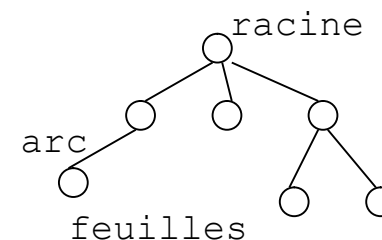


Structures d'arbre

- Liens hiérarchiques entre éléments

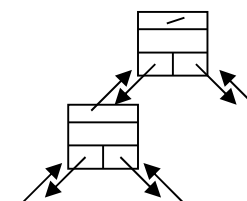
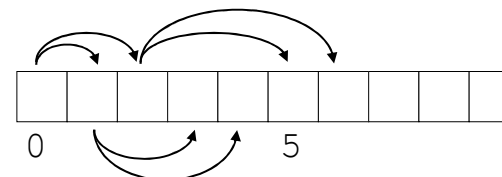


Terminologie

- Arbre (avec racine), sommet (= nœud), lien (= arc) hiérarchique
- Racine, sommet intermédiaire, feuille. Forêt = collection d'arbres
- Ancêtres/descendants, père/fils, frère. Taille d'un arbre (nombre de sommets)
- Sous-arbres. Arbre quelconque. Arbre M-aire (au plus M fils par sommet)
- Chemin d'un sommet à la racine. Hauteur d'un arbre (= nombre de niveaux)
- Profondeur d'un sommet (= longueur du chemin à la racine)
- Arbre binaire. Fils gauche/droit

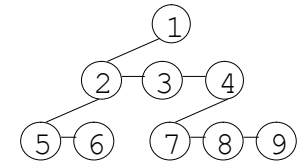
Représentation d'arbres binaires

- Chaînage de nœuds :
arbre = \emptyset ou élément + parent + sous-arbre gauche + sous-arbre droit
- Dans un tableau (représentation implicite) :
 - chaque case contient un élément ou indique son absence. Racine en 0
 - fils gauche du sommet i dans la case $2i+1$; fils droit dans la case $2i+2$
 - parent du sommet i dans la case $(i-1)/2$
- Parfois, un chaînage plus simple suffit (seulement vers le haut/bas)

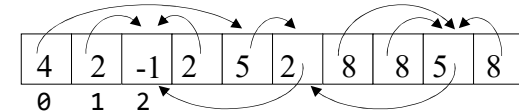


Représentation d'arbres quelconques

- Au moyen d'un arbre binaire !
 - principe du "***fils gauche, frère droit***"
 - attention à ne pas confondre la structure "logique" et "physique"...



- Chaînage de nœuds : arbre = \emptyset ou élément + parent + collection de sous-arbres
 - Tableau des parents
-

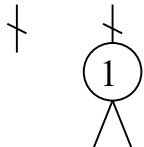


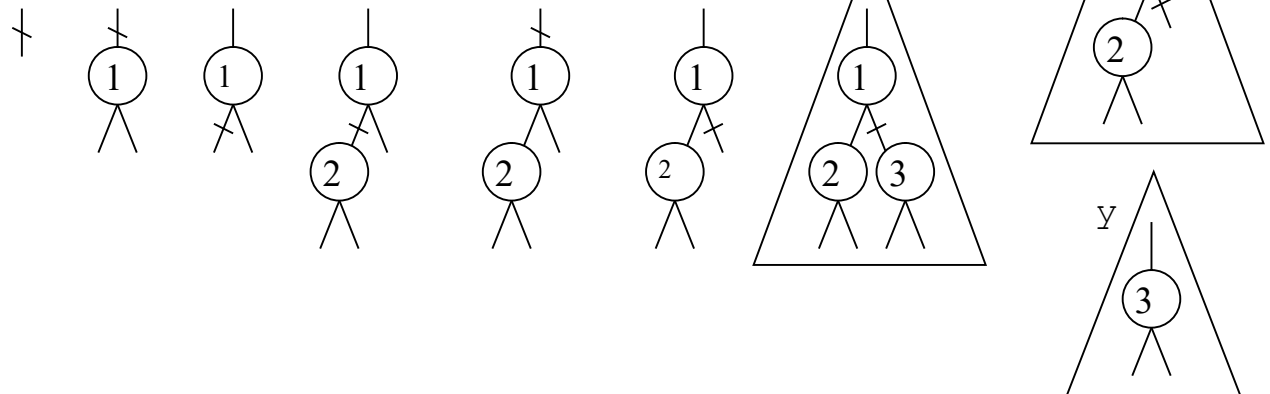
Spécifications Java

- Cf. discussions sur les listes : approche récursive/itérative, plusieurs spécifications possibles, sommet/arc courant, problèmes de symétrie des opérations, intérêt/danger des itérateurs multiples... Arbre (en entier) vs itérateur (sur un arc, entre 2 nœuds).

Arbre binaire itératif

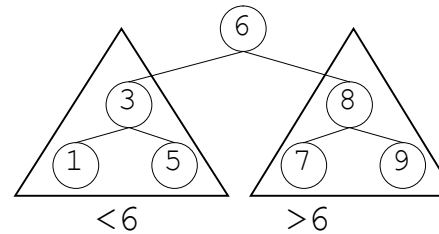
- Opérations toujours sur le "bas" de l'arc, l'insertion crée une feuille

- x=new BTree(); a=x.root();
 - a.insert(1)
 - a=a.left()
 - a.insert(2)
 - a=a.up()
 - a.consult()==1
 - a=a.right()
 - a.insert(3)
 - y=a.cut();
- 

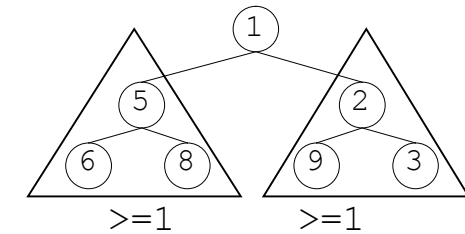


Domaines d'application (très nombreux !)

- File system
document balisés (HTML)
ensembles d'éléments triés
file de priorité
...



binary search tree

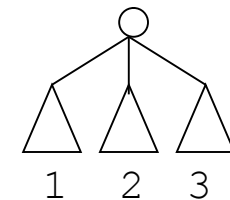
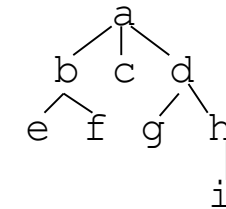


heap

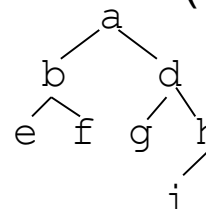
- Un exemple de type abstrait "arbre quelconque" : librairie DOM (Document Object Model) pour les arbres XML/HTML

Parcours d'arbres

- Le parcours d'arbre est une opération de base pour plusieurs algos
- Comment "linéariser" un arbre ? Deux approches principales
- *En profondeur d'abord* : "traiter tout un sous-arbre avant le suivant"
- Mais quand faut-il traiter le parent ? 3 variantes :



- *avant* les sous-arbres : pré-ordre a-b-e-f-c-d-g-h-i
- *après* les sous-arbres : post-ordre e-f-b-c-g-i-h-d-a
- *entre* les deux sous-arbres : in-ordre (seulement pour arbre binaire !)



e-b-f-a-g-d-i-h

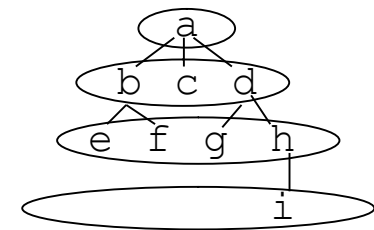
Pseudo-code récursif :

```
void depthFirst(Node root) {  
    Node n;  
    if (root == null) return;  
    processPreOrder(root.elc);  
    for each son n of root  
        depthFirst(n);  
    processPostOrder(root.elc);  
}
```

- *En largeur d'abord :*

- "traiter niveau par niveau"

a-b-c-d-e-f-g-h-i



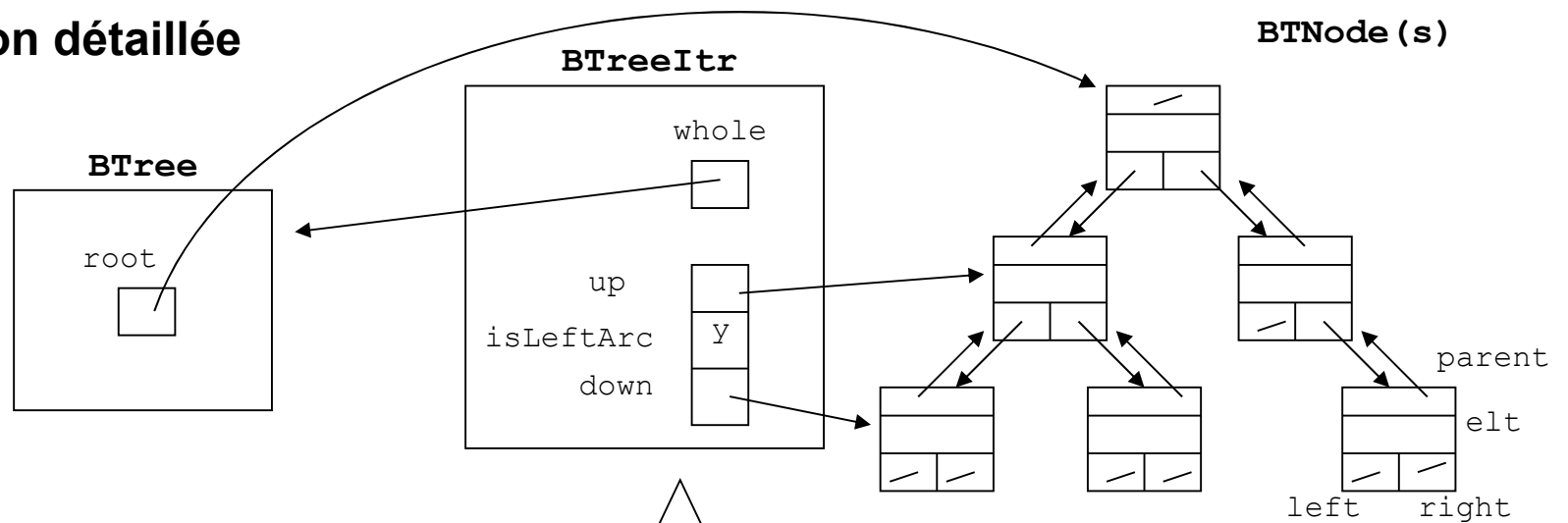
Pseudo-code itératif, utilisation d'une file FIFO

```
void breadthFirst(Node root) {  
    Node crt, n; Queue q = new Queue();  
    q.enqueue(root);  
    while(! q.isEmpty())  
        crt = q.dequeue();  
        process(crt.elc);  
        for each son n of crt  
            q.enqueue(n);  
}
```

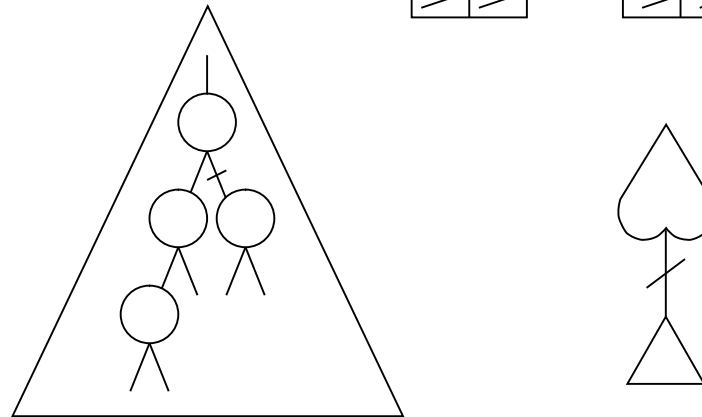
- Evidemment, on peut aussi coder pour parcourir "à l'envers", de droite à gauche

Type abstrait BTree, avec arc courant

Représentation détaillée



Représentations abstraites

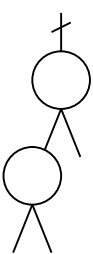


Cas particuliers

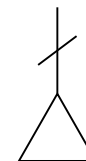
`isEmpty()`



`isRoot()`



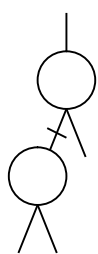
`isRoot()`



`isEmpty()`



`isLeafNode()`



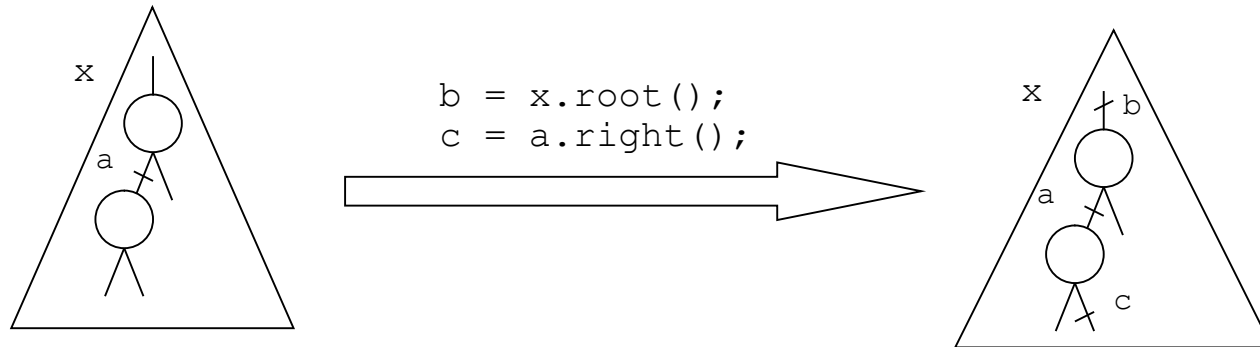
`isBottom()`



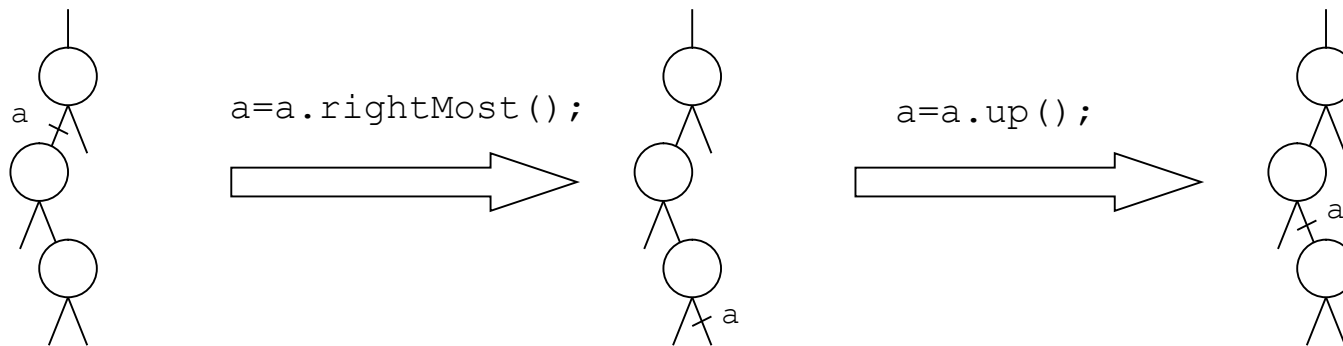
`isBottom()`



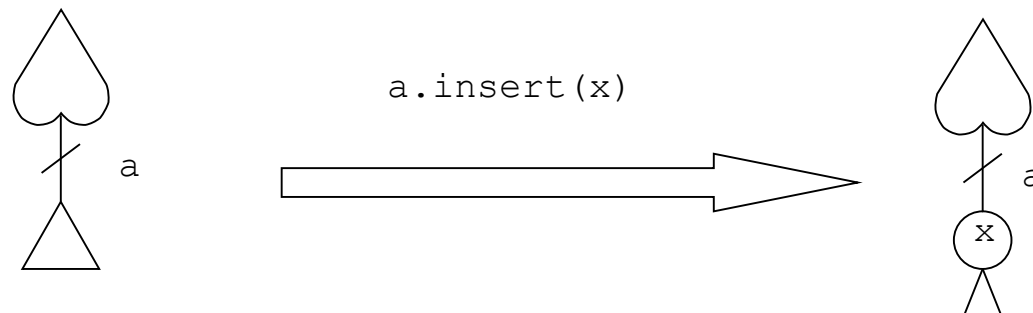
Arbre (dans sa totalité) et itérateurs (positions précises)



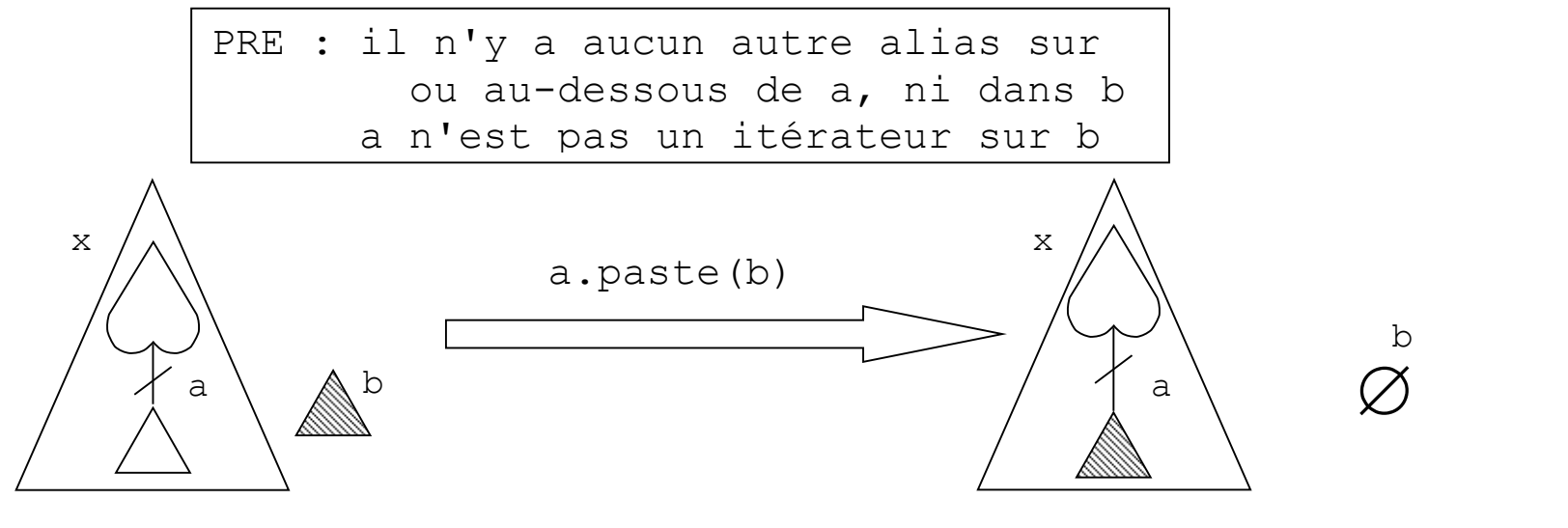
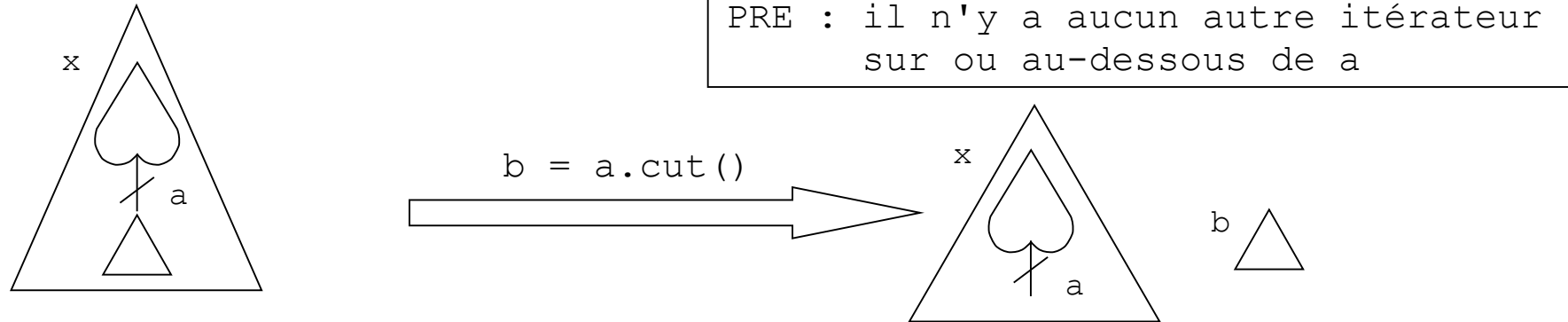
Accès aux positions voisines


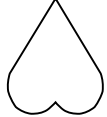


Insertion



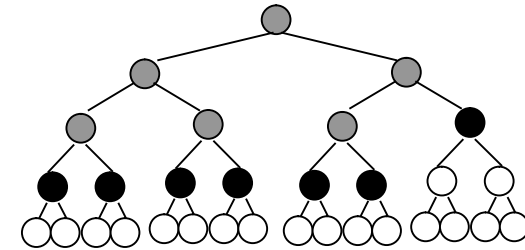
Spécification du cut/paste



Ces spécifications s'appliquent aussi lorsque  ou  sont vides, c.-à-d. lorsque `a.isRoot()`, `a.isBottom()` ou `b.isEmpty()`

Parcours en largeur avec file

```
static void breadthFirst (BTree t) {  
    BTreeItr crt=t.root();  
    Queue q = new Queue();  
    q.enqueue(crt);  
    while(! q.isEmpty()) {  
        crt = (BTreeItr) (q.dequeue());  
        if (crt.isBottom()) continue;  
        System.out.print(" "+crt.consult());  
        q.enqueue(crt.left ());  
        q.enqueue(crt.right());  
    }  
}
```



Parcours en profondeur préordre récursif

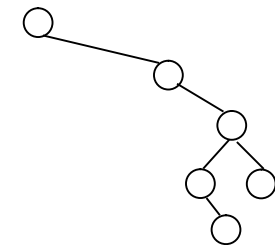
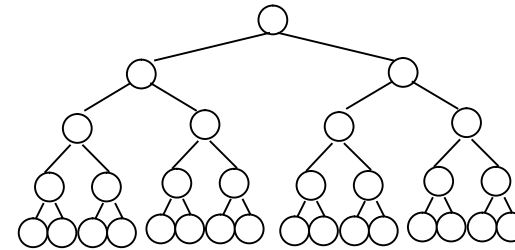
```
static void depthFirst (BTreeItr ti) {  
    if (ti.isBottom()) return;  
    System.out.print(" " + ti.consult());  
    depthFirst(ti.left ());  
    depthFirst(ti.right());  
}
```

- On suppose qu'on dispose d'une *méthode d'amorce* :

```
public static void depthFirstAll (BTree t) {  
    depthFirst(t.root());  
}
```


Arbres et complexité

- Considérons un arbre binaire à H niveaux complètement rempli
- Nombre d'éléments au niveau $i = 2^i$
- Nombre total d'éléments $N = 2^{H-1} + 2^{H-2} + \dots + 1 = 2^H - 1$
- Donc autant d'éléments dans le dernier niveau que dans tout le reste de cet arbre !
- La hauteur est proportionnelle au logarithme du nombre d'éléments :
 $H = \log_2(N+1)$ H est en $O(\ln n)$
- Plus difficile à prouver : si on énumère toutes les combinaisons d'arbres binaires à N éléments, la hauteur moyenne reste en $O(\ln n)$
(on a en moyenne une hauteur double de l'optimum)
- Généralement, l'utilisation d'arbres n'a d'intérêt que pour autant que l'arbre ne soit pas *dégénéré*. La plupart des traitements ont une complexité proportionnelle à la hauteur de l'arbre.

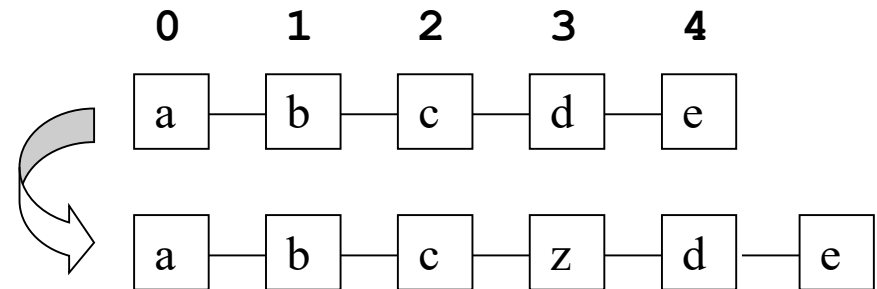


Java Collections : l'exemple ArrayList

- La classe `ArrayList<E>` simule un type abstrait *liste avec positions absolues*, représenté par un tableau avec redimensionnement automatique

```
ArrayList<String> v = new ArrayList<String>();  
v.add("a");           insertion at the end  
v.add(3, "b");        (*) insertion at index 3  
v.set(4, "c");        update at index 4  
z = v.get(3);         consult at index 3  
v.remove(4);          (*) remove at index 4  
v.remove("a");        (*) remove first occurrence of "a"  
y = v.contains("b");  (*) test the presence of "b"  
w = v.indexOf("b");   (*) find the index of the first "b"  
for(String s:v) {...}  
Collections.sort(v);
```

`add(3, "z")`



- Les méthodes (*) sont en $O(n)$.
- Voir aussi `TreeSet`, `HashSet`, `PriorityQueue`, `ConcurrentHashMap`, `BlockingQueue`...
- "LinkedList: *All of the operations perform as could be expected for a doubly-linked list [...]*"

Complexité "amortie" (amortized complexity)

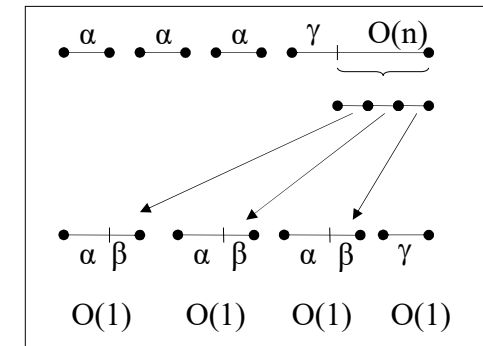
- On a parlé jusqu'ici de complexité d'un appel
- Parfois, on peut garantir une complexité seulement "sur le long terme"; p. ex. :
une séquence de M appels coûte $O(M)$

- Exemple : pile avec tableau, on double la taille quand c'est plein

Stack s = new Stack(3) tableau de 3 éléments

| | | |
|-------------|---------------------|-------------------------|
| s.push('a') | $O(1)$ | α [ns] |
| s.push('b') | $O(1)$ | α [ns] |
| s.push('c') | $O(1)$ | α [ns] |
| s.push('d') | $O(n)$ (array copy) | $\gamma + 3*\beta$ [ns] |
| s.push('e') | $O(1)$ | |

...



- Dans cet exemple, on a $O(n)$ seulement après avoir eu au moins n fois $O(1)$ avant !
On peut dire que ça compense ! En effet, si on calcule la
moyenne par opération : $(n * (\alpha) + 1*(\gamma+n*\beta)) / n = (\alpha + \beta) + (\gamma/n)$
- Analyse de complexité "amortie" : principe de l'épargne (boîte à bons points)
- A chaque appel "bon marché", on gagne un bon point
- Il faut *garantir* qu'au moment d'un appel "coûteux", on a assez de bons points
- Cf. [Weiss10] pour une discussion plus rigoureuse. Mathématiquement, ça peut devenir un peu ardu. Pas seulement valable avec $O(1)$ et $O(n)$

Complexité espérée (*expected complexity*)

- *Algo probabiliste* = qui prend certaines décisions de façon aléatoire
- La complexité dépend alors parfois de la séquence de nombres pseudo-aléatoires utilisés. Si le pire des cas *dépend des données*, c'est embêtant; s'il *dépend du hasard*, c'est mieux !

```
~if (anInputData==64)           ~if (randomInt()==64)
~  for(int i=0; i<n; i++) a++;    ~  for(int i=0; i<n; i++) a++;
```

- Complexité espérée : moyenne sur tous les tirages aléatoires possibles
- Le mieux, c'est de garantir une complexité au pire des cas.
Mais en pratique, la complexité espérée est très appréciable.

Ne pas confondre...

- "Complexité *amortie*" d'une suite d'appels (généralement au pire des cas)
- "Complexité *moyenne*" d'un appel; moyenne sur toutes les données possibles
(= cas moyen; parfois très difficile à définir clairement)
- "Complexité *espérée*" d'un appel; moyenne sur des tirages aléatoires
- "Pire des cas", "cas moyen", "meilleur des cas" :
suivant la configuration des données en entrée (input)
- Algorithmes "*output sensitive*" : la complexité dépend de la taille du résultat
Exemple : afficher les éléments <x présents dans un ensemble donné

Généricité avec Java 1.5

- *Type safety*. Faire détecter les incompatibilités de type par le compilateur.

Utilisation d'une classe générique (utilisez systématiquement ! **Jamais** de "raw type" !)

```
Stack<Integer> s;  
s = new Stack<Integer>();  
s.push(new Integer(43));  
s.push(44); // autoboxing;  
int x=s.pop(); // unboxing;
```

```
package java.util;  
public class Stack <E> {...  
    public Stack();  
    public void push (E elt);  
    public E pop ();  
}
```

Implémentation d'une classe générique

- Le type générique s'utilise (presque !) comme un type normal (et ne doit pas engendrer de warnings du compilateur).

```
public class MyStack <E> {  
    ArrayList<E> buf; int top = -1;  
    public MyStack() { buf = new ArrayList<E>(); }  
    public void push (E elt) { buf.add(elt); }  
    public E pop () { return buf.get(top--); }  
}
```

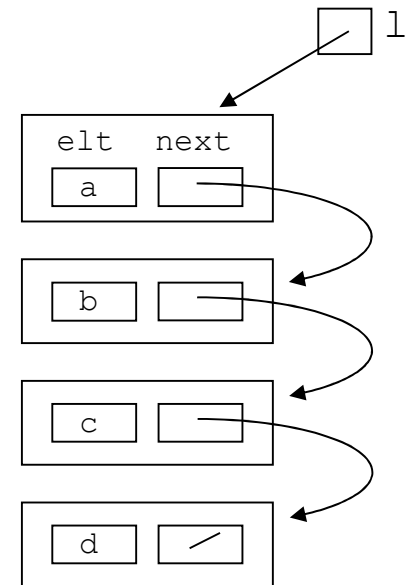
- *Type erasure* : le type générique existe à la compilation, pas à l'exécution. Certaines instructions n'ont pas de sens :
- Utilisation avancée (wildcards, subtyping, generic methods) : cf. "*Generics in the Java Programming Language*"

```
x = new E();  
x = (E) y;  
t = new E[10];  
if(x instanceof E)
```

Notion de pseudo-pointeurs

- Ok, il n'y a pas de vrais pointeurs en Java, mais des types références... Fonctionnellement, c'est la même chose, en plus sûr !
- Exemple de structure chaînée, et d'une opération possible :

```
class ListNode {  
    char elt;    ListNode next;  
}  
void append(ListNode l, char e) {  
    while (l.next != null)  
        l=l.next;  
    l.next = new ListNode(e, null);  
}
```



- On parle de pseudo-pointeurs quand un programme simule un allocateur de mémoire avec un tableau :
 - mémoire tableau
 - adresse indice
 - pointeurs pseudo-pointeur
 - attribution gérée par l'OS attribution gérée par le programme
- Inconvénient : il faut coder la gestion (`new()`, `destroy()`, ...); espace fixe
- Avantage : allocation/libération rapide (pas d'appels système)
- C'est une technique d'*optimisation de code* (on en verra d'autres plus tard)

- Une "gestion de mémoire" rudimentaire : maintenir une liste des cases "vides", garder un accès à la première case vide, et définir un pointeur nul (p. ex. -1)
- Voyons comment faire pour l'exemple ci-dessus de chaînage simple.
- Utilisation de tableaux parallèles, le chaînage ayant un double rôle.

```

final int NIL = -1;
char[] elts = new char[10];
int[] nexts = {1, 2, 3, ..., 8, 9, NIL};
int firstFreeCell = 0;

```

- Deux méthodes pour gérer l'allocation des cases :

```

// alloue une case et retourne son indice
int allocate()
    if(firstFreeCell == NIL) throw new Exception("no more memory");
    int i = firstFreeCell;
    firstFreeCell = nexts[i];  nexts[i] = NIL;
    return i;

```

```

// restitue au système la case d'indice i
void deallocate(int i) {
    nexts[i] = firstFreeCell;  // and maybe: elts[i]=null;
    firstFreeCell = i;
}

```

- Plusieurs listes peuvent se partager notre tableau-mémoire
- Exemples d'opérations sur des listes

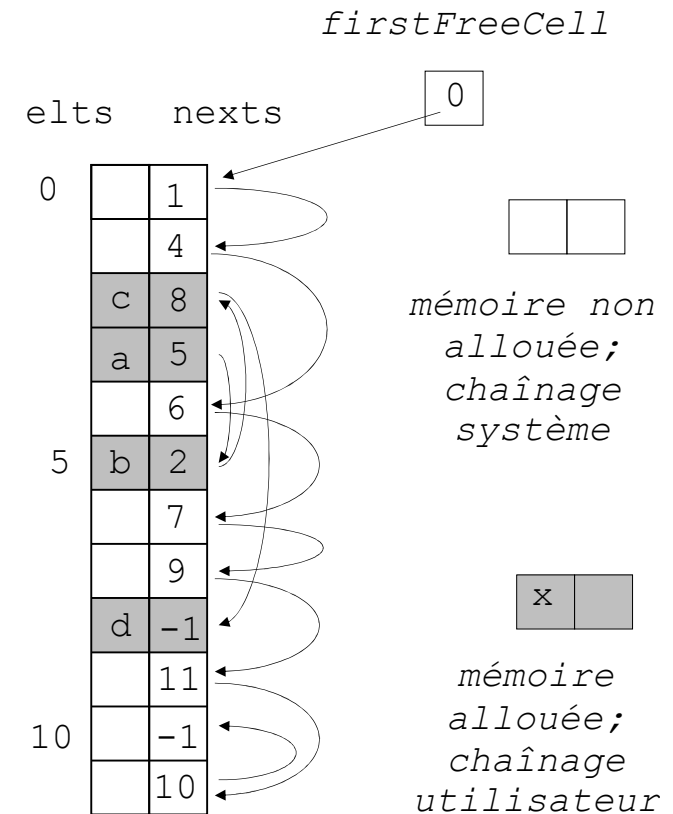
```

int createList(char firstElt) {
    int n=allocate();
    elts[n] = firstElt;
    nexts[n] = NIL;
    return n;
}

void append(int list, char e) {
    while (nexts[list] != NIL)
        list = nexts[list];
    int n=allocate();
    elts[n] = e; nexts[n] = NIL;
    nexts[list] = n;
}

void remove(int list, char e) {
    while (elts[nexts[list]] != e)
        list=nexts[list];
    int deletedNode = nexts[list];
    nexts[list] = nexts[deletedNode];
    deallocate(deletedNode);
}

```

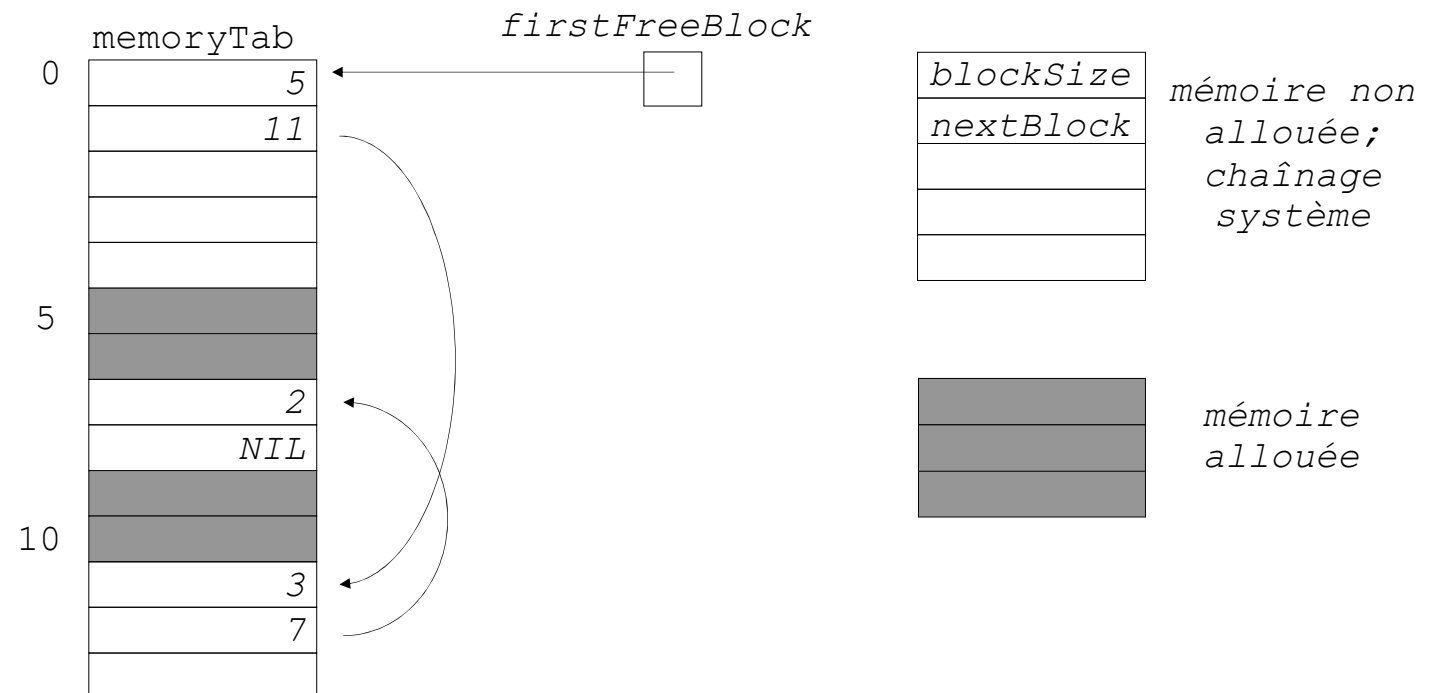


Vers un gestionnaire de mémoire plus général

- En poussant le concept, on reconstruit un vrai "memory manager"
- Mémoire : gros tableau de bytes (ou d'entiers), sans support de typage
- Primitives :

