- adjlis (l, p, v) <==>

```
fonction adjlis (l InOut : ListeChainéeTableau, p : entier, v : Elément)
        début
        recherche d'une place libre pL
        l[pL].val ← v
        l[pL].suc ← l[p].suc
        l[p].suc ← pL
        fin
```

- suplis (l, p) <==>

```
fonction suplis (l InOut : ListeChainéeTableau , p : entier)
        début
        ancP ← 0
        tantque l[ancP].suc ≠ p faire
                ancP ← l[ancP].suc
        ftantque
        l[ancP].suc ← l[p] .suc      (* ancP est la place précédant la place p à supprimer ; c'est l'indice
                                        de la tête si l'élément à supprimer était le 1er de la liste *)
        restitution de la place libre p
        fin
```

- chglis (l,p,v) <==>

```
fonction chglis (l InOut : ListeChainéeTableau , p : entier, v : Elément)
        début
        l[p].val ← v
        fin
```

Gestion de l'espace libre sans récupération des places libérées

On se positionne en début de tableau et on consomme les places les unes après les autres sans s'occuper de restituer les places libérées lors des suppressions. Un marqueur noté pLibre dans la suite, permet de désigner la première place non encore utilisée dans le tableau . Au fur et à mesure des adjonctions, cette place avance dans le tableau. Elle peut être mémorisée dans le champ suc du tableau à l'indice -1. Le tableau sera alors défini sur l'intervalle [-1..bs] .

Exemple :

	-1	0	1	2	3	4	5	**6**					
val			y	b	b		a						
suc	**6**	3	0	1	5		2						

↑
pLibre

Voyons comment se modifient les expressions des fonctions logiques :

initialisation de l'espace libre :

```
l[-1].suc ← 1
```

recherche d'une place pl libre :

```
pl ← l[-1].suc
l[-1].suc ← l[-1].suc + 1
```

Remarque :
Lorsque le marqueur de place libre arrive en fin de tableau, on peut faire une opération de "ramasse-miettes". Cette opération consiste à concentrer toutes les valeurs en début de tableau afin de récupérer les places libres dispersées dans le tableau.

Gestion de l'espace libre avec marquage et récupération des places libres

Cette solution permet de profiter des suppressions d'éléments pour récupérer de la place et la réutiliser. Elle consiste à marquer les places libres en y mettant une valeur particulière, par exemple -1 dans le champ *suc*.

Lors de l'initialisation de la liste à "vide", on marque à -1 le champ suc de toutes les places. Chaque fois qu'on supprime un élément, on indique que la place qu'il occupait est libre en mettant -1 dans le champ suc. Chaque fois qu'on veut ajouter un élément, on cherche dans le tableau une case dont le champ suc est marqué à -1.

Exemple :

	0	1	2	3	4	5	6	7						indices
l		y	b	b		a								val
	3	0	1	5	**-1**	2	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	suc

tête

Les algorithmes correspondant à cette gestion sont donnés en annexe 3.

Variantes :

- On peut aussi marquer les places libres en mettant une valeur "bidon" dans le champ val. Mais il n'est pas toujours possible de trouver une valeur "bidon".
- On peut ne pas marquer les places libres au fur et à mesure des libérations mais seulement en cas de problème (on consomme en suivant, tant qu'on peut). Cela se fait en marquant alors les places occupées, par un parcours de la liste. Les places libres sont les autres. Ennui: il faut un tableau supplémentaire, de booléens: lOccup.

Gestion de l'espace libre à l'aide d'une liste ("liste libre")

Le tableau l contient deux listes :

- la liste sur laquelle on travaille (liste de travail) ;
- la liste des places inoccupées (liste libre).

Les créations, suppressions et adjonctions mettent les deux listes à jour. Lorsque l'on veut ajouter un élément dans la liste de travail, on prend l'élément de la tête de la liste libre. Lorsqu'on supprime un élément dans la liste de travail, on ajoute sa place dans la liste libre (en tête).

Souvent, on représente la tête de la liste libre par le champ suc de l'élément d'indice -1 du tableau l. Nous retiendrons cette hypothèse.

Exemple :

	-1	0	1	2	3	**4**	5	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
val			y	b	b		a								
suc	**4**	3	0	1	5	**6**	2	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**0**

↑

tête de la liste libre

Remarque :

Ceci fait partie du problème plus général de la gestion simultanée de plusieurs listes.

