm on the same input.

There are multiple ways of proving that an algorithm is competitive. First, we could directly compare the performance of the candidate competitive algorithm to that of an arbitrary competitor and show that the candidate's performance is not much worse, regardless of the input. If we succeed in proving that the candidate performs almost as well as the competitor, then we have proved that the candidate is competitive because both the competitor and the input were arbitrary. This was the strategy first used in showing that the move-to-front list update heuristic was constant-competitive [53].

However, sometimes it is not clear how to show that the candidate performs well compared to an arbitrary algorithm. In such cases, an alternative approach to proving competitiveness is first to prove a lower bound that tightly bounds the minimum cost required by any algorithm to serve a particular sequence of queries, and second to show that the cost of the competitive algorithm on an input is never much more than the lower bound for that input. All of the known algorithms that have nontrivial competitive ratios with the optimal BST algorithm use this approach [21, 61, 32, 42, 10]. Chapter 3 expounds many of the lower bounds that have been proven for the BST model, and discusses their usefulness in proving competitiveness.

Although competitiveness of an algorithm within a particular model of computation can be a strong result, it is important to consider whether we may have sacrificed too

much by limiting ourselves to a particular model of computation. After all, if we sufficiently limit the model of computation, then it may become easy to prove an algorithm to be competitive, but the algorithm may not actually perform well. For example, even though move-to-front is a good algorithm for the list update problem, it is not a good algorithm for the search problem. Despite being dynamically optimal, move-to-front suffers $\Omega(n)$ expected cost for random queries compared to a balanced BST, which requires only $O(\lg n)$ time.

One could argue that the BST model, even though it allows binary search, is too restrictive. Relative to the pointer-based comparison model, the BST model restricts how we can organize our data, and relative to the RAM model, the BST model ignores the possibility of speeding up search by using direct-addressing or word-level parallelism. Even though the BST model is interesting in its own right as a simple and elegant model, it is worthwhile to consider whether we may have thrown away too much flexibility by limiting ourselves to such a restrictive model. Chapter 2 provides some additional discussion of this possibility by describing some of the differences in what is known to be achievable in some of the most popular models of computation.

One way of vetting a model of computation is to prove that various formulaic bounds, such as the working set bound and the dynamic finger bound, are achievable in the model so that any competitive algorithm is also guaranteed to meet these bounds. Before the first BST algorithm with a nontrivial competitive ratio was found, it was already clear that the BST model permitted good performance when a variety of types of nonrandomness were present in the input, and Chapter 4 discusses some of these results. A nice side benefit to proving an algorithm to be competitive when many adaptive bounds have already been proven in that model is that the competitive algorithm provably inherits all of the previously proved adaptive properties. Conversely, proving a formulaic bound for some new algorithm after a competitive result has already been proven demonstrates not only that the new algorithm satisfies the bound, but also that the competitive algorithm satisfies a corresponding bound as well. For example, when the cache-splay BST algorithm of Chapter 6 was shown to satisfy the Unified Bound, the Tango BST algorithm [21, 22] was automatically shown to satisfy the Unified Bound to within a factor of $O(\lg \lg n)$ because Tango is $O(\lg \lg n)$-competitive in the BST model.

An even stronger way to vet a model of computation is to show that the complexity of the problem it is being used to solve is identical to the complexity of the data structure. Such a result demonstrates definitively that the selected model of computation is not too restrictive, and moreover that the search for better algorithms for solving the *problem* can be reduced to the search for better *data structures* in the selected model of computation. In the case of the BST model, Chapter 3 will provide evidence that suggests that BSTs,

due to the fact that they can be augmented to store information about their subtrees, might fully encapsulate the partial-sums problem. Although it is straightforward to show that the partial-sums problem can be solved with any BST, it is not immediately clear that the partial-sums problem cannot be solved asymptotically faster on any instance.

Other than concerns about the performance of a data structure, another concern is its simplicity. Not only is the simplicity of a data structure important for aesthetic reasons, but simple data structures also are more likely to be used in practice because they are easy to implement and often have low constant factors in their running times. For example, although $O(\lg \lg n)$-competitive BST algorithms such as Tango and multi-splay trees have competitive guarantees that splay trees lack, they are both significantly more complicated than splay trees, and are unlikely to be implemented in practice despite their better guarantees of competitiveness. We will see another example of such a tradeoff in Chapter 5 when we introduce the extremely simple skip-splay algorithm, which is nearly identical to splaying and within additive $O(\lg \lg n)$ of being constant-competitive to the Unified Bound. By comparison, the more complicated cache-splay algorithm in Chapter 6 is constant-competitive to the Unified Bound.

This introduction has sketched out some of the key motivating ideas and goals associated with adaptive search in general and with the BST model in particular. To reiterate the ideals behind the work in this thesis, the ultimate goal is to prove competitive bounds for the simplest algorithm in a computational model that is as general and flexible as possible. The work contained in this thesis adds to our understanding of the performance that can be achieved in the BST model when we look beyond simple worst-case analysis to consider instance-specific bounds. Chapter 7 concludes this thesis with a discussion of various directions for future work that would resolve some of the lingering questions from this work.

Chapter 2

The Binary Search Tree Model

This chapter formalizes the definition of the BST model that was sketched in Chapter 1. To begin, we reiterate the formal definition of the one-dimensional search problem as follows. Given as input a set S of keys, which we take for simplicity's sake to be $\{1, \ldots, n\}$, and a sequence of queries $\sigma = \sigma_1 \cdots \sigma_m$, where each $\sigma_j \in S$, return a sequence of pointers to the memory cells that represent each σ_j in the order specified by the input.

We will typically have some information associated with each σ_j that we will want to return. For example, if we were storing a set of words and their associated definitions, the input would be a sequence of words to look up, and the output would be the memory locations of the definitions of the words. Also, it is important to note that there are other operations that one might want to support besides a simple lookup. For example, in addition to successful queries, we might want to support queries to elements that are not in S. As we will show in this Chapter, not all algorithms for solving the above simplified version of the problem have an easy extension when we demand more from the algorithm.

There are a number of models of computation that we may use when designing and analyzing algorithms for the one-dimensional search problem. In this thesis, we will not be presenting any new results for one-dimensional search outside of the BST model, but it is important to understand how results in the BST model compare with results obtained in other computational models. Therefore, Section 2.1 describes some other models of computation, and assumes familiarity with at least an informal definition of the BST model for the purpose of comparison. We briefly summarize how the search problem can be solved in those alternative models, and at what cost. Section 2.2 will cover the BST model more formally and in greater detail than the other models. It will specify in detail what a BST is allowed to do and what it must pay for.

2.1 Alternatives to the BST Model

In this section, we consider two variables for models of computation. First, we consider what operations will be *permitted*. Algorithms that achieve upper bounds using a restricted set of operations are generally more powerful than those that are allowed greater freedom in their instruction sets, while lower bounds in a restricted model are weaker. Second, we specify what operations, or events, we will *charge* for. This second specification may at first seem a little strange, but to simplify our analysis, it can sometimes be useful to consider models of computation that provide some operations for free. This allows us to concentrate on analyzing one particular type of cost that we expect to dominate in practice. With these variables in mind, we will define a few of the popular models of computation that are relevant to our discussion of one-dimensional search, and summarize what is achievable in those models.

2.1.1 The RAM Model

In the RAM model, the computer's memory is modeled as a collection of w-bit memory cells that can be accessed by the algorithm via their addresses so that no explicit pointers to memory are needed. We assume that w bits is enough to address all of the memory that we will need, and that we will only be storing keys that fit in a constant number (usually just 1) of w-bit words. An algorithm is permitted to perform all of the standard arithmetic operations, such as addition and multiplication, on pairs of words. The cost of an algorithm is defined to be the number of memory accesses or arithmetic operations that are performed. The RAM model is the classic model of computation that is used most frequently in the field of algorithms and data structures, and can be used to implement algorithms defined in most more restrictive models with just a constant factor slowdown.

One natural solution to the search problem in the RAM model is hashing, which costs just $O(1)$ worst-case time per query if perfect hashing is used. Hashing requires access to a random number generator, and crucially relies on there being no unsuccessful queries, since hashing does not support finding successors. Note that if the set S were really $\{1, \ldots, n\}$, as we have assumed for simplicity, then we would not even need randomization. We could simply store all of the keys in an array, and use each key's value as its address in the array.

Unfortunately, when we stray only slightly from our simple definition of the search problem, and allow queries that fall in between two elements of S, we need to support efficiently finding the successor (or predecessor) of the queried element in the search space. With this extension, the RAM model still provides the ability to achieve better worst-case

performance than that which can be achieved in more restrictive models such as the comparison model, which is summarized in Section 2.1.2.

For example, we could use the y-fast tries of Willard [63], which store all of the prefixes of each $x \in S$ in a hash table T, and serve each query σ_j in $O(\lg w)$ time by performing binary search to find the length of the longest prefix of σ_j that appears in T. With a little more work, space usage can be reduced to linear by bucketing bunches of contiguous elements together, and storing the prefixes of only a single representative element of each bucket in T. To eliminate the need for randomization, the same bound on queries can be achieved via the van Emde Boas data structure [59, 58], although the use of randomization and hashing is still helpful in tightening the space usage bound from $O(2^w)$ to $O(n)$.

A slightly more complicated data structure for the search problem in the RAM model is fusion trees [29]. Essentially, fusion trees are B-trees (see Section 2.1.3) with a branching factor of $w^{\Omega(1)}$. By using word-level parallelism, fusion trees can find the correct child on the search path by essentially performing $\Omega(\lg w)$ comparisons in $O(1)$ time to achieve a worst-case query time of $O(\lg n / \lg w)$. Combining the van Emde Boas and fusion tree bounds to get the optimal tradeoff in terms of n yields a bound of

$$O(\min\{\lg w, \lg n / \lg w\}) = O(\sqrt{\lg n}).$$

It is important to note that the van Emde Boas data structure and fusion trees both require the keys to be integers, and they do not support augmentation within their specified time bounds. To guarantee such functionality, we need more restrictive models of computation.

2.1.2 The Comparison Model

Although the RAM model gives us more freedom as algorithm designers to exploit a rich instruction set, this flexibility places constraints and caveats on the input if we wish to exploit this flexibility to achieve the minimum possible running time. One simpler and more restrictive computational model is the comparison model, in which memory is modeled as a collection of memory cells, each with at most a constant number of movable pointers to other cells. The only mathematical operations that are permitted on keys are comparisons that determine whether one element is less than another element. The cost of an algorithm is defined to be the number of comparisons that are executed. This restriction is helpful because it allows us to abstract the type of data that is being stored. Instead of requiring the keys to be integers, as the RAM model does, the comparison model encapsulates search among arbitrary comparable objects.

The restrictions imposed by the comparison model relative to the RAM model come at a price because it is easy to see that no online search algorithm in the comparison model can achieve a worst-case running time of $o(\lg n)$ per query. However, the simple $\Omega(\lg n)$ lower bound on worst-case search cost assumes that queries are random, and this is often not the case. Although all of the BST algorithms with input-sensitive running time bounds are valid examples of comparison-based algorithms that can beat the $\Omega(\lg n)$ lower bound, we mention some notable adaptive comparison-based data structures that are not BSTs in the following paragraphs.

If the user of a search data structure has a reasonably accurate guess as to the location of the key for which they are searching (i.e., that particular access is *not* uniformly at random), then we can accelerate search by using a finger search tree. In a finger search tree, for each query σ_j, the user supplies a pointer to a *finger* f_j whose rank is ideally as near as possible to that of σ_j. Finger search trees serve queries with a running time of $O(\lg(|f_j - \sigma_j| + 2))$, where $|f_j - \sigma_j|$ represents the difference in sorted order ranks between σ_j and f_j. A finger search tree can be built to work in the comparison model by adding level links to a balanced BST, starting the search at the finger, and exponentially expanding the search space.

For example, Brown and Tarjan showed how to achieve finger search by adding level-links to a 2-3 tree [15]. Note that if we were to allow random access and did not care about supporting insertion and deletion, we could achieve the above finger search bound using an array and executing each search by starting at the finger and exponentially expanding the search radius in the obvious manner. Also, it is worth noting that finger search cannot be supported in the BST model because the user can always choose a finger f_j such that $|f_j - \sigma_j| = O(1)$ and the shortest path of pointers from f_j to σ_j has a length that is $\Omega(\lg n)$. Thus, a BST's pointer traversals may cost too much even if we charged only for pointer traversals rather than requiring the accessed element to be rotated to the root as specified in Section 2.2.

Another example of an input-sensitive bound that can be achieved in the comparison model is the Unified Bound, which can be achieved by using the Unified Structure [37, 16]. The Unified Bound stipulates that queries run much faster than $O(\lg n)$ in cases in which many of the keys that are queried are near to a recently accessed key. The Unified Structure achieves this bound by acting like a multi-level cache. It keeps a small set of recently used fingers in a small BST for which search is fast, and uses the recently accessed fingers to perform a quick finger search in a finger search tree. More details regarding the Unified Bound will be discussed in Chapter 4. Prior to the work in this thesis, it was unknown whether a BST could achieve the Unified Bound. The first BST algorithms to compete with the Unified Bound appear in Chapters 5 and 6.

2.1.3 Alternative Memory Models

The above alternatives to the BST model are similar to the BST model in that they assume a flat memory structure and charge for individual instructions. A different line of research is motivated by the observation that if a computer uses a hierarchical memory structure, as most computers do, then the running time of its programs is often dominated by the amount of time spent accessing slow memory. This phenomenon motivated the development of external memory and cache-oblivious models of computation. These two models allow rigorous analysis of the performance of algorithms whose running times are dominated by accesses to slower memory.

The external memory model [1] splits memory into two levels: fast memory of size M bits and slow memory of unlimited size. All data is assumed to be divided into contiguous pages of memory containing B bits each. All computation is performed on data that resides in fast memory, and pages are swapped into fast memory from slow memory whenever their data is needed. A constant cost is associated with each swap of a page from slow memory to fast memory, and all other operations are assumed to be free. Assuming b keys fit into a page of memory, balanced B-trees provide a worst-case running time of $O(\log_{b+1} n)$. This bound is overly pessimistic if several search paths fit into fast memory and recently accessed elements are likely to be queried so that it is possible that there are virtually no cache misses.

The cache-oblivious model [48, 30] assigns costs similarly to the external memory model, but the parameters M and B are unknown. Therefore, the goal is to lay out the data structure in memory in a way such that it performs well regardless of the values of M and B that occur when the data structure is used. Because results in the cache-oblivious model hold for all values of M and B, the cache-oblivious model yields more powerful results than the external memory model. In fact, since modern memory architectures often have more than two levels of memory, results in the cache-oblivious model are especially powerful because they obviate the need to study each level separately. By laying out a balanced BST carefully in memory, it is possible to achieve a worst-case running time of $O(\log_{b+1} n)$ per query [5, 7, 13, 6], the same bound that B-trees achieve in the external memory model.

These results in alternative memory models are not directly comparable to the results in this thesis, but they are included because they provide context for alternative views on how to measure the cost of a search algorithm, and because finding adaptive data structures in such memory models may be a fruitful direction for future research. It is worth noting that accelerating splay trees using increased arity seems to be difficult because the natural extension of splay trees yields a bound that is no better than that which is achieved by ordinary splay trees [52, 45].

2.2 Definition of the BST Model

Although most people who study computer science are familiar with the concept of using a BST to solve the search problem, it is important to formally define what a BST algorithm is to facilitate the proof of input-sensitive lower bounds and develop competitive BST algorithms. In this thesis, we use the definition of the BST model that is similar to that which was described by Wilber [62], and we describe this definition below.

We define a BST node to be a memory cell that contains three pointers: a parent pointer, a left child pointer, and a right child pointer. In addition, each node also stores a key and may store additional data in auxiliary fields. Each of the n keys that are stored in a BST has its own BST node, and these n BST nodes are linked together into a binary tree T in symmetric order so that every node that can be reached by following child pointers starting from the left child of a node has a key value that is less than that of the original node. Typically, we only want to use $O(1)$ auxiliary fields per node, and we forbid extra pointers from being used, but these extra requirements are not necessary for our strict definition of the BST model. Even without these extra restrictions, augmentation can still be achieved for any BST, and the proofs of the lower bounds in Chapter 3 are still valid.

During the query sequence, the structure of T is only modified by an operation called a *rotation*, which is defined on every BST node that has a parent. During a rotation of node x over its current parent p, the node x becomes the parent of p, and one of x's children is moved to be a child of p so that the constraint that T is binary is maintained. Note that if T were to be represented by the ancestor relation over the set of BST nodes, a rotation removes the pair (p, x), adds the pair (x, p), and keeps all other ancestor relationships the same. An important characteristic of the rotation operation is that it is entirely local. Only a constant number of pointers need to be changed to execute it, and it does not affect any parent-child relationships anywhere else in the tree. Examples of each of the two types of rotations, right and left, are shown in Figure 2.1.

Next, we define a BST *algorithm* A to be a procedure that, given as input an initial set of elements $S = \{1, \ldots, n\}$ and a sequence of queries $\sigma = \sigma_1 \cdots \sigma_m$, produces an initial BST T containing S and a sequence of valid rotations $r_1 \cdots r_t$ on that tree. For A to be considered a valid BST algorithm for serving sequence σ, there must be a monotonically increasing sequence of time indices $k_1 \cdots k_m$ such that each $k_j \in \{0, \ldots, t\}$ and for $j \in \{1, \ldots, m\}$, the node representing σ_j is the root node of T immediately after rotation r_{k_j} is executed (if $k_j = 0$ then σ_j must be the initial root of T, before any rotations have been performed).

The cost of a BST algorithm on access sequence σ is defined to be $m + t$. All other computation is considered to be free. Note that for an online BST algorithm, the initial

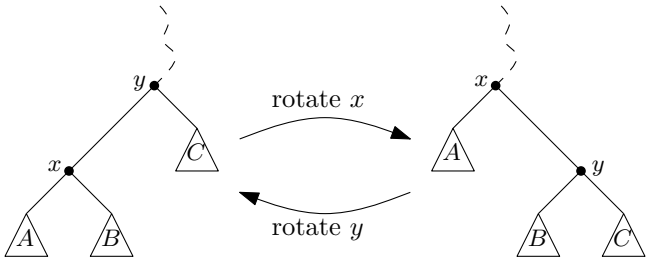


Figure 2.1: An example of a BST rotation and its mutual inverse. The rotations of x and y are termed, respectively, right and left rotations.

structure of T is independent of the input, and the sequence of rotations $r_1 \cdots r_{k_j}$ is independent of $\sigma_{j+1} \cdots \sigma_m$ for some valid setting of $k_1 \cdots k_m$ as defined above.

Essentially, this definition is saying that a BST algorithm must rotate each accessed node to the root of the tree. This definition poses no problems for the splay algorithm [55]. However, other classic BST algorithms, such as red-black trees [4, 34], do not rotate the accessed node to the root of T, and are analyzed by charging for pointer traversals rather than rotations. Nevertheless, such BST algorithms can easily be coerced into Wilber's model with just a constant factor penalty to running time by rotating each accessed node to the root and back down to its location before the extra rotate-to-root operation was performed. In Chapters 5 and 6, we will describe the skip-splay and cache-splay BST algorithms, which must be coerced into this model because they do not rotate the queried node all of the way to the root.

Even though this definition of the BST model does not strictly adhere to the comparison model, a BST algorithm with the above definition can be coerced into the comparison model by noting that for each query, the full access path must be rotated for the queried node to become the root. Each of these rotations corresponds to a comparison that would be performed if we were analyzing a BST algorithm in the comparison model.

Before we conclude this chapter, it is important to note how the BST model compares with other models of computation. Since the BST model allows for a strict subset of the algorithms that are permitted in the RAM and comparison models, a BST algorithm can be no faster than the best algorithms in the more flexible models, though the cost accounting may be lower for a BST in some cases because BSTs are only charged for rotations. Due to the constraints imposed by the BST model, there is trivial lower bound of $\Omega(\lg n)$ on the worst-case cost per query for an online BST. The proof that offline BSTs require $\Omega(\lg n)$ time in the worst case is a little more difficult, but can nevertheless be shown [62, 9]. As in the comparison model, we can beat these lower bounds by exploiting nonrandomness

in the input to achieve running times that are a function of some property of the input. Numerous examples of this will be discussed in Chapters 4, 5, and 6.

The added restrictions on BSTs also convey benefits. Every BST can be *augmented* so that each node stores the value of some associative function over the elements in its subtree. For example, if a numerical value, not necessarily the key, is stored in each node x, then we can store inside x the sum of all such values that belong to nodes in x's subtree. It is straightforward to update such sums in constant time whenever a rotation is performed, and the value itself can be changed when the node is at the root of T because it appears in no other node's subtree. This is an important attribute of BSTs, and this feature makes BSTs useful for countless applications, including solving the partial-sums problem as will be described in Section 3.6.

Chapter 3

Lower Bounds in the BST Model

Using the formal definition of the BST model originally presented by Wilber [62] and reproduced in Chapter 2, we can get a better understanding of what work must be performed by any BST algorithm that serves a sequence of queries $\sigma = \sigma_1 \cdots \sigma_m$, where each query σ_j is to a member of $S = \{1, \ldots, n\}$.

A straightforward information-theoretic argument shows that any online BST algorithm must pay $\Omega(\lg n)$ rotations for each query in the worst case because if each successive query is chosen randomly, the expected depth of each queried node is $\Omega(\lg n)$, so $\Omega(\lg n)$ rotations are required to bring the queried node to the root as required. Moreover, even an offline BST algorithm can be shown to require $\Omega(\lg n)$ rotations per query in the worst case via an information-theoretic argument that was presented by Blum *et al.* in [9]. Blum *et al.* further showed that at most $2^{O(km)}$ sequences of length m have optimal cost at most $O(km)$.

To summarize their argument, we can show that the rotations $r_1 \cdots r_t$ of a BST algorithm can be encoded in $O(t)$ bits, so algorithms with t rotations can serve at most $2^{O(t)}$ distinct sequences. To see that a BST algorithm can be encoded in $O(t)$ bits, note that any BST algorithm can be converted, at no additional cost, into one that during each query performs rotations only on a connected set of nodes including the root [44]. Further, the rotations performed on a connected set of nodes of size k including the root can be encoded in $O(k)$ bits by writing a binary encoding of two Euler tours of these k nodes: one tour that shows the structure of these nodes prior to the access and another that shows the structure of them afterward.

It is worth emphasizing the importance of being able to prove a nontrivial lower bound for an offline algorithm. Without this capability, we would have little hope of proving an online algorithm to be dynamically optimal. Consider the comparison model with arbitrary

pointers. The optimal offline algorithm can essentially guess every query and serve any sequence at $O(1)$ cost per operation, making it impossible for any online search algorithm in the comparison model to have a competitive factor that is any better than the trivial factor of $O(\lg n)$.

Although the above information-theoretic bound gives us an idea of how many sequences may have lower optimal BST access costs than the pessimistic bound of $O(\lg n)$, it is not clear what those sequences might be. Chapter 4 discusses some upper bounds that give us an idea of some of the kinds of sequences for which it *is* possible to achieve better than $O(\lg n)$ cost per access, but this still does not tell us what *is not* possible. To determine what we cannot achieve with a BST, we need *lower bounds* that are capable of assigning a minimum cost that *any* BST algorithm must pay to execute a query sequence.

The first nontrivial instance-specific lower bounds for the BST model were shown by Wilber [62], who proved two lower bounds that we will call "Wilber's first lower bound" and "Wilber's second lower bound". Subsequent work has built on the ideas in the Wilber bounds to achieve an even better understanding of what is, and is not, achievable in the BST model.

