CSC 316 – Dynamic Union-Find Structures.
Sample undergraduate in-course projects – 13 March 2016

***1.2 Generic Data Structures*:**
(A) General Description
Here we briefly describe some generic data structures that could be built, and possibly used for other purposes:

- A Simple General Lattice Structure.
- Static set partitions (where we only add, but not remove values and partitions, and merge partitions) from Static Union Find problem.
- Build over the static Union Find problem to obtain dynamic set partitions (where addition and deletion of values and merging of partitions are done) for dynamic union find problems.
- Construct Range Data Structures: A data structure that circumscribes sets of adjacent nodes in a graph that share a common property.

(1) Dynamic Union-Find Problem
We COPY Brass (2008), building on the static version.
The Union-find problem seeks to keep track of a partition of a set, in which partition classes may be merged, and to answer queries whether two elements are in the same class" (Brass, 2008).

(B) Union-Find: Merging Classes of a Partition
"There is a set of items on which some partition is maintained. Items can be inserted into that set, each initially forming a one-element partition class. Items are identified by a pointer, a finger into the structure, which is obtained from the insertion operation." … "The underlying partition can be changed by joining two classes, the classes identified by giving items in these classes. And the partition can be queried by asking whether two items are in the same class. So we have the following operations:

- *insert*: Takes an item, returns pointer to the node representing the item, and creates a one-element class for it.
- *join*: Takes two pointers to nodes and joins the classes containing these items.
- *same class*: Takes two pointers to nodes and decides whether their items are in the same class.

We represent each class by a directed tree, with all edges oriented to the root. Then each node representing an item needs just one outgoing pointer to that neighbor in the tree that is nearer to the root; for the root itself we use the NULL pointer.

Given this representation, we can query whether two items are in the same class by following from both nodes the path to their respective roots; they are in the same class if they reach the same root. And we can join two classes by connecting the root of one tree to the root of the other tree.

This outline still leaves a lot of freedom: we have to decide on joining two trees, which of the two roots should become the root of the union. And we can restructure the tree, ideally making all vertices point directly to the root, because the time taken by the query is the length of the path to the root. In the best-known solution, we use the following two techniques:

- *Union by rank*: Each node has another field, the rank, which starts on insertion as 0. Each time we join two classes, the root with the larger rank becomes the new root, and if both roots have the same rank, we increase the rank in one of them.
- *Path compression*: In each query and each update, when we followed a path to the root, we go along that path a second time and make all the nodes point directly to the root.

Union by rank implementation is given below for: Data structure, and the insert, same_class, and join operations.

Node implementation: insert, same_class, join

```
typedef struct uf_n_t {
      int rank;
      item_t *item;
      struct uf_n_t *up;
} uf_node_t;

uf_node_t *insert(item_t *new_item){
uf_node_t *new_node;
      new_node = get_node();
      new_node->item = new_item;
      new_node->rank = 0;
      new_node->up = NULL;
return( new_node );
}

int same_class( uf_node_t *node1,
uf_node_t *node2 )
{ uf_node_t *root1, *root2, *tmp;
/* find both roots */
for( root1 = node1; root1->up != NULL;
root1 = root1->up)
; /* follow path to root for node1 */
for( root2 = node2; root2->up != NULL;
root2 = root2->up)
; /* follow path to root for node2 */
/* make both paths point directly to
their respective roots */
tmp = node1->up;
while( tmp != root1 && tmp != NULL )
{ node1->up = root1;
node1 = tmp; tmp = node1->up;
}
tmp = node2->up;
while( tmp != root2 && tmp != NULL )
{ node2->up = root2;
node2 = tmp; tmp = node2->up;
}
/* return result */
return( root1 == root2 );
}
```

```
void join( uf_node_t *node1, uf_node_t
*node2 )
{ uf_node_t *root1, *root2, *new_root,
*tmp;

      /* find both roots */
      for( root1 = node1; root1->up !=
NULL;
      root1 = root1->up)
      ; /* follow path to root for node1 */
      for( root2 = node2; root2->up !=
NULL;
      root2 = root2->up)
; /* follow path to root for node2 */

/* perform union by rank */
if( root1->rank > root2->rank )
{ new_root = root1; root2->up = new_root;
}
else if( root1->rank < root2->rank )
{ new_root = root2; root1->up = new_root;
} else /* same rank */
{ new_root = root1; root2->up = new_root;
new_root->rank += 1;
}

/* make both paths point directly to
the new root */
tmp = node1->up;
while( tmp != new_root && tmp != NULL )
{ node1->up = new_root;
node1 = tmp; tmp = node1->up;
}
tmp = node2->up;
      while( tmp != new_root && tmp != NULL
)
      { node2->up = new_root;
      node2 = tmp; tmp = node2->up;
      }
}
```

