# Maps Collection

ANJU MUNOTH

# Maps

- Java Collection Framework provides a variety of interfaces and classes specifically designed for managing key-value pairs, known as Map collections.

# Map Interface

Map<K, V>:

- Root interface for maps.
- Represents a collection of key-value pairs where each key maps to exactly one value.

Key methods:

- put(K key, V value): Adds a key-value pair to the map.
- get(Object key): Retrieves the value associated with the key.
- remove(Object key): Removes the key-value pair.
- containsKey(Object key): Checks if the map contains the key.
- containsValue(Object value): Checks if the map contains the value.
- keySet(): Returns a Set of all keys.
- values(): Returns a Collection of all values.
- entrySet(): Returns a Set of all key-value pairs.

# SortedMap<K, V>: Interface

- Extends Map to provide a total ordering on its keys.
- Keys are sorted either by their natural ordering or by a specified comparator.
- Key methods:
- firstKey(): Returns the first (lowest) key.
- lastKey(): Returns the last (highest) key.
- headMap(K toKey): Returns a view of the portion of the map whose keys are less than toKey.
- tailMap(K fromKey): Returns a view of the portion of the map whose keys are greater than or equal to fromKey.
- subMap(K fromKey, K toKey): Returns a view of the portion of the map whose keys range from fromKey, inclusive, to toKey, exclusive.

# NavigableMap<K, V>: Interface

- Extends SortedMap to provide methods for navigating the map.

Key methods:

- lowerKey(K key): Returns the greatest key strictly less than the given key.
- floorKey(K key): Returns the greatest key less than or equal to the given key.
- ceilingKey(K key): Returns the least key greater than or equal to the given key.
- higherKey(K key): Returns the least key strictly greater than the given key.
- descendingMap(): Returns a reverse order view of the map.

# Class :HashMap<K, V>:

- Implements the Map interface.
- Allows null values and one null key.
- Provides constant time performance for basic operations (e.g., get, put).

Example:

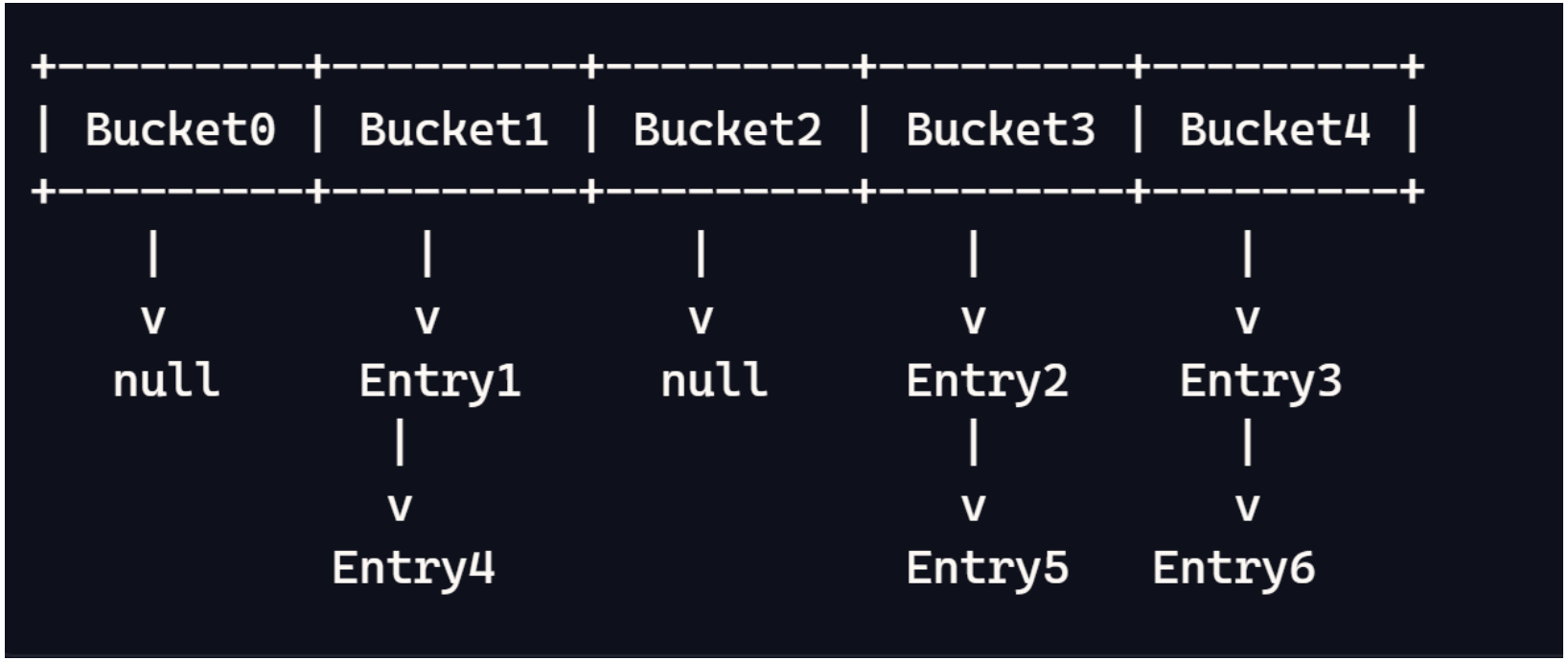
```
Map<Integer, String> hashMap = new HashMap<>();

hashMap.put(1, "One");

hashMap.put(2, "Two");
```

# Class :HashMap<K, V>:

- **Structure**: HashMap uses an array of buckets, where each bucket is a linked list (or a balanced tree for Java 8 and later if the list gets too long).
- **Storage Mechanism**:

1. **Hash Function**: The key's hash code is computed, and the hash function determines the bucket index.
2. **Bucket Array**: The key-value pairs are stored in the appropriate bucket based on the index.
3. **Collision Handling**: If multiple keys map to the same bucket, they are stored in a linked list (or a balanced tree for performance improvement).

# Class :HashMap<K, V>:



# Time Complexity

- **Average Case:**get(key), put(key, value), and remove(key) operations generally have O(1) time complexity.
  - This constant-time performance is achieved when the hash function distributes keys uniformly across the buckets, leading to minimal collisions.
- **Worst Case:**In the worst-case scenario, all keys hash to the same bucket, forming a linked list (or a balanced tree in Java 8 and later).
- In such cases, the time complexity can degrade to O(n), where n is the number of elements in the bucket.
- With the introduction of balanced trees (Red-Black trees) for handling collisions in Java 8, the worst-case performance for operations within a bucket improves to O(log n).

# Factors Affecting Performance

- Load Factor:
  - Measure of how full the hash table is allowed to get before it is resized.
  - Default load factor is 0.75, which provides a good balance between time and space efficiency.
  - A lower load factor reduces the likelihood of collisions but increases memory usage.
  - A higher load factor saves memory but may lead to more collisions and degraded performance

# Factors Affecting Performance

**Initial Capacity:**

- initial capacity -- number of buckets in the hash table at the time of its creation.
- Choosing an appropriate initial capacity based on the expected number of entries can improve performance by reducing the need for resizing.
- If the number of entries exceeds the product of the load factor and current capacity, the hash table is resized (typically doubling its size) and rehashed, which can be an expensive operation.

# Empirical Performance

- **Uniform Distribution:**When keys are uniformly distributed across buckets, HashMap provides near-constant time performance for basic operations.
- **Poor Hash Function:**A poor hash function that causes many collisions can significantly degrade performance, making operations slower as linked lists or trees within buckets grow.

# When is hash map stored as a tree

- HashMap is stored as a balanced tree (specifically a Red-Black tree) when the number of elements in a single bucket exceeds a certain threshold.
- This mechanism was introduced in Java 8 to optimize performance and reduce the likelihood of hash collisions degrading the performance to O(n).

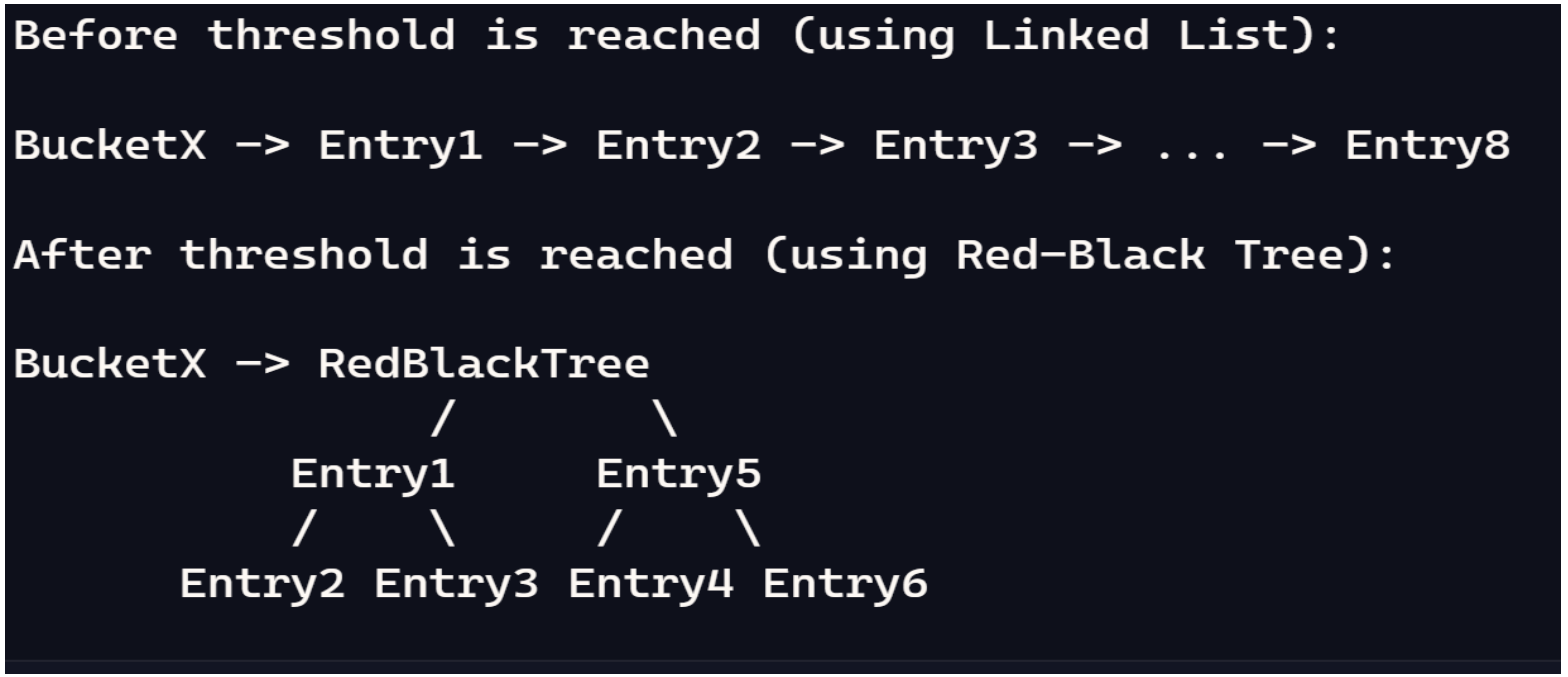
# When is hash map stored as a tree

**Threshold**:
•When the number of elements in a bucket reaches a threshold of 8, the linked list is converted into a Red-Black tree.
•This threshold helps to balance the trade-off between the constant-time complexity of a hash table and the logarithmic-time complexity of a balanced tree.

