Complexity of Union-Split-Find Problems

by

Katherine Jane Lai

S.B., Electrical Engineering and Computer Science, MIT, 2007
S.B., Mathematics, MIT, 2007

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2008

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by .....
Erik D. Demaine
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Complexity of Union-Split-Find Problems

by

Katherine Jane Lai

Submitted to the
Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we investigate various interpretations of the Union-Split-Find problem, an extension of the classic Union-Find problem. In the Union-Split-Find problem, we maintain disjoint sets of ordered elements subject to the operations of constructing singleton sets, merging two sets together, splitting a set by partitioning it around a specified value, and finding the set that contains a given element. The different interpretations of this problem arise from the different assumptions made regarding when sets can be merged and any special properties the sets may have. We define and analyze the Interval, Cyclic, Ordered, and General Union-Split-Find problems. Previous work implies optimal solutions to the Interval and Ordered Union-Split-Find problems and an $\Omega(\log n/\log\log n)$ lower bound for the Cyclic Union-Split-Find problem in the cell-probe model. We present a new data structure that achieves a matching upper bound of $O(\log n/\log\log n)$ for Cyclic Union-Split-Find in the word RAM model. For General Union-Split-Find, no $o(n)$ bound is known. We present a data structure which has an $\Omega(\log^2 n)$ amortized lower bound in the worst case that we conjecture has polylogarithmic amortized performance. This thesis is the product of joint work with Erik Demaine.

Thesis Supervisor: Erik D. Demaine
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my thesis supervisor, Erik Demaine, for taking me on as his student and for our conversations that taught me much about research, teaching, academia, and life in general. I would also like to thank Mihai Pătraşcu for his contribution to this work. Thank you also to David Johnson, Timothy Abbott, Eric Price, and everyone who has patiently discussed these problems with me. Finally, I thank my family and friends for their support and their faith in my ability to do research.

Contents

List of Figures

Chapter 1

Introduction

While the Union-Find problem [6] is well-studied in the realm of computer science, we argue that there are several different but natural interpretations of an analogous "Union-Split-Find" problem. These different versions of the problem arise from making various assumptions regarding when sets can be merged and any special properties the sets may have. This thesis considers four natural interpretations of the Union-Split-Find problem. While tight bounds have previously been found for some versions of the Union-Split-Find problem, the complexity of other variations, especially the most general form of the problem, have not been well studied. This thesis is the product of joint work with Erik Demaine.

The Union-Find problem operates on disjoint sets of elements and requires three operations: *make-set*(x), *union*(x, y), and *find*(x). The *make-set*(x) operation constructs a new singleton set containing x. The *union*(x, y) operation merges the two sets of items that contain x and y, respectively, into a new set, destroying the two original sets in the process. Finally, the *find*(x) operation returns a canonical name for the set that contains x. Tarjan [20] showed that these operations can be supported in $\Theta(\alpha(m, n))$ amortized time, where $\alpha(m, n)$ is the inverse Ackermann function, and that this bound is optimal in the pointer-machine model. The inverse Ackermann function grows very slowly and for all practical purposes is at most 4 or 5. Fredman and Saks [10] showed a matching lower bound of $\Omega(\alpha(m, n))$ in the cell-probe model under the assumption that all machine words are of size $O(\log n)$.

The *General Union-Split-Find* problem adds support for a *split* operation that divides a set of elements into two sets, in addition to the original make-set, union, and find operations.

Here we require that the elements come from an ordered universe so that the split operation has an intuitive and efficient meaning. Specifically, $split(x, y)$ partitions the set that contains x into two new sets, the first containing the set of elements with value less than or equal to y, and the second containing the set of the elements with value greater than y. The original set that contained x is destroyed in the process. Despite the naturality of this general Union-Split-Find problem, it has not been studied before to our knowledge. Other, more restricted versions of the Union-Split-Find problem have been studied in the past. In this thesis, we define four natural variations: Interval, Cyclic, Ordered, and General Union-Split-Find.

1.1 Models of Computation

To analyze the complexity of these problems and their algorithms, we primarily consider three different models of computation.

The first model, the *pointer-machine model* [20], is the most restricted model we use. A pointer-machine data structure is represented by a directed graph of nodes each containing some constant number of integers and having a constant number of outgoing edges—*pointers*—to other nodes. Supported instructions in this model are generally of the form of storing data, making comparisons, and creating new nodes and pointers. The pointer machine is limited in that it is not able to perform arithmetic on the pointers. Since it is the weakest model we use, upper bounds in this model hold in the other two models as well.

Second is the *word Random Access Machine (RAM) model* [1, 11], one of the more realistic models for the performance of present-day computers. In a word RAM, values are stored in an array of machine words. Under the standard transdichotomous assumption, machine words consist of $w = \Omega(\log n)$ bits. The ith word in the array can be accessed (read or written) in $O(1)$ time. The word RAM model also allows arithmetic and bitwise operations (and, or, bit shifts, etc.) on $O(1)$ words in $O(1)$ time. Thus the running time of an algorithm evaluated in the word RAM model is proportional to the number of memory accesses and the number of machine-word operations performed.

Lastly, the *cell-probe model* [15, 22] is the strongest model of computation we consider. This model is like the word RAM model, except that all machine-word operations are free. The cost of an algorithm in the cell-probe model is just the number of memory accesses.

Because this model is strictly more powerful than all the previous models, any lower bound in this model applies to the other two models as well.

1.2 Interval Union-Split-Find Problem

Mehlhorn, Näher, and Alt [14] are the only ones to previously define a problem called "Union-Split-Find", but it is a highly specialized form which we call *Interval Union-Split-Find* to distinguish from the other variations under discussion. This problem is not the intuitive counterpart for Union-Find because it operates on intervals of values over a static universe of elements instead of arbitrary disjoint sets over a dynamic universe of elements. Only adjacent intervals can be unioned together, and there is no equivalent to the make-set operation required by the Union-Find problem. This problem is remarkable, however, in that the van Emde Boas priority queue [21] solves it optimally in $\Theta(\log \log n)$ time per operation.

Mehlhorn et al. [14] show that the elements and their set membership can be modeled as a list of marked or unmarked elements where the marked elements indicate the left endpoints of the intervals, as shown in the example in Figure 1-1. The union and split operations then correspond to $unmark(x)$ and $mark(x)$, which unmark or mark the endpoint x between the two affected sets. The $find(x)$ operation corresponds to finding the nearest marked element before or at x. This problem is thus equivalent to the classic dynamic predecessor problem over a linear universe, with the mark and unmark operations corresponding to inserting and deleting elements, and the find operation to the predecessor query. The van Emde Boas priority queue, as described by van Emde Boas, Kaas, and Zulstra [21], can perform the necessary operations and provides an upper bound of $O(\log \log n)$ time per operation in the pointer-machine model.

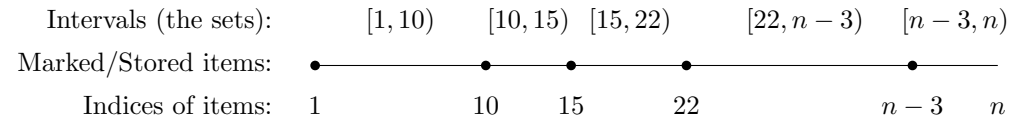

Figure 1-1: Interval Union-Split-Find: Intervals in the ordered finite universe can be represented by storing the left endpoints in a van Emde Boas priority queue.

In their analysis of the problem, Mehlhorn et al. [14] prove an $\Omega(\log \log n)$ lower bound for this problem in the pointer-machine model. Pătraşcu and Thorup [18] prove a stronger

result, showing that the van Emde Boas structure is optimal in the cell-probe model when the size of the universe is $O(n)$, which is the case here. Thus we have a tight bound of $\Theta(\log \log n)$ time per operation for the complexity of Interval Union-Split-Find.

1.3 Cyclic Union-Split-Find Problem

The *Cyclic Union-Split-Find problem* is a problem proposed by Demaine, Lubiw, and Munro in 1997 [8], as a special case of the dynamic planar point location problem. In the dynamic planar point location problem, we maintain a decomposition of the plane into polygons, or *faces*, subject to edge insertions, edge deletions, and queries that return the face containing a given query point. In the Cyclic Union-Split-Find problem, we support the same operations, but the faces are limited to a decomposition of a convex polygon divided by chords. Solving this computational geometry problem is equivalent to maintaining the set of points contained in each face of the polygon. Thus the union operation combines two adjacent faces by deleting a chord, and the split operation splits a face by inserting a chord. The find operation returns which face the given query point is in. Similar to Interval Union-Split-Find, Cyclic Union-Split-Find is a special case of General Union-Split-Find where restrictions are placed on which sets can be unioned together; for example, faces that are not touching cannot be unioned together with the deletion of a chord.

To indicate a canonical name for each set or face, we note that each face except one can be represented by a unique chord in the graph. Given that a particular face is represented by some chord c_i, if a new chord c_j is inserted that splits this face, we continue to represent one of these new faces with the same chord c_i and represent the other new face with the chord c_j. The one face that is not represented by a chord stems from the initial state of a convex polygon with no chords and one face.

