Structures de Données – Rapport de TP

Axel Delsol, Pierre-Loup Pissavy

Mai 2014

Table des matières

Exercice 1 **2**
1.1 Tableaux Multiples . . . 2
1.1.1 Principe . . . 2
1.1.2 Suppression . . . 2
1.1.3 Insertion . . . 3
1.1.4 Code source . . . 3
1.2 Tableau unique . . . 7
1.2.1 Principe . . . 7
1.2.2 Suppression . . . 8
1.2.3 Insertion . . . 8
1.2.4 Code source . . . 8

Exercice 2 **13**
2.1 Allocation-Libération . . . 13
2.1.1 Allocation . . . 13
2.1.2 Libération . . . 13
2.2 Compacte . . . 13
2.3 Densifier . . . 14
2.4 Code source . . . 14

Exercices 3 – 5 **17**
3.1 Principe . . . 17
3.2 Allocation . . . 17
3.3 Libération . . . 18
3.4 Code source . . . 18

Exercice 6 **20**
6.1 Principe . . . 20
6.2 Code source . . . 20

Exercice 1

1.1 Tableaux Multiples

1.1.1 Principe

Pour implémenter une liste avec des tableaux multiples, on va utiliser 3 tableaux et quelques informations supplémentaires :

- Un tableau contenant les données à stocker (`tDonnees`)
- Un tableau contenant les indices suivant d'un élément (`tSuivant`)
- Un tableau contenant les indices précédent d'un élément (`tPrecedent`)
- Un entier indiquant la capacité maximale de la liste car on utilise des tableaux statiques pour l'implémentation (`MAX`)
- Un entier indiquant le nombre d'élément que contient la liste (`taille`)
- Un indice de début de la liste même si le début de la liste se caractérise par le fait que le précédent du début de la liste est l'indice -1 (`debut`)
- Un indice de fin de liste même si la fin de la liste se caractérise par le fait que le suivant de fin est l'indice -1 (`fin`)

Lorsqu'on se trouve à une case `i` de `tDonnees`, l'élément suivant se trouve à l'indice contenu dans le tableau `tSuivant` de la case `i`. Par exemple : si l'élément 3 se trouve à une case `i` de `tDonnees`. Si on veut connaitre l'indice de l'élément 4, il suffit de récupérer la valeur `tSuivant[i]`. Ainsi, on accède à l'élément 4 depuis l'élément 3 avec : `tDonnees[tSuivant[i]]`. Le fonctionnement est le même pour précédent.

1.1.2 Suppression

La suppression se fait de manière à avoir une liste compacte en permanence. Elle se base sur les instructions suivantes si la liste n'est pas vide (on renvoie un message d'erreur sinon) :

1. On se place à la position du tableau de l'élément qu'on veut supprimer (avec un parcours classique de liste)
2. On regarde l'élément le plus à droite dans le tableau (il se trouve dont à l'indice `taille`-1)
3. On échange les informations entre l'élément le plus à droite et l'élément qu'on veut supprimer
4. On remet à jour les indices suivants et précédents de tous les éléments concernés (suivant-précédent de l'élément supprimé et ceux de l'élément le plus à droite dans le tableau)
5. Décrémenter `taille`.

Il y a cependant quelques points auxquels il faut penser. Tout d'abord, si on supprime l'élément le plus à droite, faire un échange ne marchera pas, il faut uniquement changer les pointeurs. De plus, si l'élément le plus à droite était le début ou la fin, il faut penser à changer les attributs `debut`/`fin`.

Ainsi, dès qu'on supprime un élément, on ne crée jamais de trou dans le tableau car on le "rebouche" dès qu'il est créé donc il est toujours compact.

1.1.3 Insertion

Du fait que la liste soit compacte grâce à la méthode de suppression, on sait exactement où on doit insérer notre prochain élément. Il suffit donc d'appliquer les instructions suivantes si la liste n'est pas pleine (on envoie un message d'erreur sinon) :

1. Récupérer l'indice de l'élément précédent la position de l'insertion (avec un parcours classique de liste)
2. Placer la donnée dans la case `taille` du tableau de données
3. Mettre à jour les tableaux `tSuivant`/`tPrecedent` pour les éléments précédent/suivant du nouvel élément inséré
4. Incrémenter `taille`

On remarque que si l'on insère au début ou à la fin, il faut aussi mettre à jour les indices `debut`/`fin` de la structure.

