Union-find with deletions

Haim Kaplan *   Nira Shafrir †   Robert E. Tarjan ‡

Abstract

In the classical union-find problem we maintain a partition of a universe of n elements into disjoint sets subject to the operations union and find. The operation $union(A, B, C)$ replaces sets A and B in the partition by their union, given the name C. The operation $find(x)$ returns the name of the set containing the element x. In this paper we revisit the union-find problem in a context where the underlying partitioned universe is not fixed. Specifically, we allow a $delete(x)$ operation which removes the element x from the set containing it. We consider both worst-case performance and amortized performance. In both settings the challenge is to dynamically keep the size of the structure representing each set proportional to the number of elements in the set which may now decrease as a result of deletions.

For any fixed k, we describe a data structure that supports find and delete in $O(\log_k n)$ worst-case time and union in $O(k)$ worst-case time. This matches the best possible worst-case bounds for find and union in the classical setting. Furthermore, using an incremental global rebuilding technique we obtain a reduction converting any union-find data structure to a union-find with deletions data structure. Our reduction is such that the time bounds for find and union change only by a constant factor. The time it takes to delete an element x is the same as the time it takes to find the set containing x plus the time it takes to unite a singleton set with this set.

In an amortized setting a classical data structure of Tarjan supports a sequence of m finds and at most n unions on a universe of n elements in $O(n + m\alpha(m + n, n, \log n))$ time where $\alpha(m, n, l) = \min\{k \mid A_k(\lfloor \frac{m}{n} \rfloor) > l\}$ and $A_i(j)$ is Ackermann's function as described in [6]. We refine the analysis of this data structure and show that in fact the cost of each find is proportional to the size of the corresponding set. Specifically, we show that one can pay for a sequence of union and find operations by charging a constant to each participating element and $O(\alpha(m, n, \log(l)))$ for a find of an element in a set of size l. We also show how keep these amortized costs for each find and each participating element while allowing deletions. The amortized cost of deleting an element from a set of l elements is the same as the amortized cost of finding the element; namely, $O(\alpha(m, n, \log(l)))$.

1 Introduction

A union-find data structure allows the following operations on a collection of disjoint sets.

- *make-set*(x): Creates a set containing the single element x.
- $union(A, B, C)$: Combines the sets A and B into a new set C, destroying sets A and B.
- $find(x)$: Finds and returns (the name of) the set that contains x.

We can extend in a straightforward way a data structure supporting these operations to also support an $insert(x, A)$ operation that inserts an item x not yet in any set into set A. We perform *insert(x,A)* by first performing $B =$ *make-set*(x) followed by $union(A, B)$. The time it takes to perform *insert* is the time it takes to perform *make-set* plus the time it takes to perform *union* of a set with a single element with another set.

In this paper we study the *union-find with deletions* problem where we allow in addition to the three operations above a delete operation, defined as follows.

- $delete(x)$: Deletes x from the set that contains it. Note that the delete operation does not get the set containing x as a parameter.

Classical analysis of union-find data structures assumes a fixed universe of n items dynamically partitioned into a collection of disjoint sets. The starting point is a collection of n sets each containing a single item. Some of these sets are subsequently combined by performing unions but the underlying universe of items in all sets remains the original universe. In the union-find with deletions problem the underlying universe is a moving target. We can remove an item x from the universe by performing $delete(x)$ and we can add an item x to a set S by doing an $insert(x, S)$ operation.

We believe that the union-find with deletions problem is of general interest. A data structure that allows an efficient delete operation would be useful in any context where the partition that we maintain is not over a fixed set of items. Consider for example an application where the size of a set may be too large to fit into memory if we count deleted elements while without them it is much smaller and always fits into main memory. Our starting point for studying this problem was the classical meldable heap data type implemented for example

*School of Computer Science, Tel Aviv University, Tel Aviv 69978, Tel Aviv, Israel. haimk@math.tau.ac.il

†School of Computer Science, Tel Aviv University, Tel Aviv 69978, Tel Aviv, Israel. nira@math.tau.ac.il

‡Department of Computer Science, Princeton University, Princeton, NJ 08544, and InterTrust Technologies Corporation 4750 Patrick Henry Drive, Santa Clara, 95054-1851. ret@cs.princeton.edu

by Fibonacci heaps [4]. A data structure implementing this data type maintains an item-disjoint[1] set of heaps subject to the following operations.

make-heap: Return a new, empty heap.

insert(i, h): Insert a new item i with predefined key into heap h.

find-min(h): Return an item of minimum key in heap h. This operation does not change h.

delete-min(h): Delete an item of minimum key from heap h and return it.

meld(h_1, h_2): Return the heap formed by taking the union of the item-disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.

decrease-key(Δ, i, h): Decrease the key of item i in heap h by subtracting the nonnegative real number Δ. This operation assumes that the position of i in h is known.

delete(i, h): Delete item i from heap h. This operation assumes that the position of i in h is known.

Notice that the operations *decrease-key* and *delete* get both the target item and the heap containing it as parameters. Therefore in order to use this data structure we have to keep track of which heap contains an item while heaps undergo melds. One could do this using an external union-find data structure containing a set for each heap. When melding two heaps we unite the corresponding sets and when we need to find out which heap contains an item x we perform $find(x)$ and discover the corresponding set. Furthermore, when we delete an item from a heap we may want to be able to delete the item also from the corresponding set. This will prevent the union-find operations from becoming too expensive because they act on large sets.

