



Middlebury

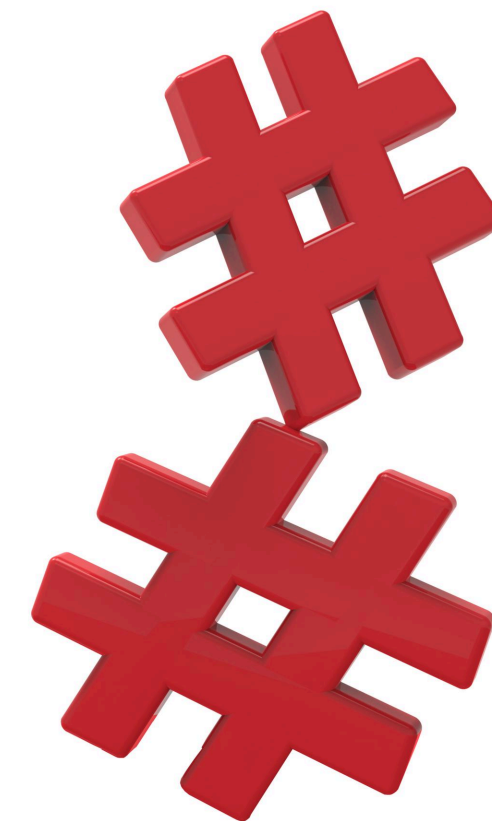
CSCI 201: Data Structures

Spring 2025

Lecture 10W: Hash Tables (Part 2)

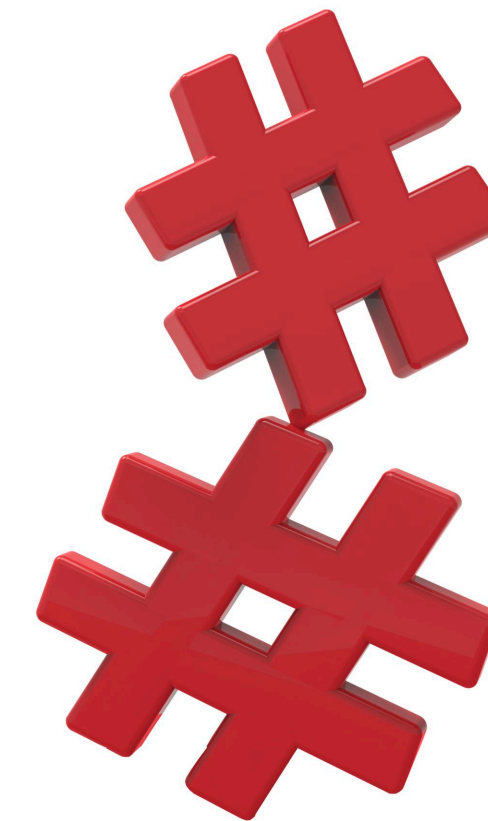
Recap from Monday:

- 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:

- Do a bit (🤔) of review of bitwise operations: `|`, `&`, `^`, `>>>`, `<<`.
- Handle collisions in a hash table using linked lists to represent buckets (separate chaining).
- Resize a hash table (and rehash items) based on the load factor.
- Handle collisions in a hash table using linear probing (open addressing).



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!

trick for modulus (%): if m is a power of 2, i.e. $m = 2^p$, then $x \% m$ is equal to $x \& (m - 1)$.

or

$$\begin{array}{r} 101 \\ 011 \\ \hline 111 \end{array}$$

and

$$\begin{array}{r} 101 \\ 011 \\ \hline 001 \end{array}$$

xor

$$\begin{array}{r} 101 \\ 011 \\ \hline 110 \end{array}$$

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**:
 - Evaluate hash function and determine table index.
 - Check if key already exists in the linked list at this index.
 - Insert at the front (head) of the linked list.
- Searching for a **key** can be expensive if many items in one list.
 - Evaluate hash function and determine table index.
 - 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.