To give the structure in more detail now, each node has two pointers:

- *up*, which is NULL for a root, points to the next node on the path to the root for all other nodes.
- *list*, which points to its list of lower neighbors for a root, points to the next on that list for a node that is lower neighbor of the root and is unspecified otherwise.

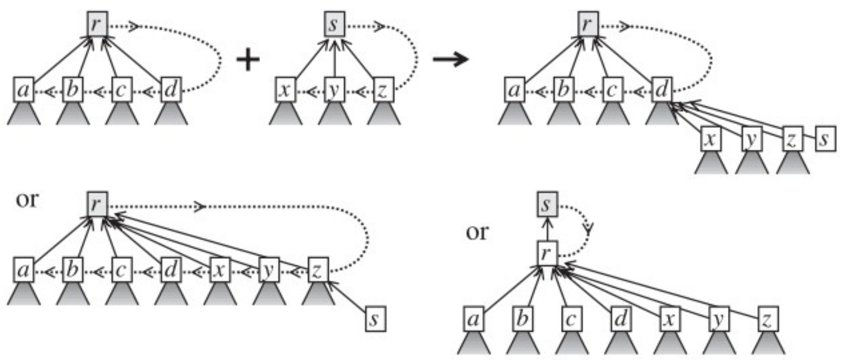
The node also contains two numbers: the height and the indegree. Then
the rules for joining two components with roots r and s are as follows:
Let r->height ≥ s->height ≥ 2, then:

If r->height > s->height, all lower neighbors of s, as well as s itself, are made to point to a lower neighbor of r.

Else r->height = s->height. All lower neighbors of s are added to the list of lower

neighbors of r,
    If r->height > r->indegree, s is made to point to one lower neighbor of r.
    Else s becomes the new root, with r as its only lower neighbor.

With these definitions, we can now give the code for the operations of the [union-find] structure [in the box below.



