A Simple and Efficient Union-Find-Delete Algorithm

Amir Ben-Amram[*] Simon Yoffe[*]

September 5, 2010

Abstract

The Union-Find data structure for maintaning disjoint sets is one of the best known and widespread data structures, in particular the version with constant-time Union and efficient Find. Recently, the question of how to handle deletions from the structure in an efficient manner has been taken up, first by Kaplan, Shafrir and Tarjan (2002) and subsequently by Alstrup et al. (2005). The latter work shows that it is possible to implement deletions in constant time, without affecting adversely the asymptotic complexity of other operations, even when this complexity is calculated as a function of the current size of the set.

In this note we present a conceptual and technical simplification of the algorithm, which has the same theoretical efficiency, and is probably more attractive in practice.

Keywords: Data structures, Disjoint sets, Union-Find

1 Introduction

A union-find data structure maintains a collection of disjoint sets under the operations:

- MAKESET(a) - generates a singleton set including the new element a.
- UNION(A, B) - takes two sets A, B and unites them, destroying the two original sets.
- FIND(a) - takes an element a and returns an identifier of the set currently containing it (depending on the programming idiom, a reference to the set could be returned).

[*] Tel-Aviv Yaffo Academic College, PO Box 8401, 61083 Jaffa, Israel; email: amirben@mta.ac.il (corresponding author), simon.yoffe@gmail.com

The classic data structure for the Union-Find problem represents each set A as a rooted tree T_A which contains only parent links, and each element in A as a node of T_A. The UNION operation links the root of the shallower tree to the root of the taller tree (so the height will be increased only in a case of trees of equal height) in $O(1)$ time. The FIND operation climbs from the provided element up to the root and returns the root node as an identifier of the set. For the sake of efficiency, it also performs *path compression*, which is to link all the nodes in the path directly to the root (reducing the cost of subsequent FINDs). The FIND operation costs $O(\log(n))$ worst-case time and $O(\alpha(n))$ amortized time where n is the number of nodes in the set returned by the FIND operation and $\alpha(n)$ is the inverse-Ackermann function [4, 1].

Terminology: the *height* of a node $v \in T$, denoted by $h(v)$, is defined to be 0 if v is a *leaf*, and $\max\{h(w) | w \text{ is child of } v\} + 1$, otherwise. For a tree T let $root(T)$ denote the *root* of T and for a set A, let $root(A)$ denote the root of T_A. The height of a tree is the height of its root. For any node $v \in T$, let $p(v)$ denote the parent of node v; if v is a root, $p(v) = v$. Each node $v \in T$ is assigned an integer *rank*, denoted by $rank(v)$. The rank of a tree is defined to be the rank of the root of the tree and the rank of a set is defined to be the rank of the tree representing the set. By default, log will mean $\log_2$. The classic algorithm has the following invariant:

Invariant 1. *The parent of a node always has a strictly higher rank than the node itself.*

The UNION operation is actually done by rank (rather than height; this is not the same because of path compression). Since ranks are strictly increasing when following parent pointers, the time of a FIND operation applied to an element in a set A is clearly proportional to the tree height, bounded by $rank(A)$. The fact that $rank(A) \leq \log |A|$ implies the $O(\log(n))$ worst-case time of the FIND operation, and together with the path compression during FIND implies the $O(\alpha(n))$ amortized time [4, 1].

Our goal is to implement a DELETE operation in $O(1)$ time and keep an $O(N)$ space complexity for the whole data structure, where N is the number of elements currently maintained by the data structure. Note that the bounds on the cost of FIND are in terms of the size of the set returned, while a DELETE operation decreases the number of elements in the set; thus, maintaining these time bounds means that a DELETE operation should make subsequent FIND operations faster while decreasing the size of the data structure.

The inclusion of the delete operation in the Union-Find structure has been studied by Kaplan, Shafrir and Tarjan [2] and later by Alstrup et al. [1] who improved the efficiency of DELETE operation to constant time while maintaining worst case, amortized time and space bounds as above.

