Set (abstract data type)

From Wikipedia, the free encyclopedia

In computer science, a **set** is an abstract data type that can store certain values, without any particular order, and no repeated values. It is a computer implementation of the mathematical concept of a finite set. Unlike most other collection types, rather than retrieving a specific element from a set, one typically tests a value for membership in a set.

Some set data structures are designed for **static** or **frozen sets** that do not change after they are constructed. Static sets allow only query operations on their elements — such as checking whether a given value is in the set, or enumerating the values in some arbitrary order. Other variants, called **dynamic** or **mutable sets**, allow also the insertion and deletion of elements from the set.

An abstract data structure is a collection, or aggregate, of data. The data may be booleans, numbers, characters, or other data structures. If one considers the structure yielded by packaging[a] or indexing,[b] there are four basic data structures:[1][2]

1. unpackaged, unindexed: bunch
2. packaged, unindexed: set
3. unpackaged, indexed: string (sequence)
4. packaged, indexed: list (array)

In this view, the contents of a set are a bunch, and isolated data items are elementary bunches (elements). Whereas sets *contain* elements, bunches *consist of* elements.

Further structuring may be achieved by considering the multiplicity of elements (sets become multisets, bunches become hyperbunches)[3] or their homogeneity (a record is a set of fields, not necessarily all of the same type).

Contents [hide]

Type theory [edit]

In type theory, sets are generally identified with their indicator function (characteristic function): accordingly, a set of values of type A may be denoted by 2^A or $\mathcal{P}(A)$. (Subtypes and subsets may be modeled by refinement types, and quotient sets may be replaced by setoids.) The characteristic function F of a set S is defined as:

$$F(x) = \begin{cases} 1, & \text{if } x \in S \\ 0, & \text{if } x \notin S \end{cases}$$

In theory, many other abstract data structures can be viewed as set structures with additional operations and/or

Wikipedia The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Català
Deutsch
Español
Esperanto
Français
Italiano
עברית
Nederlands
日本語
Norsk bokmål
Polski
Português
Русский
Српски / srpski
ไทย
Edit links

additional axioms imposed on the standard operations. For example, an abstract heap can be viewed as a set structure with a `min(S)` operation that returns the element of smallest value.

Operations [edit]

Core set-theoretical operations [edit]

One may define the operations of the algebra of sets:

- `union(S, T)`: returns the union of sets *S* and *T*.
- `intersection(S, T)`: returns the intersection of sets *S* and *T*.
- `difference(S, T)`: returns the difference of sets *S* and *T*.
- `subset(S, T)`: a predicate that tests whether the set *S* is a subset of set *T*.

Static sets [edit]

Typical operations that may be provided by a static set structure *S* are:

- `is_element_of(x, S)`: checks whether the value *x* is in the set *S*.
- `is_empty(S)`: checks whether the set *S* is empty.
- `size(S)` or `cardinality(S)`: returns the number of elements in *S*.
- `iterate(S)`: returns a function that returns one more value of *S* at each call, in some arbitrary order.
- `enumerate(S)`: returns a list containing the elements of *S* in some arbitrary order.
- `build(`x_1`, `x_2`, ..., `x_n`,)`: creates a set structure with values $x_1, x_2, \ldots, x_n$.
- `create_from(collection)`: creates a new set structure containing all the elements of the given collection or all the elements returned by the given iterator.

Dynamic sets [edit]

Dynamic set structures typically add:

- `create()`: creates a new, initially empty set structure.
  - `create_with_capacity(n)`: creates a new set structure, initially empty but capable of holding up to *n* elements.
- `add(S, x)`: adds the element *x* to *S*, if it is not present already.
- `remove(S, x)`: removes the element *x* from *S*, if it is present.
- `capacity(S)`: returns the maximum number of values that *S* can hold.

Some set structures may allow only some of these operations. The cost of each operation will depend on the implementation, and possibly also on the particular values stored in the set, and the order in which they are inserted.

