2019-2020, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

LIFAP6: Algorithmique, Programmation et Complexité

Chaine Raphaëlle (responsable semestre automne)
E-mail : raphaelle.chaine@liris.cnrs.fr
http://liris.cnrs.fr/membres?idn=rchaine

- Il s'agit du temps d'exécution d'un algorithme qui divise un problème de taille n en a≥1 sous-problèmes de taille n/b avec b>1
(stratégie **« diviser pour régner »**)

$$T(n) = aT(\frac{n}{b}) + f(n) \qquad \text{n>0}$$

f(n) décrit le coût de la division du problème en a sous-problèmes et de la recomposition des résultats

Master Theorem

1. Si il existe $\varepsilon > 0$ t. que $f(n) = O(n^{\log_b(a)-\varepsilon})$
   alors $T(n) = \Theta(n^{\log_b(a)})$

2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log_b(n))$

3. Si il existe $\varepsilon > 0$ t.que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$
   et il existe $c < 1$ t. que $af(\frac{n}{b}) \leq cf(n)$
   pour n grand alors $T(n) = \Theta(f(n))$

Intuition de preuve au tableau

Master Theorem

1. Si il existe $\varepsilon > 0$ t. que $f(n) = O(n^{\log_b(a)-\varepsilon})$
   alors $T(n) = \Theta(n^{\log_b(a)})$

   Coût de la décomposition et recomposition négligeable devant le coût lié à la récursion

2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \log_b(n))$

   Coût de la décomposition et recomposition similaire à celui lié à la récursion

3. Si il existe $\varepsilon > 0$ t.que $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$
   et il existe $c < 1$ t. que $af(\frac{n}{b}) \leq cf(n)$
   pour n grand alors $T(n) = \Theta(f(n))$

   Coût de la décomposition et recomposition non négligeable devant le coût lié à la récursion, … mais maîtrisé!

– Construisons ensemble une procédure dont la complexité augmente avec la valeur d'un entier n passé en paramètre.
– On souhaite connaître son comportement asymptotique quand n augmente.
– Cette fonction contient des instructions d'affichage (« Coucou ») sur la sortie standard et répond à une stratégie récursive.
  - Cette fonction s'appelle elle-même 6 fois (a=6) pour une valeur de n divisée par b=3.
  - Les instructions d'affichage sont situées en dehors du cas d'arrêt

```
void proc(int n)
  {
       for(int i=0;i<6;i++)
          proc(n/3);
       for(int j=0;j<n;j++)
         affiche(« coucou »);
    }
```

– Application du Master Theorem

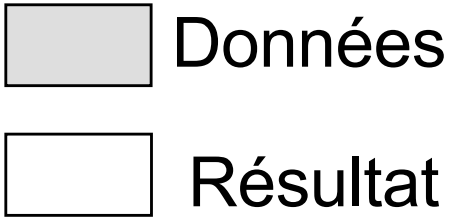
- f(n)=n affichages de « Coucou » hors des appels récursifs emboîtés.
- $\log_3(6) = 1.6309\ldots$ (ie de la forme 1+epsilon)
- Cas 1 du Master Theorem : $T(n)=\theta(n^{\log_3(6)})$

Retour sur le tri fusion

- Remarque : La version récursive du tri fusion étudiée aujourd'hui diffère de celle que vous aviez découverte en LIFAP3.
- Il s'agissait d'une version où on ne coupait pas la séquence S des éléments à trier de la même manière!
  - La taille de la séquence n'était pas connue à priori
  - Version adaptée au tri des séquences rangées dans des fichiers ou des listes chaînées