1.1.4 Code source

```
1  class ListeTabM implements Liste {
2    protected int MAX;
3    protected int taille;
4    protected Object[] tDonnees; // Tableau contenant les données
5    protected int[] tSuivant; // Contient les indices suivants
6    protected int[] tPrecedent;
7    protected int debut; // Indice du premier élément
8    protected int fin;
9
10   public ListeTabM (int max) {
11     this.MAX = max;
12     this.taille = 0;
13     this.tDonnees = new Object[max];
14     this.tSuivant = new int[max];
15     this.tPrecedent = new int[max];
16     this.debut = 0;
17     this.fin = 0;
18   }
19
20   public boolean estVide() {
21     return this.taille == 0;
22   }
23
24   public boolean estPleine(){
25     return this.taille == this.MAX;
26   }
27
28   public int taille() {
29     return this.taille;
30   }
31
32   public void insererDebut(Object o) {
33     if (this.estPleine()) {
34      System.err.println("La liste est pleine");
35     }else {
36       tDonnees[this.taille] = o;
37       tPrecedent[this.taille] = -1;
38       if (this.estVide()) {
```

```
39          tSuivant[this.taille] = -1;
40          this.fin = this.taille;
41        }else {
42          tSuivant[this.taille] = this.debut;
43          tPrecedent[this.debut] = this.taille;
44        }
45        this.debut = this.taille;
46        this.taille = this.taille +1;
47      }
48    }
49
50    public void insererFin(Object o) {
51      if (this.estPleine()) {
52       System.err.println("La liste est pleine");
53      }else {
54        tDonnees[this.taille] = o;
55        tSuivant[this.taille] = -1;
56        if (this.estVide()) {
57          tSuivant[this.taille] = -1;
58        }else {
59          tPrecedent[this.taille] = this.fin;
60          tSuivant[this.fin] = this.taille;
61        }
62        this.fin = this.taille;
63        this.taille = this.taille +1;
64      }
65    }
66
67    public void inserer(Object o, int i ) {
68      if (this.estPleine()) {
69       System.err.println("La liste est pleine");
70      }else {
71        int cpt = 1;
72        int courant = this.debut;
73        while (cpt != i-1) {
74          courant = tSuivant[courant];
75          cpt = cpt +1;
76        }
77        tDonnees[this.taille] = o;
78        tSuivant[this.taille] = tSuivant[courant];
79        tPrecedent[tSuivant[courant]] = this.taille;
80        tSuivant[courant] = this.taille;
81        tPrecedent[this.taille] = courant;
82        this.taille = this.taille +1;
83      }
84    }
85
86    public void supprimerDebut() {
87      if (this.taille -1 == debut) {
88        tPrecedent[tSuivant[this.debut]] = -1;
89        debut = tSuivant[this.debut];
90      }else{
91        int tmp = tSuivant[debut];
92        tPrecedent[tSuivant[this.debut]] = -1;
93        tDonnees[debut] = tDonnees[taille-1];
94        tSuivant[debut] = tSuivant[taille-1];
```

```
95        tPrecedent[debut] = tPrecedent[taille-1];
96        tSuivant[tPrecedent[taille-1]] = debut;
97        tPrecedent[tSuivant[taille-1]] = debut;
98        debut = tmp;
99      }
100     this.taille = this.taille-1;
101   }
102
103   public void supprimerFin() {
104     if (this.taille-1 == fin) {
105       tSuivant[tPrecedent[this.fin]] = -1;
106       fin = tPrecedent[this.fin];
107     }else{
108       int tmp = tPrecedent[fin];
109       tSuivant[tPrecedent[this.fin]] = -1;
110       tDonnees[fin] = tDonnees[taille-1];
111       tSuivant[fin] = tSuivant[taille-1];
112       tPrecedent[fin] = tPrecedent[taille-1];
113       tSuivant[tPrecedent[taille-1]] = fin;
114       tPrecedent[tSuivant[taille-1]] = fin;
115       fin = tmp;
116     }
117     this.taille = this.taille - 1;
118   }
119
120
121
122   public void supprimer(int i) {
123     if (this.estVide()) {
124       System.err.println("La liste est vide");
125     }else {
126       int cpt = 1;
127       int courant = this.debut;
128       while (cpt != i) {
129         courant = tSuivant[courant];
130         cpt = cpt + 1;
131       }
132       if (courant == fin) {
133         this.supprimerFin();
134       }else if (courant == debut){
135         this.supprimerDebut();
136       }else{
137         if (courant == taille-1) {
138           tSuivant[tPrecedent[courant]] = tSuivant[courant];
139           tPrecedent[tSuivant[courant]] = tPrecedent[courant];
140         }else{
141           tSuivant[tPrecedent[courant]] = tSuivant[courant];
142           tPrecedent[tSuivant[courant]] = tPrecedent[courant];
143           tDonnees[courant] = tDonnees[this.taille-1];
144           tSuivant[courant] = tSuivant[this.taille-1];
145           tPrecedent[courant] = tPrecedent[this.taille-1];
146           tSuivant[tPrecedent[taille-1]] = courant;
147             tPrecedent[tSuivant[taille-1]] = courant;
148         }
149         if (this.taille-1 == this.fin) {
150           this.fin = courant;
```

```
151         }else if (this.taille -1 == this.debut) {
152           this.debut = courant;
153         }
154         this.taille = this.taille -1;
155       }
156     }
157   }
158
159   public Object tete() {
160     return tDonnees[debut];
161   }
162
163   public Object element(int i) {
164     int cpt = 1;
165     int courant = this.debut;
166     while (cpt != i) {
167       courant = tSuivant[courant];
168       cpt = cpt + 1;
169     }
170     return tDonnees[courant];
171   }
172
173   public Object suivant(int i) {
174     int cpt = 1;
175     int courant = this.debut;
176     while (cpt != i+1) {
177       courant = tSuivant[courant];
178       cpt = cpt + 1;
179     }
180     return tDonnees[courant];
181   }
182
183   public Object precedent(int i) {
184     int cpt = 1;
185     int courant = this.debut;
186     while (cpt != i-1) {
187       courant = tSuivant[courant];
188       cpt = cpt + 1;
189     }
190     return tDonnees[courant];
191   }
192
193   public void afficher(){
194     System.out.print("Données : ");
195     for (int k = 0; k < tDonnees.length ;k++ ) {
196       System.out.print(tDonnees[k]+" ");
197     }
198     System.out.println();
199
200     System.out.print("Suivant : ");
201     for (int k = 0; k < tSuivant.length ;k++ ) {
202       System.out.print(tSuivant[k]+" ");
203     }
204     System.out.println();
205
206     System.out.print("Precedent : ");
```

```
207       for (int k = 0; k < tPrecedent.length ;k++ ) {
208         System.out.print(tPrecedent[k]+" ");
209       }
210       System.out.println();
211     }
212
213     @Override
214     public String toString() {
215       StringBuilder s = new StringBuilder();
216       int courant = this.debut;
217       while (courant != -1) {
218         s.append(tDonnees[courant]+" ");
219         courant = tSuivant[courant];
220       }
221       s.append("\n");
222       return s.toString();
223     }
224
225     public Object[] elements(){
226       Object[] res = new Object[this.taille];
227       int i = 0;
228       int courant = debut;
229       while (courant != -1) {
230         res[i] = tDonnees[courant];
231         courant = tSuivant[courant];
232         i = i+1;
233       }
234       return res;
235     }
236
237 }
```

