

# Structures de données

---

Semestre 4



---

# Avant-propos

---

Suite du module d'algorithmique et programmation, accent sur les structures de données

- Pile
- File
- Arbre
- ...

## Heures

- 24h de CTDI
- 26 de TDM

## Notation

**Contrôle intermédiaire** 30%

**Contrôle terminal** 50%

**TP** 20%

**TP Noté** 50%

**Devoir écrit** 25%

**Devoir TP** 25 %

---

# Table des matières

---

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Types de données Abstraits (TAD)</b>             | <b>5</b>  |
| 1.1      | Syntaxe des TAD . . . . .                           | 5         |
| 1.2      | Implémentation d'un TAD . . . . .                   | 6         |
| 1.3      | Protection du TAD . . . . .                         | 7         |
| <b>2</b> | <b>Structures de données classiques</b>             | <b>9</b>  |
| 2.1      | Pile . . . . .                                      | 9         |
| 2.2      | File . . . . .                                      | 13        |
| 2.3      | File avec priorité . . . . .                        | 18        |
| 2.4      | Liste avec priorité . . . . .                       | 19        |
| <b>3</b> | <b>Parcourir une collection</b>                     | <b>23</b> |
| 3.1      | Itérateur sur la liste doublement chaînée . . . . . | 23        |
| <b>4</b> | <b>Les structures arborescentes</b>                 | <b>26</b> |
| 4.1      | L'arbre GRD : «Gauche Racine Droite» . . . . .      | 26        |
| 4.2      | Les arbres rouges noirs . . . . .                   | 31        |
| <b>A</b> | <b>Cours sur les pointeurs en C</b>                 | <b>37</b> |
| A.1      | Syntaxe . . . . .                                   | 37        |
| A.2      | Opérateur autorisés sur les pointeurs . . . . .     | 38        |
| A.3      | Pointeur sur fonction . . . . .                     | 39        |
| <b>B</b> | <b>Liste des codes sources</b>                      | <b>41</b> |
| <b>C</b> | <b>Table des figures</b>                            | <b>42</b> |

---

|  |           |
|--|-----------|
| <b>D Exercices</b>   | <b>43</b> |
| D.1 Pointeurs . . . . .  | 43        |
| <b>E Cours obtenu sur Moodle</b>                                   | <b>46</b> |
| E.1 Les Types Abstraits de Données . . . . .                       | 46        |
| E.2 Le TAD <b>File</b> et ses implémentations . . . . .            | 63        |
| E.3 Liste doublement chaînée en dynamique . . . . .                | 65        |
| E.4 Itérateur pour Liste doublement chaînée en dynamique . . . . . | 67        |
| E.5 Arbre binaire d'entiers en dynamique . . . . .                 | 69        |
| E.6 Arbres rouges et noirs . . . . .                               | 73        |

# Les structures arborescentes

## Sommaire


|   |           |
|---|-----------|
| <b>4.1 L'arbre GRD : «Gauche Racine Droite»</b> | <b>26</b> |
| <b>4.2 Les arbres rouges noirs</b>              | <b>31</b> |

Nous allons voir deux types d'arbres :

1. L'arbre GRD : « Gauche Racine Droite »

2. Les arbres rouges noirs

Ce sont des arbres binaires : chaque noeud de l'arbre à au lu deux fils.

 Les informations sont rangés dans l'arbre en respectant un certain critère

## 4.1 L'arbre GRD : «Gauche Racine Droite»

### 4.1.1 Critère de rangement

Quelque soit le noeud de l'arbre :

- les informations rangées à gauche de la racine de ce noeud sont inférieur ou égal à cette racine.
- les informations rangées à droite de la racine de ce noeud sont supérieur à cette racine.

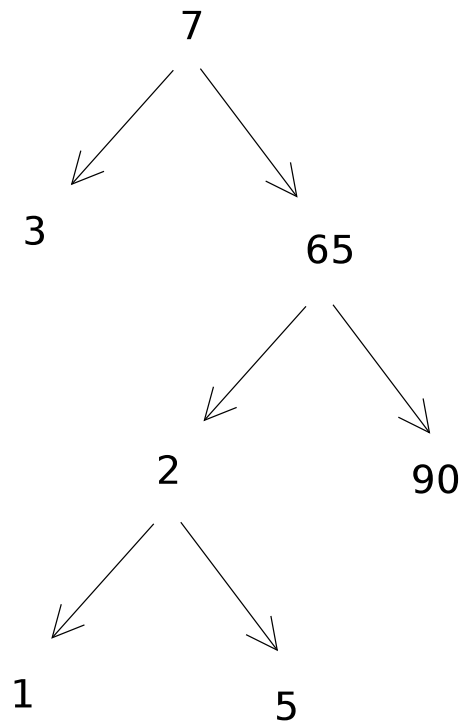


FIGURE 4.1 – Arbre GRB

Cet arbre est prévu pour effectuer un parcours en profondeur.

#### 4.1.2 Implémentation du TAD

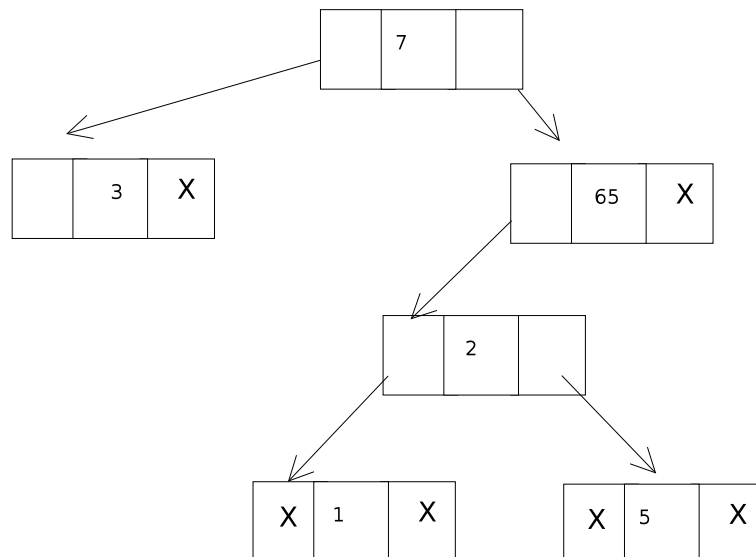


FIGURE 4.2 – Implémentation de l'arbre GRB

```

1 typedef struct etCell* Arbre;
2
3 Arbre creerGrd(void);
4 Arbre ajout(const Arbre arbre, int e);
5 void afficher(const Arbre arbre);
6 Arbre afficherIteratif(Arbre pArbre);

```

```
7 | int estVide(const Arbre arbre);
8 | int appartient(const Arbre arbre, int racine);
```

Listing 4.1 – Arbre GRD – Header

```
1 | typedef struct etCell {
2 |     struct etCell* gauche;
3 |     struct etCell* droite;
4 |     int racine;
5 | } Cell;
6 |
7 | Arbre creerGrd(void) {
8 |     return (NULL);
9 | }
10 |
11 | Arbre ajout(const Arbre arbre, int e) {
12 |     Cell* newCell;
13 |     newCell = (Cell*)malloc(sizeof(Cell));
14 |     newCell->racine = e;
15 |     newCell->gauche = NULL;
16 |     newCell->gauche = NULL;
17 |     if(estVide(arbre)) {
18 |         arbre = newCell;
19 |     } else {
20 |         if(e <= arbre->racine) {
21 |             arbre->gauche = ajout(arbre->gauche, e);
22 |         } else {
23 |             arbre->droite = ajout(arbre->droite, e);
24 |         }
25 |     }
26 |
27 |     return arbre;
28 | }
29 |
30 | // Parcours infixe
31 | void afficher(const Arbre arbre) {
32 |     Cell* cous = a;
33 |     if(!estVide(a)) {
34 |         affiche(a->gauche);
35 |         printf("%d", a->racine);
36 |         afficher(a->droite);
37 |     }
38 | }
39 |
40 | int estVide(const Arbre arbre) {
41 |     return (arbre != NULL);
42 | }
43 |
44 | int appartient(const Arbre arbre, int val) {
45 |     if(estVide(arbre))
46 |         return 0;
47 |     if(val <= arbre->racine)
48 |         return appartient(arbre->gauche, val);
49 |     if(val > arbre->racine)
50 |         return appartient(arbre->droite, val);
51 |
52 |     return 1;
53 | }
54 | // Private
55 | Arbre restructure(Arbre pArbre) {
56 |     // on est sur le noeud à créer
57 |     Arbre droit, gauche, aux;
58 |     droit = pArbre->droite;
```



```

59 gauche = pArbre->gauche;
60
61 if(droit == NULL) {
62     // on a rien à insérer
63     return gauche;
64 }
65 aux = droit;
66 while(aux->gauche != NULL) {
67     aux = aux->gauche;
68 }
69 aux->gauche = gauche;
70 free(pArbre);
71
72 return droit;
73 }
74 // Private
75 Arbre supprimerUnElement(Arbre pArbre, int val) {
76     assert(!estVide(pArbre));
77     if(pArbre->racine == val) {
78         return(restructure(pArbre, val));
79     }
80     if((pArbre->racine < val) && (pArbre->gauche != NULL)) {
81         pArbre->droit = supprimerUnElement(a->droit, val);
82     } else if((a->racine > val) && (a->gauche != NULL)) {
83         pArbre->gauche = supprimerUnElement(a->gauche, val);
84     }
85
86     return pArbre;
87 }
88
89 Arbre supprime(Arbre pArbre, int v) {
90     while(appartient(pArbre, v)) {
91         pArbre = supprimerUnElement(pArbre, v);
92     }
93     return pArbre;
94 }

```

Listing 4.2 – Arbre GRD – Implémentation

### 4.1.3 Différents types de parcours

**Parcours infixe** On parcourt à gauche, on appelle la valeur, on parcourt à droite.

**Parcours préfixe** On parcourt on appelle la valeur puis on parcourt à gauche et à droite.

**Parcours postfixe** On parcourt à droite puis à gauche et ensuite on appelle la valeur.

#### 4.1.3.1 Exercices de parcours

Écrire une fonction qui permette l'affichage en profondeur d'un arbre GRD mais sans utiliser la récursivité.

Nous avons utilisé une Pile afin de simuler des appels récursifs (Pile système). La pile contient le nœud courant.

On suppose que l'on dispose du TAD Pile d'Element avec le type élément qui est une Cell.

```

1 Arbre afficherIteratif(Arbre pArbre) {
2     Cell n;
3     Arbre aux;

```

```
4  if(!estVide(pArbre)) {
5      File p = creer();
6      p = empiler(p, *pArbre);
7      while(!pileEstVide(p)) {
8          n = sommetPile(p);
9          p = depiler(p);
10         if((n.gauche == NULL) && (n.droite == NULL)) {
11             printf("%d ", n.racine);
12         } else {
13             if(n.droit != NULL) {
14                 p = empiler(p, *(n.droit));
15             }
16             aux = n.gauche;
17             n.droit = NULL;
18             n.gauche = NULL;
19             p = empiler(p,n);
20             if(aux != NULL) {
21                 p = empiler(p, *aux);
22             }
23         }
24     }
25 }
26 }
```

Listing 4.3 – Arbre GRD – Implémentation fonction affichage en profondeur itératif

Pour parcourir l'arbre en largeur, le principe est le même, à la place d'utiliser une *Pile* nous allons utiliser une *File*.

```
1  Arbre afficherLargeurIteratif(Arbre pArbre) {
2      Cell n;
3      if(!estVide(pArbre)) {
4          File p = creer();
5          p = enfiler(p, *pArbre);
6          while(!fileEstVide(p)) {
7              n = sommetFile(p);
8              p = defiler(p);
9              printf("%d ", n.racine);
10             if(n.gauche != NULL) {
11                 p = enfiler(p, *(n.gauche));
12             }
13             p = enfiler(p,n);
14             if(n.droit != NULL) {
15                 p = enfiler(p, n.droit);
16             }
17         }
18     }
19 }
```

Listing 4.4 – Arbre GRD – Implémentation fonction affichage en longueur

#### 4.1.4 Hauteur

Si on considère  $h$  la hauteur d'un arbre<sup>1</sup>, la complexité des fonctions d'ajout et de recherche d'un arbre sont de complexité  $O(h)$ ,  $h$  doit donc être le plus petit possible.

---

1. La hauteur d'un arbre est la distance entre la racine et les feuilles

Pour cela, l'arbre doit être le plus équilibré possible pour cela, nous pouvons utiliser un arbre rouge-noir.

## 4.2 Les arbres rouges noirs

Cet arbre est presque équilibré, il comporte plusieurs propriétés.

1. Par convention, un arbre vide est **noir**
2. Les fils d'un nœud rouge sont **noirs**
3. Le nombre de nœud noirs le long d'une branche dans l'arbre est indépendant de la branche

Le problème principal est donc de maintenir l'arbre rouge-noir.