- Il s'agissait d'un algorithme itératif qui traitait la séquence initiale comme une séquence de monotonies de longueur 1
- A chaque passage dans la boucle
  - Répartition des monotonies dans 2 autres séquences S1 et S2, à raison d'1 monotonie sur 2 (éclatement)
  - Fusion de la kième monotonie de S1 avec la kième monotonie de S2 et réécriture dans S qui contient ainsi des monotonies de longueur double (sauf peut-être la dernière)

```
procédure TriFusionItératif(
      donnée-résultat S : séquence)
variables S1,S2 : séquence
début
      lgmono←1
      répéter
         Éclatement( S , lgmono , S1 , S2 )
         Fusion( S1 , S2 , lgmono , S , nbmono )
         lgmono=lgmono*2
      tantque nbmono≠1
fin
```



- Eléments supplémentaires de comparaison
  - TriFusionItératif n'est pas un tri sur place (besoin de séquences supplémentaires, le tri ne se fait pas sur place)
  - A votre avis TriFusionRec est-il un tri sur place?

– De même que TriFusionItératif, TriFusionRec n’est pas un tri sur place :
  - Fusionner(tab,p,q,r) nécessite un espace supplémentaire proportionnel à la taille du sous-tableau à réorganiser
  - il ne faut pas oublier l’espace requis pour gérer la récursion!

Types abstraits (rappels)

- **Type abstrait de données (TAD):**
  - Ensemble des valeurs codées par le type
  - Ensemble des opérations que l'on peut effectuer sur les valeurs et variables de ce type
- Il n'est pas nécessaire de connaître la manière dont les valeurs et les opérations sont codées pour pouvoir les utiliser.
- On utilise les types de façon abstraite, sans connaître leur implantation interne
- Ex : Entier + - * /

- **Description des Types Abstraits dans le cadre de modules**
- Module :
  Regroupe un ensemble de définitions de **types**, de **procédures** et de **fonctions** qui forment un ensemble cohérent (éventuellement aussi des constantes et des variables globales).
  - **Interface du module (partie visible)**
    Présentation claire des **types**, **procédures**, **fonctions**, constantes et variables offertes par le module
  - **Implantation du module (partie cachée)**
    Mise en œuvre des **types**, **procédures** et **fonctions** proposées dans l'interface. Définition des constantes et variables globales du module.

Retour sur le type abstrait **séquence (ou liste)**

- Les structures linéaires / séquences permettent de stocker une suite d'élément
- Cette suite peut être amenée à évoluer…
- Il existe 2 sortes de Séquences (ou Listes)
  - Les listes/séquences avec accès récursif :
    - on n'accède pas directement à l'emplacement du ième élément
    - l'accès au i+1 ème élément se fait à partir de l'accès au ième élément
  - Les listes/séquences avec accès itératif
    - Il existe une fonctionnalité d'accès direct au ième élément
    - **Si** les éléments sont en outre garantis être **contigüs (ie. consécutifs)** on obtient le Type Abstrait Tableau Dynamique

Interface du TAD Liste Récursive

- **module** ListeRec
  - **importer**

    Module Element, Entier
  - **exporter**

    Type ListeRec, Emplacement

    procédure initialisation(Résultat s : ListeRec)
    {Préc° : s- non initialisée , Postc° : s+ ListeRec vide}

    procédure ajoutEltTete(Donnée-résultat s : ListeRec,
    Donnée e : Element)
    {Préc° : s- initialisée, Postc° : e inséré en 1ère position}

    pointeur sur Emplacement **fonction emplacementTete** (s : ListeRec)
    {Préc° : s initialisée, Résultat : adresse 1er Emplacement, nul sinon}

    pointeur sur Emplacement **fonction emplacementSuivant** (s : ListeRec,
    p : pointeur sur Emplacement)
    {Préc° : s initialisée, p adresse d'un Emplacement de s
    Résultat : adresse Emplacement suivant, nul sinon}

