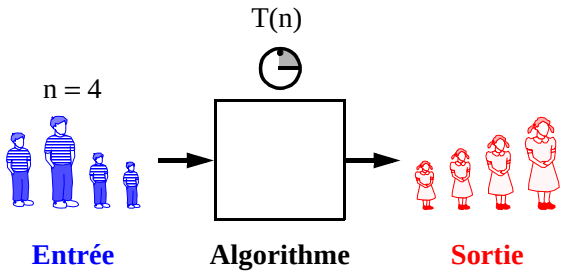
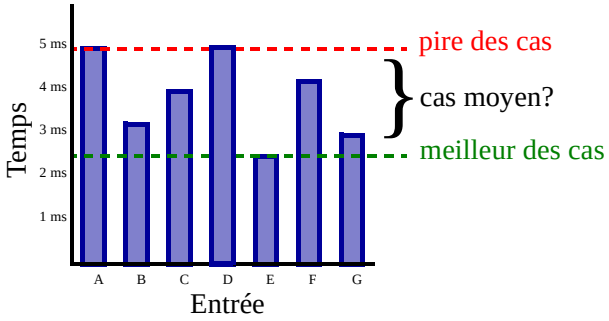
ANALYSE D'ALGORITHMES

- Révision mathématique rapide
- Temps d'exécution
- Pseudo-code
- Analyse d'algorithmes
- Notation asymptotique
- Analyse asymptotique



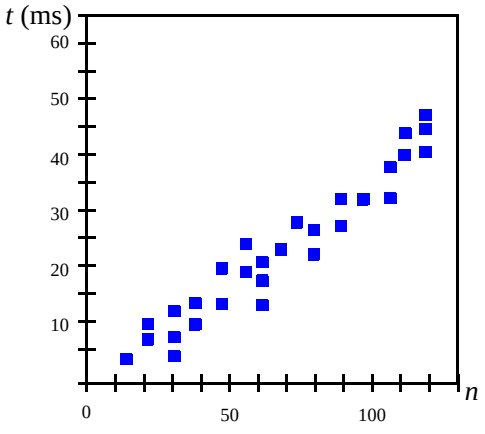
Cas moyen vs. Pire des cas: Temps d'exécution d'un algorithme

- Un algorithme peut être plus performant avec certains ensembles de données qu'avec d'autres,
- Trouver le cas moyen peut s'avérer difficile, alors les algorithmes sont mesurés typiquement selon la complexité temporelle du pire des cas.
- De plus, pour certain domaines d'application (par ex. contrôle aérien, chirurgie, gestion de réseau) connaître la complexité temporelle du pire des cas est d'importance cruciale.



Mesurer le temps d'exécution

- Comment devrions-nous mesurer le temps d'exécution d'un algorithme?
- Étude expérimentale:
  - Écrivez un programme qui réalise l'algorithme.
  - Exécutez le programme avec des ensembles de données de taille et de contenu variés.
  - Utilisez une méthode (System.currentTimeMillis()) pour mesurer précisément le temps d'exécution.
  - Les mesures résultantes devraient ressembler à:



Au-delà des études expérimentales

- Les études expérimentales ont quelques restrictions:
  - Il est nécessaire de réaliser et de tester l'algorithme afin de déterminer son temps d'exécution.
  - Les essais peuvent être faits seulement sur un ensemble limité d'entrées, et ils peuvent ne pas être indicatifs du temps d'exécution d'autres entrées non considérées.
  - Afin de comparer deux algorithmes, les mêmes environnements matériel et logiciel devraient être utilisés.
- Nous développerons maintenant une méthodologie générale pour analyser le temps d'exécution d'algorithmes qui:
  - Utilise une description de haut niveau de l'algorithme au lieu de tester sa réalisation.
  - Considère toutes les entrées possibles.
  - Permet d'évaluer l'efficacité d'un algorithme indépendamment des environnements matériels et logiciels.

Pseudo-code

- Le pseudo-code est une description d'algorithme qui est plus structurée que la prose ordinaire mais moins formelle qu'un langage de programmation.
- Exemple: trouver l'élément maximal d'un vecteur (*array*).

```
Algorithm arrayMax(A, n):
    Entrée: Un vecteur A contenant n entiers.
    Sortie: L'élément maximal de A.
  currentMax ← A[0]
  for i ← 1 to n −1 do
    if currentMax < A[i] then
      currentMax ← A[i]
  return currentMax
```

- Le pseudo-code est notre notation de choix pour la description d'algorithmes.
- Cependant, le pseudo-code cache plusieurs problèmes liés à la conception de programmes.

Qu'est-ce que le pseudo-code?

- Un mélange de langage naturel et de concepts de programmation de haut niveau qui décrit les idées générales derrière la réalisation générique d'une structure de données ou d'un algorithme.
  - Expressions: utilisez des symboles mathématiques standards pour décrire des expressions booléennes et numériques
    - utilisez ← pour des affectations ("=" en Java)
    - utilisez = pour la relation d'égalité ("==" en Java)
  - Déclaration de méthodes:
    - **Algorithm** nom(*param1*, *param2*)
  - Éléments de programmation:
    - décision: **if** ... **then** ... **[else** ... **]**
    - boucle while: **while** ... **do**
    - boucle repeat: **repeat** ... **until** ...
    - boucle for: **for** ... **do**
    - indexage de vecteur: **A[*i*]**
  - Méthodes:
    - appel: object method(args)
    - retour: **return** value

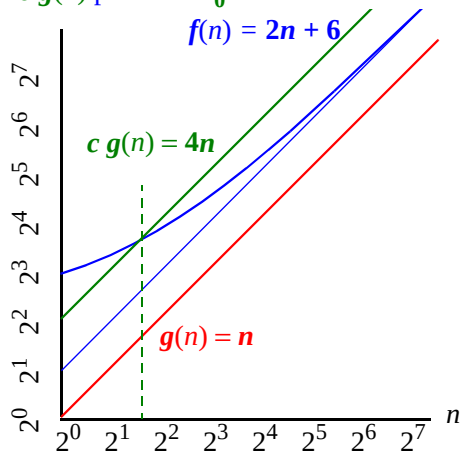
Analyse d'algorithmes

- Opérations primitives: opérations de bas niveau qui sont largement indépendantes du langage de programmation et qui peuvent être identifiées en pseudo-code, par exemple:
  - Appel et retour d'une méthode
  - effectuer une opération arithmétique (addition)
  - comparer deux nombres, etc.
- En inspectant le pseudo-code, nous pouvons compter le nombre d'opérations primitives exécutées par un algorithme.
- Exemple:

```
Algorithm arrayMax(A, n):
    Entrée: Un vecteur A contenant n entiers.
    Sortie: L'élément maximal de A.
  currentMax ← A[0]
  for i ← 1 to n −1 do
    if currentMax < A[i] then
      currentMax ← A[i]
  return currentMax
```

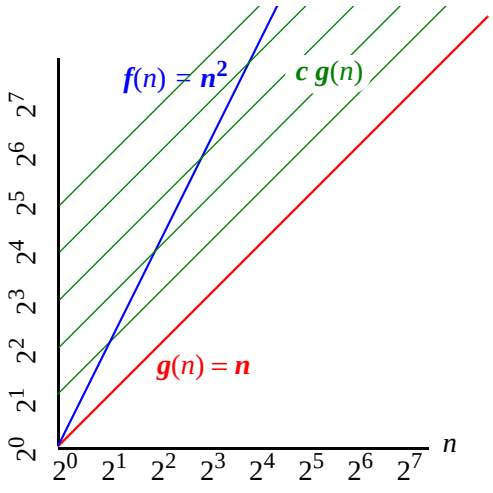

Notation asymptotique

- But: simplifier l'analyse en se débarrassant de l'information superflue.
  - comme "arrondir" 1 000 001 ≈ 1 000 000
  - nous désirons indiquer formellement que $3n^2 \approx n^2$
- La notation "Grand-O"
  soit les fonctions $f(n)$ et $g(n)$, nous disons que $f(n)$ est $O(g(n))$ **si et seulement si**
  il y a des constantes positives c et n_0 tel que
  $f(n) \leq c\,g(n)$ pour $n \geq n_0$



Un autre exemple

- n^2 **n'est pas** O(n)
- nous ne pouvons pas trouver c et n_0 tel que $n^2 \leq c\,n$ for $n \geq n_0$



Notation asymptotique (suite)

- Note: Même si il est correct de dire "7n - 3 est $\boldsymbol{O}(n^3)$", une meilleure formulation est "7n - 3 est $\boldsymbol{O}(n)$", c'est-à-dire, nous devrions faire l'approximation la plus juste possible.
- Règle simple: laissez tomber les termes d'ordre inférieur de même que les facteurs
  - 7n - 3 est $\boldsymbol{O}(n)$
  - $8n^2\log n + 5n^2 + n$ est $\boldsymbol{O}(n^2\log n)$
- Classes spéciales d'algorithmes:
  - logarithmique: $\boldsymbol{O}(\log n)$
  - linéaire $\boldsymbol{O}(n)$
  - quadratique $\boldsymbol{O}(n^2)$
  - polynomial $\boldsymbol{O}(n^k)$, $k \geq 1$
  - exponentiel $\boldsymbol{O}(a^n)$, $n > 1$
- "Parenté" de Grand-O
  - Ω(f(n)): Grand-Oméga
  - Θ(f(n)): Grand-Thêta

Analyse asymptotique et temps d'exécution

- Utilisez la notation Grand-O pour indiquer le nombre d'opérations primitives exécutées en fonction de la taille d'entrée.
- Par exemple, nous disons que l'algorithme arrayMax a un temps d'exécution $\boldsymbol{O}(n)$.
- En comparant les temps d'exécution asymptotiques
  - un algorithme d'ordre $\boldsymbol{O}(n)$ est meilleur qu'un autre d'ordre $\boldsymbol{O}(n^2)$
  - de la même façon, $\boldsymbol{O}(\log n)$ est meilleur que $\boldsymbol{O}(n)$
  - hiérarchie de fonctions:
  - $\log n << n^{-2} << n << n \log n << n^2 << n^3 << 2^n$
- Attention!
  - Méfiez-vous des facteurs constants très grands. Un algorithme au temps d'exécution 1 000 000 n est quand même $\boldsymbol{O}(n)$ et peut être moins efficace sur votre ensemble de données qu'un autre au temps d'exécution $2n^2$, qui est $\boldsymbol{O}(n^2)$.

Exemple d'analyse asymptotique

- Un algorithme pour calculer les moyennes préfixes:

Algorithm prefixAverages1(X):
Entrée: Un vecteur de nombres X à n éléments.
Sortie: Un vecteur de nombres A à n éléments tel que A[i] est la moyenne des éléments X[0], ... , X[i].
Soit A un vecteur de n nombres.
for $i \leftarrow 0$ **to** n - 1 **do**
  $a \leftarrow 0$
  for $j \leftarrow 0$ **to** i **do**
    $a \leftarrow a + X[j]$
  A[i] $\leftarrow a/(i + 1)$
return array A

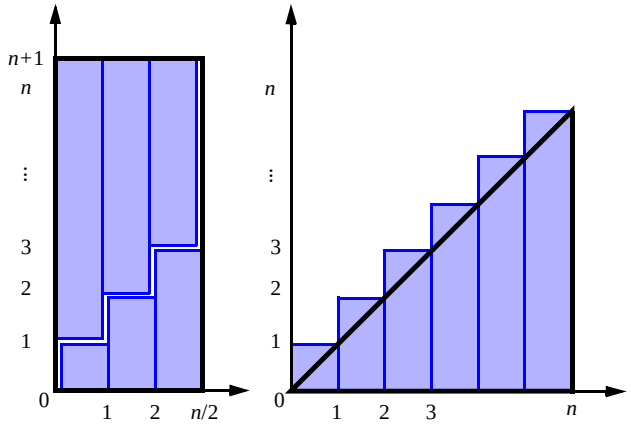
- Analyse ...

Révision mathématique rapide

- Progression arithmétique:
  - Un exemple

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n^2 + n}{2}$$

  - deux représentations visuelles



Un autre exemple

- Un meilleur algorithme pour calculer les moyennes préfixes:

Algorithm prefixAverages2(X):
Entrée: Un vecteur de nombres X à n éléments.
Sortie: Un vecteur de nombres A à n éléments tel que A[i] est la moyenne des éléments X[0], ... , X[i].
Soit A un vecteur de n nombres.
$s \leftarrow 0$
for $i \leftarrow 0$ **to** $n - 1$ **do**
  $s \leftarrow s + X[i]$
  A[i] $\leftarrow s/(i + 1)$
return array A

- Analyse ...

Mathématiques à réviser

- Logarithmes et exposants
  - propriétés des logarithmes:

$$\log_b(xy) = \log_b x + \log_b y$$
$$\log_b(x/y) = \log_b x - \log_b y$$
$$\log_b x^{\alpha} = \alpha \log_b x$$
$$\log_b a = \frac{\log_x a}{\log_x b}$$

  - propriétés des exposants:

$$a^{(b+c)} = a^b a^c$$
$$a^{bc} = (a^b)^c$$
$$a^b / a^c = a^{(b-c)}$$
$$b = a^{\log_a b}$$
$$b^c = a^{c * \log_a b}$$

Mathématiques à réviser (suite)

- Plancher (*Floor*)

$\lfloor x \rfloor$ = le plus grand entier $\leq x$

- Plafond (*Ceiling*)

$\lceil x \rceil$ = le plus petit entier $\geq x$

- Sommations
  - définition générale:

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \ldots + f(t)$$

  - où f est une fonction, s est l'index de départ, et t est l'index d'arrivée

- Progression géométrique: $f(i) = a^i$
  - soit un entier $n \geq 0$ et un nombre réel $0 < a \neq 1$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

  - les progressions géométriques ont une croissance exponentielle.

Sujets avancés: techniques de justification simples

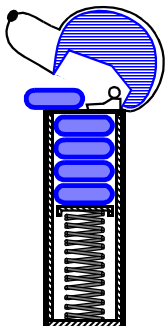
- Par exemple
  - Trouvez un exemple
  - Trouvez un contre-exemple
- Par contradiction (*"Contra" Attack*)
  - Trouvez une contradiction dans l'inverse de l'énoncé
  - Contrapositive
- Induction et invariants de boucle
  - Induction
    - 1) Prouvez le cas de base
    - 2) Prouvez que n'importe quel cas n implique que le prochain cas $(n + 1)$ est aussi vrai
  - Invariants de boucle
    - 1) Prouvez l'énoncé initial S_0
    - 2) Démontrez que S_{i-1} implique que S_i sera vrai après l'itération i

Sujets avancés: autres techniques de justification

- Preuve par excès d'agitation des mains
- Preuve par diagramme incompréhensible
- Preuve par corruption
  - voir le professeur ou l'AE après la classe
- La méthode des nouveaux habits de l'Empereur
  - "Cette preuve est tellement évidente que seul un idiot serait incapable de la comprendre"

PILES, FILES ET LISTES CHAÎNÉES

- Types abstraits de données (TAD)
- Piles
- Exemple: Analyse boursière
- Files
- Listes chaînées
- Files à deux bouts (*deques*)

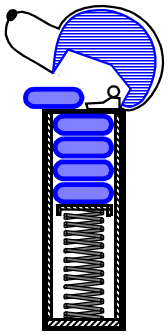


Types abstraits de données (TAD)

- Un **type abstrait de données** (*Abstract Data Type —ADT*) est une abstraction de structure de données: aucun codage n'est impliqué.
- Un TAD spécifie:
  - ce qui est contenu dans le TAD
  - les opérations qui peuvent être effectuées sur ou par le TAD.
- Par exemple, si nous cherchons à modéliser un sac de billes avec un TAD, nous pourrions spécifier que:
  - ce TAD contient des billes
  - ce TAD supporte l'insertion d'une bille et le retrait d'une bille.
- Il y a beaucoup de TAD standards et formalisés. Un sac de billes n'est pas l'un d'entre eux.
- Dans ce cours, nous apprendrons différents TAD standards (piles, files, listes...).

Piles (*Stacks*)

- Une pile est un contenant pour des objets insérés et retirés selon le principe dernier entré, premier sorti (*last-in-first-out*, ou *LIFO*).
- Les objets peuvent être insérés à tout moment, mais seulement le dernier (le plus récemment inséré) peut être retiré.
- Insérer un item correspond à empiler l'item (*pushing*). Dépiler la pile (*popping*) correspond au retrait d'un item.
- Analogie: distributeur de bonbons PEZ®

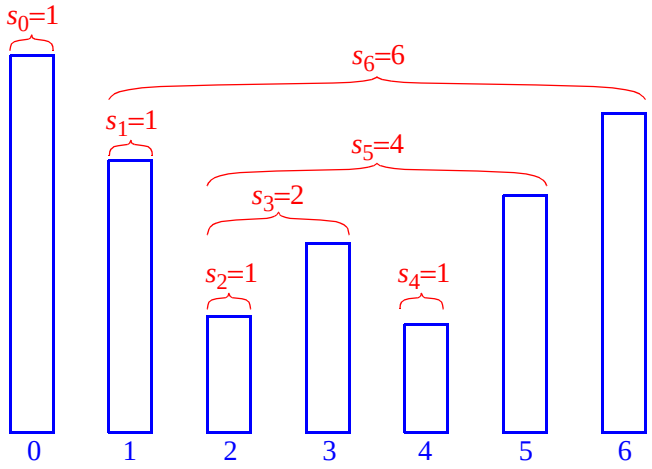


Le TAD Pile (ou Stack)

- Une pile est un type abstrait de données (TAD) qui supporte deux méthodes principales:
  - push(*o*): Insère l'objet *o* sur le dessus de la pile.
  - pop(): Retire l'objet du dessus de la pile et retourne-le; si la pile est vide, alors une erreur survient.
- Les méthodes secondaires suivantes devraient aussi être définies:
  - size(): Retourne le nombre d'objets dans la pile.
  - isEmpty(): Retourne un booléen indiquant si la pile est vide.
  - top(): Retourne l'objet du dessus de la pile, sans le retirer; si la pile est vide, alors une erreur survient.

Exemple

- L'**étendue** (*span*) du prix d'une action à un certain jour, *d*, est le nombre maximum de jours consécutifs (jusqu'à aujourd'hui) où le prix de l'action a été plus bas ou égal à son prix au jour *d*.



Un algorithme inéfficace

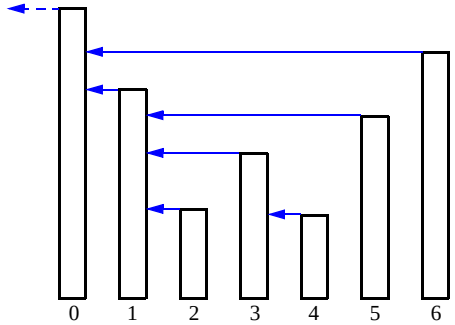
- Il y a une façon directe de calculer l'étendue d'une action à un jour donné pour *n* jours:

```
Algorithm computeSpans1(P):
  Entrée: Un vecteur de nombres P à n éléments.
  Sortie: Un vecteur de nombres A à n éléments tel que
          S[i] est l'étendue de l'action au jour i
  Soit S un vecteur de n nombres
  for i=0 to n–1 do
    k ←0
    done←false
    repeat
      if P[i-k] ≤P[i] then
        k←k+1
      else
        done←true
    until (k=i) or done
    S[i]←k
  return array S
```

- Le temps d'exécution de cet algorithme est (ouf!) $O(n^2)$. Pourquoi?

Une pile peut aider!

- Nous voyons que s_i au jour *i* peut être calculé facilement si nous connaissons le jour le plus proche avant *i* où le prix est plus haut lors de ce jour que le prix au jours *i*. Si un tel jour existe, appelons-le h_i.
- L'étendue est maintenant définie par $s_i = i - h_i$



Nous utilisons une ***pile*** pour calculer h_i

Étude de cas: Une *applet* pour analyse boursière (suite)

- Le pseudo-code pour notre nouvel algorithme:

```
Algorithm computeSpan2(P):
  Entrée: Un vecteur de nombres P à n éléments.
  Sortie: Un vecteur de nombres A à n éléments tel que
          S[i] est l'étendue de l'action au jour i
  Soit S un vecteur de n nombres et D une pile vide
  for i=0 to n-1 do
    done←false
    while not(D.isEmpty() or done) do
      if P[i]≥P[D.top()] then
        D.pop()
      else
        done←true
      if D.isEmpty() then
        h← -1
      else
        h←D.top()
      S[i]←i-h
      D.push(i)
    return array S
```

- Analysons le temps d'exécution de computeSpan2...

À propos de Java

- Étant donné le TAD pile, nous devons coder cet ADT afin de l'utiliser dans nos programmes.
- Vous devez comprendre deux concepts de programmation: les **interfaces** et les **exceptions**.
- Une interface est une façon de déclarer ce qu'une classe peut faire. Elle n'indique pas comment le faire.
- Pour une interface, vous écrivez simplement les noms de méthodes et leurs paramètres. Ce qui est important dans un paramètre est son **type**.
- Plus tard, quand vous écrirez une classe pour cette interface, vous coderez alors le contenu de ces méthodes.
- Séparer l'interface de la réalisation est une technique de programmation très utile. Exemple d'interface:

```
public interface radio {
   public void play();
   public void stop();
}
```


Une interface de pile en Java

- Même si la structure de donnée pile est déjà incluse comme classe Java dans le "*package*" java.util, il est possible, et parfois même préférable, de définir votre propre pile spécifique, comme ceci:

```
public interface Stack {

// accessor methods
  public int size(); // return the number of
                     // elements in the stack
  public boolean isEmpty();  // see if the stack
                             // is empty
  public Object top() // return the top element
    throws StackEmptyException; // if called on
                                // an empty stack

// update methods
  public void push (Object element); // push an
       // element onto the stack. Note that
       // the type of the parameter is
       // specified as an Object
  public Object pop() // return and remove the
                      // top element of the stack
    throws StackEmptyException; // if called on
                                // an empty stack
}
```


Exceptions

- Les **exceptions** sont un autre concept de programmation très utile, surtout dans un contexte de gestion d'erreurs.
- Quand vous détectez une erreur (ou un cas ***exceptionel***), vous lancez (***throw***) une exception.
- Exemple

```
public void mangePizza() throws MalAuVentreException
{
   ...

   if (tropMangé)
      throw new MalAuVentreException("Ouch");

   ...
}
```

- Aussitôt l'exception lancée, le flux de contrôle sort de la méthode en cours d'exécution.
- Alors quand MalAuVentreException est lancée, nous sortons de la méthode mangePizza() pour aller là où cette méthode a été appelée.

Encore des exceptions

- Supposons que le fragment de code suivant ait appelé la méthode mangePizza() en premier lieu.

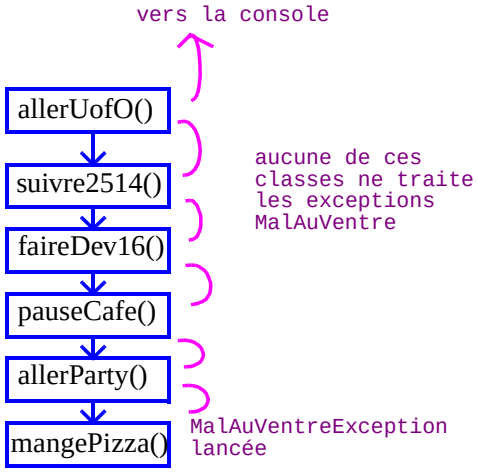
```
private void simuleRencontre()
{
   ...
   try
     {
       unStupideAE.mangePizza();
     }
   catch(MalAuVentreException e)
     {
       System.out.println("quelqu'un a mal au ventre");
     }
   ...
}
```

Toujours des exceptions

- Nous retournerons à unStupideAE.mangePizza(); parce que, souvenez-vous, mangePizza() lança l'exception.
- Le bloc try et le bloc catch indiquent que nous sommes à l'écoute des exceptions qui sont spécifiées dans le paramètre de catch.
- Parce que catch est à l'écoute de MalAuVentreException, le contrôle ira au bloc catch, et System.out.println sera alors exécuté.
- Notez que le bloc catch peut contenir n'importe quoi, pas seulement un System.out.println. Vous pouvez gérer les erreurs détectées comme bon vous semble, et vous pouvez même les relancer.
- Notez aussi que si vous lancez une exception dans votre méthode, vous devez ajouter une clause throws à la suite du nom de votre méthode.
- Pourquoi utiliser les exceptions? Vous pouvez déléguer vers le haut la responsabilité de traiter les erreurs, c'est-à-dire que le code qui a appelé la méthode en cours aura à gérer le problème.

Toujours des exceptions

- Si vous ne traitez pas une exception (avec catch), elle sera propagée vers le haut le long de la chaîne d'appels de méthodes jusqu'à ce que l'utilisateur l'observe.



Exceptions finales

- Ainsi, nous savons comment lancer et traiter des exceptions. Mais que sont-elles exactement en Java? Des classes!
- Observez MalAuVentreException.

```
public class MalAuVentreException extends
  RuntimeException {
  public MalAuVentreException(String err)
    {
      super(err);
    }
}
```


Pile à base de vecteur

- Créez une pile en utilisant un vecteur et en spécifiant une taille maximale N, par ex. N = 1 024.
- La pile est composée d'un vecteur de N éléments S et d'une variable entière t, l'index de l'élément au-dessus de la pile S.

- Les indices acceptables pour ce vecteur commencent à 0, alors nous initialisons t à -1.
- Pseudo-code

```
Algorithm size():
  return t +1

Algorithm isEmpty():
  return (t < 0)

Algorithm top():
  if isEmpty() then
    throw a StackEmptyException
  return S[t]
...
```

Pile à base de vecteur (suite)

- Pseudo-Code (suite)

Algorithm push(*o*):
  if size() = *N* **then**
    throw a StackFullException
  t ← *t* + 1
  S[*t*] ← *o*

Algorithm pop():
  if isEmpty() **then**
    throw a StackEmptyException
  e←*S*[*t*]
  S[*t*]←**null**
  t←*t*-1
  return *e*

- Chacune des méthodes ci-haut a un temps d'exécution constant ($O(1)$)
- La réalisation avec vecteur est simple et efficace.
- Il y a une limite supérieure, N, pour la taille de la pile. Une valeur arbitraire N pourrait être trop petite pour une application, ou gaspiller de la mémoire.

Pile à base de vecteur: Une réalisation en Java

```
public class ArrayStack implements Stack {
  // Implementation of the Stack interface
  // using an array.

  public static final int CAPACITY = 1000;  // default
                    // capacity of the stack
  private int capacity; // maximum capacity of the
                    // stack.
  private Object S[ ]; // S holds the elements of
                   // the stack
  private int top = -1; // the top element of the
                    // stack.

  public ArrayStack( ) { // Initialize the stack
     this(CAPACITY);// with default capacity
  }

  public ArrayStack(int cap) { // Initialize the
                // stack with given capacity
     capacity = cap;
     S = new Object[capacity];
  }
```


Pile à base de vecteur — Réalisation en Java (suite)

```
public int size( ) { //Return the current stack
                  // size
    return (top + 1);
}
public boolean isEmpty( ) {  // Return true iff
                     // the stack is empty
    return (top < 0);
}

public void push(Object obj) {  // Push a new
                        // object on the stack
    if (size() == capacity) {
      throw new StackFullException("Stack overflow.");
    }
    S[++top] = obj;
}

public Object top( ) // Return the top stack
                  // element
    throws StackEmptyException {
    if (isEmpty( )) {
      throw new StackEmptyException("Stack is
        empty.");
      }
    return S[top];
}
```


Pile à base de vecteur — Réalisation en Java (suite)

```
public Object pop() // Pop off the stack element
    throws StackEmptyException {
  Object elem;
  if (isEmpty( )) {
     throw new StackEmptyException("Stack is Empty.");
  elem = S[top];
  S[top--] = null; // Dereference S[top] and
                // decrement top
  return elem;
  }
}
```

Pile extensible à base de vecteur

- Au lieu d'abandonner avec StackFullException, nous pouvons remplacer le vecteur S par un plus grand vecteur et continuer à traiter les opérations *push*.

```
Algorithm push(o):
  if size() = N then
    A ← new array of length f(N)
   for i ← 0 to N – 1
     A[i] ← S[i]
   S ← A
  t ← t + 1
  S[t] ← o
```

- ***De quelle taille devrait être le nouveau vecteur?***
  - ***stratégie ajustée*** (additionner c): $f(N) = N + c$
  - ***stratégie de croissance*** (doubler): $f(N) = 2N$
- Afin de comparer ces deux stratégies, nous utiliserons le modèle de coût suivant:

opération *push* régulière: ajouter un élément	1
opération *push* spéciale: créer un vecteur de taille $f(N)$, copier N éléments, et ajouter un élément	$f(N) + N + 1$

Stratégie ajustée (c=4)

- Débuter avec un vecteur de taille 0
- Le coût d'une opération *push* spéciale est $2N + 5$

push	phase	n	N	coût
1	1	0	0	5
2	1	1	4	1
3	1	2	4	1
4	1	3	4	1
5	2	4	4	13
6	2	5	8	1
7	2	6	8	1
8	2	7	8	1
9	3	8	8	21
10	3	9	12	1
11	3	10	12	1
12	3	11	12	1
13	4	12	12	29

Performance de la stratégie ajustée

- Nous considérons k phases, où $k = n/c$
- Chaque phase correspond à une nouvelle taille de vecteur
- Le coût d'une phase i est de $2ci$
- le coût total de n opérations *push* est le coût total de k phases, avec $k = n/c$:

  $2c\ (1 + 2 + 3 + ... + k)$,

  qui est $O(k^2)$ et $\mathbf{O(n^2)}$.

Stratégie de croissance

- Débuter avec un vecteur de taille 0, ensuite 1, 2, 4, ...
- Le coût d'un *push* spécial est de $3N + 1$, où $N > 0$

push	phase	n	N	coût
1	0	0	0	2
2	1	1	1	4
3	2	2	2	7
4	2	3	4	1
5	3	4	4	13
6	3	5	8	1
7	3	6	8	1
8	3	7	8	1
9	4	8	8	25
10	4	9	16	1
11	4	10	16	1
12	4	11	16	1
...	...	...	...	...
16	4	15	16	1
17	5	16	16	49

Performance de la stratégie de croissance

- Nous considérons k phases, où $k = \log n$
- Chaque phase correspond à une nouvelle taille de vecteur
- Le coût d'une phase i est de 2^{i+1}
- le coût total de n opérations *push* est le coût total de k phases, avec $k = \log n$

  $2 + 4 + 8 + \ldots + 2^{\log n + 1} =$

  $2n + n + n/2 + n/4 + \ldots + 8 + 4 + 2 = 4n - 1$

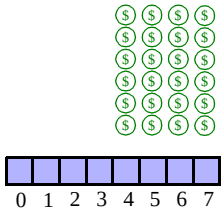
- La stratégie de croissance gagne!

Analyse amortie

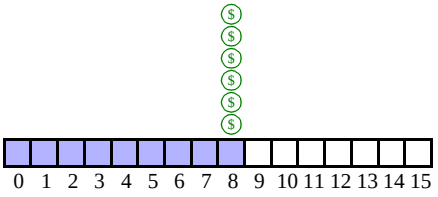
- Le **temps d'exécution amorti** d'une opération parmi une série d'opérations est le temps d'exécution du pire des cas de la série d'opérations toute entière divisé par le nombre d'opérations.
- La **méthode de comptabilité** détermine le temps d'exécution amorti à l'aide d'un système de crédits et de débits.
- Nous considérons l'ordinateur comme un appareil à sous qui exige un cyber-dollar pour une quantité constante de temps de calcul.
- Nous fixons un procédé pour facturer les opérations. Il s'agit là d'un ***procédé d'amortissement***.
- Nous pouvons surfacturer certaines opérations et en sousfacturer d'autres. Par exemple, nous pouvons ***facturer un même montant pour chaque opération***.
- Le procédé doit toujours nous procurer suffisament d'argent pour payer le coût réel de l'opération.
- Le coût total de la série d'opérations n'est pas plus élevé que le montant total facturé.
- (temps amorti) ≤ (total $ facturé) / (# opérations)

Procédé d'amortissement pour la stratégie de croissance

- À la fin d'une phase, nous devons avoir assez économisé pour payer le *push* spécial de la phase suivante.
- À la fin de la phase 3, il faut avoir économisé $24.



- Les économies payent pour la croissance du vecteur.



- Nous facturons **$7** pour un *push*. Les $6 économisés par *push* régulier sont "conservés" dans la seconde moitié du vecteur.

Analyse d'amortissement pour la stratégie de croissance

- Nous facturons **$5** (offre spéciale de lancement) pour le premier *push* et **$7** pour les suivants.

push	*n*	*N*	***solde***	***facture***	coût
1	0	0	$0	$5	$2
2	1	1	$3	$7	$4
3	2	2	$6	$7	$7
4	3	4	$6	$7	$1
5	4	4	$12	$7	$13
6	5	8	$6	$7	$1
7	6	8	$12	$7	$1
8	7	8	$18	$7	$1
9	8	8	$24	$7	$25
10	9	16	$6	$7	$1
11	10	16	$12	$7	$1
12	11	16	$18	$7	$1
...	...	...	...	...	...
16	15	16	$42	$7	$1
17	16	16	$48	$7	$49

"Casting" avec une pile générique

- Avoir un ArrayStack qui peut contenir seulement des objets Entier ou des objets Étudiant.
- Afin de réaliser ceci à l'aide d'une pile générique, les objets retournés doivent être "moulés" (*cast*) dans le bon type de donnée.
- Un exemple en Java:

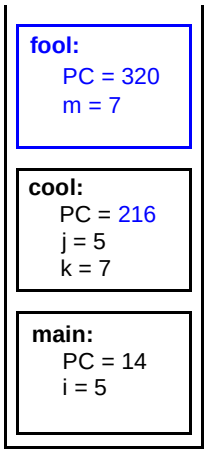
```
public static Integer[] reverse(Integer[] a) {
  ArrayStack S = new ArrayStack(a.length);
  Integer[] b = new Integer[a.length];
  for (int i = 0; i < a.length; i++)
    S.push(a[i]);
  for (int i = 0; i < a.length; i++)
    b[i] = (Integer)(S.pop()); // the popping
      // operation gave us an Object, and we
      // casted it to an Integer before
      // assigning it to b[i].
  return b;
}
```


Piles dans la Machine Virtuelle Java (JVM)

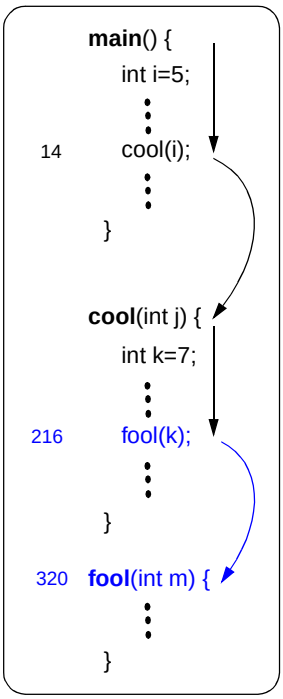
- Chaque processus en exécution dans un programme Java a sa propre pile de méthodes (Method Stack).
- Chaque fois qu'une méthode est appelée, elle est empilée sur une telle pile.
- L'utilisation d'une pile pour cette opération permet à Java de faire plusieurs choses utiles:
  - Exécuter des appels récursifs de méthode
  - Afficher la trace d'une pile pour localiser une erreur.
- Java inclut aussi une pile d'opérandes qui est utilisée pour évaluer les instructions arithmétiques:

```
Integer add(a, b):
  OperandStack Op
  Op.push(a)
  Op.push(b)
  temp1 ← Op.pop()
  temp2 ← Op.pop()
  Op.push(temp1 + temp2)
  return Op.pop()
```


Pile de méthodes Java



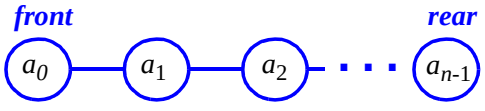
Pile Java



Programme Java

Files (*Queues*)

- Une file se distingue d'une pile par ses routines d'insertion et de retrait qui suivent le principe premier entré, premier sorti (*first-in-first-out*, ou *FIFO*).
- Des éléments peuvent être insérés à tout moment, mais seulement l'élément qui a été le plus longtemps dans la file peut être retiré.
- Les éléments sont enfilés (*enqueued*) par l'arrière (*rear*) et défilé (*dequeued*) par l'avant (*front*)



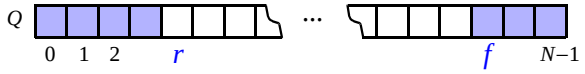
Le TAD File (ou Queue)

- La file supporte deux méthodes fondamentales:
  - enqueue(*o*): Insère l'objet *o* à l'arrière de la file
  - dequeue(): Retire l'objet du devant de la file et retourne-le; une erreur survient lorsque la file est vide
- Les méthodes secondaires suivantes devraient aussi être définies:
  - size(): Retourne le nombre d'objets dans la file
  - isEmpty(): Retourne un booléen indiquant si la pile est vide
  - front(): Retourne, sans le retirer, l'objet au devant de la file; si la pile est vide, alors une erreur survient

File à base de vecteur

- Créez une file en utilisant un vecteur circulaire.
- Spécifiez une taille maximale *N*, par ex. *N* = 1 000.
- La file est composée d'un vecteur de *N* éléments *Q* et de deux variables entières:
  - *f*, l'index de l'élément du devant
  - *r*, l'index de l'élément suivant celui de l'arrière
- Configuration "normale"

- Configuration circulaire ("*wrapped around*")



- Que veut dire *f*=*r*?

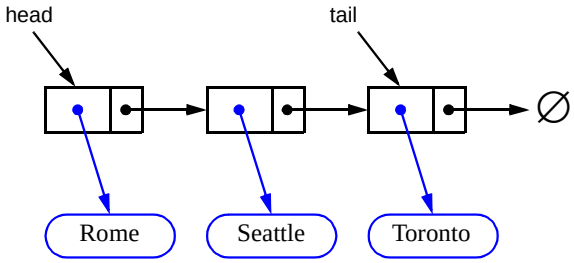
File à base de vecteur (suite)

- Pseudo-code

```
Algorithm size():
  return (N - f + r) mod N

Algorithm isEmpty():
  return (f = r)

Algorithm front():
  if isEmpty() then
    throw a QueueEmptyException
  return Q[f]

Algorithm dequeue():
  if isEmpty() then
    throw a QueueEmptyException
  temp ← Q[f]
  Q[f] ← null
  f ← (f + 1) mod N
  return temp

Algorithm enqueue(o):
  if size = N - 1 then
    throw a QueueFullException
  Q[r] ← o
  r ← (r +1) mod N
```

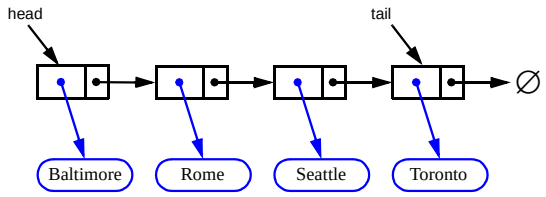

Réalisation d'une file à l'aide d'une liste simplement chaînée

- nœuds connectés en chaîne par des liens (*links*)

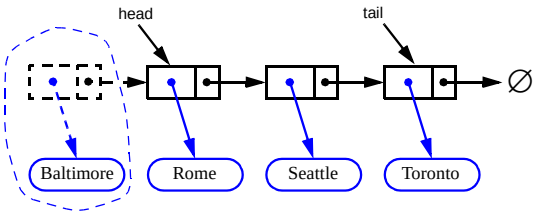


- la tête (*head*) de la liste est le devant de la file, la queue de la liste (*tail*) est le derrière de la file.
- pourquoi pas le contraire?

Retirer l'élément de tête



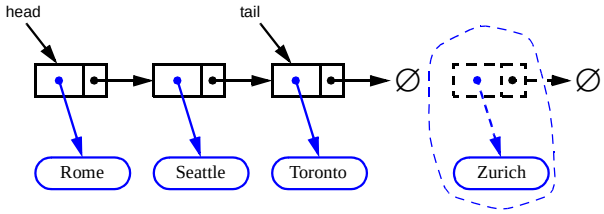
- avancez la référence de la tête



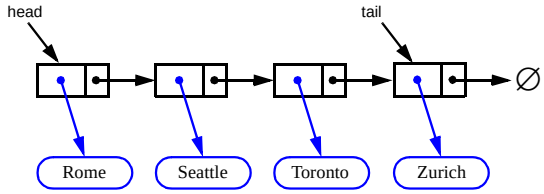
- insérer un élément à la tête est tout aussi facile.

Insérer un élément à la queue

- créez un nouveau nœud



- enchaînez-le et déplacez la référence à la queue



- comment retirer l'élément de queue?

Files à deux bouts (Double-Ended Queues)

- une file à deux bouts, ou deque, supporte l'insertion et le retrait à l'avant comme à l'arrière.
- Le TAD Deque:
  - insertFirst(*e*): Insère *e* au début de la deque
  - insertLast(*e*): Insère *e* à la fin de la deque
  - removeFirst(): retire et retourne le premier élément
  - removeLast(): retire et retourne le dernier élément
- Les méthodes secondaires incluent:
  - first()
  - last()
  - size()
  - isEmpty()

Réalisations de piles et de files à l'aide de Deques

- Piles avec Deques:

Méthode de Pile	Réalisation avec Deque
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

- Files avec Deques:

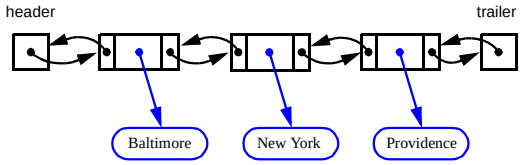
Méthode de File	Réalisation avec Deque
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

Le patron de conception Adaptateur (*Adaptor Pattern*)

- L'utilisation d'une deque pour réaliser une pile ou une file est un exemple du patron de conception adaptateur (*adoptor pattern*). Ce patron réalise une classe en utilisant des méthodes d'une autre classe.
- Souvent, les classes *adaptateur* spécialisent des classes générales.
- Voici deux applications:
  - Spécialisation d'une classe générale en changeant quelques méthodes:
    Ex: réalisation d'une pile avec une deque.
  - Spécialisation de types d'objets utilisés par une classe générale:
    Ex: définir une classe IntegerArrayStack qui adapte ArrayStack pour ne contenir que des entiers.

Réalisation de deques à l'aide de listes doublement chaînées

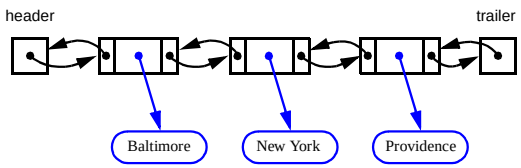
- Effacer l'élément de queue d'une liste simplement chaînée ne peut pas être fait en un temps constant.
- Pour réaliser une deque, nous utilisons une liste doublement chaînée avec des nœuds spéciaux pour l'avant (*header*) et l'arrière (*trailer*).



- Un nœud de liste doublement chaînée a un lien suivant (*next*) et un lien précédent (*prev*). Ce nœud supporte les méthodes suivantes:
  - setElement(Object e)
  - setNext(Object newNext),
  - setPrev(Object newPrev)
  - getElement(), getNext(), getPrev()
- En utilisant une liste doublement chaînée, toutes les méthodes de deque ont un temps d'exécution constant (c'est-à-dire,O(1))!

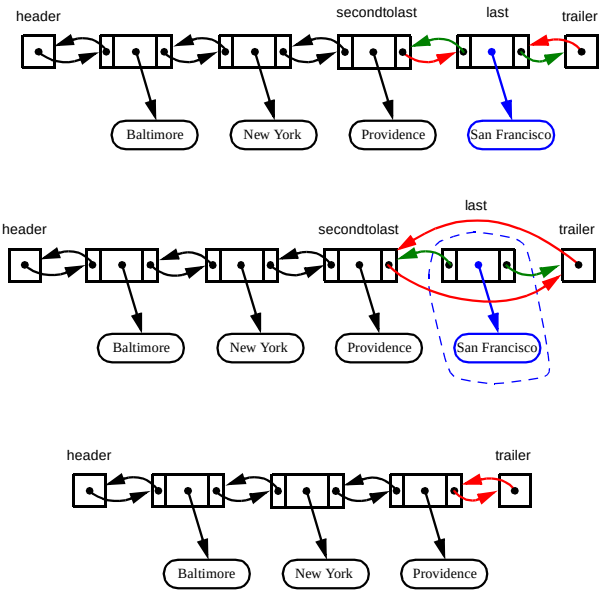
Réalisation de deques à l'aide de listes doublement chaînées (suite)

- En réalisant une liste doublement chaînée, nous ajoutons deux nœuds spéciaux aux extrémités: les nœuds *header* et *trailer.*
  - Le nœud *header* est placé avant le premier élément de la liste. Il a un prochain lien valide, mais un lien précédent vide.
  - Le nœud *trailer* est placé après le dernier élément de la liste. Il a un lien précédent valide, mais un prochain lien vide.
- les nœuds *header* et *trailer* sont des sentinelles ou nœuds "bidon" parce qu'ils ne contiennent pas d'éléments.
- Diagramme de notre liste doublement chaînée:



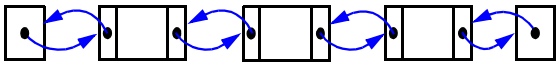
Réalisation de deques à l'aide de listes doublement chaînées (suite)

- Visualisons le code de removeLast().



SÉQUENCES

- Vecteurs
- Positions
- Listes
- Séquences générales
- Étude de cas: le tri à bulle (*Bubble Sort*)



Le TAD Vecteur (*Vector*)

- Une séquence S (avec n éléments) qui supporte les méthodes suivantes:

  - elemAtRank(r):
    Retourne l'élément de S au rang r; une erreur survient si $r < 0$ ou $r > n - 1$

  - replaceAtRank(r,e):
    Remplace l'élément au rang r avec e et retourne l'ancien élément; une erreur survient si $r < 0$ ou $r > n - 1$

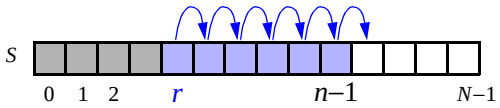
  - insertAtRank(r,e):
    Insère un nouvel élément dans S qui aura le rang r; une erreur survient si $r < 0$ ou $r > n - 1$

  - removeAtRank(r):
    Retire de S l'élément au rang r; une erreur survient si $r < 0$ ou $r > n - 1$

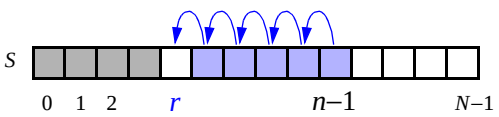
Réalisation avec vecteur (*array*)

- Extraits de pseudo-code:

```
Algorithm insertAtRank(r,e):
  for i = n - 1, n - 2, ... , r do
    S[i+1] ← s[i]
  S[r] ← e
  n ← n + 1
```




```
Algorithm removeAtRank(r):
  e ← S[r]
  for i = r, r + 1, ... , n - 2 do
    S[i] ← S[i + 1]
  n ← n - 1
  return
```



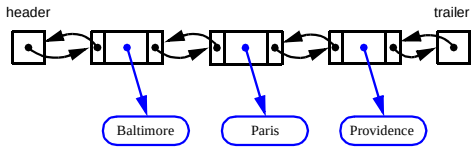
Réalisation avec vecteur (suite)

- Complexité temporelle des diverses méthodes:

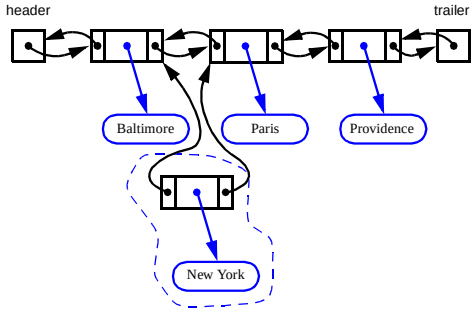
Méthode	Temps
size	$O(1)$
isEmpty	$O(1)$
elemAtRank	$O(1)$
replaceAtRank	$O(1)$
insertAtRank	$O(n)$
removeAtRank	$O(n)$

Réalisation avec liste doublement chaînée

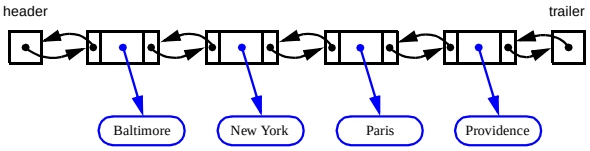
- la liste avant une insertion:



- création d'un nouveau nœud à insérer:



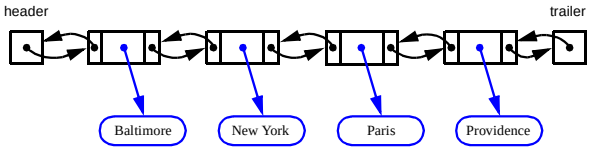
- la liste après l'insertion:



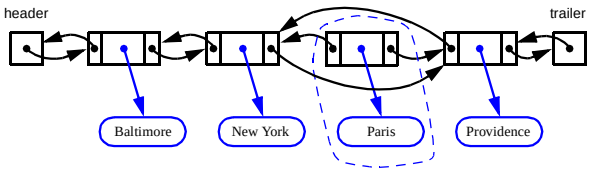

```
public void insertAtRank (int rank, Object element)
   throws BoundaryViolationException {
     if (rank < 0 || rank > size())
        throw new BoundaryViolationException("invalid rank");
     DLNode next = nodeAtRank(rank); // the new node
                    //will be right before this
     DLNode prev = next.getPrev(); // the new node
                    //will be right after this
     DLNode node = new DLNode(element, prev, next);
     // new node knows about its next & prev. Now
     // we tell next & prev about the new node.
     next.setPrev(node);
     prev.setNext(node);
     size++;
   }
```


Réalisation avec liste doublement chaînée (suite)

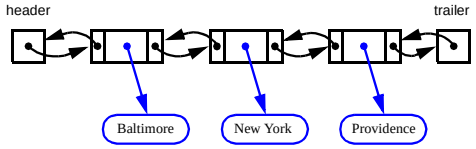
- la liste avant une suppression:



- suppression d'un nœud:



- la liste après la suppression:



Réalisation en Java

- code pour supprimer un nœud

```
public Object removeAtRank (int rank)
                  throws BoundaryViolationException {
   if (rank < 0 || rank > size()–1)
      throw new BoundaryViolationException("Invalid
          rank.");
   DLNode node = nodeAtRank(rank);  // node to
                                    // be removed
   DLNode next = node.getNext(); // node before it
   DLNode prev = node.getPrev(); // node after it
   prev.setNext(next);
   next.setPrev(prev);
   size--;
   return node.getElement(); // returns the
             // element of the deleted node
   }
```

Réalisation en Java (suite)

- code pour trouver un nœud à un certain rang

```
private DLNode nodeAtRank (int rank) {
  // auxiliary method to find the node of the
  // element with the given rank. We make
  // auxiliary methods private or protected.
  DLNode node;
  if (rank <= size()/2) { //scan forward from head
     node = header.getNext();
     for (int i=0; i < rank; i++)
     node = node.getNext();
  }
  else { // scan backward from the tail
     node = trailer.getPrev();
     for (int i=0; i < size()-rank-1 ; i++)
     node = node.getPrev();
  }
  return node;
}
```


Nœuds

- Les listes chaînées supportent l'exécution efficace d'***opérations basées sur les nœuds***:
  - removeAtNode(Node v) et insertAfterNode(Node v, Object e), sont de complexité $O(1)$.
- Cependant, les opérations basées sur les nœuds ne sont pas significatives dans une réalisation basée sur un vecteur car il n'y a pas de nœud dans un vecteur.
- Les nœuds sont spécifiques à la réalisation.
- **Dilemme**:
  - Si nous ne définissons pas d'opérations basées sur les nœuds, nous ne profitons pas pleinement des listes doublement chaînées.
  - Si nous en définissons, nous violons la généralité des types abstraits de données.

