



Programación 2

Árboles

Fernando Orejas

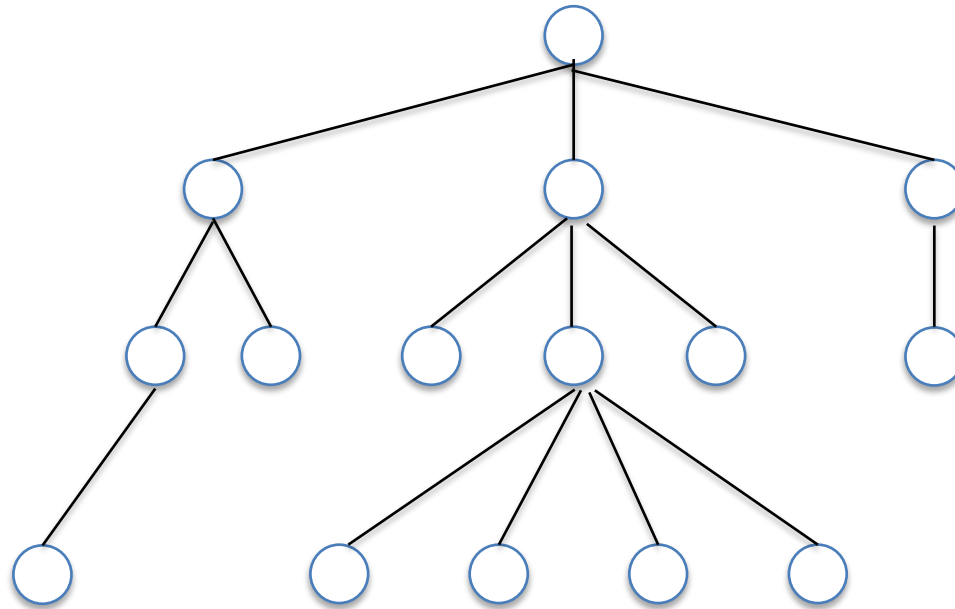
Transparencias basadas en las de Ricard Gavaldà y Conrado Martínez

1. Estructuras arborescentes
2. Árboles binarios
3. Recorridos

Estructuras arborescentes

Terminología

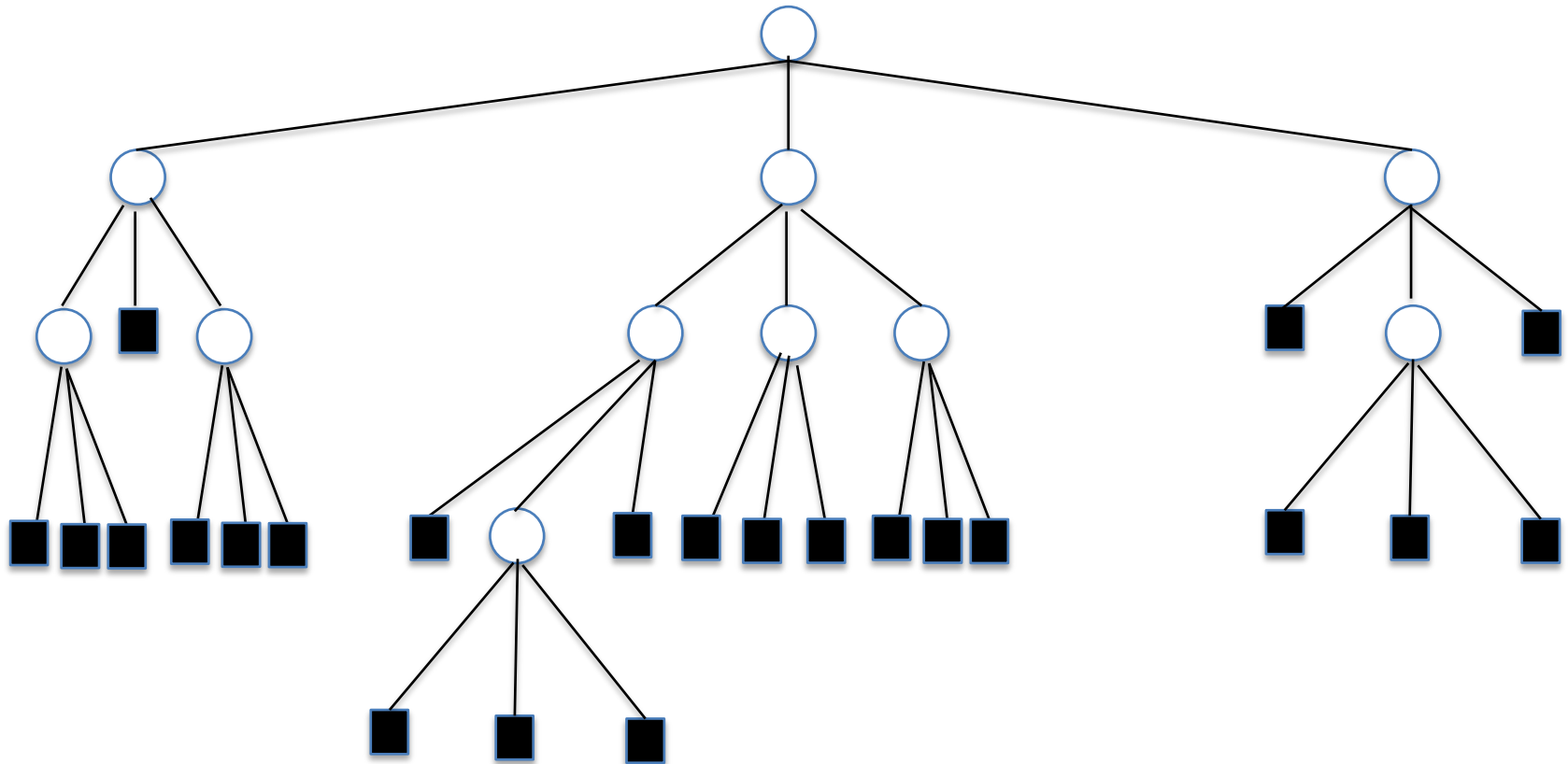
- nodo
- padre, hijo
- ascendiente
descendiente
- hermano
- raiz, hoja
- camino
- nivel, altura