**Conversion to Tree**:
•If a bucket contains more than 8 nodes after insertion, the linked list at that bucket is transformed into a Red-Black tree.
•This transformation ensures that the operations such as get, put, and remove are performed in O(log n) time for that bucket instead of O(n) when it was a linked list.

**Tree to List Conversion**:
•If the number of nodes in the tree falls below 6 (after deletions), the tree is converted back into a linked list.
•Helps to optimize memory usage.

# When is hash map stored as a tree



# When is hash map stored as a tree

```
 Map<Integer, String> hashMap = new HashMap<>();
// Insert multiple elements with the same hash code
for (int i = 1; i <= 12; i++) {
       hashMap.put(i, "Value" + i);
  }
 // At this point, the bucket containing these keys will be converted into a Red-Black tree
System.out.println(hashMap);
```

- In this example, if all the keys have the same hash code, they will be stored in the same bucket.
- As the number of elements exceeds the threshold of 8, the linked list will be converted into a Red-Black tree to maintain efficient performance.

# Hash Code?

- A hash code is an integer value generated by a hashing algorithm.
- Used to uniquely identify objects and facilitate quick lookups, comparisons, and storage of data.
- In Java, the hashCode() method returns an integer hash code for an object.

# How Hash Codes Work in HashMap

- In a HashMap, hash codes play a crucial role in determining the bucket index where an entry (key-value pair) will be stored.

Process:

- Calculate Hash Code: The hashCode() method of the key object is called to get its hash code.
- Compute Bucket Index: The hash code is then used to compute the index of the bucket where the entry should be stored, typically using the formula:

**javaindex = (hashCode & 0x7FFFFFFF) % arrayLength;**

- Store Entry: The entry is stored in the bucket corresponding to the computed index.

# How Hash Code Works with Various Data Types

**Primitive Data Types**

- Primitive data types like int, float, double, long, etc., have predefined hash code calculations.
- **int**: The hash code is simply the value of the int.
  - Performance: This is highly efficient with a time complexity of O(1).

**int value = 42;int hashCode = Integer.hashCode(value); // hashCode is 42**

- **float**: The hash code is computed using Float.floatToIntBits(value).
- **double**: The hash code is computed using Double.doubleToLongBits(value)
- **long**: The hash code is computed by combining the higher and lower 32 bits.

# Reference Data Types -- String

- Reference data types (objects) can have custom hashCode() implementations.
- String: The hash code for a String is computed as:
- **hashCode() method for String calculates the hash code by iterating over all characters in the string.**
- has a time complexity of O(n), where n is the length of the string.
- Choice of multiplying by 31 is due to its efficiency (since multiplication by 31 can be replaced by a shift and subtraction).

# Reference Data Types -- String

```
public int hashCode() {
    int h = 0;
    int length = this.length();
    for (int i = 0; i < length; i++) {
        h = 31 * h + this.charAt(i);
    }
    return h;
}
```

```
String str = "hello";
int hashCode = str.hashCode(); // hashCode is 99162322
```

# Reference Data Types

- User-Defined Objects: For custom objects, need to override the hashCode() method in a way that considers relevant fields.

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

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

# Hash Collisions

- Hash Collisions occur when multiple keys produce the same hash code, resulting in the same bucket index.
- While HashMap is designed to handle collisions, they can still impact its **performance** in several ways.
- 1. **Increased Time Complexity**When collisions happen, multiple key-value pairs are stored in the same bucket.
  - The bucket initially uses a linked list to store these entries.
  - The time complexity for basic operations like get, put, and remove degrades from O(1) to O(n) in the worst case, where n is the number of elements in the bucket.
- 2. **Buckets and Linked Lists**
  - In a scenario with excessive collisions, many elements end up in the same bucket, resulting in longer linked lists.
  - As the linked list grows, traversal time for searching, inserting, and deleting elements increases, leading to slower performance.

# Hash Collisions

- 3. **Conversion to Balanced Trees**
  - To mitigate the performance impact of collisions, Java 8 introduced a feature that converts the linked list in a bucket to a balanced tree (Red-Black tree) once the list exceeds a threshold (default is 8 elements).
  - This conversion improves the time complexity for operations within the bucket from O(n) to O(log n).

# Performance Optimization Strategies

- Proper Hash Function Implementation:
  - Ensure that the hashCode() method for custom objects is well-designed to distribute keys uniformly across buckets.
- Initial Capacity and Load Factor:
  - Adjust the initial capacity and load factor of the HashMap to reduce the likelihood of collisions.
  - A lower load factor increases the size of the internal array, reducing collisions but using more memory
- Rehashing:When the number of elements exceeds the product of the load factor and the current capacity, HashMap automatically resizes and rehashes the entries to distribute them across the new buckets.

# HashMap --Best Scenarios

- **Fast Access and Modifications**: When you need fast insertion, deletion, and retrieval operations.
- **No Specific Order**: When the order of entries does not matter.
- **General-Purpose Storage**: Ideal for implementing caches, dictionaries, and counting occurrences.

# When is hashmap beneficial?

- **1. Fast Lookups and Inserts**
- **Use Case**: When you need constant-time performance for lookups and inserts.
- **Example**: Implementing a cache where you need to quickly access and store data based on unique keys.

```
Map<String, String> cache = new HashMap<>();
cache.put("user1", "Alice");
cache.put("user2", "Bob");

// Fast lookup
String user = cache.get("user1");
```

# When is hashmap beneficial?

- **2. Unique Key-Value Pairs**
- **Use Case**: When you need to maintain unique keys and map each key to exactly one value.
- **Example**: Storing user information where the user ID is the key and the user details are the values.

```
Map<Integer, User> userMap = new HashMap<>();
userMap.put(1, new User("Alice", 30));
userMap.put(2, new User("Bob", 25));

// Retrieve user details
User user = userMap.get(1);
```

# When is hashmap beneficial?

- **3. Sparse Data**
- **Use Case**: When dealing with a large number of potential keys but only a few are used.
- **Example**: Storing configuration settings where only a subset of possible settings is actually used.

```
Map<String, String> config = new HashMap<>();
config.put("url", "http://example.com");
config.put("timeout", "30");

// Retrieve configuration
String url = config.get("url");
```

# When is hashmap beneficial?

- **4. Indexing by Custom Keys**
- **Use Case**: When you need to index data by custom keys rather than numeric indices.
- **Example**: Implementing a dictionary where words (keys) are mapped to their definitions (values).

```
Map<String, String> dictionary = new HashMap<>();
dictionary.put("apple", "A fruit");
dictionary.put("java", "A programming language");

// Lookup definition
String definition = dictionary.get("java");
```

# When is hashmap beneficial?

- **5. Handling Dynamic Data**
- **Use Case**: When the dataset is dynamic and changes frequently, requiring efficient additions and deletions.
- **Example**: Managing sessions in a web application where sessions are created and destroyed frequently.

```
Map<String, Session> sessions = new HashMap<>();
sessions.put("session1", new Session("User1", System.currentTimeMillis()));
sessions.put("session2", new Session("User2", System.currentTimeMillis()));

// Manage sessions
Session session = sessions.get("session1");
```

# When is hashmap beneficial?

- **6. When Order Doesn't Matter**
- **Use Case**: When the order of elements is not important.
- **Example**: Counting occurrences of items where the order of items doesn't matter.

```
import java.util.HashMap;
import java.util.Map;

public class WordFrequency {
    public static void main(String[] args) {
        String text = "This is a sample text. This text is just a sample.";
        String[] words = text.split("\\W+");

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

        for (String word : words) {
            word = word.toLowerCase();
            wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
        }

        wordCount.forEach((k, v) -> System.out.println(k + ": " + v));
    }
}
```

| Feature | Array | Linked List | HashMap |
| --- | --- | --- | --- |
| **Access Time** | O① | O(n) | O① (Average), O(n) (Worst) |
| **Search Time** | O(n) | O(n) | O① (Average), O(n) (Worst) |
| **Insertion Time** | O(n) | O① (if reference known) | O① (Average), O(n) (Worst) |
| **Deletion Time** | O(n) | O① (if reference known) | O① (Average), O(n) (Worst) |
| **Memory Usage** | Fixed size | Dynamic, memory overhead | Dynamic, memory overhead |
| **Best Use Cases** | Static, fixed-size data | Dynamic data, frequent insertions/deletions | Fast lookups, key-value pairs |

# LinkedHashMap<K, V>:

- Extends HashMap and maintains a doubly-linked list of its entries.
- Orders elements based on insertion order or access order.

```
Map<Integer, String> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put(1, "One");
linkedHashMap.put(2, "Two");
```

