Liste doublement chaînée d'entiers

1 Buts

- Créer et utiliser une librairie de liste doublement chaînée d'entiers.

2 Énoncé

- Créer une librairie de gestion de liste doublement chaînée d'entiers avec les fonctionnalités associées.

2.1 La structure de liste doublement chaînée

Le type de liste doublement chaînée doit être la structure suivantes:

```
typedef struct element {
    int data;
    struct element *next;
    struct element *prev;
} element_t;
typedef struct {
    element_t *head;
    element_t *pos;
} dll;
```

Le pointeur `head` pointe toujours sur le premier élément de la liste chaînée. Le pointeur `pos` pointe lui sur un élément arbitraire de la liste. C'est notre "curseur".

2.2 Fonctions de création et consultation

- Initialisation de la liste

  ```
  void dll_init(dll *list);
  ```

  où le sommet et la position actuelle sont initialisés à `NULL`.

- Consultation à `pos`

  ```
  bool dll_value(dll list, int *data);
  ```

  assigne à `data` la valeur à la position actuelle dans la liste. Si la liste est vide on retourne `false`.

- La liste est-elle vide

```
bool dll_is_empty(dll list);
```

- Est-ce que `pos` est le 1er élément?

  ```
  bool dll_is_head(dll list);
  ```

  Si la liste est vide, cette fonction retourne `false`.

- Est-ce que `pos` est le dernier élément?

  ```
  bool dll_is_tail(dll list);
  ```

  Si la liste est vide, cette fonction retourne `false`.

- Est-ce que la valeur `data` est dans la liste?

  ```
  bool dll_is_present(dll list, int data);
  ```

- Transformation de la liste en chaîne de caractères en partant de la tête

  ```
  char *dll_to_str(dll list);
  ```

  Si la liste est vide, cette fonction retourne `""`. Si la liste n'est pas vide elle retourne la liste de nombre en commençant par la tête de liste et tous les nombres sont séparés par des espaces.

2.3 Fonctions de manipulation

- Déplacement de `pos` au début de la liste

  ```
  void dll_move_to_head(dll *list);
  ```

- Déplacement de `pos` à la position suivante dans la liste

  ```
  void dll_next(dll *list);
  ```

  Si `pos` est déjà le dernier élément on ne fait rien.

- Déplacement de `pos` à la position précédente dans la liste

  ```
  void dll_prev(dll *list);
  ```

  Si `pos` est déjà le premier élément on ne fait rien.

2.4 Fonctions d'insertion

- Insertion de `data` dans l'élément après `pos`

  ```
  bool dll_insert_after(dll *list, int data);
  ```

  Si on a une erreur on retourne `false`. On met `pos` sur l'élément nouvellement inséré.

- Insertion de `data` en tête de liste

  ```
  bool dll_push(dll *list, int data);
  ```

  Si on a une erreur on retourne `false`. On met `pos` sur l'élément nouvellement inséré.

2.5 Fonctions d'extraction et destructions

- Extraction de la valeur se trouvant dans l'élément `pos`, libération de l'élément pointé par `pos`, et déplacement de `pos` à son élément suivant

  ```
  bool dll_extract(dll *list, int *data);
  ```

  Si la liste est vide, on retourne `false` et on ne modifie pas `data`. Si l'élément extrait était le dernier de la liste, `pos` est déplacé à la nouvelle fin de la liste.

- Extraction de la donnée en tête de liste

  ```
  bool dll_pop(dll *list, int *data);
  ```

  `pos` est inchangé s'il n'est pas en tête de liste. Si la liste est vide, on retourne `false` et on ne modifie pas `data`.

- Destruction de la liste

  ```
  void dll_clear(dll *list);
  ```

  Si la liste est déjà vide, la fonction ne fait rien.

2.6 Tests

N'oubliez pas de tester **chacune** de vos fonctions au fur et à mesure que vous les écrivez. Créez ainsi un prorgamme qui petit à petit appelle chacun des fonctions que vous écrivez et testez le bon fonctionnement de celles-ci (en affichant, par exemple, l'état de la liste à chaque instant). Il est inutile d'écrire tout le code en une fois et de tenter ensuite de le tester. Écrivez également un `Makefile` permettant la compilation de votre projet.