Modul 6 Implementasi Struktur Data
Hashing dan Map

6.1. Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal Konsep dan Implementasi Hashing dan Hash Table pada Java
2. Mengenal Konsep dan Implementasi Mapping pada Java

6.2. Alat dan Bahan

Alat & Bahan yang digunakan adalah hardware perangkat PC beserta kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC

6.3. Dasar Teori

Setelah mempelajari struktur dari data Array dan List yang memiliki suatu yang sama yaitu: Struktur data ini akan menyimpan data/ element pada tempat sesuai dengan dimana kita menyimpannya. Tetapi berbeda dengan Hashing dan Map, disini kita tidak akan memperhatikan dimana kita meletakkan datanya, karena indeks bukan ditentukan oleh kita tetapi dari algoritma HashCode.

Kenapa harus menggunakan Hashing? anggap saja kalian membuat list/ *collection* dengan banyak data, tetapi tanpa memperhatikan urutan dari datanya yang disebut ***set***. Dalam *set* tersebut, **tidak boleh ada** duplikasi elemen yang tersimpan. Proses ini lebih cepat dalam proses penambahan, penghapusan, dan pencarian dari pada menggunakan Array / Linked List, dikarenakan kenapa? karena kita menggunakan indeks untuk pencarian apakah data tersebut sudah terisi pada indeks tersebut, maka pencarian berhasil, untuk penambahan yang telah ada pada indeks tersebut, maka akan tertolak, begitu pula dalam penghapusan, proses akan menolak jika dalam indeks tersebut tidak ada data yang terisi.

Contohnya
Pada *Collection* Mahasiswa, kalian ingin menempatkan data Mahasiswa berdasarkan nama, kalian bisa menggunakan HashCode, jadi hasil dari HashCode tersebut disebut indeks, yang akan digunakan untuk menempatkan data pada *Collection*.

6.3.1 Hashing

Pada suatu pencarian sebuah array, umumnya digunakan dengan metode Linear atau Pencarian Linear yang kalian tidak perlu dilakukan jika kalian menggunakan teknik *Hashing*.

Hashing adalah suatu teknik yang digunakan untuk menentukan indeks pada suatu array melalui *key* yang disediakan tanpa perlu lagi melakukan pencarian linear. HashCode berfungsi dengan mengubah *key* menjadi indeks bertipe integer dari suatu objek dan hasil dari HashCode ini akan selalu unik, jadi *key* yang berbeda akan menghasilkan HashCode yang berbeda juga. Anggap saja kalian sebagai Mahasiswa, dan sebagai Mahasiswa telkom pasti kalian punya nama yang kembar dengan yang lainnya kan? Contohnya: Udin, dan Udin Syam, mereka berdua memiliki kesamaan nama yaitu Udin, nah nama mereka itu adalah **key,** dan mereka berdua pasti memiliki kepribadian yang berbeda kan? sama seperti HashCode.

Key	HashCode
Kaiser	-2054957619
Kunigami	1574846011
Sae	82871

Gambar 1 Contoh hasil HashCode

Seperti yang telah disebutkan sebelumnya, setiap Key disana dan hasil hash code nya menggunakan *method* bawaan Java yang dapat langsung digunakan, untuk pemanggilan *method* hashcode adalah sebagai berikut.

```
String x = "contoh key";
int h = x.hashCode();
```

Collision akan terjadi jika terdapat dua objek yang memiliki hash code yang sama. Sangat jarang terjadi *Collision* pada tipe data String. Apabila terjadi *Collision* alangkah baiknya untuk melakukan hashing ulang dan mengubah rumusnya. Hash function yang baik tanpa terjadi *Collision* disebut sebagai Perfect Hash Function. Untuk melakukan pencarian suatu objek apakah ada di *set* tersebut atau tidak kalian hanya perlu menghitung hash code dari objek tersebut dan melihat apakah array dengan indeks yang sesuai dengan hash code tersebut telah berisi objek atau tidak.

