Tipos de datos arborescentes

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

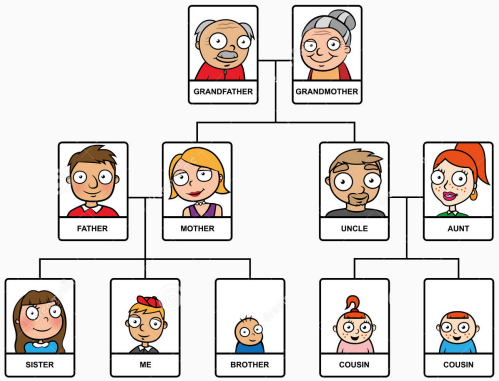
Bibliografía

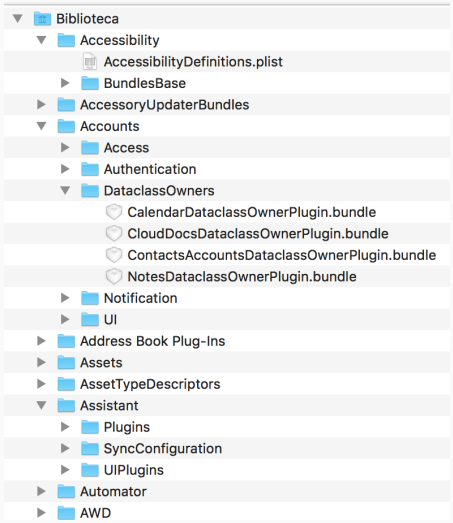
- R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.
  Capítulo 7
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
  Capítulos 6 y 7
- M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Fourth edition. Pearson, 2014.
  Capítulo 4

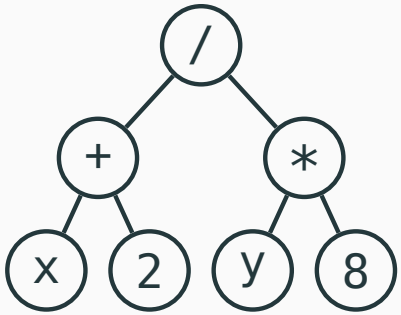
Árboles generales

Los TADs arborescentes se utilizan para representar datos organizados en jerarquías.

- Árboles genealógicos.
- Organización de un libro en capítulos, secciones, etc.
- Estructura de directorios y archivos de un sistema operativo.
- Árboles sintácticos, de análisis de expresiones.



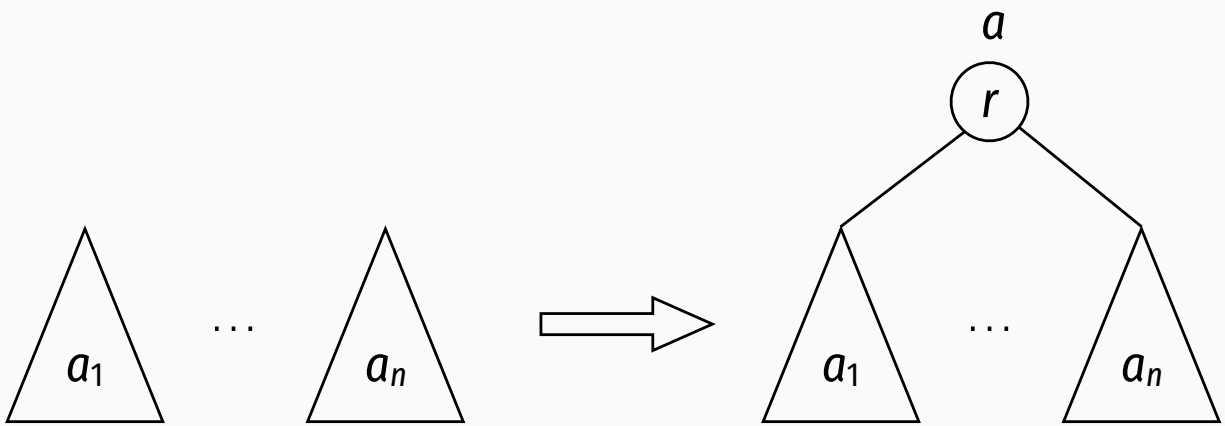




Construcción de árboles generales

Los árboles están formados por nodos (con información asociada). Se construyen de manera inductiva:

- Un solo nodo es un árbol a. El nodo es la raíz del árbol.
- Dados n árboles $a_1, \ldots, a_n$, podemos construir un nuevo árbol a añadiendo un nuevo nodo r como raíz y conectándolo con las raíces de los árboles a_i, que se llaman subárboles o hijos de a. A r se le llama padre de las raíces de estos subárboles.



Construcción de un árbol

Terminología

- Se llaman hojas a los nodos sin hijos. El resto de nodos son nodos internos.
- Dos nodos que comparten padre se dice que son hermanos.
- Un camino es una sucesión de nodos en la que cada nodo es padre del siguiente. Al número de nodos en un camino se le llama longitud.
- Una rama es cualquier camino que empieza en la raíz y acaba en una hoja.
- El nivel o profundidad de un nodo es la longitud del camino que va desde la raíz hasta él. En particular, el nivel de la raíz es 1.
- La talla o altura de un árbol es el máximo de los niveles de todos los nodos del árbol.
- El grado o aridad de un nodo es su número de hijos. La aridad de un árbol es el máximo de las aridades de todos sus nodos.
- Decimos que un nodo α es antepasado de β (resp. β es descendiente de α) si existe un camino desde α hasta β.

