REFERENCE MACHINES REQUIRE NON-LINEAR TIME TO MAINTAIN DISJOINT SETS

by

Robert E. Tarjan

STAN-CS-77-603
MARCH 1977

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

Reference Machines Require Non-Linear Time to Maintain Disjoint Sets

Robert Endre Tarjan

Computer Science Department
Stanford University
Stanford, California 94305

Abstract.

This paper describes a machine model intended to be useful. in deriving realistic complexity bounds for tasks requiring list processing. As an example of the use of the model, the paper shows that any such machine requires non-linear time in the worst case to compute unions of disjoint sets on-line. All set union algorithms known to me are instances of the model and are thus subject to the derived bound. One of the known algorithms achieves the bound to within a constant factor.

Keywords and Phrases: Ackermann's function, analysis of algorithms, concrete computational complexity, data structures, disjoint set union, equivalence relation, linking automaton, list processing, machine model, pointer, record, reference, storage manipulation machine.

'This research was supported in part by National Science Foundation grant MCS75-22870 and by the Office of Naval Research contract N00014-76-C-0668. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Some of this work was done while the author was visiting the Faculty of Mathematics at the University of Bielefeld, Bielefeld, West Germany.

Introduction.

Computer scientists have attempted for many years to derive lower bounds on the complexity of computational problems. This effort has met with some success, providing, for example, exponential lower bounds on the complexity of equivalence for regular expressions [13], validity in Presburger arithmetic [14], and circularity in attribute grammars [7]. In addition to these bounds for hard problems, several results for simpler problems exist, including bounds on the number of comparisons required for ordering problems [9], on the number of data accesses required for testing properties of graphs [15], and on the number of arithmetic operations required for evaluating various polynomials [2].

In spite of this progress, one domain, that of list processing problems, is almost entirely devoid of lower bound results. The subject of data structures is now part of the standard computer science curriculum, and every computer science library contains many books on the subject. Yet, with the exception of a few results on the relative power of various data structures, nothing is known about the inherent power of pointer manipulation.

One reason for this state of affairs is the lack of a thoroughly understood machine model which is both realistic and theoretically accessible. One candidate, the random access machine [1], which has been used by several authors to provide realistic measures of the complexity of various algorithms, seems too powerful to analyze easily. It also has certain defects, such as allowing unbounded parallelism if a "uniform cost" measure [1] is used.

However, another possible model exists. In 1953 Kolmogorov [11,12] proposed a machine which operates by manipulating pointers connecting nodes. Fifteen years later Knuth [8] proposed a similar machine, which he called

a linking automaton. Later and independently Schönhage[16] defined such a machine, which he called a storage manipulation machine, and showed that such machines can simulate Turing machines with multidimensional tapes in real time. Although these machines provide a useful tool for describing pointer manipulation algorithms, no bounds on their computations3 power except Schönhage's seem to exist.

This paper describes an extension of Knuth's machine, called a reference machine. The paper examines the ability of such a machine to solve a problem requiring manipulation of disjoint sets, and proves that any reference machine which solves the disjoint set problem requires non-linear time (in the worst case) to do so, under certain natural restrictions. The lower bound is tight to within a constant factor. This result shows that it is possible (in at least one case) to derive a non-linear lower bound on the complexity of a list-processing problem using a realistic computer model. The result also provides a partial solution to Knuth's exercise 2.6.1[8] which asks us to "Explore the properties of linking automata...".

2. Reference Machines.

A reference machine consists of a memory and a finite number of registers. The registers are of two types: data registers and reference registers. The memory consists of a finite but expandable pool of records. Each record consists of a finite number of items, each of which is either a data item or a reference item. Each item has an identifying name. All records are identical in structure; that is, they contain the same items.

A reference machine manipulates data and references. A reference either specifies a particular record or is null (ϕ). Each reference register and reference item can store one reference. Data can be of any kind whatsoever (integers, logical values, strings, real numbers, vectors, etc.). Each data register and data item can store one datum.

A program for a register machine consists of a sequence of instructions, numbered consecutively from one. Each instruction is of one of the following eight types. (Each r below denotes a reference register, each s denotes a data register, each t denotes a register of any type, and each n denotes an item name.)

$r \leftarrow \phi$ Place a null reference in register r.

$t_1 \leftarrow t_2$ (t_1 and t_2 must be of the same type).
Place the contents of register t_2 in register t_1, erasing what was there previously.

$t \leftarrow n(r)$ (n and t must be of the same type).
Let N be the n item of the record specified by the contents of r. Place the contents of N in register t, erasing what was there previously. (If r contains ϕ, this instruction does nothing.)

$n(r) \leftarrow t$ (n and t must be of the same type)

Let N be the n item of the record specified by the contents of r . Place the contents of t in item N , erasing what was there previously. (If r contains $\emptyset$, this instruction does nothing.)

$s_1 \leftarrow s_2 \ \theta \ s_3$ Combine the data in registers s_2 and s_3 by applying the operation θ . Store the result in s_1 , erasing what was there previously.

create r Create a new record (not specified by any existing reference) and place a reference to it in r .

halt Cease execution.

if condition then go to i

If the condition is true, then transfer control to instruction i. If the condition is false, do nothing.

Each condition in an if instruction is of one of the following types.

true Always true.

$t_1 = t_2$ (t_1 and t_2 must be of the same type)

True if the contents of t_1 and t_2 are the same.

$p(s_1, s_2)$ True if the contents of s_1 and s_2 satisfy the predicate p , where p is any predicate on data.

A reference machine executes a program instruction-by-instruction in consecutive order, beginning with instruction one. Execution of an *if* instruction may cause control to be transferred to a non-consecutive instruction, in which case consecutive execution resumes from this new instruction. When the machine reaches a *halt* instruction, execution ceases. The last instruction of every program is a *halt* .

A *reference machine step* consists of the execution of a single instruction. The *running time* of a reference machine program is the number of steps the machine requires to execute the program, as a function of the initial state of the registers and memory. The *storage space* required by a reference machine program is the number of records initially in memory plus the number created during execution.

When a new record is created all its items initially contain a special value called *undefined* (A) . The initial value of any register may also be Λ . If a reference machine attempts to use the contents of a register or item containing Λ , it halts. However, the machine is allowed to store another value into a register or item containing Λ .

I shall be uninterested in constant factors in running time and storage space. With this assumption, the register-to-register assignment is a redundant instruction type since it can be simulated by a *create* , a register-to-memory assignment, and a memory-to-register assignment. Similarly uses of the null reference value can be deleted without affecting running time by more than a constant factor, Extending the machine model by allowing several types of records has the effect only of saving a constant factor in storage space.

To completely specify a register machine, one must describe the data and the types of operations allowed on the data. Knuth's linking automaton is a register machine whose data consists of symbols selected from some set. No operations on data are allowed except testing for equality. Henceforth we shall use the term symbol in a technical sense to refer to data on which no operations are permitted except testing for equality.

A pure reference machine is a register machine with no data. It is not hard to show that any linking automaton with a finite set of symbols can be simulated by a pure reference machine with a loss of only a constant factor in running time. I shall consider examples of reference machines which have integers as data and addition and comparison as allowed operations. The lower bound-result holds for all reference machines, whatever their data.