Definiciones

- Grafo dirigido que, o es vacío, o contiene un nodo (la raíz) del que hay un único camino a cada nodo del grafo
- Grafo no dirigido y conexo, con un arco menos que nodos y con un nodo distinguido (la raíz)
- Un árbol es, o bien un árbol vacío, o bien es un nodo con cero o más árboles sucesores

Árboles n-arios

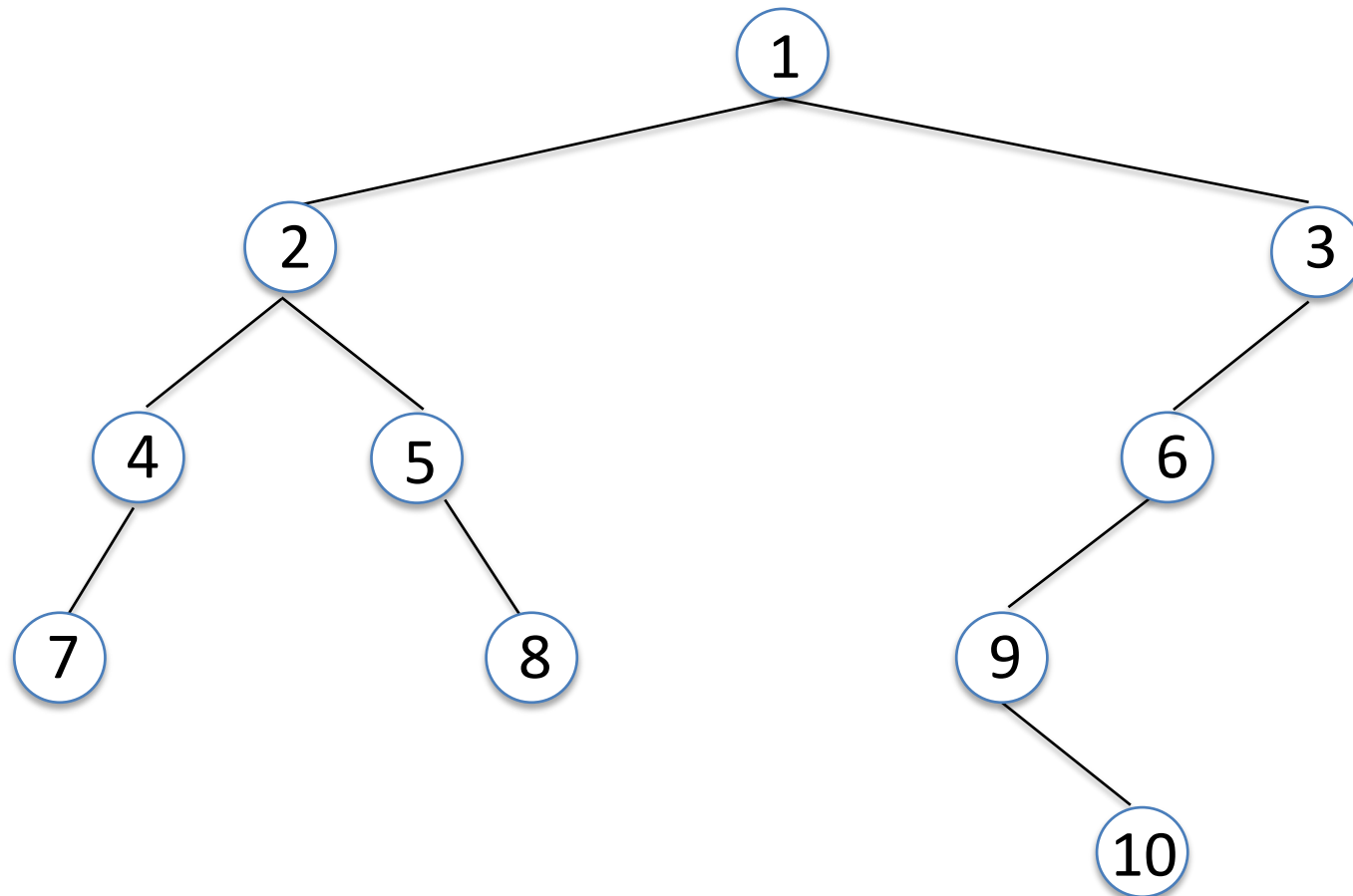


Árboles binarios

Árboles Binarios

- Árboles n-arios con $n=2$
- Los dos hijos de un nodo son el izquierdo y el derecho

Árboles binarios



Especificación de la clase BinTree

```
template <class T> class BinTree {  
    public:  
        // Constructoras  
  
        // Pre: true  
        // Post: crea un árbol vacía  
        BinTree ();  
  
        // Pre: true  
        // Post: crea un árbol con x como raiz, left como hijo  
        // izquierdo y right como hijo derecho  
        BinTree (const T& x, const BinTree& left, const BinTree&  
        right);  
  
        // Destructora  
        ~BinTree();
```

// Consultoras

// Pre: true

// Post: Retorna true si el árbol y false en caso

// contrario

bool empty ();

// Pre: El parámetro implícito no está vacío

// Post: retorna el hijo izqdo del parámetro implícito

