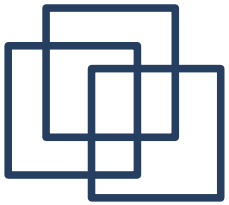
TDAs Contenedores Complejos



Objetivos

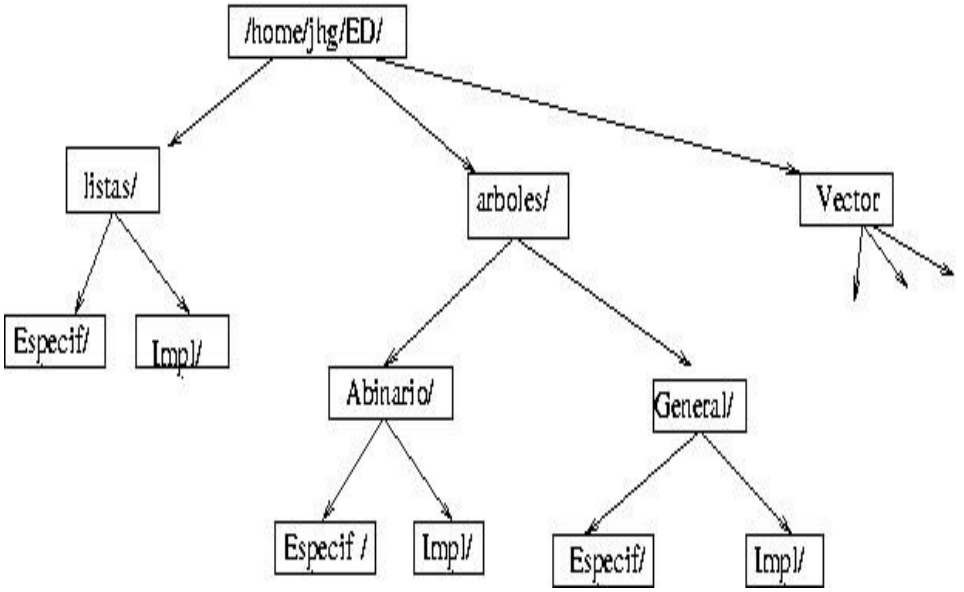
- Comprender el concepto de *árbol*
- Conocer la terminología y las formas de recorrer el contenido de un árbol
- Manejar los TDAs *tree* y *bintree*
- Manejar los BST, AVL, POT
- Introducir el concepto de *grafo* y su especificación.

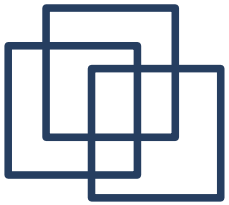
Árboles

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).

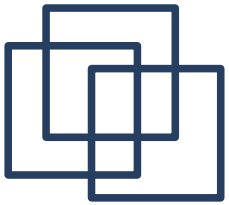




Árbol: Definición recursiva

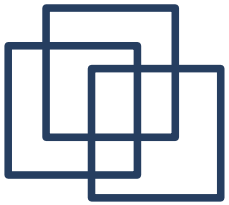
Un árbol T es un conjunto de nodos que almacenan elementos mediante una relación padre-hijo con la siguiente propiedad:

- Un nodo es un árbol.
- Un árbol vacío (con cero nodos) también es un árbol.
- Si tomamos N árboles y cada una de sus raíces las hacemos hijo de un nodo, el resultado también es un árbol.



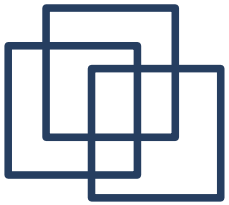
Terminología I

- Existe una relación entre nodos en la que el nodo que está por encima se llama *padre* y el de abajo *hijo*.
- Nodo *raíz* es el único nodo que no tiene padre.
- Nodo *hoja* es aquel que no tiene hijos.
- 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.



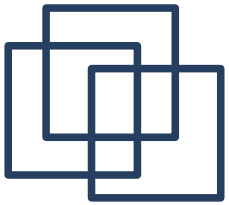
Terminología II

- 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.
- 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.



Terminología III

- 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

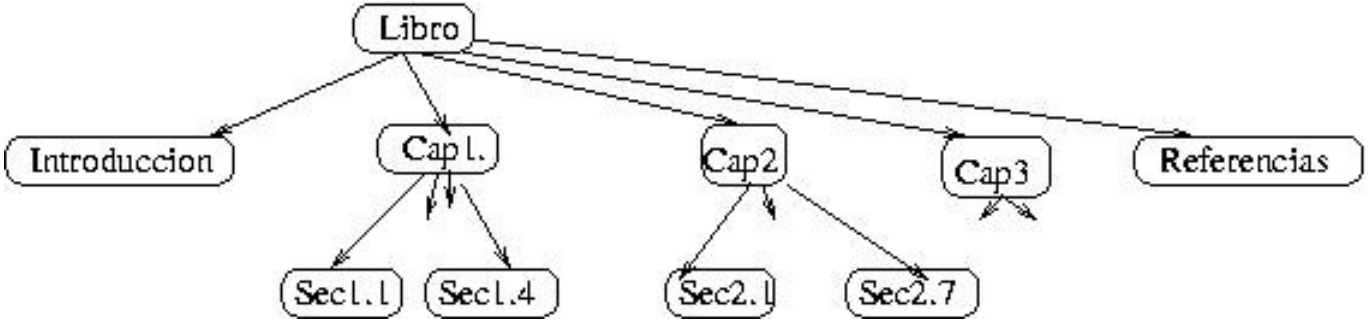


Terminología IV

- Un *subárbol parcial* es un subárbol del que no tenemos todos sus descendientes.
- En los árboles generales no hay un número máximo de hijos por nodo.
  - Si el número máximo de hijos por nodo es 2 diremos que el árbol es **binario**,
  - si es tres, ternario,
  - ...
  - y en general n-ario.
- Un árbol se dice ordenado si se establece un orden lineal entre los hijos de cada nodo (1º, 2º, 3º, ...)

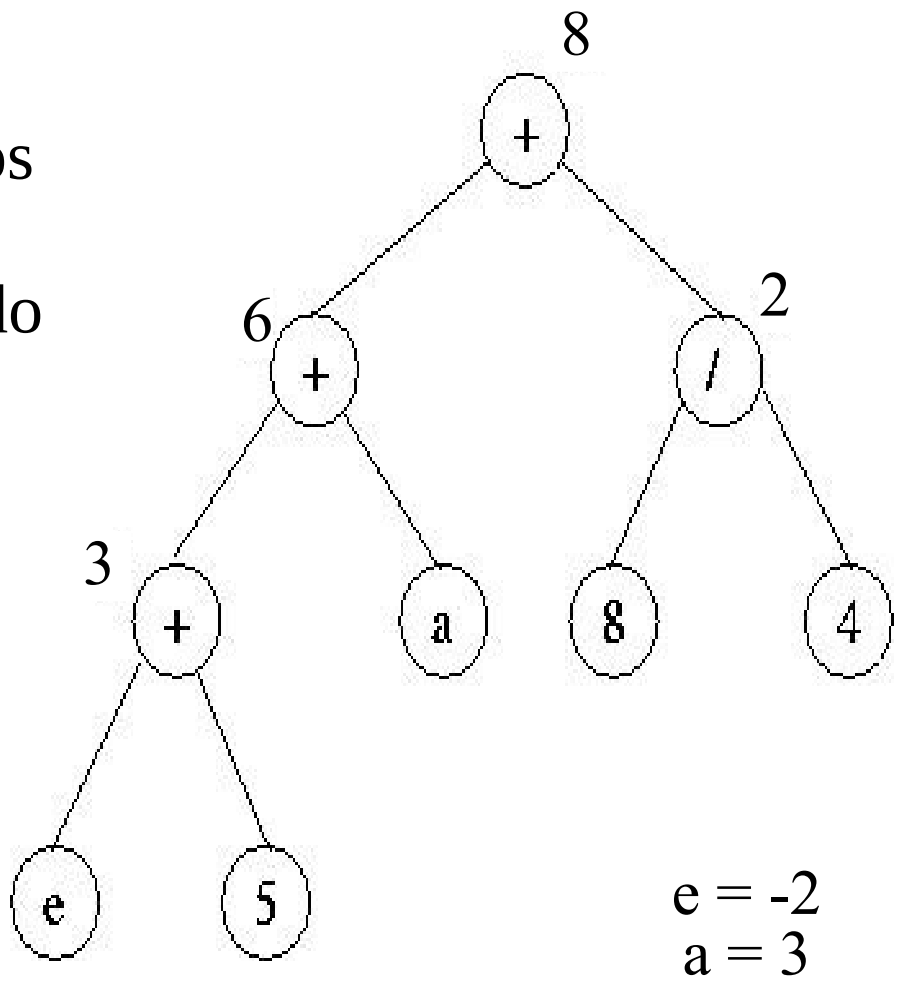
Ejemplo 1

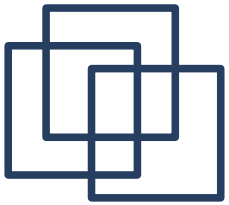
- Un documento estructurado (libro) se organiza jerárquicamente como un árbol ordenado cuyos nodos internos son los capítulos, secciones y subsecciones y cuyos nodos externos son los párrafos.



Ejemplo 2

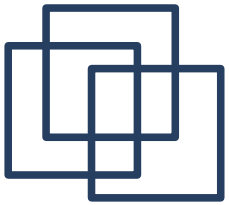
- Una expresión aritmética se puede representar como un árbol cuyos nodos externos son variables o constantes y cuyos nodos internos son operadores (+,*,/,-).
- Cada nodo de árbol tiene asociado un valor:
  - Si es externo, el que indica la variable o constante
  - Si es interno, el valor se determina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





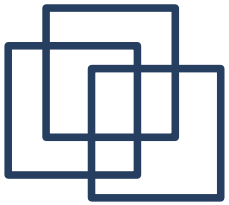
Otros Ejemplos

- Jerarquía de clases (directorio de yahoo)
- Árbol de sintaxis (estructura de una oración)
- Árbol genealógico
- Jerarquía Territorial (País, Comunidad Autónoma, Provincia, Municipio, ...)
- Estructura de una competición por eliminatorias.
- .....



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.



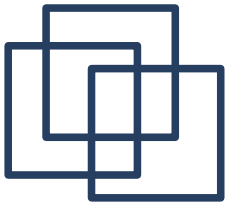
Tipo de Dato Árbol (2)

- Tenemos TDA tree y TDA node
  - 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;

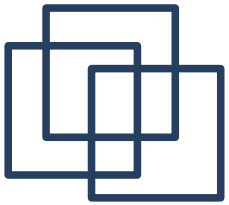
    Verdadero si el nodo es externo, false en caso contrario
  - size_type size() const ; devuelve en número de nodos de un árbol.
  - .....