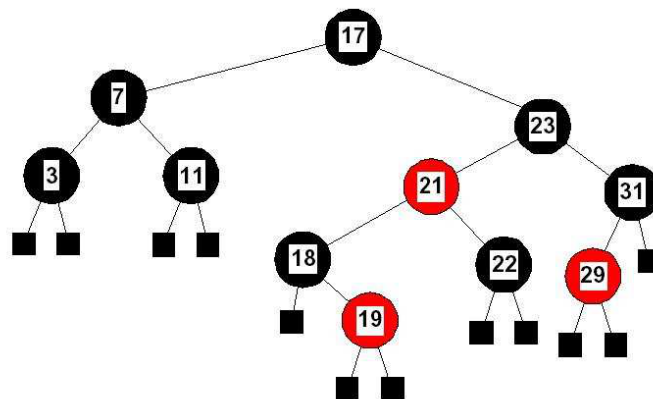


FIGURE 4.3 – Exemple d'un arbre Rouge-Noir

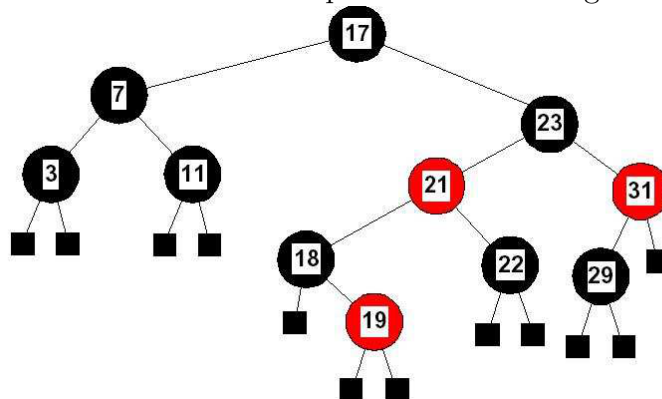


FIGURE 4.4 – Exemple d'un arbre non Rouge-Noir

### 4.2.1 Utilisation de rotation

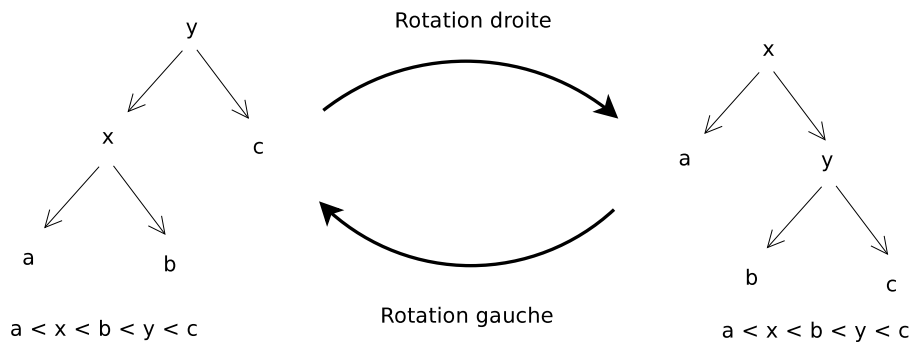


FIGURE 4.5 – Principe de la rotation

**⚠** Les arbres rouges noirs sont des arbres GRD particuliers, ils possèdent donc les propriétés de l'arbre GRD

### 4.2.2 Ajout d'une valeur dans un arbre rouge-noir

La valeur ajoutée sera mise dans un noeud rouge dont les fils sont vides (donc noirs) avec une instruction classique dans l'arbre (comme un arbre GRD)

Plusieurs cas sont possibles :

- C'est le 1<sup>er</sup> noeud de l'arbre  $\Rightarrow$  rien à faire
- Son père est noir  $\Rightarrow$  rien à faire
- Son père est rouge  $\Rightarrow$  plusieurs sous cas.

1. Le père est la racine de l'arbre  $\Rightarrow$  la racine devient noire

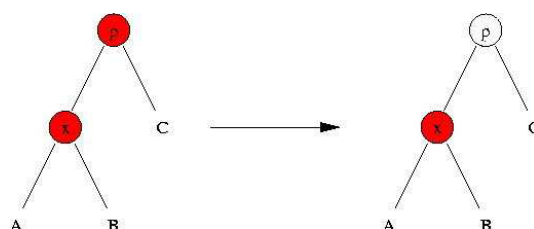


FIGURE 4.6 – Le père est la racine de l'arbre

2. Le frère du père est rouge  $\Rightarrow$  Alors le frère du père et le père passe noir et le grand-père devient rouge et on propage

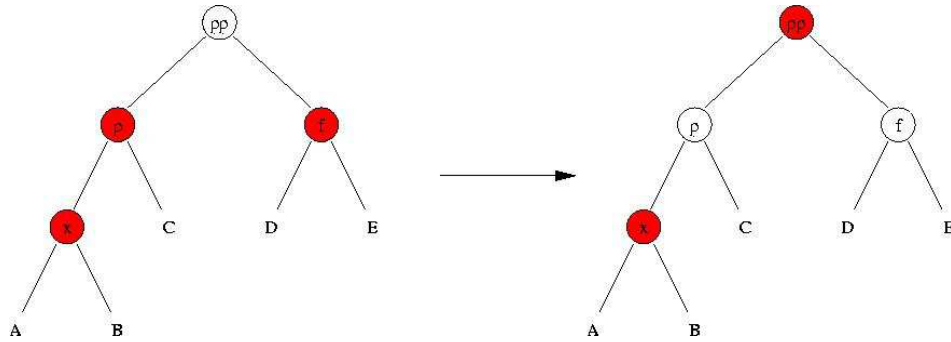


FIGURE 4.7 – Le frère du père est rouge

3. Le frère f de p est noir
  - a. x est le fils gauche de p

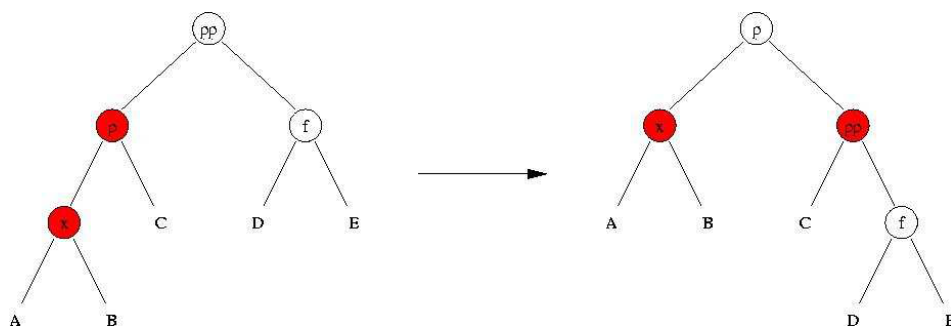


FIGURE 4.8 – x est le fils gauche de p

- b. x est le fils droit de p

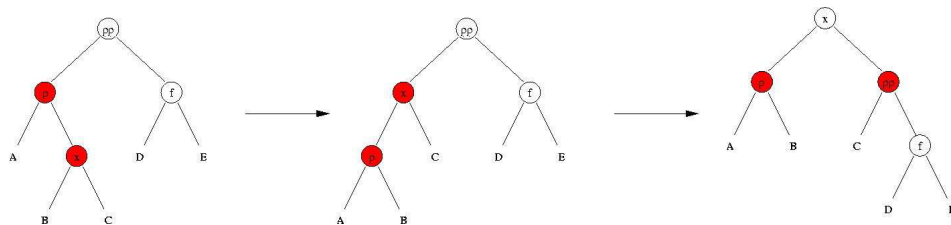


FIGURE 4.9 – x est le fils droit de p

**R** La seule propagation se fait dans le cas 1 (pas de propagation dans le cas 2)

```

1 #define NOIR 1
2 #define ROUGE 0
3
4 typedef struct etNoeud {
5     int racine;
6     struct etNoeud *sag;
7     struct etNoeud *sad;
8     struct etNoeud *pere;
9     int couleur; /* 1 = noir, 0 rouge */
10 } Noeud;
11 typedef Noeud* ARN;
12
13 ARN creerRN(void) {
14     return (NULL);
15 }

```

```
16
17 ARN rotationDroite(ARN a) {
18     Noeud buff;
19     if((a != NULL) && (a->sag != NULL)) {
20         buff = a->sag;
21         buff->pere = a->pere;
22         a->sad = aux->sag;
23         if(a->sag != NULL)
24             a->sag->pere = a;
25         buff->sad = a ;
26         buff->sad->pere = buff;
27
28         return buff;
29     } else {
30         return a;
31     }
32
33     return a;
34 }
35
36 ARN rotationGauche(ARN a) {
37     Noeud buff;
38     if((a != NULL) && (a->sad != NULL)) {
39         buff = a->sad;
40         buff->pere = a->pere;
41         a->sag = aux->sad;
42         if(a->sad != NULL)
43             a->sad->pere = a;
44         buff->sag = a ;
45         buff->sag->pere = buff;
46
47         return buff;
48     }
49
50     return a;
51 }
52
53 void miseAJourLien(ARN nouveauFils, ARN ancienFils, ARN pere, ARN* origineArbre) {
54     if(pere == NULL) {
55         *origineArbre = nouveauFils;
56     } else if(pere->sag == ancienFils) {
57         pere->sag = nouveauFils;
58     } else {
59         pere->sad = nouveauFils;
60     }
61 }
62 /*
63  * @param aux pointeur sur x
64  * @param a pointeur sur le pointeur sur la racine de l'arbre
65  * @return Le pointeur sur la cellule à partir de laquelle on propage
66  *         Si NULL pas de propagation
67  */
68 ARN equilibre(ARN aux, ARN* a) {
69     // aux est la racine de l'arbre ou aux à un père non vide
70     if(aux->pere == NULL || aux->pere->couleur == NOIR) {
71         return NULL;
72     }
73
74     // le père de aux n'est pas vide et il est rouge
75     if(aux->pere->pere == NULL) { // Cas 0
76         aux->pere->couleur = NOIR;
77         return NULL;
```

```

78 }
79
80 // le grand père de aux n'est pas vide
81 // Donc il existe un frère du père
82 // Identification du frère et des symétries
83 int pereAGauche = 0;
84 int auxAGauche = 0;
85 ARN frere;
86 ARN pere = aux->pere;
87
88 if(pere->pere->sag == pere) {
89     pereAGauche = 1;
90     frere = pere->pere->sad;
91 } else {
92     frere = pere->pere->sag;
93 }
94
95 auxAGauche = (pere->sag == aux);
96
97 if((frere != NULL) && (frere->couleur == ROUGE)) { // cas 1
98     pere->pere->couleur = ROUGE;
99     frere->couleur = NOIR;
100     return (pere->pere);
101 }
102
103 // le frère du père est noir
104 // Cas 2
105 if(pereAGauche == auxAGauche) { // Cas 2a
106     ARN = aGdPere = pere->pere->pere;
107     ARN = gdPere = pere->pere;
108     aux = (pereAGauche) ? rotationDroite(gdPere) : rotationGauche(gdPere);
109     // mise à jour eventuelle de la racine de l'arbre
110     miseAJourLien(aux, gdPere, a);
111     aux->couleur = NOIR;
112
113     // Mise à jour de la couleur de l'ancien grand père
114     if(pereAGauche) {
115         aux->sad->couleur = ROUGE;
116     } else {
117         aux->sag->couleur = ROUGE;
118     }
119 } else { // cas 2b
120     ARN gdPere = pere->pere;
121
122     aux = (pereAGauche) ? rotationGauche(pere) : rotationDroite(pere);
123
124     miseAJourLien(aux, pere, gdPere, a);
125     gdPere = aux->pere->pere;
126     pere = aux->pere;
127
128     aux = (pereAGauche) ? rotationDroite(pere) : rotationGauche(pere);
129
130     miseAJourLien(aux, pere, gdPere, a);
131     aux->couleur = NOIR;
132
133     if(pereAGauche) {
134         aux->sad->couleur = ROUGE;
135     } else {
136         aux->sag->couleur = ROUGE;
137     }
138
139     return NULL;

```

```
140     }
141 }
142
143 /*
144  * @param aux L'adresse de l'adresse ou on à ajouté
145  */
146 ARN ajoutGrd(ARN a, int v, ARN* aux) {
147     return ajoutGrdaux(a,v,aux,NULL); // NULL <=> L'adresse du père
148 }
149 ARN ajoutGrdAux(ARN a, int v, ARN* aux, ARN pere) {
150     if(a != NULL) {
151         if(a->racine < v) {
152             a->sad = ajoutGrdAux(a->sad, v, aux, a);
153         } else {
154             a->sag = ajouterARN(a->sad, v, aux, a);
155         }
156
157         return a;
158     } else {
159         // on ajoute ici
160         *aux = (ARN) malloc(sizeof(sizeof(noeud)));
161         assert(*aux != NULL);
162         (*aux)->racine = v;
163         (*aux)->racine = v;
164         (*aux)->sad = NULL;
165         (*aux)->sag = NULL;
166         (*aux)->couleur = rouge;
167         (*aux)->pere = pa;
168
169         return *aux;
170     }
171 }
172 ARN ajouterARN(ARN a, int v) {
173     ARN aux;
174     /* en completant avec la mise à jour du père et de la couleur rouge*/
175     a = ajoutGrd(a, v, &aux); // TODO Ajout d'un troisième paramètre
176
177     while(aux != NULL) {
178         aux = equilibre(aux, &a);
179     }
180
181     return a;
182 }
```

Listing 4.5 – Arbre GRD – Implémentation



# Liste des codes sources

|      |  |    |
|------|--|----|
| 1.1  | Opérations du TAD Pile . . . . .   | 6  |
| 1.2  | Type de la pile statique originel – Présent dans le .h . . . . .                               | 7  |
| 1.3  | Type de la pile statique – Présent dans le .h . . . . .  | 8  |
| 1.4  | Type de la pile statique – Présent dans le .c . . . . .  | 8  |
| 1.5  | Modification de la fonction <code>creer</code> s’adaptant à la protection de données . . . . . | 8  |
| 2.1  | Pile sans protection du type – Header . . . . .  | 9  |
| 2.2  | Pile statique sans protection du type – Implémentation . . . . .                               | 9  |
| 2.3  | Pile statique – Ajout de <code>remplacerOccurence</code> . . . . .                             | 10 |
| 2.4  | Pile statique avec protection du type – Implémentation . . . . .                               | 11 |
| 2.5  | Pile dynamique – Implémentation . . . . .  | 12 |
| 2.6  | File – Axiones . . . . .   | 13 |
| 2.7  | File – Headers . . . . .   | 14 |
| 2.8  | File statique – Implémentation . . . . .   | 14 |
| 2.9  | File dynamique – Implémentation . . . . .  | 16 |
| 2.10 | File dynamique – Ajout <code>concat</code> et <code>mixe</code> . . . . .                      | 17 |
| 2.11 | File – Application de fusion de voies routières . . . . .                                      | 18 |
| 2.12 | Element – Prototype <code>comparer</code> . . . . .  | 19 |
| 2.13 | Liste doublement chaînée – Header . . . . .  | 20 |
| 2.14 | Liste doublement chaînée – Implémentation . . . . .  | 20 |
| 3.1  | Iterateur sur Liste – Header . . . . .   | 24 |
| 3.2  | Iterateur sur Liste – Implémentation . . . . .   | 24 |
| 4.1  | Arbre GRD – Header . . . . .   | 27 |
| 4.2  | Arbre GRD – Implémentation . . . . .   | 28 |
| 4.3  | Arbre GRD – Implémentation fonction affichage en profondeur itératif . . . . .                 | 29 |
| 4.4  | Arbre GRD – Implémentation fonction affichage en longueur . . . . .                            | 30 |
| 4.5  | Arbre GRD – Implémentation . . . . .   | 33 |
| A.1  | Syntaxe de déclaration d’un pointeur . . . . .   | 37 |
| A.2  | Exemple de déclaration . . . . .   | 37 |
| A.3  | Syntaxe utilisation d’un pointeur . . . . .  | 37 |
| A.4  | Exemple d’utilisation d’un pointeur . . . . .  | 37 |
| A.5  | Exemple d’utilisation de la constante <code>NULL</code> . . . . .                              | 37 |
| A.6  | Syntaxe d’allocation dynamique . . . . .   | 38 |
| A.7  | Exemple d’allocation dynamique . . . . .   | 38 |
| A.8  | Syntaxe de libération de mémoire . . . . .   | 39 |
| A.9  | Déclaration d’un pointeur de fonction . . . . .  | 39 |
| A.10 | Utilisation d’un pointeur de fonction . . . . .  | 39 |
| A.11 | Exemple d’utilisation d’un pointeur de fonction . . . . .                                      | 39 |
| D.1  | Pointeurs – Exercice 1 . . . . .   | 43 |
| D.2  | Pointeurs – Exercice 2 . . . . .   | 43 |
| D.3  | Pointeurs – Exercice 3 . . . . .   | 44 |
| D.4  | pointeurs – Exercice 4 . . . . .   | 44 |