In a reference machine, access to memory is by explicit reference only; no computation on references is possible. The reference machine model is thus apparently less powerful than the random access model with uniform cost measure [1]; reference machines lack the ability to use address arithmetic for such purposes as manipulating a hash table [9], performing a radix sort [9], or accessing a dense matrix [8]. These machines are, however, powerful enough to simulate such list-processing languages as LISP and to model the list-processing features of Algol-W, PL/1, and other general purpose languages.

It would of course be possible to study the general properties of reference machines, comparing their power with that of other classes of automata, as Schönhage[16] has done. Here, however, I analyze the ability of reference machines to solve a specific problem in list processing.

3. The Disjoint Set Union Problem.

Let $S_1, S_2, \ldots S_n$ be n disjoint sets, each containing a single element. The disjoint set union problem is to carry out a sequence of operations of the following two types on the sets.

find(x) : determine the name of the set containing element x .

union(A,B) : add all elements of set B to set A (destroying set B).

The operations are to be carried out on-line; that is, each instruction must be completed before the next one is known. We shall assume that the sequence of operations contains exactly n-1 union operations (so that after the last union all elements are in one set) and $m > n$ intermixed find operations (if $m < n$, some elements are never found).

The disjoint set union problem is an abstraction of the operations necessary to implement FORTRAN EQUIVALENCE and COMMON statements [5]. Algorithms for this problem and for a generalization of it have applications in graph theory [18], global code optimization [18,19], and linear algebra [19]. A number of algorithms exist [1,4,5,6].

A reference machine solution to the set union problem consists of a reference machine, a representation of the input sets as collections of records, a program for carrying out a find , and a program for carrying out a union . The reference machine solves the set union problem in the following way. Initially the machine memory represents the input sets. Each find is carried out by executing the find program, which halts having identified the set containing the desired element. Each union is carried out by executing the union program, which halts having modified the contents of memory to reflect the union. I shall make the following assumptions concerning the details of this process.

(3.1) Each set and each element has a distinct associated symbol.

(3.2) No record in the collection for an input set contains the symbol of any other set or of any element outside the set.

(3.3) No record in the collection for an input set contains a reference to any record outside the collection.

(3.4) Before the find program is executed to locate the set containing an element x , a reference to some record containing the symbol for x is placed in the designated input register r_1 and A is placed in all other registers. The find program halts with the symbol for the set containing x in the designated output register s_0 .

(3.5) Before the union program is executed to add elements in set B to set A, references to records containing the symbols for A and B are placed in the designated input registers r_1 and r_2 , respectively, and Λ is placed in all other registers. The union program halts with no output.

The sequence of steps associated with a set union problem and a reference machine solution is the sequence of steps executed by the machine when it carries out the finds and unions . The length of this sequence measures the total running time of the machine. The main result of this paper is a non-linear lower bound (as a function of n and m) on the length of any sequence of steps which solves a worst-case instance of the set union problem.

The formulation described above is intended to be realistic and to facilitate derivation of a lower bound. Assumption (3 .1) above, requiring that sets and elements be represented by symbols, makes it impossible to encode all elements of a set into a single datum and to move this datum at

a cost of one step per move; without this restriction there is a reference machine which can solve any set union problem in linear time. Assumptions (3.2), (3.3), and (3.4) imply that the machine, when performing a find on some element x , has access only to records representing the set containing x . Assumptions (3.2), (3.3), and (3.5) imply that the machine, when performing a union on sets A and B , has access only to records representing the sets A and B . It follows by induction on the number of finds and unions that (3.2) and (3.3) hold for the sets existing at any time during the computation, not just for the input sets. In other words, the contents of memory after any particular find or union can be partitioned into collections of records such that each collection corresponds to a currently existing set, all symbols for the set and its elements occur only in the corresponding collection of records, and no record in one collection contains a reference to a record in another collection. Without assumptions (3.2)-(3.5) any particular instance of the set union problem can be solved in linear time by initially moving symbols for all sets and elements into a single record and solving all finds by accessing only this record, though I conjecture that even without assumptions (3.2)-(3.5) no single reference machine can solve all instances of the set union problem in linear time,

If an algorithm for the set union problem is to be useful in practice, the symbol of each set and of each element should be stored in exactly one record,- so that the initialization for finds (3.4) and unions (3.5) is uniquely defined. All the algorithms to be considered have this property, but the lower bound proof does not require it.

4. Algorithms for the Set Union Problem.

All algorithms for the set union problem known to me can be implemented on reference machines. This section describes six such algorithms. These algorithms are of two general types, *quick find*, requiring constant time for each find, and *quick union*, requiring constant time for each union. All the algorithms represent each input set by a single record, containing the symbol for the corresponding set and the symbol for the corresponding element in data items *set* and *element*, respectively. Each element is permanently associated with the record containing its symbol, and no new records are ever created. During the computation, a currently existing set is represented by the collection of records corresponding to its elements and the symbol for the set is contained in exactly one of these records.

In the quick find method, each record contains two reference items, *parent* and *next*. One record in the collection representing a set contains the symbol of the set. The *parents* of all records in the collection refer to this *header* record. The *next* items link all records in the collection into a list whose first element is the header. Figure 4.1 illustrates this data structure.

[Figure 4.1]

With this representation, a find requires two reference machine steps; one to access the *parent* of the input record (which refers to the header) and one to access the *set* of the header. A union of A and B requires seven steps per element in B; each record in the collection for B must have its *parent* modified to refer to the header of A and must

be linked into the list for A . Table 4.1 contains programs in Algol-like notation for union and find . It is easy to translate these into reference machine programs.

[Table 4.1]

Adding a heuristic to the union program improves its performance considerably. Each record needs an additional data item, size . The size item is only meaningful for headers; it counts the number of elements in the corresponding set. To perform a union of A and B , the size of A is compared to the size of B . If B is smaller, the union proceeds as before. If A is smaller, the symbols for A and B in the headers of the sets are interchanged, the references in r_1 and r_2 to the headers are interchanged, and the union proceeds as before. The time required for such a weighted union is proportional to the size of the smaller of A and B. Table 4.2 contains a program for this heuristic.

[Table 4.2]

In the quick union method, each record contains only one reference item, parent . The collection of records representing a set forms a rooted tree*/ with the parent of each record referring to its parent in the tree;

---

*/ A rooted tree T is a connected, acyclic, undirected graph with a unique distinguished vertex r , called the root of T . If v and w are vertices of T such that v is on the (unique) simple path from r to w , then v is an ancestor of w and w is a descendant of v . This relationship is denoted by $v \overset{*}{\rightarrow} w$. The relationship $v \overset{*}{\rightarrow} w$ and $v \neq w$ is denoted by $v \overset{+}{\rightarrow} w$. If $v \overset{*}{\rightarrow} w$ and (v,w) is an edge of T , then v is the parent of w and w is a child of v . This relationship is denoted by $v \rightarrow w$. A leaf is a vertex with no children. The height of a vertex v is the length (number of edges) of the longest simple path from v to a descendant of v . The sub-tree of T rooted at vertex v is the subgraph of T induced by the descendants of v , with v as root.

the parent of the root is ϕ , The root contains the symbol of the set, Figure 4.2 illustrates this data structure.

[Figure 4.2]

With this representation, a union of A and B requires only one machine step, to place a reference to the root of A in the parent of the root of B . A find is performed by starting from the input record and following parent references until reaching a record with a null parent this record is the root of the tree representing the set and contains the set symbol. The find requires time proportional to the number of records on the path from the input record to the root. Table 4.3 contains programs for these versions of union and find .

