

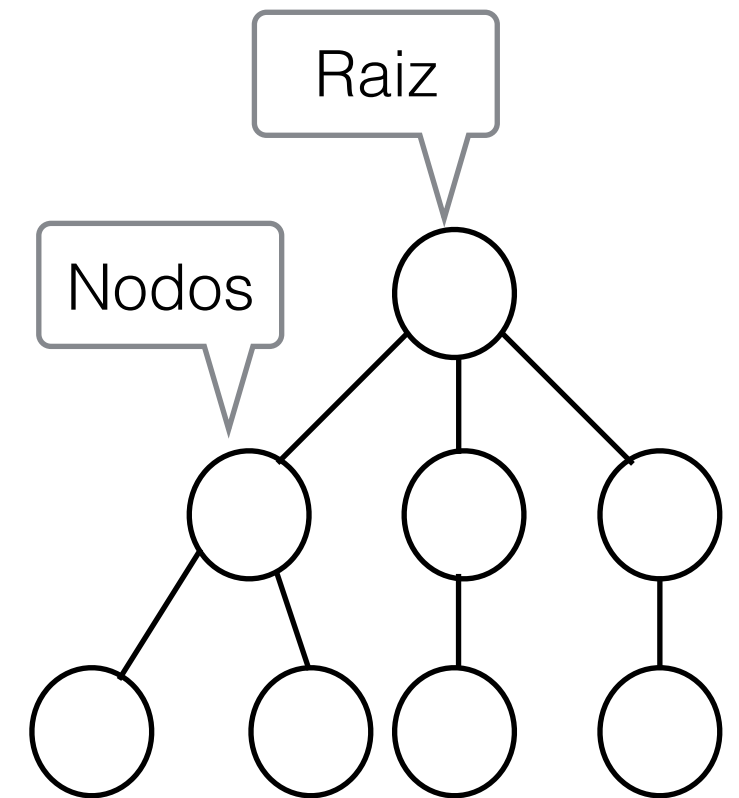
Árboles

Pablo Castro
Algoritmos I - UNRC

Árboles

Los árboles tienen las siguientes características:

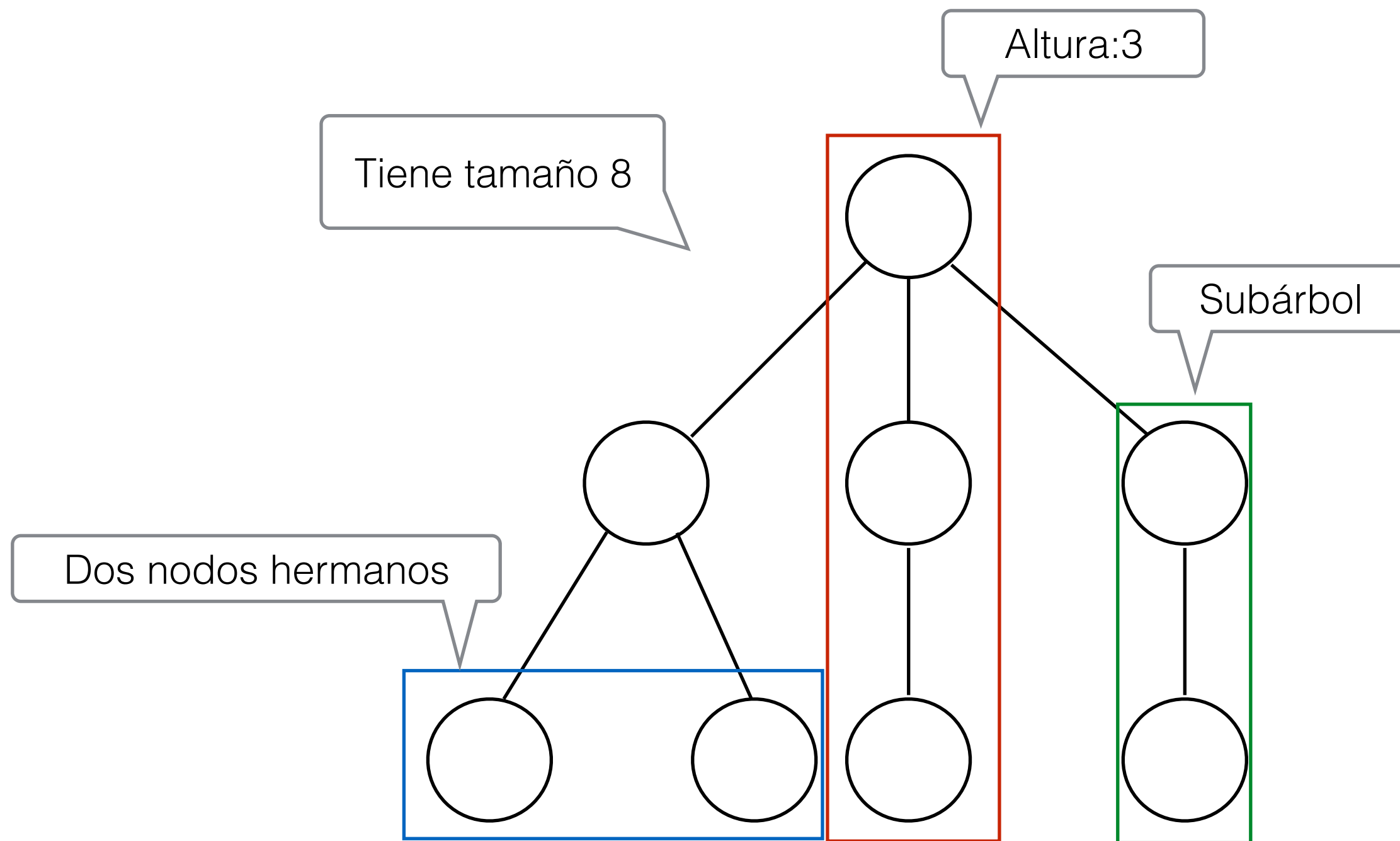
- No poseen una organización lineal,
- Tenemos una raíz,
- Cada nodo tiene un padre, excepto la raíz,
- Un nodo puede tener 0 ó muchos hijos



