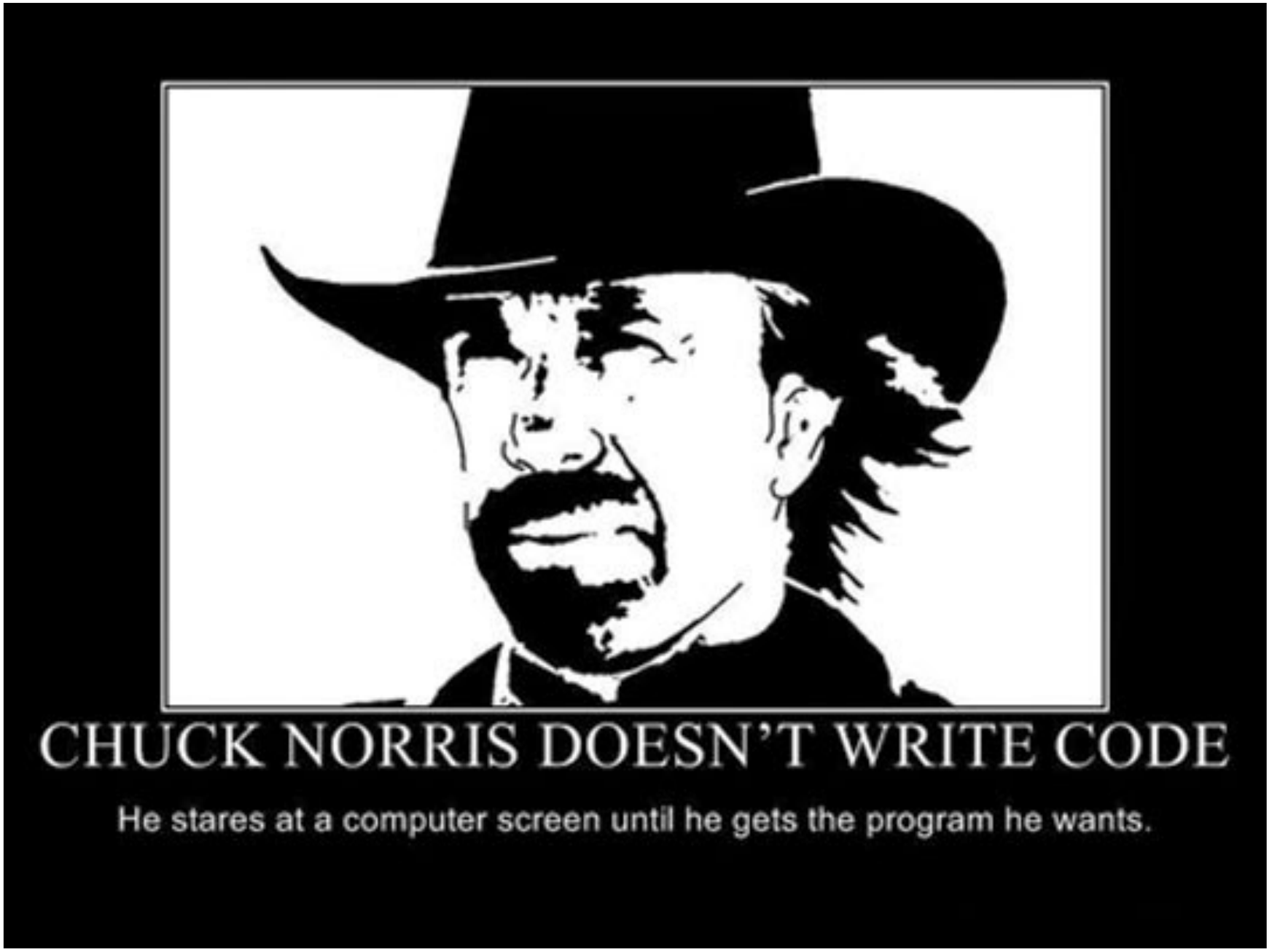
LINFO 1121

DATA STRUCTURES AND ALGORITHMS

UCLouvain

Les tables de hachage (Hash tables)

Pierre Schaus

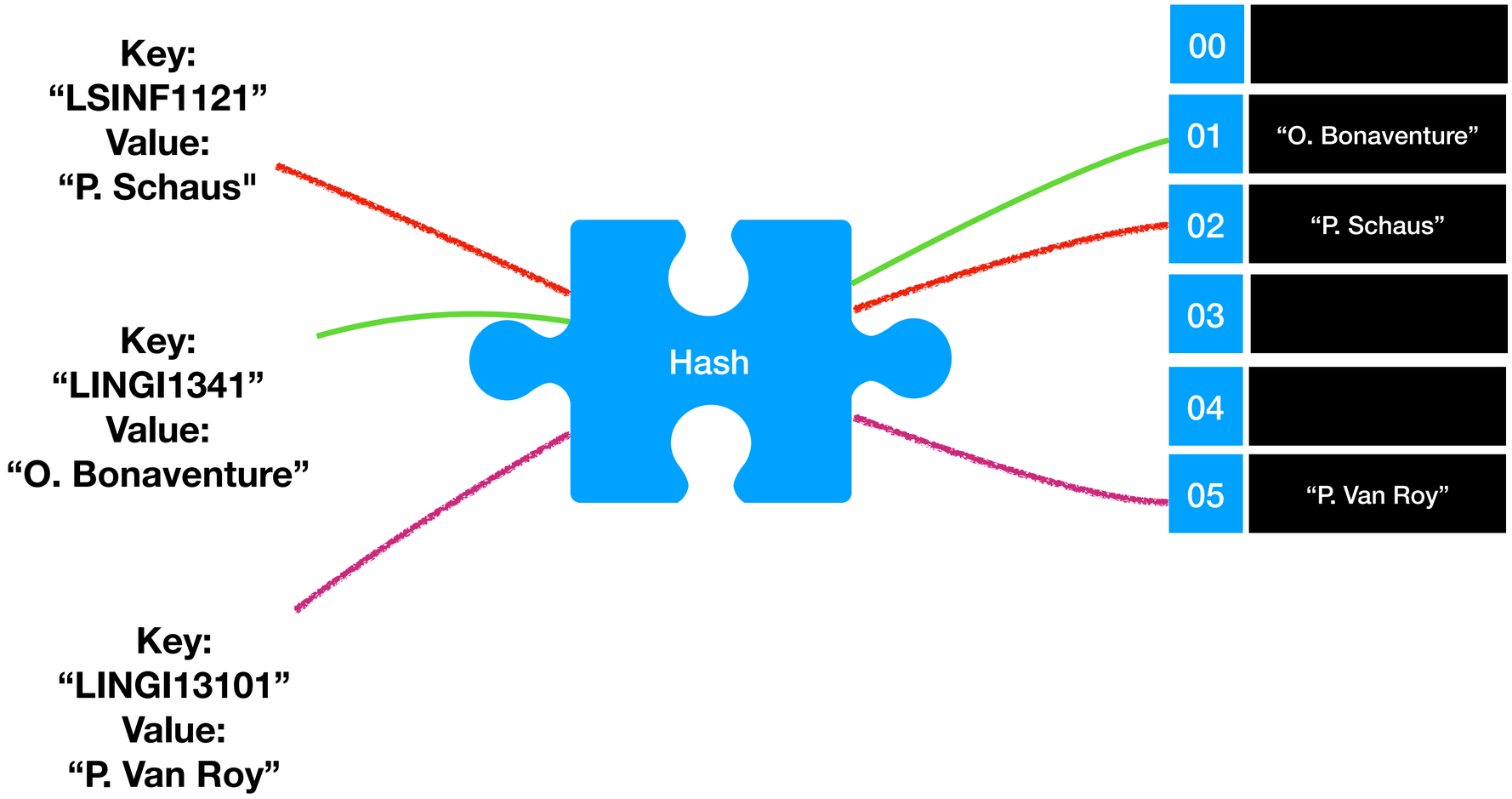

CHUCK NORRIS DOESN'T WRITE CODE
He stares at a computer screen until he gets the program he wants.