One simple way to add a delete operation to any union-find data structure is simply by doing nothing during delete. Deleted items remain in the data structure mixed with live ones. The drawback of this simple scheme is that the size of the data structure does not remain proportional to the number of live items in it. As a result the space requirements of the data structure may become prohibitive and the performance of find operations is degraded.

For some applications, such as the incremental graph biconnectivity algorithm implemented in [5], this

straightforward implementation of delete suffices. The incremental graph biconnectivity algorithm keeps track of nodes containing deleted items and reuses them to store new items that are added to the set. As a result the total number of elements does not become too large.

In this paper we suggest some general techniques and data structures, not specific to a particular application, to overcome the difficulties that arise when deletions are allowed. We consider both worst-case and amortized performance.

In a worst-case setting, a classical result of Smid [7] (building upon a previous result of Blum [2]) gives for any fixed k a data structure that supports *union* in $O(k)$ time and *find* in $O(\log_k n)$ time where n is the size of the corresponding set. Our first result is a simple data structure with the same performance for *find* and *union* as the data structure of Smid, that also supports *delete* in $O(\log_k n)$ time, where n is the size of the set containing the element that we delete. Like Smid and Blum, we use k-ary balanced trees as our data structures. The essence of our result is a technique to keep such trees balanced when we perform deletions, while paying only $O(\log_k n)$ time per deletion. Note that with respect to *find* and *union* these time bounds are known to be optimal in the cell probe model [3]. (See also [1].) We believe that our technique for incrementally shrinking a k-ary tree that undergoes deletions, while keeping it balanced, may be useful in other contexts.

Still with respect to worst-case performance, we develop a general technique to add a delete operation to a union-find data structure that does not support delete to begin with. We use an incremental rebuilding technique, where each set is gradually being rebuilt without the deleted items in it. Let D be a union-find data structure that supports *find*, *union*, and *insert*, in $O(t_f(n))$, $O((t_u(n))$, and $O(t_i(n))$, respectively, where n is the maximum size of a set involved in the operation. Then by applying our transformation to D we obtain a data structure that supports *delete* from a set of size n in $O(t_f(n) + t_i(n))$ time, without hurting the time bounds of the other operations. By applying this reduction to the structure of Smid, we obtain an alternative to the data structure we obtained by directly modifying Smid's structure. This alternative also supports *union* in $O(k)$ time and find and delete in $O(\log_k n)$ time [2]. Our direct approach however, consumes less space than the data structure obtained via this reduction. We also point out that the $O(t_f(n))$ component in the time bound of *delete* stems from the need to find the corresponding set in order to delete from it. We can obtain a faster implementation of *delete* by this reduction if we assume that delete gets not only a pointer to the deleted element but also a pointer to the set containing it.

In an amortized setting Tarjan [8] and Tarjan and Van Leeuwen [?] showed that a classical data structure supports a sequence of m finds and at most n unions on a universe of n elements in $O(n + m\alpha(m + n, n, \log n))$ time where $\alpha(m, n, l) = \min\{k \mid A_k(\lfloor \frac{m}{n} \rfloor) > l\}$ and

[1] Items may have the same key.

[2] Smid's data structure supports insert in constant time.

$A_i(j)$ is Ackermann's function as defined in [6]. This data structure uses a tree to represent each set and its efficiency is due to two simple heuristics. The first heuristic called *union by rank* makes the root of the set of smaller rank a child of the root of the set of larger rank to carry out a union[3]. The second heuristic called *path compression* makes all nodes on a find-path children of the root. We refine the analysis of this data structure and show that in fact the cost of each find is proportional to the size of the corresponding set. Specifically, we show that one can pay for a sequence of union and find operations by charging a constant to each participating element and $O(\alpha(m, n, \log(l)))$ for a find of an element in a set of size l. Here m is the total number of finds and n is the total number of elements participating in the sequence of operations. This refined analysis raises the question of whether we can keep this time bound while adding a delete operation. In such a context we would like the charge per find to be $O(\alpha(m, n, \log(l)))$ where l is the actual size of the corresponding set when we perform the find.

We show that indeed this is possible. By marking items as deleted and rebuilding each set when the number of non-deleted items in it drops by a factor of 2 we can also support *delete* in $O(\alpha(m, n, \log(l)))$ amortized time where l is the size of the set containing the deleted item. This time bound for delete stems from the need to discover the set from which we delete in order to check whether we need to rebuild it. A possible drawback of set rebuilding is the bad worst-case time bound for delete (proportional to the size of the corresponding set). By applying the incremental set rebuilding technique of Section 3 to the union by rank with path compression data structure, we can preserve the amortized time bounds mentioned above while keeping the worst-case time bound of *delete* logarithmic. The space requirements however, will increase by a constant factor.

The structure of this paper is as follows. Section 2 describes a union-find with deletions data structure that builds upon the structure of Smid. In this section we develop a technique to keep a k-ary tree balanced while it undergoes deletions. Section 3 shows how to add a delete operation to any union-find data structure using incremental rebuilding. Section 4 refines the analysis of Kozen [6] for the compressed tree data structure, to show that the amortized time per find is proportional to the size of the set in which the find is performed. In Section 5 we show how to maintain these refined time bounds while allowing deletions. We conclude in Section 6 where we introduce some open questions. A simple presentation of Smid's data structure is provided in the Appendix.

[3] An alternative heuristic in which we make the union by *size* has similar performance.

