# INF421-a
# Bases de la programmation et de l'algorithmique

## (Bloc 4 / 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

16 septembre 2005

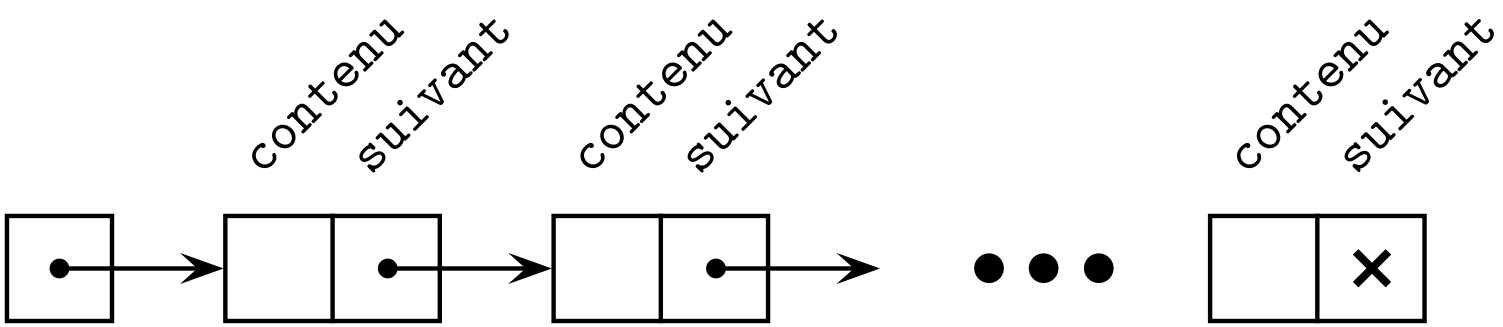
# Aujourd'hui

# Une liste d'entiers en Java

```
class Liste {
  int contenu;
  Liste suivant;
  Liste (int x, Liste a) {
    contenu = x;
    suivant = a;
  }
}
```

Ainsi, `null` est la liste vide .
Pour créer une liste d'un élément (2 ici) : `new Liste(2, null)`.
Pour la liste des $n$ premiers entiers pairs :

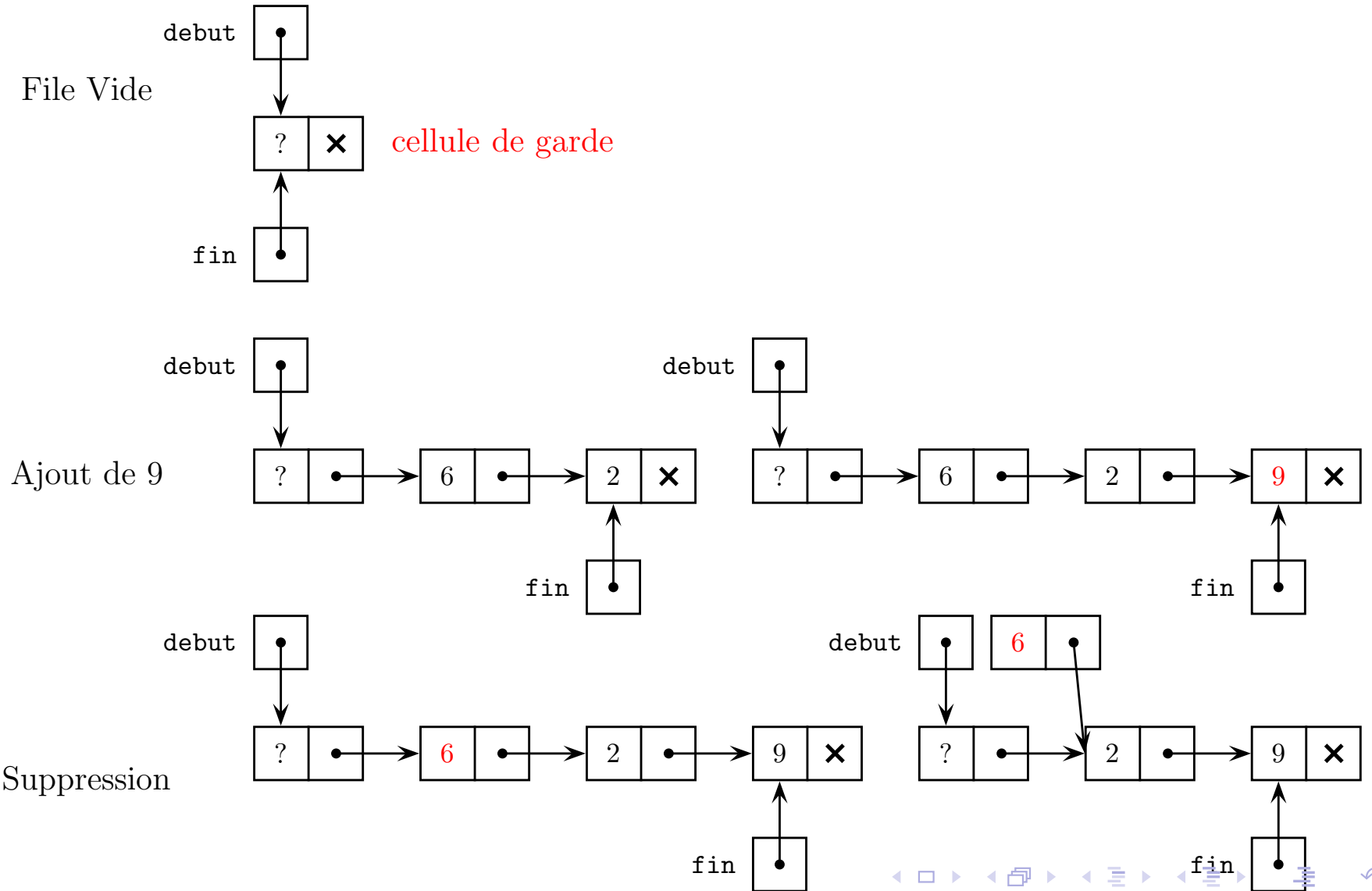
```
Liste l = null;
for (int i = n; i ≥ 1; i--)
  l = new Liste(2 * i, l);
```



