Praktikum DAA: Hashing #2

Pada modul ini kita akan menambahkan mekanisme collision handling pada kelas yang sudah dibuat pada modul sebelumnya. Ada tiga cara collision handling, yaitu Open Addressing, Separate Chaining, dan Coalesced Hashing. Hash table yang dibuat pada modul sebelumnya sudah menggunakan array of Data, karena itu collision handling yang cocok dipakai adalah cara Open Addressing. Teknik ini sendiri terbagi lagi menjadi tiga macam berdasarkan fungsi yang dipakai untuk menentukan posisi pengganti ketika terjadi tabrakan. Pada modul ini kita akan mengimplementasikan cara Quadratic Probing.

1. Quadratic Probing

Fungsi untuk quadratic probing adalah: $h(k,i) = (h`(k) + c_1 i + c_2 i^2) \bmod m$
Di mana *h'()* adalah fungsi hash yang dipakai.

Untuk mengimplementasikan cara ini, kita perlu menambahkan atribut `c1` dan `c2` pada kelas `HashTable` dan memodifikasi constructor kelas ini dan kelas-kelas turunannya.

```
abstract class HashTable<K,V>{
    protected Data[] table;
    protected int capacity;
    protected double c1, c2;

    ...

    public HashTable(int capacity, double c1, double c2){
        this.capacity = capacity;
        this.table = (Data[]) new HashTable.Data[capacity];
        this.c1 = c1;
        this.c2 = c2;
    }

    ...
}

class ModularHashInteger<V> extends HashTable<Integer,V>{
    public ModularHashInteger(int capacity, double c1, double c2){
        super(capacity, c1, c2);
    }

    ...
}
```

Selanjutnya kita tambahkan sebuah method yang mengimplementasikan *h(k,i)*. Perhatikan bahwa *h'(k)* selalu dihitung pada setiap pemanggilan *h(k,i)*, dan nilainya tidak pernah berubah. Karena itu sebenarnya nilai ini tidak perlu kita hitung berulang-ulang. Untuk implementasi, kita tambahkan method `quadraticProbing()` yang menerima parameter berupa nilai *h'(k)* yang sudah dihitung, sebutlah h_0. Karena nilai c1 dan c2 mungkin tidak bulat, maka kita tambahkan type casting (int) untuk pembulatan. Alternatifnya, bisa juga dibulatkan dengan `Math.round()` atau yang sejenisnya.

```
protected int quadraticProbing(int k0, int i){
    return ((int)(k0 + this.c1*i + this.c2*i*i))%this.capacity;
}
```

2. Operasi Insert

Selanjutnya kita modifikasikan method `insert()`. Untuk memasukkan sebuah `Data`, kita cari posisinya menggunakan fungsi *h(k,i)* dengan *i=0,1,2,…,M-1*. Jika lokasi kosong sudah ditemukan, `Data` boleh dimasukkan, jika tidak, nilai *i* ditambah 1. Demikian seterusnya hingga *M* kali percobaan. Setelah *M* kali gagal menemukan lokasi kosong, maka `Data` gagal dimasukkan.

```
public boolean insert(K key, V value){
    Data newData = new Data(key, value);
    int k0 = this.hashFunction(key);
    int idx;
    for(int i=0; i<this.capacity; i++){
        idx = this.quadraticProbing(k0, i);
        if(this.table[idx] == null){
            this.table[idx] = newData;
            return true;
        }
    }
    return false;
}
```

3. Operasi Search

Operasi search mirip dengan insert. Pencarian dilakukan pada lokasi-lokasi yang ditunjuk oleh fungsi *h(k,i) dengan i=0,1,2,…,M-1*. Jika `key` ditemukan, `value`-nya dikembalikan. Jika ditemukan lokasi yang berisi `null`, artinya pencarian boleh dihentikan. Jika *M* kali looping selalu menemukan `Data` yang `key`-nya tidak sesuai, artinya tidak ada `Data` dengan `key` tersebut. Sebagai latihan, modifikasilah method `search()` anda sesuai aturan yang telah dijelaskan.

4. Operasi Delete

Untuk menghapus data, perlu dilakukan pencarian terlebih dahulu, karena itu prosesnya sangat mirip dengan method `search()`. Perhatikan kode berikut ini:

```
public V delete(K key){
    int k0 = this.hashFunction(key);
    int idx;
    for(int i=0; i<this.capacity; i++){
        idx = this.quadraticProbing(k0, i);

        if(this.table[idx] == null)      //mencapai lokasi yang isinya null
            return null;                 //artinya key tidak ditemukan

        else if(this.table[idx].key.equals(key)){    //key ditemukan
            V result = this.table[idx].value;
            this.table[idx] = null;
            return result;
        }
    }

    return null;              //sudah M kali mengulang dan key tidak ditemukan
}
```

Kode di atas masih salah. Kita tidak boleh sembarangan mengganti isi lokasi yang dihapus dengan `null`, karena akan mempengaruhi proses pencarian. Ingat bahwa operasi `search()` akan dihentikan ketika ditemukan nilai `null`. Karena itu kita akan membuat sebuah object `Data` *dummy* sebagai penanda (tombstone).

Tambahkan atribut berikut ini:

```
protected Data tombstone = new Data(null, null);
```

Selanjutnya objek ini akan dipakai untuk mengisi lokasi-lokasi yang isinya dihapus. Jadi kode

```
this.table[idx] = null;
```

diganti menjadi

```
this.table[idx] = this.tombstone;
```

Dengan demikian, isi table ada tiga jenis: *null*, objek `Data`, atau `tombstone`. Tentunya objek `tombstone` ini hanya penanda, jadi bukan `Data` yang sebenarnya. Pada contoh ini, kita buat `tombstone` berisi `key=null` dan `value=null`. Jika Hash Table dipakai untuk kasus di mana tidak ada key bernilai `null`, maka objek `tombstone` tidak pernah akan dianggap sebagai `Data` yang ditemukan. Namun akan menjadi masalah jika kasusnya memiliki key bernilai `null`. Karena itu perlu kita beri penganganan khusus, objek `Data` yang diperiksa `key`-nya hanyalah yang bukan `tombstone`.

```
else if(this.table[idx] != this.tombstone &&     //bukan tombstone
        this.table[idx].key.equals(key)){        //key ditemukan
    V result = this.table[idx].value;
    this.table[idx] = this.tombstone;
    return result;
}
```

Modifikasikan juga method `insert()` dan `search()` untuk menangani `tombstone`! Pada method `insert()`, jika ditemukan lokasi berisi `tombstone`, maka lokasi tersebut boleh diisi dengan `Data` baru.

5. Tester

Ujilah method-method anda. Misalnya dengan kelas Tester berikut ini:

```
class TesterHash{
    public static void main(String[] args){
        ModularHashInteger<String> h = new ModularHashInteger<String>(11, 3, 2);
        h.insert(5, "Alice");
        h.insert(16, "Bob");
        h.insert(27, "Charlie");

        h.delete(16);

        System.out.println(h.search(5));
        System.out.println(h.search(16));
        System.out.println(h.search(27));
    }
}
```

Pada contoh ini, ada 3 `Data` yang dimasukkan, dan semuanya akan saling bertabrakan (mulai pada h_0=5). Kemudia `Data` kedua dihapus. Maka hasil yang diharapkan adalah search pertama dan ketiga mengembalikan `value`, sedangkan yang kedua mengembalikan `null`. Jika implementasi tombstone ada yang salah, maka kemungkinan data ketiga jadi tidak ditemukan.

Output:

```
Alice
null
Charlie
```