Université Claude Bernard Lyon 1

Année universitaire : 2016 / 2017

LIFAP3 : Algorithmique et programmation avancée

Contrôle final
5 janvier 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 : Questions diverses (5 points)

Chacune question est sur un point.
Pour les trois questions à choix multiples suivantes : 1 point si toutes propositions correctes cochées, -0.5 point par proposition fausse cochée, pas de note négative.

Question 1.1 : Cochez les propositions correctes concernant une classe.

- □ N'importe quelle partie d'un programme peut toujours lire la valeur d'une donnée membre d'une instance de classe déclarée dans ce programme
- □ Une classe a toujours un constructeur et un destructeur par défaut
- □ L'opérateur = doit être surchargé pour permettre l'affectation entre instances de classe
- □ Le mot clé const appliqué à une fonction membre indique que les paramètres ne peuvent pas être modifiés par cette fonction
- □ Une instance de classe réservée sur le tas est automatiquement libérée de la mémoire en sortant du bloc d'instructions dans lequel elle a été allouée

Question 1.2 : Cochez les propositions correctes concernant les listes chaînées.

- □ L'insertion en tête dans une liste simplement chaînée est plus coûteuse que dans une liste doublement chaînée
- □ La recherche par dichotomie d'un élément dans une liste triée peut être faite en $O(\log_2 n)$
- □ Une liste simplement chaînée circulaire (où le dernier élément pointe sur le premier) prend plus de place en mémoire qu'une liste simplement chaînée non circulaire
- □ Une file et une pile peuvent être implémentées sous forme d'une liste chaînée

Question 1.3 : Cochez les propositions correctes concernant les arbres.

- □ Chaque nœud d'un arbre binaire a exactement deux fils
- □ Le parcours en ordre (infixé) permet de visiter les nœuds niveau après niveau
- □ Les parcours en pré-ordre (préfixé) et post-ordre (postfixé) visitent les nœuds d'un arbre binaire de profondeur 1 dans le même ordre
- □ Un arbre dégénéré (ex. en peigne) prend plus de place en mémoire qu'un arbre équilibré

Soit la classe **`Fonction`** suivante, donnée en notation algorithmique.

```
Classe Fonction
 public :
   abscisseMin : entier
   ordonnees : TableauDynamique de réels

   Procédure mystere (a : entier, b : entier, c : réel)
   Précondition : a est strictement inférieur à b
   Paramètres en mode donnée : a et b
   Paramètre en mode donnée-résultat : c
   Variables locales : i, abscisseMax : entiers
   Début
     c ← 0
     abscisseMax ← abscisseMin + ordonnees.taille_utilisee
     Si a ≥ abscisseMin et b ≤ abscisseMax alors
       Pour i allant de a - abscisseMin à b - abscisseMin - 1 par pas de 1 faire
          c ← c + (ordonnees[i]+ordonnees[i+1]) / 2.0
       Fin pour
     Fin si
   Fin mystere

Fin classe
```

Question 1.4 : Traduisez cette classe en C++.

Question 1.5 : Supposons que le tableau dynamique **ordonnees** contient les valeurs 1.0, -2.0, 0.0, 2.0, 2.0, 1.0, -1.0, et que l'abscisse minimale vaut -2. Que vaut c à la fin de l'exécution de **mystere** avec a=-1 et b=3 ? Justifier votre réponse. En déduire ce que fait la procédure membre **mystere**. Autrement dit, si vous deviez lui donner un nom plus explicite, lequel choisiriez-vous ?

Exercice 2 : Min/Max dans une liste simplement chaînée (4 points)

Dans cet exercice, on considère la classe **Liste** permettant de gérer des listes de valeurs réelles (**double**) simplement chaînées et non circulaires (i.e. comme vue en CM et TD).

Pour rappel, cette classe **Liste** a une donnée membre **adPremiere** de type pointeur sur **Cellule** et la classe **Cellule** a deux données membres : **info** de type **double** et **suivant** de type pointeur sur **Cellule**.

Question 2.1 : Compléter la nouvelle procédure membre **adrMinMax** ci-dessous qui recherche et « renvoie » les adresses des cellules de la liste contenant le minimum et le maximum des valeurs réelles contenues dans la liste (donnée membre **info**). On supposera que la liste ne contient pas de doublons de valeurs réelles. Vous ne ferez aucun appel à d'autres fonctions membres de la classe **Liste**.

```
void Liste::adrMinMax (.....................................) {









}
```

Question 2.2 : Compléter la nouvelle procédure membre **supprimerMinMax** ci-dessous qui supprime les cellules contenant le minimum et le maximum des valeurs réelles contenues dans la liste. Vous avez le droit d'utiliser les autres fonctions membres de la classe **Liste** (voir annexe) ainsi que la procédure **adrMinMax** de la question précédente.

```
void Liste::supprimerMinMax (void) {









}
```

Exercice 3 : Insertion dans une liste doublement chaînée et triée (3 points)

Dans cet exercice, on considère la classe **Liste** permettant de gérer des listes de valeurs réelles (**double**) doublement chaînées et non circulaires (i.e. comme vue en TP, voir l'annexe pour les fonctions membres).

