**Data Structure Implementation – Hashing**

**Part 1 – Hashing Data Structures:**

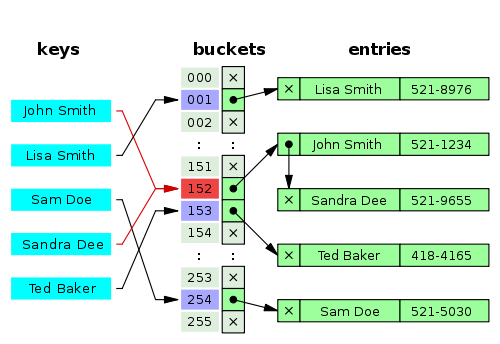
|  |  |
| --- | --- |
| **Data Structure** | **Characteristics** |
| Map | * General abstract class that is extended by Hash Map, Hash Set and Hash Table. |
| Hash Set | * Underlying data structure for HashSet is hashmap. * As it implements the Set Interface, duplicate values are not allowed. * Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code * Does not maintain insertion order. The order of the the hash map items is not about which was inserted first (like in array list) but it depends on their keys (and more specifically, which bucket they are). * When two elements end up in the same bucket, the first one that was inserted will also be the first one returned during iteration. |
| Hash Map | * HashMap is non synchronized. It is not-thread safe and can’t be shared between many threads without proper synchronization code whereas * Allows one nul key and multiple null values * Generally, better to use hash map for performance if you don’t need thread synchronization. |
| Hash Table | * Hashtable is synchronized. It is thread-safe and can be shared with many threads. * Doesn’t allow null key or null values |
| Linked Hash Set | * Exactly like a hash set but will maintain insertion order. |

**Note:** HashTable and HashMap are almost the same and are used interchangeably. The only differences they have in Java are the differences above.

**Part 2 – Hashing Set Implementation:**

|  |
| --- |
| **Hash Set Implementation:**  **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>{  **private** **transient** HashMap<E,Object> map;  **private** **static** **final** Object *PRESENT* = **new** Object();    **public** HashSet() {  map = **new** HashMap<E,Object>();  }    **public** Iterator<E> iterator() {  **return** map.keySet().iterator();  }    **public** **int** size() {  **return** map.size();  }    **public** **boolean** isEmpty() {  **return** map.isEmpty();  }    **public** **boolean** contains(Object o) {  **return** map.containsKey(o);  }    **public** **boolean** add(E e) {  **return** map.put(e, *PRESENT*)==**null**;  }    **public** **boolean** remove(Object o) {  **return** map.remove(o)==*PRESENT*;  }    **public** **void** clear() {  map.clear();  }  }  **Explanation:** Hash Set is really just a hash map. For example, if you declare a hash set of strings, it creates a hashmap of <String, Object>. Every time you add an item to the hash set, it puts your item as key and PRESENT as the value in the hash map. When you call “contains”, it calls “containsKey” and looks for your string. Iterator returns an iterator of all the keys in the hashmap. |

**Part 3.1 – Hash Map High Level Design & Implementation:**



So the way hashing works is that you have an array list of buckets. Each bucket contains a linked list of hash map nodes. So suppose you wanted to search the employee number of “John Smith”. You would pass in the key “John Smith”. This key would get you a bucket index which in this case is ‘152’. You would go into your array list of buckets and get your bucket head for this list (i.e. bucketList.get(152)). This would give you the bucket head which is the head of a linked list of hashmap nodes in a particular bucket. You would iterate through that bucket until you find a hashmap node with the key “John Smith”. In this case, the first bucket item (the bucket head) is the one with the matching key.

There is a method called “ get bucket index” that takes in a key object and returns a bucket index in an integer. You can see the implementation below of how it does this.

A collision occurs when you try and put an object in the map (i.e. put“Sandra Dee”, “521-9655”), and there is already a hashmap node (the one with the key “john smith”) already there. In this case you have to insert the new hashmap node in the bucket list.