Tipos de árboles

Distinguimos distintos tipos de árboles en función de sus características:

- Ordenados o no ordenados. Un árbol es ordenado si el orden de los hijos de cada nodo es relevante. No debe confundirse este tipo de árbol ordenado, basado en la estructura del árbol, con los *árboles binarios de búsqueda*, en los que el orden está basado en el valor de los nodos del árbol.
- Árboles n-arios. Un árbol es n-ario si el máximo número de hijos de cualquier nodo es n. Los árboles generales son la unión de todos los n-arios.
- Árboles binarios. Un árbol binario es un árbol ordenado cuyos nodos tienen siempre dos hijos, el hijo izquierdo y el hijo derecho, aunque estos pueden ser vacíos.

Problemas

- 17 - Altura de un árbol general
- 18 - Los becarios precarios

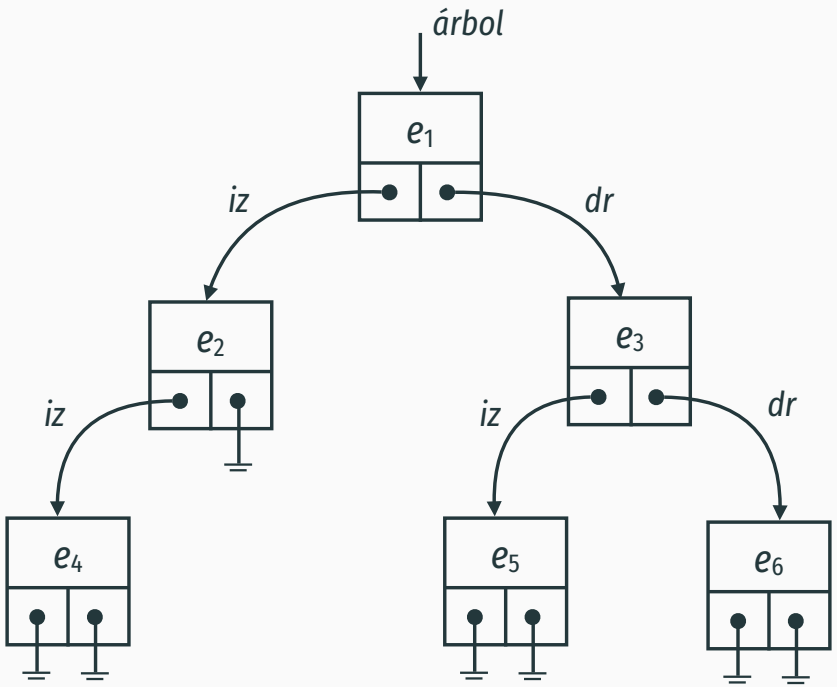
TAD de los árboles binarios

El TAD de los árboles binarios, `bintree<T>`, cuenta con las siguientes operaciones:

- crear el árbol vacío, `bintree`
- construir una hoja, `bintree(T const& e)`
- construir un árbol a partir de la raíz y sus dos hijos, `bintree(bintree<T> const& l, T const& e, bintree<T> const& r)`
- consultar la raíz, si existe, `T root() const`
- consultar el hijo izquierdo, si existe, `bintree<T> left() const`
- consultar el hijo derecho, si existe, `bintree<T> right() const`
- determinar si el árbol es vacío, `bool empty() const`

Implementación

Mediante nodos dinámicos enlazados y *compartición* de nodos entre árboles. Todas las operaciones tienen coste constante.



`bintree_eda.h`

Ejemplo de construcción de un árbol

```
bintree<char> arbsint = { { {'x'}, '+', {'2'} },
                          '/',
                          { {'y'}, '*', {'8'} }
                        };

cout << arbsint.left().root() << '\n';
cout << arbsint.right().right().root() << '\n';


bintree<char> arbsint2 = { arbsint, '-', '5' };

cout << arbsint2.left().right().right().root() << '\n';
```

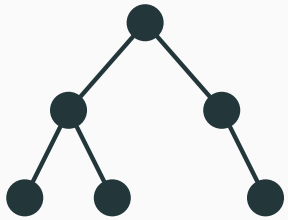
Lectura de un árbol de la entrada estándar

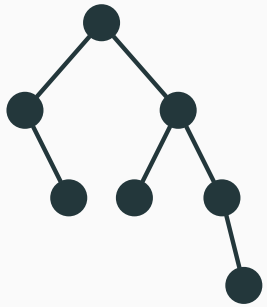
```
// lee un árbol binario de la entrada estándar, dado su preorden
template <class T>
bintree<T> leerArbol(T vacio) {
   T raiz;
   std::cin >> raiz;
   if (raiz == vacio) { // es un árbol vacío
      return {};
   } else { // leer recursivamente los hijos
      auto iz = leerArbol(vacio);
      auto dr = leerArbol(vacio);
      return { iz, raiz, dr };
   }
}
```

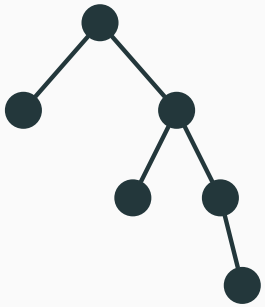
```
   / + x _ _ 2 _ _ * y _ _ 8 _ _

   auto arbsint = leerArbol('_');
```

Árboles binarios equilibrados

Un árbol binario está equilibrado si bien es vacío o bien cumple que la diferencia de alturas de sus dos hijos es como mucho 1 y además ambos están equilibrados.







