Data Structures
Fall 2018
(Other) Non linear DS

Luis de Marcos Ortega

luis.demarcos@uah.es

Contents

- Dictionaries
- Hash tables
  - Open hashing
  - Closed hashing

Bibliography

- Chapter 4 of:
  - A.V. AHO., J.E. HOPCROFT., J.D. ULLMAN. 1987. “Data Structures and Algorithms.” Addison-Wesley.

Dictionaries

- A *set* is a collection of *members* (or *elements*)
- All members of a set are different, which means no set can contain two copies of the same element.
- A *dictionary* is a set ADT with the operations INSERT, DELETE, and MEMBER
  - We shall also include MAKENULL

Dictionaries

- MEMBER(x, A) takes set A and object x, whose type is the type of elements of A, and returns a boolean value -- true if $x \in A$ and false if $x \notin A$.
- INSERT(x, A), where A is a set-valued variable, and x is an element of the type of A's members, makes x a member of A. Note that if x is already a member of A, then INSERT(x, A) does not change A.
- DELETE(x, A) removes x from A. If x is not in A originally, DELETE(x, A) does not change A.
- MAKENULL(A) makes the null set be the value for set variable A.

The ADT dictionary

```
spec DICTIONARY[ITEM]
     genres dictionary, item
     operations
          member: dictionary item -> boolean
          insert: dictionary item -> dictionary
          delete: dictionary item -> dictionary
          makenull: dictionary -> dictionary
endspec
```

Simple dictionary implementations

- by a sorted or unsorted linked list
- by a bit vector
  – provided the elements of the underlying set are restricted to the integers 1, . . . , N for some N, or are restricted to a set that can be put in correspondence with such a set of integers.

Simple dictionary implementations

- by a fixed- length array with a pointer to the last entry of the array in use
  - This implementation is only feasible if we can assume our sets never get larger than the length of the array.
    - Advantage: simplicity over the linked-list representation
    - Disadvantages: (1) sets cannot grow arbitrarily, (2) deletion is slower, and (3) space is not utilized efficiently if sets are of varying sizes.
- We will not consider these implementations

Hash tables

- Hashing is a widely useful technique for implementing dictionaries
  - It requires constant time per operation, on the average, and there is no requirement that sets be subsets of any particular finite universal set.
  - In the worst case, this method requires time proportional to the size of the set for each operation, just as the array and list implementations do.
  - By careful design, however, we can make the probability of hashing requiring more than a constant time per operation be arbitrarily small.

Hash tables

- two somewhat different forms of hashing
  - *open* or *external* hashing, allows the set to be stored in a potentially unlimited space, and therefore places no limit on the size of the set.
  - *closed* or *internal* hashing, uses a fixed space for storage and thus limits the size of sets.

