TRAVAUX PRATIQUES: série n°3

Implémenter les types abstraits VECTEUR ,LISTE, PILE et FILE

Christoph-Samuel / Jankowiak-Matthias

I-Problématique:

Ce TP a pour but de fournir une implémentation correcte pour les types abstraits VECTEUR, LISTE, PILE et FILE en s'appuyant sur les spécifications CASL proposées en cours. Quatre choix s'offre à nous : VECTEUR, LISTE, **PILE** et FILE. Pour ce TP nous avons choisi d'implémenter le type abstrait PILE. Une pile est liste particulière c'est une structure de données fondée sur le principe « dernier arrivé, premier sorti » ce qui veut dire que le dernier élément, ajouté à la **pile**, sera le premier à en sortir.

Pour cela nous allons réaliser les 4 étapes du cycle de développement initié lors de la série n°1:

– étape 1 : la spécification du type abstrait,
– étape 2 : la validation de la spécification sous hets (DOLiator : hets en ligne),
– étape 3 : l'implémentation de la spécification,
– étape 4 : la vérification de l'implémentation,

II-Réalisation:

Étape 1: Spécification du type abstrait

La spécification s'effectue en 3 parties, on distingue l'en-tête (1 à 3), la signature (4 à 15) et la sémantique(16 à 20). La signature de la spécification est la partie visible ou interface de celle-ci. La sémantique est un ensemble de propriétés (axiomes), elle consiste à énoncer les propriétés des opérations du type abstrait.

```
1  library libraryPile
2  spec PILE [sort Elem] =
3      generated type
4     Pile[Elem] ::= pileVide |
5                    empiler(Pile[Elem];Elem)
6     pred
7       estVide: Pile[Elem]
8     ops
9       depiler: Pile[Elem]->?Pile[Elem] ;
10      sommet: Pile[Elem]->?Elem;
11
12 forall e: Elem; p: Pile[Elem]
13
14    . def sommet(p) <=> not estVide(p)
15    . def depiler(p) <=> not estVide(p)
16    . estVide(pileVide)
17    . not estVide(empiler(p,e))
18    . sommet(empiler(p,e))=e
19    . depiler(empiler(p,e))=p
20 end
```

Spécification en CASL :

Dans une spécification, une opération peut avoir deux statuts possibles les constructeurs et les accesseurs. Une opération est un constructeur si elle retourne un résultat, dont le type est défini par la spécification en cours. Ici les constructeurs sont : pileVide(), empiler(p,e) et dépiler(p).

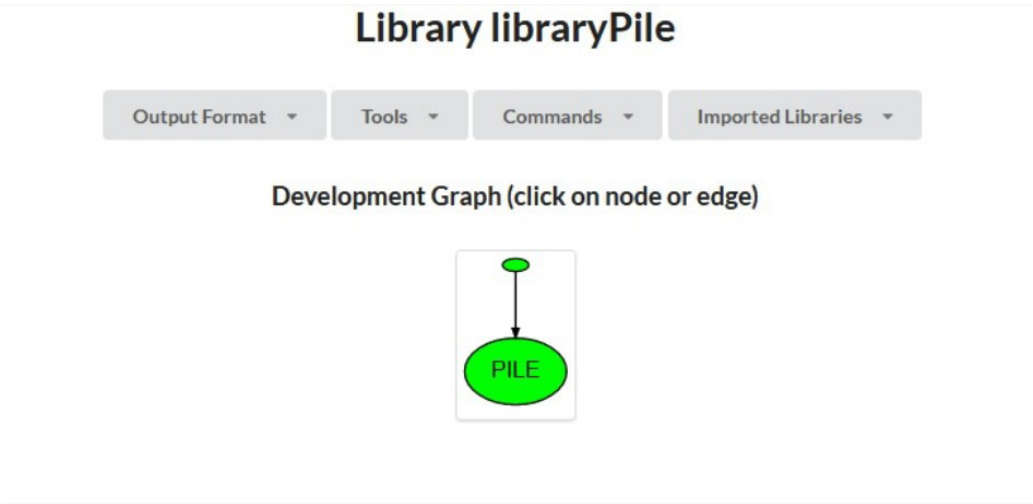
Un accesseur du type est une opération qui permet d'observer l'état d'un objet du type et sans y apporter la moindre modification. Mais cette opération est un accesseur si et seulement si elle a au moins un argument de type défini, elle rend un résultat de type importée. Dans notre cas les accesseurs sont : estVide(v) et sommet(p).

