Exercice 1 : Maillons chainés en C

Les maillons chaînés sont une structure de données séquentielle qui représente une collection de valeurs. L'un des intérêt des maillons chaînés par rapport à un tableau est qu'il est aisé de supprimer ou ajouter des nouveaux éléments entre les éléments déjà présents. Ainsi que le fait que la taille de la collection n'est pas strictement définie à sa création (et qu'elle peut donc changer au fil de l'exécution du programme).

Q. 1 En utilisant le fait qu'il est possible de définir une structure récursive, définir une structure `struct maillon` contenant :
- un entier (de nom `value`) ;
- un pointeur vers une structure `struct maillon` (de nom `next`)

Q. 2 En déduire une définition de type `int_chain` représentant une collection d'éléments par maillons chaînés.

Dans la suite on appelle *chaine* une `int_chain`.

Q. 3 Implanter une fonction `int_chain empty()` retournant une chaine vide.

Q. 4 Implanter une fonction `int_chain push(int value, int_chain chain);` qui permet d'ajouter un nouveau noeud au début de la collection.

Q. 5 Implanter une fonction `bool is_empty(int_chain l);` permettant de tester si une chaine est vide.

Q. 6 Implanter une fonction `int_chain next(int_chain chain)` retournant la collection d'éléments privée de l'élément en tête de la chaîne.

Q. 7 Implanter une version récursive et une version itérative de la fonction `void print_chain(int_chain chain);` permettant l'affichage de la chaîne passée en argument. Utiliser cette fonction pour vérifier que vos fonctions fonctionnent correctement.

Q. 8 Implanter une version récursive et une version itérative de la fonction `void free_chain(int_chain chain);` permettant de libérer l'espace occupé par la chaîne passée en argument.

Q. 9 Implanter une fonction récursive et une version itérative de la fonction `int size_chain(int_chain chain);` calculant la taille d'une chaîne.

Q. 10 Implanter une fonction `int_chain remove_index(int_chain chain, int i);` permettant de supprimer l'élément d'indice `i` de la chaîne `chain`.

Q. 11 On vous donne les trois fonctions suivantes calculant la concaténation de deux listes.

```
int head(int_chain l) {
  return l->value;
}

int_chain concatenation1(int_chain l1, int_chain l2) {
  if (is_empty(l1)) {return l2;}
  else {
    int_chain ptr;
    for (ptr = l1; ptr->next != NULL; ptr=ptr->next) {}
    ptr->next = l2;
```

```
    return l1;
  }
}

int_chain concatenation2(int_chain l1, int_chain l2) {
  if (is_empty(l1) && is_empty(l2)) {return Empty;}
  else if (is_empty(l1)) {return l2;}
  else { return push(head(l1), concatenation2(next(l1), l2)); }
}

int_chain concatenation3(int_chain l1, int_chain l2) {
  if (is_empty(l1) && is_empty(l2)) {return Empty;}
  else if (is_empty(l1)) {return push(head(l2), concatenation3(l1,
↪   next(l2)));}
  else { return push(head(l1), concatenation3(next(l1), l2)); }
}
```

Pour chacune des trois fonctions donner une représentation de la mémoire au point de programme ① en supposant que la fonction main soit la suivante :

```
1 int main() {
2   int_chaine l1 = push(1, Empty) ;
3   int_chaine l2 = push(2, Empty) ;
4   int_chaine l = concatenation(l1, l2);①
5 }
```

Q. 12 Définir une fonction `int_chaine reverse(int_chaine chain);` impérative renversant une chaine.

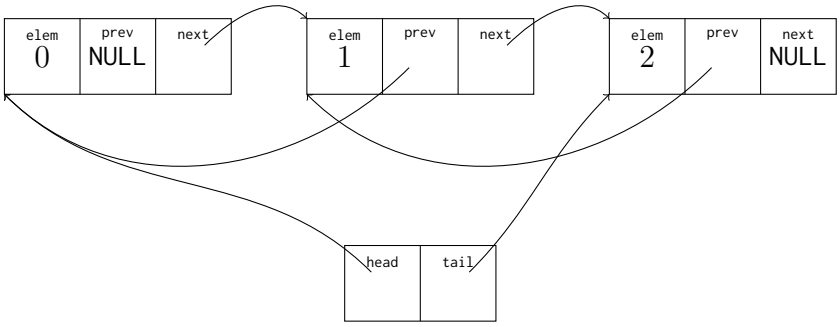
Q. 13 Définir une fonction `int_chaine rev_append(int_chaine chain, int_chaine acc)` récursive retournant une copie du renversement de la chaîne `chain` concaténée à la chaîne `acc`.

