5

Topics in Data Structures

Giuseppe F. Italiano
University Venezia, "Ca' Foscari" di Venezia

Rajeev Raman
King's College, London

5.1 Introduction

In this chapter, we describe advanced data structures and algorithmic techniques, mostly focusing our attention to two important problems: set union and persistence. We first describe set union data structures. Their discovery required a new set of techniques and tools that have proved useful in other areas as well. We survey algorithms and data structures for set union problems, and attempt to provide a unifying theoretical framework for this growing body of algorithmic tools. **Persistent data structures** maintain information about their past states and find uses in a diverse spectrum of applications. The body of work relating to persistent data structures brings together quite a surprising cocktail of techniques, from real-time computation to techniques from functional programming.

0-8493-2649-4/99/$0.00+$.50

Set Union Data Structures

The *set union problem* consists of maintaining a collection of disjoint sets under an intermixed sequence of the following two kinds of operations:

union(A, B)*:* Combine the two sets A and B into a new set named A.
find(x)*:* Return the name of the set containing element x.

The operations are presented on line, namely each operation must be processed before the next one is known. Initially, the collection consists of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$, and the name of set $\{i\}$ is i, $1 \leq i \leq n$. Figure 5.1 illustrates an example of set union operations.

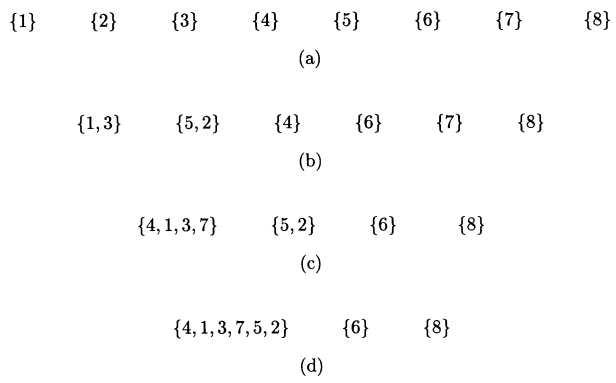

FIGURE 5.1 Examples of set union operations. (a) The initial collection of disjoint sets. (b) The disjoint sets of (a) after performing union(1, 3) and union(5, 2). (c) The disjoint sets of (b) after performing union(1, 7) followed by union(4, 1). (d) The disjoint sets of (c) after performing union(4, 5).

The set union problem has been widely studied, and finds application in a wide range of areas, including Fortran compilers [10, 38], property grammars [78, 79], computational geometry [49, 67, 68], finite state machines [4, 44], string algorithms [5, 48], logic programming and theorem proving [7, 8, 47, 95], and several combinatorial problems such as finding minimum spanning trees [4, 53], solving dynamic edge- and vertex-connectivity problems [98], computing least common ancestors in trees [3], solving off–line minimum problems [34, 45], finding dominators in graphs [83], and checking flow graph reducibility [82].

Several variants of set union have been introduced, in which the possibility of backtracking over the sequences of unions was taken into account [9, 39, 59, 63, 97]. This was motivated by problems arising in logic programming interpreter memory management [40, 60, 61, 96].

Persistent Data Structures

Data structures that one encounters in traditional algorithmic settings are *ephemeral*, i.e., if the data structure is updated then the previous state of the data structure is lost. A *persistent* data structure, on the other hand, preserves old versions of the data structure. Several kinds of persistence can be distinguished based upon what kind of access is allowed to old versions of the data structure. Accesses to a data structure can be of two kinds: *updates*, which change the information content of the data structure, and *queries*, which do not. For the sake of ease of presentation, we will assume that queries do not even change the internal representation of the data, i.e., read-only access to a data structure suffices to answer a query.

In the persistent setting we would like to maintain multiple *versions* of data structures. In addition to the arguments taken by its ephemeral counterparts, a persistent query or update operation takes as

an argument the version of the data structure to which the query or update refers. A persistent update also returns a *handle* to the new version created by the update. We distinguish between three kinds of persistence:

- A *partially* persistent data structure allows updates only to the latest version of the data structure. All versions of the data structure may be queried, however. Clearly, the versions of a partially persistent data structure exhibit a linear ordering as shown in Fig. 5.2(a).
- A *fully* persistent data structure allows all existing versions of the data structure to be queried or updated. However, an update may operate only on a single version at a time—for instance combining two or more old versions of the data structure to form a new one is not allowed. The versions of a fully persistent data structure form a tree, as shown in Fig. 5.2(b).
- A **purely functional language** is one that does not allow any *destructive* operation—one that overwrites data—such as the assignment operation. Purely functional languages are side-effect-free, i.e., invoking a function has no effect other than computing the value returned by the function. In particular, an update operation to a data structure implemented in a purely functional language returns a new data structure containing the updated values, while leaving the original data structure unchanged. Data structures implemented in purely functional languages are therefore persistent in the strongest possible sense, as they allow unrestricted access for both reading and updating all versions of the data structure.
  An example of a purely functional language is pure LISP [64]. Side-effect-free code can also be written in *functional* languages such as ML [70], most existing variants of LISP (e.g., Common LISP [80]) or Haskell [46], by eschewing the destructive operations supported by these languages.

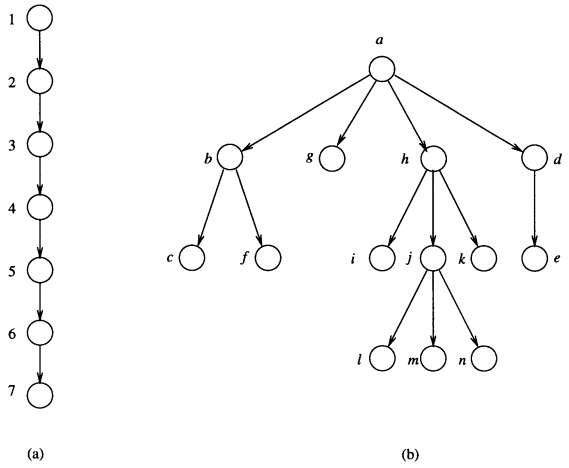

FIGURE 5.2 Structure of versions for (a) partial and (b) full persistence.

This section aims to cover a selection of the major results relating to the above forms of persistence. The body of work contains both *ad hoc* techniques for creating persistent data structures for particular problems, as well as general techniques to make ephemeral data structures persistent. Indeed, early work on persistence [17, 20, 30] focused almost exclusively on the former. Sarnak [75] and Driscoll et al. [28] were the first to offer very efficient general techniques for partial and full persistence. These and related results will form the bulk of the material in this chapter dealing with partial and full persistence. However, the prospect of obtaining still greater efficiency led to the further development of some *ad hoc* persistent

data structures [25, 26, 41]. The results on functional data structures will largely focus on implementations of individual data structures.

There has also been some research into data structures that support *backtrack* or *rollback* operations, whereby the data structure can be reset to some previous state. We do not cover these operations in this section, but we note that fully persistent data structures support backtracking (although sometimes not as efficiently as data structures designed especially for backtracking). Data structures with backtracking for the union–find problem are covered in Section 5.4.

Persistent data structures have numerous applications, including text, program and file editing and maintenance, computational geometry, tree pattern matching and inheritance in object-oriented programming languages. One elegant application of partially persistent search trees to the classical geometric problem of *planar point location* was given by Sarnak and Tarjan [76]. Suppose the Euclidean plane is divided into polygons by a collection of n line segments that intersect only at their endpoints (see Fig. 5.3), and we want to preprocess the collection of line segments so that, given a query point p, we can efficiently determine the polygon to which p belongs. Sarnak and Tarjan achieve this by combining the well-known *plane sweep* technique with a persistent data structure.

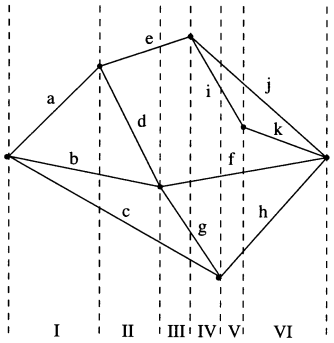

FIGURE 5.3 A planar subdivision.

Imagine moving an infinite vertical line (called the *sweep line*) from left to right across the plane, beginning at the leftmost endpoint of any line segment. As the sweep line moves, we maintain the line segments currently intersecting the sweep line in a balanced binary search tree, in order of their point of intersection with the sweep line (i.e., of two line segments, the one that intersects the sweep line at a higher location is considered smaller). Figure 5.4 shows the evolution of the search tree as the sweep line continues its progress from left to right. Note that the plane is divided into vertical *slabs*, within which the search tree does not change.

Given a query point p, we first locate the slab in which the x-coordinate of p lies. If we could remember what our search tree looked like while the sweep line was in this slab, we could query the search tree using the y-coordinate of p to find the two segments immediately above and below p in this slab; these line segments uniquely determine the polygon in which p lies. However, if we maintained the line segments in a partially persistent search tree as the sweep line moves from left to right, all incarnations of the search tree during this process are available for queries.

Sarnak and Tarjan show that it is possible to perform the preprocessing (which merely consists of building up the persistent tree) in $O(n \log n)$ time. The data structure uses $O(n)$ space and can be queried in $O(\log n)$ time, giving a simple optimal solution to the planar point location problem.

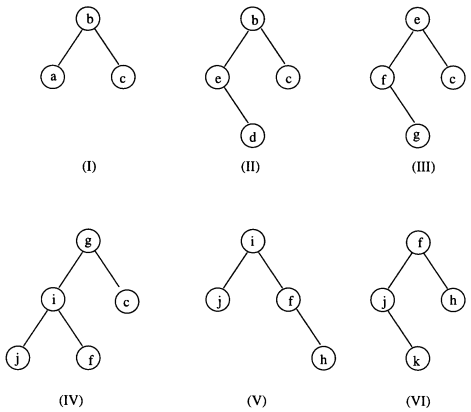

FIGURE 5.4 The evolution of the search tree during the plane sweep.

Models of Computation

Different models of computation have been developed for analyzing data structures. One model of computation is the **random access machine,** whose memory consists of an unbounded sequence of registers, each of which is capable of holding an integer. In this model, arithmetic operations are allowed to compute the address of a memory register. Usually, it is assumed that the size of a register is bounded by $O(\log n)$[1] bits, where n is the input problem size. A more formal definition of random access machines can be found in [4]. Another model of computation, known as the **cell probe model** of computation, was introduced by Yao [99]. In the cell probe, the cost of a computation is measured by the total number of memory accesses to a random access memory with $\lceil \log n \rceil$ bits cell size. All other computations are not accounted for and are considered to be free. Note that the cell probe model is more general than a random access machine, and thus, is more suitable for proving lower bounds. A third model of computation is the **pointer machine** [13, 54, 55, 77, 85]. Its storage consists of an unbounded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information but no arithmetic is allowed to compute the address of a register. The only possibility to access a register is by following pointers. This is the main difference between random access machines and pointer machines. Throughout this chapter, we use the terms *random-access algorithms*, *cell-probe algorithms*, and *pointer-based algorithms* to refer to algorithms respectively for random access machines, the cell probe model, and pointer machines.

Among pointer-based algorithms, two different classes were defined specifically for set union problems: *separable pointer algorithms* [85] and *nonseparable pointer algorithms* [69].

Separable pointer algorithms run on a pointer machine and satisfy the **separability** assumption as defined in [85] (see below). A separable pointer algorithm makes use of a linked data structure, namely a collection of records and pointers that can be thought of as a directed graph: each record is represented by a node and each pointer is represented by an edge in the graph. The algorithm solves the set union problem according to the following rules [14, 85]:

(i) The operations must be performed on line, i.e., each operation must be executed before the next one is known.