But you are not Chuck !

- So don’t forget to practice!

Au menu

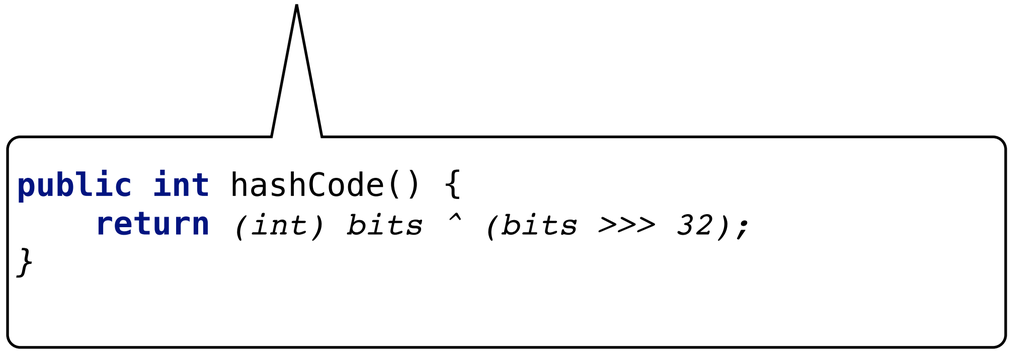
- Nous allons essentiellement parler de la fonction de Hash et ce qui est important pour avoir une fonction de qualité.



Comment faire un hash sur les Long

Voici la formule utilisée par Java pour calculer une fonction de hachage sur les doubles (bits est un tableau de 64 bit représenté sous forme de long):

return (int) bits ^ (bits >>> 32)



```
public int hashCode() {
    return (int) bits ^ (bits >>> 32);
}
```

Pourquoi ne pas simplement utiliser ?

return (int) bits

```
long a = 10101266124699712 94L;

System.out.println(Long.toBinaryString(a));
```

```
0000111000000100101100001100110101011010011100001111100101011110
```

```
System.out.println(Integer.toBinaryString((int) a));
```

```
01011010011100001111100101011110
```

```
System.out.println(Integer.toBinaryString(a >>> 32));
```

```
00000000000000000000000000000000000011100000010010110000110011 01
```

Bitwise operation

```
System.out.println(Integer.toBinaryString(-8));         // 11111111111111111111111111111000
```

- Signed bit shift >>: le bit le plus à gauche pour le padding

```
System.out.println(Integer.toBinaryString(-8 >> 2));    // 11111111111111111111111111111110
```

- Un-signed bit shift >>>: zero est utilisé pour le padding

```
System.out.println(Integer.toBinaryString(-8 >>> 2));   // 00111111111111111111111111111110
```

- Dans l’autre sens, uniquement <<

```
public int hashCode() {
    return (int) bits ^ (bits >>> 32);
}
```

bits

00001110000000100101100001100110101011010011100001111100101011110

bits >>> 32

00000000000000000000000000000000000011100000001001011000011001101

bits ^ (bits >>> 32)

00001110000000100101100001100110101010100011101000100100110010011

Hash sur des doubles

Un double en Java est représenté en 64 bits sous la forme

$$(-1)^s \times m \times 2^{(e-1023)}$$

Le premier bit est le signe, les 11 bits suivants représentent l'exposant sous forme binaire (non signé) et les 52 derniers bits représentent la mantisse sous forme binaire.

Signe	Exposant décalé	Mantisse
(1 bit)	(*e* bits)	(*m* bits)

Est-ce qu'un nombre décimal positif et son opposé obtiennent des fonctions de hachage différentes ?

Rappel Double

$$(-1)^s \times m \times 2^{(e-1023)}$$

Le nombre -6 est représenté comme

$$(-1)^1 \cdot 1.5 \cdot 2^2$$

1-10000000001-1000000000000000000000000000000000000000000000000000.

1025

1+(mantisse = nombre décimal en binaire)
=1+1*1/2+0*1/4+0*1/8+…
=1.5

- Première observation: le hash des doubles est le meme que le hash sur les longs:

```
public static int hashCode(double value) {
    long bits = doubleToLongBits(value);
    return (int)(bits ^ (bits >>> 32));
}
```

- hashCode(6.0)=1075314688,
- hashCode(-6L)=-1072168960.

Le nombre 6 est représenté

0-10000000001-10000000000000000000000000000000000000000000000000.

Le nombre -6 est lui représenté comme

1-10000000001-10000000000000000000000000000000000000000000000000.

C’est grâce à (bits >>> 32), que tous les bits ont bien un impact sur la fonction de hachage, y compris le bit de signe.

Hash int -> double

- Est-ce que la fonction de hachage d'un entier sur 32 bits et celle de ce même entier qui serait casté en double sont les mêmes?

- C'est faux. ((double)5) est représenté comme $(-1)^0 \cdot 1.25 \cdot 2^2$

- 0-10000000001-0100000000000000000000000000000000000000000000000000.

- Ce nombre de 64 bit est sans grand rapport avec

- 0-00000000000-0000000000000000000000000000000000000000000000000101.

- or la formule de hash (sur les long) est la même.

- Le hashcode sera donc différent.

Hash int -> long

- Est-ce que la fonction de hachage d'un entier sur 32 bits et celle de ce même entier qui serait casté en long sont les mêmes?

```
public int hashCode() {
    return (int) bits ^ (bits >>> 32);
}
```

- Observation 1: Le hash d'un entier sur 32 bits est l'entier lui-même.

- Pour le cas, deux cas possibles:
  - Si le int est positif c’est vrai.
    - Si on caste par exemple l'entier 5 en Long on obtient en représentation binaire (61x0)101 (il y a juste 32 zeros mis devant).
    - La formule de hash sur un long est la même sur sur les doubles. (bits >>> 32) va donner un masque de 32x0. Le xor va donc laisser l'entier initial intact.
    - 5 = (61 x 0)101 => hashCode() => 5 car le 0 est neutre pour le xor
  - Si le int est négatif c'est faux. Car l'entier de 32 bits casté en Long aura une représentation différente.
    - -5 = (61 x 1) 011 => hashCode() => 111..011 xor 111..111 => 000…100 = 4

Hash sur les String

- La fonction de hachage pour un string donnée dans le livre p460 est la suivante:

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % M;
```

- Dans l'implémentation du livre, la taille de M (le tableau) est une puissance de deux.

- La valeur suggérée pour R est *un petit nombre premier tel que 31 de sorte que les bits de tous les caractères jouent un rôle.*

Hash sur les String

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % M;
```

M est une puissance de 2

- Supposons que R soit un multiple de M. Que se passerait-il lors du calcul ?