In this chapter, we summarize some of the previous work in BST lower bounds, and then introduce a more general framework for computing BST lower bounds in which a set of *boxes* is packed onto a two-dimensional representation of the query sequence σ. The number of boxes will be shown to be a lower bound on the cost that any BST algorithm must pay to execute σ, and the largest such bound is called the maximum independent box-set (MIBS) lower bound. We will see that each of the previous lower bounds is at most the value of the MIBS lower bound. Additionally, we will see that if we allow fractional boxes, the optimal fractional MIBS solution also provides a lower bound on the cost of the optimal BST, and is at least as large as the optimal integral solution. Moreover, this fractional solution can be computed in polynomial time using linear programming.

One interesting aspect of the MIBS bound is that it bolsters the connection between the BST model and the partial-sums problem, which also has a worst-case lower bound of $\Omega(\lg n)$ per operation, even in the more general cell-probe model of computation [50, 51]. It is obvious that any BST algorithm, with augmented BST nodes, can be used to solve the partial-sums problem with a cost that is dominated by the BST operations required by this approach. What is interesting about the MIBS lower bound is that it provides evidence that the converse may be true. That is, it provides evidence that the partial-sums problem cannot be solved asymptotically faster than by using an augmented dynamically optimal BST. We will present more details about this in Section 3.6.

In this chapter, we will frequently make use of a two-dimensional visual representation of a query sequence. To construct the two-dimensional representation of a query sequence,

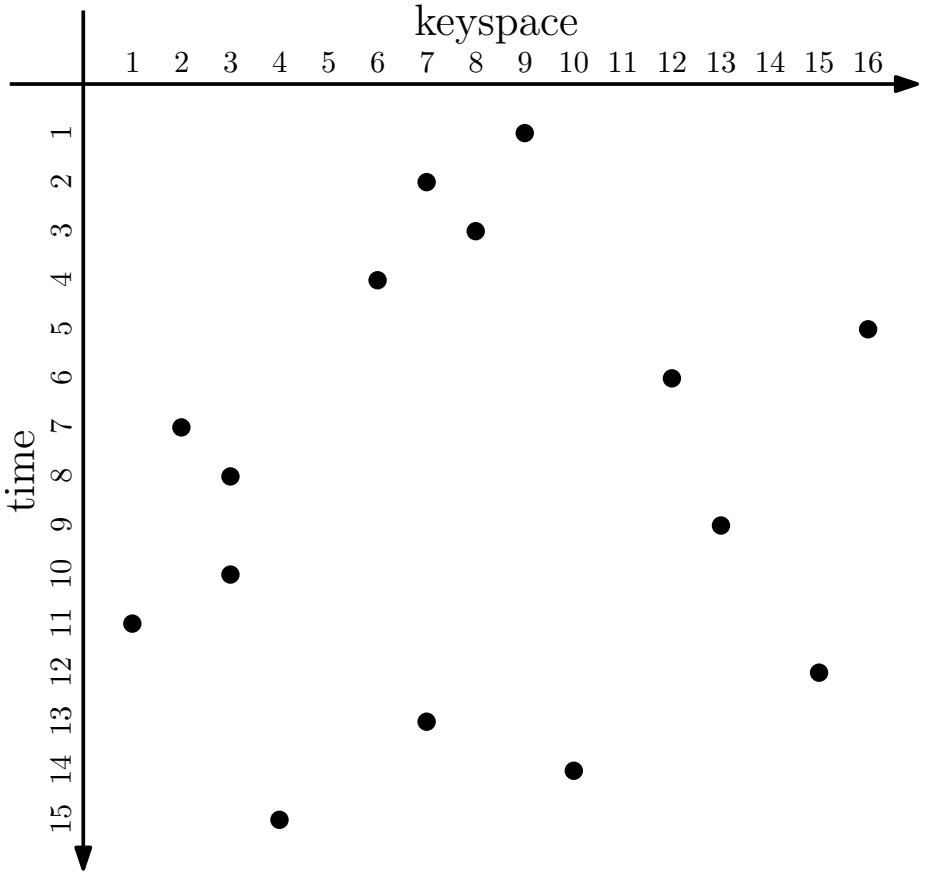


Figure 3.1: A two-dimensional visual representation of the following query sequence: $\sigma = 9, 7, 8, 6, 16, 12, 2, 3, 13, 3, 1, 15, 7, 10, 4$.

we simply plot the queries on a scatter plot with keyspace on the horizontal axis, and time on the vertical axis, increasing in the downward direction. An example of such a two-dimensional visualization of a query sequence is shown in Figure 3.1.

3.1 Wilber's First Bound and the Interleave Bound

Wilber's first lower bound uses a fixed lower bound tree P with $2n - 1$ nodes. The leaves of P are exactly the elements of S, and each internal node v of P has at least one *preferred child*, which is typically the child whose subtree contains the most recent query. The only exception occurs when neither child's subtree contains a previous query, in which case v

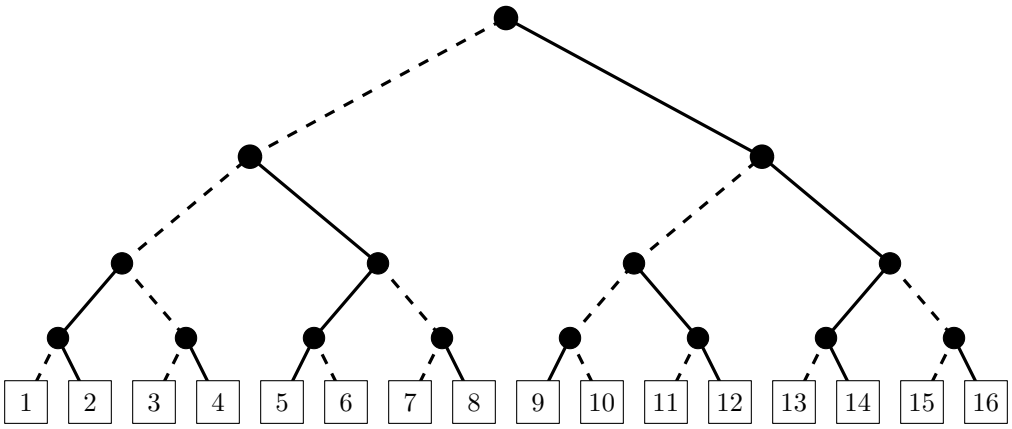


Figure 3.2: The state of a lower bound tree P for Wilber's first bound. Note that the set $S = \{1, \ldots, 16\}$ of elements that is stored in T is stored in the *leaves* of P.

prefers both children, if v has two children. If we execute the query sequence σ on P, count the number of times an unpreferred child of v becomes a preferred child (i.e., the number of times the preferred child switches from one to the other), and add up these counts across all nodes of P, then the resulting sum, $\mathrm{Wil}_1(\sigma, P)$, is a lower bound on the number of rotations required by any BST algorithm for executing the query sequence σ. An example of the state of the reference tree for Wilber's first lower bound is shown in Figure 3.2.

The interleave bound, introduced by Demaine *et al.* in [21, 22], is a modification of Wilber's first lower bound that changes the definition of P to include exactly the n nodes of S. Other than that change, the interleave bound, denoted by $\mathrm{IB}(\sigma, P)$, is defined identically to Wilber's first lower bound. The important difference between the interleave lower bound and Wilber's first lower bound is that the lower bound tree for the interleave lower bound does not have extra nodes in addition to those of the keys of S. This characteristic is helpful for proving BST algorithms to be $O(\lg \lg n)$-competitive as we will discuss in more detail in Chapter 4. An example of the state of the lower bound tree for the interleave bound is shown in Figure 3.3. Note how this tree compares with the lower bound tree of Wilber's first lower bound that is shown in Figure 3.2.

Note that Wilber's first lower bound and the interleave lower bound are both valid for any lower bound tree, and the optimal lower bound tree for a specific sequence can be found in polynomial time using dynamic programming. Also, these two bounds are superficially very similar, and we can show that they are the same, up to a constant factor, if the optimal lower bound tree is used for both of them. We show this by proving the

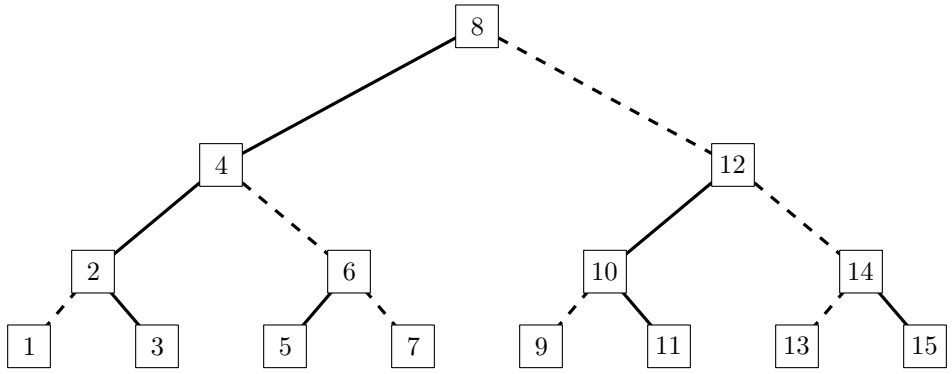


Figure 3.3: The state of a lower bound tree P for the interleave bound. Note that P contains the set $S = \{1, \ldots, 15\}$, exactly the same set of elements that is stored in T.

following two theorems.

Theorem 1. *For every query sequence σ and n-node lower bound tree P for the interleave bound, there exists a corresponding lower bound tree P' for Wilber's first lower bound containing $2n - 1$ nodes such that $\mathrm{Wil}_1(\sigma, P') \geq \mathrm{IB}(\sigma, P)$.*

Proof. Create a corresponding reference tree P' that is initially identical to P. Then, redefine all of the keys of P' so that each key $x \in P'$ changes to $x + \frac{1}{2}$. Next, insert the elements $\{1, \ldots, n\}$ into P' using the standard BST insert algorithm. Finally, delete the node $n + \frac{1}{2}$ in P' so that each of the internal nodes of P' has two children, and the set S is completely stored at the leaves of P.

Now, execute the query sequence σ for both lower bound trees. Consider a switch in P at an arbitrary node x that has its preferred child changed from the left to the right. Notice that x was previously set to prefer left when a query σ_j was executed in x's left child's subtree, and that the internal node $x' = x + \frac{1}{2}$ also preferred left after σ_j was executed. Further, the next query $\sigma_{j'}$ that causes a switch at x in P is to a member of x's right child's subtree in P. It follows that x' also prefers to the right in P' after $\sigma_{j'}$ is executed. Therefore, at least one switch has been performed at x' during this time interval. A similar argument applies to all nodes in P and to right-to-left switches as well so that $\mathrm{Wil}_1(\sigma, P') \geq \mathrm{IB}(\sigma, P)$. □

Theorem 2. *For every query sequence σ and lower bound tree P' for Wilber's first lower bound with $2n - 1$ nodes, there exists a corresponding n-node lower bound tree P for the interleave bound such that $3\,\mathrm{IB}(\sigma, P) + 5m + n \geq \mathrm{Wil}_1(\sigma, P')$.*

Proof. Our first step is to create a new bound $\mathrm{IB}'(\sigma, P)$ that is identical to $\mathrm{IB}(\sigma, P)$ except that a query to an internal node x sets all of x's children to be unpreferred. The value of $\mathrm{IB}'(\sigma, P)$ is defined to be the number of times an unpreferred child changes to a preferred child (recall that at the beginning all edges are defined to be preferred edges). Clearly, $\mathrm{IB}'(\sigma, P) \leq \mathrm{IB}(\sigma, P) + m$, so it suffices to show that $3\,\mathrm{IB}'(\sigma, P) + 2m + n \geq \mathrm{Wil}_1(\sigma, P')$ for some P.

To construct P, given an arbitrary reference tree P' for Wilber's first lower bound, we store every key x' of P' except n in its successor in P', and we delete all of the leaves except n.

To show that $3\,\mathrm{IB}'(\sigma, P)+2m+n \geq \mathrm{Wil}_1(\sigma, P')$, we will define a potential function on P' that is equal to the negative of the number of right preferred edges in P' that correspond to unpreferred edges in P (we will call such an edge a "bogus right edge"). The amortized number of switches in P' is equal to the number of actual switches in P', plus the change in potential.

Let us consider the amortized number of switches resulting from an arbitrary query to node v in P'. Notice that v's access path in P' is the same as its access path in P, except that the path in P' may consist of an additional left inner path (as defined in [62]), consisting of only right edges, whose shallowest node is the left child of $v' = v + \frac{1}{2}$. (This definition of v' assumes that the internal nodes of P' represent the midpoints between the members of S.)

We pay for the switch (and possible destruction of a bogus right edge) at v' in P with our extra allotment of two switches per query, and we note that the amortized number of switches that occur in this left inner path is zero because each such switch creates a bogus right edge (this is true because if a node u on the corresponding path in P prefers to the right, then the last query in u's subtree was in its right child's subtree, which implies that $u' = u + \frac{1}{2}$ should prefer to the right in P' as well).

Therefore, we restrict our attention to the switches that occur on the access path to v in P and to $v' = v + \frac{1}{2}$ in P'. Our goal will be to show that each switch in P is responsible for paying for at most three amortized switches in P'. To show this, we consider the following three exhaustive cases for switches that occur at an arbitrary node $x' = x + \frac{1}{2}$ in P'.

First, suppose there is a switch at both x and x'. In this case the switch at x pays for the switch at x', and we are done because there is no change in potential due to these switches.

Second, suppose there is a switch at x but not at x'. In this case, there may be an amortized switch in P' due to the destruction of a bogus right edge, and this is paid for by the switch at x.

Third, suppose there is a switch at x' but not at x. In this case, note that the switch at

x' must be right-to-left because if x prefers right before this query, x' must also. In this case, we charge the switch at x' (and the amortized switch resulting from the destruction of the bogus right edge) to the deepest node u that is switched in P on the path from x to x's right parent. Because u is required to be on this path, it is only charged for one switch of a bogus right edge. Further, u must exist because every bogus right edge is created by a query to its right parent p in P, which creates an unpreferred child edge to p's left. While the edge remains bogus, it cannot be traversed in P, and every query that does not alter this edge's status as a bogus right edge leaves an unpreferred edge to the right of the deepest member of p's left inner path that is traversed during that query.

Note that in the above cases, no switch in P is charged for more than three amortized switches in P', and the potential function's minimum value is no less than $-n$, so the theorem follows. □

In Section 3.5.1, after we define and prove the MIBS lower bound, we will show that the MIBS lower bound is at least as large as Wilber's first lower bound. In addition to showing the strength of the MIBS lower bound, this essentially serves as an alternative proof to Wilber's original proof of his first bound. Together with Theorem 1, this proves that the interleave bound is also a valid lower bound, and this serves as an alternative proof to the original proof of the interleave bound.

As stated above, the interleave lower bound, and by extension, its original version, Wilber's first lower bound, are useful for proving a variety of BST algorithms to be $O(\lg \lg n)$-competitive. This raises the question of whether it might be possible to prove an algorithm is $o(\lg \lg n)$-competitive using a similar technique. One difficulty of achieving such a result is that the interleave lower bound is loose by a factor of $\Omega(\lg \lg n)$ for any specific lower bound tree. To see this, note that any lower bound tree for the interleave bound must have at least one path of length $\lg n$, and accessing only nodes on this path causes no switches, while random access to this path requires $\Omega(\lg \lg n)$ rotations per operation. By Theorem 2, every lower bound tree for Wilber's first lower bound has sequences for which the lower bound is also loose by a factor of $\Omega(\lg \lg n)$. It is not immediately clear whether the optimal tree for any specific access sequence yields a bound that can be loose by a factor of $\omega(1)$ though this seems likely.

3.2 The Dynamic Interleave Lower Bound

There are a couple of shortcomings of the original interleave lower bound. First, it is a static lower bound because it does not handle insertion or deletion, so it is not clear how the lower bound could handle competitiveness in a dynamic setting. Second, as discussed

in Section 3.1, for every fixed lower bound tree, the interleave bound is loose by a factor of $\Omega(\lg \lg n)$ for some access sequences. This is troubling because it suggests that it will be difficult to prove a Tango-like BST to be $o(\lg \lg n)$-competitive using a fixed lower bound tree. This motivates the idea of allowing rotations on the lower bound tree, which was explored by Wang, Sleator, and me in [61].

To summarize the technique here, we can create a version of the interleave lower bound that allows rotations as follows. A particular instance of the dynamic interleave lower bound can be described by an initial lower bound tree P along with a sequence of rotations to be performed on P at the end of each access. These rotations can depend on the access sequence if needed. As in the static interleave lower bound, each internal node has no unpreferred children at the beginning of the access sequence and all of its children are preferred. During a sequence of BST operations, after each access, every node on the access path except the root is defined to be a preferred child, and every sibling of some node on the access path is defined to be an unpreferred child. As for the interleave bound, we increase the lower bound by one for every unpreferred child that becomes preferred. After the access path is transformed into a preferred path, the lower bound tree is potentially modified via rotations. For each node x that is one of the two nodes that are involved in such a rotation, both of x's children are set to be preferred in the lower bound tree with no increase to the lower bound. Note that for the purpose of creating a Tango-like algorithm, we would maintain an invariant that every internal node in the lower bound tree has at most one preferred child with the appropriate adjustment to the lower bound.

As suggested above, there are two benefits to this bound. First, as described in [61], we can keep the lower bound tree P balanced while inserting and deleting elements into and from P. This requires only a constant number of rotations per dynamic operation, so it does not greatly affect our lower bound, and allows us to achieve $O(\lg \lg n)$-competitiveness in a dynamic setting using Tango, multi-splay trees, or any other algorithm using the same approach to BST competitiveness. To fully support competitiveness in a dynamic setting, we also need to modify Wilber's BST model to allow insertions and deletions. This detail is described in [61].

Second, the ability to rotate the lower bound tree suggests an approach to improve upon the current-best competitive factor of $O(\lg \lg n)$ for a BST. Note that the factor of $O(\lg \lg n)$ for Tango and multi-splay trees stems from the fact that each switch of a child from unpreferred to preferred corresponds to a constant number of BST operations in a BST of size $O(\lg n)$. Intuitively, if many queries visit the same preferred path of length $\Omega(\lg n)$, then the lower bound tree for the dynamic interleave bound could perform rotations on this path to decrease its length so that each traversal of this preferred path would have a smaller running time bound. Although this idea for improving the best known

competitive ratio for a BST algorithm shows some promise, it seems to remain difficult to achieve any provable results.

To get a sense of the difficulty, suppose we wanted to prove that splay trees were dynamically optimal by using the splay tree itself as the lower bound tree. During each access, we would create a solid path to the accessed node in the splay tree with unpreferred edges to the siblings of nodes on the access path. However, when we splayed the accessed node to the root, we would destroy all of the unpreferred edges that we just created before we could switch them to preferred and count them towards the lower bound. This prevents us from showing a lower bound of more than a constant per operation for any access sequence using a simple application of the dynamic interleave lower bound.

One might think that this problem can be circumvented by performing path compressions without rotating every edge on the access path so that many unpreferred edges remain after the rotations for each access are executed (to conform to the BST model, we could rotate these edges and then unrotate them without affecting the lower bound). However, the above problem of many unpreferred edges being destroyed still occurs when a few repeated accesses are executed on the same node. Thus, new ideas are needed for this approach to proving dynamic optimality to work.

3.3 Wilber's Second Lower Bound

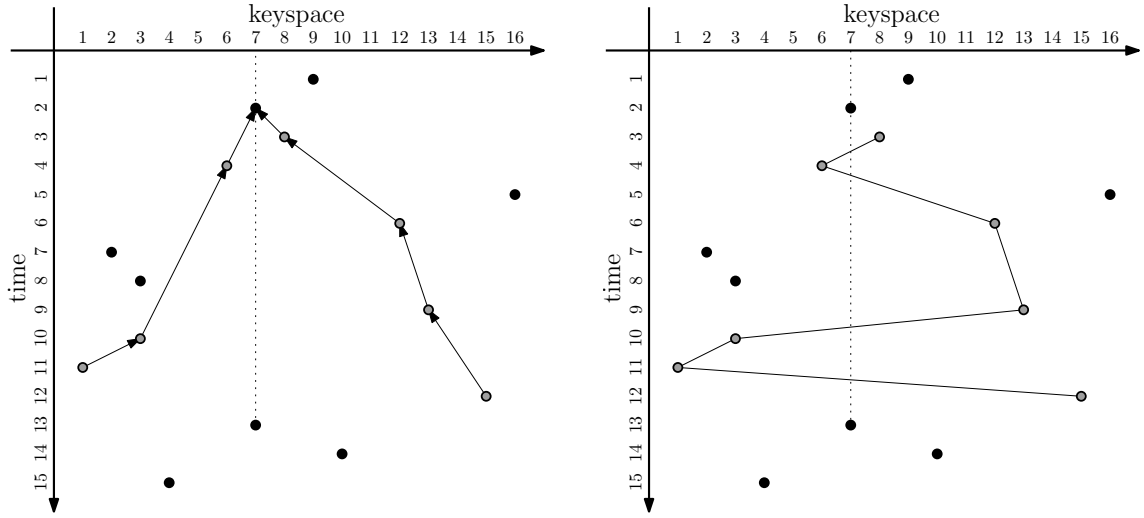
Although Wilber's first lower bound has so far proved to be the more useful bound in terms of developing competitive BST algorithms, it is worth noting that Wilber introduced a second lower bound [62], which we briefly describe below.

For an access sequence $\sigma = \sigma_1 \cdots \sigma_m$ consisting only of queries, we define the Wilber number of each query σ_j as follows (See Figure 3.4 first for a more intuitive visual definition of the Wilber number). First, we give the following two definitions:

$$\begin{aligned} \mathrm{right}(j_1, j_2) &= \{j' \in \{j_1 + 1, \ldots, j_2 - 1\} \mid \sigma_{j'} \geq \sigma_{j_2}\}, \\ \mathrm{rightRecords}(j) &= \{j' \in \mathrm{right}(0, j) \mid \sigma_{j'} < \min_{j'' \in \mathrm{right}(j', j)} \sigma_{j''}\}. \end{aligned}$$

Then, we define $\mathrm{left}(j)$ and $\mathrm{leftRecords}(j)$ analogously, and following that we define $\mathrm{records}(j) = \mathrm{rightRecords}(j) \cup \mathrm{leftRecords}(j)$. Figure 3.4(a) shows a visual depiction of these sets of records for one access. Second, define query j' to be a *crossing access* for query j if one of the following two conditions holds:

$$\begin{aligned} j' \in \mathrm{leftRecords}(j) \wedge \mathrm{succ}_{\mathrm{records}(j)}(j') \in \mathrm{rightRecords}(j) \\ j' \in \mathrm{rightRecords}(j) \wedge \mathrm{succ}_{\mathrm{records}(j)}(j') \in \mathrm{leftRecords}(j). \end{aligned}$$



(a) An example of the progression of "records" on each side of a the query σ_{13}. The right path corresponds to the set $\mathrm{rightRecords}(13)$, and the left path corresponds to the set $\mathrm{leftRecords}(13)$. The queries that are filled in with light gray are the members of $\mathrm{records}(13)$.

(b) The connections between each member of the set $\mathrm{records}(13)$ and its successor in the same set. Note that the crossing accesses for query σ_{13} are the members of $\mathrm{records}(13)$ whose successor crosses to the other side of the dotted line.

Figure 3.4: A visualization of the definition of Wilber's second lower bound. Subfigures (a) and (b) show how to compute the Wilber number for a particular access, in this case σ_{13}.

Figure 3.4(b) shows a visualization of these crossing accesses as compared to the records shown in Figure 3.4(a). The Wilber number, $\mathrm{Wil}_2(\sigma, j)$, of query σ_j is defined to be the number of crossing accesses for query j, and Wilber's second lower bound is defined as $\mathrm{Wil}_2(\sigma) = \sum_{j=1}^{m} \mathrm{Wil}_2(\sigma, j)$.

