Programmation C
TP n° 11 : Multi-ensembles

La notion de *multi-ensemble* (multiset) généralise celle d'ensemble au sens suivant : un multi-ensemble est un collection d'objets, mais contrairement à un ensemble, il peut contenir plusieurs exemplaires d'un même objet. Par exemple, le multi-ensemble $\langle 0, 2, 0, 2, 1, 1, 3, 1, 1\rangle$ contient 2 fois la valeur 0, 2 fois la valeur 2, une fois la valeur 3 et 4 fois la valeur 1.
Le but de ce TP est d'écrire une implémentation des multi-ensembles d'entiers en C.

Représentation ds multi-ensembles

Le contenu des multi-ensembles doit pouvoir croître et décroître de façon arbitraire pendant l'exécution du programme. Nous utiliserons donc pour les représenter un type de structure analogue a celui utilisé pour représenter les listes simplement chaînées vues en cours :

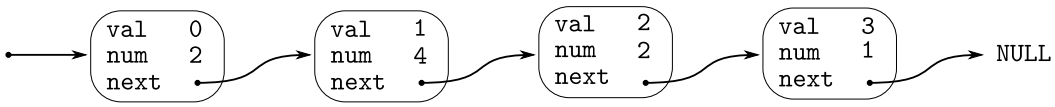
```c
typedef struct node {
  int val;
  unsigned num;
  struct node *next;
} node;

typedef node* mset;
```

Un multi-ensemble `mset` sera représenté par un pointeur vers son premier nœud, ou un pointeur nul s'il est vide. Le champ `val` d'un noeud représente une valeur, le champ `num` représentant le nombre de fois où cette valeur apparaît dans le multi-ensemble.
Un multi-ensemble sera dit *optimal* si son chaînage vérifie les deux conditions suivantes :

- la valeur du champ `num` de chaque nœud est non nulle et,
- si un nœud n'est pas le dernier du chaînage, la valeur de son champ `val` est strictement inférieur à la valeur du champ `val` de son successeur.

La première condition assure que tous les noeuds d'un multi-ensemble sont indispensables. La seconde impose que la suite des champs `val` des éléments du chaînage forme une suite strictement croissante. Par exemple, la représentation optimale du multi-ensemble $\langle 0, 2, 0, 2, 1, 1, 3, 1, 1\rangle$ aura en mémoire la forme suivante :



Ces deux conditions permettront d'améliorer les performances des primitives de gestion des multi-ensembles. Elle devront bien sûr être toutes les deux preservées par les fonctions modifiant le contenu d'un multi-ensemble optimal, et exploitées par les autres à chaque fois que cela permet d' améliorer le temps de traitement.

Remarques sur les tests

Les exercices ci-dessous sont de difficulté croissante, mais tant qu'une primitive d'ajout n'aura pas été écrite, vous ne pourrez pas tester les fonctions des exercices 1 et 2 sans construire explicitement au moins un multi-ensemble. Vous pouvez vous inspirer par exemple du fragment de code suivant, construisant un multi-ensemble optimal (n'oubliez pas de varier les valeurs numériques).

```
1  int vals[4] = {0, 1, 2, 3};
2  int nums[4] = {2, 4, 2, 1};
3  int i;
4  mset m = NULL;
5  for (i = 3; i >= 0; i--){
6    node *nn = malloc(sizeof(node));
7    assert(nn != NULL);
8    nn -> val = vals[i];
9    nn -> num = nums[i];
10   nn -> next = m;
11   m = nn;
12 }
```

Exercice 1 : Affichage

Ecrire un fonction `void print_mset(mset m, short verbose)` affichant le contenu d'un multi-ensemble de la manière suivante :

- si la valeur de `verbose` est nulle, la fonction affichera la suite (croissante) de toutes les valeurs aparaissant au moins une fois dans `m` en précisant, pour chaque valeur et entre parenthèses, son nombre d'occurrences – soit, avec l'exemple ci-dessus :

  ```
  0 (2) 1 (4) 2 (2) 3 (1)
  ```

