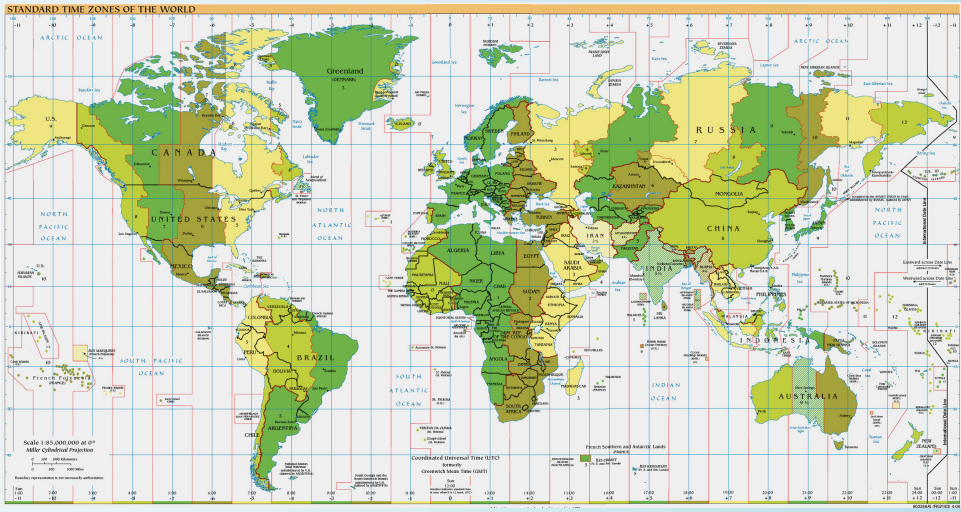
HashMap



Map and Set are most important data structures we use in development

There are multiple different maps and sets that are in Java

HashMap

Java: Map<A,B>

Map<String, Integer> hm = new HashMap<>();

Basic Operations

Key: Object; Value: Object

Return Type	Method
Value 🤔	put(key, value)
Value	get(key)
Value	remove(key)
boolean	containsKey(key)
boolean	containsValue(value)
Set<Map.Entry<Key, Value>>	entrySet()
Set<Key>	keySet()
Collection<Value>	values()
...	size(); isEmpty(); putAll(Map<>);

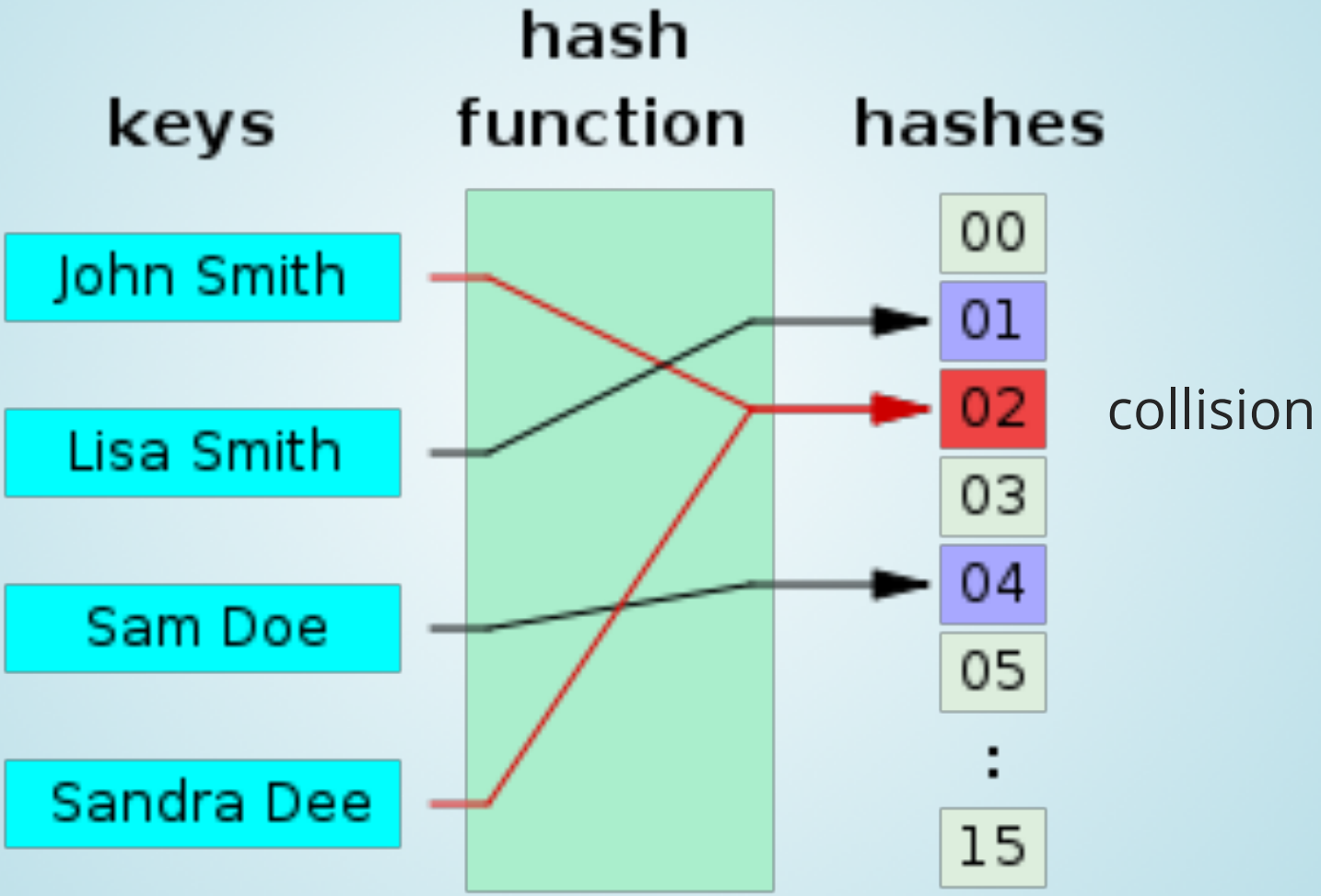
Traverse a HashMap

```
// Map -> Set -> Iterator -> Map.Entry -> troublesome, not recommend!
Iterator<Entry<String,String>> iterator = map.entrySet().iterator();
while (iterator.hasNext()) {
        Map.Entry<String,String> entry = (Map.Entry<String,String>) iterator.next();
        System.out.println("Key: " + entry.getKey() + " Value:" + entry.getValue());
}

// more elegant way, this should be the standard way, recommend!
for (Map.Entry<String, String> entry : map.entrySet()) {
        System.out.println("Key: " + entry.getKey() + " Value: " + entry.getValue());
}

// weired, but works anyway
for (Object key : map.keySet()) {
        System.out.println("Key: " + key.toString() + " Value: " + map.get(key));
}
```