BinTree left ();

// Pre: El parámetro implícito no está vacío

// Post: retorna el hijo dcho del parámetro implícito

BinTree right ();

// Pre: El parámetro implícito no está vacío

// Post: retorna la raiz del parámetro implícito

BinTree value ();

Tamaño de un árbol

```
/* Pre: true */  
/* Post: retorna el número de nodos del árbol t*/  
int size(const BinTree <int>& t){  
    if (t.empty()) return 0;  
    else return 1 + size(t.left()) + size(t.right());  
}
```

Búsqueda en un árbol

```
/* Pre: true */  
/* Post: nos dice si x está en t*/  
bool busq(const BinTree <int>& t, int x){  
    if (t.empty()) return false;  
    else  
        return (t.value() == x) or busq(t.left(),x) or  
            busq(t.right(),x);  
}
```

Suma k a todos los valores de un árbol

```
/* Pre: true */  
/* Post: retorna un arbol t' con la misma forma que t,  
tal que el valor de cada nodo de t' es igual a k + el  
valor del nodo correspondiente de t */  
BinTree sumak(const BinTree <int>& t, int k){  
    if (t.empty()) return BinTree();  
    else  
        return BinTree(t.value()+k,  
                        sumak(t.left(),k),  
                        sumak(t.right(),k));  
}
```