An alternative way of understanding Wilber's second lower bound is that it is the number of "corners" (i.e., a right child followed by a left child or vice versa) that are encountered on the access path of each queried node during the execution of the most basic "rotate-to-root" BST algorithm that simply rotates each queried element repeatedly until it becomes the root of the tree. To see this, note that rotate-to-root is equivalent to a treap for which the priorities are set according to how recently a node has been accessed, and each node on an access path corresponds to a record using the above definition of a record while each corner corresponds a new record on a different side of the accessed node (i.e., a crossing access).

Although Wilber's second lower bound does not currently have any practical use, one

advantage that it has over Wilber's first lower bound is that it is not known that $\mathrm{Wil}_2(\sigma)$ is loose by more than a constant factor for any query sequence σ. On the other hand, however, it is also not known whether $\mathrm{Wil}_2(\sigma)$ can ever be smaller than $\mathrm{Wil}_1(\sigma, P)$ by more than a constant factor for any σ and lower bound tree P for Wilber's first bound.

3.4 The Independent Rectangle Lower Bound

Demaine *et al.* devised a lower bound that was at least within a constant factor of both Wilber's first lower bound and Wilber's second lower bound [20, 36]. This lower bound was developed independently of our generalization of Wilber's bounds [26], and it is not clear what the relationship is between them, though it seems likely that they are asymptotically the same on all inputs.

In their lower bound, Demaine *et al.* use a slightly different definition for the BST model from that which we are using in this thesis. They require all searches to start from the root and follow the pointers of the BST, and charge a BST algorithm for every node it *touches* while allowing arbitrary restructuring on all touched nodes. It is easy to see that this definition is computationally equivalent to Wilber's definition, as defined in Chapter 2, to within a constant factor because any BST of k nodes can be transformed into any other tree of k nodes using just $2k - 6$ rotations [54]. Even though Wilber's model does not require the rotated nodes to form a connected set, it is easy to see that such "disconnected rotations" can be performed lazily at no additional cost [44].

Essentially, the lower bound of Demaine *et al.* is defined as follows. Let each pair of queries (i, j) to distinct elements form an axis-aligned *rectangle* in a two-dimensional representation of an access sequence, such as the one shown in Figure 3.1. These rectangles are identically specified to those of the MIBS lower bound defined in Section 3.5 and shown in Figure 3.5(a), with the exception that there is no divider in the rectangles used for the independent rectangle lower bound.

Demaine *et al.* define two rectangles to be *dependent* if one of the rectangles has a corner inside of the other or one of the points defining the box is on the border of the other, and they define the rectangles to be *independent* otherwise. They show that if $\mathcal{R}$ is a set of independent rectangles for a query sequence σ, then the cost of the optimal BST algorithm for executing σ in the "node-touch" definition of the BST model is at least $|\mathcal{R}|/2 + m$.

It is worth noting that Harmon's thesis contains another type of lower bound called a *cut bound*, which was also independently developed using the node-touch model. The definition of this bound is more complicated, but it is equivalent [36] to the independent rectangle bound of [20, 36] that is described above.

3.5 The MIBS Lower Bound

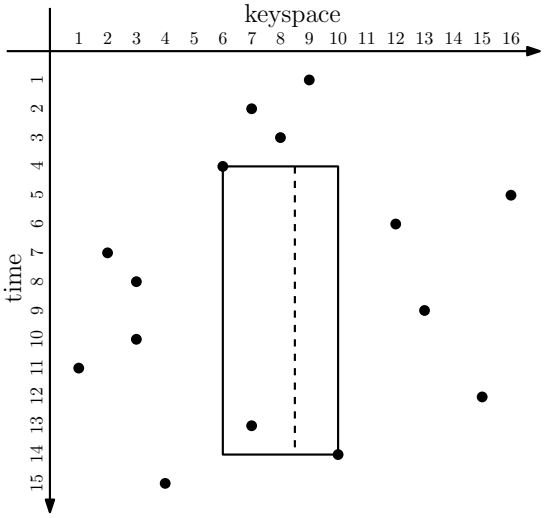
In this section, we introduce a lower bound called the independent box-set lower bound that is similar to the independent rectangle lower bound defined in [20]. As stated in Section 3.4, is it not currently known whether either bound is always at least within a constant factor of the other, and the two bounds apply to different definitions of the BST model. However, both bounds are always at least as large as Wilber's second lower bound, and it is also unknown whether Wilber's second lower bound is within a constant factor of the cost of the optimal BST, so it could be the case that all three bounds are within a constant factor of $\mathrm{OPT}(\sigma)$ for all sequences σ.

Essentially, the independent box-set lower bound is a simple, geometric framework that facilitates proofs that certain rotations must be performed by any BST algorithm for serving the specified sequence. Thus, the ultimate goal is to show that some BST algorithm "must" perform all of its rotations, or at least a constant fraction of them, so that the algorithm will be proved to be dynamically optimal. Although such a result has not been achieved, we hope that this framework will be helpful in understanding near-optimal BST algorithms in the future.

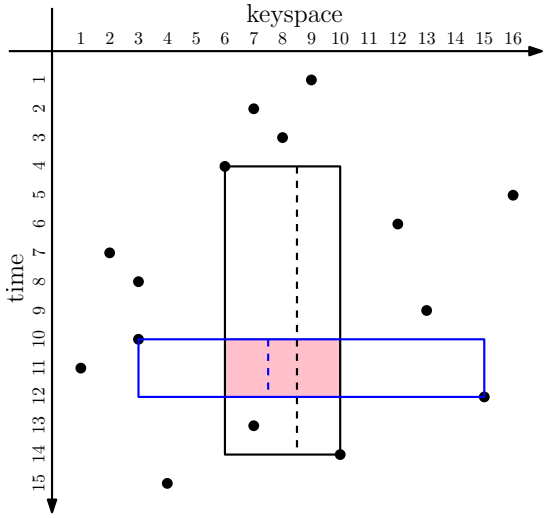
The description of the independent box-set bound here is slightly modified from the original presentation of the bound in [26], but the difference in the definitions is superficial. Using a two-dimensional representation of an access sequence, such as that which appears in Figure 3.1, we define a *box* to be an axis-aligned rectangle that has two of its corners located at points corresponding to two queries to distinct elements. Additionally, each box has a vertical *divider* located in the box's horizontal range.[1] The horizontal coordinate of this divider is restricted to be at the midpoint between two successive keys. Formally, given two queries σ_i and σ_j with $i < j$, box (i, j, z) is valid exactly when $\sigma_i \neq \sigma_j$ and z is a midpoint between two integers such that $\min\{\sigma_i, \sigma_j\} < z < \max\{\sigma_i, \sigma_j\}$ (for a general keyset, the constraints on z could be defined differently to ensure that the divider is distinct from the keyset). To give an example, if $\sigma_3 = 4$ and $\sigma_5 = 8$ then $(3, 5, 6.5)$ is a valid box. Another example of a valid box is shown in Figure 3.5(a). We define $\mathcal{B}(\sigma)$ to be the set of valid boxes for query sequence σ.

If $\sigma_i < \sigma_j$ then box (i, j, z) is directed *left-to-right*, else it is directed *right-to-left*. Note that the box in Figure 3.5(a) is directed left-to-right. We declare that two boxes (i, j, z) and (i', j', z') *conflict* if both boxes have the same direction, the boxes intersect, and their intersection contains part or all of both dividers. Examples of conflicting and non-conflicting boxes are shown in Figures 3.5(b) and 3.5(c).

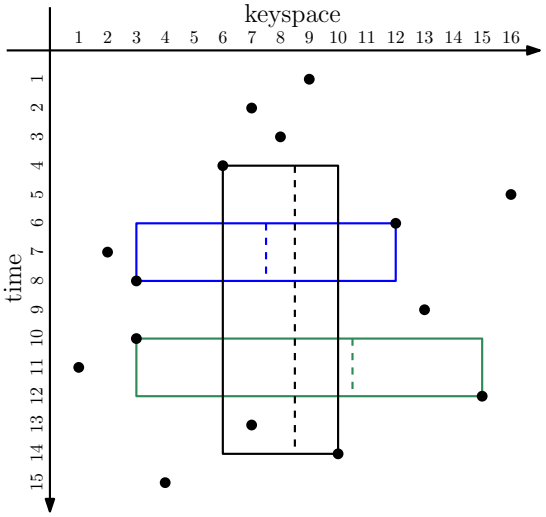
[1] As noted in [26], the divider could traverse any path that moves monotonically across the keyspace from the first query of the box to the second. It is unknown whether this is helpful on any sequences.



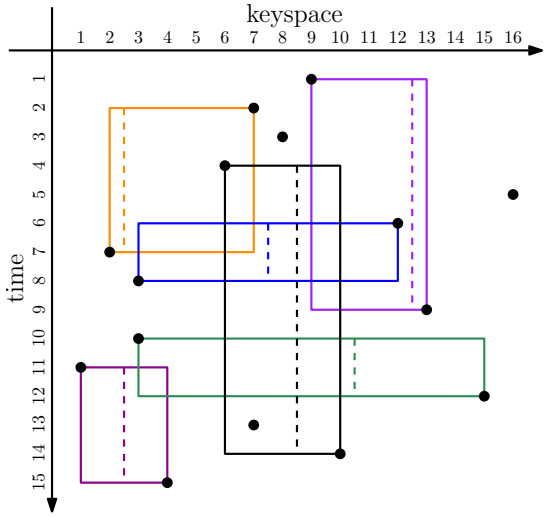
(a) The valid box $(4, 14, 8.5)$ for the depicted access sequence. This box is directed left-to-right.



(b) Two boxes that conflict because both are left-to-right and their intersection contains both dividers.



(c) The black and blue boxes do not conflict because they are in different directions, and the black and green boxes do not conflict because their intersection does not contain both dividers.



(d) An example of how to pack many independent boxes onto the scatter plot of an access sequence. This set is not maximal because additional boxes could be added without creating conflicts.

Figure 3.5: An isolated box is shown in (a), and a conflicting box is shown in (b). Examples of intersecting but nonconflicting boxes are shown in (c), and (d) shows how an algorithm might choose many independent boxes to achieve a good lower bound (the depicted set of independent boxes is not maximal).

It is straightforward to define the independent box-set lower bound once we have defined when boxes conflict. The size of any set of independent boxes, such as that which is shown in Figure 3.5(d), is a lower bound on the cost that a BST algorithm must pay to execute σ. Formally, we have the following theorem:

Theorem 3. *For the query sequence $\sigma = \sigma_1 \cdots \sigma_m$, let $B \subseteq \mathcal{B}(\sigma)$ be a set of valid, non-conflicting boxes. Any BST algorithm for executing the queries in σ must perform at least $|B|$ rotations.*

Proof. It suffices to provide a one-to-one mapping $f : B \to \{r_1, \ldots, r_t\}$ where $r_1 \cdots r_t$ is the sequence of rotations performed by an arbitrary BST algorithm for executing sequence σ. We map each box $(i, j, z) \in B$ to the first rotation r that occurs after query σ_i such that the LCA of σ_i and σ_j moves from one side of z to the other (See Figure 3.6).

It suffices to show that no two boxes map to the same rotation. Clearly, if two boxes do not overlap in time, they cannot be mapped to the same rotation. Moreover, if two boxes are oriented in different directions, they cannot map to the same rotation because a rotation of a left child over its parent can only move an LCA to the left and a rotation of a right child over its parent can only move an LCA to the right. Finally, if rotation r changes the LCA of box (i, j, z), then any box (i', j', z') whose divider is outside the horizontal interval of σ_i and σ_j will not have its corresponding LCA changed by rotation r, so box (i', j', z') cannot be mapped to r. □

Because we typically want to show lower bounds that are as large as possible, Theorem 3 motivates the question of finding a *maximum* independent box-set for a specified sequence σ. We will use $\mathrm{MIBS}(\sigma)$ to refer to the maximum size of an independent set of boxes for access sequence σ. Note that here and elsewhere, we omit mention of the set S that is associated with $\mathrm{MIBS}(\sigma)$ because even though the set of dividers depends on S, this dependence is only for our convenience in later sections, and it is straightforward to remove this dependence. For the purposes of defining and computing the optimal MIBS bound, we only need to ensure that there is at least one potential divider between each pair of successive keys that are accessed, and this can be determined directly from σ without knowledge of the full set of keys S.

Finding the value of $\mathrm{MIBS}(\sigma)$ in polynomial time is not straightforward, and we do not currently know how to achieve even a constant factor approximation to $\mathrm{MIBS}(\sigma)$, though as we will see, an $O(\lg \lg n)$-approximation follows from the fact that there are BST algorithms that provably achieve a running time that is within a factor $O(\lg \lg n)$ of optimal. Nevertheless, as suggested by Anupam Gupta, it turns out that we can solve for a maximum independent box-set if we allow fractional solutions and use linear programming [35]. In short, the relaxation of the MIBS lower bound states that we can choose

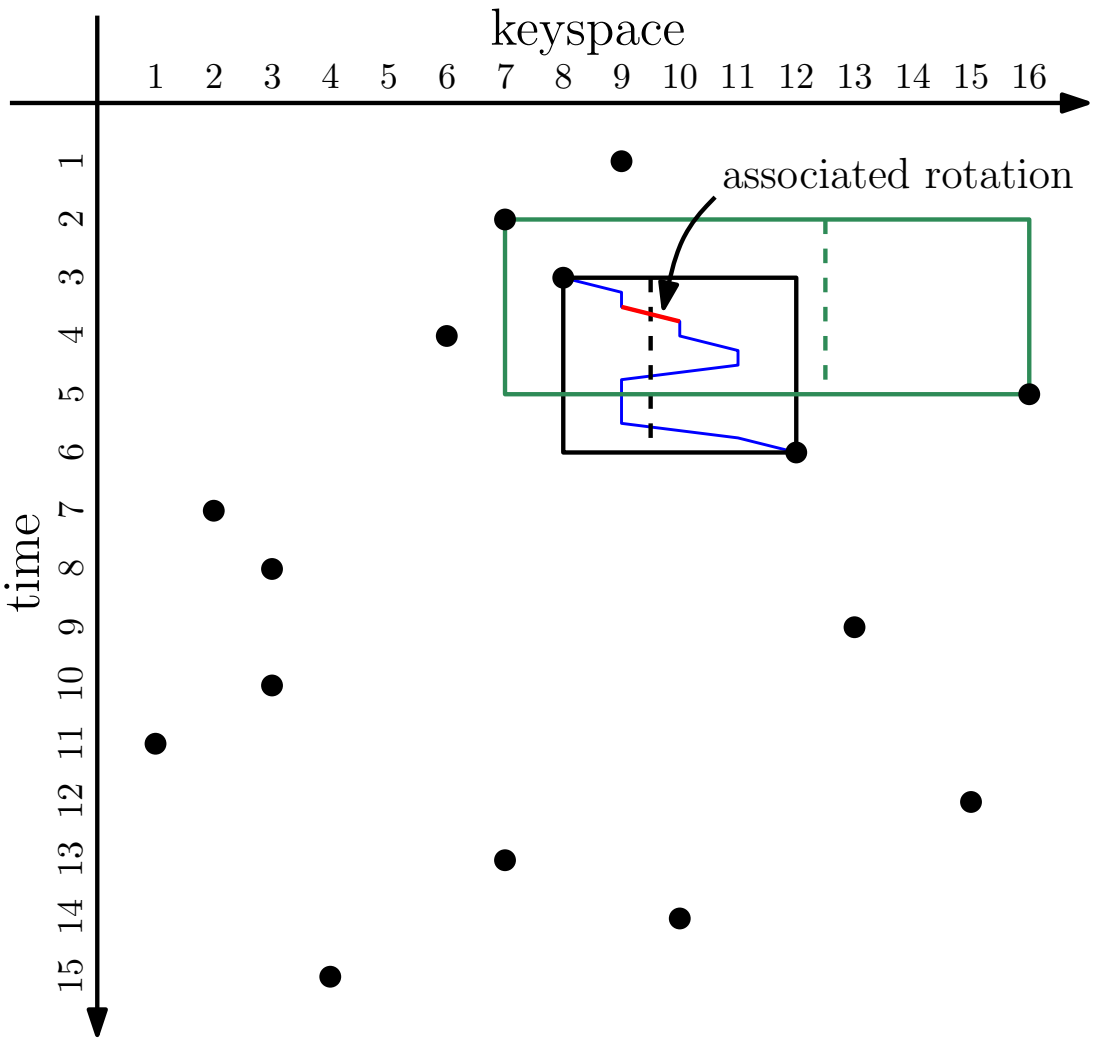


Figure 3.6: The blue path inside box $(3, 6, 9.5)$ traces the location of the LCA of 8 and 12 (i.e., σ_3 and σ_6) for some unspecified BST algorithm for serving the depicted sequence. Note that the location of the LCA of 8 and 12 only changes during a rotation of the current LCA with another node from the interval $[8, 12]$. The red segment of the path is the rotation that is associated with this box in the proofs of Theorems 3 and 4. Note that the red rotation is between two nodes that are members of the interval $[8, 12]$, so it cannot cause the LCA of the green box to change to the other side of the green divider, one way or the other. Therefore, the red rotation cannot be mapped to the green box.

each valid box with a fractional weight, and there is no conflict as long as the weighted sum of the boxes conflicting with each box is at most 1. One can imagine that it might be possible to prove that some sort of rounding scheme for an optimal fractional MIBS solution might produce an integral MIBS solution that was provably close to the fractional one. However, it turns out that this is not necessary, at least for the purpose of efficiently computing a lower bound because not only is the optimal fractional MIBS solution solvable in polynomial time, it is also a valid lower bound on $\mathrm{OPT}(\sigma)$.

To see this, we formally state the fractional MIBS lower bound as the following linear program. Let $w(b) : \mathcal{B}(\sigma) \to [0, 1]$, and for $b, b' \in \mathcal{B}(\sigma)$ let

$$c(b, b') : \mathcal{B}(\sigma)^2 \to \{0, 1\} \tag{3.1}$$

be a function such that $c(b, b') = 1$ exactly when b and b' conflict. (Note that $c(b, b) = 1$.) We seek to maximize

$$\sum_{b \in \mathcal{B}(\sigma)} w(b) \tag{3.2}$$

subject to the constraints,

$$\forall b' \in \mathcal{B}(\sigma), \sum_{b \in \mathcal{B}(\sigma)} c(b, b')w(b) \leq 1. \tag{3.3}$$

Note that if the weights $w(b)$ were constrained to be from $\{0, 1\}$ instead of $[0, 1]$, this definition would be equivalent to the integral version of the MIBS lower bound proved in Theorem 3.

We can show that the objective function in Equation 3.2 is a lower bound on the cost of $\mathrm{OPT}(\sigma)$ by proving the following theorem:

Theorem 4. *Let w be a solution to the optimization problem given in Equations 3.2 and 3.3. Any BST algorithm for σ must perform at least $\sum_{b \in \mathcal{B}(\sigma)} w(b)$ rotations.*

Proof. To prove that the sum of weights $w(b)$ is a lower bound on $\mathrm{OPT}(\sigma)$, we will create a many-to-one mapping from valid boxes to rotations of an arbitrary BST algorithm, and show that the sum of the weights of the boxes that are mapped to each rotation is at most one. We map each box (i, j, z), as in the proof of Theorem 3, to the first rotation after time i that switches the LCA of σ_i and σ_j from one side of z to the other.

Now, consider the set of boxes B_r that are mapped to an arbitrary rotation r of the BST algorithm. We know that each box in B_r overlaps the time at which r is executed because the LCA of each member $(i, j, z) \in B_r$ must switch from one side of z to the other before

query σ_j is completed. Further, we know that each box has the same direction because the change in LCA during this rotation is identical for all such boxes, so the LCA of the horizontal interval of each box must have started on the same side of its divider as all of the others. Finally, for each $(i, j, z) \in B_r$, if we choose another box $(i', j', z') \in B_r$, we know that $z' \in [\min\{\sigma_i, \sigma_j\}, \max\{\sigma_i, \sigma_j\}]$, or else the LCA of box (i, j, z) would be unchanged by rotation r. Therefore, boxes (i, j, z) and (i', j', z') conflict, and by Equation 3.3 the sum of the weights of the boxes of B_r is at most one. □

The MIBS lower bound is less structured than other lower bounds, so it is not clear how it can be used to prove a good competitive ratio for some BST algorithm, especially when it is compared to the interleave bound in this respect. Nevertheless, one possible approach to proving dynamic optimality would be to map each rotation $r_1 \cdots r_t$ of a particular BST algorithm to a box, and show that there are at least αt independent boxes for some constant $\alpha \leq 1$ from among the set of boxes chosen by $r_1 \cdots r_t$.

Also, what the MIBS bound lacks in specificity, it makes up for in its flexibility. The following sections show that Wilber's lower bounds can be stated as valid integral solutions to the MIBS bound. Therefore, both the optimal integral MIBS bound and the optimal fractional MIBS bound are at least as good as these bounds.

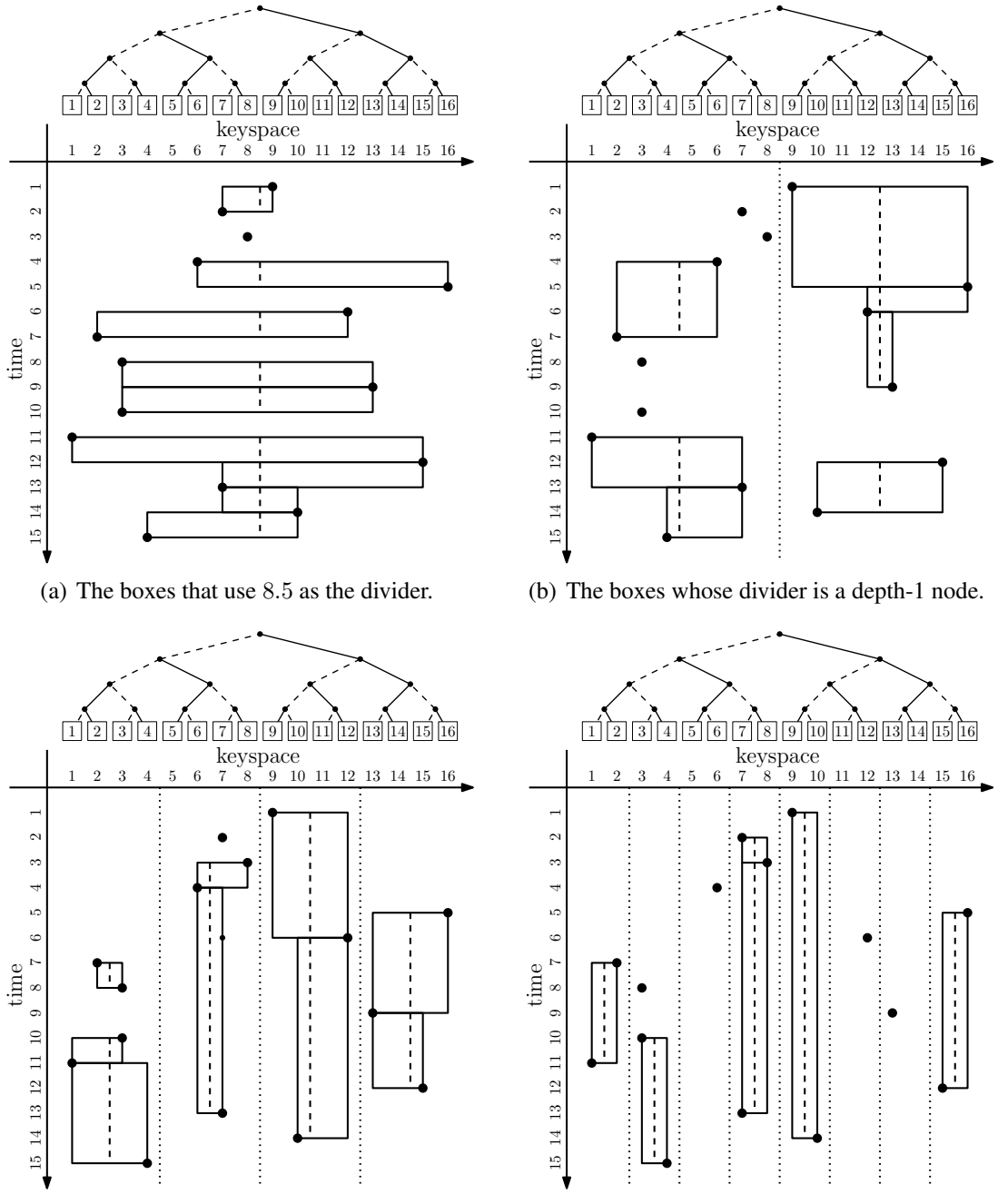
3.5.1 Proving Wilber's Lower Bounds with the MIBS Lower Bound

To show the flexibility of the independent box-set framework, we show that Wilber's first and second lower bounds are valid because they are never more than $\mathrm{MIBS}(\sigma)$ for any sequence σ. To show this, we prove both bounds using the independent box-set framework as follows.