Hashing

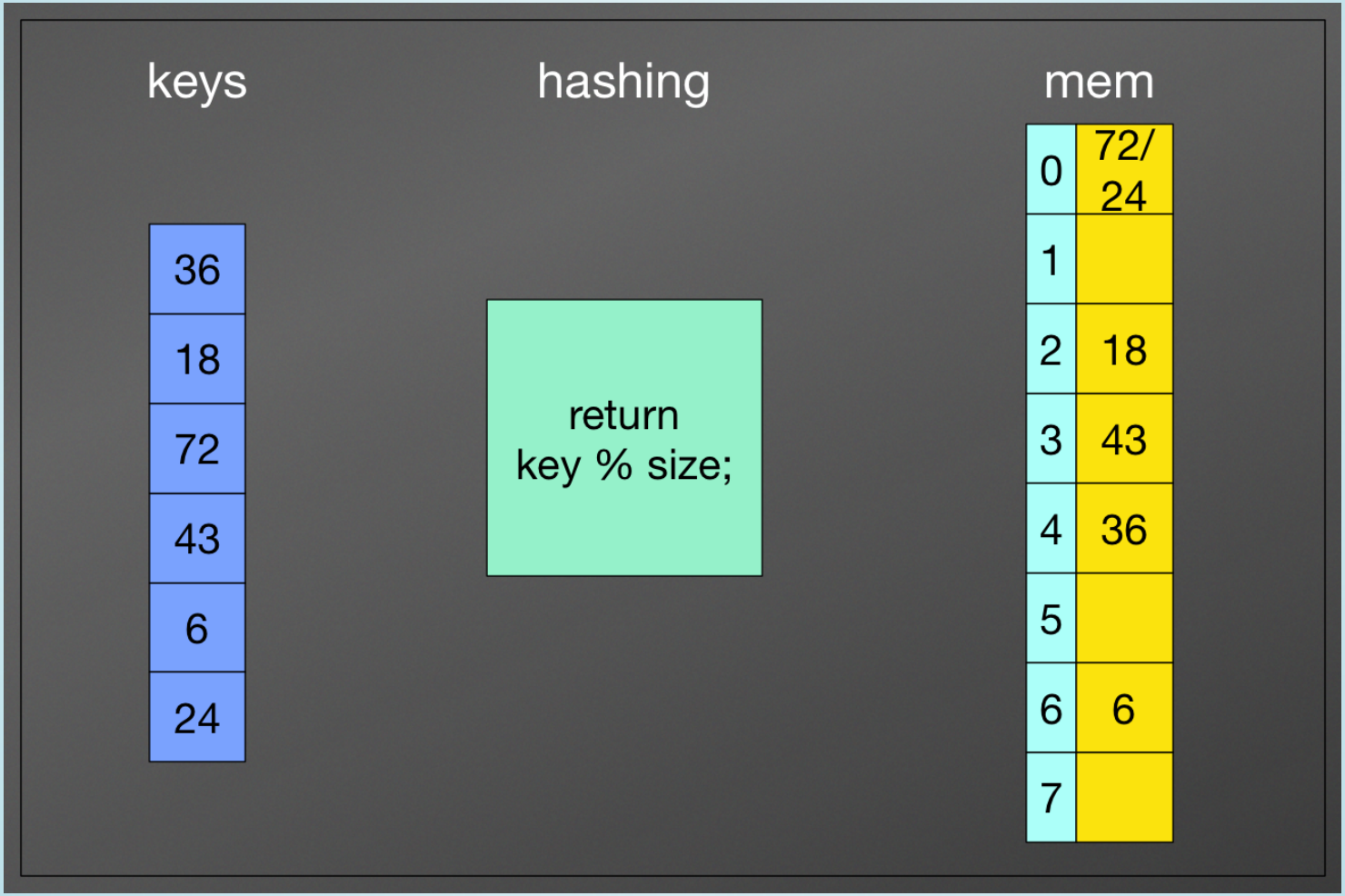


Hash Function

a function that can take a **key** and compute an **integer** (or an index in a table) for it

What really matters: How could we find a hash function that could fast compute the index without **collisions**

Collision



Collision

No matter which function, you are using, you have to deal with collision

reason that there is collision:

1. some keys just happen to map to the same index
2. keys > slots

Collision

How to solve them:

1. We cannot, just try to find a better algo
2. Open hashing
3. Closed hashing
4. Expand the space (Load Factor)

Load factor: size/capacity

Normally if (LF > 0.75), we double the space.

