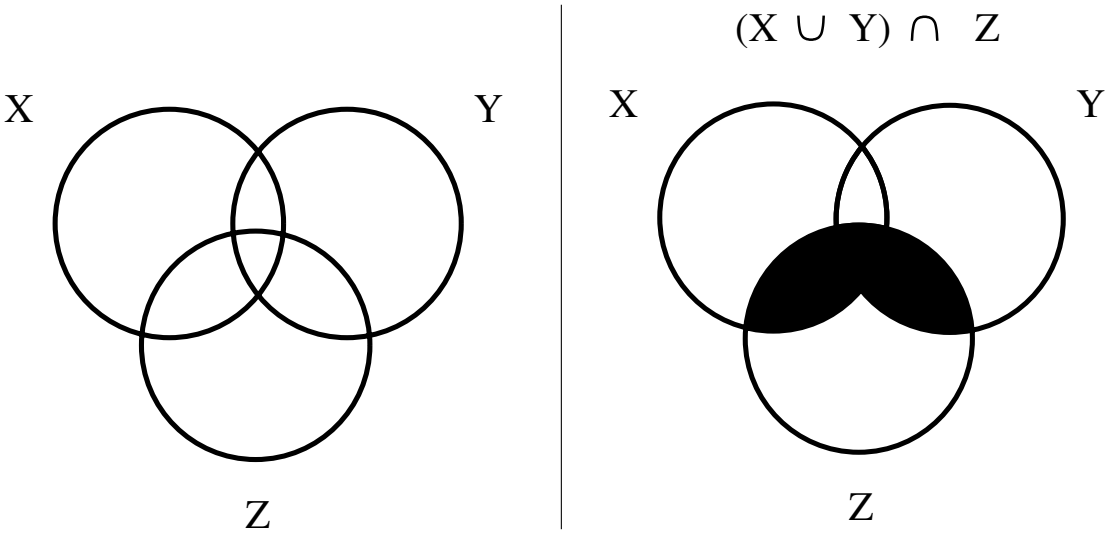

12.5 Set Data Structures

Input description: A universe of items $U = \{u_1, \ldots, u_n\}$ on which is defined a collection of subsets $S = \{S_1, \ldots, S_m\}$.

Problem description: Represent each subset so as to efficiently (1) test whether $u_i \in S_j$, (2) compute the union or intersection of S_i and S_j, and (3) insert or delete members of S.

Discussion: In mathematical terms, a set is an unordered collection of objects drawn from a fixed universal set. However, it is usually useful for implementation to represent each set in a single *canonical order*, typically sorted, to speed up or simplify various operations. Sorted order turns the problem of finding the union or intersection of two subsets into a linear-time operation—just sweep from left to right and see what you are missing. It makes possible element searching in sublinear time. Finally, printing the elements of a set in a canonical order paradoxically reminds us that order really doesn't matter.

We distinguish sets from two other kinds of objects: dictionaries and strings. A collection of objects *not* drawn from a fixed-size universal set is best thought of as a *dictionary*, discussed in Section 12.1 (page 367). Strings are structures where order matters—i.e. , if $\{A, B, C\}$ is not the same as $\{B, C, A\}$. Sections 12.3 and 18 discuss data structures and algorithms for strings.

Multisets permit elements to have more than one occurrence. Data structures for sets can generally be extended to multisets by maintaining a count field or linked list of equivalent entries for each element.

If each subset contains exactly two elements, they can be thought of as edges in a graph whose vertices represent the universal set. A system of subsets with no restrictions on the cardinality of its members is called a *hypergraph*. It is worth considering whether your problem has a graph-theoretical analogy, like connected components or shortest path in a graph/hypergraph.

Your primary alternatives for representing arbitrary subsets are:

- *Bit vectors* – An n-bit vector or array can represent any subset S on a universal set U containing n items. Bit i will be 1 if $i \in S$, and 0 if not. Since only one bit is needed per element, bit vectors can be very space efficient for surprisingly large values of $|U|$. Element insertion and deletion simply flips the appropriate bit. Intersection and union are done by "and-ing" or "or-ing" the bits together. The only drawback of a bit vector is its performance on sparse subsets. For example, it takes $O(n)$ time to explicitly identify all members of sparse (even empty) subset S.

- *Containers or dictionaries* – A subset can also be represented using a linked list, array, or dictionary containing exactly the elements in the subset. No notion of a fixed universal set is needed for such a data structure. For sparse subsets, dictionaries can be more space and time efficient than bit vectors, and easier to work with and program. For efficient union and intersection operations, it pays to keep the elements in each subset sorted, so a linear-time traversal through both subsets identifies all duplicates.

- *Bloom filters* – We can emulate a bit vector in the absence of a fixed universal set by hashing each subset element to an integer from 0 to n and setting the corresponding bit. Thus, bit $H(e)$ will be 1 if $e \in S$. Collisons leave some possibility for error under this scheme, however, because a different key might have hashed to the same position.