Node

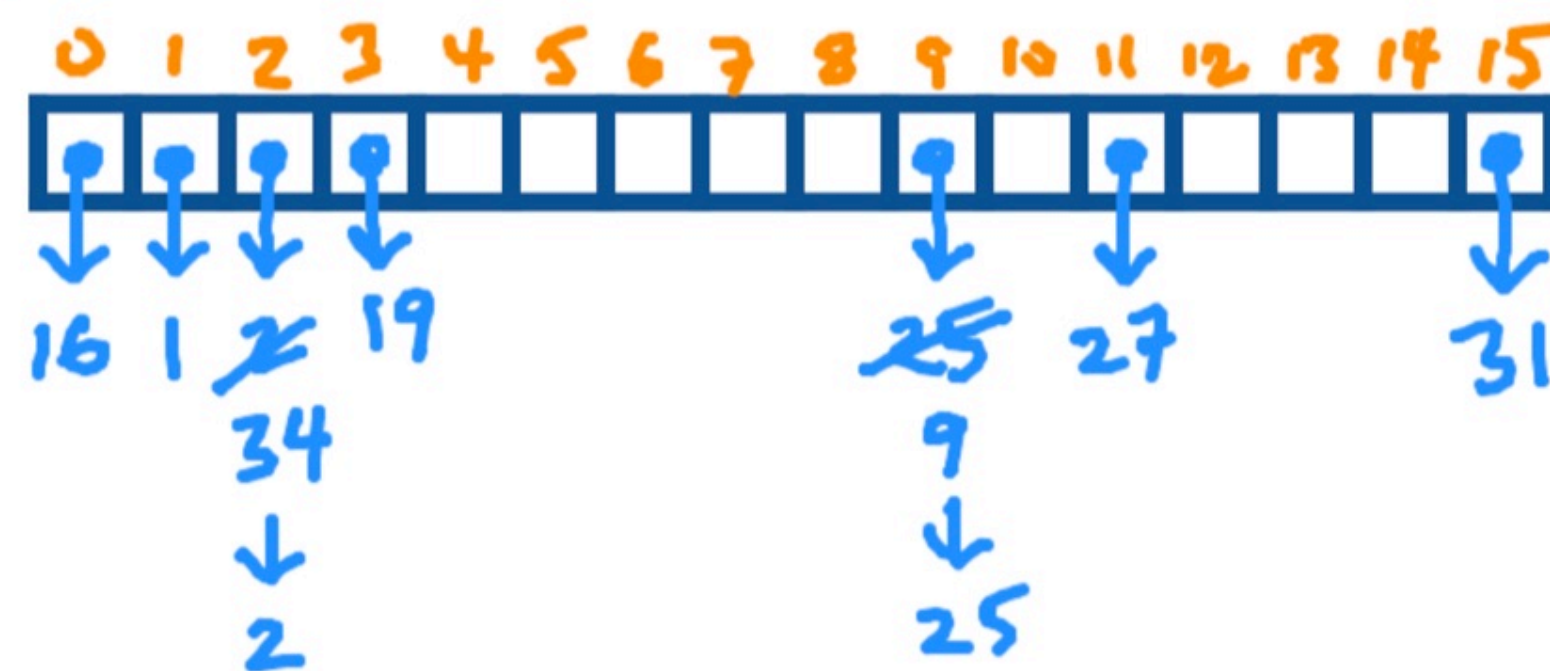
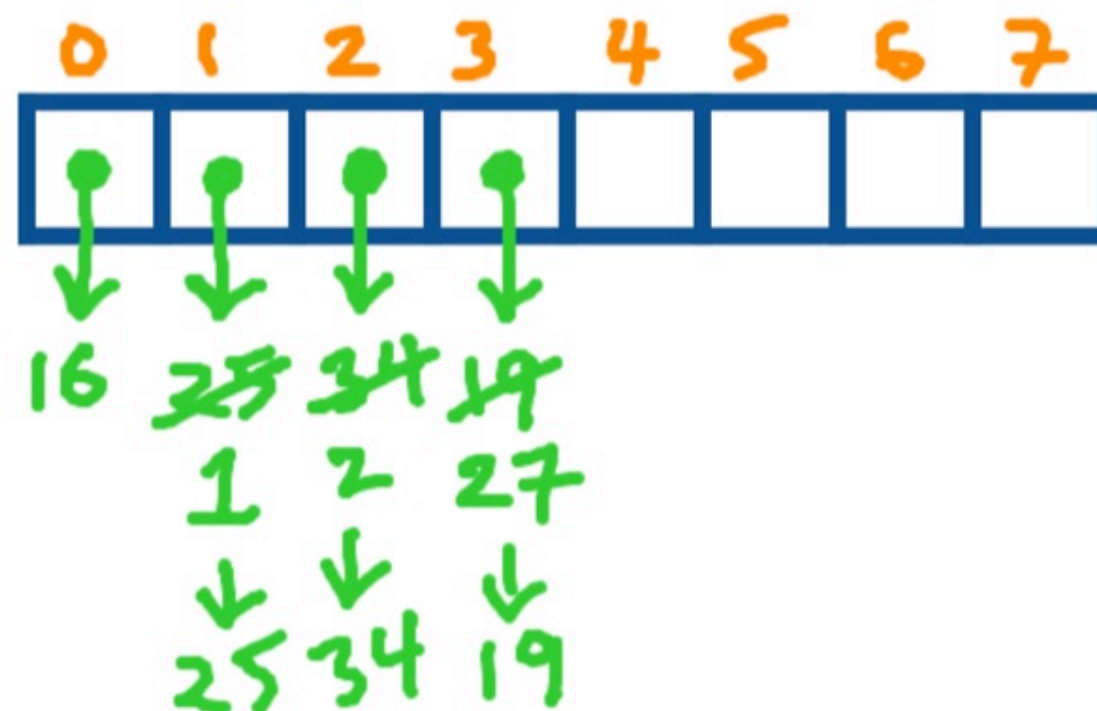


