HASH TABLES

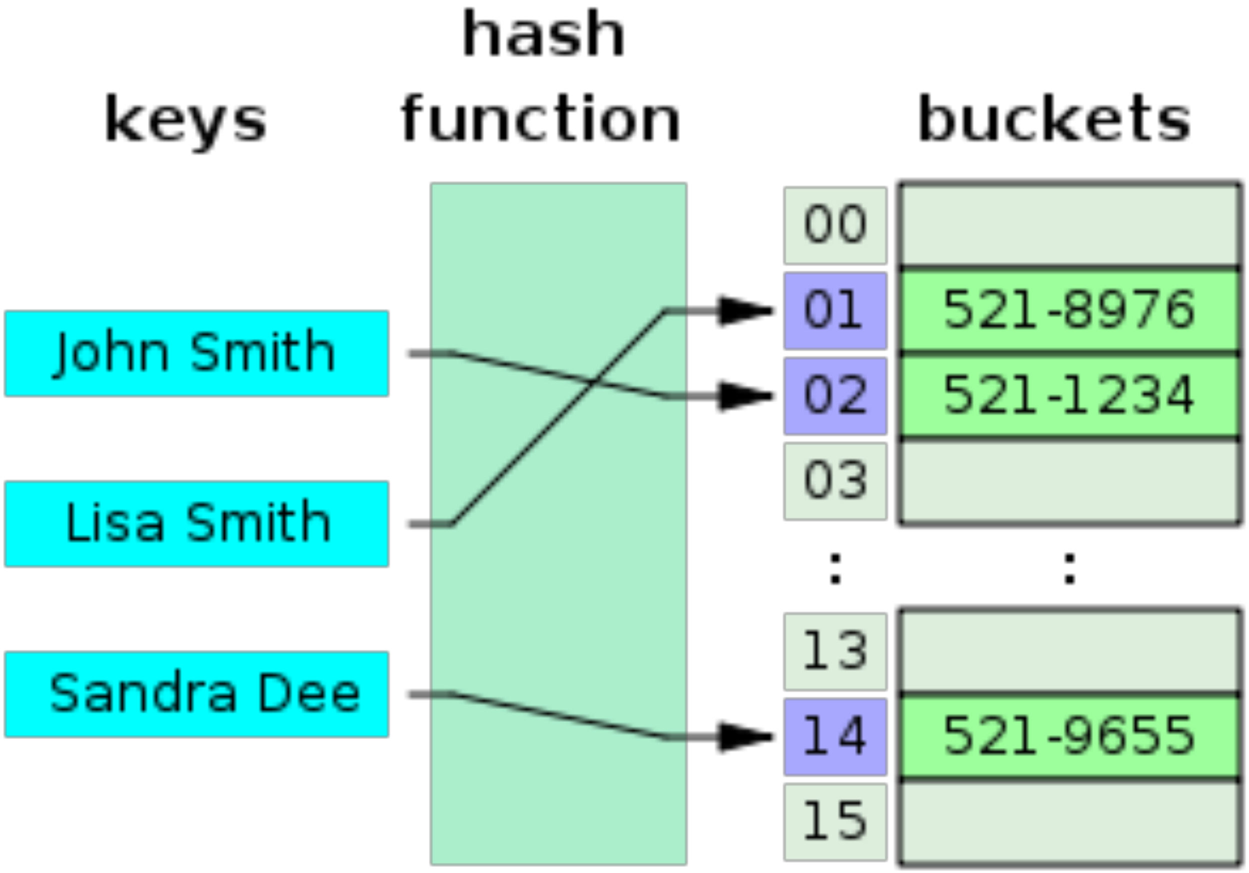
SUPPLEMENT

WHAT IS A HASH TABLE?

- **A hash table is just an array BUT instead of assignment incremental positions for data elements we use a special hashing function.**
  - The hashing function operates on a key.
  - The result maps a location where data can be retrieved or stored.
  - Keys are may/may not be stored.

hash(key) → data element position

EXAMPLE (WIKIPEDIA)



HASH FUNCTION

- **Features of an ideal hash function:**
  - Fast
  - Maps to unique locations

HASH FUNCTION #1

Assume: Keys are integers.

```
H(key) = key % tableSize
```

- **How good is the fit?**
  - What if keys are uniformly random?
  - What is all of the keys end in 0?

