Hash Tables (v4)

Juan Manuel Gimeno Illa

2025/2026

Bibliography

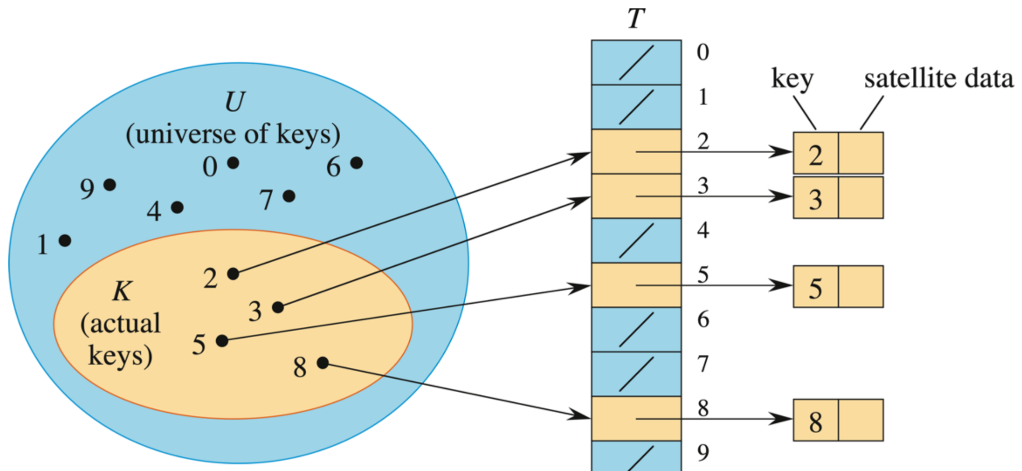
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein, Introduction to Algorithms – Fourth Edition, Massachusetts Institute of Technology (2022)
  - Chapter 11
- Robert Sedgewick & Kevon Wayne, Algorithms - Fourth Edition, Pearson Education Inc. (2011)
  - Section 3.4
- Josep Maria Ribó, Apropament a les estructures de dades des del programari lliure. Edicions de la Universitat de Lleida. 2018.
  - Chapter 5
- Maurice Naftalin & Philip Wadler with Stuart Marks, Java Generics and Collections (2nd edition), O'Reilly Media Inc. (2025)
  - Chapter 15

Hash Tables

- In the previous theme we've studied (binary) search trees that provide **efficient access to values given a key**
    - They need the key to be comparable (or a comparator for keys)
    - The costs are logarithmic, when the tree is balanced
- In this theme we'll present an alternative data structure to solve the same problem
    - That does not need to compare keys for ordering
    - But that needs a **function to map each key to an integer**
    - And, **given good properties** for the former function, it'll give **constant time costs** (with a **higher cost in space**)

Hash Tables

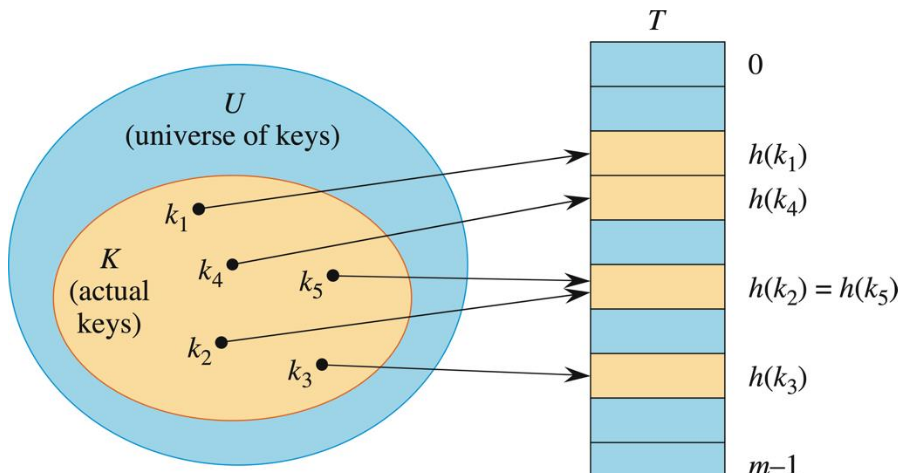
- Suppose that the universe of keys U is of limited size $|U|$
- And we have the guarantee that the **hash function** $h: U \to [0, |U|)$, that maps each key to the position of an array, is **injective**
    - i.e. no two keys get the same position
- Then we can implement that map using a simple array of size $|U|$