---

# Table des figures

---

|     |   |    |
|-----|---|----|
| 1.1 | Principe de base d'un TAD . . . . .               | 5  |
| 2.1 | Pile avec une liste simplement chaînée . . . . .  | 12 |
| 2.2 | Pile avec une liste doublement chaînée . . . . .  | 12 |
| 2.3 | File avec une liste simplement chaînée . . . . .  | 15 |
| 2.4 | File avec une liste doublement chaînée . . . . .  | 16 |
| 2.5 | Liste doublement chaînée . . . . .                | 19 |
| 3.1 | Liste doublement chaînée avec itérateur . . . . . | 23 |
| 4.1 | Arbre GRB . . . . .                               | 27 |
| 4.2 | Implémentation de l'arbre GRB . . . . .           | 27 |
| 4.3 | Exemple d'un arbre Rouge-Noir . . . . .           | 31 |
| 4.4 | Exemple d'un arbre non Rouge-Noir . . . . .       | 31 |
| 4.5 | Principe de la rotation . . . . .                 | 32 |
| 4.6 | Le père est la racine de l'arbre . . . . .        | 32 |
| 4.7 | Le frère du père est rouge . . . . .              | 33 |
| 4.8 | x est le fils gauche de p . . . . .               | 33 |
| 4.9 | x est le fils droit de p . . . . .                | 33 |

# Types abstraits de données (TAD)

## CTD 1

J.P. Bahsoun, M.C. Lagasquie, M.Paulin

16 août 2011

### Table des matières

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>2</b>  |
| <b>2</b> | <b>La spécification d'un TAD</b>  | <b>3</b>  |
| <b>3</b> | <b>Implémentation d'un TAD</b>  | <b>4</b>  |
| 3.1      | Implanter une représentation interne du type et implémenter les fonctions . . . . . | 4         |
| 3.1.1    | Implémentation Statique du TAD PILE . . . . .                                       | 4         |
| 3.1.2    | Cours sur les pointeurs . . . . .   | 5         |
| 3.1.3    | Rappel sur les structures . . . . .   | 5         |
| 3.1.4    | Les pointeurs + les structures . . . . .  | 5         |
| 3.1.5    | Implémentation Dynamique du TAD PILE . . . . .                                      | 5         |
| 3.2      | Séparer interface et corps . . . . .  | 7         |
| 3.3      | Protection du type . . . . .  | 8         |
| <b>A</b> | <b>Cours sur les pointeurs</b>  | <b>9</b>  |
| A.1      | Définition . . . . .  | 9         |
| A.2      | Syntaxe en langage C . . . . .  | 9         |
| A.3      | Exemples sur les pointeurs . . . . .  | 10        |
| A.4      | Conclusion sur les pointeurs . . . . .  | 12        |
| <b>B</b> | <b>Rappels sur les structures</b>   | <b>12</b> |
| B.1      | Définition . . . . .  | 12        |
| B.2      | Syntaxe en langage C . . . . .  | 12        |
| <b>C</b> | <b>Les pointeurs + les structures</b>   | <b>14</b> |

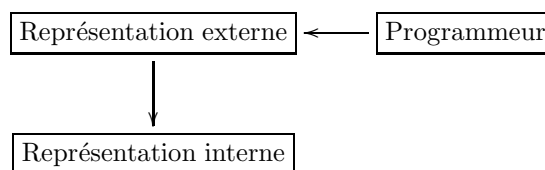
# 1 Introduction

Ici, on va travailler la manière de spécifier un type abstrait, en illustrant sur le TAD PILE, puis on étudiera les diverses implémentations possibles (statiques et dynamiques).

Dans une programmation “en large” il est nécessaire d’identifier et de spécifier les données assez tôt dans le processus de développement (en général, la programmation “en large” est réalisée par une équipe, par opposition à la programmation “en petit” qui est réalisée par une personne seule).

Une fois la donnée complexe identifiée, nous nous intéressons à sa spécification. Le but de cette spécification est de définir l’interface d’utilisation (**représentation externe**) de cette donnée et de lui donner une sémantique abstraite indépendante de l’implantation (**représentation interne**).

Les langages de programmation impératifs typés offrent des types dits prédéfinis. Ceci conduit à une utilisation naturelle et simple des variables définies à partir de ces types. Ces types sont souvent définis d’une façon abstraite puis implémentés dans le langage de programmation.



## Exemple des entiers

- Représentation externe des entiers vue par le programmeur (en C) :

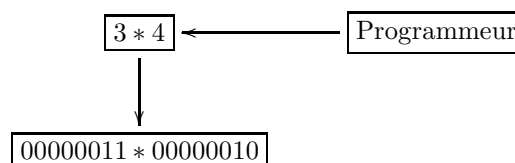
```
int          /* le nom du type */
5, -6, 21    /* des nombres */
+, -, *, /    /* les opérations permises */
```

Ceci représente l’interface des entiers. Cette représentation externe donne la possibilité d’une utilisation naturelle des entiers .

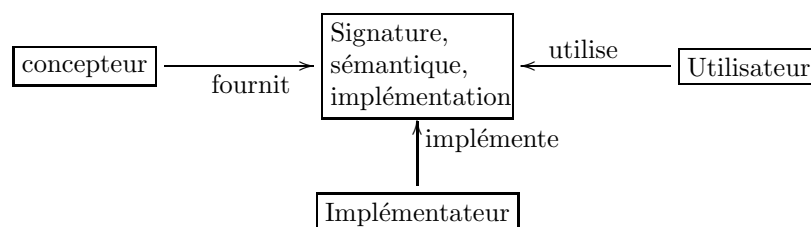
- Représentation interne des entiers : il s’agit d’une représentation binaire ; par exemple, l’entier 3 est codé par 00000011 sur un octet.

La représentation externe est incomparablement plus facile à utiliser qu’une représentation interne.

Exemple de la multiplication de deux entiers : il est plus facile d’écrire  $3 * 4$  que  $00000011 * 00000010$ . On aura donc :



On a donc le schéma suivant :



Dans ce cours, nous nous intéressons aux types abstraits de données les plus souvent utilisés. Nous proposons la classification suivante :

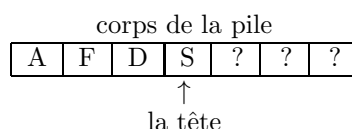
- Les types à structure linéaire : Liste, Pile, File
- Les types à structure arborescente : Arbre Binaire, Forêt
- Les Graphes

## 2 La spécification d'un TAD

Elle se fait en deux phases :

- on donne tout d'abord la *spécification fonctionnelle* appelée aussi *signature*, c'est-à-dire on décrit l'interface qui permettra d'utiliser ce TAD : donner un nom significatif au type, donner les profils des opérations du type, sachant qu'une opération est considérée comme une fonction.
- puis on donne la *sémantique abstraite* de ce TAD : pour cela nous utilisons le langage de la logique équationnelle. Ce travail se décompose en trois phases :
  - Nous commençons par partager les opérations du type en deux catégories : les **constructeurs** et les **opérateurs**. Un constructeur est indispensable à la représentation des valeurs du type et l'ensemble des constructeurs choisi doit être nécessaire et suffisant pour "construire" toutes les valeurs possibles du type.
  - Puis nous identifions les restrictions des opérations (constructeurs ou opérateurs) ; en effet, les opérations sont toutes considérées comme des fonctions totales ou partielles dont la restriction s'exprime par des **préconditions**.
  - Et enfin, nous caractérisons chaque opération par des **axiomes**. En général, ces axiomes sont construits par l'application d'un opérateur sur un constructeur si la précondition est satisfaite.

**Exemple des piles** Une pile est un type de données permettant de rassembler des données d'un type donné et auquel on accède par un seul point d'entrée (la tête) ; les piles sont donc gérées sur le principe du "dernier entré, premier sorti" (LIFO).



Les idées que l'on veut exploiter pour définir ce type PILE sont :

- La pile doit être vide à la création
- Une pile vide est construite par *Créer*
- Une pile non vide est construite par *Créer* suivie par une suite d'*Empiler*

*Créer* et *Empiler* seront les constructeurs ; elles sont indispensables pour représenter n'importe quel terme du type PILE (pile vide ou non).

Exemple : *Empiler (Empiler (Empiler (Empiler (créer, 'A'), 'F'), 'D'), 'S')*

### Spécification du TAD pile d'entiers

**Sorte :** PILE

**Utilise :** ENTIER, BOOLÉEN

**Opérateurs\_Constructeurs :**

*Créer* :  $\rightarrow$  PILE

*Empiler* :  $\text{PILE} \times \text{ENTIER} \rightarrow \text{PILE}$

**Opérateurs\_Projecteurs :**

*Appartient* :  $\text{PILE} \times \text{ENTIER} \rightarrow \text{BOOLÉEN}$

*Est\_Vide* :  $\text{PILE} \rightarrow \text{BOOLÉEN}$

*Dépiler* :  $\text{PILE} \rightarrow \text{PILE}$

*Sommet* :  $\text{PILE} \rightarrow \text{ENTIER}$

**Préconditions :**

$\text{Dépiler}(p), \text{Sommet}(p) \text{ ssi } \neg \text{Est\_Vide}(p)$

**Axiomes :**

$\text{Appartient}(\text{Créer}, x) = \text{faux}$

$\text{Appartient}(\text{Empiler}(p, x), y) = (x = y) \vee \text{Appartient}(p, y)$

$\text{Est\_Vide}(\text{Créer}) = \text{vrai}$

$\text{Est\_Vide}(\text{Empiler}(p, x)) = \text{faux}$

$Dépiler(Empiler(p,x)) = p$

$Sommet(Empiler(p,x)) = x$

### 3 Implémentation d'un TAD

L'implémentation d'un type abstrait consiste à :

- Implémenter une représentation interne du type
- Implémenter les fonctions
- Assurer la séparation entre l'implémentation de l'interface (signature) et le corps. Deux grands avantages :
  1. Permet de modifier le corps sans toucher à l'interface donc sans modifier les programmes utilisateurs
  2. manipuler par abstraction
- Assurer la protection de la représentation interne

#### 3.1 Implanter une représentation interne du type et implémenter les fonctions

Nous sommes souvent devant un choix entre une implémentation statique ou une implémentation dynamique :

**Statique** : signifie que la réservation des variables est effectuée au chargement du programme, et que lors de l'exécution ces variables ne changent pas de place par rapport à l'espace mémoire du programme. (cf. cours sur les tableaux – semestre précédent)

- Avantages : accès rapide
- Inconvénients : occupation inutile de l'espace mémoire pendant l'exécution

**Dynamique** : les réservations et libérations des variables s'effectuent en cours d'exécution. (cf. cours sur les pointeurs)

- Avantages : optimisation de l'espace mémoire occupé
- Inconvénients : l'efficacité est diminuée par le temps passé à la gestion de la mémoire (qui se fait au cours de l'exécution).

##### 3.1.1 Implémentation Statique du TAD PILE

Pour représenter la pile, nous avons besoin de représenter 2 informations : le corps de la pile (un tableau d'entier de taille  $N$ ) et l'indice de tête (un entier). Nous pouvons regrouper ces deux informations dans la même structure en C :

```
typedef struct {
    int indice;
    int t[N];} pile;
pile p;
```

L'accès au champ `indice` se fait par `p.indice` qui est utilisé comme une variable de type `int`. Par exemple, `p.indice = 5`; L'accès au corps se fait par `p.t` qui est utilisé comme un tableau de taille  $N$ . Par exemple, `p.t[p.indice] = 0`;

Attention : l'utilisation de l'instruction `assert` sera expliquée lorsqu'on parlera de la séparation corps/interface (section 3.2).

```
#include <stdio.h>
#define N 10;
```

```
typedef struct {
    int indice;                /* représente la dernière case remplie : la tête */
    int t[N];} pile;
```

```
void creer_pile (pile *p) {
    p->indice = -1; } /* la pile est créée vide */
```

```

void empiler (pile *p, int e) {
    assert(p->indice != N-1) ; /* précondition en liaison avec */
    /* l'implémentation statique correspondant à la pile est pleine */
    p->indice ++;
    p->t[p->indice] = e; }

void depiler (pile *p) {
    assert (!est_vide(*p));      /* précondition*/
    p->indice--; }

int est_vide (pile p) {
    return (p.indice == -1);}

int sommet (pile p) {
    assert (!est_vide(p)) ;      /* précondition*/
    return (p.t[p.indice]);}

```

assert(condition) est une operation permettant de sortir du programme quand la condition n'est pas vérifiée et si elle est vérifiée le sous-programme poursuit son exécution.

p->indice est une notation simplifiée correspondant à (\*p).indice.

Exemple d'utilisation de ce TAD :

```

pile P, Q;
int x;
créer(&P); créer (&Q); empiler(&P, 5); empiler(&Q,6); x=sommet(P);

```

### 3.1.2 Cours sur les pointeurs

Voir Annexe A.

### 3.1.3 Rappel sur les structures

Voir Annexe B.

### 3.1.4 Les pointeurs + les structures

Voir Annexe C.

