ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

Funciones sobre árboles binarios

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Recordatorio: interfaz de BinTree<T>

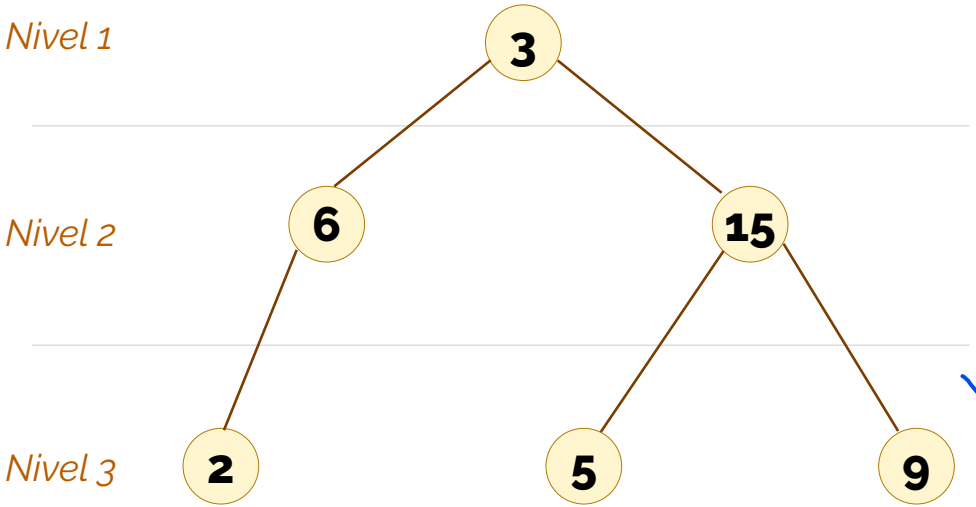
```
template<class T>
class BinTree {
public:
  BinTree();
  BinTree(const T &elem);
  BinTree(const BinTree &left, const T &elem, const BinTree &right);

  const T & root() const;
  BinTree left() const;
  BinTree right() const;
  bool empty() const;

private:
   ...
};
```

Vamos a ver dos funciones sencillas relacionadas con estos árboles binarios

Recordatorio: altura de un árbol binario



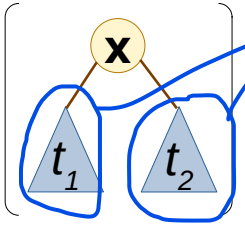
- La **altura** de un árbol es el máximo de los niveles de los nodos.

En este caso la altura de esto es 3.

Definición recursiva de altura

- Es posible definir recursivamente la altura de un árbol binario:

if(tree.empty) ***height***(—) = 0 En el momento que sea empty clavamos que devuelva un 0.

else ***height***(

) = 1 + max(***height***(t_1) , ***height***(t_2))

Llamada recursiva a la función que lo calcule. Primero calculamos la altura máxima de t1, y luego la de t2 haciendo otra llamada recursiva.

Función height

$$height(\text{—}) = 0$$

$$height\left(\begin{array}{c} x \\ t_1 \quad t_2 \end{array}\right) = 1 + \max(\, height(t_1), height(t_2)\,)$$

```
template<typename T>
int height(const BinTree<T> &tree) {
  if (tree.empty()) {
    return 0;
  } else {
    return 1 + std::max(height(tree.left()), height(tree.right()));
  }
}
```

Si el árbol es vacío devuelvo 0

Si no debo calcular la altura del árbol izquierdo, del derecho y quedarnos con la máxima altura entre ambos porque eso es lo que nos interesa

Coste en tiempo

```
template<typename T>
int height(const BinTree<T> &tree) {
  if (tree.empty()) {
    return 0;
  } else {
    return 1 + std::max(height(tree.left()), height(tree.right()));
  }
}
```

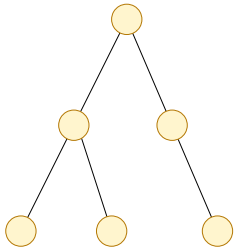
$$T(n) \begin{cases} c_0 & n = 0 \\ T(n_1) + T(n_2) + k^2 & n > 0 \end{cases}$$

Esto es lineal con respecto a n. El ha definido la recurrencia de esta manera. Pero yo la hubiera definido como dos llamadas de coste la mitad.

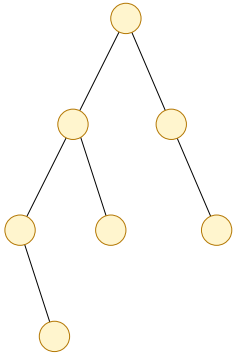
Árboles equilibrados en altura

Un árbol está **equilibrado en altura** si:

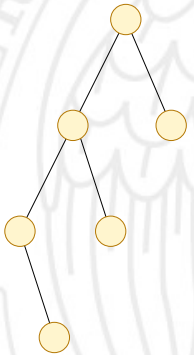
- Es el árbol vacío, o bien
- La diferencia entre las alturas de sus hijos es, como mucho, 1, y ambos están equilibrados en altura.



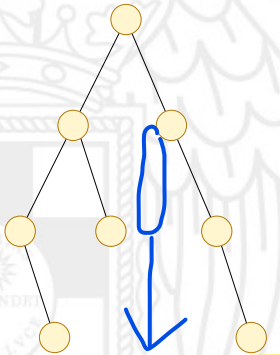
equilibrado ya que no hay diferencias de alturas



equilibrado, la diferencia de alturas es 1.