(ii) Each element of each set is a node of the data structure. There can be also additional (working) nodes.

[1] Throughout this chapter all logarithms are assumed to be to the base 2, unless explicitly otherwise specified.

(iii) (*Separability*). After each operation, the data structure can be partitioned into disjoint subgraphs such that each subgraph corresponds to exactly one current set. The name of the set occurs in exactly one node in the subgraph. No edge leads from one subgraph to another.

(iv) To perform find(x), the algorithm obtains the node v corresponding to element x and follows paths starting from v until it reaches the node which contains the name of the corresponding set.

(v) During any operation the algorithm may insert or delete any number of edges. The only restriction is that rule (iii) must hold after each operation.

The class of *nonseparable pointer algorithms* [69] does not require the separability assumption. The only requirement is that the number of edges leaving each node must be bounded by some constant $c > 0$. More formally, rule (iii) above is replaced by the following rule, while the other four rules are left unchanged:

(iii) There exists a constant $c > 0$ such that there are at most c edges leaving a node.

As we will see later on, often separable and nonseparable pointer-based algorithms admit quite different upper and lower bounds for the same problems.

5.2 The Set Union Problem

As defined in the previous section, the *set union problem* consists of performing a sequence of union and find operations, starting from a collection of n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. The initial name of set $\{i\}$ is i. As there are at most n items to be united, the number of unions in any sequence of operations is bounded above by $(n - 1)$. There are two invariants which hold at any time for the set union problem: first, the sets are always disjoint and define a partition of $\{1, 2, \ldots, n\}$; second, the name of each set corresponds to one of the items contained in the set itself. Both invariants are trivial consequences of the definition of union and find operations.

A different version of this problem considers the following operation in place of unions:

unite(A,B): Combine the two sets A and B into a new set, whose name is either A or B.

The only difference between union and unite is that unite allows the name of the new set to be arbitrarily chosen (e.g., at run time by the algorithm). This is not a significant restriction in many applications, where one is mostly concerned with testing whether two elements belong to the same set, no matter what the name of the set can be. However, some extensions of the set union problem have quite different time bounds depending on whether unions or unites are considered. In the following, we will deal with unions unless explicitly specified otherwise.

Amortized Time Complexity

In this section we describe algorithms for the set union problem [84, 89] giving the optimal amortized time complexity per operation. We only mention here that the amortized time is the running time per operation averaged over a worst-case sequence of operations, and refer the interested reader to [88] for a more detailed definition of amortized complexity. For the sake of completeness, we first survey some of the basic algorithms that have been proposed in the literature [4, 31, 38]. These are: the *quick-find*, the *weighted quick-find*, the *quick-union*, and the *weighted quick-union* algorithms. The quick-find algorithm performs find operations quickly, while the quick-union algorithm performs union operations quickly. Their weighted counterparts speed these computations up by introducing some weighting rules during union operations.

Most of these algorithms represent sets as rooted trees, following a technique introduced first by Galler and Fischer [38]. There is a tree for each disjoint set, and nodes of a tree correspond to elements of the

corresponding set. The name of the set is stored in the tree root. Each tree node has a pointer to its parent: in the following, we refer to $p(x)$ as the parent of node x.

The quick-find algorithm can be described as follows. Each set is represented by a tree of height 1. Elements of the set are the leaves of the tree. The root of the tree is a special node which contains the name of the set. Initially, singleton set $\{i\}$, $1 \leq i \leq n$, is represented by a tree of height 1 composed of one leaf and one root. To perform a union(A, B), all the leaves of the tree corresponding to B are made children of the root of the tree corresponding to A. The old root of B is deleted. This maintains the invariant that each tree is of height 1 and can be performed in $O(|B|)$ time, where $|B|$ denotes the total number of elements in set B. Since a set can have as many as $O(n)$ elements, this gives an $O(n)$ time complexity in the worst case for each union. To perform a find(x), return the name stored in the parent of x. Since all trees are maintained of height 1, the parent of x is a tree root. Consequently a find requires $O(1)$ time.

A more efficient variant attributed to McIlroy and Morris (see [4]) and known as weighted quick-find uses the freedom implicit in each union operation according to the following weighting rule.

> *Union by size:* Make the children of the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires that the size of each tree is maintained throughout any sequence of operations.

Although this rule does not improve the worst-case time complexity of each operation, it improves to $O(\log n)$ the amortized bound of a union (see, e.g., [4]).

The quick-union algorithm [38] can be described as follows. Again, each set is represented by a tree. However, there are two main differences with the data structure used by the quick-find algorithm. The first is that now the height of a tree can be greater than 1. The second is that each node of each tree corresponds to an element of a set and therefore there is no need for special nodes. Once again, the root of each tree contains the name of the corresponding set. A union(A, B) is performed by making the tree root of set B child of the tree root of set A. A find(x) is performed by starting from the node x and by following the pointer to the parent until the tree root is reached. The name of the set stored in the tree root is then returned. As a result, the quick-union algorithm is able to support each union in $O(1)$ time and each find in $O(n)$ time.

This time bound can be improved by using the freedom implicit in each union operation, according to one of the following two union rules. This gives rise to two weighted quick-union algorithms.

> *Union by size:* Make the root of the smaller tree point to the root of the larger, arbitrarily breaking a tie. This requires maintaining the number of descendants for each node, in the following referred to as the *size* of a node, throughout all the sequence of operations.
>
> *Union by rank:* [89] Make the root of the shallower tree point to the root of the other, arbitrarily breaking a tie. This requires maintaining the height of the subtree rooted at each node, in the following referred to as the *rank* of a node, throughout all the sequences of operations.

After a union(A, B), the name of the new tree root is set to A. It can be easily proved (see, e.g., [89]) that the height of the trees achieved with either the "union by size" or the "union by rank" rule is never more than $\log n$. Thus, with either rule each union can be performed in $O(1)$ time and each find in $O(\log n)$ time.

A better amortized bound can be obtained if one of the following *compaction rules* is applied to the path examined during a find operation (see Fig. 5.5).

> *Path compression* [45]: Make every encountered node point to the tree root.
>
> *Path splitting* [93, 94]: Make every encountered node (except the last and the next to last) point to its grandparent.
>
> *Path halving* [93, 94]: Make every other encountered node (except the last and the next to last) point to its grandparent.

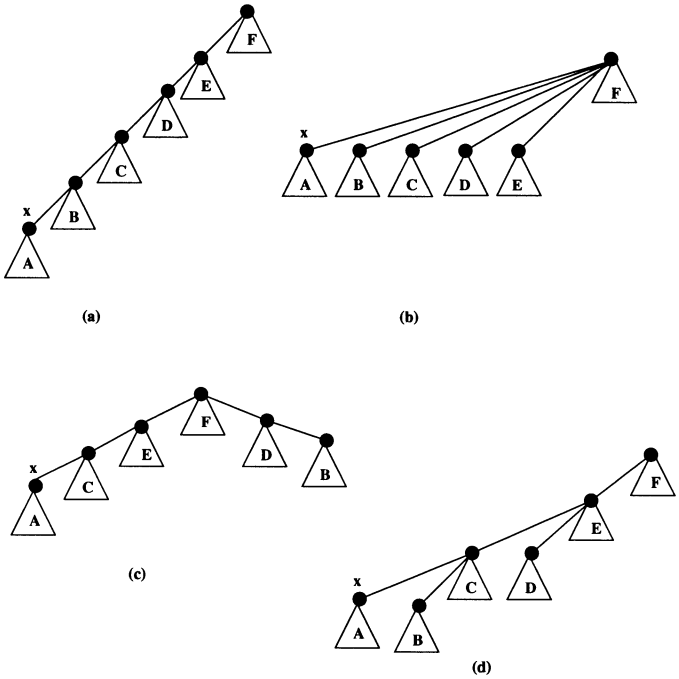

FIGURE 5.5 Illustrating path compaction techniques: (a) the tree before performing a find(x) operation; (b) path compression; (c) path splitting; (d) path halving.

Combining the two choices of a union rule and the three choices of a compaction rule, six possible algorithms are obtained. As shown in [89] they all have an $O(\alpha(m+n,n))$ amortized time complexity, where α is a very slowly growing function, a functional inverse of Ackermann's function [1].

THEOREM 5.1 **[89]** *The algorithms with either linking by size or linking by rank and either compression, splitting or halving run in* $O(n+m\alpha(m+n,n))$ *time on a sequence of at most* $(n-1)$ *unions and m finds.*

No better amortized bound is possible for separable and nonseparable pointer algorithms or in the cell probe model of computation [32, 56, 89].

THEOREM 5.2 **[32, 56, 89]** *Any pointer-based or cell-probe algorithm requires* $\Omega(n+m\alpha(m+n,n))$ *worst-case time for processing a sequence of* $(n-1)$ *unions and m finds.*

Single-Operation Worst-Case Time Complexity

The algorithms that use any union and any compaction rule have still single-operation worst-case time complexity $O(\log n)$ [89], since the trees created by any of the union rules can have height as large as $O(\log n)$. Blum [14] proposed a data structure for the set union problem that supports each union and find in $O(\log n/\log\log n)$ time in the worst case, and showed that this is the actual lower bound for separable pointer-based algorithms.

The data structure used to establish the upper bound is called *k–UF tree*. For any $k \geq 2$, a k–UF tree is a rooted tree such that: (i) the root has at least two children; (ii) each internal node has at least k children; and (iii) all the leaves are at the same level. As a consequence of this definition, the height of a k–UF tree

with n leaves is at most $\lceil \log_k n \rceil$. We refer to the root of a k–UF tree as *fat* if it has more than k children, and as *slim* otherwise. A k–UF tree is said to be *fat* if its root is fat, otherwise it is referred to as *slim*.

Disjoint sets can be represented by k–UF trees as follows. The elements of the set are stored in the leaves and the name of the set is stored in the root. Furthermore, the root also contains the height of the tree and a bit specifying whether it is fat or slim. A find(x) is performed as described in the previous section by starting from the leaf containing x and returning the name stored in the root. This can be accomplished in $O(\log_k n)$ worst-case time. A union(A, B) is performed by first accessing the roots r_A and r_B of the corresponding k–UF trees T_A and T_B. Blum assumed that his algorithm obtained in constant time r_A and r_B before performing a union(A, B). If this is not the case, r_A and r_B can be obtained by means of two finds (i.e., find(A) and find(B)), due to the property that the name of each set corresponds to one of the items contained in the set itself. We now show how to unite the two k–UF *trees* T_A and T_B. Assume without loss of generality that $height(T_B) \leq height(T_A)$. Let v be the node on the path from the leftmost leaf of T_A to r_A with the same height as T_B. Clearly, v can be located by following the leftmost path starting from the root r_A for exactly $height(T_A) - height(T_B)$ steps. When merging T_A and T_B, only three cases are possible, which give rise to three different types of unions.

Type 1: Root r_B is fat (i.e., has more than k children) and v is not the root of T_A. Then r_B is made a sibling of v.

Type 2: Root r_B is fat and v is fat and equal to r_A (the root of T_A). A new (slim) root r is created and both r_A and r_B are made children of r.

Type 3: This deals with the remaining cases, i.e., either root r_B is slim or $v = r_A$ is slim. If root r_B is slim, then all the children of r_B are made the rightmost children of v, and r_B is deleted. Otherwise, all the children of the slim node $v = r_A$ are made the rightmost children of r_B, and r_A is deleted.