Open hashing

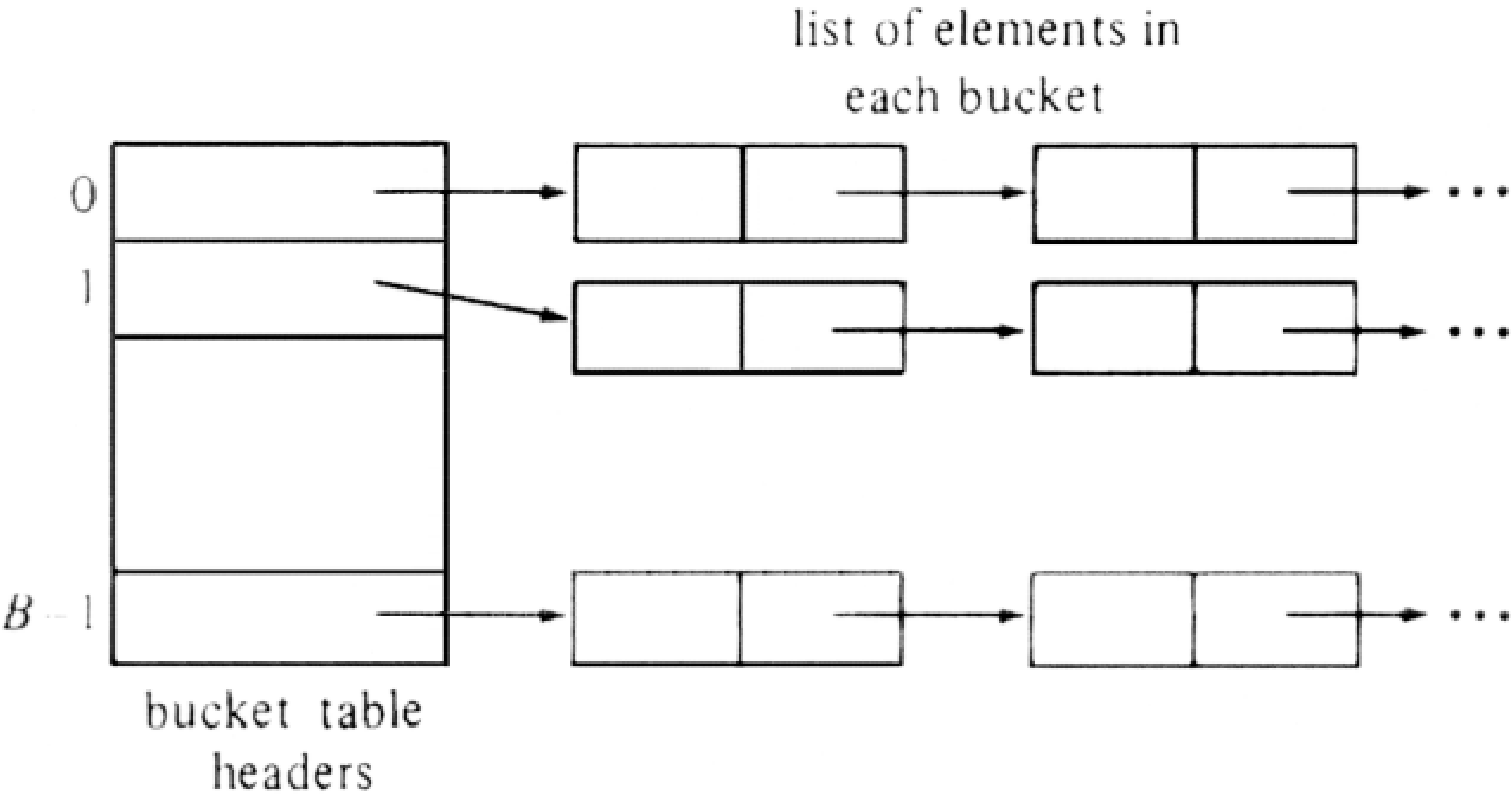
- The essential idea is that the (possibly infinite) set of potential set members is partitioned into a finite number of classes.

- To do that, we use a hash function

Open hashing

- If we wish to have B classes, numbered 0, 1, . . . , B-1, then we use a *hash function* h such that for each object x of the data type for members of the set being represented, $h(x)$ is one of the integers 0 through B-1.
- The value of $h(x)$, naturally, is the class to which x belongs. We often call x the *key* and $h(x)$ the *hash value* of x. The "classes" we shall refer to as *buckets*, and we say that x belongs to bucket $h(x)$.

Open hashing



Open hashing

- In an array called the *bucket table*, indexed by the *bucket numbers* 0, 1, . . . , B-1, we have headers for B lists.
- The elements on the ith list are the members of the set being represented that belong to class i, that is, the elements x in the set such that $h(x) = i$.

Open hashing

- It is our hope that the buckets will be roughly equal in size, so the list for each bucket will be short. If there are *N* elements in the set, then the average bucket will have *N/B* members.
- If we can estimate *N* and choose *B* to be roughly as large, then the average bucket will have only one or two members, and the dictionary operations take, on the average, some small constant number of steps, independent of what *N* (or equivalently *B*) is.