In [2] the motivation for including a DELETE operation in the data structure was implementing meldable priority queues. Another application is implementing an operation which moves an element to a new set. This has been used in work on ownership transfer in object-oriented systems [3].

We present a Union-Find-Delete algorithm which we derived by a simplification of the solution of Alstrup et al. [1]. We removed the use of *vacant* nodes and the *tidying up* operations. We replaced the complicated *local compress* procedure with a much simpler one; not only is this advantageous in practical terms, it also made the analysis much simpler. Even the asymptotic bounds were slightly improved. While our work relies on [1], the reader need not be familiar with their algorithm for following this note.

As by-product, we show how to support an operation that given a node, returns an arbitrary leaf in the same tree, in $O(1)$ worst-case time.

2 The algorithm

To support a Delete operation in $O(1)$ time, we extend the data structure as follows.

1. For each node which is not a leaf we maintain a doubly-linked list Clist of its children. We consider the children to be ordered, according to this list, from right to left.

2. For the root we maintain a doubly-linked list NLlist of the children that are not leaves.

3. For each tree we maintain a cyclic doubly-linked list DFSlist of the tree nodes in DFS order starting from the root and proceeding from the rightmost child towards the left.

4. The data structure will be extrinsic. This means that each element is distinguished from the tree node with which it is associated. Each element object is doubly-linked to a tree node (in previous work, there was no distinction between a tree node and an element. This change helps us to avoid vacant nodes). Objects of type *node* hold *next/prev* pointers of all the associated lists. So by providing a pointer to a node we can easily perform local operations on the lists.

Definition 2.1. A tree is said to be *reduced* if it is either

- A tree composed of a single node of rank 0, or
- A tree of height 1 with a root of rank 1 and leaves of rank 0.

Definition 2.2. A tree is said to be *full* if each of its nodes is either

- A leaf of rank 0, or
- A parent with at least three children. The parent's rank is strictly higher than all of the ranks of its children.

The fact that each non-leaf node has at least three children will be used in proving the bound on the rank of a root. For implementing the Delete operation in $O(1)$ worst-case time, without preserving the bounds on the other operations, having two children would suffice. We introduce an additional invariant:

Invariant 2. *A tree is either full, or reduced.*

Consequently, all trees of $size \geq 4$ are full, while trees of $size < 4$ have to be reduced.

We next show the implementation of the Makeset, Union, Find and Delete operations over the extended data structure. Recall that our goal is to maintain the following efficiency: Makeset and Union operations in $O(1)$ time, Find in $O(\log(n))$ worst-case time and $O(\alpha(n))$ amortized time where n is the number of elements in the returned set, and Delete in $O(1)$ time.

2.1 Implementation of the Makeset operation

Makeset(a):

- Create a node x for the element a. Set $p(x) \leftarrow x$, $rank(x) \leftarrow 0$.
- Create empty NLlist and Clist.
- Create the DFSlist containing x.

Clearly, this can be done in $O(1)$ worst-case time and the invariants hold.

2.2 Implementation of the Union operation

Union(A, B): Recall that A and B are sets. Let T_A, T_B be the trees of A, B respectively. If one of the trees is of size < 4 (without loss of generality, T_B), do as follows:

- For every node x in T_B : set $p(x) \leftarrow root(T_A)$, $rank(x) \leftarrow 0$.
- Set $rank(root(T_A)) \leftarrow \max\{rank(root(T_A)), 1\}$.
- Update Clist and DFSlist (straightforward).

It remains to handle trees of $size \geq 4$, which are full by Invariant 2. Assume, without loss of generality, that $rank(A) \geq rank(B)$, Then:

- Set $p(root(B)) \leftarrow root(A)$, and if $rank(A) = rank(B)$, increase $rank(A)$ by one.
- Insert $root(B)$ into Clist and NLlist of $root(A)$ (straightforward: we add it at the beginning of the lists).

- Merge the DFSLISTs of T_A and T_B (straightforward: we insert the DFSLIST of T_B into the DFSLIST of T_A immediately after $root(A)$).
- Free the NLLIST of $root(T_B)$. We assume a garbage collection or a free-list mechanism to which we can move the list in $O(1)$ time.

