Algorithmes dans les arbres binaires [tn03] - Exercices

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - Algorithmes dans les arbres binaires (Solution)

Mots-Clés Structures arborescentes ■
Requis Axiomatique objet, Modèles, Gestion des exceptions ■
Difficulté ●●○ (2 h) ■

Objectif
Cet exercice réalise les opérations de base des arbres binaires (construction, affichage, copie, calcul de la profondeur, traversées).

1 Noeud Binaire

1.1 Classe TNoeudBinaire

Écrivez une classe `TNoeudBinaire<T>` qui représente un noeud d'un arbre binaire d'éléments de type `T`.

Validez votre classe avec la solution.

Solution C++ @[TNoeudBinaire.hpp]

```
#ifndef TNOEUD_BINAIRE_CLASS
#define TNOEUD_BINAIRE_CLASS
#ifndef NULL
#include <cstddef>
#endif
/** Noeud dans un arbre binaire */
template<typename T>
class TNoeudBinaire
{
   public:
    // Constructeur par defaut
    TNoeudBinaire()
    : left(NULL), right(NULL)
    {}

    // Constructeur issu d'une valeur
    TNoeudBinaire(const T& item)
    : data(item), left(NULL), right(NULL)
    {}

    // Constructeur normal
    TNoeudBinaire(const T& item, TNoeudBinaire<T>* lptr, TNoeudBinaire<T>* rptr)
    : data(item), left(lptr), right(rptr)
    {}

    // Champs public
    T data; //! donnee
    TNoeudBinaire<T> *left; //! fils gauche
    TNoeudBinaire<T> *right; //! fils droit
};
#endif
```

1.2 Classe TNoeudShadow

Écrivez une classe `TNoeudShadow<T>` qui maintient la donnée de type `T` d'un noeud d'un arbre binaire sous forme de chaîne de caractères.

Validez votre classe avec la solution.

Solution C++ @[TNoeudShadow.hpp]

```
#ifndef TNOEUD_SHADOW_CLASS
#define TNOEUD_SHADOW_CLASS
#include <string>
#include <iostream>
#include <iomanip>
#include <queue>
#ifndef NULL
#include <cstddef>
#endif
using namespace std;
/**
  Maintient la donnee (sous forme de chaine de caracteres)
*/
class TNoeudShadow
{
  public:
    string dataStr; // donnee formatee
    TNoeudShadow *left; // filsgauche
    TNoeudShadow *right; // filsdroit
    int level; // niveau
    int column; // numero de colonne

    TNoeudShadow()
    : left(NULL), right(NULL), level(0), column(0)
    {}
};

/**
  Trace un arbre Shadow enracine
  @param[in] t - noeud racine de l'arbre
  @param[in] maxChars - valeur affichee sur au plus maxChars
*/
void drawShadowTree(TNoeudShadow *shadowRoot, int maxChars)
{
  string label;
  const int colWidth = maxChars + 1;
  int currLevel = 0;
  int currCol = 0;
  // use during the level order scan of the shadow tree
  TNoeudShadow *currNode;

  // Traverse par niveau le shadowArbre (utilisation d'une file)
  // store siblings of each tnodeShadow object in a queue so that
  // they are visited in order at the next level of the tree
  queue<TNoeudShadow*> q;
  // insert the root in the queue and set current level to 0
  q.push(shadowRoot);
  // continue the iterative process until the queue is empty
  while (not q.empty())
  {
    // delete front node from queue and make it the current node
    currNode = q.front();
    q.pop();

    // si les niveaux changent, saut de ligne
    if (currNode->level > currLevel)
    {
```

```
      currLevel = currNode->level;
      currCol = 0;
      cout << endl;
    }
    // enfile le fils gauche, s'il existe
    if (currNode->left != NULL)
    {
      q.push(currNode->left);
    }
    //  enfile le fils droit, s'il existe
    if (currNode->right != NULL)
    {
      q.push(currNode->right);
    }
    // Affiche la donnee formattee
    if (currNode->column > currCol)
    {
      cout << setw((currNode->column-currCol)*colWidth) << " ";
      currCol = currNode->column;
    }
    cout << setw(colWidth) << currNode->dataStr;
    ++currCol;
  }
  cout << endl << endl;
}

/**
  Supprime un arbre Shadow enracine
  @param[in] t - un TNoeudShadow
*/
void deleteShadowTree(TNoeudShadow* t)
{
  if (t != NULL)
  {
    deleteShadowTree(t->left);
    deleteShadowTree(t->right);
    delete t;
  }
}
#endif
```