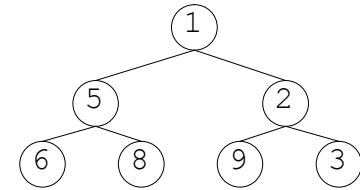
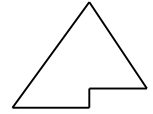
```
int    allocate(int nbOfCells);  
void deallocate(int blockPtr, int nbOfCells);
```

- Chaînage des zones libres (non allouées) (*free list*)
- Mémorisation des tailles des blocs libres
- Stratégie d'allocation des blocs (*first fit, best fit*), problème de fragmentation



Heap (= tas = monceau = maximier)

- Arbre binaire complet : tous les niveaux sont remplis, sauf éventuellement les derniers nœuds du dernier niveau
- Arbre binaire absolument complet : le dernier niveau est aussi rempli (quasi-complet : les nœuds du dernier niveau sont éparpillés)
- Propriété du tas : arbre partiellement ordonné :
 - sommet \leq chacun de ses fils
 - donc, chaque chemin racine-feuille est trié
 - donc, le minimum est à la racine
 - mais rien n'est garanti entre 2 frères

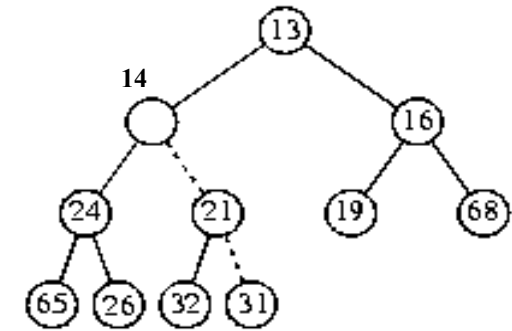
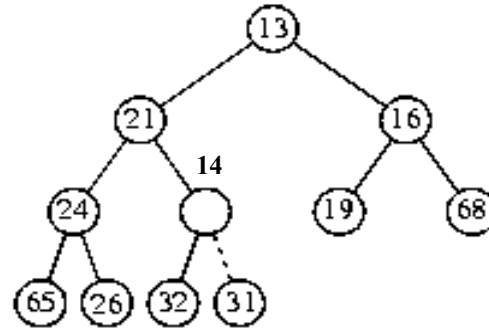
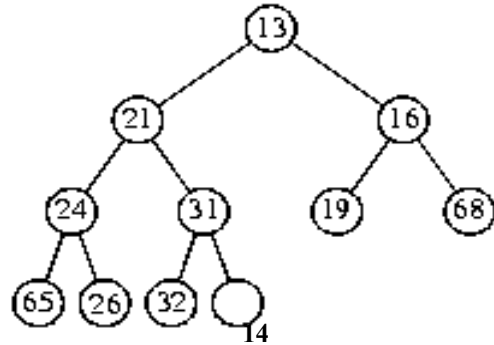


Tas complet : une implémentation efficace de la file de priorité

- Opération d'ajout :
 - ajouter l'élément à la prochaine case libre (après la dernière feuille)
 - le faire remonter (percoler) tant qu'il viole la contrainte
- Opération de retrait du minimum :
 - remplacer la racine par la dernière case occupée
 - le faire descendre (tamiser) par le plus petit côté tant qu'il viole la contrainte
- On peut transformer un arbre complet en Heap en $O(n)$ (cf. [Weiss10])
Le tas est donc aussi à la base d'un bon algorithme de tri
- Implémentation : représentation par *chaînage implicite* (dans un tableau)

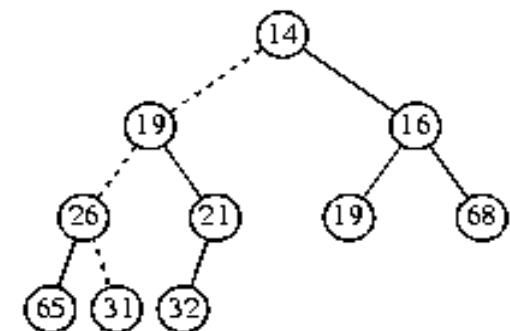
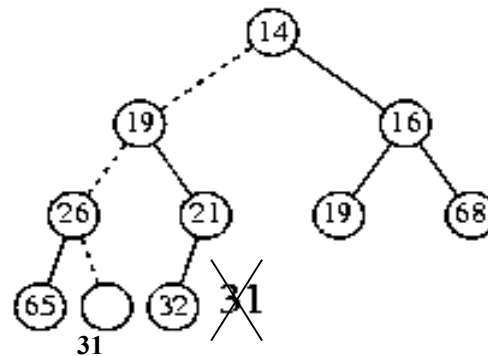
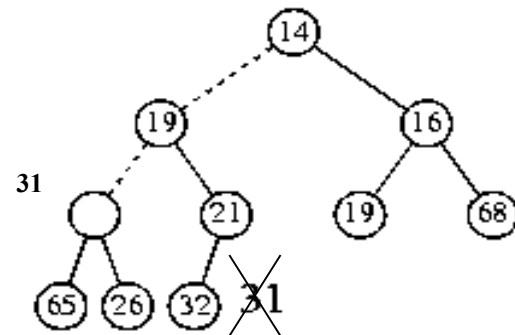
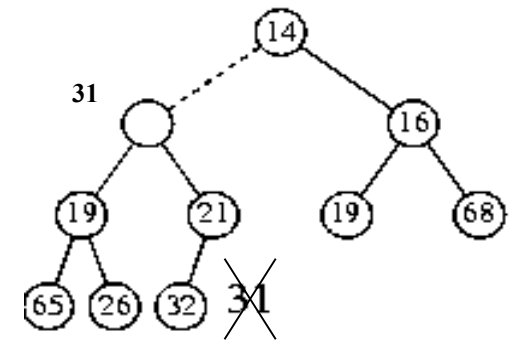
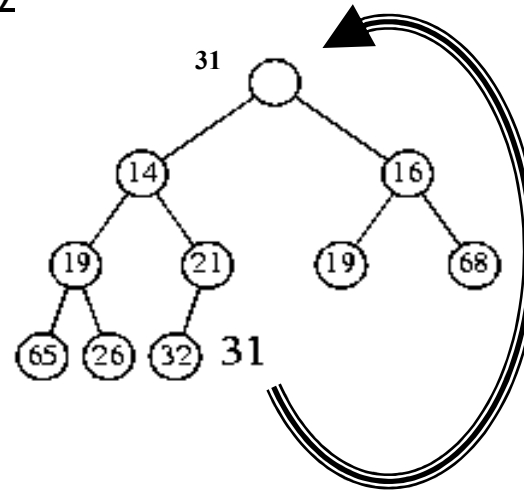
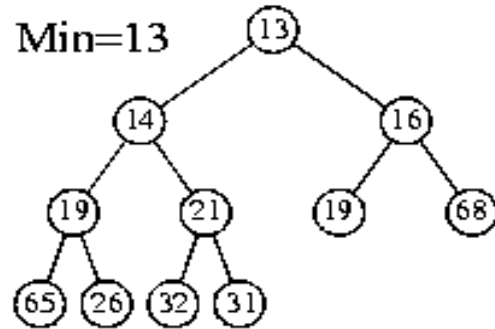
Exemple

add(14)



removeMin()

Min=13



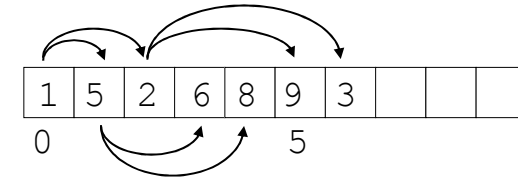
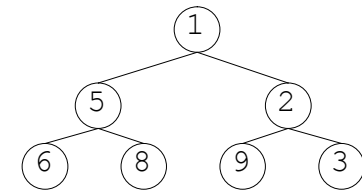
Codage

- Dans [Weiss98] : sentinelle "au-dessus" de la racine (bof !)

- Attributs : `double[] elements; int size;`

- Méthodes privées :

```
int parent (int nodeIndex);  
int leftChild (int nodeIndex);  
int rightChild (int nodeIndex);  
void percolateUp(int nodeIndex);  
void siftDown (int nodeIndex);  
void checkSize (); // expand array if needed
```



Tas et files de priorité

- Principe : créer un tas de couples (elt,pty) avec $(e1,p1) < (e2,p2)$ ssi $p1 < p2$

- Réalisation :

```
HeapElt[] heap;  
class HeapElt {Object e; double pty;}
```

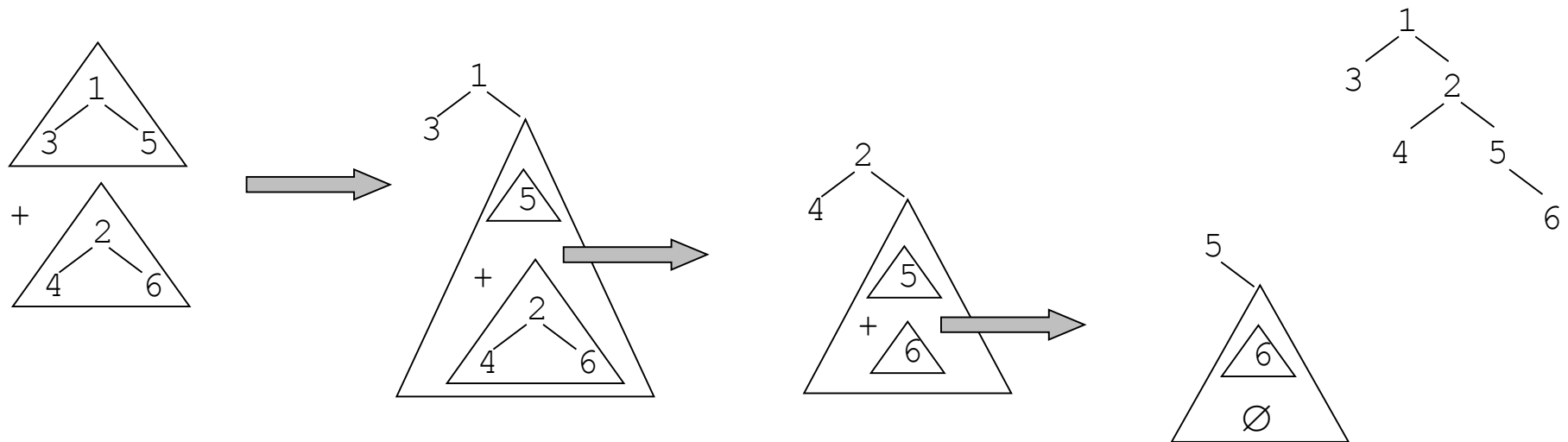
Ou alors, avec des tableaux parallèles :

```
double[] ptyHeap; Object[] eltHeap;
```

Files de priorité fusionnables

- Le tas ne peut simuler l'opération de fusion qu'en $O(n)$
- Opération de mise à jour de la priorité d'un élément aussi en $O(n)$: il faut d'abord trouver le nœud contenant l'élément !

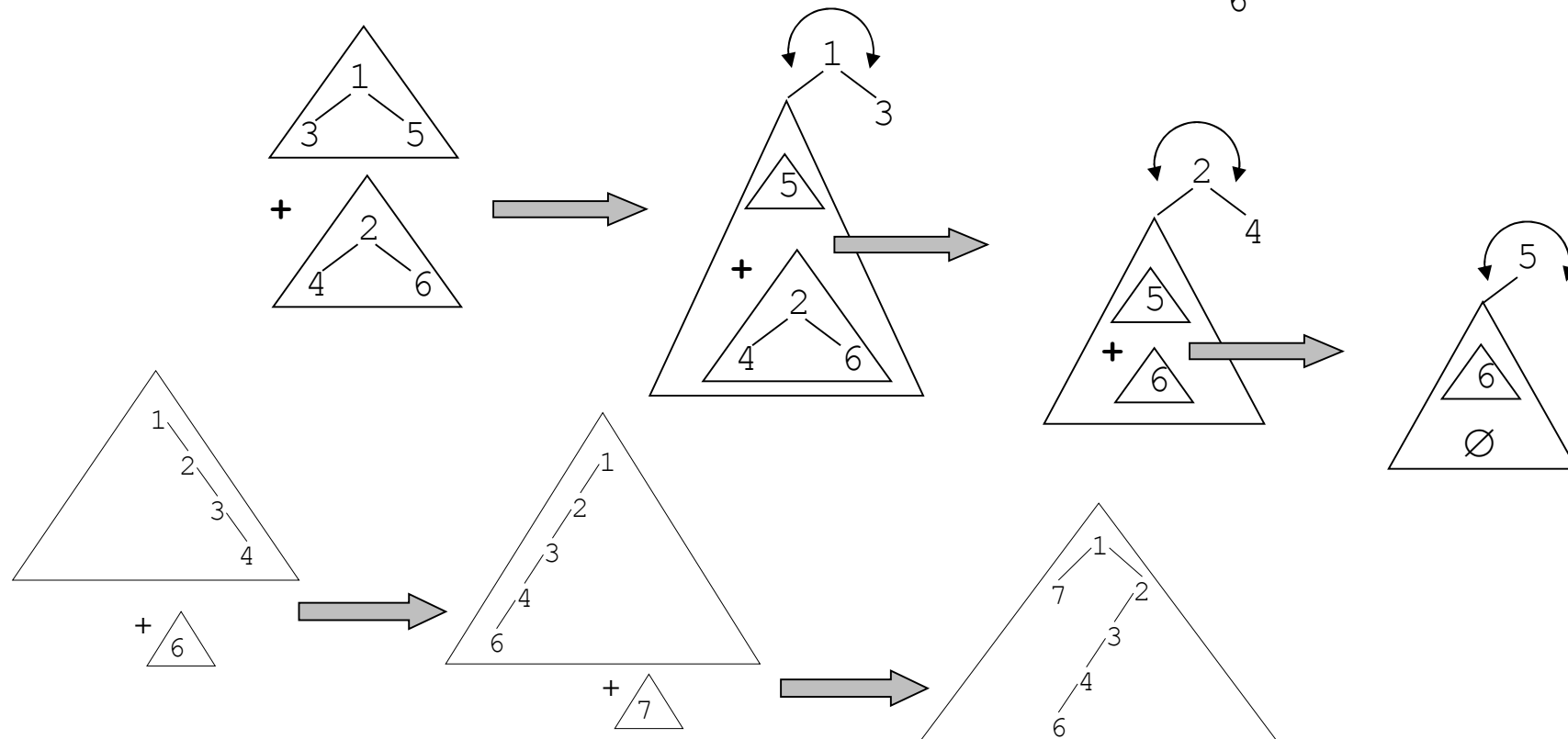
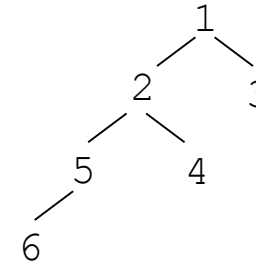
- Changeons de cap : l'opération de fusion est à la base de toutes les autres !
 - ajout : créer un arbre-racine et fusionner
 - retrait : fusionner le sous-arbre gauche et le sous-arbre droit
 - mise à jour : trouver le nœud de l'élément; couper ce sous-arbre; enlever le minimum; ajouter l'élément avec sa nouvelle priorité; fusionner les 2 arbres
- Maintenant, on prend un vrai chaînage au lieu d'une représentation implicite
Moins de contraintes sur la structure de l'arbre : le tas n'est plus forcément complet (mais on aimerait bien éviter les cas dégénérés)
- Algo naïf (et récursif) pour la fusion de 2 tas (en $O(\text{somme des hauteurs})$) :
 - si l'un des 2 est vide, retourner l'autre. Sinon :
 - soit P celui qui a la racine la plus faible, et G l'autre
 - fusionner G dans le sous-arbre droit de P (le gauche irait aussi !)



- Problème : arbre très déséquilibré, car on ne crée que des chemins à droite

- Variante "**skew heaps**" :

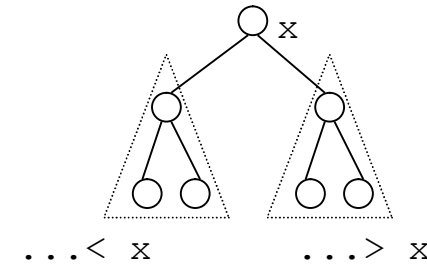
- si l'un des 2 est vide, retourner l'autre. Sinon :
- soit P celui qui a la racine la plus faible, et G l'autre
- fusionner G dans le sous-arbre droit de P
- *échanger les sous-arbres de P*



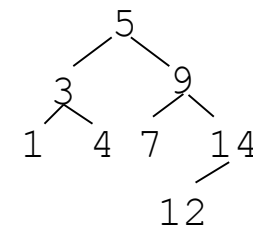
- Garantie de complexité amortie $O(\ln n)$. Preuve ardue (cf [Weiss10])
- (Optimisation de code : on peut toujours éliminer la récursivité...)
- Autre variante : "pairing heaps" (arbres quelconques, et non plus binaires)

Arbre de fouille binaire (Binary Search Tree)

- Alternative aux tables de hachage
- Ensemble d'éléments supportant une relation d'ordre : $x < y$
- Arbre binaire avec contraintes locales sur chaque sommet :
 - éléments inférieurs dans le sous-arbre gauche
 - éléments supérieurs dans le sous-arbre droit



- Parcourir "dans l'ordre" = parcours en profondeur in-ordre
- Permet d'implémenter les types abstraits Ensemble et Map
- Trouver un élément : descendre dans l'arbre, selon la comparaison
- Ajout d'un élément : descendre dans l'arbre et ajouter une feuille
- Retrait d'un élément :
 - si c'était une feuille, c'est facile
 - s'il n'a qu'un sous-arbre, on "remonte" celui-ci
 - sinon, le remplacer par le plus petit (grand) élément du sous-arbre droit (gauche)
- Nombreux arbres de fouille possibles pour une même collection d'éléments
- Accès à un élément : dépend de la hauteur de l'arbre
- Généralement, on interdit les éléments strictement égaux ($x=y$). Au besoin (pour le multi-ensemble), on traite le cas différemment...



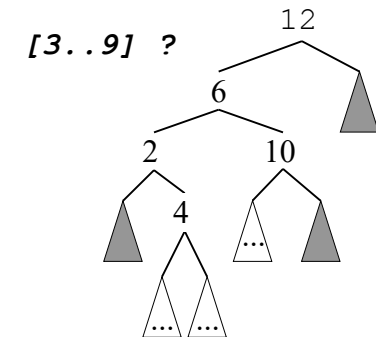
Recherche du k^{ème} élément (entre 1 et n)

- Efficace, si on stocke à chaque nœud la taille du sous-arbre (on parle de structure de données *augmentée*, au sens où on ajoute d'autres attributs à la structure de base)
- Pseudo-code :

```
int findKth(Tree t, int k) {  
    if (t.isEmpty()) throw new Exception();  
    if (k == 1+sizeof(t.left)) return t.rootElt;  
    if (k < 1+sizeof(t.left)) return findKth(t.left, k);  
    return findKth(t.right, k-1-typeof(t.left));  
}
```


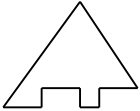
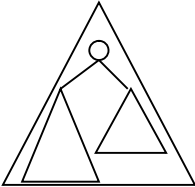
Recherche par intervalle

- Trouver tous les éléments compris dans un certain intervalle [a, b]
- Algo naïf : un parcours complet et on teste chaque élément pour voir s'il est concerné
- Meilleur algo : variante du parcours in-ordre :
si l'élément courant est plus grand que la borne supérieure,
alors pas besoin de visiter le sous-arbre droit ! (idem à gauche)



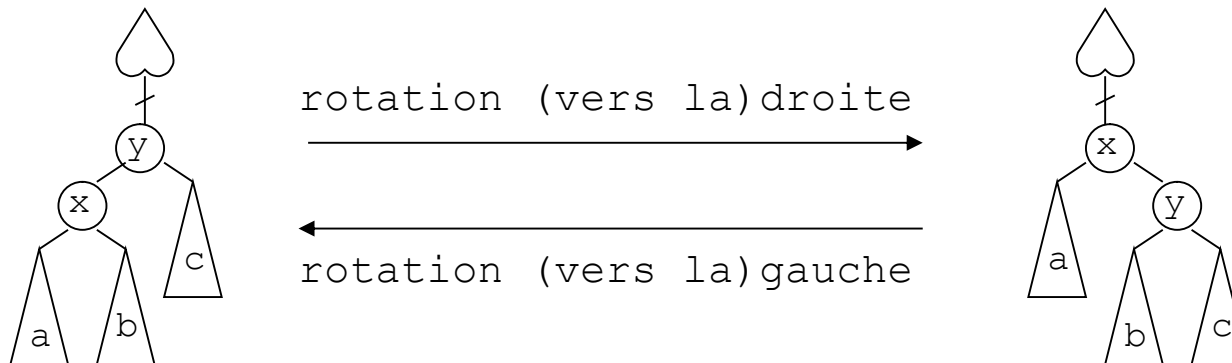
- Algorithme "output-sensitive" : la complexité dépend aussi des résultats trouvés

Arbres de fouille binaires équilibrés (*balanced*)

- Comment éviter les arbres dégénérés, et garantir la hauteur $O(\ln n)$?
- Politiques de rééquilibrage :
 - global : laisser dégénérer, et reconstruire un arbre idéal de temps en temps
 - local : transformer légèrement l'arbre après chaque accès
- Il faut parfois maintenir dans les nœuds des informations utiles à l'équilibrage
- Ajouter des contraintes sur l'arbre de fouille, pour le maintenir équilibré
 - équilibre absol. parfait : arbre absol. complet 
 - équilibre parfait : 
 - équilibre AVL : $|H(\text{left}) - H(\text{right})| \leq 1$ 
 - équilibre Red-Black : hauteur $\leq 2 * (\text{hauteur idéale})$
- Autres variantes : arbres rouge/noir, 2-3-4, AA, splay, treaps, B-arbres...

Opération de rotation (rotation simple)

- Opération de base pour restructurer un arbre de fouille (codage : cf. exercices)

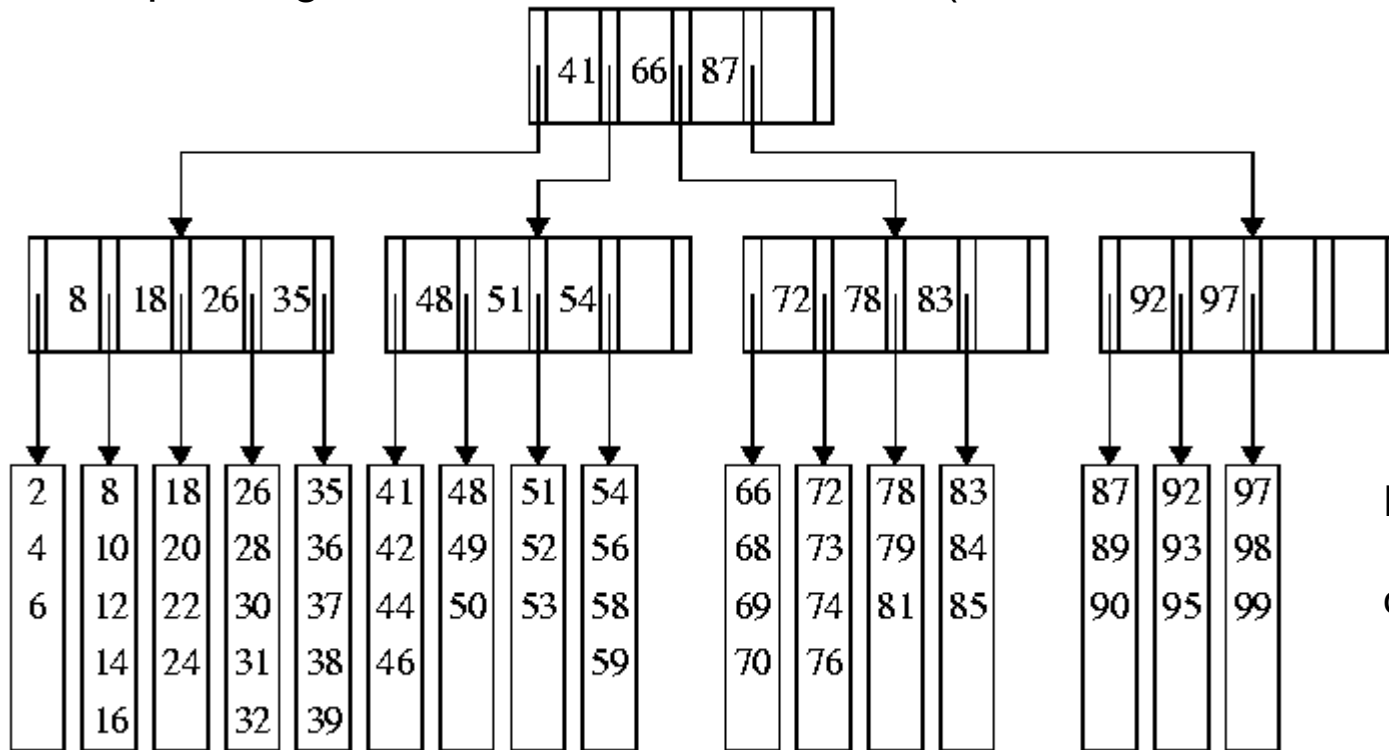


Ces deux opérations suffisent pour engendrer toutes les formes possibles d'un arbre de fouille.

- Le package `java.util` contient `HashSet`, `HashMap`, et aussi `TreeSet` et `TreeMap` :
 - " Red-Black tree based implementation of the SortedMap interface. This class guarantees that the map will be in ascending key order [...] (see Comparable [...]). This implementation provides guaranteed $\log(n)$ time cost for the containsKey, get, put and remove operations. [...]"*

Données sur fichier et B-arbres

- Comment gérer un dictionnaire volumineux sur disque (base de données) ?
- Première idée : un BST avec un "fichier" par nœud (référence=n° disc block)
Beaucoup trop d'accès disques (très très lents)
- Principe : augmenter les embranchements (arbre M-aire de recherche)



Dans cette variante,
les feuilles
contiennent tous les
éléments

- Reste à balancer tout ça :
 - en hauteur
 - en "taux d'occupation" des nœuds
 - et sans jamais trop bouleverser !

- Un B-arbre est un arbre de fouille M-aire où :
 - la racine ne peut pas avoir de fils unique, les autres nœuds ont plus de $M/2$ fils
 - les éléments sont stockés aux feuilles; les nœuds internes n'ont que des copies
 - toutes les feuilles ont la même profondeur

- Algo de recherche : rien de spécial, on descend dans l'arbre

- Algo d'insertion à la feuille :

```

~si le noeud n'est pas plein
~  insérer correctement l'élément
~sinon
~  si le noeud frère (gauche ou droit) n'est pas au maximum
~    lui refiler un élément (le min ou le max)
~  sinon
~    diviser le noeud en 2, répartir les élts
~  adapter le parent (évt. un fils de plus → récursif)

```

} *facultatif*

- Algo de retrait à la feuille :

```

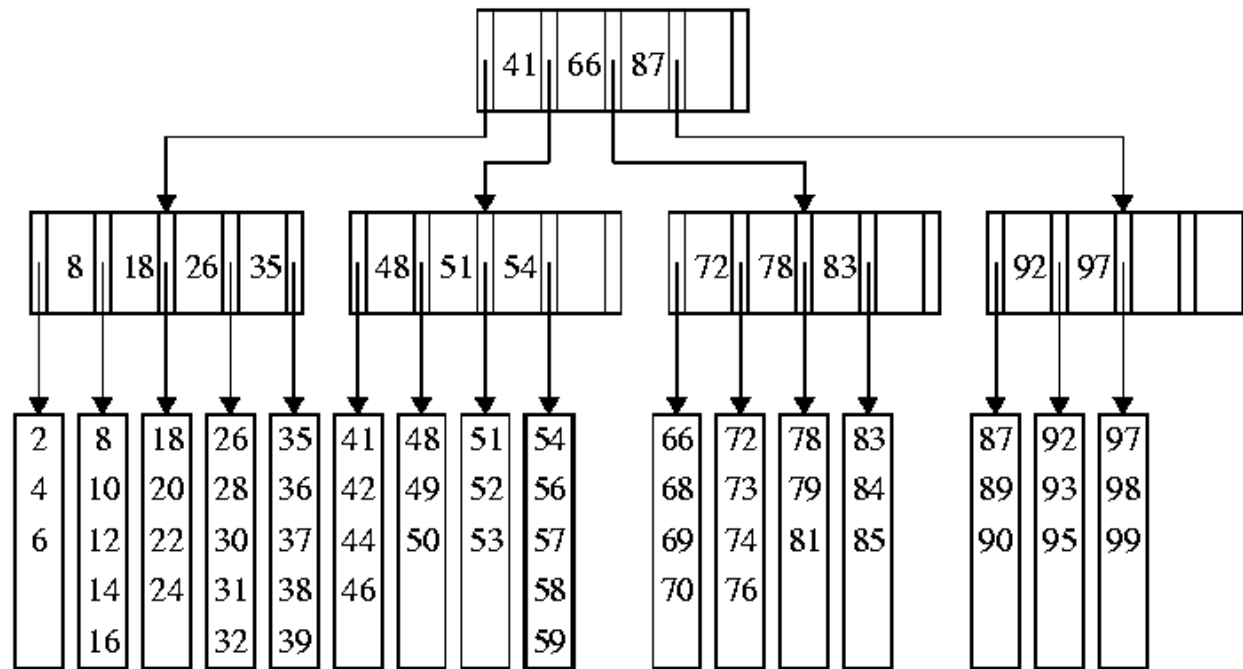
~si le noeud n'était pas au minimum
~  supprimer correctement l'élément
~sinon
~  si le noeud frère (gauche ou droit) n'est pas au minimum
~    lui voler un élément (le min ou le max)
~  sinon
~    fusionner ces 2 noeuds (grouper les élts)
~  adapter le parent (évt. un fils de moins → récursif)

```

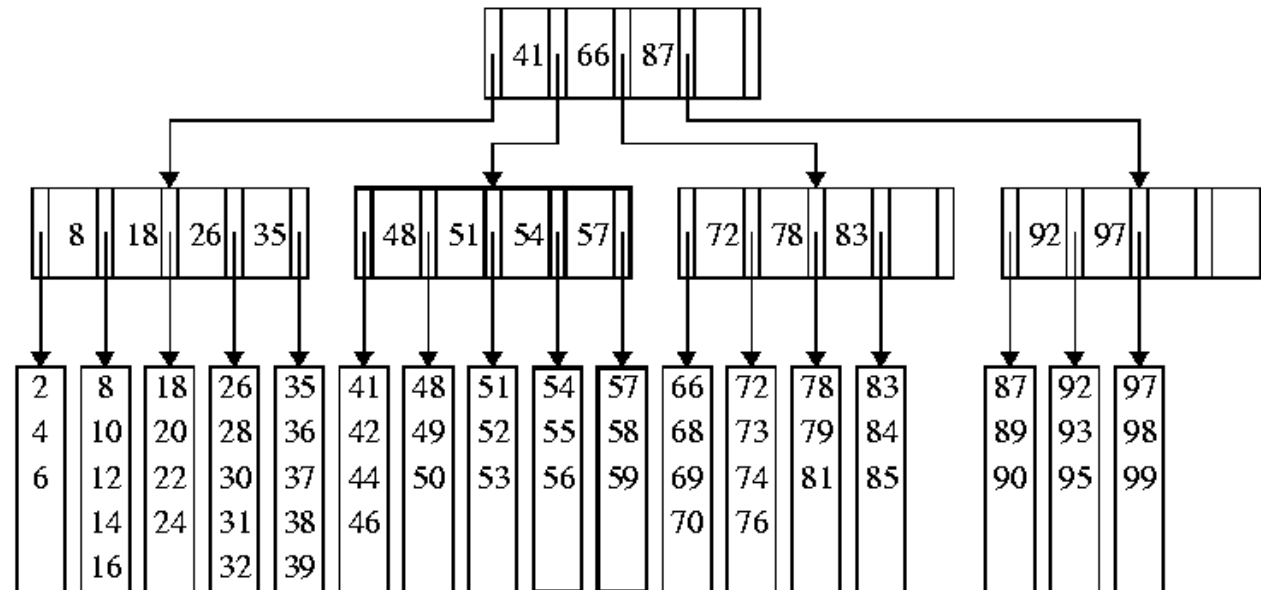
- Il y a plusieurs variantes de B-arbres. Le codage est un peu ardu...

Example

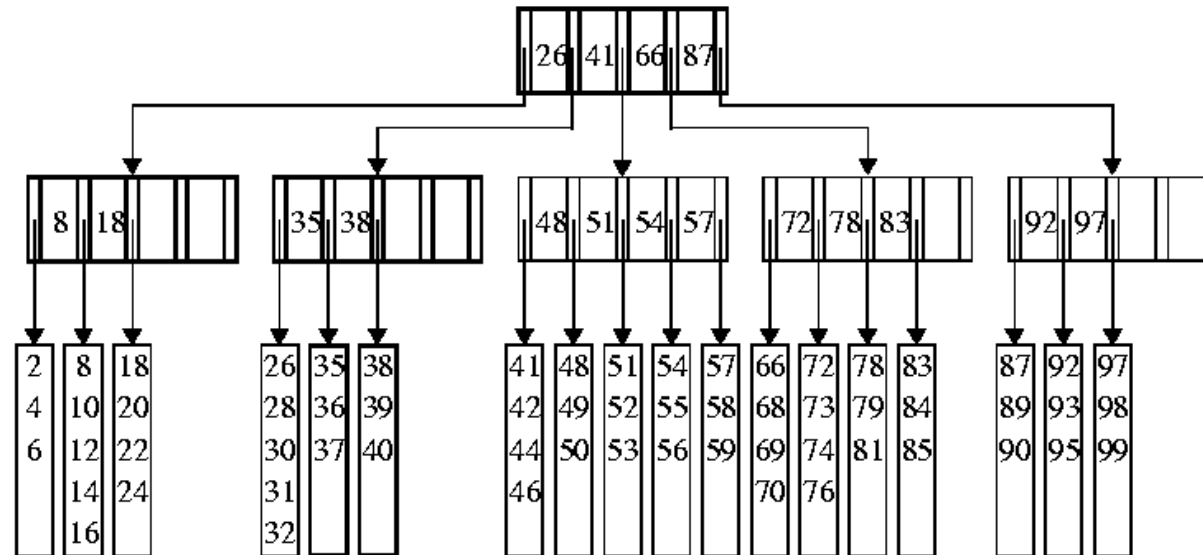
- add(57);



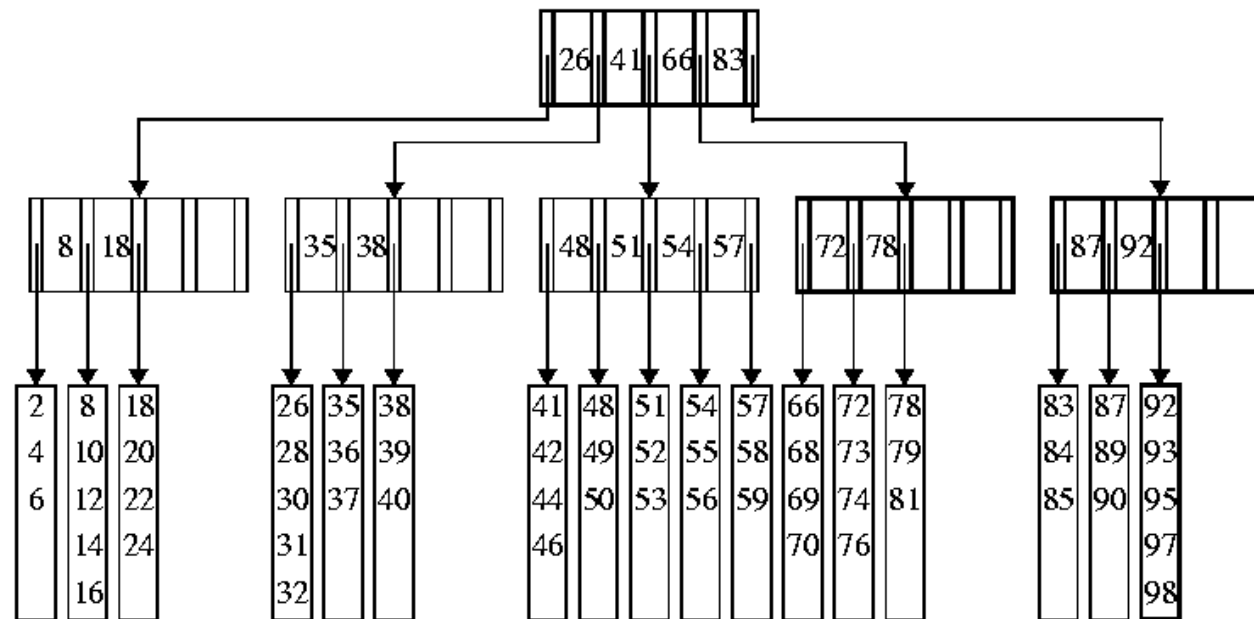
- add(55);



- add(40);



- remove(99)



Techniques de simulation

- Simulation : expérience imitant le comportement d'un phénomène
- Notion de *modèle* : représentation simplifiée de la réalité
- Il existe des logiciels complets qui facilitent la mise en place d'une simulation (Omnet++, Desmo-J...). Il existe des cours entiers sur la simulation

Simulation discrète (*discrete event simulation*)

- On suppose que le système évolue comme une suite d'événements discrets :
 - des événements sont datés (ils surviennent à des instants précis)
 - entre 2 événements successifs, le monde ne varie pas

Simulation continue

- On décrit l'évolution par des lois continues (le monde change en permanence)
 - systèmes hydrauliques (flux, vannes, écoulements)
 - populations biologiques (maladies, proies/prédateurs)
 - cela revient souvent à résoudre des équations différentielles

Autres situations d'analyse de comportement

- Systèmes temps réel, résolution de problèmes distribués, systèmes multi-agents, robotique, programmation concurrente, multi-thread en Java, etc.
- On parle parfois aussi de simulation dans ce contexte...