De nœuds à positions

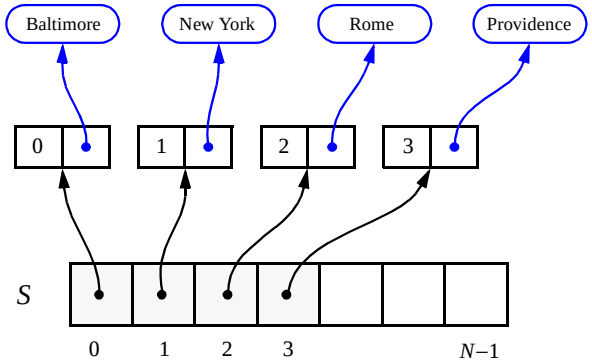
- Nous présentons le TAD ***Position***
- Notion intuitive de "place" d'un élément
- Les positions n'ont qu'une seule méthode: element(): Retourne l'élément à cette position
- Les positions sont définies relativement aux autres positions (relation avant/après)
- Les positions ne sont pas liées à un élément ou à un rang.

Le TAD Liste (*List*)

- TAD avec méthodes basées sur les positions
- méthodes génériques size(), isEmpty()
- méthodes de requête isFirst(p), isLast(p)
- méthodes accessoires first(), last(), before(p), after(p)
- méthodes de mise à jour swapElements(p,q), replaceElement(p,e), insertFirst(e), insertLast(e), insertBefore(p,e), insertAfter(p,e). remove(p)
- chaque méthode est de complexité $O(1)$ lorsque réalisées avec une liste doublement chaînée.

Le TAD Séquence

- Combine les TAD Vecteur et Liste (héritage multiple)
- Ajoute des méthodes qui jettent un pont entre rangs et positions
  - atRank(r) retourne une position
  - rankOf(p) retourne un rang (entier)
- Une réalisation basée sur un vecteur nécessite l'utilisation d'objets pour représenter les positions



Comparaison entre réalisations de séquences

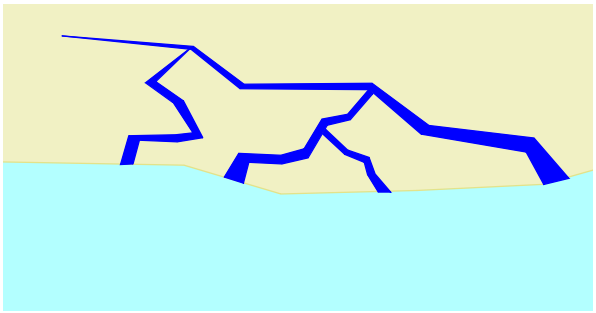
Opérations	Vecteur (*Array*)	Liste
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last	$O(1)$	$O(1)$
before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Itérateurs

- Abstraction du processus de recherche au sein d'une collection d'éléments (un élément à la fois)
- Patron de conception
- Encapsulation des notions de "place" et de "prochain"
- Extension du TAD Position
- Itérateurs génériques et spécialisés
- ***ObjectIterator***
  - hasNext()
  - nextObject()
  - object()
- ***PositionIterator***
  - nextPosition()
- Méthodes utiles qui retournent des itérateurs:
  - elements()
  - positions()

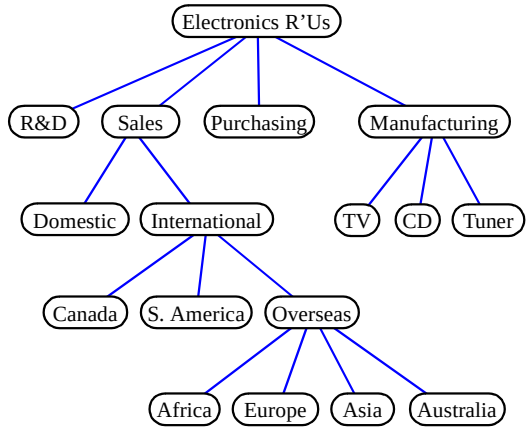
ARBRES

- Arbres
- Arbres binaires
- Traversées d'arbres
- Patron de conception: gabarit de méthode (*template method pattern*)
- Structures de données pour arbres

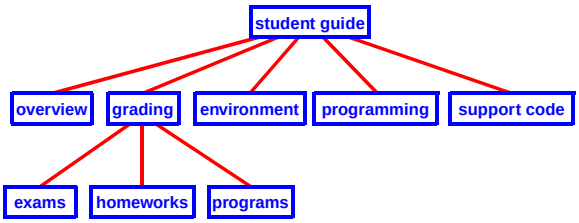


Arbres

- un arbre représente une hiérarchie
  - structure organisationnelle d'une corporation

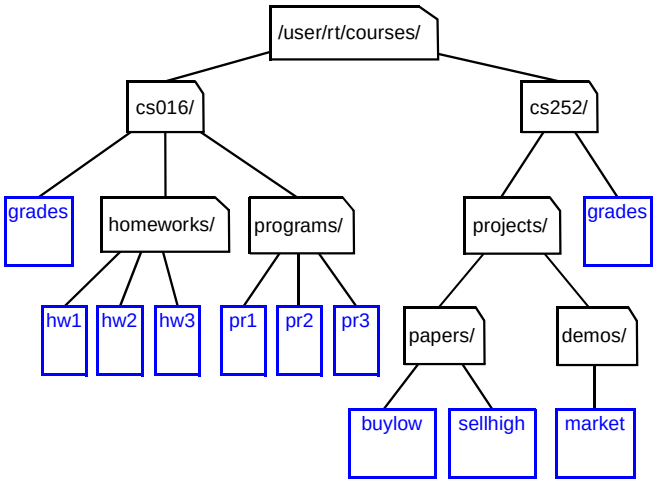


  - table des matières d'un livre



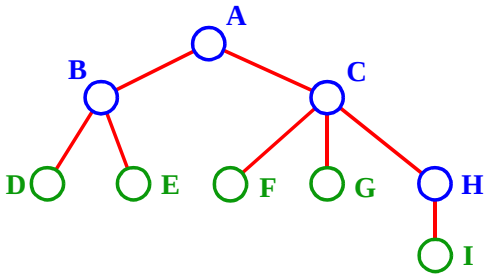
Un autre exemple

- Système de fichier de Unix ou de DOS/Windows



Terminologie

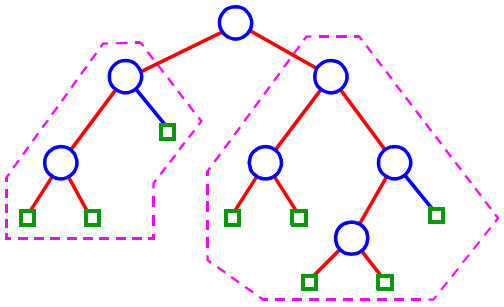
- **A** est le nœud ***racine***.
- **B** est le ***parent*** (ou père) de **D** et **E**.
- **C** est le ***frère*** (*sibling*) de **B**.
- **D** et **E** sont les ***enfants*** (ou descendants) de **B**.
- **D, E, F, G, I** sont des ***nœuds extérieurs***, ou ***feuilles***.
- **A, B, C, H** sont des ***nœuds intérieurs***.
- La ***profondeur*** (***niveau***) de **E** est ***2***
- La ***hauteur*** de l'arbre est ***3***.
- Le ***degré*** (nombre d'enfants) du nœud **B** est ***2***.



Propriété: (*# liens*) = (*#nœuds*) – 1

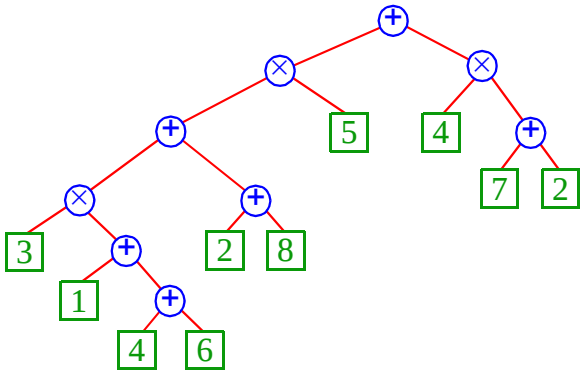
Arbres binaires

- ***Arbre ordonné:*** les enfants de chaque nœud sont ordonnés.
- ***Arbre binaire:*** arbre ordonné où tous les nœuds intérieurs sont de ***degré 2***.
- Définition récursive d'un arbre binaire:
- Un *arbre binaire* est
  - un nœud extérieur (feuille), **ou**
  - un nœud intérieur (la *racine*) et deux arbres binaires (***sous-arbre gauche*** et ***sous-arbre droit***)



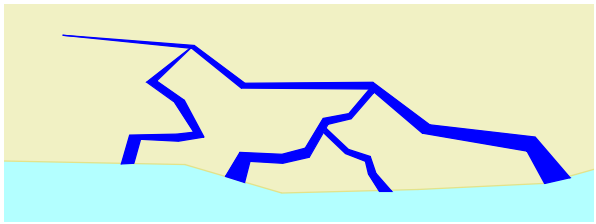
Exemples d'arbres binaires

- expression arithmétique



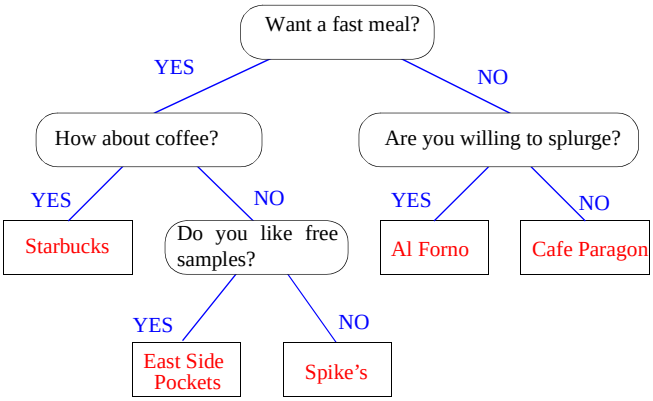
((((3 × (1 + (4 + 6))) + (2 + 8)) × 5) + (4 × (7 + 2)))

- rivière



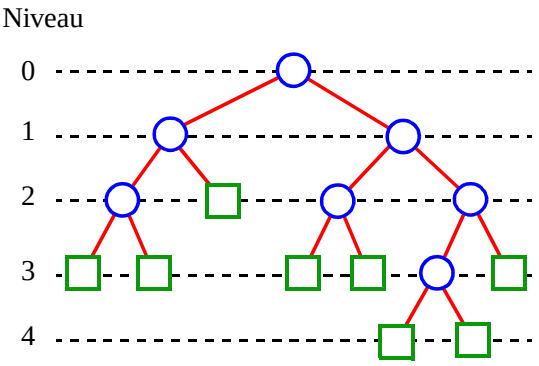
Exemples d'arbres binaires

- arbres de décision



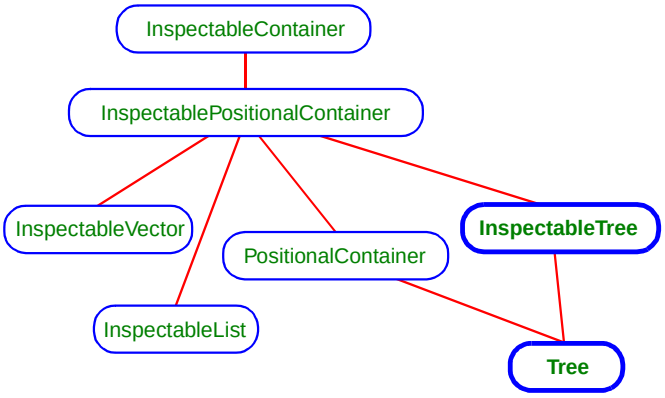
Propriétés des arbres binaires

- (# nœuds extérieurs) = (# nœuds intérieurs) + 1
- (# nœuds au niveau **i**) ≤ 2^{i}
- (# nœuds extérieurs) ≤ $2^{(\text{hauteur})}$
- (hauteur) ≥ $\log_2$ (# nœuds extérieurs)
- (hauteur) ≥ $\log_2$ (# nœuds) – 1
- (hauteur) ≤ (# nœuds intérieurs) = ((# nœuds) – 1)/2



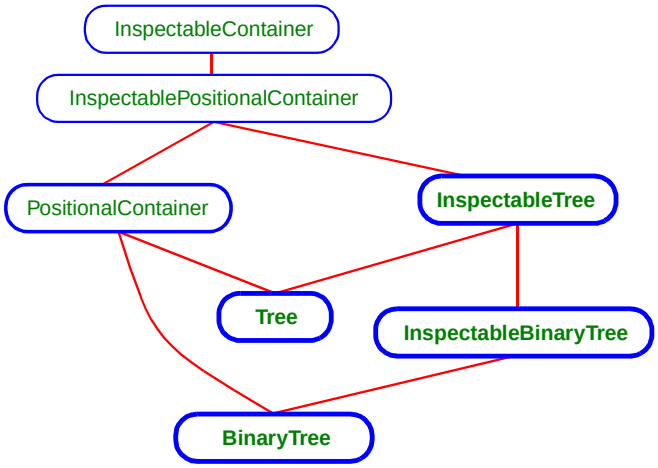
Le TAD Arbres (*Trees*)

- méthodes génériques de contenant
  - size(), isEmpty(), elements()
- méthodes positionnelles de contenant
  - positions(), swapElements(p,q), replaceElement(p,e)
- méthodes de requête
  - isRoot(p), isInternal(p), isExternal(p)
- méthodes accessoires
  - root(), parent(p), children(p)
- méthodes de mise à jour
  - spécifiques à l'application



TADs pour Arbres Binaires

- méthodes accessoires
  - leftChild(p), rightChild(p), sibling(p)
- méthodes de mise à jour
  - expandExternal(p), removeAboveExternal(p)
  - autres méthodes spécifiques à l'application

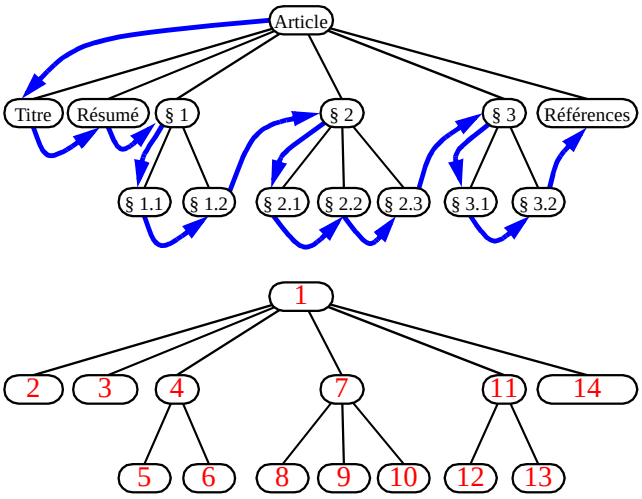


Traversée d'arbres

- traversée en pré-ordre

  Algorithm preOrder(v)
  "visit" node v
  for each child w of v **do**
  recursively perform preOrder(w)

- comme lire un document du début à la fin

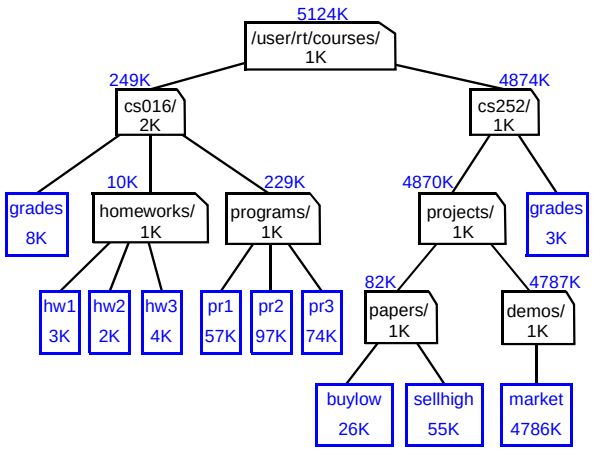


Traversée d'arbres

- traversée en post-ordre

  Algorithm postOrder(v)
  for each child w of v **do**
  recursively perform postOrder(w)
  "visit" node v

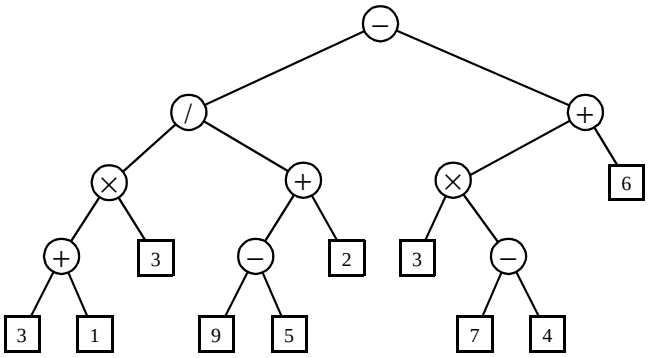
- commande Unix du (*disk usage*)



Évaluation d'expressions arithmétiques

- spécialisation d'une traversée post-ordre

```
Algorithm evaluateExpression(v)
  if v is an external node
    return the variable stored at v
  else
    let o be the operator stored at v
    x ← evaluateExpression(leftChild(v))
    y ← evaluateExpression(rightChild(v))
    return x o y
```

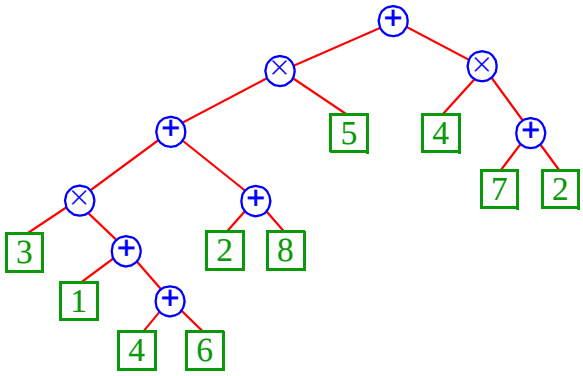


Traversée d'arbres binaires

- traversée in-ordre

```
Algorithm inOrder(v)
  recursively perform inOrder(leftChild(v))
  "visit" node v
  recursively perform inOrder(rightChild(v))
```

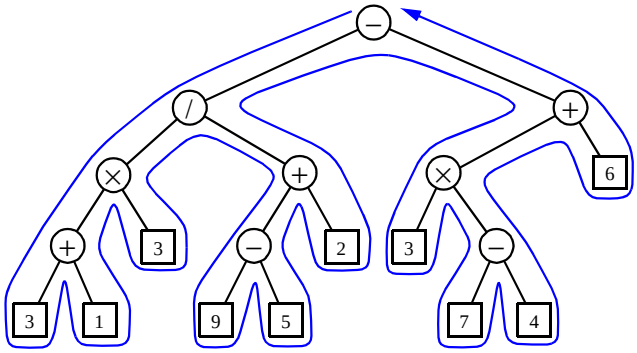
- afficher une expression arithmétique
  - spécialisation d'une traversée in-ordre
  - afficher "(" avant la traversée du sous-arbre gauche
  - afficher ")" après la traversée du sous-arbre droit



((((3 × (1 + (4 + 6))) + (2 + 8)) × 5) + (4 × (7 + 2)))

Traversée par tour d'Euler

- traversée générique d'un arbre binaire
- les traversées pré-ordre, in-ordre et post-ordre sont des cas spéciaux de la traversée par tour d'Euler
- "marche autour" de l'arbre et visite de chacun des nœuds à trois reprises:
  - à la gauche
  - par-dessous
  - à la droite



Gabarit de méthode (*Template Method Pattern*)

- mécanisme de calcul générique qui peut être spécialisé en redéfinissant certaines étapes (un autre patron de conception)
- réalisation en utilisant une classe abstraite Java avec des méthodes qui peuvent être redéfinies par ses sous-classes.

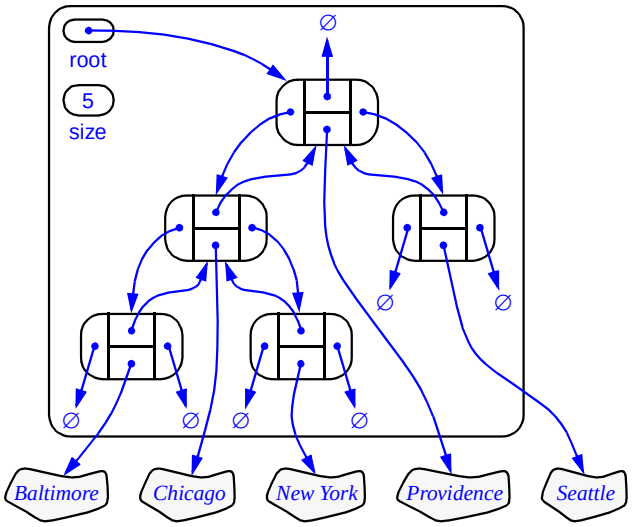
```
public abstract class BinaryTreeTraversal {

protected BinaryTree tree;
...
protected Object traverseNode(Position p) {
   TraversalResult r = initResult();
   if (tree.isExternal(p)) {
     external(p, r);
   } else {
     left(p, r);
     r.leftResult = traverseNode(tree.leftChild(p));
     below(p, r);
     r.rightResult = traverseNode(tree.rightChild(p));
     right(p, r);
   }
   return result(r);
 }
```

Spécialisation de la traversée générique d'arbres binaires

- affichage d'une expression arithmétique

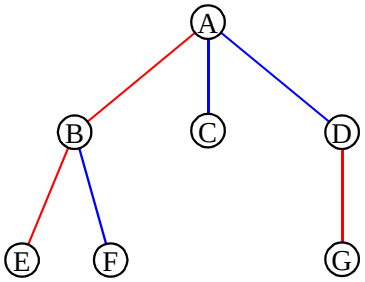
```
public class PrintExpressionTraversal
    extends BinaryTreeTraversal {

...

  protected void external(Position p, TraversalResult r) {
   System.out.print(p.element());
  }

  protected void left(Position p, TraversalResult r) {
   System.out.print("(");
  }

  protected void below(Position p, TraversalResult r) {
   System.out.print(p.element());
  }

  protected void right(Position p, TraversalResult r) {
   System.out.print(")");
  }

}
```


Structure de données chaînée pour arbres binaires

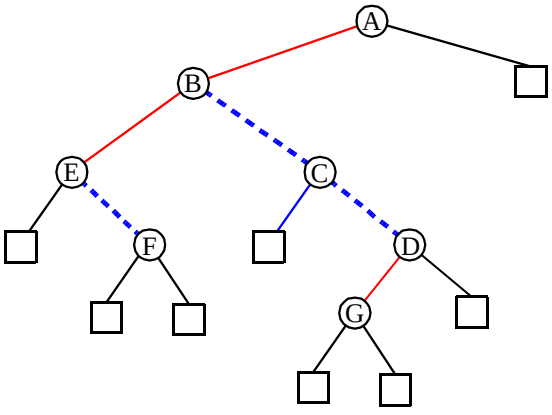


Représentation d'arbres généraux

- arbre T

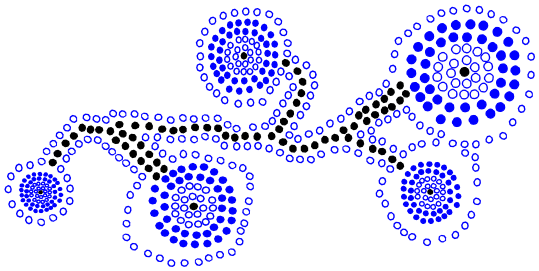


- arbre binaire T' représentant T



FILES À PRIORITÉ

- Application boursière (motivation)
- Le TAD file à priorité (*Priority Queue*)
- Réalisation d'une file à priorité avec une séquence
- Le tri (*sorting*)
- Problèmes liés au tri



Application boursière

- Nous nous concentrerons sur la vente d'un seul titre, Akamai Technologies, fondée en 1998 par des professeurs et des étudiants du MIT (200 employés, 20 milliards de dollars en capital action)
- Les investisseurs font des ***commandes*** qui comprennent trois items (***action***, ***prix***, ***quantité***), où ***action*** est un ***achat*** ou une ***vente***, ***prix*** est le pire prix que vous êtes prêt à débourser (achat) ou à accepter (vente), et ***quantité*** est le nombre d'actions
- À l'équilibre, toutes les commandes d'achat (***offres***) ont des prix plus bas que toutes les commandes de ventes (***demandes***)
- Une ***cote de niveau 1*** donne l'offre la plus haute et la demande la plus basse (telles que fournies par les sites financiers populaires et les courtiers ou *e-brokers*)
- Une ***cote de niveau 2*** donne toutes les offres et les demandes pour certains seuils de prix (Island ECN sur le Web et cotes pour agents professionnels (*traders*))
- Une ***transaction*** survient lorsqu'une nouvelle commande peut être jumelée à une ou plusieurs commandes existantes, ce qui résulte en une série de transactions de ***suppression***.
- Les commandes peuvent être ***annulées*** à tout moment.

Structures de données pour le marché boursier

- Pour chaque titre, nous conservons deux structures, la première pour les offres et la seconde pour les demandes
- Les opérations qui doivent être supportées:

Action	***Structure Offre***	***Structure Demande***
faire une commande	***insert***(prix,quantité)	***insert***(prix,quantité)
obtenir une cote de niveau 1	***min***()	***max***()
effectuer la transaction	***removeMin***()	***removeMax***()
annuler	***remove***(commande)	***remove***(commande)

- Ces structures de données sont appelées ***files à priorité***.
- Les files à priorité de la bourse NASDAQ supportent en moyenne un volume de transaction quotidien de 1 milliard d'actions (50 milliards de dollars)

Clés et relations d'ordre total

- Une file à priorité (*Priority Queue*) classe ses éléments par ***clé*** avec une relation ***d'ordre total***
- Clés:
  - Chaque élément a sa propre clé
  - Les clés ne sont pas nécessairement uniques
- Relation d'ordre total
  - Dénotée par $\leq$
  - **Réflexive:** $k \leq k$
  - **Antisymétrique:** si $k_1 \leq k_2$ et $k_2 \leq k_1$, alors $k_1 \leq k_2$
  - **Transitive:** si $k_1 \leq k_2$ et $k_2 \leq k_3$, alors $k_1 \leq k_3$
- Une file à priorité supporte ces méthodes fondamentales sur des paires clé-élément:
  - min()
  - insertItem(k, e)
  - removeMin()

Tri par file à priorité

- Une file à priorité P peut être utilisée pour trier une séquence S:
  - en insérant les éléments de S dans P avec une suite d'opérations insertItem(e, e)
  - en retirant les éléments de P en ordre croissant et en les remettant dans S avec une suite d'opérations removeMin()

Algorithm PriorityQueueSort(S, P):
Entrée: Séquence S contenant n éléments, avec une relation d'ordre total, et une file à priorité P qui compare les clés avec cette même relation
Sortie: Séquence S triée à l'aide de la relation d'ordre total

while !S.isEmpty() **do**
  $e \leftarrow S$.removeFirst()
  P.insertItem(e, e)
while P is not empty **do**
  $e \leftarrow P$.removeMin()
  S.insertLast(e)

Le TAD File à priorité

- Une file à priorité P supporte les méthodes suivantes:
  - size():
    Retourne le nombre d'éléments dans P
  - isEmpty():
    Vérifie si P est vide
  - insertItem(k,e):
    Insère un nouvel élément e avec sa clé k dans P
  - minElement():
    Retourne (mais ne retire pas) un élément de P à la plus petite clé; une erreur survient si P est vide
  - minKey():
    Retourne la plus petite clé de P; une erreur survient si P est vide
  - removeMin():
    Retire et retourne un élément de P à la plus petite clé; une erreur survient si P est vide.

Comparateurs

- Patron de conception (*Comparator*)
- La forme la plus générale et la plus réutilisable de file à priorité utilise des objets appelés **comparateurs.**
- Les comparateurs sont externes aux clés à comparer et permettent de comparer deux objets.
- Quand la file à priorité a besoin de comparer deux clés, elle utilise le comparateur qui lui a été fourni.
- Ainsi, une file à priorité peut être suffisamment générale pour contenir n'importe quel objet.
- Le TAD **Comparateur** inclut:
  - isLessThan(a, b)
  - isLessThanOrEqualTo(a,b)
  - isEqualTo(a, b)
  - isGreaterThan(a,b)
  - isGreaterThanOrEqualTo(a,b)
  - isComparable(a)

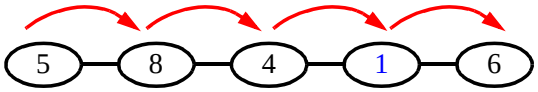
Réalisation avec séquence non-triée

- Essayons de réaliser une file à priorité avec une séquence non-triée S.
- Les éléments de S sont composés de k, la clé, et de e, l'élément.
- Nous pouvons réaliser insertItem() en utilisant insertLast() sur les séquences. Le temps d'exécution sera alors $\boldsymbol{O}(1)$.

- Cependant, comme nous insérons toujours à la fin, sans tenir compte de la valeur de la clé, notre séquence n'est pas ordonnée.

Réalisation avec séquence non-triée (suite)

- Ainsi, pour les méthodes telles minElement(), minKey(), et removeMin(), nous devons ***regarder tous les éléments*** de *S*. La complexité du pire des cas est $O(n)$.



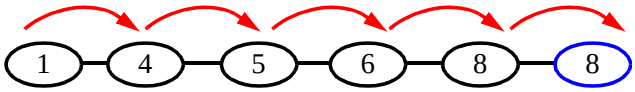
- Sommaire des performances

insertItem	$O(1)$
minKey, minElement	$O(n)$
removeMin	$O(n)$

Réalisation avec séquence triée

- Une autre réalisation possible utilise une séquence *S*, triée par ordre croissant de clés.
- minElement(), minKey(), et removeMin() deviennent alors $O(1)$

- Cependant, pour réaliser insertItem(), nous devons maintenant parcourir la séquence entière ***dans le pire des cas***. Ainsi, insertItem() s'exécute en un temps $O(n)$



- Sommaire des performances

insertItem	$O(n)$
minKey, minElement	$O(1)$
removeMin	$O(1)$

Réalisation avec séquence triée (suite)

```
public class SequenceSimplePriorityQueue
implements SimplePriorityQueue {
  //Implementation of a priority queue
  using a sorted sequence
 protected Sequence seq = new NodeSequence();
 protected Comparator comp;

// auxiliary methods
 protected Object key (Position pos) {
    return ((Item)pos.element()).key();
 } // note casting here

 protected Object element (Position pos) {
    return ((Item)pos.element()).element();
 } // casting here too

 // methods of the SimplePriorityQueue ADT
 public SequenceSimplePriorityQueue (Comparator c) {
    comp = c; }
 public int size () {return seq.size(); }
```

...suite à la page suivante...

Réalisation avec séquence triée (suite)

```
public void insertItem (Object k, Object e) throws
InvalidKeyException {
   if (!comp.isComparable(k)) {
      throw new InvalidKeyException("The key is not valid");
   }
   else {
      if (seq.isEmpty()) {
         //if the sequence is empty, this is the
         seq.insertFirst(new Item(k,e));//first item
      }
      else { //check if it fits right at the end
         if (comp.isGreaterThan(k,key(seq.last()))) {
            seq.insertAfter(seq.last(),new Item(k,e));
         }
         else {
            //we have to find the right place for k.
            Position curr = seq.first();
            while (comp.isGreaterThan(k,key(curr))) {
               curr = seq.after(curr);
            }
            seq.insertBefore(curr,new Item(k,e));
         }
      }
   }
```

...suite à la page suivante...

Réalisation avec séquence triée (suite)

```
public Object minElement () throws
EmptyContainerException {
if (seq.isEmpty()) {
      throw new EmptyContainerException("The priority
      queue is empty");
}
else {
      return element(seq.first());
}


public boolean isEmpty () {
      return seq.isEmpty();
}
}
```


Tri par sélection

- Le tri par sélection est une variation du tri par file à priorité (*PriorityQueueSort*) qui utilise une ***séquence non-triée*** pour réaliser la file à priorité *P*.
- **Phase 1**, l'insertion d'un item dans *P* est ***O***(1)
- **Phase 2**, le retrait d'un item de *P* prend un temps proportionnel au nombre d'éléments présents dans *P*

	Séquence *S*	File à priorité *P*
Entrée	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Tri par sélection (suite)

- Comme vous pouvez le constater, la phase 2 est le goulot d'étranglement. La première opération removeMin est ***O***(*n*), la seconde ***O***(*n*–1), et ainsi de suite jusqu'à la dernière, qui est ***O***(1).
- Le temps total nécessaire à la phase 2 est:

$$O(n + (n-1) + \ldots + 2 + 1) \equiv O\left(\sum_{i=1}^{n} i\right)$$

- Et comme:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- Le temps d'exécution de la phase 2 est donc ***O***(n^2). Ainsi, la complexité temporelle de l'algorithme est ***O***(n^2).

Tri par insertion

- Le tri par insertion résulte de l'utilisation d'un tri par file à priorité où la file est réalisée avec ***séquence triée***.

	Séquence *S*	File à priorité *P*
Entrée	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

Tri par insertion (suite)

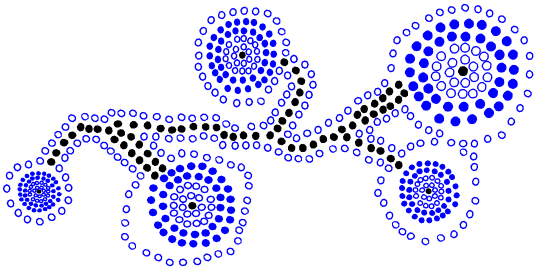
- Nous améliorons ainsi la phase 2, qui est $\boldsymbol{O}(n)$.
- Cependant, la phase 1 devient maintenant le goulot d'étranglement. Le premier insertItem est $\boldsymbol{O}(1)$, le second $\boldsymbol{O}(2)$, jusqu'au dernier qui lui est $\boldsymbol{O}(n)$, pour un temps d'exécution total $\boldsymbol{O}(n^2)$
- Le tri par sélection et le tri par insertion ont tous deux un temps d'exécution $\boldsymbol{O}(n^2)$
- Le tri par sélection va ***toujours*** exécuter un nombre d'opérations proportionnel à n^2, peu importe la séquence d'entrée
- Le temps d'exécution du tri par insertion varie selon la séquence d'entrée
- Aucune n'est une bonne méthode de tri, sauf pour les petites séquences
- Nous cherchons encore la file à priorité ultime...

Le tri

- Maintenant que vous avez une certaine connaissance du tri, parlons-en un peu plus à fond
- Le tri est essentiel parce qu'une ***recherche*** efficace dans une base de données ne peut être faite que si les enregistrements sont triés.
- Certains estiment qu'environ 20% du temps de calcul planétaire est dédié au tri
- Nous observerons qu'il existe un compromis entre *"simplicité"* et *efficacité* des algorithmes de tri:
- Les tris élémentaires vus jusqu'ici, qui étaient simples à comprendre et à réaliser, ont un temps d'exécution $\boldsymbol{O}(n^2)$ (inutilisables pour de grands n)
- Il existe des algorithmes plus sophistiqués $\boldsymbol{O}(n \log n)$
- Comparaison de clés: *comparons-nous la clé entière ou seulement une partie de la clé?*
- Espace requis: *tri à même la structure (**in-place**) versus l'utilisation de structures auxiliaires*
- Stabilité: *un algorithme de tri **stable** conserve l'ordre relatif des clés égales.*

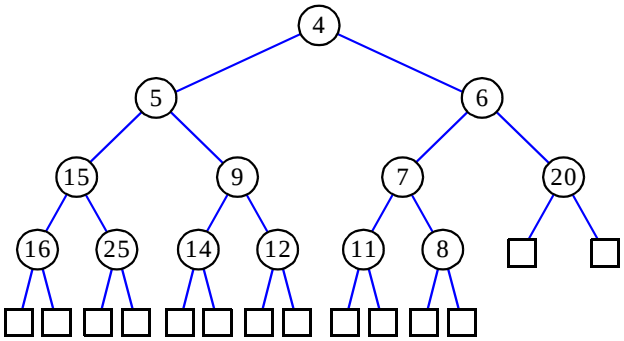
TAS

- Tas (*Heap*)
- Propriétés des tas
- Tri *Heap-Sort*
- Construction ascendante de tas (*Bottom-Up*)
- Repéreurs (*Locator Design Pattern*)



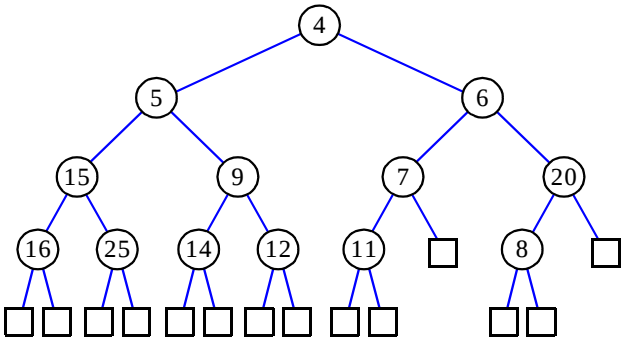
Tas

- Un ***tas*** (*heap*) est un arbre binaire *T* qui emmagasine une collection de clés (ou paires clé-élément) comme nœuds internes et qui satisfait aux deux propriétés suivantes:
  - ***Propriété d'ordre:*** clé(parent) ≤ clé(enfant)
  - ***Propriété structurelle***: tous les niveaux sont pleins, excepté le dernier, ce dernier étant cependant plein à gauche (***arbre binaire complet***)

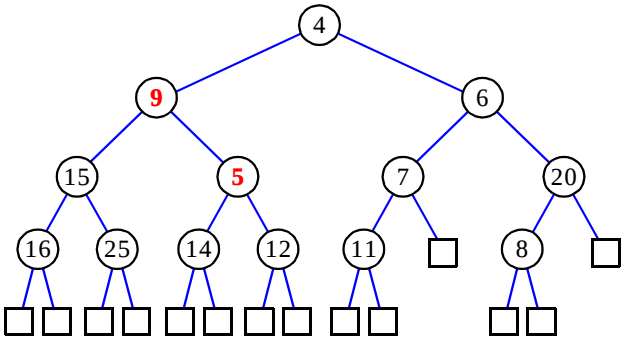


Exemples de non-tas

- le dernier niveau n'est pas plein à gauche



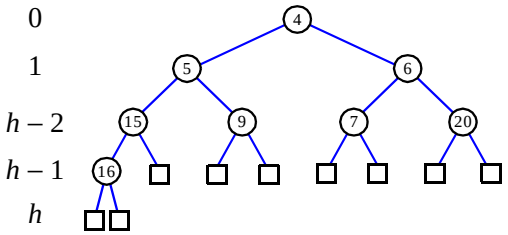
- clé(parent)> clé(enfant)



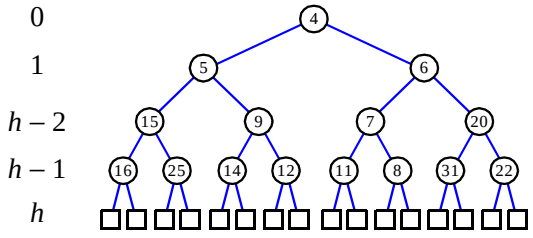
Hauteur d'un tas

Un tas *T* qui emmagasine *n* clés a une hauteur $h = \lceil \log(n + 1) \rceil$, qui est $O(\log n)$

- $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1 = 2^{h-1} - 1 + 1 = 2^{h-1}$



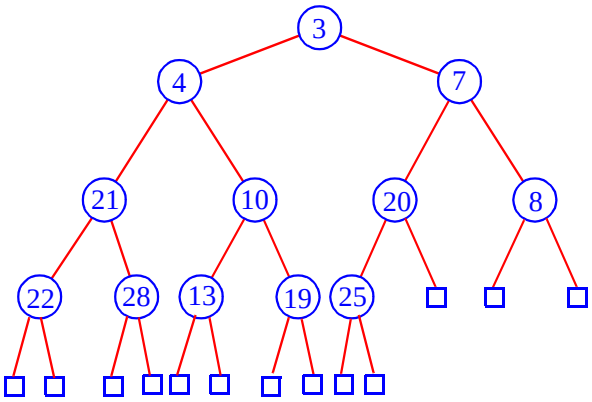
- $n \leq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$



- Ainsi $2^{h-1} \leq n \leq 2^h - 1$
- En calculant le logarithme, nous obtenons $\log (n + 1) \leq h \leq \log n + 1$, et donc $\boldsymbol{h = \lceil \log(n+1) \rceil}$

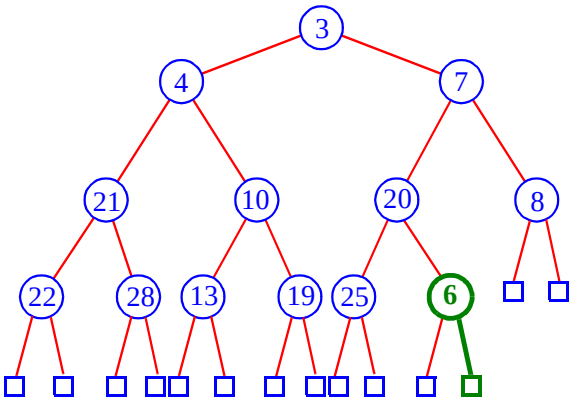
Insertion dans un tas

La clé à insérer est **6**



Insertion dans un tas (suite)

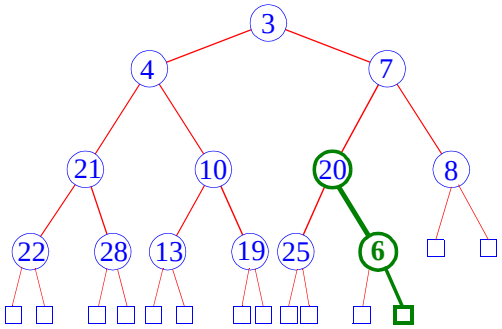
Ajoutez la clé à la ***prochaine position disponible*** dans le tas.

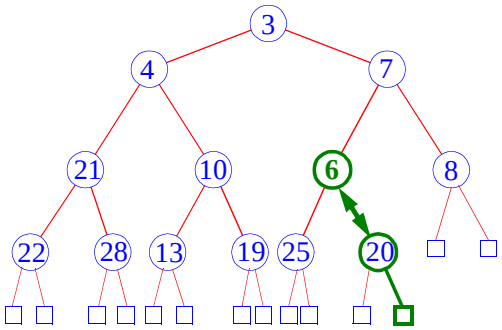


Commencez maintenant la procédure ***Upheap***.

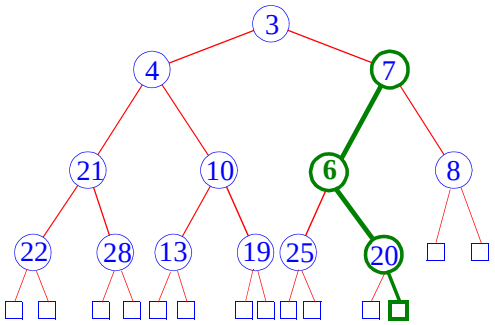
Procédure *Upheap*

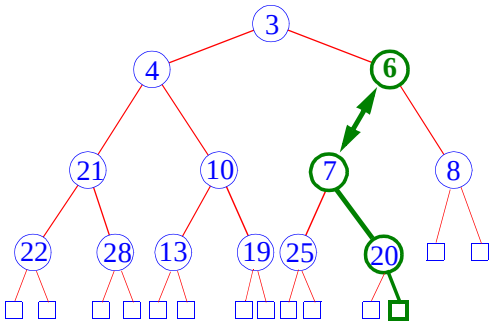
- Échangez (*swap*) les clés parent-enfant non-ordonnées



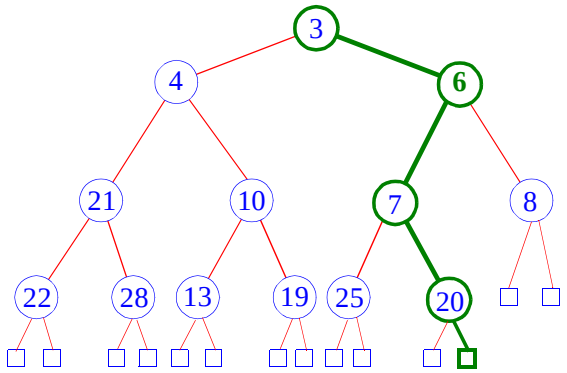


Suite de *Upheap*





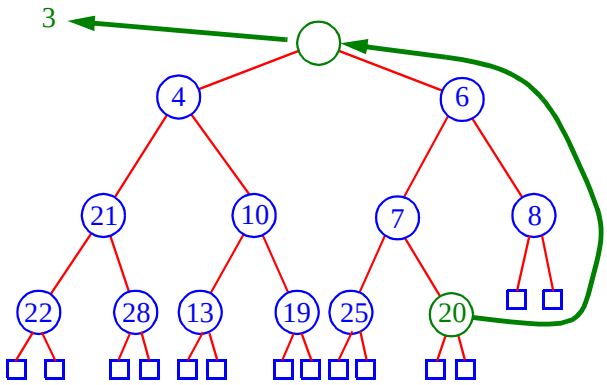
Fin de *Upheap*



- ***Upheap*** se termine quand la nouvelle clé est plus grande que la clé de son parent **ou** quand le haut du tas est atteint.
- (#échanges total) $\leq$ $(h-1)$, qui est $O(\log n)$

Suppression dans un tas

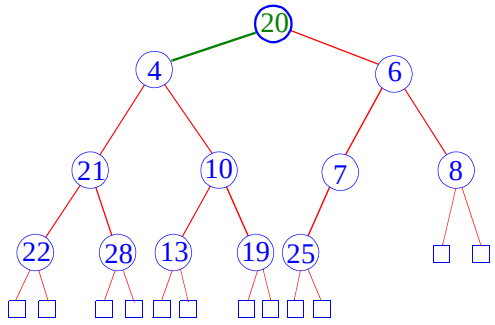
RemoveMin()

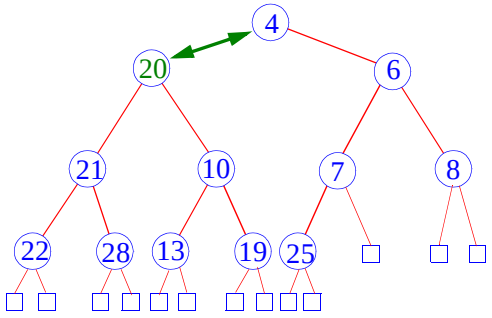


- La suppression de la clé racine laisse un trou
- Nous devons réparer le tas
- Premièrement, remplacez le trou par la toute dernière clé du tas
- Ensuite, appliquez la procédure ***Downheap***

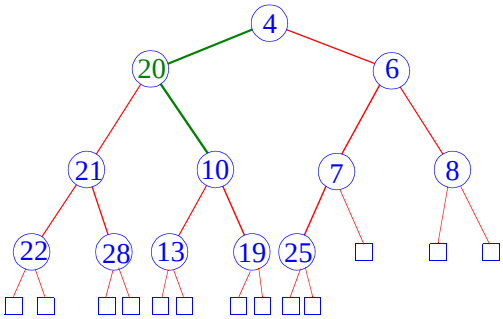
Procédure *Downheap*

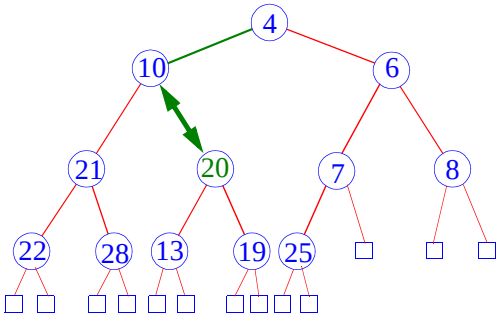
Downheap compare le parent avec son enfant le plus petit. Si cet enfant est plus petit que le parent, alors on les échange l'un pour l'autre.



