L3-INFO Algorithmique

Quick 1 – 7 octobre 2022 – durée 1 h

Les documents et les téléphones (incluant smartphone, tablettes,... tout ce qui contient une interface réseau) interdits. Les calculettes sont autorisées.
Seuls les dictionnaires pour les personnes de langue étrangère sont autorisés.
Le barème est indicatif.

Vous êtes vivement encouragés à illustrer vos recherches et vos réponses par des schémas.

Question 1 : Écriture et preuve d'algorithme (13 points)

On considère un tableau T de n entiers (ces entiers peuvent être négatifs). Le tableau T est indexé de 0 à $n-1$.

L'objectif est de construire un algorithme qui renvoie un indice i tel que $T[i] = i$, ou bien -1 s'il n'en existe pas.

1. (a) *(1 pts)* Écrire un algorithme (simple) qui résoud ce problème.
   (b) *(2 pts)* Calculer son coût dans le meilleur et dans le pire des cas en justifiant bien toute votre réponse.

2. On se place ensuite dans un cas particulier : le tableau T est **trié en ordre croissant** et ne contient que des **entiers distincts**.
   (a) *(2 pt)* Démontrer que si $T[j] > j$, alors pour tout indice $k \geqslant j$ on a $T[k] > k$.
   (b) *(3 pts)* En déduire un algorithme en $\mathcal{O}(\log_2 n)$ pour ce cas particulier où T est trié et sans doublon.
   (c) *(1 pts)* Démontrer que la complexité de l'algorithme que vous avez écrit est effectivement en $\mathcal{O}(\log_2 n)$.

3. On s'intéresse enfin à une variante du problème initial : pour un tableau T de n entiers (quelconques, non ordonnés), déterminer **deux** indices **distincts** i et j tels que $T[i] = j$ et $T[j] = i$, ou bien poser $i = j = -1$ s'il n'en existe pas.
   (a) *(2 pts)* Écrire un algorithme, le plus efficace possible, résolvant ce dernier problème.
   (b) *(1 pt)* Déterminer sa complexité au pire.
   (c) *(1 pt)* Dans le cas où T est trié en ordre croissant, pouvez-vous améliorer la complexité de votre algorithme pour cette variante du problème ?

Réponse 1 :

1. On effectue une Simple recherche séquentielle, donc un coût de n au pire (aucun indice i ne convient, ou seulement $i = n$) et de 1 au mieux ($i = 1$ convient).

```
RechercheSeq(T)
Données : T : un tableau sur [0..n-1] d'entiers
Résultat : un indice i tel que T[i]=i, -1 s'il n'y en a pas
i : entier
i := 0
while (i < n) et (T[i] ≠ i)
    i := i + 1
if i < n
    return i
else
    return -1
```

2. (a) Si $T[j] > j$ alors $T[j] \geqslant j + 1$. Puisque le tableau est ordonné « strictement », $T[j + 1] > T[j] \geqslant j + 1$. On peut poursuivre ce raisonnement pour tous les indices $k > j$.

   Pour mettre ce raisonnement au propre, démontrons par récurrence que $T[k] > k$ pour tout entier $k \geqslant j$. Les cas $k = j$ et $k = j + 1$ ont été traités ci-dessus. Supposons que $T[k] > k$ pour un certain rang k : alors, de même que ci-dessus, $T[k + 1] > T[k] \geqslant k + 1$.

   (b) On peut faire la même remarque dans le cas $T[j] < j$: tous les entiers $k \leqslant j$ vérifient alors $T[k] < k$. Il s'ensuit que si $T[j] \neq j$, il est inutile de chercher dans l'une des moitiés du tableau. D'où l'algorithme suivant (la complexité en $\log_2(n)$ nous guidait vers une recherche dichotomique).

```
RechercheDichotomique(T)
Données : T : un tableau sur [0..n-1] d'entiers
Résultat : un indice i tel que T[i]=i, -1 s'il n'y en a pas
Pré-condition : T est trié par ordre croissant « strict »
g,d,i : entiers
g := 0
d := n-1
i := (g + d)/2
while (T[i] ≠ i) et (g ⩽ d)
    if T[i] > i
        d := i-1
    else
        g := i+1
    i := (g + d)/2
if T[i] = i
    return i
else
    return -1
```

   (c) Voir la complexité de la dichotomie : $d-g$ est divisé par 2 à chaque itération, il y a donc au pire $\log_2(d-g)$ itérations, avec initialement $g = 0$ et $d = n-1$.

3. (a) Si on s'y prend mal, on écrit deux boucles imbriquées pour tester tous les couples (i, j).

   Un petit peu moins mal, on ne teste que pour $j \geqslant i$.

   Mais en fait il suffit d'une recherche séquentielle d'un i tel que $T[T[i]] = i$, après avoir pris la précaution de vérifier si $1 \leqslant T[i] \leqslant n$ bien sûr :

```
RecherchePaire(T)
Données : T : un tableau sur [0..n-1] d'entiers
Résultat : un couple d'indices (i, j) tels que T[i]=j et T[j]=i, ou (-1,-1) s'il n'y en a pas
i : entier
i := 0
while (i < n)
    if (1 ⩽ T[i] ⩽ n) et puis (T[i] ≠ i) et puis (T[T[i]] = i)
        return (i, T[i])
    else
        i := i + 1
return (-1,-1)
```

   (b) Complexité quadratique si on s'y prend mal, linéaire sinon.

