ADTs – Map/Dictionary and Set typical operations

- searching and lookup (access by value)

Map / **Dictionary** variations • **OrderedDictionary** – also supports min/max, predecessor(k)/ successor(k) based on an ordering of the keys	lookup (no duplicate keys)	• find(k) – find elt with key k if it exists • insert(k,v) – add elt v with key k • delete(k) – remove elt, key with key k (may return elt)
Set	membership (no duplicate elements)	• add(x) – add elt x if not already present • remove(x) – remove elt x • contains(x) – return whether x is present

  - a key conceptual difference between sets and dictionaries is the notion that set elements come from a fixed *universal set*
  - sets are unordered, but having a *canonical order* for elements facilitates mathematical set operations like union, intersection

Set Implementation

operation	dictionary – balanced search tree	dictionary – hashtable	array – sorted	array – unsorted
add	**O(log n)**	**O(1)** expected	**O(n)** – shift	**O(1)** – at end
remove	**O(log n)**	**O(1)** expected	**O(n)** – shift	**O(n)** – find, then swap
contains	**O(log n)**	**O(1)** expected	**O(log n)** – binary search	**O(n)** – sequential search
union	**O(n)** – inorder traversal	**O(N)** – traverse A + traverse B × contains	**O(n)** – inorder traversal	**O(n²)** – traverse A + traverse B × contains
intersection		**O(N)** – traversal × contains		**O(n²)** – traversal × contains

Set Implementation

- hashtable is O(1) for add/remove/contains but poor for union/intersection because elements are not in order

Can we do better?

- leverage the notion of a fixed universal set
  - *bit vector* – bit *i* = 1 if *i* is in the set, 0 otherwise
    - conceptually a boolean array, more efficient (space and time) using `int`(s) in binary form

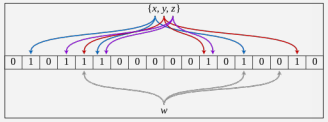
Set Implementation

operation	dictionary – balanced search tree	dictionary – hashtable	bit vector
add	**O(log n)**	**O(1)** expected	**O(1)** – flip bit
remove	**O(log n)**	**O(1)** expected	**O(1)** – flip bit
contains	**O(log n)**	**O(1)** expected	**O(1)** – check bit
union	**O(n)** – inorder traversal	**O(N)** – traverse A + traverse B × contains	**O(|U|)** – bitwise OR
intersection		**O(N)** – traversal × contains	**O(|U|)** – bitwise AND

Bit vectors are very space efficient (one bit per element), but require a fixed universal set *U* and traversal is inefficient for sparse sets. Can we do better?

Set Implementation

- Bloom filter
  - space-efficient probabilistic data structure optimized for contains operations
    - does not require an explicit universal set
    - appropriate in circumstances where occasional false positives are OK
  - consists of a bit vector with *m* bits and *k* different hash functions
    - *k* is a small constant
    - *m* depends on *k* and *n*
  - all bits are initialized to 0
  - add(*e*) – O(*k*)
    - compute $h_i(e)$ for each hash function h_i
    - set those bits to 1
  - contains(*e*) – O(*k*)
    - compute $h_i(e)$ for each hash function h_i
    - if any of those bits is 0, *e* is not in the set
    - if all of those bits are 1, *e* is probably in the set
  - does not store the elements themselves
  - remove(*e*) is not supported
    - other elements may hash to the same bits



https://en.wikipedia.org/wiki/Bloom_filter