Árboles Binarios

Accel Israel Magaña Rodrgiuez

Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv17a.amagana@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Mayo 18, 2019

I. Árbol Binario Básico

Un árbol T es un grafo simple que cumple que entre los vértices v, wT existe un único camino simple.

Un árbol binario es un tipo de árbol en que cada vértice máximo puede tener dos hijos; su nodo raíz está enlazado a dos subárboles binarios disjuntos denominados subárbol izquierdo y subárbol derecho. Los árboles binarios no son vacíos ya que como mínimo tienen el nodo raíz.

Representación de árboles

Los árboles binarios pueden representarse en un vector o en una lista ligada. Nuestro interés se centrará en los vectores.

Para representar a un árbol binario en un vector se escriben por niveles los nodos del árbol de manera ordenada, de izquierda a derecha (hijo izquierdo — hijo derecho).

Esta representación es poco eficiente cuando el árbol no es completo, en vista del gran desperdicio de memoria que podría haber por las posiciones libres que quedarían en el vector.

Si k es la posición en un vector de un nodo de un árbol con n niveles, se tiene:

- El vector tendrá 2n+1 -1 posiciones.
- El padre estará en la posición k/2 (con k>1)
- El hijo izquierdo estará en la posición 2*I (con 2*k<n) y el hijo derecho en la posición 2*k+1.

Recorridos de Árboles

Recorrido **INORDEN**. Este recorrido se realiza así: primero recorre el subárbol izquierdo, segundo visita la raíz y por último, va al subárbol derecho. En síntesis:

hijo izquierdo — raíz — hijo derecho

Recorrido **PREORDEN**. Este recorrido se realiza así: primero visita la raíz; segundo recorre el subárbol izquierdo y por último va a subárbol derecho. En síntesis:

raíz — hijo izquierdo — hijo derecho

Recorrido **POSTORDEN**. Primero recorre el subárbol izquierdo; segundo, recorre el subárbol derecho y por último, visita la raíz. En síntesis:

hijo izquierdo– hijo derecho — raíz

Implementación en código C++:

Para el árbol completo con todo y las hojas utilicé un patrón de clases anidadas, ya que no quiero que las hojas se traten como independientes.
Además implementé un sistema de templates, por lo que en teoría debería ser posible insertar cualquier tipo de dato. Sin embargo, el test es únicamente con números y letras individuales.

```
template <typename T>
class BinaryTree
{
 /**
  * @brief Nested class that contains the leaf
  */
 template <typename T>
 class NNode
 {
 public:
  NNode(T _data): m_data(_data) {}

  /**
   * @brief insert recursive function
   *
   * NOTE: it doesn't re-balance the tree yet
   */
  inline void
  insert(T data) {

    if (data < m_data)
    {
      if (nullptr ==  m_left)
      {
        m_left = new NNode<T>(data);
        m_left->m_parent = this;
      }
      else
      {
        m_left->insert(data);
      }
    }
```

```
  else
  {
   if (nullptr == m_right)
   {
    m_right = new NNode<T>(data);
    m_right->m_parent = this;
   }
   else
   {
    m_right->insert(data);
   }
  }
}

void
inOrder() {
  if (nullptr != m_left)
  {
   m_left->inOrder();
  }
  std::cout << m_data << ", ";
  if(nullptr !=  m_right)
  {
   m_right->inOrder();
  }
}

void
postOrder() {
  if(nullptr != m_left)
  {
   m_left->preOrder();
  }
  if(nullptr != m_right)
  {
   m_right->preOrder();
  }
  std::cout << m_data << ", ";
}

void
preOrder() {
  std::cout << m_data << ", ";
  if (nullptr != m_left )
  {
   m_left->preOrder();
  }
  if(nullptr != m_right)
```

```
  {
    m_right->preOrder();
  }
}

NNode<T>*
find(T data) {
  if (m_data == data)
  {
    return this;
  }
  else if(data > m_data)
  {
    return m_right->find(data);
  }
  else if(data < m_data)
  {
    return m_left->find(data);
  }
  else
  {
    //TODO: error, data not found
    return nullptr;
  }
}

NNode<T>*
getParent() const {
  return m_parent;
}

NNode<T>*
getLeft() {
  return m_left;
}

NNode<T>*
getRight() {
  return m_right;
}

void
setLeft(NNode<T>* node) {
  m_left = node;
}

void
setRight(NNode<T>* node) {
```

```
    m_right = node;
  }

  void
  setParent(NNode<T>* node) {
    m_parent = node;
  }

  NNode<T>*
  getLastLeft() {
    if (nullptr != m_left)
    {
      return m_left->getLastLeft();
    }
    return this;
  }

  inline T
  getData() const { return m_data; }

 private:
  T m_data;
  NNode<T>* m_left = nullptr;
  NNode<T>* m_right = nullptr;
  NNode<T>* m_parent;
};

public:
/*
 * @brief Default constructor
 */
 BinaryTree() = default;

/*
 * @brief Default destructor
 */
 ~BinaryTree();

 void
 initialize(T);

 void
 insert(T);

 void
 remove(T);
```

```
  void
  printInOrder() {
    if (nullptr != m_root)
    {
      std::cout << "InOrder() :\t" ;
      m_root->inOrder();
      std::cout << "\n\n";
    }
  }

  void
  printPostOrder() {
    if(nullptr != m_root)
    {
      std::cout << "PostOrder() :\t";
      m_root->postOrder();
      std::cout << "\n\n";
    }
  }

  void
  printPreOrder()
  {
    if(nullptr != m_root)
    {
      std::cout << "PreOrder() :\t";
      m_root->preOrder();
      std::cout << "\n\n";
    }
  }

private:

  inline NNode<T>*
  find(const T& data) {
    return m_root->find(data);
  }


 private:

   NNode<T>* m_root;

};

/*
*/
template <typename T>
```

```
void
BinaryTree<T>::remove(T data)
{
  NNode<T>* d = m_root->find(data);
  NNode<T>* dR = d->getRight();
  NNode<T>* dL = d->getLeft();

  if(d != nullptr) {
    NNode<T>* parent = d->getParent();

    if (nullptr == dR && nullptr == dL)
    {
      (parent->getLeft() == d) ? parent->setLeft(nullptr) : parent->setRight(nullptr);
    }
    else if(nullptr != dR && nullptr != dL)
    {
      NNode<T>* nP = dR->getLastLeft();
      (parent->getLeft() == d) ? parent->setLeft(dR) : parent->setRight(dR);

      dL->setParent(nP);
      nP->setLeft(dL);
    }
    else if(nullptr != dL)
    {
      dL->setParent(parent);
      (parent->getLeft() == d) ? parent->setLeft(dL) : parent->setRight(dL);
    }
    else if (nullptr != dR)
    {
      dR->setParent(parent);
      (parent->getLeft() == d) ? parent->setLeft(dR) : parent->setRight(dR);
    }
  }
}

/*
*/
template <typename T>
void
BinaryTree<T>::insert(T data)
{
  if (nullptr != m_root)
  {
    m_root->insert(data);
  }
  else
  {
    m_root = new NNode<T>(data);
```

```
   }
}

/*
*/
template <typename T>
void
BinaryTree<T>::initialize(T data)
{
  m_root = new NNode<T>(data);
}

/*
*/
template <typename T>

BinaryTree<T>::~BinaryTree()
= default;
```