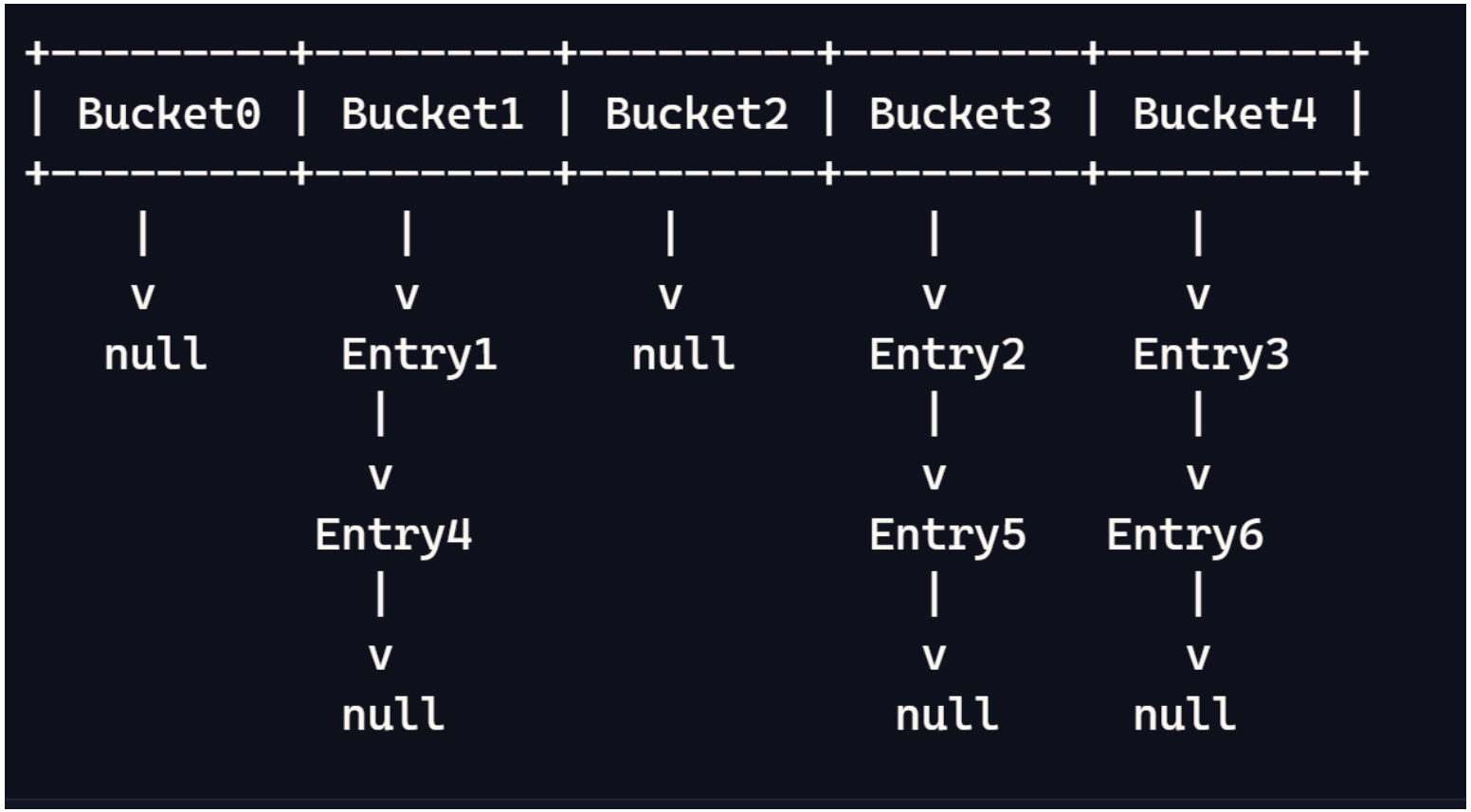
- Allows it to maintain the order of entries, either by insertion order or access order, making it a useful data structure in specific scenarios where order matters.

# LinkedHashMap<K, V>:

- **Structure**: LinkedHashMap extends HashMap by maintaining a doubly-linked list to preserve insertion order or access order.
- **Storage Mechanism**:

1. **Hash Function and Bucket Array**: Same as HashMap.
2. **Linked List**: Each entry in the bucket array is linked to the next entry, preserving insertion order.

# LinkedHashMap<K, V>:



# LinkedHashMap constructor

- LinkedHashMap class in Java can be configured using three parameters in its constructor.
- Parameters are initialCapacity, loadFactor, and accessOrder.

# LinkedHashMap constructor -- parameters

**initialCapacity**:
- **Type**: int
- **Description**: Specifies the initial number of buckets in the hash table. This determines how many elements the LinkedHashMap can initially hold before needing to resize.
- **Default**: 16
- **Usage**: Useful to set based on the expected number of entries to minimize resizing.

**loadFactor**:
- **Type**: float
- **Description**: Controls the threshold at which the LinkedHashMap will resize. It is a measure of how full the map can get before it needs to expand.
- **Default**: 0.75
- **Usage**: Lower values reduce the chance of collisions but use more memory. Higher values save memory but increase the chance of collisions.

# LinkedHashMap constructor -- parameters

**accessOrder**:
- **Type**: boolean
- **Description**: Determines whether the order of entries should be based on insertion order (false) or access order (true).
- **Default**: false (insertion order)
- **Usage**: Set to true to create an LRU (Least Recently Used) cache or to keep track of the order of access.

# Characteristics -- **LinkedHashMap**

- Order Preservation:
  - Insertion Order:
    - By default, it maintains the order in which entries are inserted.
  - Access Order: It can be configured to maintain the order based on the access of entries, making it suitable for use cases like LRU caches.
- Performance:
  - Time Complexity:
    - get(key): O(1)put(key, value): O(1)remove(key): O(1)
  - The overhead is slightly higher than HashMap due to the maintenance of the doubly-linked list, but the average performance remains close to O(1) for basic operations.

# LinkedHashMap -Best Scenarios

- **Preserving Order**: When you need to maintain insertion order or access order.
  - **Example**: Logging events in order, preserving the sequence of user actions.
- **LRU Caching**: Perfect for implementing LRU (Least Recently Used) caches where the order of access needs to be tracked.
- **Iterating in Specific Order**:
  - When the iteration order matters, such as generating reports where the order of records is important.

# LRU (Least Recently Used) Caches:

- Use Case: When you need to evict the least recently accessed entries.

```
Map<String, String> lruCache = new LinkedHashMap<>(16, 0.75f, true) {
    protected boolean removeEldestEntry(Map.Entry<String, String> eldest) {
        return size() > 100; // Maximum size of 100 entries
    }
};
```

# Iterating in Specific Order:

- When the iteration order matters, such as generating reports where the order of records is important.

```
Map<Integer, String> accessOrderMap = new LinkedHashMap<>(16, 0.75f, true);
accessOrderMap.put(1, "Data1");
accessOrderMap.put(2, "Data2");
accessOrderMap.put(3, "Data3");
System.out.println(accessOrderMap); // {1=Data1, 2=Data2, 3=Data3}
```

# Real-World Use Cases :Web Session Management

- Managing user sessions in a web application, where sessions need to be evicted based on access patterns.

```
Map<String, String> sessionCache = new LinkedHashMap<>(16, 0.75f, true) {
    protected boolean removeEldestEntry(Map.Entry<String, String> eldest) {
        return size() > 1000; // Maximum of 1000 sessions
    }
};
```

# Real-World Use Cases :**Access Logs**:

- Keeping track of access logs in order, preserving the sequence of access events

```
Map<String, String> accessLogs = new LinkedHashMap<>();
accessLogs.put("2025-01-01 10:00:00", "User A accessed the system");
accessLogs.put("2025-01-01 10:05:00", "User B accessed the system");
```

# Real-World Use Cases :**Application Caches**:

- Implementing application-level caches that need to maintain the order of entries.

```
Map<String, String> appCache = new LinkedHashMap<>(16, 0.75f, true) {
    protected boolean removeEldestEntry(Map.Entry<String, String> eldest) {
        return size() > 50; // Maximum size of 50 entries
    }
};
```

# Why Use LinkedHashMap Over Other Structures

- **Order Preservation:** Unlike HashMap, LinkedHashMap maintains insertion order, making it suitable for scenarios where the order of entries is important.
- **Access Order**: It can be configured to maintain access order, which is useful for implementing LRU caches.
- **Iteration Predictability**: Provides predictable iteration order, which is essential for tasks that require consistent order, such as generating reports or logs.

| Feature | LinkedHashMap | HashMap | TreeMap |
| --- | --- | --- | --- |
| **Structure** | Hash table + doubly linked list | Hash table | Red-Black tree |
| **Order** | Insertion order (default) or access order | No order | Natural ordering or custom comparator |
| **Time Complexity** | O① for `put`, `get`, `remove` | O① for `put`, `get`, `remove` | O(log n) for `put`, `get`, `remove` |
| **Memory Overhead** | Higher due to linked list | Lower | Higher due to tree nodes |
| **Null Handling** | Allows one null key and multiple null values | Allows one null key and multiple null values | Does not allow null keys, allows null values |
| **Use Cases** | LRU caches, order-preserving collections | General-purpose map | Sorted data storage, range queries |
| **Iteration** | Predictable order (insertion/access) | Unpredictable order | Sorted order (natural/comparator) |
| **Concurrency** | Not thread-safe | Not thread-safe | Not thread-safe |

```
public class LRUCache<K, V> extends LinkedHashMap<K, V>
{
        private final int capacity;
        public LRUCache(int capacity)
        {
                super(capacity, 0.75f, true);
                this.capacity = capacity;
        }
        @Override
        protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
        {
                return size() > capacity;
        }
}
```

# Linked hashmap example maintaining the access order

## Linked hashmap example maintaining the access order

```
public static void main(String[] args)
{
        // Create an LRU Cache with a capacity of 3
         LRUCache<Integer, String> lruCache = new LRUCache<>(3);
        lruCache.put(1, "One");
        lruCache.put(2, "Two");
        lruCache.put(3, "Three");
        System.out.println("Initial cache: " + lruCache);
        // Access some entries
        lruCache.get(1);
        lruCache.get(2);
        System.out.println("Cache after accessing keys 1 and 2: " + lruCache);
        // Add a new entry, which causes the eldest entry (key 3) to be evicted
        lruCache.put(4, "Four");
        System.out.println("Cache after adding key 4: " + lruCache);    }
```

Output:

Initial cache: {1=One, 2=Two, 3=Three}
Cache after accessing keys 1 and 2: {3=Three, 1=One, 2=Two}
Cache after adding key 4: {1=One, 2=Two, 4=Four}

# TreeMap<K, V>:

- Implements NavigableMap and SortedMap interfaces.
- Uses a Red-Black tree to store the map.
- Orders keys based on their natural ordering or a specified comparator.

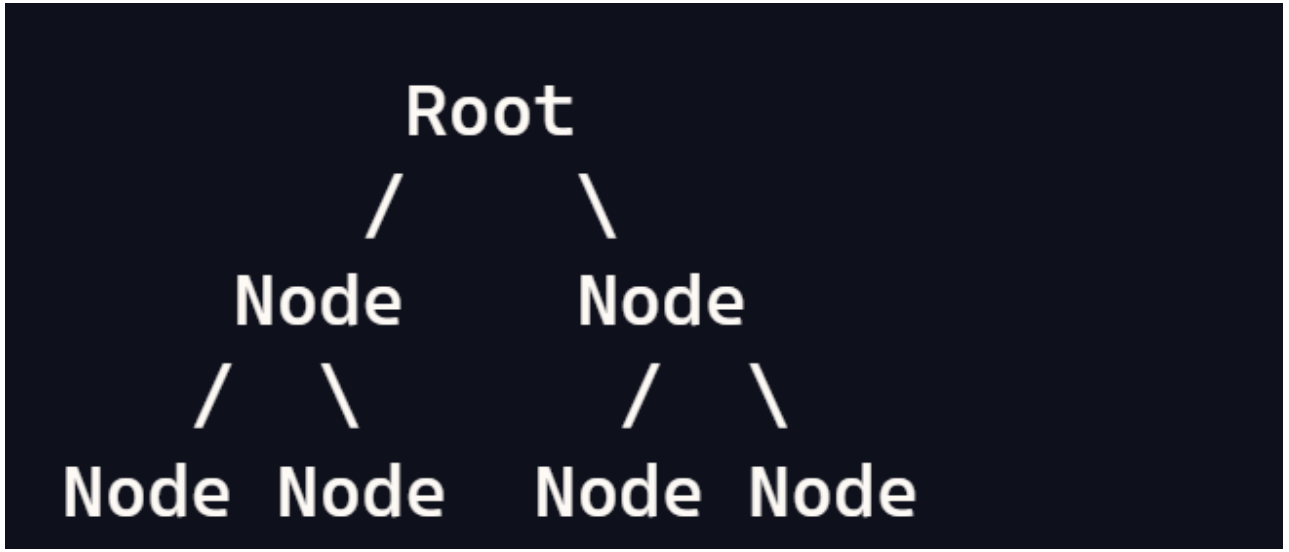
```
Map<Integer, String> treeMap = new TreeMap<>();
treeMap.put(1, "One");
treeMap.put(2, "Two");
```

# TreeMap<K, V>:

- **Structure**: TreeMap uses a Red-Black tree to store the map.
- **Storage Mechanism**:

1. **Red-Black Tree**: keys are stored in a balanced binary search tree.
2. **Node Insertion**: keys are sorted based on their natural ordering or a comparator.