Definiciones

- **Hermanos:** Nodos con el mismo padre,
- **Hojas:** Nodos sin hijos,
- **Altura:** Longitud del camino más largo desde la raíz a una hoja,
- **Tamaño:** Cantidad de nodos,
- **Subárbol:** Es un nodo del árbol junto con todos sus descendientes

Ejemplo



Árboles Binarios

Los árboles binarios son aquellos que:

- Cada nodo tiene a lo sumo dos hijos,
- Cada hijo de un nodo es llamado hijo izq. o hijo derecho.

En Haskell se definen recursivamente:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

TAD Árbol Binario

El TAD árbol binario consta de las siguientes operaciones:

- raiz: Retorna la raíz del árbol,
- hi: retorna el subárbol izquierdo,
- hd: retorna el subárbol derecho,
- preorder: recorrido preorder,
- inorder: recorrido inorder,
- posorder: recorrido posorder.

Implementación JAVA

Primero definimos una interface con operaciones básicas:

```
// Ejemplo de una interfaz basica para arboles contiene la funcionalidad minima para este tipo
// de estructuras, puede ser enriquecida con mas operaciones
public interface BinaryTreeBasis{

    // Devuelve el elemento de la raiz
    public Object getRoot();

    // Setea la raiz
    public void setRoot(Object item);

    // Dice si el arbol es vacio
    public boolean isEmpty();

    //Remueve todo los nodos del arbol
    public void makeEmpty();

    // recorrido preOrder
    public void printPreOrder();

    // recorrido postOrder
    public void printPostOrder();

    // recorrido inOrder
    public void printInOrder();
}
```

Implementación con Memoria Dinámica

Una posible implementación con memoria dinámica:

```
public class TreeNode{
    private Object element; // elemento del nodo
    private TreeNode left;  // hijo izquierdo
    private TreeNode right; // hijo derecho

    // constructor del NodoArbol por defecto
    public TreeNode(){
        element = null;
        left = null;
        right = null;
    }
    // implementar el resto..
}
```

Parecida Node de
LinkedList

Los recorridos preorder, inorder,
posorder deben ser implementados
acá

```
public class LinkedBinaryTree implements BinaryTreeBasis{
    // raíz del arbol
    private TreeNode root;

    public LinkedBinaryTree(){
        root = null;
    }
    // Implementar el resto...
}
```

