Hoofdstuk 9

Hashing

Het zoeken in een verzameling van één object is in deze cursus al verschillende malen aan bod gekomen. In hoofdstuk 2 werd uitgelegd hoe men een object kan zoeken in een array die veel objecten bevat. Indien in de array de records in een willekeurige volgorde voorkomen, is de zoektijd evenredig met *n*, het aantal elementen. Daarentegen, wanneer de elementen volgens sleutelwaarde geordend zijn, is de zoektijd veel korter. (In de orde van $\log_2 n$.) Voor een array moet zijn afmeting op voorhand worden gedeclareerd. Dat is, gezien de omvang van de verzameling objecten, een vervelende beperking.
In hoofdstuk 7 werden zoekbomen (binaire bomen) besproken. Dergelijke bomen worden dynamisch opgebouwd. Het aantal elementen moet dus niet a priori bekend zijn, omdat in deze bomen de gegevens ook geordend zijn, is de zoektijd in de orde van $\log_2 n$. De combinatie van dynamisch geheugen en van een logaritmische zoektijd verklaart waarom bomen de meest gebruikte techniek zijn voor het zoeken van een record in grote verzamelingen gegevens.
In dit hoofdstuk wordt een derde zoekmethode uitgelegd. Deze is meestal nog veel sneller dan het gebruik van bomen, maar hij vereist een betrouwbare bovenlimiet voor het aantal objecten die a-priori bekend is. Deze methode heet **hashing**. Het is een belangrijke, veelgebruikte methode.

9.1. Hashfuncties

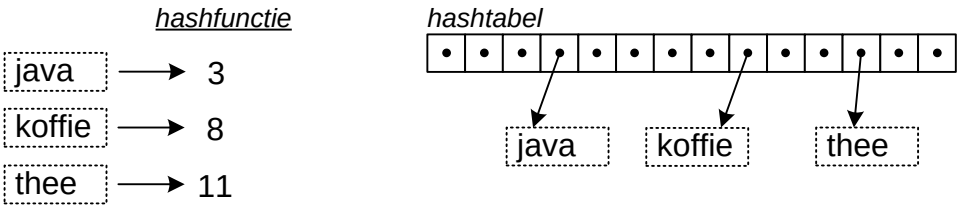
Mathematisch gezien is een zoekalgoritme een functie die voor elke sleutelwaarde een index of een geheugenadres berekent (of zoekt). Als we zoeken in een niet-geordende array bepaalt de functie het adres door opeenvolgend alle waarden van de index te proberen. In het geval van bomen gebeurt dit door een keten pointers te volgen. Uiteraard is het mogelijk andere functies te bedenken die met elk mogelijke sleutel een index in een array associëren.

We gaan proberen een methode op te stellen om op een directe manier de plaats van een object in een array te bepalen. Dit doen we door een functie op te stellen die de index geeft aan de hand van de sleutel.

Hiervoor zetten we alfabetische sleutels om in getallen door bijvoorbeeld de binaire ASCII codes van alle karakters naast elkaar te schrijven. Een alfabetische sleutel met twintig karakters wordt zo een 20*8=160-bit numerieke sleutel. Een karakter in ASCII beschrijft men met 8 bits.
Als we aannemen dat er nooit meer dan *n* elementen zullen worden gebruikt, kunnen we een array declareren waarvan de index een waarde zal hebben tussen 0 en *n*-1. Om met elke (numerieke) sleutel een indexwaarde te associëren, volstaat het dan *sleutel MOD (n)* als index te gebruiken (MOD is de modulo-operator).

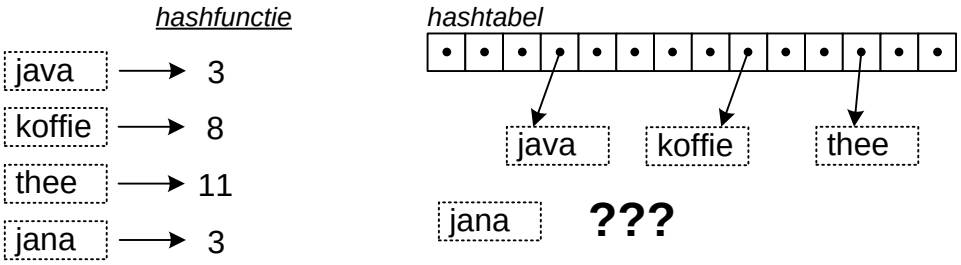
De functie die, vertrekkend van een sleutel een indexwaarde bepaalt, noemt men een **hashfunctie**. Het interessante aan dergelijke functies is dat de tijd die nodig is om de waarde van de index te bepalen onafhankelijk is van *n*, wat wil zeggen dat de zoektijd

