Programación 2
Diseño recursivo y árboles

Fernando Orejas

1. Principios del diseño recursivo
2. Árboles binarios. La clase BinTree
3. Operaciones con árboles binarios
4. Recorridos de un árbol

Principios del diseño recursivo

Algoritmos recursivos

Un algoritmo recursivo no es más que la implementación directa de una definición inductiva.

Definiciones inductivas

La idea básica de una definición inductiva es que:

1. Para algunos casos básicos definimos la función directamente (los *más pequeños*)

2. Para el resto de los casos definimos la función en base a elementos *más pequeños*.

3. Tenemos *funciones de descomposición* que nos permiten obtener esos elementos más pequeños.

Factorial

```
n! = 1 * 2 * ... * (n-1) * n
```

1. Caso base n = 0

2. El factorial de n lo podemos definir a partir del factorial de n-1

3. La función de descomposición es n-1

Definición inductiva

$$
n! = \begin{cases} 1 & n=0 \\ n*((n-1)!) & n>0 \end{cases}
$$

```
int factorial(int n){

// Pre:  n >= 0
// Post: devuelve el factorial de n

    if (n == 0) return 1;
    else return n*factorial(n-1);
}
```

Suma de los elementos de una pila

Si P = e_1 e_2 ... e_n

Suma(p) = e_1 + e_2 ... + e_n

1. Caso base: la pila está vacía
2. Suma(p) se puede definir a partir de la suma del elemento que está en la cumbre de p y del resto de la pila
3. Las funciones de descomposición son top y pop

Suma de los elementos de una pila

Si P = e_1 e_2 ... e_n

Suma(p) = e_1 + e_2 ... + e_n

$$\textbf{Suma(p)} = \begin{cases} 0 & \text{si P está vacía} \\ \text{top(P)+Suma(pop(P))} & \text{en otro caso} \end{cases}$$

```
// Pre:  true
// Post: devuelve la suma de los valores de P

int Suma(Stack <int> P) {
    if (P.empty()) return 0;
    else {
        x = P.top();
        P.pop();
        return x+Suma(P);
    }
}
```

Búsqueda en una pila

Si P = e_1 e_2 ... e_n

1. Caso base: la pila está vacía
2. busq(p,x) se puede definir a partir del elemento que está en la cumbre de p y del resto de la pila
3. Las funciones de descomposición son top y pop

Búsqueda en una pila

```
// Pre:  true
// Post: Nos dice si x está en P

bool busq(Stack <int> P, int x) {
    if (P.empty()) return false;
    else
      if (P.top() == x) return true
      else {
         P.pop();
         return busq(P,x);
      }
}
```

Diseño recursivo

1. Casos básicos

2. Caso general

Corrección de un algoritmo recursivo

Hemos de demostrar que para cada valor de los parámetros que cumpla la Pre:

- El algoritmo termina.
- Los resultados cumplen la postcondición (corrección parcial).

Análisis de terminación

Corrección parcial

Terminación de una función recursiva

Para estar seguros de que una función recursiva termina:

- Hay que estar seguros de que las funciones de descomposición realmente nos dan elementos *más pequeños,*
- Los casos base son los más pequeños de todos.

Terminación de una función recursiva

La terminación se puede garantizar usando una función $|x|$ de medida o tamaño que cumple:

- $|x|$ nos devuelve un entero
- Si $|x| \leq 0$ entonces x es un caso base
- Si y es un parámetro de una llamada recursiva, entonces $|y| < |x|$.

```
int factorial(int n){

// Pre:  n >= 0
// Post: devuelve el factorial de n

    if (n == 0) return 1;
    else return n*factorial(n-1);
}
```

- $|n| = n$

```
// Pre:  true
// Post: devuelve la suma de los valores de P

int Suma(Stack <int> P) {
    if (P.empty()) return 0;
    else {
        x = P.top();
        P.pop();
        return x+Suma(P);
    }
}
```

- |P| = P.size()

Corrección parcial de un algoritmo recursivo

Hemos de demostrar:

- Si X es un caso inicial: directamente.
- En el caso general, lo demostramos por inducción:
  - Hipótesis de inducción: Si todo X’ *más pequeño* que X usado en una llamada recursiva cumple Pre, entonces podemos suponer que f(X’) cumple Post.

Corrección parcial de un algoritmo recursivo

Hemos de demostrar:

- Si X es un caso inicial: directamente.
- En el caso general:
  - Hemos de comprobar que todo X’ usado en las llamadas recursivas cumple Pre.
  - Comprobamos que los cálculos adicionales nos garantizan que el resultado de la función cumple Post.

```
int factorial(int n){

// Pre:  n >= 0
// Post: devuelve el factorial de n

    if (n == 0) return 1;
    else return n*factorial(n-1);
}
```

- **Caso base**. $0! = 1$
- **Caso general**.
  - Si $n>0$ entonces $n-1 \geq 0$ ($n-1$ cumple la pre)
  - Si factorial(n-1) = $(n-1)! = 1*2*3*...*(n-1)$

    entonces factorial(n) = $n* 1*2*3*...*(n-1) = n!$

```
// Pre:  true
// Post: devuelve la suma de los valores de P
int Suma(Stack <int> P) {
    if (P.empty()) return 0;
    else {
        x = P.top();
        P.pop();
        return x+Suma(P);
    }
}
```

- **Caso base**. Si P está vacía la suma es 0.
- **Caso general**.