   (c) Comme le tableau est croissant, le cas $T[i] = j$ et $T[j] = i$ est impossible (on aurait $T[i] > T[j]$ si $i < j$).

   On peut donc répondre (-1,-1) en temps constant.

Question 2 : Structure de données (7 points)

On souhaite implémenter une structure de données appelée $PileDouble$ vérifiant la spécification suivante :

Opérations

Vide	:	$void \rightarrow PileDouble$
Empiler	:	$Element \times PileDouble \rightarrow PileDouble$
Enfiler	:	$Element \times PileDouble \rightarrow PileDouble$
EstVide	:	$PileDouble \rightarrow bool$
Dépiler	:	$PileDouble \rightarrow PileDouble$
Sommet	:	$PileDouble \rightarrow Element$

Préconditions

Dépiler(p)	:	p est non vide
Sommet(p)	:	p est non vide

Axiomes

EstVide(Vide())	=	vrai	
EstVide(Empiler(e, p))	=	faux	
EstVide(Enfiler(e, p))	=	faux	
Sommet(Enfiler(e, Vide()))	=	e	
Sommet(Enfiler(e, p))	=	Sommet(p)	si p est non vide
Sommet(Empiler(e, p))	=	e	
Dépiler(Enfiler(e, Vide()))	=	Vide()	
Dépiler(Enfiler(e, p))	=	Enfiler(e, Dépiler(p))	si p est non vide
Dépiler(Empiler(e, p))	=	p	

Intuitivement, cette structure se comporte comme une pile, et permet en plus d'insérer un nouvel élément en bas de la pile, à l'aide de l'opération Enfiler.

1. *(1 pt)* Comment se comporte cette structure si on interdit l'opération Empiler ?
2. *(3 pts)* Proposer une implémentation du type $PileDouble$ basée sur un tableau d'éléments (et des compteurs auxiliaires si besoin). Pour cette question, il est possible de proposer une réalisation par effets de bord.

   Remarque : Comme notre tableau est de taille limitée, la $PileDouble$ pourra être « pleine » ; dans cette situation, on affichera un message d'erreur si on cherche à ajouter un nouvel élément.

   Plus vous parviendrez à réaliser d'opérations en temps constant, plus votre solution sera valorisée.

   Un schéma préliminaire vous sera d'un grand secours et sera apprécié par le correcteur.
3. *(3 pts)* Démontrer que votre implémentation respecte les deux axiomes suivants :

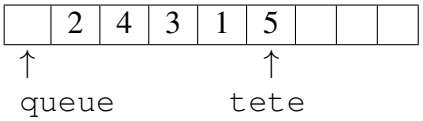
Sommet(Enfiler(e, p))	=	Sommet(p)	si p est non vide
Dépiler(Empiler(e, p))	=	p	

Réponse 2 :

1. Sans l'opération Empiler, l'élément qui est dépilé est toujours le premier arrivé, donc cette structure est indentique à une File (l'opération nommée Dépiler est alors plutôt un "Défiler").
2. Comme dans le cours, on peut utiliser un tableau "circulaire" avec deux indices pour la tête et la queue de la structure.

   Plus précisément, l'indice `tete` désigne l'élément au sommet de la structure, et l'indice `queue` la prochaine place disponible pour Enfiler.

Par convention, la structure est vide si les indices sont identiques ; par conséquent avec un tableau de taille n on ne pourra stocker que $n-1$ éléments (sinon on ne peut pas distinguer la pile vide de la pile pleine).



3. Les opérations s'écrivent alors :

— Vide :
```
tete := 0
queue := 0
```
— Empiler(p, e) :
```
Si (tete + 1) % n == queue :
  // Le modulo est pour le cas où la tete est tout à droite
  raise "La PileDouble est pleine"
tete := tete + 1
Si tete >= n :
  tete := 0  // circulaire : on reprend à partir de la gauche
T[tete] := e
```
— Enfiler(p, e) :
```
Si (tete + 1) % n == queue :
  raise "La PileDouble est pleine"
T[queue] := e
queue := queue - 1
Si queue < 0 :
  queue := n-1  // circulaire : on reprend à partir de la droite
```
— EstVide(p) :
```
Return (tete == queue)
```
— Dépiler(p) :
```
tete := tete - 1
Si tete < 0 :
  tete := n-1
```
— Sommet(p) :
```
Return (T[tete])
```

3.

— Comme on travaille par effets de bord, Sommet(Enfiler(e, p)) revient en fait à exécuter Enfiler(e, p) puis Sommet(p).
La première opération consiste à affecter `T[queue] := e` et à décrémenter `queue`, elle ne modifie pas `tete`.
Si la pile n'est pas vide, on a la garantie que `tete != queue`, donc que la première affectation ne modifie pas `T[tete]`.
Par conséquent, Sommet(p) renverra la même chose, que l'on ait ou pas effectué Enfiler(e, p) avant.

— De même, Dépiler(Empiler(e, p)) revient à effectuer successivement Empiler(e, p) puis Dépiler(p).
La première opération incrémente `tete` puis affecte `T[tete] := e`. La seconde opération décrémente `tete`, cette variable est donc revenue à sa valeur initiale. Il reste à vérifier que la modification du tableau n'a pas d'importance.
Si la pile n'était pas pleine, les cases d'indices compris entre `tete + 1` et `queue` ne contiennent pas d'éléments pertinents pour notre PileDouble ; or l'élément `e` a été affecté à l'une de ces cases, donc sans conséquence pour le contenu de la structure.