Voici comment compléter les expressions des fonctions logiques :

Déclarations

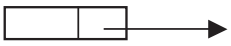
l : tableau <val : Elément, suc : entier > [-1..bs] où Elément est le type des éléments de la liste.

```
Initialisation de l'espace libre
        l [-1].suc ←1
        pour i de 1 à bs-1 faire
                l[i].suc ← i+1
        fpour
        l [bs].suc ← 0

Recherche d'une place libre pl  (en tête)   (on suppose qu'il y en a)
        pl ← l [-1].suc
        l [-1].suc ← l [pl].suc

Restitution d'une place libre pl  (en tête)
        l [pl].suc ← l [-1].suc
        l [-1].suc ← pl
```

3.3 Représentation chaînée à l'aide de pointeurs

Un **pointeur** est une variable dont la valeur est l'adresse d'une autre variable. On dit que le pointeur "pointe vers" l'autre variable. Les pointeurs permettent de gérer la mémoire de manière dynamique, c'est-à-dire lors de l'exécution du programme: on ne crée des variables que lorsque l'on en a besoin. De la même manière, lorsque l'on n'a plus besoin d'une variable, on peut récupérer la place qu'elle occupait.

Comment représenter une liste chaînée par des pointeurs en ne réservant en mémoire que la place nécessaire ? Une liste chaînée est, comme nous l'avons vu, une suite de couples (valeur, suivant) que l'on peut représenter par le schéma :



Le champ suc contient l'adresse de l'élément suivant c'est-à-dire un élément de type pointeur.

Pour matérialiser la tête de la liste, il faut un pointeur vers le premier élément. La connaissance de ce pointeur donne complètement accès à la liste ; pour cette raison, on confond la liste et son pointeur de tête.

Cette représentation est très utilisée et sera détaillée dans le langage C.

3.4 Exercice

Ecrire l'algorithme de programmation correspondant à l'algorithme logique de l'exercice 4 du paragraphe 1.4 en représentant les listes de manière chaînée dans un tableau et avec une gestion de l'espace libre sans récupération des places libérées.

La solution classique et généralement conseillée consiste à écrire simplement les fonctions traductions des fonctions logiques utilisées sur les listes. La solution demandée dans cet exercice ne devra pas utiliser de fonctions pour les traductions afin de permettre de réaliser facilement des optimisations.

Rappel de l'énoncé :

Soit une suite de valeurs entières (toutes comprises entre 0 et 1000). Construire une liste *lEntier*, triée par ordre croissant, contenant les entiers figurant un nombre impair de fois dans la suite des données.
(Principe : chaque donnée est cherchée dans la liste partiellement construite ; si elle appartient déjà à la liste, on la supprime, sinon on la rajoute.)

Algorithme logique (rappel) :

```
fonction lEntierCréer ( ) : Liste (entier)
début (*1*)
    lEntier ← lisvide( )
    nbValeur ← lire( )
    pour i de 1 à nbValeur faire (*2*)
            valeur ← lire( )
            place ← tête (lentier)
```

```
                si finliste (lEntier, place) alors (*8a*)  (*la liste est vide*)
                        adjqlis (lEntier, valeur)
                sinon (*8s*)
                        placePrécédente ← place  (* initialisation artificielle *)
                        (*recherche de la valeur*)
                        fini ← faux
                        tantque(*3*) non finliste (lEntier, place) et non fini faire
                                élément ← val (lEntier, place)
                                si  valeur ≤ élément alors (*4a*)
                                        fini ← vrai
                                sinon (*4s*)
                                        placePrécédente ← place
                                        place ← suc (lEntier, place)
                                fsi (*4*)
                        fintantque (*3*)
                        si valeur = élément alors (*5a*)
                                (*valeur appartient déjà à la liste*)
                                suplis (lEntier, place)
                        sinon(*5s*)
                                 (*valeur n'appartient pas à la liste*)
                                si place = placePrécédente alors(*7a*)
                                        (*valeur est inférieur à toutes les valeurs*)
                                        adjtlis (lEntier, valeur)
                                sinon(*7s*)
                                        adjlis (lEntier, placePrécédente, valeur)
                                fsi(*7*)
                        fsi(*5*)
                fsi (*8*)
        fpour(*2*)
        retourne lEntier
    fin(*1*)
```

