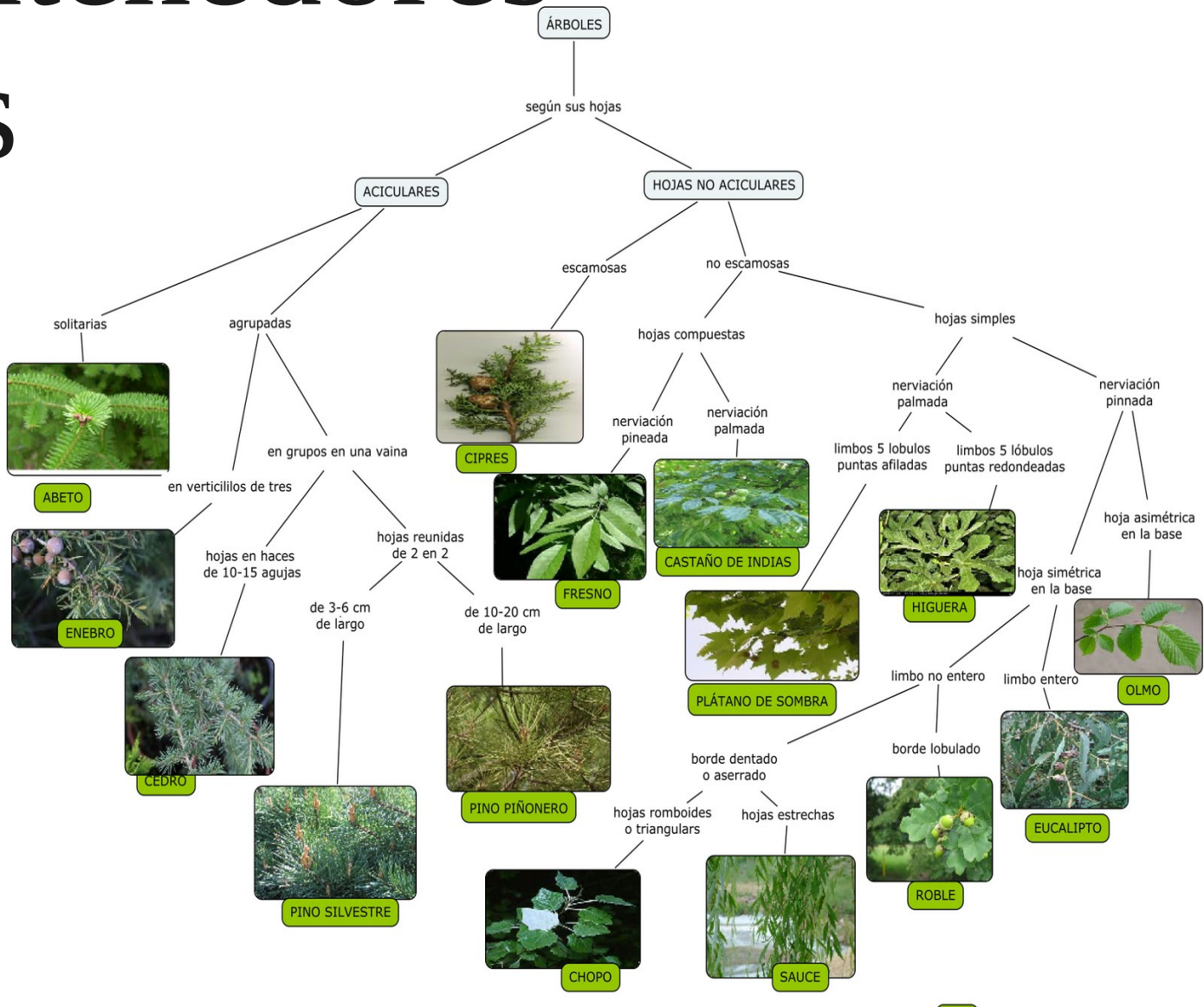
Módulo 4

TDAs Contenedores Complejos



Objetivos

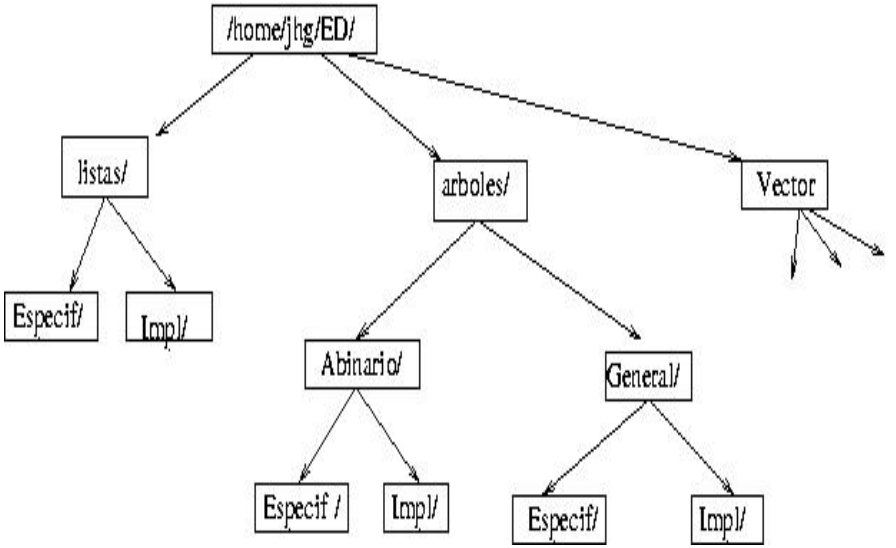
- Comprender el concepto de *árbol*
- Conocer la terminología y las formas de recorrer el contenido de un árbol
- Conocer las especificaciones de los TDAs *tree* y *bintree*
- Manejar los ABB, APO, AVL
- Introducir el concepto de *grafo* y su especificación

Árboles

Árboles

Es una estructura jerárquica:

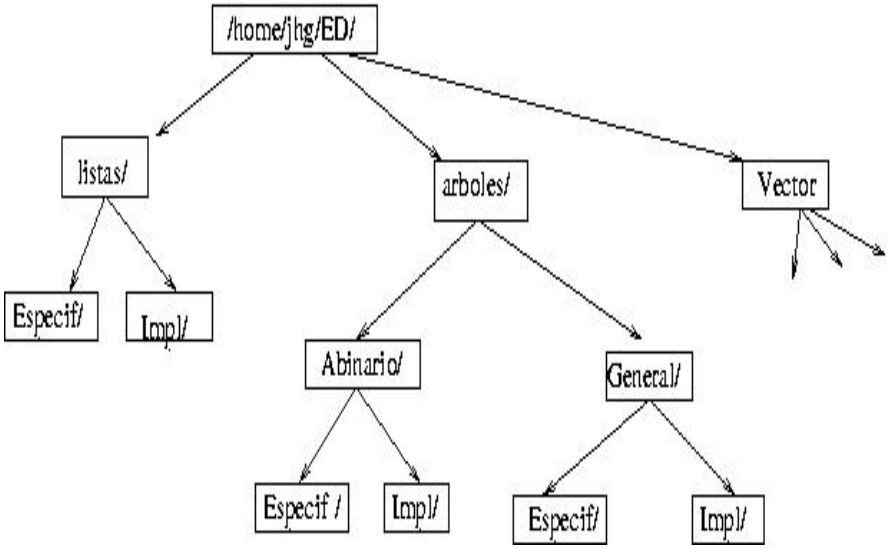
- Para cada elemento no hay tan sólo un anterior y un siguiente.
- Existen elementos por encima (padres) y por debajo (hijos).



