DSA

Module 8

Hashing

Author
Srinivas Dande

8. Hashing

- **Hashing** is a technique used to store and retrive data as quickly as possible.
- Hashing is used to perform Optimal Searches
- Consider the following two Use-Cases.

Use-Case 1:

- Consider we have an array of integer numbers.
- I want to perform the following 3 Operations on the Array

Operation	Time Complexity
Search the number in the Array	O(n)
Insert the number in the Array	O(n)
Delete the number from the Array	O(n)

Use-Case 1:

- Consider we have an Binary Search Tree(BST).
- I want to perform the following 3 Operations on BST

Operation	Time Complexity
Search the number in the BST	O(log n)
Insert the number in the BST	O(log n)
Delete the number from the BST	O(log n)

- Hashing works **best for these 3 Operations** and beats all remaining Data Structures.
- Hashing **provides O(1) average time complexity** for 3 Important Operations - Search , Insert , Delete

- Hashing does exact key search
- Hashing is not useful when you want to find closet values or prefix searching etc.

8.1. Applications of Hashing

1) Dictionaries/Map
2) Sets
3) Database Indexing
4) Symbol Tables in Compilers
5) ryptography
6) Password Verification (MD5,SHA256)
7) Robin-Karp Algorithm

   Etc

8.2. Direct Address Table

- Consider we have 25 keys with the values ranging from 0 to 24.
- How can we implement the following 3 operations in O(1) time complexity?
  - a) Search
  - b) Insert
  - c) Delete

Lab1.java

```
package com.jlcindia.hashing1;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
class MyTable {
	int mytable[];
	int capacity;

	public MyTable(int capacity) {
		this.capacity = capacity;
		mytable = new int[capacity];
	}

	public void insert(int key) {
		mytable[key]++;
	}
```

```
	public void delete(int key) {
		mytable[key]--;
	}
	public int search(int key) {
		return mytable[key];
	}
	public void show() {
		for(int i=0;i<capacity;i++) {
			if(mytable[i]!=0) {
				System.out.println(i);
			}
		}
	}
}
public class Lab1 {
	public static void main(String[] args) {
		MyTable mytable = new MyTable(25);
		mytable.insert(5);
		mytable.insert(2);
		mytable.insert(7);
		mytable.insert(9);
		mytable.insert(0);

		mytable.show();

		mytable.delete(0);
		mytable.delete(2);

		System.out.println("After Delete--");
		mytable.show();
		System.out.println("----------");

		System.out.println(mytable.search(5));
		System.out.println(mytable.search(2));
	}
}
```

Problems with above solution:

- It does not handle the following
    a) Large Numbers
    b) Negative Numbers
    c) Floating Point numbers
    d) Strings
    e) Addresses (Any Object Address)

- We can use Hashing to solve the above problems

8.3. Exploring Hashing

- Hashing is a technique used to store and retrive keys as quickly as possible
- Keys can be any of the following
    a) Large Numbers
    b) Negative Numbers
    c) Floating Point numbers
    d) Strings
    e) Addresses (Any Object Address)

- You can have Universe of Keys -
- You can store the keys in Hashtable by converting the keys to small values.

8.3.1. Hash Functions

- Hash function is used to transform the key into the index.

- Ideally, **Hash Function** should map each possible key to a unique slot index, but it is difficult to achieve in practice.

- Given a collection of elements, a hash function that maps each item into a unique slot is referred to as a perfect hash function

8.3.2. Hash Function Examples

1) for numbers

- **hash(key) = key % m**
- where m is table size (consider prime number)

2) for Strings

- str = "abc"
- hash(key) = (str[0] * x^0 + str[1] * x^1 + str[2] * x^2+ ...) % m

3) Universal Hashing:

- Group of Hash Functions and Pick one Randomly.

Example-1

keys = {53,51,55,52,50,54,49} **hash(key) = key %7**

Example-2

keys = { 53,51,55,52,50,54,49} **hash(key) = key %11**

8.3.3. Load Factor

Load Factor = Number of items stored in the table / Size of the table

Ex:

Load Factor = 7/11 => 0.66

- This is the decision parameter used when we want to rehash or expand the existing hash table entries.
- This also helps us in determining the efficiency of the hashing function.
- That means, it tells whether the hash function is distributing the keys uniformly or not.