2 Union-find with deletions via k-ary trees

In this section we extend the data structure of Smid [7] to support *delete* in $O(\frac{\log n}{\log k})$ time (A simple presentation of this data structure is provided in the Appendix.). we store each set in a tree such that the elements of the set reside at the leaves of the tree. We maintain the trees such that all leaves are at the same distance from the root. Each internal node v is classified as either *short*, *gaining*, or *losing*. Node v is *short* if $h(p(v)) > h(v)+1$, where $p(v)$ is the parent of v and $h(v)$ is the distance from v to a leaf. If v is not short then it is either gaining or losing. Each node v such that $h(v) > 1$ has exactly one gaining child. For every internal node v we denote by $g(v)$ its unique gaining sibling.

We represent the trees such that each node v points to its parent (except when v is the root), to its gaining child, to a list containing its short children and to a list containing its losing children. In addition each node is marked as *short*, *losing*, or *gaining*, and has a counter field that stores the number of its non-short children. If v is a root then it also stores the height of the tree and the name of the corresponding set.

Our forest satisfies the following invariant.

INVARIANT 2.1. *A root of a tree does not have short children.*

In addition if we cut subtrees rooted at short nodes from the trees of our union-find forest then the resulting set of trees will always satisfy the following invariants.

INVARIANT 2.2. *1) A root of height $h > 1$ has at least two children. A root of height $h = 1$ has at least one leaf child.*
2) A gaining node of height h has at least k children.
3) A losing node of height $h > 0$ has at least two children.

Each node v such that $h(v) > 1$ has a gaining child and therefore by invariant 2.2(2) v has at least k grandchildren of height $h-2$. This implies the following lemma.

LEMMA 2.1. *The height of a tree representing a set with n elements is $O(\log n/ \log k)$.*

2.1 operations

We will show how to implement *find*, *union*, and *delete* without violating any of the invariants.

Find(x): We follow parent pointers until we get to the root; we return the name of the set stored at the root.

Union(A,B,C): Let a be the root of A and let b be the root of B. Assume without loss of generality that the height of A is no greater than the height of B and that if A and B are of the same height then b has at least as many children as a. Recall that by invariant 2.1, a and b do not have short children. There are three cases.

1. The height of A is strictly smaller than the height of B. Let v be an arbitrary child of b. There are three subcases.
   Case a. $h(a) < h(v) - 1$. We make a a short child of v.
   Case b. $h(a) = h(v) - 1$. If a has less than k children we make the children of a be short children of v. To achieve this we change the parent pointers of the children of a to point to v and we add the children of a to the list of short children of v. We change the gaining child of a to be a short child of v and add it to the list of short children of v
   If, on the other hand, a has at least k children then we make a a losing child of v and add it to the list of losing children of v.
   Case c. $h(a) = h(v)$. If a has less than k children we make the children of a be losing children of v. To that end, we change the parent pointers at the children of a to point to v. We concatenate the list of losing children of a with the list of losing children of v, and we change the gaining child of a to be a losing child of v and add it to the list of losing children of v.
   If, on the other hand, a has at least k children we make a a losing child of b.

   In all three cases we update the number of children of b and v if it changes and store at b the name of the new set.

2. The trees A and B have equal heights, and the number of children of a is smaller than k. We make the children of a point to b instead of a. We concatenate the lists of losing children of a and b. We store the resulting list with b. We set the gaining child of a to be a losing child of b and add it to the list of losing children of b. We store in b the new name of the set, and increase the field that counts the number of children of b by the number of children of a.

3. The trees A and B have equal heights, and the number of children of a is at least k. We create a new root c. We make a and b be children of c. We make b a gaining child of c and a a losing child of c. We store with c the name of the new set, set its children counter to two, and set its height to be one greater than the height of a.

It follows immediately from the definition of *union* that the resulting tree satisfies Invariants 2.1 and 2.2. It is also easy to see that in all cases we change at most k pointers and therefore the running time of union is $O(k)$.

Delete(x): Let T be the tree in which x is a leaf. First we assume that there are no short nodes in T. Later we show how to extend the algorithm to the case where T may have short nodes.

Our deletion algorithm uses a recursive procedure $delete'$ that deletes a node w with no children from T. As a result of the deletion a new node with no children may be created, in which case the procedure applies itself recursively to the new node. We start the delete by applying $delete'$ to x. Any subsequent recursive application of $delete'$ is on a losing node that has lost all its children. Here is the definition of $delete'$. We denote the node to delete by w and its parent by v. The procedure $delete'$ has three cases.

1. **Node v is losing.** Delete w from the list of children of v. If after deleting w, v has one child, (a gaining child if $h(v) > 1$, a leaf otherwise), we move this child to be a losing child of $g(v)$ and apply $delete'$ to v.

2. **Node v is gaining.** Node v has a losing sibling u. We switch w with a losing child of u, say y, as follows. We add w to the list of losing children of u, and add y to the list of losing children of v. We change parent pointers in all the nodes that change parents. We continue as in Case 1.

3. **Node v is the root of the tree.** If w is the only child of v we get an empty set so we discard both w and v. If w is a leaf but it is not the only child of v we simply delete w. If w is not a leaf and w and $g(w)$ are the only children of v we discard w and v, and node $g(w)$ becomes the new root of the tree representing the set. We move the name of the set to $g(w)$ and set its height to be one less than the height of v.

