Sets, Tables, and Dictionaries

- Set specification
- Set representations — characteristic function, lists, ordered lists
- Table specification
- Table representations

Reading: Lambert and Osborne, Sections 12.2 and 13.1
Goodrich and Tamassia, Chapter 10

Introduction

In this section, we examine three ADTs: *sets*, *tables*, and *dictionaries*, used to store collections of elements with no repetitions.

Note that these names are used (eg in different texts) for a range of similar ADTs — we define them as follows:

Set

- used when set-theoretic operations are required
- elements may or may not be ordered
- includes "membership" operations: *isEmpty*, *insert*, *delete*, *isMember*
- includes "set-theoretic" operations: *union*, *intersection*, *difference*, *size*, *complement*

Table

- simpler version of Set without the set-theoretic operations
- elements assumed to be unordered

Dictionary

- like Table but assumes elements are totally ordered
- includes "order related" operations: *isPredecessor*, *isSuccessor*, *predecessor*, *successor*, *range*

Elements, Records, and Keys

Elements may be a single items, or “records” with unique *keys* (such as those typically found in databases).

We will usually talk about elements as if they are single items.

eg. “if $e_1 < e_2$ then…”

In the case of record elements, this can be considered shorthand for

“if $k_1 < k_2$, where k_1 is the key of record e_1 and k_2 is the key of record e_2, then…”

Examples

The following are examples of situations where the ADTs might be used:
Set:

"I have one set of students who do CITS2200 and one set of students who do CITS2210. What is the set of students who do both?"

Table:

"I begin with the set of students originally enrolled in CITS2200. These two students joined. This one withdrew. Is a particular student currently enrolled?"

Dictionary:

"Here is the set of students enrolled in CITS2200 ordered by (exact) age. Which are the students between the ages of 18 and 20?"

Set Specification

1. **Constructors**
2. *Set()*: create an empty set.
3. **Checkers**
4. *isEmpty()*: returns *true* if the set is empty, *false* otherwise.
5. *isMember(e)*: returns *true* if *e* is a member of the set, *false* otherwise.

6. **Manipulators**

7. *size()*: returns the cardinality of (number of elements in) the set.

8. *complement()*: returns the complement of the set (only defined for finite universes).

9. *insert(e)*: forms the union of the set with the singleton $\{e\}$

10. *delete(e)*: removes e from the set

11. *union(t)*: returns the union of the set with t.

12. *intersection(t)*: returns the intersection of the set with t.

13. *difference(t)*: returns the set obtained by removing any items that appear in t.

14. *enumerate()*: returns the "next" element of the set. Successive calls to *enumerate* should return successive elements until the set is exhausted.

Set Representations

Characteristic Function Representation

Assume A is a set from some universe U.
The *characteristic function* of A is defined by:

$$f(e) = \begin{cases} \textit{true} \ \ (\text{or } \ 1) & e \in A \\ \textit{false} \ \ (\text{or } \ 0) & \text{otherwise} \end{cases}$$

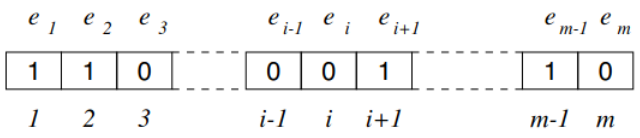
$\rightarrow$ thus a set can be viewed as a boolean function.

If U is finite and '$\leq$' is a total order on U, the elements of U can be enumerated as the sequence

$$e_1, \ldots, e_m$$

where $e_i \leq e_j$ if $i < j$, and m is the cardinality of U.

The characteristic function maps this sequence to a sequence of 1s and 0s. Thus the set can be represented as a block of 1s and 0s, or a *bit vector*...



Sometimes called a *bitset* — eg. `java.util.BitSet`

Advantage

Translates set operations into efficient bit operations:

- *insert* — *or* the appropriate bit with 1
- *delete* — *and* the appropriate bit with 0
- *isMember* — is the (boolean) value of the appropriate bit
- *complement* — complement of a bit vector
- *union* — *or* two bit vectors
- *intersection* — *and* two bit vectors
- *difference* — *complement* and *intersection*

Also *enumerate* — can cycle through the m positions reporting 1s.

