LINFO 1121
DATA STRUCTURES AND ALGORITHMS

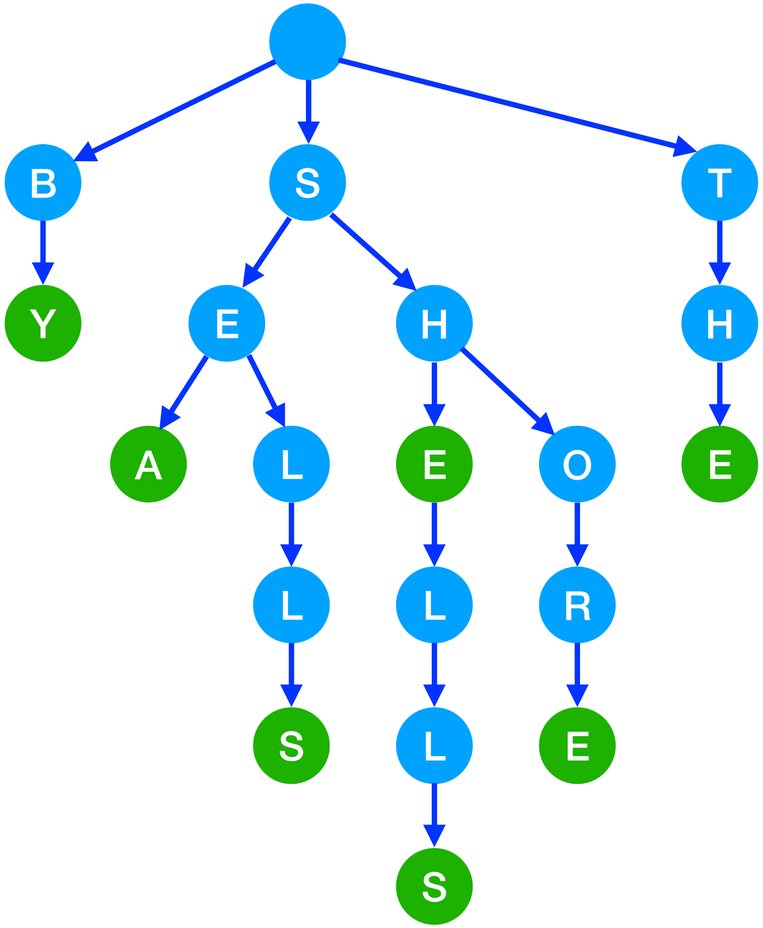
UCLouvain

Les tries et les tables de hachage (Hash tables)

Trie

- Un Trie est un arbre de recherche utilisé pour stocker et récupérer efficacement les clés d'un ensemble de strings.
  - Chaque noeud représente une lettre d'un string.
  - Chaque noeud à R enfants, ou R est la taille de l'alphabet.
  - Chaque noeud stocke une valeur associé au string formé par le chemin depuis la root
    - Exemple : si le nœud forme un string de l'ensemble ou pas.

