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

- Liste des Types Abstraits que vous connaissez?
  - File
  - Pile
  - Séquence (ou Liste)
    Y compris le cas particulier du Tableau Dynamique
  - Séquence Triée
  - …

207

207

TAD Ensemble

- Pour un ensemble d'éléments, l'ordre des éléments n'a pas d'importance.
- On veut pouvoir
  - tester l'appartenance d'un élément à un ensemble,
  - ajouter ou supprimer un élément,
  - tester si un ensemble est vide, …

Le plus efficacement possible!

208

208

Interface du TAD Ensemble

- **module** Ensemble
  - **importer**
    Module Element
  - **Exporter**
    Type Ensemble
    procédure initialisation(Résultat f : Ensemble)
    {Préc° : f- non initialisé , Postc° : f+ Ensemble vide}
    procédure ajouterElt(Donnée-résultat f : Ensemble, Donnée e : Element)
    {Préc° : f- initialisé, Postc° : e présent dans f+}
    procédure supprimerElt(Donnée-résultat f : Ensemble, Donnée e : Element)
    {Préc° : f- initialisé, Postc° : e non présent dans f+}
    booléen **fonction rechercherElt** (f : Ensemble, e : Element)
    {Préc° : f- et e initialisé, Résultat : vrai si e dans f, faux sinon}
    entier **fonction nombreElement**(f : Ensemble)
    {Préc° : f initialisé, Résultat : nbre d'éléments contenus dans f}
    Éventuellement Ensemble **fonction union**(f1 : Ensemble, f2 : Ensemble)
    {Préc° : f1 et f2 initialisés, Résultat : elts appartenant à f1 ou f2}
    Ensemble **fonction intersection**(f1 : Ensemble, f2 : Ensemble)
    {Préc° : f1 et f2 initialisés, Résultat : elts appartenant à f1 et f2}
    procédure testament(Donnée-résultat f : Ensemble)
- **Finmodule**

209

209

Implantations possibles

- Représentation par des tableaux de booléens
  - **Lorsque l'univers des valeurs possibles des Eléments sont en nombre fini et raisonnable**, et peuvent permettre d'indicer un tableau,
  - Utilisation d' **un tableau**
    - contenant vrai dans chaque case correspondant à un Elément de l'Ensemble,
    - faux sinon
  - Quelle est la complexité des opérations sur les ensembles avec cette implantation?

210

210

Implantations possibles

- Représentation par des tableaux de booléens
  - **Lorsque l'univers des valeurs possibles des Eléments sont en nombre fini et raisonnable**, et peuvent permettre d'indicer un tableau,
  - Utilisation d' **un tableau**
    - contenant vrai dans chaque case correspondant à un Elément de l'Ensemble,
    - faux sinon
  - Quelle est la complexité des opérations sur les ensembles avec cette implantation?
    - TEMPS CONSTANT POUR L'AJOUT, LA RECHERCHE ET LA SUPPRESSION

211

211

Autres implantations possibles

- Représentation d'un Ensemble par une Séquence/Liste de ses Eléments
- Si le type des Eléments bénéficie d'une relation d'ordre total,
  - on peut utiliser une Séquence/Liste triée (par exemple implantée avec une *skip-list*)
  - ou un Arbre Binaire de Recherche

212

212

Notion de clé

- En général, quand on stocke et recherche des éléments complexes, les opérations de comparaison ne sont pas effectuées directement sur les éléments, mais sur une clé unique qui leur est associée.
  - Exemple : numéro de sécurité sociale d'une personne.
- Avantages :
  - Il est souvent plus rapide de comparer 2 éléments sur la base de leurs clés, que sur la base des informations qui leur sont associées.
  - Il est possible que le type Elément ne bénéficie pas d'une relation d'ordre total, mais que ce soit le cas pour le type Clé
  - On peut alors bénéficier de structurations performantes

Un élément = Une clé et des informations associées

213

213

TAD Table

- Table : Ensemble de clés (auxquelles on peut associer une valeur)
- Vocabulaire :
  - Si la clé c est présente dans une table t
  - On dit qu'il existe une entrée dans la table t pour la clé c
- Chaque clé est (généralement) unique
  - Information identifiante et discréminante permettant de distinguer les entrées de la table

