Practicum Algoritmiek 2

Week 1: n over k

Deze week ga je een wiskundige expressie onderzoeken die veel gebruikt wordt. De wiskundige expressie geeft aan op hoeveel manieren je *k* verschillende elementen uit een verzameling van *n* elementen kunt halen. Men noemt deze expressie '*n* over *k*' en de schrijfwijze is als volgt:

$$\binom{n}{k}$$

Er gelden een paar regels die je zelf eenvoudig kunt nagaan:

1. $als\, k > n\, geldt: \binom{n}{k} = 0$. Je kunt immers niet meer elementen pakken dan er zijn.
2. $\binom{n}{n} = 1$ Je kunt op één manier alle elementen uit een verzameling halen: door ze allemaal eruit te halen.
3. $\binom{n}{k} = \binom{n}{n-k}$ want als je nu *k* elementen pakt dan laat je *n-k* element liggen. Bij de linkse uitdrukking kijk je naar de elementen die je gepakt hebt en bij de rechtse uitdrukking naar de elementen die je hebt laten liggen.

$\binom{n}{k}$ kun je op allerlei manieren berekenen. Een manier is sneller dan een andere. Je gaat bepalen welke methode gemakkelijk te programmeren is en welke weinig tijd kost.

Het bepalen van de snelheid van een algoritme noemt men in goed Nederlands: 'benchmarking'. Het is in de regel nodig om het algoritme een aantal keer uit te voeren (bijvoorbeeld duizend of een miljoen keer). Geef tijdens het benchmarken aan hoe vaak je een algoritme uitgevoerd hebt en hoe lang de meting duurde. Voer de meting een aantal keer uit, voor een hogere nauwkeurigheid. Geef de kortste tijd weer van de verschillende metingen.

De volgende methode kun je gebruiken voor het bepalen van een tijdsduur:

Java	System.currentTimeMillis()
MFC	clock();
C#	System.Diagnostics.Stopwatch

Er geldt voor de methode $\binom{n}{k}$:

1. Als $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$ dan is $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
2. $\binom{n}{k} = \frac{n}{1} \times \frac{n-1}{2} \times \frac{n-2}{3} \ldots \times \frac{n-k+1}{k}$
3. $\binom{n}{k} = \binom{n-1}{k-1} \times n \,/\, k$
4. $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Opdrachten:

a. Maak vier methodes voor elke van deze vier formules één. Lever de broncode van methodes in.
b. Bepaal de rekentijd voor de volgende waarden door middel van benchmarking:

   1. $\binom{6}{3}$
   2. $\binom{10}{5}$
   3. $\binom{15}{8}$

   Lever de tijden in in een tabel. Voer alle 4 de methodes uit opdracht A uit voor alle 3 de waarden. Let wel: een rekentijd van 0 is een foute waarde!!!

c. Bij de volgende waarden kun je problemen ondervinden:

   Geef aan wat er mis gaat en ga na waarom het misgaat:

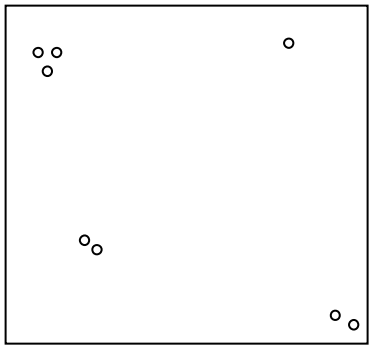
   1. $\binom{1000}{1}$ Kijk naar de uitkomsten en verklaar eventuele afwijkingen.
   2. $\binom{80}{38}$ Hou hier rekening met de mogelijkheid dat de berekening niet eindigt! Als de berekening niet eindigt, ga dan na waarom. Als de berekening onverwachte uitkomsten geeft, ga dan ook na hoe dat komt.

   Lever de uitkomsten en de verklaringen daarvoor in.

