PRO2: BinTree

8 de marzo de 2024

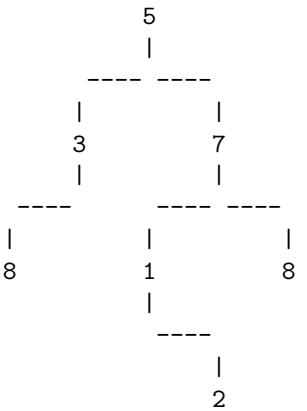
1. Árboles binarios: idea intuitiva

En el contexto de la asignatura PRO2, un árbol binario es una estructura jerárquica que permite guardar elementos de un cierto tipo prefijado.

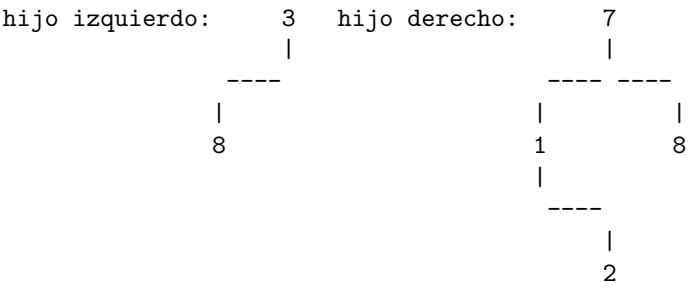
En el caso en que un árbol binario no tenga ningún elemento, le llamamos el árbol vacío.

En el caso de que un árbol binario tenga uno o más elementos, hay un elemento concreto, el más prioritario, que se halla en la raíz del árbol. De la raíz, cuelgan dos subárboles, el subárbol izquierdo y el subárbol derecho, que a su vez pueden ser vacíos o contener más elementos.

La siguiente es una representación visual, más o menos intuitiva, de un árbol binario que almacena enteros.

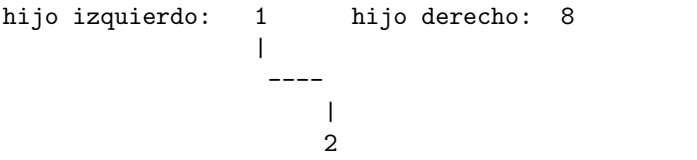


En la raíz del árbol se guarda el valor 5. Los hijos izquierdo y derecho del árbol son los siguientes:



El hijo izquierdo del árbol inicial guarda un 3 en la raíz y tiene, a su vez, un hijo derecho vacío (que aquí representamos con el string vacío), y tiene también un hijo izquierdo con un 8 en la raíz y dos hijos vacíos.

El hijo derecho del árbol inicial guarda un 7 en la raíz y tiene, a su vez, los siguientes dos hijos:



El primero de los dos últimos subárboles dibujados guarda un 1 en la raíz y tiene, a su vez, un hijo izquierdo vacío, y un hijo derecho con un 2 en la raíz y dos subárboles vacíos.

El segundo de los dos últimos subárboles dibujados guarda un 8 en la raíz y tiene, a su vez, dos subárboles vacíos.

2. Árboles binarios: definición formal

El conjunto de los árboles binarios de un tipo fijado `T` se pueden definir inductivamente así:

- El árbol binario vacío (sin elementos) és un árbol binario de tipo `T`. Lo denotamos mediante `()`, o mediante el string vacío.
- Dados dos árboles binários t_1, t_2 de tipo `T`, y un elemento `e` de tipo `T`, `e(`t_1`,`t_2`)` es un árbol binario.

Por ejemplo, $5(3(8(,),),7(1(,2(,)),8(,)))$ es un árbol que almacena elementos de tipo `T = int`.

Para simplificar la representación, cuando un subárbol tiene sus dos correspondientes subárboles vacíos, podemos simplemente escribir su raíz y nada más. De este modo, el árbol anterior también se puede representar así: $5(3(8,),7(1(,2),8))$

Como habréis notado, estamos representando el mismo árbol que presentábamos en la sección anterior de forma más visual e intuitiva.

3. El tipo `BinTree`

En el contexto de PRO2, incluiremos en nuestros programas el fichero `BinTree.hh`, que define la clase parametrizable `BinTree`, la cual nos permite crear y manejar árboles binarios.

```
#include "BinTree.hh"
```

Si, por ejemplo, nos interesa trabajar con árboles binarios de enteros, podemos definir el tipo de datos `BT` para representarlos, del siguiente modo:

```
typedef BinTree<int> BT;
```

A partir de ese momento, cualquier variable declarada con tipo `BT` es un árbol binario de enteros. Si declaramos una variable `BT` sin parámetros, será un árbol vacío. En cambio, si declaramos una variable `BT` con un parámetro entero `e` y dos parámetros BT `t1, t2`, entonces esa variable será un árbol no vacío, con `e` en la raíz y `t1` como hijo izquierdo y `t2` como hijo derecho. Observad el siguiente ejemplo:

```
#include "BinTree.hh"

typedef BinTree<int> BT;

int main()
{
  BT t1;              // t1 es un árbol vacío
  BT t2(8, t1, t1)    // t2 = 8(,) = 8 (arbol con 8 en la raíz
                                       y dos subárboles vacíos)
  BT t3(3, t2, t1)    // t3 = 3(8,)
  BT t4(2, t1, t1)    // t4 = 2(,) = 2
  BT t5(1, t1, t4)    // t5 = 1(,2)
  BT t6(7, t5, t2)    // t6 = 7(1(,2),8)
  BT t7(5, t3, t6)    // t7 = 5(3(8,),7(1(,2),8))
  ...
}
```

Como véis, la secuéncia de instrucciones acaba almacenando el árbol de ejemplo de las secciones anteriores en la variable `t7`.