HASH FUNCTION #2

String Keys (8 chars or less):

```
public static int hash(String key, int tableSize){
        int hashVal = 0;
        for(int i = 0; i < key.lenth(); i++)
                hashVal += key.charAt(i);  //sum of ASCII
        return hashVal % tableSize;
}

//Looks promising?
//What about small tables?
//Large tables?
```

HASH FUNCTION #3

String Keys (3 chars only):

```
public static int hash(String key, int tableSize){
        return (key.charAt(0) + 27 * key.charAt(1) +
                729 * key.charAt(2)) % tableSize;
}

//Key is weighted on the first three characters of a string
//only.  If letters are random, this is pretty good, even for
//tables ~10,000 elements in size.

//But…
```

HASH FUNCTION #4

String Keys:

```
public static int hash(String key, int tableSize){
        int hashVal = 0;
        for(int i = 0; i < key.length(); i++)
                hashVal = 37 * hashVal + key.charAt(i);

        hashVal %= tableSize;
        if(hashVal < 0)
                hashVal += tableSize;
        return hashVal;
}
```

HOW TO DEAL WITH COLLISIONS?

We have seen this before: How to deal with collisions?

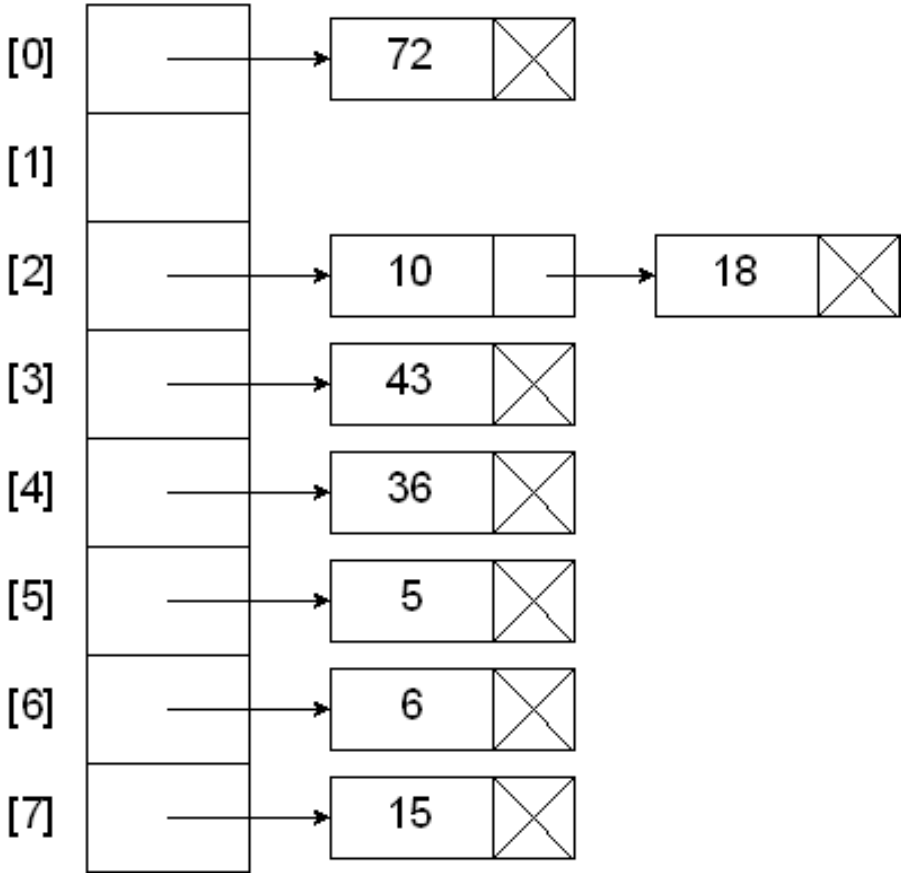
- **Perfect hashing is difficult in most cases so we need to deal with the likelihood of colliding values and want to minimize the insert and retrieval of such duplicates.**
- **Remember, different keys can map to the same location so we may have different values at the same location.**

- **We ~could~ ignore collisions and treat data as first-come-first-served but this may not be possible/desirable.**