[Cormen et al. 11.2]

Hash Tables

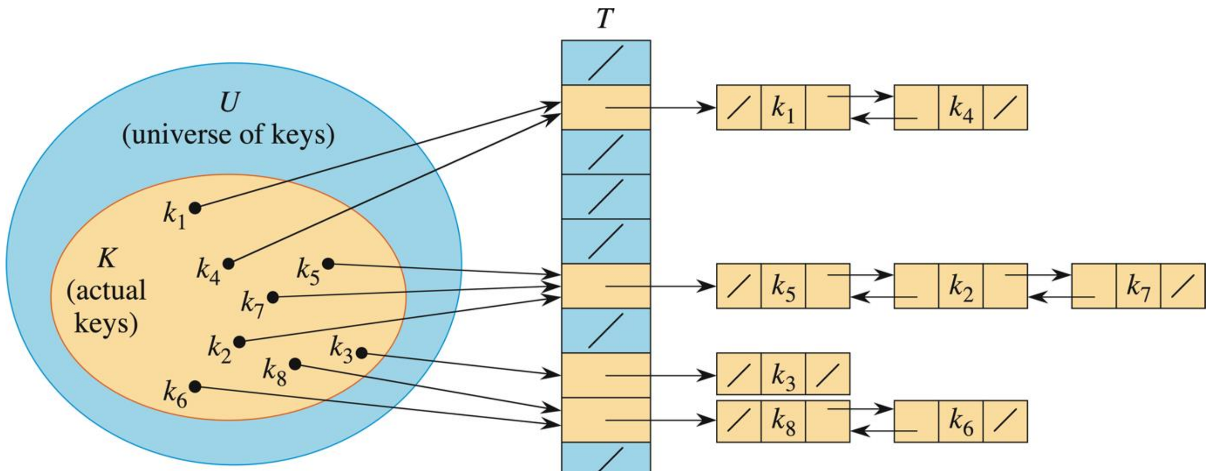
- What happens when we **lose injectivity** of the hash function?
    - E.g. because the codomain of the function is much smaller that its domain (allowing for smaller arrays)
    - That different keys get the same value
    - This is called a **collision**, i.e. different keys that collide in a value
- Different implementations of hash tables deal with collisions in distinct ways



[Cormen et al. 11.3]

Hash Tables

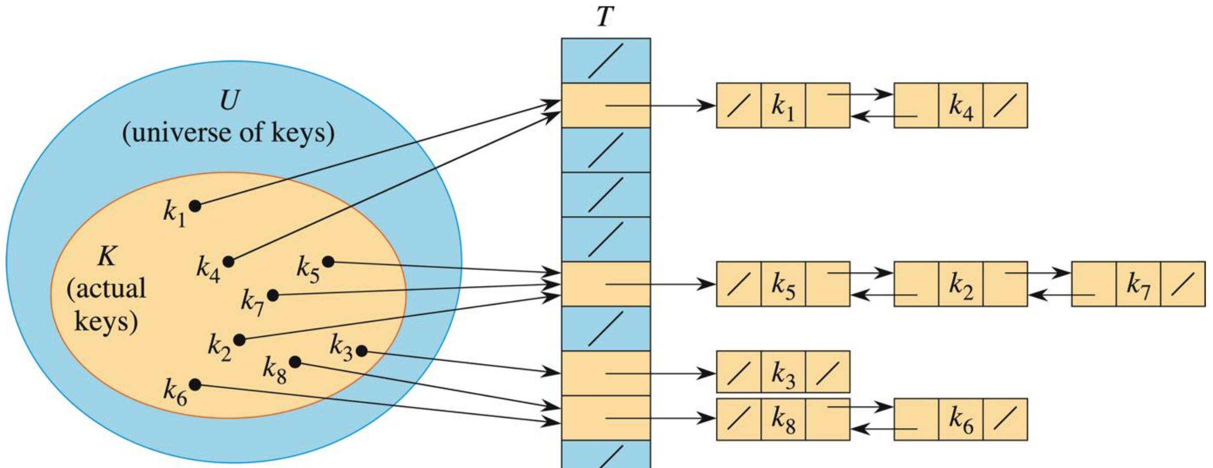
- The simplest (and most evident) way to deal with collisions is by **chaining**
  - Each position of the array points to a **list** (singly or doubly linked) **of synonyms**
- Reasoning about the costs of the operations involves
  - Properties of the hash function
  - Dimension of the array
- NOTE: As we've done with search trees most of the time, we'll show only the keys in the diagrams