|  |
| --- |
| **Part 3.2 – HashMap Class Design:**  **public** **class** HashMap<K,V> {  **private** ArrayList<HashNode<K, V>> buckets = **new** ArrayList<>();  **private** **static** **final** **double** *MAX\_LOAD\_FACTOR* = 0.7;  **private** **static** **final** **int** *INIT\_NUM\_BUCKETS* = 10;  **public** **int** numOfBuckets = 0;  **int** size;  **public** HashMap(){  **for**(**int** i = 0; i < *INIT\_NUM\_BUCKETS*; i++){  buckets.add(**null**);  numOfBuckets++;  }  }  **public** **int** size(){}  **public** **boolean** isEmpty(){}  **private** **int** getBucketIndex(K key){ }  **public** HashNode<K, V> getBucketHead(K key){ }  **public** **boolean** needMoreBuckets(){}  **public** **void** updateBucketList(){}  **public** **void** doubleBucketListSize(){}  **public** **boolean** keyEquals(K nodeKey, K key){}  **public** V get(K key){}  **public** V removeBucketHead(HashNode<K, V> bucketHead){}  **public** V removeNodeFromBucket(HashNode<K, V> bucketHead, K key){}  **public** V remove(K key){}  **public** **void** initializeBucketHead(HashNode<K, V> addItem){}  **public** **void** prependToBucket(HashNode<K, V> bucketHead, HashNode<K, V> addItem){}  **public** **void** putNodeInBucket(K key, V value){}  **public** **boolean** collision(K key){ }  **public** **void** put(K key,V value) { }  } |

|  |  |
| --- | --- |
| **Method** | **High Level Description** |
| size() | Returns the size of the hashmap stored in the class attribute size. |
| isEmpty() | Returns true if the hashmap is empty. |
| getBucketIndex() | Given the key as an input paramter, it returns the integer index of where its corresponding bucket is in the array list of buckets. |
| getBucketHead() | Given the key as an input paramter, it returns the head of the bucket (which is a linked list type data structure) of where the lookup value is located. |
| needMoreBuckets() | Determines whether or not there is a shortage of buckets based on the number of elements in the hashmap. The load factor should not be greater than the maximum acceptable load factor. |
| doubleBucketListSize() | Doubles the size of the bucket list, while filling these new indexes with null values. |
| updateBucketList() | Decides whether or not there is a need for more buckets. If there is a need, it will double the bucket list. |
| keyEquals() | Determines if the two keys in the paramter are equal. |
| get() | This is the classical hashmap get that is being implemented. |
| removeBucketHead() |  |
| removeNodeFromBucket() |  |
| remove() | This is the classical hashmap remove that is being implemented. |
| initializeBucketHead() | This is for cases when you want to put a key-value pair in the map and this key does not have a collision. In this case, you just initialize the bucket head to this hash map node. |
| putNodeInBucket() | This is for cases when you want to put a key-value pair in the map and this key does have a collision. In this case, there are two possible outcomes.   1. The key of the hash map node you are trying to add already exists in the hash map. You must iterate through the bucket and find this node and update its value to the new value. 2. The key does not exist. Prepend this hashmap node to the bucket so that it is the bucket head. |
| collision() | Given the in put paramter ‘key’ it checks to see if there is an upcoming collision. A collision is when the key of the object your are trying to add has a bucket index that already contains an item. |
| put() | This is the classical hashmap put that is being implemented. |

