Université Claude Bernard Lyon 1

Année universitaire : 2023 / 2024

LIFAPSD : Algorithmique, Programmation et Structures de Données

Epreuve Anonyme Commune – Session 1
8 janvier 2024
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 : Multiples occurrences dans un arbre binaire de recherche (ABR)

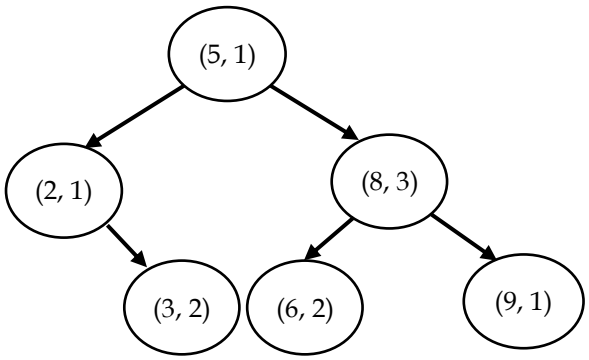
Dans cette UE nous n'autorisions pas les arbres binaires de recherche à posséder plusieurs occurrences de la même valeur. Nous nous en assurions dans la procédure d'ajout dans l'arbre qui n'ajoutait pas la valeur si elle était déjà présente. Dans cet exercice nous allons nous intéresser à une implémentation des ABR qui autorise plusieurs occurrences de la même valeur.

Pour rappel, dans un ABR tous les éléments dans le sous-arbre gauche d'un nœud sont strictement plus petits que ce nœud et tous les éléments dans le sous-arbre droit sont strictement plus grands.

Pour rappel encore, dans l'UE, le **`Noeud`** d'un arbre binaire était 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.

Les fonctions membres de la classe **`Arbre`** sont rappelées en annexe.

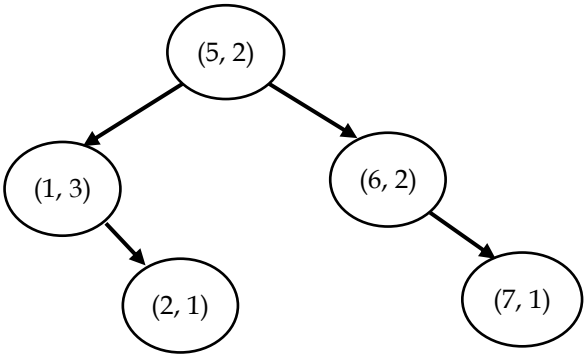
Afin de permettre les multiples occurrences, nous allons maintenant stocker le nombre d'occurrences de chaque valeur dans chaque **`Noeud`** correspondant.
Par exemple, l'ABR suivant :



indiquera qu'il y a une occurrence de la valeur 5, une occurrence de la valeur 2, trois occurrences de la valeur 8, deux occurrences de la valeur 3, etc.

Question 1 : Dessiner l'ABR obtenu après insertion des valeurs suivantes (insérées dans cet ordre) : 5 6 1 1 2 5 7 1 6

L'ABR obtenu est le suivant :



Le nombre d'occurrences de chaque valeur est ainsi stocké dans chaque **`Noeud`**.

Question 2 : Donner le code C++ de la déclaration de la structure **`Noeud`** prenant en compte ce nouveau champ que l'on nommera **`occurence`**.

```
struct Noeud {
    ElementA info;
    Noeud * fg;
    Noeud * fd;
    unsigned int occurence;
};
```

Lors de l'ajout d'une nouvelle valeur dans l'arbre, nous devons maintenant incrémenter le nombre d'occurrences si la valeur existe déjà. Et nous devons toujours créer un nouveau nœud (avec les bonnes valeurs des champs et à la bonne place) si la valeur n'existe pas encore.

Question 3 : Donner le code C++ de la fonction membre **`insererElement`** de la classe **`Arbre`** autorisant les multiples occurrences. Si vous avez besoin de créer une fonction intermédiaire, donner également son code.

```
void Arbre::insererElement (ElementA e) {
    insererElementDepuisNoeud(e, adRacine);
}
```

```
void Arbre::insererElementDepuisNoeud(ElementA e, Noeud * & n) {
    if (n == NULL) {
        n = new Noeud;
        n->info = e;
        n->fg = NULL;
        n->fd = NULL;
        n->occurence = 1;
    }
    else  {
        if (estInferieurElementA(e, n->info)) insererElementDepuisNoeud(e, n->fg);
        else if (estSuperieurElementA(e, n->info)) insererElementDepuisNoeud(e, n->fd);
             else n->occurence += 1;
    }
}
```

Les fonctions d'affichage de l'arbre doivent également tenir compte du nombre d'occurrences de chaque valeur. Par exemple, l'affichage en largeur de l'arbre de la première page doit afficher à l'écran : 5 2 8 8 8 3 3 6 6 9.

