Chaining

```
import java.util.LinkedList;
import java.util.Objects;

/**
 * A simple generic hash table using separate chaining.
 */
public class HashChainingDemo {
    public static void main(String[] args) {
        // Create a hash table with initial capacity 5
        Hash<String, Integer> table = new Hash<>(5);

        // Insert some key–value pairs
        table.put("one", 1);
        table.put("two", 2);
        table.put("three", 3);
        table.put("four", 4);
        table.put("five", 5);

        // Access
        System.out.println("Get 'three': " + table.get("three"));
        System.out.println("Get 'ten' (absent): " + table.get("ten"));

        // Remove
        System.out.println("Remove 'two': " + table.remove("two"));
        System.out.println("Get 'two' after removal: " + table.get("two"));

        // Show contents
```

```
        table.display();
        System.out.println("Size: " + table.size());

        // Trigger resize by adding more elements
        table.put("six", 6);
        table.put("seven", 7);
        table.put("eight", 8);
        table.put("nine", 9);
        table.put("ten", 10);

        System.out.println("After adding more entries (resize may occur):");
        table.display();
        System.out.println("Size: " + table.size());
    }
}

interface HashI<K, V> {
    void put(K key, V value);
    V get(K key);
    V remove(K key);
    int size();
}

class Hash<K, V> implements HashI<K, V> {
    /** Node storing a key–value pair */
    private static class HashElement<K, V> implements Comparable<HashElement<K, V>> {
        K key;
        V value;
        HashElement(K key, V value) {
```

```
        this.key = key;
        this.value = value;
    }
    @SuppressWarnings("unchecked")
    public int compareTo(HashElement<K, V> o) {
        return ((Comparable<K>) this.key).compareTo(o.key);
    }
}

private LinkedList<HashElement<K, V>>[] table;
private int numElements;
private double maxLoadFactor;

/**
 * Constructs a Hash with given capacity and load factor.
 */
@SuppressWarnings("unchecked")
public Hash(int initialCapacity, double maxLoadFactor) {
    this.table = (LinkedList<HashElement<K, V>>[]) new LinkedList[initialCapacity];
    this.maxLoadFactor = maxLoadFactor;
    for (int i = 0; i < table.length; i++) {
        table[i] = new LinkedList<>();
    }
    this.numElements = 0;
}

/**
 * Default load factor = 0.75
 */
```

```
public Hash(int initialCapacity) {
   this(initialCapacity, 0.75);
}

/**
 * Returns the current number of entries.
 */
public int size() {
   return numElements;
}

/**
 * Display each bucket and its chain.
 */
public void display() {
   System.out.println("\nHash table contents:");
   for (int i = 0; i < table.length; i++) {
      System.out.print("bucket[" + i + "]: ");
      for (HashElement<K, V> e : table[i]) {
         System.out.print("(" + e.key + "→" + e.value + ") ");
      }
      System.out.println();
   }
}

/**
 * Computes a normalized index for a key.
 */
private int indexOf(K key) {
```

```
    int h = Objects.hashCode(key);
    h = h & 0x7fffffff;          // clear sign bit
    return h % table.length;
}

@Override
public void put(K key, V value) {
    int idx = indexOf(key);
    // Update if exists
    for (HashElement<K, V> e : table[idx]) {
        if (Objects.equals(e.key, key)) {
            e.value = value;
            return;
        }
    }
    // Otherwise insert new element
    table[idx].addFirst(new HashElement<>(key, value));
    numElements++;

    // Resize if load factor exceeded
    if ((double) numElements / table.length > maxLoadFactor) {
        resize();
    }
}

@Override
public V get(K key) {
    int idx = indexOf(key);
    for (HashElement<K, V> e : table[idx]) {
```

```
        if (Objects.equals(e.key, key)) {
            return e.value;
        }
    }
    return null;
}

@Override
public V remove(K key) {
    int idx = indexOf(key);
    var bucket = table[idx];
    for (HashElement<K, V> e : bucket) {
        if (Objects.equals(e.key, key)) {
            V val = e.value;
            bucket.remove(e);
            numElements--;
            return val;
        }
    }
    return null;
}

/**
 * Doubles the table size and rehashes all elements.
 */
@SuppressWarnings("unchecked")
private void resize() {
    var oldTable = table;
    table = (LinkedList<HashElement<K, V>>[]) new LinkedList[oldTable.length * 2];
```

```
        for (int i = 0; i < table.length; i++) {
            table[i] = new LinkedList<>();
        }
        numElements = 0;
        for (var bucket : oldTable) {
            for (var e : bucket) {
                // re-insert into new table
                put(e.key, e.value);
            }
        }
    }
}
```