We now extend the delete operation to the case where there are short nodes in T. We traverse the path from x to the root to discover whether x has a short ancestor. If x has a short ancestor, let v be the short ancestor of x closest to x. We delete x, if x was the only child of its parent we delete the parent of x, and we keep discarding ancestors of x until either we hit an ancestor with more than one child or we delete v. If x does not have a short ancestor, then we first simulate *delete'* as described above without actually performing the updates. If none of the nodes that would have been deleted by *delete'* and their gaining siblings have short children we perform *delete'* again, this time carrying out the updates. So assume that we discover a node v that has short children. We follow a path from v to a leaf x'. We replace x and x'. Then we delete x as described above in the case where x has a short ancestor.

The following lemma proves that our implementation is correct.

LEMMA 2.2. *A sequence of* union, find, *and* delete *operations on a forest that initially satisfies Invariants 2.1 and 2.2 results in forest that satisfies Invariants 2.1 and 2.2.*

Proof. It is easy to see that *union* produces a tree that satisfies Invariants 2.1 and 2.2 . Next we consider a *delete* operation. We assume that the nodes we refer to in the remainder of the proof don't have short ancestors.

When the root loses its next to last child, say w, and w is not a leaf, then $g(w)$ becomes the new root. Since the root may lose a child only if $g(w)$ does not have short children we obtain that Invariant 2.1 holds throughout the sequence. Since $g(w)$ has at least k children when it becomes a root then it follows that Invariant 2.2(1) also holds throughout the sequence.

Let v be a gaining node. Node v becomes gaining when it becomes a child of another node when performing union. At that point v has at least k children of height $h(v)-1$. As long as v is gaining, *delete* may not delete a child of v of height $h(v)-1$; *delete* may only replace a losing child of v with another losing child of a losing sibling. Therefore the number of children of height $h(v)-1$ of a gaining node remains at least k as long as the node is gaining, so Invariant 2.2(2) holds.

Since *delete'* recursively deletes every losing node that has less than 2 children we obtain that Invariant 2.2(3) holds throughout the sequence. When we create a new node v and $h(v)>1$ we assign a gaining child to it. The algorithm does not move or delete this child unless v is deleted too. Therefore every node v such that $h(v)>1$ always has a gaining child. ■

It is easy to see that each application of *delete'* takes O(1) time. Therefore the running time of delete is no greater than the height of the tree representing the corresponding set. For a set with n elements Lemma 2.1 shows that this height is at most $O(\log n/\log k)$.

3 Union-find with deletions using incremental copying

All the proposed algorithms for union-find represent a set by a tree and handle finds by following the path of ancestors to the root. The time bound for find and union can be stated in terms of the sizes of the sets involved, not in terms of the total universe size. In this section we show how to add a delete operation to any such union-find algorithm by using incremental copying. By applying this technique to the union-find data structure of Smid (described in Appendix A) we obtain a data structure with performance similar to the data structure of Section 2. The advantage of the data structure of Section 2 over the one we obtain here is its smaller (by a constant factor) space requirements.

Given a union-find data structure which supports find(x) in $O(t_f(n))$ worst-case time where n is the size of the set containing x, and insert (make-set + union in which one of the sets is a singleton set) in $O(t_i(n))$ worst-case time where n is the size of the set to which we insert the new item, we will augment this data structure to support delete(x) in $O(t_f(n)+t_i(n))$ worst-case time where n is the size of the set containing x, while keeping the worst-case time bounds for union and find the same as in the original data structure. In particular if we apply this technique to Smid's data structure we get a union-find data structure that supports delete in $O(\frac{\log n}{\log k})$ worst-case time, since Smid's structure supports insert in $O(1)$ time.

We represent each set S by one or two sets, each in a different union-find data structure without deletions. We denote the first such set by S_n and the second set if it exists by S_o. If x is an item in S then x is represented by a node either in S_n or in S_o. Item x points to the node representing it. In case x is represented by a node in S_n there may also be a node associated with x in S_o that is not being used any more.

At the beginning we represent S only by S_n, and S_o is empty. When we perform a delete operation on S we mark the item as deleted and increment the number of deleted items in S_n by one. We perform union of S and S' by uniting S_n with S'_n and S_o with S'_o. When at least 1/4 of the items in S_n are marked deleted we rename S_n to be S_o and start a new set S_n. Each time we delete an element from S and both S_n and S_o exist, we mark the item as deleted in the set that contains it and insert four items that are not marked deleted from S_o into S_n. We maintain S_o to contain at least four undeleted items that are not contained in S_n. If after renaming S_n into S_o or after we insert four undeleted items from S_o to S_n, S_o contains less than four items that are not in S_n, we insert these remaining items into S_n and discard S_o. When an item x from S_o is inserted into S_n we consider the node corresponding to x in S_n as representing x, and make x point to it. When there are no more undeleted items in S_o we discard it and represent S by S_n only. To establish the correctness of this algorithm we will show that S_o is empty when at least 1/4 of the items in S_n are marked deleted.

In order to implement this algorithm, with each set S_n and S_o we maintain a list of nodes, denoted by $L(S_n)$ and $L(S_o)$, respectively. The list $L(S_n)$ contains a node for each undeleted item in S_n. The node which corresponds to x in S_n has a pointer to the node associated with x in $L(S_n)$. The list $L(S_o)$ contains a node for each undeleted item in S_o that has not yet been inserted into S_n. The node which corresponds to x in S_o points to the node corresponding to x in $L(S_o)$. The node corresponding to x in $L(S_n)$ or $L(S_o)$ has a pointer to x. We also maintain the total number of items in S_n, and the number of items marked deleted in S_n. We assume that from the node identifying S we can get to the nodes identifying S_n and S_o and vice versa. We also assume that a find in S_n or S_o returns the node identifying S_n or S_o respectively, from which we can easily get to the node identifying S.