Pour dérouler le cycle de développement du type abstrait, il faut d'abord étudier le cahier des charges. Pour manipuler les piles, on se limitera ici aux 6 opérations suivantes, nécessaire pour bonne implémentation :

– créer une pile vide: pileVide : Pile[Elem]
– empiler un objet: empiler: Pile[Elem]x Elem → Pile[Elem]
– retirer l'objet qui se trouve au sommet: dépiler : Pile[Elem] →? Pile[Elem] **si et seulement si une pile est non vide:** not estVide(p)
– tester si une pile est vide: estVide : Pile[Elem] → Booléen
– consulter le sommet d'une pile si la pile est non vide: sommet: Pile[Elem] →? Elem

Étape 2: Validation de la spécification du type abstrait.

Cette étape permet de prouver que la spécification satisfait la consistante (pas d'axiomes exprimant des propriétés contradictoires) et la complétude (suffisamment d'axiomes pour savoir si une certaine propriété est vraie ou fausse). Dans notre cas, la spécification est validée.



Résultat à partir de l'analyseur HETS:

L'un des outils les plus puissants permettant de valider une spécification est hets (Heterogeneous ToolSet). Il est développé par CoFi (Universität Magdeburg) Il est possible d'appeler l'analyseur HETS en ligne à partir de DOLiator (comme ci-dessus).

Étape 3: Implémentation de la spécification

Fichier.h

Pour cela on réalise un fichier.h et un fichier.c : Le fichier.h est un fichier d'interface qui est consultable par le futur utilisateur du type. Ce fichier est ici appelé libraryPile.h.

Voici notre fichier.h : libraryPile.h

```c
/*
définition du fichier .h */
#ifndef LIBRARYPILE_H
#define LIBRARYPILE_H

```

Cette première partie nous sert a définir le nom et la zone d'action de notre fichier libraryPile.h.

```c
/*
en-tête standard d'entrée/sortie en langage c */
#include <stdio.h>
/*
permet de gérer la mémoire dynamiquement (free, malloc...) */
#include <stdlib.h>
/*
propose un ensemble de fonction de traitement de caractères */
#include <ctype.h>

```

Ensuite ce sont trois fichiers incluant des bibliothèques nécessaire pour coder en langage C parmi lesquelles on retrouve <stdio.h>, <stdlib.h> et <ctype.h>.

```c
/*
définition de faux à la valeur 0 */
#define FAUX 0
/*
définition de vrai à la valeur 1 */
#define VRAI 1
/*
taille maximale de 100 éléments à la pile */
#define maxsize 100

```

```c
/*
type entier déf des booléens */
typedef int BOOLEEN;
/*
type entier déf d'un élement */
typedef int ELEMENT;

```

On trouve les définitions de trois constantes FAUX et VRAI d'une part initialisés par défauts a 0 et 1 et d'autre part maxzize a 100, mais également les types entiers BOOLEEN et ELEMENT.

```
33  /*
34  propose un type CONCRET pour implémenter le type ABSTRAIT des piles */
35  typedef struct une_place{
36    ELEMENT element;
37    struct une_place *suivante;
38  }une_place;
39
40  /*
41  définition du type des places: un type pointeur vers un objet de type pile */
42  typedef struct une_place*  PILE;
43
```

Cette parte définit une structure pour une place dans la pile et pointe cette place vers un objet de type pile.

```
44  /*
45  constructeur qui créer une pile vide */
46  PILE pileVide();
47  /*
48  constructeur qui permet d'empiler un élement */
49  PILE empiler(PILE p,ELEMENT e);
50  /*
51  constructeur qui permet de dépiler le dernier élément inséré */
52  PILE depiler(PILE p);
53  /*
54  accesseur qui permet de savoir si une pile est vide */
55  BOOLEEN estVide(PILE p);
56  /*
57  accesseur définissant le sommet de la pile */
58  ELEMENT sommet(PILE p);
59
```

Nous avons ensuite les définitions et en-tête des trois constructeur du type abstrait des piles; pileVide(), empiler() et dépiler() ainsi que les deux accesseurs est Vide() et sommet().

```
60  /*
61  fin de la définition du fichier .h */
62  #endif LIBRARYPILE_H
63
```

Et pour finir la fin de la zone d'action de notre fichier libraryPile.h.

