Jean-Baptiste Bianquis et Lilian Besson Si besoin : mp2i@besson.link

TD XXIII

STRUCTURES DE DONNÉES CHAÎNÉES EN C

On rappelle les options de compilations (particulièrement importantes aujourd'hui) :

```
$ gcc -o fichier.out -fsanitize=address,undefined -Wall -Wextra -Wvla fichier.c
```

On ignorera les *warnings* relatifs aux fuites de mémoire jusqu'à avoir écrit la fonction `free_list` (mais on y sera très attentif ensuite).

I Listes simplement chaînées

Plusieurs variations sont possibles pour définir des listes simplement chaînées en C. Ici, on va utiliser l'une des plus simples :

```c
typedef int datatype;

struct Node {
  datatype data;
  struct Node *next;
};

typedef struct Node node;
```

- En général, on n'aura pas besoin de supposer que `datatype` est égal à `int`, mais on supposera en revanche que c'est un type numérique (pour pouvoir faire des additions, des comparaisons...).
- Une liste sera simplement un `node*` :
  - la liste vide est représentée par le pointeur `NULL` ;
  - une liste non vide est représentée par un pointeur vers son premier nœud.

Exercice XXIII.1 – Création de listes

p. 7

1. Écrire une fonction créant un nouveau `node` (et renvoyant un pointeur vers ce `node`), avec le champ `data` initialisé avec l'argument fourni et le champ `next` initialisé à `NULL`.

   ```c
   node *new_node(datatype data);
   ```

2. Écrire une fonction `cons` prenant en entrée une liste `list` et un argument `data`, et renvoyant une liste constitué d'un nœud contenant `data`, suivi de `list`.

   ```c
   node *cons(node *list, datatype data);
   ```

3. Écrire une fonction `from_array` ayant le prototype suivant :

   ```c
   node *from_array(datatype array[], int len);
   ```

   - `array` est un tableau d'objets de type `datatype` de longueur `len`.
   - La fonction renvoie une liste contenant les mêmes éléments que `array`, dans l'ordre (autrement dit, l'élément de tête de la liste est `array[0]`).

À partir de ce point, vous trouverez dans le squelette un certain nombre de fonctions vous permettant de tester votre code. Il est strictement interdit de passer à la question $n+1$ si le test associé à la question n échoue !

Exercice XXIII.2 – Parcours de listes

p. 7

1. Écrire une fonction `free_list` qui libère toute la mémoire utilisée par une liste.

```
void free_list(node *u);
```

2. Écrire une fonction `length` calculant la longueur d'une liste.

```
int length(node *u);
```

On écrira une fonction purement itérative.

3. Écrire une fonction `print_list` affichant les éléments d'une liste. On produira l'affichage suivant, terminé ou non par un retour à la ligne suivant la valeur du paramètre `newline` :

```
[7 1 3 4 12]
```

On pourra supposer ici que `datatype` est égal à `int`.

```
void print_list(node *u, bool newline);
```

4. Écrire une fonction `to_array` convertissant une liste de longueur n en un tableau de taille n (l'élément en tête de la liste se retrouvera à l'indice 0).

```
datatype *to_array(node *u);
```

5. Écrire une fonction `is_equal` qui teste l'égalité de deux listes. On parle ici d'égalité *structurelle* (les deux listes ont les mêmes éléments, dans le même ordre) et non d'égalité *physique* (les deux pointeurs désignent la même liste).

```
bool is_equal(node *u, node *v);
```

Exercice XXIII.3 – Ordre et tri

p. 9

1. Écrire une fonction (non récursive) `is_sorted` qui prend en entrée une liste et renvoie un booléen indiquant si elle est triée par ordre croissant.

```
bool is_sorted(node *u);
```

2. Écrire une fonction récursive `insert_rec` ayant le prototype suivant :

```
node *insert_rec(node *u, datatype x);
```

   - `u` est une liste (éventuellement vide) supposée triée par ordre croissant.
   - La fonction renvoie une liste triée par ordre croissant contenant les mêmes éléments que `u`, plus une occurrence de `x`.
   - Cette fonction créera un seul nouveau nœud !