Árboles

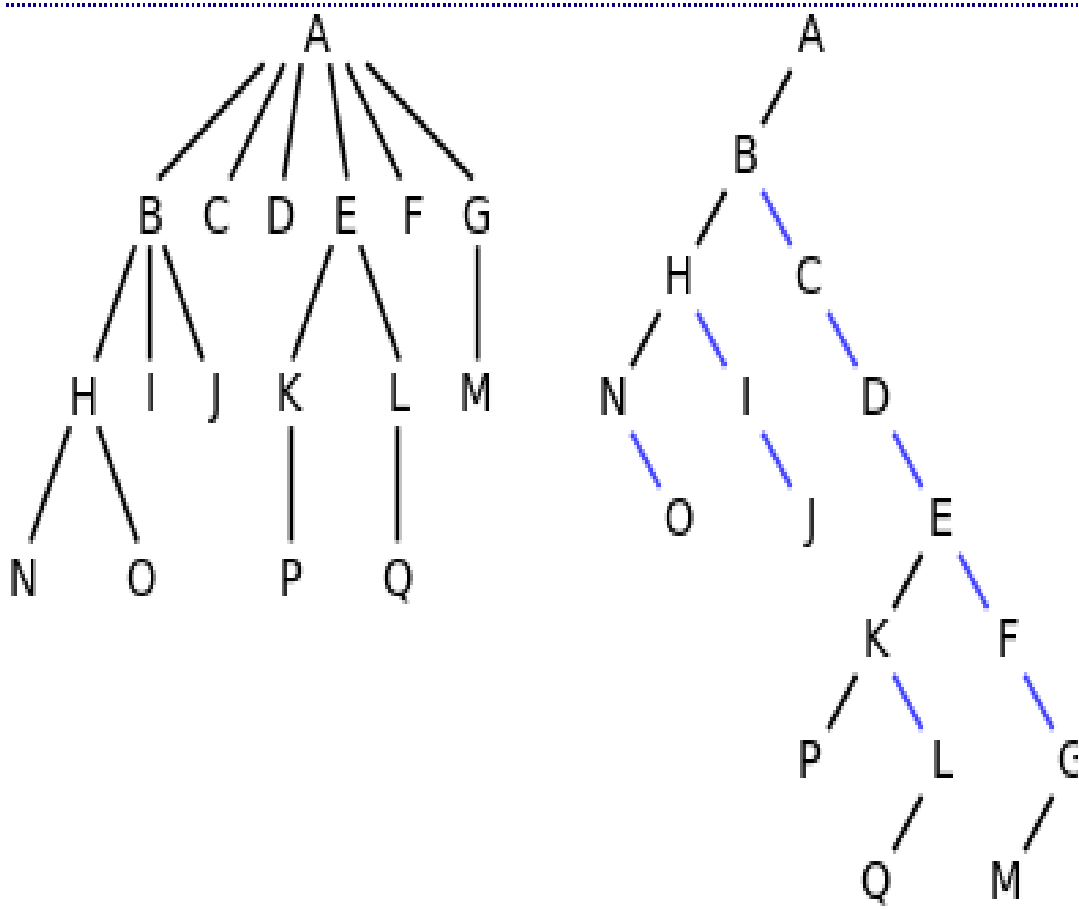
Un **nodo** es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él.

Un **árbol** se puede ver como un conjunto de nodos, sobre los que se establece la relación jerárquica (mediante una relación padre-hijo)



Árboles

- **Raíz.** Es un nodo que no tiene padres. Sólo puede haber una raíz en el árbol.
- **Hoja** Es un nodo que no tiene hijos
- **Árbol Binario**: Un nodo puede tener a lo sumo dos hijos
- **Árbol General**: Un nodo puede tener cualquier número de hijos.



Árbol: Definición recursiva

Formalmente, podemos definir un árbol como:

- Caso base: un árbol con sólo un nodo (es a la vez raíz del árbol y hoja).
- Caso Recursivo:
  - Dados un árbol A0 con un sólo nodo (con raíz r) y k árboles A1, A2, ... Ak de raíces n1, n2, ..., nk
  - Podemos construir un nuevo árbol, A, estableciendo una relación padre-hijo entre r y cada una de las raíces de los k árboles.
  - Propiedades.
    - A.size() = 1 + A1.size() + ... + Ak.size();
    - A.hojas() = A1.hojas() U ... U Ak.hojas()
  - A cada uno de los árboles A_i se les denota ahora subárboles de la raíz.

Más Terminología

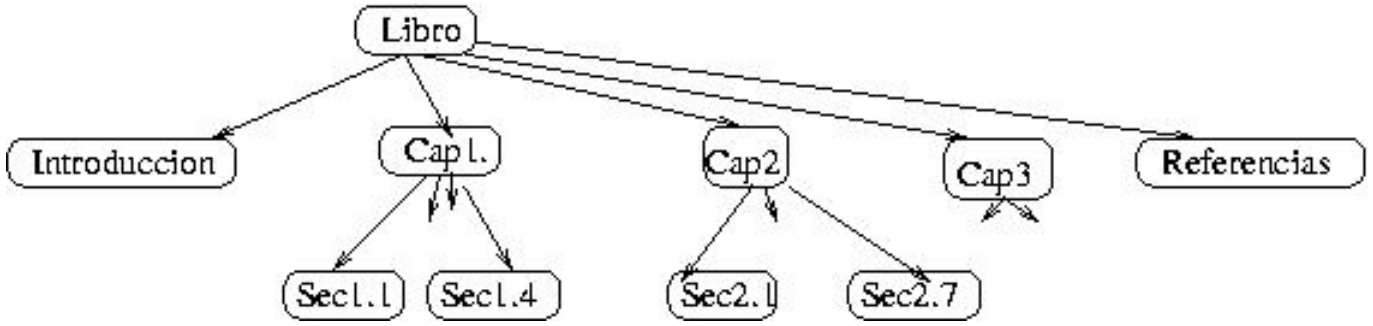
- Dos nodos que son hijos de un mismo nodo se llaman *hermanos*. Pueden estar ordenados de izquierda a derecha y en ese caso diremos que tenemos un árbol ordenado.
- Un nodo v es *ancestro* de un nodo n si se puede llegar desde n hasta v siguiendo la relación padre.
- Un nodo v es *descendiente* de un nodo n si se puede llegar desde n hasta v siguiendo los hijos.
- Un nodo n es un *interior* si tiene hijos. Un nodo n es un *exterior* si no tiene hijos (hoja).
- Un nodo puede contener información, a la que se llama *etiqueta* del nodo.
- Un *árbol etiquetado* es aquél cuyos nodos poseen etiquetas.