Fichier.c

Le fichier.c est un fichier d'implémentation, non consultable pour implémenter les corps de toutes les opérations déclarées dans libraryPile.h. Ce fichier est ici appelé libraryPile.c.

Voici notre fichier.c : libraryPile.c

```c
/*
en-tête standard d'entrée/sortie en langage c */
#include <stdio.h>
/*
permet de gérer la mémoire dynamiquement (free, malloc...) */
#include <stdlib.h>
/*
propose un ensemble de fonction de traitement de caractères */
#include <ctype.h>
/*
pour les en-têtes des constructeurs et accesseurs prédéfinis */
#include "libraryPile.h"

/*
définition de faux à la valeur 0 */
#define FAUX 0
/*
définition de vrai à la valeur 1 */
#define VRAI 1
```

Comme pour libraryPile.h on retrouve trois fichiers incluant des bibliothèques nécessaire pour coder en langage C ainsi que les deux constantes FAUX et VRAI.

```c
/*
création d'une pile vide */
PILE pileVide(){
  return NULL;
}
```

Le premier constructeur pileVide(), retourne seulement NULL car il sert seulement a crée une pile.

```c
/*
empiler d'un élément dans une pile non-vide */
PILE empiler(PILE laPile,ELEMENT e){
  PILE sommet;
  sommet=malloc(sizeof(struct une_place));
  if(sommet == NULL){
    fprintf(stderr,"Allocation impossible \n");
    exit(EXIT_FAILURE);
  }
  else{
    sommet->element=e;
    sommet->suivante=laPile;
    laPile=sommet;
  }
}
```

Ensuite nous avons le constructeur empiler qui tout d'abord fait une l'allocation mémoire pour l'élément se retrouvant au sommet, s'il n'y à pas de sommet(la pile est vide) un message d'erreur indiquant une allocation impossible apparaît, sinon le sommet prend l'élément empilé.

```
43  /*
44  depiler un élément d'un pile non-vide */
45  PILE depiler(PILE laPile){
46    PILE autrePile;
47    autrePile=laPile;
48    laPile=autrePile->suivante;
49    free(autrePile);
50    autrePile=NULL;
51    return laPile;
52  }
```

Le constructeur depiler lui crée une seconde pile pour mettre l'élément pointé par le sommet dans cette deuxième pile, vide la pile avec un free(autrePile) pour effacer l'élément dépilé, effectue un autrePile==NULL pour effacer celle-ci et retourne la première pile.

```
54  /*
55   tester si la pile est vide */
56   BOOLEEN estVide(PILE laPile){
57     return(laPile==NULL);
58   }
```

L'accesseur estVide est un booléen qui nous permet de savoir si la pile demandée est vide.

```
60  /*
61  indique quel élément se trouve au sommet de la pile */
62  ELEMENT sommet(PILE laPile){
63    return laPile->element ;
64  }
```

L'accesseur sommet nous retourne la valeur du dernier élément placer dans la pile

Étape 4: Vérification de l'implémentation

Afin de vérifier si l'implémentation est correcte vis à vis de sa spécification, nous allons créer un fichier preuvePile.c (processus de la vérification formelle) incluant libraryPile.c (permettant d'afficher les erreurs demandées en cas d'échec de compilation). Le principe de mise en œuvre consiste à vérifier que les constructeurs du type construisent correctement les objets du type. Cela signifie que ces constructeurs doivent satisfaire tous les axiomes énoncés en spécification.

La fonction main() va permettre de vérifier que:
– chacun des constructeurs implémentés
– satisfait tous les axiomes des accesseurs

Voici notre preuvePile.c

```c
/*
en-tête standard d'entrée/sortie en langage c */
#include <stdio.h>
/*
permet de gérer la mémoire dynamiquement (free, malloc...) */
#include <stdlib.h>
/*
propose un ensemble de fonction de traitement de caractères */
#include <ctype.h>
/*
pour les fonctions des constructeurs et accesseurs prédéfinis */
#include "libraryPile.c"

/*
programme principal qui vérifie les différents test de précondition */
int main(){
  PILE p, p1;
  int success;
  ELEMENT e;

  /*
  allocation mémoire et vérification */
  p=malloc(sizeof(struct une_place));
  p1=malloc(sizeof(struct une_place));
  if(p==NULL || p1==NULL){
    fprintf(stderr,"Allocation impossible \n");
    exit(EXIT_FAILURE);
  }
```

Au début du programme, on crée deux piles (p et p1), on initialise une variable success qui va vérifier pour chaque constructeurs et accesseurs leur bon fonctionnement et on crée également un élément e.