Árboles binarios equilibrados: Posible implementación

```
template <class T>
int altura(bintree<T> const& arbol) {
   if (arbol.empty()) {
      return 0;
   } else {
      return max(altura(arbol.left()), altura(arbol.right())) + 1;
   }
}
```

Árboles binarios equilibrados: Posible implementación

```
template <class T>
int altura(bintree<T> const& arbol) {
   if (arbol.empty()) {
      return 0;
   } else {
      return max(altura(arbol.left()), altura(arbol.right())) + 1;
   }
}

template <class T>
bool equilibrado(bintree<T> const& arbol) { // O(N^2)
   if (arbol.empty()) {
      return true;
   } else {
      bool eq_iz = equilibrado(arbol.left());
      bool eq_dr = equilibrado(arbol.right());
      return eq_iz && eq_dr &&
             abs(altura(arbol.left()) - altura(arbol.right())) <= 1;
   }
}
```

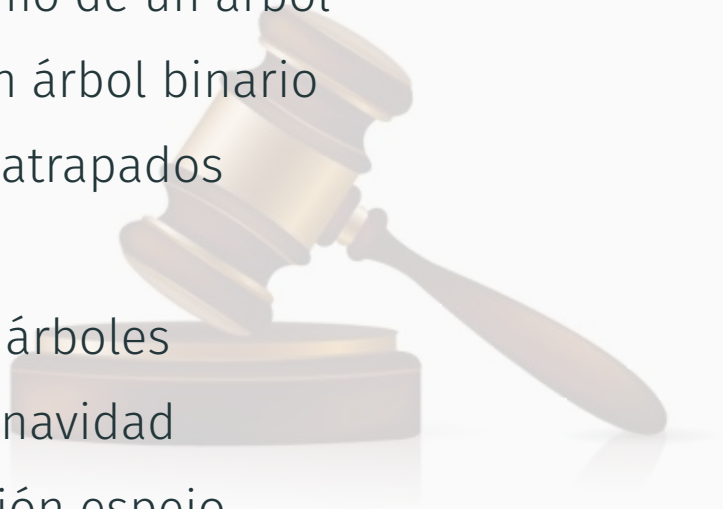
Árboles binarios equilibrados: Implementación eficiente

~~Otra posibilidad.~~

```
// dado un árbol binario averigua si está equilibrado y calcula su altura
template <class T>
pair<bool,int> equilibrado(bintree<T> const& arbol) {   // O(N)
   if (arbol.empty()) {
      return { true, 0 };
   } else {
      auto res_iz = equilibrado(arbol.left());
      auto res_dr = equilibrado(arbol.right());

      return { res_iz.first && res_dr.first &&
                  abs(res_iz.second - res_dr.second) < 2,
               std::max(res_iz.second, res_dr.second) + 1 };
   }
}
```

Problemas

- 19 - Número de nodos, hojas y altura de un árbol binario
- 20 - La frontera
- 21 - Elemento mínimo de un árbol
- 22 - Diámetro de un árbol binario
- 23 - Excursionistas atrapados



- ACR 203 - Suma de árboles
- ACR 204 - Árbol de navidad
- ACR 228 - Codificación espejo

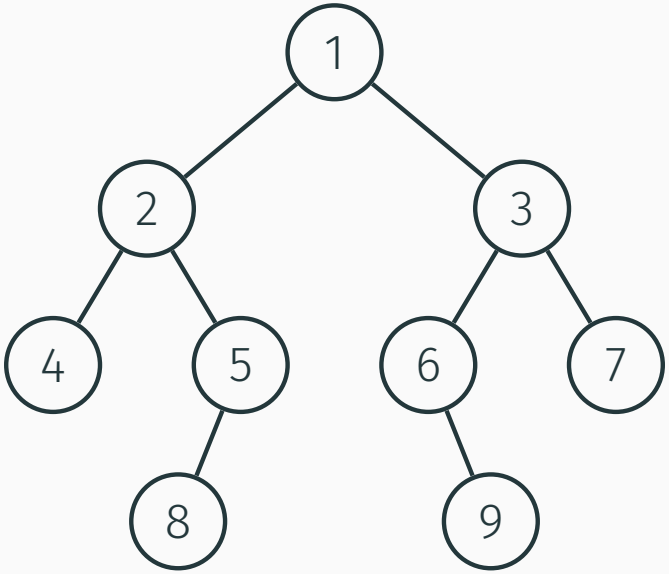
Recorridos de árboles

Recorrer un árbol consiste en visitar en cierto orden todos los nodos del árbol, haciendo algo con ellos.

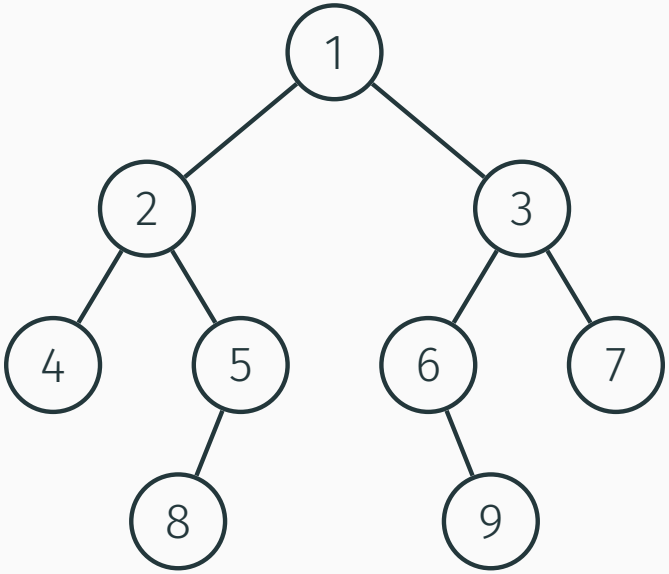
- Recorridos en profundidad
  - Preorden: se visita en primer lugar la raíz del árbol y, a continuación, se recorren en preorden el hijo izquierdo y el hijo derecho. *R I D*
  - Inorden: se recorre el hijo izquierdo, después se visita la raíz, y por último se recorre el hijo derecho. *I R D*
  - Postorden: primero se recorren los hijos izquierdo y derecho, en ese orden, y después se visita la raíz. *I D R*
- Recorrido por niveles o en anchura