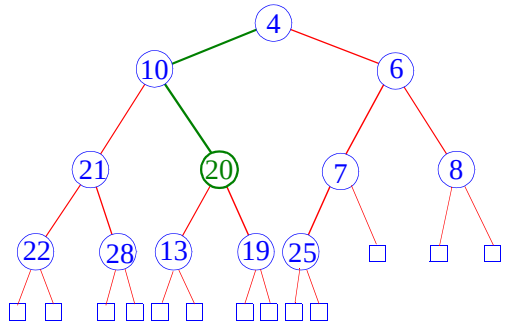


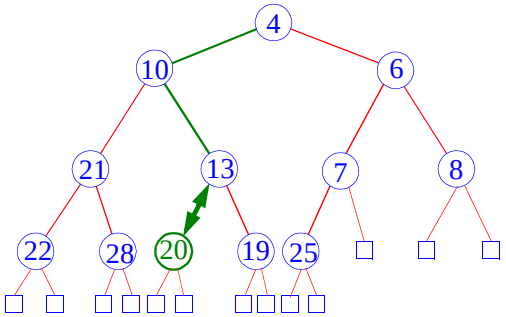
Suite de *Downheap*



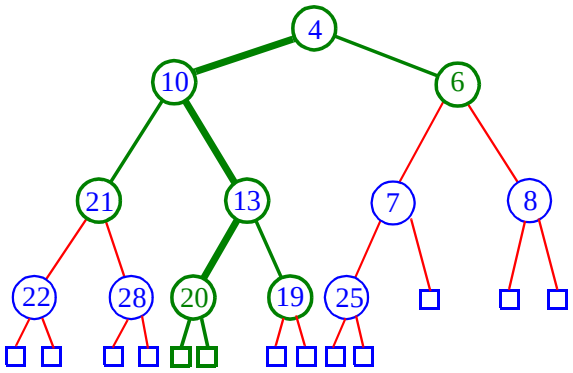


Suite de *Downheap* (2)





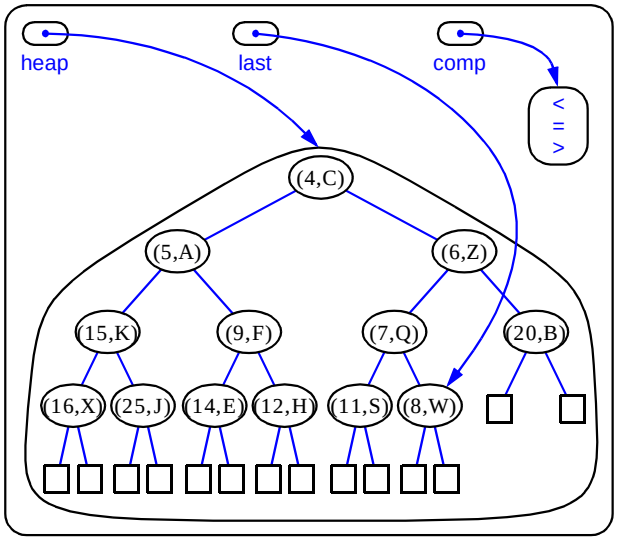
Fin de *Downheap*



- ***Downheap*** se termine quand la clé est plus grande que les clés de ses deux enfants **ou** quand le bas du tas est atteint.
- (#échanges total) $\leq$ $(h-1)$, qui est $O(\log n)$

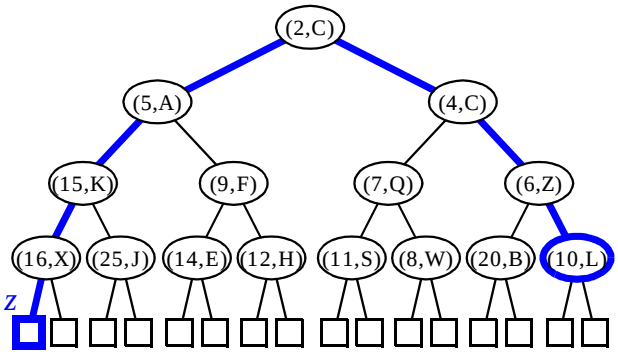
Réalisation d'un tas

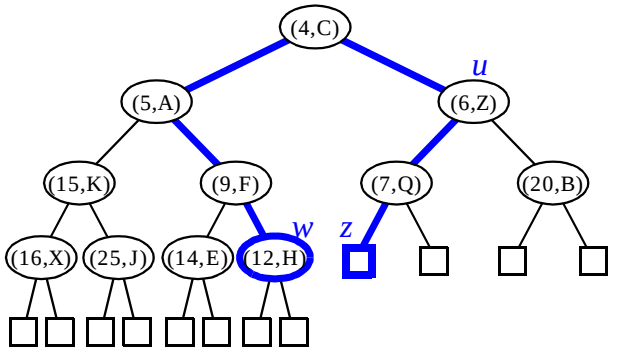
```
public class HeapPriorityQueue implements PriorityQueue
{
   BinaryTree T;
   Position last;
   Comparator comparator;
   ...
}
```



Réalisation d'un tas (suite)

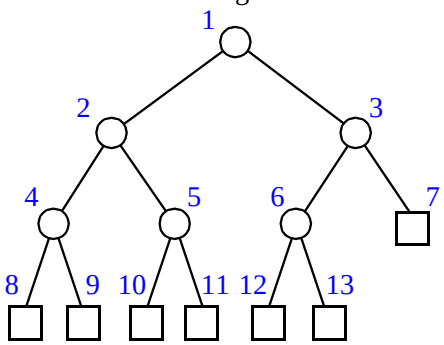
- Deux façons de trouver la position d'insertion z:





Réalisation par vecteur (*Vector*)

- Les mises à jour dans l'arbre sous-jacent ne surviennent seulement qu'au "dernier élément".
- Un tas peut être représenté par un vecteur (*vector*), où le nœud au rang i a:
  - l'enfant de gauche au rang $2i$ et
  - l'enfant de droite au rang $2i + 1$



- Les feuilles n'ont pas à être emmagasinées.
- L'insertion et la suppression de clés dans le tas correspondent respectivement à insertLast et à removeLast dans le vecteur.

Tri *Heap-Sort*

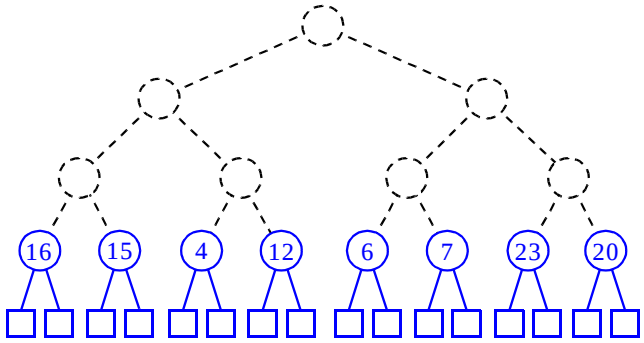
- Toutes les méthodes d'un tas s'exécutent en un temps logarithmique, ou mieux.
- Si nous réalisons le tri *PriorityQueueSort* avec un tas comme file à priorité, insertItem et removeMin prennent alors $O(\log k)$ chacun, où k est le nombre d'éléments dans le tas à un moment donné.
- Nous avons toujours au plus n éléments dans le tas, alors le pire des cas en terme de complexité pour ces méthodes est $O(\log n)$.
- Chaque phase prend donc $O(n \log n)$, et le temps d'exécution de l'algorithme est aussi de $O(n \log n)$.
- Ce tri est connu sous le nom de ***heap-sort***.
- Le temps d'exécution $O(n \log n)$ d'un tri *heap-sort* est bien meilleur que le temps d'exécution $O(n^2)$ d'un tri à bulle, par sélection, ou par insertion.

Tri Heap-Sort *in-place*

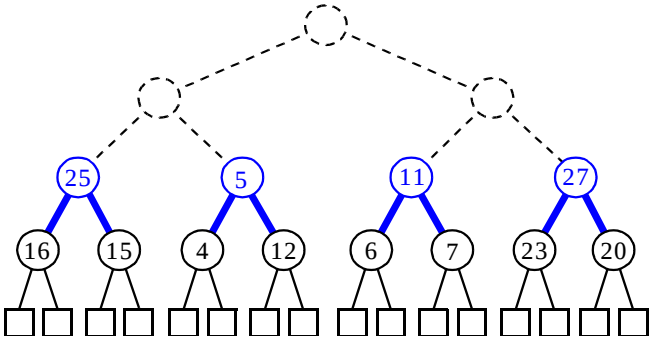
- N'utilise pas de tas (ou d'autre structure) externe.
- Utilise une représentation par vecteur pour contenir le tas. Construction ascendante (*bottom-up*)...

Construction ascendante du tas (1)

- construisez $(n + 1)/2$ tas à un seul élément (trivial)

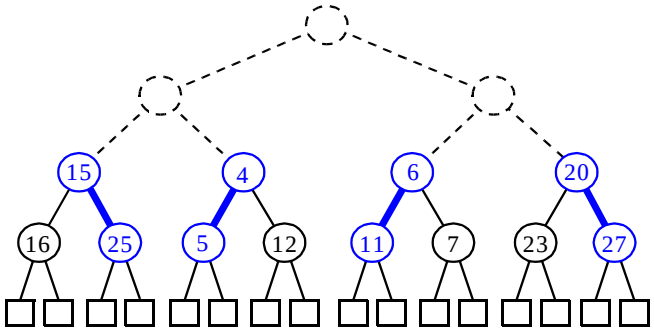


- construisez maintenant des tas à trois éléments

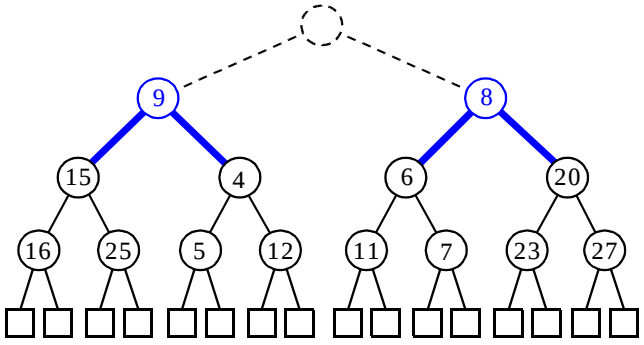


Construction ascendante du tas (2)

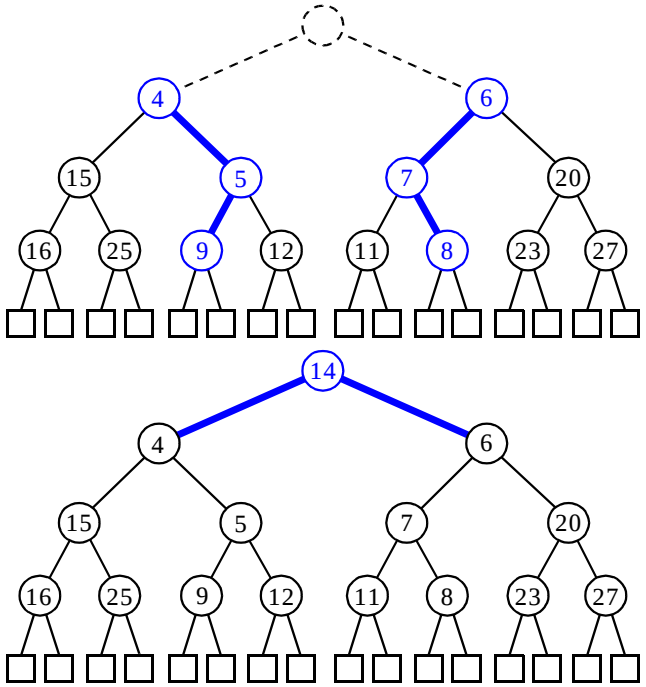
- préservez la propriété d'ordre avec ***downheap***



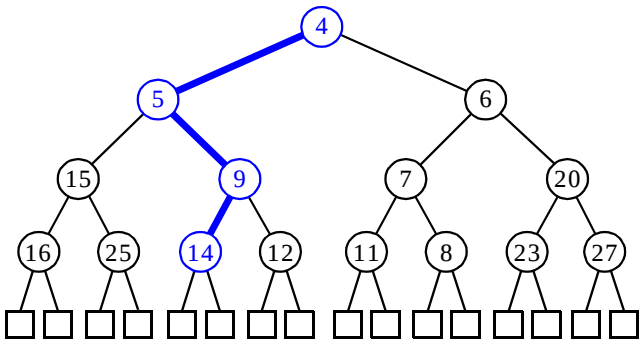
- formez maintenant des tas à 7 éléments



Construction ascendante du tas (3)



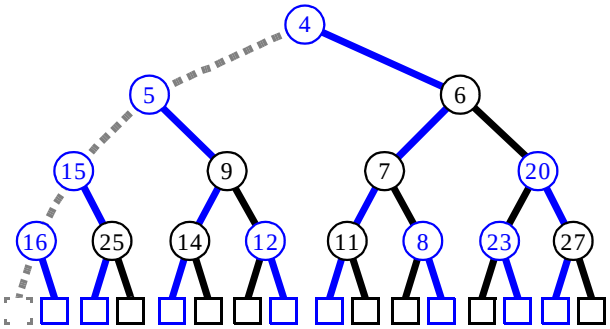
Construction ascendante du tas (4)



Fin!

Analyse de la construction ascendante de tas

- Proposition: la construction ascendante de tas avec n clés a un temps d'exécution $\boldsymbol{O}(n)$.
  - Insérer $(n + 1)/2$ nœuds
  - Insérer $(n + 1)/4$ nœuds et utiliser *downheap*
  - Insérer $(n + 1)/8$ nœuds et utiliser *downheap*
  - ...
  - analyse visuelle:



- n insertions, $n/2$ *downheap* pour un temps d'exécution total d'ordre $\boldsymbol{O}(n)$.

Repéreurs (*Locators*)

- Des repéreurs peuvent être utilisés pour suivre les éléments lorsqu'ils sont déplacés dans un contenant.
- Un repéreur (patron de conception *locator*) suit un élément spécifique, même si cet élément change de position dans son contenant.
- Le TAD locator contient les méthodes fondamentales suivantes:
  - element(): retourne l'élément de l'item associé au locator.
  - key(): retourne la clé de l'item associé au locator.
- À l'aide de repéreurs nous définissons des méthodes additionnelles pour le TAD file à priorité:
  - insert(k,e): insère (k,e) dans P et retourne son locator
  - min(): retourne le locator de l'élément à la plus petite clé
  - remove(l): supprime l'élément au locator l
- Dans notre application boursière, nous retournons un repéreur quand une commande est faite. Un repéreur permet de spécifier sans ambiguïté une commande lors d'une annulation.

Positions et Repéreurs

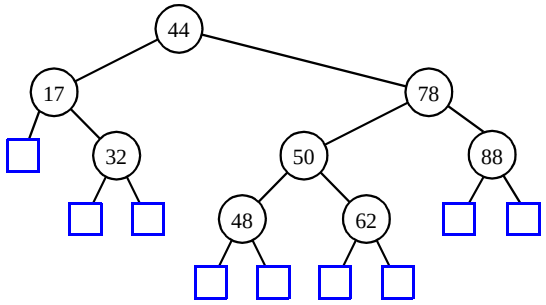
- Vous pourriez être en train de vous demander quelle est la différence entre repéreurs et positions, et pourquoi les distinguer.
- Il est vrai qu'ils ont des méthodes semblables.
- La différence se situe au niveau de leur utilisation primaire.
- Les positions font abstraction de la réalisation spécifique de l'accès aux éléments (indices ou nœuds).
- Les positions sont définies relativement l'une par rapport à l'autre (précédent/prochain, père/enfant).
- Les repéreurs surveillent où se situent les éléments. Dans la réalisation d'un TAD pour repéreurs, un repéreur conserve typiquement la position courante de l'élément.
- Les repéreurs associent les éléments avec leurs clés.

Positions et Repéreurs au travail

- Par exemple, considérez le Service de valet de stationnement CSI2514 (créé par les AE parce qu'ils avaient trop de temps libre).
- Lorsqu'ils ont démarré leur entreprise, André et Daniel décidèrent de créer une structure de données pour déterminer où les voitures sont situées.
- André suggère qu'une ***position*** représente ***l'espace de stationnement*** dans lequel la voiture se trouve.
- Cependant Daniel sait bien que les AE se promènent avec les voitures partout sur le campus et qu'elles ne seront pas toujours stationnées au même endroit.
- Alors ils décident d'installer un ***repéreur*** (un ***appareil sans fil***) dans chaque voiture. Chaque repéreur a un identifiant, qui est inscrit sur le coupon de retour.
- Quand un client demande sa voiture, l'AE active le repéreur, et alors la voiture klaxonne et ses lumières clignotent! Si la voiture est stationnée, André et Daniel sauront où la retrouver dans le stationnement, sinon, l'AE conduisant cette voiture saura qu'il est temps de la rapporter.

DICTIONNAIRES ET RECHERCHE

- Le TAD Dictionnaire
- Recherche binaire
- Arbres de recherche binaires

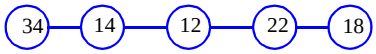


Le TAD Dictionnaire

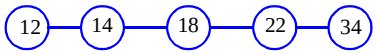
- un dictionnaire (*dictionary*) est un modèle abstrait de base de données.
- tel une file à priorité, un dictionnaire emmagasine des paires clé-élément
- la recherche par clé est la principale opération offerte par un dictionnaire
- méthodes simples de contenant:
  - size()
  - isEmpty()
  - elements()
- méthodes de requête:
  - findElement(*k*)
  - findAllElements(*k*)
- méthodes de mise à jour:
  - insertItem(*k*, *e*)
  - removeElement(*k*)
  - removeAllElements(*k*)
- élément spécial
  - NO_SUCH_KEY, retourné lors d'une recherche infructueuse.

Réalisation d'un dictionnaire à l'aide d'une séquence

- ***séquence non-ordonnée***



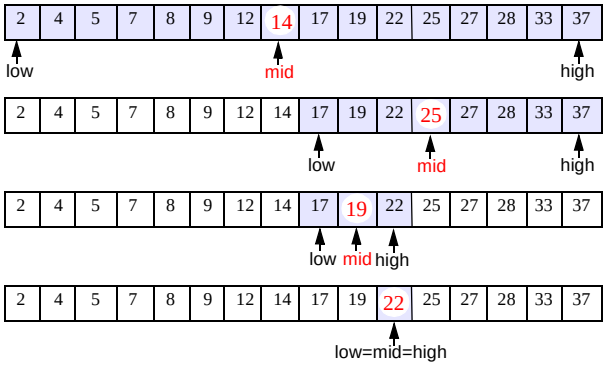
  - chercher et supprimer prennent un temps $O(n)$
  - insérer prend un temps $O(1)$
  - application aux registres et journaux (*logs*) (insertions fréquentes, recherches et suppressions plutôt rares)
- ***séquence ordonnée à base de vecteur*** (en supposant que les clés puissent être ordonnées)



  - chercher prend un temps $O(\log n)$ (***recherche binaire***)
  - insérer et supprimer prennent un temps $O(n)$
  - application aux tables de recherche (*look-up tables* — recherches fréquentes, insertions et suppressions plutôt rares)

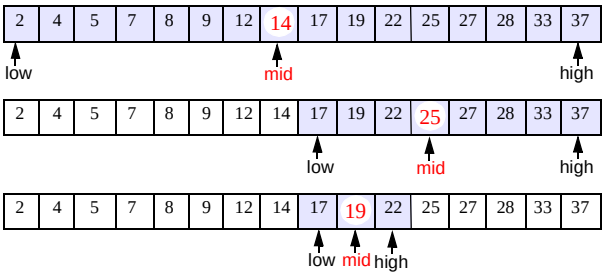
Recherche binaire

- restreindre l'intervalle de recherche par stages
- jeu "trop haut - trop bas" (*high-low*)
- findElement(22)



Pseudo-code pour recherche binaire

```
Algorithm BinarySearch(S, k, low, high)
if low > high then
  return NO_SUCH_KEY
else
  mid ← (low+high) / 2
  if k = key(mid) then
    return key(mid)
  else if k < key(mid) then
    return BinarySearch(S, k, low, mid−1)
  else
    return BinarySearch(S, k, mid+1, high)
```



Temps d'exécution de la recherche binaire

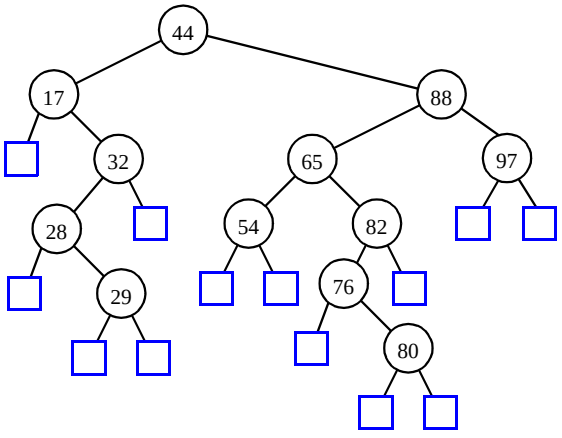
- L'intervalle des items qui seront considérés est ***réduit de moitié après chaque comparaison***

comparaison	intervalle de recherche
0	n
1	$n/2$
2	$n/4$
...	...
2^i	$n/2^i$
$\log_2 n$	1

- Dans la réalisation à base de vecteur, l'accès par rang prend un temps $O(1)$, et donc la recherche binaire s'exécute en un temps ***O*(log *n*)**

Arbres de recherche binaires

- Un arbre de recherche binaire est un arbre binaire T où:
  - chaque nœud interne v emmagasine un item de dictionnaire (k, e).
  - les clés se trouvant dans les nœuds du sous-arbre gauche de v sont plus petit ou égal à k.
  - les clés se trouvant dans les nœuds du sous-arbre droit de v sont plus grand ou égal à k.
  - les nœuds externes ne contiennent pas d'éléments.



Recherche

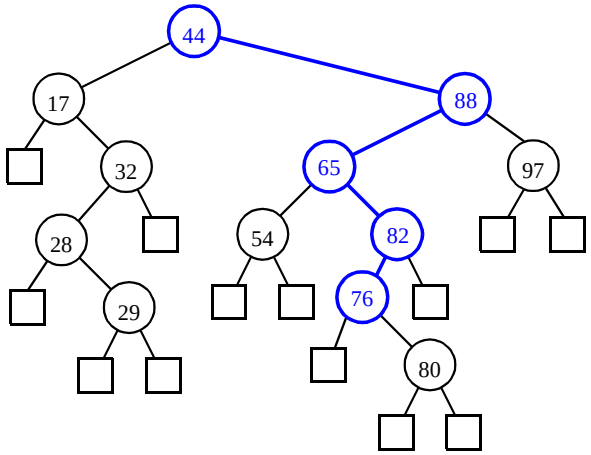
- Un arbre de recherche binaire T est un ***arbre de décision*** où la question posée à un nœud interne v se résume à: est-ce que la clé k est plus petite, égale, ou plus grande que la clé se trouvant dans v?
- Pseudo-code:

Algorithm TreeSearch(*k*, *v*):
Entrée: une clé de recherche k et un nœud v d'un arbre de recherche binaire T.
Sortie: un nœud w du sous-arbre $T(v)$ de T avec v comme racine, tel que w est un nœud interne emmagasinant k, ou w est un nœud externe visité lors de la traversée in-ordre de $T(v)$ après tous les nœuds internes aux clés plus petites que k et avant tous les nœuds internes aux clés plus grandes que k.

```
if v is an external node then
  return v
if k = key(v) then
  return v
else if k < key(v) then
  return TreeSearch(k, T.leftChild(v))
else
  { k > key(v) }
  return TreeSearch(k, T.rightChild(v))
```

Exemple de recherche I

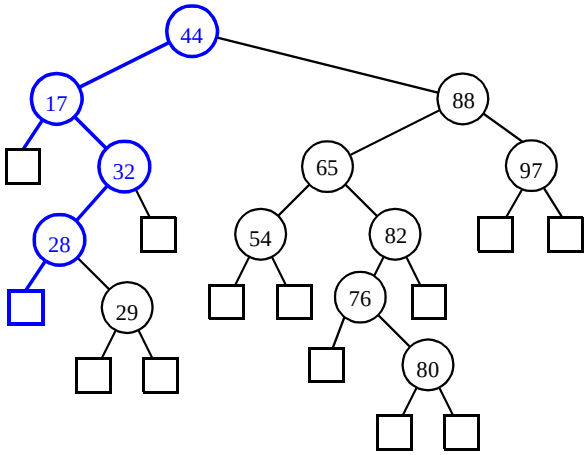
- findElement(76) réussi avec succès



- Une recherche fructueuse traverse un chemin débutant de la racine et se terminant à un nœud interne.
- Que dire de findAllelements(*k*)?

Exemple de recherche II

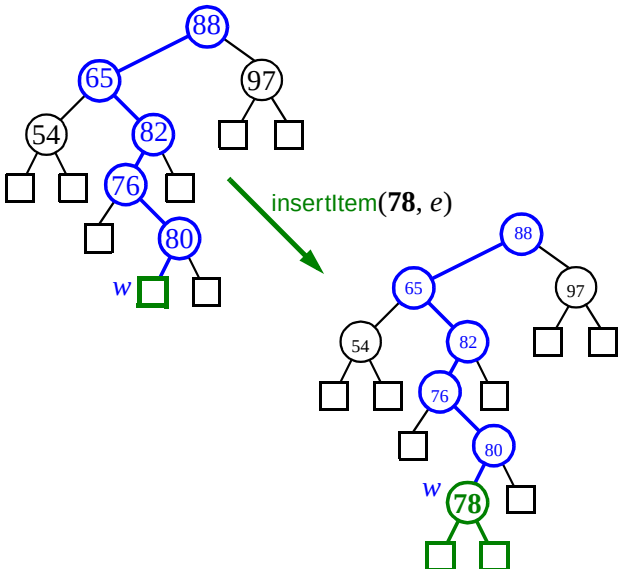
- findElement(25) qui ne réussit pas



- Une recherche infructueuse traverse un chemin débutant de la racine et se terminant à un nœud externe.

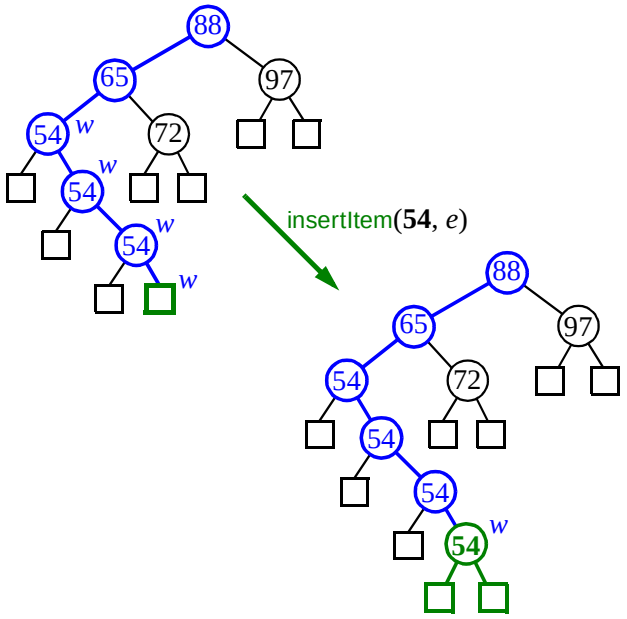
Insertion

- Pour exécuter insertItem(*k*, *e*), définissons *w* comme étant le nœud retourné par TreeSearch(*k*, *T*.root())
- Si *w* est externe, alors nous savons que *k* ne se trouve pas dans *T*. Nous appelons alors expandExternal(*w*) sur *T* et emmagasinons (*k*, *e*) dans *w*



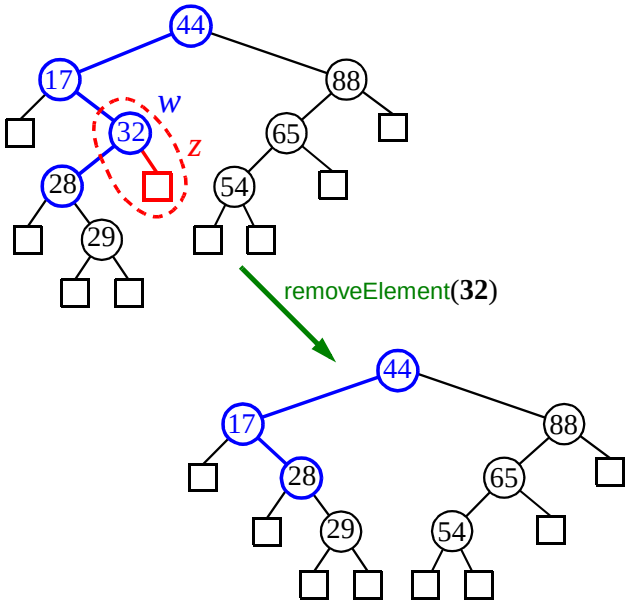
Insertion II

- Si *w* est interne, alors nous savons qu'un autre item avec une clé *k* se trouve à *w*. Nous appelons l'algorithme récursivement à partir de *T*.rightChild(*w*) ou de *T*.leftChild(*w*)



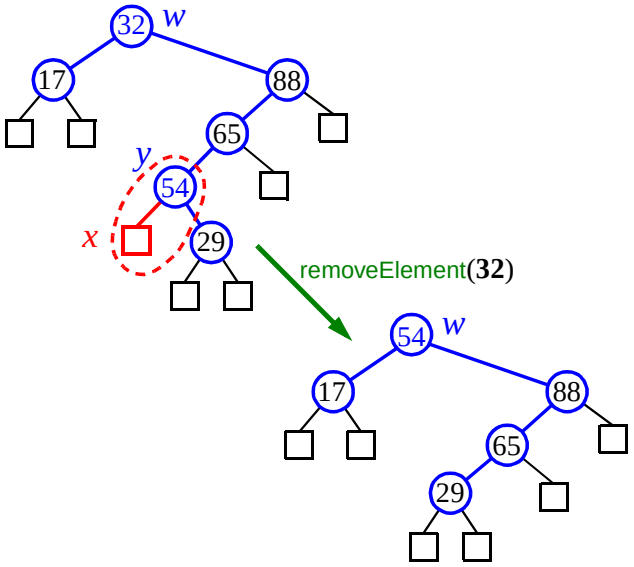
Suppression I

- Nous repérons le nœud *w* où la clé est emmagasinée avec l'algorithme TreeSearch
- Si *w* ***a un fils externe*** *z*, alors nous supprimons *w* et *z* avec removeAboveExternal(*z*)



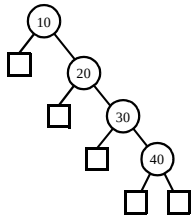
Suppression II

- Si *w* ***n'a pas de fils externe***:
  - trouvez le nœud interne *y* suivant *w* selon le parcours in-ordre
  - déplacez l'item de *y* vers *w*
  - exécutez removeAboveExternal(*x*), où *x* est le fils gauche de *y* (qui sera toujours externe)



Complexité temporelle

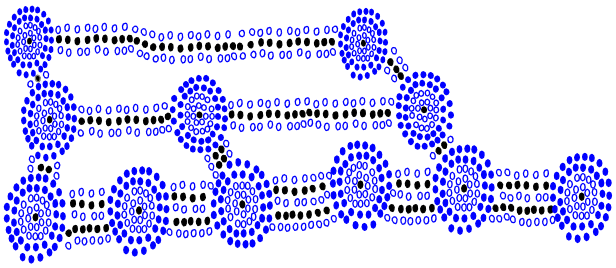
- Une recherche, une insertion, ou une suppression visite les nœuds de la **racine aux feuilles** (*root-to-leaf path*), et peut-être aussi les **frères** de ces nœuds
- Une durée $O(1)$ est nécessaire à chaque nœud
- Le temps d'exécution de chaque opération est $O(h)$, où h est la hauteur de l'arbre
- La hauteur d'un arbre de recherche binaire est n dans le pire des cas. Un tel arbre ressemble alors à une séquence triée



- Afin d'obtenir un bon temps d'exécution, nous devons garder l'arbre ***équilibré***, c'est-à-dire avec une hauteur de $O(\log n)$
- Différentes stratégies d'équilibrage seront explorées dans les prochains cours.

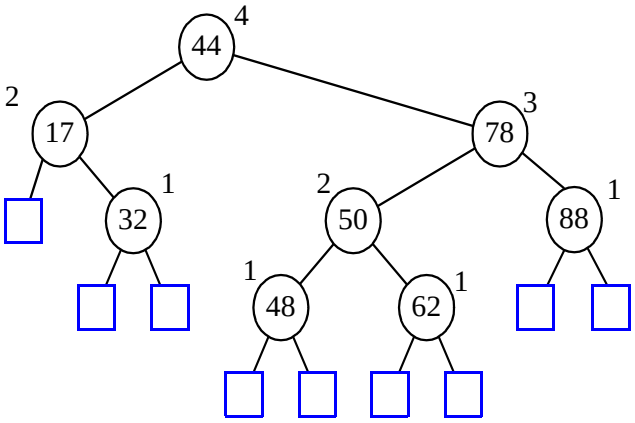
ARBRES AVL

- Arbres AVL



Arbre AVL

- **Les arbres AVL sont équilibrés.**
- Un arbre AVL est un arbre de recherche binaire où, pour tout nœud interne *v* de *T*, les hauteurs des enfants de *v* sont égales ou différentes de 1 niveau.
- Voici un exemple d'arbre AVL où les hauteurs sont indiquées près des nœuds:



Hauteur d'un arbre AVL

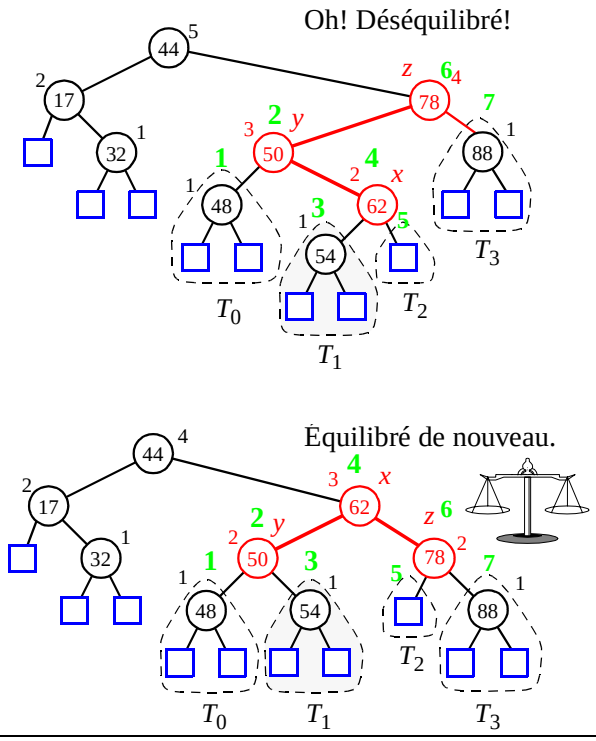
- **Proposition**: La hauteur d'un arbre AVL *T* emmagasinant *n* clés est ***O***(log *n*).
- **Justification**: l'approche la plus simple est d'essayer de trouver le nombre minimal de nœuds internes d'un arbre AVL de hauteur *h*: *n*(*h*).
- Nous observons que *n*(1) = 1 et *n*(2) = 2
- Pour $n \geq 3$, un arbre AVL de hauteur *h* avec *n*(*h*) contient au minimum le nœud racine, un sous-arbre AVL de hauteur *n*-1 et un autre de hauteur *n*-2.
- Ainsi *n*(*h*) = 1 + *n*(*h*-1) + *n*(*h*-2)
- Sachant que *n*(*h*-1) > *n*(*h*-2), nous obtenons *n*(*h*) > 2n(*h*-2)
  - *n*(*h*) > 2n(*h*-2)
  - *n*(*h*) > 4n(*h*-4)
  - ...
  - $n(h) > 2^i n(h-2i)$
- Résolution du cas de base: $n(h) \geq 2^{h/2-1}$
- Utilisation du logarithme: $h < 2\log n(h) + 2$
- Ainsi la hauteur d'un arbre AVL est ***O***(log *n*)

Insertion

- Un arbre de recherche binaire *T* est équilibré si, pour chaque nœud *v*, la hauteur des enfants de *v* sont égales ou différentes de 1 niveau.
- L'insertion d'un nœud dans un arbre AVL implique l'application de expandExternal(*w*) à *T*, qui change alors les hauteurs de quelques-uns des nœuds de *T*.
- Si une insertion fait que *T* devienne déséquilibré, alors nous traversons l'arbre vers le haut à partir du nœud nouvellement créé jusqu'à ce que nous trouvions le premier nœud *x* dont le grand-père *z* est un nœud déséquilibré.
- Puisque *z* est devenu déséquilibré par l'insertion dans le sous-arbre enraciné à son enfant *y*, hauteur(*y*) = hauteur(frère(*y*)) + 2
- Afin de rééquilibrer le sous-arbre enraciné à *z*, nous devons faire une *restructuration*
  - nous renommons *x*, *y*, et *z* par *a*, *b*, et *c* en se basant sur l'ordre des nœuds (traversée in-ordre)
  - *z* est remplacé par *b*, dont les enfants sont maintenant *a* et *c*. Les enfants de ces derniers sont les quatre autres sous-arbres qui étaient auparavant enfants de *x*, *y*, et *z*.

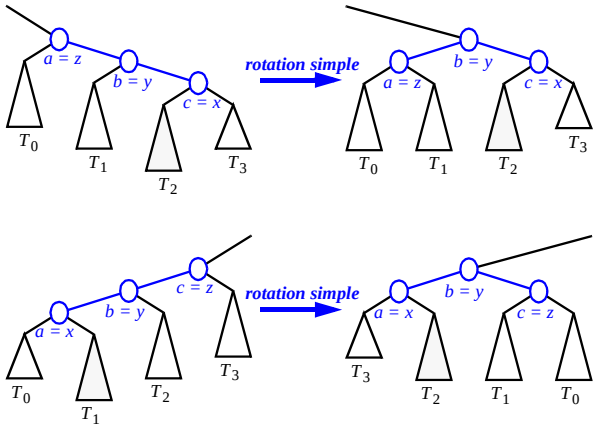
Insertion (suite)

- Exemple d'insertion dans un arbre AVL.



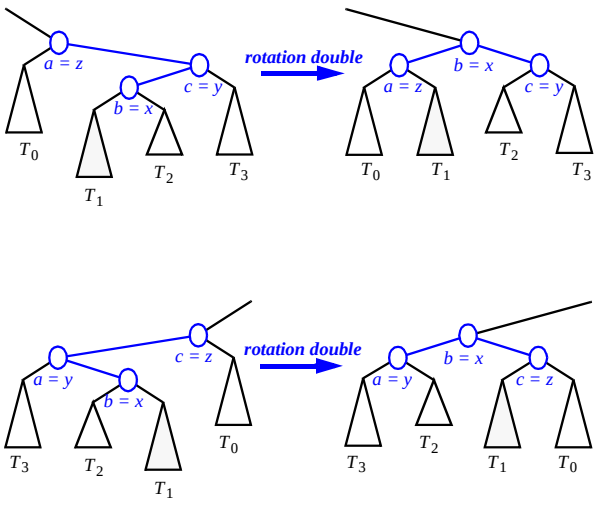
Restructuration

- Voici les quatre façons de faire la rotation des nœuds dans un arbre AVL, représentées graphiquement:
  - Rotations simples:



Restructuration (suite)

- Rotations double:



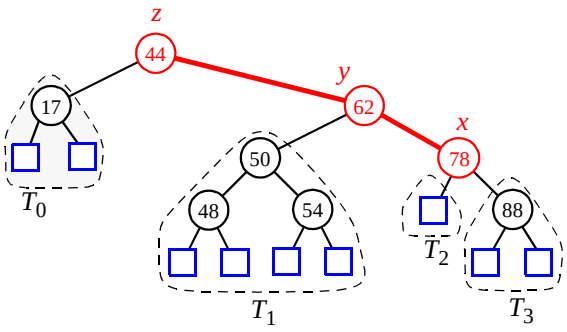
Algorithme de restructuration

Algorithm restructure(x):
Entrée: Un nœud x d'un arbre de recherche binaire T qui a y pour père et z pour grand-père
Sortie: L'arbre T restructuré par rotation (soit simple ou double) impliquant les nœuds x, y, et z.

1: Soit (a, b, c) une liste in-ordre des nœuds x, y, et z, et soit (T_0, T_1, T_2, T_3) une liste in-ordre des quatre sous-arbres de x, y, et z non-enraciné à x, y, ou z
2. Remplacez le sous-arbre enraciné à z par un nouveau sous-arbre enraciné à b
3. Placez a comme enfant de gauche de b et placez T_0 et T_1 comme sous-arbres de gauche et de droite de a, respectivement.
4. Placez c comme enfant de droite de b et placez T_2 et T_3 comme sous-arbres de gauche et de droite de c, respectivement.

Algorithme de restructuration Couper/Lier (*Cut/Link*)

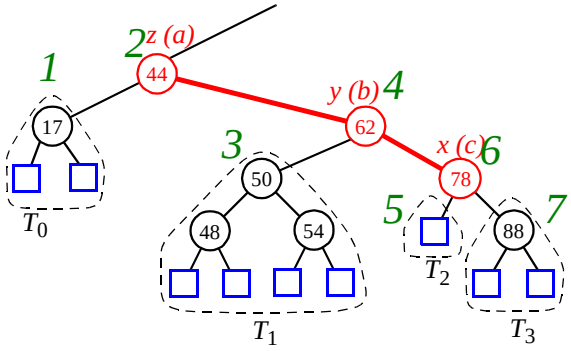
- Étudions cet algorithme de plus près...
- Tout arbre qui a besoin d'être restructuré peut être divisé en 7 parties: *x*, *y*, *z* et les 4 sous-arbres enracinés aux enfants de ces nœuds ($T_{0\text{-}3}$)



- Créez un nouvel arbre équilibré en déplaçant les 7 parties de l'arbre original de façon à ce que l'ordre soit le même lorsque nous faisons une traversée in-ordre du nouvel arbre.
- Ceci fonctionne peu importe la façon dont l'arbre original est déséquilibré. Observez...

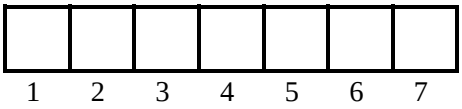
Algorithme de restructuration Couper/Lier (suite)

- Numérotez les 7 parties en parcourant l'arbre (in-ordre). Notez que *x*, *y*, et *z* sont maintenant renommés selon leur ordre dans la traversée.

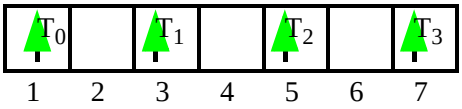


Algorithme de restructuration Couper/Lier (suite)

- Maintenant créez un vecteur, numéroté de 1 à 7 (l'élément 0 peut être ignoré avec une perte d'espace minimale)

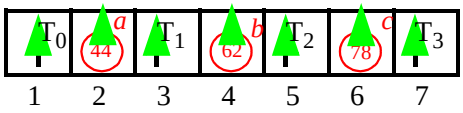


- Coupez les quatre arbres *T* et placez-les dans le vecteur selon leur rang in-ordre.

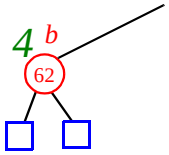


Algorithme de restructuration Couper/Lier (suite)

- Maintenant coupez *x*, *y*, et *z* dans cet ordre (fils, père, grand-père) and placez-les dans le vecteur selon leur rang in-ordre.

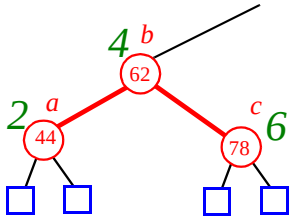


- Maintenant nous pouvons relier ces sous-arbres à l'arbre principal.
- Liez le rang 4 (*b*) comme étant la racine du sous-arbre original



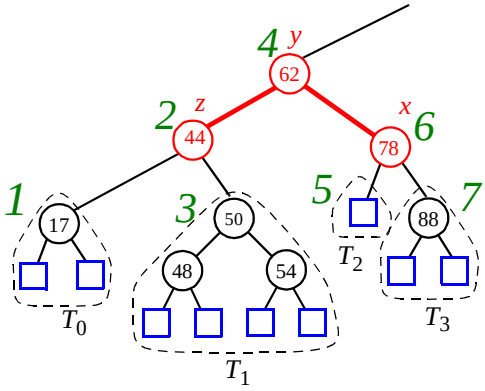
Algorithme de restructuration Couper/Lier (suite)

- Liez les rangs 2 (a) et 6 (c) comme enfants de 4.



Algorithme de restructuration Couper/Lier (suite)

- Finalement, liez les rangs 1, 3, 5 et 7 comme enfants de 2 et 6.



- Vous avez maintenant un arbre équilibré!

Algorithme de restructuration Couper/Lier (suite)

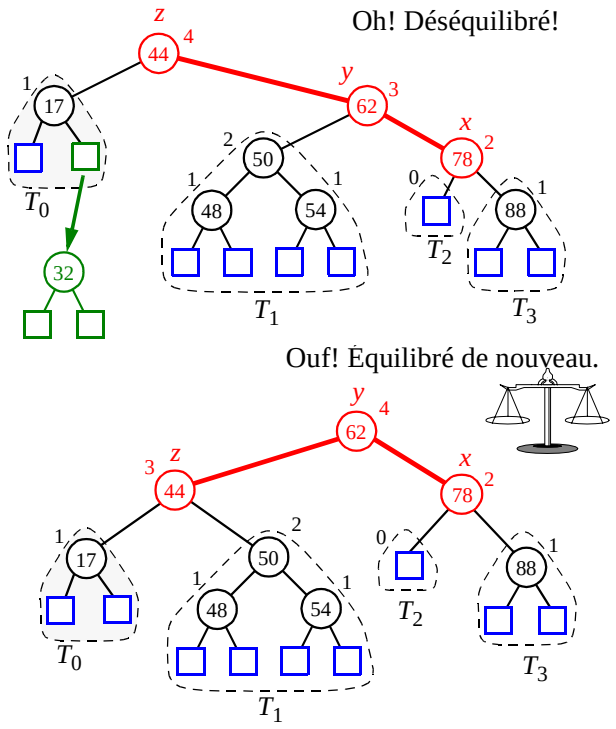
- Cet algorithme de restructuration a exactement le même effet que l'utilisation des quatre cas de rotation discutés plus tôt.
- Avantages: pas d'analyse de cas, plus élégant.
- Désavantage: peut exiger plus de code.
- Même complexité temporelle.

Suppression

- Nous pouvons voir facilement que l'application de removeAboveExternal(w) peut causer un déséquilibre dans T.
- Soit z le premier nœud déséquilibré rencontré en traversant l'arbre vers le haut à partir de w. Aussi, soit y l'enfant de z à la plus grande hauteur, et x l'enfant de y à la plus grande hauteur.
- Nous pouvons appliquer restructure(x) pour rééquilibrer le sous-arbre enraciné à z.
- Comme cette restructuration pourrait déséquilibrer un autre nœud plus haut dans l'arbre, nous devons continuer à vérifier l'équilibre jusqu'à ce que la racine de T soit atteinte.

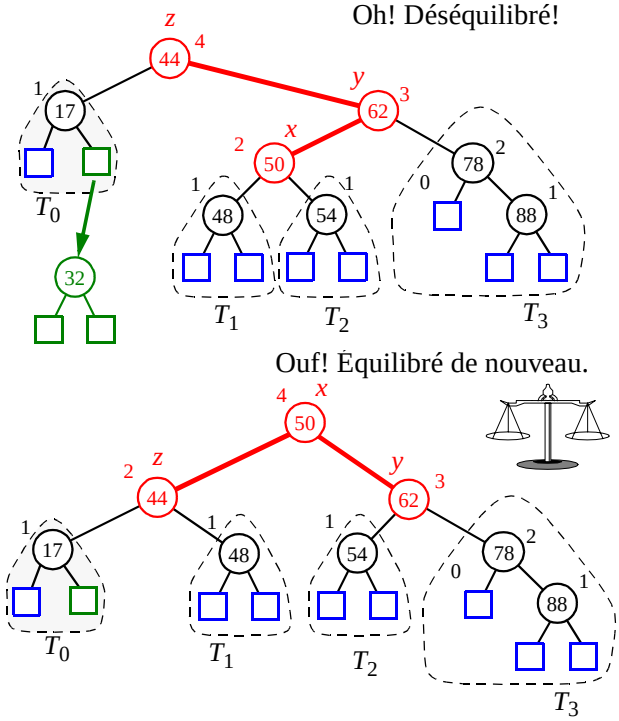
Suppression (suite)

- exemple de suppression dans un arbre AVL:



Suppression (suite)

- exemple de suppression dans un arbre AVL:



Réalisation

- La réalisation d'un arbre AVL en Java requiert la classe de nœud suivante:

```
public class AVLItem extends Item {
  int height;

  AVLItem(Object k, Object e, int h) {
    super(k, e);
    height = h;
  }

  public int height() {
    return height;
  }

  public int setHeight(int h) {
    int oldHeight = height;
    height = h;
    return oldHeight;
  }
}
```


Réalisation (suite)

```
public class SimpleAVLTree
  extends SimpleBinarySearchTree
  implements Dictionary {

public SimpleAVLTree(Comparator c) {
    super(c);
    T = new RestructurableNodeBinaryTree();
  }
  private int height(Position p) {
    if (T.isExternal(p))
      return 0;
    else
      return ((AVLItem) p.element()).height();
  }
  private void setHeight(Position p) { // called only
                                       // if p is internal
    ((AVLItem) p.element()).setHeight
      (1 + Math.max(height(T.leftChild(p)),
                    height(T.rightChild(p))));
  }
```

Réalisation (suite)

```
private boolean isBalanced(Position p)  {
      // test whether node p has balance factor
      // between -1 and 1
    int bf = height(T.leftChild(p)) - height(T.rightChild(p));
    return ((-1 <= bf) &&  (bf <= 1));
  }


private Position tallerChild(Position p)  {
       // return a child of p with height no
       // smaller than that of the other child
    if(height(T.leftChild(p)) >= height(T.rightChild(p)))
       return T.leftChild(p);
    else
       return T.rightChild(p);
  }
```


Réalisation (suite)

```
private void rebalance(Position zPos) {
//traverse the path of T from zPos to the root;
//for each node encountered recompute its
//height and perform a rotation if it is
//unbalanced
  while (!T.isRoot(zPos)) {
    zPos = T.parent(zPos);
    setHeight(zPos);
    if (!isBalanced(zPos)) { // perform a rotation
      Position xPos =  tallerChild(tallerChild(zPos));
      zPos = ((RestructurableNodeBinaryTree)
              T).restructure(xPos);
      setHeight(T.leftChild(zPos));
      setHeight(T.rightChild(zPos));
      setHeight(zPos);
    }
  }
}
```


Réalisation (suite)

```
public void insertItem(Object key, Object element)
    throws InvalidKeyException  {
  super.insertItem(key, element);// may throw an
                          // InvalidKeyException
  Position zPos = actionPos; // start at the
                          // insertion position
  T.replace(zPos, new AVLItem(key, element, 1));
  rebalance(zPos);
  }

public Object remove(Object key)
      throws InvalidKeyException {
    Object toReturn = super.remove(key);  // may throw
                        // an InvalidKeyException
    if (toReturn != NO_SUCH_KEY) {
      Position zPos = actionPos;  // start at the
                          // removal position
      rebalance(zPos);
    }
    return toReturn;
  }
}
```

Hachage (*Hashing*)

Qu'est-ce que c'est?

