Université Claude Bernard Lyon 1

Année universitaire : 2016 / 2017

LIFAP3 : Algorithmique et programmation avancée

Contrôle final
9 mai 2017
Durée : 2 heures

Note : / 20

coller ici

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

coller ici

Documents et téléphones portables interdits. Le barème est donné à titre indicatif. 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 sera tenu compte de la présentation et de la clarté de vos réponses**.

Exercice 1 : Tableau dynamique (7 points)

Supposons la classe **`TableauDynamique`** vue en cours, TD et TP contenant trois données membres : **`capacite`** de type **`unsigned int`**, **`taille_utilisee`** de type **`unsigned int`** et **`ad`** de type pointeur sur **`ElementTD`**. Les fonctions membres de la classe **`TableauDynamique`** sont aussi rappelées en annexe.

Dans cet exercice, on souhaite modifier la stratégie de gestion du tableau. On veut que le tableau ait toujours exactement la même taille que le nombre d'éléments du tableau (pas d'espace mémoire alloué pour rien).

Question 1.1 : Que changeriez-vous aux données membres de la classe **`TableauDynamique`** ?

Si la capacité du tableau est toujours égale à son nombre d'éléments, alors les deux données sont redondantes. Il ne faut en garder qu'une seule, que l'on pourrait simplement appeler **`taille`**.

Question 1.2 : Donner le code C++ du constructeur par défaut de la classe **`TableauDynamique`** basé sur cette nouvelle stratégie.

```
TableauDynamique::TableauDynamique () {
   ad = NULL;
   taille = 0;
}
```

Question 1.3 : Donner le code C++ de la procédure membre **`ajouterElement`** de la classe **`TableauDynamique`** basé sur cette nouvelle stratégie.

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

Question 1.4 : Quel est l'inconvénient majeur de cette stratégie ?

L'inconvénient majeur est que à chaque ajout et chaque suppression dans le tableau, il faut ré-allouer un autre tableau (de taille +1 ou -1), et le recopier. Cela prend du temps (complexité linéaire).

Question 1.5 : Donner le code C++ de l'opérateur [ ] surchargé dans la classe **`TableauDynamique`**. Cet opérateur retournera une référence à l'élément du tableau à l'indice donné en paramètre.

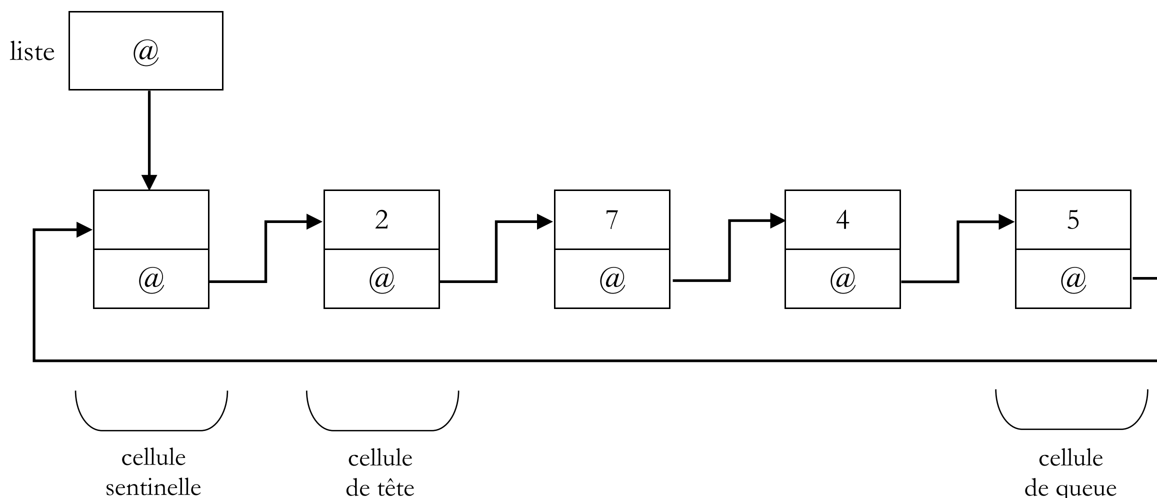
```
ElementTD& TableauDynamique::operator [] (int i) {
   if (i < 0 || i >= taille) {
     cout << "Erreur dans l'operateur [] de TableauDynamique avec l'indice " << i << endl;
     exit(1);
   }
   else return ad[i];
}
```