|  |
| --- |
| **Part 3.3 - Hash Map Implementation:**  **public** **class** HashMap<K,V> {  **private** ArrayList<HashNode<K, V>> buckets = **new** ArrayList<>();  **private** **static** **final** **double** *MAX\_LOAD\_FACTOR* = 0.7;  **private** **static** **final** **int** *INIT\_NUM\_BUCKETS* = 10;  **public** **int** numOfBuckets = 0;  **int** size;  **public** HashMap(){  **for**(**int** i = 0; i < *INIT\_NUM\_BUCKETS*; i++){  buckets.add(**null**);  numOfBuckets++;  }  }  **public** **int** size(){  **return** size;  }  **public** **boolean** isEmpty(){  **return** size == 0;  }  **private** **int** getBucketIndex(K key){  **return** key.hashCode() % numOfBuckets;  }  **public** HashNode<K, V> getBucketHead(K key){  **int** bucketIndex = getBucketIndex(key);  HashNode<K, V> bucketHead = buckets.get(bucketIndex);  **return** bucketHead;  }  **public** **boolean** needMoreBuckets(){  **double** loadFactor = (**double**) size / numOfBuckets;  **return** loadFactor >= *MAX\_LOAD\_FACTOR*;  }  **public** **void** updateBucketList(){  **if**(needMoreBuckets()){  doubleBucketListSize();  }  }  **public** **void** doubleBucketListSize(){  **for**(**int** i = 0; i < numOfBuckets; i++){  buckets.add(**null**);  }  numOfBuckets = numOfBuckets \* 2;  }  **public** **boolean** keyEquals(K nodeKey, K key){  **return** nodeKey.equals(key);  }  **public** V get(K key){  HashNode<K, V> bucketHead = getBucketHead(key);  HashNode<K, V> currentNode = bucketHead;  **while**(currentNode != **null**){  **if**(keyEquals(currentNode.key, key)){  **return** currentNode.value;  }  currentNode = currentNode.next;  }  **return** **null**;  }  **public** V removeBucketHead(HashNode<K, V> bucketHead){  **int** index = getBucketIndex(bucketHead.key);  buckets.set(index, bucketHead.next); //set new bucket head in bucket list  size--;  **return** bucketHead.value;  }  **public** V removeNodeFromBucket(HashNode<K, V> bucketHead, K key){  HashNode<K, V> previous = bucketHead;  HashNode<K, V> currentNode = bucketHead.next;//already checked head  **while**(currentNode!= **null**){  **if**(keyEquals(currentNode.key, key)){  previous.next = currentNode.next; // remove from bucket list  size--;  **return** currentNode.value;  }  previous = currentNode;  currentNode = currentNode.next;  }  **return** **null**;  }  **public** V remove(K key){  HashNode<K, V> bucketHead = getBucketHead(key);  **if**(bucketHead == **null**){  **return** **null**;  }**else** **if**(keyEquals(bucketHead.key, key)){  **return** removeBucketHead(bucketHead);  }**else**{  **return** removeNodeFromBucket(bucketHead, key);  }  }  **public** **void** initializeBucketHead(HashNode<K, V> addItem){  **int** index = getBucketIndex(addItem.key);  buckets.set(index, addItem);  size++;  }    **public** **void** prependToBucket(HashNode<K, V> bucketHead, HashNode<K, V> addItem){  addItem.next = bucketHead;  **int** index = getBucketIndex(addItem.key);  buckets.set(index, addItem);  size++;  }  **public** **void** putNodeInBucket(K key, V value){  HashNode<K, V> bucketHead = getBucketHead(key);  HashNode<K, V> currentNode = bucketHead;  **while**(currentNode!= **null**) {  **if**(keyEquals(currentNode.key,key)) {  currentNode.value = value;//update its value  **return**;  }  currentNode = currentNode.next;  }  HashNode<K, V> addItem = **new** HashNode<K, V>(key, value);  prependToBucket(bucketHead, addItem);  }  **public** **boolean** collision(K key){  HashNode<K, V> bucketHead = getBucketHead(key);  **return** bucketHead != **null**;  }  **public** **void** put(K key,V value) {  HashNode<K, V> addItem = **new** HashNode<> (key, value);  **if**(!collision(key)) {  initializeBucketHead(addItem);  } **else** {  putNodeInBucket(key, value);  }  updateBucketList();  }  } |

**Complexity Analysis:**

If you look at the implementation of the hash map, all operations have guaranteed constant time O(1) except for methods that try and search through a particular bucket. These methods are get(),putNodeInBucket(),removeNodeFromBucket(). All these methods iterate over a bucket (for cases when multiple hash map nodes are sititng in the same bucket). If we made the settings such that there was only one bucket, then all items would be in the same bucket and the hashmap would really be just a linked list of hash map nodes. This would make or search and access methods cost us O(n). Even our delete and insert would cost us O(n) because we need to still search for the hashmap node with the matching key to delete or to update. This means our worst case is O(n).

However, since we regulate the load factor to be within a specific range, there will always be more buckets than there are hashmap nodes. This means that realistically, the size of any bucket will be low (probably less than two) and we can assume that searching through a bucket (which is a linked list) linearly will still be constant time because the elements in it are less than two in almost all cases. However, this is constant time for the average case and the best case. The worst case could still be O(n). This happens if you design the hash map in such a way where all items are in one bucket. You could do this by adjusting the minimum load factor settings, or specfically adding elements such that they will always cause collisions, or maybe override that hashcode() method to return the same values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Time Complexity –Best Case** | **Time Complexity – Worst Case** | **Space Complexity** |
| *Remove/Deletion* | *O(1)* | *O(n)* | *O(n)* |
| *Put/Insertion* | *O(1)* | *O(n)* |
| *Get/Searching* | *O(1)* | *O(n)* |
| *Access* | *O(1)* | *O(n)* |