Université Claude Bernard Lyon 1

Année universitaire : 2022 / 2023

LIFAPSD : Algorithmique, Programmation et Structures de Données

Epreuve Anonyme Commune – Session 1
3 janvier 2023
Durée : 1h30

Note : / 20

Nom :
Prénom :
N° d'étudiant :
Signature :

coller ici

Documents et téléphones portables interdits. Répondez dans les emplacements prévus à cet effet. Travaillez au brouillon d'abord de sorte à rendre une copie propre – nous ne pouvons pas vous garantir une copie supplémentaire. **Il se tenu compte de la présentation et de la clarté de vos réponses**.

Exercice 1 : Type de donnée abstrait pour un ensemble

Dans un ensemble fini en mathématiques une seule occurrence d'un élément est possible. Dans cet exercice nous souhaitons créer un type de donnée abstrait afin de représenter un ensemble fini de valeurs entières. Ce type de donnée abstrait sera décrit à travers une classe C++ dédiée nommée **`Ensemble`**.

Les opérations possibles sur un ensemble qui nous intéressent sont les suivantes :

- **`estPresent`** qui prend une valeur entière en paramètre et indique si elle est présente dans l'ensemble
- **`ajouterElement`** qui prend en paramètre une valeur entière à ajouter à l'ensemble et qui l'ajoute si elle n'est pas déjà présente
- **`supprimerElement`** qui prend en paramètre une valeur entière et qui la supprime de l'ensemble si elle était présente
- **`vider`** qui vide l'ensemble de valeurs
- **`intersection`** qui prend un autre ensemble en paramètre et qui retourne un nouvel ensemble stocké sur le tas contenant l'intersection des deux ensembles (c'est-à-dire toutes les valeurs communes aux deux ensembles)
- **`sousEnsemble`** qui prend un autre ensemble en paramètre et qui indique si l'ensemble de l'instance est un sous-ensemble de l'ensemble passé en paramètre (c'est-à-dire que toutes ses valeurs sont dans l'ensemble passé en paramètre)

Nous allons implémenter notre ensemble grâce à un tableau dynamique. Pour rappel, un tableau dynamique (classe **`TableauDynamique`**) a trois données membres : l'entier **`taille_utilisee`** qui contient le nombre d'éléments du tableau, l'entier **`capacite`** qui est la taille maximale actuelle du tableau, et le tableau d'entiers **`ad`**. Voir l'annexe pour la liste des fonctions membres de la classe **`TableauDynamique`**.

Question 1 : Donner, en notation O en fonction de n le nombre d'éléments du tableau, et justifier brièvement les complexités amorties des fonctions membres **`ajouterElement`** et **`rechercherElement`** de la classe **`TableauDynamique`**.

`ajouterElement` est en O(1) en amortie car pour n ajouts on a 3n opérations, donc 3 opérations pour 1 ajout.
`rechercherElement` est en O(n) car dans le pire des cas il faut parcourir tout le tableau pour rechercher l'élément.

Pour implémenter notre ensemble grâce à un tableau dynamique nous ajoutons simplement une donnée membre nommée **tab** de type **TableauDynamique** dans la classe **Ensemble**.

Question 2 : Compléter le code ci-dessous en indiquant la déclaration de cette donnée membre ainsi que le constructeur de la classe **Ensemble**.

```
class Ensemble {
   public :
      // ajouter ci-dessous la donnée membre
      TableauDynamique tab;

      // ajouter ci-dessous le code du constructeur
      // précondition : aucune
      // postcondition : l'ensemble créé est un ensemble vide (c.-à-d. à aucun élément)
      Ensemble ( ) {
         // rien à faire, la donnée membre est automatique créée et correspond bien à
         // un ensemble vide (tableau vide)
      }

      // les autres fonctions membres iront ici

};
```

L'entête de la fonction membre **estPresent** de la classe **Ensemble** est la suivante :

```
Fonction estPresent (element : entier) : entier
Précondition : aucune
Résultat : l'indice de l'élément dans l'ensemble (c.-à-d. dans le tableau dynamique) si
l'élément est présent et la valeur -1 sinon
```

Question 3 : Donner le code C++ de la fonction membre **estPresent**.

```
int Ensemble::estPresent (int element) const {
   for (unsigned int i = 0; i < tab.taille_utilisee; i++)
      if (tab.valeurIemeElement(i) == element) return i;
   return -1;
}

// ou bien simplement en utilisant rechercherElement de la classe TableauDynamique :

   return tab.rechercherElement(element);
```

Question 4 : Donner et justifier la complexité, dans le pire des cas, de cette fonction.

estPresent est en O(n) car dans le pire des cas l'élément n'est pas dans l'ensemble et il faut parcourir l'entièreté de l'ensemble à n éléments pour s'en apercevoir. La fonction **valeurIemeElement** est en O(1) car fait un simple accès à l'élément d'indice i dans un tableau.