Más Terminología

- La *profundidad de un nodo* es el número de relaciones padre que hay que seguir para llegar desde ese nodo hasta el nodo raíz. La raíz tiene profundidad cero.
- Un *nivel* consiste en el conjunto de nodos que se encuentran a la misma profundidad.
- La *altura de un nodo* es el número de relaciones hijo que hay que seguir hasta alcanzar su descendiente hoja más lejano. Una hoja tiene altura cero.
- La *altura de un árbol* es la altura de su nodo raíz.
- Tenemos un *subárbol* de un árbol si cogemos un nodo y todos sus descendientes.
- Un *árbol parcial de un árbol* es un árbol que tan sólo tiene algunos de los nodos del árbol original

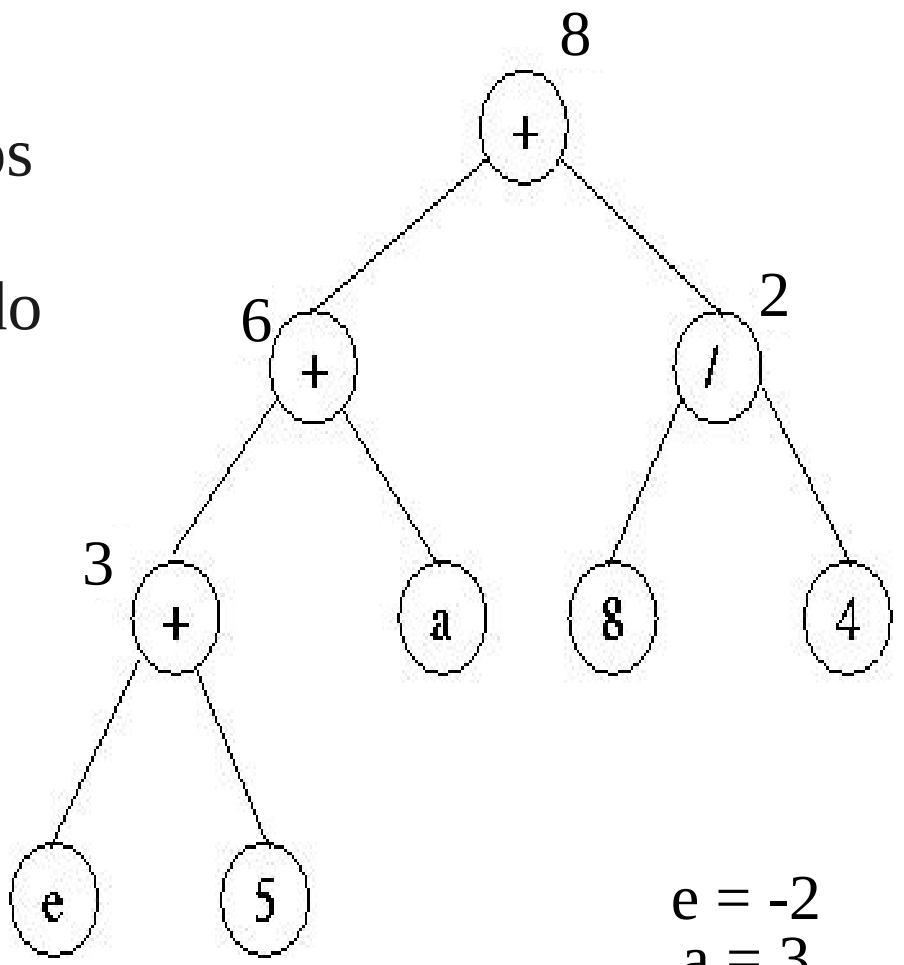
Ejemplo 1

- Un documento estructurado (libro) se organiza jerárquicamente como un árbol ordenado cuyos nodos internos son los capítulos, secciones y subsecciones y cuyos nodos externos son los párrafos.



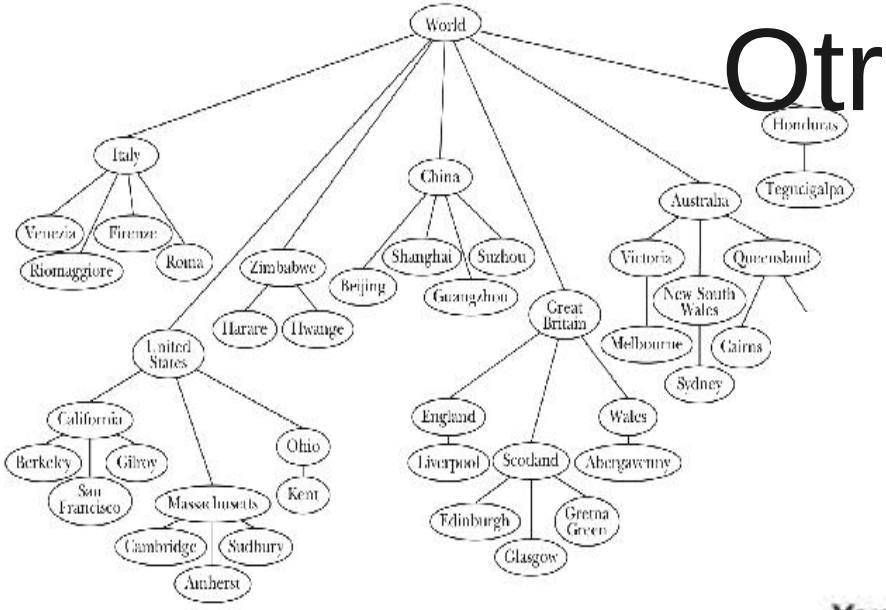
Ejemplo 2

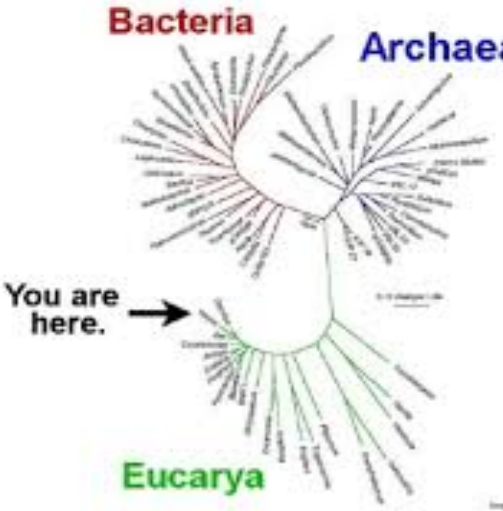
- Una expresión aritmética se puede representar como un árbol cuyos nodos externos son variables o constantes y cuyos nodos internos son operadores (+,*,/,-).
- Cada nodo de árbol tiene asociado un valor:
  - Si es externo, el que indica la variable o constante
  - Si es interno, el valor se determina al aplicar las op. para cada uno de sus hijos.

