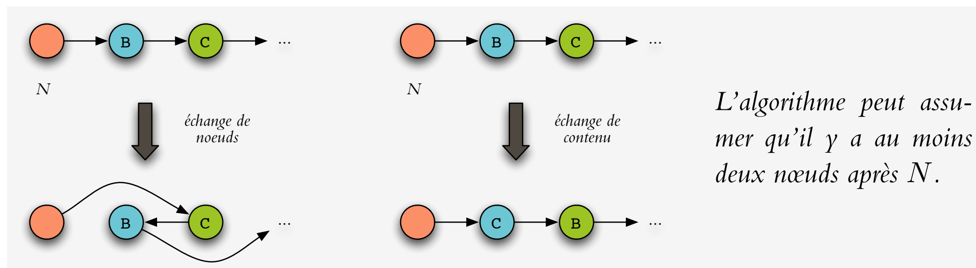
2 Exercices avec listes chaînées

Notation. Sauf si autrement spécifié, les exercices suivants représentent une liste comme un ensemble de nœuds : chaque nœud x (sauf le nœud terminal $x = \mathsf{null}$) contient les variables $x.\mathsf{next}$ (prochain élément) et $x.\mathsf{val}$ (contenu : élément stocké).

$\mathcal{X}$2.1 Échange d'éléments

▶ Montrer le code pour l'algorithme EXCHANGE(N) qui échange deux *nœuds* suivant N. ▶ Montrer le code pour l'algorithme EXCHANGEVAL(N) qui échange *le contenu* des deux nœuds suivant N.

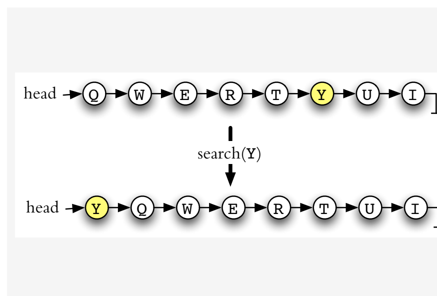


L'algorithme peut assumer qu'il y a au moins deux nœuds après N.

$\mathcal{X}$2.2 Recherche séquentielle

On veut un algorithme SEARCH(N, x) qui retourne le premier nœud M avec $M.\mathsf{val} = x$ sur la liste débutant avec nœud N. Si aucun nœud ne contient x, l'algorithme doit retourner null.

Recherche récursive. ▶ Donner une implantation récursive de SEARCH(N, x).

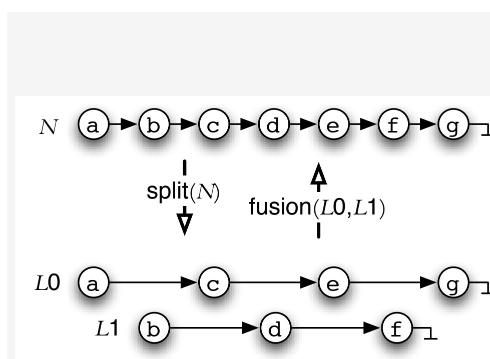


La heuristique MTF (*move-to-front*) déplace l'élément trouvé à la tête. Lors d'une recherche infructueuse, la liste ne change pas. La heuristique est utile quand on cherche des éléments avec des fréquences différentes car les éléments souvent recherchés se trouvent vers le début de la liste pendant une série d'appels.

Move-to-front. ▶ Montrer l'implantation de SEARCH(N, x) qui performe la recherche séquentielle pour une clé x sur la liste chaînée débutant avec N selon la heuristique MTF.

$\mathcal{X}$2.3 Tri de liste chaînée

Dans cet exercice, on développe le tri par fusion pour une liste chaînée. On a donc une liste chaînée des nœuds comparables, et on veut les trier.



D'abord, on veut une procédure split(N) qui sépare une liste simplement chaînée débutant avec le nœud N en deux listes contenant les nœuds originaux. Pour cela, on prend les nœuds pendant le parcours de la liste originale, et on les ajoute à la fin des sous-listes en altérnant entre les deux. (Ici, l'ordre des nœuds n'est pas important.)
On veut aussi l'opération inverse fusion qui fusionne deux listes déjà triées.

Dans les exercices suivants, on détruit la liste ou les listes à l'entrée pour placer les nœuds sur autres listes. Ne créez aucun nouveau nœud.

Séparation récursive. ▶ Implanter split par un algorithme *récursif*. L'algorithme doit retourner une paire de nœuds L_0, L_1 : pour la liste originale $x_0, x_1, \ldots, x_{\ell-1}$, L_0 est la tête de la liste $x_0, x_2, x_4, \ldots$ et L_1 est la tête de la liste des nœuds $x_1, x_3, x_5, \ldots$.