Simulation discrète

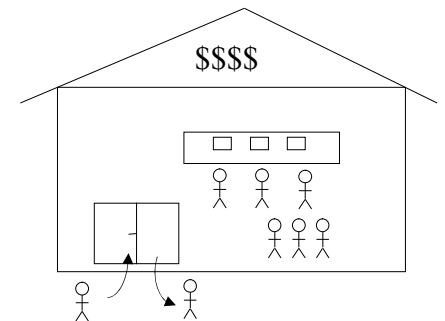
- Soit une banque avec K guichets, et des clients qui arrivent :
 - les clients arrivent au fil du temps (loi de probabilité)
 - chaque transaction demande une certaine durée (loi de probabilité)
 - si aucun guichet n'est libre, le client se met dans la file d'attente (variante : il part, de rage)
 - quel est p. ex. le taux d'occupation moyen des guichets ?

- On simule une "horloge logique" (ticks, minutes, années...)

- Algo de simulation discrète par événement. Pseudo-code :

```
« Construire le "monde"
« Placer des événements initiaux
« Tant qu'on veut (durée max de la simulation) {
«   Prendre le prochain événement e
«   Traiter e : avancer l'horloge à l'instant de e
«               modifier le monde,
«               générer de futurs événements
« }
« Retourner des statistiques sur le déroulement
```

- Comment gérer l'*agenda* des futurs événements ? En file de priorité !
- Comment programmer l'arrivée des clients (instants 4, 9, 10, 10, 22, ...) ?
Chaque arrivée "déclenche" la prochaine; tirer au sort *le délai jusqu'au prochain*



Exemple de la banque

- Monde :
 - un ensemble de K guichets (initialement libres)
 - des clients numérotés (mémoriser le dernier, pour produire le suivant)
 - une file d'attente de clients (initialement vide)
- Types d'événements (datés) :
 - arrivée d'un client *i* à la banque
 - départ d'un client *i* d'un guichet
- Paramètres
 - nombre K de guichets
 - durée moyenne d'une transaction (et une loi de probabilité)
 - durée moyenne entre 2 arrivées (loi de Poisson et loi exponentielle)
- Variantes : plusieurs types de guichets, de transactions, ou de clients, ...

Implémentation

```
class SimulEvent { long    time;  
                  int     who;          // client id  
                  boolean isArrival;  
}  
long    crtTime;  
PtyQueue events;          // elts: SimulEvents, pty: time  
Queue    clientQueue;    // elts: client ids  
int      lastClient, nbOfTellers, freeTellers;
```

```

void handleEvent(SimulEvent e) {
    crtTime= e.time;
    if (e.isArrival) handleArrival(e) else handleDeparture(e);
}

void handleArrival(SimulEvent e) {
    if (freeTellers==0) clientQueue.enqueue(e.who);
    else
        handleStartTransaction(e.who);
    tirer au sort une durée d;
    lastClient++;
    poster l'événement "arrivée de lastClient à crtTime+d"
}

void handleDeparture(SimulEvent e) {
    freeTellers++;
    if (clientQueue.isEmpty()) return;
    c = clientQueue.dequeue();
    handleStartTransaction(c);
}

void handleStartTransaction(int clientId) {
    freeTellers--;
    tirer au sort une durée de transaction d
    poster l'événement "départ de clientNb à crtTime+d"
}

```

- **Manque** : "mettre à jour les statistiques", aux bons endroits !

- Représentation des données s'il y a plus de 2 types d'événements :

(1) représenter chaque type par une constante entière :

```

static final int ARRIVAL    = 1;
static final int DEPARTURE = 2;
int kind = ARRIVAL;
if (kind == DEPARTURE) ...

```

(2) créer des classes dérivées de SimulEvent :

```

class ArrivalEvt    extends SimulEvent {...}
class DepartureEvt extends SimulEvent {...}
SimulEvent e = new ArrivalEvt(...)
if (e instanceof DepartureEvt ) ...

```

(3) s'approcher de vrais *types énumérés* (*type safe*):

```

class EvtKind {
    public static final EvtKind ARRIVAL    = new EvtKind();
    public static final EvtKind DEPARTURE = new EvtKind();
    private EvtKind() {} // private constructor !!!
}
EvtKind kind = EvtKind.ARRIVAL;
if (kind == EvtKind.DEPARTURE) ...

```

- Cette dernière approche est aussi parfois utilisée lorsqu'on cherche à représenter "l'absence d'objet" ... par un objet quand même !

```

public static final Object MY_NULL = new Object();

```

Types énumérés en Java 1.5

- Technique (3), mais mieux "emballée" !
- Possibilité d'associer des attributs et méthodes

```
void g(Answer f) {  
    switch (f) {  
        case YES:      ...  
        case NO:       ...  
        case MAYBE:    ...  
    }  
}
```

```
public enum Answer {  
    YES, NO, MAYBE;  
}
```

```
public enum Month {  
    JANUARY (31),  
    FEBRUARY (28) ... ;  
    public final int nbOfDays;  
    Month(int d) {nbOfDays=d; }  
    public boolean isLong() {  
        return nbOfDays==31;  
    }  
}
```

Boucles for-each en Java 1.5

- Simplifie le parcours usuel des tableaux, collections, ou énumérations, en cachant le compteur ou l'itérateur
- Les boucles normales restent indispensables pour des parcours plus riches, ou si on doit utiliser le compteur/itérateur

```
void f(int[] t) {  
    for(int e : t)  
        print(e);  
}
```

```
for(Answer a : Answer.values())  
    print(a);
```

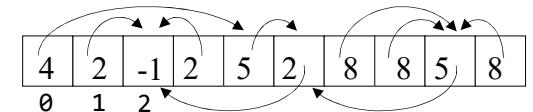
Affichage formaté en Java 1.5

- Très proche des sorties formatées de C (printf). Traite aussi des dates...

```
String s = String.format("v with 2 digits %.2f", aFloat);  
System.out.printf("decimal %d, octal %o", 21, 22);
```

Ensembles disjoints (*quick-union*) ("une dizaine de lignes de code")

- Vu : plusieurs réalisations du type ensemble (BitSet, hachage, arbre de fouille)
- Un point faible : opération d'union en $O(n)$
- Type abstrait "ensembles disjoints" ("union-find") :
 - les éléments sont tous connus à l'avance (entiers $0..n-1$)
 - au départ, chaque élément forme un ensemble
 - désigner 2 éléments, et fusionner leurs ensembles respectifs
 - tester si 2 éléments sont dans le même ensemble (relation R)
- Modélise des "*partitions*" ou des "*classes d'équivalence*". Relations d'équivalence :
 $(R(x, x)) \quad \&\& \quad (R(x, y) \Rightarrow R(y, x)) \quad \&\& \quad (R(x, y) \&\& R(y, z) \Rightarrow R(y, z))$
- C'est un type abstrait utile à quelques algorithmes (p. ex. sur les graphes, labyrinthes...)
- Principes d'implémentation :
 - un ensemble = un arbre (quelconque)
 - les fils sont dans le même ensemble que le parent
 - 2 éléments appartiennent au même ensemble s'ils ont même racine
 - fusion : admettre la racine de l'un comme fils de la racine de l'autre !
 - pas besoin de chaînage "vers le bas"
- Représentation (tableau des parents) :
 - éléments : entiers entre 0 et max
 - toute la forêt : *un* tableau d'entiers !!
 - *indices* du tableau = éléments *contenu* d'une case = indice du parent

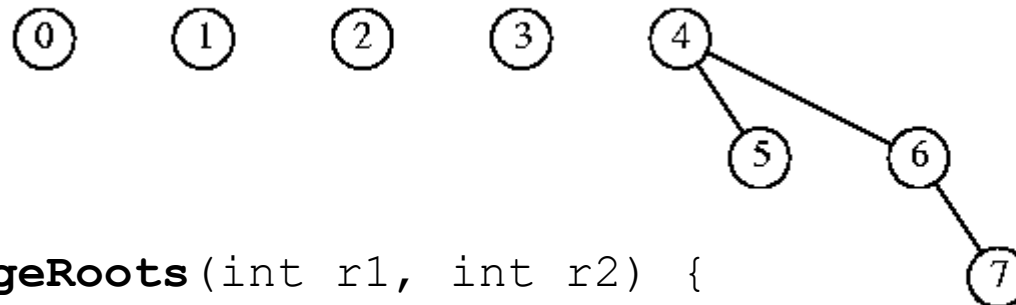


Exemple



```
int root(int elt) {  
    if (parent[elt] == -1) return elt;  
    return root(parent[elt]);  
}
```

| | |
|---|----|
| 0 | -1 |
| 1 | -1 |
| 2 | -1 |
| 3 | -1 |
| 4 | -1 |
| 5 | 4 |
| 6 | -1 |
| 7 | 6 |



```
void mergeRoots(int r1, int r2) {  
    if (r1 == r2) return;  
    parent[r1] = r2;  
}
```

| | |
|---|----|
| 0 | -1 |
| 1 | -1 |
| 2 | -1 |
| 3 | -1 |
| 4 | -1 |
| 5 | 4 |
| 6 | 4 |
| 7 | 6 |

- On pourrait déclarer une constante

```
static final int NULL_INDEX = -1;
```

Problème : comment garantir des arbres équilibrés ?

- Une solution : "*union-by-height*" : attacher l'arbre le moins haut à l'autre
- Une autre : "*union-by-size*" : attacher l'arbre qui a le moins d'élts à l'autre
- Dans les deux cas, la hauteur est alors en $O(\ln n)$. Preuve dans [Weiss10]
- Oui, mais comment gérer l'information de la taille de chaque arbre ?
- Une solution économe, mais discutable :
 - stocker la taille seulement pour les racines (ça, d'accord !)
 - changer `parent[root]==-1` en `parent[root] == -sizeofTree`
- Très astucieux !... Mais surtout ne pas prendre exemple !
Ça viole un principe important : attribuer *un rôle clair à chaque donnée*.
- Codage :

```
void mergeRoots(int r1, int r2) {  
    // if r1 is exactly r2, nothing to do  
    // ensure that size(r1) >= size(r2)  
    //     (i.e. swap them if necessary)  
    // add sizeof(r2) to sizeof(r1)  
    // attach r2 as a child of r1  
}
```

- Difficile de faire mieux pour l'opération d'union. Voyons l'opération d'accès

Compression de chemins

- Idée : une fois qu'on a trouvé la racine, on peut y rattacher directement le nœud accédé (et tous les nœuds sur le chemin !)

- Codage récursif :

```
int root(int elt) {  
    // si elt est une racine, alors return elt  
    // soit R la racine du parent (appel récursif)  
    // elt prend R comme nouveau parent  
    // return R;  
}
```

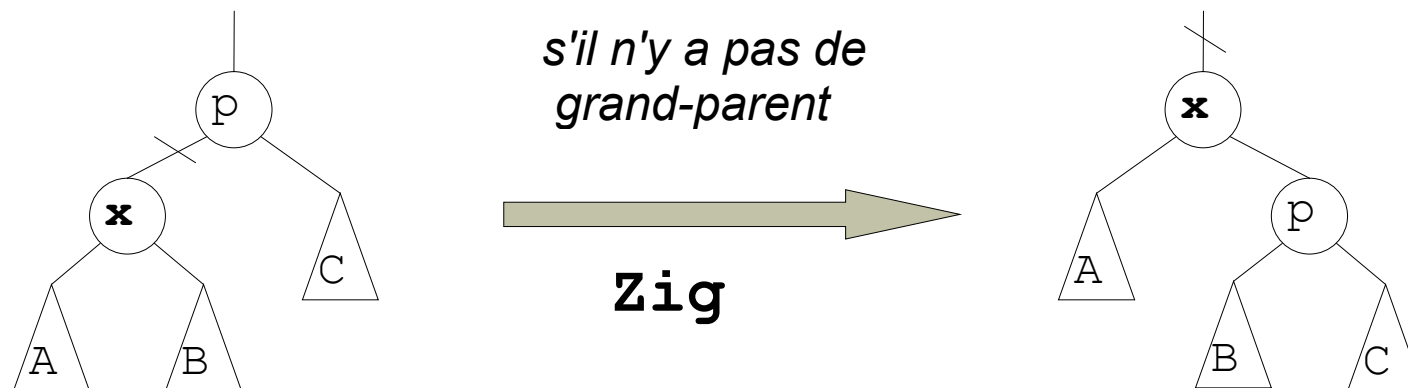
- Ça ne change pas la taille de l'arbre !
- Si on avait choisi "*union-by-height*", la compression diminue parfois la hauteur et il serait trop coûteux de la recalculer; ça devient donc une estimation (pessimiste) de la vraie hauteur. Cette variante est la meilleure, on l'appelle "*union-by-rank*"
- Le code final est remarquablement court (et quand même clair, et très efficace)
- Complexité amortie : une séquence de M opérations est en $O(M \ln n)$, et même presque en $O(M)$. C'est donc extrêmement efficace.
- L'analyse mathématique est un peu délicate...

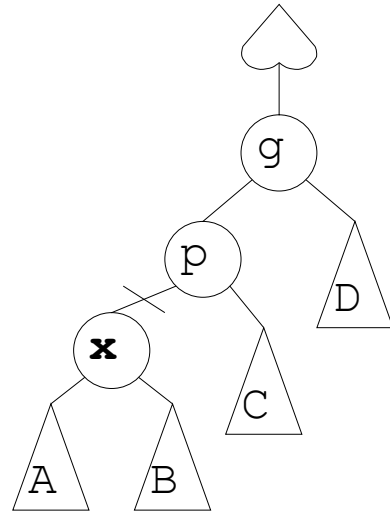
Arbres Splay (une structure de données auto-organisée)

Règle 20/80

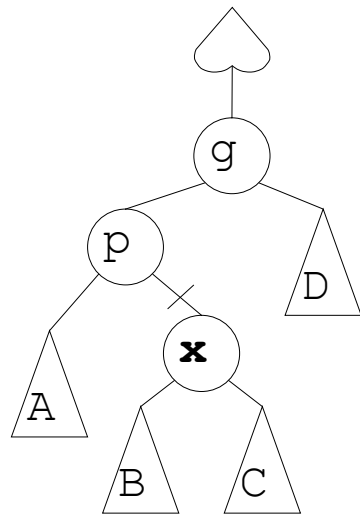
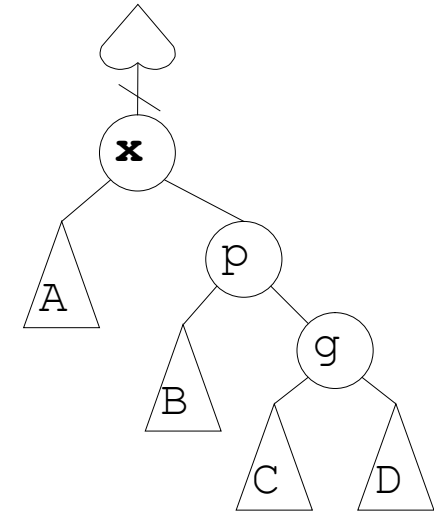
- La règle 20/80 énonce que 20% des efforts suffisent pour produire 80% des résultats (20% des causes provoquent 80% des effets)
- Cette règle s'applique dans de nombreuses situations (pas toutes, bien sûr...). Il s'agit d'un principe, pas d'un pourcentage précis
- En économie :
 - dans un magasin, 80% du chiffre d'affaire est atteint avec seulement 20% des sortes d'articles
 - 80% des biens d'un pays sont détenus par 20% des gens
- En informatique :
 - 80% du temps de calcul est passé dans 20% des lignes de codes
 - dans une base de données, 80% des requêtes portent sur 20% des données
 - à court terme, presque tous les accès RAM portent sur la même zone
- Conséquences :
 - pour accélérer un code, il faut cibler les lignes à modifier
 - placer les données peu demandées "vers la fin du fichier"
 - mémoire cache

- Comment tenir compte de la *règle 20/80* (ou *5/95*) dans un arbre de fouille ?
- Mettre les éléments souvent demandés proches de la racine !
- ... Et profiter d'équilibrer au mieux l'arbre à chaque accès (2ème avantage)
- Une solution : les arbres Splay. Complexité amortie $O(\ln n)$
Une structure de données très efficace en pratique
- Après chaque accès, l'élément accédé se retrouve à la racine
- "Accès" = "recherche d'un élément existant ou non" ou "insertion" ou "suppression"
- Si l'élément est absent, on s'arrête au dernier nœud qu'on rencontre en descendant
- Principe d'accès à un nœud : trouver l'élément, puis le faire remonter par des rotations, en respectant les cas de figure suivants (manquent les cas symétriques) :

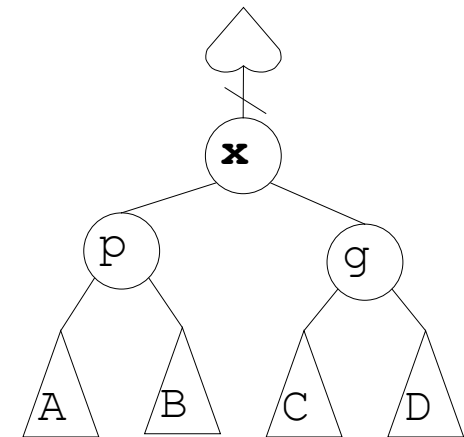




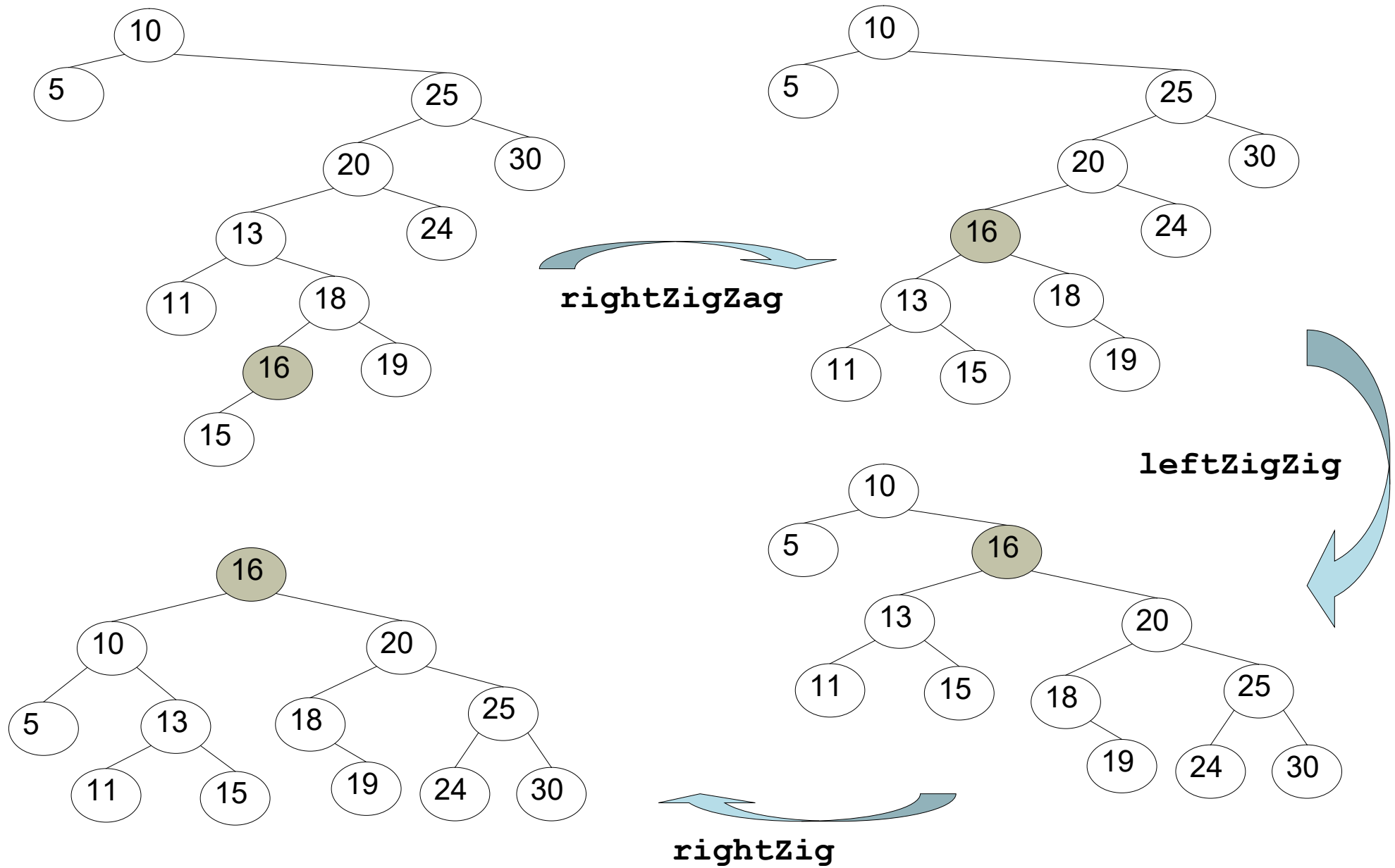
ZigZig



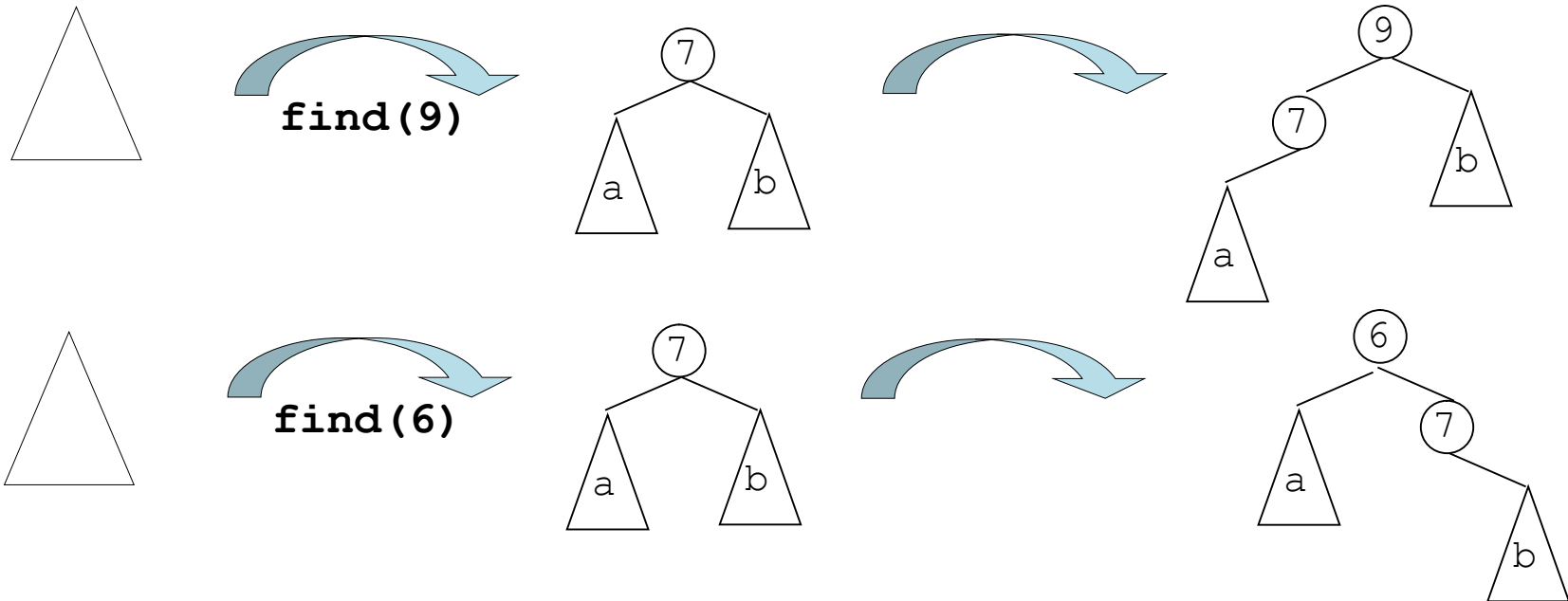
ZigZag



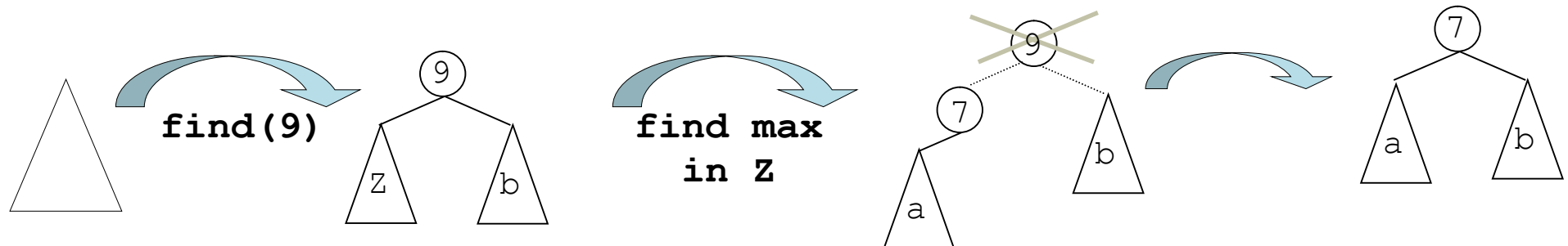
Exemple : locate(17) puis splayToRoot



- Insertion d'un élément:
 - accès à l'élément : ça monte le plus proche à la racine
 - s'il est bien absent, on l'ajoute comme racine et on recolle les bons sous-arbres



- Suppression d'un élément :
 - accès à cet élément (s'il est absent on a fini)
 - s'il n'y a qu'un sous-arbre, on peut facilement enlever la racine
 - Sinon faire remonter le maximum du sous-arbre gauche puis recoller



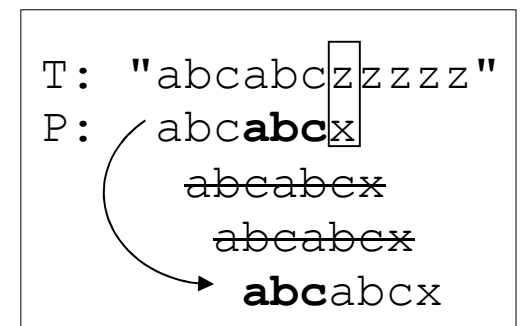
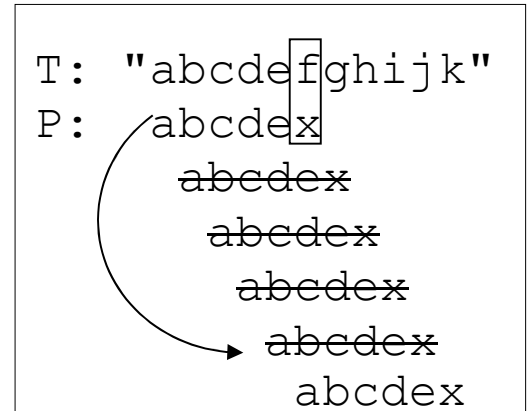
Recherche de sous-chaînes

- Trouver l'indice du début d'une occurrence de la chaîne P dans la chaîne T
- Retourner -1 s'il n'y a aucune occurrence (comme `String.indexOf(String)`)
- Soit n la longueur de T, et m la longueur de P

Algo de force brute en $O(nm)$ au pire des cas

- Idée : essayer de placer P à chaque position 0..(n-m) de T. Codage :

```
int indexOf(String t, String p) {  
    int i, j; int n=t.length(); int m=p.length();  
    for (i=0; i<=n-m; i++) {  
        for (j=0; j<m; j++)  
            if (t.charAt(i+j) != p.charAt(j))  
                break;  
        if (j==m) return i;  
    }  
    return -1;  
}
```



Algo KMP (Knuth-Morris-Pratt) en $O(n+m)$

- Idée : dans l'exemple, au moment où constate que 'f' != 'x', on sait aussi que ça ne sert à rien de tester les positions intermédiaires (elles ne peuvent pas commencer par un 'a' !)

- Idée : avancer davantage la position de début, quand c'est possible :

T : abcggggabcgggcdeeeeeeeeeeeeeeeee

P : abcggggabkj

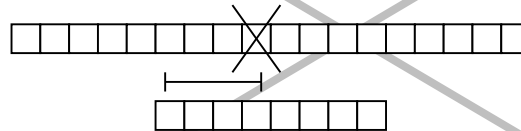
← abcggggabkj

abcggggabkj

décalage : 2

décalage : 0

- On ne revient jamais en arrière dans T.
- A chaque position (d'arrêt) sur P correspond un "réajustement" idéal
- Fonction d'échec de P : associer un décalage \leftarrow à chaque position de P
 $f(j)$ = longueur du plus grand préfixe de P qui soit un suffixe de $P[1..j]$
- En cas d'échec à position j, $f(j-1)$ nous indique comment placer P contre la position courante de T



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| c | c | d | a | a | b | b | c | c | a |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 0 |

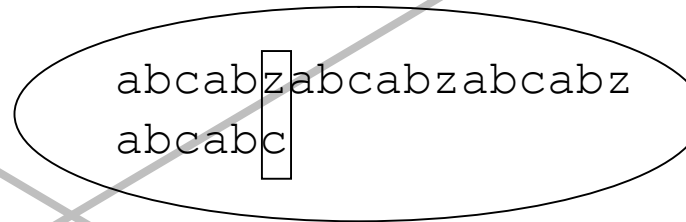
- Plus $f(i)$ est petit, plus on avancera vite !
- Cette fonction peut être pré-calculée (tableau d'entiers f)

```
int[] kmp_failure(String p);
int kmp_indexOf(String t, String p, int[] f);
```

- Les deux implémentations se ressemblent
- $O(n+m)$. Pour sentir l'amélioration, T doit contenir de nombreux préfixes de P...

Vers un algo encore meilleur (Boyer-Moore)

- On reprend l'idée de KMP, mais on ajoute des subtilités. Par exemple :
- Si on compare P de droite à gauche, on peut faire de plus grands sauts !
Il y aura des caractères de T qu'on ne regardera même pas !
- Au meilleur des cas, on vérifie l'absence d'occurrence en N/M étapes !
- Au pire des cas, on en reste à $O(n+m)$



Recherche de chaînes avec "wild cards" (grep 'ab*cd')

- On parle d' "appariement de motifs" (*string pattern matching*)
- Ça devient des *expressions régulières*
- Algo : transformer P en un automate d'états fini, et faire tourner l'automate
- Il y a d'autres algorithmes...

Algorithme de Rabin-Karp

- Considérons les caractères comme des entiers sur $[0..d-1]$. Objections ?
- Un String " $c_3c_2c_1c_0$ " = un (grand) entier $c_3*d^3 + c_2*d^2 + c_1*d + c_0$ $d=256$
- On peut calculer le nombre (sous-chaîne) $t[i+1 .. j+1]$ à partir de $t[i..j]$

$$t[i+1 .. j+1] = d * (t[i..j] - c_i * d^{j-i}) + c_{j+1}$$

- Exemple :

| | | |
|--------------------------------|--|----------|
| 823436543211234567897654322123 | | 54321 |
| 82343 | T | P |
| 23436 | | |
| | $23436 = 10 * (82343 - 8 * 10000) + 6$ | |

- Amorce : $12343 = ((((\underline{1} * 10 + \underline{2}) * 10 + \underline{3}) * 10 + \underline{4}) * 10 + \underline{3})$
- Hachons : on ne calcule jamais ces nombres gigantesques, mais juste leur *empreinte* :
 $h(u) = u \% \text{hasher}$ (un grand nombre premier sera un bon hacheur)

- En plus, on évite des overflows en distribuant le modulo ! (cf. RSA et `powerMod()`)
 $(w * x + y) \% z = ((w \% z) * (x \% z) + (y \% z)) \% z$

- Si une valeur de hachage correspond à celle de P, il suffit de s'assurer que la chaîne est identique (ça sera presque toujours le cas)
- $O(n+m)$ dans le cas moyen. $O(nm)$ dans le pire des cas, mais :
 - on pourrait choisir au hasard le hacheur (voire la base $d=256$)
 - efficace pour chercher plusieurs mots à la fois (mettre plusieurs empreintes dans une table de hachage pour tester la présence d'un des mots en $O(1)$...)