2 Opérations de base

2.1 Opérations de traversée

Écrivez une procédure `affichageParNiveau(t,sep)` qui réalise l'affichage par niveau des noeuds d'un arbre binaire enraciné en `t`, le paramètre `sep` étant le séparateur entre les valeurs.

Validez votre procédure avec la solution.

Solution C++ @[tnaffichNiv.cpp]

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;
#include "TNoeudBinaire.hpp"

/**
  Affichage par niveau des noeuds d'un arbre binaire
  @param[in] t - un noeud binaire
  @param[in] sep - le separateur
*/
template <typename T>
void affichageParNiveau(TNoeudBinaire<T>* t, const string& sep = " ")
{
   // Memorise les freres de chaque noeud
   queue<TNoeudBinaire<T>* > q;
   TNoeudBinaire<T> *p;
   // Initialise la file en inserant la racine
   q.push(t);
   // Continue le processus iteratif jusqu'a ce que la file soit vide
   while(not q.empty())
   {
      // Supprime la tete et affiche la valeur du noeud
      p = q.front();
      q.pop();
      cout << p->data << sep;
      // Insere le fils gauche dans la file (s'il existe)
      if (p->left != NULL)
      {
        q.push(p->left);
      }
      // Insere le fils droit dans la file (s'il existe)
      if (p->right != NULL)
      {
        q.push(p->right);
      }
   }
}
```

Écrivez les procédures récursives `affichagePrefixe(t,sep)`, `affichageInfixe(t,sep)` et `affichagePostfixe(t` qui réalisent l'affichage préfixé, infixé et suffixé des noeuds d'un arbre binaire enraciné en `t`, le paramètre `sep` étant le séparateur entre les valeurs.

Validez vos procédures avec la solution.

Solution C++ @[tnaffichRec.cpp]

```
#include <iostream>
#include <string>
using namespace std;
#include "TNoeudBinaire.hpp"

/**
  Affichage recursif prefixe des noeuds d'un arbre binaire
  @param[in] t - un noeud binaire
  @param[in] sep - le separateur
*/
template<typename T>
void affichagePrefixe(TNoeudBinaire<T>* t, const string& sep = " ")
{
   if (t != NULL)
   {
      cout << t->data << sep;
      affichagePrefixe(t->left, sep);
      affichagePrefixe(t->right, sep);
   }
}

/**
  Affichage recursif infixe des noeuds d'un arbre binaire
  @param[in] t - un noeud binaire
  @param[in] sep - le separateur
*/
template<typename T>
void affichageInfixe(TNoeudBinaire<T>* t, const string& sep = " ")
{
   if (t != NULL)
   {
      affichageInfixe(t->left, sep);
      cout << t->data << sep;
      affichageInfixe(t->right, sep);
   }
}

/**
  Affichage recursif postfixe des noeuds d'un arbre binaire
  @param[in] t - un noeud binaire
  @param[in] sep - le separateur
*/
template <typename T>
void affichagePostfixe(TNoeudBinaire<T>* t, const string& sep = " ")
{
   if (t != NULL)
   {
      affichagePostfixe(t->left, sep);
```

```
      affichagePostfixe(t->right, sep);
      cout << t->data << sep;
   }
}
```

2.2 Calcul de la profondeur

Écrivez une fonction récursive `profondeur(t)` qui calcule et renvoie la profondeur d'un arbre binaire enraciné en `t`.

Validez votre fonction avec la solution.

Solution C++ @[tndepth.cpp]