Additional operations [edit]

There are many other operations that can (in principle) be defined in terms of the above, such as:

- `pop(S)`: returns an arbitrary element of *S*, deleting it from *S*.[4]
- `pick(S)`: returns an arbitrary element of *S*.[5][6][7] Functionally, the mutator `pop` can be interpreted as the pair of selectors `(pick, rest)`, where `rest` returns the set consisting of all elements except for the arbitrary element.[8] Can be interpreted in terms of `iterate`.[c]
- `map(F, S)`: returns the set of distinct values resulting from applying function *F* to each element of *S*.
- `filter(P, S)`: returns the subset containing all elements of *S* that satisfy a given predicate *P*.
- `fold(`A_0`, F, S)`: returns the value $A_{|S|}$ after applying $A_{i+1} := F(A_i, e)$ for each element *e* of *S*, for some binary operation *F*. *F* must be associative and commutative for this to be well-defined.
- `clear(S)`: delete all elements of *S*.
- `equal(`S_1`, `S_2`)`: checks whether the two given sets are equal (i.e. contain all and only the same elements).
- `hash(S)`: returns a hash value for the static set *S* such that if `equal(`S_1`, `S_2`)` then `hash(`S_1`) = hash(`S_2`)`

Other operations can be defined for sets with elements of a special type:

- `sum(S)`: returns the sum of all elements of *S* for some definition of "sum". For example, over integers or reals, it may be defined as `fold(0, add, S)`.
- `collapse(S)`: given a set of sets, return the union.[9] For example, `collapse({{1}, {2, 3}}) == {1,`

`2, 3}`. May be considered a kind of `sum`.

- `flatten(S)`: given a set consisting of sets and atomic elements (elements that are not sets), returns a set whose elements are the atomic elements of the original top-level set or elements of the sets it contains. In other words, remove a level of nesting – like `collapse,` but allow atoms. This can be done a single time, or recursively flattening to obtain a set of only atomic elements.[10] For example, `flatten({1, {2, 3}}) == {1, 2, 3}`.
- `nearest(S,x)`: returns the element of *S* that is closest in value to *x* (by some metric).
- `min(S)`, `max(S)`: returns the minimum/maximum element of *S*.

Implementations [edit]

Sets can be implemented using various data structures, which provide different time and space trade-offs for various operations. Some implementations are designed to improve the efficiency of very specialized operations, such as `nearest` or `union`. Implementations described as "general use" typically strive to optimize the `element_of`, `add`, and `delete` operations. A simple implementation is to use a list, ignoring the order of the elements and taking care to avoid repeated values. This is simple but inefficient, as operations like set membership or element deletion are *O*(*n*), as they require scanning the entire list.[d] Sets are often instead implemented using more efficient data structures, particularly various flavors of trees, tries, or hash tables.

As sets can be interpreted as a kind of map (by the indicator function), sets are commonly implemented in the same way as (partial) maps (associative arrays) – in this case in which the value of each key-value pair has the unit type or a sentinel value (like 1) – namely, a self-balancing binary search tree for sorted sets (which has O(log n) for most operations), or a hash table for unsorted sets (which has O(1) average-case, but O(n) worst-case, for most operations). A sorted linear hash table[11] may be used to provide deterministically ordered sets.

Further, in languages that support maps but not sets, sets can be implemented in terms of maps. For example, a common programming idiom in Perl that converts an array to a hash whose values are the sentinel value 1, for use as a set, is:

```
my %elements = map { $_ => 1 } @elements;
```

Other popular methods include arrays. In particular a subset of the integers 1..*n* can be implemented efficiently as an *n*-bit bit array, which also support very efficient union and intersection operations. A Bloom map implements a set probabilistically, using a very compact representation but risking a small chance of false positives on queries.

The Boolean set operations can be implemented in terms of more elementary operations (`pop`, `clear`, and `add`), but specialized algorithms may yield lower asymptotic time bounds. If sets are implemented as sorted lists, for example, the naive algorithm for `union(S,T)` will take time proportional to the length *m* of *S* times the length *n* of *T*; whereas a variant of the list merging algorithm will do the job in time proportional to *m*+*n*. Moreover, there are specialized set data structures (such as the union-find data structure) that are optimized for one or more of these operations, at the expense of others.

Language support [edit]

One of the earliest languages to support sets was Pascal; many languages now include it, whether in the core language or in a standard library.

