# HASH

hash table stores data in (k, v) form

key is unique

values can/will repeat

instead of a key, use a hash

every hash function has two parts:

Hash Code, Compressor

Hash code 🡪 number

in java, every object has it’s own hash code

compressor 🡪 % (modulo)

Hash collision:

two or more keys resulting in same hash

Separate Chaining Technique, Open addressing, Double Hashing

Load Factor:

total number of buckets we decided to fill initially

key-value pair

|  |  |  |
| --- | --- | --- |
| ‘go’ | 456 |  |
| ‘hello’ | 789 |  |
| ‘hi’ | 234 |  |
| ‘Friday’ | 111 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| *0* |  |  |  |
| *1* |  |  |  |
| *2* | node(hello) |  |  |
| *3* |  |  |  |
| *4* | node(go) | node(abc) |  |
| *5* |  |  |  |
| *6* |  |  |  |
| *7* |  |  |  |
| *8* |  |  |  |
| *9* | node(hi) |  |  |

**package** valentine;

**import** java.util.ArrayList;

**public** **class** HashTable <K, V> {

// store array of chain (bucket)

**private** ArrayList<HashNode<K, V>> chainArray;

//capacity

**private** **int** numChains;

// current size of array List

**private** **int** size;

**public** HashTable() {

chainArray = **new** ArrayList<>();

numChains = 10;

size = 0;

//create empty chain, equating all to NULL

**for**(**int** i=0; i<numChains; i++) {

chainArray.add(**null**);

}

}

//function to get hash (index) for a key

// using hashCode%compressor

**private** **int** getBucketIndex(K key){

**int** hc = key.hashCode();

**int** i = hc % numChains; // we have numChains = 10 intially

System.***out***.println("for key = "+key+" hashcode = "+hc+" index = "+i);

**return** i;

}

**public** **void** put(K key, V value)

{

// Find head of chain for given key

**int** bucketIndex = getBucketIndex(key);

HashNode<K, V> head = chainArray.get(bucketIndex);

// Check if key is already present

**while** (head != **null**)

{

**if** (head.key.equals(key))

{

head.value = value;

**return**;

}

head = head.next;

}

// Insert key in chain

size++;

head = chainArray.get(bucketIndex);

HashNode<K, V> newNode = **new** HashNode<K, V>(key, value);

newNode.next = head;

chainArray.set(bucketIndex, newNode);

// If load factor crosses max value then increase hash table size

// code not written yet

}

**public** V remove(K key){

//find the hash index for given key

**int** bucketIndex = getBucketIndex(key);

//get head of chain

HashNode<K,V> head = chainArray.get(bucketIndex);

//search for key in the above retrieved chain

HashNode<K, V> prev = **null**;

**while**(head != **null**) {

**if**(head.key.equals(key)) {

**break**;

}

prev = head;

head = head.next;

}

**if**(head == **null**) {

**return** **null**;

}

size--;

**if**(prev != **null**) {

prev.next = head.next;

}

**else** {

chainArray.set(bucketIndex, head.next);

}

**return** head.value;

}

**public** V get(K key)

{

// Find head of chain for given key

**int** bucketIndex = getBucketIndex(key);

HashNode<K, V> head = chainArray.get(bucketIndex);

// Search key in chain

**while** (head != **null**)

{

**if** (head.key.equals(key))

**return** head.value;

head = head.next;

}

// If key not found

**return** **null**;

}

**public** **void** getSize(){

}

**public** **int** size() {

**return** size;

}

**public** **boolean** isEmpty(){

**if** (size()==0) {

**return** **true**;

}

**else** {

**return** **false**;

}

}

**public** **static** **void** main(String[] args)

{

HashTable<String, Integer>ht = **new** HashTable<>();

ht.put("go",456 );

ht.put("hello",789 );

ht.put("hi",234 );

ht.put("abc", 777);

// ht.put("go", 888);

// System.out.println(ht.size());

// System.out.println(ht.remove("hello"));

// System.out.println(ht.remove("hello"));

// System.out.println(ht.size());

// System.out.println(ht.isEmpty());

}

}