# Week 6 Lecture 16

Theory

# What's in this lecture?

- Hash Tables: how they work

# Arrays and Trees

- Arrays are O(1) (constant time) for get(index) and set(index), thanks to random access memory
- find(value) in an unsorted array or list is O(n) since index is unknown
- find(value) in a sorted array or balanced binary search tree is O(lg n) because half the possibilities are discarded at each step
- Can we find a way to improve on O(lg n)?

# Hash Tables

- With arrays, we map an integer index to some value
- With Hash Tables, we use a *hash function* to map a key (usually a string) to an integer
- The integer is then mapped to a position in the array using the modulo (%) function
- Hash functions are chosen such that for most keys, the computed indices are randomly distributed around the array

# A Simple Hash Function

```
function simple_hash(key, table_size) {
  var hash = 0;

  for (var i = 0; i < key.length; i++) {
    hash = (hash * 31) + key.charCodeAt(i);
  }

  return Math.abs(hash) % table_size;
}
```

# Hash Functions

- Writing hash functions is an art and a science
- Goal is to find good hash functions that reduce *collisions* (2 keys mapping to same hash)
- Hash functions exist for many applications: simple string hashing, cryptography, distributed systems
- Always choose hashes that are already well-understood: jenkins, murmur 2, sha1, sha256...
- Better hash functions typically use more CPU

# Another Hash Function

```
function jenkins_hash(key, table_size) {
  var hash = 0;
  for (var i = 0; i < key.length; i++) {
    hash += key.charCodeAt(i);
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }
  hash += (hash << 3);
  hash ^= (hash >> 11);
  hash += (hash << 15);

  return Math.abs(hash) % table_size;
}
```

# Handling Collisions

- There is always a chance that 2 keys will hash to the same value: a *collision*
- In this case, we use linear chaining to handle the collision: linked list of entries
- Another strategy is open addressing, which we will not discuss here

# Hash Table

```
function HashEntry(key, value, next) {
  this.key = key;
  this.value = value;
  this.next = next;
}

function HashTable(capacity) {
  this.capacity = capacity;
  this.size = 0;
  this.entries = new Array();
}
```

# Hash Table Get

```
HashTable.prototype.get = function(key) {
  var hash = simple_hash(key, this.capacity);
  var found = this.entries[hash];
  for (; found != null; found = found.next) {
    if (found.key === key) {
      return found.value;
    }
  }
  return null;
}
```

# Hash Table Put

```
HashTable.prototype.put = function(key, val) {
  var hash = simple_hash(key, this.capacity);
  var found = this.entries[hash];

  for (; found != null; found = found.next) {
    if (found.key === key) {
      found.value = val;
      return;
    }
  }

  this.entries[hash] =
    new HashEntry(key, val, this.entries[hash]);
}
```

# Resizing

- In real-world applications, we can't just have fixed-size hash tables
- Typically, we define a *load factor* (such as 0.75) and an *expansion factor* to grow the table (such as double the size)
- When the hash table fills past the load factor, we re-hash all entries into a new array of new size = size * expansion factor

# Exercises

- Make the hash function configurable (first-class function)
- Implement remove(key), keys(), and values() functions (keys and values return an array of all keys and values respectively)
- Implement resizing & re-hashing when the table grows beyond a given load factor