Theorem 5. *For an arbitrary query sequence σ and lower bound tree P for Wilber's first lower bound,* $\mathrm{MIBS}(\sigma) \geq \mathrm{Wil}_1(\sigma, P)$.

Proof. For each switch of an unpreferred child v to a preferred child, create the box (i, j, z) where i is the time index of the previous access in v's sibling's subtree, j is the time index of the current access, and z is equal to the value of v's parent p. Note that any other box using p as its divider does not overlap box (i, j, z) in time, and any divider outside of p's subtree does not overlap (i, j, z). Applying this logic over all boxes from switches at the bottom of P to the top shows that no two boxes defined in this way conflict. An example of the boxes used in this proof is shown in Figure 3.7.

□



(a) The boxes that use 8.5 as the divider.

(b) The boxes whose divider is a depth-1 node.

(c) The boxes whose divider is a depth-2 node.

(d) The boxes whose divider is a depth-3 node.

Figure 3.7: The boxes that are included in the proof that $\mathrm{MIBS}(\sigma) \geq \mathrm{Wil}_1(\sigma, P)$. Note that no box overlaps a divider from a shallower level in P.

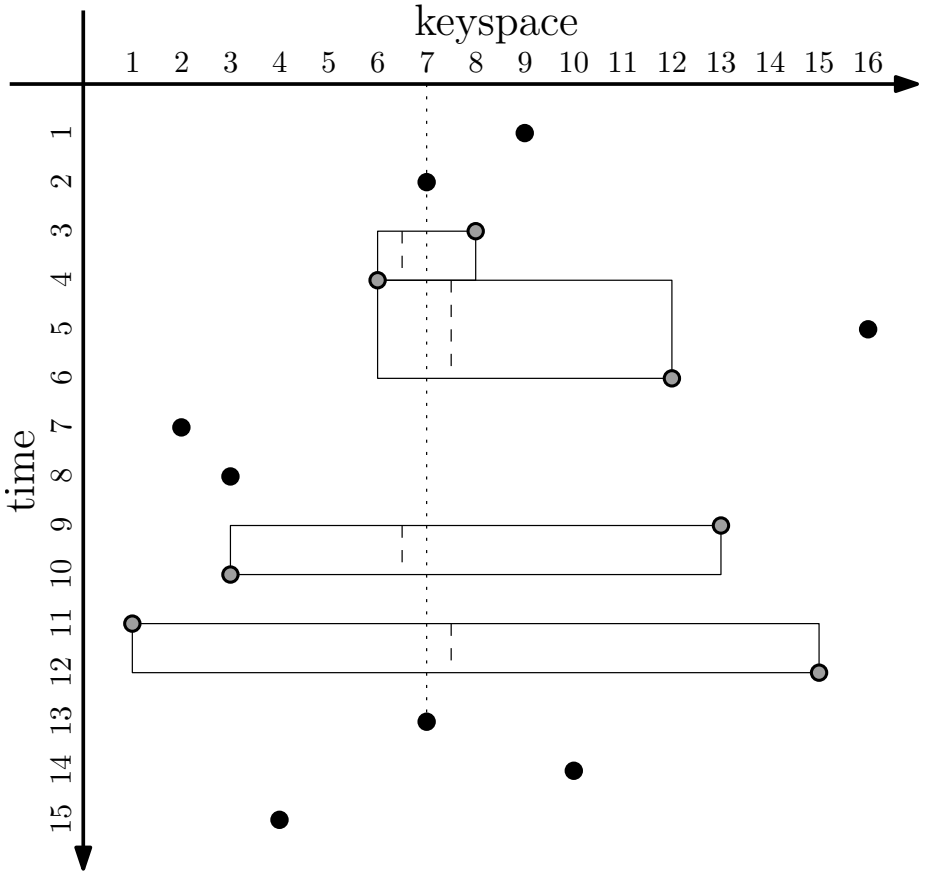


Figure 3.8: All boxes created for query σ_{13} in the proof that $\mathrm{MIBS}(\sigma) \geq \mathrm{Wil}_2(\sigma)$.

Theorem 6. *For an arbitrary query sequence σ, it is true that* $\mathrm{MIBS}(\sigma) \geq \mathrm{Wil}_2(\sigma)$.

Proof. For every query σ_j, we create the following boxes. For each crossing access $\sigma_{j'}$ of σ_j, we include the box $(j', \mathrm{succ}_{\mathrm{records}(j)}(j'), \sigma_j + \delta)$, where $\delta = 0.5$ if the box is directed left-to-right and $\delta = -0.5$ if the box is directed right-to-left. An example of all such boxes for a single access is shown in Figure 3.8.

To see that no two boxes conflict, suppose σ_j is an arbitrary query, and $\sigma_{j'}$ is an arbitrary query occurring after σ_j. Additionally, suppose that $\sigma_{j'} > \sigma_j$. Note that any right-to-left box (i'', j'', z'') formed from the crossing accesses of query $\sigma_{j'}$ does not overlap the vertical line through $\sigma_j - 0.5$ if $i'' < j$ because otherwise j would be in $\mathrm{leftRecords}(j')$, so such boxes do not conflict with the right-to-left boxes formed by the crossing accesses of σ_j. Further, note that any left-to-right box (i'', j'', z'') formed from the crossing accesses of

query $\sigma_{j'}$ does not overlap the vertical line through $\sigma_j + 0.5$ if $i'' < j$ because any member of $\text{leftRecords}(j')$ occurring at time $i'' < j$ must be at least $\sigma_j + 1$. An analogous argument applies when $\sigma_{j'} < \sigma_j$, and the case when $\sigma_{j'} = \sigma_j$ is trivial. Applying this reasoning to every query proves the theorem. □

3.6 The BST Model and the Partial-Sums Problem

One question that is important to ask whenever we are studying any model of computation is how powerful it is. Although the BST model has many nice properties, such as supporting both good adaptive performance and competitive analysis, it is not immediately clear whether the conditions imposed on a BST algorithm are overly restrictive. One way to show that this is not the case is to find a reasonably general problem, and prove that its complexity on any input is identical to the cost of the optimal BST algorithm on a corresponding sequence. In this section, we do not achieve this result, but we do make some progress that suggests that this might be true. In particular, we show that the MIBS lower bound, which seems likely to be tight for the BST model, is also a valid lower bound for the partial-sums problem with a few constraints. The importance of this is that it suggests that the BST model may not only describe a class of *data structures*, but also fully encapsulate all possible solutions to the partial-sums *problem*.

The partial-sums problem is the problem of maintaining an array of values $x_1, \ldots, x_n$ with two types of operations allowed. The operation $\text{update}(i, v)$ changes x_i's value to v, and the operation $\text{sum}(i)$ returns $\sum_{j=1}^{i} x_j$ using the current value of each x_j (actually, here we allow the "add" operation to be any associative and commutative operator). To formalize this, let $x(i, j)$ represent the value of the i^{th} array cell, for all $i \in \{1, \ldots, n\}$, after the j^{th} operation has been executed. Let $x(i, 0) = 0$ for all i. The input to the partial-sums problem is a sequence of operations $\sigma = \sigma_1 \cdots \sigma_m$ where each σ_j is either of the form $\text{update}(i, v)$ or $\text{sum}(i)$. We define $x(i, j) = v$ if σ_j is $\text{update}(i, v)$; otherwise, we let $x(i, j) = x(i, j - 1)$. The output is a sequence of values $y_1 \cdots y_m$, one for each operation, such that if σ_j is an update operation, then $y_j = 0$, and if σ_j is $\text{sum}(i)$, then the output is $\sum_{i'=1}^{i} x(i', j)$.

The partial-sums problem clearly can be solved by any BST algorithm that stores the keys $\{1, \ldots, n\}$ in its tree T, stores each array value $x(i, j)$ in the node $i \in T$ at time j, and maintains the sum of the array values that appear in i's subtree in another field of i. To see this, note that if we rotate i to the root, it costs only $O(1)$ additional time to execute an $\text{update}(i, v)$ or $\text{sum}(i)$ operation using only only add instructions. Thus, it is clear that the optimum BST provides an upper bound on the cost to execute the sequence σ. On the other hand, Pătraşcu and Demaine [50, 51] showed that in the worst case, one can do

no better than using a BST. They proved a bound of $\Omega(\lg n)$ for the partial-sums problem by showing that there were instances of the partial-sums problem that required $\Omega(\lg n)$ memory probes to solve the instance. Pătraşcu further suggested that Wilber's first lower bound, with the time index representing the keys and array index representing time, was a lower bound on each *instance* of the partial-sums problem [49]. (Although this lower bound is instance-specific, it still requires randomization over the update values.)

In this section, we take the first steps toward proving something even stronger, that $\mathrm{MIBS}(\sigma)$ is a valid lower bound for the partial-sums problem if the array indices are mapped to keys in a BST. Showing that the MIBS bound is a valid lower bound for the partial-sums problem provides strong evidence that each instance of the partial-sums problem can be solved no faster than how it is solved using a BST because the MIBS lower bound is an excellent candidate for a tight lower bound in the BST model.

There are a few caveats worth stressing about this new result, however. First, we will be using a weaker model of computation than the cell-probe model that was used in [50, 51]. In particular, we will be using the *set-sum* model of computation that charges only for instructions of the form $\mathrm{add}(c_1, c_2)$ rather than charging for all memory accesses as in the cell-probe model. Aside from only charging for arithmetic operations, the set-sum model places constraints on how an algorithm can compute the output v_j of a $\mathrm{sum}(i)$ operation. Define the set-sum $\mathrm{sum}(S, j)$ to be $\sum_{i=S} x(i, j)$. We define a memory cell c to *store* the set-sum $\mathrm{sum}(S, j)$ exactly when one of the following two conditions holds:

- Cell c stores $\mathrm{sum}(S, j-1)$ and σ_j is not an operation of the form $\mathrm{update}(i, v)$ such that $i \in S$.
- Cell c was written by an instruction $\mathrm{add}(c_1, c_2)$, where cell c_1 stores $\mathrm{sum}(S_1, j)$, cell c_2 stores $\mathrm{sum}(S_2, j)$, the sets S_1 and S_2 do not intersect, and the union of S_1 and S_2 is S.

Informally, the above constraints say that the only arithmetic computation an algorithm is permitted is the addition of two non-overlapping sums. The use of inverses, double-counting, or any other fancy arithmetic tricks is forbidden. This model of computation is significantly weaker than the cell-probe model, which is essentially used to prove information-theoretic lower bounds on what running times are achievable in [50, 51].

Nevertheless, this model of computation is not as weak as it may superficially appear. First, it permits augmented BSTs because augmented BSTs always only store valid set-sums and only need addition to perform updates and rotations. Second, one might argue that this model is so restrictive that the only algorithms that are permitted essentially *are* BSTs, so that proving a lower bound in the set-sum model is trivial when such a bound

has already been shown in the BST model. This complaint is refuted by the observation that the set-sum model allows not just BSTs, but also multiple BSTs. Thus, it is not at all clear how to transform an arbitrary sequence of set-sum operations into a BST because this would require, as a special case, the ability to combine two arbitrary BST algorithms into one BST whose running time was the minimum of the two original BST algorithms, and even this special case seems difficult. Consider, for example, what would be required to combine Tango [21] and splay trees [55] into a single BST.

With the above constraints and definitions, we are ready to prove the main theorem for this section.

Theorem 7. *Let $\sigma = \sigma_1 \cdots \sigma_m$ be a sequence of partial-sums operations, and moreover let $B \subseteq \mathcal{B}(\sigma)$ represent a set of valid, non-conflicting boxes for the MIBS bound that contains only left-to-right boxes (i, j, z) where σ_i is an* update *operation, σ_j is a* sum *operation, and the horizontal coordinate of each point is defined by the array index of the corresponding instruction. The number of* add *instructions required to serve σ in the set-sum model of computation is at least $|B|$.*

Proof. Our approach will be to map each box to an add instruction and show that these operations are distinct. This mapping will be analogous to the mapping of boxes to rotations that was used in the proofs of Theorems 3 and 4. Box (i, j, z) is mapped to the earliest instruction $c = \mathrm{add}(c_1, c_2)$ that is executed after time i such that c stores $\mathrm{sum}(S, i)$, for which the following three conditions are true. First, $x(\sigma_i, i) \in S$, where σ_i here represents the index of the update operation. Second, for some $k \in [z, \sigma_j]$, where σ_j here represents the index of the sum operation, it is true that $x(k, i) \in S$. Third, for no $k' \in (\sigma_j, n]$ does S include $x(k', i')$, for any i'.

First, note that this instruction occurs during the time interval $(i, j]$ because at the latest, such an instruction must be executed prior to the sum operation σ_j. Therefore, two boxes that do not intersect in time cannot map to the same instruction. Second, note that both operands of the instruction to which box (i, j, z) is mapped must contain some value that is in the horizontal range of the box. Further, note that the one operand of this instruction must not contain any values to the right of the divider z. Therefore, if boxes (i, j, z) and (i', j', z') intersect in time and $z < \sigma_{i'}$, then the add instruction mapped to by (i, j, z) contains one operand that does not intersect box (i', j', z')'s horizontal range so they cannot be mapped to the same instruction. On the other hand, if boxes (i, j, z) and (i', j', z') intersect in time and $z > \sigma_{j'}$, then both of the operands of the instruction that box (i', j', z') maps to must not contain any value to the right of z, so box (i, j, z) cannot be mapped to this instruction. These cases suffice to prove the theorem. □

Note that as for Theorem 3, we can relax the integral version of Theorem 7, and the fractional bound is also a lower bound on the cost of an algorithm for executing the sequence of partial-sums operations. Further, we could include right-to-left boxes in the bound if the sum operation were required to additionally return the sum of the current array values from the specified index to the end of the array, so that it returned both a "prefix-sum" and a "suffix-sum".

It is also worth noting that if updates and sums were batched together, as in a sequence of operations that included n update operations followed by n sum operations, such a sequence could be served in linear time regardless of ordering among the update operations despite the fact that using a BST to serve some of such sequences would require $\Omega(n \lg n)$ time. A way to get around this shortcoming would be to assume such batched updates and sums could be reordered into a BST-friendly order, such as sorted index order, but this is pure speculation.

Chapter 4

Adaptive Binary Search Bounds

In Chapter 3, we showed a variety of *lower bounds* for the BST model, and in so doing we gained some insights into what operations must be performed by a BST. In this chapter, we describe insights from the other direction by covering various *upper bounds* for search data structures, mostly BSTs. First, we will discuss competitive data structures, and summarize work related to developing a competitive BST. Then, we will consider other kinds of adaptive bounds that have a more clear intuitive meaning than competitiveness, but that are not necessarily closely related to competitiveness in any particular model of computation.

4.1 Competitive Search in a BST

The richest adaptive search property that one can prove for a BST, or any other data structure, is *competitiveness*. When a family of data structures exhibits the possibility of adapting to patterns in the query sequence, a natural question to ask is "which is the best"? In the offline setting, one could use brute force and try out all algorithms for each input to determine which one was fastest for that particular input. Obviously, such a brute force solution has two drawbacks. First, this requires an inordinate amount of time. Second, when a data structure is actually used, it does not have knowledge of the future.

The concept of *dynamic optimality* solves this by hypothesizing that there exists an online data structure whose cost on each input is within a constant factor of the best cost possible, even without the benefit of hindsight or boundless computation. The possible existence of a dynamically optimal BST algorithm was suggested in [55], when Sleator and Tarjan posited the *Dynamic Optimality Conjecture*, which suggested that their extremely simple splay algorithm was dynamically optimal in the BST model.

At first, only special cases of dynamic optimality were proven for splay trees. Eventually, after proving the dynamic optimality of splay trees began to seem increasingly difficult, Demaine *et al.* suggested proving a small but non-constant competitive factor for a different BST algorithm called Tango, which they showed to be $O(\lg \lg n)$-competitive [21, 22].

The idea behind Tango can be understood as follows. Wilber's first lower bound, as suggested in Chapter 3, can be recast so that the lower bound tree is, itself, a valid BST for the set of keys S that the query sequence σ accesses (See Figure 3.3). Note that we could show that this tree was $O(1)$-competitive if we could show that we only traverse $O(1)$ preferred edges for every unpreferred edge we traverse. Thus, to prove the trivial statement that a balanced BST is $O(1)$-competitive on uniformly random access sequences, we need only to note that the probability of a switch at each node on the access path is at least a constant. This avoids the intermediate step of showing that both the information-theoretic lower bound and the algorithm itself cost $\Theta(\lg n)$.

Of course, in general, a query sequence containing significant nonrandomness may traverse at least $\lg n$ preferred edges for each unpreferred edge encountered. This suggests the idea of restructuring long preferred paths so that the BST algorithm does not spend as much time in between unpreferred edges. Demaine *et al.* achieved this in Tango by rotating each preferred path into the shape of a red-black tree. This showed that Tango was $O(\lg \lg n)$-competitive, but had the unfortunate side effect of burdening Tango with a worst-case performance bound of $O(\lg n \lg \lg n)$, which was tight for cases in which $\Omega(\lg n)$ switches were performed. (They would later suggest a modification to Tango to improve the worst-case running time to $O(\lg n)$ in the journal version of the paper [22].)

To remedy this shortcoming of Tango, Wang, Sleator, and I introduced the multi-splay algorithm [61], which essentially replaced the red-black trees of Tango with splay trees, creating a data structure that was similar to link-cut trees. This seemingly minor change enabled multi-splay trees to provably achieve not only $O(\lg \lg n)$-competitiveness, but also $O(\lg n)$ amortized running time, the working set bound (which subsumed the $O(\lg n)$ amortized bound), and the deque property, which was not even proven for splay trees [25, 60]. In addition, it was shown how multi-splay trees could be made dynamic while still maintaining $O(\lg \lg n)$-competitiveness by introducing a dynamic BST model and modifying the interleave bound to allow rotations on the lower bound tree so that the lower bound tree's balance could be restored after insertions and deletions as suggested in Section 3.2.

Other later work on BST competitiveness all involved other variations on the original idea in Tango. These data structures include chain-splay trees [32], which are similar to multi-splay trees and independently discovered; Poketree, a non-BST whose running

time is worst-case $O(\lg n)$ and $O(\lg \lg n)$-competitive to the optimal BST [42]; and the zipper-tree, a BST that achieves the same guarantee as Poketree [10].

4.2 Other Kinds of Competitiveness

In addition to competitiveness in the classic BST model, there are a couple of notable variations. Lucas [44] and Munro [46] independently devised an offline BST algorithm, which we call "Greedy Future" (as named in [20]), that used its knowledge of the future to rotate the access path into a tree that was heap ordered according to how recently each node and child-subtree of the access path would be queried. Roughly, the access path was rotated so that the next node to be accessed would be as shallow as possible in the tree, and this process was repeated recursively on each side of the next query's region of the access path. Remarkably, Demaine *et al.* showed how to create an online algorithm whose running time is within a constant factor of this greedy offline algorithm [20].

Intuitively, Greedy Future seems like it should be a good candidate for dynamic optimality even though it still has not been shown that its cost is amortized $O(\lg n)$. The reason this greedy algorithm seems like it should be efficient is that if we have to touch every node on the access path, why not rotate the path greedily so as to minimize the number of nodes blocking the search paths of the accesses that will be soonest to occur? Greedy Future only really seems to be limited by the fact that it can only rotate the access path, because in general, the exact optimum BST algorithm requires the BST to rotate nodes that form a tree starting at the root, and Munro gave an example for which this occurs [46]. This last observation raises an important point, however.

One might hypothesize that the only reason that Greedy Future failed to achieve exact optimality was because it was forbidden from rotating any node that was off the access path. However, at least when only considering such greedy algorithms, the fact that it is restricted to rotating the access path is the only constraint that keeps its cost from being vastly higher than optimal. To see this, suppose that after each access the BST is restructured so that depths obey heap order according to the time of next access. This corresponds to the time-reversal of the rotate-to-root heuristic which is known to suffer $\Theta(n)$ cost per operation for sequential access. Thus, the unconstrained greedy algorithm is worse than the optimal algorithm by a factor of $\Theta(n)$ on some sequences.

Another notable contribution to work on competitive BSTs is the algorithm of Blum *et al.* [9]. They introduced an online exponential time algorithm that achieved what they called *dynamic search optimality*, which they defined to mean that their algorithm had a cost that was within a constant factor of optimal if their algorithm was granted free

rotations and computation time so that it only had to pay for the length of the search path. One way of viewing this result is that it showed how to optimally (for a BST, to within a constant factor, given access to unbounded computing resources) compress a stream of keys that appeared online, one by one. This was an important result because it showed that dynamic optimality was achievable, at least in principle, from an information-theoretic standpoint, by an online algorithm.

4.3 Exploiting Spatial and Temporal Locality

Although competitive bounds are powerful, they say little about what the actual performance of the algorithm is, other than proving that it is almost as good as possible. An alternative approach is to show that an algorithm achieves good performance as a function of some kind of nonrandomness in the input.

For example, one might want to show that an algorithm performs well when the access distribution is highly skewed, or whenever recently accessed elements are likely to be queried during each access. The *working set bound* characterizes this behavior as follows. For a query sequence $\sigma = \sigma_1 \cdots \sigma_m$, define $w(x, j)$ to be, at time j, the number of distinct elements including x that have been queried since the previous query to x, or n if no such previous query exists. The working set bound states that the cost to execute the sequence σ is $O(\lg w(x, j))$ for each query σ_j.

Splay trees have been shown to satisfy the working set bound using amortization [55], and layered working-set trees later showed that it was possible to achieve the working set bound using worst-case analysis in the BST model [11, 12]. Iacono showed that the working set bound was equivalent to *key-independent optimality*, which he introduced as a term for the expected running time of the optimal algorithm on each query sequence σ if a random permutation was applied to the keys [39]. In a result that was related to the working set bound for splay trees, Georgakopoulos introduced a variation on the access lemma for splay trees, which was originally used to prove the working set bound for splay trees, called the *reweighing lemma*. This result extended the access lemma for splay trees to allow the weights of nodes to be reweighed during the course of the access sequence, even if they had not been accessed [31].