1.2 Tableau unique

1.2.1 Principe

Pour implémenter une liste avec un unique tableau , on va utiliser 1 tableau et quelques informations supplémentaires :

- Un tableau contenant les données à stocker, l'indice suivant et précédent de la liste.
- Un entier indiquant la capacité maximale de la liste car on utilise des tableaux statiques pour l'implémentation (`MAX`)
- Un entier indiquant le nombre d'élément que contient la liste (`taille`)
- Un indice de début de la liste même si le début de la liste se caractérise par le fait que le précédent du début de la liste est l'indice -1 (`debut`)
- Un indice de fin de liste même si la fin de la liste se caractérise par le fait que le suivant de fin est l'indice -1 (`fin`)

Le fonctionnement de la liste est le suivant : si on se trouve à une case `i` multiple de 3 alors : la case `i+1` contient l'indice de l'élément précédent dans le tableau et la case `i+2` contient l'indice de l'élément suivant dans le tableau. Ceci crée donc des blocs de 3 cases dans le tableau Pour obtenir cela, le tableau a donc une taille réelle de `3*MAX` et toutes les données sont situées sur des cases multiples de 3. La gestion des suivants/précédents/debut/fin est la même que la méthode avec des tableaux multiples, la seule différence est la localisation des suivants/précédents pour un élément donné.

On peut remarquer que dans ce cas là, on ne stocke qu'une seule donnée. Cependant, il suffit d'augmenter la longueur d'un bloc et de donner un indice pour les autres données. Ainsi, on peut généraliser cette méthode pour stocker des objets plus complexes.

1.2.2 Suppression

La suppression se base sur le même principe que pour les tableaux multiples, le même algorithme est utilisé. Cependant, l'accès aux données se fait par décalage par rapport à une position donnée.

1.2.3 Insertion

L'insertion se base sur le même principe que pour les tableaux multiples, le même algorithme est utilisé. Cependant, l'accès aux données se fait par décalage par rapport à une position donnée.