`bintree_eda.h`

Recorridos de árboles

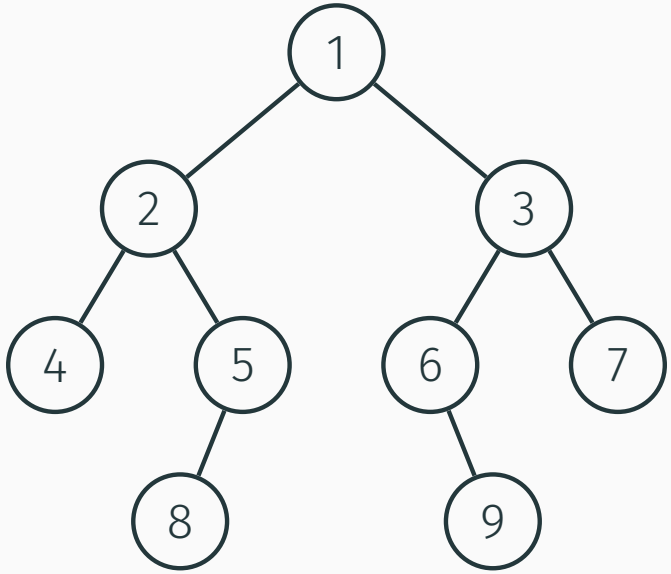


Recorridos de árboles



Preorden: 1 2 4 5 8 3 6 9 7

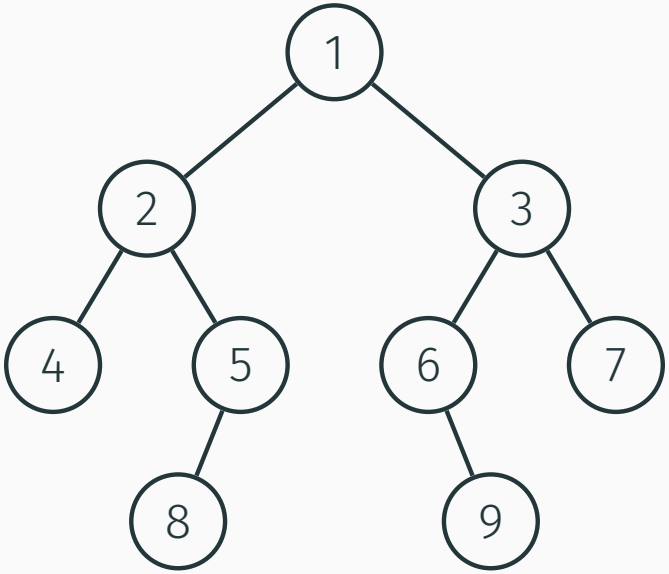
Recorridos de árboles



Preorden: 1 2 4 5 8 3 6 9 7

Inorden: 4 2 8 5 1 6 9 3 7

Recorridos de árboles

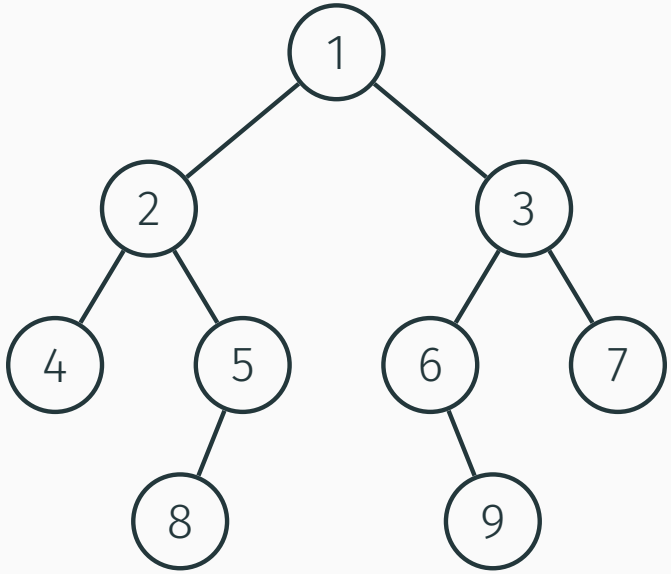


Preorden: 1 2 4 5 8 3 6 9 7

Inorden: 4 2 8 5 1 6 9 3 7

Postorden: 4 8 5 2 9 6 7 3 1

Recorridos de árboles



Preorden: 1 2 4 5 8 3 6 9 7

Inorden: 4 2 8 5 1 6 9 3 7

Postorden: 4 8 5 2 9 6 7 3 1

Por niveles: 1 2 3 4 5 6 7 8 9

Recorrido en inorden con iterador

Necesitamos funciones `begin()` y `end()` que devuelvan un iterador al primer elemento en inorden y a una posición *detrás* del último.

También una operación `++` para avanzar el iterador. Si el iterador está apuntando a un nodo con hijo derecho, el siguiente será el *primero* del hijo derecho.