SEPARATE CHAINING

Hash key = key % table size

4 = 36 % 8
2 = 18 % 8
0 = 72 % 8
3 = 43 % 8
6 = 6 % 8
2 = 10 % 8
5 = 5 % 8
7 = 15 % 8



http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm

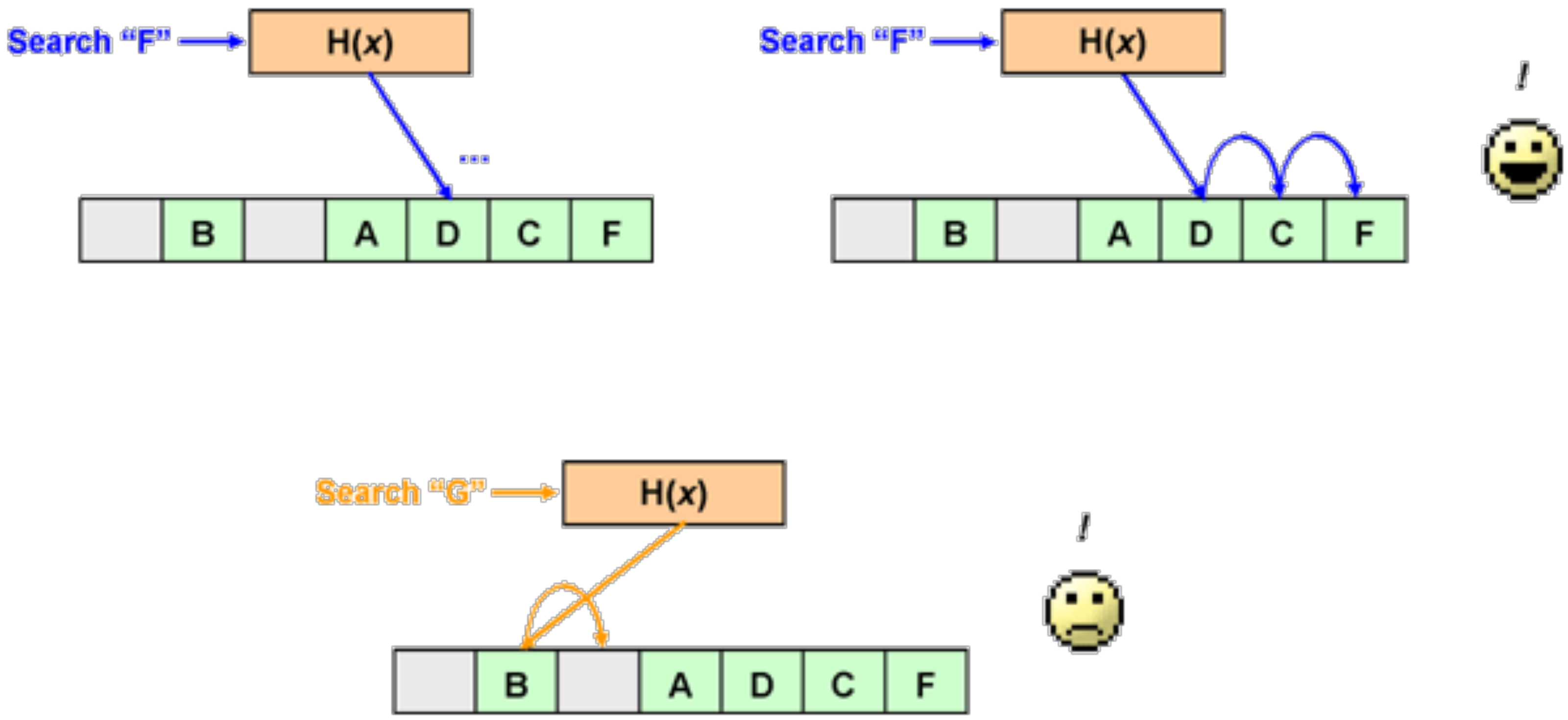
LINEAR PROBING

One way to avoid linked lists is to hash to a given location and IF an element exists, probe through the array until a free location is found.

- **This can result in "primary clustering" and creates issues for retrieval.**

- **Algorithm:**
  - Calculate a hash code from the key
  - Access that hash element
    - If the hash element is empty, add straight away
    - If not, probe through subsequent elements (looping back if necessary), trying to find a free place
      - If a free place is found, add the data at that position
      - If no free place is found, the add will fail.