On the other hand, one might want to exploit spatial patterns in the query sequence. There are two primary types of bounds that capture such performance. First, along with each query σ_j, the user of a data structure could specify a *finger* f_j that was believed to be close to σ_j, and achieve a running time that varies in accordance with how accurate these guesses are. This property is called *finger search*, and the running time bound of

a data structure that achieves the finger search bound is $O(\lg(|f_j - \sigma_j| + 2))$, assuming the keys are $\{1, \ldots, n\}$ (otherwise, the cost is the logarithm of the difference in ranks). A variety of data structures achieve the finger search bound in the comparison model, including [14, 33, 15], and even better finger search bounds can be achieved in the RAM model [2, 3, 41]. As noted in Section 2.1.2, it is not possible for a BST to achieve the finger search bound.

Strictly speaking, a finger search data structure does not directly exploit spatial locality in the access *sequence*, but it is trivial to use a finger search data structure to achieve this. For example, it is straightforward for a finger search data structure to achieve the *dynamic finger* bound, which states that the cost of each query is $O(\lg(|\sigma_j - \sigma_{j-1}| + 2))$, assuming $j > 1$. The dynamic finger bound *is* achievable in the BST model, and was shown to hold for splay trees in [19, 18]. Further, [8] showed how to achieve the dynamic finger bound for any balanced BST by adding a small auxiliary data structure called a *hand*.

Aside from the dynamic finger bound, there are a couple of additional noteworthy bounds that BSTs can achieve. It is clear that an optimal BST can operate like a linked list, and be either scanned sequentially (e.g., $\sigma = 1, 2, \ldots, n$) or used as a deque using only $O(1)$ amortized time per operation. Even though the scanning theorem, which states that splay trees pay $O(n)$ cost to execute the sequence $1, 2, \ldots, n$, is an extremely simple theorem, it requires non-trivial proofs [57, 56, 28]. Further, despite intense effort, splay trees have only been proved to have the deque property to within a factor of $\alpha(n)$ by Sundar [56], and to within a slightly better factor of $\alpha^*(n)$ by Pettie [47].

4.4 The Unified Bound

Although the working set bound and the dynamic finger bound show that data structures such as splay trees exhibit good performance whenever locality exists in either time *or* space, they do not say anything about performance on sequences that exhibit a hybrid of *both* kinds of locality. Consider the sequence $\sigma = 1, \frac{n}{2} + 1, 2, \frac{n}{2} + 2, \ldots, \frac{n}{2}, n$. This sequence exhibits a great amount of structure, and it is easy to see that there is a BST algorithm that serves this sequence in $O(n)$ total time, but both the working set bound and the dynamic finger bound provide a bound of only $O(n \lg n)$ for this sequence. This could be remedied by devising a new dynamic "fingers" bound that allowed two fingers, but the sequence $\sigma = 1, \frac{n}{3} + 1, \frac{2n}{3} + 1, 2, \ldots, n$ would serve as a bad example for a two-finger dynamic finger bound. Similarly, using any constant number of fingers has an analogous counterexample.

Motivated by this observation, Iacono introduced the Unified Bound, which was a

more robust generalization of the working set bound and the dynamic finger bound than a simple dynamic finger bound with multiple fingers [37]. Roughly, a data structure that satisfies the Unified Bound has good performance for sequences of operations in which most accesses are likely to be near a recently accessed element. More formally, for the query sequence $\sigma = \sigma_1 \cdots \sigma_m$, where each query σ_j is to a member of $\{1, \ldots, n\}$, the Unified Bound can be defined as follows:

$$\mathrm{UB}(\sigma) = \sum_{j=1}^{m} \min_{j'<j} \lg(w(\sigma_{j'}, j) + |\sigma_{j'} - \sigma_j|). \tag{4.1}$$

To achieve a running time of $O(m + \mathrm{UB}(\sigma))$, Iacono introduced a data structure called the Unified Structure. The Unified Structure did not require amortization to achieve this bound, and the Unified Structure was later improved by Bădoiu *et al.* to be simpler and allow insertion and deletion [16]. The Unified Structure was comparison-based but did not adhere to the BST model. Thus, in addition to leaving open questions regarding how powerful the BST model was, it was not clear, for example, how to achieve the Unified Bound while keeping track of aggregate information on subsets of elements as can be done with augmented BSTs.

These unresolved issues motivated the question of whether a BST algorithm exists that achieves the Unified Bound. It is worth stressing that achieving this goal contrasts with the separate pursuit of a provably dynamically optimal BST algorithm in that it is possible for a data structure that achieves the Unified Bound to have the trivial competitive ratio of $\Theta(\lg n)$ to an optimal BST algorithm, as will be shown in Section 4.5.

Conversely, prior to the work in Chapters 5 and 6, even if a dynamically optimal BST algorithm had been found, it would not have been clear whether it satisfied the Unified Bound to within any factor that was $o(\lg n)$ since dynamic optimality by itself says nothing about actual formulaic bounds, and prior to the work in Chapters 5 and 6, no competitive factor better than $O(\lg n)$ was known for the cost of the optimal BST algorithm in comparison to the Unified Bound.

4.5 Beyond the Unified Bound

Although the Unified Bound is a robust bound that generalizes the dynamic finger bound by allowing an arbitrary number of fingers with a working set bound penalty for using stale fingers, there are still counterexamples that show that the optimal BST can execute some sequences of queries faster than the Unified Bound by a factor of $\Omega(\lg n)$. Consider

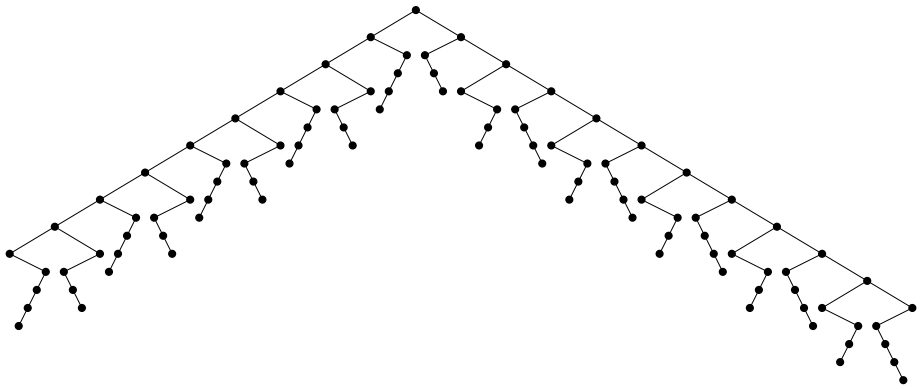


Figure 4.1: The state of an $O(n)$-time BST algorithm for serving a sequence of the form shown in Equation 4.2.

the following sequence σ^*:

$$\begin{aligned}
&1, n^{1/2}+1, 2n^{1/2}+1, 3n^{1/2}+1, \ldots, (n^{1/2}-1)n^{1/2}+1,\\
&2, n^{1/2}+2, 2n^{1/2}+2, 3n^{1/2}+2, \ldots, (n^{1/2}-1)n^{1/2}+2,\\
&3, n^{1/2}+3, 2n^{1/2}+3, 3n^{1/2}+3, \ldots, (n^{1/2}-1)n^{1/2}+3,\\
&\vdots\\
&n^{1/2}, n^{1/2}+n^{1/2}, 2n^{1/2}+n^{1/2}, 3n^{1/2}+n^{1/2}, \ldots, n.
\end{aligned} \tag{4.2}$$

Essentially, σ^* is the sequence that results if the keys are laid out on a square grid left-to-right, top-to-bottom, and then queried top-to-bottom, left-to-right. The Unified Bound cost for each query is $\Omega(\lg n)$, but nevertheless a BST can easily handle this sequence at a total cost of $O(n)$. An example of the state of such a BST is shown in Figure 4.1.

The intuition for how this BST works is the following. First, observe that the access sequence $\sigma = 1, 2, \ldots, 10$ can be served by a BST that behaves much like a 10-inch rope slinking over a nail on a wall. The sequence starts with the first inch of the left end of the rope hanging over the nail, and each access is executed by sliding the rope one inch to the left. Second, note that if we attach 10 pins spaced one inch apart on this rope, and slink a 10-inch piece of string over each of these pins, we can create a physical version of a BST for serving a sequence of the form in Equation 4.2 at a cost of just $O(1)$ per operation (assuming "10" is allowed to be an arbitrary positive integer). Each access typically corresponds to slinking the main rope one inch to the left, and then slinking the string nearest to the nail one inch to the left over its pin. Occasionally, a "carriage return" must be executed on the main rope by slinking it all the way to the right for the next access

so that the leftmost inch of the main rope returns to being situated directly over the nail.

Notice that this type of sequence still appears to have spatial and temporal locality even though it causes the Unified Bound to fail to give a tight bound for the optimal BST. In particular, it is essentially $\sqrt{n}$ interleaved sequential access sequences. This suggests an extension to the Unified Bound that allows the keyspace to be partitioned into a two-level hierarchy so that the cost of each query is broken down into the cost of finding the contiguous subset of keys in which a query resides, and then finding the queried element within that set. This bound could be achieved for a fixed two-level hierarchy, for example, by storing the minimum and maximum element of each subset in one data structure that achieves the Unified Bound, such as the cache-splay trees of Chapter 6, and then storing each subset separately in its own such data structure. Note that the fact that the optimal BST can achieve such a bound implies that the $O(\lg \lg n)$-competitive BSTs have good performance on sequences for which this two-level Unified Bound has a low value.

Unfortunately, even this two-level hierarchical extension of the Unified Bound is loose by a factor of $\Omega(\lg n)$ when compared to the optimal BST cost for some sequences. To see this, note that the example in Equation 4.2 can be extended one more level as follows. Number a cube with the elements of S left-to-right, top-to-bottom, front-to-back, and generate the query sequence by traversing the cube front-to-back, top-to-bottom, left-to-right. The two-level Unified Bound would specify a cost of $O(n \lg n)$ when the optimal BST could access this sequence in $O(n)$ time. Still, it would constitute significant progress in BST analysis to prove that a simple BST algorithm like splay trees met such a bound (indeed, proving that splay trees satisfy even the Unified Bound would be major progress).

However, as a first step, we could prove something simpler. Suppose we write the elements $\{1, \ldots, n\}$ using an arbitrary base b, and suppose that $n = b^k$ for some k. Define the base-b digit-reversal permutation to be the permutation $\sigma(b) = \sigma_1 \cdots \sigma_m$, where $\sigma_{d_1 d_2 \ldots d_k} = d_k d_{k-1} \ldots d_1$. We define the base-$b$ digit reversal conjecture as follows.

Conjecture 1. *The cost of splay trees on a base-b digit-reversal permutation is $O(n \cdot \frac{\lg n}{\lg b})$.*

Note that Conjecture 1 is a generalization of the scanning theorem (base-n) and the balance theorem applied to the bit-reversal permutation (base-2). Further, note that Conjecture 1 is a special case of the Dynamic Optimality Conjecture of [55] because it is easy to prove that the optimal BST algorithm for such a sequence costs $\Theta(n \cdot \frac{\lg n}{\lg b})$. The upper bound portion of this claim follows from the extension of the "hanging rope with pins" algorithm to $\frac{\lg n}{\lg b}$ levels (i.e., a nail with a hanging rope with pins with hanging strings with needles with hanging threads, and so on). The lower bound portion of this claim follows by using Wilber's first lower bound with a balanced lower bound tree. The same logic that Wilber applied to the bit-reversal permutation [62] shows that there are $\Omega(n)$ switches of

an unpreferred child to a preferred child in every contiguous set of $\lg b$ levels of the lower bound tree, so $\Omega(n \cdot \frac{\lg n}{\lg b})$ switches occur in total.

4.6 Adaptive Search in Higher Dimensions

Although this thesis focuses mainly on one-dimensional search and the binary search tree model, it is worth briefly discussing the pursuit of input-sensitive bounds for higher-dimensional versions of the search problem.

Ideally one would like to have a model of computation similar to the BST model for higher-dimensional search. This would facilitate the development of a competitive algorithms for higher-dimensional search. Unfortunately, it seems difficult to define an analog of the BST model for higher-dimensional search that permits reasonably good bounds. Without a reasonable computational model from which to choose search algorithms, we cannot even begin to design an algorithm that is competitive in any meaningful way, even for two-dimensional search. Also, as dimension increases, the curse of dimensionality makes it difficult to find high quality algorithms, even when we are only concerned with the performance in the worst case.

Nevertheless, several adaptive search algorithms have been developed for search in a higher number of dimensions. Demonstrating the possibility of *temporal* adaptivity for the two-dimensional planar point location problem, Iacono [38] and Arya *et al.* independently showed how to achieve the entropy bound of $O(\lg(1/p(x)))$ for each query to a region x, where $p(x)$ is the probability that x is queried. Demonstrating the possibility of *spatial* adaptivity for two-dimensional search, Iacono and Langerman showed how to achieve a two-dimensional version of the dynamic finger bound for both planar point location [40] and a more restrictive version of two-dimensional search that required all queries to be successful searches to stored points [23]. Finally, Sheehy, Sleator, Woo, and I [24] showed how to achieve spatial adaptivity for approximate nearest neighbor search in an arbitrary but fixed dimension by modifying and carefully analyzing the space-filling curve technique of [43, 17].

Chapter 5

Skip-Splay Trees

This chapter discusses a new BST algorithm called skip-splay that was originally introduced in [27]. The skip-splay algorithm has three important qualities. First, it conforms to the BST model and has a running time of $O(m \lg \lg n + \mathrm{UB}(\sigma))$, where $\mathrm{UB}(\sigma)$ is Iacono's Unified Bound, which is defined in Equation 4.1 of Chapter 4. Thus, skip-splay trees nearly achieve the same robust performance as the Unified Structure [37, 16] for sequences of queries in which most queries are likely to be near a recent query. Skip-splay trees achieve this despite being restricted to the BST model, unlike the Unified Structure, which uses arbitrary pointers and cannot, for example, be used to solve the partial-sums problem. Second, the skip-splay algorithm is very simple in comparison to the Unified Structure. The majority of the complexity of skip-splay trees resides in the analysis of skip-splaying, not in the design of the algorithm itself. Finally, skip-splaying is almost identical to splaying, which suggests that a similar analysis, in combination with new insight, might be used to prove that splay trees satisfy the Unified Bound, at least to within some nontrivial multiplicative factor or additive term. The Unified Conjecture of Iacono [37] originally suggested that splay trees might achieve the Unified Bound, and skip-splay trees show that with just a small amount of additional structure added to the splay algorithm, this conjecture can be proved to within an additive $O(\lg \lg n)$ term per query using a significantly simpler proof than the simplest known, but extremely long and complicated, proof of the less general dynamic finger bound for splay trees [19, 18].

5.1 The Skip-Splay Algorithm

We assume for simplicity that a skip-splay tree T stores all elements of $\{1, \ldots, n\}$ where $n = 2^{2^{k-1}} - 1$ for some positive integer k, and that T is initially perfectly balanced. We

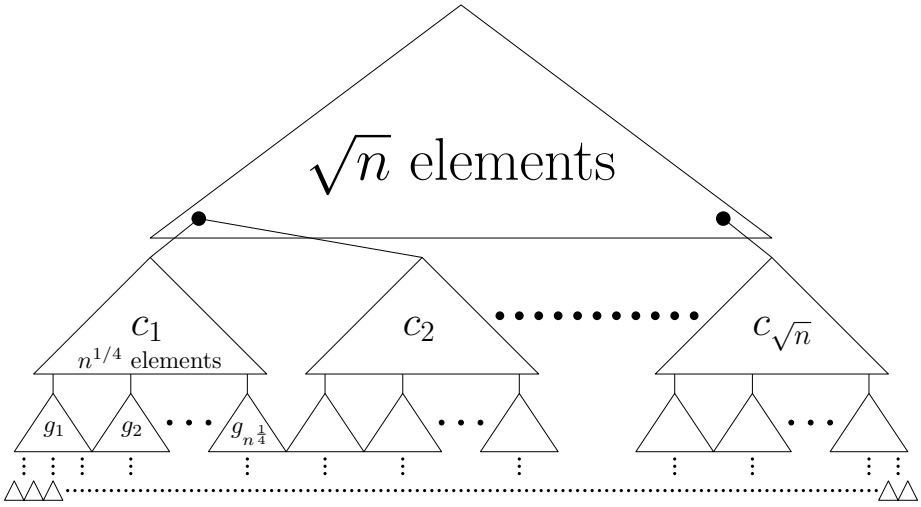


Figure 5.1: A schematic of a skip-splay tree. The size of a splay tree at each level is the square of the size of a splay tree one level deeper in the skip-splay tree, and the splay trees at the deepest level have a constant number of nodes. Note that there are $\sqrt{n}$ "child trees" of the top-level splay tree, and there are $n^{1/4}$ "grandchild trees" of the top level splay tree that are children of its first child c_1. The number of child trees of a tree is roughly equal to the number of nodes in the tree.

mark as a splay tree root every node whose height (starting at a height of 1 for the leaves) is 2^i for $i \in \{0, \ldots, k-1\}$.[1] Note that the set of all of these resulting splay trees partitions the elements of T. A schematic of what this decomposition of splay trees looks like is shown in Figure 5.1, and the actual initial structure of a small skip-splay tree is shown in Figure 5.2.

The following definitions will help us describe the algorithm more clearly:

1. Let T_i be the set of all keys x whose path to the root of T contains at most i root nodes, including x itself if x is marked as a root.

2. Define level i of T to be the set of keys x whose path to the root contains *exactly* i root nodes. We will sometimes use the adjective "level-i" to refer to objects associated with level i in some way.

3. Let $\mathrm{tree}(x)$ be the splay tree that contains x. Also, $\mathrm{tree}(x)$ can represent the set of elements in $\mathrm{tree}(x)$.

[1] If we allow the ratio between the initial heights of successive roots to vary, we can achieve a parameterized running time bound. We use a ratio of 2 for simplicity.

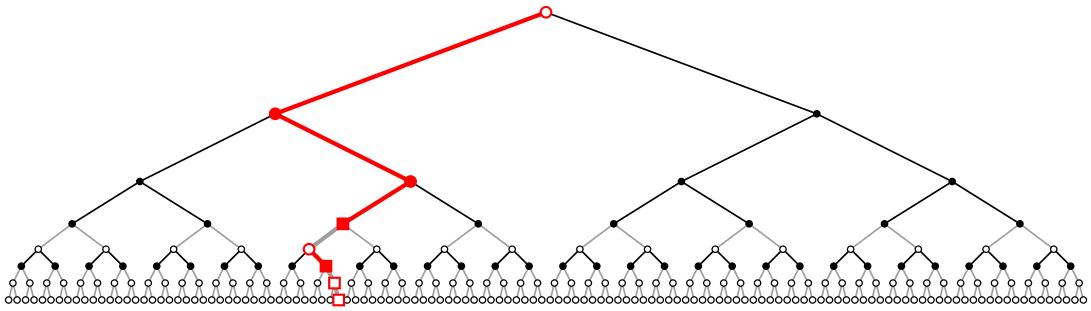


Figure 5.2: An example of a four-level skip-splay tree T at the beginning of a query sequence. The nodes filled with white are the roots of the splay trees that make up T, and the gray edges are never rotated. If the bottom element of the bold red path is queried, then each of the boxed nodes is splayed to the root of its splay tree.

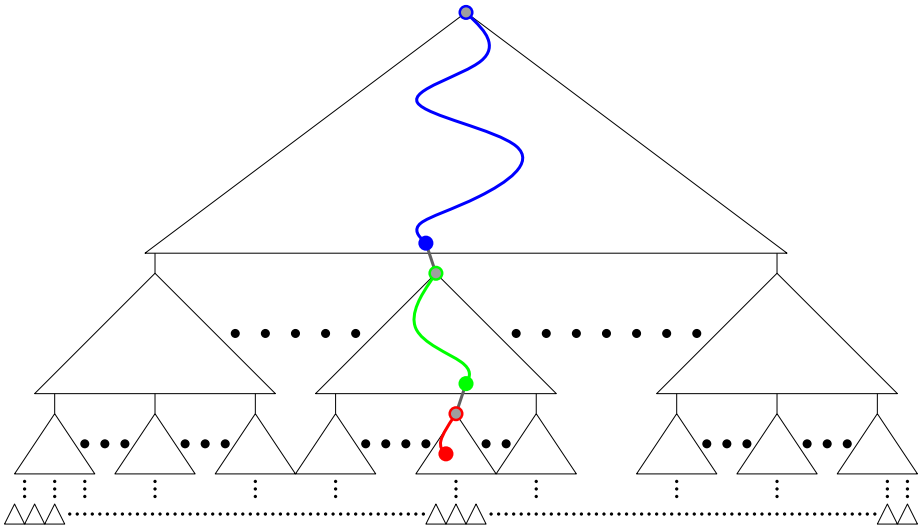
We assume that all operations are queries, and we use $\sigma = \sigma_1 \cdots \sigma_m$ to denote the sequence of queries. To query an element σ_j, we first perform binary search through T to locate σ_j. Then, we splay σ_j to the root of $\mathrm{tree}(\sigma_j)$, and transfer the relevant root marker to σ_j. If we are at the root of T, we terminate, else we "skip" to σ_j's new parent x, and repeat this process by splaying x to the root of $\mathrm{tree}(x)$. The cost of a query is defined to be the number of nodes on the access path to σ_j.[2] Figure 5.2 shows an example of what a skip-splay tree looks like at the beginning of an access sequence, and depicts how a query is performed. Figure 5.3 gives a schematic of what a skip-splay tree looks like before and after a query.

Intuitively, skip-splaying is nearly competitive to the Unified Bound because if the currently queried element σ_j is near to a recently queried element σ_f, then many of the elements that are splayed while querying σ_j are likely to be the same as the ones that were splayed when σ_f was queried. Therefore, by the working set bound for splay trees, these splays should be fairly cheap. The analysis in Section 5.2 formalizes this intuition.

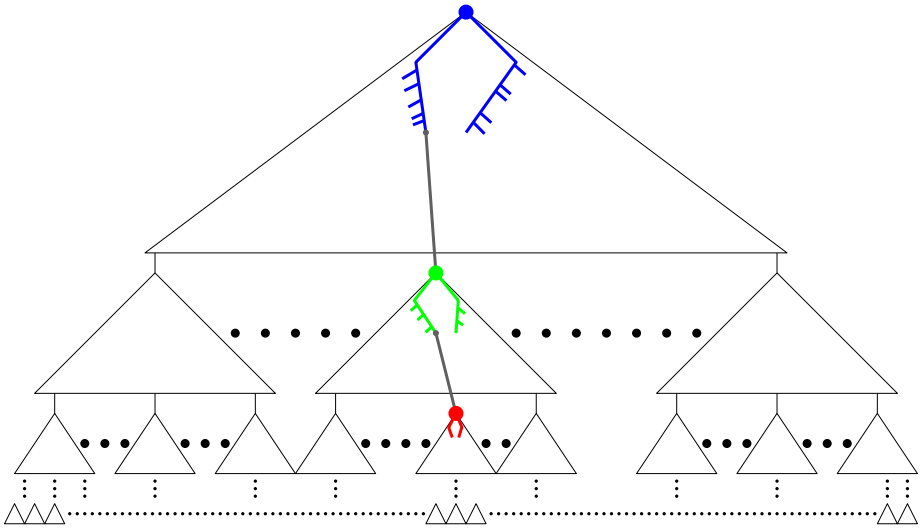
5.2 Analyzing Skip-Splay Trees

Our analysis in this section consists of three lemmas that together prove that skip-splay trees nearly achieve the Unified Bound with a running time of $O(m \lg \lg n + \mathrm{UB}(\sigma))$ on query sequence σ. The purpose of the first lemma is to decompose the cost of skip-splay

[2]Note that this algorithm can be coerced into the BST Model defined in [62] by rotating σ_j to the root and back down, incurring only a constant factor of additional cost.