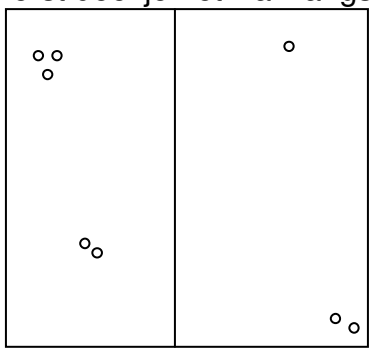
Week 2 tot en met week 5: BSP–boom

In de aankomende drie weken ga je een belangrijk onderdeel van een computerspel maken. Je gaat een container maken waarin objecten die in het spel dicht bij elkaar zitten ook in de container zoveel mogelijk dicht bij elkaar zetten. Zo'n container heet een BSP-boom waarbij BSP staat voor 'binary space partitioning'. We gaan een variatie op zo'n BSP–boom maken.

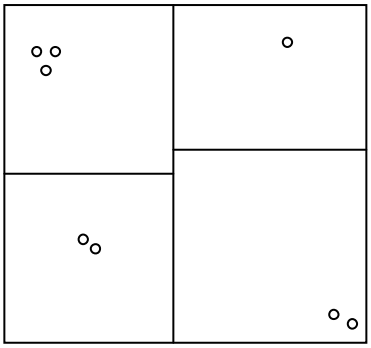
Om de opbouw van een BSP–boom uit te leggen begin ik met een vlak met spelobjecten (cirkels). Het vlak kan je je voorstellen als de spelwereld en de spelobjecten zijn de karakters die er in rondlopen:



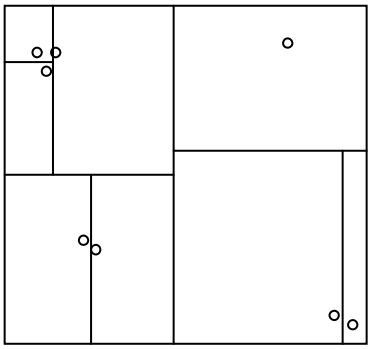
De werkwijze is nu als volgt: Eerst deel je het vlak langs de verticale as in tweeën:



Vervolgens deel je iedere helft ook weer in 2 stukken op de horizontale as zodat je nu 4 stukken hebt:



Je gaat net zolang door totdat elk object in een apart hokje zit:



Je ziet dat bij de onderverdeling de grenzen steeds zo gekozen worden dat het onderscheid optimaal is: in elk subvlak wordt een waarde gekozen die onderscheid mogelijk maakt tussen de verschillende objecten.

Door middel van de opdrachten die hieronder staan zal dit idee in de komende 3 weken steeds verder worden uitgewerkt.

Week 2: BSP–boom en quicksorttechnologie

Het startpunt van het maken van een BSP–boom is een array met de spelobjecten. Een spelobject kan er voor deze opdracht als volgt uitzien (in Java):

```
public class SpelObject
{
   public static final int  DIMENSION = 2;
   private double[]         position  = new double[DIMENSION];

   // nog in te vullen

   public double getPosition( int index )
   {
      return position[index];
   }
}
```

```
Hierbij staat DIMENSION voor het aantal dimensies dat het
spelobject heeft. In ons geval zijn dit er dus 2, namelijk het
X en het Y coördinaat van het spelobject.
De index in bovengenoemd spelObject geeft aan over welke
dimensie het gaat. 0 = X, 1 = Y.
```

Werkwijze

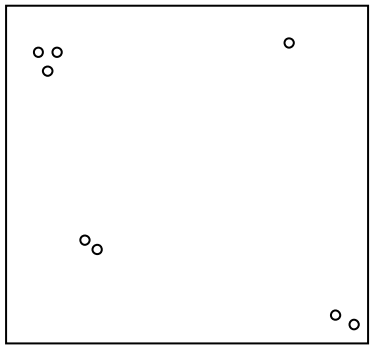
Je mag bij deze opdracht alleen gebruik maken van technieken die je kent van quicksort.

1. herschik de array voor index 0:
   a. Bepaal een waarde *a* die redelijkerwijs de waarden verdeelt in twee ongeveer gelijke helften. Kies daarom een ‘pivot’ die volgens de quicksorttechnologie de gegevens redelijkerwijs in tweeën deelt.
   b. Verdeel de array in een linkerhelft die kleiner of gelijk is aan de waarde van *a* door middel van de in opdracht A. gekozen waarde. en een rechterhelft waarbij de waarden groter of gelijk zijn aan die waarde.