We can transform the Cyclic Union-Split-Find problem into an equivalent problem by making a single cut at some point of the perimeter of the polygon and "unrolling" the polygon so that the problem becomes a one-dimensional line segment with paired endpoints preserved from the chords (see Figure 1-2 for an example). This graph can be represented by a balanced-parenthesis string as shown by Munro and Raman [16], with the two endpoints of each chord becoming the open and close symbols of a matching pair of parentheses. Balanced-parenthesis strings are strings of symbols "(" and ")" such that each "(" symbol

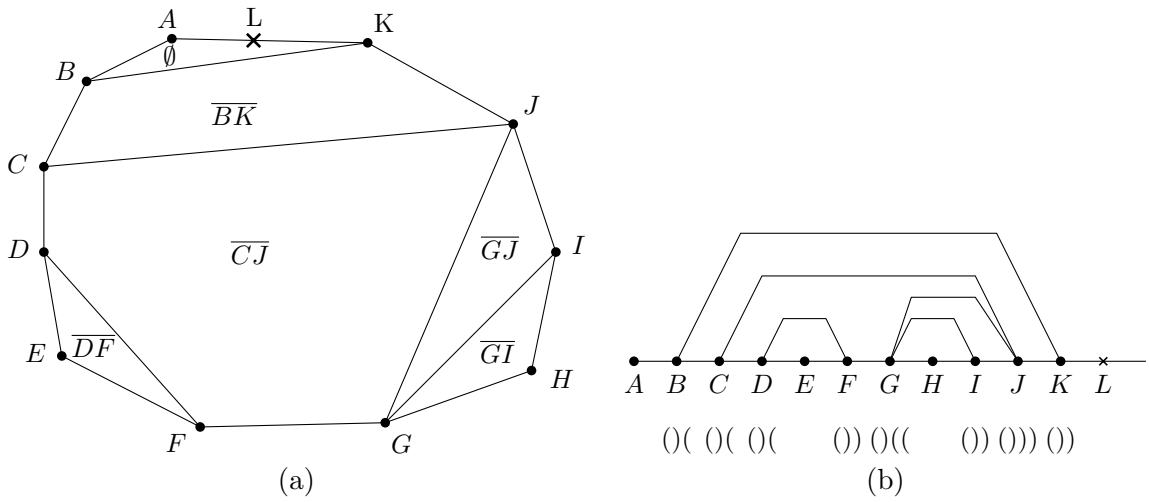

Figure 1-2: Cyclic Union-Split-Find: (a) A polygon with chords before it is unrolled; all faces except one can be uniquely represented by a neighboring chord. (b) The corresponding one-page graph and balanced-parenthesis string. The cut used to construct the one-page graph is located at the point L shown on the polygon in (a).

can be paired with a ")" symbol that appears after it and vice versa, and these pairs do not "cross". Thus "(()())()" is a valid string while "(()))(" is not. Thus the corresponding operations in this balanced-parenthesis problem are the following: inserting pairs of parentheses (split), deleting pairs of parentheses (union), and finding the immediately enclosing parent pair of parentheses of a given query pair (find).

We can observe that the Interval Union-Split-Find problem can be reduced to the Cyclic Union-Split-Find problem by representing the items and the boundaries of the sets, or intervals, as a balanced-parenthesis string. We represent each item as a pair of parentheses "()", and sets are denoted by additional pairs of parentheses that enclose the interval of pairs of parentheses that represent the set's items. Thus the union operation is equivalent to first deleting the appropriate pairs of parentheses that denote the two adjacent sets to be merged, and then inserting the new pair of parentheses that appropriately contains the newly merged set. The split operation is the reverse of the union operation and involves deleting the original set's delimiters and inserting two pairs of parentheses that denote the two new smaller sets. The find operation is simply the parent operation that returns the parentheses that encloses the query item's parentheses. This reduction implies that there is at least a lower bound of $\Omega(\log \log n)$ for Cyclic Union-Split-Find in all three models under consideration.

If we translate the balanced-parenthesis string into an array of integers by replacing each open parenthesis with a value of $+1$ and each close parenthesis with a value of -1, the parent operation is equivalent to solving a predecessor problem on a signed prefix sum array. For example, if a query pair's open parenthesis has value 3 in the array of prefix sums for the balanced-parenthesis string, the open parenthesis of the parent pair is located at the first parenthesis to the left of the query parenthesis that has a value of 2. The signed prefix sum array must be dynamically maintained during insertions and deletions of parentheses. Husfeldt, Rauhe, and Skyum [12] proved a lower bound of $\Omega(\log n / \log\log n)$ for the dynamic signed prefix sum problem and for the planar point location problem in the cell-probe model, where the word sizes were assumed to be $\Theta(\log n)$. The proof more generally establishes a lower bound of $\Omega(\log_w n)$ on Cyclic Union-Split-Find. It remained open whether this lower bound is tight. We develop a data structure in Chapter 2 that achieves a matching upper bound of $O(\log_w n)$ time per operation in the word RAM model.

1.4 Ordered Union-Split-Find Problem

Another interesting variation is the *Ordered Union-Split-Find* problem, or perhaps more appropriately, the *Concatenate-Split-Find* problem, where the problem operates on ordered lists or paths over a dynamic universe of elements. New elements can be created using the make-set operation. For this problem, $union(x, y)$ can be performed only when all items in the set containing x are less than all the items in the set containing y. Merging two sets that lack this ordering property would require multiple split and union operations. See Figure 1-3 for an example.

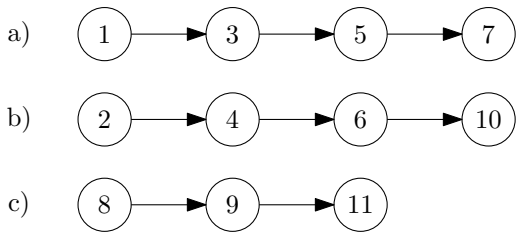

Figure 1-3: Ordered Union-Split-Find: Unlike Interval Union-Split-Find, sets of ordered lists with arbitrarily many interleaving values can be formed. List (b) in this example cannot be unioned with either of the other two because concatenation would not maintain the sorted order. However, list (a) can be unioned with list (c).

The Ordered Union-Split-Find problem generalizes both Interval Union-Split-Find and Cyclic Union-Split-Find. The Interval Union-Split-Find problem easily reduces to Ordered Union-Split-Find if we represent each interval or set as an actual ordered list of all its members. Two intervals in the Interval Union-Split-Find problem can be unioned only when they are adjacent to each other, so they can always be unioned by the union operation of Ordered Union-Split-Find. The split and find operations are trivially equivalent for the two problems.

Similarly, the operations required by the Cyclic Union-Split-Find problem can be implemented using those of the Ordered Union-Split-Find problem. In the balanced-parenthesis form of the Cyclic Union-Split-Find problem, a set is a list of parenthesis pairs that share the same parent. As such, Ordered Union-Split-Find can be used to maintain those sets as paths. In the Ordered Union-Split-Find problem, the equivalent of inserting a pair of parentheses in the Cyclic Union-Split-Find problem is taking the ordered list that contains the set denoted by the new pair's parent, splitting it at the locations of the parenthesis insertions, and concatenating the two outer pieces of the original list (if they have nonzero length) and the new list together in order of appearance in the balanced-parenthesis string. This in effect splits off a new set formed out of a subsequence in the original set. A deletion of a pair of parentheses translates to the reverse operations: one split operation at the location where the child set will be inserted into the parent set, another split and deletion for the element corresponding to the pair of parentheses to be deleted, and up to two union operations to concatenate the three remaining sets together. Performing the parent operation is simply a find operation as before.

Because the sets are ordered lists and can be thought of as paths, we can easily represent the Ordered Union-Split-Find problem as that of maintaining paths of elements that may be concatenated or split. The find operation is thus the same as returning the path containing the query element. Together, these operations support the problem of dynamic connectivity on disjoint paths. Dynamic connectivity is the problem of maintaining connected components of undirected graphs, subject to three operations: inserting edges, deleting edges, and testing whether two vertices are connected. Pătraşcu and Demaine [17] prove a lower bound of $\Omega(\log n)$ on dynamic connectivity in the cell-probe model that applies even for graphs composed of only disjoint paths; this bound is therefore a lower bound on the Ordered Union-Split-Find problem. Link-cut trees, data structures first described by

Sleator and Tarjan [19], support operations on a forest of rooted unordered trees in $O(\log n)$ time. In particular, they support *make-tree*(), *link*(v, w), *cut*(v, w), and *find_root*(v), which correspond to the Ordered Union-Split-Find operations of make-set, union, split, and find, respectively. In fact, because we only need to maintain paths rather than rooted unordered trees, we can also just use the special case of link-cut paths, which are the building blocks of link-cut trees. Link-cut trees and paths work in the pointer-machine model. Thus, in all three models, the upper and lower bounds for Ordered Union-Split-Find match, yielding a tight bound of $\Theta(\log n)$ time.

