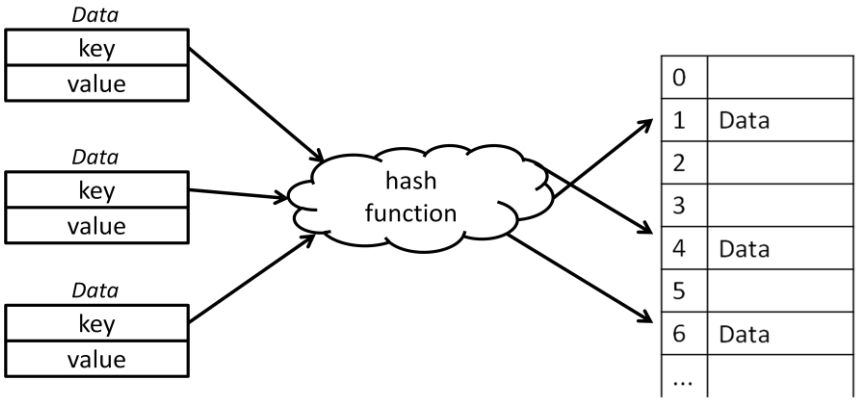
Praktikum DAA: Hashing #1

Pada modul ini kita akan mengimplementasikan struktur data Hash Table. Pada struktur data ini, *Data* yang berupa pasangan *key* dan *value* disimpan pada tabel dengan posisi tertentu. Posisi ditentukan oleh suatu fungsi yang dikenal dengan istilah *hash function*. Jika posisi yang ingin ditempati sudah ada isinya, maka perlu suatu penanganan yang dikenal dengan teknik *collision handling* (dibuat di modul berikutnya). Pada modul ini, jika posisi yang ingin ditempati sudah ada isinya, maka proses insert dianggap gagal.



1. Hash Table

Mula-mula buatlah sebuah class `HashTable`. Tipe data untuk *key* dan *value* kita buat sebagai tipe data generic. Perhatikan contoh untuk cara deklarasi lebih dari satu buah tipe data generic. Buatlah juga inner class `Data` yang memiliki dua atribut, `key` dan `value`. Kelas ini hanya dipakai untuk `HashTable`, karena itu dibuat private. Selain itu, untuk menangani berbagai jenis fungsi hash, kita buat kelas `HashTable` sebagai abstract class yang memiliki method abstract `hashFunction()`. Nantinya kelas ini bisa diturunkan menjadi berbagai jenis hash table dengan fungsi hash berbeda-beda.

```
abstract class HashTable<K,V>{
    protected Data[] table;
    protected int capacity;

    private class Data{
        K key;
        V value;

        Data(K key, V value){
            this.key = key;
            this.value = value;
        }
    }

    public HashTable(int capacity){
        this.capacity = capacity;
        this.table = (Data[]) new HashTable.Data[capacity];
    }

    abstract protected int hashFunction(K key);
}
```

Perhatikan potongan kode berikut ini:

```
this.table = (Data[]) new HashTable.Data[capacity];
```

Di sini kita ingin membuat array yang tipenya `Data[]`, namun mengapa penulisannya seperti ini ? Ini karena kelas `Data` merupakan inner class dari kelas `HashTable<K,V>` yang mengandung tipe data generic `K dan V`, sehingga sebenarnya ia adalah kelas `HashTable<K,V>.Data`. Pada Java kita tidak diperbolehkan membuat array untuk tipe data generic karena masalah keamanan. Karena `K dan V` tidak diketahui apa tipenya pada saat di-*compile*, ada kemungkinan saat *run time* terjadi ketidakcocokan tipe data, misalnya seperti berikut:

```
HashTable<String,String>.Data x = new HashTable<Integer,Integer>.Data();
```

Ingat bahwa kita boleh menyimpan suatu objek pada reference yang tipenya lebih umum, misalnya ada dua kelas sebagai berikut:

```
class TipeA{…}
class TipeB extends TipeA {…}
```

maka kode berikut ini adalah legal:

```
TipeA myObj = new TipeB();
```

Dengan konsep yang sama, kita boleh menyimpan objek bertipe `HashTable<String,String>.Data` pada reference yang tipenya lebih umum, yaitu `HashTable.Data`. Jadi kode berikut adalah legal:

```
HashTable.Data x = new HashTable<String,String>.Data();
HashTable.Data y = new HashTable<Integer,Integer>.Data();
```

Jadi untuk membuat array `table` kita menginstansiasi dengan tipe data yang lebih umum, yaitu `HashTable.Data[…]`. Namun demikian supaya tipe datanya cocok dengan atribut `table`, maka perlu diberi type casting `(Data[])`, yang sebenarnya adalah kependekan dari `(HashTable<K,V>.Data[])`. Cara ini bisa dilakukan, tapi akan menyebabkan warning saat compile. Pada kasus ini, kita dapat mengabaikan warning tersebut selama kita yakin bahwa kode kita tidak pernah memasukkan objek dengan tipe data yang salah pada array `table`.