Distance d'édition (Levenshtein distance)

- Deux mots sont parfois complètement différents, parfois très similaires...
- Définir une mesure de différence entre deux mots

- Nombre minimal d'insertions/suppressions/substitutions nécessaires pour passer de l'un à l'autre
- Il s'agit bien d'une *distance* mathématique

Exemple : $\text{dist}(\text{"abcde"}, \text{"bgdfe"}) == 3$

abcde

delete a

bcde

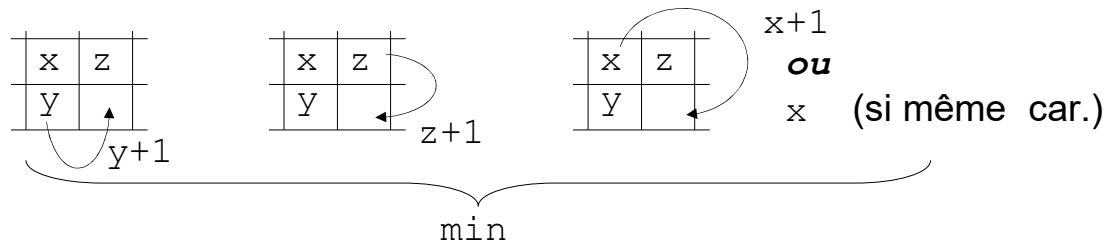
replace c by g

bgde

insert f

bgdfe

- Algorithme de *programmation dynamique* :
tableau des solutions,
à remplir depuis la case (0,0)



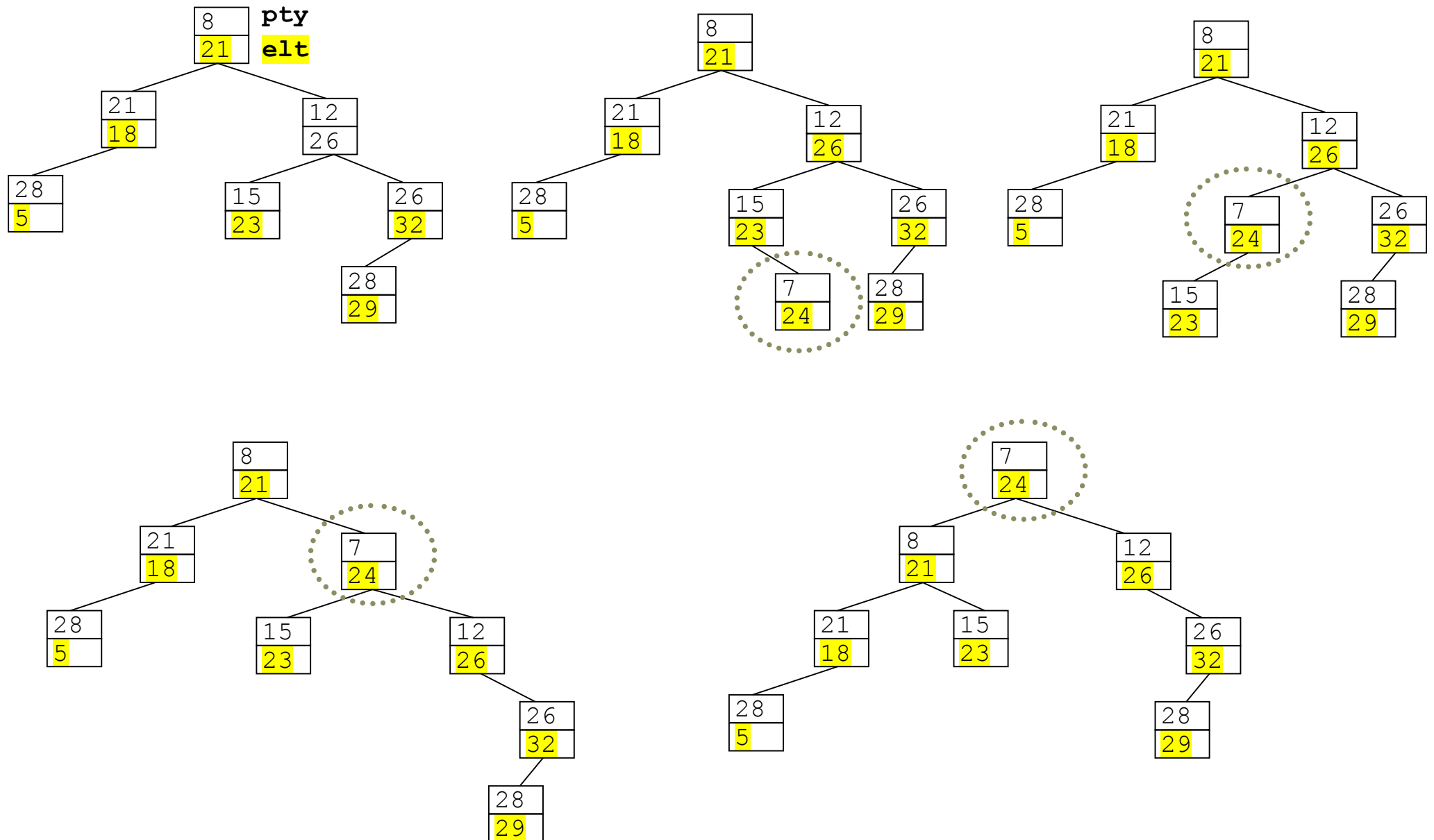
| | | . | b | bg | bgd | bgdf | bgdfe |
|-------|---|---|---|----|-----|------|-------|
| - | 0 | 1 | 2 | 3 | 4 | 5 | |
| a | 1 | 1 | 2 | 3 | 4 | 5 | |
| ab | 2 | 1 | 2 | 3 | 4 | 5 | |
| abc | 3 | 2 | 2 | 3 | 4 | 5 | |
| abcd | 4 | 3 | 3 | 2 | 3 | 4 | |
| abcde | 5 | 4 | 4 | 3 | 3 | 3 | |

Treaps (une structure de données probabiliste)

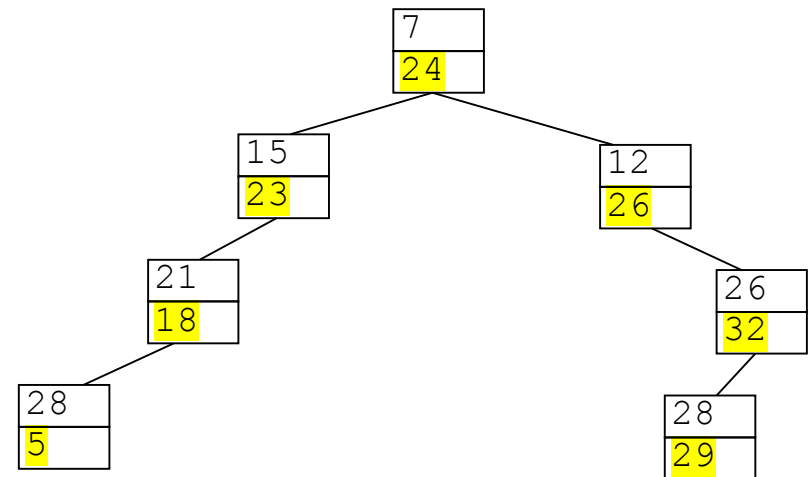
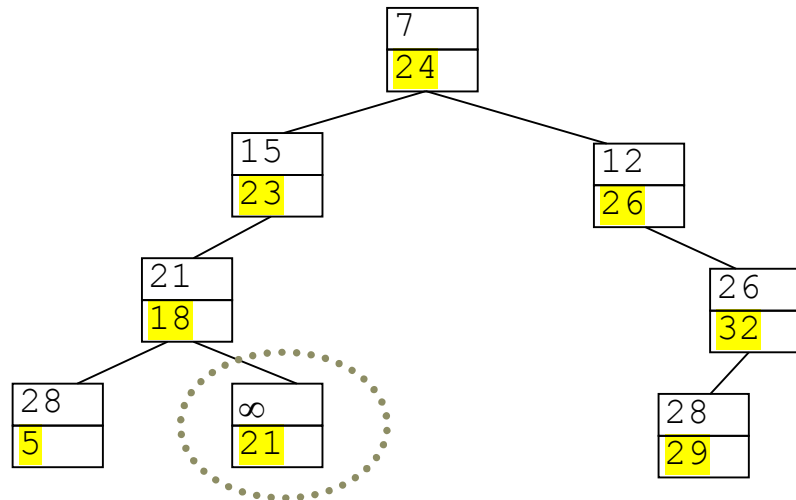
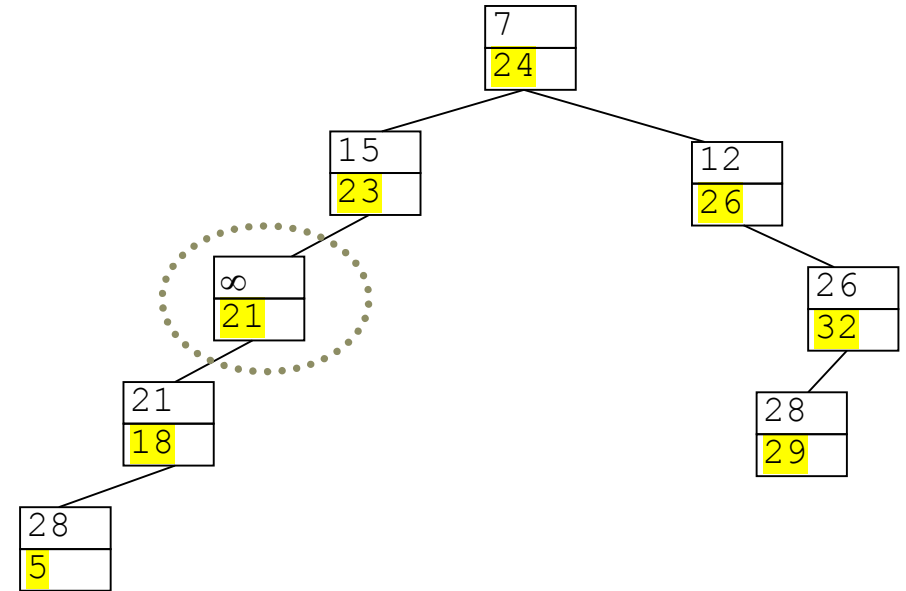
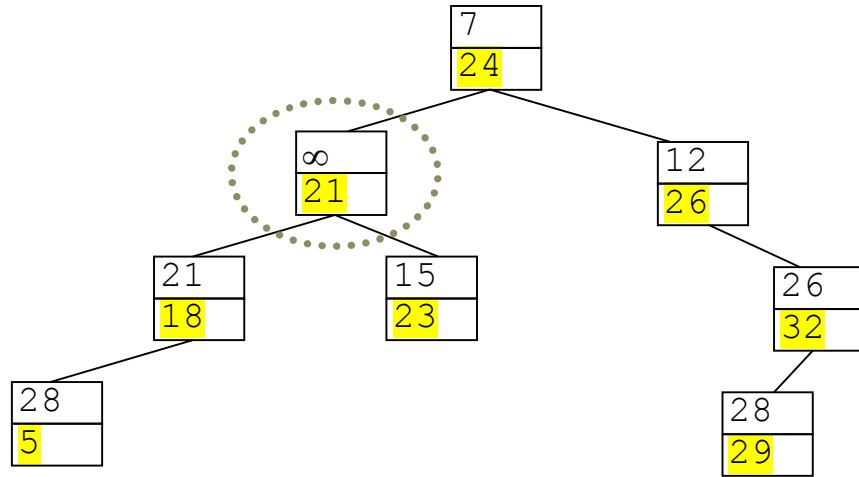
- Comment garantir des arbres de fouille équilibrés... en comptant sur le hasard !
- Pour des données aléatoires, le BST standard n'est pas trop mauvais (en moyenne $O(\ln n)$). Hélas l'input est rarement aléatoire...
- Idée : utiliser des nombres aléatoires pour façonner l'arbre de fouille
- A chaque élément est associée une priorité entière, tirée au sort à l'insertion
- Treap : contraction de "binary search TRee" et "hEAP" :
 - les éléments doivent respecter la propriété BST
 - les priorités doivent respecter la propriété du Heap
- Algo de recherche : comme dans tout BST
- Algo d'insertion :
 - tirer au sort une priorité; construire le nœud d'arbre
 - insérer normalement à la feuille, comme dans tout BST
 - faire remonter par des rotations jusqu'à satisfaire la propriété Heap
- Algo de suppression :
 - rechercher le nœud
 - changer la priorité en $+\infty$
 - faire descendre par des rotations jusqu'à satisfaire la propriété Heap (choisir le côté de façon à faire remonter la plus petite priorité)
 - supprimer le nœud (c'est maintenant une feuille !)

Example

```
insert(24)
pty = 7
```



delete (21)



- Complexité *espérée* $O(\ln n)$.
Le codage peut être assez simple

Codage

```
class TreapElt implements Comparable {
    private Comparable elt;
    private int        pty;
    private static Random rnd=new Random();
    public TreapElt(Comparable elt) {
        this.elt=elt; this.pty=rnd.nextInt();
    }
    public int compareTo(Object o) {          // au sens "BST"
        return elt.compareTo( ((TreapElt)o).elt);
    }    ...
}

public class Treap extends BST {...
    void percolateUp(BTreeItr ti) {
        while((!ti.isRoot()) && isLess(ti, ti.up())) {
            if(ti.isLeftArc()) { ti=ti.up(); ti.rotateRight(); }
            else                 { ti=ti.up(); ti.rotateLeft(); }
        }
    }

    boolean isLess(BTreeItr a, BTreeItr b) {    // au sens "heap"
        TreapElt ca = (TreapElt)a.consult();
        TreapElt cb = (TreapElt)b.consult();
        return ca.pty()<cb.pty();
    }
}
```

L'ingénieur et les tests

- La *rigueur* est une qualité qu'on recherche chez un ingénieur (en informatique comme dans les autres domaines)
- La démarche de test est un point important de tout projet

Tester* \neq *Essayer
(niveau ingénieur) (niveau apprenti)

Propriétés d'un *test* :

- **conçu avec soin** : objectifs clairs, souci d'exhaustivité...
- **reproductible** : automatiser...
- **documenté** : démarche et observations...

*"Software features that can't be demonstrated by automated tests
simply don't exist" [Beck]*

- La programmation est un métier dangereux...
1 min pour écrire du code Java, 2 jours (semaines) pour le comprendre...

Comment prévenir les bugs ? → **Améliorer la qualité du code**

- Bien réfléchir, bien coder.

Comment détecter un bug ? → **Augmenter la confiance dans le code**

- *Tester*. Notion de scénario de test (test case) :
 - jeux de données et résultats attendus (appels et vérifier les propriétés)
 - tests positifs ou négatifs.

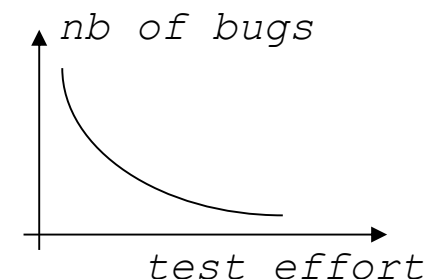
Comment localiser un bug ?

- *Isoler* le problème. Créer un scénario qui permet de *reproduire* le bug.
- Maîtriser le *debugger*

Comment éliminer un bug ?

- Le *comprendre* complètement.
- En profiter pour s'améliorer.

"The rest is **attitude**..."



Quelques conseils

- "Code a little, test a little" "Work 50% on features, 50% on tests"
- Détecter les problèmes le plus tôt possible. Automatiser
- Spécifier clairement chaque morceau de programme (pré/post-conditions)

Analyse *statique* d'un programme (=code inspection=sans exécuter)

Quelques instruments

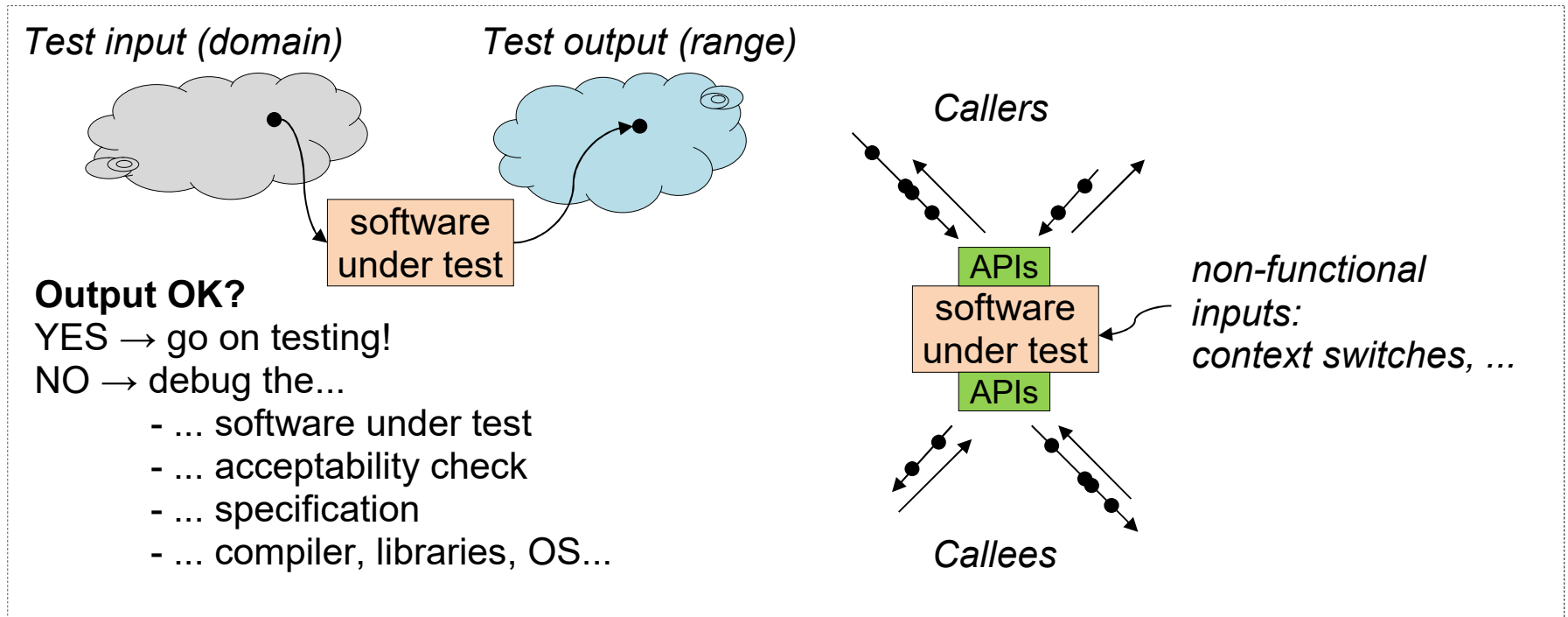
- *Formal verification* : prouver que c'est correct, formuler les hypothèses
- *Code review* : soumettre le code à des "experts". Critiquer en groupe
- *Coding checklists*
- *Coding standards* (conventions de codage). La vérification peut être automatisée !
- Variables/méthodes inutiles, attributs redondants (information re-stockée)
attributs au lieu de variables locales, syndrome du copy/paste (factorisation), codage "spaghetti", méthodes trop longues, magic numbers, ...
- Outils de *refactoring*
- *Program (code) checkers* :
 - warnings du compilateur,
 - jlint/antic, jml, escjava, findBugs...
 - outils orientés sécurité (code injection, buffer overflow attacks...)
- *Analyse de complexité* (gaspillage de CPU ou de RAM)
- Analyse de la qualité d'une découpe modulaire (cohésion des classes) : il existe certains indicateurs... Soigner le génie logiciel !
- *Graphe d'appel (call graph)* : quelle méthode appelle quelles autres ?
Code understanding, reverse engineering

Analyse dynamique d'un programme (en l'exécutant)

- *Debugger* : se discipliner à recourir au debugger pour ramper au moins une fois dans chaque ligne de code... C'est du temps gagné et non perdu !
- *Runtime (memory) checker* :
 - comment garantir qu'on ne fait que des accès mémoire autorisés ?
 - en C : `char *t = malloc(7); char a=t[6]; t[8]='x';`
 - produits existants : Valgrind, dmalloc, DUMA, CCured, PurifyPlus...
 - c'est une aide indispensable en C
- *Profiler* : produire des statistiques d'une exécution : répartition du temps CPU, trace des allocations dynamiques de mémoire... Ça permet d'identifier les portions de code qui méritent d'être optimisées
- *Software verification*: Have we built the software right? (Spécification respectée)
Software validation: Have we built the right software? (Satisfaction du client)
- On peut tester à différents niveaux :
 - *unit testing* : ciblé sur une petite unité de codage isolée (classe); souvent réalisé par le même développeur qui implémente, et en même temps (ou avant !)
 - *integration testing* : ciblé sur la mise en commun de plusieurs classes/modules
 - *system testing (functional testing)* : logiciel complet, point de vue utilisateur
- Tester peut être vraiment **fun**, ça demande d'être très créatif, d'avoir de l'imagination !
- Testing : blackbox vs whitebox testing.
- *Fault injection* : simulation d'une défectuosité d'un composant

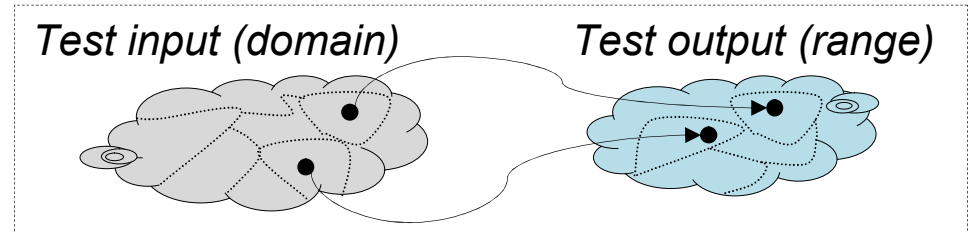
Blackbox testing

- On teste une boîte noire; on connaît la spécification (pré/post), mais pas le codage.
- Préparer des scénarios (*test cases*). Plateformes de tests (JUnit)



- *Regression testing* : Fréquemment tout re-tester (pas seulement ce qui vient d'être modifié, mais vérifier que les autres fonctionnalités n'ont pas été affectées)
- Génération automatique de scénarios de tests. *Random testing*. *Fuzzing*
- *Mutation testing* : modifications aléatoires du code, en espérant que ce soit détecté par les tests; le but est de mesurer la qualité des tests.
- On ne peut tester que ce qui est bien spécifié... Exemple : racine carrée

- Essayer de partitionner le domaine de l'input en "*zones qui auront le même comportement*", et tester un représentant de chaque zone.



- *Parameter-value coverage* : chaque paramètre a pris chaque (catégorie de) valeur. Pour un `int`, par exemple : `MIN_VALUE`, `-24`, `-1`, `0`, `1`, `+37`, `MAX_VALUE`.

Whitebox testing = structural testing = code coverage = couverture de code

- On connaît le codage, et on veut savoir si tout le code a été testé
- *Code unit* : l'unité testée (une méthode, une classe, un paquetage).
- Il y a de nombreuses *métriques* possibles; voici les principales
- *Method coverage* : chaque méthode a été appelée
- *Statement coverage* : chaque instruction a été exécutée
- *Line coverage* : chaque ligne de code source (avec au moins une instruction)
- *Branch coverage* : chaque embranchement a été exécuté

```
int max(int a, int b) {
    int res=0;
    if(a>res) a=res;
    if(b>res) res=b;
    return res;
}
```

(0,0) → line coverage
 (2,3) → statement coverage
 (0,0)(2,3) → branch coverage
 (0,0)(2,0)(0,3)(2,3) → path coverage

- *MC/DC* (modified condition decision coverage) : en gros, toute expression booléenne doit avoir pris les deux valeurs possibles, et pour chaque condition (expression booléenne non décomposable), il faut un test où sa valeur est déterminante pour le choix d'un embranchement. C'est un critère difficile à atteindre, un standard qui est imposé pour certains logiciels critiques (p. ex. en aéronautique)
- *Path coverage* : chaque cheminement possible dans le programme a été suivi

```

for (int i=0; i<20; i++) {
    if (...) continue;
    if (...) {    if (...) a(); else b();
    } else      {    if (...) c(); else d();
    }
}

```

Dans cet exemple, il y a déjà 5^{20} cheminements possibles (env. 100'000 mias)...

- *Loop coverage* : chaque boucle a été exercée dans 3 situations : 0 fois, 1 fois, et plusieurs fois.
- *Synchronization coverage* : pour un programme concurrent, chaque lock doit avoir effectivement empêché un accès

Exemples

```
protected void locate(Comparable e) {  
    int order = 0;  
    ti=t.root();  
    while ( !ti.isBottom() && order != 0 ) {  
        order = ((Comparable)ti.consult()).compareTo(e);  
        if (order == 0) return;  
        if (order > 0) ti=ti.left (); else ti=ti.right();  
    }  
}
```

- Ici, on ne pourra pas obtenir 100% de *statement coverage*... et pour cause !
- Certains outils d'analyse statique peuvent détecter des erreurs de ce genre.
- On ne peut pas tester les instructions qui manquent ! (*omission faults*)
Même 100% de couverture ne signifie pas que le code est correct !...

```
int max(int a, int b) {  
    return a;  
}
```

(5,2) → path coverage !!!

- Métriques : mesures *quantitatives* de la qualité d'un code ou d'un test :
 - 0 avertissements du compilateur (warnings)
 - 100% de "statement coverage"
 - 100% de "branch coverage"

Gestion de la mémoire, ses dangers

- Pour les données d'un programme, il y a 3 zones de mémoire :

- la *data segment* contient certaines variables (globales, statiques)
allocation : compilateur, chargement du programme
libération : déchargement (*unload*) du programme
- la *pile d'exécution* contient les variables locales et les paramètres
allocation : appel de procédure/fonction
libération : retour de procédure/fonction

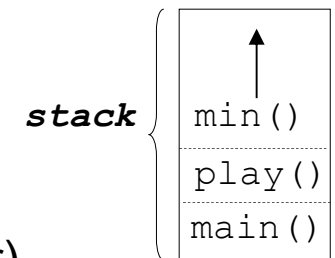
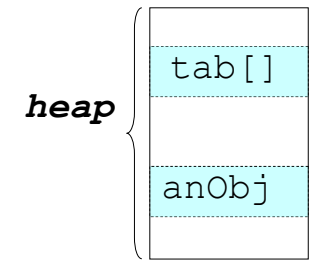
- le "*heap*" gère les allocations dynamiques

allocation : explicite (`new A()`, `malloc()`, ...)
libération : explicite (`free()`) ou ramasse-miette (garbage collector)

- En général, il faut définir clairement qui a la responsabilité de l'allocation et de la déallocation (libération) des données. En C, le danger est très grand :

```
char *p = (char*) malloc(10); p=NULL;  
int a; free(&a);  
char *t = (char*)malloc(7); char a=t[7];
```

- *Memory leaks* (fuite) : mauvaise libération des cases mémoires inutilisées
- Un ramasse-miettes (garbage collector) ne résout pas tout : il faut s'assurer qu'on supprime toutes les références vers les objets devenus "inutiles"
- En Java, rôle et dangers de la méthode `finalize()`



Compression de données (codage de source)

- Comment coder des données avec un minimum d'espace mémoire ?
- Supprimer les redondances (on se limite à la compression "sans perte")
- Toute donnée n'est pas compressible. Exemple : données aléatoires
- La "bonne" méthode dépend du contexte (type de données)

Codage par plage

- Idée : remplacer 'XX...X' par 'NX'. Exemple :
AAABBBBAACAAAAAAAAAABBBB → 3A4B2A1C8A4B
- Le décodage est plutôt simple...
- ... si on sait distinguer les N_i des X_i ! Problème du "*escape character*"
- Pour des données binaires, on ne représente que les N_i (commençant par 0)

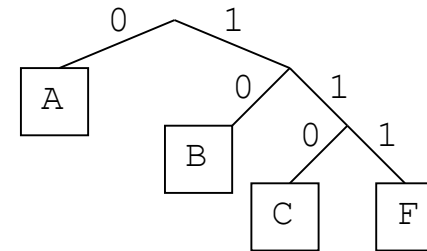
Codage de Huffman

- Codage binaire de longueur variable (alors que ASCII est de longueur fixe)
- Idée : codes courts pour symboles fréquents, codes longs pour symboles rares
- Exemple : ABACABABAF. A=0, B=10, C=110, F=111
 01001100100100111 17 bits (contre 80=10x8 bits en ASCII)
- On devra (en plus des données) stocker la table de codage (p. ex. au début)

- On parle de "code de préfixe" (aucun code n'est le préfixe d'un autre !)
- On doit connaître les fréquences d'apparition (a priori, ou en les estimant)
- L'algorithme de Huffman permet de construire les codes optimaux

Arbre binaire digital (= bitwise trie = arbre à lettre binaire)

- Idée : une feuille d'un arbre binaire = un chemin depuis la racine. Notation : "0" pour "gauche", "1" pour "droit"
- Donc un arbre binaire = table de codage pour les éléments des feuilles



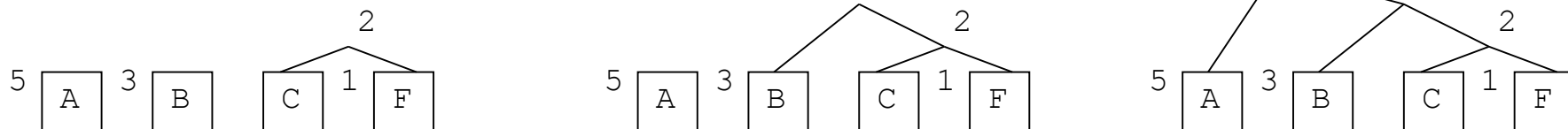
Algorithme (glouton) de Huffman

- Représentation de la table de codage par un arbre binaire digital

- Amorce : créer les feuilles individuelles, avec leurs fréquences respectives



- A chaque étape, fusionner les composants les moins fréquents, et additionner leurs fréquences



- Implémentation : `buildHuffmanTree()` utilise une file de priorité !

Une implémentation complète

- Ici, une implémentation complète est un bon exercice...

ACCÈS AUX FICHIERS

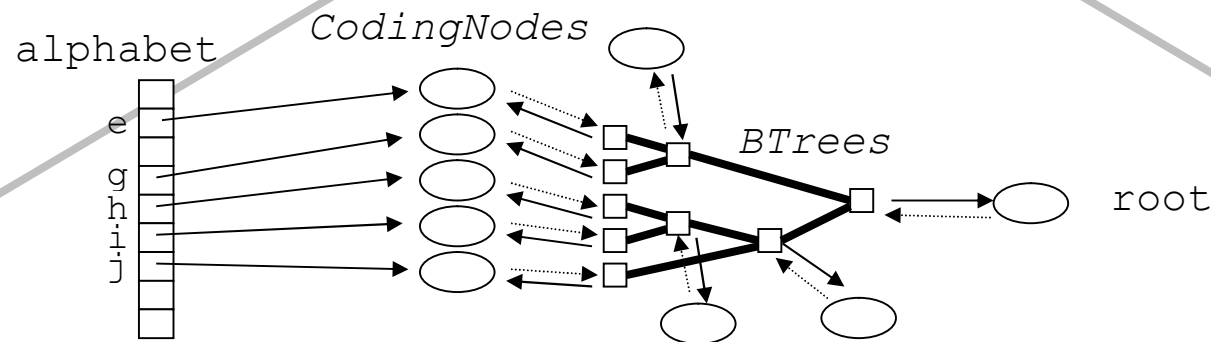
- On va traiter des fichiers texte (codes ASCII 0-255)
- Accès aux fichiers : l'algorithme de Huffman suppose qu'on lit/écrit des bits isolés. On va définir des classes BitReader/BitWriter
- Bien sûr, on ne peut pas écrire un fichier avec exactement 19 bits... Il faut trouver une astuce pour détecter la fin du fichier : un pseudo-caractère (code 256) final, ou bien un encodage de la taille elle-même
- Une bonne idée : tester séparément BitReader/BitWriter avant d'écrire le reste...

```
public class BitWriter {  
    public BitWriter (String filename)    throws IOException;  
    public void close()                  throws IOException;  
    public void put    (boolean b)        throws IOException;  
}  
  
public class BitReader {  
    public BitReader (String filename)    throws IOException;  
    public void      close ()             throws IOException;  
    public boolean next ()                throws IOException;  
    public boolean isOver();  
}
```

ARBRE DE CODAGE

- Chaque symbole correspond à une feuille d'un arbre binaire
- Durant l'algorithme de Huffman (construction de l'arbre), on doit manipuler une collection de sous-arbres, créer des noeuds, et rattacher des fils existants
- Durant le décodage, on descend dans l'arbre de la racine vers une feuille
- Durant le codage, on aimerait un accès direct à la feuille d'un symbole donné
- Une solution :
 - les éléments d'arbre sont des CodingNode
 - pendant la construction, un CodingNode possède un alias sur le BTree (racine) qui le contient
 - on gère un tableau de CodingNode, indicé par les symboles
 - on gère une file de priorité de CodingNodes

```
>>> CodingNode    rootOfCodingTree;  
>>> CodingNode[] alphabet;
```



PROGRAMME PRINCIPAL

- C'est ici qu'on doit percevoir le pseudo-code général pour l'algorithme de Huffman, l'encodage, et le décodage
- Au codage, on va lire 2 fois le fichier : d'abord pour compter les occurrences, ensuite pour coder avec l'arbre créé entretemps

```

CodingNode[] countOccurrences(BufferedReader tis);
CodingNode buildHuffmanTree(CodingNode[] alphabet) {
    tout mettre dans une file de priorité
    tant qu'il reste plus de 1 élément dans la file
        retirer le minimum
        retirer le minimum
        créer un nouveau noeud avec la somme des deux
        ajouter dans la file
    retourner le noeud restant
}

void bitFromText(CodingNode[] alpha, BufferedReader tis,
                  BitWriter      bos);

void storeTree(CodingNode root, PrintWriter tos);

CodingNode loadTree(BufferedReader tis);

void textFromBit(CodingNode root, BitReader bis,
                  PrintWriter tos);

```

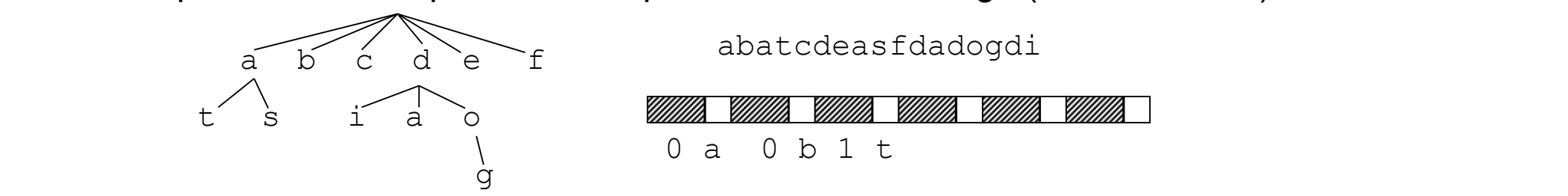
- Cette implémentation est juste un exercice, pas une version de référence
- Une fois construit, on peut voir l'arbre de codage comme un dictionnaire qui associe à un code binaire un (numéro de) symbole : 0010101 → 'b'
- On peut aussi considérer des groupes de caractères comme symboles
- Classes (Inflater, Deflater, Zip..., Gzip...) du paquetage java.util.zip
- Autres aspects du codage : détection/correction d'erreurs (codage de voie)

- Cette implémentation est juste un exercice, pas une version de référence
- Une fois construit, on peut voir l'arbre de codage comme un dictionnaire qui associe à un code binaire un (numéro de) symbole : 0010101 → 'b'
- On peut aussi considérer des groupes de caractères comme symboles
- Classes (Inflater, Deflater, Zip..., Gzip...) du paquetage java.util.zip
- Autres aspects du codage : détection/correction d'erreurs (codage de voie)

Compression Lempel-Ziv (LZ)

- Approche "inverse" de Huffman : on considère la redondance des ss-chaînes
"abcdaefgabcdaefgabcdaefgabcd" avantage Lempel-Ziv
"abacadaeafagahaiajbcdbdbefbgbh" avantage Huffman
- Codes de longueur fixe, mais "symboles" de longueur variable
- On peut aussi se représenter LZ par un arbre de codage (arbre à lettres)

- ## Compression Lempel-Ziv (LZ)
- Approche "inverse" de Huffman : on considère la redondance des ss-chaînes
"abcdaefgabcdaefgabcdaefgabcd" avantage Lempel-Ziv
"abacadaeafagahaiajbcdbdbefbgbh" avantage Huffman
 - Codes de longueur fixe, mais "symboles" de longueur variable
 - On peut aussi se représenter LZ par un arbre de codage (arbre à lettres)



- Grâce à une astuce, on transmet la table de codage en même temps que les informations codées (les deux sont mélangées)

- Exemple d'encodage/décodage LZV

Original

abaacaabcbcbba
~~a~~baacaabcbcbba
~~ab~~aacaabcbcbba
~~aba~~aacaabcbcbba
~~abaa~~caabcbcbba
~~abaa~~c**aab**cbcbba
~~abaa~~caab**cb**cbba
~~abaa~~caabcb**cba**

Encoding

0 'a' 0 'b' 1 'a' 0 'c' 3 'b' 4 'b' 6 'a'
~~0 'a'~~ 0 'b' 1 'a' 0 'c' 3 'b' 4 'b' 6 'a'
~~0 'a'~~ ~~0 'b'~~ 1 'a' 0 'c' 3 'b' 4 'b' 6 'a'
~~0 'a'~~ ~~0 'b'~~ ~~1 'a'~~ 0 'c' 3 'b' 4 'b' 6 'a'
~~0 'a'~~ ~~0 'b'~~ ~~1 'a'~~ ~~0 'c'~~ 3 'b' 4 'b' 6 'a'
~~0 'a'~~ ~~0 'b'~~ ~~1 'a'~~ ~~0 'c'~~ 3 'b' 4 'b' 6 'a'
~~0 'a'~~ ~~0 'b'~~ ~~1 'a'~~ ~~0 'c'~~ 3 'b' 4 'b' 6 'a'
~~0 'a'~~ ~~0 'b'~~ ~~1 'a'~~ ~~0 'c'~~ 3 'b' 4 'b' 6 'a'

Table de codage (Code: String)

| | | | | | | | |
|--------|-------------------|------------------------------|--|---|--|--|---|
| 0 : "" | 0 : "" 1 : "a" | 0 : "" 1 : "a" 2 : "b" | 0 : "" 1 : "a" 2 : "b" 3 : "aa" | 0 : "" 1 : "a" 2 : "b" 3 : "aa" 4 : "c" | 0 : "" 1 : "a" 2 : "b" 3 : "aa" 4 : "c" 5 : "aab" | 0 : "" 1 : "a" 2 : "b" 3 : "aa" 4 : "c" 5 : "aab" 6 : "cb" | 0 : "" 1 : "a" 2 : "b" 3 : "aa" 4 : "c" 5 : "aab" 6 : "cb" 7 : "cba" |
|--------|-------------------|------------------------------|--|---|--|--|---|

```

class CodeTable {
    boolean contains    (String s);
    int      putAndCode  (String s,  char c);
    String    putAndDecode(int index, char c);
}

```

| | |
|----|-------|
| 0: | "" |
| 1: | "a" |
| 2: | "b" |
| 3: | "aa" |
| 4: | "c" |
| 5: | "aab" |
| 6: | "cb" |
| 7: | "cba" |

- **Algorithme d'encodage**

```

int code; char c; String s = "";
CodeTable t=new CodeTable();
while(in.hasMoreChars())
    c=in.nextChar();
    if (t.contains(s+c))
        s += c;
    else
        code = t.putAndCode(s, c);
        out.print(code, c);
        s = "";

```

| | | | | | | |
|---------------|---------------|---------------|---------------|-----------|----|----|
| 0a | 0b | 1a | 0c | 3b | 4b | 6a |
|---------------|---------------|---------------|---------------|-----------|----|----|

- **Algorithme de décodage**

```

CodeTable t=new CodeTable();
while(! in.endOfFile())
    int code = in.readInt(); char c = in.readChar();
    String s = t.putAndDecode(code, c);
    out.print(s);

```

| | | | | | | |
|---------------|---------------|---------------|---------------|-----------|----|----|
| 0a | 0b | 1a | 0c | 3b | 4b | 6a |
|---------------|---------------|---------------|---------------|-----------|----|----|

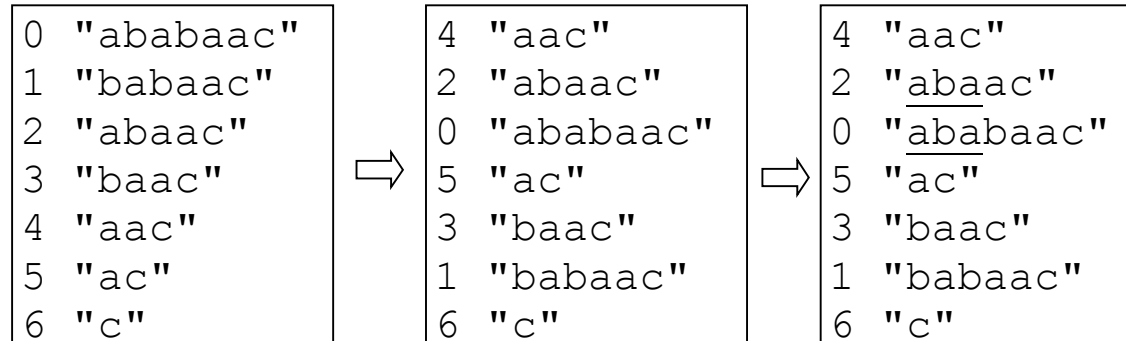
- **Implémentation : régler les détails (fin de fichier), choisir le format du fichier (binaire ?)**

Quelques "petits" problèmes

Plus grande sous-chîne redondante (tableau de suffixes)

- Idée : construire un tableau des (indices des) suffixes, et le trier ($O(n \ln n)$ voire $O(n)$)
- Rechercher dans 2 cases consécutives le plus grand préfixe possible

- Exemple : "ababaac"



Maximum et minimum dans un tableau

- On prend les cases 2 à 2, et on les compare entre elles : la plus petite est un candidat au minimum, la plus grande au maximum. Au total, on économise 25% des comparaisons !

