RAPHAËL GOUL - Rapport Algo3 - Séquence 3

1. Clarification

> **Dans ce document :**
>
> - Je répond aux question du TP.
> - Je décrit les algorithmes mis en œuvre.
>   - Je les analyse en complexité en temps et en espace (chose que j'avais oublier pour le TP2).
>   - J'explique mes démarche de programmation.
>   - J'explique les difficulté rencontrées et leurs solution trouvée.
> - Je référence les documentations et aides utilisée.

J'ajoute que si j'ai mis ce chapitre/point c'est principalement pour que les chiffres des différentes partie correspondent avec celle du sujet du TP.

---

2. Implantation des constructeurs et de l'opérateur map

a. List* list_create(void)

Comme indiquer, pour cette fonction j'ai repris l'idée du `stack_create()` de la Séquence 2 :

```
Stack* create_stack(int max_size) {
    Stack* s;
    size_t capacity = (max_size > 0 ? max_size : STACK_SIZE);
    s = malloc(sizeof(struct s_stack) + sizeof(void *) * capacity);
    s->stack = (const void**)(s+1);
    s->capacity = capacity;
    s->top=-1;
    return (s);
}
```

J'ai défini `List* l` et modifier le `malloc` en prenant donc la taille de la structure `s_List` + celle d'un `LinkedElement` pour que la liste et la sentinel soit proche dans la mémoire. Puis j'ai

modifier la suite pour convenir aux structure données. C'est a dire ont cast la valeur de la sentinel avec un `LinkedElement*`, ont construit la boucle et ont défini la taille a 0 :

```
l->sentinel = (LinkedElement*)(l+1);
l->sentinel->previous = l->sentinel;
l->sentinel->next = l->sentinel;
l->size = 0;
```

b. void list_delete(ptrList* l)

Ici j'ai opter pour un curseur qui parcours toute la liste. C'est donc `while (actuel!=(*l)->sentinel)`, temps que ont est pas de retour a la sentinel, ont mémorise le pointeur vers la suite, ont `free` l'élément actuel, et on déplace le curseur. Quand c'est fini, ont libère notre curseur et ont le met a `NULL`.

> J'avais eu un petit problème ici, en effet par étourderie dans le `while` j'avais mis `actuel!=(*l)->sentinel->next)` mais j'ai très vite régler le problème.

Complexité N vue que c'est un parcours de liste.

c. List *list_push_back(List* l, int v)

J'ai repris le code donner en cours, je l'ai juste adapter pour que cela corresponde à la structure du TP. Dans l'idée ont va allouer un nouvel élément, définir sa valeur, faire pointer sont `next` vers la sentinel, son `previous` sur `l->sentinel->previous`. Et ont va mettre a jour les pointeurs de la list, c'est a dire le dernier élement avant le notre va pointer son `next` vers le nouvel élément, et la sentinel `previous` pointera vers ce nouvel élément aussi.

```
List* list_push_back(List* l, int v) {
    LinkedElement* e = malloc(sizeof(LinkedElement));
    e->value = v;
    e->next = l->sentinel;
    e->previous = e->next->previous;
    e->previous->next = e;
    e->next->previous = e;
    l->size++;
    return l;
}
```

Complexité 1, ce n'est pas un parcours ont récupère directement le pointeur previous de la sentinel

d. List *list_map(List* l, Functor f, void* environment)

Là aussi j'ai repris le code du cours. C'est un for qui va parcourir toute la liste et appliqué a chaque valeur la fonction `ListFunctor f` souhaiter, cela va donc changer les valeurs de la list.

```
List* list_map(List* l, ListFunctor f, void* environment) {
    for (LinkedElement* e = l->sentinel->next; e!= l->sentinel; e = e->next)
    e->value=f(e->value,environment);
    return l;
}
```

Complexité N, parcours de liste.

e. bool list_is_empty(const List* l)

Ont retourne juste `true` si taille == 0, `false` sinon :

```
bool list_is_empty(const List* l) {
    return l->size==0;
}
```

Complexité 1, c'est juste une vérification de donnée (taille == 0)

f. int list_size(const List* l)

Exactement la même chose que `list_is_empty`, la seule différence c'est qu'on retourne la taille, un `int`.

```
int list_size(const List* l) {
    return l->size;
}
```

Complexité 1 (comme demandé), ont renvoie la taille

---

3. Implantation de l'opérateurs push_front.

List *list_push_front(List* l, int v)