### 3.1.5 Implémentation Dynamique du TAD PILE

```

typedef struct cel {
    int info;
    struct cel * suiv;
} cel ; /* cel = structure à 2 champs (entier, ptr sur cel) */

typedef struct cel * pile ; /* pile = pointeur sur une cel */
/* cel et struct cel sont des synonymes */
/* cel *, struct cel * et pile sont des synonymes */

void creer ( pile * p) {
    /* création d'une pile à vide */
    *p = NULL ; }

int est_vide (pile p) {
    /* vérifier si une pile est vide
    (renvoie 0 si non et une valeur différente de 0 si oui)*/
    return (p == NULL); }

```

```

int sommet (pile p) {
    /* récupérer la valeur en sommet d'une pile */
    assert (!est_vide(p)) ;
    return( p->info) ; }

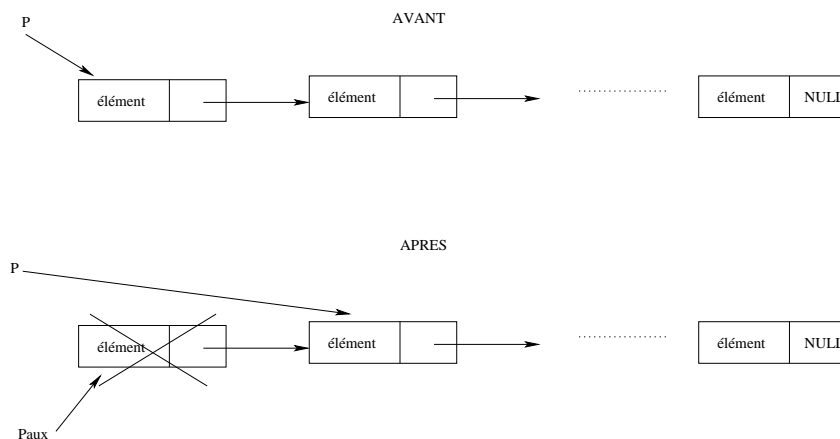
void depiler (pile * p) {
    /* supprime le sommet de pile */
    pile paux;
    assert (!est_vide(*p)) ;
    paux = *p ;
    *p = (*p).suiv ;
    free(paux);
}

void empiler (pile * p, int x) {
    /* rajoute une valeur à la pile */
    pile paux ;
    paux = *p ;
    *p = (pile)malloc(sizeof(struct cel));
    assert (!est_vide(*p)) ; /* plus de place mémoire => allocation échoue */
    (*p).info = x ;
    (*p).suiv = paux ;
}

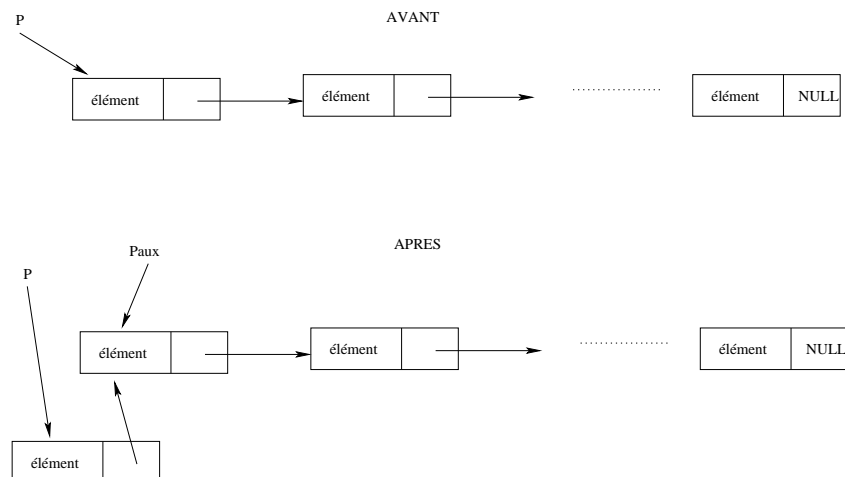
```

Le fonctionnement de `depiler` et de `empiler` est décrit par les figures suivantes

#### *Fonctionnement de "depiler"*



#### *Fonctionnement de "empiler"*





### 3.2 Séparer interface et corps

Un premier pas dans l'implémentation des données de type abstrait est de séparer l'interface de l'implémentation des fonctions. L'interface est constituée ici par la représentation du type et les profils des fonctions.

Dans un fichier qui représente l'interface `piles.h` (le "s" indiquant statique), nous implémentons l'interface; ce fichier est appelé un fichier "header" son nom contient l'extension `.h`.

```
#include <stdio.h>
#include <assert.h>

// les deux lignes suivantes permettent de se protéger contre
// les inclusions multiples
#ifndef __SYMBOLEUNIQUE__
#define __SYMBOLEUNIQUE__

#define N 10
typedef struct pile {
    int indice; /* représente la dernière case remplie : la tête */
    int t[N];} pile;

void creer (pile *);
void empiler (pile *, int );
void depiler (pile *);
int est_vide (pile );
int sommet (pile);

#endif // ferme le ifndef __SYMBOLEUNIQUE__
```

Dans un fichier qui contient l'implémentation du type `piles.c`, on implémente les corps des fonctions en incluant le fichier `piles.h` pour avoir accès au type `PILE`.

```
#include "piles.h"

void creer (pile *p) {
    p->indice = -1; } /*la pile est créée vide */
void empiler (pile *p, int e){
    assert (p->indice != N-1) ;
    p->indice ++;
    p->t[p->indice] = e; }
void depiler (pile *p) {
    assert (!est_vide(*p)) ;
    p->indice--;}
int est_vide (pile p) {
    return (p.indice == -1);}
int sommet (pile p) {
    assert (!est_vide(p)) ;
    return (p.t[p.indice]);}
```

L'utilisateur (programmeur), pour pouvoir manipuler des piles, n'a qu'à inclure `piles.h` dans son fichier source et lier son fichier objet (avec un suffixe `.o`) au `piles.o`. Par exemple, dans le fichier `testPile.c`, on trouvera :

```
#include "piles.h"

int main() {
    pile P, Q;
    int x;
    créer(&P); créer (&Q); empiler(&P, 5); empiler(&Q,6); x=sommet(P);}
```

### 3.3 Protection du type

L'implémentation précédente permet au programmeur utilisateur de manipuler explicitement la structure du type. Par contre rien interdit à un programmeur d'écrire dans son code :

```
p.indice = 5;
```

alors que la valeur d'indice précédente était, par exemple, 1. Ceci casse la définition de la pile.

Pour protéger cette structure interne nous avons intérêt à la cacher dans le fichier `piles.c`. Le problème est de pouvoir donner la possibilité au programmeur de faire respecter la définition des piles. La structure interne sera dans `piles.c`. L'accès à cette structure protégée se fera à travers un type pointeur implémenté dans `piles.h`

Dans notre cas, cela donnera :

```
#include <stdio.h>
typedef struct pile *pile;

void creer (pile *);
void empiler (pile *, int );
void depiler (pile *);
int est_vide (pile);
int sommet (pile);
```

Dans le fichier `piles.c`, on inclut le fichier `piles.h` puis on implémente la structure et le corps des fonctions :

```
#include "pile.h"
#define N 10
struct pile {
    int indice; /* représente la dernière case remplie : la tête */
    int t[N];
};

void creer (pile *p) {
    p->indice = -1; }
void empiler (pile *p, int e){
    assert (p->indice != N-1) ;
    p->indice ++;
    p->t[p->indice] = e; }
void depiler (pile *p) {
    assert (!est_vide(*p)) ;
    p->indice--;}
int est_vide (pile p) {
    return (p->indice == -1);}
int sommet (pile p) {
    assert (!est_vide(p)) ;
    return (p->t[p->indice]);}
```

En fournissant seulement le fichier `piles.h` aux utilisateurs, on leur interdit de toucher à la structure interne du type et on la protège. Et cela ne doit rien changer au fichier `testPile.c` qui utilise la pile :

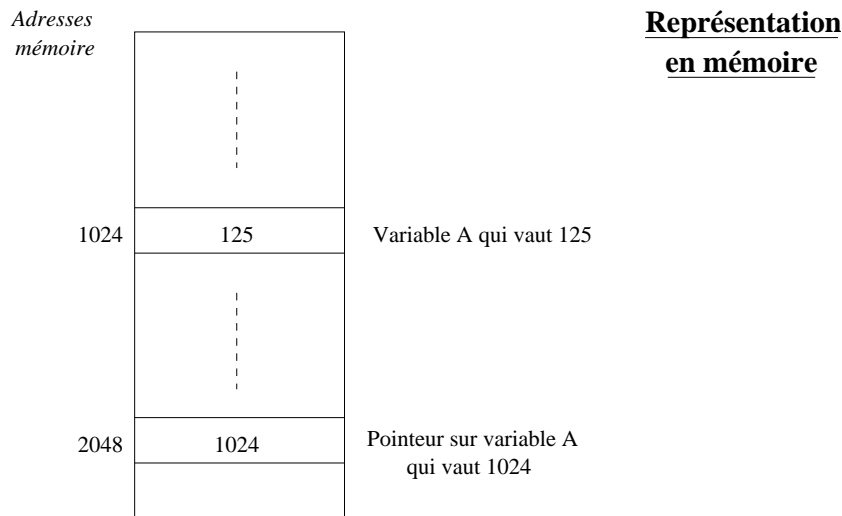
```
#include "piles.h"

int main() {
    pile P, Q;
    int x;
    créer(&P); créer (&Q); empiler(&P, 5); empiler(&Q,6); x=sommet(P);}
```

## A Cours sur les pointeurs

### A.1 Définition

Un *pointeur* = variable dont la valeur correspond à une adresse.



### A.2 Syntaxe en langage C

Les pointeurs correspondent à un type spécial : le type *pointeur*.

- en zone déclarative, on écrit : *type\_que\_l'on\_veut \* p* ;  
par exemple, si on écrit dans le programme :

*int \* p*

il y a alors réservation mémoire dans le programme d'une case qui peut contenir l'adresse d'un entier, cette case est repérée par l'identificateur *p* et sa valeur au moment de la déclaration est non significative (exactement comme lorsqu'on déclare *int n*, il y a réservation mémoire dans le programme d'une case qui peut contenir un entier, cette case étant repérée par l'identificateur *n* et sa valeur au moment de la déclaration étant non significative) ;

- dans le corps du programme, on utilise
  - soit *p* pour manipuler l'adresse,
  - soit *\*p* pour manipuler la valeur pointée (le symbole *\** est appelé un constructeur et il sert aussi d'opérateur d'indirection).

**Remarque :** on a donc utilisé (sans le dire !) des pointeurs pour “simuler” le passage de paramètres par référence en langage C.

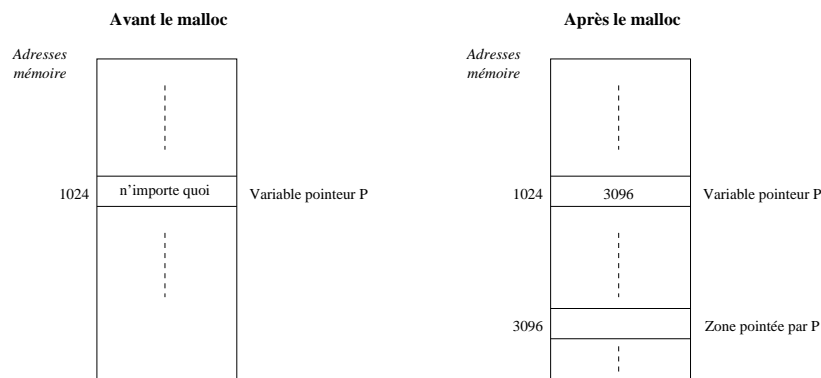
**Opérations autorisées** En langage C, il existe 6 opérations possibles sur les pointeurs :

- Les 4 premières ont été étudiées au semestre 1 :
  - L'affectation :
    - $p = q$  ; (*p* et *q* doivent être des pointeurs sur un même type),
    - $p = NULL$  ; (*NULL* est l'élément nul pour le type pointeur),
    - $p = \&x$  ; (*x* est une variable de type quelconque noté T et *p* est une variable de type pointeur sur le type T) (le symbole *&* sert à obtenir l'adresse d'une variable, c'est l'opérateur d'adressage).
  - L'addition et la soustraction d'un pointeur avec un entier (on parle aussi de *décalage*) :
    - $p = q + 1$  ; (*p* et *q* doivent être des pointeurs sur un même type et sur une même entité mémoire<sup>1</sup>),
    - $p = q - 10$  ; (*p* et *q* doivent être des pointeurs sur un même type et sur une même entité mémoire).
  - La soustraction de 2 pointeurs de même type :
    - $i = p - q$  ; (*p* et *q* doivent être des pointeurs sur un même type et sur une même entité mémoire et *i* est un entier).

---

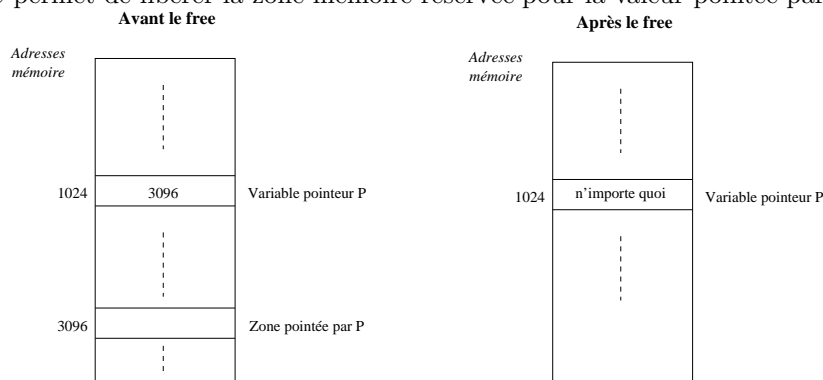
1. C'est-à-dire sur un même tableau.

- La comparaison de 2 pointeurs de même type :  
 $p > q$  ( $p$  et  $q$  doivent être des pointeurs sur un même type et sur une même entité mémoire).
- Les 2 dernières sont liées à la gestion dynamique de la mémoire :
  - La création dynamique (réservation et mise à jour) :  $p = (\text{type\_de\_p}) \text{ malloc}(\text{taille\_zone\_pointée})$  avec  $p$  une variable de type pointeur.  
 Cette fonction C permet de réserver une zone mémoire pour la valeur pointée par  $p$  et met à jour le pointeur  $p$  avec l'adresse de cette zone.



Remarque : si, après l'exécution de cette instruction, le pointeur résultat est égal à NULL, cela signifie que la mémoire est saturée et qu'on ne peut plus faire d'allocation dynamique.

- La suppression dynamique :  $\text{free}(p)$  avec  $p$  une variable de type pointeur.  
 Cette fonction C permet de libérer la zone mémoire réservée pour la valeur pointée par  $p$ .



Attention : il arrive qu'après un  $\text{free}(p)$ , on puisse encore accéder à l'ancienne zone pointée par  $p$  (cela dépend de la machine et de son gestionnaire mémoire), c'est donc une source importante d'erreur ; un conseil : après  $\text{free}(p)$ , faire  $p = \text{NULL}$ .