Next we describe the implementations of the operations using this representation.

Union(A, B, C)**:** We perform $union(A_n, B_n, C_n)$ and $union(A_o, B_o, C_o)$. We also concatenate $L(A_n)$ with $L(B_n)$ to form $L(C_n)$, and $L(A_o)$ with $L(B_o)$ to form $L(C_o)$. We set the number of items in C_n to be the sum of the number of items in A_n and the number of items in B_n. We similarly set the counter of the number of deleted items in C_n. We make the node identifying C point to the nodes identifying C_n and C_o and vice versa.

Find(x)**:** We perform find using the node identifying x in a union-find data structure without deletions. We get to a node identifying either S_n or S_o for some set S. From that node we get to the node identifying S and return it.

Delete(x)**:** We first perform find(x) on the node representing x in a union-find data structure without deletions as described above. We get the node identifying the set S containing x, and a node identifying the set among S_n and S_o containing x. We denote this set by S_x. We delete the node which corresponds to x in $L(S_x)$. If $S_x = S_n$ we also increment the number of deleted items in S_n. If the number of deleted items in S_n after the increment reaches $1/4$ of the total number of items in S_n then we rename S_n to S_o and set S_n to be empty.

Next if $L(S_o)$ is not empty, we remove four nodes (or less if there are less than four such items in $L(S_o)$) from $L(S_o)$ and insert them to S_n (performing 4 make set operations and 4 union operations of each of these new sets and S_n). We also insert four nodes corresponding to these 4 items into $L(S_n)$. If after removing these items from $L(S_o)$, $L(S_o)$ contains less than four items, we insert those items to S_n, insert corresponding nodes into $L(S_n)$, remove them from $L(S_o)$, and discard S_o.

To establish the correctness of this algorithm we will show that the fraction of deleted items in S_n is never greater than $1/4$. Furthermore, when the fraction of deleted items in S_n reaches $1/4$, S_o must be empty.

Lemma 3.1. *For every set S, at most $1/4$ of the items in S_n are deleted. When exactly $1/4$ of the items in S_n are deleted, S_o is empty, and we rename S_n to S_o.*

Proof. The proof is by induction on the sequence of operations. Assume the claim holds before the ith operation. If the ith operations is a find then the claim clearly holds after the ith operation. If the ith operation is a union(A, B, C) then since less than $1/4$ of A_n is deleted and less than $1/4$ of B_n is deleted then also less than $1/4$ of C_n is deleted. Therefore the claim also holds after the ith operation.

Assume that the ith operation is delete(x). Let S be the set containing x. If $x \in S_o$ and $x \notin S_n$ then the claim clearly holds after the deletion. If $x \in S_n$ and $S_o \neq \emptyset$ then the number of deleted items in S_n before the delete was less than $(1/4)|S_n|$. After the delete the number of deleted item increases by one but we also insert at least four new items from S_o. Therefore the fraction of deleted items in S_n is less than $((1/4)|S_n| + 1)/(|S_n| + 4) = 1/4$. It follows that the fraction of deleted items in S_n can reach $1/4$ only when S_o is empty. If indeed S_o is empty and the fraction of deleted items in S_n reaches $1/4$ we rename S_n to S_o and then insert items into a newly created S_n, so the fraction of deleted items in S_n is 0 after the delete. ■

The running time of the operations is dominated by the time it takes to manipulate the underlying union-find data structure without deletions. To analyze the running time of the operations on the underlying union-find data structure we bound the fraction of deleted items in any one of these sets. Lemma 3.1 proved that for every S the fraction of deleted items in S_n is at most $1/4$. We now show that a similar claim is also true for S_o. Specifically we show that at most $1/2$ of the items in S_o are deleted. Here when we count the number of items in S_o we do consider elements that were moved to S_n but nodes corresponding to them still belong to S_o. Furthermore, an element in S_o that was moved and subsequently deleted is counted as one of the deleted elements in S_o. When for some S we rename S_n to S_o, the fraction of deleted items in S_o is $1/4$. Subsequently we may delete more items in S_o, but the following lemma shows that by the time the fraction of deleted items in S_o reaches $1/2$ we have inserted all undeleted items in S_o into S_n and discarded S_o.

Lemma 3.2. *Let c be the fraction of deleted items in S_o. (Recall that c is the ratio between the number of* *deleted items in S_o and the total number of items in S_o including ones that have been inserted into S_n.) The number of undeleted items from S_o that have already been inserted into S_n is at least $(4c-1)|S_o|$.*

Proof. the proof is by induction on the number of operations. We assume that the claim holds before the ith operation. The claim clearly holds after the ith operation if the ith operation is a find. Assume the ith operation is delete(x). If S_o is empty after the delete, then the fraction of deleted items in it is defined as 0 and the claim holds. Otherwise let c be the fraction of deleted items in S_o before the ith operation and let c' be the fraction of deleted items in S_o after the ith operation. Clearly $c' \leq c + \frac{1}{|s_o|}$. Since we copied 4 undeleted items from S_o to S_n, the number of items already copied from S_o to S_n after the delete is at least $(4c-1)|S_o| + 4 = (4(c+1/|S_o|)-1)|S_o| \geq (4c'-1)|S_o|$ so the claim holds after the delete.