# TreeMap<K, V>:



# TreeMap<K, V>:

- **Sorted Order**: TreeMap maintains the keys in natural order (ascending order) or by a specified comparator.
- **Navigable**: It provides methods to navigate the map, such as lowerKey, floorKey, ceilingKey, and higherKey.
- **Null Handling**: TreeMap does not allow null keys but allows null values.
- **Thread-Safety**: TreeMap is not synchronized. For concurrent access, consider using Collections.synchronizedMap.
- **Time Complexity**: O(log n) for put, get, remove, and containsKey operations due to the underlying Red-Black Tree structure.
- **Memory Usage**: Uses more memory compared to HashMap due to the tree nodes and balancing information.

```
public class Main {
    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new TreeMap<>();

        // Adding key-value pairs to TreeMap
        treeMap.put(3, "Three");
        treeMap.put(1, "One");
        treeMap.put(4, "Four");
        treeMap.put(2, "Two");

        // Displaying the TreeMap
        System.out.println("TreeMap: " + treeMap);

        // Accessing elements
        System.out.println("First Key: " + treeMap.firstKey());
        System.out.println("Last Key: " + treeMap.lastKey());

        // Navigating the TreeMap
        System.out.println("Lower Key of 3: " + treeMap.lowerKey(3));
        System.out.println("Higher Key of 3: " + treeMap.higherKey(3));
    }
}
```

Output

```
TreeMap: {1=One, 2=Two, 3=Three, 4=Four}
First Key: 1
Last Key: 4
Lower Key of 3: 2
Higher Key of 3: 4
```

# Red-Black Tree

**Red-Black Tree** is a type of self-balancing binary search tree with additional properties to ensure the tree remains balanced.

**Characteristics**:

- **Binary Search Tree**: Each node has at most two children, referred to as the left child and the right child.
- **Self-Balancing**: The tree maintains a balanced height by ensuring that no path from the root to a leaf is more than twice as long as any other path.
- **Red-Black Properties**:
    1. **Node Color**: Every node is either red or black.
    2. **Root Property**: The root is always black.
    3. **Leaf Property**: All leaves (NIL nodes) are black.
    4. **Red Property**: Red nodes cannot have red children (no two consecutive red nodes).
    5. **Black Property**: Every path from a node to its descendant NIL nodes has the same number of black nodes.

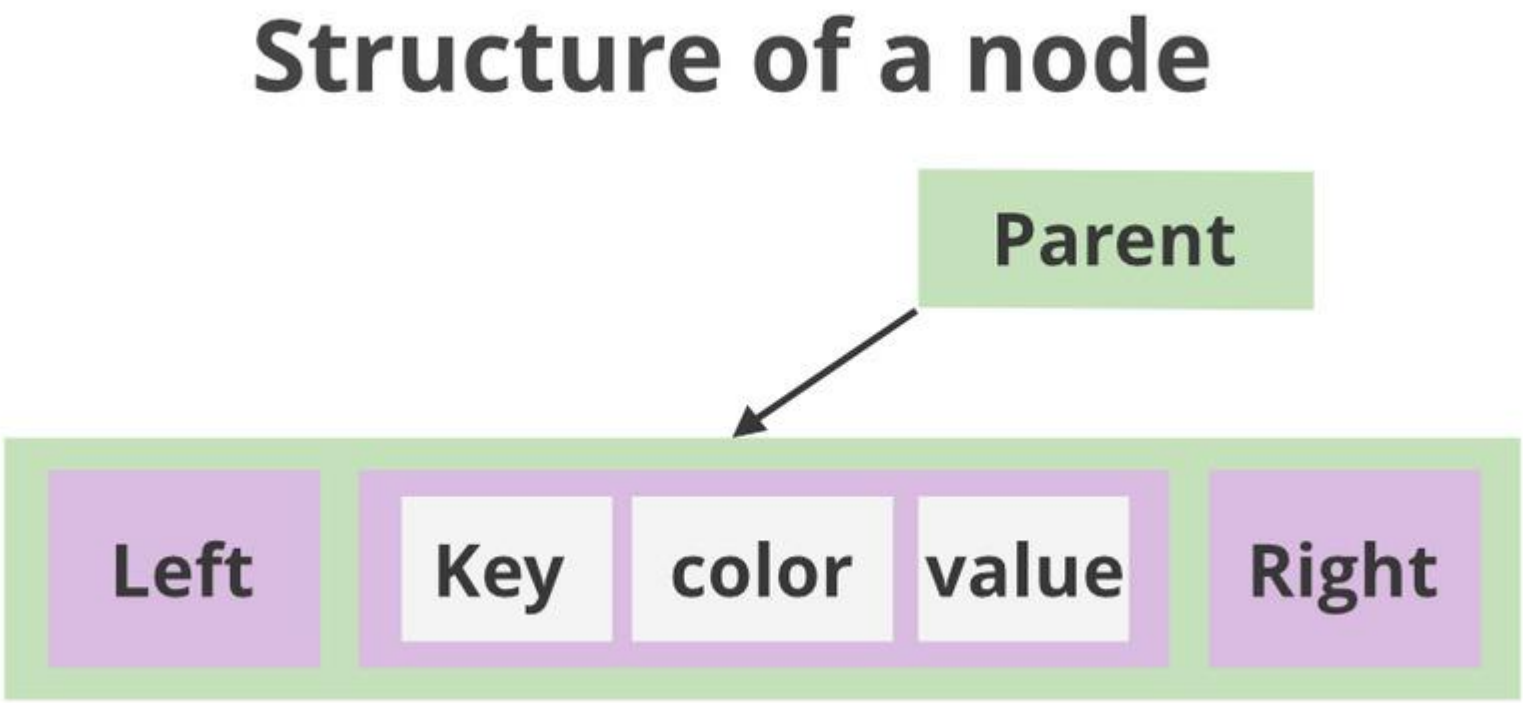
# Key Properties of Red-Black Tree

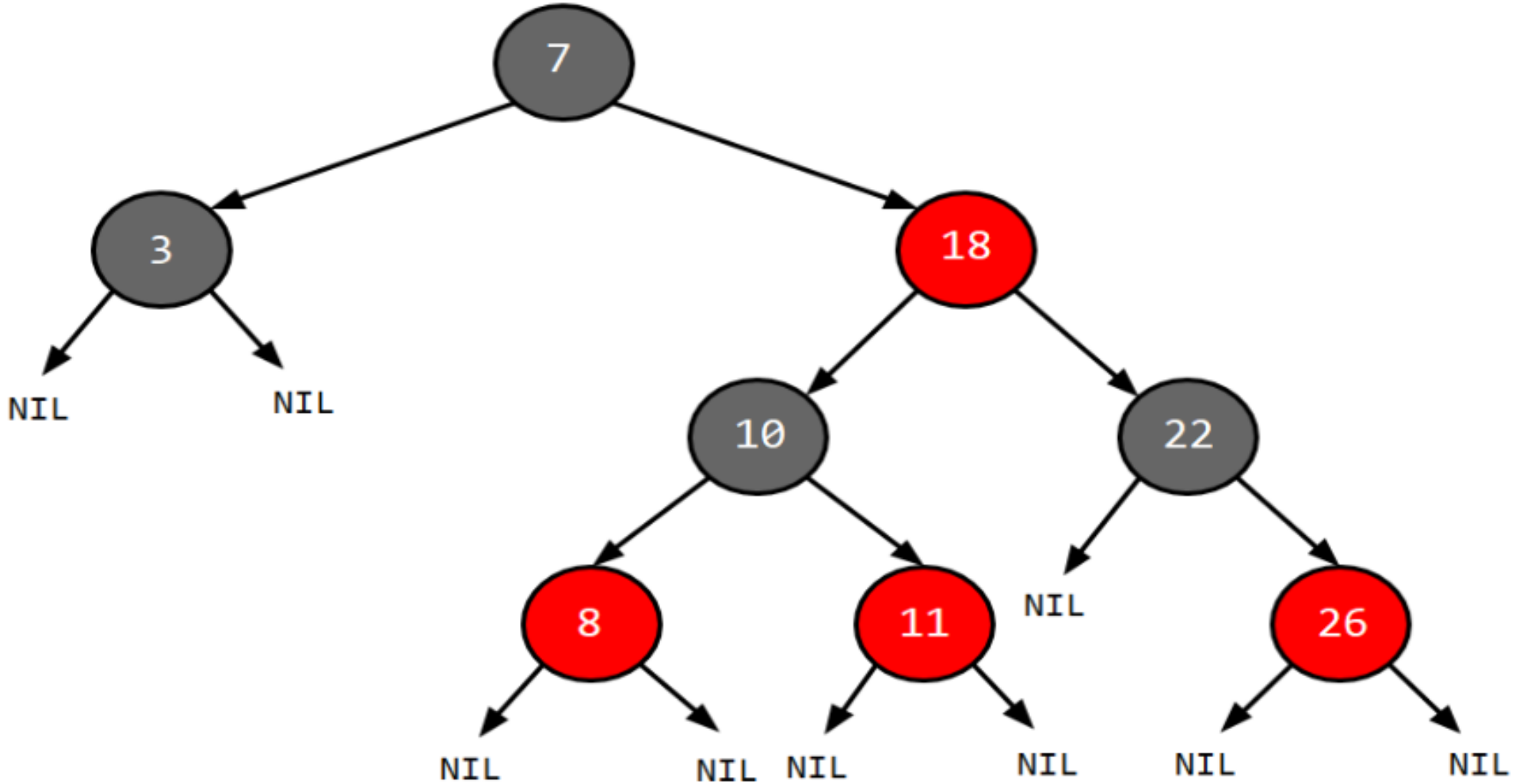
1. **Binary Search Tree Properties**: Each node's left subtree contains only nodes with keys less than the node's key, and each node's right subtree contains only nodes with keys greater than the node's key.
2. **Balancing Properties**: tree remains approximately balanced, ensuring that the longest path from the root to a leaf is no more than twice the length of the shortest path.