Three Cases for Joining the Classes with Roots r and s

Union-find data structures

```
typedef struct uf_n_t {
      int height;
      int indegree;
      item_t *item;
      struct uf_n_t *up;
      struct uf_n_t *list;
 } uf_node_t;

uf_node_t *uf_insert(item_t *new_item)
{ uf_node_t *new_node;
new_node = get_uf_node();
new_node->item = new_item;
new_node->height = 0;
new_node->indegree = 0;
new_node->up = NULL;
new_node->list = NULL;
return( new_node );
}

int same_class( uf_node_t *node1,
uf_node_t *node2 )
{ uf_node_t *tmp1, *tmp2;
/* find both roots */
for( tmp1 = node1; tmp1->up != NULL;
tmp1 = tmp1->up)
; /* follow path to root for node1 */
for( tmp2 = node2; tmp2->up != NULL;
tmp2 = tmp2->up)
; /* follow path to root for node2 */
/* return result */
return( tmp1 == tmp2 );
}

void join( uf_node_t *node1, uf_node_t
*node2 )
{ uf_node_t *root1, *root2, *tmp;
int i;
/* find both roots */
for( root1 = node1; root1->up != NULL;
root1 = root1->up)
; /* follow path to root for node1 */
for( root2 = node2; root2->up != NULL;
root2 = root2->up)
; /* follow path to root for node2 */
if( root1->height < root2->height )
{ tmp = root1; root1 = root2; root2 = tmp;
} /* now root1 is the larger subtree */
if( root1->height >=2 )
{ /* inserting two levels below root 1,
height stays the same */
if( root2->height < root1->height )
{ tmp = root2->list;
/* go through list below root2 */
while( tmp != NULL )
{ tmp->up = root1->list;
/* point to node on root1 list */
tmp = tmp->list;
}
root2->up = root1->list;

/* also point root2 to that node */
}
else /* root2->height == root1->height */
{ /* join root2 list to root1 list,
pointing to root1 */
tmp = root2->list; tmp->up = root1;
while( tmp->list != NULL )
{ tmp = tmp->list;
/* move to end of root2 list */
tmp->up = root1;
}
tmp->list = root1->list;
root1->list = root2->list;
/* linked lists */
root1->indegree += root2->indegree;
/* now lists joined together
below root 1 */
if( root1->indegree <= root1->height )
root2->up =root1->list;
/* point to node on root1 list */
else /* root2 becomes new root,
root1 goes below */
{ root1->up = root2;
root1->list = NULL;
root2->height += 1;
root2->indegree = 1;
root2->list = root1;
}
}
}
else /* root1->height <= 1*/
{ if( root1->height == 0 )
{ root1->height = 1;
root1->indegree = 1;
root1->list = root2;
root2->up = root1;
/* root1 is new root */
}
else /* root1->height == 1 */
/* any root at height 1 has exactly
one lower neighbor */
{ if( root2->height == 1 )
/* both height 1 */
root2->list->up = root1;
/* now make root1 lower neighbor
of root2 */
root2->height = 2;
root2->indegree = 1;
root2->list = root1;
root1->list = NULL;
root1->up = root2;
/* now root2 is the new root */
}
}
}
```

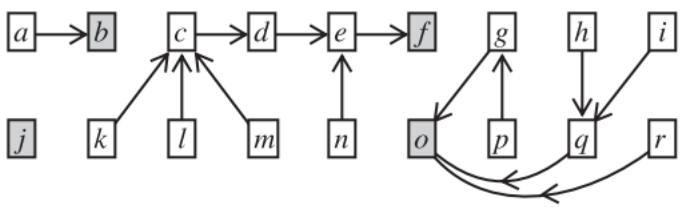
In this structure, each node at height h has indegree at least h once it becomes a nonroot node and has indegree at most h while it is the root. All the lower neighbors of a root, which has height h, themselves have height h − 1, although later further subtrees get attached that might have smaller height. So each nonroot node that is at height h has at any time at least h lower neighbors that are at height h − 1, in addition to some possible lower neighbors at smaller height. This implies that a tree of height h in this structure contains at least (h − 1)! nodes; so with (h − 1)! ≤ n

(C) Union-Find with Copies and Dynamic Segment Trees

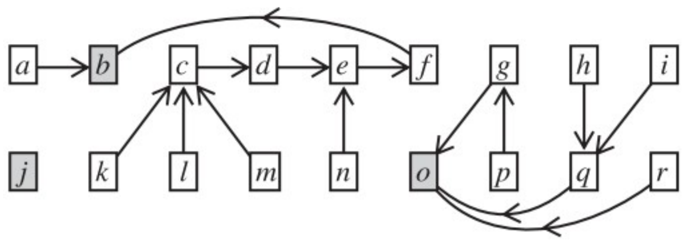
A structure that kept track of general set systems would be very useful. Up to now, our model is very restricted, the sets have to be disjoint, and we can take only unions of them. So we keep track of a sequence of coarser and coarser partitions until after n − 1 unions everything is in the same class. Another less obvious, but equally important, restriction is that our elements are presented by fingers, not by keys. There is no search-tree variant that supports the union of two sets. Of course, we can use a search tree to keep track of the fingers and then we get an O(log n) overhead on every operation, so the trivial O(log n) bound for the union-find structure would be sufficient.