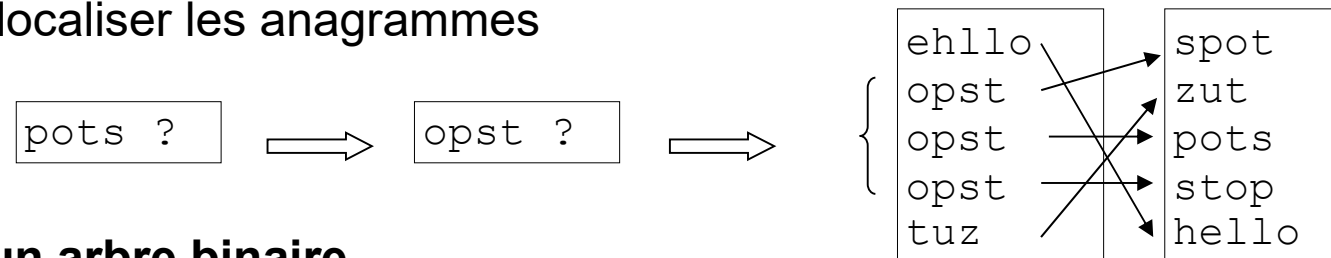
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 3 | 5 | 8 | 7 | 6 | 9 |
|---|---|---|---|---|---|---|---|

$$N/2 + N/2 + N/2$$

```
static void computeMinAndMax (int[] t) {  
    min=max=t[t.length-1];  
    for (int i=0; i<t.length-1; i+=2) {  
        if (t[i]<t[i+1]) { if (t[i] <min) min=t[i];  
                        if (t[i+1]>max) max=t[i+1];  
        } else { if (t[i+1]<min) min=t[i+1];  
                if (t[i] >max) max=t[i];  
        }  
    }  
}
```


Recherche des anagrammes

- Idée : trier les lettres de chaque mot du dictionnaire, et trier ensuite le tableau
- On doit conserver une référence vers le mot d'origine
- Pour trouver les anagrammes, on trie les lettres du mot donné, et on effectue une recherche binaire pour localiser les anagrammes

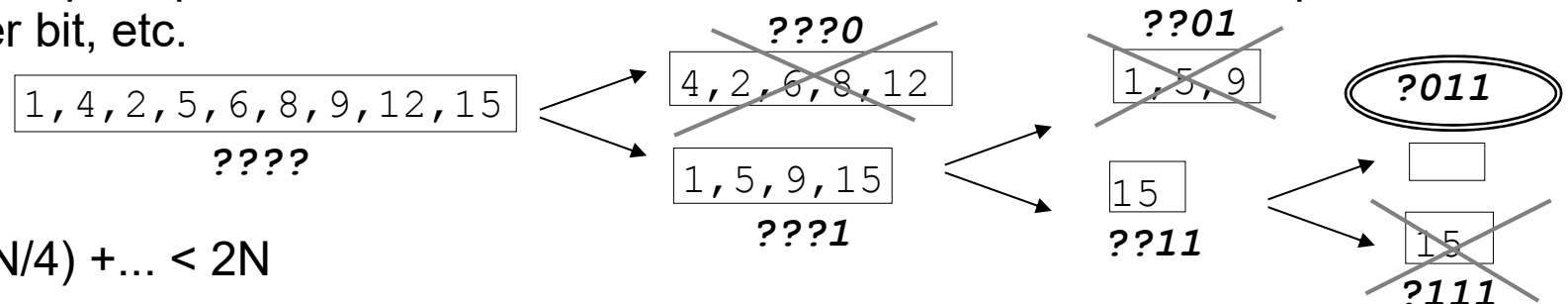


Ancêtre commun dans un arbre binaire

- Calculer la profondeur des deux nœuds, se mettre des deux côtés au même niveau, puis remonter au même rythme.

Trouver un des nombres manquants

- En un passage, on peut séparer en deux fichiers les nombres pairs/impairs. On choisit alors le plus petit ensemble, et le dernier bit devient connu. Répéter avec l'avant-dernier bit, etc.



- $N + (N/2) + (N/4) + \dots < 2N$
- Variante : trouver le nombre manquant entre 0 et n.
Profiter de la formule de la somme des entiers de 1 à n.

FIFO à l'aide de LIFO

- On maintient deux piles :
 - la pile des éléments entrés
 - la pile des éléments prêts à sortir

→ hgfedcba →

hgf edcba

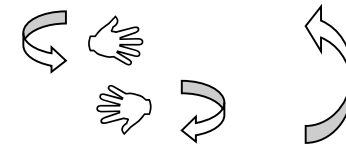
- Si on doit ressortir un élément alors que la pile de sortie est vide, on transvase, en l'inversant, la première pile :

hgfe  hgfe
inverse


- On est sûr que cette opération en $O(n)$ ne se produira qu'après n opérations `push()` en $O(1)$. Les n prochains `pop()` seront en $O(1)$, la complexité amortie est garantie.

Rotation à gauche dans un tableau

- Facile, mais avec un tableau auxiliaire.
Facile pour une rotation de 1 case...



- L'algorithme astucieux demande 3 appels à une méthode d'inversion, celle-ci très facile à écrire en respectant les contraintes demandées
- On peut inventer une solution plus directe, mais le risque de faire une erreur est un peu plus grand !

| | |
|---------------------|---|
| 3, 4, 5, 6, 7, 8, 9 |  4 |
| 3, 4, 5, 6, 9, 8, 7 | |
| 6, 5, 4, 3, 9, 8, 7 | |
| 7, 8, 9, 3, 4, 5, 6 | |

Détection de cycle pour liste simplement chaînée

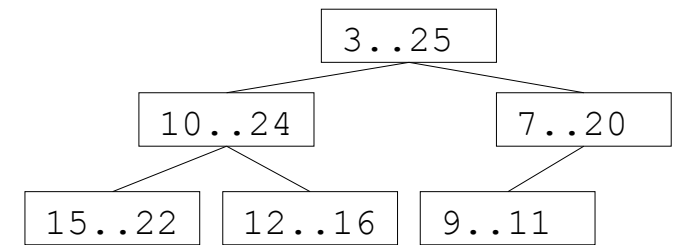
- Le lièvre et la tortue* : faire avancer simultanément deux pointeurs, l'un faisant des pas doubles; ils finiront par se rejoindre...

Partage de périmètre

- Envisager des $[i..j]$ dans $[0..n-1]$, par incrémentation de i ou de j ...

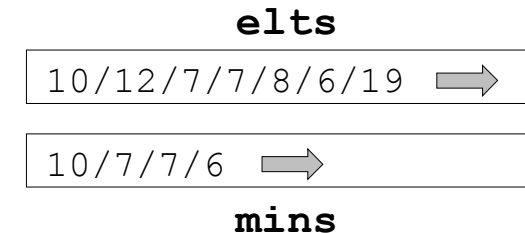
Interval Heap

- Chaque noeud contient 2 éléments, et tout le sous-arbre est inclus dans l'intervalle formé par ces 2 élts (seul le dernier noeud peut contenir 1 ou 2 élts)
- Finalement très similaire au heap normal !



LIFO + consultMin()

- Gérer deux piles, dont une pour mémoriser les minima

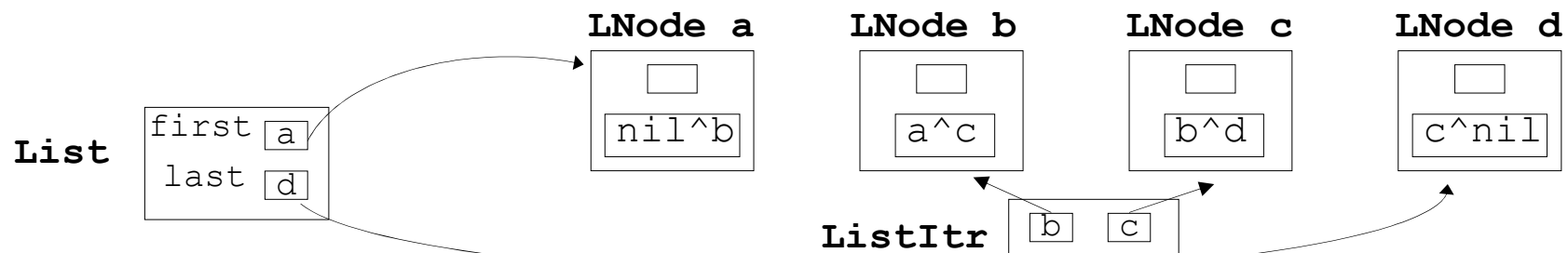
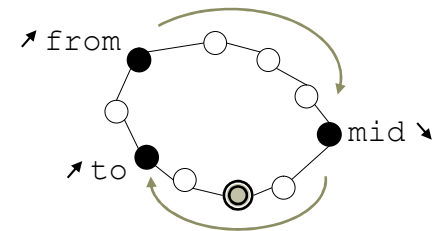


Extrémité dans un polygone convexe

- Recherche binaire et détection de la "tendance" en un point

Chaînage double sans mémoire supplémentaire

- Combiner les pointeurs de nos deux voisins dans un XOR (en Java ^) (On parle de *superposition d'états* en physique quantique...)
- Particulièrement bien adapté à notre version de liste avec position sur les arcs
- Connaissant un voisin (celui d'où on vient) et le résultat du XOR, on peut retrouver l'autre voisin



Optimisation de codes

- Importance de l'efficacité d'un programme (temps et espace-mémoire)
... en plus d'autres aspects ! (correct, robuste, lisible, ergonomique)
- LE moyen : choix des algos et structures de données (analyse de complexité)

"Premature optimization is the root of all evil"

"Don't optimize inside a bad design" "Forget about optimization ~97% of the time"

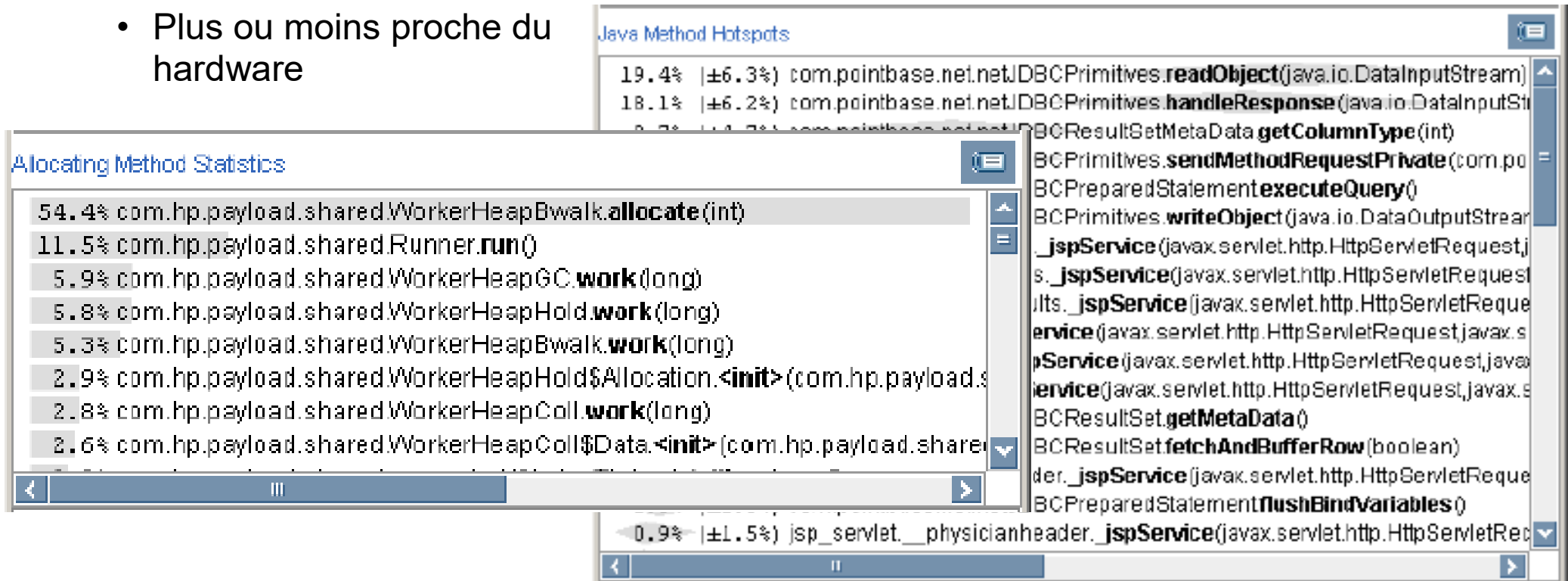
- Il faut privilégier la clarté du codage.
Parfois (rarement), il faut "optimiser".... et vérifier le gain obtenu
- Optimiser est très difficile, et dépend de l'environnement :
 - compilateur
 - système d'exploitation (gestion de mémoire)
 - matériel (cache, pipeline, ...)

Compilateur

- Options d'optimisation du compilateur (ou d'autres "optimiseurs")
(le compilateur GCC a plus de 50 options d'optimisations...)
(certains compilateurs sont capables d'exploiter un *profil* pour mieux optimiser)
- Parfois, on peut l'aider ! (déclarer les méthodes `final/private/static`)
- Prix à payer :
 - code objet plus gros
 - informations perdues pour le debugger

Profileur

- Outil qui rapporte où un programme passe son temps, où il consomme de la mémoire.
- Génération de données de *profil* durant l'exécution, puis présentation des résultats
- Plus ou moins proche du hardware



- Deux approches techniques :
 - Par *instrumentation* du code (ajout automatique d'opérations d'analyse)
 - Par *sampling* (consultation périodique de l'état du programme)
- Profiler un programme modifie plus ou moins ses performances, suivant la technique utilisée (émulation de code, instrumentation, sampling, hardware counters...)

Programmeur

- Quelques techniques de programmeur, pour "affiner" (*tune*) le code
- Changer le code, mais *seulement aux bons endroits* !
(règle 80/20 ou 90/10 : 90% du temps passé dans le 10% du code)
- Après chaque changement, il faut absolument *mesurer le gain* de performance, et *tester* le programme
- Mesurer les performances est difficile (spécialement avec un langage comme Java)

Assume nothing !

- Il ne faut jamais détruire la version "claire" (sans optimisation), afin de :
 - vérifier que la nouvelle version reste correcte
 - pouvoir optimiser un jour pour un autre environnement (CPU, OS, ...)
- Ressources "optimisables" : CPU, RAM, disque dur, réseau, appels systèmes...
- Estimer un *modèle de coûts* : combien coûte (dure) chaque instruction ?
(une division sur les int/double, un new, une affectation, un if, ...)
- Difficulté de mesurer les performances, notamment en Java
cf *micro-benchmarking* et des outils comme JMH

Moyen 0 : Réduire les allocations dynamiques

- Le gestionnaire de mémoire Java est excellent. Mais un mauvais "profil" RAM finit par coûter cher : mémoire cache, pression sur le Garbage Collector, etc.

Moyen 1 : Aplatir les appels ("Inlining")

- Idée : remplacer un appel de méthode directement par son corps

```
<<<for(i=0;i<n;i++)
<<<    m = min(t[i], m);
<<<
<<<int min(int a, int b) {
<<<    if(a<b) return a;
<<<    else    return b;
<<<}
<<<
```

```
<<<for(i=0;i<n;i++)
<<<    if (t[i]<m)
<<<        m = t[i];
<<<    //else
<<<    //    m = m;
<<<
```

Moyen 2 : Stocker au lieu de calculer ("Caching") (compromis espace/temps)

- Idée : éviter de calculer en mémorisant un résultat. Précalculer (lookup table). Mémoriser les derniers inputs/outputs (cf. mémoire cache, arbres Splay)

```
<<<int factorial(int n) {
<<<    res = 1;
<<<    for (int i=1;i<=n;i++)
<<<        res = res*i;
<<<    return res;
<<<}
<<<
```

```
<<<static int[] factorials =
<<<    {1,1,2,6,24,120,720,...};
<<<
<<<int factorial(int n) {
<<<    return factorials[n];
<<<}
<<<
```

"Optimization is almost always an exercise of caching"

Moyen 3 : Sortir les invariants

- Idée : éviter de calculer des expressions constantes, surtout dans des boucles

```
>>for (i=0; i<n; i++)           >>int e = li.l.first elt;
>>    t[i] = li.l.first.elt;    >>for (i=0; i<n; i++)
>>                                >>    t[i] = e;
>>
```

Moyen 4 : Identités algébriques

- Idée : trouver une expression moins coûteuse qui a les propriétés voulues

```
float avg1 = (float)(s1)/n1;
float avg2 = (float)(s2)/n2;
if (avg1 < avg2) ...           if (s1*n2 < s2*n1) ...
```

Moyen 5 : Ne pas calculer dans les cas où c'est inutile

```
>>a=sqrt(3);                     >>if (x==0) return 0;
>>if (x==0) return 0; else return a; >>else return sqrt(3);
>>
```

Moyen 6 : Dérouler une boucle

- Idée : diminuer le nombre d'itérations d'une boucle par une répétition des lignes de code (p. ex. grouper les itérations par 2). Qu'est-ce qu'on gagne ?

```
>>for (i=0; i<6; i++)           >>s += t[0]; s += t[1]; s += t[2];
>>    s += t[i];                >>s += t[3]; s += t[4]; s += t[5];
>>                                >>
>>for (i=0; i<n; i++)           >>for (i=0; i<n; i+=2)
>>    s += t[i];                >>    s += t[i] + t[i+1];
>>
```

A discuter...

Moyen 7 : Réduction des tests par sentinelles

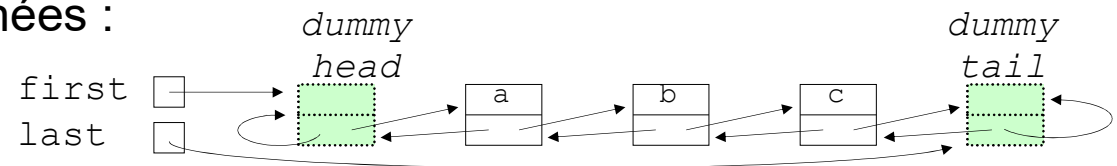
- Idée : limiter le nombre de tests dans les boucles en ajoutant une donnée *spéciale* à l'extrémité (mettre du "contrôle" dans des "données")

```
boolean isIn(int[] t,int e){  
    int i=0;  
    while(i<n && t[i] != e)  
        i++;  
    return (i<n);  
}
```

```
boolean isIn(int[] t,int e){  
    int i=0;  
    t[n] = e; // t.length==n+1...  
    while(t[i] != e)  
        i++;  
    return (i<n);  
}
```

- Donc plus généralement, changer la dernière case, puis la restituer après la boucle

- Sentinelle dans les structures chaînées :
(simplifie insert/remove)



```
void remove(LNode n) {  
    if(n.prev!=null) n.prev.next=n.next;  
    else list.first=n.next;  
    if(n.next!=null) n.next.prev=n.prev;  
    else list.last=n.prev;  
}
```

```
void remove(LNode n) {  
    n.prev.next=n.next;  
    n.next.prev=n.prev;  
}
```

Moyen 8 : Optimiser pour le cas le plus fréquent ("know your data")

- Accélérer le cas fréquent, quitte à ralentir le cas rare

```
    ...  
    res += Math.pow(x, y);
```

```
    if (x==1.0)  
        res += 1.0;  
    else  
        res += Math.pow(x, y);
```

Moyen 9 : Remplacer le compilateur

- Idée : programmer dans un langage de bas niveau (assembleur, C)
- Java Native Interface (JNI) : passerelle entre Java et C

Et encore... (cf. aussi compléments online)

- Le multi-thread est impératif pour exploiter la puissance des machines modernes...
- Qualité de la machine virtuelle
Exécution avec ou sans JIT ? (cf. Just-In-Time compilers; >man java)

- ```
for(...) b[i]=a[i]; ↔ System.arraycopy(...);
String ↔ StringBuffer
double ↔ int
```

- Pseudo-pointeurs; chaque allocation dynamique (malloc) est coûteuse
- Pourquoi y aurait-il une (grosse !) différence entre ces 2 codages ?

```
for (i=0;i<n;i++)
 for (j=0;j<m;j++)
 sum+=t[i][j];
```

```
for (j=0;j<m;j++)
 for (i=0;i<n;i++)
 sum+=t[i][j];
```



- Exceptions, synchronized, récursivité. I/O buffering.
- Utiliser des bibliothèques de "calcul scientifique" disponibles (LAPACK, ...), optimisées
- S'adapter au processeur (data alignment, instructions machine)

## Calculs parallèles au niveau du processeur

- Avec nos machines Multi-Core, la découpe en plusieurs threads est indispensable pour exploiter toute la puissance disponible
- Plusieurs opérations par cycle (Vector unit, Float-Multiply-Add unit)
- Séparer les tâches I/O-bounded et les CPU-bounded
- GPU (carte graphique comme un coprocesseur)

Séquence de calculs indépendants (parallélisables).

```
for (i=0;i<n;i++)
 sum += t[i]*u[i];
```

```
for (i=0;i<n;i+=2) {
 a += t[i]*u[i];
 b += t[i+1]*u[i+1];
}
sum = a+b;
```

- Architecture pipeline : chaque embranchement est coûteux: éviter les 'if' (le CPU tente de prédire la "bonne" branche...)

Fusionner plusieurs boucles successives

(s'assurer que ça ne change pas la sémantique !)

```
for (i=0;i<n;i++) f();
for (i=0;i<n;i++) g();
```

```
for (i=0;i<n;i++) {
 f();g();
}
```

# Programmation parallèle/distribuée

- Idée : profiter de plusieurs CPU en parallèle (machines dédiées, cluster de PCs)
- Compromis entre gain de parallélisation et coût (overhead) lié à la communication et synchronisation
- Réfléchir d'abord à l'optimisation, ensuite à la parallélisation de la version optimisée
- Identifier les traitements ou les données parallélisables
- Equilibrer les taux de charge entre les CPUs
- La programmation parallèle/distribuée, c'est un art, un vaste domaine...
- Types d'architectures matérielles :
  - SIMD : Single Instruction Multiple Data      Contrôle synchrone
  - MIMD : Multiple Instruction Multiple Data      Contrôle asynchrone
- Plateformes de programmation parallèle/distribuée :
  - OpenMP : modèle de mémoire partagée (shared memory)
  - PVM, MPI : modèle d'envoi de messages (message passing)  
(distributed memory)
  - CUDA, OpenCL (programmation de GPU)
- Grid computing et les plateformes existantes (Globus, etc.)

## Au-delà de l'optimisation...

- L'optimisation est une technique bien délimitée, à laquelle on recourt dans des situations très précises
- L'optimisation, c'est une recherche constante d'amélioration du code, par essais successifs. ***Le code "le plus rapide" n'existe pas !*** *TANSTATFC*
- Il y a une autre forme de *recherche constante d'amélioration du code* : la lisibilité, clarté du codage ! Celle-là, elle doit s'appliquer tout le temps !
- Et là, il n'y a hélas pas de mesure quantitative du gain obtenu !  
Ni même de mesure quantitative pour détecter le problème  
Alors que le problème entraîne des coûts bien réels
- Analogie avec les expressions mathématiques :

$$5(a+b) = a+b+75-b+2a+2a+75+2a-a+4b-150$$

Un code source aussi, ça peut se *factoriser*...

|                                                                                                                                       |                                                                                                                                                      |
|---------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>int power(int x, int y) {<br/>    int r=1;<br/>    if (y==0) return 1;<br/>    while(y&gt;0) r*=x;<br/>    return r;<br/>}</pre> | <pre>if (isHorizontal)<br/>    ... // 30 lignes<br/>else<br/>    ... // 30 lignes, mais<br/>        // la dernière<br/>        // différente !</pre> |
|---------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|

- Attention, "clarté" ne veut pas forcément dire "nbre de lignes minimum"

## Backtracking, espace de solutions, arbres de décision,

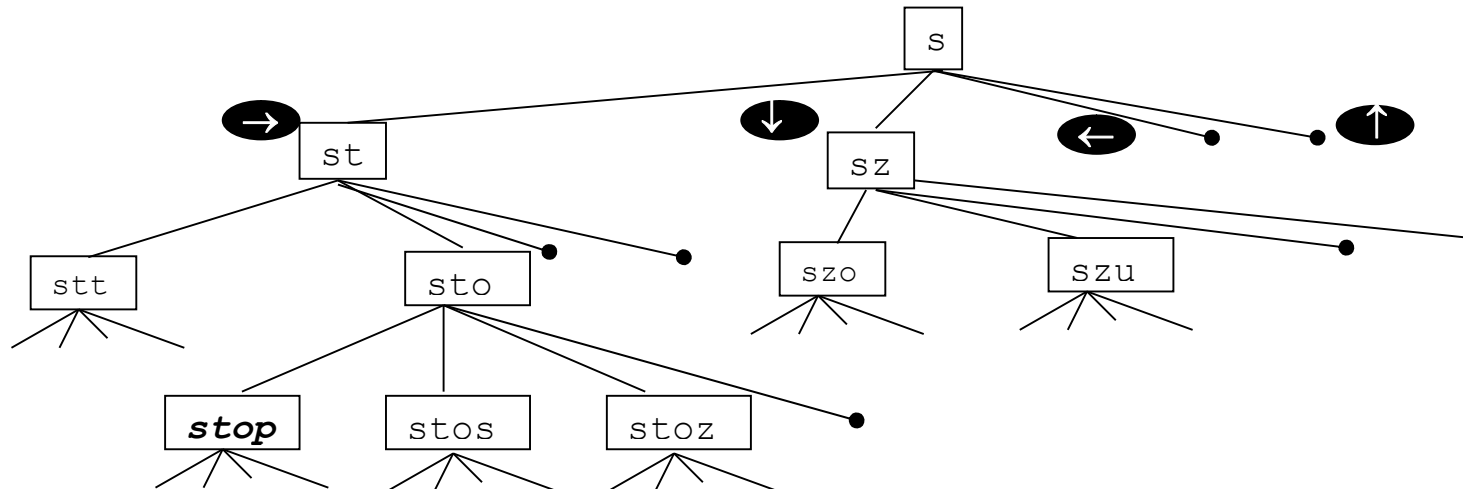
- Plusieurs problèmes reviennent à trouver un chemin dans un arbre de décision, à explorer un espace de solutions
- Le but peut être de trouver *une* solution, *toutes*, ou bien *la meilleure*

### Exemple 1 : les mots croisés Boggle

- Trouver des mots du dico, en "serpentant"

|   |   |   |
|---|---|---|
| s | t | t |
| z | o | p |
| u | s | y |

- Espace de solutions : "st", "sz", "stt", "sto", "szu", "szo", "stop", "stos", "stoz"...
- Arbre de décision ( $\rightarrow \downarrow \leftarrow \uparrow$ )



## Comment parcourir les solutions

- Parcours exhaustif, généralement en profondeur d'abord
- *Backtracking* : utiliser la récursivité pour énumérer toutes les configurations (parcours en profondeur d'abord)
- Eviter les boucles infinies (p. ex. mémoriser les configurations rencontrées)
- Si on ne trouve pas, on "revient en arrière" jusqu'au dernier embranchement où il reste des décisions inexplorées
- Le codage peut alors être assez simple

## Comment parcourir de façon intelligente

- Si on cherche *une* solution : recherche exhaustive, et arrêt si on trouve
- Souvent la recherche exhaustive de la *meilleure* solution n'est pas praticable
- *Pruning* : utiliser une information pour abandonner les branches non intéressantes (couper, élaguer). (pour Boggle : test du préfixe)
- *Branch-and-Bound* : utiliser comme seuil la meilleure solution rencontrée (Boggle modifié : trouver le mot qui minimise la somme pondérée des lettres)
- L'intelligence artificielle (IA) étudie des heuristiques pour mieux explorer l'arbre
- Certains langages de programmation sont construits autour de la notion de backtracking

## Exemple 2 : jeux de stratégie, à information complète, pour 2 joueurs

- Exemples : échecs, dames, othello, tic-tac-toe, puissance-4, ...
- De stratégie : le hasard n'intervient pas dans le jeu  
A information complète : aucune information n'est cachée à un joueur  
(aussi appelés jeux *combinatoires*)
- On parlera de *configuration* pour l'état d'une partie (pièces sur l'échiquier)
- Certaines configurations sont *terminales* (partie gagnée par A ou B, égalité)
- A et B jouent à tour de rôle. Pour calculer le meilleur coup pour A, on suppose que B joue aussi de façon rationnelle
- On définit une mesure pour toute configuration terminale (nombre de points) : plus elle est grande, plus A est "gagnant" (et donc plus B est "perdant")  
Ça peut être simplement 0/1, ou bien le jeu définit le gain (nbre de pièces restantes)
- A (B) doit jouer. Il envisage chaque coup. Si on lui prédit la conséquence (mesure terminale) pour chaque coup, il choisira le maximum (minimum)  
Algorithme récursif (à tour de rôle, on minimise/maximise)
- Parfois, tout explorer jusqu'aux configurations terminales est trop coûteux. On impose alors une profondeur limite, où se fait une estimation de la mesure (p. ex. somme pondérée des pièces de chaque joueur). On parle de configuration *quasi-terminale*
- Un cas spécial : les arbres de décision ET/OU : mesures booléennes (gagné/perdu)
- Avoir une stratégie gagnante = une certaine suite de coups nous assure la victoire



## Stratégie mini-max

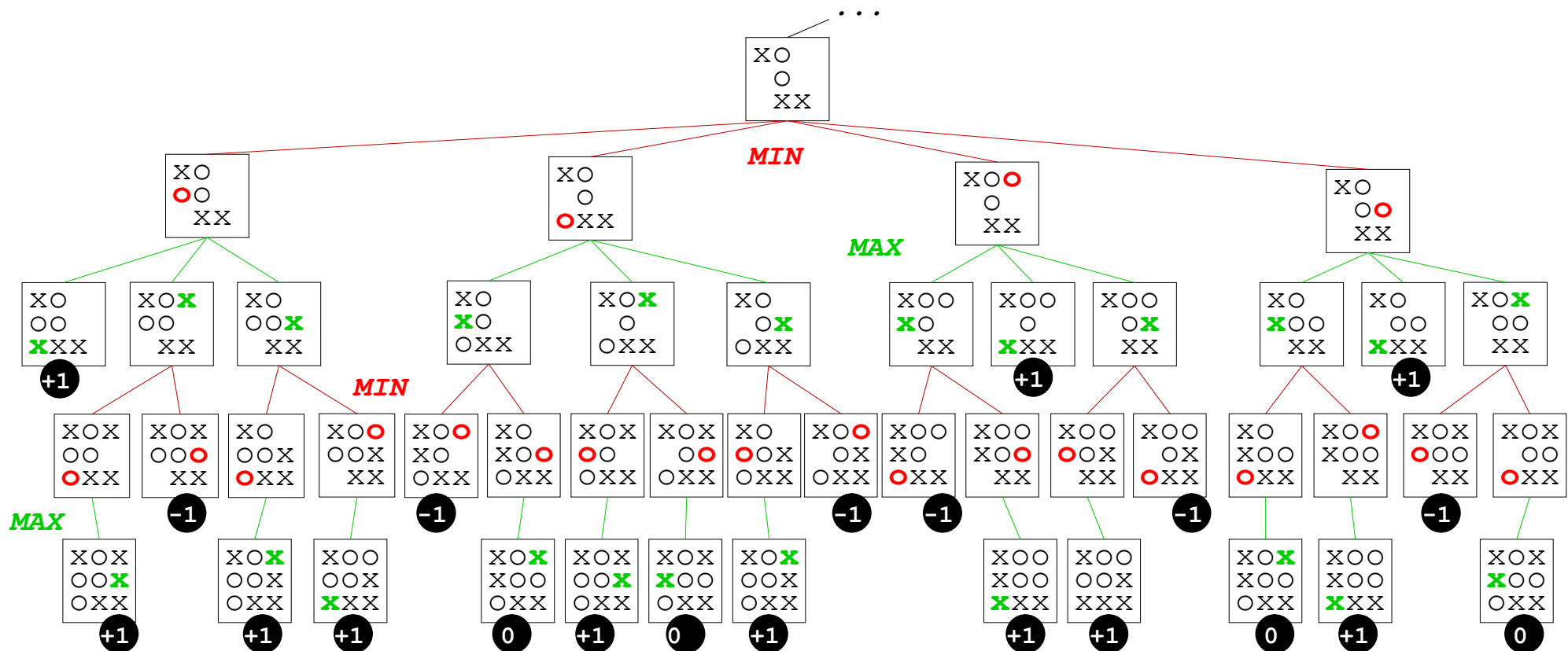
- Algorithme normal de backtracking. Pseudo-code :

```
int expectedScore(boolean joueurA, Object config) {
 if (config est (quasi-)terminale)
 return mesure(config);
 if (joueurA) best= -∞;
 else best= +∞;
 for each coup possible CP {
 changer la config pour jouer le coup CP
 m = expectedScore(!joueurA, config)
 remettre la config comme avant (défaire CP)
 if (joueurA) best=max(best,m);
 else best=min(best,m);
 }
 return best;
}
```

- Cette routine retourne le meilleur score possible. Il faut l'adapter pour retourner aussi le coup qui permet ce meilleur score !

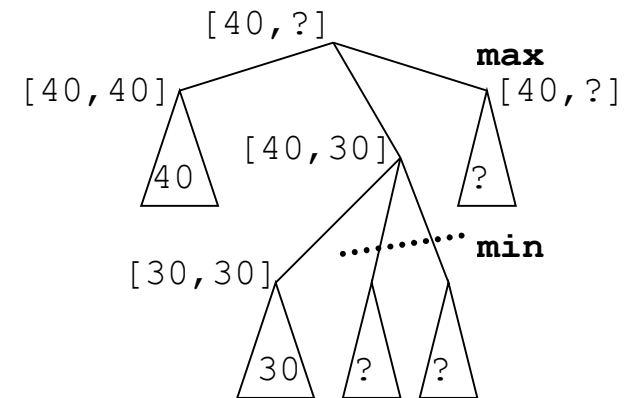
## Exemple : Tic-Tac-Toe

- Voici un extrait de l'arbre de décision depuis une configuration donnée
- A dessine les croix (x) et gagne avec la mesure +1
- Dans quel état va se terminer le jeu ?



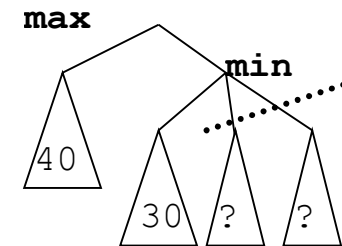
## Stratégie alpha/beta

- Une meilleure stratégie : "pruning" par "branch-and-bound"
- Au fil de l'exploration de l'arbre, on retient l'intervalle des valeurs qui restent intéressantes
- Alpha : meilleur score trouvé pour A  
Beta : meilleur score trouvé pour B
- On peut alors couper bien des branches...
- Initialisation :  $[\alpha, \beta] = [-\infty, +\infty]$

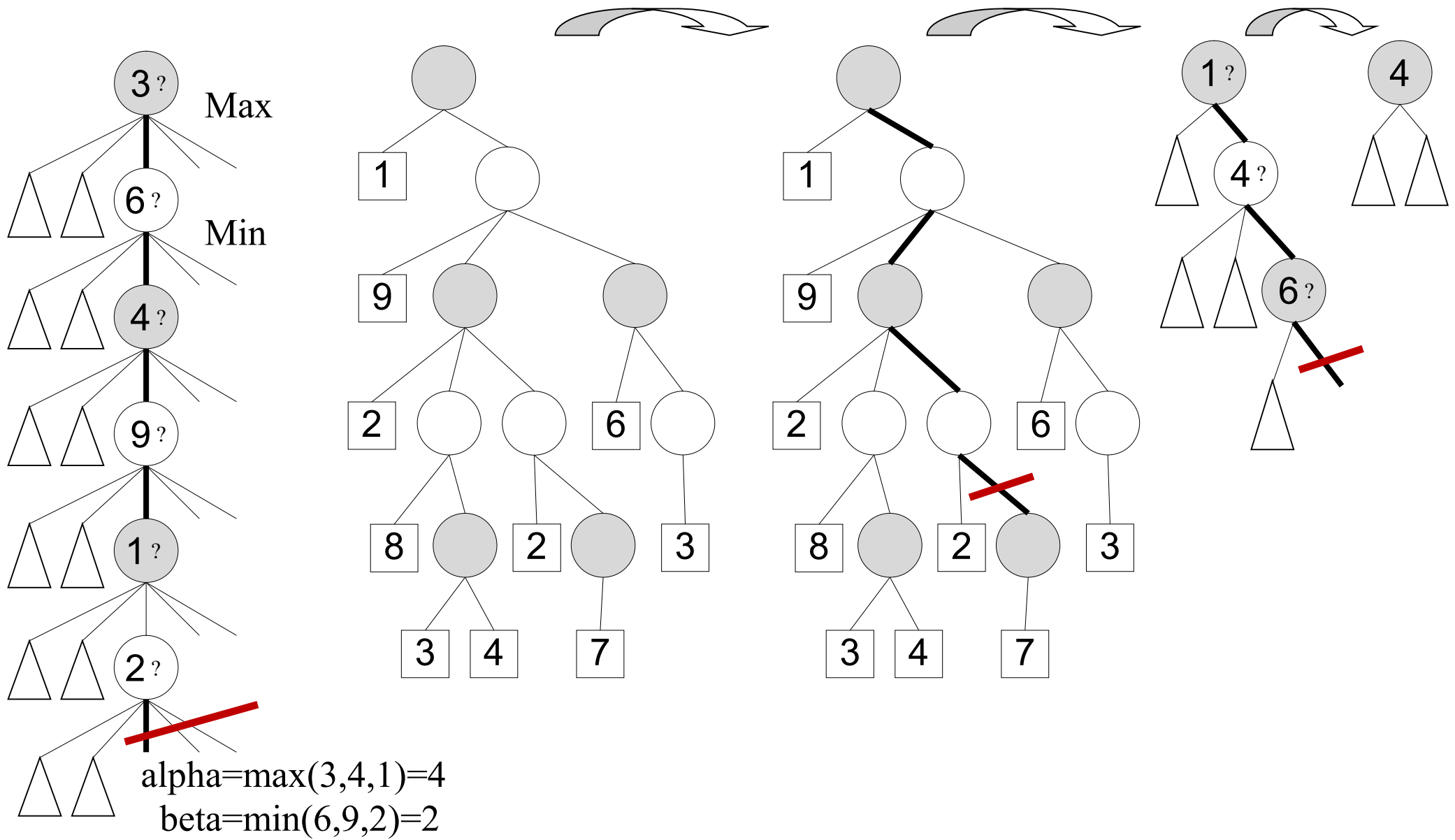


- Petite modification au pseudo-code :

```
int expectedScore(..., int alpha, int beta){
 ...
 for each coup possible CP {
 changer la config (jouer CP)
 m= expectedScore(!joueurA, config, alpha, beta);
 remettre la config comme avant (défaire CP)
 if (joueurA) best=max(best,m); alpha=max(alpha,m);
 else best=min(best,m); beta=min(beta ,m);
 if (alpha >= beta) break; // pruning
 }
 ...
}
```



## Alpha/beta : illustration, exemple...



## Comment éviter de recalculer les mêmes choses

- Il arrive qu'on rencontre plusieurs fois la même configuration (ou une configuration "symétrique") dans l'arbre de décision. Suivant les jeux, ça peut devenir un problème
- Principe de la programmation dynamique : mémoriser au lieu de recalculer
- Mais "toute une configuration" ne peut pas directement servir d'indice dans un tableau : on construit un dictionnaire (p. ex. par hachage).  
Dans ce contexte, on parle de *table de transposition*
- Avant de faire un appel récursif, tester si la configuration est dans la table

## Exercice : le jeu MaxIt ([Weiss10], ex. 10.10, p. 472)

- Soit une grille NxN d'entiers, et une case initiale. A chaque étape, un joueur choisit une case dans la ligne ou colonne courante. La valeur dans cette case est ajoutée au score de ce joueur, cette case ne pourra plus être utilisée, et c'est à l'autre de jouer. A la fin, celui qui a le plus de points gagne.

|   |    |   |   |
|---|----|---|---|
| 3 | 12 | 4 | 6 |
| 1 | 2  | 3 | 4 |
| 9 | 9  | 8 | 2 |
| 4 | 5  | 7 | 6 |

A=0  
B=0

|   |               |   |   |
|---|---------------|---|---|
| 3 | <del>12</del> | 4 | 6 |
| 1 | 2             | 3 | 4 |
| 9 | 9             | 8 | 2 |
| 4 | 5             | 7 | 6 |

A=12  
B=0

|   |               |   |   |
|---|---------------|---|---|
| 3 | <del>12</del> | 4 | 6 |
| 1 | 2             | 3 | 4 |
| 9 | <del>9</del>  | 8 | 2 |
| 4 | 5             | 7 | 6 |

A=12  
B=9

|   |               |   |   |
|---|---------------|---|---|
| 3 | <del>12</del> | 4 | 6 |
| 1 | 2             | 3 | 4 |
| 9 | <del>9</del>  | 8 | 2 |
| 4 | <del>5</del>  | 7 | 6 |

A=17  
B=9

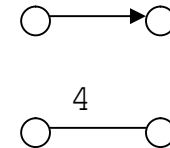
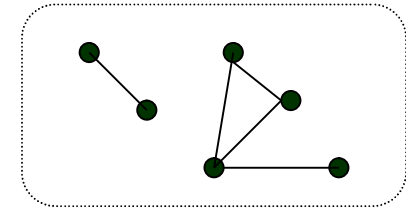
...

