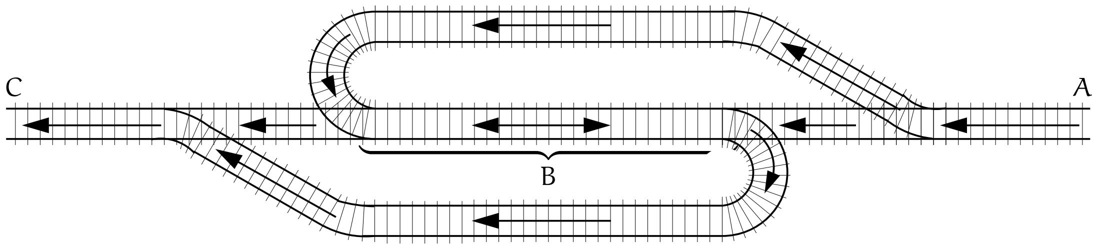
Permutation d'un train

Présentation du problème



On considère l'échangeur ferroviaire ABC dessiné ci-dessus et un train $w = [w_0, \ldots, w_{n-1}]$ de n wagons étiquetés par les entiers de $[\![0, n-1]\!]$ tel que w est une permutation de $[0, \ldots, n-1]$. Le problème consiste à faire passer les wagons à travers l'échangeur de sorte que le train w' sortant en C soit trié par ordre croissant. Pour réaliser le tri, on détache les wagons à leur arrivée en A, on les engage un à un sur la voie de manœuvre B soit par la gauche soit par la droite, on les transfère de B vers C via les issues gauche ou droite de B et on rattache les wagons en C. Naturellement,

– les wagons doivent respecter les sens de parcours autorisés sur les voies ;
– un wagon ne peut pas sauter par dessus un autre wagon ;
– il est supposé que B est suffisament longue pour contenir tous les wagons en même temps si besoin ;
– les wagons sont supposés symétriques donc un changement d'orientation d'un wagon est sans importance.

Exemple 1 : $n = 5$, $w = [2,\ 4,\ 0,\ 1,\ 3]$

manœuvre	C	B	A
début			2 4 0 1 3
tête(A) $\longrightarrow$ droite(B)		2	4 0 1 3
tête(A) $\longrightarrow$ gauche(B)		4 2	0 1 3
tête(A) $\longrightarrow$ gauche(B)		0 4 2	1 3
gauche(B) $\longrightarrow$ queue(C)	0	4 2	1 3
tête(A) $\longrightarrow$ gauche(B)	0	1 4 2	3
gauche(B) $\longrightarrow$ queue(C)	0 1	4 2	3
droite(B) $\longrightarrow$ queue(C)	0 1 2	4	3
tête(A) $\longrightarrow$ droite(B)	0 1 2	4 3	
droite(B) $\longrightarrow$ queue(C)	0 1 2 3	4	
gauche(B) $\longrightarrow$ queue(C)	0 1 2 3 4		

Exemple 2 : $n = 5$, $w = [4,\ 1,\ 2,\ 3,\ 0]$

manœuvre	C	B	A
début			4 1 2 3 0
tête(A) $\longrightarrow$ droite(B)		4	1 2 3 0
tête(A) $\longrightarrow$ gauche(B)		1 4	2 3 0
tête(A) $\longrightarrow$ droite(B)		1 4 2	3 0

Après ces trois manœuvres on s'apperçoit que si l'on engage le wagon 3 à gauche de B alors on va « coincer » le wagon 1 qui pourra pas quitter B avant que l'un des wagons 3 ou 4 l'ait fait, donc w' ne sera pas trié. De même, si l'on introduit 3 à droite de B, c'est alors 2 qui est coincé. En fait, quelle que soit la façon d'introduire les wagons $4, 1$ et 2 dans B, on aboutit à un blocage lors de l'introduction de 2 ou de 3. Le train w n'est pas triable sur l'échangeur ABC.