Question 5 : Donner le code C++ de la fonction membre **ajouterElement** de la classe **Ensemble**.

```
void Ensemble::ajouterElement (int element) {
   if (estPresent(element) == -1)
      tab.ajouterElement(element);
}
```

Question 6 : Donner et justifier la complexité amortie et la complexité dans le pire des cas de cette fonction.

ajouterElement est en O(n) en amortie car l'appel à **estPresent** est en O(n) et l'appel à **ajouterElement** est en O(1) en amortie.

Elle est en O(2n)=O(n) dans le pire des cas. Ce cas arrive quand l'élément n'est pas encore présent (O(n) pour le savoir) puis au pire on a besoin de recopier le tableau s'il n'y a plus de place (en O(n) aussi).

Question 7 : Donner le code C++ des fonctions membres **intersection** et **sousEnsemble** de la classe **Ensemble**.

```
Ensemble * Ensemble::intersection (const Ensemble & e) const {
   Ensemble * inter = new Ensemble;
   for (unsigned int i = 0; i < tab.taille_utilisee; i++) {
      if (e.estPresent(tab.valeurIemeElement(i)) != -1)
         inter->ajouterElement(tab.valeurIemeElement(i));
   return inter;
}

bool Ensemble::sousEnsemble (const Ensemble & e) const {
   for (unsigned int i = 0; i < tab.taille_utilisee; i++) {
      if (e.estPresent(tab.valeurIemeElement(i)) == -1)
         return false;
   }
   return true;
}
```

Question 8 : Compléter le code du programme principal suivant qui créé sur la pile deux ensembles, ajoute des éléments dans ces deux ensembles et calcule leur intersection.

```
#include <time.h>
#include <stdlib.h>
#include "Ensemble.h"

int main () {
   // Création de 2 ensembles sur la pile
   Ensemble e1,e2;


   // Ajout de 5 valeurs entières aléatoires entre -4 et 4 dans chacun des ensembles
   srand((unsigned int) time(NULL));
   for (unsigned int i = 0; i < 5; i++) {
      e1.ajouterElement(rand()%9-4);
      e2.ajouterElement(rand()%9-4);
   }



   // Calcul de l'intersection des deux ensembles
   Ensemble * e3 = e1.intersection(e2);      // ou bien e2.intersection(e1)

   // Libération mémoire
   delete e3;


   return 0;
}
```

Exercice 2 : Stratégie d'agrandissement d'un tableau dynamique

Nous avons vu en cours que la stratégie de doubler la capacité d'un tableau dynamique lorsque l'on n'avait plus de place pour ajouter un élément donnait une complexité en O(n) pour n ajouts et donc en coût constant pour 1 ajout. Nous avons aussi vu en TD qu'ajouter 10 emplacements lorsque l'on a plus de place donne une complexité en $O(n^2)$ pour n ajouts et donc un coût linéaire pour 1 ajout. La stratégie de multiplier est donc meilleure que d'additionner.

Cette stratégie de doubler la capacité est implémentée dans beaucoup de structures de données de différents langages comme les « vector » en C++, les « Vec » en Rust, les « sequences » en Nim etc. Mais d'autres langages ont préféré utiliser un rapport plus petit. En effet, doubler la capacité peut s'avérer être trop prenant en mémoire lorsque l'on gère des tableaux assez grands (ex. lors de l'ajout du 8193ème élément, le tableau devient de taille 16384 dont 8191 cases non utilisées). Les « ArrayList » de Java et les « PyListObject » de Python utilisent donc par exemple respectivement un facteur de 1.5 et 1.125 à la place d'un facteur de 2.

Question 9 : Donner le code C++ de la fonction membre **`ajouterElement`** de la classe **`TableauDynamique`** qui utilise un facteur de 1.5 au lieu d'un facteur de 2 pour augmenter la taille du tableau lorsqu'il n'y a plus de place.

```
void TableauDynamique::ajouterElement (ElementTD e) {
   ElementTD * temp;
   if (taille_utilisee == capacite) {
      temp = ad;
      ad = new ElementTD [(int) 1.5*capacite+1];
      capacite = (int) 1.5*capacite+1;
      for (unsigned int i = 0; i < taille_utilisee; i++) ad[i] = temp[i];
      delete [] temp;
  }
  ad[taille_utilisee] = e;
  taille_utilisee++;
}
```

On a besoin du (int) pour convertir le calcul entre le réel 1,5 et l'entier capacite en un entier (seul type possible pour la taille d'un tableau).
On a besoin du +1 pour arrondir à l'entier supérieur (en effet si capacite=1 alors (int)1.5*1=(int)1.5=1 et donc le tableau ne grandit pas).