# How Data is Stored in a Red-Black Tree (TreeMap)

- data is stored in a Red-Black Tree where each node represents a key-value pair. The keys are used to maintain the order of the nodes.
- **Node Structure**
- A node in a Red-Black Tree typically contains:
- **Key**: The key used for ordering.
- **Value**: The associated value with the key.
- **Color**: The color of the node (red or black).
- **Left Child**: Reference to the left child node.
- **Right Child**: Reference to the right child node.
- **Parent**: Reference to the parent node.

# How Data is Stored in a Red-Black Tree (TreeMap)




7
3
18
NIL
NIL
10
22
8
11
NIL
26
NIL
NIL
NIL
NIL
NIL
NIL
NIL

# Operations and Balancing

- **Insertion**
- When a new node is inserted:

1. The node is added as a red node.
2. The tree is adjusted to maintain the Red-Black properties. This may involve:
    1. **Rotations**: Left and right rotations to maintain balance.
    2. **Recoloring**: Changing the color of nodes to maintain the Red-Black properties.

# Operations and Balancing

- **Deletion**
- When a node is deleted:

1. The node is removed, and the tree is restructured if necessary.
2. The tree is adjusted to maintain the Red-Black properties, which may involve rotations and recoloring.

# Reason for Using Red and Black Colors

- **Simplified Balancing**
- The use of red and black colors in a Red-Black Tree simplifies the balancing of the tree. These colors help in ensuring that the tree remains balanced with fewer rotations and recoloring operations.
- **Red Nodes**: Represent lighter weight paths, allowing flexibility in adding nodes without immediate rebalancing.
- **Black Nodes**: Ensure that the tree maintains a balanced structure by enforcing rules on the number of black nodes along paths.

# Reason for Using Red and Black Colors

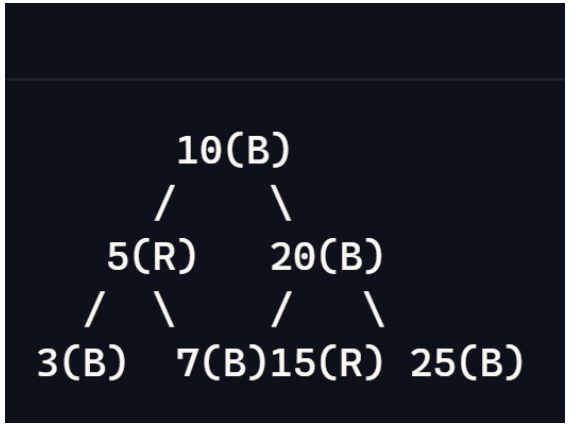
- **Efficiency**
- The Red-Black Tree provides a good balance between efficient insertion, deletion, and lookup operations, typically taking O(log n) time for each.
- **Historical Context**
- The concept of using red and black colors was introduced by Rudolf Bayer in 1972 to simplify the implementation of balanced binary search trees.

# Benefits of Using Red-Black Trees in TreeMap

- **Efficient Searches**: O(log n) complexity for search operations.
- **Balanced Structure**: Automatically maintains balance with minimal overhead.
- **Predictable Performance**: Guarantees a balanced tree, providing consistent performance.

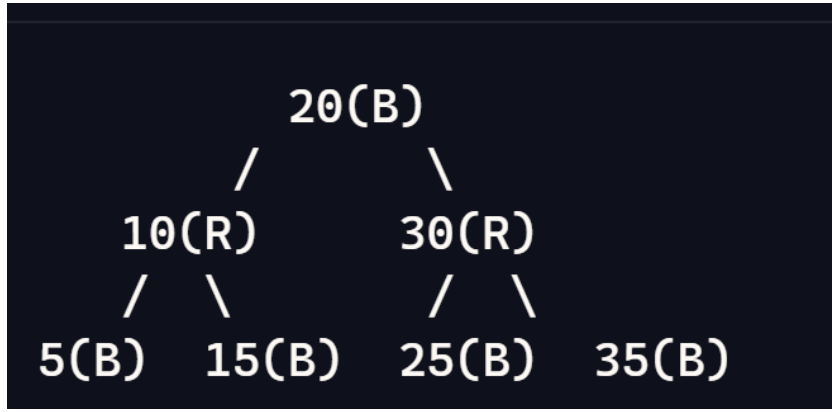
# Performance of red black tree

- **Time Complexity**: O(log n) for insertion, deletion, and lookup due to the balancing properties.
- **Balancing Operations**: Insertion and deletion may require rebalancing the tree, which involves rotations and color changes.



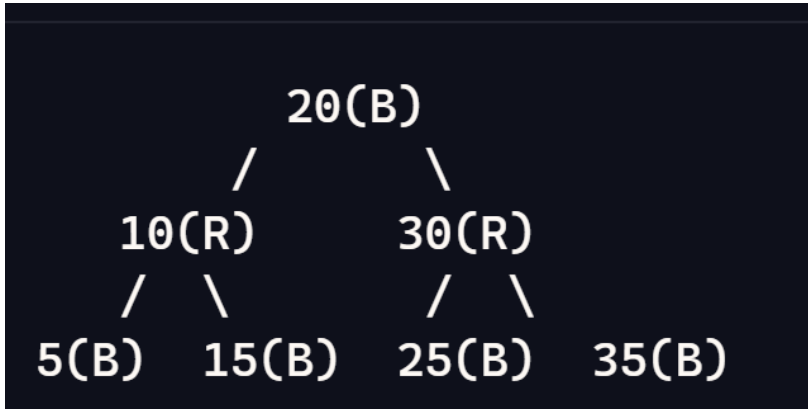
# red black tree

- Imagine a Red-Black Tree with the following nodes and values:
- Root: 20 (black)
- Left Child of Root: 10 (red)
- Right Child of Root: 30 (red)
- Left Child of 10: 5 (black)
- Right Child of 10: 15 (black)
- Left Child of 30: 25 (black)
- Right Child of 30: 35 (black)



# red black tree

- **Black Root**: The root node (20) is always black.
- **Red Nodes**: The children of the root node (10 and 30) are red.
- **Black Children**: The children of red nodes (5, 15, 25, and 35) are black.
- **Balancing**: The tree maintains balance through rotations and recoloring, ensuring no two red nodes are adjacent, and every path from the root to the leaves has the same number of black nodes.
- **Key Points**
- **No Two Red Nodes Adjacent**: This helps in maintaining balance.
- **Equal Black Nodes**: Each path from the root to the leaves has the same number of black nodes, ensuring the tree remains balanced.



# Sorted Data Storage

- **Use Case**: Storing data in sorted order for quick retrieval and traversal.
  **Example**: Maintaining a directory of employees sorted by employee ID.

```
TreeMap<Integer, String> employeeDirectory = new TreeMap<>();
employeeDirectory.put(1003, "Alice");
employeeDirectory.put(1001, "Bob");
employeeDirectory.put(1002, "Charlie");

System.out.println(employeeDirectory); // {1001=Bob, 1002=Charlie, 1003=Alice}
```

# Range Queries

- **Use Case**: Performing range queries on the data, such as finding elements within a specific range.
- **Example**: Managing stock prices and retrieving prices within a given range.

# TreeMap --Best Scenarios

- **Sorted Data**: When you need the keys to be sorted in natural order or a custom order.
- **Range Queries**: Ideal for range queries and ordered traversal.

# Constructors

- TreeMap() This default constructor constructs an empty TreeMap
- TreeMap(Map map) It creates a TreeMap with the entries from a map.
- TreeMap(Comparator compare) This is an argument constructor and it takes Comparator object to constructs an empty tree-based map. It will be sorted by using the Comparator compare.
- TreeMap(SortedMap sortedMap) It can be initialized as TreeMap with the entries from sortedMap.

```
public class TreeMapExample {
    public static void main(String[] args) {
        // Create a TreeMap to store Integer keys and String values
        TreeMap<Integer, String> treeMap = new TreeMap<>();
        // Add elements to the TreeMap
        treeMap.put(3, "Apple");
        treeMap.put(1, "Banana");
        treeMap.put(2, "Cherry");
        treeMap.put(4, "Date");
        // Print the TreeMap
        System.out.println("TreeMap: " + treeMap);
        // Access an element by its key
        String value = treeMap.get(2);
        System.out.println("Value for key 2: " + value);
        // Remove an element by its key
        treeMap.remove(3);
        System.out.println("TreeMap after removing key 3: " + treeMap);
        // Iterate over the elements
        System.out.println("Iterating over TreeMap:");
        for (Integer key : treeMap.keySet()) {
            System.out.println("Key: " + key + ", Value: " + treeMap.get(key));
        }
    }
}
```

**storing data in treemap using the default comparison**

## storing data in treemap using the Custom Comparotor

```
/ Define the Person class
class Person {
        String name;
        int age;
// Constructor
public Person(String name, int age) {
       this.name = name;
       this.age = age;
}
// Override toString() method for easy printing
@Override    public String toString() {
       return name + " (" + age + ")";
}}
```

## storing data in treemap using the Custom Comparotor

```
// Create a custom comparator to compare Person objects by age
        Comparator<Person> customComparator = new Comparator<Person>() {
            @Override
            public int compare(Person p1, Person p2) {
                return Integer.compare(p1.age, p2.age); // Compare by age
            } };
         // Create a TreeMap with the custom comparator
        TreeMap< Person,String> treeMap = new TreeMap<>(customComparator);
```

# Hashtable<K, V>:

- Implements the Map interface.
- Uses a hash table to store the map.
- Does not allow null keys or values.
- Synchronized, making it thread-safe.