1.2.4 Code source

```
1  class ListeTab implements Liste{
2    protected int debut;
3    protected int fin;
4    protected int taille;
5    protected int MAX;
6    protected Object[] liste; // Organisé sous forme donnees-precedent-suivant-donnes ...
7
8    public ListeTab (int max) {
9      this.debut = 0;
10     this.fin = 0;
11     this.taille = 0;
12     this.MAX = max;
13     this.liste = new Object[3*max];
14   }
15
16   public boolean estVide() {
17     return this.taille == 0;
18   }
19
20   public boolean estPleine() {
21     return this.taille ==  this.MAX;
22   }
23
24   public int taille(){
25     return this.taille;
26   }
27
28   public void insererDebut(Object o) {
29     if (this.estPleine()) {
30       System.err.println("La liste est pleine");
31     }else {
32       int position = 3*this.taille;
33       this.liste[position] = o; // On ajoute l'élément à la case libre
34       this.liste[position+1] = -1; // Il n'a pas de précédent
35       if(this.estVide()){
```

```
36          this.liste[position+2] = -1;
37          this.fin = position;
38        }else{
39          this.liste[position+2] = this.debut; // Son suivant est l'ancien début
40          this.liste[this.debut+1] = position; // Son précédent devient position
41        }
42        this.debut = position; // Le début est maintenant position
43        this.taille = this.taille + 1; // On incrémente la taille
44      }
45    }
46
47    public void insererFin(Object o) {
48      if (this.estPleine()) {
49        System.err.println("La liste est pleine");
50      }else {
51        int position = 3*this.taille;
52        this.liste[position] = o;
53        if(this.estVide()){
54          this.liste[position+2] = -1;
55          this.debut = position;
56        }else{
57          this.liste[position+1] = fin;
58          this.liste[position+2] = -1;
59          this.liste[this.fin+2] = position;
60        }
61        this.fin = position;
62        this.taille = this.taille + 1;
63      }
64    }
65
66    public void inserer(Object o, int i) {
67      if (this.estPleine()) {
68        System.err.println("La liste est pleine");
69      }else {
70        int cpt = 1;
71        int courant = this.debut;
72        int position = 3*this.taille;
73        while (cpt != i-1) {
74          courant = ((Integer) this.liste[courant+2]).intValue();
75         cpt = cpt+1;
76        }
77        int suivantCourant = ((Integer) this.liste[courant+2]).intValue(); // Indice du
      suivant de courant
78        this.liste[position] = o;
79        this.liste[position+1] = courant;
80        this.liste[position+2] = suivantCourant;
81        this.liste[suivantCourant+1] = position;
82        this.liste[courant+2] = position;
83        this.taille = this.taille +1;
84      }
85    }
86
87    public void supprimerDebut() {
88      if (this.debut == 3*(this.taille-1)) {
89        int suivantDebut = ((Integer) this.liste[this.debut+2]).intValue();
90        this.liste[suivantDebut+1] = -1;
```

```
91        this.debut = suivantDebut;
92      }else {
93        int suivantDebut = ((Integer) this.liste[this.debut+2]).intValue();
94        int dernier = ((Integer) this.liste[3*(this.taille-1)]).intValue();
95        int precedentDernier = ((Integer) this.liste[dernier+1]).intValue();
96        this.liste[suivantDebut+1] = -1;
97        this.liste[debut] = this.liste[dernier];
98        this.liste[debut+1] = this.liste[dernier+1];
99        this.liste[debut+2] = this.liste[dernier+2];
100       this.liste[precedentDernier] = debut;
101       this.debut = suivantDebut;
102     }
103     this.taille = this.taille -1;
104   }
105
106   public void supprimerFin() {
107     if (this.fin == 3*(this.taille-1)) {
108       int precedentFin = ((Integer) this.liste[this.fin+1]).intValue();
109       this.liste[precedentFin+2] = -1;
110       this.fin = precedentFin;
111     }else {
112       int precedentFin = ((Integer) this.liste[this.fin+1]).intValue();
113       int dernier = ((Integer) this.liste[3*(this.taille-1)]).intValue();
114       int precedentDernier = ((Integer) this.liste[dernier+1]).intValue();
115       int suivantDernier = ((Integer) this.liste[dernier+2]).intValue();
116       this.liste[precedentFin+2] = -1;
117       this.liste[fin] = this.liste[dernier];
118       this.liste[fin+1] = this.liste[dernier+1];
119       this.liste[fin+2] = this.liste[dernier+2];
120       this.liste[precedentDernier+2] = fin;
121       this.liste[suivantDernier+1] = fin;
122       this.fin = precedentFin;
123     }
124     this.taille = this.taille -1;
125   }
126
127   public void supprimer(int i) {
128     int cpt = 1;
129     int courant = debut;
130     while (cpt != i) {
131       courant = ((Integer) this.liste[courant+2]).intValue();
132       cpt = cpt +1;
133     }
134     if (courant == debut) {
135       this.supprimerDebut();
136     }else if (courant == fin) {
137       this.supprimerFin();
138     }else{
139       if (courant == 3*(this.taille-1)) {
140         int precedentDernier = ((Integer) this.liste[courant+1]).intValue();
141         int suivantDernier = ((Integer) this.liste[courant+2]).intValue();
142         this.liste[precedentDernier+2] = suivantDernier;
143         this.liste[suivantDernier+1] = precedentDernier;
144       }else{
145         int precedentCourant = ((Integer) this.liste[courant+1]).intValue();
146         int suivantCourant = ((Integer) this.liste[courant+2]).intValue();
```

```
147         int dernier = ((Integer) this.liste[3*(this.taille-1)]).intValue();
148         int precedentDernier = ((Integer) this.liste[dernier+1]).intValue();
149         int suivantDernier = ((Integer) this.liste[dernier+2]).intValue();
150         this.liste[precedentCourant+2] = suivantCourant;
151         this.liste[suivantCourant+1] = precedentCourant;
152         this.liste[courant] = this.liste[dernier];
153         this.liste[courant+1] = this.liste[dernier+1];
154         this.liste[courant+2] = this.liste[dernier+2];
155         this.liste[precedentDernier+2] = courant;
156         this.liste[suivantDernier+1] = courant;
157       }
158       if (3*(this.taille-1) == this.fin) {
159         this.fin = courant;
160       }else if (this.taille -1 == this.debut) {
161         this.debut = courant;
162       }
163       this.taille = this.taille -1;
164     }
165   }
166
167   public Object tete() {
168     return this.liste[debut];
169   }
170
171   public Object element(int i) {
172     int cpt = 1;
173     int courant = debut;
174     while (cpt != i) {
175       courant = ((Integer) this.liste[courant+2]).intValue();
176       cpt = cpt +1;
177     }
178     return this.liste[courant];
179   }
180
181   public Object suivant(int i) {
182     int cpt = 1;
183     int courant = debut;
184     while (cpt != i-1) {
185       courant = ((Integer) this.liste[courant+2]).intValue();
186       cpt = cpt+1;
187     }
188     return this.liste[courant+2];
189   }
190
191   public Object precedent(int i) {
192     int cpt = 1;
193     int courant = debut;
194     while (cpt != i) {
195       courant = ((Integer) this.liste[courant+2]).intValue();
196       cpt = cpt+1;
197     }
198     return this.liste[courant+1];
199   }
200
201   public void afficher() {
202     for(int i =0; i<this.liste.length;++i){
```

```
203         System.out.print(this.liste[i]+" ");
204       }
205       System.out.println();
206     }
207
208     @Override
209     public String toString() {
210       StringBuilder s = new StringBuilder();
211       int courant = debut;
212       while (courant != -1) {
213         s.append(this.liste[courant]+" ");
214         courant = ((Integer) this.liste[courant+2]).intValue();
215       }
216       s.append("\n");
217       return s.toString();
218     }
219
220     public Object[] elements(){
221       Object[] res = new Object[this.taille];
222       int i = 0;
223       int courant = debut;
224       while (courant != -1) {
225         res[i] = this.liste[courant];
226         courant = ((Integer) this.liste[courant+2]).intValue();
227         i = i+1;
228       }
229       return res;
230     }
231 }
```

