université de BORDEAUX

Collège Sciences et technologies
Licence Informatique Maths-Informatique CMI-ISI
Etapes : 4LIN2, 4TLTI2, 4TLCM2
Année universitaire 2016-2017
Session 2 d'automne

Epreuve : Algorithmique des structures de données élementaires.
Code UE : 4TIN302U **Date** : 12 juin 2017 **Heure** : 11h30 **Durée** : 1h30
Aucun document autorisé.

Attention : Dans ce sujet les exercices sont indépendants. Il y a beaucoup de lecture et peu de questions. Prenez le temps nécessaire pour bien lire et comprendre les différents sujets d'exercices.

▸ Exercice 1. *Listes doublement chaînées*

Le type abstrait **liste doublement chaînée** *peut être implémenté de la manière suivante :*

```
struct DCList{
  int data;
  struct DCList* next;
  struct DCList* previous;
};
typedef struct DCList* DCList;
```

On considère par la suite que la primitive `creerListe` *(voir l'annexe pour le détail) vous est fournie et peut être utilisée pour répondre aux questions de l'exercice.*

1. *En utilisant l'implémentation* `DCList`*, écrire la primitive* `insererAvant` *pour les listes doublement chaînées. Pour rappel, la primitive* `insererAvant` *insère un élément* `v` *dans une liste doublement chaînée* `L`*, avant un élément "pointé" par* `p` *:* `fonction insererAvant(L: DCList, p: DCList, v: objet): DCList`*.*
2. *Quelle est la complexité de la primitive* `insererAvant` *que vous avez implémentée ?*
3. *Lire et comprendre le code de la fonction* `enigma`*, pour laquelle* `suivant`*,* `insererApres` *et* `insererEnTete` *sont des primitives des listes doublement chaînées (voir l'annexe pour le détail) :*

```
DCList enigma(DCList L, data v){
  if (L == NULL)
    return insererEnTete(L, v);
  DCList p = L;
  while (suivant(p) != NULL)
    p = suivant(p);
  return insererApres(L, p, v);
}
```

(a) *Que fait la fonction* `enigma` *si elle est appelée pour la liste doublement chaînée* L *composée de la séquence de valeurs suivante :* $L = 1, 2, 3, 4, 5$ *et la valeur* $v = 0$ *?*

(b) *Que fait la fonction* `enigma` *dans le cas général ?*

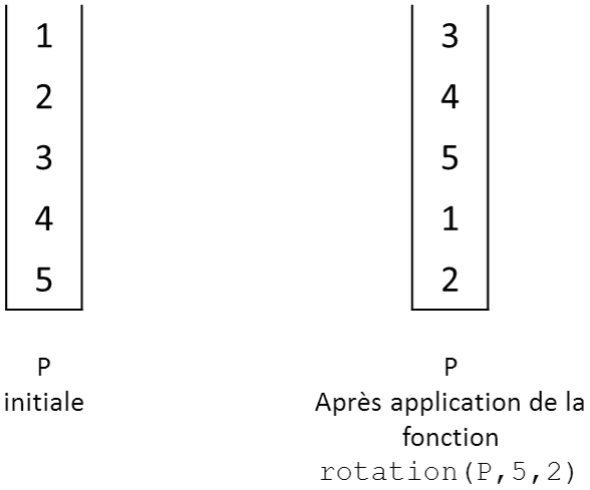
(c) *Quelle est sa complexité ?* ***Justifiez votre réponse.***

▸ **Exercice 2.** ***Piles***

Pour cet exercice les primitives du type **Pile** *vous sont fournies et peuvent être utilisées pour répondre aux questions de l'exercice (voir l'annexe pour le détail).*

