Structures de données
TD3 Liste doublement chaînée

1 Exercice 1 Liste chaînée

- Une liste chaînée permet de parcourir ses éléments de sa tête à sa queue grâce à un pointeur sur l'élément suivant (successeur).
- **Une chaîne doublement chaînée permet de naviguer dans les deux sens, tête → queue et queue → tête grâce à un pointeur sur l'élément suivant et un pointeur sur l'élément précédent. (comme de ce TD)**.
- Une liste est dite circulaire si le successeur du dernier élément pointe sur le premier élément de la liste. Si c'est une liste doublement chaînée, le prédécesseur du premier élément pointe sur le dernier élément.

Les listes sont très utilisées en informatique et des questions sur leur programmation et/ou fonctionnement sont souvent posées durant un entretien d'embauche. C'est pourquoi il peut être utile de connaitre par coeur son fonctionnement (au moins l'ajout et la suppression).

Dans cet exercice une liste est définie par deux structures :

- **List** composée d'un pointeur sur le premier élément et un pointeur sur le dernier élément
- **Node** représentant un noeud de la liste, composée d'une valeur entière, d'un pointeur sur l'élément suivant et un pointeur sur l'élément précedent.

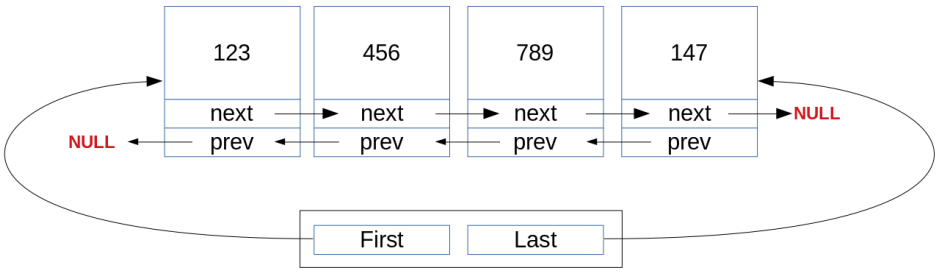

FIGURE 1 – Représentation d'une liste à deux structures

1.1 Création et initialisation d'une liste

```
// Initialisation d'une liste (même si allouee statiquement)
inline void list_init(List* list)
{
    list->first = NULL;
    list->last  = NULL;
}

// Allocation dynamique et initialisation d'une liste
List* list_create()
{
    List* list = (List*) malloc(sizeof(List));
    if(list != NULL) {
        list_init(list);
    }
    return list;
}
```

1.2 Ajouter un élément

```
void list_add(List* list, int value) // ajout en tete
{
    Node* n = list_create_node(value);
    if(n != NULL) {
        n->next = list->first;
        // Si c'est le premier noeud alors c'est aussi le dernier
        if(list_is_empty(list)) {
            list->last = n;
        } else { // Sinon on met a jour la tete actuelle
            list->first->prev = n;
        }
        // nouvelle tete de liste
        list->first = n;
    }
}
```

La fonction list_add permet d'ajouter un nouvel élément en tête de liste. On commence par créer un nouvel élément (un noeud, un maillon, un node... peu importe le nom). Puis, si la liste est vide, alors ce nouvel élément est le premier mais aussi le dernier. Sinon, il faut mettre à jour la tête actuelle pour que ce nouvel élément devienne son prédécesseur. Enfin, on met à jour la structure liste pour que ce nouvel élément soit la tête.

Pour ajouter en queue (c'est-à-dire à la fin de liste), le principe est le même : nouvel élément, vérifier si la liste est vide, mettre à jour le succésseur de la queue actuelle, déclarer le nouvel élément comme queue de liste.

On ne l'a pas vu dans ce TD mais on pourrait également ajouter avant/après un élément.

Attention : les listes reposent sur la manipulation de pointeurs, assurez vous qu'ils soient valides et non nuls pour ne pas avoir d'erreur de segmentation. Par exemple, quand vous faîtes *list->first->prev = n ;* vous devez être sûrs que *first* n'est pas null.

1.3 Recherche, affichage...

Certaines opérations telles que la recherche dans une liste ou bien son affichage vont nécessiter de la parcourir. Pour cela, on utilise une variable servant de curseur et une boucle while qui tourne tant qu'on a pas atteint la fin de la liste, c'est à dire un successeur null.

```
Node* list_search(List* list, int value)
{
    Node* n = list->first;
    while(n!=NULL && n->value != value) {
        n = n->next;
    }
    return NULL;
```

Les erreurs de segmentation arrivent souvent dans ces fonctions utilisant des boucles, prenez garde aux pointeurs nuls !

1.4 Suppression d'un élément

```
int list_remove_node(List* list, Node* n)
{
    // Variables de stockage temporaire
    int value = n->value;
    Node* prev = n->prev;
    Node* next = n->next;

    // Si j'ai un predecesseur, je le met a jour
    if(prev != NULL) prev->next = next;
    // Si j'ai un successeur, je le met a jour
    if(next != NULL) next->prev = prev;

    // Si n etait le premier alors son successeur est le nouveau premier
    // (ou null si pas de successeur)
    if(list->first == n) list->first = next;
    // Pareil si j'etais dernier
    if(list->last  == n) list->last  = prev;

    free(n);
    return value;
}
```

Pour supprimer en tête ou supprimer en queue, il suffit d'appeler cette fonction avec le noeud *list->first* ou bien *list->last*.