THEOREM 5.3 **[14]** *k–UF trees can support each union and find in $O(\log n / \log\log n)$ time in the worst case. Their space complexity is $O(n)$.*

PROOF Each find can be performed in $O(\log_k n)$ time. Each union(A, B) can require at most $O(\log_k n)$ time to locate the nodes r_A, r_B and v as defined above. Both type 1 and type 2 unions can be performed in constant time, while type 3 unions require at most $O(k)$ time, due to the definition of a slim root. Choosing $k = \lceil \log n / \log\log n \rceil$ yields the claimed time bound. The space complexity derives from the fact that a k–UF tree with ℓ leaves has at most $(2\ell - 1)$ nodes. Thus, the forest of k–UF trees requires at most a total of $O(n)$ space to store all the disjoint sets.

Blum showed also that this bound is tight for the class of separable pointer algorithms, while Fredman and Saks [32] showed that the same lower bound holds in the cell probe model of computation.

THEOREM 5.4 **[14, 32]** *Every separable pointer or cell-probe algorithm for the disjoint set union problem has single-operation worst-case time complexity at least $\Omega(\log n / \log\log n)$.*

Special Linear Cases

The six algorithms using either union rule and either compaction rule as described in "Amortized Time Complexity" run in $O(n + m\alpha(m, n))$ time on a sequence of at most $(n - 1)$ union and m find operations. As stated in Theorem 5.2, no better amortized bound is possible for either pointer-based algorithms or in the cell probe model of computation. This does not exclude, however, that a better bound is possible for a special case of set union. Gabow and Tarjan [34] indeed proposed a random-access algorithm that

runs in linear time in the special case where the structure of the union operations is known in advance. Interestingly, Tarjan's lower bound for separable pointer algorithms applies also to this special case, and thus, the power of a random access machine seems necessary to achieve a linear-time algorithm. This result is of theoretical interest as well as being significant in many applications, such as scheduling problems, the off-line minimum problem, finding maximum matching on graphs, VLSI channel routing, finding nearest common ancestors in trees, and flow graph reducibility [34].

The problem can be formalized as follows. We are given a tree T containing n nodes which correspond to the initial n singleton sets. Denoting by $p(v)$ the parent of the node v in T, we have to perform a sequence of union and find operations such that each union can be only of the form union$(p(v), v)$. For such a reason, T is called the *static union tree* and the problem will be referred to as the *static tree set union*. Also the case in which the union tree can dynamically grow by means of new node insertions (referred to as *incremental tree set union*) can be solved in linear time.

THEOREM 5.5 **[34]** *If the knowledge about the union tree is available in advance, each union and find operation can be supported in $O(1)$ amortized time. The total space required is $O(n)$.*

The same algorithm given for the static tree set union can be extended to the incremental tree set union problem. For this problem, the union tree is not known in advance but is allowed to grow only one node at the time during the sequence of union and find operations. This has application in several algorithms for finding maximum matching in general graphs.

THEOREM 5.6 **[34]** *The algorithm for incremental tree set union runs in a total of $O(m + n)$ time and requires $O(n)$ preprocessing time and space.*

Loebl and Nešetřil [58] presented a linear-time algorithm for another special case of the set union problem. They considered sequences of unions and finds with a constraint on the subsequence of finds. Namely, the finds are listed in a *postorder* fashion, where a postorder is a linear ordering of the leaves induced by a drawing of the tree in the plane. In this framework, they proved that such sequences of union and find operations can be performed in linear time, thus, getting $O(1)$ amortized time per operation. A preliminary version of these results was reported in [58].

5.3 The Set Union Problem on Intervals

In this section, we describe efficient solutions to the *set union problem on intervals*, which can be defined as follows. Informally, we would like to maintain a partition of a list $\{1, 2, \ldots, n\}$ in adjacent intervals. A union operation joins two adjacent intervals, a find returns the name of the interval containing x and a split divides the interval containing x (at x itself). More formally, at any time we maintain a collection of disjoint sets A_i with the following properties. The A_i's, $1 \leq i \leq k$, are disjoint sets whose members are ordered by the relation $\leq$, and such that $\cup_{i=1}^{k} A_i = \{1, 2, \ldots, n\}$. Furthermore, every item in A_i is less than or equal to all the items in A_{i+1}, for $i = 1, 2, \ldots, n-1$. In other words, the intervals A_i partition the interval $[1, n]$. Set A_i is said to be adjacent to sets A_{i-1} and A_{i+1}. The set union problem on intervals consists of performing a sequence of the following three operations:

union(S_1, S_2, S)*:* Given the adjacent sets S_1 and S_2, combine them into a new set $S = S_1 \cup S_2$;

find(x)*:* Given the item x, return the name of the set containing x;

split(S, S_1, S_2, x)*:* Partition S into two sets $S_1 = \{a \in S | a < x\}$ and $S_2 = \{a \in S | a \geq x\}$.

Adopting the same terminology used in [69], we will refer to the set union problem on intervals as the *interval union-split-find problem*. After discussing this problem, we consider two special cases:

the *interval union-find problem* and the *interval split-find problem*, where only union-find and split-find operations are allowed, respectively. The interval union-split-find problem and its subproblems have applications in a wide range of areas, including problems in computational geometry such as dynamic segment intersection [49, 67, 68], shortest paths problems [6, 66], and the longest common subsequence problem [5, 48].

Interval Union-Split-Find

In this section we will describe optimal separable and nonseparable pointer algorithms for the interval union-split-find problem. The best separable algorithm for this problem runs in $O(\log n)$ worst-case time for each operation, while nonseparable pointer algorithms require only $O(\log \log n)$ worst-case time for each operation. In both cases, no better bound is possible.

The upper bound for separable pointer algorithms can be easily obtained by means of balanced trees [4, 21], while the lower bound was proved by Mehlhorn et al. [69].

THEOREM 5.7 **[69]** *For any separable pointer algorithm, both the worst-case per operation time complexity of the interval split-find problem and the amortized time complexity of the interval union-split-find problem are* $\Omega(\log n)$.

Turning to nonseparable pointer algorithms, the upper bound can be found in [52, 68, 91, 92]. In particular, van Emde Boas et al. [92] introduced a priority queue which supports among other operations *insert*, *delete*, and *successor* on a set with elements belonging to a fixed universe $S = \{1, 2, \ldots, n\}$. The time required by each of those operation is $O(\log \log n)$. Originally, the space was $O(n \log \log n)$ but later it was improved to $O(n)$. It is easy to show (see also [69]) that the above operations correspond respectively to union, split, and find, and therefore the following theorem holds.

THEOREM 5.8 **[91]** *Each union, find and split can be supported in* $O(\log \log n)$ *worst-case time. The space required is* $O(n)$.

We observe that the algorithm based on van Emde Boas' priority queue is inherently nonseparable. Mehlhorn et al. [69] proved that this is indeed the best possible bound that can be achieved by a nonseparable pointer algorithm:

THEOREM 5.9 **[69]** *For any nonseparable pointer algorithm, both the worst-case per operation time complexity of the interval split-find problem and the amortized time complexity of the interval union-split-find problem are* $\Omega(\log \log n)$.

Notice that Theorems 5.7 and 5.8 imply that for the interval union-split-find problem the separability assumption causes an exponential loss of efficiency.

Interval Union-Find

The interval union-find problem can be seen from two different perspectives: indeed it is a special case of the union-split-find problem, when no split operations are performed, and it is a restriction of the set union problem described in Section 5.2, where only adjacent intervals are allowed to be joined. Consequently, the $O(\alpha(m+n, n))$ amortized bound given in Theorem 5.1 and the $O(\log n / \log \log n)$ single-operation worst-case bound given in Theorem 5.3 trivially extend to interval union–find. Tarjan's proof of the $\Omega(\alpha(m+n, n))$ amortized lower bound for separable pointer algorithms also holds for the interval union–

find problem, while Blum and Rochow [15] have adapted Blum's original lower bound proof for separable pointer algorithms to interval union–find. Thus, the best bounds for separable pointer algorithms are achieved by employing the more general set union algorithms. On the other side, the interval union–find problem can be solved in $O(\log\log n)$ time per operation with the nonseparable algorithm of van Emde Boas [91], while Gabow and Tarjan used the data structure described in "Special Linear Cases" to obtain an $O(1)$ amortized time for interval union–find on a random access machine.

Interval Split–Find

According to Theorems 5.7, 5.8, and 5.9, the two algorithms given for the more general interval union-split-find problem, are still optimal for the single-operation worst-case time complexity of the interval split–find problem. As a result, each split and find operation can be supported in $\Theta(\log n)$ and in $\Theta(\log\log n)$ time, respectively, in the separable and nonseparable pointer machine model.

As shown by Hopcroft and Ullman [45], the amortized complexity of this problem can be reduced to $O(\log^* n)$, where $\log^* n$ is the iterated logarithm function.[2] Their algorithm works as follows. The basic data structure is a tree, for which each node at level i, $i \geq 1$, has at most $2^{f(i-1)}$ children, where $f(i) = f(i-1)2^{f(i-1)}$, for $i \geq 1$, and $f(0) = 1$. A node is said to be complete either if it is at level 0 or if it is at level $i \geq 1$ and has $2^{f(i-1)}$ children, all of which are complete. A node that is not complete is called incomplete. The invariant maintained for the data structure is that no node has more than two incomplete children. Moreover, the incomplete children (if any) will be leftmost and rightmost. As in the usual tree data structures for set union, the name of a set is stored in the tree root.

Initially, such a tree with n leaves is created. Its height is $O(\log^* n)$ and therefore a find(x) will require $O(\log^* n)$ time to return the name of the set. To perform a split(x), we start at the leaf corresponding to x and traverse the path to the root to partition the tree into two trees. It is possible to show that using this data structure, the amortized cost of a split is $O(\log^* n)$ [45]. This bound can be further improved to $O(\alpha(m,n))$ as shown by Gabow [33]. The algorithm used to establish this upper bound relies on a sophisticated partition of the items contained in each set.

THEOREM 5.10 **[33]** *There exists a data structure supporting a sequence of m find and split operations in $O(m\alpha(m,n))$ worst-case time. The space required is $O(n)$.*

La Poutré [56] proved that this bound is tight for (both separable and nonseparable) pointer-based algorithms.

THEOREM 5.11 **[56]** *Any pointer-based algorithm requires $\Omega(n + m\alpha(m,n))$ time to perform $(n-1)$ split and m find operations.*

Using the power of a random access machine, Gabow and Tarjan were able to achieve $\Theta(1)$ amortized time for the interval split–find problem [34]. This bound is obtained by employing a slight variant of the data structure sketched in "Special Linear Cases."

---

[2] $\log^* n = \min\{i \mid \log^{[i]} n \leq 1\}$, where $\log^{[i]} n = \log\log^{[i-1]} n$ for $i > 0$ and $\log^{[0]} n = n$.

5.4 The Set Union Problem with Deunions

Mannila and Ukkonen [59] defined a generalization of the set union problem, which they called *set union with deunions*. In addition to union and find, the following operation is allowed.

deunion: Undo the most recently performed union operation not yet undone.

Motivations for studying this problem arise in logic programming, and more precisely in memory management of interpreters without function symbols [40, 60, 61, 96]. In Prolog, for example, variables of clauses correspond to the elements of the sets, unifications correspond to unions and backtracking corresponds to deunions [60].