1. *Soit* P *une* `Pile` *de* n *entiers. Ecrivez (***en pseudo-code***) un algorithme* `rotation(val P: Pile, val n:int, val k:int)` *qui prend en entrée la* `Pile` P*, le nombre* n *d'éléments de* P*, un entier* $k < n$ *et effectue une rotation de* k *éléments dans* P*.*
   Par exemple si P *contient]1,2,3,4,5] avec* 1 *en tête de pile et* $k = 2$ *alors la pile* P *contiendra* $]3, 4, 5, 1, 2]$ *avec* 3 *en tête de pile après l'application de l'algorithme.*

   Attention : *votre algorithme doit modifier et retourner la pile initiale* `P`*.*



2. *Modifiez votre algorithme pour gérer les cas où* $k \geq n$
3. *Quelle est la complexité de votre algorithme ? (Justifiez votre réponse)*

▸ **Exercice 3.** ***Listes simplement chainées : supprimer les répétitions***

Pour cet exercice les primitives du type **ListeSC** *vous sont fournies et peuvent être utilisées pour répondre aux questions de l'exercice (voir l'annexe pour le détail).*

Ecrivez une fonction `supprimerRepetition(L)` *qui permet de supprimer les répétitions consécutives d'un élément dans une liste* L.
Par exemple si $L = [10, 10, 10, 10, 20, 10, 10, 30, 10, 10, 10]$ *après l'application de la fonction* `supprimerRepetition(L)` *on aura* $L = [10, 20, 10, 30, 10]$

```
fonction supprimerRepetition(val L:SClist):SCList;
```

▸ **Exercice 4.** ***Tables de hachage***

Rappel de cours *: Une* **fonction de hachage** h_m *est une fonction définie sur un ensemble* K *(un ensemble de valeurs appelées* **clés***) dans* $\{0, \ldots, m-1\}$. *Si* k *est une clé,* $h_m(k)$ *est dite* **valeur de hachage** *de* k. **Une table de hachage** *est une structure de données telle que le nombre d'éléments est fixe et l'accès aux éléments s'effectue indirectement par les valeurs de hachage de leurs clés.*

Pour une séquence d'éléments, il est clair qu'il peut exister deux clés différentes k_1 *et* k_2 *de* K *telles que* $h_m(k_1) = h_m(k_2)$, *on dit alors qu'il y a* **collision** *des clés* k_1 *et* k_2. *Une méthode qui permet de gérer les collisions, dite* **par adressage ouvert**, *consiste à parcourir la table pour une clé* k *à partir de l'indice* $h_m(k)$ *afin de trouver une première case libre. Si au bout de* m *sondages, aucune case n'a été détectée comme étant libre, la table est dite* **saturée**.

On considère la séquence de clés $1, 4, 12, 7, 9, 10, 11$ *que l'on veut stocker dans une table de hachage de taille* $m = 6$ *en utilisant la fonction de hachage* $h_6(k) = k$ *modulo* 6. *Les collisions seront traitées par adressage ouvert en partant de la case* $h_m(k)$ *et en se déplaçant de case en case par pas de 1, jusqu'à trouver une case libre.*

1. *Décrire le processus d'ajout de la séquence de clés* $1, 4, 12, 7, 9, 10, 11$ *à partir d'une table vide, en montrant son évolution, ainsi que les étapes de gestion des collisions.*
2. *Décrire les différentes étapes nécessaires lorsqu'on supprime la clé* 7 *de la table.*
3. *Décrire les différentes étapes nécessaires à la recherche de la clé* 13.

▸ **Exercice 5.** ***Caractères stockés dans un arbre binaire***

Compréhension de la structure de données

Un arbre binaire est une structure de données hiérarchisée dans laquelle chaque objet est en relation avec au plus 3 autres objets de la structure quand ils existent :

1. *un père*
2. *un fils gauche*
3. *un fils droit*

*Par exemple, dans l'arbre binaire de caractères A*1 *de la Figure 1, les liaisons vers les fils sont représentées, quand elles existent, par des flèches pleines et la liaison vers le père est représentée par une flèche en pointillé. Dans cet exemple le* `sommet` *de valeur M admet donc :*

