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.

Eléments de correction :

- *Union: Ensemble x Ensemble -> Ensemble*

Propriétés:

- *appartient(e, union(A,B))=appartient(e,A) ou appartient(e,B)*

On peut le dire autrement:

- *les éléments de A U B sont les éléments de A et les éléments de B*

On peut aussi le dire de la façon suivante :

- *A U ensemble_vide = A*
- *A U ajouter(e,B)=ajouter(e, A U B)*

Cette dernière façon de définir les propriétés nous donne quasimment un algorithme implantable (mais pas forcémment eficace) pour calculer l'union de deux ensembles. Si on suppose définie la fonction ***premier*** *qui retourne l'un des éléments de l'ensemble, la fonction* ***reste*** *qui retourne l'ensemble privé de l'élément retourné par* ***premier*** *et la fonction* ***copie*** *qui retourne un nouvel*

ensemble copie de l'ensemble passé en argument :

```
ensemble union(ensemble A, ensemble B){
       if (estVide(B))
              return copie(A);
       else {
         element e=premier A;
         ensemble res= union(reste(A), B);
          ajouter(e, &res);
          return res;
       }
}
```

Un algorithme classique proche consiste à ajouter successivement les éléments de A puis ceux de B à un ensemble initialement vide :

```
ensemble union(ensemble A, ensemble B){
  ensemble res;
  while(! EstVide(A)){
    ajouter(premier(A), &res);
    A=reste(A);
  }
  while(! EstVide(B)){
    ajouter(premier(B), &res);
    B=reste(B);
  }
return res;
}
```

Cette algorithme sera valable quelque soit la représentation des ensembles. On peut même le généraliser à autre chose que des ensembles: il sera valable pour tout type de structure de données à partir du moment où l'on souhaite obtenir une structure de données contenant les éléments de A et ceux de B. C'est la fonction ***ajouter*** *qui garantit que le résultat est une structure de données viable (ici, un ensemble).*

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;
}
```

Eléments de correction :

Pour construire unon(t1, t2), on va :

- *recopier t1 dans un tableau résultat (pour éviter le test d'appartenance présent dans la fonction ajouter et qui est inutile ici car on sait que les éléments ne sont pas encore dans le tableau cible vu que t1 est un ensemble (tableau non trié sans redondance).*
- *insérer un à un les éléments de t2 dans le tableau résultat à l'aide d'une fonction sachant ajouter un élément à un ensemble et garantissant le fait qu'un élément n'apparaît pas deux fois. Tester l'appartenance est utile car un élément de t2 peut aussi appartenir à t1.*

```
// recopie le tableau s dans le tableau dest
// le contenu initial du tableau destination est détruit
// le tableau destination doit être suffisamment grand :
// dest->tailleMax >= s.taille
void copie(tableau s, tableau * dest){
  int cb;
  assert(dest->tailleMax >= s.taille);
  for (cb=0; cb < s.taille, cb ++)
    dest->tab[cb]=s.tab[cb];
  dest->taille=s.taille;
}
// indique si un tableau est plein :
// t.taille==t.tailleMax
// ajoute un élément à un tableau (à la fin)
// si l'élément est déjà dans le tableau, on ne fait rien
// dans le cas où l'élément n'est pas dans le tableau,
// le tableau ne doit pas être plein
void ajouter(int e, tableau * pt){
  if (! appartient(e, *pt)){ // si l'élément n'est dans le tableau
    assert(! Est Plein(*pt));
    pt->tab[pt->taille]=e;
    pt->taille++;
  }
}
// retourne un nouveau tableau contenant l'union de t1 et de t2.
// on choisit taille.t1+taille.t2 comme taille pour le tableau résultat ce qui
// est discutable car 1) dans certains cas, c'est trop et 2) dans d'autres,
// ça ne laisse aucune marge pour ajouter des éléments au tableau.
// la bonne solution consiste à profiter du fait que nos tableaux sont
//  alloués dynamiquement pour les redimentionner si ils sont pleins
//  (ou trop vide). Ceci dit, on sort du cadre de ce TD.
tableau union(tableau t1, tableau t2){
```

```
  tableau res;
  int cb;
  res=creeTableau(t1.taille+t2.taille);
  copie(t1, &res);  // aucune comparaisons; t1.taille affectations d'éléments
  for (cb=0; cb< t2.taille; cb ++)
    ajouter(t2.tab[cb], &res); // de 1 à res.taille comparaisons;
                               // 0 ou 1 affectation
  return res;
}
```

Complexité: dans la fonction ajouter(t2.tab[cb], &res), on a de 1 à res.taille comparaisons et de 0 à 1 affectations suivant que l'on ajoute ou non l'élément. L'un des pires cas est celui où aucun élément du second ensemble n'est dans le premier car :

- *chaque élément est compaté avec tous ceux de res (qui en contient t1.taille, puis t1.taille+1, ..., t1.taille+t2.taille-1) au total: t1.taille*t2.taille+(1+2+3+...+t2.taille-1) =t1.taille*t2.taille+t2.taille * (t2.taille+1)/2-1=Θ(n²) si t1.taille et t2.taille valent n tous les deux.*
- *Chaque élément est ajouté à res (1 affectation).*

Dans ce pire cas, on a :

- *t1.taille + t2.taille affectations (t1.taille dans copie puis t2.taille dans ajouter)*

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 ?

Elements de correction: la solution la plus simple consiste à modifier la fonction ajouter en y remplaçant l'ajout à la fin par un ajout à la bonne place après un décalage des éléments situés à sa droite.

On peut optimiser un peu les choses en remplaçant l'appel à appartient par un appel à une fonction renvoyant la place que l'élément devrait avoir dans le tableau. Il suffit ensuite de voir s'il y est vraiment pour éviter de l'y ajouter dans ce cas. La fonction place peut utiliser une recherche dichotomique dont la complexité est en log n (non vu encore, sera vu lors du prochain cours). Ici, on se contentera de la recherche séquencielle utilisée ci-dessus pour le cas non trié.

L'algorithme devient :

