UGA - UFR IM^{2}AG **Master 2 CCI**

Programmation et Langages 1

Examen du 24 novembre 2017 (session 1)

Duré : 2h – **Documents :** autorisés – Barême indicatif (sur 20 points) – 3 exercices indépendants

Exercice 1 (∼ 3 points)

Donnez les valeurs des expressions `a` et `*p` aux points d'observation (1), (2), (3), (4) et (5). Indiquez sur lesquels de ces points d'observation une **erreur d'exécution** va avoir lieu en précisant pourquoi.

```
int a, b ;
int t[10] ;
int *p, *q ;

int f1(int x) {
  x = x + 1;
    /* (2) valeurs de a et *p ? */
}

int f2(int *x, int y) {
  *x = y + 1 ;
    /* (4) valeurs de a et *p ? */
  return y ;
}

int main() {
  a = 9 ;
  q = &a ;
  p = q ;
    /* (1) valeurs de a et *p ? */
  f1(a);
    /* (3) valeurs de a et *p ? */
  b = f2 (p, a) ;
    /* (5) valeurs de a et *p ? */
  t[a] = *p ;
  return 0 ;
}
```

Exercice 2 (3 points).

Une liste **doublement chainée** est une liste dans laquelle chaque cellule est reliée par un pointeur à la cellule suivante et par un autre pointeur à la cellule précédente.

Q1. Dessinez une liste **doublement chainée** d'entiers comportant les 3 entiers `5`, `8` et `2`.

Q2. Donnez une implémentation en C du **typ**e liste doublement chaînée d'entiers

Exercice 3 (∼ 14 points)

On souhaite implémenter une librairie permettant à un utilisateur d'allouer des blocs contigus de cellules mémoire dans un (grand) tableau noté `Mem` [1]. A tout moment `Mem` contiendra donc des blocs **libres** (qui ne sont pas alloués) ou **occupé** (qui sont alloués). Une fonction `blocSuiv` permet de parcourir l'ensemble de ces blocs. On introduit donc tout d'abord le lexique suivant :

```
ADMAX : la constante entiere 1024
adresse : le type entier sur -1..ADMAX
Mem : un tableau sur 0..ADMAX-1 d'adresse

Init : action
{ e. i. : indifferent
  e.f. : Mem contient un seul bloc libre de taille ADMAX-1}

estLibre : fonction (ad : adresse) -> booleen
{estLibre(ad) est vrai si ad est l'adresse d'un bloc libre}

tailleBloc : fonction (ad : adresse) -> entier
{tailleBloc(ad) est la taille (en nombre de cellules) du bloc d'adresse ad (libre ou occupe)}

premBloc : fonction -> adresse
{premBloc renvoie l'adresse du premier bloc (ce bloc peut etre libre ou occupe)}

blocSuiv : fonction (ad : adresse) -> adresse
{blocSuiv(ad) renvoie l'adresse du bloc suivant le bloc d'adresse ad (libre ou occupe),
    et renvoie -1 si ce bloc n'a pas de bloc suivant}

nbLibre : fonction -> entier
{nbLibre renvoie le nombre de blocs libres dans Mem}
```

Q1. Donnez le contenu d'un fichier `alloc.h` implémentant ce lexique.

Q2. En utilisant les primitives `Init`, `estLibre`, `premBloc` et `blocSuiv` écrivez le corps de la fonction `nbLibre`.

On suppose maintenant que les blocs sont représentés de la manière suivante dans le tableau `Mem` :

- pour un bloc d'adresse `a`, `Mem[a-1]` contient la taille de ce bloc. Cette taille est positive si ce bloc est libre, elle est négative dans le cas contraire (une taille de bloc ne peut jamais être nulle).
- le premier bloc est toujours à l'adresse 1.
- les blocs sont contigus : pour un bloc d'adresse `a` et de taille `t`, le bloc suivant débute à l'adresse `a+t+1`, et la taille de ce bloc suivant est donc dans `Mem[a+t]` (c.f Figure 1).