She sells sea shells by the sea shore



AutoCompleter

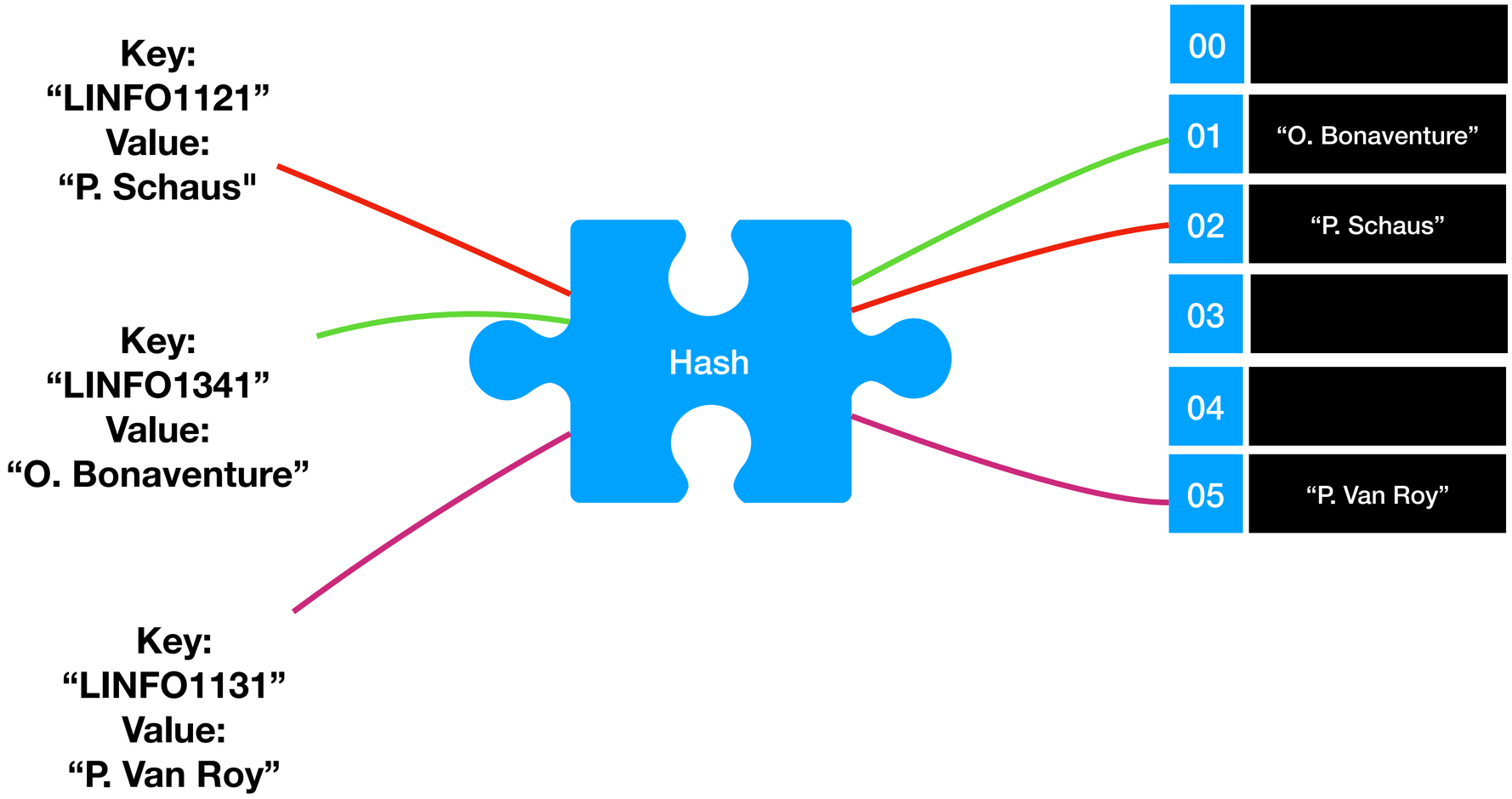
- Etant donné une liste de strings stockée dans un Trie et un préfixe, trouver le mot le plus court qui complète le préfixe.

- Solution ?

  1. Parcourir le trie en suivant le préfix.
  2. À partir du nœud trouvé, faire un BFS dans le sous-trie pour trouver le mot le plus court s'il existe.

Hash

- 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 Longs ?

- Java utilise cette fonction :

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

- Pourquoi ne pas simplement utiliser ? :

```
public static int hashCode(long value) {
    return (int) value;
}
```

Casting

```
long a = 10101266124699712 94L;

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

00001110000000100101100001100110101011010011100001111100101011110

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

01011010011100001111100101011110

Bitwise operation

```
System.out.println(Long.toBinaryString(-8));
```

```
11111111111111111111111111111000
```

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

```
System.out.println(Long.toBinaryString(-8 >> 2));
```

```
11111111111111111111111111111110
```

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

