Examen PG116 "Atelier algorithmique et programmation"
G. Eyrolles, F. Herbreteau, F. Mazoit et D. Renault 13 février 2024

Ce devoir consiste à mettre en œuvre la structure de données *piece list* qui est utilisée par de nombreux éditeurs de texte (Abiword, Atom, Visual Code Studio, etc.) pour représenter les modifications apportées au texte en cours d'édition, et mettre en œuvre les opérations *undo* et *redo*.

1. Présentation des *piece lists*

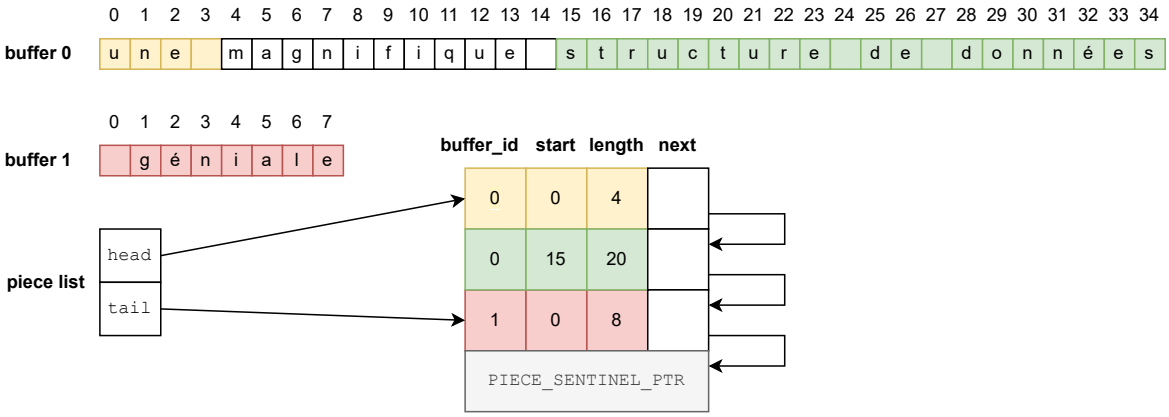


FIGURE 1 – Une *piece list* qui représente le texte "une structure de données géniale".

Une *piece list* représente un texte comme une liste de portions de texte (ou *pieces*). La figure 1 montre une *piece list* constituée de 3 portions (*pieces*) :

— en jaune, la 1ère portion débute au caractère en position 0 de `buffer 0`, d'une longueur de `4` caractères,
— en vert, la 2ème portion débute au caractère en position 15 de `buffer 0`, de longueur 20 caractères,
— enfin, en rouge, la 3ème portion débute au caractère en position 0 de `buffer 1`, de longueur 8 caractères.

La *piece list* représente le texte obtenu en concaténant les trois portions : "une structure de données géniale".

Cette *piece list* a pu, par exemple, être construite à partir de la *piece list* `[{0, 0, 35}]`, qui représente l'intégralité du `buffer 0`, en retirant tout d'abord les caractères en positions 4 à 14. On obtient ainsi la *piece list* `[{0, 0, 4}, {0, 15, 20}]` constituée des portions jaune et verte en figure 1. Puis en ajoutant les caractères en positions 0 à 7 de `buffer 1` en fin de la *piece list* (en position 24). Cette structure permet d'insérer et de supprimer efficacement des caractères dans un texte sans devoir les décaler.

Ce devoir s'intéresse uniquement à la *piece list* : on ignorera les *buffers* qui seront représentés par un numéro (`0` et `1` dans l'exemple en figure 1). Notons qu'un même texte peut être representé par de multiples *piece lists*. On permet notamment de placer des portions de longueur 0 dans la *piece list*. Nos tests automatiques vérifient que vos *piece lists* représentent

les textes escomptés, mais ils ne vérifient généralement pas la structure de la *piece list* elle-même.