(a) A skip-splay tree before an access to the red node. To execute the access the solid red node is splayed to the root of its splay tree, the solid green node is splayed to the root of its splay tree, and the solid blue node is splayed to the root of its splay tree.



(b) A skip-splay tree after the three splays have been executed to access the red node. Each node that was splayed is now at the root of its splay tree.

Figure 5.3: A schematic of the skip-splay algorithm. Subfigure (a) shows what a skip-splay tree looks like before an access to the red node, and (b) shows what the tree looks like after the access is performed, which consists, in this case, of three splays, one in each tree that is touched during the access.

trees into a series of "local working set costs," with one cost term for each level in T. The second lemma is the main step of the analysis, and it uses the first lemma to prove that skip-splay trees satisfy a bound that is very similar to the Unified Bound, plus an additive $O(\lg \lg n)$ term for each query. The third lemma shows that this similar bound is within a constant factor of the Unified Bound, so our main analytical result, that skip-splay trees run in $O(m \lg \lg n + \mathrm{UB}(\sigma))$ time, follows immediately from these three lemmas.

In the first lemma and in the rest of this chapter, we will use the following custom notation for describing various parts of T:

1. Let $\rho_k = 1$ and for $i < k$ let $\rho_i = 2^{2^{k-i-1}}$ so that $\rho_i = \rho_{i+1}^2$ for $i < k-1$. Note that if element $x \in T$ is in level i for $i < k$, then $|\operatorname{tree}(x)| = \rho_i - 1$.

2. Let $R_i(x)$, the level-i region of $x \in T$ be defined as follows. First, define the offset $\delta_i = \delta \bmod \rho_i$, where δ is an integer that is arbitrary but fixed for all levels of T. (Our analysis will later make use of the fact that we can choose δ to be whatever we want.) Then, let $R_i(x) = R_i^*(x) \cap T$ where

$$R_i^*(x) = \left\{ \left\lfloor \tfrac{x+\delta_i}{\rho_i} \right\rfloor \rho_i - \delta_i, \ldots, \left\lfloor \tfrac{x+\delta_i}{\rho_i} \right\rfloor \rho_i - \delta_i + \rho_i - 1 \right\}.$$

   Note that the level-i regions partition the elements of T, and the level-$i+1$ regions are a refinement of the level-i regions. Two regions R and R' are said to be *adjacent* if they are distinct, occupy the same level, and their union covers a contiguous region of keyspace. Note that $|R_i(x)| = \rho_i$ if $R_i^*(x) \subseteq T$. Also, note that we omit δ from the notation for a region out of convenience even though, strictly speaking, every region's identity depends on δ.

3. Let $\mathcal{R}_i(x)$, the level-i region set of x, be the set of level-i regions that are subsets of $R_{i-1}(x)$ with $\mathcal{R}_1(x)$ defined to be the set of all level-1 regions. Note that $|\mathcal{R}_i(x)| = \rho_i$ if $1 < i < k$ and $R_{i-1}^*(x) \subseteq T$.

Additionally, we give the following definitions of working set numbers and some auxiliary definitions that will also be helpful (these definitions assume we are working with a fixed query sequence σ):

1. Let $\operatorname{splays}(j)$ be the set of elements that are splayed during query σ_j.

2. Let $p(x, j)$ represent the index of the previous access to x before time j. More formally, assuming such an access exists, let

$$p(x, j) = \max(\{1, \ldots, j-1\} \cap \{j' \mid \sigma_{j'} = x\}).$$

   We define $p(x, j) = -n$ if the argument to $\max$ is the empty set.

3. Let $p_i(x, j)$ represent the index of the previous access to region $R_i(x)$. More formally, assuming such an access exists, let

$$p_i(x, j) = \max(\{1, \ldots, j-1\} \cap \{j' \mid R_i(\sigma_{j'}) = R_i(x)\}).$$

We define $p_i(x, j) = -\rho_i$ if the argument to $\max$ is the empty set. Also, let $p_i(R, j)$ be equivalent to $p_i(x, j)$ if $R = R_i(x)$.

4. For $x \in T$, let $w(x, j)$ represent the number of elements queried since the previous access to x. More formally, if $p(x, j) > 0$ let

$$w(x, j) = \left|\left\{\sigma_{j'} \mid j' \in \{p(x, j), \ldots, j-1\}\right\}\right|.$$

Else, if $p(x, j) \leq 0$ then let $w(x, j) = -p(x, j)$.

5. For $x \in T$, let $w_i(x, j)$ represent the number of regions in $\mathcal{R}_i(x)$ that contain a query since the previous access to a member of $R_i(x)$. More formally, if $p_i(x, j) > 0$ let

$$w_i(x, j) = \left|\left\{R_i(\sigma_{j'}) \mid j' \in \{p_i(x, j), \ldots, j-1\}\right\} \cap \mathcal{R}_i(x)\right|.$$

Else, if $p_i(x, j) \leq 0$ then let $w_i(x, j) = -p_i(x, j)$. Also, let $w_i(R, j)$ be equivalent to $w_i(x, j)$ if $R = R_i(x)$.

In the proof of the first lemma, we will be making use of the reweighing lemma of Georgakopoulos [31], which is an extension of the access lemma of Sleator and Tarjan [55] that allows the weights of nodes to be modified at a cost of $O(\max\{0, \lg(w'/w)\})$, where w' and w are the new and old weights, respectively. For simplicity, we assume we are starting with a minimum potential arrangement of each splay tree, so the final potential can be ignored in our tabulation of the cost of a sequence of accesses. With this in mind, we proceed to prove the following lemma to make our later analysis easier.

Lemma 1. *For a query sequence σ that is served by a skip-splay tree T with k levels, the amortized cost of query σ_j, using an arbitrary value of δ to define the regions, is*

$$O\left(k + \sum_{i=1}^{k} \lg w_i(\sigma_j, j)\right). \tag{5.1}$$

Proof. Our proof will make use of the reweighing lemma of [31], and we maintain an invariant that at time j the weight of each node x in each level-i splay tree is at least $\frac{1}{w_i(R,j)^2}$ for any level-i region R such that x can be splayed during an access to R. As

long as we can maintain these weights and keep the sum of weights in each splay tree bounded by $O(1)$ without paying more credits than our allotment for each query, we will have proved the lemma.

We set up our weighing scheme as follows. First, for any level-i node x that can be splayed as a result of an access to two different level-i region sets, we assign a permanent weight of 1, and we call such nodes *divider nodes*. Note that there are at most two such nodes in any splay tree. Second, besides the divider nodes, the weight of every other node x is defined to be $\max_{R \in \mathcal{R}(x)} \frac{1}{w_i(R,j)^2}$, where $\mathcal{R}(x)$ is the set of regions R for which a query to R can result in a splay of x. Note that at most 6 nodes in any splay tree (3 for each of the 2 level-i region sets that overlap a level-i tree) can have a weight of $\frac{1}{k^2}$ for any k (the count of 6 does not include the divider nodes for $k = 1$). Therefore, the sum of the weights in every splay tree is $O(1)$.

Note that whenever we access a level-i region R, splaying the node we need to splay (if any), and reweighing the nodes we need to reweigh, costs $O(\lg w_i(R, j))$ by construction because at most one node needs to be splayed and at most three nodes need to have their weight increased to 1 from a weight of at least $\frac{1}{w_i(R,j)^2}$. □

We note that the splay trees start at their minimum potential configuration so the sum of the amortized costs of each query, according to Lemma 1, is an upper bound on the cost of the sequence. Using Lemma 1, we can prove a bound that is similar to the Unified Bound, but has an additive $O(\lg \lg n)$ term per query. This bound differs from the Unified Bound in that the working set portion of the cost consists not of the number of *elements* accessed since the previous query to the relevant element, but instead of the number of *queries* since the previous query to the relevant element. Before we prove this bound, we give the following definitions, which will be useful in formally describing the bound and proving it:

1. Let f_j represent the element $\sigma_{j'}$ such that

$$j' = \operatorname*{argmin}_{j'' < j} \lg(w(\sigma_{j''}, j) + |\sigma_j - \sigma_{j''}|).$$

   To provide some intuition for this definition, f_j represents the "finger" for query σ_j because it represents the previously-queried element that yields the smallest Unified Bound value for query σ_j.

2. For $x \in T$, let $t(x, j)$ represent the number of *queries* (rather than distinct elements accessed) since the previous access to x. More formally, let

$$t(x, j) = |\{p(x, j), \ldots, j - 1\}| = j - p(x, j).$$

Note that the above definition handles the case in which $p(x, j) \leq 0$.

3. For $x \in T$, let $t_i(x, j)$ represent the number of queries to all members of $\mathcal{R}_i(x)$ since the previous access to a member of $R_i(x)$. More formally, let

$$t_i(x, j) = \left| \left\{ j' \in \{\max(1, p_i(x, j)), \ldots, j - 1\} \mid R_i(\sigma_{j'}) \in \mathcal{R}_i(x) \right\} \right|,$$

with an additional $-p_i(x, j)$ added if $p_i(x, j) \leq 0$.

4. For $x \in T$, let $\hat{t}_i(x, j)$ represent the number of queries to all members of $\mathcal{R}_i(x)$ since the previous access to x. More formally, let

$$\hat{t}_i(x, j) = \left| \left\{ j' \in \{\max(1, p(x, j)), \ldots, j - 1\} \mid R_i(\sigma_{j'}) \in \mathcal{R}_i(x) \right\} \right|,$$

with an additional ρ_i^2 added if $p(x, j) \leq 0$.

Next, we define $\mathrm{UB}'(\sigma)$, a variant of the Unified Bound, as

$$\mathrm{UB}'(\sigma) = \sum_{j=1}^{m} \lg(t(f_j, j) + |\sigma_j - f_j|), \tag{5.2}$$

and we are ready to proceed with our second lemma.

Lemma 2. *Executing the skip-splay algorithm on query sequence $\sigma = \sigma_1 \cdots \sigma_m$ costs time $O(m \lg \lg n + \mathrm{UB}'(\sigma))$.*

Proof. In this proof, we will be making use of the bound in Lemma 1 with a randomly chosen offset δ that is selected uniformly at random from $\{0, \ldots, \rho_1 - 1\}$. We will use induction on the number of levels i from the top of the tree while analyzing the expected amortized cost of an arbitrary query σ_j. In the inductive step, we will prove a bound that is similar to the one in Lemma 2, and this similar bound will cover the cost associated with levels i and deeper. Even though we are directly proving the inductive step in expectation only, because the bound in Lemma 1 is proven for all values of δ, we know that there exists at least one value of δ such that the bound holds *without* using randomization if we amortize over the entire query sequence. Therefore, the *worst-case* bound on the *total* cost of the access sequence in Lemma 2 will follow.

Our inductive hypothesis is that the cost of skip-splaying σ_j that is associated with levels $i + 1$ and deeper according to Lemma 1 is at most

$$\alpha \lg \hat{t}_{i+1}(f_j, j) + \beta \lg \min(1 + |\sigma_j - f_j|^2, \rho_{i+1}) + \gamma(k - i), \tag{5.3}$$

where k, as before, represents the number of levels of splay trees in T.

We choose levels k and $k-1$ to be our base cases. The inductive hypothesis is trivially true for these base cases as long as we choose the constants appropriately. Also, the bound for the inductive hypothesis at level 1, summed over all queries, is $O(m \lg \lg n + \mathrm{UB}'(\sigma))$, so proving the inductive step suffices to prove the lemma.

To prove the inductive step, we assume Equation 5.3 holds for level $i+1$, and use this assumption to prove the bound for level i. Thus, our goal is to prove the following bound on the cost that Lemma 1 associates with query σ_j for levels i and deeper:

$$\alpha \lg \hat{t}_i(f_j, j) + \beta \lg \min(1 + |\sigma_j - f_j|^2, \rho_i) + \gamma(k - i + 1). \tag{5.4}$$

As a starting point for the proof of the inductive step, Lemma 1 in addition to the inductive hypothesis allows us to prove an upper bound of

$$\lg w_i(\sigma_j, j) + \alpha \lg \hat{t}_{i+1}(f_j, j) + \beta \lg \min(1 + |\sigma_j - f_j|^2, \rho_{i+1}) + \gamma(k - i), \tag{5.5}$$

where we have suppressed the constant from Lemma 1 multiplying $\lg w_i(\sigma_j, j)$.

Our proof of the inductive step consists of three cases. First, if $|\sigma_j - f_j|^2 \geq \rho_i$, then substituting ρ_i for ρ_{i+1} increases the bound in Equation 5.5 by

$$\lg \rho_i - \lg \rho_{i+1} = \lg \left(\tfrac{\rho_i}{\rho_{i+1}} \right) = \lg (\rho_{i+1}) = \lg \left(\rho_i^{1/2} \right) \geq \lg \left(w_i(\sigma_j, j)^{1/2} \right), \tag{5.6}$$

which offsets the elimination of the cost $\lg w_i(\sigma_j, j)$ as long as $\beta \geq 2$. The other substitutions only increase the bound, so for this case we have proved the inductive step.

Second, if $|\sigma_j - f_j|^2 < \rho_i$ and $R_i(\sigma_j) \neq R_i(f_j)$, then we simply pay $\lg w_i(\sigma, j)$ which is at most $\lg \rho_i$. However, we note that the probability of this occurring for a random choice of δ is at most $\rho_i^{1/2}/\rho_i = \rho_i^{-1/2}$, so the expected cost resulting from this case is at most $\rho_i^{-1/2} \lg \rho_i$, which is at most a constant, so it can be covered by γ.

The third and most difficult case occurs when $|\sigma_j - f_j|^2 < \rho_i$ and $R_i(\sigma_j) = R_i(f_j)$, and we will spend the rest of the proof demonstrating how to prove the inductive step for this case. First, we note that $\lg t_i(f_j, j) \geq \lg w_i(f_j, j) = \lg w_i(\sigma_j, j)$, so we can replace $\lg w_i(\sigma_j, j)$ with $\lg t_i(f_j, j)$ and ρ_{i+1} with ρ_i in Equation 5.5 without decreasing the bound and prove a bound of

$$\lg t_i(f_j, j) + \alpha \lg \hat{t}_{i+1}(f_j, j) + \beta \lg \min(1 + |\sigma_j - f_j|^2, \rho_i) + \gamma(k - i). \tag{5.7}$$

It remains only to eliminate the term $\lg t_i(f_j, j)$ by substituting $\hat{t}_i(f_j, j)$ for $\hat{t}_{i+1}(f_j, j)$ while incurring an additional amortized cost of at most a constant so that it can be covered by γ.

Observe that if σ_j satisfies

$$\hat{t}_{i+1}(f_j, j) \leq \frac{\hat{t}_i(f_j,j)}{t_i(f_j,j)^{\frac{1}{2}}}, \tag{5.8}$$

then we have an upper bound of

$$\lg t_i(f_j, j) + \alpha(\lg \hat{t}_i(f_j, j) - \tfrac{\lg t_i(f_j,j)}{2}) + \beta \lg \min(1 + |\sigma_j - f_j|^2, \rho_i) + \gamma(k - i), \tag{5.9}$$

which would prove the inductive step if $\alpha \geq 2$. However, it is possible that $\hat{t}_{i+1}(f_j, j)$ does not satisfy the bound in Equation 5.8. In this latter case, we pessimistically assume that we must simply pay the additional $\lg t_i(f_j, j)$. In the rest of the proof, we show that the amortized cost of such cases is at most a constant per query in this level of the induction, so that it can be covered by the constant γ.

We first give a few definitions that will make our argument easier. A query σ_b is *R-local* if $R_i(\sigma_b) = R$. Further, if σ_b is R-local and satisfies $R_i(f_b) = R$ as well as the bound $\hat{t}_{i+1}(f_b, b) > \hat{t}_i(f_b, b)/t_i(f_b, b)^{\frac{1}{2}}$, then we define σ_b also to be *R-dense*. Note that if σ_b is R-dense then $p(f_b, b) > 0$. Finally, if σ_b additionally satisfies the inequality $\tau < t_i(f_b, b) \leq 2\tau$, then we define σ_b also to be *R-τ-bad*. Notice that all queries that have an excess cost at level i due to being in this third case and not meeting the bound in Equation 5.8 are R-τ-bad for some level-i region R and some value of τ (actually a range of values τ).

Our plan is to show that the ratio of R-τ-bad queries to R-local queries is low enough that the sum of the excess costs associated with the R-τ-bad queries can be spread over the R-local queries so that each R-local query is only responsible for a constant amount of these excess costs. Further, we show that if we partition the R-dense queries by successively doubling values of τ, with some constant lower cutoff, then each R-local query's share of the cost is exponentially decreasing in $\lg \tau$, so each R-local query bears only a constant amortized cost for the excess costs of all of the R-dense queries. Lastly, note that in our analysis below we are only amortizing over R-local queries for some specific but arbitrary level-i region R, so we can apply the amortization to each level-i region separately without interference.

To begin, we bound the cost associated with the R-τ-bad queries for arbitrary level-i region R and constant τ as follows. Let σ_b be the latest R-τ-bad query. First, note that the number of R-τ-bad queries σ_a where $a \in \{p(f_b, b) + 1, \ldots, b\}$ is at most $\hat{t}_i(f_b, b)/\tau$ because there are $\hat{t}_i(f_b, b)$ queries to $\mathcal{R}_i(f_b)$ in that time period, and immediately prior to each such σ_a, the previous $\tau - 1$ queries to $\mathcal{R}_i(f_b)$ are all outside of R so that $t_i(f_a, a) \geq \tau$. Second, note that because σ_b was chosen to be R-τ-bad we have

$$\hat{t}_{i+1}(f_b, b) > \frac{\hat{t}_i(f_b,b)}{t_i(f_b,b)^{1/2}} \geq \frac{\hat{t}_i(f_b,b)}{(2\tau)^{1/2}}. \tag{5.10}$$

Thus, the ratio of the number of R-local queries in this time period, $\hat{t}_{i+1}(f_b, b)$, to the number of R-τ-bad queries in this time period is strictly greater than

$$\frac{\hat{t}_i(f_b,b)}{(2\tau)^{1/2}} \cdot \frac{\tau}{\hat{t}_i(f_b,b)} = (\tfrac{\tau}{2})^{1/2}. \tag{5.11}$$

The constraint that $t_i(f_a, a) \leq 2\tau$ for each of the aforementioned R-τ-bad queries σ_a implies that the excess level-i cost of each is at most $\lg(2\tau)$, so we charge each R-local query with a time index in $\{p(f_b, b) + 1, \ldots, b\}$ a cost of $\lg(2\tau)/(\frac{\tau}{2})^{1/2}$ to account for the R-τ-bad queries that occur during this time interval. Notice that we can iteratively apply this reasoning to cover the R-τ-bad queries with time indices that are at most $p(f_b, b)$ without double-charging any R-local query.

To complete the argument, we must account for all R-dense queries, not just the R-τ-bad ones for some particular value of τ. To do this, for all R-dense queries σ_j such that $t_i(f_j, j) \leq \tau_0$, for some constant τ_0, we simply charge a cost of $\lg \tau_0$ to γ. Next, let $\tau_q = 2^q\tau_0$ for integer values $q \geq 0$. From above, we have an upper bound on the amortized cost of the R-τ_q-bad queries of $\lg(2^{q+1}\tau_0)/(2^{q-1}\tau_0)^{1/2}$, so the sum over all values of q is at most a constant and can be covered by γ. □

To complete the argument that skip-splay trees run in $O(m \lg \lg n + \mathrm{UB}(\sigma))$ time, it suffices to show that $\mathrm{UB}'(\sigma)$ is at most a constant factor plus a linear term in m greater than $\mathrm{UB}(\sigma)$. Thus, the following lemma completes the proof that skip-splay trees run in time $O(m \lg \lg n + \mathrm{UB}(\sigma))$.

Lemma 3. *For query sequence $\sigma = \sigma_1 \ldots \sigma_m$, the following inequality is true:*

$$\sum_{j=1}^{m} \lg(t(f_j, j) + |\sigma_j - f_j|) \leq \tfrac{m\pi^2 \lg e}{6} + \lg e + \sum_{j=1}^{m} 2\lg(w(f_j, j) + |\sigma_j - f_j|). \tag{5.12}$$

Proof. To begin, we give a new definition of a working set number that is a hybrid between $w(f_j, j)$ and $t(f_j, j)$ for arbitrary time index j. Let $h_i(f_j, j)$ be defined as

$$h_i(f_j, j) = \max(w(f_j, j)^2, \min(t(f_j, j), j - i)).$$

Note that $\lg h_m(f_j, j) = 2 \lg w(f_j, j)$ and $h_{-n}(f_j, j) \geq t(f_j, j)$ for all j. Also, note that if $p(f_j, j) > 0$ then $\lg h_{-n}(f_j, j) - \lg h_0(f_j, j) = 0$, else if $p(f_j, j) \leq 0$, which is true for at most n queries, then

$$\lg h_{-n}(f_j, j) - \lg h_0(f_j, j) \leq \lg(n^2 + n) - \lg(n^2) \leq \tfrac{\lg e}{n}.$$

Next, note that $\lg h_i(f_j, j) - \lg h_{i+1}(f_j, j) = 0$ if $i \geq j$ or $t(f_j, j) \leq j - i - 1$, and for all j we have

$$\lg h_i(f_j, j) - \lg h_{i+1}(f_j, j) \leq \tfrac{\lg e}{w(f_j, j)^2}.$$

Also, we know that at most w_0 queries σ_j, for $w_0 \in \{1, \ldots, n\}$, satisfy the following three constraints:

$$\begin{aligned} i &< j \\ t(f_j, j) &\geq j - i \\ w(f_j, j) &\leq w_0. \end{aligned}$$

This is true because each such query is to a distinct element since they all use a finger that was last queried at a time index of at most i (if two of these queries were to the same element, then the second query could use the first as a finger). If there were $w_0 + 1$ such queries, the latest such query σ_ℓ would have $w(f_\ell, j) \geq w_0 + 1$ because of the previous w_0 queries after time i to distinct elements, a contradiction. Therefore,

$$\sum_{j=1}^{m} (\lg h_i(f_j, j) - \lg h_{i+1}(f_j, j)) \leq \sum_{k=1}^{n} \tfrac{\lg e}{k^2} \leq \tfrac{\pi^2 \lg e}{6},$$

so that

$$\sum_{j=1}^{m} (\lg t(f_j, j) - 2 \lg w(f_j, j)) \leq \sum_{j=1}^{m} (\lg h_{-n}(f_j, j) - \lg h_m(f_j, j)) \leq \tfrac{m\pi^2 \lg e}{6} + \lg e.$$

The fact that

$$\lg(t(f_j, j) + d) - 2 \lg(w(f_j, j) + d) \leq \lg t(f_j, j) - 2 \lg w(f_j, j)$$

for all j and non-negative d completes the proof. □

5.3 Remarks on Improvements to Skip-Splay

The ideal improvement to this result is to show that splay trees satisfy the Unified Bound with a running time of $O(m + \mathrm{UB}(\sigma))$. However, achieving this ideal result could be extremely difficult since the only known proof of the dynamic finger theorem is very complicated, and the Unified Bound is stronger than the dynamic finger bound.

In light of this potential difficulty, one natural path for improving this result is to apply the analysis of skip-splay to splay trees, perhaps achieving the same competitiveness to

the Unified Bound as skip-splay trees. Intuitively, this may work because the skip-splay algorithm is essentially identical to splaying, except a few rotations are skipped to keep the elements of the tree partitioned into blocks with a particular structure that facilitates our analysis. One potential first step to accomplishing this would be to show that semi-splaying [55] satisfies the reweighing lemma of Georgakopoulos (or even just the working set bound). If this were true, then semi-splay trees could replace splay trees as the auxiliary data structure of skip-splay trees, and the difference between semi-splay trees and "skip-semi-splay trees" would seem to be even less than the difference between splay trees and skip-splay trees.