# Les gardes : une CONVENTION

- Il arrive qu'on souhaite représenter une "structure vide" non pas par `null` mais par une des briques de base de la structure
- Ex 1 : Une liste gardée. La première cellule ne contient rien, elle sert à simplifier le codage de certaines méthodes
- Ex 2 : Les files (rappel)
  - On conserve à la fois une référence sur le premier et sur le dernier élément de la liste
  - Les ajouts se font à la fin de la liste
  - Il faut donc modifier à chaque ajout la référence sur le dernier élément de la liste
  - Les opérations de suppression concernent la tête de la liste
  - La dernière cellule de la liste existe toujours ($\rightarrow$ ajout simplifié du fait de la garde)

# Les files avec une liste chaînée gardée



# Les listes d'entiers (QCM de l'an passé)

```
static void affiche(Liste l) {
    if (l == null) return;
    affiche(l.suivant);
    System.out.print(l.contenu+ " ");
}
```

Que fait alors le code suivant ?

```
class Test {
    public static void main(String args[]) {
        Liste l = new Liste(2,new Liste(3,new Liste(1,null)));
        Liste.affiche(l);
    }
}
```

# Les listes d'entiers (QCM de l'an passé)

```
static void affiche2(Liste l) {
    if (l == null) return;
    affiche2(l.suivant);
    System.out.print(l.contenu+" ");
    affiche2(l.suivant);
}
```

Que fait alors le code suivant ?

```
class Test {
    public static void main(String args[]) {
        Liste l = new Liste(2,new Liste(3,new Liste(1,null)));
        Liste.affiche2(l);
    }
}
```

# Les listes d'entiers (QCM de l'an passé)

```
static void decale(Liste l) {
    while (l ≠ null) l = l.suivant;
}
static void affiche(Liste l) {
    if (l == null) return;
    affiche(l.suivant);
    System.out.print(l.contenu+ " ");
}
```

Que fait alors le code suivant ?

```
class Test {
    public static void main(String args[]) {
      Liste l = new Liste(2,new Liste(3,new Liste(1,null)));
      Liste.decale(l);
      Liste.affiche(l);
    }
}
```

# Les listes d'entiers (QCM de l'an passé)

```
static void decale2(Liste l) {
    if ((l ≠ null) && (l.suivant ≠ null)) {
        decale2(l.suivant);
        l.suivant = new Liste(l.contenu,l.suivant);  } }
static void affiche(Liste l) {
    if (l == null) return;
    affiche(l.suivant);
    System.out.print(l.contenu+ " "); }
```

Que fait alors le code suivant ?

```
class Test {
    public static void main(String args[]) {
      Liste l = new Liste(2,new Liste(3,new Liste(1,null)));
      Liste.decale2(l);
      Liste.affiche(l); }}
```

# Aujourd'hui

Un ultime retour sur les listes

Skiplists suite et fin

Quelques aspects objets de Java

# L'art du raccourci

- Rappel : recherche d'un élément d'une liste chaînée en $O(n)$
  - Question : Dans une liste triée ?
  - Question : Dans un tableau trié ?
- Rappel : suppression d'un élément d'une liste chaînée en $O(n)$
  - Question : Dans une liste triée ?
  - Question : Dans un tableau trié ?

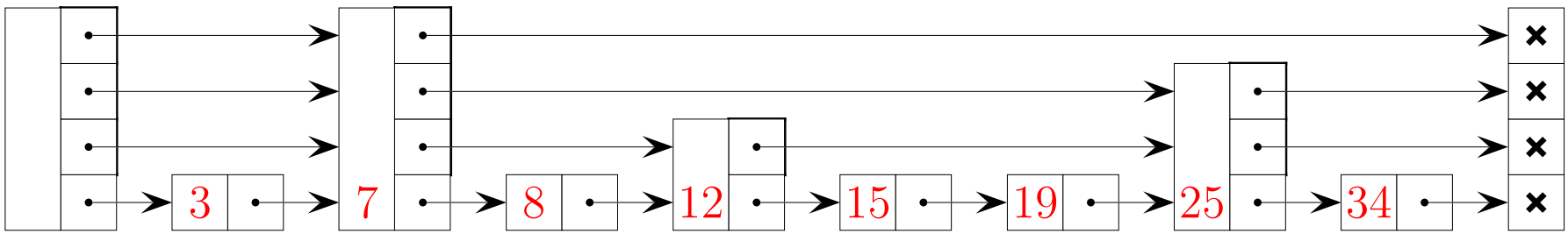
Pb fondamental : La liste triée ne permet pas d'accéder rapidement à des éléments "lointains"