  Supongamos que P **=** $\mathbf{e_1\ \ e_2\ \ ...\ \ e_n}$

  – Si P no está vacía entonces trivialmente P.pop() cumple la pre

  – Si Suma(P.pop()) = $e_2 + ... + e_n$ entonces

  Suma (P) = P.top() + $e_2 + ... + e_n = e_1 + e_2 + ... + e_n$

```
// Exponenciación rápida

// Pre: y >= 0
// Post: Retorna x^y

int Potencia(int x, int y);
```

```
// Exponenciación rápida

// Pre: y >= 0
// Post: Retorna xʸ

int Potencia(int x, int y) {
   if (y == 0) return 1;
   else if (y % 2 == 0)
      return potencia(x*x, y/2);
   else
      return x*potencia(x, y-1);
   }
}
```

$|x,y| = y$

- **Caso base**. Si y = 0, entonces exp(x,y) = 1 = $\mathbf{x^y}$

- **Caso general**.

  – Si y > 0, y/2 e y – 1 son mayores o iguales que 0. Por tanto, los parámetros de las llamadas recursivas cumplen la precondición

  – Si y es par entonces podemos asumir que potencia(x*x, y/2) = $(x*x)^{(y/2)} = x^{2*(y/2)} = x^y$. Es decir que potencia(x, y) = x^y.

  – Si y es impar entonces podemos asumir que potencia(x, y-1) = $x^{(y-1)}$. Es decir que potencia(x, y) = $x * x^{(y-1)} = x^y$.