The union-copy structure by van Kreveld and Overmars keeps track of a set of items, represented by fingers, and sets, also represented by fingers. It supports the following operations, which are symmetric with respect to the role of items and sets:

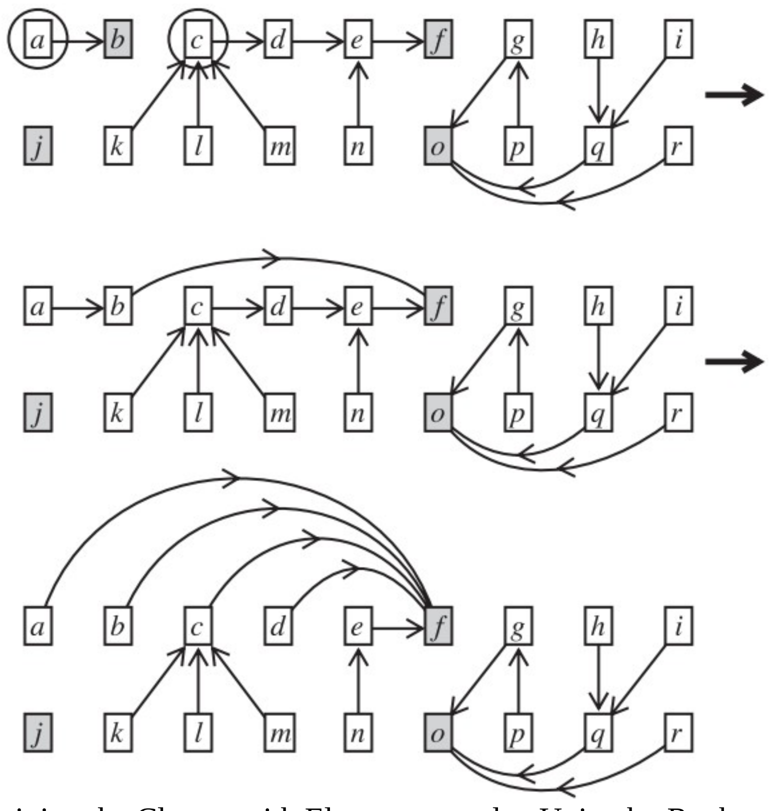
- *create item*: Creates representation for a new item and returns a finger to it.
- *create set*: Creates representation for a new set and returns a finger to it.
- *insert*: Inserts a given item in a given set. Requires that the item was not already contained in the set.



Classes { a, b } , { c, d, e, f, k, l, m } , { g, h, i, o, p, q, r } , { j }
with Marked Root Nodes.



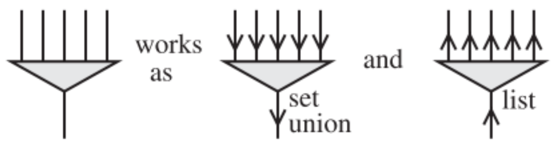
Joining Two Classes by Connecting One Root to the Other



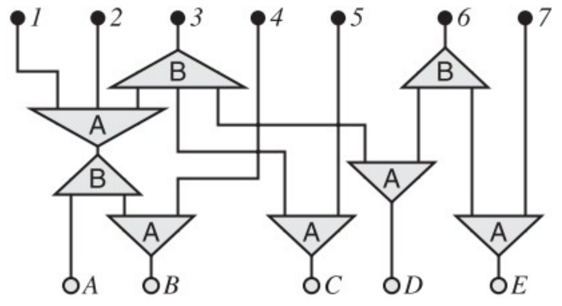
Joining the Classes with Elements a and c: Union by Rank Followed by Path Compression

- *list sets*: Lists all sets containing a given item.
- *list items*: Lists all items contained in a given set.
- *join sets*: Replaces the first set by the union of two given sets and destroys the other set. Requires the two sets to be disjoint.
- *join items*: Replaces the first item by an item that is contained in all sets which contained one of the two given items, and destroys the other item. Requires that there is no set that contains both items.
- *copy set*: Creates representation for a new set, which is a copy of the given set, and returns a finger to it.
- *copy item*: Creates representation for a new item, which is a copy of the given item, and returns a finger to it.
- *destroy set*: Destroys the given set.
- *destroy item*: Destroys the given item.