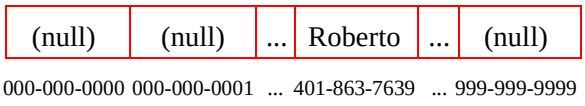
Une forme de narcotique?

Une forme de découpage?

Une combinaison des deux?

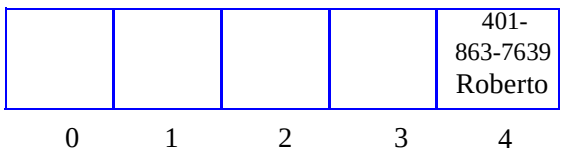
Problème

- RT&T est une grande compagnie téléphonique qui veut offrir un service d'identification de l'appelant:
  - étant donné un numéro de téléphone, retourne le nom de l'appelant
  - les numéros sont dans l'intervalle 0 à $R = 10^{10}-1$
  - n est le nombre de numéros utilisés
  - nous désirons une réalisation efficace
- Nous connaissons deux façons de concevoir ce dictionnaire:
  - un ***arbre de recherche équilibré*** (AVL, *red-black*) ou une ***liste "skip"*** avec le numéro de téléphone comme clé a un temps de requête $O(\log n)$ et un espace $O(n)$ — bon usage de l'espace mémoire et bon temps de recherche, mais peut-on réduire le temps de recherche à une constante?
  - un vecteur (***bucket array***) indexé par le numéro de téléphone a un temps de requête optimal $O(1)$, mais il y a un grand gaspillage d'espace: $O(n + R)$



Autre solution

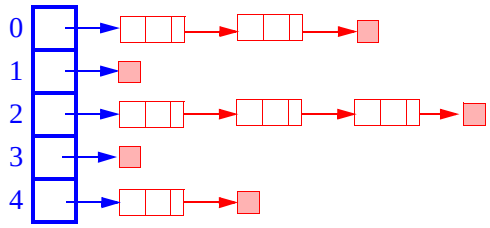
- Une ***table de hachage*** (*hash table*) est une solution alternative avec un temps de requête anticipé $O(1)$ et un espace $O(n + N)$, où N est la taille de la table.
- Comme un vecteur, mais avec une fonction projetant un grand ensemble de clés sur un plus petit.
  - ex.: prenez la clé originale ***modulo*** la taille de la table, et utilisez cette valeur comme index
- Insérez l'item (401-863-7639, Roberto) dans une table de taille 5
  - 4018637639 mod 5 = 4, alors l'item (401-863-7639, Roberto) est emmagasiné dans l'espace #4



- Une consultation (*lookup*) utilise le même processus: projection de la clé sur un index, et vérification de l'espace à cet index
- Insérez à la table (401-863-9350, André) et ensuite (401-863-2234, Daniel). Nous avons une ***collision***!

Résolution de collision

- Comment gérer deux clés qui sont projetée sur le même espace d'un vecteur?
- Utilisez le ***chaînage*** *(chaining)*
  - Créez des ***listes*** d'items avec le même index



- Le temps anticipé de recherche/insertion/suppression est $O(n/N)$, en supposant que les index soient distribués uniformément.
- La performance de la structure de données peut être affinée en changeant la taille de la table N

De clé à index

- La projection des clés vers les index de la table de hachage est appelée ***fonction de hachage***
- Une fonction de hachage est habituellement composée de deux parties:
  - ***code de hachage***: clé → integer
  - ***compression***: integer → $[0, N-1]$
- La fonction de hachage doit absolument projeter deux clés égales vers deux index égaux.
- Une "bonne" fonction de hachage minimise la probabilité de collision.
- Java offre la méthode hashCode() pour la classe Object, qui retourne typiquement l'adresse-mémoire (32 bits) de l'objet.
- Ce code de hachage par défaut ne serait pas très performant pour les objets Integer et String.
- La méthode hashCode() devrait être redéfinie de façon adéquate par les classes.

Codes de hachage populaires

- ***Mettre entier*** *(integer cast)*: pour les types numériques avec 32 bits ou moins, nous pouvons réinterpréter les bits du nombre comme un int.
- ***Somme des composantes*** *(component sum)*: pour les types numériques avec plus de 32 bits (ex.: long et double), nous pouvons additionner les composantes de 32 bits.
- ***Accumulation polynomiale***: pour les chaînes de caractères en langage naturel, combinez les valeurs de chaque caractère (ASCII, ISO Latin ou Unicode) $a_0 a_1 \ldots a_{n-1}$ en les considérant comme coefficients d'un polynôme: $a_0 + a_1 x + \ldots + x^{n-1} a_{n-1}$
  - Le polynôme est calculé avec la ***règle de Horner***, en ignorant les dépassements de capacité, avec une valeur fixe pour x:

    $$a_0 + x\,(a_1 + x\,(a_2 + \ldots x\,(a_{n-2} + x\,a_{n-1}) \ldots))$$
  - Le choix $x = 33, 37, 39,$ ou 41 donne au plus 6 collisions sur un vocabulaire de 50,000 mots anglais!
- Pourquoi la somme des composantes n'est-elle pas bonne pour les chaînes de caractères?

Méthodes de compression populaires

- ***Division***: $h(k) = |k| \bmod N$
  - $N = 2^k$ est un mauvais choix parce que ce ne sont pas tous les bits qui sont pris en compte
  - La taille de la table N est habituellement un nombre premier
  - certains patrons (*patterns*) dans le code de hachage sont propagés
- ***Multipliez, additionnez, et divisez***: $h(k) = |ak + b| \bmod N$
  - élimine les patrons lorsque $a \bmod N \neq 0$
  - même formule utilisée dans les générateurs de (pseudo) nombres aléatoires linéaires congruents

Encore des collisions

- Une clé est projeté sur un espace de la table qui est déjà occupé
  - que faire?!?
- Utilisez une technique de gestion des collisions
- Nous avons vu le ***chaînage***
- Nous pouvons aussi utiliser ***l'adressage ouvert***
  - ***Hachage double***
  - ***Sondage linéaire*** *(Linear Probing)*

Sondage linéaire

- Si l'espace courant est occupé, essayez l'espace suivant

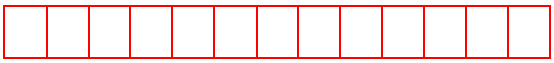
```
linear_probing_insert(K)
  if (table is full) error

  probe = h(K)

  while (table[probe] occupied)
       probe = (probe + 1) mod M

  table[probe] = K
```

- Une consultation parcours la table jusqu'à ce que la clé ou un espace vide soit trouvé.
- Utilise moins de mémoire que le chaînage
  - pas besoin d'emmagasiner tous ces liens
- Plus lent que le chaînage
  - peut résulter en un long parcours de la table
- La suppression est plus complexe
  - marquage de l'espace effacé, ou
  - remplir l'espace en déplaçant quelques éléments

Exemple de sondage linéaire

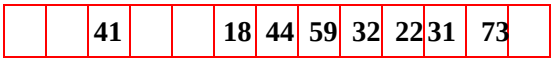
- $h(k) = k \bmod 13$
- Insérez les clés:

18 41 22 44 59 32 31 73



0 1 2 3 4 5 6 7 8 9 10 11 12

Exemple de sondage linéaire (suite)



0 1 2 3 4 5 6 7 8 9 10 11 12

Hachage double

- Utilise deux fonctions de hachage
- Si M est premier, éventuellement tous les espaces de la table seront examinés

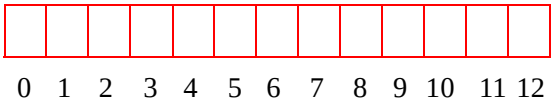
```
double_hash_insert(K)
  if (table is full) error

  probe = h1(K)
  offset = h2(K)

  while (table[probe] occupied)
     probe = (probe + offset) mod M

  table[probe] = K
```

- Plusieurs avantages et désavantages semblables à ceux du sondage linéaire
- Distribue les clés plus uniformément que le sondage linéaire

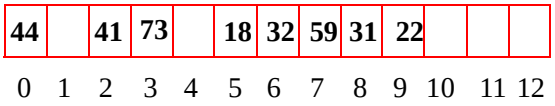
Exemple de hachage double

- h1(K) = K mod 13
  h2(K) = 8 - K mod 8
  - nous voulons que h2 soit un déplacement à ajouter

18 41 22 44 59 32 31 73



Exemple de hachage double (suite)



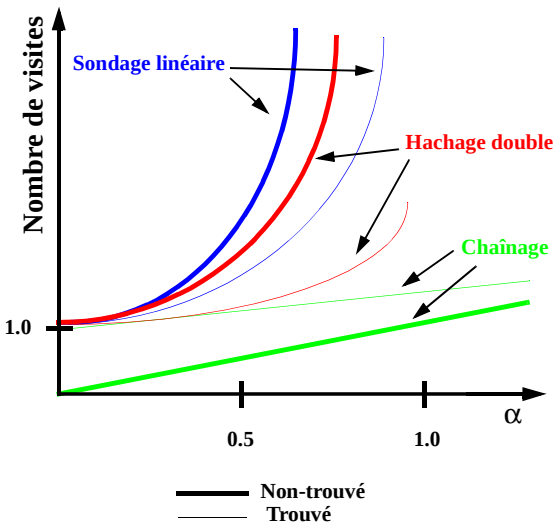
Résultats théoriques

- Soit $\alpha = N/M$
  - le facteur de charge: nombre moyen de clés par index du vecteur
- L'analyse utilise les probabilités plutôt que le pire des cas

Nombre de visites anticipé

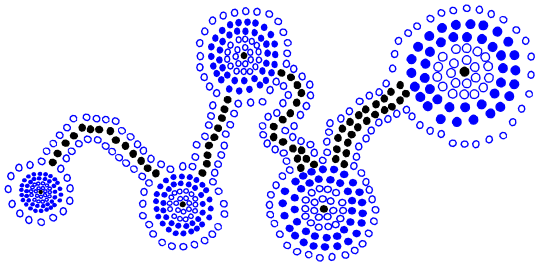
	non trouvé	*trouvé*
Chaînage	$1 + \alpha$	$1 + \frac{\alpha}{2}$
Sondage linéaire	$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$	$\frac{1}{2} + \frac{1}{2(1-\alpha)}$
Hachage double	$\frac{1}{(1-\alpha)}$	$\frac{1}{\alpha} ln \frac{1}{1-\alpha}$

Nombre de visites anticipé / Facteur de charge



TRI AVANCÉ

- Révision sur le tri
- Tri par fusion (*Merge Sort*)
- Ensembles (*Sets*)
- Tri rapide (*Quick-Sort*)
- À quelle vitesse peut-on trier?



Algorithmes de tri

- Le tri par sélection utilise une file à priorité P réalisée à l'aide d'une séquence non-ordonnée:
  - **Phase 1**: l'insertion d'un item dans P prend un temps $\boldsymbol{O}(1)$; en tout $\boldsymbol{O}(n)$
  - **Phase 2**: le retrait d'un item requiert un temps proportionnel au nombre d'éléments dans P, c'est-à-dire $\boldsymbol{O}(n)$; en tout $\boldsymbol{O}(n^2)$
  - Complexité temporelle: $\boldsymbol{O}(n^2)$

Algorithmes de tri (suite)

- Le tri par insertion utilise une file à priorité P réalisée à l'aide d'une séquence ordonnée:
  - **Phase 1**: le premier insertItem prend $\boldsymbol{O}(1)$, le second $\boldsymbol{O}(2)$, jusqu'au dernier insertItem qui prend $\boldsymbol{O}(n)$; en tout $\boldsymbol{O}(n^2)$
  - **Phase 2**: le retrait d'un item prend un temps $\boldsymbol{O}(1)$; en tout $\boldsymbol{O}(n)$.
  - Complexité temporelle: $\boldsymbol{O}(n^2)$
- Le tri *Heap Sort* utilise une file à priorité K réalisée à l'aide d'un tas.
  - insertItem et removeMin prennent chacun $\boldsymbol{O}(\log k)$, où k est le nombre d'éléments du tas à un moment donné.
  - **Phase 1**: n éléments insérés: temps $\boldsymbol{O}(n \log n)$
  - **Phase 2**: n éléments retirés: temps $\boldsymbol{O}(n \log n)$
  - Complexité temporelle: $\boldsymbol{O}(n \log n)$

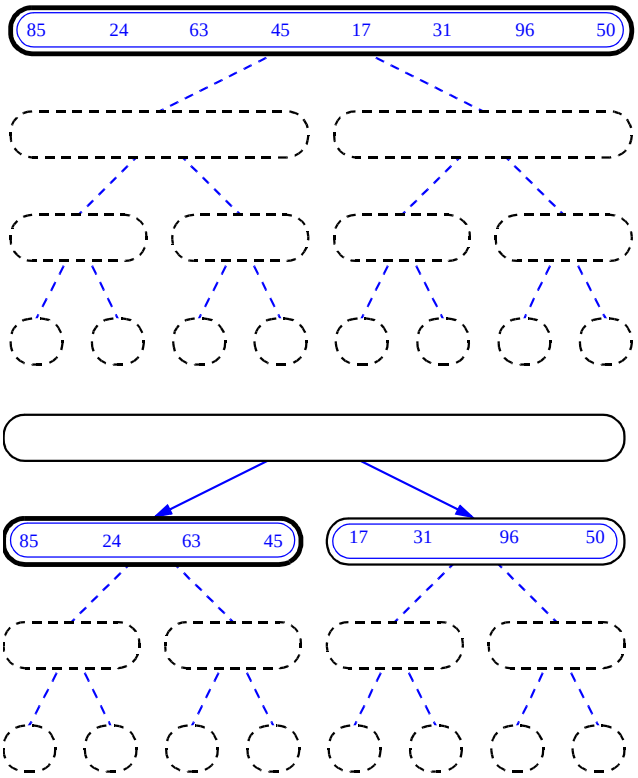
Diviser pour régner (*Divide-and-Conquer*)

- *Diviser pour régner* est bien plus qu'une stratégie militaire; il s'agit aussi d'une méthode de conception d'algorithmes qui a mené à la création d'algorithmes efficaces tel le tri par fusion.
- En termes d'algorithmes, cette méthode a trois étapes distinctes:
  - **Diviser**: Si la taille de l'entrée est trop grande pour la traiter de façon directe, alors divisez les données en deux ou plusieurs sous-ensembles disjoints.
  - **Appliquer récursivement**: Utilisez l'approche diviser pour régner afin de résoudre les sous-problèmes associés aux sous-ensembles de données.
  - **Conquérir**: Prenez les solutions aux sous-problèmes et "fusionnez" ces solutions afin d'obtenir la solution au problème initial.

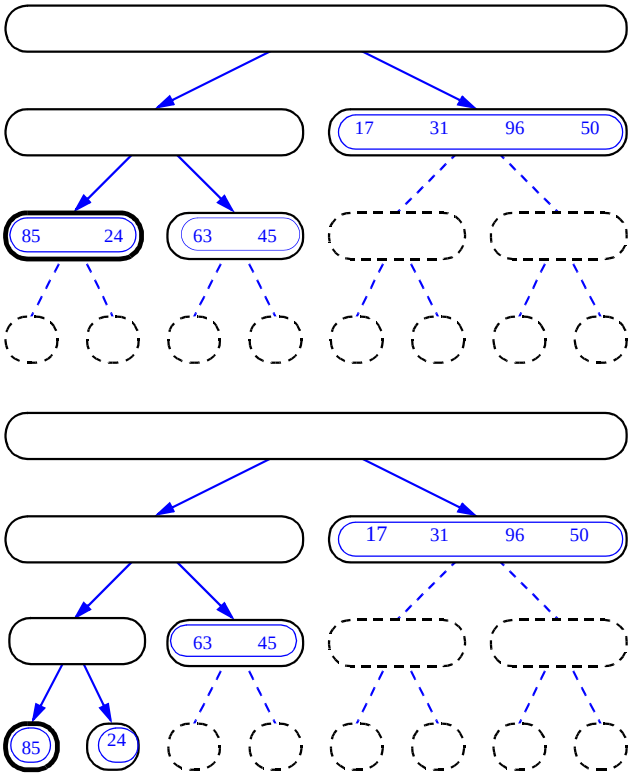
Tri par fusion (*Merge Sort*)

- Algorithme:
  - **Diviser**: Si S a au moins deux éléments (il n'y a rien à faire si S a zéro ou un élément), retirez tous les éléments de S et placez-les dans 2 séquences, S_1 et S_2, chacune contenant environ la moitié des éléments de S (S_1 contient les premiers $\lceil n/2 \rceil$ éléments et S_2 contient les $\lfloor n/2 \rfloor$ éléments restants)
  - **Appliquer récursivement**: Triez récursivement les séquences S_1 et S_2.
  - **Conquérir**: Replacez les éléments dans S en fusionnant les séquences triées S_1 et S_2 en une séquence triée unique.
- Arbre de tri par fusion:
  - Prenez un arbre binaire T
  - Chaque nœud T représente un appel récursif à l'algorithme de tri par fusion.
  - Nous associons à chaque nœud v de T l'ensemble des entrées à l'invocation que v représente.
  - Les nœuds externes sont associés aux éléments individuels de S, sur lesquels il n'y a pas d'appel récursif.

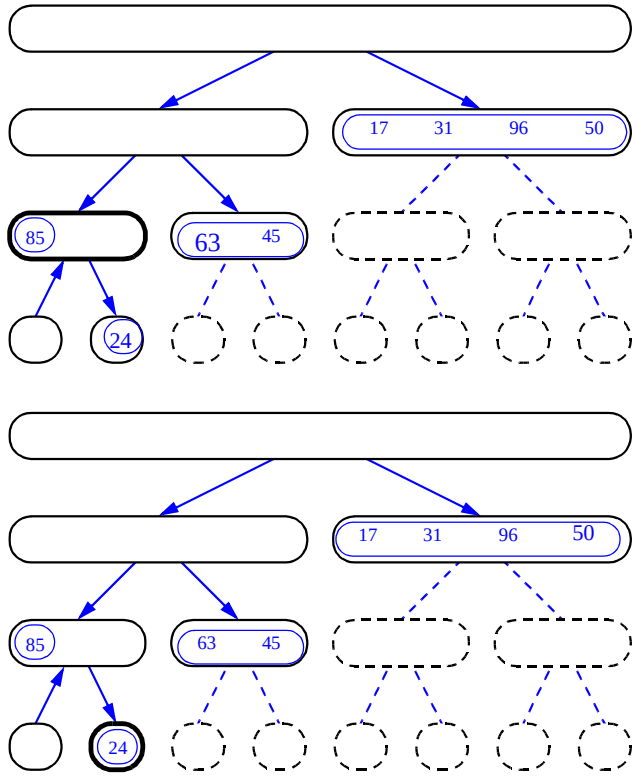
Tri par fusion



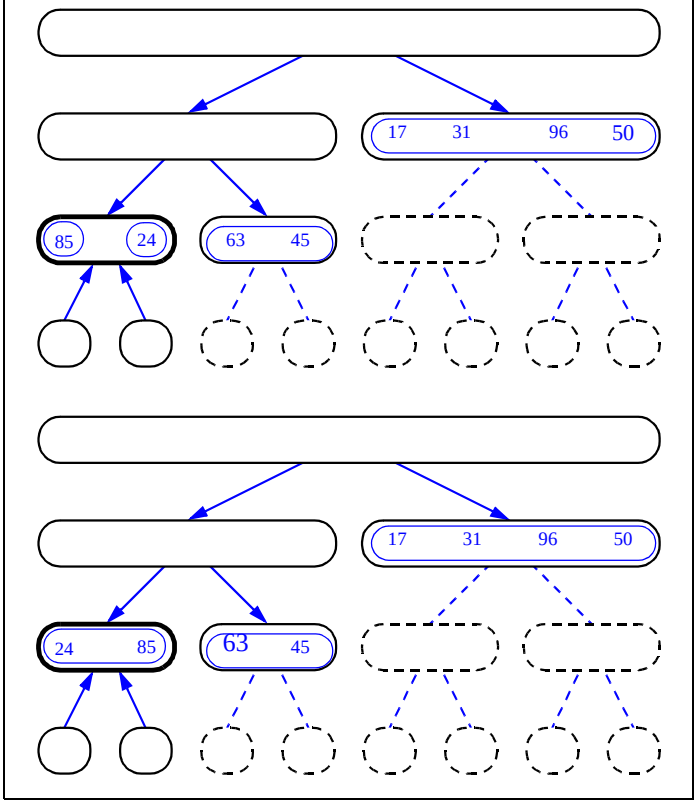
Tri par fusion (suite)



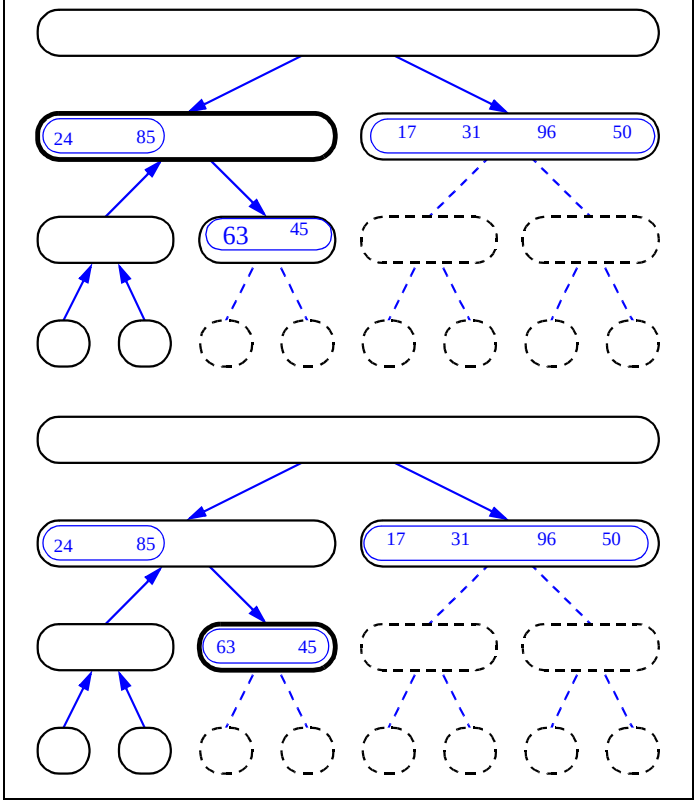
Tri par fusion (suite)



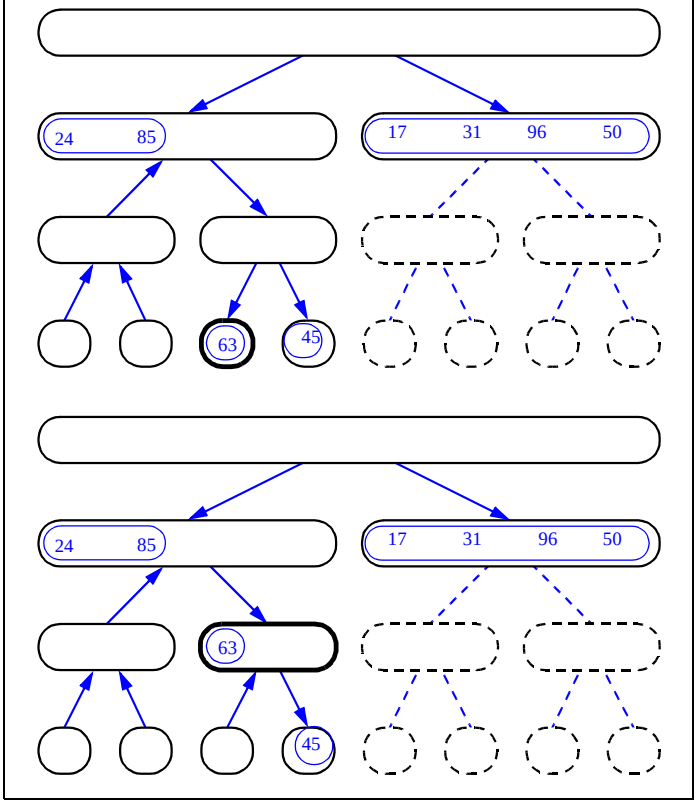
Tri par fusion (suite)



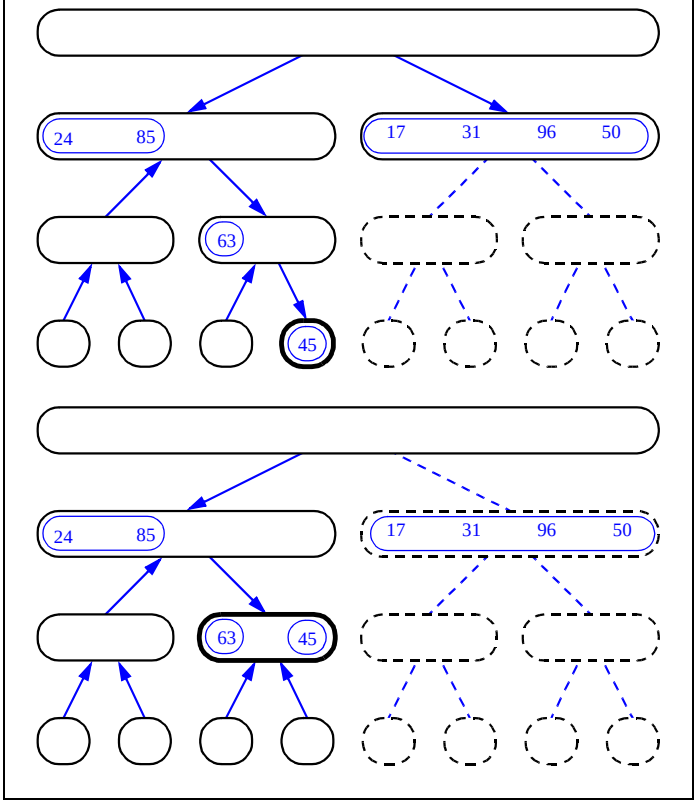
Tri par fusion (suite)



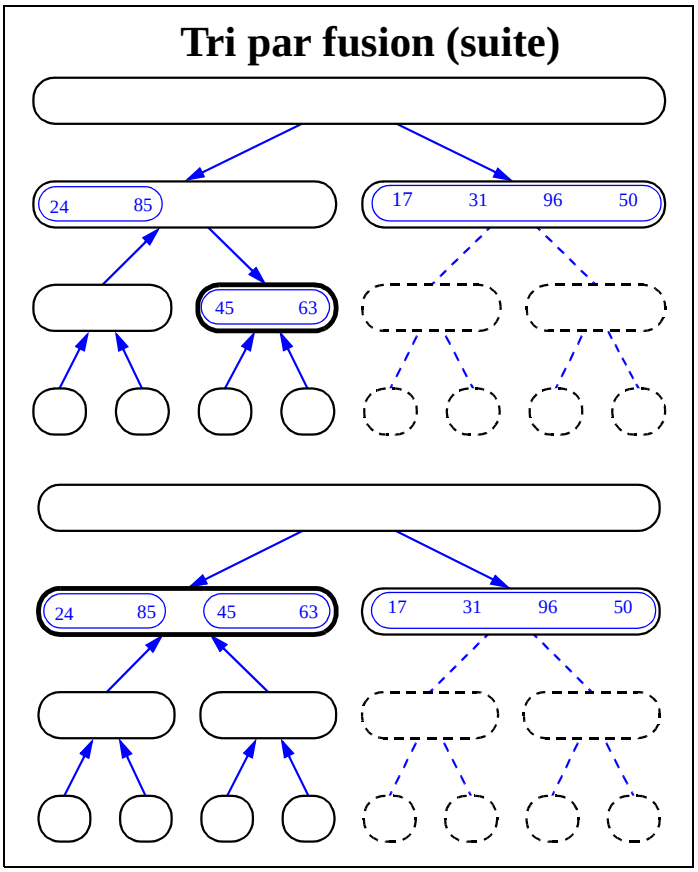
Tri par fusion (suite)



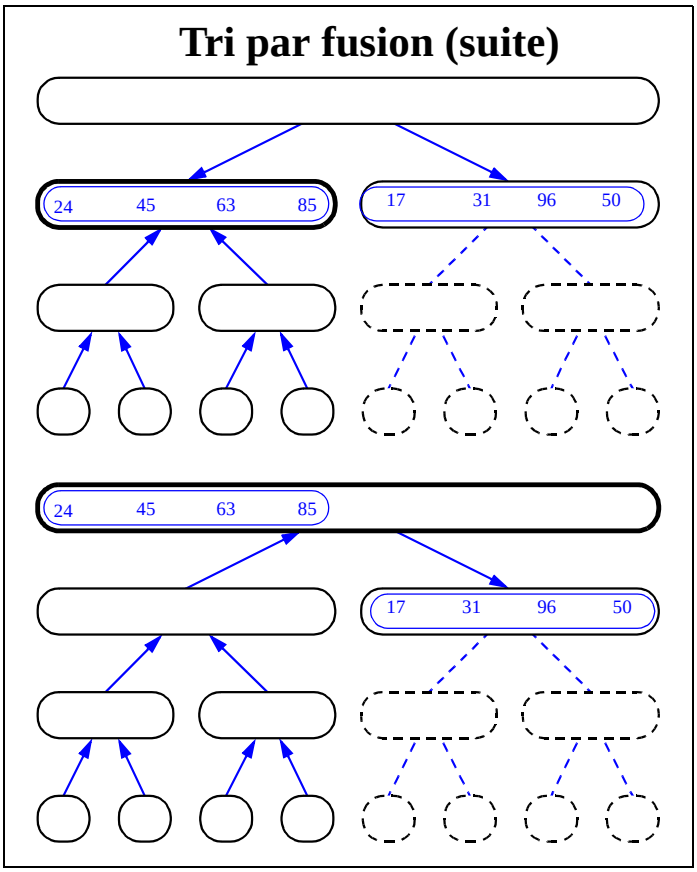
Tri par fusion (suite)



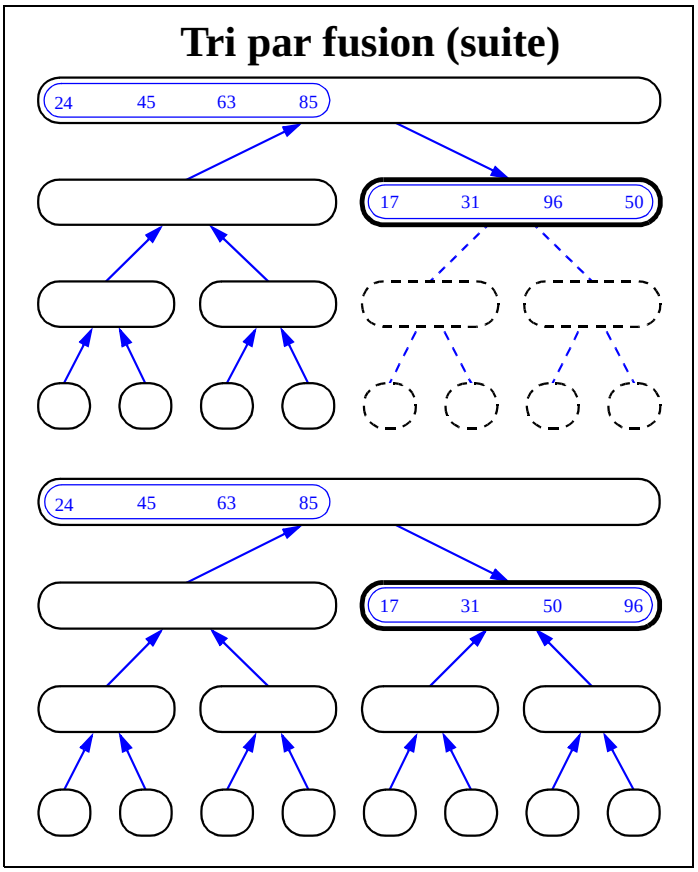
Tri par fusion (suite)



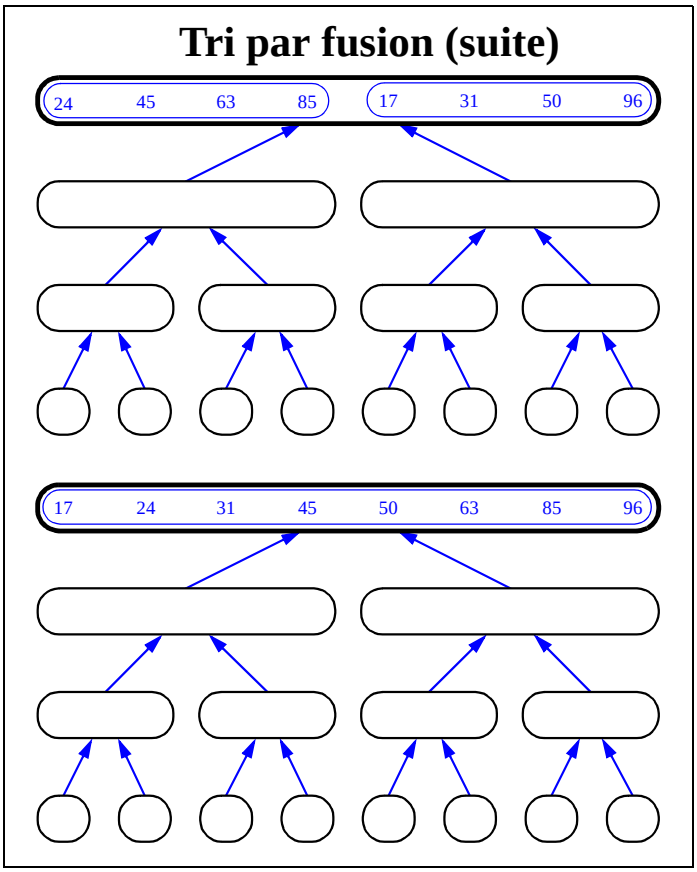
Tri par fusion (suite)



Tri par fusion (suite)



Tri par fusion (suite)



Fusionner deux séquences

- Pseudo-code pour fusionner deux séquences triées en une séquence triée unique

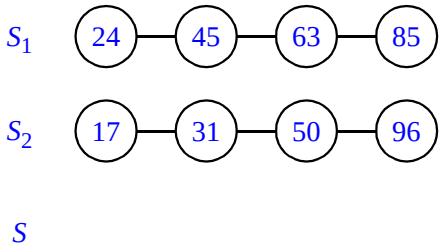
Algorithme merge (S1, S2, S):
Entrée: Séquence *S1* et *S2* (où une relation totale sur les éléments est définie) triée en ordre non-décroissant, et une séquence vide *S*.
Sortie: Séquence *S* contenant l'union des éléments de *S1* et *S2* triés en ordre non-décroissant; les séquences *S1* et *S2* deviennent vides à la fin de l'exécution

```
while S1 is not empty and S2 is not empty do
  if S1.first().element() ≤ S2.first().element() then
    {déplace le 1er élément de S1 vers la fin de S}
    S.insertLast(S1.remove(S1.first()))
  else
    { déplace le 1er élément de S2 vers la fin de S}
    S.insertLast(S2.remove(S2.first()))
while S1 is not empty do
  {déplace les éléments restants de S1 vers S}
  S.insertLast(S1.remove(S1.first()))
while S2 is not empty do
  {déplace les éléments restants de S2 vers S}
  S.insertLast(S2.remove(S2.first()))
```

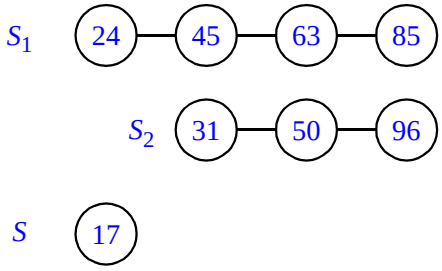

Fusionner deux séquences (suite)

- Quelques illustrations:

a)

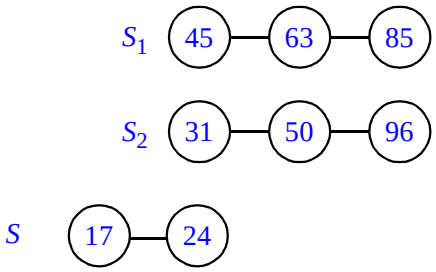


b)

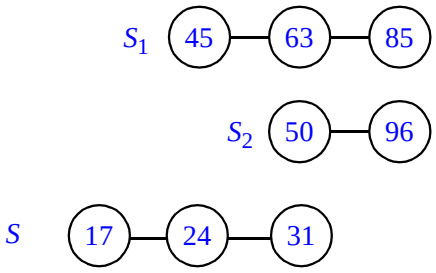


Fusionner deux séquences (suite)

c)

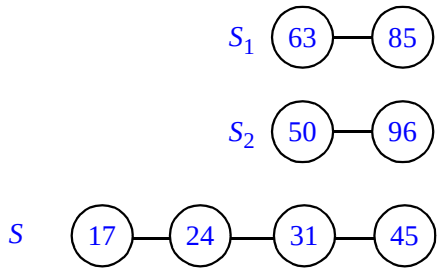


d)

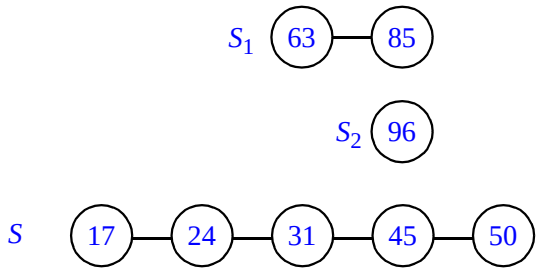


Fusionner deux séquences (suite)

e)

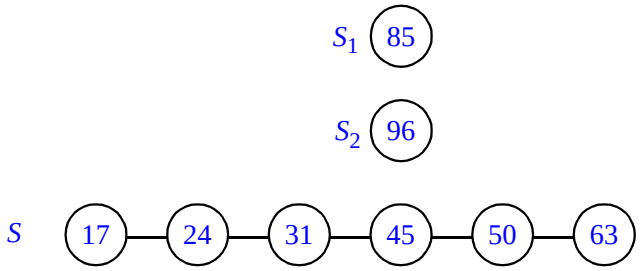


f)

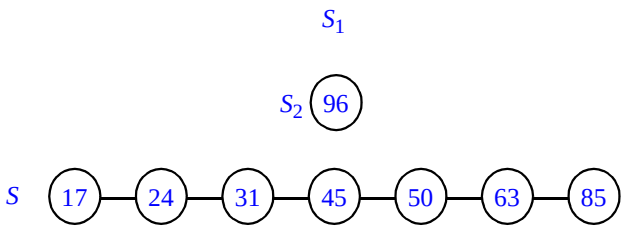


Fusionner deux séquences (suite)

g)

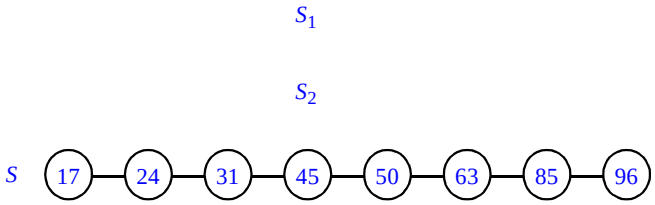


h)



Fusionner deux séquences (suite)

i)



Réalisation Java du tri par fusion

- Interface SortObject

```
public interface SortObject {
  //sort sequence S in nondecreasing order
  using compartor c
  public void sort (Sequence S, Comparator c);
}
```


Réalisation Java du tri par fusion (suite)

```
public class ListMergeSort implements SortObject {
  public void sort(Sequence S, Comparator c) {
    int n = S.size();
   if (n < 2) return; // a sequence with 0 or
  1 element is already sorted.
    // divide
    Sequence S1 = (Sequence)S.newContainer();
    // put the first half of S into S1
    for (int i=1; i <= (n+1)/2; i++) {
     S1.insertLast(S.remove(S.first()));
    }
    Sequence S2 = (Sequence)S.newContainer();
    // put the second half of S into S2
    for (int i=1; i <= n/2; i++) {
      S2.insertLast(S.remove(S.first()));
    }
    sort(S1,c); // recur
    sort(S2,c);
    merge(S1,S2,c,S); // conquer
  }
```

Réalisation Java du tri par fusion (suite)

```
public void merge(Sequence S1, Sequence S2,
    Comparator c, Sequence S) {
  while(!S1.isEmpty() && !S2.isEmpty()) {
    if(c.isLessThanOrEqualTo(S1.first().element(),
      S2.first().element())) {
      // S1's 1st elt <= S2's 1st elt
      S.insertLast(S1.remove(S1.first()));
    }
    else { // S2's 1st elt is the smaller one
      S.insertLast(S2.remove(S2.first()));
    }
  }

  if(S1.isEmpty()) {
    while(!S2.isEmpty()) {
      S.insertLast(S2.remove(S2.first()));
    }
  }
  if(S2.isEmpty()) {
    while(!S1.isEmpty()) {
      S.insertLast(S1.remove(S1.first()));
    }
  }
}
```


Temps d'exécution du tri par fusion

- **Proposition 1**: L'arbre associé à l'exécution du tri par fusion sur une séquence de n éléments a une hauteur de $\lceil \log n \rceil$
- **Proposition 2**: Un algorithme de tri par fusion trie une séquence de taille n en un temps $\boldsymbol{O}(n \log n)$
- Nous supposons seulement que la séquence d'entrée S et chacune des sous-séquences crées par chaque appel récursif de l'algorithme peut accéder, insérer, et supprimer les premier et dernier nœuds en un temps $\boldsymbol{O}(1)$.
- Nous appelons le temps passé à un nœud v d'un arbre de tri par fusion T le temps d'exécution de l'appel récursif associé à v, en excluant les appels récursifs faits aux enfants de v.

Temps d'exécution du tri par fusion (suite)

- Si i représente la profondeur du nœud v dans l'arbre de tri par fusion, alors le temps passé au nœud v est $\boldsymbol{O}(n/2^i)$ puisque la taille associée à v est $n/2^i$.
- Observez que T possède exactement 2^i nœuds à la profondeur i. Le temps total passé à la profondeur i dans l'arbre est alors $\boldsymbol{O}(2^i n/2^i)$, qui est donc $\boldsymbol{O}(n)$. Nous savons que l'arbre a une hauteur $\lceil \log n \rceil$. Ainsi, la complexité temporelle est $\boldsymbol{O}(n \log n)$.

Le TAD Ensemble (*Set*)

- Un ensemble (*set*) est une structure de donnée modélisée selon le concept mathématique d'ensemble. Les opérations fondamentales sur les ensembles sont l'*union*, l'*intersection*, et la *soustraction*.
- Un bref rappel du concept mathématique d'ensemble:
  - $A \cup B = \{ x: x \in A \text{ ou } x \in B \}$
  - $A \cap B = \{ x: x \in A \text{ et } x \in B \}$
  - $A - B = \{ x: x \in A \text{ et } x \notin B \}$
- Les méthodes spécifiques pour un ensemble A incluent:
  - union(B):
    L'ensemble A devient $A \cup B$.
  - intersect(B):
    L'ensemble A devient $A \cap B$.
  - subtract(B):
    L'ensemble A devient $A - B$.

Fusion générique

Algorithme genericMerge(*A*, *B*):
  Entrée: Séquences triées *A* et *B*
  Sortie: Séquence triée *C*
  let *A'* be a copy of *A* { We won't destroy *A* and *B*}
  let *B'* be a copy of *B*
  while *A'* and *B'* are not empty **do**
    a←*A'*.first()
    b←*B'*.first()
    if *a*<*b* **then**
      aIsLess(*a*, *C*)
      A'.removeFirst()
    else if *a*=*b* **then**
      bothAreEqual(*a*, *b*, *C*)
      A'.removeFirst()
      B'.removeFirst()
    else
      bIsLess(*b*, *C*)
      B'.removeFirst()
    while *A'* is not empty **do**
      a←*A'*.first()
      aIsLess(*a*, *C*)
      A'.removeFirst()
    while *B'* is not empty **do**
      b←*B'*.first()
      bIsLess(*b*, *C*)
      B'.removeFirst()

Opérations sur les ensembles

- Nous pouvons spécialiser l'algorithme de fusion générique pour réaliser des opérations sur les ensembles telles l'union, l'intersection, et la soustraction.
- L'algorithme de fusion générique examine et compare les éléments courants *A* et *B*.
- En se basant sur le résultat de la comparaison, il détermine s'il doit copier l'un des éléments *a* ou *b* dans *C*, ou ne rien faire.
- Cette décision dépend de l'opération présentement en cours (union, intersection ou soustraction).
- Dans le cas de l'union, nous copions le plus petit élément (*a* ou *b*) dans *C*; si *a*=*b* alors l'un ou l'autre est copié.
- Pour copier, nous définissons nos actions comme étant aIsLess, bothAreEqual, et bIsLess.
- Allons voir la réalisation...

Opérations sur les ensembles (suite)

- Pour l'union

```
public class UnionMerger extends Merger {
  protected void aIsLess(Object a, Object b, Sequence C) {
    C.insertLast(a);
  }
  protected void bothAreEqual(Object a, Object b,
                              Sequence C) {
    C.insertLast(a);
  }
  protected void bIsLess(Object b, Sequence C) {
    C.insertLast(b);
  }
```

- Pour l'intersection

```
public class IntersectMerger extends Merger {
  protected void aIsLess(Object a, Object b, SequenceC) {
  }
  protected void bothAreEqual(Object a, Object b,
                              Sequence C) {
    C.insertLast(a);
  }
  protected void bIsLess(Object b, Sequence C) { }
}
```


Opérations sur les ensembles (suite)

- Pour la soustraction

```
public class SubtractMerger extends Merger {
  protected void aIsLess(Object a, Object b,
                         Sequence C) {
    C.insertLast(a);
  }

  protected void bothAreEqual(Object a, Object b,
                              Sequence C) {
  }

  protected void bIsLess(Object b, Sequence C) {
  }
}
```

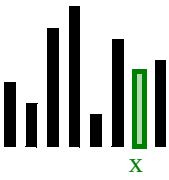
Tri rapide *Quicksort*

Tri rapide *Quick-Sort*

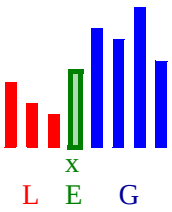
- Afin de comprendre le tri rapide *quick-sort*, regardons une description de haut niveau de l'algorithme.
- 1) **Diviser**: Si la séquence *S* a plus d'un élément, sélectionnez un élément *x* de *S* comme pivot. N'importe quel élément, par exemple le dernier, fera l'affaire. Retirez tous les éléments de *S* et divisez-les en 3 séquences:
  - *L*, contient les éléments de *S* plus petits que *x*
  - *E*, contient les éléments de *S* égaux à *x*
  - *G*, contient les éléments de *S* plus grands que *x*
- 2) **Appliquer récursivement**: Triez récursivement *L* et *G*
- 3) **Conquérir**: Afin de remettre les éléments dans *S* en ordre, insérez premièrement les éléments de *L*, suivis de ceux de *E*, et enfin de ceux de *G*.
- Voici quelques jolies illustrations...