# Une skiplist d'entiers

Une suite triée de cellules constituées d'un entier et d'un tableau de références "`suivants`" vers d'autres cellules

- La hauteur d'une cellule = la taille du tableau `suivants` (variable d'une cellule à l'autre)
- La hauteur de la liste est la plus grande des hauteurs des cellules
- La dernière et la première cellule (de garde) sont de hauteurs maximales et le tableau `suivants` de la dernière cellule ne contient que des `null`
- `suivants[i]` est une référence vers la prochaine cellule de hauteur $\geq i + 1$
- Les hauteurs sont $\geq 1$ et `suivants[0]` est une référence vers la cellule immédiatement consécutive

# Une skiplist d'entiers

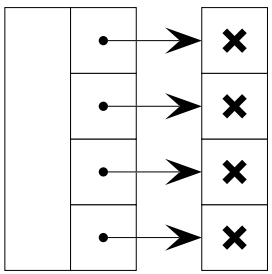



- La hauteur d'une cellule est déterminée de façon aléatoire.
- La distribution des hauteurs contrôle la complexité de l'algorithme
- Des détails plus tard sur cette dimension aléatoire

# Une SkipList d'entiers

```
class SkipListInt {
    int val;
    SkipListInt[] suivants;
    // probabilite
    static final double p = 0.5;
    // une hauteur max pour simplifier
    static final int maxH = 6;
    SkipListInt() {
        suivants = new SkipListInt[maxH + 1]; }
    SkipListInt(int haut, int x) {
        this.val = x;
        suivants = new SkipListInt[haut + 1]; }
}
```

# Une SkipList vide

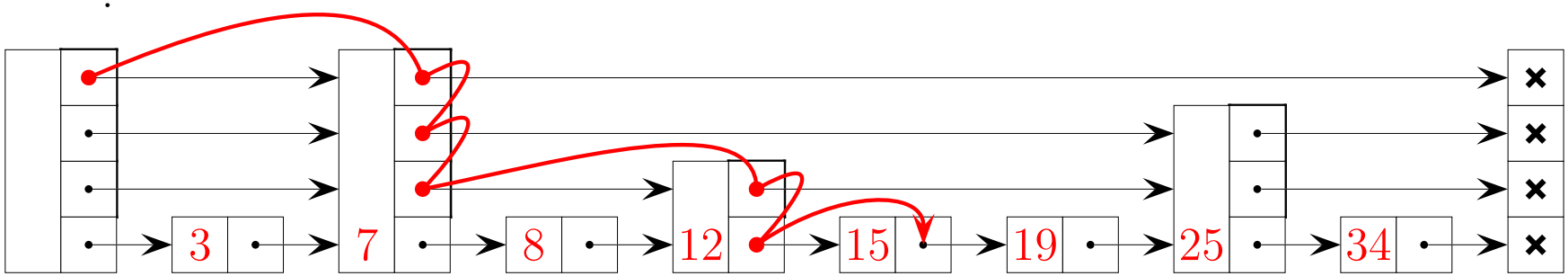


Une cellule de garde pour simplifier le codage (de hauteur `maxH+1`)

```
SkipListInt() {
    suivants = new SkipListInt[maxH + 1];
}
```

# Rechercher un élément dans une skiplist

- Utiliser les cellules "hautes" pour aller le plus loin possible dans la liste
- sans dépasser l'élément que l'on cherche
- Exemple : Recherche de 15 dans la liste ci-dessous : *Droite, bas, bas, droite, bas, droite*




- Analyse de la complexité dans le pire des cas : ? ? ?
- En moyenne c'est bien mieux

# Rechercher un élément dans une skiplist

La référence de niveau $u$ d'une cellule (de hauteur $\geq u$) est la $u$ème référence du tableau `suivants`

1. Partir du niveau le plus haut de la cellule de tête
2. Tant qu'on ne "dépasse pas" l'élément que l'on cherche, se déplacer "vers la droite" en restant au même niveau
3. Si c'est possible, descendre d'un niveau et itérer 2 sinon (*i.e.*, on est au niveau 0) sortir

En pratique :

- Une méthode `boolean chercher(int niveau, int x, SkipListInt a)` qui lance la recherche à partir du niveau `niveau` de la skiplist `a`
- Une méthode `boolean chercher(int x, SkipListInt a)` qui lance la recherche sur toute la skiplist

