TD XXIV

LISTES DOUBLEMENT CHAÎNÉES

Une *liste doublement chaînée* est une liste dans laquelle chaque maillon contient un lien vers le maillon suivant et un lien vers le maillon précédent. C'est une structure fondamentalement impérative, et assez pratique, en particulier parce qu'elle permet de supprimer un élément arbitraire en temps constant.

1 Première version

On peut utiliser la structure suivante :

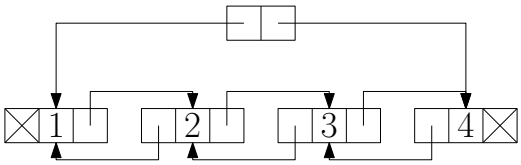

FIGURE XXIV.1 – Liste doublement chaînée $(1, 2, 3, 4)$, première version.

Cette structure correspond aux définitions ci-dessous :

```c
struct Node {
    int value;
    struct Node *prev;
    struct Node *next;
};

typedef struct Node node;

struct Dll_1 {
    node *start;
    node *end;
};

typedef struct Dll_1 dll_1;
```

On a comme invariants (si `u` est de type `dll_1*`)

- la liste est vide si et seulement si `u->start == NULL` si et seulement si `u->end == NULL` ;
- si la liste n'est pas vide, alors `u->start->prev == NULL`, `u->end->next == NULL`, et ce sont les deux seuls pointeurs nuls présents dans la chaîne.

On donne les fonctions permettant de créer un nouveau nœud (avec une valeur donnée et des pointeurs nuls des deux côtés) et une nouvelle liste (vide) :

```c
node *new_node(int x){
    node *n = malloc(sizeof(node));
    n->value = x;
    n->next = NULL;
    n->prev = NULL;
    return n;
}
```

```c
dll_1 *new_dll_1(void){
    dll_1 *d = malloc(sizeof(dll_1));
    d->start = NULL;
    d->end = NULL;
    return d;
}
```

Exercice XXIV.1 p. 4

1. Écrire une fonction `delete_node` qui supprime un nœud d'une liste. Cette fonction prend un pointeur vers le nœud et un pointeur vers la liste en arguments, et elle gère à la fois la suppression du nœud et la libération de la mémoire correspondante.
2. Écrire une fonction `insert_before` qui insère un nœud (avec la valeur fournie) juste avant le nœud passé en argument.
3. Écrire la fonction symétrique `insert_after`.
4. Ces fonctions sont-elles suffisantes pour écrire une fonction `from_array` (par exemple) ? Si non, pourquoi ?

```c
void delete_node(dll_1 *d, node *n);
void insert_before(dll_1 *d, node *n, int x);
void insert_after(dll_1 *d, node *n, int x);
```

2 Version avec sentinelle

La structure proposée n'est pas satisfaisante (même si on pourrait la faire marcher) : on peut en fait simplifier le code et enlever presque tous les cas particuliers en la modifiant légèrement. On ne change rien au type `node`, mais on convient de rajouter un nœud « fictif », appelé *sentinelle*, à l'extrémité de la liste. La valeur présente dans le champ `value` de ce nœud ne sera pas significative, et la liste aura la structure suivante :

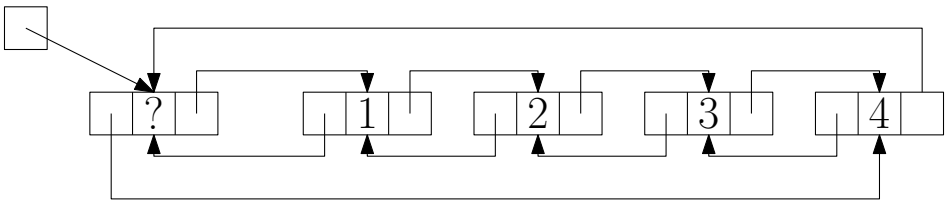

Figure XXIV.2 – Liste doublement chaînée $(1, 2, 3, 4)$, version avec sentinelle.

```c
struct Dll {
    struct Node *sentinel;
};

typedef struct Dll dll;
```

Exercice XXIV.2 p. 5

1. Écrire la fonction `new_dll` qui crée une nouvelle liste doublement chaînée. Cette liste sera vide, ce qui signifie qu'elle ne contiendra que le nœud sentinelle, correctement initialisé.
2. Ré-écrire les fonctions `delete_node`, `insert_before` et `insert_after` pour la nouvelle structure. La fonction `delete_node` pourra supposer sans le vérifier que le nœud passé en argument n'est pas le nœud sentinelle (et aucune de ces fonctions n'aura besoin de prendre la liste elle-même en argument).

```c
dll *new_dll(void);
void delete_node(node *n);
node *insert_before(node *n, int x);
node *insert_after(node *n, int x);
```

3. Écrire une fonction `free_dll` qui libère la totalité de la mémoire utilisée par une liste doublement chaînée.

```c
void free_dll(dll *d);
```

4. Une liste doublement chaînée permet de réaliser facilement la structure abstraite de *deque* (**d***ouble* **e***nded* **que***ue*, ou *file bilatère*).
Écrire les quatre fonctions suivantes, dont la spécification devrait être facile à deviner. Pour `pop_left` et `pop_right`, on vérifiera la licéité de l'appel à l'aide d'un `assert`.

```c
void push_left(dll *d, int x);
void push_right(dll *d, int x);

int pop_left(dll *d);
int pop_right(dll *d);
```

5. Écrire une fonction `from_array` qui convertit un tableau en liste doublement chaînée.

```c
dll *from_array(int t[], int len);
```

3 Nombres chanceux

On considère le processus suivant :

- on part de la liste des entiers impairs (jusqu'à une certaine borne n) ;
- l'entier 1 est *chanceux* ;
- on considère l'entier qui suit 1 dans la liste (c'est 3) ;
- on élimine un nombre sur 3 de la liste, en commençant au début ;
- l'entier 3 est chanceux ;
- on considère l'entier qui suit 3 dans la liste (c'est 7) ;
- on élimine un nombre sur 7 de la liste, en commençant au début ;
- l'entier 7 est chanceux...