Namun, pada kasus array yang membatasi kapasitasnya untuk menampung semua data yang ada. Biasanya, dilakukan komputasi modulasi yang berfungsi mengurangi skala untuk menyimpan objek dan biasanya akan terjadi *Collision* dikarenakan hasil dari hash code dari objek akan dimodulasi dengan besarnya array.

```
int[] buckets = new int[5];
String x = "contoh key";
int h = x.hashCode();
if (h < 0) h = -h;
int i = h % buckets.length;
```

Code diatas dapat juga digunakan sebagai salah satu *method* penanganan *Collision*, yaitu dengan menggunakan *separate chaining*. Algoritma *separate chaining* untuk pencarian adalah sebagai berikut:

1. Menghitung hash code untuk objek x, hasil dari hashcode akan di modulus dengan ukuran array. Dengan begitu kita mendapatkan indeks h untuk hash tabelnya.
2. Iterate/ pencarian seluruh elemen pada bucket pada posisi h. Setiap element pada bucket, dicek apakah elemen ini sama dengan x?.
3. Jika terdapat elemen yang sama, berarti x sudah di *set*. Jika tidak, maka x tidak ditemukan.

Algoritma *separate chaining* untuk menambahkan objek adalah sebagai berikut :

1. Menghitung hash code untuk objek x, seperti yang dilakukan di pencarian, ini bertujuan untuk menemukan indeks bucket.
2. Iterate/ pencarian seluruh elemen pada bucket tersebut. Jika sudah ada, maka abaikan (dikarenakan *set* tidak mengizinkan data yang terduplikasi). Jika tidak ada, maka masukan objeknya.

Algoritma *separate chaining* untuk menghapus objek adalah sebagai berikut :

1. Menghitung hash code untuk objek x, seperti yang dilakukan diatas-atasnya, bertujuan untuk menemukan indeks bucket.
2. Iterate/ pencarian seluruh elemen pada bucket tersebut. Jika objek ketemu, maka dihapus.

6.3.2 Studi Kasus

Kita akan memasuki bab Studi kasus, disini kita akan mengimplementasikan Hashing pada sebuah program Java. Sebelum memasuki codingan, kalian harus tau setiap method yang ada di “HashSet.java” nanti yang kita akan buat

Table 1 Menunjukan List-list lengkap method pada HashSet

Modifier and Type	Method	Keterangan
public int	hash(Object x)	Melakukan hashing
public boolean	contains(Object x)	Melakukan pencarian apakah object ada
public int	size()	Mengambil jumlah objek dari hash table
public boolean	add(Object x)	Menambahkan Objek pada set hash table
public boolean	remove(Object x)	Menghapus Objek pada set hash table
public void	print()	Mencetak seluruh isi data hash table

Setelah kalian memahami setiap fungsi itu apa, kita akan langsung membuat *class* baru bernama "**HashSet.java**" untuk tempat menampung sebuah hash table, lalu buatlah variable buckets dan size, variable buckets adalah sebagai penampung hash table dengan menggunakan *class* Node, yang sudah diajarkan sebelumnya dan dipakai pada Singly Linked List

HashSet.java

```java
public class HashSet {  no usages
    private Node[] buckets;  1 usage
    private int size;  1 usage

    /**
     * Constructor hash table jika tidak menggunakan parameter
     * default: 10
     */
    public HashSet() {  no usages
        this( bucketsLength: 10);
    }

    /**
     * Constructor hash table
     *
     * @param bucketsLength ukuran bucket
     */
    public HashSet(int bucketsLength) {  1 usage
        buckets = new Node[bucketsLength];
        size = 0;
    }
}
```

Lalu, buatlah *class* lagi bernama "**Node.java**", buat variable data untuk menampung data yang akan dimasukan kepada hash table nanti, dan variable next untuk kebutuhan iterator.

Node.java

```java
public class Node {  10 usages
    public Object data;
    public Node next;  8 usages

    /**
     * Node Constructor, jika parameter tidak digunakan
     */
    public Node() { this( data: null); }

    /**
     * Node Constructor
     * @param data an object
     */
    public Node(Object data) {  2 usages
        this.data = data;
        this.next = null;
    }
}
```

Setelah kita sudah membuat *class* HashSet dan Node, maka kita akan menambahkan *method* dibawah untuk *class* HashSet.java.