Idée derrière Quick-Sort

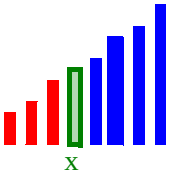
1. Sélectionner
choisissez *un* élément



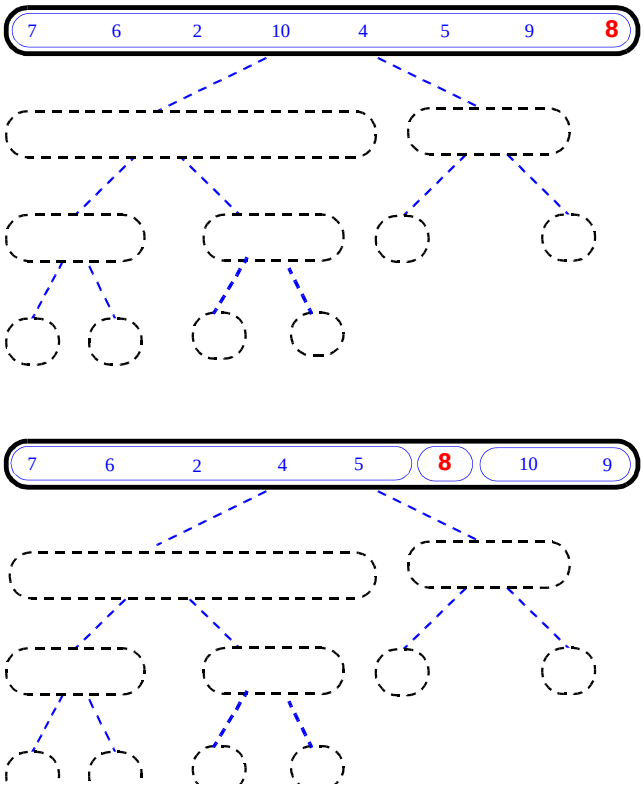
2. Diviser
réorganisez les éléments de façon à ce que
- x aille à sa position finale E



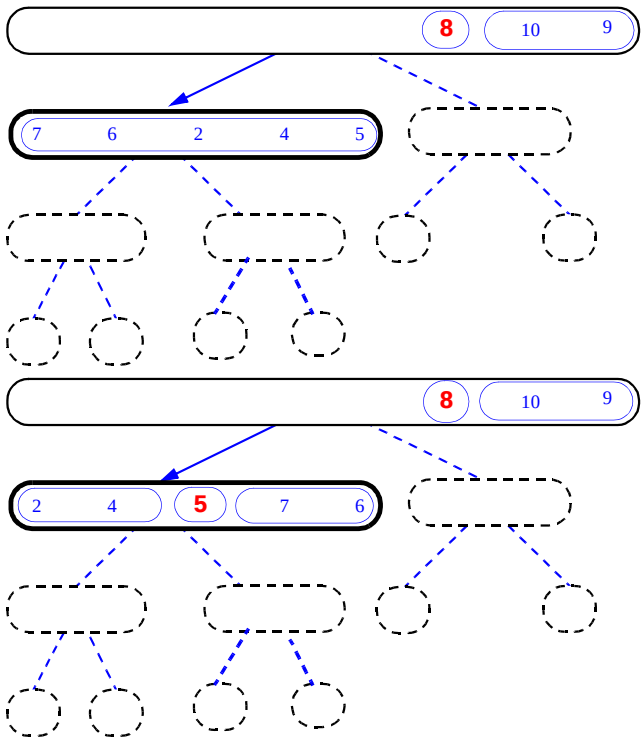
3. Appliquer récursivement et conquérir
triez récursivement



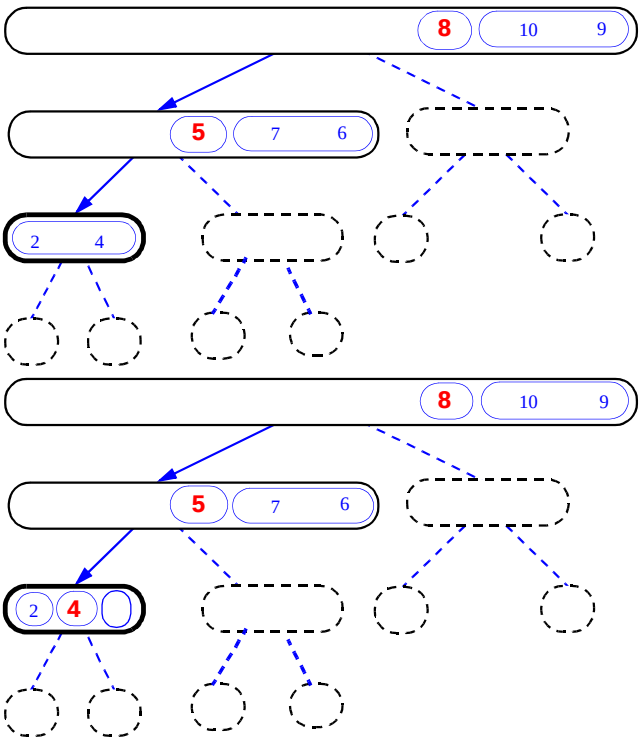
Arbre Quick-Sort



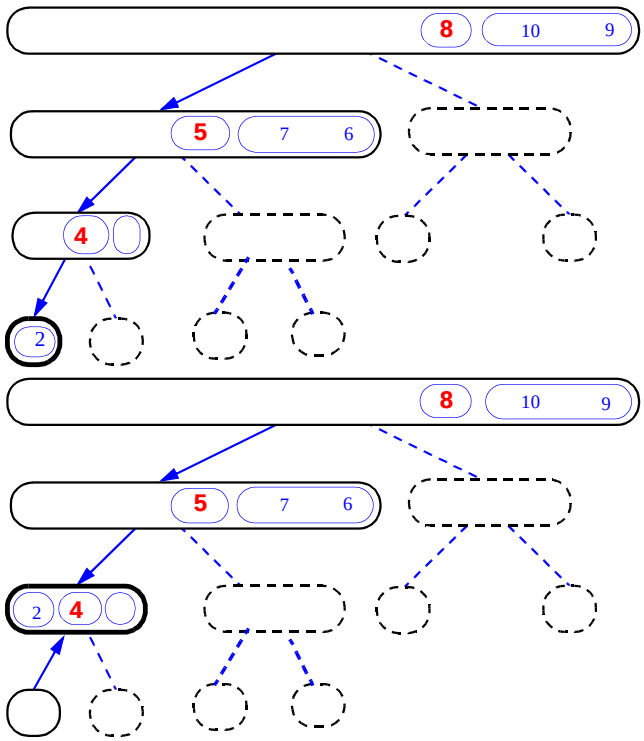
Arbre Quick-Sort



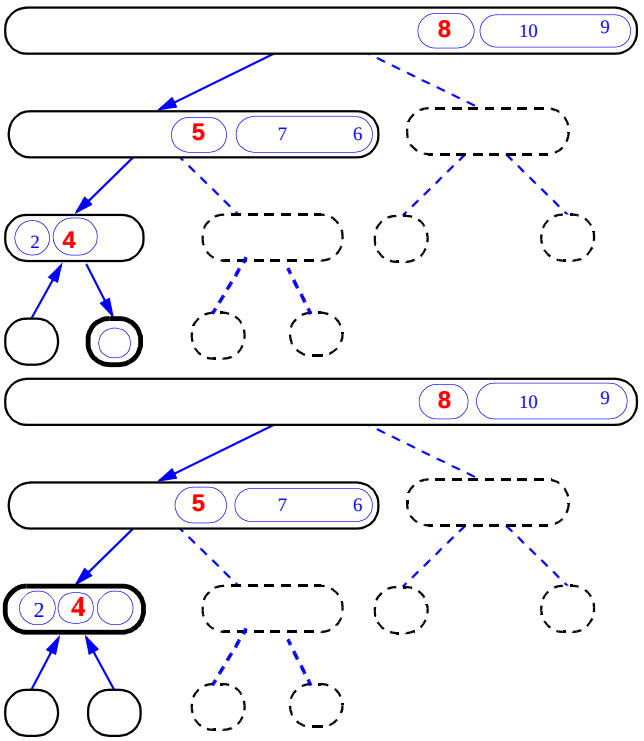
Arbre Quick-Sort



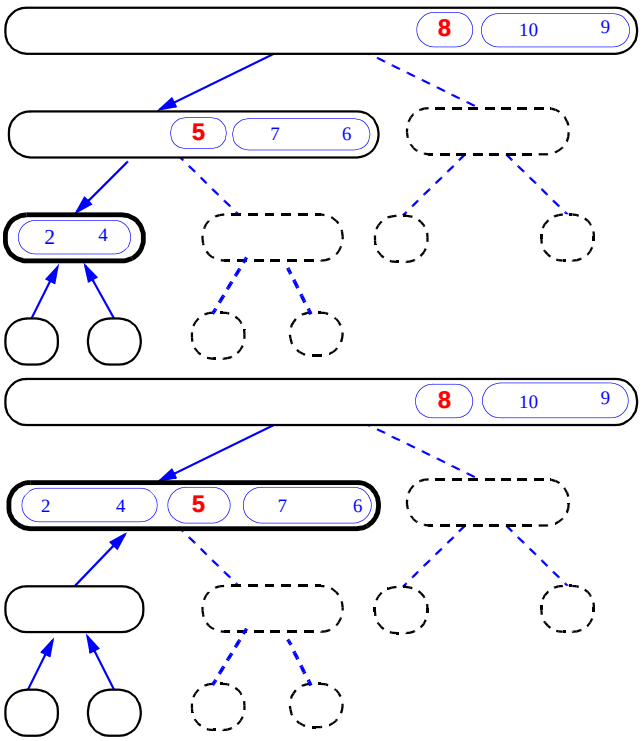
Arbre Quick-Sort



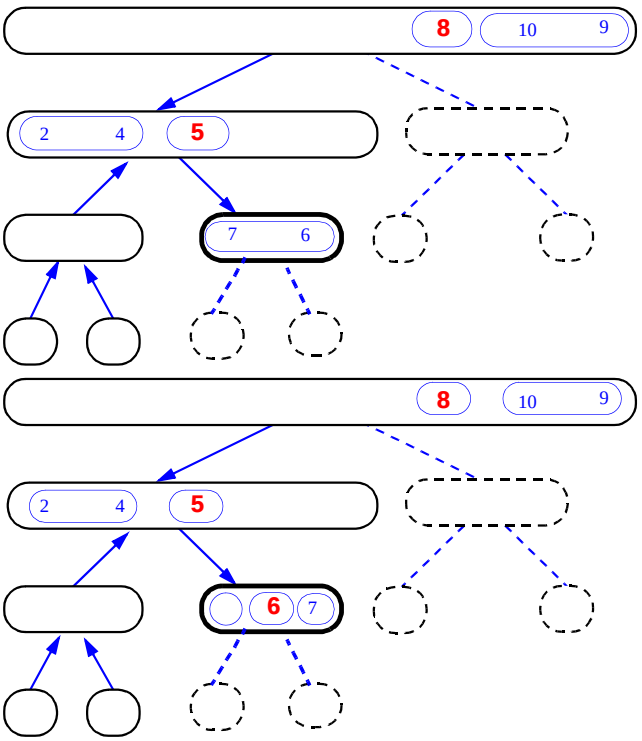
Arbre Quick-Sort



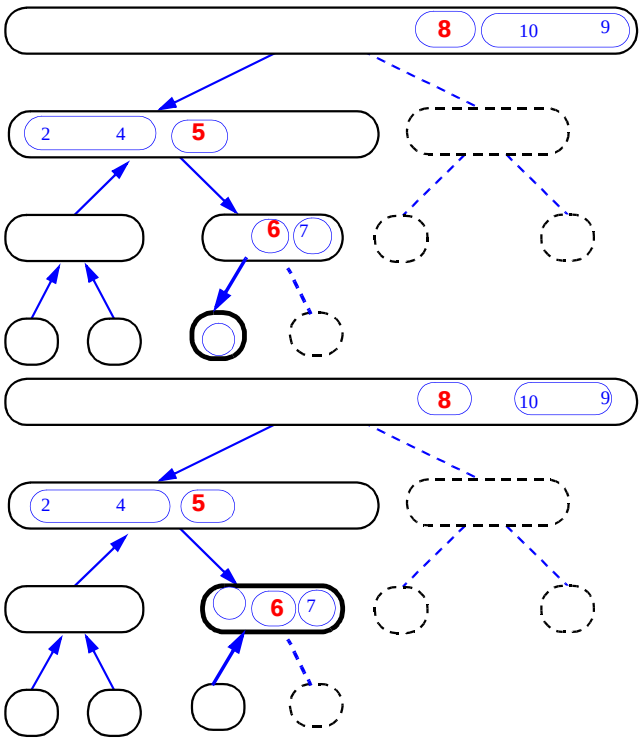
Arbre Quick-Sort



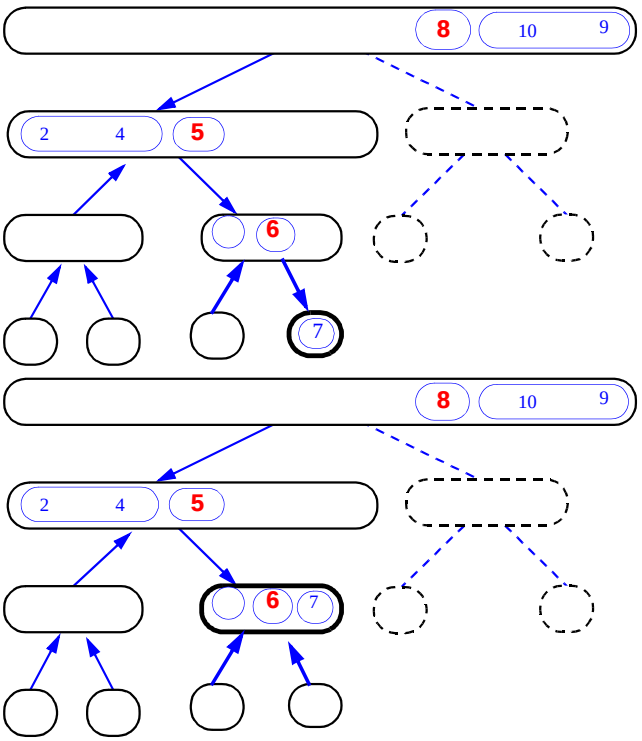
Arbre Quick-Sort



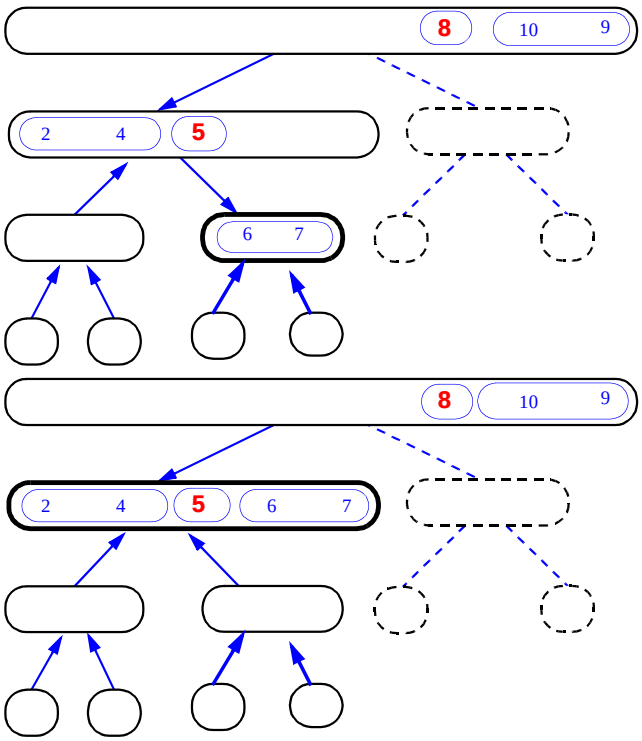
Arbre Quick-Sort



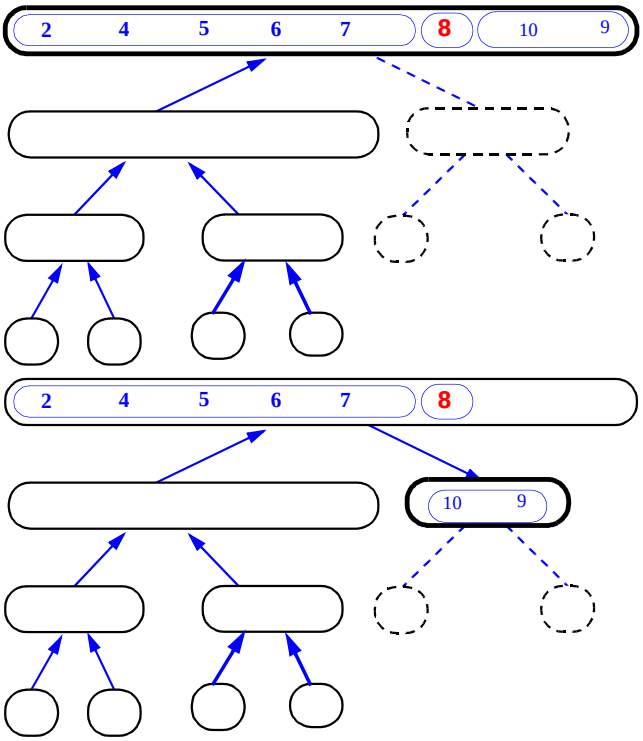
Arbre Quick-Sort



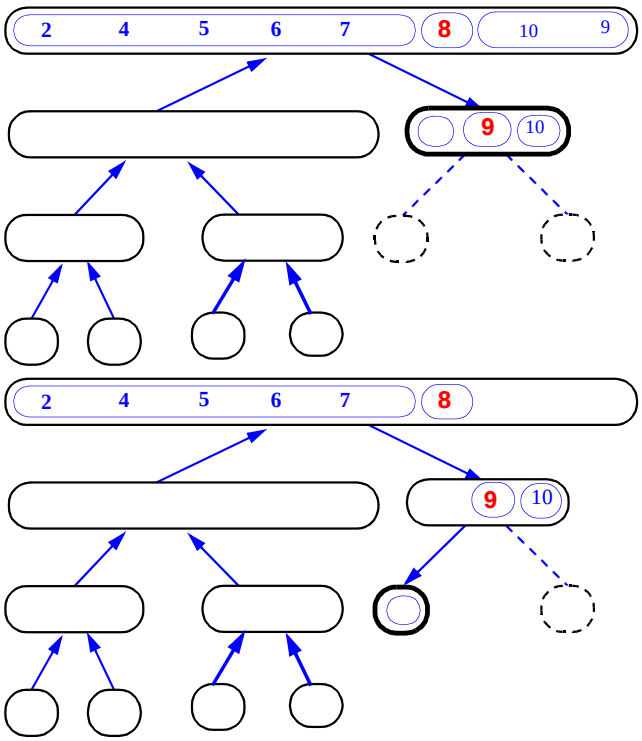
Arbre Quick-Sort



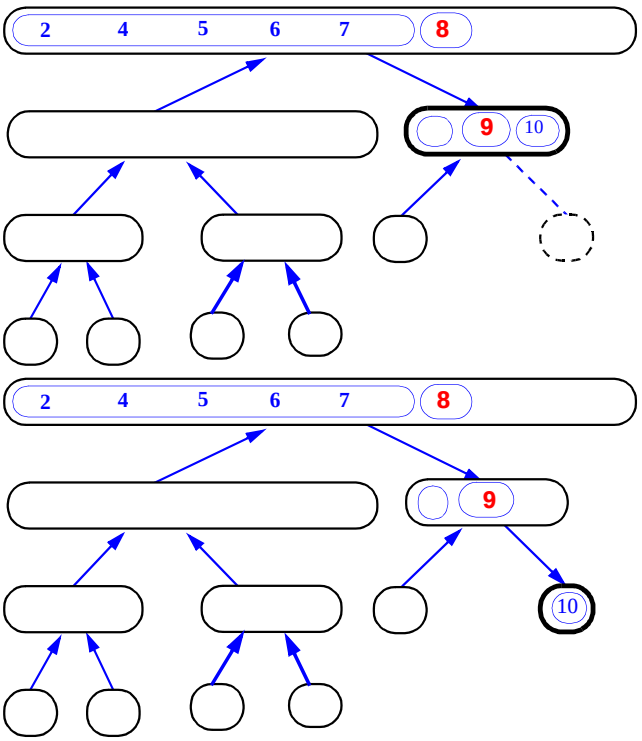
Arbre Quick-Sort



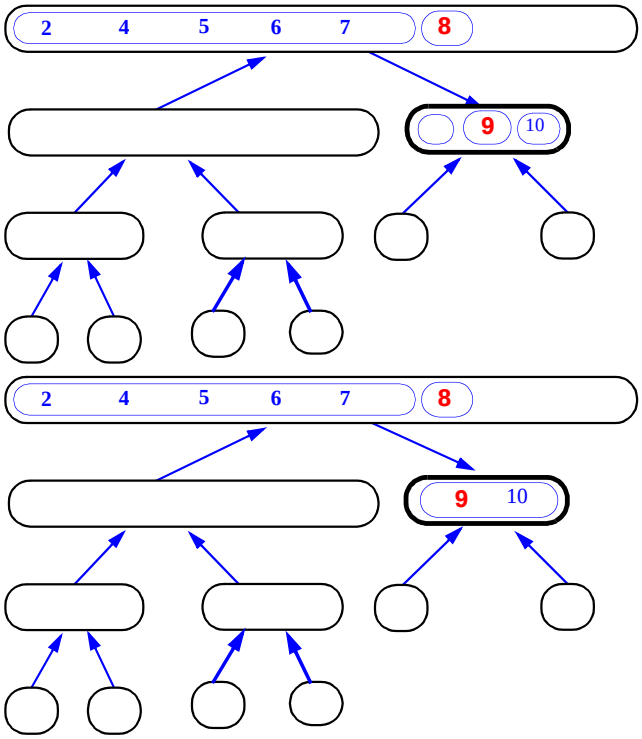
Arbre Quick-Sort



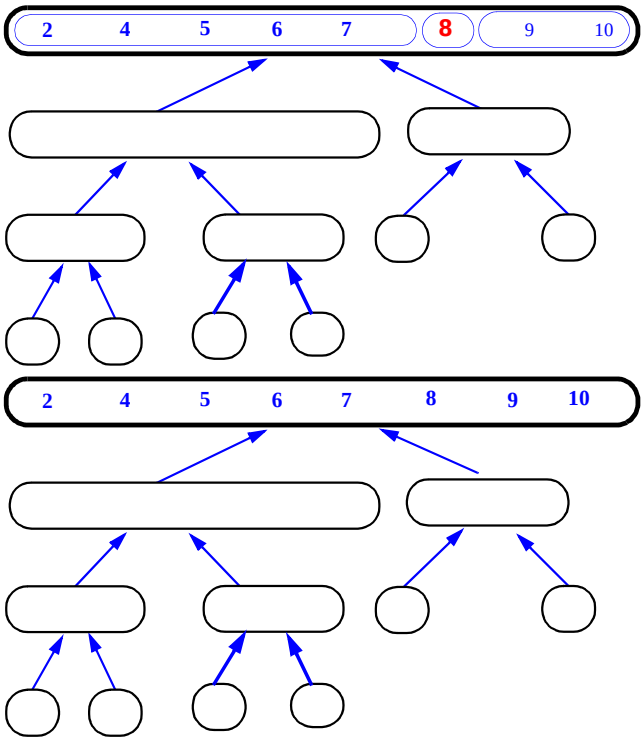
Arbre Quick-Sort



Arbre Quick-Sort

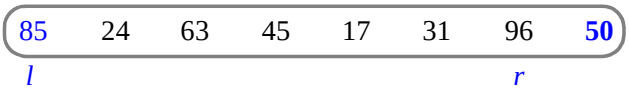


Arbre Quick-Sort

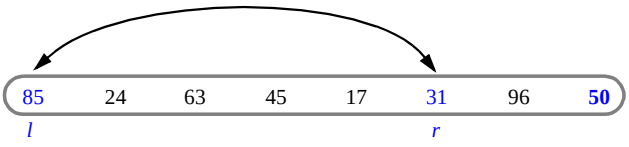


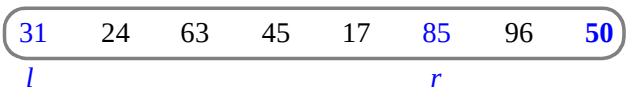
Quick-Sort sur place (*In-Place*)

- **Étape de division**: *l* parcours la séquence à partir de la gauche, et *r* de la droite.

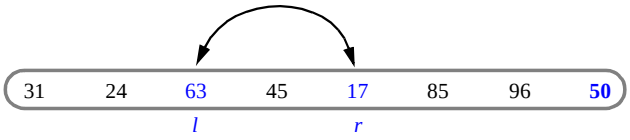


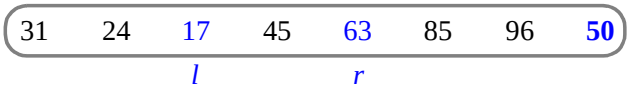
- Un échange a lieu quand *l* est un élément plus grand que le pivot et *r* est plus petit que le pivot.

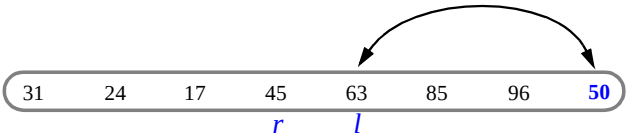




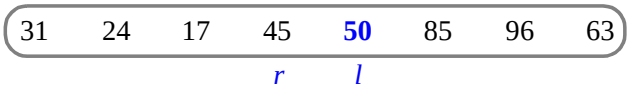
Quick-Sort sur place (suite)







- Un dernier échange avec le pivot complète l'étape de division



Réalisation Java du Quick-Sort sur place

```
public class ArrayQuickSort implements SortObject {

  public void sort(Sequence S, Comparator c){
    quicksort(S, C, 0, S.size()-1);
  }

  private void quicksort (Sequence S, Comparator c,
                          int leftBound,
                          int rightBound) {
              // left and rightmost ranks of
                   // sorting range
    if (S.size() < 2) return; //a sequence with 0 or
                   // 1 elements is already sorted
    if (leftBound >= rightBound) return; //terminate
                                  //recursion
    // pick the pivot as the current last
    // element in range
    Object pivot = S.atRank(rightBound).element();
  // indices used to scan the sorting range
    int leftIndex = leftBound; // will scan
                               // rightward
    int rightIndex = rightBound - 1; //will scan
                                  // leftward
```


Réalisation Java du Quick-Sort sur place (suite)

```
// outer loop
while (leftIndex <= rightIndex) {

  //scan rightward until an element larger than
  //the pivot is found or the indices cross
  while ((leftIndex <= rightIndex) &&
              (c.isLessThanOrEqualTo
              (S.atRank(leftIndex).element(),pivot))
    leftIndex++;

  //scan leftward until an element smaller than
  //the pivot is found or the indices cross
  while (rightIndex >= leftIndex) &&
              (c.isGreaterThanOrEqualTo
              (S.atRank(rightIndex).element(),pivot))
    rightIndex--;

  //if an element larger than the pivot and an
  //element smaller than the pivot have been
  //found, swap them
  if (leftIndex < rightIndex)
    S.swap(S.atRank(leftIndex),S.atRank(rightIndex));

} // the outer loop continues until
  // the indices cross. End of outer loop.
```


Réalisation Java du Quick-Sort sur place (suite)

```
  //put the pivot in its place by swapping it
  //with the element at leftIndex
  S.swap(S.atRank(leftIndex),S.atRank(rightBound));

  // the pivot is now at leftIndex, so recur
  // on both sides
  quicksort (S, c, leftBound, leftIndex-1);
  quickSort (S, c, leftIndex+1, rightBound);
 } // end quicksort method
} // end ArrayQuickSort class
```


Analyse du temps d'exécution

- Supposez un arbre quick-sort T:
  - $s_i(n)$ indique la somme des tailles d'entrée des nœuds à la profondeur i dans T.
- Nous savons que $s_0(n) = n$ puisque la racine de T est associée avec l'ensemble des entrées tout entier.
- Aussi, $s_1(n) = n$-1 puisque le pivot n'est pas propagé.
- Donc: $s_2(n) = n$ - 3, ou encore n - 2 (si l'un des nœuds a une taille d'entrée à zéro).
- Le temps d'exécution de quick-sort est, dans le pire des cas:

$$O\left(\sum_{i=0}^{n-1} s_i(n)\right)$$

Ce qui revient à:

$$O\left(\sum_{i=0}^{n-1} (n-i)\right) = O\left(\sum_{i=1}^{n} i\right) = O(n^2)$$

Donc le tri quick-sort s'exécute en $O(n^2)$ dans le pire des cas.

Analyse du temps d'exécution (suite)

- Maintenant observons le meilleur des cas:
- Quick-sort se comporte de façon optimale lorsque la séquence S est divisée en sous-séquences L et G de tailles égales.
- Plus précisément:
  - $s_0(n) = n$
  - $s_1(n) = n - 1$
  - $s_2(n) = n - (1 + 2) = n - 3$
  - $s_3(n) = n - (1 + 2 + 2^2) = n - 7$

    ...
  - $s_i(n) = n - (1 + 2 + 2^2 + ... + 2^i\text{-}1) = n - 2^i + 1$

    ...
- Ceci implique que T a une hauteur $\boldsymbol{O}(\log n)$
- Complexité temporelle dans le meilleur des cas: $\boldsymbol{O}(n \log n)$

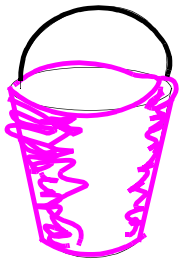
Quick-Sort aléatoire

- Sélectionnez un élément de la séquence ***au hasard*** comme pivot
- Le temps d'exécution attendu d'un tel tri sur une séquence de taille n est $O(n \log n)$. Le temps passé à un niveau de l'arbre quick-sort est $O(n)$
- Nous démontrons que la ***hauteur escomptée*** de l'arbre quick-sort est $O(\log n)$
- Bons et mauvais pivots

  - ***Bon***: $1/4 \le n_L/n \le 3/4$
  - ***Mauvais***: $n_L/n < 1/4$ **ou** $n_L/n > 3/4$
- La probabilité d'obtenir un bon pivot est 1/2, donc nous espérons $k/2$ bons pivots
- Après un bon pivot, la taille de chaque sous-séquence est au plus 0.75 fois la taille de la séquence originale
- Après h pivots, nous espérons $(3/4)^{h/2}\, n$ éléments
- La hauteur escomptée h de l'arbre quick-sort est d'au plus: $\mathbf{2 \log_{4/3} n}$

Encore du tri

- Tri numérique (*radix sort*)
- Tri *bucket sort*
- Tri sur place (*in-place*)
- À quelle vitesse peut-on trier?



Tri numérique (*Radix Sort*)

- Contrairement aux autres méthodes, le tri numérique (*radix sort*) considère la structure des clés
- Supposons des clés représentées dans un système numérique à base M (*radix*); si M = 2, alors les clés sont représentées en binaire

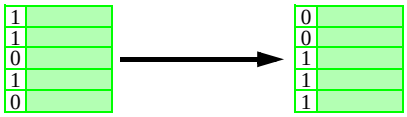
	8	4	2	1	poids
9 =	1	0	0	1	(b = 4)
	3	2	1	0	bit #

- Le tri se fait en comparant les bits à la même position
- Extension aux clés formées de chaînes alphanumériques

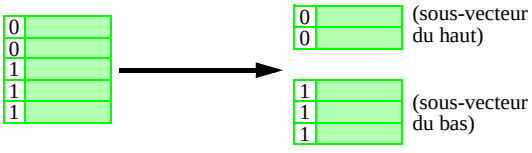
Tri numérique avec échange (*Radix Exchange Sort*)

Examinez les bits de ***gauche*** à ***droite***:

1. **Triez le vecteur selon le bit le plus à gauche**



2. **Partitionnez le vecteur**



3. **Récursivité**
   - triez récursivement le sous-vecteur du haut, en ignorant le bit le plus à gauche
   - triez récursivement le sous-vecteur du bas, en ignorant le bit le plus à gauche

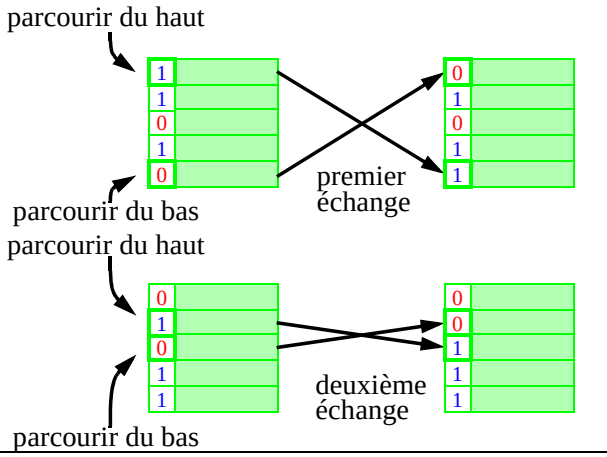
Temps requis pour trier n nombres à b bits:

$$O(b\ n)$$

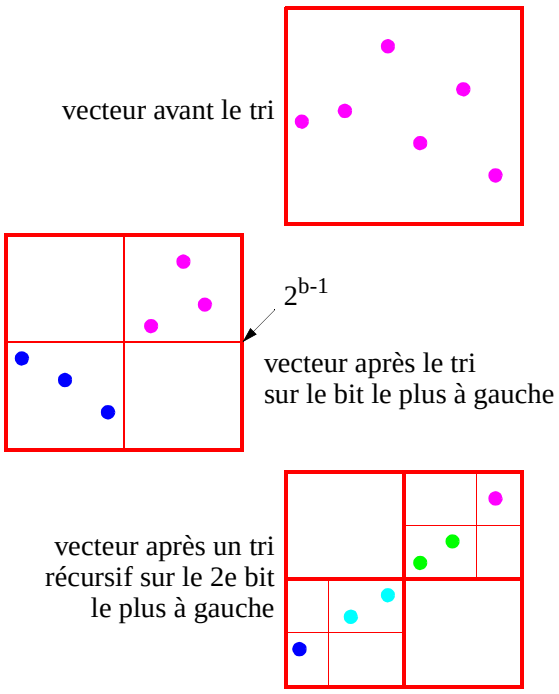
Tri numérique avec échange

Comment réalisons-nous le tri de la page précédente? Même idée que la partition dans Quicksort.

répétez
  parcourir de haut en bas pour trouver une clé débutant par un 1;
  parcourir de bas en haut pour trouver une clé débutant par un 0;
  échangez les clés;
jusqu'à ce que les indices de parcours se croisent



Tri numérique avec échange



Tri numérique avec échange versus Quicksort

Similarités

- Les deux partitionnent le vecteur
- Les deux trient les sous-vecteurs récursivement

Différences

- ***Méthode de partitionnement***
  - le tri numérique divise le vecteur selon la relation plus grand ou égal à 2^{b-1}
  - quicksort partitionne le vecteur selon la relation plus grand ou égal à un élément du vecteur
- ***Complexité temporelle***
  - Numérique avec échange O (*bn*)
  - Quicksort, cas typique O (*n* log *n*)
  - Quicksort, pire des cas O (n^2)

Tri numérique direct

Examinez les bits de ***droite*** à ***gauche***

for k := 0 **to** b–1
  triez le vecteur de façon ***stable***,
  en ne regardant que le bit k

Première-ment, triez ceux-ci | Ensuite, triez ces bits | Enfin, triez ceux-ci

0	1	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0
1	0	1	1	0	0	1	0	1	0	1	0
0	0	1	1	1	0	0	1	0	1	1	0
1	1	1	1	0	1	0	1	1	0	0	1
0	1	1	0	0	1	1	0	1	0	1	1
1	0	0	1	1	1	1	1	1	1	0	1
1	1	0	0	1	1	0	1	1	1	1	1

Notez l'ordre de ces bits après le tri.

Mais que signifie "trier de façon stable"?

Dans un tri stable, l'ordre initial relatif de clés égales demeure inchangé.

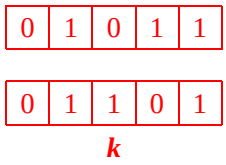
Par exemple, observez la première étape du tri de la page précédente:

0	1	0	→				0	1	0		
0	0	0					0	0	0		
1	0	1					1	0	0		
0	0	1					1	1	0		
1	1	1					1	0	1		
0	1	1					0	0	1		
1	0	0					1	1	1		
1	1	0					0	1	1		

Notez que l'ordre relatif des clés se terminant par 0 est inchangé, et que la situation est semblable pour les éléments se terminant par 1.

L'algorithme est correct (vrai?)

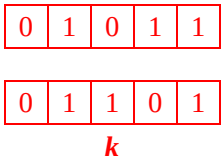
- Nous démontrerons que n'importe quelle paire de clés se trouve correctement ordonnée à la fin de l'algorithme
- Étant donné deux clés, définissons ***k*** comme étant la position du bit le plus à gauche où elles diffèrent

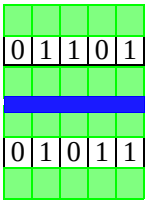


- À l'étape ***k*** les deux clés sont mises dans un ordre relatif correct
- Grâce à la ***stabilité***, les étapes subséquentes ne changent pas l'ordre relatif des deux clés!

Par exemple,

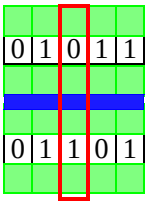
Considérez un tri sur un vecteur avec ces deux clés:

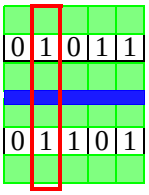




Leur ordre relatif initial n'a aucune importance.

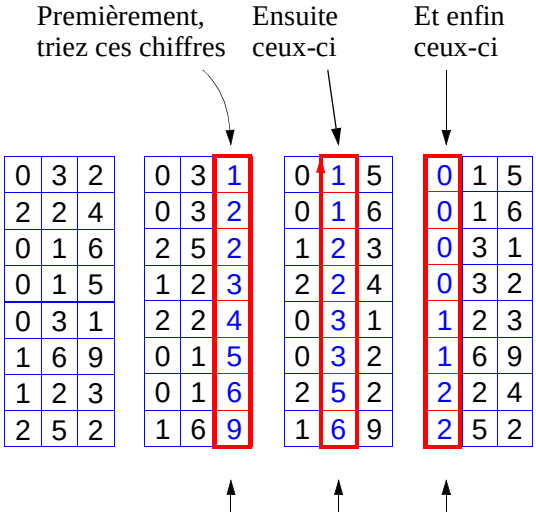
Quand le tri visite le bit ***k***, les clés sont mises dans un ordre relatif correct.





Comme le tri est stable, l'ordre des deux clés ne changera pas lorsque les bits > ***k*** seront comparés.

Le tri numérique peut être appliqué aux nombres décimaux



Notez l'ordre des chiffres après le tri.

Voilà!

Complexité temporelle du tri numérique direct

for $k = 0$ **to** $b - 1$
  triez le vecteur de façon *stable*,
  en ne regardant que le bit k

Supposons que nous puissions exécuter ce tri stable en un temps $O(n)$. La complexité temporelle totale serait:

$$O(bn)$$

Comme vous l'avez peut-être deviné, nous pouvons faire un tri stable basé sur le $\boldsymbol{k}^{\text{e}}$ chiffre des clés en un temps $O(n)$.

Par quelle méthode? Par un tri *bucket sort*, bien sûr. →

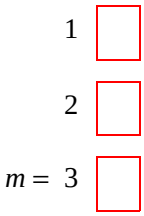
Tri *Bucket Sort*

- n nombres
- Chaque nombre $\in \{1, 2, 3, \ldots m\}$
- Stable
- Temps: $O(n + m)$

Par exemple, $m = 3$ et notre vecteur initial est:

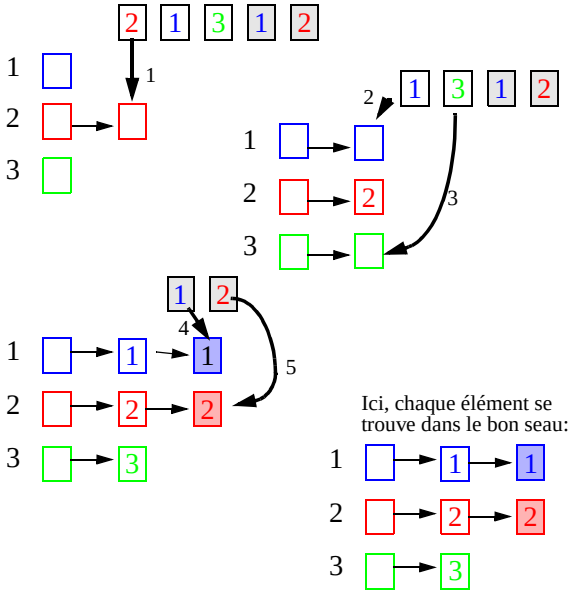
(notez qu'il y a deux "2" et deux "1")

Premièrement, nous créons m "seaux" (*buckets*)



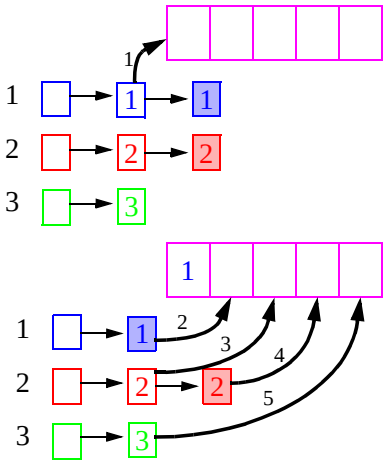
Tri *Bucket Sort*

Chaque élément du vecteur est placé dans l'un des m "seau"

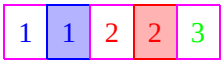


Tri *Bucket Sort*

Maintenant, transférez les éléments des seaux vers le vecteur



Enfin, le vecteur trié (trié de façon ***stable***):



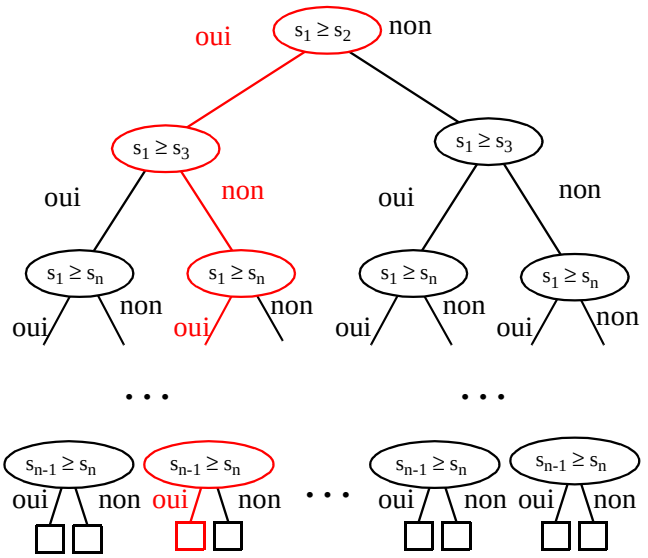
Tri sur place (*in-place*)

- Un algorithme de tri est dit ***sur place*** si
  - il n'utilise aucune structure de données auxiliaire (cependant, $O(1)$ variables auxiliaires sont permises)
  - il met à jour la séquence d'entrée en n'utilisant seulement que les opérations replaceElement et swapElements
- Quels algorithmes de tri vus jusqu'à maintenant peuvent fonctionner *sur place*?

tri à bulle (*bubble sort*)	Y
tri par sélection	
tri par insertion	
tri par tas (*heap sort*)	
tri par fusion (*merge sort*)	
tri rapide (*quick sort*)	
tri numérique (*radix sort*)	
tri *bucket sort*	

Arbre de décision pour tri basé sur des comparaisons

- nœud interne: comparaison
- nœud externe: permutation
- exécution de l'algorithme: racine vers feuille

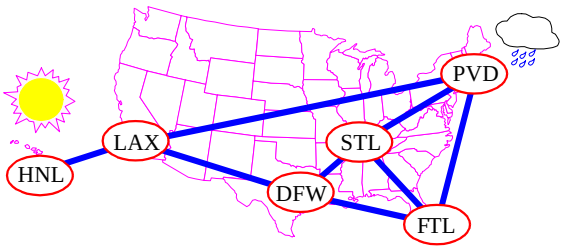


À quelle vitesse peut-on trier?

- **Proposition**: Le temps d'exécution de tout algorithme basé sur des comparaisons et servant à trier une séquence S de n éléments est $\Omega(n \log n)$.
- **Justification**:
- Le temps d'exécution d'un algorithme de tri basé sur des comparaisons doit être plus grand ou égal à la profondeur de l'arbre de décision T associé à cet algorithme.
- Chaque nœud interne de T est associé à une comparaison qui établit l'ordre de deux éléments de S.
- Chaque nœud externe de T représente une permutation distincte des éléments de S.
- Ainsi T doit avoir au moins $n!$ nœuds externes, ce qui implique que T a une hauteur d'au moins $\log(n!)$
- Puisque $n!$ a au moins $n/2$ termes qui sont plus grand ou égal à $n/2$, nous avons:
$$\log(n!) \geq (n/2) \log(n/2)$$
- Complexité temporelle totale: $\Omega(n \log n)$.

GRAPHES

- Définitions
- Le TAD Graphe
- Structures de données pour graphes



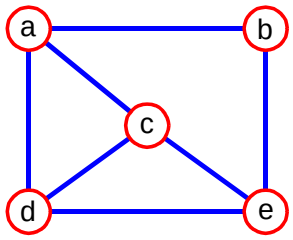
Qu'est-ce qu'un graphe?

- Un **graphe G** = (**V**,**E**) est composé de:

  V: ensemble de **sommets** (*vertices*)

  E: ensemble d'**arcs** (*edges*) reliant les sommets de **V**

- Un **arc e** = (u,v) est une paire de sommets
- Exemple:

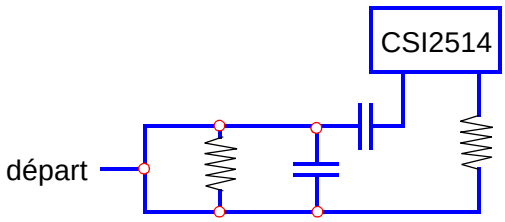


V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

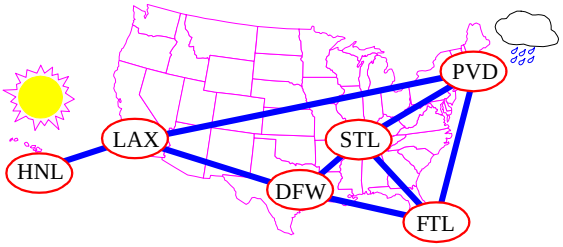
Applications

- circuits électroniques



trouvez le chemin à la moindre résistance menant à CSI2514

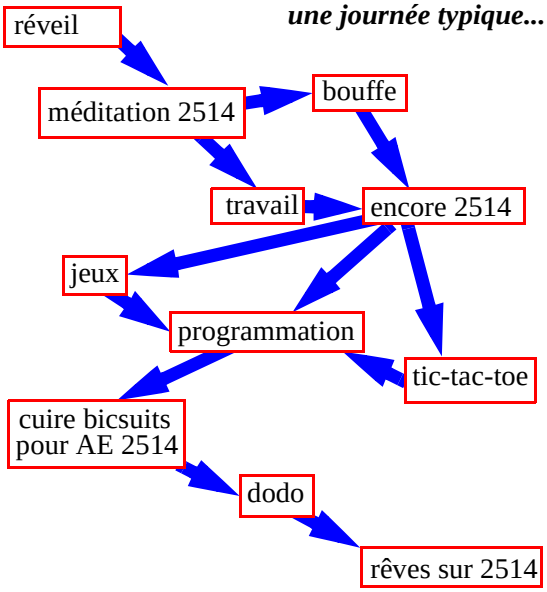
- réseaux (routiers, aériens, de communication)



De meilleurs exemples...

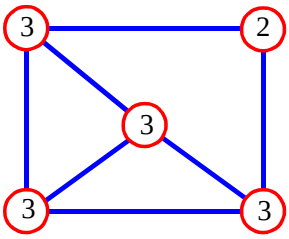
- conception d'horaires (planification de projet)

une journée typique...



Terminologie des graphes

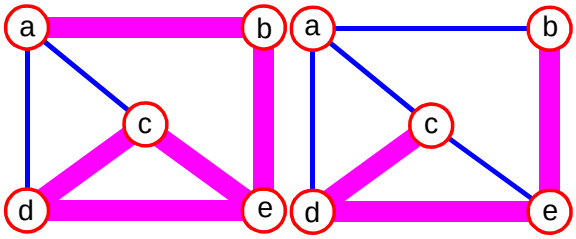
- sommets adjacents: reliés par un arc
- degré (d'un sommet): # de sommets adjacents



$\sum_{v \in V} \deg(v) = 2(\#\text{ arcs})$

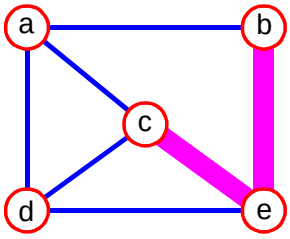
- Comme des sommets adjacents comptent tous deux l'arc les reliant, celui-ci sera compté deux fois.

chemin: séquence de sommets $v_1, v_2, \ldots v_k$ où les sommets consécutifs v_i et v_{i+1} sont adjacents.



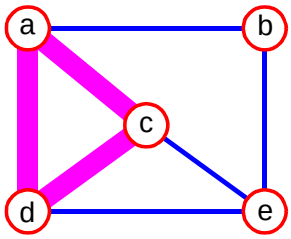
Encore de la terminologie...

- chemin simple: sans aucun sommet répété

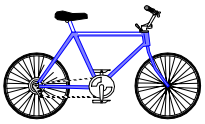


b e c

- cycle: chemin simple, sauf que le dernier sommet est le même que le tout premier

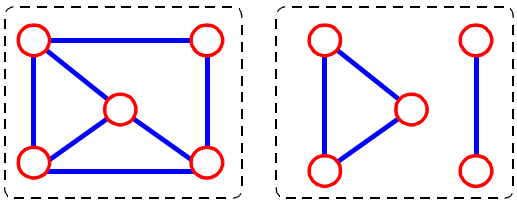


a c d a



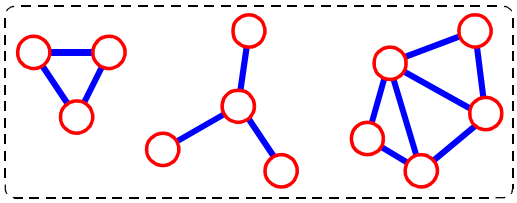
Et encore de la terminologie...