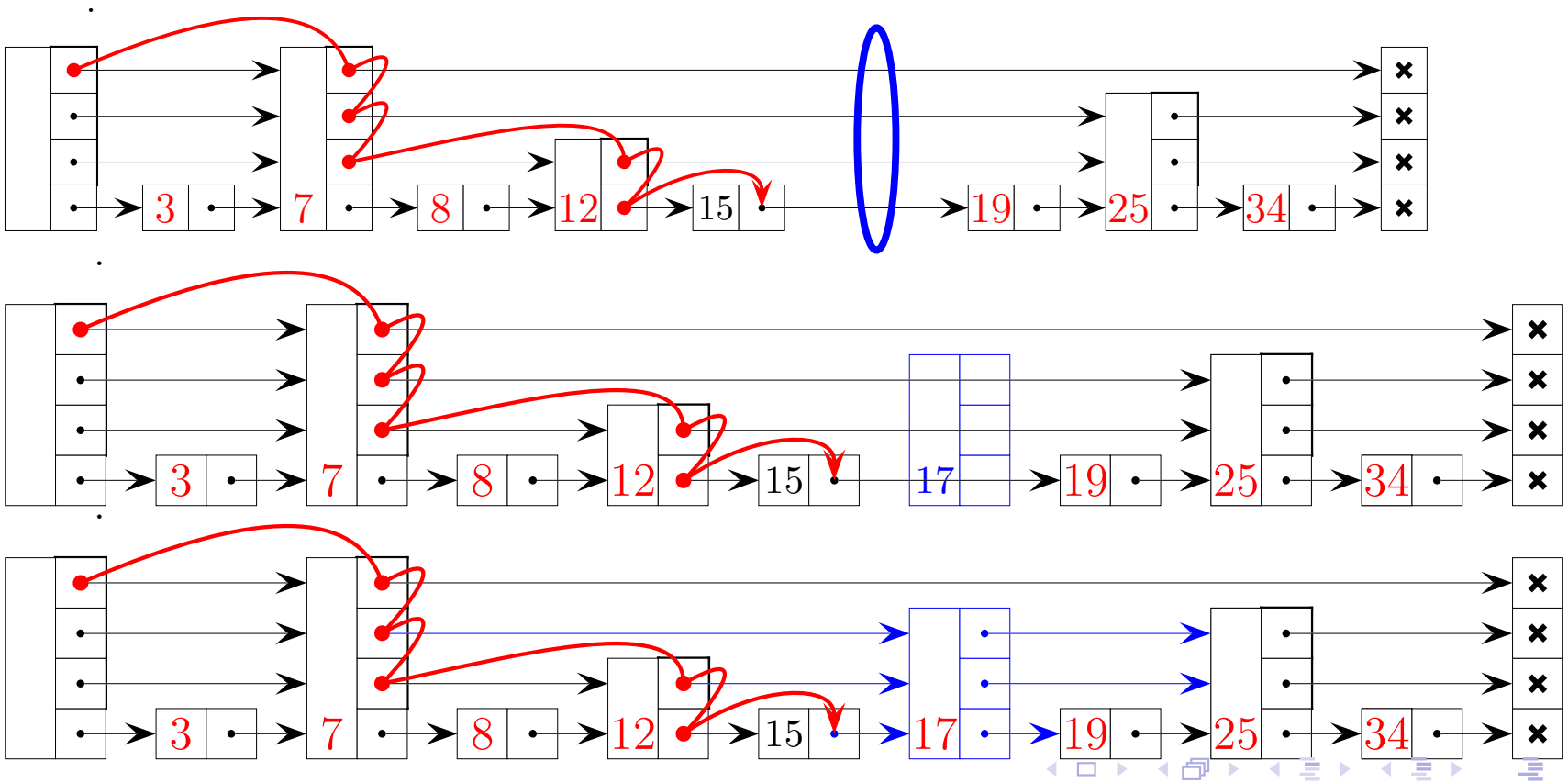
# Rechercher un élément dans une skiplist

```
static boolean chercher(int x, SkipListInt a) {
    return chercher(a.suivants.length - 1, x, a);
}

/* Interrompre la recherche juste avant x */
static boolean chercher(int niveau, int x, SkipListInt a) {
    if (niveau < 0)
        return (a.suivants[0] ≠ null &&  a.suivants[0].val == x);
    if (a.suivants[niveau] == null ||
        a.suivants[niveau].val ≥ x)
        return chercher(niveau - 1, x, a);
    return chercher(niveau, x, a.suivants[niveau]);
}
```

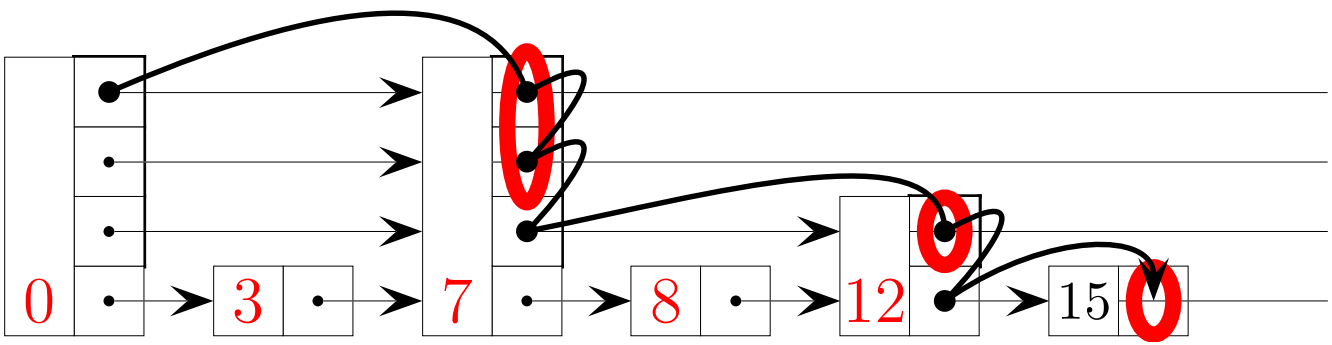
# Insérer un élément dans une skiplist

- Chercher l'endroit où insérer l'élément, calculer le niveau (rnd)
- Mettre à jour la structure en calculant le front
- Exemple : insertion de 17 (hauteur arbitraire 3)