Lalu, buat *method* hash(Object x) dibawah *constructor*, dan *method* ini akan melakukan hashCode pada x, dan mengembalikan hasil yang sudah dimodulasi dengan ukuran buckets.
Kenapa di modulasi? Karena ini menggunakan metode *separate chaining* untuk mendapatkan indeks-h, maka harus dimodulasi dengan ukuran bucket-nya. Seperti codingan dibawah.

HashSet.java

```
/**
 * Berfungsi untuk melakukan hashing.
 *
 * @param x an object
 * @return angka hashing
 */
public int hash(Object x) {  4 usages
    int h = x.hashCode();
    if (h < 0) h = -h;
    return h % buckets.length;
}
```

Lalu *method* contains(Object x) di bawah hash. *Method* ini berfungsi untuk mengecek apakah object sudah ada dalam. Seperti codingan dibawah.

HashSet.java

```
/**
 * Berfungsi untuk mengecek Objek ada didalam set atau tidak.
 *
 * @param x an object
 * @return true jika objek ditemukan, dan false jika tidak ditemukan
 */
public boolean contains(Object x) {  1 usage
    Node iterator = buckets[hash(x)];
    while (iterator != null) {
        if (iterator.data.equals(x)) {
            return true;
        }
        iterator = iterator.next;
    }
    return false;
}
```

Bagaimana dengan *method* size()? Ini berfungsi untuk mendapatkan jumlah **Node** yang ada pada hash table. Seperti codingan dibawah.

HashSet.java

```java
/**
 * Berfungsi untuk mendapatkan jumlah element pada set.
 *
 * @return jumlah element
 */
public int size() {  no usages
    return size;
}
```

Untuk *method* add(Object x) berfungsi untuk menambahkan objek pada set. Objek yang dicari dalam iterator tidak ketemu **dan** iterator masih null (belum ada Node), maka buat Node baru dengan parameter(x), lalu mengatur Next dari Node ke buckets[h] (kalau baru membuat Node maka buckets[h] dianggap null karena (iterator == null), dan buckets[h] diarahkan ke newNode, serta size++ yang akan menambahkan jumlah pada variabel size sebagai indikator jumlah objek dalam hash table. Seperti codingan dibawah.

HashSet.java

```java
/**
 * Berfungsi untuk menambahkan Objek pada set.
 *
 * @param x an object
 * @return true jika x adalah objek baru dan belum di set
 */
public boolean add(Object x) {
    int h = hash(x);
    Node iterator = buckets[h];

    if (iterator != null) {
        if (contains(x)) {
            return false;
        }
    }

    Node newNode = new Node(x);
    newNode.next = buckets[h];
    buckets[h] = newNode;
    size++;

    return true;
}
```

Pada fungsi add tersebut, kita akan mengecek apakah iterator tersebut sudah ada Node atau belum, jika sudah maka dia melakukan pengecekan apakah dalam iterator sudah ada x (objek), jika sudah maka jangan lakukan apapun / return false.

Lalu setelah *method* add, maka kita akan menambahkan *method* penghapusan yaitu remove(Object x). Seperti codingan dibawah.

HashSet.java

```java
/**
 * Berfungsi untuk melakukan penghapusan Objek pada set.
 *
 * @param x an object
 * @return true jika x berhasil terhapus, false jika x tidak ditemukan
 */
public boolean remove(Object x) {  1 usage
    Node iterator = buckets[hash(x)];
    Node previous = null;

    while (iterator != null) {
        if (iterator.data.equals(x)) {
            if (previous == null) {
                buckets[hash(x)] = iterator.next;
            } else {
                previous.next = iterator.next;
            }

            size--;
            return true;
        }
        previous = iterator;
        iterator = iterator.next;
    }

    return  false;
}
```

Cara kerja penghapusan diatas yaitu mirip dengan cara penghapusan di Singly Linked List, yaitu dengan memberi iterator sebelumnya variabel, bertujuan untuk dapat mengatur ulang .next pada iterator sebelumnya tersebut.

