Hash Table Sets: Use

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

```cpp
typedef enum {ACMS, ECON, EE, MATH, ...} dept;
HashTable<dept> D;
```

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:

```cpp
template <class HashedObj>
class HashTable {
  private:
    int tablesize;
    HashedObj* data;
};
```

Sample Operation:

- `HashTable::insert(HashedObj& key)`
- hash `key` to get an index, `i`
- check whether `data[i]` is empty (or already storing `key`)
- if so, set `data[i] = key`
- otherwise deal with the conflict

Hashing Records

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

```cpp
class student {
  name first, last;
  int UWID;
  name email;
  dept major;
  int year;
};
```

Implementation:

Use a hash table of students rather than departments:

```cpp
HashTable<student> studentSet;
```
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?

Food For Thought

**Question:** How to implement a simple database?

**Goals:**

- store records as in class list example
- be able to search based on *any* field
- 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?

- $D.\text{insert}(EE)$
- $D.\text{insert}(ACMS)$
- $D.\text{insert}(SPAN)$

- $hash(EE) = 2$
- $hash(ACMS) = 5$
- $hash(SPA\text{N}) = 2$

---

 UW, Autumn 1999  
 CSE 373 – Data Structures and Algorithms  
 Band Chamberlain
Solution: Separate Chaining

Idea: At each position, store a list of the keys that hash to that position

Separate Chaining: Implementation

```
template <class HashedObj>
class HashTable {
  private:
    int tableszie;
    List<HashedObj>* datalist;
};

HashTable::insert(HashedObj& key)
  • hash key (i = hash(key))
  • see if key is already in list (datalist[i].find(key))
  • if not, insert into the list (datalist[i].insert(key))

(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 $\lambda$ reaches 1.0?
- whenever $\lambda$ reaches $k$?
- whenever $n\%$ of the slots are in use?

Can we simply resize the data array and copy values over as we did with lists and stacks?

---

**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 $2 \times$ tablesze)
- what is the total running time of inserting $n$ keys?
Solution 3: Open Addressing

Goal: Use available space in table to store collisions rather than lists or resizing
- linear probing
- quadratic probing
- double hashing

Linear Probing

If there’s a collision, insert data in next blank slot:

D.insert(SAN)  D.insert(MATH)

\[
\begin{array}{ccc}
\text{EE} & \text{EE} & \text{EE} \\
\text{hash (SAN)}=2 & \text{hash (MATH)}=2 & \\
\text{ACMS} & \text{ACMS} & \text{ACMS}
\end{array}
\]

Note that if there is an open slot in the table, linear probing will always find it (eventually)
**Finding, Deleting w/ Linear Probing**

D. **find**(SPAN)  D. **remove**(SPAN)  D. **find**(MATH)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>SPAN</td>
<td>SPAN</td>
<td>SPAN</td>
</tr>
<tr>
<td>MATH</td>
<td>MATH</td>
<td>MATH</td>
</tr>
<tr>
<td>ACMS</td>
<td>ACMS</td>
<td>ACMS</td>
</tr>
</tbody>
</table>

hash (SPAN) = 2
hash (MATH) = 2

**Primary Clustering**

Linear probing has the tendency to result in clusters of data in the table
- increases search time for values hashing to that area

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>SPAN</td>
<td>SPAN</td>
</tr>
<tr>
<td>MATH</td>
<td>MATH</td>
</tr>
<tr>
<td>ACMS</td>
<td>ACMS</td>
</tr>
</tbody>
</table>

cluster```
Open Addressing Requirements

- The selection of alternate slots must be recomputable and deterministic
  - so that we can find() data that we’ve inserted
- Deletion from the table must be “lazy”
  - similar to binary search trees
  - don’t remove data, simply mark it as being deleted

Open Addressing: General Form

Open addressing is generally expressed as:

\[(\text{hash(key)} + f(i)) \mod \text{tablesize}, \text{ for } i = 0, 1, 2, \ldots\]

The hashing procedure is therefore:

1) Try \((\text{hash(key)} + f(0)) \mod \text{tablesize}\)
2) If it’s full, try \((\text{hash(key)} + f(1)) \mod \text{tablesize}\)
3) Continue until you find an empty slot

Design decision: what to use for \(f()\)?

- Linear probing uses \(f(i) = i\)
**Quadratic Probing**

Uses $f(i) = i^2$

**D.insert (SPAN)**

- **hash (SPAN)=2**

**D.insert (MATH)**

- **hash (MATH)=2**

**Quadratic Probing: Evaluation**

- **Intuition**: spreads things out more, so primary clustering should not be as much of a problem

- It can be proven that quadratic probing is guaranteed to find a free slot if...
  - number of slots is prime
  - table is less than half full
  - (therefore, resize when $\lambda = 0.5$)
**Double Hashing**

\[ f(i) = i \cdot \text{hash}_2(\text{key}) \]

*Intuition:* since good hash functions result in fairly random distributions, this spreads values out in a less predictable pattern.

---

**Quadratic Probing**

Uses \( f(i) = i^2 \)

<table>
<thead>
<tr>
<th>( \text{D.insert(SPAN)} )</th>
<th>( \text{D.insert(MATH)} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{EE} )</td>
<td>( \text{EE} )</td>
</tr>
<tr>
<td>( \text{hash(SPAN)} = 2 )</td>
<td>( \text{hash(MATH)} = 2 )</td>
</tr>
<tr>
<td>( \text{hash}_2(SPAN) = 5 )</td>
<td>( \text{hash}_2(MATH) = 3 )</td>
</tr>
<tr>
<td>( \text{ACMS} )</td>
<td>( \text{ACMS} )</td>
</tr>
</tbody>
</table>
Applications: Compilers

Compilers use hash tables to store information about all user-defined identifiers.

```cpp
int i;
List<int> mylist;

void getname() {
    char name[20];
    ...
}
```

Applications: AI

- Create a hash function for a game’s “position”
- Store “good moves” from each position as they are discovered
- While playing, can quickly check if there is a known good move from the current position