```
System.out.println(Long.toBinaryString(-8 >>> 2));
```

```
00111111111111111111111111111110
```

- Dans l’autre sens, uniquement <<

Hash sur les Longs ?

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

value

00001110000000100101100001100110101011010011100001111100101011110

value >>> 32

00000000000000000000000000000000000011100000010010110000110011 01

value ^ (value >>> 32)

00001110000000100101100001100110101010100011101000100100110010011

(int) value ^ (value >>> 32)

00000000000000000000000000000000010101000111010001001001100100 11

Tous les bits ont un impact pour le hash

Hash sur les Doubles ?

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

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

1 bit de signe, 11 bits pour l'exposant non signé et 52 bits pour la mantisse

- Le nombre -6 est représenté comme : $(-1)^1 \times 1.5 \times 2^{(1025-1023)}$

1-10000000001-1000000000000000000000000000000000000000000000000.

1025

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

- Quelle fonction de hash ? :

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

Hash sur les Doubles ?

- Est-ce qu'un nombre décimal positif et son opposé obtiennent toujours des fonctions de hachages différentes ?
  - Le nombre 6 est représenté comme :

    0-10000000001-1000000000000000000000000000000000000000000000000000.
  - Le nombre -6 est représenté comme :

    1-10000000001-1000000000000000000000000000000000000000000000000000.
  - Grace à `(bits >>> 32)` le bit de signe est aussi utilisé :
    - `hashCode(6.0)  ->  1075314688`
    - `hashCode(-6.0) -> -1072168960`

Hash et le casting

- 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 ?
  - Observation : le hash d'un entier sur 32 bits est l'entier lui-même
  - Faux
    - `((double) 5)` est représenté comme :

0-10000000001-0100000000000000000000000000000000000000000000000000

    - Ce qui après le shift et le XOR ne donne pas la même chose que la représentation de 5:

00000000000000000000000000000101

Hash et le casting

- 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 ?
  - Observation : le hash d’un entier sur 32 bits est l’entier lui-même
  - Deux cas possible :
    - Si le int est positif c’est vrai :
      - Si on cast un entier positif en long, 32 zeros sont simplement mis devant.
      - La formule du hash sur les long va appliquer le XOR avec un masque de 32x0 ce qui laisse l’entier initial intact
      - Exemple : 5 = (61x0)101 –> hashCode(5) = 000…101 ^ 000…000 = 000…101 = 5
    - Si le int est négatif c’est faux :
      - Si on cast un entier négatif en long, le nombre aura une représentation différente
      - Exemple : -5 = (61x1)011 –> hashCode(-5) = 111…011 ^ 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:

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

- 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 $n - i - 1 \geq k$. Donc tous les caractères (et donc tous les bits) ne seront pas pris en compte.

Hash sur les String

- Expliquez pourquoi utiliser R = 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 $2^8 = 256$.

- Deux clés entières sont ajoutées dans une table de hachage (separate chaining) : 2560 et 3072

  - Rappel : le hash d'un entier sur 32 bits est l'entier lui-même

Collision

- 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 ?
  - Ajout de $2560 = 10 \times 256$ et $3072 = 12 \times 256$
  - Il y a une collision car après %256 on arrive à 0 pour les deux
  - N'importe quel multiple de 256
- Si il y a une collision, peut-elle disparaître lors du prochain redimensionnement du tableau telle que dans l'implémentation du livre ?
  - Au prochain dimensionnement on arrive à M = 512 ce qui ne change rien à la collision.
  - $2560 = 5 \times 512$ et $3072 = 6 \times 512$

Collision

- 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 ?
  - 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 plus 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.

Hash de véhicule

- 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 ?
  - 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 $Y = \sum_{i=0}^{5} (26)^{5-i} Y_i \,.$
  - Final hash: $H = X \cdot 26^6 + Y$

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

Hash de véhicule

- Est-ce que votre fonction de hachage a la propriété que pour une taille de tableau N hypothétique de $10^{11} \cdot 26^6$ il n'y a pas de collision ?
  - Oui
    - Le maximum de H vaut : $(10^{11} - 1) \cdot 26^6 + (26^6 - 1) = 10^{11} \cdot 26^6 - 1 < |N|$
    - Le minimum de H vaut : $0 \cdot 26^6 + 0 \geq 0$
    - Il n'y a pas d'overlap entre la contribution des chiffres et des lettres. Donc chaque string à une valeur de hash unique.

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.

Hash Citoyens

- 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

- 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;
    }
}
```

