13-0: **Searching & Selecting**

- Maintian a Database (keys and associated data)
- 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

13-1: **Sorted List Implementation**
If database is implemented as a sorted list:

- **Add**
- **Remove**
- **Find**

13-2: **Sorted List Implementation**
If database is implemented as a sorted list:

- **Add** $O(n)$
- **Remove** $O(n)$
- **Find** $O(\lg n)$

13-3: **BST Implementation**
If database is implemented as a Binary Search Tree:

- **Add**
- **Remove**
- **Find**

13-4: **BST Implementation**
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

13-5: **Unsorted List**
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?

13-6: **Hash Function**

- 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

13-7: **Hash Function**

- The "magic function" is called a *Hash function*
- If `hash(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?

13-8: **Hash Function**

- The "magic function" is called a *Hash function*
- If `hash(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?
  - Too many possible keys
  - If keys are strings of up to 15 letters, there are 10^{21} different keys
  - 1 sextillion – roughly the total number of transistors that have ever been produced.

13-9: **Integer Hash Function**

- 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 $1 \ldots n$
  - $n << m$

13-10: **Integer Hash Function**

- 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 $1 \ldots n$
  - $n << m$
- $\texttt{hash}(k) = k \texttt{ mod } n$

13-11: **Integer Hash Function**

- What if table size = 10, all keys end in 0?

13-12: **Integer Hash Function**

- What if table size = 10, all keys end in 0?
- What if table size is even, all keys are even?

13-13: **Integer Hash Function**

- 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?

13-14: **Integer Hash Function**

- 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?

13-15: **Integer Hash Function**

- 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.

13-16: **Integer Hash Function**

- 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*.

13-17: **String Hash Function**

- 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?

13-18: **String Hash Function**

- 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

```
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;
}
```

13-19: **String Hash Function**

- 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

$$\sum_{k=0}^{keysize-1} key[k] * 256^{keysize-k-1}$$

13-20: **String Hash Function**

- 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)

```
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;
}
```

13-21: **ElfHash**

- 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.

13-22: **ElfHash**

```
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;
}
```

13-23: **Collisions**

- 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)

13-24: **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

13-25: **Open Hashing**

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

13-26: **Open Hashing**

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 $\Theta(1)$

13-27: **Closed Hashing**

- 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)

13-28: **Closed Hashing**

- To add element X to a closed hash table:
  - Find the smallest i, such that Array[hash(x) + f(i)] is empty (wrap around if necessary)
  - Add X to Array[hash(x) + f(i)]
  - If f(i) = i, linear probing

13-29: **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.

13-30: **Closed Hashing**

- Quadradic 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

13-31: **Closed Hashing**

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

13-32: **Closed Hashing**

- Quadradic 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
- 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)

13-33: **Closed Hashing**

- Pseudo-Random
  - Create a “Permutation Array” P
  - f(i) = P[i]

13-34: **Closed Hashing**

- 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 * hash2(key)

13-35: **Deletion**

- Deletion from an open hash table is easy.
  - Find the element.
  - Delete it.
- Deletion from a closed hash table is harder.
  - Why?

13-36: **Deletion**

- 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)

13-37: **Deletion**

- To insert an element:
  - Find the smallest i such that hash(x) + f(i) is either empty or deleted
- To find an element
  - Try all values of i (starting with 0) until either
    - Table[hash(x) + f(i)] = x
    - Table[hash(x) + f(i)] is empty (*not* deleted)

13-38: **Rehashing**

- 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

13-39: **Rehashing**

- When we creata 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!