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;

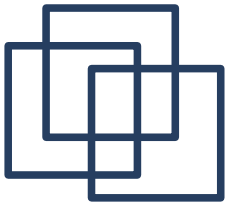
    devuelve si el nodo es nulo (no confundir con que valga NULL!!)



Moviéndonos por un árbol

Al no ser lineal, existen varias formas. Por ejemplo:

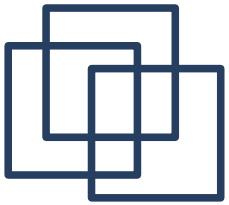
```
int profundidad(const tree<T> & A, const
  typename tree<T>::node &v)
{
  int prof = 0;
  typename tree<T>::node aux=v;
  while (!A.is_root(aux)) {
    prof++;
    aux = aux.parent();
  }
  return prof;
}
```



Moviéndonos por un árbol (2)

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

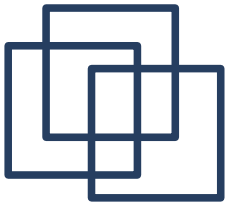
  Definición recursiva:
  - Si v es externo, la altura es 0.
  - En otro caso, uno más la máxima altura de los hijos de v.



Moviéndonos por un árbol (3)

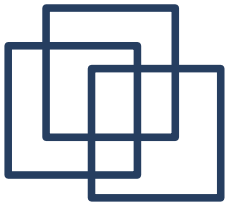
- Implementación (Orden O(n))

```
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 I

- Mecanismo por el cual podemos visitar (acceder) todos los nodos de un árbol.
- No existe un único criterio (orden):.
  - *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

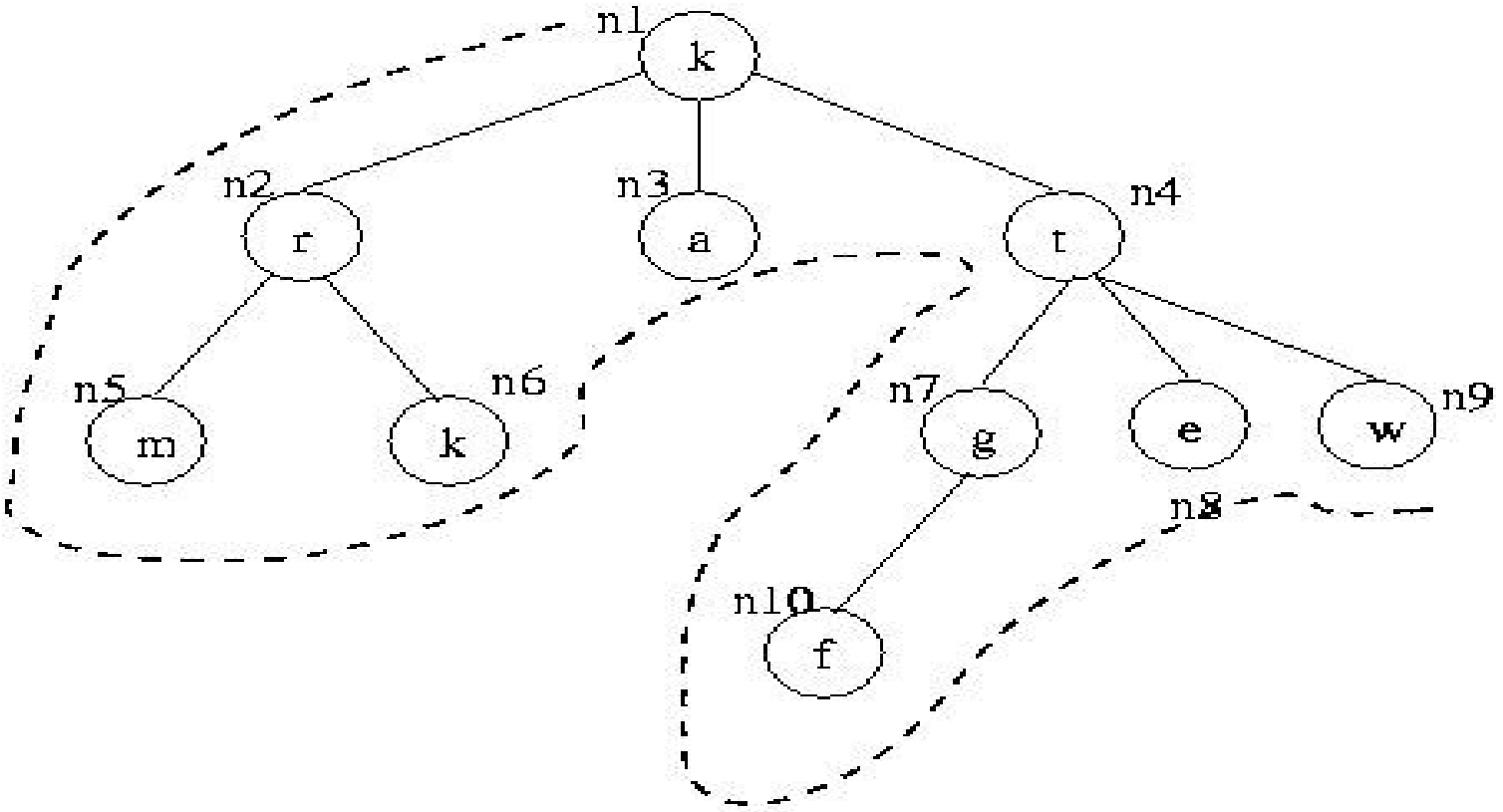


Recorridos II

- *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, ...

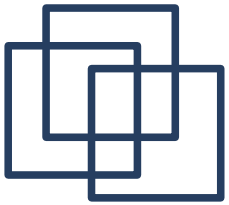
Preorden

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)



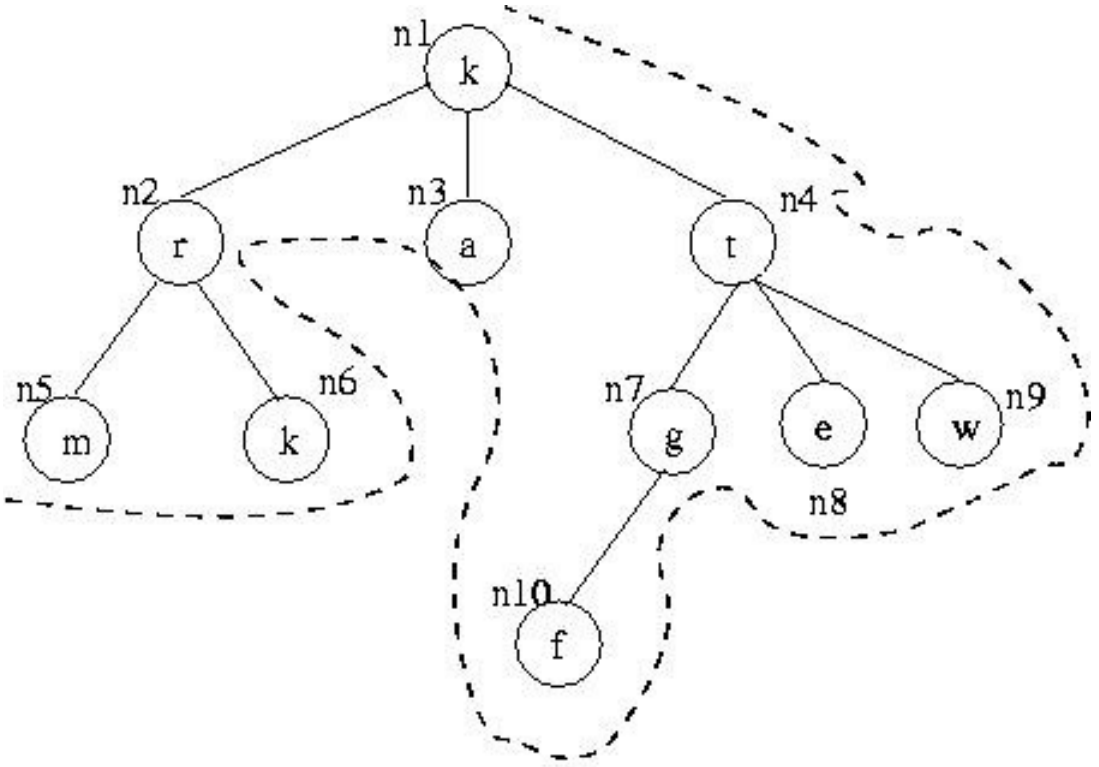
Algoritmo preorden, O(n)

```
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);
  }
}
```

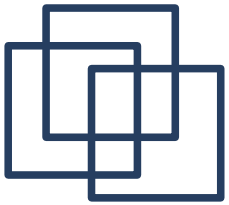
- Produce un orden lineal de los nodos donde un nodo aparece antes que los hijos.
- Si A es un documento estructurado, entonces preorden(A, A.root()) examina el documento secuencialmente, del principio al final.

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



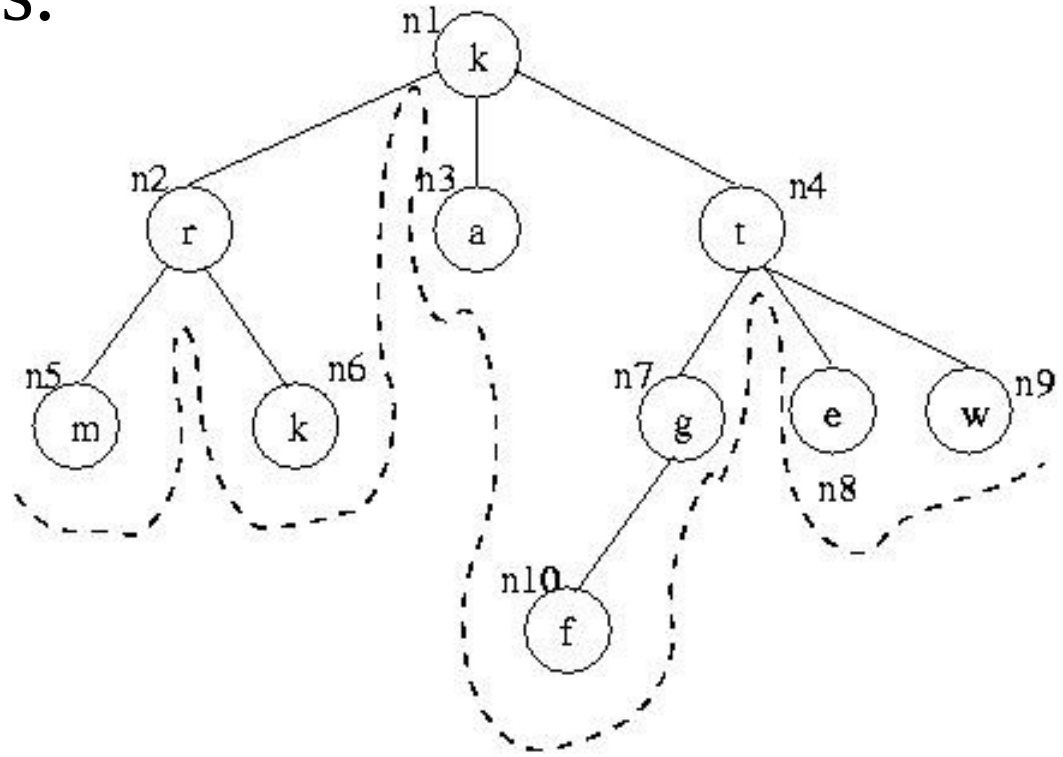
Algoritmo postorden, O(n)

```
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.

