Hash

a. **Hash functions**:

Suppose we need to have M number of key-value pairs

1) Keys are positive integers

   Modular hashing

   hashIndex = key % M;

2) Floating point numbers

   Turn the floating number to binary representation and use it as the key, then do modular hashing as for positive integers

3) Turn string to positive integer code, for example:

```
String s;
int hashIndex = 0;
for(int i=0; i< s.length; i++)
{
   hashIndex = (R*hashIndex + s.charAt(i)) % M
}
```

   (java uses R = 31)

4) Java conventions

   Every data type must implement a method called hashCode,

   The JVM provides a default one (this can be understood as using address)

   Or users are able to override it.

   Convert hashcode to array index,

```
Private int hashFunction(Key key)
{
   Return (key.hashCode() & 0x7fffffff ) %M
}
```

   This mask turns 32 bit Integer to 31 bit non negative integer

b. Collision resolving using chaining

1) Idea

For each cell in the hashtable, use a list to store <key, value> pair that has the same hash index

Hash Index	list<key, value>
0	
1	<15, v1> <8, v2>
2	
3	
4	<11, v3>
5	
6	<27, v4>

2) Example in c++

```
class HashNode {
private:
    int key;
    int value;
    HashNode *next;

public:
   // constructor
   //HashNodeint key, int value)
   // implement getters and setters
   //  getKey()  getValue() setValue()  getNext()setNext()
};

const int TABLE_SIZE = 128;
class HashMap
{
private:
HashNode **table;
Public:
    HashMap() {
        table = new HashNode*[TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i++)
            table[i] = NULL;
    }


    int get(int key) {
        int hash = (key % TABLE_SIZE);
        if (table[hash] == NULL)
```

```
                return -1;
            else {
                HashNode *entry = table[hash];
                while (entry != NULL && entry->getKey() != key)
                    entry = entry->getNext();
                return entry == NULL? -1 : entry->getValue();
            }
        }
        void put(int key, int value) {
            int hash = (key % TABLE_SIZE);
            if (table[hash] == NULL)
                table[hash] = new HashNode(key, value);
            else {
                HashNode *entry = table[hash];
                while (entry->getNext() != NULL)
                    entry = entry->getNext();
                if (entry->getKey() == key)
                    entry->setValue(value);
                else
                    entry->setNext(new HashNode(key, value));
            }
        }
        void remove(int key) {
            int hash = (key % TABLE_SIZE);
            if (table[hash] != NULL) {
                HashNode *prevEntry = NULL;
                HashNode *entry = table[hash];
                while (entry->getNext() != NULL && entry->getKey() != key) {
                    prevEntry = entry;
                    entry = entry->getNext();
                }
                if (entry->getKey() == key) {
                    HashNode *nextEntry = entry->getNext();
                    if (prevEntry == NULL) {
                       table[hash] = nextEntry;
                    } else {
                    prevEntry->setNext(next);
                    }
                    delete entry;
                }
            }
        }
    };
```

c. Collision resolving using open addressing

1) Idea
   Open addressing solves the hash collision as follows,
   If collision, find the next empty cell
   If full, increase the size of the table

2) Example
   Here, to mark a node deleted we have used **DeletedNode** with key and value -1. This can differenciate if table cell is empty by default or it is empty due to deletion.
   Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

```
class DeletedEntry: public HashEntry {
private:
    static DeletedEntry *entry;
    DeletedEntry() :
        HashEntry(-1, -1) {
    }
public:
    static DeletedEntry *getUniqueDeletedEntry() {
        if (entry == NULL)
            entry = new DeletedEntry();
        return entry;
    }
};

DeletedEntry *DeletedEntry::entry = NULL;
const int TABLE_SIZE = 128;

class HashMap {
private:
    HashEntry **table;
public:
    HashMap() {
        table = new HashEntry*[TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i++)
            table[i] = NULL;
    }




    int get(int key) {
```

```
    int hash = (key % TABLE_SIZE);
    int initialHash = -1;
    while (hash != initialHash && (table[hash]
            == DeletedEntry::getUniqueDeletedEntry() ||
            table[hash] != NULL && table[hash]->getKey() != key)) {
        if (initialHash == -1)
            initialHash = hash;
        hash = (hash + 1) % TABLE_SIZE;
    }
    if (table[hash] == NULL || hash == initialHash)
        return -1;
    else
        return table[hash]->getValue();
}


void put(int key, int value) {
    int hash = (key % TABLE_SIZE);
    int initialHash = -1;
    int indexOfDeletedEntry = -1;
    while (hash != initialHash && (table[hash]
            == DeletedEntry::getUniqueDeletedEntry() ||
           table[hash] != NULL && table[hash]->getKey() != key)) {
        if (initialHash == -1)
            initialHash = hash;
        if (table[hash] == DeletedEntry::getUniqueDeletedEntry())
            indexOfDeletedEntry = hash;
        hash = (hash + 1) % TABLE_SIZE;
    }
    if ((table[hash] == NULL || hash == initialHash)
         && indexOfDeletedEntry!= -1)
        table[indexOfDeletedEntry] = new HashEntry(key, value);
    else if (initialHash != hash)
        if (table[hash] != DeletedEntry::getUniqueDeletedEntry()
               && table[hash] != NULL && table[hash]->getKey() == key)
            table[hash]->setValue(value);
        else
            table[hash] = new HashEntry(key, value);
}
```

```
    void remove(int key) {
        int hash = (key % TABLE_SIZE);
        int initialHash = -1;
        while (hash != initialHash && (table[hash]
                == DeletedEntry::getUniqueDeletedEntry() ||
                 table[hash] != NULL && table[hash]->getKey() != key)) {
            if (initialHash == -1)
                initialHash = hash;
            hash = (hash + 1) % TABLE_SIZE;
        }
        if (hash != initialHash && table[hash] != NULL) {
            delete table[hash];
            table[hash] = DeletedEntry::getUniqueDeletedEntry();
        }
    }

};
```

Open addressing vs. chaining

	Chaining	**Open addressing**
Collision resolution	Using external data structure	Using hash table itself
Memory waste	Pointer size overhead per entry (storing list heads in the table)	No overhead [1]
Performance dependence on table's load factor	Directly proportional	Proportional to `(loadFactor) / (1 - loadFactor)`
Allow to store more items, than hash table size	Yes	No. Moreover, it's recommended to keep table's load factor below 0.7
Hash function requirements	Uniform distribution	Uniform distribution, should avoid clustering
Handle removals	Removals are ok	Removals clog the hash table with "DELETED" entries
Implementation	Simple	Correct implementation of open addressing based hash table is quite tricky