Certaines opérations de la *piece list* se basent sur des positions. On souhaitera par exemple "supprimer le caractère en position 4" ou encore "découper la portion qui contient le caractère en position 26 dans le texte". **Les positions utilisées par ces opérations font référence aux positions des caractères dans le texte représenté par la *piece list***. Dans l'exemple en figure 1, la première portion (jaune) représente les caractères de position 0 à 3 dans le texte, la seconde portion (verte), les caractères de position 4 à 23, enfin la troisième portion (rouge) représente les caractères en position 24 à 31. Ainsi donc, "supprimer le caractere en position 4" revient à supprimer le premier caractère de la portion verte, alors que "découper la portion qui contient le caractère en position 26 dans le texte" revient à diviser la portion rouge en deux avant son 3ème caractère (qui est en position 26 dans le texte).

2. Structures de données et opérations

La structure de données choisie pour représentée la *piece list* est donnée en annexe A. La structure **`struct`** `piece_list` est une liste chaînée de portions (*pieces*) qui possède un pointeur `head` sur la première portion, et un pointeur `tail` sur la derniere portion. La liste est terminée par une portion sentinelle `PIECE_SENTINEL_PTR`. Une portion est représentée par la structure **`struct`** `piece` définie par un identifiant de buffer `buffer_id` (ex : `0` ou `1` en figure 1), une position de premier caractère `start`, une longueur `length`, et un pointeur `next` sur la portion suivante dans la liste. La figure 1 illustre cette représentation.

Cette structure de données est très similaire aux listes chaînées vues en TD, avec deux différences notables : la nature des données stockées (ici une portion plutôt qu'un entier en TD), et le pointeur sur le dernier élément de la liste pour ajouter en queue plus efficacement.

L'annexe A liste un ensemble d'opérations :

— `add_head`, `add_after`, `add_tail`, `remove_head` et `remove_after` sont des opérations de chaînage comme celles vues en TD. Ces opérations vous sont fournies ;

— `pl__debug` affiche la structure de données **`struct`** `piece_list`. Cette fonction est fournie ;

— les fonctions `pl__empty`, `pl__free`, `pl__split_piece`, `pl__erase`, `pl__to_string` et `pl__compress` permettent la manipulation de la *piece list*. Elles ne sont pas fournies et doivent être programmées.

Attention à lire attentivement les spécifications des fonctions avant de programmer.

Exercice 1
En vous inspirant de la figure 1, dessiner les *piece list* indiquées en dernière page.

Exercice 2
Mettre en œuvre les fonctions suivantes (l'ordre suggéré tient compte de la difficulté) :

1. `pl__create` et `pl__free`
2. `pl__to_string`
3. `pl__erase`
4. `pl__split_piece`
5. `pl__compress`

3. Questions de cours

Exercice 3
Répondre aux questions suivantes. La clarté et la précision de vos réponses sont primordiales pour une évaluation positive.

1. Que signifie le mot “abstrait” dans un “type abstrait de données” ?
2. Indiquez quels problèmes surviennent lors de l’exécution du programme ci-dessous et comment empêcher cette situation. Justifier.

```
1  struct tests {
2    int * scores;
3    size_t size;
4  };
5
6  void remove_last_test(struct tests * ts)
7  {
8    if (ts->size > 0) {
9      ts->scores[ts->size - 1] = -99;
10     ts->size -= 1;
11   }
12 }
13
14 int sum_of_tests(struct tests * ts)
15 {
16   int sum = 0;
17   for (size_t i = 0; i < ts->size; ++i)
18     sum += ts->scores[i];
19   return sum;
20 }
21
22 int main()
23 {
24   struct tests t1 = ...; //crée t1 avec data = {1, 2, 3, 4, 5}, size = 5
25
26   struct tests t2 = t1;
27
28   remove_last_test(&t1);
29
30   printf("%d\n", sum_of_tests(&t2));
31
32   return 0;
33 }
```

3. On souhaite trier le tableau d’entiers `int t[] = {12,3,1,78,65};` par ordre croissant à l’aide de la fonction fonction `qsort` dont le prototype est donné ci-dessous (voir la page de manuel `man 3 qsort` pour la documentation). Donner le code de la fonction `compar` adéquate et l’appel à `qsort`

```
1  void qsort(void *base, size_t nel, size_t width,
2             int (*compar)(const void *, const void *));
```