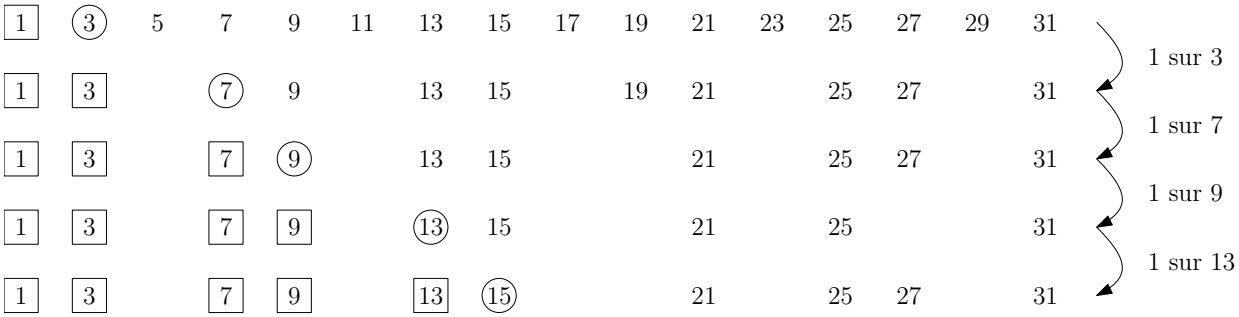

Figure XXIV.3 – Génération des nombres chanceux.

Exercice XXIV.3

Combien vaut la somme des nombres chanceux inférieurs ou égaux à 10^5.
Remarque
L'idée est d'utiliser une liste doublement chaînée : je ne dis pas que c'est le plus simple ou le plus efficace, c'est juste un exercice...

Solutions

Correction de l'exercice XXIV.1 page 2

```c
void delete_node_1(dll_1 *d, node *n){
    if (n->prev != NULL){
        n->prev->next = n->next;
    } else {
        d->start = n->next;
    }
    if (n->next != NULL){
        n->next->prev = n->prev;
    } else {
        d->end = n->prev;
    }
    free(n);
}

void insert_before_1(dll_1 *d, node *n, int x){
    node *new = new_node(x);
    new->prev = n->prev;
    new->next = n;
    n->prev = new;
    if (new->prev != NULL){
        new->prev->next = new;
    } else {
        d->start = new;
    }
}

void insert_after_1(dll_1 *d, node *n, int x){
    node *new = new_node(x);
    new->next = n->next;
    new->prev = n;
    n->next = new;
    if (new->next != NULL){
        new->next->prev = new;
    } else {
        d->end = new;
    }
}
```

Telles que nous les avons écrites, ces fonctions ne permettent pas de créer une liste non vide à partir d'une liste vide. En effet, `insert_before_1` et `insert_after_1` n'acceptent pas que le nœud `n` passé en argument soit `NULL`.

Correction de l'exercice XXIV.2 page 2

1. Le nœud sentinelle pointe vers lui-même :

```
dll *new_dll(void){
    dll *d = malloc(sizeof(dll));
    node *sentinel = new_node(0);
    sentinel->prev = sentinel;
    sentinel->next = sentinel;
    d->sentinel = sentinel;
    return d;
}
```

2. C'est nettement plus simple, il n'y a plus aucun cas particulier :

```
void delete_node(node *c){
    assert(c != NULL);
    c->prev->next = c->next;
    c->next->prev = c->prev;
    free(c);
}

node *insert_before(node *c, int x){
    node *n = new_node(x);
    n->next = c;
    n->prev = c->prev;
    c->prev = n;
    n->prev->next = n;
    return n;
}

node *insert_after(node *c, int x){
    node *n = new_node(x);
    n->prev = c;
    n->next = c->next;
    c->next = n;
    n->next->prev = n;
    return n;
}
```

3. Il sera pratique pour la suite de disposer de deux fonctions renvoyant respectivement le premier et le dernier nœùd de la liste.

```c
node *start(dll *d){
    return d->sentinel->next;
}

node *end(dll *d){
    return d->sentinel->prev;
}

void free_dll(dll *d){
    node *n = start(d);
    while (n != d->sentinel){
        node *tmp = n->next;
        free(n);
        n = tmp;
    }
    free(d->sentinel);
    free(d);
}
```

4. C'est immédiat avec ce que l'on a déjà écrit :

```c
void push_left(dll *d, int x){
    insert_after(d->sentinel, x);
}

void push_right(dll *d, int x){
    insert_before(d->sentinel, x);
}

int pop_left(dll *d){
    node *first = start(d);
    assert(first != d->sentinel);
    int value = first->value;
    delete_node(first);
    return value;
}

int pop_right(dll *d){
    node *last = end(d);
    assert(last != d->sentinel);
    int value = last->value;
    delete_node(last);
    return value;
}
```

5. À nouveau, on dispose de tout le nécessaire :

```c
dll *from_array(int t[], int len){
    dll *d = new_dll();
    for (int i = 0; i < len; i++){
        push_right(d, t[i]);
    }
    return d;
}
```