```
#include "TNoeudBinaire.hpp"
/**
  Profondeur d'un arbre binaire (parcours postordre)
  @param[in] t - un noeud binaire
  @return la Profondeur de t
*/
template <typename T>
int profondeur(TNoeudBinaire<T>* t)
{
  int depthLeft, depthRight, depthval;
  // Si l'arbre vide est vide, sa profondeur est -1
  if (t == NULL)
  {
    depthval = -1;
  }
  else
  {
    // calcule la profondeur du sous-arbre gauche de t
    depthLeft = profondeur(t->left);
    // calcule la profondeur du sous-arbre droit de t
    depthRight = profondeur(t->right);
    // la profondeur de l'arbre enracine en t est 1 + max(profondeurs)
    depthval = 1 + (depthLeft > depthRight ? depthLeft : depthRight);
  }
  return depthval;
}
```

2.3 Calcul du nombre de feuilles

Écrivez une fonction récursive `nfeuilles(t)` qui calcule et renvoie le nombre de feuilles d'un arbre binaire enraciné en `t`.

Validez votre fonction avec la solution.

Solution C++ @[tnleaves.cpp]

```
#include "TNoeudBinaire.hpp"
/**
  Accumule recursivement le nombre de feuilles
  @param[in] t - un noeud binaire
  @param[in,out] count - compteur du nombre de feuilles
*/
template <typename T>
void nfeuillesRec(TNoeudBinaire<T>* t, int& count)
{
  if (t != NULL)
  {
    // Si pas de fils, incremente le nombre de feuilles
    if (t->left == NULL and t->right == NULL)
    {
      count += 1;
    }
    // Appelle recursif sur la gauche
    nfeuillesRec(t->left, count);
    // Appelle recursif sur la droite
    nfeuillesRec(t->right, count);
  }
}

/**
  Nombre de feuilles d'un arbre binaire
  @param[in] t - un noeud binaire
  @return le nombre de feuilles de t
*/
template <typename T>
int nfeuilles(TNoeudBinaire<T>* t)
{
  int nf = 0;
  nfeuillesRec(t, nf);
  return nf;
}
```

2.4 Opérations de manipulation

Écrivez une fonction récursive `copierArbre(t)` qui crée une copie d'un arbre binaire enraciné en `t` et renvoie un pointeur sur la racine.

Écrivez une procédure récursive `supprimerArbre(t)` qui supprime chacun des noeuds d'un arbre binaire enraciné en `t`.

Déduisez une procédure `viderArbre(t)` qui supprime chacun des noeuds d'un arbre binaire enraciné en `t` puis assigne `t` à l'arbre vide.

Validez vos opérations avec la solution.

Solution C++ @[tnmanip.cpp]

```cpp
#include "TNoeudBinaire.hpp"
/**
  Cree une copie d'un arbre enracine
  @param[in] t - un noeud binaire
  @return un pointeur sur la racine
*/
template <typename T>
TNoeudBinaire<T>* copierArbre(TNoeudBinaire<T>* t)
{
  // Arret sur un noeud vide
  if (t == NULL)
  {
    return NULL;
  }
  else
  {
    // Realise une copie du sous-arbre gauche et du sous-arbre droit
    TNoeudBinaire<T>* newLeft = copierArbre(t->left);
    TNoeudBinaire<T>* newRight = copierArbre(t->right);
    // Cree un nouveau noeud avec la valeur de t et assigne ses fils
    TNoeudBinaire<T>* newNode = new TNoeudBinaire<T>(t->data, newLeft, newRight);
    // Renvoie le pointeur sur le noeud
    return newNode;
  }
}

/**
  Supprime un arbre enracine (Traverse en postordre)
  @param[in] t - un noeud binaire
*/
template <typename T>
void supprimerArbre(TNoeudBinaire<T>* t)
{
   if (t != NULL)
   {
      supprimerArbre(t->left);
      supprimerArbre(t->right);
      delete t;
   }
}

/**
  Vide un arbre enracine en t puis assigne t à NULL
  @param[in,out] t - un noeud binaire
*/
template <typename T>
void viderArbre(TNoeudBinaire<T>*& t)
{
  supprimerArbre(t);
  t = NULL;
}
```

3 Programmes de test

3.1 Procédure de construction d'un arbre

Validez votre procédure avec la solution.

Solution C++ @[construireArbre.cpp]