Un LinkedBinaryTree
tiene como raíz un
TreeNode

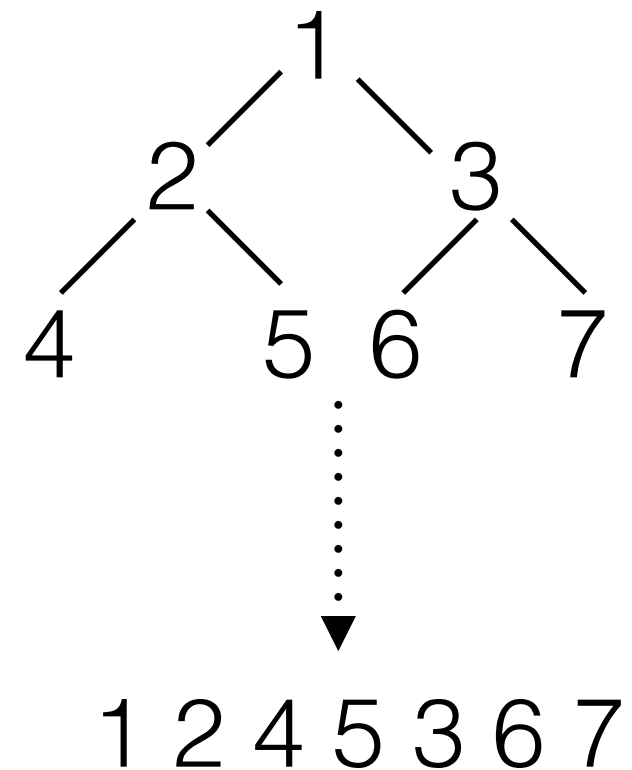
Preorder

Con preorder se recorre primero la raíz, después el hi y finalmente el hd.

```
// Recorrido preorder
public void printPreOrder(){
    // se imprime la raíz
    System.out.println(element);
    // se recorre el hi
    if (left != null){
        left.printPreOrder();
    }

    // se recorre el hd
    if (right != null){
        right.printPreOrder();
    }
}
```

Implementado en
TreeNode, se
puede llamar desde
LinkedBinaryTree



Este código solo imprime se
puede modificar para hacer
otras tareas

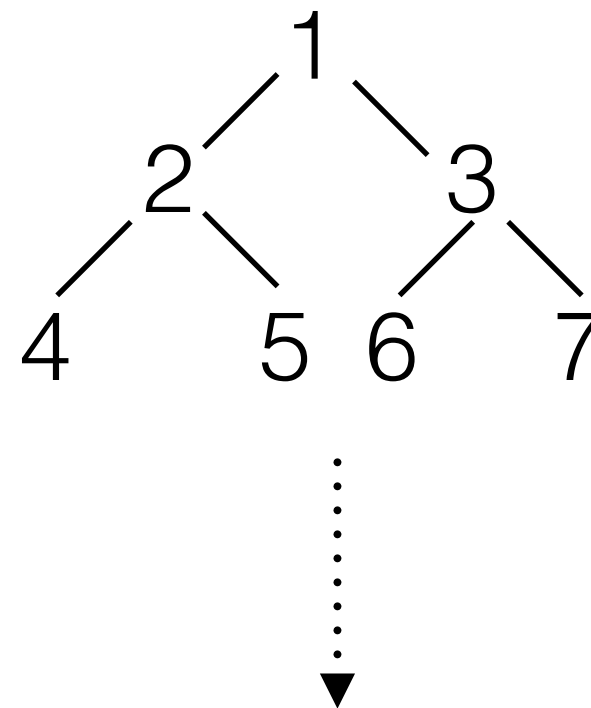
El Tiempo de ejecución es
 $O(n)$

Inorder

Primero el hi, después la raíz y finalmente el hd.

```
// Recorrido inorder
public void printInOrder(){
    // se recorre el hi
    if (left != null){
        left.printInOrder();
    }

    // se imprime la raíz
    System.out.println(element);
    // se imprime el hd
    if (right != null){
        right.printInOrder();
    }
}
```