Collision

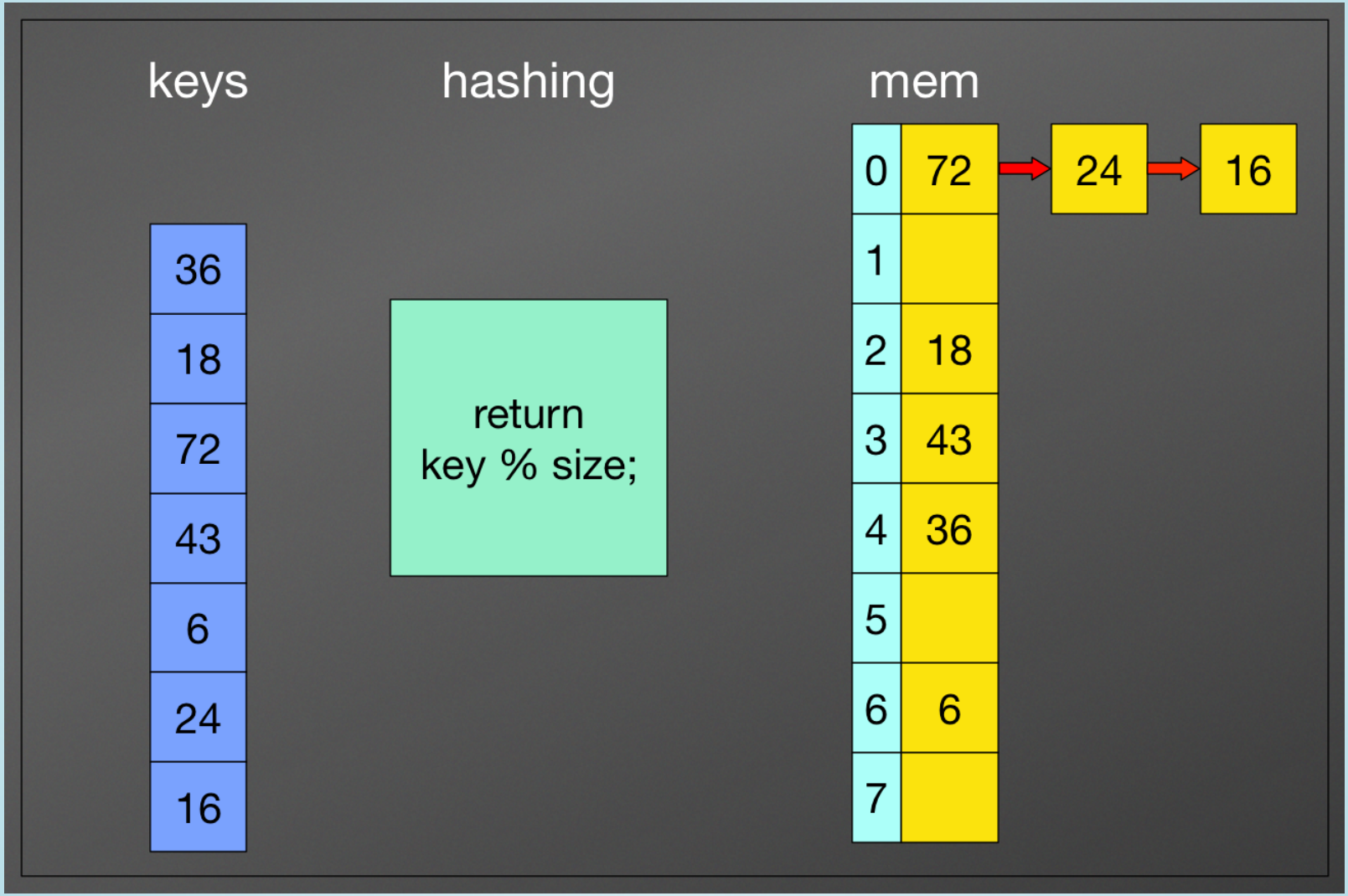
Open hashing:

for each collision, we use a linked list to store them

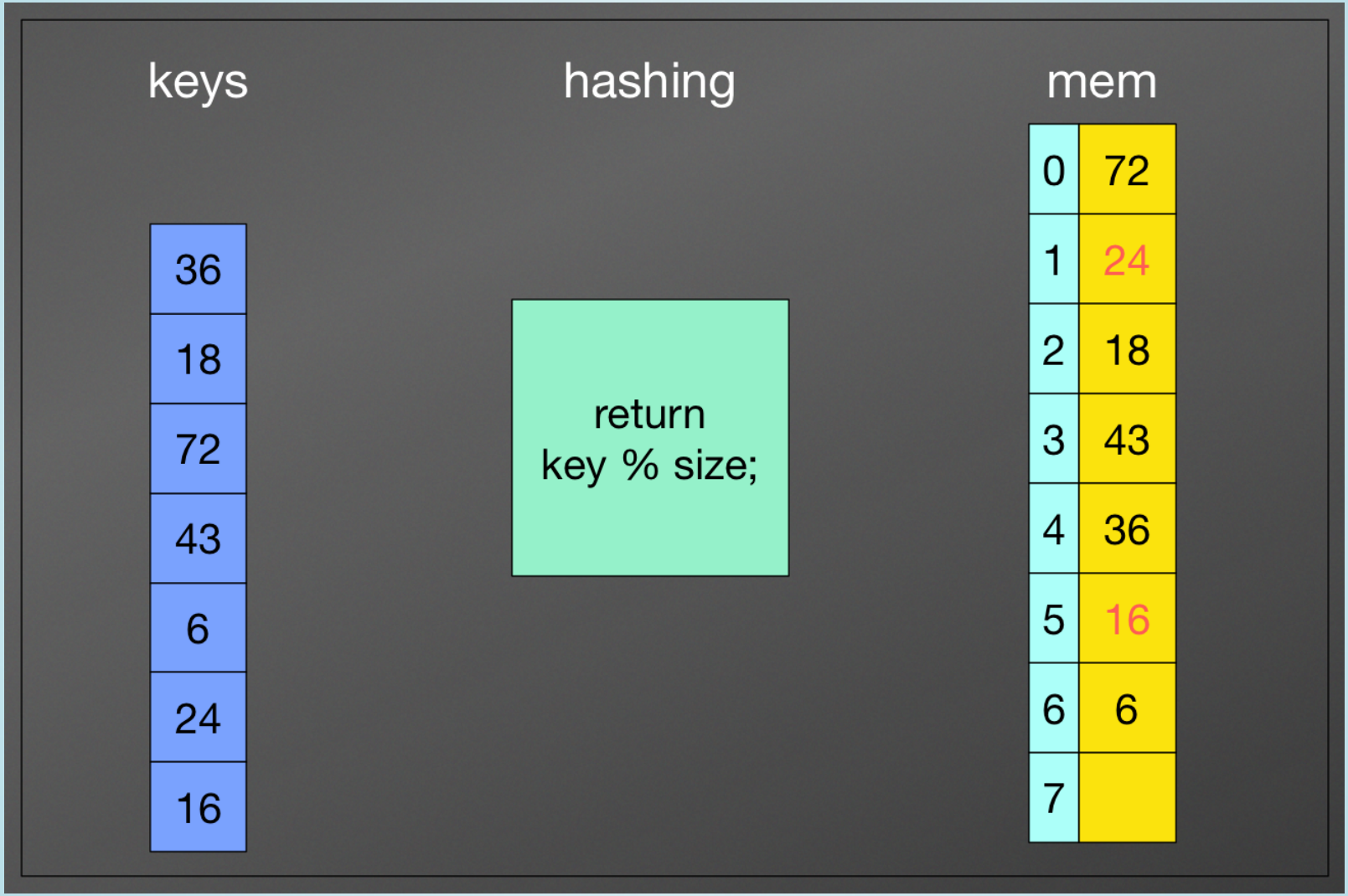
Closed hashing:

we store them in somewhere else in the table

Open hashing



Closed hashing



Hash Function Key Points

1. Hash function is **not random**, given the same key, you can always find the corresponding hashing value.
2. **Easy** and **quick** to compute.
3. **Distribution** as even as possible

Hashcode & Equals

These two functions are very important in Java

They are functions in Object class

We can override them and it will affect our HashMap collision and basic operations

Can we have Map<int, Object>?

Hashcode & Equals

Primitive Type Wrapper class:

Short, Long, Byte, Char, Float, Double, Boolean

Unless compare primitive type. Don't use ==

Don't rely on two references being identical.

Always use .equals() (Constant Pool)

Should always override .equals() for data objects.

If no override, find its parent class till Object (==)

Hashcode & Equals

```
// Object equals will be same as ==
public boolean equals(Object obj) {
    return (this == obj);
}
```