Pertama-tama kita melakukan loop iterator sampai kita menemukan Objek yang ingin kita hapus, jika Objek ketemu maka dicek lagi apakah objek tersebut berada di Head? Jika iya maka langsung mengatur buckets[hash(x)] ke iterator selanjutnya, dan jika tidak maka diatur iterator sebelumnya .next nya ke iterator selanjutnya.

Lalu terakhir, kita akan membuat *method* print yang akan mencetak seluruh dari isi data, ini juga mirip dengan cara cetak di Singly Linked List. Seperti codingan dibawah.

HashSet.java

```java
/**
 * Berfungsi untuk mencetak seluruh data HashSet
 */
public void print() {  2 usages
    for (int i = 0; i < buckets.length; i++) {
        Node iterator = buckets[i];
        if (iterator != null) {
            System.out.printf("index %d : ", i);
            while (iterator != null) {
                System.out.print(iterator.data + " ");
                iterator = iterator.next;
            }
            System.out.println();
        }
    }
}
```

Sebenarnya, Java sudah menyediakan *class* bawaan HashSet. Namun tujuan kita membuat ulang *class* HashSet disini untuk memahami bagaimana alur dan codingan algoritma hashing yang menggunakan *separate chaining*. Kedepannya disarankan untuk langsung menggunakan HashSet.

6.3.3 Contoh Penggunaan

Buka *class* **Main.java** lalu deklarasi variabel list dengan *class* HashSet yang sudah dibuat tadi, lalu masukan panjang array ke parameter (defaultnya adalah 10). Yang dicontohkan di bawah ini.

```java
HashSet list = new HashSet();
```

Kenapa defaultnya 10? Karena pada *class* HashSet disana ada 2 *constructor*, jika parameter tidak diberikan maka *constructor* yang terpakai adalah yang dibawah ini.

```java
public HashSet() {  no usage
    this( bucketsLength: 10);
}
```

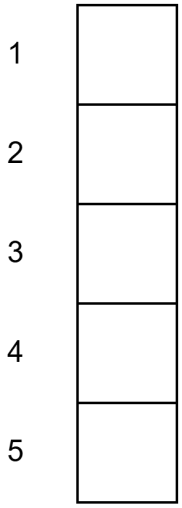
Jika diberi parameter, maka *constructor* yang terpakai adalah yang dibawah ini.

```java
public HashSet(int bucketsLength) {
    buckets = new Node[bucketsLength];
    size = 0;
}
```

Pada kali ini, kita akan membuat HashSet dengan panjang 5 pada *class* **main.java.** Codingan akan seperti di bawah ini.

```java
HashSet list = new HashSet( bucketsLength: 5);
```

Ketika sudah membuat HashSet dengan panjang 5, maka ilustrasi visualnya akan seperti di (Gambar 2).



Gambar 2 Ilustrasi visual Hash

Setelah kita mendeklarasi HashSet tersebut, maka kita akan menambahkan Objek-objek dibawah ini.

Isagi
Ego
Rin
Lorenzo
Sae

Masukan data dari tabel diatas dengan menggunakan method add(Object x). Contoh cara menambahkannya seperti dibawah ini.

```
list.add("Isagi");
list.add("Ego");
list.add("Rin");
list.add("Lorenzo");
list.add("Sae");
```

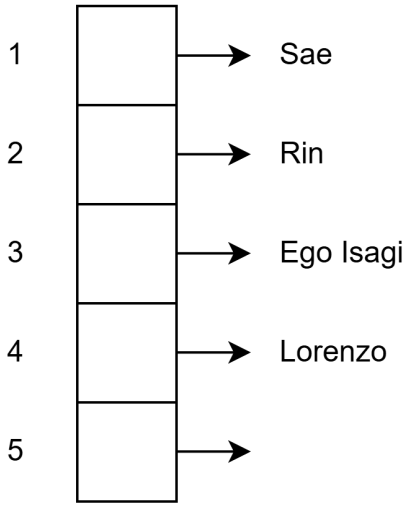
Output (hasil Hashing perObjek) : 3,3,2,4,1

Hasil output diatas adalah indeks yang akan dimasukan kedalam hash table, dan karena Isagi dan Ego memiliki indeks yang sama, maka akan langsung ditambahkan di sebagai Head di Node indeks 3 pada lalu coba kalian cetak dengan menggunakan *method* print() hasil akan seperti di (Gambar 3).

```
index 1 : Sae
index 2 : Rin
index 3 : Ego Isagi
index 4 : Lorenzo
```

Gambar 3 Output dari print setelah penambahan objek

Seperti gambar diatas, Ego dan Isagi berada di index yang sama, dan untuk ilustrasi visualnya akan seperti di (Gambar 4).