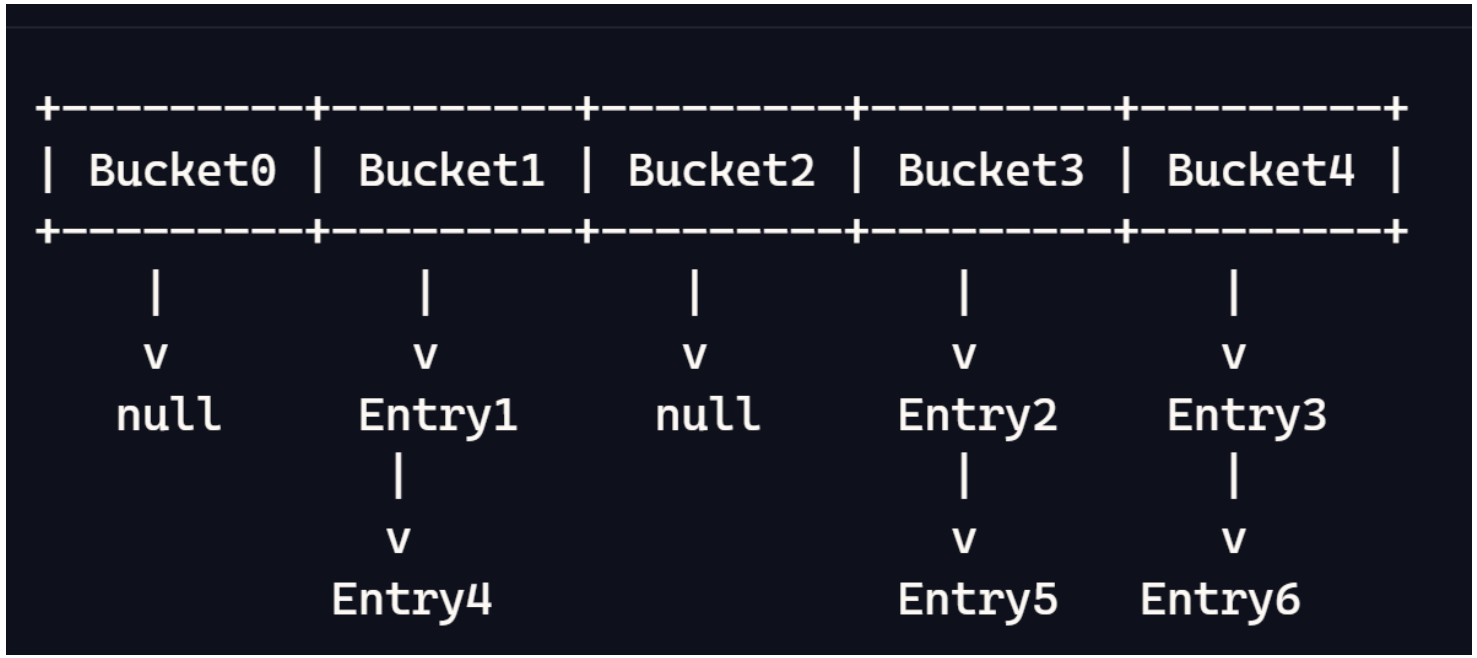
```
Map<Integer, String> hashtable = new Hashtable<>();
hashtable.put(1, "One");
hashtable.put(2, "Two");
```

# Hashtable<K, V>:

- **Structure**: Hashtable uses a synchronized hash table.
- **Storage Mechanism**:

1. **Hash Function and Bucket Array**: Same as HashMap.
2. **Synchronization**: All methods are synchronized for thread safety.

# Hashtable<K, V>:



# Hashtable --Best Scenarios:

- **Legacy Code:** When working with legacy systems that require thread-safe operations.
- **Strict Synchronization:** When a simpler, synchronized map is required, though ConcurrentHashMap is generally preferred now.

# ConcurrentHashMap<K, V>

- In java.util.concurrent package
- Implements the Map interface.
- Designed for high-concurrency environments.
- Provides thread-safe operations without locking the entire map.
- designed to handle concurrent access by multiple threads without synchronization issues, making it a scalable alternative to the traditional HashMap class.
- Fine-grained locking mechanism, which locks only the portion of the map being modified rather than the entire map.
- Enhances scalability and efficiency for concurrent operations.
- Provides various methods for atomic operations such as `putIfAbsent()`, `replace()`, and `remove()`.

# ConcurrentHashMap<K, V>

- **Thread-Safe**: Multiple threads can operate on a single object without complications
- **Fine-Grained Locking**: Only the portion of the map being modified is locked, enhancing scalability
- **No Nulls**: Inserting null objects as keys or values is not allowed[1].
- **Segment Locking**: The object is divided into segments according to the concurrency level, allowing multiple threads to perform retrieval operations simultaneously

# Constructors

- ConcurrentHashMap provides several constructors to initialize the map with different configurations:

1. `ConcurrentHashMap()`: Creates a new, empty map with default initial capacity (16), load factor (0.75), and concurrency level (16).
2. `ConcurrentHashMap(int initialCapacity)`: Creates a new, empty map with the specified initial capacity.
3. `ConcurrentHashMap(int initialCapacity, float loadFactor)`: Creates a new, empty map with the specified initial capacity and load factor.
4. `ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)`: Creates a new, empty map with the specified initial capacity, load factor, and concurrency level.
5. `ConcurrentHashMap(Map<? extends K, ? extends V> m)`: Creates a new map with the same mappings as the given map

# ConcurrentHashMap<K, V>

- **Structure**: ConcurrentHashMap uses a segmented bucket array for concurrent access.
- **Storage Mechanism**:Bucket Mechanism
- ConcurrentHashMap uses a hash table structure similar to HashMap, but with a twist to handle concurrency:
- **Buckets** (Bins): The map is divided into segments called **buckets**.
- Each bucket can be thought of as an array index where entries are stored.
- **Nodes**: Each bucket contains nodes that hold key-value pairs.
- If multiple entries hash to the same bucket, they are linked together, initially in a linked list.
- **TreeBins** (Java 8 and later): When the number of nodes in a bucket exceeds a certain threshold (usually 8), the linked list is transformed into a balanced tree (using Red-Black tree algorithms) to improve lookup times.

# Concurrency Control

**Locking Mechanism**

Instead of locking the entire map during updates, ConcurrentHashMap employs a finer-grained locking strategy:

- **Segment Locks (Pre-Java 8):** The map was divided into segments, each with its own lock. This allowed multiple threads to work on different segments simultaneously.
- **Bucket-Level Locks (Java 8 and later):** Locks are implemented at the bucket level using built-in synchronization and non-blocking algorithms.

**Optimistic Reads**

- **Non-blocking Reads:** Most read operations are lock-free and proceed without waiting, using volatile variables to ensure visibility across threads.

**Atomic Operations**

- **Compare-And-Swap (CAS):** Utilizes low-level atomic CPU instructions to update values without full locks, ensuring high performance during concurrent writes.

# Concurrency

## Bucket-Level Synchronization

- **Synchronization on Buckets**: When a thread needs to write to a bucket, it locks only that bucket, allowing other threads to access different buckets simultaneously.

## Lock-Free Reads

- **Memory Consistency**: Reads are performed without locking, and changes made by one thread become visible to others due to the use of volatile variables and memory barriers.

## Concurrent Resizing

- **Table Expansion**: When the map needs to resize (grow), it does so in a thread-safe manner, with threads assisting in the resizing process to reduce contention.