Assume the ith operation is union(A, B, C). Let a be the fraction of deleted items in A_o, and b be the fraction of deleted items in B_o. Clearly, $|C_o| = |A_o| + |B_o|$. The fraction of deleted items in C_o is $c = \frac{a|A_o|+b|B_o|}{|C_o|}$. The number of items already copied from C_o to C_n is at least

$$
\begin{aligned}
(4a-1)|A_o| + (4b-1)|B_o| = & \\
&= 4(a|A_o| + b|B_o|) - (|A_o| + |B_o|) \\
&= 4(a|A_o| + b|B_o|) - |C_o| \\
&= \left(\frac{4(a|A_o| + b|B_o|)}{|C_o|} - 1\right)|C_o| \\
&= (4c-1)|C_o|
\end{aligned}
$$

so the claim holds after the union as well. ∎

As a corollary we get that the fraction of deleted items in S_o is between 1/4 and 1/2.

LEMMA 3.3. *Let c be the fraction of items marked as deleted in S_o. Then $1/4 \leq c < 1/2$.*

Proof. When S_o is created (by renaming S_n), 1/4 of its items are deleted. Let c be the fraction of deleted items in S_o. By Lemma 3.2 the number of items already copied from S_o to S_n is at least $(4c-1)|S_o|$. If $c = 1/2$ then we have already copied all undeleted items from S_o to S_n and discarded S_o. Therefore $c < 1/2$. ∎

Lemma 3.3 and Lemma 3.1 show that the number of items in S_n and S_o is within a constant factor of the number of undeleted items in S. Therefore the time it takes to do union and find in our union-find data structure with deletions is proportional to the time it takes to do union and find, respectively, in the underlying union-find data structure without deletions. We perform a delete operation by a find followed by a constant number of insert operations. Therefore the time for delete is proportional to the time it takes to do a find plus the time it takes to do insert on the underlying union-find data structure without deletions. Assume that we start with a union find data structure without deletions in which union of sets of size at most n takes $O(t_u(n))$, find of an item in a set of size n takes $O(t_f(n))$ and insert into a set of size n takes $O(t_i(n))$. We obtain a union-find with deletions data structure in which union takes $O(t_u(n))$ time, find of an item in a set of size n takes $O(t_f(n))$, and delete of an item from a set of size n takes $O(t_f(n) + t_i(n))$.

4 Union-find via path compression and linking by rank or size – revisited

In this section and the following one n will denote the total number of elements involved in the sequence of operations, and m will denote the total number of finds. We represent each set by a rooted tree where each node points to its parent. We denote the parent of a node x by $p(x)$. Each node represents an element, and the root of a tree also represents the corresponding set. Each node has a *rank* associated with it. We maintain in the data structure the ranks of the roots. Ranks of other nodes are static and need not be maintained in the data structure. The *rank of a set* is defined to be the rank of the root of the tree representing the set.

We perform $find(x)$ by following parent pointers starting from x until we get to the root. We return the root. We also use a heuristic called *path compression*, where we make all nodes on the find path from x to the root children of the root. We perform *make-set*(x) by creating a new tree with x as its root. We set the rank of x to be 0. We perform $union(A, B, C)$ by making the root with smaller rank a child of the root with higher rank. In case $rank(A) = rank(B)$ we arbitrarily choose one of the roots and make it a child of the other. If $rank(A) = rank(B)$ we also increment the rank of the node that becomes the root of the new set C.

It is clear from the definitions of the operations that only the rank of a root node can increase by performing a *union*. Therefore, the rank of a node does not change once it stops being a root of a tree. It is easy to prove by induction that the number of nodes in the subtree rooted at a node of rank r is at least 2^r (see [8]).

To analyze the algorithm we use the following definition of Ackermann's function.

$A_0(x) = x + 1$ for $x \geq 1$,
$A_{k+1}(x) = A_k^{x+1}(x)$ for $x \geq 1$. (where $A^0(x) = x$ and $A_k^{i+1}(x) = A_k(A_k^i(x))$.

We also define the following three-parameter inverse to Ackermann's function $\alpha(m,n,l) = \min\{k \mid A_k(\lfloor \frac{m}{n} \rfloor) > l\}$.

We define the cost of each *union* or *make-set* operation to be one, and the cost of $find(x)$ to be equal to the number of vertices on the path from x to the root of the set containing it at the time of the find (inclusive). Thus, the actual cost of a sequence of operations is proportional to number of *union* and *make-set* operations, which is $O(n)$, plus the sum of the lengths of all find paths. We show that if we charge a constant to each participating element and charge $O(\alpha(m+n,n,r))$ to a find on a set of rank r then the sum of the charges suffices to pay for the sequence.

For a non-root node x, we define the *level* of x, $k(x) = \max\{k \mid A_k(r(x)) \leq r(p(x))\}$. We also define the *index* of x, $i(x)$, to be the largest i for which $r(p(x)) \geq A^i_{k(x)}(r(x))$. By the definition of Ackermann's function and the level function we have that $0 \leq i(x) \leq r(x)$.

The *threshold* of a find operation f is $t(f) = \alpha(m+n,n,r)$ where r is the rank of the corresponding set. We define S^i_f, for $0 \leq i < t(f)$ to be the set of all nodes on the find path whose level is i. We also define $S^{t(f)}_f$ to be the set of all nodes on the find path except the last whose level is at least $t(f)$.[4] We denote by L_f the set containing the last node on the find path in S^i_f, for every $0 \leq i \leq t(f)$, and all nodes on the find path whose rank is at most $t(f)$. We denote by N^i_f the set of nodes $v \in S^i_f - L_f$ for every $0 \leq i \leq t(f)$. Clearly the sets L_f and the sets N^i_f, $0 \leq i \leq t(f)$ partition the set of all the nodes on the find path except the root.