During implementation

If obj1.equals(obj2) return true

then must have

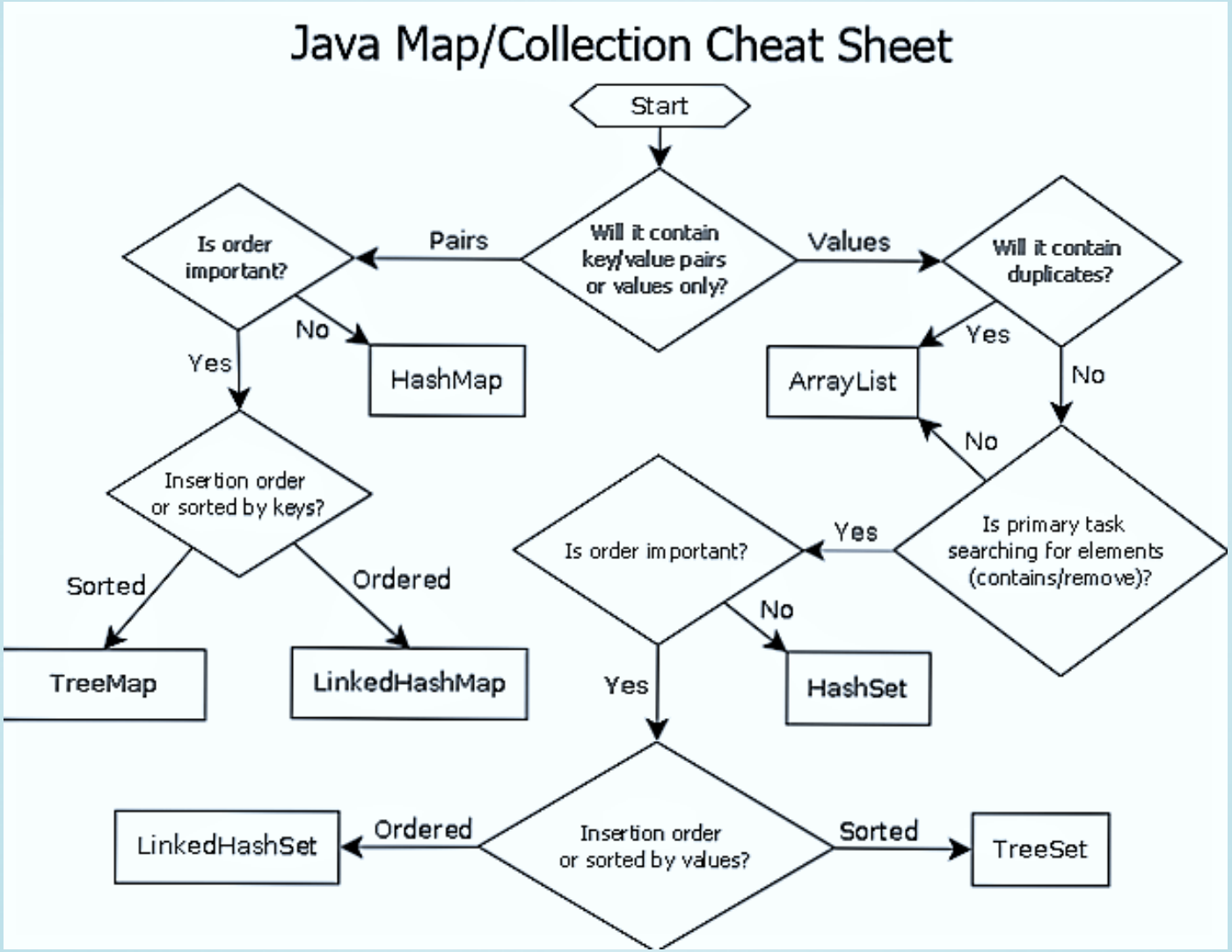
obj1.hashcode() == obj2.hashcode()

Java Map & Set

	Hash	**Tree**	**LinkedHash**
Map	HashMap	TreeMap	LinkedHashMap
Set	HashSet	TreeSet	LinkedHashSet

http://javaconceptoftheday.com/hashset-vs-linkedhashset-vs-treeset-in-java/

Different Maps and Sets



Word Pattern

Given a pattern and a string str, find if str follows the same pattern.

```
pattern = "abba", str = "dog cat cat dog" should return true.
pattern = "abba", str = "dog cat cat fish" should return false.
pattern = "aaaa", str = "dog cat cat dog" should return false.
pattern = "abba", str = "dog dog dog dog" should return false.
```

Notes:

1. You may assume pattern contains only lowercase letters,
2. str contains lowercase letters separated by a single space.

Word Pattern

```
public class Solution {
    public boolean wordPattern(String pattern, String str) {
        Map<Character, String> hm = new HashMap<>();
        Map<String, Character> hm_r = new HashMap<>();
        String[] words = str.split(" ");
        if (pattern.length() != words.length) {
            return false;
        }
        for (int i = 0; i < pattern.length(); i++) {
            char a = pattern.charAt(i);
            String b = words[i];
            if (!hm.containsKey(a)) {
                hm.put(a, b);
            } else if (!hm.get(a).equals(b)){
                return false;
            }
            if (!hm_r.containsKey(b)) {
                hm_r.put(b, a);
            } else if (hm_r.get(b) != a) {
                return false;
            }
        }
        return true;
    }
}
```

Word Pattern

```
public boolean wordPattern(String pattern, String str) {
    Map<Character, Integer> hm = new HashMap<>();
    Map<String, Integer> hm_r = new HashMap<>();
    String[] words = str.split(" ");
    if (pattern.length() != words.length) {
        return false;
    }
    for (int i = 0; i < pattern.length(); i++) {
        if (!Objects.equals(hm.put(pattern.charAt(i), i), hm_r.put(words[i], i))) {
            return false;
        }
    }
    return true;
}
```

1. Use the return value of put method
2. Character - index - String

Word Pattern II

Given a pattern and a string str, find if str follows the same pattern.

Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty substring in str.

Examples:

1. pattern = "abab", str = "redblueredblue" should return true.
2. pattern = "aaaa", str = "asdasdasdasd" should return true.
3. pattern = "aabb", str = "xyzabcxzyabc" should return false.

Word Pattern II

So apart from using HashMap to store the words to Character mapping, we also need to know what the words are.