Exercice 2

2.1 Allocation-Libération

Pour les deux implémentations, la méthode appliquée est la même. On utilise une liste indiquant le numéro des cellules libres. L'insertion des éléments dans la liste se fait à la manière d'une pile : on insère toujours au début de la liste (mais pas forcément au début du tableau dans les implémentations précédentes).

2.1.1 Allocation

Lorsqu'on veut allouer de la mémoire pour stocker un object, on va ajouter la donnée au debut de la liste de données pour garantir une allocation en temps constant (car on a un attribut pointant sur le début de la liste). L'algorihme utilisé est le suivant :

1. On récupère le numéro de la prochaine case libre (on récupère la tête de la liste des cellules libres)
2. On ajoute la donnée au début de la liste de données (voir insertion de l'implémentation concernée)
3. On retire la tête de la liste des cellules libres

2.1.2 Libération

Lorsqu'on veut libérer de la mémoire, on lui donne le numéro de cellule dans le tableau de données (par exemple : supprime moi la 3eme cellule). A noter que pour le cas de l'implémentation avec un tableau unique, la case concernée dans le tableau est la 9eme car les cellules occupent 3 cases dans le tableau. L'algorithme utilisé est le suivant :

1. On supprime la cellule dans la liste de données en modifiant uniquement les valeurs suivants/précédents des éléments suivant/précédents de l'élément concerné par la cellule en cours de suppression
2. On ajoute cette cellule dans la liste des libres (en ajoutant au début)

2.2 Compacte

Pour obtenir une liste compacte, on peut :

— On modifie libération qui décale à chaque fois d'une cellule les éléments dans le tableau de données. Ainsi, la liste des libres sera les numéros de cellules situés aprés les cellules occupées. Donc l'allocation ne change pas. On perd cependant la suppression de cellule en temps constant

— On peut utiliser les implémentations de l'exercice précédent car les listes sont toujours compactes et on a pas besoin de gérer une liste de libres.