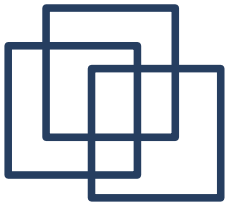
Inorden

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))



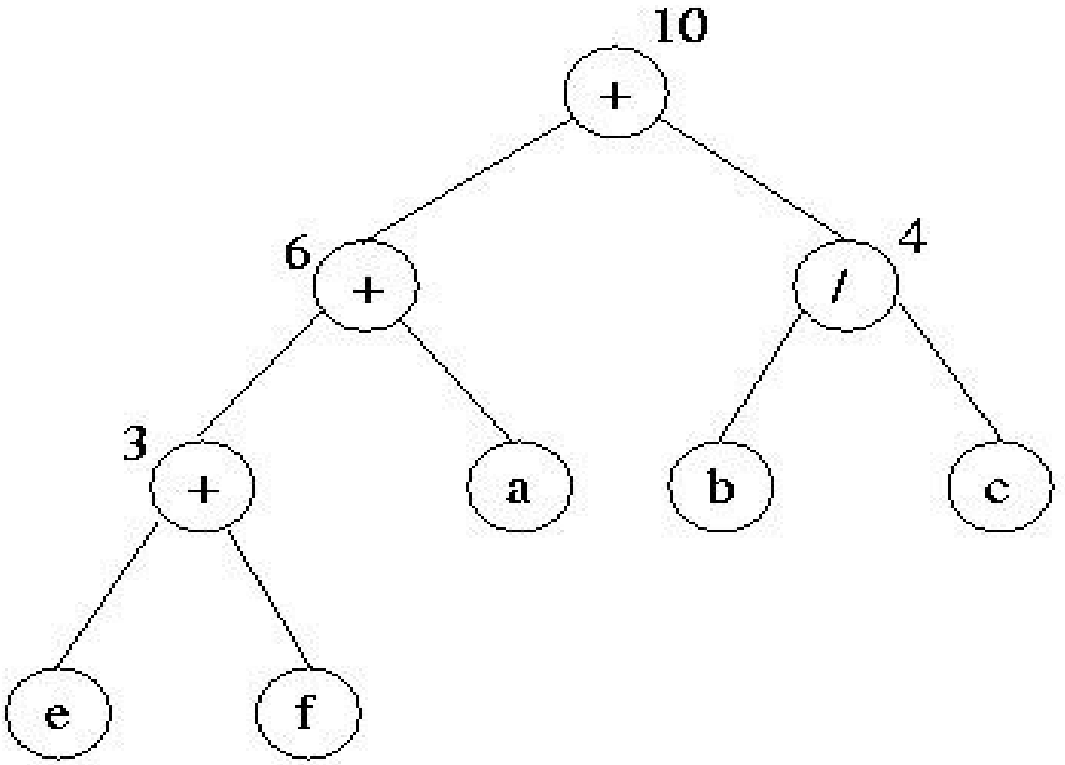
Algoritmo inorden, O(n)

```
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.

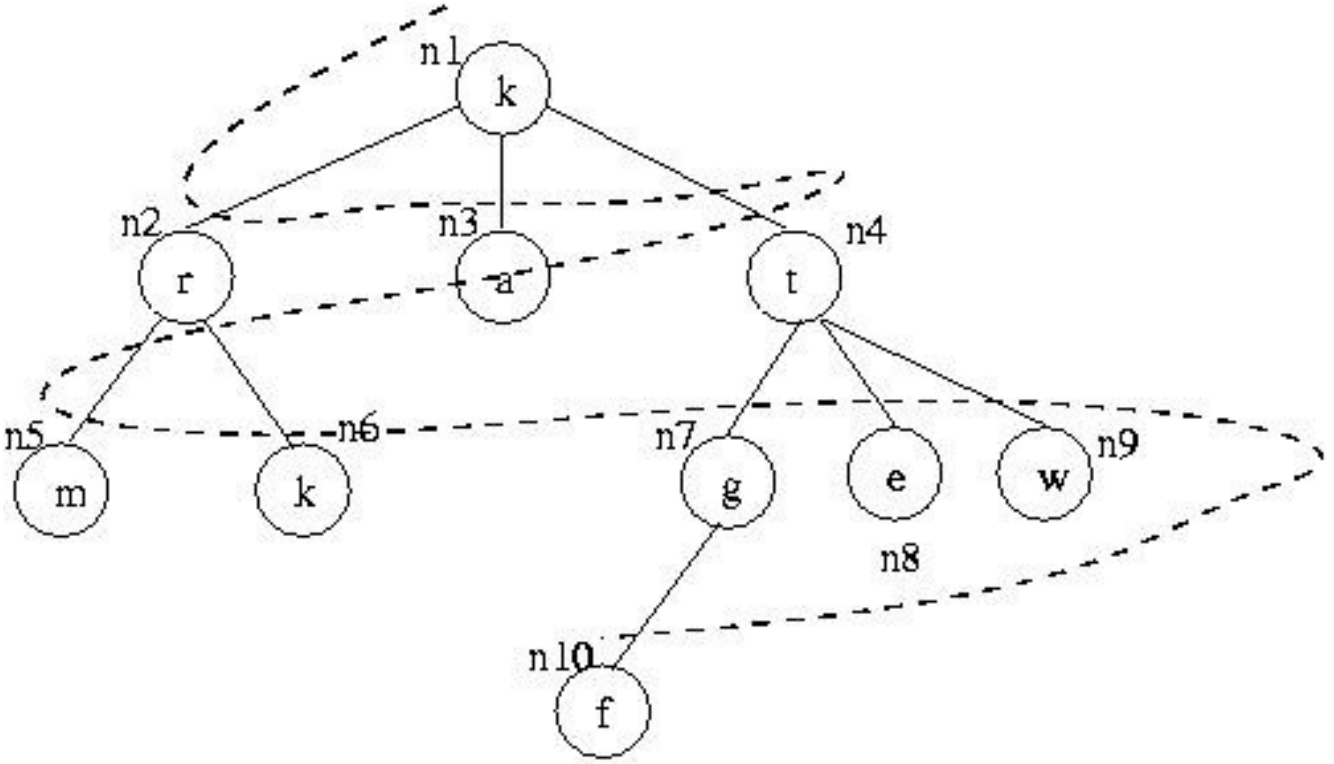


e= 1, f=2, a=3, b= 4, c=1

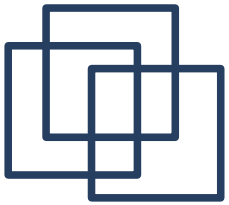
prefijo: +++efa/bc	parentizado: +(+(+(ef)a)/(bc))
infijo: e+f+a+b/c	parentizado: ((e+f)+a)+(b/c)
postfijo:ef+a+bc/+	parentizado:(((ef)+a)+(bc)/)+

Recorrido por Niveles (anchura)

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

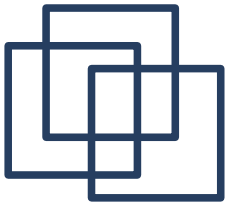


n1 n2 n3 n4 n5 n6 n7 n8 n9 n10



Recorrido por 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();
    }
 }
}
```

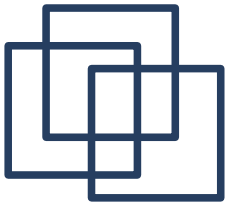


Árboles binarios

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

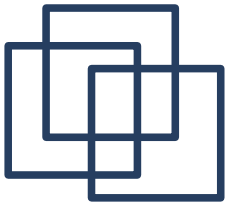
  Sea A un árbol *propio* (todos los nodos internos tienen 2 hijos), n el número de nodos y h su altura, entonces:

  - 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



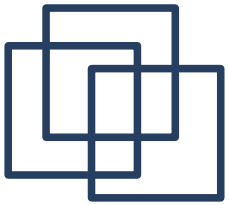
TDA bintree (I)

- Es necesaria una subclase interna *node* para poder movernos por el árbol.
- TDA bintree<T>::node :
  - *constructor* por defecto: **`node();`**
  - *constructor* que crea un nodo con etiqueta **`e`**: **`node(const T& e);`**
  - 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;`



TDA bintree (II)

- devuelve el nodo *hijo a la derecha* de un receptor no nulo:
  `node right() const;`
- devuelve la *etiqueta* de un nodo receptor no nulo:
  `const T& operator*() const; T& operator*();`
- *elimina* el contenido de un *nodo* receptor no nulo, convirtiéndolo en el nodo nulo(sólo si el nodo es una hoja):
  `void erase();`
- *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;`
- determina si un nodo **`a`** es *distinto* al receptor:
  `bool operator!=(const node& a) const;`

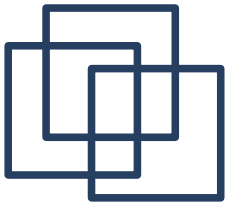


TDA bintree (III)

```
/**   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);`

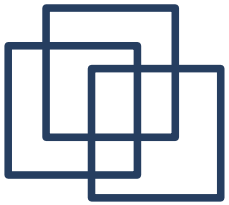
```
Exporta tipos:
node, preorder_iterator, postorder_iterator,
  inorder_iterator, level_iterator
y sus versiones const
   Son mutables y residen en memoria dinámica.
*/
```



TDA bintree (IV)

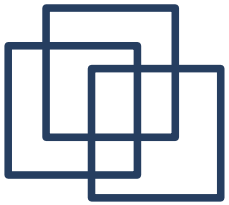
- el constructor por defecto: **`bintree();`**
- el constructor con un nodo raíz de etiqueta **`e`**: **`bintree(const T &e);`**
- el constructor de copia:
  `bintree(const bintree<T> &A);`
- asignación al receptor de un subárbol de otro árbol **`A`**, subárbol que empieza en el nodo **`n`**:

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



TDA bintree (V)

- 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);
  ```