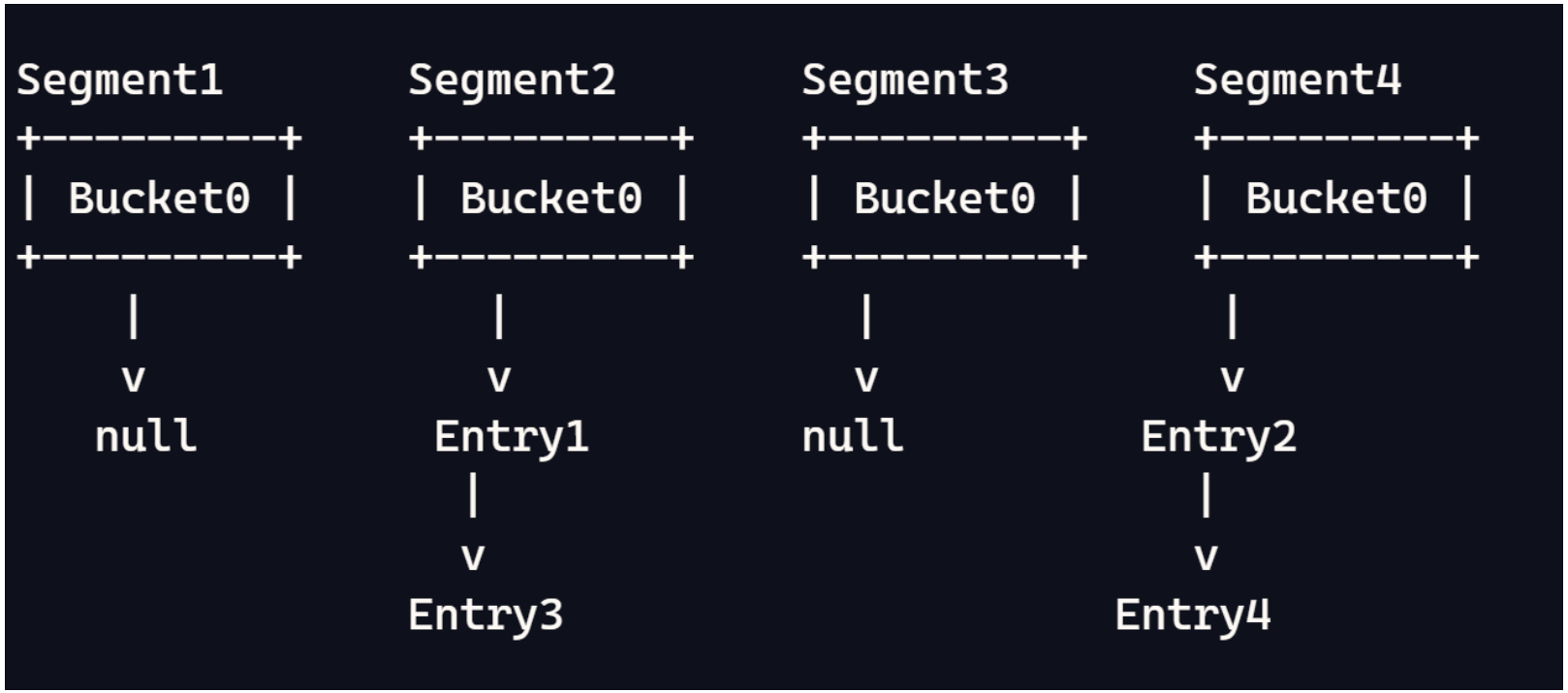
# Concurrent Updates

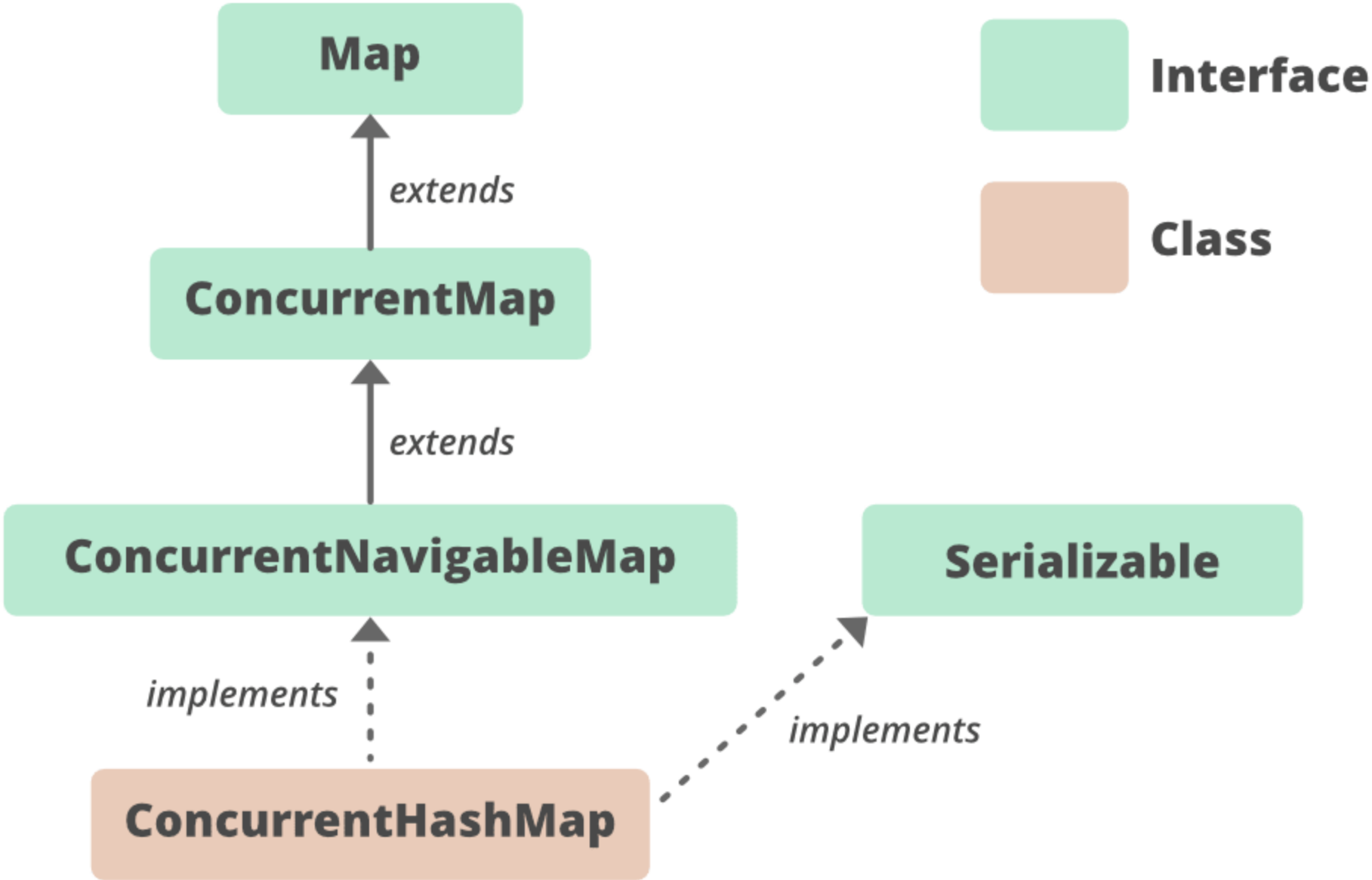
- Suppose two threads attempt to update the same key simultaneously:
- **Lock Acquisition:** Both threads try to lock the same bucket.
- **Conflict Resolution:** One thread acquires the lock first, updates the value, and releases the lock. The second thread then acquires the lock and proceeds.
- **Result:** Updates are applied sequentially, ensuring data consistency.

# Concurrency

- At a time any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.
- In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
- Default concurrency-level of ConcurrentHashMap is **16**.
- In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updations in the object, the thread must lock the particular segment in which the thread wants to operate.
- This type of locking mechanism is known as **Segment locking or bucket locking.**
- Hence at a time, 16 update operations can be performed by threads.

# ConcurrentHashMap<K, V>




Map
extends
ConcurrentMap
extends
ConcurrentNavigableMap
implements
ConcurrentHashMap
Serializable
implements
Interface
Class

# ConcurrentMap

- ConcurrentMap is an extension of the Map interface. It aims to provides a structure and guidance to solving the problem of reconciling throughput with thread-safety.
- By overriding several interface default methods, ConcurrentMap gives guidelines for valid implementations to provide thread-safety and memory-consistent atomic operations.
- Several default implementations are overridden, disabling the null key/value support:
- getOrDefault
- forEach
- replaceAll
- computeIfAbsent
- computeIfPresent
- compute
- merge

# ConcurrentMap

- The following APIs are also overridden to support atomicity, without a default interface implementation:
- putIfAbsent
- remove
- replace(key, oldValue, newValue)
- replace(key, value)

# Atomic Operations

- Atomic operations are indivisible actions that occur completely or not at all, ensuring thread safety without the need for explicit synchronization.
- In the context of ConcurrentHashMap, these operations allow multiple threads to interact with the map concurrently without corrupting the data.

# Why Atomic Operations Matter in ConcurrentHashMap

- When multiple threads access and modify a shared ConcurrentHashMap:

Consistency is Key:

- Without atomic operations, simultaneous updates could lead to race conditions, leaving the map in an inconsistent state.

Performance Boost:

- Atomic operations minimize locking overhead, improving throughput in concurrent applications.

Simplified Code:

- They reduce the complexity of handling synchronization manually, making your code cleaner and less error-prone.

# Atomic Operations

**Atomic Methods**

•putIfAbsent(K key, V value): Inserts the key-value pair only if the key is not already present.

•replace(K key, V oldValue, V newValue): Replaces the entry only if it is currently mapped to a specific value.

**Bulk Operations**

•forEach, reduce, search: Support for parallel operations using lambda expressions.

# putIfAbsent(K key, V value)

- **Purpose**: Inserts a key-value pair only if the key is not already present.
- **Atomicity**: The check for the key's presence and the insertion happen as a single atomic operation.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");

// This will not overwrite the existing value for "A"
map.putIfAbsent("A", "Avocado");

System.out.println(map.get("A")); // Output: Apple
```

# remove(Object key, Object value)

- **Purpose**: Removes the entry for a key only if it is currently mapped to a specified value.
- **Atomicity**: Ensures that the removal is based on an exact match of the key and value.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");

map.remove("A", "Apple"); // Returns true and removes the entry
map.remove("B", "Banana"); // Returns false if "B" doesn't map to "Banana"
```

# replace(K key, V oldValue, V newValue)

- **Purpose**: Replaces the entry for a key only if it is currently mapped to a specified value.
- **Atomicity**: The check and replacement are atomic.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");

map.replace("A", "Apple", "Apricot"); // Returns true and updates the value
System.out.println(map.get("A")); // Output: Apricot
```

# computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)

- **Purpose**: Computes the value using the given function and inserts it if the key is not already present.
- **Atomicity**: Ensures that the computation and insertion are atomic.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");

map.computeIfAbsent("B", k -> "Banana");
System.out.println(map.get("B")); // Output: Banana
```

# computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)

- **Purpose**: Updates the value for a key if it is present.
- **Atomicity**: The update function is applied atomically.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");
map.put("B", " Banana ");

map.computeIfPresent("B", (k, v) -> v + " Split");
System.out.println(map.get("B")); // Output: Banana Split
```

# compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)

•**Purpose**: Atomically updates the value for the key using the provided remapping function, regardless of its presence.
•**Atomicity**: The computation and update are atomic.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");
map.put("B", " Banana ");


map.compute("C", (k, v) -> (v == null) ? "Cherry" : v + " Pie");
System.out.println(map.get("C")); // Output: Cherry
map.compute("C", (k, v) -> v + " Tart");
System.out.println(map.get("C")); // Output: Cherry Tart
```

# merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)

- **Purpose**: Merges the specified value with the existing value using the remapping function.
- **Atomicity**: The merge operation is atomic.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("A", "Apple");

map.merge("D", "Date", (oldVal, newVal) -> oldVal + " & " + newVal);
System.out.println(map.get("D")); // Output: Date
map.merge("D", "Fig", (oldVal, newVal) -> oldVal + " & " + newVal);
System.out.println(map.get("D")); // Output: Date & Fig
```

# Compare-And-Swap (CAS) Mechanism:

- The foundation of these atomic operations is the CAS algorithm, which is an uninterruptible operation used to achieve synchronization.
- It works by comparing the current value at a memory location with an expected value and, only if they match, changing it to a new value.
- This whole process is atomic, so no two threads can perform a CAS operation on the same memory location simultaneously.

# ConcurrentHashMap<K, V>

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
map.put("A", 1);
map.put("B", 2);
map.put("C", 3);
System.out.println("Map size: " + map.size());
int valueA = map.get("A");
System.out.println("Value of A: " + valueA);
map.remove("B");
System.out.println("Map size: " + map.size());
```

```
Output:
Map size: 3
Value of A: 1
Map size: 2
```

```
// create an instance of ConcurrentHashMap
        ConcurrentHashMap<Integer, String> m = new ConcurrentHashMap<>();


        // Insert mappings using
        // put method
        m.put(100, "Hello");
        m.put(101, "Geeks");
        m.put(102, "Geeks");


        // Here we cant add Hello because 101 key
        // is already present in ConcurrentHashMap object
        m.putIfAbsent(101, "Hello");


        // We can remove entry because 101 key
        // is associated with For value
        m.remove(101, "Geeks");


        // Now we can add Hello
        m.putIfAbsent(103, "Hello");


        // We cant replace Hello with For
        m.replace(101, "Hello", "For");
        System.out.println(m);
```

**Output**{100=Hello, 102=Geeks, 103=Hello}

# computeIfAbsent()

```
// create a HashMap and add some values
        HashMap<String, Integer> mapcon
           = new HashMap<>();
        mapcon.put("k1", 100);
        mapcon.put("k2", 200);
        mapcon.put("k3", 300);
        mapcon.put("k4", 400);
        System.out.println("HashMap values :\n " + mapcon.toString());
        mapcon.computeIfAbsent("k5", k -> 200 + 300);
        mapcon.computeIfAbsent("k6", k -> 60 * 10);
        System.out.println("New HashMap after computeIfAbsent :\n "+ mapcon);
```

**Output:**

*HashMap values : {k1=100, k2=200, k3=300, k4=400}*

*New HashMap after computeIfAbsent : {k1=100, k2=200, k3=300, k4=400, k5=500, k6=600}*

```
public class WordCountExample {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<String, Integer> wordCounts = new ConcurrentHashMap<>();

        String[] words = {"apple", "banana", "orange", "apple", "banana", "apple"};

        ExecutorService executor = Executors.newFixedThreadPool(2);

        Runnable task = () -> {
            for (String word : words) {
                // Atomic operation to update the count
                wordCounts.merge(word, 1, Integer::sum);
            }
        };

        // Start two threads
        executor.execute(task);
        executor.execute(task);
        executor.shutdown();
        while (!executor.isTerminated()) {
            // Wait for all tasks to finish
        }
        System.out.println("Final Word Counts: " + wordCounts);
    } }
```

Final Word Counts: {orange=2, banana=4, apple=6}

# ConcurrentHashMap --Best Scenarios

- **Concurrency**: When multiple threads need to read and write data concurrently without locking the entire map.
- **Thread-Safe Operations**: Suitable for shared data structures in multi-threaded environments.
- **High-Load Servers:** Ideal for web servers, caching mechanisms, and real-time systems.
- **When Reads Far Outnumber Writes:** Optimal performance when the majority of operations are reads.

# Advantages and Disadvantages

- **Advantages**:
- **High Performance**: Due to fine-grained locking, it achieves high performance even under heavy concurrent access[1].
- **Atomic Operations**: Provides methods for atomic operations, useful for implementing complex concurrent algorithms[1].
- **Disadvantages**:
- **Higher Memory Overhead**: The fine-grained locking mechanism requires additional memory[1].
- **Complexity**: The locking mechanism can make the code more complex, especially for developers unfamiliar with concurrent programming[1].