Gambar 4 Ilustrasi Visual setelah menambahkan objek

Sekarang, coba kalian hapus “Ego” dengan menggunakan *method* remove(Object x).

```
list.remove( x: "Ego");
```

Setelah itu kalian coba cetak lagi dengan menggunakan *method* print() hasil akan seperti di (Gambar 5).

```
index 1 : Sae
index 2 : Rin
index 3 : Isagi
index 4 : Lorenzo
```

Gambar 5 Output dari print setelah penghapusan

6.4.1 Map

Kenapa sih kita harus menggunakan Map? Coba kalian pikirkan ada berapa banyak mahasiswa yang berkuliah di Telkom University, tidak menutup kemungkinan bahwa akan ada orang dengan nama yang sama namun berbeda jurusan maupun fakultas, jadi bagaimana cara Telkom University dapat membedakan setiap mahasiswanya? Benar Telkom menggunakan sistem bernama NIM yang memiliki fungsi sebagai Key dalam suatu Map.

Key dalam suatu Map dapat mengarah ke berbagai value contohnya dengan NIM kita dapat mengetahui nama, kelas, fakultas bahkan tahun masuk dari seorang Mahasiswa. Untuk mendapatkan NIM tidak mungkin Telkom membuatnya secara manual bukan? tentu saja tidak mungkin karena itu akan memakan banyak waktu, seperti penjelasan di atas mengenai Hashing di mana Hashing menggunakan suatu algoritma pemrograman yang akan membuat suatu key yang bersifat unik sehingga tidak akan terjadi tabrakan data maupun duplikasi data.

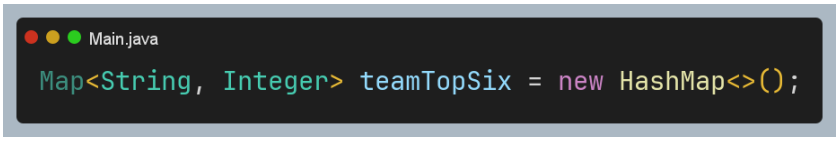
6.4.2 Contoh Penggunaan

Setelah mengetahui cara kerja dari suatu map, selanjutnya kita akan mencoba melakukan mapping menggunakan HashMap. Sebelum itu berikut adalah method yang akan kita gunakan.

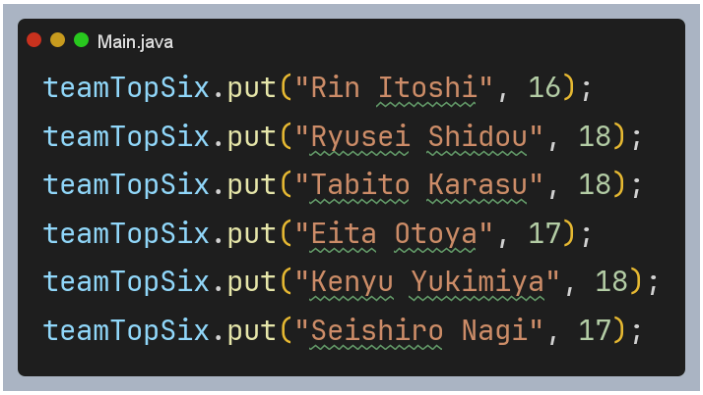
Modifier and Type	Method	Keterangan
`Set<K>`	keySet()	Mengembalikan semua key yang ada pada map
`V`	put()	Memasukkan value dan key kedalam map
`int`	size()	Mengembalikan jumlah masukkan pada map
`boolean`	cont	

Pertama tama kita akan melakukan import, dikarenakan java telah menyediakan kelas untuk melakukan HashMap maka kita tinggal melakukan import untuk memanggil function functionnya yang telah disediakan.