- Supposons par exemple R=kM pour un entier k>0, on a donc comme indice calculé dans le tableau pour le string s:

$$\left(\sum_{i=0}^{n-1}(kM)^{n-i-1}s_i\right)\%\,M=\sum_{i=0}^{n-1}\left((kM)^{n-i-1}s_i\right)\%\,M=s_{n-1}\%\,M\,.$$

- C'est vraiment très triste car on voit bien que seul le dernier caractère est pris en compte pour calculer la fonction de hachage. Il faut donc faire très attention à l'interaction entre M et R.

Hash sur les String

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = (R * hash + s.charAt(i)) % M;
```

M est une puissance de 2

- Supposons que R soit un nombre pair. Que se passerait-il lors du calcul ?

- M s'écrit comme une puissance de deux, disons 2^k. R est pair, on l'écrit par exemple 2l. Notre calcul d'indice s'écrit donc comme suit:

$$\sum_{i=0}^{n-1} \left((2l)^{n-i-1} s_i \right) \% \, 2^k = s_{n-1} \% \, M$$

- Encore une fois, on voit bien que tous les premiers termes vont donner zero. Plus précisément ceux tels que n-i-1 >= k.

- Donc tous les caractères (et donc tous les bits) ne seront pas pris en compte. Pas bien!

- Expliquez pourquoi utiliser 31 est un choix judicieux pour des tailles de tableau qui sont des puissances de deux ?

- Plusieurs atouts pour 31:
  - Il n’est pas un multiple de 2
  - Expérimentalement il produit très peu de collisions: moins que 8 pour plus 50K mots anglais. Il s’écrit aussi 11111

- Serait-ce aussi un bon choix pour une taille de tableau qui commencerait à 31 et qui serait multipliée par deux à chaque fois qu'il faut redimensionner ?

Collision

- Dans l'implémentation du livre la taille de M (le tableau) est une puissance de deux initialisée à 16.

- Supposons qu'à moment donné la taille de M soit 2^8=256.

- Ensuite deux clefs entières sont ajoutées dans une table de hachage implémentée avec separate chaining: respectivement 2560 et 3072 (on suppose que ces ajouts ne causent pas de redimensionnement de la table). Comme vous le savez, le code de hachage d'une clef entière (int) est le nombre lui-même.

- Est-ce que l'ajout de ces deux valeurs va causer une collision entre elles dans la table ? Si oui pourquoi ? Si oui, pouvez-vous proposer une troisième valeur qui va aussi entrer en collision ?

- Si collision il y a, peut-elle disparaître lors du prochain redimensionnement du tableau telle que dans l'implémentation du livre ?

- Que suggérez-vous pour éviter ce problème ? Quelle a la politique d'initialisation de M et de redimensionnement utilisée dans java.util.HashMap ? Est-ce que cela résout le problème sur notre exemple ?

Collision

- M = 2^8=256.
- Ajout de 2560=10*256 et 3072 =12*256
- Collision car en faisant %256 on arrive à 0 pour les deux.
- Au prochain dimensionnement on arrive à 512 donc ça ne change rien.
- Par défaut, le tableau interne dans HashTable est 11. Son redimensionnement garde une taille impaire: (currentSize * 2 + 1).
- Dans ce cas-ci, on n'a pas de problème. L'avantage de la stratégie de Java est qu'une collision peut disparaître au prochain dimensionnement. Alors que pour la stratégie du livre pas nécessairement. En effet, lorsqu'on passe à M=512, les deux collisions sont toujours là.

Hash de véhicules

- Que suggèreriez-vous comme fonction de hachage pour l'identification de véhicules qui sont des strings de nombres et de lettres de la forme: "9X9XX99X9XX999999" où un 9 représente un chiffre et un "X" une lettre de A à Z.

- Est-ce que votre fonction de hachage a la propriété que pour une taille de tableau N hypothétique de $10^{11}.26^6$ il n'y a pas de collision ?

- 11 chiffres (max = 9), et 6 lettres (max = 26)

- X = hash des chiffres, il suffit de le lire comme un nombre.

- Y = hash des lettres $\sum_{i=0}^{5} (26)^{5-i} Y_i \,.$

- Final hash: $X*26^6+Y$

Hash Citoyens

- Répertoire des citoyens belges

- Accéder à chaque citoyen par son numéro de carte d'identité (12 chiffres) = clé unique utilisable comme l'indice dans un tableau.

- A chaque indice correspondrait une référence vers une instance de la classe Citoyen dont les champs constituent les informations que l'on désire mémoriser pour chacun.

- Quelle est la complexité temporelle des opérations suivantes ?
  - rechercher les informations relatives à un citoyen à partir de son numéro de carte
  - ajouter un nouveau citoyen.

- Cette implémentation d'un dictionnaire n'est-elle pas encore meilleure qu'une table de hachage ? Peut-on avoir un problème de collision dans ce cas ? Justifiez.

- 10^12 > 25 milliard, c’est donc plus que MAX_INT +2 147 483 647.
- Impossible de créer un tableau de cette taille.
- Remarque: la mémoire à utiliser est fort importante. Un tableau de int de la taille de MAX_INT prendra 8 Go.
- Si 10^12 était possible, cela demanderait plus de 400 Go.

Question Subsidiaire HashMap

Q 4 (5 points) Une application embarquée doit faire appel de manière intensive à une méthode de calcul de distance asymétrique entre deux positions u et v (les positions sont identifiées par un entier entre 0 et 99.999). Le calcul de la distance nécessite de faire une requête à un web-service (`webServiceDistance`) couteux en temps. La bonne nouvelle est que les paramètres (identifiants des noeuds) donnés à la méthode sont plus ou moins prévisibles. En effet pour un instant donné on considère que seulement 300 identifiants u et v sont susceptibles d'être passés à la méthode. Cet ensemble de 300 identifiants évolue relativement lentement dans le temps : certains disparaissent et d'autres apparaissent régulièrement et de manière imprévisible. En charge du développement, vous avez donc l'idée de mettre en cache les résultats d'une partie des précédentes requêtes au webservice. Afin limiter la mémoire allouée (l'application est embarquée sur un smartphone), le système de cache ne peut enregistrer plus de 300x300=90000 entrées de distance. Idéalement pour plusieurs appels rapprochés avec une même paire u, v passée à en paramètre de `distance`, une seule requête à `webServiceDistance` sera effectuée en moyenne (pour mettre à jour/stocker en cache la distance). Lorsque la distance se trouve en cache, elle devra pouvoir être retournée en temps constant. Nous vous demandons d'imaginer et d'implémenter le système de cache et la méthode `distance` utilisant celui-ci et faisant appel à `webServiceDistance` si l'information ne s'y trouve pas pour le mettre à jour. Hint : Nous vous suggérons d'utiliser/implémenter une sorte de table de Hashage ou la stratégie de de gestion des collisions serait le remplacement de l'entrée.