3. Faire un schéma mémoire illustrant ce qui se passe, en OCaml d'une part et en C d'autre part, lorsque :

- on part de u vide et l'on fait `v = insert(u, 10)` (ou `let v = insert u 10`);
- on part de $u = (1,2,5,6)$ et l'on fait `v = insert(u, 0)` (ou l'équivalent OCaml);
- on part de $u = (1,2,5,6)$ et l'on fait `v = insert(u, 4)` (ou l'équivalent OCaml).

4. Pourquoi vaut-il mieux considérer que la variable `u` est « invalide » après l'instruction `v = insert(u, x)` (autrement dit, pourquoi vaut-il mieux se limiter à des instructions du type `u = insert(u, x)`)?
5. Pourquoi serait-il problématique d'avoir en C une fonction d'insertion se comportant comme en OCaml?
6. Écrire une fonction récursive `insertion_sort_rec` prenant en entrée une liste (éventuellement vide) et renvoyant une liste triée contenant les mêmes éléments. La liste renvoyée ne partagera pas sa représentation mémoire avec la liste initiale (qui n'aura pas été modifiée).

```
node *insertion_sort_rec(node *u);
```

Exercice XXIII.4

p. 10

1. Écrire une fonction `reverse_copy` qui prend en argument une liste et renvoie une copie de cette liste, à l'envers. La liste et sa copie seront totalement indépendantes en mémoire.

```
node *reverse_copy(node *u);
```

2. Écrire une fonction `copy_rec`, récursive, qui effectue une copie d'une liste (à l'endroit).

```
node *copy_rec(node *u);
```

3. Écrire une fonction `copy` ayant la même spécification mais n'étant pas récursive.
4. Écrire une fonction `reverse` renversant une liste « en place » (c'est-à-dire sans créer aucun nouveau nœud). On renverra un pointeur vers le nœud de tête de la nouvelle liste (qui était donc le nœud de queue de l'ancienne liste).

```
node *reverse(node *u);
```

Exercice XXIII.5

p. 11

Nous avons déjà utilisé la fonction `scanf` : l'appel `scanf("%d", &n)`, par exemple, lit un entier sur l'entrée standard (en commençant par sauter les caractères d'espacement) et stocke sa valeur dans la variable `n` (qui doit avoir été préalablement définie). Cependant, nous avons omis un point important : `scanf` renvoie un entier. Cet entier est égal :

- à la constante (strictement négative) prédéfinie `EOF` si l'on est arrivé à la fin de l'entrée sans rien pouvoir lire;
- au nombre d'éléments que l'on a réussi à lire, sinon. Par exemple, un appel `scanf("%d %f", &n, &x)` renverra 2 si l'entrée commence par `"12 3.14"`, 1 si elle commence par `"12 P3.14"` et 0 si elle commence par `"bonjour 17"`.

1. Écrire une fonction `read_list` qui lit des entiers sur l'entrée standard tant que c'est possible, et renvoie la liste lue (dans l'ordre). On s'arrêtera si l'on obtient `EOF` et aussi si l'on obtient 0 (comme résultat de `scanf`).
2. Écrire un programme complet qui lit une série d'entiers sur l'entrée standard et écrit ces entiers par ordre croissant sur la sortie standard, à raison d'un par ligne.
3. Utiliser ce programme pour lire un fichier d'entiers et écrire un fichier contenant les mêmes entiers triés par ordre croissant.

2 Piles et files

En général, il est nettement plus pertinent d'utiliser des tableaux plutôt que des listes chaînées pour implémenter les piles et les files en C (les tableaux peuvent être dynamiques si l'on ne peut pas se contenter de structures ayant une capacité fixée à la création). Il faut donc voir ce qui suit comme des exercices, intéressants dans le cadre scolaire mais ne correspondant pas vraiment à ce qu'on ferait en réalité.