The find path contains at most one last node from each S^i_f, and at most one node of each rank smaller than $t(f)$. Therefore the number of nodes in L_f is at most $2t(f) = 2 * \alpha(m+n,n,r)$, where r is the rank of the corresponding set.

Next we count the total number of nodes in sets N^i_f, for $0 \leq i \leq \alpha(m+n,n,n)$, and for all find operations. We will count separately the nodes in sets $N^{t(f)}_f$ and the nodes in the sets N^i_f for $i < t(f)$. To count the total number of nodes in the sets N^i_f for $i < t(f)$ we will repartition them into multisets M_t, $1 \leq t \leq \alpha(m+n,n,n)$, defined as follows. The multiset M_t contains all nodes that occur in sets N^i_f where $t(f) = t$ and $i < t$.

We observe that node x cannot occur more than $r(x)$ times in a set N^i_f for a find f with $i < t(f)$. This is because each time x occurs in a set N^i_f its level must be i and its index is incremented. After $r(x)$ such increments the level of the node increases to $i+1$ and therefore it cannot belong to N^i_f if $i < t(f)$.

We have at most $n/2^r$ nodes of rank r, each occurring in M_t at most r times at a fixed level by the observation above. Therefore for every level $i < t$ there are at most $\sum_{r=t+1}^{\infty} \frac{n}{2^r} r$ nodes in M_t of level i. Summing over the t levels $0 \leq i < t$ we find that

$$M_t \leq t \sum_{r=t+1}^{\infty} \frac{n}{2^r} r.$$

By summing up the sizes of M_t for all values of t, $1 \leq t \leq \alpha(m+n,n,n)$, we find that

$$\sum_{t=1}^{\alpha(m+n,n,n)} M_t \leq \sum_{t=1}^{\alpha(m+n,n,n)} \frac{t}{2^t} \sum_{r=1}^{\infty} \frac{n}{2^r}(r+t) = O(n).$$

Last we count the number of nodes in sets $N^{t(f)}_f$ for all find operations. Suppose $x \in N^{t(f)}_f$ for some find operation f. We will show that $r(p(x)) < \lfloor \frac{m+n}{n} \rfloor$. From the definition of $N^{t(f)}_f$, it follows that $k(x) \geq t(f)$, and x is followed by another node y with $k(y) \geq t(f)$. Clearly, $r(y) \geq r(p(x))$. Let r be the rank of the set containing this find path. By the definition of $t(f)$ and $k(y)$ we have that $A_{t(f)}(\lfloor \frac{m+n}{n} \rfloor) > r \geq r(p(y)) \geq A_{k(y)}(r(y))$. Since $A_k(w)$ is increasing both in k and in w, it must be the case that $r(y) < \lfloor \frac{m+n}{n} \rfloor$, and therefore $r(p(x)) < \lfloor \frac{m+n}{n} \rfloor$. Since following each find where $x \in N^{t(f)}_f$, $r(p(x))$ increases by at least one, x cannot be in a set $N^{t(f)}_f$ more than $\lfloor \frac{m+n}{n} \rfloor$ times. Summing over all nodes we obtain that the number of nodes in sets $N^{t(f)}_f$ for all find operations is at most $m+n$.

Thus we have proved the following theorem

THEOREM 4.1. *: A sequence of m finds mixed with at most n unions on sets containing n elements takes $O(n+\sum_{i=1}^{m} \alpha(m+n,n,\log(n_i)))$ where n_i is the number of nodes in the set returned by the i-th find.*

5 Handling deletions by set rebuilding

To add deletions to the data structure described in the previous section while keeping the same time bounds for union and find, we have to associate two counters with each set. A pointer to these counters is stored at the root of the tree corresponding to each set. The first counter counts the total number of elements in the set and the second counter counts the total number of elements that have been deleted but are still represented by a node in the corresponding tree. We also keep a mark bit with each node in each set. This mark bit is set in every node that corresponds to a deleted item. We perform *find* as before. We also perform *union* as

[4]Note that the root node has no level associated with it and therefore does not belong to one of the sets S^i_f.

before and in addition we set each of the two counters of the resulting sets to store the sum of the values of the corresponding counters in the original sets.

We perform $delete(x)$ as follows. First we mark the node corresponding to x as deleted. Then we perform $find(x)$ to discover the corresponding set S. We increment the counter that counts the number of deleted items in S. Then if the number of deleted items in S at least $\lfloor |S|/2 \rfloor$, we rebuild S. To rebuild S, we pick one of its live nodes to be the root, and set its rank to 1. We make all other elements children of the root. We update the counters to show that the number of deleted items is zero and the total number of elements is $|S| - \lfloor |S|/2 \rfloor$.

The analysis of this data structure is similar to the analysis of the data structure in Section 4. Here however we can no longer assume that the rank of a node never decreases, and that the rank of the parent of a node never decreases, since this happens when we rebuild sets. To overcome this difficulty we think of the elements in the set after rebuilding as new elements. By thinking of the elements this way we at most double the number of elements involved in the sequence. This is because we can associate the elements in the set after rebuilding with the deleted elements in the set before the rebuilding. This way each real element is associated with at most one artificial element resulting from rebuilding. Clearly the total cost of rebuilding is $O(n)$ since we can charge the cost of each rebuilding to the deleted items. Thus each item gets only a constant charge. The dominating factor in the amortized cost of *delete* is the need to find the corresponding set by doing a *find*. In summary, we have obtained the following generalization of Theorem 4.1.