Este ya NO es equilibrado ya que la diferencia de altura entre hijo derecho e hijo izquierdo es >1



El hijo izquierdo tiene altura 0 y el otro tiene altura 2.

Definición recursiva

$$balanced(—) = true$$

Si el árbol es vacío entonces está equilibrado. If(tree.empty()) return true.

$$balanced\left(\begin{array}{c} x \\ t_1 \quad t_2 \end{array}\right) = balanced(t_1) \wedge balanced(t_2) \wedge \left| height(t_1) - height(t_2) \right| \leq 1$$

else
{
Esta equilibrado si el subárbol izquierdo está equilibrado , si el subárbol derecho está equilibrado y si la diferencia entre la altura del subárbol izquierdo menos la del derecho es menor o igual que 1

return bal_left && bal_right && abs(height_left-height_right)<=1

]

Función balanced

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  if (tree.empty()) {
    return true;
  } else {
    bool bal_left = balanced(tree.left());
    bool bal_right = balanced(tree.right());
    int height_left = height(tree.left());
    int height_right = height(tree.right());
    return bal_left && bal_right && abs(height_left - height_right) ⩽ 1;
  }
}
```

¿Cuál es el coste en tiempo?

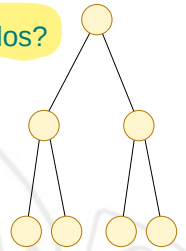
$$T(n) \begin{cases} k_1 & n=0 \\ T(n_i) + T(n_d) + k_2 n_i + k_3 h_i + k_4 \end{cases}$$

Función balanced: caso mejor

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  if (tree.empty()) {
    return true;
  } else {
    bool bal_left = balanced(tree.left());
    bool bal_right = balanced(tree.right());
    int height_left = height(tree.left());
    int height_right = height(tree.right());
    return bal_left && bal_right && abs(height_left - height_right) <= 1;
  }
}
```

izquierdo y derecho equilibrados?

alturas



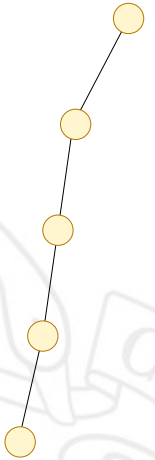
$$T(n) = \begin{cases} k_1 & n = 0 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + k_2 n + k_3 \frac{n}{2} + k_4 & \end{cases}$$

$2T\left(\frac{n}{2}\right) + k_n$

n log(n)

Función balanced: caso peor

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
  if (tree.empty()) {
    return true;
  } else {
    bool bal_left = balanced(tree.left());
    bool bal_right = balanced(tree.right());
    int height_left = height(tree.left());
    int height_right = height(tree.right());
    return bal_left && bal_right && abs(height_left - height_right) <= 1;
  }
}
```



$n = h - 1$

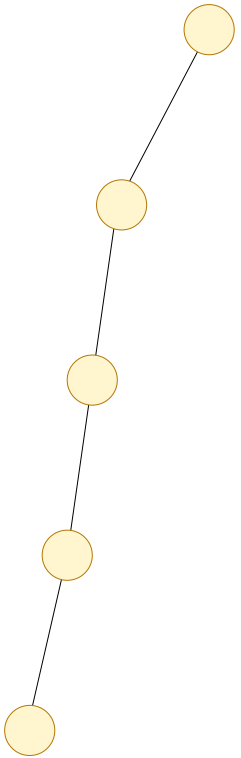
$n = 0$

$$T(n) = \begin{cases} K_1 & n = 0 \\ T(n-1) + T(0) + K_2(n-1) + K_3 \cdot 0 + K_4 & \end{cases}$$

$T(n-1) + Kn$

Coste n^2 en el caso peor.

Problema de llamar a `height`



¿Cómo solucionarlo?

- Implementando una función auxiliar recursiva que **simultáneamente** calcule la altura y determine si un árbol está equilibrado.
- Esta función devuelve dos valores: pair<,>
  - `balanced` (`bool`) – si el árbol está equilibrado o no.
  - `height` (`int`) – altura del árbol.

  Que lo haga a la vez.
- La función `balanced_height` devuelve un `pair<bool, int>`.

Función balanced_height

```
template <typename T>
std::pair<bool, int> balanced_height(const BinTree<T> &tree) {
  if (tree.empty()) {
    return {true, 0};
  } else {
    auto [bal_left, height_left] = balanced_height(tree.left());
    auto [bal_right, height_right] = balanced_height(tree.right());
    bool balanced =
      bal_left && bal_right && abs(height_left - height_right) <= 1;
    int height = 1 + std::max(height_left, height_right);
    return {balanced, height};
  }
}
```

El arbol vacío es balanceado y de altura 0

árbol no vacñio

llamar a la función a partir de los dos subárboles.

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n_i) + T(n_d) + k_2 & n > 0 \end{cases} \rightarrow T(n) = n$$

Función balanced

```
template <typename T>
bool balanced(const BinTree<T> &tree) {
  return balanced_height(tree).first;
}
```

Ya que la primera componente de la tupla es balanced.

Moraleja

- La mayoría de las funciones que operan sobre árboles son recursivas.
- En muchos casos estas funciones deben devolver valores auxiliares adicionales para evitar costes en tiempo elevados.

Como en la función balanced.