
Middlebury

# CSCI 201: Data Structures

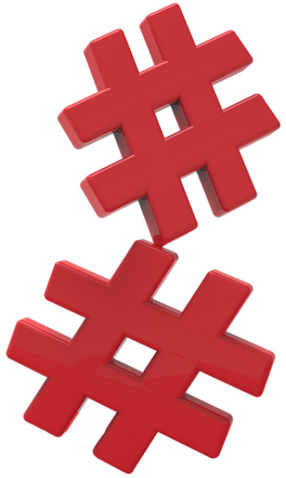
## Fall 2025

---

# Lecture 11T: Hash Tables (Part 2)

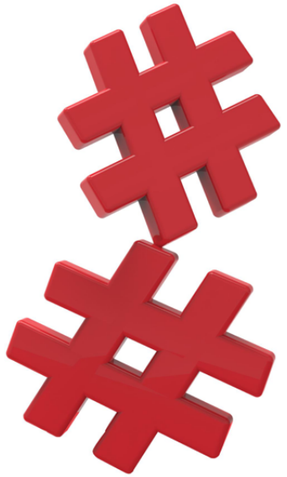
# Recap from last Tuesday:

- Trying to develop a container to save unordered set of keys (possibly associated with values).
- Would like $\mathcal{O}(1)$ runtime (average) for `add` (`put`), `get`, `contains` (`containsKey`), `remove`.
- Use a hash function to determine hash table (array) index.
- Requires that our items implement `hashCode` and `equals` methods.
- Hash functions should be quick to evaluate.
- Multiple keys might map to the same hash table index: **collision** :(



# Goals for today:

- Finish handling collisions in a hash table using linked lists to represent buckets (separate chaining).
- Handle collisions in a hash table using linear probing (open addressing).
- Do a bit (🙃) of review of bitwise operations: `|`, `&`, `^`, `>>>`, `<<`.
- Learn how hash table indices are computed for custom objects in `java`.



# If multiple keys map to the same table index, we have a collision.

## Option 1: Separate Chaining

- Each table item is a `LinkedList`.
- To `add`:
  1. Evaluate hash function and determine table index.
  2. Check if key already exists in the linked list at this index.
  3. Insert at the front (head) of the linked list.
- Searching for a `key` can be expensive if many items in one list.
  1. Evaluate hash function and determine table index.
  2. Loop through all list nodes and compare node `key` with query `key`.
- Resize table (and rehash) when load factor $\alpha = n/m$ is $> \alpha_{\max}$ (threshold). `Java` uses 0.75.

# Designing a hash map with separate chaining: inserting (`put`).

```
public class ChainedHashMap<K, V> {

  private static class Node<K, V> {
    public K key;
    public V value;
    public Node<K, V> next;

    public Node(K key, V value) {
      this.key = key;
      this.value = value;
    }
  }

  // array of buckets (each item will be a Node)
  private Object[] table;
  private int size; // current number of items

  ChainedHashMap(int initialCapacity) {
    table = new Object[initialCapacity];
  }

  private int getIndex(K key, int m) {
    int h = key.hashCode();
    return h % m;
  }

  public int capacity() {
    return table.length;
  }
}
```

```
@SuppressWarnings("unchecked")
public void put(K key, V value) {

  int index = getIndex(key, table.length);
  if (table[index] == null) {
    table[index] = new Node<K, V>(key, value);
  } else {
    // this is why we suppress the warning
    Node<K, V> head = (Node<K, V>) table[index];
    Node<K, V> node = head;
    while (node != null) {
      if (key.equals(node.key)) {
        node.value = value; // overwrite
        return; // no item added
      }
      node = node.next;
    }
    node = new Node<K, V>(key, value);
    node.next = head;
    table[index] = node;
  }
  size++; // we added an item!

  // TODO: check load factor and rehash
}

  private void rehash() {
    Object[] newTable = new Object[2 * table.length];
    // TODO: implement rehash
    table = newTable;
  }
```