    Element **fonction contenuEmplacement** (s : ListeRec, p : pointeur sur Emplacement)
    {Préc° : s initialisée non vide, p adresse d'un Emplacement de s
    Résultat : Element contenu dans l'Emplacement concerné}

    booléen **fonction testerListeVide**(s : ListeRec)
    {Préc° : s initialisée, Résultat : vrai si s vide, faux sinon}

    procédure suppressionEltTete(Donnée-résultat s : ListeRec)
    {Préc° : s- initialisée non vide Postc° : le 1er elt a disparu de s}

finmodule

Les « Fichiers » répondent à cette interface

Interface du TAD Séquence Itérative

- **module** SequenceIter
  - **importer**

    Module Element, Entier
  - **exporter**

    Type SequenceIter

    procédure initialisation(Résultat s : SequenceIter)
    {Préc° : s- non initialisée , Postc° : s+ SequenceIter vide}

    Entier **fonction longueur**(s : SequenceIter)
    {Préc° : s initialisée, Résultat : nombre d'elts présents dans s}

    Element **fonction consulterIèmeElt** (s : SequenceIter, i : Entier)
    {Préc° : s initialisée, 0<i<=longueur(s) , Résultat : ième Element}

    procédure insertionElt(Donnée-résultat s : SequenceIter,
    Donnée e : Element, i : Entier)
    {Préc° : s- initialisée, 0<i<=longueur(s)+1, Postc° : e inséré en ième pos}

    procédure suppressionElt(Donnée-résultat s : SequenceIter,
    Donnée i : Entier)
    {Préc° : s- initialisée, 0<i<=longueur(s), Postc° : ième elt disparu de s}

    pointeur sur Element **fonction rechercheElt**(s : SequenceIter, e :Element)
    {Préc° : s initialisée , Résultat: adresse de e dans s, nul sinon}

finmodule

Attention : Ne pas oublier d'ajouter aux modules ListeRec et SequenceIter :

procédure **testament**(Donnée-résultat s : Séquence)

{Préc° : s- initialisée , Postc° : s+ prêt à disparaître}

procédure **initialisation**(Résultat s1 : Séquence,
Donnée s2 : Séquence)

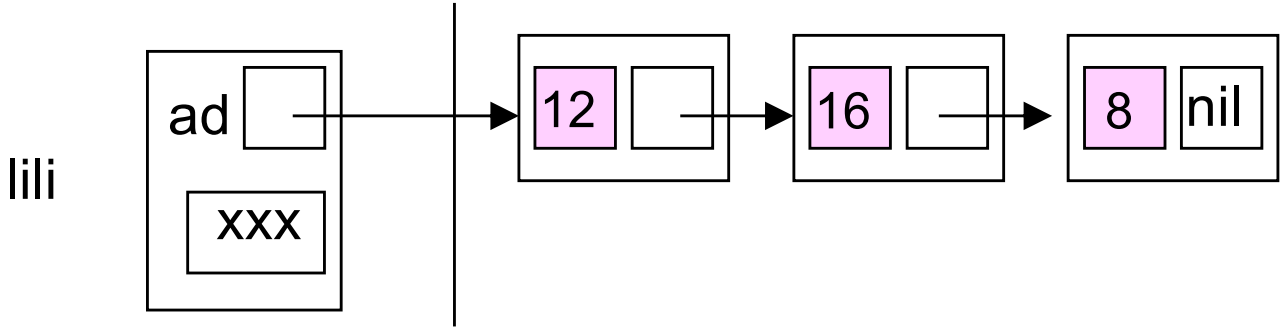
{Préc° : s1- initialisée , Postc° : s1+ est une copie de s2}

procédure **affectation**(Donnée-Résultat s1 : Séquence,
Donnée s2 : Séquence)