A. Fichier en-tête `piece_list.h`

```c
#ifndef PIECE_LIST
#define PIECE_LIST

#include <stddef.h>

/* Représentation d'une portion de texte qui débute au caractère
   `start` du buffer `buffer_id`, de longueur `length` caractères.
   `next` pointe sur la portion suivante dans la piece list
*/
struct piece {
    unsigned int buffer_id;
    size_t start;
    size_t length;
    struct piece * next;
};

/* Représentation d'une piece list comme une liste chaînée de portions,
   terminée par la sentinel PIECE_SENTINEL_PTR (cf. ci-dessous).
   `head` pointe sur la première portion
   `tail` pointe sur la derniere portion
*/
struct piece_list {
    struct piece * head;
    struct piece * tail;
};

/* Sentinelle pour la fin de la piece list (struct piece_list) */
#define PIECE_SENTINEL_PTR ((struct piece *)&piece_sentinel)
extern struct piece const piece_sentinel;

/* Ajoute la portion `p` en tête de la liste `pl`
   PRE: `pl` et `p` ne sont pas NULL
   POST: `p` a été ajoutée en tête de `pl`
   RETOURNE: 0
*/
int add_head(struct piece_list * pl, struct piece * p);

/* Ajoute la portion `p` après la portion `after` dans `pl`
    PRE: `pl`, `after` et `p` ne sont pas NULL
         `p` est une piece de `pl`
    POST: `p` a été ajoutée après `after` dans la liste `pl`
    RETOURNE: -1 si `after` ou `p` est PIECE_SENTINEL_PTR
              0 sinon (ajout effectué)
*/
int add_after(struct piece_list * pl, struct piece * after, struct piece * p);

/* Ajoute la portion `p` en queue de la liste `pl`
    PRE: `pl` et `p` ne sont pas NULL
    POST: `p` a été ajoutée en queue de la liste `pl`
    RETOURNE: -1 si `p` est PIECE_SENTINEL_PTR
              0 sinon (ajout effectué)
*/
int add_tail(struct piece_list * pl, struct piece * p);

/* Retire la portion en tête de la liste `pl`
```

```
56      PRE: `pl` n'est pas NULL
57      POST: la portion en tête de la liste `pl` a ete retirée
58      RETOURNE: NULL si `pl` est vide
59                la portion retirée sinon
60  */
61  struct piece * remove_head(struct piece_list * pl);
62
63  /* Retire la portion qui suit `after` dans la liste `pl`
64      PRE: `pl` et `after` ne sont pas NULL
65           `after` est une portion de la liste `pl`
66      POST: la portion qui suit `after` dans `pl` a été retirée
67      RETOURNE: NULL si `after` ou la portion qui suit `after` est
68                PIECE_SENTINEL_PTR
69                la portion retirée sinon
70  */
71  struct piece * remove_after(struct piece_list * pl, struct piece * after);
72
73  /* Affiche le contenu de la piece list `pl` sur la sortie d'erreur
74  */
75  void pl__debug(const struct piece_list* pl);
76
77  /* Alloue et initialise une liste vide
78      RETOURNE: une liste vide allouée et initialisée
79  */
80  struct piece_list * pl__create();
81
82  /* Libère la mémoire utilisée par la liste `pl`
83      PRE: `pl` n'est pas NULL
84      POST: la mémoire allouée pour `pl` et les portions qu'elle contient a été
85            libérée
86  */
87  void pl__free(struct piece_list * pl);
88
89  /* Retourne une chaîne de caractère terminée par 0 qui contient le texte
90     représenté par la liste `pl` dont les portions sont extraites de `buffers`
91      PRE: `buffers` et `pl` ne sont pas NULL
92           les `buffer_id` des portions de `pl` sont des indices valides dans le
93           tableau `buffers`
94      RETOURNE: la chaîne de caractères terminée par 0 qui contient le texte
95                représenté par la liste `pl` dont les portions sont extraites de
96                `buffers`
97      NOTE: le pointeur retourné doit être libéré par le code qui appelle la
98            fonction `pl__to_string`
99  */
100 char * pl__to_string(struct piece_list const * pl, char const * buffers[]);
101
102 /* Découpe la portion de la liste `pl` qui contient le caractère en position
103    `pos` en deux portions, et retourne ladite portion
104     PRE: `pl` n'est pas NULL
105     POST: la portion {buffer_id, start, length} qui contient le caractère en
106           position `pos` dans `pl` a été scindée en deux portions dans `pl`:
107           {buffer_id, start, l} pour la première et
108           {buffer_id, start+l, length-l} pour la seconde
109           où `l` est le nombre de caractères avant celui en position `pos`
110           dans `pl`
111           La seconde portion a été ajoutée à la suite de la première dans la
112           liste `pl`
```