```java
public class CacheDistance {

    // TODO

    /**
     * Computes the distance based on the cache,
     * and calls webServiceDistance if the distance is not in the cache
     */
    public double distance(int u, int v) {
        // TODO
    }

    /**
     * Computes the costly distance on edge u,v based on the trafic
     */
    private double webServiceDistance(int u, int v) {
        // we assume it is implemented
    }
}
```

Question subsidiaire: HashMap

- Je souhaite implementer une structure de donnée avec l'API suivante:
  - Put(Key,Value)
  - Get(Key)
- Ma structure doit contenir maximum N clefs (taille mémoire bornée).
- Lorsque je fais un Put et que N clefs sont présentes, je dois supprimer l'entrée qui a été consultée (put ou get) il y a le plus longtemps.
- Chaque Put/Get doit se faire en O(1) expected.
- Votre solution ?

Solution

```
public class LRUCache<K,V> {

    private int capacity;
    private HashMap<K, Node> map = new HashMap<>();
    private Node head = null; // the MRU (most recently used)
    private Node tail = null; // the LRU (least recently used)

    public LRUCache(int capacity) {
        this.capacity = capacity;
    }
    public V get(K key) {
        if (!map.containsKey(key)) {return null;}
        // remove the node from the linked list
        Node node = map.get(key);
        remove(node);
        // add the node to the front of the linked list
        addToFront(node);
        return node.value;
    }
    public void put(K key, V value) {
        if (map.containsKey(key)) {
            // update the value of the existing node
            Node node = map.get(key);
            node.value = value;
            // move the node to the front of the linked list
            remove(node);
            addToFront(node);
        } else {
            // create a new node
            Node node = new Node(key, value);
            // add the node to the front of the linked list
            addToFront(node);
            // add the node to the map
            map.put(key, node);
            // if the capacity is reached, remove the least recently used element
            if (map.size() > capacity) { removeLRU(); }
        }
    }
    private void remove(Node node) {
        if (node.prev != null) { node.prev.next = node.next; }
        else { head = node.next; }
        if (node.next != null) { node.next.prev = node.prev; } else { tail = node.prev;}
    }
    private void addToFront(Node node) {
        node.next = head;
        node.prev = null;
        if (head != null) { head.prev = node; }
        head = node;
        if (tail == null) { tail = head; }
    }
    private void removeLRU() {
        map.remove(tail.key);
        remove(tail);
    }
}
```

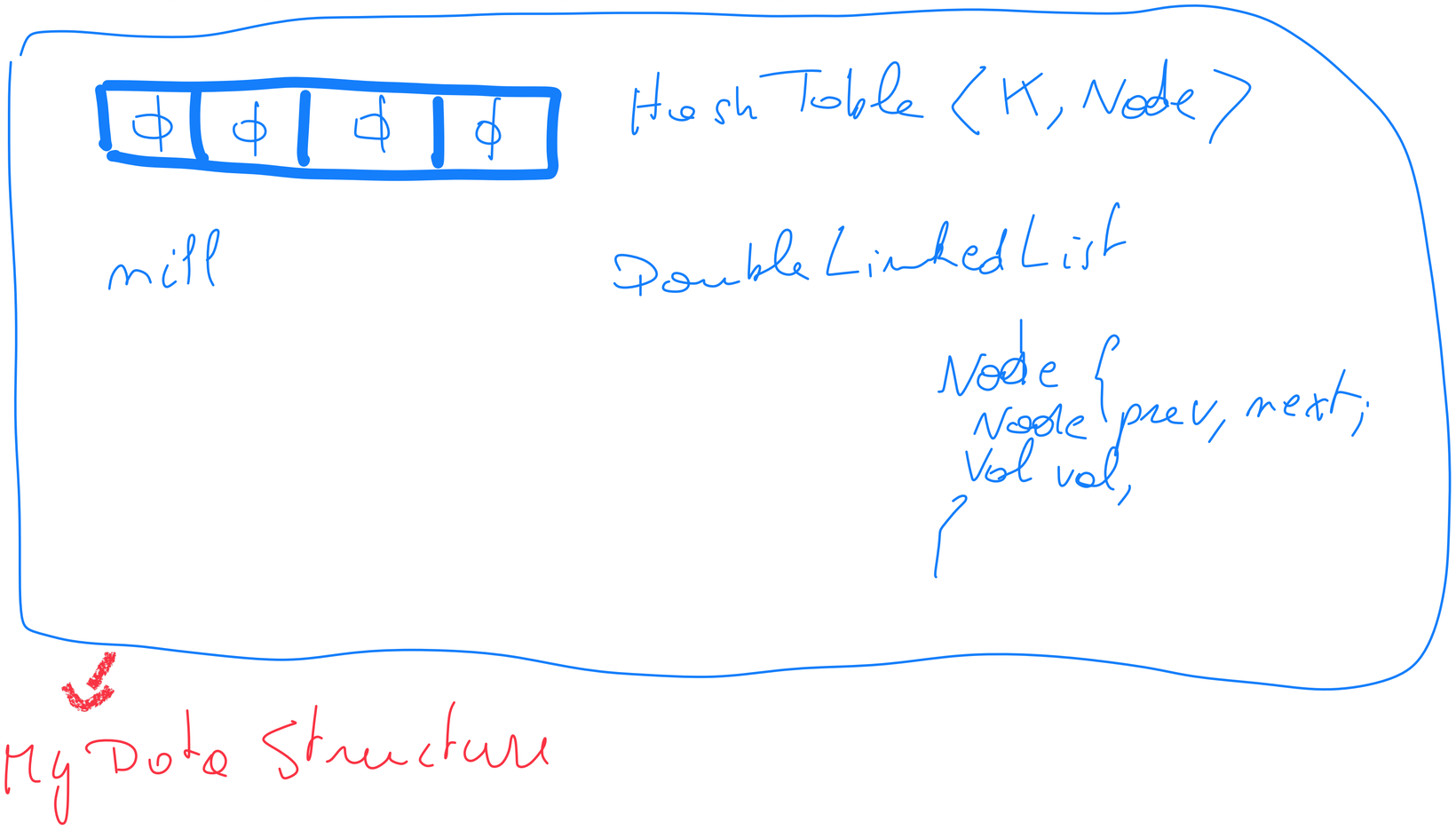
```
private class Node {
    K key;
    V value;
    Node prev;
    Node next;