   Je krijgt nu twee helften.
2. Voer nu voor iedere helft het algoritme van punt 1. uit. De index is echter nu 1. Je krijgt nu voor elke helft weer twee helften.
3. Voer nu voor deze helften het algoritme van punt 1 uit. De index is nu weer 0 (je verhoogt de index steeds met 1 totdat de index groter of gelijk is aan DIMENSION. Dan wordt de index weer 0)
4. Ga nu weer naar punt 2.
5. ga net zolang door totdat een helft nog maar uit één element bestaat.

Hint: maak een onderverdeling voor als je nog 1, 2 of meer elementen hebt die je sorteert.

Voorbeeld
We starten met



De punten zijn `(40, 800) (60, 800) (700, 850) (50, 750) (100, 100) (110, 90) (900,100) (950, 50)`

De elementen worden willekeurig in een array geplaatst
`{ (900,100), (100,100), (950,50), (50,750), (110,90), (60,800), (40,800), (700,850) }`

`Onderstaand is het voorbeeld uitgewerkt zoals beschreven in bovenstaand stappenplan. Iedere stap gaat een laag dieper en zal dus een volgend punt zijn in de werkwijze zoals hierboven beschreven.`

stap 1:

Deze array wordt in tweeën verdeeld op basis van de *x*-waarde:
links: `{ (50,750), (100,100), (60,800), (40,800), (110,90) }`
rechts:`{ (700,850), (950,50), (900,100) }`

stap 2:

De linkse helft van stap 1 (*{ (50,750), (100,100), (60,800), (40,800), (110,90) }*) wordt in tweeën verdeeld op basis van de *y*-waarde:
links: *{ (110, 90), (100, 100) }*
rechts: *{ (50, 750), (40,800), (60, 800) }*

De rechtse helft van stap 1 (*{ (950,50), (900,100), (700,850) }*) wordt ook in tweeën verdeeld op basis van de *y*-waarde:
links: *{ (950,50) }* → één element, dus klaar
rechts: *{ (900,100) , (700,850) }*

stap 3:

We hebben nog drie stukken om naar te kijken:
1: *{ (110, 90), (100, 100) }*
2: *{ (50, 750), (60,800), (40, 800) }*
3: *{ (700,850), (900,100) }*

Ze worden allemaal onderverdeeld op basis van de *x*-waarde:

linkerhelft	rechterhelft
{ (100,100) } → één element, dus klaar	*{ (110,90) }* → één element, dus klaar
{ (40,800) } → één element, dus klaar	*{ (40,800), (60,800) }*
{ (700,850) } → één element, dus klaar	*{ (900, 100) }* → één element, dus klaar

stap 4:

We hebben nog maar één stuk om naar te kijken:
{ (40,800), (60,800) }
Die wordt als volgt onderverdeeld (op basis van de *y*-waarde):
links: *{ (40,800) }*
rechts: *{ (60,800) }*

Opdracht:

Lever de broncode in die bovenstaand algoritme implementeert. Denk er aan: gebruik quicksort–technologie.

Week 3: BSP–boom maken

Deze week ga je een boomstructuur maken. Je gaat hierbij de code van week 2 uitbreiden. Elk knooppunt van de boomstructuur is afgeleid van een basisklasse Node. Je hebt hierbij twee afgeleide klassen EndNode waarin een SpelObject is opgeslagen en een SplitNode die een verwijzing heeft naar een linker- en een rechterdeel. Een SplitNode zelf bevat dus geen spelobject met coördinaten.
Zoals je in het voorbeeld van week 2 kon zien, kun je de handelingen onderverdelen in twee soorten:

1. een helft bestaat uit slechts één element. In dat geval plaats je dat ene element in een instantie van een EndNode.
2. een helft bestaat uit meerdere elementen. In dat geval krijg je een linkerdeel en een rechterdeel. Zo'n linkerdeel en rechterdeel wordt in een instantie van een SplitNode geplaatst.

De code van de klassen Node, EndNode en SplitNode kan als volgt zijn (in Java):

```
public class Node
{
     private Node   ouder;

     public Node( Node ouder )
     {
          this.ouder     = ouder;
     }
}

public class EndNode extends Node
{
     private SpelObject  spelObject;

     public EndNode( Node ouder, SpelObject spelObject )
     {
          super( ouder );

          this.spelObject     = spelObject;
     }
}

public class SplitNode extends Node
{
     private Node   linkerKind,
                    rechterKind;

}
```

De klassen zoals ze hier beschreven zijn, zijn natuurlijk nog niet af. Je dient in ieder geval nog wat getters en/of setters toe te voegen.