Clearly can be done in $O(1)$ worst-case time and the invariants hold.

2.3 Implementation of the Find operation

The classic Find algorithm [4] follows parent pointers from x all the way to the root while performing path compression: each node in the path is linked directly to the root.

Tarjan and van Leeuwen [5] presented alternatives to path compression which achieve the same asymptotic time complexity. The simplest alternative is the *path splitting* process: each node y in the path such that $p(p(y)) \neq p(y)$ is moved from its parent to its grand-parent. We refer to this modification as *relinking* y.

We will use path splitting in our implementation of FIND. When a node is disconnected it may affect the full tree property; to fix that, function $relink(y)$ checks whether the original $p(y)$ is left with only two offsprings. In that case, it relinks them as well.

We next implement the relink operation. Note that moving a node x actually moves the entire subtree rooted at x.

1. Remove x from the CLIST of $p(x)$ and insert it in the CLIST of $p(p(x))$ in one of two locations: if x has a left sibling, it is inserted before (to the right of) $p(x)$; otherwise, after (to the left of) $p(x)$.
2. Update the DFSLIST of the tree. This means extracting the DFSLIST of the subtree that starts at node x and inserting it at the new location, as shown later.
3. If $p(x)$ becomes a leaf, set $rank(p(x)) \leftarrow 0$ and if $p(p(x))$ is the root, remove $p(x)$ from the NLLIST of the root.
4. If NLLIST became empty (this means that the tree is reduced), set $rank(root) \leftarrow 1$.

It remains to show how to update the DFSLIST when disconnecting node x:

If x has a left sibling (l), it means that the subtree rooted at x is represented in the DFSLIST by the segment $[x, l)$. It is simple to disconnect the segment and insert it after $p(x)$ in $O(1)$ time.

If x is the leftmost child of $p(x)$, we do not have to change the DFSLIST.

Clearly this relink operation can be done in $O(1)$ worst-case time. Let us summarize the FIND operation.

FIND(a):

Let x be the tree node associated with the element a.

- While $p(x)$ is not the root, let $t \leftarrow p(x)$, relink x to $p(t)$, set $x \leftarrow t$.
- Return $p(x)$.

In principle, we could also use the familiar path compression technique. However it requires a more complex algorithm for fixing the DFSlist and maintaining the invariants, while path splitting is simple and uses the relink primitive, also used in the implementation of Delete.

2.4 Implementation of the Delete operation

First, we show how to delete a node in a tree of a *size* ≤ 4, this means that after the delete the tree should become reduced. To accomplish that we will rebuild the tree to a reduced tree, which can be achieved in $O(1)$ time due to the size of the tree.

Next, we consider larger trees. If the tree is reduced, the delete is simple.
Reduced-Tree-Delete(a):

If the element a is associated with a leaf node, simply delete the node. Otherwise the element a is associated with the root. Let l be a leaf node. Switch the associations of l and the root to set elements, so that a becomes associated with the leaf. Finally delete the leaf.

Clearly, the above can be achieved in $O(1)$ time and the tree remains reduced.

It remains to show how to delete a node in a tree of a *size* > 4 which is not reduced, this means that the tree is full (Invariant 2) and after the Delete operation it should remain full. The idea is to find a leaf node in $O(1)$ time.
Find-Leaf(a):

Let x be the tree node associated with the element a. If x is a leaf, we are done, otherwise: If x is the root, we have direct access to the last member of DFSlist, which is a leaf. If x is not the root, either:

- The node x has a left sibling (l); then the node DFSlist$[l]$.*prev* is the leftmost leaf in the subtree of x, or
- The node x has a right sibling r; then the node DFSlist$[x]$.*prev* is the leftmost leaf in the subtree of r.

Now, our problem reduces to deletion of a leaf node from a full and non-reduced tree.
Delete-Leaf(x):

- Update the Clist of $p(x)$, as well as DFSlist, to bypass x.
- Delete the node x.
- If the tree is not reduced, apply Local-Rebuild (below) to $y = p(x)$.