- *le sommet de valeur Z pour* `père`
- *le sommet de valeur D pour* `fils gauche`
- *le sommet de valeur L pour* `fils droit`

On remarque également que le sommet de valeur P n'a pas de fils gauche et que le sommet de valeur L n'a pas de fils droit. De plus dans un arbre binaire il existe un sommet unique qui n'a pas de père on dit que c'est la **racine** *de l'arbre. Dans notre exemple la racine est le sommet de valeur Z.*

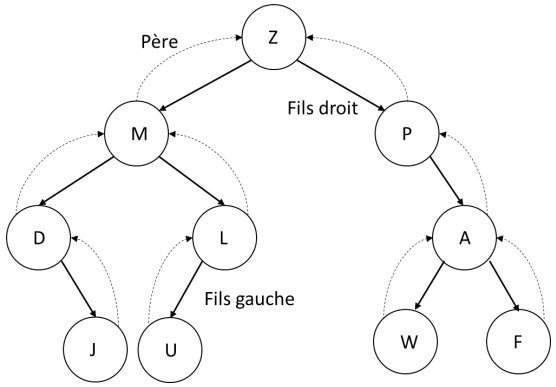


FIGURE 1 – A1 : Arbre binaire de caractères

Pour stocker un arbre binaire on peut utiliser un tableau $\mathbf{T}$*[1, ..., NMAX] (de taillle NMAX) qui sera organisé de la façon suivante :*

- *La valeur de la racine est stockée dans la case* 1.
- *Si un sommet de valeur s est stocké dans la case d'indice i du tableau :*
  - *la valeur de son fils gauche sera stockée dans la case d'indice* $2 * i$,
  - *la valeur de son fils droit sera stockée dans la case d'indice* $(2 * i) + 1$,
  - *la valeur de son père sera stockée dans la case d'indice* $i//2$ *(division entière),*

Dans notre exemple quand un fils n'existe pas on stocke la valeur null dans la case correspondante.

On obtient ainsi le tableau T ci-dessous :

$\boldsymbol{T}$:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	...	*NMAX*
Z	*M*	*P*	*D*	*L*	*null*	*A*	*null*	*J*	*U*	*null*	*null*	*null*	*W*	*F*	*null*	...	*null*

1. *Construisez le tableau qui permettra de stocker l'arbre binaire A2 dans la figure 2.*

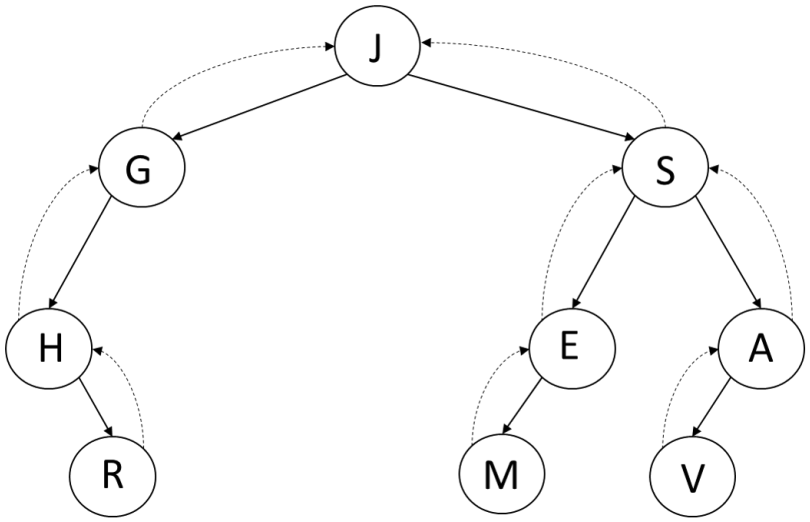


FIGURE 2 – A2 : Arbre binaire de caractères à stocker dans un tableau

Implémentation : lecture de code

Lorsqu'on implémente un arbre binaire de caractères avec un tableau, un sommet s est un indice du tableau. Un sommet est donc un ***entier****.*
L'arbre est défini par le sommet racine c'est à dire l'entier 1*.*