2. Operasi insert, delete, search

Berikutnya kita akan mengimplementasikan operasi insert, delete, dan search. Ketiganya menggunakan method `hashFunction()` untuk menemukan nomor index pada table, tapi tidak terpengaruh pada apa teknik hash function yang digunakan, karena itu ketiganya dapat diimplementasikan pada kelas `HashTable`. Perhatikan contoh berikut untuk method `search()`. Pada method ini, data dicari pada lokasi sesuai hasil perhitungan fungsi hash terhadap key. Jika lokasi tersebut kosong, berarti data tidak ditemukan. Jika tidak kosong, tetap perlu diperiksa lagi apakah key-nya sama, karena ada kemungkinan beda key tapi dipetakan ke lokasi yang sama. Perhatikan bahwa `key` merupakan objek yang bertipe `K`, maka untuk membandingkan nilainya, kita gunakan method `.equals()`, bukan menggunakan operator ==. Sebagai latihan, lengkapi method `insert()` dan `delete()`!

```
public V search(K key){
    int idx = this.hashFunction(key);
    if(this.table[idx] != null && this.table[idx].key.equals(key))
        return this.table[idx].value;
    else
        return null;
}

public boolean insert(K key, V value){
    // lengkapi...
}

public V delete(K key){
    // lengkapi...
}
```

3. Hash Function

Ada banyak cara untuk mengimplementasikan hash function. Beberapa teknik yang umum digunakan adalah: Modular/Division, Truncation, Multiplicative, Folding/Shifting, dan Length-dependent. Sebagai contoh, kita akan membuat hash function dengan cara modular/division. Untuk itu kita akan menurunkan kelas `HashTable` menjadi sub-class yang mengimplementasikan method `hashFunction()`. Selain fungsi hash, perlu ditentukan juga apa tipe key-nya, karena cara pengolahan terhadap tipe data yang berbeda bisa berbeda juga. Pada contoh ini kita menggunakan key bertipe Integer.

```
class ModularHashInteger<V> extends HashTable<Integer,V>{
    public ModularHashInteger(int capacity){
        super(capacity);
    }

    protected int hashFunction(Integer key){
        return key%this.capacity;
    }
}
```

Perhatikan bahwa ketika kita menurunkan dari kelas `HashTable<K,V>`, tipe data `K` dan `V` harus didefinisikan secara jelas. `K` dan `V` boleh diisi dengan tipe data yang sudah ada, atau lewat generic lagi. Sebagai contoh, kode ini legal. Pada contoh ini, `K` dan `V` diisi String milik library Java:

```
class Foo extends HashTable<String, String>{…}
```

Cara kedua adalah dengan diisi generic lagi, misal pada contoh kelas `ModularHashInteger<V>`, `V` di sini merupakan tipe data generic yang kemudian dipakai untuk mendefinisikan `V` pada `HashTable<Integer,V>`, sedangkan K diisi dengan `Integer` milik library Java.

Ujilah kelas anda, misalnya dengan Tester sebagai berikut:

```
class TesterHash{
    public static void main(String[] args){
        ModularHashInteger<String> h = new ModularHashInteger<String>(11);
        h.insert(5, "John Smith");
        h.insert(16, "Jane Smith");
        System.out.println(h.search(5));
        System.out.println(h.search(7));
    }
}
```

Hasil yang diharapkan adalah insert pertama berhasil, insert kedua gagal (karena 16 mod 11 adalah 5, dan lokasi tersebut sudah terisi). Maka outputnya adalah:

```
John Smith
null
```

Sebagai latihan, implementasikanlah kelas `MultiplicativeHashInteger<V>` dan `FoldingHashInteger<V>`! Ujilah kelas yang anda buat. Kedua kelas ini mirip seperti contoh sebelumnya, hanya saja fungsi hash-nya menggunakan cara multiplicative dan folding.

4. Konversi String Menjadi Integer

Beberapa fungsi hash hanya cocok digunakan untuk mengolah bilangan bulat, seperti misalnya cara modular. Tapi key tidak selalu bertipe bilangan bulat, karena itu perlu dikonversi terlebih dahulu. Proses konversi ini bisa langsung dikerjakan di dalam method `hashFunction()`. Lengkapi method `hashFunction()` untuk kelas berikut ini, sesuai dengan cara yang dibahas pada slide (`String` dianggap sebagai bilangan basis 256).

```
class ModularHashString<V> extends HashTable<String, V>{
    public ModularHashString(int capacity){
        super(capacity);
    }

    protected int hashFunction(String key){
        //lengkapi...
    }
}
```