Exercice 2 : Liste simplement chaînée circulaire avec sentinelle (5 points)

Dans cet exercice, on s'intéresse à une classe **`Liste`** simplement chaînée circulaire avec sentinelle. La seule différence avec une liste simplement chaînée circulaire est la présence d'une cellule supplémentaire en tête de liste, la sentinelle.

Pour rappel, la classe **`Liste`** simplement chaînée circulaire a une donnée membre **`adPremiere`** de type pointeur sur **`Cellule`** et la classe **`Cellule`** a deux données membres : **`info`** de type **`ElementL`** et **`suivant`** de type pointeur sur **`Cellule`**.

La première cellule de la liste, la sentinelle, ne contient aucune information (donnée membre **`info`**) valide. Elle n'est là que pour aider à insérer facilement des éléments en tête et en queue de liste. Lorsque la liste est vide, seule la cellule sentinelle existe, et sa donnée **`suivant`** pointe sur elle-même. Dans l'exemple ci-dessous, la cellule en tête de liste est celle qui contient la valeur 2 en info, et celle de queue est celle qui contient la valeur 5. Le suivant de la sentinelle est la tête, le suivant de la queue est la sentinelle, et la première cellule indiquée par la liste est la sentinelle.



Question 2.1 : Ecrire, en C++, le constructeur de la classe **`Liste`** simplement chaînée circulaire avec sentinelle.

```
Liste::Liste () {
   adPremiere = new Cellule;
   adPremiere->suivant = adPremiere;
}
```

Question 2.2 : Ecrire, en C++, la procédure membre **ajouterEnTete** qui insère l'élément passé en paramètre en tête de liste. Le coût de cet ajout doit être en $O(1)$, c'est-à-dire qu'il ne doit pas dépendre du nombre d'éléments présents dans la liste.

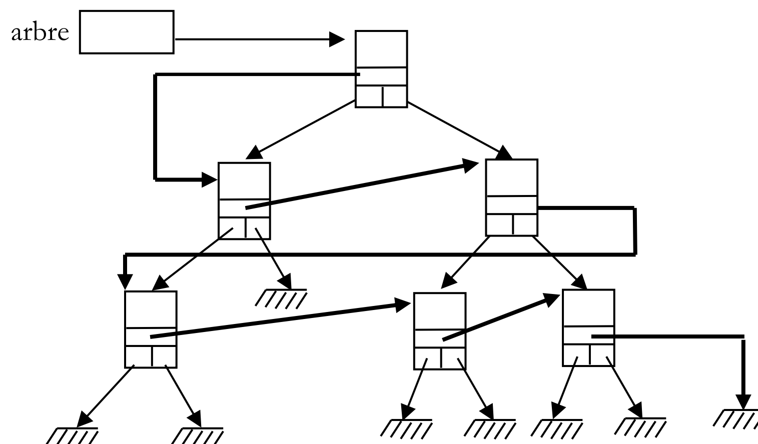
```
void Liste::ajouterEnTete (ElementL e) {
   Cellule * temp = new Cellule;
   temp->info = e;
   temp->suivant = adPremiere->suivant;
   adPremiere->suivant = temp;
}
```

Question 2.3 : Ecrire, en C++, la procédure membre **ajouterEnQueue** qui insère l'élément passé en paramètre en queue de liste. Le coût de cet ajout doit être en $O(1)$, c'est-à-dire qu'il ne doit pas dépendre du nombre d'éléments présents dans la liste. Vous avez le droit d'utiliser les autres fonctions membres de la classe, y compris **ajouterEnTete**.

```
void Liste::ajouterEnQueue (ElementL e) {
   ajouterEnTete(e);
   adPremiere->info = adPremiere->suivant->info;
   adPremiere = adPremiere->suivant;
}
```

Exercice 3 : Arbre binaire (5 points)

Dans cet exercice, on souhaite pouvoir parcourir un arbre binaire quelconque comme une liste simplement chaînée suivant la représentation donnée dans la figure ci-dessous. Pour cela, on ajoute la donnée membre **suivant** dans la classe **Noeud**.