**Avantages** Gestion dynamique de la mémoire et souplesse d'implémentation.

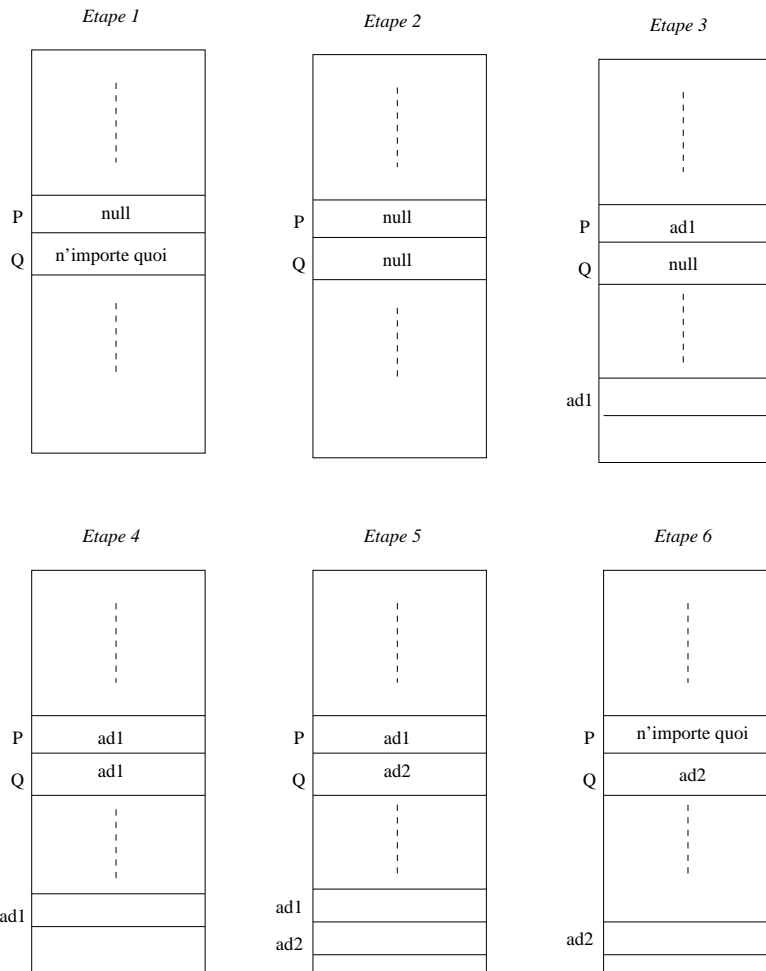
### A.3 Exemples sur les pointeurs

1. Soit le programme suivant :

```
...
#include <stdio.h>
...
int * P, * Q;
...
P = NULL;                               /* étape 1 */
Q = P;                                   /* étape 2 */
P = (int *) malloc(sizeof(int));         /* étape 3 */
Q = P;                                   /* étape 4 */
Q = (int *) malloc(sizeof(int));         /* étape 5 */
free (P);                                /* étape 6 */
...
```

Chaque étape est représentée par un schéma de la mémoire, et le déroulement du programme donne la

chose suivante :



2. Soit le programme suivant :

```
int *p, *q;
p = (int *) malloc(sizeof(int));
printf("Après malloc : %d \n", *p);
q = p;
printf("Après q = p : %d et %d \n", *p, *q);
*p = 15;
printf("Après maj : %d et %d \n", *p, *q);
free(p);
printf("Après free : %d \n", *q);
```

Ici, le résultat obtenu à l'impression est :

```
Après malloc : Valeur1_quelconque
Après q = p : Valeur1_quelconque et Valeur1_quelconque
Après maj : 15 et 15
Après free : 15          /* ATTENTION : erreur possible (dépend de la machine!) */
                        /* En effet, q pointe sur une zone mémoire qui a été libérée! */
```

3. Soit le programme suivant :

```
/* définition de nouveaux types */
typedef int zone;
typedef zone * ptrzone;
void maj_zone_pointee(ptrzone p, zone val) {
    *p = val;
}
```

/\* zone sera le type int \*/  
/\* ptrzone sera le type pointeur sur zone \*/

```
main() {
    ptrzone p;
    p = (int *) malloc(sizeof(int));
    printf("Avant maj : %d\n", *p);
    maj_zone_pointee(p, 10);
    printf("Après maj : %d\n", *p);
}
```

Ici, le résultat obtenu à l'impression est :

```
Avant maj : Valeur_quelconque
Après maj : 10
```

Remarque : cette fonction permet de modifier une zone par l'intermédiaire de son adresse. C'est ainsi que l'on procède dans les langages où seul le passage par valeur est autorisé pour modifier des valeurs lors de l'appel d'un sous-programme (ex : le langage C).

## A.4 Conclusion sur les pointeurs

Les pointeurs sont des variables dont la valeur est une adresse. Ils servent à faire de l'allocation dynamique de mémoire et à "simuler" un passage de paramètre par référence.

# B Rappels sur les structures

## B.1 Définition

Il s'agit d'un type de données composées permettant donc de "regrouper" des valeurs. Par contre, à la différence des **tableaux** qui regroupent des données de **même type**, une **structure** permet de regrouper des données de types **différents**.

## B.2 Syntaxe en langage C

Il s'agit de définir un modèle de données que l'on pourra utiliser ensuite comme type pour définir des variables.

```
struct nom_modele {
    type1 champ1;
    type2 champ2;
    ...
    typeN champN;
} ;
```

Chaque élément ou *champ de la structure* est défini par son type et son nom.

Exemple :

```
/* définition d'un modèle de donnée regroupant 2 */
/* entiers et une chaîne de caractères          */

struct Date {
    int jour;
    char mois[10];
    int annee;
};
```

La combinaison du mot-clé **struct** et du nom donné au modèle est utilisée ensuite pour définir des variables, à moins qu'un type spécifique ait été défini au moyen de l'opérateur **typedef**.

**Utilisation de l'opérateur typedef** Il s'agit de la possibilité de renommer un type puisque celui-ci doit être décrit au moment de la définition.

```
typedef description_type nom_type;
```

Exemple :

```

/* définition d'un type nommé type_date et */
/* correspondant au modèle de donnée      */
/* struct Date                             */

typedef struct Date {
    int jour;
    char mois[10];
    int annee;
}      type_date;

```

Le nom du type défini peut alors être utilisé pour la définition de variables.

Remarque : L'opérateur **typedef** fréquemment utilisé avec les structures peut également s'utiliser pour définir d'autres types de données.

**Définition d'une variable correspondant à une structure** Ceci n'est possible qu'une fois que le modèle ou le type de la structure a été défini. La définition d'une variable entraîne l'allocation mémoire de la zone nécessaire pour stocker les données correspondantes.

```

/* définition d'une variable de nom Jour_J et contenant*/
/* 3 champs : jour, mois et annee                      */
struct Date Jour_J;

```

ou

```

type_date Jour_J;
/* dans la mesure où le type type_date a été défini */
/* comme ci-dessus                                  */

```

**Initialisation d'une variable correspondant à une structure** Comme pour les tableaux, il faut spécifier la liste des valeurs de chaque élément.

```
struct nom_modele nom_var = {val_chp1, ..., val_chpN};
```

Exemple :

```

struct Date Noel = {25, "décembre", 2000 };
type_date Nouvel_an = {1, "janvier", 2001};

```

**Utilisation d'une structure dans le corps d'un programme** En dehors de l'initialisation faite au moment de la définition de la donnée, on peut :

- accéder à chaque champ individuellement en utilisant la notation pointée (on pourra alors utiliser ce champ comme n'importe quelle variable du même type) :

```
nom_variable.nom_champ
```

Exemple :

```

Jour_J.jour= 14;
strcpy(Jour_J.mois, "juillet");
Jour_J.annee = 2000;

```

```

/* pour attribuer une valeur à une chaîne de caractères */
/* (=tableau de caractères) il faut utiliser la          */
/* fonction strcpy définie dans la bibliothèque string.h */

```

- affecter une valeur à une structure :

Contrairement au tableau qu'il faut traiter élément par élément, il est possible d'affecter la valeur d'une structure à une autre structure.

Exemple :

```

type_date Autre_jour;
...
Autre_jour = Jour_J;

```

**Remarque : imbrication de structures** Une structure peut admettre un champ qui soit lui même une structure. Seul cas impossible, lorsque le champ doit être du type de la structure en cours de définition. Dans ce cas particulier, celui des structures récursives, il faut passer par un champ qui soit un pointeur sur une donnée du type en cours de définition.

Exemple :

```
/* définition d'un autre type de structure */
struct Personne
{
    char nom[20];
    char prenom[20];
    struct Date ne_le;
};

/* définition et initialisation d'une variable */
/* comportant une structure imbriquée */
struct Personne P1 = {"Durand", "Paul", {12, "mai", 1980}};

/* exemple d'accès au champ d'une structure imbriquée */
if(P1.ne_le.annee > 1970)
    printf("%s %s a moins de 30 ans", P1.prenom, P1.nom);
```

**Les structures auto-référentielles** Le seul moyen d'avoir une structure récursive c'est-à-dire qui admet au moins un champ correspondant à une donnée du même type que la structure en cours de définition est d'utiliser un pointeur sur le type en question.

Exemple :

```
struct membre_famille          /* erreur */
{
    char nom[20];
    char prenom[20];
    int age;
    struct membre_famille pere, mere; } ;

/* n'est pas possible car un type tel que */
/* struct membre_famille ne peut être utilisé */
/* qu'une fois complètement défini */
/* il faut donc utiliser un pointeur */

struct membre_famille          /* correct */
{
    char nom[20];
    char prenom[20];
    int age;
    struct membre_famille *ptr_pere, *ptr_mere; } ;
```

Ce type de structure est utilisé comme élément de base d'une structure plus complexe où des données de même type sont liées les unes aux autres.

## C Les pointeurs + les structures

On peut définir un pointeur sur des données de tout type y compris sur des structures et mêmes sur des pointeurs.

Exemple, en utilisant les structures Date et Personne :

```
struct Date *ptr_date ;
/* allocation des 4 octets nécessaires pour */
/* stocker une adresse. Tel qu'il est le */
/* pointeur ptr_date ne pointe sur aucune */
/* donnée valide */
```



```

struct Personne * ptr_pers;
/* allocation des 4 octets nécessaires pour */
/* stocker une adresse. Tel qu'il est le */
/* pointeur ptr_pers ne pointe sur aucune */
/* donnée valide */

struct Date une_date = {14, "Juillet", 2000};
struct Personne une_personne= {"Pierre", "Dubois", {12, "Avril", 1980}};
/* définition et allocation des octets nécessaires */
/* pour stocker une première donnée correspondant à */
/* une structure date et une autre correspondant à */
/* une structure personne */

```

**Accès au champ d'une structure via un pointeur** Si le type de donnée associé à un pointeur (= type de la donnée pointée) est une structure, et que bien sûr le pointeur a reçu l'adresse d'une variable du type de données en question alors le pointeur permet d'accéder indirectement aux champs de la structure pointée. Deux notations sont possibles et totalement équivalentes :

**nom\_ptr->nom\_champ** La flèche représentée par la combinaison des symboles - (moins) et > (supérieur) indique que l'on accède indirectement au champ spécifié de la donnée pointée par le pointeur.

**(\*ptr\_nom).nom\_champ** La notation pointée ne peut être utilisée que si la partie gauche correspond à une variable de type structure. Pour obtenir cela à partir d'un pointeur, il faut d'abord appliquer l'opérateur d'indirection \* au pointeur ; le parenthésage est obligatoire. On obtient ainsi la donnée pointée. On peut donc ensuite utiliser la notation pointée pour accéder au champ recherché.

Exemple :

```

ptr_date = &une_date ;
/* le pointeur re,çoit l'adresse d'une donnée */
/* ptr_date pointe sur la donnée une_date */

/* affichage de la valeur de la donnée une_date de 3 fa,cons */
/* différentes */
printf(" une date = %d %s %d", une_date.jour, une_date.mois, une_date.annee);
printf(" même chose = %d %s %d",ptr_date->jour, ptr_date->mois, ptr_date->annee);
printf(" encore la même chose = %d %s %d", (*ptr_date).jour, (*ptr_date).mois,
      (*ptr_date).annee);

ptr_pers = &une_personne;
/* le pointeur re,çoit l'adresse d'une donnée */
/* ptr_pers pointe sur la donnée une_personne */

/* affichage de la date de naissance de la donnée */
/* une_personne de 3 fa,cons différentes */
printf(" une date = %d %s %d", une_personne.ne_le.jour, une_personne.ne_le.mois,
      une_personne.ne_le.annee);
printf(" même chose = %d %s %d", ptr_pers->ne_le.jour, ptr_pers->ne_le.mois,
      ptr_pers->ne_le.annee);
printf(" encore la même chose = %d %s %d", (*ptr_pers).ne_le.jour, (*ptr_pers).ne_le.mois,
      (*ptr_pers).ne_le.annee);

```

## Exemples sur les pointeurs + les structures

1. Soit le programme suivant :

```

typedef struct et_cellule {
    int valeur;
    int * suivant;
} cellule; /* cellule est une structure à 2 champs (entier, pointeur sur entier) */

```

```

/* cellule et struct et_cellule sont des synonymes */
main()
{
    cellule c;
    c.valeur = 10;
    c.suivant = (int *) malloc(sizeof(int));
    *(c.suivant) = 11;
    printf( "%d %x %d ", c.valeur, c.suivant, *(c.suivant));
}

```

Ici, le résultat obtenu à l'impression est :

10 Adresse 11

2. Soit le programme suivant :

```

typedef struct et_cellule {
    int valeur;
    struct et_cellule * suivant;
} cellule; /* cellule est une structure à 2 champs (entier, pointeur sur cellule) */
typedef cellule * ptrcellule; /* ptrcellule est un pointeur sur une cellule */
/* cellule et struct et_cellule sont des synonymes */
/* cellule *, struct et_cellule * et ptrcellule sont des synonymes */

```

```

main()
{
    ptrcellule p;
    p = (ptrcellule) malloc(sizeof(cellule));
    p->valeur = 10;
    p->suivant = (ptrcellule) malloc(sizeof(cellule));
    p->suivant->valeur = 11;
    p->suivant->suivant = (ptrcellule) malloc(sizeof(cellule));
    p->suivant->suivant->valeur = 12;
    printf( "%x, %d, %x, %d, %x, %d, %x, %d ",
           p,
           p->valeur,
           p->suivant,
           p->suivant->valeur,
           p->suivant->suivant,
           p->suivant->suivant->valeur,
           p->suivant->suivant->suivant,
           p->suivant->suivant->suivant->valeur);
}

```

On n'a pas fait de malloc pour `p->suivant->suivant->suivant`. On a donc deux cas possibles :

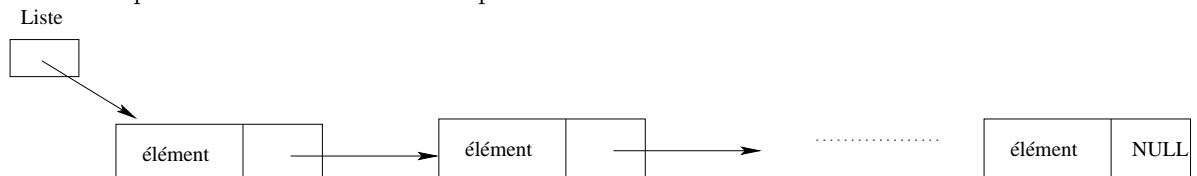
- soit le programme “plante”, car `p->suivant->suivant->suivant` contient une adresse interdite au programme et alors l'accès à `p->suivant->suivant->suivant->valeur` provoquera une erreur ;
- soit le programme s'exécutera sans erreur mais les deux dernières valeurs affichées sont non significatives. Le résultat obtenu à l'impression sera donc :

Adresse0 10 Adresse1 11 Adresse2 12 ? ?