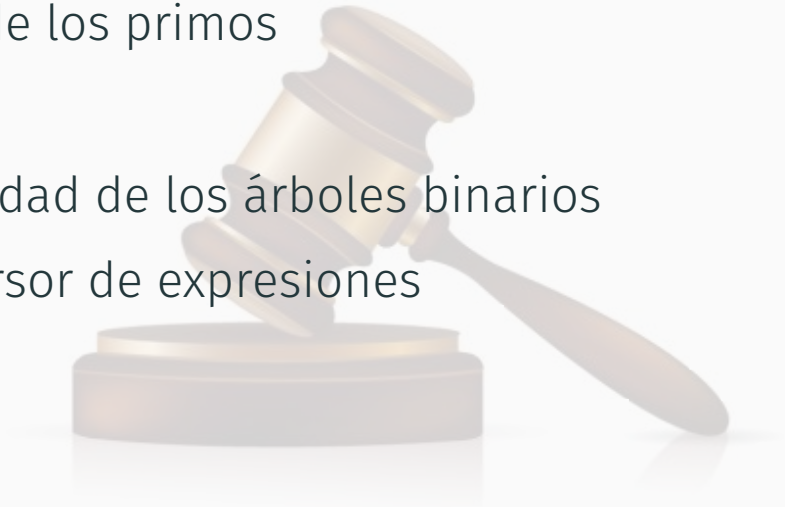
Si no existe ese hijo derecho, el siguiente, si existe, está por encima en el árbol. Hay que *retroceder* hasta el primer antecesor no visitado aún (aquel nodo en cuyo subárbol izquierdo nos encontramos).

Utilizamos una pila de árboles para poder retroceder. Se apila al bajar al hijo izquierdo.

`bintree_eda.h`

Problemas

- 24 - Reconstrucción de un árbol binario
- 25 - La barrera de los primos

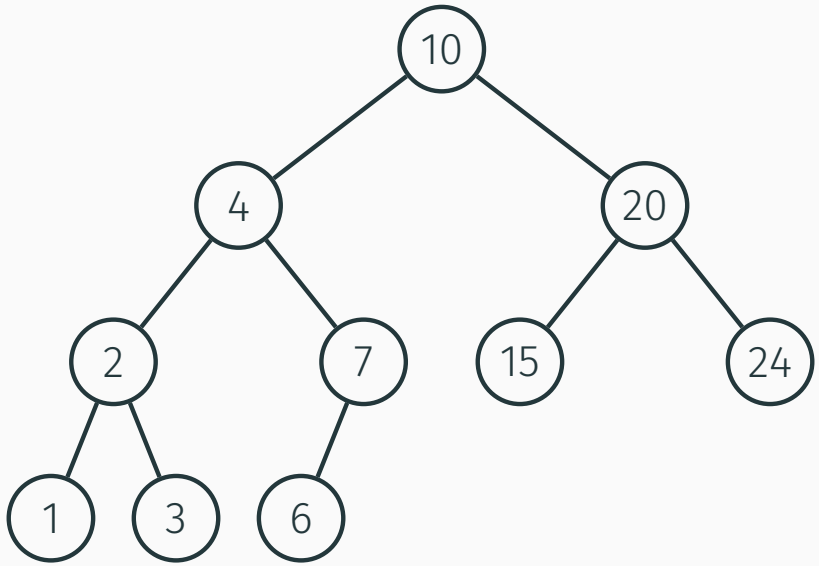


- ACR 218 - Ductilidad de los árboles binarios
- ACR 231 - Conversor de expresiones

Árboles binarios de búsqueda

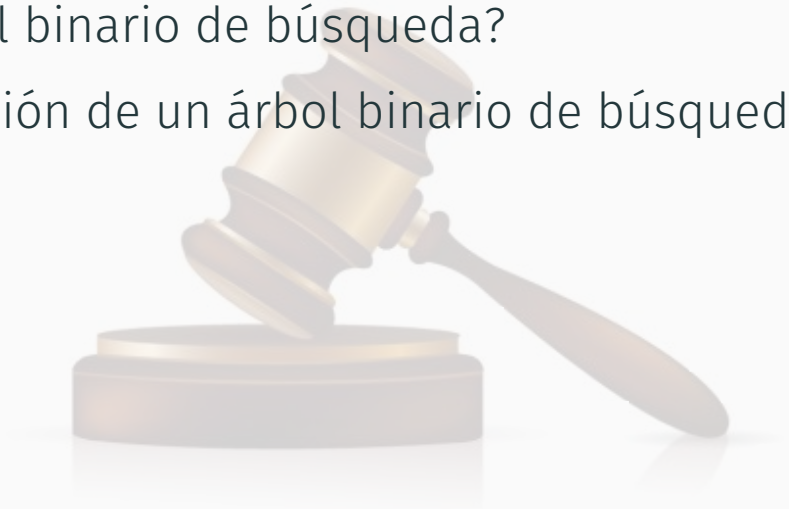
Los árboles binarios de búsqueda son árboles binarios cuyos nodos guardan elementos sobre los cuales hay definido un orden total estricto y que satisfacen la siguiente propiedad adicional: el elemento en cada nodo es mayor que todos sus descendientes izquierdos y menor que todos sus descendientes derechos.

Equivalentemente, o bien el árbol es vacío, o bien el elemento en la raíz es mayor que los elementos del hijo izquierdo y menor que los elementos del hijo derecho, y recursivamente los dos hijos son a su vez árboles binarios de búsqueda.