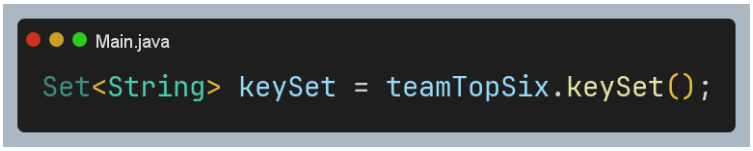
Lalu kita akan membuat HashMap dengan tipe data String sebagai Key dan Integer sebagai valuenya, seperti gambar dibawah.



Lalu kita akan memasukkan berbagai macam data, namun perlu diingat bahwa Key haruslah unik dimana String pada code diatas merupakan key.



Lalu kita akan melakukan Set untuk mengSet key yang ada pada HashMap kita.



Terakhir kita akan melakukan metode print menggunakan For each serta jangan lupa untuk melakukan set value dimana value pada kode diatas adalah variabel umur yang berupa int.

Main.java
```
for (String key : keySet)
{
    int umur = teamTopSix.get(key);
    System.out.println("Nama : " + key + ", Umur : " + umur);
}
```

Pada saat dijalankan akan mengeluarkan hasil sebagai berikut.

```
Nama : Rin Itoshi, Umur : 16
Nama : Kenyu Yukimiya, Umur : 18
Nama : Tabito Karasu, Umur : 18
Nama : Seishiro Nagi, Umur : 17
Nama : Ryusei Shidou, Umur : 18
Nama : Eita Otoya, Umur : 17
```

6.5.1 Hash Code pada Kelas Bentukan

Pernah tidak kalian berfikir, “bagaimana sih cara kerja dari hashing?” dan “Apakah kita bisa membuat hashing berdasarkan keinginan kita tanpa menggunakan method bawaan dari java?”. Tentu saja kita bisa membuat hashing berdasarkan keinginan kita, seperti yang telah dijelaskan pada bab **6.3.1** mengenai hashing. Pada sub bab ini kita akan membuat method hashing berdasarkan keinginan kita

6.5.2 Contoh Penggunaan

Karena kita membuat method hashing atau hashcode berdasarkan keinginan kita sendiri, Perhitungan hash code pada fungsi hash pada dasarnya berbeda beda berdasarkan programmer yang membuat kode-nya, yang harus diperhatikan adalah hasil dari suatu hash code haruslah **Unik** antara objek yang berbeda. Berikut adalah contoh hash code yang kurang baik.

VanillaHash.java
```
public static int hashCode(String input) {  2
    int hash = 0;
    for(int i = 0; i < input.length(); i++) {
        hash = hash + input.charAt(i);
    }
    return hash;
}
```

Mengapa kurang baik mari kita bahas lebih lanjut, pada function diatas ada variabel hash yang akan berfungsi sebagai tempat untuk menyimpan hasil hashcode, dapat diperhatikan bahwa hash code diatas menggunakan loop, dimana dia akan mengubah setiap huruf dalam input menjadi int dan menjumlahkannya, pasti kalian bingung kenapa charAt bisa dijumlahkan, hal tersebut dapat terjadi dikarenakan setiap huruf memiliki **Unicode** yang berbeda beda dimana yang akan dijumlahkan adalah unicode dari setiap huruf, namun oleh karena itu hal ini kurang efektif mengapa? Mari kita lihat contoh masukkan dan keluarannya

Berikut gambar dibawah ini adalah contoh masukkan dan cara penggunaan fungsinya, dapat dilihat ada nama dan nama1 yang memiliki jumlah dan kapital yang sama dimana perbedaannya terdapat pada nama mereka yang dibalik. Oleh karena itu hasil yang akan dikeluarkan juga akan tetap sama dikarenakan jumlah dan hurufnya sama hanya posisinya saja yang ditukar

VanillaHash.java

```
public static void main(String[] args) {
    String nama = "Bachira Meguru";
    String nama1 = "Meguru Bachira";

    System.out.println("Hasil hashCode dari (Bachira Meguru) = " + hashCode(nama));
    System.out.println("Hasil hashCode dari (Meguru Bachira) = " + hashCode(nama1));
}
```