[Cormen et al. 11.4]

Hash Tables

- The ideal hash function would give, for each possible key k, a value $h(k)$ that is a randomly and independently chosen value in the range $[0, m)$
  - That way, the expected length of the collision lists in a table of size n is $\frac{n}{m}$, which is the expected cost of finding a value given the key
  - So, if $m = \Omega(n)$, i.e. $n = O(m)$, the expected time is $\frac{n}{m} = \frac{O(m)}{m} = O(1)$



[Cormen et al. 11.4]

Hash Functions

- So, the **hash function** must return values **as they were randomly selected** but, at the same time, has to be **completely deterministic**
  - And **efficient** to compute (this is a different case than with cryptographic hashes)
- Usually, as it is the case in Java, the hash function is decomposed into two components:
  - The **hash code** function, $\boldsymbol{g}: \boldsymbol{U} \rightarrow \mathbb{Z}$
  - The **compression** function, $\boldsymbol{f}\colon \mathbb{Z} \rightarrow [\mathbf{0}, \boldsymbol{m})$
    - Usually, $\mathbf{f}(\mathbf{n}) = \mathbf{n}\ \boldsymbol{mod}\ \boldsymbol{m}$
  - And then $\boldsymbol{h} = \boldsymbol{f} \circ \boldsymbol{g}$

Hash Functions

- We will present some of the ideas behind the hash code functions used by Java
- As the main idea is to scatter the keys between all possible integer values, the computation must consider **all the bits** of the **key**:
  - For int value, the same value is used
    - For smaller integral types, we can do the same (well, characters has Unicode encodings into account)
  - For long value, (int) (value ^ (value >>> 32))
  - For floating point numbers, convert to bits and then hash the int or long
  - For booleans, value ? 1231 : 1237 (two enough large prime numbers)

Hash Functions

- We’ll consider the case for Strings (and the reasoning behind them will apply to the general case of computing the hash code for composite types)
- First idea: sum the hashes of the characters in the string
  - Problem1: for most strings (in ascii) very low values
  - Problem2: all permutations of the same characters, the same hash code (”east”, “seat”, “teas”, “eats”)
- To avoid these problems, **polynomial rolling** is used
  - A kind of polynomial
  - Computed by Horner’s rule (to not need exponentiation)

Hash Functions

- $str = “s_{l-1}s_{l-2}\cdots s_0” \rightarrow hash(str) = \sum_{i=0}^{l-1} hash(s_i) * p^i$
  - For some **prime** $\boldsymbol{p}$ (in Java, **31** is used)
  - NOTE: as we’ll see later, in Java the compression function uses a power of two as divisor, so the computation of the sum is not affected by overflow as the higher bits are not used
- This polynomial can be evaluated with this simple loop (not the real code):

```
int hash = 0;
for (int i = 0; i < s.length(); i++) {
   hash = 31 * hash + s.charAt(i);
}
return hash;
```