- graphe connexe: toutes les paires de sommets sont reliées par un chemin



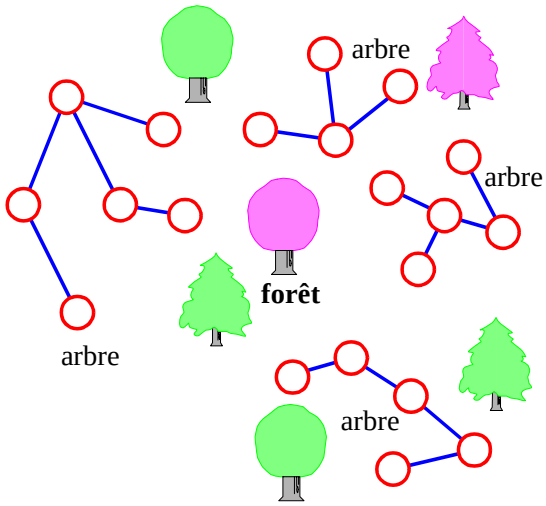
connexe non-connexe

- sous-graphe: sous-ensemble de sommets et d'arcs formant un graphe
- composante connexe: sous-graphe connexe maximal. Par exemple, le graphe ci-dessous a 3 composantes connexes.



¡Caramba! Encore de la terminologie!

- arbre (libre) - graphe connexe sans cycle
- forêt - ensemble d'arbres



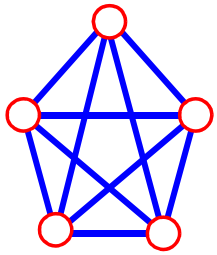
Connectivité

Soit **n** = #sommets
m = #arcs

- graphe complet - toutes les paires de sommets sont adjacentes

$$\mathbf{m} = (1/2)\sum_{v \in \mathbf{V}} \deg(\mathbf{v}) = (1/2)\sum_{v \in V} (\mathbf{n} - 1) = \mathbf{n}(\mathbf{n}-1)/2$$

- Chacun des **n** sommets est attaché à **n** - 1 arcs, cependant, nous aurons compté chaque arc deux fois!!! Ainsi, intuitivement, **m** = **n**(**n**-1)/2.



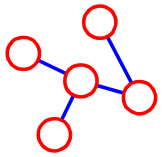
n = 5
m = $(5 * 4)/2 = 10$

- Donc, si un graphe n'est ***pas*** complet,
  m < **n**(**n**-1)/2

Plus de connectivité

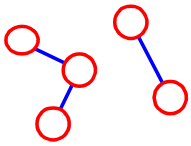
n = #sommets
m = #arcs

- Pour un arbre **m** = **n** - 1



n = 5
m = 4

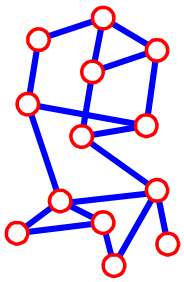
- Si **m** < **n** - 1, alors le graphe **G** n'est pas connexe



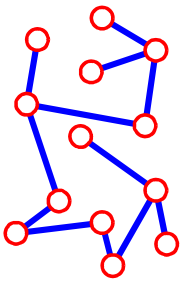
n = 5
m = 3

Arbre recouvrant (*Spanning Tree*)

- Un **arbre recouvrant** (*spanning tree*) de **G** est un sous-graphe qui:
  - est un arbre
  - contient tous les sommets de **G**



G



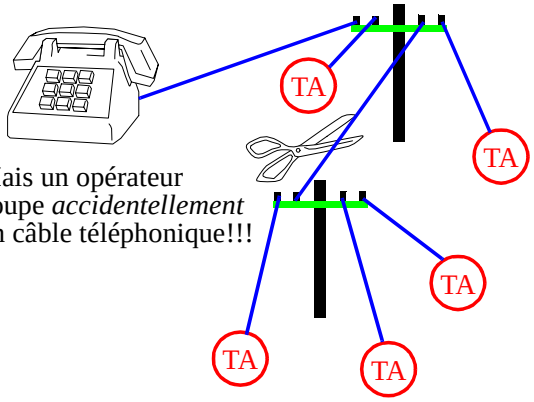
arbre recouvrant de **G**

- Une faute affectant n'importe quel arc rend le système non-connexe (l'arbre recouvrant est la configuration la moins tolérante aux fautes)

Bell Canada contre SM&T

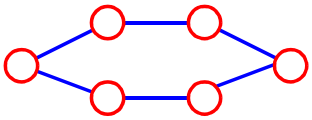
(Stan Matwin & Telephone)

- Stan désire appeler ses AE afin de suggérer une extension pour le prochain devoir...

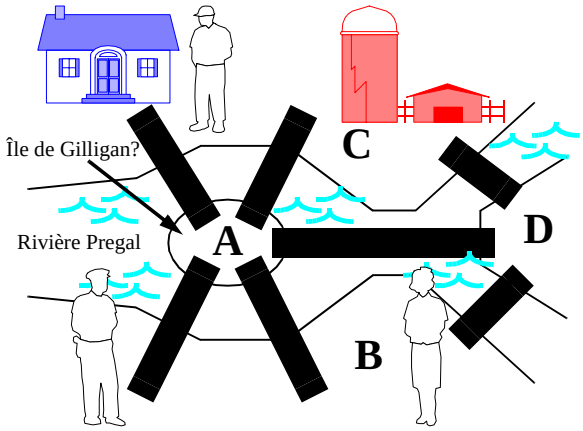


Mais un opérateur coupe *accidentellement* un câble téléphonique!!!

- Une faute va déconnecter une partie du graphe!
- Un cycle serait plus tolérant aux fautes et n'exige que **n** arcs.



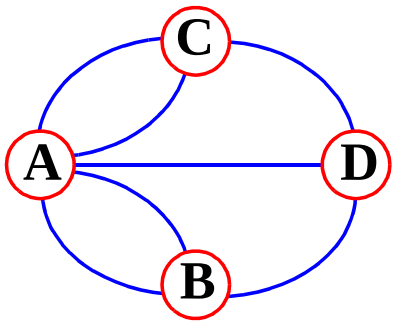
Euler et les ponts de Koenigsberg



Peut-on traverser chaque pont exactement une fois et retourner au point de départ?

- Mettez-vous à la place d'un conducteur de UPS ou de Fedex qui ne voudrait pas revenir sur son chemin.
- En 1736, Euler a prouvé que ce n'est pas possible.

Modèle de graphe (avec arcs parallèles)



- Tour d'Euler: chemin qui traverse chaque arc une fois exactement et qui retourne au premier sommet
- Théorème d'Euler: un graphe a un tour d'Euler si et seulement si tous les sommets ont un degré pair.
- Trouvez-vous intéressantes de telles idées?
- Aimeriez-vous passer une session entière à faire de telles preuves...? Il existe un tel cours!

Le TAD Graphe (*Graph*)

- Le **TAD Graphe** est un contenant positionnel dont les positions sont les sommets et les arcs du graphe.

  - size() Retourne le nombre de sommets plus le nombre d'arcs contenus dans *G*.
  - isEmpty()
  - elements()
  - positions()
  - swap()
  - replaceElement()

Notation: Graphe *G*; Sommets *v*, *w*; Arc *e*; Objet *o*

- numVertices()
  Retourne le nombre de sommets de *G*.
- numEdges()
  Retourne le nombre d'arcs de *G*.
- vertices() Retourne une énumération des sommets de *G*.
- edges() Retourne une énumération des arcs de *G*.

Le TAD Graphe (suite)

- directedEdges()
  Retourne une énumération de tous les arcs orientés de *G*.
- undirectedEdges()
  Retourne une énumération de tous les arcs non-orientés de *G*.
- incidentEdges(*v*)
  Retourne une énumération de tous les arcs attachés à *v*.
- inIncidentEdges(*v*)
  Retourne une énumération de tous les arcs entrant dans *v*.
- outIncidentEdges(*v*)
  Retourne une énumération de tous les arcs sortant de *v*.
- opposite(*v*, *e*)
  Retourne le sommet de l'arc *e* qui n'est pas *v*.
- degree(*v*)
  Retourne le degré de *v*.
- inDegree(*v*)
  Retourne le degré d'entrée de *v*.
- outDegree(*v*)
  Retourne le degré de sortie de *v*.

Encore des méthodes...

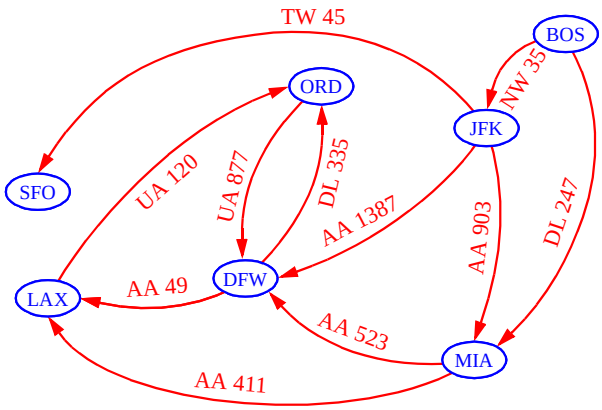
- adjacentVertices(*v*)
  Retourne une énumération des sommets adjacents à *v*.
- inAdjacentVertices(*v*)
  Retourne une énumération des sommets adjacents à *v* qui ont un arc entrant dans *v*.
- outAdjacentVertices(*v*)
  Retourne une énumération des sommets adjacents à *v* qui ont un arc sortant de *v*.
- areAdjacent(*v*,*w*)
  Indique si les sommets *v* et *w* sont adjacents.
- endVertices(*e*)
  Retourne un vecteur de taille 2 emmagasinant les sommets aux bouts de *e*.
- origin(*e*)
  Retourne le sommet duquel *e* sort.
- destination(*e*)
  Retourne le sommet auquel *e* entre.
- isDirected(*e*)
  Retourne vrai ssi *e* est orienté.

Méthodes de mise à jour

- makeUndirected(*e*)
  Déclare *e* comme arc non-orienté.
- reverseDirection(*e*)
  Inverse les sommets d'origine et de destination de *e*.
- setDirectionFrom(*e*, *v*)
  Ajuste la direction de *e* de façon à sortir de *v*, l'un de ses sommets.
- setDirectionTo(*e*, *v*)
  Ajuste la direction de *e* de façon à entrer dans *v*, l'un de ses sommets.
- insertEdge(*v*, *w*, *o*)
  Insère et retourne un arc non-orienté entre *v* et *w*, tout en emmagasinant *o* à cette position.
- insertDirectedEdge(*v*, *w*, *o*)
  Insère et retourne un arc orienté entre *v* et *w*, tout en emmagasinant *o* à cette position.
- insertVertex(*o*)
  Insère et retourne un nouveau sommet (isolé) emmagasinant *o* à cette position
- removeEdge(*e*)
  Retire l'arc *e*.

Structures de données pour graphes

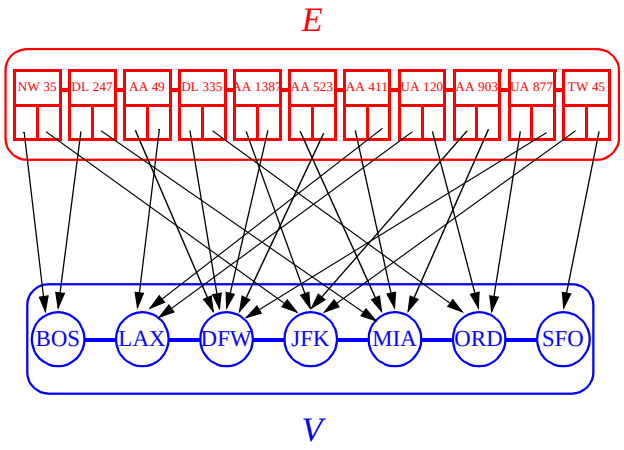
- Un graphe! Comment le représenter?
- Pour débuter, nous conservons les sommets et les arcs dans deux contenants, et chaque objet arc a des références vers les sommets qu'il relie.



- Des structures additionnelles peuvent être utilisées afin de mieux exécuter les méthodes de Graphe.

Liste d'arcs (*Edge List*)

- La structure **liste d'arcs** emmagasine tout simplement les sommets et les arcs dans des séquences non-triées.
- Facile à réaliser.
- Trouver l'arc attaché à un sommet donné n'est pas efficace parce que cela exige le parcours de la séquence d'arcs toute entière.

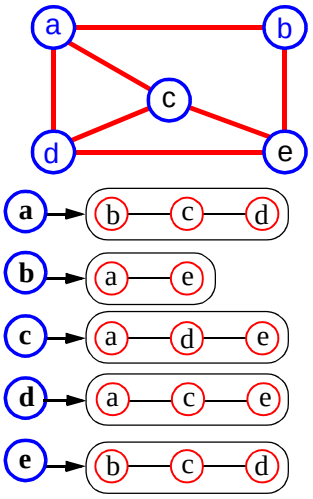


Performance de la structure Liste d'arcs

Opération	Temps
size, isEmpty, replaceElement, swap	*O(1)*
numVertices, numEdges	*O(1)*
vertices	*O(n)*
edges, directedEdges, undirectedEdges	*O(m)*
elements, positions	*O(n+m)*
endVertices, opposite, origin, destination, isDirected	*O(1)*
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices, areAdjacent, degree, inDegree, outDegree	*O(m)*
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	*O(1)*
removeVertex	*O(m)*

Liste d'adjacence (traditionnelle)

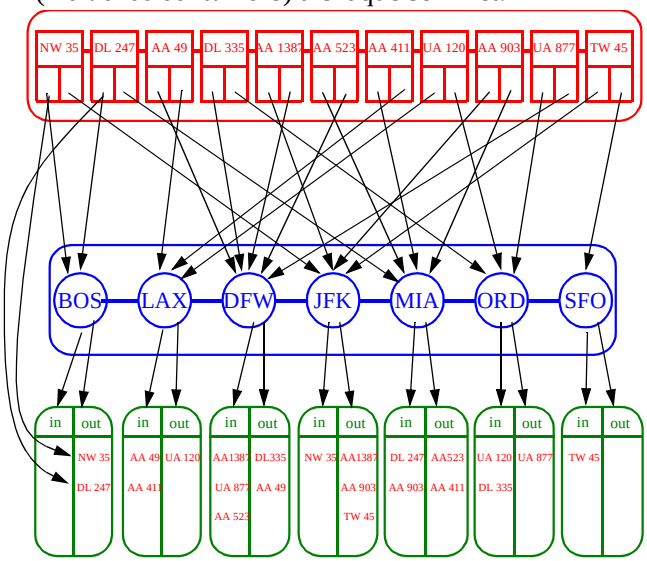
- Liste d'adjacence d'un sommet v:
  séquence de sommets adjacents à v
- Représentez le graphe par les listes d'adjacence de tous les sommets



- Espace requis = $\Theta(N + \sum \deg(v)) = \Theta(N + M)$

Liste d'adjacence (moderne)

- La structure **liste d'adjacence** améliore la structure de liste d'arcs en ajoutant des contenants de liaison (*incidence containers*) à chaque sommet.

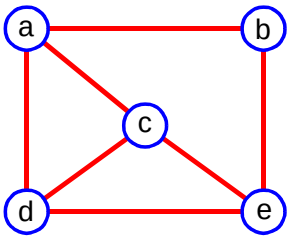


- L'espace requis est *O(n + m)*.

Performance de la structure Liste d'adjacence

Opération	Temps
size, isEmpty, replaceElement, swap	*O(1)*
numVertices, numEdges	*O(1)*
vertices	*O(n)*
edges, directedEdges, undirectedEdges	*O(m)*
elements, positions	*O(n+m)*
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	*O(1)*
incidentEdges(v), inIncidentEdges(v), outIncidentEdges(v), adjacentVertices(v), inAdjacentVertices(v), outAdjacentVertices(v)	*O*(deg(*v*))
areAdjacent(u, v)	*O*(min(deg(*u*), deg(*v*)))
insertVertex, insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection,	*O(1)*
removeVertex(v)	*O*(deg(*v*))

Matrice d'adjacence (traditionnelle)



	a	b	c	d	e
a	F	T	T	T	F
b	T	F	F	F	T
c	T	F	F	T	T
d	T	F	T	F	T
e	F	T	T	T	F

- Matrice M avec entrées pour toutes les paires de sommets
- M[i,j] = vrai signifie qu'il y a un arc (i,j) dans le graphe.
- M[i,j] = faux signifie qu'il n'y a aucun arc (i,j) dans le graphe.
- Il y a une entrée pour chaque arc possible, donc:
  espace requis = $\Theta(N^2)$

Matrice d'adjacence (moderne)

- Les structures à matrice d'adjacence ajoutent à la structure liste d'arcs une matrice où chaque rangée et colonne correspond à un sommet.

	0	1	2	3	4	5	6
0	Ø	Ø	NW 35	Ø	DL 247	Ø	Ø
1	Ø	Ø	Ø	AA 49	Ø	DL 335	Ø
2	Ø	AA 1387	Ø	Ø	AA 903	Ø	TW 45
3	Ø	Ø	Ø	Ø	Ø	UA 120	Ø
4	Ø	AA 523	Ø	AA 411	Ø	Ø	Ø
5	Ø	UA 877	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø

BOS 0 DFW 1 JFK 2 LAX 3 MIA 4 ORD 5 SFO 6

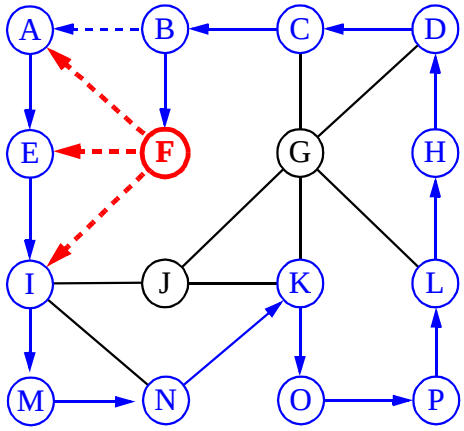
- L'espace requis est $O(n^2 + m)$

Performance de la structure Matrice d'adjacence

Opération	Temps
size, isEmpty, replaceElement, swap	$O(1)$
numVertices, numEdges	$O(1)$
vertices	$O(n)$
edges, directedEdges, undirectedEdges	$O(m)$
elements, positions	$O(n+m)$
endVertices, opposite, origin, destination, isDirected, degree, inDegree, outDegree	$O(1)$
incidentEdges, inIncidentEdges, outIncidentEdges, adjacentVertices, inAdjacentVertices, outAdjacentVertices,	$O(n)$
areAdjacent	$O(1)$
insertEdge, insertDirectedEdge, removeEdge, makeUndirected, reverseDirection, setDirectionFrom, setDirectionTo	$O(1)$
insertVertex, removeVertex	$O(n^2)$

TRAVERSÉES DE GRAPHES

- En profondeur (*Depth-First Search*)
- En largeur (*Breadth-First Search*)
- Patron de conception: méthode du gabarit (*Template Method Pattern*)



Explorer un labyrinthe sans se perdre

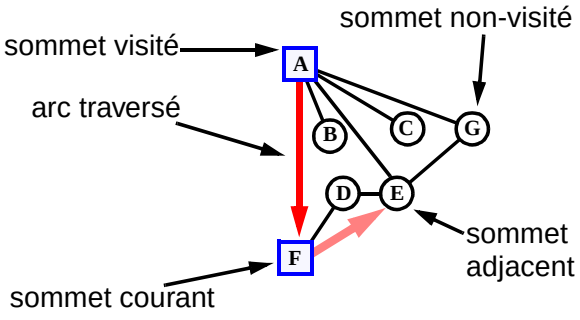
- Une **recherche en profondeur (***depth-first search* ou **DFS)** dans un graphe non-orienté G, c'est comme vagabonder dans un labyrinthe avec une corde et une cannette de peinture rouge, sans se perdre.
- Nous partons d'un sommet *s*, en attachant un bout de notre corde à ce point et en peinturant "visité" sur *s*. Ensuite, nous étiquetons *s* comme étant notre sommet courant appelé *u*.
- Maintenant, nous allons vers un arc arbitraire (*u*,*v*).
- Si l'arc (*u*,*v*) nous mène à un sommet *v* déjà visité, alors nous retournons à *u*.
- Si le sommet *v* n'a pas été visité, alors nous déroulons notre corde en allant à *v*, peinturons "visité" sur *v*, étiquetons *v* comme notre sommet courant, et répétons les étapes précédentes.
- Éventuellement, nous seront au point où tous les arcs attachés à *u* mènent à des sommets visités. Alors, nous revenons sur nos pas en déroulant la corde vers un sommet déjà visité *v*. Ainsi *v* devient notre sommet courant et nous répétons les étapes précédentes.

Explorer un labyrinthe sans se perdre (suite)

- Si tous les arcs attachés à v mènent à des sommets visités, alors nous revenons sur nos pas comme nous l'avons fait précédemment. Nous continuons à revenir sur nos pas en trouvant et en explorant les arcs inexplorés, et en répétant la procédure.
- Quand nous retournons au sommet *s* et qu'il n'y a plus d'arc inexploré attaché à ce point, alors nous avons terminé notre recherche DFS.

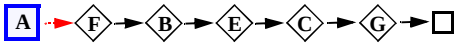
Recherche en profondeur DFS

Algorithme DFS(*v*);
Entrée: un sommet *v* dans un graphe
Sortie: un étiquetage des arcs comme étant découverts (*discovery edges*) ou arrières (*backedges*)
for chaque arc *e* attaché à *v* **do**
if l'arc *e* est inexploré **then**
soit *w* l'autre extrémité de *e*
if le sommet *w* est inexploré **then**
étiqueter *e* comme arc de découverte
appeler récursivement **DFS**(*w*)
else
étiqueter *e* comme arc arrière

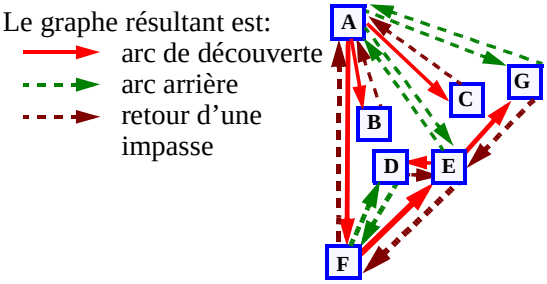


Déterminer les arcs attachés

- DFS dépend de la façon dont ces arcs sont obtenus.
- Si nous commençons à A et examinons l'arc vers F, ensuite vers B, E, C, et enfin G:



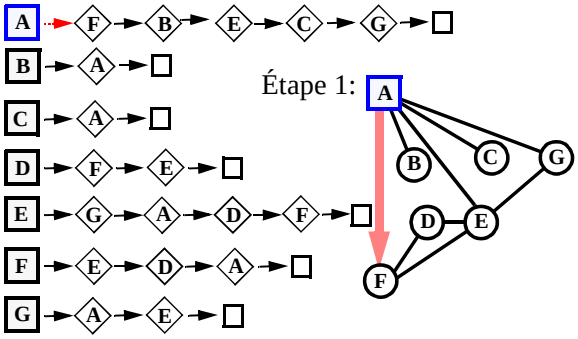
Le graphe résultant est:

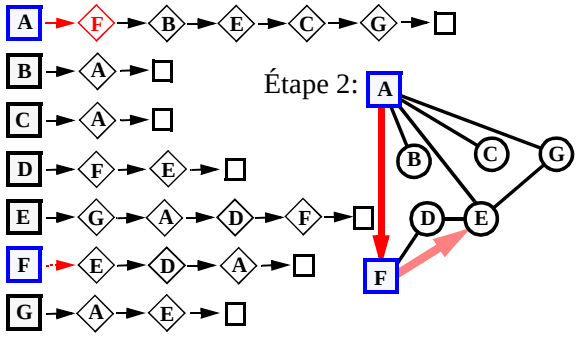


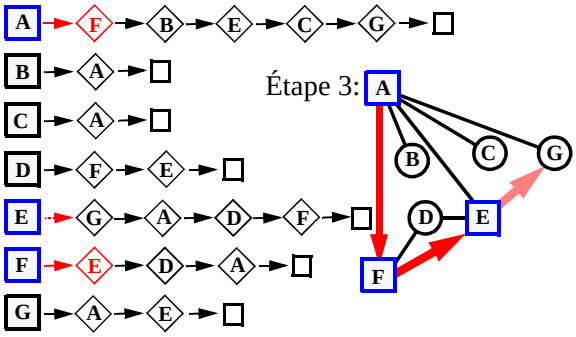
Si maintenant nous examinons l'arbre en commençant par A et ensuite G, C, E, B, et enfin F.

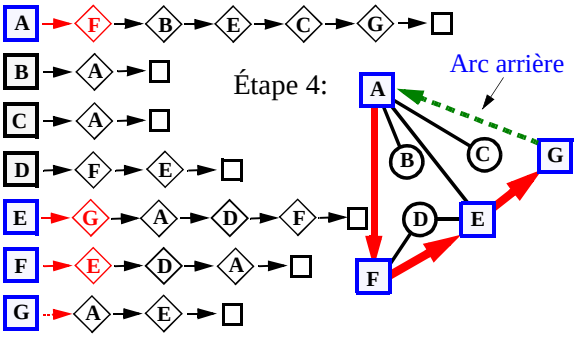
l'ensemble final d'arcs arrières et de découverte, de même que les points de retour, sont différents.

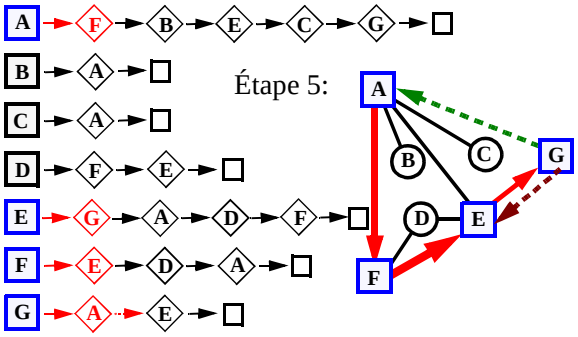
- Passons maintenant à un exemple de DFS.

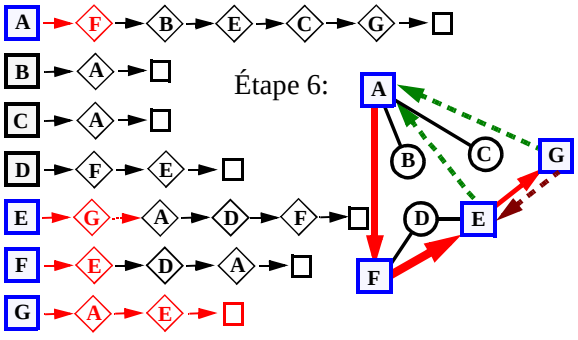


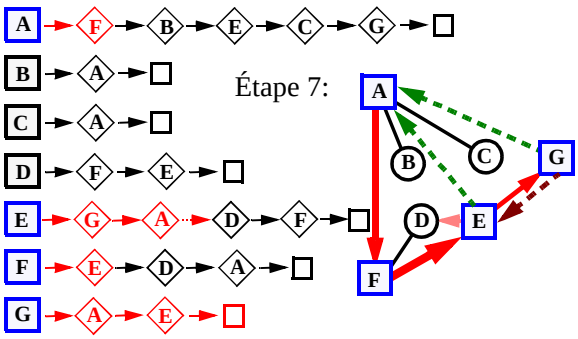




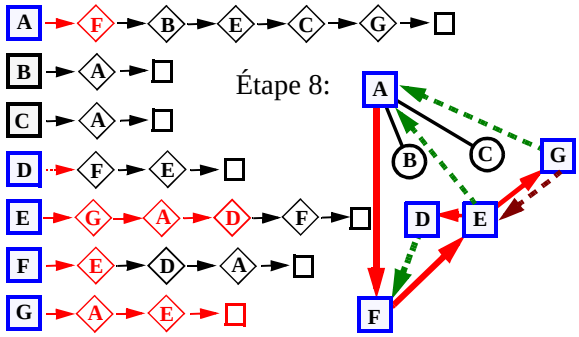




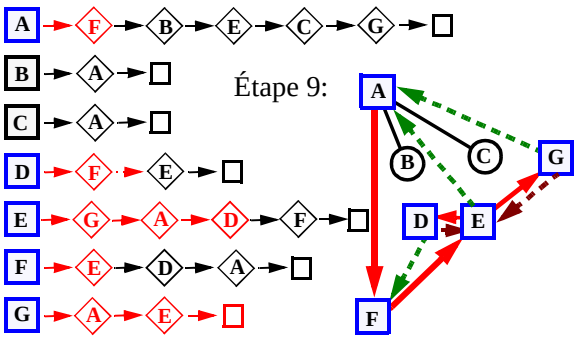




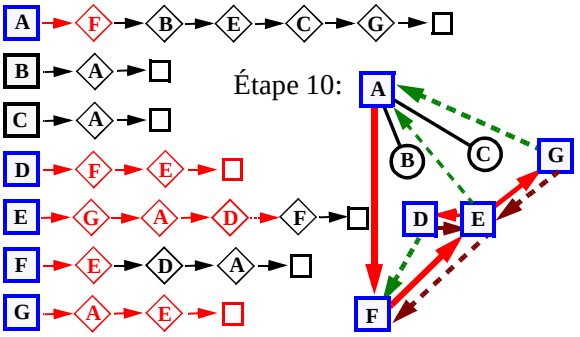
Étape 7:



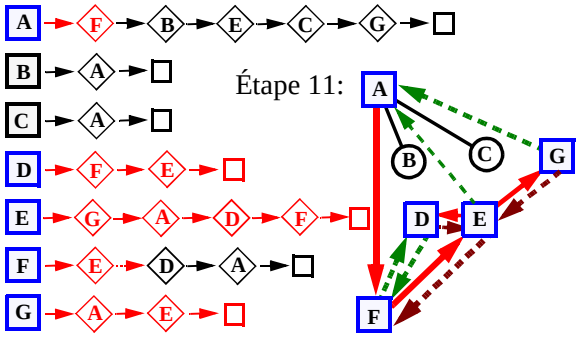
Étape 8:



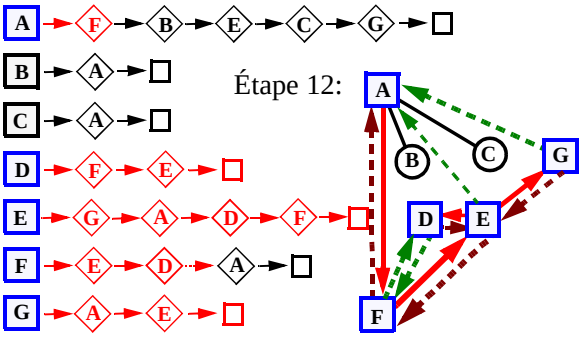
Étape 9:



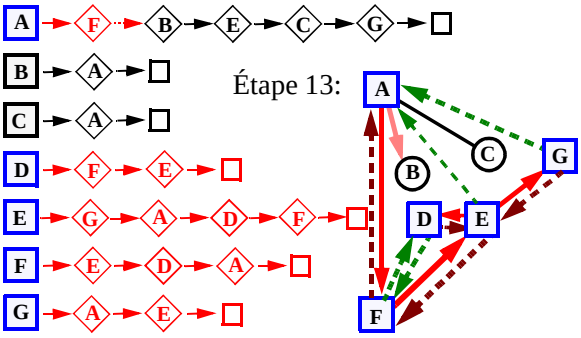
Étape 10:



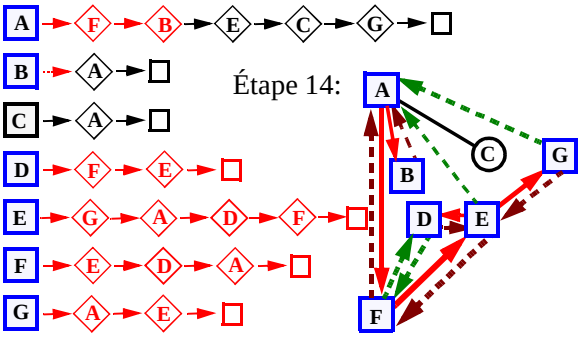
Étape 11:



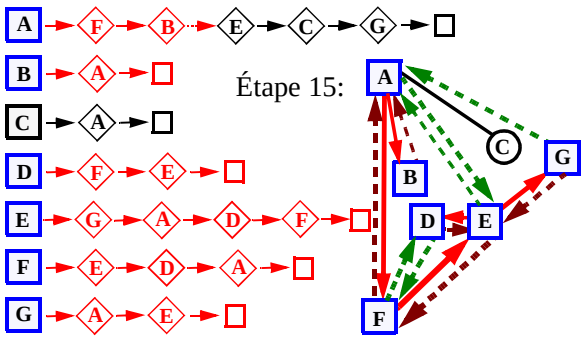
Étape 12:

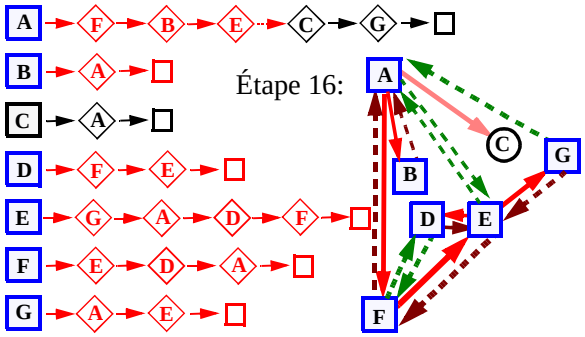


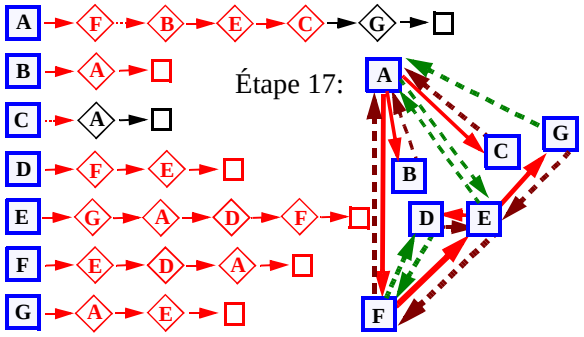
Étape 13:

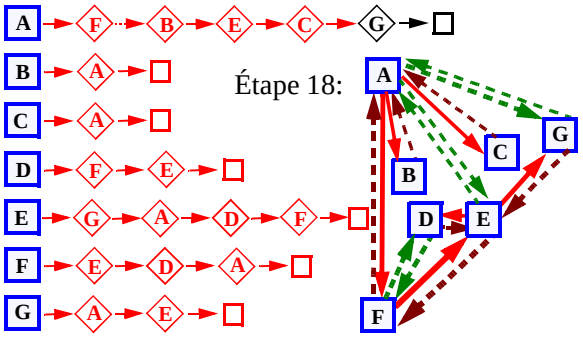


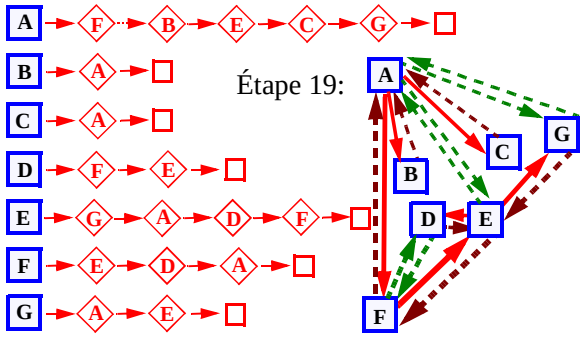
Étape 14:











Et c'est tout!

Propriétés de DFS

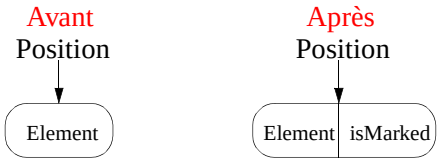
- Proposition 9.12 : Soit G un graphe non-orienté sur lequel une traversée DFS commençant au sommet *s* a été faite. Alors:
  1) La traversée visite tous les sommets dans la composante connexe de *s*
  2) Les arcs de découverte forment un arbre recouvrant de la composante connexe de *s*
- Justification de 1):
  - Essayons une contradiction: supposons qu'il y ait au moins un sommet *v* non-visité et soit *w* le premier sommet non-visité sur un chemin de *s* à *v*.
  - Comme *w* est le premier sommet non-visité sur le chemin, il y a un voisin *u* qui a été visité.
  - Mais quand nous avons visité *u* nous devons avoir observé l'arc (*u*, *w*). Donc *w* doit avoir été visité.
- Justification de 2):
  - Nous étiquetons seulement les arcs à partir du moment où nous allons vers des sommets non-visités. Ainsi, nous ne formons jamais de cycle d'arcs de découverte; ces arcs forment un arbre.
  - C'est un arbre recouvrant car DFS visite chaque sommet dans la composante connexe de *s*.

Analyse du temps d'exécution

- Souvenez-vous:
  - DFS sur chaque sommet une fois exactement.
  - Chaque arc est examiné exactement deux fois, une fois pour chacun de ses sommets.
- Pour n_s sommets et m_s arcs dans la composante connexe du sommet *s*, une DFS commençant à *s* s'exécute en un temps $O(n_s + m_s)$ si:
  - Le graphe est représenté dans une structure de données, comme une liste d'adjacence, où les méthodes pour les sommets et les arcs s'exécutent en un temps constant;
  - Étiqueter un sommet comme étant exploré et tester si un sommet a été exploré prend O(degré);
  - En étiquetant les nœuds visités, nous pouvons systématiquement considérer les arcs attachés au sommet courant, de façon à ne pas examiner le même arc plus d'une fois.

Étiquetage des sommets

- Étudions les façons d'étiqueter les sommets de façon à satisfaire les conditions mentionnées à la page précédente.
- Extension des positions de sommet pour inclure une variable servant à l'étiquetage.



- Utilisation d'un mécanisme de table de hachage qui satisfait ces conditions dans un sens probabiliste, parce qu'un tel mécanisme supporte les opérations d'étiquetage et de test en un temps attendu $O(1)$.

Patron de conception: méthode du gabarit (*Template Method Pattern*)

- le patron de conception **méthode du gabarit** offre un ***mécanisme de calcul générique*** qui peut être spécialisé en redéfinissant certaines étapes
- pour l'appliquer, nous concevons une classe qui:
  - réalise le ***squelette*** d'un algorithme
  - invoque des méthodes auxiliaires qui peuvent être redéfinies par ses sous-classes afin de faire des calculs utiles
- **Bénéfices**
  - fait que la rectitude des calculs spécialisés dépend de celle de l'algorithme squelette
  - démontre la puissance de l'héritage
  - promeut la réutilisation de code
  - encourage le développement de code générique
- **Exemples**
  - ***traversée générique d'un arbre binaire*** (qui inclut pré-ordre, in-ordre, et post-ordre) et ses applications
  - ***recherche en profondeur générique d'un graphe non-orienté*** et ses applications

Recherche en profondeur générique

```
public abstract class DFS {
 protected Object dfsVisit(Vertex v) {
   protected InspectableGraph graph;
   protected Object visitResult;
   initResult();
   startVisit(v);
   mark(v);
   for (Enumeration inEdges = graph.incidentEdges(v);
        inEdges.hasMoreElements();) {
    Edge nextEdge = (Edge) inEdges.nextElement();
    if (!isMarked(nextEdge)) { // found an unexplored edge
     mark(nextEdge);
     Vertex w = graph.opposite(v, nextEdge);
     if (!isMarked(w)) { // discovery edge
      mark(nextEdge);
      traverseDiscovery(nextEdge, v);
      if (!isDone())
       visitResult = dfsVisit(w); }
     else  // back edge
      traverseBack(nextEdge, v);
    }
   }
   finishVisit(v);
   return result();
  }
```

Méthodes auxiliaires de recherche DFS générique

```
public Object execute(InspectableGraph g, Vertex start,
                      Object info) {
  graph = g;
  return null;
}

protected void initResult() {}

protected void startVisit(Vertex v) {}

 protected void traverseDiscovery(Edge e, Vertex from) {}

protected void traverseBack(Edge e, Vertex from) {}

protected boolean isDone() { return false; }

protected void finishVisit(Vertex v) {}

protected Object result() { return new Object(); }
```


Observons maintenant 4 façons de spécialiser DFS générique!

- la classe FindPath spécialise DFS afin de retourner un chemin du sommet start vers le sommet target.

```
public class FindPathDFS extends DFS {
  protected Sequence path;
  protected boolean done;
  protected Vertex target;
  public Object execute(InspectableGraph g, Vertex start,
                        Object info) {
    super.execute(g, start, info);
    path = new NodeSequence();
    done = false;
    target = (Vertex) info;
    dfsVisit(start);
    return path.elements();
  }
  protected void startVisit(Vertex v) {
    path.insertFirst(v);
    if (v == target) { done = true; }
  }
  protected void finishVisit(Vertex v) {
    if (!done) path.remove(path.first());
  }
  protected boolean isDone() { return done; }
```


Autre spécialisation de DFS générique...

- FindAllVertices spécialise DFS afin de retourner une énumération des sommets dans la composante connexe contenant le sommet start.

```
public class FindAllVerticesDFS extends DFS {
  protected Sequence vertices;
  public Object execute(InspectableGraph g, Vertex start,
                        Object info) {
    super.execute(g, start, info);
    vertices = new NodeSequence();
    dfsVisit(start);
    return vertices.elements();
  }

  public void startVisit(Vertex v) {
    vertices.insertLast(v);
  }
}
```


Plus de spécialisations de DFS générique...

- ConnectivityTest utilise une spécialisation de DFS pour déterminer si un graphe est connecté.

```
public class ConnectivityTest {
  protected static DFS tester=new FindAllVerticesDFS();
  public static boolean isConnected(InspectableGraph g)
  {
    if (g.numVertices() == 0) return true; //empty is
                                           //connected
    Vertex start = (Vertex)g.vertices().nextElement();
    Enumeration compVerts =
          (Enumeration)tester.execute(g, start, null);
    // count how many elements are in the enumeration
    int count = 0;
    while (compVerts.hasMoreElements()) {
      compVerts.nextElement();
      count++;
    }
    if (count == g.numVertices()) return true;
    return false;
  }
}
```

Et une autre spécialisation de DFS générique!

- FindCycle spécialise DFS afin de déterminer si la composante connexe du sommet start contient un cycle, et alors le retourne.

```
public class FindCycleDFS extends DFS {
  protected Sequence path;
  protected boolean done;
  protected Vertex cycleStart;
  public Object execute(InspectableGraph g, Vertex start,
                        Object info) {
    super.execute(g, start, info);
    path = new NodeSequence();
    done = false;
    dfsVisit(start);
    //copy the vertices up to cycleStart from the path to
    //the cycle sequence.
    Sequence theCycle = new NodeSequence();
    Enumeration pathVerts = path.elements();
```

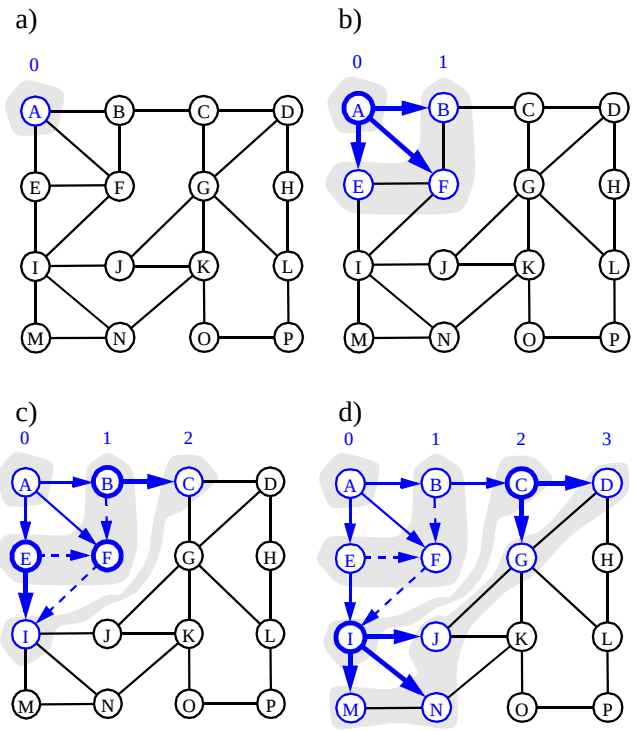


```
    while (pathVerts.hasMoreElements()) {
      Vertex v = (Vertex)pathVerts.nextElement();
      theCycle.insertFirst(v);
      if ( v == cycleStart) {
        break;
      }
    }
    return theCycle.elements();
  }
  protected void startVisit(Vertex v) {path.insertFirst(v);}
  protected void finishVisit(Vertex v) {
    if (done) {path.remove(path.first());}
  }
  //When a back edge is found, the graph has a cycle
  protected void traverseBack(Edge e, Vertex from) {
    Enumeration pathVerts = path.elements();
    cycleStart = graph.opposite(from, e);
    done = true;
  }
  protected boolean isDone() {return done;}
}
```


Recherche en largeur BFS (*Breadth-First Search*)

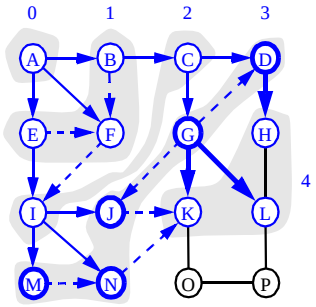
- Comme DFS, une recherche en largeur (BFS) traverse une composante connexe d'un graphe, et ce faisant définit un arbre recouvrant qui a quelques propriétés utiles
  - Le sommet de départ *s* a un niveau 0; comme dans DFS, définissons ce point comme point d'ancrage.
  - Au premier tour, la corde est déroulée de la longueur d'un arc, et tous les arcs à une distance d'un arc du point d'ancrage sont visités.
  - Ces arcs sont placés dans le niveau 1.
  - Au second tour, tous les nouveaux arcs qui peuvent être atteints en déroulant la corde d'une longueur de 2 arcs sont visités et placés dans le niveau 2.
  - Ceci se poursuit jusqu'à ce que tous les sommets aient été placés dans un niveau.
  - L'étiquette de tout sommet *v* correspond à la longueur du plus court chemin de *s* à *v*.

BFS - Une Représentation graphique

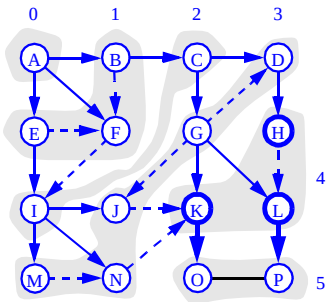


Encore BFS

e)



f)



Pseudo-code BFS

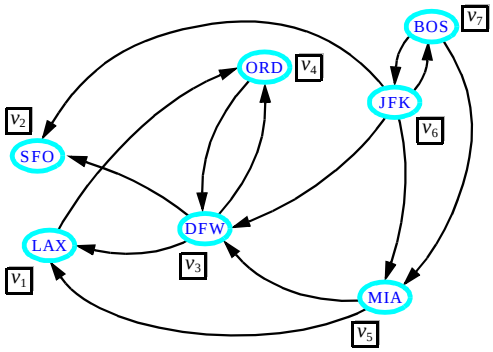
Algorithme BFS(s):
  Entrée: Un sommet s dans un graphe
  Sortie: Un étiquetage des arcs comme étant découverts (*discovery edges*) ou traversés (*cross edges*)
  initialiser le contenant L_0 avec le sommet s
  $i \leftarrow 0$
  while L_i n'est pas vide do
    créer le contenant L_{i+1} initialement vide
    for chaque sommet v dans L_i do
      if l'arc e est attaché à v do
        soit w l'autre extrémité de e
        if le sommet w est inexploré then
          étiqueter e comme arc de découverte
          insérer w dans L_{i+1}
        else
          étiqueter e comme arc traversé
  $i \leftarrow i + 1$

Propriétés de BFS

- Proposition: Soit G un graphe non-orienté sur lequel une traversée BFS débutant au sommet s a été faite. Alors:
  - La traversée visite tous les sommets dans la composante connexe de s.
  - Les arcs de découverte forment un arbre recouvrant T, que nous appelons arbre BFS, de composante connexe de s.
  - Pour chaque sommet v au niveau i, le chemin de l'arbre BFS T entre s et v a i arcs, et tout autre chemin de G entre s et v a au moins i arcs.
  - Si (u, v) est un arc qui n'est pas dans l'arbre BFS, alors les niveaux de u et v diffèrent de 1 au plus.
- Proposition: Soit G un graphe avec n sommets et m arcs. Une traversée BFS de G a un temps $O(n + m)$. Aussi, il existe des algorithmes au temps $O(n + m)$ basés sur BFS pour les problèmes suivants:
  - Tester si G est connexe
  - Calculer l'arbre recouvrant de G
  - Calculer les composantes connexes de G
  - Calculer, pour chaque sommet v de G, le nombre minimum d'arcs de tout chemin entre s et v.

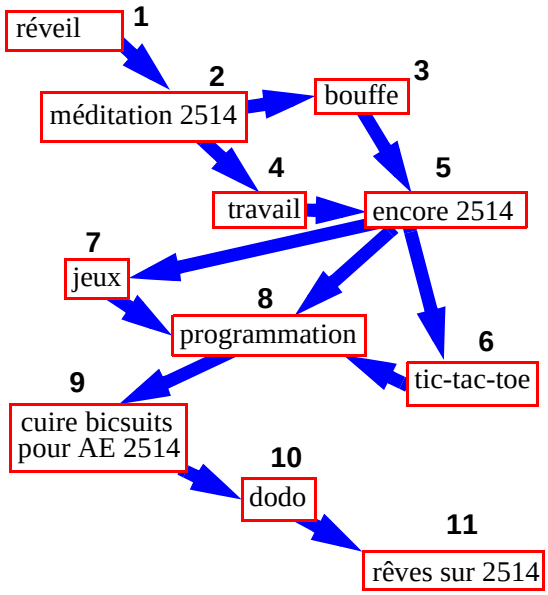
DIGRAPHES

- Accessibilité (*reachability*)
- Connectivité
- Fermeture transitive (*closure*)
- Algorithme de Floyd-Warshall



DIGRAPHES

une journée typique...