- sinon, elle affichera ces valeurs en affichant autant de fois chaque valeur que son nombre d'occurrences :

  ```
  0 0 1 1 1 1 2 2 3
  ```

Exercice 2 : Propriétés

Ecrire *par récurrence* (sans boucles) les fonctions suivantes :

1. `int length_mset(mset m)` renvoyant la longueur de `m`, c'est à dire son nombre de nœuds (4 dans l'exemple ci-dessus).
2. `int size_mset(mset m)` renvoyant le nombre total d'éléments de `m` (9 dans l'exemple ci-dessus).
3. `int num_mset(mset m, int val)` renvoyant le nombre d'occurrences de `val` dans `m`. La fonction supposera `m` optimal, et devra exploiter cette propriété.
4. `int is_optimal(mset m)` renvoyant `1` si `m` est optimal, et `0` sinon (pensez à la tester sur des représentations de multi-ensembles ne vérifiant *pas* cette condition).

A partir de ce point de l'énonce, nous supposerons *tous* les multi-ensembles considérés comme optimaux. Chaque fonction modifiant le contenu d'un multi-ensemble devra présever cette propriété (la fonction `is_optimal` pourra être utilisée dans vos tests) ;

Exercice 3 : Primitive d'ajout

Ecrire les fonctions suivantes :

1. `mset new_node(int val, unsigned num)` construisant et renvoyant un nouveau multi-ensemble contenant `num` fois l'unique valeur `num` – en s'assurant par un `assert` que le multi-ensemble construit est optimal.
2. `mset add_val(int val, unsigned num, mset m)` renvoyant le multi-ensemble obtenu en ajoutant `num` exemplaires de la valeur `val` à `m`. Cette fonction doit être récursive (pas de boucles) et ne pourra appeler que la précédente, `assert` et elle-même.

Noter que pour `add_val`, il y a deux cas à gérer : ou bien `val` apparaît dans `m`, ou bien il s'agit d'une nouvelle valeur. Comme indiqué ci-dessus, `m` est supposé optimal et le multi-ensemble résultant doit lêtre aussi.

Exercice 4 : Primitive de suppression

Ecrire la fonction suivante :

1. `mset remove_val(int val, unsigned num, mset m, unsigned *num_rem)` renvoyant :
   - si `val` apparaît au moins `num + 1` fois dans `m`, le multiset obtenu en retirant `num` exemplaires de `val` de `m`,
   - sinon, le multiset obtenu en retirant *tous* les exemplaires de `val` de `m`,

   La fonction devra en outre écrire à l'adresse `num_rem` le nombre d'éléments effectivement retirés de `m`. Elle doit être écrite par récurrence, et ne peut appeler que `free` et elle-même.

Exercice 5 : Construction et extraction

Ecrire les fonctions suivantes :

1. `mset build(int *values, size_t size)`. Cette fonction suppose que `values` est l'adresse d'un vecteur contenantt `size` entiers, dans un ordre quelconque. Elle doit construire un multi-ensemble (optimal) contenant tous les éléments de ce vecteur (servez-vous de `add_val`).
2. Ajoutez à votre programme la déclaration de type de structure suivante :

```
typedef struct pair {
  int val;
  unsigned num;
} pair;
```

   Ecrire une fonction `void *extract(mset m, size_t *size, short pairs)` :
   - si `pairs` est non nul, la fonction doit allouer un vecteur d'éléments de type `pair` de même longueur (au sens de la fonction `length_mset` de l'exercice 2) que `m` et recopier dans chaque élément de ce vecteur les valeurs des champs `val` et `num` du nœud de même rang dans `m`,
   - sinon la fonction doit allouer un vecteur d'entiers de même taille (au sens de la fonction `size_mset`) que `m` et recopier dans ce vecteur chaque élément de `m`, dans l'ordre croissant.

   Dans chaque cas, la fonction devra renvoyer l'adresse du vecteur alloué, et écrire à l'adresse `size` le nombre d'éléments ce vecteur.