Une fois qu'on a coder `list_push_back`, c'est très simple ! Ont **inverse** les `next` et les `previous`.
Autrement dit, au lieu d'ajouter un élément a la fin, ont l'ajoute au début, il suffit juste de modifier les pointeur en question.

```
List* list_push_front(List* l, int v) {
    LinkedElement* e = malloc(sizeof(LinkedElement));
    e->value = v;
    e->previous = l->sentinel;
    e->next = e->previous->next;
    e->next->previous = e;
    e->previous->next = e;
    l->size++;
    return l;
}
```

Complexité 1, ici aussi ce n'est pas un parcours ont récupère directement le pointeur next de la sentinel

---

4. Implantation des opérateurs d'accès et de suppression en tête et en fin de liste.

> Dans toutes les fonctions du 4. j'ai ajouter une assert en début de fonction qui vérifie que la liste ne soit pas vide. C'était pas demandé mais je me suis dit que c'était plutôt intéressant de faire cela.

a. int list_front(const List* l)

Extrêmement simple, ont retourne la valeur du premier élément de la liste. (Bon en vrai j'ai eu un petit problème j'avais mis `l->sentinel->value`, sauf que je me suis rendu compte que la sentinel ne contenait aucune valeur, c'était une information que j'avais zapé.)

```
int list_front(const List* l) {
    assert(!list_is_empty(l));
    return l->sentinel->next->value;
}
```

Complexité 1 ont renvoie la valeur du premier élément

b. int list_back (const List* l)

La aussi, c'est juste que au lieu de `next` ont va prendre le pointeur `previous`

```
int list_back(const List* l) {
    assert(!list_is_empty(l));
```

```
    return l->sentinel->previous->value;
}
```

Complexité 1 ont renvoie la valeur du dernier élément

c. List *list_pop_front(List* l)

J'ai suivi l'algorithme donné dans le cours pour cette fonction. Ont récupère et sauvegarde temporairement le premier élément, ont modifie le pointeur de la sentinel pour le mettre vers l'élément suivant, ensuite ont fait pointer l'élément suivant (qui est désormais le premier) vers la sentinel. Ont oublie pas de mettre a jour `size` vue qu'on a diminué la taille de la liste, ont free l'élément sauvegarder et ont retourne la liste.

```
List* list_pop_front(List* l) {
    assert(!list_is_empty(l));
    LinkedElement* e = l->sentinel->next;
    l->sentinel->next = e->next;
    l->sentinel->next->previous = l->sentinel;
    l->size--;
    free(e);
    return l;
}
```

Complexité 1 ont free le premier élément de la liste

d. List *list_pop_front(List* l)

C'est exactement la même chose, cependant ici ont va pop le back, donc le dernier élément. Ont inverse les `next` et les `previous`.

```
List* list_pop_back(List* l){
    assert(!list_is_empty(l));
    LinkedElement* e = l->sentinel->previous;
    l->sentinel->previous = e->previous;
    l->sentinel->previous->next = l->sentinel;
    l->size--;
    free(e);
    return l;
}
```

Complexité 1 ont free le dernier élément de la liste

---

5. Implantation des opérateurs d'accès, d'insertion et de suppression à une position donnée dans la liste

a. List *list_insert_at(List* l, int p, int v)

J'ai encore suivi l'algorithme du cours. Ont met un curseur qui commence au premier élément, dans un `while` ont va donc avancer `p` fois le curseur.

```
LinkedElement* curseur = l->sentinel->next;
while (p!=0) {
    p--;
    curseur=curseur->next;
}
```

Ont créer un nouvel élément, qu'on alloue. Ont lui met une valeur, et ont commence a mettre les pointeur en place. Le suivant de l'élément sera l'élément désigné par le curseur, et le précédent sera l'ancien `previous` du curseur. Ensuite ont modifie les pointeur `next` de l'élément précédent et le pointeur `previous` de l'élément suivant pour qu'ils pointent vers nouvel élément qu'on vient d'insérer. Ont oublie pas d'augmenter la taille de la liste et ont renvoie la liste.

```
LinkedElement* e = malloc(sizeof(LinkedElement));
e->value = v;
e->next = curseur;
e->previous = curseur->previous;
e->previous->next = e;
e->next->previous = e;
l->size++;
return l;
```