Algorithms for Set Union with Deunions

The set union problem with deunions can be solved by a modification of Blum's data structure described in "Single-Operation Worst-Case Time Complexity." To facilitate deunions, we maintain a *union stack* that stores some bookkeeping information related to unions. Finds are performed as in "Single-Operation Worst-Case Time Complexity." Unions require some additional work to maintain the union stack. We now sketch which information is stored in the union stack. For sake of simplicity we do not take into account names of the sets (namely, we show how to handle unite rather than union operations): names can be easily maintained in some extra information stored in the union stack. Initially, the union stack is empty. When a type 1 union is performed, we proceed as in "Single-Operation Worst-Case Time Complexity" and then push onto the union stack a record containing a pointer to the old root r_B. Similarly, when a type 2 union is performed, we push onto the union stack a record containing a pointer to r_A and a pointer to r_B. Finally, when a type 3 union is performed, we push onto the union stack a pointer to the leftmost child of either r_B or r_A, depending on the two cases.

Deunions basically use the top stack record to invalidate the last union performed. Indeed, we pop the top record from the union stack, and check whether the union to be undone is of type 1, 2, or 3. For type 1 unions, we follow the pointer to r_B and delete the edge leaving this node, thus, restoring it as a root. For type 2 unions, we follow the pointers to r_A and r_B and delete the edges leaving these nodes and their parent. For type 3 unions, we follow the pointer to the node, and move it together with all its right sibling as a child of a new root.

It can be easily showed that this augmented version of Blum's data structure supports each union, find, and deunion in $O(\log n / \log\log n)$ time in the worst case, with an $O(n)$ space usage. This was proved to be a lower bound for separable pointer algorithms by Westbrook and Tarjan [97]:

THEOREM 5.12 **[97]** *Every separable pointer algorithm for the set union problem with deunions requires at least* $\Omega(\log n / \log\log n)$ *amortized time per operation.*

All of the union rules and path compaction techniques described in "Amortized Time Complexity" can be extended in order to deal with deunions using the same bookkeeping method (i.e., the union stack) described above. However, path compression with any one of the union rules leads to an $O(\log n)$ amortized algorithm, as it can be seen by first performing $(n-1)$ unions which build a binomial tree (as defined, for instance, in [89]) of depth $O(\log n)$ and then by repeatedly carrying out a find on the deepest leaf, a deunion, and a redo of that union. Westbrook and Tarjan [97] showed that using either one of the union rules combined with path splitting or path halving yield $O(\log n / \log\log n)$ amortized algorithms for the set union problem with deunions. We now describe their algorithms.

In the following, a union operation not yet undone will be referred to as *live*, and as *dead* otherwise. To handle deunions, again a *union stack* is maintained, which contains the roots made nonroots by live unions. Additionally, we maintain for each node x a *node stack* $P(x)$, which contains the pointers leaving

x created either by unions or by finds. During a path compaction caused by a find, the old pointer leaving x is left in $P(x)$ and each newly created pointer (x, y) is pushed onto $P(x)$. The bottommost pointer on these stacks is created by a union and will be referred to as a *union pointer*. The other pointers are created by the path compaction performed during the find operations and are called *find pointers*. Each of these pointers is associated with a unique union operation, the one whose undoing would invalidate the pointer. The pointer is said to be *live* if the associated union operation is live, and it is said to be *dead* otherwise.

Unions are performed as in the set union problem, except that for each union a new item is pushed onto the union stack, containing the tree root made nonroot and some bookkeeping information about the set name and either size or rank. To perform a deunion, the top element is popped from the union stack and the pointer leaving that node is deleted. The extra information stored in the union stack is used to maintain set names and either sizes or ranks.

There are actually two versions of these algorithms, depending on when dead pointers are removed from the data structure. *Eager algorithms* pop pointers from the node stacks as soon as they become dead (i.e., after a deunion operation). On the other hand, *lazy algorithms* remove dead pointers in a lazy fashion while performing subsequent union and find operations. Combined with the allowed union and compaction rules, this gives a total of eight algorithms. They all have the same time and space complexity, as the following theorem shows.

THEOREM 5.13 **[97]** *Either union by size or union by rank in combination with either path splitting or path halving gives both eager and lazy algorithms which run in* $O(\log n / \log\log n)$ *amortized time for operation. The space required by all these algorithms is* $O(n)$.

The Set Union Problem with Unlimited Backtracking

Other variants of the set union problem with deunions have been considered such as set union with arbitrary deunions [36, 63], set union with dynamic weighted backtracking [39], and set union with unlimited backtracking [9]. In this chapter, we will discuss only set union with unlimited backtracking and refer the interested readers to the references for the other problems.

As before, we denote a union not yet undone by live, and by dead otherwise. In the set union problem with unlimited backtracking, deunions are replaced by the following more general operation:

backtrack(i): Undo the last i live unions performed. i is assumed to be an integer, $i \geq 0$.

The name of this problem derives from the fact that the limitation that at most one union could be undone per operation is removed.

Note that this problem is more general than the set union problem with deunions, since a deunion can be simply implemented as backtrack(1). Furthermore, a backtrack(i) can be implemented by performing exactly i deunions. Hence, a sequence of m_1 unions, m_2 finds, and m_3 backtracks can be carried out by simply performing at most m_1 deunions instead of the backtracks. Applying either Westbrook and Tarjan's algorithms or Blum's modified algorithm to the sequence of union, find, and deunion operations, a total of $O((m_1 + m_2)\log n / \log\log n)$ worst-case running time will result. As a consequence, the set union problem with unlimited backtracking can be solved in $O(\log n / \log\log n)$ amortized time per operation. Since deunions are a special case of backtracks, this bound is tight for the class of separable pointer algorithms because of Theorem 5.12.

However, using either Westbrook and Tarjan's algorithms or Blum's augmented data structure, each backtrack(i) can require $\Omega(i \log n / \log\log n)$ in the worst case. Indeed, the worst-case time complexity of backtrack(i) is at least $\Omega(i)$ as long as one insists on deleting pointers as soon as they are invalidated by backtracking (as in the eager methods described in "Algorithms for Set Union with Deunions," since in this case at least one pointer must be removed for each erased union. This is clearly undesirable, since i can be as large as $(n - 1)$.

The following theorem holds for the set union with unlimited backtracking, when union operations are taken into account.

THEOREM 5.14 **[37]** *It is possible to perform each union, find and backtrack(i) in $O(\log n)$ time in the worst case. This bound is tight for nonseparable pointer algorithms.*

Apostolico et al. [9] showed that, when unites instead of unions are performed (i.e., when the name of the new set can be arbitrarily chosen by the algorithm), a better bound for separable pointer algorithms can be achieved:

THEOREM 5.15 **[9]** *There exists a data structure which supports each unite and find operation in $O(\log n / \log\log n)$ time, each backtrack in $O(1)$ time, and requires $O(n)$ space.*

No better bound is possible for any separable pointer algorithm or in the cell probe model of computation, as it can be shown by a trivial extension of Theorem 5.4.

5.5 Partial and Full Persistence

In this section we cover general techniques for partial and full persistence. The time complexities of these techniques will generally be expressed in terms of *slowdowns* with respect to the ephemeral query and update operations. The slowdowns will usually be functions of m, the number of versions. A slowdown of $T_q(m)$ for queries means, for example, that a persistent query to a version which is a data structure of size n is accomplished in time $O(T_q(m) \cdot Q(n))$ time, where $Q(n)$ is the running time of an ephemeral query operation on a data structure of size n.

Methods for Arbitrary Data Structures

The Fat Node Method

A very simple idea for making any data structure partially persistent is the *fat node* method, which works as follows. The m versions are numbered by integers from 1 (the first) to m (the last). We will take the convention that if a persistent query specifies version t, for some $1 \leq t \leq m$, then the query is answered according to the state of the data structure as it was after version t was created but before (if ever) version $t+1$ was begun.

Each memory location μ in the ephemeral data structure can be associated with a set $C(\mu)$ containing pairs of the form $\langle t, v\rangle$, where v is a value and t is a version number, sometimes referred to as the *time stamp* of v. A pair $\langle t, v\rangle$ is present in $C(\mu)$ if and only if (a) memory location μ was modified while creating version t and (b) at the completion of version t, the location μ contained the value v. For every memory location μ in the ephemeral data structure, we associate an auxiliary data structure $A(\mu)$, which stores $C(\mu)$ ordered by time stamp.

In order to perform a persistent query in version t we simulate the operation of the ephemeral query algorithm. Whenever the ephemeral query algorithm attempts to read a memory location μ, we query $A(\mu)$ to determine the value of μ in version t. Let t^* be the largest time stamp in $C(\mu)$ which is less than or equal to t. Clearly, the required value is v^* where $\langle t^*, v^*\rangle \in C(\mu)$. Creating version $m+1$ by modifying version m is also easy: if memory locations $\mu_1, \mu_2, \ldots$ were modified while creating version $m+1$, and the values of these locations in version $m+1$ were $v_1, v_2, \ldots$, we simply insert the pair $\langle m+1, v_i\rangle$ to $A(\mu_i)$ for $i = 1, 2, \ldots$.

If we implement the auxiliary data structures as red-black trees [21] then it is possible to query $A(\mu)$ in $O(\log |C(\mu)|) = O(\log m)$ time and also to add a new pair to $A(\mu)$ in $O(1)$ amortized time (this is

possible because the new pair will always have a time stamp greater than or equal to any time stamp in $C(\mu)$). In fact, we can even obtain $O(1)$ worst-case slowdown for updates by using a data structure given in [57]. Note that each ephemeral memory modification performed during a persistent update also incurs a space cost of $O(1)$ (in general this is unavoidable). We thus obtain the following theorem.

THEOREM 5.16 **[28]** *Any data structure can be made partially persistent with slowdown $O(\log m)$ for queries and $O(1)$ for updates. The space cost is $O(1)$ for each ephemeral memory modification.*

The fat node method can be extended to full persistence with a little work. Again, we will take the convention that a persistent query on version t is answered according to the state of the data structure as it was after version t was created but before (if ever) it was modified to create any descendant version. Again, each memory location μ in the ephemeral data structure will be associated with a set $C(\mu)$ containing pairs of the form $\langle t, v\rangle$, where v is a value and t is a version (the timestamp). The rules specifying what pairs are stored in $C(\mu)$ are somewhat more complicated. The main difficulty is that the versions in full persistence are only partially ordered. In order to find out the value of a memory location μ in version t, we need to find the deepest ancestor of t in the version tree where μ was modified (this problem is similar to the inheritance problem for object-oriented languages).

One solution is to impose a total order on the versions by converting the version tree into a *version list*, which is simply a pre-order listing of the version tree. Whenever a new version is created, it is added to the version list immediately after its parent, thus inductively maintaining the pre-ordering of the list. We now compare any two versions as follows: the one which is further to the left in the version list is considered smaller.

For example, a version list corresponding to the tree in Fig. 5.6 is $[a, b, c, f, g, h, i, j, l, m, n, o, k, d, e]$, and by the linearization, version f is considered to be less than version m, and version j is considered to be less than version l.

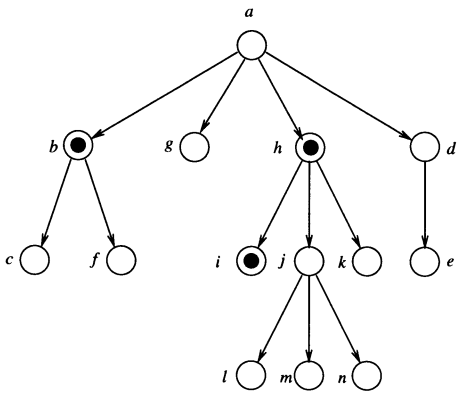

FIGURE 5.6 Navigating in full persistence: an example version tree.