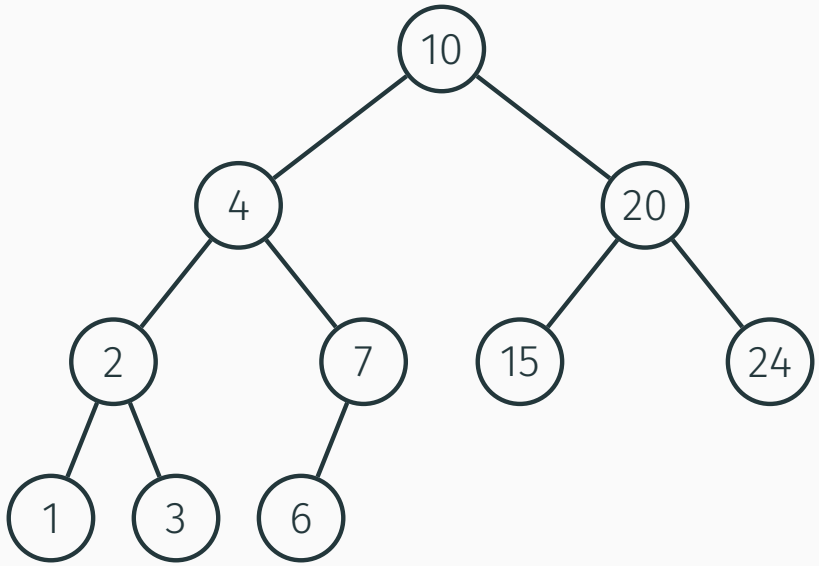
Problemas

- 26 - ¿Es un árbol binario de búsqueda?
- 27 - Reconstrucción de un árbol binario de búsqueda



Árboles binarios de búsqueda

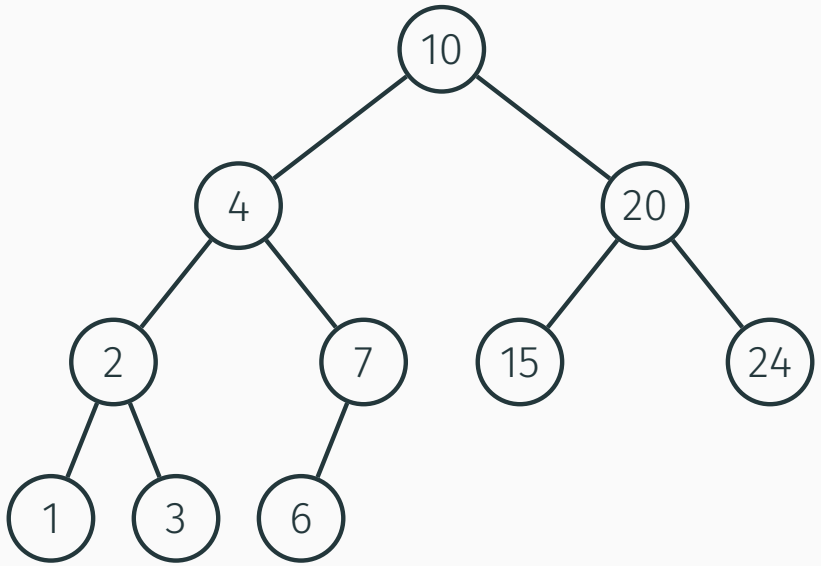
Operaciones: buscar, insertar o eliminar un elemento.



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.

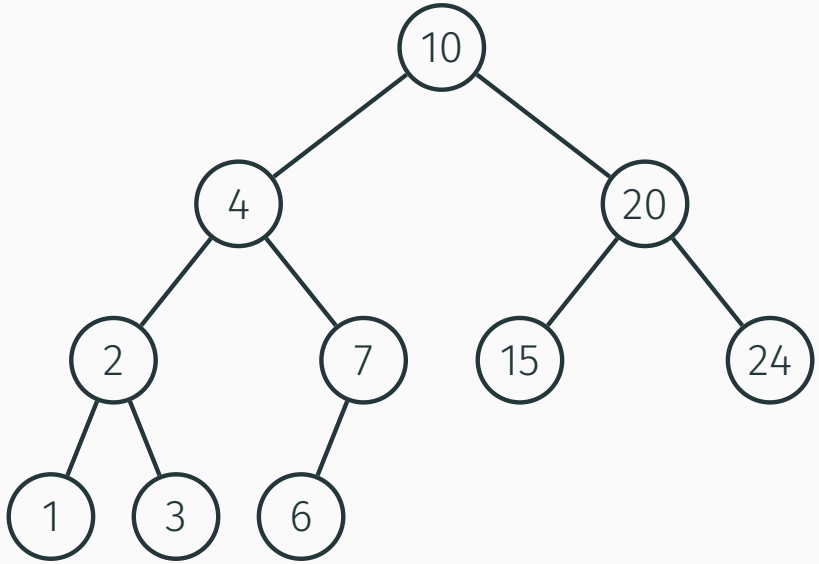
Buscar el 6



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.