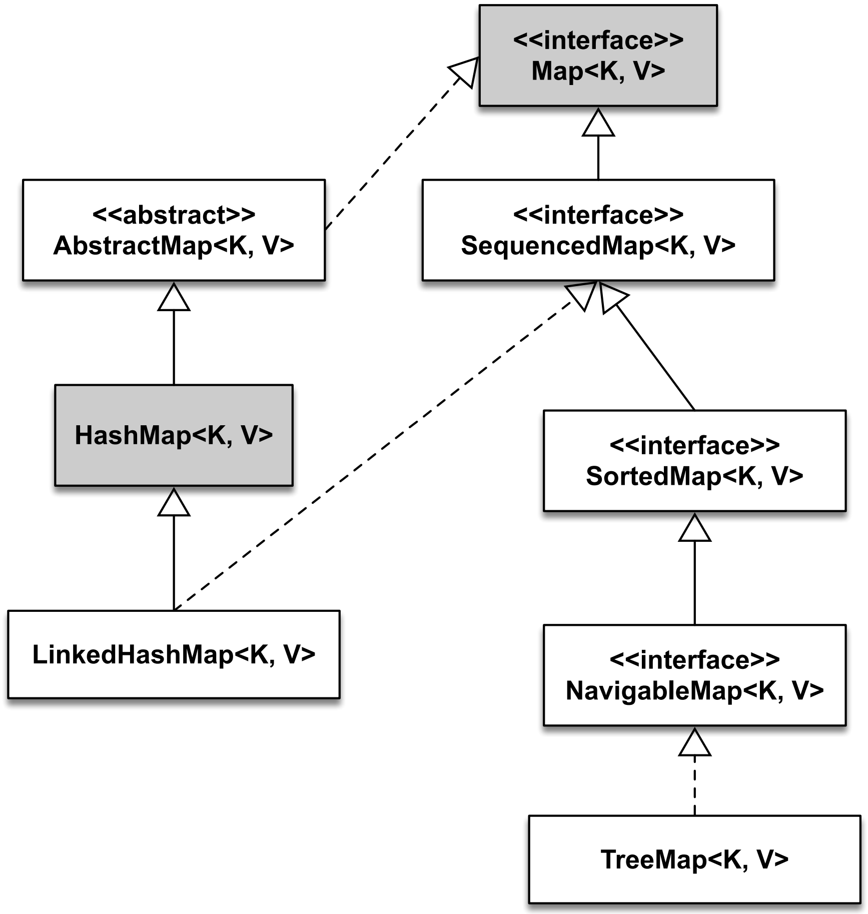
Hash Functions

- To compress the values returned by hashCode to an index in the interval $[0, table.length)$ to access the table of nodes we use %
  - But we must take care of the case when the integer returned by hashCode is negative
  - NOTE: The real implementation uses tables of size power of two to compute the % using a mask, as we'll see

```
public int hash(K key) {
   return (key.hashCode() & 0x7FFFFFFF) % table.length;
}
```

Maps in the Java Collections Framework

- There are many types related to Maps in the JFC
- In this theme, for sake of brevity, we'll concentrate on:
    - the interface Map<K, V>
    - the class HashMap<K, V>



<<interface>> Map<K, V>
<<abstract>> AbstractMap<K, V>
<<interface>> SequencedMap<K, V>
HashMap<K, V>
<<interface>> SortedMap<K, V>
LinkedHashMap<K, V>
<<interface>> NavigableMap<K, V>
TreeMap<K, V>

The Map<K, V> interface

- **V get(Object key)**
  - it is the fundamental operation on maps
  - returns value associated to key, or null when key not in map
  - if key is null throws NullPointerException if the map does not accept them
- **boolean containsKey(Object key)**
  - returns if the key has an association in the map
- **int size()**
  - returns the number of associations in the map
- **boolean isEmpty()**
  - returns whether the map is empty or not

```
public interface Map<K, V> {
   V get(Object key);
   boolean containsKey(Object key);
   V put(K key, V value);
   V remove(Object key);

   int size();
   boolean isEmpty();

   Set<K> keySet();
   Collection<V> values();
   Set<Map.Entry<K, V>> entrySet();

    interface Entry<K, V> {
      K getKey();
      V getValue();
      V setValue(V value);
    }

   // ...
}
```