- In C++, the Standard Template Library (STL) provides the `set` template class, which is typically implemented using a binary search tree (e.g. red-black tree); SGI's STL also provides the `hash_set` template class, which implements a set using a hash table. In sets, the elements themselves are the keys, in contrast to sequenced containers, where elements are accessed using their (relative or absolute) position. Set elements must have a strict weak ordering.
- Java offers the `Set` interface to support sets (with the `HashSet` class implementing it using a hash table), and the `SortedSet` sub-interface to support sorted sets (with the `TreeSet` class implementing it using a binary search tree).
- Apple's Foundation framework (part of Cocoa) provides the Objective-C classes `NSSet`, `NSMutableSet`, `NSCountedSet`, `NSOrderedSet`, and `NSMutableOrderedSet`. The CoreFoundation APIs provide the `CFSet` and `CFMutableSet` types for use in C.
- Python has built-in `set` and `frozenset` types since 2.4, and since Python 3.0 and 2.7, supports non-

empty set literals using a curly-bracket syntax, e.g.: `{x, y, z}`.

- The .NET Framework provides the generic `HashSet` and `SortedSet` classes that implement the generic `ISet` interface.
- Smalltalk's class library includes `Set` and `IdentitySet`, using equality and identity for inclusion test respectively. Many dialects provide variations for compressed storage (`NumberSet`, `CharacterSet`), for ordering (`OrderedSet`, `SortedSet`, etc.) or for weak references (`WeakIdentitySet`).
- Ruby's standard library includes a `set` module which contains `Set` and `SortedSet` classes that implement sets using hash tables, the latter allowing iteration in sorted order.
- OCaml's standard library contains a `Set` module, which implements a functional set data structure using binary search trees.
- The GHC implementation of Haskell provides a `Data.Set` module, which implements immutable sets using binary search trees.[12]
- The Tcl Tcllib package provides a set module which implements a set data structure based upon TCL lists.
- The Swift standard library contains a `Set` type, since Swift 1.2.

As noted in the previous section, in languages which do not directly support sets but do support associative arrays, sets can be emulated using associative arrays, by using the elements as keys, and using a dummy value as the values, which are ignored.

Multiset [edit]

A generalization of the notion of a set is that of a **multiset** or **bag**, which is similar to a set but allows repeated ("equal") values (duplicates). This is used in two distinct senses: either equal values are considered *identical,* and are simply counted, or equal values are considered *equivalent,* and are stored as distinct items. For example, given a list of people (by name) and ages (in years), one could construct a multiset of ages, which simply counts the number of people of a given age. Alternatively, one can construct a multiset of people, where two people are considered equivalent if their ages are the same (but may be different people and have different names), in which case each pair (name, age) must be stored, and selecting on a given age gives all the people of a given age.

Formally, it is possible for objects in computer science to be considered "equal" under some equivalence relation but still distinct under another relation. Some types of multiset implementations will store distinct equal objects as separate items in the data structure; while others will collapse it down to one version (the first one encountered) and keep a positive integer count of the multiplicity of the element.

As with sets, multisets can naturally be implemented using hash table or trees, which yield different performance characteristics.

The set of all bags over type T is given by the expression bag T. If by multiset one considers equal items identical and simply counts them, then a multiset can be interpreted as a function from the input domain to the non-negative integers (natural numbers), generalizing the identification of a set with its indicator function. In some cases a multiset in this counting sense may be generalized to allow negative values, as in Python.

- C++'s Standard Template Library implements both sorted and unsorted multisets. It provides the `multiset` class for the sorted multiset, as a kind of associative container, which implements this multiset using a self-balancing binary search tree. It provides the `unordered_multiset` class for the unsorted multiset, as a kind of unordered associative containers, which implements this multiset using a hash table. The unsorted multiset is standard as of C++11; previously SGI's STL provides the `hash_multiset` class, which was copied and eventually standardized.
- For Java, third-party libraries provide multiset functionality:
  - Apache Commons Collections provides the `Bag` and `SortedBag` interfaces, with implementing classes like `HashBag` and `TreeBag`.
  - Google Guava provides the `Multiset` interface, with implementing classes like `HashMultiset` and `TreeMultiset`.
- Apple provides the `NSCountedSet` class as part of Cocoa, and the `CFBag` and `CFMutableBag` types as part of CoreFoundation.
- Python's standard library includes `collections.Counter`, which is similar to a multiset.
- Smalltalk includes the `Bag` class, which can be instantiated to use either identity or equality as predicate for inclusion test.

Where a multiset data structure is not available, a workaround is to use a regular set, but override the equality predicate of its items to always return "not equal" on distinct objects (however, such will still not be able to store

multiple occurrences of the same object) or use an associative array mapping the values to their integer multiplicities (this will not be able to distinguish between equal elements at all).