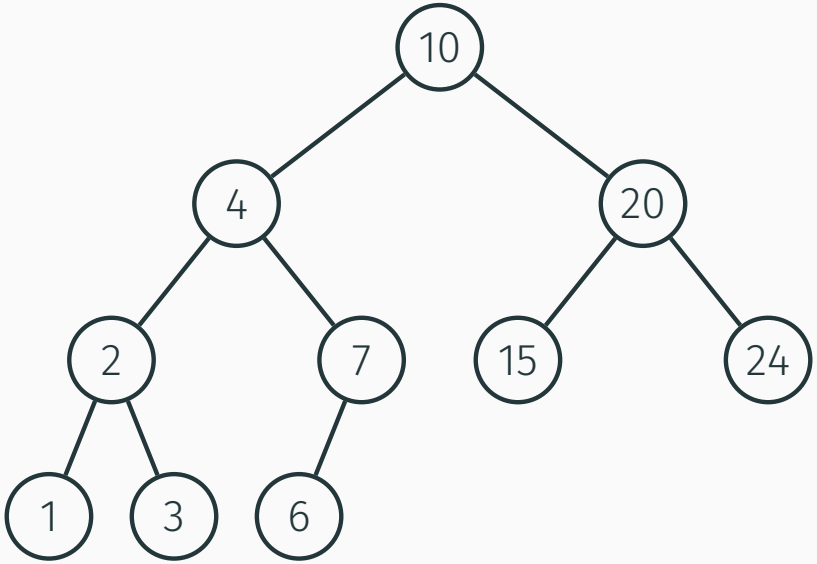
Buscar el 9



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.

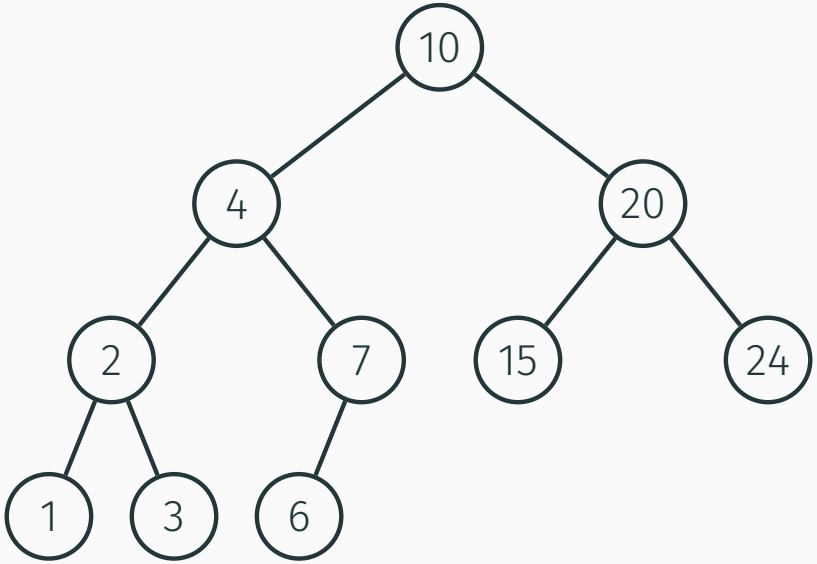
Insertar el 20



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.

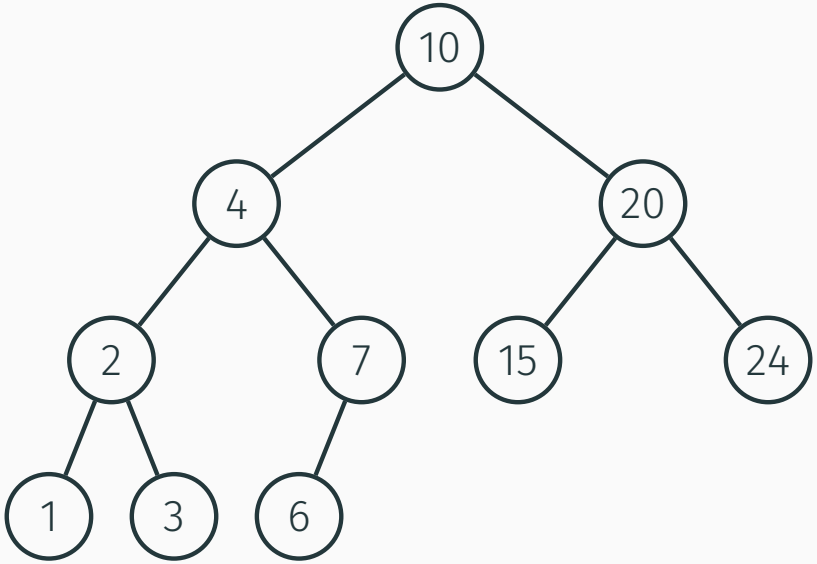
Insertar el 12



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.

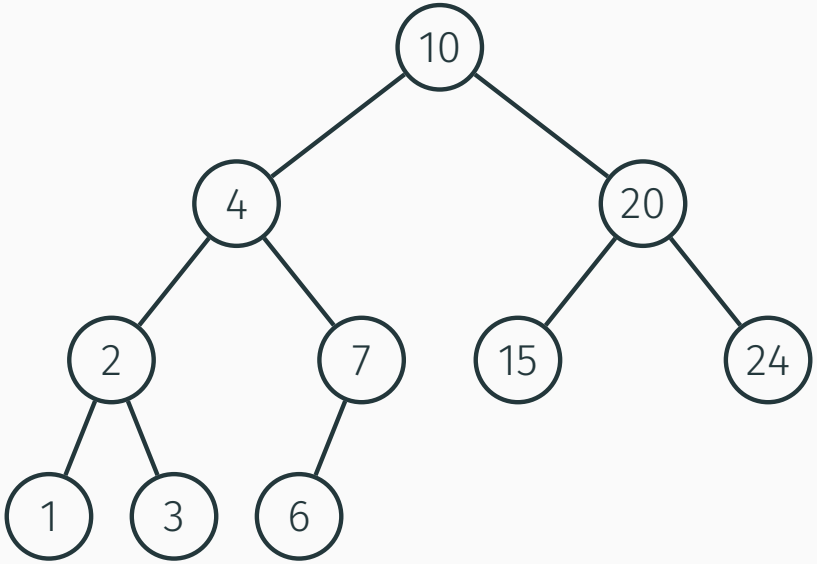
Borrar el 1



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.