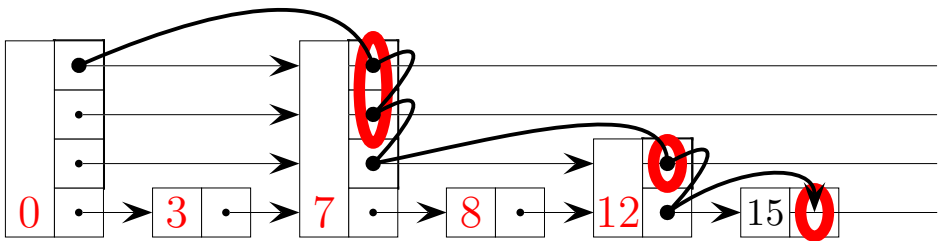
# Calcul du front

- Le front = les références qui correspondent aux diminutions du niveau
- Ce sont les références du front qui vont être modifiées
  - Toutes les autres références sont conservées
  - Le nombre de modification est au plus égal à la hauteur de la liste




En pratique, le front est un tableau de skiplist.

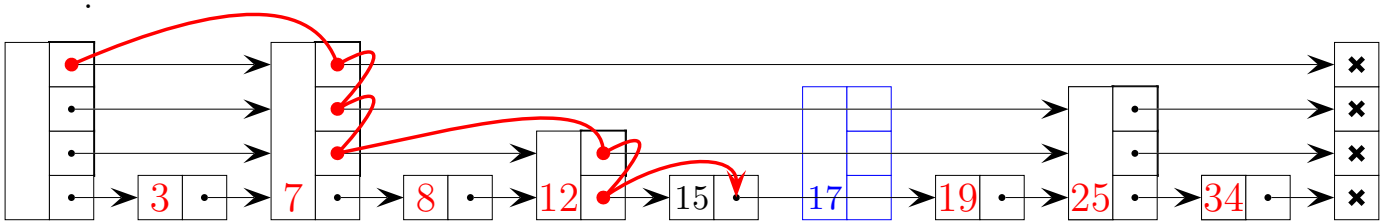
# Le front




Calcul du front = comme une recherche (à droite tant que possible puis en bas)

```
static SkipListInt[] FrontAvant(int x, SkipListInt a) {
    SkipListInt[] frt = new SkipListInt[maxH + 1];
    for (int i = maxH; i ≥ 0; i--) {
        while (a.suivants[i] ≠ null &&
                a.suivants[i].val < x)
            a = a.suivants[i];
        frt[i] = a;
    }
    return frt;
}
```

# Insertion




- Les éléments du front de hauteur $\leq$ au niveau de l'élément à insérer sont égales à cet élément
- Les autres éléments du front ne bougent pas
- Les suivants de l'élément inséré = "anciens" suivants du front

```
static void inserer(int x, SkipListInt a) {
    SkipListInt[] t = FrontAvant(x, a);
    int niveau = 0; // générer un niveau aléatoire (détails + tard)
    while (niveau < maxH && Math.random() < p) niveau++;
    SkipListInt b = new SkipListInt(niveau, x);
    for (int i = 0; i ≤ niveau; i++) {
        b.suivants[i] = t[i].suivants[i];
        t[i].suivants[i] = b;
    }
}
```

# Suppression (d'une occurrence)

- Calculer le front
- Si l'élément après le front n'est pas celui qui est à supprimer sortir (rien à enlever dans la liste)
- Mettre à jours les références du front :
  - Les éléments du front qui pointent sur l'élément à supprimer pointent sur l'élément suivant

```
static void supprimer(int x, SkipListInt a) {
    SkipListInt[] t = FrontAvant(x, a);
    SkipListInt b = t[0].suivants[0];
    if (b.val == x)
        for (int i = 0; i ≤ maxH; i++)
            if (t[i].suivants[i] == b)
                t[i].suivants[i] = b.suivants[i];
}
```

# Les skiplists : Un algorithme randomisé

- Rappel : un algorithme randomisé est contrôlé par une suite de tirages aléatoires. (*e.g.* pile ou face)
- L'unique dimension aléatoire des skiplists : le niveau d'un élément à insérer
  - $p$ un réel fixé $(0 < p < 1)$
  - Probabilité [niveau $\geq q$] $= p^q$
  - Probabilité [niveau $< q$] $= 1 - p^q$
  - Probabilité [niveau $= q$] $= p^q(1 - p^{q+1})$
- En pratique, dans la méthode insérer :

```
int niveau = 0; // Un niveau aléatoire
while (niveau < maxH && Math.random() < p)
   niveau++;
```

`Math.random()` renvoie un `double` tiré aléatoirement dans l'intervalle $[0, 1)$

# Les skiplists : Analyse de la complexité

Quel est l'espace utilisé ?