THEOREM 5.1. *: A sequence of m finds mixed with $d \leq n$ delete operations and at most n unions and n make-set operations takes $O(n + \sum_{i=1}^{m} \alpha(m+n, n, \log(n_i)) + \sum_{j=1}^{d} \alpha(m+n, n, \log(d_j)))$ time where n_i is the number of live nodes in the set returned by the i-th find and d_j is the number of live node in the set where we do the j-th delete operation. The size of the data structure at any time during the sequence is proprtional to the number of live items in it.*

6 Summary

We have presented several union-find data structures that support deletions. Some are designed for applications where worst-case performance is important and another has good amortized performance.

In an amortized setting, if one is only interested in keeping the overall size of the data structure proportional to the number of live elements in it, it suffices to use a single global counter. This counter counts the number of live elements in the data structure. Each time an item is deleted we just mark it as such, and decrement the counter of live items. When the number of live elements changes by a constant factor we can rebuild the whole collection of sets. With this global approach we can have individual sets in which the fraction of items marked deleted is large even though the overall number of such items is only a constant fraction of the total number of items. In applications where the space requirement of each set separately matters, this approach may not be good enough. Furthermore, the performance of finds on sets full of deleted items degrades. We have shown that by maintaining a counter per set we can keep the size of each set proportional to the actual size of the set and avoid any degradation in the performance of finds.

The issue is even more subtle to solve in a worst-case setting, where we need to remove deleted elements from sets incrementally while the sets are subject to regular operations. We have described an incremental rebuilding technique to achieve this. Furthermore we have shown directly how to modify the data structure of Smid, which has best possible worst-case performance, to support deletions. The direct approach is more space-efficient.

In all our structures, the time bound for delete is the same as the time bound for find. Intuitively, this is a result of the need to discover the set we are deleting from in order to do rebuilding or rebalancing operations. Whether a faster implementation of delete is possible is an open question.

References

[1] A. M. Ben-Amram and Z. Galil. A generalization of a lower bound technique due to fredman and saks. *Algorithmica*, 30:34–66, 2001.

[2] N. Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *SIAM J. Computing*, 15(4):1021–1024, 1985.

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

[4] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[5] M. Korupolu, R. Mettu, V. Ramachandran, and Y. Zhao. Experimental evaluation of algorithms for incremental graph connectivity and biconnectivity, 1996.

[6] D. Kozen. The design and analysis of algorithms, 1992.

[7] M. Smid. A data structure for the union-find problem having good single-operation complexity. *ALCOM:*

algorithms review, Newsletter of the ESPRIT II Basic Research action program project no. 3075, 1, 1990.

[8] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

Appendix

A Simple Union-Find structure with optimal worst-case performance

In this section we give a simple description of Smid's data structure for union-find [7]. For a fixed parameter k the data structure that we describe supports *union* in $O(k)$ time and *find* in $O(\frac{\log n}{\log k})$ time.

We represent each set by a tree such that the elements of the set reside at the leaves of the tree. Each node of a tree contains a pointer to its parent if it is not the root, and a linked list of its children if it is not a leaf. The root of the tree also contains the name of the set, the number of its children, and the height of the tree. We call every node which is not the root and not a leaf an *internal node*. The height of a node v, denoted by $h(v)$, is the length of the longest path from v to a leaf. Each tree also satisfies the following two invariants.

1. The height of the root is at least 1. A root of height $h = 1$ has at least one child. A root of height $h > 1$ has at least two children. All children of the root are of height $h - 1$.

2. Each internal node of height h has at least k children of height $h - 1$. Internal nodes of height h may have any number of children of height $< h - 1$.

It is easy to see that these properties ensure that the height of a tree T representing a set with n elements is $O(\frac{\log n}{\log k})$. Next we describe how to implement *union* and *find*.

Find(x): We follow parent pointers from the leaf containing x until we get to the root. We return the name of the set stored at the root.

Union(A,B,C): Let a be the root of A and let b be the root of B. Assume without loss of generality that the height of A is no greater than the height of B and that if A and B are of the same height then b has at least as many children as a. There are three cases.

1. The height of A is strictly smaller than the height of B. Let v an arbitrary child of b. If a has less than k children, we make the children of a point to v, and concatenate the list of children of v with the list of children of a. We discard a. If a has at least k children and $h(a) = h(b) - 1$ we make a a child of b. Otherwise, $h(a) < h(b) - 1$, and we make a a child of v. We change the name of the set stored at b to be the name of the new set. We also increment the field that stores the number of children of b in case the root a becomes a child of b.

2. The trees A and B have equal heights, and the number of children of a is smaller than k. We make the children of a point to b instead of a, concatenate the lists of children of a and b and store the resulting list with b. We store in b the new name of the set. We increase the number of children of b by the number of children of a.

3. The trees A and B have equal heights, and the number of children of a is at least k. We create a new root c. We make a and b be children of c by concatenating them into a list, making this list the list of children of c, and setting the parent pointers of a and b to point to c. We store with c the name of the new set, set its children counter to two, and set its height to be one greater than the height of a.

It is easy to see that the properties of the trees stated above are maintained through a sequence of union and finds. therefore *union* takes $O(k)$ time and *find* takes $O(\frac{\log n}{\log k})$ time.