[Table 4.3]

The weighted union heuristic can be added to quick union ; it uses extra time on unions but may save time on later finds . A heuristic for finds called path compression is also useful. After a find , every record on the path from the input record to the root has its parent modified to refer directly to the root, Path compression increases the running time of a find by a constant factor but may save time on later finds . Table 4.4 contains programs for union and find with these heuristics.

[Table 4.4]

The quick find algorithms are apparently part of the folklore of compiler construction; a description of these algorithms appears in [1]. The quick union algorithm with the weighted union heuristic was first presented in [5]. The path compression heuristic is apparently due to McIlroy and Morris [1]. Worst-case analysis of these algorithms appears in [1,4,5,6,17]; Table 4.5 summarizes the results. The theoretically best algorithm in the

worst case is quick union with both heuristics; its running time is $O(m\ \alpha(m,n))$, where $\alpha(m,n)$ is a functional inverse of Ackermann's function defined as follows.

For $i,j \geq 0$ let the function $A(i,j)$ be defined by

(4.1)
$$A(i,0) = 0 ;$$
$$A(0,j) = 2j \quad \text{for } j \geq 1 ;$$
$$A(i,1) = A(i-1,2) \quad \text{for } i \geq 1 ;$$
$$A(i,j) = A(i-1, A(i,j-1)) \quad \text{for } i \geq 1 , j \geq 2 .$$

Let

(4.2)
$$a(i,n) = \min\{j \mid A(i,j) > \log_2 n\}$$

and

(4.3)
$$\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor 2m/n \rfloor) > \log_2 n\} .$$ */

[Table 4.5]

Yao [21], Doyle and Rivest [3], and Knuth and Schönhage [10] have carried out average-time analyses of the algorithms for several reasonable probability measures under the assumption that m and n are proportional, Table 4.6 contains the results of Yao and Knuth and Schönhage for one measure (see [21]).

[Table 4.6]

The quick union algorithm is simpler and requires less storage than the quick find algorithm and is thus more useful in practice. Whether either of the two heuristics should be used with this algorithm depends upon the size of the problem and the cost of time versus the cost of space, The average running time of the quick union algorithm with path compression but without weighted union is unknown for the probability measure used by Yao and Knuth and Schönhage.

*/ For any real number x, $\lfloor x \rfloor$ denotes the greatest integer not larger than x .

When path compression is used, the running time of the quick union algorithm tends rapidly to $O(m)$ as m/n increases. For instance, if weighted union is not used and $m/n \geq cn^{1+\epsilon}$ for some positive constants c and ϵ, the running time is $O(m)$. If weighted union is used and $m/n \geq ca(k,n)$ for some positive constants c and k, the running time is $O(m)$. Note that $a(0,n)$ is $O(\log \log n)$ and $a(1,n)$ is $O(\log^* n)$, where

$$\log^* n = \min\{i \mid \overbrace{\log \log \ldots \log}^{i \text{ times}} n \leq 1\} .$$

The weighted union rule requires that records contain integer data items and that reference machines add and compare. It is natural to ask whether the weighted union rule can be implemented on a pure reference machine in such a way that the total time for all unions is $O(n)$. The answer is--yes.

Each non-negative integer is represented by a list which encodes the binary digits of the integer, A zero is encoded by a null pointer; a one is encoded by a non-null pointer. The digit list is singly linked from the low order digit to the high order digit. Figure 4.3 illustrates this representation.

[Figure 4.3]

Two integers are added by scanning the digit lists and adding digit-by-digit, propagating carries in the usual fashion. The scan stops after the end of the shorter list is reached and the last carry stops propagating. Two integers are compared by scanning both simultaneously and noting the highest order digit on which they differ. The scan need only extend to the end of the shorter digit list; the integer with the longer digit list must be larger. I leave as an exercise the implementation of these algorithms as register machine programs,

The n-1 union operations carried out by the quick union method perform the following arithmetic. Initially there are n integers, each

equal to one. During a union , two of the integers are compared and then added. After n-1 unions , a single integer equal to n remains. Since comparing two integers requires no more time than adding them, it will suffice to bound the time required by all the additions.

Lemma 4.1. Let a, b, c be integers such that a+b = c and let (a_i), (b_i), (c_i), respectively, be their binary digit lists ($a = \sum_{i=0}^{\infty} a_i 2^i$, $b = \sum_{i=0}^{\infty} b_i 2^i$, $c = \sum_{i=0}^{\infty} c_i 2^i$; $a_i, b_i, c_i \in \{0,1\}$). Let d_i be the carry from the i-th position when a and b are added. Then

$$\sum_{i=0}^{k} (a_i + b_i) = d_k + \sum_{i=0}^{k} (c_i + d_i)$$ for all k . In particular,

$$\sum_{i=0}^{\infty} (a_i + b_i) = \sum_{i=0}^{\infty} (c_i + d_i) .$$

Proof. For $i > 0$, $a_i + b_i + d_{i-1} = c_i + 2d_i$ (assuming $d_{-1} = 0$). Thus $a_i + b_i = c_i + d_i + (d_i - d_i\ 1)$. Summingfrom i = 0 to i = k gives the lemma. Cl

The time needed to add two binary integers by reference machine is proportional to the length of the shorter integer plus the number of carries. By Lemma 4.1, the total number of ones in the binary representations of both integers is equal to the number of ones in the binary representation of the sum plus the number of carries. Consider the arithmetic performed during the union operations, Initially, the total number of ones in the binary representations of all the set sizes is n . Each carry performed during an addition causes the total number of ones to decrease by one. Thus the total number of carries cannot exceed n-1, and the time required for all carries is $O(n)$.

It remains to bound the total length of the shorter of each pair of integers added during union operations. Let f(n) be a worst-case bound on this total length as a function of n . Then $f(1) = 0$, and

$$f(n) = \max\{\lfloor \log_2 k \rfloor + 1 + f(k) + f(n-k) \mid 1 \leq k \leq n/2\} \text{ for } n > 1 ,$$

since the length of the binary representation of k is $\lfloor \log_2 k \rfloor + 1$.

Lemma 4.2. $f(n) \leq 3n - 2 \log_2(n+1) - 1$.

Proof. By induction on n .

$$f(1) = 0 \leq 3 - 2 \log_2 2 - 1 .$$

$$f(2) = 1 \leq 6 - 2 \log_2 3 - 1 .$$

Let $n > 3$ and suppose the lemma is true for all values less than n . Let k be such that $1 \leq k \leq n/2$ and

$$f(n) = \lfloor \log_2 k \rfloor + 1 + f(k) + f(n-k) .$$

By the induction hypothesis

$$f(n) \leq \log_2 k + 1 + 3k - 2\log_2(k+1) - 1 + 3(n-k) - 2\log_2(n-k+1) - 1$$

$$\leq 3n - 1 - \log_2(k+1) - 2\log_2(n-k+1) \quad .$$

The function $-\log_2(k+1) - 2\log_2(n-k+1)$ for $1 \leq k \leq n/2$ is maximum when $k = 1$. Thus

$$f(n) \leq 3n - 1 - \log_2 2 - 2\log_2 n$$

$$< 3n - 2 - 2\log_2 n \quad .$$

Also, $n \geq 3$ implies $\sqrt{2}n \geq n+1$, which means

$$-2\log_2(n+1) \geq -2\log_2\sqrt{2}n = -2\log_2 n - 1 \ ,$$

and

$$f(n) \leq 3n - 2\log_2(n+1) - 1 \ . \ \square$$

It follows that the total time to perform all arithmetic associated with the union operations is $O(n)$, and the following theorem holds.