1.5 Désallouer une liste

```
void list_free(List* list)
{
    Node* cur = list->first;
    Node* n;
    while(cur != NULL) {
        n = cur;
        cur = n->next;
        free(n);
    }
    free(list);
}
```

Le *free(list)* à la fin peut poser problème :

— Si list est allouée statiquement (donc dans la pile) vous ne pouvez pas la libérer de vous même, vous obtiendrez une erreur.

— Si list est allouée dynamiquement (donc dans le tas) et que vous ne faites pas ce free alors vous aurez une fuite mémoire.

Solution : n'utilisez que des listes allouées dynamiquement ?

2 Exercice 2

Une file correspond au principe FIFO (first in first out). Pour construire une file, on peut utiliser une liste dont les ajouts se font uniquement en tête et les suppressions uniquement en queue, ainsi la première valeur entrée est toujours la première valeur sortie.

Dans cet exercice, on représente un bureau de poste par une structure composée d'un nombre entier nbGuichet indiquant le nombre de guichets ainsi que d'un tableau guichets contenant nbGuichets listes, et qui représente les guichets de la poste.

2.1 Créer les guichets

La fonction creerGuichets crée une structure Poste, affecte le champ nbGuichet, alloue le tableau tab puis initialise chaque guichet (= chaque liste).

```
Poste* creerGuichets(int nb)
{
    Poste* poste = (Poste*)malloc(sizeof(Poste));
    if(poste != NULL) {
        poste->guichets = (List*)malloc(nb * sizeof(List));
        if(poste->guichets == NULL) {
            free(poste);
            return NULL;
        }
        poste->nb = nb;
        for(int i=0;i<nb;i++) list_init(&poste->guichets[i]);
    }
    return poste;
}
```

2.2 Afficher les guichets

```
void afficherBureauPoste(Poste* poste)
{
    printf("Nombre de files : %d\n", poste->nb);
    for(int i=0;i<poste->nb;i++) {
        if(list_is_empty(&poste->guichets[i])) {
            printf("- File du guichet %d : ", i);
            list_show(&poste->guichets[i]);
        } else {
            printf("- File du guichet %d : vide\n", i);
        }
```

```
    }
}
```

2.3 Ajouter une personne à un guichet

```
void ajouterAuGuichet(Poste* poste, int nbGuichet, int noPersonne)
{
    list_add(&poste->guichets[nbGuichet], noPersonne);
}
```

2.4 Appeler la personne suivante à un guichet

```
int appelerAuGuichet(Poste* poste, int noGuichet)
{
    if(list_is_empty(&poste->guichets[noGuichet])) return -1;
    else return list_remove_tail(&poste->guichets[noGuichet]);
}
```

3 Exercice 3

Dans cet exercice nous avons besoin d'une pile. Il est possible de construire une pile à partir d'une liste :

- **fonction Empiler :** correspond à l'ajout en tête dans une liste
- **fonction Dépiler :** correspond à la suppression en tête dans une liste

Vous pouvez réécrire des fonctions empiler(...) et depiler() pour que votre code soit plus propre (et vous pourrez réutiliser ses fonctions un jour si besoin).

Comme nous travaillons avec des caractères et une liste qui stocke des entiers, nous utilisons le code ASCII de ceux-ci quand il s'agit d'un opérateur. Les opérateurs ont un code qui précède le code du chiffre 0, voilà pourquoi nous faisons des +'0' ou -'0'.

```
int evalExpr(char* expr)
{
    List* pile = list_create();
    char c;
    int n1, n2, res;
    char op;
    for(int i=0; expr[i] != '\0'; i++) {
        c = expr[i];
        // Si le caractere est un operateur ou un chiffre
        if(c == '+' || c == '-' || c == '*' || c == '/'
```

```
            || (c >= '0' && c <= '9')) {
            // On empile = on ajoute en tete
            list_add(pile, (int)c);

        }
        // Si le caractere est une parenthese fermante on depile et on calcule
        else if(c == ')') {
            // on depile le second operande = on supprime en queue
            n2 = list_remove_tail(pile) - (int)'0';
            // on depile l'operateur
            op = list_remove_tail(pile);
            // on depile le premier operande
            n1 = list_remove_tail(pile) - (int)'0';

            switch(op) { // realisation de l'operation
                case '+':
                    res = n1 + n2;
                    break;
                case '-':
                    res = n1 - n2;
                    break;
                case '*':
                    res = n1 * n2;
                    break;
                case '/':
                    res = n1 / n2;
                    break;
            }

            // on empile le resultat
            list_add(pile, res + (int)'0');
        }
    }
    res = list_remove_head(pile) - (int)'0';
    free(pile);
    return res;
}
```