Lexique

lEntier : Liste (entier), liste des entiers par ordre croissant
nbValeur : entier, nombre de valeurs à traiter
i : entier, indice d'itération
valeur : entier, suite des données
place : Place, place courante
placePrécédente : Place, place précédent la place *place*
fini : booléen, vrai lorsqu'on a trouvé l'entier ou lorsque la valeur courante est supérieure à la valeur cherchée.
élément : entier, valeur courante dans la liste

Algorithme de programmation

```
fonction lentierCréer ( ) : LentierChaînéeTableau
    début(*1*)
    lentier[0].suc ← 0
    lentier[-1].suc ← 1
    nbvaleur ←  lire( )
    pour i de 1 à nbValeur faire (*2*)
        valeur ← lire( )
        place ← lEntier[0].suc
        si place = 0 alors (*8a*)
                placeLibre ← lEntier[-1].suc
                lEntier[-1].suc ←  lEntier[-1].suc + 1
                lEntier [placeLibre ].val ← valeur
                lEntier [placeLibre ].suc ← 0
                lEntier[0].suc ← placeLibre
        sinon (*8s*)
                placePrécédente ← place
                fini ← faux
                (*recherche de la valeur*)
                tantque(*3*) place ≠ 0 et non fini faire
                        élément ←  lEntier[place].val
                        si valeur ≤ élément alors(*4a*)
                                fini ← vrai
                        sinon(*4s*)
```

```
                                placePrécédente ← place
                                place ← lEntier[place].suc
                        fsi(*4*)
                ftantque(*3*)
                si valeur = élément alors(*5a*)
                        (*valeur appartient déjà à la liste*)
                        si place = placePrécédente alors(*6a*)
                                (*c'est le 1er élément*)
                                lEntier[0].suc ← lEntier[place].suc
                        sinon(*6s*)
                                lEntier[placePrécédente].suc ← lEntier[place].suc
                                (* en appliquant la traduction systématique, on écrirait :
                                placePrécédente ← 0
                                tantque(*20*) lEntier[placePrécédente].suc ≠ place faire
                                        placePrécédente ← lEntier[placePrécédente].suc
                                ftantque(*20*)
                                lEntier[placePrécédente].suc ← lEntier[place].suc
                                *)
                        fsi(*6*)
                sinon(*5s*) (*valeur n'appartient pas à la liste*)
                        placeLibre ← lEntier[-1].suc
                        lEntier[-1].suc ← lEntier[-1].suc+1
                        si place = placePrécédente alors(*7a*)
                                (*valeur < à toutes les valeurs*)
                                lEntier[placeLibre].val ← valeur
                                lEntier[placeLibre].suc ← lEntier[0].suc
                                lEntier[0].suc ← placeLibre
                        sinon(*7s*)
                                lEntier[placeLibre].val ← valeur
                                lEntier[placeLibre].suc ← lEntier[placePrécédente].suc
                                (*ou lEntier[placeLibre].suc ← place*)
                                lEntier[placePrécédente].suc ← placeLibre
                        fsi(*7*)
                fsi(*5*)
        fsi(*8*)
    fpour(*2*)
    retourne lEntier
    fin(*1*)
Lexique
    LEntierChaînéeTableau = tableau < val : entier, suc : entier > [-1...bs]
    lEntier : LEntierChaînéeTableau, tableau des valeurs et des successeurs
    nbValeur : entier, nombre de valeurs à traiter
    i : entier, indice d'itération
    bs : entier, nombre maximum d'éléments de la liste lEntier
    valeur : entier, suite des données
    place : entier, place courante dans lEntier
    placePrécédente : entier, place précédant place dans lEntier
    fini : booléen, vrai lorsqu'on a trouvé l'entier ou lorsque la valeur courante est supérieure à la valeur cherchée
    élément : entier, valeur courante
    placeLibre : entier, place libre sélectionnée pour la valeur courante
```

4 Les piles et les files

Pour beaucoup d'applications, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités.

4.1 Les piles

Une **pile** est une liste dans laquelle toutes les adjonctions et toutes les suppressions se font à une seule extrémité appelée *sommet*. Ainsi, le seul élément qu'on puisse supprimer est le plus récemment entré. Une pile a une structure « LIFO » pour « Last In, First Out » c'est-à-dire « dernier entré premier sorti ». Une bonne image pour se représenter une pile est une pile d'assiettes : c'est en haut de la pile qu'il faut prendre ou mettre une assiette !