2022-2023, Semestre d'automne
L3, Licence Sciences et Technologies
Université Lyon 1

LIFAPC: Algorithmique, Programmation et Complexité

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

1

1

Spécificités des algorithmes itératifs et récursifs

- Outils de preuve
  - Algorithmes itératifs :
    - Invariant de boucle permettant de progresser vers une condition d'arrêt
    - Satisfaction d'une assertion au moment où la condition d'arrêt est atteinte
  - Algorithmes récursifs
    - Raisonnement par récurrence

122

122

- **Exemple de preuve d'un algorithme itératif**
- **procédure** rechercheDansSequence(
  donnée s : séquence, e: element,
  résultat i : position, b :booléen)
  {précondition : s, e initialisés
  postcondition : si b contient vrai
  alors e est en ième position de s,
  sinon e n'est pas dans s}
  début
  i←1, b ←faux
  tantque (i ≤ taille(s)) **etalors** (e≠s[i]) **faire**
  i++
  fintantque
  si i ≤ taille(s) **alors**
  b←true
  finsi
  fin

123

123

- Invariant de boucle
- i←1, b ←faux
  tantque (i ≤ taille(s)) **etalors** (e≠s[i]) **faire**
  {Assertion : Au kième passage éventuel dans la boucle, k=i et pour tout j t.que 1 ≤ j < k on a s[j] ≠ e}
  i++
  fintantque
- Preuve de l'invariant de boucle
- Récurrence
  - Vrai pour k=1
  - Supposons le vrai pour k=K
  - Si on passe une K+1 ième fois dans la boucle, cela signifie que K+1 ≤ taille(s) et que s[K+1] ≠ e donc s[j] ≠ e pour tout 1 ≤ j < K+1, ce qui clôt la récurrence

124

124

- Il faut également montrer que le corps de la boucle permet de progresser vers la condition d'arrêt
  - Au K^{ième} passage dans la boucle, i augmente strictement : on ne pourra pas passer une infinité de fois dans la boucle, puisqu'une des conditions d'arrêt sera atteinte quand i excèdera taille(s)
- A la sortie de la boucle :
  - 1er cas : si i> taille(s) alors l'invariant du dernier passage dans la boucle assure que pour 1 ≤ j ≤ taille(s) on a s[j] ≠ e, donc e n'est pas dans s, ce qui est cohérent avec le fait que b contienne faux
  - 2ème cas : si i ≤ taille(s) alors s[i] vaut e ce qui est cohérent avec l'affectation de vrai à b

125

125

- **Exemple de preuve d'un algorithme récursif**
- Tri fusion d'un tableau
- **procédure** TriFusionRec(
  donnée-résultat tab : tableau[1..n] d'Elements,
  données p, r : 1..n)
  {précondition : aucune,
  postcondition : tab trié entre les indices p et r}
  variable
  q : 1..n
  début
  si p<r **alors**
  q←(p+r)/2
  TriFusionRec(tab,p,q) tri 1ère moitié
  TriFusionRec(tab,q+1,r) tri 2nde moitié
  Fusionner(tab,p,q,r)
  finsi
  fin

p			q	q+1			r
1	2	3	4	5	6	7	8
24	56	2	26	10	100	45	9

126

126

- Raisonnement par récurrence :
  - Pour un(e partie de) tableau de taille 0 ou 1, l'algorithme est correct
  - On suppose que l'algo est correct pour un tableau de taille m ≥1, montrons qu'il est correct pour un tableau de taille m+1
    - Comme m+1 ≥ 2 on a p≤q<r
      les 2 appels récursifs se font donc sur des tableaux de taille inférieure à m+1
    - Sous réserve que la procédure de fusion soit correcte, l'algorithme de tri fusion sera donc correct

127

127

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 + - * /

143

143

- **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.

144

144

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

- Les structures linéaires / séquences permettent de stocker une suite d'élément
- Cette liste 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 **contigus** on obtient le Type Abstrait Tableau Dynamique

145

145

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

146

146

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

147

147

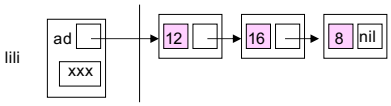
Attention : Ne pas oublier d'ajouter aux modules ListeRec et SequenceIter :

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}

148

Implantations possibles

- Utilisation d'une structure chaînée



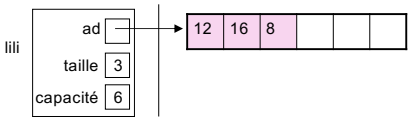
  - Eléments dispersés dans l'espace mémoire
  - Eléments encapsulés dans une structure emplacement (ou Cellule) qui contient également un accès à la place suivante
  - Attention : Cette implantation n'est pas valable si on désire obtenir le TAD Tableau Dynamique

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

149

Implantations possibles

- Utilisation d'une représentation contiguë

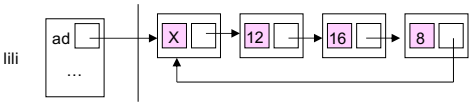


  - Eléments rangés les uns à la suite des autres dans la mémoire
  - Nécessité de savoir où se finit la séquence (tableau dynamique)
    - Stockage du **nombre d'éléments**
    - Pour éviter les déménagement d'Eléments à chaque insertion on peut prévoir un espace plus grand (**capacité** jusqu'à laquelle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

150

Implantations possibles

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

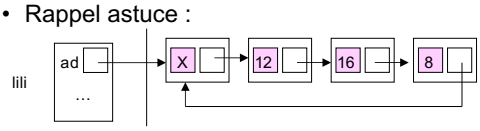


  - On remplace le pointeur à *nil* de la dernière cellule de la liste par un pointeur sur une cellule bidon, laquelle contient un pointeur sur la première cellule
  - Utile pour insérer en queue sans parcourir toute la liste

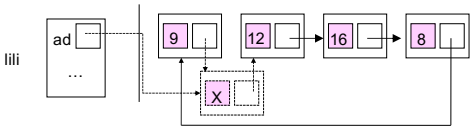
151

Implantations possibles

- Rappel astuce :



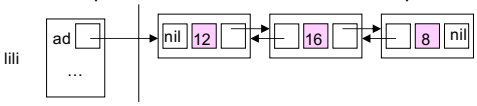
  - Utilisation de la cellule bidon pour y stocker le nouvel Elément
  - Insertion de 9 en queue de la liste <12, 16, 8>



152

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

153

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)			

154

154

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 + déménagement parfois
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

155

155

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 séquences/listes triées en θ(lg_2n)
  - Les performances seraient donc meilleures qu'avec une implantation contiguë pour laquelle l'insertion dans une séquence/liste triée se fait en θ(n)

156

156

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 :**
    $T_D(n)=T_D(n/2)+T(\text{accès à l'élément moitié})+T(\text{comparaison})$
  - Master Theorem, cas 2 (a=1, b=2, $lg_2(1)=0$) :
    - $T_D(n)=lg_2(n)$

157

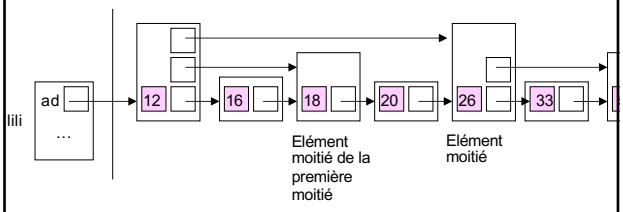
157

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)

158

158



- 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…

159

159

- 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 ☺

160

160