Theorem 4.1. There exists a pure reference machine which solves any disjoint set union problem in $O(m\ \alpha(m,n))$ time.

5. A Non-Linear Lower Bound.

This section shows that for all m and n there is a set union problem which requires at least $cm\alpha(m,n)$ steps to solve by reference machine, where c is a positive constant independent of m and n. Rather than consider reference machines, I consider sequences of reference machine steps. Given a set union problem, a sequence of reference machine steps is said to solve it if there is some reference machine, some set of union programs, one for each union, and some set of find programs, one for each find, such that when the sequence of programs corresponding to the sequence of union and find operations is executed according to the conventions of Section 3, the given sequence of reference machine steps results and the find programs produce correct answers. Note that any sequence of reference machine steps can be carried out by a non-branching reference machine program. The first step in the lower bound proof is to convert into a simple normal form any sequence of reference machine steps which solves a set union problem,

Theorem 5.1. Let S_1 be any sequence of reference machine steps which solves a set union problem, Then there is a sequence of reference machine steps S_2 which also solves the set union problem and has the following properties:

(5.1) $|S_2| \leq 2(m+n+|S_1|)$.

(5.2) S_2 manipulates no data except set and element symbols.

(5.3) S_2 represents each input set by a single record and contains no create instruction.

(5.4) S_2 fetches a symbol from memory only as the last instruction of a find.

Proof. Let S_1 be a sequence of reference machine steps which solves some set union problem. Delete from S_1 all steps which manipulate data other than set and element symbols. The sequence S_1 now satisfies (5.2) and still solves the set union problem.

The sequence S_2 to be constructed manipulates records corresponding to the sets, the elements, and the records manipulated by S_1. Initially the memory of S_2 consists of one record for each input set $A = \{a\}$. This record is the representative of the set A, of the element a, and of each record in the initial collection of records by which S_1 represents A. Each record created by S_1 also has a representative in the memory of S_2, defined as follows. The representative of a record created during execution of find(a) is the representative of a. The representative of a record created during execution of union(A,B) is the representative of A. For any object x (set, element, or record), let x^* denote the representative of x.

S_2 simulates S_1 step-by-step. If S_1 and S_2 are executed in parallel, the memory and registers of S_2 correspond to the memory and registers of S_1 in the following way.

(5.5) If R_1 and R_2 are records in the memory of S_1 such that R_1 contains a reference to R_2, then R_1^* contains a reference to R_2^* (unless $R_1^* = R_2^*$).

(5.6) If R is a record containing a set or element symbol x, then R^* contains a record to x^* and x^* contains a reference to R^* (unless $R^* = x^*$).

(5.7) If some register of S_1 contains a reference to a record R, then some register of S_2 contains a reference to R^*.

(5.8) If some regiser of S_1 contains a set or element symbol x , then some register of S_2 contains a reference to x^* .

(5.9) During execution of find(a) , S_2 maintains a reference to a* in a register. During execution of union(A,B) , S_2 maintains a reference to A* in a register.

Initially the memory of S_2 consists of all the representatives, each containing the symbol of the corresponding set, the symbol of the corresponding element, and no pointers. Properties (5.5) - (5.9) hold initially.

Let find(a) be a typical find. S_1 begins find(a) with a reference in r_1 to a record R containing the symbol for a . If (5.6) holds before the find, either $R^* = a^*$ or a* contains a reference to R* . S_2 begins the find with a reference to a* in r_1 . S_2's first step is to fetch a reference to R* into a register. This preserves (5.5) - (5.9).

Let union(A,B) be a typical union. S_1 begins union(A,B) with references in r_1 , r_2 to records R_1 , R_2 containing the symbols for A , B , respectively. If (5.6) holds before the find, either $R^* = A^*$ or A* contains a reference to R_1^* ; similarly either $R_2^* = B^*$ or B* contains a reference to R_2^* . S_2 begins the union with references to A* , B* in r_1 , r_2 , respectively. S_2's first two steps are to fetch references to R_1^* and R_2^* into registers. This preserves (5.5) - (5.9).

S_2 simulates each step of S_1 in the following way. Each time S_1 fetches a reference to a record R_2 from a record R_1 , S_2 fetches a reference to R_2^* from R_1^* (possible by (5.5)). Each time S_2 stores a reference to a record R_2 in a record R_1 , S_2 stores a reference to R_2^* in R_1^* (possible by (5.7)). Each

time S_1 fetches a set or element symbol x from a record R , S_2 fetches a reference to x^* from R* (possible by (5.6)). Each time S_1 stores a set or element symbol x into a record R , S_2 stores a reference to x^* in R^* and a reference to R* in x^* (possible by (5.7) and (5.8)). Each time S_1 creates a record, S_2 does nothing. At the end of each find, S_2 fetches the appropriate set symbol. Each of these steps preserves (5.5) - (5.9). The sequence S_2 constructed in this way carries out the finds and satisfies (5.1) - (5.4). □

One can represent the memory manipulated by a reference machine as an undirected graph, with one vertex R^* for each record R and one edge for each reference. If a record R_1 contains a reference to a record R_2, then (R_1^*, R_2^*) is an edge in the graph. This representation motivates the following definition, which reformulates the set union problem as a graph construction problem.

A link solution to a set union problem consists of a set of vertices V , one for each initial set and element, and a sequence of instructions of the form link(v,w) where $v, w \in V$. The sequence of link instructions constructs a graph edge-by-edge, starting from the graph with vertex set V and no edges; link(x,y) constructs edge (x,y) . For any initial set or element x , let x^* denote the corresponding vertex. The sequence of link instructions must satisfy the following properties.

(5.10) The sequence of links can be partitioned into contiguous subsequences, each subsequence corresponding to a union or find operation.

(5.11) Let find(a) with answer A be a typical find. Each link(x,y) in the subsequence for find(a) is such that $x = A^*$ and the distance between x and y in the graph existing before the

link is two. The instruction link(A^*, a^*) occurs either in the subsequence for find(a) or earlier in the sequence.

(5.12) Let union(A,B) be a typical union. Each link(x,y) in the subsequence for union(A,B) is such that $x = A^*$ and either $y = B^*$ or the distance between x and y in the graph existing before the link is two.

Theorem 5.2. Any set union problem solvable in k reference machine steps has a link solution of length not exceeding $5m + 4n + 4k$.

Proof. Let S_1 be a sequence of k reference machine steps which solves a set union problem. Let S_2 be a sequence of reference machine steps satisfying Theorem 5.1. Then $|S_2| \leq 2(m+n+k)$. From S_2 we construct a link solution S_3 satisfying the theorem. The vertex set for S_3 consists of one vertex R^* for each record R manipulated by S_2. If S_2 and S_3 are executed in parallel, the following properties hold.

(5.13) If a record R_1 contains a reference to a record R_2, then the distance between R_1^* and R_2^* is at most two.