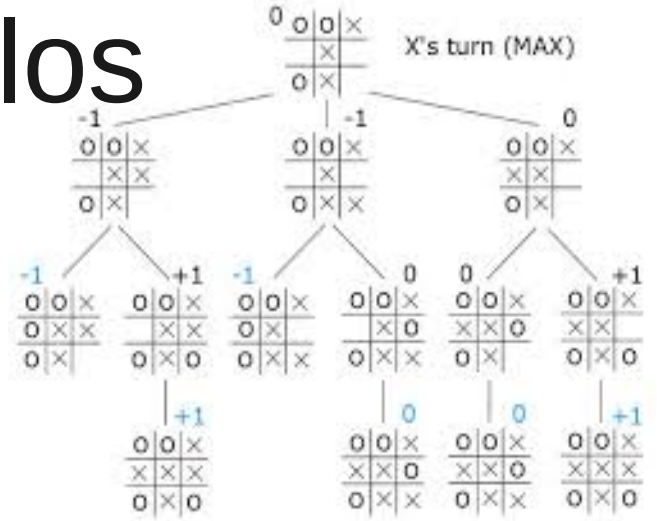


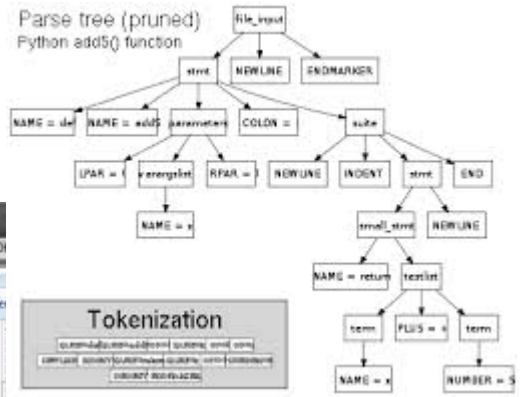
e = -2
a = 3

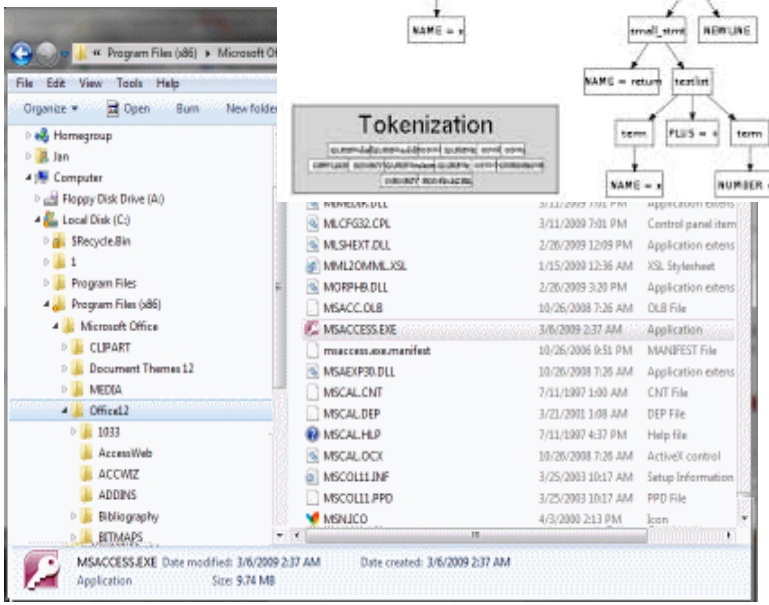
Otros Ejemplos

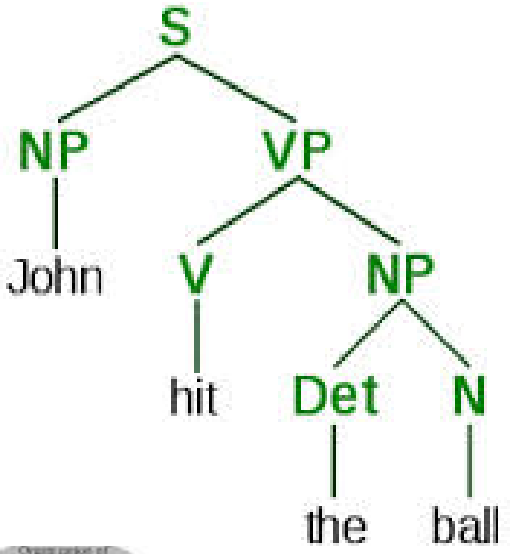


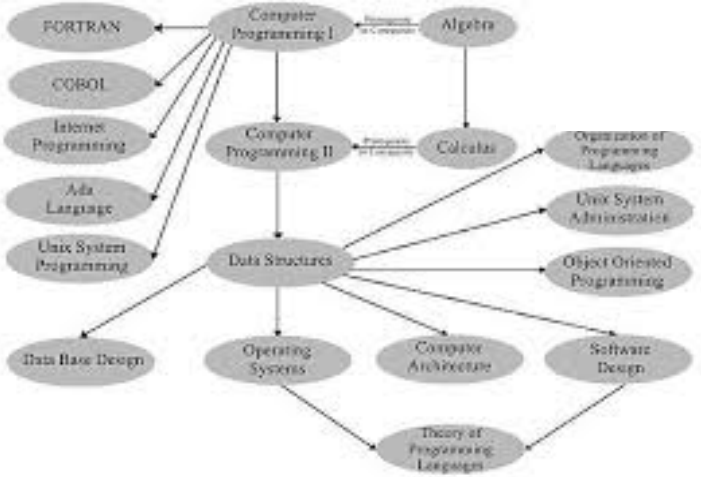




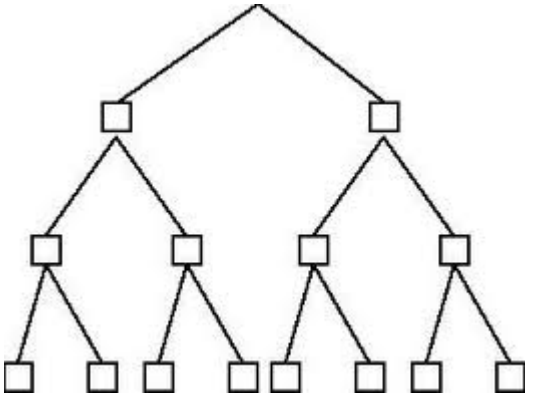






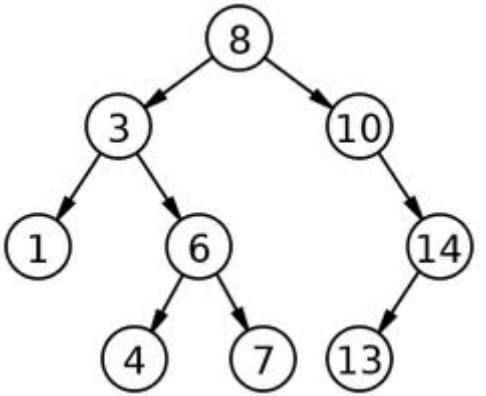


Tipos de Árboles

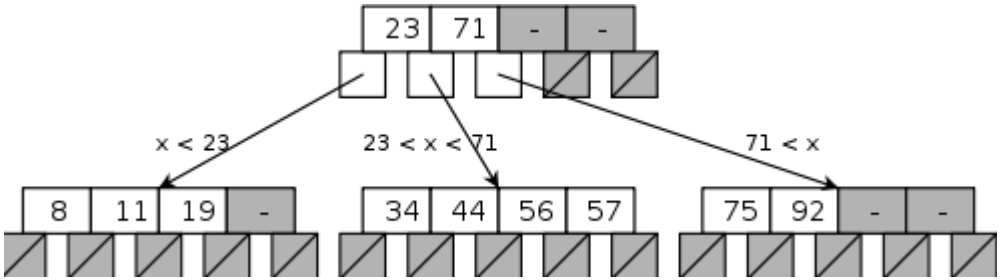


- Árboles Binarios
  - No consideran una relación orden sobre las etiquetas
    - Bintree
  - Consideran una relación de orden sobre etiquetas
    - Árbol Binario de Búsqueda

      Árbol de búsqueda binario auto-balanceable
      - Árboles AVL
      - Árboles Rojo-Negro:
        - Representación de Set y Map en STL



- Árboles Generales
  - No consideran una relación orden sobre las etiquetas
    - Tree
  - Consideran una relación de orden sobre etiquetas
    - Árboles B Árboles de búsqueda multicamino autobalanceados)
    - Árbol-B+
    - Árbol-B*



Operaciones usuales sobre árboles

- Enumerar todos los elementos.
- Buscar un elemento.
- Dado un nodo, listar los hijos (si los hay).
- Borrar un elemento.
- Eliminar un subárbol (algunas veces llamada podar).
- Añadir un subárbol (algunas veces llamada injertar).
- Encontrar la raíz desde cualquier nodo.

Tipo de Dato Árbol

- El TDA árbol almacena elementos en los nodos de éste. Así, un nodo de un árbol se puede ver como el equivalente a una posición del TDA list.

  Tipos:

  - tree<T> => tree<int> A;
  - tree<T>::node => tree<int>::node n;

Tipo de Dato Árbol (2)

- TDA tree
  - node root() const; Devuelve la raíz del árbol;
  - bool is_root(node v) const;

    Devuelve true si v es la raíz del árbol, falso en caso contrario;
  - bool is_internal(node v) const;

    Verdadero si el nodo es interno, false en caso contrario
  - bool is_external(node v) const;

    Verdadero si el nodo es externo, false en caso contrario
  - size_type size() const ; devuelve en número de nodos de un árbol.
  - .....