LINEAR RETRIEVAL



http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html

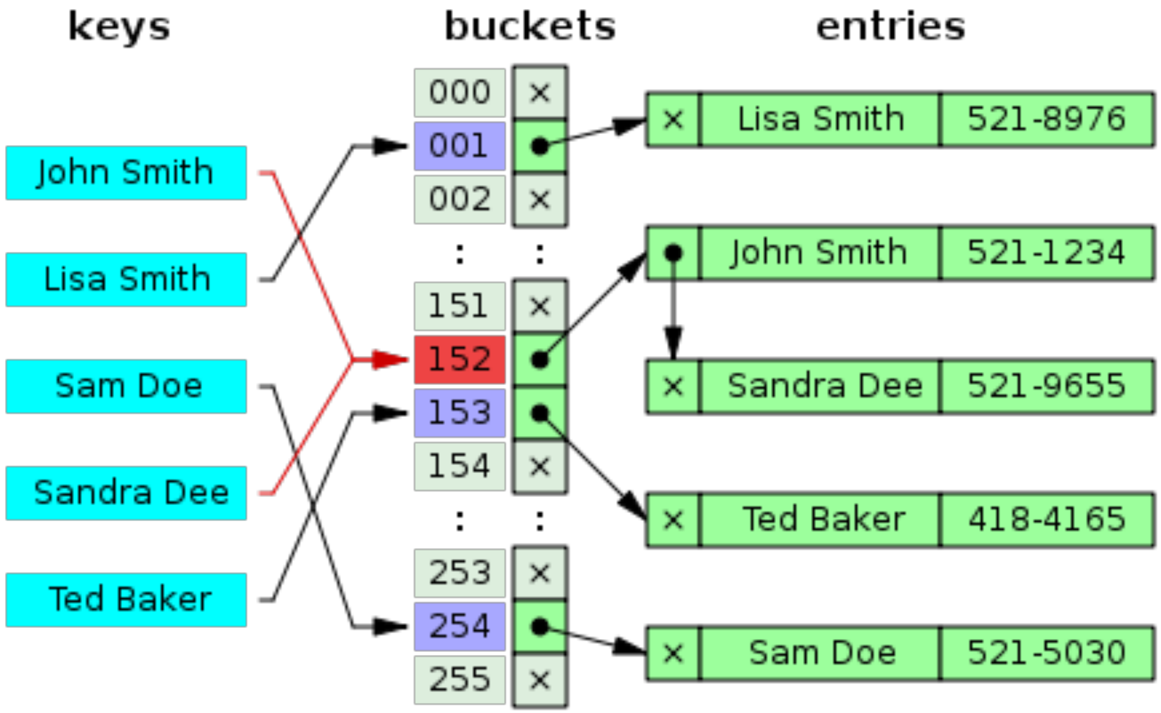
LINEAR RETRIEVAL

Algorithm:

- **Calculate the hash code for the given search key**
- **Access the hash element**
- **If the hash element is empty, the search has immediately failed.**
- **Otherwise, check for a match between the search and data key**
  - If there is a match, return the data.
  - If there is no match, probe the table until either:
    - A match is found between the search and data key
    - A completely empty hash element is found.

- ***** assumes complex data and that key data is stored and can be verified.**

LINEAR RETRIEVAL



keys

John Smith

Lisa Smith

Sam Doe

Sandra Dee

Ted Baker

buckets

000 ×
001 ●
002 ×
: :
151 ×
152 ●
153 ●
154 ×
: :
253 ×
254 ●
255 ×

entries

× Lisa Smith 521-8976

● John Smith 521-1234

× Sandra Dee 521-9655

× Ted Baker 418-4165

× Sam Doe 521-5030

QUADRATIC PROBING

To avoid primary clustering quadratic probing is often used?

Linear probing:

$$H + 1, H + 2, H + 3, H + 4, ..., H + k$$

Quadratic probing:

$$H + 1^2, H + 2^2, H + 3^2, H + 4^2, ..., H + k^2$$

REHASHING

- **Hash tables can become saturated resulting in frequent collisions and longer searches.**
- **One solution is to “rehash”**
    - Create a new hash table approx. 2x the size (often choosing the next prime number close to 2x)
    - Take all the values from the old table and rehash then into the new table.