214

214

- Les opérations sur les Tables sont similaires à celles sur les Ensembles
  - Recherche, insertion et suppression de Clés/Eléments
- Opérations supplémentaires
  - Fournir l'information associée à une clé présente dans la table
  - Modifier l'information associée à une clé présente dans la table
  - La suppression de l'entrée correspondant à une clé doit s'accompagner de la suppression de l'information associée

215

215

Table de hachage et adressage dispersé

•Principe

–Gestion d'un ensemble C de clés dont les valeurs peuvent appartenir à un **univers U très grand**

–La **taille maximale** de la table C est connue et est **petite** par rapport à la taille de U

U : ensemble des clés possibles

C : ensemble des clés effectivement stockés dans la table

216

216

U : ensemble des clés possibles

C : ensemble des clés effectivement stockés dans la table

•Exemple :

– U est l'ensemble de tous les mots possibles de 10 lettres

– C est un ensemble de 7 mots de 10 lettres

– La taille de U $\left(card(U) = 26^{10} \approx 10^{13}\right)$ interdit de représenter C en réservant une place en mémoire pour chaque clé possible …

–On souhaiterait pourtant utiliser un tableau de taille raisonnable m>7 pour stocker C

1	2	3	4	5	6	7	8	m=9
"soleil"		"oui"	"Turing"	"y"	"LIF"	"le"	"La"	

217

217

- On utilise alors
  - une fonction de hachage **associant à chaque clé** x **un entier compris entre 1 et m** (m choisi plus grand que la taille maximale de C)
    $$h : U \to [1, m]$$
  - une table de hachage qui est un tableau de taille fixe m pour stocker les éléments
- h(x) est appelée valeur de hachage primaire
  - donne l'indice de la place de x dans le tableau T de m éléments
  - servira à vérifier si x appartient à T, à l'ajouter ou à le supprimer
  - Exemple : si x entier on peut utiliser h(x)=x%m

218

218

- Pour insérer une clé x dans une table :
  - On calcule h(x) et on range x dans la case d'indice h(x)
- Pour rechercher une clé x dans une table :
  - On calcule h(x) et on regarde si x est présente dans la case h(x)
- Exemple : Si la fonction h est telle que
  h("soleil") vaut 1
  h("Turing") vaut 4
  etc…

1	2	3	4	5	6	7	8	m=9
"soleil"		"oui"	"Turing"	"y"	"LIF"	"le"	"La"	

219

219

- Remarques
  - La taille m de la table est beaucoup plus petite que le nombre de clés possibles
  - Pas gênant **TANT QUE l'on respecte l'hypothèse que le nombre d'éléments à stocker est inférieur à m**
  - Fonction de hachage : relation directe entre une clé et l'adresse de la case où on va la ranger
  - Cette approche de stockage et de consultation dans une table permet de trouver une clé en un temps indépendant de la taille de la table (performance en θ(1))

220

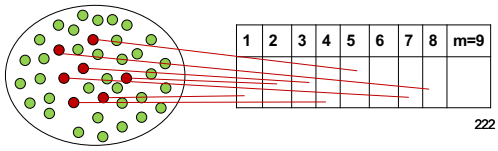
220

- Le choix de la fonction de hachage est fondamental :
  - Il faut distribuer les clés de U sur [1,m] de manière aussi uniforme que possible : proba(h(x) = i) = 1/m
  - le calcul de la fonction de hachage doit être rapide (temps constant)
- Collision dite primaire si la fonction de hachage fournit un même indice pour 2 clés différentes

221

221

- Même avec une bonne fonction de hachage, il est impossible d'éviter les collisions primaires :
  - Supposons que C ait n<m clés/éléments et que h soit une fonction de C dans [1, m] uniforme.
  - Quelle est la probabilité que h soit injective (valeurs différentes pour clés différentes)?



222

222

- P = m(m-1)…(m-n+1)/m^n
  - En effet m(m-1)…(m-n+1) fonctions de hachage sans collision sur les n clés à stocker
  - m^n comportements possibles pour la fonction de hachage sur les n clés à stocker
- Si m = 365 et n = 23, alors P<1/2
  - Si on réunit plus de 23 personnes, il y a plus d'une chance sur deux que deux d'entre elles soient nées le même jour du même mois!