Le type abstrait `Arbre Binaire` *est défini par un ensemble de primitives d'accès et de modification. Dans cet exercice nous n'utiliserons que les primitives d'accès qui sont les suivantes :*

```
typedef int sommet;
typedef sommet arbreBinaire_de_car;

fonction valeurSommet (val s:sommet):car;
/* retourne la valeur du sommet s et vaut NULL si le sommet n'existe pas */

fonction filsGauche(val s:sommet):sommet;
/* retourne le fils gauche de s et vaut -1 si s n'a pas de fils gauche */

fonction filsDroit(val s:sommet):sommet;
/* retourne le fils droit de s et vaut -1 si s n'a pas de fils droit */

fonction pere(val s:sommet):sommet;
/* retourne le pere de s et vaut -1 si s est la racine de l'arbre */
```

On considère la fonction mystere suivante :

```
fonction mystere2(val A : arbreBinaire de car): vide ;
  var s : sommet
  P : pile de sommet
  {
  creerPile(P);
  empiler(P, 1);
  tantque !pilevide(P){
    s=valeurpile(P)
    afficher(valeurSommet(s))
    depiler(P)
    si filsDroit(s)!= -1 alors
      empiler(P,filsDroit(s));
    si filsGauche(s)!= -1 alors
      empiler(P,filsGauche(s));
  }
}
```

2. *En détaillant l'évolution de la pile utilisée dites quel sera le résultat de l'appel* `mystere(A1)` *où* $A1$ *est l'arbre binaire de caractères donné en exemple dans la figure 1 ?*
3. *Que fait la fonction* `mystere` *dans le cas général ?*
4. *Quelle est sa complexité ? (Justifiez votre réponse)*

1 Annexe : Primitives des différents Types Abstraits vus en cours

Liste simplement chainée

```
fonction creerListe(): SCList;
fonction detruireListe(L : SCList) : vide;
fonction listeVide(L : SCList) : booleen;
fonction valeurListeSC(L : SCList) : objet;
fonction suivant(L : SCList): SCList;
fonction insererEnTete(L : SCList, v: objet): SCList;
fonction supprimerEnTete(L : SCList): SCList;
fonction insererApres(L : SCList, p :SCList, v: objet):SCList;
fonction supprimerApres(L : SCList, p :SCList): SCList;
```

Liste doublement chainée

```
fonction creerListe(): DCList;
fonction detruireListe(D: DCList) : vide;
fonction listeVide(D: DCList) : booleen;
fonction valeurListeDC(D: DCList) : objet;
fonction suivant(D: DCList): DCList;
fonction precedent(D: DCList): DCList;
fonction insererEnTete(D: DCList, v: objet): DCList;
fonction supprimerEnTete(D: DCList): DCList;
fonction insererApres(D: DCList, p: DCList, v: objet): DCList;
fonction supprimerApres(D: DCList, p: DCList): DCList;
fonction insererAvant(D: DCList, p: DCList, v: objet): DCList;
fonction supprimerAvant(D: DCList, p: DCList): DCList;
```

Pile

```
fonction creerPile(): pile de objet;
fonction detruirePile(P: pile de objet): vide;
fonction pileVide(P: pile de objet): booleen;
fonction valeurPile(P: pile de objet): objet;
fonction depiler(P: pile de objet): pile de objet;
fonction empiler(P: pile de objet, val: objet): pile de objet;
```

File

```
fonction creerFile(): file de objet;
fonction detruireFile(F: file de objet): vide;
fonction fileVide(F: file de objet): booleen;
fonction valeurFile(F: file de objet): objet;
fonction defiler(F: file de objet): file de objet;
fonction enfiler(F: file de objet, val: objet): file de objet;
```