Question 10 : Compléter le tableau suivant (cases grisées) qui indique le nombre d'opérations (affectations et copies d'éléments) effectuées à chaque ajout d'un n^{ème} élément.

Nb éléments ajoutés	Capacité	Taille utilisée	Indices existants	Nb opérations
0	1	0	0	0
1	1	1	0	1 aff.
2	2	2	0-1	1 aff. + 1 copie
3	4	3	0-3	1 aff. + 2 copies
4	4	4	0-3	1 aff.
5	7	5	0-6	1 aff. + 4 copies
...				
7	7	7	0-6	1 aff.
8	11	8	0-10	1 aff. + 7 copies
9	11	9	0-10	1 aff.
...				
12	17	12	0-16	1 aff. + 11 copies
...				

Question 11 : Conclure sur la complexité de cette stratégie.

On remarque que le nombre de copies est de plus en plus grand avec le nombre d'éléments ajoutés. Il augmente de manière linéaire. Par rapport à doubler la capacité, on a juste des tableaux plus petits pour un nombre d'éléments ajoutés donnés. On fait des extensions plus souvent (mais plus petites) donc on fait plus d'opérations qu'en doublant la capacité mais au final le nombre de copies est du même ordre de grandeur, c'est-à-dire O(n) pour n ajouts et donc O(1) pour 1 ajout en amortie. On a perdu (un peu) en complexité mais on a gagné en mémoire car les tableaux sont plus petits.

Exercice 3 : Parcours d'un arbre binaire

Pour rappel un **Noeud** d'un arbre binaire est défini comme une structure contenant trois champs : **info** de type **ElementA**, **fg** et **fd** tous les deux de type pointeur sur **Noeud**. Et la seule donnée membre de la classe **Arbre** est **adRacine**, un pointeur sur le **Noeud** racine de l'arbre.

Nous souhaitons calculer le nombre de nœuds d'un arbre.

Question 12 : Donner le code C++ d'une fonction membre RECURSIVE de la classe **Arbre** nommée **nbNoeudsRec** qui calcule et retourne le nombre de nœuds de l'arbre. Si vous avez besoin d'une fonction auxiliaire donner aussi son code.

```
unsigned int Arbre::nbNoeudsRec () const {
    return nbNoeudsRecDepuisNoeud(adRacine);
}

unsigned int Arbre::nbNoeudsRecDepuisNoeud (const Noeud * n) const {
    if (n == NULL) return 0;
    return 1 + nbNoeudsRecDepuisNoeud(n->fg) + nbNoeudsRecDepuisNoeud(n->fd);
}
```

Question 13 : Donner le code C++ d'une fonction membre ITERATIVE de la classe **Arbre** nommée **nbNoeudsIter** qui calcule et retourne le nombre de nœuds de l'arbre.

```
unsigned int Arbre::nbNoeudsIter () const {
   File f;
   Noeud * n;
   if (estVide()) return 0;
   unsigned int cpt = 0;

   f.enfiler(adRacine);
   while (!f.estVide()) {
      n = (Noeud *) f.premierDeLaFile();
      f.defiler();
      cpt++;
      if (n->fg != NULL) f.enfiler(n->fg);
      if (n->fd != NULL) f.enfiler(n->fd);
   }
   return cpt;
}


// parcours également possible avec une pile
```

Annexe : liste des fonctions membres des classes TableauDynamique, Liste, File, Pile et Arbre (constructeurs et destructeurs omis)

Classe TableauDynamique

```
void vider ();
void ajouterElement (ElementTD e);
ElementTD valeurIemeElement (unsigned int indice) const;
void modifierValeurIemeElement (ElementTD e, unsigned int indice);
void afficher () const;
void supprimerElement (unsigned int indice);
void insererElement (ElementTD e, unsigned int indice);
int rechercherElement (ElementTD e) const;
```

Classe Liste

```
void vider ();
bool estVide () const;
ElementL iemeElement (unsigned int indice) const;
void modifierIemeElement (unsigned int indice, ElementL e);
void afficherGaucheDroite () const;
void afficherDroiteGauche () const;
void ajouterEnTete (ElementL e);
void ajouterEnQueue (ElementL e);
void supprimerTete ();
int rechercherElement (ElementL e) const;
void insererElement (ElementL e, unsigned int indice);
void supprimerElement (ElementL e);
```

Classe File

```
void enfiler (ElementF e);
ElementF premierDeLaFile () const;
void defiler ();
bool estVide () const;
unsigned int nbElements () const;
```

Classe Pile

```
void empiler (ElementP e);
ElementP consulterSommet () const;
void depiler ();
bool estVide () const;
```

Classe Arbre

```
bool estVide () const;
void vider ();
void insererElement (ElementA e);
void supprimerElement (ElementA e);
Noeud * rechercherElement (ElementA e) const;
```