223

223

- Il faut savoir gérer les collisions
- Certaines clés/éléments ne pourront pas être directement placés à l'emplacement désigné par leur fonction de hachage
- Méthodes de résolution des collisions
  - par calcul / sondage : on propose une nouvelle valeur de hachage
  - On parle alors d'**adressage ouvert** ou de **hachage fermé**
  - Il faut bien entendu que la séquence des case essayées pour y ranger une clé puisse être reproduite pour pouvoir ensuite la retrouver!

224

224

- Jeu de l'oie?

  - Tout se passe comme si on rangeait des clés dans un jeu de l'oie avec un dé électronique au comportement reproductible!
  - **Si la case visée est prise** on utilise le dé électronique qui nous donne le **nombre de pas** à effectuer pour tester une nouvelle case
  - La séquence des valeurs de pas fournies par le dé doit être reproductible
  - Il s'agit donc d'un dé pipé en fonction de la valeur de la clé et du nombre i d'essais

225

225

- Principe général de la **recherche** d'une clé x :
  - détermination de la première place possible pour x en utilisant la fonction de hachage h(x)
  - si cette place est vide, arrêt
  - sinon comparaison de la clé présente à cette place avec x
  - si x trouvé, arrêt
  - **sinon, passer à une autre place possible par calcul d'un décalage** et recommencer le même traitement …

226

226

- Recherche d'une clé x dans la table :
  - **recherche positive** : il existe une clé égale à x dans la table
  - **recherche négative** : il n'y a pas de clé égale à x dans la table

227

227

- Principe similaire pour l'**insertion** d'une clé x
  - Détermination de la première place possible pour x
  - si cette place est vide, on y range x (*et l'information associée*) puis arrêt
  - Sinon, si on a trouvé x, *modification éventuelle de l'information associée* puis arrêt
  - **sinon, passer à une autre place possible par calcul d'une autre adresse** et recommencer le même traitement …

228

228

Adressage ouvert : résolution des collisions par calcul

- On réalise des essais successifs (re-hachage) :

$$essai : U \rightarrow \{1,2,...,m\}^m$$

qui associe à chaque clé x de U une permutation de {1, 2, …, m} (CAS IDEAL)

$$essai(x) = (essai_1(x), essai_2(x), ..., essai_m(x))$$
$$\text{avec } essai_i(x) \neq essai_j(x) \text{ si } i \neq j$$

- Recherche de x dans le tableau T :
  on explore successivement les places $essai_1(x)$, puis $essai_2(x)$, …,
  jusqu'à ce qu'on trouve x
  ou qu'on arrive à une place vide
  ou à la fin des m essais

229

229

- Même démarche pour l'insertion :
  - si on trouve x, on ne fait rien ;
  - Sinon on le range dans la première place vide trouvée (dans l'ordre des essais);
  - Si ce n'est pas le cas au bout des m essais : le tableau est plein et l'insertion impossible!
- Avec l'adressage ouvert apparaît la notion de collision secondaire :
  - Une clé trouve sa place prise par une autre dont ce n'est pas la place conformément à la fonction de hachage …

230

230

- **Pas de rehachage** :
  - Nombre de cases à sauter pour passer d'une case à la suivante que l'on examine

    $Pas(i,c) = essai_{i+1}(c) - essai_i(c)$

231

231

procédure ajouter_HF(données x : clé,
données-résultat T: HTable,
résultat plein : booléen)

variables

$i, v : 1..m+1$ — i pour compter les essais, v pour l'indice des cases testées

début

$i \leftarrow 1$; plein $\leftarrow$ faux;

répéter

$v \leftarrow essai(i, x)$

$i \leftarrow i+1$

tant que $(vide(T, v) \neq vrai)$ et $(T[v] \neq x)$ et $(i \leq m)$

...