2.3 Densifier

Pour densifier une liste, on peut utiliser l'algorithme suivant :

1. On déclare un tableau caractéristique de taille la taille du tableau de données (qui varie selon l'implémentation de la liste) rempli de 0
2. On parcourt la liste libre et pour chaque élément `k` de cette liste, on met 1 dans la case `k` du tableau caractéristique.
3. Maintenant qu'on sait quelles sont les cases libres, on peut savoir quelles sont les cases occupées : ce sont les indices du tableau caractéristique dont la valeur stockée est 0 (attention pour l'implémentation avec tableau unique, ce sont uniquement ceux qui sont multiples de 3)
4. On remarque que lorsque la liste est dense, tous les éléments libres sont situés aprés la taille de la liste et tous les éléments occupés sont situés avant. Donc dans notre tableau caractéristique, on va rechercher :
   (a) Les cases libres qui sont situées entre 0 et la taille du tableau -1
   (b) Les cases occupées qui sont situées entre la taille du tableau et la fin.
5. Une fois qu'on a ces indices,on peut associer une case libre avec une case occupée et il suffit de déplacer les données des cases occupées dans celles des cases libres.
6. Enfin, on actualise la liste des cases libres : soit on la vide et on insère tous les numéros de cellules aprés `taille` soit, pour chaque case libre dans la liste vide, on échange la valeur avec la case occupée associée

2.4 Code source

```
1  class ListeLibre extends ListeTab{
2    private ListeTab libre;
3
4    public ListeLibre (int max) {
5      super(max);
6      this.libre = new ListeTab(max);
7      for (int i=0;i<max;++i ) {
8        this.libre.insererDebut(new Integer(i));
9      }
10   }
11
12   public void allouer(Object o) { // Alloue une cellule pour stocker l'objet o
13       int position = ((Integer) this.libre.tete()).intValue();
14       this.libre.supprimerDebut();
15       this.liste[position+1] = -1;
16       if (this.estVide()) {
17         this.liste[position] = o;
18         this.liste[position+2] = -1;
19       }else {
20         this.liste[position] = o;
21         this.liste[position+2] = this.debut;
22         this.liste[this.debut+1] = position;
23       }
24       this.taille = this.taille+1;
25     }
26
27     public void liberer(int i) { // Libère la i-eme cellule de la mémoire
28       int position = 3*i;
29       if (position == this.debut) {
30         this.supprimerDebut();
```

```
31        }else if (position == this.fin) {
32          this.supprimerFin();
33        }else {
34          int precedent = ((Integer) this.liste[position+1]).intValue();
35          int suivant = ((Integer) this.liste[position+2]).intValue();
36          this.liste[precedent+2] = suivant;
37          this.liste[suivant+1] = precedent;
38        }
39        this.liste.insererDebut(new Integer(i));
40      }
41
42    public void densifierListe(){
43      int[] carac = new int[this.MAX*3];
44      Object[] elts = this.elements();
45      int[] tabLibre = new int[this.taille];
46      int[] tabOcc = new int[this.taille];
47      int courant = debut;
48      int debutLibre = 0;
49      int debutOcc = 0;
50      for (int i = 0; i<carac.length; ++i) {
51        carac[i] = 0;
52      }
53      for (int i =0; i<elts.length; ++i) {
54        tabLibre[i] = -1;
55        tabOcc[i] = -1;
56      }
57      for (int i =0; i<tabLibre.length; ++i) { // On construit le tableau caractérisque
        des données
58        int tmp = ((Integer) tabLibre[i]).intValue();
59        carac[tmp] = 1;
60      }
61      for (int i = 0; i<this.taille; ++i) { //tabLibre contient les indices libres < this.
        taille
62        if (carac[i] == 1) {
63          tabLibre[debutLibre] = i;
64          debutLibre = debutLibre +1;
65        }
66      }
67      for (int i = this.taille;i<carac.length;++i ) { //tabOcc contient les indices occupé
        es >= this.taille
68        if (carac[i] == 0 && (i% this.taille) == 0) {
69          tabOcc[i] = i;
70          debutOcc = debutOcc +1;
71        }
72      }
73      int j = 0;
74      while (j < this.taille && tabLibre[j] != -1) { // On swap les positions des données
75        int anciennePos = tabOcc[j];
76        int nouvellePos = tabLibre[j];
77        int precedentAnciennePos = ((Integer) this.liste[anciennePos+1]).intValue();
78        int suivantAnciennePos = ((Integer) this.liste[anciennePos+2]).intValue();
79        this.liste[nouvellePos] = this.liste[anciennePos];
80        this.liste[nouvellePos+1] = this.liste[anciennePos+1];
81        this.liste[nouvellePos+2] = this.liste[anciennePos+2];
82        this.liste[precedentAnciennePos+2] = nouvellePos;
83        this.liste[suivantAnciennePos+1] = nouvellePos;
```

```
      if (anciennePos == this.debut) {
        this.debut = nouvellePos;
      }
      if (anciennePos == this.fin) {
        this.fin = nouvellePos;
      }
    }
    // On remet la liste des cases libres à jour
    while (this.libre.estVide() == false) {
      this.libre.supprimerDebut();
    }
    for (int i = 3*this.taille; i< this.MAX*3; i = i+3 ) {
      this.libre.insererDebut(new Integer(i));
    }
  }
}
```

```
class ListeLibreM extends ListeTabM{
  private ListeTabM libre;

  public ListeLibre (int max) {
    super(max);
    this.libre = new ListeTab(max);
    for (int i=0;i<max;++i ) {
      this.libre.insererDebut(new Integer(i));
    }
  }
   public void allouer(Object o) { // Alloue une cellule pour stocker l'objet o
    int position = ((Integer) this.libre.tete()).intValue();
    this.libre.supprimerDebut();
    this.tPrecedent[position] = -1;
    if (this.estVide()) {
      this.liste[position] = o;
      this.tSuivant[position] = -1;
    }else {
      this.liste[position] = o;
      this.tSuivant[position] = this.debut;
      this.tPrecedent[debut] = position;
    }
    this.taille = this.taille+1;
  }

  public void liberer(int i) { // Libère la i-eme cellule de la mémoire
    if (i == this.debut) {
      this.supprimerDebut();
    }else if (i == this.fin) {
      this.supprimerFin();
    }else {
      int precedent = this.tPrecedent[position];
      int suivant = this.tSuivant[position]
      this.tSuivant[precedent] = suivant;
      this.tPrecedent[suivant] = precedent;
    }
    this.libre.insererDebut(new Integer(i));
  }
}
```