Séparation itérative. ▶ Donner une implantation sans récursion.

Indice: *Vous aurez besoin de variables pour stocker les deux têtes, et des curseurs pour stocker la fin des deux sous-listes pendant la construction.*

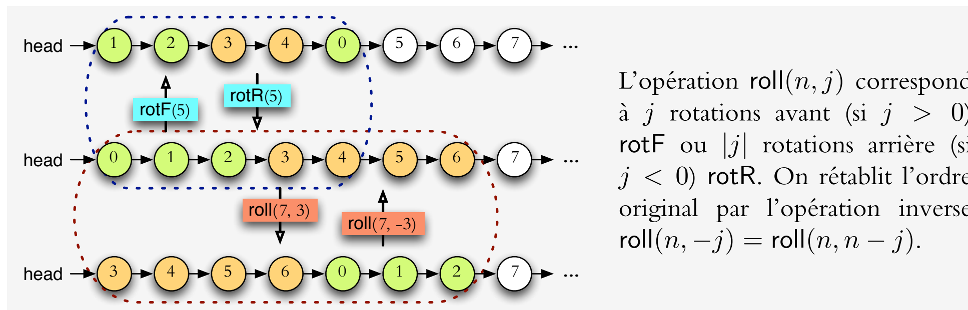
Fusion récursive. ▶ Donner une implantation récursive de l'opération fusion(L_0, L_1) qui prend deux listes triées à l'entrée, et retourne un troisième qui contient tous les nœuds des deux listes dans l'ordre croissant. Si L_0 ou L_1 est null, l'algorithme retourne simplement l'autre liste. (Ainsi on peut fusionner des listes de tailles différentes.)

Fusion récursive. ▶ Donner une implantation itérative de l'opération fusion(L_0, L_1).

$\mathcal{X}$2.4 Rotations

On veut une structure qui supporte des rotations d'éléments sur une liste. Soit $(x_0, x_1, \ldots, x_{\ell-1})$ l'ordre des nœuds sur la liste (x_0 est le premier nœud). Les opérations suivantes changent l'ordre des éléments $x_0, \ldots, x_{n-1}$ avec $n \leq \ell$. En une **rotation avant** (rotF), on avance les nœuds $x_1, \ldots, x_{n-1}$ vers la tête et on place x_0 après x_{n-1}. En une **rotation arrière** (rotR), les nœuds $x_0, \ldots, x_{n-2}$ reculent vers la queue et x_{n-1} se place à la tête. En un **décalage circulaire** (roll) on performe plusieurs rotations avant ou arrière.

Opération	Résultat
$\mathsf{rotF}(n)$	$\bigl(x_1, x_2, \ldots, x_{n-1}, x_0, \underbrace{x_n, x_{n+1}, \ldots, x_{\ell-1}}_{\text{ne change pas}}\bigr)$
$\mathsf{rotR}(n)$	$\bigl(x_{n-1}, x_0, x_1, \ldots, x_{n-2}, \underbrace{x_n, x_{n+1}, \ldots, x_{\ell-1}}_{\text{ne change pas}}\bigr)$
$\mathsf{roll}(n, j)$	$\bigl(x_{j \bmod n}, x_{(j \bmod n)+1}, \ldots, x_{n-1}, x_0, x_1, \ldots, x_{(j \bmod n)-1}, \underbrace{x_n, \ldots, x_{\ell-1}}_{\text{ne change pas}}\bigr)$



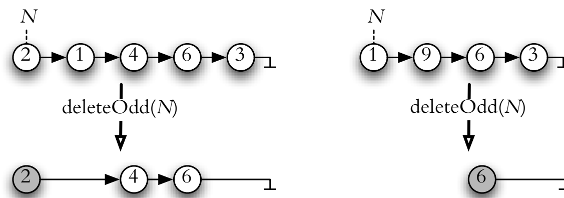
L'opération $\mathsf{roll}(n, j)$ correspond à j rotations avant (si $j > 0$) rotF ou $|j|$ rotations arrière (si $j < 0$) rotR. On rétablit l'ordre original par l'opération inverse $\mathsf{roll}(n, -j) = \mathsf{roll}(n, n-j)$.

► Montrez comment implanter les opérations $\mathsf{rotF}(H, n)$, $\mathsf{rotR}(H, n)$, et $\mathsf{roll}(H, n, j)$ sur une liste simplement chaînée. L'argument H dénote le début de la liste : on appelera p.e., $\mathsf{head} \leftarrow \mathsf{rotF}(\mathsf{head}, 10)$.