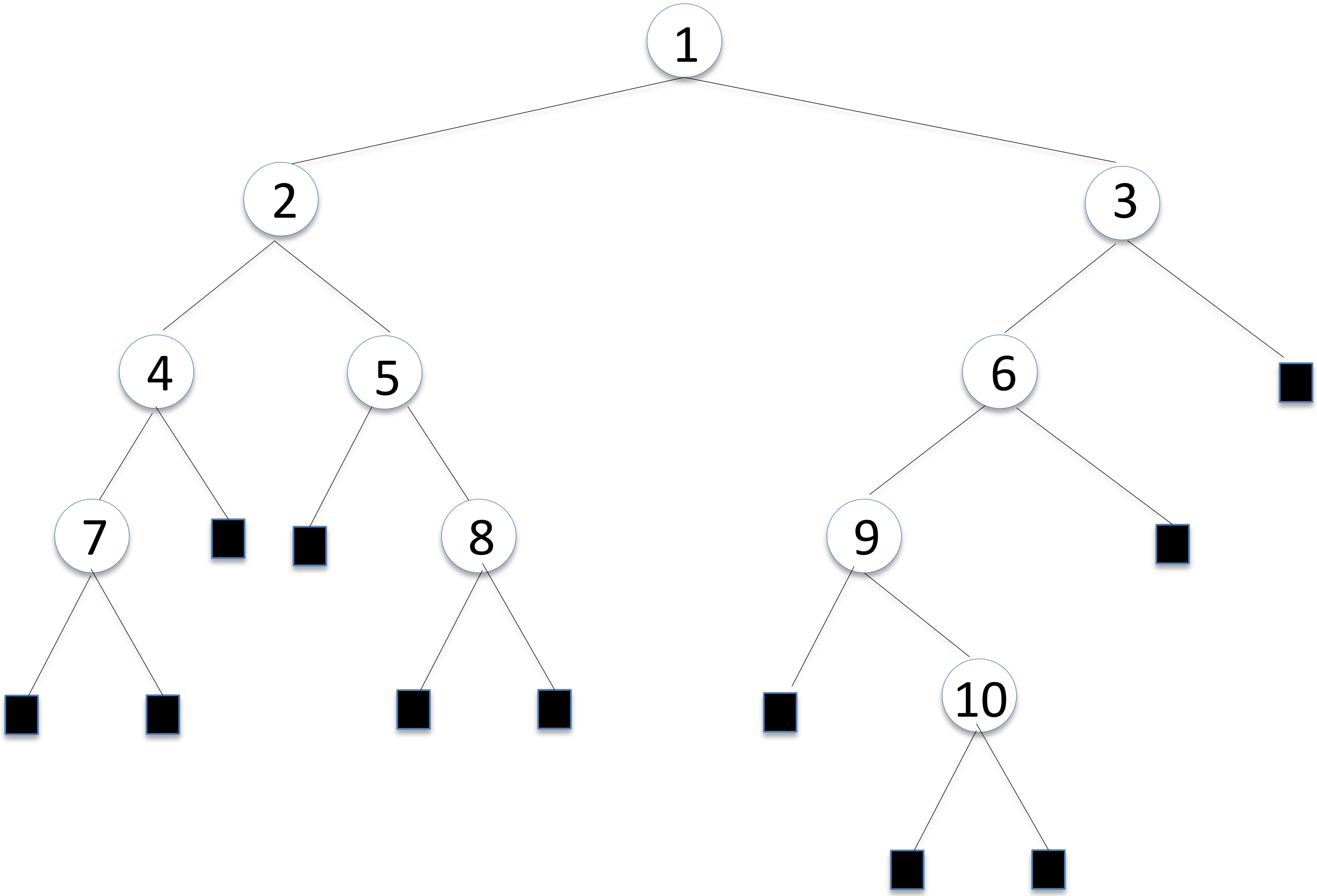
Árboles binarios

Árboles Binarios

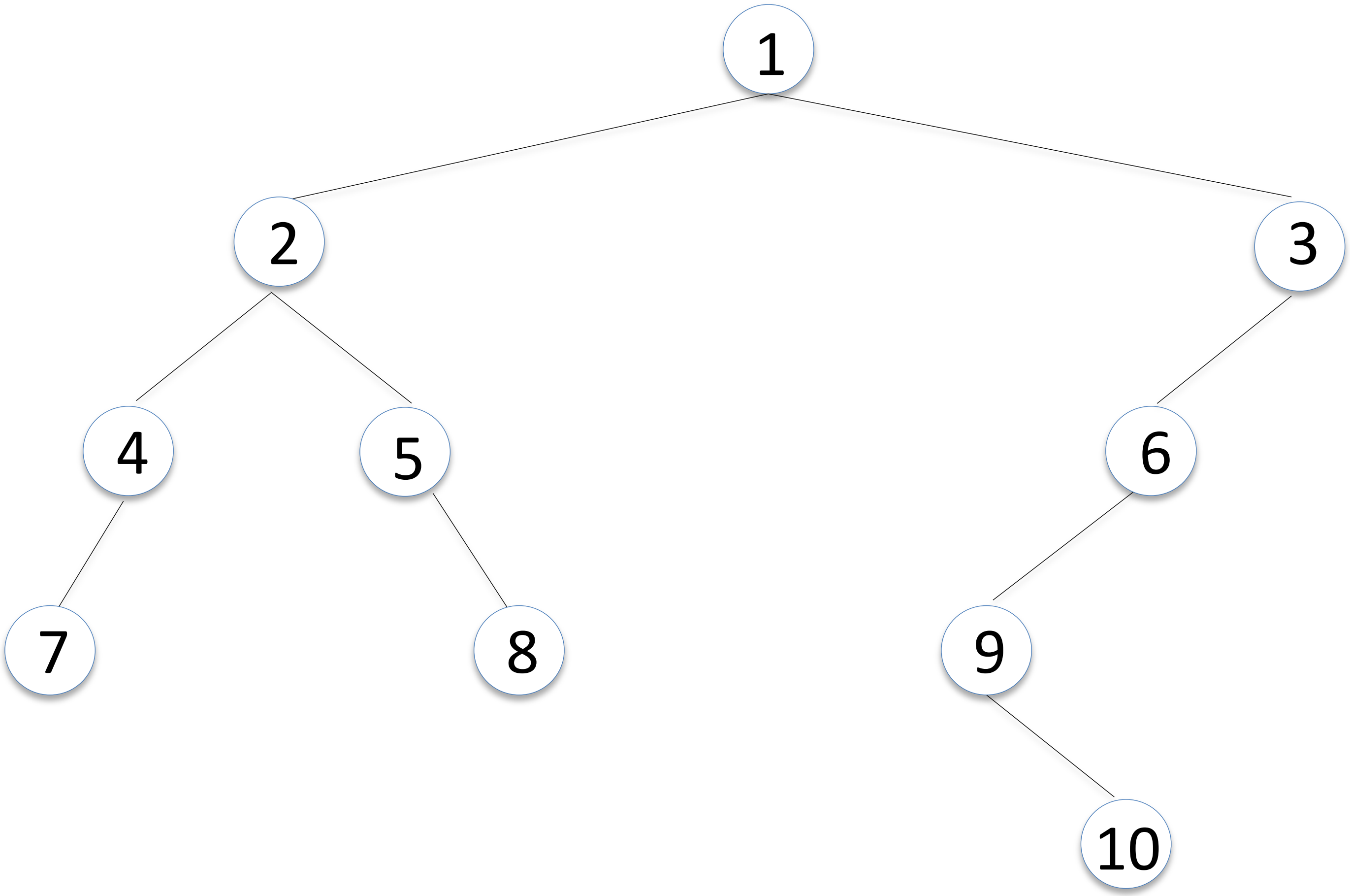
- Un árbol binario es, o bien un árbol vacío, o bien es un nodo llamado raiz, que tiene dos sucesores (subárboles) que son árboles binarios