TDA bintree (VI)

- 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:

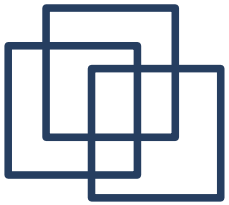
```
void prune_right (typename bintree<T>::node n,
   bintree<T>& dest);
```

- el método que *cuelga un árbol* **branch** como subárbol *izquierdo* de un nodo **n** y anula el árbol **branch**:

```
void insert_left (typename bintree<T>::node n,
   bintree<T>& branch);
```

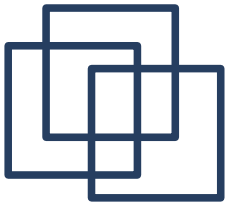
- el método que *cuelga un árbol* **branch** como subárbol *derecho* de un nodo **n** y anula el árbol **branch**:

```
void insert_right (typenamebintree<T>::node n,
   bintree<T>& branch);
```



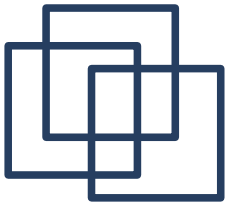
TDA bintree (VII)

- *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;`



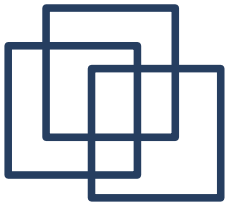
Ejemplos de Uso (I)

```
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() );
}
```



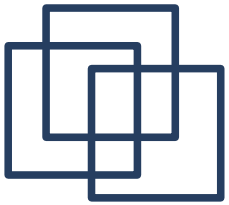
Ejemplos de Uso (II): Recorrido en Preorden

```
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());
  }
}
```



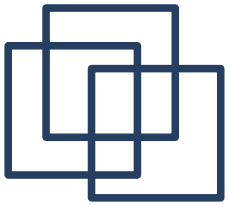
Ejemplos de Uso (III): Recorrido en Inorden

```
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());
  }
}
```



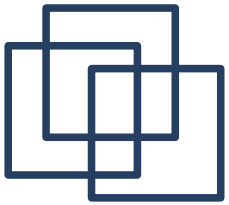
Ejemplos de Uso (IV): Recorrido en Postorden

```
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.
  }
}
```



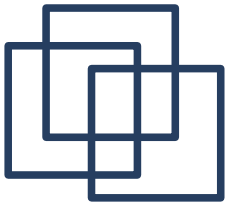
Ejemplos de Uso (V): Recorrido por Niveles

```
void ListarPostNiveles(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());
    }
  }
}
```



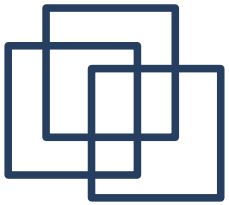
Ejemplos de Uso (VI)

```
#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));
```

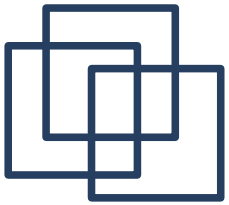


Iteradores para árboles

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

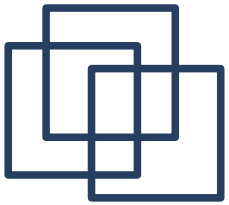
- `preorder_iterator`: itera sobre los nodos recorridos en preorden,
- `postorder_iterator`: itera sobre los nodos recorridos en postorden,
- `inorder_iterator`: itera sobre los nodos recorridos en inorden,
- `level_iterator`: itera sobre los nodos recorridos por niveles.

  ATENCIÓN: su uso es muy ineficiente.



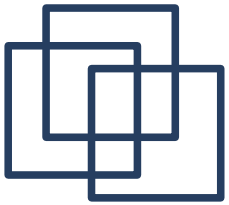
Recorrido en PreOrden

```
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;
```



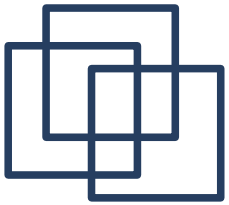
Recorrido en PostOrden

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

  cout << *i << " ";

cout << endl;
```



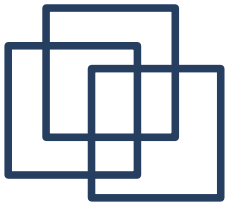
Recorrido en InOrden

```
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;
```

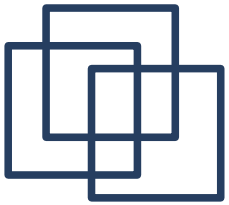


Tipos de Árboles Binarios

Árboles Binarios de Búsqueda (BST)

Árboles Equilibrados (AVL)

Árboles Parcialmente Ordenados (POT)

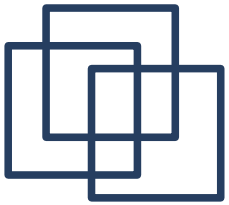


Árboles binarios de búsqueda (BST)

La *búsqueda binaria* es un proceso rápido de búsqueda de elementos en un vector ordenado O(log(n)).

Sin embargo, las *inserciones y borrados* son muy ineficientes (O(n)).

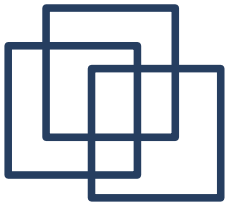
Los BST son de utilidad para representar TDAs donde las operaciones de búsqueda, inserción y borrado sean eficientes, además de permitir acceder a los elementos de forma ordenada.



BST: Definición

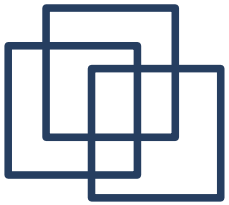
Árbol Binario de Búsqueda (BST): árbol binario que verifica que:

- todos los elementos almacenados en el subárbol izquierdo de un nodo *n* son menores que el elemento almacenado en *n*,
- todos los elementos del subárbol derecho de *n* son mayores que el elemento almacenado en *n*.



BST: Propiedades (I)

- Son árboles binarios en los que para cada nodo se cumple que:
  - Tanto su hijo a la izquierda como los descendientes de éste tienen una etiqueta menor que la del nodo.
  - Tanto su hijo a la derecha como los descendientes de éste tienen una etiqueta mayor que la del nodo
- ATENCIÓN: ¡no basta con que el hijo de la izquierda sea menor y el de la derecha mayor!

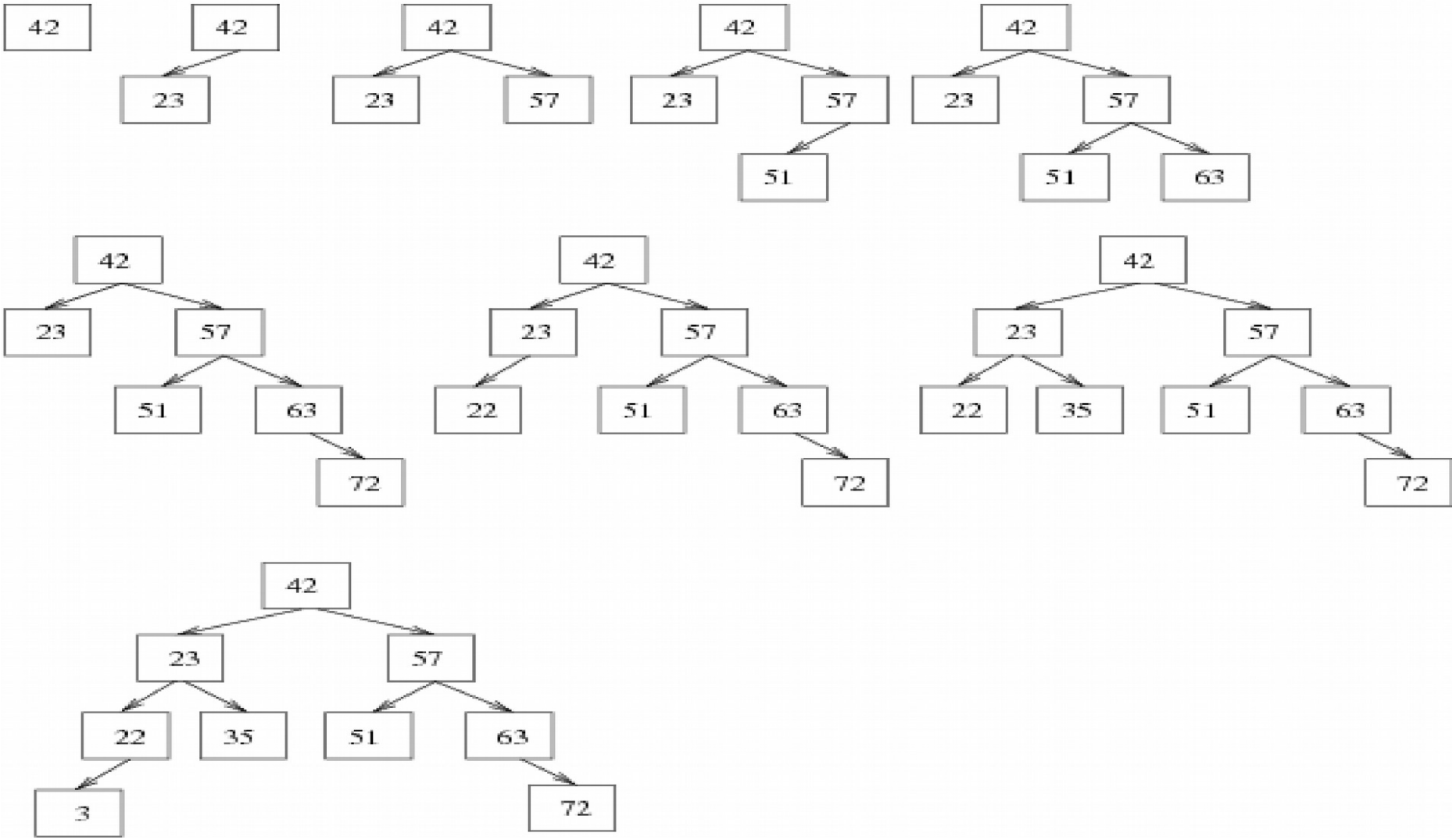


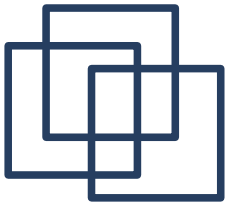
BST: Propiedades (II)

- Es un tipo de dato orientado a búsqueda, es decir, que tan sólo tiene operaciones para crear, destruir, insertar etiquetas, borrar etiquetas y buscar etiquetas (aparte de un iterador)
- Si el árbol está equilibrado, las operaciones de inserción, búsqueda y borrado de un elemento se pueden realizar en orden O(log(n))
- El recorrido en inorden da el conjunto de etiquetas ordenado

BST: Ejemplo

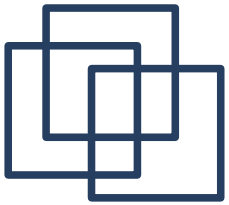
Claves: 42,23,57,51,63,72,22,35,3





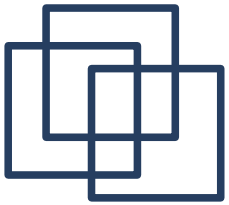
BST: Especificación (I)

```
/**  TDA BST::BST, insert, find, erase, size,
  clear, empty, begin, end, ~BST

 El TDA BST representa objetos que abstraen el
  concepto de Árbol Binario de Búsqueda. Es un tipo
  contenedor, cuyos componentes son Entradas
  formadas por dos partes {clave, valor}, de tipos
  Key y T, resp.