niet stijgt met de omvang van de zoekruimte. We zeggen dat het algoritme van O(0) is. Dit is aantrekkelijk voor zeer grote gegevensbanken. De zoektijd hangt enkel af van de tijd om de functie uit te rekenen. De functie mag dan ook niet te ingewikkeld zijn.



In dit eerste voorbeeldje slaan we drie strings op. De waarde van de hashfunctie is voor elke string gegeven. Rechts zien we de plaats waar de strings opgeslagen worden in de array.

Elke mogelijke sleutelwaarde stemt overeen met een unieke indexwaarde, maar niets kan garanderen dat er geen twee verschillende sleutels tot dezelfde indexwaarde zullen leiden. Wanneer dit het geval is, spreekt men van een ***botsing***. Uiteraard is het wenselijk dat de gebruikte sleutels zo min mogelijk aanleiding tot botsingen geven. Stel dat "jana" dezelfde hashwaarde geeft als "java", namelijk 3. Dit geeft een probleem omdat het vakje met index 3 al is ingenomen. Oplossingen hiervoor bespreken we in de volgende sectie.



Laat ons eerst een hashfunctie uitwerken.
We kunnen de eerste letter nemen van elk woord ('a' = 0; 'b' = 1; ...). Dan zal de index tussen 0 en 25 liggen. Als we de eerste 2 letters van elk woord volgens de volgende formule:

```
index(woord) = index(eerste letter) * 26 + index(tweede letter)
```

Dan krijgen we een index tussen 0 ("aa") en $25*26+25=26^2-1$ ("zz"). Merk op dat we de eerste letter vermenigvuldigen met een factor zodat die de meeste impact heeft. Alle woorden die beginnen met dezelfde waarde zitten dus samen. De array moet dus een grootte van 26^2 hebben. Groter heeft geen zin, want het einde van array wordt dan niet gebruikt. Meestal wordt een maximale hashwaarde genomen die groter is dan de arraygrootte en vervolgens de modulo nemen:

```
index = hashwaarde modulo arraygrootte.
```

De modulo zorgt er voor dat de index binnen de grootte van de array valt.

In het voorbeeld komt de botsing doordat we enkel de eerste twee letters van het woord gebruikt hebben om de hashwaarde te bepalen. Het is dus beter dat *alle* karakters van de sleutel een invloed hebben op de waarde van de index. Dit is niet altijd eenvoudig. Laat ons de hashwaarde met bvb 8 bits uitdrukken en stel dat de

arraygrootte *n* een macht van 2 is. Dan zijn enkel de $\log_2 n$ minst-betekenisvolle bits van de numerieke sleutel bepalend zijn voor de indexwaarde. Immers, door de *MOD* $\log_2 n$ operatie zullen alle bits groter dan *n* niet meetellen in de hashwaarde. Stel dat onze sleutels bestaan uit 8 bits (waarden tussen 0 en 255) en dat n=16. $\log_2 n$ is dan 4 wat betekent dat enkel de 4 rechtse bits gebruikt zullen worden. De hashwaarden 0000110, 01010110, 10000110 en 11110110 geven allen dezelfde index, namelijk 0110, want de modulo van een macht van 2 resulteert in het 'afkappen' van linkse bits. De vier meest-betekenisvolle bits worden dus niet gebruikt. Dit is het geval in ons voorbeeld. Het is dus beter om bijvoorbeeld *n* een priemgetal te nemen.

In het algemeen wensen we te bekomen dat de hashwaarde een kans op botsing geeft evenredig met de vullingsgraad van de array. Als de array voor *p*% gevuld, moet de kans op botsing *p*% zijn. In ons geval is dat niet, voor sommige beginletters zijn er meer woorden dan voor andere, omdat de beginletter de regio van de array bepaalt. Die vakken zullen dus meer gevuld zijn.