Question 4 : Donner le code C++ de la fonction membre **`afficherParcoursEnLargeur`** de la classe **`Arbre`** pouvant inclure de multiples occurrences.

```
void Arbre::afficherParcoursEnLargeur () const {
    if (estVide()) return ;

    File f;
    Noeud * n;

    f.enfiler(adRacine);
    while (!f.estVide()) {
        n = (Noeud *) f.premierDeLaFile();
        f.defiler();
        for (unsigned int i = 0; i < n->occurence; i++) {
            afficherElementA(n->info);
            cout << " " ;
        }
        if (n->fg != NULL) f.enfiler(n->fg);
        if (n->fd != NULL) f.enfiler(n->fd);
    }
    cout << endl;
}
```

Question 5 : Donner le code C++ d'une nouvelle fonction membre **`nbOccurence`** de la classe **`Arbre`** qui récupère et retourne le nombre d'occurrences d'une valeur passée en paramètre (cf. entête ci-dessous). Si vous avez besoin de créer une fonction intermédiaire, donner également son code.

Fonction nbOccurence (e : ElementA) : entier naturel
Pré-condition : aucune
Post-condition : l'arbre est inchangé
Paramètres en mode donnée : e
Résultat : le nombre d'occurrences de la valeur e dans l'arbre. La fonction retourne 0 si e n'est pas présent.

```
unsigned int Arbre::nbOccurence (ElementA e) const {
    return nbOccurenceDepuisNoeud(e,adRacine);
}

unsigned int Arbre::nbOccurenceDepuisNoeud (ElementA e, const Noeud * n) const {
    if (n == NULL) return 0;
    else {
        if (estInferieurElementA(e, n->info)) return nbOccurenceDepuisNoeud(e, n->fg);
        else if (estSuperieurElementA(e, n->info)) return nbOccurenceDepuisNoeud(e, n->fd);
             else return n->occurence;
    }
}
```

Question 6 : Soit un arbre non vide, donner et justifier la complexité, dans le pire des cas et le meilleur des cas, et avec la notation O, de la fonction **`nbOccurence`** en fonction de n, le nombre de nœuds de l'arbre.

Le pire des cas est lorsque l'arbre est dégénéré et la valeur recherchée est au bout de la branche unique. Il faut alors parcourir tous les nœuds de l'arbre. Dans ce cas, la fonction est de complexité $O(n)$.

Le meilleur des cas est lorsque l'arbre est équilibré (i.e. de hauteur minimale). Il faut donc parcourir au maximum une des branches de l'arbre qui est de longueur maximale $\log_2 n$ d'où une complexité en $O(\log_2 n)$ dans ce cas.

Question 7 : Compléter le code du programme principal ci-dessous permettant de tester cette nouvelle implémentation des ABR.

```
#include "ElementA.h"
#include "Arbre.h"

#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main() {
   // initialisation des tirages aléatoires
   srand(time(NULL));

   // création d'un arbre nommé a sur la pile
   Arbre a;

   cout << "Ajouts de ";
   // ajout de 30 valeurs entières aléatoires dans l'arbre
   for (unsigned int i = 0; i < 30; i++) {
      // tirage d'un entier entre -2 et 7 (inclus)
      ElementA e = (rand() % 10) - 2;
      // insertion dans l'arbre
      a.insererElement(e);
      // affichage de la valeur insérée
      cout << e << " ";
   }
   cout << endl;

   cout << "Parcours en largeur : ";
   // affichage du parcours en largeur de l'arbre
   a.afficherParcoursEnLargeur();

   // affichage du nombre d'occurrences de la valeur 4
   cout << "La valeur 4 apparait " << a.nbOccurence(4) << " fois" << endl;

   return 0;
}
```

Exercice 2 : Copie profonde d'un tableau dynamique générique

Dans cet exercice nous allons nous intéresser à copier le contenu d'un tableau dynamique générique. Nous avons vu dans l'UE que la classe **`TableauDynamique`** peut stocker des éléments de n'importe quel type grâce à la déclaration suivante des types des éléments du tableau dans le fichier `ElementTD.h` :

```
typedef void * ElementTD;
```

En effet les éléments du tableau sont alors des pointeurs, et le type des éléments pointés est décidé à l'exécution au moment des ajouts et peut être différent pour chaque tableau dynamique créé.

Question 8 : Donner le code C++ d'une nouvelle fonction membre **`copie`** de la classe **`TableauDynamique`** qui copie le tableau de l'instance dans un tableau passé en paramètre, et dont l'entête est donné ci-dessous.

