Compte rendu de la séquence 3 - algo3

Durant cette séquence, nous avons implémenté une liste doublement chaînée. Nous devions coder plusieurs fonctionnalités, telles que l'insertion d'élément, une fonction `map` ou encore un trifusion. Le but de ce document est d'expliquer mes choix d'implémentations, et de données une analyse de mon travail.

Implémentation

Opérateurs constructeurs et destruction

list_create()

Cette fonction créer une liste vide

```
List* list_create(void) {

 List* l =  unwrapMalloc(sizeof(List)+sizeof(LinkedElement));
 LinkedElement *p_sentinel = (LinkedElement *)(l+1);

 /*sentinel assignation*/

 return l;
}
```

L'implémentation est relativement simple. On alloue la mémoire puis l'on assigne une sentinelle à notre liste.

gestion de la mémoire

Cette fonction utilise unwrapMaloc, afin d'allouer la mémoire pour notre liste et notre sentinelle, dans un but de continuité de la mémoire, une seule allocation est faite, cela permet aussi qu'au moment de détruire la liste, un seul `free` sera utilisé dans le but de libérer à la fois la sentinelle et la liste.

push_back et push_front

```
List* list_push_back(List* l, int v) {
 LinkedElement *new_el = unwrapMalloc(sizeof(LinkedElement));
 /*assignation de la valeur*/
 /*assignation de l'élément à la liste*/

 //augmentation de la taille de la liste
```

```
 l->size++;
 return l;
}
```

Cette implémentation est encore plus simple que la précédente. On alloue simplement un nouvel élément et on l'assigne à la liste. Grâce à la sentinelle, il n'y a pas de disjonction de cas à faire.

L'implémentation de `push_front` est similaire, pour unique changement d'assigner au début de la liste.

list_delete

Cette fonction vide et libéré la mémoire occupée par une liste

```
void list_delete(ptrList* l) {
 while (!list_is_empty(*l)){
  /*on retire les éléments*/
 }
 //no need to free the sentinel, they share the same alloc with l
 free(*l);
 *l = NULL;

}
```

On vide dans un premier temps la liste de tous ses éléments, ensuite, on libère la mémoire occupée par la liste.

Gestion de la mémoire

Un seul `free` est requis ici afin de libérer la sentinelle et la structure liste, car à la création de la liste, une seule allocation a été utilisé pour allouer la liste et sa sentinelle.

Opérateurs

- suppression d'éléments
- obtention d'information sur la liste
- insertion d'éléments

Suppression d'éléments

Au cours de l'utilisation de ces listes, nous sommes amenés à supprimer des éléments, pour cela plusieurs fonctions sont fournies par cette implémentation. Pour cela, il nous est demandé d'implémenter trois fonctions `list_pop_front`, `list_pop_back` et `list_pop_at`, qui, respectivement, supprime un élément au début de la liste, à sa fin, et a un indice précis.
On se penche sur l'exemple de `list_pop_front`.

```
List* list_pop_front(List* l) {
 assert(!list_is_empty(l));
```

```
 /*on retire les liens du premiere élément de la liste*/
 free(to_remove);
 l->size--;

 return l;
}
```

Cette implémentation est simple, on retire l'élément de la liste, puis on libère, la mémoire, et évidemment on réduit la taille de la liste de 1.
L'implémentation est très similaire pour les autres fonctions, avec pour détail que la fonction `list_pop_at` prend un entier en paramètre afin de savoir quel élément retirer, et que dans son implémentation, elle parcourt les éléments de la liste avant de trouver celui qu'elle doit traiter, sa complexité est donc en O(n), avec n le nombre d'éléments de la liste.

Obtention d'information sur la liste

Nous devons pouvoir donner la possibilité d'obtenir certaines informations sur la liste, sa taille, si elle est vide, et la valeur d'un élément en une certaine position.

Afin de savoir sa taille ou si elle est vide, nous avons la fonction `list_size`.

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

Cette fonction renvoie simplement la taille de la liste enregistrer dans son champ `size`. Dans le but de savoir si la liste est vide, nous avons `list_is_empty`, qui fonctionne comme cette fonction, mais en nous renvoyant un booléen correspondant à si `size` est nulle.

Afin d'obtenir la valeur de certains éléments de la liste, nous avons les fonctions `list_front`, `list_back` et `list_at`, qui, respectivement, accèdent au premier élément de la liste, le dernier élément, et celui présent a un certain indice.

Voici l'implémentation de`list_front`

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

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

Cette fonction nous renvoie simplement la valeur de l'élément qui vient après la sentinelle, pour `list_back` on fait pareil en renvoyant l'élément qui vient avant la sentinelle. Dans le cas de `list_at` on parcourt la liste jusqu'à arriver à l'élément souhaité, sa complexité est donc de O(n).

Insertion d'éléments

L'implémentation donne la possibilité d'insérer un élément à un certain indice précis avec la fonction `list_insert_at`

```
List* list_insert_at(List* l, int p, int v) {
 assert(p <= l->size);
 LinkedElement *cur_elem = l->sentinel->next;
 LinkedElement *to_add = unwrapMalloc(sizeof(LinkedElement));
 to_add->value = v;

 for(int i = 0 ; i<p ; i++){
  /*On parcour la liste jusq'au p-ème élément*/
 }

 /*On insére le nouvel élément en posisition p*/

 l->size++;
 return l;
}
```

Cette fonction ne fait que parcourir la liste jusqu'au p-ème élément, puis ajoute le nouvel élément créé avant cet élément, afin que notre valeur soit bien en position p. On augmente la taille de la liste à la fin de cette opération.