    public Node(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

HashMap taille bornée

- N = 4

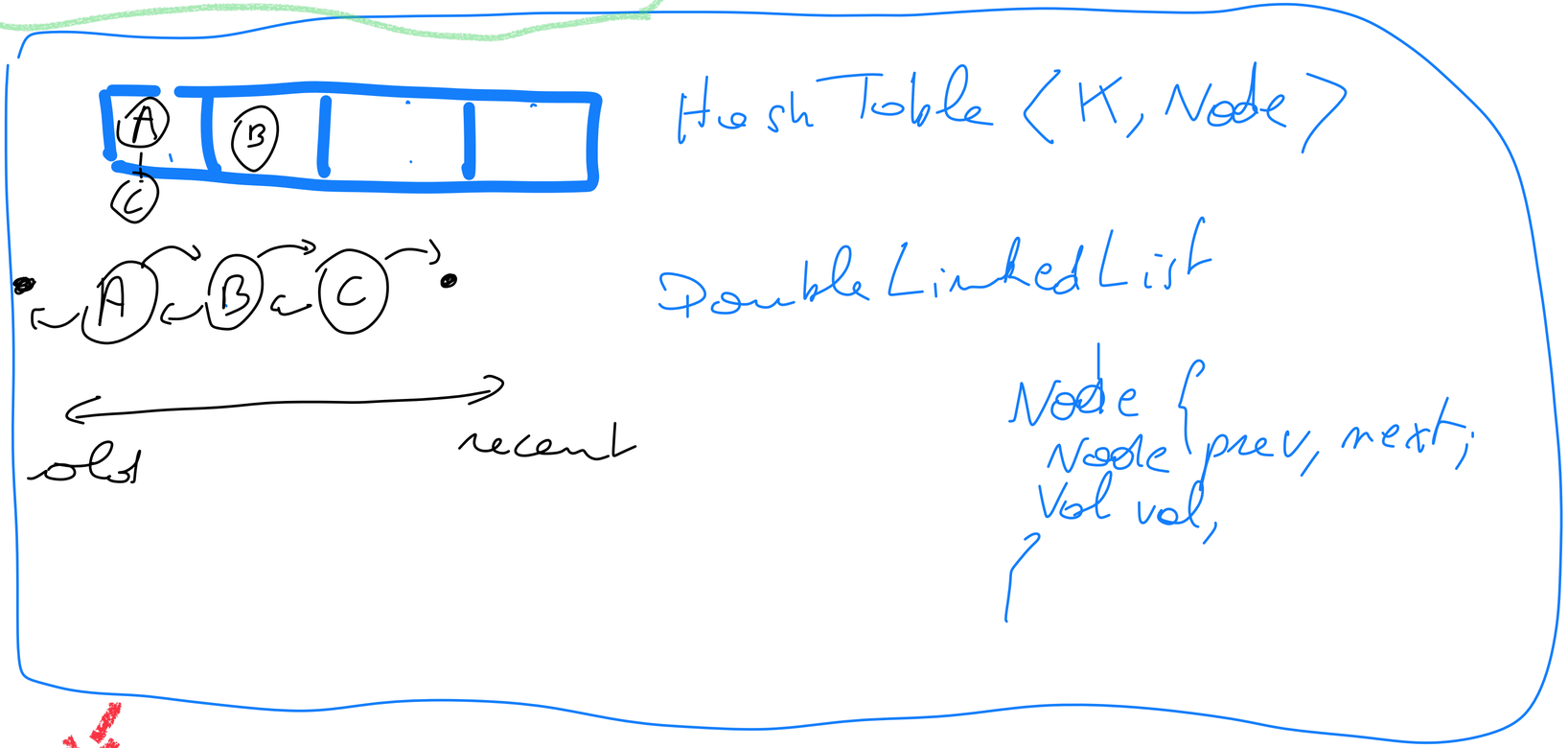
- Put(0,’),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



HashMap taille bornée

- N = 4

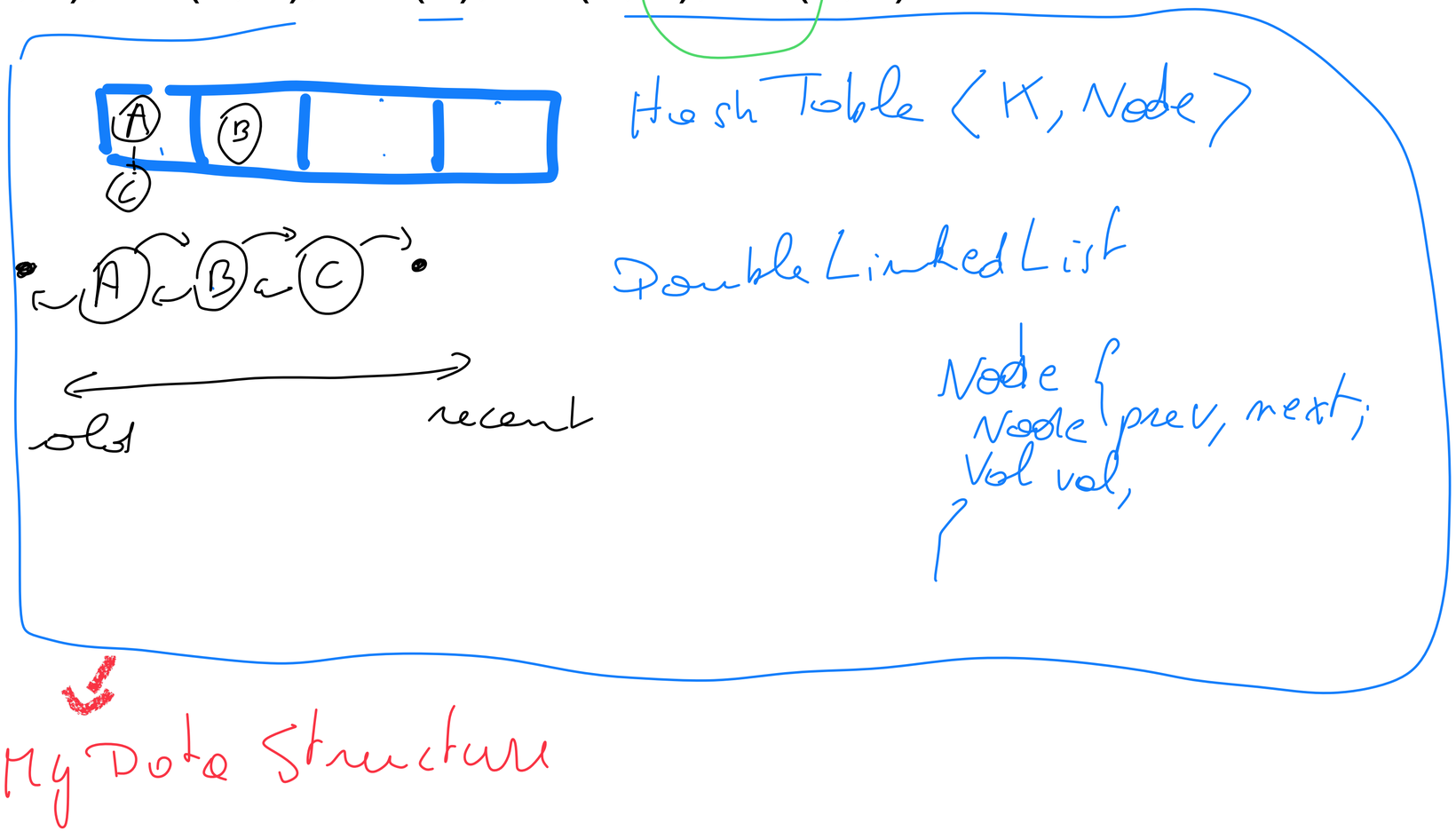
- Put(0,A),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