The Map<K, V> interface

- **V put(K key, V value)**
  - Associates the key to the value
  - Returns the previous value associated to the key (or *null* if it didn't exist)
  - Throws NullPointerException if the key or the value are *null* and the map doesn't accept them
- **V remove(Object key)**
  - Removes the key from the map and returns current values (or *null* if it didn't exist)
  - If key is null throws NullPointerException if the map does not accept them

```
public interface Map<K, V> {
  V get(Object key);
  boolean containsKey(Object key);
  V put(K key, V value);
  V remove(Object key);

  int size();
  boolean isEmpty();

  Set<K> keySet();
  Collection<V> values();
  Set<Map.Entry<K, V>> entrySet();

   interface Entry<K, V> {
     K getKey();
     V getValue();
     V setValue(V value);
   }

  // ...
}
```

The Map<K, V> interface

- **Set<K> keySet()**
  - a view of the keys
- **Collection<V> values()**
  - a view on the values
- **Set<Map.Entry<K, V>> entrySet()**
  - a view on the pairs
- As the maps are not iterable, they can be used to iterate over keys, values or pairs
  - All of them return **views**, so they are backed by the map itself.

```
public interface Map<K, V> {
   V get(Object key);
   boolean containsKey(Object key);
   V put(K key, V value);
   V remove(Object key);

   int size();
   boolean isEmpty();

   Set<K> keySet();
   Collection<V> values();
   Set<Map.Entry<K, V>> entrySet();

    interface Entry<K, V> {
      K getKey();
      V getValue();
      V setValue(V value);
    }

   // ...
}
```

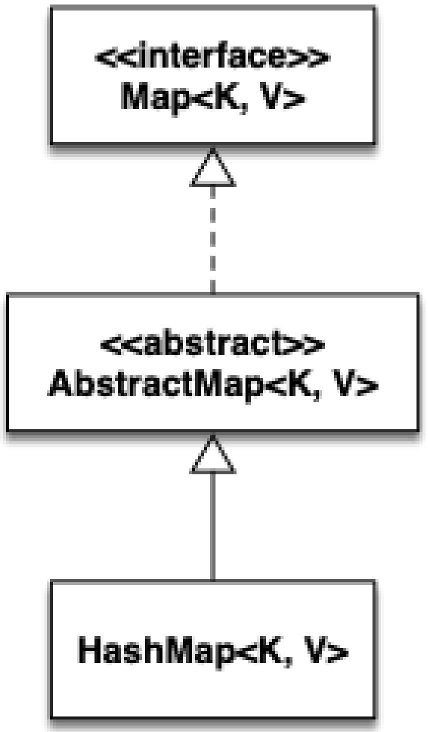
The Map<K, V> interface

- Map<K, V> does not define an order for the iterations on the views
- SequencedMap<K, V> there is a well-defined encounter order
- SortedMap<K, V> has a total ordering on the keys
- NavigableMap<K, V> with navigation methods (closest matches for keys)

```
void printAll(Map<K, V> map) {
  for (Map.Entry<K, V> entry: map.entrySet()) {
    System.out.printf(
        "%s -> %s",
        entry.getKey(),
        entry.getValue());
  }
}
```

Implementation on the JCF

- JCF provides different implementations of the Map<K, V> interface and an abstract class to facilitate them
- This abstract class, named AbstractMap<K, V>
  - Implements Map<K, V> without defining the concrete representation of the map
  - Some of the operation of the map interface are implemented in terms of iterators of key-value pairs
    - So, as you can imagine, their cost won't be acceptable for most uses of tables



```
public abstract class AbstractMap<K,V> implements Map<K,V> {
  // ...
  public V get(Object key) {
    Iterator<Entry<K,V>> i = entrySet().iterator();
    if (key==null) {
      while (i.hasNext()) {
        Entry<K,V> e = i.next();
        if (e.getKey()==null)
          return e.getValue();
      }
    } else {
      while (i.hasNext()) {
        Entry<K,V> e = i.next();
        if (key.equals(e.getKey()))
          return e.getValue();
      }
    }
    return null;
  }
  // ...
}
```

