CSE 373: Hash Tables
(using them and dealing with collisions)

Chapter 5

Hash Table Sets: Use

Hash tables can be used to store sets
e.g., the set of all departments represented in CSE 373

\[
\text{typedef enum \{ACMS, ARCH, ART, BIOCHM, ...\} dept;}
\]

Approach: Just store the departments themselves in the hash table:
- to add a new department, Insert() it
- to see if a department is represented, Find() it
**Hash Table Sets: Implementation**

*Data Structure*

```c
typedef struct _HashTableStruct {
    int tablesiz;e;
    dept *data;
} HashTableStruct;
```

*Sample Operation*

```c
Insert (HashTable T, dept D)
```

- hash D to get an index, I
- check whether data[I] is empty (or already storing D)
- if so, set data[I] = D
- else deal with the conflict

---

**Hashing Records**

*Goal:* store the CSE 373 class list as a Hash Table

```c
typedef struct _student {
    name first, last;
    int UWID;
    name email;
    char college;
    dept major;
    int class;
} student;
```

*Implementation:*

Same as set, but array of students rather than departments
Hashing Records: Design Decisions

Design Decisions:
What to hash on?
- last name?
- first name?
- student ID?
- email?
- some combination thereof?

How to look someone up?
- supply entire record?
- supply just a single field?

Another Hash Table Interface

Some hash tables separate key from data:

```c
void Insert(HashTable, KeyType, DataType);
DataType Find(HashTable, KeyType);
```

Question: How to implement a database?

Goals:
- store records as in class list example
- be able to search based on any field (or some subset)
- minimize space requirements
Load Factor

Load Factor: Density of hash table, $\lambda$

$\lambda = \frac{\text{# of stored elements}}{\text{table size}}$

$\lambda = \frac{3}{7}$

Ideally, we’d like $\lambda \approx 1.0$

Dealing with Collisions

What can we do when two keys hash to the same slot?

<table>
<thead>
<tr>
<th>Insert (T, EE)</th>
<th>Insert (T, ACMS)</th>
<th>Insert (T, SPAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(EE) = 2$</td>
<td>$f(ACMS) = 5$</td>
<td>$f(SPAN) = 2$</td>
</tr>
<tr>
<td>EE</td>
<td>EE</td>
<td>!!!</td>
</tr>
<tr>
<td>ACMS</td>
<td>ACMS</td>
<td></td>
</tr>
</tbody>
</table>
Solution: Separate Chaining

Idea: At each position, store a list of the data that hashes to that position

Separate Chaining: Implementation

typedef struct _HashTable {
    int tablesiz e;
    List *datalist;
} HashTable;

Insert ()

- hash key
- see if key is already in list (Find (datalist [I], ...))
- if not, insert it into the list (Insert (datalist [I], ...))

(Note that we could replace lists with BSTs, hash tables)
Solution 2: Rehashing

Grow the size of the hash table as it gets full
But when?
  - whenever there is a collision?
  - whenever the \( \lambda \) reaches 1.0?
  - whenever \( \lambda \) reaches \( k \)?
  - whenever \( n\% \) of the slots are in use?

Can we simply \texttt{realloc()} the data array?

Running Time of Rehashing

Assume that we'll rehash whenever \( \lambda = 1.0 \)...
  - starting with an array of size 11
  - approximately doubling the size of the array
    (use the next prime larger than \( 2 \times \) tables\( z e \))
  - what is the total running time of inserting \( n \) keys?