- Probabilité [niveau $= q$] $= p^q(1 - p^{q+1})$
- Espace utilisé $\sum_{i=1}^{n} \sum_{q=0}^{\infty} q p^q (1 - p^{q+1}) = n \times \sum_{q=0}^{\infty} q p^q (1 - p^{q+1})$
- Espace linéaire $O(n)$

# Les skiplists : Analyse de la complexité

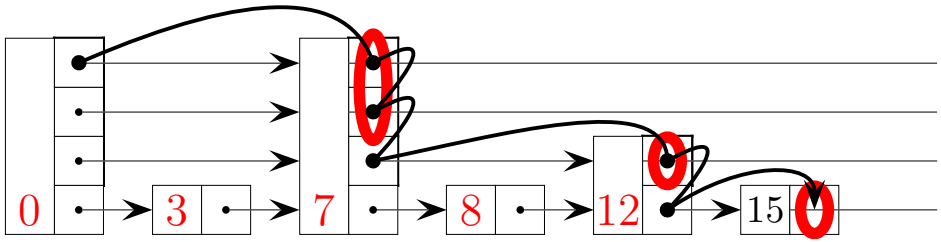
Quel est la hauteur de la skiplist ?

- Probabilité [niveau d'une cellule $< q$] $= 1 - p^q$
- Probabilité [au moins une cellule de hauteur $q$] $= 1 - (1 - p^q)^n$
- Pour $q$ grand, probabilité [au moins une cellule de hauteur $q$] $= np^q$
- La probabilité que la liste soit de hauteur $\geq 3 \log_{\frac{1}{p}} n$ est donc

$$np^{3 \log_{\frac{1}{p}} n} = \frac{n}{n^3} = \frac{1}{n^2}$$

- Pour $p = \frac{1}{2}$, $n = 10^6$, la probabilité d'avoir une hauteur plus grande que $3 \times 20 = 60$ est plus petite que $10^{-12}$.
- Avec une très grande prob., la skiplist est de hauteur $O(\log n)$

# Insérer dans une skiplist : Analyse de la complexité




- Une fois le front calculé, la complexité est linéaire en la hauteur
- Pour évaluer la complexité, on "remonte" le front en partant de la dernière cellule du front (niveau 0)
- Soit $C(h)$ le coût qui nous permet de grimper de $h$ niveaux consécutifs.
- La cellule précédente dans le chemin est de même niveau (probabilité $1-p$) ou du même niveau $+1$ (probabilité $p$) et donc $C(h) = (1-p)(O(1)+C(h)) + p(O(1)+C(h-1))$
  Soit donc $C(h) = O(h)$.
- Le calcul du front est donc proportionnel à la hauteur de la liste ; Soit donc, avec une très grande probabilité, en $O(\log n)$.

# Le téléphone

Problématique de la semaine dernière : Recherche d'information dans un ensemble dynamique

- Associer un nom (*i.e.*, une chaîne de caractères)
- à un numéro de téléphone (une autre chaîne de caractères)

```
"Camille Abate"        "04.91.63.50.30"
"Philippe Zito"        "06.94.16.13.38"
"Laurie Zimmerman"     "04.32.32.39.63"
"Ugo Zelaya"           "04.70.61.50.56"
"Laure Zamor"          "02.67.37.88.41"
"Oceane Archambault"   "06.76.06.22.09"
```

# Une SkipList de “couples”

```
class Couple {
    String nom;
    String tel;
    Couple(String nom,
           String tel) {
        this.nom = nom;
        this.tel = tel;
    }
    // uniquement pour
    // l'initialisation de
    // la skiplist
    Couple() {
        this.nom = ‘‘’’;
        this.tel = ‘‘’’;
    }
}
```

```
class SkipList {
    Couple cpl;
    SkipList[] suivants;
    // probabilite (détails plus tard)
    static final double p = 0.5;
    // Le niveau maximal possible
    static final int maxH = 20;
    SkipList() { // pour la dern cellule
        this.cpl = new Couple("", "");
        suivants = new SkipList[maxH + 1];
    }
    SkipList(int niveau, Couple cpl) {
        this.cpl = cpl;
        suivants = new SkipList[niveau + 1];
    }
}
```

# Rechercher dans une skiplist de “couples”

```
static String telDe(int niveau, String nom, SkipListCpl a) {
    if (niveau < 0)
        if (a.suivants[0] ≠ null &&
            a.suivants[0].cpl.nom.equals(nom))
            return a.suivants[0].cpl.tel; // GAGNE
        else return null;                 // PERDU
    if (a.suivants[niveau] == null ||
        a.suivants[niveau].cpl.nom.compareTo(nom) ≥ 0)
        return telDe(niveau - 1, nom, a); // EN BAS
    return telDe(niveau, nom, a.suivants[niveau]); // A DROITE
}
static String telDe(String nom, SkipListCpl a) {
    //if (a == null) return "unknown";
    return telDe(a.suivants.length - 1, nom, a);
}
```