8.3.4. Characteristics of Good Hash Functions

- A good hash function should have the following characteristics:

    1) Should generate same value every time for the given key.
    2) Should generate the values from 0 to m-1
    3) Should generate fast - O(1) for Integers and O(len) for Strings.
    4) Should uniformly distribute Large keys into Hashtable slots.
    5) Minimize collision
    6) Have a Low load factor for a given set of keys

8.4. Collisions

- Hash functions are used to map each key to a different Hashtable slots, but practically it is not possible to create such a hash function and the problem is called collision.

- Collision is the condition where two or more keys are stored in the Same Hashtable slots.

Collision Handling:

- If you know the keys in Advance then we can do the hashing perfectly (called Perfect Hashing).
- Collision is bound to happen if you dont know the keys in advance.
- **The process of finding an Alternative Hashtable slot is called collision resolution.**
- Even though hash tables have collision problems, they are more efficient in many cases compared to all other data structures, like search trees.
- There are a number of collision resolution techniques, and the most popular are **direct chaining** and **open addressing**.

A) Direct Chaining:

- Separate Chaining or Chaining

B) Open Addressing:

- Linear probing
- Quadratic probing
- Double hashing

8.5. Chaining

- When two or more Keys with same hash goes to same hashtable slot, these Keys are stored into a Separate Data Structure called a chain.
- Data Structures for Storing Chains:
  a) Linked List
  b) ArrayList
  c) Self Balanced BST

a) Linked List

✓ Not Cache Friendly ✓ Extra Space for Node Representation	Insert delete Search	->->->	O(l) O(l) O(l)

b) ArrayList

✓ Cache Friendly ✓ No Extra Space	Insert delete Search	->->->	O(l) O(l) O(l)

c) Self Balanced BST

✓ Not Cache Friendly ✓ Used from Java8	Insert delete Search	->->->	O(log l) O(log l) O(log l)

Example-1

keys = { 50, 21, 58, 17, 15, 49, 56, 22, 23, 25} **hash(key) = key % 7**

Lab2.java

```
package com.jlcindia.hashing2;

/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

class MyHashSet {

	int bucket;
	List<LinkedList<Integer>> mytable;

	public MyHashSet(int bucket) {
		this.bucket = bucket;
		this.mytable = new ArrayList<LinkedList<Integer>>(bucket);

		for(int i=0;i<bucket;i++) {
			mytable.add(new LinkedList<Integer>());
		}
	}

	public void insert(int key) {
		int index = key%bucket;
		if( !search(key))
			mytable.get(index).add(key);
	}
```

```
	public void delete(int key) {
		int index = key%bucket;
		mytable.get(index).remove((Integer)key);
	}

	public boolean search(int key) {
		int index = key%bucket;
		return mytable.get(index).contains(key);
	}

	public String toString() {
		return mytable.toString();
	}

}

public class Lab2 {
	public static void main(String[] args) {

		MyHashSet  myset = new MyHashSet(7);

		myset.insert(50);
		myset.insert(21);
		myset.insert(58);
		myset.insert(17);
		myset.insert(15);
		myset.insert(49);
		myset.insert(56);
		myset.insert(22);
		myset.insert(23);
		myset.insert(25);
		myset.insert(23);
		myset.insert(25);

		System.out.println(myset);
```

```
                myset.delete(23);
                myset.delete(25);
                myset.delete(50);

                System.out.println(myset);

                System.out.println(myset.search(50));
                System.out.println(myset.search(49));
        }
}
```