As discussed earlier, the Interval and Cyclic Union-Split-Find problems both reduce to the Ordered Union-Split-Find problem. The Ordered Union-Split-Find problem is not only a generalization of both those problems, but it is also a step closer to the most general version of the Union-Split-Find problem. It supports the make-set operation which allows the number of managed elements to get arbitrarily large without rebuilding the entire data structure. It also allows for the existence of different sets occupying the same interval of values as seen with the first two sets in Figure 1-3. It is not completely general, however, because the union operation can only operate between the end of one path and the beginning of another, and the split operation preserves the order of the paths. This can be useful for certain applications where order is meaningful and needs to be preserved, but it does not easily support the more general idea of splitting an arbitrary set on the basis of the elements' associated values.

1.5 General Union-Split-Find Problem

To our knowledge, the General Union-Split-Find problem has not explicitly been studied previously, even though it is the most natural interpretation of the Union-Split-Find problem. Like the Union-Find problem, it maintains arbitrary sets of disjoint elements. The sets can be created through any sequence of *make-set*(x), *union*(x, y), and *split*(x) operations, starting from the empty collection of sets. The make-set, union, and find operations function exactly as in the original Union-Find problem. The split operation splits a given set by value. The difference between Ordered and General Union-Split-Find is that the union operation now needs to be capable of merging two arbitrary disjoint sets of elements. Referring back to the example of Ordered Union-Split-Find in Figure 1-3, there is now no

restriction on which two sets can be unioned together—even if the interval of values represented in one set overlaps completely with the interval of the other. All of the previously mentioned variations of the Union-Split-Find problem can be reduced to this generalized problem, so it too has a lower bound of $\Omega(\log n)$ per operation.

There is also clearly a trivial $O(n)$ upper bound per operation: simply store a set's elements in an unordered linked list. The union operation is then simply a concatenation of the two sets' lists in $O(1)$ time, and make-set is still an $O(1)$ time operation. The split operation can be performed in $O(n)$ time by simply partitioning a set around the value used for the split via a linear scan through the set's items, and the find operation can also be performed in $O(n)$ time by scanning an entire set's list for its minimum element.

While the General Union-Split-Find problem may not have been explicitly studied before, others have studied the related problem of merging and splitting sets of items stored in ordered B-trees. The make-set operation creates a new tree in $O(1)$ time, the find operation can be performed in $O(\log n)$ time to traverse the B-tree from a leaf to the root, and splitting B-trees has been shown to take $O(\log n)$ time [13, pages 213–216]. The main open question is the amortized cost of union operations. The union operation may have to merge two sets of items with exactly alternating values, resulting in an $O(n)$-time operation. On the other hand, the operations take $O(\log n)$ to concatenate two trees in the case where all items in one tree are less than all the items in the other tree, as in Ordered Union-Split-Find. Demaine, López-Ortiz, and Munro [7] generalize Mehlhorn's work by solving the problem of merging arbitrarily many sets, and they prove tight upper and lower bounds for the problem.

It remains open whether General Union-Split-Find can be solved using operations that take $O(\log n)$ amortized time, which would prove the known lower bound tight. In Chapter 3, we present a data structure for which there exists an expensive sequence of operations that forces the data structure to take $\Theta(\log^2 n)$ amortized per operation; this sequence can be repeated any number of times, so there is a lower bound of $\Omega(\log^2 n)$ amortized in the worst case. We conjecture that this data structure has polylogarithmic amortized performance.

1.6 Known Lower and Upper Bounds

Table 1.1 summarizes the best known bounds for each of the variations of the Union-Split-Find problem. Both the Cyclic and the General Union-Split-Find problems have not been specifically studied before.

Union-Split-Find	**Lower bound**	**Upper bound**
Interval	$\Omega(\log\log n)$ [14, 18]	$O(\log\log n)$ [14]
Cyclic	$\Omega(\log_w n)$ [12]	$O(\log_w n)$ (new)
Ordered	$\Omega(\log n)$ [17]	$O(\log n)$ [19]
General	$\Omega(\log n)$ [17]	$O(n)$ (obvious)

Table 1.1: The best known bounds for the four variations of the Union-Split-Find problem. The upper bound for the Cyclic problem applies only to the word RAM model (and thus to the cell-probe model as well). All other bounds apply to all three models of computation under discussion.

Chapter 2

Cyclic Union-Split-Find Problem

Recall that the unrolled linear form of the Cyclic Union-Split-Find problem requires the following operations on a balanced-parenthesis string: *insert*, *delete*, and *parent*. Insertions and deletions of pairs of parentheses must maintain the invariant that the resulting string remains balanced. The pair of parentheses inserted or deleted must also form a matching pair in the context of the entire balanced-parenthesis string. The parent operation returns the immediately enclosing pair of parentheses of a query pair.

The language of balanced-parenthesis strings is also classically known as the Dyck language. The problem of simultaneously handling insertions, deletions, and parent queries is one of several natural dynamic problems for the Dyck language. Frandsen et al. [9] discuss the various dynamic membership problems of maintaining balanced-parenthesis strings subject to operations that modify the string and check whether the string is a valid balanced-parenthesis string. They establish a lower bound of $\Omega(\log n / \log \log n)$ in the word RAM model for any data structure solving the related problem of supporting insertions, deletions, and a *member* operation for checking whether the string is in the language. Frandsen et al. also establish a lower bound of $\Omega(\log n / \log \log n)$ in the bit probe model (the cell-probe model wherein the cell sizes are of 1 bit each) for the related problem of supporting the operations of changing a single character in the string and testing the string's membership in the language. Alstrup, Husfeldt, and Rauhe [2] give a data structure that solves the latter of these two problems optimally in $\Theta(\log n / \log \log n)$ in the word RAM model. Their data structure is essentially a balanced tree of degree $O(\log n)$ that stores each parenthesis symbol in the leaves of the tree. We use a similar data structure to solve the Cyclic

Union-Split-Find problem.

Theorem 1. *Under the standard transdichotomous assumption that the word size is* $\Omega(\log n)$*, the Cyclic Union-Split-Find problem can be solved in* $\Theta(\log_w n)$ *amortized time per operation.*

We achieve this result by modifying an existing data structure. We store the balanced-parenthesis string in the leaves of a strongly weight-balanced B-tree, a data structure first proposed by Arge and Vitter [3] and further improved by Bender, Demaine, and Farach-Colton [4]. The key to achieving the desired time bound is setting the branching factor to be $O(w^\varepsilon)$ for some constant ε where $0 < \varepsilon < 1$. There will therefore be $O(\log_{w^\varepsilon} n) = O(\log_w n)$ levels in the B-tree.

The key to supporting the parent operation quickly in this tree is the following property.

Property 1. *Given the position* i *of the query pair's open parenthesis "(" in the balanced-parenthesis string, finding the open parenthesis of the parent pair is equivalent to finding the last unmatched "(" parenthesis in the first* $i - 1$ *characters of the string.*

Using this property, we store at each internal node a succinct representation of the unmatched parentheses of the substring stored in the subtree rooted at that node. Given a query pair of parentheses, the parent operation thus walks up the tree from the leaf storing the open parenthesis of the query pair, recursively performing essentially a predecessor query at each internal node, and stopping once evidence of the parent parenthesis has been found. (An equivalent algorithm would be obtained by starting from the close parenthesis and performing successor queries.)

We can observe that all queries originating from the children of the parent pair of parentheses stored at the leaves u and v will terminate by the time the algorithm reaches the least common ancestor (LCA) of u and v. This means that nodes above their LCA do not need to know about u or v. For convenience, we will overload terminology and define the LCA of a single node or parenthesis to be the LCA of that node and the leaf that contains the parenthesis it matches in the string. We also define the *Parent's LCA* (or PLCA) to be the LCA of a query pair's parent pair of parentheses. Note that to support the parent operation at any internal node x, we only need to maintain data for parentheses below it that have their LCA at x or at some ancestor of x. Enough of this data can be compressed into $O(1)$ machine words such that the parent operation only needs to spend $O(1)$ time at

each node to query the necessary data, resulting in a total running time of $O(\log_w n)$, the number of levels in the B-tree.

We first discuss what additional information we store and maintain in the data structure and how the parent operation is supported. We then show that all this information can still be maintained with insertions and deletions that take $O(\log_w n)$ time.

2.1 Data Structure

2.1.1 A Balanced-Parenthesis String

In our representation of the balanced-parenthesis string, it will be useful to collapse consecutive parentheses of the same orientation into one symbol of that orientation. For the sake of clarity, we shall henceforth refer to a single character in the original balanced-parenthesis string as a *parenthesis*, and we shall refer to a character that represents some number of consecutive parentheses of the same orientation as a *symbol*. The number of parentheses represented by a symbol is the *weight* of the symbol.

At each internal node v, we store an $O(w^{\varepsilon})$-size string that contains a representation of the unmatched symbols from each child subtree of v. The unmatched symbols from a child subtree represent the unmatched parentheses in the string represented by that subtree. These unmatched parentheses must match parentheses in some other subtree. See Figure 2-1 for an example. For compactness, consecutive parentheses of the same orientation can be represented as a single symbol.

Using this scheme, we can observe that a parenthesis is represented in a symbol at each internal node on the path from its leaf node to its LCA. At the LCA, the parentheses are matched, so they are not represented in any higher nodes.