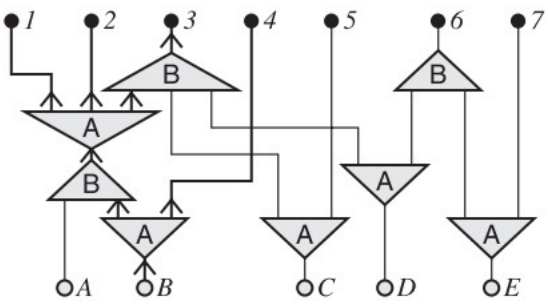
Of these operations, the creation and insertion operations are O(1), and the complexity of the others depends on the complexity of the underlying union-find structure, which is used as a building block of the union-copy structure. … The underlying union-find structure must also perform – in addition to the normal operation of returning the current name (root) of the set containing a given element – the reverse operation, listing all the elements of a set with a given root. This is easy to add because we perform only disjoint unions: we must attach to the root a list of pointers to the elements. These lists are just put together in a union operation; to avoid pointers to beginning and end, we can just use a cyclic linked list.



Building Block: Extended Union-Find Structure: One Direction: Finds Union, Other Direction: Lists Elements



Structure Representing the Set System: A = { 1, 2, 3 } , B = { 1, 2, 3, 4 } , C = { 3, 5 } , D = { 3, 6 } , E = { 6, 7 }



Listing the Items in Set B

The underlying representation of the set system is as follows: The data structure consists of item nodes, set nodes, and sets in two extended union-find structures – labeled A and B – which allow both normal and listing queries. It is symmetric, like the operations supported by it, but because pointers are necessarily directed graph edges and the two union-find structures exchange their roles, we describe both directions.

If we wish to go from the items to the sets, the structure is as follows:

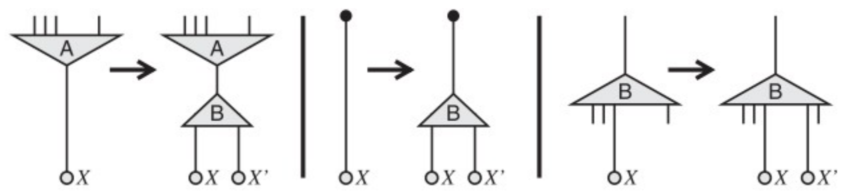
1. Each item node has exactly one outgoing edge.
2. Each set in the union-find structure A has at least two incoming edges (the elements of the set) and exactly one outgoing edge (the current name of the set).
3. Each set in the union-find structure B has exactly one incoming edge (the current name of the set) and at least two outgoing edges (the elements of the set).
4. Each set node has exactly one incoming

edge.
5. An item belongs to a set if there is a directed path from the item node to the set node.
6. Between any item node and any set node there is at most one directed path.
7. There are no edges between sets in the same structure (from A to A or from B to B).

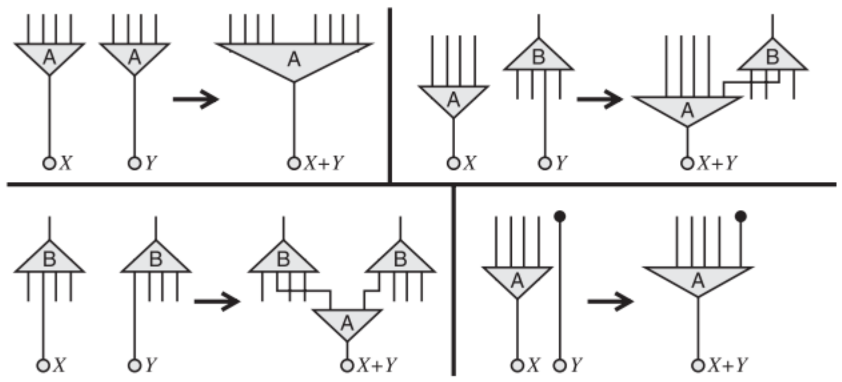
If we wish to go from the sets to the items, the properties 1–4 are replaced by their reflected versions:

1'. Each set node has exactly one outgoing edge.
2'. Each set in the union-find structure B has at least two incoming edges (the elements of the set) and exactly one outgoing edge (the current name of the set).
3'. Each set in the union-find structure A has exactly one incoming edge (the current name of the set) and at least two outgoing edges (the elements of the set).
4'. Each item node has exactly one incoming edge.