Exercices 3 – 5

3.1 Principe

On souhaite allouer de la mémoire sous forme de blocs, et également que les blocs alloués soient positionnés de manière contigüe dans la mémoire. Ces blocs peuvent être de taille différente, et on souhaite pouvoir les libérer.

Puisque la mémoire allouée doit être contigüe, on devra retourner à l'utilisateur un identifiant qui permettra plus tard de retrouver l'emplacement réel de la donnée.

On va donc utiliser 3 tableaux et une liste :
— Un tableau qui contiendra les données (`data`),
— Une liste qui permettra de stocker les espaces libres (`free`),
— Un tableau caractéristique de la mémoire qui a chaque indice de début de bloc dans la mémoire fait correspondre la longueur de ce bloc (`allocated`),
— Un tableau qui servira de table de correspondance entre la valeur retournée à l'utilisateur et l'emplacement dans la mémoire (`redirect`), dont les éléments sont initialisés à -1,
— Enfin, on aura besoin de stocker la capacité totale de la mémoire (`length`) et la taille d'unité d'allocation, c'est-à-dire la taille par défaut d'un bloc mémoire (`allocLength`).

3.2 Allocation

Pour allouer de la mémoire, on exécutera l'algorithme suivant :

1. On récupère l'indice de la première case libre (tête de la liste `free`). Si la liste est vide, la mémoire est pleine, on lance alors une exception,
2. Si une case est libre, on vérifie qu'il reste assez de mémoire pour réserver l'espace demandé, sinon on lance une exception,
3. Si l'espace est disponible, on stocke la longueur du bloc dans `allocated` à l'indice récupéré précédemment,
4. On calcule la position du prochain bloc libre et on la stocke dans la liste `free`,
5. On recherche ensuite dans `redirect` la première case libre (valant -1) et on récupère son indice (`i`),
6. On stocke l'indice de l'emplacement mémoire dans `redirect[i]` et on retourne `i`.

3.3 Libération

Pour libérer de la mémoire, l'opération est plus délicate, car on doit décaler les blocs mémoire qui sont après le bloc libéré, s'il y en a, afin de garantir que les blocs alloués seront contigus en mémoire.

On donnera à la fonction l'entier qui a été retourné lors de l'allocation.

On pourra utiliser l'algorithme suivant :

1. On récupère l'adresse réelle où est stocké l'objet avec le tableau `redirect`,
2. On récupère la longueur du bloc dans `allocated`,
3. On indique dans `redirect` que la case est libre (passage à -1),
4. On calcule la position `i` du début du prochain bloc,
5. On décale tous les blocs suivants en pensant à actualiser les indices dans `redirect` et les longueurs des blocs dans `allocated`,
6. On indique que l'indice du premier bloc libre correspond maintenant à celui indiqué précédemment auquel on a soustrait la longueur réelle du bloc libéré.