Complexité P, ont doit déplacer le curseur P fois dans la liste

b. List *list_remove_at(List* l, int p)

Algorithme du cours. C'est le même principe que `list_insert_at`, sauf que ici au lieu de créer un élément ont va supprimé l'élément sélectionner par le curseur. Ont met a jour les pointeurs pour que le `next` du précédent pointe vers le nouveau suivant et que le `previous` du nouveau suivant pointe vers le nouveau précédent. Ont free le curseur et ont oublie pas de réduire la taille de la liste.

```
List* list_remove_at(List* l, int p) {
    LinkedElement* curseur = l->sentinel->next;
```

```
    while (p!=0) {
        p--;
        curseur=curseur->next;
    }
    curseur->previous->next = curseur->next;
    curseur->next->previous = curseur->previous;
    free(curseur);
    l->size--;
    return l;
}
```

Complexité P, ont doit déplacer le curseur P fois dans la liste

c. int list_at(const List* l, int p)

Ont suit le même principe que les deux précédent. Ont avance p fois dans la liste, cette fois ont retourne la valeur du curseur.

```
int list_at(const List* l, int p) {
    LinkedElement* curseur = l->sentinel->next;
    while (p!=0) {
        p--;
        curseur=curseur->next;
    }
    return curseur->value;
}
```

Complexité P, ont doit déplacer le curseur P fois dans la liste

> Je n'ai eu aucun problème pour les première partie, tout les output désiré était présent.
> Par contre, c'est vraiment dans cette dernière partie que les problèmes ont commencé.

---

6. Algorithme de tri fusion sur liste doublement chaînée

> Pour répondre a la question "*Dans le fichier list.h ou dans le fichier list.c ? Justifier votre choix.*"
> Il a deux raison de pourquoi il faut mettre la structure dans le `list.c` et ne pas déclarer les fonction dans le `list.h` :
> - La première est très simple, au début du document il est spécifié
> "*Le fichier list.h fourni propose une interface pour l'implantation de ce TAD.* ***Ce fichier ne***

> ***devra pas être modifié lors de ce TP***."
> - La deuxième plus sérieuse, c'est une structure interne utiliser lors du tri, en aucun cas elle ne va être utilise par une fonction externe qui pourrait utiliser la librairie `list.h`.
> C'est comme pour les fonctions qui ne sont pas défini dans le `list.h`, ce sont des fonctions annexe permettant de remplir des tâches précise, elle forme ensemble la fonction `list_sort` et ne vont pas être utilisé hors de cette fonction.
>
> A noter que `list_sort` elle est défini dans `list.h` car elle est la fonction finale utilisé pour faire le tri fusion.

---

a. Définition de la structure SubList

Voici la structure utilisé, comme demander un pointeur tête et un pointeur queue.

```
typedef struct s_SubList {
    struct s_LinkedElement* tete;
    struct s_LinkedElement* queue;
} SubList;
```

> D'ailleurs, au départ j'avais appeler `tete` avec un `ê` cela m'a valu pas mal de problème lors de tentative de compilation.

b. SubList list_split(SubList l)

Au départ j'était partie sur : ont compte les élément de la liste, ont divise par deux, ont réavance jusqu'à l'endroit au milieu et ont fait le split. Le problème : C'est absolument pas optimisé, heureusement je m'en suit rendu compte, j'ai donc rapidement fait quelque recherche internet pour voir si y'avais des moyen plus rapide. J'ai donc fait la découverte de l'algorithme du Lièvre et de la tortue qui permet principalement de détecter des cycle, cependant je peut l'utiliser pour faire un sorte que le lièvre arrive a la fin, et lorsque le lièvre sera a la fin, la tortue ne sera que a la moitié de la `Sublist`.

J'ai donc tenter d'appliquer cela, mais ça n'a pas marcher. La raison principale étant que n'avais absolument pas compris comment était réellement former la `SubList`. En effet dans le sujet du TP il y a une zone que j'avais complètement sauté.

"*Bien que l'on puisse appliquer l'algorithme de tri fusion sur les listes doublement chaînées circulaires, ce qui est le cas de notre liste, la sentinelle jouant le rôle de maillon de rebouclage, le fait que la sentinelle ne définisse pas un élément de la liste peut être gênant pour l'implantation directe de l'algorithme tel que décrit sur la page Wikipédia ci-dessus.*"

J'était parti du principe que la `SubList` boucler, sauf que lors de ma relecture du sujet, je me suis rendu compte que j'avais tout mélanger. J'était arrivé a la fin, j'ai exécuter le programme et sa ma mis un "Segmentation Fault". J'ai tout repassez de fond en comble pour finalement me rendre compte que je m'était trompé depuis le début.