Suma k a todos los valores de un árbol

```

/* Pre: true */
/* Post: nos dice si x está en t*/
void sumak(BinTree <int> & t, int k){
    if (not t.empty()) {
        t = BinTree(t.value()+k,
                    sumak(t.left(),k),
                    sumak(t.right(),k));
    }
}

```

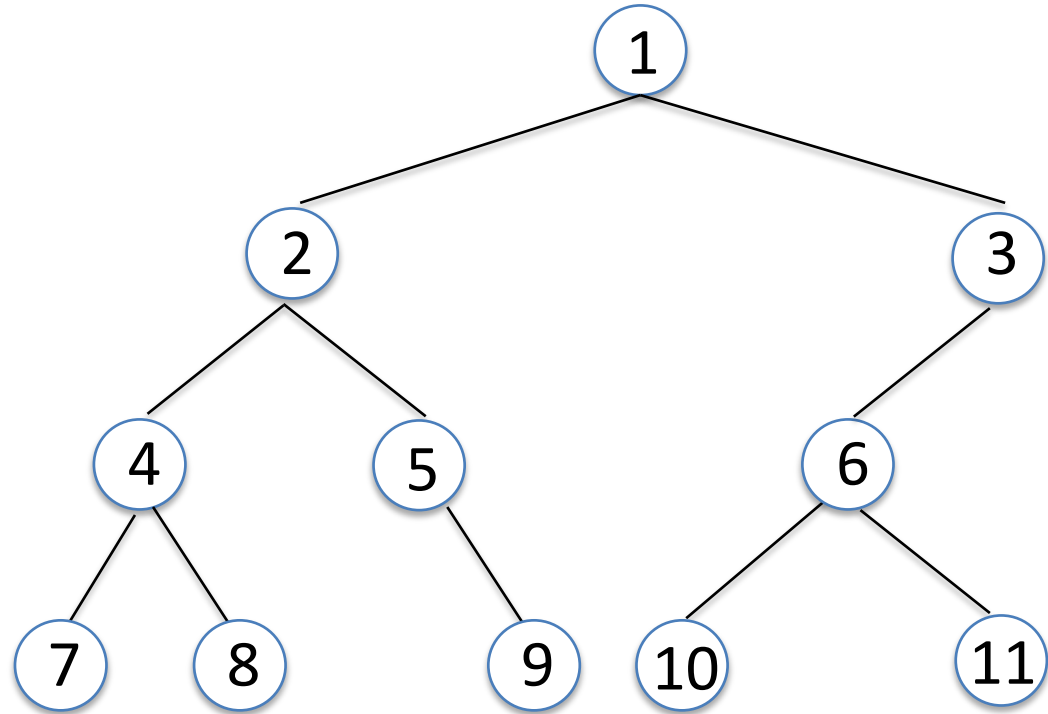
Recorridos

Recorridos de árboles

- En profundidad
 - Preorden
 - Postorden
 - inorden
- En amplitud (o por niveles)