- Los dos sucesores de un nodo son su hijo izquierdo y su hijo derecho

Árboles binarios

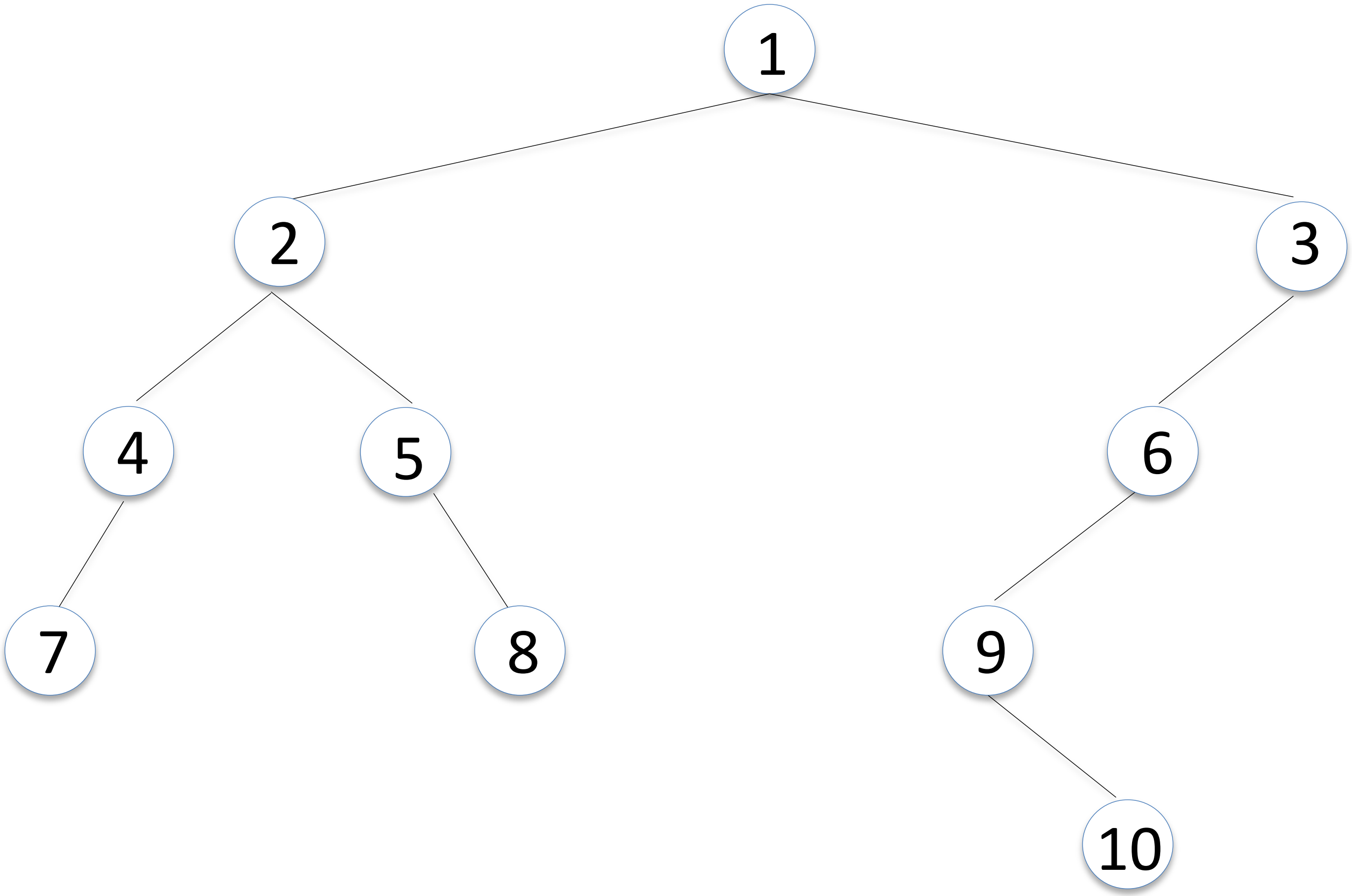


Árboles binarios



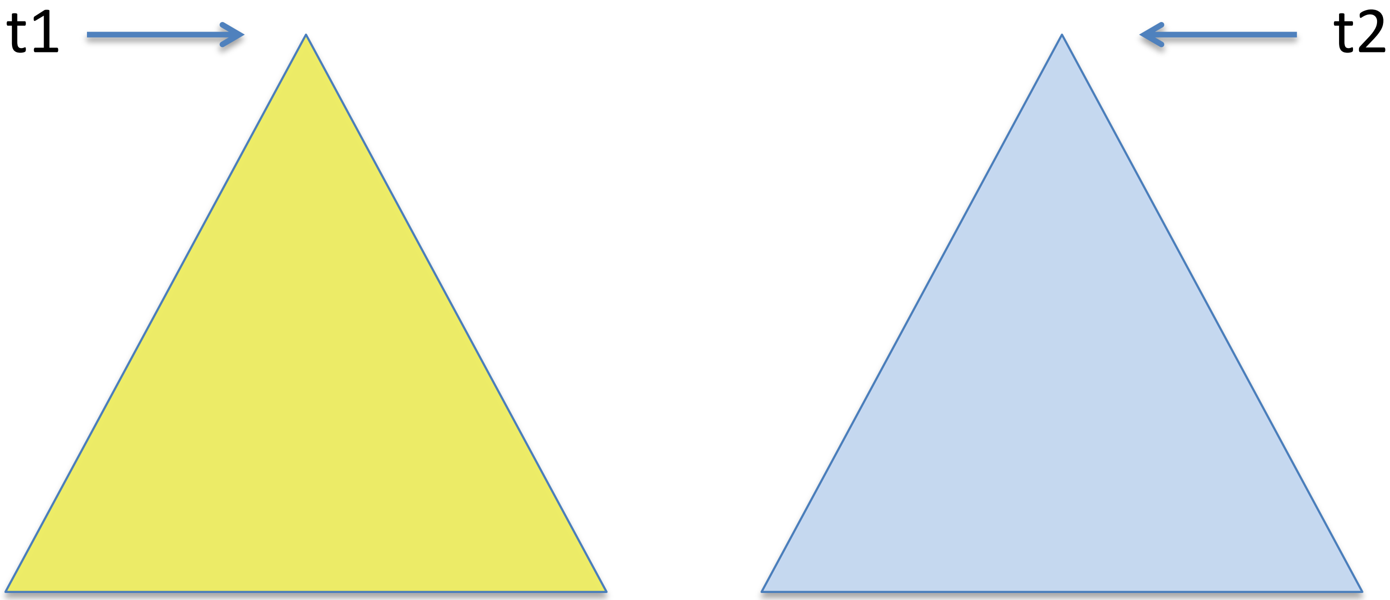
Terminología

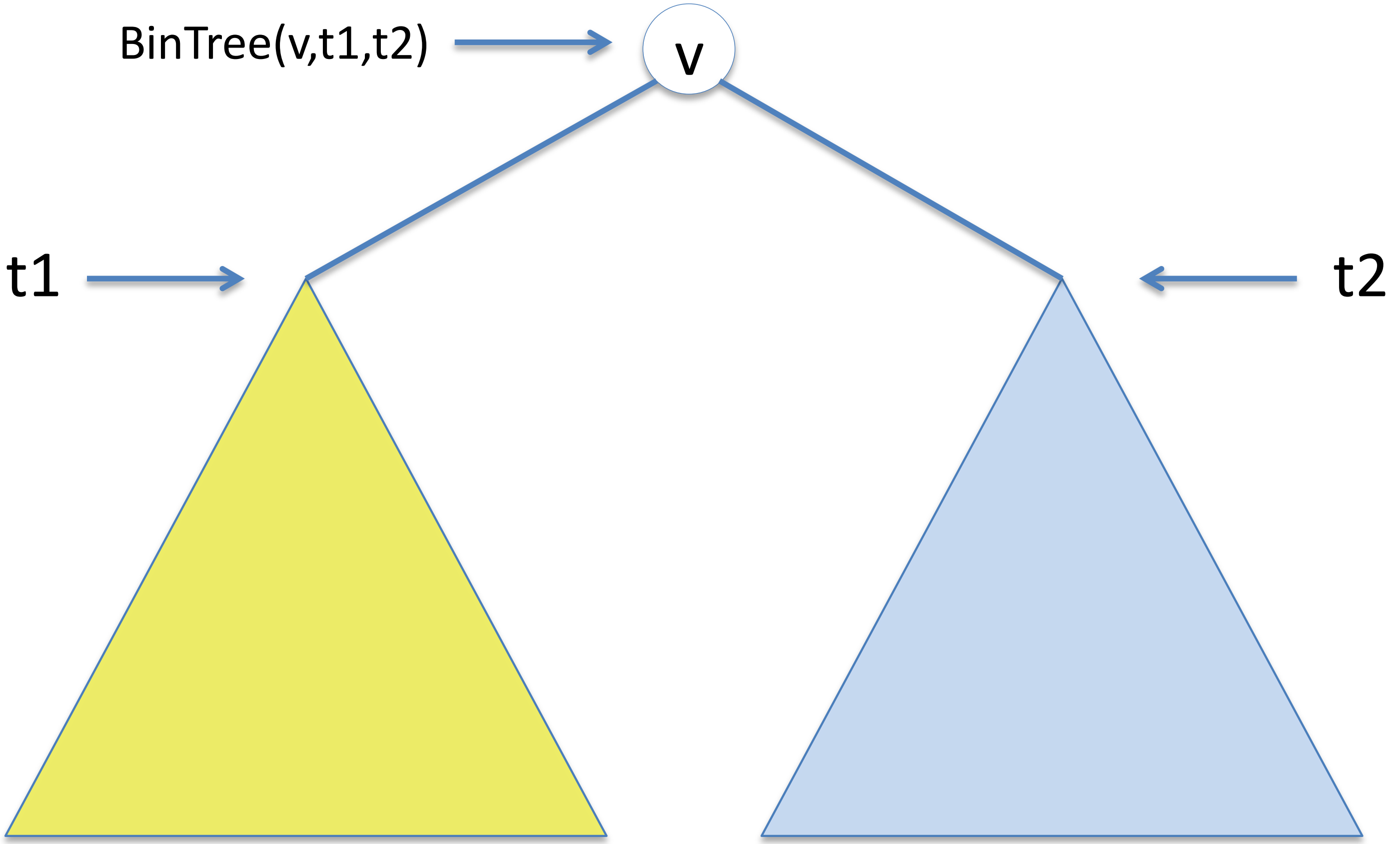
- nodo
- padre, hijo
- hijo mayor, hijo menor
- ascediente descendiente
- hermano
- raiz, hoja, rama
- camino
- nivel, altura



Especificación de la clase BinTree

```
template <class T> class BinTree {
    public:
    // Constructoras
    // Pre: true
    // Post: crea un árbol vacío
    BinTree ();
    // Pre: true
    // Post: crea un árbol con x como raiz, y árboles vacíos
    // como hijos
    BinTree (const T& x);
    // 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);
```


t1
t2



BinTree(v,t1,t2)
v
t1
t2

```
// 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
T value ();
```

Operaciones de BinTree

- BinTree no tiene modificadoras: la única manera de modificar un árbol binario es construir un arbol modificado y asignarlo al árbol original.
- Todos los métodos que hemos visto se ejecutan en tiempo constante.
- ***No se hacen copias de los hijos.***

Operaciones con árboles binarios

Tamaño de un árbol

```
template <typename T>
/* Pre: true */
/* Post: retorna el número de nodos del árbol t*/
int size(const BinTree <T>& t);
```

Tamaño de un árbol

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

- **Función de medida**. |t| = número de nodos de t
- **Caso base**. Si t está vacío el resultado es 0.
- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición
  - Podemos asumir que size(t.left()) es el número de nodos del hijo izquierdo de t y que size(t.right()) es el número de nodos del hijo hijo derecho de t.
  - Por tanto 1+ size(t.left())+ size(t.right()) es el número de nodos de t.

Altura de un árbol

```
/* Pre: true */
/* Post: retorna la altura del árbol t*/
int size(const BinTree <T>& t);
```

Altura de un árbol

```
/* Pre: true */
/* Post: retorna la altura del árbol t*/
int altura(const BinTree <T>& t){
  if (t.empty()) return 0;
  else return 1+max(altura(t.left()),altura(t.right());
}
```

- **Función de medida**. |t| = número de nodos de t
- **Caso base**. Si t está vacío el resultado es 0.
- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición
  - Podemos asumir que altura(t.left()) es la altura del hijo izquierdo de t y que altura(t.right()) es la altura del hijo derecho de t.
  - Por tanto 1+ max(altura(t.left()), altura(t.right()) es la altura de t.

Búsqueda en un árbol

```
/* Pre: true */
/* Post: nos dice si x está en t*/
bool busq(const BinTree <T>& t, int x);
```

Búsqueda en un árbol

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

- **Función de medida**. |t| = número de nodos de t

- **Caso base**. Si t está vacío el resultado es false.

- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición

  - Podemos asumir que busq(t.left(),x) nos dice si x está en el hijo izquierdo de t y que busq(t.right(),x) nos dice si x está en el hijo izquierdo de hijo derecho de t.

  - Por tanto (t.value == x) or busq(t.left(),x) or busq(t.right(),x) nos dice si x está en t.