# Les skiplists d'entiers *vs.* les skiplists de couples

- Quelle partie du code a changé ?
  - Test d'égalité
  - Comparaison d'éléments
- Comment éviter de réécrire deux fois le même code ?
- Un début de réponse par les objets

Objectif : Construire une skip liste d'objets quelconques que l'on peut facilement réutiliser

# Aujourd'hui

Un ultime retour sur les listes

Skiplists suite et fin

Quelques aspects objets de Java

# OO : une très brève introduction

- En C, en Pascal, on “découpe” le code en plusieurs procédure
- C’est une décomposition “spécifique” $\longrightarrow$ Problème de génie logiciel : réutilisabilité
- Idée simple : Travailler sur les fonctionnalités ; les regrouper dans des objets
- Un objet
  - a des attributs et
  - peut envoyer/recevoir des messages (invocation de méthodes)

# L'héritage

- Créer une classe à partir d'une autre classe
- la première = classe parente
- la deuxième = sous-classe ou classe dérivée
- Lorsqu'une classe X hérite d'une classe Y, elle dispose des données et des méthodes de Y.
- On peut ajouter des méthodes / données à X et on peut redéfinir certaines méthodes (rappel `toString`)
- Le cours d'informatique fondamentale (INF 431) pour une introduction détaillée
- Un aperçu de l'héritage : les interfaces

# Les interfaces

- Une méthode abstraite = une méthode sans corp (= une signature rien de plus), qui sera définie dans une sous classe
- Une interface = une classe dont toutes les méthodes sont abstraites (et dont toutes les données membres sont finales)
- Une interface doit toujours être implémentée (*i.e.*, il faut définir une autre classe qui `implements` l'interface en remplissant ses méthodes abstraites)
- Attention, les méthodes définies dans les interfaces sont toujours publiques quand elles sont implémentées
- Attention, les méthodes définies dans les interfaces ne peuvent pas être statiques
- Exemple : l'interface `Comparable` du package `java.lang`

# L'interface Comparable

```
public interface Comparable {
    public int compareTo(Object o);
}
```

- Ordre global sur les instances des classes qui implémentent cette interface
- Une seule méthode : `int compareTo(Object o)` → comparer un objet o passé en argument à `this` l'instance de classe courante.
- L'ordre imposé par cette interface se réfère à l'ordre naturel de la classe et la méthode compareTo() de la classe se réfère également à sa méthode de comparaison naturelle.
- Exemple : la classe `String` implémente Comparable

# L'interface Comparable

```
class test {
    public static void main(String[] args) {
        String s1 = "azertyuiop";
        String s2 = "nbvcxw";
        System.out.println("s1.compareTo(s2) = " +  s1.compareTo(s2));
        System.out.println("s2.compareTo(s1) = " +  s2.compareTo(s1));
    }
}
```

Nous donne

```
java test
s1.compareTo(s2) = -13
s2.compareTo(s1) = 13
```

# L'interface Comparable

```
class X implements Comparable {
    String nom;
    int rangSortie;
    X(String nom, int rangSortie) {
        this.nom = nom;
        this.rangSortie = rangSortie;}
    // Attention NON STATIQUE
    public int compareTo(Object unAutre) {
        // Un Cast (conversion de type)
        X unAutreX = ((X) unAutre);
        return rangSortie - unAutreX.rangSortie;}
    public String toString() {
        return nom + " (" + rangSortie + ")";}
}
```

Quel intérêt ?

- Utiliser les méthodes définies dans la classe Comparable !

# L'interface Comparable

```
class test {
    public static void main(String[] args) {
        X[] Xs = new X[3];
        Xs[0] = new X("John Ford", 69);
        Xs[1] = new X("John Cassavetes", 45);
        Xs[2] = new X("Jean Eustache", 67);
        Arrays.sort(Xs); // méthode statoque de Arrays
        for(int i = 0; i < Xs.length; i++)
            System.out.println(Xs[i]);
    }
}
```

Nous donne

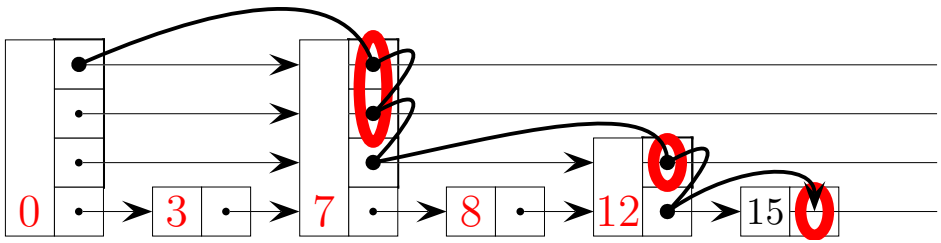
```
John Cassavetes (45)
Jean Eustache (67)
John Ford (69)
```

# Une skiplist générique

- Utiliser et manipuler une skiplist d'objets triés selon une clef
- Les clefs sont des `Comparables`

```
class SkipListGen  {
    Object o;
    Comparable key;
    SkipListGen[] suivants;
    static final double p = 0.5;
    static final int maxH = 20;
    SkipListGen() {
        suivants = new SkipListGen[maxH + 1];
    }
    SkipListGen(int niveau, Object o, Comparable key) {
        this.o = o;
        this.key = key;
        suivants = new SkipListGen[niveau + 1];
    }
}
```