Procédure copie (tab : TableauDynamique)
Pré-condition : aucune
Post-condition : le tableau de l'instance est inchangé, le tableau tab en paramètre contient une copie du contenu du tableau de l'instance (c'est-à-dire que les adresses des pointeurs sont copiées)
Paramètre en mode donnée-résultat : tab

```
void TableauDynamique::copie (TableauDynamique & tab) const {
    delete [] tab.ad;
    tab.ad = new ElementTD [capacite];
    tab.capacite = capacite;
    tab.taille_utilisee = taille_utilisee;
    for (unsigned int i = 0; i < taille_utilisee; i++) tab.ad[i] = ad[i];
}

// ou bien vider tab puis ajouter les éléments de l'instance un à un dans tab
```

Cette fonction copie les adresses mémoires des éléments et non les éléments eux-mêmes. Après appel à cette fonction nous nous retrouvons donc avec deux tableaux dont les éléments pointent sur les mêmes emplacements en mémoire. Le changement d'une valeur via un tableau changera donc la valeur de l'autre tableau (les valeurs pointées sont « partagées »). Pour éviter ceci, et faire en sorte que les deux tableaux aient leurs propres valeurs, nous allons ajouter une fonctionnalité de copie dite profonde, c'est-à-dire qui copie (duplique) les valeurs pointées dans de nouveaux emplacements en mémoire (qui ont donc des adresses différentes). Afin de savoir combien d'octets doivent être alloués pour les éléments pointés d'un tableau dynamique, la fonction va prendre cette valeur en paramètre. L'entête de la fonction membre de copie profonde est la suivante :

Procédure copieProfonde (tab : TableauDynamique, nboctets : entier naturel)
Pré-condition : aucune
Post-condition : le tableau de l'instance est inchangé, le tableau tab en paramètre contient une copie profonde du du tableau de l'instance (i.e. les éléments pointés, de taille nboctets, sont dupliqués dans de nouveaux emplacements)
Paramètre en mode donnée : nboctets
Paramètre en mode donnée-résultat : tab

Question 9 : Compléter le code du programme principal ci-dessous permettant de tester la copie et la copie profonde.

```
#include "TableauDynamique.h"
#include <iostream>
using namespace std;

int main() {

    // Déclaration des tableaux
    TableauDynamique monTab;
    TableauDynamique monTabCopie;
    TableauDynamique monTabCopieProfonde;

    // Ajout de valeurs (pointeurs vers des int)
    cout << "Ajouts des valeurs 7 2 4" << endl;
    monTab.ajouterElement(new int(7));
    monTab.ajouterElement(new int(2));
    monTab.ajouterElement(new int(4));

    // Affichage monTab
    cout << "Tableau : ";
    monTab.afficher();

    // Copie de monTab dans monTabCopie
    monTab.copie(monTabCopie);

    // Affichage monTabCopie
    cout << "Tableau copie : ";
    monTabCopie.afficher();
```

```
    // Copie profonde de monTab dans monTabCopieProfonde
    monTab.copieProfonde(monTabCopieProfonde, sizeof(int));

    // Affichage monTabCopieProfonde
    cout << "Tableau copie profonde : ";
    monTabCopieProfonde.afficher();

    // Libération de la mémoire
    for (unsigned int i = 0; i < monTab.taille_utilisee; i++) {
        delete monTab.ad[i]; // ou monTabCopie
        delete monTabCopieProfonde.ad[i];
    }

    return 0;
}
```

Question 10 : Que se passe-t-il si l'on décide que les éléments d'un tableau dynamique sont maintenant des pointeurs vers des pointeurs ? Que peut-on alors faire ?

Si on a des pointeurs de pointeurs, alors la copie profonde réalise la copie des octets de l'adresse mémoire pointée. C'est cette adresse qui est copiée et non la valeur pointée. Nous nous retrouvons dans la situation précédente où la valeur va être « partagée » et non dupliquée.
Il faudrait réaliser une copie profonde des pointeurs de pointeurs, et de manière générale suivre les pointeurs jusqu'à ce que l'on trouve une « vraie » valeur (autre chose qu'un pointeur) et en réaliser une copie dans un nouvel emplacement mémoire.

Question 11 (bonus) : Donner le code C++ de la fonction **`copieProfonde`** qui réalise la copie profonde du tableau de l'instance dans un tableau passé en paramètre. La fonction prévoira uniquement la copie de valeurs qui font 1, 4 ou 8 octets (respectivement correspondant à la taille d'un **`char`**, d'un **`int`** et d'un **`double`**). Si le nombre d'octets demandé est différent alors la fonction réalisera une copie « simple » non profonde.
Pour rappel, l'opérateur **`*`** devant une adresse mémoire réalise le déréférencement du pointeur et retourne la valeur pointée et un changement de type, également possible sur les pointeurs, peut être réalisé par : `(nouveau_type) (valeur_dont_on_change_le_type)`. Par exemple : `(int *)(ptr)` convertit le type de `ptr` en un pointeur sur un `int`.

```
void TableauDynamique::copieProfonde (TableauDynamique & tab, unsigned int nboctets) const {
    copie(tab);
    for (unsigned int i = 0; i < taille_utilisee; i++) {
        switch (nboctets) {
            case 1 : tab.ad[i] = new char (*(char*)(ad[i])); break;
            case 4 : tab.ad[i] = new int (*(int*)(ad[i])); break;
            case 8 : tab.ad[i] = new double (*(double*)(ad[i])); break;
            default : break;
        }
    }
}
```

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;
```