Opdracht

De opdracht van deze week:

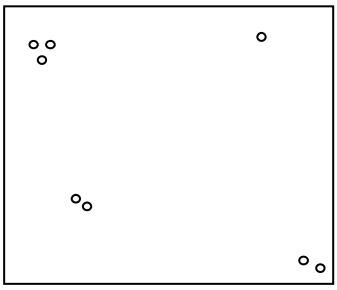
1. maak bovenstaande klassen verder af
2. breid het algoritme van week 2 dusdanig uit dat de beschreven of vergelijkbare boomstructuur ontstaat door middel van recursie. Breid hierbij de Sort methode van week 2 uit, zodat deze tijdens het sorteren een BSP boom opsteld en de root hiervan teruggeeft.

Lever werkende code in.

Voorbeeld

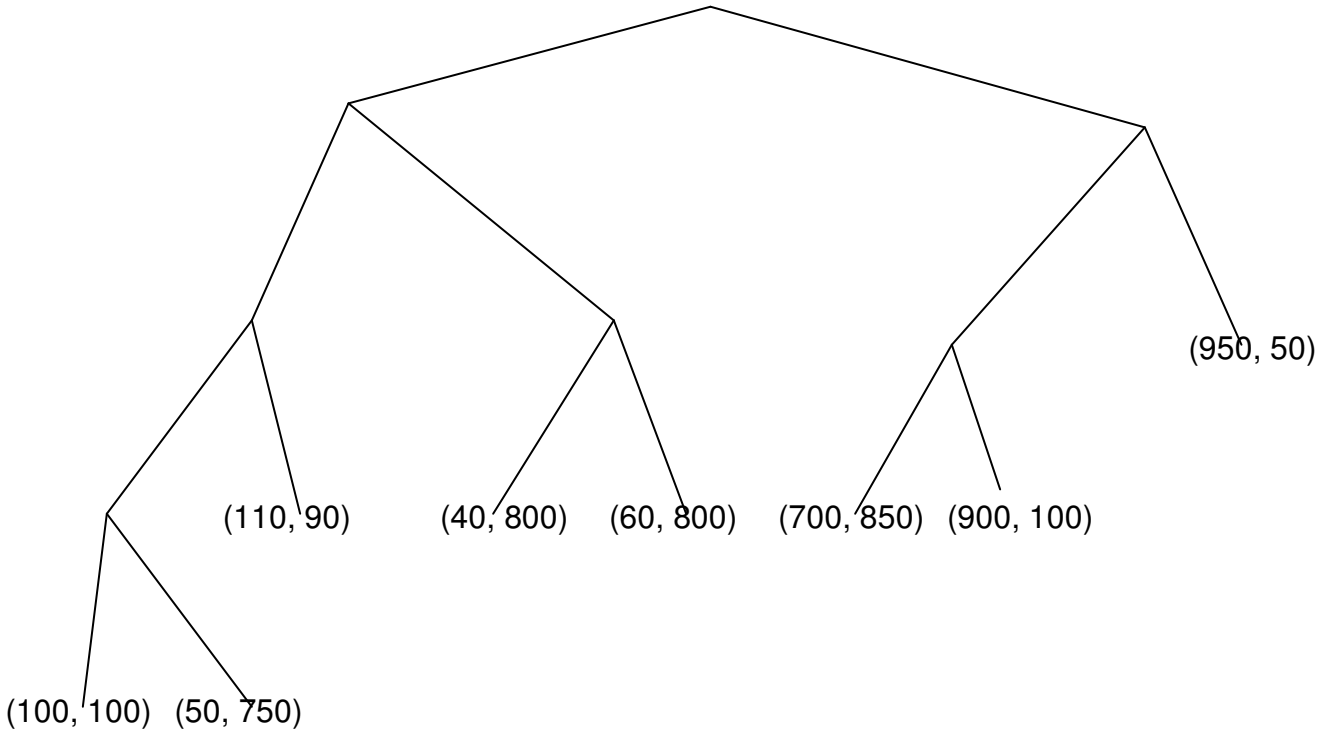
Het resultaat is dat je een boom krijgt waar alleen in de bladeren (EndNodes) spelobjecten zitten. In alle knooppunten (SplitNodes) zitten dus alleen referenties naar de volgende nodes.