So besides HashMap, we need to use recursion to go through all the possible split cases to find right words for the Map.

Word Pattern II

```
public boolean wordPatternMatch(String pattern, String str) {
    Map<Character, String> map = new HashMap<>();
    Set<String> set = new HashSet<>();
    return isMatch(str, 0, pattern, 0, map, set);
}
boolean isMatch(String str, int i, String pattern, int j, Map<Character, String> map, Set<String> set
    if (i == str.length() && j == pattern.length()) return true;
    if (i == str.length() || j == pattern.length()) return false;
    char c = pattern.charAt(j);
    if (map.containsKey(c)) {
      String s = map.get(c);
      if (!str.startsWith(s, i)) return false;
      else return isMatch(str, i + s.length(), pattern, j + 1, map, set);
    }
    for (int k = i; k < str.length(); k++) {
      String p = str.substring(i, k + 1);
      if (set.contains(p)) {
        continue;
      }
      map.put(c, p);
      set.add(p);
      if (isMatch(str, k + 1, pattern, j + 1, map, set)) {
        return true;
      }
      map.remove(c);
      set.remove(p);
    }
    return false;
}
```

Group Anagrams

Given an array of strings, group anagrams together.

For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"]

```
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

Note

1. For the return value, each inner list's elements must follow the lexicographic order.
2. All inputs will be in lower-case.

Group Anagrams

How to know two words are anagrams?

"ate" and "eat" ->

when we sort them, ate and eat both become "aet"

So we need a hashmap to tell if we already have the list or not.

Group Anagrams

```
public List<List<String>> groupAnagrams(String[] strs) {
    HashMap<String, List<String>> stringmap = new HashMap();
    List<List<String>> result = new ArrayList();
    for(String str: strs) {
        char[] chars = str.toCharArray();
        Arrays.sort(chars);
        String sorted = new String(chars);
        if(stringmap.containsKey(sorted)) {
            stringmap.get(sorted).add(str);
        } else {
            ArrayList<String> stringList = new ArrayList();
            stringList.add(str);
            stringmap.put(sorted, stringList);
        }
    }
    for(Map.Entry<String, List<String>> entry: stringmap.entrySet()) {
        List stringresult = entry.getValue();
        Collections.sort(stringresult);
        result.add(stringresult);
    }
    return result;
}
```

Group Anagrams

```
public List<List<String>> groupAnagrams(String[] strs) {
    HashMap<String, List<String>> hm = new HashMap();
    List<List<String>> result = new ArrayList();
    for(String s: strs) {
        char[] chars = s.toCharArray();
        Arrays.sort(chars);
        String sorted = new String(chars);
        if(!hm.containsKey(sorted)) {
            hm.put(sorted, new ArrayList());
        }
        hm.get(sorted).add(s);
    }
    // keySet -> list and sort will make outer list lexicographic
    for (List<String> list : hm.values()) {
        Collections.sort(list);
        result.add(list);
    }
    return result;
}
```

Contiguous Array

Given a binary array, find the maximum length of a contiguous subarray with equal number of 0 and 1.

Example:

Input: [0,1] Output: 2
Explanation: [0, 1] is the longest contiguous subarray with equal number of 0 and 1.

Input: [0,1,0] Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.

Contiguous Array

How do we know the diff?

We keep counting from the beginning to the end

We use a Hashmap to store

If we get something that diff[i] == diff[j]

Meaning between i - j the number of 0's and 1's are the same

Contiguous Array

```
public int findMaxLength(int[] nums) {
    int res = 0;
    int n = nums.length;

    int[] diff = new int[n + 1];

    Map<Integer, Integer> map = new HashMap<>();
    map.put(0, 0);

    for (int i = 1; i <= n; i++) {
        diff[i] = diff[i - 1] + (nums[i - 1] == 0 ? -1 : 1);

        if (!map.containsKey(diff[i]))
            map.put(diff[i], i);
        else
            res = Math.max(res, i - map.get(diff[i]));
    }

    return res;
}
```

Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

Examples:

Given "abcabcbb", the answer is "abc", which the length is 3.

Given "bbbbb", the answer is "b", with the length of 1.

Given "pwwkew", the answer is "wke", with the length of 3. Note that the answer must be a **substring**, "pwke" is a *subsequence* and not a substring.

Longest Substring Without Repeating Characters

Given a string, find the length of the **longest substring** without repeating characters.

```
Examples:

Given "abbcdefdgh", the answer is "bcdef", which the length is 5

0 1 2 3 4 5 6 7 8 9
a b b c d e f d g h
_ _ i _ _ _ _ j _ _

// Sliding Window
// Two Pointers
```

Longest Substring Without Repeating Characters

```
public int lengthOfLongestSubstring(String s) {
    int result = 0;
    if (s == null || s.length() == 0) {
        return 0;
    }
    char[] arr = s.toCharArray();
    Map<Character, Integer> hm = new HashMap<>();
    for (int i = 0, j = 0; i < arr.length; i++) {
        char c = arr[i];
        if (hm.containsKey(c)) {
            j = Math.max(j, hm.get(c) + 1);
        }
        hm.put(c,i);
        result = Math.max(result, i - j + 1);
    }
    return result;
}
```

Longest Substring Without Repeating Characters