Q. 14 En déduire une fonction `int_chaine reverse2(int_chaine chain)` retournant une copie renversée de la chaîne `chain`.

Exercice 2 : Maillons doublement chaînées

Nous avons vu qu'il était difficile d'accéder aux derniers éléments d'une collection chaînée (nécessité de parcourir toute la collection). Ceci peut être ennuyeux lorsqu'on aimerait avoir accès à la fois au début et à la fin d'une chaine (par exemple pour représenter une file d'attente où les gens entrent d'un côté et sortent de l'autre).
Dans cet exercice nous définissons une nouvelle structure de donnée, les maillons doublement chaînées permettant un accès à la fois aux éléments en tête et en fin de collection.
Une collection doublement chainées sera donc un pointeur vers une structure contenant :
`head` un pointeur vers la tête de la collection ;
`tail` un pointeur vers la fin de la collection ;
De plus afin de faciliter la manipulation de la collection (par exemple, la suppression du dernier élément) il est important de garder en mémoire l'élément précédant le dernier élément de la collection (sous peine d'avoir à reparcourir toute la chaine pour devoir trouver le nouveau dernier élément). Ainsi en plus de l'usuel pointeur vers le prochain élément de la liste nous devons garder un mémoire pour chaque élément, un pointeur vers l'élément le précédant dans la chaîne. Ainsi chaque maillon de notre chaîne aura :

`elem` un entier ;
`prev` un pointeur vers le maillon qui le précède.
`next` un pointeur vers le maillon de la liste qui le suit.
Ci-dessous une représentation graphique d'une telle collection doublement chaînée contenant les éléments $1, 2, 3$ dans cet ordre.



Q. 1 Définir la structure `struct dchains` contenant un entier `elem`, un pointeur `prev` vers une `struct dchains` et un pointeur `next` vers une `struct dchains`.

Q. 2 Définir le type `dchain` comme étant un pointeur structure contenant un pointeur `head` vers une `struct dchains` et un pointeur `tail` vers une `struct dchains`.

Q. 3 Définir une fonction `dchain empty();` renvoyant une `dchain` vide. La chaine vide *ne* sera *pas* représentée par un pointeur `NULL` mais par un pointeur vers une structure dont les deux champs `head` et `tail` sont des pointeurs `NULL`.

Q. 4 Définir une fonction `int is_empty(dchain dl);` permettant de tester si une chaine est vide.

Q. 5 Définir une fonction `void print_dchain(dchain dc);` permettant l'affichage de la chaine passée en argument.

Q. 6 Définir une fonction `void push_head(int elem, dchain dc);` permettant l'ajout en place d'un élément en tête de chaine.

Q. 7 Définir une fonction `int pop_head(int elem, dchain dc);` permettant la suppression en place d'un élément en tête de chaine, on retournera l'élément ainsi supprimé.

Q. 8 Définir une fonction `void push_tail(int elem, dchain dc);` permettant l'ajout en place d'un élément en fin de chaine.

Q. 9 Définir une fonction `int pop_tail(int elem, dchain dc);` permettant la suppression en place d'un élément en fin de chaine, on retournera l'élément ainsi supprimé.

Q. 10 Définir une fonction `void reverse(dchain dc);` permettant de renverser en place la chaine passée en argument.

Q. 11 Définir une fonction `void rem(dchain dc, int e);` permettant la suppression de la valeur `e` de la chaine `dc`, on pourra supposer que `e` apparaît au plus une fois dans `dc`.