3. Représentation de listes (structures de données linéaires) en dynamique.

On a au moins 4 cas possibles de listes :

- la liste simplement chaînée avec un seul point d'accès :



Cela correspondant au type suivant en langage C :

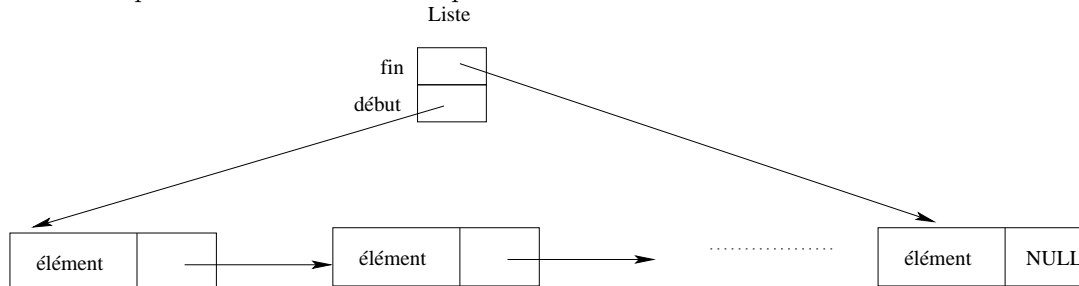
```

typedef struct et_cel_liste_simple {
    int info;
    struct et_cel_liste_simple * suiv;
} cel_liste_simple;
/* cel_liste_simple = structure à 2 champs (entier, pointeur sur cel_liste_simple) */

```

```
typedef struct cel_liste_simple * liste_simple_1_pt;
/* liste_simple_1_pt = pointeur sur une cel_liste_simple */
/* cel_liste_simple et struct et_cel_liste_simple sont des synonymes */
/* cel_liste_simple *, struct cel_liste_simple * et liste_simple_1_pt sont des synonymes */
```

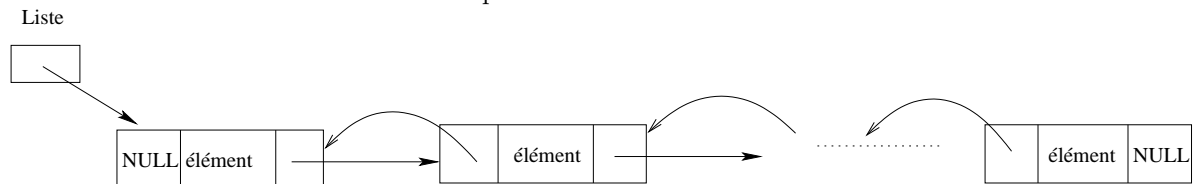
- la liste simplement chaînée avec deux points d'accès :



Cela correspondant au type suivant en langage C qui réutilise le type de la cellule simplement chaînée :

```
typedef struct et_liste_simple_2_pt {
    cel_liste_simple * premier;
    cel_liste_simple * dernier;
} liste_simple_2_pt;
/* liste_simple_2_pt = structure à 2 champs : 2 pointeurs sur cel_liste_simple */
/* liste_simple_2_pt et struct et_liste_simple_2_pt sont des synonymes */
```

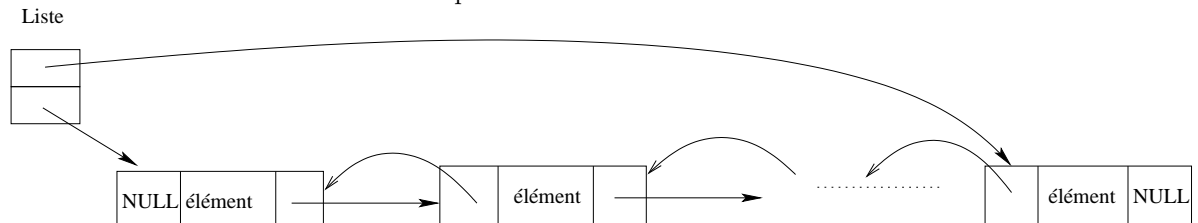
- la liste doublement chaînée avec un seul point d'accès :



Cela correspondant au type suivant en langage C :

```
typedef struct et_cel_liste_double {
    int info;
    struct et_cel_liste_double * suiv;
    struct et_cel_liste_double * prec;
} cel_liste_double;
/* cel_liste_double = structure à 3 champs (entier, 2 pointeurs sur cel_liste_double) */
typedef struct cel_liste_double * liste_double_1_pt;
/* liste_double_1_pt = pointeur sur une cel_liste_double */
/* cel_liste_double et struct et_cel_liste_double sont des synonymes */
/* cel_liste_double *, struct cel_liste_double * et liste_double_1_pt sont des synonymes */
```

- la liste doublement chaînée avec deux points d'accès :



Cela correspondant au type suivant en langage C qui réutilise le type de la cellule doublement chaînée :

```
typedef struct et_liste_double_2_pt {
    cel_liste_double * premier;
    cel_liste_double * dernier;
} liste_double_2_pt;
/* liste_double_2_pt = structure à 2 champs : 2 pointeurs sur cel_liste_double */
/* liste_double_2_pt et struct et_liste_double_2_pt sont des synonymes */
```

# Le TAD FILE et ses implémentations

7 février 2013

On veut implémenter de manière statique puis dynamique le type abstrait FILE permettant de gérer des files d'éléments. On utilisera ce TAD dans une application permettant la fusion de voies routières.

Pour l'implémentation statique, on utilisera une structure contenant un tableau et deux entiers (un pour le nombre d'éléments dans la file et l'autre pour indiquer la tête de file – on peut aussi ajouter un entier pour indiquer la queue de la file).

Pour l'implémentation dynamique, on utilisera une structure contenant deux pointeurs, un sur la tête de la file et l'autre sur la queue de la file.

La signature complète du type abstrait FILE est la suivante (voir dans le header une autre version possible de ce TAD – en commentaire pour chaque fonction) :

```
Sorte : FILE
Utilise : BOOLÉEN, NAT
Opérateurs_Constructeurs :
  CreerFile :  $\rightarrow$  FILE
  Enfiler : FILE  $\times$  ÉLÉMENT  $\rightarrow$  FILE
Opérateurs_Projecteurs :
  Appartient : FILE  $\times$  ÉLÉMENT  $\rightarrow$  BOOLÉEN
  EstVide : FILE  $\rightarrow$  BOOLÉEN
  Defiler : FILE  $\rightarrow$  FILE
  TailleFile : FILE  $\rightarrow$  NAT
  Position1 : FILE  $\times$  ÉLÉMENT  $\rightarrow$  NAT
  TêteFile : FILE  $\rightarrow$  ÉLÉMENT
  QueueFile : FILE  $\rightarrow$  ÉLÉMENT
  Concat : FILE  $\times$  FILE  $\rightarrow$  FILE
  Mixage : FILE  $\times$  FILE  $\rightarrow$  FILE

Pré-conditions :
  TêteFile( $f$ ) :  $\neg$ EstVide( $f$ )
  QueueFile( $f$ ) :  $\neg$ EstVide( $f$ )
Axiomes :
  Appartient(CreerFile(),  $x$ ) = faux
  Appartient(Enfiler( $f$ ,  $x$ ),  $y$ ) = ( $x = y$ )  $\vee$  Appartient( $f$ ,  $y$ )

  EstVide(CreerFile()) = vrai
  EstVide(Enfiler( $f$ ,  $x$ )) = faux

  TailleFile(CreerFile()) = 0
  TailleFile(Enfiler( $f$ ,  $x$ )) = TailleFile( $f$ ) + 1

  Position1(CreerFile(),  $x$ ) = 0
  Position1(Enfiler( $f$ ,  $x$ ),  $y$ ) =
    si Appartient ( $f$ ,  $y$ ) alors Position1( $f$ ,  $y$ )
    sinon si  $x = y$  alors TailleFile( $f$ ) + 1
    sinon 0

  Defiler(CreerFile()) = CreerFile()
  Defiler(Enfiler( $f$ ,  $x$ )) =
    si TailleFile( $f$ ) > 1 alors Enfiler(Defiler( $f$ ),  $x$ )
    sinon si TailleFile( $f$ ) = 1 alors Enfiler(CreerFile(),  $x$ )
    sinon CreerFile() /* c'est le cas où TailleFile ( $f$ ) = 0
```

$TêteFile(Enfiler(f,x)) =$   
     *si*  $TailleFile(f) \neq 0$  *alors*  $TêteFile(f)$   
     *sinon*  $x$

$QueueFile(Enfiler(f,x)) = x$

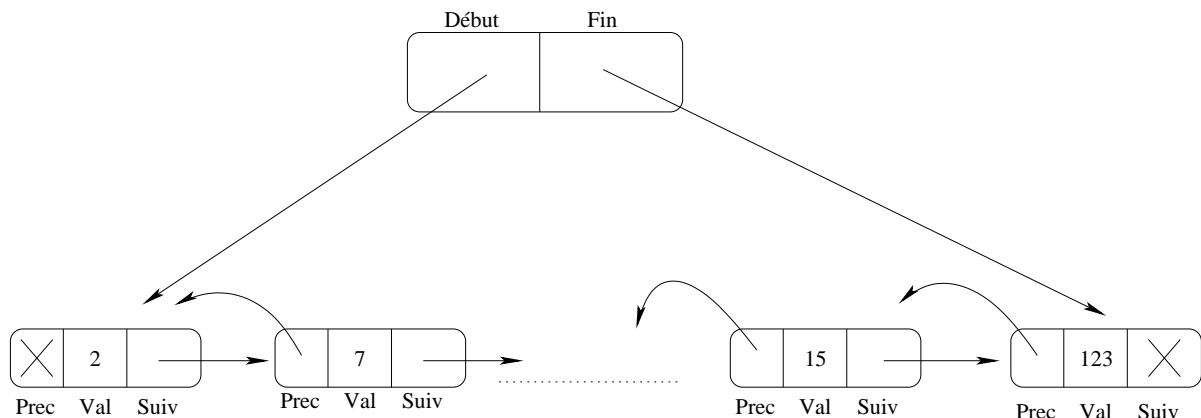
$Concat(CreerFile(), CreerFile()) = CreerFile()$   
 $Concat(CreerFile(), Enfiler(f,x)) = Enfiler(f,x)$   
 $Concat(Enfiler(f,x), CreerFile()) = Enfiler(f,x)$   
 $Concat(Enfiler(f_1,x_1), Enfiler(f_2,x_2)) =$   
      $Concat(Enfiler(Enfiler(f_1, x_1), TêteFile(Enfiler(f_2,x_2))),$   
      $Defiler(Enfiler(f_2,x_2)))$

$Mixage(CreerFile(), CreerFile()) = CreerFile()$   
 $Mixage(CreerFile(), Enfiler(f,x)) = Enfiler(f,x)$   
 $Mixage(Enfiler(f,x), CreerFile()) = Enfiler(f,x)$   
 $Mixage(Enfiler(f_1,x_1), Enfiler(f_2,x_2)) =$   
      $Concat(Enfiler(Enfiler(CreerFile(), TêteFile(Enfiler(f_1, x_1))), TêteFile(Enfiler(f_2,x_2))),$   
      $Mixage(Defiler(Enfiler(f_1,x_1)), Defiler(Enfiler(f_2,x_2))))$

# Liste doublement chaînée en dynamique

## 1 Contexte

Il s'agit d'implanter la structure de données LDCD (Liste\_Doublement\_Chainée\_Dynamique) d'entiers décrite par le schéma suivant :



**Les valeurs sont rangées par ordre croissant**

Cette structure de données permet de stocker des entiers par ordre croissant et de les afficher soit par ordre croissant, soit par ordre décroissant.

Vous devrez utiliser le mécanisme de la compilation séparée.

## 2 Questions à résoudre

1. Proposer en langage C un type de données dynamique pour une LDCD d'entiers.
2. Écrire la fonction `INIT_LDCD` qui retourne la LDCD vide.
3. Écrire la fonction `AFFICHER_CROISSANT_LDCD` qui affiche par ordre croissant tous les entiers stockés dans la LDCD donnée en paramètre.
4. Écrire la fonction `AFFICHER_DECROISSANT_LDCD` qui affiche par ordre décroissant tous les entiers stockés dans la LDCD donnée en paramètre.
5. Écrire la fonction `AJOUTER_A_LDCD` qui renvoie la LDCD construite en ajoutant l'entier donné en paramètre dans la LDCD donnée en paramètre. L'entier doit être ajouté à la "bonne place", c'est-à-dire en respectant l'ordre.
6. Écrire la fonction `SUPPRIMER_A_LDCD` qui renvoie la LDCD construite en supprimant l'entier donné en paramètre de la LDCD donnée en paramètre. Si l'entier n'appartient pas à la LDCD, celle-ci sera renvoyée sans modification
7. Écrire la fonction `MAP` qui prend en paramètre :
  - une fonction  $f$  qui prend un entier en paramètre et renvoie un autre entier,
  - et une LDCD,et qui renvoie le résultat de l'application de  $f$  à chaque élément de la LDCD. Ce résultat est donc forcément sous la forme d'une liste. Vous considérerez que la fonction  $f$  est accessible par un pointeur de fonction.