{Préc° : s1- initialisée , Postc° : s1+ est une copie de s2}

Implantations possibles

- Utilisation d'une structure chaînée

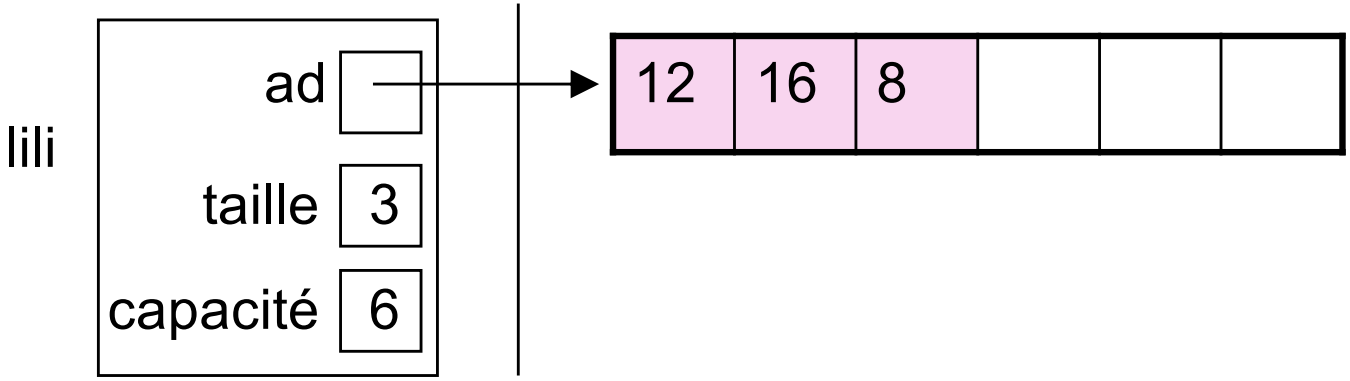


  - Eléments dispersés dans l'espace mémoire
  - Eléments encapsulés dans un emplacement (ou Cellule) qui contient également un accès à la place suivante
  - Attention : Cette implantation n'est pas valable si on désire obtenir le TAD Tableau Dynamique

xxx : nombre d'Elements, pointeur sur le dernier emplacement, etc.

Implantations possibles

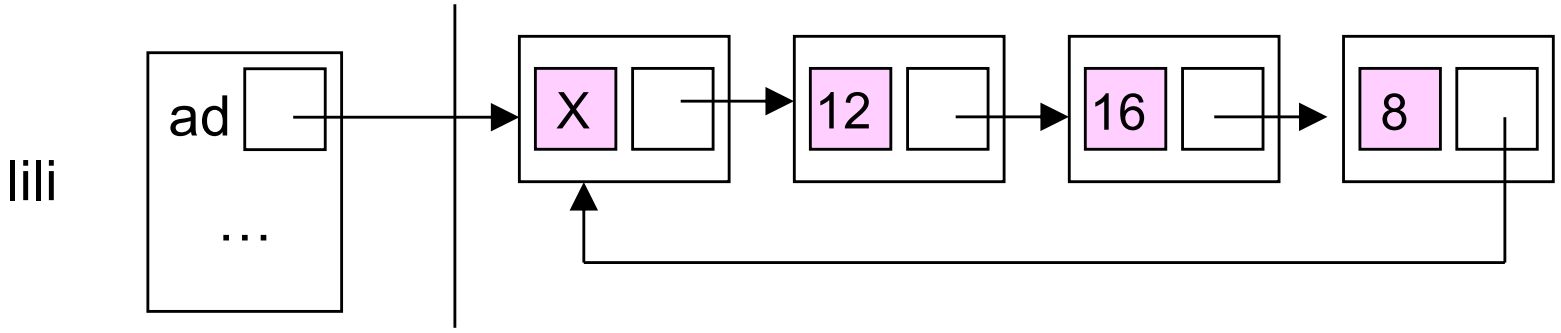
- Utilisation d'une représentation contiguë



  - Eléments rangés les uns à la suite des autres dans la mémoire
  - Nécessité de savoir où se finit la séquence (tableau dynamique)
    - Stockage du **nombre d'éléments**
    - Pour éviter les déménagement d'Eléments à chaque insertion on peut prévoir un espace plus grand (**capacité** jusqu'à laquelle il n'y aura pas de réorganisation de la mémoire)