LINFO 1121
DATA STRUCTURES AND ALGORITHMS

UCLouvain

Union-Find, Heap, Text Compression

Union-Find

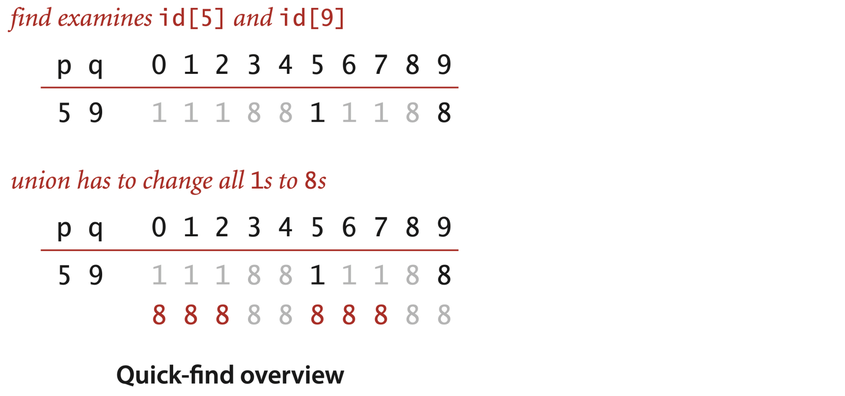
public class UF		
	UF(int N)	*initialize N sites with integer names* (0 *to* N-1)
void	union(int p, int q)	*add connection between* p *and* q
int	find(int p)	*component identifier for* p (0 *to* N-1)
boolean	connected(int p, int q)	*return* true *if* p *and* q *are in the same component*
int	count()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

Union-find API

- Compter des groupes
- Vérifier si deux éléments sont dans le même groupe

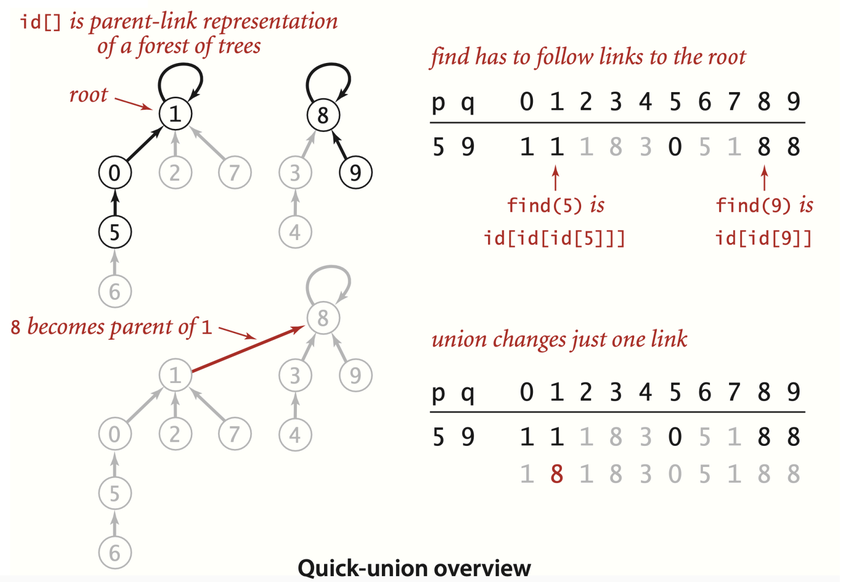
Implémentation

- Deux implémentation différentes en fonction des besoins :
  - Quick-find
    - Stock pour chaque élément l'id de son groupe



Quick-find overview

  - Quick-Union
    - Basé sur des arbres représentés dans un tableau
    - Stock pour chaque élément l'id de l'élément parent
    - Un composant est un arbre dans le tableau



