

Highlights of Tables and Hashing

Tables: Ch. 11, pp.515-522
Hashing: Ch. 12 pp.598-604

8/2400 Y-1

Looking Up Data

- A common pattern in many programs is to look up data
 - Find student record, given ID#
 - Find phone #, given person's
 - CS example: a compiler's "symbol table"
look up identifier to find its type, location, etc.
- Because it is so common, many data structures for tables have been investigated
 - We could use arrays, linked lists, general trees, binary search trees, etc.
 - Could also step back and consider look-up as an abstract operation

8/2400 Y-2

Table Terminology

- **Key:** Portion of pair used to look up data, like an index (aka *domain* value)
- **Value:** Portion of pair that contains data (aka *range* value)

aTable:	
"4476542K"	23,440
"3828122E"	27,640
"24601JVJ"	15,203
"994802WE"	45,210
"8675309A"	28,776

Key (employee ID) Value (salary)

8/2400 Y-3

Table ADT

- Characteristics of *table* ADT type
 - Set of key/value pairs
 - No duplicate keys
- Operations on tables
 - retrieve value from key
 - insert value at key
 - delete key and associated value from table
- Uses:
 - Phone book, class roster, book index, databases
- Sometimes also called a *dictionary* or *map*

8/2400 Y-4

Example of Operations

```
// phone book example: name is key, phone# is value
Table pb;

pb.insert("Sarah", 5552345);
pb.insert("Richard", 3450011);
pb.insert("Bart", 6661212);
int bartsNumber = pb.retrieve("Bart");
pb.remove("Richard");
```

8/2400 Y-5

Structures for Efficient Retrieval

- In many Table applications, *retrieve* is the most common operation
 - So retrieve needs to be efficient
- Many different data structures are possible
 - Array (sorted or unsorted?), List, Binary tree, Binary search tree. But not... stack, queue
 - Footnote: Some languages (Perl, Javascript) contain a built-in "associative array" construct
- With a sorted array and binary search, *retrieve* would be $O(\log N)$
 - Same for binary search tree *find*

8/2400 Y-6

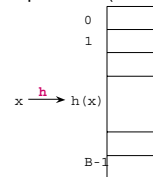
Can we do better than $O(\log N)$?

- Answer: Yes ... sort of, if we're lucky.
 - General idea: *take the key of the data record you're inserting, and use that number directly as the index into an array.*
 - Example:
 - Assume you want quick access to a table of your friends. All of them have unique social security numbers (in the range 000-00-0000 to 999-99-9999).
 - If you had an array with 1,000,000,000 elements, each friend could be instantly located in the array by their social security number.
- What's wrong with the above scheme?*

8/2400 Y-7

Hash Functions

- Basic idea:
 - Don't use the data value directly.
 - Given an array of size B , use a *hash function*, $h(x)$, which maps the given data record x to some (hopefully) unique index ("bucket") in the array.



8/2400 Y-8

Hashing and Tables

- Hashing gives us another implementation of Table ADT
- Retrieve algorithm: Hash the key; this gives an index; use it to find the value stored in the table
- If this scheme worked, it would be $O(1)$
 - Great improvement over $\log N$.
- Main problems
 - Finding a good hash function
 - Collisions
 - Wasted space in the table

8/2400 Y-9

Slides past this point were not covered during Summer 2000

8/2400 Y-10

An Apparent Sidetrack

- Problem to solve: *Given a list of n integers, determine if there is a pair of duplicate values*

37591	31576	64085	42782	25475	70900	79953	76186
67887	84848	81309	30822	77867	45852	65289	8322
79367	40520	58053	16030	34723	22116	41073	60522
34399	31616	85965	82102	73707	38316	153	11282
7623	61416	10741	46686	73123	69780	65105	21866
75567	5760	66525	80214	63835	48652	49593	42066
20055	16248	12213	35758	12147	13828	7729	2266
13263	57984	73181	34246	51755	58053	31817	52754
23863	9160	56677	62462	65715	68404	48097	66762
37519	52480	28045	68294	71131	6252	81689	51570
72119	71944	9797	77822	56563	67348	51553	86986
88303	10656	6925	89654	63099	25036	84393	47426

8/2400 Y-11

Element Uniqueness: Two Solutions

1. Nested loop: for each element in the array, scan the rest of the array to see if there is a duplicate.
 - $O(n^2)$
 2. Sort the data. Then scan array (once) for duplicates.
 - $O(n \log n)$ time to sort, $O(n)$ to scan.
- Anything simpler??
 - Any solution that looks like hashing?