Dapat dilihat hasil dari hashing ke 2 String diatas menghasilkan angka yang sama, dimana hal tersebut kurang baik dikarenakan key haruslah unik pada suatu map.

```
Hasil hashCode dari (Bachira Meguru) = 1343
Hasil hashCode dari (Meguru Bachira) = 1343
```

Berikut adalah contoh penggunaan hashing yang tepat

VanillaHash.java

```
public static int hashCode2(String input) {  2 usages
    final int HASH_MULTIPLIER = 31;
    int hash = 0;
    for(int i = 0; i < input.length(); i++) {
        hash = hash * HASH_MULTIPLIER + input.charAt(i);
    }
    return hash;
}
```

Dapat dilihat dari function di atas kita menggunakan perkalian saat melakukan hashing, dimana hal tersebut dapat membuat angka yang dikembalikan menjadi lebih rumit dan tidak mungkin sama.

Ini adalah contoh masukkan dan penggunaan function hashCode ke 2 yang kita buat

```java
String nama = "Bachira Meguru";
String nama1 = "Meguru Bachira";

System.out.println("Hasil hashCode dari (Bachira Meguru) = " + hashCode2(nama));
System.out.println("Hasil hashCode dari (Meguru Bachira) = " + hashCode2(nama1));
```

Ini adalah contoh keluarannya

```
Hasil hashCode dari (Bachira Meguru) = -867809739
Hasil hashCode dari (Meguru Bachira) = 84703517
```

Dapat dilihat hasilnya berbeda walaupun namanya hanya dibalik.

> Penjelasan!
>
> Perhitungan integer bagi setiap karakter menggunakan nilai ASCII untuk setiap karakter

6.5.3 Studi Kasus

Setelah mengetahui cara penggunaan hashing yang dan mapping dengan baik, selanjutnya kita akan membuat daftar pemain sepak bola, dimana key-nya akan dibuat berdasarkan nama pemain dan posisi dari si pemain sepak bola menggunakan metode hashing.

Langkah pertama yang perlu dilakukan adalah membuat class pojo dengan nama Pemain.java.

```java
private String nama;  5 usages
private String posisi;  5 usages

public Pemain(String nama, String posisi) {
    this.nama = nama;
    this.posisi = posisi;
}
```

Masih di kelas Pemain.java, selanjutnya kita akan membuat function hash code di kelas pojo kita, perlu diingat kita harus menggunakan @Override agar object yang telah dipetakan dapat diambil kembali, kemudian agar hasil hashing lebih maksimal gunakan bilangan prima untuk konstanta

Pemain.java

```
@Override
public int hashCode() {
    final int HASH_MULTIPLIER = 31; //Bisa menggunakan bilangan prima lainnya
    int hash = 0;
    for(int i = 0; i < nama.length(); i++) {
        hash = hash * HASH_MULTIPLIER + nama.charAt(i);
    }
    for(int i = 0; i < posisi.length(); i++) {
        hash = hash * HASH_MULTIPLIER + posisi.charAt(i);
    }
    return hash;
}
```

HASH_MULTIPLIER. Begitu juga dengan function equals kita juga harus menggunakan @Override agar objectnya dapat diambil kembali. Sebelum itu object dapat dikatakan sama apabila object tersebut memiliki hashCode yang sama persis.

Pemain.java

```
@Override
public boolean equals(Object object) {
    if (this == object) {
        return true;
    }
    if (object == null || getClass() != object.getClass()) {
        return false;
    }
    Pemain pemain = (Pemain) object;

    return nama.equals(pemain.nama) && posisi.equals(pemain.posisi);
}
```

Gunakan juga toString() agar saat kita ingin melakukan print suatu data dari kelas object, data yang muncul bukan alamat dari data object tersebut.

Pemain.java

```
@Override
public String toString() {
    return  "Nama: " + nama + "\n" +
            "Posisi: " + posisi + "\n";
}
```

Berikutnya kita akan membuat kelas Main yang akan menjalankan semua function yang telah kita buat serta untuk membuat object objectnya. Pertama tama kita akan membuat import utility java terlebih dahulu.