Exercise: insert the following keys into a hash table with initial capacity of 8. Use the division method to determine the table index. Double the capacity when $\alpha > 0.75$.

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

resize and rehash when $\alpha = \frac{7}{8} > .75$

% 8



% 16

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

```
1 public class ChainedHashMap<K, V> {
2
3     class Node {
4         public K key;
5         public V value;
6         public Node next;
7
8         public Node(K key, V value) {
9             this.key = key;
10            this.value = value;
11        }
12    }
13
14    // array of buckets (each item will be a Node)
15    private Object[] table;
16    private int size; // current number of items
17
18    ChainedHashMap(int initialCapacity) {
19        table = new Object[initialCapacity];
20    }
21
22    private int getIndex(K key, int m) {
23        int h = key.hashCode();
24        // uncomment to hash similarly to Java
25        // h = h ^ (h >>> 16);
26        return h % m;
27    }
28
29    public int capacity() {
30        return table.length;
31    }
32 }
```

```
1 @SuppressWarnings("unchecked")
2 public void put(K key, V value) {
3
4     int index = getIndex(key, table.length);
5     if (table[index] == null) {
6         table[index] = new Node(key, value);
7     } else {
8         // this is why we suppress the warning
9         Node head = (Node) table[index];
10        Node node = head;
11        while (node != null) {
12            if (key.equals(node.key)) {
13                node.value = value; // overwrite
14                return; // no item added
15            }
16            node = node.next;
17        }
18        node = new Node(key, value);
19        node.next = head;
20        table[index] = node;
21    }
22    size++; // we added an item!
23
24    // TODO check load factor and rehash
25 }
26
27 private void rehash() {
28     Object[] newTable = new Object[2 * table.length];
29     // TODO!
30     table = newTable;
31 }
```

