Middlebury

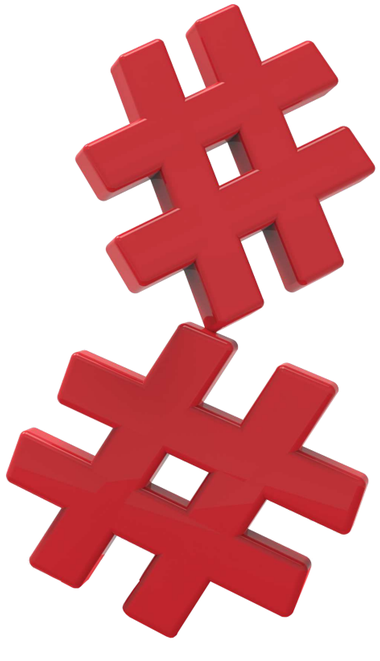
CSCI 201: Data Structures

Fall 2024

Lecture 10R: Hash Tables (Part 2)

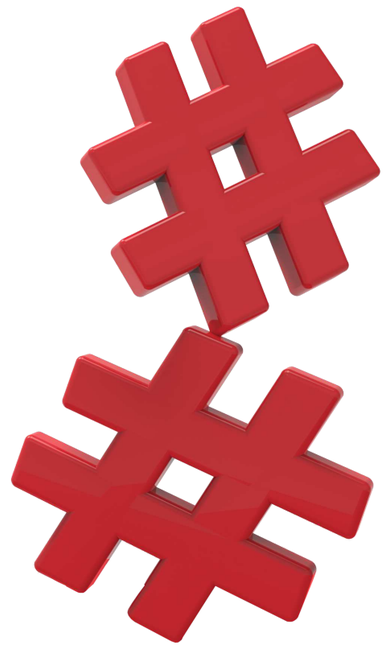
Recap from Tuesday:

- Trying to develop a container to save unordered set of keys (possibly associated with values).
- Would like $O(1)$ 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. $\texttt{m} = 2^p$, then `x % m` is equal to `x & (m - 1)`.

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)`.

e.g. x = 73 m = 16 x % m = 9

x in binary 1001001
m in binary 0010000
(m-1) in binary 0001111 — and
15
0001001 = 9

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.

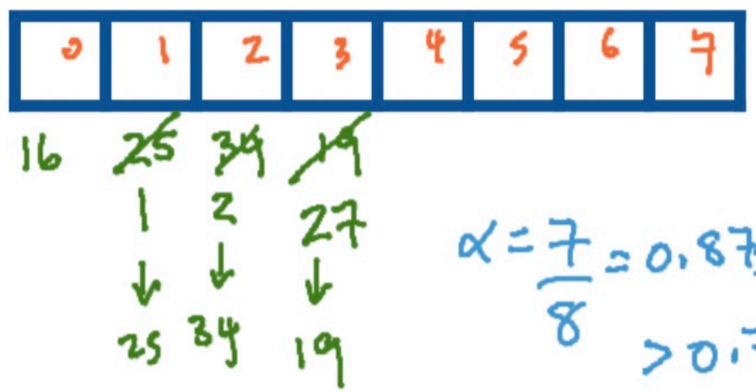
keys or key-value pairs

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 $\alpha > 0.75$.

% 8

1 1 3 2 0 3 2

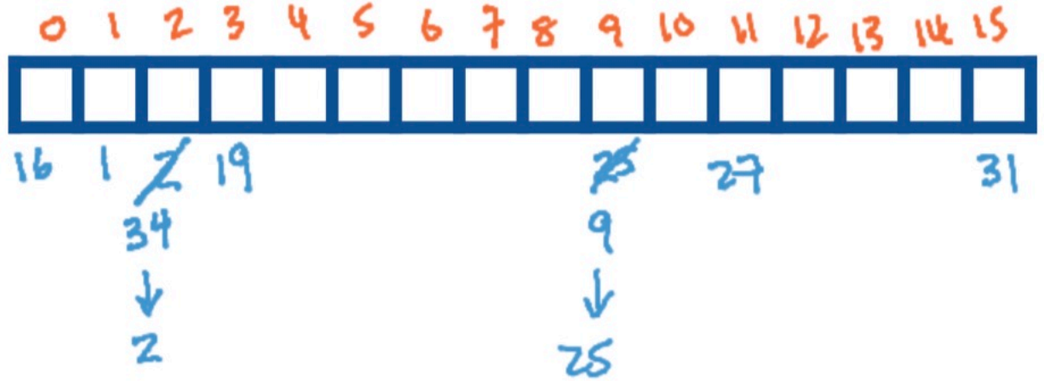
25, 1, 19, 34, 16, 27, 2, 9, 31.



$\alpha = \frac{7}{8} = 0.875 > 0.75$

resize and rehash items

m=16



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

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

  class Node {
    public K key;
    public V value;
    public Node 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();
    // uncomment to hash similarly to Java
    // h = h ^ (h >>> 16);
    return h % m;
  }

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

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

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

Implementing `rehash` when `alpha > 0.75`.

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

   class Node {
     public K key;
     public V value;
     public Node 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();
     // uncomment to hash similarly to Java
     // h = h ^ (h >>> 16);
     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 node = (Node) 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;
}
```

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

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

   class Node {
     public K key;
     public V value;
     public Node 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 stored

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

   private int getIndex(K key, int m) {
     int h = key.hashCode();
     // uncomment to hash similarly to Java
     // h = h ^ (h >>> 16);
     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 node = (Node) table[index];
  while (node != null) {
    if (key.equals(node.key)) {
      return node.value;
    }
    node = node.next;
  }
  return null;
}
```

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 $1, 4, 9, \ldots$).

→ h

→ take (h+i) % m

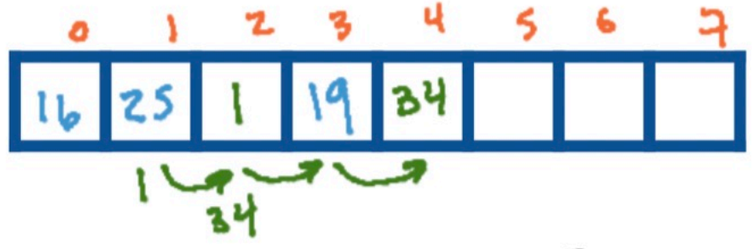
i

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 $\alpha > 0.5$.

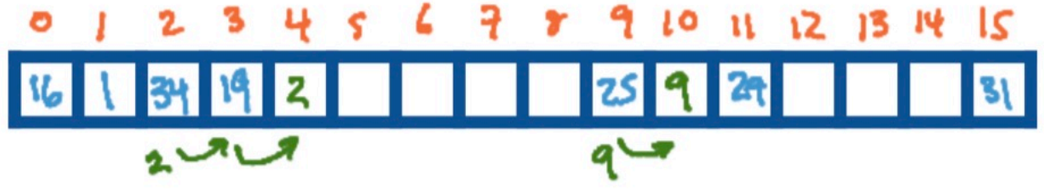
% 8

1 1 3 2 0

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



Stop $\alpha = \frac{5}{8} > 0.5$ rehash



$\alpha = \frac{9}{16} > 0.5$ so would need to resize, rehash again.

Additional notes:

- Complete Exit Ticket 10R by end of today.
- Rest of class (if time): work on Homework 8!
- **Homework 8** due on Friday 11/15: use a `TreeMap` to solve a problem and then implement a DIY-version based on what your algorithm needs.
- **Next week:** graphs and a few graph algorithms.