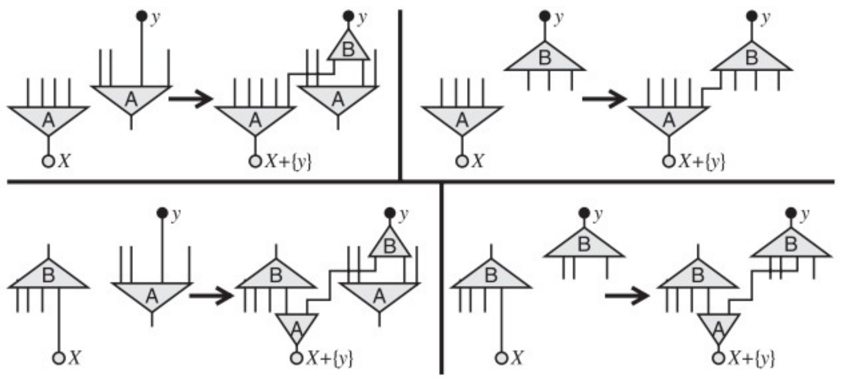
So the items are connected by unique alternating paths through structures A and B to their sets. The alternation property will be maintained in the updates by performing a set union whenever a set is directly connected to another set in the same structure; this preserves the existence and uniqueness of the paths between the item nodes and the set nodes.



Creating a Copy of a Set Node



Creating the Union of Two Set Nodes X and Y



Inserting Item y in Set X

The alternation property is central because it allows us to bound the total number of edges between the structures and make our listing queries output sensitive. Consider all the items contained in a given set; they correspond to a set of directed paths, which by the uniqueness of these paths has to form a directed tree, from the set node through the nodes in A and B to the item nodes. In this tree, each node has only one incoming edge, and each node in B has also only one outgoing edge (by property 2'). There are no two consecutive B nodes (alternation property), so if we contract the incoming and outgoing edges of each B node to one edge, we get a graph on the A and item nodes, in which each A node has at least two outgoing edges (by property 3'). So if the total number of leaves in this tree, that is, item nodes corresponding to items contained in the set, is k, then total number of A nodes is at most k − 1. Because each B node subdivides an edge of this graph and each edge is subdivided at most once, there are at most 2k − 1 B nodes in this tree. So if the set contains k elements, there are at most 3k − 2 nodes in structures A and B that are traversed by the tree.

If we take the sum over all sets, this gives an immediate bound on the the total number of edges between the structures A and B and the set and item nodes: it is of the order of the total size of the set system. Let n be that total size, that is, the sum of the sizes of the sets in the system. Then both structures A and B are union-find structures on an underlying set of size n.

From this description follows immediately the algorithm for list items. To list all items for a given set, we perform the following steps:

```
0. Put the initial outgoing edge of the set node on the stack.
1. While the stack is not empty, take the next edge from the stack.
    1.1 If this edge goes to an item node, list that item.
    1.2 If this edge goes to union-find structure A, perform a listing query and put all outgoing edges listed in the answer on the stack.
    1.3 If this edge goes to union-find structure B, perform a naming query and put the one outgoing edge in the answer on the stack.
```

(C) Least Common Ancestor (LCA) Queries. Given two nodes of a root-directed tree, each node tacitly defines a path to the root. What is the first node that lies on both paths? i.e., what is their common ancestor. (Brass 2008; section 6.4)

(D) Range Data Structure
Essentially a breadth-first search, say, for nodes in a graph sharing a common property. We record the all the adjacent nodes that share the property and/or the boundary nodes for the nodes that share the property.