```
#ifndef CONSTRUIRE_ARBRE
#define CONSTRUIRE_ARBRE
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <queue>
#ifndef  NULL
#include <cstddef>
#endif
using namespace std;
#include "TNoeudBinaire.hpp"

/**
  Cree un arbre binaire avec des caractères
  @param[in] n - numero de l'arbre (arbre 0 -- arbre 2)
  @return le pointeur sur la racine
*/
TNoeudBinaire<char>* construireArbre(int  n)
{
  // 9 pointeurs
  TNoeudBinaire<char> *root, *b, *c, *d, *e, *f, *g, *h, *i;

  switch(n)
  {
    // noeuds d et e sont des feuilles
    case  0:
      d = new TNoeudBinaire<char>('D');
      e = new TNoeudBinaire<char>('E');
      b = new TNoeudBinaire<char>('B',(TNoeudBinaire<char>*)NULL,d);
      c = new TNoeudBinaire<char>('C',e,(TNoeudBinaire<char>*)NULL);
      root = new TNoeudBinaire<char>('A',b,c);
      break;

    // noeuds g, h, i, d sont des feuilles
    case  1:
      g = new TNoeudBinaire<char>('G');
      h = new TNoeudBinaire<char>('H');
      i = new TNoeudBinaire<char>('I');
      d = new TNoeudBinaire<char>('D');
      e = new TNoeudBinaire<char>('E',g,(TNoeudBinaire<char>*)NULL);
      f = new TNoeudBinaire<char>('F',h,i);
      b = new TNoeudBinaire<char>('B',d,e);
      c = new TNoeudBinaire<char>('C',(TNoeudBinaire<char>*)NULL,f);
      root = new TNoeudBinaire<char>('A',b,c);
      break;

    // nodes g, h, i, f sont des feuilles
```

```
    case 2:
      g = new TNoeudBinaire<char>('G');
      h = new TNoeudBinaire<char>('H');
      i = new TNoeudBinaire<char>('I');
      d = new TNoeudBinaire<char>('D',(TNoeudBinaire<char>*)NULL,g);
      e = new TNoeudBinaire<char>('E',h,i);
      f = new TNoeudBinaire<char>('F');
      b = new TNoeudBinaire<char>('B',d,(TNoeudBinaire<char>*)NULL);
      c = new TNoeudBinaire<char>('C',e,f);
      root = new TNoeudBinaire<char>('A',b,c);
      break;
  }
  return  root;
}
#endif
```

3.2 Procédure d'affichage d'un arbre

Validez votre procédure avec la solution.

Solution C++ @[displayShadowTree.cpp]

```
#ifndef DISPLAY_SHADOW_TREE
#define DISPLAY_SHADOW_TREE
#include <sstream>
#ifndef NULL
#include <cstddef>
#endif
using namespace std;
#include "TNoeudBinaire.hpp"
#include "TNoeudShadow.hpp"

/**
  Construit un ShadowTree (traverse en infixe)
  @param[in] t - noeud binaire
  @param[in] level - niveau actuel
  @param[in,out] column - indice de colonne
  @return l'Arbre Shadow equivalent à celui enracine en t
*/
template<typename T>
TNoeudShadow* buildShadowTree(const TNoeudBinaire<T>* t, int level, int& column)
{
  // pointeur vers le nouveau noeud du ShadowArbre
  TNoeudShadow *newNode = NULL;
  // ostr est utilise pour la conversion
  ostringstream ostr;
  if (t != NULL)
  {
    // Cree un nouveau noeud du shadowArbre
    newNode = new TNoeudShadow;
    // Alloue le fils gauche au niveau suivant et l'attache au noeud
    TNoeudShadow *newLeft = buildShadowTree(t->left, level+1, column);
    newNode->left = newLeft;
    // initialise les donnees membres du noeud
    ostr << t->data << ends;
```