(5.14) Let find(a) with answer A be a typical find. If during this find some register of S_2 contains a reference to R, then either $A^* = R^*$ or (A^*, R^*) is a previously constructed edge.

(5.15) Let union(A,B) be a typical union. If during this union some register of S_2 contains a reference to R, then either $A^* = R^*$ or (A^*, R^*) is a previously constructed edge.

S_3 simulates S_2 instruction-by-instruction, Certainly (5.13) - (5.15) hold initially. Let union(A,B) be a typical union. To begin the union,

S_3 links A* and B^*. This Preserves (5.13) - (5.15). Let find(a) with answer A be a typical find. Suppose S fetches ℓ items from memory while carrying out the find, If (5.13) holds before the find, there must be a path of length 2ℓ or less between A* and a* in the graph existing before the find, To begin the find S_3 links each vertex on this path to A* . This preserves (5.13) - (5.15).

Consider a subsequence of S_2 corresponding either to a find(a) with answer A or a union(A,B) . Suppose S_2 fetches a reference (say to R_2) from a record (say R_1). If (5.13) -(5.15) hold before the fetch, then there is a path between A* and R_2^* of length at most three. S_3 links each vertex on this path to A* , This preserves (5.13) - (5.15). All other instructions in S_2 do not affect (5.13) -(5.15); in order to store a reference (say R_2) in a record (say R_1), S_2 must first have references to R_1 and R_2 in registers, By (5.14) and (5.15) this means that the distance between R_1^* and R_2^* in the graph existing before the store is at most two.

The total length of the sequence S_3 constructed in this way is at most $5m + 4n + 4k$, and the sequence clearly solves the set union problem. □

In the following discussion I shall not distinguish between an initial set, its single element, and the vertex representing the set and the element, Corresponding to the sequence of unions in any set union problem is a rooted tree, called the union tree, whose vertices are the initial sets and whose edges are the pairs (A,B) such that union(A,B)

occurs in the sequence. The root of the tree is the set remaining after all unions are carried out. With this definition, every link(v,w) in a link solution to a set union problem has the property that $v \overset{+}{\rightarrow} w$ in the union tree. In the worst-case set union problems to be constructed below, the union tree is a complete binary tree.

The lower bound proof makes use of a rapidly growing function B(i,j) defined for i,j > 1 as follows.

(5.16)
$$B(1,j) = 1 \qquad \text{for } j > 1 ;$$
$$B(i,1) = B(i-1,2)+1 \qquad \text{for } i > 2 ;$$
$$B(i,j) = B(i,j-1) + B(i-1,2^{B(i,j-1)}) \qquad \text{for } i,j \geq 2 .$$

Lemma 5.1. $B(i,j)+1 \leq A(i,2j)$ for $i,j > 1$.

Proof. It is easy to show by induction that $A(i,j) < \min\{A(i+1,j), A(i,j+1)\}$ for $i > 0$, $j > 1$. Also,

(5.17)
$$A(i,j) = A(i-1,A(i,j-1)) = A(i-2,A(i-1,A(i,j-1)))$$
$$\geq 2^{2^{A(i,j-1)}} > 2^{A(i,j-1)+2} \qquad \text{for } i,j > 2 .$$

The lemma follows by double induction on i and j :

(5.18)
$$B(1,j)+1 = 2 \leq 2j = A(0,j) \leq A(1,2j) \qquad \text{for } j \geq 1 ;$$

(5.19)
$$B(i,1)+1 = B(i-1,2)+2 \leq A(i-1,4)+2 \leq A(i-1,6)$$
$$< A(i-1,A(2,1)) < A(i-1,A(i,1)) = A(i,2) \qquad \text{for } i > 2 ,$$
$$\text{if } B(i-1,2)+1 \leq A(i-2,4) ;$$

(5.20) $B(i,j)+1 = B(i,j-1) + B(i-1,2^{B(i,j-1)})+1$

$< A(i,2j-2) + A(i-1,2\cdot 2^{A(i,2j-2)})$

$< A(i-1,2\cdot 2^{A(i,2j-2)} + A(i,2j-2))$

$< A(i-1,2^{A(i,2j-2)+2})$

$\leq A(i-1,A(i,2j-1))$ by (5.17)

$= A(i,2j)$ for $i,j > 2$, if $B(i,j-1)+1 \leq A(i,2j-2)$ and $B(i-1,2^{B(i,j-1)})+1 < A(i-1,2\cdot 2^{B(i,j-1)})$,

□

Theorem 5.3. For any $k,s > 1$, let T be a complete binary tree of height $h > B(k,s)$. Let $\{v_i \mid 1 < i < s2^{B(k,s)}\}$ be a set of pairwise unrelated vertices in T, each of height strictly less than $h-B(k,s)$, such that exactly s vertices in $\{v_i\}$ occur in each subtree of T rooted at a vertex of height $h-B(k,s)$. Then for $n = 2^{h+1}-1$ and $m = s2^{B(k,s)}$ there is a set union problem for which

(5.21) the union tree is T;

(5.22) the set of finds is $\{\underline{find}(v_i) \mid 1 \leq i \leq m\}$;

(5.23) the answer to each find is a vertex of height strictly greater than $h-B(k,s)$; and

(5.24) any link solution has length at least km, even if every edge (v, w) such that $v \overset{+}{\rightarrow} w$ and $h(v) \leq h-B(k,s)$ in T is allowed for free, and after each $\underline{link}(v,w)$ every edge (x,y) such that $v \overset{*}{\rightarrow} x \rightarrow y \overset{*}{\rightarrow} w$ is added for free.

Proof. The proof is by double induction on k and s and is similar to the lower bound proof in [17]. Suppose $k = 1$. Consider any set union

problem consisting of $n-1$ unions which form T followed by a find on each vertex in $\{v_i\}$. The answer to each find is the root of T; (5.23) holds since $h > B(k,s)$. None of the originally free edges solves a find. Since the vertices in $\{v_i\}$ are pairwise unrelated, any link(x,y) can solve only one find, even including the appropriate free edges. Thus (5.24) holds.

Suppose the theorem holds for $k-1$, $s = 2$. The following argument proves the theorem for k with $s = 1$. Suppose the hypotheses of the theorem hold. Let $\{u_i \mid 1 \leq i \leq m\}$ be the set of vertices of height $h-B(k,1)$ in T, numbered so that $u_i \overset{+}{\rightarrow} v_i$. The vertices in $\{u_i\}$ are pairwise unrelated and exactly two occur in each subtree of T rooted at a vertex of height $h-B(k,1)+1 = h-B(k-1,2)$. By the induction hypothesis there is a set union problem satisfying the theorem for $k' = k-1$, $s' = 2$, T, $\{u_i\}$. Let the sequence of finds and unions in this set union problem be P_1. Form P_2 from P_1 by replacing each $\underline{\text{find}}(u_i)$ by $\underline{\text{find}}(v_i)$. I claim the resulting sequence satisfies the theorem for k, $s = 1$, T, $\{v_i\}$.

Certainly (5.21) - (5.23) hold. Consider any sequence S_2 of links which carries out P_2, allowing for free the edges described in (5.24). Form a sequence S_1 from S_2 by replacing each $\underline{\text{link}}(x,y)$ such that $v_I \overset{*}{\rightarrow} y$ for some (uniquely determined) i by $\underline{\text{link}}(x,u_i)$. Delete from S_1 all links which do not create new edges. I claim S_1 carries out P_1 (allowing appropriate edges for free) and that $|S_1| < |S_2|-m$.