Typical operations on bags:

- `contains(B, x)`: checks whether the element *x* is present (at least once) in the bag *B*
- `is_sub_bag(B1, B2)`: checks whether each element in the bag B_1 occurs in B_1 no more often than it occurs in the bag B_2; sometimes denoted as $B_1 \sqsubseteq B_2$.
- `count(B, x)`: returns the number of times that the element *x* occurs in the bag *B*; sometimes denoted as *B* # *x*.
- `scaled_by(B, n)`: given a natural number *n*, returns a bag which contains the same elements as the bag *B*, except that every element that occurs *m* times in *B* occurs *n* * *m* times in the resulting bag; sometimes denoted as n ⊗ B.
- `union(B1, B2)`: returns a bag that containing just those values that occur in either the bag B_1 or the bag B_2, except that the number of times a value *x* occurs in the resulting bag is equal to (B_1 # x) + (B_2 # x); sometimes denoted as $B_1 \uplus B_2$.

Multisets in SQL [edit]

In relational databases, a table can be a (mathematical) set or a multiset, depending on the presence on unicity constraints on some columns (which turns it into a candidate key).

SQL allows the selection of rows from a relational table: this operation will in general yield a multiset, unless the keyword `DISTINCT` is used to force the rows to be all different, or the selection includes the primary (or a candidate) key.

In ANSI SQL the `MULTISET` keyword can be used to transform a subquery into a collection expression:

```
SELECT expression1, expression2... FROM table_name...
```

is a general select that can be used as *subquery expression* of another more general query, while

```
MULTISET(SELECT expression1, expression2... FROM table_name...)
```

transforms the subquery into a *collection expression* that can be used in another query, or in assignment to a column of appropriate collection type.

See also [edit]

- Bloom filter
- Disjoint set

Notes [edit]

a. ^ "Packaging" consists in supplying a container for an aggregation of objects in order to turn them into a single object. Consider a function call: without packaging, a function can be called to act upon a bunch only by passing each bunch element as a separate argument, which complicates the function's signature considerably (and is just not possible in some programming languages). By packaging the bunch's elements into a set, the function may now be called upon a single, elementary argument: the set object (the bunch's package).

b. ^ Indexing is possible when the elements being considered are totally ordered. Being without order, the elements of a multiset (for example) do not have lesser/greater or preceding/succeeding relationships: they can only be compared in absolute terms (same/different).

c. ^ For example, in Python `pick` can be implemented on a derived class of the built-in `set` as follows:

```
class Set(set):
    def pick(self):
        return next(iter(self))
```

d. ^ Element insertion can be done in *O*(1) time by simply inserting at an end, but if one avoids duplicates this takes *O*(*n*) time.

References [edit]

1. ^ Hehner, Eric C. R. (1981), "Bunch Theory: A Simple Set Theory for Computer Science", *Information Processing*

Letters **12** (1): 26, doi:10.1016/0020-0190(81)90071-5
2. ^ Hehner, Eric C. R. (2004), *A Practical Theory of Programming, second edition*
3. ^ Hehner, Eric C. R. (2012), *A Practical Theory of Programming, 2012-3-30 edition*
4. ^ Python: pop()
5. ^ *Management and Processing of Complex Data Structures: Third Workshop on Information Systems and Artificial Intelligence, Hamburg, Germany, February 28 - March 2, 1994. Proceedings,* ed. Kai v. Luck, Heinz Marburger, p. 76
6. ^ Python Issue7212: Retrieve an arbitrary element from a set without removing it; see msg106593 regarding standard name
7. ^ Ruby Feature #4553: Add Set#pick and Set#pop
8. ^ *Inductive Synthesis of Functional Programs: Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning,* Ute Schmid, Springer, Aug 21, 2003, p. 240
9. ^ *Recent Trends in Data Type Specification: 10th Workshop on Specification of Abstract Data Types Joint with the 5th COMPASS Workshop, S. Margherita, Italy, May 30 - June 3, 1994. Selected Papers, Volume 10,* ed. Egidio Astesiano, Gianna Reggio, Andrzej Tarlecki, p. 38
10. ^ Ruby: flatten()
11. ^ Wang, Thomas (1997), *Sorted Linear Hash Table*
12. ^ Stephen Adams, "*Efficient sets: a balancing act*", Journal of Functional Programming 3(4):553-562, October 1993. Retrieved on 2015-03-11.

v · t · e **Data types** [show]

v · t · e **Data structures** [show]

Categories: Data types | Composite data types | Abstract data types