Map

Notre implémentation offre la possibilité d'appliquer une fonction sur chacun des éléments de la liste avec une fonction dite de "mapping", en utilisant donc la fonction `list_map`.
Cette fonction nous permet de modifier une liste avec une fonction, sans avoir à retirer et remettre chaque élément un par un.

```
List* list_map(List* l, ListFunctor f, void* environment) {
 if(l->size == 0)
  return l;

 LinkedElement * elem=l->sentinel;

 do{
  /*application de la fonction à chaque élément de la liste*/
 }while(elem->next!= l->sentinel);

 return l;
}
```

L'implémentation est aussi relativement simple, on parcourt simplement tous les éléments de la liste, en y appliquant la fonction donner `ListFunctor f`.

Trifusion

La partie la plus dure à produire de cette implémentation fût celle de l'écriture d'une fonction de trie, à partir de l'algorithme de trifusion.

Afin de pouvoir avoir une implémentation très efficace, nous avons utilisé une structure résument une liste (une sous-liste dans notre cas) du nom de `Sublist`

```
typedef  struct s_Sublist{

LinkedElement * head;//first elem
LinkedElement * tail;//last elem
}Sublist;
```

Elle représente une liste en ne pointant uniquement qu'au premier élément, la tête et au dernier, la queue.

Cette structure n'est volontairement pas placée dans `list.h` afin que l'utilisateur n'y est pas accès, car elle est utile exclusivement dans l'implémentation de notre algorithme de trie.

Puis une fonction list_split a pour but de séparer une liste en deux sous-listes.

```
Sublist list_split(Sublist l){
Sublist result;
result.head = l.head;

LinkedElement *curpos = l.head;

__uint8_t iter =  0;

while (curpos!=l.tail){
/*On parcour la liste, en passant à l'élément suivant de result.head qu'une
fois sur deux, afin d'atteindre la moitié*/
}

result.tail = result.head->next;
return result;

}
```

Cette fonction parcourt `Sublist l` avec un élément qui passe à son suivant qu'une fois sur deux, afin de pouvoir atteindre le milieu de la liste.

Pour implémenter le trie fusion, nous avons besoin d'une fonction nous permettant de fusionner deux listes triées en une seule triée, pour cela une fonction `list_merge` est écrite.

```
Sublist list_merge(Sublist leftlist, Sublist rightlist,OrderFunctor f){
```

```
Sublist merged;
Sublist *to_treat;
LinkedElement *elem;

//si une des deux liste est vide on renvoie l'autre

if(leftlist.head ==  NULL){

return rightlist;
}

else  if(rightlist.head ==  NULL){

return leftlist;
}

/*on choisi la list à traiter et on assigne elem*/

/*on retire elem de la list */

//on fusionne les deux sous listes, sans elem

merged =  list_merge(leftlist,rightlist,f);



/*on rajoute elem au début de cette list*/

return merged;
}
```

Cette fonction fonctionne en regardant le premier élément des deux listes, en les comparant, on retire l'élément qui est ordonné le plus bas, et on fusionne les deux sous listes tirées entre elle avec le même algorithme, puis on attache notre élément en tête de liste, il n'y a plus qu'à retourner notre élément.

On a aussi l'utilisation d'une fonction qui va trier une sous-liste `list_mergesort`

```
Sublist list_mergesort(Sublist l, OrderFunctor f){
if(l.head == l.tail){
return l;
}

/*séparation de la liste en deux*/

//on retourne une fusion des deux sous listes triée

return  list_merge( list_mergesort(left,f),list_mergesort(right,f),f);



}
```

Cette fonction renvoie donc une sous-liste triée, grâce à une application directe du tri fusion, en utilisant les deux fonctions défini au-dessus.

Toutes les fonctions retournant une `Sublist` ne sont pas publics, c'est-à-dire que l'utilisateur ne peut pas y accéder en utilisant `list.h` avec `list.c`, si elles ont privé, c'est, car elle offre des fonctions écrites uniquement pour être utiles pour notre algorithme de tri. On pose donc une fonction que l'utilisateur utilisera pour trier sa liste, `list_sort`.

```
List*  list_sort(List*  l, OrderFunctor f) {

if(list_is_empty(l))
return l;

Sublist sub_l;
/*convertion de l en Sublist*/



sub_l =  list_mergesort(sub_l,f);



/*convertion de sub_l en List*/

return l;
}
```

Cette fonction ne fait que convertir une liste `List` en liste `Sublist`, pour pouvoir la passer dans notre fonction de tri, puis on reconvertit le résultat en liste `list` que l'on retourne. L'utilisateur peut donc trier une liste avec l'ordonnance de son choix, grâce au paramètre `OrderFunctor f`.

Fonction auxiliaire

unwrapMaloc

Pour éviter les comportements non définis qui peuvent subvenir quand une allocation mémoire échoue avec la fonction `malloc`

Une fonction `unwrapMalloc` est définie.

```
void  *  unwrapMalloc(size_t  size){

void  * result =  malloc(size);

if (!result){
fprintf(stderr,"could not initialize %ld bytes into the heap\n",size);
```

```
perror("stopping program ");
exit(1);
}
return result;
}
```

Elle fait s'arrêter le programme si le malloc échoue, en nous donnant un message d'erreur.