TDA node

- Dado un nodo, un TDA nodo debe presentar métodos que permita movernos por el árbol.
  - node parent() const;

    devuelve el padre del nodo en el árbol o nodo nulo si es la raíz del árbol.
  - node left() const;

    devuelve el hijo izquierda en el árbol, o nodo nulo si no tiene.
  - node next_sibling() const;

    devuelve el hermano derecha en el árbol, o nodo nulo no tiene.
  - T & operator*();

    devuelve la etiqueta del nodo (también existe la versión constante const T& operator*() const)
  - bool null() const;

    devuelve si el nodo es nulo (no confundir con que valga NULL!!)

Moviéndonos por un árbol

```
int profundidad(const tree<T> & A, const
  typename tree<T>::node &v)
{
   int prof = 0;
   typename tree<T>::node aux=v;
   while (!A.is_root(aux)) {
     prof++;
     aux = aux.parent();
   }
   return prof;
}
```

Moviéndonos por un árbol (2)

- Altura del nodo v: (Máxima profund. de un nodo externo)

  Definición recursiva:
  - Si v es externo, la altura es 0.
  - En otro caso, uno más la máxima altura de los hijos de v.

Moviéndonos por un árbol (3)

- Implementación (Orden O(n))

```
int altura(const tree<T> & A, const
 typename tree<T>::node &v)
{    if (A.is_external(v)) return 0;
     else {
        int alt = 0;
        tree<T>::node aux;
        for (aux = v.left(); !aux.null();
            aux = aux.next_sibling())
            alt = max(alt,altura(A,aux));
        return 1+alt; }
}
```

Recorridos

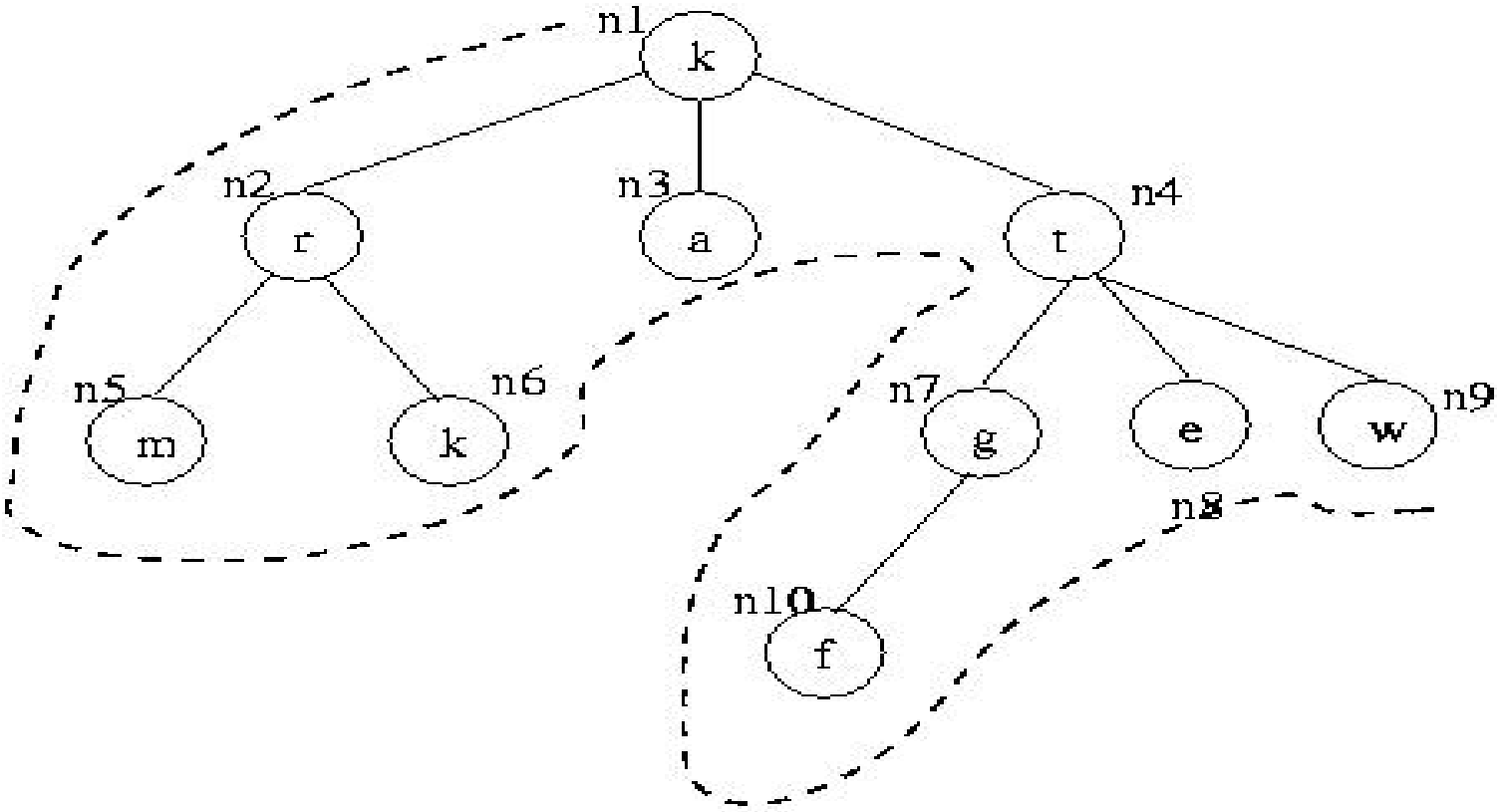
- Mecanismo por el cual podemos visitar (acceder) todos los nodos de un árbol.
- No existe un único criterio.
  - *Preorden*:
    Visitar primero la raíz y luego visitar en preorden cada uno de los subárboles que son “hijos” del nodo raíz
  - *Postorden*:
    Visitar primero en postorden cada uno subárboles “hijos” del nodo raíz y finalmente visitar la raíz

Recorridos (2)

- *Inorden*:
  Visitar primero en inorden el subárbol izquierdo, después la raíz y después en inorden el resto de los hijos.
- *En anchura*:
  Visitar en orden los nodos de profundidad 0, después los de profundidad 1, profundidad 2, ...

Preorden

Visitar primero la raíz y luego visitar en preorden cada uno de los subárboles que son “hijos” del nodo raíz



n1 n2 n5 n6 n3 n4 n7 n10 n8 n9

parentizado: n1(n2 (n5 n6) n3 n4 (n7 (n10) n8 n9)

