Université de Nice-Sophia Antipolis — POLYTECH
ELEC4 — 2017–2018

Examen de COO

Durée : 1h
Aucun document autorisé
Mobiles interdits Nom : Prénom :

▶ 1. Expliquez de façon claire et synthétique le patron de conception « composite ».

Voir cours.

.....

▶ 2. Dessinez le diagramme de classes UML du patron composite.

Voir cours.

.....

Un arbre binaire est un ensemble de nœuds reliés par des arêtes. Chaque nœud possède un sous-arbre binaire gauche et droit, éventuellement vides. Un nœud sans aucun sous-arbre est une feuille. Les nœuds et les feuilles peuvent posséder une valeur, auquel cas l'arbre est étiqueté. Ci-dessous, un exemple d'arbre binaire étiqueté avec des valeurs entières.

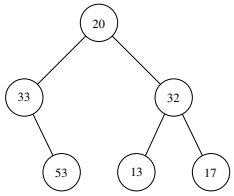

FIGURE 1 – Un exemple d'arbre binaire étiqueté

Les opérations de base sur les arbres binaires sont *valeur*, *sag*, *sad* qui renvoient, respectivement la valeur, le sous-arbre gauche et le sous-arbre droit du nœud courant. La constante *ArbreVide* représentera un arbre vide.

▶ 3. En utilisant le patron de conception composite, écrivez en C++ les classes

génériques qui permettent de définir un arbre binaire avec les opérations précédentes et la constante *ArbreVide*. Avec ces classes, on pourra construire l'arbre de la figure 1 comme suit :

```
ArbreBinaire<int> *f1 = new Feuille<int>(13);
ArbreBinaire<int> *f2 = new Feuille<int>(17);
ArbreBinaire<int> *n2 = new Noeud<int>(32, f1, f2);
ArbreBinaire<int> *f3 = new Feuille<int>(53);
ArbreBinaire<int> *n3 = new Noeud<int>(33,
                            ArbreBinaire<int>::ArbreVide, f3);
ArbreBinaire<int> *n1 = new Noeud<int>(20, n3, n2);
```

et l'instruction suivante écrira 32 sur la sortie standard :

```
std::cout << n1->sad()->valeur() << std::endl;
```

```
/*--------------------- A r b r e B i n a i r e-----------------*/
#pragma once

template<typename T>
class ArbreBinaire {
protected:
  T val;
public:
  static constexpr ArbreBinaire<T> *ArbreVide = nullptr;
  //
  virtual ~ArbreBinaire() {};
  ArbreBinaire(T x) : val(x) {};
  //
  const &T valeur() const {return this->val;}
  virtual ArbreBinaire<T> *sag() const =0;
  virtual ArbreBinaire<T> *sad() const =0;
};
/*------------------------ N o e u d --------------------------*/
#pragma once
#include <cassert>
#include "ArbreBinaire.hpp"

template<typename T>
class Noeud : public ArbreBinaire<T> {
private:
  typedef ArbreBinaire<T> super;
  ArbreBinaire<T> *g;
  ArbreBinaire<T> *d;
public:
  Noeud(const T &x, ArbreBinaire<T> *g, ArbreBinaire<T> *d)
    : ArbreBinaire<T>(x)
  {
    assert(g!= super::ArbreVide || d!=super::ArbreVide);
    this->g = g;
    this->d = d;
  }
  ~Noeud() {delete this->g; delete this->d;}
```

```
  //
  ArbreBinaire<T> *sag() const override { return this->g; }
  ArbreBinaire<T> *sad() const override { return this->d; }
};

/*----------------------- F e u i l l e -----------------------*/
#pragma once

#include "ArbreBinaire.hpp"
#include "ArbreVideException.hpp"

template<typename T>
class Feuille : public ArbreBinaire<T> {
private:
  typedef ArbreBinaire<T> super;
public:
  Feuille(const T & x) : ArbreBinaire<T>(x) {}
  ArbreBinaire<T> *sag() const override {
      throw ArbreVideException();
  }
  ArbreBinaire<T> *sad() const override {
      throw ArbreVideException();
  }
};
```

.....

▶ 4. Complétez vos classes avec la méthode `parcours` qui parcourt (de façon infixe, *i.e.* GND) l'arbre courant et applique une procédure `traiter` à chacun des nœuds visités. La procédure `traiter` est un paramètre formel de la méthode `parcours` et c'est une fonction anonyme (lambda).

```
/*---------------------- A r b r e B i n a i r e-----------------*/
#include <functional>
template<typename T>
class ArbreBinaire {
// .... comme précédemment
public:
  virtual void parcours(std::function<void(T)> traiter) const =0;
};
/*------------------------ N o e u d --------------------------*/
#include <functional>
template<typename T>
class Noeud : public ArbreBinaire<T> {
// .... comme précédemment
public:
    void parcours(std::function<void(T)> traiter) const override {
    // parcours du sous-arbre gauche du noeud courant
    if (this->sag() != super::ArbreVide)
            this->sag()->parcours(traiter);
    // traiter le noeud courant
    traiter(this->valeur());
```



```
    // parcours du sous-arbre droit du noeud courant
    if (this->sad() != super::ArbreVide)
            this->sad()->parcours(traiter);
  }
};

/*----------------------- F e u i l l e -----------------------*/
#include <functional>
template<typename T>
class Feuille : public ArbreBinaire<T> {
// .... comme précédemment
public:
  void parcours(std::function<void(T)> traiter) const override {
    // traiter la feuille courante
    traiter(super::valeur());
  }
};
```

.....

▶ 5. Avec la méthode `parcours` précédente, écrivez l'instruction qui écrit chacun des nœuds de l'arbre de la figure 1 sur la sortie standard. Le résultat obtenu est 33 53 20 13 32 17.

```
n1->parcours([] (int x) -> void {std::cout << x << " ";});
```

.....

▶ 6. Expliquez de façon claire et synthétique le patron de conception « itérateur ».

Voir cours.

.....

▶ 7. Ajoutez un itérateur à la classe `ArbreBinaire` et donnez un exemple de son utilisation.

```
#include <iterator>
#include <vector>
template<typename T>
class ArbreBinaire {
  // .... comme précédemment
  //
  class iterator :
        public std::iterator<std::forward_iterator_tag,T> {
  protected:
    int pos;
    std::vector<T> v;
```

```
  public:
    iterator(ArbreBinaire<T> *a, bool first) {
      a->parcours([this] (T x) -> void { v.push_back(x); });
      pos = (first) ? 0 : v.size();
    }
    //incrémentation préfixe
    iterator &operator++() { this->pos++;  return *this; }
    // incrémentation postfixe
    iterator operator++(int) {
         iterator i=*this; this->pos++; return i;
    }
    // add
    iterator &operator+(int n) { pos+=n;  return *this; }
    // accès
    T &operator*()   { return this->v[pos]; }
    T* operator->()  { return this->v[pos]; }
    bool operator==(const iterator &it) const {
        return this->pos == it.ps;
    }
    bool operator!=(const iterator &it) const {
        return this->pos != it.pos;
    }
  };

  iterator begin() noexcept {return iterator(this,true);}
  iterator end() noexcept {return iterator(this,false);}
};
```

Un exemple d'utilisation :

```
for (ArbreBinaire<int>::iterator it = n1->begin();
                                 it != n1->end(); it++)
    std::cout << *it << " ";
```

mais qui permet d'écrire aussi :

```
for (int x : *n1) std::cout << x << " ";
```

.....