Visualization

[harray] → [slot0] → HashElement(key,value) → HashElement → …

[slot1] → HashElement → …

[slot2] → (empty)

…

Output

```
Get 'three': 3
Get 'ten' (absent): null
Remove 'two': 2
Get 'two' after removal: null

Hash table contents:
bucket[0]:
```

bucket[1]:

bucket[2]: (one→1)

bucket[3]:

bucket[4]: (four→4)

bucket[5]:

bucket[6]: (five→5) (three→3)

bucket[7]:

bucket[8]:

bucket[9]:

Size: 4

After adding more entries (resize may occur):

Hash table contents:

bucket[0]:

bucket[1]:

bucket[2]: (one→1)

bucket[3]:

bucket[4]:

bucket[5]: (seven→7)

bucket[6]: (three→3) (five→5) (nine→9)

bucket[7]:

bucket[8]:

bucket[9]:

bucket[10]: (six→6)

bucket[11]:

bucket[12]:

bucket[13]:

```
bucket[14]: (four→4)
bucket[15]:
bucket[16]:
bucket[17]: (ten→10)
bucket[18]:
bucket[19]: (eight→8)

Size: 9
```

HashExample.java

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.NoSuchElementException;

/**
 * A generic hash table implementation using separate chaining.
 */
interface HashI<K,V> {
   boolean add(K key, V value);
   V getValue(K key);
   boolean remove(K key);
   int size();
   Iterator<K> keyIterator();
}

public class HashExample<K extends Comparable<K>, V> implements HashI<K, V>, Iterable<K> {
   // ---------- Inner Node Class ----------
   private static class HashElement<K,V> implements Comparable<HashElement<K,V>> {
```

```
    K key;
    V value;
    public HashElement(K key, V value) {
        this.key   = key;
        this.value = value;
    }
    @Override
    public int compareTo(HashElement<K,V> h) {
        // Allows ordering/searching by key if needed
        return ((Comparable<K>) h.key).compareTo(this.key);
    }
}

// ---------- Fields ----------
private LinkedList<HashElement<K,V>>[] hash_array;
private int tableSize;
private int numElements;
private double maxLoadFactor;

// ---------- Constructor ----------
@SuppressWarnings("unchecked")
public HashExample(int initialTableSize) {
    this.tableSize     = initialTableSize;
    this.maxLoadFactor = 0.75;
    this.numElements   = 0;
    // Create the array of empty chains
    hash_array = (LinkedList<HashElement<K,V>>[]) new LinkedList[tableSize];
    for (int i = 0; i < tableSize; i++) {
        hash_array[i] = new LinkedList<>();
```

```
    }
}

// ---------- Utility Methods ----------
private double loadFactor() {
    return (double) numElements / tableSize;
}

@SuppressWarnings("unchecked")
private void resize(int newTableSize) {
    LinkedList<HashElement<K,V>>[] oldArray = hash_array;
    hash_array   = (LinkedList<HashElement<K,V>>[]) new LinkedList[newTableSize];
    tableSize    = newTableSize;
    numElements  = 0;
    for (int i = 0; i < tableSize; i++) {
        hash_array[i] = new LinkedList<>();
    }
    // Rehash all existing elements
    for (var bucket : oldArray) {
        for (var he : bucket) {
            add(he.key, he.value);
        }
    }
}

private int indexOf(K key) {
    int h = (key == null ? 0 : key.hashCode());
    h = h & 0x7fffffff;              // clear sign bit → non-negative
    return h % tableSize;
```