```
public int lengthOfLongestSubstring(String s) {
    int result = 0;
    int[] cache = new int[256];
    for (int i = 0, j = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (cache[c] > 0) {
            j = Math.max(j, cache[c]);
        }
        cache[c] = i + 1;
        result = Math.max(result, i - j + 1);
    }
    return result;
}
```

The idea of Hashing

Longest Substring Without Repeating Characters

```
public int lengthOfLongestSubstring(String s) {
    int result = 0;
    int[] cache = new int[256];
    for (int i = 0, j = 0; i < s.length(); i++) {
        j = Math.max(j, cache[s.charAt(i)]);
        cache[s.charAt(i)] = i + 1;
        result = Math.max(result, i - j + 1);
    }
    return result;
}
```

Minimum Window Substring

Given a string S and a string T, find the minimum window in S which will contain all the characters in T in complexity O(n).

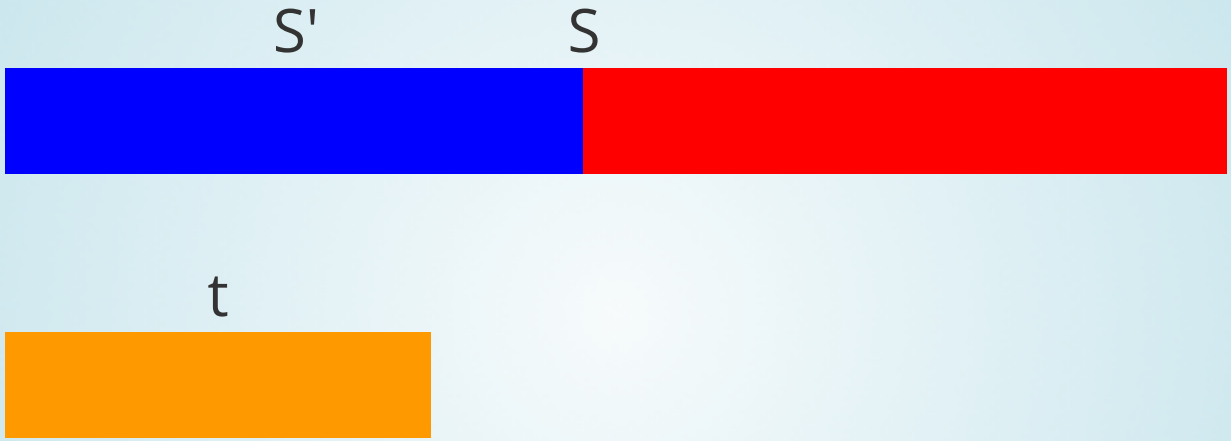
For example,

S = "ADOBECODEBANC"

T = "ABC"

Minimum window is "BANC".

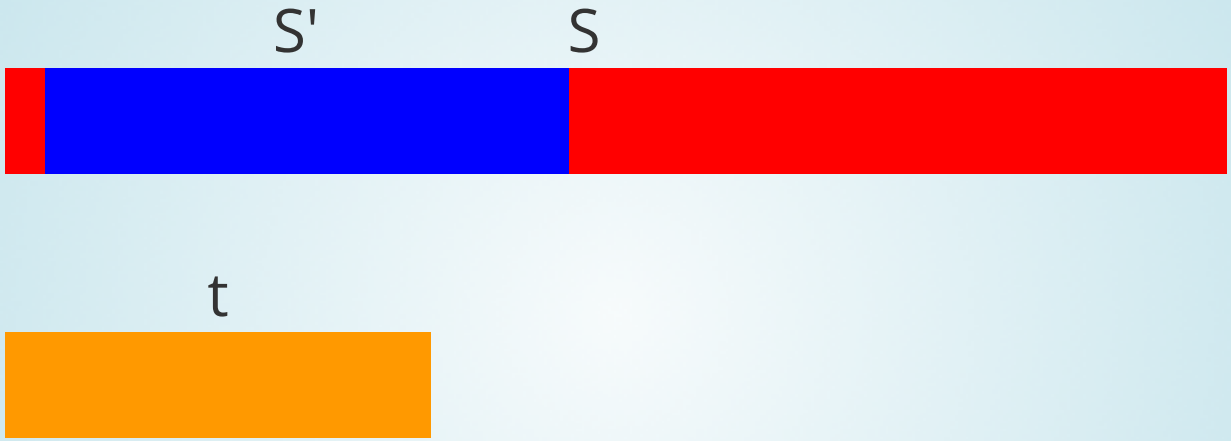
Minimum Window Substring



End pointer:

keep including characters to find a feasible substring

Minimum Window Substring



Start pointer:

keep excluding characters to make the feasible substring shorter