# Designing a hash map with separate chaining: retrieving (`get`).

```java
public class ChainedHashMap<K, V> {

  private static class Node<K, V> {
    public K key;
    public V value;
    public Node<K, V> next;

    public Node(K key, V value) {
      this.key = key;
      this.value = value;
    }
  }

  // array of buckets (each item will be a Node)
  private Object[] table;
  private int size; // current number of items

  ChainedHashMap(int initialCapacity) {
    table = new Object[initialCapacity];
  }

  private int getIndex(K key, int m) {
    int h = key.hashCode();
    return h % m;
  }

  public int capacity() {
    return table.length;
  }
}
```

```java
public V get(K key) {
  int index = getIndex(key, table.length);
  if (table[index] == null) {
    return null; // empty bucket
  }
  // TODO: see if linked list has this key
  // and return the associated value
  return null;
}
```

```java
@SuppressWarnings("unchecked")
public V get(K key) {
  int index = getIndex(key, table.length);
  if (table[index] == null) {
    return null; // empty bucket
  }
  Node<K, V> node = (Node<K, V>) table[index];
  while (node != null) {
    if (key.equals(node.key)) {
      return node.value;
    }
    node = node.next;
  }
  return null;
}
```

# Implementing `rehash` when `alpha > 0.75`.

```java
public class ChainedHashMap<K, V> {

  private static class Node<K, V> {
    public K key;
    public V value;
    public Node<K, V> next;

    public Node(K key, V value) {
      this.key = key;
      this.value = value;
    }
  }

  // array of buckets (each item will be a Node)
  private Object[] table;
  private int size; // current number of items

  ChainedHashMap(int initialCapacity) {
    table = new Object[initialCapacity];
  }

  private int getIndex(K key, int m) {
    int h = key.hashCode();
    return h % m;
  }

  public int capacity() {
    return table.length;
  }
}
```

```java
@SuppressWarnings("unchecked")
private void rehash() {
  Object[] newTable = new Object[2 * table.length];
  size = 0;
  for (int i = 0; i < table.length; i++) {
    if (table[i] == null) {
      continue; // skip empty buckets
    }
    Node<K, V> node = (Node<K, V>) table[i];
    while (node != null) {
      // note that we have refactored the 'put'
      // method to use a helper for putting in
      // a desired array
      put(newTable, node.key, node.value);
      node = node.next;
    }
  }
  table = newTable;
}
```

**Exercise:** download the starter code for today. In `ChainedHashMap.java`, try writing the new refactored version of `put`.

# If multiple keys map to the same table index, we have a collision.

**Option 2: Open addressing** (linear probing).

- One key (or key-value pair) per item in the table.
- To add:
  1. Evaluate hash function and determine table index.
  2. If this index has a non-`null` item, keep iterating (offset from initial table index by $1, 2, 3, \ldots$) until we find an empty slot.
- Searching for a key:
  1. Evaluate hash function and determine table index.
  2. `while` key at this index is not the key we're looking for (and item at index is not `null`):
     - Go to next item offset from table index (using same offset $1, 2, 3, \ldots$).
- Resize table (and rehash) when load factor $\alpha = n/m$ is $> \alpha_{\max}$.
- Items can be *clustered*, alternative is to use *quadratic probing* (offset is $1, 4, 9, \ldots$).

**Exercise:** on your worksheet, insert the following keys into a hash table with initial capacity of 8. Use the division method to determine the table index. Double the capacity when $\alpha > 0.5$.

25, 1, 19, 34, 16, 27, 2, 9, 31.

## Discussion

What are the pros and cons of separate chaining vs. open addressing?

# Implementing a hash table for custom types.

```java
class Point {
  public int x;
  public int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  @Override
  public int hashCode() {
    // TODO: how should we hash **two** integer values?
  }

  @Override
  public boolean equals(Object otherObject) {
    Point otherPoint = (Point) otherObject;
    return (otherPoint.x == x) && (otherPoint.y == y);
  }
}
```

# Review of bitwise operations.

For more review, please see these CSCI 145 notes (representing data).