HashMap taille bornée

- N = 4

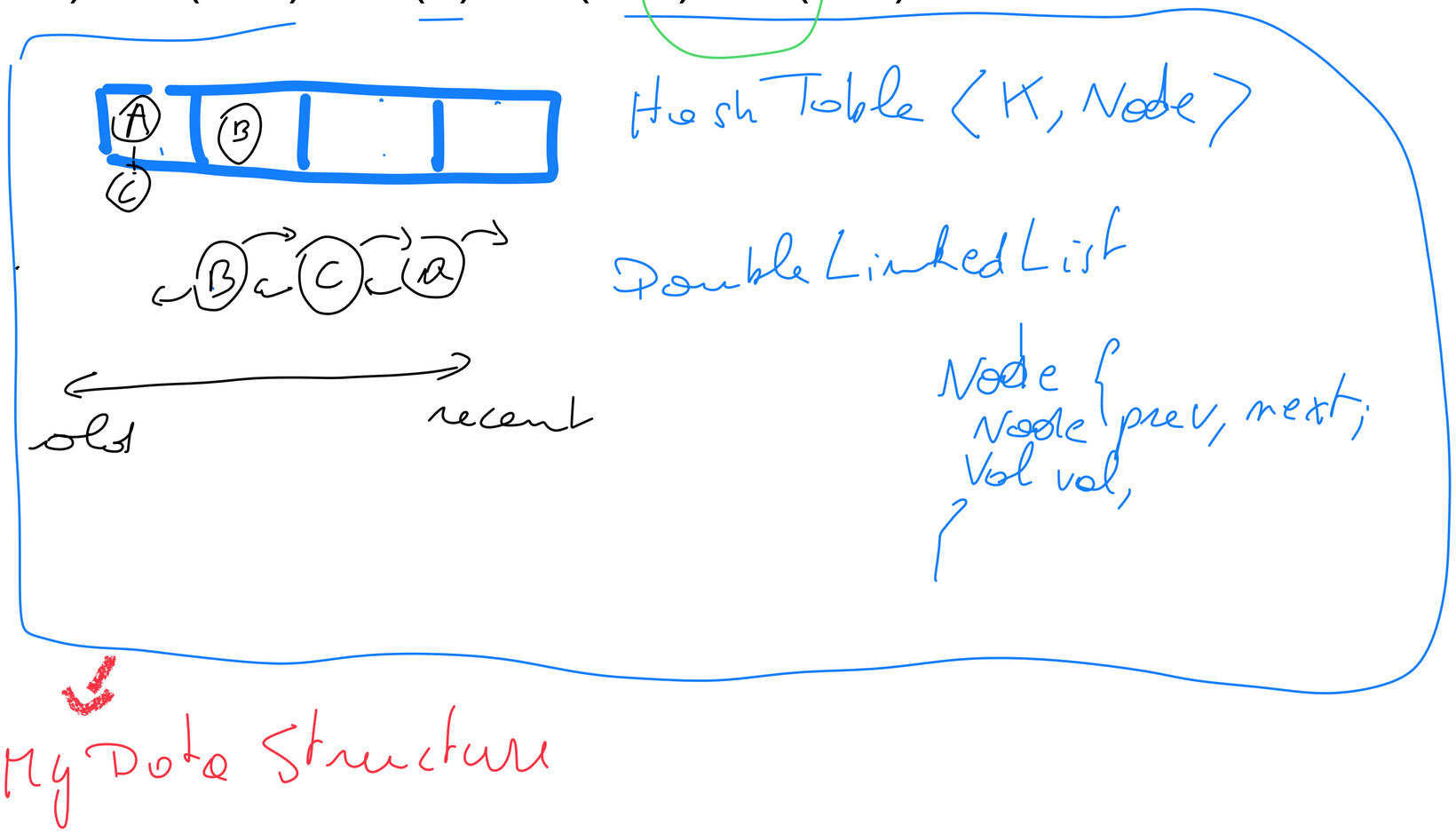
- Put(0,A),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



HashMap taille bornée

- N = 4

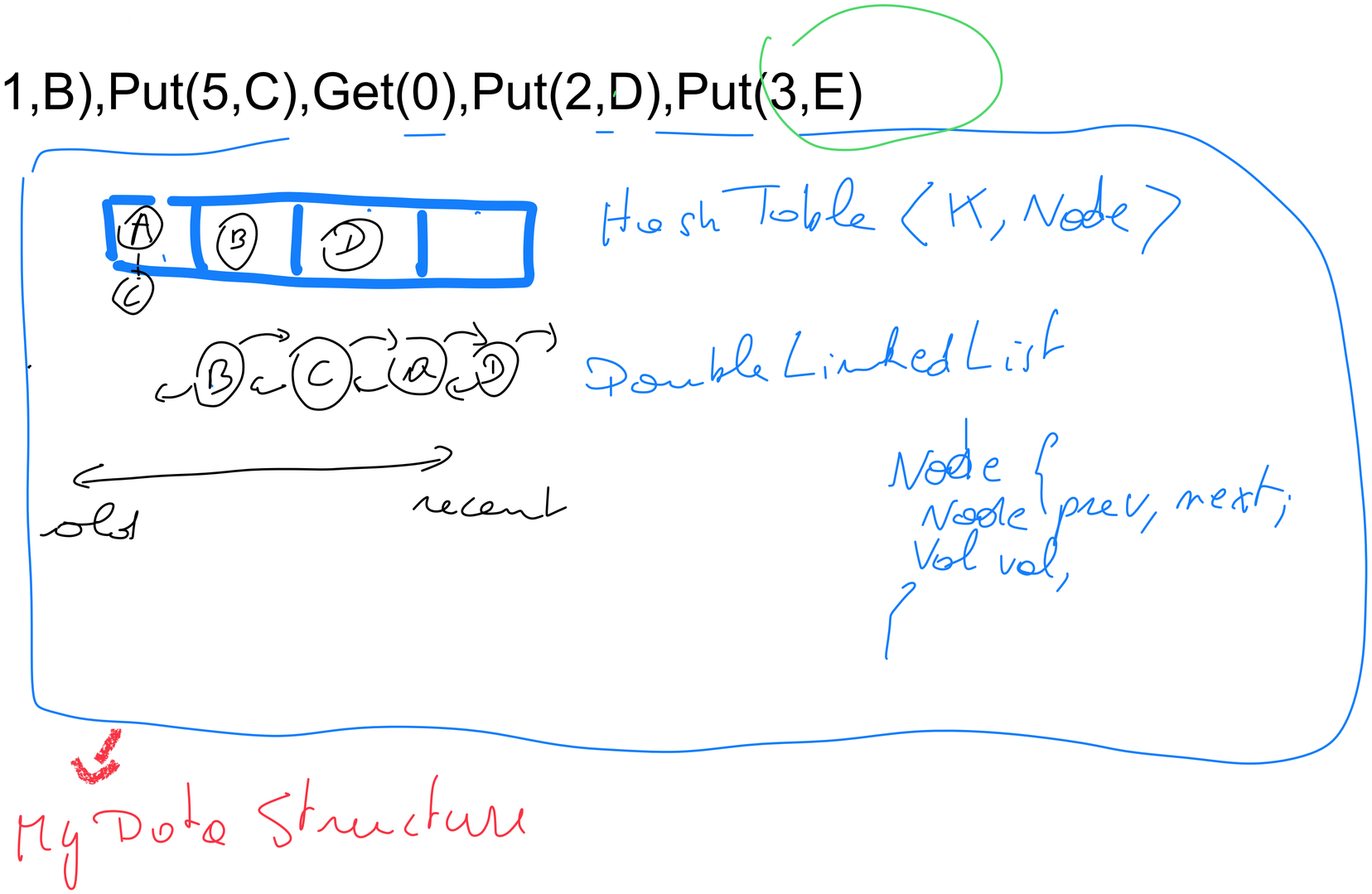
- Put(0,A),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