Ainsi il existe des trains qui sont triables sur ABC, et d'autres qui ne le sont pas. L'objet du TP est de programmer l'algorithme décrit ci-dessous qui dit si un train w donné est triable ou non, et le cas échéant qui indique les manœuvres à effectuer pour trier w.

1. *Si* B *contient à une de ses extrémités un wagon qui peut sortir vers* C*, faire sortir ce wagon.*
2. *Sinon, tester si l'introduction du wagon de tête de* A *à l'une des deux extrémités de* B *ne conduit pas à un blocage. Soit* i *le numéro du wagon de tête de* A *et* $i_1, \ldots, i_p$ *les numéros des wagons actuellement présents dans* B *ordonnés de gauche à droite :*
   – *l'introduction de* i *à gauche de* B *produit un blocage si* $i > i_1$ *et la suite* $(i_1, \ldots, i_p)$ *n'est pas décroissante ;*
   – *l'introduction de* i *à droite de* B *produit un blocage si* $i > i_p$ *et la suite* $(i_1, \ldots, i_p)$ *n'est pas croissante.*
3. *S'il n'y a qu'une extrémité de* B *pour laquelle l'introduction du wagon de tête de* A *ne conduit pas à un blocage, effectuer cette introduction.*
4. *Si les deux extrémités de* B *sont acceptables pour l'introduction du wagon de tête de* A*, introduire le wagon à l'une des extrémités et poursuivre le déroulement de l'algorithme. En cas d'impossibilité ultérieure, essayer l'introduction à l'autre extrémité.*

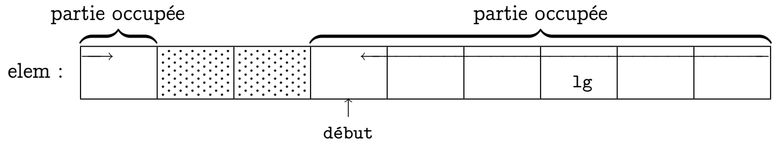
1. Listes à double entrée

On représentera l'état à un moment donné des voies A, B, C par un triplet de listes L_A, L_B, L_C contenant les numéros des wagons dans la voie correspondante, ordonnés de gauche à droite. Ces listes doivent pouvoir permettre de réaliser efficacement les opérations suivantes :

– examen de l'élément figurant en début ou en fin de liste ;
– insertion et extraction d'un élément en début ou en fin de liste ;
– parcours de la liste.

Les listes chaînées que fournit CAML en standard sont mal adaptées à ce problème car les opérations en fin de liste chaînée imposent de parcourir toute la liste et donc ont une complexité proportionnelle à la longueur de la liste considérée. On utilisera dans le TP une représentation des listes par des *vecteurs circulaires* c'est-à-dire des vecteurs dont les deux extrémités sont conceptuellement « reliées ». Une liste occupe un segment de ce vecteur défini par son indice de début et sa longueur.

```
type dbliste = {
  elem  : int vect;
  mutable début : int;
  mutable lg    : int
};;
```



Si L est une variable de type `dbliste` alors la longueur de L est `L.lg` et les éléments de L sont dans l'ordre : `L.elem.(L.début)`, `L.elem.((L.début + 1) mod n)`, ..., `L.elem.((L.début + L.lg − 1) mod n)` où n est la longueur maximale de L : $n =$ `vect_length(L.elem)`. Dans la déclaration ci-dessus, les champs `début` et `lg` sont déclarés *mutables* de façon à pouvoir être modifiés. On écrira : `L.début <-` *qqch* ou `L.lg <-` *qqch* pour modifier les valeurs de ces champs.