4251637

El Tiempo de ejecución es
 $O(n)$

Posorder

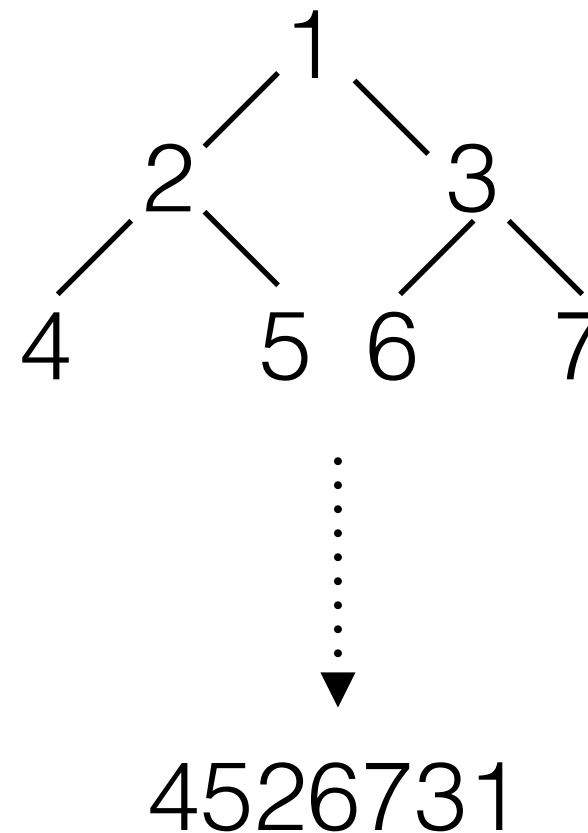
La raíz se recorre al último:

```
// Recorrido posorder
public void printPostOrder(){
    // se recorre hi
    if (left != null){
        left.printPostOrder();
    }

    // se recorre hd
    if (right != null){
        right.printPostOrder();
    }

    // se imprime la raíz
    System.out.println(element);
}
```

El Tiempo de ejecución es $O(n)$ (n cantidad de nodos)



Propiedades...

Sea t un árbol binario, en donde $size(t)$ es su tamaño, $alt(t)$ su altura. y $full(t)$ dice si el árbol está lleno, entonces:

Propiedad 1: $size(t) \leq 2^{alt(t)} - 1$

Propiedad 2: $\log_2(size(t) + 1) \leq alt(t)$

Propiedad 3: $full(t) \Rightarrow size(t) = 2^{alt(t)} - 1$

Propiedad 4: $full(t) \Rightarrow alt(t) = \log_2(size(t) + 1)$

un árbol se dice full (o lleno) cuando la altura de sus hijos es igual, y ambos hijos son full (o llenos)

Aplicaciones

Los árboles tienen diversas aplicaciones, por ejemplo:

- Representación de expresiones (aritméticas, booleanas, etc),
- Implementación eficiente de colecciones de datos (bases de datos),
- Organización de datos (sistema de archivos),
- Varios algoritmos importantes usan árboles para obtener una implementación elegante y eficiente.

Codificación de Huffman

La idea es codificar datos tal que la cantidad de bits usados es la menor posible.

- En código Ascii tenemos 256 caracteres,
- Cada símbolo se codifica con 8 bits ($2^8 = 256$),
- Si el mensaje tiene n caracteres necesitamos, $n \cdot 8$ bits,
- Desperdiciamos espacio debido a que todos los símbolos se codifican con la misma cantidad de bits.

Codificación de Huffman

Idea: utilizar la frecuencia de los caracteres para asignándole una codificación más corta a aquellos símbolos que más veces aparecen.

- Si un carácter aparece más veces, tiene una frecuencia más alta, por lo tanto se le debe asignar una codificación más corta
- Por ejemplo, en castellano es más frecuente una “a” que una “x”,
- Esto permitir acorta la cantidad de bits de la información a codificar.

Huffman (cont.)

Supongamos que en el mensaje aparecen:

a,b,c,d,e

Necesitamos 3 bits, es decir, aproximadamente $\log 5$ bits.