2.1 Pile à l'aide d'une liste chaînée

On pourrait fournir l'interface minimale d'une pile en utilisant simplement un `node*` (une liste, autrement dit). Ici, on choisit une variante légèrement différente :

```
struct Stack {
  int len;
  node* top;
};

typedef struct Stack stack;
```

Exercice XXIII.6 p. 12

Écrire les fonctions suivantes :

1. `empty_stack` qui renvoie un pointeur vers une nouvelle pile vide ;
2. `peek` qui renvoie l'élément situé au sommet d'une pile (on mettra un `assert` pour vérifier que la pile n'est pas vide) ;
3. `push` qui rajoute un élément au sommet d'une pile ;
4. `pop` qui supprime l'élément situé au sommet d'une pile, et le renvoie (à nouveau, on mettra une assertion pour générer une erreur prévisible si la pile est vide) ;
5. `free_stack` qui libère toute la mémoire associée à une pile.

```
stack *empty_stack(void);

datatype peek(stack *s);

void push(stack *s, datatype x);

datatype pop(stack *s);

void free_stack(stack *s);
```

On fera bien attention à maintenir l'invariant sur la longueur (`s.len` doit toujours être égal à la longueur de la liste `s.top`).

2.2 File à l'aide d'une liste simplement chaînée

On peut implémenter une file de manière efficace [1] en utilisant une liste simplement chaînée, à condition que cette liste soit mutable. En C cela ne posera aucun problème, mais en OCaml cela demanderait de définir un nouveau type liste. L'idée a été expliquée dans le chapitre *Piles et files*, on remet les illustrations :

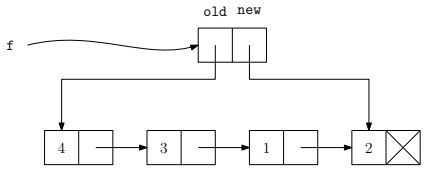

Figure XXIII.7 – La file $\leftarrow (4,3,1,2) \leftarrow$ (4 est l'élément le plus ancien).

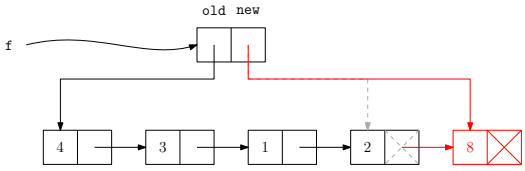

Figure XXIII.8 – Ajout de l'élément 8.

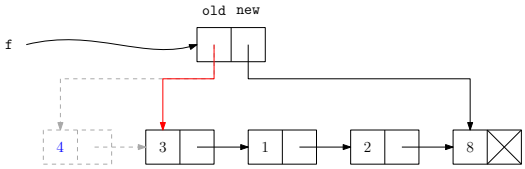

Figure XXIII.9 – Extraction d'un élément.

À nouveau, on choisit de stocker également la longueur dans la structure de file. On obtient donc :

```c
struct Queue {
  int len;
  node *left;
  node *right;
};

typedef struct Queue queue;
```

La convention est la même que sur les illustrations :

- les pointeurs de la liste vont de la gauche vers la droite ;
- les insertions se font à droite ;
- les extractions se font à gauche.

Exercice XXIII.7 p. 13

1. Pourrait-on (efficacement) faire des insertions à gauche ? des extractions à droite ?
2. Écrire les fonctions suivantes, dont la spécification devrait être claire :

1. C'est-à-dire avec insertion et extraction en $O(1)$, le facteur constant étant en revanche assez mauvais par rapport à ce qu'on obtiendrait avec un tableau.

```c
queue *empty_queue(void);

void free_queue(queue *q);

datatype peek_left(queue *q);

void push_right(queue *q, datatype data);

datatype pop_left(queue *q);
```

3 Retour sur les listes chaînées

Exercice XXIII.8 p. 14

Écrire une version purement itérative du tri insertion.