Enkel wanneer de volledige verzameling sleutels a priori bekend is, kunnen we in plaats van de modulo-hashfunctie, andere, complexere functies aanwenden die het voordeel hebben geen botsingen te veroorzaken (**perfect hashing**). Dit wordt bijvoorbeeld gebuikt in een compiler om de gereserveerde woorden (zoals *class*, *public*, *int*, …) snel te herkennen.

Merk op dat java automatisch een hashfunctie definieert voor elke klasse. Deze staat gedefinieerd in de **Object**-klasse als:

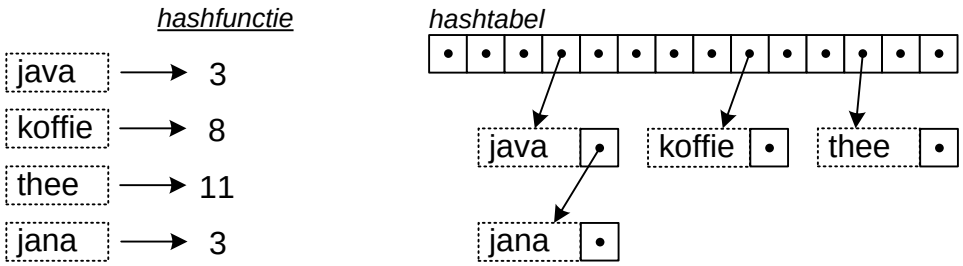
```
int hashCode()
Returns a hash code value for the object.
```

Ze neemt de waarden van alle attributen in rekening. Omdat alle javaklassen automatisch een subklasse zijn van Object, hebben alle klassen deze methode. Je kan ze overschrijven zodat je eigen, betere hashfunctie gebruikt wordt.

9.2. Oplossen van botsingen

Enkel bij perfect hashing komen geen botsingen voor, maar in het algemeen moeten we van uit gaan dat meerdere records op hetzelfde adres opgeslagen moeten worden.

Dit kan door de elementen op te slaan in gelinkte lijsten (Hoofstuk 6)! In die gelinkte lijsten plaats je dan alle objecten waarvan de sleutel naar eenzelfde index leidt.



Door in elke link expliciet de sleutel te vermelden kan je in de gelinkte lijst zoeken naar het gewenste record. Maar dergelijke gelinkte lijsten met expliciete pointers zijn niet bevorderlijk voor de snelheid omdat elke toegang tot gegevens voorafgegaan wordt door minstens één toegang tot pointers. Om die reden wordt er gekozen voor een tweede oplossing waarin geen pointers nodig zijn.

Een tweede oplossing is het berekenen van een tweede hashwaarde met de hoop dat dit vakje nog vrij is. Meestal werkt men met *lineaire* of *kwadratische open adressering*. Hierbij worden de gegevensrecords rechtstreeks in de geïndexeerde array geplaatst, maar, omdat er zich een botsing voordoet, wordt een alternatieve index berekend. De indices die opeenvolgend geprobeerd worden, zijn bij lineaire open adressering:

$h_i = (hash(sleutel) + i)\ MOD\ n$

en bij kwadratische open adressering:

$h_i = (hash(sleutel) + i^2)\ MOD\ n$

waarbij h_0 de initiële index is, *hash(sleutel)* de normale hashfunctie en h_i de index die geprobeerd wordt na de *i*-de botsing. Ter illustratie toont de volgende tabel wat er gebeurt in een initieel lege hashing tabel met 1001 plaatsen waarin opeenvolgend de sleutels 4215, 1212, 7220 en 8219 worden opgenomen. De gebruikte indices worden zowel voor lineaire als voor kwadratische open adressering getoond. Geschrapte indexwaardes komen overeen met de al bezette plaatsen.