232

232

...

si $T[v] = x$ alors message : x déjà présent

sinon

si $vide(T, v) = vrai$ alors $T[v] \leftarrow x$

sinon plein $\leftarrow$ vrai

finsi

fin

233

233

1. Re-hachage linéaire

Lorsque il y a collision sur la case d'indice v, on essaie la case d'indice v+1 (modulo la taille du tableau)

Pas(i,c) = 1

On peut démontrer les résultats suivants :

$$Moy_{rech_s}(m,n) \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$Moy_{rech_i}(m,n) \approx \frac{1}{2}\left(1 + \frac{1}{(1+\alpha)^2}\right)$$

Par exemple, si $\alpha = 0.5$ (facteur de remplissage de la table correspondant à n/m), on obtient 1,5 examens de place en moyenne pour une recherche positive et 2,5 pour une recherche négative

234

234

- Inconvénient :
  - Accumulation des collisions dans des zones contiguës de la table …
    - Quand on commence à rentrer dans une zone embouteillée, le chemin pour en sortir reste toujours le même quel que soit l'endroit où on entre. On ne fait qu'allonger le chemin pour ceux qui arriveront par la suite....(puisqu'on s'installe ensuite à la sortie!!!)
    - Quand on commence à entrer dans cette zone (pour une recherche ou une insertion), on met de plus en plus de temps à en sortir au fur et à mesure qu'elle grossit!
  - **Il est préférable de mieux dissocier les suites de collisions pour des valeurs de h(x) différentes**

235

235

2. *Re-hachage quadratique*
   - $essai_i(x)=(h(x)+(i-1)^2) \bmod m$
- Rappel :
  - Calcul des carrés successifs de i avec la suite vue au TD1
    - $a_0=0$, $d_0=1$
    - $a_{i+1}=a_i+d_i$
    - $d_{i+1}=d_i+2$

    qui calcule $a_i=i^2$
  - D'où l'itération permettant de passer d'un essai au suivant
    - essai = (essai+d) mod m
    - d←d+2
  - Cela revient à « marcher » avec des pas de plus en plus grands… Pas(i,x)=2i+1

- On considère que la table déborde si on ne trouve pas de place pour y stocker une clé.
- On peut choisir de s'arrêter lorsqu'on se retrouve à réexaminer une position h(x)
- Cela ne signifie pas forcément que l'on a examiné tous les emplacements de la table…
- Choisir de préférence m>2 premier :
  - En cas de débordement, ne permet pas toujours de visiter toutes les entrées de la table…
  - Néanmoins, si on effectue (m+1)/2 essais avant de jeter l'éponge, on aura visité (m+1)/2 entrées distinctes de la table, et tout essai supplémentaire correspondra à une visite multiple

3. Double hachage

$$essai_i(x) = (h(x) + d(x) * (i-1)) \bmod m$$

- d(x) doit être telle que pour tout élément x de U, la suite des m essais considère bien toutes les places du tableau
- ceci n'est le cas que si d(x) est premier avec m pour tout x

on peut choisir
- Choix 1 : m premier et d à valeurs dans [1,m-1]
- Choix 2 : $m = 2^p$ et d(x) = 2d'(x)+1, où d' est à valeurs dans $[0,2^p - 1]$

- Cela revient à ce que les éléments « marchent » avec des largeurs de pas différents Pas(i,x)=d(x)

- Quid de la suppression dans le cas d'un adressage ouvert?
  - Problème plus compliqué que l'insertion : on peut avoir une réorganisation importante de la table
  - **plutôt que de supprimer la clé, on marque son emplacement comme libéré** (différent de vide!)
- Lors d'une insertion, on pourra occuper un espace marqué comme **libéré**
- Lors d'une recherche, le passage sur un espace marqué comme libéré ne marque pas la fin de la recherche :
  - on essaye l'emplacement suivant en suivant la procédure classique de résolution des collisions

- Il existe une seconde approche pour gérer les collisions
- Par chaînage des clés en conflit dans une même entrée de la table.
  - On parle d'**adressage fermé (hachage ouvert)**

- Principe général de la **recherche** d'une clé x :
  - détermination de la première place possible pour x
  - si cette place est vide, arrêt
  - sinon comparaison de la clé présente à cette place avec x
  - si x trouvé, arrêt
  - **sinon, passer à une autre place possible par chaînage** et recommencer le même traitement …

- Principe similaire pour l'**insertion** d'une clé x
  - Détermination de la première place possible pour x
  - si cette place est vide, on y range x (*et l'information associée*) puis arrêt
  - Sinon, si on a trouvé x, *modification éventuelle de l'information associée* puis arrêt
  - **sinon, passer à une autre place possible par chaînage** et recommencer le même traitement …