# ConcurrentHashMap

- **Scalability:** Handles a high level of concurrency efficiently, making it ideal for multi-threaded applications.
- **Throughput:** Due to reduced contention, it offers better throughput compared to synchronized data structures like Hashtable.
- **Memory Footprint**
- **Overhead:** Slightly higher memory usage compared to HashMap because of additional structures (like tree bins and synchronization constructs).

# Additional Insights

- **Weakly Consistent Iterators**
- **Behavior:** Iterators reflect the state of the hash table at some point since the creation of the iterator but may not include all concurrent updates.
- **Use Cases:** Safe for concurrent access but should be used with understanding of their limitations.
- **Resilience to DOS Attacks**
- **Hash Collision Attacks:** The tree bin structure helps mitigate denial-of-service attacks that exploit hash collisions.

# When Not to Use ConcurrentHashMap

- **Single-Threaded Applications:** The overhead isn't justified where thread safety isn't a concern.
- **Need for Strong Consistency in Iteration:** If you require a snapshot of the map's state during iteration, consider using other mechanisms.

# Points to remember with concurrency

- Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove).
- Retrievals reflect the results of the most recently completed update operations holding upon their onset.
- For aggregate operations such as putAll and clear, concurrent retrievals may reflect insertion or removal of only some entries.
- Similarly, Iterators, Spliterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration.
  - Do not throw ConcurrentModificationException.
  - Iterators are designed to be used by only one thread at a time.
  - Results of aggregate status methods including size, isEmpty, and containsValue are typically useful only when a map is not undergoing concurrent updates in other threads.
  - Otherwise the results of these methods reflect **transient states** that may be adequate for monitoring or estimation purposes, but **not for program control.**

| Properties | Hashtable | ConcurrentHashMap |
|---|---|---|
| Creation | Map ht = new Hashtable(); | Map chm = new ConcurrentHashMap(); |
| Is Null Key Allowed ? | No | No |
| Is Null Value Allowed ? | No | No (does not allow either null keys or values) |
| Is Thread Safe ? | Yes | Yes, Thread safety is ensured by having separate locks for separate buckets, resulting in better performance. Performance is further improved by providing read access concurrently without any blocking. |
| Performance | Slow due to synchronization overhead. | Faster than Hashtable. ConcurrentHashMap is a better choice when there are **more reads than writes**. |
| Iterator | Hashtable uses enumerator to iterate the values of Hashtable object. Enumerations returned by the Hashtable keys and elements methods are not fail-fast. | Fail-safe iterator: Iterator provided by the ConcurrentHashMap is fail-safe, which means it will not throw **ConcurrentModificationException**. |

| Feature | HashMap | LinkedHashMap | TreeMap | ConcurrentHashMap |
| --- | --- | --- | --- | --- |
| **Structure** | Hash table | Hash table + Linked list | Red-Black tree | Segmented hash table |
| **Order** | No order | Insertion/access order | Natural/custom order | No order |
| **Performance (Avg)** | O① | O① | O(log n) | O① |
| **Thread-Safety** | Not thread-safe | Not thread-safe | Not thread-safe | Thread-safe |
| **Use Cases** | General-purpose | Order-preserving | Sorted data | Concurrent environments |
| **Example** | Fast access | LRU cache | Range queries | Shared caches |

# WeakHashMap

- type of map implementation provided by Java that uses weak references for its keys.
- Entries in a WeakHashMap can be garbage collected when the key is no longer in ordinary use, even if the map itself still holds a reference to the key.

# Internal Storage and Structure

**Weak References**

- Weak Reference: A weak reference is a type of reference that does not prevent its referent from being made eligible for garbage collection.
- Entry Removal: When a key becomes weakly reachable (i.e., no strong references to it exist), the entry associated with that key in the WeakHashMap is eligible for garbage collection.

**Structure**

- Hash Table: WeakHashMap internally uses a hash table similar to HashMap to store its entries.
- Reference Queue: It uses a ReferenceQueue to keep track of keys that have been garbage collected.
- When a key is garbage collected, its corresponding entry is enqueued in the reference queue. The map periodically checks this queue and removes entries that have been enqueued.

```
import java.util.WeakHashMap;

public class WeakHashMapExample1 {
    public static void main(String[] args) {
        WeakHashMap<String, String> weakMap = new
WeakHashMap<>();
        String key1 = new String("key1");
        String value1 = "value1";

        weakMap.put(key1, value1);
        System.out.println("Before GC: " + weakMap);

        key1 = null; // Make the key eligible for garbage collection
        System.gc(); // Request garbage collection

        System.out.println("After GC: " + weakMap);
    }
}
```

Before GC: {key1=value1} After GC: {}

```
public class WeakHashMapExample2 {
    private WeakHashMap<Integer, String> cache = new WeakHashMap<>();

    public String getData(Integer key) {
        String value = cache.get(key);
        if (value == null) {
            value = "Data for key " + key;
            cache.put(key, value);
        }
        return value;
    }
    public static void main(String[] args) {
        WeakHashMapExample2 example = new WeakHashMapExample2();
        Integer key1 = new Integer(1);
        Integer key2 = new Integer(2);
        System.out.println(example.getData(key1));
        System.out.println(example.getData(key2));
        key1 = null; // Make key1 eligible for garbage collection
        System.gc(); // Request garbage collection
        // After GC, key1 should be removed from the cache
        System.out.println(example.cache);
    } }
```

```
Data for key 1
Data for key 2
{2=Data for key 2}
```

# Use Cases

- **Caching**: WeakHashMap is useful for caching scenarios where you want entries to be automatically removed when they are no longer referenced elsewhere in the application. This helps in managing memory efficiently and avoids memory leaks.
- **Metadata Storage**: It can be used to store metadata associated with objects, where the metadata should be removed when the object is no longer in use.
- **Listeners Management**: Managing listeners or callbacks that should be automatically removed when the associated object is no longer referenced.

| Feature | HashMap | WeakHashMap |
|---|---|---|
| **Reference Type** | Strong references | Weak references |
| **Garbage Collection** | Entries are not automatically garbage collected based on key references | Entries are eligible for garbage collection when keys are no longer in use |
| **Use Case** | General-purpose map implementation | Caching, metadata storage, scenarios where automatic entry removal is desired |
| **Performance** | Generally faster due to lack of reference tracking | Slightly more overhead due to weak references and periodic cleanup |
| **Memory Management** | Requires explicit management of entries to avoid memory leaks | Helps in managing memory by allowing unused entries to be garbage collected |
| **Implementation** | Uses a standard hash table implementation | Uses a hash table with weak references for keys |
| **Entry Removal** | Entries remain in the map unless explicitly removed | Entries are automatically removed when keys are garbage collected |
| **Thread Safety** | Not thread-safe by default; requires external synchronization | Not thread-safe by default; requires external synchronization |
| **Example Usage** | HashMap<String, Integer> map = new HashMap<>(); | WeakHashMap<String, Integer> weakMap = new WeakHashMap<>(); |
| **Memory Overhead** | Lower memory overhead compared to WeakHashMap | Slightly higher memory overhead due to reference queue and weak references |

# How Garbage Collection Works in WeakHashMap

- **Weak References for Keys**:
- The keys in a WeakHashMap are stored as weak references. A weak reference is a reference that does not prevent its referent (the object it points to) from being made eligible for garbage collection.
- **Reference Queue**:
- WeakHashMap uses a ReferenceQueue to keep track of keys that have been garbage collected. When a key is garbage collected, the weak reference to that key is enqueued in this reference queue.
- **Periodic Cleanup**:
- WeakHashMap periodically checks the reference queue for entries. When it finds that a key has been enqueued, it means that the key has been garbage collected, and the corresponding entry can be safely removed from the map.

# How Garbage Collection Works in WeakHashMap

1. **Entry Insertion**:
    1. When a new entry is added to the WeakHashMap, a weak reference to the key is created and associated with the value. The entry is then inserted into the hash table.
2. **Key Becomes Weakly Reachable**:
    1. At some point, if there are no strong references to a key, it becomes weakly reachable. This means that the key is only reachable through weak references.
3. **Garbage Collection**:
    1. During the garbage collection process, the garbage collector identifies the weakly reachable keys and reclaims the memory used by those keys. The weak references to these keys are enqueued in the reference queue.
4. **Reference Queue Processing**:
    1. The WeakHashMap periodically processes the reference queue. It checks for any weak references that have been enqueued.
    2. For each enqueued reference, the corresponding entry is removed from the hash table.

# Benefits of WeakHashMap

- **Automatic Cleanup**:
  - Entries are automatically removed when keys are no longer in use, preventing memory leaks.
- **Memory Efficiency**:
  - Helps manage memory more efficiently by allowing unused entries to be garbage collected.
- **Simplicity**:
  - Simplifies code by eliminating the need for manual cleanup of entries.

| Feature | HashMap | WeakHashMap | Hashtable | ConcurrentHashMap | LinkedHashMap |
|---|---|---|---|---|---|
| **Reference Type** | Strong references | Weak references | Strong references | Strong references | Strong references |
| **Garbage Collection** | No automatic entry removal | Entries removed when keys are no longer in use | No automatic entry removal | No automatic entry removal | No automatic entry removal |
| **Thread Safety** | Not thread-safe | Not thread-safe | Thread-safe (synchronized) | Thread-safe (lock-free) | Not thread-safe |
| **Performance** | High performance in single-threaded contexts | Slightly higher overhead due to weak references and cleanup | Slower due to synchronization | High performance in multi-threaded contexts | |

| Feature | HashMap | WeakHashMap | Hashtable | ConcurrentHashMap | LinkedHashMap |
|---|---|---|---|---|---|
| **Ordering** | No ordering | No ordering | No ordering | No ordering | Maintains insertion order |
| **Use Case** | General-purpose, single-threaded | Caching, metadata storage, automatic entry removal | Thread-safe applications | High-concurrency applications | Maintaining insertion order |
| **Memory Efficiency** | Requires explicit management | More memory efficient | Requires explicit management | Requires explicit management | Requires explicit management |
| **Entry Removal** | Explicit removal required | Automatic removal when keys are garbage collected | Explicit removal required | Explicit removal required | Explicit removal required |

| Feature | HashMap | WeakHashMap | Hashtable | ConcurrentHashMap | LinkedHashMap |
|---|---|---|---|---|---|
| **Reference Type** | Strong references | Weak references | Strong references | Strong references | Strong references |
| **Garbage Collection** | No automatic entry removal | Entries removed when keys are no longer in use | No automatic entry removal | No automatic entry removal | No automatic entry removal |
| **Thread Safety** | Not thread-safe | Not thread-safe | Thread-safe (synchronized) | Thread-safe (lock-free) | Not thread-safe |
| **Performance** | High performance in single-threaded contexts | Slightly higher overhead due to weak references and cleanup | Slower due to synchronization | High performance in multi-threaded contexts | |