8/2400 Y-12

Element Uniqueness (2)

- Step 1: Assign to buckets, based on value mod 100

```

37591 31576 64085 42782 25475 70900 79953 76186
67887 84848 81309 30822 77867 45852 65289 8322
79367 40520 58053 16030 34723 22116 41073 60522
34399 31616 85965 82102 73707 38316 153 11282
7623 61416 10741 46686 73123 69780 65105 21866
75567 5760 66525 80214 63835 48652 49593 42066
20055 16248 12213 35758 12147 13828 7729 2266
13263 57984 73181 34246 51755 58053 31817 52754
23863 9160 56677 62462 65715 68404 48097 66762
37519 52480 28045 68294 71131 6252 81689 51570
72119 71944 9797 77822 56563 67348 51553 86986
88303 10656 6925 89654 63099 25036 84393 47426
    
```

8/2400 Y-13

Element Uniqueness (3)

- Step 2: Look inside each bucket for duplicates

```

37591 31576 64085 42782 25475 70900 79953 76186
67887 84848 81309 30822 77867 45852 65289 8322
79367 40520 58053 16030 34723 22116 41073 60522
34399 31616 85965 82102 73707 38316 153 11282
7623 61416 10741 46686 73123 69780 65105 21866
75567 5760 66525 80214 63835 48652 49593 42066
20055 16248 12213 35758 12147 13828 7729 2266
13263 57984 73181 34246 51755 58053 31817 52754
23863 9160 56677 62462 65715 68404 48097 66762
37519 52480 28045 68294 71131 6252 81689 51570
72119 71944 9797 77822 56563 67348 51553 86986
88303 10656 6925 89654 63099 25036 84393 47426
    
```

8/2400 Y-14

Hashing Functions

- The hash function we choose depends on the type of the key field (the key we use to do our lookup).

- Finding a good one can be hard

- Example:

- Student Ids (integers)

$h(\text{idNumber}) = \text{idNumber} \% B$

eg. $h(678921) = 678921 \% 100 = 21$

- Names (char strings)

$h(\text{name}) = (\text{sum over the ascii values}) \% B$

eg. $h("Bill") = (66+105+108+108) \% 100 = 87$

8/2400 Y-15

Collisions

- Collisions occur when multiple items are mapped to same cell

```

h(idNumber) = idNumber % B
h(678921) = 21
h(354521) = 21
    
```

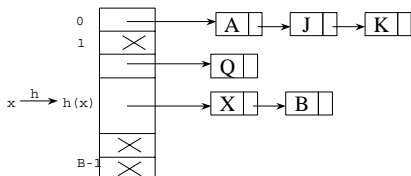
- Issues

- Relative size of table to number of data items
- Choice of hash function
- With a bad choice of hash function we can have lots of collisions
- Even with a good choice of hash functions there may be some collisions

8/2400 Y-16

Collision Resolution

- One strategy: *Bucket hashing* (aka open hashing)
- Each cell (bucket) in the array contains a *linked list* of items:



Many other solutions have been studied

8/2400 Y-17

Analysis of hash table ops

- Insert* is easy to analyze:

- It is just the cost of calculating the hash value $O(1)$, plus the cost of inserting into the front of a linked list $O(1)$

- Retrieve* and *Delete* are harder. To do the analysis, we need to know:

- The number of elements in the table (N)
- The number of buckets (B)
- The quality of the hash function

8/2400 Y-18

Hashing Analysis (2)

- We'll assume that our hash function distributes items evenly through the table
 - so each bucket contains N/B items on average (N/B is called the *load factor*)
- On average, doing a lookup or a deletion is $O(N/B)$ (Which is $O(1)$, if N/B is constant)
- Using a good hash function and keeping B large with respect to N , we can guarantee constant time insertion, deletion, and lookup
- Note that this means growing (rehashing) the hash table as more items are inserted.

8/24/00 Y-19

Open vs. Closed Hashing

- Open hashing uses linked lists for the buckets
- What is closed hashing??
- All data is stored in the array
 - If there is a collision, the next available cell is used
 - Avoids overhead of linked lists and works well in practice
 - Example: hash the following into a table of size 10
17, 23, 47, 52, 71, 86, 63, 96



8/24/00 Y-20

Dynamic Hashing

- Another implementation concept
- As number of stored records increases, dynamically increase the number of buckets
- Brute force: make a new (larger) array and copy (rehash) all the data to it
- More subtle implementations are also known

8/24/00 Y-21

Hashing and Files

- We've spoken of the hashed data as being stored in an array (in memory)
- Hashing is also very appropriate for disk files
- Efficient look-up techniques for disk data are essential
 - Disks are thousands of times slower than memory
 - Even a $\log N$ look-up algorithm is too slow for a database application!
 - Many structures we have studied (linked lists, trees, etc.) do *not* scale well to large disk files

8/24/00 Y-22

Drawbacks to Hashing

- Finding a good hash function
 - Small risk of bad behavior
- Dealing with collisions
 - Simplest method to use linked list for buckets
- Wasted space in the array
 - Not a big deal if memory is cheap
- Doesn't support ordering queries (such as we would want for a real dictionary)

8/24/00 Y-23

Summary

- Hash tables are specialized for dictionary operations: *Insert, Delete, Lookup*
- Principle: **Turn the key field of the record into a number, which we use as an index for locating the item in an array.**
- $O(1)$ in the ideal case; less in practice
- Problems: collisions, wasted space
- Implementations: open hashing, closed hashing, dynamic hashing
- Highly suitable for database files, too

8/24/00 Y-24