 Los objetos Key deben tener las operaciones:

 - bool operator<(const Key & c);

 - bool operator==(const Key & c);

 Los objetos T deben tener las operaciones:

 - bool operator==(const T & c);
   bool operator<(const T & c);

*/
```



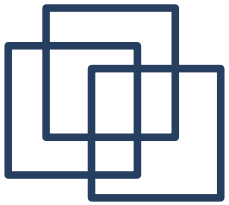
BST: Especificación (II)

- Qué métodos debe tener:
  - el constructor por defecto: **`BST();`**
  - modificador que inserta o actualiza una entrada **`entrada`** de clave especificada dentro del BST:
    `void insert(const pair<Key, T> & entrada);`
  - método que busca un valor de clave **`clave`** específico dentro del BST y devuelve un iterador que lo contiene (si no está, devuelve **`end()`**):
    `iterator find(const Key & clave);`
  - método que devuelve el número de entradas del BST:
    `size_type size() const;`
  - modificador que elimina todas las entradas del receptor:
    `void clear();`



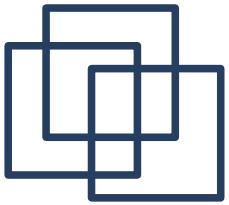
BST: Especificación (III)

- Qué métodos debe tener:
  - método que indica si el receptor está vacío:
    `bool empty() const;`
  - método que devuelve la posición dela primera entrada del BST:
    `iterator begin() const;`
  - método que devuelve la posición siguiente a la última entrada del BST receptor:
    `iterator end() const;`
  - destructor que libera todos los recursos asociados al BST receptor:
    `~BST();`



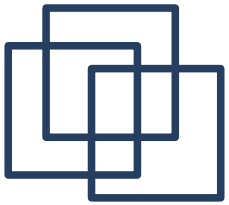
BST::iterator especificación

- Qué métodos debe tener:
  - Método que devuelve el par (clave, dato) que contiene la posición del iterador (que no puede estar en la posición **end**):
    `pair<Key, T> operator*() const;`
  - método que avanza el iterador a la siguiente posición del BST:
    `void operator++ ();`



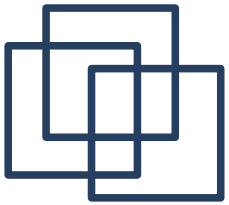
Ejemplo Uso BST

```
BST<string, string> arb;
string palabras[NUM] = {"piedra", "tiza",... };
string definiciones[NUM] = {"objeto muy duro",
    "objeto muy blando", ...};
for (int i= 0; i<NUM; i++)
  arb.insert(pair<string,string>(palabras[i],
    definiciones[i]));
cout << "Num.datos: " << arb.size();

BST<string, string>::iterator i;
for (i = arb.begin(); i != arb.end(); ++i)
   cout << (*i).first << ": " << (*i).second << endl;
```



Ejemplo Uso BST: Ordenación

```
template <class T>
void ordenar(T v[], int num_datos)
{
  int i;
  BST<T,char> abb;

  for (i = 0; i < num_datos; i++)
    abb.insert(pair<T,char>(v[i], ''));