The following property is true initially and is preserved if S_1 and S_2 are executed in parallel (on separate graphs).

(5.25) For $1 \leq i \leq m$, u_i is adjacent in the graph manipulated by S_1 to all vertices adjacent to at least one descendant of v_i in the graph manipulated by S_2.

It follows that S_1 carries out P_1.

For any v_i, consider the first link(x,y) in S_2 such that $x \xrightarrow{+} u_i \xrightarrow{+} v_i \xrightarrow{*} y$. There must be such a link since none of the initially free edges solves find(v_i) by (5.27). There must be a path of length two, say (x,z)(z,y), between x and y in the S_2 graph existing before the link, Furthermore z must satisfy $u_i \xrightarrow{*} z \xrightarrow{*} v_i$. It follows that (x,u_i) is an edge of the existing S_1 graph. Thus S_1 need not contain an instruction link(x, corresponding to link(x,y). This is true for any value of i. Hence $|S_1| \leq |S_2| - m$.

Since $(k-1)m < |S_1|$ by the induction hypothesis, $|S_2| < km$, and (5.24) holds.

Suppose the theorem holds for k, s-1 and also for k-1, B(k,s-1). The following argument proves the theorem for k, s. Suppose the hypotheses of the theorem hold. Let $\{w_i \mid 1 \leq i \leq 2^{B(k,s)}\}$ be a subset of $\{v_i\}$ such that exactly one vertex w_i occurs in each subtree of T rooted at a vertex of height h-B(k,s). Let $\{u_i \mid 1 < i < 2^{B(k,s)}\}$ be the set of vertices of height h-B(k,s), numbered so that $u_i \xrightarrow{+} w_i$.

Consider the sub-trees T_j, $1 \leq j \leq 2^{B(k,s)-B(k,s-1)}$, rooted at vertices of height $h\text{-}B(k,s) + B(k,s-1) = h\text{-}B(k-1,2^{B(k,s-1)})$ in T. Each sub-tree T_j contains $(s-1)2^{B(k,s-1)}$ vertices in $\{v_i\}-\{w_i\}$, exactly s-1 in each subtree rooted at a vertex of height h-B(k,s). By the induction hypothesis there is a set union problem satisfying the theorem for k' = k, s' = s-1, T_j, $(v \mid v$ is a vertex in T_j and $v \in \{v_i\}-\{w_i\}\}$. Let P_j be the sequence of unions and finds in this set union problem,

The vertices in the set $\{u_i\}$ are pairwise unrelated and exactly $2^{B(k, s-1)}$ occur in each subtree T_j of T. By the induction hypothesis there is a set union problem satisfying the theorem for $k' = k-1$, $s' = 2^{B(k,s-1)}$, T, $\{u_i\}$. Let Q be the sequence of unions and finds in this set union problem. The sequence Q can be permuted, without increasing the number of links required to carry out Q, so that all unions forming the subtrees T_j occur before all other operations.

Let Q' be formed from the permuted version of Q by deleting all unions forming the subtrees T_j, let Q'' be formed from Q' by replacing each $\underline{\text{find}}(u_i)$ by $\underline{\text{find}}(w_i)$, and let $P'' = P_1, P_2, \ldots, P_{2^{B(k,s)-B(k,s-1)}}, Q''$. I claim P'' defines a set union problem which satisfies the theorem for k, s, T, $\{v_i\}$.

Certainly (5.21) - (5.23) hold. Consider any sequence S'' of links which carries out P'', allowing for free the edges described in (5.24). Form a new sequence S from S'' by replacing each $\underline{\text{link}}(x,y)$ that $w_i \overset{*}{\rightarrow} y$ for some (uniquely determined) i by $\underline{\text{link}}(x,u_i)$. Delete from S all links which do not create new edges. The following property is true initially and is preserved if S and S'' are executed in parallel (on separate graphs).

(5.26) For $1 \leq i \leq 2^{B(k,s)}$, u_i is adjacent in the graph manipulated by S to all vertices adjacent to at least one descendant of w_i in the graph manipulated by S''.

It follows by an argument like that in the previous case that S carries out $P' = P_1, P_2, \ldots, P_{2^{B(k,s)-B(k,s-1)}}, Q'$ and that $|S| \leq |S''| - 2^{B(k,s)}$. S can be written as $S = S_1, S_2, \ldots, S_{2^{B(k,s)-B(k,s-1)}}, U$,

where S_i carries out Pi for $1 \leq i \leq 2^{B(k,s)-B(k,s-1)}$, allowing for free the edges described in (5.24), and U carries out Q', allowing for free the edges (v,w) such that $v \overset{+}{\rightarrow} w$ and $h(v) \leq h\text{-}B(k\text{-}1,2^{B(k,s-1)})$ and after each link(v,w) allowing for free the edges (x,y) such that $v \overset{*}{\rightarrow} x \overset{+}{\rightarrow} y \overset{*}{\rightarrow} w$. This means that U carries out Q, allowing the appropriate edges for free, By (5.24), $|S_i| \geq k(s-1)2^{B(k,s-1)}$ for $1 \leq i \leq 2^{B(k,s)-B(k,s-1)}$, and $|U| \geq (k-1)2^{B(k,s)}$. It follows that $|S''| \geq |S| + 2^{B(k,s)} \geq k(s-1)2^{B(k,s)} + (k-1)2^{B(k,s)} + 2^{B(k,s)} = ks2^{B(k,s)} = km$. Thus (5.24) holds, By double induction, the theorem is true in general. □

Corollary 5.1. Let $k,s > 1$. Let T be a complete binary tree of height $B(k,s)$. Then there is a set union problem whose union tree is T, which contains $m = s2^{B(k,s)}$ finds, and which requires at least $(k-1)m$ links for its solution.

Proof. Choose $\ell \geq 1$ such that $2^{\ell} > s$. Let T' be a complete binary tree formed by replacing each leaf of T by a complete binary tree of height ℓ. Let $\{v_i \mid 1 \leq i \leq m\}$ be any set of vertices satisfying the hypotheses of Theorem 5.3 for k, s, T'. For $1 \leq i \leq m$, let u_i be the vertex of height ℓ in T' such that $u_i \overset{+}{\rightarrow} v_i$. Let P' be a sequence of unions and finds defining a set union problem satisfying the conclusions of Theorem 5.3 for k, s, T', $\{v_i\}$. Without loss of generality we can assume that the unions which form the sub-trees of T' rooted at height ℓ occur at the front of P'.

Form P from P' by deleting the unions which form the sub-trees of T' rooted at height ℓ and replacing each find(v_i) by find(u_i). We claim P defines a set union problem satisfying the conclusions of the

corollary. Certainly P contains m finds and the union tree of P is T. Suppose S is a sequence of links which carries out P. Form S' from S by following each $\underline{\text{link}}(x,u_i)$ which solves a $\underline{\text{find}}(u_i)$ by $\underline{\text{link}}(x,v_i)$. Then S' carries out P' if all edges (v,w) with $h(v) < \ell$ are allowed for free, Thus $|S'| \geq km$, and $|S''| \geq (k-1)m$. □

Theorem 5.2, Lemma 5.1, and Corollary 5.1 combine to establish the main result of this paper.