Implementing **rehash** when **alpha > 0.75**.

```
1 public class ChainedHashMap<K, V> {
2
3     class Node {
4         public K key;
5         public V value;
6         public Node next;
7
8         public Node(K key, V value) {
9             this.key = key;
10            this.value = value;
11        }
12    }
13
14    // array of buckets (each item will be a Node)
15    private Object[] table;
16    private int size; // current number of items
17
18    ChainedHashMap(int initialCapacity) {
19        table = new Object[initialCapacity];
20    }
21
22    private int getIndex(K key, int m) {
23        int h = key.hashCode();
24        // uncomment to hash similarly to Java
25        // h = h ^ (h >>> 16);
26        return h % m;
27    }
28
29    public int capacity() {
30        return table.length;
31    }
32 }
```

```
1 @SuppressWarnings("unchecked")
2 private void rehash() {
3     Object[] newTable = new Object[2 * table.length];
4     size = 0;
5     for (int i = 0; i < table.length; i++) {
6         if (table[i] == null) {
7             continue; // skip empty buckets
8         }
9         Node node = (Node) table[i];
10        while (node != null) {
11            // note that we have refactored the 'put'
12            // method to use a helper for putting in
13            // a desired array
14            put(newTable, node.key, node.value);
15            node = node.next;
16        }
17    }
18    table = newTable;
19 }
```

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

```
1 public class ChainedHashMap<K, V> {
2
3     class Node {
4         public K key;
5         public V value;
6         public Node next;
7
8         public Node(K key, V value) {
9             this.key = key;
10            this.value = value;
11        }
12    }
13
14    // array of buckets (each item will be a Node)
15    private Object[] table;
16    private int size; // current number of items stored
17
18    ChainedHashMap(int initialCapacity) {
19        table = new Object[initialCapacity];
20    }
21
22    private int getIndex(K key, int m) {
23        int h = key.hashCode();
24        // uncomment to hash similarly to Java
25        // h = h ^ (h >>> 16);
26        return h % m;
27    }
28
29    public int capacity() {
30        return table.length;
31    }
32 }
```

```
1 public V get(K key) {
2     int index = getIndex(key, table.length);
3     if (table[index] == null) {
4         return null; // empty bucket
5     }
6     // TODO see if linked list has this key
7     // and return the associated value
8     return null;
9 }
```

```
1 @SuppressWarnings("unchecked")
2 public V get(K key) {
3     int index = getIndex(key, table.length);
4     if (table[index] == null) {
5         return null; // empty bucket
6     }
7     Node node = (Node) table[index];
8     while (node != null) {
9         if (key.equals(node.key)) {
10            return node.value;
11        }
12        node = node.next;
13    }
14    return null;
15 }
```

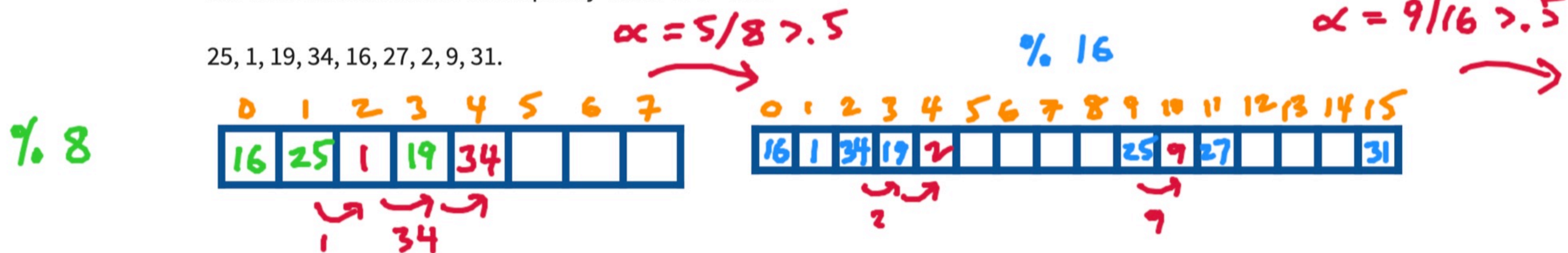

If multiple keys map to the same table index, we have a collision.

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, ...) 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, ...).
- 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, ...).

Exercise: 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.



Additional notes:

- **Homework 8** due tomorrow (Thursday 4/24):
 - 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
- **Lab 8** (Friday 4/25): Hash tables with linear probing
- **Next week:** graphs and a few graph algorithms