# La clacul du front dans une skiplist générique




À droite tant que possible puis en bas

```
static SkipListGen[] FrontAvant(Comparable k, SkipListGen a) {
    SkipListGen[] frt = new SkipListGen[maxH + 1];
    for (int i = maxH; i ≥ 0; i--) {
        while (a.suivants[i] ≠ null &&
                a.suivants[i].key.compareTo(k) < 0)
            a = a.suivants[i];
        frt[i] = a;
    }
    return frt;
}
```

# L'insertion dans une skiplist générique

```
static void inserer(Comparable k, Object o, SkipListGen a) {
    SkipListGen[] t = FrontAvant(k, a);
    int niveau = 0; // générer un niveau aléatoire
    while (niveau < maxH && Math.random()() < p)
        niveau++;
    SkipListGen b = new SkipListGen(niveau, o, k);
    for (int i = 0; i ≤ niveau; i++) {
        b.suivants[i] = t[i].suivants[i];
        t[i].suivants[i] = b;
    }
}
```

# La suppression dans une skiplist générique

```
static void supprimer(Comparable k, SkipListGen a) {
    SkipListGen[] t = FrontAvant(k, a);
    SkipListGen b = t[0].suivants[0];
    if (b.key.compareTo(k) == 0)
        for (int i = 0; i ≤ maxH; i++)
            if (t[i].suivants[i] == b)
                t[i].suivants[i] = b.suivants[i];
}
```

# Un exemple d'utilisation d'une skiplist générique

Pour un annuaire téléphonique

```
SkipListGen skl = new SkipListGen();
String nom = ``Roberto'';
String tel = ``01 69 33 38 00'';
SkipListGen.inserer(nom, tel, skl);

// Bien plus loin
Object o = SkipListGen.chercher(nom, skl);
String s = ((String) o);
System.out.println (nom + " Téléphone :" + s);
```

# Un exemple d'utilisation d'une skiplist générique

- On veut travailler sur des entiers. Quel est le probleme ?
  - `int` n'est pas un objet ! ! !
- Il faut "encapsuler" les entiers dans des objets

```
class Integer  implements Comparable {
   int entier ;
   Integer (int i) {
     entier = i;
   }
   int intValue() {
     return entier;
   }
}
```

Cette classe est disponible dans le package java.lang

# Deux implémentations d'une interface

Nous avons vu plusieurs méthodes pour rechercher des informations dans un ensemble dynamique

- Les listes d'association
- Les tables de hachage
  - Le chaînage : Les clefs qui ont la même valeur de hachage sont mises dans la même liste
  - L'adressage ouvert : On cherche la première place libre dans le tableau après la valeur de hachage
- Les skiplists
- D'autres techniques plus tard

Dans tous les cas, on effectue des recherches, des insertions et des suppressions. Un "utilisateur" veut pouvoir changer facilement basculer d'un algorithme à l'autre.

# Deux implémentations d'une interface

```
interface RechercheDynInfo {
    Object chercher(Comparable k);
    void inserer(Comparable k, Object o);
    void supprimer(Comparable k);
}
```

Il nous reste à implémenter cette interface

- Les listes d'association
- les skiplists

Rappel : pas de static dans les interfaces !
$\longrightarrow$ Enlever les static

# Deux implémentations d'une interface

```
class ListeAssoc implements RechercheDynInfo {
    Object o; Comparable key; ListeAssoc suivant;
    ListeAssoc() {}
    ListeAssoc(Comparable k, Object o, ListeAssoc suivant) {
        this.key = k; this.o = o; this.suivant = suivant;
    }
    Object chercherRec(Comparable k) {
        if (key ≠ null && key.compareTo(k) == 0) return o;
        if (suivant == null) return null;
        return suivant.chercherRec(k);
    }
    public Object chercher(Comparable k) {
        if (suivant == null) return null;
        return suivant.chercherRec(k);
    }
    public void inserer(Comparable k, Object o) {
        suivant = new ListeAssoc(k, o, suivant);
    }
```

# Deux implémentations d'une interface

```
class SkipListGen implements RechercheDynInfo  {
    Object o; Comparable key; SkipListGen[] suivants;
    static final double p = 0.5; static final int maxH = 20;
    // constructeurs ...
    Object chercher(int niveau, Comparable k) {
        if (niveau < 0)
            if (suivants[0] ≠ null && suivants[0].key.compareTo(k) == 0
                return suivants[0].o;
            else return null;
        if (suivants[niveau] == null ||
            suivants[niveau].key.compareTo(k) ≥ 0)
            return chercher(niveau - 1, k);
        return suivants[niveau].chercher(niveau, k);
    }
    public Object chercher(Comparable k) {
        //if (a == null) return "unknown";
        return chercher(suivants.length - 1, k);
    }
```

# Deux implémentations d'une interface

Pour utiliser votre structure :

```
RechercheDynInfo info = new ListeAssoc();
```

ou

```
RechercheDynInfo info = new SkipListGen();
```

Remarquer que lors de l'utilisation, on manipule `info` de type `RechercheDynInfo`. Le choix de l'implémentation se fait sur une unique ligne !

# Le TP du jour

Un moteur de recherche pour faire de l'ombre a Google