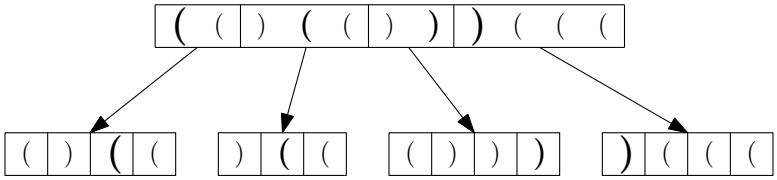

Figure 2-1: The unmatched symbols in the strings represented by the child subtrees are propagated up to the parent (in this figure, they are not yet compressed). If the query is the medium-sized pair of parentheses, the large pair of parentheses is the parent pair.

The actual string that is stored at the node is a balanced-parenthesis string with weighted symbols—insofar as the string can be balanced. To compute this string, we perform the following steps.

1. Concatenate the unmatched parentheses from the node's children.
2. Collapse identical consecutive symbols that are from the same child subtree into a single compressed symbol. Symbols from different child subtrees are not to be mixed.
3. Split up the compressed symbols appropriately such that the string is balanced with respect to the weights of the symbols. If there are dangling symbols that cannot be balanced in this string, they are left in the string in their compressed state.

It should be noted that for the most part, the weights of the final symbols in the string are not stored in any fashion. Only the weights of the unmatched symbols are maintained and propagated up the tree to calculate the balanced weighted string in the parent subtree. In Figure 2-2, we see an example of this calculation using the previous example shown in Figure 2-1.

Represented string:	(	(	)	(	(	)	)	)	(	(	(
Compressed child substrings:	$($[2]		$)$[1]	$($[2]		$)$[2]		$)$[1]	$($[3]		
Final balanced string:	$($[1]	$($[1]	$)$[1]	$($[2]		$)$[2]		$)$[1]	$($[3]		
String that will be passed up to the parent:									$($[3]		

Figure 2-2: The example here continues the example from Figure 2-1 and illustrates the calculation of a balanced weighted string. The actual weights themselves are not stored.

Lemma 1. *Calculating the balanced weighted string at each internal node takes $O(w^\varepsilon)$ time, and the string will have length $O(w^\varepsilon)$.*

Proof. After the compression step, the string clearly has length $O(w^\varepsilon)$ because a node has $O(w^\varepsilon)$ child subtrees, and each child subtree will contribute at most one ")" symbol and one "(" symbol, in that order. If it contributed any other symbols, the symbols would either result in at least a partial match and would not be included in the string, or they would have the same orientation as the other unmatched symbols and could be compressed with them.

In the splitting step, the string does not grow by more than a constant factor. We can see that this is true through a simple splitting algorithm and its invariant.

To split a compressed string, we seek through the string until we find the first ")" symbol. If it is a leading ")" symbol, then it does not match anything, and we can mark it as *done*. If it is preceded by a "(" symbol, then we can make at least a partial match. If one of them represents more parentheses than the other, we split the larger one up so that one piece of it perfectly matches the other symbol, and we have a leftover unmatched symbol. The symbols that have been perfectly matched are marked as *done* and are ignored from then on. If the two symbols are already of equal weight and can be perfectly matched, both are marked *done* and no splitting occurs. We continue to process the ")" symbols until no more symbols can be matched (there are only "(" symbols left), taking care that we only match symbols that are adjacent in the string (skipping over symbols that have been marked *done*) and are not yet marked *done*. Matching adjacent symbols ensures that we are not mismatching symbols.

With this algorithm, we can observe a simple invariant: whenever we match symbols, more symbols are marked *done* than are added to the final string. When we perform a split, one symbol is added to the string, and two symbols are marked as *done*. If we simply match two symbols, no symbols are added, and two symbols are marked *done*. For example, if we had "$(^2$ $)^1$" and had to split the "(", we know that after we do so, the ")" and the resulting new "(" will never cause any other symbols to split, and will not be splitting for any reason because they are now perfectly matched. Thus with each step of the algorithm, the number of symbols that are not marked *done* strictly decreases by at least 1, and at most one new symbol is added to the string. Since we start with $O(w^\varepsilon)$ symbols that are not marked *done*, we add at most $O(w^\varepsilon)$ symbols to the original compressed string, so we only get a constant factor increase in the number of symbols in the string. By the same counting argument, the algorithm also takes only $O(w^\varepsilon)$ time as long as it only takes $O(1)$ time to find the next symbols to split.

We can take only $O(1)$ time to find a ")" symbol by representing the string as a bit string consisting of 0's for "(" and 1's for ")" and finding the most significant bit (MSB). Fredman and Willard [11] show that finding the MSB on a bit string can be done in $O(1)$ time in the word RAM model as long as the string fits into a constant number of words. We achieve this restriction as long as $0 < \varepsilon < 1$. Secondly, when symbols are marked *done*, we can simply mask, shift, and OR the machine word to itself to cut out the old symbols from consideration in $O(1)$ time. Thus we get a running time linear in the length of the

string, or $O(w^{\varepsilon})$.

□

2.1.2 Prefix Sums Array

Along with the balanced-parenthesis string, an internal node also stores a representation of the same string as an array of prefix sums. A prefix sum at a position i in the array corresponds to the sum of the values for all the symbols of the string from the beginning of the string to position i in the string. We let each "(" symbol contribute +1 and each ")" symbol contribute -1, and we also increase all the sums by a positive offset if necessary so that all values are nonnegative. Thus the prefix sum array of the balanced-parenthesis string ")((()()))(" is $[1, 2, 3, 2, 3, 2, 1, 0, 1]$.

Since there are $O(w^{\varepsilon})$ symbols in the string, the maximum value for a prefix sum is $O(w^{\varepsilon})$, so the maximum number of bits necessary to represent a prefix sum is $O(\varepsilon \log w)$, and there are $O(w^{\varepsilon})$ such sums. This brings a total of $O(\varepsilon w^{\varepsilon} \log w)$ bits to represent an array of prefix sums for the entire string. We can fit that entire array into a word if $w = \Omega(\varepsilon w^{\varepsilon} \log w)$. This is true for any ε where $0 < \varepsilon < 1$. Along with the actual sums, we can pad the string so that we leave an extra bit between all the values and still fit the entire string in a word. This extra padding will be useful for parallel computation later on.

2.1.3 Summary Data

As mentioned earlier, each internal node also stores the actual total weights of the unmatched parentheses represented in the balanced-parenthesis string. In the example shown in Figure 2-1, if we assume that each symbol in the bottom level represents exactly one parenthesis, then the parent internal node stores the pair of numbers $(0, 3)$, for the zero unmatched ")" parentheses and the three unmatched "(" parentheses in the string. Both these numbers require $O(\log n)$ bits, so both should fit into one machine word each for $O(1)$ space per internal node. We keep no information on the weights of the matched parentheses.

2.1.4 Auxiliary Pointers

In addition to the balanced-weighted string at each internal node, we also store a set of auxiliary pointers. There is one auxiliary pointer for each of the symbols in the balanced weighted string. The auxiliary pointer for a given symbol points to the leaf that stores

the innermost parenthesis of the symbol. The innermost parenthesis of a ")" symbol is the leftmost ")" parenthesis represented, and the innermost parenthesis of a "(" symbol is the rightmost "(" parenthesis represented. The example in Figure 2-3 illustrates the auxiliary pointers.

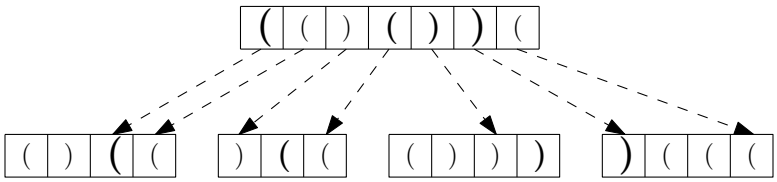

Figure 2-3: The dashed pointers here point to what would be considered the corresponding innermost parentheses. The actual auxiliary pointers point to the leaves that store the targeted parentheses (not shown here). The pointers that exist as part of the structure of the B-tree are not shown and are not changed.

2.1.5 Space Analysis

Storing the extra data at each internal node takes extra storage. The balanced-parenthesis string, prefix sum array, and the summary data take up a constant number of words at each node. This extra data should not affect the complexity of the regular B-tree operations as long as they can be updated quickly. As for the auxiliary pointers, since there is one per symbol in the balanced-parenthesis string, there are $O(w^{\varepsilon})$ pointers per internal node. This is the same number of B-tree pointers at each node since the branching factor is also $O(w^{\varepsilon})$. Again, as long as the data can be updated quickly, the complexity of the B-tree operations will not be affected. In total, the entire data structure takes up only a constant factor more space than the same underlying B-tree.

2.2 Parent Operation

We can now use this new data structure to support the parent operation. As described earlier, the operation first seeks to the leaf of the open parenthesis of the query pair, using $O(1)$ time to do so. It then traverses up towards the root of the tree, testing each internal node to find an open unmatched symbol to the left. Once it is found, it follows the symbol's auxiliary pointer, which points to the innermost and thus last unmatched parenthesis that occurs before the query pair.