```
}

// ---------- Core Operations ----------
@Override
public boolean add(K key, V value) {
   // Resize if load factor too high
   if (loadFactor() > maxLoadFactor) {
      resize(tableSize * 2);
   }
   int idx = indexOf(key);
   // Insert at head of chain
   hash_array[idx].addFirst(new HashElement<>(key, value));
   numElements++;
   return true;
}

@Override
public V getValue(K key) {
   int idx = indexOf(key);
   for (HashElement<K,V> he : hash_array[idx]) {
      if (key.equals(he.key)) {
         return he.value;
      }
   }
   return null;
}

@Override
public boolean remove(K key) {
```

```
    int idx = indexOf(key);
    var bucket = hash_array[idx];
    for (HashElement<K,V> he : bucket) {
        if (key.equals(he.key)) {
            bucket.remove(he);
            numElements--;
            return true;
        }
    }
    return false;
}

@Override
public int size() {
    return numElements;
}

// ---------- Iterator Implementation ----------
private class IteratorHelper implements Iterator<K> {
    private K[] keys;
    private int pos = 0;

    @SuppressWarnings("unchecked")
    public IteratorHelper() {
        keys = (K[]) new Object[numElements];
        int p = 0;
        for (var bucket : hash_array) {
            for (var he : bucket) {
                keys[p++] = he.key;
```

```
            }
        }
    }

    @Override
    public boolean hasNext() {
        return pos < keys.length;
    }

    @Override
    public K next() {
        if (!hasNext()) throw new NoSuchElementException();
        return keys[pos++];
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
}

@Override
public Iterator<K> keyIterator() {
    return new IteratorHelper();
}

@Override
public Iterator<K> iterator() {
    return keyIterator();
```

```
    }

    // -------- Demonstration Main Method --------
    public static void main(String[] args) {
        HashExample<String,Integer> ht = new HashExample<>(5);
        ht.add("one",   1);
        ht.add("two",   2);
        ht.add("three", 3);
        ht.add("four",  4);
        ht.add("five",  5);
        System.out.println("Get 'three': " + ht.getValue("three"));
        System.out.println("Remove 'two': " + ht.remove("two"));
        System.out.println("Get 'two': " + ht.getValue("two"));
        System.out.println("All keys in table:");
        for (String k : ht) {
            System.out.println("  " + k + " → " + ht.getValue(k));
        }
    }
}
```

Output

```
Get 'three': 3
Remove 'two': true
Get 'two': null

All keys in table:
  one → 1
  four → 4
  five → 5
```

three → 3

HashCode Example

```
import java.util.Objects;


public class Person {
   private String name;
   private int age;


   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }


   // Override equals(): two Persons are equal if their name and age are equal
   @Override
   public boolean equals(Object o) {
      if (this == o) return true;              // same reference
      if (o == null || getClass() != o.getClass()) return false;
      Person person = (Person) o;
      return age == person.age &&
            Objects.equals(name, person.name);
   }


   // Override hashCode() to be consistent with equals():
   @Override
   public int hashCode() {
      int result = 17;                     // non-zero arbitrary seed
```

```
        result = 31 * result + (name != null ? name.hashCode() : 0);
        result = 31 * result + age;
        return result;
    }

    // For testing
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 30);
        Person p2 = new Person("Alice", 30);
        Person p3 = new Person("Bob", 25);

        System.out.println("p1.equals(p2)? " + p1.equals(p2));        // true
        System.out.println("p1.hashCode() == p2.hashCode()? " +
                           (p1.hashCode() == p2.hashCode()));       // true

        System.out.println("p1.equals(p3)? " + p1.equals(p3));        // false
        System.out.println("p1.hashCode() == p3.hashCode()? " +
                           (p1.hashCode() == p3.hashCode()));       // likely false
    }
}
```