Data Structures/Sets

Sets are helpful tools in a software application where, just as in mathematics, similar abstract objects are aggregated into collections. A mathematical set has a fairly simple interface and can be implemented in a surprising number of ways.

`Set<item-type>` ADT

`contains``(`*`test-item`*`:item-type):Boolean`
    True if *test-item* is contained in the Set.
`insert``(`*`new-item`*`:item-type)`
    Adds *new-item* into the set.
`remove``(`*`item`*`:item-type)`
    Removes *item* from the set. If *item* wasn't in the set, this method does nothing.
`remove``(`*`item-iter`*`:List Iterator<item-type>)`
    Removes the item referred to by *item-iter* from the set.
`get-begin``():List Iterator<item-type>`
    Allows iteration over the elements in the Set.
`get-end``():List Iterator<item-type>`
    Also allows iteration over the elements in the Set.
`union``(`*`other`*`:Set<item-type>):Set<item-type>`
    Returns a set containing all elements in either this or the *other* set. Has a default implementation.
`intersect``(`*`other`*`:Set<item-type>):Set<item-type>`
    Returns a set containing all elements in both this and the *other* set. Has a default implementation.
`subtract``(`*`other`*`:Set<item-type>):Set<item-type>`
    Returns a set containing all elements in this but not the *other* set. Has a default implementation.
`is-empty``():Boolean`
    True if no more items can be popped and there is no top item.
`get-size``():Integer`
    Returns the number of elements in the set.

All operations can be performed in time.

List implementation

There are several different ways to implement sets. The simplest, but in most cases least efficient, method is to simply create a linear list (an array, linked list or similar structure) containing each of the elements in the set. For the most basic operation, testing membership, a possible implementation could look like

```
function contains(List<T> list, T member)
    for item in list
        if item == member
            return True
    return False
```

To add new members to this set, simply add the element to beginning or end of the list. (If a check is done to ensure no duplicate elements, some other operations may be simpler.) Other operations can be similarly implemented in terms of simple list operations. Unfortunately, the membership test has a worst-case running time of if the item is not in the list, and even an average-case time of the same, assuming the item is equally likely to be anywhere in the list. If the set is small, or if frequently accessed items can be placed near the front of the list, this may be an efficient solution, but other options can have a faster running time.

Assuming elements can be ordered and insertions and deletions are rare, a list guaranteed to be in sorted order with no duplicate elements can be much more efficient. Using an ordered list, the membership test can be efficient to the order of . Additionally, union, intersection and subtraction can be implemented in linear time, whereas they would take quadratic time with unordered lists.

Bit array implementation

For certain data, it may be more practical to maintain a bit array describing the contents of the set. In this representation, there is a 1 or 0 corresponding to each element of the problem domain, specifying whether the object is an element of the set. For a simple case, assume that only integers from 0 to n can be members of the set, where n is known beforehand. This can be represented by a bit array of length n+1. The *contains* operation is simple:

```
function contains(BitArray array, Int member)
    if member >= length(array)
       return False
    else if array[member] == 1
       return True
    else
       return False
```

To add or remove an element from this sort of set, simply modify the bit array to reflect whether that index should be 1 or 0. The membership runs in exactly (constant) time, but it has a severely restricted domain. It is possible to shift the domain, going from m to n with step k, rather than 0 to n with step 1 as is specified above, but there is not much flexibility possible here. Nevertheless, for the right domain, this is often the most efficient solution.

Bit arrays are efficient structures for storing sets of Boolean variables. One example is a set of command line options that enable various run-time behavior for the application. C and similar languages offer bit-wise operators that let the programmer access a bit field in a single machine instruction, where array access would normally need two or three instructions including a memory read operation. A full-featured bit set implementation includes operators for computing a set union, set intersection, set difference, and element values.[1]

Associative array implementation

Associative arrays—that is, hash tables and binary search trees, represent a heavyweight but general representation of sets. Binary trees generally have time implementations for lookup and mutation for a particular key, and hash tables have a implementation (though there is a higher constant factor). Additionally, they are capable of storing nearly any key type. The membership test is trivial: simply test if the potential set member exists as a key in the associative array. To add an element, just add the set member as a key in the associative array, with a dummy value. In optimized implementations, instead of reusing an existing associative array implementation, it is possible to write a specialized hash table or binary tree which does not store values corresponding to keys. Values are not meaningful here, and they take up a constant factor of additional memory space.

References

1. Samuel Harbison and Guy Steele. C: a reference manual. 2002.

Retrieved from "https://en.wikibooks.org/w/index.php?title=Data_Structures/Sets&oldid=3329049"