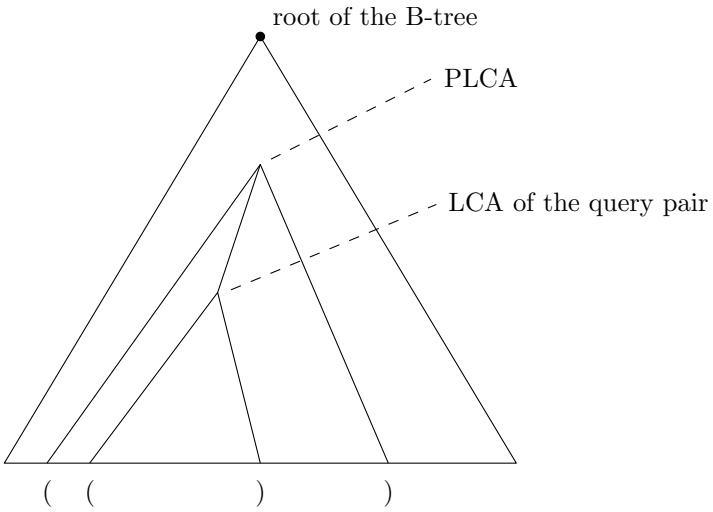

Figure 2-4: In the worst case, the path that a query follows starts from a leaf containing the query pair parenthesis, goes up to their LCA, and then up towards the root until either the PLCA or the leaf-to-root path of the left parent parenthesis is found.

The data stored at each internal node in our data structure allows an efficient algorithm for finding the symbol representing the parent parenthesis at each internal node if it is there. The stored balanced-parenthesis string at an internal node has been balanced in terms of weight, so finding the parent in that string given a query position is equivalent to finding the parent in a string where each symbol has the same weight.

Lemma 2. *We can perform the parent operation on a balanced-parenthesis string of length* $O(w^{\varepsilon})$ *in* $O(1)$ *time.*

Proof. We can view the parent problem in this string as solving a prefix sums problem and use the corresponding prefix sums array for this purpose. When searching for the first unmatched "(" to the left of the "(" symbol of the query pair, we are searching for the first symbol to the left for which the prefix sum is 1 less than the prefix sum of the query symbol. Thus when we search for the parent of the query pair that is represented by the third and fourth symbols in the string ")(()()))(" (with prefix sum array $[1, 2, 3, 2, 3, 2, 1, 0, 1]$), we search for the first value of $3 - 1 = 2$ to the left of the query pair, yielding the second symbol.

Because the entire array fits inside one machine word, we can use machine-word operations to compare values in parallel. We can subtract the query symbol's prefix sum from all the prefix sums in the array in parallel, mask the side of the array we're looking at,

Prefix Sums	1 01	1 10	1 11	1 10	1 11	1 10	1 01	1 00	1 01
Query Prefix Sum	0 10	0 10	0 10	0 10	0 10	0 10	0 10	0 10	0 10
Subtracted Result	0 11	1 00	1 01	1 00	1 01	1 00	0 11	0 10	0 11
NOT of result	1 00	0 11	0 10	0 11	0 10	0 11	1 00	1 01	1 00
Mask	0 00	0 00	0 00	0 00	1 00	1 00	1 00	1 00	1 00
Masked Result	0 00	0 00	0 00	0 00	0 00	0 00	1 00	1 00	1 00

Figure 2-5: The example here uses the balanced-parenthesis string ")((()))(" and searches for the first ")" symbol to the right of the fourth symbol. Taking the index of the most significant bit of the final masked result gives us the location of the ")" symbol of the parent pair.

and find the first location for which a carry was needed for the subtraction, which is then the location where the prefix sum is first less than the query prefix sum. The first prefix sum that is less than some query value is necessarily one less than the query value because adjacent prefix sums always differ by exactly 1. See Figure 2-5 for an illustration of the sequence of operations. All of these operations take $O(1)$ time, so the claim is proved. □

We can use this result to attempt to find the parent pair at each internal node. The only tricky details left are that of the query position to be used in the balanced-parenthesis string when traveling to a new internal node.

Lemma 3. *The parent operation can be performed in $O(\log_w n)$ time.*

Proof. As stated earlier, the parent operation starts at the leaf containing the query parenthesis and traverses the tree towards the root. When the parent operation traverses up a level in the B-tree and has not yet reached the LCA of the query pair, it performs the query on the balanced-parenthesis string using the leftmost (unmatched) "(" from the contribution of the child subtree it just traversed. This outermost "(" parenthesis represented by this symbol must be that of the query pair because otherwise the "(" parent parenthesis would have been represented in this symbol and would have been found earlier by the algorithm in a previous level of the B-tree.

When the algorithm reaches the LCA of the query pair and continues further up the B-tree, the predecessor queries can continue to be made. Since the parent pair has still not been found after querying the node of the query pair's LCA, we know that none of the unmatched "(" symbols of the subtree rooted at the LCA are part of the parent pair. At this point, it should instead use the query position of directly after the unmatched ")" symbol contribution of the child subtree it just traversed. The rightmost parenthesis of

that symbol must belong to a sibling pair of parentheses since the open parenthesis of the parent has not yet been found (and therefore the parent pair also directly encloses this pair of parentheses).

Once we find a parent symbol, we simply follow its auxiliary pointer and jump directly to the leaf containing the parent parenthesis in $O(1)$ time. Since the auxiliary pointer points to the innermost parenthesis, it points to the rightmost unmatched "(" parenthesis to the left of the query pair.

Using the result from Lemma 2, we know this algorithm spends $O(1)$ at each internal node on the path from the query parenthesis towards the root. We are also guaranteed to eventually find evidence of the parent symbol. When that happens, we perform $O(1)$ work to actually find the parent parenthesis. Since the path from a leaf to the root of the tree can be at most the height of the tree, this algorithm takes $O(\log_w n)$ time in total. □

2.3 Insertions and Deletions

To insert or delete a pair of parentheses, we first perform the standard B-tree operations of inserting or deleting leaves. In addition, the auxiliary data at all the internal nodes must be updated. After these updates are done, we rebalance the B-tree if necessary, resulting in additional costs for splitting and merging nodes.

2.3.1 Incremental Updates

When inserting or deleting pairs of parentheses, we need to update all the internal nodes that are on the paths from the new/deleted leaves to the LCA of those leaves. We start from the locations of the affected leaves and walk up simultaneously towards their LCA.

Lemma 4. *Updating all internal nodes in the B-tree given an insertion or deletion of a pair of parentheses takes $O(\log_w n)$ time.*

Proof. At the parents of the new or deleted leaves, we simply insert or delete the symbol from the balanced-parenthesis string. Updating the array of prefix sums can be done easily by using shifting and adding operations to make room for the new value or delete the old value, and adding or subtracting 1 to or from all values subsequent to the new parenthesis's position as appropriate. If the auxiliary pointers it passes up to the next level are affected,

then these are recalculated in $O(1)$ time using machine word operations on the balanced parenthesis string.

Before we reach the LCA, we know that the parentheses will not match any of the other parentheses in the balanced-parenthesis strings. Thus for each internal node on the path from the leaf to the LCA (except that of the LCA), we simply increment or decrement the weight of the unmatched symbol passed from an internal node to its parent. If a node previously did not have an unbalanced symbol of the orientation of an inserted parenthesis, the balanced-parenthesis string at its parent will need to be updated with a new symbol. A similar update must occur if a deletion removes an unbalanced symbol from the representation at a particular internal node. Any changes to the balanced-parenthesis string result in similar changes to the prefix sum array and the addition or deletion of the appropriate auxiliary pointers. All these updates can occur in $O(1)$ time per node.

When the two paths meet at the LCA, we insert two matching symbols into the balanced-parenthesis string. There are two cases: either both the symbols merge with existing symbols, or they must be inserted as entirely new symbols. It is impossible for exactly one of them to merge with a symbol because the two must match in the represented string in symbols of equal weight. The prefix sum array again needs to be modified. In particular, we add 1 to all the positions starting with the inserted "(" symbol and ending with the symbol directly before the inserted ")" symbol. The auxiliary pointers are updated if necessary. Deletions are processed in much the same way with just the operations reversed.

The ancestors of the LCA of the affected leaves do not need to be touched because the parentheses have been matched, and we are done. Touching a node only requires $O(1)$ time, and each insertion or deletion touches $O(\log_w n)$ nodes for the paths from the leaves to their LCA. Costs of rebalancing the strongly weight-balanced B-tree has been shown to be $O(1)$ amortized per insertion or deletion. Thus the total cost of inserting or deleting a pair of parentheses is $O(\log_w n)$ as desired. □

2.3.2 Splits and Merges of B-Tree Nodes

Insertions may cause splitting of nodes if the nodes are too full, and deletions may cause nodes to merge if they are too small. Such updates can be costly if updating the extra data at each internal node is too expensive or if they are forced too happen too often. We show that this is not the case.

