**Hash Table**

**Dictionary [words -** keys, **values** **-** explanations**]**

|  |
| --- |
| **Dictionary** is a data structure, which is an association of keys [unique] with values [not unique].  **Operations**: bind a value to a key, delete a key and lookup for a value by the key. |

**Hash Table**

|  |
| --- |
| **Hash table** (hash map) is an implementation of [dictionary ADT](http://www.algolist.net/Data_structures/Dictionary_(ADT)). It **maps** unique keys to associated values. It is an **array** that uses **hash function** to convert the key into index of an array element, where associated value is to be sought. |

**Hash function**

|  |
| --- |
| **Hash function** is good, if it provides uniform distribution of hash values. The poor hash functions cause collisions and some other unwanted effects, which badly affect hash table overall performance. |

**Load Factor**

|  |
| --- |
| The **load factor** is the ratio between the number of stored items and array's size. Hash table can whether be of a constant size or being dynamically resized, when load factor exceeds some threshold. Resizing is done before the table becomes full to keep the number of collisions under certain amount and prevent performance degradation. |

**Collisions/Type of Collision Resolution**

|  |
| --- |
| **Collision** occurs if hash function returns the same hash value for different keys. Collisions are unavoidable. Due to collisions, keys are also stored in the table, so one can distinguish between key-value pairs having the same hash. |
| **Closed addressing - chaining [open hashing – memory waste but efficient].**Each slot of the hash table contains a link to another data structure (linked list), which stores key-value pairs with the same hash. When collision occurs, this data structure is searched for key-value pair, which matches the key. |
| **Open addressing – linear probing, quadratic probing, double hashing [closed hashing – no memory waste but inefficient].** Each slot actually contains a key-value pair. When collision occurs, open addressing algorithm calculates another location (i.e. next one) to locate a free slot. Hash tables, based on open addressing strategy experience drastic performance decrease, when table is tightly filled (load factor >= 0.7). |

**Complexity analysis**

|  |
| --- |
| **Space complexity**: time linearly depends on table's load factor. The computational complexity of both singly-linked list and constant-sized hash table is **O(n)** [**the best/worst/average case]** |
| **Time complexity: O(n)** **[the worst case]**  If hash function is goodand hash table is well dimensioned, the operation time is **constant O(1)** [**best/average case]**. |

**Probing vs. Chaining**

|  |
| --- |
| **Linear probing**: if the slot, indicated by hash function, has already been occupied, algorithm tries to find an empty one by probing consequent slots in the array. When load factor > 0.7, hash table performance will decrease nonlinearly.  **Linear probing:** distance between probes is constant (i.e. 1, when probe examines consequent slots); |
| **Quadratic probing:**distance between probes increases by certain constant at each step. |
| **Double hashing:**distance between probes is calculated using another hash function. |
| **Chaining**: each slot of the array contains a link to a [singly-linked list](http://www.algolist.net/Data_structures/Singly-linked_list) containing key-value pairs with the same hash. New key-value pairs are added to the end of the list. Lookup algorithm searches through the list to find matching key. Initially table slots contain nulls. List is being created, when value with the certain hash is added for the first time. |

**Class LinkedHashEntry**

|  |
| --- |
| **class** LinkedHashEntry  {  **private** **int** key;  **private** **int** value;  **private** LinkedHashEntry next;    **public** LinkedHashEntry(**int** key, **int** value)  {  **this**.key = key;  **this**.value = value;  next = **null**;  }    **public** **void** setValue(**int** value)  {  **this**.value = value;  }    **public** **void** setNext(LinkedHashEntry next)  {  **this**.next = next;  }    **public** **int** getKey()  {  **return** key;  }    **public** **int** getValue()  {  **return** value;  }    **public** LinkedHashEntry getNext()  {  **return** next;  }  } |

**Class HashMap**

|  |
| --- |
| **class** HashMap  {  **private** **final** **static** **int** *TABLE\_SIZE* = 5;  **private** LinkedHashEntry[] table;    **public** HashMap()  {  table = **new** LinkedHashEntry[*TABLE\_SIZE*];  **for**(**int** i = 0; i < *TABLE\_SIZE*; i++)  table[i] = **null**;  }  >>>>>*THE REST OF THE METHODS GO HERE*<<<<<  **}** |

**The get Method**

|  |
| --- |
| **public** **int** get(**int** key)  {  **if**(key < 0 || key >= table.length)  **throw** **new** IndexOutOfBoundsException();  **int** hash = (key % *TABLE\_SIZE*);  **if** (table[hash] == **null**)  **return** -1;  LinkedHashEntry entry;  **for**(entry = table[hash]; entry != **null** && entry.getKey() != key; entry = entry.getNext())  entry = entry.getNext();  **if** (entry == **null**)  **return** -1;  **else**  **return** entry.getValue();  } |

**The put Method**

|  |
| --- |
| **public** **void** put(**int** key, **int** value)  {  **if**(key < 0 || key >= table.length)  **throw** **new** IndexOutOfBoundsException();  **int** hash = (key % *TABLE\_SIZE*);  **if** (table[hash] == **null**)  table[hash] = **new** LinkedHashEntry(key, value);  **else**  {  LinkedHashEntry entry;  **for**(entry = table[hash]; entry != **null** && entry.getKey() != key; entry = entry.getNext())  entry = entry.getNext();  **if** (entry.getKey() == key)  entry.setValue(value);  **else**  entry.setNext(**new** LinkedHashEntry(key, value));  }  } |

**The remove Method**

|  |
| --- |
| **public** **void** remove(**int** key)  {  **int** hash = (key % *TABLE\_SIZE*);  **if** (table[hash] == **null**)  **return**;  LinkedHashEntry prevEntry = **null**;  LinkedHashEntry entry = table[hash];  **while** (entry.getNext() != **null** && entry.getKey() != key)  {  prevEntry = entry;  entry = entry.getNext();  }    **if** (entry.getKey() == key)  **if** (prevEntry == **null**)  table[hash] = entry.getNext();  **else**  prevEntry.setNext(entry.getNext());  } |

**The display Method**

|  |
| --- |
| **public** **void** displayTable()  {  **for**(**int** i = 0; i < *TABLE\_SIZE*; i++)  **if**(table[i] == **null**)  System.*out*.println(i + ". " + table[i]);  **else**  System.*out*.println("Key: " + table[i].getKey() + " Value: " + table[i].getValue());  } |

**The main() Method**

|  |
| --- |
| **public** **class** PointApp  {  **public** **static** **void** main(String[] args)  {  HashMap map = **new** HashMap();  map.put(0, 4); map.put(1, 3); map.put(2, 2); map.put(3, 1); map.put(4, 0);  map.displayTable();  System.*out*.println(map.get(2));  map.remove(0); map.remove(4);  map.displayTable();  }  } |