Exercice XXIII.9 p. 15

1. Écrire une fonction `split` qui prend en argument une liste u et un entier n et a le comportement suivant :
   - elle renvoie une liste composée des $|u| - n$ derniers éléments de u ;
   - elle modifie u de manière à ce qu'elle ne contienne plus que ses n premiers éléments.

   Cette fonction sera purement itérative, ne créera aucun nouveau nœud, et traitera les cas où $n > |u|$ avec des assertions.

   ```c
   node *split(node *u, int n);
   ```

2. Écrire une fonction `merge` qui prend en entrée deux listes supposées croissantes et renvoie leur fusion. Cette fonction sera purement itérative et ne créera aucun nouveau nœud.

   ```c
   node *merge(node *u, node *v);
   ```

3. Écrire une fonction `merge_sort` qui trie une liste en utilisant l'algorithme du tri fusion. Ce tri se fera « en place », dans le sens où l'on ne créera aucun nouveau nœud (on ne fera que réarranger des pointeurs).

   ```c
   node *merge_sort(node *u);
   ```

Solutions

Correction de l'exercice XXIII.1 page I

1. On alloue un nouveau nœud (sur le tas) et on l'initialise :

```
node *new_node(datatype data){
  node *new = malloc(sizeof(node));
  new->data = data;
  new->next = NULL;
  return new;
}
```

2. On renvoie un pointeur vers un nouveau nœud, dont on a fait pointer le champ `next` vers la liste fournie :

```
node *cons(node *list, datatype data){
  node *new = new_node(data);
  new->next = list;
  return new;
}
```

3. Il faut parcourir le tableau à l'envers pour obtenir la liste dans le bon ordre :

```
node *from_array(datatype array[], int len){
  node *current = NULL;
  for (int i = len - 1; i >= 0; i--) {
    current = cons(current, array[i]);
  }
  return current;
}
```

Correction de l'exercice XXIII.2 page 2

1. On libère les nœuds dans l'ordre :

```
void free_list(node* n) {
  while (n != NULL) {
    node *next = n->next;
    free(n);
    n = next;
  }
}
```

Il est possible de procéder récursivement :

```c
void free_list_rec(node *n){
  if (n == NULL) { return; }
  free_list_rec(n->next);
  free(n);
}
```

Cependant, cette fonction (qui commence en fait par libérer le *dernier* nœud), n'est pas récursive terminale et peut donc poser des problèmes de dépassement de pile (la profondeur de récursion est égale à la longueur de la liste).

2. Pas de difficulté :

```c
int length(node* list){
  int i = 0;
  while (list != NULL) {
    i++;
    list = list->next;
  }
  return i;
}
```

3. Pour respecter exactement l'affichage demandé, il faut être méticuleux : tous les éléments *sauf le dernier* sont suivis d'une espace.

```c
void print_list(node *n, bool newline){
  printf("[");
  while (n != NULL && n->next != NULL) {
      printf("%d ", n->data);
      n = n->next;
  }
  if (n != NULL) {
    printf("%d", n->data);
  }
  printf("]");
  if (newline) { printf("\n"); }
}
```

4. On détermine la longueur, on alloue, puis on parcourt la liste pour remplir le tableau :

```c
datatype *to_array(node *u){
  int len = length(u);
  datatype *t = malloc(len * sizeof(datatype));
  for (int i = 0; i < len; i++){
    t[i] = u->data;
    u = u->next;
  }
  return t;
}
```

5. Attention, il est facile d'écrire une fonction qui considère comme égales les listes `[1 2 3]` et `[1 2]` !

```c
bool is_equal(node *u, node *v){
  while (u != NULL && v != NULL){
    if (u->data != v->data) { return false; }
    u = u->next;
    v = v->next;
  }
  return (u == NULL && v == NULL);
}
```

