Algorithmique et structures de données 2
- Contrôle de connaissances -

8 avril 2023

VERSION 1

Nom Prénom : _____

Numéro d'étudiant.e : _____

Consignes

- durée : 1H ;
- lisez tout le sujet et commencez par ce qui vous semble le plus facile, les questions peuvent être faites dans l'ordre que vous voulez ;
- un coup d'oeil sur la copie du voisin ou de la voisine donne lieu à un avertissement. Le deuxième avertissement donne lieu à une exclusion de la salle et un 0/20 ;

Rappels :

- `sizeof(char) = 1`
- Python utilise le passage de paramètre *par variable*. Si une variable d'un type mutable est passée en paramètre d'une fonction et y est modifiée, la modification perdure après la fin de l'exécution de la fonction.

1. Quels sont les risques liés aux fuites mémoires ? (1 pt)

   La mémoire peut être saturée, cela peut déclencher un ralentissement du système

[Questions 2–6] (V/F) Parmi les affirmations suivantes lesquelles sont vraies ? Justifiez votre réponse avec une courte phrase, un exemple ou un contre-exemple. (2,5 pts)

2. Un tableau est un objet de taille fixe en C.

   Vrai, la taille est fixée à la déclaration

   int tab[10];

   tab est un tableau de 10 entiers.

3. Un pointeur n’est pas toujours une variable

Vrai, on peut par exemple afficher l'adresse d'une variable var (le pointeur vers cette variable) en utilisant l'opérateur & et sans déclarer de variable additionnelle.
Ex : printf("%ld", &var);

4. La taille d’un pointeur dépend du type de la variable vers laquelle il pointe

Faux, la taille d'un pointeur est toujours la même sur un système donné :
8 octet sur un système 64 bits, 4 octets sur un système 32 bits, car un pointeur est une adresse.

5. Si p est de type `int*` alors *p est de type int.

Vrai, l'opérateur * permet de déréférencer le pointeur p.
p pointe sur un entier, donc *p est un entier.

6. Si p est de type `int*` alors &p est de type `int**`.

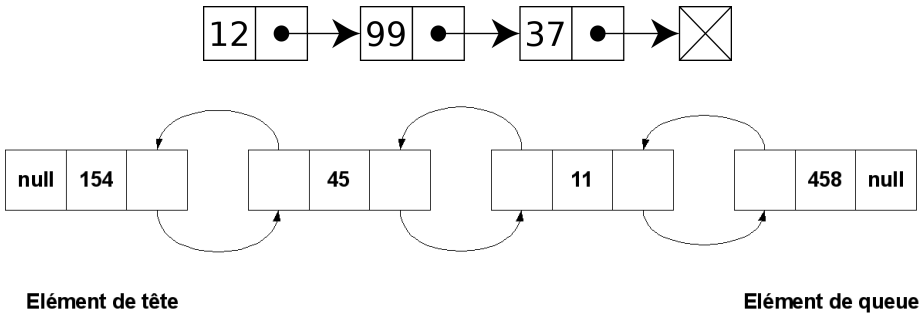
Vrai, l'opérateur & permet d'accéder à l'adresse d'une variable. Si p est un pointeur sur un entier, son adresse est un "pointeur de pointeur" sur un entier.

FIN du V/F

7. Proposez l'implémentation de **structures** en C pour
   - — une liste simplement chaînée contenant des valeurs entières (exemples ci-dessous).
   - — une liste doublement chaînée chaînée contenant des valeurs entières (exemple ci-dessous)

   Nommez correctement votre structure pour qu'on puisse déclarer un pointeur L1 vers une liste simplement chaînée et un pointeur L2 vers une liste doublement chaînée comme ceci (3 pts) :

```
liste_s* L1;
liste_d* L2;
```




```
typedef struct liste_simplement_chainee {
        int val ;
        struct liste_simplement_chainee* suiv;
} liste_s;



typedef struct doublement_chainee {
        struct doublement_chainee* prec;
        int valeur ;
        struct doublement_chainee* suiv ;
} liste_d;

liste_s* L1;
liste_d* L2;
```

8. Pourquoi les indices des tableaux sont ils des entiers ? Quel est le lien avec la mémoire ? Vous pouvez prendre un exemple pour illustrer. (1,5 pt)

lorsqu'on déclare un tableau par int tab[5], "tab" correspond à l'adresse du premier élément du tableau (tab[0]. Un int en C est stocké sur 4 octets (32 bits). Lorsqu'on appelle tab[1], on va chercher la valeur à l'adresse tab+1*sizeof(int), c'est à dire tab+4. La troisième valeur du tableau est tab[2], elle est stockée à l'adresse tab+2*sizeof(int), c'est à dire tab+8.