3.4 Code source

```
1  class MemoryErrorException extends Exception{};
2  class Memory {
3    private int[] redirect;
4    private int[] allocated;
5    private Object[] data;
6    private int length;
7    private Liste free;
8    private int allocLength;
9
10   Memory(int capacity, int allocLength){
11     this.redirect = new int[capacity];
12     for (int i = 0; i < capacity; ++i) {
13       this.redirect[i] = -1;
14     }
15     this.allocated = new int[capacity];
16     this.data = new Object[capacity];
17     this.free = new ListeTab(capacity);
18     this.free.insererDebut(new Integer(0));
19     this.length = capacity;
20     this.allocLength = allocLength;
21   }
22
23   public int malloc(int n) throws MemoryErrorException{
24     if (!this.free.estVide()){
25       int firstFree = ((Integer) this.free.tete()).intValue();
26       if (firstFree + n*this.allocLength <= this.length){
27         this.free.supprimerDebut();
28         this.allocated[firstFree] = n;
29         int i = 0;
30         while (i<this.length && this.redirect[i]!=-1) {
31           ++i;
32         }
```

```
        if (i == this.length){
          throw new MemoryErrorException();
        }else{
          this.redirect[i] = firstFree;
          if (firstFree + n*this.allocLength < this.length){
            this.free.insererDebut(firstFree+(n*this.allocLength));
          }
          return i;
        }
      }else{
        throw new MemoryErrorException();
      }
    }else{
      throw new MemoryErrorException();
    }
  }

  public void free(int A) throws MemoryErrorException{
    int index = this.redirect[A];
    int length = this.allocated[index];
    this.redirect[A] = -1;
    int i = index + allocated[index]*this.allocLength;
    int offset = index;
    int end;
    if (this.free.estVide()){
      end = this.length-length*this.allocLength;
    }else{
      end = ((Integer) this.free.tete()).intValue()-length*this.allocLength;
      this.free.supprimerDebut();
    }
    while (i < end) {
      int j = 0;
      while (j < this.length && this.redirect[j] != i){
        ++j;
      }
      this.allocated[offset]=this.allocated[this.redirect[j]];
      this.redirect[j]=offset;
      for (int k = i; k<i+this.allocated[i]*this.allocLength; ++k){
        this.data[offset]=this.data[k];
        offset++;
        i++;
      }
    }
    this.free.insererDebut(new Integer(end));
  }
```

Exercice 6

6.1 Principe

Les tableaux dynamiques gèrent tout seuls leur taille.

Afin de garantir qu'on pourra toujours ajouter des éléments, on vérifie lors de l'insertion de données que le nombre d'éléments reste inférieur ou égal au nombre de cases disponibles. Lorsqu'on ne dispose plus de suffisamment d'espace, on doit créer un nouveau tableau plus grand dans lequel on recopie le contenu de l'ancien. On peut alors ajouter l'élément.

Dans le cas présent, on choisira de doubler la taille du tableau dès qu'il ne reste plus de place.

Les autres opérations sur ce genre de tableau ne changent pas.

6.2 Code source

```
1  class TableauDynamique {
2    Object[] tab; // Tableau qui va contenir les données
3    int taille; // taille du tableau (nombre d'élément réellement insérés)
4
5    public TableauDynamique(int t) {
6      this.tab = new Object[t];
7      this.taille = 0;
8    }
9
10   public Object get(int i){
11     if (i > this.size()) {
12       throw new IllegalArgumentException("\n Indice invalide");
13     }
14     return this.tab[i];
15   }
16
17   public int size() {
18     return this.taille;
19   }
20
21   public void add(Object val) {
22     if (this.size() < this.tab.length) { // Si on a encore de la place, on ajoute l'élé
       ment et on incrémente la taille
23       this.tab[this.size()] = val;
24       taille++;
```

```
25      }
26      else { // Sinon, on doit copier les éléments dans un plus grand tableau avant de
        pouvoir ajouter
27        Object[] nvTab = new Object[2*this.size()];
28        for (int i = 0; i < this.tab.length; ++i) {
29          nvTab[i] = this.get(i);
30        }
31        nvTab[this.size()] = val;
32        taille++;
33        this.tab = nvTab;
34      }
35    }
36
37    public boolean contains(Object val) {
38      boolean res = false;
39      int j = 0;
40      while (j < this.size() && !res) {
41        if (this.get(j) == val) {
42          res = true;
43        }
44        j++;
45      }
46      return res;
47    }
48
49    public void remove(Object val) {
50      int j = 0;
51      boolean res = false;
52      while (j < this.size() && !res) {
53        if (this.get(j) == val) {
54          res = true;
55        }
56        j++;
57      }
58      if (res) { // Si on a trouvé l'élément, on doit décaler d'un cran vers la gauche
        tous les éléments du tableau à partir de l'indice de l'élément (inclus).
59        for (int i = j-1; i < this.size() -1; ++i) {
60          this.tab[i] = this.get(i+1);
61        }
62        this.taille--;
63      }
64    }
65
66    public void reverse() {
67      Object tmp;
68      for (int i = 0; i<this.size()/2; ++i) {
69        tmp = this.get(this.size()-1-i);
70        this.tab[this.size()-1-i] = this.get(i);
71        this.tab[i] = tmp;
72      }
73    }
74
75    @Override
76    public String toString() {
77      StringBuilder s= new StringBuilder("Le tableau contient "+this.size()+" éléments : "
        );
```

```
78      for (int i=0; i < this.size();++i) {
79        s.append(this.get(i)+" ");
80      }
81      return s.toString();
82    }
83  }
```