Correction de l'exercice XXIII.3 page 2

1. À nouveau, on privilégie une version purement itérative :

```c
bool is_sorted(node *u){
  if (u == NULL) { return true; }
  while (u->next != NULL){
    if (u->data > u->next->data) { return false; }
    u = u->next;
  }
  return true;
}
```

2. En récursif ça ne pose aucun problème :

```c
node *insert_rec(node *u, datatype x){
  if (u == NULL || u->data >= x) { return cons(u, x); }
  u->next = insert_rec(u->next, x);
  return u;
}
```

3. On obtient les schémas suivants :

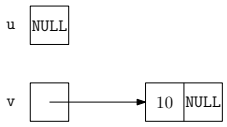

Figure XXIII.10 – C, insertion dans une liste vide.

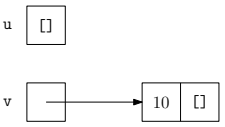

Figure XXIII.11 – OCaml, insertion dans une liste vide.

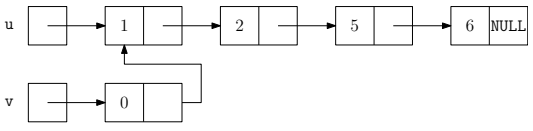

Figure XXIII.12 – C, insertion en tête.

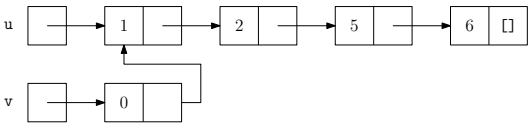

Figure XXIII.13 – OCaml, insertion en tête.

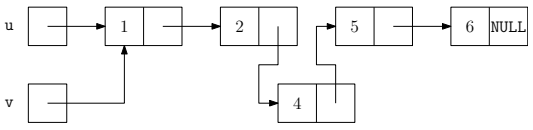

Figure XXIII.14 – C, insertion en milieu de liste.

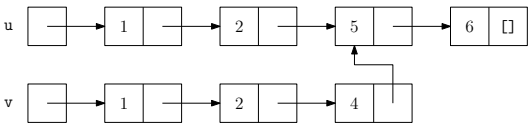

Figure XXIII.15 – OCaml, insertion en milieu de liste.

4. Après un appel `v = insert(u, x)`, on a plusieurs situations possibles :

- Soit u était vide, et dans ce cas u vaut toujours `NULL` et v n'a fondamentalement « rien à voir » avec u.
- Soit u était non vide et on a inséré en tête. Dans ce cas, u pointe vers le deuxième élément de v, ce qui est une situation dangereuse. Une modification de l'un peut ou non affecter l'autre suivant les cas, et la libération est très dangereuse : si l'on appelle `free_list(v)`, alors u pointe vers un bloc qui n'existe plus ; si l'on appelle `free_list(u)`, alors l'élément de tête de v pointe vers un bloc qui n'existe plus.
- Soit u était non vide et on a inséré ailleurs qu'en tête. Dans ce cas, u et v désignent tous deux la nouvelle liste (celle avec un élément en plus). Le point vraiment problématique, c'est que la situation est complètement différente du cas précédent !

5. Si la fonction C se comportait comme la fonction OCaml (u n'est pas modifiée, le début est recopié avec un élément de plus et la fin est partagée entre u et v), on serait essentiellement toujours dans un cas équivalent au deuxième cas de la question précédente. C'est (très) problématique pour les raisons précédemment énoncées (en particulier, il est essentiellement impossible de libérer correctement la mémoire).
   Il serait bien sûr possible d'écrire une fonction qui agit comme la version OCaml, sauf qu'elle libère les nœuds de u au fur et à mesure qu'elle les copie. Cependant, ce serait assez absurde : autant ne pas faire de copie.
   En OCaml, il n'y a pas de problème pour deux raisons :
   - la libération de la mémoire est automatique ;
   - les listes ne sont pas mutables, donc il n'est pas gênant d'avoir une situation dans laquelle u et v sont partiellement aliasées (même sans savoir quelle partie de leur représentation mémoire elles partagent).
6. Pas de difficulté majeure :

```c
node *insertion_sort_rec(node *u){
  if (u == NULL) { return NULL; }
  return insert_rec(insertion_sort_rec(u->next), u->data);
}
```

