Data Structures and Algorithms
Searching and Hashing

Chris Brooks

Department of Computer Science
University of San Francisco
We know how to arrange things (sorting)

What about efficient ways of *finding* things?

We’ve seen a couple of ways to do this already ...

- List
- Binary Search Tree
Let’s assume that we want to store records in some sort of database.

Records have keys and values (data).

Our database should have the following operations:

- **Add** a key / value pair to the database
- **Remove** a key (and associated value) from the database
- **Find** the value associated with a key
If database is implemented as a sorted list:

- Add
- Remove
- Find
15-3: Sorted List Implementation

If database is implemented as a sorted list:

- **Add** $O(n)$
- **Remove** $O(n)$
- **Find** $O(\log n)$
If database is implemented as a Binary Search Tree:

- Add
- Remove
- Find
If database is implemented as a Binary Search Tree:

- **Add** $O(\lg n)$ best, $O(n)$ worst
- **Remove** $O(\lg n)$ best, $O(n)$ worst
- **Find** $O(\lg n)$ best, $O(n)$ worst
Maintain an *unsorted, non-contiguous* array of elements

| 15 | 4  | 3  | 13 | 8  | 6  |

- How long does a Find take?
- How long does a Remove take?
- How long does an Add take?

Does this sound like a good idea?
What if we had a “magic function” –
- Takes a key as input
- Returns the index in the array where the key can be found, if the key is in the array

To add an element
- Put the key through the magic function, to get a location
- Store element in that location

To find an element
- Put the key through the magic function, to get a location
- See if the key is stored in that location
The “magic function” is called a *Hash function*

If $\text{hash}(\text{key}) = i$, we say that the key hashes to the value $i$.

We’d like to ensure that different keys will always hash to different values.

Why is this not possible?
The “magic function” is called a *Hash function*

If \( \text{hash}(\text{key}) = i \), we say that the \text{key} hashes to the value \( i \)

We’d like to ensure that different keys will always hash to different values.

Why is this not possible?

- Too many possible keys
- If keys are strings of up to 15 letters, there are \( 10^{21} \) different keys
When two keys hash to the same value, a collision occurs.

We cannot avoid collisions, but we can minimize them by picking a hash function that distributes keys evenly through the array.

Example: Keys are integers
- Keys are in range $1 \ldots m$
- Array indices are in range $1 \ldots n$
- $n << m$
When two keys hash to the same value, a collision occurs.

We cannot avoid collisions, but we can minimize them by picking a hash function that distributes keys evenly through the array.

Example: Keys are integers
- Keys are in range $1 \ldots m$
- Array indices are in range $1 \ldots n$
- $n << m$

hash$(k) = k \mod n$
6. What if table size = 10, all keys end in 0?
What if table size = 10, all keys end in 0?
What if table size is even, all keys are even?
What if table size = 10, all keys end in 0?

What if table size is even, all keys are even?

In general, what if the table size and many of the keys share factors?
What if table size = 10, all keys end in 0?

What if table size is even, all keys are even?

In general, what if the table size and many of the keys share factors?

What can we do?
What if table size = 10, all keys end in 0?

What if table size is even, all keys are even?

In general, what if the table size and many of the keys share factors?

What can we do?
- Prevent keys and table size from sharing factors.
- Typically, we don’t get to pick the keys - they’re given externally
- We can control the table size
What if table size = 10, all keys end in 0?

What if table size is even, all keys are even?

In general, what if the table size and many of the keys share factors?

What can we do?

- Prevent keys and table size from sharing factors.
- No control over the keys.
- Make the table size prime.
Hash tables are usually used to store string values

If we can convert a string into an integer, we can use the integer hash function

How can we convert a string into an integer?
Hash tables are usually used to store string values

If we can convert a string into an integer, we can use the integer hash function

How can we convert a string into an integer?
- Add up ASCII values of the characters in the string

```java
int hash(String key, int tableSize) {
    int hashvalue = 0;
    for (int i=0; i<key.length(); i++)
        hashvalue += (int) key.charAt(i);
    return hashvalue % tableSize;
}
```
Hash tables are usually used to store string values.

If we can convert a string into an integer, we can use the integer hash function.

How can we convert a string into an integer?
- Concatenate ASCII digits together

\[
\sum_{k=0}^{\text{keysize}-1} \text{key}[k] \times 256^{\text{keysize}-k-1}
\]
Concatenating digits does not work, since numbers get big too fast. Solutions:

- Overlap digits a little (use base of 32 instead of 256)
- Ignore early characters (shift them off the left side of the string)

```java
static long hash(String key, int tablesize) {
    long h = 0;
    int i;
    for (i=0; i<key.length(); i++)
        h = (h << 4) + (int) key.charAt(i);
    return h % tablesize;
}
```
For each new character, the hash value is shifted to the left, and the new character is added to the accumulated value.