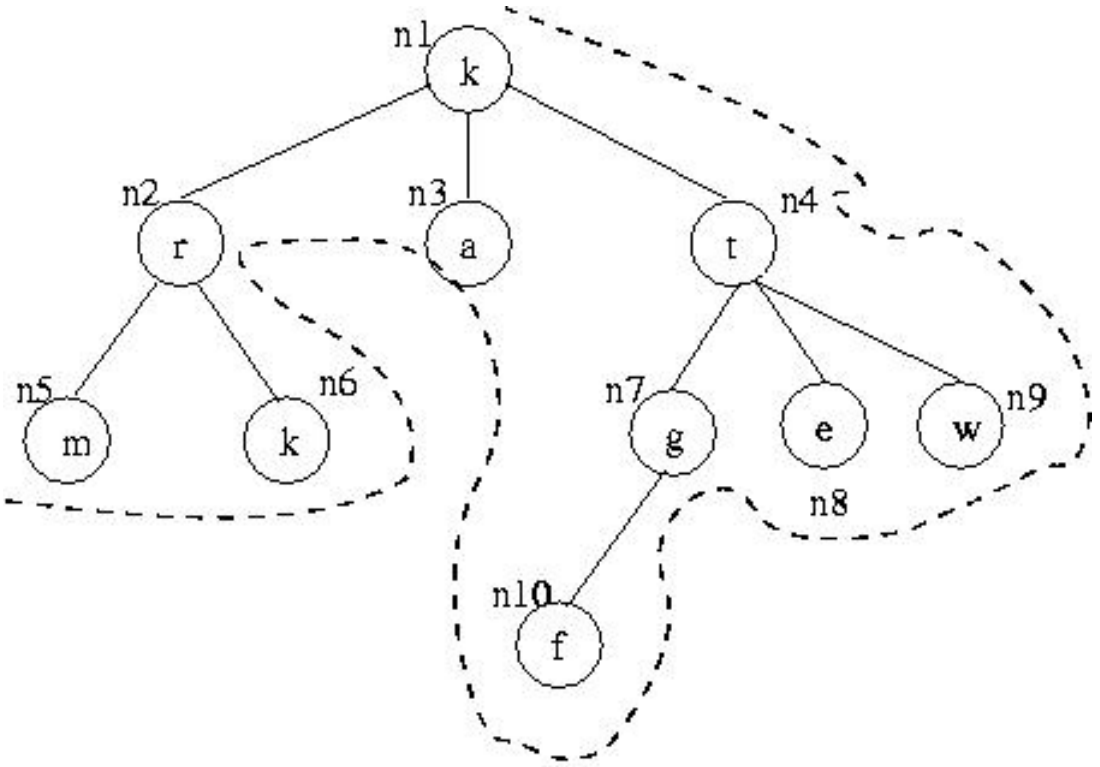
Algoritmo preorden, O(n)

```
void preorden(const tree<T> & A, const typename
  tree<T>::node &v)
{
  typename tree<T>::node aux;
  if (!v.null()) {
     cout << *v;   //  acción sobre el nodo v.
     for (aux = v.left(); !aux.null();
        aux = aux.next_sibling())
           preorden(A, aux);
  }
}
```

- Produce un orden lineal de los nodos donde un nodo aparece antes que los hijos.
- Si A es un documento estructurado, entonces preorden(A, A.root()) examina el documento secuencialmente, del principio al final.

Postorden

Visitar primero en postorden cada uno subárboles “hijos” del nodo raíz y finalmente visitar la raíz



n5 n6 n2 n3 n10 n7 n8 n9 n4 n1

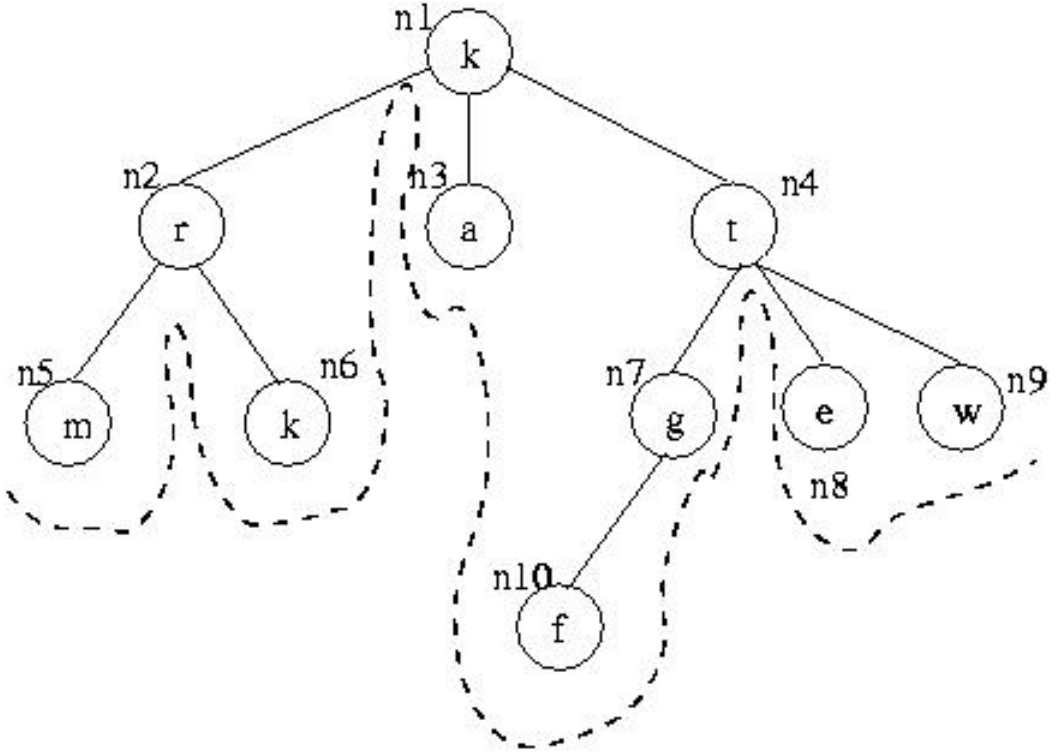
Algoritmo postorden, O(n)

```
void postorden(const tree<T> &A, const typename
  tree<T>::node &v)
{
   typename tree<T>::node aux;
   if (!v.null ()) {
      for (aux = v.left(); !aux.null();
        aux =aux.next_sibling())
         postorden(A, aux);
      cout << *v;    //  acción sobre el nodo v.
   }
}
```

- Produce un orden lineal de los nodos donde un nodo aparece después que los hijos.
- Si A es un árbol de directorios donde los nodos externos representan los ficheros, entonces postorden(A, v) nos permite conocer el espacio de disco ocupado por v.

Inorden

Visitar primero en inorden el subárbol izquierdo, después la raíz y después en inorden el resto de los hijos.



n5 n2 n6 n1 n3 n10 n7 n4 n8 n9

parentizado: ((n5) n2 (n6)) n1 (n3 ((n10) n7) n4 (n8 n9))

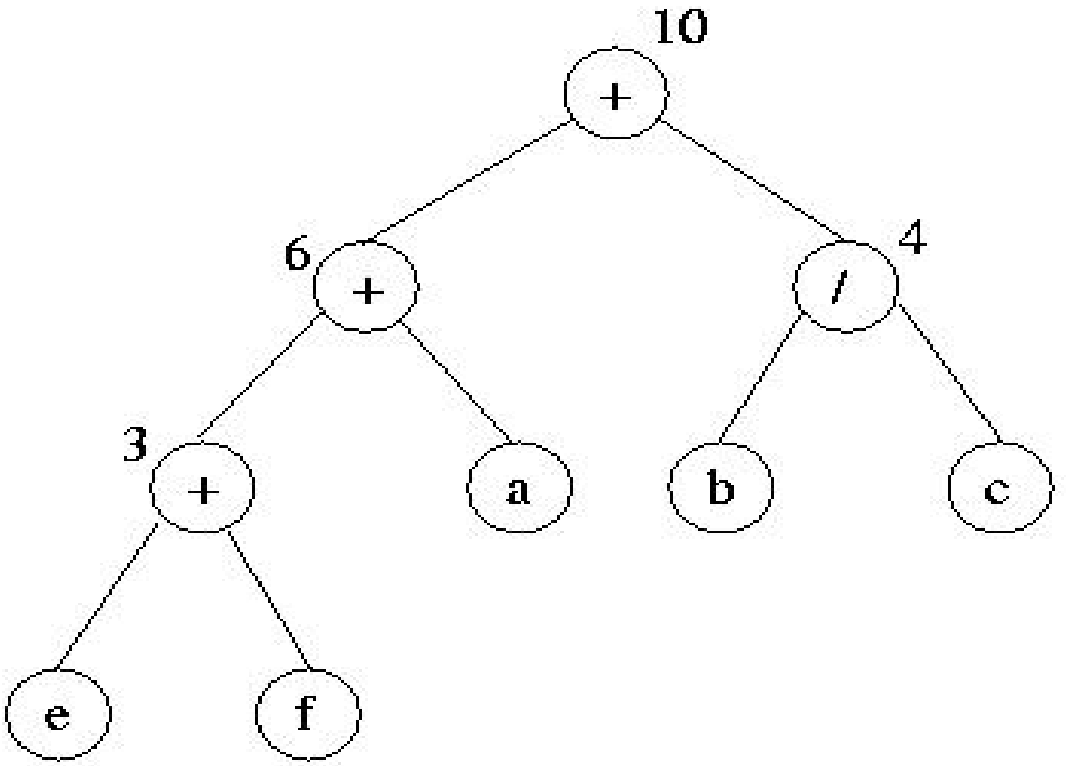
Algoritmo inorden, O(n)

```
void inorden(const tree<T> & A, const typename
  tree<T>::node &v)
{
  typename tree<T>::node aux;
   if (!v.null()) {
      aux = v.left();
      if (!aux.null())
         inorden(A, v.left());
      cout << *v;   //  acción sobre el nodo v.
      while (!aux.null()) {
         aux = aux.next_sibling())
         inorden(A,aux);
      }
   }
}
```