AbstractMap<K, V>

- Classes that use AbstractMap<K, V> to implement Map<K, V> must:
  - For unmodifiable maps:
    - Implement entrySet() (sometimes over AbstractSet<Map.Entry<K, V>>)
  - For modifiable maps:
    - put(K, V)
    - remove() for the iterators returned from entrySet()
  - And it would be convenient to redefine operations with unacceptable cost

HashMap<K, V>

- An implementation of the Map<K, V> interface
  - Uses **hash tables** and **collision** resolution by **chaining**
  - All optional operations are implemented
  - Allows null values and null keys
  - No guarantees on encountering order
- Two parameters that affect performance
  - **Capacity**: number of buckets
  - **Load Factor**: measure of fullness
- In terms of costs
  - **Given a ”good hash function”, get & put work in constant time**
  - Iteration on collection views proportional to capacity plus size

hashCode()

- In Java, any type can be used as a key in a HashMap<K, V>
- So, how does the HashMap<K, V> implementation know what function to invoke to compute the hash value?
  - It uses the hashCode() method that is defined in Object, so it exists for any class in Java
- As in the case of equals, **the contract of hashCode imposes some rules** in order to implement it correctly
  - If we do not follow this rules, the implementation of some classes in the JCF (or other libraries) may fail

hashCode()

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must return the same integer, provided no information used in equals comparisons on the object is modified.
  - This integer need not remain consistent from one execution of an application to another execution of the same application.
- **If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result**.
  - It is *not* required that if two objects are unequal according to the equals method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

hashCode()

```
@Override
public boolean equals(Object o) {
   if (this == o) return true;
   if (!(o instanceof Person other))
      return false;
   return this.age == other.age
         && Objects.equals(this.nif, other.nif)
         && Objects.equals(this.nme, other.name);
}
```

```
// Handmade combination
@Override
public int hashCode() {
   int result = Objects.hashCode(nif);
   result = 31 * result + Objects.hashCode(name);
   result = 31 * result + age;
   return result;
}
```

```
// Or by using Objects.hash
@Override
public int hashCode() {
   return Objects.hash(nif, name, age);
}
```

HashMap implementation in the JCF

```
public class HashMap<K,V> extends AbstractMap<K,V>
      implements Map<K,V>, Cloneable, Serializable {

  static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
  static final int MAXIMUM_CAPACITY = 1 << 30

  static final int TREEIFY_THRESHOLD = 8
  static final int UNTREEIFY_THRESHOLD = 6
  static final int MIN_TREEIFY_CAPACITY = 64

  transient Node<K,V>[] table;

  transient int size;
  transient int modCount;

  // ...
```

The capacity of the array of buckets will always be a power of 2

The implementation will convert the list of synonyms to a red-black tree (and vice-versa) depending on some thresholds

Number of key-values in the map

Array of buckets; its size is the capacity

```
public class HashMap<K,V> extends AbstractMap<K,V>
      implements Map<K,V>, Cloneable, Serializable {

  // ...

  static final float DEFAULT_LOAD_FACTOR = 0.75f;

  int threshold;
  final float loadFactor;

  public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
  }

  public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
  }

  // ...
```

Threshold = capacity * loadFactor

- Next value of size that will produce a re-dimensioning of the table

How much the table size is allowed to get before increasing the capacity (the default is 75%)

NOTE: Constructors do not create the table of buckets (**lazy initialization**)

```
public class HashMap<K,V> extends AbstractMap<K,V>
     implements Map<K,V>, Cloneable, Serializable {

  public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
      throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
      initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
      throw new IllegalArgumentException("Illegal load factor: " + loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
  }


  // smallest power of 2 >= cap
  static final int tableSizeFor(int cap) {
    int n = -1 >>> Integer.numberOfLeadingZeros(cap - 1);
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
  }
```