In week twee hadden we het volgende voorbeeld:



```
( 40,800) ( 60,800) (700,850) ( 50,750)
(100,100) (110, 90) (900,100) (950, 50)
```

Als we het voorbeeld van week twee uitwerken kan de boom de volgende vorm krijgen:



Op de punten waar coördinaten staan, heb je een EndNode in de boom. Als er een splitsing getekend is, heb je te maken met een SplitNode.

(Bovenstaand voorbeeld van een BSP –boom dient slechts als illustratie; het is niet noodzakelijkerwijs een realistische structuur.)

Week 4: Eigenschappen en doorloop van BSP–boom

Opdracht 1: eigenschappen toevoegen aan de BSP–boom

We beginnen deze week met het bepalen wat de minimale *x*-waarde is van alle spelobjecten in een subboom. Ook dien je de maximale *x*-waarde van alle spelobjecten in een subboom te bepalen. Dat dien je niet alleen voor de *x*-waarden te doen maar ook voor de *y*-waarden. Als ieder spelobject *i* een driedimensionale ruimte beslaat dien je ook nog de minimale en maximale *z*-waarde in een bepaalde subboom te bepalen.

Werkwijze

De werkwijze is als volgt:
Definieer in de klasse Node twee abstracte methodes:

```
public abstract class Node
{
     private Node   ouder;

     public Node( Node ouder )
     {
          this.ouder     = ouder;
     }

     public abstract double lowerBound( int index );
     public abstract double upperBound( int index );
}
```

De abstracte methodes moeten natuurlijk ingevuld worden in de klassen `EndNode` en `SplitNode`:

- Voor de klasse `EndNode` is het niet zo moeilijk: het object `spelObject` van die klasse bepaalt de waarde. Je kunt hier de methode `GetPosition` van de klasse `SpelObject` gebruiken om deze waarde te achterhalen.
- Voor de klasse `SplitNode` geldt dat
    - `lowerBound(index)` het minimum is van de `lowerBound` van het `linkerKind` of de `lowerBound` van het `rechterKind`. Het is dus de kleinste van de 2.
    - `upperBound(index)` het maximum is van de `upperBound` van het `linkerKind` of de `upperBound` van het `rechterKind`. Het is dus de grootste van de 2.

  Sla voor de klasse `SplitNode` de minimale en maximale waarden op in de arrays `lowerArray[DIMENSION]` en `upperArray[DIMENSION[`. Deze array's zijn natuurlijk attributen van de klasse `SplitNode` en bevatten de waarden voor de indexen (in ons voorbeeld is dit X en Y).
  Het vullen van deze lowerArray en upperArray doe je met een methode die je aanroept nadat je de lijst hebt gesorteerd. Maak hierbij gebruik van recursie om deze arrays te vullen met de juiste data.

Opdracht 2: de BSP–boom doorlopen

In de tweede opdracht van deze week ga je door de boom lopen. Je gaat na op welke spelobjecten geklikt is met een muis. Invoer van je handeling is de coördinaat van de muisklik.Hiervoor kun je dus ook gewoon een coördinaat in de zoekmethode invullen. Het resultaat is een array of gekoppelde lijst met de spelobjecten waarop geklikt is. Hierbij is er op een spelobject geklikt als deze dezelfde coördinaten heeft als die van de muisklik.

Hieronder vind je een werkwijze die werkt (het is niet noodzakelijkerwijs de beste werkwijze):

1. Je kijkt steeds eerst naar het linkerkind: ligt voor iedere index de positie van de muisklik tussen de `lowerBound` en `upperBound`? Zo ja, ga naar het linkerkind en kijk daar weer naar het linkerkind. Als het linkerkind een `EndNode` –object is, dan heb je een spelobject gevonden waar op geklikt is. Voeg dit element toe aan je lijst. Zo nee, ga dan door met het volgende punt.
2. Kijk naar het rechterkind. Dan moeten hier de waarden van de muisklik tussen de `lowerBound` en `upperBound` liggen. Als het linkerkind een `EndNode` –object is, dan heb je een spelobject gevonden waar op geklikt is.
3. Voldoet het rechterkind niet, ga dan zover terug in de boom tot de plek waar je de laatste keer naar een linkerkind keek. Bekijk vanaf die plek het rechterkind (met andere woorden: ga vanaf daar verder met actiepunt 2).