```
public String minWindow(String s, String t) {
    int[] pattern = new int[256];
    int[] text = new int[256];
    for(int i = 0; i < t.length(); i ++) {
        pattern[t.charAt(i)] ++;
    }
    int start = 0, end = 0, minStart = 0, minEnd = 0;
    int minLen = s.length() + 1;
    int count = t.length();
    boolean isIncluded = false;
    text[s.charAt(0)] ++;
    if(pattern[s.charAt(0)] >= 1) count --;
    while(true) {
        if(count == 0) {
                isIncluded = true;
                while(text[s.charAt(start)] > pattern[s.charAt(start)]) {
                        text[s.charAt(start ++)] --;
                }
                if(minLen > end - start + 1) {
                        minStart = start;
                        minEnd = end;
                        minLen = end - start + 1;
                }
        }
        if(end < s.length() - 1) {
                text[s.charAt(++end)] ++;
                if(pattern[s.charAt(end)] >= text[s.charAt(end)]) count --;
        }
        else break;
    }
    if(isIncluded) {
        return s.substring(minStart, minEnd + 1);
    }
    else return "";
}
```

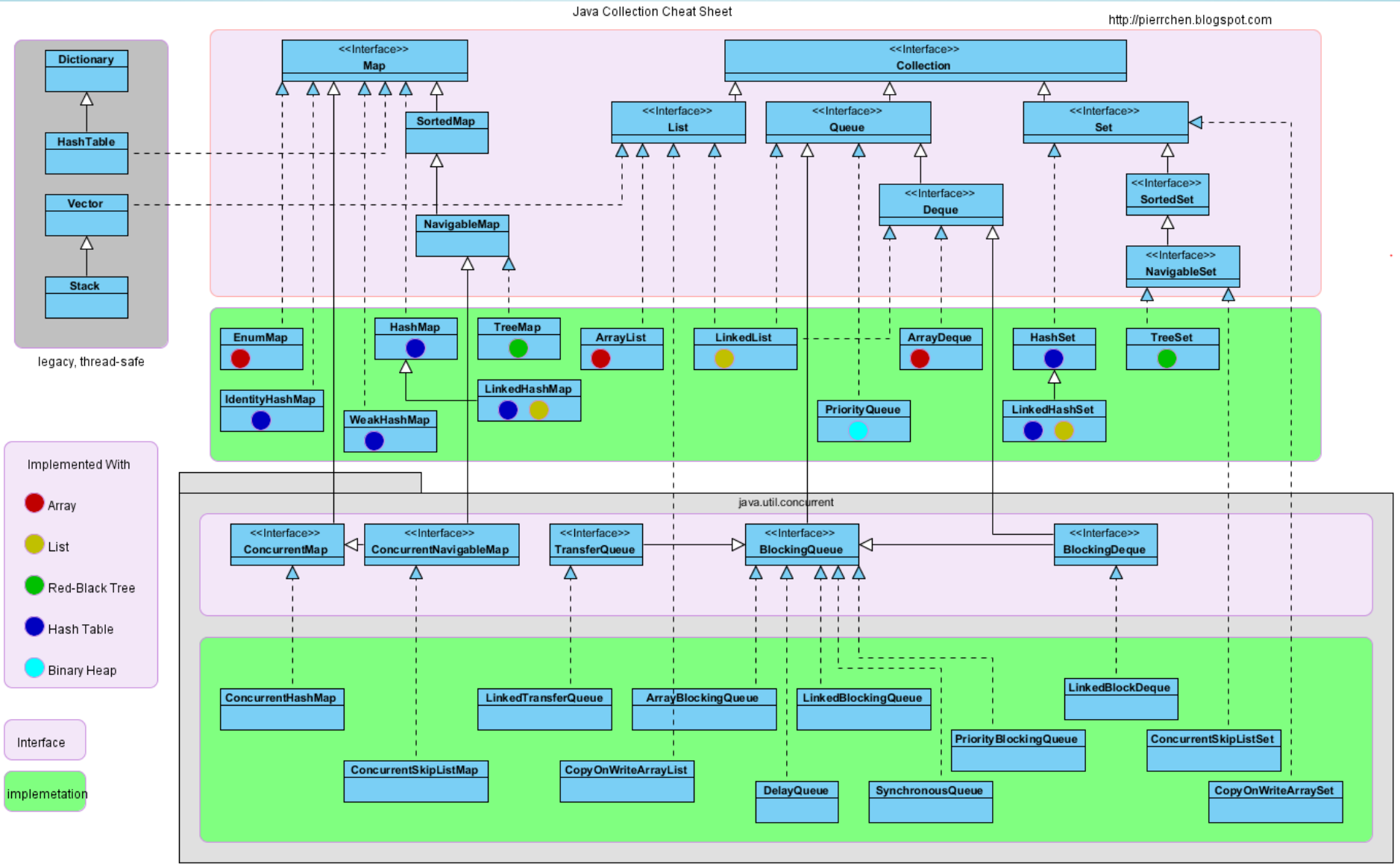
HashSet

HashMap	HashSet
put(key, value)	add(element)
get(key)	
remove(key)	remove(object)
containsKey(key)	contains(object)
containsValue(value)	
entrySet()	
keySet()	
values()	
size(); isEmpty(); putAll(Map<>);	size(); isEmpty(); addAll(Collection);

Java Map & Set

Property	HashMap	TreeMap	LinkedHashMap
Order	no guarantee order	sorted according to order it is defined	insertion-order
Get/put containsKey	O(1)	O(log(n))	O(1)
Interfaces	Map	Map, SortedMap, NavigableMap	Map
Implementation	Buckets	Red-Black Tree	Double-linked buckets
Null value/keys	allowed	only values	allows

Java Collections Framework (JCF) Cheat Sheet



Homework

Required

1. Happy Number
2. Isomorphic Strings
3. Substring with Concatenation of All Words

Optional

1. Max Points on a Line
2. Copy List with Random Pointer
3. Valid Sudoku
4. Brick Wall
5. Fraction to Recurring Decimal