The purpose of local rebuilding is to ensure that tree remains balanced after the DELETE operation, as necessary to ensure the efficiency of FIND.
LOCAL-REBUILD(y):

- If y is not the root, relink the two leftmost children of y.
- If y is the root, let c be a non-leaf child of y. Relink the three leftmost children of c. Note that a non-leaf node can be found via the NLLIST of the root, and does exist because the tree is not reduced.

Note that the relink operation preserves the full tree property. All of the above takes $O(1)$ worst-case time and the invariants are maintained. Note that ranks are unaffected by LOCAL-REBUILD in general, however if the tree becomes reduced, the root's rank is updated; and a sufficiently long sequence of DELETE*s* must eventually arrive at a reduced tree.

We can now put together the DELETE operation.
DELETE(a):
Let x be the tree node associated with the element a.

- If the tree that contains x is of $size \leq 4$, rebuild as a reduced tree, else
- If the tree is reduced, call REDUCED-TREE-DELETE(a), else
- Let l = FIND-LEAF(a), switch elements between l and x and call DELETE-LEAF(l).

3 Analysis

The goal of this section is to analyze the worst-case cost of FIND in the presence of deletions. The analysis is a much-simplified version of that from [1].

Definition 3.1. The value $val(v)$ of a node v is defined as

$$val(v) = (\frac{3}{2})^{rank(p(v))}$$

The value of a set A is defined as the sum of the values of all nodes in T_A:

$$VAL(A) = \sum_{v \in T_A} val(v)$$

We will show that the MAKESET, UNION, FIND and DELETE operations over our data structure preserve the following invariant:

Invariant 3. $VAL(A) \geq 2^{rank(A)}$.

Since the tree representing a set A contains exactly $|A|$ nodes, each of value at most $(\frac{3}{2})^{rank(A)}$ it will follow that

$$\begin{aligned} |A|(\frac{3}{2})^{rank(A)} &\geq 2^{rank(A)} \\ |A| &\geq \frac{2^{rank(A)}}{(\frac{3}{2})^{rank(A)}} = (\frac{4}{3})^{rank(A)} \\ rank(A) &\leq \log_{\frac{4}{3}}(|A|) = O(\log|A|) \end{aligned}$$

As the rank of a tree is always an upper bound on its height, the worst-case time of the FIND operation is $O(\log|A|)$.

Lemma 3.2. *If the tree representing a set A is reduced,* $VAL(A) \geq 2^{rank(A)}$.

Proof. If T_A is of height 0 it implies $VAL(A) = (\frac{3}{2})^0 = 1$ and $2^{rank(A)} = 1$. If T_A is of height 1 it implies $VAL(A) \geq (\frac{3}{2})^1 + (\frac{3}{2})^1 = 3$ while $2^{rank(A)} = 2$. □

3.1 Makeset

A MAKESET operation creates a reduced tree, so according to Lemma 3.2, Invariant 3 is preserved.

3.2 Union

A UNION operation creates a new set $C = union(A, B)$. There are two cases. If one of the trees is initially of higher rank (say T_A), we link T_B to its root and ranks do not change: we have

$$VAL(C) > VAL(A) \geq 2^{rank(A)} = 2^{rank(C)}$$

where the second inequality uses the invariants assumed for A and B. If the trees are of equal rank, the root's rank is increased. We have

$$VAL(C) > VAL(A) + VAL(B) \geq 2^{rank(A)} + 2^{rank(B)} = 2^{1+rank(A)} = 2^{rank(C)}.$$

In both cases, Invariant 3 is preserved.

3.3 Find

A FIND operation changes the tree T_A. If the tree becomes reduced after the FIND operation then according to Lemma 3.2, Invariant 3 is preserved. This leaves the case where the tree after the FIND operation is not reduced.