HashMap taille bornée

- N = 4

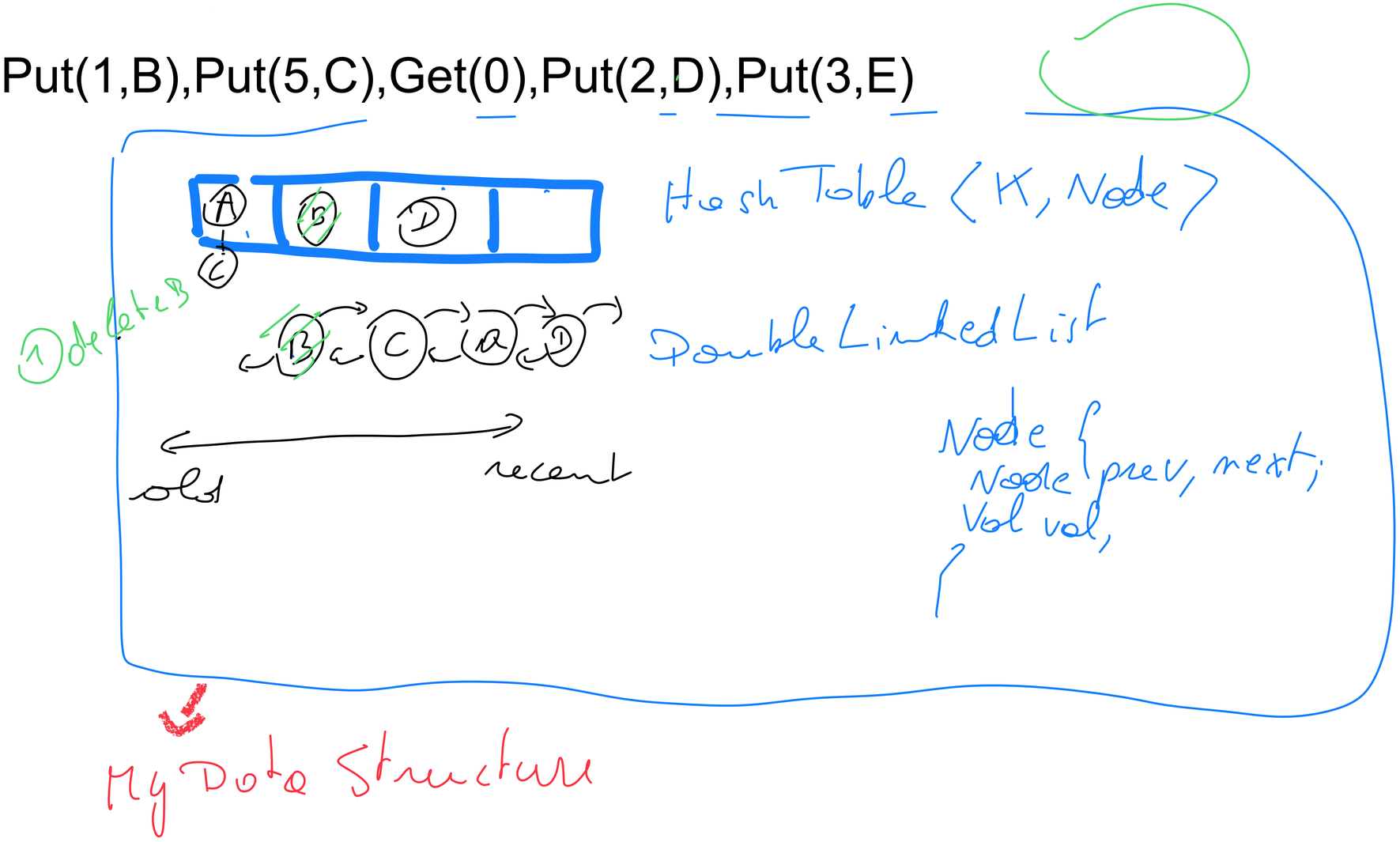
- Put(0,A),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



HashMap taille bornée

- N = 4

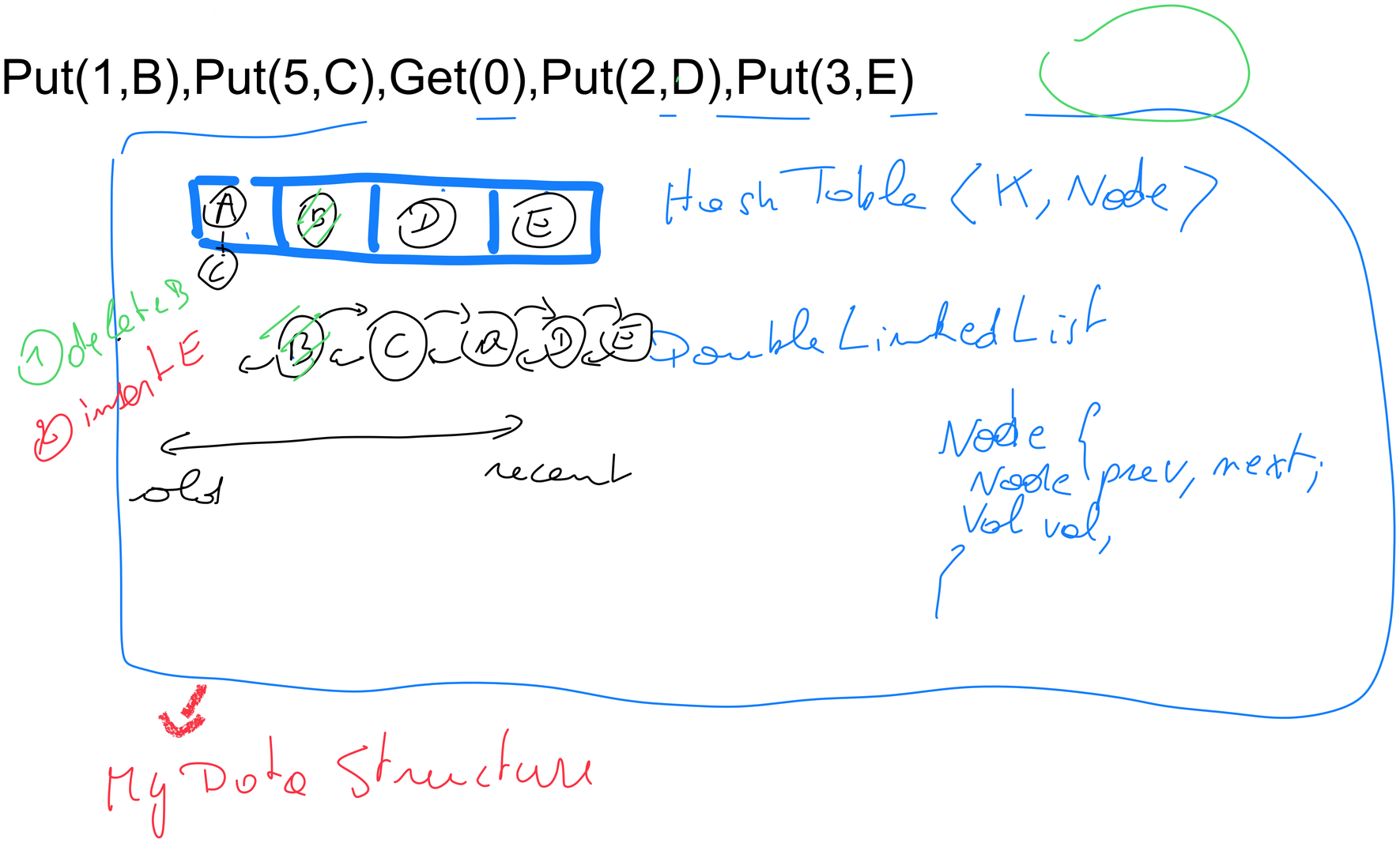
- Put(0,A),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