Implantations possibles

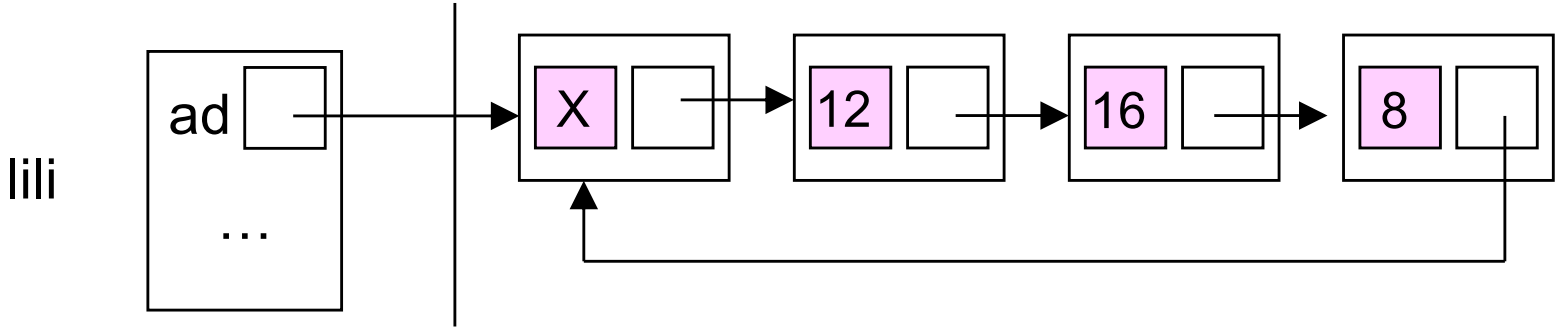
- Variantes :
  - Structure chaînée circulaire (avec éventuellement une cellule bidon)



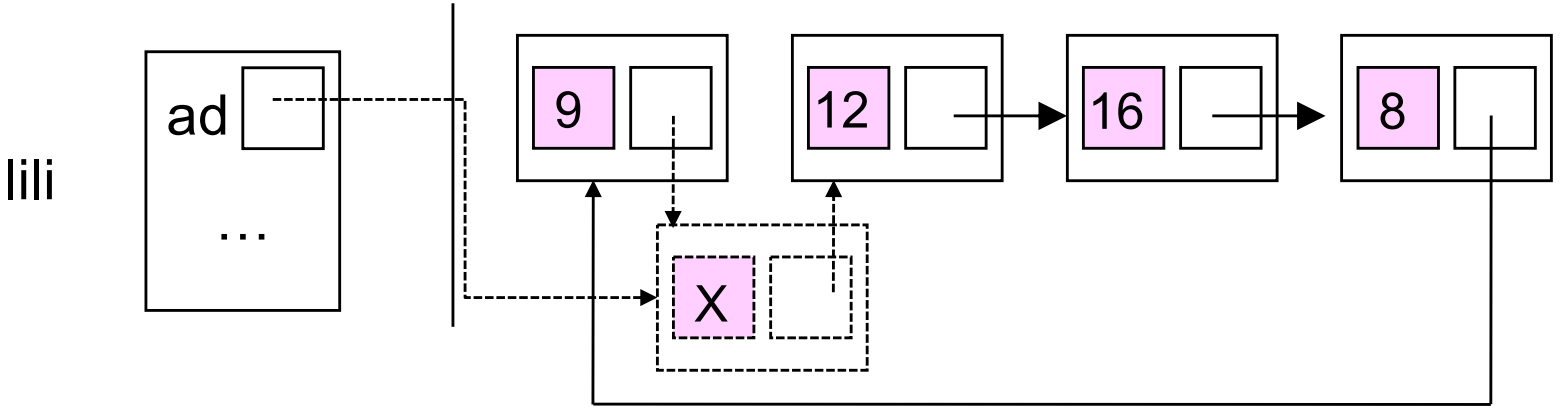
  - On remplace le pointeur à *nil* de la dernière cellule de la liste par un pointeur sur une cellule bidon, laquelle contient un pointeur sur la première cellule
  - Utile pour insérer en queue sans parcourir toute la liste

Implantations possibles

- Rappel astuce :