Now consider a particular memory location π which was modified in versions b, h, and i of the data structure, with values B, H, and I being written to it in these versions. The following table shows the value of π in each version in the list (a $\perp$ means that no value has yet been written to π and hence its value may be undefined):

Version	a	b	c	f	g	h	i	j	l	m	n	o	k	d	e
Value	$\perp$	B	B	B	$\perp$	H	I	H	H	H	H	H	H	$\perp$	$\perp$

As can be seen in the above example, if π is modified in versions b, h and i, the version list is divided into

intervals containing respectively the sets $\{a\}$, $\{b, c, f\}$, $\{g\}$, $\{h\}$, $\{i\}$, $\{j, l, m, n, o, k\}$, $\{d, e\}$, such that for all versions in that interval, the value of π is the same. In general, the intervals of the version list for which the answer is the same will be different for different memory locations.

Hence, for each memory location μ, we define $C(\mu)$ to contains pairs of the form $\langle t, v\rangle$, where t is the leftmost version in its interval, and v is the value of μ in version t. Again, $C(\mu)$ is stored in an auxiliary data structure $A(\mu)$ ordered by time-stamp (the ordering among versions is as specified by the version list). In the example above, $C(\pi)$ would contain the following pairs:

$$\langle a, \perp\rangle, \langle b, B\rangle, \langle g, \perp\rangle, \langle h, H\rangle, \langle i, I\rangle, \langle j, H\rangle, \langle d, \perp\rangle .$$

In order to determine the value of some memory location μ in version t, we simply search among the pairs stored in $A(\mu)$, comparing versions, until we find the left endpoint of the interval to which t belongs; the associated value is the required answer.

How about updates? Let μ be any memory location, and firstly notice that if a new version is created in which μ is not modified, the value of μ in this new version will be the same as the value of μ in its parent, and the new version will be added to the version list right after its parent. This will simply enlarge the interval to which its parent belongs, and will also not change the left endpoint of the interval. Hence, if μ is not modified in some version, no change need be made to $A(\mu)$. On the other hand, adding a version where μ is modified creates a new interval containing only the new version, and in addition may split an existing interval into two. In general, if μ is modified in k different versions, $C(\mu)$ may contain up to $2k + 1$ pairs, and in each update, up to two new pairs may need to be inserted into $A(\mu)$. In the above example, if we create a new version p as a child of m and modify π to contain P in this version, then the interval $\{j, l, m, n, o, k\}$ splits into two intervals $\{j, l, m\}$ and $\{n, o, k\}$, and the new interval consisting only of $\{p\}$ is created. Hence, we would have to add the pairs $\langle n, H\rangle$ and $\langle p, P\rangle$ to $C(\pi)$.

Provided we can perform the comparison of two versions in constant time, and we store the pairs in say a red-black tree, we can perform a persistent query by simulating the ephemeral query algorithm, with a slowdown of $O(\log |C(\mu)|) = O(\log m)$, where m is the total number of versions. In the case of full persistence, updates also incur a slowdown of $O(\log m)$, and incur a $O(1)$ space cost per memory modification. Maintaining the version list so that two versions can be compared in constant time to determine which of the two is leftward is known as the *list order* problem, and has been studied in a series of papers [22, 90], culminating in an optimal data structure by Dietz and Sleator [24] which allows insertions and comparisons each in $O(1)$ worst-case time. We conclude:

THEOREM 5.17 **[28]** *Any data structure can be made fully persistent with slowdown $O(\log m)$ for both queries and updates. The space cost is $O(1)$ for each ephemeral memory modification.*

Faster Implementations of the Fat Node Method

For arbitrary data structures, the slowdown produced by the fat node method can be reduced by making use of the power of the RAM model. In the case of partial persistence, the versions are numbered with integers from 1 to m, where m is the number of versions, and special data structures for predecessor queries on integer sets may be used. For instance, the van Emde Boas data structure [91, 92] processes insertions, deletions, and predecessor queries on a set $S \subseteq \{1, \ldots, m\}$ in $O(\log \log m)$ time each. By using dynamic perfect hashing [27] to minimize space usage, the space required by this data structure can be reduced to linear in the size of the data structure, at the cost of making the updates run in $O(\log \log m)$ expected time. We thus obtain:

THEOREM 5.18 **[28, 27]** *Any data structure can be made partially persistent on a RAM with slowdown $O(\log \log m)$ for queries and expected slowdown $O(\log \log m)$ for updates. The space cost is $O(1)$ per ephemeral memory modification.*

At first sight it does not appear possible to use the same approach for full persistence because the versions are not integers. However, it turns out that algorithms for the list order problem work by assigning integer labels to the elements of the version list such that the labels increase monotonically from the beginning to the end of the list. Furthermore, these labels are guaranteed to be in the range $1..m^c$ where m is the number of versions and $c > 1$ is some constant. This means we can once again use the van Emde Boas data structure to search amongst the versions in $O(\log\log m)$ time. Unfortunately, each insertion into the version list may cause many of the integers to be relabeled, and making the changes to the appropriate auxiliary structures may prove expensive. Dietz [23] shows how to combine modifications to the list order algorithms together with standard bucketing techniques to obtain:

THEOREM 5.19 **[23]** *Any data structure can be made fully persistent on a RAM with slowdown $O(\log\log m)$ for queries and expected slowdown $O(\log\log m)$ for updates. The space cost is $O(1)$ per ephemeral memory modification.*

Methods for Linked Data Structures

The methods discussed above, while efficient, are not optimal and some of them are not simple to code. By placing some restrictions on the class of data structures which we want to make persistent, we can obtain some very simple and efficient algorithms for persistence. One such subclass of data structures is that of *linked* data structure.

A linked data structure is an abstraction of pointer-based data structures such as linked lists, search trees, etc. Informally, a linked data structure is composed of a collection of *nodes*, each with a finite number of named fields. Some of these fields are capable of holding an atomic piece of information, while others can hold a pointer to some node (or the value nil). For simplicity we assume the nodes are homogenous (i.e., of the same type) and that all access to the data structure is through a single designated *root* node. Any version of a linked data structure can be viewed as a directed graph, with vertices corresponding to nodes and edges corresponding to pointers.

Queries are abstracted away as *access operations* which consist of a series of *access steps*. The access algorithm has a collection of *accessed* nodes, which initially contains only the root. At each step, the algorithm either reads information from one of the accessed nodes or follows a non-nil pointer from one of the accessed nodes; the node so reached is then added to the set of accessed nodes. In actual data structures, of course, the information read by the query algorithm would be used to determine the pointers to follow as well as to compute an answer to return. Update operations are assumed to consist of an intermixed sequence of access steps as before and *update steps*. An update step either creates an explicitly initialized new node or writes a value to a field of some previously accessed node. We now discuss how one might implement persistent access and update operations.

Path Copying

A very simple but wasteful method for persistence is to copy the entire data structure after every update. *Path copying* is an optimization of this for linked data structures, which copies only "essential" nodes. Specifically, if an update modifies a version v by changing values in a set S of nodes, then it suffices to make copies of the nodes in S, together with all nodes that lie on a path from the root of version v to any node in S. The handle to the new version is simply a pointer to the new root. One advantage of this method is that traversing it is trivial: given a pointer to the root in some version, traversing it is done exactly as in the ephemeral case.

This method performs reasonably efficiently in the case of balanced search trees. Assuming that each node in the balanced search tree contains pointers only to its children, updates in balanced search trees such as AVL trees [2] and red-black trees [21] would cause only $O(\log n)$ nodes to be copied (these would be nodes either on the path from the root to the inserted or deleted item, or nodes adjacent to this path).

Note that this method does not work as well if the search tree only has an amortized $O(\log n)$ update cost, e.g., in the case of splay trees [87, p. 53 ff]. We therefore get the following theorem, which was independently noted by [74, 81].

THEOREM 5.20 *There is a fully persistent balanced search tree with persistent update and query times $O(\log n)$ and with space cost $O(\log n)$ per update, where n is the number of keys in the version of the data structure which is being updated or queried.*

Of course, for many other data structures, path copying may prove prohibitively expensive, and even in the case of balanced search trees, the space complexity is non-optimal, as red-black trees with lazy recoloring only modify $O(1)$ locations per update.

The Node Copying and Split Node Data Structures

An (ephemeral) *bounded-degree* linked data structure is one where the maximum *in-degree*, i.e., the maximum number of nodes that are pointing to any node, is bounded by a constant. Many, if not most, pointer-based data structures have this property, such as linked lists, search trees and so on (some of the data structures covered earlier in this chapter do not have this property). Driscoll et al. [28] showed that bounded-degree linked data structures could be made partially or fully persistent very efficiently, by means of the *node copying* and *split node* data structures respectively.

The source of inefficiency in the fat node data structure is searching among all the versions in the auxiliary data structure associated with an ephemeral node, as there is no bound on the number of such versions. The *node copying* data structure attempts to remedy this by replacing each fat node by a *collection* of "plump" nodes, each of which is capable of recording a bounded number of changes to an ephemeral node. Again, we assume that the versions are numbered with consecutive integers, starting from 1 (the first) to m (the last). Analogously to the fat node data structure, each ephemeral node x is associated with a set $C(x)$ of pairs $\langle t, r\rangle$, where t is a version number, and r is a record containing values for each of the fields of x. The set $C(x)$ is stored in a collection of plump nodes, each of which is capable of storing $2d+1$ pairs, where d is the bound on the in-degree of the ephemeral data structure.

The collection of plump nodes storing $C(x)$ is kept in a linked list $L(x)$. Let X be any plump node in $L(x)$ and let X' the next plump node in the list, if any. Let τ denote the smallest time stamp in X' if X' exists, and let $\tau = \infty$ otherwise. The list $L(x)$ is sorted by time stamp in the sense that all pairs in X are sorted by time stamp and all time stamps in X are smaller than τ. Each pair $\langle t, r\rangle$ in X is naturally associated with a *valid interval*, which is the half-open interval of versions beginning at t, up to, but not including the time stamp of the next pair in X, or τ if no such pair exists. The valid interval of X is simply the union of the valid intervals of the pairs stored in X. The following invariants always hold:

(i) For any pair $p = \langle t, r\rangle$ in $C(x)$, if a data field in r contains some value v then the value of the corresponding data field of ephemeral node x during the entire valid interval of p was also v. Furthermore, if a pointer field in r contains a pointer to a plump node in $L(y)$ or nil then the corresponding field in ephemeral node x pointed to ephemeral node y or contained nil, respectively, during the entire valid interval of p.

(ii) For any pair $p = \langle t, r\rangle$ in $C(x)$, if a pointer field in r points to a plump node Y, then the valid interval of p is contained in the valid interval of Y.

(iii) The handle of version t is a pointer to the (unique) plump node in $L(root)$ whose valid interval contains t.

A persistent access operation on version t is performed by a step-by-step simulation of the ephemeral access algorithm. For any ephemeral node x and version t, let $P(x, t)$ denote the plump node in $L(x)$ whose valid interval contains t. Since the valid intervals of the pairs in $C(x)$ are disjoint and partition

the interval $[1, \infty)$, this is well-defined. We ensure that if after some step, the ephemeral access algorithm would have accessed a set S of nodes, then the persistent access algorithm would have accessed the set of plump nodes $\{P(y,t)|y \in S\}$. This invariant holds initially, as the ephemeral algorithm would have accessed only $root$, and by (iv), the handle of version t points to $P(root, t)$.

If the ephemeral algorithm attempts to read a data field of an accessed node x then the persistent algorithm searches among the $O(1)$ pairs in $P(x,t)$ to find the pair whose valid interval contains t, and reads the value of the field from that pair. By (ii), this gives the correct value of the field. If the ephemeral algorithm follows a pointer from an accessed node x and reaches a node y, then the persistent algorithm searches among the $O(1)$ pairs in $P(x,t)$ to find the pair whose valid interval contains t, and follows the pointer specified in that pair. By invariants (i) and (ii) this pointer must point to $P(y,t)$. This proves the correctness of the simulation of the access operation.