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 <int> sumak(const BinTree <int>& t, int k);
```

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 <int> sumak(const BinTree <int>& t, int k){
   if (t.empty()) return t;
   else return BinTree(t.value()+k,
                          sumak(t.left(),k),
                          sumak(t.right(),k));
}
```

- **Caso base**. Si t está vacío, entonces el resultado es el propio t.

- **Caso general**.

  - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición

  - Podemos asumir que sumak(t.left()) es el resultado de sumar k a todos los valores en el hijo izquierdo de t.

  - También podemos asumir que sumak(t.right()) es el resultado de sumar k a todos los valores en el hijo derecho de t.

  - Por tanto BinTree (t.value()+k, sumak(t.left()), sumak(t.right())) es el árbol resultante de sumar k a todos los valores en t.

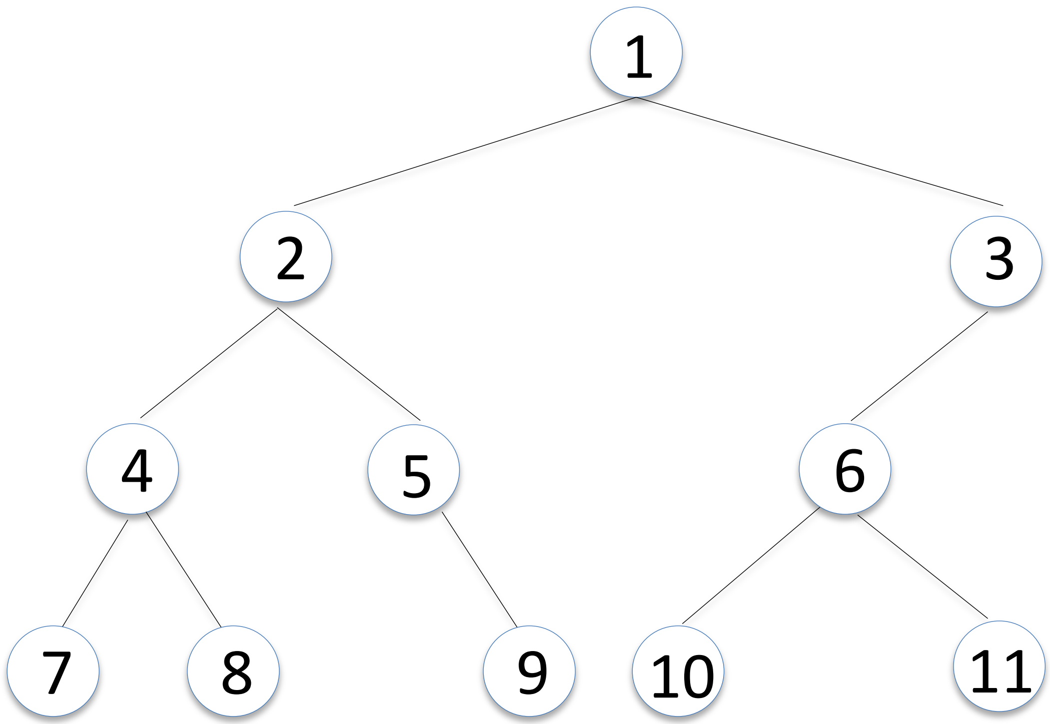
Recorridos

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

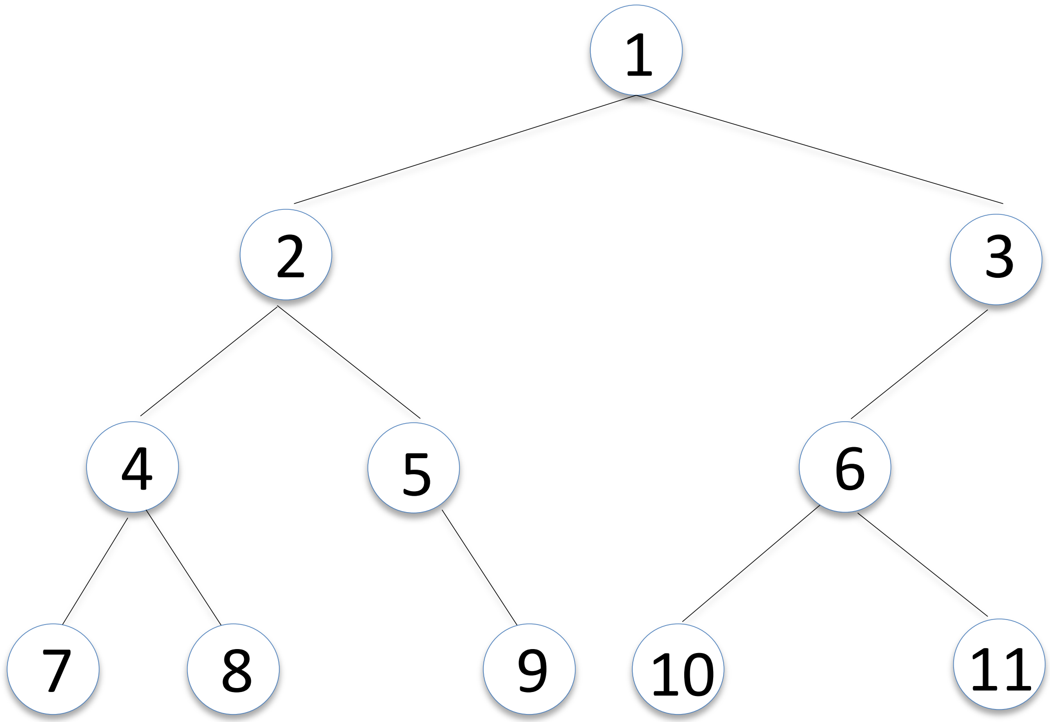


- Recorrido

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 raiz

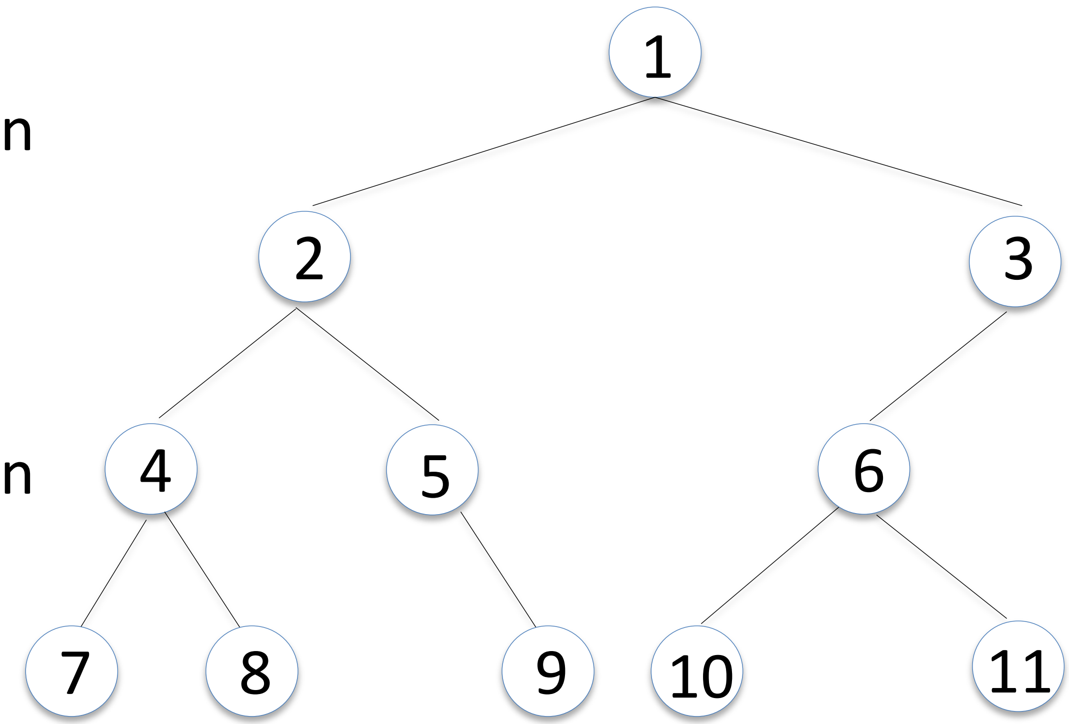


- Recorrido

7 8 4 9 5 2 10 11 6 3 1