On procède à une allocation mémoire pour les deux piles

```
30    /*
31    vérification de l'implémentation du constructeur pileVide */
32    p=pileVide();
33    /*
34    initialisation de l'indice success */
35    success=0;
36    /*
37    vérification avec l'accesseur estVide
38    vérifier la propriété : estVide(p)= True */
39    if(!(estVide(p))) success+=1;
40    /*
41    bilan de la vérification */
42    if(success!= 0)
43    {
44      printf("\n Implémentation incorrecte du constructeur pileVide");
45      printf("\n Interruption de la vérification : revoir l'implémentation du type asbtrait");
46      exit(EXIT_FAILURE);
47    }
```

```
49    /*
50    vérification de l'implémentation du constructeur empiler */
51    printf("\n saisir un élément, p1 est le premier argument");
52    printf("\n saisir un élément, e est le second argument");
53    p=empiler(p1,e);
54
```

On effectue un test de précondition pour le premier constructeur, si le test n'est pas fonctionnel la variable success prend pour valeur 1 rentre dans une boucle et affiche une erreur d'implémentation. On répète ensuite ce schéma pour les autres constructeurs et accesseurs, réinitialisation de success a 0, test et bilan de la vérification si erreur sinon on passe au prochain...

```
55    /*
56    réinitialisation de la variable success */
57    success=0 ;
58
59    /*
60    vérification avec l'accesseur estVide */
61    if(estVide(p)) success+=1;
62
63    /*
64    bilan de la vérification */
65    if(success!= 0){
66      printf ("\n Implémentation incorrecte de estVide(empiler)");
67      printf ("\n Interruption de la vérification: revoir l'implémentation du type abstrait");
68      exit(EXIT_FAILURE);
69    };
```

```
71    /*
72    réinitialisation de la variable success */
73    success=0 ;
74
75    /*
76    vérification avec sommet */
77    if(sommet(p)!= e) success+=1;
78
79    /*
80    bilan de la vérification */
81    if(success!= 0){
82      printf ("\n Implémentation incorrecte de sommet(empiler)");
83      printf ("\n Interruption de la vérification : revoir l'implémentation du type abstrait");
84      exit(EXIT_FAILURE);
85    };
```

```
87    /*
88    réinitialisation de la variable success */
89    success=0 ;
90
91    /*
92    vérification avec depiler */
93    if(depiler(p)!= p1) success+=1;
94
95    /*
96    bilan de la vérification */
97    if(success!= 0){
98      printf ("\n Implémentation incorrecte de depiler(empiler)");
99      printf ("\n Interruption de la vérification : revoir l'implémentation du type abstrait ");
100     exit(EXIT_FAILURE);
101   };
```

```
103   /*
104   bilan des vérifications sur le type abstrait des piles */
105   printf("\n L'implementation du type abstrait est vérifiée");
106   printf("\n Fin normale de la vérification de l'implémentation du type abstrait");
107   return EXIT_SUCCESS;
108 }
109
```

Et pour finir si seulement les deux dernier printf s’affichent (105 et 106) l’étape de vérification de l’implémentation est bonne

III-Bilan/Conclusion:

Sur le plan théorique, réaliser ce TP nous a permis de comprendre le cycle de développement afin d'implémenter un type abstrait en lien avec notre problématique. En effet, l'utilisateur implémente des types abstraits afin d'étudier un type PILE.L'utilisation d'un type PILE est très efficace pour ajouter(empiler) ou enlever(dépiler) un élément, si bien entendu la pile est non vide.

La définition du type abstrait est ici avantageuse par rapport à la définition d'un type concret car elle est indépendante du langage de codage utilisé. Donc tout changement de langage ne remet pas cause cette définition. Il en résulte une grande stabilité des logiciels développés.