Suppose during an ephemeral update operation on version m of the data structure, the ephemeral update operation writes some values into the fields of an ephemeral node x. Then the pair $\langle m+1, r\rangle$ is added to $C(x)$, where r contains the field values of x at the end of the update operation. If the plump node $P(x,m)$ is not full then this pair is simply added to $P(x,m)$. Otherwise, a new plump node that contains only this pair is created and added to the end of $L(x)$. For all nodes y that pointed to x in version m, this could cause a violation of (ii). Hence, for all such y, we add a new pair $\langle m+1, r'\rangle$ to $C(y)$, where r' is identical to the last record in $C(y)$ except that pointers to $P(x,m)$ are replaced by pointers to the new plump node. If this addition necessitates the creation of a new plump node in $L(y)$ then pointers to $P(m,y)$ are updated as above. A simple potential argument in [28] shows that not only does this process terminate, but the amortized space cost for each memory modification is $O(1)$. At the end of the process, a pointer to the last node in $L(root)$ is returned as the handle to version $m+1$. Hence, we have that:

THEOREM 5.21 **[28]** *Any bounded-degree linked data structure can be made partially persistent with worst-case slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized space cost $O(1)$ per memory modification.*

Although we will not describe them in detail here, similar ideas were applied by Driscoll et al. in the *split node* data structure which can be used to make bounded-degree linked data structures fully persistent in the following time bounds:

THEOREM 5.22 **[28]** *Any bounded-degree linked data structure can be made fully persistent with worst-case slowdown $O(1)$ for queries, amortized slowdown $O(1)$ for updates, and amortized space cost $O(1)$ per memory modification.*

Driscoll et al. left open the issue of whether the time and space bounds for Theorems 5.21 and 5.22 could be made worst-case rather than amortized. Toward this end, they used a method called *displaced storage of changes* to give a fully persistent search tree with $O(\log n)$ worst-case query and update times and $O(1)$ amortized space per update, improving upon the time bounds of Theorem 5.20. This method relies heavily on the property of balanced search trees that there is a unique path from the root to any internal node, and it is not clear how to extract a general method for full persistence from it. A more direct assault on their open problem was made by [25], which showed that all bounds in Theorem 5.21 could be made worst-case on the RAM model. In the same paper it was also shown that the space cost could be made $O(1)$ worst-case on the pointer machine model, but the slowdown for updates remained $O(1)$ amortized. Subsequently, Brodal [11] fully resolved the open problem of Driscoll et al. for partial persistence by showing that all bounds in Theorem 5.21 could be made worst-case on the pointer machine model. For the case of full persistence it was shown in [26] how to achieve $O(\log \log m)$ worst-case slowdown for updates and queries and a worst-case space cost of $O(1)$ per memory modification, but the open problem

of Driscoll et al. remains only partially resolved in this case. It should be noted that the data structures of [11, 26] are not much more complicated than the original data structures of Driscoll et al.

5.6 Functional Data Structures

In this section we will consider the implementation of data structures in functional languages. Although implementation in a functional language automatically guarantees persistence, the central issue is maintaining the same level of efficiency as in the imperative setting.

The state-of-the-art regarding general methods is quickly summarized. The path-copying method described at the beginning of the previous section can easily be implemented in a functional setting. This means that balanced binary trees (without parent pointers) can be implemented in a functional language, with queries and updates taking $O(\log n)$ worst-case time, and with a suboptimal worst-case space bound of $\Theta(\log n)$. Using the functional implementation of search trees to implement a dictionary which will simulate the memory of any imperative program, it is possible to implement any data structure which uses a maximum of M memory locations in a functional language with a slowdown of $O(\log M)$ in the query and update times, and a space cost of $O(\log M)$ per memory modification.

Naturally, better bounds are obtained by considering specific data structuring problems, and we summarize the known results at the end of this section. First, though, we will focus on perhaps the most fundamental data structuring problem in this context, that of implementing *catenable lists*. A catenable list supports the following set of operations:

makelist(a): Creates a new list containing only the element a.

head(X): Returns the first element of list X. Gives an error if X is empty.

tail(X): Returns the list obtained by deleting the first element of list X without modifying X. Gives an error if X is empty.

catenate(X, Y): Returns the list obtained by appending list Y to list X, without modifying X or Y.

Driscoll et al. [29] were the first to study this problem, and efficient but nonoptimal solutions were proposed in [16, 29]. We will sketch two proofs of the following theorem, due to Kaplan and Tarjan [50] and Okasaki [71]:

THEOREM 5.23 *The above set of operations can be implemented in $O(1)$ time each.*

The result due to Kaplan and Tarjan is stronger in two respects. Firstly, the solution of [50] gives $O(1)$ worst-case time bounds for all operations, while Okasaki's only gives amortized time bounds. Also, Okasaki's result uses "memoization" which, technically speaking, is a side-effect, and hence, his solution is not purely functional. On the other hand, Okasaki's solution is extremely simple to code in most functional programming languages, and offers insight into how to make amortized data structures fully persistent efficiently. In general, this is difficult because in an amortized data structure, some operations in a sequence of operations may be expensive, even though the average cost is low. In the fully persistent setting, an adversary can repeatedly perform an expensive operation as often as desired, pushing the average cost of an operation close to the maximum cost of any operation.

We will briefly cover both these solutions, beginning with Okasaki's. In each case we will first consider a variant of the problem where the *catenate* operation is replaced by the operation *inject*(a, X) which adds a to the end of list X. Note that *inject*(a, X) is equivalent to *catenate*(*makelist*(a), X). Although this change simplifies the problem substantially (this variant was solved quite long ago [43]) we use it to elaborate upon the principles in a simple setting.

Implementation of Catenable Lists in Functional Languages

We begin by noting that adding an element a to the front of a list X, without changing X, can be done in $O(1)$ time. We will denote this operation by $a :: X$. However, adding an element to the end of X involves a destructive update. The standard solution is to store the list X as a pair of lists $\langle F, R\rangle$, with F representing an initial segment of X, and R representing the remainder of X, stored in reversed order. Furthermore, we maintain the invariant that $|F| \geq |R|$.

To implement an *inject* or *tail* operation, we first obtain the pair $\langle F', R'\rangle$, which equals $\langle F, a :: R\rangle$ or $\langle tail(F), R\rangle$, as the case may be. If $|F'| \geq |R'|$, we return $\langle F', R'\rangle$. Otherwise we return $\langle F' +\!\!+ reverse(R'), [\,]\rangle$, where $X +\!\!+ Y$ appends Y to X and $reverse(X)$ returns the reverse of list X. The functions $+\!\!+$ and *reverse* are defined as follows:

$$\begin{aligned} X +\!\!+ Y &= Y \text{ if } X = [\,], \\ &= head(X) :: (tail(X) +\!\!+ Y) \text{ otherwise}. \\ reverse(X) &= rev(X, [\,]), \text{ where :} \\ rev(X, Y) &= Y \text{ if } X = [\,], \\ &= rev(tail(X), head(X) :: Y) \text{ otherwise}. \end{aligned}$$

The running time of $X +\!\!+ Y$ is clearly $O(|X|)$, as is the running time of $reverse(X)$. Although the amortized cost of *inject* can be easily seen to be $O(1)$ in an ephemeral setting, the efficiency of this data structure may be much worse in a fully persistent setting, as discussed above.

If, however, the functional language supports *lazy evaluation* and *memoization* then this solution can be used as is. Lazy evaluation refers to delaying calculating the value of expressions as much as possible. If lazy evaluation is used, the expression $F' +\!\!+ reverse(R')$ is not evaluated until we try to determine its *head* or *tail*. Even then, the expression is not fully evaluated unless F' is empty, and the list $tail(F' +\!\!+ reverse(R'))$ remains represented internally as $tail(F') +\!\!+ reverse(R')$. Note that *reverse* cannot be computed incrementally like $+\!\!+$: once started, a call to *reverse* must run to completion before the first element in the reversed list is available. Memoization involves caching the result of a delayed computation the first time it is executed, so that the next time the same computation needs to be performed, it can be looked up rather than recomputed.

The amortized analysis uses a "debit" argument. Each element of a list is associated with a number of debits, which will be proportional to the amount of delayed work which must be done before this element can be accessed. Each operation can "discharge" $O(1)$ debits, i.e., when the delayed work is eventually done, a cost proportional to the number of debits discharged by an operation will be charged to this operation. The goal will be to prove that all debits on an element will have been discharged before it is accessed. However, once the work has been done, the result is memoized and any other thread of execution which require this result will simply use the memoized version at no extra cost. The debits satisfy the following invariant. For $i = 0, 1, \ldots$, let $d_i \geq 0$ denote the number of debits on the ith element of any list $\langle F, R\rangle$. Then:

$$\sum_{j=0}^{i} d_i \leq \min\{2i, |F| - |R|\}, \text{ for } i = 0, 1, \ldots$$

Note that the first (zeroth) element on the list always has zero debits on it, and so *head* only accesses elements whose debits have been paid. If no list reversal takes place during a *tail* operation, the value of $|F|$ goes down by one, as does the index of each remaining element in the list (i.e., the old $(i+1)$st element will now be the new ith element). It suffices to pay of $O(1)$ debits at each of the first two locations in the list where the invariant is violated. A new element *inject*ed into list R has no delayed computation associated with it, and is give zero debits. The violations of the invariant caused by an *inject* where no list reversal occurs are handled as above. As a list reversal occurs only if $m = |F| = |R|$ before the operation which caused the reversal, the invariant implies that all debits on the front list have been paid off before

the reversal. Note that there are no debits on the rear list. After the reversal, one debit is placed on each element of the old front list (to pay for the delayed incremental $++$ operation) and $m+1$ debits are placed on the first element of the reversed list (to pay for the reversal), and zero on the remaining elements of the remaining elements of the reversed list, as there is no further delayed computation associated with them. It is easy to verify that the invariant is still satisfied after discharging $O(1)$ debits.

To add catenation to Okasaki's algorithm, a list is represented as a tree whose left-to-right pre-order traversal gives the list being represented. The children of a node are stored in a functional queue as described above. In order to perform *catenate*(X, Y) the operation *link*(X, Y) is performed, which adds root of the tree Y is added to the end of the child queue for the root of the tree X. The operation *tail*(X) removes the root of the tree for X. If its children of the root are $X_1, \ldots, X_m$ then the new list is given by *link*$(X_1,$ *link*$(X_2, \ldots,$ *link*$(X_{m-1}, X_m)))$. By executing the *link* operations in a lazy fashion and using memoization, all operations can be made to run in $O(1)$ time.

Purely Functional Catenable Lists

In this section we will describe the techniques used by Kaplan and Tarjan to obtain a purely functional queue. The critical difference is that we cannot assume memoization in a purely functional setting. This appears to mean that the data structures once again have to support each operation in worst-case constant time. The main ideas used by Kaplan and Tarjan are those of *data-structural bootstrapping* and *recursive slowdown*. Data-structural bootstrapping was introduced by [29] and refers to allowing a data structure to use the same data structure as a recursive sub-structure.

Recursive slowdown can be viewed as running the recursive data structures "at a slower speed." We will now give a very simple illustration of recursive slowdown. Let a 2-queue be a data structure which allows the *tail* and *inject* operations, but holds a maximum of 2 elements. Note that the bound on the size means that all operations on a 2-queue can be can be trivially implemented in constant time, by copying the entire queue each time. A queue Q consists of three components: a front queue $f(Q)$, which is a 2-queue, a rear queue $r(Q)$, which is also a 2-queue, and a center queue $c(Q)$, which is a recursive queue, each element of which is a *pair* of elements of the top-level queue. We will ensure that at least one of $f(Q)$ is non-empty unless Q itself is empty.