# Structure de graphe

- Liens (arc, edge) quelconques entre éléments (sommet, vertex)

## Terminologie

- Notation :  $G = (V, E)$      $e = |E|$      $n = |V|$
- Graphe, sommet (= nœud), arête (arc si orienté). Sous-graphe
- Sommets adjacents (= voisin), degré (in/out) d'un sommet.
- Graphe pondéré (weighted). Graphe orienté
- Graphe connexe. Chemin entre 2 sommets. Cycle (= circuit)
- Graphe acyclique (= simple = sans cycle). Graphes isomorphes
- Autre structure : multi-graphe (plusieurs arcs entre 2 sommets)
- Graphe complet. Graphe bipartite (formé de 2 ss-graphes sans arcs "internes")



## Représentation de graphes

- Un chaînage naïf ne suffit plus : pas de sommet de référence, de "point d'entrée"
- On suppose généralement que chaque sommet possède un numéro unique
- Plusieurs approches possibles, influençant la complexité des opérations :
  - liste de sommets et liste d'arcs
  - matrice d'adjacence : à chaque couple de sommets, la description du lien
  - listes d'adjacence : à chaque sommet, la liste des voisins

- Suivant l'approche, on exprime les complexités en fonction de  $e$  ou de  $n$
- Voir aussi représentation des matrices (matrices creuses...)

### Une spécification possible en Java

- Déjà pas de consensus sur les listes, alors pour les arbres et graphes...
- Chaque sommet a un numéro ( $0 \leq \text{vid} < n$ ).  
Le nombre de sommets est fixé une fois pour toutes (constructeur)
- Possibilité de stocker une *couleur* (Object) sur les sommets et les arcs

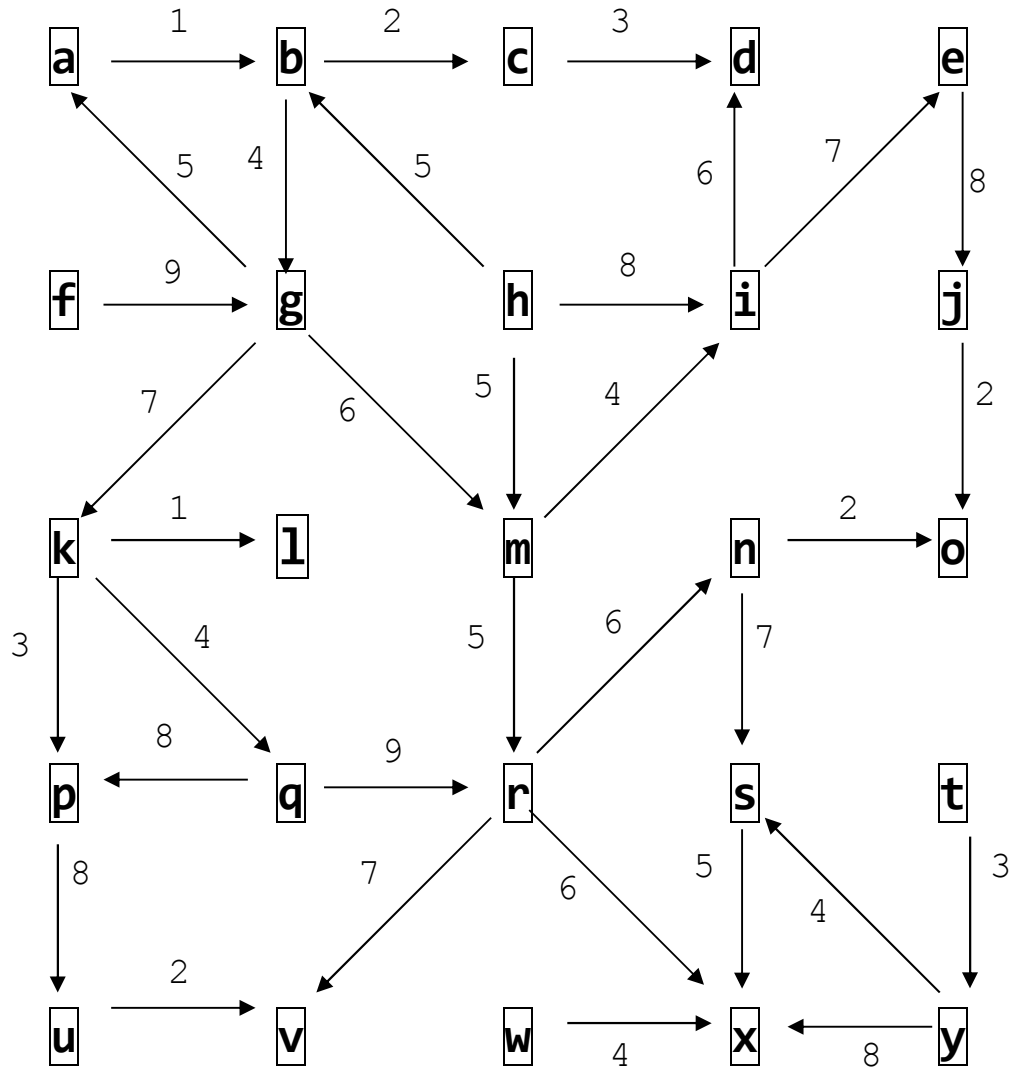
```

public class DirectedGraph {
 DirectedGraph(int nbOfVertices);
 int nbOfVertices(); // vertex IDs in [0..n-1]
 int nbOfEdges ();
 void addEdge (int fromVid, int toVid);
 void removeEdge (int fromVid, int toVid);
 boolean isEdge (int fromVid, int toVid);
 int inDegree (int toVid);
 int outDegree (int fromVid);
 int[] neighboursFrom(int fromVid);
 int[] neighboursTo (int toVid);
 setVertexData(int v, Object c);
 getVertexData(int v);
 getEdgeData(int from, int to, Object c);
 getEdgeData(int from, int to);
}

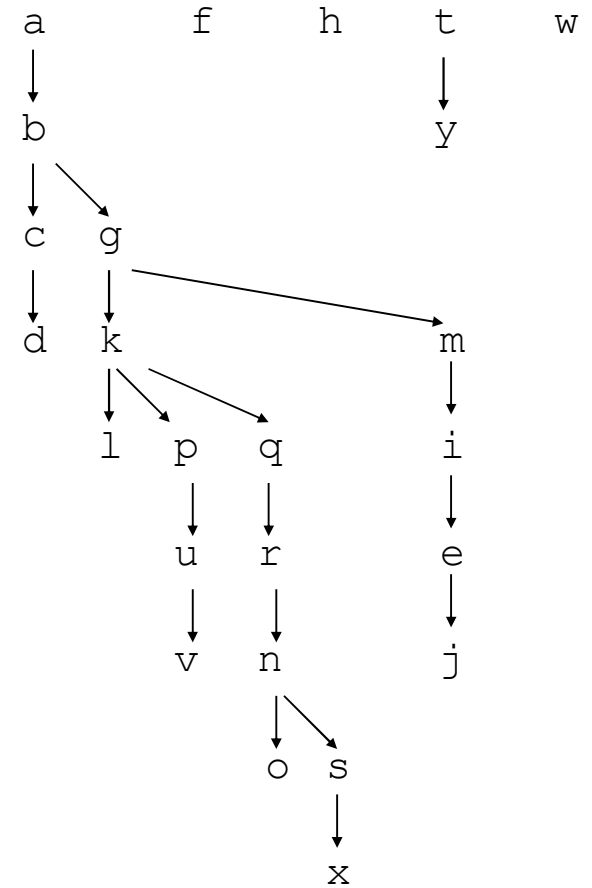
public class WeightedDiGraph extends DirectedGraph {
 void setEdgeWeight(int fromVid, int toVid, int weight);
 int getEdgeWeight(int fromVid, int toVid); ...}

```

## Exemple de graphe



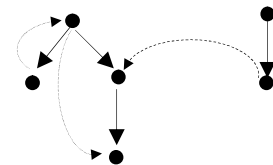
## Forêt de parcours en profondeur





## Parcours de graphe

- Comment traverser systématiquement tous les sommets d'un graphe ?
- Opération de base pour plusieurs autres problèmes
- Principe : construire une "forêt de parcours (recouvrement)" (spanning forest) : un sous-graphe acyclique avec tous les sommets
- Construction de la forêt : A chaque étape, il y a la collection d'arcs candidats en suspens, avec un ordre de préférence
- 3 variantes :
  - en profondeur d'abord                      traiter récursivement chaque voisin
  - en largeur d'abord                      traiter chaque voisin, puis les voisins des voisins...
  - en meilleur d'abord                      notion de priorité d'un arc candidat
- En profondeur : pré-ordre ou post-ordre  
En largeur = par niveau (level-order)
- Soit un graphe orienté  $G$  et une forêt de recouvrement possible. On définit 4 types parmi les arcs de  $G$  :
  - arcs d'arbre                      (liens père  $\rightarrow$  fils)
  - arcs avant                      (liens père  $\rightarrow$  descendants d'un fils)
  - arcs arrière                      (liens fils  $\rightarrow$  ancêtres)
  - arcs transversaux                      (tous les autres liens)



## Calcul du plus court chemin depuis un sommet, graphe non pondéré

- Parcours en largeur d'abord, en incrémentant un compteur à chaque niveau

## Codage des parcours

- Il est nécessaire de mémoriser quels sommets ont déjà été visités (ensemble, tableau de booléens)
- Si nécessaire, on peut construire une représentation de la forêt de parcours (p. ex. tableau des parents).

```
void depthFirst(Graph g, int startVid)
 boolean[] isVisited = new boolean[n];
 for i in startVid, 0, 1, 2, ..., n
 depthFirst(g, i, isVisited, -1);

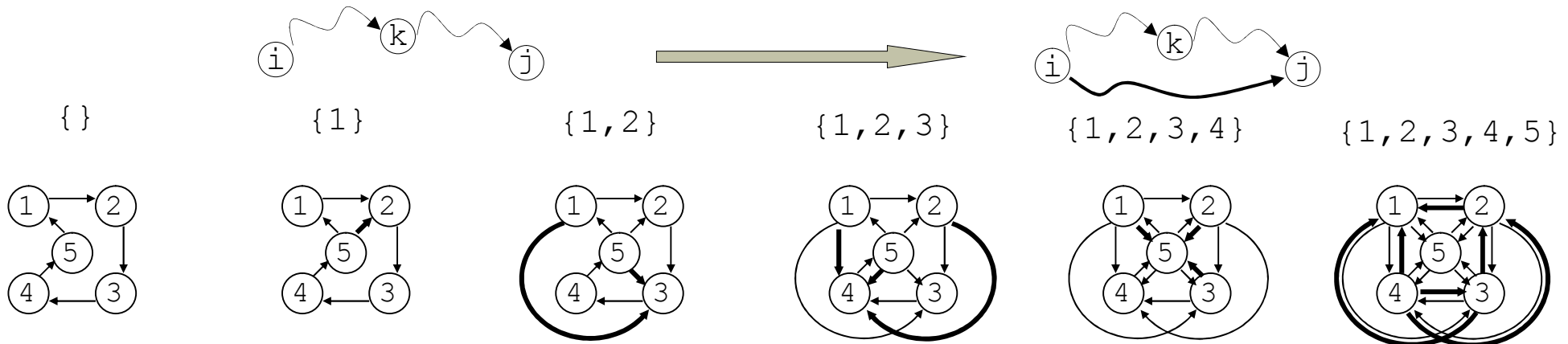
void depthFirst(Graph g, int vid, boolean[] isV, int dad)
 if (isV[vid]) return;
 isV[vid]=true; visit(vid); // parents[vid]=dad;
 for each of its neighbors v
 depthFirst(g, v, isV, vid);
```

- Ici, on voit le graphe comme type abstrait. Le modèle doit être adapté suivant le contexte (des machines en réseau où personne ne connaît le graphe global, et où "supprimer un arc" nécessite un accord entre les 2 sommets)

## Détection de cycle, graphe orienté, fermeture transitive

- Détection de cycle : parcours en profondeur d'abord, et détection d'un arc arrière (mémoriser le parent dans le parcours)
- Fermeture transitive de la matrice d'adjacence  $M = M * M * \dots * M = M^n$  en  $O(n^4)$
- La **fermeture transitive** d'un graphe orienté  $G$  est un autre graphe orienté  $T$  tel que : il y a l'arc  $(i \rightarrow j)$  dans  $T$  s'il y a un chemin de  $i$  vers  $j$  dans  $G$
- Algorithme de Warshall (utilise la programmation dynamique) en  $O(n^3)$  :

```
boolean[][] isReachable = new boolean[n][n];
for (i=0; i<n; i++) for (j=0; j<n; j++)
 isReachable[i][j] = g.isEdge(i, j); // isReachable directly
for (k=0; k<n; k++) // isReachable using vertices 0..k
 for (i=0; i<n; i++) for (j=0; j<n; j++)
 isReachable[i][j] |= isReachable[i][k] && isReachable[k][j]
// cycle iff isReachable[i][i] exists
// we may also update an array pred[i][j] to hold paths...
```



## Algorithme de Dijkstra

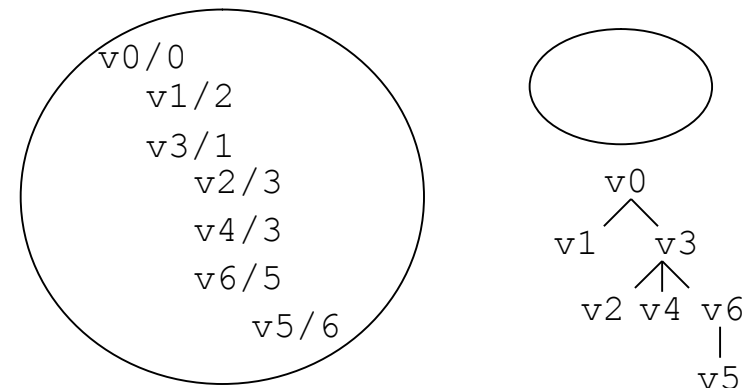
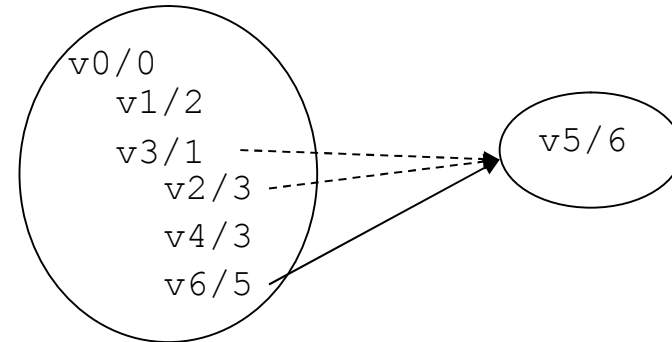
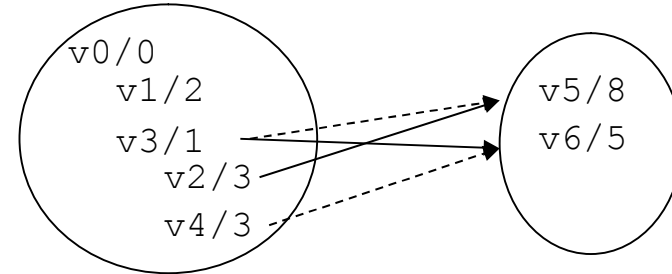
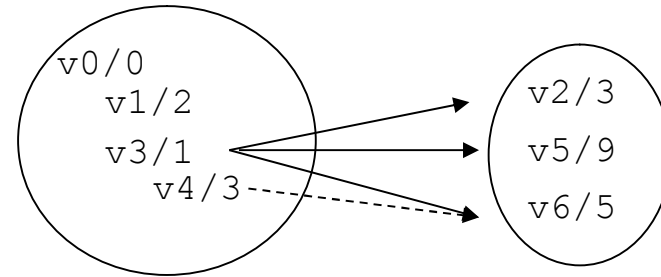
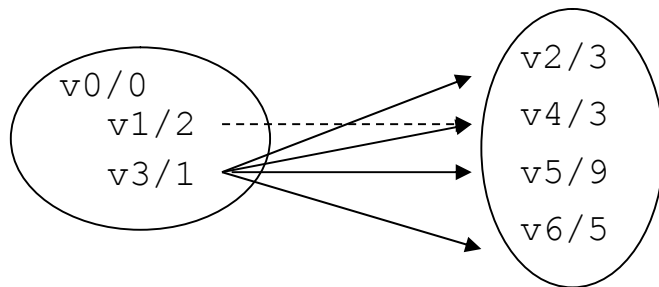
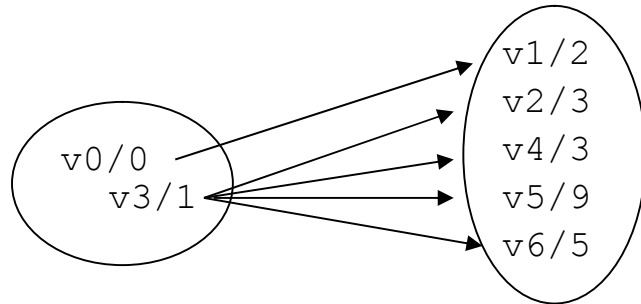
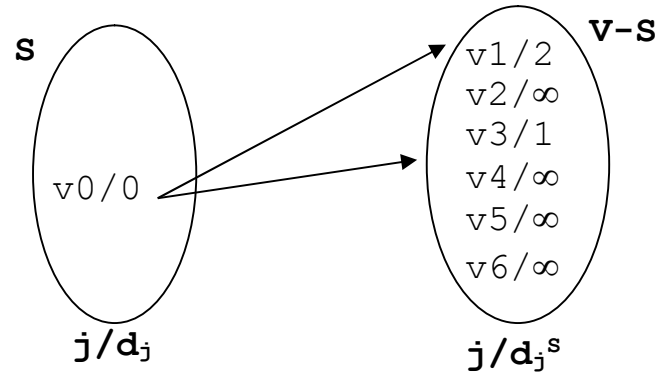
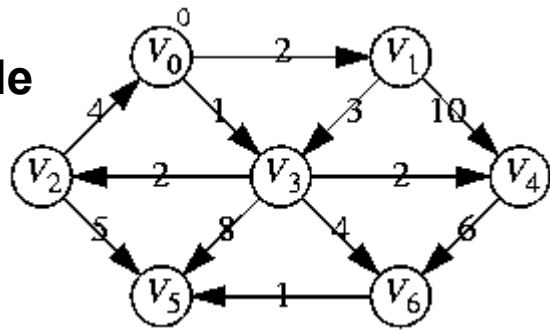
---

- Plus court chemin d'un sommet  $a$  vers tous les autres
- Soit un graphe  $G=(V,E)$ , orienté, pondéré (coûts  $w_{ij}$ ). Soit un sommet de départ  $a$ . Pour tous les sommets  $j$ , on cherche  $d_j$  le coût du meilleur chemin de  $a$  vers  $j$ .
- Hypothèse : uniquement des arcs avec poids positifs.  
(si on tolère des poids négatifs, il faut un autre algorithme (Bellman-Ford), capable de détecter les "cycles négatifs")
- Soit  $S$  l'ensemble des sommets déjà visités. Au départ  $S=\{a\}$ , finalement  $S=V$
- Soit  $d_j^S$  le coût du meilleur chemin de  $a$  vers  $j$  ne passant que par des sommets de  $S$
- Comment garantir qu'une fois visité, un sommet aura sa solution définitive ?  
en choisissant à chaque étape le sommet  $j$  de  $V-S$  qui minimise  $d_j^S$ .
- Dans ce cas, on a :  $d_j^{S \cup \{k\}} = \min(d_j^S, d_k^S + w_{kj})$

### Implémentation

- Comment gérer la collection de sommets non visités ( $V-S$ ) ?
- En file de priorité ! Mais la mise à jour des priorités est gênante...
- Solution :
  - mettre à jour = ajouter un nouvel élt
  - gérer séparément l'indication `isVisited[j]`

## Example



- Pseudo-code

```

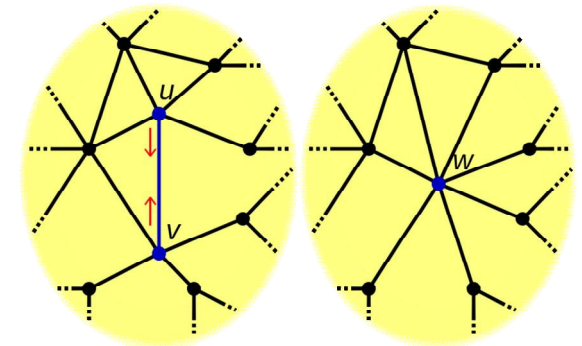
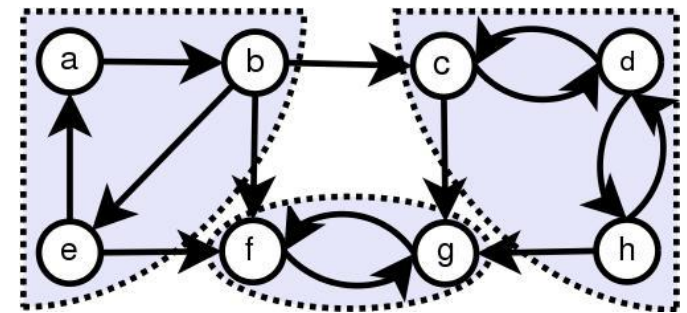
void dijkstra(WeightedGraph g, int a, int[] d) {
 d = {∞, ∞, ∞ ...};
 boolean[] isVisited = {false, false, false...};
 PtyQueue pq;
 d[a] = 0;
 pq.enqueue(a, d[a]);
 while (!pq.isEmpty())
 int k = pq.dequeue();
 if (isVisited[k]) continue;
 isVisited[k] = true;
 for each neighbor i of k
 d[i] = min(d[i], d[k] + g.getEdgeCost(k, i));
 pq.enqueue(i, d[i]); // useless if d[i] not updated
 }

```

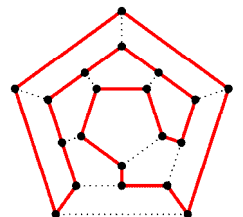
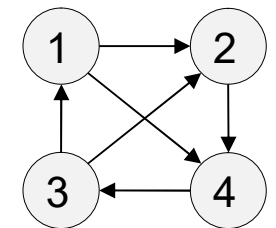
- Si on cherche uniquement le meilleur chemin entre a et b, on arrête dès que b devient visité. Si on veut ressortir le chemin en plus du coût total, on peut le stocker dans un tableau parent[j], correctement mis à jour
- Complexité (si on fait les bons choix) :  $O(e \ln e) = O(e \ln n)$
- La recherche du plus *long* chemin (sans cycle) entre 2 sommets n'est pas aussi efficace... (complexité apparemment exponentielle !)

## Condensation, tournois, chemins hamiltoniens/eulériens...

- Dans un graphe orienté, un ensemble de sommets est dit "**fortement connexe**" s'il existe un chemin entre tout couple de sommets. Quand l'ensemble est maximal, on parle de **composantes fortement connexes** (strongly connected components).
- La **contraction** d'un arc ou arête est une transformation du graphe par laquelle les deux sommets concernés sont "fusionnés" en un seul (qui récupère tous les arcs entrants ou sortants des deux sommets).
- La **condensation** d'un graphe orienté est le DAG obtenu après avoir contracté toutes ses composantes fortement connexes. La complexité de cette opération est linéaire ( $O(n+e)$ , grâce à un parcours de graphe).
- Un **tournoi** est un graphe orienté où chaque paire de sommets est reliée par un et un seul arc. Sa **suite de scores** est la liste triée des out-degrees (nombre d'arcs sortants) de tous les sommets.
- Un chemin/cycle **eulérien** visite chaque *edge* exactement une fois. (*chinese postman problem* : cycle qui visite chaque *edge* au moins une fois)
- Un chemin/cycle **hamiltonien** visite chaque sommet exactement une fois.



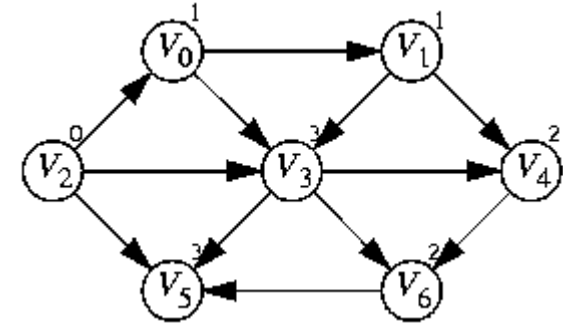
[some figs. from Wikipedia]



## DAG, tri topologique, ordonnancement

---

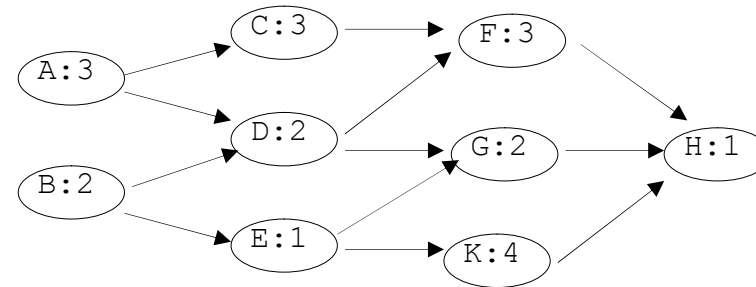
- Une famille de graphes intéressante : les Directed Acyclic Graphs (DAG)
- Une application : Relations d'ordre *partiel* (relations de préférence)
- **Tri topologique** :  
transformation en un *ordre total* compatible  
(une liste telle que :  $x$  apparaît avant  $y$  s'il y a un arc  $x \rightarrow y$  dans le graphe)
- Il existe toujours au moins 1 ordre total compatible dans un DAG
- Dans cet exemple, il n'y en a qu'un :  $V_2 - V_0 - V_1 - V_3 - V_4 - V_6 - V_5$
- Une solution :
  - parcours en profondeur d'abord depuis n'importe quel sommet
  - appliquer un parcours post-ordre sur chaque arbre de la forêt
  - concaténer les sous-listes (dans l'ordre d'apparition dans la forêt)
  - inverser le résultat
- Autre application : Problèmes d'ordonnancement (projets de constructions) :  
un ensemble de tâches  $t_1 \dots t_n$ , avec leurs durées  $d(t_i)$ , et des contraintes du genre " $t_i$  doit commencer après la fin de  $t_j$ "
- Il y a deux représentations différentes basées sur les graphes





- *Graphe d'activité* :

- sommet : tâche
- arête : dépendance

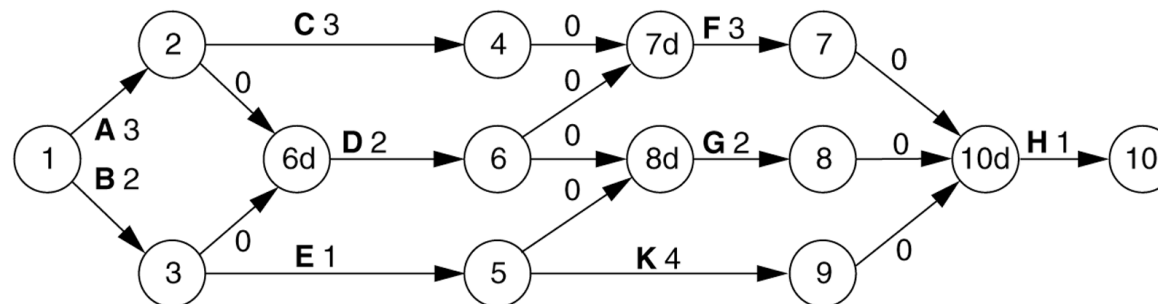


- Exemple :

ici, la tâche D dure 2 unités,  
et doit attendre sur les tâches A et B

- *Graphe d'événement* : mêmes informations, mais autre représentation ("duale")

- sommet : événement (fin d'une tâche) (ou pseudo-événement)
- arête : tâche (ou pseudo-tâche à durée nulle)



- Questions intéressantes (cf. algos dans [Weiss10]) :

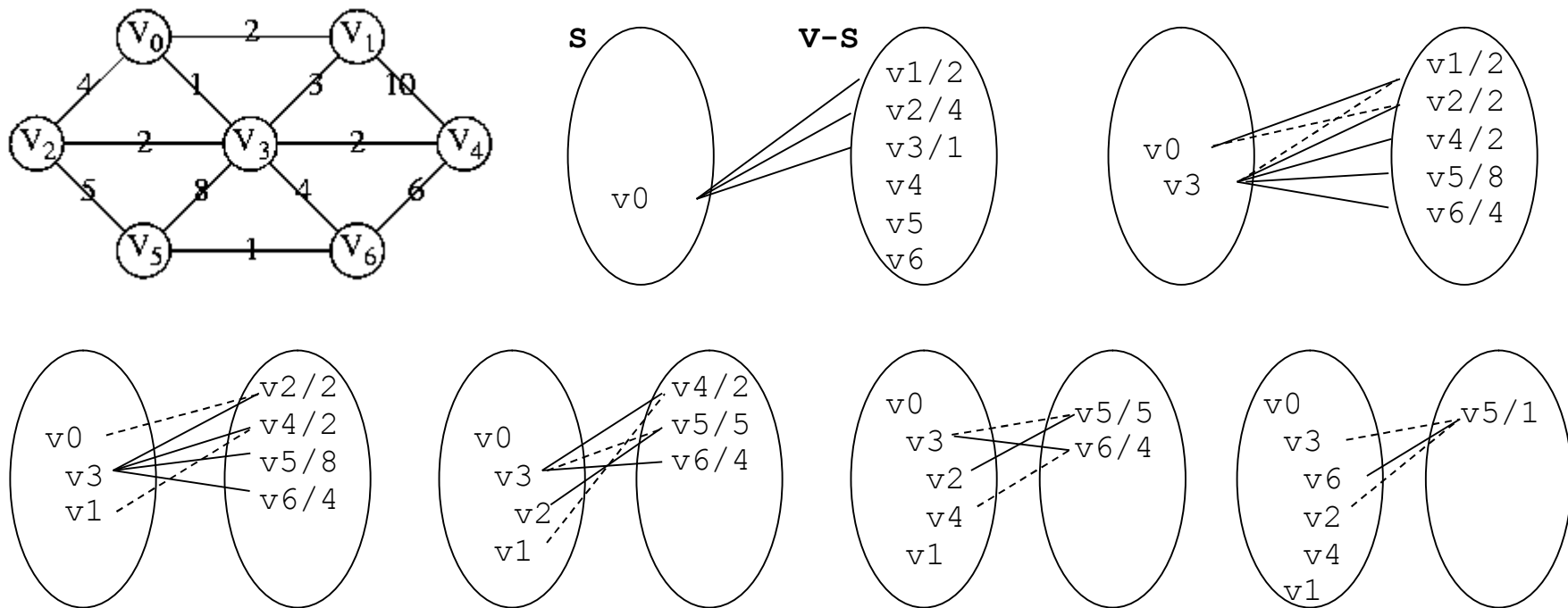
- durée totale minimale (plus long chemin)
- l'instant minimal/maximal de début (et de fin) de chaque tâche
- marge possible pour chaque tâche (retard sans conséquence globale)
- *chemin critique* (formé de tâches à marge nulle)

## Arbre de recouvrement minimal (Minimum Spanning Tree, MST)

- On parle ici de graphes connexes non orientés, pondérés
- Arbre de recouvrement = sous-graphe connexe acyclique avec tous les sommets
- Le *coût* d'un arbre de recouvrement est la somme des poids des arcs
- Problème : trouver un arbre de coût minimal (MST)

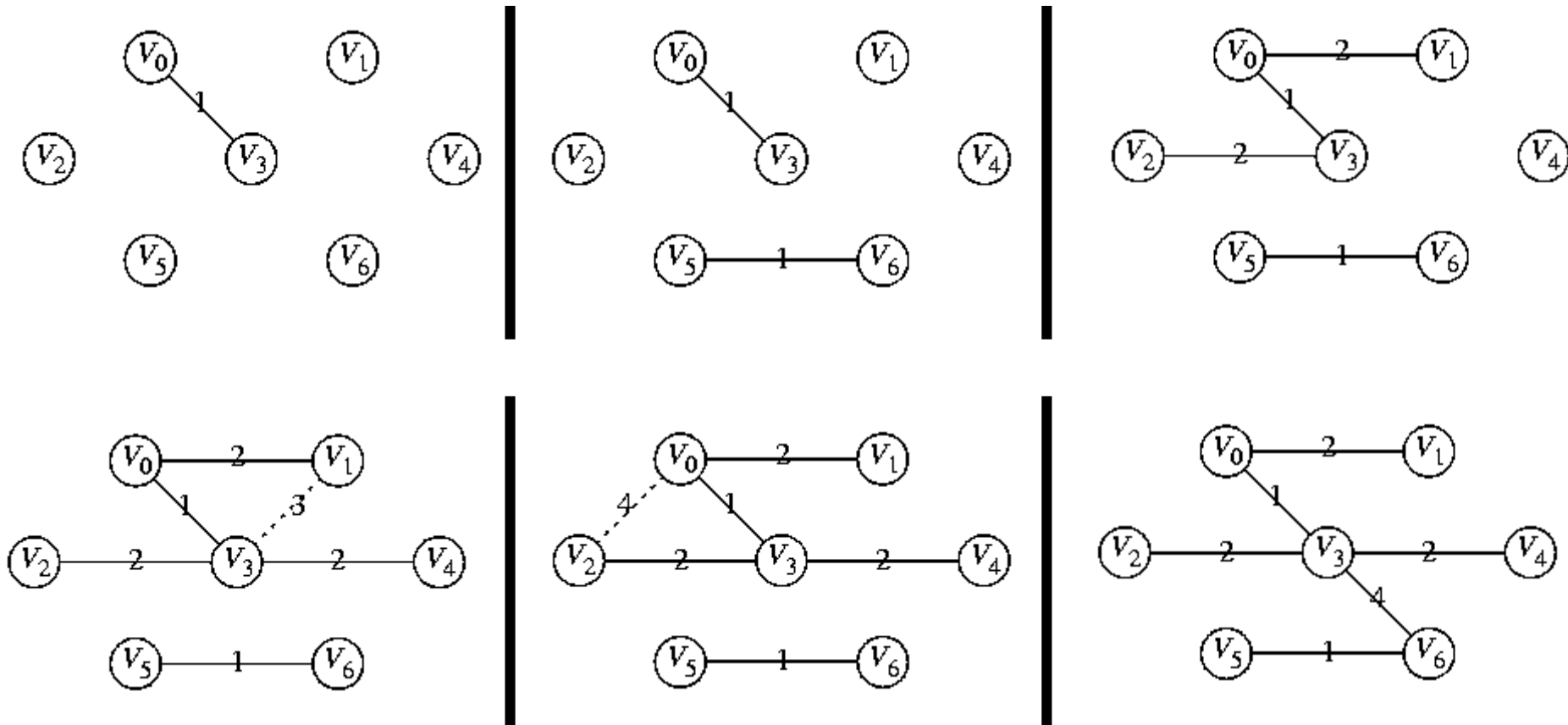
### Algo (glouton) de Prim, $O(e \ln n)$

- Idée : augmenter l'*arbre* courant avec l'arc candidat de poids minimal



## Algo (glouton) de Kruskal, $O(e \ln n)$

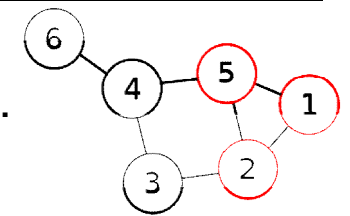
- Idée : ajouter au *sous-graphe* courant l'arc le plus léger qui ne crée pas de cycle
- Comment gérer la collection des sommets, pour détecter les cycles ?  
En ensembles disjoints !