Lab3.java

```
package com.jlcindia.hashing3;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

//Implement HashSet using Chaining
// To Store Integers
// Updated

class MyHashSet {

        private int bucketSize;
        private int currentSize;
        List<LinkedList<Integer>> mytable;

        public MyHashSet(int bucketSize) {
                this.bucketSize = bucketSize;
                this.mytable = new ArrayList<LinkedList<Integer>>(bucketSize);
```

```
        for(int i=0;i<bucketSize;i++) {
            mytable.add(new LinkedList<Integer>());
        }
    }

    public int size() {
        return this.currentSize;
    }

    public boolean isEmpty() {
        return currentSize==0;
    }

    public void clear() {

        this.mytable.clear();

        for(int i=0;i<bucketSize;i++) {
            mytable.add(new LinkedList<Integer>());
        }
        this.currentSize=0;
    }

    public void insert(Integer key) {
        int index = myhash(key);
        if( !search(key)) {
            mytable.get(index).add(key);
            currentSize++;
        }
    }

    public void delete(Integer key) {
        int index = myhash(key);
        mytable.get(index).remove((Integer)key);
        currentSize--;
    }
```

```
	public boolean search(Integer key) {
		int index = myhash(key);
		return mytable.get(index).contains(key);
	}

	public int myhash(Integer key) {
		int hash = key.hashCode() % this.bucketSize;
		return hash;
	}

	public String toString() {
		return mytable.toString();
	}

}

public class Lab3 {
	public static void main(String[] args) {

		MyHashSet  myset = new MyHashSet(7);

		System.out.println("------1-------");
		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);

		myset.insert(50);
		myset.insert(21);
		myset.insert(58);
		myset.insert(17);
		myset.insert(15);
		myset.insert(49);
		myset.insert(56);
		myset.insert(22);
		myset.insert(23);
		myset.insert(25);
		myset.insert(23);
```

```
                myset.insert(25);

                System.out.println("------2-------");

                System.out.println(myset.size());
                System.out.println(myset.isEmpty());
                System.out.println(myset);

                myset.delete(23);
                myset.delete(25);
                myset.delete(50);

                System.out.println("------3-------");

                System.out.println(myset.size());
                System.out.println(myset.isEmpty());
                System.out.println(myset);

                System.out.println("------4-------");
                System.out.println(myset.search(50));
                System.out.println(myset.search(49));

                System.out.println("------5-------");
                myset.clear();
                System.out.println(myset.size());
                System.out.println(myset.isEmpty());
                System.out.println(myset);
        }

}
```

Lab4.java

```
package com.jlcindia.hashing4;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

//Implement HashSet using Chaining
//To Store Strings

class MyHashSet {

	private int bucketSize;
	private int currentSize;
	List<LinkedList<String>> mytable;

	public MyHashSet(int bucketSize) {
		this.bucketSize = bucketSize;
		this.mytable = new ArrayList<LinkedList<String>>(bucketSize);

		for(int i=0;i<bucketSize;i++) {
			mytable.add(new LinkedList<String>());
		}
	}

	public int size() {
		return this.currentSize;
	}

	public boolean isEmpty() {
		return currentSize==0;
	}
```

```
public void clear() {

        this.mytable.clear();

        for(int i=0;i<bucketSize;i++) {
                mytable.add(new LinkedList<String>());
        }
        this.currentSize=0;
}

public void insert(String key) {
        int index = myhash(key);
        if( !search(key)) {
                mytable.get(index).add(key);
                currentSize++;
        }
}

public void delete(String key) {
        int index = myhash(key);
        mytable.get(index).remove(key);
        currentSize--;
}

public boolean search(String key) {
        int index = myhash(key);
        return mytable.get(index).contains(key);
}

public int myhash(String key) {
        int hash = key.hashCode() % this.bucketSize;
        return hash;
}
```

```
	public String toString() {
		return mytable.toString();
	}

}

public class Lab4 {
	public static void main(String[] args) {

		MyHashSet  myset = new MyHashSet(7);

		System.out.println("------1-------");
		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);

		myset.insert("aa");
		myset.insert("bb");
		myset.insert("cc");
		myset.insert("dd");
		myset.insert("ee");

		myset.insert("ab");
		myset.insert("bc");
		myset.insert("cd");
		myset.insert("de");
		myset.insert("ef");


		System.out.println("------2-------");

		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);
```

```
            myset.delete("ab");
            myset.delete("de");
            myset.delete("dd");

            System.out.println("------3-------");

            System.out.println(myset.size());
            System.out.println(myset.isEmpty());
            System.out.println(myset);

            System.out.println("------4-------");
            System.out.println(myset.search("aa"));
            System.out.println(myset.search("dd"));

            System.out.println("------5-------");
            myset.clear();
            System.out.println(myset.size());
            System.out.println(myset.isEmpty());
            System.out.println(myset);
        }

}
```

8.6. Open Addressing

- In open addressing, all keys are stored in the hash table itself.
- This approach is also known as closed hashing.
- This procedure is based on probing.
- Collision is resolved by probing.

- Open Addressing can be implemented in 3 ways
    1) Linear probing
    2) Quadratic probing
    3) Double hashing ***

8.7. Linear Probing

- Interval between probes is fixed at 1.
- In linear probing, we search the hash table sequentially, starting from the original hash location.
- If a location is occupied, we check the next location.
- We wrap around from the last table location to the first table location if necessary.