Donc, repartons sur des bonnes base, j'ai refait le `split` et je suis partie sur un principe simple :
Je prend un curseur au début, je prend un curseur a la fin. J'avance mon premier curseur, je fait reculer l'autre, `while (curdebut != curfin && curdebut->next != curfin)`.

```
LinkedElement* curdebut = l.tete;
LinkedElement* curfin = l.queue;
while (curdebut != curfin && curdebut->next != curfin) {
    curdebut = curdebut->next;
    curfin = curfin->previous;
}
```

Une fois placé au milieu, ont créer la structure, ont place les pointeur, et ont oublie pas de couper le lien des curseur entre les deux listes. Ont retourne la structure.

```
output.tete=curdebut; // DERNIER ELEMENT Liste GAUCHE
output.queue=curdebut->next; // PREMIER ELEMENT Liste DROITE
output.tete->next=NULL; // COUPURE
output.queue->previous=NULL;
return output;
```

Complexité N/2, ont parcours que la moitié de la liste réelement

c. SubList list_merge(SubList leftlist, SubList rightlist, OrderFunctor f)

Après être reparti sur mon split, j'ai donc refait mon merge, étonnamment je n'avais pas trop de chose a modifié. Ont met un curseur sur la liste gauche et un curseur sur la liste droite. Ont applique la fonction avec les premier élément respectif des deux listes, en fonction du résultat ont avance le curseur de la liste droite ou gauche et ont défini la tête de la `Sublist` a renvoyé avec le curseur correspondant.

Suite a cela, créer un curseur qui va chercher l'élément final. Ont fait une boucle qui va trier chaque élément temps qu'une des deux listes n'est pas vide. A la fin ont vérifie quel liste était vide en premier en fonction ont ajoute a la suite tout les élément correspondant a celle ci. Et ont oublie pas de couper la continuité des liste pour l'output.

Complexité 2n

d. SubList list_mergesort(SubList l, OrderFunctor f)

Pour cette fonction j'ai suivi ce qui était indiqué dans le TP, 1er temps ont découpe en deux partie. (A noter que j'avais eu quelque problème car par étourderie de copier collé j'avais mal défini les pointeur de la sous liste droite). 2nd temps ont fait les récursion sur les deux listes, ont en profite pour définir tout en haut la sortie en cas de liste vide ou a un élément. 3ieme temps ont fusionne les liste avec le merge ont envoie la fusion.

e. List *list_sort(List* l, OrderFunctor f)

C'est la fonction qui regroupe tout. Donc ont convertie les liste a sentinel en sous liste (donc sans sentinel), ont coupe la circularité des sous listes. Ont utilise `merge_sort` pour trier. Puis ont reforme la liste avec sentinel en remettant les pointeurs a la place et ont renvoie la liste trié.

---

7. Conclusion

J'ai eu beaucoup de problème lors de la dernière partie du TP, je croyait avoir fini mais en réalité j'était partie sur de mauvaise base. J'obtiens l'output voulu.

```
-------- TEST PUSH_BACK --------
List (10) : 0 1 2 3 4 5 6 7 8 9
-------- TEST PUSH_FRONT --------
List (10) : 9 8 7 6 5 4 3 2 1 0
Sum is 45
-------- TEST POP_FRONT --------
Pop front : 9
List (9) : 8 7 6 5 4 3 2 1 0
-------- TEST POP_BACK --------
Pop back : 0
List (8) : 8 7 6 5 4 3 2 1
-------- TEST INSERT_AT --------
List (10) : 0 2 4 6 8 9 7 5 3 1
-------- TEST REMOVE_AT --------
List (4) : 2 6 9 5
List cleared (0)
-------- TEST AT        --------
List (10) : 0 1 2 3 4 5 6 7 8 9

------------------------------

-------- TEST SORT      --------
Unsorted list    : List (8) : 5 3 4 1 6 2 3 7
```

```
Decreasing order : List (8) : 7 6 5 4 3 3 2 1
Increasing order : List (8) : 1 2 3 3 4 5 6 7
```