Theorem 5.4. There is a positive constant c such that, for all $m \geq n \geq 1$, there is a set union problem consisting of m finds and n-1 intermixed unions whose solution by reference machine requires at least $cm\alpha(m,n)$ steps.

Proof. Let $s = \lfloor m/n \rfloor$. Choose k as large as possible such that $2^{B(k,s)+1}-1 < n$. Partition the n elements into as many sets as possible of size $2^{B(k,s)+1}-1$, plus leftover elements. At most n/2 elements are left over. On each set of $2^{B(k,s)+1}-1$ elements, define a set union problem satisfying Corollary 5.1. Concatenate these problems, add enough additional unions to combine all elements, including the leftovers, into a single set, and add enough additional finds to bring the total to m.

The resulting set union problem contains m finds, n-1 intermixed unions, and requires at least $(k-1)s2^{B(k,s)}\ n/2^{B(k,s)+2} = (k-1)sn/4 \geq (k-1)m/8$ links for its solution. By Theorem 5.2, this set union problem requires at least $(k-1)m/32 - 5m/4 - n > (k-73)m/32$ reference machine steps for its solution.

If $\alpha(m,n) > 2$, $k \geq \alpha(m,n)-1$ in this construction since

$$B(\alpha(m,n)-1,s)+1 \leq A(\alpha(m,n)-1,2s) \quad \text{by Lemma 5.1}$$

$$< \log_2 n \quad \text{by the definition of } \alpha .$$

Thus the selected set union problem requires at least $(\alpha(m,n)-74)m/32 > \alpha(m,n)m/64$ reference machine steps, if $\alpha(m,n) \geq 148$. But if $\alpha(m,n) \leq 148$, any set union problem requires at least $m \geq m\alpha(m,n)/148$ reference machine steps. Choosing $c = 1/148$ gives the theorem. □

Conclusions.

This paper has described a machine model, called a reference machine, suitable for analyzing list processing problems. The model is similar to several previously -proposed [8,11,12,16]. Reference machines are quite powerful; Schönhage [16] has shown that they can simulate Turing machines with multidimensional tapes in real time, and one can show that they can simulate random access machines with logarithmic cost in real time.

The paper has analyzed the ability of reference machines to compute disjoint set unions. Under certain natural restrictions, all reference machines require non-linear time to solve this problem. This lower bound characterizes the efficiency with which one can represent dynamic information of a certain kind in a list structure. The bound does not require that the machine be deterministic, or that the program of the machine be fixed while the problem size grows, or that the complexity of memory (number of fields per record) be fixed while the problem size grows.

This generality is achieved by making the assumption that the description of each set is stored separately and that moving the description of a set requires constant time per element. Without these assumptions the lower bound is not valid. I conjecture, however, that the lower bound holds if the separate storage assumption is replaced by an assumption about the complexity of memory; namely, that every record contains only a fixed number of fields independent of the problem size.

The paper has presented a number of known set union algorithms and has shown that they all fit into the reference machine model. One of the algorithm: achieves the lower bound to within a constant factor. This algorithm requires that arithmetic be performed, but the arithmetic can be simulated

using list processing with only a constant factor loss in running time. I believe that any algorithm, even one which uses address arithmetic, requires non-linear time to solve the set union problem. Proving such a statement seems to require a better understanding of random access machines.

The set union problem can be generalized to a problem requiring evaluation of functions defined on paths in trees. The techniques used here and in [17] lead to a non-linear lower bound for some special cases of this generalized problem [20]. Certain cases of the problem can be solved in almost-linear time by using complicated extensions of the best set union algorithm presented here [18]. Whether the most general version of the function evaluation problem can be solved in almost-linear time is unknown.

Acknowledgment.

I would like to thank Professor Wolfgang Paul for his thoughtful criticism and valuable insights which contributed substantially to the lower bound proof.

References

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

[2] A. Borodin and I. Munro, The Computational Complexity of Algebraic and Numeric Problems, Elsevier, New York (1975).

[3] J. Doyle and R. L. Rivest, "Linear expected time of a simple union-find algorithm," Info. Proc. Letters 5(1976),146-148.

[4] M. J. Fischer, "Efficiency of equivalence algorithms," Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York (1972),153-168.

[5] B. A. Galler and M. J. Fischer, "An improved equivalence algorithm," Comm. ACM 7(1964),301-303.

[6] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," SIAM J. Computing 2 (1973), 294-303.

[7] M. Jazayeri, W. F. Ogden, and W. C. Rounds, "The intrinsically exponential complexity of the circularity problem for attribute grammars," Comm. ACM 18(1975),697-706.

[8] D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1968).

[9] D. E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass.(1975).

[10] D. E. Knuth and A. Schönhage, "The expected linearity of a simple equivalence algorithm," Technical Report STAN-CS-77-599, Computer Science Department, Stanford University (1977).

[11] A. N. Kolmogorov, "On the notion of algorithm," Uspehi Mat. Nauk. 8 (1953), 175-176.

[12] A. N. Kolmogorov and V. A, Uspenskii, "On the definition of an algorithm," Uspehi Mat. Nauk. 13(1958),3-28; English translation in Amer. Math. Soc. Transl. II Vol.29(1963), 217-245.

[13] A. R. Meyer and L. J. Stockmeyer, "The equivalence problem for regular expressions with squaring requires exponential space," Proc. 13th Annual Symp. on Switching and Automata Theory, 1972, 125-129.

[14] M. J. Rabin and M. J. Fischer, "Super-exponential complexity of Presburger arithmetic," Project MAC Technical Memorandum 43, MIT (1974).

[15] R. Rives-t and J. Vuillemin, "A generalization and proof of the Anderaa-Rosenberg conjecture," Proc. Seventh Annual ACM Symp. on Theory of Computing (1975),6-11.

[16] A. Schönhage, "Real-time simulation of multidimensional Turing machines by storage modification machines," Project MAC Technical Memorandum 37, MIT (1973).

[17] R. E. Tarjan, "Efficiency of a good but not linear disjoint set union algorithm," Jour. ACM 22 (1975), 215-225.

[18] R. E. Tarjan, "Applications of path compression on balanced trees," Technical Report STAN-CS-75-512, Computer Science Dept., Stanford University (1975).

[19] R. E. Tarjan, "Solving path problems on directed graphs," Technical Report STAN-CS-75-528, Computer Science Dept., Stanford University (1975).

[20] R. E. Tarjan, "Complexity of monotone networks for computing conjunctions," Technical Report STAN-CS-76-553, Computer Science Dept., Stanford University (1976).