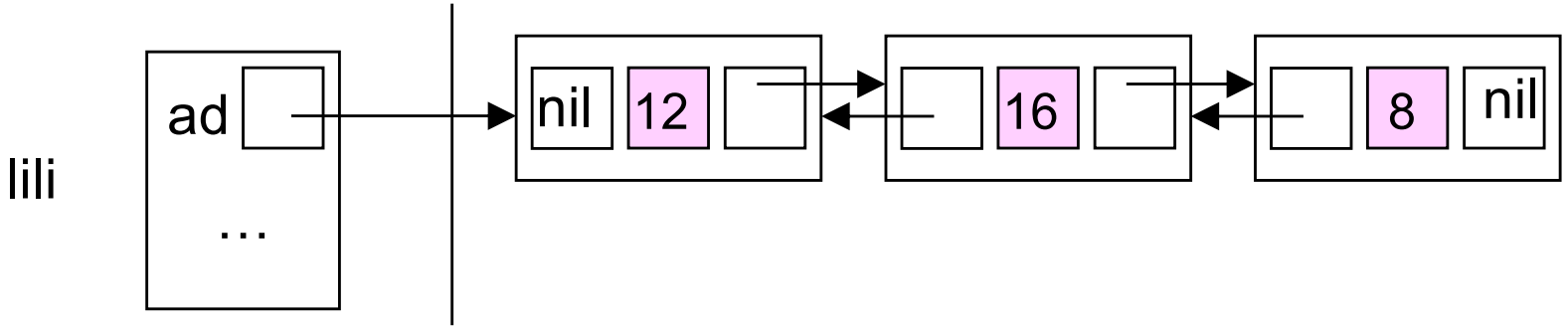


  - Utilisation de la cellule bidon pour y stocker le nouvel Elément
  - Insertion de 9 en queue de la liste <12, 16, 8>



Implantations possibles

- Variantes :
  - Structure doublement chaînée
    - Chaque emplacement comporte un accès à l'emplacement suivant et un accès au précédent



    - Utile pour les parcours à l'envers

Complexité opérations sur les séquences

- En fonction du nombre n d'Elements

• Quizz	Structure chaînée	Structure chaînée circulaire (avec elt bidon)	Représentation contiguë (Tableau Dynamique)
ajoutEnTete			
ajoutEnQueue			
insertionIèmeElt			
rechercheElt (liste non triée)			
rechercheElt (liste triée)			
insertionElt (liste triée)			

Complexité opérations sur les séquences

	Structure chaînée	Structure chaînée circulaire (avec elt bidon)	Représentation contiguë (Tableau Dynamique)
ajoutEnTete	Θ(1)	Θ(1)	Θ(n) décalage
ajoutEnQueue	Θ(n)	Θ(1)	Θ(1) (parfois déménagement Θ(n))
insertionIèmeElt	Θ(i)	Θ(i)	Θ(n-i) (parfois déménagement Θ(n))
rechercheElt (liste non triée)	Θ(n) en moyenne	Θ(n) en moyenne	Θ(n) en moyenne
rechercheElt (liste triée)	**Θ(n) en moyenne**	**Θ(n) en moyenne**	**Θ(lg_2n) en moyenne** Dichotomie
insertionElt (liste triée)	**Θ(n) en moyenne**	**Θ(n) en moyenne**	**Θ(n) en moyenne** Dichotomie+décalage