Key	h0 = Hash(Key)	$h_1 = h_0 + 1$	$h_2 = h_0 + 2$	$h_3 = h_0 + 3$ (lineair)	$h_2 = h_0 + 2^2$ (kwadratisch)
4215	211				
1212	~~211~~	212			
7220	213				
8219	~~211~~	~~212~~	~~213~~	214	215

Bij lineaire open adressering hebben de bezette plaatsen de neiging om zich in blokken te concentreren. Dit heeft tot gevolg dat het gemiddeld aantal pogingen per zoekoperatie vaak groter is dan bij een kwadratische open adressering. Anderzijds, als de gegevens opgeslagen zijn op een schijfgeheugen is het qua toegangstijd juist voordelig wanneer er opeenvolgende zoekoperaties dichtbij elkaar plaatsvinden. Daarom kan lineaire open adressering aantrekkelijk zijn op schijf terwijl het kwadratische beter is in het centrale geheugen.

9.3. Voor- en nadelen van hashing

Het voornaamste voordeel van hashing is zonder twijfel de snelheid. Het vinden van een element is onafhankelijk van de grootte van de lijst!

Maar hashing heeft ook enkele belangrijke nadelen. Het belangrijkste nadeel is het statische karakter van de datastructuur. De omvang van de tabel waarin de gegevens worden opgeslagen, moet op voorhand bekend zijn en kan onmogelijk worden uitgebreid zonder alle adressen opnieuw te berekenen!
Een tweede nadeel is het feit dat de gegevens in een willekeurige orde opgeslagen zijn. Om een bruikbare lijst van de opgeslagen gegevens uit te printen, moeten alle gegevens worden geordend.

Een laatste nadeel is de enorme moeilijkheid waarmee gegevens uit een hashtabel kunnen worden verwijderd. Dit obstakel wordt eenvoudig omzeild door de objecten die moeten worden verwijderd te markeren met een booleaanse vlag.

In de *Object*-klasse van Java vinden we de volgende functie terug:

```
public int hashCode();
```

Deze functie geeft de hashcode van het object terug. En omdat *Object* de oermoeder is van alle klasses – alle klasses erven automatisch van deze klasse - krijgt elk object standaard een hashcode toegewezen. Handig, je kan dus steeds een hashtabel gebruiken. Niets garandeert echter dat het een goede hashcode is. Soms is het beter om zelf een code op te stellen. Je *overschrijft* dan bovenstaande methode. In de **package *datastructuren*** vind je de **klasse *Hashing*** die voorbeelden toont van de default Java hashcode voor verschillende gegevens.

9.4. Mapimplementaties

We hebben de definitie van een Map al geïntroduceerd. De Map-interface definieert hoe je met een Map werkt, maar voorziet geen implementatie. Het uitwerken van de achterliggende datastructuur en de operaties kan op verschillende manieren. Hiervan bespreken we er twee. Dit is een belangrijke les: enerzijds definieer je de *interface* met het object; de dingen die je met het object moet kunnen doen. Anderzijds voorzie je een *implementatie*. Zo'n implementatie kan op verschillende manieren gebeuren. Maar als gebruiker trek je je daar niets van aan omdat je enkel via de interface met het object communiceert.
Omdat je via de *key* de *value* opzoekt, moet het vinden van een *key* snel gebeuren. Dit hebben we in de vorige hoofdstukken besproken. Twee flexibele en snelle manieren zijn de binaire boom en de hashtabel. Beide geven een oplossing. Deze zijn dan ook allebei in Java voorzien: *TreeMap* en *HashMap*. De eerste slaat de keys geordend op in een binaire boom terwijl de tweede een hashtabel gebruikt. Voor dit laatste gebruikt java de *hashCode()*-methode als hashfunctie (zie einde 9.1).

Definiëren doen we zo:

```
Map<String, String> map = new TreeMap<String, String>();
```

Of

```
Map<String, String> map = new HashMap<String, String>();
```

We declareren de variabele als zijnde een Map:

```
Map<String, String> map;
```

maar we maken een TreeMap- of een HashMap-object aan:

```
map = new TreeMap<String, String>();
```

of

```
map = new HashMap<String, String>();
```

We moeten kiezen voor een implementatie, maar in het gebruik van het object werken we met de methodes gedefinieerd door de Map.
De Map definieert als het ware het 'servicecontract'. Ze bepaalt de 'services' die een map-object aanbiedt. In de documentatie staat eenduidig wat het object zal doen. Elke implementatie zal beloven hieraan te voldoen. Als gebruiker kun je nu op beide oren slapen.