242

242

- **Adressage fermé : résolution des collisions par chaînage**

  On chaîne entre eux les éléments en collision
  - soit dans une zone de débordement à l'extérieur du tableau de hachage,
  - soit à l'intérieur du tableau de hachage (hachage coalescent)

1. Hachage avec chaînage séparé

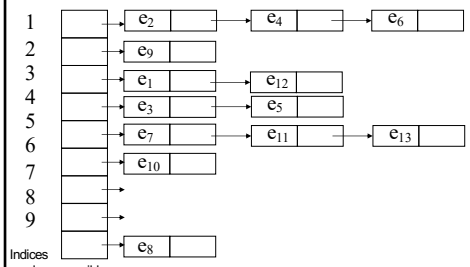
  - Les éléments sont chaînés entre eux *à l'extérieur* du tableau de hachage
  - Algorithmes de recherche, création et suppression très proches de ceux sur les ensembles représentés par chaînage

243

243

- *Exemple* : on insère les éléments de clés $e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}, e_{13}$ ayant pour valeur de hachage

  3 , 1, 4, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5



244

244

- ***Complexité en moyenne***
  - *cas d'une recherche négative*
  - h étant uniforme, il y a la même probabilité 1/m d'effectuer la recherche dans chacune des listes
  - Soit λ_i la longueur de la liste L_i
  - le coût moyen d'une recherche négative dans la table T est

$$rech\_(T) = \frac{1}{m}\sum_{i=1}^{m}\lambda_i$$

  - comme il y a en tout n éléments, $\sum_{i=1}^{m}\lambda_i = n$ donc

$$rech\_(T) = \frac{n}{m} = \alpha$$

245

245

- *cas d'une recherche positive*
  - le coût de recherche d'une clé est égal au coût de l'insertion de cette clé **plus 1**
  - si l'élément a été le (i+1)ième ajouté, son insertion a le même coût qu'une recherche négative parmi i éléments, soit en moyenne

$$Moy_{rech\_}(m,i) = \frac{i}{m}$$

246

246

- Si on considère que les n éléments présents ont la même probabilité d'être recherchés, on a :

$$Moy_{rech_+}(m,n) = \frac{1}{n}\sum_{i=1}^{n-1}\left(Moy_{rech\_}(m,i)+1\right)$$

d'où

$$Moy_{rech_+}(m,n) = \frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{i}{m}+1\right) = \frac{n(n-1)}{2nm}+1 = \frac{\alpha}{2}-\frac{1}{2m}+1$$

la recherche ou l'insertion est en $\Theta(n/m)$, où n est le nombre d'éléments ($n < m$) et m le nombre de listes

247

247

2. *Hachage coalescent*

Cas où on ne peut pas allouer de mémoire dynamiquement

- Réservation a priori d'une zone contiguë de mémoire dans le tableau de taille m,
- Division en 2 zones :
  une zone d'adresses primaires de taille p
  et une réserve de taille r, telle que p + r = m

248

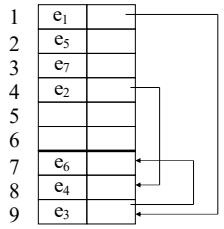
248

Exemple : on a un tableau de taille 9 dont les 3 dernières cases forment la réserve ; on ajoute successivement les éléments de clés

$e_1, e_2, e_3, e_4, e_5, e_6, e_7$

dont la valeur de hachage est

1, 4, 1, 4, 2, 1, 3



249

249

- Algorithme
- procédure ajouter_HCO(données x : clé,
  données-résultat T: HTable, résultat plein : booléen)

```
variable
    i, r : 1..m
début
  i ← h(x); r ← m; plein ← faux;
  si vide(T[i]) = vrai, alors
    T[i].val ← x; T[i].lien ← -1
  sinon //collision
    tant que (i ≠ -1) et (T[i].val ≠ x) et (T[i].lien ≠ -1) faire
      i ← T[i].lien
    fintantque
    ...
```

r : indice du prochain indice libre dans la réserve
i : indice de recherche d'une clé

250

250

```
    ...
    si T[i].val ≠ x alors
      tant que (r ≥ 1) et (vide(T[r]) = faux) faire
        r ← r-1
      fintantque
      si r ≥ 1 alors
        T[i].lien ← r; T[r].val ← x; T[r].lien ← NULL
      sinon plein ← vrai
      finsi
    finsi
  finsi
fin
```

