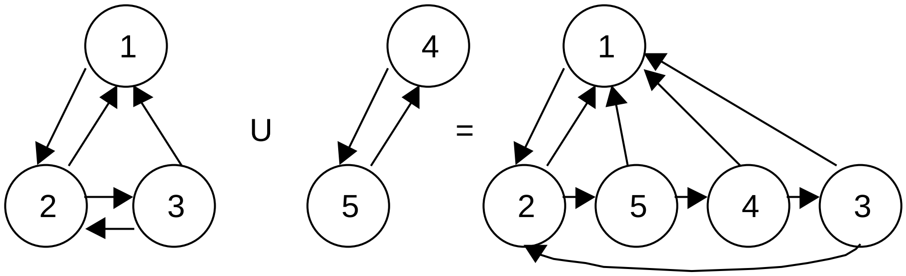
Cenk Gazen, bcg+@cs

1. Union Find using Trees of Bounded Depth

1. ***DESCRIPTION:*** In addition to the parent pointers, we also maintain a next-sibling pointer and a pointer to the 'last' child if the node is a parent node. The children nodes are kept in a singly linked circular list. Here's an example set of trees. (The extra pointers are similar to those maintained for a binomial heap.)



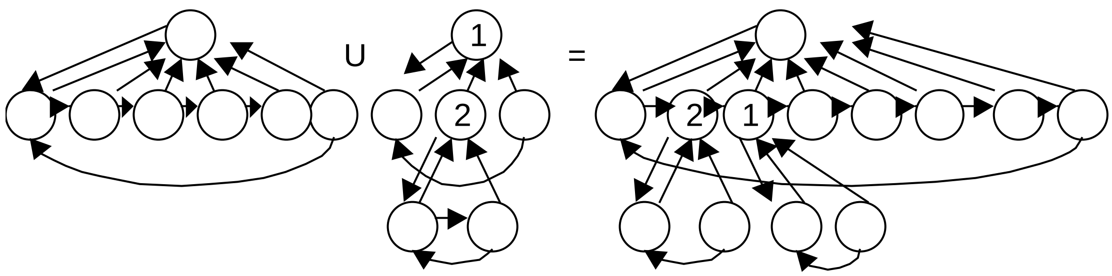
Then **Union** can be implemented as:

a. Without loss of generality, let *u* be the smaller-sized tree.
b. For each node in *u* (we can traverse it using the circularly linked list), update the parent pointer to point to *v*.
c. Insert *u* into the circular list of *v*'s children.
d. Insert *u*'s children into the circular list of *v*'s children.

CORRECTNESS: After **Union**(u, v), *u* and all its children have parent pointers pointing to *v*. Since all parent pointers point to a root-node, the new children of node *v* are at depth at most 1 and the old children remain at their old depths. This is the only way to create trees, so the depth of any tree is bounded by 1. Also, *v*'s (new) children are correctly kept in a singly-linked circular list.

TIMING: We use amortized analysis for timing. Each time the parent pointer of a node is changed, we charge the cost to the node. Let tree_size(x) denote the size of the tree that node x is in. Now whenever the parent-pointer of any node x is modified, tree_size(x) is at least doubled. This is because we only modify the parent-pointers for the nodes in the smaller tree. So we know tree_size(x) = size(u) <= size(v) and after the union operation tree_size(x) = size($u \cup v$) >= 2*size(u). This means after we charge i units of credit on some node x, tree_size(x) >= 2^i. But tree_size(x) is bounded by n, so the total charge on any node must be less than log n. We have a total of n nodes, therefore, the total cost of all the unions is $O(n \log n)$. The other steps of the algorithm (a, c, d) run in $O(1)$ time. The only tricky operation is in step d, where two circular lists are joined together, but this is $O(1)$ time, too, since we can store a pointer to the next-to-last element of *u*'s children while traversing that list in step b and use it to do step d in $O(1)$.

2. ***DESCRIPTION:*** We maintain the same node structure as in the first part, but we also keep track of the size of the tree at the roots. Also, we need to be able to traverse the non-leaf children of a tree in linear time. This can be done by maintaining them in a separate list or by simply keeping at the head of the list of children.



The new **Union-2** algorithm is:

a. Without loss of generality, let *u* be the smaller-sized tree.
b. If size(*u*) <= log *n*, apply **Union**.
c. Else
   c1. For each non-leaf node in *u* (we can traverse the nodes using the circularly linked list), update the parent pointer to point to *v*.
   c2. Insert *u* into the circular list of *v*'s children.
   c3. Insert *u*'s non-leaf children into the circular list of *v*'s children.

CORRECTNESS: If size(*u*) <= log *n*, then we know from part 1 that all nodes in *u* are inserted to *v* at depth 1. If size(*u*) > log *n*, then all non-leaf nodes in *u* end up at depth 1 of tree *v* (step c), which by definition means that the leaf nodes of *u* are at depth 2. The old children of *v* remain at their current depths. Since this is the only way to create a tree, the depth is bounded by 2. Also, the nodes of *u* are correctly inserted into the linked lists of *v*.

TIMING: The analysis is similar to part 1. Suppose we charge one time unit to a node when its parent pointer changes. First, let's calculate the charge on any node *x* that ends up as a leaf node in the final tree. The charge on such *x* can only come from the **Union** (and not from **Union-2**). This is because if a node ends up as a leaf, it must have been a leaf node before step c1 and **Union-2** charges only non-leaf nodes. To find the charge from **Union**, we again observe that tree_size(*x*) >= 2^i (where *i* is the number of charges we put on node *x*)and that we apply **Union** only when tree_size(*x*) <= log *n*. So the charge on any leaf node *x* is at most log log *n* and the total charge O(*n* log log *n*).

The only nodes we need to consider are non-leaf nodes at depth 1. The charge on such a node can only come from **Union-2**. This is because a non-leaf node has to be a root-node during **Union** operations (a leaf-node never becomes non-leaf), so the charge on it from **Union** is 0. We can calculate the charges by **Union-2** by observing that after *i* charges on node *x*, tree_size(*x*) >= 2^i log *n* because the initial size of a tree is at least log *n* and at least doubles every time the node is charged. So *i* <= log (*n*/log *n*). By definition of **Union-2**, it only operates on trees of size at least log *n*, so the number of such trees (thus roots) is at most (*n* / log *n*). Multiplying the two expressions we get
$n/\log n * (\log (n/\log n)) = n/\log n * (\log n - \log\log n) = n - \log\log n = O(n)$

Finally, the total charge is $O(n \log\log n)+O(n) = O(n \log\log n)$.