`BinTree` tiene métodos que devuelven el valor de la raíz (`value`), y el valor de los hijos izquierdo y derecho (`left` y `right`).

Por ejemplo, podemos añadir las siguientes instrucciones a continuación del programa anterior:

```
  int x = t7.value();               // x = 5
  BT t8 = t7.left();                // t8 = 3(8,)
```

```
BT t9 = t7.right();                // t9 = 7(1(,2),8)
BT t10 = t9.left();                // t10 = 1(,2)
BT t11 = t7.right().left();        // t11 = 1(,2)
int y = t7.right().left().value(); // y = 1
int z = t11.value();               // z = 1
```

Pero los árboles binarios son inmutables, es decir, no los podemos modificar:

```
t8.value() = 11;          // COMPILATION ERROR
t8.left() = BT(t3, t4);   // COMPILATION ERROR
t8.right() = BT(t1, t10); // COMPILATION ERROR
```

También podemos preguntar si un árbol binario es vació:

```
cout << t1.empty() << endl; // output: 1 (true)
cout << t2.empty() << endl; // output: 0 (false)
```

4. Escritura y lectura de árboles binarios

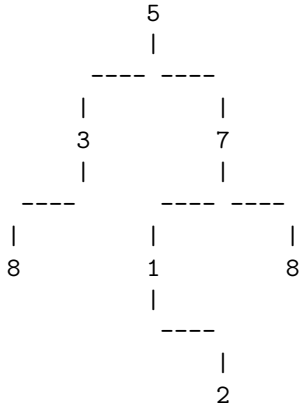
Las operaciones >> y << están sobrecargadas de modo que podemos leer y escribir árboles binarios. Estos se pueden escribir en un formato inline como el que hemos estado usando en las secciones anteriores para representarlos. Por ejemplo, siguiendo con el ejemplo anterior:

```
...
  t7.setOutputFormat(BT::INLINEFORMAT);
  cout << t7 << endl; // output: 5(3(8,),7(1(,2),8))
```

Pero también los podemos escribir con el formato visual, más intuitivo, que veíamos en la primera sección:

```
...
  t7.setOutputFormat(BT::VISUALFORMAT);
  cout << t7 << endl;
```

y eso producirá esta salida:



También podemos leer árboles de la entrada con formatos inline y visual. Sin embargo, el formato lineal es más recomendable si pretendemos escribir esas entradas a mano, porque es muy simple, y en cambio el formato visual es muy aparatoso.

```
...
  BT t;
  t.setInputFormat(BT::INLINEFORMAT);
  cin >> t;
```

5. Recomendaciones sobre el uso de las funciones de lectura y escritura de `BinTree`

Como hemos mencionado al final de la sección anterior, si queremos preparar tests a mano para programas que leen árboles binarios de la entrada, es preferible poder leer esas entradas con INLINEFORMAT, que es más fácil de escribir.

Por otro lado, de cara a escribir árboles binarios, si nuestra intención és facilitar, a quien tenga que leer esos árboles, captar su estructura rápidamente, entonces es preferible escribirlos con VISUALFORMAT.

Sin embargo, hay casos en los que querremos escribir los árboles también con INLINEFORMAT. Uno de ellos es cuando queremos poder visualizar muchos árboles simultáneamente para compararlos. En tal caso, INLINEFORMAT es muy ventajoso porque es muy escueto, mientras que VISUALFORMAT ocupa mucho espacio de pantalla y permite ver pocos árboles al mismo tiempo.

Pero hay un motivo todavía más importante que nos puede hacer preferir INLINEFORMAT a VISUALFORMAT cuando se trata de leer y escribir árboles binarios, y este es la eficiencia. Las instrucciones de lectura y escritura son lineales con INLINEFORMAT, debido a que este formato muestra árboles con strings que tienen un tamaño proporcional al propio tamaño del árbol, y

están implementadas con un coste lineal. Ese no es el caso, ni mucho menos, de VISUALFORMAT.

Los juegos de pruebas privados grandes de `https://jutge.org` de ejercicios sobre `BinTree` trabajan siempre con INLINEFORMAT. En cambio, algunos de los juegos de pruebas públicos usan VISUALFORMAT para facilitar la comprensión de los mismos y del propio ejercicio.

El siguiente es un programa que transforma árboles binarios de INLINEFORMAT a VISUALFORMAT:

```
#include "BinTree.hh"

typedef BinTree<int> BT;

int main()
{
  BT t;
  t.setInputFormat(BT::INLINEFORMAT);
  cin >> t;
  t.setOutputFormat(BT::VISUALFORMAT);
  cout << t << endl;
}
```

Para finalizar, cabe remarcar que >> y << funcionaran con `BinTree` siempre y cuando el tipo de los elementos del árbol tenga sobrecargadas también las operaciones >> y <<. Por ejemplo, `int, float, double` no deberían dar problema. Tampoco `string` debería dar problemas, a menos que trabajemos con strings que puedan contener elementos sintácticos relevantes para la representación de árboles (símbolos `(` y `)` y `,` en el caso de INLINEFORMAT, o símbolos `|` y `-` en el caso de VISUALFORMAT). En esos casos, o bien usamos tipos alternativos que escapen esos caracteres especiales, o tendremos que construir funciones de lectura y escritura propias. Similarmente, en el caso de trabajar con tipos de datos que no tengan >> y << sobrecargados, tendremos que, o bien sobrecargar esos operadores, o bien construir nuestras propias funciones de lectura y escritura.