The operations are handled as follows An *inject* adds an element to the end of $r(Q)$. If $r(Q)$ is full, then the two elements currently in $r(Q)$ are inserted as a pair into $c(Q)$ and the new element is inserted into $r(Q)$. Similarly, a *tail* operation attempts to remove the first element from $f(Q)$. If $f(Q)$ is empty then we extract the first pair from $c(Q)$, if $c(Q)$ is non-empty and place the second element from the pair into $f(Q)$, discarding the first element. If $c(Q)$ is also empty then we discard the first element from $r(Q)$.

The key to the complexity bound is that only every alternate *inject* or *tail* operation accesses $c(Q)$. Therefore, the recurrence giving the amortized running time $T(n)$ of operations on this data structure behaves roughly like $2T(n) = T(n/2) + k$ for some constant k. The term $T(n/2)$ represents the cost of performing an operation on $c(Q)$, since $c(Q)$ can contain at most $n/2$ pairs of elements, if n is the number of elements in Q as a whole. Rewriting this recurrence as $T(n) = \frac{1}{2}T(n/2) + k'$ and expanding gives that $T(n) = O(1)$ (even replacing $n/2$ by $n-1$ in the RHS gives $T(n) = O(1)$).

This data structure is not suitable for use in a persistent setting as a single operation may still take $\Theta(\log n)$ time. For example, if $r(Q)$, $r(c(Q))$, $r(c(c(Q)))\ldots$ each contain two elements, then a single *inject* at the top level would cause changes at all $\Theta(\log n)$ levels of recursion. This is analogous to carry propagation in binary numbers—if we define a binary number where for $i = 0, 1, \ldots,$ the ith digit is 0 if $c^i(Q)$ contains one element and 1 if it contains two (the 0th digit is considered to be the least significant) then each *inject* can be viewed as adding 1 to this binary number. In the worst case, adding 1 to a binary number can take time proportional to the number of digits.

A different number system can alleviate this problem. Consider a number system where the ith digit still has weight 2^i, as in the binary system, but where digits can take the value 0, 1 or 2 [19]. Further, we require that any pair of 2's be separated by at least one 0 and that the rightmost digit, which is not a 1 is a 0. This

number system is *redundant*, i.e., a number can be represented in more than one way (the decimal number 4, for example, can be represented as either 100 or 020). Using this number system, we can increment a value by one in constant time by the following rules: (i) add one by changing the rightmost 0 to a 1, or by changing x 1 to $(x+1)$ 0; then (ii) fixing the rightmost 2 by changing x 2 to $(x+1)$ 0. Now we increase the capacity of $r(Q)$ to 3 elements, and let a queue containing i elements represent the digit $i-1$. We then perform an *inject* in $O(1)$ worst-case time by simulating the algorithm above for incrementing a counter. Using a similar idea to make *tail* run in $O(1)$ time, we can make all operations run in $O(1)$ time.

In the version of their data structure which supports catenation, Kaplan and Tarjan again let a queue be represented by three queues $f(Q)$, $c(Q)$, and $r(Q)$, where $f(Q)$ and $r(Q)$ are of constant size as before. The center queue $c(Q)$ in this case holds either (i) a queue of constant size containing at least two elements or (ii) a pair whose first element is a queue as in (i) and whose second element is a catenable queue. To execute *catenate*(X, Y), the general aim is to first try and combine $r(X)$ and $f(Y)$ into a single queue. When this is possible, a pair consisting of the resulting queue and $c(Y)$ is *inject*ed into $c(X)$. Otherwise, $r(X)$ is *inject*ed into $c(X)$ and the pair $\langle f(X), c(X)\rangle$ is also *inject*ed into $c(X)$. Details can be found in [50].

Other Data Structures

A *deque* is a list which allows single elements to be added or removed from the front or the rear of the list. Efficient persistent deques implemented in functional languages were studied in [18, 35, 42], with some of these supporting additional operations. A *catenable deque* allows all the operations above defined for a catenable list, but also allows deletion of a single element from the end of the list. Kaplan and Tarjan [50] have stated that their technique extends to give purely functional catenable deques with constant worst-case time per operation. Other data structures which can be implemented in functional languages include finger search trees [51] and worst-case optimal priority queues [12]. (See [72, 73] for yet more examples.)

5.7 Research Issues and Summary

In this chapter we have described the most efficient known algorithms for set union and persistency.

Most of the set union algorithms we have described are optimal with respect to a certain model of computation (e.g., pointer machines with or without the separability assumption, random access machines). There are still several open problems in all the models of computation we have considered. First, there are no lower bounds for some of the set union problems on intervals: for instance, for nonseparable pointer algorithms we are only aware of the trivial lower bound for interval union–find. This problem requires $\Theta(1)$ amortized time on a random access machine as shown by Gabow and Tarjan [34]. Second, it is still open whether in the amortized and the single operation worst-case complexity of the set union problems with deunions or backtracking can be improved for nonseparable pointer algorithms or in the cell probe model of computation.

5.8 Defining Terms

Cell probe model: Model of computation where the cost of a computation is measured by the total number of memory accesses to a random access memory with $\lceil \log n \rceil$ bits cell size. All other computations are not accounted for and are considered to be free.

Persistent data structure: A data structure that preserves its old versions. Partially persistent data structures allow updates to their latest version only, while all versions of the data structure may be queried. Fully persistent data structures allow all their existing versions to be queried or updated.

Pointer machine: Model of computation whose storage consists of an unbounded collection of registers (or records) connected by pointers. Each register can contain an arbitrary amount of additional information, but no arithmetic is allowed to compute the address of a register. The only possibility to access a register is by following pointers.

Purely functional language: A language that does not allow any destructive operation—one which overwrites data—such as the assignment operation. Purely functional languages are side-effect-free, i.e., invoking a function has no effect other than computing the value returned by the function.

Random access machine: Model of computation whose memory consists of an unbounded sequence of registers, each of which is capable of holding an integer. In this model, arithmetic operations are allowed to compute the address of a memory register.

Separability: Assumption that defines two different classes of pointer-based algorithms for set union problems. An algorithm is separable if after each operation, its data structures can be partitioned into disjoint subgraphs so that each subgraph corresponds to exactly one current set, and no edge leads from one subgraph to another.

Acknowledgments

The work of the first author was supported in part by the Commission of the European Communities under project no. 20244 (ALCOM-IT) and by a research grant from University of Venice "Ca' Foscari." The work of the second author was supported in part by a Nuffield Foundation Award for Newly-Appointed Lecturers in the Sciences.

References

[1] Ackermann, W., Zum Hilbertshen Aufbau der reelen Zahlen, *Math. Ann.,* 99, 118–133, 1928.

[2] Adel'son-Vel'skii, G.M, and Landis, E.M., An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR,* 146, 263–266, (in Russian), 1962.

[3] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., On computing least common ancestors in trees, *Proc. 5th Annual ACM Symposium on Theory of Computing,* 253–265, 1973.

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

[5] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *Data Structures and Algorithms,* Addison-Wesley, Reading, MA, 1983.

[6] Ahuja, R.K., Mehlhorn, K., Orlin, J.B., and Tarjan, R.E., Faster algorithms for the shortest path problem, *J. Assoc. Comput. Mach.,* 37, 213–223, 1990.

[7] Aït-Kaci, H., An algebraic semantics approach to the effective resolution of type equations, *Theoret. Comput. Sci.,* 45, 1986.

[8] Aït-Kaci, H. and Nasr, R., LOGIN: A logic programming language with built-in inheritance, *J. Logic Program.,* 3, 1986.

[9] Apostolico, A., Italiano, G.F., Gambosi, G., and Talamo, M., The set union problem with unlimited backtracking, *SIAM J. Computing,* 23, 50–70, 1994.

[10] Arden, B.W., Galler, B.A., and Graham, R.M., An algorithm for equivalence declarations, *Comm. ACM,* 4, 310–314, 1961.

[11] Brodal, G.S., Partially persistent data structures of bounded degree with constant update time, Technical Report BRICS RS-94-35, BRICS, Department of Computer Science, University of Aarhus, 1994.

[12] Brodal, G.S. and Okasaki, C., Optimal purely functional priority queues, *J. Functional Programming,* to appear, 1996.

[13] Ben-Amram, A.M. and Galil, Z., On pointers versus addresses, *J. Assoc. Comput. Mach.,* 39, 617–648, 1992.

[14] Blum, N., On the single operation worst-case time complexity of the disjoint set union problem, *SIAM J. Comput.,* 15, 1021–1024, 1986.

[15] Blum, N. and Rochow, H., A lower bound on the single-operation worst-case time complexity of the union–find problem on intervals, *Inform. Proc. Lett.,* 51, 57–60, 1994.

[16] Buchsbaum, A.L. and Tarjan, R.E., Confluently persistent deques via data-structural bootstrapping, *J. Algorithms,* 18, 513–547, 1995.

[17] Chazelle, B., How to search in history, *Information and Control,* 64, 77–99, 1985.

[18] Chuang, T-R. and Goldberg, B., Real-time deques, multihead Turing machines, and purely functional programming, *Proceedings of the Conference of Functional Programming and Computer Architecture,* 289–298, 1992.

[19] Clancy, M.J. and Knuth, D.E., A programming and problem-solving seminar, Technical Report STAN-CS-77-606, Stanford University, 1977.

[20] Cole, R., Searching and storing similar lists, *J. Algorithms,* 7, 202–220, 1986.

[21] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms,* MIT Press, Cambridge, MA, 1990.

[22] Dietz, P.F., Maintaining order in a linked list, *Proc. 14th Annual ACM Symposium on Theory of Computing,* 122–127, 1982.

[23] Dietz, P.F., Fully persistent arrays, *Proc. Workshop on Algorithms and Data Structures (WADS '89), Lecture Notes in Computer Science,* 382, Springer-Verlag, Berlin, 67–74, 1989.

[24] Dietz, P.F. and Sleator, D.D., Two algorithms for maintaining order in a list, *Proc. 19th Annual ACM Symposium on Theory of Computing,* 365–372, 1987.

[25] Dietz, P.F. and Raman, R., Persistence, amortization and randomization, *Proc. 2nd Annual ACM-SIAM Symposium on Discrete Algorithms,* 77–87, 1991.

[26] Dietz, P.F. and Raman, R., Persistence, amortization and parallelization: On some combinatorial games and their applications, *Proc. Workshop on Algorithms and Data Structures (WADS '93), Lecture Notes in Computer Science,* 709, Springer-Verlag, Berlin, 289–301, 1993.

[27] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnhert, H., and Tarjan, R.E., Dynamic perfect hashing: upper and lower bounds, *Proc. 29th Annual IEEE Conference on the Foundations of Computer Science,* 1988, 524–531. The application to partial persistence was mentioned in the talk. A revised version of the paper appeared in *SIAM J. Computing,* 23, 738–761, 1994.

[28] Driscoll, J.R., Sarnak, N., Sleator, D.D., and Tarjan, R.E., Making data structures persistent, *J. Computer Systems Sci.,* 38, 86–124, 1989.

[29] Driscoll, J.R., Sleator, D.D.K., and Tarjan, R.E., Fully persistent lists with catenation, *J. ACM,* 41, 943–959, 1994.

[30] Dobkin, D.P. and Munro, J.I., Efficient uses of the past, *J. Algorithms,* 6, 455–465, 1985.

[31] Fischer, M.J., Efficiency of equivalence algorithms, in *Complexity of Computer Computations,* R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 153–168.