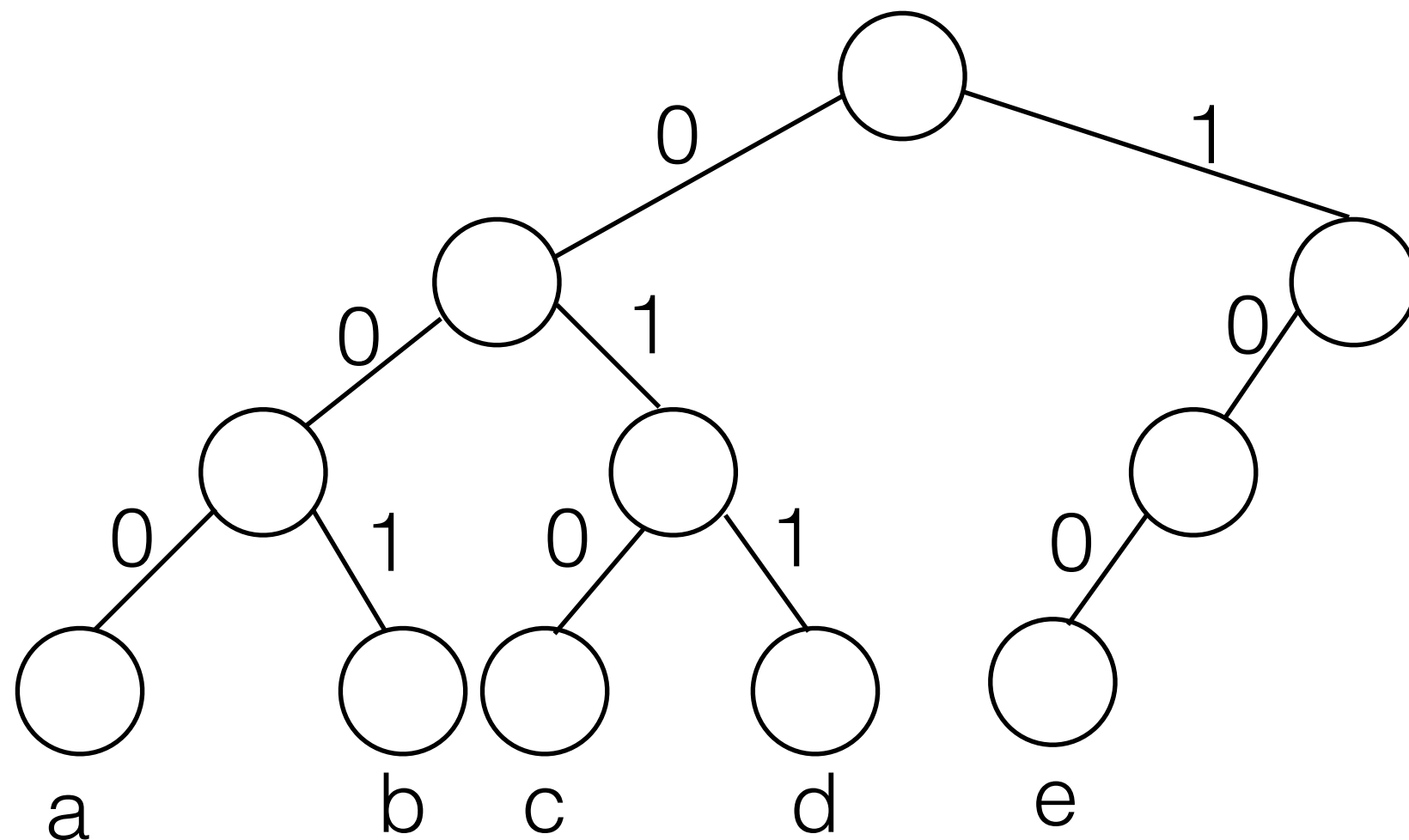
a	000
b	001
c	010
d	011
e	100

Una posible
codificación

Si la 'a' aparece más veces que la
'b' estamos desperdiciando
espacio

Codificación de Huffman

Podemos representar cualquier codificación utilizando árboles:



Codificación de Huffman

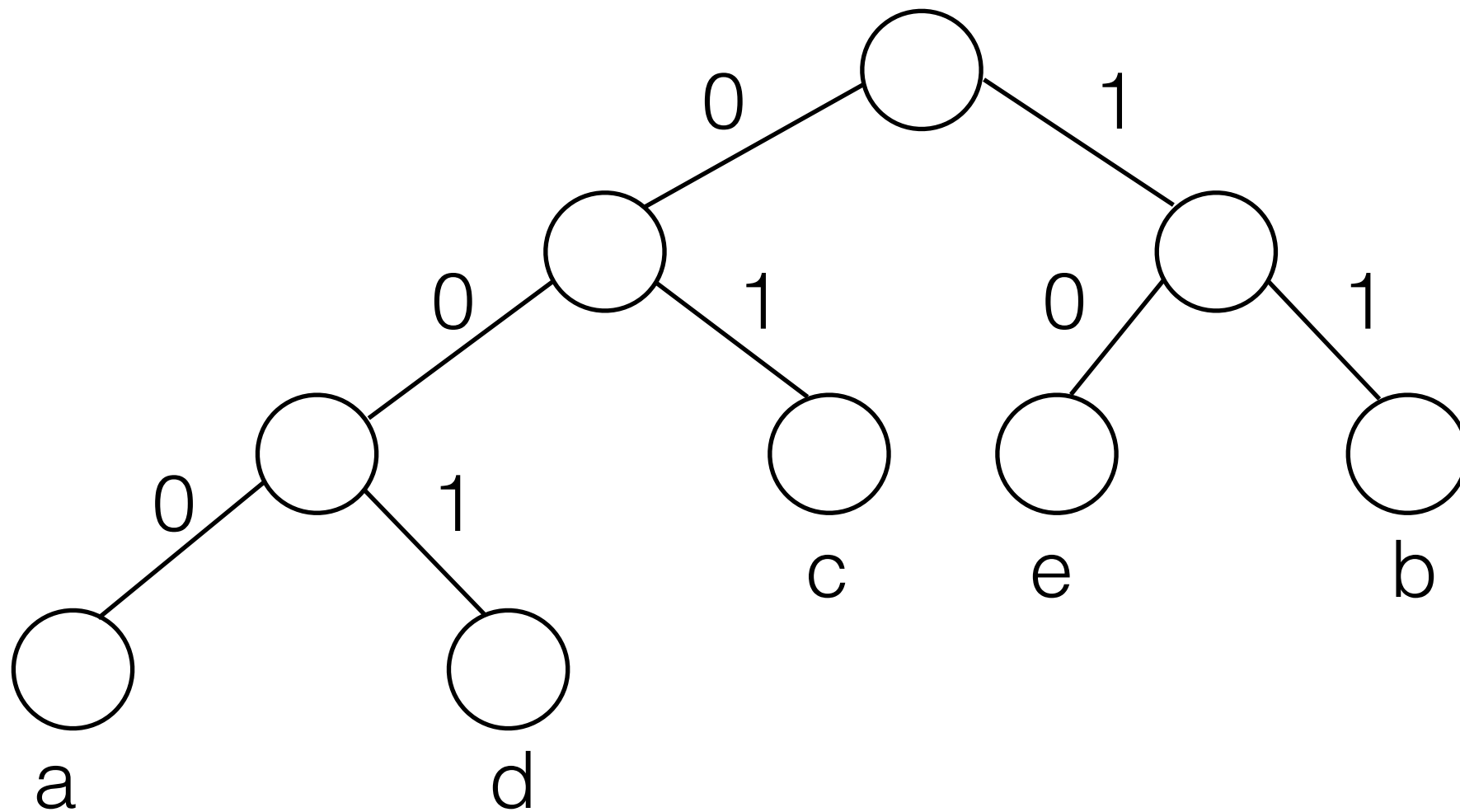
La codificación de Huffman aprovecha la frecuencia de la aparición de letras:

	Frecuencia
a	0.12
b	0.30
c	0.15
d	0.08
e	0.25

Supongamos que las letras tienen la siguiente probabilidad (frecuencia) de aparición

Codificación Huffman

Utilizando Huffman construimos el siguiente árbol:



Codificación Huffman

La codificación de Huffman posee la propiedad de que el código de ningún carácter es prefijo de otro.

Mensaje: “abbbeebbadcbbebebebe”

Codificación sin Huffman:

000001001001100100001001000011010001001100001000010100001100001100

Codificación Huffman:

00011111101011110000010111111011000011011101110

\

- Creamos un árbol por cada símbolo,
- Cada árbol está etiquetado con la frecuencia del símbolo que contiene,
- En cada paso se seleccionan los dos árboles con menor peso, y se unen, su peso es la suma de los pesos,
- Se repite este paso hasta que quede un solo árbol
- La codificación lograda no es ambigua y es óptima!