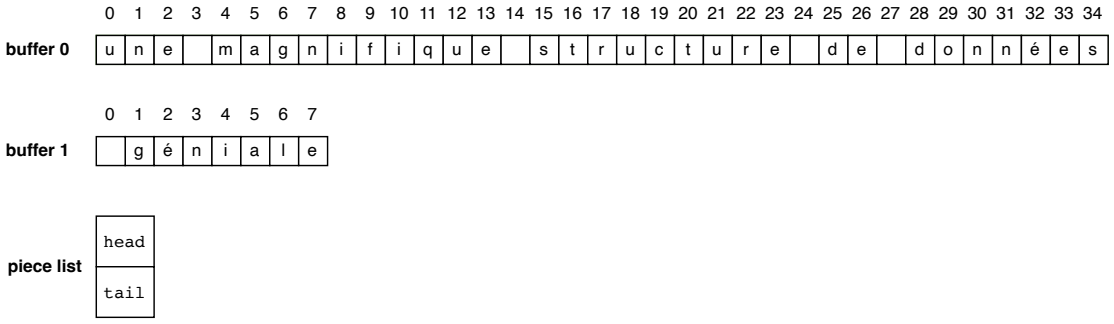
```
113      RETOURNE: un pointeur sur la portion qui a été scindée si `pos` est une
114                position valide dans `pl`
115                NULL sinon
116
117      Exemple 1:
118      `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 9}] et `pos` = 4
119      le caractère en position 4 dans `pl` est le caractère en position 2
120      dans la portion {1, 4, 5}. La fonction `pl__split_piece` scinde donc la
121      portion {1, 4, 5} en {1, 4, 2} et {1, 6, 3} dans `pl`:
122      `pl` = [{0, 0, 2}, {1, 4, 2}, {1, 6, 3}, {0, 7, 9}]
123      et retourne un pointeur sur la portion {1, 4, 2} qui vient d'être scindée
124
125      Exemple 2:
126      `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 9}] et `pos` = 0
127      le caractère en position 0 dans `pl` est le caractère en position 0
128      dans la portion {0, 0, 2}. La fonction `pl__split_piece` scinde donc la
129      portion {0, 0, 2} en {0, 0, 0} et {0, 0, 2} dans `pl`:
130      `pl` = [{0, 0, 0}, {0, 0, 2}, {1, 4, 5}, {0, 7, 9}]
131      et retourne un pointeur sur la portion {0, 0, 0} qui vient d'être scindée
132
133      Exemple 3:
134      `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 9}] et `pos` = 23
135      la fonction `pl__split_piece` retourne NULL car `23` n'est pas une position
136      valide
137
138      Exemple 4:
139      `pl` = [{0, 0, 2}, {1, 0, 0}, {2, 4, 5}] et `pos` = 2
140      le caractère en position 2 dans `pl` est le caractère en position 0
141      dans la portion {2, 4, 5}. La fonction `pl__split_piece` scinde donc la
142      portion {2, 4, 5} en {2, 4, 0} et {2, 4, 5} dans `pl`:
143      `pl` = [{0, 0, 2}, {1, 0, 0}, {2, 4, 0}, {2, 4, 5}]
144      et retourne un pointeur sur la portion {2, 4, 0} qui vient d'être scindée
145  */
146  struct piece * pl__split_piece(struct piece_list * pl, size_t pos);
147
148  /* Efface le caractère en position `pos` dans la liste `pl`
149      PRE: `pl` n'est pas NULL
150      POST: le caractère en position `pos` a été retiré de la liste `pl`
151      RETOURNE: -1 si la suppression n'a pas pu être effectuée (`pos` invalide,
152                ...)
153                0 si la suppression a pu être effectuée
154
155      Exemple 1:
156           `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 9}] et `pos` = 4
157           la fonction `pl__erase` retourne 0 et la liste `pl` devient:
158           `pl` = [{0, 0, 2}, {1, 4, 2}, {1, 7, 2}, {0, 7, 9}]
159
160      Exemple 2:
161           `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 9}] et `pos` = 15
162           la fonction `pl__erase` retourne 0 et la liste `pl` devient:
163           `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 8}, {0, 16, 0}]
164
165      Exemple 3:
166           `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 7, 9}] et `pos` = 56
167           La fonction `pl__erase` retourne -1 car `pos` n'indique pas une
168           position valide dans la liste `pl`
169
```

```
170      NOTE: La fonction `pl__erase` ne cherche pas à "simplifier" la liste.
171      En particulier, elle peut créer des portions de longueur 0 dans la liste
172      `pl`
173  */
174  int pl__erase(struct piece_list * pl, size_t pos);
175
176  /* Retire les portions de longueur 0, et fusionne les portions consécutives
177     adjacentes dans la liste `pl`
178      PRE: `pl` n'est pas NULL
179      POST: `pl` ne contient pas de portion de longueur 0, ni de portions
180             consécutives adjacentes.
181             le texte représenté par `pl` n'a pas été modifié
182
183      Exemple 1:
184          `pl` = [{0, 0, 2}, {1, 4, 5}, {0, 2, 9}]
185          la fonction `pl__compress` laisse la liste `pl` inchangée
186
187      Exemple 2:
188          `pl` = [{1, 4, 0}, {2, 3, 0}, {0, 2, 7}, {1, 4, 0}, {6, 2, 0},
189                  {7, 89,0}, {0, 9, 3}, {1, 12, 1}, {1, 13, 2}, {1, 15, 5}
190                  {3, 4, 0}]
191          la fonction `pl__compress` met à jour `pl` en retirant les portions de
192          longueur 0 et en fusionnant les portions adjacentes consécutives:
193          `pl` = [{0, 2, 10}, {1, 12, 8}]
194  */
195  void pl__compress(struct piece_list * pl);
196
197  #endif // PIECE_LIST
```