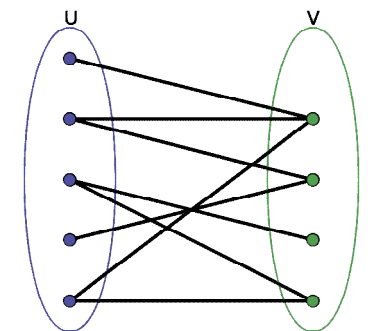
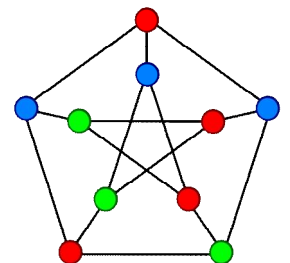
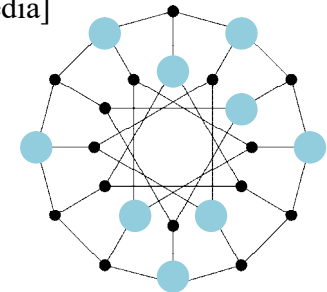
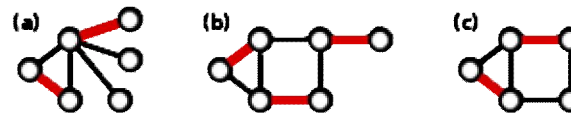
## Graphes non orientés : clique, coloration, et autres...

- Dans un graphe non orienté, une (k-) **clique** est un sous-ensemble de sommets qui sont tous reliés entre eux (sous-graphe complet à k sommets).
- Un "**independent set**" (ou un stable) est un sous-ensemble de sommets n'ayant aucune arête en commun.
- Un "**matching**" est un sous-ensemble d'arêtes n'ayant aucun sommet en commun.
- Une **coloration** de graphe consiste à attribuer une "couleur" (numéro) de façon à ce que deux sommets voisins aient une couleur différente. Le **nombre chromatique** d'un graphe est le nombre minimum de couleurs nécessaires à sa coloration.
- Déterminer les plus grands "independent set" et clique, si un graphe est k-colorable ou trouver un chemin hamiltonien sont des problèmes NP-Complets : les meilleurs algorithmes connus sont exponentiels.
- Un graphe non orienté est dit "**bipartite**" (ou biparti) si on peut partitionner ses sommets en deux "independent sets". C'est équivalent de dire qu'il est 2-colorable, ou encore qu'il n'a pas de cycle de longueur impaire.

Détecter un cycle impair peut facilement se faire par un parcours en profondeur, en vérifiant les différences de niveaux lors d'un arc arrière.

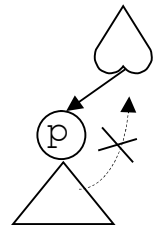
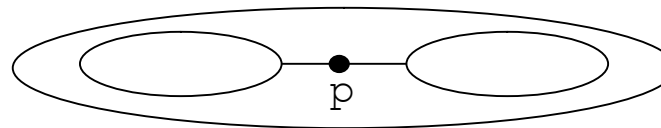


[some figs. from Wikipedia]

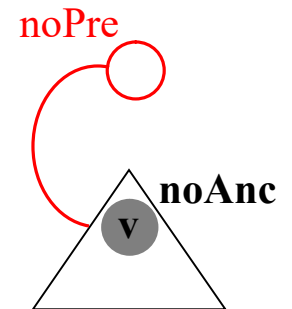


## Connectivité et points d'articulation

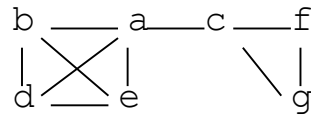
- On parle ici de graphes non orientés, non pondérés
- Réseau de communication, résistance aux pannes, fiabilité...
- Graphe connexe = qui a un chemin entre tout couple de sommets
- Graphe  $k$ -connexe = reste connexe après la suppression de  $x$  sommets quelconques,  $1 \leq x < k$ . Cas particulier : graphe 2-connexe (biconnexe)
- Point d'articulation = sommet dont la suppression rend le graphe non connexe
- Idée : détecter cette "configuration" dans un arbre de parcours



- On peut détecter les points d'articulation par un "simple" parcours en profondeur d'abord. On s'intéresse à l'arbre de parcours depuis un sommet
- Soit  $\text{noPre}(v)$  le numéro de parcours pré-ordre du sommet  $v$
- Soit  $\text{anc}(v)$  le plus lointain ancêtre voisin (y. c. par *arc arrière*) depuis tout le sous-arbre de  $v$ . Alors  $\text{noAnc}(v) = \text{noPre}(\text{anc}(v)) = \min(\dots)$
- Quels sont les points d'articulation ?
  - la racine, si elle a au moins 2 fils
  - un noeud intermédiaire  $v$ , si on a  $\text{noPre}(v) \leq \text{noAnc}(f)$  pour un de ses fils  $f$
  - jamais une feuille

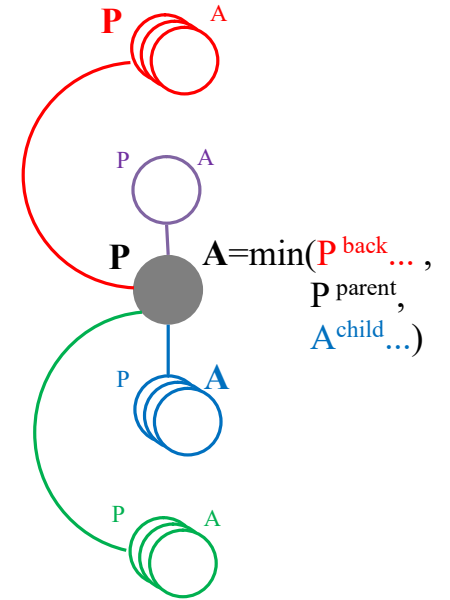
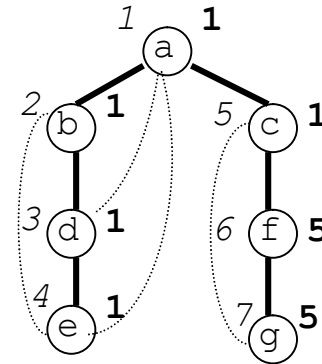


## Exemple



**Points d'articulation :**

a : racine a 2 fils  
c :  $\text{noPre}[c] \leq \text{noAnc}[f]$



## Codage

```
static int preOrderCounter=0;
static int[] noAnc, noPre;
static boolean[] isArtPoint, isVisited;
static int nbOfRootSons = 0;

static void findArticulationPoints(Graph g, int vid, ...) {
 // POST : noAnc[vid] and isArtPoint[vid] are known
 isVisited[vid] = true;
 noAnc[vid] = noPre[vid] = preOrderCounter++; // *
 for each neighbor v of vid
 if (child edge) {
 findArticulationPoints(g, v); // POST : noAnc[v] is known
 if (noAnc[v] >= noPre[vid]) isArtPoint[vid] = true;
 noAnc[vid]=Math.min(noAnc[vid], noAnc[v]); // *
 } else { // parent, forward edge or back edge
 noAnc[vid]=Math.min(noAnc[vid], noPre[v]); // *
 }
 }
}
```

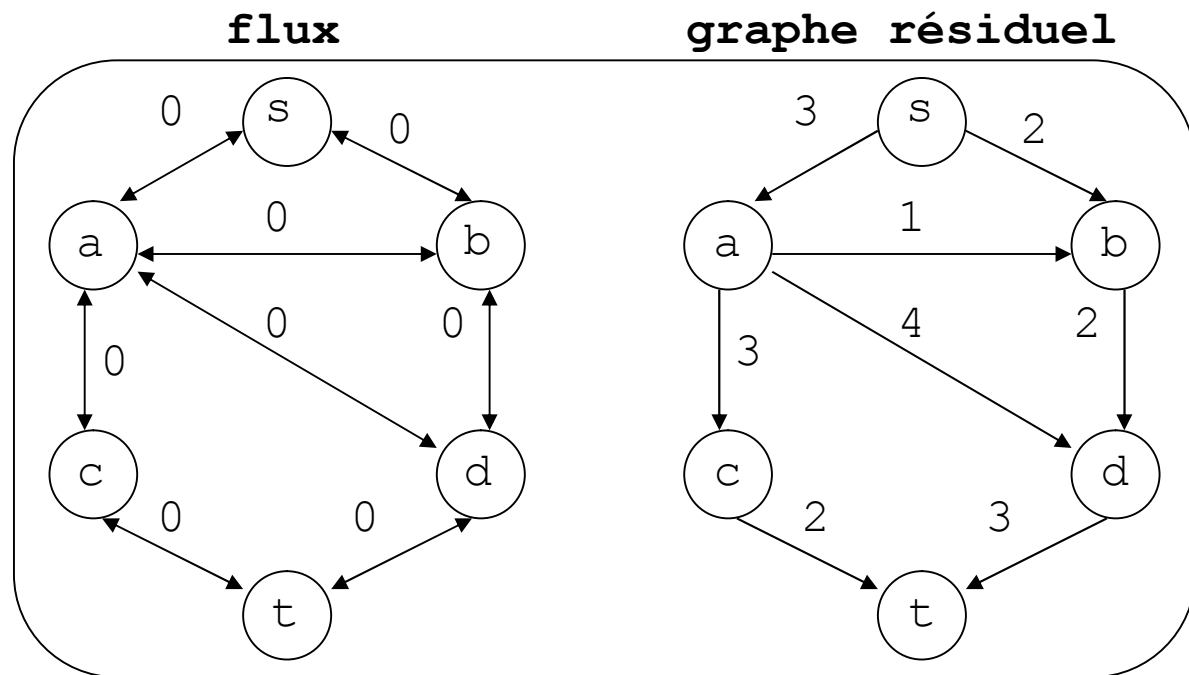
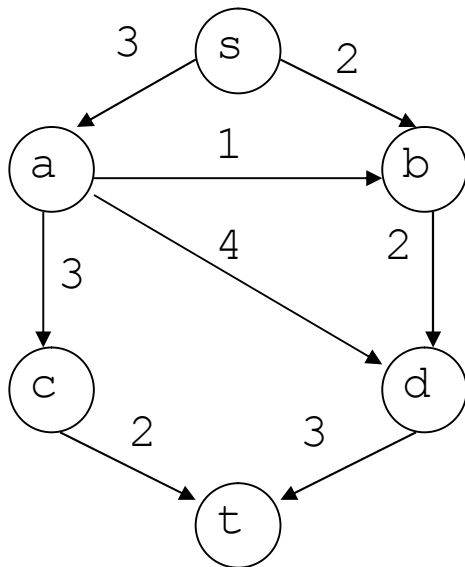
## Réseaux et flux

---

- Réseau : graphe orienté/pondéré avec un sommet-source et un sommet-puits
- Le poids ( $>0$ ) vu comme la *capacité* de débit de la connexion entre sommets  
L'absence d'arc peut s'interpréter comme un arc de poids 0
- Question : quel est le débit maximum entre la source  $s$  et le puits  $t$  ?
- On appelle **flux** d'un graphe  $g$  une fonction  $f: (V \times V) \rightarrow \text{Nombre}$ ,  
(qu'on peut représenter comme un graphe) si elle respecte ces contraintes :
  - *conservation* :  $\sum_{j \in V} f(i, j) == 0, \forall i \in V$ , sauf la source et le puits
  - *consistance* :  $f(i, j) == -f(j, i)$
  - *capacité* :  $f(i, j) \leq g.\text{edgeWeight}(i, j)$
- **Débit total** d'un flux  $f = \sum_{j \in V} f(\text{source}, j)$
- **Capacité résiduelle** de  $i$  vers  $j$ , par rapport à un flux  $f$ :  
 $r(i, j, f) = g.\text{edgeWeight}(i, j) - f(i, j)$  toujours  $\geq 0$  !
- **Graphe résiduel** : en ne gardant que les capacités résiduelles non nulles
- **Chemin augmentant** = chemin de la source au puits dans le graphe résiduel
- **Gain** d'un chemin augmentant = poids du plus faible arc
- Le flux est maximum  $\Leftrightarrow$  il n'y a plus de chemin augmentant.

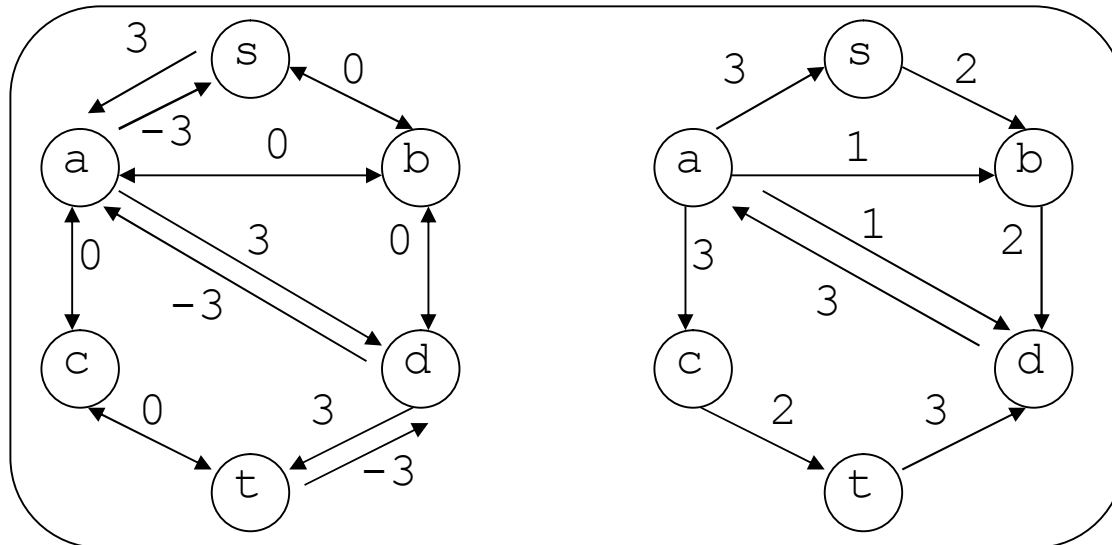
- Algorithme glouton de Ford-Fulkerson :
  - initialiser le flux  $F$  (nul partout) et le graphe résiduel  $R$  (réseau)
  - autant que nécessaire :
    - choisir un chemin augmentant dans  $R$ , de gain  $G$
    - adapter  $F$  (ajouter  $G$  aux étapes, soustraire dans le sens inverse)
    - adapter  $R$  (soustraire  $G$  aux étapes, ajouter dans le sens inverse)
- L'algo est correct quel que soit le chemin augmentant choisi à chaque étape.
- Complexité en  $O(Fn^2)$ , où  $F$  est le débit total. En choisissant le chemin par un parcours en largeur, on a au pire des cas  $O(en^3)$ . En pratique, c'est rapide...

### Exemple

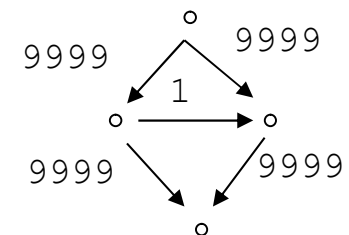
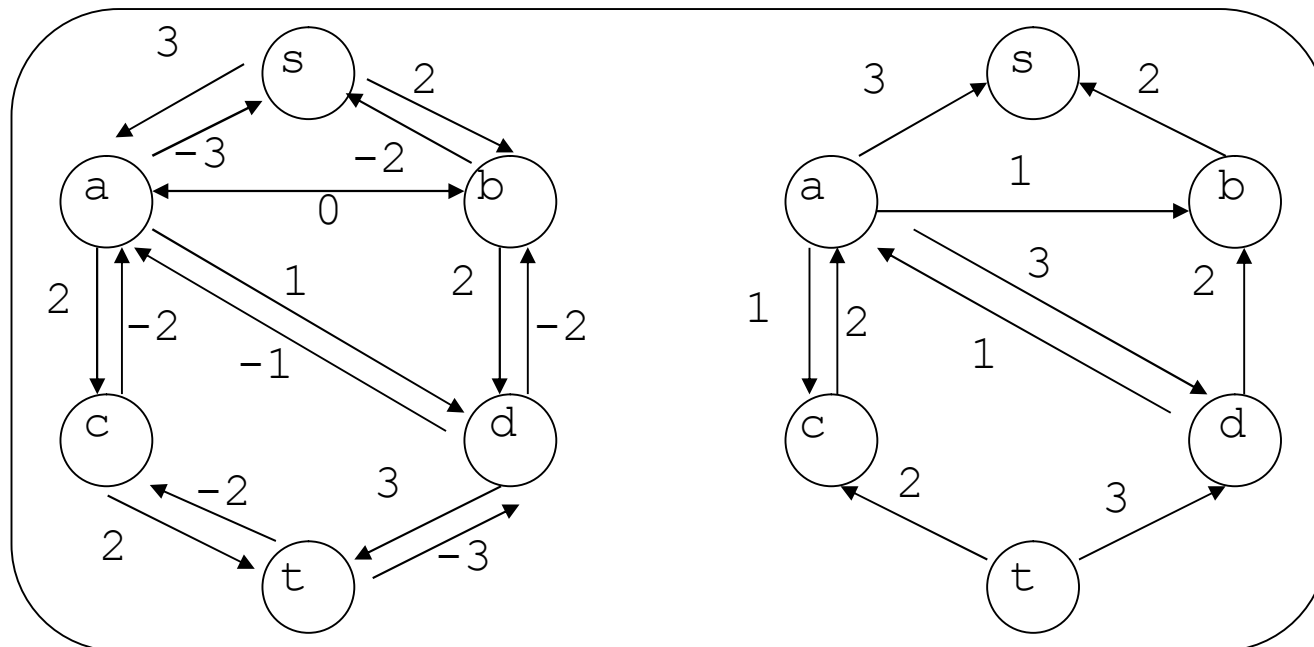




"s-a-d-t", gain=3



"s-b-d-a-c-t", gain=2



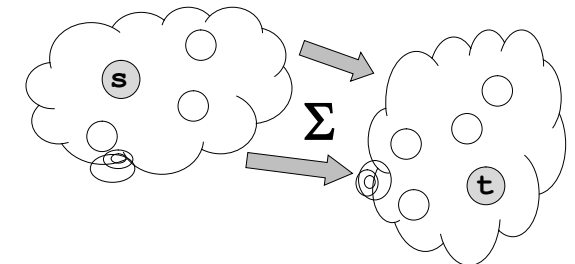
```

WeightedGraph maxFlow(WeightedGraph cap, int src, int sink) {
 WeightedGraph f = empty flow;
 WeightedGraph r = cap (without edges of null cost);
 while (true) {
 Path p = r.pathBetween(src, sink);
 if p does not exist break;
 update(r, f, p, p.minStepCost());
 }
 return f;
}

void update(WeightedGraph r, f, Path p, int benefit) {
 for each step (i, j)
 decrement r.edgeWeight(i,j) by benefit; suppress if 0
 increment f.edgeWeight(i,j) by benefit;
 adapt "reverse" edge (j,i) in r and f
}

```

**Pseudo-code**



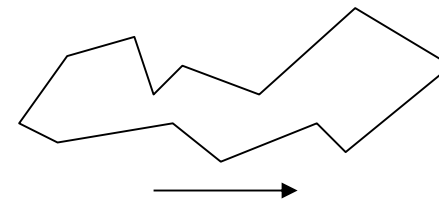
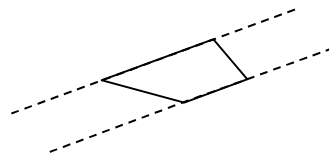
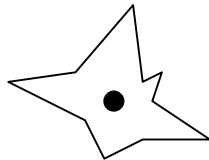
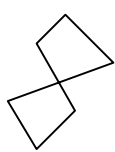
### Minimal Cut Problem (problème de la coupe minimale)

- Problème : couper le réseau en 2 groupes (source et puits séparés), tel que la somme des poids d'arcs du groupe-source vers le groupe-puits soit minimale
- Le théorème *maxFlow-minCut* montre que "débit maximal" == "coupe minimale"
- Groupe-source = ensemble des sommets atteignables depuis la source dans le graphe résiduel *final* obtenu par Ford-Fulkerson !

## Géométrie calculatoire (computational geometry)

---

- *Point* dans un espace à  $n$  dimensions, de coordonnées entières
- Exemple classique, à 2 dimensions : le plan
- *Vecteur directeur* ( $\neq$  point !)
- *Distance* (euclidienne) entre 2 points
- *Surface* : ensemble de points du plan. Surface *connexe* (contiguë)
- *Segment* (fermé) entre 2 points; ensemble des points qu'il contient
- *Droite* passant par 2 points; *demi-droite*
- *Polygone* : séquence de points reliés; la surface ainsi délimitée
- Propriété des polygones :
  - *simple* : sans intersection de segments
  - *convexe* : tout segment reliant 2 points du polygone est intérieur
  - *en étoile* : possède un "centre" qui joint chaque point sans sortir
  - *trapézoïdal* : délimité par 2 droites parallèles et 2 segments
  - *monotone* sur un axe : l'intersection avec toute droite orthogonale est connexe



- Outre la complexité algorithmique et la simplicité du codage, il y a 2 importants critères de qualité des algos (parfois laissés "en exercice") ...  
(sans compter les problèmes d'*overflow*...)

### 1) Robustesse aux erreurs d'arrondi

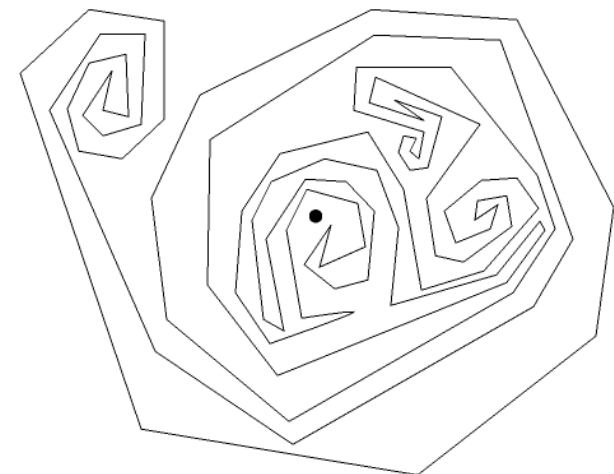
- Problèmes numériques inévitables dans ce domaine
- Au pire, ça provoque une boucle infinie, ou des résultats farfelus
- Au mieux, ça provoque des résultats légèrement erronés

### 2) Traitement des cas dégénérés

- Polygone à 1 point, à 2 points, à 3 points alignés, à points identiques...
- Au pire, on doit décupler les lignes de code pour traiter tous les cas
- Au mieux, l'algorithme normal est automatiquement compatible

### Exemples de primitives sur les types de bases

- Sens de rotation d'une suite de 3 points
- Détection d'intersection de segments
- Inclusion d'un point dans un triangle
- Inclusion d'un point dans un polygone (algo du "fil à plomb")

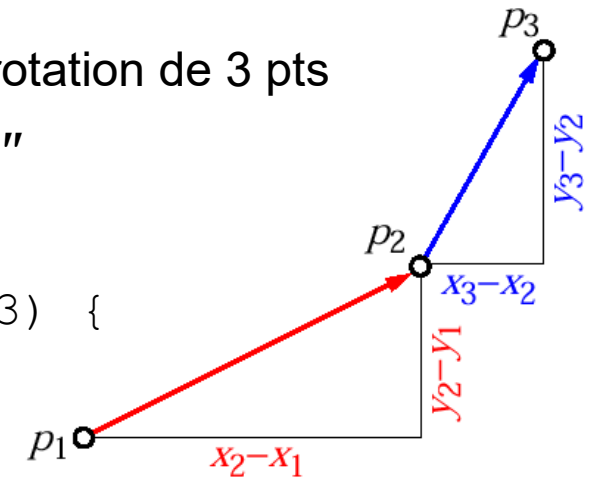


- Surface signée d'un triangle (cf. produit vectoriel), sens de rotation de 3 pts

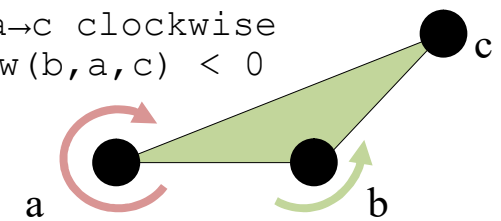
```
// returns twice the triangle "signed area"
// negative if p1-p2-p3 turns clockwise
```

```
int signedArea(Point p1, Point p2, Point p3) {
 return (p2.x-p1.x)*(p3.y-p1.y)
 - (p3.x-p1.x)*(p2.y-p1.y);
}

int ccw(Point p1, Point p2, Point p3) {
 int s=signedArea(p1, p2, p3);
 return (s==0)?0:((s>0)?1:-1)
}
```



b→a→c clockwise  
ccw(b,a,c) < 0

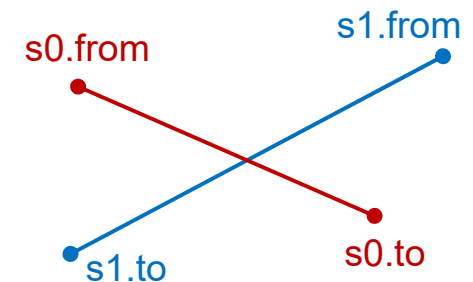


ccw(a,b,c) > 0  
a→b→c counter-clockwise

## Détection d'intersection de 2 segments

- Basé sur le sens de rotation de chacun des 4 triplets de points possibles (un point de chaque côté pour le premier segment, respectivement le deuxième)
- 3<sup>4</sup> "configurations" possibles, avec les cas dégénérés (points alignés) !

```
boolean intersect(Segment s0, Segment s1) {
 return ((ccw(s0.from, s0.to, s1.from)
 *ccw(s0.from, s0.to, s1.to)) <= 0)
 && ((ccw(s1.from, s1.to, s0.from)
 *ccw(s1.from, s1.to, s0.to)) <= 0);
} // incomplet !
```



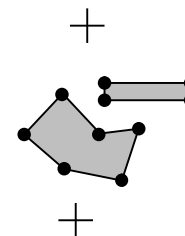
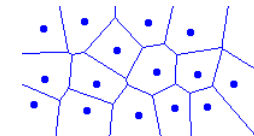
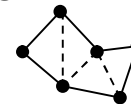
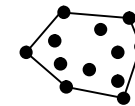
## Coordonnées composites

- Souvent, on interdit que les points aient des coordonnées X ou Y identiques
- Solution : comparer avec l'autre coordonnée comme critère subsidiaire :  

```
isHorizSmaller(a,b) == (a.x < b.x) || (a.x == b.x && a.y < b.y);
isVertSmaller(a,b) == (a.y < b.y) || (a.y == b.y && a.x < b.x);
```
- On parle d'*espace de nombres composites* ou de *coordonnées composites*

## Autres problèmes "classiques"

- Calcul de l'enveloppe convexe d'un ensemble de points
- Détection d'intersections entre N segments (orthogonaux, quelconques)
- Triangulation de polygones (éclairages !)
- Diagramme de Voronoï d'un ensemble de points
- Robotique et calcul de déplacement
- Infographie, détection de visibilité/masquage



## Références

- [Overmars 98] : M. Overmars, M. de Berg, M. van Kreveld, O. Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Second Edition. 1998. Springer. (disponible à la bibliothèque de l'EIA-FR).

## Proximité : "closest-pair problem"

---

- Soit un plan et N points. Quels sont les 2 points les plus proches (cas 1-D ?)

### Technique du balayage

- On "balaie" le plan de gauche à droite, avec une droite verticale
- Principe analogue à la simulation discrète :
  - temps qui passe  $\leftrightarrow$  droite de balayage qui avance
  - événement  $\leftrightarrow$  rencontre du balai avec un objet
- Cette technique suppose 2 structures de données :
  - E : la collection d'événements (géré en file de priorité)
  - S : le "monde" à un instant (p. ex. les objets au voisinage de la droite)
- Pseudo-code général :

```
~~~~~initialiser E (avec au moins un événement)
~~~~~initialiser S
~~~~~while (! E.isEmpty())
~~~~~    e = e.dequeue();
~~~~~    handleEvent(e); // update E and S
```

### Application au "closest-pair problem"

- E = liste des points restants à droite, triés selon l'axe X (subsidièrement Y)
- S = crtMinDist      crtSol      leftMostCandidate  
    candidates : liste des points à gauches, triés selon l'axe Y (et subsid. X)

- Initialisation.  $E$  : liste de tous les points triés  $\left. \begin{array}{l} S : \text{crtMinDist} = +\infty; \\ \text{leftMostCandidate} = \emptyset \end{array} \right\} O(n \ln n)$   
 $\text{crtSol} = \emptyset;$   
 $\text{candidates} = \emptyset$

- Traitement d'un événement. Pseudo-code

```

void handleEvent(Point p) {                                     } N événements
    shrinkCandidates(p);
    t=candidates.inRange(p.y-crtMinDist,                        } O(ln n)
                        p.y+crtMinDist);
    for each point x in t {                                     }
        if (dist(p, x)<crtMinDist)                             } O(1)
            crtMinDist = dist(p, x);
            crtSol = (p, x);
        }
    candidates.add(p);                                         } O(ln n)
}

void shrinkCandidates(Point p) {
    while(p.x-leftMostCandidate.x>crtMinDist) {
        candidates.remove(leftMostCandidate);                 } O(ln n)
        leftMostCandidate = next one in X axis                 } O(1)
    }                                                            } N fois
                                                                } en tout
}

```

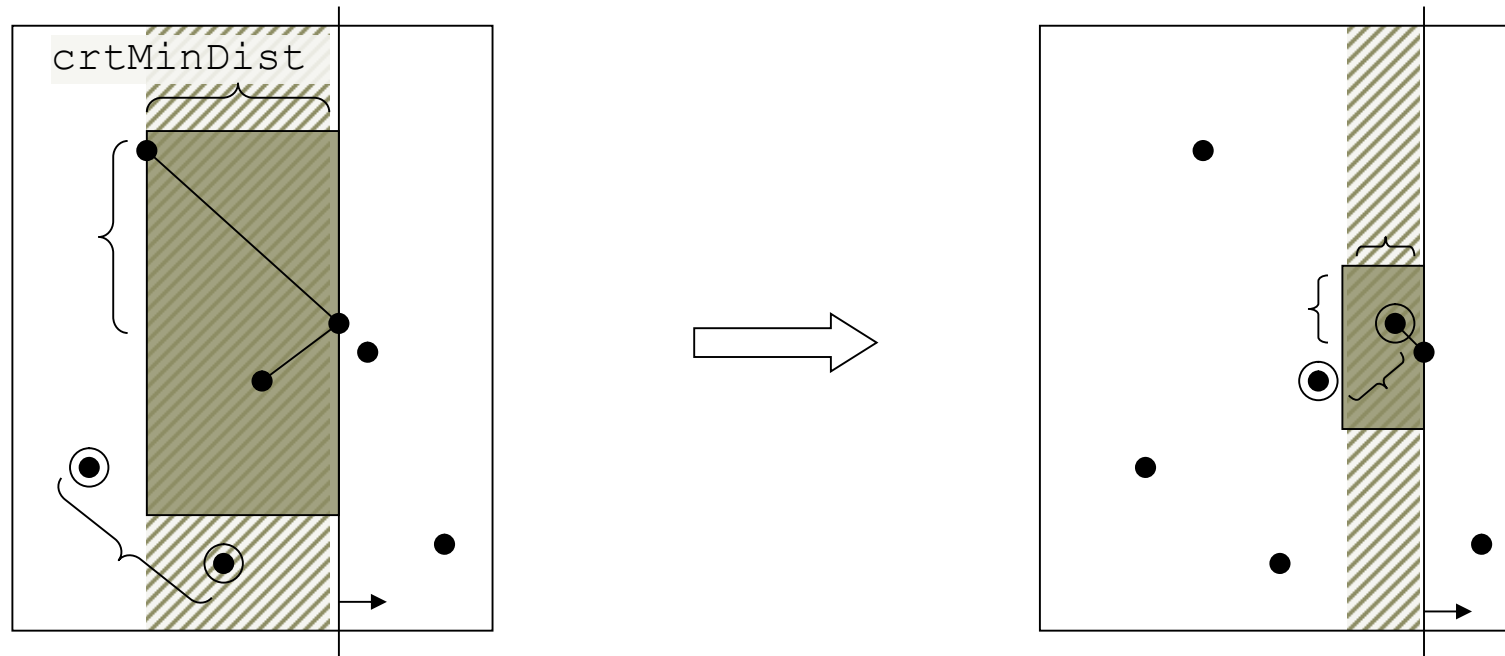
- La recherche d'intervalle donnera au plus 6 points !



- Complexité totale  $O(n \ln n)$

## Implémentation

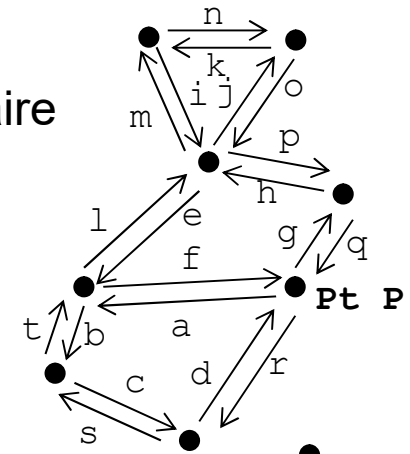
- E peut être représenté par un simple tableau de points (selon X)
- `leftMostCandidate` devient l'indice du point dans le tableau E  
Son voisin de droite se trouve alors immédiatement
- `candidates` peut se gérer en arbre de fouille (équilibré, bien sûr) (selon Y)



- Cf. démo (lien sur Moodle)
- Beaucoup d'autres algorithmes se basent sur la technique de balayage
- Dans d'autres cas, on ajoute des points-événements en cours de traitement

# Carte polygonale connexe

- Comment représenter une découpe géographique
- Carte polygonale connexe : graphe non orienté, connexe, planaire
- Représentation :
  - un objet de la classe Graph
  - une collection de Polygons
  - une structure DCEL



## Type abstrait "Doubly-Connected Edge List" (DCEL)

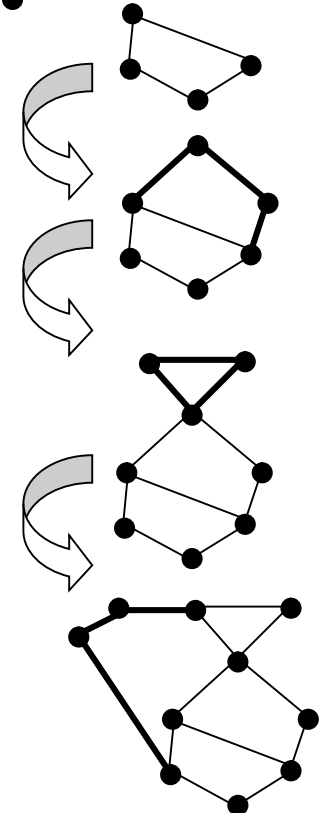
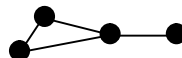
- Notion de demi-arc (half edge), et d'arc jumeau (twin)

```
origin(a) == P      twin(a) == f
prev(a) == d        next(a) == b
```

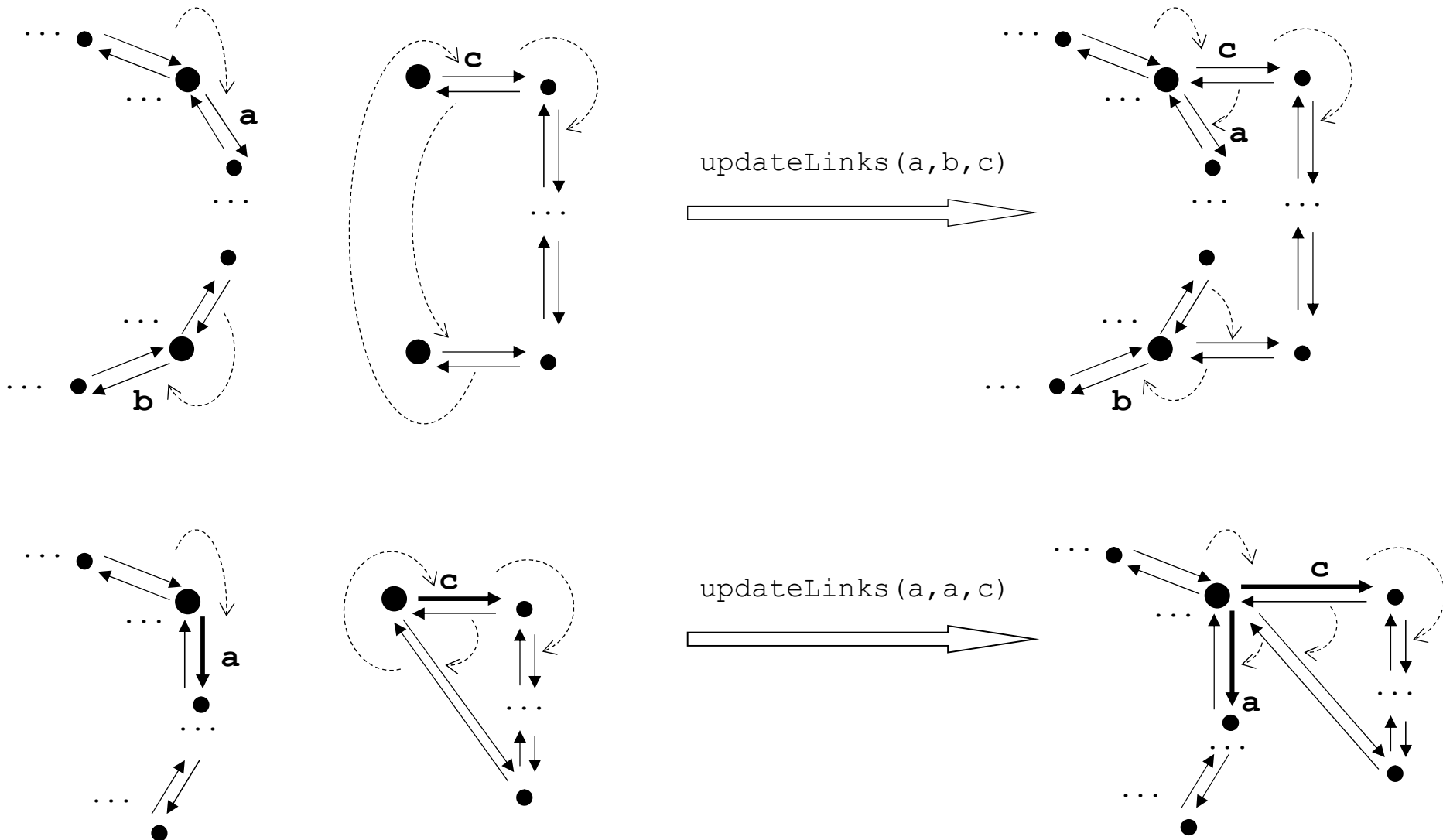
- Un demi-arc délimite la "facette" qui est "sur sa gauche"
- Construction incrémentale d'une carte polygonale :

