1. Implement the class **DoubleHashingHashMap** that provides a hash map with a double hashing collision handling scheme.

   The **HashMap** has initial size 17 and it has to maintain the load factor of the table under 0.5.

   The class must implement also a method **getCollisions()** that returns the number of collisions found during the lifetime of a **DoubleHashingHashMap** object: i.e., suppose that a **DoubleHashingHashMap** object is created, and three insert operations are executed on this object, with the second colliding with the first and the third one colliding with both the first and the second, then the method **getCollisions()** on this object must return 3. N.B.: you are allowed to use auxiliary attributes for implementing this method.

   The keys of entries inserted in the **HashMap** will be three capital letter codes for currencies according to the standard ISO 4217 (see, e.g., https://en.wikipedia.org/wiki/ISO_4217). So you must implement opportune hash functions for handling with these keys. N.B.: hash functions must be designed in order to minimize the number of collisions.

   Your choice of hash functions will be tested against a prefixed sequence of 100 insert and remove operations (70% of operations will be insert operations and the remaining ones will be remove operations).

   The five groups whose hash functions generate the minimum number of collisions during the above test will receive a bonus point.

2. Implement the class **Currency**, whose objects have the following attributes:
   - **Code**: The code three capital letters identifying a currency according to the standard ISO 4217;
   - **Denominations**: A map defining the denominations allowed for the currency represented by the current object (e.g., for currency EUR, the map contains 0,05; 0,1; 0,2; 0,5; 1; 2; 5; 10; 20; 50; 100; 200; 500);
   - **Changes**: A **DoubleHashingHashMap** whose keys are currency codes different from the code of the given currency, and whose values are float numbers indicating the rate exchange between the currency represented by the current object and the currency given as key.

   The constructor of the class takes as input a code **c** and initializes **Code** to this value, **Denominations** as an empty map, and **Change** as an empty hash map.
   The class moreover has the following methods:
   - **AddDenomination(value)**: add **value** in the **Denominations** map. It raises an exception if **value** is already present;
   - **DelDenomination(value)**: remove **value** from the **Denominations** map. It raises an exception if **value** is not present;
   - **MinDenomination([value])**: the parameter **value** is optional. If it is not given, it returns the minimum denomination (it raises an exception if no denomination exists), otherwise it returns the minimum denomination larger than **value** (it raises an exception if no denomination exists larger than **value**);
   - **MaxDenomination([value])**: the parameter **value** is optional. If it is not given, it returns the maximum denomination (it raises an exception if no denomination exists), otherwise it returns the maximum denomination smaller than **value** (it raises an exception if no denomination exists larger than **value**);
   - **nextDenomination(value)**: return the denomination that follows **value**, if it exists, **None** otherwise. If **value** is not a denomination it raises an exception;

- **prevDenomination(value)**: return the denomination that precedes **value**, if it exists, **None** otherwise. If **value** is not a denomination it raises an exception;
- **hasDenominations()**: return true if the **Denominations** map is not empty;
- **numDenominations()**: returns the number of elements in the **Denominations** map;
- **clearDenominations()**: remove all elements from the **Denominations** map;
- **iterDenominations(reverse=false)**: returns an iterator over the **Denominations** map. If **reverse** is false (default value), the iterator must iterate from the smaller to the larger denomination, otherwise it must iterate from the larger to the smaller denomination;
- **addChange(currencycode, change)**: add an entry in the **Changes** hash map, whose key is **currencycode** and whose value is **change**. It raises an exception if the key **currencycode** is already present;
- **removeChange(currencycode)**: remove the entry with key **currencycode** from the **Changes** hash map. It raises an exception if the key **currencycode** is not present;
- **updateChange(currencycode, change)**: updates the value associated with key **currencycode** to **change**. If the key **currencycode** does not exist, it will be inserted in the **Changes** hash map;
- **copy()**: create a new object **Currency** whose attributes are equivalent to the ones of the current currency;
- **deepcopy()**: create a new object **Currency** whose attributes are equivalent but not identical to the ones of the current currency.

For each operation, you must evaluate its computational complexity. In implementing the **Currency** class, you must choice an implementation of the **Denominations** map that minimizes the complexity of these operations.

N.B. You are allowed to define, if needed, other attributes or methods for supporting the implementation of above methods.

3. Implement a (2,8)-Tree that memorizes **Currency** objects, by using their **Code** as key.

4. Implement a function **change(value, currency)**, where **value** is a float number with at most two decimal points and **currency** is an object of the class **Currency**.

   The function must use a **PriorityQueue** to return the minimum number of coins of the given currency that sums up to the given value. E.g.: on input 12,85 and EUR currency, the function must return 6 corresponding to 10+2+0,50+0,20+0,10+0,5. Note that the function is required to return both the number and the list of the corresponding coins.