Recorrido en preorden

1. Visitamos la raiz
2. Recorremos en preorden el hijo izquierdo
3. Recorremos en preorden el hijo derecho

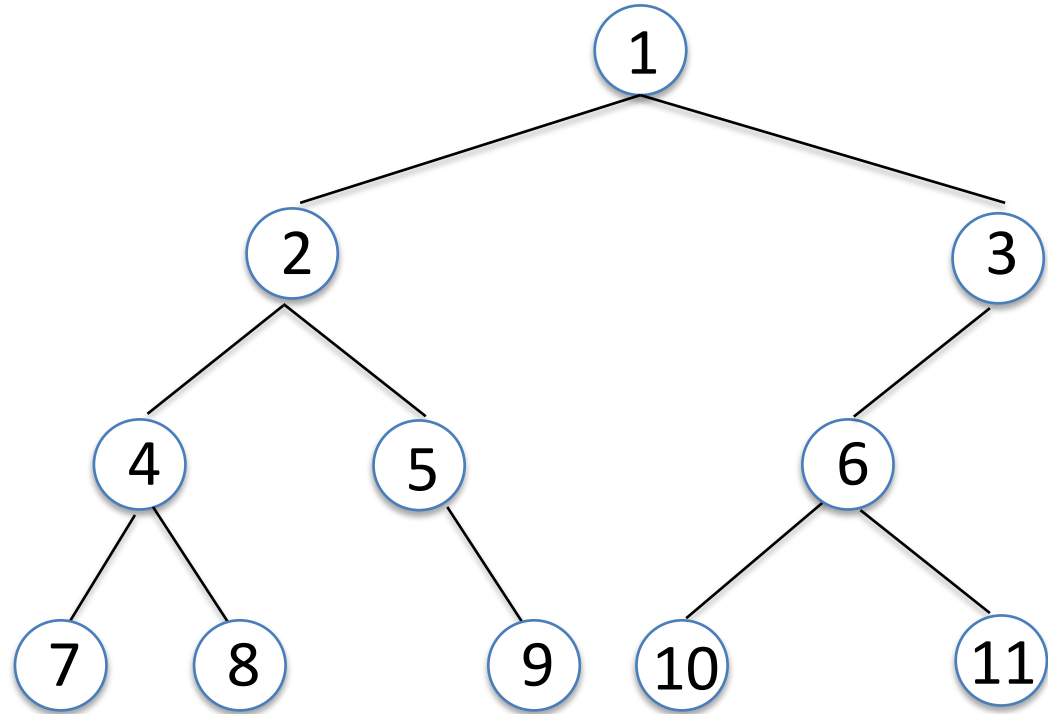


- Recorrido

1 2 4 7 8 5 9 3 6 10 11

Recorrido en postorden

1. Recorremos en postorden el hijo izquierdo
2. Recorremos en postorden el hijo derecho
3. Visitamos la raíz

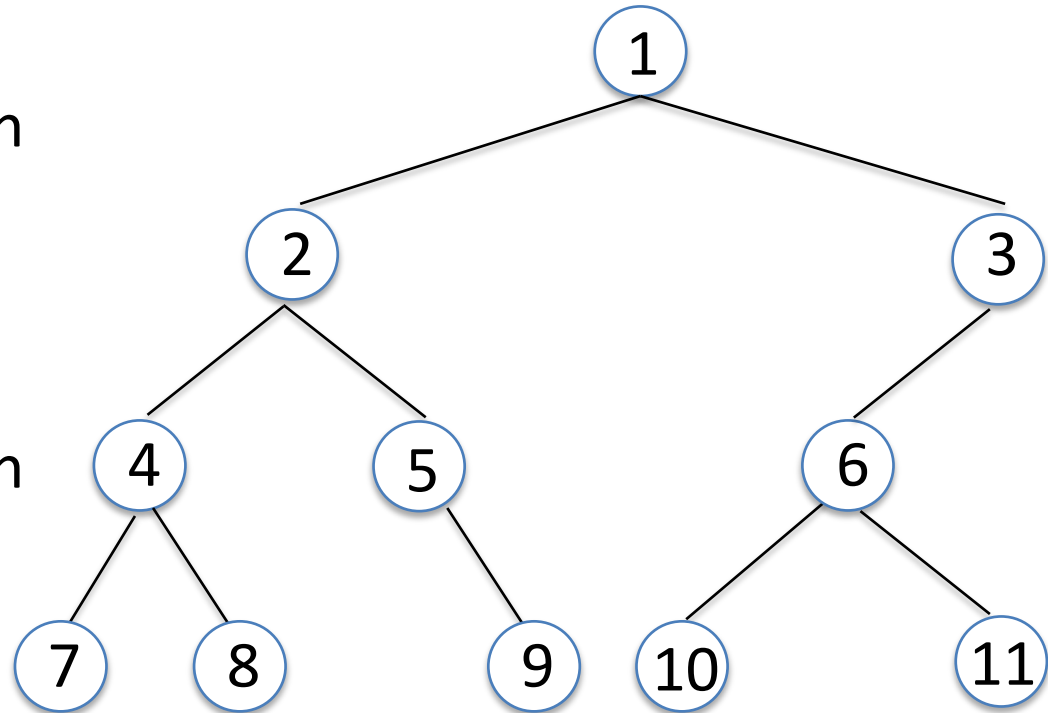


- Recorrido

7 8 4 9 5 2 10 11 6 3 1

Recorrido en inorden

1. Recorremos en inorden el hijo izquierdo
2. Visitamos la raíz
3. Recorremos en inorden el hijo derecho

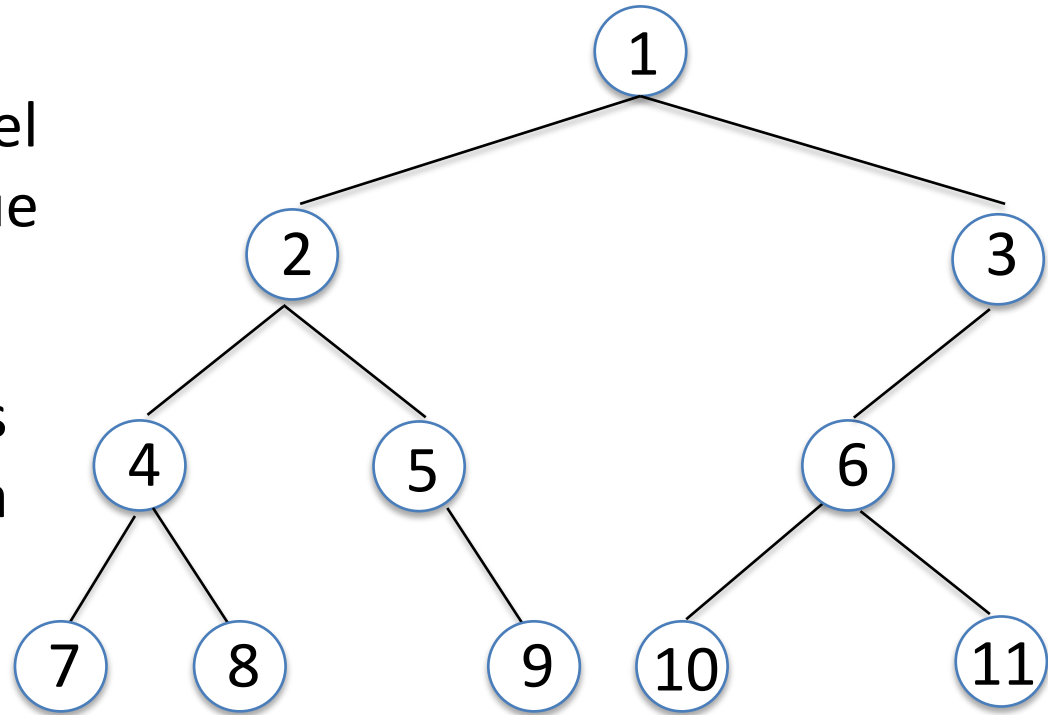


- Recorrido

7 4 8 2 5 9 1 10 6 11 3

Recorrido por niveles

- Todos los nodos del nivel k son visitados antes que los del nivel $k+1$
- En cada nivel, los nodos se visitan de izquierda a derecha



- Recorrido

1 2 3 4 5 6 7 8 9 10 11

Recorrido en preorden

```
/* Pre: true */
```

```
/* Post: El resultado es la lista en preorden de los  
elementos de t */
```

```
list<int> preorden(const BinTree <int>& t,) {  
    list<int> L;  
    if (not t.empty()) {  
        L.insert(L.begin(), t.value()),  
        L.splice(L.end(), preorden(t.left())),  
        L.splice(L.end(), preorden(t.right()));  
    return L;  
}
```

Recorrido en preorden (iterativo)

```
/* Pre: true */
/* Post: El resultado es la lista de los elementos de t
recorridos en preorden */
list<int> preorden (const BinTree <int>& t,) {
    list <int> L;
    if (not t.empty()) {
        stack <BinTree <int>> s; s.push(t);
        while (not s.empty()) {
            BinTree <int>& aux = s.top(); s.pop();
            L.insert(L.end(), aux.value()),
            if (not aux.right().empty()) s.push(aux.right()),
            if (not aux.left().empty()) s.push(aux.left());
        }
    }
    return L;
}
```

Recorrido por niveles

```
/* Pre: true */
/* Post: El resultado es la lista de los elementos de t
recorridos por niveles */
list<int> niveles (const BinTree <int>& t,) {
    list <int> L;
    if (not t.empty()) {
        queue <BinTree <int>> q; q.push(t);
        while (not q.empty()) {
            BinTree <int>& aux = q.front(); q.pop();
            L.insert(L.end(), aux.value()),
            if (not aux.left().empty()) q.push(aux.left()),
            if (not aux.right().empty()) q.push(aux.right());
        }
    }
    return L;
}
```