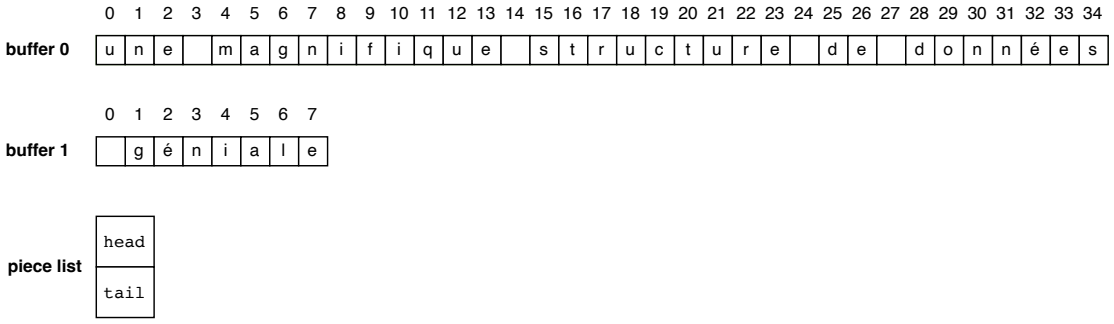
NOM : **PRÉNOM :**

En vous inspirant de la figure 1, compléter les figures ci-dessous afin qu'elles représentent :

1. la *piece list* vide ;



2. la *piece list* obtenue à partir de l'exemple en figure 1 après avoir découpé la portion contenant le caractère en position 12 dans le texte représenté par la *piece list*, avant ce caractère ;



3. une *piece list* obtenue à partir de l'exemple en figure 1 après avoir supprimé le caractère en position 6 dans le texte représenté par la *piece list*.