```
void addEar(Point[] ear);
```

- PRE : le polygone est bien une "oreille" de la carte  
avec 2 "points d'ancrage", sens horaire
- POST : demi-arcs créés (avec twins !), re-chaînage
- Extensions :
  - définir 2 classes supplémentaires : "facette" et "sommet"
  - traiter les décompositions non connexes (polygone dans un autre)
  - traiter les arcs "ouverts"



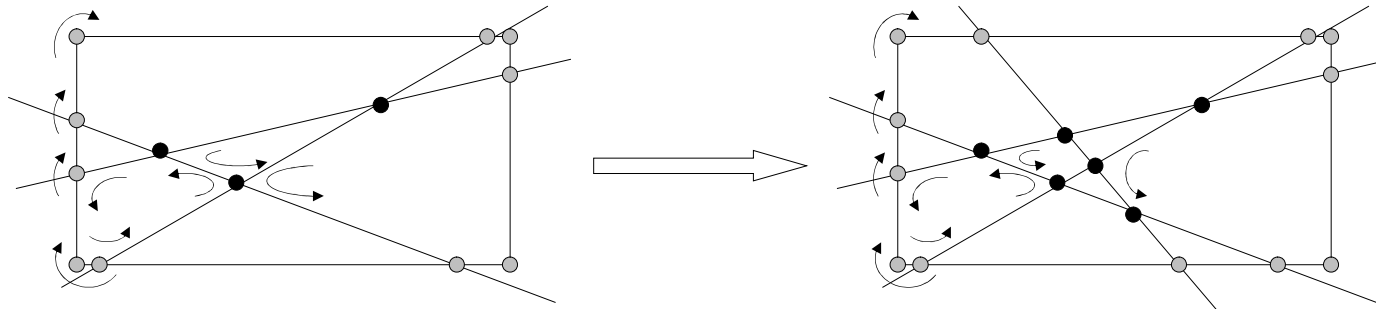
## Opération "ajouter une oreille", 2 situations



```
void updateLinks(HalfEdge a, HalfEdge b, HalfEdge c) {...}
```

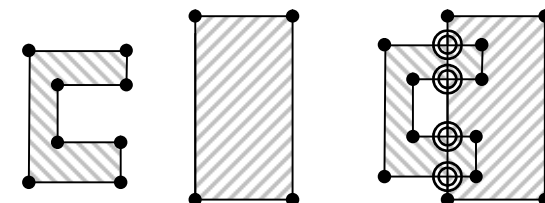
## Arrangement de droites

- Un ensemble de  $n$  droites distinctes engendre également une découpe géographique, un peu particulière : il y aura (au pire)  $n*(n-1)/2$  sommets (intersections des droites). En général, on ajoute un "cadre" autour de la zone intéressante (contenant toutes les intersections)
- On peut construire un DCEL à partir des  $n$  droites en  $O(n^2)$ , par un algorithme incrémental (on affine la carte en ajoutant chaque droite l'une après l'autre)
- De nombreux algorithmes se basent sur un tel "arrangement de droites".



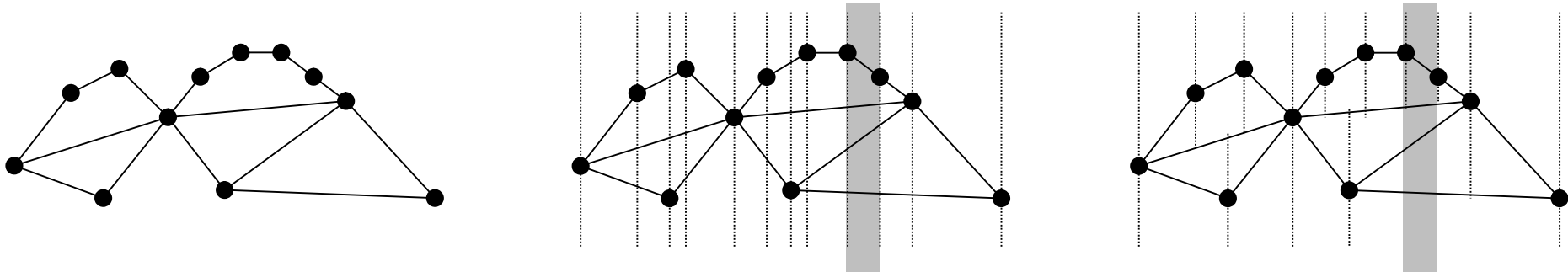
## Overlay et opérations booléennes sur les polygones

- Opération d'*overlay* : superposer deux cartes de géo (joli algo de balayage...)  
Complexité :  $O(n \ln n + k \ln n)$  où  $k$  est le nombre d'intersections
- On définit sur les polygones les opérations ensemblistes courantes :  
union, intersection, différence
- Toutes ces opérations reviennent à faire un "overlay",  
en coloriant les facettes



## Localisation d'un point dans une carte polygonale

- On transforme la carte en carte trapézoïdale. Nombre de points =  $n$



- Problème : complexité en espace mémoire  $O(n^2)$  au pire des cas
- La bonne variante est en  $O(n)$ , (plus tordue à coder...) : lancer deux rayons verticaux depuis chaque point, et s'arrêter à la première intersection
- Découper l'espace en "slabs" (colonnes), délimités par les coordonnées horizontales des points. Gérer les slabs en tableau trié
- On trouvera le slab contenant un point en  $O(\ln n)$  par recherche binaire sur  $X$
- Contenu d'un slab : collection de segments de polygone, non verticaux
- Entre les segments d'un slab, jamais d'intersection !
- Donc, ils sont complètement ordonnés sur l'axe des  $Y$ , par la relation de comparaison "être au-dessus de"
- On va gérer la collection en tableau trié; recherche binaire en  $O(\ln n)$
- Tester si un point est "au-dessous" d'un segment, c'est détecter le sens de rotation entre les 3 points !

## Disque minimal

---

### Tout autre chose : un joli petit algorithme probabiliste

- Trouver le disque minimal  $D_P$  recouvrant un ensemble  $P$  de  $N$  points
- Application : soit une table de montage. Positionner et dimensionner la grue de montage en vue d'une automatisation
- Extensions :  $>2$  dimensions, ellipses...
- Théorèmes utilisés par l'algorithme :
  - il y a un unique cercle passant par 3 points distincts (non colinéaires)
  - si un point  $q \in D_P$ , alors  $D_{P \cup \{q\}} = D_P$
  - si un point  $q \notin D_P$ , alors  $D_{P \cup \{q\}}$  contient  $q$  sur sa frontière
- On va écrire 3 méthodes, selon qu'on impose 0-2 points sur la frontière :

```
// minimal disc enclosing p[0..n-1], with a and b on the circle  
Disc miniDiscWith2Points(Point[] p, int n, Point a, Point b);
```

```
// minimal disc enclosing p[0..n-1], with a on the circle  
Disc miniDiscWith1Point (Point[] p, int n, Point a);
```

```
// minimal disc enclosing every point in p[]  
public Disc miniDisc(Point[] p);
```

```

Disc miniDiscWith2Points(Point[] p, int n, Point a, Point b){
    Disc d = discFromDiameter(a,b); // (3)
    for(int i=0; i<n; i++)
        if (p[i] ∉ d) d = discFromPoints(a, b, p[i]); // (4)
    return d;
}

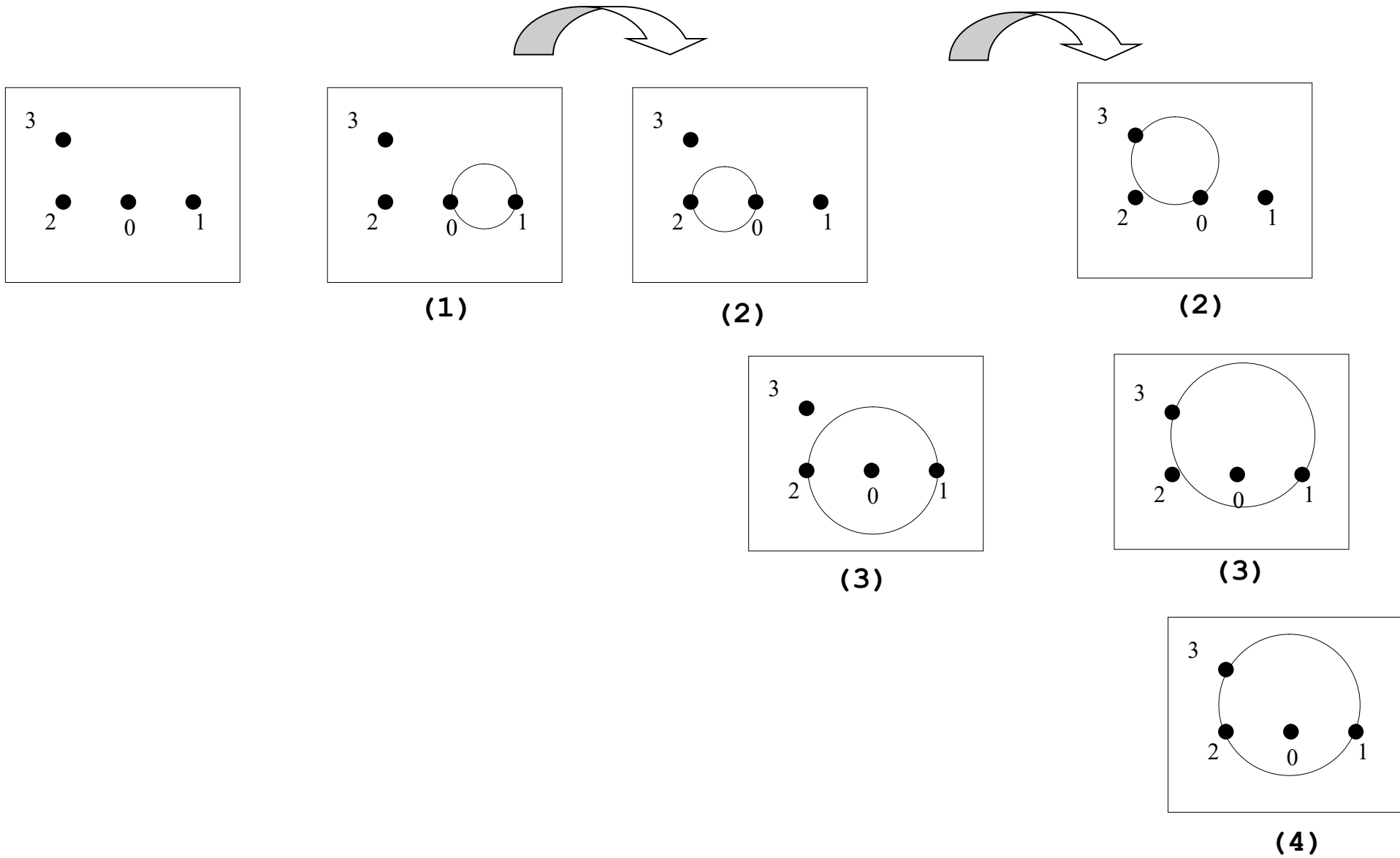
Disc miniDiscWith1Point (Point[] p, int n, Point a){
    Disc d = discFromDiameter(a, p[0]); // (2)
    for(int i=0; i<n; i++)
        if (p[i] ∉ d) d = miniDiscWith2Points(p, i, a, p[i]);
    return d;
}

Disc miniDisc(Point[] p){
    (*)
    Disc d = discFromDiameter(p[0], p[1]); // (1)
    for(int i=0; i<p.length; i++)
        if (p[i] ∉ d) d = miniDiscWith1Point(p, i, p[i]);
    return d;
}

```

- Problème : certains ordres des points sont moins efficaces que d'autres  
Dessiner le cas le plus défavorable à 5 points...
- Solution : calculer en (\*) une permutation aléatoire des points
- On peut montrer que l'algorithme a alors une complexité *espérée* de  $O(n)$  !

- Exemple :

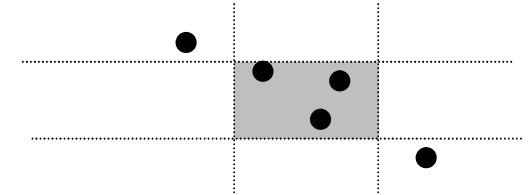




## Recherche des points dans un intervalle

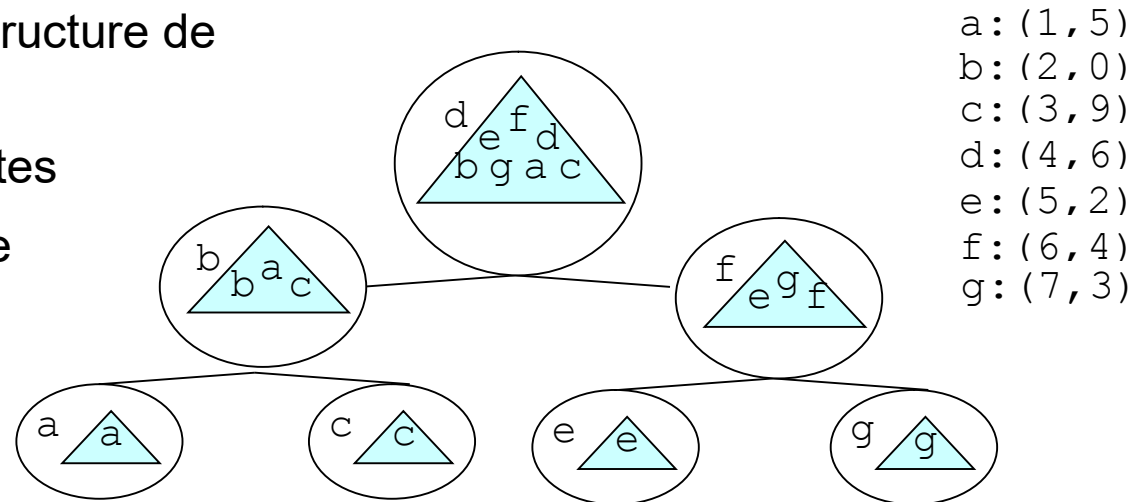
---

- Soit une collection de points dans le plan
- Trouver ceux qui sont dans une région délimitée par un intervalle sur les 2 axes (*orthogonal range searching*)
- Principe : construire une fois une structure de données permettant de traiter efficacement ce genre de requêtes
- Algo "output-sensitive" : s'il y a peu de réponses, on veut que ce soit rapide
- Cas à 1 dimension : un arbre de fouille suffit
  - trier les points
  - construire l'arbre de fouille optimal (facile ou non ?)
  - algo de recherche par intervalle (déjà discuté)
- Applications cruciales dans les bases de données (n dimensions) :
  - plan            hyper-plan dont les axes sont les champs
  - point            record
  - search            requête typique dans un SGBD
- Il y a plusieurs structures possibles (arbres k-d, range tree, ...)
- Certaines structures permettent aussi des ajouts/retraits assez efficaces



## 2-dimensional Range Trees

- A la base, tous les points sont organisés comme arbre de fouille  $T_x$ , ordonné selon la 1ère coordonnée (axe des X)
- Chaque noeud contient un point, et une *copie*  $T_y$  de tous les points présents dans le sous-arbre
- C'est un exemple de structure de données *multi-niveaux*
- Coordonnées composites
- Généralisation possible à K dimensions

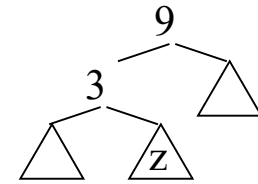


- Le deuxième niveau est aussi organisé en arbre de fouille, mais selon la 2ème coordonnée (axe Y)
- Algorithme de recherche :

Recherche par intervalle dans  $T_x$ , et vérification de la condition sur l'axe Y

Quand on constate que tout le sous-arbre  $\in [x1..x2]$   
 recherche par intervalle dans  $T_y$

- Recherche par intervalle sur Tx : comment trouver les noeuds intermédiaires dont tous les descendants sont concernés ?
- On sait que dans le sous-arbre z, il n'y a pas d'éléments  $<3$  ou  $>9$



```

void query2D(BTree tx, int xFrom, int xTo,
              int xMin, int xMax) {
    if (tx.isBottom()) return;
    m = (tx.consult());
    if (m.x > xTo) {
        query2D(tx.left (), xFrom, xTo, xMin, m.x); return;
    } else if (m.x < xFrom) {
        query2D(tx.right (), xFrom, xTo, m.x, xMax); return;
    }
    if (xMin >= xFrom && xMax <= xTo) { // whole subtree fits X
        query1D(associatedTree(tx), ...); return;
    }
    Vérifier si m satisfait l'intervalle [yFrom..yTo]
    query2D(tx.left (), xFrom, xTo, xMin, m.x );
    query2D(tx.right (), xFrom, xTo, m.x, xMax);
}

```

- ]xMin..xMax[      intervalle effectif du sous-arbre courant (au départ  $-\infty..+\infty$ )
- [xFrom..xTo]      intervalle recherché

## Complexité du Range Tree (à d dimensions)

- Recherche  $O(\ln^d(n) + Z)$       Mémoire  $O(n \ln^{d-1}(n))$     Construction  $O(n \ln^{d-1}(n))$
- Avec  $Z$  = nbre de points ressortis (algo output-sensitive)

## Construction du Range Tree

- Construire 2 listes triées des points, une selon X, une selon Y

```
BTREE build2dRangeTree(Point[] xP, int left, int right,
                        int [] yOrder) {
    BTREE tx = new BTREE();
    if (left>right) return tx;
    BST ty = buildBST(points in yOrder);
    int mid = (left+right) / 2;
    ... arrange yOrderLeft, yOrderRight corresponding to both halves
    tx.insert(new RTElt(ty, xP[mid]))
    tx.left().paste(build2dRangeTree(xP, left, mid-1, yLeftOrder));
    tx.right().paste(build2dRangeTree(xP, mid+1, right, yRightOrder));
    return tx;
}
```

|        | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|--------|---|---|---|---|---|---|---|
| xP     | a | b | c | d | e | f | g |
| yOrder | 1 | 4 | 6 | 5 | 0 | 3 | 2 |

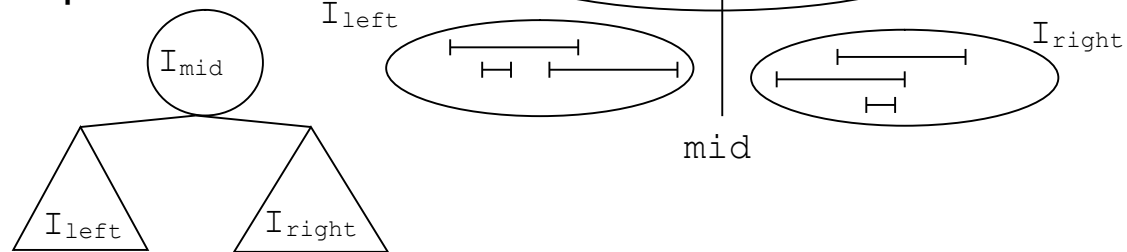
- yOrder donne les points de XP triés verticalement

## Interval trees

- Soit un ensemble d'intervalles sur la droite des entiers (ou des réels)
- Construire une structure de données pour trouver quels intervalles contiennent un nombre donné
- Algo naïf : gérer une liste d'intervalles, et la parcourir en  $O(n)$   
Arbres d'intervalle : une structure de données plus efficace
- Complexité souhaitée :
  - $O(k + \ln n)$  pour traiter une requête,  $k$ =nbre d'intervalles rapportés
  - $O(n \ln n)$  pour la construction de la structure
  - $O(n)$  en espace mémoire

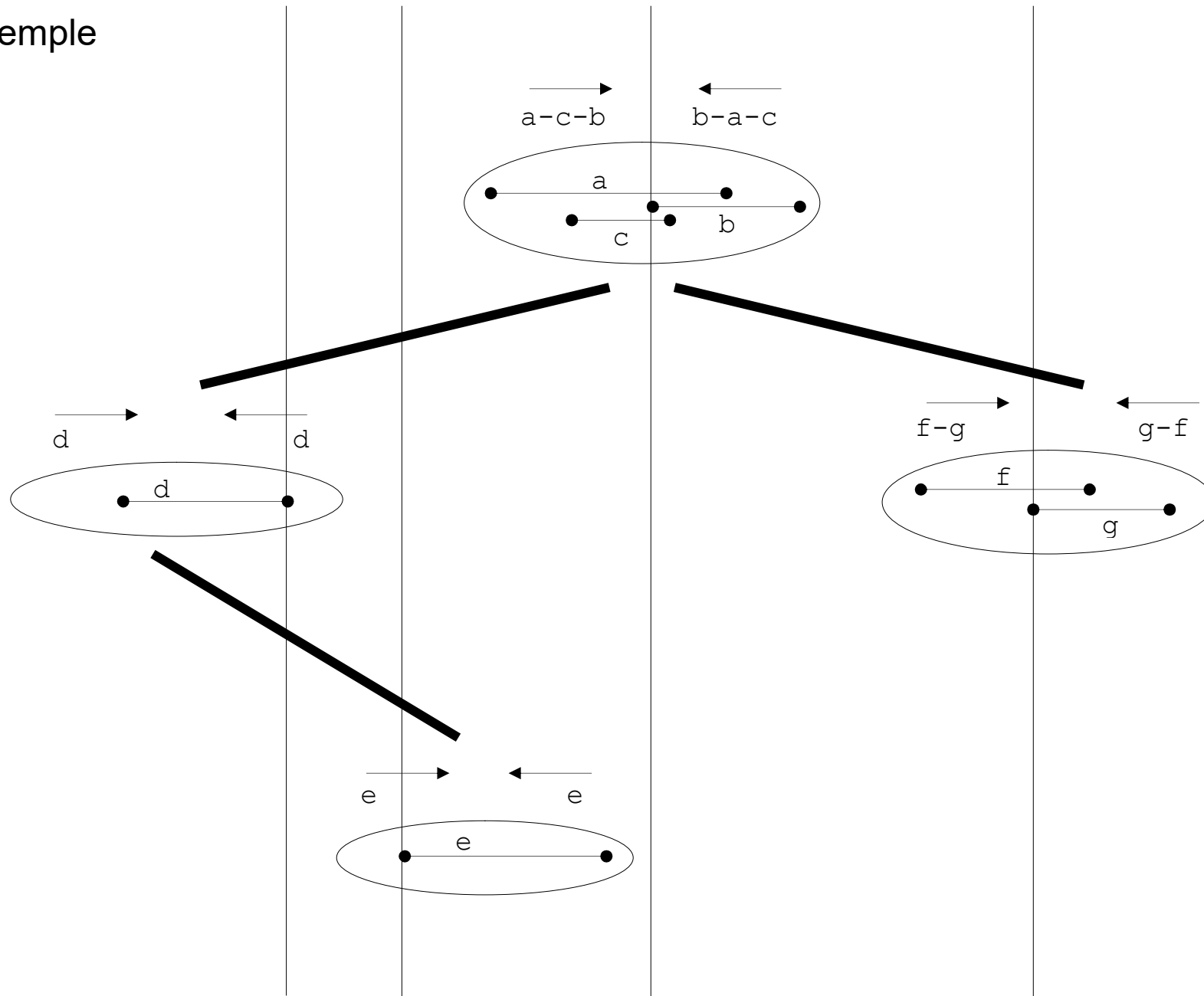
- Soit  $mid$  le point médian. 3 groupes

- On construit un arbre binaire  
*interval tree*



- Ok. Mais comment traiter toute requête (pas forcément où  $x == mid$ ) ?
- Représenter  $I_{mid}$  "à double", par 2 Listes triées :
  - $L_{left}$  trié par ordre croissant des extrémités gauches a-c-b
  - $L_{right}$  trié par ordre décroissant des extrémités droites b-a-c
- On peut alors traiter des requêtes  $X$  avant ou après le point  $mid$  !

- Exemple

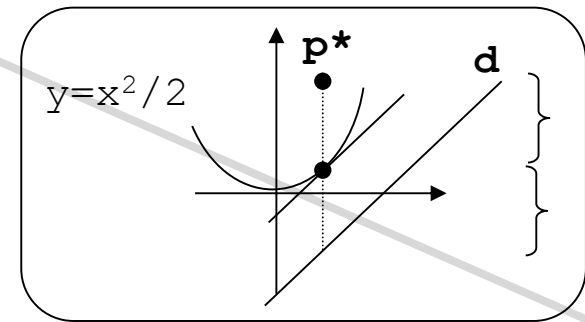


## Représentation duale dans le plan (droite = point)

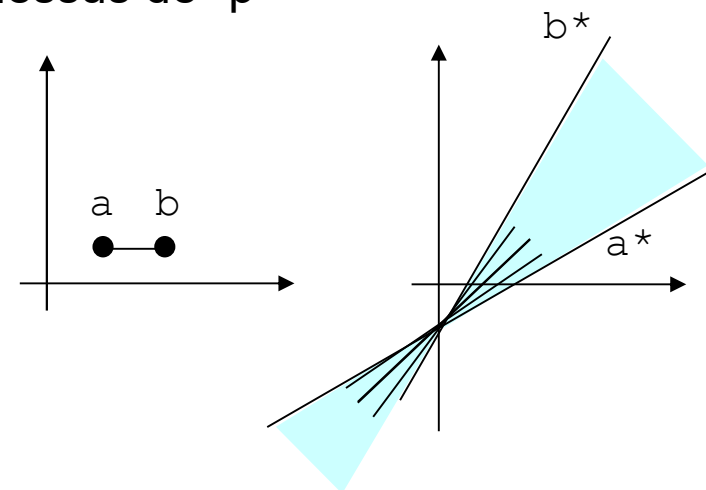
- Parfois, changer de représentation aide à trouver une solution...
- Une droite *non-v verticale* "d:  $y=ax + b$ " : (a=pente, b=ordonnée à l'origine)
- Plan dual\* : on représente une droite (primale) d par un point dual  $d^*$  (a,-b) !

|          |          |                     |          |        |
|----------|----------|---------------------|----------|--------|
| - droite | $y=ax+b$ | $d \rightarrow d^*$ | (a, -b)  | point  |
| - point  | (a, b)   | $p \rightarrow p^*$ | $y=ax-b$ | droite |

(Interprétation géométrique de la transformation)



- Propriétés de cette transformation : elle préserve :
  - l'incidence :  $p \in d \Leftrightarrow d^* \in p^*$
  - l'ordre :  $p$  "au-dessus de"  $d \Leftrightarrow d^*$  "au-dessus de"  $p^*$
- On peut étendre la correspondance à d'autres constructions...
- Exemple : segment de droite
- Certains problèmes sont plus faciles à manipuler dans leur version duale...



# Priority Search Tree

- Idem Range Trees, mais pour des requêtes  $[xFrom..xTo] [-\infty..yTo]$  ?

- Principe du Treap, avec quelques adaptations :

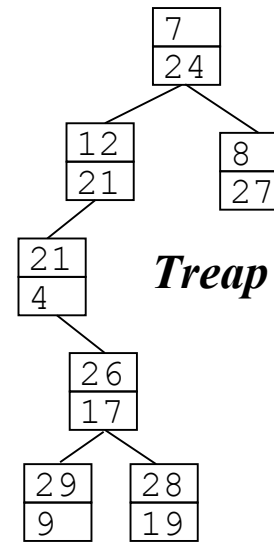
- élt = coordX    pty = coordY
- on connaît toute la collection par avance

- Si on partitionne selon `racine.x`, l'arbre n'est pas forcément balancé  
Solution : ajouter un champ `xMid` à chaque noeud, et partager à la moitié.

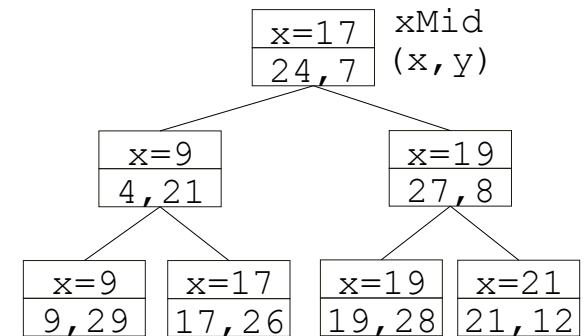
- Pseudo-code :

```
BTree build(Point[] points)
    racine = point de coord Y minimum.
    calculer xMid sur les points restants
    partitionner le reste en L ( $x \leq xMid$ ) et R ( $x > xMid$ ).
    left = build(L); right = build(R)

void search(BTree t, int xFrom, int xTo, int yTo)
    if (t.isBottom()) return;
    Elt e = t.consult();
    if (e.p.y > yTo) return;
    test if  $xFrom \leq e.p.x \leq xTo$ 
    ... recherche par intervalle classique sur X d'après e.xMid
```



## Priority Search Tree





# Clustering

---

- Soient N points dans un (hyper-) plan. Former des regroupements "cohérents" (clusters) entre points proches
- Notion de distance/métrique : fonction à 2 arguments avec les propriétés suivantes :
  - $d(a, b) \geq 0$
  - $d(a, b) = d(b, a)$  symétrie
  - $d(a, b) + d(b, c) \geq d(a, c)$  inégalité triangulaire
  - $d(a, b) = 0 \iff a = b$
- Définitions de distance entre 2 points : double distance(Point a, Point b)
  - euclidienne :  $\sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$
  - de Manhattan :  $|a.x - b.x| + |a.y - b.y|$
- Notion de centre de gravité d'un cluster
- Définition de distance entre 2 clusters :
  - distance euclidienne entre leurs centres de gravité respectifs
  - autres définitions possibles
- Application à la reconnaissance des formes : chaque coordonnée correspond à une caractéristique (*feature*) des formes manipulées
- Deux variantes, selon qu'on connaît à l'avance ou non le nombre de clusters
- Suivant l'application, on peut trouver une mesure pour estimer le "meilleur" nombre de clusters

## Algorithme K-Means (ISODATA)

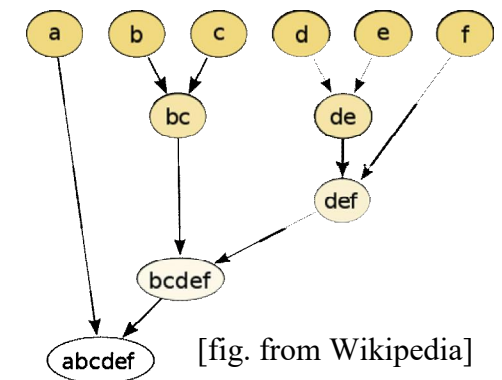
- On connaît le nombre K de clusters
- Chaque cluster est caractérisé par un centre de gravité

```
créer K clusters (vides) placés aléatoirement
répéter
  pour chaque point p
    placer p dans le cluster c le plus proche
    recalculer si nécessaire les centres de gravité
    éventuellement : fusionner ou diviser des clusters (*)
tant qu'il y a des changements
```

## Clustering hiérarchique

- Idée : construire un arbre (un peu comme dans l'algo de Huffman)

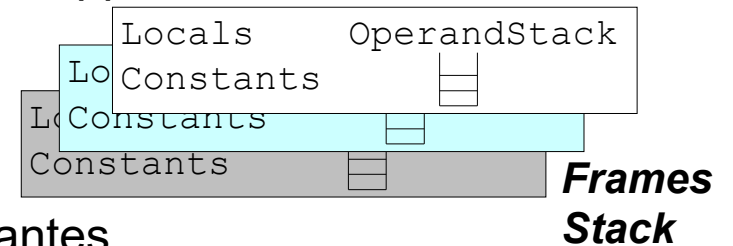
```
créer N clusters, chacun avec un seul point
tant qu'il reste plusieurs clusters (*)
  fusionner les 2 clusters les plus proches
  recalculer les centres de gravité affectés
```



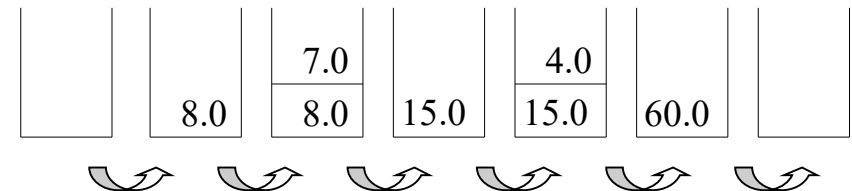
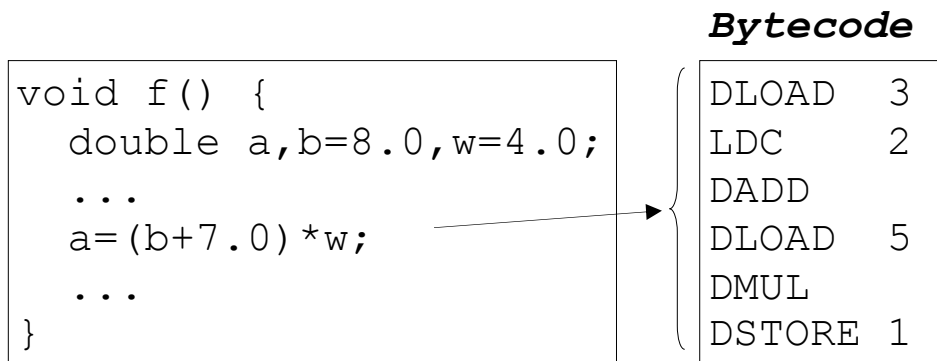
- L'arbre nous donne toute une famille de solutions;  
on appelle ça un *dendrogramme*
- (\*) Parfois, on donne un seuil      pour la distance minimale inter-cluster  
                                         ou    pour la variance maximale intra-cluster  
                                         ou    pour le nombre de clusters désiré

# Java Bytecode

- Le bytecode est le "langage machine" pour la "*machine à pile*" Java Virtual Machine
- Chaque méthode Java s'exécute dans un certain contexte : un **Frame** de la JVM. Ces frames sont empilés, ce qui correspond à la pile des appels



- Un Frame contient :
  - les variables locales et paramètres
  - un (lien vers un) "constant pool" qui gère les constantes
  - une pile d'opérandes (32 bits) sur laquelle vont agir toutes les opérations
- Chaque instruction est formée d'un opcode d'un byte, suivi par ses paramètres



**Operand Stack**

**Locals**

|          |      |   |     |     |
|----------|------|---|-----|-----|
| Offsets  | 0    | 1 | 3   | 5   |
| Id       | this | a | b   | w   |
| Contents | ...  | ? | 8.0 | 4.0 |

**Constants**

|        |     |     |     |     |
|--------|-----|-----|-----|-----|
| Number | 0   | 1   | 2   | 3   |
| Value  | ... | ... | 7.0 | ... |

- Il existe de bons outils pour manipuler le bytecode (bibliothèque/plugin ASM)
- Voici quelques instructions du bytecode, pour se faire une idée...

| <b>Mnemonic</b>    | <b>Other bytes</b> | <b>Stack: before → after</b>           | <b>Description</b>           |
|--------------------|--------------------|----------------------------------------|------------------------------|
| <u>i</u> sub       |                    | val1, val2 → result                    | (val1-val2) on ints          |
| <u>i</u> feq       | offset(2b)         | value →                                | if (value==0) jump at offset |
| goto               | offset(2b)         | →                                      | jump at offset               |
| dup                |                    | val → val, val                         | duplicate top of stack       |
| ldc                | constantId         | → cstVal                               | load a constant from pool    |
| <u>i</u> const_0   |                    | → 0                                    | load the constant zero       |
| <u>l</u> cmp       |                    | val1, val2 → cmpResult                 | val1 >? val2 (-1,0,+1)       |
| <u>i</u> load      | varAddr            | → varValue                             | load a variable              |
| <u>i</u> store     | varAddr            | value →                                | variable assignment          |
| <u>d</u> aload     |                    | tab, index → tab[index]                | load an array cell           |
| <u>i</u> inc       | varAddr,k          | →                                      | var+=k                       |
| putfield           | fieldId(2b)        | objectRef, val →                       | modify an attribute          |
| <u>a</u> return    |                    | objectRef →                            | return a reference           |
| bipush             | val                | → val                                  | push val on stack            |
| pop                |                    | val →                                  | discard top of stack         |
| invoke-<br>virtual | methId(2b)         | objectRef,<br>arg1,arg2,... → [result] | invoke an instance method    |

land, castore, drem, istore\_1, pop2, dup2, ifne, iflt, ifnull, arraylength, i2d...