Recorrido en inorden

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



- Recorrido

7 4 8 2 5 9 1 10 6 11 3

Recorrido en preorden

```
/* Pre: true */
/* Post: El resultado es la lista en preorden de los
elementos de t */
list<T> preorden(const BinTree <T>& t,) {
   list<T> L;
   if (not t.empty()) {
      L.insert(L.end(), t.value()),
      L.splice(L.end(), preorden(t.left())),
      L.splice(L.end(), preorden(t.right()));
   return L;
}
```

- **Función de medida**. |t| = número de nodos de t

- **Caso base**. Si t está vacío el resultado es falsela lista vacía.

- **Caso general**.
  - Los parámetros de las llamadas recursivas cumplen trivialmente la precondición

  - Podemos asumir que preorden(t.left()) nos retorna una lista que contiene el recorrido en preorden del hijo izquierdo de t y lo mismo para preorden(t.right())

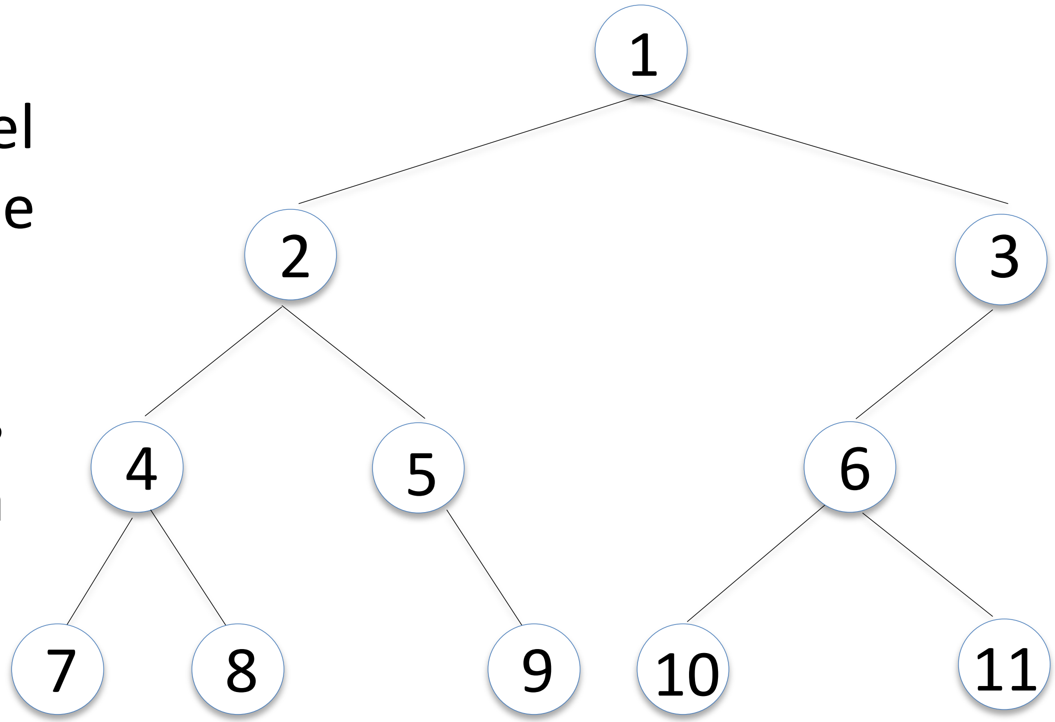
  - Si retornamos la lista resultante de a) insertar al final el valor de la raiz de t b) añadir por el final la lista que contiene el preorden del hijo izquierdo de t y c) añadir por el final la lista que contiene el preorden del hijo izquierdo de t , retornaremos la lista que contiene el recorrido en preorden de t..