Ex:

keys ={50, 51, 49, 16, 56, 15, 19} **hash(key) = key % 7**

- Problem with linear probing is that table items forms cluster together in the hash table.
- This means that the table contains groups of consecutively occupied locations that are called clustering.
- Clustering causes long probe searches and therefore decreases the overall efficiency.

8.8. Quadratic Probing

- Interval between probes increases proportionally to the hash value
- Problem of Clustering can be reduced if we use the quadratic probing method.
- In quadratic probing, we start from the original hash location.
- If a location is occupied, we check the locations i + 1^2 , i +2^2, i + 3^2, i + 4^2...

Ex:

keys ={31, 19, 2, 13, 25, 24, 21, 9} **hash(key) = key % 11**

- Even though clustering is avoided by quadratic probing, still there are chances of thin clustering.

8.9. Double Hashing

- Interval between probes is computed by another hash function.
- Double hashing reduces clustering in a better way.
- Increments for the probing sequence are computed by using a second hash function.
- In Double hashing, We first probe the location h1(key).
- If the location is occupied, we probe the location (h1(key) + i * h2(key)) % m

For Example, as follows

(h1(key) + i * h2(key)) % m

(h1(key) + 1* h2(key)) % m
(h1(key) + 2 * h2(key)) % m
(h1(key) + 3 * h2(key)) % m
etc

Example-1

keys = {49, 63, 56, 52, 54, 48}

m = 7 (0 to 6)
h1(key) = key % 7
h2(key) = 6- (key % 6)

Lab5.java

```
package com.jlcindia.hashing5;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
//Implement HashSet using Double Hashing

class MyHashSet {
	private int bucketSize;
	private int currentSize;
	private Integer[] mytable;

	public MyHashSet(int bucketSize) {
		this.bucketSize = bucketSize;
		this.mytable = new Integer[bucketSize];
		this.currentSize = 0;
	}

	public int size() {
		return this.currentSize;
	}

	public boolean isEmpty() {
		return currentSize == 0;
	}

	public void clear() {

		this.currentSize = 0;
		for (int i = 0; i < bucketSize; i++) {
			mytable[i] = null;
		}

	}
```

```
public void add(Integer key) {
        int index = myhash1(key);
        if (mytable[index] != null) {
                index = myhash(key,"add");
        }

        mytable[index] = key;
        this.currentSize++;

}

public void remove(Integer key) {

        int index = myhash1(key);
        if (mytable[index] != key) {
                index = myhash(key,"remove");
        }

        if (mytable[index] == key) {
                mytable[index] = null;
                this.currentSize--;
        }

}

public boolean contains(Integer key) {
        int index = myhash1(key);
        if (mytable[index] != key) {
                index = myhash(key,"search");
        }

        return mytable[index] == key;
}

public int myhash1(Integer key) {
        int hash1 = key.hashCode() % this.bucketSize;
        return hash1;
}
```

```
	public int myhash2(Integer key) {
	int hash2 = (this.bucketSize - 1) - key.hashCode() % (this.bucketSize - 1);
		return hash2;
	}

	public int myhash(Integer key, String ops) {
		int hash1 = myhash1(key);
		int hash2 = myhash2(key);

		int myhash = 0;
		for (int i = 1; i <= this.bucketSize; i++) {
			myhash = (hash1 + i * hash2) % this.bucketSize;
			if (ops.equals("add")) {
				if (mytable[myhash] == null) {
					return myhash;
				}
			} else {
				if (mytable[myhash] == key) {
					return myhash;
				}
			}
		}
		return myhash;
	}

	public String toString() {
		String str = "[";
		for (int i = 0; i < this.bucketSize; i++) {
			if (mytable[i] != null) {
				str += mytable[i] + "\t";
			}
		}
		str = str + "]";
		return str;
	}

}
```

```
public class Lab5 {
	public static void main(String[] args) {
		MyHashSet myset = new MyHashSet(7);

		System.out.println("------1-------");
		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);

		myset.add(49);		myset.add(63);
		myset.add(56);		myset.add(52);
		myset.add(54);		myset.add(48);

		System.out.println("------2-------");
		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);

		myset.remove(49);	myset.remove(48);
		myset.remove(99);

		System.out.println("------3-------");
		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);

		System.out.println("------4-------");
		System.out.println(myset.contains(49));
		System.out.println(myset.contains(52));

		myset.clear();
		System.out.println("------5-------");
		System.out.println(myset.size());
		System.out.println(myset.isEmpty());
		System.out.println(myset);
	}
}
```