Let A' denote the set representation after the FIND operation. We assume that $VAL(A) \geq 2^{rank(A)}$ and we have to prove that $VAL(A') \geq 2^{rank(A')}$. The FIND operation changes the tree by relinking nodes. We will show that Invariant 3 is preserved after each relink operation.

The relink operation disconnects node x and connects it to $p(p(x))$. Let $y = p(x)$, $g = p(y)$ and $k = rank(g)$ then $rank(y)$ is at most $k - 1$. When x is disconnected we loose at most $(\frac{3}{2})^{k-1}$ and when x is connected to g we gain $(\frac{3}{2})^k$ so $VAL(A)$ only increases, while $rank(A)$ does not change (Find changes the rank only if the tree becomes reduced). This means that Invariant 3 is preserved.

3.4 Delete operation

A Delete operation changes the tree T_A. If the tree becomes reduced after the Delete operation then according to Lemma 3.2, Invariant 3 is preserved, so we are left with the case when the tree after the Delete operation $T_{A'}$ is not reduced.

We assume that $VAL(A) \geq 2^{rank(A)}$ and wish to prove that $VAL(A') \geq 2^{rank(A')}$. The Delete operation changes the tree by the Delete-Leaf operation on node x (which includes Local-Rebuild).

If $y = p(x)$ is not the root, let $g = p(y)$ and $k = rank(g)$ then $rank(y) \leq k - 1$. The Delete-Leaf operation will delete x and relink two children of y to g. By deleting and unlinking we lose at most $3(\frac{3}{2})^{k-1}$ and by linking again we gain $2(\frac{3}{2})^k$; the total change is at least

$$-3(\frac{3}{2})^{k-1} + 2(\frac{3}{2})^k = 0$$

If $y = p(x)$ is the root, let $k = rank(y)$; then $rank(x) \leq k-1$. Let c be a non-leaf child of y. Then $rank(c) \leq k - 1$ and the Delete-Leaf operation will delete x and relink three children of c to y. By deleting and unlinking we lose at most $(\frac{3}{2})^k + 3(\frac{3}{2})^{k-1}$ and by linking again we gain $3(\frac{3}{2})^k$; the total change is at least

$$-(\frac{3}{2})^k - 3(\frac{3}{2})^{k-1} + 3(\frac{3}{2})^k = 0$$

Note that Delete-Leaf may relink more nodes for preserving the full property of the tree, which only increases the value. We have shown that $VAL(A)$ is not decreasing and $rank(A)$ does not change (Delete changes the rank only if the tree becomes reduced). Hence, Invariant 3 is preserved.

Lemma 3.3. *The data structure satisfies Invariant 3, namely $VAL(A) \geq 2^{rank(A)}$, before and after every operation.*

Proof. Induction over the number of operations: Lemma 3.2 is the base case and the above analysis of the operations constitutes the inductive step. □

Corollary 3.4. *The height of trees, hence the worst-case time of* Find, *is $O(\log |A|)$.*

Corollary 3.5. Find *takes $O(\alpha(n))$ amortized time.*

This follows directly from [1]: in essence, they show that Invariant 3 suffices for deriving the inverse-Ackermann bound for Find with path-splitting.

4 Conclusion

Our Union-Find-Delete data structure has asymptotic worst-case and amortized complexity which is similar to those obtained by Alstrup et al. (though constant factors appear to be smaller). The notable difference is a conceptual simplification (no vacant nodes) and the significant simplification of "local compression".

We would like to point out the following questions for further research:
1. Are there other applications to finding a leaf fast in the type of dynamic tree considered here?
2. How can the memory usage be reduced?

References

[1] S. Alstrup, I. L. Gørtz, T. Rauhe, M. Thorup, and U. Zwick. Union-find with constant time deletions. In *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP 2005).*, volume 3580 of *Lecture Notes in Computer Science*, pages 78–89. Springer-Verlag, July 2005.

[2] Haim Kaplan, Nira Shafrir, and Robert Endre Tarjan. Union-find with deletions. In *SODA*, pages 19–28, 2002.

[3] Yoshimi Takano. Implementing uniqueness and ownership transfer in the universe type system. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

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

[5] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.