NOTE: Constructors do not create the table of buckets (lazy initialization)

Threshold = capacity * loadFactor
- Next value of size that will produce a re-dimensioning of the table
- Initially will be the smallest power of 2 >= the given capacity
- Why? Because the first put will dimension the table to it

```
static final int tableSizeFor(int cap) {
    int n = -1 >>> Integer.numberOfLeadingZeros(cap - 1);
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
 }
```

- **Calculate the highest one bit position**:
  - Decreasing cap ensures that if is already a power of two, it doesn't get incremented to the next power of two.
  - Counts the number of leading zeros in the binary representation of cap - 1.
  - Shifts -1 (all bits set to 1) right by the number of leading zeros, effectively creating a bitmask with the highest bit set to 0 and all lower bits set to 1.
- **Determine the next power of two**:
  - If n is less than 0, it means cap was 0, so the method returns 1.
  - If n is greater than or equal to MAXIMUM_CAPACITY, it returns MAXIMUM_CAPACITY to prevent overflow.
  - Otherwise, it returns n + 1, which is the next power of two greater than or equal to cap.

```
public class HashMap<K,V> extends AbstractMap<K,V>
        implements Map<K,V>, Cloneable, Serializable {

  static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(
          int hash,
          K key,
          V value,
          Node<K,V> next) {

       this.hash = hash;
       this.key = key;
       this.value = value;
       this.next = next;
    }
```

- Implementation using open addressing
- It will be a table of linked-lists of synonyms (at least initially)
- Each Entry will be a node in this list
- The hash value is cached

```
public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

// ...
```

```
public class HashMap<K,V> extends AbstractMap<K,V>
        implements Map<K,V>, Cloneable, Serializable {

  static class Node<K,V> implements Map.Entry<K,V> {

    // ...

    public final K getKey()        { return key; }
    public final V getValue()      { return value; }
    public final String toString() { return key + "=" + value; }

    public final boolean equals(Object o) {
        if (o == this) return true;

        return o instanceof Map.Entry<?, ?> e
          && Objects.equals(key, e.getKey())
          && Objects.equals(value, e.getValue());
    }

    public final int hashCode() {
      return Objects.hashCode(key) ^ Objects.hashCode(value);
    }
  }
```

equals takes into account key and value

and so does hashCode

HashMap<K, V>

- get(key) & put(key, value) are the most important functions in the Map<K, V> interface
- their implementations use two auxiliar functions:
  - getNode
  - putVal

```
public class HashMap<K,V> extends AbstractMap<K,V>
        implements Map<K,V>, Cloneable, Serializable {

  public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(key)) == null ? null : e.value;
  }

  public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
  }

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

Spreads higher bits into lower to mitigate the effect of using only the lower bits to compute bucket (see getNode or putVal)

```
final Node<K,V> getNode(Object key) {
   Node<K,V>[] tab; Node<K,V> first, e; int n, hash; K k;
   if ((tab = table) != null && (n = tab.length) > 0 &&
      (first = tab[(n - 1) & (hash = hash(key))]) != null) {
      if (first.hash == hash && // always check first node
         ((k = first.key) == key || (key != null && key.equals(k))))
         return first;
      if ((e = first.next) != null) {
         if (first instanceof TreeNode)
            return ((TreeNode<K,V>)first).getTreeNode(hash, key);
         do {
            if (e.hash == hash &&
               ((k = e.key) == key || (key != null && key.equals(k))))
               return e;
         } while ((e = e.next) != null);
      }
   }
   return null;
}
```

Checks if the table exists (lazy initialization, see putVal)

Finds the bucked using the last bits of the hash of key (n is power of two so n-1 is mask for last bits and "&" computes "% n")

We check if it's in the first node of the bucket

If there are more nodes to check ...

If the synonyms are in the form of a tree, we search in this tree

If they form a linked list of nodes, we use a linear search