[21] A. C. Yao, "On the average behavior of set merging algorithms," Proc. Eighth Annual ACM Symp. on Theory of Computing (1976),192-195.

```
procedure quick find
    s_0 ← set(parent(r_1));

procedure slow union;
    while r_2 ≠ ∅ do
        begin
        save ← next(r_2);
        parent(r_2) ← r_1;
        next(r_2) ← next(r_1);
        next(r_1) ← r_2;
        r_2 ← save
        end; --
```

Table 4.1. Programs for find and union using the quick find data structure.

```
procedure slow weighted union;
   begin
   if size(r1) < size(r2) then
      begin
      set(r1) ↔ set(r2);
      r1 ↔ r2;
      end;
   size(r1) ← size(r1) + size(r2);
   slow union
   end;
```

Table 4.2. Program for weighted union heuristic with quick find data structure.

```
procedure quick union;
   parent(r2) ← r1;

procedure slow find;
   begin
   root ← r1;
   while parent(root) ≠ ∅ do root ← parent(root);
   s0 ← set(root)
   end;
```

Table 4.3. Programs for union and find using the quick union data structure.

```
procedure quick weighted union;
    if size(r1) < size(r2) then
        begin
        set(r2) ↔ set(r1);
        parent(r1) ← r2;
        size(r2) ← size(r1) + size(r2);
        end
    else begin
        parent(r2) ← r1;
        size(r1) ← size(r1) + size(r2)
        end;

procedure find with path compression;
    begin
    slow find;
    current ← r1;
    while parent(current) ≠ ∅ do
        begin
        save ← parent(current);
        parent(current) ← root;
        current ← save
    end end;
```

Table 4.4. Programs for weighted union and path compression heuristics with quick union data structure.

	Time	
Quick find	$O(mn)$	[1]
with weighted union	$O(m \log n)$	[1]
Quick union	$O(mn)$	[4]
with weighted union	$O(m \log n)$	[4]
with path compression	$O(m \cdot \max(1, \log(n^2/m)/\log(2m/n)))$	[17]
with both heuristics	$O(m\ \alpha(m,n))$	[17]

Table 4.5. Worst-case running times of set union algorithms.

	Time	
Quick find	$O(n^2)$	[21]
with weighted union	$O(n)$	[10]
Quick union	$O(n^2)$	[21]
with weighted union	$O(n)$	[10]
with path compression	?	
with both heuristics	$O(n)$	[10]

Table 4.6. Average running times of set union algorithms if m and n are proportional.

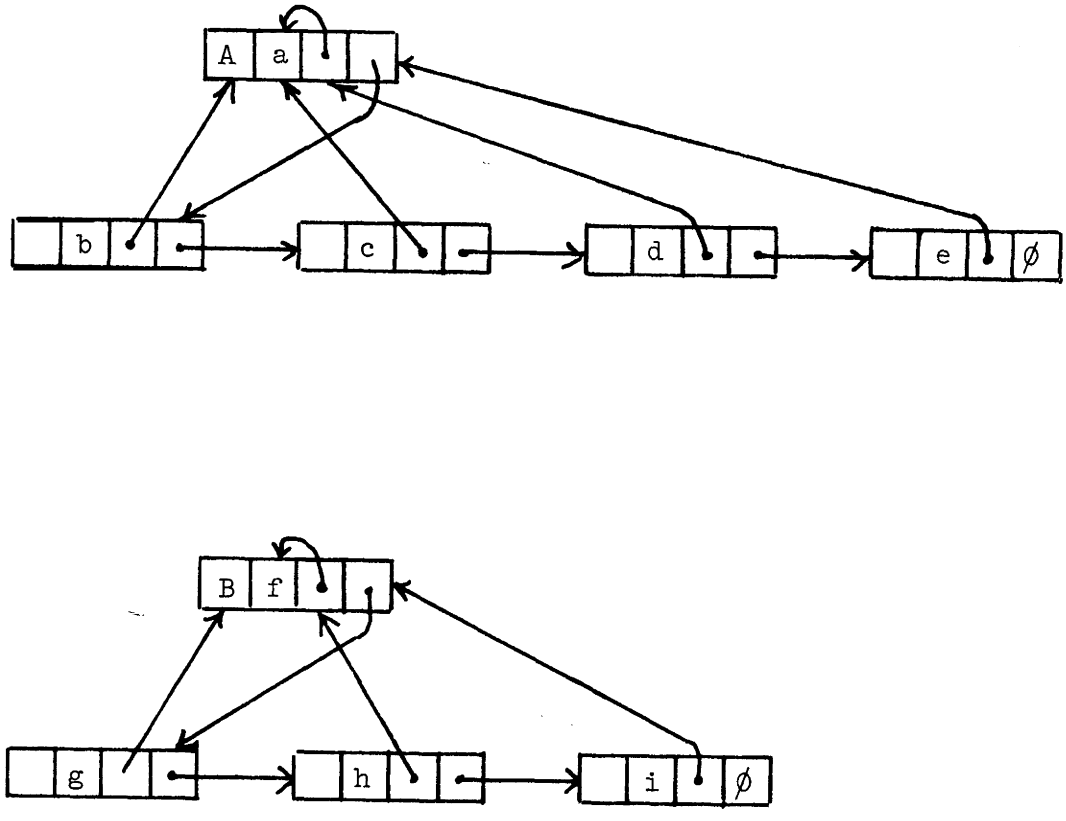

Figure 4.1. Data structure for quick find algorithm.

Sets are A = {a,b,c,d,e} , B = {f,g,h,i} .

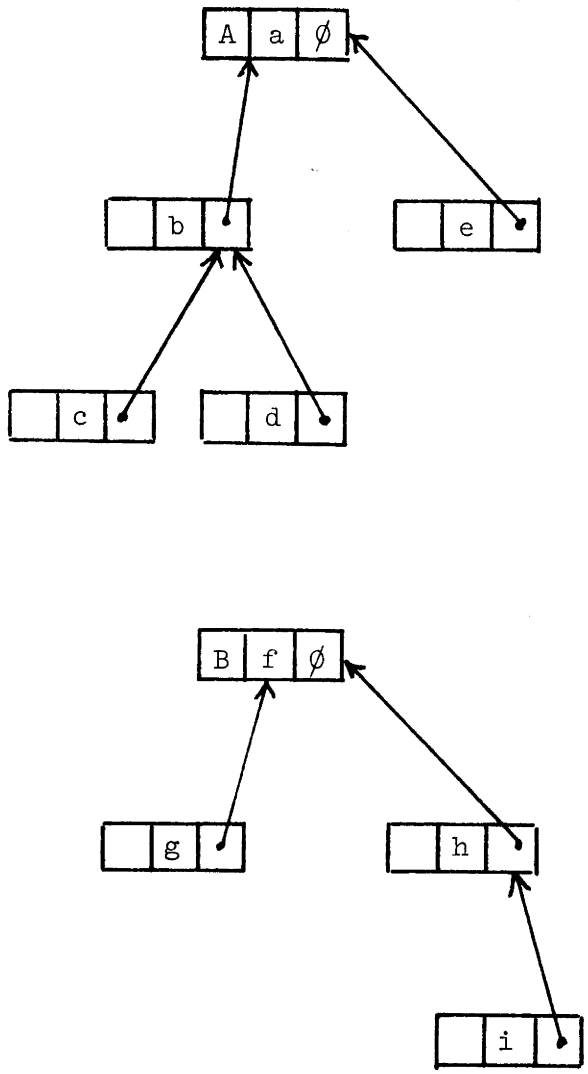

Figure 4.2. Data structure for quick union algorithm.

Sets are A = {a,b,c,d,e} , B = {f,g,h,i} .

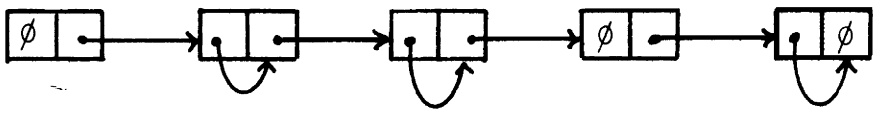

Figure 4.3. Representation of $26 = 10110_2$ as a list.