Saisir la déclaration du type `dbliste` ci-dessus et écrire les fonctions CAML suivantes :

`premier :`	`dbliste -> int`	retourne le premier élément de la liste donnée, sans l'extraire.
`dernier :`	`dbliste -> int`	retourne le dernier élément de la liste donnée, sans l'extraire.
`insère_début :`	`dbliste -> int -> unit`	insère l'entier donné au début de la liste donnée.
`insère_fin :`	`dbliste -> int -> unit`	insère l'entier donné à la fin de la liste donnée.
`extrait_début :`	`dbliste -> int`	extrait le premier élément de la liste donnée et le retourne en résultat.
`extrait_fin :`	`dbliste -> int`	extrait le dernier élément de la liste donnée et le retourne en résultat.
`croissant :`	`dbliste -> bool`	dit si la liste donnée est croissante.
`décroissant :`	`dbliste -> bool`	dit si la liste donnée est décroissante.

Ces fonctions devront déclencher des erreurs appropriées en cas d'opération impossible (consultation ou extraction d'un élément dans une liste vide, insertion dans une liste pleine). Une liste vide sera considérée comme croissante et décroissante.

2. Possibilité de trier un train

Programmer l'algorithme de test de « triabilité » d'un train. On écrira une fonction récursive

```
recherche :  dbliste -> dbliste -> dbliste -> bool
```

qui prend en entrée les listes L_A, L_B, L_C représentant l'occupation des voies A, B, C à un instant donné, et qui retourne en résultat un booléen disant si l'on peut faire sortir vers C tous les wagons restant dans $L_A \cup L_B$ par ordre croissant. Cette fonction détermine s'il existe un mouvement de wagon non bloquant selon l'algorithme donné en introduction, effectue le cas échéant ce mouvement (c'est-à-dire modifie en conséquence les listes L_A, L_B, L_C) puis se rappelle elle-même pour poursuivre le déroulement de l'algorithme. Dans le cas *4* où l'on doit éventuellement essayer les deux possibilités d'insertion dans B, on procèdera à une copie préalable des listes L_A, L_B, L_C avant d'essayer la première possibilité (car l'essai va modifier ces listes et on doit repartir de la situation en vigueur au moment du choix). On utilisera la fonction de copie suivante :

```
let copie l = { elem  = copy_vect l.elem; début = l.début; lg = l.lg };;
```

On utilisera les fonctions suivantes pour tester `recherche` :

```
(* retourne une permutation aléatoire de [0,n-1] *)
let random_perm n =
  let l = { elem = make_vect n 0; début = 0; lg = n } in
  for i=1 to n-1 do l.elem.(i) <- i done;
  for c = 1 to 2*n do
    let i = random__int(n) in
    let j = random__int(n) in
    let x = l.elem.(i) in l.elem.(i) <- l.elem.(j); l.elem.(j) <- x
  done;
  l
;;

(* dit si le train présent dans l est triable *)
let triable(l) =
  let n  = l.lg    in
  let la = copie l in
  let lb = {elem = make_vect n 0; début = 0; lg = 0} in
  let lc = {elem = make_vect n 0; début = 0; lg = 0} in
  recherche la lb lc
;;
```

3. Manœuvres à effectuer pour trier un train

On introduit des types de données suivants pour décrire les manœuvres et les listes de manœuvres à effectuer pour trier un train :

```
type manoeuvre =
  | IG    (* tête(A)   -> gauche(B) *)
  | ID    (* tête(A)   -> droite(B) *)
  | XG    (* gauche(B) -> queue(C)  *)
  | XD    (* droite(B) -> queue(C)  *)
and résultat =
  | Possible of manoeuvre list (* manoeuvres pour trier le train *)
  | Impossible                 (* train non triable             *)
;;
```

Modifier la fonction `recherche` de la question précédente pour qu'elle retourne un résultat de type `résultat` indiquant si un train est triable et dans ce cas la liste des manœuvres à effectuer pour le trier.

Remarque : voir KNUTH, *The art of computer programming vol. 1*, exercice 2.2.1-13 pour l'origine de ce problème. KNUTH indique qu'il n'y a que quatre trains de cinq wagons non triables : [4, 1, 2, 3, 0], [3, 1, 2, 4, 0], [1, 4, 2, 3, 0] et [1, 3, 2, 4, 0]. On pourra, si l'on a le temps, vérifier ce fait.