If the string is long, the early characters will “fall off” the end of the hash value when it is shifted.
   - Early characters will not affect the hash value of large strings.

Instead of falling off the end of the string, the most significant bits can be shifted to the middle of the string, and XOR’ed.

Every character will influence the value of the hash function.
static long ELFhash(String key, int tablesize) {
    long h = 0;
    long g;
    int i;

    for (i=0; i<key.length(); i++) {
        h = (h << 4) + (int) key.charAt(i);
        g = h & 0xF0000000L;
        if (g != 0)
            h ^= g >>> 24
            h &= ~g
    }
    return h \% M;
}
When two keys hash to the same value, a collision occurs.

A collision strategy tells us what to do when a collision occurs.

Two basic collision strategies:

- Open Hashing (Closed Addressing, Separate Chaining)
- Closed Hashing (Open Addressing)
15-25: Open Hashing

- Array does not store elements, but linked-lists of elements
- To Add an element to the hash table:
  - Hash the key to get an index $i$
  - Store the key/value pair in the linked list at index $i$
- To find an element in the hash table
  - Hash the key to get an index $i$
  - Search the linked list at index $i$ for the key
- This should remind you of a sorting algorithm ...
Under the following conditions:

- Keys are evenly distributed through the hash table
- Size of the hash table = # of keys inserted

What is the running time for the following operations:

- Add
- Remove
- Find
Under the following conditions:

- Keys are evenly distributed through the hash table
- Size of the hash table = # of keys inserted

What is the running time for the following operations:

- Add $\Theta(1)$
- Remove $\Theta(1)$
- Find $\Theta(1)$
Values are stored in the array itself (no linked lists)

The number of elements that can be stored in the hash table is limited to the table size (hence closed hashing)
To add element $X$ to a closed hash table:

- Find the smallest $i$, such that $\text{Array}[\text{hash}(x) + f(i)]$ is empty (wrap around if necessary)
- Add $X$ to $\text{Array}[\text{hash}(x) + f(i)]$
- If $f(i) = i$, linear probing
15-30: Closed Hashing

Problems with linear probing:

- Primary Clustering
  - “Clumps” – large sequences of consecutively filled array elements – tend to form
  - Positive feedback system – the larger the clumps, the more likely an element will end up in a clump.
6 Quadratic probing
   △ Find the smallest i, such that Array[hash(x) + f(i)] is empty
   △ Add X to Array[hash(x) + f(i)]
   △ f(i) = \(i^2\)
15-32: Closed Hashing

- Quadratic probing
  - Find the smallest \( i \), such that \( \text{Array[hash(x) + f(i)]} \) is empty
  - Add \( X \) to \( \text{Array[hash(x) + f(i)]} \)
  - \( f(i) = i^2 \)

- Problems:
  - Can’t reach all elements in the list
Quadratic probing

- Find the smallest $i$, such that $\text{Array[hash(x) + f(i)]}$ is empty
- Add $X$ to $\text{Array[hash(x) + f(i)]}$
- $f(i) = i^2$

Problems:

- Can’t reach all elements in the list
- (if table is less than $1/2$ full, and table size is an integer, guaranteed to be able to add an element)
Pseudo-Random

- Create a “Permutation Array” $P$
- $f(i) = P[i]$
Multiple keys hash to the same element
  ▷ Secondary clustering

Double Hashing
  ▷ Use a secondary hash function to determine how far ahead to look
  ▷ \( f(i) = i \times \text{hash2(key)} \)
Deletion from an open hash table is easy.
   ▲ Find the element.
   ▲ Delete it.

Deletion from a closed hash table is harder.
   ▲ Why?
Deletion a closed hash table can cause problems

Three different kinds of entries
- Empty cells
- Cells that contain data
- Cells that have been deleted (tombstones)
To insert an element:
- Find the smallest $i$ such that $\text{hash}(x) + f(i)$ is either empty or deleted

To find an element
- Try all values of $i$ (starting with 0) until either
  - $\text{Table}[\text{hash}(x) + f(i)] = x$
  - $\text{Table}[\text{hash}(x) + f(i)]$ is empty (*not* deleted)
What can we do when our closed hash table gets full?
– Or if the load (# of elements / table size) gets larger than 0.5
  ▶ Create a new, larger table
    ▶ New hash table will have a different hash function, since the table size is different
  ▶ Add each element in the old table to the new table
When we create a new table, it should be approx. twice as large as the old table

- A single insert can now require $\Theta(n)$ work
- ... but only after $\Theta(n)$ inserts
- Time for $n$ inserts is $\Theta(n)$
- Average time for an insert is still $\Theta(1)$

What happens if we make the table 100 units larger, instead of twice as large?

- Rememeber to keep the table size prime!