```
    newNode->dataStr = ostr.str();
    newNode->level = level;
    newNode->column = column;
    // incremente le numero de colonnes
    ++column;
    // Alloue le fils droit au niveau suivant et l'attache au noeud
    TNoeudShadow *newRight = buildShadowTree(t->right, level+1, column);
    newNode->right = newRight;
  }
  return newNode;
}

/**
  Affiche un arbre binaire enracine
  @param[in] t - noeud racine de l'arbre
  @param[in] maxChars - valeur affichee sur au plus maxChars
*/
template <typename T>
void displayShadowTree(const TNoeudBinaire<T>* t, int maxChars)
{
  if (t == NULL)
  {
    return;
  }
  // Construit le shadowArbre
  int level = 0;
  int column = 0;
  TNoeudShadow *shadowRoot = buildShadowTree(t, level, column);
  // Trace le shadowArbre
  drawShadowTree(shadowRoot, maxChars);
  // Supprime le ShadowArbre
  deleteShadowTree(shadowRoot);
}
#endif
```

3.3 Copie d'un arbre

Validez votre programme avec la solution.

Solution C++ @[pgtncopy.cpp]

```
#include <iostream>
using  namespace  std;
#include "TNoeudBinaire.hpp"
#include "construireArbre.cpp"
#include "displayShadowTree.cpp"
#include "TNoeudFuncs.cpp"

int  main()
{
  int n;
  cout << "Numero de l'arbre (0,1,2)? ";
  cin >> n;
  // Construit l'arbre n
  TNoeudBinaire<char> *root = construireArbre(n);
```

```
  // Affiche l'arbre
  cout  <<  "Arbre originel" << endl;
  displayShadowTree(root, 1);
  cout << endl << endl;
  // Realise une copie de l'arbre enracine en root
  TNoeudBinaire<char> *root2 = copierArbre(root);
  // Affiche la copie
  cout <<  "Copie de l'arbre" << endl;
  displayShadowTree(root2, 1);
  cout << endl;
  // Supprime les noeuds des arbres
  viderArbre(root2);
  viderArbre(root);
}
```

3.4 Profondeur d'un arbre

Validez votre programme avec la solution.

Solution C++ @[pgtndepth.cpp]

```
// Nombre de feuilles et profondeur d'un arbre
#include <iostream>
using namespace std;
#include "TNoeudBinaire.hpp"
#include "construireArbre.cpp"
#include "displayShadowTree.cpp"
#include "TNoeudFuncs.cpp"

int main()
{
  int n;
  cout << "Numero de l'arbre (0,1,2)? ";
  cin >> n;
  TNoeudBinaire<char> *root = construireArbre(n);
  cout  <<  "Arbre originel" << endl;
  displayShadowTree(root, 1);
  cout << endl << endl;
  cout << "Nombre de feuilles = " << nfeuilles(root) << endl;
  cout << "Profondeur de l'arbre = " << profondeur(root) << endl;
  viderArbre(root);
}
```

3.5 Traversées d'un arbre

Validez votre programme avec la solution.

Solution C++ @[pgtntraverse.cpp]

```
// Realise les traverses (recursives et par-niveau)
using namespace std;
#include <iostream>
```

```
#include "TNoeudBinaire.hpp"
#include "construireArbre.cpp"
#include "displayShadowTree.cpp"
#include "TNoeudFuncs.cpp"
int main()
{
  int n;
  cout << "Numero de l'arbre (0,1,2)? ";
  cin >> n;
  TNoeudBinaire<char> *root = construireArbre(n);
  cout  <<  "Arbre originel" << endl;
  displayShadowTree(root, 1);
  cout << endl << endl;
  cout << "Parcours prefixe:    " ;
  affichagePrefixe(root);
  cout << endl;
  cout << "Parcours infixe:     " ;
  affichageInfixe(root);
  cout << endl;
  cout << "Parcours postfixe:   " ;
  affichagePostfixe(root);
  cout << endl;
  cout << "Parcours par-niveau: " ;
  affichageParNiveau(root);
  cout << endl;
  viderArbre(root);
}
```

4 Références générales

Comprend [] ■