Rêvons un peu…

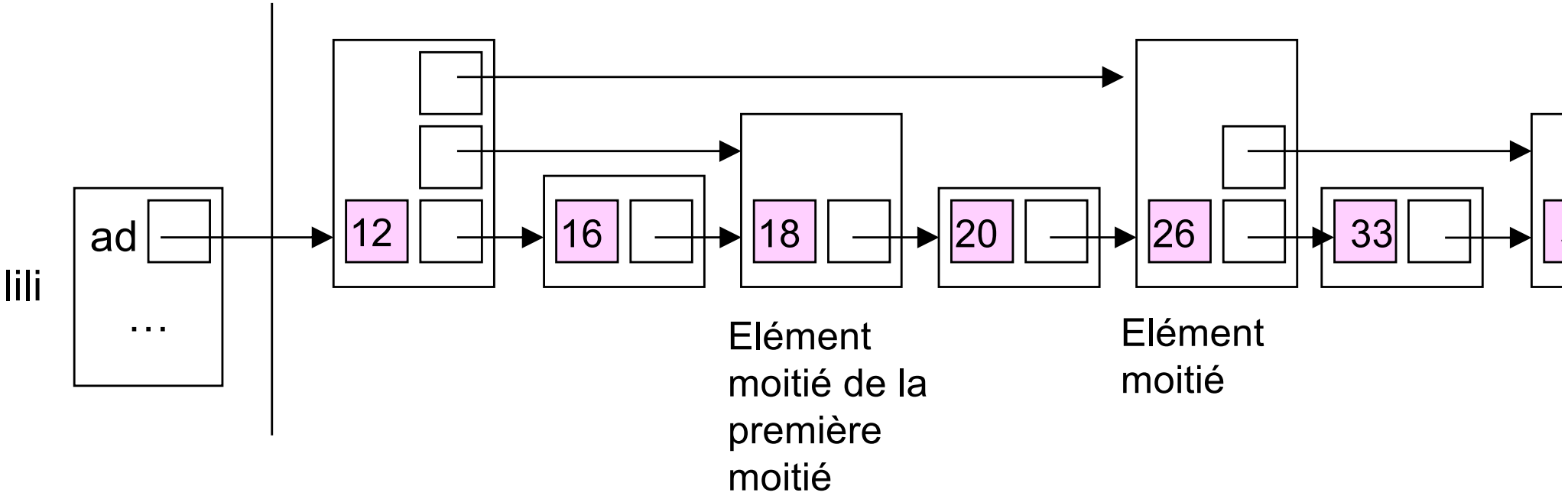
- Si on pouvait bénéficier du pouvoir de la recherche dichotomique avec les implantations chaînées
  - On effectuerait les recherches d'éléments et les insertions dans les listes triées en $\theta(\lg_2 n)$
  - Les performances seraient donc meilleures qu'avec une implantation contiguë pour laquelle l'insertion dans une liste triée se fait en $\theta(n)$

Rappel

- Recherche dichotomique d'un élément dans une séquence triée avec accès direct :
  - Positionnement de l'élément par rapport à l'élément moitié.
  - La recherche d'un élément dans une séquence de taille n se ramène alors à la recherche dans la première ou seconde moitié
  - **Equation de récurrence :**
    T(n)=T(n/2) + T(accès à l'élément moitié)+T(comparaison)
  - Master Theorem, cas 2 (a=1, b=2, $lg_2(1)=0$) :
    - $T(n)=lg_2(n)$

Le rêve est-il inaccessible?

- Pour bénéficier du pouvoir de la dichotomie sur les listes chaînées, il faudrait :
  - avoir un accès direct à l'élément milieu,
  - puis un accès direct à l'élément milieu de la première moitié,
  - à l'élement milieu de la seconde moitié,
  - et ainsi de suite...
- Cela peut-être réalisé à l'aide d'ajouts de pointeurs supplémentaires
(raccourcis vers l'avant)



- Exemple d'une Liste de 10 éléments (pas la place de l'écrire en entier sur ce slide!)
- Une telle structure de donnée serait dure à maintenir au fur et à mesure des insertions/retraits d'éléments…

- Il existe une variante de cette idée basée sur un principe probabiliste : les ***skip-lists***
- Les *skip-lists* entrent dans la catégorie des algorithmes et des structures de données randomisées
- Skip-Lists :
  - Hiérarchie de séquences, chaque séquence correspondant à un échantillonnage de la précédente
  - Probabilité *p* pour un élément d'un niveau, d'appartenir au niveau au dessus
- On reviendra sur cette structuration efficace en TD et en TP ☺