Pour rappel, la classe **Arbre** a une seule donnée membre : **adRacine** de type pointeur sur **Noeud**. La classe **Noeud** a originellement trois données membres : **info** de type **ElementA**, **fg** et **fd** tous deux de type pointeur sur **Noeud**. Voir l'annexe pour les fonctions membres.

Question 3.1 : Quel est le type de la nouvelle donnée membre **suivant** ?

La donnée membre **suivant** est de type pointeur sur **Noeud** : **Noeud * suivant**.

Question 3.2 : Ecrire une fonction membre ITERATIVE de la classe **Arbre** s'appelant **getListe** qui ne prend pas de paramètre et qui rend un pointeur sur une liste allouée sur le tas où les éléments de la liste sont les données **info** des nœuds parcourus dans l'ordre indiqué par la figure.

```
Liste * Arbre::getListe () {

   if (estVide()) return NULL;

   Liste * l = new Liste;
   Noeud * n = adRacine;

   while (n != NULL) {
      l->ajouterEnQueue(n->info);
      n = n->suivant;
   }

   return l;
}

// ou bien (sans utiliser la donnée membre suivant)

Liste * Arbre::getListe () {

   if (estVide()) return NULL;

   Liste * l = new Liste;
   File f;
   Noeud * n;

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

   return l;
}
```

Question 3.3 : Quelle est la complexité de cette fonction **getListe** en fonction de n le nombre de nœuds de l'arbre ? Justifiez brièvement votre réponse.

En suivant les données membres « suivant », nous réalisons un parcours en largeur de l'arbre, en ne visitant qu'une seule fois chaque nœud. La fonction **getListe** est donc de complexité linéaire $O(n)$.

Exercice 4 : Pile d'appels (3 points)

En informatique, les piles sont notamment utilisées pour gérer les appels récursifs de fonction. Dans cet exercice, on souhaite utiliser une pile pour stocker les valeurs successives que prend un paramètre lors d'un appel à une fonction récursive. Pour cela, on ajoute une pile en paramètre, en mode donnée-résultat, afin de stocker les valeurs successives du paramètre. Au début de la fonction récursive la valeur du paramètre est ajoutée à la pile, qui est ensuite dépilée juste avant la fin de la fonction. Entre les deux opérations, des appels récursifs sont effectués qui ont pour effet d'empiler et dépiler les valeurs suivantes du paramètre. On terminera finalement la récursivité avec une pile vide.

Soit la fonction récursive suivante donnée en notation algorithmique :

Fonction factorielle (nombre : entier positif) : entier positif
Résultat : retourne la factorielle de nombre
Paramètre en mode donnée : nombre
Début
  Si n > 1 alors
    retourne nombre * factorielle(n-1)
  Sinon
    retourne 1
Fin factorielle

Question 4.1 : Traduire cette fonction en C++ et y apporter les modifications nécessaires afin de gérer une pile des valeurs successives que prend le paramètre **nombre**. Vous ajouterez donc une pile en paramètre et empilerez et dépilerez le paramètre **nombre** aux bons endroits.

```
unsigned int factorielle (unsigned int nombre, Pile & p) {
   p.empiler(nombre);
   if (n > 1) {
      unsigned int n = factorielle(nombre-1);
      p.depiler();
      return nombre * n;

   }
   else {
      p.depiler();
      return 1;
   }
}
```

Question 4.2 : Ecrire un programme principal qui calcule la factorielle de 10 en appelant la fonction **`factorielle`** et qui vérifie l'état correct de la pile après l'appel.

```
int main () {
   Pile p;
   unsigned int x = factorielle(10,p);
   if (!p.estVide()) {
      cout << "Erreur de pile lors de l'appel a factorielle";
      return 1;
   }
   return 0;
}
```

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;
unsigned int nbElements () 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);
void supprimerCellule (Cellule * c);
```

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;
```