Programmation C
TP n° 12 : Multi-ensembles et pointeurs de fonction

Préambule : pour ce TP, on respectera les principes de programmation modulaire : un fichier .c pour le code source, un header .h associé et un dernier fichier .c avec juste un main pour tester les fonctions.

La notion de *multi-ensemble* (multiset) généralise celle d'ensemble au sens suivant : un multi-ensemble est une 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 des 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 :

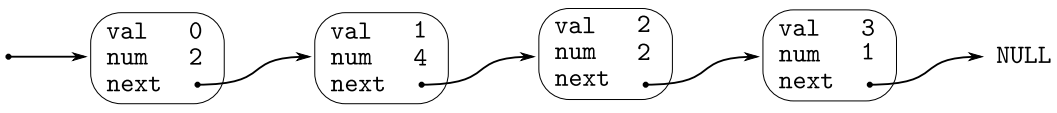
```c
//à déclarer dans le fichier .c contenant les fonctions d'accès à un mset
struct node {
  int val;
  unsigned num;
  struct node *next;
};

//à déclarer dans le fichier .h
typedef struct node 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 la multiplicité de la valeur, c'est-à-dire 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 de donner une représentation unique à tout multi-ensemble et 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.
Á partir de ce point de l'énoncé, nous supposerons *tous* les multi-ensembles considérés comme optimaux. Chaque fonction modifiant le contenu d'un multi-ensemble devra présever cette propriété.

Exercice 1 : construction

Écrire les fonctions suivantes :

1. `mset new_node(int val, unsigned num)` construisant et renvoyant un nouveau multi-ensemble contenant `num` fois l'unique valeur `val` – 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`. Attention, il y a deux cas à gérer : ou bien `val` apparaît dans `m`, ou bien il s'agit d'une nouvelle valeur.
3. `mset build(int *values, size_t size)`. Cette fonction suppose que `values` est l'adresse d'un vecteur contenant `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`).

Exercice 2 : affichage

Écrire 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
  ```

Vous pouvez maintenant vérifier que la création de multi-ensembles fonctionne bien.

Exercice 3 : suppression

Écrire la fonction `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`. N'oubliez pas de libérer la mémoire.

Soit les multi-ensembles E, F. On considère que la multiplicité d'un élément x dans un multi-ensemble E est 0 si x n'est pas présent dans E. Il existe alors 4 opérations usuelles sur les multi-ensembles :

- *l'addition* $E + F$: la multiplicité d'un élément x est la somme des multiplicités de x dans les multi-ensembles E et F. Par exemple,
  ```
  premier multi-ensemble : 0 0 0 1 1 2 2 3 3 3 3 4 4 5
  deuxième multi-ensemble : 1 1 1 3 3 6 6 6
  resultat : 0 0 0 1 1 1 1 1 2 2 3 3 3 3 3 3 4 4 5 6 6 6
  ```
- *la soustraction* $E - F$: la multiplicité d'un élément x est la différence entre la multiplicité de x dans E et celle dans F. Si la multiplicité de x dans F est plus élevée, alors la multiplicité de x dans $E - F$ est 0. Par exemple,
  ```
  premier multi-ensemble : 0 0 0 1 1 2 2 3 3 3 3 4 4 5
  deuxième multi-ensemble : 1 1 1 3 3 6 6 6
  resultat : 0 0 0 2 2 3 3 4 4 5
  ```
- *l'intersection* $E \bigcap F$: la multiplicité d'un élément x est le minimum des multiplicités de x dans E et F. Par exemple,
  ```
  premier multi-ensemble : 0 0 0 1 1 2 2 3 3 3 3 4 4 5
  deuxième multi-ensemble : 1 1 3 3 6 6 6
  resultat : 1 1 3 3
  ```
- *l'union* $E \bigcup F$: la multiplicité d'un élément x est le maximum des multiplicités de x dans E et F. Par exemple,
  ```
  premier multi-ensemble : 0 0 0 1 1 2 2 3 3 3 3 4 4 5
  deuxième multi-ensemble : 1 1 1 3 3 6 6 6
  resultat : 0 0 0 1 1 1 2 2 3 3 3 3 4 4 5 6 6 6
  ```

Exercice 4 : opérations

Écrire les fonctions suivantes :

1. `mset mplus(mset m, mset n)` qui retourne un multi-ensemble résultat de l'adddition des deux multi-ensembles `m` et `n`,
2. `mset mmoins(mset m, mset n)` qui retourne un multi-ensemble résultat de la soustraction du multi-ensemble `n` au multi-ensemble `m`,
3. `mset mintersection(mset m, mset n)` qui retourne un multi-ensemble résultat de l'intersection des deux multi-ensembles `m` et `n`,
4. `mset munion(mset m, mset n)` qui retourne un multi-ensemble résultat de l'union des deux multi-ensembles `m` et `n`.

Exercice 5 : pointeur de fonction

Écrire la fonction `operation` qui étant donné deux multi-ensembles et une opération dont la signature est identique à celles des fonctions écrites à l'exercice précédent, applique l'opération aux deux multi-ensembles et affiche les deux multi-ensembles ainsi que le multi-ensemble résultat. Il faudra pour cela utiliser un pointeur de fonction. Testez votre fonction avec les différents opérateurs écrits.