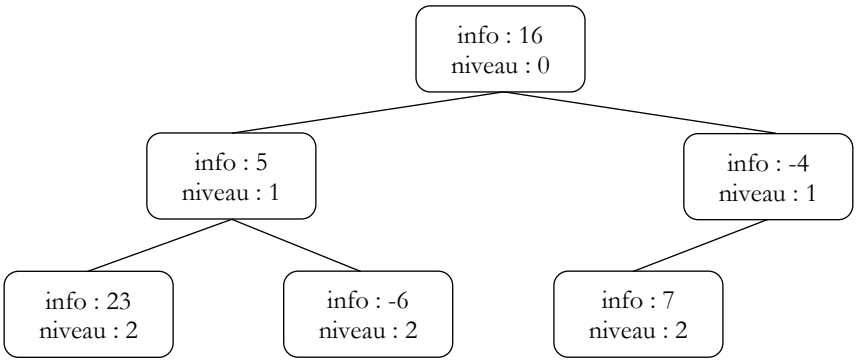
Pour rappel, cette classe **Liste** a une donnée membre **prem** de type pointeur sur **Cellule** et une donnée membre **last** de type pointeur sur **Cellule**. La classe **Cellule** a trois données membres : **info** de type **double**, **suivant** de type pointeur sur **Cellule** et **precedent** de type pointeur sur **Cellule**.

Ecrire, en C++, la procédure membre **insererTriee** de la classe **Liste**, qui insère un élément dans une liste triée par ordre croissant des données **info**.

Exercice 4 : Arbres binaires à niveaux (6 points)

Dans cet exercice, on considère la classe **`Arbre`** permettant de gérer des arbres binaires d'entiers signés (i.e. comme vue en CM, TD et TP). On souhaite stocker dans chaque nœud le niveau dans lequel il se situe dans l'arbre, sachant que la racine est au niveau 0 (voir schéma ci-dessous). Pour cela on ajoute un entier naturel en tant que donnée membre de la classe **`Noeud`**, et déclaré ainsi : **`unsigned int niveau`**.

Pour rappel, la classe **`Arbre`** a originellement une donnée membre **`adRacine`** de type pointeur sur **`Noeud`**, et la classe **`Noeud`** a trois données membres : **`info`** de type **`int`**, et **`fg`** et **`fd`** qui sont des pointeurs sur **`Noeud`**.



Question 4.1 : Ecrire la nouvelle procédure membre **`calculerNiveau`** qui calcule et stocke la donnée membre **`niveau`** dans chaque nœud de l'arbre.

Question 4.2 : Ecrire une procédure membre **`stockerNiveau`** ITERATIVE qui stocke les adresses des nœuds d'un niveau donné dans un tableau dynamique. Le niveau et le tableau dynamique seront passés en paramètres de la procédure. L'arbre sera supposé non vide.

Par exemple sur le schéma précédent, l'appel à **`stockerNiveau`** avec une valeur de niveau 2 mettra à jour le tableau dynamique avec les adresses des nœuds ayant pour info 23, -6 et 7.

Vous pouvez vous référez à l'annexe pour les fonctions membres des classes **`TableauDynamique`**, **`File`** et **`Pile`**.

Exercice 5 : Tri par arbre binaire de recherche (2 points)

Attention cet exercice est plus difficile, gardez le pour la fin.

Dans cet exercice, on considère connue une classe **`Arbre`** permettant de gérer des arbres binaires de recherche (ABR) d'entiers signés. Pour rappel, un ABR est un arbre où pour chaque nœud les valeurs dans le sous-arbre de gauche sont toutes plus petites que celle du nœud et dans le sous-arbre de droite toutes plus grandes.

On souhaite trier un tableau dynamique d'entiers signés en utilisant un ABR. Ecrire une procédure globale qui prend en paramètre un tableau dynamique d'entiers signés en mode donnée-résultat et qui le trie grâce à un ABR.

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 i) const;
void modifierValeurIemeElement (ElementTD e, unsigned int i);
void afficher () const;
void supprimerElement (unsigned int i);
void insererElement (ElementTD e, unsigned int i);
int rechercherElement (ElementTD e) const;
```

Classe Liste

```
void vider ();
bool estVide () const;
unsigned int nbElements () const;
ElementL iemeElement (unsigned int i) const;
void modifierIemeElement (unsigned int i, 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 i);
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;
```