La Figure 1 représente le tableau `Mem` contenant 3 blocs : le premier bloc d'adresse 1 est occupé, sa taille est 3, le bloc suivant d'adresse 5 est libre, sa taille est 2, le bloc suivant d'adresse 8 est occupé, sa taille est 1016.

Q3. Ecrire le corps de l'action `Init`.

Q4. Ecrire le corps des fonctions `estLibre`, `tailleBloc`, `premBloc` et `blocSuiv`.

1. l'objectif est ici de comprendre un fonctionnement possible des primitives `malloc` et `free` du langage C

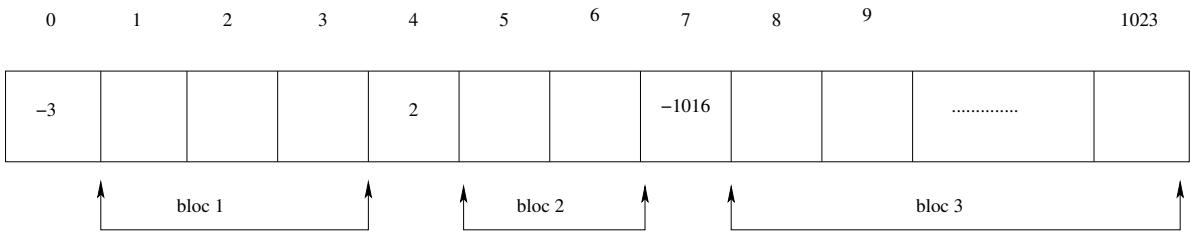

FIGURE 1 – Tableau `Mem` contenant 3 blocs

Partie 3

On complète maintenant le lexique précédent en ajoutant les trois actions suivantes :

```
Allouer : action (donnee t : entier, resultat ad : adresse)
{ e.i. : indifferent
  e.f. : ad est l'adresse d'un bloc libre de taille t, -1 si il n'existe pas de tel bloc}

 Liberer : action (donnee ad : adresse)
{ e.i. : ad est l'adresse d'un bloc occupe
  e.f. : ce bloc a ete libere
  }

Reallouer : action (donnee-resultat ad : adresse, t : entier)
{ e.i. : ad est l'adresse d'un bloc occupe de taille <= t
  e.f. : si un nouveau bloc d'adresse ad ete alloue, de taille t, alors
       le contenu de l'ancien bloc d'adresse ad y a ete copie et cet ancien bloc est libere ;
       si aucun nouveau bloc n'a pu etre re-allouee ad vaut -1, et l'ancien bloc n'est pas libere}
  }
```

Q5. Ecrivez un exemple de programme principal qui permet de construire l'état du tableau `Mem` représenté sur la Figure 1.

Q6. Ecrivez le corps de l'action `Reallouer` en utilisant les fonction `Allouer` et `Liberer` (et sans oublier le cas échéant de copier le contenu du bloc initial dans le bloc ré-alloué)

Q7. Ecrivez le corps de l'action `Liberer`.

Q8. Ecrivez le corps de l'action `Allouer`. On pourra se baser sur l'algorithme suivant :

1. rechercher dans le tableau `Mem` un premier bloc libre de taille $p \geq t$;
2. cas 1 : si un tel bloc existe, alors marquer ce bloc comme "occupé" en lui donnant la taille t (et en affectant son adresse à ad) et marquer comme "libre" le morceau de bloc restant de taille $p - t$ (si $p - t > 0$) ;
   cas 2 : sinon renvoyer dans ad la valeur -1.

Q9. Lorsque l'on libère un bloc, il est avantageux de voir si celui-ci peut être **fusionné avec le bloc suivant**, si celui-ci est libre également, pour obtenir un bloc libre de plus grande taille. Ré-écrivez l'action `Liberer` en prenant en compte cette contrainte.

Q10. Est-il également possible de fusionner un bloc libéré avec le bloc précédent (si celui-ci est libre) ? Indiquez les solutions possibles éventuelles (en quelques phrases, sans écrire le code C).