  BST<T,char>::iterator ite;
  for (ite = abb.begin(), i = 0;
     ite != abb.end(); ++ite, i++)
        v[i] = (*ite).first;
}
```



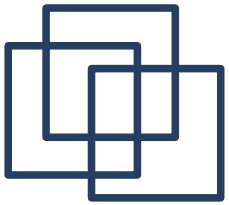
Arbol Equilibrados: AVL

Motivación:

- Las operaciones de Búsqueda, Inserción y Borrado en un BST genérico son de orden O(altura(arbol)) !!!

Arboles Equilibrados

Altura (arbol) es de O(log(n))



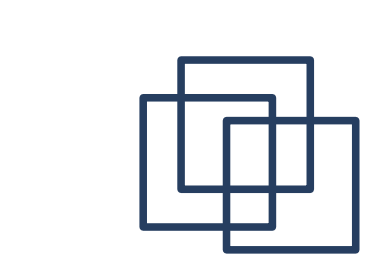
Arboles AVL

BST Equilibrado donde las operaciones de búsqueda e inserción y borrado tienen un orden de eficiencia logarítmico.

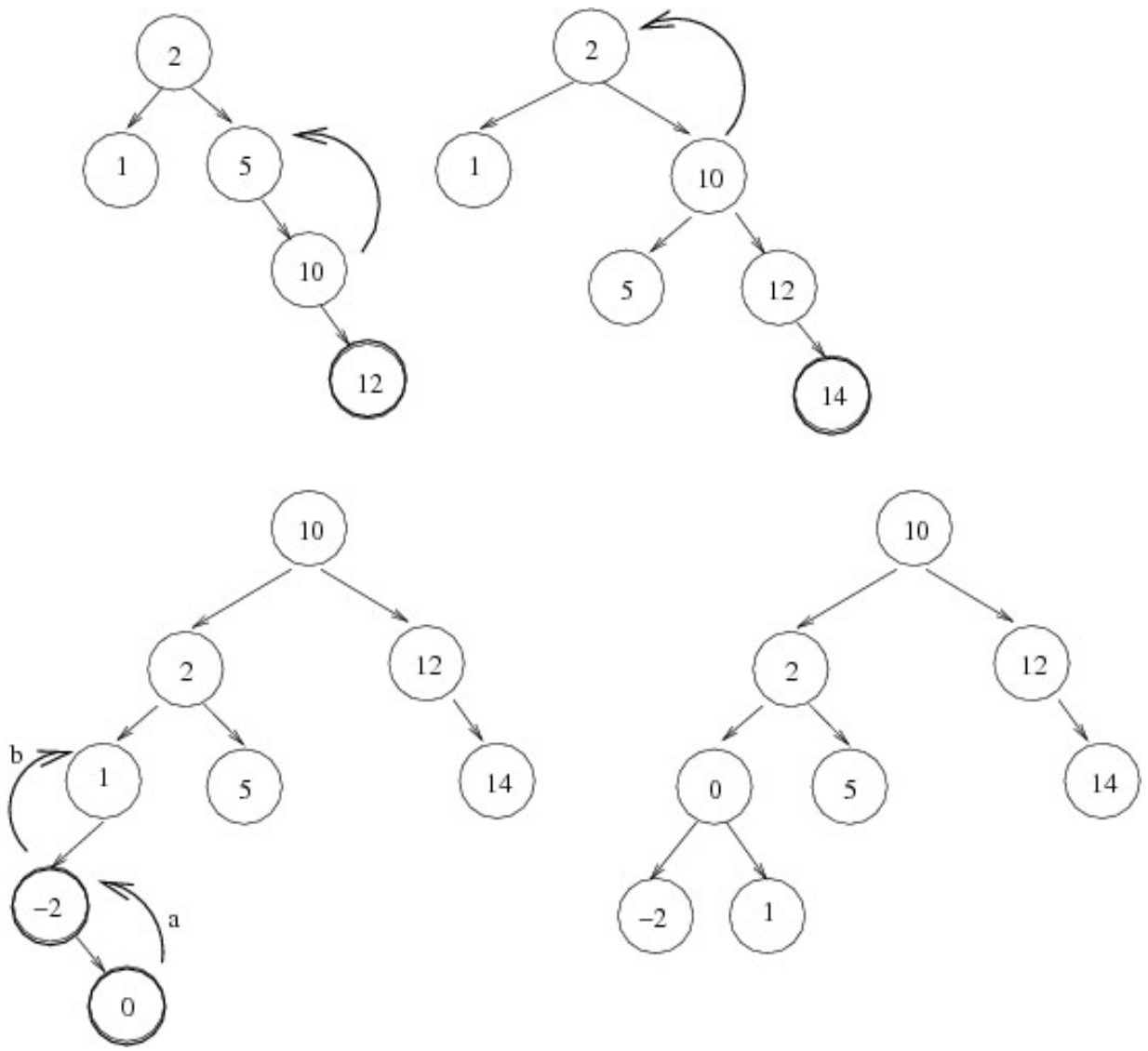
para cada nodo se cumple que:

la diferencia de la altura de sus dos hijos es menor que uno (en valor absoluto).

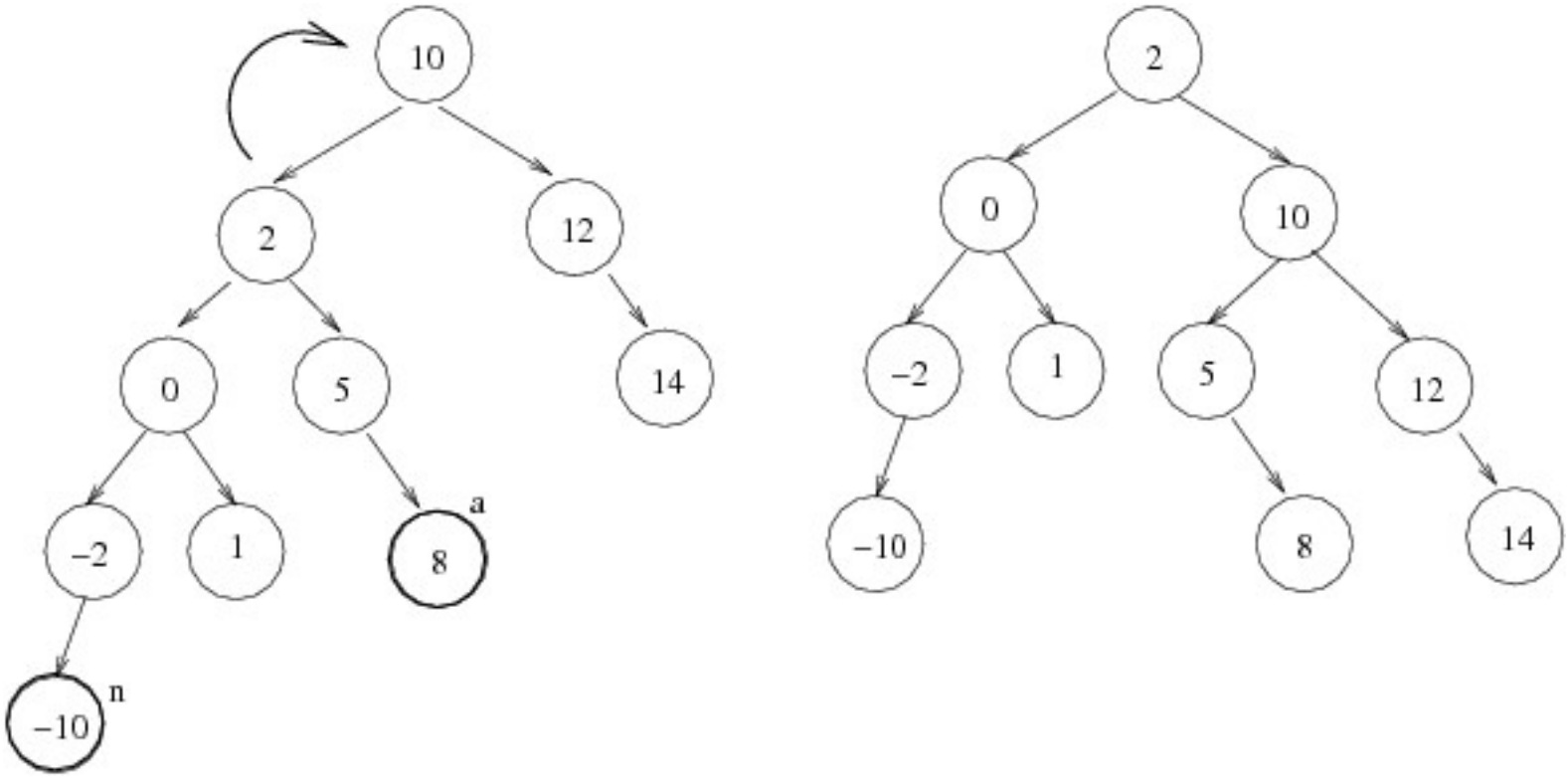
- La especificación coincide con la del BST
- La implementación varía en las operaciones que modifican la altura de un nodo: insertar y borrar.

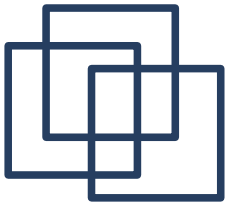