Lemma 5. *A split or merge of an internal node costs $O(w^\varepsilon)$ time, which can then be amortized to $O(1)$ time per insertion and deletion.*

Proof. In a strongly weight-balanced B-tree with a branching factor of $O(w^\varepsilon)$, any non-root node will undergo $\Omega((w^\varepsilon)^h)$ insertions or deletions before it is split or merged again, where h is the height of the node with leaves defined to be at height 1 [4]. When processing a split or merge, we recalculate the necessary balanced-parenthesis strings in the new nodes by using the summary data from their respective children. As shown earlier in Section 2.1.1, it takes $O(w^\varepsilon)$ time to calculate a balanced-parenthesis string. Calculating the prefix sum arrays takes time linear in the length of the balanced-parenthesis strings, so it also takes $O(w^\varepsilon)$.

Auxiliary pointers will also need to be updated. This can require information that is not immediately available at the internal node. If a weighted symbol had to be split in a new node to make the string balanced, we must now find a pointer to the innermost parenthesis of the newly created symbol. We have the index of the unmatched parenthesis within the larger symbol, and so we can navigate down the B-tree, recalculating the relative index of the parenthesis as parts of the original symbol branch off into other subtrees.

In navigating down the B-tree, we use linear time in the branching factor to discover which child subtree contains the desired parenthesis. Thus to find a new auxiliary pointer, it will take $O(hw^\varepsilon)$. At most $O(w^\varepsilon)$ auxiliary pointers will need to change, so it will take $O(h(w^\varepsilon)^2)$ time to create the necessary pointers for a modified node.

In addition to the changes internal to the split or merged nodes, we must update their parent node with correct pointers to its modified children and with the changes to its compressed balanced-parenthesis string. Recall from Section 2.1.1 that no compression of symbols is allowed between symbols contributed by two different child subtrees; this invariant will need to be preserved. No other node above the parent node will be affected because there will be no change in what symbols the subtree represents.

Below is a list of all possible changes that will need to occur because of a split:

1. An unmatched symbol will split, and nothing else in the balanced-parenthesis string is affected.

2. A matched symbol will split, and the other matching symbol in the string will also need to be split.

3. Symbols that used to match in the original child subtree are now split into two sibling subtrees, so new symbols need to be added in the string. This may cause another symbol to split, in which case we will perhaps have to split a matching symbol. For example, if a node that has the string ")())(" is split, its parent would have seen ")^2(" before the split, and after the split, the parent now has two children with ")(" and "))(", and none of the symbols will be compressed together because the string needs to stay balanced.

In the first case, only a constant number of simple machine word operations are needed to insert the new symbol, and all is done. The auxiliary pointer can just be passed up from the child subtree. In the second and third cases, it is possible to check if another symbol needs to be split in $O(1)$ time by querying the prefix sum array for the matching symbol. If a matching symbol is found, then we split it, update the prefix sum array, and find its new auxiliary pointer as before. Since we only need to find at most one new auxiliary pointer at a height of only one more than the original node, this does not affect the runtime of the split.

There are similar opposite changes when processing merges, but no new auxiliary pointers will need to be found because merging symbols will only eliminate auxiliary pointers.

In its entirety, processing a split or merge takes $O(h(w^\varepsilon)^2)$ time. This is less than or equal to $O((w^\varepsilon)^h)$ time for all h except 1. Since the nodes that have height 1 are defined as the leaves, they will never be split or merged. Thus each valid split or merge will take $O((w^\varepsilon)^h)$ time, and as a result, we can amortize the entire cost of splits or merges to be $O(1)$.

□

2.4 Proof of Tight Bounds

Having explored the low-level details for implementing insert, delete, and parent, we can now prove Theorem 1.

Proof. Lemma 4 shows that the updates required by insertions and deletions take $O(\log_w n)$. B-tree nodes need to be merged or split rarely enough that this adds only $O(1)$ amortized work per insertion or deletion by Lemma 5. Insertions and deletions in the B-tree therefore take $O(\log_w n)$ amortized.

Lemma 3 shows that parent takes $O(\log_w n)$.

In addition, we have a lower bound of $\Omega(\log_w n)$ on the parent operation because of the lower bound on the signed prefix sum problem found by Husfeldt, Rauhe, and Skyum [12], so we have overall tight bounds $\Theta(\log_w n)$ amortized for all three operations.

□

2.5 Complexity

We are able to achieve a tight bound of $\Theta(\log_w n)$ amortized time per operation for the Cyclic Union-Split-Find problem in the word RAM model and in the cell-probe model by implication. There are no known tight bounds for the pointer-machine model. Given that both the lower and upper bounds in the word RAM model are strongly dependent on the ability to perform arithmetic on pointers and machine words, solving the problem on the pointer machine is probably computationally harder. From what we know of the problem in other models and of the Ordered Union-Split-Find problem, we have a lower bound of $\Omega(\log n / \log \log n)$ and an upper bound of $O(\log n)$.

Open Problem 1. *What is the complexity of Cyclic Union-Split-Find in the pointer-machine model?*

Chapter 3

General Union-Split-Find Problem

In contrast to all other Union-Split-Find problems discussed thus far, the General Union-Split-Find problem is most like the original Union-Find problem and maintains arbitrary disjoint sets of elements. The sets can be created through any sequence of *make-set*(x), *union*(x, y), and *split*(x) operations, and the query *find*(x) returns a name or representative for the set containing the query x. The only additional assumption used to formulate the General Union-Split-Find problem is that the elements are from an ordered universe. The *split*(x) operation partitions a given set into two new sets by the value x.

As the most general version of all the other problems, General Union-Split-Find inherits the lower bound of $\Omega(\log n)$ amortized from Ordered Union-Split-Find. It remains open whether this bound is tight. We give an algorithm that has a stronger lower bound of $\Omega(\log^2 n)$ amortized in the worst case and has no known upper bound other than the trivial $O(n)$, though we conjecture that there is a polylogarithmic upper bound.

At a high level, we store a given set's elements in an augmented B-tree. To perform the required operations, we simply perform the requisite tree operations that maintain order in the newly formed sets. Our algorithm is essentially identical to those of Mehlhorn [13] and Demaine et al. [7]. Our main contribution here is the amortized analysis of this algorithm's performance under the conditions of the General Union-Split-Find problem.

3.1 Data Structure

We use a *level-linked tree*, first described by Brown and Tarjan [5] and later also as *finger trees* by Mehlhorn [13, page 228], to store the elements for each set. A level-linked tree is a

B-tree that is augmented such that each node has *level links*, or pointers to its two neighbors on its level in the tree (crossing into other subtrees if necessary). In addition to the B-tree, we keep a *finger*, or a pointer to some leaf in the tree, for each set. The finger is sometimes used as an alternative starting point for searches in favor of the root, and it is used to take advantage of locality of reference. Figure 3-1 shows an example of a level-linked tree and its finger.

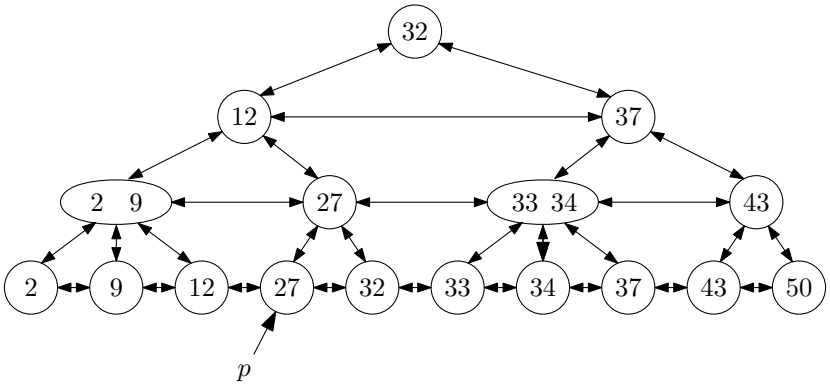

Figure 3-1: A level-linked tree: edges connect all nodes to their neighbors, crossing boundaries for different subtrees. Here, a single finger or pointer p is shown pointing to the leaf containing the value 27.

The make-set operation simply creates a new level-linked tree with the value x at the root in $O(1)$ time. The find operation traverses from the location of x in its tree to the root of the tree to find the representative of the set in $O(\log n)$ time. The split operation can be performed using a normal tree splitting operation that takes $O(\log n)$ time [13, pages 213–216]. The union operation is more difficult because we require that the merged set is in sorted order.

Union

To merge two level-linked trees of items A and B, we use an algorithm given by Demaine et al. [7]. We first break the trees into a minimal set C of smaller level-linked trees such that each item in $A \cup B$ is represented in some tree, each tree contains only items from A or only items from B, and the ranges of items represented by any two level-linked trees in C do not overlap. To visualize these sets, we can draw the items for each level-linked tree along the real line (see Figure 3-2 for an example). Then C is formed by minimizing the number of line segments needed to cover all items while maintaining that no two line

segments occupy the same range of values. To create the single level-linked tree with all items in $A \cup B$, we concatenate all level-linked trees in C in the order of the trees' values. We define the number of *interleaves* between A and B to be $|C|$, the number of trees to be merged.