Main.java

```
import java.util.HashMap;
import java.util.Map;
```

Lalu kita akanmembuat HashMapnya, jangan lupa bahwa tipe data yang diperbolehkan sebagai Key maupun value haruslah tipe data object atau reference

Main.java

```
HashMap<Integer, Pemain> mapPemain = new HashMap<>();
```

Berikut merupakan data data yang nantinya akan kita pakai untuk implementasi function function yang telah kita buat

Main.java

```
Pemain player1 = new Pemain( nama: "Don Lorenzo",  posisi: "CM");
Pemain player2 = new Pemain( nama: "Marc Snuffy",  posisi: "CF");
Pemain player3 = new Pemain( nama: "Gagamaru",  posisi: "GK");
Pemain player4 = new Pemain( nama: "Nagi",  posisi: "RW");
Pemain player5 = new Pemain( nama: "Chigiri",  posisi: "LW");
Pemain player6 = new Pemain( nama: "Don Lorenzo",  posisi: "CM");
```

selanjutnya kita akan melakukan langkah hashing yang bertujuan membuat key yang unik dari suatu map, dan membuat key tersebut menggunakan function yang telah kita buat di dalam kelas pojo

Main.java

```
int player1Code = player1.hashCode();
int player2Code = player2.hashCode();
int player3Code = player3.hashCode();
int player4Code = player4.hashCode();
int player5Code = player5.hashCode();
int player6Code = player6.hashCode();
```

Lalu kita akan memasukkan key yang telah kita buat (playerCode) kedalam HashMap, serta value yang telah kita buat pada langkah ke dua yaitu (player)

Main.java

```
mapPemain.put(player1Code, player1);
mapPemain.put(player2Code, player2);
mapPemain.put(player3Code, player3);
mapPemain.put(player4Code, player4);
mapPemain.put(player5Code, player5);
mapPemain.put(player6Code, player6);
```

Selanjutnya kita akan mencetak isi dari map yang telah kita buat

Main.java

```
System.out.println("Daftar Pemain");
for (Map.Entry<Integer, Pemain> entry : mapPemain.entrySet()) {
    System.out.println("Kode Pemain : " + entry.getKey() + "\n"+ entry.getValue());
}
```

Berikut adalah hasil dari printnya, dapat dilihat bahwa yang tercetak hanyalah 5 pemain padahal sebelumnya kita masukkan 6 pemain, hal tersebut dapat terjadi dikarenakan ada 2 pemain yang sama sehingga yang ditampilkan hanya salah satu

```
Daftar Pemain
Kode Pemain : -2068928945
Nama: Gagamaru
Posisi: GK

Kode Pemain : 753427357
Nama: Marc Snuffy
Posisi: CF

Kode Pemain : 1771161448
Nama: Chigiri
Posisi: LW

Kode Pemain : -2073961542
Nama: Don Lorenzo
Posisi: CM

Kode Pemain : -1969139974
Nama: Nagi
Posisi: RW
```

Langkah terakhir kita harus mengecek apakah ada pemain yang merupakan pemain yang sama, kita akan melakukan hal tersebut menggunakan function equals() yang telah kita buat sebelumnya

Main.java

```java
    if (player1.equals(player6) && player1.hashCode() == player6.hashCode()) {
        System.out.println("Player 1 dan Player 6 adalah pemain yang sama");
    } else {
        System.out.println("Player 1 dan Player 6 adalah pemain yang berbeda");
    }

    if (player1.equals(player2) && player1.hashCode() == player2.hashCode()) {
        System.out.println("Player 1 dan Player 2 adalah pemain yang sama");
    } else {
        System.out.println("Player 1 dan Player 2 adalah pemain yang berbeda");
    }
}
```

Berikut adalah hasil printnya dapat dilihat dengan menggunakan kode diatas kita dapat mengidentifikasi pemain yang sama atau duplikat

```
Player 1 dan Player 6 adalah pemain yang sama
Player 1 dan Player 2 adalah pemain yang berbeda
```

Dengan begitu selesai sudah materi Hash & Map kali ini, Gimana dengan materi hari ini? Seru bukan?

Selamat Belajar dan Semangat Codingnya!!!