9. Quelle est la différence entre le passage de paramètre *par valeur* et le passage de paramètre *par variable*? Pourquoi utilise-t-on le passage par référence en C? Écrivez une courte fonction en C permettant d'illustrer votre propos. (1 pt.)

le langage C n'autoriste pas le passage de paramètre par variable qui permet de modifier la valeur d'une variable dans une fonction au delà du "scope" de la fonction.
Pour modifier durablement la valeur d'une variable, on utilise donc le passage par référence, en donnant en paramètre l'adresse de la variable à modifier.

```
void swap(int *x, int *y)
{
	int temp = *x;
	*x = *y;
	*y = *temp;
}

int main(){
	int x = 3;
i	int y = 4;
	swap(&x, &y); // après l'appel, x vaut 4 et y vaut 3)
	return °;
}
```

10. Sachant le code suivant :

```
 5 int main(){
 6     int* p=NULL;
 7     int tab[4];
 8     printf("%ld, %ld, %ld\n",sizeof(tab), sizeof(&tab), sizeof(tab[0]));
 9     p = malloc(5*sizeof(int));
10     printf("%ld, %ld\n",sizeof(p), sizeof(*p));
11     p = malloc(4*sizeof(int));
12     printf("%ld, %ld\n",sizeof(p), sizeof(*p));
13     for (int i=0; i<5; i++){
14         if (i%2 == 0) { // i est pair
15             p[i]=i;
16         } else {
17             p[i] = 0;
18         }
19     }
20     for (int i=0; i<5; i++){
21         printf("%d",p[i]);
22     }
23     free(p);
24     return 0;
25 }
```

(a) Qu'affiche l'exécution de ce programme ? (1 pt.)

16, 8, 4
8, 4
8, 4
00204 // attention ici aux bornes ! i va de 0 à 4 donc cela fait 5 valeurs

(b) Ce programme génère-t-il une fuite mémoire ? Si non, pourquoi, si oui de combien d'octet(s) et comment la corriger ? (1 pt.)

Vrai, le programme génère une fuite de 20 octets (seule la mémoire allouée au deuxième malloc est libérée). Il faut faire un free après la ligne 9 pour libérer l'espace alloué ligne 9, ou utiliser realloc.

(c) Que se passerait-il si on supprimait les lignes 9, 10, 11 et 12, qu'on recompilait puis qu'on exécutait le programme ? Pourquoi ? (1 pt.)

On génèrerait une erreur de segmentation en essayant d'accéder aux zones mémoire p[i] car p est déclaré comme un pointeur qui vaudrait null.

(d) Quel est le test correspondant à une précaution d'usage qui manque dans ce code pour s'assurer de ne pas faire un accès mémoire illégal à l'exécution ? Pourquoi ce test est-il nécessaire ? Quelles sont les lignes à englober dans le bloc `if` correspondant ? (1 pt.)

Il faut tester si p est NULL, car le malloc pourrait avoir échoué, renvoyant un pointeur null. Le bloc doit englober les lignes 13 à 22.

11. Citez 4 propriétés d'une variable. (1 pt)

nom, valeur, adresse, type

12. Comment s'appelle l'opération qui consiste à utiliser l'opérateur `*` sur une variable de type pointeur ? (0,25 pts)

le déréférencement

13. Expliquez la raison d'être des tables de hachage ainsi que leur fonctionnement. Vous pouvez vous appuyer sur un exemple (autre que celui vu en TP). Qu'appelle-t-on une collision ? (2,5 pts)

Les tables de hachage permettent de stocker des couples (clé, valeur) lorsque la clé n'est pas un entier.
Grâce à une fonction de hachage, la clé (par exemple une chaîne de caractères) est transformée en une clé entière comprise entre deux bornes.
Cette clé entière est utilisée comme indice pour stocker la valeur dans un tableau.

Exemple : on souhaite stoker l'âge d'étudiant.e.s dans un tableau et retrouver celui-ci à partir de leur nom.

Une collision intervient lorsque la fonction de hachage renvoie le même entier pour deux chaînes de caractères différentes. Dans ce cas, on doit stocker deux valeurs pour une même clé (indice). Une manière de faire cela est de stocker les valeurs dans des listes chaînées stockées à l'indice correspondant.

[Questions 14–17] Le stockage des données est il contigu dans le cas (marquez vrai ou faux à côté des réponses possibles) (2 pts) :

14. d'un tableau
15. d'une liste chaînée
16. d'un ABR
17. d'un un type composé (`struct`)

FIN du V/F