The open hash table DS

```
const B:= {some suitable constant}

celltype = record
     element: elementtype
     next: ^celltype
endrecord

A: array[0..B-1] of ^celltype {in a class}

item: elementtype
```

Open hashing example

- Hash function: h(x) = x mod 5.

1. Create the bucket table
2. Show the hash table after inserting the following elements
   – 12, 8, 137, 20, 38, 14, 27, 42
3. Delete the following elements
   – 137, 42, 38, 20

Open hashing

- Running times of operations: open hash table

Operation	Average	Worst case
Member	O(1)	O(n)
Insert	O(1)	O(1)
Delete	O(1)	O(n)

- Time analysis of open and closed hashing
  - in (AHO, HOPCROFT & ULLMAN, 1987) “Data Structures and Algorithms.” Chapter 4. Section 4.8.

Closed hashing

- A closed hash table keeps the members of the dictionary in the bucket table itself, rather than using that table to store list headers.
- Associated with closed hashing is a *rehash strategy*.
  - If we try to place x in bucket $h(x)$ and find it already holds an element, a situation called a *collision* occurs, and the rehash strategy chooses a sequence of alternative locations, $h_1(x)$, $h_2(x)$, . . . , within the bucket table in which we could place x. We try each of these locations, in order, until we find an empty one. If none is empty, then the table is full, and we cannot insert x.

Closed hashing

- Example:
  - Suppose $B = 8$ and keys a, b, c, and d have hash values $h(a) = 3$, $h(b) = 0$, $h(c) = 4$, $h(d) = 3$.
  - We shall use the simplest rehashing strategy, called *linear rehashing*, where $h_i(x) = (h(x) + i)$ **mod B**.
  - Thus, for example, if we wished to insert a and found bucket 3 already filled, we would try buckets 4, 5, 6, 7, 0, 1, and 2, in that order.

Closed hashing

- Example

0	*b*
1	
2	
3	*a*
4	*c*
5	*d*
6	
7	

Closed hashing

- For *deletions* the most effective approach is to place a constant called *deleted* into a bucket that holds an element that we wish to delete.
- The *membership test* for an element x requires that we examine $h(x)$, $h_i(x)$, $h_2(x)$, . . . , until we either find x or find an empty bucket.

The closed hash table DS

```
const
      B:= {some suitable constant}
      empty := '          ' {10 blanks}
      deleted := '**********' {10 *'s}

A: array[0..B-1] of elementtype {in a class}

item: elementtype
```

Closed hashing

- In a closed hashing scheme the speed of insertion and other operations depends not only on how randomly the hash function distributes the elements into buckets, but also on how well the rehashing strategy avoids additional collisions when a bucket is already filled.

- Time analysis of open and closed hashing
  - in (AHO, HOPCROFT & ULLMAN, 1987) "Data Structures and Algorithms." Chapter 4. Section 4.8.

Restructuring hash tables

- If we use an open hash table, the average time for operations increases as *N/B*, a quantity that grows rapidly as the number of elements exceeds the number of buckets. Similarly, for a closed hash table, and it is not possible that *N* exceeds *B*.
- If *N* gets too large,
  - for example $N \geq .9B$ for a closed table or $N \geq 2B$ for an open one,
- we simply create a new hash table with twice as many buckets, and insert the current elements into the new table. Hash function has to be redefined too.

Closed hashing example

- Hash function: h(x) = x mod 5.

1. Create the bucket table
2. Show the hash table after inserting the following elements
   - 12, 8, 137, 20, 38, 14, 27, 42
   - (restructure the table if needed)
3. Delete the following elements
   - 137, 42, 38, 20

(Other) Non linear DS

Luis de Marcos Ortega

luis.demarcos@uah.es