Indice: *Il n'est pas nécessaire de performer j rotations pour implanter* roll. *Identifiez plutôt les nœuds où* .next *doit changer.*

$\mathcal{X}$2.5 Pair-impair

On veut une procédure deleteOdd(N) qui supprime les nœuds avec clés impaires à partir de N et retourne la nouvelle tête de la liste. L'algorithme doit préserver l'ordre des nœuds qui restent. Montrez tous les détails de vos algorithmes (p.e., il ne suffit pas de dire «suppression après x», il faut montrer les affectations exactes).



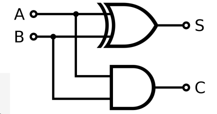
Exemples du fonctionnement de deleteOdd *: on retourne une référence au nœud ombré.*

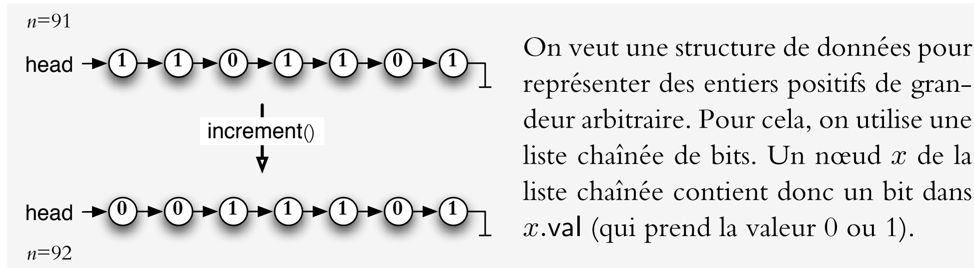
a. Solution itérative. ▶ Donnez une implantation *itérative* de deleteOdd.

b. Solution récursive. ▶ Donnez une implantatation *récursive* de deleteOdd.

c. Récursion terminale. ▶ Donnez une implantation avec récursion terminale.

$\mathcal{X}$2.6 Arithmétique binaire





On veut une structure de données pour représenter des entiers positifs de grandeur arbitraire. Pour cela, on utilise une liste chaînée de bits. Un nœud x de la liste chaînée contient donc un bit dans x.val (qui prend la valeur 0 ou 1).

La tête de la liste (head) est le nœud pour le bit de poids faible. On représente le nombre $n = 0$ par une liste comportant un seul nœud x dont $x.\mathsf{bit} = 0$. Un nombre $n > 0$ avec $2^{k-1} \le n < 2^k$ est représenté par une liste de longueur k : $n = \sum_{i=0}^{k-1}(x_i.\mathsf{bit}) \cdot 2^i$ où x_i est le i-ème nœud après la tête.

- ▶ Donnez une implantation de l'opération increment(head) qui incrémente (par un) le nombre représenté. L'algorithme doit mettre à jour la liste (au lieu de créer une autre liste). Analysez la croissance asymptotique du temps de calcul au pire cas en fonction de la valeur incrémentée n. Est-ce qu'on peut dire que votre algorithme prend un temps *linéaire* (dans la taille de l'argument n) ? Justifiez votre réponse.
- ▶ Démontrez que le temps amorti de increment est $O(1)$ dans votre implantation.

  Indice: *Ici, il faut montrer que le temps total $T(n)$ pour n appels de* increment() *(après lesquels la liste représente n) est borné par $T(n)/n = O(1)$. Dans d'autres mots, même si* increment() *ne prend pas toujours $O(1)$, compter jusqu'à n prend $O(n)$ temps. Examinez donc comment le contenu des nœuds change quand on compte jusqu'à n.*
- ▶ Donnez un algorithme add(A, B) qui calcule la représentation de $a + b$ à partir de leur représentations par liste. L'argument A (B) donne le premier nœud contenant le bit de poids faible pour la liste de a (b).
- ▶ Donnez une implantation de increment pour l'*encodage Fibonacci*. Dans cet encodage, une liste de longueur k représente le nombre $n = \sum_{i=0}^{k-1}(x_i.\mathsf{bit}) \cdot F(i+2)$, où $F(i)$ est le i-ème nombre Fibonacci. (Rappel : $F(0) = 0$, $F(1) = 1$.) Notez que la représentation Fibonacci n'est pas unique : par exemple, $19 = \overline{101001} = \overline{11111}$ car $19 = 13 + 5 + 1 = 8 + 5 + 3 + 2 + 1$.

  Indice: *La clé est d'utiliser la représentation de poids minimal : c'est celle qui minimise $\sum_i x_i.\mathsf{bit}$. On y arrive en remplaçeant chaque suite $\overline{011}$ par $\overline{100}$ (possible car $F(i) = F(i-2) + F(i-1)$). Examinez d'abord comment peut-on générer la représentation minimale en parcourant la liste : si on a $0, 1, 1$ (de la tête vers la queue), alors remplacer par $1, 0, 0$. Attention, le remplacement peut créer une suite $0, 1, 1$ voisine !*