[32] Fredman, M.L. and Saks, M.E., The cell probe complexity of dynamic data structures, *Proc. 21st Annual ACM Symposium on Theory of Computing,* 345–354, 1989.

[33] Gabow, H.N., A scaling algorithm for weighted matching on general graphs, *Proc. 26th Annual Symposium on Foundations of Computer Science,* 90–100, 1985.

[34] Gabow, H.N. and Tarjan, R.E., A linear time algorithm for a special case of disjoint set union, *J. Comput. Sys. Sci.,* 30, 209–221, 1985.

[35] Gajewska, H. and Tarjan, R.E., Deques with heap order, *Information Processing Letters,* 22, 197–200, 1986.

[36] Galil, Z. and Italiano, G.F., A note on set union with arbitrary deunions, *Information Processing Letters,* 37, 331–335, 1991.

[37] Galil, Z. and Italiano, G.F., Data structures and algorithms for disjoint set union problems, *ACM Computing Surveys,* 23, 319–344, 1991.

[38] Galler, B.A. and Fischer, M., An improved equivalence algorithm, *Comm. ACM,* 7, 301–303, 1964.

[39] Gambosi, G., Italiano, G.F., and Talamo, M., Worst-case analysis of the set union problem with extended backtracking, *Theoret. Comput. Sci.,* 68, 57–70, 1989.

[40] Hogger, G.J., *Introduction to Logic Programming,* Academic Press, 1984.

[41] Italiano, G.F. and Sarnak, N., Fully persistent data structures for disjoint set union problems, *Proc. Workshop on Algorithms and Data Structures (WADS '91), Lecture Notes in Computer Science,* 519, Springer-Verlag, Berlin, 449–460, 1991.

[42] Hood, R., *The Efficient Implementation of Very-High-Level Programming Language Constructs,* Ph.D. Thesis, Cornell University, 1982.

[43] Hood, R. and Melville, R., Real-time operations in pure Lisp, *Information Processing Letters,* 13, 50–53, 1981.

[44] Hopcroft, J.E. and Karp, R.M., An algorithm for testing the equivalence of finite automata, TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, NY, 1971.

[45] Hopcroft, J.E. and Ullman, J.D., Set merging algorithms, *SIAM J. Comput.,* 2, 294–303, 1973.

[46] Hudak, P., Jones, S.P., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J., Report on the functional programming language Haskell, version 1.2, *SIGPLAN Notices,* 27, 1992.

[47] Huet, G., *Resolutions d'equations dans les langages d'ordre* 1, 2, . . . ω, Ph.D. Dissertation, Univ. de Paris VII, France, 1976.

[48] Hunt, J.W. and Szymanski, T.G., A fast algorithm for computing longest common subsequences, *Comm. Assoc. Comput. Mach.,* 20, 350–353, 1977.

[49] Imai, T. and Asano, T., Dynamic segment intersection with applications, *J. Algorithms,* 8, 1–18, 1987.

[50] Kaplan, H. and Tarjan, R.E., Persistent lists with catenation via recursive slow-down, *Proc. 27th Annual ACM Symposium on the Theory of Computing,* 93–102, 1995.

[51] Kaplan, H. Tarjan, R.E., Purely functional representations of catenable sorted lists, *Proc. 28th Annual ACM Symposium on the Theory of Computing,* 202–211, 1996.

[52] Karlsson, R.G., Algorithms in a restricted universe, Technical Report CS-84-50, Department of Computer Science, University of Waterloo, 1984.

[53] Kerschenbaum, A. and van Slyke, R., Computing minimum spanning trees efficiently, *Proc. 25th Annual Conf. of the ACM,* 518–527, 1972.

[54] Knuth, D.E., *The Art of Computer Programming,* Vol. 1: *Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1968.

[55] Kolmogorv, A.N., On the notion of algorithm, *Uspehi Mat. Nauk.,* 8, 175–176, 1953.

[56] La Poutré, J.A., Lower bounds for the union–find and the split–find problem on pointer machines, *Proc. 22nd Annual ACM Symposium on Theory of Computing,* 34–44, 1990.

[57] Levcopolous, C. and Overmars, M.H., A balanced search tree with $O(1)$ worst-case update time, *Acta Informatica,* 26, 269–278, 1988.

[58] Loebl, M. and Nešetřil, J., Linearity and unprovability of set union problem strategies, *Proc. 20th Annual ACM Symposium on Theory of Computing,* 360–366, 1988.

[59] Mannila, H. and Ukkonen, E., The set union problem with backtracking, *Proc. 13th International Colloquium on Automata, Languages and Programming* (ICALP 86), *Lecture Notes in Computer Science,* 226, Springer-Verlag, Berlin, 236–243, 1986.

[60] Mannila, M. and Ukkonen, E., On the complexity of unification sequences, *Proc. 3rd International Conference on Logic Programming, Lecture Notes in Computer Science,* 225, Springer-Verlag, Berlin, 122–133, 1986.

[61] Mannila, H. and Ukkonen, E., Timestamped term representation for implementing Prolog, *Proc. 3rd IEEE Conference on Logic Programming,* 159–167, 1986.

[62] Mannila, H. and Ukkonen, E., Space-time optimal algorithms for the set union problem with backtracking. Technical Report C-1987-80, Department of Computer Science, University of Helsinki, Finland.

[63] Mannila, H. and Ukkonen, E., Time parameter and arbitrary deunions in the set union problem, *Proc. 1st Scandinavian Workshop on Algorithm Theory* (SWAT 88), *Lecture Notes in Computer Science,* 318, Springer-Verlag, Berlin, 34–42, 1988.

[64] McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, *Commun. ACM,* 7, 184–195, 1960.

[65] Mehlhorn, K., *Data Structures and Algorithms,* Vol. 1: *Sorting and Searching,* Springer-Verlag, Berlin, 1984.

[66] Mehlhorn, K., *Data Structures and Algorithms,* Vol. 2: *Graph Algorithms and NP-Completeness,* Springer-Verlag, Berlin, 1984.

[67] Mehlhorn, K., *Data Structures and Algorithms,* Vol. 3: *Multidimensional Searching and Computational Geometry,* Springer-Verlag, Berlin, 1984.

[68] Mehlhorn, K. and Näher, S., Dynamic fractional cascading, *Algorithmica* 5, 215–241, 1990.

[69] Mehlhorn, K., Näher, S., and Alt, H., A lower bound for the complexity of the union–split–find problem, *SIAM J. Comput.,* 17, 1093–1102, 1990.

[70] Milner, R., Tofte, M., and Harper, R., *The Definition of Standard ML,* MIT Press, Cambridge, MA, 1990.

[71] Okasaki, C., Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking, *Proc. 36th Annual Symposium on Foundations of Computer Science,* 646-654, 1995.

[72] Okasaki, C., Functional Data Structures, in *Advanced Functional Programming, Lecture Notes in Computer Science,* 1129, Springer-Verlag, Berlin, 67–74, 1996.

[73] Okasaki, C., The role of lazy evaluation in amortized data structures, *Proc. 1996 ACM SIGPLAN International Conference on Functional Programming,* 62–72, 1996.

[74] Reps, T., Titelbaum, T., and Demers, A., Incremental context-dependent analysis for language-based editors, *ACM Transactions on Programming Languages and Systems,* 5, 449–477, 1983.

[75] Sarnak, N., *Persistent Data Structures,* Ph.D. Thesis, Department of Computer Science, New York University, 1986.

[76] Sarnak, N. and Tarjan, R.E., Planar point location using persistent search trees, *Commun. ACM,* 28, 669–679, 1986.

[77] Schönage, A., Storage modification machines, *SIAM J. Comput.,* 9, 490–508, 1980.

[78] Stearns, R.E. and Lewis, P.M., Property grammars and table machines, *Information and Control,* 14, 524–549, 1969.

[79] Stearns, R.E. and Rosenkrantz, P.M., Table machine simulation, *Conf. Rec. IEEE 10th Annual Symp. on Switching and Automata Theory,* 118–128, 1969.

[80] Steele Jr., G.L., *Common Lisp: The Language,* Digital Press, Bedford, MA, 1984.

[81] Swart, G.F., Efficient algorithms for computing geometric intersections, Technical Report 85-01-02, Department of Computer Science, University of Washington, Seattle, WA, 1985.

[82] Tarjan, R.E., Testing flow graph reducibility, *Proc. 5th Annual ACM Symp. on Theory of Computing,* 96–107, 1973.

[83] Tarjan, R.E., Finding dominators in directed graphs, *SIAM J. Comput.,* 3, 62–89, 1974.
[84] Tarjan, R.E., Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.,* 22, 215–225, 1975.
[85] Tarjan, R.E., A class of algorithms which require nonlinear time to maintain disjoint sets, *J. Comput. Sys. Sci.,* 18, 110–127, 1979.
[86] Tarjan, R.E., Application of path compression on balanced trees, *J. Assoc. Comput. Mach.,* 26, 690–715, 1979.
[87] Tarjan, R.E., *Data Structures and Network Algorithms,* SIAM, Philadelphia, PA, 1983.
[88] Tarjan, R.E., Amortized computational complexity, *SIAM J. Alg. Disc. Meth.,* 6, 306–318, 1985.
[89] Tarjan, R.E. and van Leeuwen, J., Worst-case analysis of set union algorithms, *J. Assoc. Comput. Mach.,* 31, 245–281, 1984.
[90] Tsakalidis, A.K., Maintaining order in a generalized linked list, *Acta Informatica,* 21, 101–112, 1984.
[91] van Emde Boas, P., Preserving order in a forest in less than logarithmic time and linear space, *Inform. Processing Lett.,* 6, 80–82, 1977.
[92] van Emde Boas, P., Kaas, R., and Zijlstra, E., Design and implementation of an efficient priority queue, *Math. Systems Theory,* 10, 99–127, 1977.
[93] van Leeuwen, J. and van der Weide, T., Alternative path compression techniques, Technical Report RUU-CS-77-3, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1977.
[94] van der Weide, T., *Data Structures: an Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms,* Mathematisch Centrum, Amsterdam, The Netherlands, 1980.
[95] Vitter, J.S. and Simons, R.A., New classes for parallel complexity: A study of unification and other complete problems for P, *IEEE Trans. Comput. C-35.* 1989.
[96] Warren, D.H.D. and Pereira, L.M., Prolog—the language and its implementation compared with LISP, *ACM SIGPLAN Notices,* 12, 109–115, 1977.
[97] Westbrook, J. and Tarjan, R.E., Amortized analysis of algorithms for set union with backtracking, *SIAM J. Comput.,* 18, 1–11, 1989.
[98] Westbrook, J. and Tarjan, R.E., Maintaining bridge-connected and biconnected components on-line, *Algorithmica,* 7, 433–464, 1992.
[99] Yao, A.C., Should tables be sorted? *J. Assoc. Comput. Mach.,* 28, 615–628, 1981.

Further Information

Research on advanced algorithms and data structures is published in many computer science journals, including *Algorithmica, Journal of ACM, Journal of Algorithms,* and *SIAM Journal on Computing.* Work on data structures is published also in the proceedings of general theoretical computer science conferences, such as the "ACM Symposium on Theory of Computing (STOC)," and the "IEEE Symposium on Foundations of Computer Science (FOCS)." More specialized conferences devoted exclusively to algorithms are the "ACM–SIAM Symposium on Discrete Algorithms (SODA)" and the "European Symposium on Algorithms (ESA)." Online bibliographies for many of these conferences and journals can be found on the World Wide Web.

Galil and Italiano [37] provide useful summaries on the state of the art in set union data structures. A in-depth study of implementing data structures in functional languages is given in [72].