Correction de l'exercice XXIII.4 page 3

1. Essentiellement, c'est une copie inversée de pile, ce qui ne pose pas de problème :

```c
node* reverse_copy(node* u){
  node *v = NULL;
  while (u != NULL){
    v = cons(v, u->data);
    u = u->next;
  }
  return v;
}
```

2. La version récursive est très simple :

```c
node *copy_rec(node *u){
  if (u == NULL) { return NULL; }
  return cons(copy_rec(u->next), u->data);
}
```

3. La version itérative est nettement plus délicate, quelques expériences avec *C Tutor* (pour rappel c'est sur : `https://pythontutor.com/c.html`) peuvent sans doute être utiles :
   - u désigne le nœud courant de la liste passée en argument (qui varie au cours de

l'exécution) ;
- `current` désigne le nœud courant de la copie ;
- `start` sauvegarde le premier nœud de la copie (puisque c'est ce que l'on doit renvoyer à la fin).

```
node* copy(node* u){
  if (u == NULL) { return NULL; }
  node* current = new_node(u->data);
  node* start = current;
  while (u->next != NULL) {
    u = u->next;
    node* new = new_node(u->data);
    current->next = new;
    current = current->next;
  }
  return start;
}
```

4. C'est aussi assez délicat. Noter que cette fois, c'est le dernier nœud (devenu le premier par inversion des pointeurs) que l'on doit renvoyer :

```
node *reverse(node *u){
  if (u == NULL) { return u; }
  node *current = u;
  node *next = u->next;
  current->next = NULL;
  while (next != NULL){
    node *tmp = current;
    current = next;
    next = next->next;
    current->next = tmp;
  }
  return current;
}
```

Correction de l'exercice XXIII.5 page 3

1. Cette version n'est pas très idiomatique, mais elle est conforme au programme et facile à comprendre :

```
node *read_list(void){
  node *u = NULL;
  while (true){
    int x;
    int code_retour = scanf("%d", &x);
    if (code_retour == 1){
      u = cons(u, x);
    } else {
      return u;
    }
  }
}
```

2. On commence par écrire une petite fonction pour gérér l'écriture :

```c
void print_list_2(node *u){
  while (u != NULL){
    printf("%d\n", u->data);
    u = u->next;
  }
}
```

Ensuite, le `main` est très simple :

```c
int main(void) {

  node *u = read_list();
  node *v = insertion_sort_rec(u);
  print_list_2(v);

  free_list(u);
  free_list(v);

  return 0;
}
```

3. Il suffit d'utiliser la ligne de commande suivante (où l'on redirige l'entrée standard, ici depuis le fichier `input.txt`, et la sortie standard, ici vers le fichier `output.txt`) :

```
$ ./linked.out <input.txt >output.txt
```

Correction de l'exercice XXIII.6 page 4

1. Tant qu'on pense bien à allouer sur le tas, il n'y a pas de problème :

```c
stack *empty_stack(void){
  stack *s = malloc(sizeof(stack));
  s->len = 0;
  s->top = NULL;
  return s;
}
```

2. Aucun problème :

```c
datatype peek(stack *s){
  assert(s->len > 0);
  // le second test est superflu si l'invariant avec s->len est maintenu
  assert(s->top != NULL);
  return s->top->data;
}
```

3. Il faut juste penser à mettre à jour la longueur :

```c
void push(stack *s, datatype x){
  s->top = cons(s->top, x);
  s->len = s->len + 1;
}
```