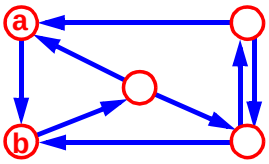


Qu'est-ce qu'un digraphe?

Un graphe orienté (de l'anglais *directed graph*)!

Chaque arc va dans une direction

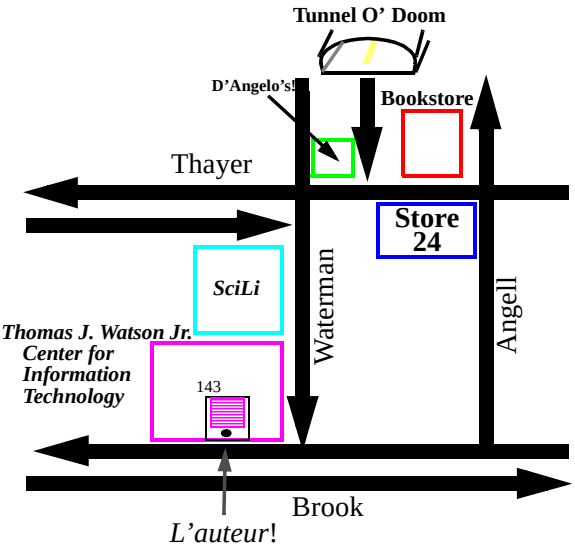
L'arc (a,b) va de a à b, mais pas de b à a



Vous dites sûrement: "Ouin, et si nous avions un exemple qui démontrerait combien nous pourrions être éclairés par l'utilisation de digraphes?!! – Et bien, si vous insistez. . .

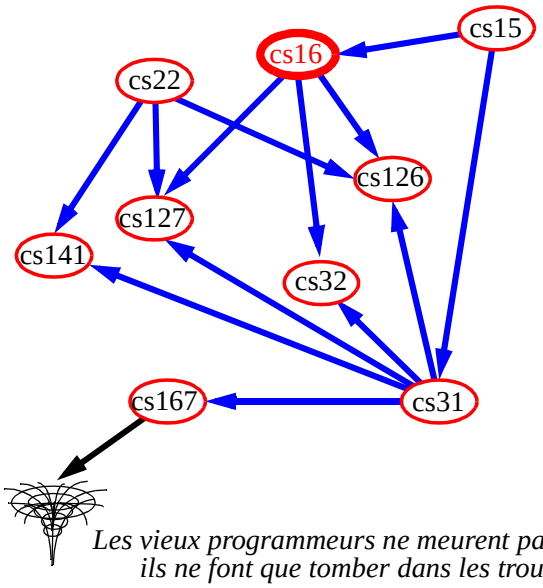
Applications

Cartes: les digraphes peuvent représenter les rues à sens unique
(utiles dans les grands centres-villes)



Une autre application

Planification d'horaires: l'arc (a,b) indique que la tâche a doit être complétée avant que b ne démarre.



Les vieux programmeurs ne meurent pas—
ils ne font que tomber dans les trous noirs

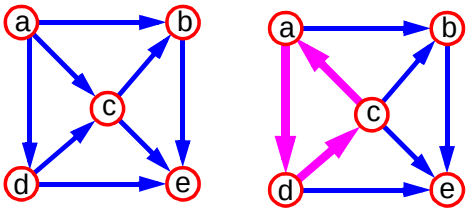
Les GOA!

GOA: Graphe Orienté Acyclique

(de l'anglais *directed acyclic graph — DAG*)

Pardon?!!

C'est un graphe orienté sans cycles orientés

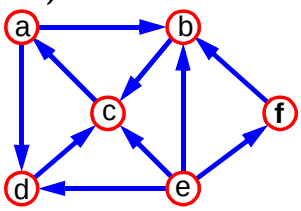


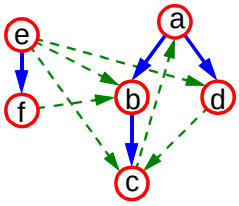
GOA pas un GOA

Recherche en profondeur

Même algorithme que pour les graphes non-orientés

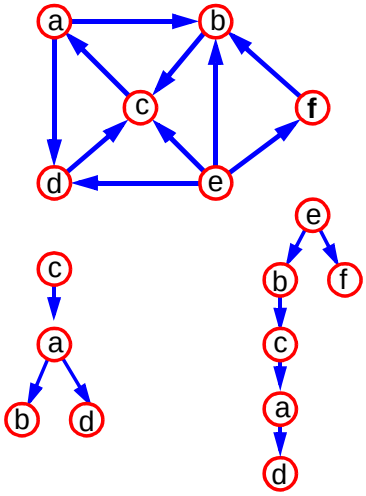
Sur un digraphe connexe, nous pouvons obtenir des arbres DFS non-connexes (c'est-à-dire, une forêt DFS)





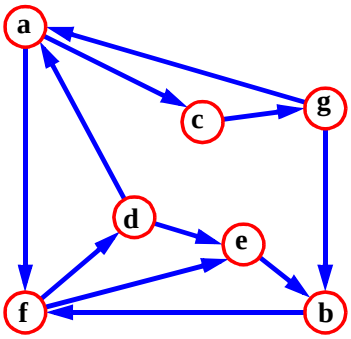
Accessibilité (*reachability*)

Arbre DFS avec racine v: sommets accessibles à partir de v via les chemins orientés

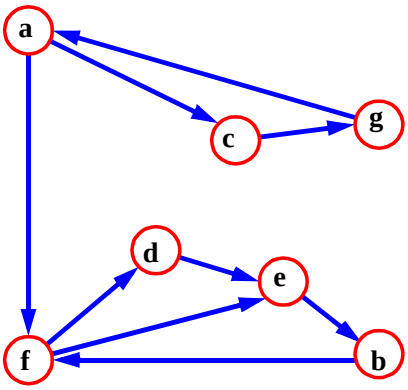


Digraphes fortement connexes

Chaque sommet peut atteindre tous les autres sommets



Composantes fortement connexes



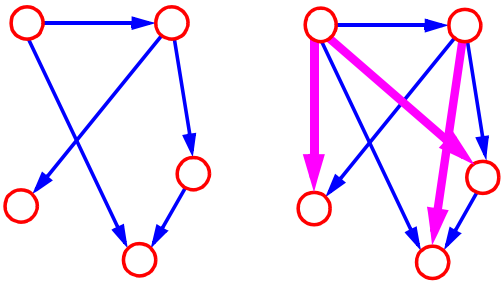
{ a , c , g }

{ f , d , e , b }

Fermeture transitive

Le digraphe G^* est obtenu de G en utilisant la règle:

Si il existe un chemin orienté dans **G** de a à b, alors ajouter l'arc (a,b) à $\mathbf{G^*}$

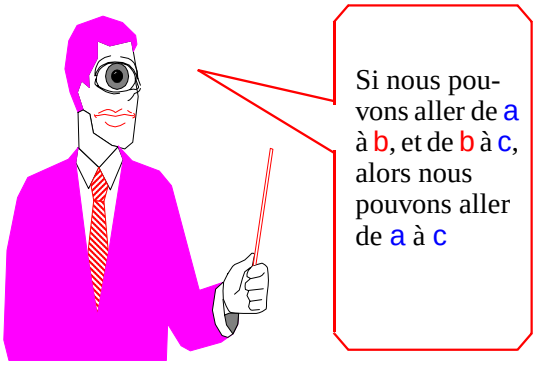


G **G^***

Calculer la fermeture transitive

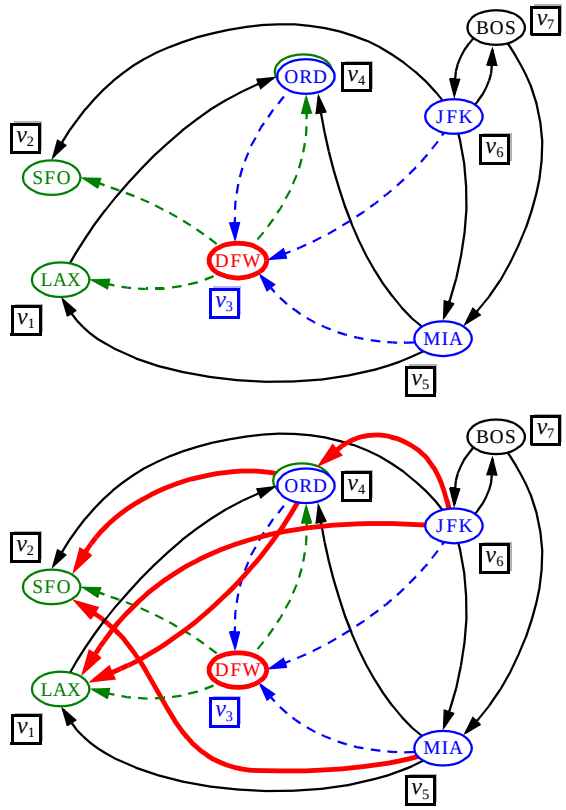
Nous pouvons utiliser DFS sur chaque sommet
Temps: $O(n(n+m))$

Ou encore... utiliser l'algorithme de Floyd-Warshall:



"Pink" Floyd

Exemple



Algorithme de Floyd-Warshall

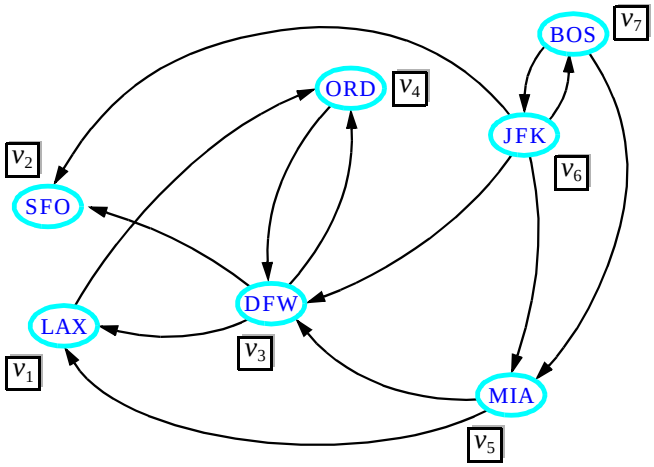
- Cet algorithme présuppose que les méthodes areAdjacent et insertDirectedEdge prennent un temps $O(1)$ (par exemple, structure en matrice d'adjacence)

Algorithme FloydWarshall(G)
  soit v_1 ... v_n un ordre arbitraire des sommets
  $G_0 = G$
  for k = 1 **to** n **do**
    // considérez tous les sommets de routage
    // possibles v_k
    $G_k = G_{k-1}$ // ce sont les seuls à conserver
    for each (i, j = 1, ..., n) (i != j) (i, j != k) **do**
      // pour chaque paire de sommets v_i et v_j
      if G_{k-1}.areAdjacent(v_i,v_k) **and**
        G_{k-1}.areAdjacent(v_k,v_j) **then**
          G_k.insertDirectedEdge(v_i,v_j,null)
    return G_n

- Le digraphe G_k est le sous-digraphe de la fermeture transitive de G induit par les chemins avec sommets intermédiaires dans l'ensemble { v_1, ..., v_k }
- Temps d'exécution: $O(n^3)$

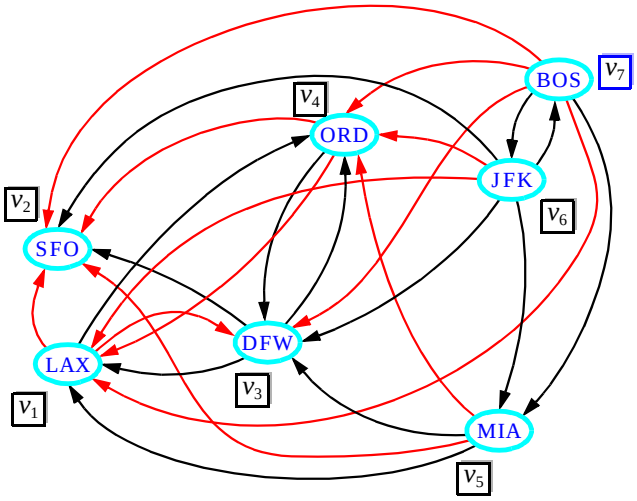
Exemple

- digraphe G



Exemple

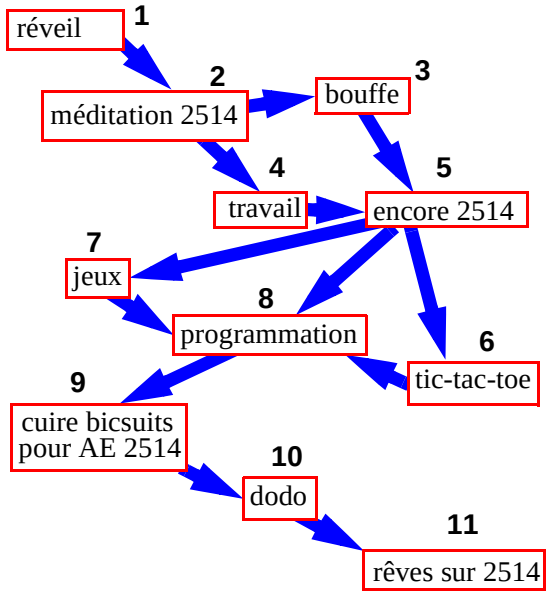
- digraphe G*



Tri topologique

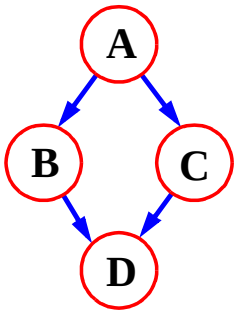
Pour chaque arc (u,v), le sommet u est visité avant le sommet v

une journée typique...



Tri topologique

Le résultat du tri topologique peut ne pas être unique



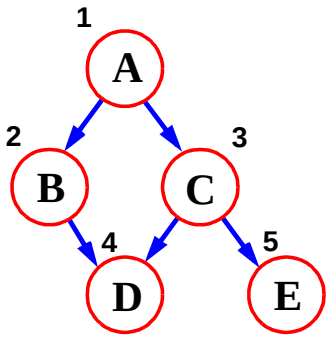
A B C D
ou
A C B D

*– **À vous** de décider*

Tri topologique

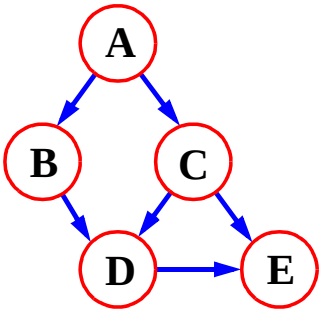
Les étiquettes augmentent le long d'un chemin orienté.

Un digraphe a un tri topologique *si et seulement si* il est acyclique (donc, un GOA)



Algorithme pour tri topologique

```
method TopologicalSort
  if il y a encore des sommets
      soit v une source;
          // un sommet sans arcs d'entrée
      étiqueter et supprimer v;
      TopologicalSort;
```



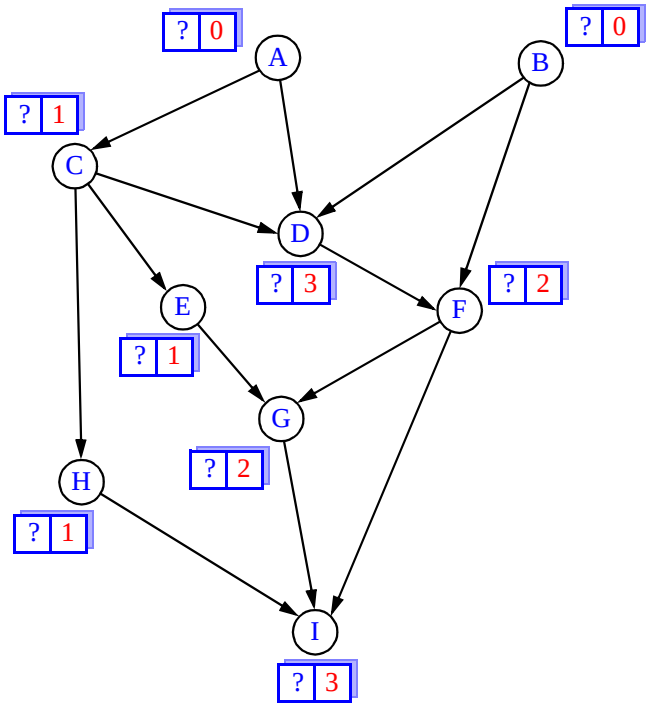
Algorithme (suite)

Simuler la suppression de sources en utilisant des compteurs de degré d'entrée

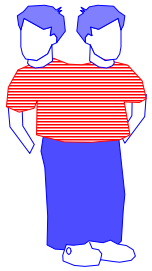
```
TopSort(Vertex v);
  étiqueter v;
  foreach arc(v,w)
        indeg(w) = indeg(w) – 1;
        if indeg(w) = 0
            TopSort(w);
```

1. Calculer indeg(**v**) pour tous les sommets
2. Foreach sommet **v** do
   if **v** non étiqueté et indeg(**v**) = 0
   then TopSort(**v**)

Exemple



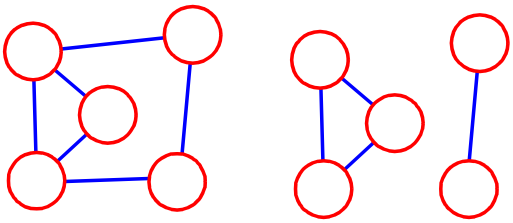
Connectivité et Biconnectivité



- composantes connexes
- sommets de séparation (*cutvertices*)
- composantes biconnexes

Composantes connexes

Graphe connexe: chaque paire de sommets reliée par un chemin.



connexe non-connexe

Composante connexe: sous-graphe connexe maximal d'un graphe

Relations d'équivalence

Une ***relation*** sur un ensemble S est un ensemble ordonné R composé de paires d'éléments de S et défini par une propriété quelconque.

Exemple:

- S = {1,2,3,4}
- R= {(i,j) ∈ S × S tel que i < j}
  = {(1,2),(1,3),(1,4),(2,3),(2,4),{3,4)}

Une ***relation d'équivalence*** satisfait les propriétés suivantes:

- (x,x) ∈ R, ∀ x ∈ S (***réflexive***)
- (x,y) ∈ R ⇒ (y,x) ∈ R (***symétrique***)
- (x,y), (y,z) ∈ R ⇒ (x,z) ∈ R (***transitive***)

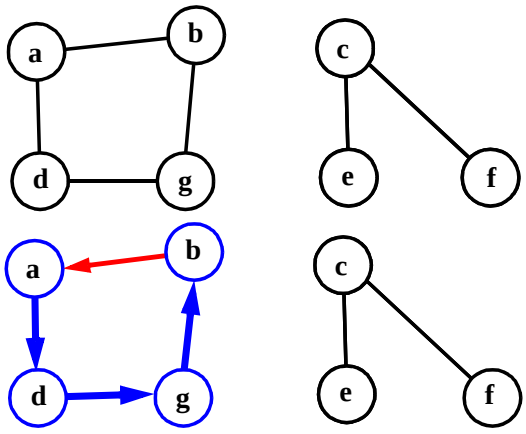
La relation C sur l'ensemble des sommets d'un graphe:

- (***u***,***v***) ∈ C ⇔ ***u*** et ***v*** sont dans la même composante connexe

est une relation d'équivalence.

DFS sur un graphe non-connexe

- DFS(*v*) visite tous les sommets et les arcs dans la composante connexe de *v.*



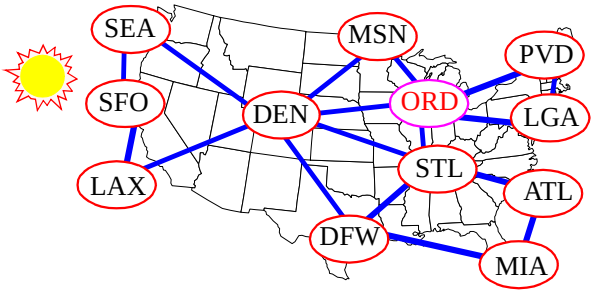
- Pour déterminer les composantes connexes:

```
k = 0 // compteur composante
foreach (vertex v)
    if unvisited(v)
    // ajouter à la composante k
    // les sommets atteints par v
    DFS(v, k++)
```

Sommets de séparation (*Cutvertices*)

Sommet de séparation (*cutvertex*):
son retrait rend le graphe non-connexe

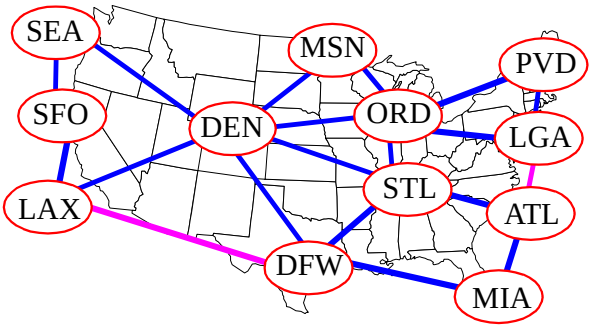
Si l'aéroport de Chicago est fermé, alors il n'y a aucun moyen d'aller dans les villes de la côte ouest à partir de Providence (PVD). Même chose pour l'aéroport de Denver.



- Sommets de séparation: ORD. DEN

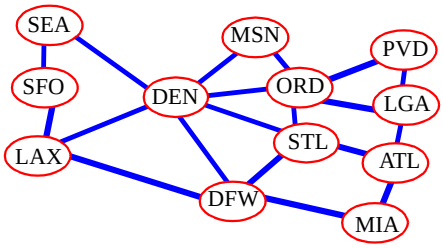
Biconnectivité

Graphe biconnexe: n'a pas de sommet de séparation.



Nouveaux vols:
LGA-ATL et DFW-LAX
rendent le graphe biconnexe.

Propriétés des graphes biconnexes

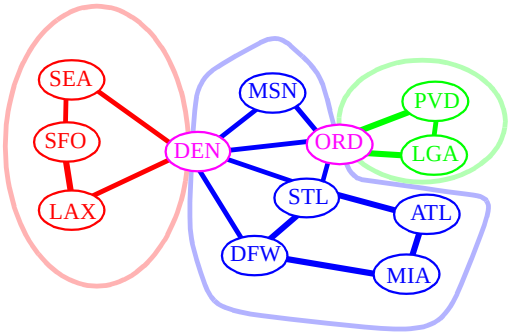


- Il y a ***deux chemins disjoint*** entre n'importe quelle paire de sommets.
- Il y a un ***cycle*** au travers de n'importe quelle paire de sommets.

Par convention, deux nœuds reliés par un arc forment un graphe biconnexe, mais ceci ne satisfait pas les propriétés mentionnées ci-haut.

Composantes biconnexes

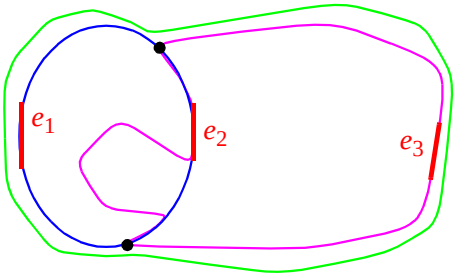
- Composante biconnexe (bloc): sous-graphe biconnexe maximal



- Les composantes biconnexes d'un graphe ne partagent pas d'arc, mais elles partagent des sommets de séparation.

Caractérisation des composantes biconnexes

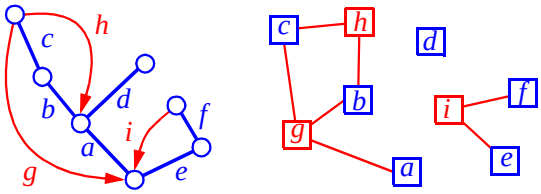
- ***Relation d'équivalence*** R sur les ***arcs*** de G: (e', e") ∈ R si il y a un cycle contenant à la fois e' et e"
- Preuve de la ***propriété transitive***



- Nous divisons les arcs de G en ***classes d'équivalence*** par rapport à R.
- Chaque classe d'équivalence correspond à:
  - une composante biconnexe de G
  - une composante connexe d'un graphe H dont les sommets sont les ***arcs*** de G et dont les arcs sont les ***paires*** dans la relation R.

DFS et composantes biconnexes

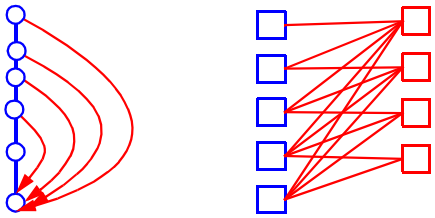
- Le graphe H a $O(m^2)$ arcs dans le pire des cas.
- Au lieu de calculer le graphe H tout entier, nous utilisons un graphe **mandataire** (*proxy*) K, qui est plus petit.
- Débutons avec un graphe K vide dont les sommets sont les arcs de G.
- Étant donné une DFS sur G, considérez les $(m - n + 1)$ cycles de G induits par les arcs.
- Pour chacun de ces cycles C = (e_0, e_1, ... , e_p) ajoutez les arcs (e_0, e_1) ... (e_0, e_p) à K.



- Les composantes connexes de K sont les mêmes que celles de H!

Un algorithme à temps linéaire

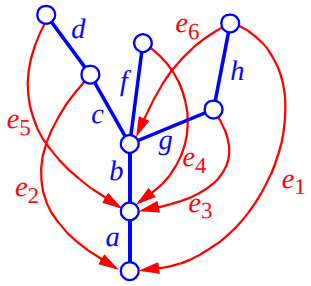
- La taille de K est $O(mn)$ dans le pire des cas.



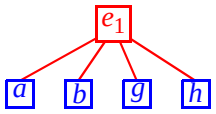
- Nous pouvons encore réduire la taille du graphe mandataire à $O(m)$
- Traitez les arcs selon une ***visite pré-ordre*** de leur sommet de destination dans l'arbre DFS
- Annotez les arcs de découverte formant les cycles
- Arrêter d'ajouter des arcs au graphe mandataire après avoir rencontré le premier arc annoté
- Le graphe mandataire résultant est une forêt!
- Cet algorithme requiert un temps $O(n+m)$.

Exemple

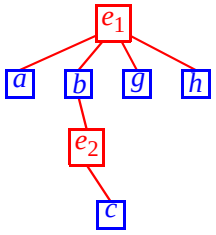
- Arcs arrières étiquetés selon la visite pré-ordre de leur sommet de destination dans l'arbre DFS



- Traitement de e_1

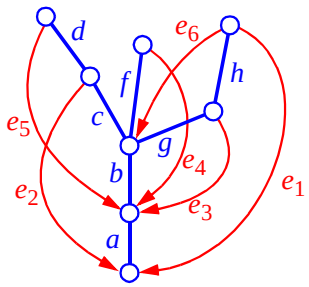


- Traitement de e_2

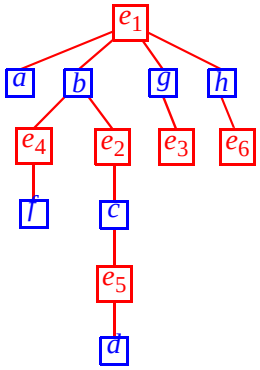


Exemple (suite)

- arbre DFS

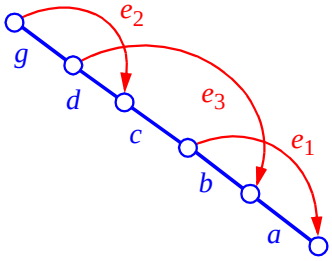


- graphe mandataire final (un arbre puisque le graphe est biconnexe)

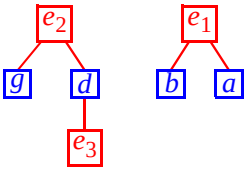


Pourquoi pré-ordre?

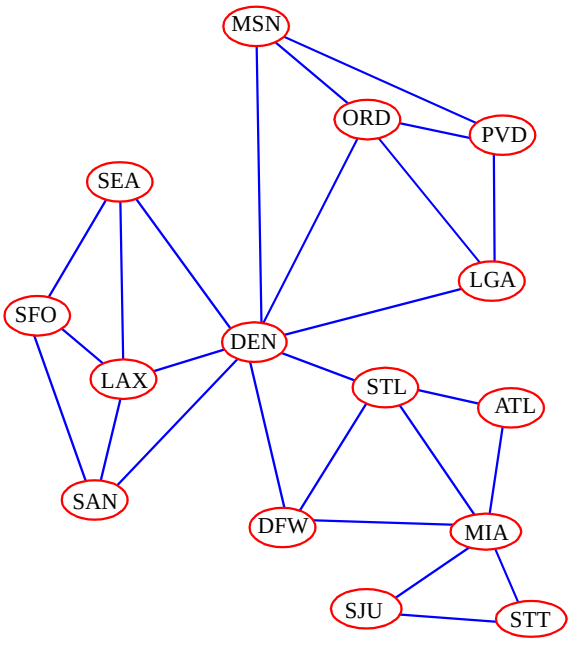
- L'ordre dans lequel les arcs arrières sont traités est essentiel pour la rectitude de l'algorithme
- L'utilisation d'un ordre différent...



- ... mène à un graphe qui contient des informations incorrectes.



Essayez l'algorithme sur ce graphe!



CHAÎNES DE CARACTÈRES ET APPARIEMENT (*STRINGS & PATTERN MATCHING*)

- *Pattern matching*, aussi appelé appariement, filtrage, ou correspondance de motifs ou de patrons.
- Force brute, Rabin-Karp, Knuth-Morris-Pratt
- Expressions régulières

Recherche dans les chaînes de caractères

- L'objectif de la recherche dans les chaînes de caractères (*string searching*) est de localiser un patron textuel (*text pattern*) spécifique au sein d'un texte plus long (phrase, paragraphe, livre, etc).
- Comme pour la plupart des algorithmes, les préoccupations principales pour la recherche dans les chaînes sont la vitesse et l'efficacité.
- Il existe plusieurs algorithmes pour la recherche dans les chaînes, mais les trois que nous étudierons sont force brute, Rabin-Karp, et Knuth-Morris-Pratt.

Force brute

- L'algorithme force brute compare le patron au texte, un caractère à la fois, jusqu'à ce que des caractères qui ne correspondent pas l'un à l'autre soient trouvés:

```
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
 ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
  ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
   ROADS
TWO ROADS DIVERGED IN A YELLOW WOOD
    ROADS
```

- Les caractères comparés sont en *italique*
- Les caractères qui correspondent sont en **gras**

- On peut demander à l'algorithme d'arrêter à la première occurrence du patron, ou à la fin du texte.

Pseudo-code pour force brute

- Voici le pseudo-code

```
repeat
  if (lettre du texte == lettre du patron)
    compare prochaine lettre du texte à la prochaine
      lettre du patron
  else
    déplacer le patron à la prochaine lettre
until (patron trouvé ou fin du texte)
```

```
cool cat Rolo went over the fence
cat
cool cat Rolo went over the fence
 cat
cool cat Rolo went over the fence
  cat
cool cat Rolo went over the fence
   cat
cool_cat Rolo went over the fence
    cat
cool cat Rolo went over the fence
     cat
```

Force brute — Complexité

- Soit un patron d'une longueur de M caractères et un texte d'une longueur de N caractères...
- **Pire des cas**: compare le patron à chaque sous-chaîne de caractères de longueur M. Par exemple, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAH **5 comparaisons**
2) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
    AAAAH **5 comparaisons**
3) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
     AAAAH **5 comparaisons**
4) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
      AAAAH **5 comparaisons**
5) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
       AAAAH **5 comparaisons**
   ....
N) AAAAAAAAAAAAAAAAAAAAAAA*AAAAH*
   5 comparaisons *AAAAH*

- Nombre total de comparaisons: M (N-M+1)
- Complexité du pire des cas: *O*(MN)

Force brute — Complexité (suite)

- Soit un patron d'une longueur de M caractères et un texte d'une longueur de N caractères...
- **Meilleur des cas si le patron est trouvé:** Trouve le patron au début du texte. Par exemple, M=5.

1) *AAAAA*AAAAAAAAAAAAAAAAAAAAH
   AAAAA **5 comparaisons**

- Nombre total de comparaisons: M
- Complexité du meilleur des cas: *O*(M)

Force brute — Complexité (suite)

- Soit un patron d'une longueur de M caractères et un texte d'une longueur de N caractères...
- **Meilleur des cas si le patron n'est pas trouvé:** Le premier caractère ne correspond jamais. Si M=5:

1) *A*AAAAAAAAAAAAAAAAAAAAAAAAH
   *O*OOOH **1 comparaison**
2) A*A*AAAAAAAAAAAAAAAAAAAAAAAH
    *O*OOOH **1 comparaison**
3) AA*A*AAAAAAAAAAAAAAAAAAAAAAH
     *O*OOOH **1 comparaison**
4) AAA*A*AAAAAAAAAAAAAAAAAAAAAH
      *O*OOOH **1 comparaison**
5) AAAA*A*AAAAAAAAAAAAAAAAAAAAH
       *O*OOOH **1 comparaison**
   ...
N) AAAAAAAAAAAAAAAAAAAAAAA*A*AAAH
   1 comparaison *O*OOOH

- Nombre total de comparaisons: N
- Complexité du meilleur des cas: *O*(N)

Rabin-Karp

- L'algorithme de recherche dans les chaînes de caractères de Rabin-Karp calcule une **valeur de hachage** pour le patron et pour chaque sous-séquence de M caractères du texte à être comparé.
- Si les valeurs de hachage sont différentes, l'algorithme calculera la valeur de hachage de la prochaine sous-séquence de M caractères.
- Si les valeurs de hachage sont égales, l'algorithme fera une comparaison selon l'approche par force brute entre le patron et la séquence de M caractères.
- De cette façon, il n'y a seulement qu'une comparaison par sous-séquence, et l'approche par force brute n'est nécessaire que quand les valeurs de hachage correspondent.
- Un exemple clarifiera probablement certains points...

Exemple avec Rabin-Karp

La valeur de hachage de “AAAAA” est 37

La valeur de hachage de “AAAAH” est 100

1) AAAAAAAAAAAAAAAAAAAAAAAAH
   AAAAH
   37≠100 **1 comparaison**
2) AAAAAAAAAAAAAAAAAAAAAAAAH
    AAAAH
   37≠100 **1 comparaison**
3) AAAAAAAAAAAAAAAAAAAAAAAAH
     AAAAH
   37≠100 **1 comparaison**
4) AAAAAAAAAAAAAAAAAAAAAAAAH
      AAAAH
   37≠100 **1 comparaison**

   ...

N) AAAAAAAAAAAAAAAAAAAAAAAAH
                AAAAH
   5 comparaisons 100=100

Algorithme de Rabin-Karp

le patron a une longueur de M caractères

hash_p = valeur de hachage du patron
hash_t = valeur de hachage des M premiers caractères du corps du texte

repeat
  if (hash_p == hash_t)
    comparaison par force brute entre le patron et la section de texte sélectionnée
  hash_t = valeur de hachage de la prochaine section de texte, un caractère plus loin
until (fin du texte **or**
  comparaison par force brute == true)

Rabin-Karp

- Questions fréquentes à propos de Rabin-Karp:

  “Quelle est la fonction de hachage utilisée pour calculer les valeurs associées aux séquences de caractères?”

  “L’application de cette fonction à chaque séquence de M caractères tirée du corps du texte ne prend-t-elle pas trop de temps?”

  “Cette matière sera-t-elle à l’examen final?”

- Afin de répondre à quelques-unes de ces questions, nous devrons faire un peu de mathématiques.

Mathématiques de Rabin-Karp

- Considérez une séquence de M caractères comme un nombre de M chiffres en base b, où b est le nombre de lettres dans l’alphabet. La sous-séquence de texte t[i .. i+M-1] est convertie au nombre suivant:

  $x(i) = t[i]\cdot b^{M-1} + t[i+1]\cdot b^{M-2} + ... + t[i+M-1]$

- De plus, étant donné x(i), nous pouvons calculer x(i+1) pour la sous-séquence suivante t[i+1 .. i+M] en un temps constant:

  $x(i+1) = t[i+1]\cdot b^{M-1} + t[i+2]\cdot b^{M-2} + ... + t[i+M]$

  $x(i+1) = x(i)\cdot b$ Déplacer à gauche de 1 chiffre...

  $- t[i]\cdot b^{M}$ Moins le chiffre le plus à gauche

  $+ t[i+M]$ Plus le nouveau chiffre le plus à droite

- De cette façon, nous n’avons jamais à calculer explicitement une nouvelle valeur. Nous ajustons tout simplement la valeur existante lorsque nous passons au caractère suivant.

Exemple mathématique avec Rabin-Karp

- Supposons que nous ayons un alphabet à 10 lettres.
- Alphabet = a, b, c, d, e, f, g, h, i, j
- Supposons que “a” corresponde à 1, que “b” corresponde à 2 et ainsi de suite.

La valeur de hachage pour la chaîne “cah” serait:

3*100 + 1*10 + 8*1 = 318

Modulo pour Rabin-Karp

- Si M est grand, alors la valeur résultante (~b^M) sera énorme. Pour cette raison, nous hachons cette valeur en la prenant **modulo** un nombre premier ***q***.
- La fonction **mod** (% en Java) est particulièrement utile dans ce cas grâce à quelques-unes de ses propriétés inhérentes:
  - [(x mod q) + (y mod q)] mod q = (x+y) mod q
  - (x mod q) mod q = x mod q
- Pour ces raisons:

$$h(i) = ((t[i]\cdot b^{M-1} \bmod q) + (t[i+1]\cdot b^{M-2} \bmod q) + \ldots + (t[i+M-1] \bmod q)) \bmod q$$

$$h(i+1) = (h(i)\cdot b \bmod q$$

Déplacer à gauche de 1 chiffre...

$$-t[i]\cdot b^{M} \bmod q$$

Moins le chiffre le plus à gauche

$$+t[i+M] \bmod q)$$

Plus le nouveau chiffre le plus à droite

$$\bmod q$$

Rabin-Karp — Complexité

- Si un nombre premier suffisamment grand est utilisé pour la *fonction de hachage*, les valeurs de hachage de deux patrons différents seront habituellement distinctes.
- Si c’est le cas, la recherche prend un temps $O(N)$, où N est le nombre de caractères dans le corps de texte le plus grand.
- Il est toujours possible de concevoir un scénario à la complexité du pire des cas $O(MN)$. Cependant, cette situation ne sera portée à survenir que si le nombre premier utilisé pour le hachage est petit.

L’algorithme Knuth-Morris-Pratt

- L’algorithme de recherche de Knuth-Morris-Pratt (KMP) diffère de l’approche par force brute en ce qu’il conserve l’information obtenue lors des comparaisons précédentes.
- Une fonction d’échec *(failure function)* (*f*) est calculée, et elle indique quelle partie de la comparaison précédente peut être réutilisée en cas d’échec.
- En fait, *f* est définie comme le plus long préfixe du patron P[0,..,j] qui est aussi un suffixe de P[1,..,j]
  - Note: **pas** un suffixe de P[0,..,j]
- Exemple — valeurs de la fonction d’échec KMP:

j	0	1	2	3	4	5
P[*j*]	a	b	a	b	a	c
f(*j*)	0	0	1	2	3	0

- Ceci indique quelle partie du début de la chaîne correspond jusqu’à la portion située juste avant une comparaison infructueuse.
  - Si la comparaison échoue à (4), nous savons que *a* et *b* aux positions 2 et 3 sont identiques aux positions 0 et 1.

L'algorithme KMP (suite)

- Pseudo-Code de l'algorithme d'appariement

```
Algorithme KMPMatch(T,P)
  Entrée: Chaînes T (texte) à n caractères et P
    (patron) à m caractères.
  Sortie: Index de départ de la première sous-chaîne de
    T qui correspond à P, ou une indication que P n'est
    pas une sous-chaîne de T.

  f ← KMPFailureFunction(P) {construit la f. d'échec}
  i ← 0
  j ← 0
  while i < n do
    if P[j] = T[i] then
      if j = m - 1 then
        return i - m - 1 {correspondent}
      i ← i + 1
      j ← j + 1
    else if j > 0 then {nous avons avancé...}
      j ← f(j-1) {considère le préfixe apparié dans P}
    else
      i ← i + 1
  return "Pas de sous-chaîne P dans T"
```

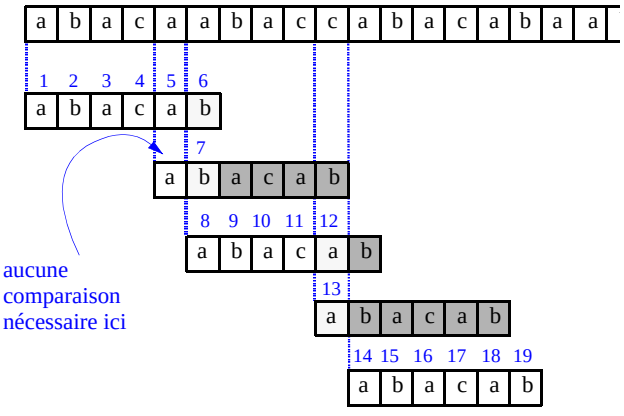

L'algorithme KMP (suite)

- Pseudo-Code de la fonction d'échec KMP

```
Algorithme KMPFailureFunction(P);
  Entrée: Chaînes P (patron) à m caractères.
  Sortie: La fonction d'échec f pour P, qui ajuste j selon
    la longueur du plus long préfixe de P qui est aussi
    suffixe de P[1,..,j].

  i ← 1
  j ← 0
  while i ≤ m-1 do
    if P[j] = P[i] then
      {nous avons apparié j + 1 caractères}
      f(i) ← j + 1
      i ← i + 1
      j ← j + 1
    else if j > 0 then
      {j considère le préfixe apparié dans P}
      j ← f(j-1)
    else
      {il n'y a pas de correspondance}
      f(i) ← 0
      i ← i + 1
```


L'algorithme KMP (suite)

- Une représentation graphique de l'algorithme de recherche dans les chaînes KMP



L'algorithme KMP (suite)

- Analyse de la complexité temporelle
- définissons $k = i - j$
- À chaque itération de la boucle *while*, l'une des trois choses suivantes surviendra:.
  - 1) si $T[i] = P[j]$, alors i est incrémenté de 1, tout comme j. k reste inchangé.
  - 2) si $T[i] != P[j]$ et $j > 0$, alors i reste inchangé et k est incrémenté d'au moins 1, puisque k change de $i - j$ à $i - f(j-1)$
  - 3) si $T[i] != P[j]$ et $j = 0$, alors i est incrémenté de 1 et k est incrémenté de 1 puisque j reste inchangé.
- Ainsi, à chaque itération, i ou k est incrémenté d'au moins 1, alors le nombre maximal d'itérations est $2n$
- Ceci présuppose bien sûr que f ait été calculé auparavant.
- Cependant, f est calculé sensiblement de la même façon que KMPMatch, alors sa complexité est semblable. KMPFailureFunction prend $O(m)$
- Complexité temporelle totale: $O(n + m)$

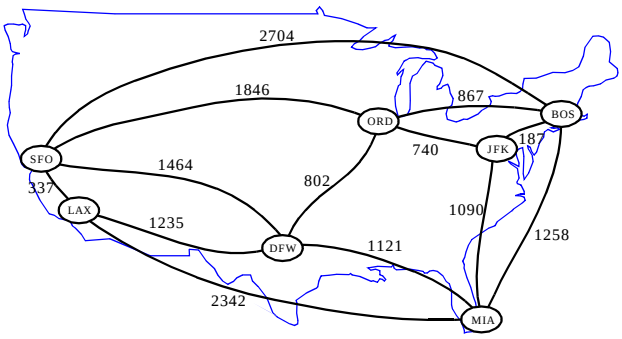
Expressions régulières

- Notation pour décrire un ensemble de chaînes de caractères, possiblement de taille infinie.
- ε dénote la chaîne vide
- **ab + c** dénote l'ensemble {ab, c}
- **a*** dénote l'ensemble {ε, a, aa, aaa, ...}
- Exemples
  - **(a+b)*** toutes les chaînes avec l'alphabet {a,b}
  - **b*(ab*a)*b*** chaînes avec un nombre pair de "a"
  - **(a+b)*sun(a+b)*** chaînes contenant le motif "sun"
  - **(a+b)(a+b)(a+b)a** chaîne de quatre lettres se terminant par "a"

PLUS COURTS CHEMINS

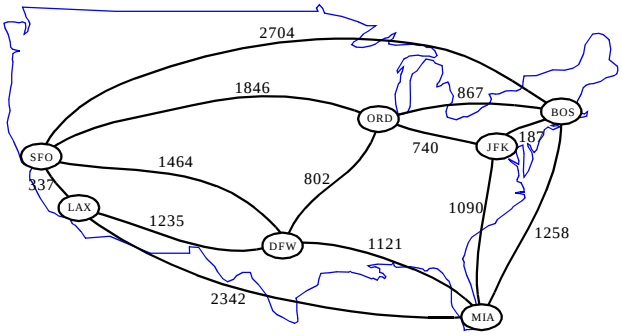
(*Shortest Paths*)

- Graphes pondérés
- Plus courts chemins



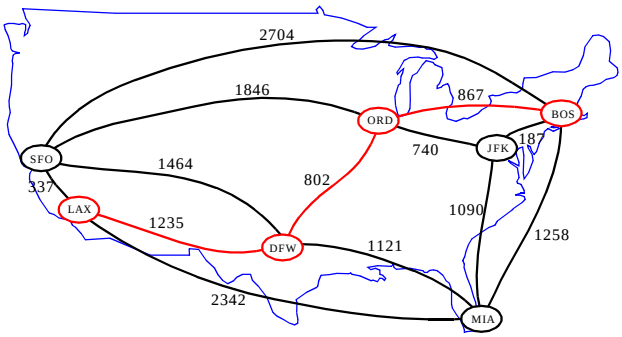
Graphes pondérés

- Les **poids** sur les arcs d'un graphe représentent des distances, des coûts, etc.
- Un exemple d'un graphe pondéré non-orienté:



Plus court chemin

- BFS trouve le(s) chemin(s) au nombre d'arcs minimal à partir du sommet de départ
- Ainsi, BFS trouve le plus court chemin en supposant que chaque arc a le même poids
- Dans plusieurs domaines, par exemple les réseaux routiers, les arcs d'un graphe ont des poids différents
- Comment trouver les chemins au poids total minimal?
- Exemple - Boston à Los Angeles:



Algorithme de Dijkstra

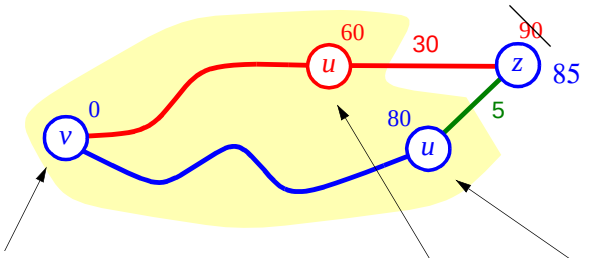
- L'Algorithme de Dijkstra trouve les plus courts chemins d'un sommet de départ *v* vers tous les autres sommets d'un graphe avec:
  - **des arcs non-orientés**
  - **des arc au poids non-négatif**
- L'algorithme calcule, pour chaque sommet *u*, la **distance** de *u* à partir du sommet *v*, et donc le poids d'un plus court chemin entre *v* et *u*.
- L'algorithme conserve l'ensemble des sommets pour lesquels la distance a été calculée, appelé **nuage** (*cloud*) C
- Chaque sommet a une étiquette D. Pour tout sommet *u*, nous appellerons son étiquette D[*u*]. D[*u*] contient une approximation de la distance entre *v* et *u*. L'algorithme met à jour une valeur D[*u*] quand il trouve un chemin plus court de *v* à *u*.
- Lorsqu'un sommet *u* est ajouté au nuage, son étiquette D[*u*] est égale à la distance actuelle (finale) entre le sommet de départ *v* et le sommet *u*.
- Initialement, nous choisissons:
  - D[*v*] = 0 ...la distance de *v* à lui-même est 0...
  - D[*u*] = ∞ pour $u \neq v$ ...ceci va changer...

L'algorithme: Expansion du nuage

- Répétez jusqu'à ce que tous les sommets soient dans le nuage:
  - soit u un sommet hors du nuage qui a la plus petite étiquette D[u]. (À la première itération, il est évident que le sommet de départ sera choisi)
  - ajoutez u au nuage C
  - mettez à jour les étiquettes des sommets adjacents à u de la façon suivante:

```
for chaque sommet z adjacent à u do
  if z est hors du nuage C then
    if D[u] + weight(u,z) < D[z] then
      D[z] = D[u] + weight(u,z)
```

- cette étape est appelée une **relaxation** de l'arc (u,z)



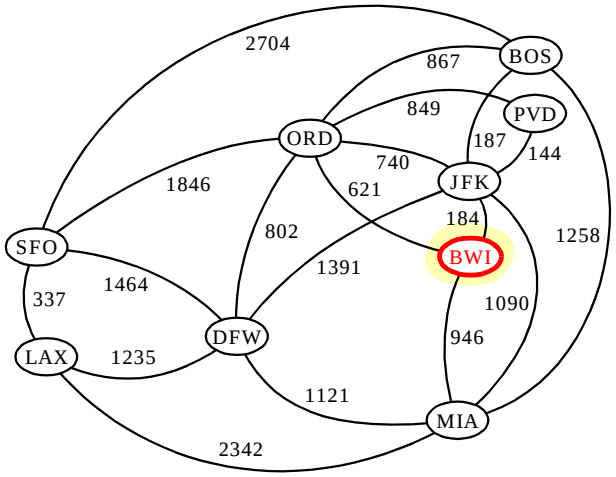
v est mis dans le nuage en premier. Puis ce u. Et ce u.