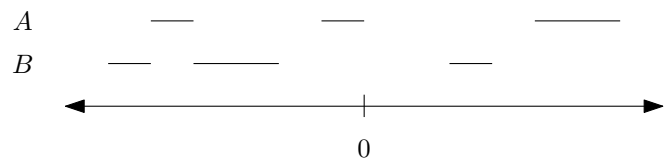

Figure 3-2: An example of a real line representation of two sets A and B. Each line segment denotes an interval of values in the respective set of A or B. No two line segments cover the same interval on the real line. In this example, there are 6 interleaves.

To find the trees in C, we split off trees from the B-trees of A and B while maintaining fingers p_A and p_B; let v_A and v_B refer to the values that the pointers point to (See Figure 3-3 for the pseudocode). When splitting trees off of the original trees A and B, we do not rebalance A or B for each split, as that would be unnecessary and expensive. Instead, we simply use the fingers to search through the tree, cutting off pieces as we go. Initially, we let the fingers point to the minimum values $A.min$ and $B.min$ in the respective trees. While both p_A and p_B still point to nodes within A and B, split off a tree from A if $v_A < v_B$ and split off a tree from B otherwise. The new tree will contain the values in the interval $[\min(v_A, v_B), \max(v_A, v_B))$. We update the appropriate finger p_A or p_B afterwards to point to the successor of $\max(v_A, v_B)$ in the tree. When either p_A or p_B points to a null pointer, the last tree in C is formed from the rest of whichever tree still has a non-null finger pointer.

To perform a split off a tree, we find the last value in the new tree to be formed by searching for the appropriate value v_B or v_A (whichever is larger) in A or B. The search takes $O(1 + \log d)$ time [13, page 229] where d is the distance between v_A and v_B as measured by the number of items contained in the interval. Carving out the tree from one of the original trees A or B takes the same amount of time $O(1 + \log d)$, the height of the resulting tree. Rebalancing this new tree to fix any node splittings along its *spines*, or the paths from the root to the first and last leaves in the tree, takes $O(1 + \log d)$ as well. Since the original trees are never rebalanced, the total cost of the splits is $O(\sum_i (1 + \log d_i))$ where d_i is the size of the ith tree that is split from A or B. This bound is equivalent to the sum of the heights of the trees in C.

```
MakeC(A, B)
 1  i ← 1
 2  p_A ← A.min
 3  p_B ← B.min
 4  while A and B are nonempty
 5        do if v_A < v_B
 6              then C_i ← tree split off from A from p_A to Predecessor(A, v_B)
 7                   p_A ← Successor(A, v_B)
 8              else C_i ← tree split off from B from p_B to Predecessor(B, v_A)
 9                   p_B ← Successor(B, v_a)
10           i ← i + 1
11  if A is nonempty
12     then C_i ← A
13     else C_i ← B
```

Figure 3-3: Pseudocode for generating the level-linked trees C_i to be merged for the union $A \cup B$.

To merge the trees in C into one final tree, we can simply start with the first tree and iteratively concatenate the rest of the trees one at a time. A single concatenate operation takes $O(1 + \log n_2 - \log n_1)$ time [1, page 154] where n_2 is the size of the larger tree and n_1 is the size of the smaller tree. This can be done by attaching the smaller tree at the correct height in the larger tree and simply rebalancing any nodes on the path from that height $O(1 + \log n_1)$ to the root of the large tree, which is at height $O(1 + \log n_2)$. A naïve analysis would show that this algorithm takes longer than $O(\sum_i (1 + \log d_i))$. For example, when merging n singleton elements, this would cost $O(\sum_{i=1}^{n}(1 + \log i - 0)) = O(n \log n)$.

To see why merging trees of C into one final tree takes only $O(\sum_i (1 + \log d_i))$ time, we can use a simple amortization. At any time during the merge, let Φ denote the total number of full nodes (with the maximum possible number of children) among nodes of the spines of the current set of trees. Initially, Φ is at most the total number of nodes on the spines of the trees in C, which is twice the height of each tree in C, and hence $O(\sum_i (1 + \log d_i))$. A concatenate operation attaches a tree as a child of a node on the spine of another tree, increasing that node's degree. If that node was not previously full, the concatenate operation finishes and Φ may increase by 1. Otherwise, the node becomes overfull and the concatenate operation splits the node, increasing the parent's degree but reducing the original node's degree into two split nodes that are less than full. Then the process repeats, say for k steps.

Now Φ increases by 1 only at the kth step of the concatenate operation, where a node might become full, and decreases by 1 at every other step where a previously full node becomes overfull and gets split. Thus the $\Theta(k)$ cost of the concatenation can be charged to the $k-2$ decrease in Φ, resulting in an amortized $O(1)$ cost per concatenation, plus the additive cost of $O(\sum_i(1+\log d_i))$ from the initial value of Φ. Thus the total cost of creating the trees in C and merging them back together runs in $O(\sum_i(1+\log d_i))$ time.

Demaine et al. [7] also prove an average-case lower bound for calculating the union of sorted sets. They achieve their result by examining the minimum amount of information needed to prove that a given set is the correct answer. For the union of two sets, the computation is bounded below by the space required to encode the sizes of all the *gaps* in the sets (except for the largest gap for each set), or essentially the sizes of the trees in C, and the space for encoding the sets involved. More formally, the lower bound on the cost is

$$\Omega\left(\log n + \sum_{i=1}^{|C|}(1+\log|C_i|) - \max_{C_i\in A}(1+\log|C_i|) - \max_{C_i\in B}(1+\log|C_i|)\right)$$

Aside from the first term, the rest of the formula is exactly the sum of the heights of the trees in C minus the largest of the trees split off from A and the largest of the trees split off from B. This lower bound matches the upper bound up to constant factors, so we have a tight bound on the algorithm's performance.

3.2 A Lower Bound

We now prove a lower bound on the amortized performance of our algorithm for solving the General Union-Split-Find problem.

Theorem 2. *The level-linked tree solution to General Union-Split-Find requires* $\Omega(\log^2 n)$ *amortized time per operation in the worst case.*

Proof. We prove this lower bound by construction. We first create n elements and form them into $\sqrt{n}$ sets of $\sqrt{n}$ consecutive elements each. Let this configuration of sets and elements be called the *initial configuration*. Constructing the initial configuration for the first time takes $\Theta(n)$ operations for a total cost of $\Theta(n)$. Let these sets be designated by $s_1, s_2, \ldots s_{\sqrt{n}}$. Repeating the following sequence of operations arbitrarily many times will then yield the lower bound:

1. Merge the sets together into a single set using an expensive sequence of $\sqrt{n}-1$ union operations.

2. Split the set of all elements to form the initial configuration using $\sqrt{n}-1$ operations.

To perform the expensive sequence of union operations, we use $\log\sqrt{n} = \Theta(\log n)$ rounds of merging. In the ith round, we merge together all sets that have subscripts that are equal mod $\sqrt{n}/2^i$. See Figures 3-4 and 3-5 for an example. The ith round involves merges that have a cost of $\Theta(\frac{\sqrt{n}}{2^i}(2^i-1)\log\sqrt{n}) = \Theta(\sqrt{n}\log n)$. Summing over all rounds gives us $\Theta(\sqrt{n}\log^2 n)$. The split operations that recreate the initial configuration take another $\sqrt{n}-1$ operations, so the total number of operations is $\Theta(\sqrt{n})$. This yields an amortized lower bound of $\Omega(\log^2 n)$ per operation. We can repeat this merge and split sequence any number of times, so there is no way to charge the work to other operations.

□

3.3 Attempted Upper Bounds

Unlike the other supported operations, the union operation is much more complicated to analyze because it can sometimes be very expensive (as much as $\Theta(n)$) while other times it is merely polylogarithmic. Intuitively, it seems that there ought to be an $o(n)$ amortized upper bound by charging some of the work to the other operations. The worst case $\Theta(n)$ time union operation that merges two sets that have $\Theta(n)$ interleaves results in one sorted set of elements. To recreate the configuration immediately before the expensive union operation requires $\Omega(n)$ operations to separate each element and union the elements such that there are $\Omega(n)$ interleaves.

One is thus tempted to create a potential function based on counting interleaves, set sizes, numbers of sets, or some combination of those values thereof. Unfortunately, all attempts at creating some natural potential function based on these values have failed thus far. For example, if we attempt to count interleaves, we could count for each set the number of contiguous intervals of values with respect to all other sets. This fails for this example in Figure 3-6. The reason this potential function does not work is that it does not take into account any pairwise information. Unfortunately, adding up all pairwise interleaves also fails (see Figure 3-7).