Als het eerste spelobject gevonden hebt, dien je vanaf daar naar een volgend object te gaan zoeken. De werkwijze is als volgt:

4. Is het object een linkerkind, kijk dan naar het rechterkind. Voldoet het element, voeg het dan toe aan je lijst. Voldoet het niet, ga dan verder met punt drie.

Maak een methode die deze werkwijze implementeert, dit kan je weer het beste doen door middel van recursie.

Opdracht:

Lever de broncode van opdracht 1 en 2 van deze week in.

Week 5: Snelheid van een BSP–boom

Een muisklik op een object kun je ook nog op een andere manier afhandelen dan in week 4: Je kijkt dan naar ieder object dat bestaat of er op geklikt is. Je loopt hierbij door de array van spelobjecten die ad random verdeeld zijn in de array. Omdat je met één muisklik tegelijkertijd naar meerdere objecten kunt wijzen, dien je alle objecten na te lopen.

Deze week is het de bedoeling dat je weer een Benchmark maakt, zoals je al eens eerder hebt gedaan in week 1.

Kijk welke methode het snelst is:
1. zoeken in een bestaande BSP–boom
2. een BSP–boom maken en er vervolgens in zoeken
3. door de array lopen om de objecten te vinden waarop geklikt is

Werk hierbij met een verschillend aantal objecten:
a. 5 spelobjecten
b. 50 spelobjecten
c. 500 spelobjecten

Ga ook na wat er gebeurt als er op een verschillend gelijktijdig aantal objecten geklikt is
I. 0 objecten
II. 1 object
III. 2 objecten
IV. 3 objecten

Opdracht:

Geef de brondcode van bovenstaande benchmark. En geef conclusies over welke methode wanneer het snelste is.

Week 6: een thread–safe hashtable

De opdracht van deze week is:
maak een thread–safe hashtable. Zorg ervoor dat je altijd op een veilige manier de volgende handelingen kunt uitvoeren in meerdere threads:

1. zoeken
2. invoegen
3. verwijderen
4. vervangen

Het spreekt voor zich (…) dat je nooit tegelijkertijd kunt invoegen en verwijderen: Als één thread wil invoegen, worden naar de waarden gekeken. Als die waarden veranderd worden door een andere thread op het moment dat je er naar kijkt, kun je als programmeur niet weten wat er precies gebeurt. Je hebt bijvoorbeeld net een waarde ingevoegd terwijl ergens voor die waarde de vlag op 'verwijderd' wordt gezet. Als de volgorde net andersom is, wordt de in te voegen waarde op de plek van het verwijderde element geplaatst.

De uitdaging in deze opdracht is na te gaan wat wèl en wat niet tegelijkertijd kan worden uitgevoerd. Zorg ervoor dat de ene handeling pas start als de andere handeling pas klaar is. Wees niet te rigoureus: je kunt wèl tegelijkertijd zoeken met meerdere threads.

Maak boven beschreven mechanisme zelf en leun niet op eventuele mechanismen die de taal aanbiedt. Kortom gebruik geen kant en klare dingen zoals een HashTable die al is geïmplementeerd in Java zelf.

In het verleden werden standaardoplossingen van het Internet gehaald. Om dit te voorkomen dien je het thread–safe maken dusdanig te programmeren dat het thread–safe maken door één of meerdere APARTE methodes uitgevoerd wordt. (Het mag duidelijk zijn dat een oplossing van het Internet inleveren een vorm van fraude is: je hebt de oplossing niet zelf bedacht en er zal als zodanig mee omgegaan worden.)

Opdracht:

1. Geef een duidelijke redenatie van wat wel en niet tegelijkertijd kan. Vul hierbij onderstaande tabel verder in. Geef ook aan waarom de al ingevulde voorbeelden WEL of NIET gelijktijdig kunnen.

	Zoeken	Invoegen	Verwijderen
Zoeken	WEL		
Invoegen			NIET
Verwijderen		NIET	

2. Implementeer de thread safe hashtable en lever de broncode hiervan in.