Pseudo-code

- Nous utilisons une file à priorité Q pour emmagasiner les sommets hors du nuage, où $D[v]$ est la clé d'un sommet v dans Q

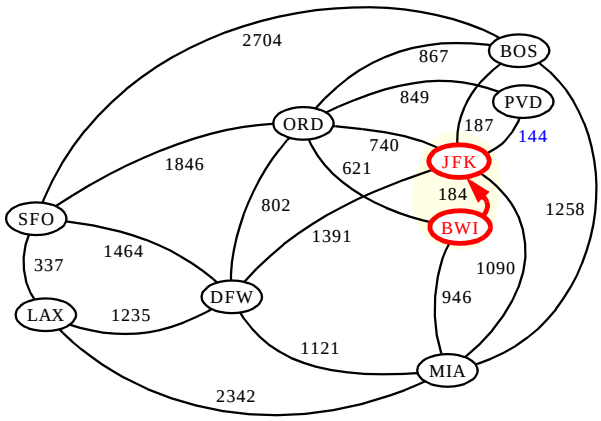
```
Algorithme ShortestPath(G, v):
  Entrée: Un graphe pondéré G et un sommet v de G.
  Sortie: Une étiquette D[u], pour chaque sommet u de
          G, où D[u] est la longueur d'un plus court
          chemin de v à u dans G.

  initialisez D[v] ← 0 et D[u] ← +∞ pour chaque
    sommet v ≠ u
  soit Q une file à priorité contenant tous les sommets
    de G utilisant les étiquettes D comme clés.
  while Q ≠ ∅ do
    {mettre u dans le nuage C}
    u ← Q.removeMinElement()
    for chaque sommet z adjacent à u où z est dans Q do
      {faire l'opération de relaxation sur l'arc (u, z)}
      if D[u] + w((u, z)) < D[z] then
        D[z] ←D[u] + w((u, z))
        changez la valeur de la clé de z dans Q à D[z]
  return l'étiquette D[u] de chaque sommet u.
```


Exemple: plus courts chemins à partir de BWI



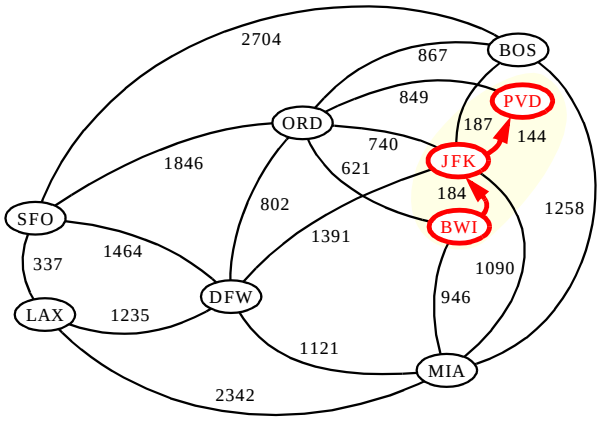
	parent	distance
BOS		∞
BWI		0
DFW		∞
JFK	BWI	184
LAX		∞
MIA	BWI	946
ORD	BWI	621
PVD		∞
SFO		∞

- JFK (New-York) est le plus près...



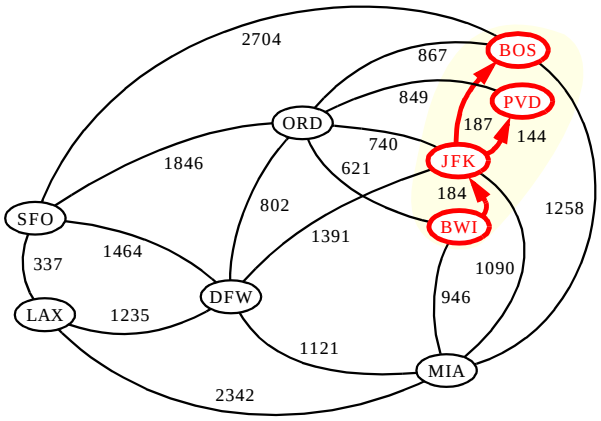
	parent	distance
BOS	JFK	371
BWI		0
DFW	JFK	1575
JFK	BWI	184
LAX		∞
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO		∞

- suivi de PVD (Providence)...



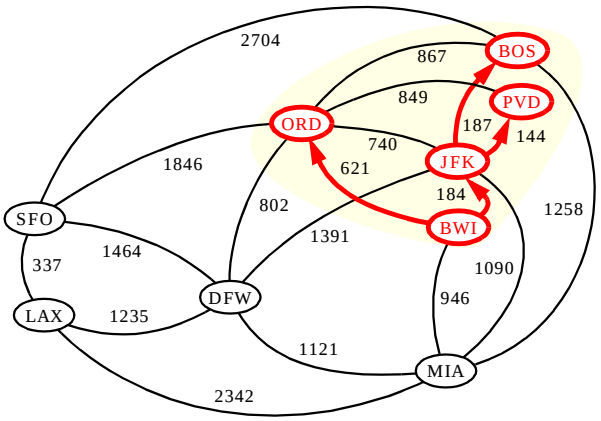
	parent	distance
BOS	JFK	371
BWI		0
DFW	JFK	1575
JFK	BWI	184
LAX		∞
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO		∞

- BOS (Boston) est juste un peu plus loin.



	parent	distance
BOS	JFK	371
BWI		0
DFW	JFK	1575
JFK	BWI	184
LAX		∞
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO	BOS	3075

- ORD (Chicago) les suit.

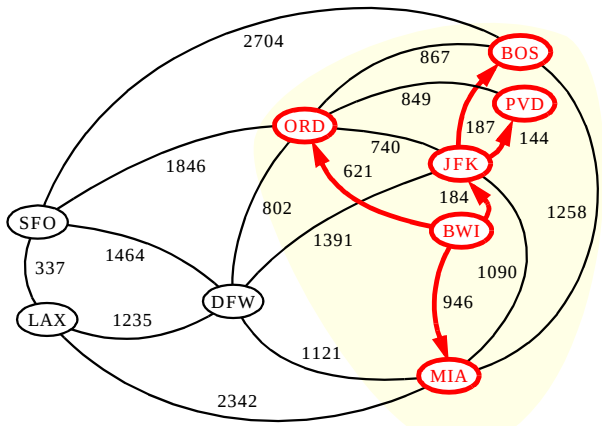


	parent	distance
BOS	JFK	371
BWI		0
DFW	ORD	1423
JFK	BWI	184
LAX		∞
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO	ORD	2467

notez que la distance D pour DWF a été ajustée à cette étape

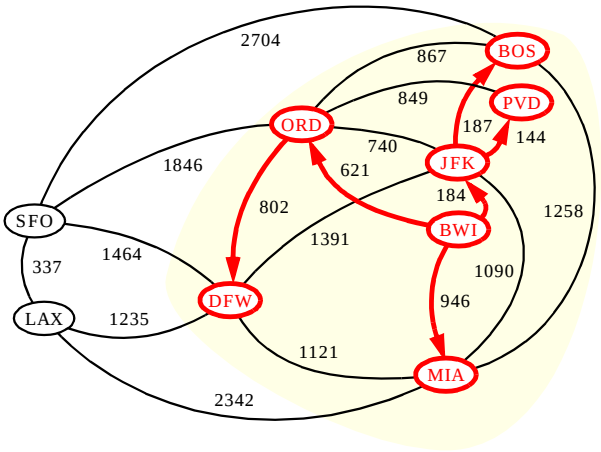
même chose pour SFO

- Puis MIA (Miami).



	parent	distance
BOS	JFK	371
BWI		0
DFW	JFK	1423
JFK	BWI	184
LAX	MIA	3288
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO	BOS	2467

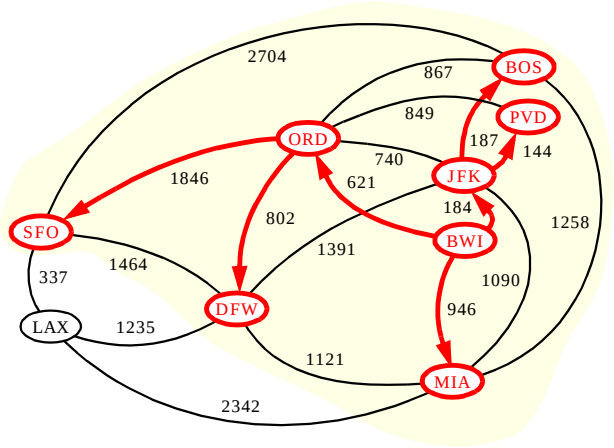
- Au tour de DFW...



	parent	distance
BOS	JFK	371
BWI		0
DFW	JFK	1423
JFK	BWI	184
LAX	DFW	2658
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO	BOS	2467

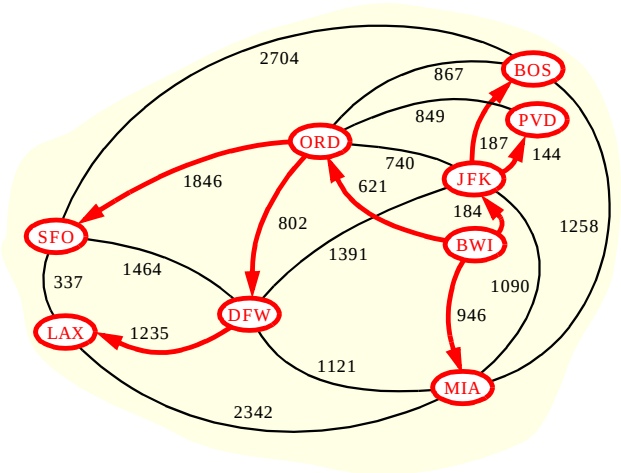
Distance D pour LAX est mise à jour

- Et de SFO...



	parent	distance
BOS	JFK	371
BWI		0
DFW	ORD	1423
JFK	BWI	184
LAX	MIA	2658
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO	BOS	2467

- Et enfin de LAX.



	parent	distance
BOS	JFK	371
BWI		0
DFW	ORD	1423
JFK	BWI	184
LAX	MIA	2658
MIA	BWI	946
ORD	BWI	621
PVD	JFK	328
SFO	BOS	2467

Temps d'exécution

- Supposons une représentation de G avec une liste d'adjacence. Nous pouvons alors parcourir tous les sommets adjacents à u en un temps proportionnel à leur nombre (donc $O(j)$ où j est le nombre de sommets adjacents à u)
- La file à priorité Q — choix à faire:
  - Un **tas**: réaliser Q avec un tas permet une extraction efficace des sommets à la plus petite étiquette D ($O(\log N)$). Si Q est réalisé avec des repéreurs (*locators*), la mise à jour des clés peut se faire en un temps $O(\log N)$. Le temps d'exécution total est $O((n+m) \log n)$ où n est le nombre de sommets dans G et m est le nombre d'arcs. En terme de n, le pire des cas est $O(n^2 \log n)$.
  - Une **séquence non-triée**: $O(n)$ pour l'extraction des éléments minimaux, mais rapide mise à jour des clés ($O(1)$). Il n'y a seulement que n-1 extractions et m relaxations. Le temps d'exécution est $O(n^2+m)$
- En ce qui concerne le **pire des cas**, le tas est bon pour de petits ensembles de données, et la séquence pour de plus grands ensembles.

Temps d'exécution (suite)

- Le ***cas moyen*** est une toute autre histoire. Considérez ceci:
  - Si la file à priorité Q est réalisée avec un tas, le goulot d'étranglement de trouve à être la mise à jour de la clé d'un sommet dans Q. Dans le pire des cas, nous aurions besoin d'une mise à jour pour chaque arc dans le graphe.
  - Cependant, pour la plupart des graphes, ceci n'arrivera pas. En supposant un **ordre aléatoire de voisinage**, nous observons que pour chaque sommet, ses sommets voisins seront placés dans le nuage dans un ordre quelconque. Ainsi il n'y a que $O(\log n)$ mises à jour de la clé d'un sommet.
  - Avec cette même supposition, le temps d'exécution de la réalisation par tas est $O(n \log n + m)$, qui est toujours $O(n^2)$.

  La réalisation par tas est donc préférable pour tous les cas sauf ceux qui sont dégénérés.

Algorithme de Dijkstra, quelques trucs auxquels penser...

- Dans notre exemple, le **poids est la** distance géographique. Cependant, le poids aurait pu tout aussi bien représenter le coût ou le temps de vol.
- Nous pouvons aisément **modifier l'algorithme de Dijkstra selon les besoins**, par exemple:
  - Si nous ne désirons que le plus court chemin de v à un sommet particulier u, nous pouvons arrêter l'algorithme aussitôt que u est mis dans le nuage.
  - Nous pourrions aussi faire que l'algorithme retourne un arbre T enraciné à v où le chemin dans T de v à u est le plus court chemin de v à u.
- **Comment conserver poids et distances?** Les arcs et sommets ne "connaissent" pas leur poids/distance. Prenez avantage du fait que D[u] est la clé pour le sommet u dans la file à priorité, et ainsi D[u] peut être retracé en connaissant le repéreur de u dans Q.
- Nous avons besoin d'un façon de:
  - associer des repéreurs PQ aux sommets
  - emmagasiner et récupérer le poids des arcs
  - retourner les distances finales

ARBRE RECOUVRANT MINIMAL

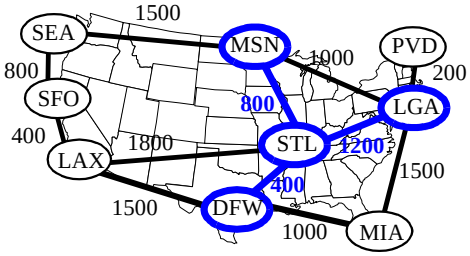
- Algorithme de Prim-Jarnik
- Algorithme de Kruskal

Graphes pondérés

(poids d'un sous-graphe **G'**) =
(somme des poids des arcs de **G'**)

$$\textbf{poids(G')} = \sum_{(e \in G')} \textbf{poids(e)}$$

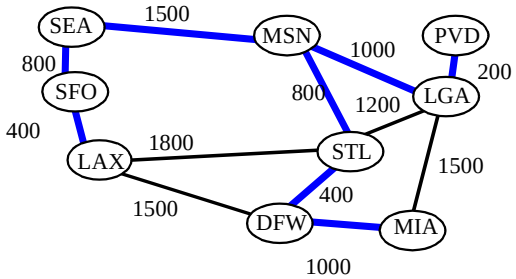
G'



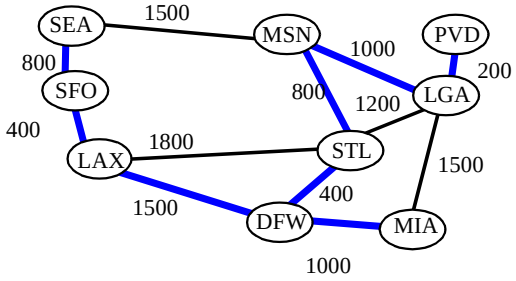
poids(G') = 800 + 400 + 1200
= 2400

Arbre recouvrant minimal (MST)

- arbre recouvrant de poids total minimal
- par exemple, pour connecter tous les ordinateurs d'un édifice avec une quantité minimale de câble
- exemple

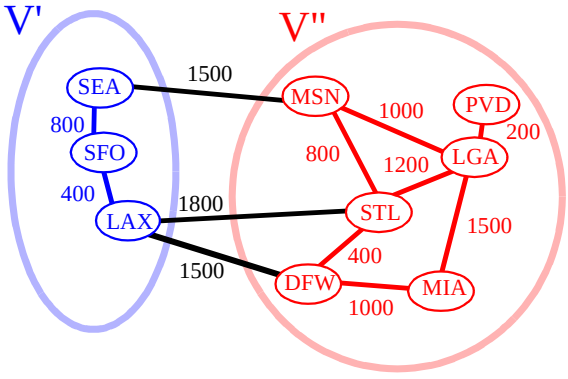


- pas unique en général



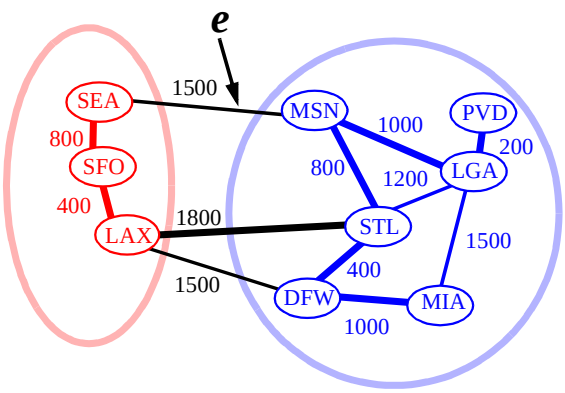
Propriété des arbres recouvrants minimaux

Soit (V',V"), une partition des sommets de G.
Soit ***e*** = (v', v"), un arc de poids minimal traversant la partition, c'est-à-dire v' ∈ V' et v" ∈ V".
Il y a un arbre recouvrant minimal (MST) contenant l'arc e.



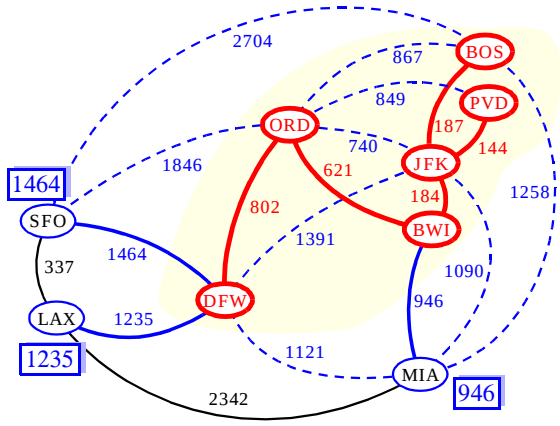
Preuve de la propriété

Si le MST ne contient pas un arc de poids minimal ***e***, alors nous pouvons trouver un MST meilleur ou égal en échangeant ***e*** pour un autre arc.



Algorithme de Prim-Jarnik pour trouver un MST

- agrandit le MST T d'un sommet à la fois
- le ***nuage*** couvre la portion de T déjà calculée
- étiquettes D[*u*] et E[*u*] associées à chaque sommet *u*
  - E[*u*] est le meilleur arc (poids le plus bas) connectant *u* à T
  - D[*u*] (distance au nuage) est le poids de E[*u*]



Différences entre les algorithmes de Prim et Dijkstra

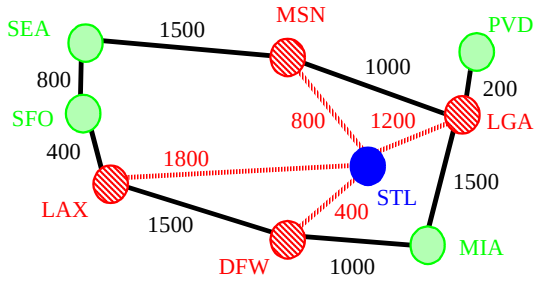
- Pour tout sommet *u*, **D[*u*] représente (à date) le poids du meilleur arc qui joint *u* au reste de l'arbre** (contrairement à la somme totale des poids d'arcs sur un chemin du sommet de départ à *u*).
- Prim utilise une file à priorité Q dont les clés sont les étiquettes D, et dont les **éléments sont des paires sommet-arc**.
- Tout sommet *v* peut être le **sommet de départ**.
- Nous initialisons toujours toutes les valeurs de D[*u*] à "infini", mais nous **initialisons aussi E[*u*] (les arcs associés à *u*) à "aucun".**
- **Retourne** l'arbre recouvrant minimal T.

Nous pouvons réutiliser le code produit par Dijkstra, et ne changer que quelques parties. Observons le pseudo-code....

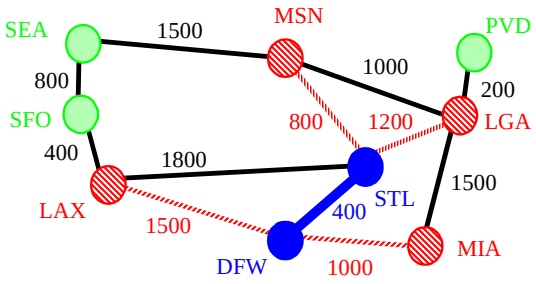
Pseudo-code

Algorithme PrimJarnik(*G*):
  Entrée: un graphe connexe pondéré *G*.
  Sortie: un arbre recouvrant minimal *T* pour *G*.
choisir n'importe quel *v* de G
{agrandir l'arbre débutant avec le sommet *v*}
T ← {*v*}
  D[*v*] ← 0
  E[*v*] ← ∅
for chaque sommet *u* ≠ *v* **do**
  D[*u*] ← +∞
soit *Q* une file à priorité qui contient des sommets
    et qui utilise les étiquettes *D* comme clés
while *Q* ≠ ∅ **do**
    {placer *u* dans le nuage C}
    u← *Q*.removeMinElement()
    ajouter le sommet *u* et l'arc *E*[*u*] à *T*
    for chaque sommet *z* adjacent à *u* **do**
      if *z* est dans *Q*
        {faire ***l'opération de relaxation*** sur l'arc (*u*, *z*)}
        if poids(*u*, *z*) < *D*[*z*] **then**
          D[*z*] ← poids(*u*, *z*)
          E[*z*] ← (*u*, *z*)
          changer la clé de *z* dans *Q* pour *D*[*z*]
  return l'arbre *T*

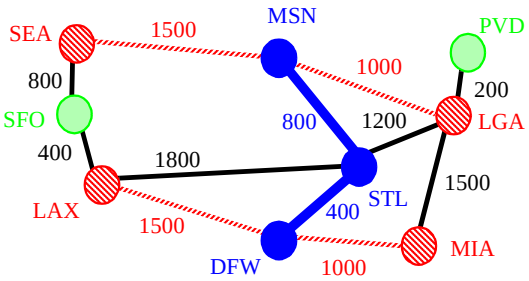
Parcourons son exécution...



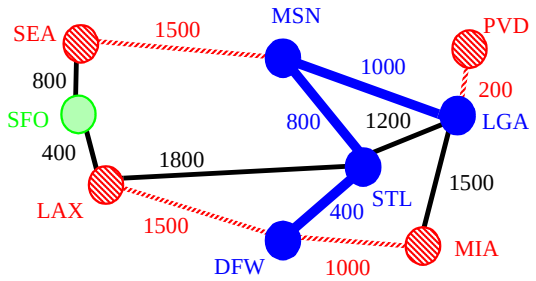
	voisin	D[u]
DFW	STL	400
LAX	STL	1800
LGA	STL	1200
MIA		
MSN	STL	800
PVD		
SEA		
SFO		
STL		



	voisin	D[u]
DFW		
LAX	DFW	1500
LGA	STL	1200
MIA	DFW	1000
MSN	STL	800
PVD		
SEA		
SFO		
STL		



	voisin	D[u]
DFW		
LAX	DFW	1500
LGA	MSN	1000
MIA	DFW	1000
MSN		
PVD		
SEA	MSN	1500
SFO		
STL		



	voisin	D[u]
DFW		
LAX	DFW	1500
LGA		
MIA	DFW	1000
MSN		
PVD	LGA	200
SEA	MSN	1500
SFO		
STL		

Temps d'exécution

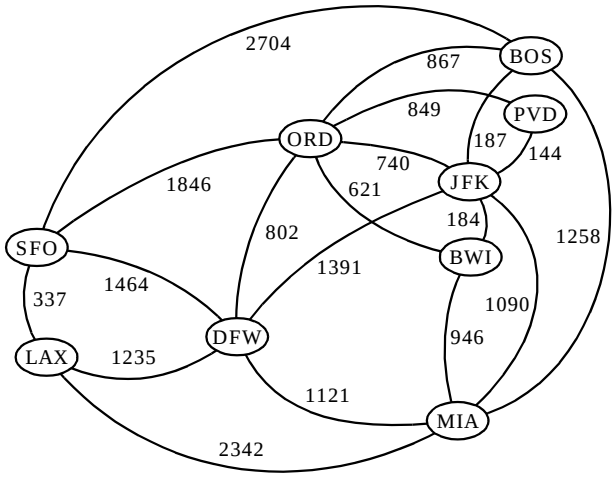
```
T ← {v}
  D[v] ← 0
  E[v] ← ∅
for chaque sommet u ≠ v do
  D[u] ← +∞
soit Q une file à priorité qui contient des sommets
    et qui utilise les étiquettes D comme clés
while Q ≠ ∅ do
    u ← Q.removeMinElement()
    ajouter le sommet u et l'arc E[u] à T
    for chaque sommet z adjacent à u do
      if z est dans Q
        if poids(u, z) < D[z] then
          D[z] ← poids(u, z)
          E[z] ← (u, z)
          changer la clé de z dans Q pour D[z]
return l'arbre T
```

$O((n+m) \log n)$

où n = nombre de sommets, m = nombre d'arcs, et Q est réalisé avec un tas.

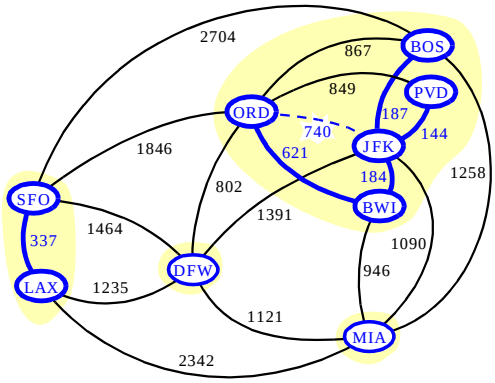
Algorithme de Kruskal

- ajoutez les arcs un à la fois, en ordre croissant de poids.
- acceptez un arc si il ne crée pas de cycle.



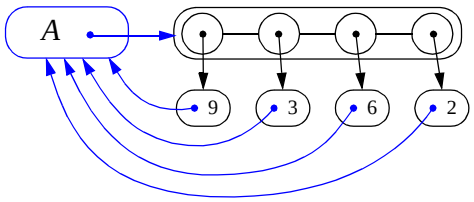
Structures de données pour l'algorithme de Kruskal

- l'algorithme maintient une forêt d'arbres
- un arc est accepté si il relie des sommets d'arbres distincts
- nous avons besoin d'une structure de données qui maintient une partition, c'est-à-dire une collection d'ensembles disjoints, avec les opérations suivantes
  - **find**(u): retourne l'ensemble contenant u
  - **union**(u,v): remplace les ensembles contenant u et v par leur union



Représentation d'une partition

- chaque ensemble est emmagasiné dans une séquence
- chaque élément a une référence vers son ensemble



- l'opération **find**(u) requiert $O(n)$ et retourne l'ensemble dont u est un membre
- dans l'opération **union**(u,v), nous déplaçons les éléments du plus petit ensemble vers la séquence du plus grand, tout en mettant à jours leurs références
- Le temps associé à l'opération **union**(u,v) est $\min(n_u,n_v)$, où n_u et n_v sont les tailles respectives des ensembles contenant u et v
- lorsqu'un élément est traité, il se retrouve dans un ensemble de taille au moins du double
- ainsi, chaque élément est traité au plus log n fois

Pseudo-code

Algorithme Kruskal(G):
  Entrée: Un graphe connexe pondéré G.
  Sortie: un arbre recouvrant minimal T pour G.

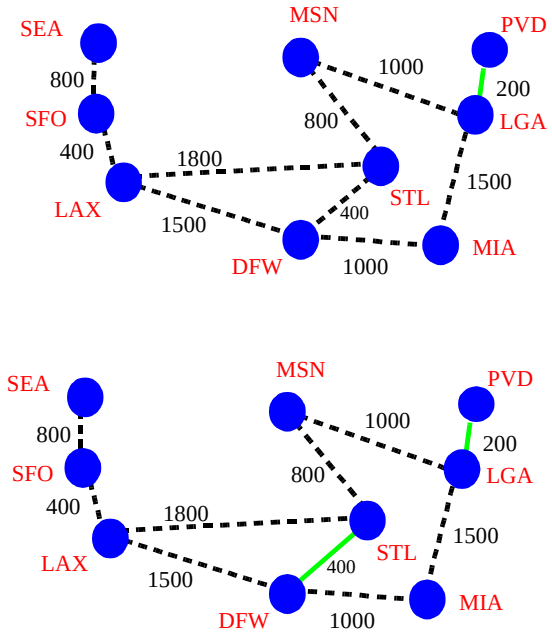
soit P une partition des sommets de G où chaque sommet forme un ensemble séparé

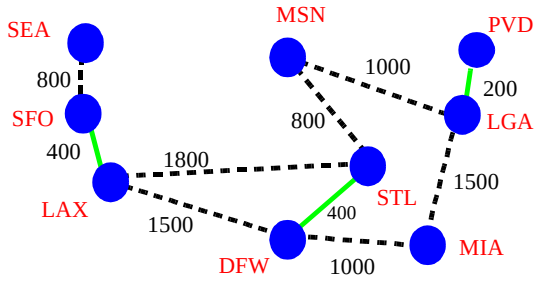
soit Q une file à priorité emmagasinant les arcs de G, triés selon leur poids

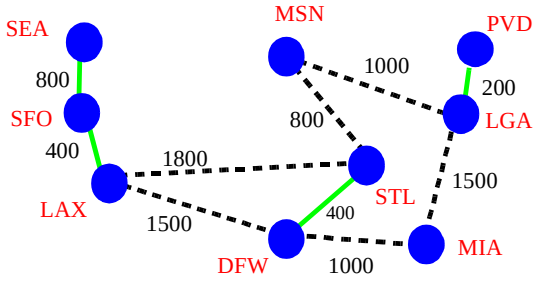
$T \leftarrow \emptyset$
while $Q \neq \emptyset$ **do**
  $(u,v) \leftarrow Q$.removeMinElement()
  if P.**find**(u) $\neq$ P.**find**(v) **then**
    ajouter l'arc (u,v) à T
    P.**union**(u,v)
return T

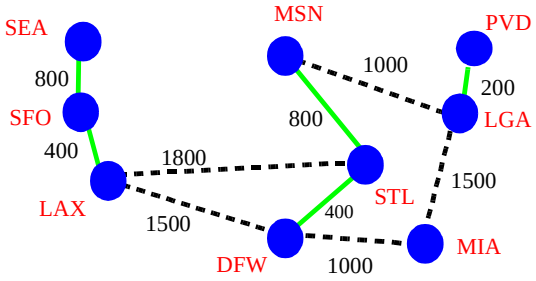
Temps d'exécution: $O((n+m) \log n)$

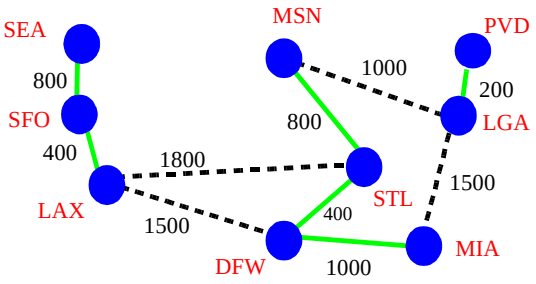
Parcourons son exécution...

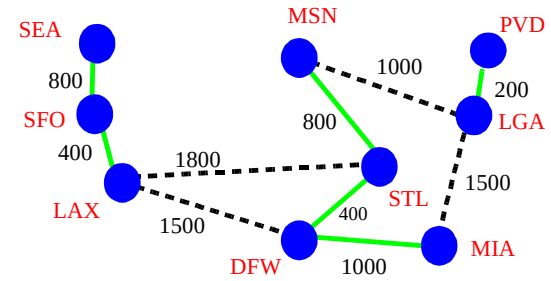


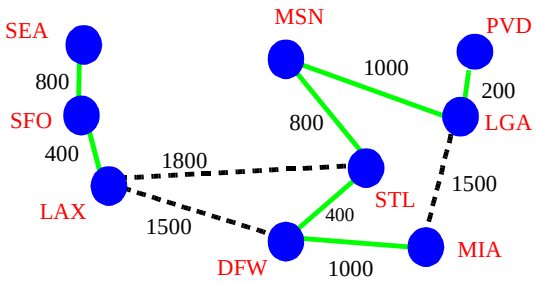




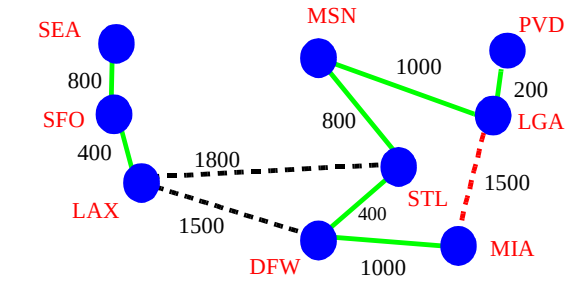


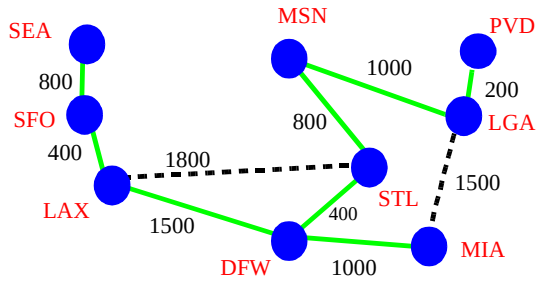






Examine LGA-MIA, mais ne l'ajoute pas à T parce que LGA&MIA sont dans le même ens.





Examine ici LAX-STL, mais ne l'ajoute pas à T parce que LAX et STL sont dans le même ensemble. Et c'est fini!

FLOT MAXIMAL

- Comment le faire...
- Pourquoi le désirer...
- Où le trouver...
- Ford-Fulkerson
- Edmonds-Karp
- Coupe minimale

Le Tao du Flot (*Flow*):

"Let your body go with the flow."
-Madonna, *Vogue*

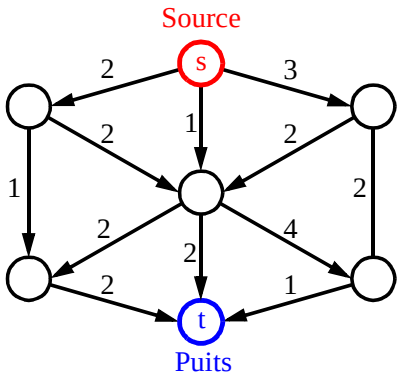
"Go with the flow, Joe."
-Paul Simon, *50 ways to leave your lover*

"Use the flow, Luke!"
-Obi-Wan Kenobi, *Star Wars*

"Connaissez le flot, ou coulez le cours..."
-Fernando Gomes, CSI 2514

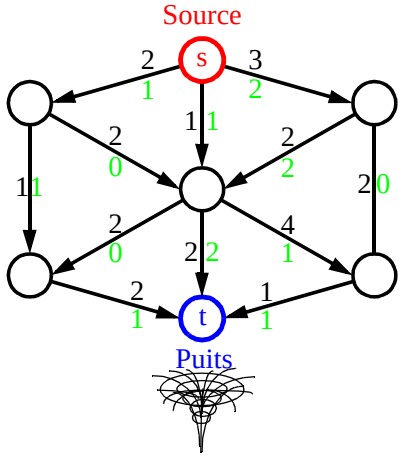
Réseaux de flots

- Réseau de flots:
  - digraphe
  - poids, appelés capacités sur les arcs
  - deux sommets distinctifs:
    - Source, "s":
      Sommet sans aucun arc en entrée
    - Puits, "t":
      Sommet sans aucun arc en sortie.



Capacité et flot

- Capacités d'arc:
  Poids non négatif sur les arcs de réseau
- Flot:
  - Fonction sur les arcs de réseau:
    $0 \le$ flot $\le$ **capacité**
    flot entrant dans un sommet = flot sortant
    valeur: flots combinés dans le *puits*



La logique du flot

- Flot:
  flot(*u*,*v*) $\forall$ **arc(*u*,*v*)**
  -Règle de la capacité: $\forall$ arc (u,v)

  $0 \le$ flot$(u,v) \le$ **capacité**(u,v)

  -Règle de la conservation: $\forall$ sommet $v \neq$ s, t

  $$\sum_{u \in \text{in}(v)} \text{flot}(u,v) = \sum_{w \in \text{out}(v)} \text{flot}(v,w)$$

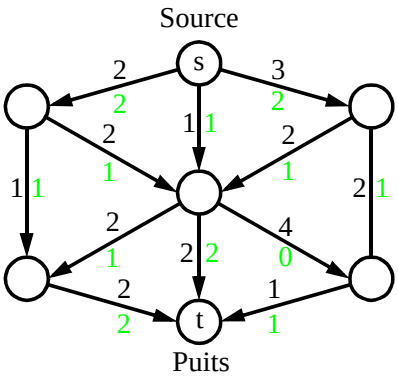
  -Valeur du flot:

  $$|f| = \sum_{w \in \text{out(s)}} \text{flot(s},w) = \sum_{u \in \text{in(t)}} \text{flot}(u,\text{t})$$

- **Note:**
  - $\forall$ signifie "pour tout"
  - in(v) est l'ensemble des sommets u où il y a un arc de u à v
  - out(v) est l'ensemble des sommets w où il y a un arc de v à w

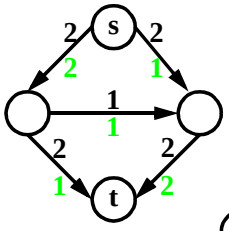
Problème du flot maximal

- "Étant donné un réseau *N*, trouvez un flot *f* de valeur maximale."
- Applications:
  - Circulation
  - Systèmes hydrauliques
  - Circuits électriques
  - Configurations

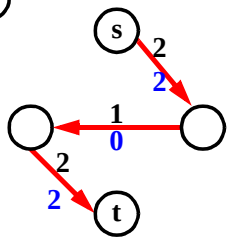


Exemple de flot maximal

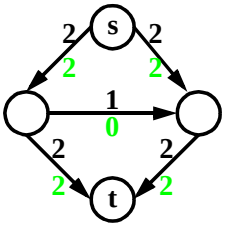
Flot augmentant



Un réseau avec flot de valeur **3**



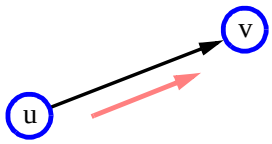
Chemin augmentant



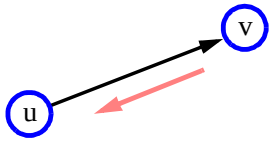
- Voila! nous avons augmenté la valeur de flot à 4! Main un instant! Qu'est-ce qu'un chemin augmentant?!?

Chemin augmentant

- **Arcs avant (*forward edges*)**
  flot(u,v) < **capacité**(u,v)
  le flot peut être augmenté!



- **Arcs arrières (*backward edges*)**
  flot(u,v) > 0
  le flot peut être diminué!



Théorème du flot maximal

Un flot a une valeur maximale
si et seulement si
il n'a pas de chemin augmentant.

Preuve:

Flot est maximal ⇒ Pas de chemin augmentant

(La partie ***seulement si*** est simple à prouver.)

Pas de chemin augmentant ⇒ Flot est maximal

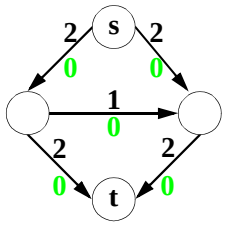
(Prouver la partie ***si*** s'avère plus difficile.)

Algorithme de Ford et Fulkerson

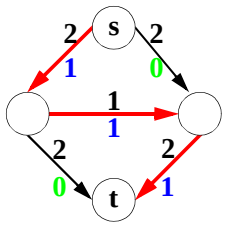
initialiser le réseau avec des flots nuls;
Méthode FindFlow
si un chemin augmentant existe alors
trouver un chemin augmentant;
accroître le flot;
appeler récursivement FindFlow;

- Et maintenant, place à un peu d'animation algorithmique...

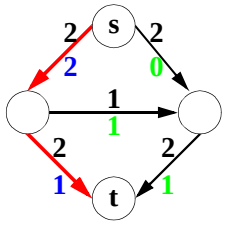
Trouver le flot maximal



Initialiser le réseau avec des flots nuls. Notez les **capacités au dessus des arcs**, et les flots sous les arcs.

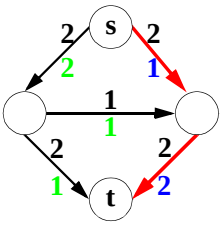


Envoyer une unité de flot dans le réseau. Notez le chemin de l'unité de flot en rouge. Les valeurs de flot augmentées sont en bleu.

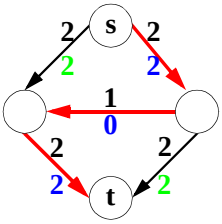


Envoyez une autre unité de flot dans le réseau.

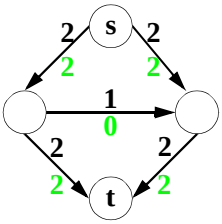
Trouver le flot maximal



Envoyez une autre unité de flot dans le réseau. Notez qu'il existe encore un chemin augmentant, qui peut aller *vers l'arrière*, contre l'arc central.



Envoyez une unité de flot dans le chemin augmentant. Notez qu'il n'y a plus de chemin augmentant. Ce qui signifie...

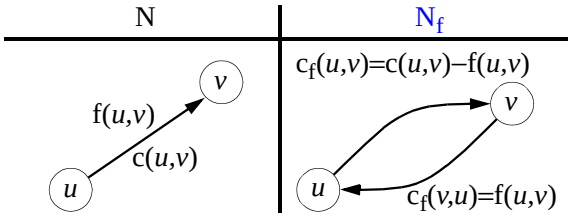


Avec l'aide de Ford & Fulkerson, nous avons atteint le ***flot maximal*** de ce réseau.

Ça c'est de la puissance!!!

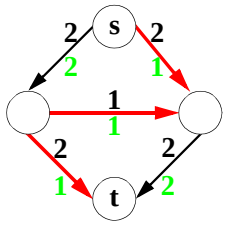
Réseau résiduel

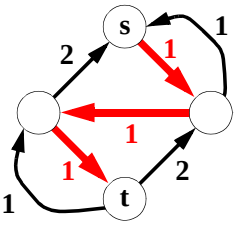
- Réseau résiduel $N_f = (V, E_f, c_f, s, t)$



- Dans le réseau résiduel N_f, tous les arcs (w,z) avec capacité $c_f(w,z) = 0$ sont supprimés.

Chemin augmentant dans le réseau N ↔ Chemin orienté dans le réseau résiduel N_f





Les chemins augmentants peuvent être trouvés avec une recherche en profondeur (DFS) sur N_f

L'algorithme de flot maximal de Ford-Fulkerson

Algorithme: MaxFlow(N)
Entrée: réseau N
Sortie: réseau N_f au flot maximal

Partie I: Mise en place
Débutez avec un flot nul:
  $f(u,v) \leftarrow 0 \; \forall \; (u,v) \in E$;
Initialisez le réseau résiduel:
  $N_f \leftarrow N$;

Partie II: Boucle
repeat
  recherchez un chemin orienté p dans N_f de s à t
  if (chemin p trouvé)
    $D_f \leftarrow \min \{c_f(u,v), f(u,v) \in p\}$;
    for (chaque $(u,v) \in p$) **do**
      if (avant (u,v))
        $f(u,v) \leftarrow f(u,v) + D_f$;
      if (arrière (u,v))
        $f(u,v) \leftarrow f(u,v) - D_f$;
    mettre à jour N_f;
until (pas de chemin augmentant);

Flot maximal: complexité temporelle

- Et maintenant, le moment tant attendu: la complexité temporelle de l'algorithme de flot maximal de Ford et Fulkerson (*roulements de tambour!!!*) [Pause pour effet dramatique]

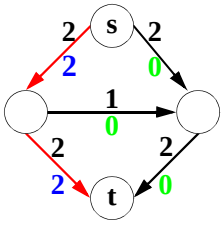
$$O(F (n + m))$$

  où F est la valeur du flot maximal, n est le nombre de sommets, et m est le nombre d'arcs

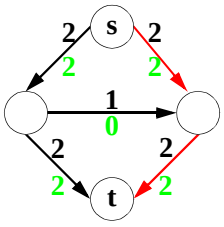
- Le problème avec cet algorithme, cependant, est qu'il dépend fortement de la valeur du flot maximal F. Par exemple, si $F=2^n$ l'algorithme pourrait prendre un temps exponentiel.
- Alors, arrivent enfin Edmonds et Karp...

Edmonds-Karp

- Variation sur l'algorithme de Ford et Fulkerson
- Utilise BFS pour choisir le chemin augmentant
- Trouver un plus court chemin de s à t. Y envoyer le plus grand flot possible.



- Répéter.



- C'est terminé.

Pseudo-code

Algorithme: Edmonds-Karp MaxFlow(N)
Entrée: réseau N
Sortie: réseau N_f au flot maximal

Partie I: Mise en place
Débutez avec un flot nul:
  $f(u,v) \leftarrow 0 \; \forall \; (u,v) \in E$;
Initialisez le réseau résiduel:
  $N_f \leftarrow N$;

Partie II: Boucle
repeat
  $p \leftarrow$ BFS-Shortest-Path(s,t,N_f)
  if (chemin p trouvé)
    $e_f \leftarrow (u_0,v_0)$, $c_f(u_0,v_0) = \min\{c_f(u,v), (u,v) \in p\}$
    $D_f \leftarrow c_f(e_f)$
      for (chaque $(u,v) \in p$)
        $f(u,v) \leftarrow f(u,v) + D_f$
        $c_f(u,v) \leftarrow c_f(u,v) - D_f$
    N_f.remove(e_f)
until (pas de chemin augmentant)

Flot maximal: amélioration

- Théorème: [Edmonds & Karp, 1972]

  En utilisant BFS (recherche en profondeur), un flot maximal peut être calculé en un temps...

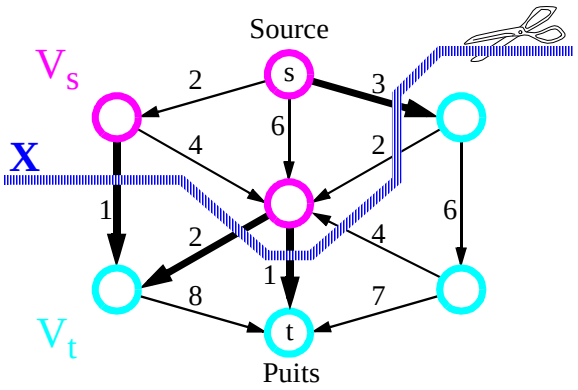
$$O((n + m)\, n\, m) = O(n^3)$$

- n est le nombre de sommets et m le nombre d'arcs

- **Note:**
  - L'algorithme d'Edmonds et Karp s'exécute en un temps $O(n^3)$ peu importe la valeur du flot maximal
  - Le pire des cas ne survient habituellement pas en pratique.

Qu'est-ce qu'une coupe?

- Une partition des sommets $X=(V_s,V_t)$, avec $s \in V_s$ et $t \in V_t$



- Capacité X = (V_s,V_t):
  - $c(X) = \sum_{v \in V_s,\, w \in V_t} \text{capacité}(v,w) = (1+2+1+3) = 7$

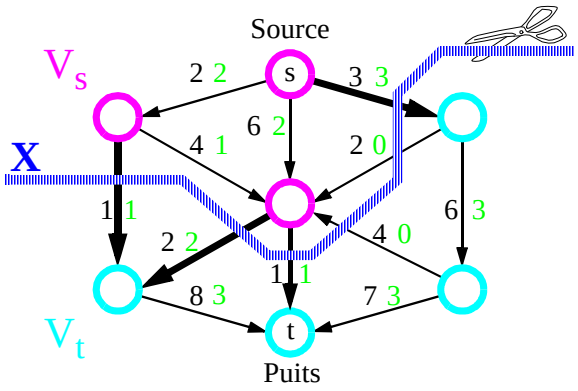
- La partition coupée (X dans notre cas) doit passer au travers du réseau entier, et ne peut pas passer au travers d'un sommet.

Flot maximal et coupe minimale

(valeur du flot maximal)

=

(capacité de la coupe minimale)



- Valeur du flot maximal: 7 unités de flot
- Capacité de la coupe minimale: 7 unités de flot

Pseudo-code

Algorithme: MinCut(N) basé sur Edmonds-Karp
Entrée: réseau N
Sortie: Séquence s d'arcs dans la coupe minimale de N

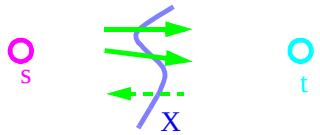
Partie I: Mise en place de Edmonds-Karp (page 56)

Partie II: Boucle
repeat
  Les sommets de N_f ne sont pas marqués
  p ← Marking-BFS(s,t,N_f)
  // une modification de BFS qui marque tout
  // sommet lorsqu'il est visité
  if (chemin *p* trouvé)
    y envoyer le plus grand flot possible
until (pas de chemin augmentant)

Partie III: Calcul de la séquence MinCut
s ← new Sequence()
foreach sommet *u* ∈ Sommets Marqués
  foreach sommet *v* ∈ Sommets Non-Marqués
    if (N a un arc e de *u* à *v*)
      s.**add**(e)

Pourquoi est-ce une coupe minimale?

- Soit f un flot de valeur |f| et X une coupe de capacité |X|. Alors, |f|<=|X|.



- Ainsi, si nous trouvons un flot f* de valeur |f*| et une coupe X* de capacité |X*|=|f*|, alors f* doit être le flot maximal et X* la coupe minimale.

- Nous avons vu que, à partir du flot obtenu via l'algorithme de Ford et Fulkerson, nous pouvons construire une coupe à capacité égale à la valeur du flot. Donc,
  - nous avons donné une preuve alternative que l'algorithme de Ford et Fulkerson génère un flot maximal
  - nous avons montré comment construire une coupe minimale