4. Rien de compliqué, mais il faut faire attention à faire les choses dans l'ordre.

```
datatype pop(stack *s){
  assert(s->len > 0);
  // le second test est superflu si l'invariant avec s->len est maintenu
  assert(s->top != NULL);
  node *top = s->top;
  datatype res = top->data;
  s->top = top->next;
  s->len = s->len - 1;
  free(top);
  return res;
}
```

5. On libère la liste, puis la `struct` elle-même.

```
void free_stack(stack *s){
  free_list(s->top);
  free(s);
}
```

Correction de l'exercice XXIII.7 page 5

1. Il n'y a aucun problème pour insérer à gauche. En revanche, il faudrait parcourir toute la liste pour supprimer à droite puisqu'on n'a pas de pointeur vers l'avant-dernier élément.
2. Pour `push_right`, il faut penser à traiter séparément le cas où la file est vide :

```c
queue *empty_queue(void){
  queue *q = malloc(sizeof(queue));
  q->len = 0;
  q->left = NULL;
  q->right = NULL;
  return q;
}

void free_queue(queue *q){
  free_list(q->left);
  free(q);
}

datatype peek_left(queue *q){
  assert(q->len > 0);
  return q->left->data;
}

void push_right(queue *q, datatype data){
  node *n = new_node(data);
  if (q->right == NULL) {
    q->right = n;
    q->left = n;
  } else {
    q->right->next = n;
    q->right = n;
  }
  q->len++;
}
```

Pour `pop_left`, le cas particulier est pour une file ne contenant qu'un seul élément :

```c
datatype pop_left(queue *q){
  assert(q->len > 0);
  datatype res = q->left->data;
  if (q->len == 1){
    free(q->left);
    q->left = NULL;
    q->right = NULL;
  } else {
    node *tmp = q->left->next;
    free(q->left);
    q->left = tmp;
  }
  q->len--;
  return res;
}
```

Correction de l'exercice XXIII.8 page 6

Il faut faire très attention pour l'insertion :

```c
node *insert_it(node *u, datatype x){
  if (u == NULL || u->data >= x) { return cons(u, x); }
  node *current = u;
  while (current != NULL){
    if (current->next != NULL && current->next->data < x){
      current = current->next;
    } else {
      current->next = cons(current->next, x);
      break;
    }
  }
  return u;
}
```

Ensuite le tri lui-même ne pose pas vraiment de problème :

```c
node *insertion_sort_it(node *u){
  node *v = NULL;
  while (u != NULL){
    v = insert_it(v, u->data);
    u = u->next;
  }
  return v;
}
```

Correction de l'exercice XXIII.9 page 6

1. C'est assez naturel mais il est quand même facile de ne pas respecter parfaitement la spécification :

```c
node *split(node *u, int n){
  node *v = u;
  while (n - 1 > 0){
    assert(v != NULL);
    v = v->next;
    n--;
  }

  assert(v != NULL);
  node *tmp = v->next;
  v->next = NULL;
  return tmp;
}
```

2. Il y a peut-être plus simple, mais en tout cas c'est pénible :

```c
node *merge(node *u, node *v){
  if (u == NULL) { return v; }
  if (v == NULL) { return u; }
  node *res = NULL;
  if (u->data <= v->data){
    res = u;
    u = u->next;
  } else {
    res = v;
    v = v->next;
  }
  node *current = res;
  while (u != NULL || v != NULL){
    if (v == NULL || (u != NULL && u->data <= v->data)){
      current->next = u;
      u = u->next;
      current = current->next;
    } else {
      current->next = v;
      v = v->next;
      current = current->next;
    }
  }
  return res;
}
```

3. Cette fonction est assez simple. Attention, l'appel `split(u, n/2)` a un effet secondaire (que l'on exploite) sur `u`.

```c
node *merge_sort(node *u){
  int n = length(u);
  if (n <= 1) { return u; }
  node *second_half = merge_sort(split(u, n/2));
  node *first_half = merge_sort(u);
  return merge_it(first_half, second_half);
}
```