  Bloom filters use several (say k) different hash functions $H_1, \ldots H_k$, and set all k bits $H_i(e)$ upon insertion of key e. Now e is in S only if all k bits are 1. The probability of false positives can be made arbitrarily low by increasing the number of hash functions k and table size n. With the proper constants, each subset element can be represented using a constant number of bits independent of the size of the universal set.

  This hashing-based data structure is much more space-efficient than dictionaries for static subset applications that can tolerate a small probability of error. Many can. For instance, a spelling checker that left a rare random string undetected would prove no great tragedy.

Many applications involve collections of subsets that are pairwise disjoint, meaning that each element is in exactly one subset. For example, consider maintaining

the connected components of a graph or the party affiliations of politicians. Each vertex/hack is in exactly one component/party. Such a system of subsets is called a *set partition*. Algorithms for generating partitions of a given set are provided in Section 14.6 (page 456).

The primary issue with set partition data structures is maintaining changes over time, perhaps as edges are added or party members defect. Typical queries include "which set is a particular item in?" and "are two items in the same set?" as we modify the set by (1) changing one item, (2) merging or unioning two sets, or (3) breaking a set apart. Your basic options are:

- *Collection of containers* – Representing each subset in its own container/dictionary permits fast access to all the elements in the subset, which facilitates union and intersection operations. The cost comes in membership testing, as we must search each subset data structure independently until we find our target.

- *Generalized bit vector* – Let the ith element of an array denote the number/name of the subset that contains it. Set identification queries and single element modifications can be performed in constant time. However, operations like performing the union of two subsets take time proportional to the size of the universe, since each element in the two subsets must be identified and (at least one subset's worth) must have its name changed.

- *Dictionary with a subset attribute* – Similarly, each item in a binary tree can be associated a field that records the name of the subset it is in. Set identification queries and single element modifications can be performed in the time it takes to search in the dictionary. However, union/intersection operations are again slow. The need to perform such union operations quickly provides the motivation for the ...

- *Union-find data structure* – We represent a subset using a rooted tree where each node points to its *parent* instead of its children. The name of each subset will be the name of the item at the root. Finding out which subset we are in is simple, for we keep traversing up the parent pointers until we hit the root. Unioning two subsets is also easy. Just assign the root of one of two trees to point to the other, so now *all* elements have the same root and hence the same subset name.

  Implementation details have a big impact on asymptotic performance here. Always selecting the larger (or taller) tree as the root in a merger guarantees logarithmic height trees, as presented with our implementation in Section 6.1.3 (page 198). Retraversing the path traced on each find and explicitly pointing all nodes on the path to the root (called *path compression*) reduces the tree to almost constant height. Union find is a fast, simple data structure that every programmer should know about. It does not support breaking up subsets created by unions, but usually this is not an issue.

Implementations: Modern programming languages provide libraries offering complete and efficient set implementations. The C++ *Standard Template Library* (STL) provides `set` and `multiset` containers. *Java Collections* (JC) contains `HashSet` and `TreeSet` containers and is included in the `java.util` package of Java standard edition.

LEDA (see Section 19.1.1 (page 658)) provides efficient dictionary data structures, sparse arrays, and union-find data structures to maintain set partitions, all in C++.

Implementation of union-find underlies any implementation of Kruskal's minimum spanning tree algorithm. For this reason, all the graph libraries of Section 12.4 (page 381) presumably contains an implementation. Minimum spanning tree codes are described in Section 15.3 (page 484).

The computer algebra system *REDUCE* (*http://www.reduce-algebra.com/*) contains `SETS`, a package supporting set-theoretic operations on both explicit and implicit (symbolic) sets. Other computer algebra systems may support similar functionality.

Notes: Optimal algorithms for such set operations as intersection and union were presented in [Rei72]. Raman [Ram05] provides an excellent survey on data structures for set operations on a variety of different operations. Bloom filters are ably surveyed in [BM05], with recent experimental results presented in [PSS07].

Certain balanced tree data structures support merge/meld/link/cut operations, which permit fast ways to union and intersect disjoint subsets. See Tarjan [Tar83] for a nice presentation of such structures. Jacobson [Jac89] augmented the bit-vector data structure to support select operations (where is the ith 1 bit?) efficiently in both time and space.

Galil and Italiano [GI91] survey data structures for disjoint set union. The upper bound of $O(m\alpha(m,n))$ on m union-find operations on an n-element set is due to Tarjan [Tar75], as is a matching lower bound on a restricted model of computation [Tar79]. The inverse Ackerman function $\alpha(m,n)$ grows notoriously slowly, so this performance is close to linear. An interesting connection between the worst-case of union-find and the length of Davenport-Schinzel sequences—a combinatorial structure that arises in computational geometry—is established in [SA95].

The *power set* of a set S is the collection of all $2^{|S|}$ subsets of S. Explicit manipulation of power sets quickly becomes intractable due to their size. Implicit representations of power sets in symbolic form becomes necessary for nontrivial computations. See [BCGR92] for algorithms on and computational experience with symbolic power set representations.

Related Problems: Generating subsets (see page 452), generating partitions (see page 456), set cover (see page 621), minimum spanning tree (see page 484).