Quick-union overview

Heap / Tas ou Priority Queue / File de Priorité

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

- API Livre:

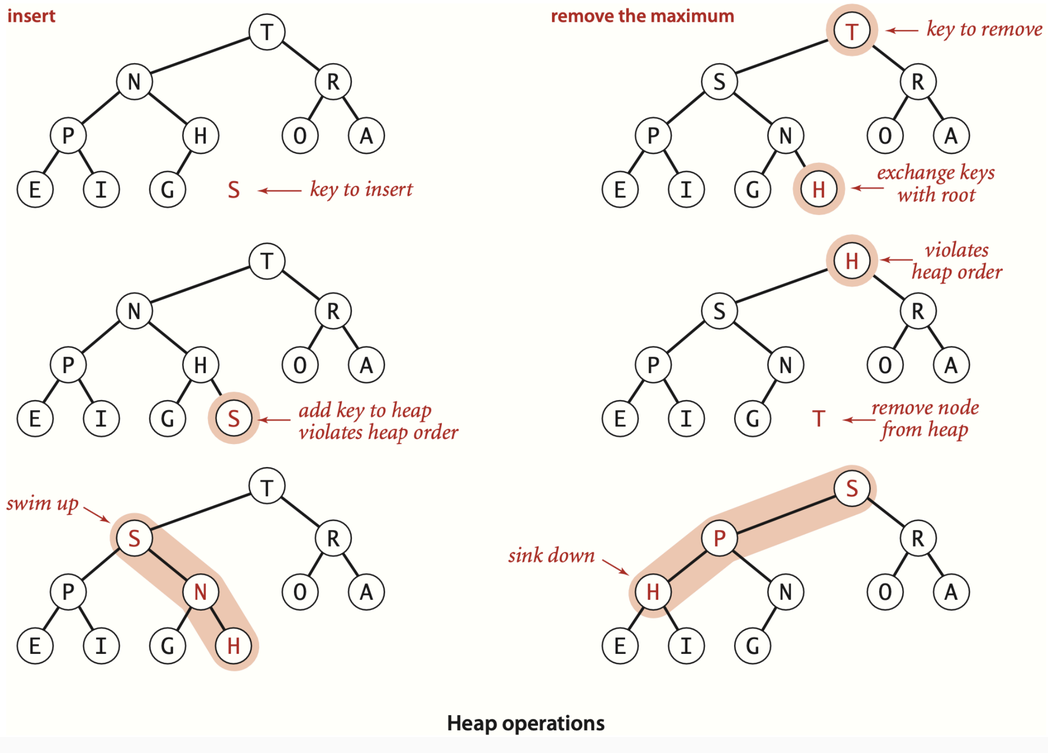
public class MaxPQ<Key extends Comparable<Key>>		
	MaxPQ()	*create a priority queue*
	MaxPQ(int max)	*create a priority queue of initial capacity* `max`
	MaxPQ(Key[] a)	*create a priority queue from the keys in* `a[]`
void	insert(Key v)	*insert a key into the priority queue*
Key	max()	*return the largest 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*

API for a generic priority queue

- Récupérer rapidement l'élément maximum/minimum d'un ensemble qui n'est pas fix.

Implementation avec Binary Heap

- Un arbre binaire représenté dans un tableau
  - Chaque noeud de l'arbre est plus grand ou égal à ses deux enfants.
    - Le maximum est la root de l'arbre.
  - `insert` : Ajoute le noeud a la base de l'arbre et faire remonter en respectant la propriété.
  - `delMax` : Retire la root et met à sa place un élément de la base de l'arbre. Fait ensuite descendre le noeud en respectant la propriété.



Heap operations

Compression de texte: Huffman

- Idée pour la compression : assigner des encodages moins long aux caractère les plus fréquents.

- Huffman construit un arbre en utilisant la fréquence de chaque caractère. Le code utilisé pour chaque caractère est le chemin dans cet arbre.