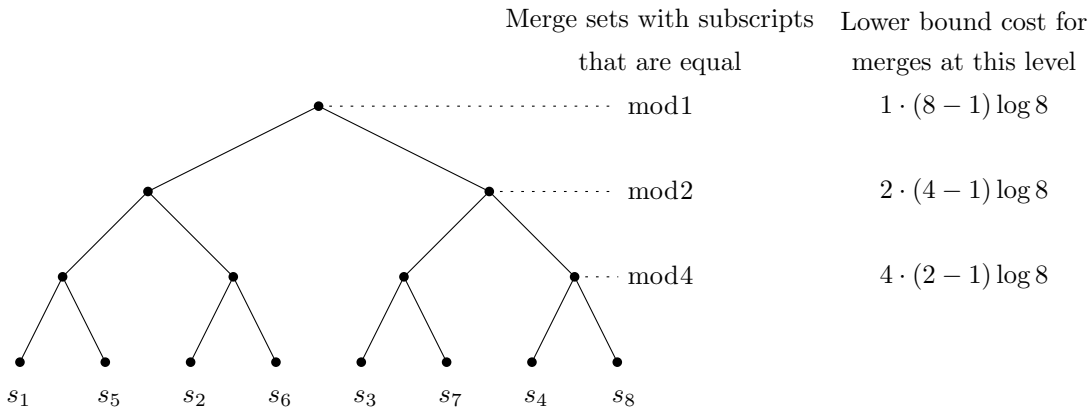

Figure 3-4: An example where $\sqrt{n} = 8$, and we have 8 equally sized sets of size 8 each. The binary tree represents the union operations used to merge them back into one set. Each level of nonzero height in the tree has a merge cost of at least $4 \log 8 = \Omega(\sqrt{n} \log n)$, and there are $\log 8 = \Theta(\log n)$ of these levels.

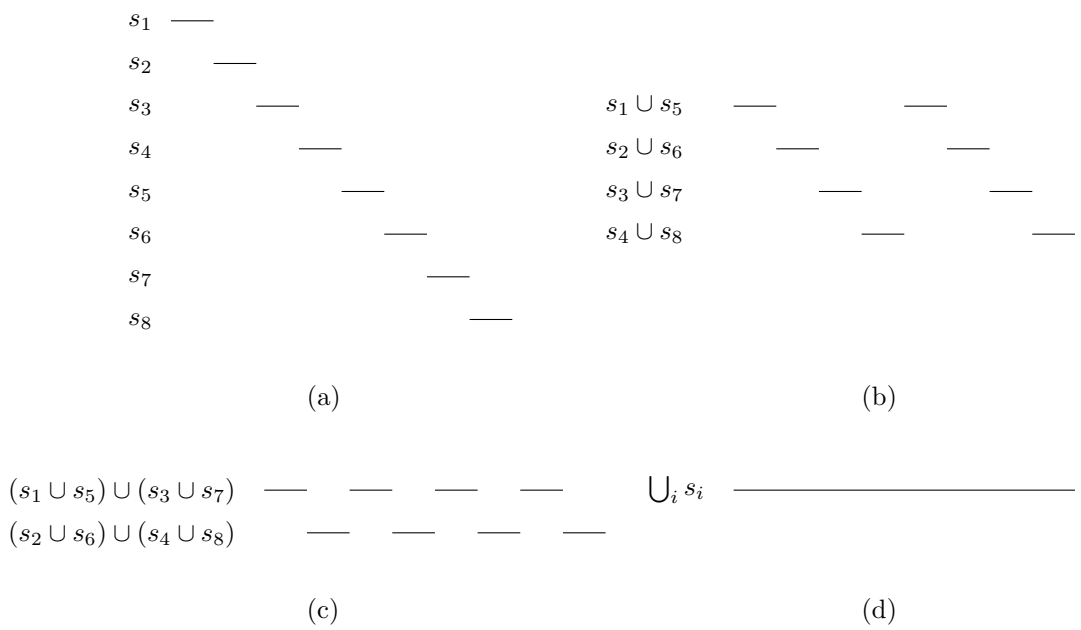

Figure 3-5: The same example with $\sqrt{n} = 8$. (a-d) represent the state of the sets before and after each of the rounds of union operations. Each line segment represents a contiguous part of a set with values that do not interleave with values in other sets. Sets are composed of collinear line segments. Thus in (a), we have the initial state where there are 8 consecutive sets with no interleaving values. After the first round, we get (b) where there are 4 distinct sets. The key to why this merge sequence is expensive is that no contiguous line segment is ever joined with another until the final step, maximizing the number of interleaves at all times and increasing the cost of the union operations.

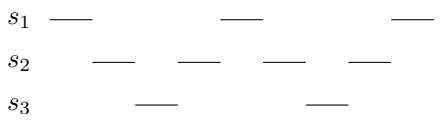

Figure 3-6: Attempting to use a potential function of the sum of the interleaves does not work because unioning s_1 and s_3 together does not decrease the total number of interleaves while it costs potentially $O(n)$.

Figure 3-7: Attempting to use a potential function of the sum of all pairwise interleaves with all sets doesn't work because inserted sets in the middle of the range (where the dashed line is) for this example can increase the potential function by $\Theta(n)$, which is far too expensive.

The ideal potential function is tracks the cost of the most expensive merge sequence possible given the current collection of sets. If, as in the example used to prove the stronger lower bound from Figure 3-4, we assume we have k equally sized non-overlapping sets that are merged, the maximum merge cost is the cost of that expensive merge sequence, $O(k \log k \log(n/k + 1))$. This is encouraging because, for the case of $k = n$, we get a total cost of $O(n \log n)$ for an amortized cost of $O(\log n)$, and the worst that happens is when $k = \sqrt{n}$ as shown before. Unfortunately, using this formula as the potential function fails to accurately represent the max cost if we choose to completely merge one side of the merge tree before performing any of the union operations for the other side. The difficulty in setting one potential function that completely captures the maximum cost comes from the fact that any formula will lose any information about the structure of the sets in the current state. Moreover, this does not address the case where we have a different initial state nor what happens when new sets are constructed.

3.4 Open Problems and Future Directions

The problem of finding an accurate potential function to analyze the performance of level-linked trees thus leads to several natural open questions.

Open Problem 2. *Given a collection of disjoint sets, what is the complexity of finding the*

maximum merge sequence?

For this problem, some intuitive greedy choices do not work. Notably, there may not exist a maximum merge sequence for which the first merge is the most costly (Figure 3-8), nor for which the first merge involves the set with the fewest number of interleaves (Figure 3-9).

Figure 3-8: Here, the costliest merge is that of either s_1 and s_3 or s_2 and s_3. This would result in a total cost of 6, assuming unit gap cost. Yet the most expensive merge sequence involves merging s_1 and s_2 first, then merging with s_3 for a total cost of 7.

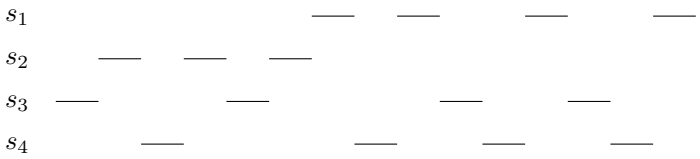

Figure 3-9: Here, s_2 has the fewest interleaves with any other set, but the first merge in the maximum merge sequence does not involve it. The intuition for this heuristic is that expensive merges come from merging sets with many interleaves, which can be formed by merging together sets with fewer interleaves. This heuristic certainly works for the expensive merge sequence used to prove the lower bound, but the maximum merge sequence here involves first merging either the pair s_3 and s_4 (followed by merging in the other sets one by one in any order) or the pair s_1 and s_4 (followed by merging it with s_3 and then s_2) for a total cost of 23, assuming unit gap cost.

The cost of the minimum merge sequence serves as an adequate potential function if only splits are performed. If we could maintain both the costs of the minimum and the maximum merge sequences, perhaps we can charge the appropriate operations to the appropriate functions. This line of thought yields the following question.

Open Problem 3. *Given a collection of disjoint sets, what is the gap in cost between those of the minimum merge sequence and the maximum merge sequence?*

Finally, perhaps analyzing the average case will yield a useful potential function. The difficulty in calculating the expectation, however, is that both the order in which sets are merged must be chosen as well as the structure of the corresponding merge tree.

Open Problem 4. *Given a collection of disjoint sets, does a random merge sequence perform badly in expectation? In particular, is it asymptotically as expensive as the maximum merge sequence?*

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe. Dynamic nested brackets. *Information and Computation*, 193(2):75–83, September 2004.

[3] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pages 560–569, October 1996.

[4] M. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, November 2000.

[5] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[7] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proc. 11th Annual ACM Symposium on Discrete Algorithms*, pages 743–752, January 2000.

[8] E. D. Demaine, A. Lubiw, and J. I. Munro. Personal communication.

[9] G. S. Frandsen, T. Husfeldt, P. B. Miltersen, T. Rauhe, and S. Skyum. Dynamic algorithms for the dyck languages. In *Proc. 4th Workshop on Algorithms and Data Structures*, pages 98–108, 1995.

[10] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Symposium on Theory of Computing*, pages 345–354, May 1989.

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

[12] T. Husfeldt, T. Rauhe, and S. Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. *Nordic J. of Computing*, 3(4):323–336, December 1996.

[13] K. Mehlhorn. *Data Structures and Algorithms*, volume 1, chapter 5. Springer-Verlag, 1984.

[14] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(6):1093–1102, December 1988.

[15] P. Miltersen. Cell probe complexity - a survey. In *Pre-Conference Workshop on Advances in Data Structures at the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1999.

[16] J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[17] M. Pătraşcu and E. D. Demaine. Lower bounds for dynamic connectivity. In *Proc. 36th Symposium on Theory of Computing*, pages 546–553, June 2004.

[18] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Symposium on Theory of Computing*, pages 232–240, May 2006.

[19] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[20] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, April 1979.

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

[22] A. C.-C. Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.