TD No2

Exercice 1 ensembles représentés par des tableaux

Nous allons représenter les ensembles par des tableaux. On rappelle le TAD ensemble vu en cours :

Sorte Ensemble

Utilise: Booléen, Element

Opérations:

- ensemble_vide: -> Ensemble
- ajouter: Element x Ensemble -> Ensemble
- retirer: Element x Ensemble -> Ensemble
- appartient: Element x Ensemble -> Booléen

Propriétés:

Tout ensemble non vide peut être construit en ajoutant ses éléments à l'ensemble vide. Un ensemble est donc soit vide, soit de la forme ajouter(e, E).

- appartient(e, ensemble_vide) = Faux
- appartient(e, ajouter(e, E)) = Vrai // l'élément ajouté appartient à l'ensemble
- si e != e', appartient(e, ajouter(e', E))= appartient(e, E) // pour les autres éléments, l'ajout ne change pas leur situation.
- appartient(e, supprimer(e, E)) = Faux // un élément supprimé n'est plus dans l'ensemble
- si e != e', appartient(e, supprimer(e', E))=appartient(e,E) // pour les autres éléments, la suppression ne change pas leur situation

Question 1 : union d'ensembles

Ajoutez l'opération union au type abstrait ensemble. L'union de deux ensembles A et B est un ensemble contenant tous les éléments de A et de B.

Question 2 : union d'ensembles: implantation à l'aide tableaux non triés

On représente les ensembles par des tableaus non triés. Proposer un algorithme permettant de réaliser l'union de deux ensembles et l'implantation en C correspondante . Vous déterminerez sa complexité au pire en nombre de comparaisons d'éléments et en nombre d'affectations d'éléments.

On pourra utiliser l'algorithme, vu en cours, testant l'appartenance d'un élément à un ensemble représenté par un tableau non trié :

```
typedef struct ttableau {
       int tailleMax;
       int taille;
       int * tab;
} tableau;
int appartient(int e, tableau t){
  int cb;
  for (cb=0; cb < t.taille; cb ++){
```

```
    if (t.tab[cb]==e)
      return 1;
  return 0;
}
```

Question 3 : union d'ensembles (cas trié)

On représente maintenant les ensembles par des tableaux triés. Que faut-il modifier dans l'algorithme de la question 2 pour le rendre compatible avec notre nouvelle représentation des ensembles ? Quelle influence cette modification a-t-elle sur la complexité de l'algorithme ?

Question 4 : union d'ensembles (cas trié)

Plutôt que d'ajouter les éléments un à un, nous nous proposons de tirer partie de la remarque suivante:

- soient A et B deux ensembles représentés par des tableaux triés
- soient minA le plus petit élément de l'ensemble A, soit minB, le plus petit élément de l'ensemble B
- Où se trouve minA dans le tableau représentant A ? Où se trouve minB dans le tableau représentant B ?
- soit minUnion le plus petit élément de A U B. Où se trouve-t-il dans l'ensemble représentant A U B ?
- peut-on exprimer minUnion en fonction de minA et de minB ?

Proposez un algorithme tirant partie de ces remarques. Vous déterminerez un majorant de sa complexité en nombre de comparaisons d'éléments et en nombre d'affectations d'éléments de façon à prouver qu'elles sont d'un ordre de grandeur inférieur à celles de l'algorithme de la question 3.

Exercice 2 pile d'entiers

Nous allons représenter les piles d'entiers par des listes chaînées. On rappelle le type abstrait de données vu en cours :

Sorte: Pile

Utilise: Booléen, Element

Opérations:

- pile_vide: -> Pile
- empiler: Pile x Element -> Pile
- dépiler: Pile -> Pile
- sommet: Pile -> Element
- est_vide: Pile -> Booléen

Propriétés:

- *dépiler(p)* est défini si *est_vide(p) == Faux*
- *sommet(p) est défini* si *est_vide(p) == Faux*
- *dépiler(empiler(p,e))= p* (rem; toute pile *p'* non vide pouvant se mettre sous la forme

p'=empiler(p,e), cette propriété permet de déterminer ce que vaut *dépiler(p')* pour toute pile *p'* non vide.

- *sommet(empiler(p,e))=e*
- *est_vide(pile_vide()) == Vrai*
- si *p=empiler(p', e)* alors *est_vide(p)=Faux*

Question 1:

En C, on représentera une pile par une liste chaînée de la façon suivante:

```
typedef struct telement{
  int valeur;
  struct telement * suivant;
} element;
typedef element * pile;
```

Implanter les opérations sur les piles.

Exercice 3 type abstrait de données : file d'entiers

Nous allons représenter les piles d'entiers par des listes chaînées. On rappelle le type abstrait de données vu en cours :

Sorte: File

Utilise : Element, Booléen

Opérations:

- file_vide: ->File
- ajouter: File x Element -> File
- supprimer : File -> File
- premier: File -> Element
- est_vide: File -> Booléen
- est_pleine: File -> Booléen

Propriétés:

- *ajouter(f,e)* est définie si *est_pleine(f)* vaut Faux
- *supprimer(f)* est définie si *est_vide(f)* vaut Faux
- *premier(f)* est définie si *est_vide(f)* vaut Faux
- si *est_vide(f) == Vrai alors premier(ajouter(f,e))==e*
- si *est_vide(f) == faux alors premier(ajouter(f,e))==premier(f)*
- si *est_vide(f) == Vrai alors supprimer(ajouter(f,e))==file_vide*
- si *est_vide(f) == faux alors supprimer(ajouter(f,e))==ajouter(supprimer(f),e)*

Question 1:

Dans un premier temps, on envisage de représenter les file par des listes chaînées. Proposez les algorithmes correspondants et évaluez leur complexité en nombre de parcours d'éléments (on utilisera l'affectation d'une adresse comme opération fondamentale).

Question 2

On se propose de maintenir un pointeur qui désignera le dernier élément de la liste. Une file sera donc représentée par un élément du type suivant :

```
typedef struct tfile {
       liste premier; // premier élément
       liste dernier; // dernier élément
} file;
```

Expliquez en quoi cette représentation permettra d'améliorer la complexité des opérations sur les file et écrire l'implantation correspondante des opérations sur les files.