Performance

- *insert*, *delete*, *isMember* — constant providing index can be calculated in constant time
- *complement*, *union*, *intersection*, *difference* — $O(m)$; linear in size of *universe*
- *enumerate* — $O(m)$ for n calls, where n is size of set
  $\rightarrow O(\frac{m}{n})$ amortized over n calls

Disadvantages

- If the universe is large compared to the size of sets then:
  - the latter operations are expensive
  - large amount of space wasted
- Requires the universe to be bounded, totally ordered, and known in advance. If not, eg Insert, will need to move things.

List Representation

An alternative is to represent the set as a list using one of the List representations. Here, we assume there is no total ordering on the elements.

Performance

Assume we have a set of size p.

insert, *delete*, *isMember* — take $O(p)$ time; the best that can be achieved in an unordered list (recall `eSearch`)

union — for each item in the first set, check if it is a member of the second, and if not, add it (to the result)

$\rightarrow O(pq)$ where p and q are the sizes of the two sets

Other set operations (*intersection*, *difference*) behave similarly.

Note that if both sets grow at the same rate (the worst case), the time performance is $O(p^2)$.

Inefficient because one list must be traversed for each element in the other. Can we traverse both at the same time. . . ?

Ordered List Representation

If the universe is totally ordered, we can obtain more efficient implementations by merging the two in sorted order.

Assume A can be enumerated as $a_1, a_2, \ldots, a_p$ and B can be enumerated as $b_1, b_2, \ldots, b_q$.

Eg. *union*

```
$i = 1$; $j = 1$;
do {
  if ($a_i$ == $b_j$)  add $a_i$ to $C$ and increment $i$ and $j$;
  else add smaller of $a_i$ and $b_j$ to $C$ and increment its index;
}
while ($i$ <= $p$ && $j$ <= $q$);
add any remaining $a_i$'s or $b_j's$ to $C$
```

Give pseudo-code for the *intersection* and *difference* operations.

```
**if** $a_i = b_j$
**then** add $a_i$ to $C$ and increment $i$ and $j$
**else** increment the index of the smaller of $a_i$ and $b_j$

**if** $a_i < b_j$ **then** add $a_i$ to $C$ and increment $i$
**else if** $b_j < a_i$ **then** increment $j$
**else** increment $i$ and $j$
```

Performance

Each list is traversed once $\rightarrow O(p+q)$ time.

This is much better than $O(pq)$.

If p and q grow at the same rate (worst case), the time performance is now $O(p)$.

Note also that *isMember* is now $O(\log p)$ (recall `bSearch`)

Table Specification

The Table operations are a subset of the Set operations:

1. **Constructors**
2. *Table()*: create an empty table.
3. **Checkers**
4. *isEmpty()*: returns *true* if the table is empty, *false* otherwise.
5. *isMember(e)*: returns *true* if e is in the table, *false* otherwise.
6. **Manipulators**
7. *insert(e)*: forms the union of the table with the singleton $\{e\}$
8. *delete(e)*: removes e from the table

Table Representations

Since the Table operations are a subset of those of Set, the (unordered) List representations can be used.

insert, *delete*, *isMember* therefore take $O(p)$ time.

The more efficient List representations and the characteristic function representation are not available since the elements are assumed to be unordered.

The operations can be made more efficient by considering the probability distribution for accesses over the list and moving more probable (or more frequently accessed) items to the front — see Wood, Section 8.3.

Later, we'll look in detail at a more efficient representation of tables using *hashing*, where such operations are close to constant time.

Summary

We have outlined several ADTs for use with collections of unique elements or records, and considered representations:

- Set — includes set-theoretic operations, elements may or may not be ordered
- Table — restriction of Set with fewer operations, elements assumed not ordered
- List — can be used for unordered sets and tables
- ordered list (block) — can improve efficiency for ordered sets.
- characteristic function — can be very efficient for ordered sets over a fixed domain.
- Next — Dictionaries...