Inserción en AVL (I)



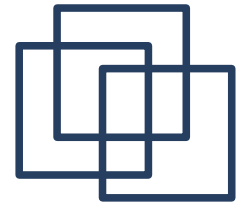
Inserción en AVL (II)



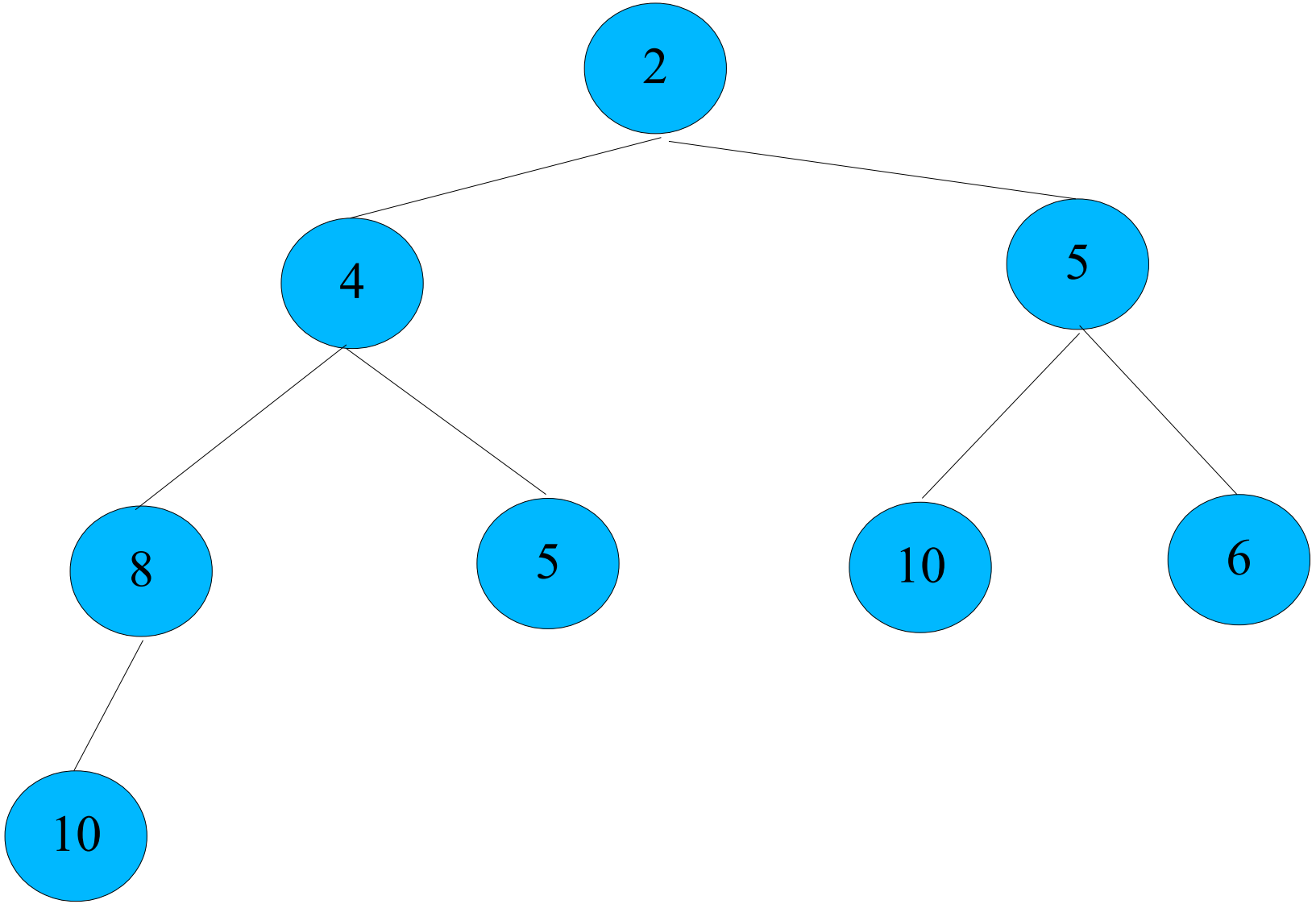


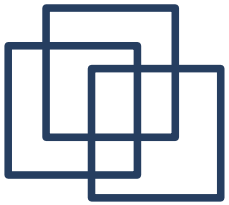
Árbol parcialmente ordenado

- Son árboles equilibrados que permiten obtener el mínimo de un conjunto de datos de forma eficiente O(log_2 n)
- Cuando se añade un elemento siempre se hace lo más a la izquierda posible en el nivel que no esté completo, y si el nivel está completo como el elemento más a la izquierda en el siguiente nivel.
- Son las estructuras utilizadas en las colas con prioridad



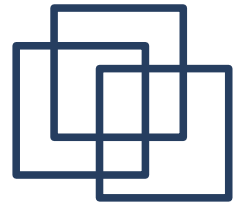
Ejemplo POT



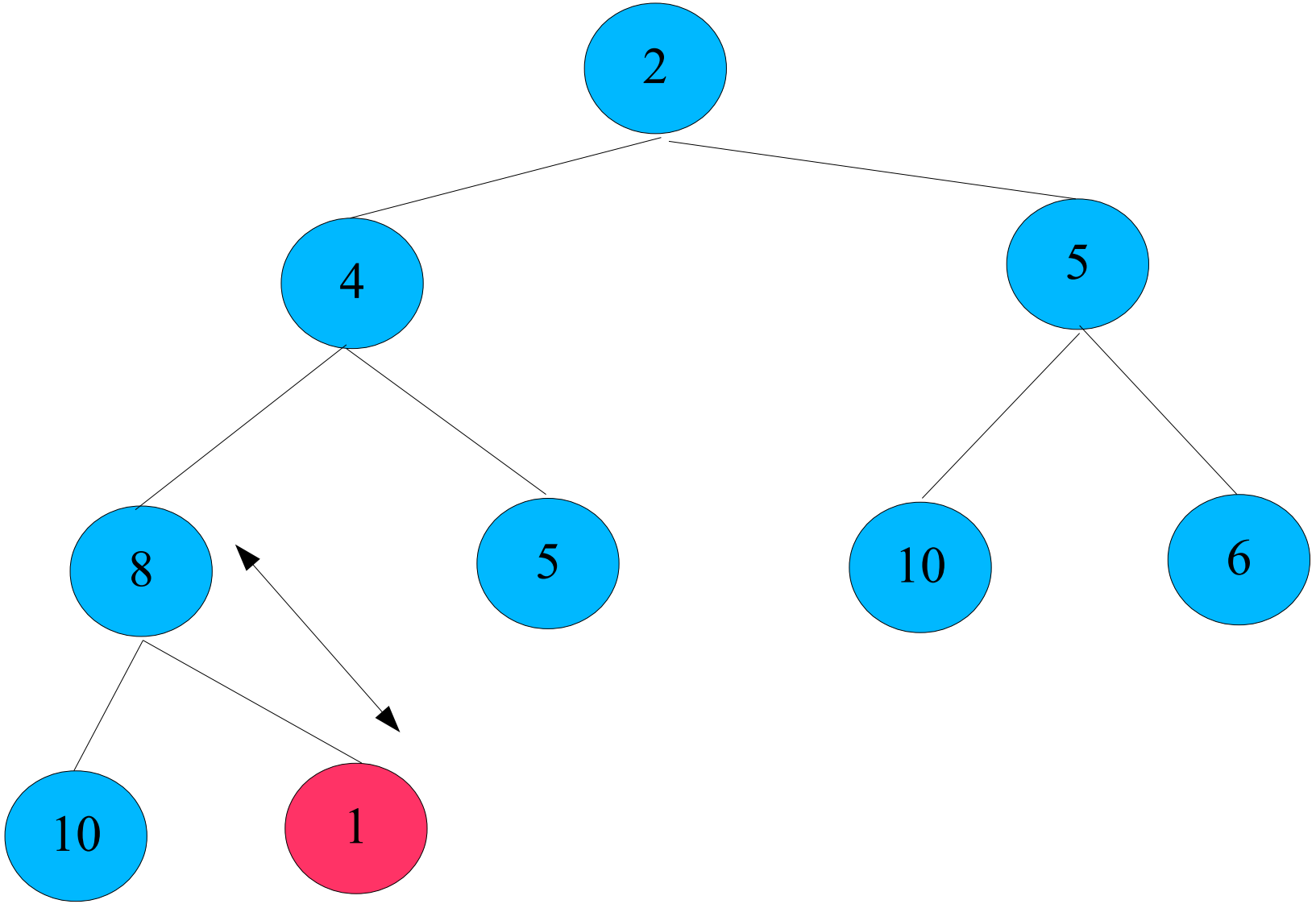


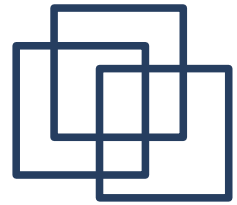
POT: Especificación

- Qué métodos debe tener:
  - el constructor por defecto: **`POT(int tam);`**
  - el constructor de copia:
    `POT (const POT<T> &copiado);`
  - método que devuelve el número de entradas del POT:
    `size_type size() const;`
  - método que devuelve el elemento más pequeño del receptor (que debe tener al menos un elemento):
    `T min() const;`
  - método que inserta el elemento **`elem`** en el receptor:
    `void insert (const T& elem);`
  - método que elimina elemento más pequeño
    `void erase ();`

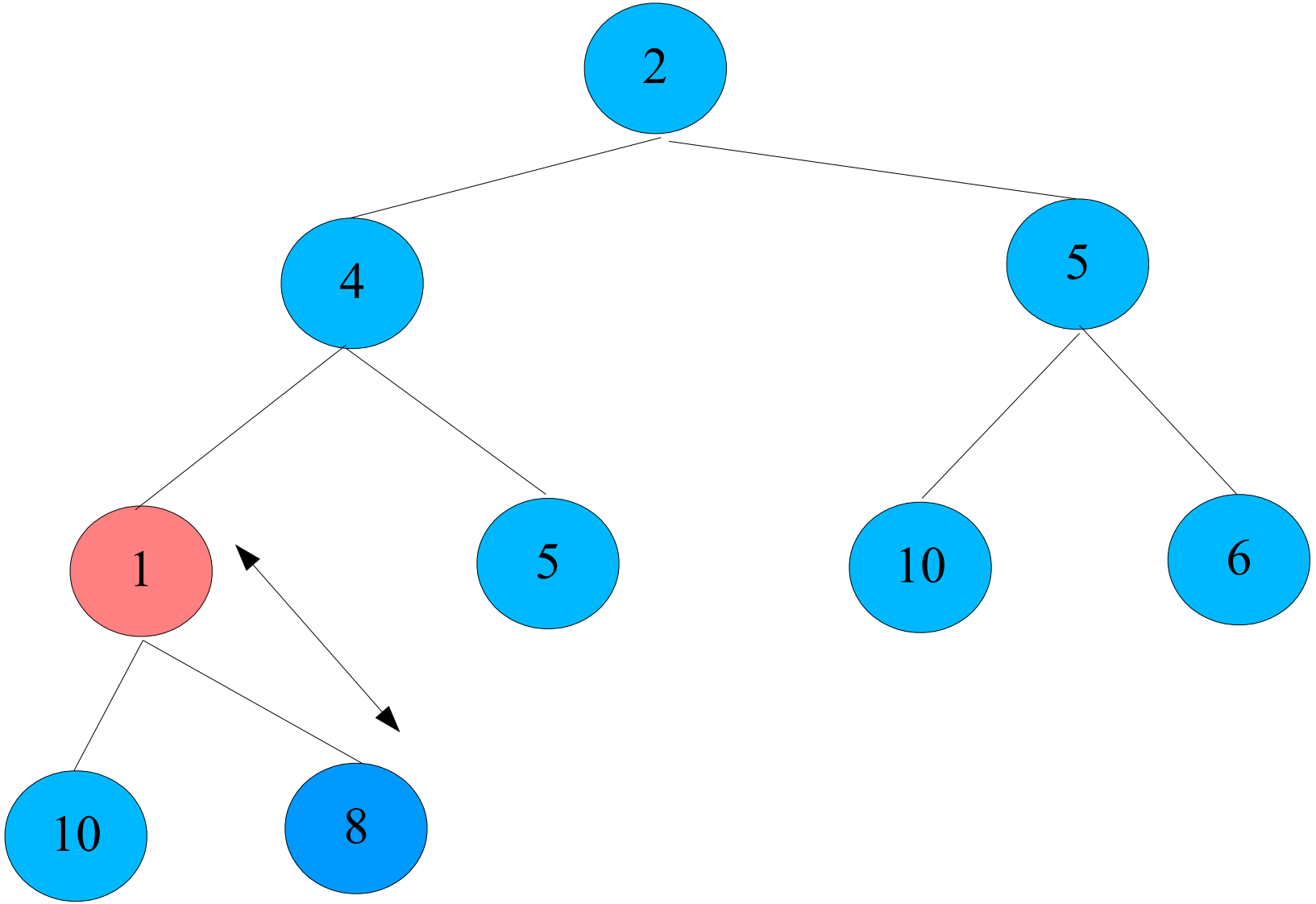


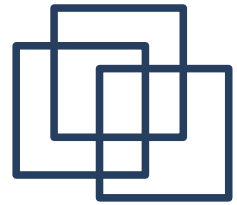
Inserción en POT(I)



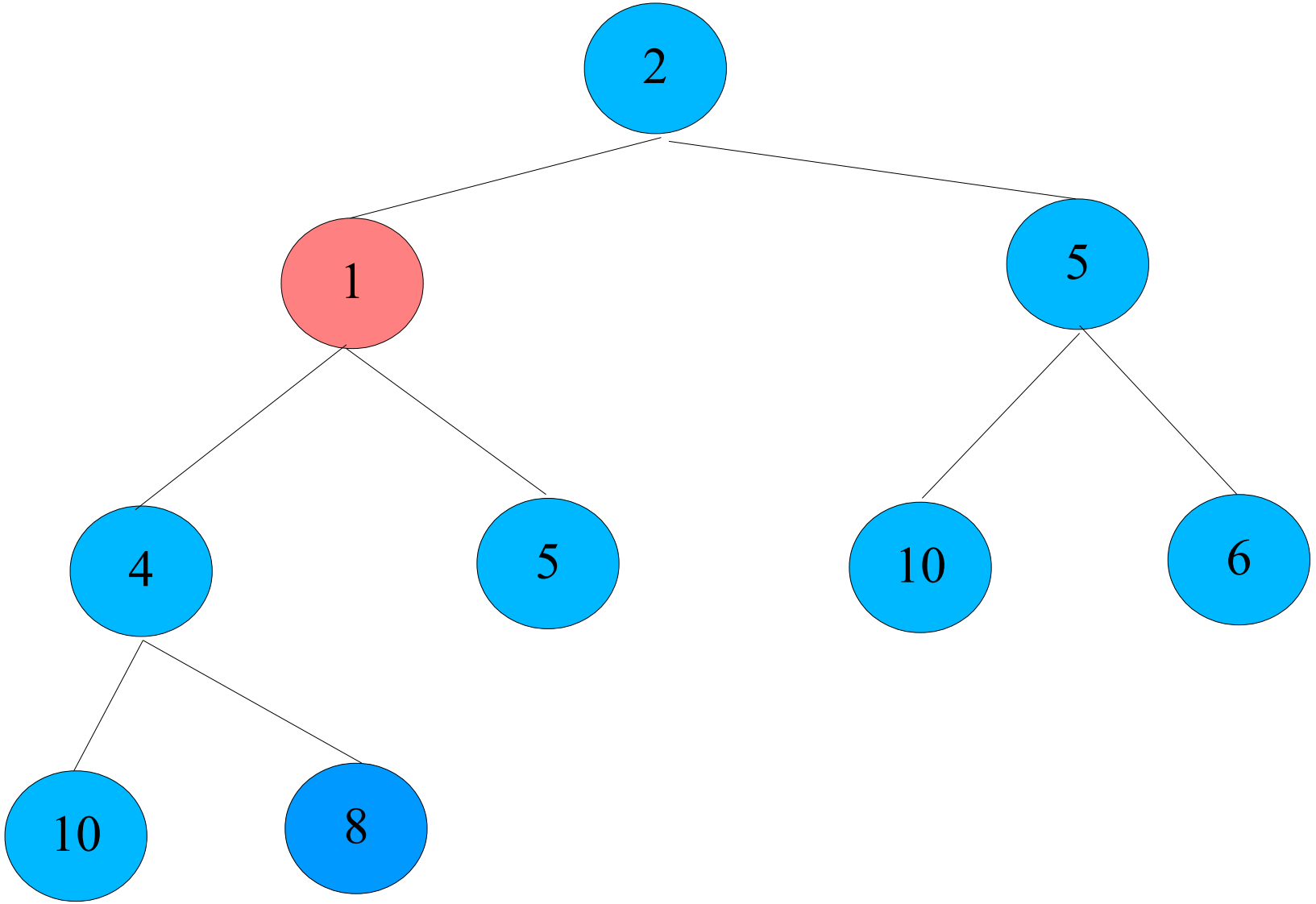


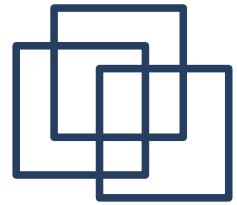
Inserción en POT (II)



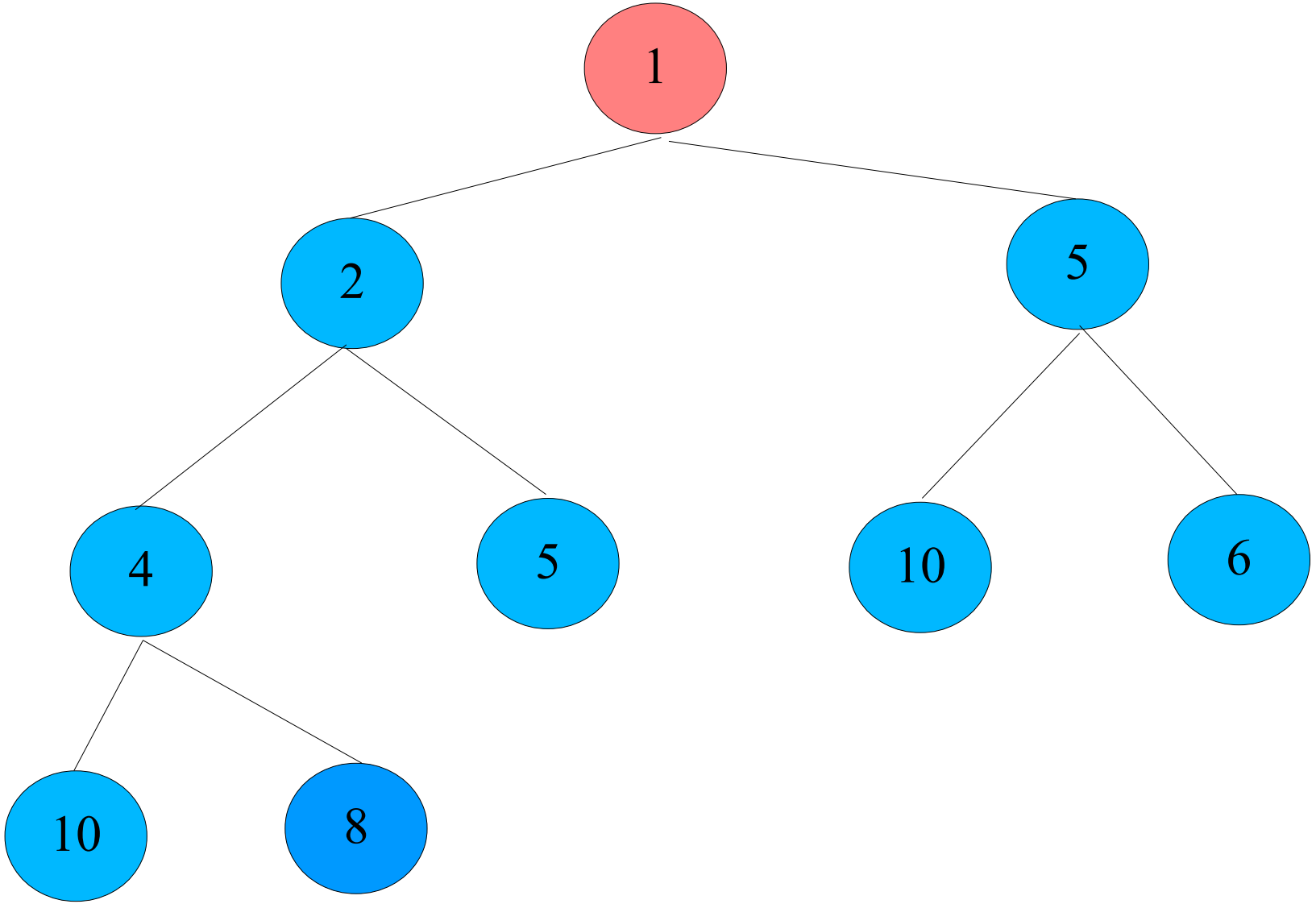


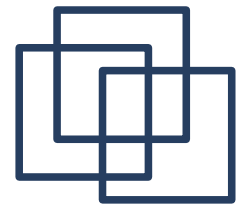
Inserción en POT (III)



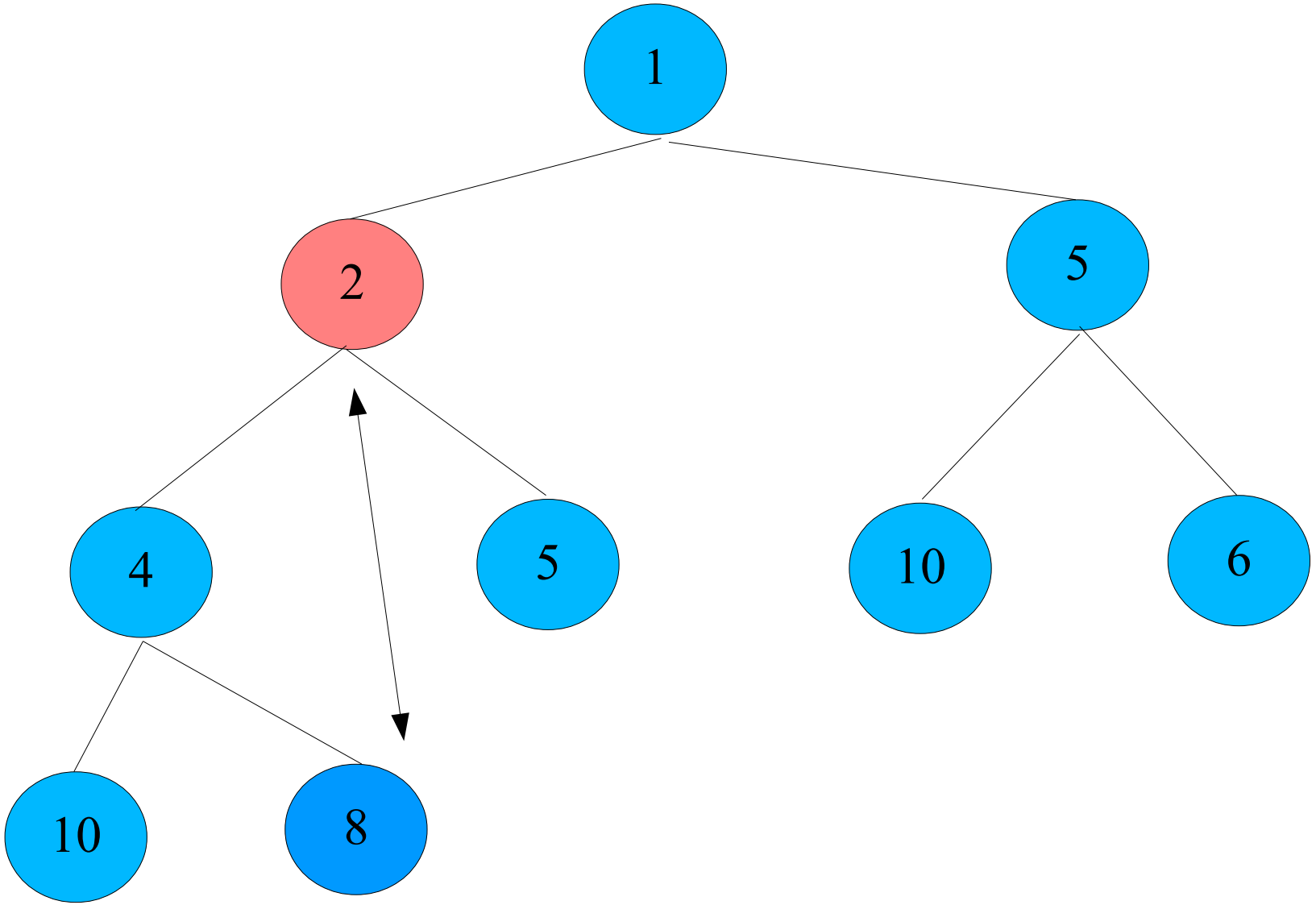


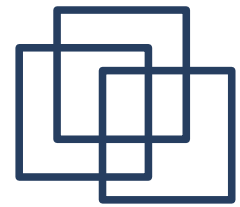
Inserción en POT (IV)



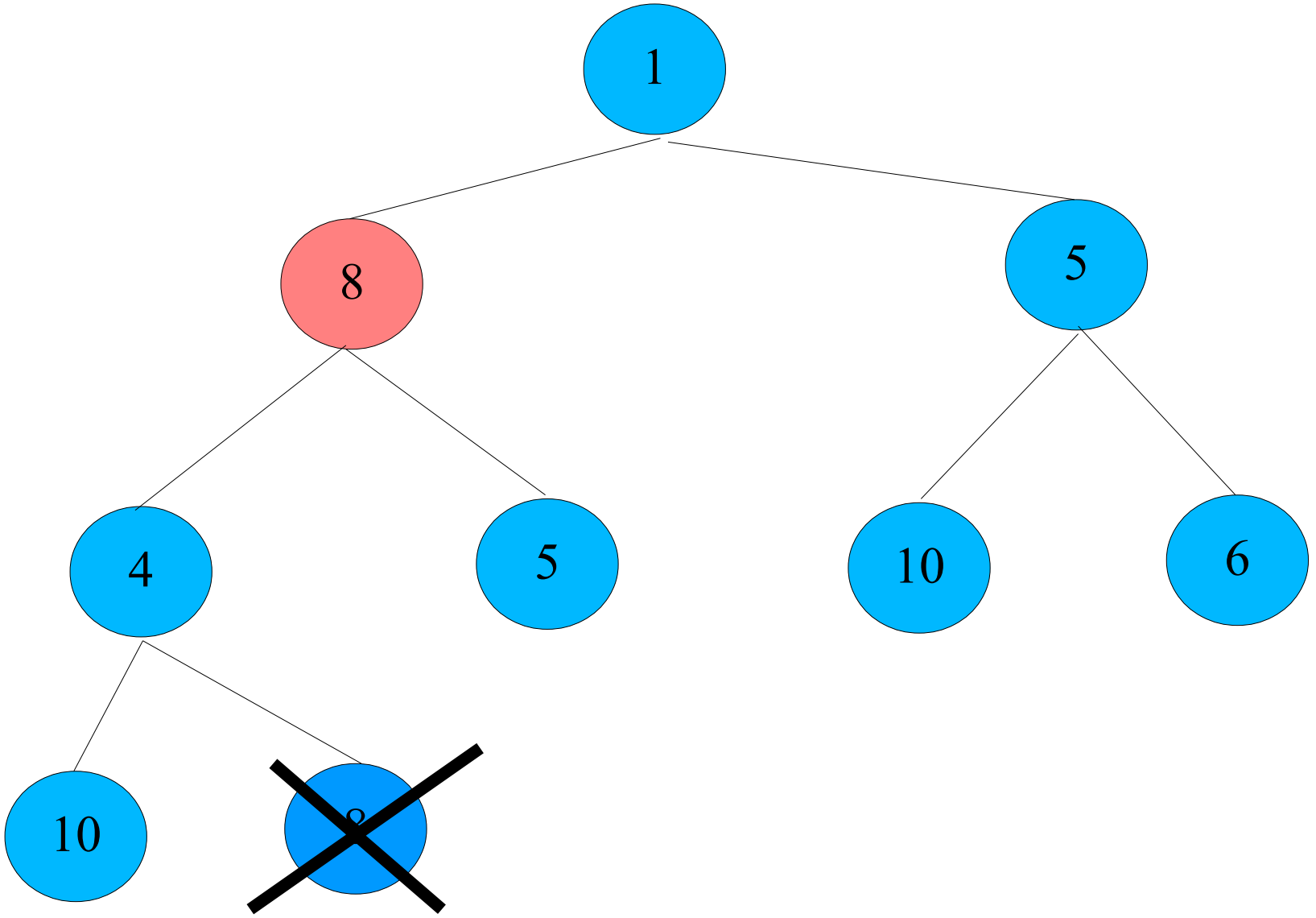


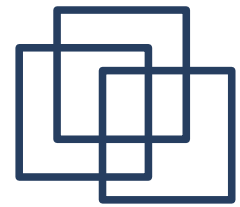
Borrado en POT (I)





Borrado en POT (II)





Borrado en POT (III)