```
static class Entry<K,V> extends HashMap.Node<K,V> {
   Entry<K,V> before, after;
   Entry(int hash, K key, V value, Node<K,V> next) {
      super(hash, key, value, next);
   }
}
```

As the LinkedHashMap class is an extension of HashMap, and TreeNodes must work for both, the TreeNode class extends the LinkedHashMap.Entry class, an extension of HashMap.Node, to preserve the insertion order (needed for LinkedHashMap)

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
   TreeNode<K,V> parent;  // red-black tree links
   TreeNode<K,V> left;
   TreeNode<K,V> right;
   TreeNode<K,V> prev;    // needed to unlink next upon deletion
   boolean red;
   TreeNode(int hash, K key, V val, Node<K,V> next) {
      super(hash, key, val, next);
   }
  //...
}
```

Tree nodes form a red-black tree of synonyms (implemented using Cormen et al.)

```
/**
 * Implements Map.put and related methods.
 *
 * @param hash hash for key
 * @param key the key
 * @param value the value to put
 * @param onlyIfAbsent if true, don't change existing value
 * @param evict if false, the table is in creation mode.
 * @return previous value, or null if none
 */
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ... }
```

1. *Must create the array of buckets if not created yet*
2. *Must find the bucket associated with the value of the hash function*
3. *Must find whether this key exists in the bucket*
4. *Must consider whether the synonyms form a linked list of a red-back tree*
5. *Must migrate the linked-list to a red-black tree if the threshold is passed*
6. *Must consider a resizing of the array if the threshold is passed*

We only insert when there is no previous value

To indicate normal mode (needed for LinkedHashMap) or creation mode (e.g. when copying a HashMap)

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {
   Node<K,V>[] tab; Node<K,V> p; int n, i;
   if ((tab = table) == null || (n = tab.length) == 0)
      n = (tab = resize()).length;
   if ((p = tab[i = (n - 1) & hash]) == null)
      tab[i] = newNode(hash, key, value, null);
   else {
      // Big else when the bucket is not empty
     // (next slide)
   }
  ++modCount;
  if (++size > threshold)
     resize();
  afterNodeInsertion(evict);
  return null;
}
```

Create the table if not exists

If bucket is empty, we insert as new node

resize if threshold passed

for LinkedHashMap actions

no previous value for key in the map

```
// Big else when the bucket is not empty

Node<K,V> e; K k;
if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
   e = p;
else if (p instanceof TreeNode)
   e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
else {
   // Big else when we have to search the bucket
   // (next slide)
}
if (e != null) { // existing mapping for key
   V oldValue = e.value;
   if (!onlyIfAbsent || oldValue == null)
      e.value = value;
   afterNodeAccess(e);
   return oldValue;
}
```

we find the key in the first node of the bucket

if the synonyms form a tree, we insert in the tree (when it does not exist) and return the previous node

if there was a previous value, we only modify it when onlyIfAbsent is false or when the previous value was null

for LinkedHashMap actions

```
// Big else when we have to search the bucket

for (int binCount = 0; ; ++binCount) {
  if ((e = p.next) == null) {
    p.next = newNode(hash, key, value, null);
    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
      treeifyBin(tab, hash);
    break;
  }
  if (e.hash == hash &&
    ((k = e.key) == key || (key != null && key.equals(k))))
    break;
  p = e;
}
```

If we get to the end, we create a new node

If we pass a threshold, we convert linked list into a red-back tree and we exit the loop

we have found the key, so we exit the loop

we continue the search with the next element in the list

HashMap<K, V>

- And all this is for only two of the operations on HashMap<K, V>
  - and some of the code has been omitted !!
- For instance, we have not analysed remove
  - which can be analysed using the same approach we have used in put
- As a conclusion:
  - A data-structure can be simple (e.g. the idea of a HashMap)
  - But real implementations, with highly optimized code, are very complex
  - But having access to these implementations (i.e. Free Software)
    - is a great source of knowledge
    - and reading code can be fun !!
    - a great place to use (**with caution**, because sometimes they hallucinate) LLMs