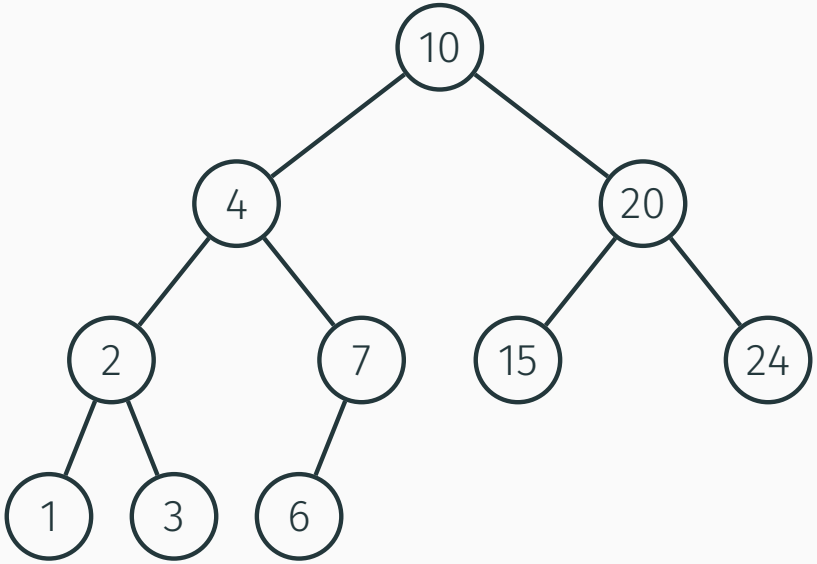
Borrar el 7



Árboles binarios de búsqueda

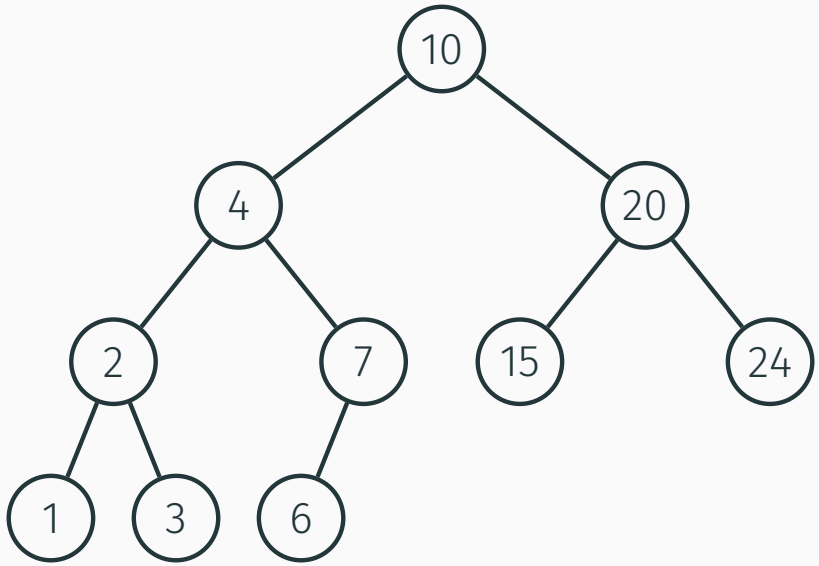
Operaciones: buscar, insertar o eliminar un elemento.

Borrar el 4



Árboles binarios de búsqueda

Operaciones: buscar, insertar o eliminar un elemento.



Todas las operaciones tienen un coste lineal respecto a la *altura* del árbol. En el caso peor, la altura es lineal respecto al número de nodos. En promedio, la altura es logarítmica respecto al número de nodos.

Implementación eficiente del TAD de los conjuntos

Los árboles binarios de búsqueda son una buena implementación del TAD de los conjuntos (de elementos ordenables) con las siguientes operaciones:

- conjunto vacío, `set`
- insertar un elemento, `bool insert(T const& elem)`
- eliminar un elemento, `bool erase(T const& elem)`
- averiguar si un elemento pertenece al conjunto, `int count(T const& elem) const`
- averiguar si el conjunto es vacío, `bool empty() const`
- averiguar el cardinal del conjunto, `int size() const`
- iterador que permita recorrer los elementos del conjunto en orden

`set_eda.h`

Ejemplos

```
set<std::string> cjto;

cjto.insert("hola");
cjto.insert("adios");
cjto.insert("caracola");
cjto.insert("pera");
cjto.insert("manzana");
cjto.insert("ciruela");
cjto.insert("zanahoria");

for (auto const& s : cjto)
    std::cout << s << ' ';
std::cout << '\n';
```

```
adios caracola ciruela hola manzana pera zanahoria
```

Ejemplos

```
set<std::string,std::greater<std::string>> cjtoReves;

cjtoReves.insert("hola");
cjtoReves.insert("adios");
cjtoReves.insert("caracola");
cjtoReves.insert("pera");
cjtoReves.insert("manzana");
cjtoReves.insert("ciruela");
cjtoReves.insert("zanahoria");

for (auto const& s : cjtoReves)
    std::cout << s << ' ';
std::cout << '\n';
```

```
zanahoria pera manzana hola ciruela caracola adios
```