`0` is `false`, `1` is `true`

| operator | each bit is... | symbol | example (decimal) | example (binary) | result (binary) | result (decimal) |
|---|---|---|---|---|---|---|
| OR | 1 if at least 1 bit is 1 | `\|` | `3 \| 8` | `0011 \| 1000` | `1011` | `11` |
| AND | 1 if both bits are 1 | `&` | `9 & 14` | `1001 & 1110` | `1000` | `8` |
| XOR | 1 if only 1 bit is 1 | `^` | `9 ^ 12` | `1001 ^ 1100` | `0101` | `5` |
| right shift | shifted to the right by $n$ bits (division by $2^n$) | `>>>` `>>` | `15 >>> 2` | `1111 >>> 2` | `0011` | `3` |
| left shift | shifted to the left by $n$ bits (multiplication by $2^n$) | `<<` | `5 << 3` | `0101 << 3` | `00101000` | `40` |

**why?** fast!

**Exercise:** practice bitwise operations on your worksheet!

# A detailed look into how Java gets hash table indices (OpenJDK).

https://github.com/openjdk/jdk/blob/7b0f273e37625461baa333a3ef20fbbd93647243/src/java.base/share/classes/java/util/HashMap.java#L320

```java
/**
 * Computes key.hashCode() and spreads (XORs) higher bits of hash
 * to lower.  Because the table uses power-of-two masking, sets of
 * hashes that vary only in bits above the current mask will
 * always collide. (Among known examples are sets of Float keys
 * holding consecutive whole numbers in small tables.)  So we
 * apply a transform that spreads the impact of higher bits
 * downward. There is a tradeoff between speed, utility, and
 * quality of bit-spreading. Because many common sets of hashes
 * are already reasonably distributed (so don't benefit from
 * spreading), and because we use trees to handle large sets of
 * collisions in bins, we just XOR some shifted bits in the
 * cheapest possible way to reduce systematic lossage, as well as
 * to incorporate impact of the highest bits that would otherwise
 * never be used in index calculations because of table bounds.
 */
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

- Table (array) size is always a power of 2.
- Table index computed from `hash(key) & (m - 1)` where $m$ is the capacity (table length).
- `^` means bitwise XOR (exclusive OR): e.g. `01101 ^ 11001` is 10100
- `>>>` means unsigned right shift (here, by 16 bits): e.g. `01101010 >>> 4` is 0110
- `&` means bitwise AND: e.g. `00110 & 111` is 00110
- In your **Terminal**, type `jshell` and use `Integer.toBinaryString` to try these out! `Ctrl-D` to exit.

# What exactly is `hash(key) & (m - 1)` doing?

Remember that `Java` (`OpenJDK`) picks the table size to be a power of 2.

- Example: $m = 16$ is `10000`, so `m - 1` is

- What is `15 & 15`?

- What is `73 & 15`?

# Exercise: compute the hash table indices for the following `Point`s (using `Java`'s technique).

```
1 static final int hash(Object key) {
2     int h;
3     return (key == null) ?
4              0 : (h = key.hashCode()) ^ (h >>> 16);
5 }
```

- Starting with a table size `m = 16`.
- Table index = `hash(key) & (m - 1)`.
- 3 points on your worksheet

```
1  class Point {
2    public int x;
3    public int y;
4    public Point(int x, int y) {
5      this.x = x;
6      this.y = y;
7    }
8
9    @Override
10   public int hashCode() {
11     return 31 * x + y;
12   }
13
14   @Override
15   public boolean equals(Object otherObject) {
16     Point otherPoint = (Point) otherObject;
17     return (otherPoint.x == x) && (otherPoint.y == y);
18   }
19 }
```

## Discussion

How would you compute the hash code for a `String`?

# Additional notes:

- Today and tomorrow's office hours will feature a dog 🐕. If you want to come and are afraid or allergic, let me know!
- **Homework 8** due Thursday:
  - Part 1: use a `TreeMap` to solve a problem
  - Part 2: implement a DIY-version based on what your algorithm needs
  - Part 3: add self-balancing to your DIYTreeMap
- **Programming exam** grades and feedback are released
- **Midterm II** grades will be posted soon
- **Thursday:** graphs!