- Es de especial interés para árboles binarios, en particular, los ordenados.

Árboles de Expresión y Recorridos

- Tanto la expresión en prefijo como en postfijo permiten recuperar unívocamente el árbol de expresión.
- La expresión en infijo sólo lo permite si está parentizada.



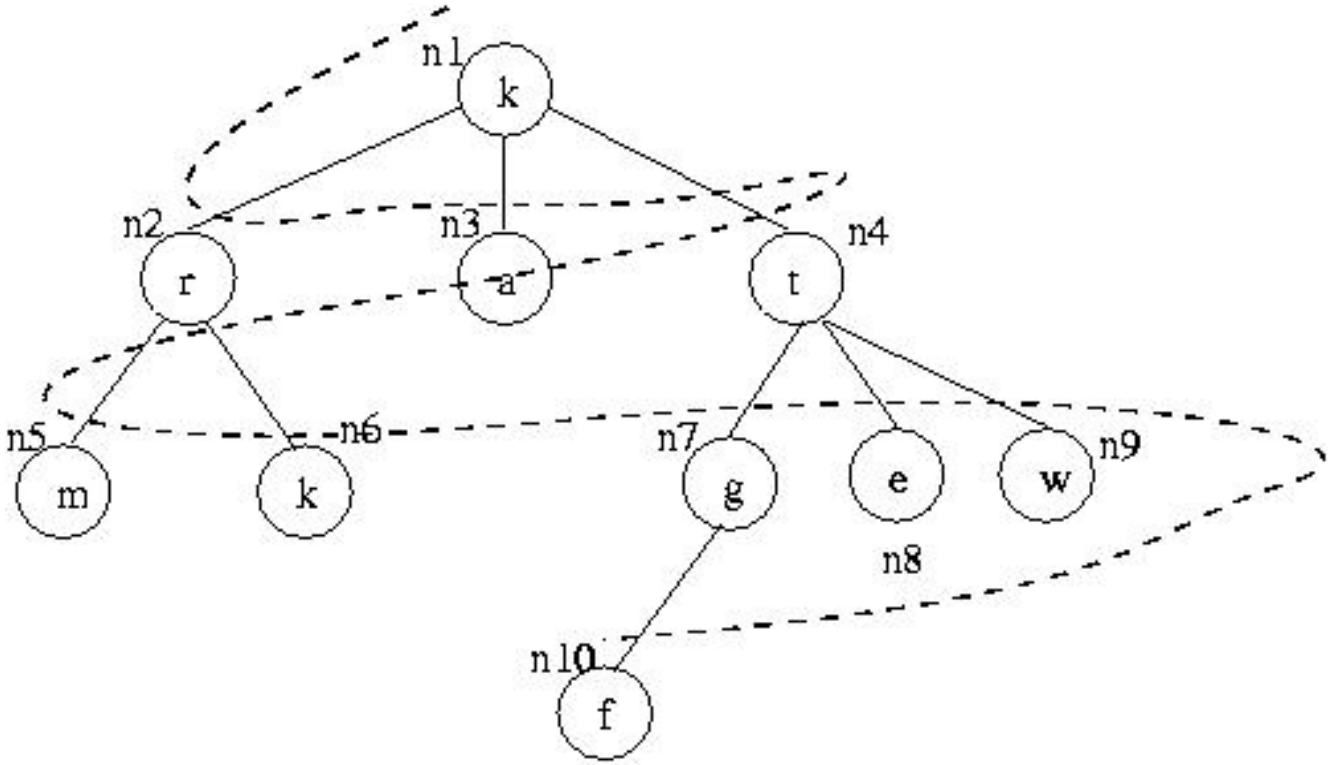
e= 1, f=2, a=3, b= 4, c=1

prefijo: +++efa/bc
infijo: e+f+a+b/c
postfijo:ef+a+bc/+

parentizado: +(+(+(ef)a)/(bc))
parentizado: ((e+f)+a)+(b/c)
parentizado:(((ef)+a)+(bc)/)+

Por Niveles o anchura

Visitar en orden los nodos de profundidad 0, después los de profundidad 1, profundidad 2, ...



n1 n2 n3 n4 n5 n6 n7 n8 n9 n10

Algoritmo Niveles, O(n)

```
void Niveles(const tree<T> & A, const typename
  tree<T>::node &v)
{
  typename tree<T>::node aux;
  queue<typename tree<T>::node> Cola;
  if (!v.null()) {
     Cola.push(v);
     while ( !Cola.empty() ) {
        aux = Cola.front();
        cout << *aux; // acción sobre el nodo v.
        for (aux=aux.left(); !aux.null();
          aux=aux.next_sibling())
           Cola.push(aux);
        Cola.pop();
     }
  }
}
```

Árboles Binarios

- Un árbol binario es un árbol ordenado en el que cada nodo interno tiene como máximo *dos hijos*.

  Sea A un árbol *propio* (todos los nodos internos tienen 2 hijos), n el número de nodos y h su altura, entonces
  - Un nivel (profundidad) d tiene como máximo 2^d nodos
  - h+1 <= número de nodos externos <= 2^h
  - h<= número de nodos internos <= (2^h) -1
  - 2h+1 <= n <= 2^(h+1) -1
  - log(n+1) -1 <= h <= (n-1)/2
  - El número de nodos externos es 1 más que el número de nodos internos

Árboles binarios

- Es necesaria una subclase interna *node* para poder movernos por el árbol.
- Qué debe tener todo nodo:
  - *constructor* por defecto: **`node();`**
  - *constructor* de copia:

    `node(const bintree<T>::node & n);`
  - determina si el receptor es el *nodo nulo*:
    `bool null() const;`
  - devuelve el nodo *padre* de un nodo receptor no nulo:
    `node parent() const;`
  - devuelve el nodo *hijo a la izquierda* de un receptor no nulo:
    `node left() const;`

Árboles binarios (2)

- Qué operaciones debe tener todo nodo:
  - devuelve el nodo *hijo a la derecha* de un receptor no nulo:
    `node right() const;`
  - devuelve la *etiqueta* de un nodo receptor no nulo:
    `T& operator*();`
  - *asigna* al receptor el contenido de un nodo **`a`**:
    `node& operator=(const node& a);`
  - determina si un nodo **`a`** es *igual* al receptor:
    `bool operator==(const node& a) const;`

Árboles binarios (3)

- Qué debe tener todo nodo:
  - determina si un nodo **a** es *distinto* al receptor:
    bool operator!=(const node& a) const;