The other natural improvement to skip-splay trees, finding a BST that satisfies the Unified Bound with no non-constant multiplicative factor or additive term is achieved by cache-splay trees, which are described in Chapter 6.

Chapter 6

Cache-Splay Trees

In this Chapter, we present the cache-splay algorithm, which is the first BST algorithm that is provably constant-competitive to the Unified Bound of Iacono [37, 16]. This shows that it is possible to build an augmentable data structure that performs well when queries exhibit a combination of locality in space and time (i.e., queries are fast when they are near to a recently accessed element). In comparison to the skip-splay trees of Chapter 5, cache-splay trees maintain a slightly more well-defined structure to the tree. On the one hand, this makes the algorithm less practical and more difficult to program. On the other hand, it greatly simplifies the proof of competitiveness to the Unified Bound, and allows cache-splay trees to eliminate the additive $O(\lg \lg n)$ term that skip-splay trees require in their running time bound relative to the Unified Bound. As elsewhere, we make the simplifying assumption that the set of keys stored in the BST is $\{1, \ldots, n\}$.

6.1 The Cache View of Cache-Splay

Before we define the cache-splay BST algorithm, we will present a simpler version of the algorithm that operates on an array in a multi-level memory hierarchy rather than a BST. This array-based algorithm will serve as a model for how cache-splay trees work. Suppose we have an array containing the elements $\{1, \ldots, n\}$ in sorted order as shown in the bottom-level rectangle of Figure 6.1. (Note that the elements stored in the array are the same as those that are stored in the cache-splay tree.)

Next, suppose we create a series of partitionings of this array. Each such partitioning splits the array into equally-sized contiguous blocks, and each successive partitioning is a refinement of the previous partitioning. The size of a block in each partitioning is 2^{2^i}

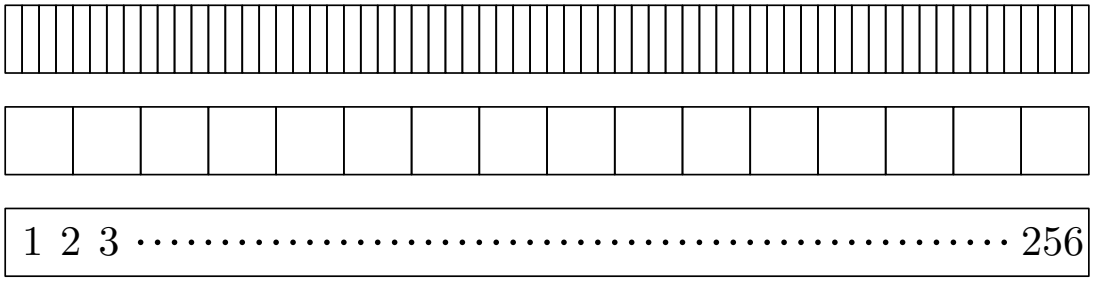


Figure 6.1: The definition of blocks for the cache view of a cache-splay tree. This figure shows the series of partitionings of an array containing the elements $\{1, \ldots, n\}$ for $n = 256$. The bottom rectangle can be considered to be an array containing the elements $\{1, \ldots, 256\}$, and also represents the level-3 partitioning of the array. The second level represents the level-2 partitioning of the array in which each block contains $2^{2^2} = 16$ elements, and the top level represents the level-1 partitioning where each block contains $2^{2^1} = 4$ elements.

for $i \in \{1, \ldots, \lg \lg n\}$ (we assume for simplicity that $\lg \lg n$ is a positive integer), and we define i to be the *level* of the partitioning. Figure 6.1 shows a visual representation of this series of partitionings of the array with level 1 at the top and level $\lg \lg n$ at the bottom.

Each element of the array is stored in exactly one level, and initially all elements are stored in the bottom level (i.e., they are initially in the "disk"). A level-i block is defined to be stored at level i if every element of that block is stored at level i *or higher* (i.e., if a level 2 block B contains some portions that are stored at level 1, and the rest are stored at level 2, then B is stored at level 2).

To perform a query of element x, we begin at level 1 and execute a binary search among all level-1 elements for x. If x is found at level 1, we terminate, else we continue by performing a binary search at the next level unil we find x in its current level i. Then, we cache x's level-$i-1$ block by storing each member of this block in level $i-1$, and continue this caching process until x is in level 1 in a block of size 4.

If we were to continue this for all queries, eventually all elements would percolate up to level 1 of the memory hierarchy, so that the cost of the top-level binary search would be high. Therefore, we impose a limit on the number of blocks that can be stored at each level in the memory hierarchy (just as in a real computer). Specifically, we allow at most 2^{2^i} blocks to be stored at level i at any time. To enforce this constraint, during a query to element x, after we have cached x's level-1 block, we eject a block from any level i that has exactly 2^{2^i} blocks stored in it. The block that is chosen to be ejected is the one that has least recently been queried. Thus, during a typical query to a level-i element x, we

perform binary searches in levels 1 through i to find x, then we cache all of x's blocks (at levels $i-1$ through 1), and then we eject the "stalest" block for levels 1 through $i-1$. This process is shown visually in Figure 6.2.

This invariant ensures that each binary search at level i costs $O(2^i)$. Additionally, we assume that caching or ejecting a level-i block costs $O(2^i)$ so that the total cost of the search is dominated by the cost incurred at the level in which the queried element is found. We will show later that if we amortize this cost over an entire sequence of accesses, this running time bound is the same as the Unified Bound to within a constant factor.

6.2 Implementing the Cache View with a BST

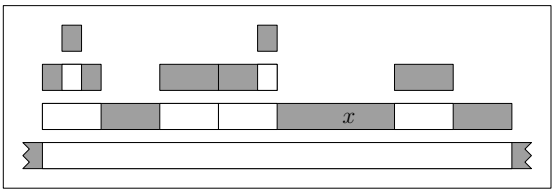
Before we prove that the above algorithm meets the Unified Bound, we show how to implement it with a BST that we call a *cache-splay tree*. Essentially, a cache-splay tree consists of a collection of splay trees that are separated by root markers much like multi-splay trees or skip-splay trees (actually, for cache-splay trees we use a root *counter*). The cache-splay algorithm consists of a series of partial splays, again much like multi-splay trees and skip-splay trees.

A cache-splay tree T is divided into a series of levels so that there is a root node on the path of parent-child pointers between every pair of nodes that are stored in different levels in the cache view of cache-splay that is described in Section 6.1. This is similar to the level structure of layered working-set trees [11]. An edge is defined to be *solid* if the root counter of the child has value 0. Otherwise, if the value of the root counter of the child is strictly greater than 0, then the edge is *dashed*. An example of the state of a cache and the corresponding cache-splay tree is shown in Figure 6.3.

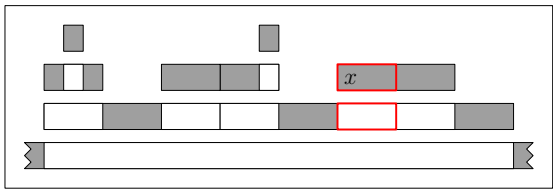
All binary searches that are performed during a single query in the cache view are implemented by a single binary search for the queried element in the cache-splay tree. This single binary search in a cache-splay tree spans as many levels as contain binary searches in the cache view. The caching and ejecting operations that are performed in the cache view are implemented by performing a constant number of splays and root counter increments and decrements at each level. This will be described more formally in Section 6.3.

6.3 The Cache-Splay Algorithm

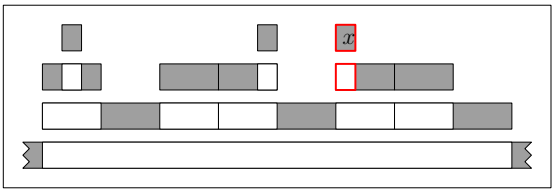
Before we formally define the cache-splay algorithm, we need some notation. We first define b_i to be 2^{2^i} for integral $i \geq 0$, and by convention we set $b_0 = 0$. The value b_i



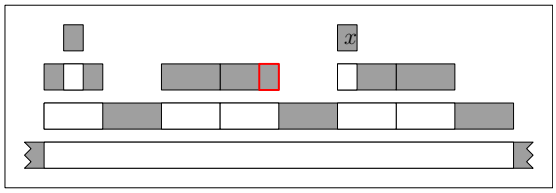
(a) The cache view of a cache-splay tree immediately prior to a query of x.



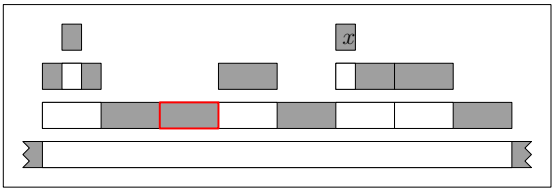
(b) The first iteration of the cache-loop is executed as x's block is lifted one level in the cache.



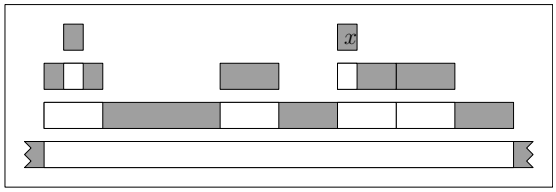
(c) The second iteration of the cache-loop is executed as x's block is lifted one more level in the cache. This block is smaller and more local than the block that is lifted in (b).



(d) The first iteration of the eject-loop is executed as the stalest block is ejected from the top level of the cache. This ejected block was chosen because all other level-1 blocks contain a more recent query.



(e) The second iteration of the eject-loop is executed as the stalest block is ejected from the second level of the cache.



(f) The final state of the cache after the query to x is complete. Note that the eject loop terminates at the same level in which the cache loop started.

Figure 6.2: The cache view of the execution of a query to x in a cache-splay tree. The initial state of the cache before the query is executed is shown in (a); the cache loop is shown in (b) and (c); the eject loop is shown in (d) and (e); and the final state of the cache after the query to x has been finished is shown in (f). The block that is chosen to be cached during each iteration of the cache step is the block that contains the queried node x, and the block that is chosen to be ejected from each level is the one that has least recently been accessed.

denotes the size of a block at level i in T. We define the level-i block of node x, denoted by $B_i(x)$, to be

$$\left\{ \left\lfloor \frac{x}{b_i} \right\rfloor b_i, \ldots, \left\lfloor \frac{x+b_i}{b_i} \right\rfloor b_i - 1 \right\}.$$

Additionally, for an arbitrary non-negative integral offset δ, we define the δ-offset level-i block of node x, denoted by $B_i(x, \delta)$, to be

$$\left\{ \left\lfloor \frac{x+(\delta \bmod b_i)}{b_i} \right\rfloor b_i - (\delta \bmod b_i), \ldots, \left\lfloor \frac{x+(\delta \bmod b_i)}{b_i} \right\rfloor b_i - (\delta \bmod b_i) + b_i - 1 \right\}.$$

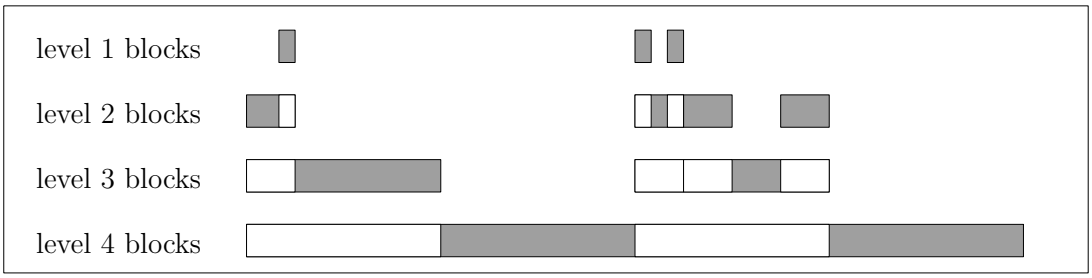
The term $B_i(x, \delta)$ is not used in defining the cache-splay algorithm, but it is helpful in the analysis of cache-splaying.

We can formally define the cache-splay algorithm as follows. We assign a *root counter* to every node in a cache-splay tree T. The tree T is an ordinary BST that is partitioned by the set of root counters that have a strictly positive value. Every such positively-valued root counter represents the root of a distinct splay tree. Whenever a node rotates over another node that has a strictly positive root counter, the root counter is transferred to the new parent. The set of these splay trees partitions the nodes of T.

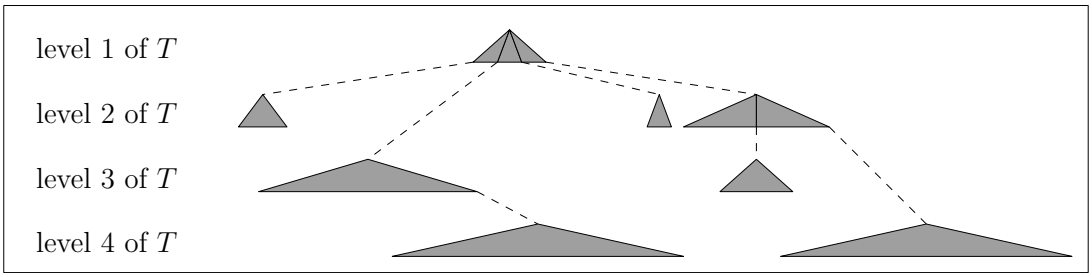
We define the *level* of node x to be the sum of the root counters of nodes on the BST access path from the root of T to x, including both the root and x. Level i of T is defined to be the set of level-i nodes in T. We say that a block $B_i(x)$ is contained in level i of T if the level of every node in $B_i(x)$ is at most i. Figure 6.3 shows a visual depiction of how the blocks of the keyspace relate to a corresponding cache-splay tree T.

In addition to using the tree, cache-splay trees use a set of linked lists $\{L_1, \ldots, L_k\}$, where k is the number of levels in T. List L_i stores a list of level-i blocks B, such that every element of B has level i. The blocks in L_i are ordered in move-to-front order according to how recently they have been accessed (i.e., the "stalest" block is at the back of L_i). The representative element of each block that is stored in L_i is the LCA of the block. We store bi-directional pointers between each element of L_i and the LCA of the block in T. Also, we keep a pointer to the front and back of each list. We use these lists to implement a least recently used paging rule at each level in the cache. Below, we will be specifying a limit on the number of blocks that can be stored at each level, and whenever we cache an additional block in a level that is already full, we use that level's linked list to find the least recently queried block so that we can eject that block from that level of the cache. To give some intuition for how cache-splay trees work before we formally define the algorithm, the operation of a single query as seen from the cache view is shown in Figure 6.2.

Although the inclusion of this linked list and extra pointers violates the strict definition of a BST model defined in Chapter 2, we still perform all of the necessary BST operations



(a) The cache view of a cache-splay tree. The gray rectangles represent regions of the keyspace that are stored at the corresponding level, and the white rectangles represent parts of the keyspace that can be thought of as being cached at that level, but that are actually stored at an even higher level in the cache-splay tree. The vertical portions of the black borders around the rectangles represent the divisions between blocks that are stored at a particular level, or they represent the boundary between a portion of a block that is stored at that level and a portion of that block that is stored at a higher level. Note that the ratio of block sizes in successive levels is not to scale in this figure.



(b) A cache-splay tree corresponding to the cache view shown in (a). Each triangle represents a splay tree that is a member of the indicated level in the cache-splay tree, and each dashed edge between two triangles represents a single edge that separates two splay trees in a cache-splay tree. Each black line inside a triangle represents the border between two blocks that are stored, at least partially, in the same splay tree. It is important to note that many blocks can be stored in the same splay tree, but only blocks of one particular level can be stored in any one splay tree. Also, the dashed edges that span more than one level correspond to root counters whose values are strictly greater than one.

Figure 6.3: The blocks of a "cache" compared to the structure of the corresponding cache-splay tree. The cache view is shown in (a), and the corresponding cache-splay tree structure is shown in (b). In both (a) and (b), the keyspace is ordered from 1 to n from left to right, and the vertical position of the rectangles and triangles represents, respectively, the level of the elements in the corresponding block or splay tree. Neither of the above subfigures is drawn to scale, and the size and number of blocks at each level has been changed to make the figure easier to understand.

to make use of BST augmentation, the key practical difference between the BST model and other models of computation supporting one-dimensional search. Further, we can emulate the linked lists with at most $O(\lg n)$ additional auxiliary bits per node as we will see in Section 6.5. For the purpose of simplicity and easier intuition, we will continue to describe the version of cache-splaying that uses auxiliary linked list data structures.

Also, for simplicity, we will only handle queries, no insertion or deletion, and we will assume that T is initially balanced and in the minimum-potential initial configuration so that the amortized bounds we will show constitute a lower bound on the cost of the algorithm when summed across all queries. Cache-splay trees will maintain the following invariant, which specifies which blocks are cached at which levels. We will assume that each level of the cache is initially arbitrarily filled with blocks to satisfy the invariant.

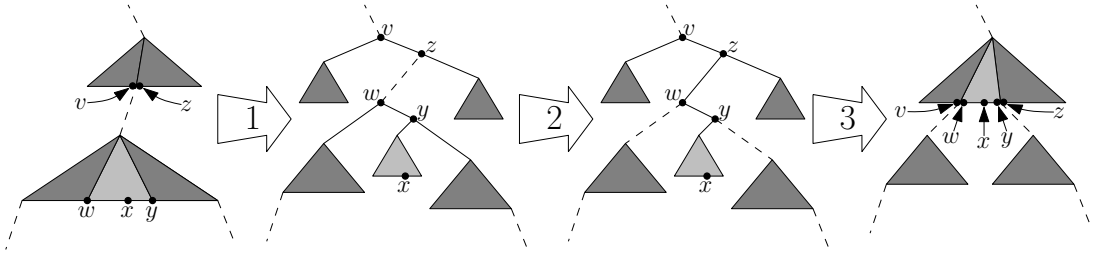
Invariant 1. *Immediately prior to every query, there are exactly $b_i - 1$ level-i blocks that have some members stored at level i or higher in a cache-splay tree T.*

To query a node x using the cache-splay algorithm, we perform an ordinary BST search starting at the root of T, keeping track of the sum of the root counters seen. When x is reached, we remember the initial level i_0 of x. Next, we need to "cache" x's blocks of nodes into the "faster levels of the cache", which is accomplished by the following steps, beginning at level $i = i_0$.

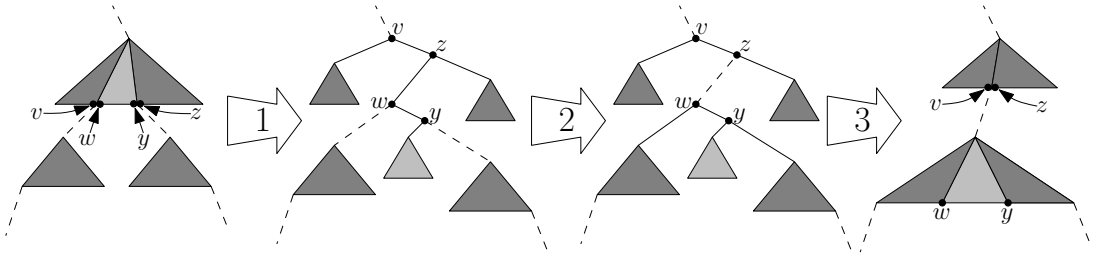
Cache Loop. While $i > 1$, repeat the following steps. Splay the minimum element w of $B_i(x)$, and splay the maximum element y of $B_i(x)$ until it is the right child of w. Then, splay x's level-$i-1$ predecessor v, and splay x's level-$i-1$ successor z until z is the right child of v. Next, change x's level by incrementing the root counters of both w's left child and y's right child, and then decrementing w's root counter. Finally, if $B_i(w)$ is in L_i, then we remove it from L_i. Figure 6.4(a) shows an example of what one iteration of the cache loop looks like.

Next, in order to restore Invariant 1 for the next query, we need to eject "stale" blocks of nodes from the fast levels of T. This is accomplished as follows, beginning with level $i = 1$.

Eject Loop. While $i < i_0$, repeat the following steps. Eject the block B corresponding to the last element of L_i by first removing its list element from L_i. Then, splay B's level-i predecessor v, and splay B's level-i successor z until z is the right child of v. Then, splay the minimum element w of B until it is a child of z, if z exists, and splay the maximum element y of B until it is the right child of w. Next, change B's level by incrementing



(a) An example of what one iteration of the cache loop looks like. The cache view of two cache loop iterations is shown in Figures 6.2(b) and 6.2(c).



(b) An example of what one iteration of the eject loop looks like. The cache view of two eject loop iterations is shown in Figures 6.2(d) and 6.2(e).

Figure 6.4: An example of what one iteration of the cache loop and eject loop looks like. Note that the steps performed during one iteration of the eject loop, shown in (b), is essentially just the reverse of the steps performed during one iteration of the cache loop, shown in (a).

w's root counter and decrementing the root counters of both w's left child and y's right child. Finally, if neither v nor z is in $B_{i+1}(w)$, then insert $B_{i+1}(w)$ at the front of L_{i+1}. Figure 6.4(b) shows an example of what one iteration of the eject loop looks like.

To strictly follow the definition of the BST model, after executing the eject loop, we would rotate x to the root, and then back to its location at the end of the execution of the eject loop. This would only cost an extra $O(1)$ per operation because after the eject loop, x is in level 1, and there there are $O(1)$ nodes in level 1.

6.4 Cache-Splay Satisfies the Unified Bound

To analyze the running time of the cache-splay algorithm, we first define some notation, which will be helpful in our analysis later on. This notation is similar, but slightly different, to that which is defined for the analysis of skip-splay trees in Section 5.2.

- Let $p(x, j)$ represent the index of the previous access to x. More formally, assuming such an access exists, let

$$p(x, j) = \max(\{1, \ldots, j-1\} \cap \{j' \mid \sigma_{j'} = x\}).$$

  Else, if there is no previous access to x at time j, then we define $p(x, j) = -n$.

- For $x \in T$, let $w(x, j)$ represent the number of elements queried since the previous access to x. More formally, assuming $p(x, j) \geq 1$, let

$$w(x, j) = \left| \left\{ \sigma_{j'} \mid j' \in \{p(x, j), \ldots, j-1\} \right\} \right|.$$

  Else, if $p(x, j) \leq 0$ then let $w(x, j) = -p(x, j)$. For block B, we assign $w(B, j)$ the natural definition (i.e., the number of elements queried since a query to B).

- Let $p_i(x, j)$ represent the index of the previous access to a member of $B_i(x)$. More formally, assuming such an access exists, let

$$p_i(x, j) = \max(\{1, \ldots, j-1\} \cap \{j' \mid \sigma_{j'} \in B_i(x)\}).$$

  Else, if there is no previous access to a member of $B_i(x)$ at time j, then we define $p_i(x, j) = -n$.

- For $x \in T$, let $w_i(x, j)$ represent the number of elements queried since the previous access to a member of $B_i(x)$. More formally, assuming $p_i(x, j) \geq 1$, let

$$w_i(x, j) = \left| \left\{ \sigma_{j'} \mid j' \in \{p_i(x, j), \ldots, j-1\} \right\} \right|.$$

  Else, if $p_i(x, j) \leq 0$ then let $w_i(x, j) = -p_i(x, j)$.

- Let $p_i(x, \delta, j)$ represent the index of the previous access to a member of $B_i(x, \delta)$. More formally, assuming such an access exists, let

$$p_i(x, \delta, j) = \max(\{1, \ldots, j-1\} \cap \{j' \mid \sigma_{j'} \in B_i(x, \delta)\}).$$

  Else, if there is no previous access to a member of $B_i(x, \delta)$ at time j, then we define $p_i(x, \delta, j) = -n$.