### 3 Le TAD

**Sorte** LDCD

**Utilise** BOOL, INT, FUNC : INT  $\rightarrow$  INT

#### Constructeurs

init :  $\rightarrow$  LDCD

ajout : LDCD  $\times$  INT  $\rightarrow$  LDCD

#### Projecteurs

affCroissant : LDCD  $\rightarrow$

affDecroissant : LDCD  $\rightarrow$

taille : LDCD  $\rightarrow$  INT

appartient : LDCD  $\times$  INT  $\rightarrow$  BOOL

index : LDCD  $\times$  INT  $\rightarrow$  INT

valeur : LDCD  $\times$  INT  $\rightarrow$  INT

nbOccurrences : LDCD  $\times$  INT  $\rightarrow$  INT

supprimer : LDCD  $\times$  INT  $\rightarrow$  LDCD

map : LDCD  $\times$  FUNC  $\rightarrow$  LDCD

#### Préconditions

valeur(l,i) ssi  $1 \leq i \leq \text{taille}(l)$

map(l,f) ssi f est monotone (conserve l'ordre)

#### Axiomes

taille(init()) = 0

taille(ajout(l,x)) = taille(l)+1

appartient(init(),x) = false

appartient(ajout(l,x),y) = (x=y)  $\vee$  appartient(l,y)

(index(l,x) = i)  $\wedge$

( $\neg$  appartient(l,x)  $\rightarrow$  (i=0))  $\wedge$

(appartient(l,x)  $\rightarrow$

(( $1 \leq i \leq \text{taille}(l)$ )  $\wedge$

( $\forall j = 1 \dots i - 1$  (valeur(l,j) < valeur(l,i)))  $\wedge$

( $\forall j = i + 1 \dots \text{taille}(l)$  (valeur(l,j)  $\geq$  valeur(l,i)))

(valeur(l,i) = x)  $\wedge$  (appartient(l,x))  $\wedge$  ( $\forall j = 1 \dots i - 1$  (valeur(l,j)  $\leq$  x))  $\wedge$  ( $\forall j = i + 1 \dots \text{taille}(l)$  (valeur(l,j)  $\geq$  x))

nbOccurrences(init(),x) = 0

(nbOccurrences(ajout(l,x),y) = n)  $\wedge$  (x=y  $\rightarrow$  n=nbOccurrences(l,y)+1)  $\wedge$  (x $\neq$ y  $\rightarrow$  n=nbOccurrences(l,y))

(supprimer(l,x) = l')  $\wedge$

(appartient(l,x)  $\rightarrow$

((taille(l')=taille(l)-1)  $\wedge$  (nbOccurrences(l',x)=nbOccurrences(l,x)-1)))

$\wedge$  ( $\neg$  appartient(l,x)  $\rightarrow$  l=l')

(map(l,f) = l')  $\wedge$  (taille(l) = taille(l'))  $\wedge$  ( $\forall i = 1 \dots \text{taille}(l)$  (valeur(l',i) = f(valeur(l,i))))

Remarque importante : pas d'axiome pour les projecteurs d'affichage puisqu'on n'a pas de "sortie" à caractériser.

# Itérateur pour Liste doublement chaînée en dynamique

## 1 Contexte Général

Quand on cherche à parcourir une suite d'éléments, on utilise la plus part du temps des *itérateurs*. Il s'agit de données pour lesquelles on peut donner :

- une valeur de début,
- une valeur de fin,
- une manière de passer à la valeur suivante.

Cela se voit sans difficulté sur le code très simple suivant :

```
for (i = 0 ; i<N; i++)
{
  ...
}
```

ou bien :

```
for (i = N-1 ; i>=0; i--)
{
  ...
}
```

A chaque fois, la variable  $i$  sert d'itérateur avec :

- comme valeur de début soit 0, soit  $N - 1$ ,
- comme valeur de fin soit  $N - 1$ , soit 0,
- comme passage au suivant soit une incrémentation, soit une décrémentation.

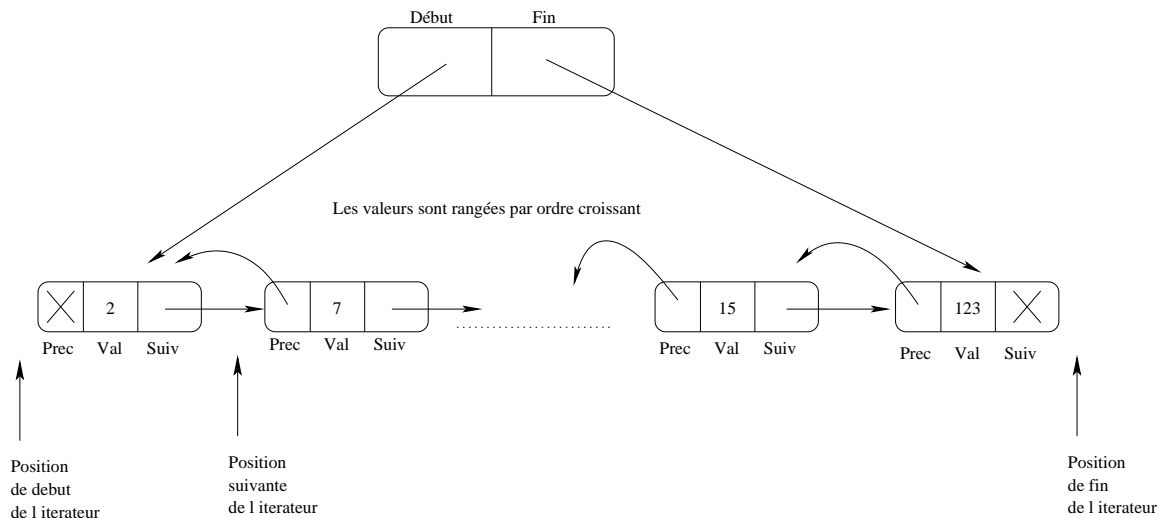
Ce mécanisme peut se généraliser de la manière suivante (en pseudo-code objet) :

```
i = creerIterateur(CROISSANT) ;
for (i.debut(i) ; ! i.fin(i); i.suivant(i))
{
  ...
}
```

## 2 Utilisation sur une LDCD

Il s'agit de définir une structure de données ITERATEUR permettant le parcours des LDCD d'entiers définies au TD précédent. On aura donc le schéma suivant :





Un itérateur sur une LDCD peut donc être vu comme un *curseur* qui se déplace dans un sens ou dans l'autre le long de la liste. On considère en général que le curseur est “entre” deux éléments (voir schéma ci-dessus). Au départ, ce curseur est situé juste avant le premier élément de la liste de façon à ce que l'appel de la fonction `next` permette de récupérer ce premier élément. De la même façon, si le curseur est en fin de liste, il est situé après le dernier élément et l'appel de la fonction `previous` permet de récupérer ce dernier élément. Cette structure de données permet ainsi de parcourir soit par ordre croissant, soit par ordre décroissant n'importe quelle LDCD.

### 3 Questions à résoudre

1. Proposer en langage C un type de données dynamique pour un `ITERATEUR` sur une LDCD d'entiers.
2. Écrire une fonction qui crée un itérateur pour une `ldcd` donnée sans préciser pour l'instant le sens de parcours.
3. Écrire les fonctions `next`, `previous`, `hasNext`, `hasPrevious` qui permettent respectivement de :
  - récupérer l'élément suivant dans la liste et de déplacer l'itérateur après cet élément,
  - récupérer l'élément précédent dans la liste et de déplacer l'itérateur avant cet élément,
  - savoir s'il existe un élément suivant,
  - savoir s'il existe un élément précédent.
4. Si on veut imposer un sens de parcours lors de la création de l'itérateur que faut-il modifier ?
5. Si on veut utiliser l'itérateur pour insérer de nouvelles données dans la `ldcd` que faut-il ajouter comme fonctions ? Donner le code de ces fonctions.

Vous devrez utiliser le mécanisme de la compilation séparée.

# Arbre binaire d'entiers en dynamique

## 1 Définitions

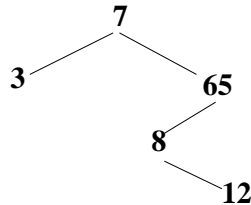
**Un arbre binaire** est une structure qui est soit vide, soit composée par une racine et deux sous-arbres : gauche et droit. Il est défini d'une façon récursive :

Arbre = Vide | <racine, gauche, droit>

racine = Élément

gauche, droit = Arbre

**Exemple :**



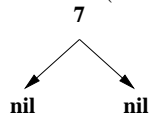
Dans cet arbre, 7 est la racine, le sous-arbre ayant 3 pour racine est le sous-arbre gauche alors que le sous-arbre ayant 65 pour racine est le sous-arbre droit.

Un arbre binaire peut servir pour structurer les données dans le but de les traiter d'une façon mieux optimisée. Dans l'exemple ci-dessus si on parcourt l'arbre avec la politique GRD (gauche-racine-droit), on obtient les valeurs triées : 3-7-8-12-65.

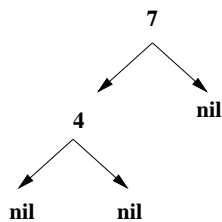
**Construction d'un arbre GRD :** La politique GRD peut se décrire de la manière suivante : on range chaque nombre d'une suite de nombres en utilisant le premier nombre de la suite comme racine pour l'arbre, puis les autres nombres récursivement en respectant la contrainte suivante : ceux qui sont inférieurs ou égaux sont placés dans le sous-arbre gauche, les autres sont placés dans le sous-arbre droit.

Si on décompose la construction de l'arbre pour la suite de nombres (7, 4, 14, 2, 6, 12, 15, 0, 5, 3), on obtient :

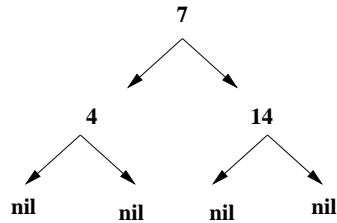
– étape 1, on traite le premier nombre 7 ; c'est la racine (**nil** représente le sous-arbre vide)



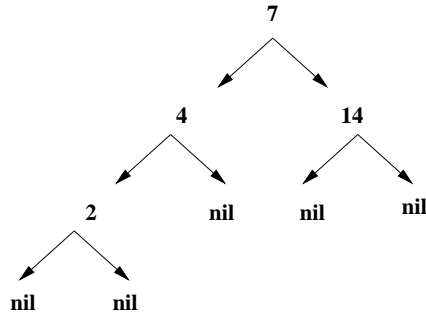
– étape 2, on traite le nombre suivant 4 ; il est  $<$  ou  $=$  à 7, on le met dans le sous-arbre gauche de 7



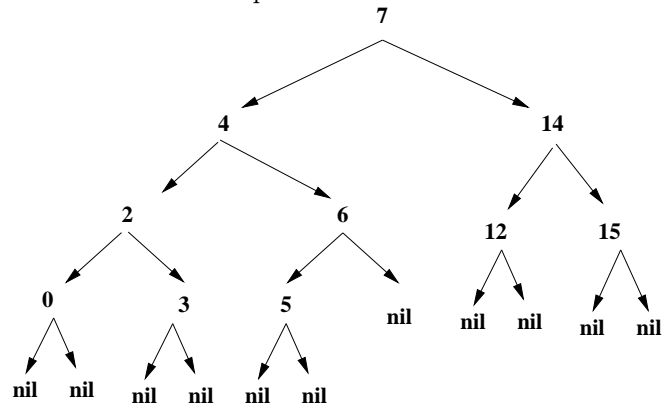
– étape 3, on traite le nombre suivant 14 ; il est  $>$  à 7, on le met dans le sous-arbre droit de 7



- étape 4, on traite le nombre suivant 2 ; il est  $<$  ou  $=$  à 4, on le met dans le sous-arbre gauche de 4



- et ainsi de suite ; on obtient alors l'arbre complet



Le TAD correspondant est :

**Sorte** : ARBB

**Utilise** : int

**Constructeurs** :

créer :  $\rightarrow$  ARBB

construire :  $\text{int} \times \text{ARBB} \times \text{ARBB} \rightarrow \text{ARBB}$

**Projecteurs** :

racine :  $\text{ARBB} \rightarrow \text{int}$

gauche :  $\text{ARBB} \rightarrow \text{ARBB}$

droit :  $\text{ARBB} \rightarrow \text{ARBB}$

estvide :  $\text{ARBB} \rightarrow \text{boolean}$

**Préconditions**

$a : \text{ARBB}$

$\text{racine}(a), \text{gauche}(a), \text{droit}(a)$  définis si not  $\text{estvide}(a)$

**Axiomes**

$\text{racine}(\text{construire}(r, ag, ad)) = r$

$\text{gauche}(\text{construire}(r, ag, ad)) = ag$

$\text{droit}(\text{construire}(r, ag, ad)) = ad$

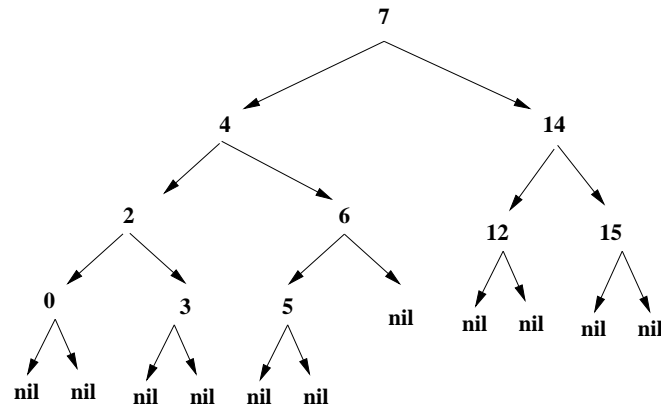
$\text{estvide}(\text{créer}) = \text{true}$

$\text{estvide}(\text{construire}(r, ag, ad)) = \text{false}$

Remarquons que ce TAD peut aussi se définir à partir d'un autre constructeur qui ajoute juste un nouvel entier à l'arbre en respectant la méthode GRD :

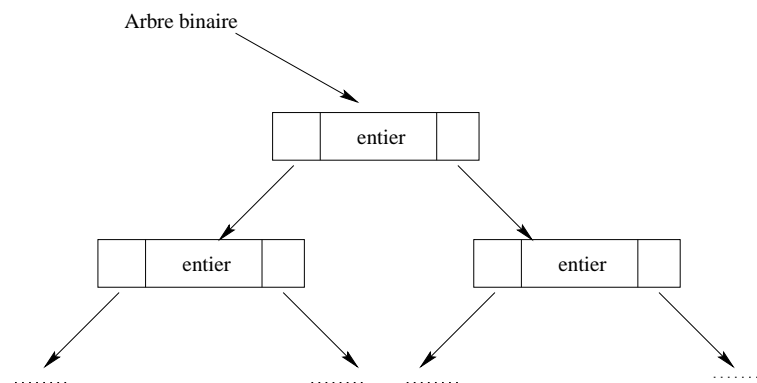
ajout :  $\text{int} \times \text{ARBB} \rightarrow \text{ARBB}$

**Les parcours** Pour être efficaces, ils dépendent de l'implémentation. Sur l'arbre suivant, le parcours en profondeur donnera la suite 0-2-3-4-5-6-7-12-14-15, alors que le parcours en largeur donnera la suite 7-4-14-2-6-12-15-0-3-5 :



## 2 Implémentation en dynamique

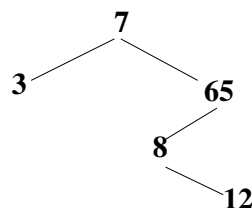
Un arbre binaire d'entiers est représenté par un pointeur vers une cellule qui contient d'une part un entier et d'autre part deux pointeurs vers d'autres cellules.