251

251

- **Le graal : une bonne fonction de hachage!**
- **Exemple :**
  - Supposons que les clés soient des mots avec chaque lettre représentée en mémoire par une suite de 5 bits (*)
    A=00001, B=00010, C=00011, …
  - Problème : calculer à partir de ces clés un entier dans l'intervalle [0,m-1]

(*suffisant pour coder l'alphabet)

252

252

1. *Modulo*

- On calcule le reste de la division par m de la « valeur » de la clé

  h(x) = x mod m

  « Valeur »: interprétation de la séquence de bits de la clé comme le codage d'un entier

- *Exemple* :
  m = 37
  h(« OU ») = 501 mod 37 = 20

253

253

2. *Extraction de certains bits*

Si on extrait p bits de la représentation binaire de la clé, on se ramène à l'intervalle [0, 2^p – 1] en les concaténant et en considérant l'entier associé.

Exemple : on extrait les bits 1, 2, 6, 7, 11 et 12 à partir de la droite et on complète par des 0 à gauche

« OUA » = 011**11**|101**01**|000**01**
=> h(« OUA ») = **110101** = 32+16+4+1=$(53)_{10}$

254

254

- *Inconvénient* :
  - la valeur de hachage ne dépend pas de tous les bits de la représentation …

3.Compression

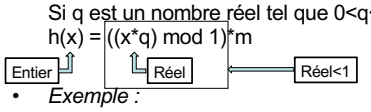
- On coupe la séquence de bits en morceaux d'égale longueur p et on fait un *ou exclusif* entre ces morceaux
- *Exemple* :
  « OUA » = 01111|10101|00001, p=3
  h(« OUA »)=011xor111xor010xor100xor001
  = 011=$(3)_{10}$

255

255

4. Multiplication

Si q est un nombre réel tel que 0<q<1, on prend
h(x) = ((x*q) mod 1)*m



- *Exemple :*
  q = 0,6125423371, m = 30
  h(« OU ») = ((501* 0,6125423371)mod 1)*30 = 26

- *Inconvénients* :
  - le choix de q ne doit pas être trop proche ni de 0, ni de 1
  - on montre qu'un bon choix est

$$\theta = \frac{\sqrt{5}-1}{2} \quad \text{ou} \quad \frac{\sqrt{5}+1}{2}$$

Nombre d'or!

256

256

Mise en oeuvre des Tables de Hachage

- Comment faire pour passer la fonction de hachage et la fonction fournissant le pas de rehachage à la Table de Hachage?
- On peut avoir des instances de Tables de Hachage qui travailleront avec des fonctions différentes ☺

257

257

Mise en oeuvre des Tables de Hachage

- Plus généralement, comment faire pour passer des traitements (procédures ou fonctions) en paramètres d'autres procédures ou fonctions, ou bien alors en données membres d'une classe?

Utilisation des pointeurs de fonction

258

258

Utilisation des pointeurs de fonction

- De la même manière que les variables, **les fonctions C/C++ possèdent une adresse mémoire**
- Utilisé seul, l'identificateur d'une fonction correspond à la valeur de son adresse
- Possibilité de créer des variables de type pointeur de fonction

Type_val_ret (*pf)(types des paramètres formels)
pf variable destinée à contenir l'adresse d'une fonction ayant une liste de paramètres correspondant aux types indiqués et retournant une valeur de type Type_val_ret

pf(8,16); // ou bien (*pf)(8,16)
Exécution de la fonction pointée par pf sur les paramètres 8 et 16

259

259

•Exemple pointeur de fonction

exemple_ptr_fct.C

```
#include <cstdio>
void affichage_concis(int i)
  {std::printf("%d",i);}

void affichage_bavard(int i)
  {std::printf("Votre entier est %d",i);}

int main()
{
  void (*paff)(int);
  int a=100, c;
  std::printf("Voulez-vous un affichage bavard? (y/n)\n");
  c=std::getchar();
  if(c=='y')
    paff=affichage_bavard;
  else
    paff=affichage_concis;
  paff(a);                    // ou (*paff)(a);
  std::putchar('\n');
  return 0;
}
```

260