- For $x \in T$, let $w_i(x, \delta, j)$ represent the number of elements queried since the previous access to a member of $B_i(x, \delta)$. More formally, assuming $p_i(x, \delta, j) \geq 1$, let

$$w_i(x, \delta, j) = \left| \left\{ \sigma_{j'} \mid j' \in \{p_i(x, \delta, j), \ldots, j-1\} \right\} \right|.$$

  Else, if $p_i(x, \delta, j) \leq 0$ then let $w_i(x, \delta, j) = -p_i(x, \delta, j)$.

In the analysis below, we will use a potential function similar to that which was used to prove the main splay tree theorems in [55], except that the root counters will block the weight in their subtrees from being felt by the rest of the tree, just as in the analysis of skip-splay trees as well as multi-splay trees [61].

More specifically, we assign assign a weight of one to every node in T, and define the size of node x, denoted by $s(x)$, to be equal to the sum of the weights in the subtree of x's splay tree that is rooted at x (i.e., the number of nodes that can be reached by following child pointers starting at x traversing only nodes whose root counter is zero). The potential of T is defined to be $\sum_{x \in T} \lg s(x)$, and the amortized cost of each access will be defined to be the sum of the actual costs of the algorithm that are described above and the change in potential. We assume that the initial configuration of T is one of minimum potential, so that the sum of amortized costs, according to the splay tree access lemma, of the splays and root counter changes is an upper bound on the actual cost of the entire sequence.

Note that we are only counting the cost of the rotations in the following analysis. Pointer traversal, field updates, and changes to the lists L_i would, in reality, have costs too, but these costs are dominated by the number of rotations, so we ignore them and stick to the BST model's defined cost metric, in which an algorithm is only charged for the rotations it performs.

We begin by proving the following lemma, which bounds the cost associated with a particular level of T during a query sequence.

Lemma 4. *During a query sequence in a cache-splay tree containing n nodes, suppose a node x is queried. The amortized cost of the operations performed at level-i in T is at most $c \log b_i$, for some constant c.*

Proof. By Invariant 1, we know that each level-i splay tree contains nodes from at most b_i level-i blocks, so each level-i splay tree contains at most b_i^2 nodes. Therefore, a splay in level i of T has an amortized cost of $2 \lg b_i$ by the access lemma for splay trees [55], with the constants of the bound in the splay tree access lemma suppressed for simplicity. Thus, the up to eight splays that are executed in a level-i splay tree cost a total of at most $16 \log b_i$. Further, the cost of the extra weight added when up to one additional level-i block is added to level i of T is at most $2 \lg b_i$ because the root of the added tree has at most two ancestors in level-i immediately before it is added to level i of T. An ejection of a block from level-i has amortized cost at most zero because such an ejection causes a decrease in potential. However, the merging in of a block that was ejected from level $i-1$ causes an additional increase in potential of up to $6 \lg b_i^2$. □

Lemma 5. *During a query sequence in a cache-splay tree containing n nodes, suppose a node x is queried. The amortized cost of the operations performed at level-i in T is zero if both $i > 1$ and $w_i(x, j) < b_{i-1}$.*

Proof. First, note that if $w_i(x, j) < b_{i-1}$, then immediately before x is queried x must reside at level $i - 1$ or less in T because, at a minimum, $b_{i-1} - 1$ queries must be executed following a query to a member of $B_i(x)$ for x to be ejected from level $i - 1$. Second, note that the definition of the cache-splay algorithm prevents any operations from occurring at a level below the level in which x was found. □

Next, we enhance the analysis of Lemmas 4 and 5 by allowing an arbitrary offset δ on the boundaries of the blocks. Note that the actual blocks used by the cache-splay algorithm do not have this offset, just the blocks used in the analysis. As above, we break the analysis into two cases, and we describe the potential function used in both proofs here.

In addition to using the ordinary splay tree potentials that were used in the proofs of Lemmas 4 and 5, we add an additional potential equal to the following. If δ is chosen to be an arbitrary nonnegative integer, for every δ-offset level-i block B_δ, we assign a potential of $c \lg b_i$ exactly when $w(B_\delta, j) < b_{i-1} \leq w(B, j)$ for one of the up to two level-i blocks B that intersect B_δ.

Lemma 6. *Let δ be an arbitrary nonnegative integer and $\phi_\delta(T)$ be the corresponding potential function as defined above. During a query sequence in a cache-splay tree containing n nodes, suppose a node x is queried. The amortized cost of the operations performed at level-i in T is at most $2c \log b_i$.*

Proof. By Lemma 4, the amortized cost of the level-i operations according to the splay tree access lemma is $c \log b_i$. Additionally, we may increase $\phi_\delta(T)$ by up to $c \log b_i$. □

Lemma 7. *Let δ be an arbitrary nonnegative integer and $\phi_\delta(T)$ be the corresponding potential function as defined above. During a query sequence in a cache-splay tree containing n nodes, suppose a node x is queried. The amortized cost of the operations performed at level-i in T is at most zero if both $i > 1$ and $w_i(x, \delta, j) < b_{i-1}$.*

Proof. We consider two cases. First, suppose $w_i(x, j) < b_{i-1}$. In this case, the level-i terms of $\phi_\delta(T)$ do not change, and Lemma 5 shows that the amortized cost associated with level i is zero. Second, suppose $w_i(x, j) \geq b_{i-1}$. In this case, the level-i terms of $\phi_\delta(T)$ decrease by at least $c \lg b_i$, which, by Lemma 4 is sufficient to pay for the cost at level i. □

Using Lemmas 6 and 7, we can prove that cache-splay trees satisfy the Unified Bound, as shown in the following theorem.

Theorem 8. *The cost of a query sequence $\sigma = \sigma_1 \cdots \sigma_m$, where $\sigma_j \in \{1, \ldots, n\}$ and $n = 2^{2^k}$ for some integral $k \geq 1$, using the cache-splay algorithm starting with an initially balanced cache-splay tree T, is $O(m + \mathrm{UB}(\sigma))$.*

Proof. Choose an offset δ randomly from $\{1, \ldots, n\}$. Let σ_j be an arbitrary query from sequence σ, and choose $j' < j$. It suffices to show that the amortized cost of query σ_j is at most $O(\lg w(\sigma_{j'}, j) + \lg(|\sigma_{j'} - \sigma_j| + 1))$. We begin by breaking the cost associated with query σ_j into the cost associated with each level. We group these per-level costs into three sums, and the cost associated with each level is included in at least one of these sums.

First, choose i_f such that $b_{i_f} \leq |\sigma_{j'} - \sigma_j| < b_{i_f+1}$. By Lemma 6 we know that the total cost associated with levels 1 through i_f in T is at most

$$\sum_{i=1}^{i_f} 2c \lg b_i \leq 4c \lg(b_{i_f} + 1) = O(\lg(|\sigma_{j'} - \sigma_j| + 1)),$$

where the argument $b_{i_f} + 1$ includes an additive one for the case $i_f = 0$.

Second, choose i_w such that $b_{i_w-2} \leq w(\sigma_{j'}, j) < b_{i_w-1}$. By Lemma 6 we know that the total cost associated with levels 1 through i_w in T is at most

$$\sum_{i=1}^{i_w} 2c \lg b_i \leq 4c \lg b_{i_w} = O(\lg w_i(\sigma_{j'}, j)).$$

Third, choose $i^* = 1 + \max\{i_f, i_w\}$. By Lemma 7 and the fact that $\delta \bmod b_i$ is distributed randomly (but not independently) for all i, we know that the expected total cost associated with levels i^* and larger in T is at most

$$\sum_{i=i^*}^{\infty} \frac{|\sigma_{j'} - \sigma_j|}{b_i} 2c \lg b_i = O(\lg b_{i^*}) = O(\lg \max\{w(\sigma_{j'}, j), |\sigma_{j'} - \sigma_j| + 1\}).$$

Summing these three bounds on the costs associated with various subsets of the levels yields a bound of $O(\lg(w(\sigma_{j'}, j) + |\sigma_{j'} - \sigma_j|))$, and this bound is sufficient to cover the expected cost associated with all levels. To finish the proof, we first note that the choice of j' was arbitrary and we could choose whatever j' minimizes the cost when analyzing each query to achieve a bound of $O(\min_{j'<j} \lg(w(\sigma_{j'}, j) + |\sigma_{j'} - \sigma_j|))$ expected cost for each query σ_j. Finally, note that because this expected cost was proved using a random choice of δ, and this random choice has no effect on the cache-splay algorithm itself, there must be some choice of δ for which this bound holds *without* using any random choices in the analysis. □

6.5 Making Cache-Splay a Strict BST Algorithm

The cache-splay algorithm, as defined in Section 6.3, does not fit the formal definition of a BST model that forbids the extra pointers that are needed by the linked lists. We stress that this is of no practical consequence because the tree portion of a cache-splay tree still performs the required rotations, so cache-splay trees can be used for anything that a strict BST can be used for. Further, it is possible to emulate the linked list using a small additional overhead, and coerce the cache-splay algorithm into this more strict definition of a BST. One way of achieving this is shown in [11]. Alternatively, we could make the following modification to cache-splay trees.

In a cache-splay tree T, for every level-i node x that is the LCA of its level-i block, we could store the "index" of $B_i(x)$ in L_i. This "index" would not be the exact index of $B_i(x)$ but would be monotonically increasing in the position of $B_i(x)$. In such a scheme, we could emulate the move-to-front list for each level as follows.

To move $B_i(x)$ to the front of L_i, we would set the index of x to be one less than the current minimum index of any level-i block. We consider the case in which this causes an integer underflow below. This minimum index could be stored for each level at the root or some separate memory location that is easy to access.

Finding the back element of list L_i would be a little harder. To accomplish this, for every node x in T, for every level i, we maintain a $\texttt{maxIndex}_i$ field that stores the maximum index that appears in its subtree, considering only the indices of level-i blocks that are fully stored in level-i. Then, it is straightforward to use binary search to find the level-i block corresponding to the back of list L_i using the standard technique. Note that although this search starts from the root, this operation would only be called during the eject loop, and the pointer traversal time used by this search could be charged to the the splays that are executed to eject this block from its level, with just a constant factor of overhead. Finally, to remove $B_i(x)$ from list L_i, where x is the LCA of $B_i(x)$, we delete x's index, and update the $\texttt{maxIndex}_i$ fields of x's ancestors. Again, these field updates can be charged to the splays executed during the query.

To handle the caveats mentioned above, if one less than the current minimum index would cause an integer underflow, we could reset the entire tree's move-to-front indexes to be at the maximum end of the index range. As long as our indexes were stored in $(1 + \varepsilon) \lg n$ bits with $\varepsilon > 0$, we would only have to perform this rebuild operation once every $\Omega(n^{1+\varepsilon})$ operations, so the amortized cost of the rebuild would be $o(1)$ per operation.

Note that each node has $\Theta(\lg \lg n)$ auxiliary fields, which naïvely requires a space usage of $\Theta(\lg \lg n \lg n)$ per node. To avoid using $\omega(\lg n)$ auxiliary bits per node to store all of the $\texttt{maxIndex}_i$ fields, we restrict the range of the indices for the shallower levels

in the tree. Specifically, we use only $\Theta(\lg b_i)$ bits to store the index at level i. Because there are $O(b_i^2)$ elements stored at level i and shallower, a similar argument to the above one shows that it is possible to achieve an amortized rebuild time of $o(1)$ per operation. Moreover, the total memory usage used when storing all of the $\texttt{maxIndex}_i$ fields is just $O(\lg n)$ because the memory usage is dominated by the storage for $\texttt{maxIndex}_k$, where k is the deepest level in T.

6.6 The Next Steps for Adaptive BSTs

It is worth noting that there is no reason to use splay trees as opposed to a balanced BST algorithm such as red-black trees for the auxiliary BST algorithm in cache-splay trees. It is convenient to use splay trees because this allows a more complete version of the algorithm to be described (no "library calls" to split and join red-black trees). Also, using splay trees as the auxiliary BST algorithm facilitates a comparison between splay trees and cache splay trees that may help uncover properties of splaying that can be used to prove the Unified Bound for splay trees.

On the other hand, if red-black trees were used, it might be possible to achieve the Unified Bound in the worst case using a cache-splay-tree-like BST algorithm. This would still require considerable work, however, because even though using red-black trees improves the worst case running time to $O(\lg n)$, proving the Unified Bound still requires amortization. To see this, note that a simple linear scan of the keys requires occasional "deep cache misses" that require a split and merge that consumes $\Omega(\lg n)$ time. To achieve the Unified Bound in the worst case, therefore, one would have to develop a scheme for gradually executing deep cache operations than cannot be fully afforded by any one operation.

Another extension would be to simplify the algorithm so that it would be easier to implement with less overhead. One idea for doing this is to randomly eject a block at each level instead of ejecting the least recently used block. This introduces randomization into the bound, but this seems like it should work since there would still be a reasonably high probability that the block stays in each level for an amount of time that is polynomial in the number of blocks stored at that level.

As suggested in Section 5.3, another natural direction for related progress is to prove better bounds for splay trees. Examples include improving upon Pettie's proof that splay trees cost $\alpha^*(n)$ per operation when used as deques, and proving *any* non-trivial competitiveness to the Unified Bound. Aside from results on splay trees, as suggested in Section 4.5, it would be interesting to further our understanding of formulaic bounds that generalize the Unified Bound and can be achieved in the BST model.

Chapter 7

Conclusion

This thesis has made a number of contributions to our knowledge of what is achievable in the BST model when we analyze the cost of BST algorithms on query sequences that may contain patterns that can be exploited to speed up running time.

We showed a general lower bound framework that not only generalized existing lower bounds, but also demonstrated a potentially deep connection between BST data structures and the partial-sums problem, in which sums over ranges of an array are computed while various array values are updated to new values. In addition to the obvious question regarding the tightness of the lower bound framework for the BST model, another equally important question is whether the lower bound holds true for even more general problems than the partial-sums problem in the set-sum model of computation.

As even deeper connections are shown between BST algorithms and such problems, the motivation to seek simpler and provably better BST algorithms is greatly increased. Knowing that a bound is achievable by a BST is a good thing, and it is even better if this bound is easy to prove. However, it is still yet better to know that the bound can be achieved by a *simple* BST algorithm that can easily be implemented both with low running time and with low cognitive load by whoever is doing the programming. Even an extremely complicated proof that a good bound holds for a simple algorithm is extremely valuable because once the proof is accepted as valid, the programmer only needs to remember the theorem and the algorithm to make practical use of the result.

To that end, we showed a variety of results for relatively simple BST algorithms that were based on splaying, which is perhaps the simplest robust adaptive BST algorithm of all. In addition to the work on multi-splay trees that was summarized in this thesis, and whose details are found in [61, 60, 25], we showed that the BST model was sufficiently flexible to achieve the Unified Bound of Iacono, resolving this open question that was

posed in 2001 [37]. In doing so, we introduced two new splay-based BST algorithms, skip-splay and cache-splay. These two algorithms illustrate the tradeoff that algorithm designers often face between the complexity of the algorithm and the complexity of the proof. Skip-splay trees are a simpler data structure with more difficult proofs and worse guarantees since they only achieve the Unified Bound to within additive $O(\lg \lg n)$. Cache-splay trees, on the other hand, are more complicated, but have simpler proofs and eliminate the additive $O(\lg \lg n)$ term for their competitiveness compared to the Unified Bound.

Now that the Unified Bound has been achieved by a BST, the two most natural directions for related progress are to prove the Unified Bound for a simpler BST algorithm, such as splaying, and to devise new bounds that generalize the Unified Bound, with the goal of finding a formulaic bound that completely encapsulates dynamic optimality in the BST model. Such results would provide even further motivation for discovering whether the splay algorithm, or some other online BST, is provably $O(1)$-competitive to the optimal BST algorithm.

Bibliography

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988. 2.1.3

[2] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC 2000)*, pages 335–342, 2000. 4.3

[3] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):Article 13, 2007. 4.3

[4] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, December 1972. 2.2

[5] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science (FOCS 2000)*, pages 399–409, 2000. 2.1.3

[6] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005. 2.1.3

[7] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 29–38, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. 2.1.3

[8] Guy E. Blelloch, Bruce M. Maggs, and Shan Leung Maverick Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 374–383, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. 4.3

[9] Avrim Blum, Shuchi Chawla, and Adam Kalai. Static optimality and dynamic search-optimality in lists and trees. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 1–8, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. 2.2, 3, 4.2

[10] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An O(log log n)-competitive binary search tree with optimal worst-case access times. Obtained on December 7, 2009 from: http://cgm.cs.mcgill.ca/ vida/pubs/papers/ZipperTrees.pdf, 2009. 1, 4.1

[11] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and John Howat. Layered working-set trees. *CoRR*, abs/0907.2071, 2009. 4.3, 6.2, 6.5

[12] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and John Howat. Layered working-set trees. In *Proceedings of the 9th Latin American Theoretical Informatics Symposium (LATIN 2010)*, 2010. 4.3

[13] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002. 2.1.3

[14] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios K. Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences*, 67(2):381–418, 2003. 4.3

[15] Mark R. Brown and Robert Endre Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980. 2.1.2, 4.3

[16] Mihai Bădoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007. 1, 2.1.2, 4.4, 5, 6

[17] Timothy M. Chan. Closest-point problems simplified on the RAM. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 472–473, 2002. 4.6

[18] Richard Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000. 4.3, 5

[19] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000. 4.3, 5

[20] Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Pătraşcu. The geometry of binary search trees. In *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA 2009)*, pages 496–505, 2009. 3.4, 3.5, 4.2

[21] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality – almost. In *Proceedings of the 45th IEEE Symposium on Foundations of Computer Science (FOCS 2004)*, pages 484–490, 2004. 1, 3.1, 3.6, 4.1

[22] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Pătraşcu. Dynamic optimality – almost. *SIAM Journal on Computing*, 37(1):240–251, 2007. 1, 3.1, 4.1

[23] Erik D. Demaine, John Iacono, and Stefan Langerman. Proximate point searching. *Computational Geometry: Theory and Applications*, 28(1):29–40, 2004. 4.6

[24] Jonathan Derryberry, Don Sheehy, Daniel D. Sleator, and Maverick Woo. Achieving spatial adaptivity while finding approximate nearest neighbors. In *Proceedings of the 20th Canadian Conference on Computational Geometry (CCCG 2008)*, pages 163–166, 2008. 4.6

[25] Jonathan Derryberry, Daniel Sleator, and Chengwen Chris Wang. Properties of multi-splay trees. Technical Report CMU-CS-09-171, Carnegie Mellon University, 2009. 4.1, 7

[26] Jonathan Derryberry, Daniel Dominic Sleator, and Chengwen Chris Wang. A lower bound framework for binary search trees with rotations. Technical Report CMU-CS-05-187, Carnegie Mellon University, 2005. 3.4, 3.5, 1

[27] Jonathan C. Derryberry and Daniel D. Sleator. Skip-splay: Toward achieving the unified bound in the BST model. In *Proceedings of the 11th International Symposium on Algorithms and Data Structures (WADS 2009)*, pages 194–205, Berlin, Heidelberg, 2009. Springer-Verlag. 5

[28] Amr Elmasry. On the sequential access theorem and deque conjecture for splay trees. *Theoretical Computer Science*, 314(3):459–466, 2004. 4.3

[29] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. 2.1.1

[30] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS 1999)*, Washington, DC, USA, 1999. IEEE Computer Society. 2.1.3

[31] George F. Georgakopoulos. Splay trees: a reweighing lemma and a proof of competitiveness vs. dynamic balanced trees. *Journal of Algorithms*, 51(1):64–76, 2004. 4.3, 5.2, 5.2

[32] George F. Georgakopoulos. Chain-splay trees, or, how to achieve and prove log log n-competitiveness by splaying. *Information Processing Letters*, 106(1):37–43, 2008. 1, 4.1

[33] Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proceedings of the 9th ACM Symposium on Theory of Computing (STOC 1977)*, pages 49–60, 1977. 4.3

[34] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science (FOCS 1978)*, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society. 2.2

[35] Anupam Gupta. Personal communication with Anupam Gupta of Carnegie Mellon University, January 2006. 3.5

[36] Dion Harmon. *New Bounds on Optimal Binary Search Trees*. PhD thesis, Massachusetts Institute of Technology, 2006. 3.4

[37] John Iacono. Alternatives to splay trees with o(log n) worst-case access times. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 516–522, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics. 1, 2.1.2, 4.4, 5, 6, 7

[38] John Iacono. Optimal planar point location. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 340–341, 2001. 4.6

[39] John Iacono. Key-independent optimality. *Algorithmica*, 42(1):3–10, 2005. 4.3

[40] John Iacono and Stefan Langerman. Proximate planar point location. In *Proceedings of the 19th ACM Symposium on Computational Geometry (SoCG 2003)*, pages 220–226, 2003. 4.6

[41] Alexis C. Kaporis, Christos Makris, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos D. Zaroliagis. Improved bounds for finger search on a RAM. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 2003)*, pages 325–336, 2003. 4.3

[42] Jussi Kujala and Tapio Elomaa. Poketree: A dynamically competitive data structure with good worst-case performance. In *Proceedings of the 17th International Symposium on Algorithms and Computation (ISAAC 2006)*, pages 277–288, 2006. 1, 4.1

[43] Swanwa Liao, Mario A. Lopez, and Scott T. Leutenegger. High dimensional similarity search with space filling curves. In *Proceedings of the 17th International Conference on Data Engineering (ICDE 2001)*, pages 615–622, 2001. 4.6

[44] J.M. Lucas. Canonical forms for competitive binary search tree algorithms. Technical Report DCS-TR-250, Rutgers University, December 1988. 3, 3.4, 4.2

[45] D.J. McClurkin and G.F. Georgakopoulos. Sphendamnœ: A proof that k -splay fails to achieve logk n behaviour. In *Proceedings of the 8th Panhellenic Conference on Informatics (PCI 2001)*, pages 480–496, 2001. 2.1.3

[46] J. Ian Munro. On the competitiveness of linear search. In *Proceedings of the 8th Annual European Symposium on Algorithms (ESA 2000)*, pages 338–345. Springer, 2000. 4.2

[47] Seth Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. In *Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms (SODA 2008)*, pages 1115–1124, 2008. 4.3

[48] Harald Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999. 2.1.3

[49] Mihai Pătraşcu. Hardness results for data structures. Theory seminar talk at Carnegie Mellon University, October 2008. 3.6

[50] Mihai Pătraşcu and Erik D. Demaine. Tight bounds for the partial-sums problem. In *In Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 20–29, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. 3, 3.6

[51] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. 3, 3.6

[52] Murray Sherk. Self-adjusting k-ary search trees. *Journal of Algorithms*, 19(1):25–44, 1995. 2.1.3

[53] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985. 1

[54] Daniel D. Sleator, Robert E. Tarjan, and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. In *Proceedings of the 18th ACM Symposium on Theory of Computing (STOC 1986)*, pages 122–135, New York, NY, USA, 1986. ACM. 3.4

[55] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985. 2.2, 3.6, 4.1, 4.3, 4.5, 5.2, 5.3, 6.4, 6.4

[56] Rajamani Sundar. On the deque conjecture for the splay algorithm. *Combinatorica*, 12(1):95–124, 1992. 4.3

[57] R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985. 4.3

[58] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, June 1977. 2.1.1

[59] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. 2.1.1

[60] Chengwen Chris Wang. *Multi-splay trees*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2006. Adviser-Daniel Sleator. 4.1, 7

[61] Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. O(log log n)-competitive dynamic binary search trees. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 374–383, New York, NY, USA, 2006. ACM. 1, 3.2, 4.1, 6.4, 7

[62] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989. 2.2, 2.2, 3, 3.1, 3.3, 4.5, 2

[63] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Information Processing Letters*, 17(2):81–84, 1983. 2.1.1