HashMap taille bornée

- N = 4

- Put(0,A),Put(1,B),Put(5,C),Get(0),Put(2,D),Put(3,E)



LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

Union-Find, Heap, Text Compression

Pierre Schaus

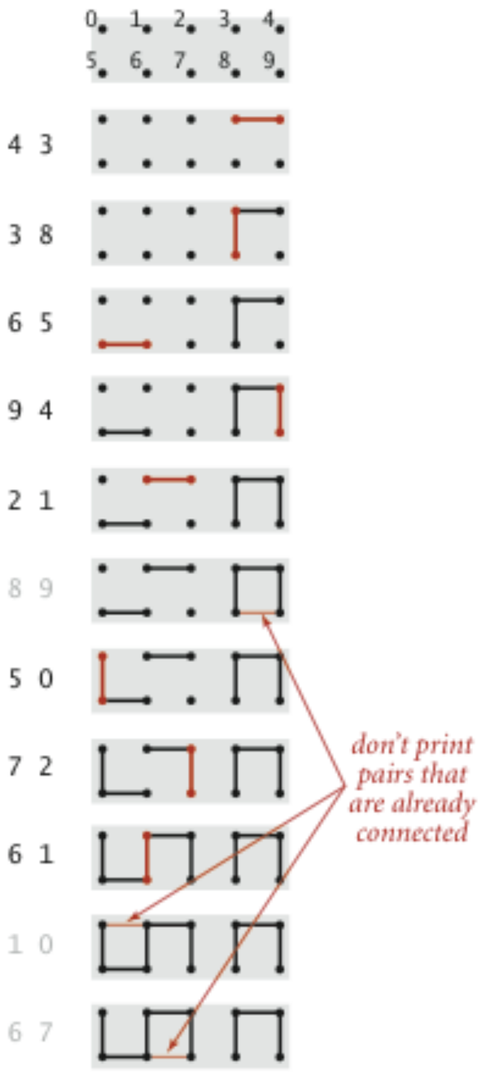
Union-Find

public class UF	
UF(int N)	*initialize N sites with integer names* (0 *to* N-1)
void union(int p, int q)	*add connection between* p *and* q
int find(int p)	*component identifier for* p (0 *to* N-1)
boolean connected(int p, int q)	*return* true *if* p *and* q *are in the same component*
int count()	*number of components*

A quoi ça sert ?

- A compter des groupes

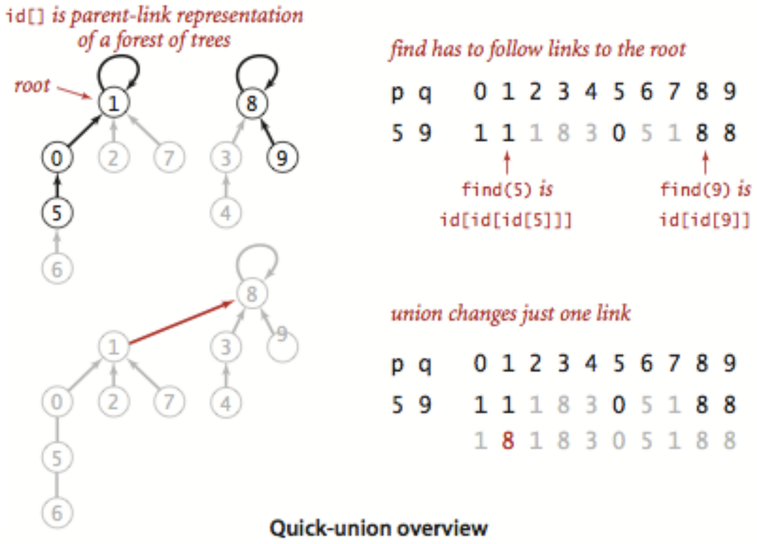
- A vérifier si deux éléments sont du même groupes



Connectivity example

Implémentation

- Basée sur des forêts et structure arborescente implémentée avec un tableau



Quick-union overview

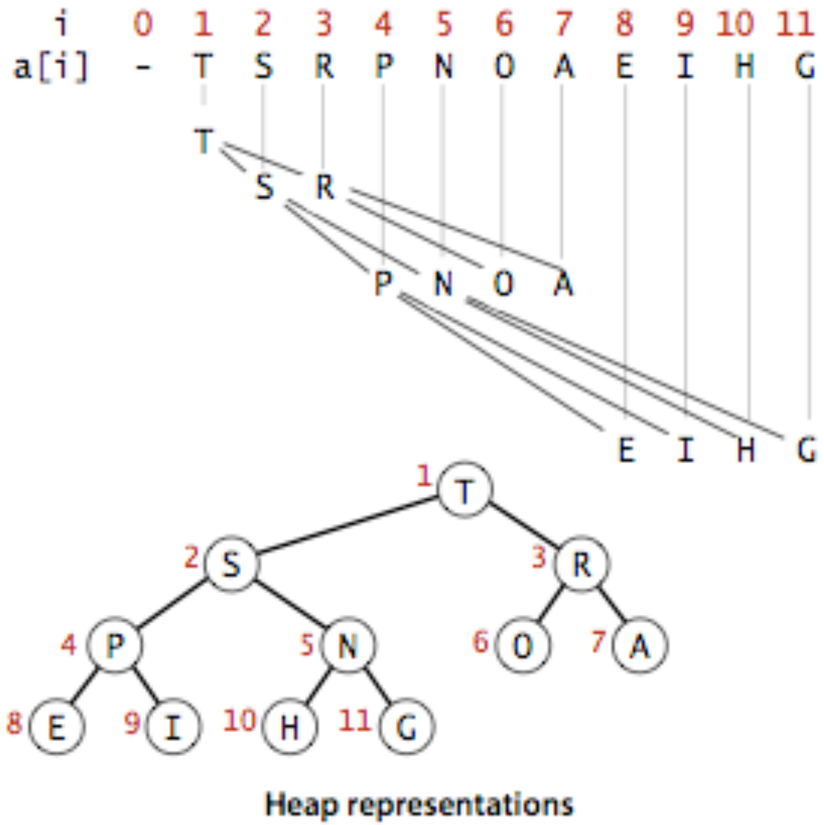
Heap / Tas ou Priority Queue/ File de Priorité

- API Java: java.util.PriorityQueue<E>

- API Livre:

public class MaxPQ<Key extends Comparable<Key>>	
MaxPQ()	*create a priority queue*
MaxPQ(int max)	*create a priority queue of initial capacity* `max`
MaxPQ(Key[] a)	*create a priority queue from the keys in* `a[]`
void insert(Key v)	*insert a key into the priority queue*
Key max()	*return the largest key*
Key delMax()	*return and remove the largest key*
boolean isEmpty()	*is the priority queue empty?*
int size()	*number of keys in the priority queue*

Implementation avec Binary Heap



Heap representations

Compression de texte: Huffman

"this is an example of a huffman tree"

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

195 bits, as opposed to 288 bits if 36 characters of 8 bits were used.