Lab6.java

```
package com.jlcindia.hashing6;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */

//Implement HashMap using Chaining To Store Integers
// 706. Design HashMap

class MyEntry<K, V> {

	public Integer key;
	public Integer value;

	public MyEntry(Integer key, Integer value) {
		this.key = key;
		this.value = value;
	}

	public String toString() {
		return "{" + key + ":" + value + "}";
	}
}

class MyHashMap {

	private int bucketSize;
	private int currentSize;
	List<LinkedList<MyEntry<Integer, Integer>>> mytable;
```

```
public MyHashMap(int bucketSize) {

this.bucketSize = bucketSize;
this.mytable =
new ArrayList<LinkedList<MyEntry<Integer, Integer>>>(bucketSize);

for (int i = 0; i < bucketSize; i++) {
mytable.add(new LinkedList<MyEntry<Integer, Integer>>());
}
}
public int size() {
        return this.currentSize;
}
public boolean isEmpty() {
        return currentSize == 0;
}
public void clear() {
        this.mytable.clear();

        for (int i = 0; i < bucketSize; i++) {
                mytable.add(new LinkedList<MyEntry<Integer, Integer>>());
        }
        this.currentSize = 0;
}

public Integer get(Integer mykey) { // 50
int index = myhash(mykey); // 1
LinkedList<MyEntry<Integer, Integer>> mylinkedlist = mytable.get(index);

for (MyEntry<Integer, Integer> myentry : mylinkedlist) {
if (myentry.key.equals(mykey)) {
return myentry.value;
}
}
return -1;
}
```

```
public void put(Integer mykey, Integer myvalue) {
int index = myhash(mykey);

LinkedList<MyEntry<Integer, Integer>> mylinkedlist = mytable.get(index);

boolean found=false;
for (MyEntry<Integer, Integer> myentry : mylinkedlist) {
if (myentry.key.equals(mykey)) {
myentry.value=myvalue;
found=true;
}
}

if(!found) {
mylinkedlist.add(new MyEntry<Integer,Integer>(mykey,myvalue));
currentSize++;
}

}

public void remove(Integer mykey) {
int index = myhash(mykey);

LinkedList<MyEntry<Integer, Integer>> mylinkedlist = mytable.get(index);

for (MyEntry<Integer, Integer> myentry : mylinkedlist) {
if (myentry.key.equals(mykey)) {
mylinkedlist.remove(myentry);
currentSize--;
break;
}
}

}
```

```
	public int myhash(Integer key) {
		int hash = key.hashCode() % this.bucketSize;
		return hash;
	}

	public String toString() {
		return mytable.toString();
	}

}

public class Lab6 {
	public static void main(String[] args) {

		MyHashMap mymap = new MyHashMap(7);

		System.out.println("------1-------");
		System.out.println(mymap.size());
		System.out.println(mymap.isEmpty());
		System.out.println(mymap);

		mymap.put(50, 5050);
		mymap.put(21, 2121);
		mymap.put(58, 5858);
		mymap.put(17, 1717);
		mymap.put(15, 1515);
		mymap.put(49, 4949);
		mymap.put(56, 5656);
		mymap.put(22, 2222);
		mymap.put(23, 2323);
		mymap.put(25, 2525);

		mymap.put(23, 8888);
		mymap.put(25, 9999);
```

```
                System.out.println("------2-------");

                System.out.println(mymap.size());
                System.out.println(mymap.isEmpty());
                System.out.println(mymap);

                mymap.remove(23);
                mymap.remove(25);
                mymap.remove(50);

                System.out.println("------3-------");

                System.out.println(mymap.size());
                System.out.println(mymap.isEmpty());
                System.out.println(mymap);

                System.out.println("------4-------");
                System.out.println(mymap.get(50));
                System.out.println(mymap.get(49));

                System.out.println("------5-------");
                mymap.clear();
                System.out.println(mymap.size());
                System.out.println(mymap.isEmpty());
                System.out.println(mymap);
        }

}
```