- **Al igual que con iteradores sobre contenedores, el árbol binario debería estar dotado del tipo de dato const_node, que será el tipo de dato utilizado cuando se trabaje con const bintree.**

Árboles Binarios (5)

```
/**   TDA bintree.

Representa un árbol binario con nodos etiquetados
con datos del tipo T, que debe tener definidas las
operaciones:
 • T & operator=(const T & e);
 • bool operator!=(const T & e);
 • bool operator==(const T & e);
 Exporta tipos:
 node, preorder_iterator, postorder_iterator,
   inorder_iterator, level_iterator
 y sus versiones const
    Son mutables y residen en memoria dinámica.
 */
```

Árboles Binarios (6)

- Qué métodos debe tener:
  - el constructor por defecto: **`bintree();`**
  - el constructor con un nodo raíz de etiqueta **`e`**:
    `bintree(const T& e);`
  - el constructor de copia:
    `bintree(const bintree<T> &A);`
  - asignación al receptor de un subárbol de otro árbol **`A`**, subárbol que empieza en el nodo **`n`**:

```
assign_subtree(const bintree<T> &A,
   const bintree<T>::node &n);
```

Árboles Binarios (7)

- Qué métodos debe tener:
  - el *destructor*: **~bintree();**
  - el *operador de asignación*, que asigna a un receptor una copia del contenido de otro árbol **A**, destruyendo previamente el contenido del receptor:
    bintree<T>& operator= (const bintree<T>& A);
  - el *acceso al nodo raíz* de un receptor:
    node root () const;
  - el método que *poda* todo el subárbol *izquierdo* de un nodo dado (es decir, el subárbol que tiene al hijo izquierdo de un nodo dado como raíz), devolviendo dicho subárbol y eliminándolo del receptor:

```
void prune_left (bintree<T>::node n,
   bintree<T>& dest);
```

Árboles Binarios (8)

- Qué métodos debe tener:
  - el método que *poda* todo el subárbol *derecho* de un nodo dado (es decir, el subárbol que tiene al hijo derecho de un nodo dado como raíz), devolviendo dicho subárbol y eliminándolo del receptor:
    `void prune_right (typename bintree<T>::node n, bintree<T>& dest);`
  - el método que *cuelga un árbol* **`branch`** como subárbol *izquierdo/derecho* de un nodo **`n`** y anula el árbol **`branch`**:
    `void insert_left (typename bintree<T>::node n, bintree<T>& branch);`

    `void insert_left(typename bintree<T>::node n, const T & e);`
  - **`void insert_right (typenamebintree<T>::node n, bintree<T>& branch);`**

Árboles Binarios (9)

- Qué métodos debe tener:
  - *destruye* todos los *nodos* del receptor convirtiéndolo en el árbol nulo:
    `void clear ();`
  - calcula el *tamaño* del receptor en número de nodos:
    `size_type size () const;`
  - especifica si un árbol es el *árbol nulo*:
    `bool null () const;`
  - determina si el receptor es *igual* a otro árbol **`a`** que se le pasa:
    `bool operator== (const bintree<T>& a) const;`
  - determina si el receptor es *distinto* a otro árbol **`a`** que se le pasa:
    `bool operator!= (const bintree<T>& a) const;`

Ejemplos de Uso (I)

```
bool esHoja(const bintree<T> & A, const
  typename bintree<T>::node &v)
{
  return ( v.left().null()  &&
  v.right().null() );
}

bool esInterno(const bintree<T> & A,
  const typename bintree<T>::node &v)
{
  return ( !v.left().null() || !
  v.right().null() );
}
```

Ejemplos de Uso (II): Recorrido Preorden

```
void PreordenBinario(const bintree<T> & A,
typename bintree<T>::node v) {
  if (!v.null()) {
        cout << *v; // acción sobre el nodo v.
        PreordenBinario(A, v.left());
        PreordenBinario(A, v.right());
  }
}
```

Ejemplos de Uso (III): Recorrido Inorden

```
void InordenBinario(const bintree<T> & A,
 typename bintree<T>::node v)
{
  if (!v.null()) {
       InordenBinario(A, v.left());
       cout << *v; //acción sobre el nodo v.
       InordenBinario(A, v.right());
  }
}
```

Ejemplos de Uso (IV): Recorrido Postorden

```
void PostordenBinario(const bintree<T> & A,
 typename bintree<T>::node v)
{
  if (!v.null()) {
       PostordenBinario(A, v.left());
       PostordenBinario(A, v.right());
       cout << *v; // acción sobre el nodo v.
  }
}
```

Ejemplos de Uso (V): Recorrido por Niveles

```
void ListarPorNiveles(
                                const bintree<T> &A,
                                typename
  bintree<T>::node n) {
  queue<bintree<T>::node> nodos;
  if (!n.null()) {
    nodos.push(n);
    while (!nodos.empty()) {
      n = nodos.front(); nodos.pop();
      cout << *n;
      if (!n.left().null()) nodos.push(n.left());
      if (!n.right().null())
  nodos.push(n.right());
    }
  }
}
```

Ejemplos de Uso (VI)

```
#include <iostream>
#include "bintree.h"
using namespace std;
int main()
{  // Creamos el árbol:
  //          7
  //        /    \
  //      1       9
  //    /   \    /
  //  6      8  5
  //           \
  //           4
```

Construimos el árbol...

```
typedef bintree<int> bti;

bintree<int> Arb(7);

Arb.insert_left(Arb.root(), bti(1));

Arb.insert_right(Arb.root(), bti(9));

Arb.insert_left(Arb.root().left(), bti(6));

Arb.insert_right(Arb.root().left(), bti(8));

Arb.insert_right(Arb.root().left().right(),
  bti(4));

Arb.insert_left(Arb.root().right(), bti(5));
```

Iteradores para árboles

De acuerdo a los cuatro recorridos definidos sobre árboles binarios, se precisan cuatro iteradores:

- `preorder_iterator`: itera sobre los nodos recorridos en preorden,
- `postorder_iterator`: itera sobre los nodos recorridos en postorden,
- `inorder_iterator`: itera sobre los nodos recorridos en inorden,
- `level_iterator`: itera sobre los nodos recorridos por niveles.

Recorrido en PreOrden

```
typename bintree<T>::preorder_iterador
  it_p;
for (it_p= Arb.begin_preorder();
        it_p!=Arb.end_preorder(); ++it_p)
       cout << *it_p << " ";
   cout << endl;
```

Recorrido en PostOrden

```
 for (typename
bintree<T>::postorder_iterator i =
Arb.begin_postorder(); i !=
Arb.end_postorder(); i++ )

   cout << *i << " ";

 cout << endl;
```

Recorrido en InOrden

```
 for (typename
bintree<T>::inorder_iterator i =
Arb.begin_inorder();i !=
Arb.end_inorder(); i++)
   cout << *i << " ";
 cout << endl;
```

Recorrido por Niveles

```
 for (typename
bintree<T>::level_iterator i =
    Arb.begin_level(); i !=
     Arb.end_level(); i++)
   cout << *i << " ";
 cout << endl;
```

Código Esqueleto General

```
template <typename T> class bintree {
public:
  bintree();   bintree(const T & e);
  ........
  class preorder_iterator { ...  };
  class inorder_iterator { ... };
  class postorder_iterator { ... };
  class node { ... };
private:
....
};
#include “bintree.template”
```

Bintree representacion:

```
private:
  // Funciones auxiliares ......
  // Representación bintree
    node laraiz;
    size_type num_nodos;
  /**    TDA nodowraper.   Modela los nodos del Arbol binario.   */
  class nodewrapper {
  Public
    T etiqueta;
    nodewrapper *pad, *izda, *dcha;
  };


};
```