Recorrido en inorden

```
/* Pre: true */
/* Post: El resultado es la lista en preorden de los
elementos de t */
list<T> preorden(const BinTree <T>& t,) {
    list<T> L;
    if (not t.empty()) {
       L.splice(L.end(), preorden(t.left())),
       L.insert(L.end(), t.value()),
       L.splice(L.end(), preorden(t.right()));
    return L;
}
```

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



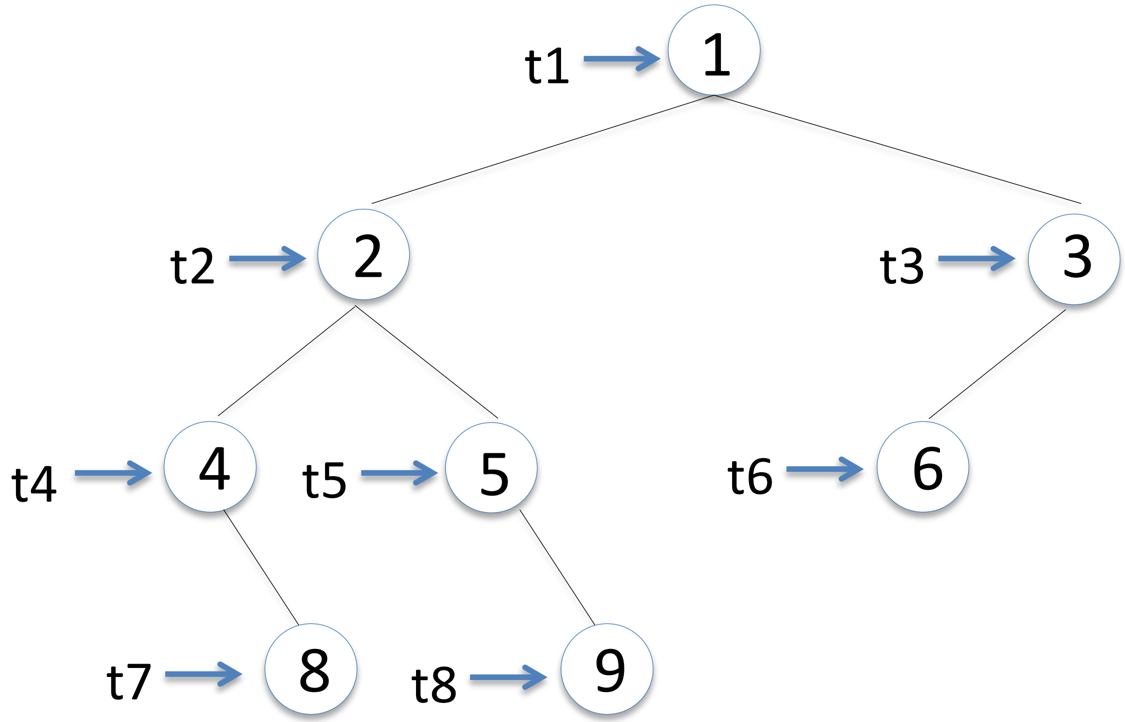
- Recorrido

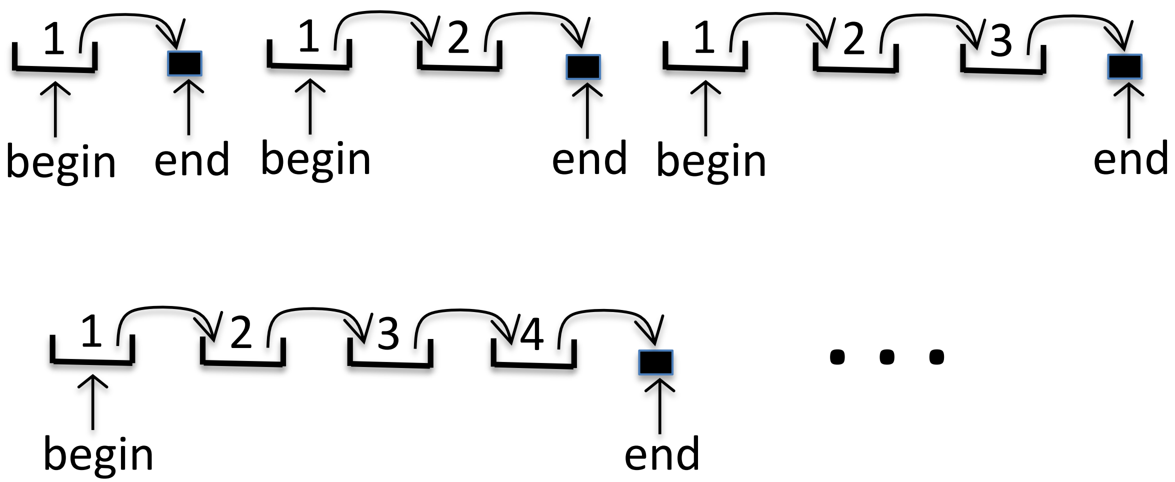
1 2 3 4 5 6 7 8 9 10 11

Recorrido 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;
}
```

Recorrido por niveles





t1		
t2	t3	
t3	t4	t5
t4	t5	t6
t5	t6	t7
t6	t7	t8
t7	t8	
t9		