```
void ajouter(int e, tableau * pt){
  int p, cb;
  p=place(e, *pt);
  if (! (pt->tab[p] == e)){ // si l'élément n'y est pas
    // on décale, noter le sens de parcours (cb --)
    for (cb = pt->taille; cb > p; cb --)
      pt->tab[cb]=pt->tab[cb-1];
    pt->tab[p]=e;
    pt->taille ++;
  }
}
```

D'un point de vue complexité, on passe en n² pour le nombre d'affectations (cas où le second tableau a tous ses éléments strictement plus petits que ceux du premier et où l'ajout se fait toujours avant les éléments de t1 que l'on est donc obligé de décaler).

Aucun changement pour le nombre de comparaisons (ou passage en nlog(n) si recherche dichotomique.

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 2.

Eléments de correction:

nos tableaux sont triés. Les plus petits éléments sont donc en début de tableau. Le plus petit élément de l'union est donc soit le plus petit élément de A, soit le plus petit élément de B.

Exemple: A={2,3,5,8,11}, B={1,4,5,8,11}, le plus petit élément de l'union est 1=min(1,2)

Cela nous conduit à l'algorithme suivant où on suppose que res est un tableau suffisamment grand pour contenir l'union des deux ensembles:

```
cbA=0;cbB=0;cbres=0;
while ((cbA < A.taille) && (cbB < b.taille)){
  if (A.tab[cbA]== B.taille[cbB])
    cbB++;
  else if ((A.tab[cbA] < B.taille[cbB]){ // le min est dans A
    res.tab[cbres]=A.tab[cbA];
    cbres++;
    cbA++;
  } else { // le min est dans B
    res.tab[cbres]=B.tab[cbB];
    cbres++;
    cbB++;
  }
// on recopie ce qui reste dans l'éventuel tableau non fini
while (cbA<A.taille){
  res.tab[cbres]=A.tab[cbA];
  cbA++;
```

```
  cbres++;
}
while (cbB<B.taille){
  res.tab[cbres]=B.tab[cbB];
  cbB++;
  cbres++;
}
res.taille=cbres;
// maintenantres = A U B
```

D'un point de vue complexité, on fait au plus une ou deux comparaison pour chaque élément de A et de B et une affectation au plus pour chaque élément de A et de B. La complexité est donc en O(n) tant en nombre de comparaisons qu'en nombre d'affectations.

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.

Eléments de correction: deux solutions s'offrent à nous :

- *on ajoute les valeurs en fin de liste: d'un point de vue complexité, cela obligera à aller en fin de liste (et donc à parcourir la liste du début à la fin) pour les opérations empiler, dépiler et sommet*
- *on ajoute les valeurs en début de liste: d'un point de vue complexité, les opérations empiler, dépiler et sommet se font en temps constant puisqu'il suffit de travailler sur le début de la liste. C'est donc la solution que nous allons adopter.*

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

Propriétés:

- *enfiler(f,e)* est définie si *est_pleine(f)* vaut Faux
- *défiler(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(enfiler(f,e))==e*
- si *est_vide(f) == faux alors premier(enfiler(f,e))==premier(f)*
- si *est_vide(f) == Vrai alors défiler(enfiler(f,e))==file_vide*
- si *est_vide(f) == faux alors défiler(enfiler(f,e))==enfiler(defiler(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).

Eléments de correction:

l'ajout et la suppression se font à des bouts différents. Quelque soit la solution choisie, l'une de ces opérations nécessitera d'aller en fin de liste ce qui nécessite de parcourir toute la liste.

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.

Eléments de correction:

pour éviter de parcourir toute la liste quand on a besoin d'agir en fin de liste, on mémorise l'adresse de la fin de liste. 2 solutions:

- *on supprime en fin de liste et on ajoute en début de liste: c'est une mauvaise solution car la suppression suppose de modifier le champ suivant de l'élément précédent l'élément à supprimer. Là, on a simplement l'adresse du dernier élément. Trouver l'adresse de l'élément précédent suppose de parcourir la liste (ou d'utiliser des listes doublement chaînées ce qui ne présente aucun avantage par rapport à la solution décrite ci-dessous). L'ajout en début de liste ne pose pas de problème.*
- *on ajoute en fin de liste et on retire en début de liste: c'est la bonne solution car l'ajout nécessite de simplement modifier le champ suivant de l'élément de f in de liste. La suppression consistera à supprimer le premier élément (remplacer premier par premier->suivant et désallouer l'ancien premier élément)*

On opte donc pour la seconde solution :

- *ajout: en fin de liste, on remplace dernier->suivant par l'adresse du nouvel élément. Si la file est vide (dernier->suivant n'a alors pas de sens), on crée simplement un élément vers lequel pointeront premier et dernier.*
- *Suppression: on supprime l'élément de tête désigné par premier ce qui est maintenant de la routine.*

D'un point de vue complexité, toutes les opérations sont en temps constant ($\Theta(1)$).

Lorsque l'on supprime le dernier élément, on se retrouve avec une file vide. L'algorithme de suppression implique que dans ce cas, on aura premier==NULL. On peut se demander s'il est utile de faire en sorte que dernier soit aussi NULL. La réponse est NON car la fonction qui teste si une file est vide regarde simplement si premier == NULL.