**Créer l'unité Arbre binaire et le fichier de test associé qui permettent de :**

1. Définir le type ARBRE d'entier(dynamique).
2. Définir le sous-programme INITIALISE l'arbre binaire (donné en paramètre) .
3. Définir un sous-programme qui permet de tester si l'arbre (donné en paramètre) est VIDE.
4. Définir un sous-programme qui permet d'ajouter un entier (donné en paramètre) à un arbre (donnée en paramètre). Pour ajouter un entier, on doit parcourir l'arbre de façon à positionner l'entier à sa place : l'arbre est trié en GRD (gauche racine droite : le plus petit entier est le plus à gauche possible de l'arbre).

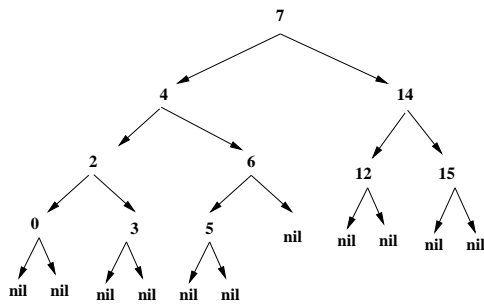
Exemple : insérer successivement 7, 65, 8, 12 et 3 donne l'arbre suivant :



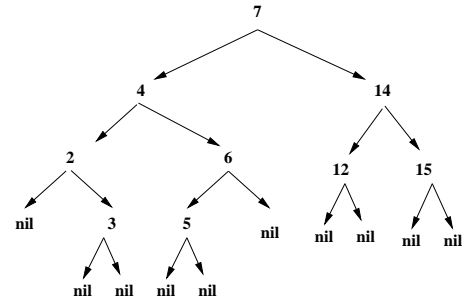
5. Définir un sous-programme qui permet d'enlever le plus petit élément de l'arbre (donné en paramètre) ce sous-programme doit renvoyer l'élément en question.
6. Définir un sous-programme récursif AFFICHE qui permet d'afficher tous les éléments de l'arbre dans l'ordre croissant.

**Rq : Cela donnera très facilement un parcours en profondeur.**

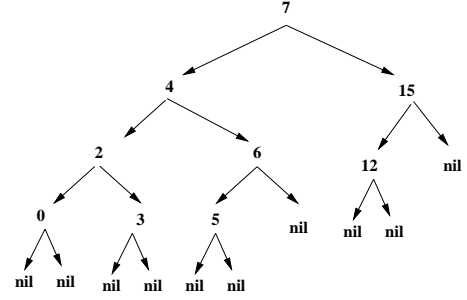
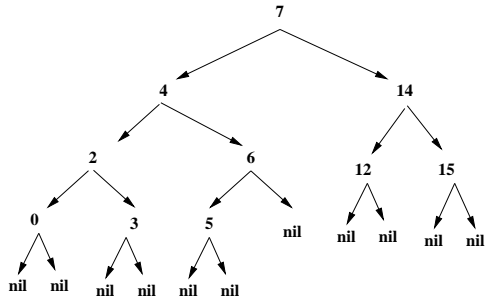
7. écrire la fonction qui permet de supprimer un entier d'un arbre GRD, par exemple :



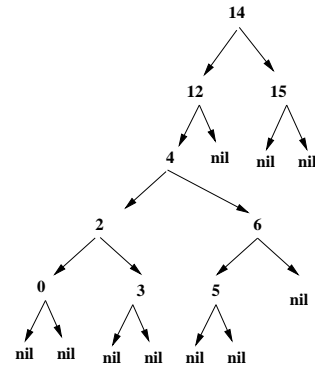
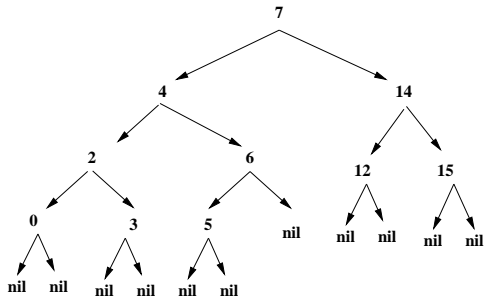
Si on supprime 0 :



Si on supprime 12 :



Si on supprime 7 :



8. (a) Peut-on transformer le sous-programme récursif AFFICHE en un sous-programme itératif, qui fait la même chose, sans utiliser de structure de données auxiliaire et sans modifier le type ARBRE ?
- (b) Si non, quelle structure de données auxiliaire, autre qu'un arbre proposez-vous ? Donnez alors le sous-programme itératif correspondant.
- (c) Peut-on éviter la structure de données auxiliaire en modifiant le type ARBRE ? Donnez alors le nouveau type ARBRE et le sous-programme itératif correspondant.

**Rq :** La réponse à cette question permet de trouver comment réaliser un parcours en largeur pour un arbre défini en dynamique.

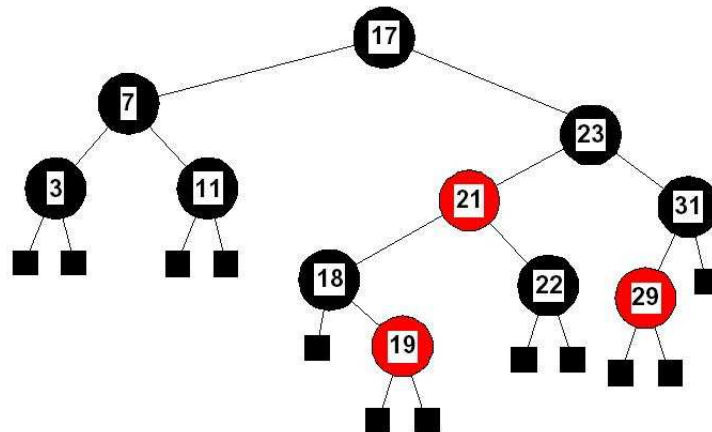
# Arbres rouges et noirs

## 1 Définitions

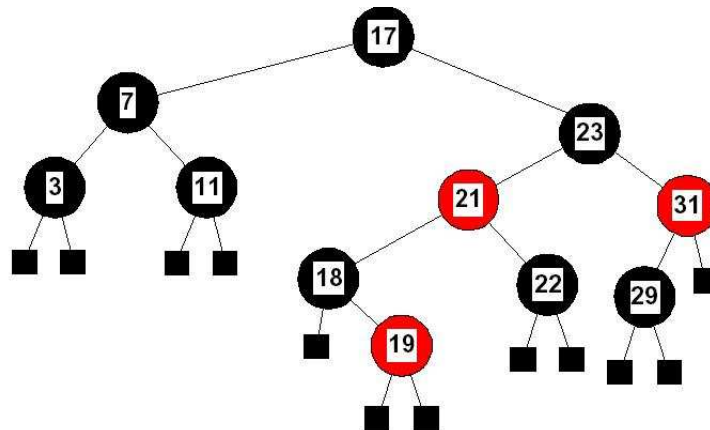
Un **arbre rouge et noir** est un arbre GRD où chaque nœud est de couleur rouge ou noire de telle sorte que

1. les enfants d'un nœud rouge sont noirs,
2. le nombre de nœuds noirs le long d'une branche de la racine à une feuille est indépendant de la branche. Autrement dit, pour tout nœud de l'arbre, les chemins de ce nœud vers les feuilles (nœud sentinelle) qui en dépendent ont le même nombre de nœuds noirs.

Cet arbre est un arbre rouge et noir :



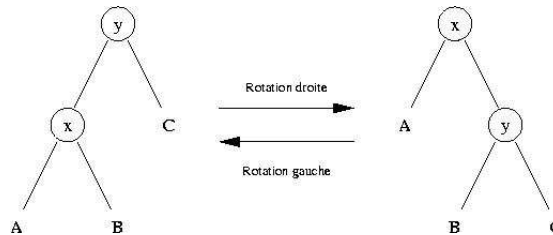
Cet arbre n'est pas un arbre rouge et noir (à cause de la branche droite du nœud 31 qui ne contient pas le même nombre de noirs que les autres branches) :



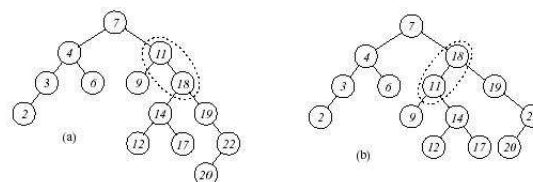
Un arbre rouge et noir est donc un arbre GRD “presque” équilibré

La première condition stipule que les nœuds rouges ne sont pas trop nombreux. La dernière condition est une condition d’équilibre. Elle signifie que si on oublie les nœuds rouges d’un arbre on obtient un arbre binaire parfaitement équilibré. En contrôlant cette information de couleur dans chaque nœud, on garantit qu’aucun chemin ne peut être deux fois plus long que n’importe quel autre chemin, de sorte que l’arbre reste équilibré.

**Les rotations** Les rotations sont des modifications locales d’un arbre binaire. Elles consistent à échanger un nœud avec l’un de ses fils. Dans la rotation droite, un nœud devient le fils droit du nœud qui était son fils gauche. Dans la rotation gauche, un nœud devient le fils gauche du nœud qui était son fils droit. Les rotations gauche et droite sont inverses l’une de l’autre. Elles sont illustrées par la figure ci-dessous. Dans les figures, les lettres majuscules comme A, B et C désignent des sous-arbres.



La figure ci-dessous montre comment une rotation permet de rééquilibrer un arbre.



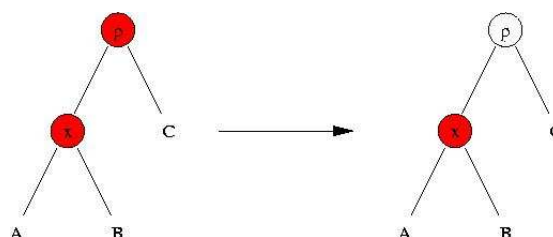
Rééquilibrage d’un arbre par une rotation. Une rotation gauche sur le nœud de clé 11 de l’arbre (a) conduit à un arbre (b) mieux équilibré : la hauteur de l’arbre est passée de 5 à 4.

## 2 Insertion d’une valeur

L’insertion d’une valeur dans un arbre rouge et noir commence par l’insertion usuelle d’une valeur dans un arbre binaire de recherche. Le nouveau nœud devient rouge de telle sorte que la propriété (3) reste vérifiée. Par contre, la propriété (2) n’est plus nécessairement vérifiée. Si le père du nouveau nœud est également rouge, la propriété (2) est violée. Afin de rétablir la propriété (2), l’algorithme effectue des modifications dans l’arbre à l’aide de rotations. Ces modifications ont pour but de rééquilibrer l’arbre. Soit x le nœud et p son père qui sont tous les deux rouges. L’algorithme distingue plusieurs cas.

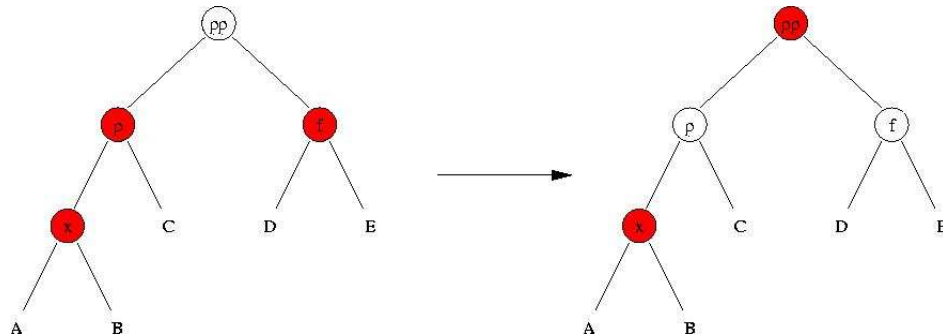
### Cas 0 : le nœud père p est la racine de l’arbre

Le nœud père devient alors noir. La propriété (2) est maintenant vérifiée et la propriété (3) le reste. C’est le seul cas où la hauteur noire de l’arbre augmente.



### Cas 1 : le frère f de p est rouge

Les nœuds p et f deviennent noirs et leur père pp devient rouge. La propriété (3) reste vérifiée mais la propriété ne l'est pas nécessairement. Si le père de pp est aussi rouge. Par contre, l'emplacement des deux nœuds rouges consécutifs s'est déplacé vers la racine.

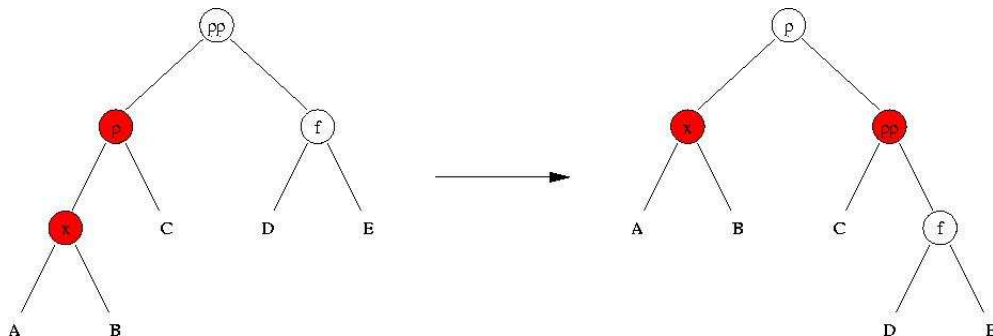


### Cas 2 : le frère f de p est noir

Par symétrie on suppose que p est le fils gauche de son père (attention : ne pas oublier de prévoir l'autre cas, celui où p est le fils droit de son père). L'algorithme distingue à nouveau deux cas suivant que x est le fils gauche ou le fils droit de p.

#### Cas 2a : x est le fils gauche de p.

L'algorithme effectue une rotation droite entre p et pp. Ensuite le nœud p devient noir et le nœud pp devient rouge. L'algorithme s'arrête alors puisque les propriétés (2) et (3) sont maintenant vérifiées.



#### Cas 2b : x est le fils droit de p.

L'algorithme effectue une rotation gauche entre x et p de sorte que p devienne le fils gauche de x. On est ramené au cas précédent et l'algorithme effectue une rotation droite entre x et pp. Ensuite le nœud x devient noir et le nœud pp devient rouge. L'algorithme s'arrête alors puisque les propriétés (2) et (3) sont maintenant vérifiées.

